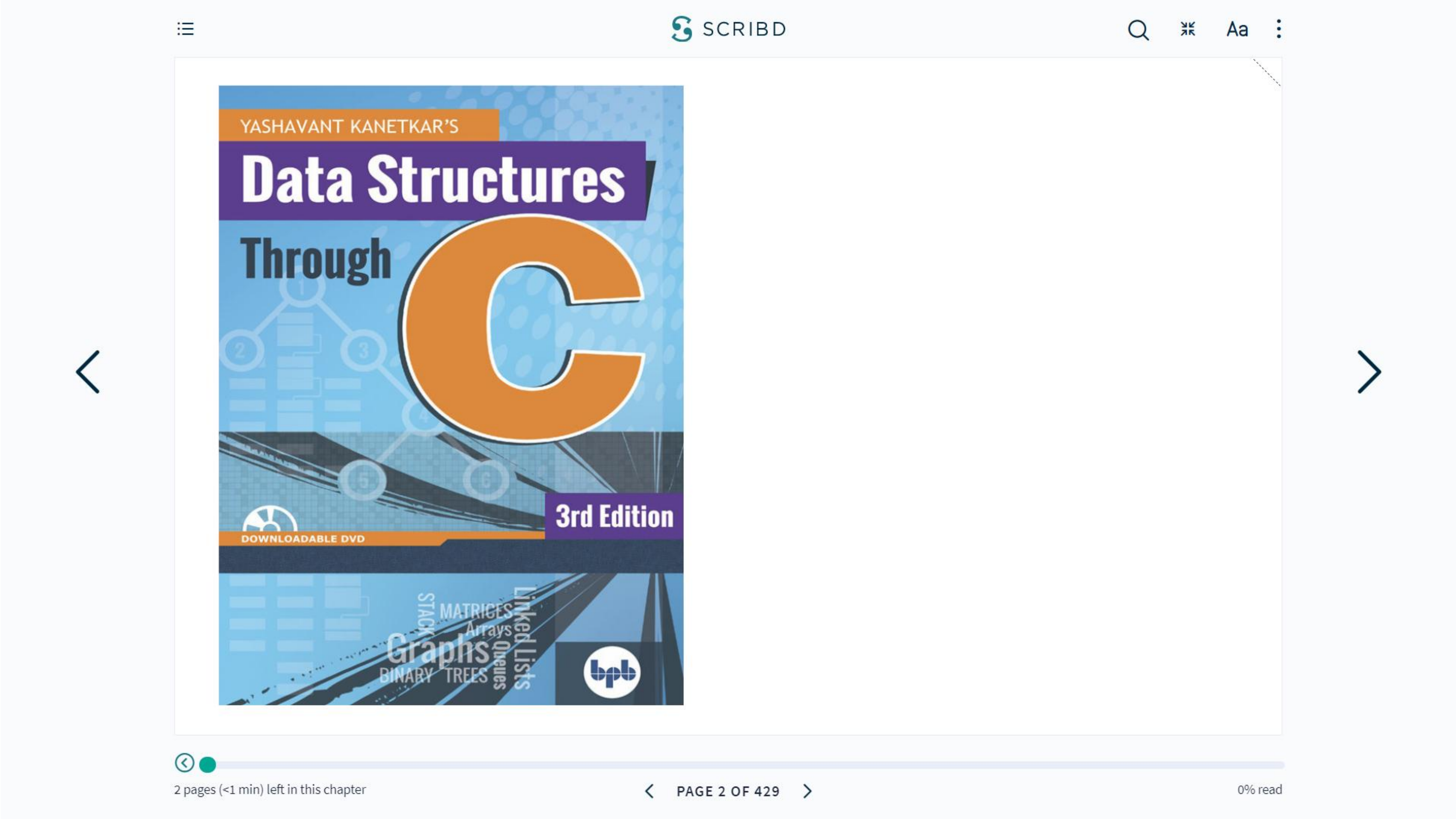
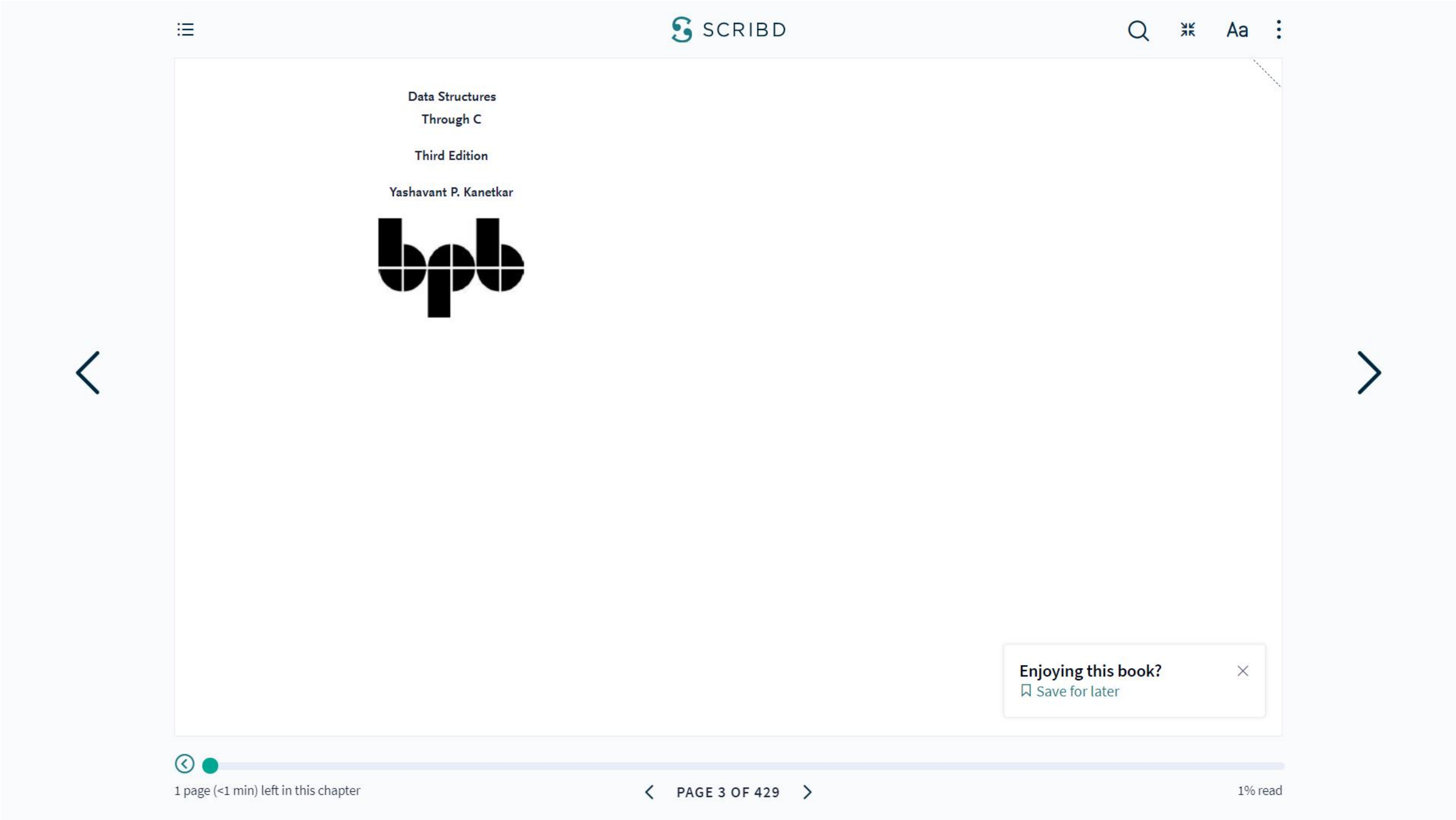


2 pages (<1 min) left in this chapter

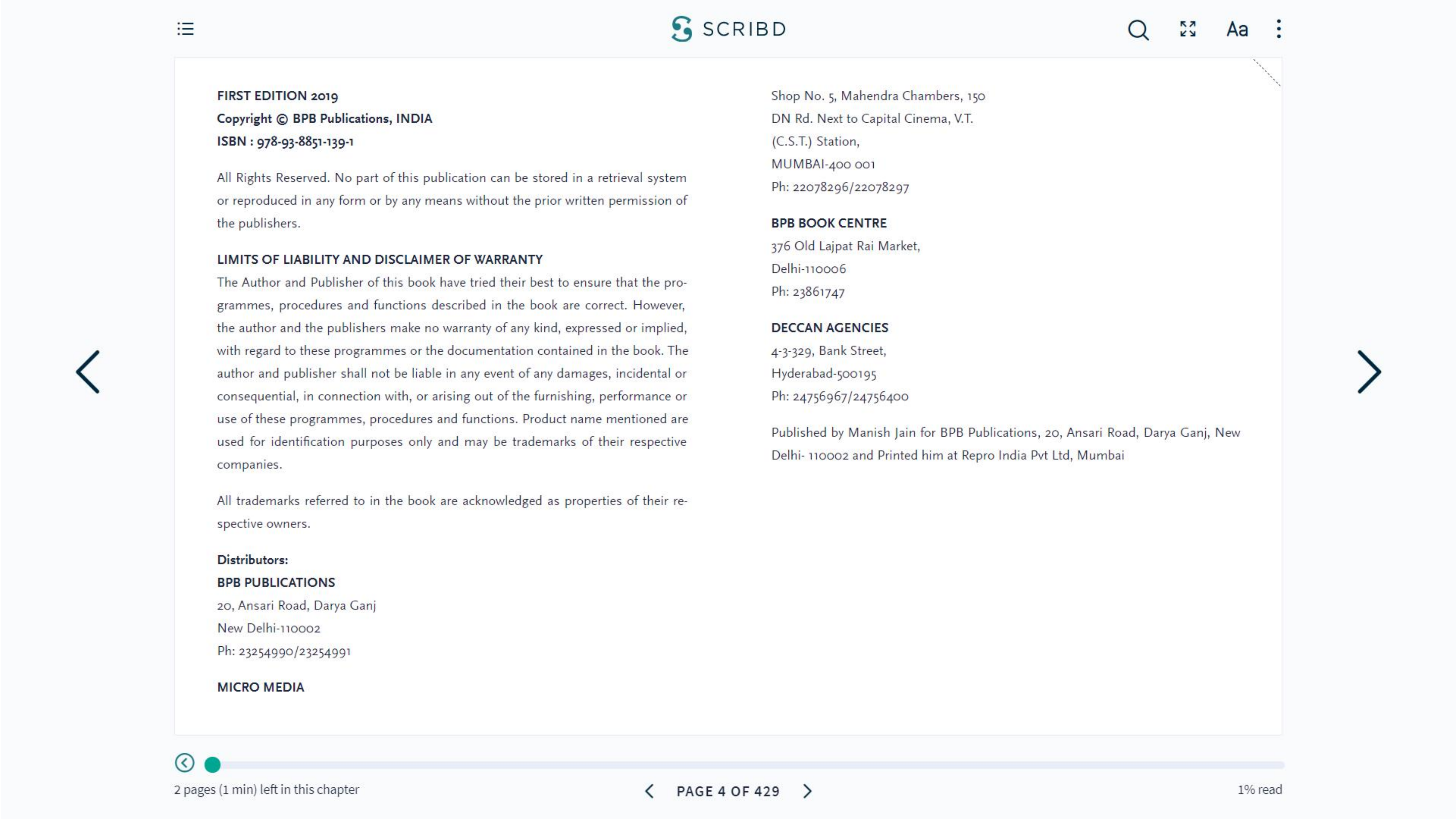
0% read



Data Structures
Through C

Third Edition

Yashavant P. Kanetkar



FIRST EDITION 2019

ISBN : 978-93-8851-139-1

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The Author and Publisher of this book have tried their best to ensure that the programmes, procedures and functions described in the book are correct. However, the author and the publishers make no warranty of any kind, expressed or implied, with regard to these programmes or the documentation contained in the book. The author and publisher shall not be liable in any event of any damages, incidental or consequential, in connection with, or arising out of the furnishing, performance or use of these programmes, procedures and functions. Product name mentioned are used for identification purposes only and may be trademarks of their respective companies.

All trademarks referred to in the book are acknowledged as properties of their respective owners.

Distributors:

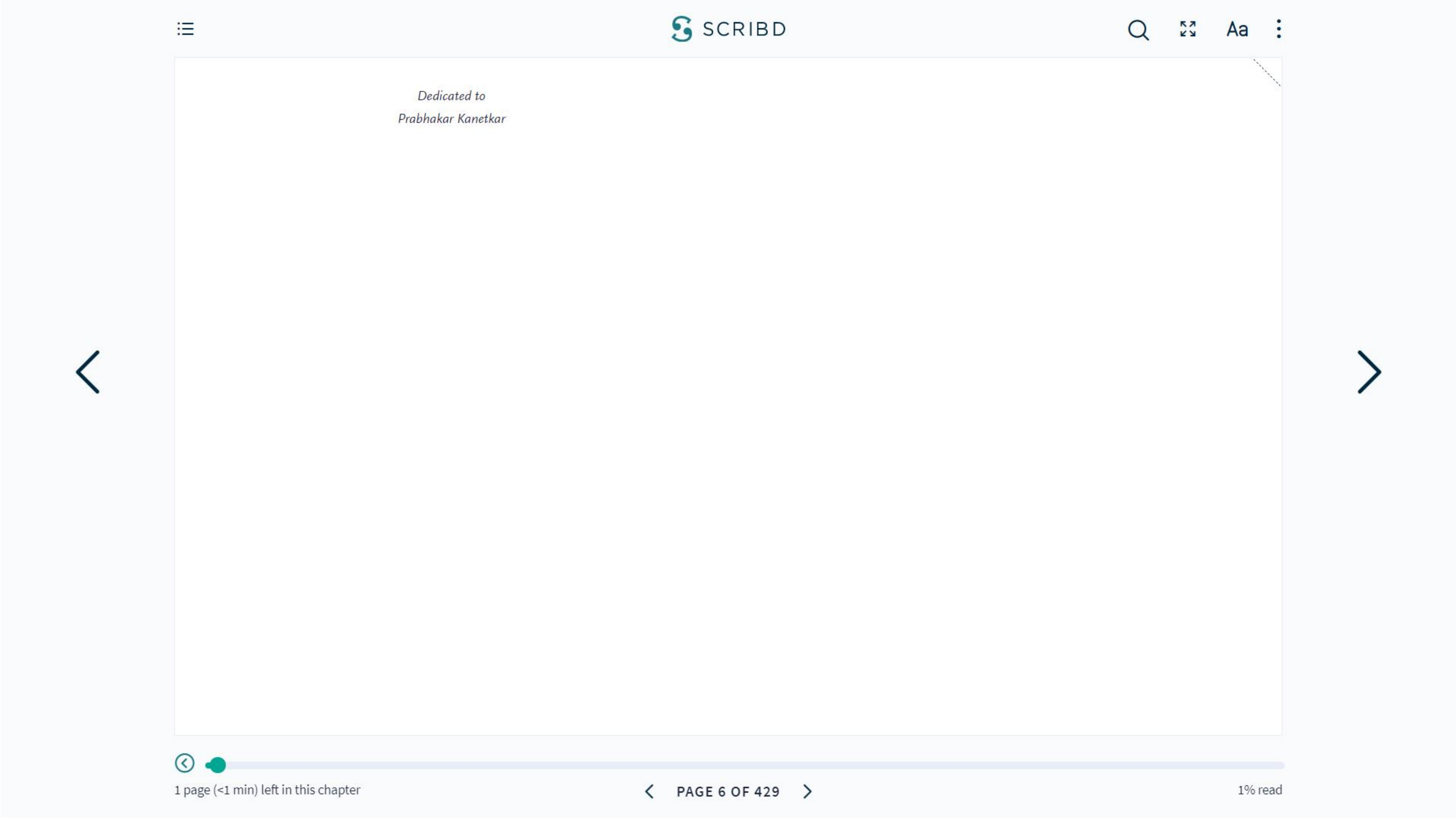
BPB PUBLICATIONS
20, Ansari Road, Darya Ganj
New Delhi-110002
Ph: 23254990/23254991

MICRO MEDIA
Shop No. 5, Mahendra Chambers, 150
DN Rd. Next to Capital Cinema, V.T.
(C.S.T.) Station,
MUMBAI-400 001
Ph: 22078296/22078297

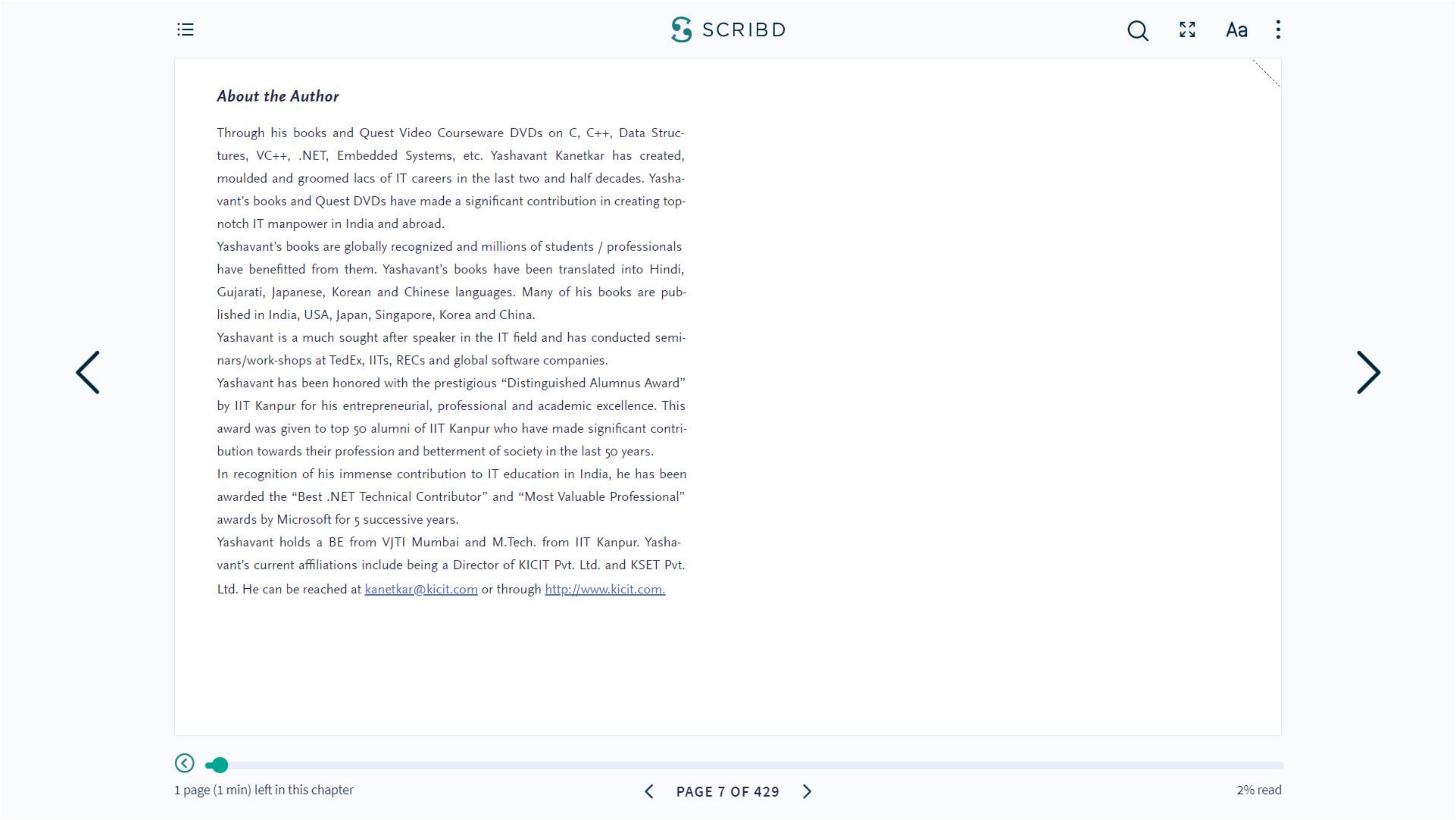
BPB BOOK CENTRE
376 Old Lajpat Rai Market,
Delhi-110006
Ph: 23861747

DECCAN AGENCIES
4-3-329, Bank Street,
Hyderabad-500195
Ph: 24756967/24756400

Published by Manish Jain for BPB Publications, 20, Ansari Road, Darya Ganj, New Delhi- 110002 and Printed him at Repro India Pvt Ltd, Mumbai



Dedicated to
Prabhakar Kanetkar



About the Author

Through his books and Quest Video Courseware DVDs on C, C++, Data Structures, VC++, .NET, Embedded Systems, etc. Yashavant Kanetkar has created, moulded and groomed lacs of IT careers in the last two and half decades. Yashavant's books and Quest DVDs have made a significant contribution in creating top-notch IT manpower in India and abroad.

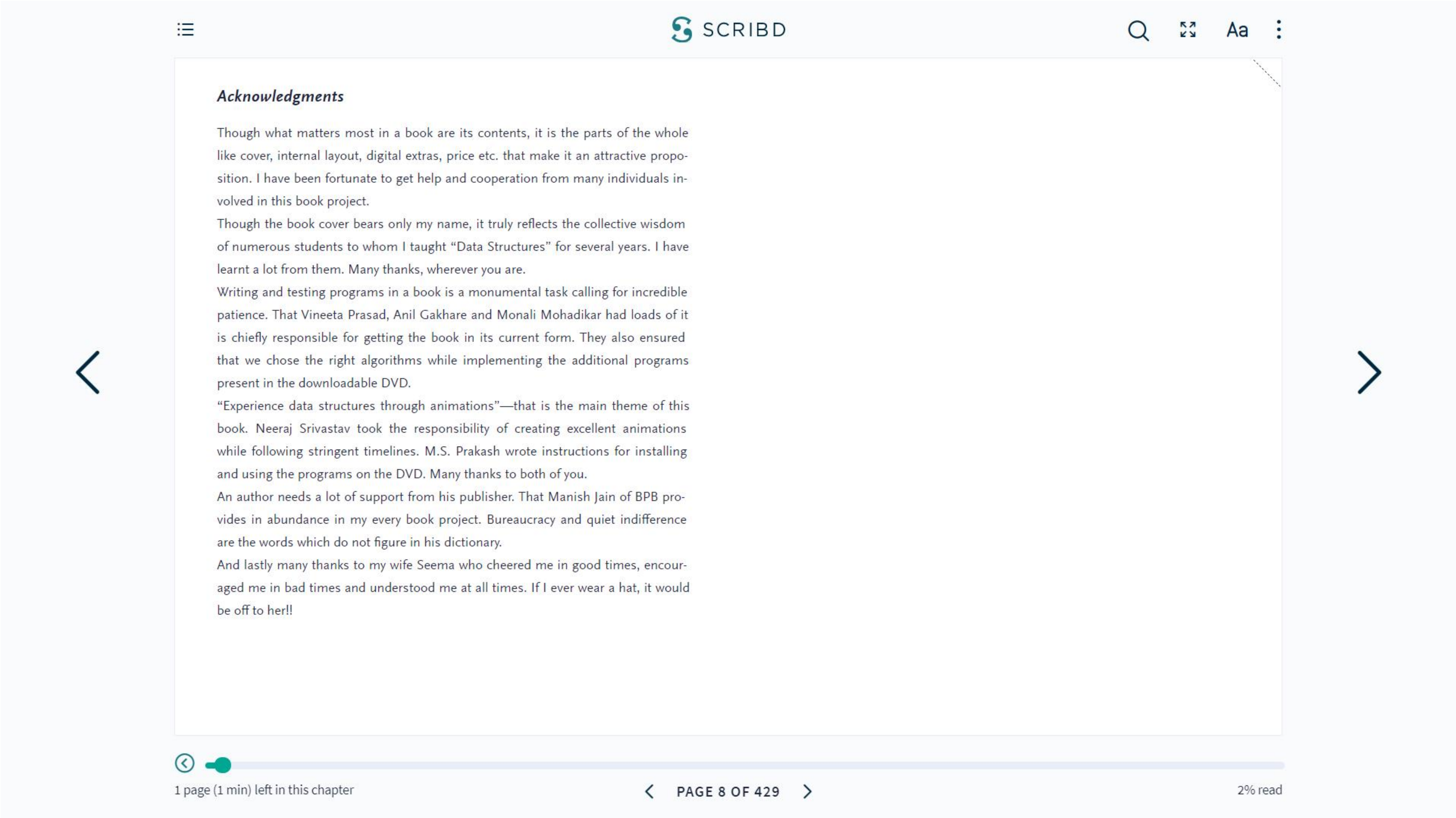
Yashavant's books are globally recognized and millions of students / professionals have benefitted from them. Yashavant's books have been translated into Hindi, Gujarati, Japanese, Korean and Chinese languages. Many of his books are published in India, USA, Japan, Singapore, Korea and China.

Yashavant is a much sought after speaker in the IT field and has conducted seminars/work-shops at TedEx, IITs, RECs and global software companies.

Yashavant has been honored with the prestigious "Distinguished Alumnus Award" by IIT Kanpur for his entrepreneurial, professional and academic excellence. This award was given to top 50 alumni of IIT Kanpur who have made significant contribution towards their profession and betterment of society in the last 50 years.

In recognition of his immense contribution to IT education in India, he has been awarded the "Best .NET Technical Contributor" and "Most Valuable Professional" awards by Microsoft for 5 successive years.

Yashavant holds a BE from VJTI Mumbai and M.Tech. from IIT Kanpur. Yashavant's current affiliations include being a Director of KICIT Pvt. Ltd. and KSET Pvt. Ltd. He can be reached at kanetkar@kicit.com or through http://www.kicit.com.



Acknowledgments

Though what matters most in a book are its contents, it is the parts of the whole like cover, internal layout, digital extras, price etc. that make it an attractive proposition. I have been fortunate to get help and cooperation from many individuals involved in this book project.

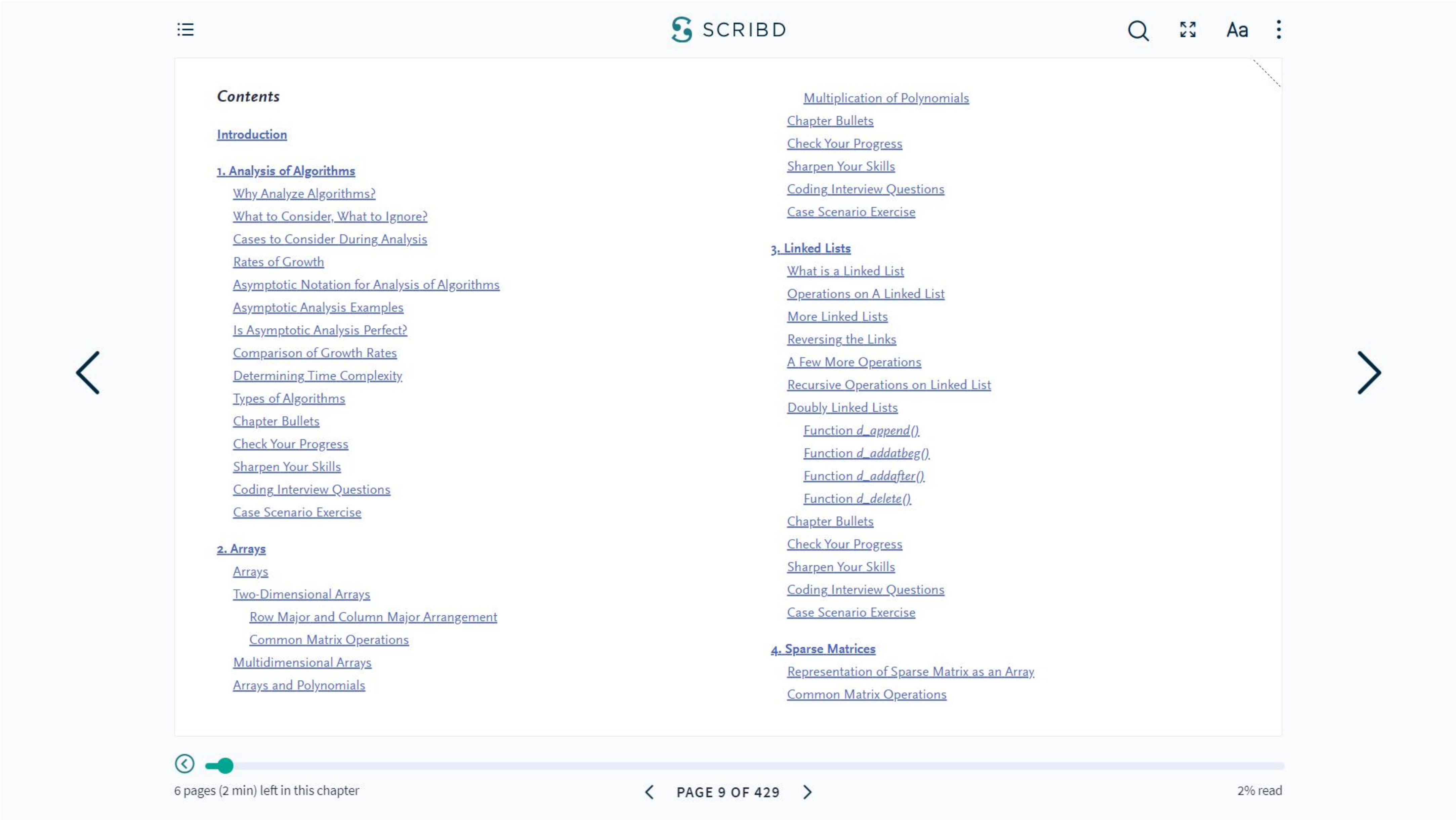
Though the book cover bears only my name, it truly reflects the collective wisdom of numerous students to whom I taught "Data Structures" for several years. I have learnt a lot from them. Many thanks, wherever you are.

Writing and testing programs in a book is a monumental task calling for incredible patience. That Vineeta Prasad, Anil Gakhare and Monali Mohadikar had loads of it is chiefly responsible for getting the book in its current form. They also ensured that we chose the right algorithms while implementing the additional programs present in the downloadable DVD.

"Experience data structures through animations"—that is the main theme of this book. Neeraj Srivastav took the responsibility of creating excellent animations while following stringent timelines. M.S. Prakash wrote instructions for installing and using the programs on the DVD. Many thanks to both of you.

An author needs a lot of support from his publisher. That Manish Jain of BPB provides in abundance in my every book project. Bureaucracy and quiet indifference are the words which do not figure in his dictionary.

And lastly many thanks to my wife Seema who cheered me in good times, encouraged me in bad times and understood me at all times. If I ever wear a hat, it would be off to her!!



Contents

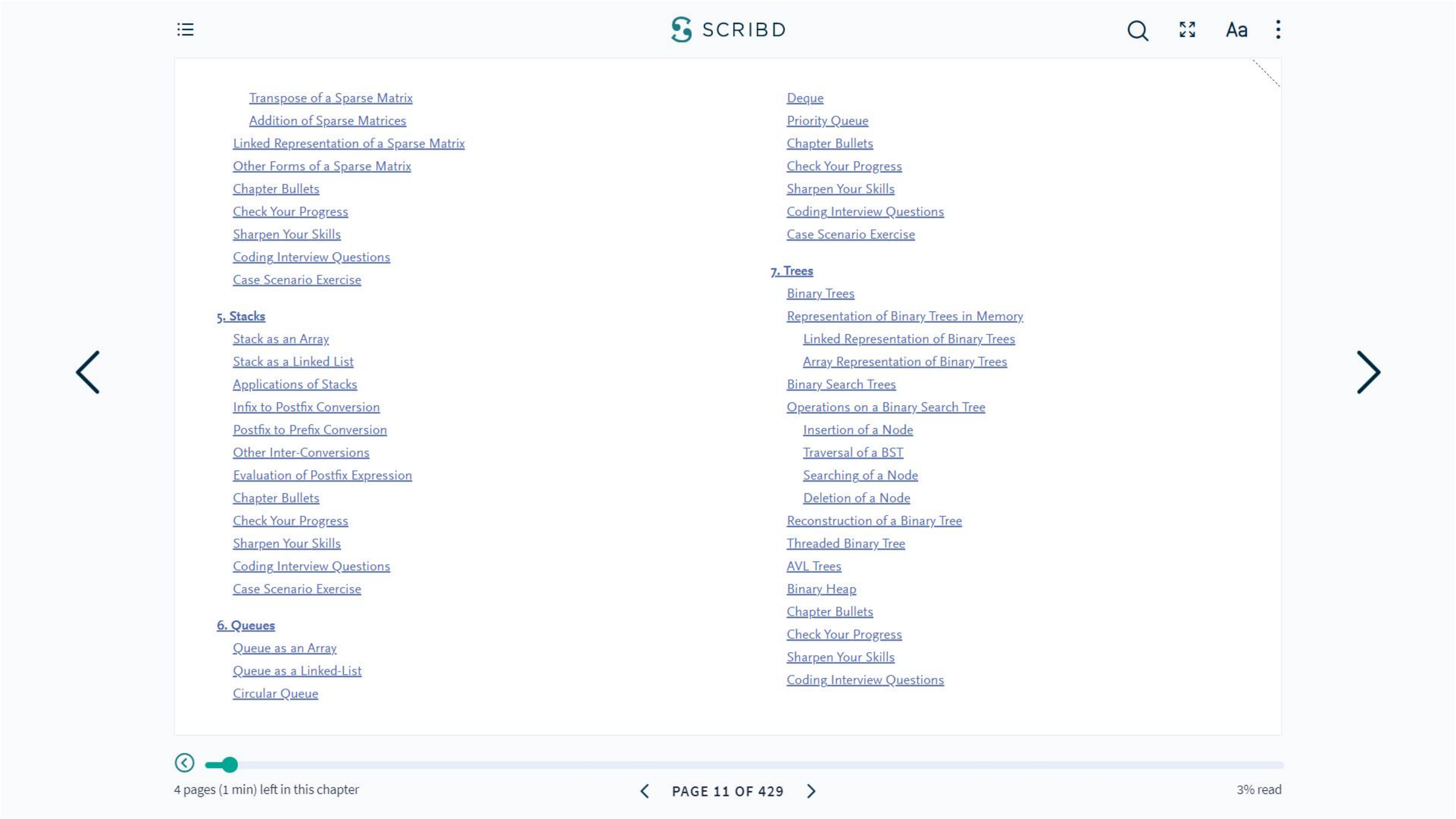
Introduction

1. Analysis of Algorithms
- Why Analyze Algorithms?
- What to Consider, What to Ignore?
- Cases to Consider During Analysis
- Rates of Growth
- Asymptotic Notation for Analysis of Algorithms
- Asymptotic Analysis Examples
- Is Asymptotic Analysis Perfect?
- Comparison of Growth Rates
- Determining Time Complexity
- Types of Algorithms
- Chapter Bullets
- Check Your Progress
- Sharpen Your Skills
- Coding Interview Questions
- Case Scenario Exercise

2. Arrays
- Arrays
- Two-Dimensional Arrays
  - Row Major and Column Major Arrangement
  - Common Matrix Operations
- Multidimensional Arrays
- Arrays and Polynomials
  - Multiplication of Polynomials
- Chapter Bullets
- Check Your Progress
- Sharpen Your Skills
- Coding Interview Questions
- Case Scenario Exercise

3. Linked Lists
- What is a Linked List
- Operations on A Linked List
- More Linked Lists
- Reversing the Links
- A Few More Operations
- Recursive Operations on Linked List
- Doubly Linked Lists
  - Function *d_append()*
  - Function *d_addatbeg()*
  - Function *d_addafter()*
  - Function *d_delete()*
- Chapter Bullets
- Check Your Progress
- Sharpen Your Skills
- Coding Interview Questions
- Case Scenario Exercise

4. Sparse Matrices
- Representation of Sparse Matrix as an Array
- Common Matrix Operations



- Transpose of a Sparse Matrix
- Addition of Sparse Matrices
- Linked Representation of a Sparse Matrix
- Other Forms of a Sparse Matrix
- Chapter Bullets
- Check Your Progress
- Sharpen Your Skills
- Coding Interview Questions
- Case Scenario Exercise

5. Stacks

- Stack as an Array
- Stack as a Linked List
- Applications of Stacks
- Infix to Postfix Conversion
- Postfix to Prefix Conversion
- Other Inter-Conversions
- Evaluation of Postfix Expression
- Chapter Bullets
- Check Your Progress
- Sharpen Your Skills
- Coding Interview Questions
- Case Scenario Exercise

6. Queues

- Queue as an Array
- Queue as a Linked-List
- Circular Queue
- Deque
- Priority Queue
- Chapter Bullets
- Check Your Progress
- Sharpen Your Skills
- Coding Interview Questions
- Case Scenario Exercise

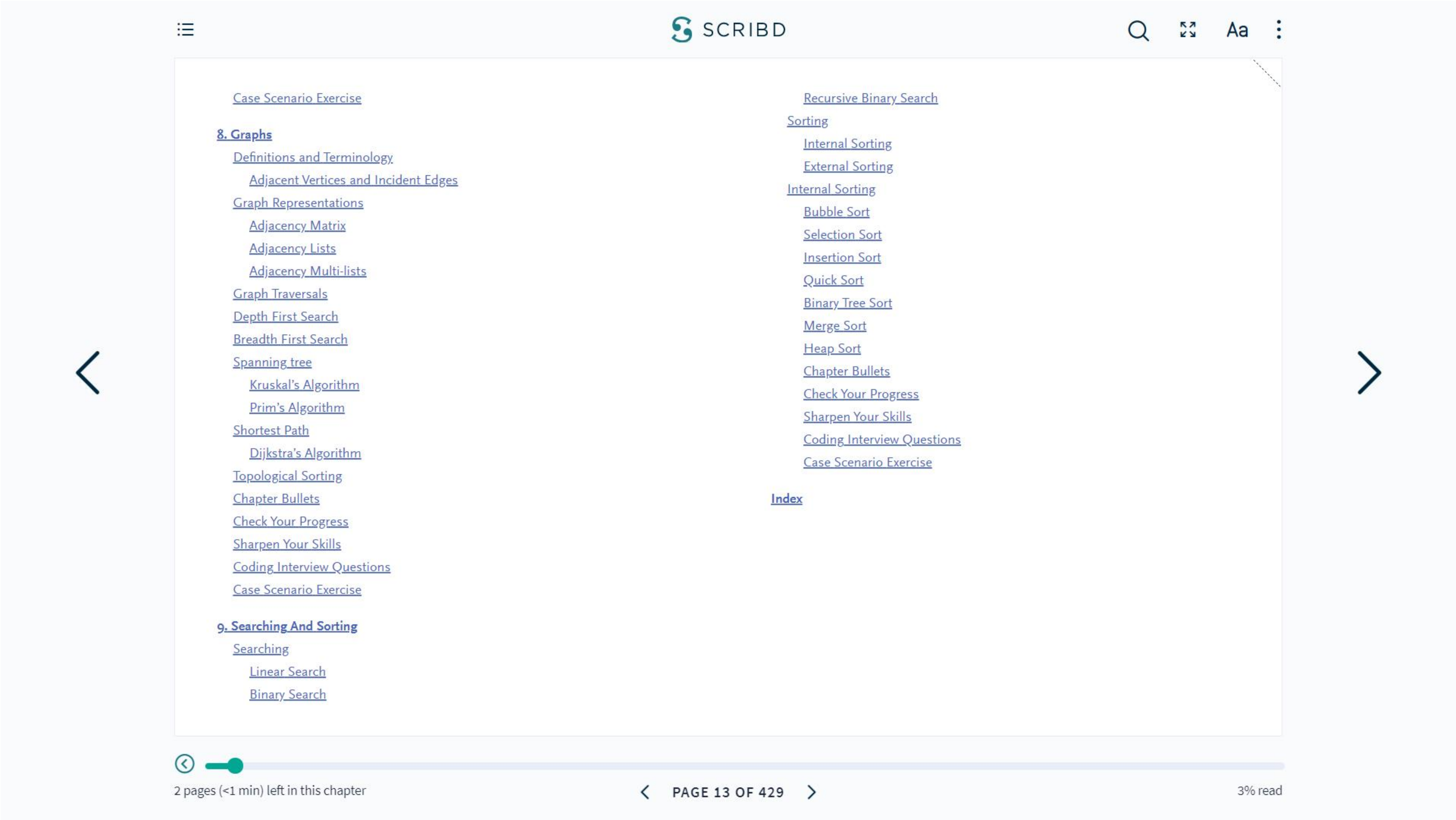
7. Trees

- Binary Trees
- Representation of Binary Trees in Memory
  - Linked Representation of Binary Trees
  - Array Representation of Binary Trees
- Binary Search Trees
- Operations on a Binary Search Tree
  - Insertion of a Node
  - Traversal of a BST
  - Searching of a Node
  - Deletion of a Node
- Reconstruction of a Binary Tree
- Threaded Binary Tree
- AVL Trees
- Binary Heap
- Chapter Bullets
- Check Your Progress
- Sharpen Your Skills
- Coding Interview Questions

4 pages (1 min) left in this chapter

PAGE 11 OF 429

3% read

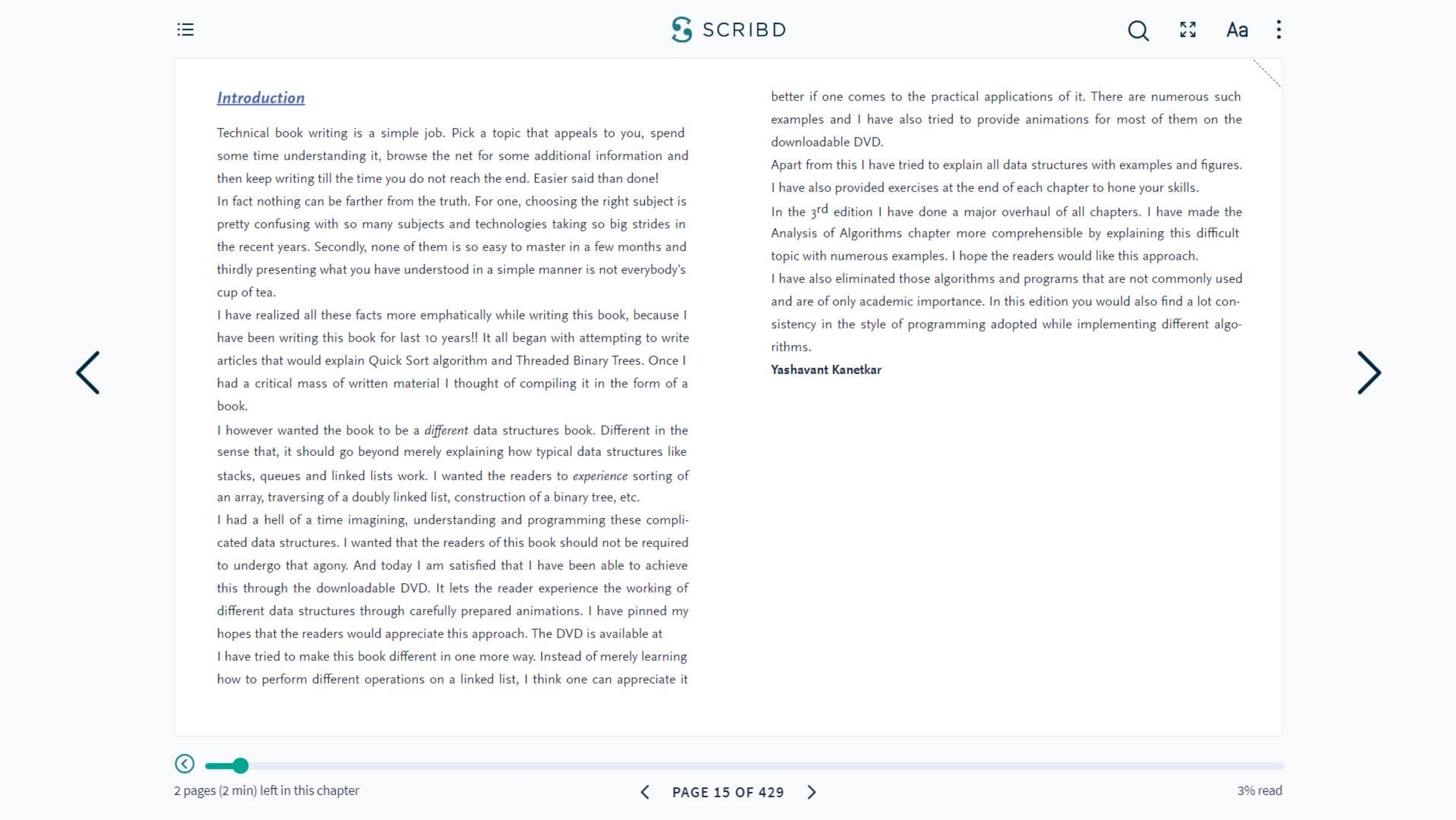


- Case Scenario Exercise

8. Graphs
- Definitions and Terminology
  - Adjacent Vertices and Incident Edges
- Graph Representations
  - Adjacency Matrix
  - Adjacency Lists
  - Adjacency Multi-lists
- Graph Traversals
- Depth First Search
- Breadth First Search
- Spanning tree
  - Kruskal's Algorithm
  - Prim's Algorithm
- Shortest Path
  - Dijkstra's Algorithm
- Topological Sorting
- Chapter Bullets
- Check Your Progress
- Sharpen Your Skills
- Coding Interview Questions
- Case Scenario Exercise

9. Searching And Sorting
- Searching
  - Linear Search
  - Binary Search
  - Recursive Binary Search
- Sorting
  - Internal Sorting
  - External Sorting
- Internal Sorting
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Quick Sort
  - Binary Tree Sort
  - Merge Sort
  - Heap Sort
  - Chapter Bullets
  - Check Your Progress
  - Sharpen Your Skills
  - Coding Interview Questions
  - Case Scenario Exercise

Index



Introduction

Technical book writing is a simple job. Pick a topic that appeals to you, spend some time understanding it, browse the net for some additional information and then keep writing till the time you do not reach the end. Easier said than done!

In fact nothing can be farther from the truth. For one, choosing the right subject is pretty confusing with so many subjects and technologies taking so big strides in the recent years. Secondly, none of them is so easy to master in a few months and thirdly presenting what you have understood in a simple manner is not everybody's cup of tea.

I have realized all these facts more emphatically while writing this book, because I have been writing this book for last 10 years!! It all began with attempting to write articles that would explain Quick Sort algorithm and Threaded Binary Trees. Once I had a critical mass of written material I thought of compiling it in the form of a book.

I however wanted the book to be a *different* data structures book. Different in the sense that, it should go beyond merely explaining how typical data structures like stacks, queues and linked lists work. I wanted the readers to *experience* sorting of an array, traversing of a doubly linked list, construction of a binary tree, etc.

I had a hell of a time imagining, understanding and programming these complicated data structures. I wanted that the readers of this book should not be required to undergo that agony. And today I am satisfied that I have been able to achieve this through the downloadable DVD. It lets the reader experience the working of different data structures through carefully prepared animations. I have pinned my hopes that the readers would appreciate this approach. The DVD is available at

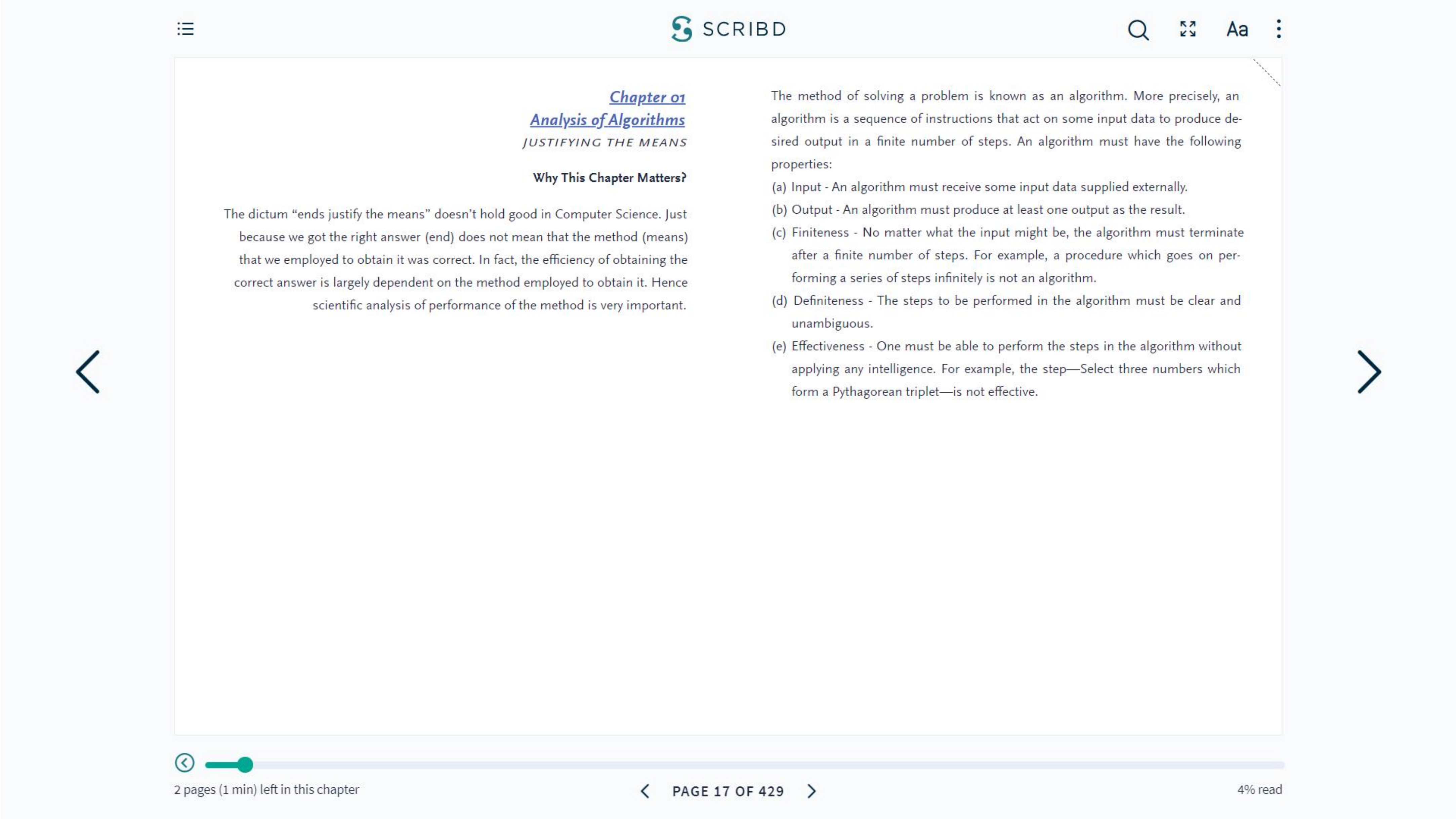
I have tried to make this book different in one more way. Instead of merely learning how to perform different operations on a linked list, I think one can appreciate it better if one comes to the practical applications of it. There are numerous such examples and I have also tried to provide animations for most of them on the downloadable DVD.

Apart from this I have tried to explain all data structures with examples and figures. I have also provided exercises at the end of each chapter to hone your skills.

In the 3rd edition I have done a major overhaul of all chapters. I have made the Analysis of Algorithms chapter more comprehensible by explaining this difficult topic with numerous examples. I hope the readers would like this approach.

I have also eliminated those algorithms and programs that are not commonly used and are of only academic importance. In this edition you would also find a lot consistency in the style of programming adopted while implementing different algorithms.

Yashavant Kanetkar



Chapter 01
Analysis of Algorithms

JUSTIFYING THE MEANS

Why This Chapter Matters?

The dictum "ends justify the means" doesn't hold good in Computer Science. Just because we got the right answer (end) does not mean that the method (means) that we employed to obtain it was correct. In fact, the efficiency of obtaining the correct answer is largely dependent on the method employed to obtain it. Hence scientific analysis of performance of the method is very important.

The method of solving a problem is known as an algorithm. More precisely, an algorithm is a sequence of instructions that act on some input data to produce desired output in a finite number of steps. An algorithm must have the following properties:

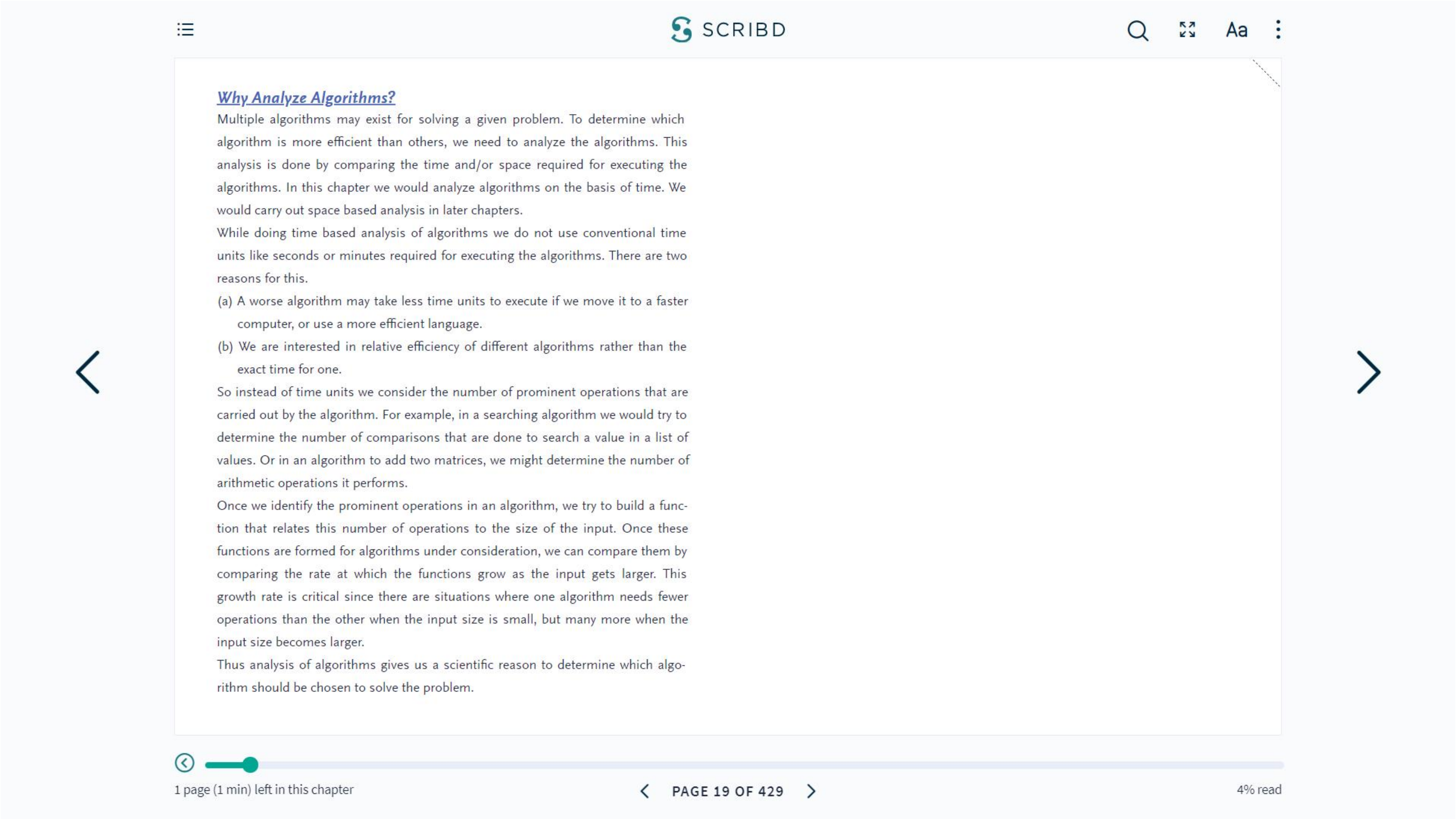
(a) Input - An algorithm must receive some input data supplied externally.

(b) Output - An algorithm must produce at least one output as the result.

(c) Finiteness - No matter what the input might be, the algorithm must terminate after a finite number of steps. For example, a procedure which goes on performing a series of steps infinitely is not an algorithm.

(d) Definiteness - The steps to be performed in the algorithm must be clear and unambiguous.

(e) Effectiveness - One must be able to perform the steps in the algorithm without applying any intelligence. For example, the step—Select three numbers which form a Pythagorean triplet—is not effective.



Why Analyze Algorithms?

Multiple algorithms may exist for solving a given problem. To determine which algorithm is more efficient than others, we need to analyze the algorithms. This analysis is done by comparing the time and/or space required for executing the algorithms. In this chapter we would analyze algorithms on the basis of time. We would carry out space based analysis in later chapters.

While doing time based analysis of algorithms we do not use conventional time units like seconds or minutes required for executing the algorithms. There are two reasons for this.

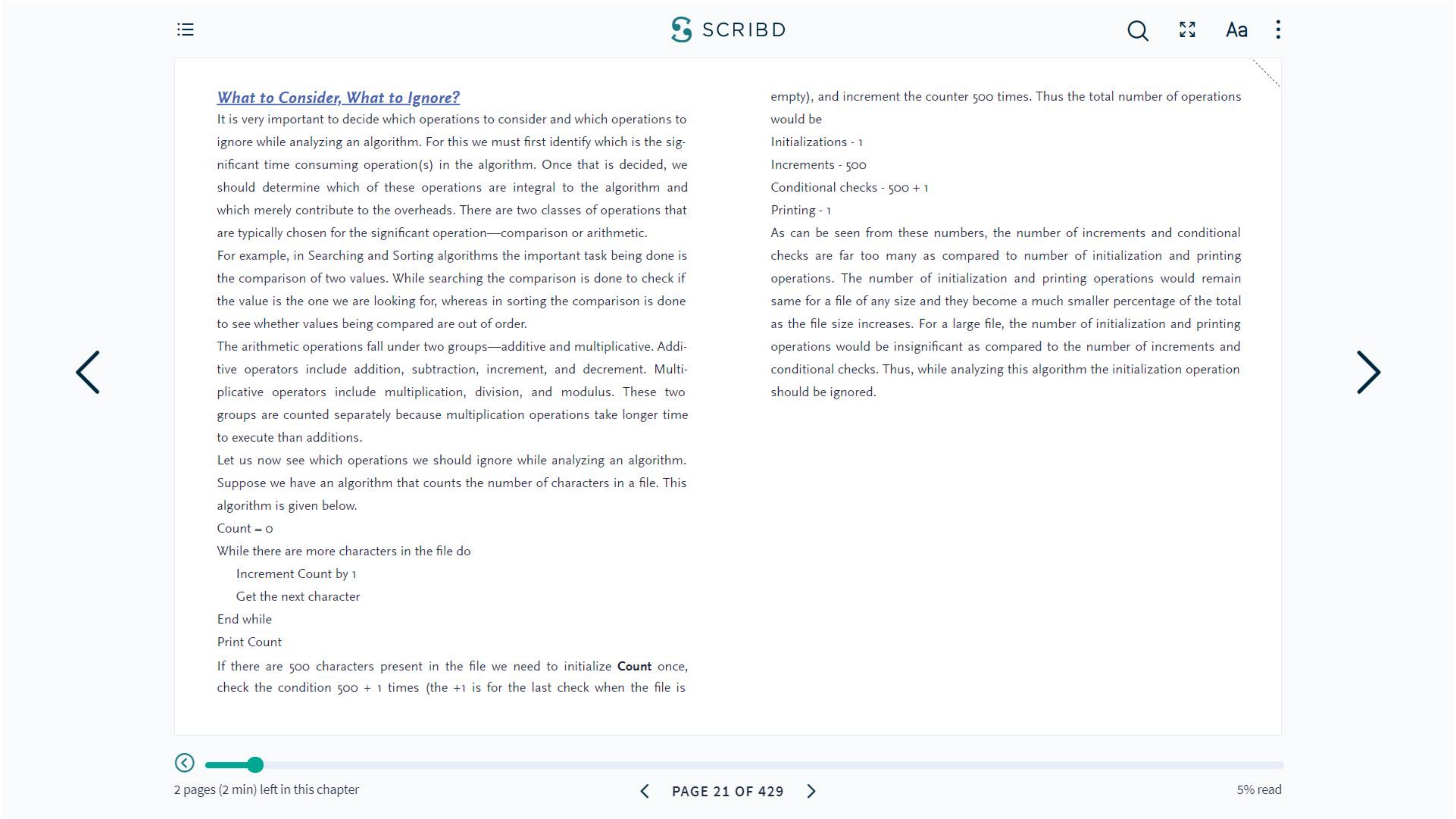
(a) A worse algorithm may take less time units to execute if we move it to a faster computer, or use a more efficient language.

(b) We are interested in relative efficiency of different algorithms rather than the exact time for one.

So instead of time units we consider the number of prominent operations that are carried out by the algorithm. For example, in a searching algorithm we would try to determine the number of comparisons that are done to search a value in a list of values. Or in an algorithm to add two matrices, we might determine the number of arithmetic operations it performs.

Once we identify the prominent operations in an algorithm, we try to build a function that relates this number of operations to the size of the input. Once these functions are formed for algorithms under consideration, we can compare them by comparing the rate at which the functions grow as the input gets larger. This growth rate is critical since there are situations where one algorithm needs fewer operations than the other when the input size is small, but many more when the input size becomes larger.

Thus analysis of algorithms gives us a scientific reason to determine which algorithm should be chosen to solve the problem.



What to Consider, What to Ignore?

It is very important to decide which operations to consider and which operations to ignore while analyzing an algorithm. For this we must first identify which is the significant time consuming operation(s) in the algorithm. Once that is decided, we should determine which of these operations are integral to the algorithm and which merely contribute to the overheads. There are two classes of operations that are typically chosen for the significant operation—comparison or arithmetic.

For example, in Searching and Sorting algorithms the important task being done is the comparison of two values. While searching the comparison is done to check if the value is the one we are looking for, whereas in sorting the comparison is done to see whether values being compared are out of order.

The arithmetic operations fall under two groups—additive and multiplicative. Additive operators include addition, subtraction, increment, and decrement. Multiplicative operators include multiplication, division, and modulus. These two groups are counted separately because multiplication operations take longer time to execute than additions.

Let us now see which operations we should ignore while analyzing an algorithm. Suppose we have an algorithm that counts the number of characters in a file. This algorithm is given below.

```
Count = 0
While there are more characters in the file do
    Increment Count by 1
    Get the next character
End while
Print Count
```

If there are 500 characters present in the file we need to initialize **Count** once, check the condition 500 + 1 times (the +1 is for the last check when the file is empty), and increment the counter 500 times. Thus the total number of operations would be

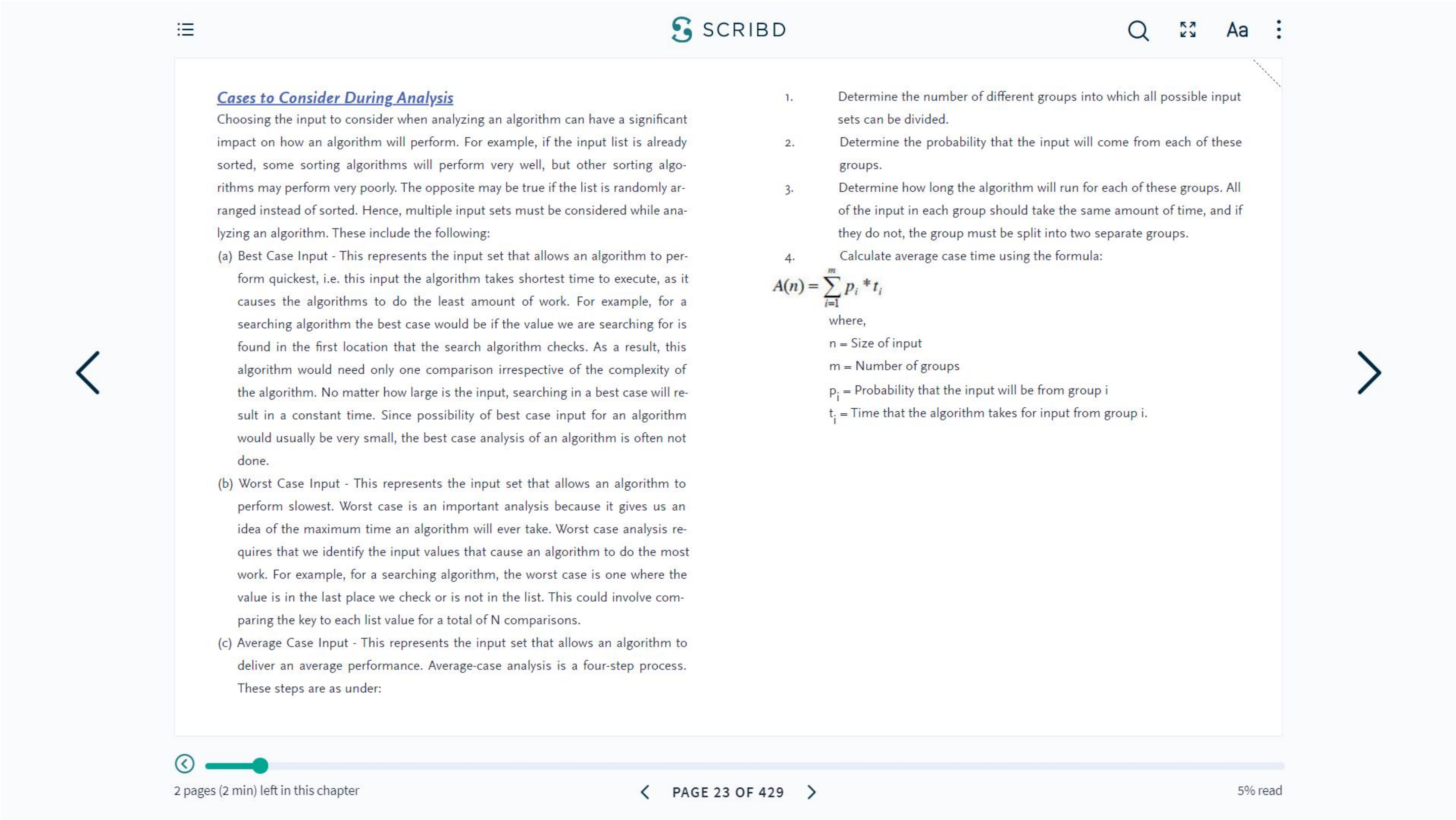
Initializations - 1

Increments - 500

Conditional checks - 500 + 1

Printing - 1

As can be seen from these numbers, the number of increments and conditional checks are far too many as compared to number of initialization and printing operations. The number of initialization and printing operations would remain same for a file of any size and they become a much smaller percentage of the total as the file size increases. For a large file, the number of initialization and printing operations would be insignificant as compared to the number of increments and conditional checks. Thus, while analyzing this algorithm the initialization operation should be ignored.



Cases to Consider During Analysis

Choosing the input to consider when analyzing an algorithm can have a significant impact on how an algorithm will perform. For example, if the input list is already sorted, some sorting algorithms will perform very well, but other sorting algorithms may perform very poorly. The opposite may be true if the list is randomly arranged instead of sorted. Hence, multiple input sets must be considered while analyzing an algorithm. These include the following:

(a) Best Case Input - This represents the input set that allows an algorithm to perform quickest, i.e. this input the algorithm takes shortest time to execute, as it causes the algorithms to do the least amount of work. For example, for a searching algorithm the best case would be if the value we are searching for is found in the first location that the search algorithm checks. As a result, this algorithm would need only one comparison irrespective of the complexity of the algorithm. No matter how large is the input, searching in a best case will result in a constant time. Since possibility of best case input for an algorithm would usually be very small, the best case analysis of an algorithm is often not done.

(b) Worst Case Input - This represents the input set that allows an algorithm to perform slowest. Worst case is an important analysis because it gives us an idea of the maximum time an algorithm will ever take. Worst case analysis requires that we identify the input values that cause an algorithm to do the most work. For example, for a searching algorithm, the worst case is one where the value is in the last place we check or is not in the list. This could involve comparing the key to each list value for a total of N comparisons.

(c) Average Case Input - This represents the input set that allows an algorithm to deliver an average performance. Average-case analysis is a four-step process. These steps are as under:

1. Determine the number of different groups into which all possible input sets can be divided.
2. Determine the probability that the input will come from each of these groups.
3. Determine how long the algorithm will run for each of these groups. All of the input in each group should take the same amount of time, and if they do not, the group must be split into two separate groups.
4. Calculate average case time using the formula:

$$A(n) = \sum_{i=1}^{m} p_i * t_i$$

where,

n = Size of input

m = Number of groups

p_i = Probability that the input will be from group i

t_i = Time that the algorithm takes for input from group i.

Rates of Growth

While analyzing algorithms, more than the exact number of operations performed by the algorithm, it is the rate of increase in operations as the size of the problem increases that is of more importance. This is often called the rate of growth of the algorithm. What happens with small sets of input data is not as interesting as what happens when the data set gets large.

Table 1-1 shows rate of growth for some of the common classes of algorithms for a wide range of input sizes. You can observe that there isn't a significant difference in values when the input is small, but once the input value gets large, there are big differences. Hence, while analyzing algorithms, we must consider what happens when the size of the input is large, because small input sets can hide rather dramatic differences.

***Table** 1-1. Rate of increase in common algorithm classes.*

The data in Table 1-1 also illustrates that the faster growing functions increase at such a rate that they quickly dominate the slower-growing functions. Thus, if the algorithm's complexity is a combination of a two of these classes, we can safely ignore the slower growing terms. On discarding these terms, we are left with what we call the order of the function or related algorithm. We usually consider one algorithm to be more efficient than another if its worst case running time has a lower order of growth.

Based on their order, algorithms can be grouped into three categories:

(a) Algorithms that grow at least as fast as some function

(b) Algorithms that grow no faster

(c) Algorithms that grow at the same rate

The categories (a), (b), (c) mentioned above are commonly represented using Asymptotic Notations Big Omega **Ω (g(n))**, Big Oh **O (g(n))** and Big Theta **θ (g(n))**, respectively. These notations are discussed below in detail.

Asymptotic Notation for Analysis of Algorithms

The Big Omega category of functions are not of much interest to us since for all values of **n** greater than some threshold value n_o all the functions inΩ have values that are at least as large as **g**. That is, all functions in this category grow as fast as **g** or even faster. Using Asymptotic Notation this is represented as

f (n) >= c g (n)

where **c** is some constant > **0** and $n >= n_o >= 1$.

Thus **g(n)** represents the best case or the lower bound. If there are positive constants n_o and**c** such that at and to the right of n_o, value of **f(n)** always lies on or above **g(n)**. This relationship has been shown graphically in Figure 1-1(a).

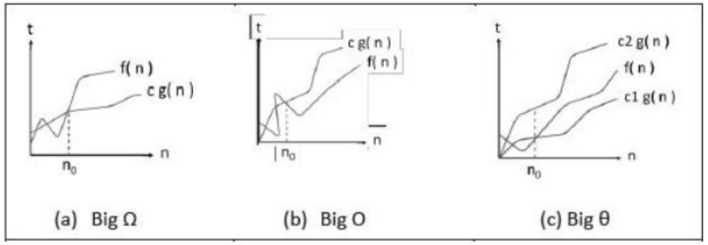


Figure 1-1. *Asymptotic representation of functions.*

The Big Oh class of functions would be of interest to us as it represents the class of functions that grow no faster than**g**. This means that for all values of **n** greater than some threshold n_o, all the functions in **O** have values that are no greater than **g**. Thus **g(n)** represents the worst case or upper bound. So, none of the functions in this class grow faster than **g**.

Using Asymptotic Notation this is represented

as f (n) <= c g (n)

where **c** is some constant > **0** and $n >= n_o >= 1$.

Thus if there are positive constants n_o and **c** such that at and to the right of n_o, value of **f(n)** always lies on or below **g(n)**. This relationship has been shown graphically in Figure 1-1(b).

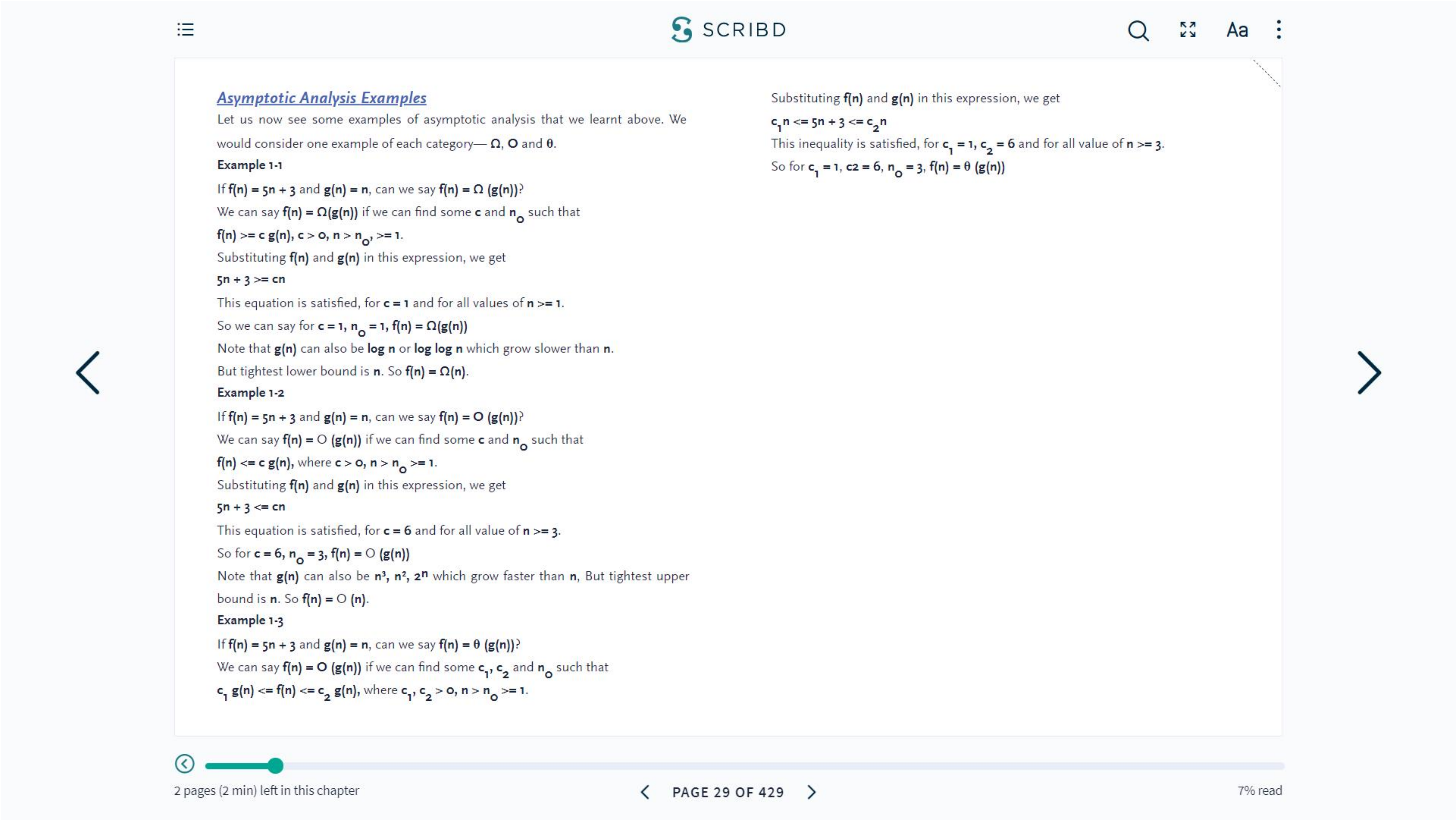
Big Theta represents the class of functions that are bounded by **g(n)** on either side. This means that for all values of **n** greater than some threshold n_o, all the functions in **?** have values that are greater than c_1**g(n)** and less than c_2**g(n)**. Using Asymptotic Notation, this is represented as

c_1 g (n) <= f (n) <= c_2 g (n)

where c_1 **and** c_2 are some constants > **0** and n >= n_o >= 1.

Thus, if there are positive constants n_o, c_1 **and** c_2 such that at and to the right of n_o, value of **f(n)** always is bounded by **g(n)** on either side. This relationship has been shown graphically in Figure 1-1(c).

While analyzing algorithms we are on the lookout for an algorithm that does better than the one that we are considering. Since big theta category represents a class of functions that grow at the same rate as the function **g** this category is usually not of interest to us.



Asymptotic Analysis Examples

Let us now see some examples of asymptotic analysis that we learnt above. We would consider one example of each category— **Ω, O** and **θ**.

Example 1-1

If $f(n) = 5n + 3$ and $g(n) = n$, can we say $f(n) = \Omega\ (g(n))$?

We can say $f(n) = \Omega(g(n))$ if we can find some **c** and n_0 such that

$f(n) >= c\ g(n), c > 0, n > n_0, >= 1$.

Substituting **f(n)** and **g(n)** in this expression, we get

$5n + 3 >= cn$

This equation is satisfied, for $c = 1$ and for all values of $n >= 1$.

So we can say for $c = 1, n_0 = 1, f(n) = \Omega(g(n))$

Note that **g(n)** can also be **log n** or **log log n** which grow slower than **n**.

But tightest lower bound is **n**. So $f(n) = \Omega(n)$.

Example 1-2

If $f(n) = 5n + 3$ and $g(n) = n$, can we say $f(n) = O\ (g(n))$?

We can say $f(n) = O\ (g(n))$ if we can find some **c** and n_0 such that

$f(n) <= c\ g(n)$, where $c > 0, n > n_0 >= 1$.

Substituting **f(n)** and **g(n)** in this expression, we get

$5n + 3 <= cn$

This equation is satisfied, for $c = 6$ and for all value of $n >= 3$.

So for $c = 6, n_0 = 3, f(n) = O\ (g(n))$

Note that **g(n)** can also be n^3, n^2, 2^n which grow faster than **n**, But tightest upper bound is **n**. So $f(n) = O\ (n)$.

Example 1-3

If $f(n) = 5n + 3$ and $g(n) = n$, can we say $f(n) = \theta\ (g(n))$?

We can say $f(n) = O\ (g(n))$ if we can find some c_1, c_2 and n_0 such that

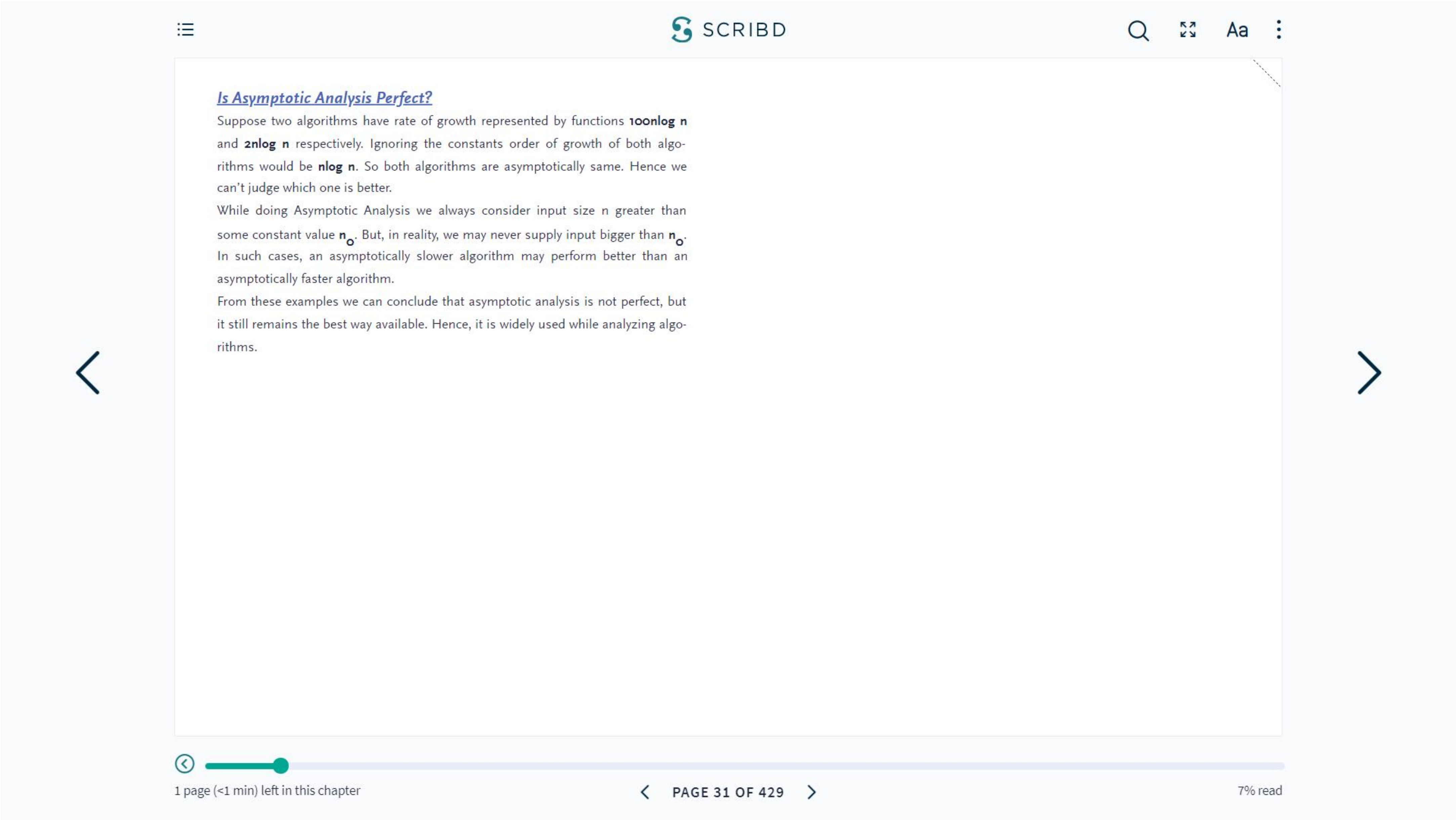
$c_1\ g(n) <= f(n) <= c_2\ g(n)$, where $c_1, c_2 > 0, n > n_0 >= 1$.

Substituting **f(n)** and **g(n)** in this expression, we get

$c_1 n <= 5n + 3 <= c_2 n$

This inequality is satisfied, for $c_1 = 1, c_2 = 6$ and for all value of $n >= 3$.

So for $c_1 = 1$, $c2 = 6$, $n_0 = 3$, $f(n) = \theta\ (g(n))$

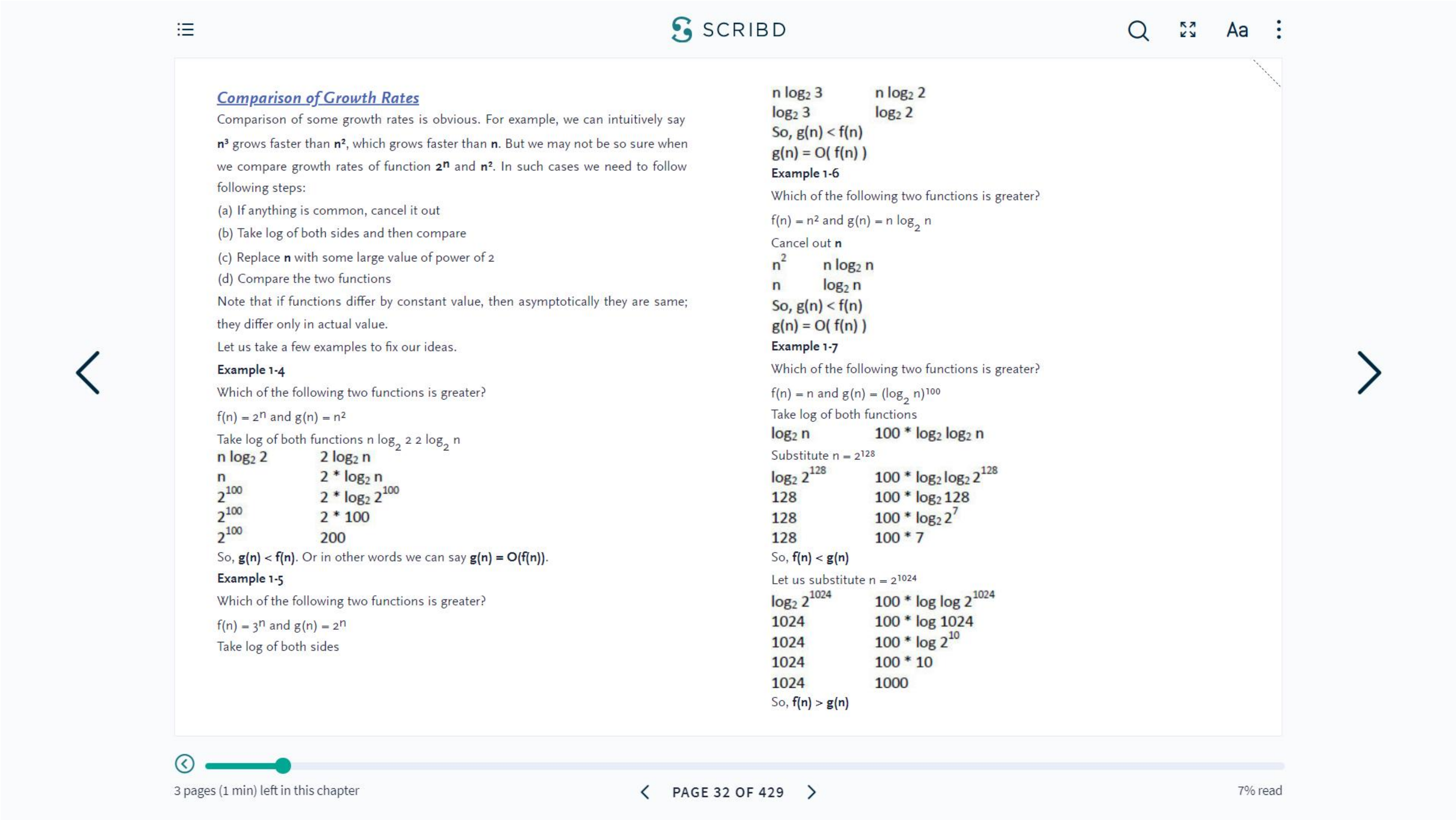


Is Asymptotic Analysis Perfect?

Suppose two algorithms have rate of growth represented by functions **100nlog n** and **2nlog n** respectively. Ignoring the constants order of growth of both algorithms would be **nlog n**. So both algorithms are asymptotically same. Hence we can't judge which one is better.

While doing Asymptotic Analysis we always consider input size n greater than some constant value $\mathbf{n_o}$. But, in reality, we may never supply input bigger than $\mathbf{n_o}$. In such cases, an asymptotically slower algorithm may perform better than an asymptotically faster algorithm.

From these examples we can conclude that asymptotic analysis is not perfect, but it still remains the best way available. Hence, it is widely used while analyzing algorithms.



Comparison of Growth Rates

Comparison of some growth rates is obvious. For example, we can intuitively say $\mathbf{n^3}$ grows faster than $\mathbf{n^2}$, which grows faster than **n**. But we may not be so sure when we compare growth rates of function $\mathbf{2^n}$ and $\mathbf{n^2}$. In such cases we need to follow following steps:

(a) If anything is common, cancel it out

(b) Take log of both sides and then compare

(c) Replace **n** with some large value of power of 2

(d) Compare the two functions

Note that if functions differ by constant value, then asymptotically they are same; they differ only in actual value.

Let us take a few examples to fix our ideas.

Example 1-4

Which of the following two functions is greater?

$f(n) = 2^n$ and $g(n) = n^2$

Take log of both functions $n \log_2 2$ $2 \log_2 n$

$n \log_2 2$	$2 \log_2 n$
n	$2 * \log_2 n$
2^{100}	$2 * \log_2 2^{100}$
2^{100}	$2 * 100$
2^{100}	200

So, **g(n) < f(n)**. Or in other words we can say **g(n) = O(f(n))**.

Example 1-5

Which of the following two functions is greater?

$f(n) = 3^n$ and $g(n) = 2^n$

Take log of both sides

$n \log_2 3$	$n \log_2 2$
$\log_2 3$	$\log_2 2$

So, g(n) < f(n)

g(n) = O(f(n))

Example 1-6

Which of the following two functions is greater?

$f(n) = n^2$ and $g(n) = n \log_2 n$

Cancel out **n**

n^2	$n \log_2 n$
n	$\log_2 n$

So, g(n) < f(n)

g(n) = O(f(n))

Example 1-7

Which of the following two functions is greater?

$f(n) = n$ and $g(n) = (\log_2 n)^{100}$

Take log of both functions

$\log_2 n$	$100 * \log_2 \log_2 n$

Substitute $n = 2^{128}$

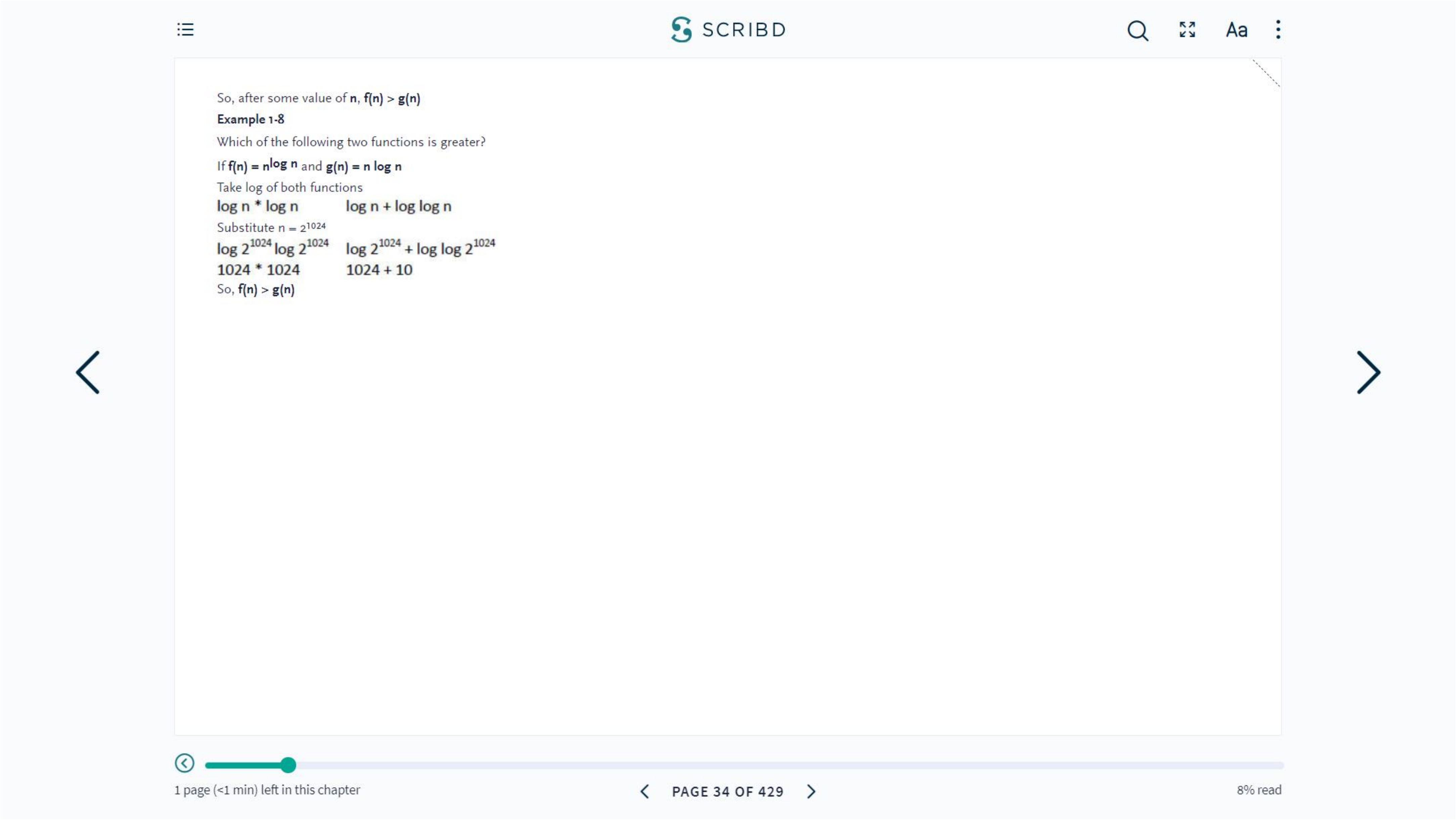
$\log_2 2^{128}$	$100 * \log_2 \log_2 2^{128}$
128	$100 * \log_2 128$
128	$100 * \log_2 2^7$
128	$100 * 7$

So, **f(n) < g(n)**

Let us substitute $n = 2^{1024}$

$\log_2 2^{1024}$	$100 * \log \log 2^{1024}$
1024	$100 * \log 1024$
1024	$100 * \log 2^{10}$
1024	$100 * 10$
1024	1000

So, **f(n) > g(n)**



So, after some value of **n**, **f(n) > g(n)**

Example 1-8

Which of the following two functions is greater?

If **f(n) = $n^{\log n}$** and **g(n) = n log n**

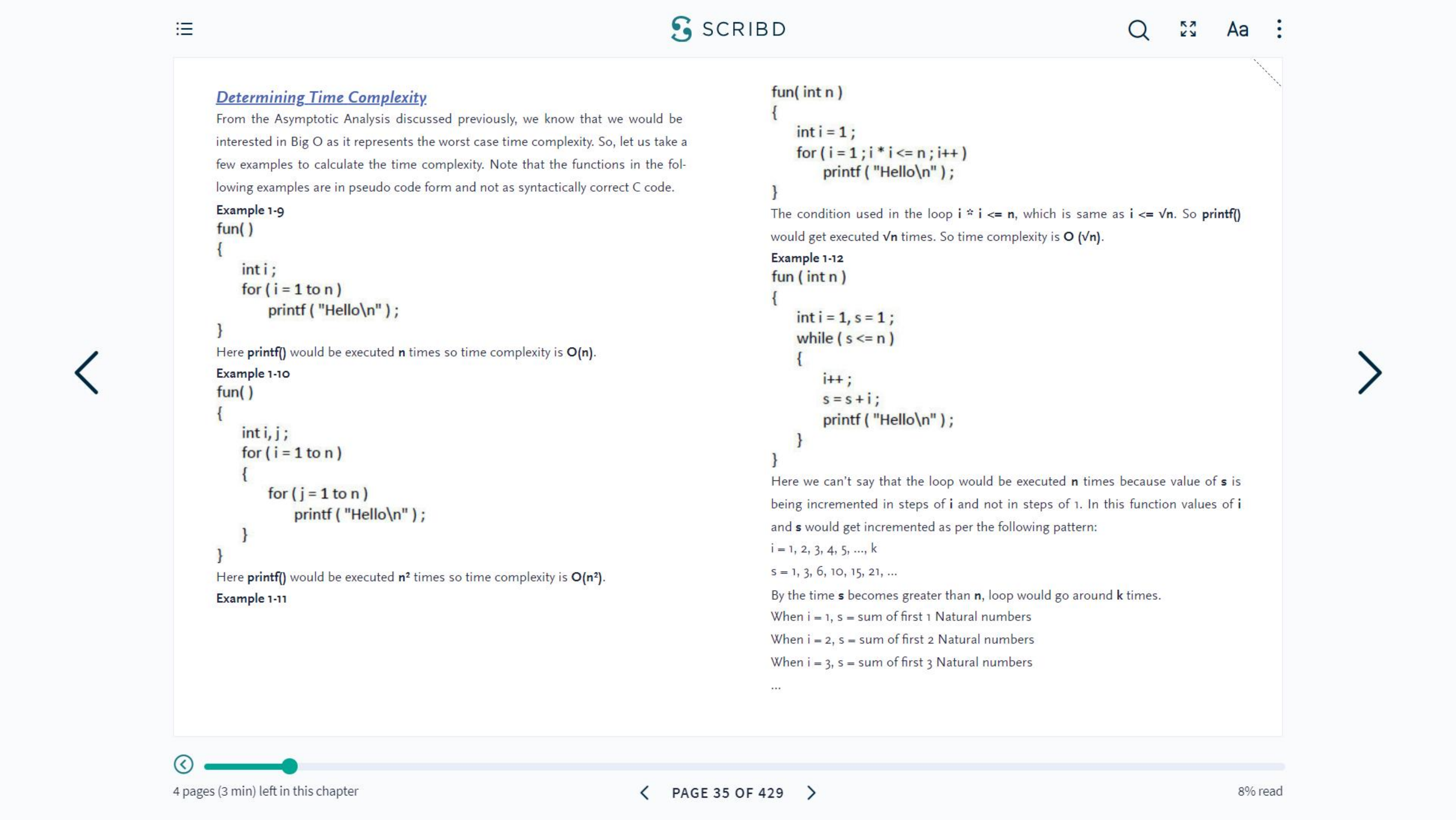
Take log of both functions

log n * log n	log n + log log n

Substitute n = 2^{1024}

$\log 2^{1024} \log 2^{1024}$	$\log 2^{1024} + \log\log 2^{1024}$
1024 * 1024	1024 + 10

So, **f(n) > g(n)**



Determining Time Complexity

From the Asymptotic Analysis discussed previously, we know that we would be interested in Big O as it represents the worst case time complexity. So, let us take a few examples to calculate the time complexity. Note that the functions in the following examples are in pseudo code form and not as syntactically correct C code.

Example 1-9

```
fun( )
{
    int i ;
    for ( i = 1 to n )
        printf ( "Hello\n" ) ;
}
```

Here **printf()** would be executed **n** times so time complexity is **O(n)**.

Example 1-10

```
fun( )
{
    int i, j ;
    for ( i = 1 to n )
    {
        for ( j = 1 to n )
            printf ( "Hello\n" ) ;
    }
}
```

Here **printf()** would be executed **n^2** times so time complexity is **$O(n^2)$**.

Example 1-11

```
fun( int n )
{
    int i = 1 ;
    for ( i = 1 ; i * i <= n ; i++ )
        printf ( "Hello\n" ) ;
}
```

The condition used in the loop **i * i <= n**, which is same as **i <= √n**. So **printf()** would get executed **√n** times. So time complexity is **O (√n)**.

Example 1-12

```
fun ( int n )
{
    int i = 1, s = 1 ;
    while ( s <= n )
    {
        i++ ;
        s = s + i ;
        printf ( "Hello\n" ) ;
    }
}
```

Here we can't say that the loop would be executed **n** times because value of **s** is being incremented in steps of **i** and not in steps of 1. In this function values of **i** and **s** would get incremented as per the following pattern:

i = 1, 2, 3, 4, 5, ..., k

s = 1, 3, 6, 10, 15, 21, ...

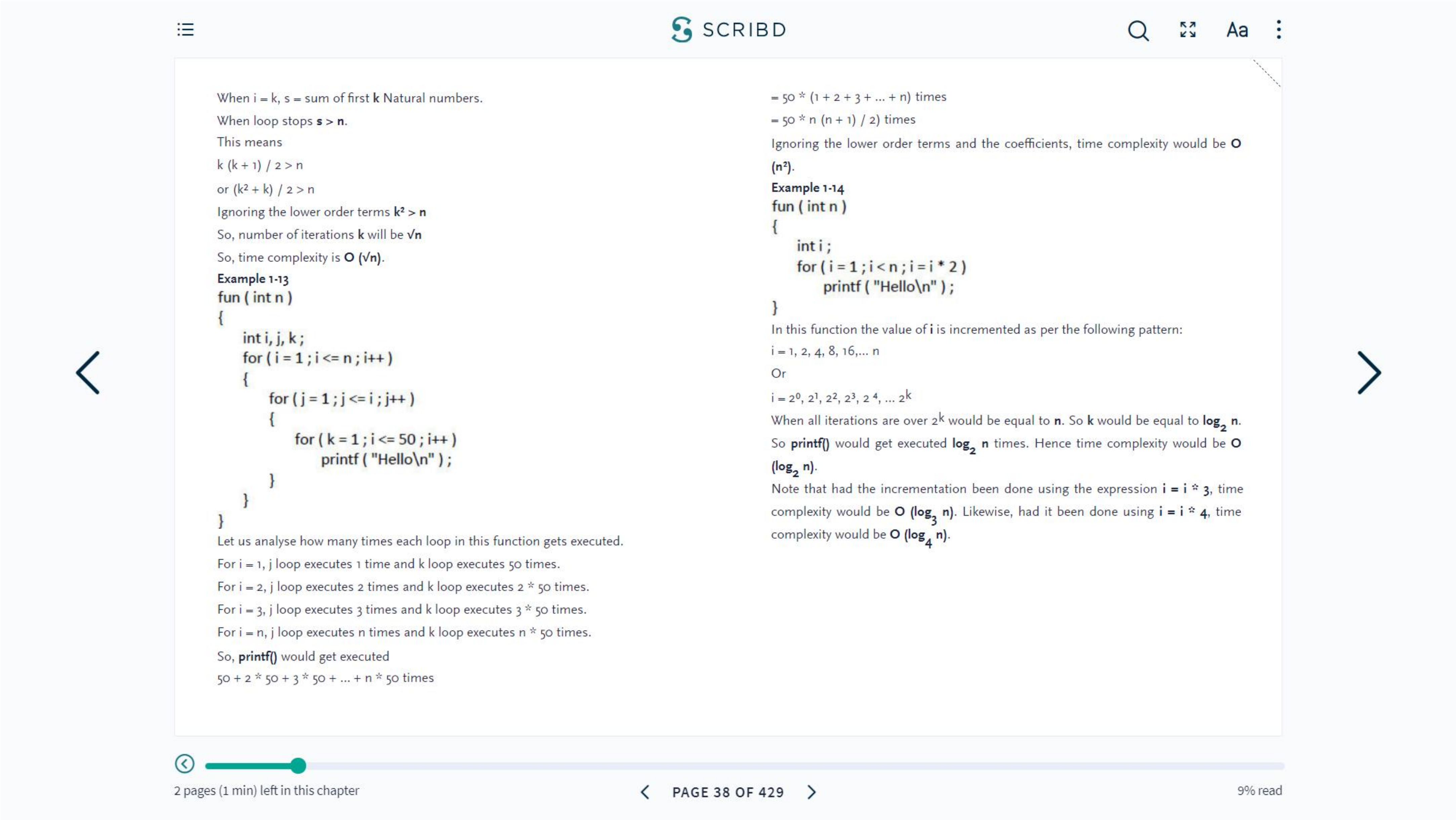
By the time **s** becomes greater than **n**, loop would go around **k** times.

When i = 1, s = sum of first 1 Natural numbers

When i = 2, s = sum of first 2 Natural numbers

When i = 3, s = sum of first 3 Natural numbers

...



When i = k, s = sum of first **k** Natural numbers.

When loop stops **s > n**.

This means

$k (k + 1) / 2 > n$

or $(k^2 + k) / 2 > n$

Ignoring the lower order terms $\mathbf{k^2 > n}$

So, number of iterations **k** will be **√n**

So, time complexity is **O (√n)**.

Example 1-13

```
fun ( int n )
{
    int i, j, k ;
    for ( i = 1 ; i <= n ; i++ )
    {
        for ( j = 1 ; j <= i ; j++ )
        {
            for ( k = 1 ; i <= 50 ; i++ )
                printf ( "Hello\n" ) ;
        }
    }
}
```

Let us analyse how many times each loop in this function gets executed.

For i = 1, j loop executes 1 time and k loop executes 50 times.

For i = 2, j loop executes 2 times and k loop executes 2 * 50 times.

For i = 3, j loop executes 3 times and k loop executes 3 * 50 times.

For i = n, j loop executes n times and k loop executes n * 50 times.

So, **printf()** would get executed

50 + 2 * 50 + 3 * 50 + ... + n * 50 times

= 50 * (1 + 2 + 3 + ... + n) times

= 50 * n (n + 1) / 2) times

Ignoring the lower order terms and the coefficients, time complexity would be $\mathbf{O (n^2)}$.

Example 1-14

```
fun ( int n )
{
    int i ;
    for ( i = 1 ; i < n ; i = i * 2 )
        printf ( "Hello\n" ) ;
}
```

In this function the value of **i** is incremented as per the following pattern:

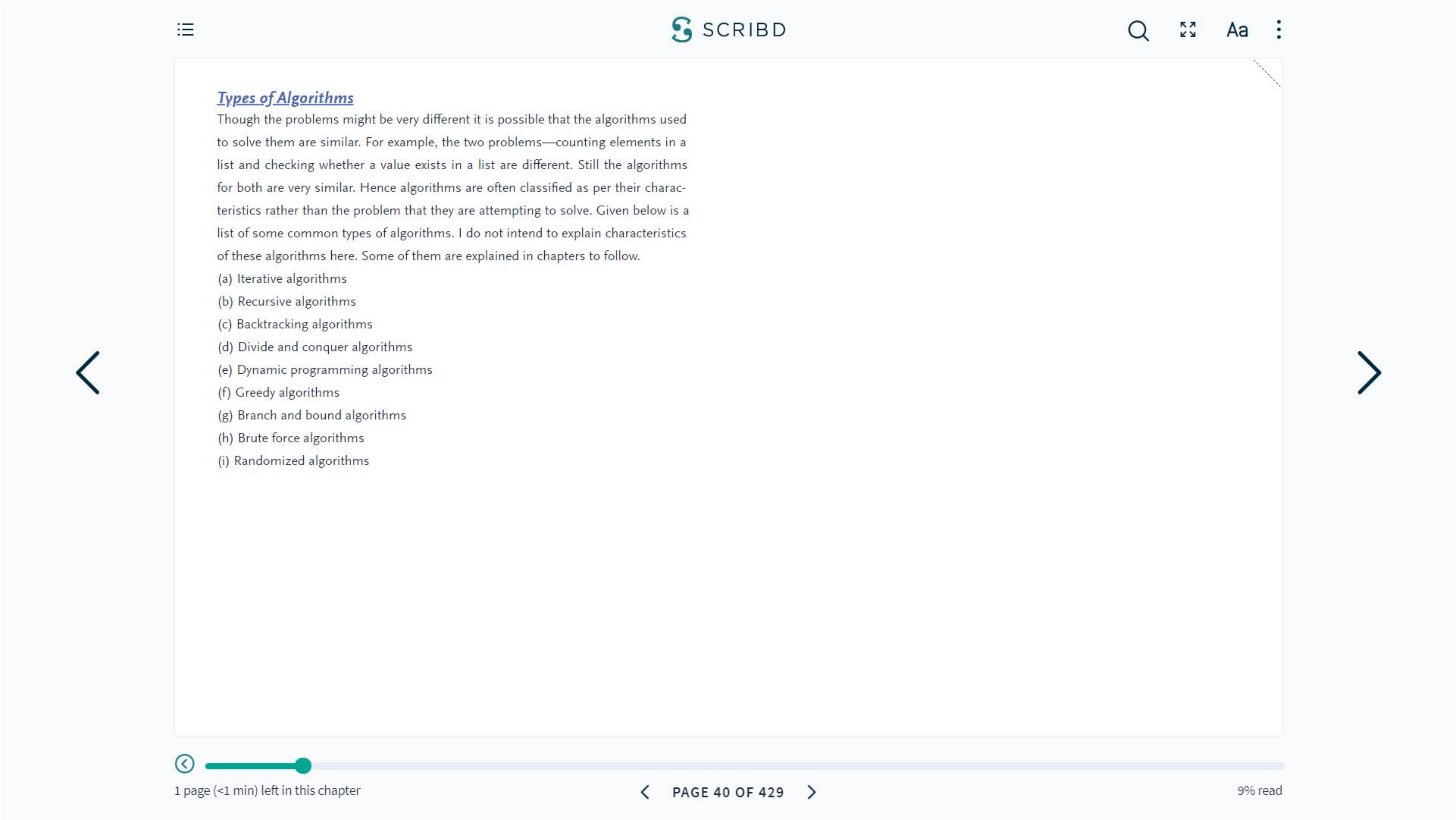
i = 1, 2, 4, 8, 16,... n

Or

$i = 2^0, 2^1, 2^2, 2^3, 2^4, \ldots 2^k$

When all iterations are over 2^k would be equal to **n**. So **k** would be equal to $\mathbf{\log_2 n}$. So **printf()** would get executed $\mathbf{\log_2 n}$ times. Hence time complexity would be $\mathbf{O (\log_2 n)}$.

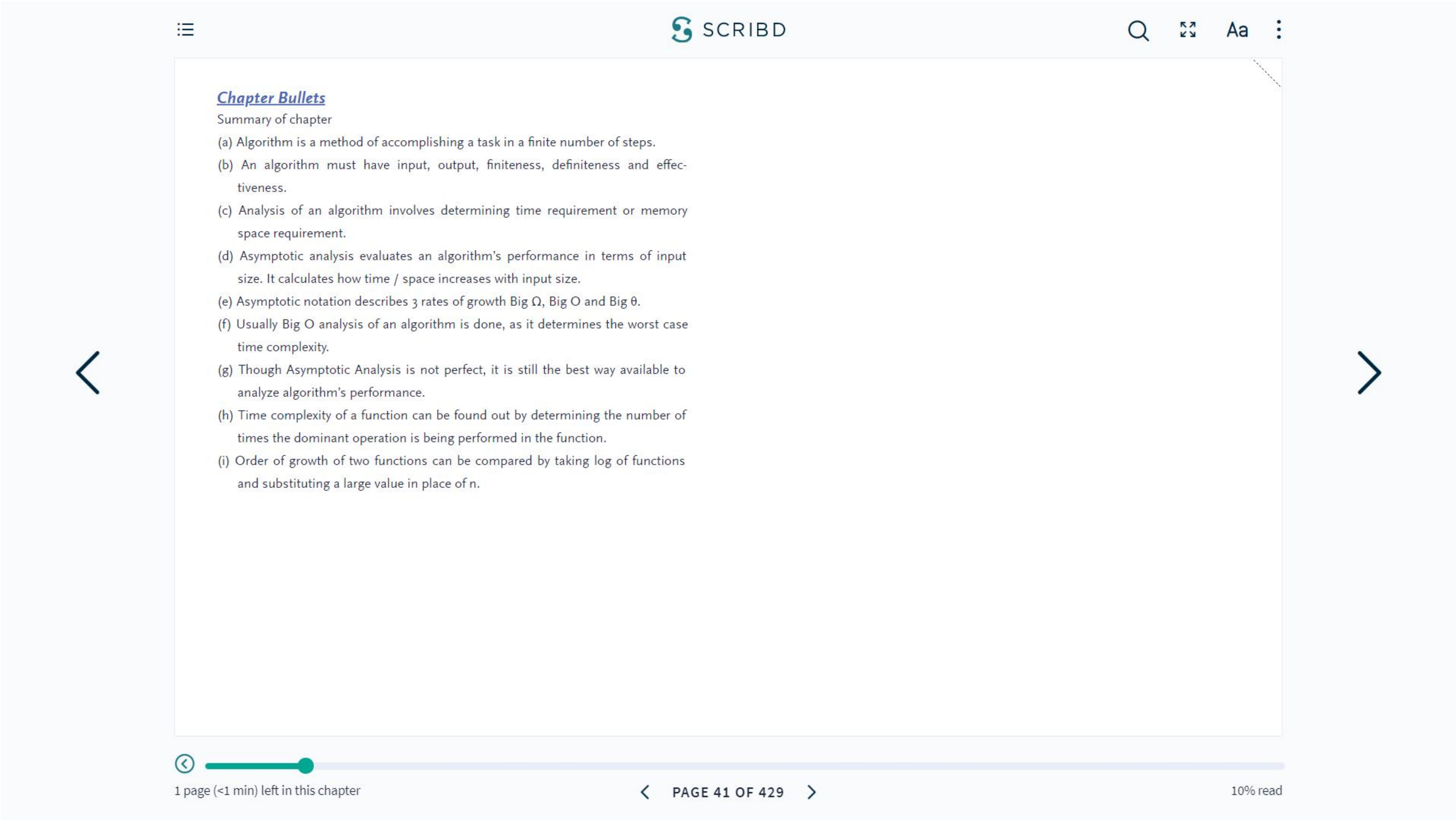
Note that had the incrementation been done using the expression **i = i * 3**, time complexity would be $\mathbf{O (\log_3 n)}$. Likewise, had it been done using **i = i * 4**, time complexity would be $\mathbf{O (\log_4 n)}$.



Types of Algorithms

Though the problems might be very different it is possible that the algorithms used to solve them are similar. For example, the two problems—counting elements in a list and checking whether a value exists in a list are different. Still the algorithms for both are very similar. Hence algorithms are often classified as per their characteristics rather than the problem that they are attempting to solve. Given below is a list of some common types of algorithms. I do not intend to explain characteristics of these algorithms here. Some of them are explained in chapters to follow.

(a) Iterative algorithms
(b) Recursive algorithms
(c) Backtracking algorithms
(d) Divide and conquer algorithms
(e) Dynamic programming algorithms
(f) Greedy algorithms
(g) Branch and bound algorithms
(h) Brute force algorithms
(i) Randomized algorithms



Chapter Bullets

Summary of chapter

(a) Algorithm is a method of accomplishing a task in a finite number of steps.

(b) An algorithm must have input, output, finiteness, definiteness and effectiveness.

(c) Analysis of an algorithm involves determining time requirement or memory space requirement.

(d) Asymptotic analysis evaluates an algorithm's performance in terms of input size. It calculates how time / space increases with input size.

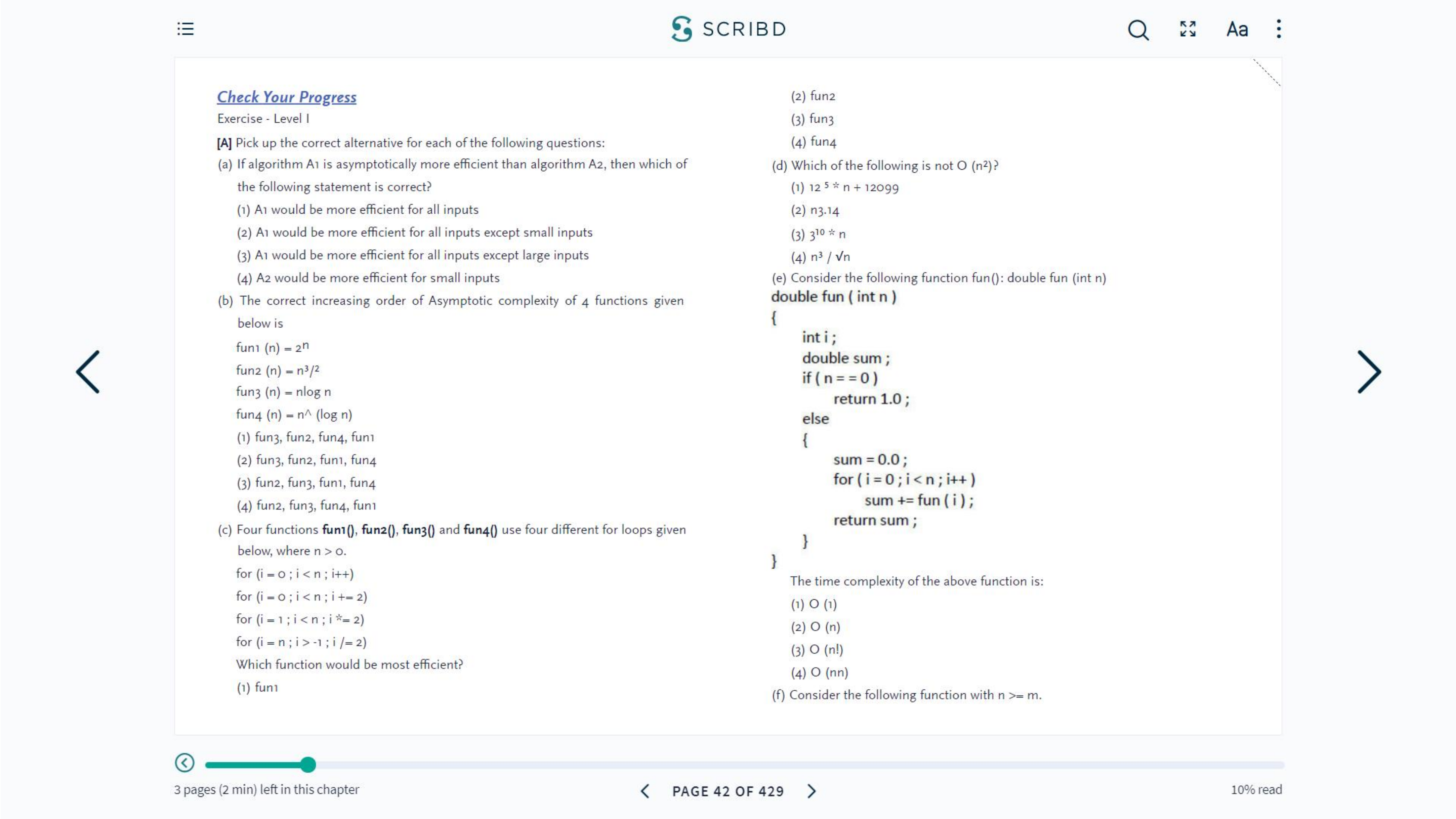
(e) Asymptotic notation describes 3 rates of growth Big Ω, Big O and Big θ.

(f) Usually Big O analysis of an algorithm is done, as it determines the worst case time complexity.

(g) Though Asymptotic Analysis is not perfect, it is still the best way available to analyze algorithm's performance.

(h) Time complexity of a function can be found out by determining the number of times the dominant operation is being performed in the function.

(i) Order of growth of two functions can be compared by taking log of functions and substituting a large value in place of n.



Check Your Progress

Exercise - Level I

[A] Pick up the correct alternative for each of the following questions:

(a) If algorithm A1 is asymptotically more efficient than algorithm A2, then which of the following statement is correct?

(1) A1 would be more efficient for all inputs

(2) A1 would be more efficient for all inputs except small inputs

(3) A1 would be more efficient for all inputs except large inputs

(4) A2 would be more efficient for small inputs

(b) The correct increasing order of Asymptotic complexity of 4 functions given below is

fun1 (n) = 2^n

fun2 (n) = $n^{3/2}$

fun3 (n) = nlog n

fun4 (n) = n^ (log n)

(1) fun3, fun2, fun4, fun1

(2) fun3, fun2, fun1, fun4

(3) fun2, fun3, fun1, fun4

(4) fun2, fun3, fun4, fun1

(c) Four functions **fun1()**, **fun2()**, **fun3()** and **fun4()** use four different for loops given below, where n > 0.

for (i = 0 ; i < n ; i++)

for (i = 0 ; i < n ; i += 2)

for (i = 1 ; i < n ; i *= 2)

for (i = n ; i > -1 ; i /= 2)

Which function would be most efficient?

(1) fun1

(2) fun2

(3) fun3

(4) fun4

(d) Which of the following is not O (n^2)?

(1) 12^5 * n + 12099

(2) n3.14

(3) 3^{10} * n

(4) $n^3 / \sqrt{n}$

(e) Consider the following function fun(): double fun (int n)

```
double fun ( int n )
{
    int i ;
    double sum ;
    if ( n = = 0 )
        return 1.0 ;
    else
    {
        sum = 0.0 ;
        for ( i = 0 ; i < n ; i++ )
            sum += fun ( i ) ;
        return sum ;
    }
}
```

The time complexity of the above function is:

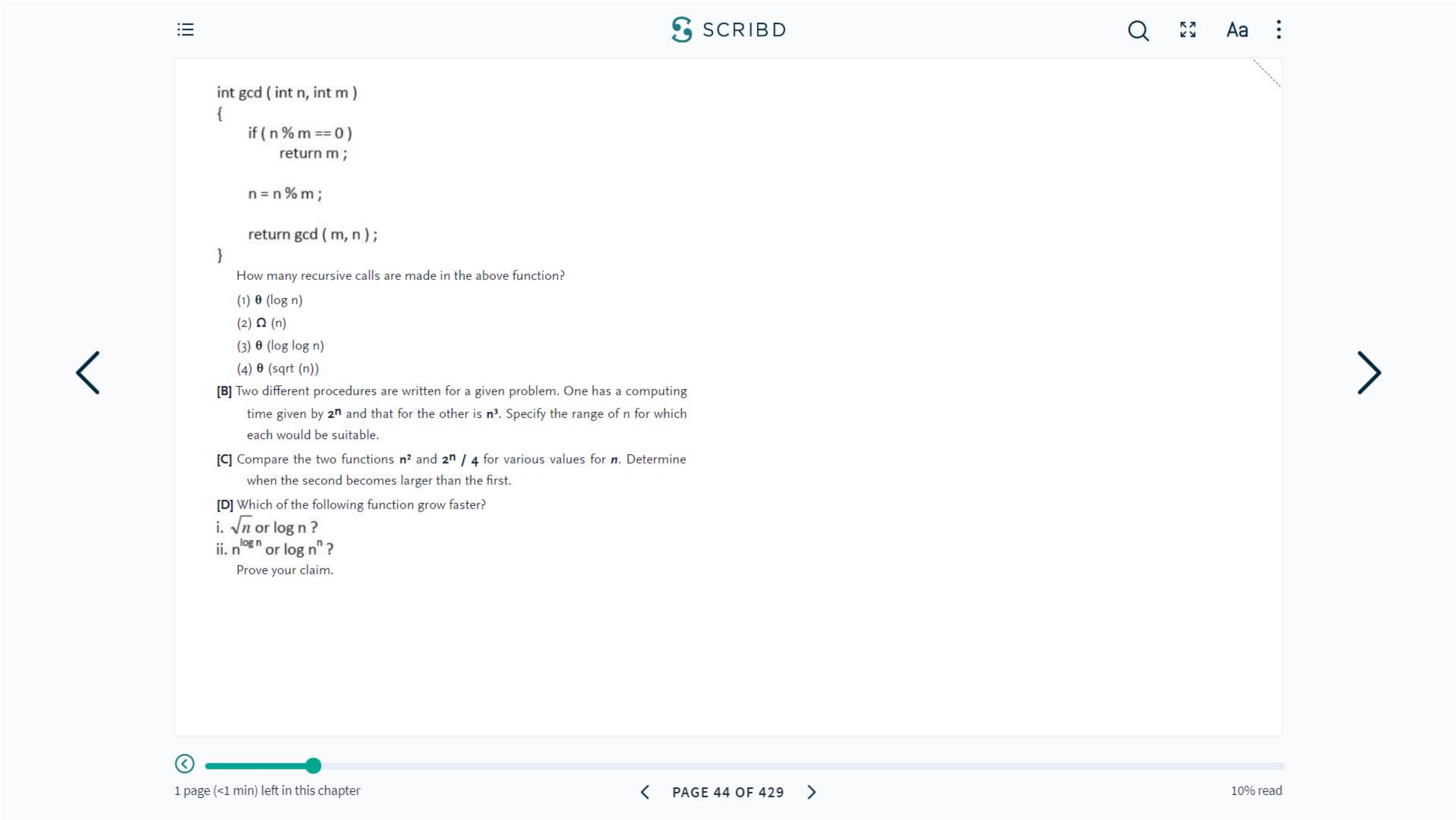
(1) O (1)

(2) O (n)

(3) O (n!)

(4) O (nn)

(f) Consider the following function with n >= m.



```
int gcd ( int n, int m )
{
      if ( n % m == 0 )
            return m ;

      n = n % m ;

      return gcd ( m, n ) ;
}
```

How many recursive calls are made in the above function?

(1) θ (log n)

(2) Ω (n)

(3) θ (log log n)

(4) θ (sqrt (n))

[B] Two different procedures are written for a given problem. One has a computing time given by 2^n and that for the other is n^3. Specify the range of n for which each would be suitable.

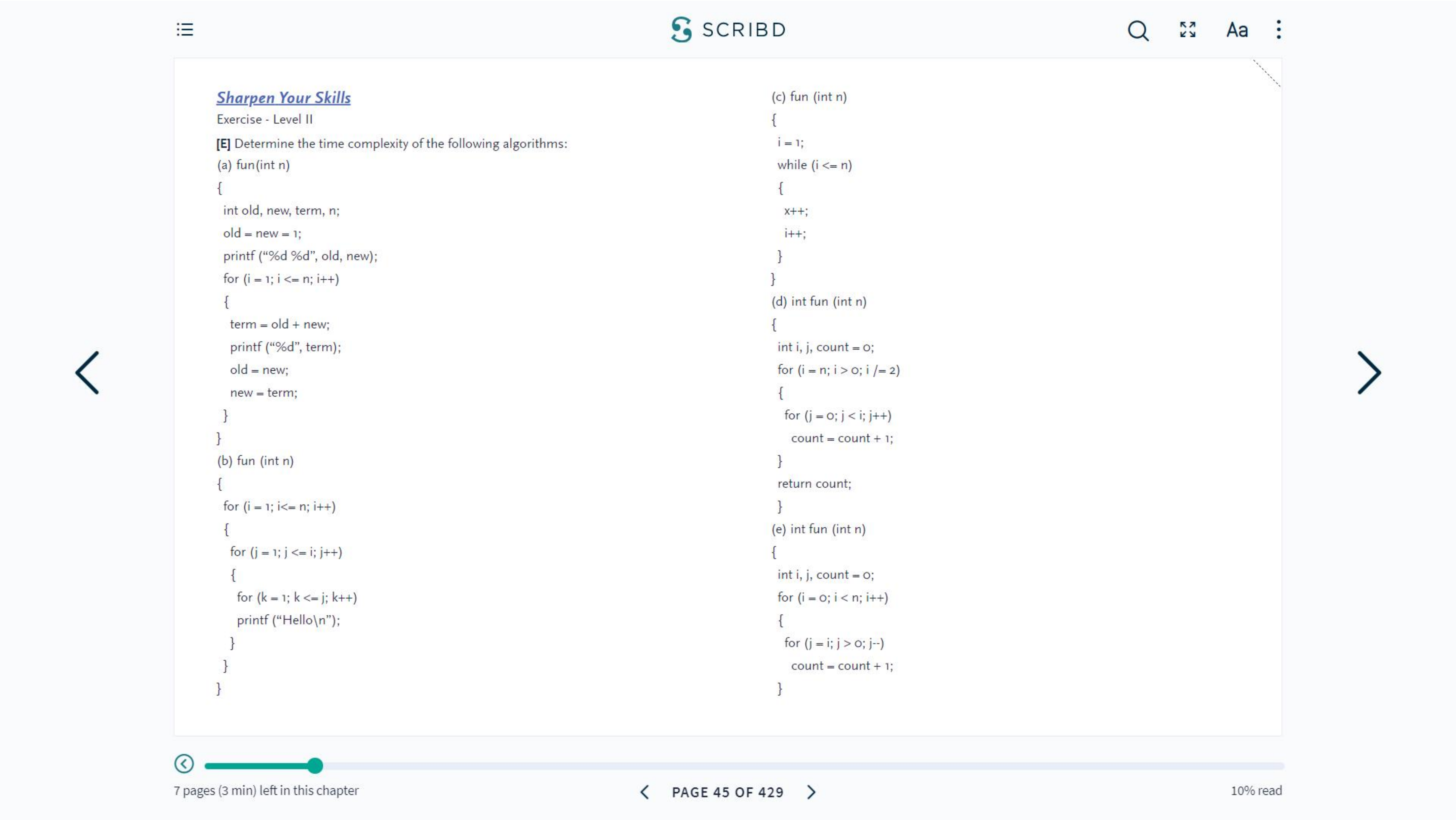
[C] Compare the two functions n^2 and $2^n / 4$ for various values for *n*. Determine when the second becomes larger than the first.

[D] Which of the following function grow faster?

i. $\sqrt{n}$ or log n ?

ii. $n^{\log n}$ or $\log n^n$?

Prove your claim.

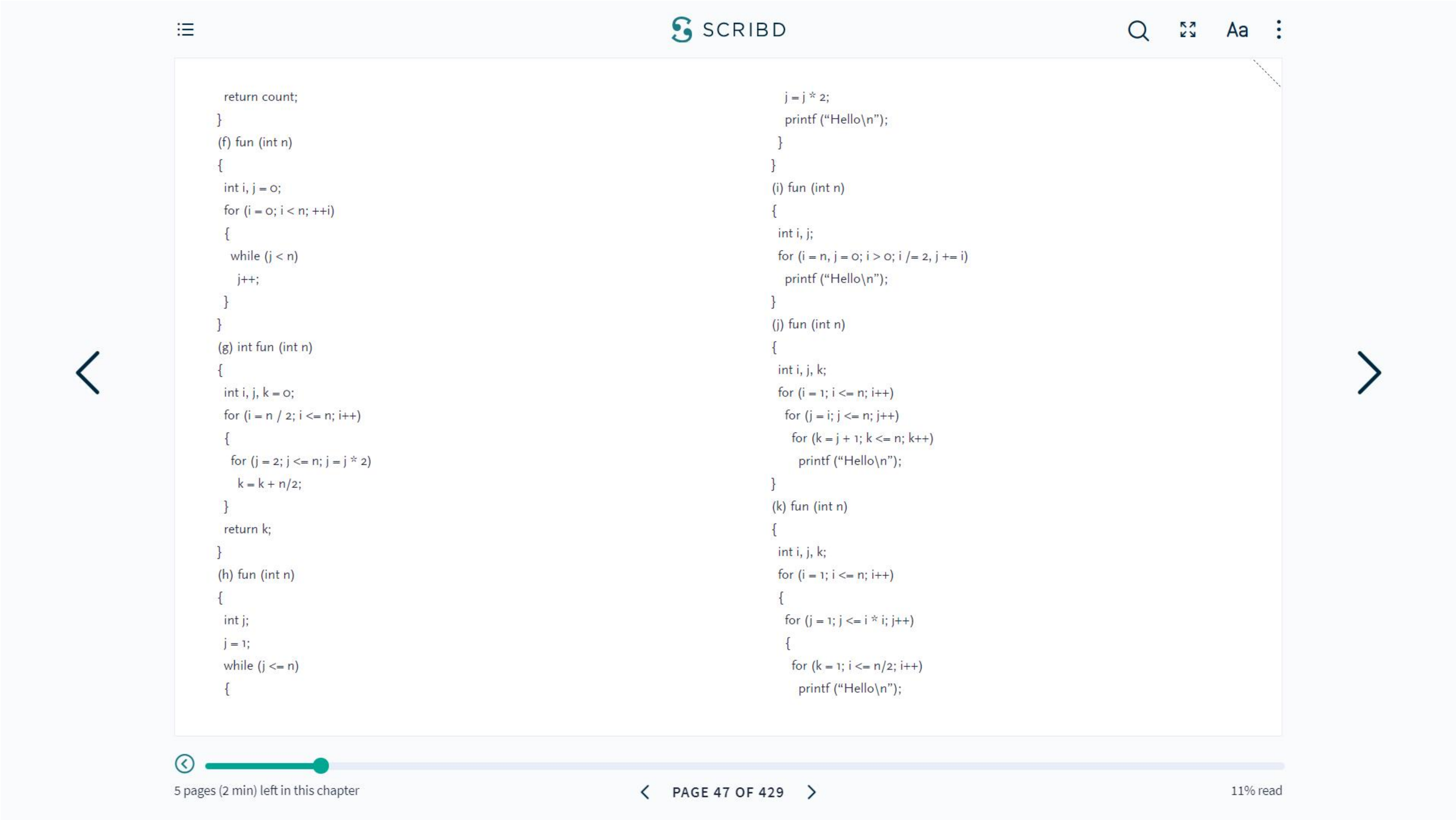


Sharpen Your Skills

Exercise - Level II

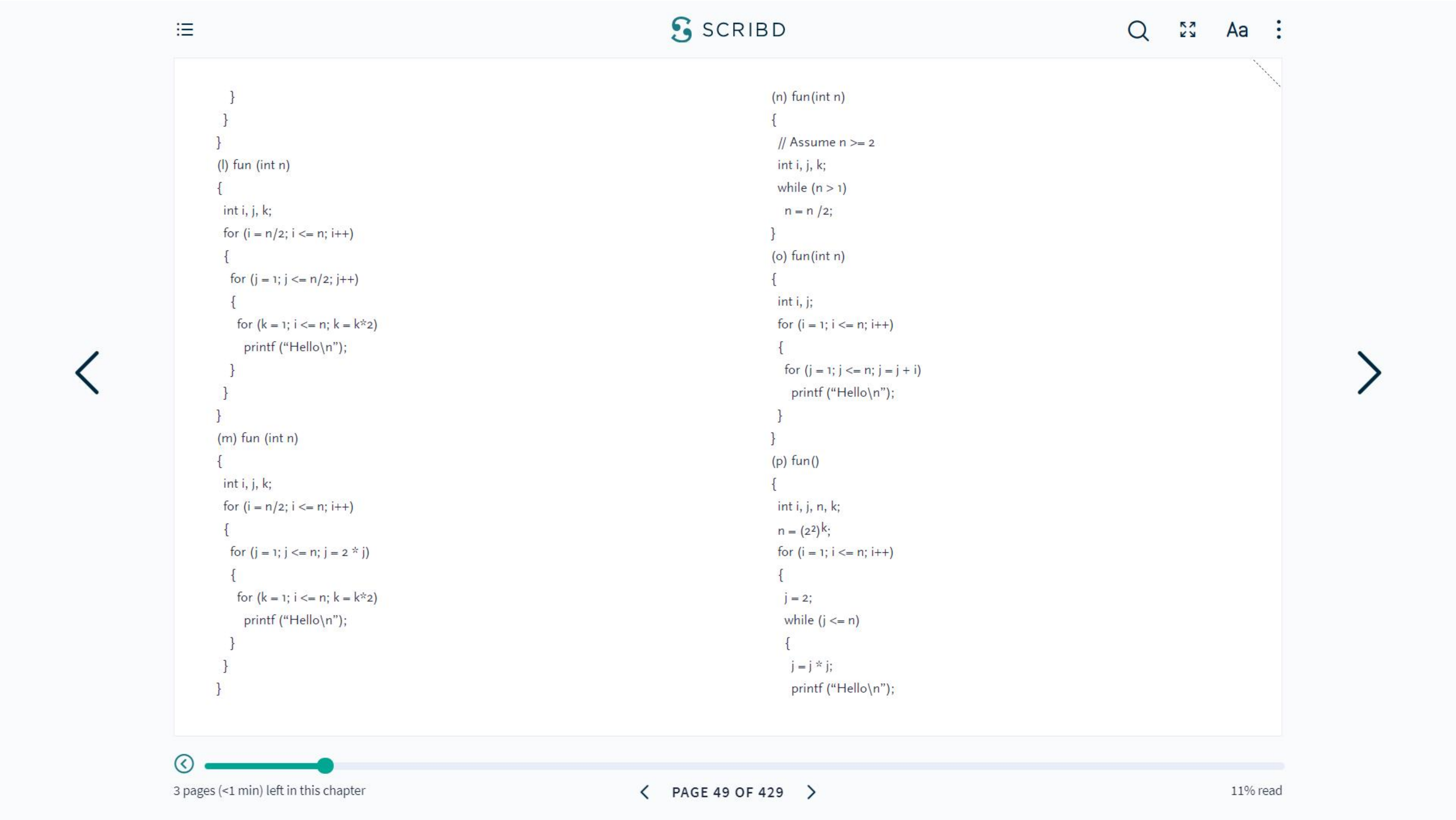
[E] Determine the time complexity of the following algorithms:

```
(a) fun(int n)
{
 int old, new, term, n;
 old = new = 1;
 printf ("%d %d", old, new);
 for (i = 1; i <= n; i++)
 {
  term = old + new;
  printf ("%d", term);
  old = new;
  new = term;
 }
}
(b) fun (int n)
{
 for (i = 1; i<= n; i++)
 {
  for (j = 1; j <= i; j++)
  {
   for (k = 1; k <= j; k++)
   printf ("Hello\n");
  }
 }
}
(c) fun (int n)
{
 i = 1;
 while (i <= n)
 {
  x++;
  i++;
 }
}
(d) int fun (int n)
{
 int i, j, count = 0;
 for (i = n; i > 0; i /= 2)
 {
  for (j = 0; j < i; j++)
   count = count + 1;
 }
 return count;
 }
(e) int fun (int n)
{
 int i, j, count = 0;
 for (i = 0; i < n; i++)
 {
  for (j = i; j > 0; j--)
   count = count + 1;
 }
```



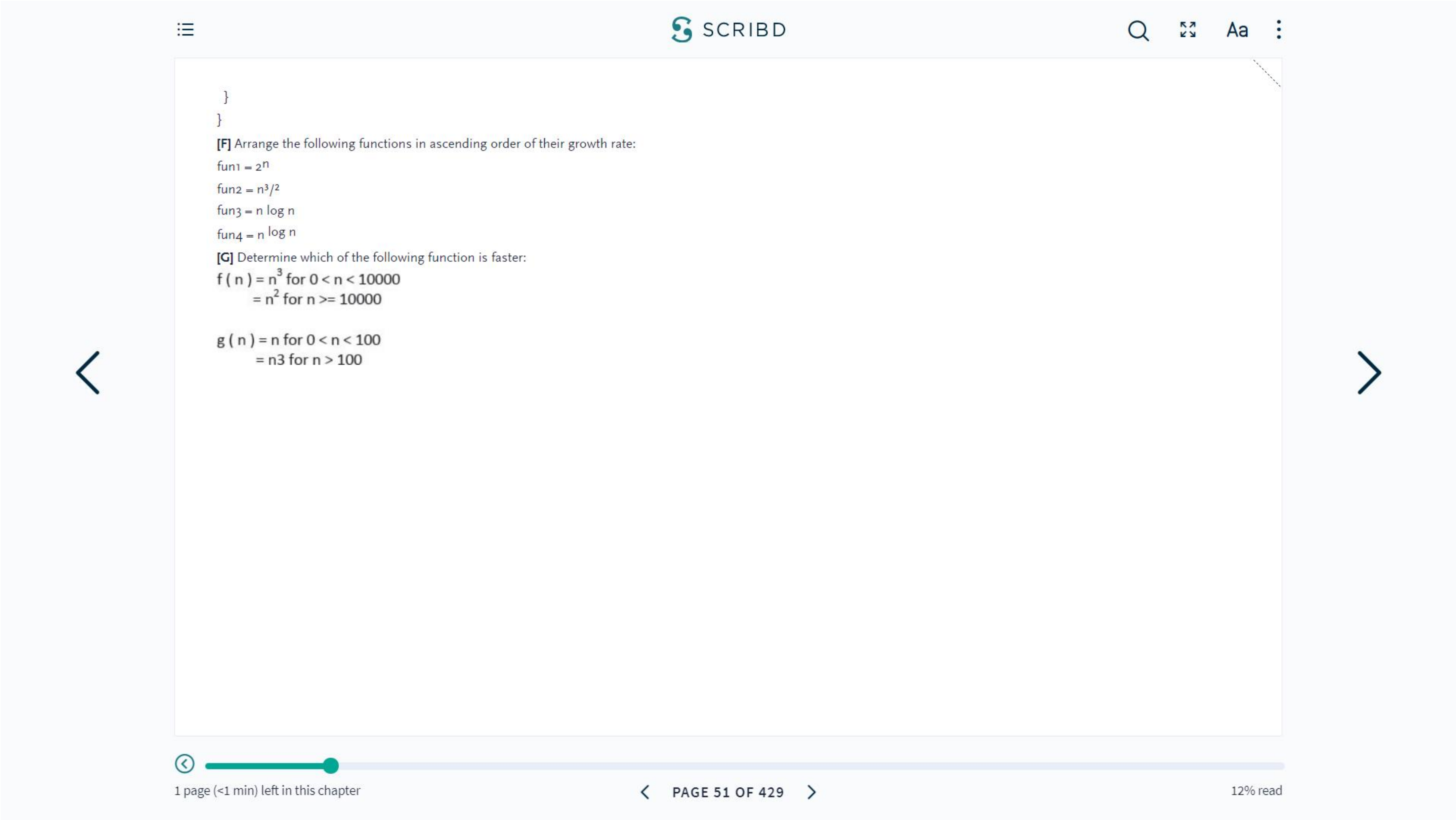
```
 return count;
}
(f) fun (int n)
{
 int i, j = 0;
 for (i = 0; i < n; ++i)
 {
  while (j < n)
   j++;
 }
}
(g) int fun (int n)
{
 int i, j, k = 0;
 for (i = n / 2; i <= n; i++)
 {
  for (j = 2; j <= n; j = j * 2)
   k = k + n/2;
 }
 return k;
}
(h) fun (int n)
{
 int j;
 j = 1;
 while (j <= n)
 {
```

```
  j = j * 2;
  printf (“Hello\n”);
 }
}
(i) fun (int n)
{
 int i, j;
 for (i = n, j = 0; i > 0; i /= 2, j += i)
  printf (“Hello\n”);
}
(j) fun (int n)
{
 int i, j, k;
 for (i = 1; i <= n; i++)
  for (j = i; j <= n; j++)
   for (k = j + 1; k <= n; k++)
    printf (“Hello\n”);
}
(k) fun (int n)
{
 int i, j, k;
 for (i = 1; i <= n; i++)
 {
  for (j = 1; j <= i * i; j++)
  {
   for (k = 1; i <= n/2; i++)
    printf (“Hello\n”);
```



```
    }
  }
}
(l) fun (int n)
{
  int i, j, k;
  for (i = n/2; i <= n; i++)
  {
    for (j = 1; j <= n/2; j++)
    {
      for (k = 1; i <= n; k = k*2)
        printf ("Hello\n");
    }
  }
}
(m) fun (int n)
{
  int i, j, k;
  for (i = n/2; i <= n; i++)
  {
    for (j = 1; j <= n; j = 2 * j)
    {
      for (k = 1; i <= n; k = k*2)
        printf ("Hello\n");
    }
  }
}
```

```
(n) fun(int n)
{
  // Assume n >= 2
  int i, j, k;
  while (n > 1)
    n = n /2;
}
(o) fun(int n)
{
  int i, j;
  for (i = 1; i <= n; i++)
  {
    for (j = 1; j <= n; j = j + i)
      printf ("Hello\n");
  }
}
(p) fun()
{
  int i, j, n, k;
  n = (2^2)^k;
  for (i = 1; i <= n; i++)
  {
    j = 2;
    while (j <= n)
    {
      j = j * j;
      printf ("Hello\n");
```



```
 }
}
```

[F] Arrange the following functions in ascending order of their growth rate:

fun1 = 2^n

fun2 = $n^{3/2}$

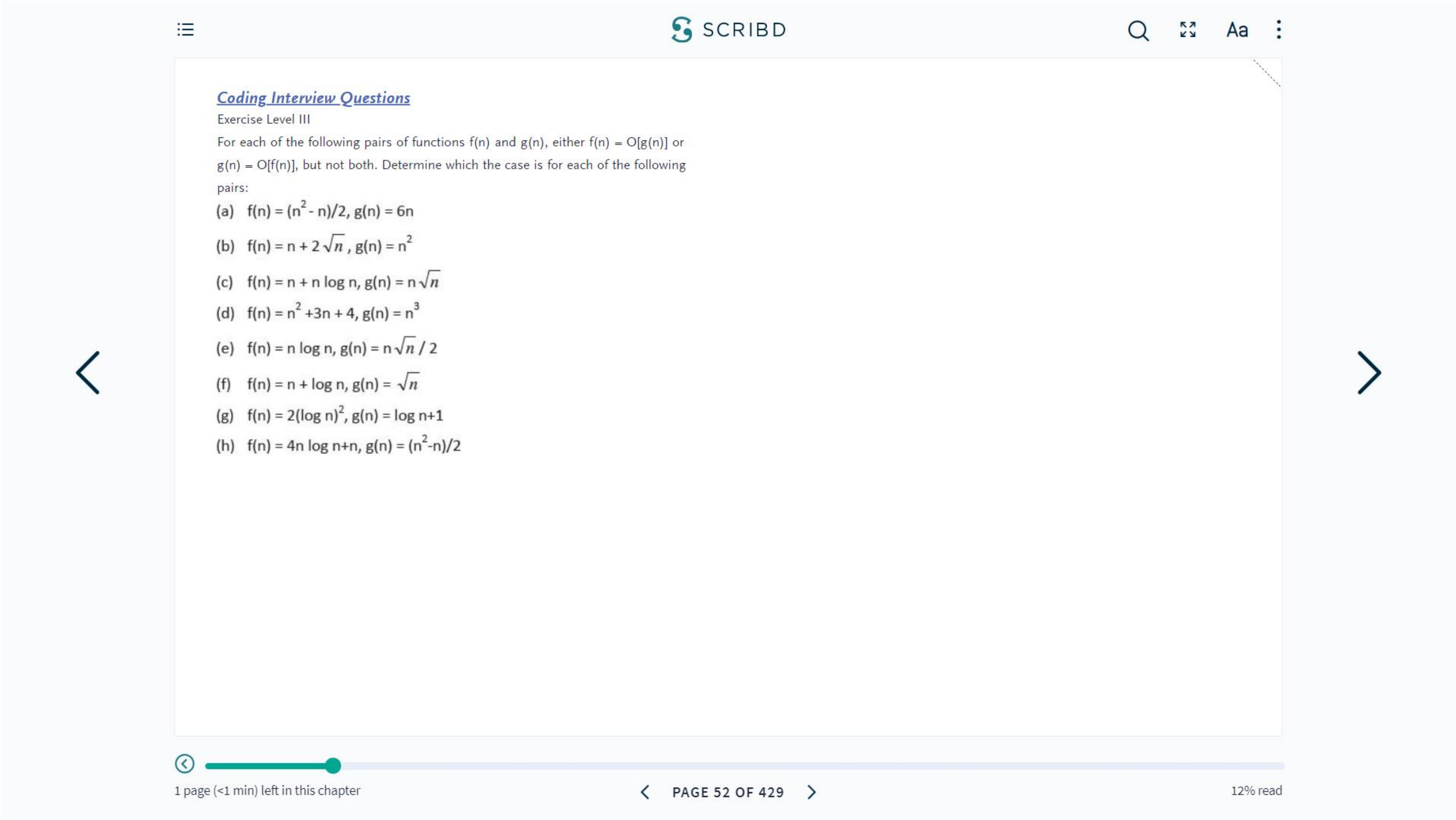
fun3 = n log n

fun4 = $n^{\log n}$

[G] Determine which of the following function is faster:

$$f(n) = n^3 \text{ for } 0 < n < 10000$$
$$= n^2 \text{ for } n >= 10000$$

$$g(n) = n \text{ for } 0 < n < 100$$
$$= n3 \text{ for } n > 100$$



Coding Interview Questions

Exercise Level III

For each of the following pairs of functions f(n) and g(n), either f(n) = O[g(n)] or g(n) = O[f(n)], but not both. Determine which the case is for each of the following pairs:

(a) $f(n) = (n^2 - n)/2,\ g(n) = 6n$

(b) $f(n) = n + 2\sqrt{n},\ g(n) = n^2$

(c) $f(n) = n + n \log n,\ g(n) = n\sqrt{n}$

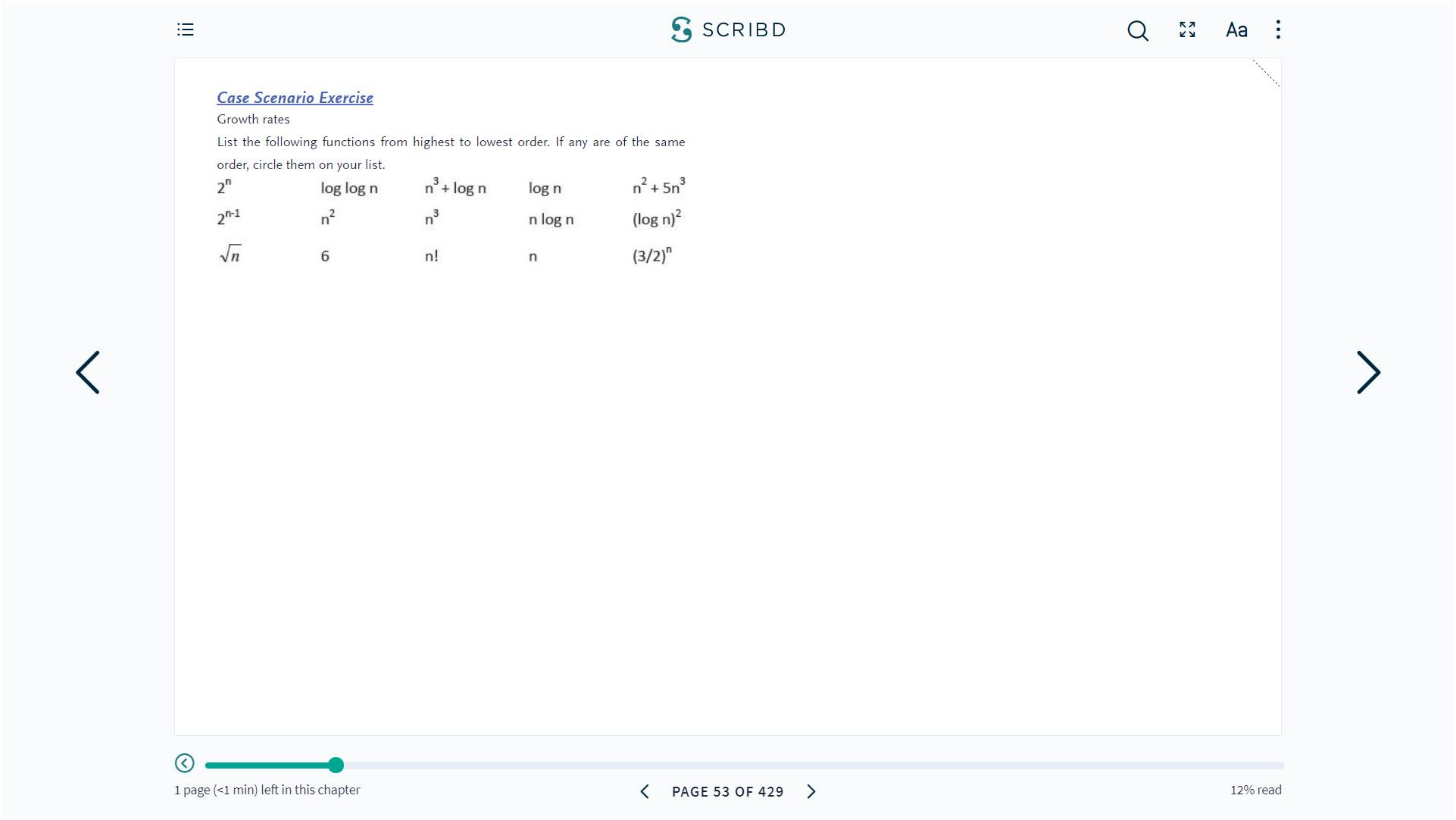
(d) $f(n) = n^2 + 3n + 4,\ g(n) = n^3$

(e) $f(n) = n \log n,\ g(n) = n\sqrt{n} / 2$

(f) $f(n) = n + \log n,\ g(n) = \sqrt{n}$

(g) $f(n) = 2(\log n)^2,\ g(n) = \log n+1$

(h) $f(n) = 4n \log n+n,\ g(n) = (n^2-n)/2$

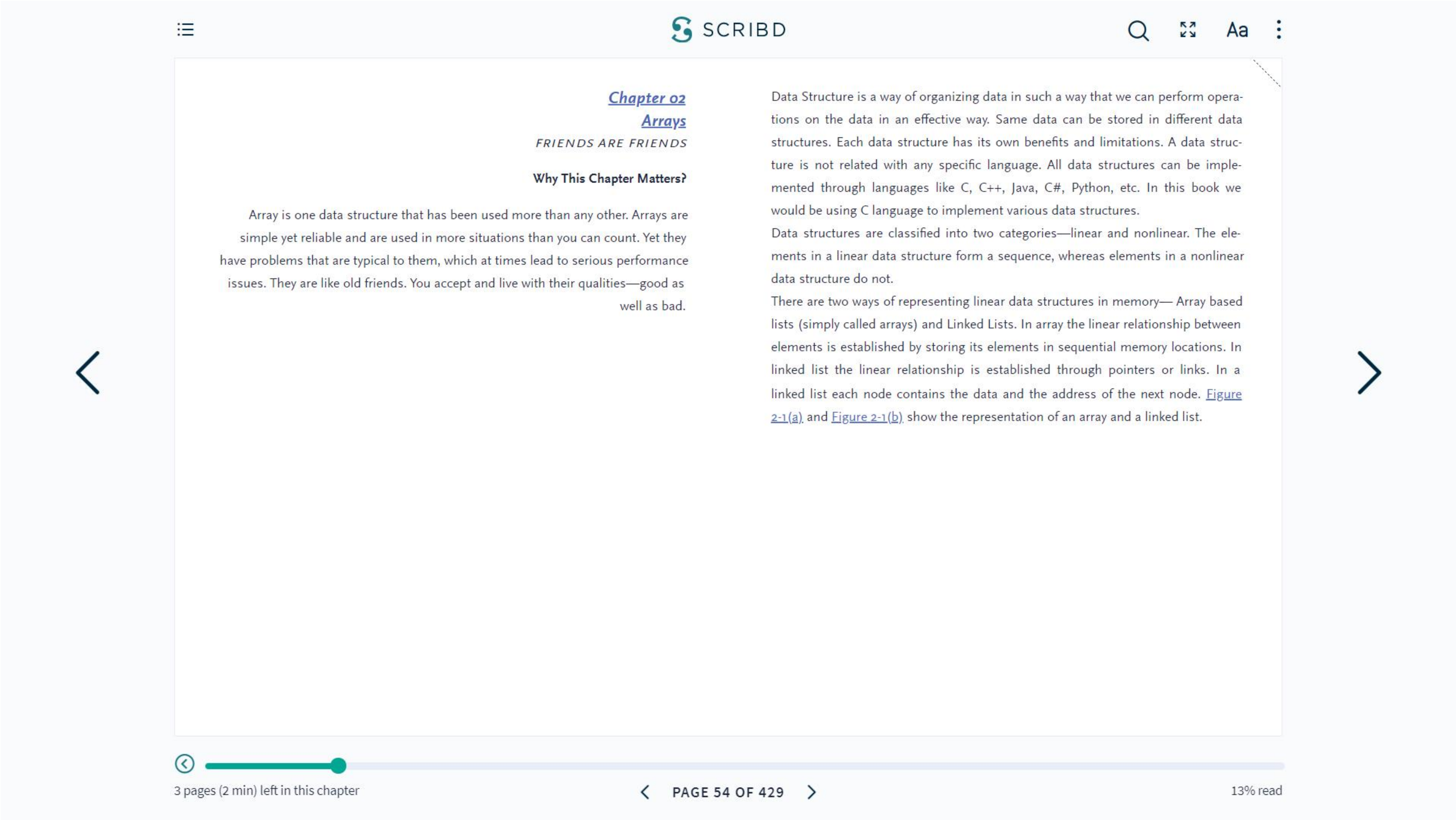


Case Scenario Exercise

Growth rates

List the following functions from highest to lowest order. If any are of the same order, circle them on your list.

2^n	$\log \log n$	$n^3 + \log n$	$\log n$	$n^2 + 5n^3$
2^{n-1}	n^2	n^3	$n \log n$	$(\log n)^2$
$\sqrt{n}$	6	$n!$	n	$(3/2)^n$



Chapter 02
Arrays

FRIENDS ARE FRIENDS

Why This Chapter Matters?

Array is one data structure that has been used more than any other. Arrays are simple yet reliable and are used in more situations than you can count. Yet they have problems that are typical to them, which at times lead to serious performance issues. They are like old friends. You accept and live with their qualities—good as well as bad.

Data Structure is a way of organizing data in such a way that we can perform operations on the data in an effective way. Same data can be stored in different data structures. Each data structure has its own benefits and limitations. A data structure is not related with any specific language. All data structures can be implemented through languages like C, C++, Java, C#, Python, etc. In this book we would be using C language to implement various data structures.

Data structures are classified into two categories—linear and nonlinear. The elements in a linear data structure form a sequence, whereas elements in a nonlinear data structure do not.

There are two ways of representing linear data structures in memory— Array based lists (simply called arrays) and Linked Lists. In array the linear relationship between elements is established by storing its elements in sequential memory locations. In linked list the linear relationship is established through pointers or links. In a linked list each node contains the data and the address of the next node. Figure 2-1(a) and Figure 2-1(b) show the representation of an array and a linked list.

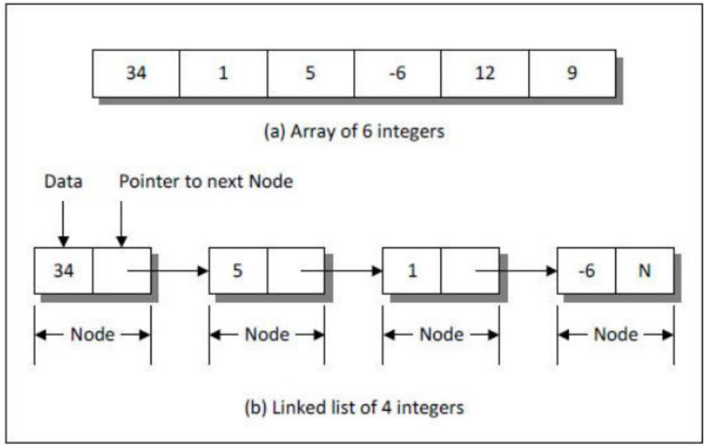
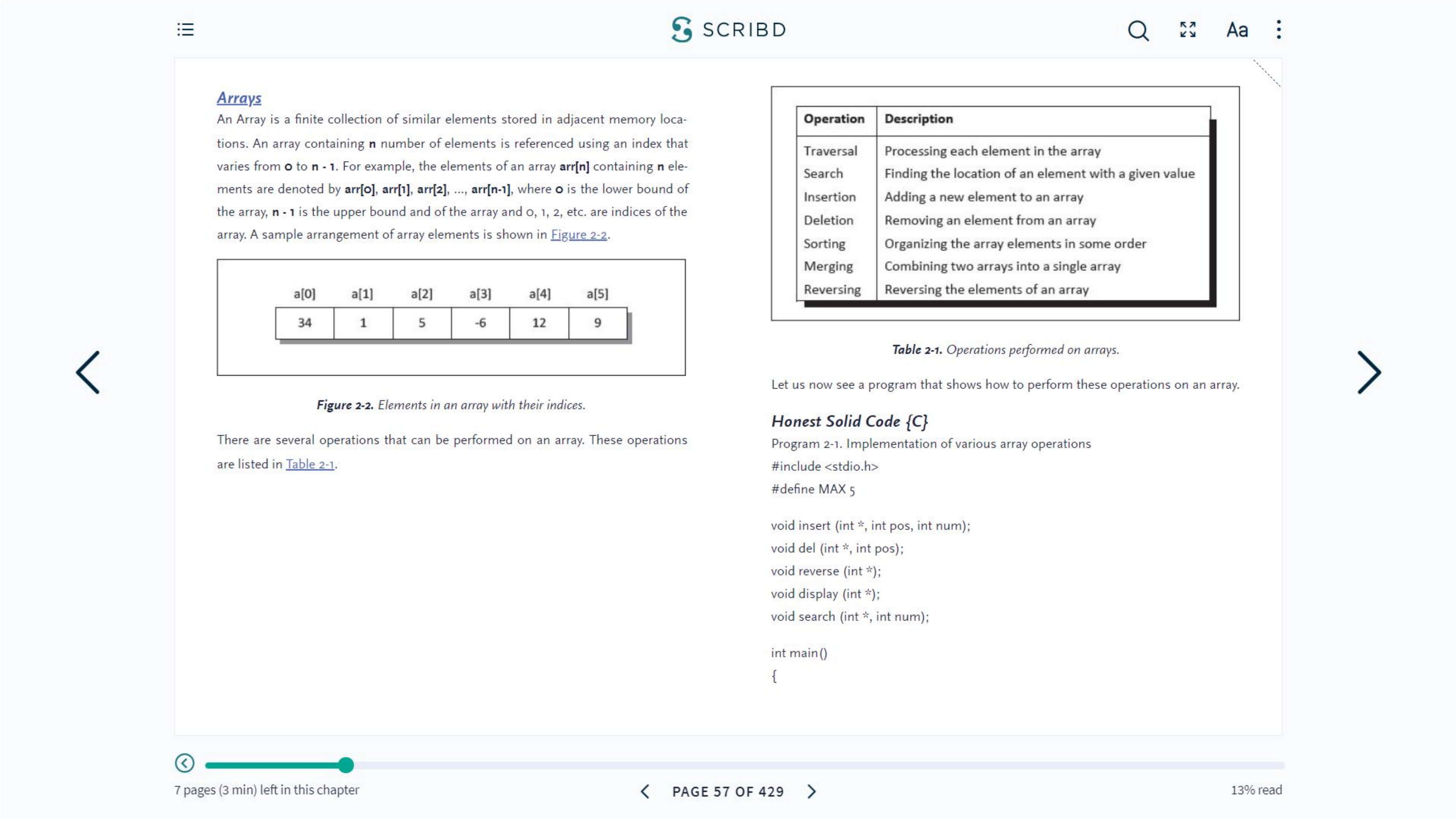


Figure 2-1. *Array and Liked list.*

Arrays are useful when the number of elements to be stored is fixed. They are easy to traverse, search and sort. On the other hand, linked lists are useful when number of data items in the collection is likely to vary. Linked lists are difficult to maintain as compared to an array. We would discuss linked lists in more detail in Chapter 3.



Arrays

An Array is a finite collection of similar elements stored in adjacent memory locations. An array containing **n** number of elements is referenced using an index that varies from **0** to **n - 1**. For example, the elements of an array **arr[n]** containing **n** elements are denoted by **arr[0]**, **arr[1]**, **arr[2]**, ..., **arr[n-1]**, where **0** is the lower bound of the array, **n - 1** is the upper bound and of the array and 0, 1, 2, etc. are indices of the array. A sample arrangement of array elements is shown in Figure 2-2.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
34	1	5	-6	12	9

Figure 2-2. *Elements in an array with their indices.*

There are several operations that can be performed on an array. These operations are listed in Table 2-1.

Operation	Description
Traversal	Processing each element in the array
Search	Finding the location of an element with a given value
Insertion	Adding a new element to an array
Deletion	Removing an element from an array
Sorting	Organizing the array elements in some order
Merging	Combining two arrays into a single array
Reversing	Reversing the elements of an array

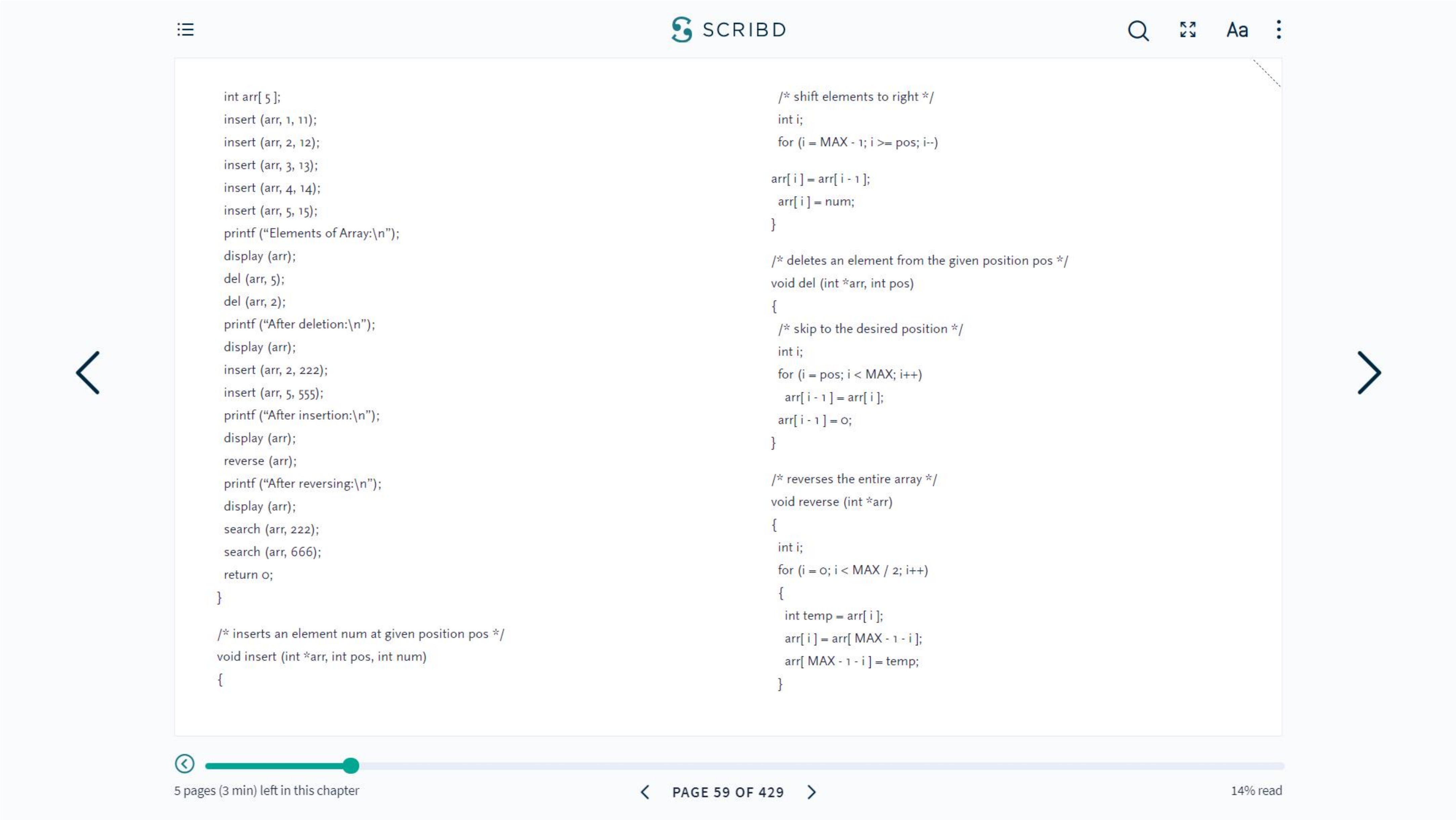
Table 2-1. *Operations performed on arrays.*

Let us now see a program that shows how to perform these operations on an array.

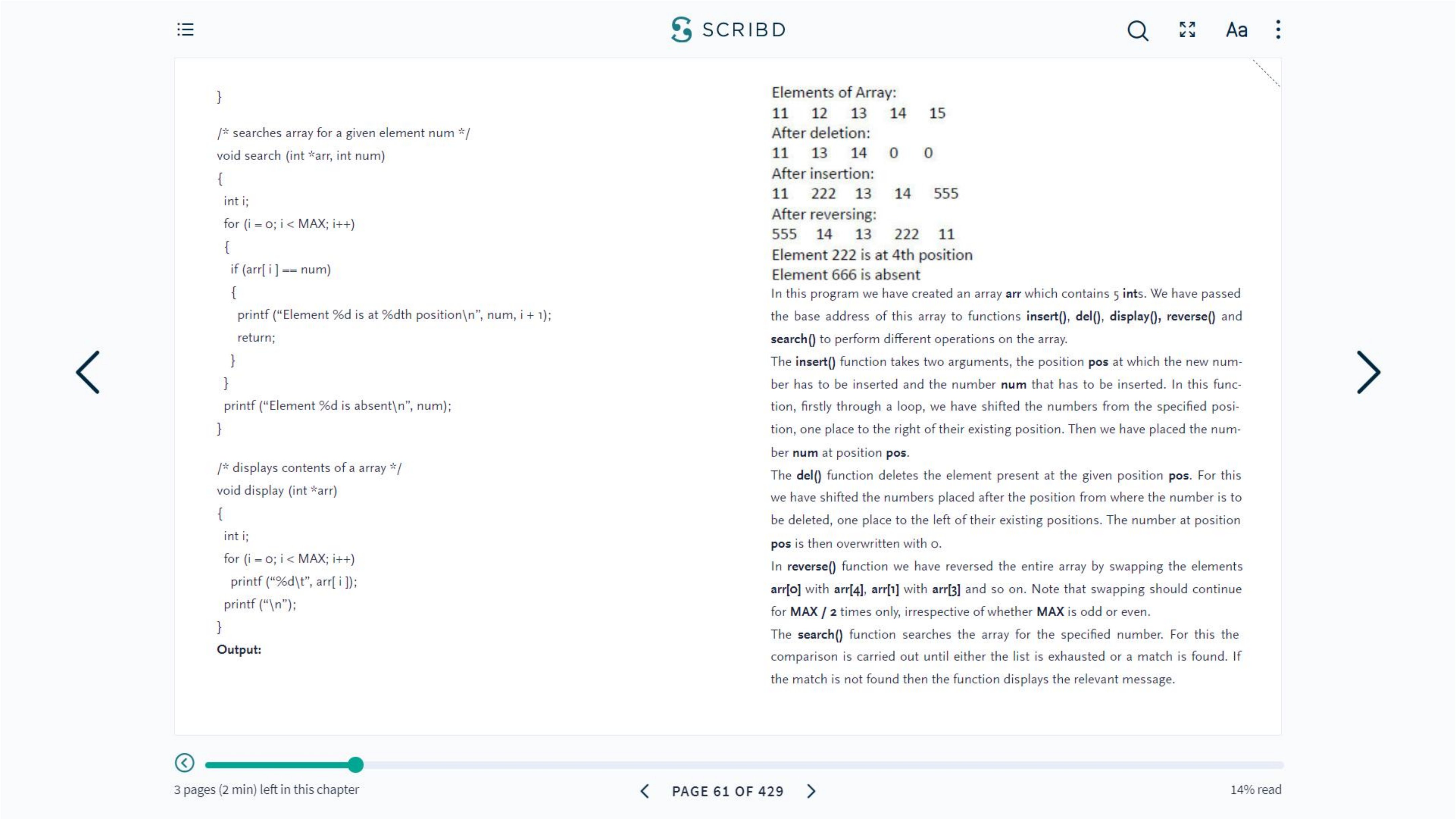
Honest Solid Code {C}

Program 2-1. Implementation of various array operations

```
#include <stdio.h>
#define MAX 5

void insert (int *, int pos, int num);
void del (int *, int pos);
void reverse (int *);
void display (int *);
void search (int *, int num);

int main()
{
```



```
  int arr[ 5 ];
  insert (arr, 1, 11);
  insert (arr, 2, 12);
  insert (arr, 3, 13);
  insert (arr, 4, 14);
  insert (arr, 5, 15);
  printf ("Elements of Array:\n");
  display (arr);
  del (arr, 5);
  del (arr, 2);
  printf ("After deletion:\n");
  display (arr);
  insert (arr, 2, 222);
  insert (arr, 5, 555);
  printf ("After insertion:\n");
  display (arr);
  reverse (arr);
  printf ("After reversing:\n");
  display (arr);
  search (arr, 222);
  search (arr, 666);
  return 0;
}

/* inserts an element num at given position pos */
void insert (int *arr, int pos, int num)
{
  /* shift elements to right */
  int i;
  for (i = MAX - 1; i >= pos; i--)

arr[ i ] = arr[ i - 1 ];
  arr[ i ] = num;
}

/* deletes an element from the given position pos */
void del (int *arr, int pos)
{
  /* skip to the desired position */
  int i;
  for (i = pos; i < MAX; i++)
    arr[ i - 1 ] = arr[ i ];
  arr[ i - 1 ] = 0;
}

/* reverses the entire array */
void reverse (int *arr)
{
  int i;
  for (i = 0; i < MAX / 2; i++)
  {
    int temp = arr[ i ];
    arr[ i ] = arr[ MAX - 1 - i ];
    arr[ MAX - 1 - i ] = temp;
  }
```



```
}

/* searches array for a given element num */
void search (int *arr, int num)
{
 int i;
 for (i = 0; i < MAX; i++)
 {
  if (arr[ i ] == num)
  {
   printf ("Element %d is at %dth position\n", num, i + 1);
   return;
  }
 }
 printf ("Element %d is absent\n", num);
}

/* displays contents of a array */
void display (int *arr)
{
 int i;
 for (i = 0; i < MAX; i++)
  printf ("%d\t", arr[ i ]);
 printf ("\n");
}
```

Output:

```
Elements of Array:
11    12    13    14    15
After deletion:
11    13    14    0    0
After insertion:
11    222    13    14    555
After reversing:
555    14    13    222    11
Element 222 is at 4th position
Element 666 is absent
```

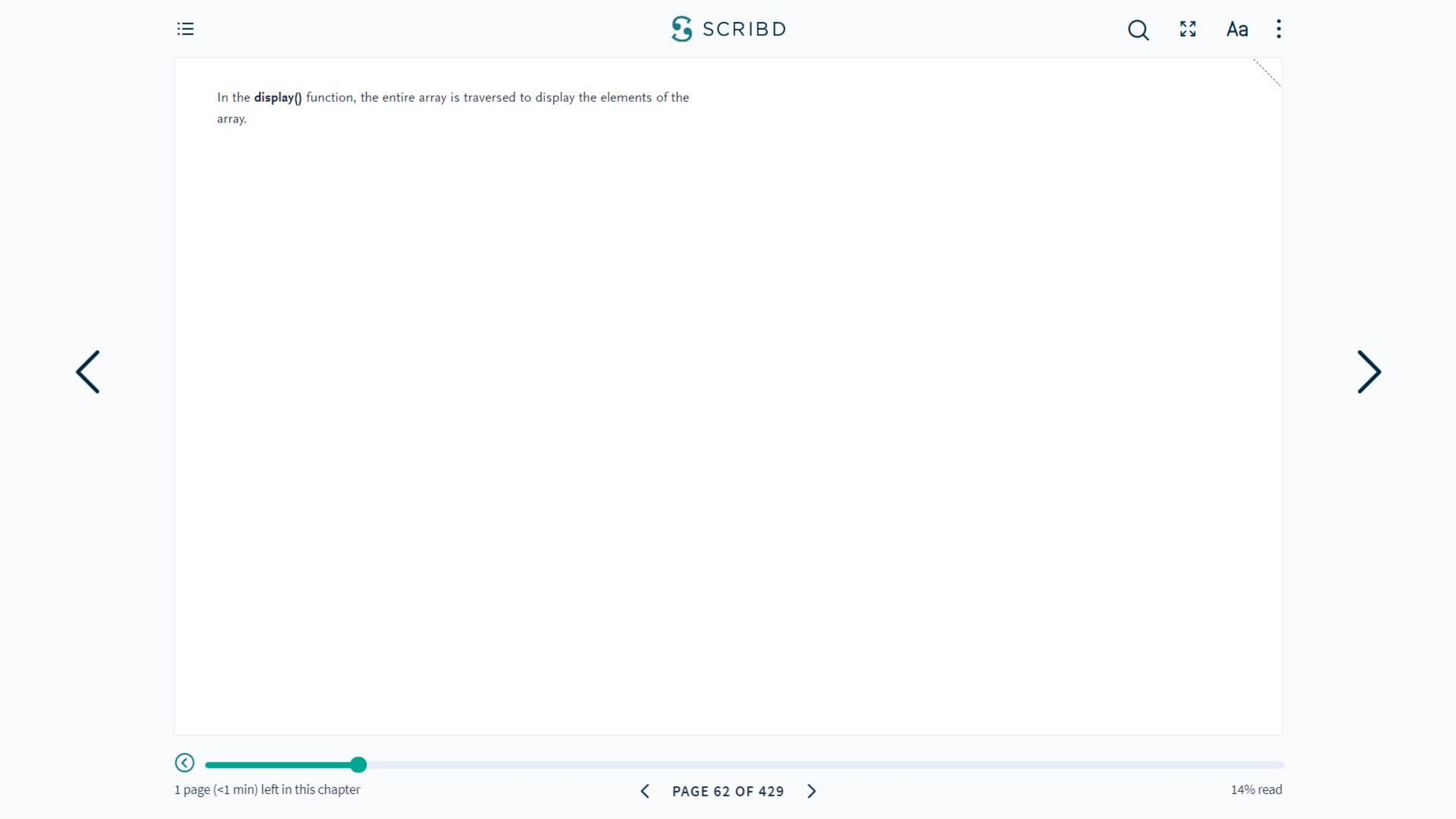
In this program we have created an array **arr** which contains 5 **int**s. We have passed the base address of this array to functions **insert()**, **del()**, **display()**, **reverse()** and **search()** to perform different operations on the array.

The **insert()** function takes two arguments, the position **pos** at which the new number has to be inserted and the number **num** that has to be inserted. In this function, firstly through a loop, we have shifted the numbers from the specified position, one place to the right of their existing position. Then we have placed the number **num** at position **pos**.

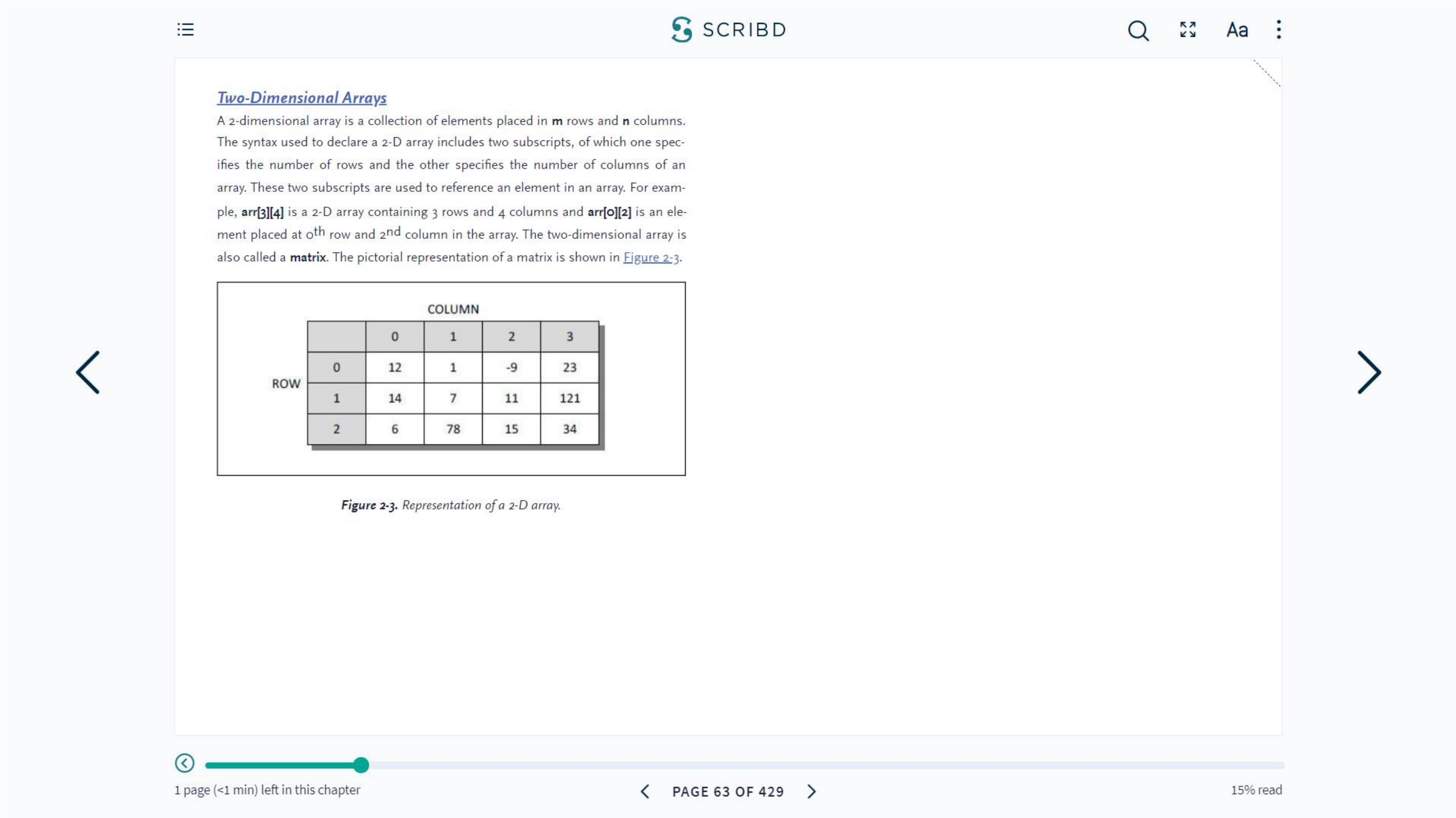
The **del()** function deletes the element present at the given position **pos**. For this we have shifted the numbers placed after the position from where the number is to be deleted, one place to the left of their existing positions. The number at position **pos** is then overwritten with 0.

In **reverse()** function we have reversed the entire array by swapping the elements **arr[0]** with **arr[4]**, **arr[1]** with **arr[3]** and so on. Note that swapping should continue for **MAX / 2** times only, irrespective of whether **MAX** is odd or even.

The **search()** function searches the array for the specified number. For this the comparison is carried out until either the list is exhausted or a match is found. If the match is not found then the function displays the relevant message.



In the **display()** function, the entire array is traversed to display the elements of the array.



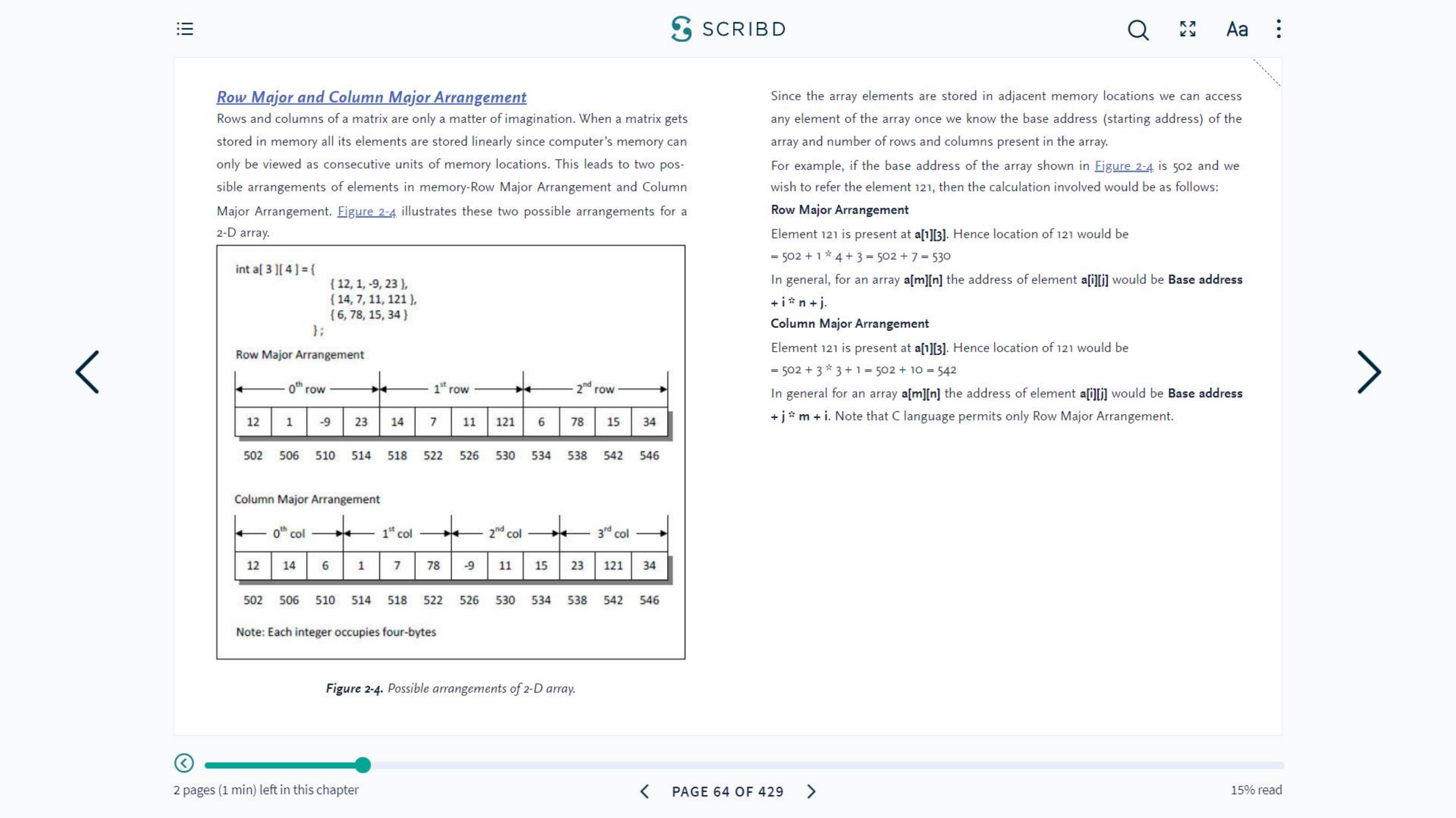
Two-Dimensional Arrays

A 2-dimensional array is a collection of elements placed in **m** rows and **n** columns. The syntax used to declare a 2-D array includes two subscripts, of which one specifies the number of rows and the other specifies the number of columns of an array. These two subscripts are used to reference an element in an array. For example, **arr[3][4]** is a 2-D array containing 3 rows and 4 columns and **arr[0][2]** is an element placed at 0th row and 2nd column in the array. The two-dimensional array is also called a **matrix**. The pictorial representation of a matrix is shown in Figure 2-3.

COLUMN

		0	1	2	3
	0	12	1	-9	23
ROW	1	14	7	11	121
	2	6	78	15	34

Figure 2-3. *Representation of a 2-D array.*



Row Major and Column Major Arrangement

Rows and columns of a matrix are only a matter of imagination. When a matrix gets stored in memory all its elements are stored linearly since computer's memory can only be viewed as consecutive units of memory locations. This leads to two possible arrangements of elements in memory-Row Major Arrangement and Column Major Arrangement. Figure 2-4 illustrates these two possible arrangements for a 2-D array.

```
int a[ 3 ][ 4 ] = {
                    { 12, 1, -9, 23 },
                    { 14, 7, 11, 121 },
                    { 6, 78, 15, 34 }
                  };
```

Row Major Arrangement

0th row				1st row				2nd row			
12	1	-9	23	14	7	11	121	6	78	15	34
502	506	510	514	518	522	526	530	534	538	542	546

Column Major Arrangement

0th col			1st col			2nd col			3rd col		
12	14	6	1	7	78	-9	11	15	23	121	34
502	506	510	514	518	522	526	530	534	538	542	546

Note: Each integer occupies four-bytes

Figure 2-4. *Possible arrangements of 2-D array.*

Since the array elements are stored in adjacent memory locations we can access any element of the array once we know the base address (starting address) of the array and number of rows and columns present in the array.

For example, if the base address of the array shown in Figure 2-4 is 502 and we wish to refer the element 121, then the calculation involved would be as follows:

Row Major Arrangement

Element 121 is present at **a[1][3]**. Hence location of 121 would be

$= 502 + 1 * 4 + 3 = 502 + 7 = 530$

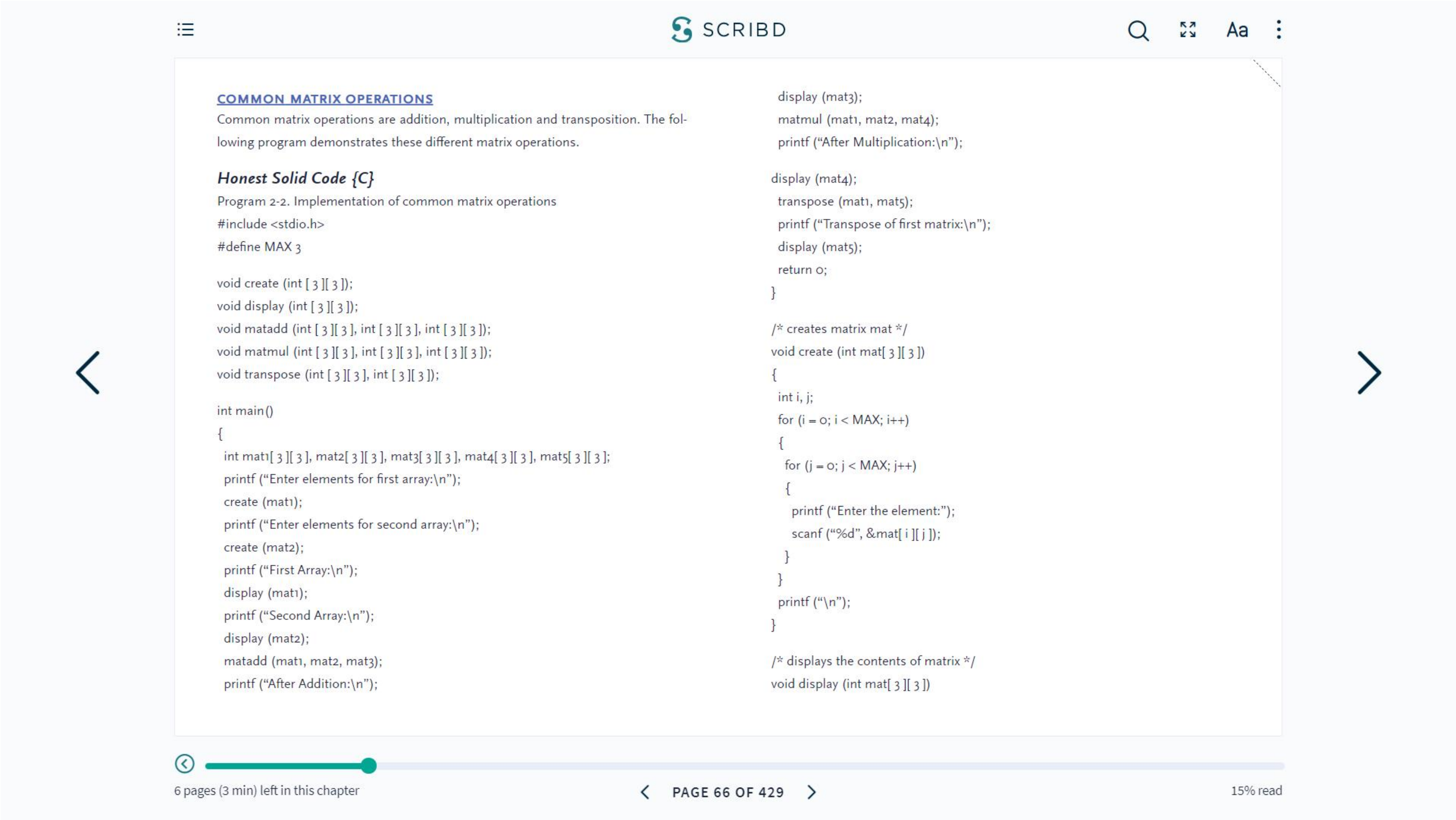
In general, for an array **a[m][n]** the address of element **a[i][j]** would be **Base address + i * n + j**.

Column Major Arrangement

Element 121 is present at **a[1][3]**. Hence location of 121 would be

$= 502 + 3 * 3 + 1 = 502 + 10 = 542$

In general for an array **a[m][n]** the address of element **a[i][j]** would be **Base address + j * m + i**. Note that C language permits only Row Major Arrangement.



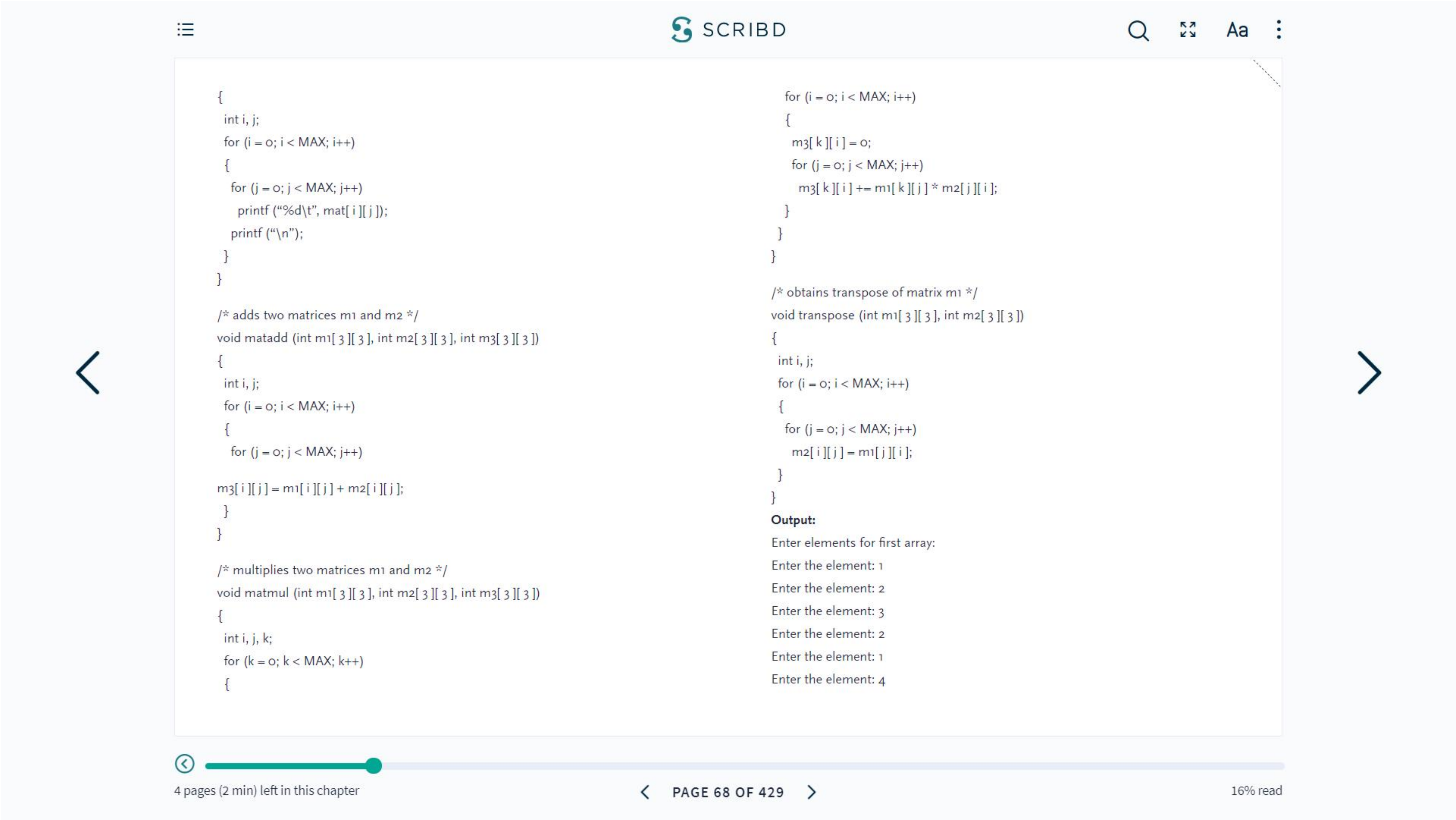
COMMON MATRIX OPERATIONS

Common matrix operations are addition, multiplication and transposition. The following program demonstrates these different matrix operations.

Honest Solid Code {C}

Program 2-2. Implementation of common matrix operations

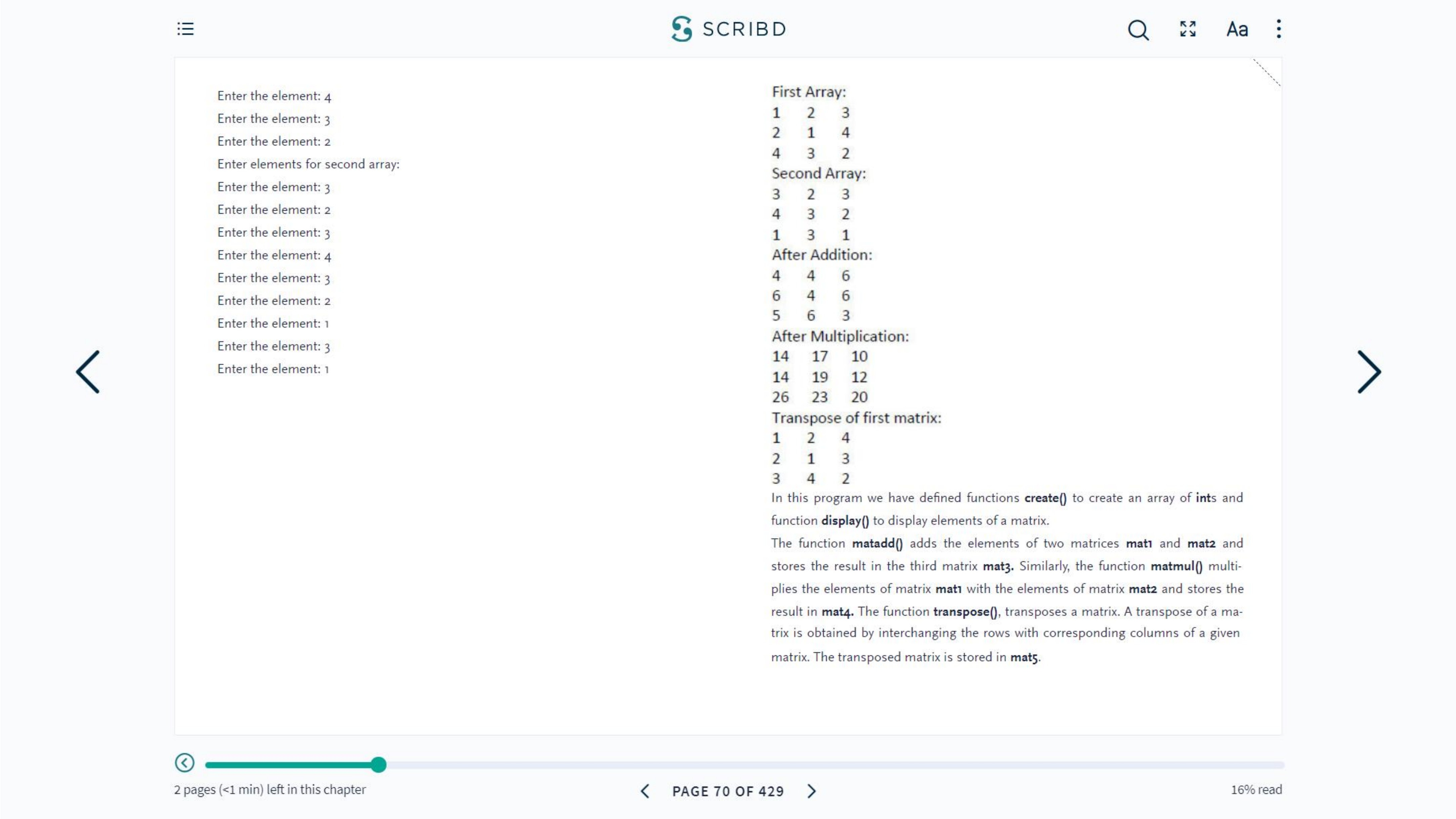
```
#include <stdio.h>
#define MAX 3

void create (int [ 3 ][ 3 ]);
void display (int [ 3 ][ 3 ]);
void matadd (int [ 3 ][ 3 ], int [ 3 ][ 3 ], int [ 3 ][ 3 ]);
void matmul (int [ 3 ][ 3 ], int [ 3 ][ 3 ], int [ 3 ][ 3 ]);
void transpose (int [ 3 ][ 3 ], int [ 3 ][ 3 ]);

int main()
{
 int mat1[ 3 ][ 3 ], mat2[ 3 ][ 3 ], mat3[ 3 ][ 3 ], mat4[ 3 ][ 3 ], mat5[ 3 ][ 3 ];
 printf (“Enter elements for first array:\n”);
 create (mat1);
 printf (“Enter elements for second array:\n”);
 create (mat2);
 printf (“First Array:\n”);
 display (mat1);
 printf (“Second Array:\n”);
 display (mat2);
 matadd (mat1, mat2, mat3);
 printf (“After Addition:\n”);
 display (mat3);
 matmul (mat1, mat2, mat4);
 printf (“After Multiplication:\n”);

display (mat4);
 transpose (mat1, mat5);
 printf (“Transpose of first matrix:\n”);
 display (mat5);
 return 0;
}

/* creates matrix mat */
void create (int mat[ 3 ][ 3 ])
{
 int i, j;
 for (i = 0; i < MAX; i++)
 {
  for (j = 0; j < MAX; j++)
  {
   printf (“Enter the element:”);
   scanf (“%d”, &mat[ i ][ j ]);
  }
 }
 printf (“\n”);
}

/* displays the contents of matrix */
void display (int mat[ 3 ][ 3 ])
```



```
{
 int i, j;
 for (i = 0; i < MAX; i++)
 {
  for (j = 0; j < MAX; j++)
   printf (“%d\t”, mat[ i ][ j ]);
  printf (“\n”);
 }
}

/* adds two matrices m1 and m2 */
void matadd (int m1[ 3 ][ 3 ], int m2[ 3 ][ 3 ], int m3[ 3 ][ 3 ])
{
 int i, j;
 for (i = 0; i < MAX; i++)
 {
  for (j = 0; j < MAX; j++)

m3[ i ][ j ] = m1[ i ][ j ] + m2[ i ][ j ];
 }
}

/* multiplies two matrices m1 and m2 */
void matmul (int m1[ 3 ][ 3 ], int m2[ 3 ][ 3 ], int m3[ 3 ][ 3 ])
{
 int i, j, k;
 for (k = 0; k < MAX; k++)
 {
  for (i = 0; i < MAX; i++)
  {
   m3[ k ][ i ] = 0;
   for (j = 0; j < MAX; j++)
    m3[ k ][ i ] += m1[ k ][ j ] * m2[ j ][ i ];
  }
 }
}

/* obtains transpose of matrix m1 */
void transpose (int m1[ 3 ][ 3 ], int m2[ 3 ][ 3 ])
{
 int i, j;
 for (i = 0; i < MAX; i++)
 {
  for (j = 0; j < MAX; j++)
   m2[ i ][ j ] = m1[ j ][ i ];
 }
}
```

Output:

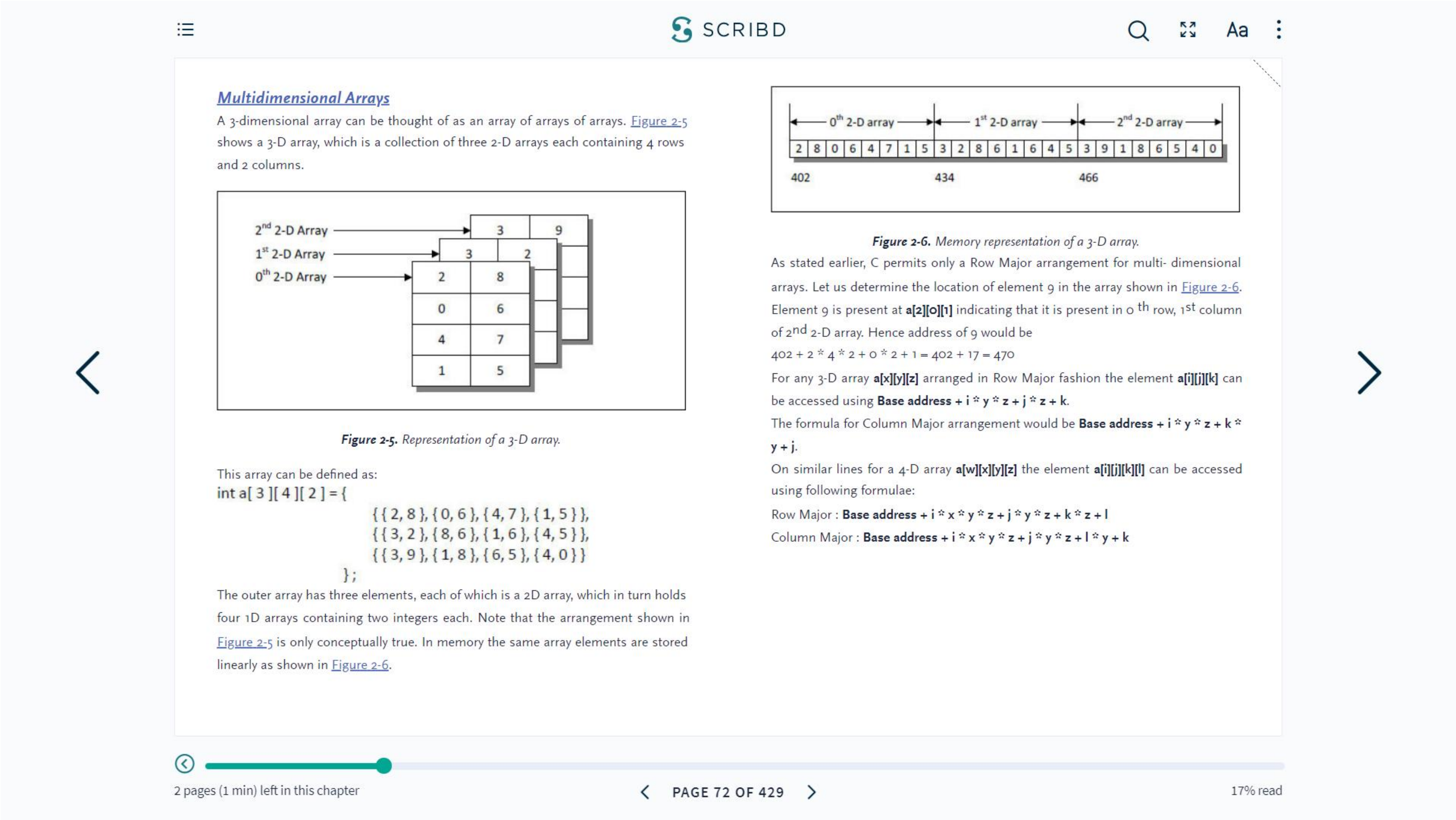
```
Enter elements for first array:
Enter the element: 1
Enter the element: 2
Enter the element: 3
Enter the element: 2
Enter the element: 1
Enter the element: 4
```



```
Enter the element: 4
Enter the element: 3
Enter the element: 2
Enter elements for second array:
Enter the element: 3
Enter the element: 2
Enter the element: 3
Enter the element: 4
Enter the element: 3
Enter the element: 2
Enter the element: 1
Enter the element: 3
Enter the element: 1
First Array:
1    2    3
2    1    4
4    3    2
Second Array:
3    2    3
4    3    2
1    3    1
After Addition:
4    4    6
6    4    6
5    6    3
After Multiplication:
14    17    10
14    19    12
26    23    20
Transpose of first matrix:
1    2    4
2    1    3
3    4    2
```

In this program we have defined functions **create()** to create an array of **int**s and function **display()** to display elements of a matrix.

The function **matadd()** adds the elements of two matrices **mat1** and **mat2** and stores the result in the third matrix **mat3.** Similarly, the function **matmul()** multiplies the elements of matrix **mat1** with the elements of matrix **mat2** and stores the result in **mat4.** The function **transpose()**, transposes a matrix. A transpose of a matrix is obtained by interchanging the rows with corresponding columns of a given matrix. The transposed matrix is stored in **mat5**.



Multidimensional Arrays

A 3-dimensional array can be thought of as an array of arrays of arrays. Figure 2-5 shows a 3-D array, which is a collection of three 2-D arrays each containing 4 rows and 2 columns.

Figure 2-5. *Representation of a 3-D array.*

This array can be defined as:

```
int a[ 3 ][ 4 ][ 2 ] = {
                    { { 2, 8 }, { 0, 6 }, { 4, 7 }, { 1, 5 } },
                    { { 3, 2 }, { 8, 6 }, { 1, 6 }, { 4, 5 } },
                    { { 3, 9 }, { 1, 8 }, { 6, 5 }, { 4, 0 } }
               } ;
```

The outer array has three elements, each of which is a 2D array, which in turn holds four 1D arrays containing two integers each. Note that the arrangement shown in Figure 2-5 is only conceptually true. In memory the same array elements are stored linearly as shown in Figure 2-6.

Figure 2-6. *Memory representation of a 3-D array.*

As stated earlier, C permits only a Row Major arrangement for multi- dimensional arrays. Let us determine the location of element 9 in the array shown in Figure 2-6. Element 9 is present at **a[2][0][1]** indicating that it is present in 0 th row, 1st column of 2nd 2-D array. Hence address of 9 would be

402 + 2 * 4 * 2 + 0 * 2 + 1 = 402 + 17 = 470

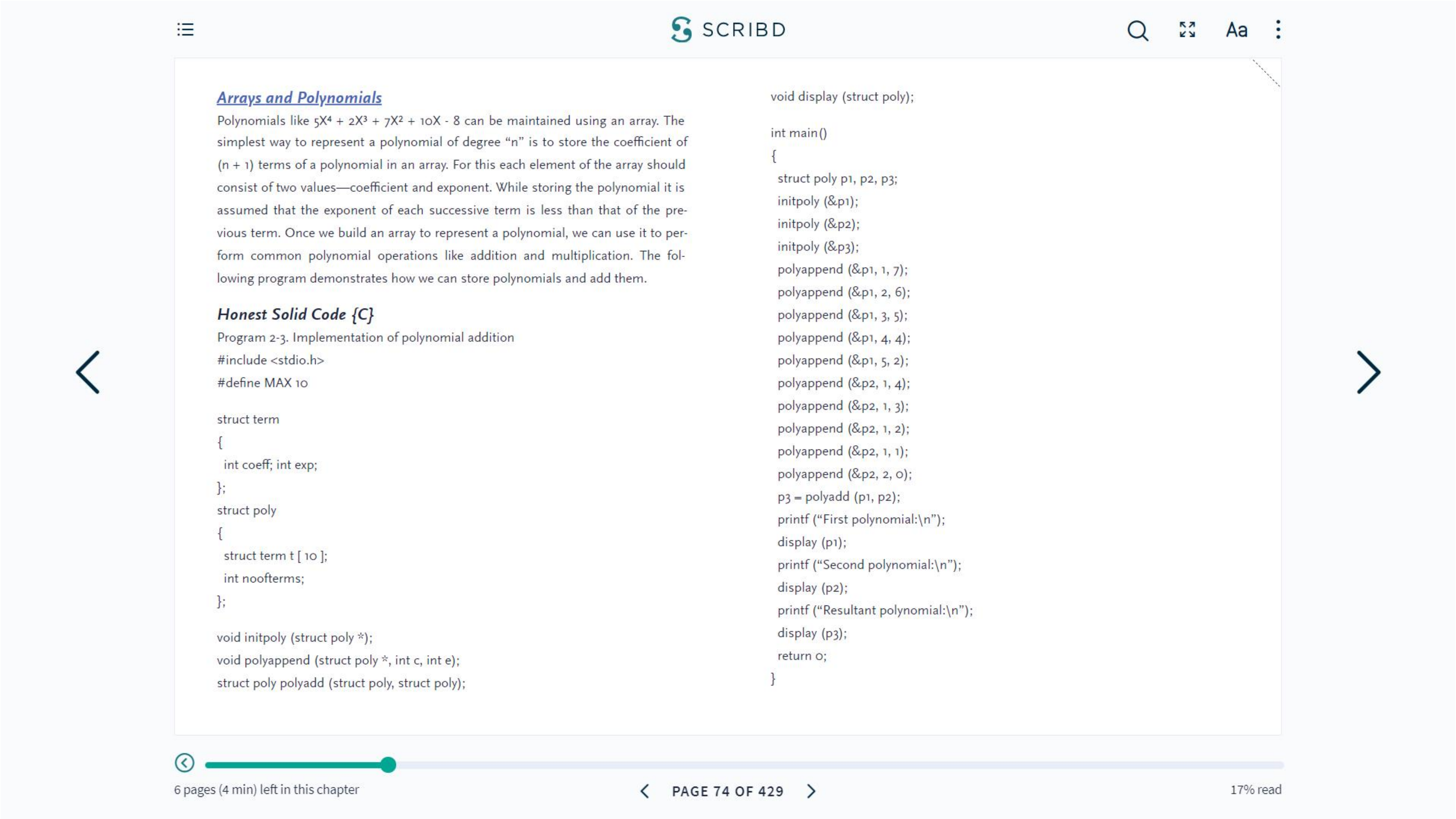
For any 3-D array **a[x][y][z]** arranged in Row Major fashion the element **a[i][j][k]** can be accessed using **Base address + i * y * z + j * z + k**.

The formula for Column Major arrangement would be **Base address + i * y * z + k * y + j**.

On similar lines for a 4-D array **a[w][x][y][z]** the element **a[i][j][k][l]** can be accessed using following formulae:

Row Major : **Base address + i * x * y * z + j * y * z + k * z + l**

Column Major : **Base address + i * x * y * z + j * y * z + l * y + k**



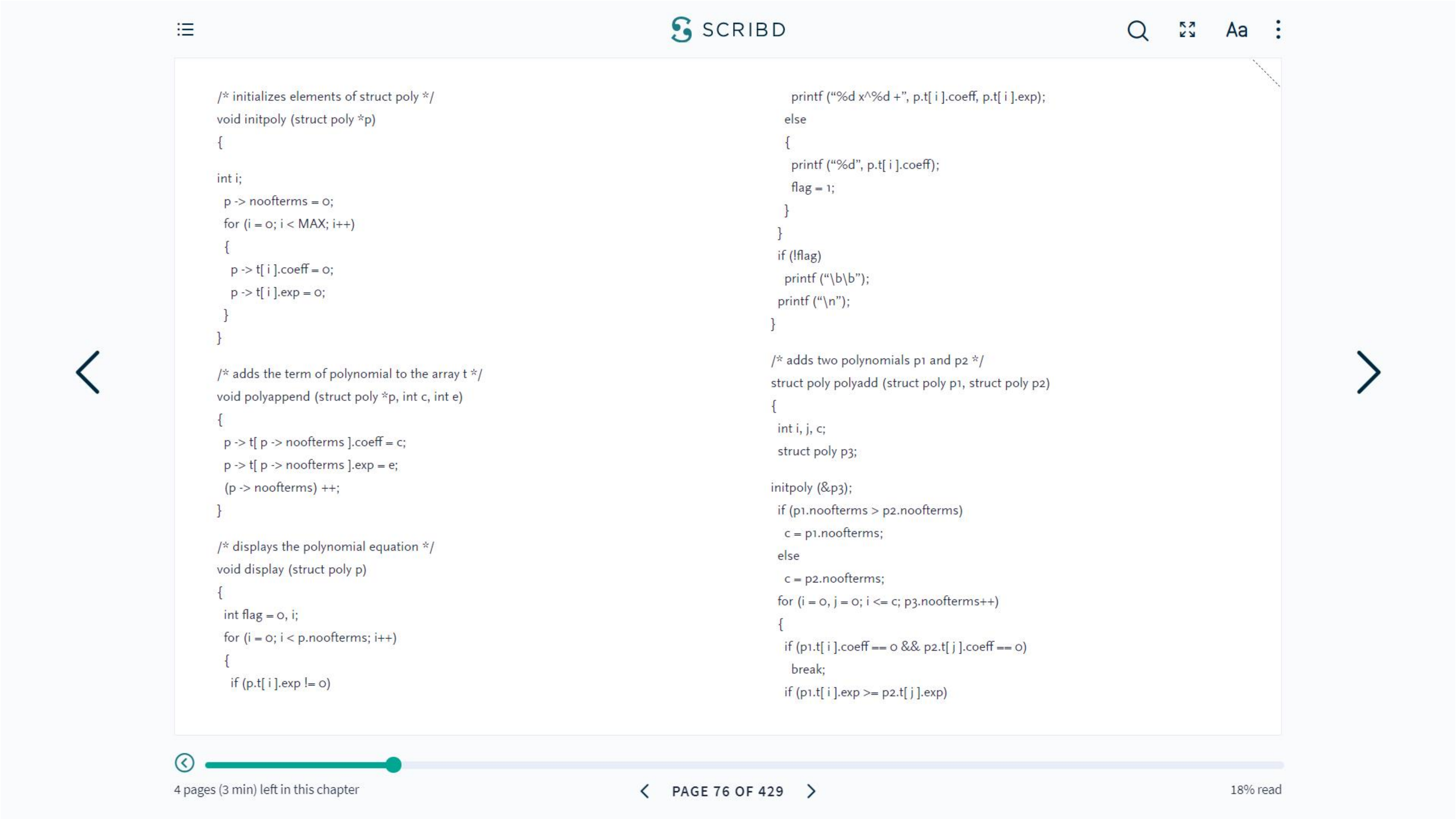
Arrays and Polynomials

Polynomials like $5X^4 + 2X^3 + 7X^2 + 10X - 8$ can be maintained using an array. The simplest way to represent a polynomial of degree “n” is to store the coefficient of (n + 1) terms of a polynomial in an array. For this each element of the array should consist of two values—coefficient and exponent. While storing the polynomial it is assumed that the exponent of each successive term is less than that of the previous term. Once we build an array to represent a polynomial, we can use it to perform common polynomial operations like addition and multiplication. The following program demonstrates how we can store polynomials and add them.

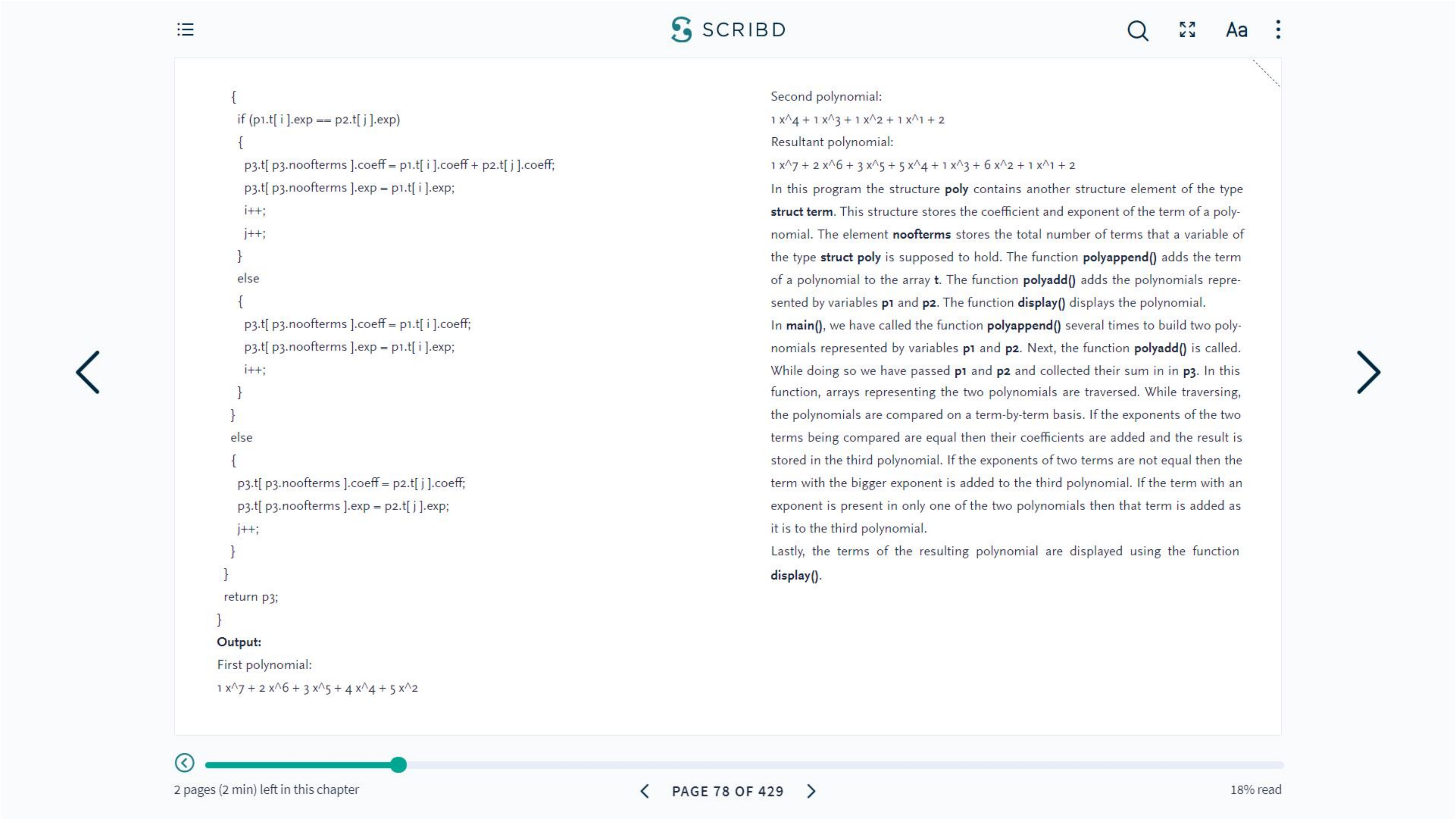
Honest Solid Code {C}

Program 2-3. Implementation of polynomial addition

```
#include <stdio.h>
#define MAX 10

struct term
{
 int coeff; int exp;
};
struct poly
{
 struct term t [ 10 ];
 int noofterms;
};

void initpoly (struct poly *);
void polyappend (struct poly *, int c, int e);
struct poly polyadd (struct poly, struct poly);
void display (struct poly);

int main()
{
 struct poly p1, p2, p3;
 initpoly (&p1);
 initpoly (&p2);
 initpoly (&p3);
 polyappend (&p1, 1, 7);
 polyappend (&p1, 2, 6);
 polyappend (&p1, 3, 5);
 polyappend (&p1, 4, 4);
 polyappend (&p1, 5, 2);
 polyappend (&p2, 1, 4);
 polyappend (&p2, 1, 3);
 polyappend (&p2, 1, 2);
 polyappend (&p2, 1, 1);
 polyappend (&p2, 2, 0);
 p3 = polyadd (p1, p2);
 printf (“First polynomial:\n”);
 display (p1);
 printf (“Second polynomial:\n”);
 display (p2);
 printf (“Resultant polynomial:\n”);
 display (p3);
 return 0;
}
```



```
/* initializes elements of struct poly */
void initpoly (struct poly *p)
{

int i;
 p -> noofterms = 0;
 for (i = 0; i < MAX; i++)
 {
  p -> t[ i ].coeff = 0;
  p -> t[ i ].exp = 0;
 }
}

/* adds the term of polynomial to the array t */
void polyappend (struct poly *p, int c, int e)
{
 p -> t[ p -> noofterms ].coeff = c;
 p -> t[ p -> noofterms ].exp = e;
 (p -> noofterms) ++;
}

/* displays the polynomial equation */
void display (struct poly p)
{
 int flag = 0, i;
 for (i = 0; i < p.noofterms; i++)
 {
  if (p.t[ i ].exp != 0)
   printf (“%d x^%d +”, p.t[ i ].coeff, p.t[ i ].exp);
  else
  {
   printf (“%d”, p.t[ i ].coeff);
   flag = 1;
  }
 }
 if (!flag)
  printf (“\b\b”);
 printf (“\n”);
}

/* adds two polynomials p1 and p2 */
struct poly polyadd (struct poly p1, struct poly p2)
{
 int i, j, c;
 struct poly p3;

initpoly (&p3);
 if (p1.noofterms > p2.noofterms)
  c = p1.noofterms;
 else
  c = p2.noofterms;
 for (i = 0, j = 0; i <= c; p3.noofterms++)
 {
  if (p1.t[ i ].coeff == 0 && p2.t[ j ].coeff == 0)
   break;
  if (p1.t[ i ].exp >= p2.t[ j ].exp)
```



```
    {
      if (p1.t[ i ].exp == p2.t[ j ].exp)
      {
        p3.t[ p3.noofterms ].coeff = p1.t[ i ].coeff + p2.t[ j ].coeff;
        p3.t[ p3.noofterms ].exp = p1.t[ i ].exp;
        i++;
        j++;
      }
      else
      {
        p3.t[ p3.noofterms ].coeff = p1.t[ i ].coeff;
        p3.t[ p3.noofterms ].exp = p1.t[ i ].exp;
        i++;
      }
    }
    else
    {
      p3.t[ p3.noofterms ].coeff = p2.t[ j ].coeff;
      p3.t[ p3.noofterms ].exp = p2.t[ j ].exp;
      j++;
    }
  }
  return p3;
}
```

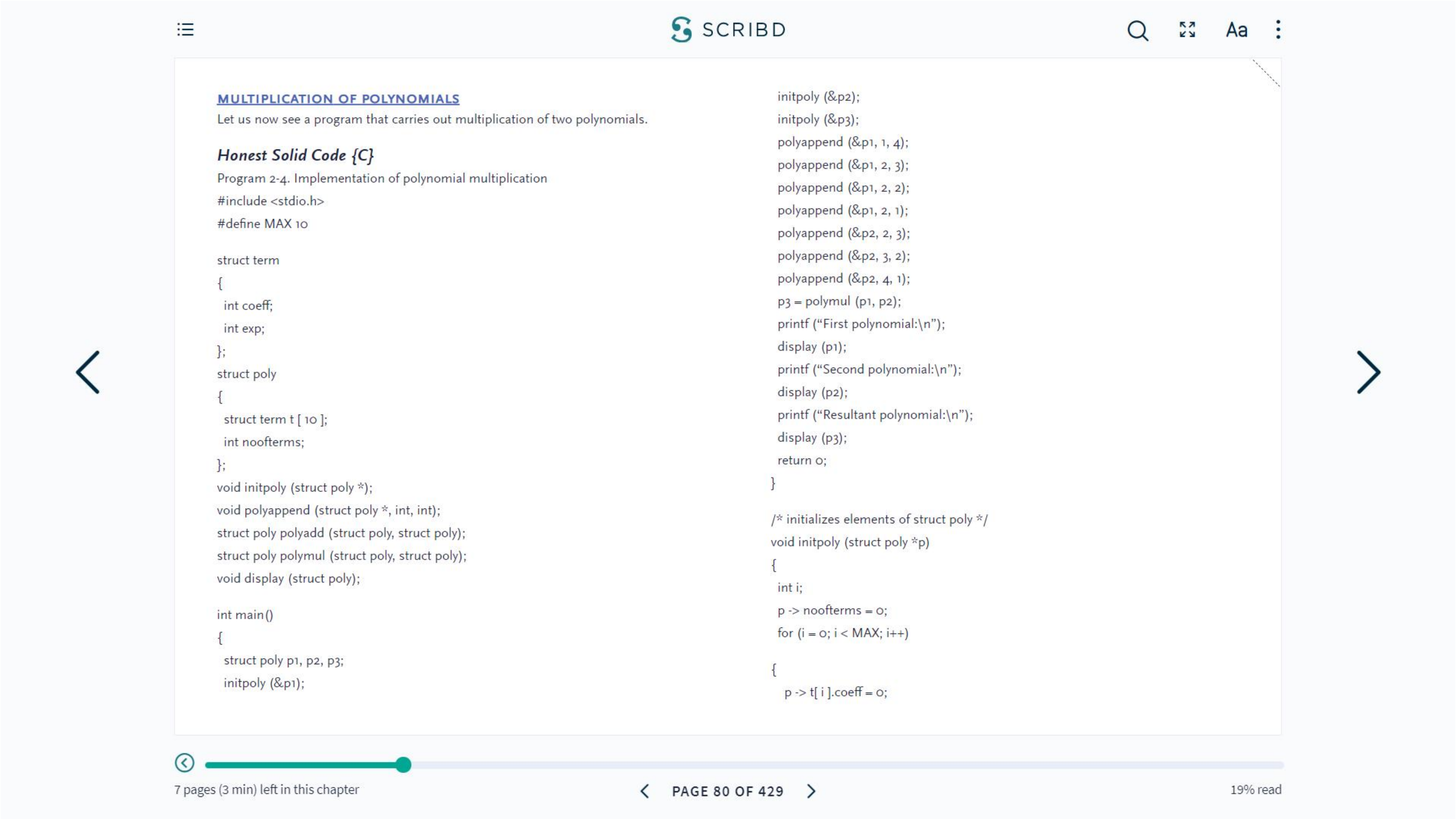
Output:

```
First polynomial:
1 x^7 + 2 x^6 + 3 x^5 + 4 x^4 + 5 x^2
Second polynomial:
1 x^4 + 1 x^3 + 1 x^2 + 1 x^1 + 2
Resultant polynomial:
1 x^7 + 2 x^6 + 3 x^5 + 5 x^4 + 1 x^3 + 6 x^2 + 1 x^1 + 2
```

In this program the structure **poly** contains another structure element of the type **struct term**. This structure stores the coefficient and exponent of the term of a polynomial. The element **noofterms** stores the total number of terms that a variable of the type **struct poly** is supposed to hold. The function **polyappend()** adds the term of a polynomial to the array **t**. The function **polyadd()** adds the polynomials represented by variables **p1** and **p2**. The function **display()** displays the polynomial.

In **main()**, we have called the function **polyappend()** several times to build two polynomials represented by variables **p1** and **p2**. Next, the function **polyadd()** is called. While doing so we have passed **p1** and **p2** and collected their sum in in **p3**. In this function, arrays representing the two polynomials are traversed. While traversing, the polynomials are compared on a term-by-term basis. If the exponents of the two terms being compared are equal then their coefficients are added and the result is stored in the third polynomial. If the exponents of two terms are not equal then the term with the bigger exponent is added to the third polynomial. If the term with an exponent is present in only one of the two polynomials then that term is added as it is to the third polynomial.

Lastly, the terms of the resulting polynomial are displayed using the function **display()**.



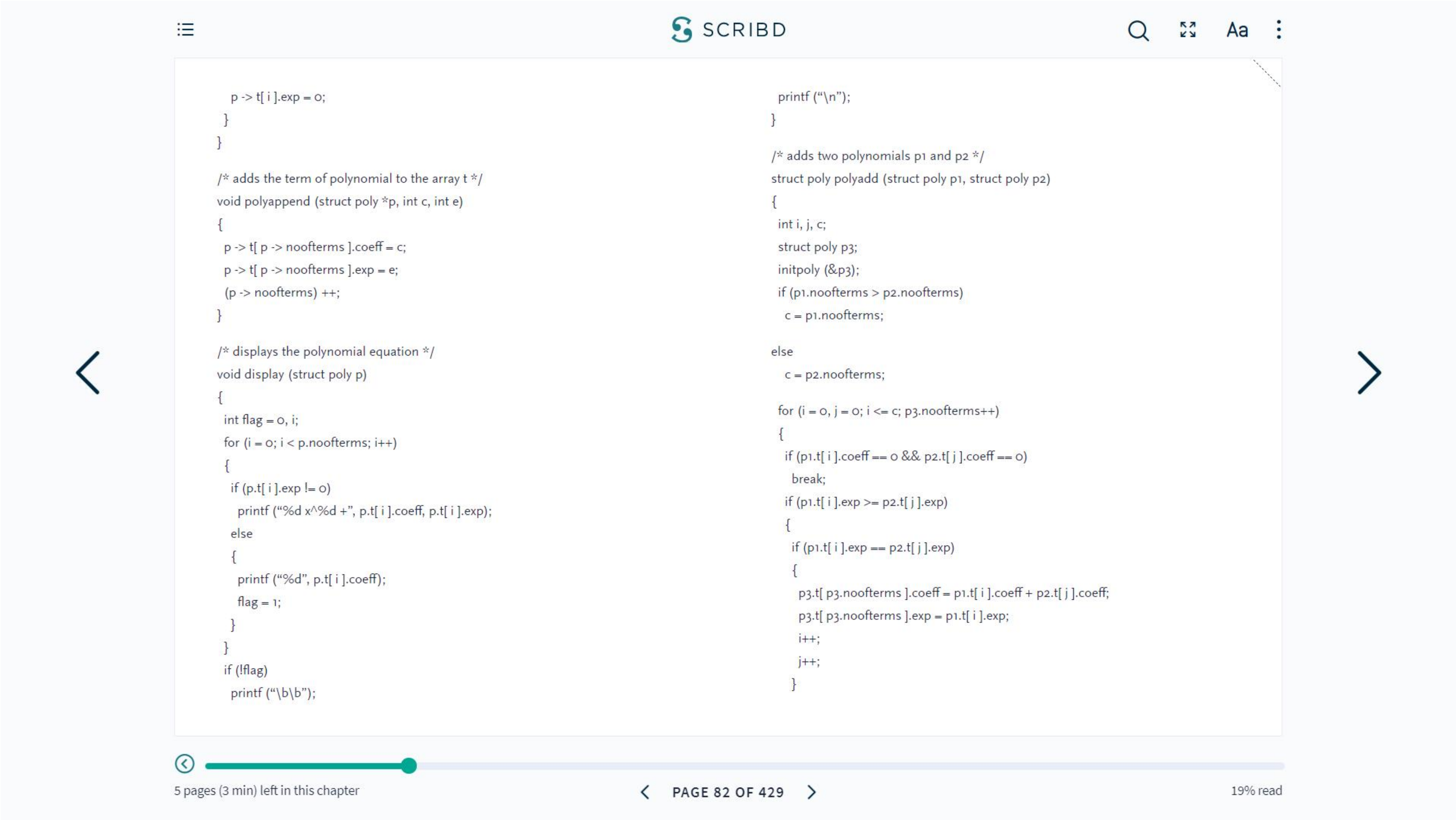
MULTIPLICATION OF POLYNOMIALS

Let us now see a program that carries out multiplication of two polynomials.

Honest Solid Code {C}

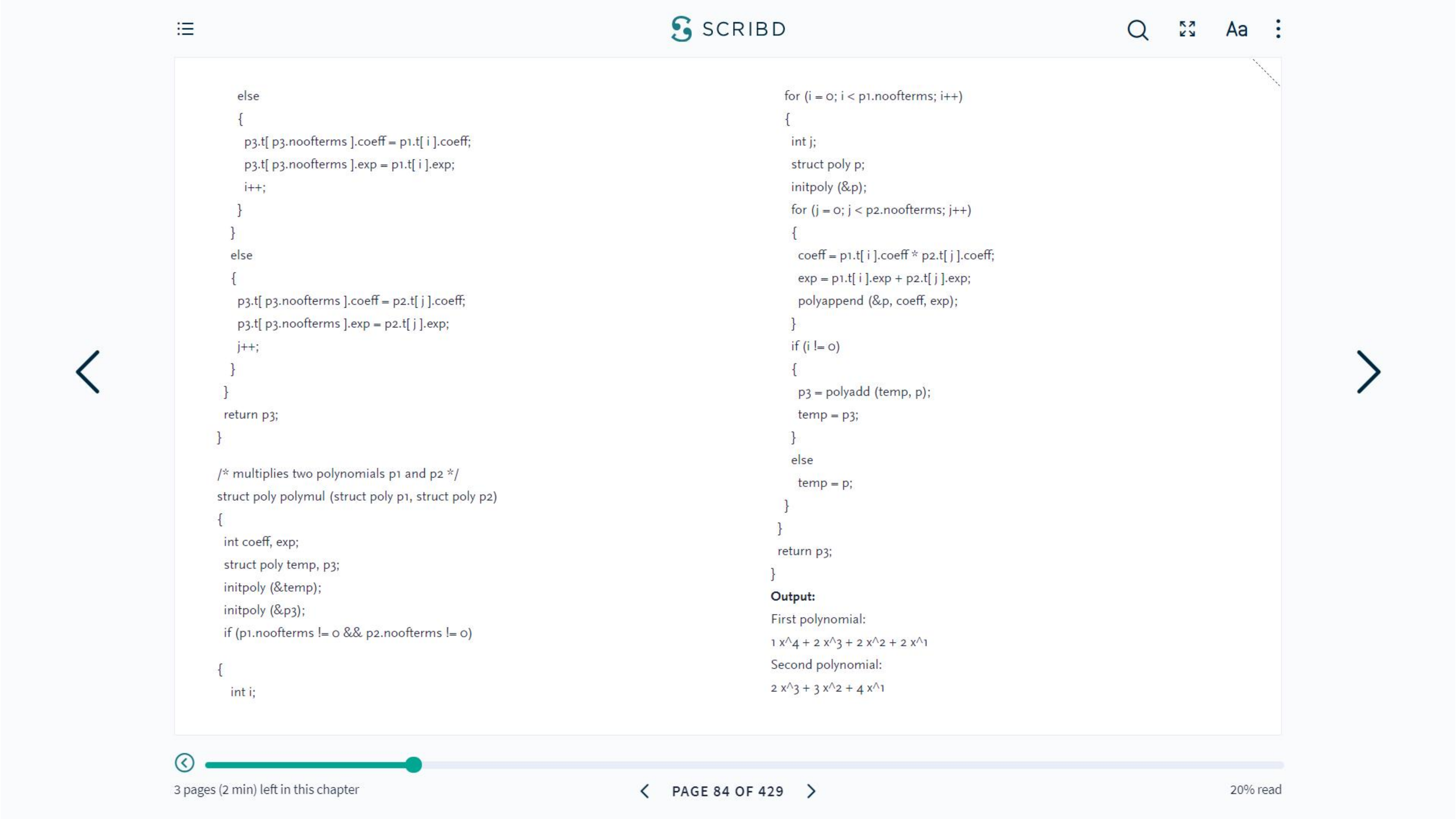
Program 2-4. Implementation of polynomial multiplication

```
#include <stdio.h>
#define MAX 10

struct term
{
  int coeff;
  int exp;
};
struct poly
{
  struct term t [ 10 ];
  int noofterms;
};
void initpoly (struct poly *);
void polyappend (struct poly *, int, int);
struct poly polyadd (struct poly, struct poly);
struct poly polymul (struct poly, struct poly);
void display (struct poly);

int main()
{
  struct poly p1, p2, p3;
  initpoly (&p1);
  initpoly (&p2);
  initpoly (&p3);
  polyappend (&p1, 1, 4);
  polyappend (&p1, 2, 3);
  polyappend (&p1, 2, 2);
  polyappend (&p1, 2, 1);
  polyappend (&p2, 2, 3);
  polyappend (&p2, 3, 2);
  polyappend (&p2, 4, 1);
  p3 = polymul (p1, p2);
  printf (“First polynomial:\n”);
  display (p1);
  printf (“Second polynomial:\n”);
  display (p2);
  printf (“Resultant polynomial:\n”);
  display (p3);
  return 0;
}

/* initializes elements of struct poly */
void initpoly (struct poly *p)
{
  int i;
  p -> noofterms = 0;
  for (i = 0; i < MAX; i++)

{
   p -> t[ i ].coeff = 0;
```



```
    p -> t[ i ].exp = 0;
  }
}

/* adds the term of polynomial to the array t */
void polyappend (struct poly *p, int c, int e)
{
  p -> t[ p -> noofterms ].coeff = c;
  p -> t[ p -> noofterms ].exp = e;
  (p -> noofterms) ++;
}

/* displays the polynomial equation */
void display (struct poly p)
{
  int flag = 0, i;
  for (i = 0; i < p.noofterms; i++)
  {
    if (p.t[ i ].exp != 0)
      printf (“%d x^%d +”, p.t[ i ].coeff, p.t[ i ].exp);
    else
    {
      printf (“%d”, p.t[ i ].coeff);
      flag = 1;
    }
  }
  if (!flag)
    printf (“\b\b”);
```

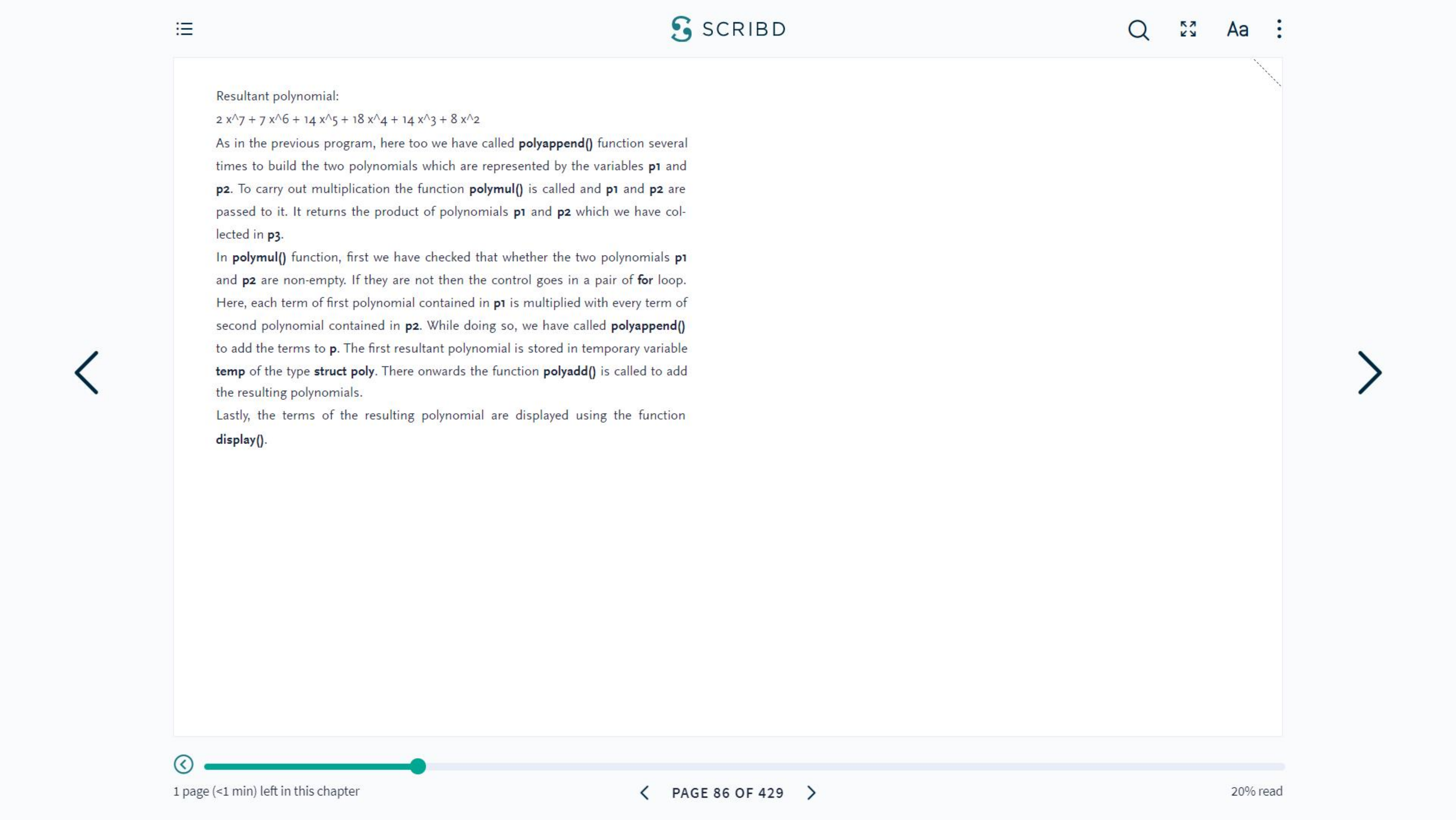
```
  printf (“\n”);
}

/* adds two polynomials p1 and p2 */
struct poly polyadd (struct poly p1, struct poly p2)
{
  int i, j, c;
  struct poly p3;
  initpoly (&p3);
  if (p1.noofterms > p2.noofterms)
    c = p1.noofterms;

else
    c = p2.noofterms;

  for (i = 0, j = 0; i <= c; p3.noofterms++)
  {
    if (p1.t[ i ].coeff == 0 && p2.t[ j ].coeff == 0)
      break;
    if (p1.t[ i ].exp >= p2.t[ j ].exp)
    {
      if (p1.t[ i ].exp == p2.t[ j ].exp)
      {
        p3.t[ p3.noofterms ].coeff = p1.t[ i ].coeff + p2.t[ j ].coeff;
        p3.t[ p3.noofterms ].exp = p1.t[ i ].exp;
        i++;
        j++;
      }
```



```
   else
   {
    p3.t[ p3.noofterms ].coeff = p1.t[ i ].coeff;
    p3.t[ p3.noofterms ].exp = p1.t[ i ].exp;
    i++;
   }
  }
  else
  {
   p3.t[ p3.noofterms ].coeff = p2.t[ j ].coeff;
   p3.t[ p3.noofterms ].exp = p2.t[ j ].exp;
   j++;
  }
 }
 return p3;
}

/* multiplies two polynomials p1 and p2 */
struct poly polymul (struct poly p1, struct poly p2)
{
 int coeff, exp;
 struct poly temp, p3;
 initpoly (&temp);
 initpoly (&p3);
 if (p1.noofterms != 0 && p2.noofterms != 0)

{
  int i;
  for (i = 0; i < p1.noofterms; i++)
  {
   int j;
   struct poly p;
   initpoly (&p);
   for (j = 0; j < p2.noofterms; j++)
   {
    coeff = p1.t[ i ].coeff * p2.t[ j ].coeff;
    exp = p1.t[ i ].exp + p2.t[ j ].exp;
    polyappend (&p, coeff, exp);
   }
   if (i != 0)
   {
    p3 = polyadd (temp, p);
    temp = p3;
   }
   else
    temp = p;
  }
 }
 return p3;
}
```

Output:

```
First polynomial:
1 x^4 + 2 x^3 + 2 x^2 + 2 x^1
Second polynomial:
2 x^3 + 3 x^2 + 4 x^1
```



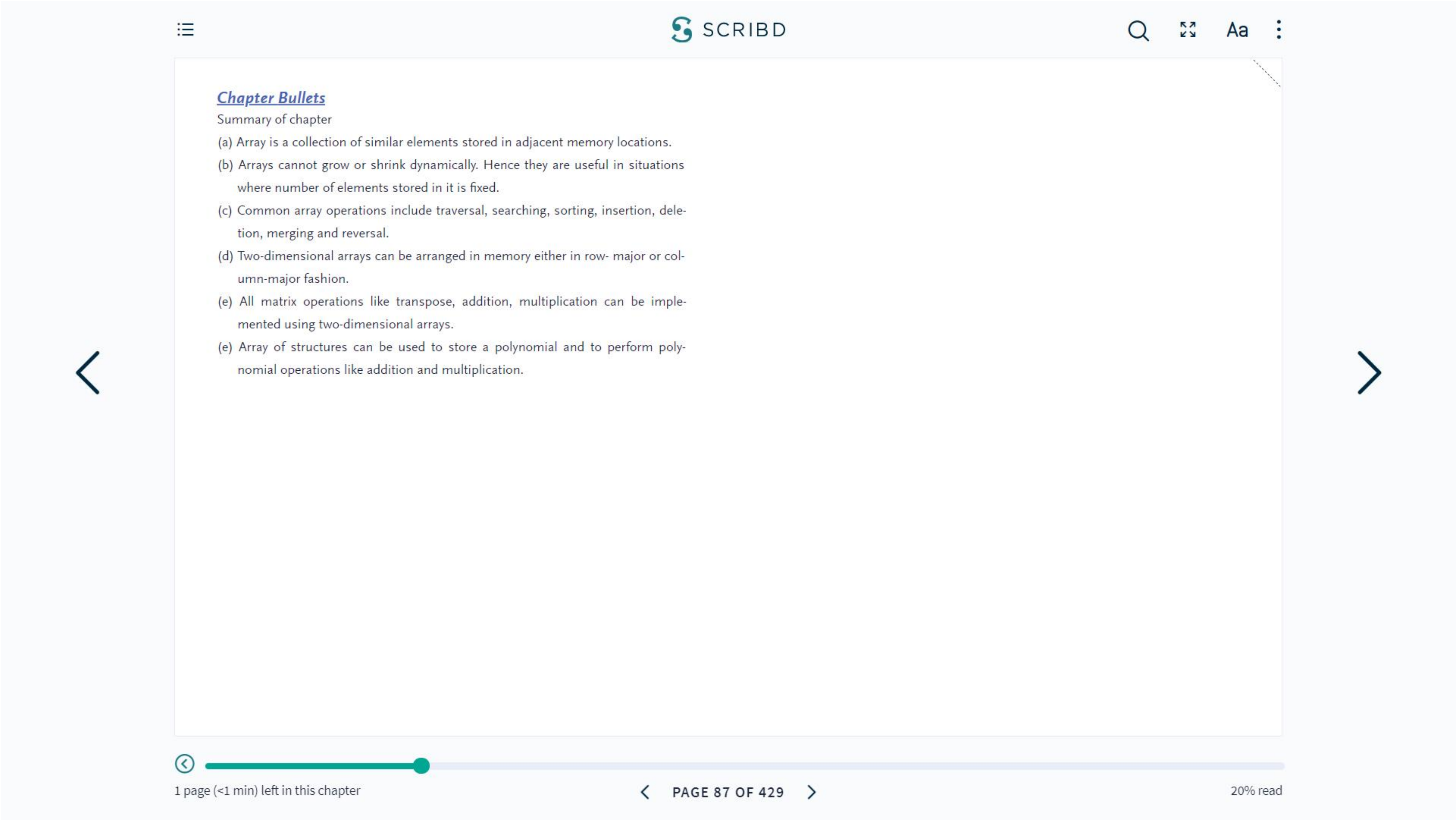
Resultant polynomial:

```
2 x^7 + 7 x^6 + 14 x^5 + 18 x^4 + 14 x^3 + 8 x^2
```

As in the previous program, here too we have called **polyappend()** function several times to build the two polynomials which are represented by the variables **p1** and **p2**. To carry out multiplication the function **polymul()** is called and **p1** and **p2** are passed to it. It returns the product of polynomials **p1** and **p2** which we have collected in **p3**.

In **polymul()** function, first we have checked that whether the two polynomials **p1** and **p2** are non-empty. If they are not then the control goes in a pair of **for** loop. Here, each term of first polynomial contained in **p1** is multiplied with every term of second polynomial contained in **p2**. While doing so, we have called **polyappend()** to add the terms to **p**. The first resultant polynomial is stored in temporary variable **temp** of the type **struct poly**. There onwards the function **polyadd()** is called to add the resulting polynomials.

Lastly, the terms of the resulting polynomial are displayed using the function **display()**.



Chapter Bullets

Summary of chapter

(a) Array is a collection of similar elements stored in adjacent memory locations.

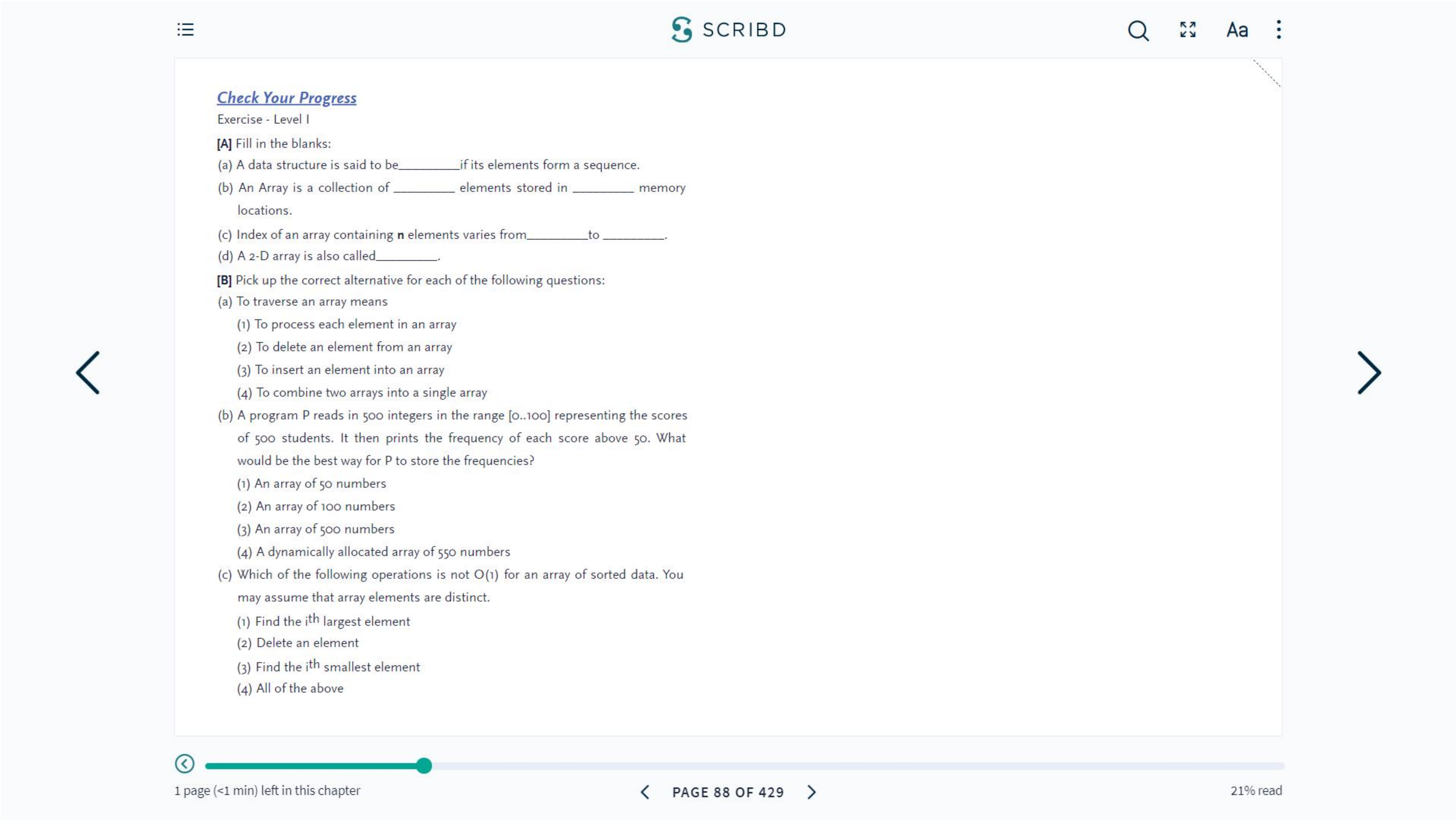
(b) Arrays cannot grow or shrink dynamically. Hence they are useful in situations where number of elements stored in it is fixed.

(c) Common array operations include traversal, searching, sorting, insertion, deletion, merging and reversal.

(d) Two-dimensional arrays can be arranged in memory either in row- major or column-major fashion.

(e) All matrix operations like transpose, addition, multiplication can be implemented using two-dimensional arrays.

(e) Array of structures can be used to store a polynomial and to perform polynomial operations like addition and multiplication.



Check Your Progress

Exercise - Level I

[A] Fill in the blanks:

(a) A data structure is said to be_________if its elements form a sequence.

(b) An Array is a collection of _________ elements stored in _________ memory locations.

(c) Index of an array containing **n** elements varies from_________to _________.

(d) A 2-D array is also called_________.

[B] Pick up the correct alternative for each of the following questions:

(a) To traverse an array means

(1) To process each element in an array

(2) To delete an element from an array

(3) To insert an element into an array

(4) To combine two arrays into a single array

(b) A program P reads in 500 integers in the range [0..100] representing the scores of 500 students. It then prints the frequency of each score above 50. What would be the best way for P to store the frequencies?

(1) An array of 50 numbers

(2) An array of 100 numbers

(3) An array of 500 numbers

(4) A dynamically allocated array of 550 numbers

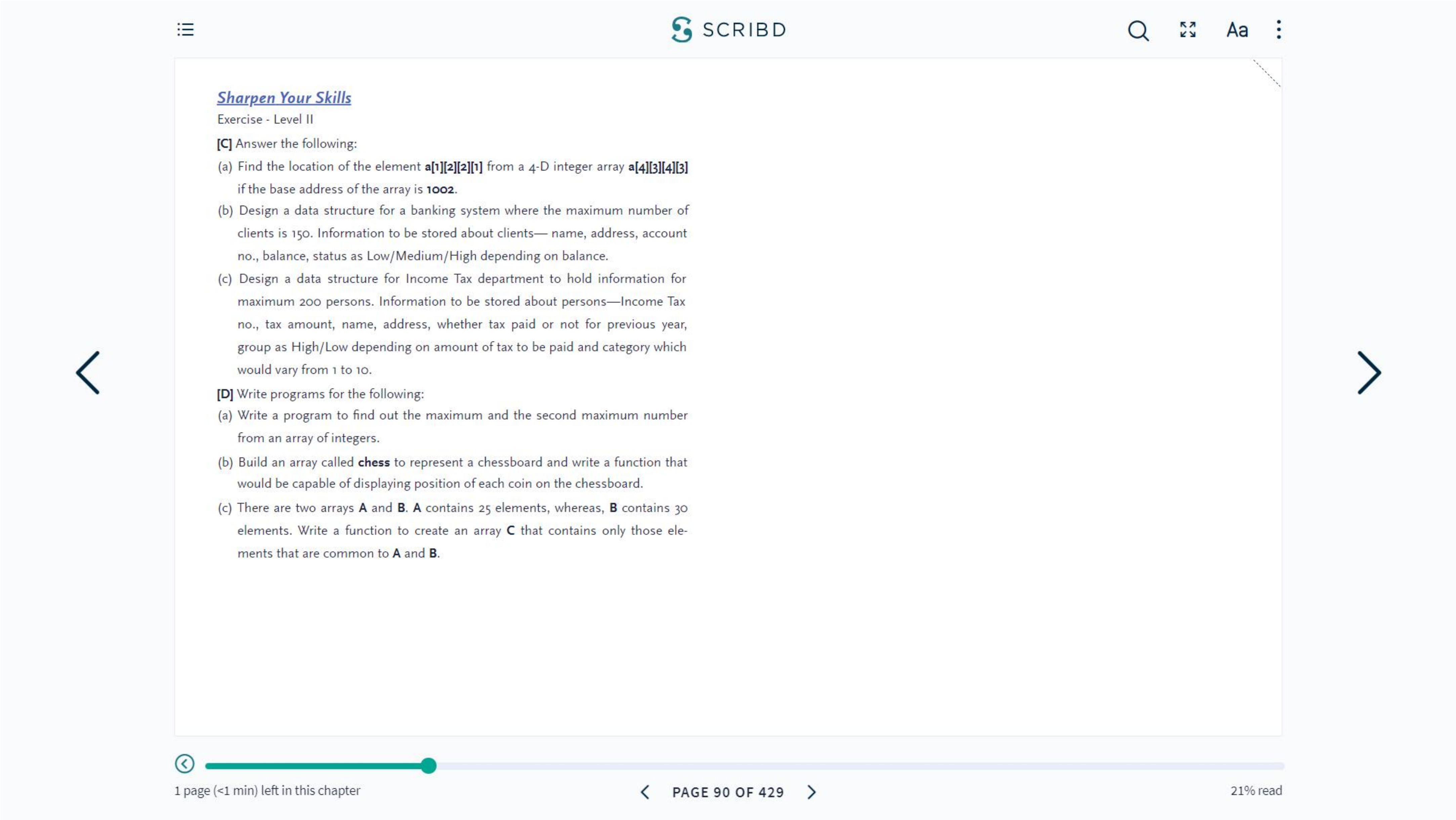
(c) Which of the following operations is not O(1) for an array of sorted data. You may assume that array elements are distinct.

(1) Find the i^{th} largest element

(2) Delete an element

(3) Find the i^{th} smallest element

(4) All of the above



Sharpen Your Skills

Exercise - Level II

[C] Answer the following:

(a) Find the location of the element **a[1][2][2][1]** from a 4-D integer array **a[4][3][4][3]** if the base address of the array is **1002**.

(b) Design a data structure for a banking system where the maximum number of clients is 150. Information to be stored about clients— name, address, account no., balance, status as Low/Medium/High depending on balance.

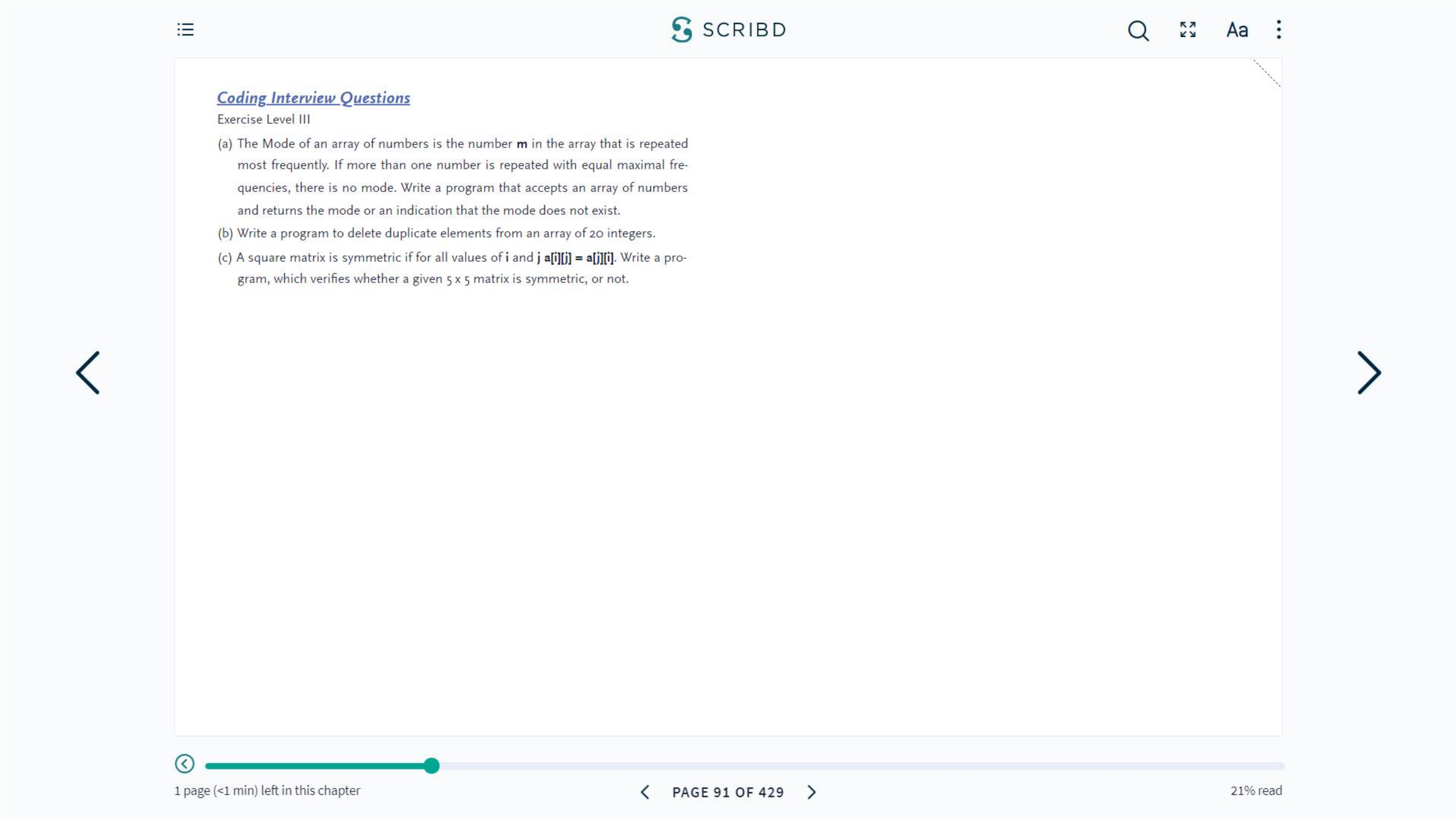
(c) Design a data structure for Income Tax department to hold information for maximum 200 persons. Information to be stored about persons—Income Tax no., tax amount, name, address, whether tax paid or not for previous year, group as High/Low depending on amount of tax to be paid and category which would vary from 1 to 10.

[D] Write programs for the following:

(a) Write a program to find out the maximum and the second maximum number from an array of integers.

(b) Build an array called **chess** to represent a chessboard and write a function that would be capable of displaying position of each coin on the chessboard.

(c) There are two arrays **A** and **B**. **A** contains 25 elements, whereas, **B** contains 30 elements. Write a function to create an array **C** that contains only those elements that are common to **A** and **B**.



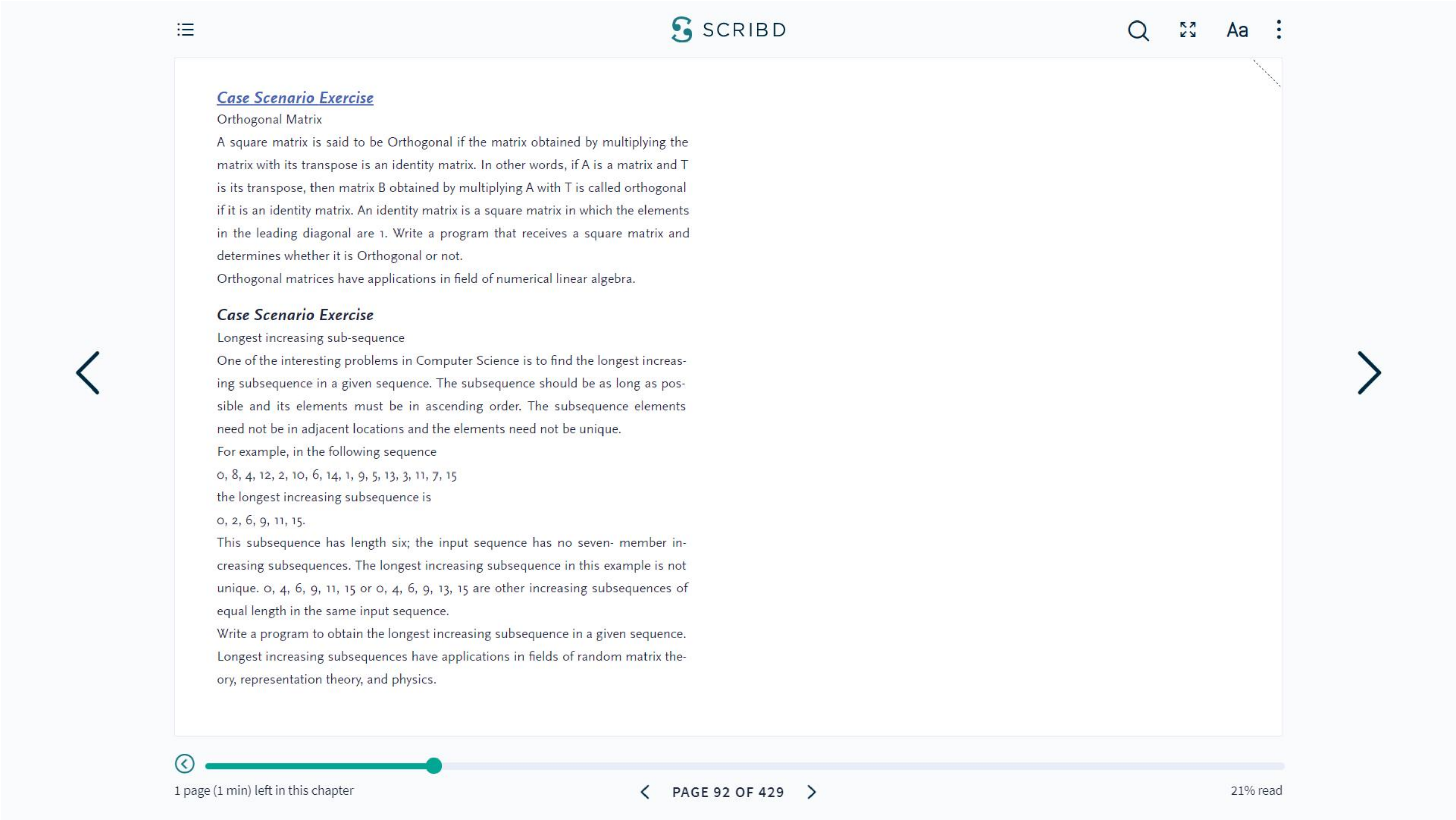
Coding Interview Questions

Exercise Level III

(a) The Mode of an array of numbers is the number **m** in the array that is repeated most frequently. If more than one number is repeated with equal maximal frequencies, there is no mode. Write a program that accepts an array of numbers and returns the mode or an indication that the mode does not exist.

(b) Write a program to delete duplicate elements from an array of 20 integers.

(c) A square matrix is symmetric if for all values of **i** and **j a[i][j] = a[j][i]**. Write a program, which verifies whether a given 5 x 5 matrix is symmetric, or not.



Case Scenario Exercise

Orthogonal Matrix

A square matrix is said to be Orthogonal if the matrix obtained by multiplying the matrix with its transpose is an identity matrix. In other words, if A is a matrix and T is its transpose, then matrix B obtained by multiplying A with T is called orthogonal if it is an identity matrix. An identity matrix is a square matrix in which the elements in the leading diagonal are 1. Write a program that receives a square matrix and determines whether it is Orthogonal or not.

Orthogonal matrices have applications in field of numerical linear algebra.

Case Scenario Exercise

Longest increasing sub-sequence

One of the interesting problems in Computer Science is to find the longest increasing subsequence in a given sequence. The subsequence should be as long as possible and its elements must be in ascending order. The subsequence elements need not be in adjacent locations and the elements need not be unique.

For example, in the following sequence

0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15

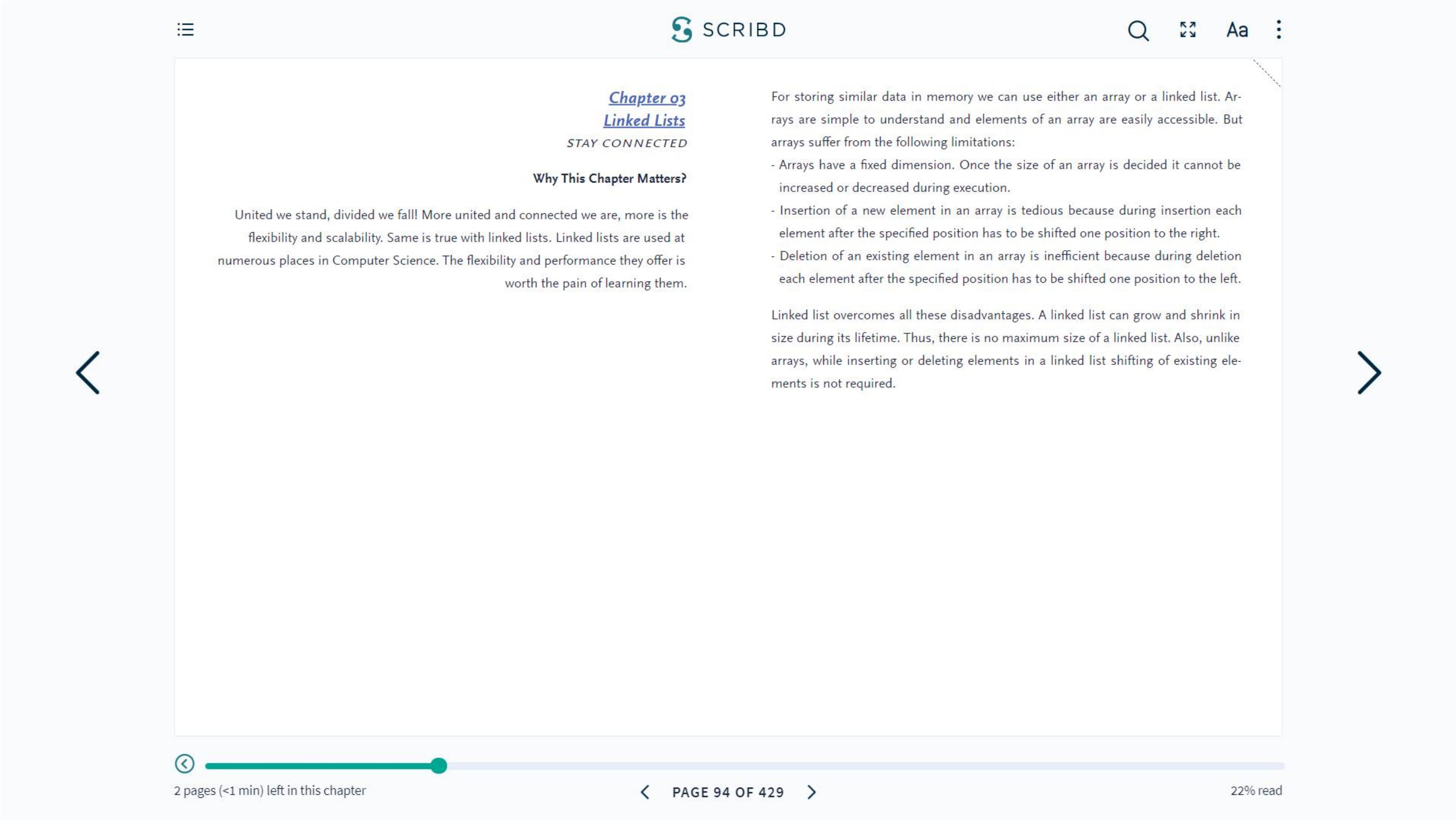
the longest increasing subsequence is

0, 2, 6, 9, 11, 15.

This subsequence has length six; the input sequence has no seven- member increasing subsequences. The longest increasing subsequence in this example is not unique. 0, 4, 6, 9, 11, 15 or 0, 4, 6, 9, 13, 15 are other increasing subsequences of equal length in the same input sequence.

Write a program to obtain the longest increasing subsequence in a given sequence.

Longest increasing subsequences have applications in fields of random matrix theory, representation theory, and physics.



Chapter 03
Linked Lists

STAY CONNECTED

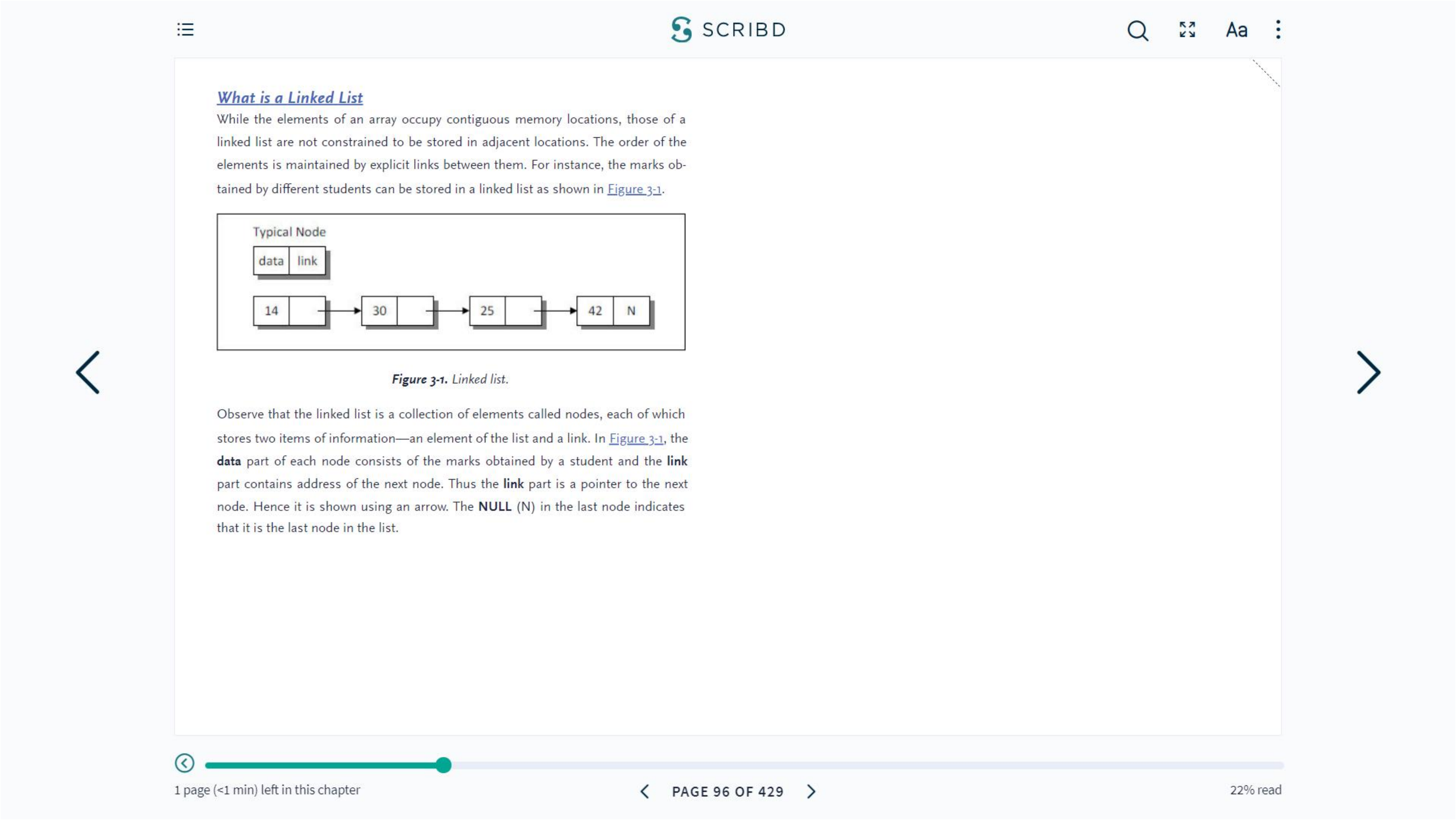
Why This Chapter Matters?

United we stand, divided we fall! More united and connected we are, more is the flexibility and scalability. Same is true with linked lists. Linked lists are used at numerous places in Computer Science. The flexibility and performance they offer is worth the pain of learning them.

For storing similar data in memory we can use either an array or a linked list. Arrays are simple to understand and elements of an array are easily accessible. But arrays suffer from the following limitations:

- Arrays have a fixed dimension. Once the size of an array is decided it cannot be increased or decreased during execution.
- Insertion of a new element in an array is tedious because during insertion each element after the specified position has to be shifted one position to the right.
- Deletion of an existing element in an array is inefficient because during deletion each element after the specified position has to be shifted one position to the left.

Linked list overcomes all these disadvantages. A linked list can grow and shrink in size during its lifetime. Thus, there is no maximum size of a linked list. Also, unlike arrays, while inserting or deleting elements in a linked list shifting of existing elements is not required.

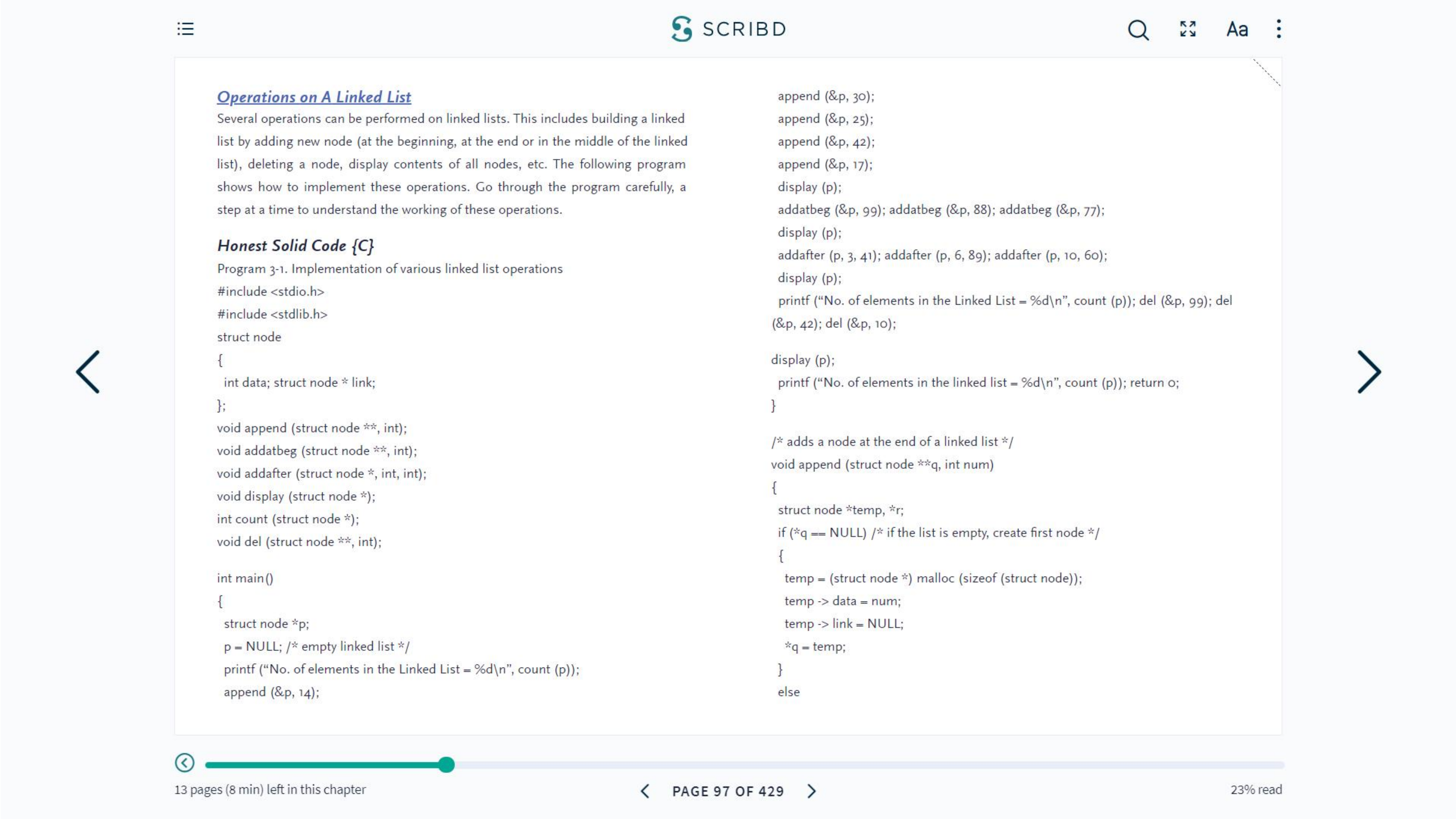


What is a Linked List

While the elements of an array occupy contiguous memory locations, those of a linked list are not constrained to be stored in adjacent locations. The order of the elements is maintained by explicit links between them. For instance, the marks obtained by different students can be stored in a linked list as shown in Figure 3-1.

Figure 3-1. *Linked list.*

Observe that the linked list is a collection of elements called nodes, each of which stores two items of information—an element of the list and a link. In Figure 3-1, the **data** part of each node consists of the marks obtained by a student and the **link** part contains address of the next node. Thus the **link** part is a pointer to the next node. Hence it is shown using an arrow. The **NULL** (N) in the last node indicates that it is the last node in the list.



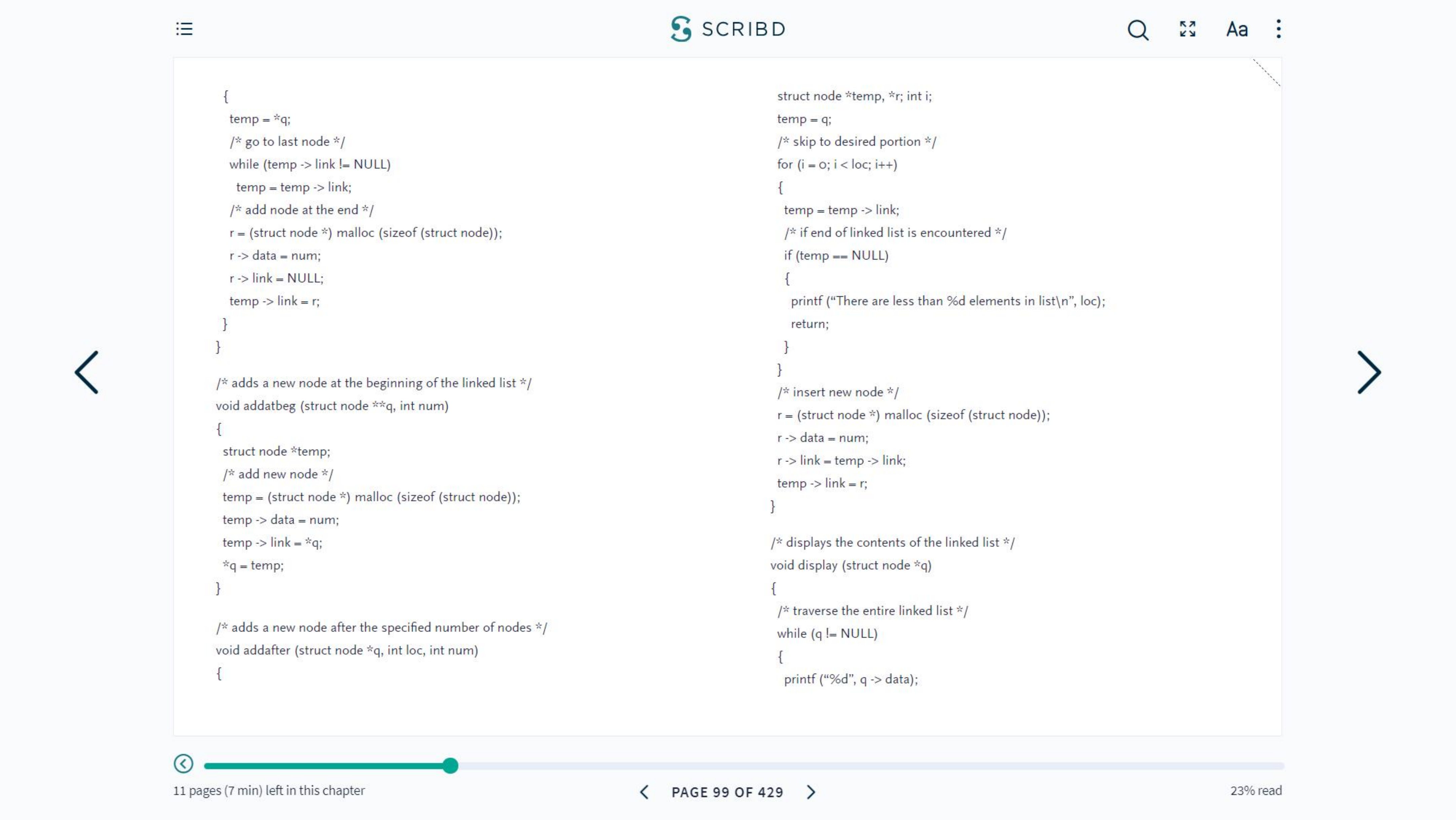
Operations on A Linked List

Several operations can be performed on linked lists. This includes building a linked list by adding new node (at the beginning, at the end or in the middle of the linked list), deleting a node, display contents of all nodes, etc. The following program shows how to implement these operations. Go through the program carefully, a step at a time to understand the working of these operations.

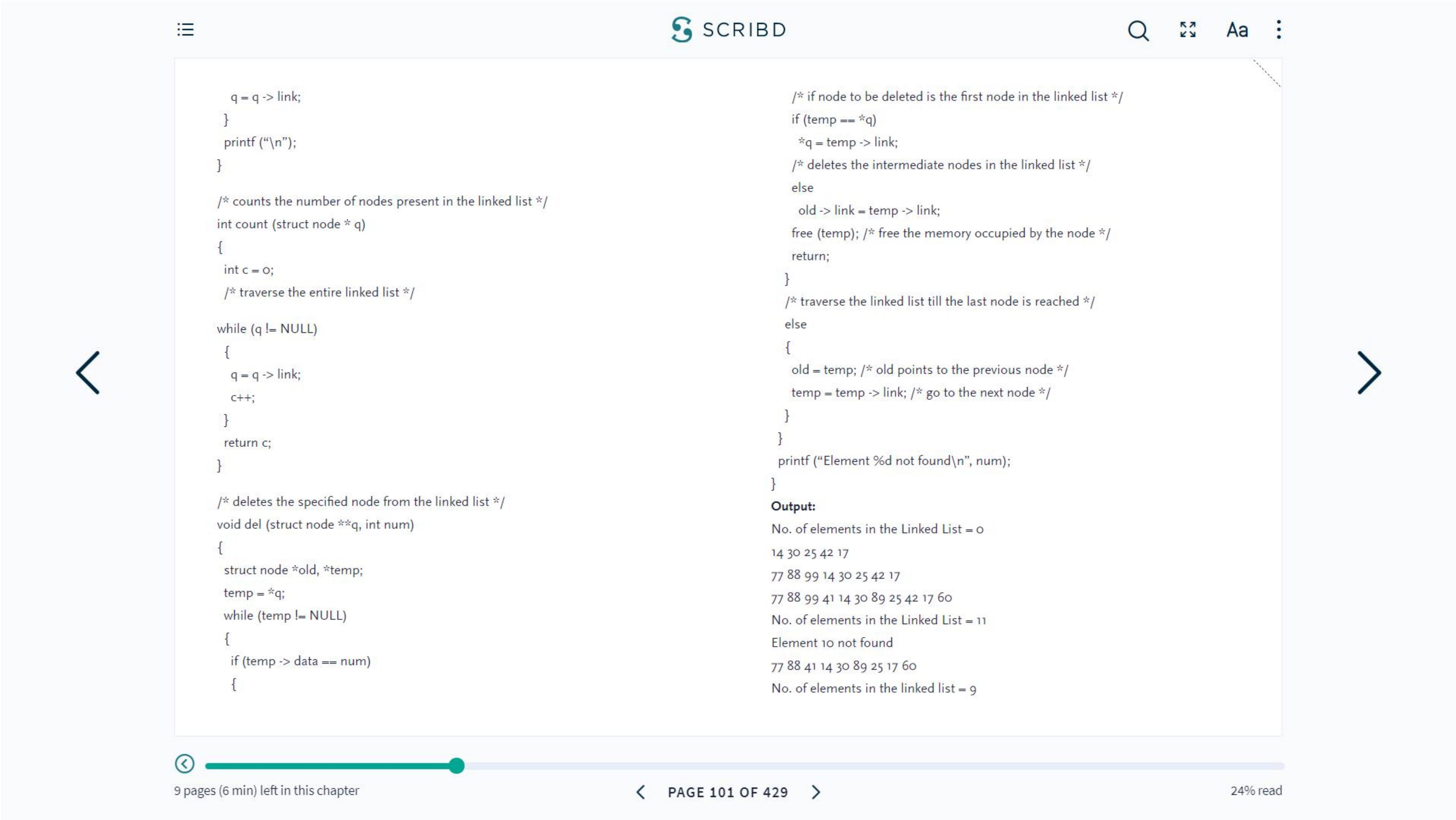
Honest Solid Code {C}

Program 3-1. Implementation of various linked list operations

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
 int data; struct node * link;
};
void append (struct node **, int);
void addatbeg (struct node **, int);
void addafter (struct node *, int, int);
void display (struct node *);
int count (struct node *);
void del (struct node **, int);

int main()
{
 struct node *p;
 p = NULL; /* empty linked list */
 printf (“No. of elements in the Linked List = %d\n”, count (p));
 append (&p, 14);
 append (&p, 30);
 append (&p, 25);
 append (&p, 42);
 append (&p, 17);
 display (p);
 addatbeg (&p, 99); addatbeg (&p, 88); addatbeg (&p, 77);
 display (p);
 addafter (p, 3, 41); addafter (p, 6, 89); addafter (p, 10, 60);
 display (p);
 printf (“No. of elements in the Linked List = %d\n”, count (p)); del (&p, 99); del
(&p, 42); del (&p, 10);

display (p);
 printf (“No. of elements in the linked list = %d\n”, count (p)); return 0;
}

/* adds a node at the end of a linked list */
void append (struct node **q, int num)
{
 struct node *temp, *r;
 if (*q == NULL) /* if the list is empty, create first node */
 {
  temp = (struct node *) malloc (sizeof (struct node));
  temp -> data = num;
  temp -> link = NULL;
  *q = temp;
 }
 else
```



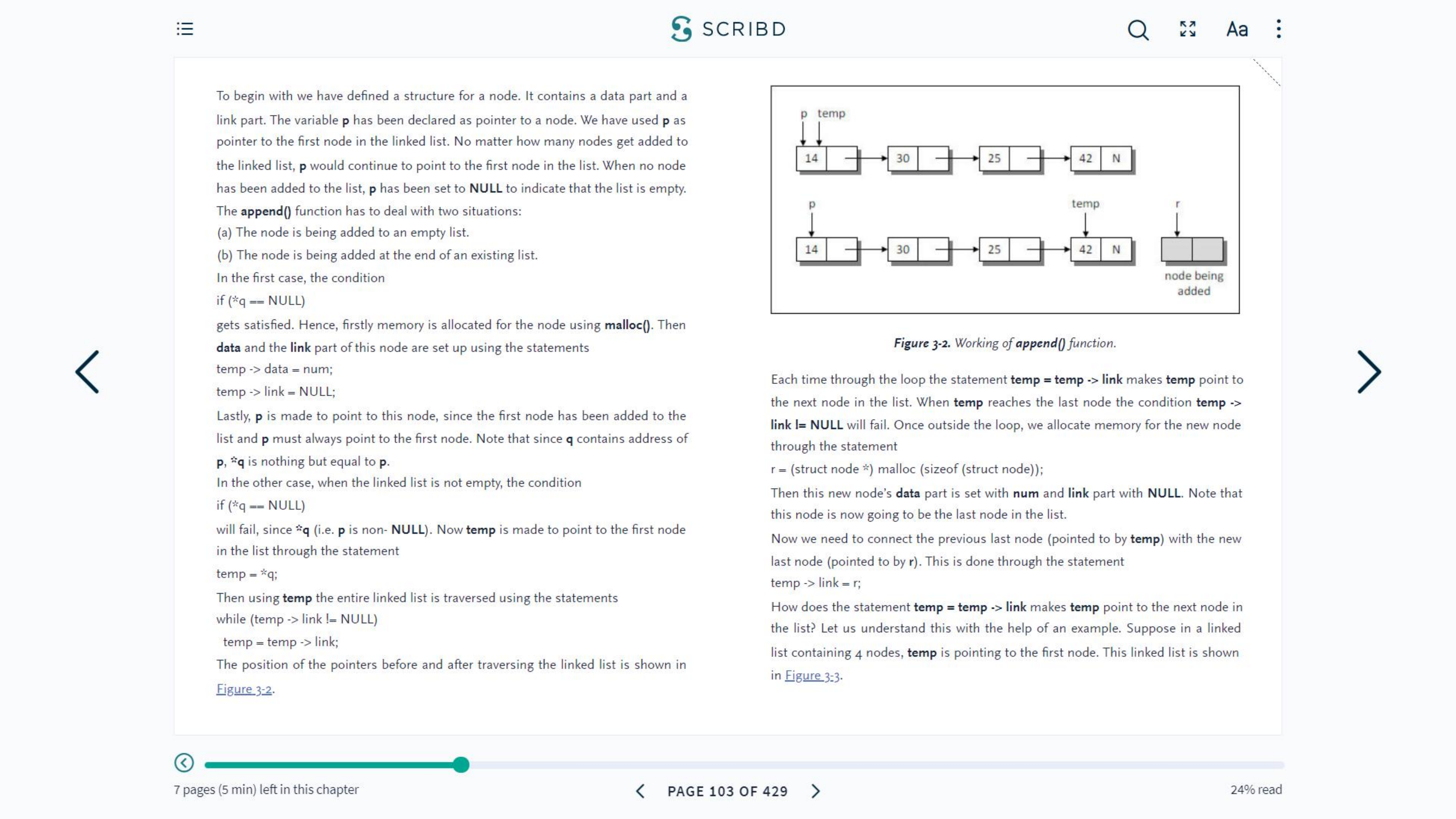
```
 {
  temp = *q;
  /* go to last node */
  while (temp -> link != NULL)
   temp = temp -> link;
  /* add node at the end */
  r = (struct node *) malloc (sizeof (struct node));
  r -> data = num;
  r -> link = NULL;
  temp -> link = r;
 }
}

/* adds a new node at the beginning of the linked list */
void addatbeg (struct node **q, int num)
{
 struct node *temp;
 /* add new node */
 temp = (struct node *) malloc (sizeof (struct node));
 temp -> data = num;
 temp -> link = *q;
 *q = temp;
}

/* adds a new node after the specified number of nodes */
void addafter (struct node *q, int loc, int num)
{
 struct node *temp, *r; int i;
 temp = q;
 /* skip to desired portion */
 for (i = 0; i < loc; i++)
 {
  temp = temp -> link;
  /* if end of linked list is encountered */
  if (temp == NULL)
  {
   printf (“There are less than %d elements in list\n”, loc);
   return;
  }
 }
 /* insert new node */
 r = (struct node *) malloc (sizeof (struct node));
 r -> data = num;
 r -> link = temp -> link;
 temp -> link = r;
}

/* displays the contents of the linked list */
void display (struct node *q)
{
 /* traverse the entire linked list */
 while (q != NULL)
 {
  printf (“%d”, q -> data);
```



```
    q = q -> link;
  }
  printf ("\n");
}

/* counts the number of nodes present in the linked list */
int count (struct node * q)
{
  int c = 0;
  /* traverse the entire linked list */

while (q != NULL)
  {
    q = q -> link;
    c++;
  }
  return c;
}

/* deletes the specified node from the linked list */
void del (struct node **q, int num)
{
  struct node *old, *temp;
  temp = *q;
  while (temp != NULL)
  {
    if (temp -> data == num)
    {
      /* if node to be deleted is the first node in the linked list */
      if (temp == *q)
        *q = temp -> link;
      /* deletes the intermediate nodes in the linked list */
      else
        old -> link = temp -> link;
      free (temp); /* free the memory occupied by the node */
      return;
    }
    /* traverse the linked list till the last node is reached */
    else
    {
      old = temp; /* old points to the previous node */
      temp = temp -> link; /* go to the next node */
    }
  }
  printf ("Element %d not found\n", num);
}
```

Output:

```
No. of elements in the Linked List = 0
14 30 25 42 17
77 88 99 14 30 25 42 17
77 88 99 41 14 30 89 25 42 17 60
No. of elements in the Linked List = 11
Element 10 not found
77 88 41 14 30 89 25 17 60
No. of elements in the linked list = 9
```



To begin with we have defined a structure for a node. It contains a data part and a link part. The variable **p** has been declared as pointer to a node. We have used **p** as pointer to the first node in the linked list. No matter how many nodes get added to the linked list, **p** would continue to point to the first node in the list. When no node has been added to the list, **p** has been set to **NULL** to indicate that the list is empty.

The **append()** function has to deal with two situations:

(a) The node is being added to an empty list.

(b) The node is being added at the end of an existing list.

In the first case, the condition

```
if (*q == NULL)
```

gets satisfied. Hence, firstly memory is allocated for the node using **malloc()**. Then **data** and the **link** part of this node are set up using the statements

```
temp -> data = num;
temp -> link = NULL;
```

Lastly, **p** is made to point to this node, since the first node has been added to the list and **p** must always point to the first node. Note that since **q** contains address of **p**, ***q** is nothing but equal to **p**.

In the other case, when the linked list is not empty, the condition

```
if (*q == NULL)
```

will fail, since ***q** (i.e. **p** is non- **NULL**). Now **temp** is made to point to the first node in the list through the statement

```
temp = *q;
```

Then using **temp** the entire linked list is traversed using the statements

```
while (temp -> link != NULL)
 temp = temp -> link;
```

The position of the pointers before and after traversing the linked list is shown in Figure 3-2.

Figure 3-2. *Working of* ***append()*** *function.*

Each time through the loop the statement **temp = temp -> link** makes **temp** point to the next node in the list. When **temp** reaches the last node the condition **temp -> link != NULL** will fail. Once outside the loop, we allocate memory for the new node through the statement

```
r = (struct node *) malloc (sizeof (struct node));
```

Then this new node's **data** part is set with **num** and **link** part with **NULL**. Note that this node is now going to be the last node in the list.

Now we need to connect the previous last node (pointed to by **temp**) with the new last node (pointed to by **r**). This is done through the statement

```
temp -> link = r;
```

How does the statement **temp = temp -> link** makes **temp** point to the next node in the list? Let us understand this with the help of an example. Suppose in a linked list containing 4 nodes, **temp** is pointing to the first node. This linked list is shown in Figure 3-3.

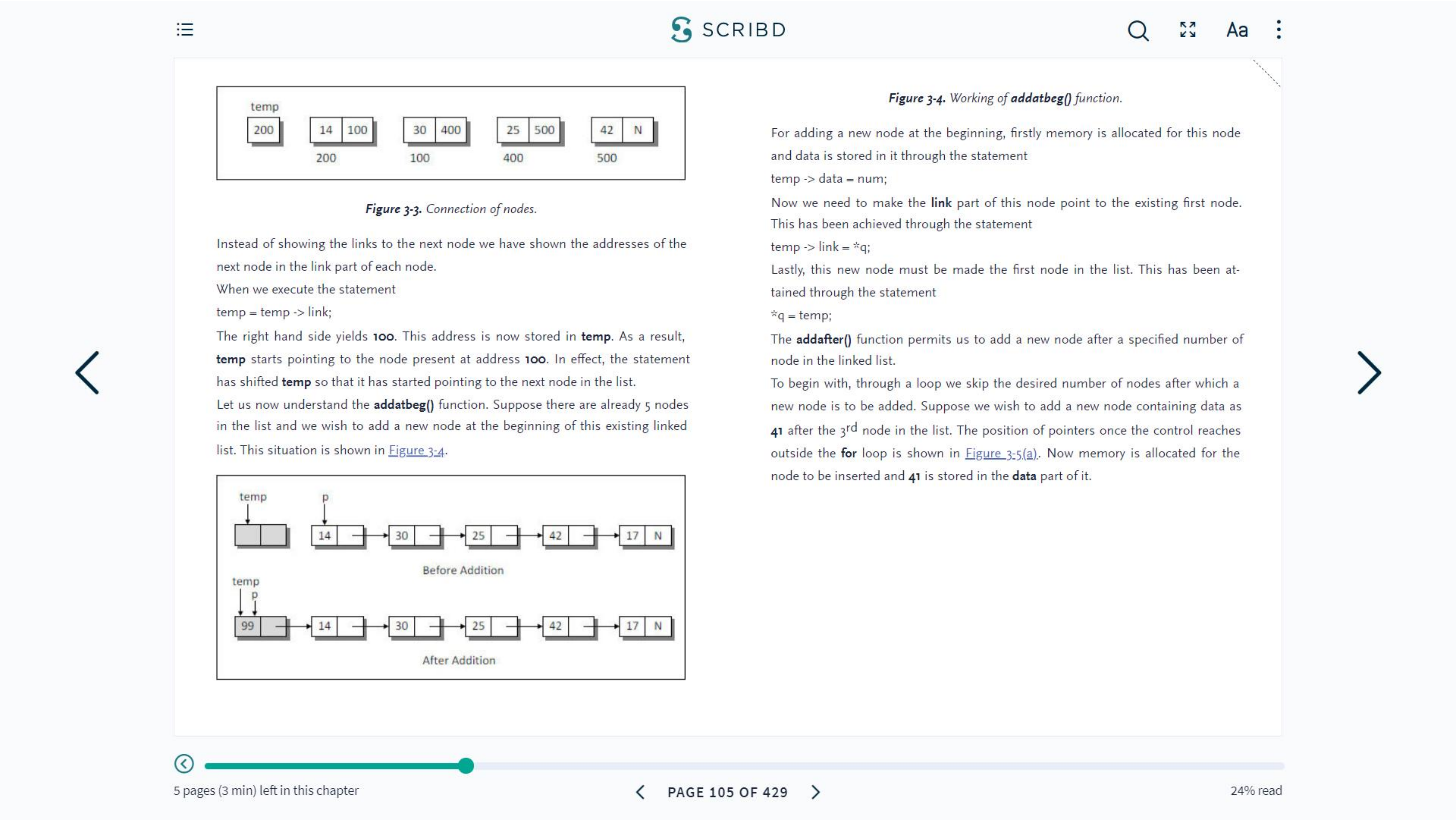


Figure 3-3. *Connection of nodes.*

Instead of showing the links to the next node we have shown the addresses of the next node in the link part of each node.

When we execute the statement

temp = temp -> link;

The right hand side yields **100**. This address is now stored in **temp**. As a result, **temp** starts pointing to the node present at address **100**. In effect, the statement has shifted **temp** so that it has started pointing to the next node in the list.

Let us now understand the **addatbeg()** function. Suppose there are already 5 nodes in the list and we wish to add a new node at the beginning of this existing linked list. This situation is shown in Figure 3-4.

Figure 3-4. *Working of* ***addatbeg()*** *function.*

For adding a new node at the beginning, firstly memory is allocated for this node and data is stored in it through the statement

temp -> data = num;

Now we need to make the **link** part of this node point to the existing first node. This has been achieved through the statement

temp -> link = *q;

Lastly, this new node must be made the first node in the list. This has been attained through the statement

*q = temp;

The **addafter()** function permits us to add a new node after a specified number of node in the linked list.

To begin with, through a loop we skip the desired number of nodes after which a new node is to be added. Suppose we wish to add a new node containing data as **41** after the 3rd node in the list. The position of pointers once the control reaches outside the **for** loop is shown in Figure 3-5(a). Now memory is allocated for the node to be inserted and **41** is stored in the **data** part of it.

5 pages (3 min) left in this chapter

PAGE 105 OF 429

24% read

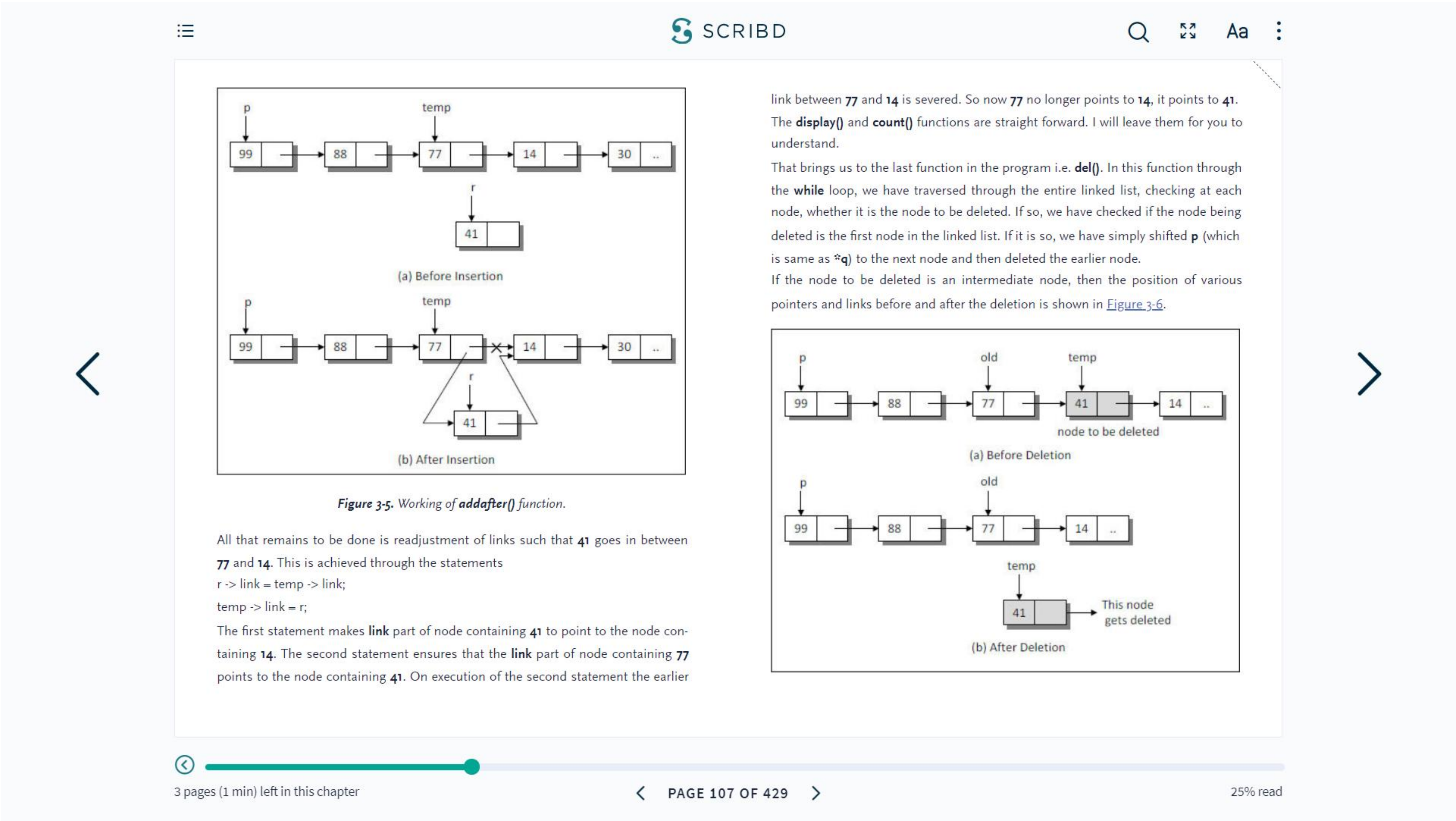


Figure 3-5. Working of ***addafter()*** *function.*

All that remains to be done is readjustment of links such that **41** goes in between **77** and **14**. This is achieved through the statements

r -> link = temp -> link;

temp -> link = r;

The first statement makes **link** part of node containing **41** to point to the node containing **14**. The second statement ensures that the **link** part of node containing **77** points to the node containing **41**. On execution of the second statement the earlier link between **77** and **14** is severed. So now **77** no longer points to **14**, it points to **41**.

The **display()** and **count()** functions are straight forward. I will leave them for you to understand.

That brings us to the last function in the program i.e. **del()**. In this function through the **while** loop, we have traversed through the entire linked list, checking at each node, whether it is the node to be deleted. If so, we have checked if the node being deleted is the first node in the linked list. If it is so, we have simply shifted **p** (which is same as ***q**) to the next node and then deleted the earlier node.

If the node to be deleted is an intermediate node, then the position of various pointers and links before and after the deletion is shown in Figure 3-6.

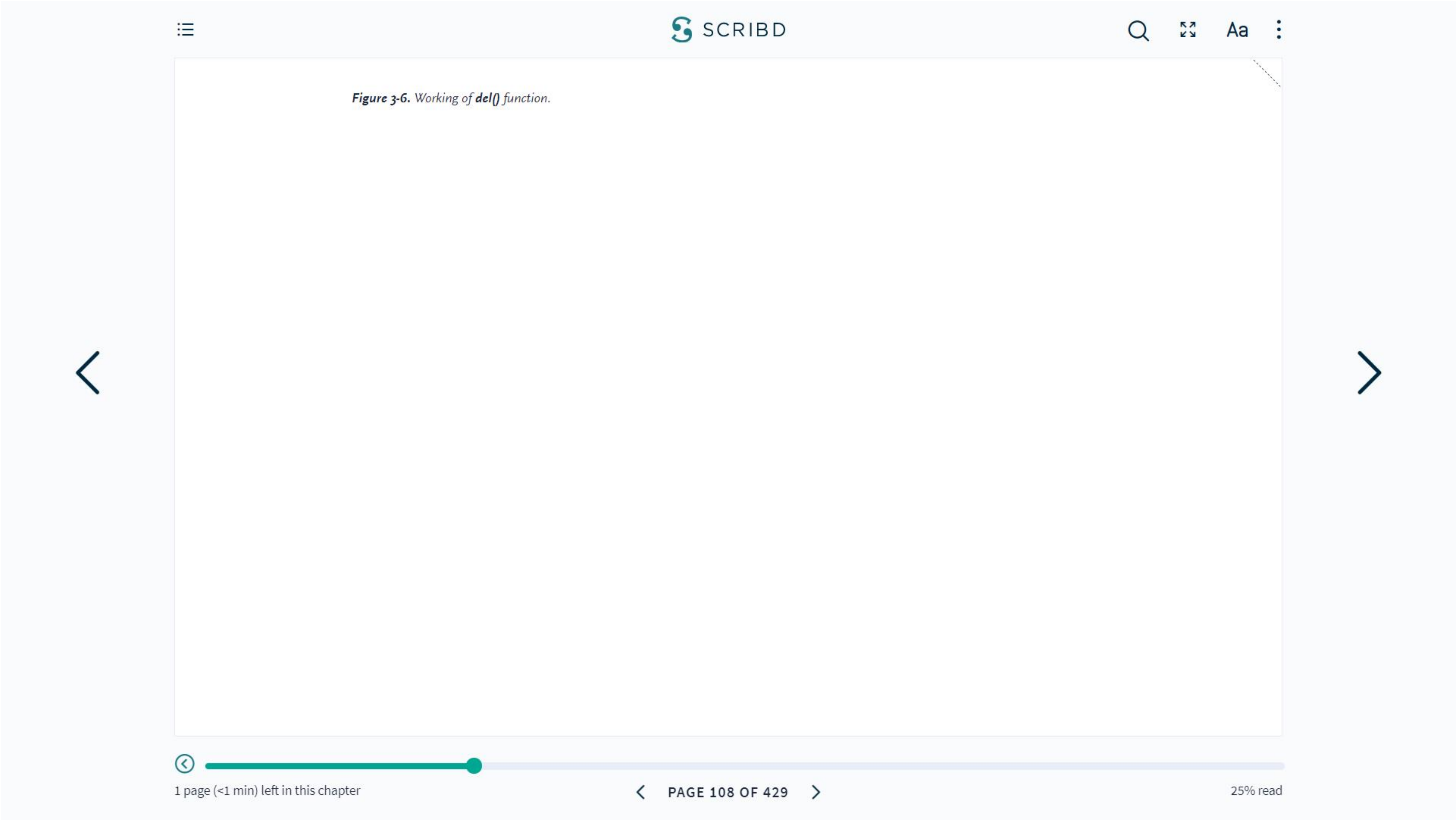
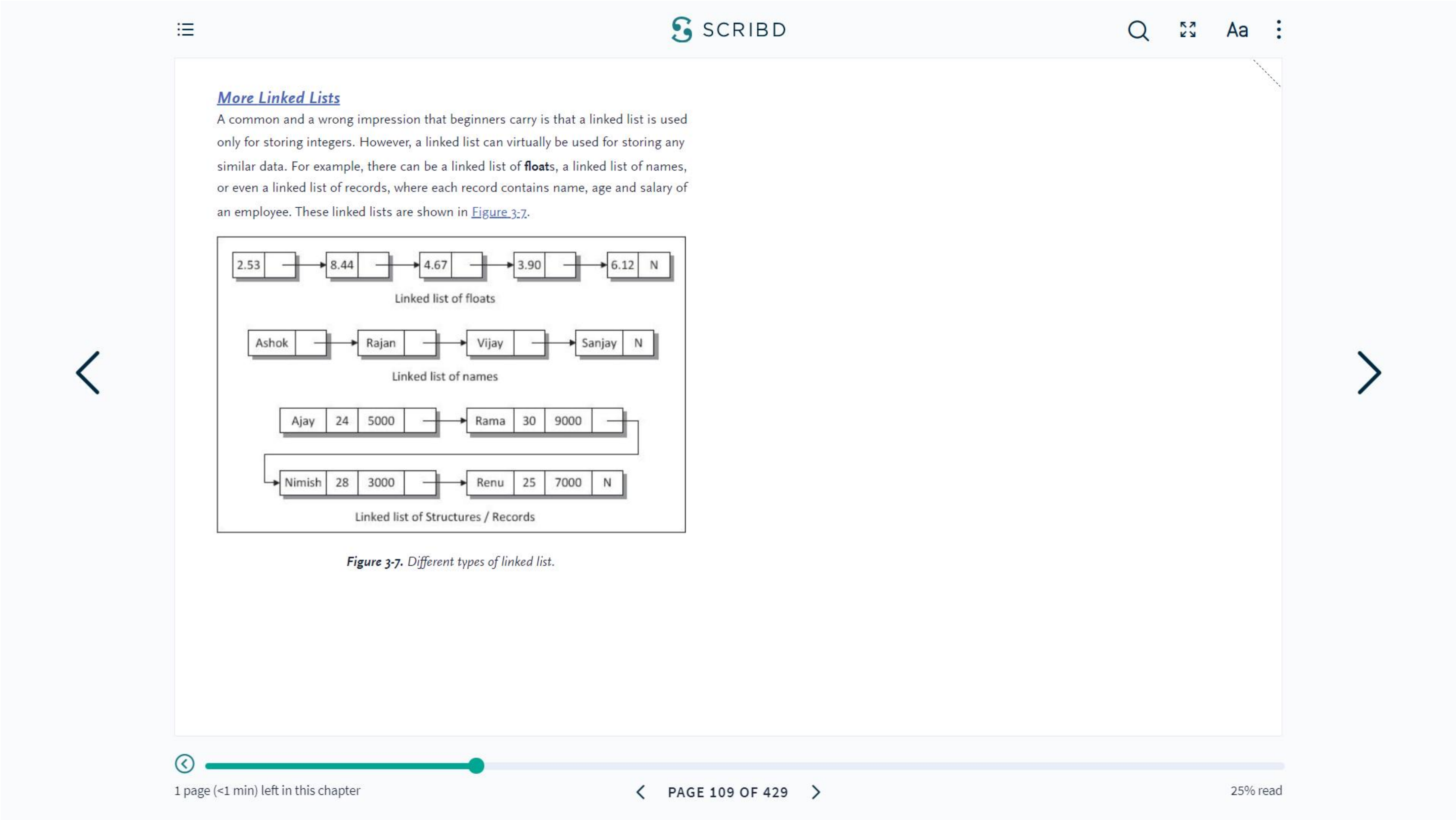


Figure 3-6. *Working of **del()** function.*

1 page (<1 min) left in this chapter

PAGE 108 OF 429

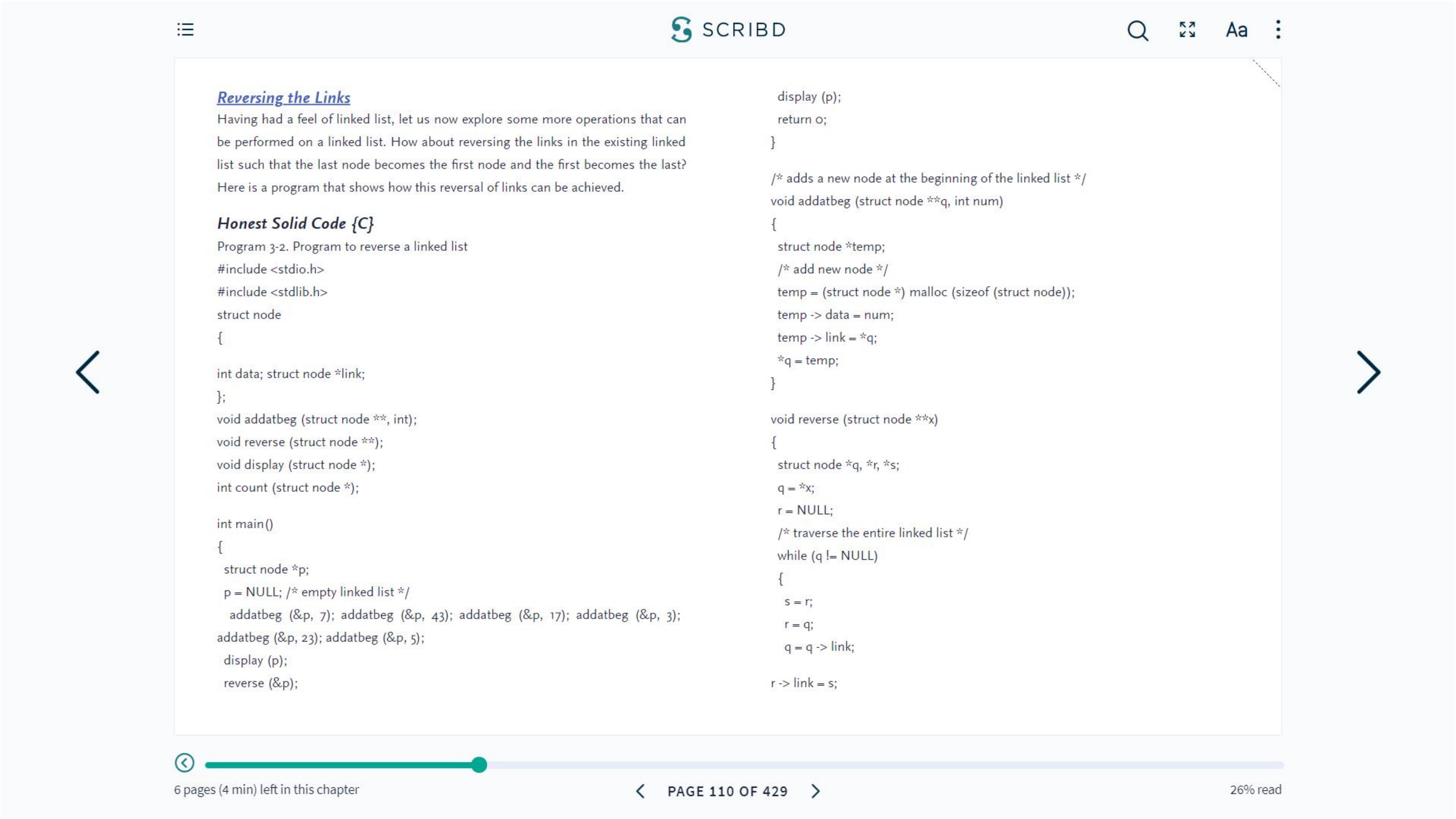
25% read



More Linked Lists

A common and a wrong impression that beginners carry is that a linked list is used only for storing integers. However, a linked list can virtually be used for storing any similar data. For example, there can be a linked list of **float**s, a linked list of names, or even a linked list of records, where each record contains name, age and salary of an employee. These linked lists are shown in Figure 3-7.

Figure 3-7. *Different types of linked list.*



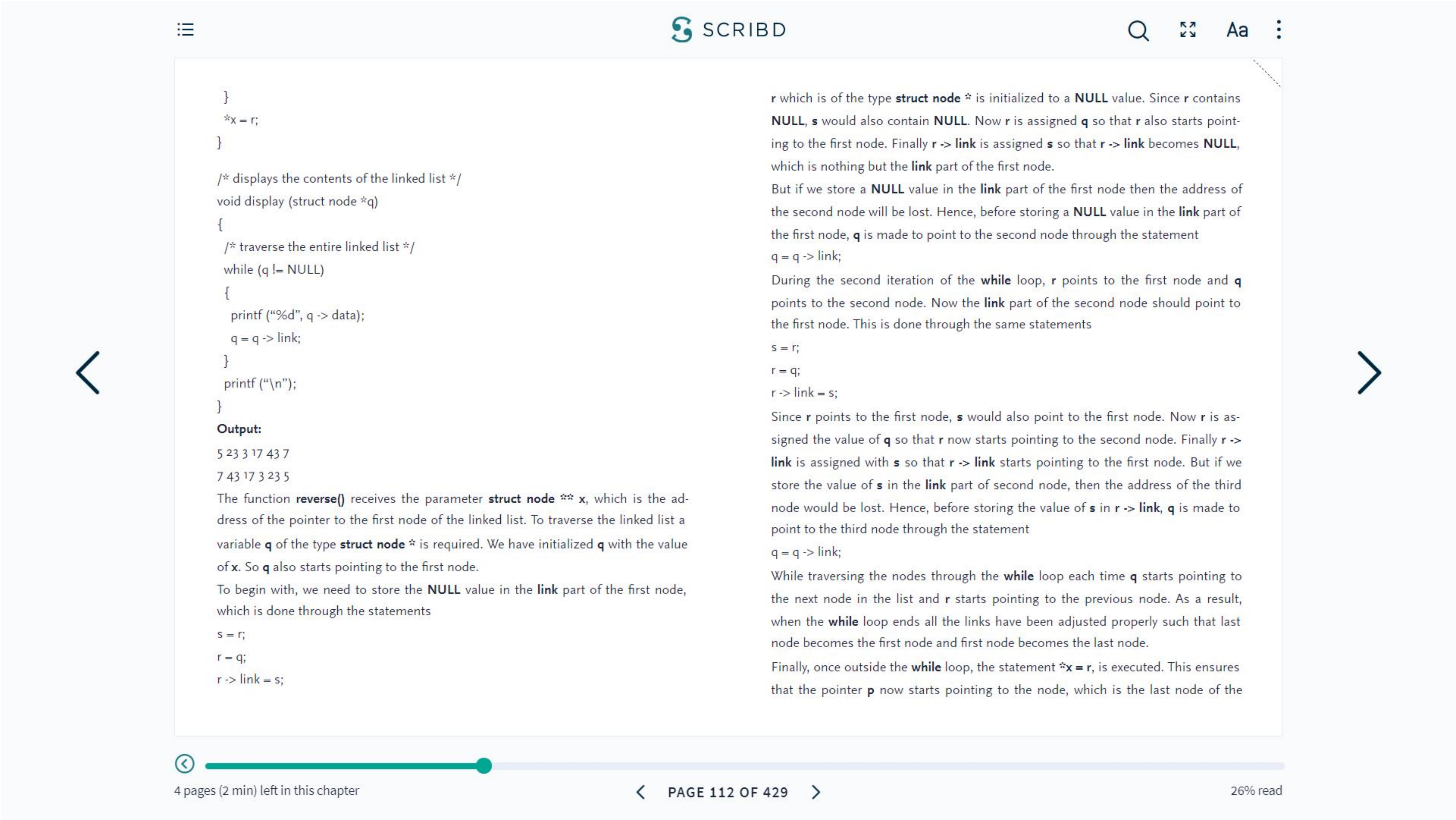
Reversing the Links

Having had a feel of linked list, let us now explore some more operations that can be performed on a linked list. How about reversing the links in the existing linked list such that the last node becomes the first node and the first becomes the last? Here is a program that shows how this reversal of links can be achieved.

Honest Solid Code {C}

Program 3-2. Program to reverse a linked list

```
#include <stdio.h>
#include <stdlib.h>
struct node
{

int data; struct node *link;
};
void addatbeg (struct node **, int);
void reverse (struct node **);
void display (struct node *);
int count (struct node *);

int main()
{
 struct node *p;
 p = NULL; /* empty linked list */
  addatbeg (&p, 7); addatbeg (&p, 43); addatbeg (&p, 17); addatbeg (&p, 3);
addatbeg (&p, 23); addatbeg (&p, 5);
 display (p);
 reverse (&p);
 display (p);
 return 0;
}

/* adds a new node at the beginning of the linked list */
void addatbeg (struct node **q, int num)
{
 struct node *temp;
 /* add new node */
 temp = (struct node *) malloc (sizeof (struct node));
 temp -> data = num;
 temp -> link = *q;
 *q = temp;
}

void reverse (struct node **x)
{
 struct node *q, *r, *s;
 q = *x;
 r = NULL;
 /* traverse the entire linked list */
 while (q != NULL)
 {
  s = r;
  r = q;
  q = q -> link;

r -> link = s;
```



```
  }
  *x = r;
}

/* displays the contents of the linked list */
void display (struct node *q)
{
  /* traverse the entire linked list */
  while (q != NULL)
  {
    printf ("%d", q -> data);
    q = q -> link;
  }
  printf ("\n");
}
```

Output:

```
5 23 3 17 43 7
7 43 17 3 23 5
```

The function **reverse()** receives the parameter **struct node ** x**, which is the address of the pointer to the first node of the linked list. To traverse the linked list a variable **q** of the type **struct node *** is required. We have initialized **q** with the value of **x**. So **q** also starts pointing to the first node.

To begin with, we need to store the **NULL** value in the **link** part of the first node, which is done through the statements

```
s = r;
r = q;
r -> link = s;
```

r which is of the type **struct node *** is initialized to a **NULL** value. Since **r** contains **NULL**, **s** would also contain **NULL**. Now **r** is assigned **q** so that **r** also starts pointing to the first node. Finally **r -> link** is assigned **s** so that **r -> link** becomes **NULL**, which is nothing but the **link** part of the first node.

But if we store a **NULL** value in the **link** part of the first node then the address of the second node will be lost. Hence, before storing a **NULL** value in the **link** part of the first node, **q** is made to point to the second node through the statement

```
q = q -> link;
```

During the second iteration of the **while** loop, **r** points to the first node and **q** points to the second node. Now the **link** part of the second node should point to the first node. This is done through the same statements

```
s = r;
r = q;
r -> link = s;
```

Since **r** points to the first node, **s** would also point to the first node. Now **r** is assigned the value of **q** so that **r** now starts pointing to the second node. Finally **r -> link** is assigned with **s** so that **r -> link** starts pointing to the first node. But if we store the value of **s** in the **link** part of second node, then the address of the third node would be lost. Hence, before storing the value of **s** in **r -> link**, **q** is made to point to the third node through the statement

```
q = q -> link;
```

While traversing the nodes through the **while** loop each time **q** starts pointing to the next node in the list and **r** starts pointing to the previous node. As a result, when the **while** loop ends all the links have been adjusted properly such that last node becomes the first node and first node becomes the last node.

Finally, once outside the **while** loop, the statement ***x = r**, is executed. This ensures that the pointer **p** now starts pointing to the node, which is the last node of the original list. This is shown in Figure 3-8.

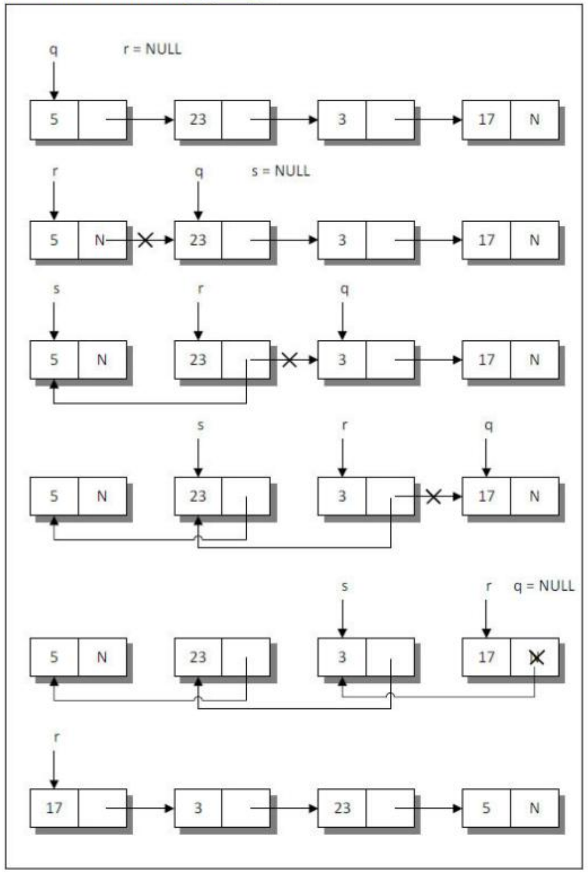
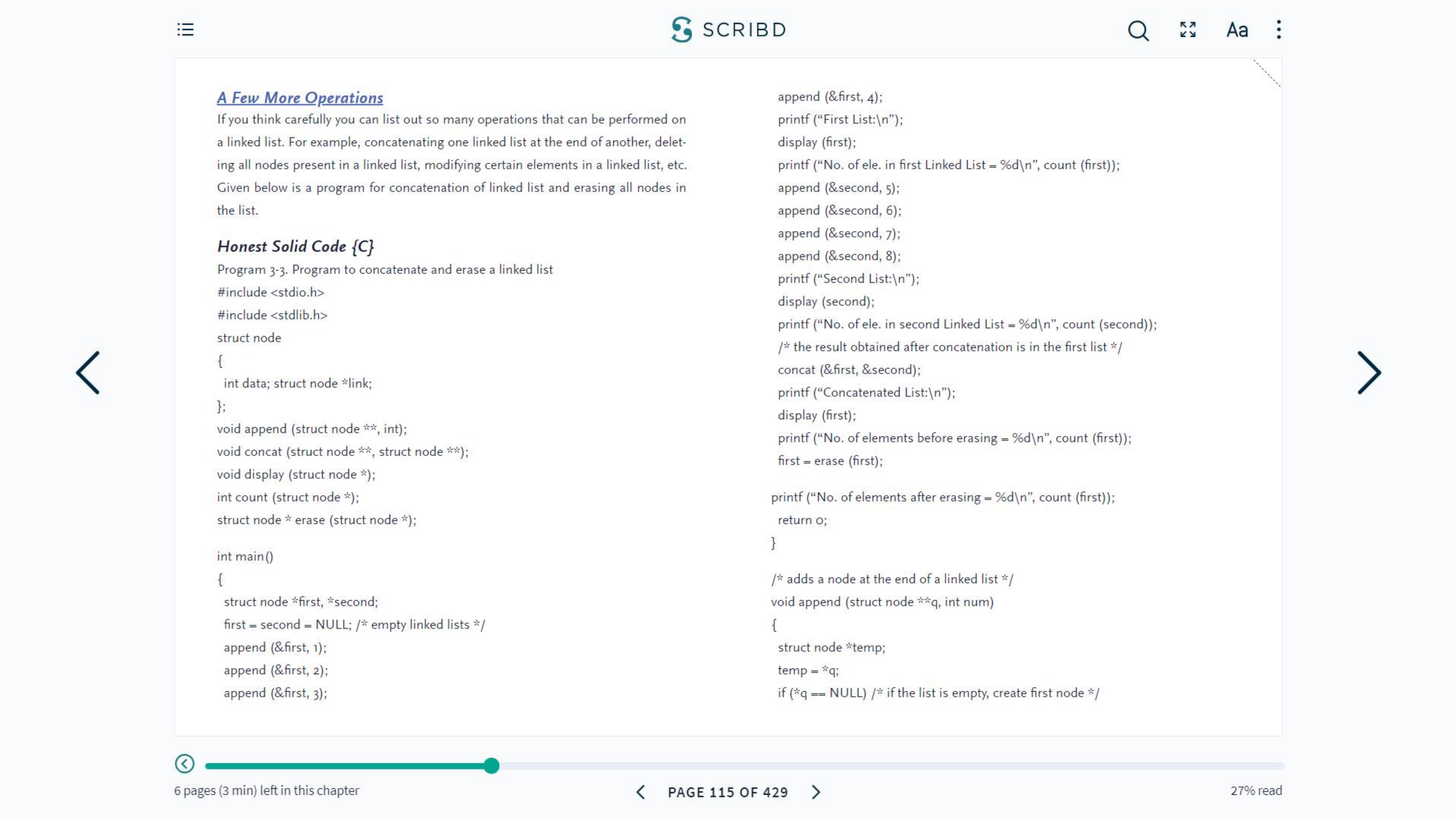


Figure 3-8. Reversing the links.



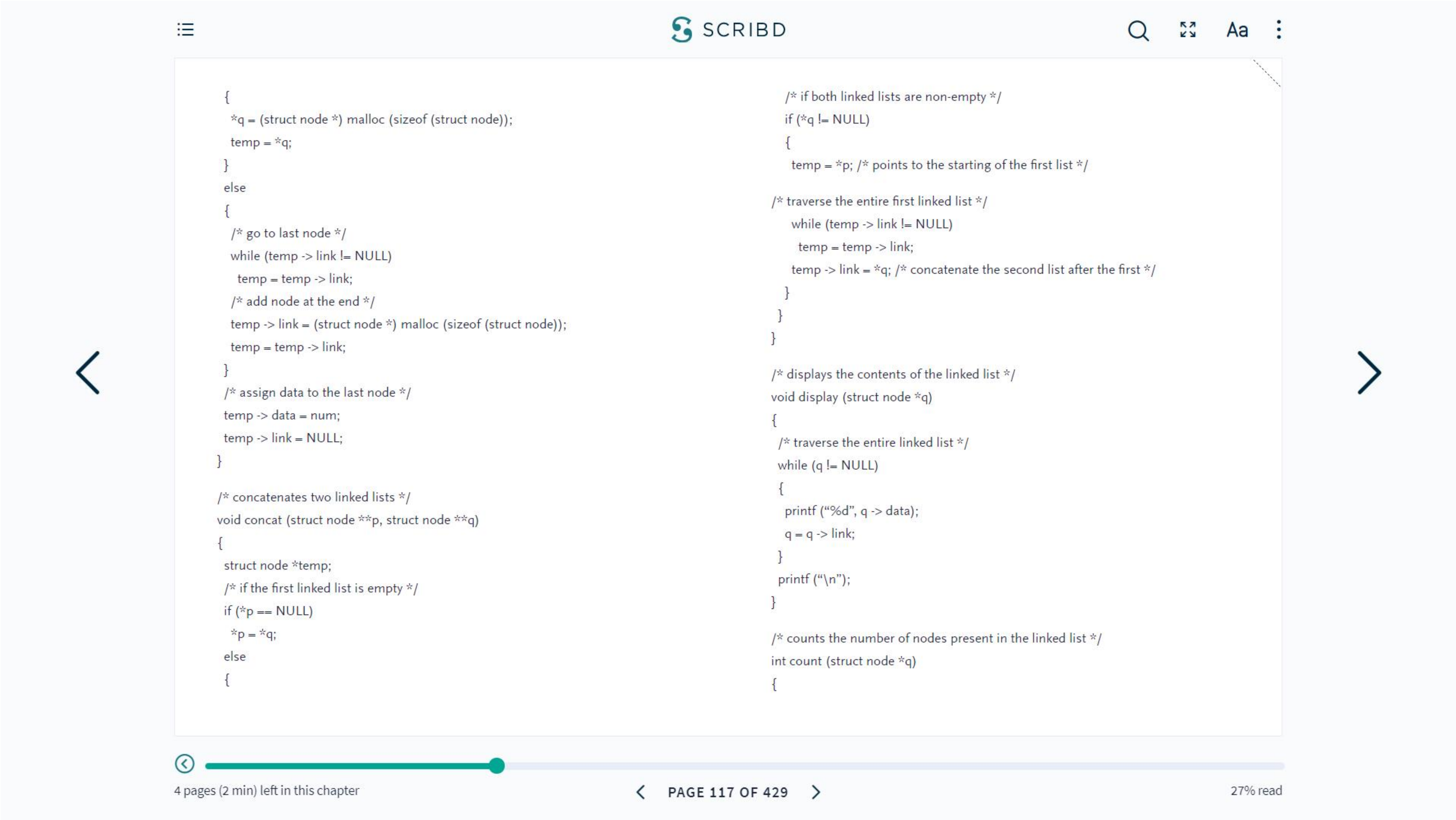
A Few More Operations

If you think carefully you can list out so many operations that can be performed on a linked list. For example, concatenating one linked list at the end of another, deleting all nodes present in a linked list, modifying certain elements in a linked list, etc. Given below is a program for concatenation of linked list and erasing all nodes in the list.

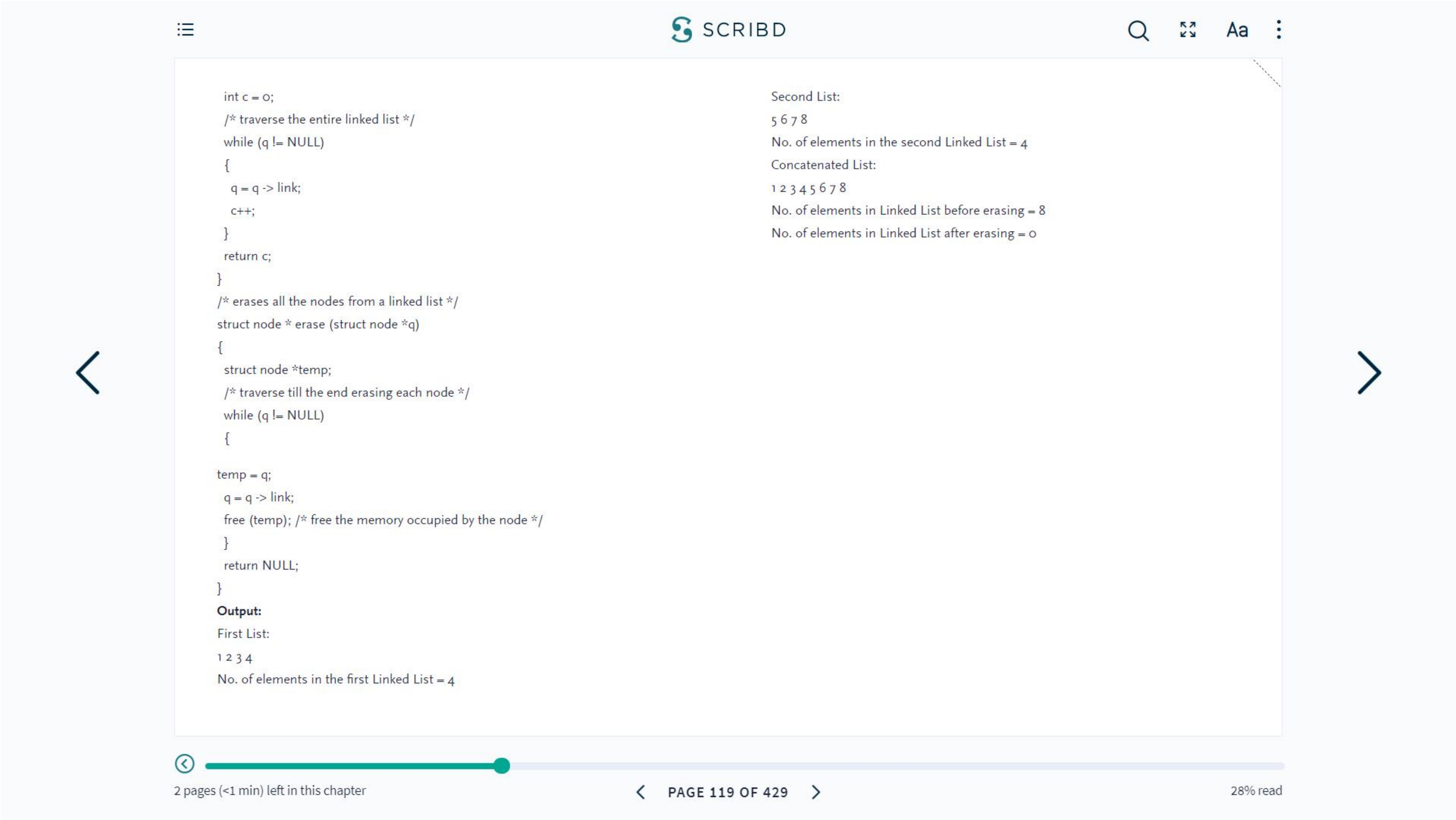
Honest Solid Code {C}

Program 3-3. Program to concatenate and erase a linked list

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
 int data; struct node *link;
};
void append (struct node **, int);
void concat (struct node **, struct node **);
void display (struct node *);
int count (struct node *);
struct node * erase (struct node *);

int main()
{
 struct node *first, *second;
 first = second = NULL; /* empty linked lists */
 append (&first, 1);
 append (&first, 2);
 append (&first, 3);
 append (&first, 4);
 printf (“First List:\n”);
 display (first);
 printf (“No. of ele. in first Linked List = %d\n”, count (first));
 append (&second, 5);
 append (&second, 6);
 append (&second, 7);
 append (&second, 8);
 printf (“Second List:\n”);
 display (second);
 printf (“No. of ele. in second Linked List = %d\n”, count (second));
 /* the result obtained after concatenation is in the first list */
 concat (&first, &second);
 printf (“Concatenated List:\n”);
 display (first);
 printf (“No. of elements before erasing = %d\n”, count (first));
 first = erase (first);

printf (“No. of elements after erasing = %d\n”, count (first));
 return 0;
}

/* adds a node at the end of a linked list */
void append (struct node **q, int num)
{
 struct node *temp;
 temp = *q;
 if (*q == NULL) /* if the list is empty, create first node */
```



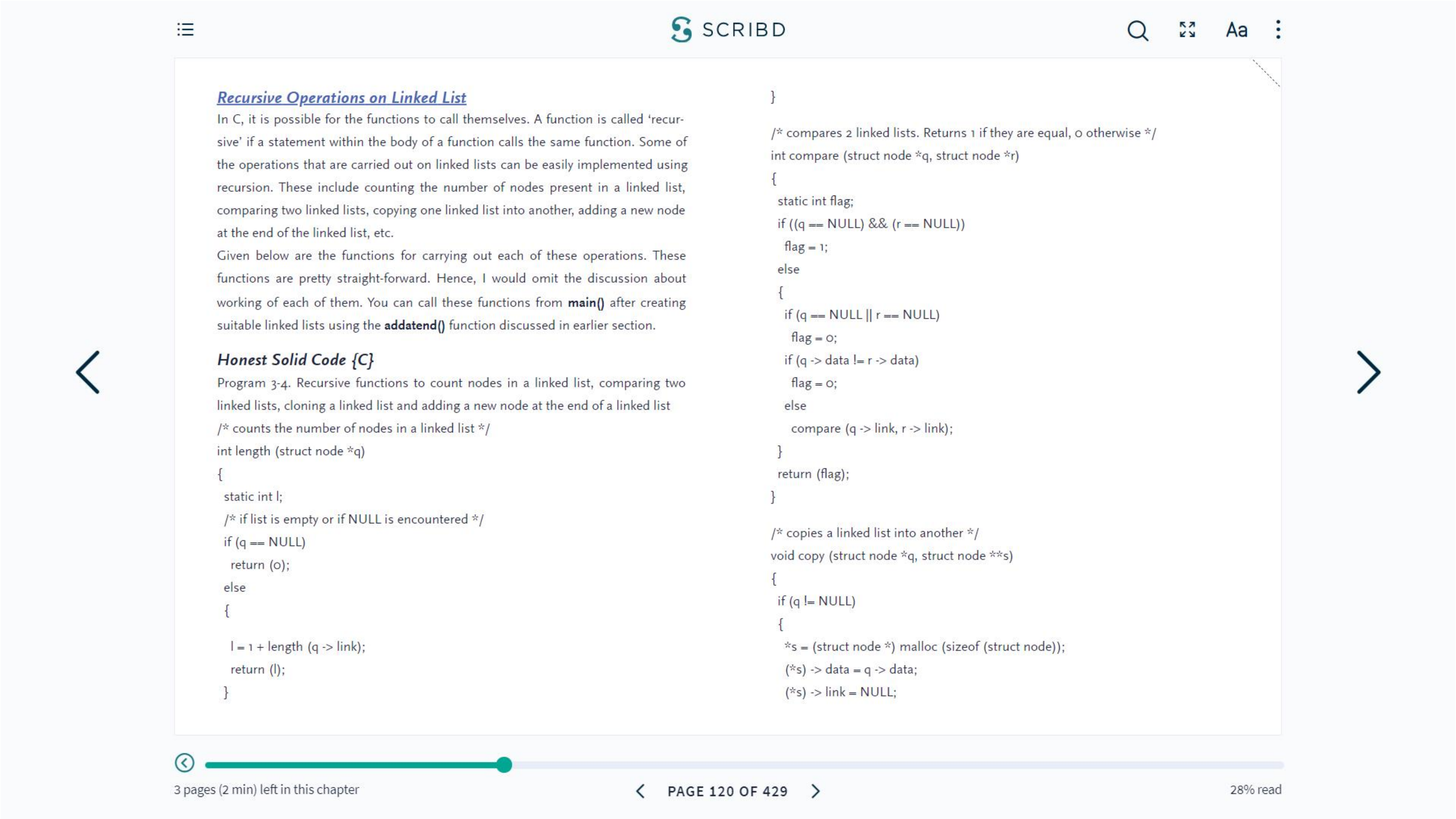
```
 {
  *q = (struct node *) malloc (sizeof (struct node));
  temp = *q;
 }
 else
 {
  /* go to last node */
  while (temp -> link != NULL)
   temp = temp -> link;
  /* add node at the end */
  temp -> link = (struct node *) malloc (sizeof (struct node));
  temp = temp -> link;
 }
 /* assign data to the last node */
 temp -> data = num;
 temp -> link = NULL;
}

/* concatenates two linked lists */
void concat (struct node **p, struct node **q)
{
 struct node *temp;
 /* if the first linked list is empty */
 if (*p == NULL)
  *p = *q;
 else
 {
  /* if both linked lists are non-empty */
  if (*q != NULL)
  {
   temp = *p; /* points to the starting of the first list */

/* traverse the entire first linked list */
   while (temp -> link != NULL)
    temp = temp -> link;
   temp -> link = *q; /* concatenate the second list after the first */
  }
 }
}

/* displays the contents of the linked list */
void display (struct node *q)
{
 /* traverse the entire linked list */
 while (q != NULL)
 {
  printf (“%d”, q -> data);
  q = q -> link;
 }
 printf (“\n”);
}

/* counts the number of nodes present in the linked list */
int count (struct node *q)
{
```



```
  int c = 0;
  /* traverse the entire linked list */
  while (q != NULL)
  {
   q = q -> link;
   c++;
  }
  return c;
 }
 /* erases all the nodes from a linked list */
 struct node * erase (struct node *q)
 {
  struct node *temp;
  /* traverse till the end erasing each node */
  while (q != NULL)
  {

 temp = q;
  q = q -> link;
  free (temp); /* free the memory occupied by the node */
  }
  return NULL;
 }
```

Output:

```
First List:
1 2 3 4
No. of elements in the first Linked List = 4
Second List:
5 6 7 8
No. of elements in the second Linked List = 4
Concatenated List:
1 2 3 4 5 6 7 8
No. of elements in Linked List before erasing = 8
No. of elements in Linked List after erasing = 0
```



Recursive Operations on Linked List

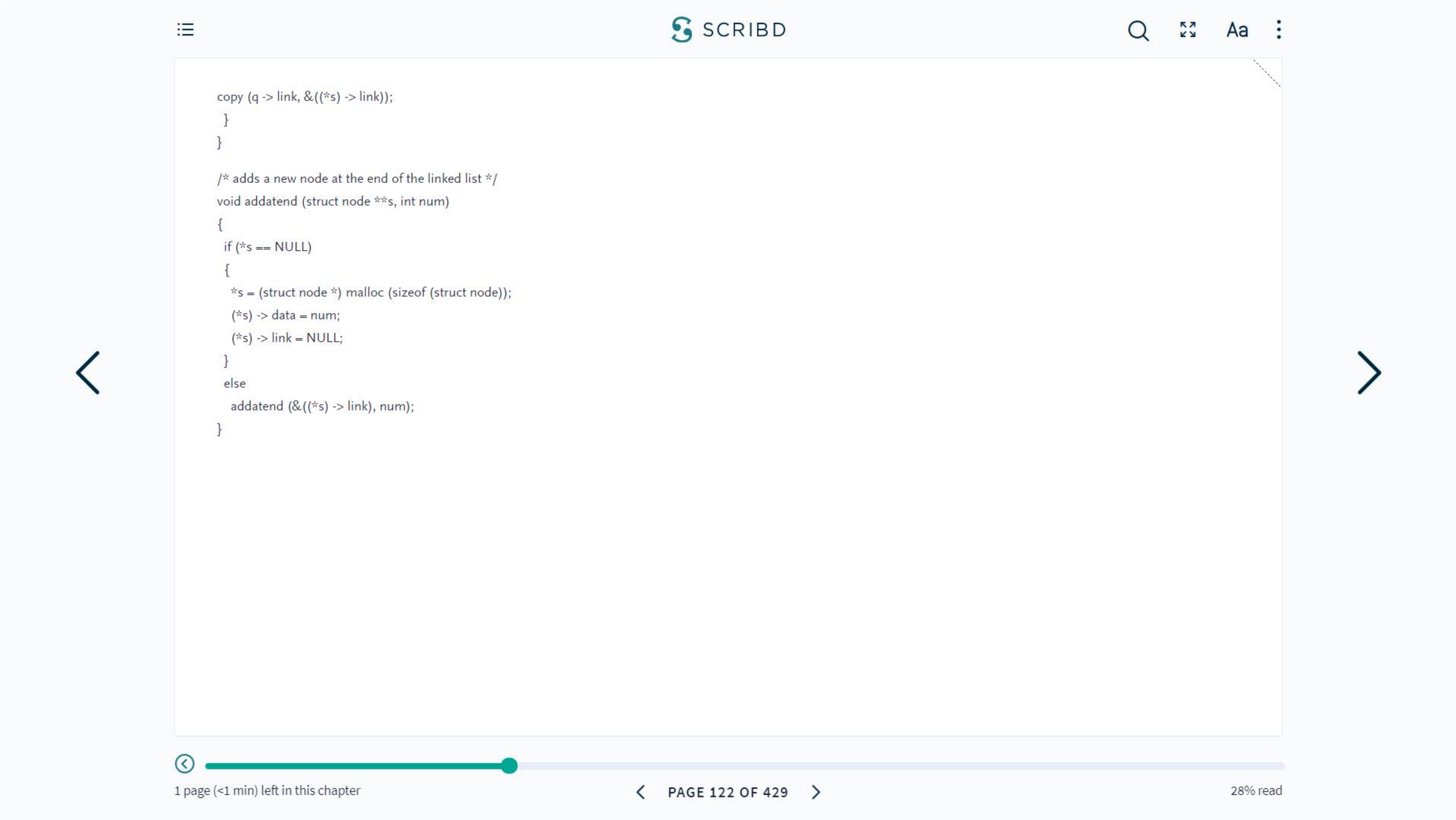
In C, it is possible for the functions to call themselves. A function is called 'recursive' if a statement within the body of a function calls the same function. Some of the operations that are carried out on linked lists can be easily implemented using recursion. These include counting the number of nodes present in a linked list, comparing two linked lists, copying one linked list into another, adding a new node at the end of the linked list, etc.

Given below are the functions for carrying out each of these operations. These functions are pretty straight-forward. Hence, I would omit the discussion about working of each of them. You can call these functions from **main()** after creating suitable linked lists using the **addatend()** function discussed in earlier section.

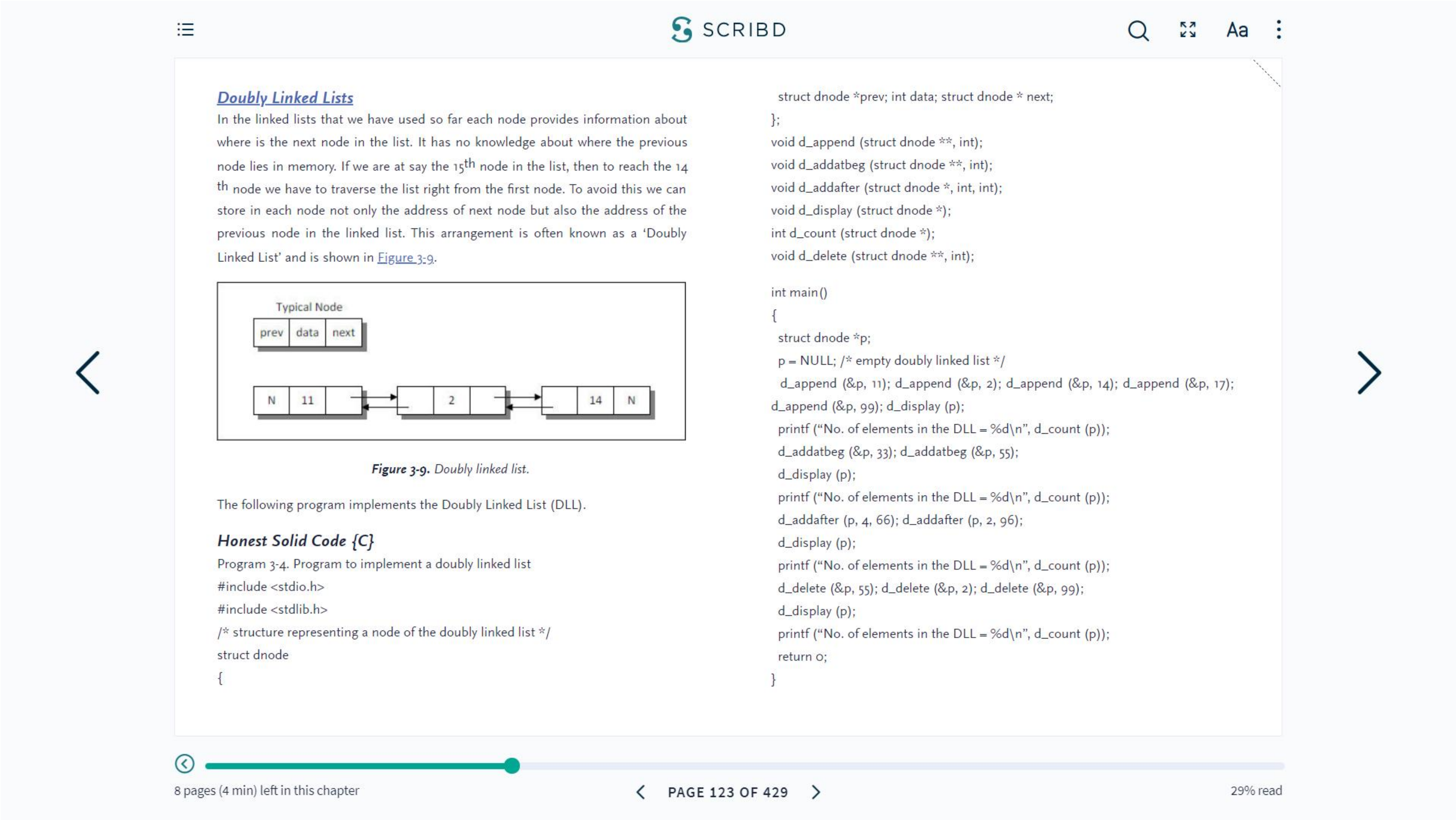
Honest Solid Code {C}

Program 3-4. Recursive functions to count nodes in a linked list, comparing two linked lists, cloning a linked list and adding a new node at the end of a linked list

```
/* counts the number of nodes in a linked list */
int length (struct node *q)
{
 static int l;
 /* if list is empty or if NULL is encountered */
 if (q == NULL)
  return (0);
 else
 {

  l = 1 + length (q -> link);
  return (l);
 }
}

/* compares 2 linked lists. Returns 1 if they are equal, 0 otherwise */
int compare (struct node *q, struct node *r)
{
 static int flag;
 if ((q == NULL) && (r == NULL))
  flag = 1;
 else
 {
  if (q == NULL || r == NULL)
   flag = 0;
  if (q -> data != r -> data)
   flag = 0;
  else
   compare (q -> link, r -> link);
 }
 return (flag);
}

/* copies a linked list into another */
void copy (struct node *q, struct node **s)
{
 if (q != NULL)
 {
  *s = (struct node *) malloc (sizeof (struct node));
  (*s) -> data = q -> data;
  (*s) -> link = NULL;
```



```
copy (q -> link, &((*s) -> link));
 }
}

/* adds a new node at the end of the linked list */
void addatend (struct node **s, int num)
{
 if (*s == NULL)
 {
  *s = (struct node *) malloc (sizeof (struct node));
  (*s) -> data = num;
  (*s) -> link = NULL;
 }
 else
  addatend (&((*s) -> link), num);
}
```



Doubly Linked Lists

In the linked lists that we have used so far each node provides information about where is the next node in the list. It has no knowledge about where the previous node lies in memory. If we are at say the 15th node in the list, then to reach the 14th node we have to traverse the list right from the first node. To avoid this we can store in each node not only the address of next node but also the address of the previous node in the linked list. This arrangement is often known as a 'Doubly Linked List' and is shown in Figure 3-9.

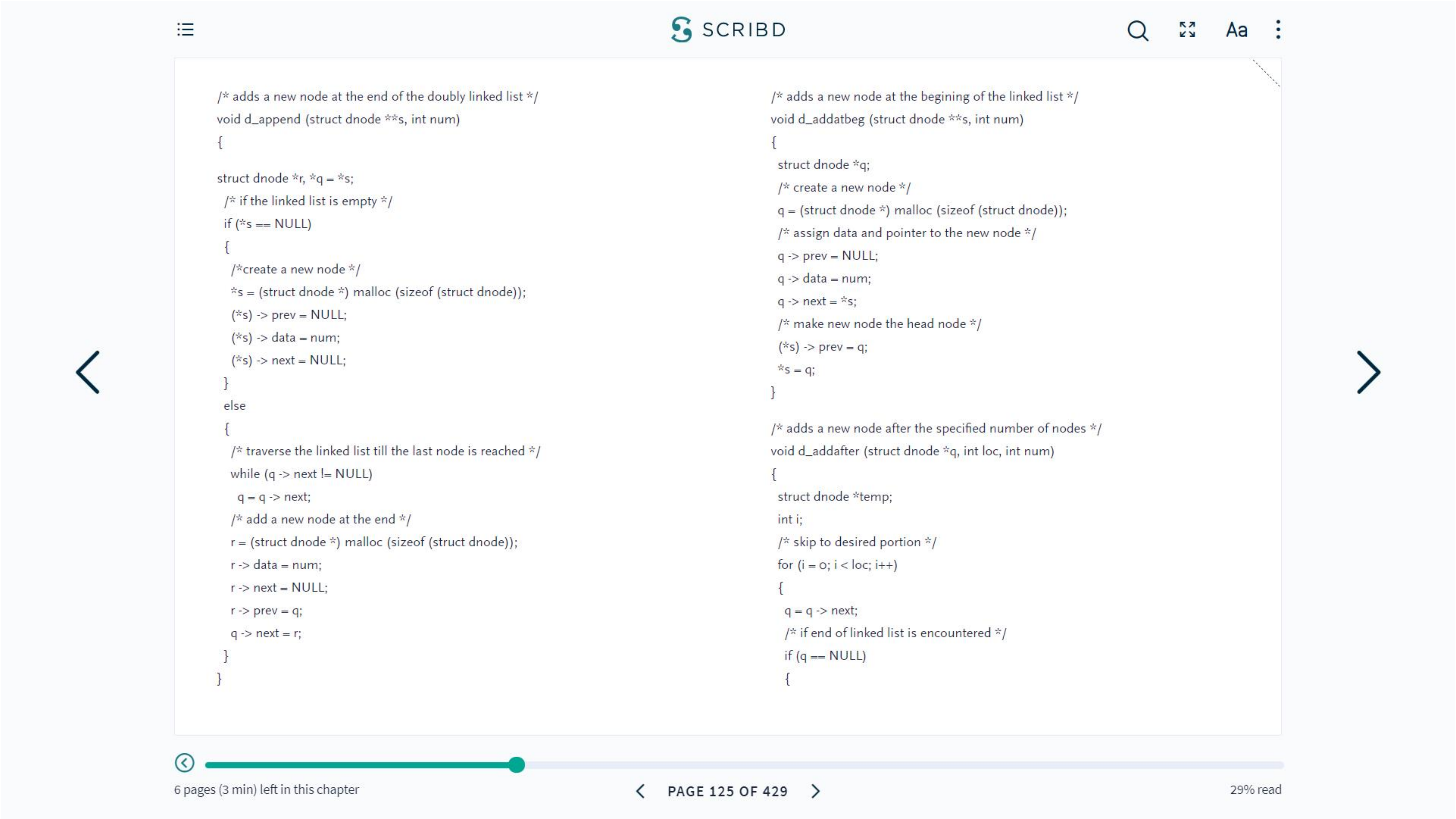
Figure 3-9. *Doubly linked list.*

The following program implements the Doubly Linked List (DLL).

Honest Solid Code {C}

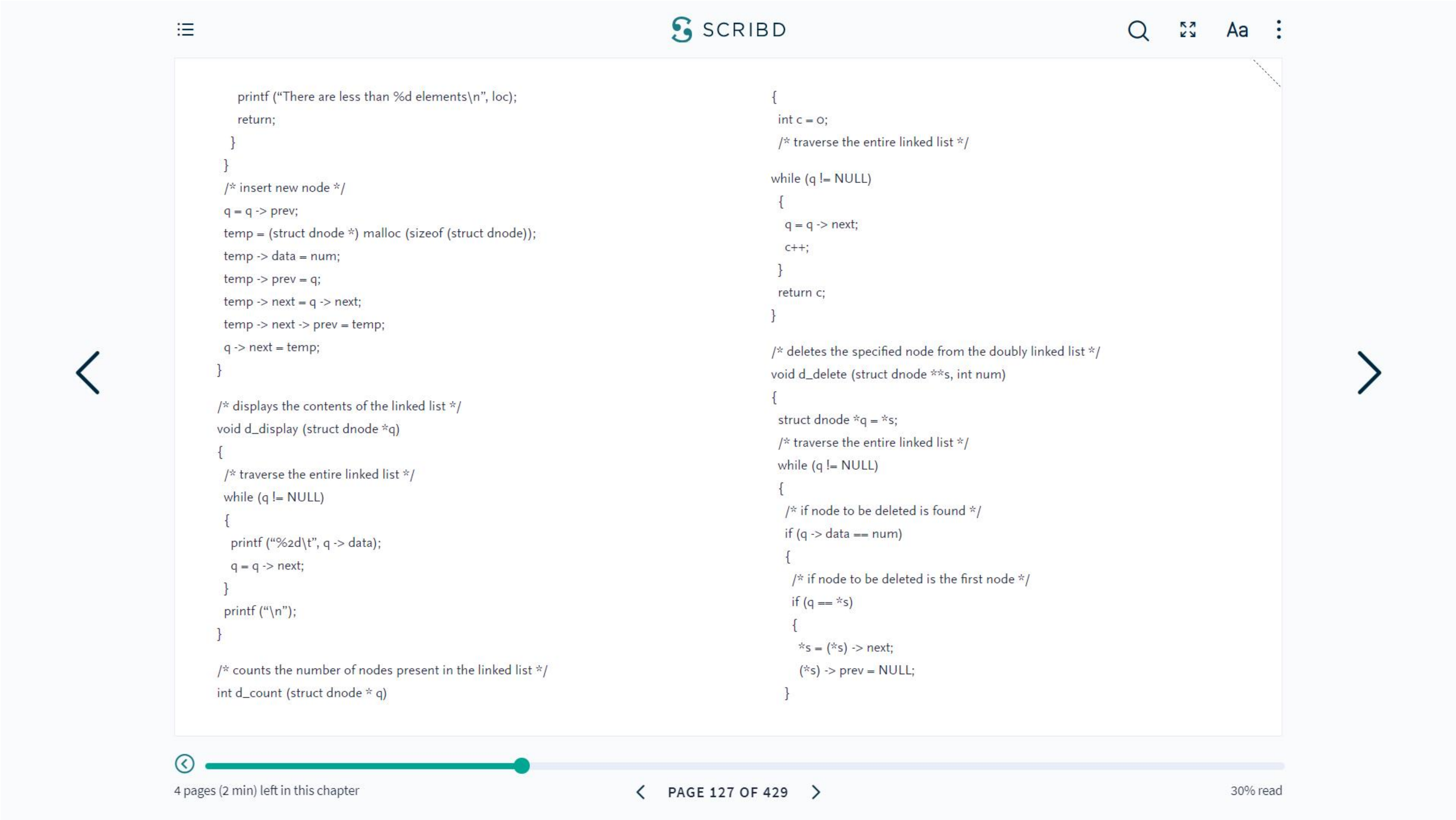
Program 3-4. Program to implement a doubly linked list

```
#include <stdio.h>
#include <stdlib.h>
/* structure representing a node of the doubly linked list */
struct dnode
{
  struct dnode *prev; int data; struct dnode * next;
};
void d_append (struct dnode **, int);
void d_addatbeg (struct dnode **, int);
void d_addafter (struct dnode *, int, int);
void d_display (struct dnode *);
int d_count (struct dnode *);
void d_delete (struct dnode **, int);

int main()
{
  struct dnode *p;
  p = NULL; /* empty doubly linked list */
   d_append (&p, 11); d_append (&p, 2); d_append (&p, 14); d_append (&p, 17);
d_append (&p, 99); d_display (p);
  printf ("No. of elements in the DLL = %d\n", d_count (p));
  d_addatbeg (&p, 33); d_addatbeg (&p, 55);
  d_display (p);
  printf ("No. of elements in the DLL = %d\n", d_count (p));
  d_addafter (p, 4, 66); d_addafter (p, 2, 96);
  d_display (p);
  printf ("No. of elements in the DLL = %d\n", d_count (p));
  d_delete (&p, 55); d_delete (&p, 2); d_delete (&p, 99);
  d_display (p);
  printf ("No. of elements in the DLL = %d\n", d_count (p));
  return 0;
}
```

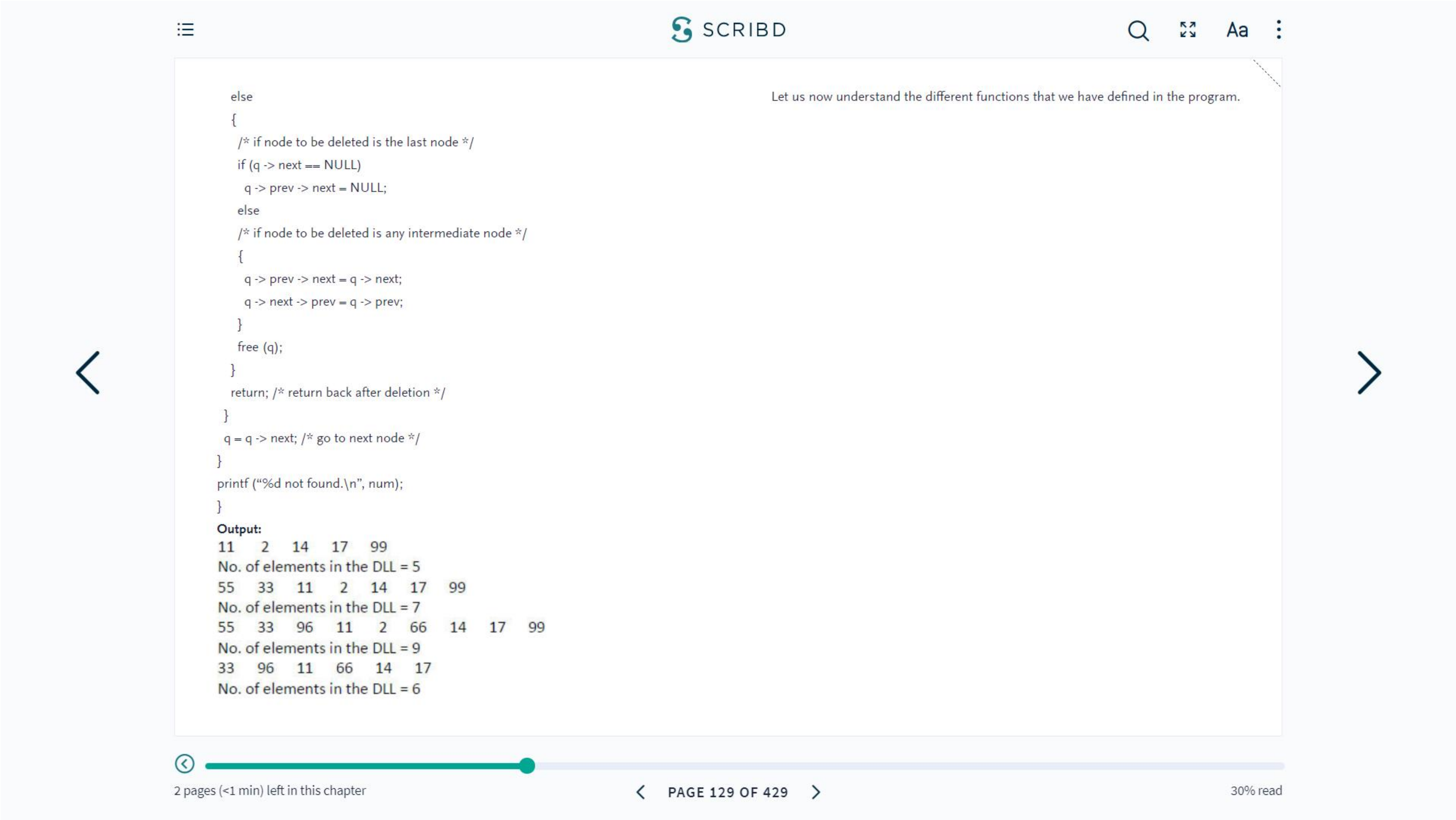


```
/* adds a new node at the end of the doubly linked list */
void d_append (struct dnode **s, int num)
{

struct dnode *r, *q = *s;
 /* if the linked list is empty */
 if (*s == NULL)
 {
  /*create a new node */
  *s = (struct dnode *) malloc (sizeof (struct dnode));
  (*s) -> prev = NULL;
  (*s) -> data = num;
  (*s) -> next = NULL;
 }
 else
 {
  /* traverse the linked list till the last node is reached */
  while (q -> next != NULL)
   q = q -> next;
  /* add a new node at the end */
  r = (struct dnode *) malloc (sizeof (struct dnode));
  r -> data = num;
  r -> next = NULL;
  r -> prev = q;
  q -> next = r;
 }
}
```

```
/* adds a new node at the begining of the linked list */
void d_addatbeg (struct dnode **s, int num)
{
 struct dnode *q;
 /* create a new node */
 q = (struct dnode *) malloc (sizeof (struct dnode));
 /* assign data and pointer to the new node */
 q -> prev = NULL;
 q -> data = num;
 q -> next = *s;
 /* make new node the head node */
 (*s) -> prev = q;
 *s = q;
}

/* adds a new node after the specified number of nodes */
void d_addafter (struct dnode *q, int loc, int num)
{
 struct dnode *temp;
 int i;
 /* skip to desired portion */
 for (i = 0; i < loc; i++)
 {
  q = q -> next;
  /* if end of linked list is encountered */
  if (q == NULL)
  {
```



```
   printf ("There are less than %d elements\n", loc);
   return;
  }
 }
 /* insert new node */
 q = q -> prev;
 temp = (struct dnode *) malloc (sizeof (struct dnode));
 temp -> data = num;
 temp -> prev = q;
 temp -> next = q -> next;
 temp -> next -> prev = temp;
 q -> next = temp;
}

/* displays the contents of the linked list */
void d_display (struct dnode *q)
{
 /* traverse the entire linked list */
 while (q != NULL)
 {
  printf ("%2d\t", q -> data);
  q = q -> next;
 }
 printf ("\n");
}

/* counts the number of nodes present in the linked list */
int d_count (struct dnode * q)
{
 int c = 0;
 /* traverse the entire linked list */

while (q != NULL)
 {
  q = q -> next;
  c++;
 }
 return c;
}

/* deletes the specified node from the doubly linked list */
void d_delete (struct dnode **s, int num)
{
 struct dnode *q = *s;
 /* traverse the entire linked list */
 while (q != NULL)
 {
  /* if node to be deleted is found */
  if (q -> data == num)
  {
   /* if node to be deleted is the first node */
   if (q == *s)
   {
    *s = (*s) -> next;
    (*s) -> prev = NULL;
   }
```

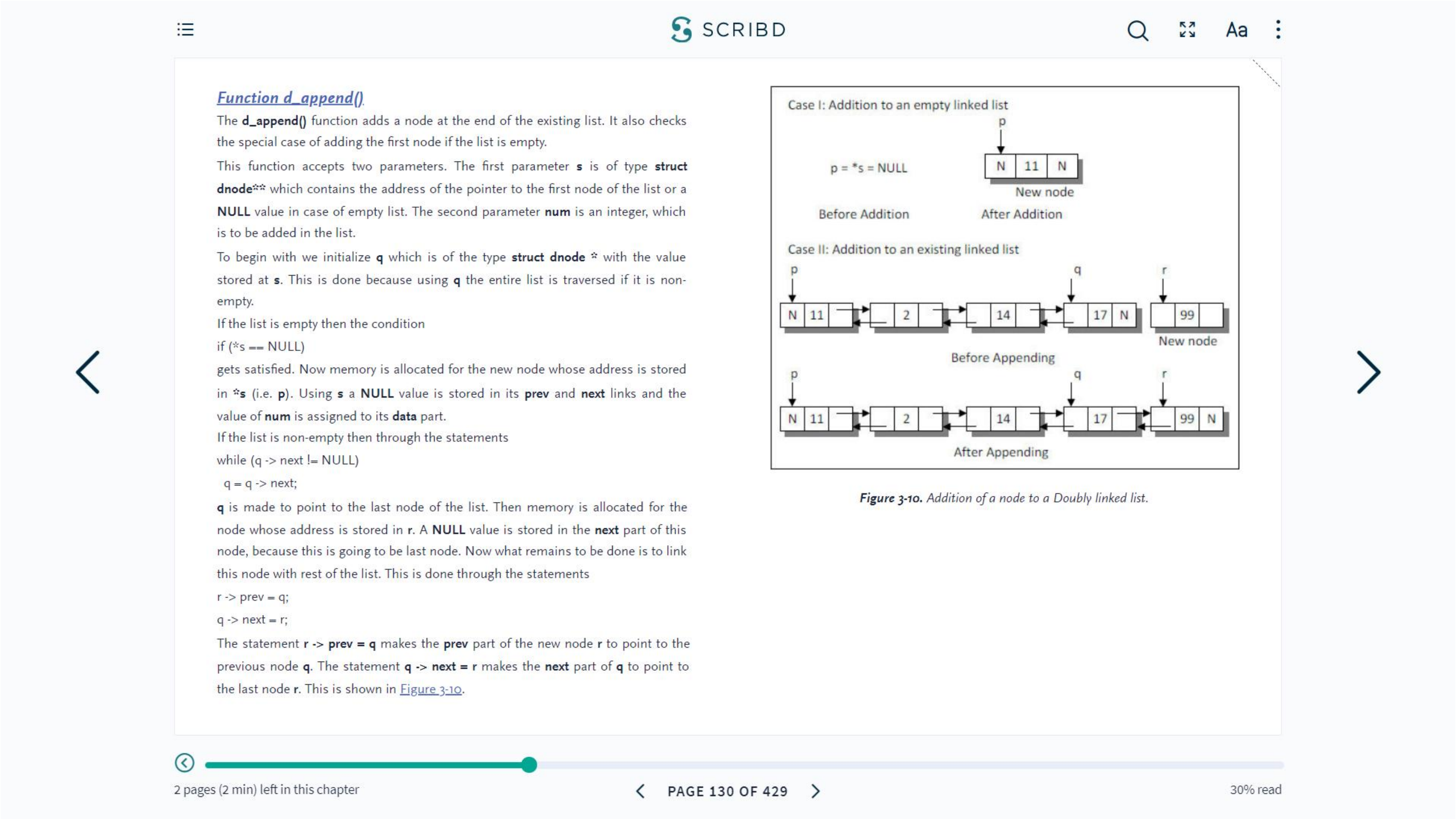


```
    else
    {
     /* if node to be deleted is the last node */
     if (q -> next == NULL)
       q -> prev -> next = NULL;
     else
     /* if node to be deleted is any intermediate node */
     {
       q -> prev -> next = q -> next;
       q -> next -> prev = q -> prev;
     }
     free (q);
    }
    return; /* return back after deletion */
  }
  q = q -> next; /* go to next node */
}
printf ("%d not found.\n", num);
}
```

Output:

```
11   2   14   17   99
No. of elements in the DLL = 5
55   33   11   2   14   17   99
No. of elements in the DLL = 7
55   33   96   11   2   66   14   17   99
No. of elements in the DLL = 9
33   96   11   66   14   17
No. of elements in the DLL = 6
```

Let us now understand the different functions that we have defined in the program.



Function d_append()

The **d_append()** function adds a node at the end of the existing list. It also checks the special case of adding the first node if the list is empty.

This function accepts two parameters. The first parameter **s** is of type **struct dnode**** which contains the address of the pointer to the first node of the list or a **NULL** value in case of empty list. The second parameter **num** is an integer, which is to be added in the list.

To begin with we initialize **q** which is of the type **struct dnode** * with the value stored at **s**. This is done because using **q** the entire list is traversed if it is non-empty.

If the list is empty then the condition

if (*s == NULL)

gets satisfied. Now memory is allocated for the new node whose address is stored in ***s** (i.e. **p**). Using **s** a **NULL** value is stored in its **prev** and **next** links and the value of **num** is assigned to its **data** part.

If the list is non-empty then through the statements

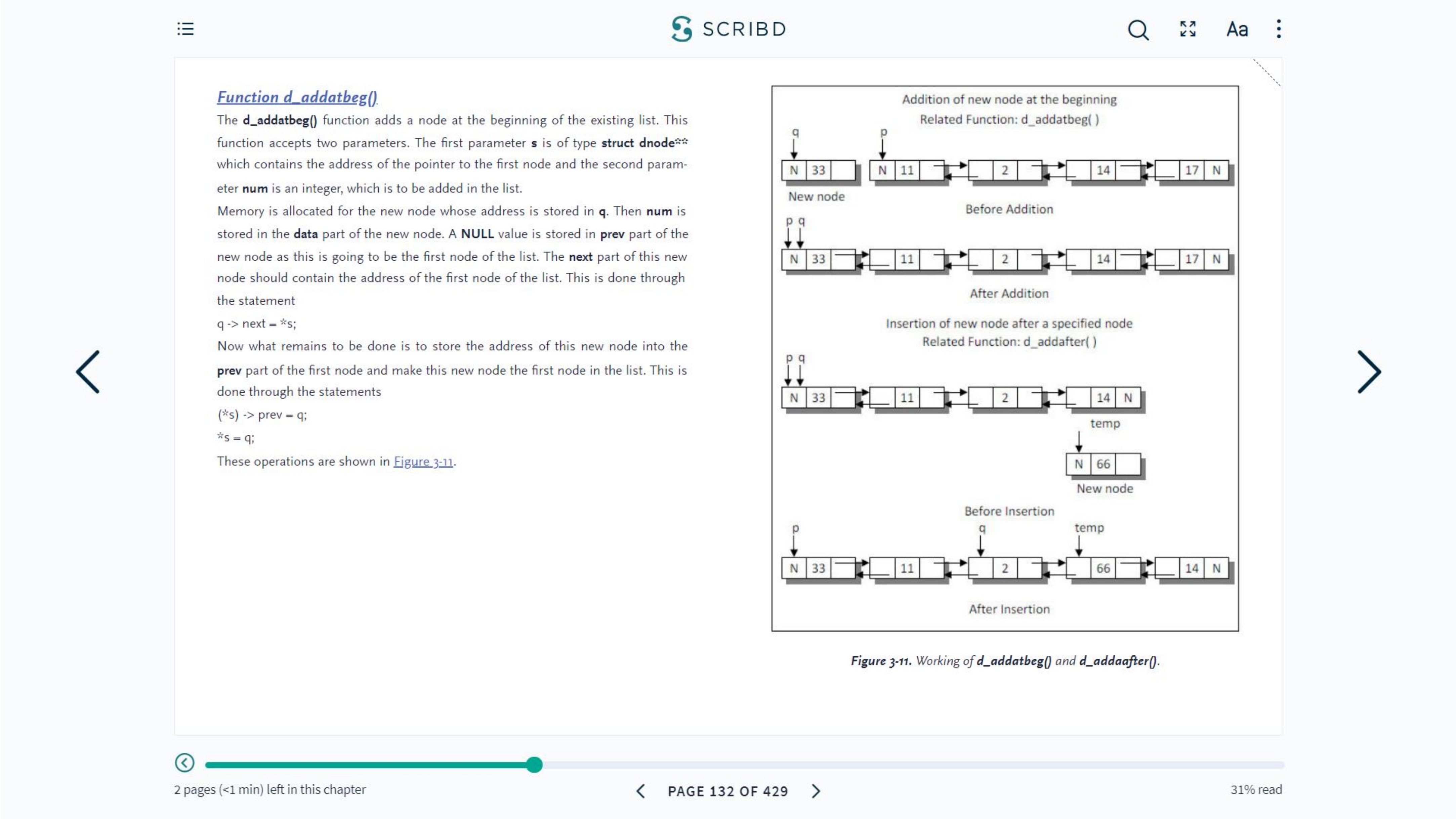
```
while (q -> next != NULL)
 q = q -> next;
```

q is made to point to the last node of the list. Then memory is allocated for the node whose address is stored in **r**. A **NULL** value is stored in the **next** part of this node, because this is going to be last node. Now what remains to be done is to link this node with rest of the list. This is done through the statements

```
r -> prev = q;
q -> next = r;
```

The statement **r -> prev = q** makes the **prev** part of the new node **r** to point to the previous node **q**. The statement **q -> next = r** makes the **next** part of **q** to point to the last node **r**. This is shown in Figure 3-10.

Figure 3-10. *Addition of a node to a Doubly linked list.*



Function d_addatbeg()

The **d_addatbeg()** function adds a node at the beginning of the existing list. This function accepts two parameters. The first parameter **s** is of type **struct dnode**** which contains the address of the pointer to the first node and the second parameter **num** is an integer, which is to be added in the list.

Memory is allocated for the new node whose address is stored in **q**. Then **num** is stored in the **data** part of the new node. A **NULL** value is stored in **prev** part of the new node as this is going to be the first node of the list. The **next** part of this new node should contain the address of the first node of the list. This is done through the statement

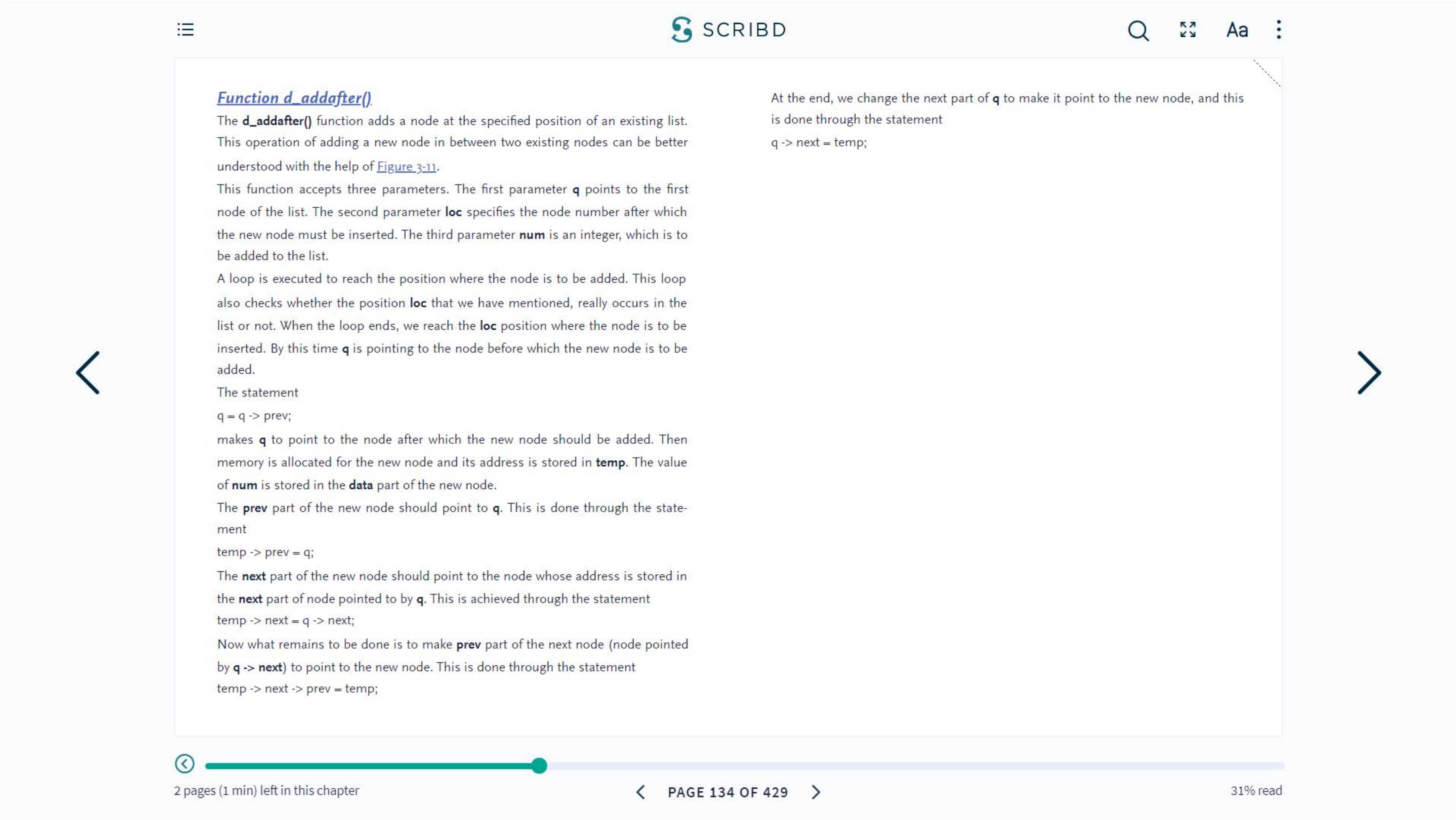
```
q -> next = *s;
```

Now what remains to be done is to store the address of this new node into the **prev** part of the first node and make this new node the first node in the list. This is done through the statements

```
(*s) -> prev = q;
*s = q;
```

These operations are shown in Figure 3-11.

Figure 3-11. Working of d_addatbeg() and d_addaafter().



Function d_addafter()

The **d_addafter()** function adds a node at the specified position of an existing list. This operation of adding a new node in between two existing nodes can be better understood with the help of Figure 3-11.

This function accepts three parameters. The first parameter **q** points to the first node of the list. The second parameter **loc** specifies the node number after which the new node must be inserted. The third parameter **num** is an integer, which is to be added to the list.

A loop is executed to reach the position where the node is to be added. This loop also checks whether the position **loc** that we have mentioned, really occurs in the list or not. When the loop ends, we reach the **loc** position where the node is to be inserted. By this time **q** is pointing to the node before which the new node is to be added.

The statement

```
q = q -> prev;
```

makes **q** to point to the node after which the new node should be added. Then memory is allocated for the new node and its address is stored in **temp**. The value of **num** is stored in the **data** part of the new node.

The **prev** part of the new node should point to **q**. This is done through the statement

```
temp -> prev = q;
```

The **next** part of the new node should point to the node whose address is stored in the **next** part of node pointed to by **q**. This is achieved through the statement

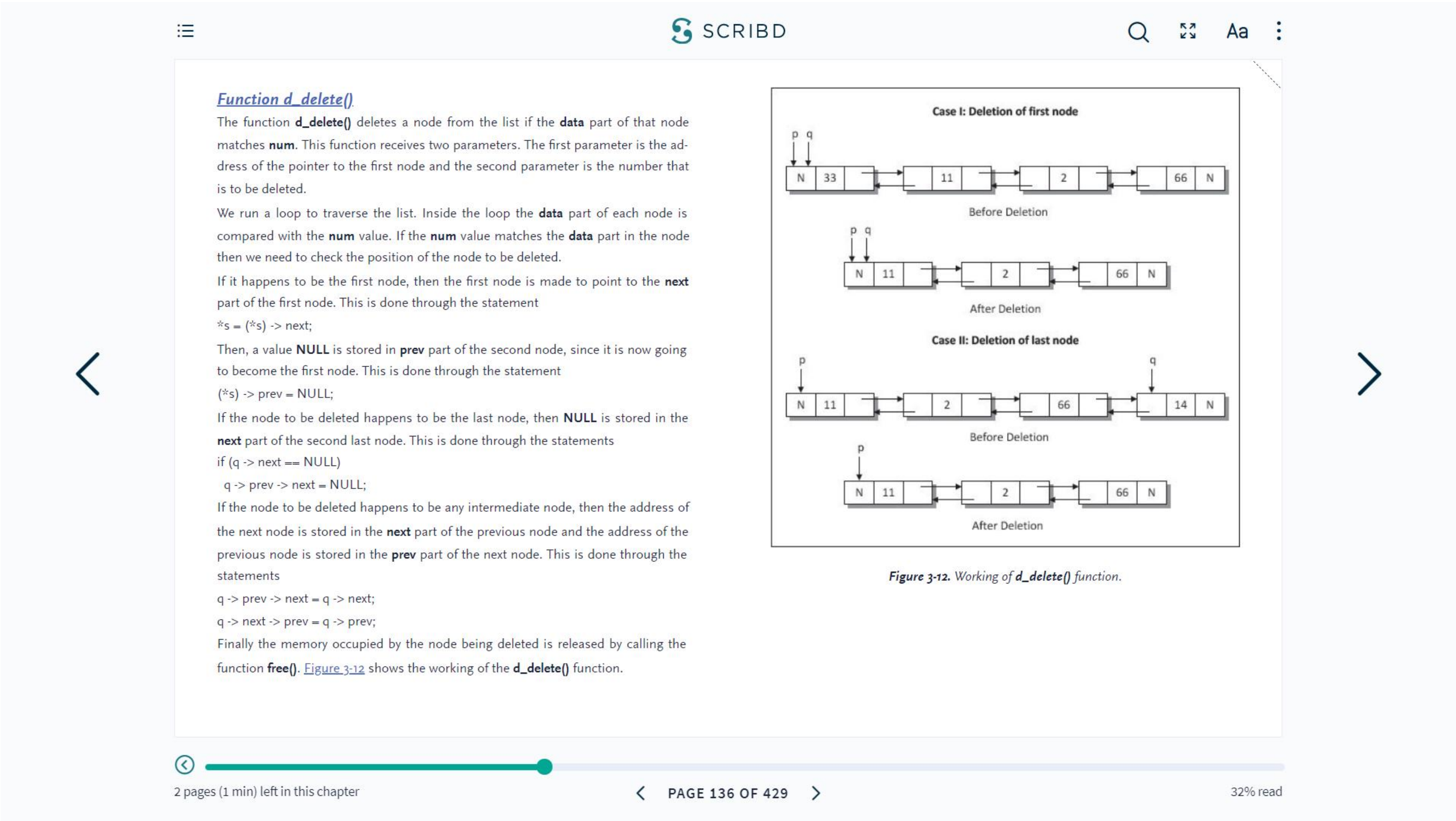
```
temp -> next = q -> next;
```

Now what remains to be done is to make **prev** part of the next node (node pointed by **q -> next**) to point to the new node. This is done through the statement

```
temp -> next -> prev = temp;
```

At the end, we change the next part of **q** to make it point to the new node, and this is done through the statement

```
q -> next = temp;
```



Function d_delete()

The function **d_delete()** deletes a node from the list if the **data** part of that node matches **num**. This function receives two parameters. The first parameter is the address of the pointer to the first node and the second parameter is the number that is to be deleted.

We run a loop to traverse the list. Inside the loop the **data** part of each node is compared with the **num** value. If the **num** value matches the **data** part in the node then we need to check the position of the node to be deleted.

If it happens to be the first node, then the first node is made to point to the **next** part of the first node. This is done through the statement

```
*s = (*s) -> next;
```

Then, a value **NULL** is stored in **prev** part of the second node, since it is now going to become the first node. This is done through the statement

```
(*s) -> prev = NULL;
```

If the node to be deleted happens to be the last node, then **NULL** is stored in the **next** part of the second last node. This is done through the statements

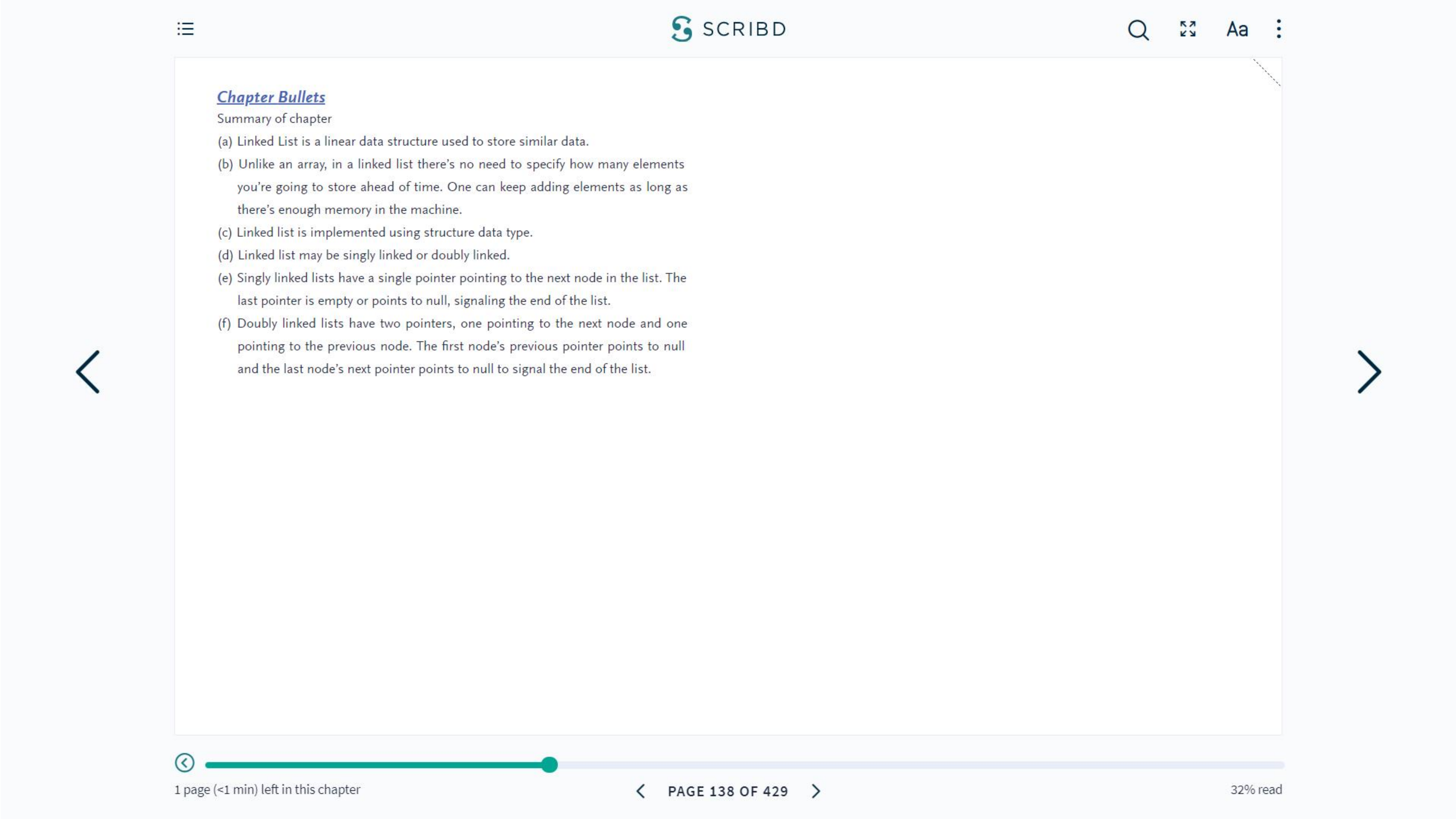
```
if (q -> next == NULL)
  q -> prev -> next = NULL;
```

If the node to be deleted happens to be any intermediate node, then the address of the next node is stored in the **next** part of the previous node and the address of the previous node is stored in the **prev** part of the next node. This is done through the statements

```
q -> prev -> next = q -> next;
q -> next -> prev = q -> prev;
```

Finally the memory occupied by the node being deleted is released by calling the function **free()**. Figure 3-12 shows the working of the **d_delete()** function.

Figure 3-12. *Working of **d_delete()** function.*



Chapter Bullets

Summary of chapter

(a) Linked List is a linear data structure used to store similar data.

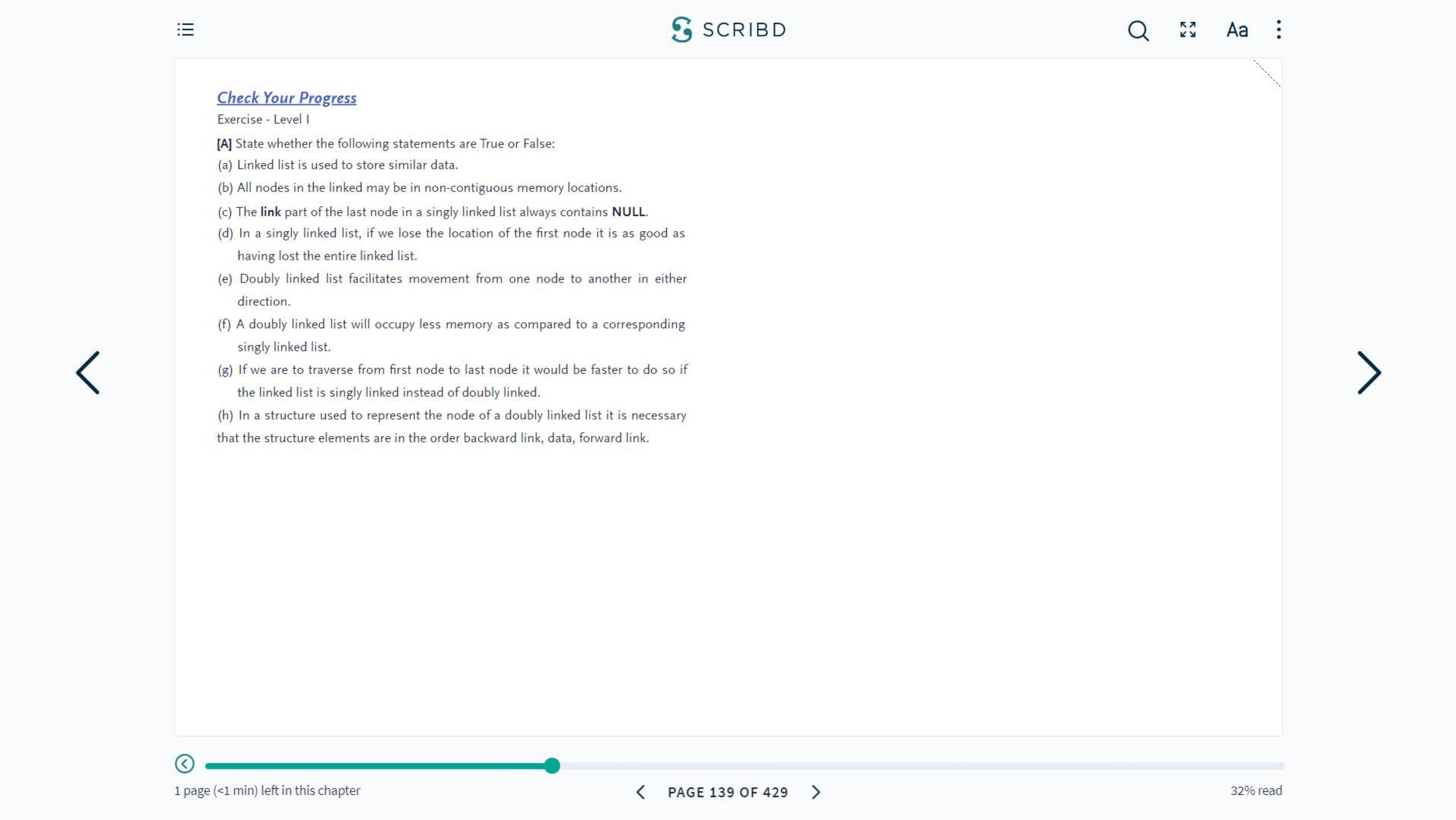
(b) Unlike an array, in a linked list there's no need to specify how many elements you're going to store ahead of time. One can keep adding elements as long as there's enough memory in the machine.

(c) Linked list is implemented using structure data type.

(d) Linked list may be singly linked or doubly linked.

(e) Singly linked lists have a single pointer pointing to the next node in the list. The last pointer is empty or points to null, signaling the end of the list.

(f) Doubly linked lists have two pointers, one pointing to the next node and one pointing to the previous node. The first node's previous pointer points to null and the last node's next pointer points to null to signal the end of the list.



Check Your Progress

Exercise - Level I

[A] State whether the following statements are True or False:

(a) Linked list is used to store similar data.

(b) All nodes in the linked may be in non-contiguous memory locations.

(c) The **link** part of the last node in a singly linked list always contains **NULL**.

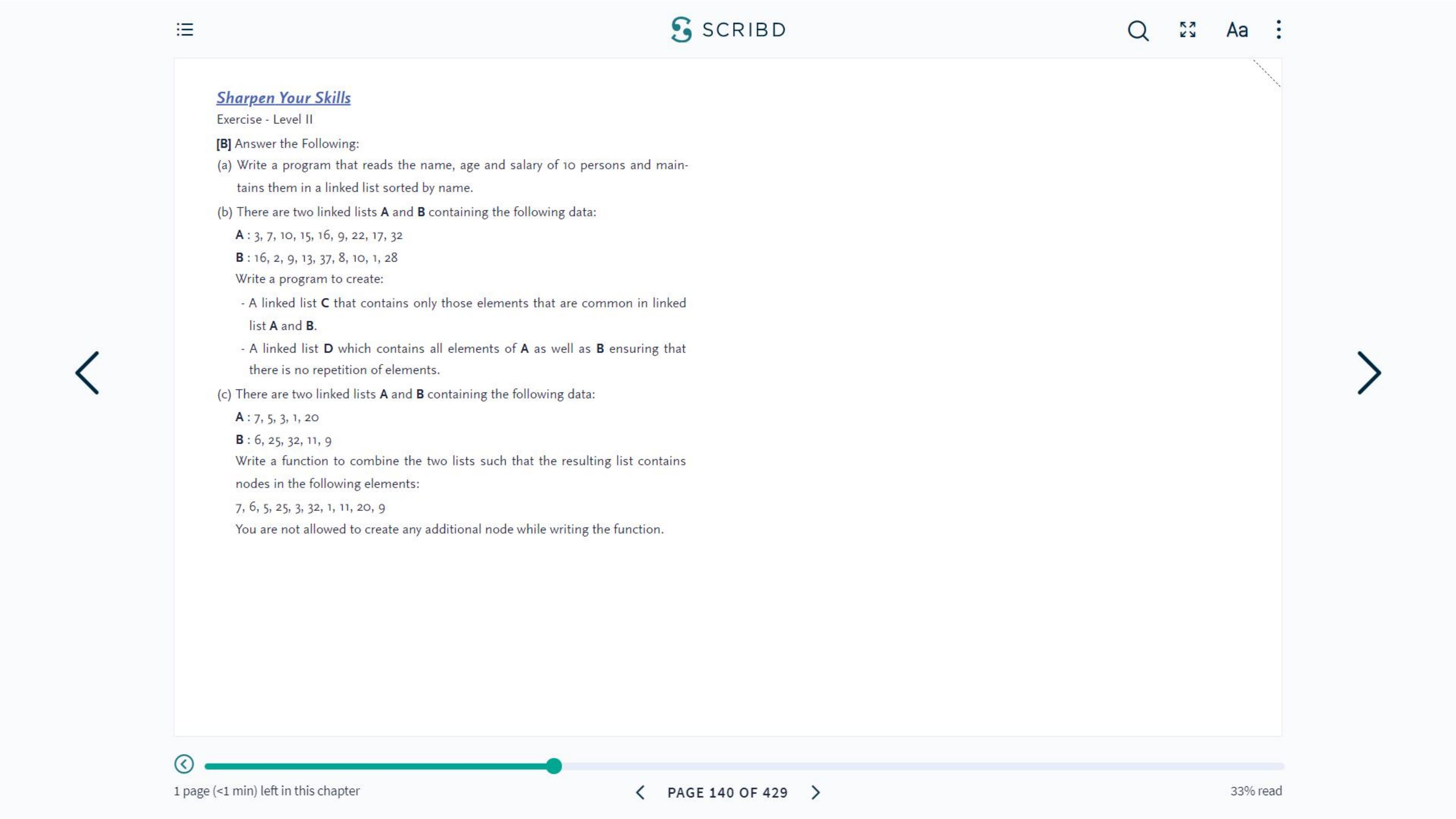
(d) In a singly linked list, if we lose the location of the first node it is as good as having lost the entire linked list.

(e) Doubly linked list facilitates movement from one node to another in either direction.

(f) A doubly linked list will occupy less memory as compared to a corresponding singly linked list.

(g) If we are to traverse from first node to last node it would be faster to do so if the linked list is singly linked instead of doubly linked.

(h) In a structure used to represent the node of a doubly linked list it is necessary that the structure elements are in the order backward link, data, forward link.



Sharpen Your Skills

Exercise - Level II

[B] Answer the Following:

(a) Write a program that reads the name, age and salary of 10 persons and maintains them in a linked list sorted by name.

(b) There are two linked lists **A** and **B** containing the following data:

A : 3, 7, 10, 15, 16, 9, 22, 17, 32

B : 16, 2, 9, 13, 37, 8, 10, 1, 28

Write a program to create:

- A linked list **C** that contains only those elements that are common in linked list **A** and **B**.
- A linked list **D** which contains all elements of **A** as well as **B** ensuring that there is no repetition of elements.

(c) There are two linked lists **A** and **B** containing the following data:

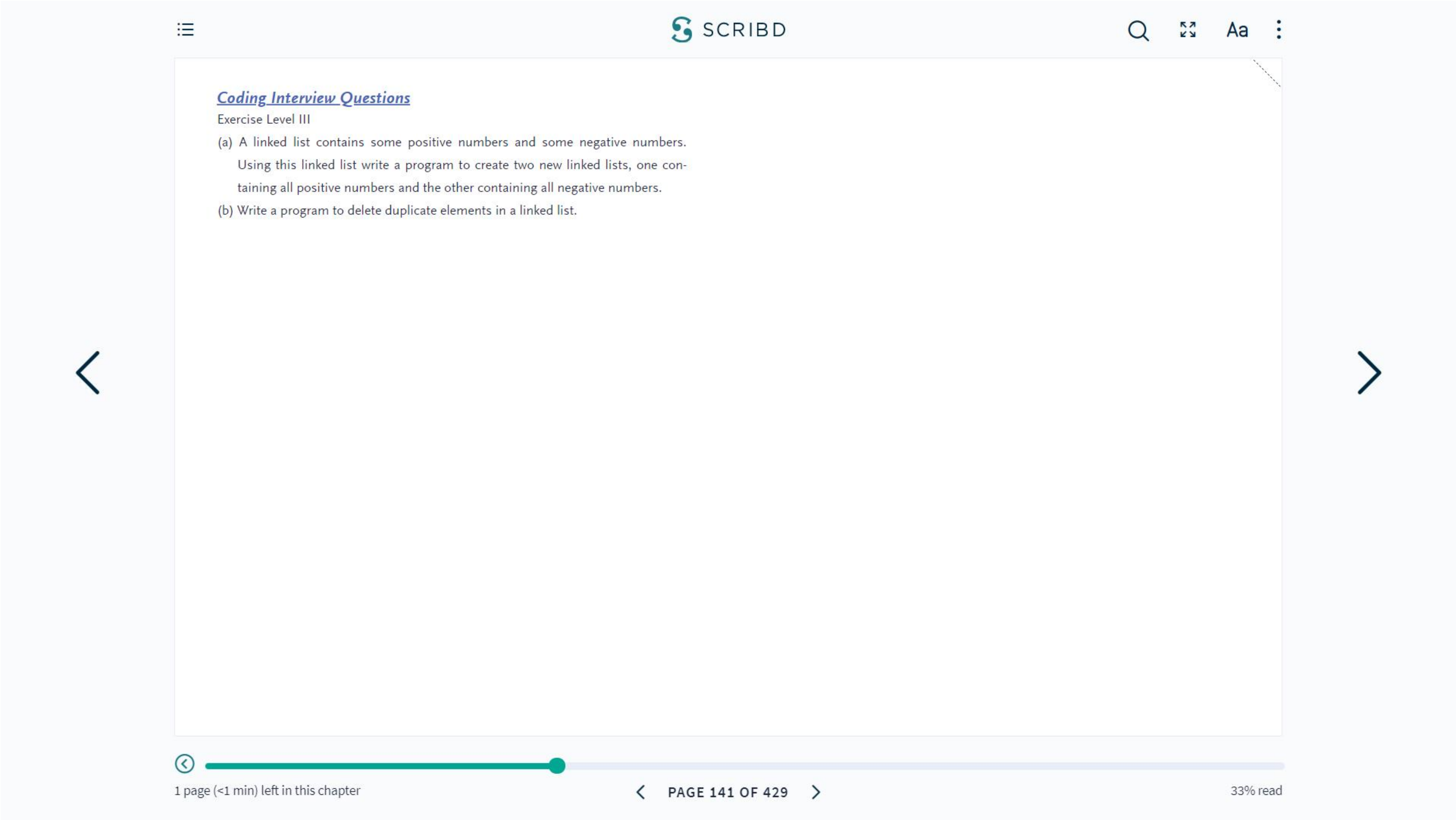
A : 7, 5, 3, 1, 20

B : 6, 25, 32, 11, 9

Write a function to combine the two lists such that the resulting list contains nodes in the following elements:

7, 6, 5, 25, 3, 32, 1, 11, 20, 9

You are not allowed to create any additional node while writing the function.

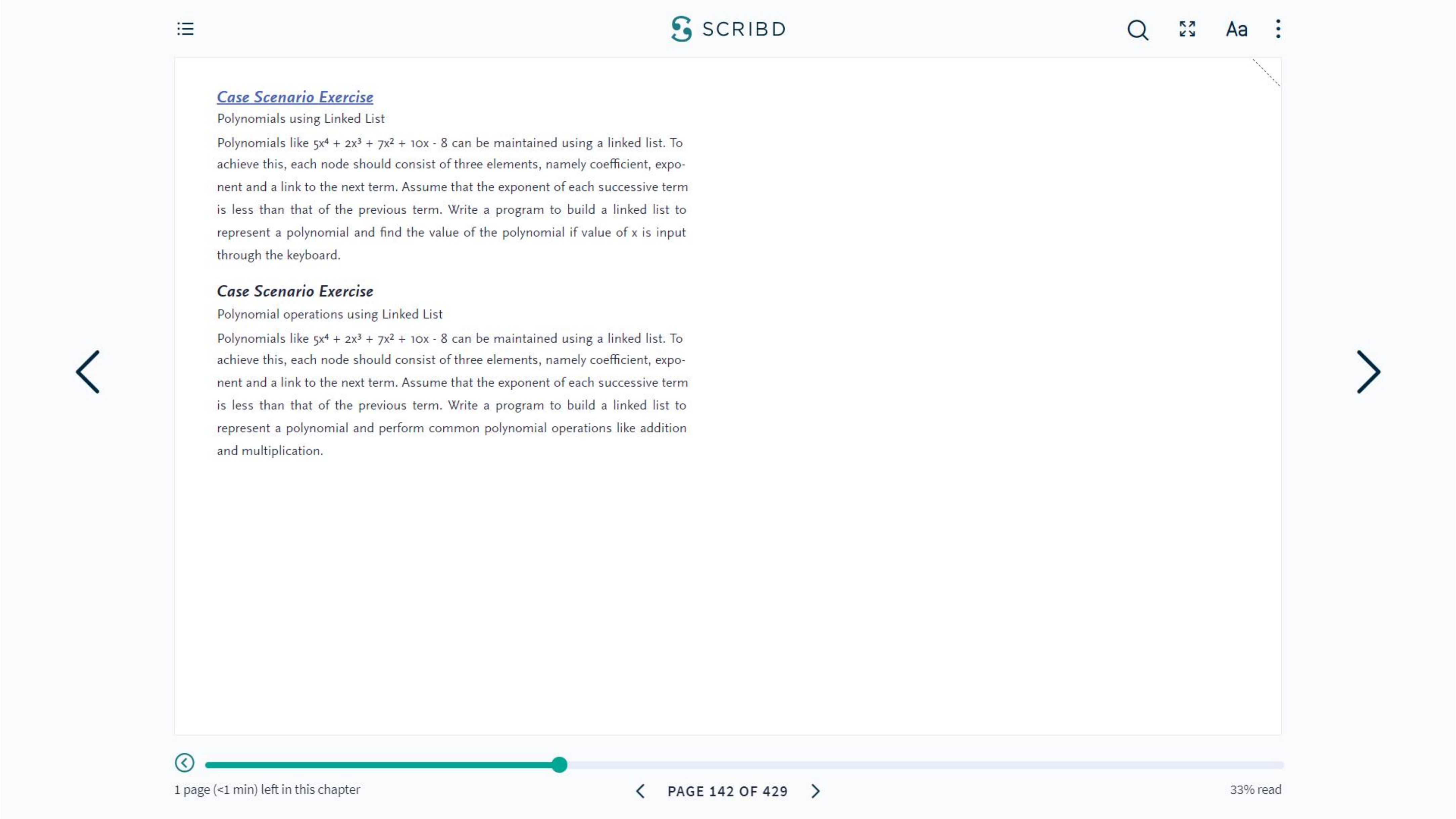


Coding Interview Questions

Exercise Level III

(a) A linked list contains some positive numbers and some negative numbers. Using this linked list write a program to create two new linked lists, one containing all positive numbers and the other containing all negative numbers.

(b) Write a program to delete duplicate elements in a linked list.



Case Scenario Exercise

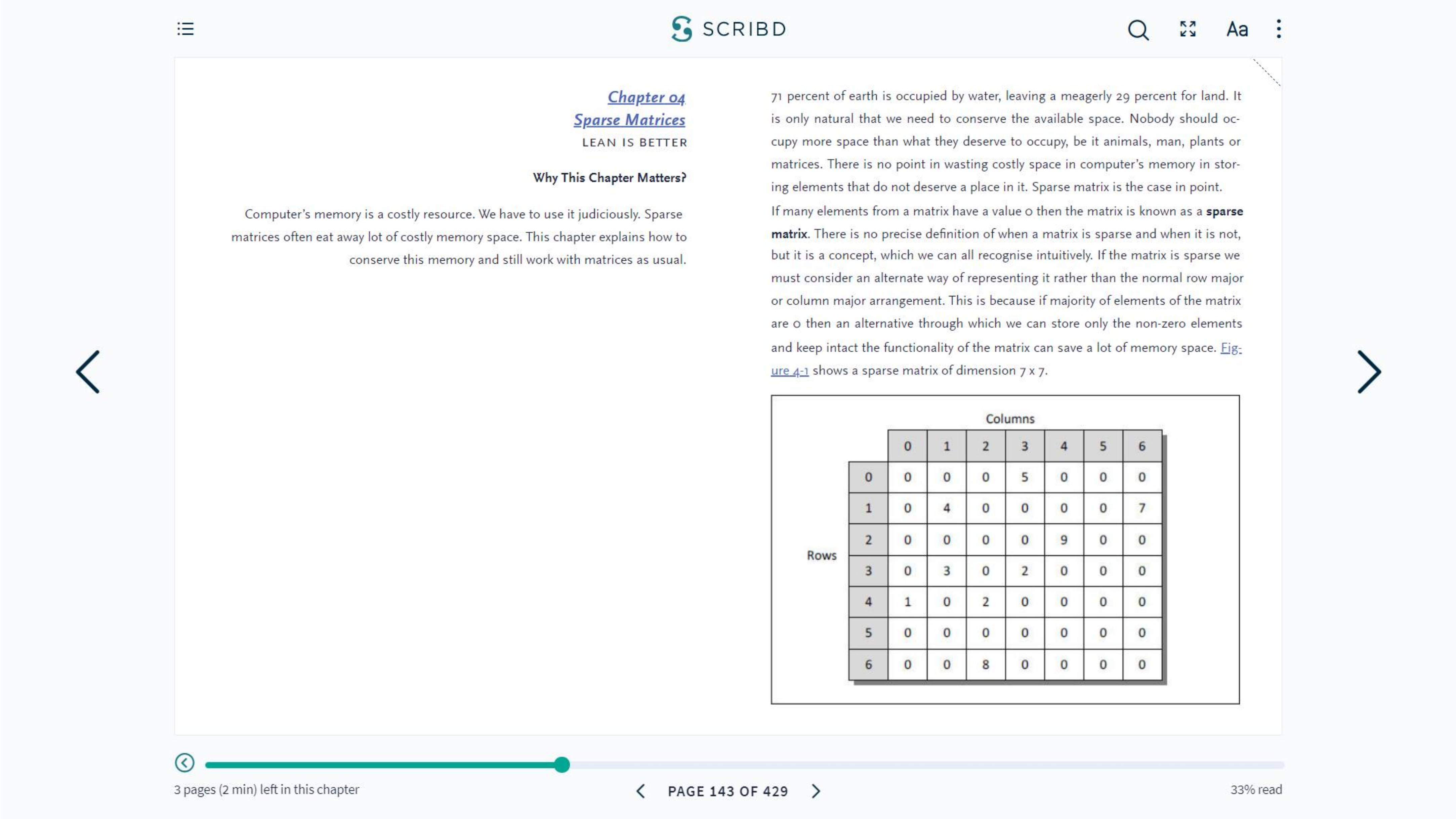
Polynomials using Linked List

Polynomials like $5x^4 + 2x^3 + 7x^2 + 10x - 8$ can be maintained using a linked list. To achieve this, each node should consist of three elements, namely coefficient, exponent and a link to the next term. Assume that the exponent of each successive term is less than that of the previous term. Write a program to build a linked list to represent a polynomial and find the value of the polynomial if value of x is input through the keyboard.

Case Scenario Exercise

Polynomial operations using Linked List

Polynomials like $5x^4 + 2x^3 + 7x^2 + 10x - 8$ can be maintained using a linked list. To achieve this, each node should consist of three elements, namely coefficient, exponent and a link to the next term. Assume that the exponent of each successive term is less than that of the previous term. Write a program to build a linked list to represent a polynomial and perform common polynomial operations like addition and multiplication.



Chapter 04
Sparse Matrices

LEAN IS BETTER

Why This Chapter Matters?

Computer's memory is a costly resource. We have to use it judiciously. Sparse matrices often eat away lot of costly memory space. This chapter explains how to conserve this memory and still work with matrices as usual.

71 percent of earth is occupied by water, leaving a meagerly 29 percent for land. It is only natural that we need to conserve the available space. Nobody should occupy more space than what they deserve to occupy, be it animals, man, plants or matrices. There is no point in wasting costly space in computer's memory in storing elements that do not deserve a place in it. Sparse matrix is the case in point.

If many elements from a matrix have a value 0 then the matrix is known as a **sparse matrix**. There is no precise definition of when a matrix is sparse and when it is not, but it is a concept, which we can all recognise intuitively. If the matrix is sparse we must consider an alternate way of representing it rather than the normal row major or column major arrangement. This is because if majority of elements of the matrix are 0 then an alternative through which we can store only the non-zero elements and keep intact the functionality of the matrix can save a lot of memory space. Figure 4-1 shows a sparse matrix of dimension 7 x 7.

Columns

Rows	0	1	2	3	4	5	6
0	0	0	0	5	0	0	0
1	0	4	0	0	0	0	7
2	0	0	0	0	9	0	0
3	0	3	0	2	0	0	0
4	1	0	2	0	0	0	0
5	0	0	0	0	0	0	0
6	0	0	8	0	0	0	0

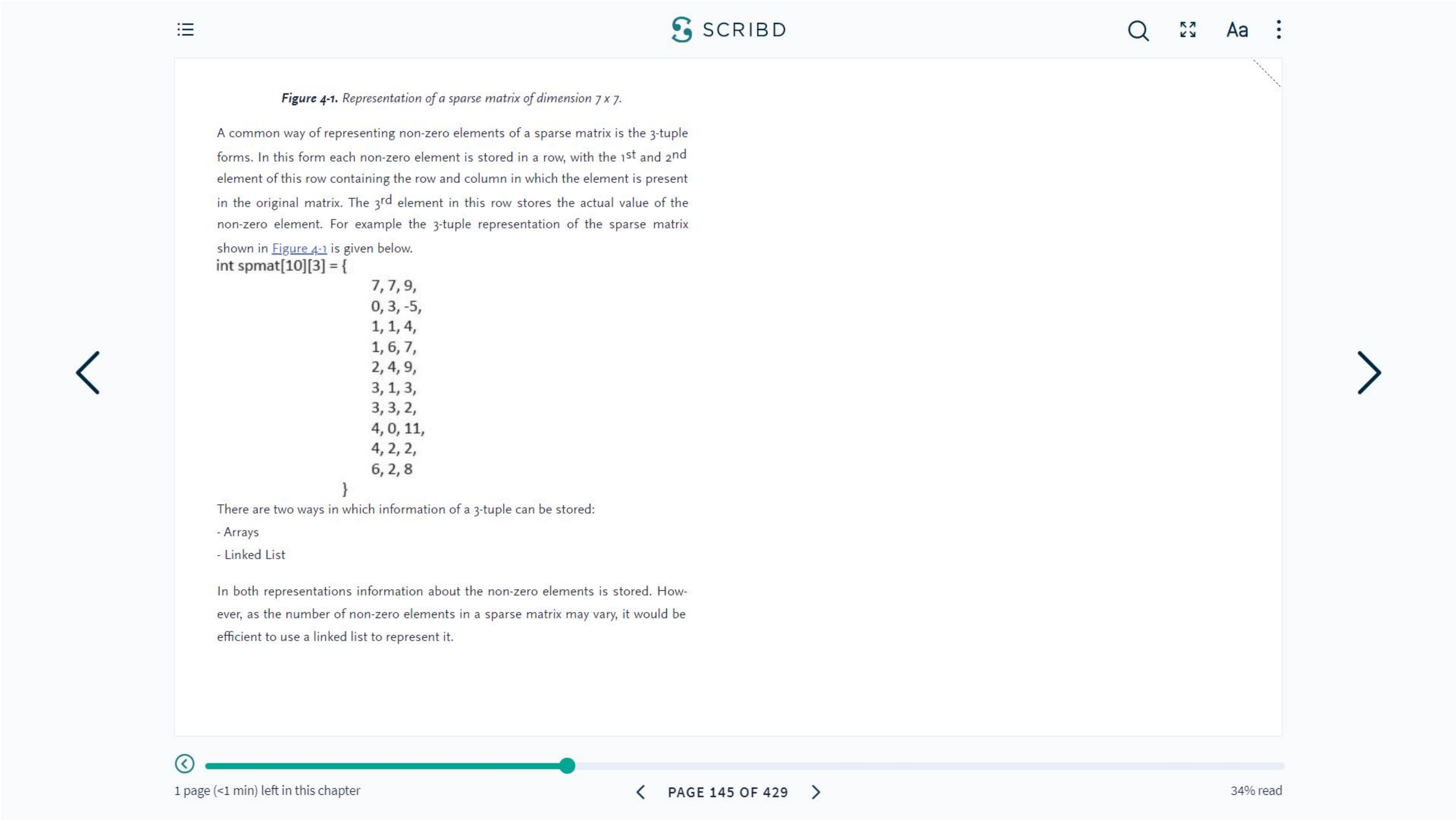


Figure 4-1. Representation of a sparse matrix of dimension 7 x 7.

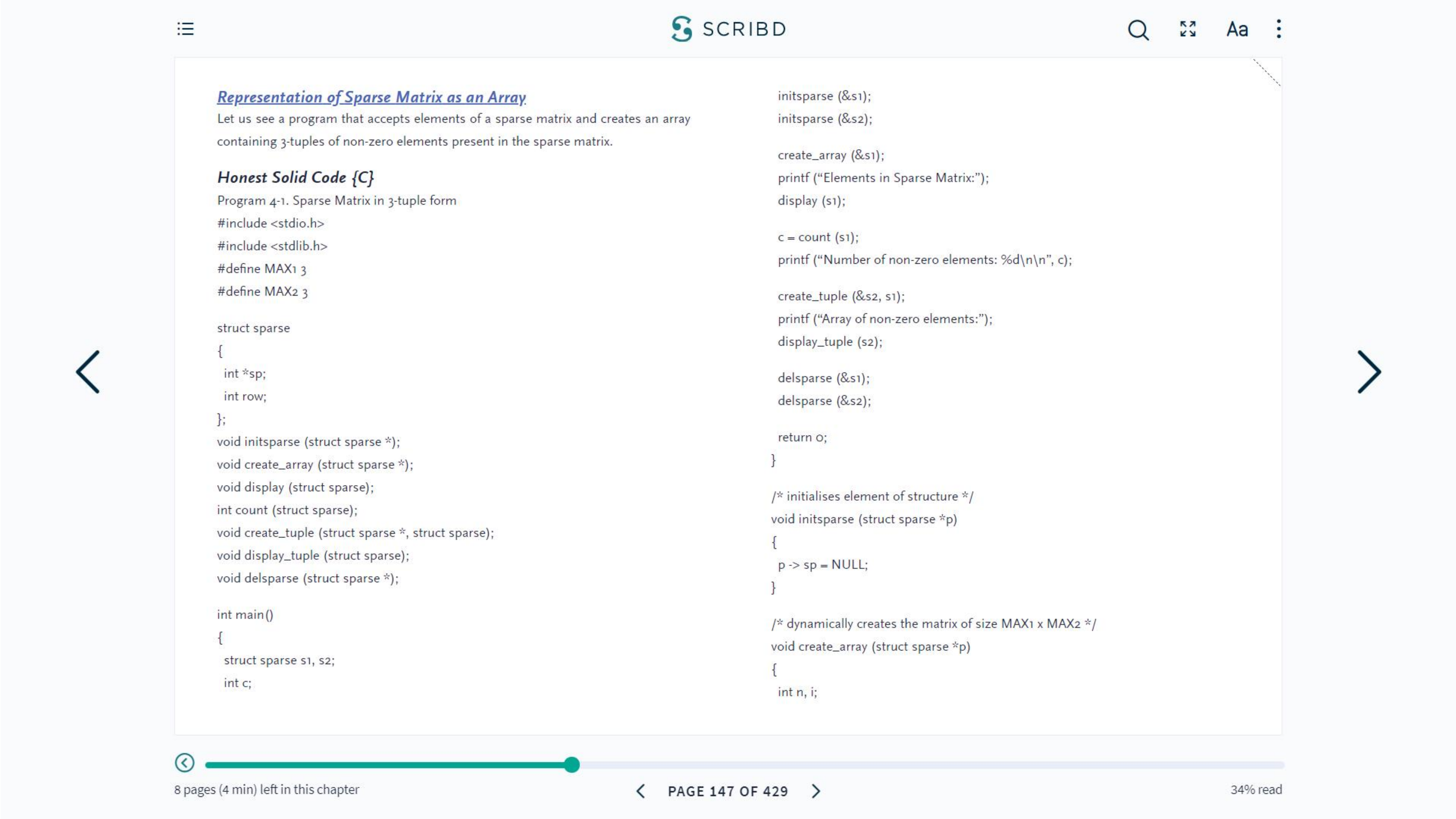
A common way of representing non-zero elements of a sparse matrix is the 3-tuple forms. In this form each non-zero element is stored in a row, with the 1st and 2nd element of this row containing the row and column in which the element is present in the original matrix. The 3rd element in this row stores the actual value of the non-zero element. For example the 3-tuple representation of the sparse matrix shown in Figure 4-1 is given below.

```
int spmat[10][3] = {
                        7, 7, 9,
                        0, 3, -5,
                        1, 1, 4,
                        1, 6, 7,
                        2, 4, 9,
                        3, 1, 3,
                        3, 3, 2,
                        4, 0, 11,
                        4, 2, 2,
                        6, 2, 8
                   }
```

There are two ways in which information of a 3-tuple can be stored:

- Arrays
- Linked List

In both representations information about the non-zero elements is stored. However, as the number of non-zero elements in a sparse matrix may vary, it would be efficient to use a linked list to represent it.



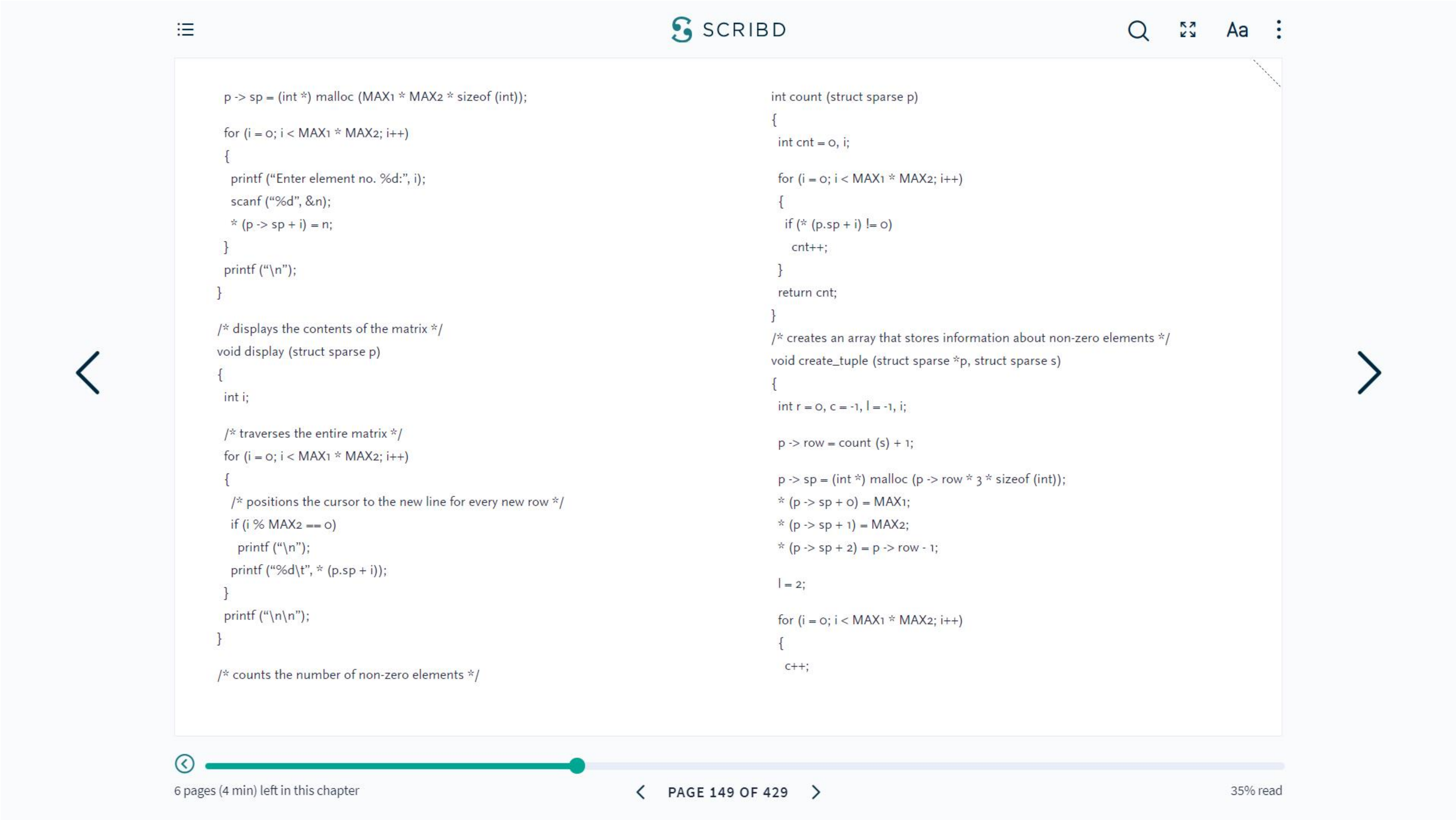
Representation of Sparse Matrix as an Array

Let us see a program that accepts elements of a sparse matrix and creates an array containing 3-tuples of non-zero elements present in the sparse matrix.

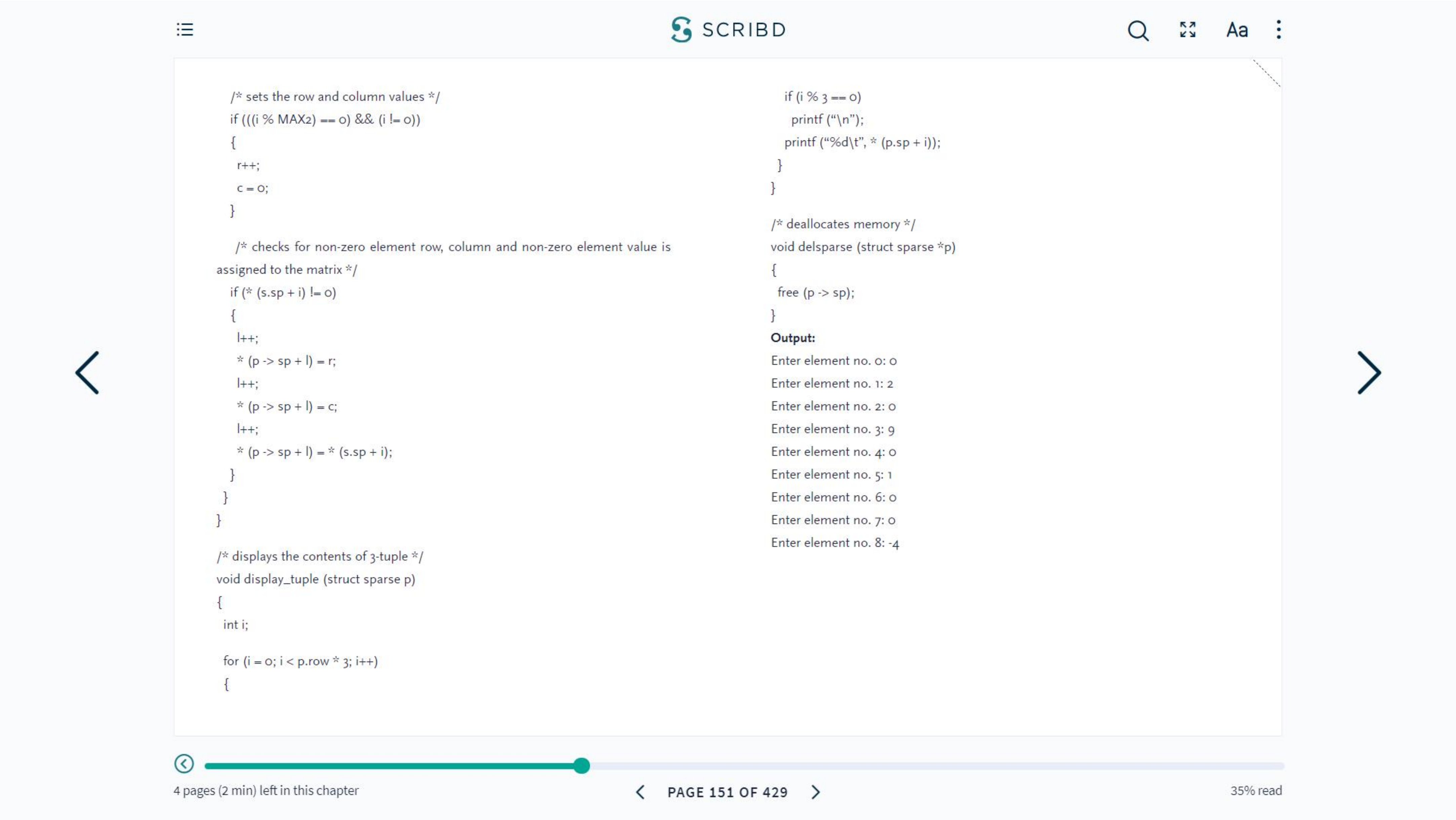
Honest Solid Code {C}

Program 4-1. Sparse Matrix in 3-tuple form

```
#include <stdio.h>
#include <stdlib.h>
#define MAX1 3
#define MAX2 3

struct sparse
{
 int *sp;
 int row;
};
void initsparse (struct sparse *);
void create_array (struct sparse *);
void display (struct sparse);
int count (struct sparse);
void create_tuple (struct sparse *, struct sparse);
void display_tuple (struct sparse);
void delsparse (struct sparse *);

int main()
{
 struct sparse s1, s2;
 int c;
 initsparse (&s1);
 initsparse (&s2);

 create_array (&s1);
 printf (“Elements in Sparse Matrix:”);
 display (s1);

 c = count (s1);
 printf (“Number of non-zero elements: %d\n\n”, c);

 create_tuple (&s2, s1);
 printf (“Array of non-zero elements:”);
 display_tuple (s2);

 delsparse (&s1);
 delsparse (&s2);

 return 0;
}

/* initialises element of structure */
void initsparse (struct sparse *p)
{
 p -> sp = NULL;
}

/* dynamically creates the matrix of size MAX1 x MAX2 */
void create_array (struct sparse *p)
{
 int n, i;
```



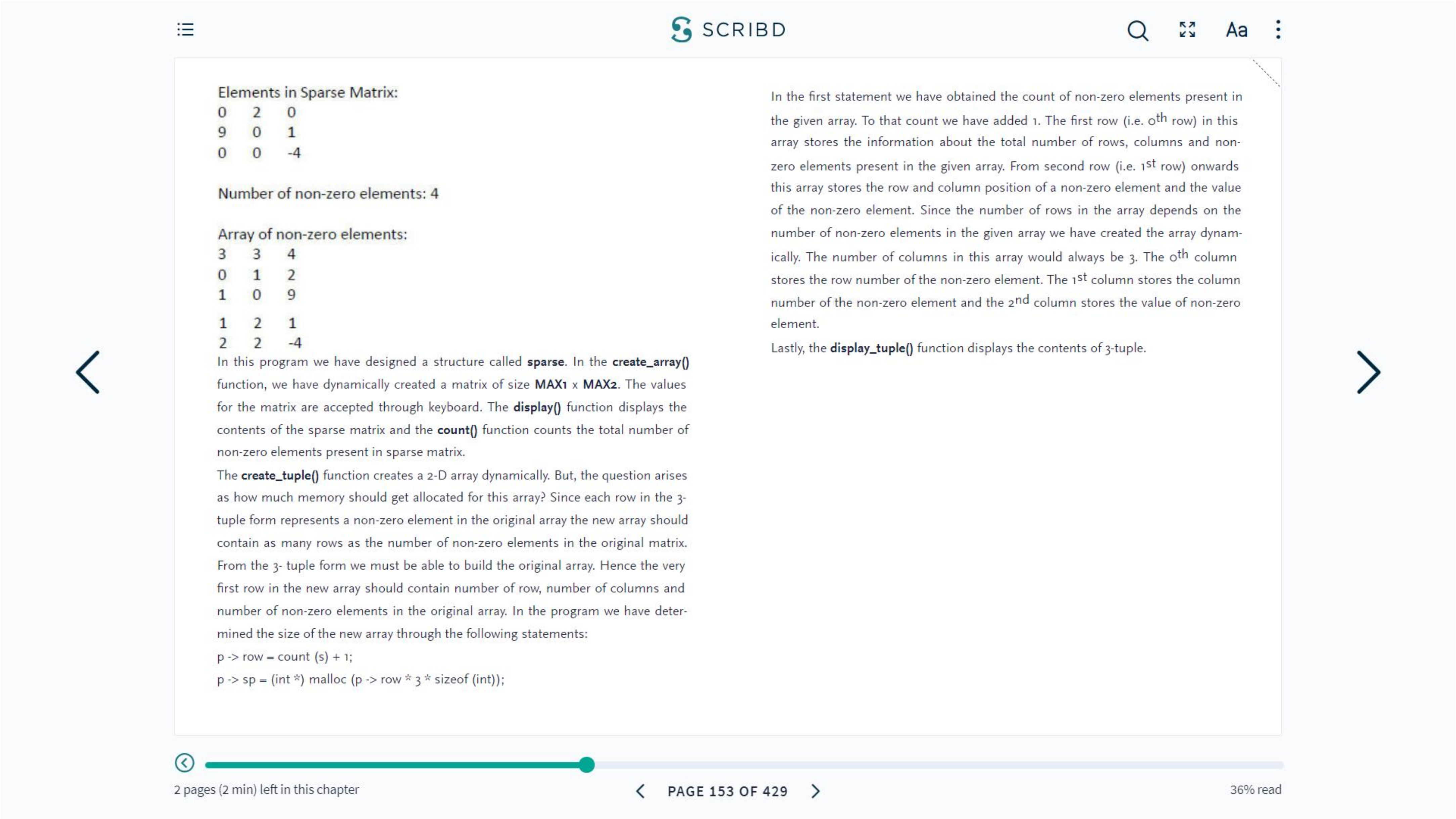
```
 p -> sp = (int *) malloc (MAX1 * MAX2 * sizeof (int));

 for (i = 0; i < MAX1 * MAX2; i++)
 {
  printf (“Enter element no. %d:”, i);
  scanf (“%d”, &n);
  * (p -> sp + i) = n;
 }
 printf (“\n”);
}

/* displays the contents of the matrix */
void display (struct sparse p)
{
 int i;

 /* traverses the entire matrix */
 for (i = 0; i < MAX1 * MAX2; i++)
 {
  /* positions the cursor to the new line for every new row */
  if (i % MAX2 == 0)
   printf (“\n”);
  printf (“%d\t”, * (p.sp + i));
 }
 printf (“\n\n”);
}

/* counts the number of non-zero elements */
int count (struct sparse p)
{
 int cnt = 0, i;

 for (i = 0; i < MAX1 * MAX2; i++)
 {
  if (* (p.sp + i) != 0)
   cnt++;
 }
 return cnt;
}
/* creates an array that stores information about non-zero elements */
void create_tuple (struct sparse *p, struct sparse s)
{
 int r = 0, c = -1, l = -1, i;

 p -> row = count (s) + 1;

 p -> sp = (int *) malloc (p -> row * 3 * sizeof (int));
 * (p -> sp + 0) = MAX1;
 * (p -> sp + 1) = MAX2;
 * (p -> sp + 2) = p -> row - 1;

 l = 2;

 for (i = 0; i < MAX1 * MAX2; i++)
 {
  c++;
```



```
  /* sets the row and column values */
  if (((i % MAX2) == 0) && (i != 0))
  {
   r++;
   c = 0;
  }

   /* checks for non-zero element row, column and non-zero element value is
assigned to the matrix */
  if (* (s.sp + i) != 0)
  {
   l++;
   * (p -> sp + l) = r;
   l++;
   * (p -> sp + l) = c;
   l++;
   * (p -> sp + l) = * (s.sp + i);
  }
 }
}

/* displays the contents of 3-tuple */
void display_tuple (struct sparse p)
{
 int i;

 for (i = 0; i < p.row * 3; i++)
 {
  if (i % 3 == 0)
   printf ("\n");
  printf ("%d\t", * (p.sp + i));
 }
}

/* deallocates memory */
void delsparse (struct sparse *p)
{
 free (p -> sp);
}
```

Output:

```
Enter element no. 0: 0
Enter element no. 1: 2
Enter element no. 2: 0
Enter element no. 3: 9
Enter element no. 4: 0
Enter element no. 5: 1
Enter element no. 6: 0
Enter element no. 7: 0
Enter element no. 8: -4
```



```
Elements in Sparse Matrix:
0    2    0
9    0    1
0    0    -4

Number of non-zero elements: 4

Array of non-zero elements:
3    3    4
0    1    2
1    0    9

1    2    1
2    2    -4
```

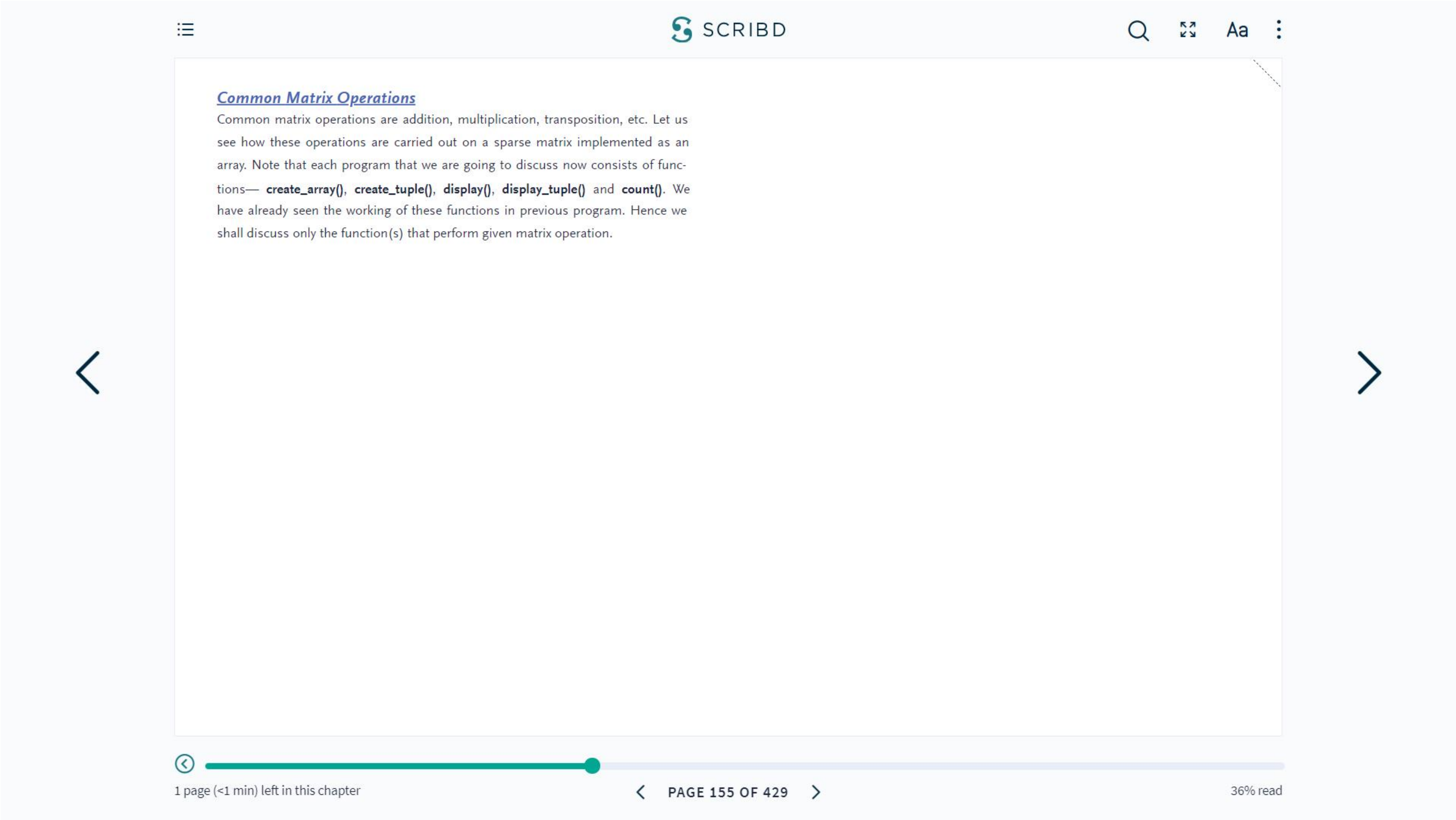
In this program we have designed a structure called **sparse**. In the **create_array()** function, we have dynamically created a matrix of size **MAX1** x **MAX2**. The values for the matrix are accepted through keyboard. The **display()** function displays the contents of the sparse matrix and the **count()** function counts the total number of non-zero elements present in sparse matrix.

The **create_tuple()** function creates a 2-D array dynamically. But, the question arises as how much memory should get allocated for this array? Since each row in the 3-tuple form represents a non-zero element in the original array the new array should contain as many rows as the number of non-zero elements in the original matrix. From the 3- tuple form we must be able to build the original array. Hence the very first row in the new array should contain number of row, number of columns and number of non-zero elements in the original array. In the program we have determined the size of the new array through the following statements:

```
p -> row = count (s) + 1;
p -> sp = (int *) malloc (p -> row * 3 * sizeof (int));
```

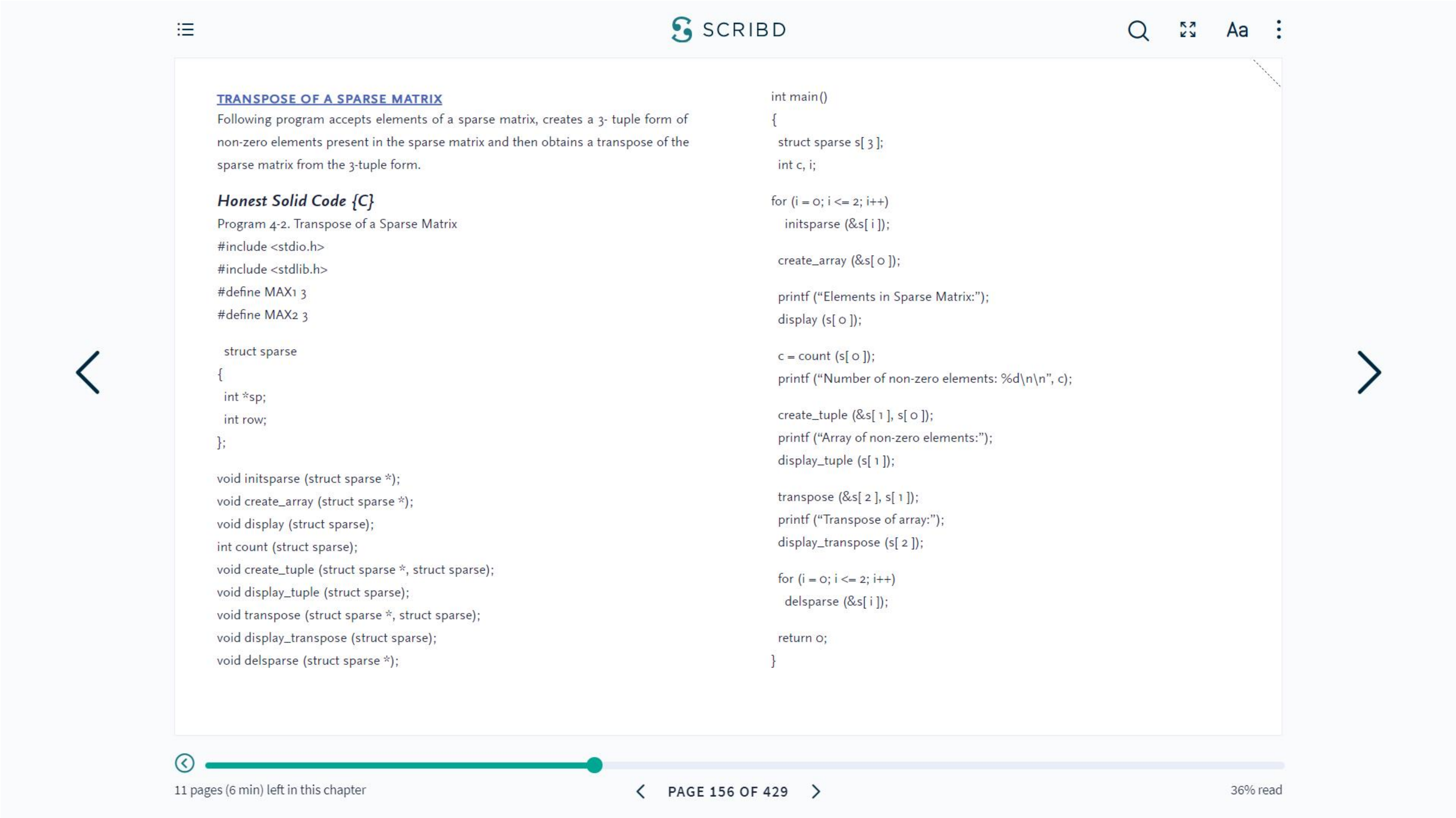
In the first statement we have obtained the count of non-zero elements present in the given array. To that count we have added 1. The first row (i.e. 0th row) in this array stores the information about the total number of rows, columns and non-zero elements present in the given array. From second row (i.e. 1st row) onwards this array stores the row and column position of a non-zero element and the value of the non-zero element. Since the number of rows in the array depends on the number of non-zero elements in the given array we have created the array dynamically. The number of columns in this array would always be 3. The 0th column stores the row number of the non-zero element. The 1st column stores the column number of the non-zero element and the 2nd column stores the value of non-zero element.

Lastly, the **display_tuple()** function displays the contents of 3-tuple.



Common Matrix Operations

Common matrix operations are addition, multiplication, transposition, etc. Let us see how these operations are carried out on a sparse matrix implemented as an array. Note that each program that we are going to discuss now consists of functions— **create_array()**, **create_tuple()**, **display()**, **display_tuple()** and **count()**. We have already seen the working of these functions in previous program. Hence we shall discuss only the function(s) that perform given matrix operation.



TRANSPOSE OF A SPARSE MATRIX

Following program accepts elements of a sparse matrix, creates a 3- tuple form of non-zero elements present in the sparse matrix and then obtains a transpose of the sparse matrix from the 3-tuple form.

Honest Solid Code {C}

Program 4-2. Transpose of a Sparse Matrix

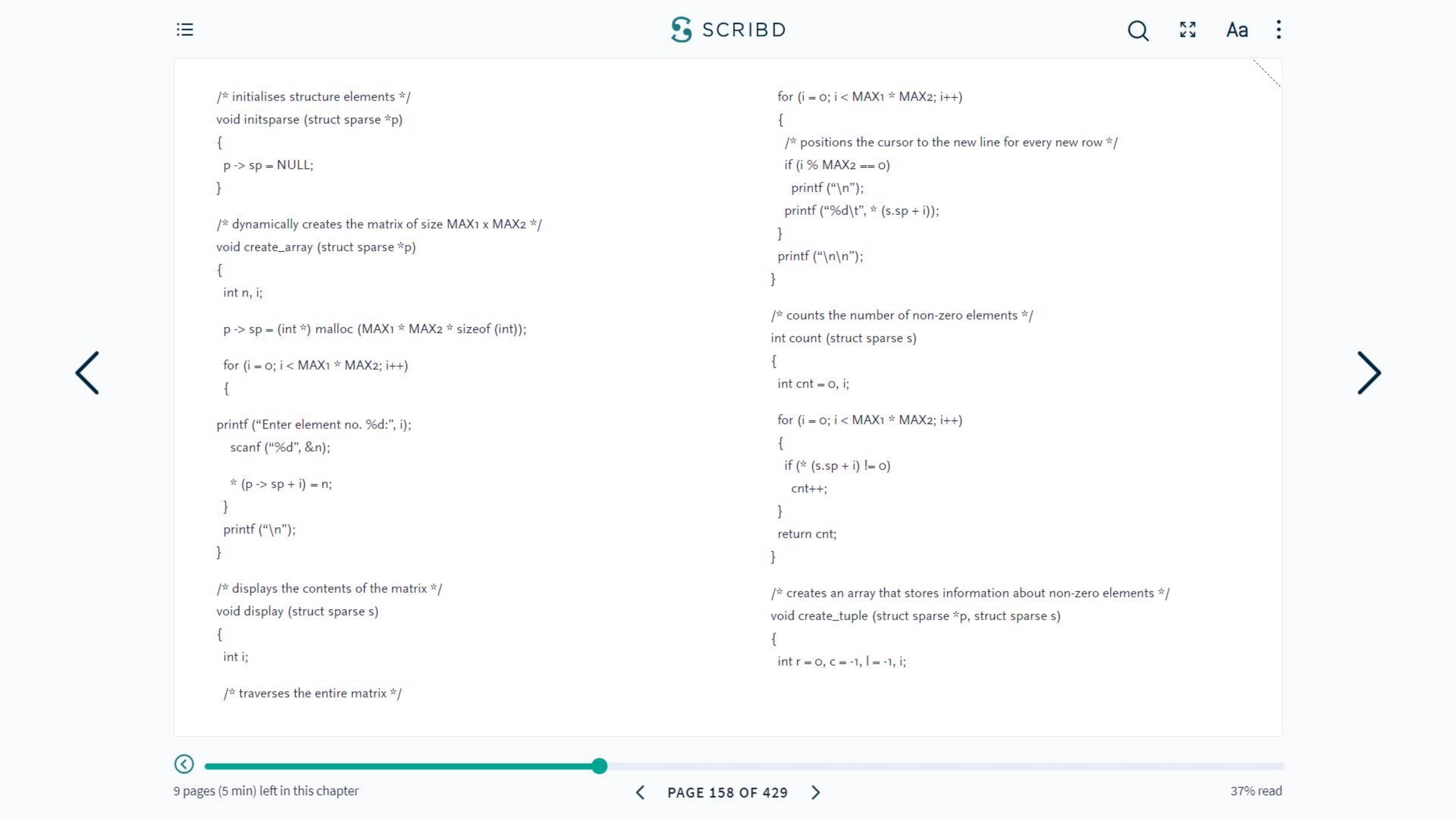
```
#include <stdio.h>
#include <stdlib.h>
#define MAX1 3
#define MAX2 3

 struct sparse
{
 int *sp;
 int row;
};

void initsparse (struct sparse *);
void create_array (struct sparse *);
void display (struct sparse);
int count (struct sparse);
void create_tuple (struct sparse *, struct sparse);
void display_tuple (struct sparse);
void transpose (struct sparse *, struct sparse);
void display_transpose (struct sparse);
void delsparse (struct sparse *);
```

```
int main()
{
 struct sparse s[ 3 ];
 int c, i;

for (i = 0; i <= 2; i++)
  initsparse (&s[ i ]);

 create_array (&s[ 0 ]);

 printf ("Elements in Sparse Matrix:");
 display (s[ 0 ]);

 c = count (s[ 0 ]);
 printf ("Number of non-zero elements: %d\n\n", c);

 create_tuple (&s[ 1 ], s[ 0 ]);
 printf ("Array of non-zero elements:");
 display_tuple (s[ 1 ]);

 transpose (&s[ 2 ], s[ 1 ]);
 printf ("Transpose of array:");
 display_transpose (s[ 2 ]);

 for (i = 0; i <= 2; i++)
  delsparse (&s[ i ]);

 return 0;
}
```

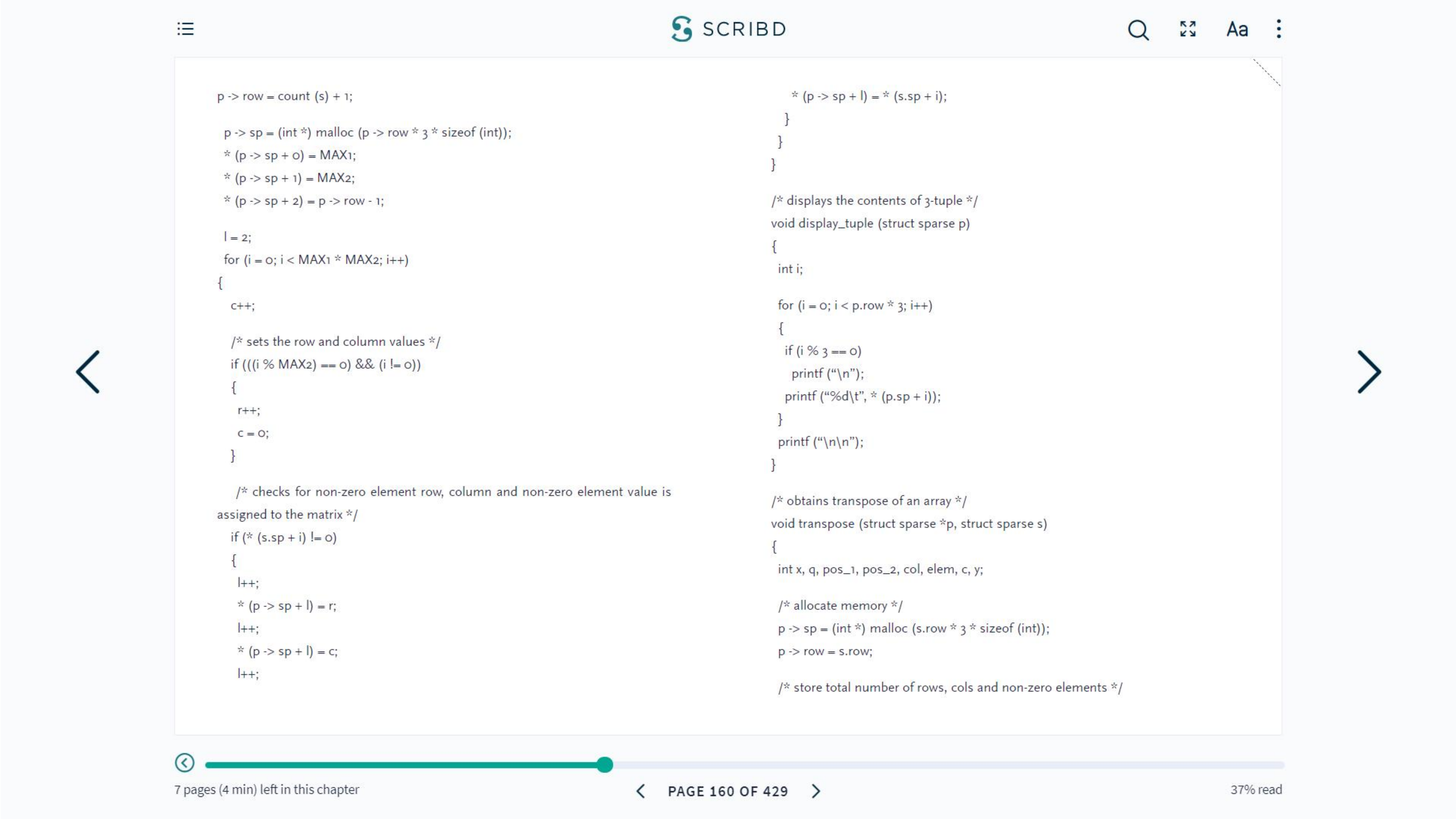
11 pages (6 min) left in this chapter

PAGE 156 OF 429

36% read

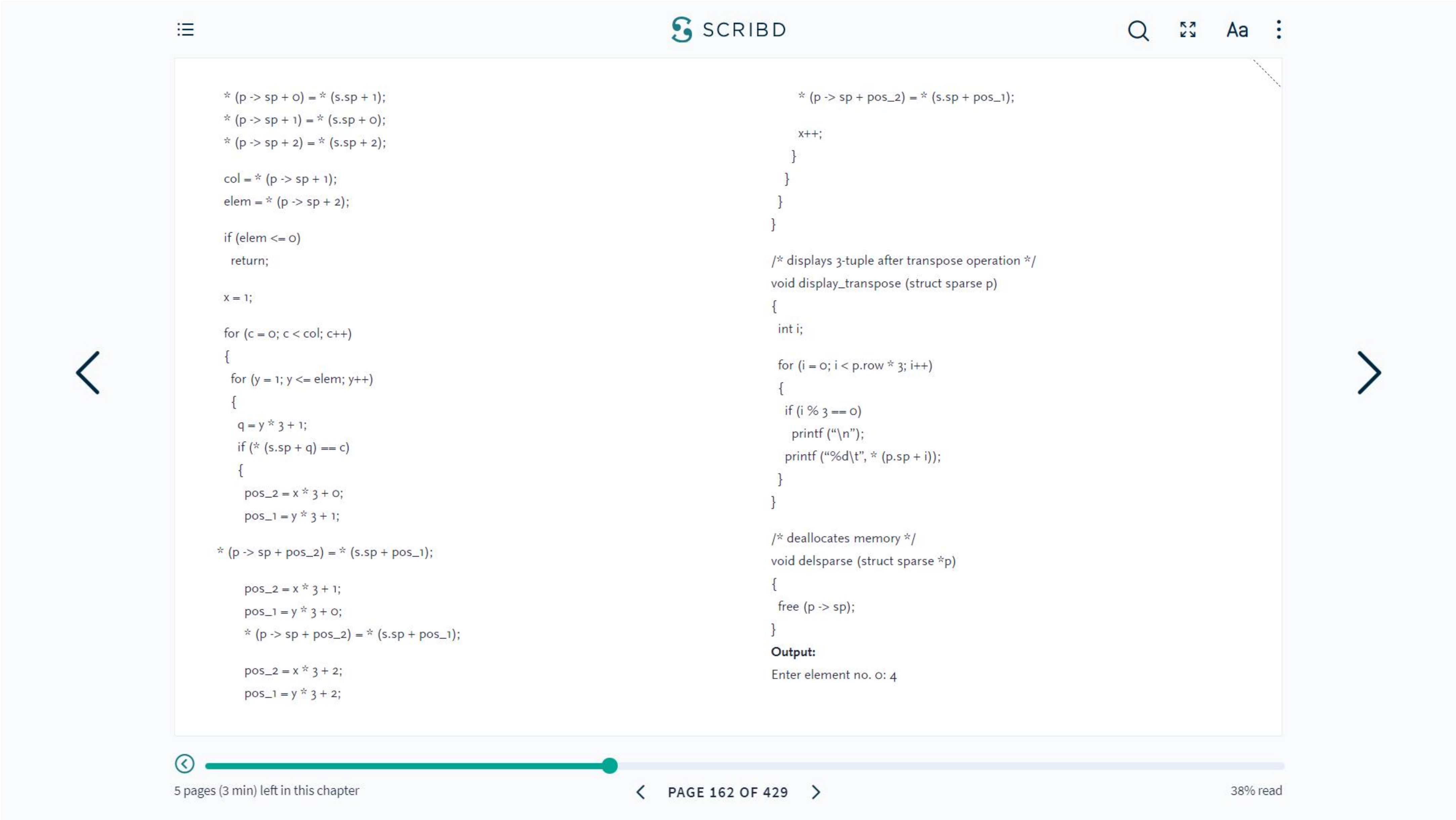


```
/* initialises structure elements */
void initsparse (struct sparse *p)
{
 p -> sp = NULL;
}

/* dynamically creates the matrix of size MAX1 x MAX2 */
void create_array (struct sparse *p)
{
 int n, i;

 p -> sp = (int *) malloc (MAX1 * MAX2 * sizeof (int));

 for (i = 0; i < MAX1 * MAX2; i++)
 {

printf (“Enter element no. %d:”, i);
  scanf (“%d”, &n);

  * (p -> sp + i) = n;
 }
 printf (“\n”);
}

/* displays the contents of the matrix */
void display (struct sparse s)
{
 int i;

 /* traverses the entire matrix */
 for (i = 0; i < MAX1 * MAX2; i++)
 {
  /* positions the cursor to the new line for every new row */
  if (i % MAX2 == 0)
   printf (“\n”);
  printf (“%d\t”, * (s.sp + i));
 }
 printf (“\n\n”);
}

/* counts the number of non-zero elements */
int count (struct sparse s)
{
 int cnt = 0, i;

 for (i = 0; i < MAX1 * MAX2; i++)
 {
  if (* (s.sp + i) != 0)
   cnt++;
 }
 return cnt;
}

/* creates an array that stores information about non-zero elements */
void create_tuple (struct sparse *p, struct sparse s)
{
 int r = 0, c = -1, l = -1, i;
```



```
p -> row = count (s) + 1;

 p -> sp = (int *) malloc (p -> row * 3 * sizeof (int));
 * (p -> sp + 0) = MAX1;
 * (p -> sp + 1) = MAX2;
 * (p -> sp + 2) = p -> row - 1;

 l = 2;
 for (i = 0; i < MAX1 * MAX2; i++)
{
  c++;

  /* sets the row and column values */
  if (((i % MAX2) == 0) && (i != 0))
  {
   r++;
   c = 0;
  }

   /* checks for non-zero element row, column and non-zero element value is
assigned to the matrix */
  if (* (s.sp + i) != 0)
  {
   l++;
   * (p -> sp + l) = r;
   l++;
   * (p -> sp + l) = c;
   l++;
```

```
    * (p -> sp + l) = * (s.sp + i);
   }
  }
}

/* displays the contents of 3-tuple */
void display_tuple (struct sparse p)
{
 int i;

 for (i = 0; i < p.row * 3; i++)
 {
  if (i % 3 == 0)
   printf (“\n”);
  printf (“%d\t”, * (p.sp + i));
 }
 printf (“\n\n”);
}

/* obtains transpose of an array */
void transpose (struct sparse *p, struct sparse s)
{
 int x, q, pos_1, pos_2, col, elem, c, y;

 /* allocate memory */
 p -> sp = (int *) malloc (s.row * 3 * sizeof (int));
 p -> row = s.row;

 /* store total number of rows, cols and non-zero elements */
```

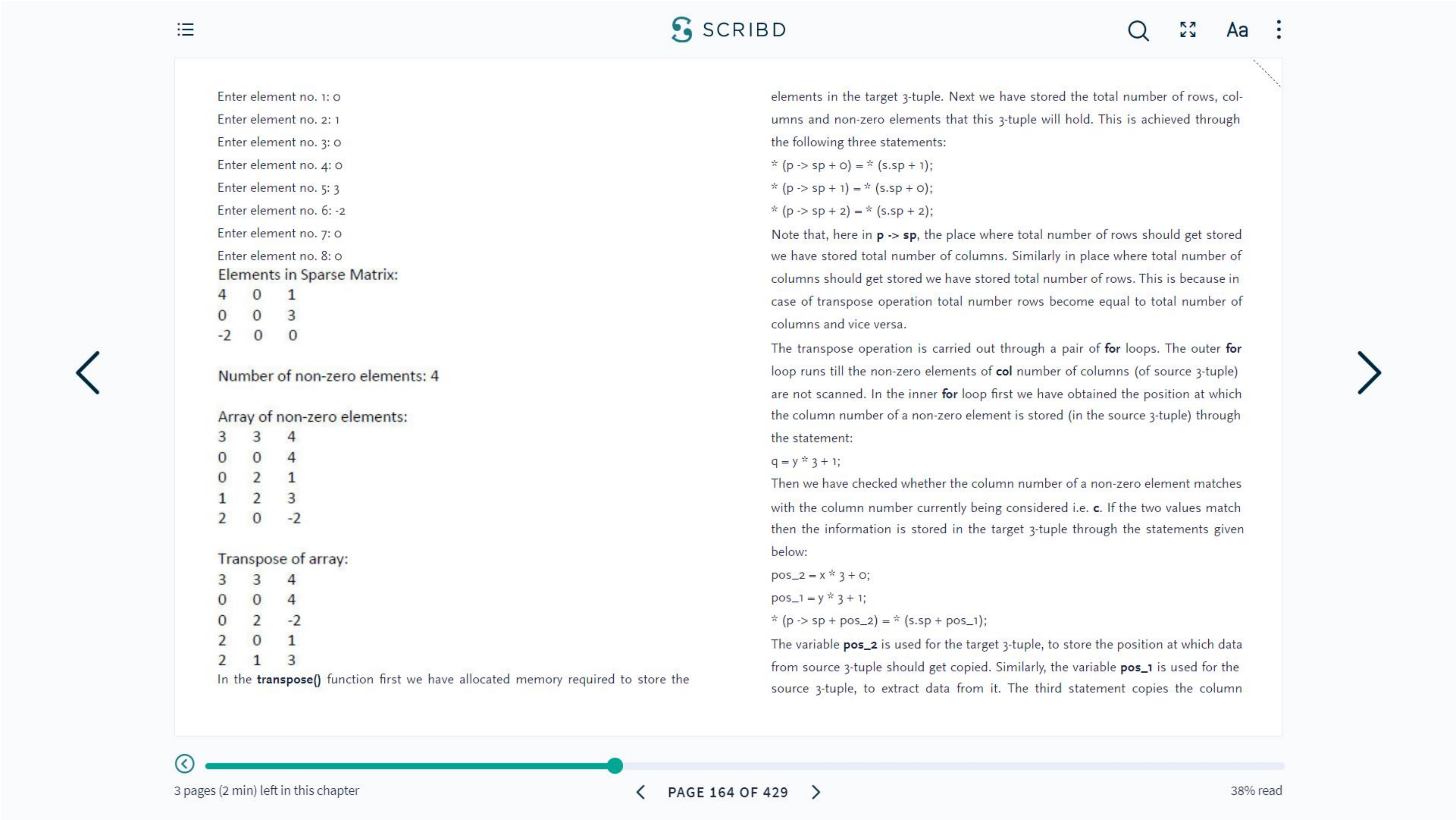


```
* (p -> sp + 0) = * (s.sp + 1);
* (p -> sp + 1) = * (s.sp + 0);
* (p -> sp + 2) = * (s.sp + 2);

col = * (p -> sp + 1);
elem = * (p -> sp + 2);

if (elem <= 0)
 return;

x = 1;

for (c = 0; c < col; c++)
{
 for (y = 1; y <= elem; y++)
 {
  q = y * 3 + 1;
  if (* (s.sp + q) == c)
  {
   pos_2 = x * 3 + 0;
   pos_1 = y * 3 + 1;

* (p -> sp + pos_2) = * (s.sp + pos_1);

   pos_2 = x * 3 + 1;
   pos_1 = y * 3 + 0;
   * (p -> sp + pos_2) = * (s.sp + pos_1);

   pos_2 = x * 3 + 2;
   pos_1 = y * 3 + 2;
```

```
   * (p -> sp + pos_2) = * (s.sp + pos_1);

   x++;
  }
 }
}
}

/* displays 3-tuple after transpose operation */
void display_transpose (struct sparse p)
{
 int i;

 for (i = 0; i < p.row * 3; i++)
 {
  if (i % 3 == 0)
   printf ("\n");
  printf ("%d\t", * (p.sp + i));
 }
}

/* deallocates memory */
void delsparse (struct sparse *p)
{
 free (p -> sp);
}
```

Output:

```
Enter element no. 0: 4
```



```
Enter element no. 1: 0
Enter element no. 2: 1
Enter element no. 3: 0
Enter element no. 4: 0
Enter element no. 5: 3
Enter element no. 6: -2
Enter element no. 7: 0
Enter element no. 8: 0
Elements in Sparse Matrix:
4    0    1
0    0    3
-2   0    0

Number of non-zero elements: 4

Array of non-zero elements:
3    3    4
0    0    4
0    2    1
1    2    3
2    0    -2

Transpose of array:
3    3    4
0    0    4
0    2    -2
2    0    1
2    1    3
```

In the **transpose()** function first we have allocated memory required to store the elements in the target 3-tuple. Next we have stored the total number of rows, columns and non-zero elements that this 3-tuple will hold. This is achieved through the following three statements:

```
* (p -> sp + 0) = * (s.sp + 1);
* (p -> sp + 1) = * (s.sp + 0);
* (p -> sp + 2) = * (s.sp + 2);
```

Note that, here in **p -> sp**, the place where total number of rows should get stored we have stored total number of columns. Similarly in place where total number of columns should get stored we have stored total number of rows. This is because in case of transpose operation total number rows become equal to total number of columns and vice versa.

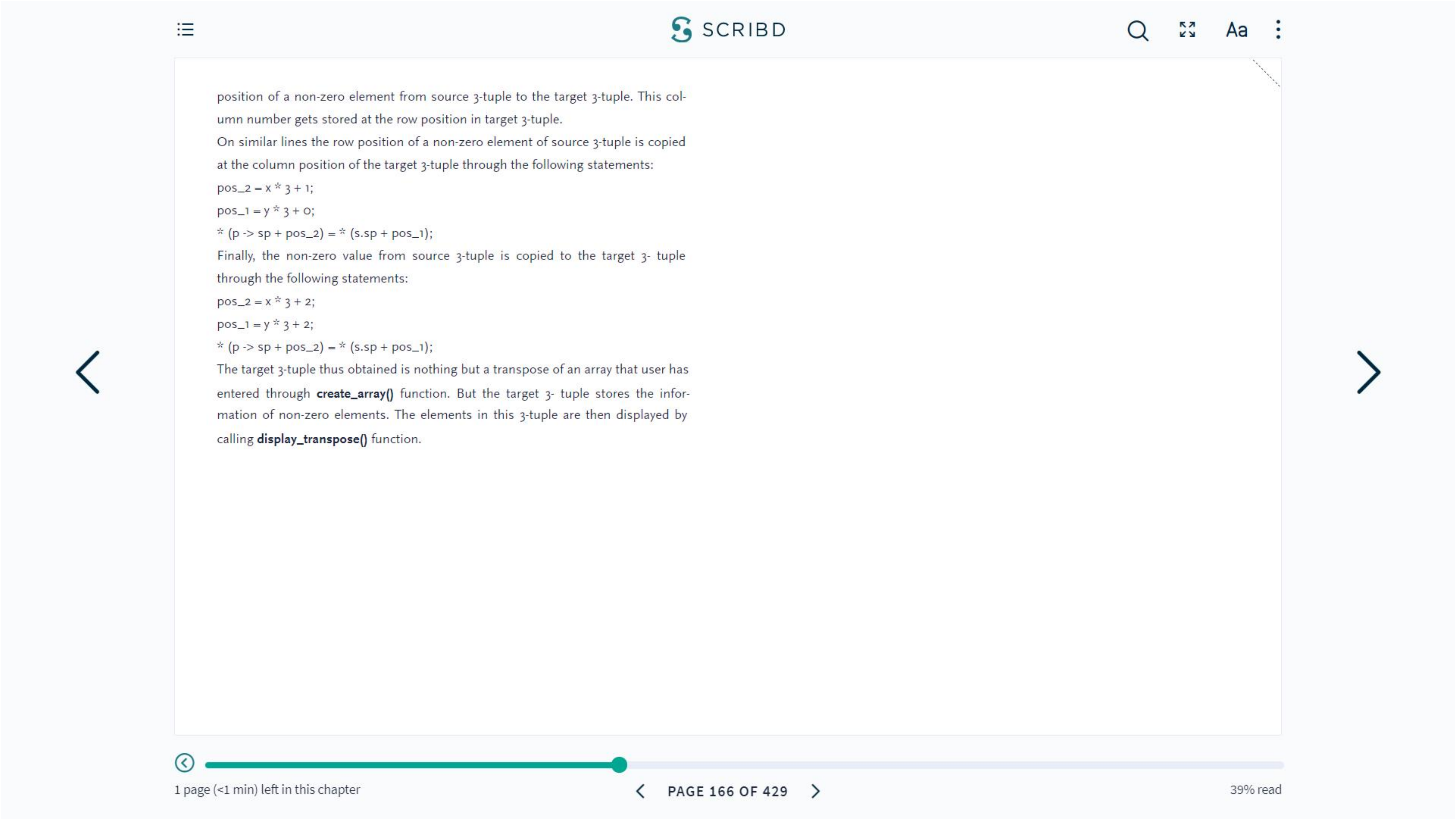
The transpose operation is carried out through a pair of **for** loops. The outer **for** loop runs till the non-zero elements of **col** number of columns (of source 3-tuple) are not scanned. In the inner **for** loop first we have obtained the position at which the column number of a non-zero element is stored (in the source 3-tuple) through the statement:

```
q = y * 3 + 1;
```

Then we have checked whether the column number of a non-zero element matches with the column number currently being considered i.e. **c**. If the two values match then the information is stored in the target 3-tuple through the statements given below:

```
pos_2 = x * 3 + 0;
pos_1 = y * 3 + 1;
* (p -> sp + pos_2) = * (s.sp + pos_1);
```

The variable **pos_2** is used for the target 3-tuple, to store the position at which data from source 3-tuple should get copied. Similarly, the variable **pos_1** is used for the source 3-tuple, to extract data from it. The third statement copies the column



position of a non-zero element from source 3-tuple to the target 3-tuple. This column number gets stored at the row position in target 3-tuple.

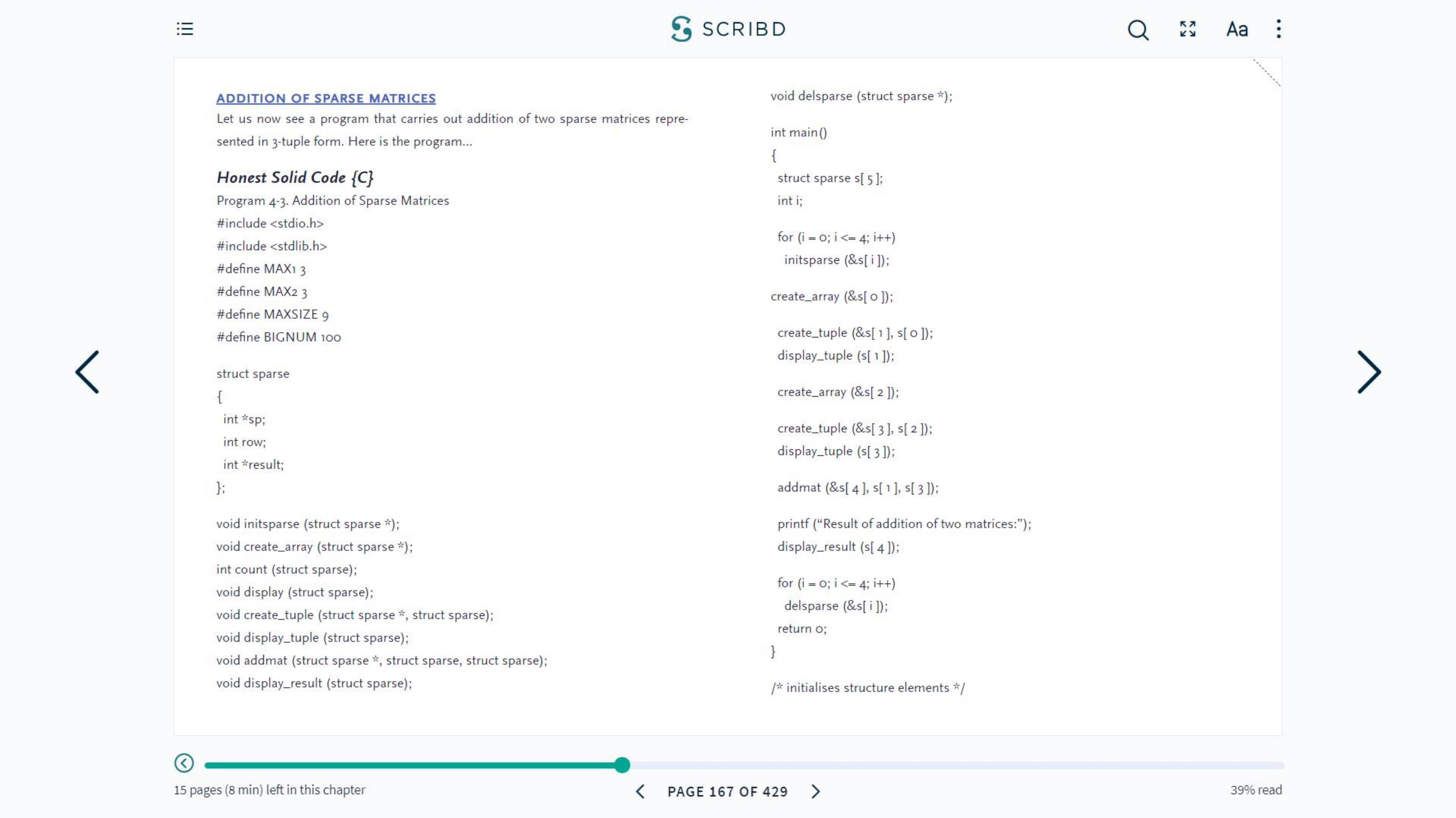
On similar lines the row position of a non-zero element of source 3-tuple is copied at the column position of the target 3-tuple through the following statements:

```
pos_2 = x * 3 + 1;
pos_1 = y * 3 + 0;
* (p -> sp + pos_2) = * (s.sp + pos_1);
```

Finally, the non-zero value from source 3-tuple is copied to the target 3- tuple through the following statements:

```
pos_2 = x * 3 + 2;
pos_1 = y * 3 + 2;
* (p -> sp + pos_2) = * (s.sp + pos_1);
```

The target 3-tuple thus obtained is nothing but a transpose of an array that user has entered through **create_array()** function. But the target 3- tuple stores the information of non-zero elements. The elements in this 3-tuple are then displayed by calling **display_transpose()** function.



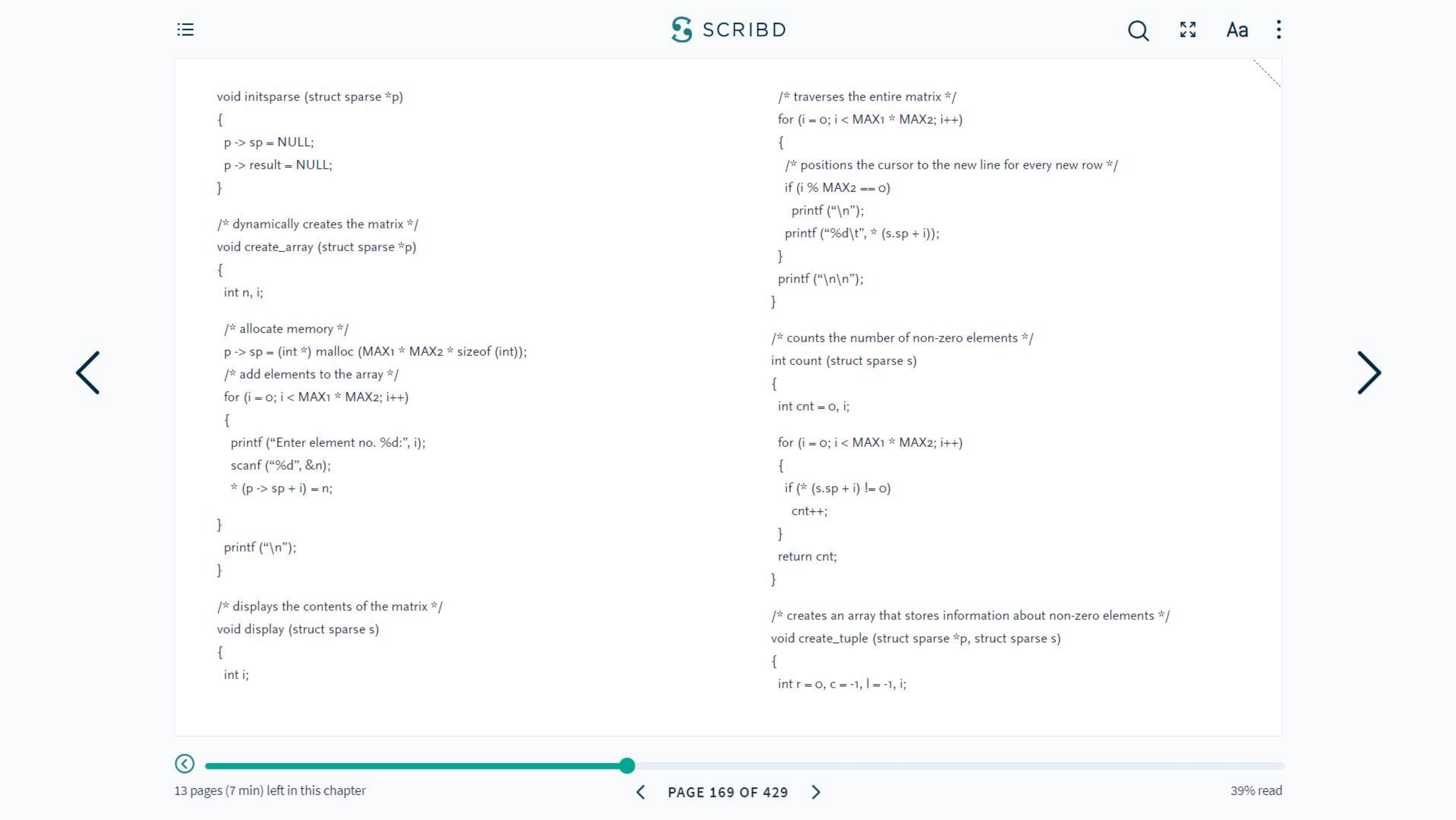
ADDITION OF SPARSE MATRICES

Let us now see a program that carries out addition of two sparse matrices represented in 3-tuple form. Here is the program...

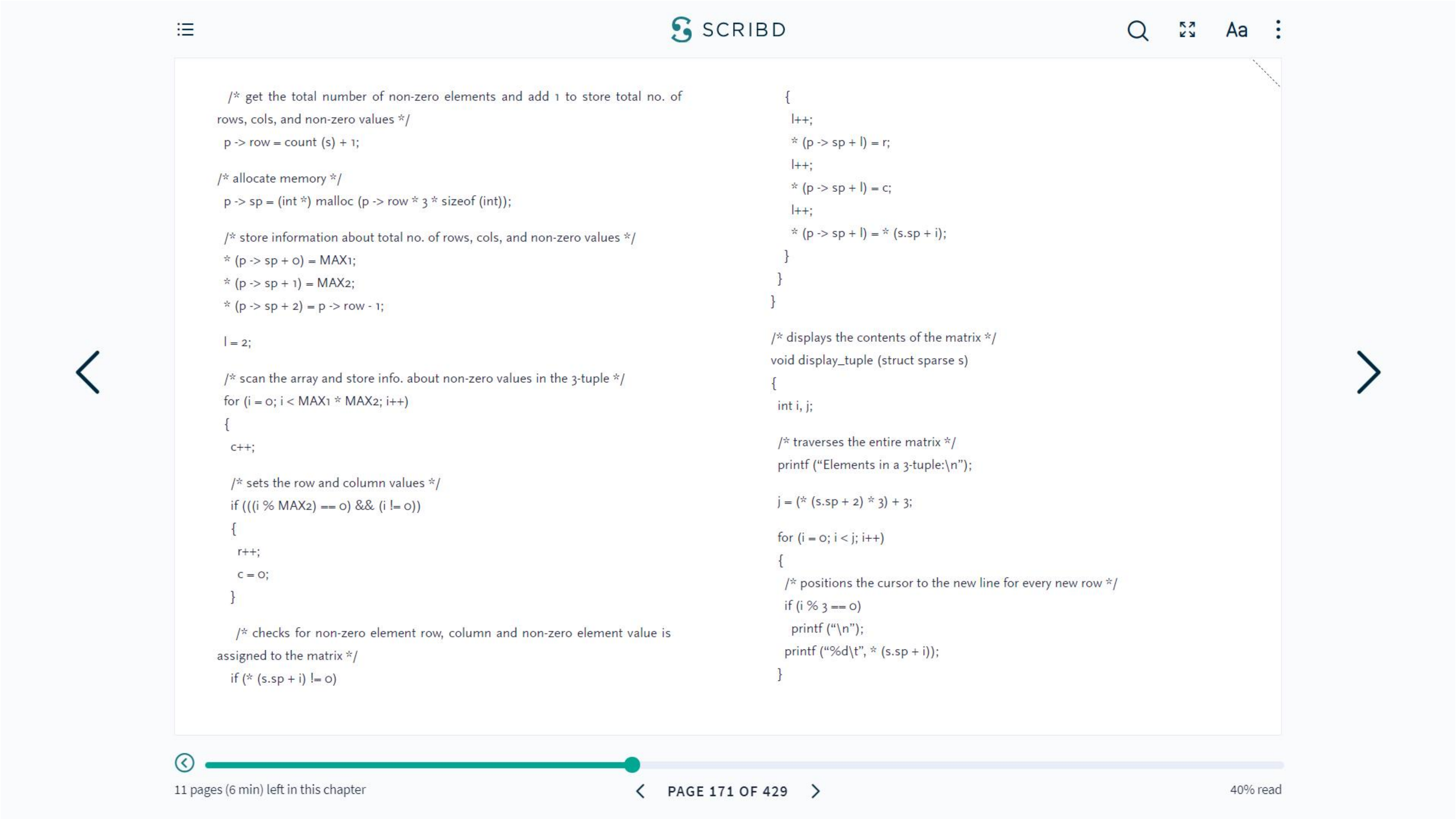
Honest Solid Code {C}

Program 4-3. Addition of Sparse Matrices

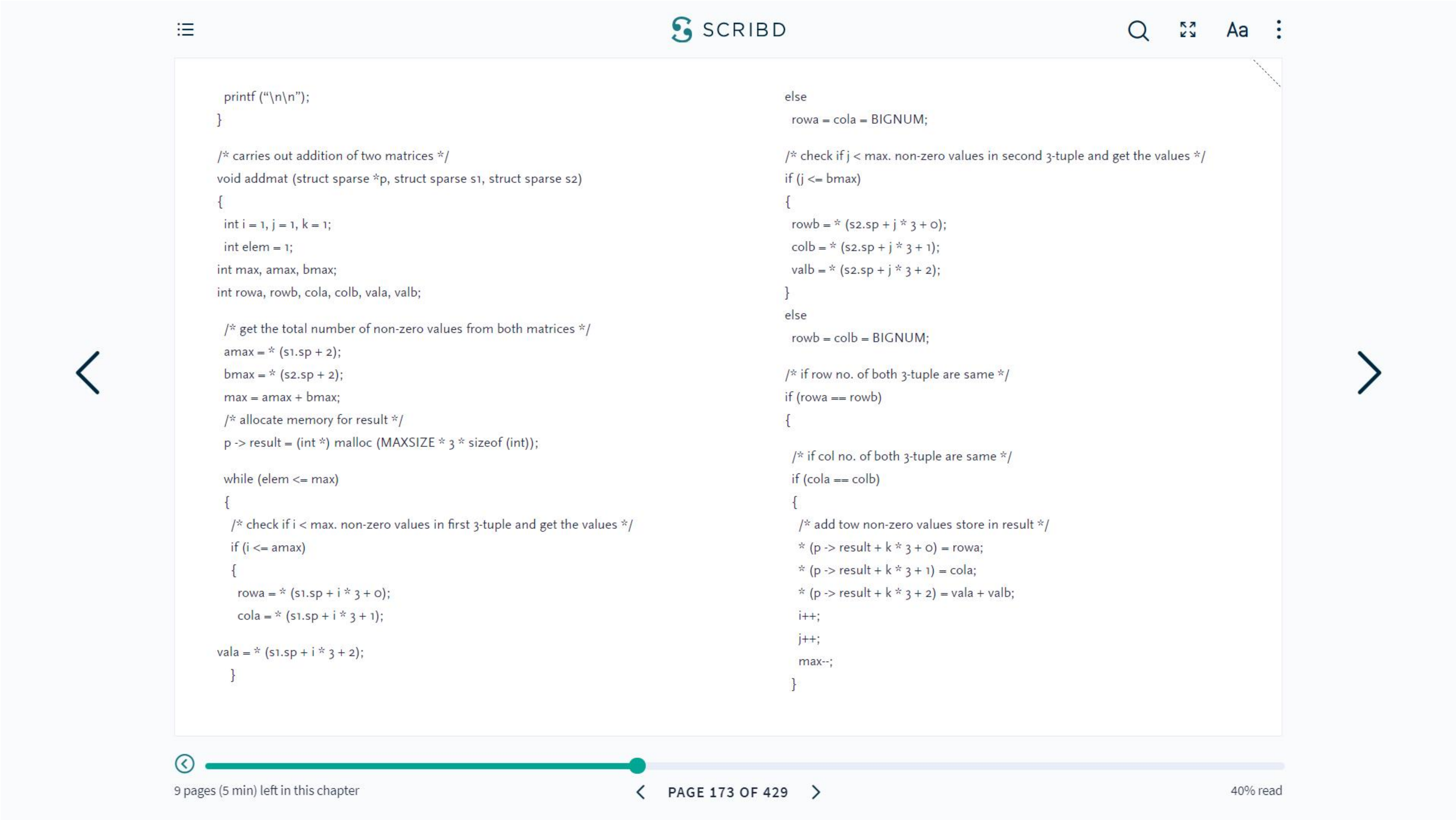
```
#include <stdio.h>
#include <stdlib.h>
#define MAX1 3
#define MAX2 3
#define MAXSIZE 9
#define BIGNUM 100

struct sparse
{
 int *sp;
 int row;
 int *result;
};

void initsparse (struct sparse *);
void create_array (struct sparse *);
int count (struct sparse);
void display (struct sparse);
void create_tuple (struct sparse *, struct sparse);
void display_tuple (struct sparse);
void addmat (struct sparse *, struct sparse, struct sparse);
void display_result (struct sparse);
void delsparse (struct sparse *);

int main()
{
 struct sparse s[ 5 ];
 int i;

 for (i = 0; i <= 4; i++)
  initsparse (&s[ i ]);

create_array (&s[ 0 ]);

 create_tuple (&s[ 1 ], s[ 0 ]);
 display_tuple (s[ 1 ]);

 create_array (&s[ 2 ]);

 create_tuple (&s[ 3 ], s[ 2 ]);
 display_tuple (s[ 3 ]);

 addmat (&s[ 4 ], s[ 1 ], s[ 3 ]);

 printf (“Result of addition of two matrices:”);
 display_result (s[ 4 ]);

 for (i = 0; i <= 4; i++)
  delsparse (&s[ i ]);
 return 0;
}

/* initialises structure elements */
```



```
void initsparse (struct sparse *p)
{
 p -> sp = NULL;
 p -> result = NULL;
}

/* dynamically creates the matrix */
void create_array (struct sparse *p)
{
 int n, i;

 /* allocate memory */
 p -> sp = (int *) malloc (MAX1 * MAX2 * sizeof (int));
 /* add elements to the array */
 for (i = 0; i < MAX1 * MAX2; i++)
 {
  printf (“Enter element no. %d:”, i);
  scanf (“%d”, &n);
  * (p -> sp + i) = n;

}
 printf (“\n”);
}

/* displays the contents of the matrix */
void display (struct sparse s)
{
 int i;

 /* traverses the entire matrix */
 for (i = 0; i < MAX1 * MAX2; i++)
 {
  /* positions the cursor to the new line for every new row */
  if (i % MAX2 == 0)
   printf (“\n”);
  printf (“%d\t”, * (s.sp + i));
 }
 printf (“\n\n”);
}

/* counts the number of non-zero elements */
int count (struct sparse s)
{
 int cnt = 0, i;

 for (i = 0; i < MAX1 * MAX2; i++)
 {
  if (* (s.sp + i) != 0)
   cnt++;
 }
 return cnt;
}

/* creates an array that stores information about non-zero elements */
void create_tuple (struct sparse *p, struct sparse s)
{
 int r = 0, c = -1, l = -1, i;
```

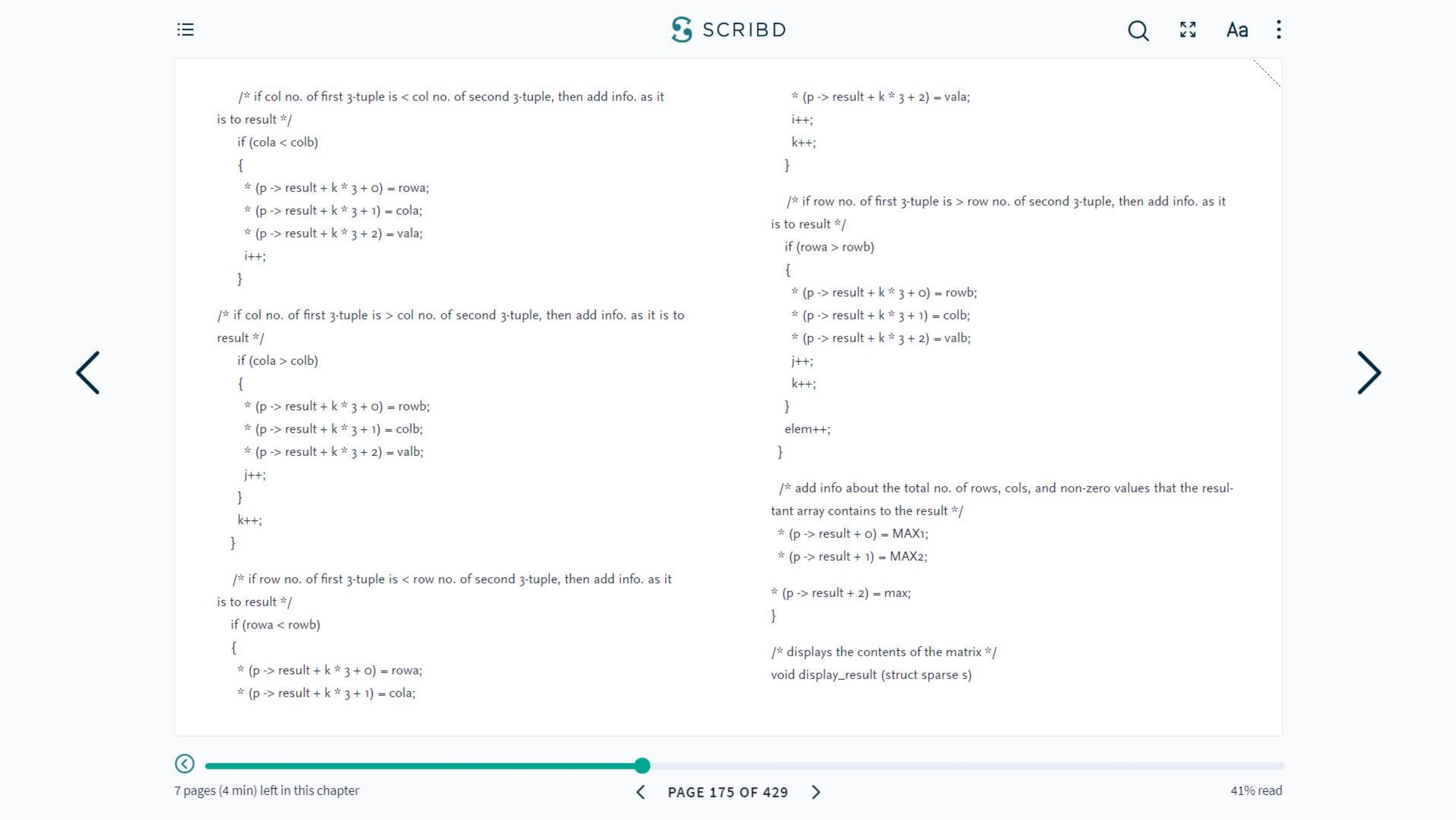


```
  /* get the total number of non-zero elements and add 1 to store total no. of
rows, cols, and non-zero values */
 p -> row = count (s) + 1;

/* allocate memory */
 p -> sp = (int *) malloc (p -> row * 3 * sizeof (int));

 /* store information about total no. of rows, cols, and non-zero values */
 * (p -> sp + 0) = MAX1;
 * (p -> sp + 1) = MAX2;
 * (p -> sp + 2) = p -> row - 1;

 l = 2;

 /* scan the array and store info. about non-zero values in the 3-tuple */
 for (i = 0; i < MAX1 * MAX2; i++)
 {
  c++;

  /* sets the row and column values */
  if (((i % MAX2) == 0) && (i != 0))
  {
   r++;
   c = 0;
  }

   /* checks for non-zero element row, column and non-zero element value is
assigned to the matrix */
  if (* (s.sp + i) != 0)
  {
   l++;
   * (p -> sp + l) = r;
   l++;
   * (p -> sp + l) = c;
   l++;
   * (p -> sp + l) = * (s.sp + i);
  }
 }
}

/* displays the contents of the matrix */
void display_tuple (struct sparse s)
{
 int i, j;

 /* traverses the entire matrix */
 printf (“Elements in a 3-tuple:\n”);

 j = (* (s.sp + 2) * 3) + 3;

 for (i = 0; i < j; i++)
 {
  /* positions the cursor to the new line for every new row */
  if (i % 3 == 0)
   printf (“\n”);
  printf (“%d\t”, * (s.sp + i));
 }
```



```
 printf (“\n\n”);
}

/* carries out addition of two matrices */
void addmat (struct sparse *p, struct sparse s1, struct sparse s2)
{
 int i = 1, j = 1, k = 1;
 int elem = 1;
int max, amax, bmax;
int rowa, rowb, cola, colb, vala, valb;

 /* get the total number of non-zero values from both matrices */
 amax = * (s1.sp + 2);
 bmax = * (s2.sp + 2);
 max = amax + bmax;
 /* allocate memory for result */
 p -> result = (int *) malloc (MAXSIZE * 3 * sizeof (int));

 while (elem <= max)
 {
  /* check if i < max. non-zero values in first 3-tuple and get the values */
  if (i <= amax)
  {
   rowa = * (s1.sp + i * 3 + 0);
   cola = * (s1.sp + i * 3 + 1);

vala = * (s1.sp + i * 3 + 2);
  }
```

```
else
 rowa = cola = BIGNUM;

/* check if j < max. non-zero values in second 3-tuple and get the values */
if (j <= bmax)
{
 rowb = * (s2.sp + j * 3 + 0);
 colb = * (s2.sp + j * 3 + 1);
 valb = * (s2.sp + j * 3 + 2);
}
else
 rowb = colb = BIGNUM;

/* if row no. of both 3-tuple are same */
if (rowa == rowb)
{

 /* if col no. of both 3-tuple are same */
 if (cola == colb)
 {
  /* add tow non-zero values store in result */
  * (p -> result + k * 3 + 0) = rowa;
  * (p -> result + k * 3 + 1) = cola;
  * (p -> result + k * 3 + 2) = vala + valb;
  i++;
  j++;
  max--;
 }
```

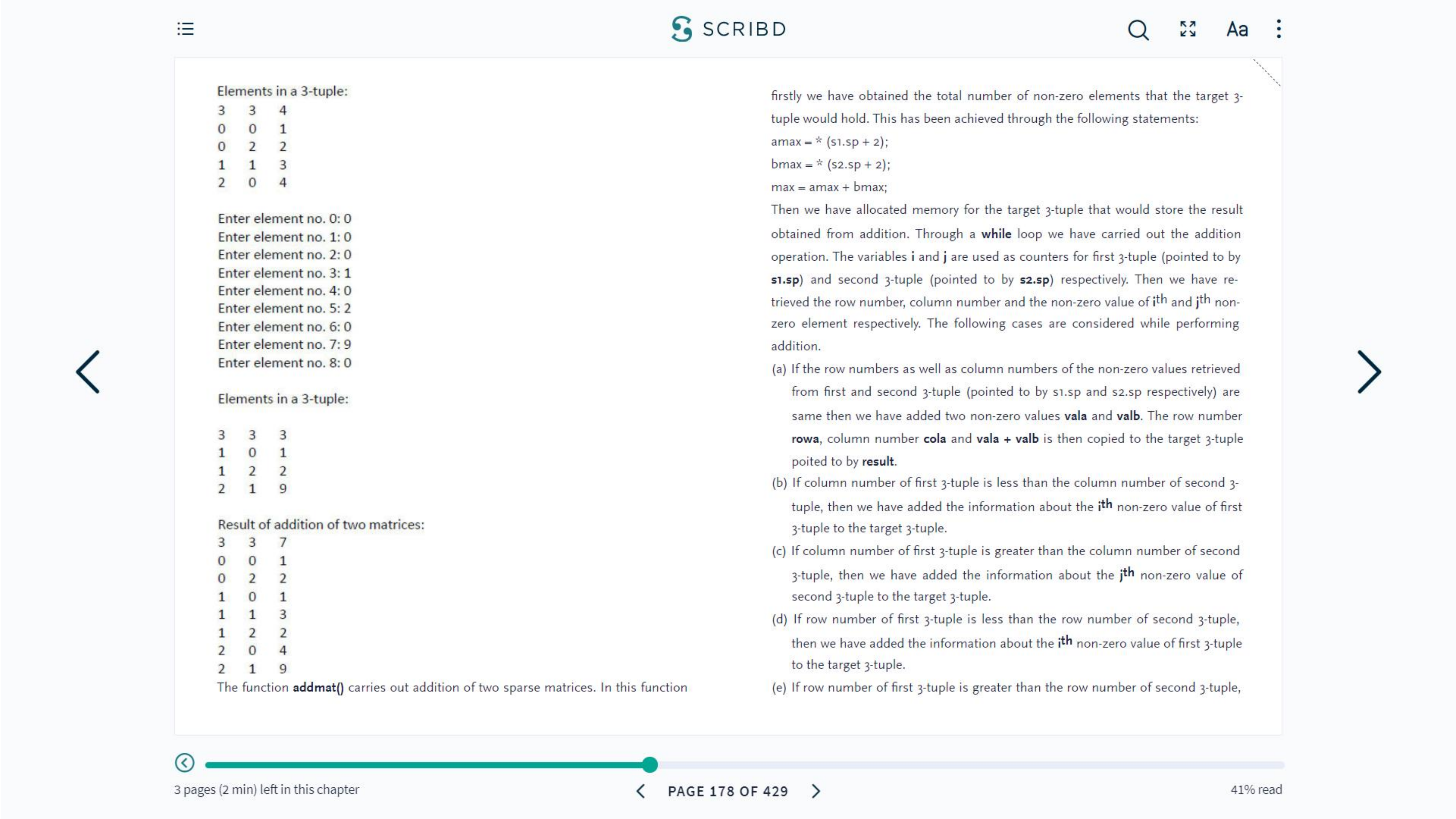


```
   /* if col no. of first 3-tuple is < col no. of second 3-tuple, then add info. as it
is to result */
   if (cola < colb)
   {
    * (p -> result + k * 3 + 0) = rowa;
    * (p -> result + k * 3 + 1) = cola;
    * (p -> result + k * 3 + 2) = vala;
    i++;
   }
/* if col no. of first 3-tuple is > col no. of second 3-tuple, then add info. as it is to
result */
   if (cola > colb)
   {
    * (p -> result + k * 3 + 0) = rowb;
    * (p -> result + k * 3 + 1) = colb;
    * (p -> result + k * 3 + 2) = valb;
    j++;
   }
   k++;
  }
  /* if row no. of first 3-tuple is < row no. of second 3-tuple, then add info. as it
is to result */
  if (rowa < rowb)
  {
   * (p -> result + k * 3 + 0) = rowa;
   * (p -> result + k * 3 + 1) = cola;
   * (p -> result + k * 3 + 2) = vala;
   i++;
   k++;
  }
  /* if row no. of first 3-tuple is > row no. of second 3-tuple, then add info. as it
is to result */
  if (rowa > rowb)
  {
   * (p -> result + k * 3 + 0) = rowb;
   * (p -> result + k * 3 + 1) = colb;
   * (p -> result + k * 3 + 2) = valb;
   j++;
   k++;
  }
  elem++;
 }
 /* add info about the total no. of rows, cols, and non-zero values that the resul-
tant array contains to the result */
 * (p -> result + 0) = MAX1;
 * (p -> result + 1) = MAX2;
* (p -> result + 2) = max;
}
/* displays the contents of the matrix */
void display_result (struct sparse s)
```

```
{
 int i;

 /* traverses the entire matrix */
 for (i = 0; i < (* (s.result + 0 + 2) + 1) * 3; i++)
 {
  /* positions the cursor to the new line for every new row */
  if (i % 3 == 0)
   printf (“\n”);
  printf (“%d\t”, * (s.result + i));
 }
 printf (“\n\n”);
}

/* deallocates memory */
void delsparse (struct sparse *p)
{
 if (p -> sp != NULL)
  free (p -> sp);
 if (p -> result != NULL)
  free (p -> result);
}
```

Output:

```
Enter element no. 0: 1
Enter element no. 1: 0
Enter element no. 2: 2
Enter element no. 3: 0
Enter element no. 4: 3
Enter element no. 5: 0
Enter element no. 6: 4
Enter element no. 7: 0
Enter element no. 8: 0
```



```
Elements in a 3-tuple:
3    3    4
0    0    1
0    2    2
1    1    3
2    0    4

Enter element no. 0: 0
Enter element no. 1: 0
Enter element no. 2: 0
Enter element no. 3: 1
Enter element no. 4: 0
Enter element no. 5: 2
Enter element no. 6: 0
Enter element no. 7: 9
Enter element no. 8: 0

Elements in a 3-tuple:

3    3    3
1    0    1
1    2    2
2    1    9

Result of addition of two matrices:
3    3    7
0    0    1
0    2    2
1    0    1
1    1    3
1    2    2
2    0    4
2    1    9
```

The function **addmat()** carries out addition of two sparse matrices. In this function firstly we have obtained the total number of non-zero elements that the target 3-tuple would hold. This has been achieved through the following statements:

```
amax = * (s1.sp + 2);
bmax = * (s2.sp + 2);
max = amax + bmax;
```

Then we have allocated memory for the target 3-tuple that would store the result obtained from addition. Through a **while** loop we have carried out the addition operation. The variables **i** and **j** are used as counters for first 3-tuple (pointed to by **s1.sp**) and second 3-tuple (pointed to by **s2.sp**) respectively. Then we have retrieved the row number, column number and the non-zero value of i^{th} and j^{th} non-zero element respectively. The following cases are considered while performing addition.

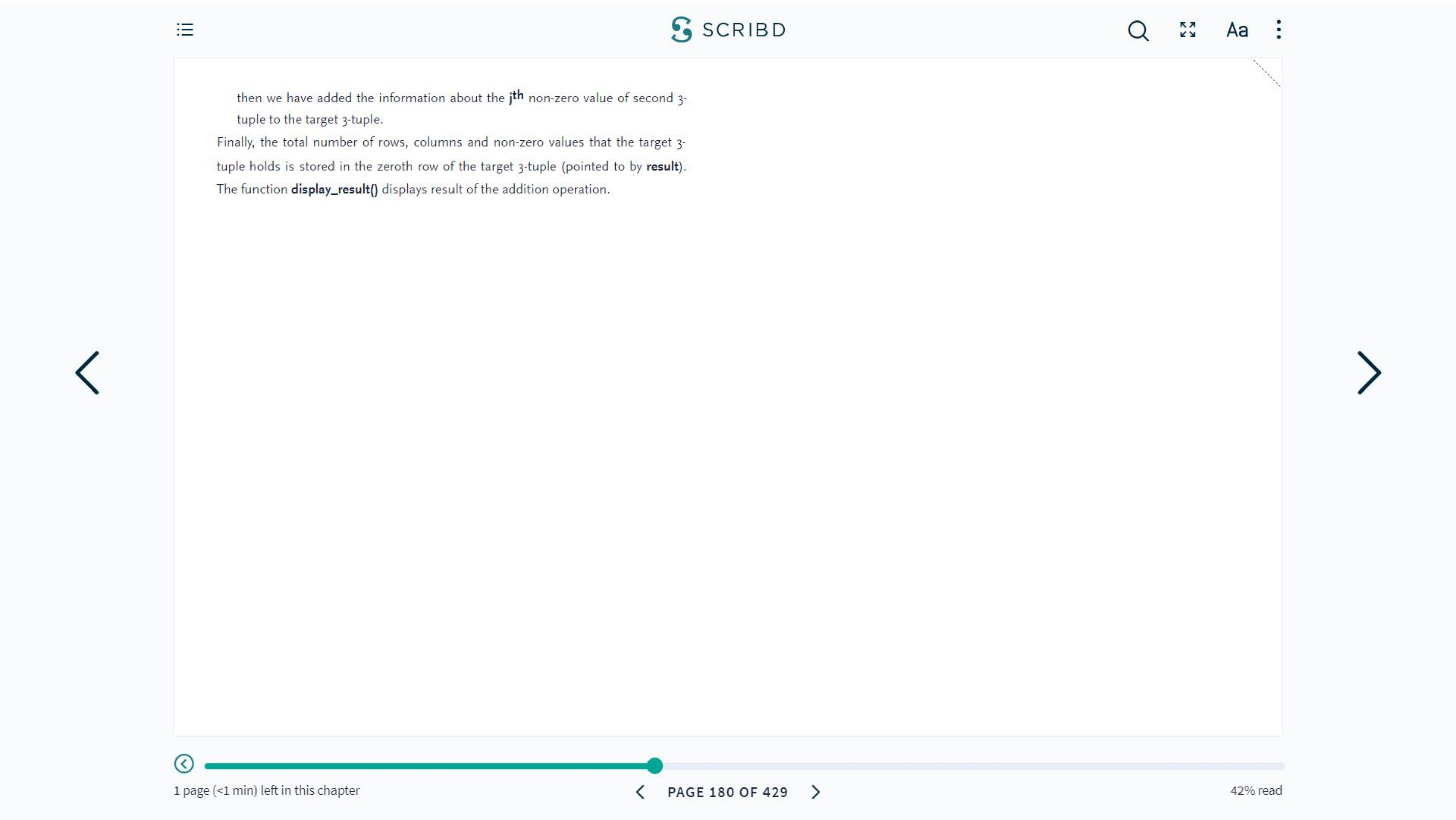
(a) If the row numbers as well as column numbers of the non-zero values retrieved from first and second 3-tuple (pointed to by s1.sp and s2.sp respectively) are same then we have added two non-zero values **vala** and **valb**. The row number **rowa**, column number **cola** and **vala + valb** is then copied to the target 3-tuple poited to by **result**.

(b) If column number of first 3-tuple is less than the column number of second 3-tuple, then we have added the information about the i^{th} non-zero value of first 3-tuple to the target 3-tuple.

(c) If column number of first 3-tuple is greater than the column number of second 3-tuple, then we have added the information about the j^{th} non-zero value of second 3-tuple to the target 3-tuple.

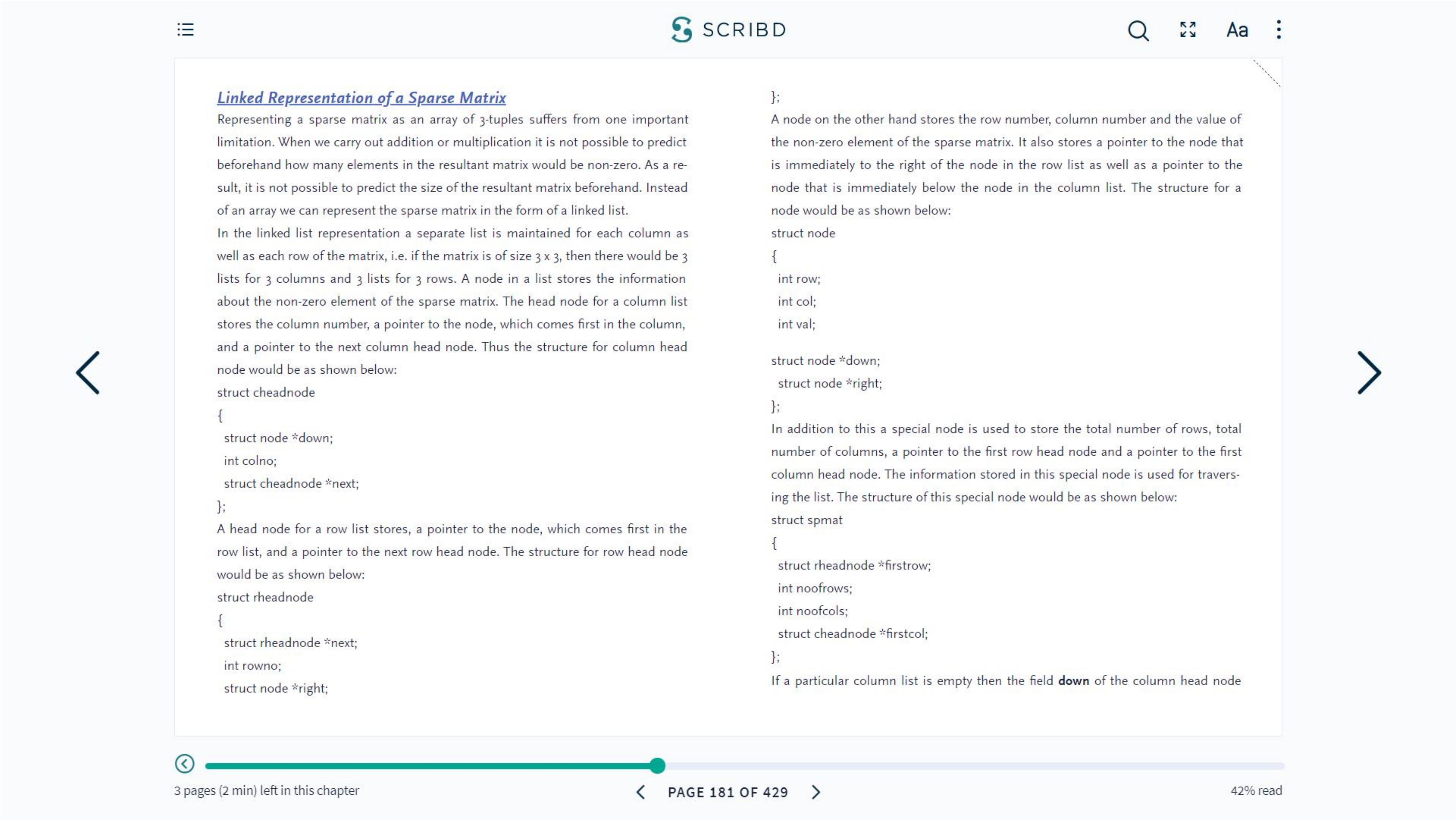
(d) If row number of first 3-tuple is less than the row number of second 3-tuple, then we have added the information about the i^{th} non-zero value of first 3-tuple to the target 3-tuple.

(e) If row number of first 3-tuple is greater than the row number of second 3-tuple,



then we have added the information about the **jth** non-zero value of second 3-tuple to the target 3-tuple.

Finally, the total number of rows, columns and non-zero values that the target 3-tuple holds is stored in the zeroth row of the target 3-tuple (pointed to by **result**). The function **display_result()** displays result of the addition operation.



Linked Representation of a Sparse Matrix

Representing a sparse matrix as an array of 3-tuples suffers from one important limitation. When we carry out addition or multiplication it is not possible to predict beforehand how many elements in the resultant matrix would be non-zero. As a result, it is not possible to predict the size of the resultant matrix beforehand. Instead of an array we can represent the sparse matrix in the form of a linked list.

In the linked list representation a separate list is maintained for each column as well as each row of the matrix, i.e. if the matrix is of size 3 x 3, then there would be 3 lists for 3 columns and 3 lists for 3 rows. A node in a list stores the information about the non-zero element of the sparse matrix. The head node for a column list stores the column number, a pointer to the node, which comes first in the column, and a pointer to the next column head node. Thus the structure for column head node would be as shown below:

```
struct cheadnode
{
 struct node *down;
 int colno;
 struct cheadnode *next;
};
```

A head node for a row list stores, a pointer to the node, which comes first in the row list, and a pointer to the next row head node. The structure for row head node would be as shown below:

```
struct rheadnode
{
 struct rheadnode *next;
 int rowno;
 struct node *right;
};
```

A node on the other hand stores the row number, column number and the value of the non-zero element of the sparse matrix. It also stores a pointer to the node that is immediately to the right of the node in the row list as well as a pointer to the node that is immediately below the node in the column list. The structure for a node would be as shown below:

```
struct node
{
 int row;
 int col;
 int val;

struct node *down;
 struct node *right;
};
```

In addition to this a special node is used to store the total number of rows, total number of columns, a pointer to the first row head node and a pointer to the first column head node. The information stored in this special node is used for traversing the list. The structure of this special node would be as shown below:

```
struct spmat
{
 struct rheadnode *firstrow;
 int noofrows;
 int noofcols;
 struct cheadnode *firstcol;
};
```

If a particular column list is empty then the field **down** of the column head node would be NULL. Similarly if a row list is empty then the field **right** of the row head node would be empty. If a node is the last node in a particular column list or a particular row list then the field **down** or the field **right** of the node would be NULL.

Figure 4-2 gives pictorial representation of linked list of a sparse matrix of size 3 x 3.

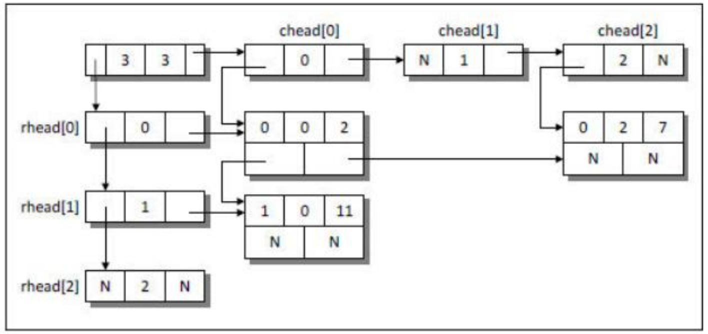
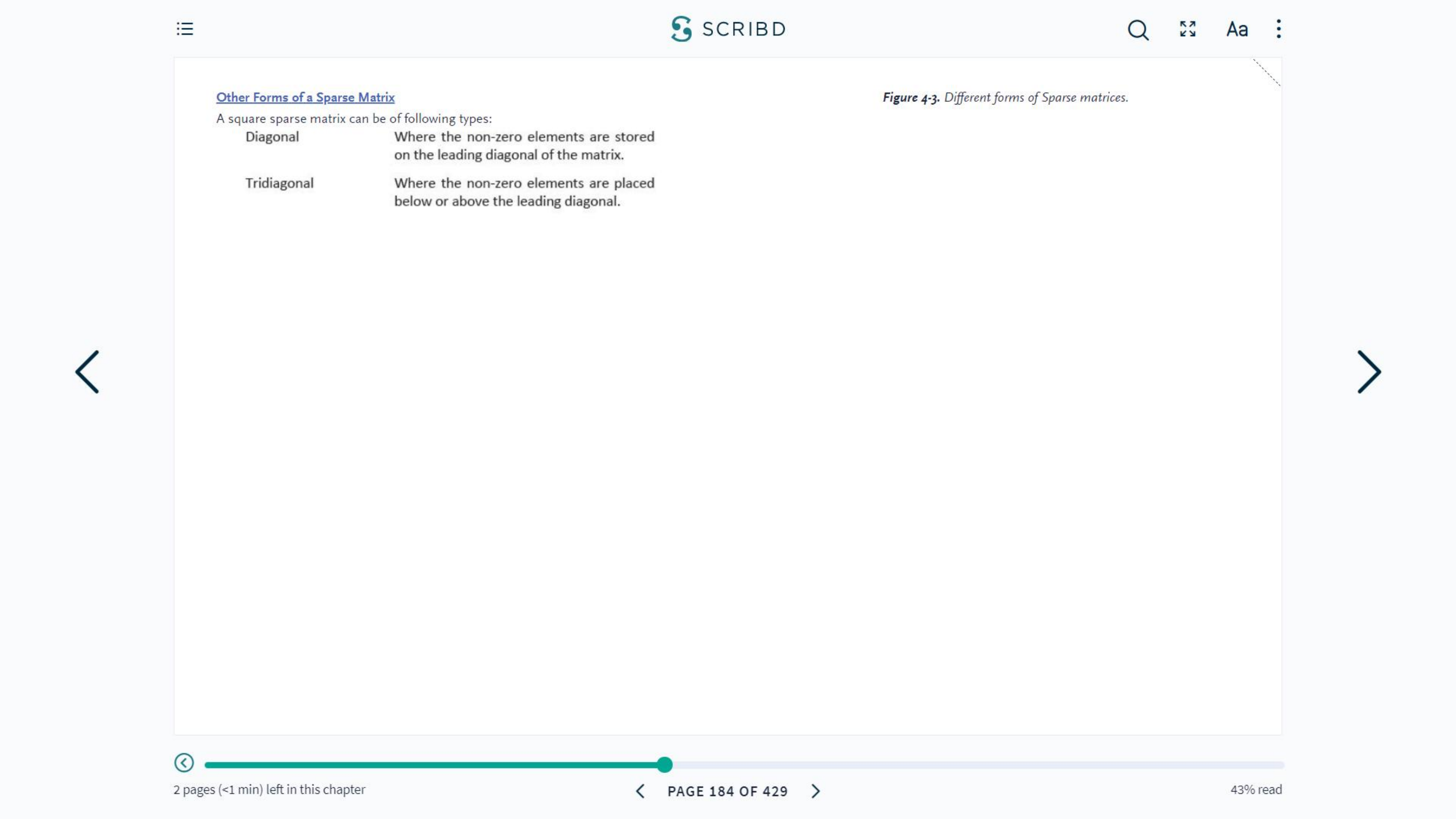


Figure 4-2. *Linked Representation of a sparse matrix.*

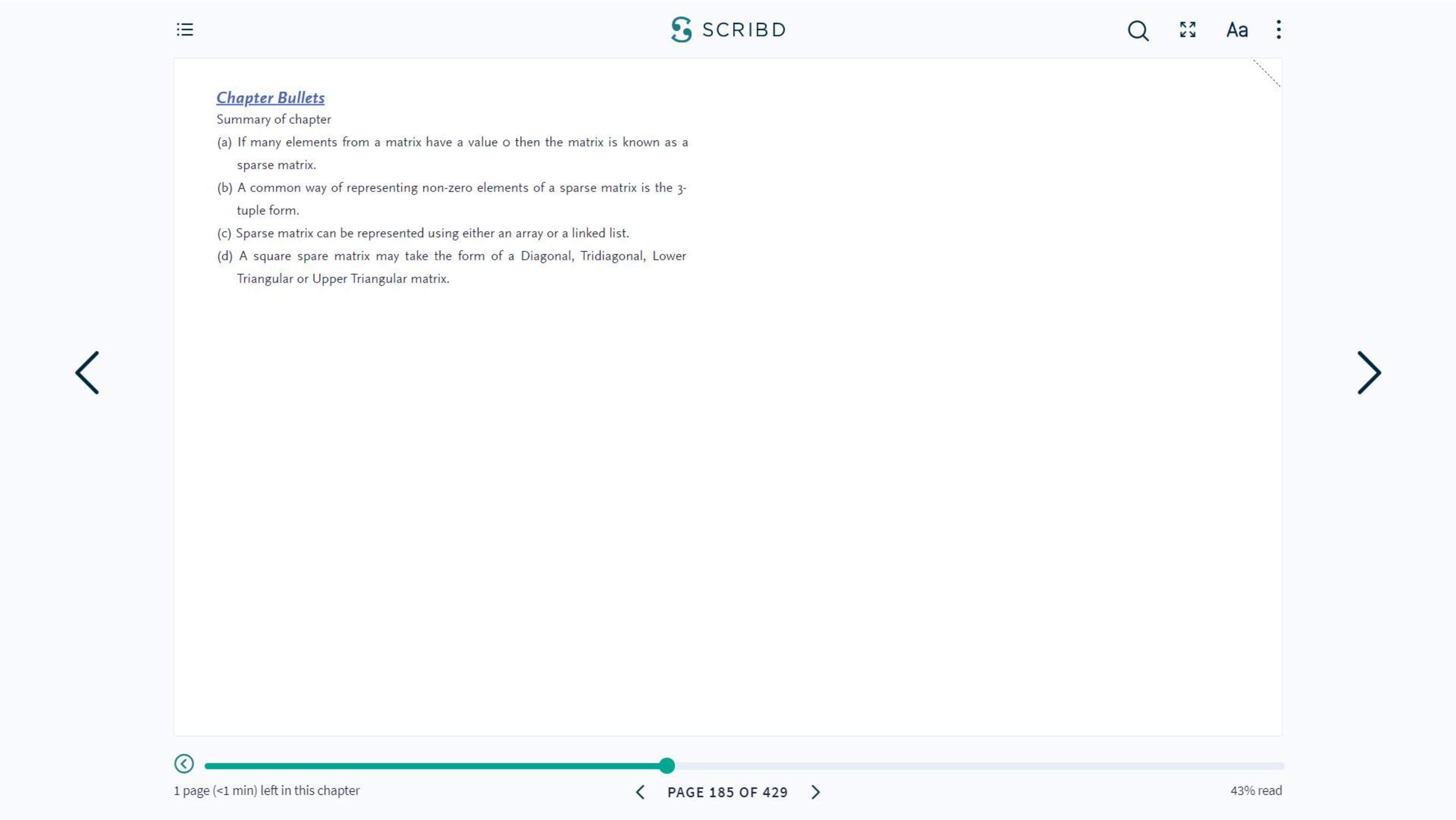


Other Forms of a Sparse Matrix

A square sparse matrix can be of following types:

Diagonal	Where the non-zero elements are stored on the leading diagonal of the matrix.
Tridiagonal	Where the non-zero elements are placed below or above the leading diagonal.

***Figure* 4-3.** *Different forms of Sparse matrices.*



Chapter Bullets

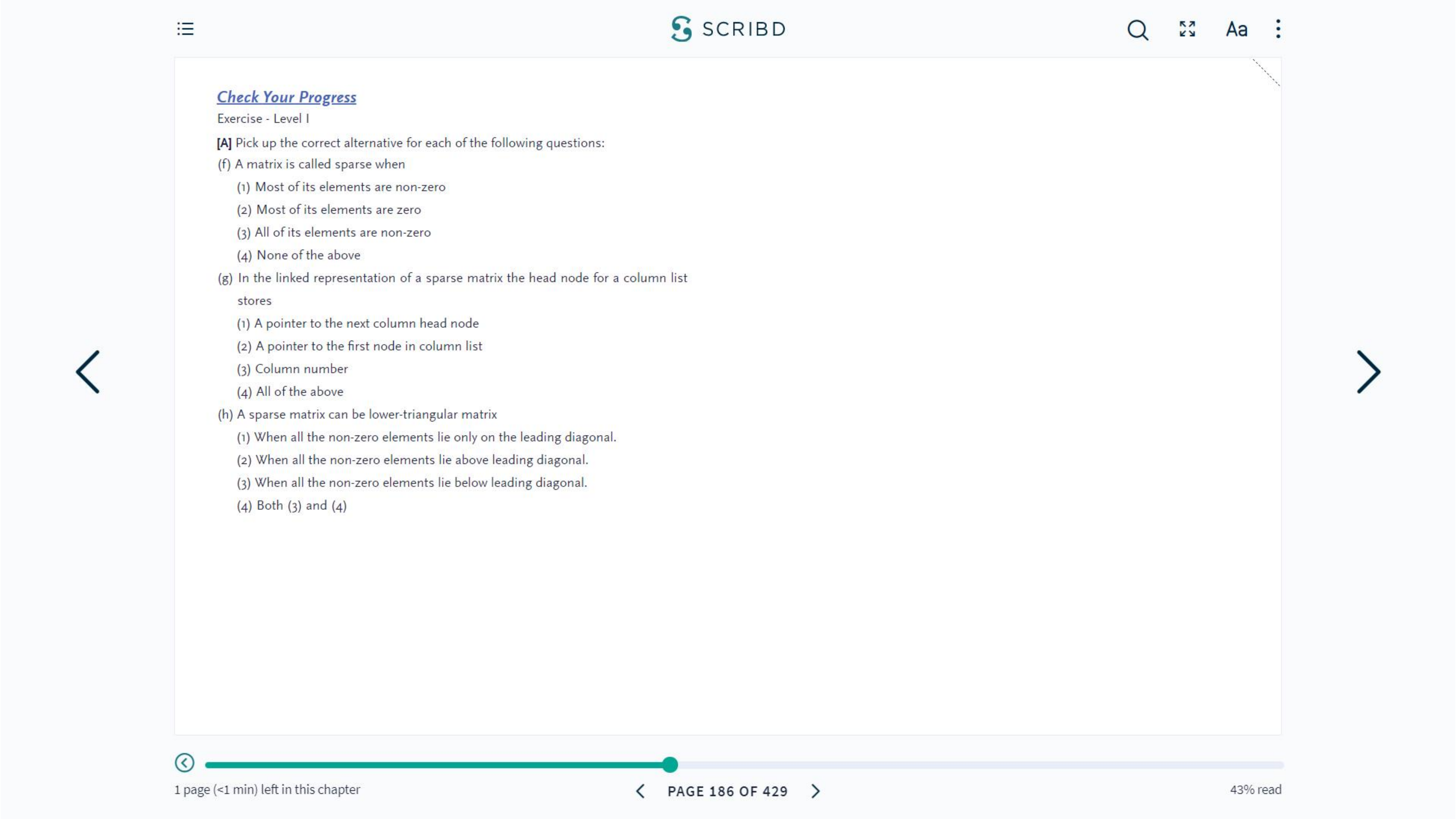
Summary of chapter

(a) If many elements from a matrix have a value 0 then the matrix is known as a sparse matrix.

(b) A common way of representing non-zero elements of a sparse matrix is the 3-tuple form.

(c) Sparse matrix can be represented using either an array or a linked list.

(d) A square spare matrix may take the form of a Diagonal, Tridiagonal, Lower Triangular or Upper Triangular matrix.



Check Your Progress

Exercise - Level I

[A] Pick up the correct alternative for each of the following questions:

(f) A matrix is called sparse when

(1) Most of its elements are non-zero

(2) Most of its elements are zero

(3) All of its elements are non-zero

(4) None of the above

(g) In the linked representation of a sparse matrix the head node for a column list stores

(1) A pointer to the next column head node

(2) A pointer to the first node in column list

(3) Column number

(4) All of the above

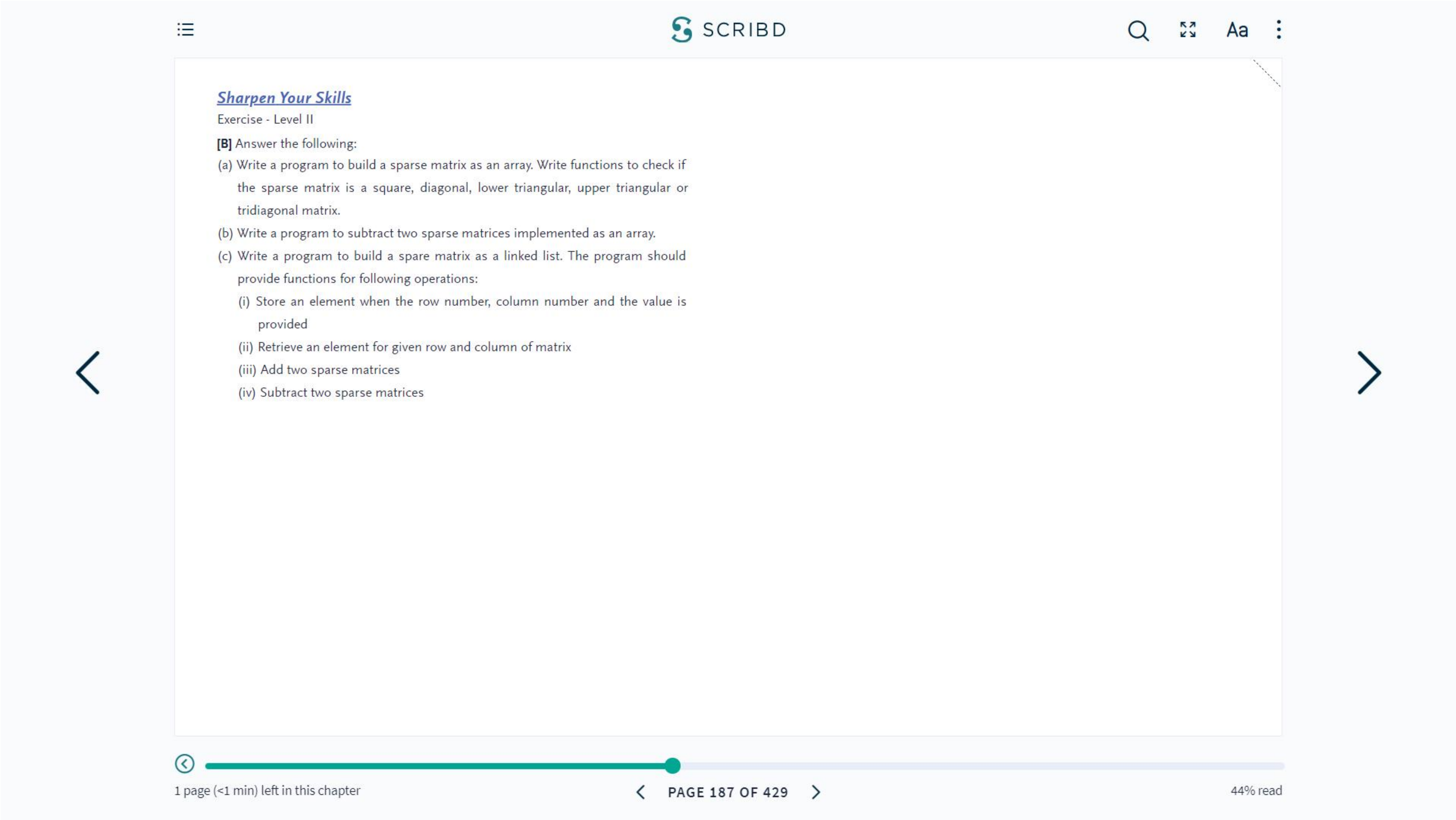
(h) A sparse matrix can be lower-triangular matrix

(1) When all the non-zero elements lie only on the leading diagonal.

(2) When all the non-zero elements lie above leading diagonal.

(3) When all the non-zero elements lie below leading diagonal.

(4) Both (3) and (4)



Sharpen Your Skills

Exercise - Level II

[B] Answer the following:

(a) Write a program to build a sparse matrix as an array. Write functions to check if the sparse matrix is a square, diagonal, lower triangular, upper triangular or tridiagonal matrix.

(b) Write a program to subtract two sparse matrices implemented as an array.

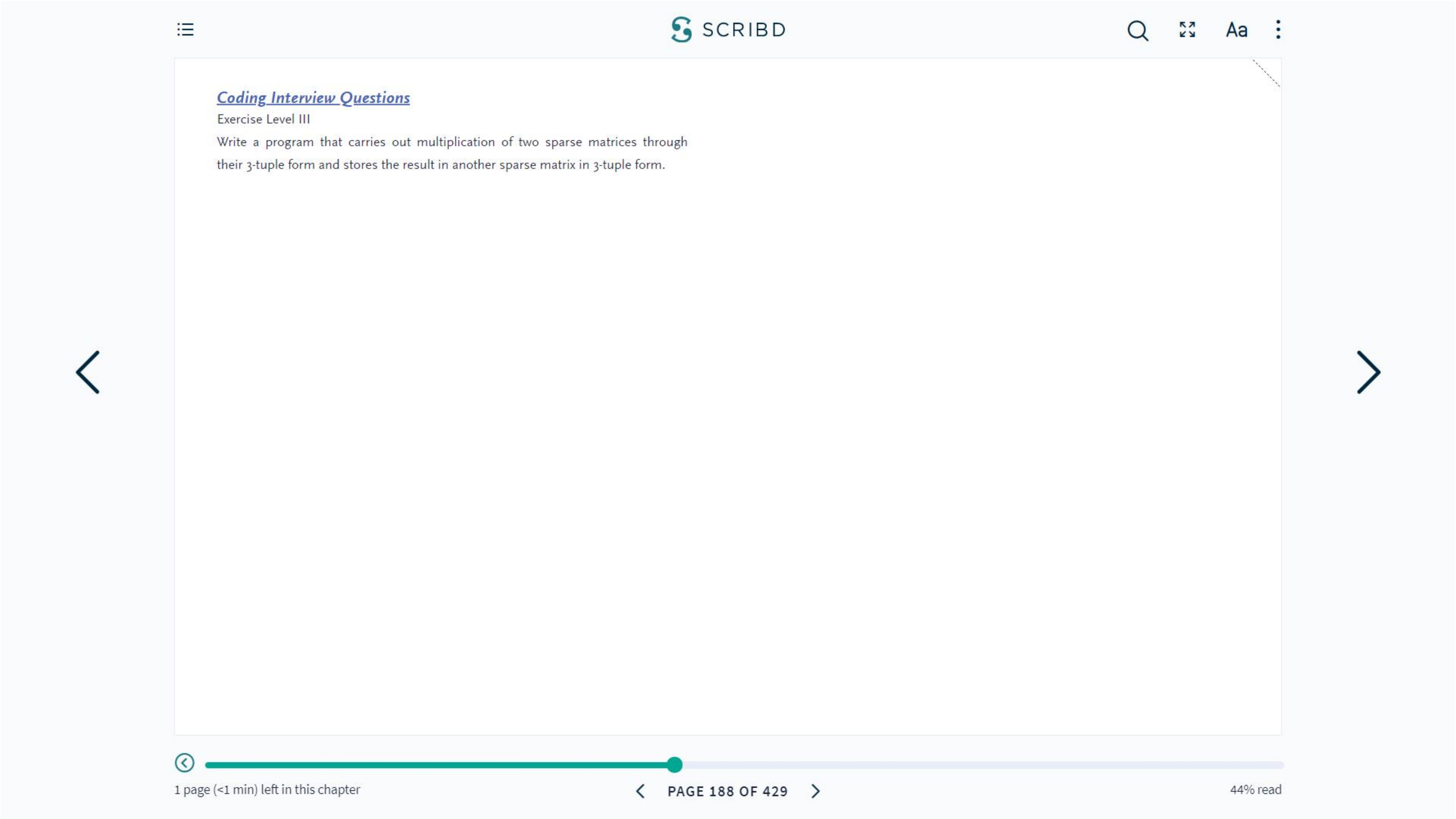
(c) Write a program to build a spare matrix as a linked list. The program should provide functions for following operations:

(i) Store an element when the row number, column number and the value is provided

(ii) Retrieve an element for given row and column of matrix

(iii) Add two sparse matrices

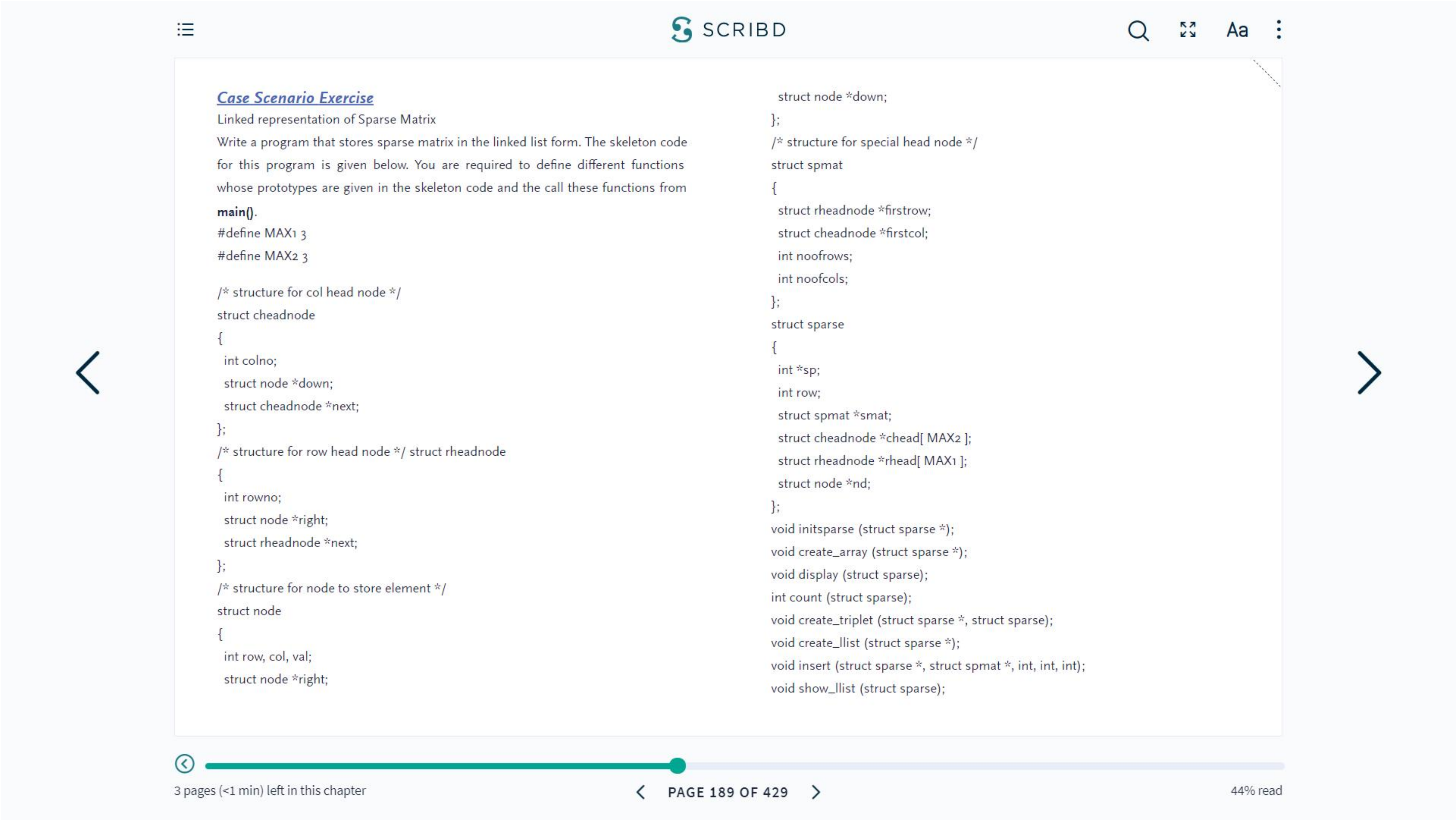
(iv) Subtract two sparse matrices



Coding Interview Questions

Exercise Level III

Write a program that carries out multiplication of two sparse matrices through their 3-tuple form and stores the result in another sparse matrix in 3-tuple form.

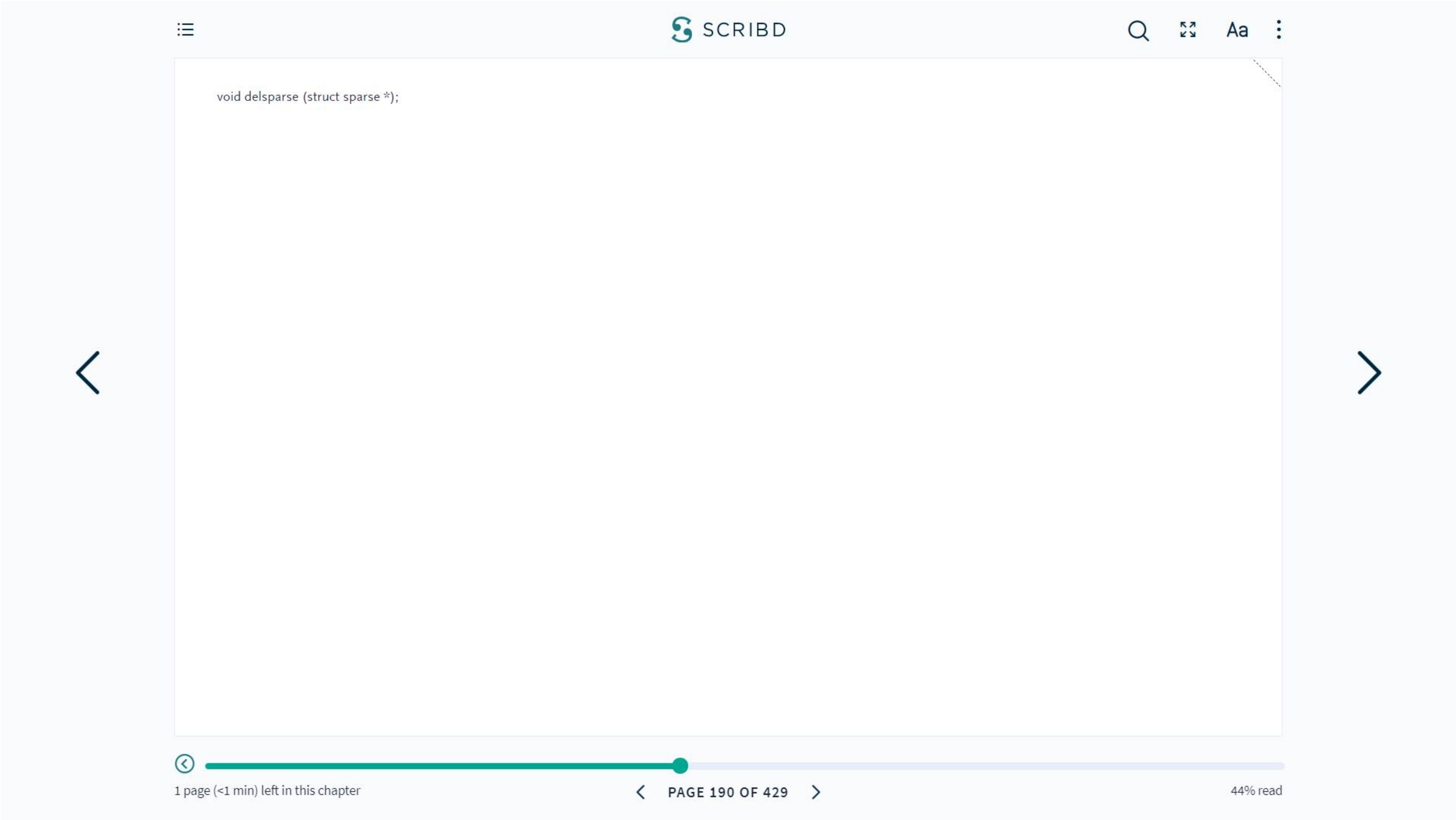


Case Scenario Exercise

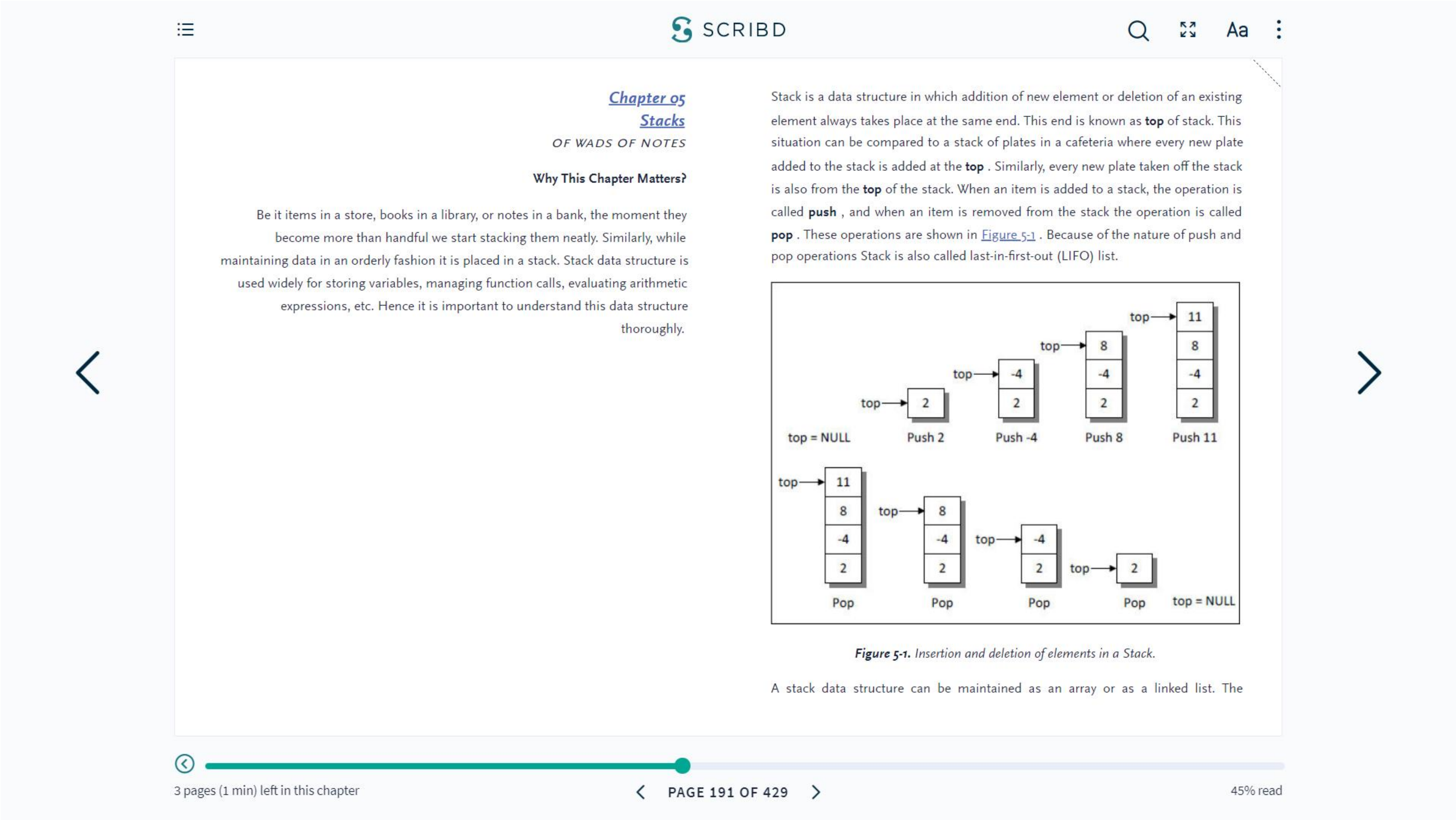
Linked representation of Sparse Matrix

Write a program that stores sparse matrix in the linked list form. The skeleton code for this program is given below. You are required to define different functions whose prototypes are given in the skeleton code and the call these functions from **main()**.

```
#define MAX1 3
#define MAX2 3

/* structure for col head node */
struct cheadnode
{
 int colno;
 struct node *down;
 struct cheadnode *next;
};
/* structure for row head node */ struct rheadnode
{
 int rowno;
 struct node *right;
 struct rheadnode *next;
};
/* structure for node to store element */
struct node
{
 int row, col, val;
 struct node *right;
 struct node *down;
};
/* structure for special head node */
struct spmat
{
 struct rheadnode *firstrow;
 struct cheadnode *firstcol;
 int noofrows;
 int noofcols;
};
struct sparse
{
 int *sp;
 int row;
 struct spmat *smat;
 struct cheadnode *chead[ MAX2 ];
 struct rheadnode *rhead[ MAX1 ];
 struct node *nd;
};
void initsparse (struct sparse *);
void create_array (struct sparse *);
void display (struct sparse);
int count (struct sparse);
void create_triplet (struct sparse *, struct sparse);
void create_llist (struct sparse *);
void insert (struct sparse *, struct spmat *, int, int, int);
void show_llist (struct sparse);
```



```
void delsparse (struct sparse *);
```



Chapter 05 Stacks

OF WADS OF NOTES

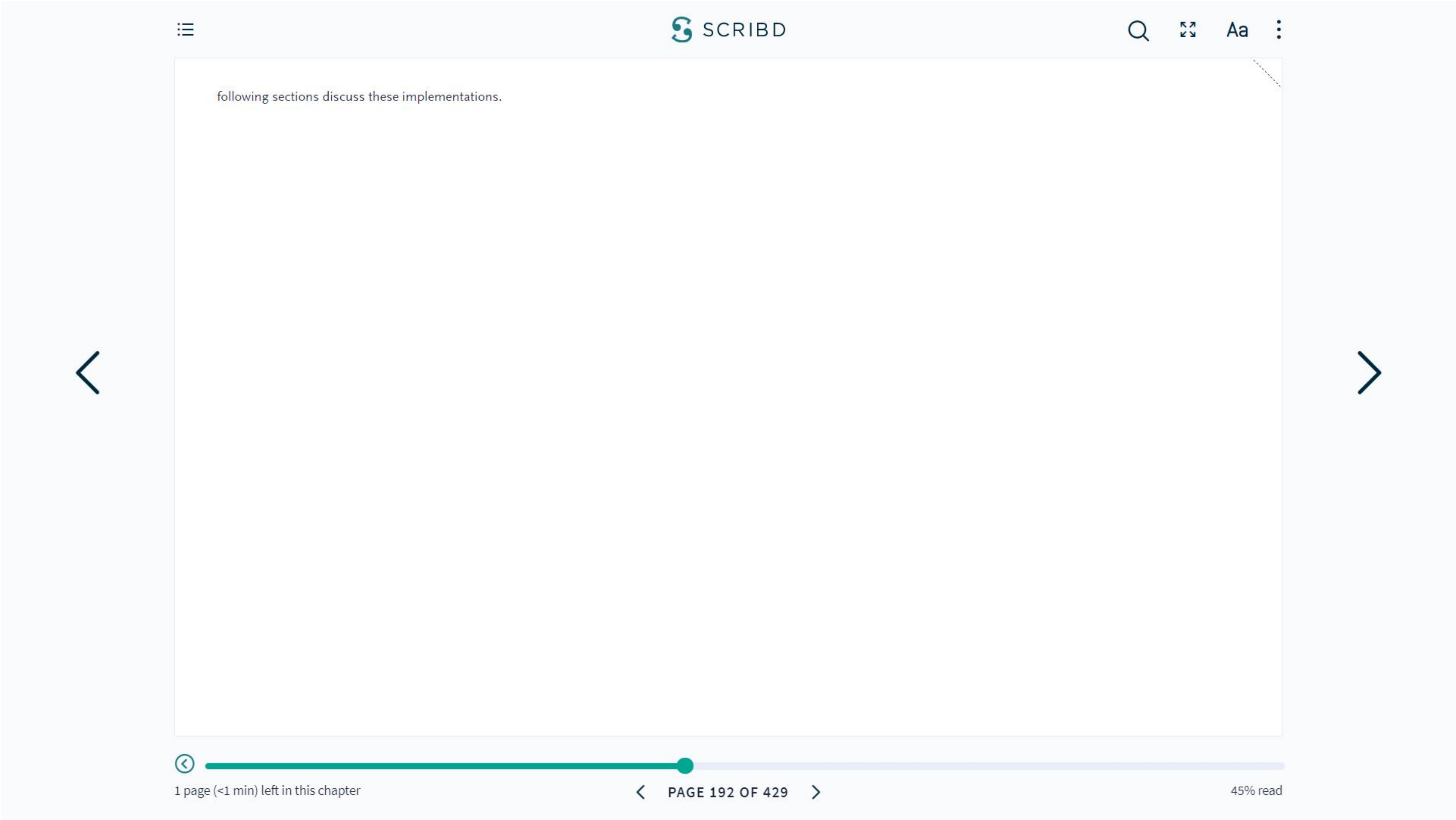
Why This Chapter Matters?

Be it items in a store, books in a library, or notes in a bank, the moment they become more than handful we start stacking them neatly. Similarly, while maintaining data in an orderly fashion it is placed in a stack. Stack data structure is used widely for storing variables, managing function calls, evaluating arithmetic expressions, etc. Hence it is important to understand this data structure thoroughly.

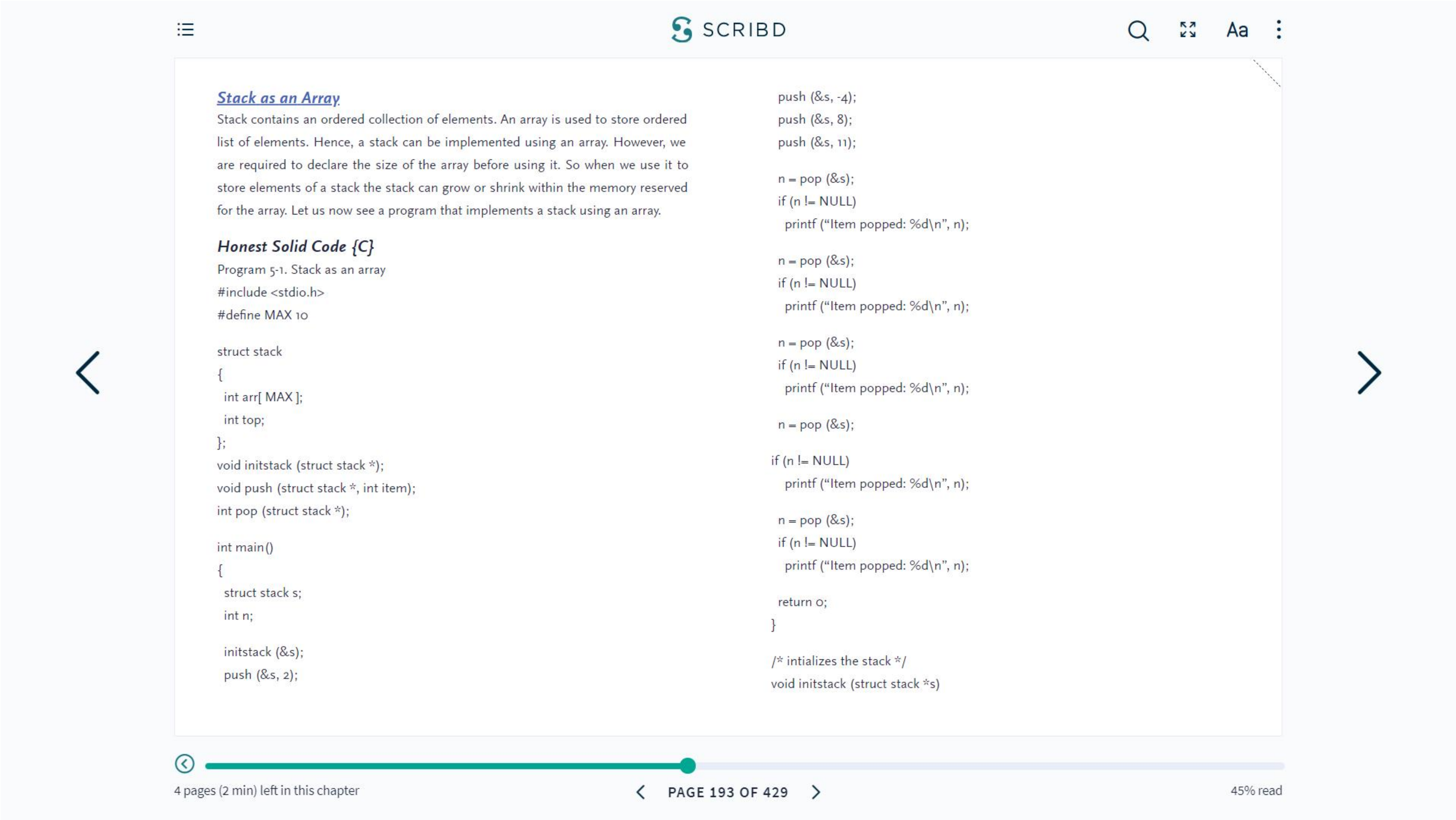
Stack is a data structure in which addition of new element or deletion of an existing element always takes place at the same end. This end is known as **top** of stack. This situation can be compared to a stack of plates in a cafeteria where every new plate added to the stack is added at the **top** . Similarly, every new plate taken off the stack is also from the **top** of the stack. When an item is added to a stack, the operation is called **push** , and when an item is removed from the stack the operation is called **pop** . These operations are shown in Figure 5-1 . Because of the nature of push and pop operations Stack is also called last-in-first-out (LIFO) list.

Figure 5-1. *Insertion and deletion of elements in a Stack.*

A stack data structure can be maintained as an array or as a linked list. The



following sections discuss these implementations.



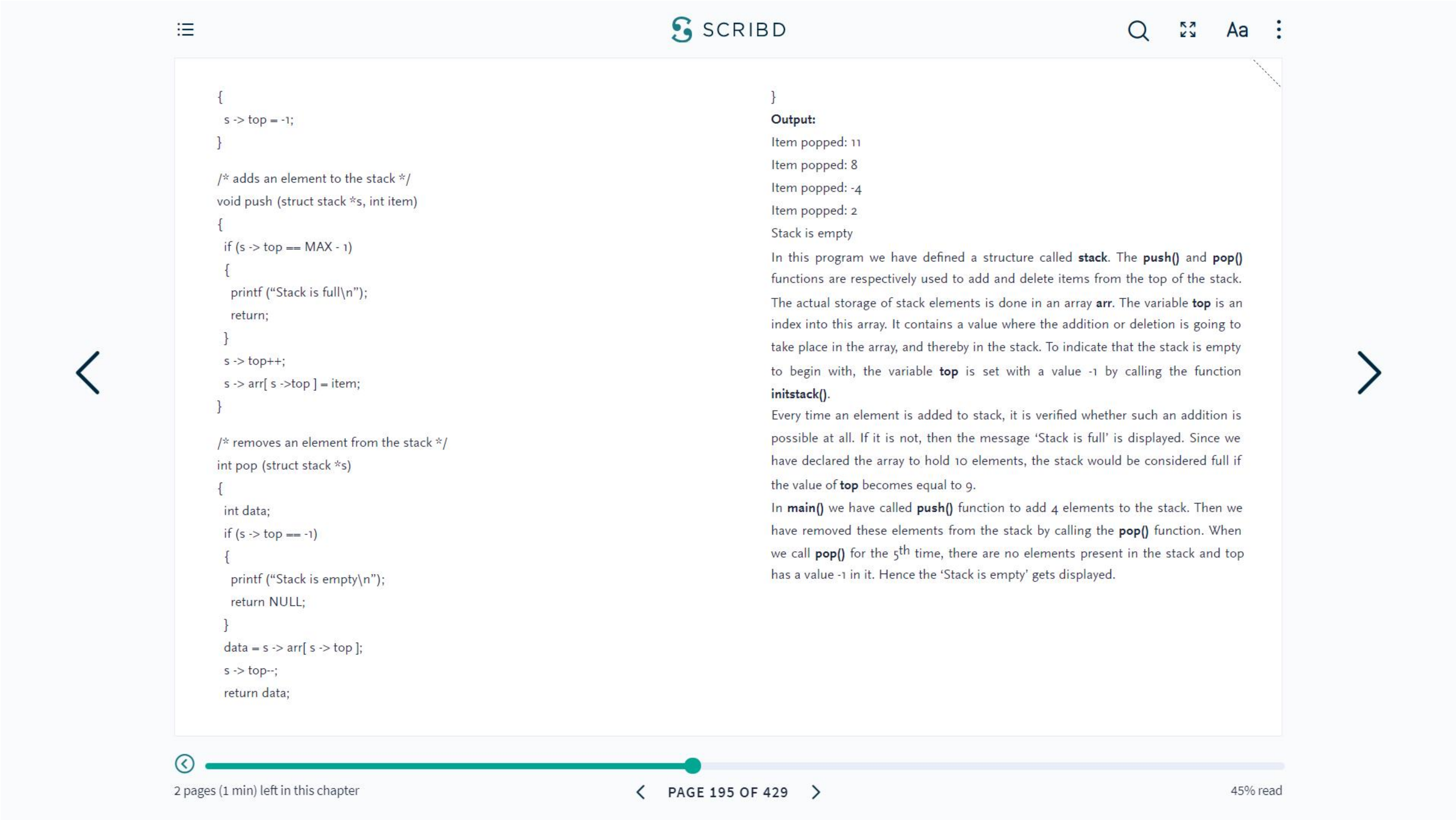
Stack as an Array

Stack contains an ordered collection of elements. An array is used to store ordered list of elements. Hence, a stack can be implemented using an array. However, we are required to declare the size of the array before using it. So when we use it to store elements of a stack the stack can grow or shrink within the memory reserved for the array. Let us now see a program that implements a stack using an array.

Honest Solid Code {C}

Program 5-1. Stack as an array

```
#include <stdio.h>
#define MAX 10

struct stack
{
 int arr[ MAX ];
 int top;
};
void initstack (struct stack *);
void push (struct stack *, int item);
int pop (struct stack *);

int main()
{
 struct stack s;
 int n;

 initstack (&s);
 push (&s, 2);
 push (&s, -4);
 push (&s, 8);
 push (&s, 11);

 n = pop (&s);
 if (n != NULL)
  printf (“Item popped: %d\n”, n);

 n = pop (&s);
 if (n != NULL)
  printf (“Item popped: %d\n”, n);

 n = pop (&s);
 if (n != NULL)
  printf (“Item popped: %d\n”, n);

 n = pop (&s);

if (n != NULL)
  printf (“Item popped: %d\n”, n);

 n = pop (&s);
 if (n != NULL)
  printf (“Item popped: %d\n”, n);

 return 0;
}

/* intializes the stack */
void initstack (struct stack *s)
```



```
{
 s -> top = -1;
}

/* adds an element to the stack */
void push (struct stack *s, int item)
{
 if (s -> top == MAX - 1)
 {
  printf ("Stack is full\n");
  return;
 }
 s -> top++;
 s -> arr[ s ->top ] = item;
}

/* removes an element from the stack */
int pop (struct stack *s)
{
 int data;
 if (s -> top == -1)
 {
  printf ("Stack is empty\n");
  return NULL;
 }
 data = s -> arr[ s -> top ];
 s -> top--;
 return data;
}
```

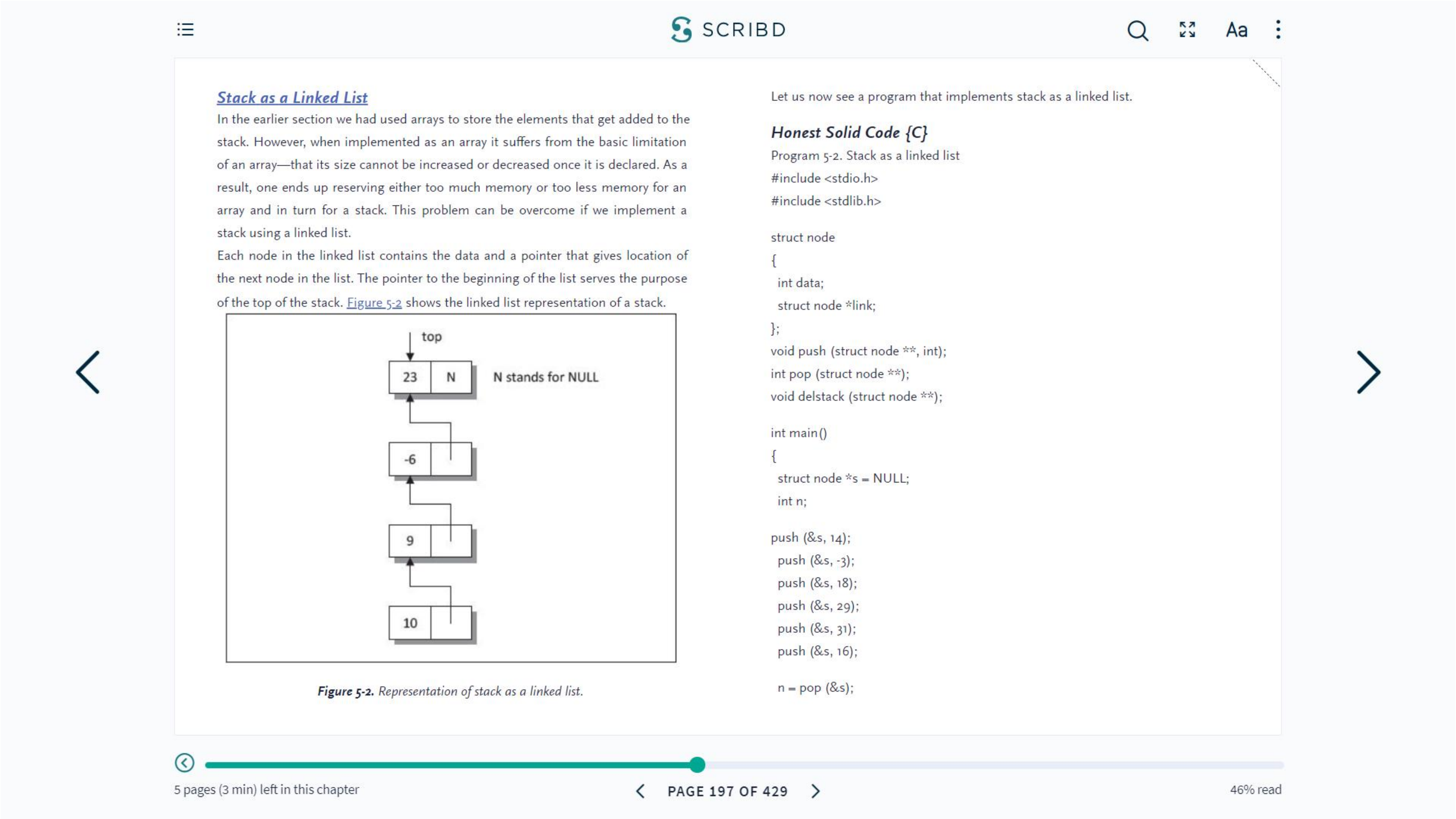
Output:

```
Item popped: 11
Item popped: 8
Item popped: -4
Item popped: 2
Stack is empty
```

In this program we have defined a structure called **stack**. The **push()** and **pop()** functions are respectively used to add and delete items from the top of the stack. The actual storage of stack elements is done in an array **arr**. The variable **top** is an index into this array. It contains a value where the addition or deletion is going to take place in the array, and thereby in the stack. To indicate that the stack is empty to begin with, the variable **top** is set with a value -1 by calling the function **initstack()**.

Every time an element is added to stack, it is verified whether such an addition is possible at all. If it is not, then the message 'Stack is full' is displayed. Since we have declared the array to hold 10 elements, the stack would be considered full if the value of **top** becomes equal to 9.

In **main()** we have called **push()** function to add 4 elements to the stack. Then we have removed these elements from the stack by calling the **pop()** function. When we call **pop()** for the 5th time, there are no elements present in the stack and top has a value -1 in it. Hence the 'Stack is empty' gets displayed.



Stack as a Linked List

In the earlier section we had used arrays to store the elements that get added to the stack. However, when implemented as an array it suffers from the basic limitation of an array—that its size cannot be increased or decreased once it is declared. As a result, one ends up reserving either too much memory or too less memory for an array and in turn for a stack. This problem can be overcome if we implement a stack using a linked list.

Each node in the linked list contains the data and a pointer that gives location of the next node in the list. The pointer to the beginning of the list serves the purpose of the top of the stack. Figure 5-2 shows the linked list representation of a stack.

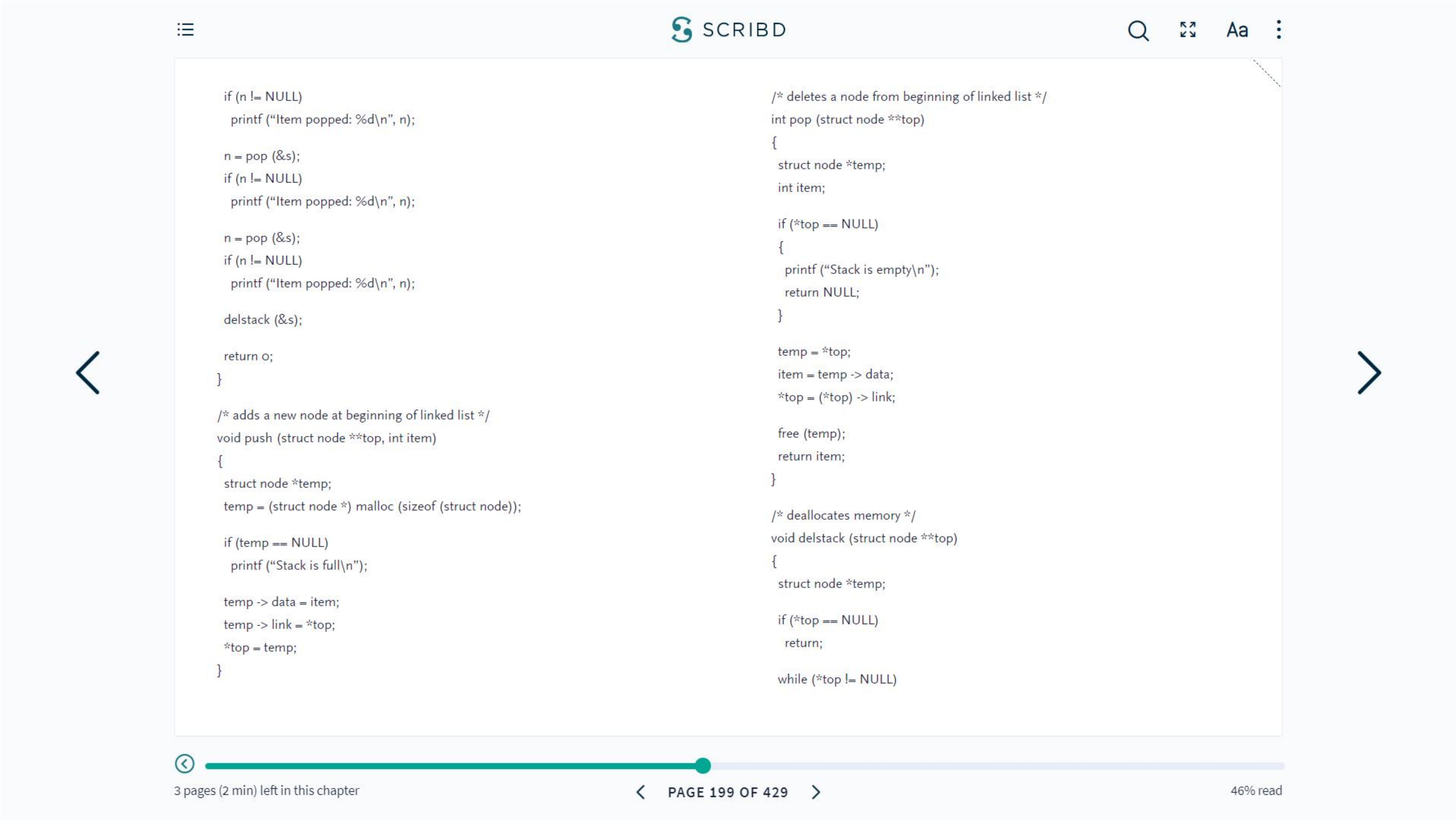
Figure 5-2. *Representation of stack as a linked list.*

Let us now see a program that implements stack as a linked list.

Honest Solid Code {C}

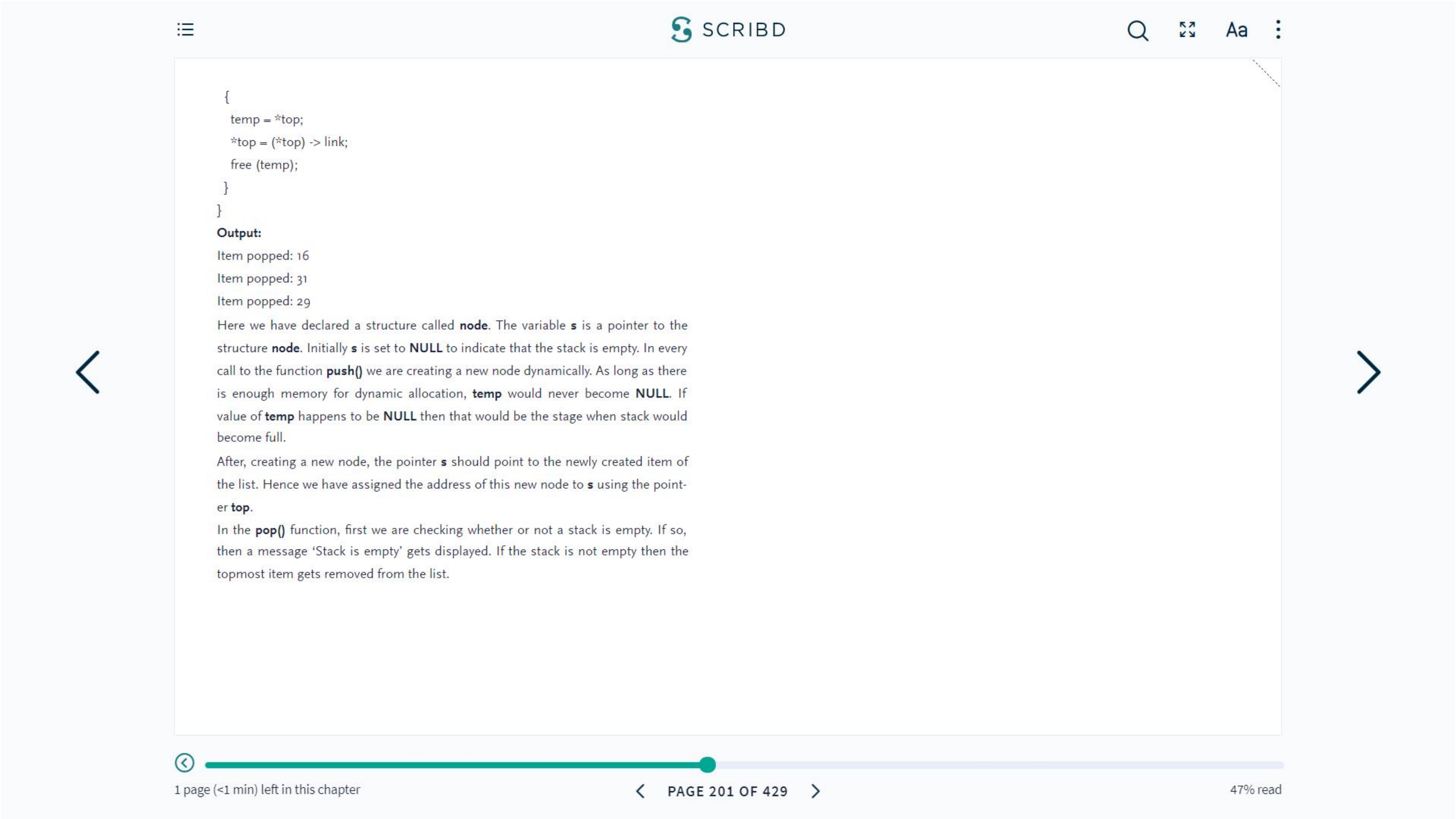
Program 5-2. Stack as a linked list

```
#include <stdio.h>
#include <stdlib.h>

struct node
{
 int data;
 struct node *link;
};
void push (struct node **, int);
int pop (struct node **);
void delstack (struct node **);

int main()
{
 struct node *s = NULL;
 int n;

push (&s, 14);
 push (&s, -3);
 push (&s, 18);
 push (&s, 29);
 push (&s, 31);
 push (&s, 16);

 n = pop (&s);
```



```
 if (n != NULL)
  printf (“Item popped: %d\n”, n);

 n = pop (&s);
 if (n != NULL)
  printf (“Item popped: %d\n”, n);

 n = pop (&s);
 if (n != NULL)
  printf (“Item popped: %d\n”, n);

 delstack (&s);

 return 0;
}

/* adds a new node at beginning of linked list */
void push (struct node **top, int item)
{
 struct node *temp;
 temp = (struct node *) malloc (sizeof (struct node));

 if (temp == NULL)
  printf (“Stack is full\n”);

 temp -> data = item;
 temp -> link = *top;
 *top = temp;
}
```

```
/* deletes a node from beginning of linked list */
int pop (struct node **top)
{
 struct node *temp;
 int item;

 if (*top == NULL)
 {
  printf (“Stack is empty\n”);
  return NULL;
 }

 temp = *top;
 item = temp -> data;
 *top = (*top) -> link;

 free (temp);
 return item;
}

/* deallocates memory */
void delstack (struct node **top)
{
 struct node *temp;

 if (*top == NULL)
  return;

 while (*top != NULL)
```



```
 {
  temp = *top;
  *top = (*top) -> link;
  free (temp);
 }
}
```

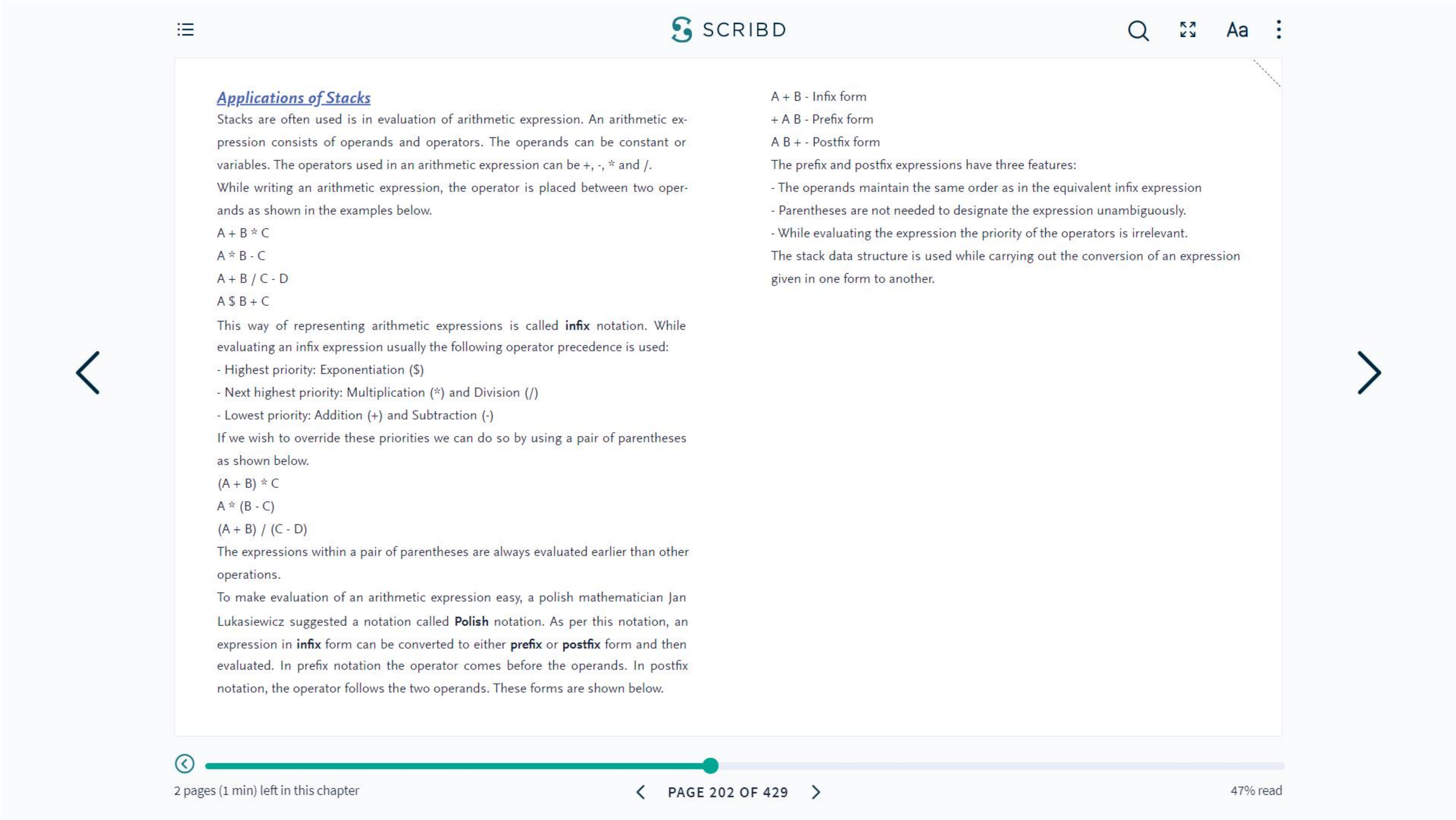
Output:

```
Item popped: 16
Item popped: 31
Item popped: 29
```

Here we have declared a structure called **node**. The variable **s** is a pointer to the structure **node**. Initially **s** is set to **NULL** to indicate that the stack is empty. In every call to the function **push()** we are creating a new node dynamically. As long as there is enough memory for dynamic allocation, **temp** would never become **NULL**. If value of **temp** happens to be **NULL** then that would be the stage when stack would become full.

After, creating a new node, the pointer **s** should point to the newly created item of the list. Hence we have assigned the address of this new node to **s** using the pointer **top**.

In the **pop()** function, first we are checking whether or not a stack is empty. If so, then a message 'Stack is empty' gets displayed. If the stack is not empty then the topmost item gets removed from the list.



Applications of Stacks

Stacks are often used is in evaluation of arithmetic expression. An arithmetic expression consists of operands and operators. The operands can be constant or variables. The operators used in an arithmetic expression can be +, -, * and /.

While writing an arithmetic expression, the operator is placed between two operands as shown in the examples below.

A + B * C

A * B - C

A + B / C - D

A $ B + C

This way of representing arithmetic expressions is called **infix** notation. While evaluating an infix expression usually the following operator precedence is used:

- Highest priority: Exponentiation ($)
- Next highest priority: Multiplication (*) and Division (/)
- Lowest priority: Addition (+) and Subtraction (-)

If we wish to override these priorities we can do so by using a pair of parentheses as shown below.

(A + B) * C

A * (B - C)

(A + B) / (C - D)

The expressions within a pair of parentheses are always evaluated earlier than other operations.

To make evaluation of an arithmetic expression easy, a polish mathematician Jan Lukasiewicz suggested a notation called **Polish** notation. As per this notation, an expression in **infix** form can be converted to either **prefix** or **postfix** form and then evaluated. In prefix notation the operator comes before the operands. In postfix notation, the operator follows the two operands. These forms are shown below.

A + B - Infix form

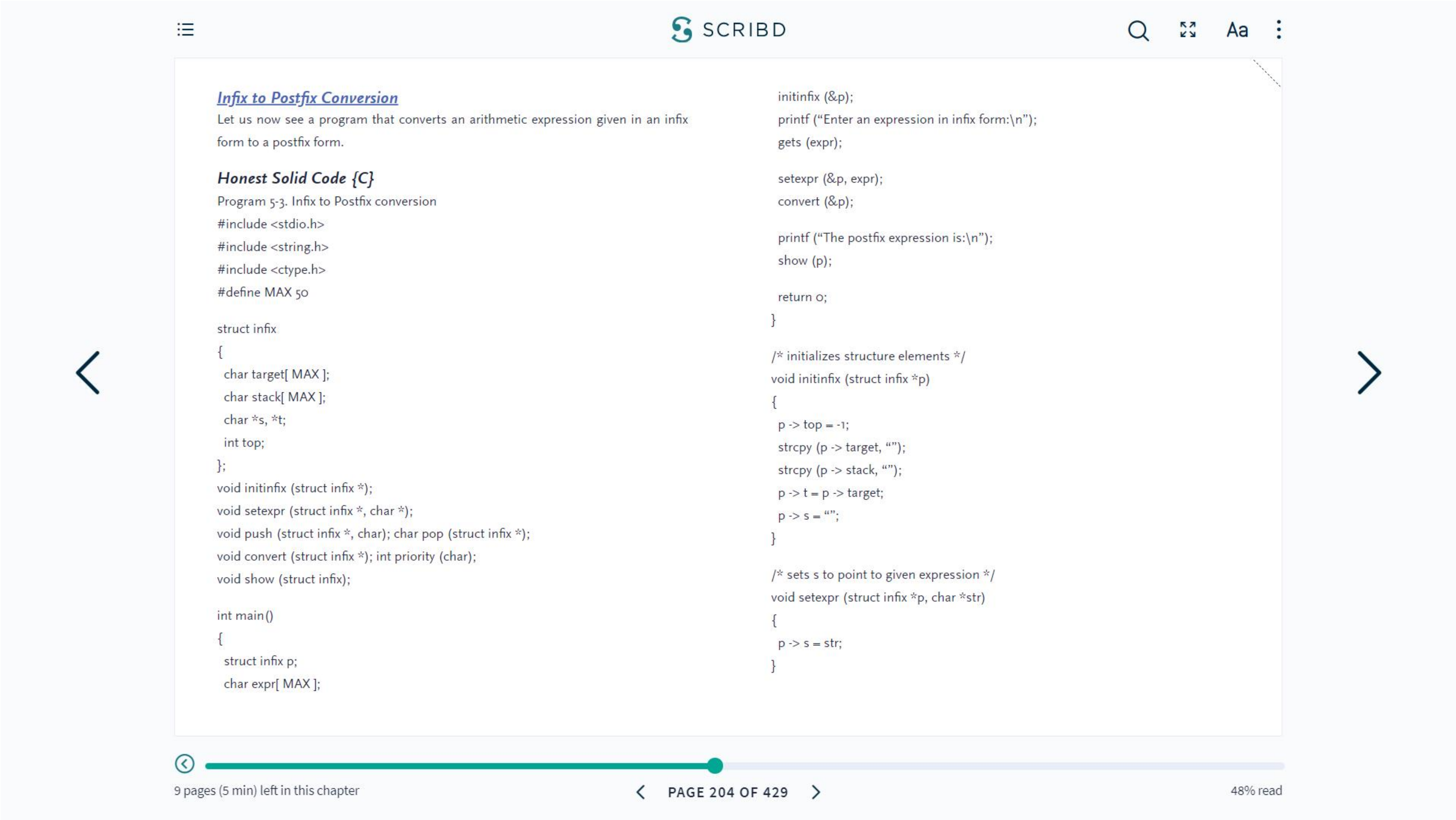
+ A B - Prefix form

A B + - Postfix form

The prefix and postfix expressions have three features:

- The operands maintain the same order as in the equivalent infix expression
- Parentheses are not needed to designate the expression unambiguously.
- While evaluating the expression the priority of the operators is irrelevant.

The stack data structure is used while carrying out the conversion of an expression given in one form to another.



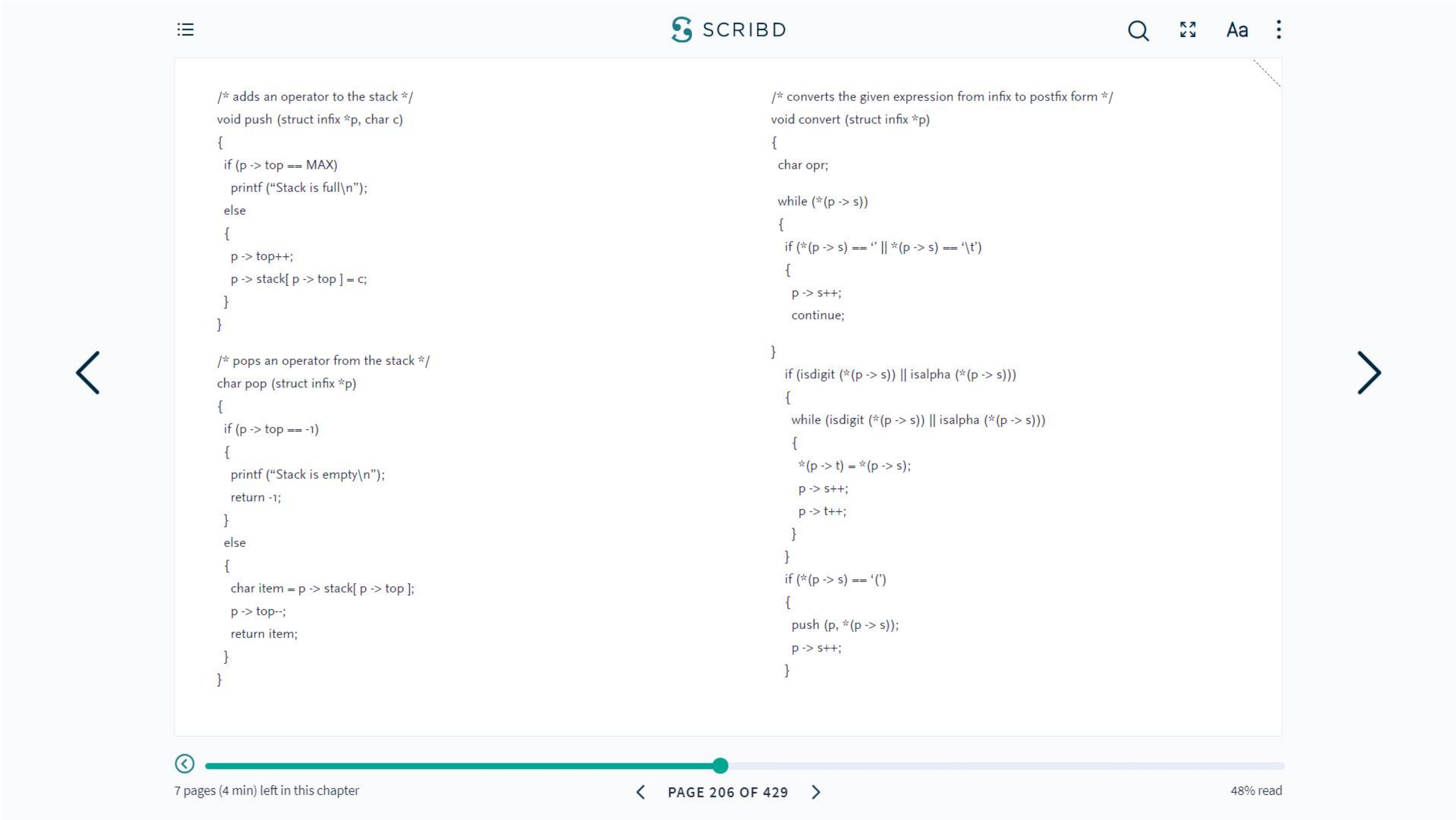
Infix to Postfix Conversion

Let us now see a program that converts an arithmetic expression given in an infix form to a postfix form.

Honest Solid Code {C}

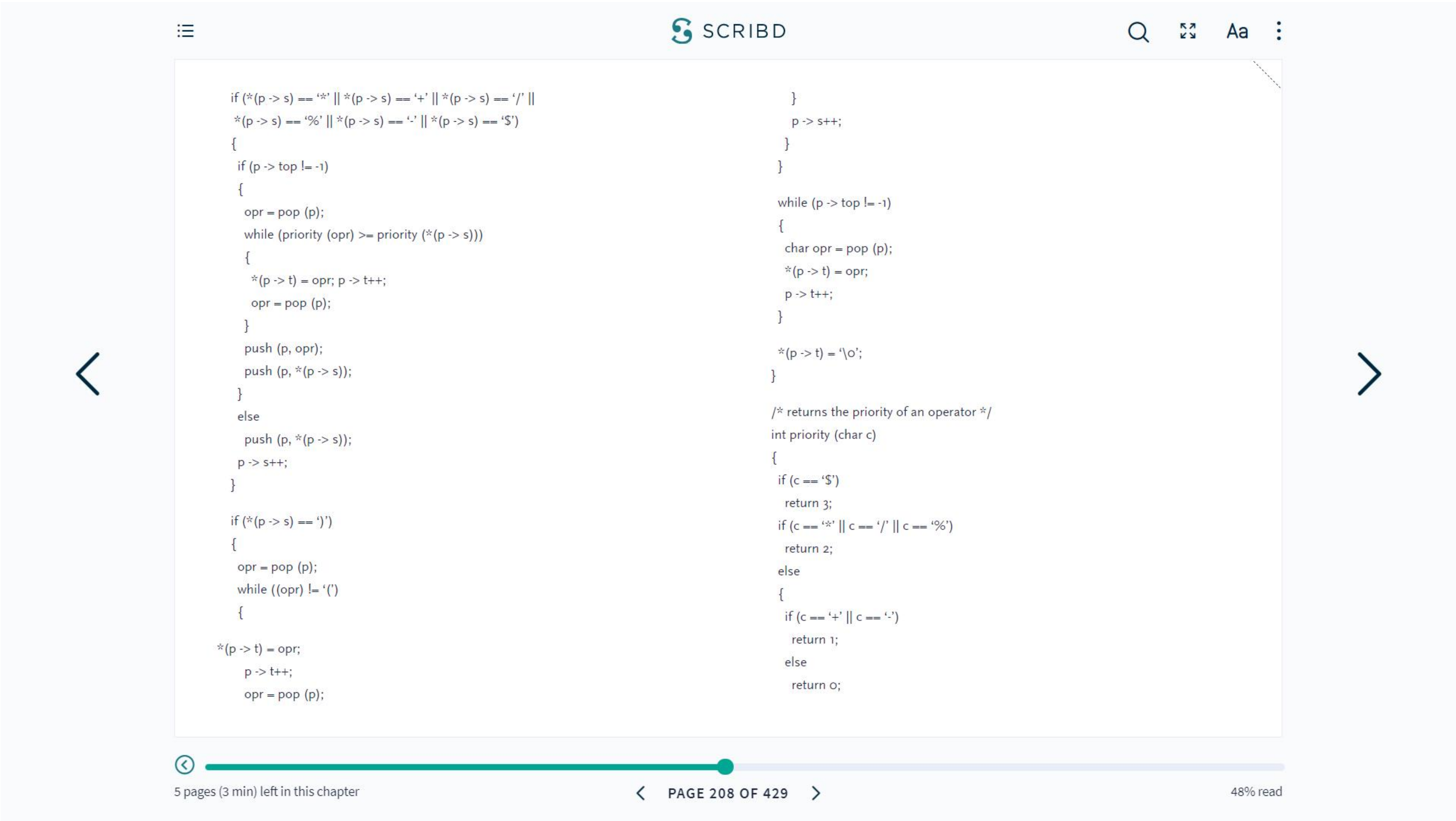
Program 5-3. Infix to Postfix conversion

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define MAX 50

struct infix
{
 char target[ MAX ];
 char stack[ MAX ];
 char *s, *t;
 int top;
};
void initinfix (struct infix *);
void setexpr (struct infix *, char *);
void push (struct infix *, char); char pop (struct infix *);
void convert (struct infix *); int priority (char);
void show (struct infix);

int main()
{
 struct infix p;
 char expr[ MAX ];
 initinfix (&p);
 printf (“Enter an expression in infix form:\n”);
 gets (expr);

 setexpr (&p, expr);
 convert (&p);

 printf (“The postfix expression is:\n”);
 show (p);

 return 0;
}

/* initializes structure elements */
void initinfix (struct infix *p)
{
 p -> top = -1;
 strcpy (p -> target, “”);
 strcpy (p -> stack, “”);
 p -> t = p -> target;
 p -> s = “”;
}

/* sets s to point to given expression */
void setexpr (struct infix *p, char *str)
{
 p -> s = str;
}
```



```
/* adds an operator to the stack */
void push (struct infix *p, char c)
{
 if (p -> top == MAX)
   printf (“Stack is full\n”);
 else
 {
   p -> top++;
   p -> stack[ p -> top ] = c;
 }
}

/* pops an operator from the stack */
char pop (struct infix *p)
{
 if (p -> top == -1)
 {
   printf (“Stack is empty\n”);
   return -1;
 }
 else
 {
   char item = p -> stack[ p -> top ];
   p -> top--;
   return item;
 }
}
```

```
/* converts the given expression from infix to postfix form */
void convert (struct infix *p)
{
 char opr;

 while (*(p -> s))
 {
   if (*(p -> s) == ‘’ || *(p -> s) == ‘\t’)
   {
     p -> s++;
     continue;

}
   if (isdigit (*(p -> s)) || isalpha (*(p -> s)))
   {
     while (isdigit (*(p -> s)) || isalpha (*(p -> s)))
     {
       *(p -> t) = *(p -> s);
       p -> s++;
       p -> t++;
     }
   }
   if (*(p -> s) == ‘(’)
   {
     push (p, *(p -> s));
     p -> s++;
   }
```



```
if (*(p -> s) == ‘*’ || *(p -> s) == ‘+’ || *(p -> s) == ‘/’ ||
 *(p -> s) == ‘%’ || *(p -> s) == ‘-’ || *(p -> s) == ‘$’)
{
 if (p -> top != -1)
 {
  opr = pop (p);
  while (priority (opr) >= priority (*(p -> s)))
  {
   *(p -> t) = opr; p -> t++;
   opr = pop (p);
  }
  push (p, opr);
  push (p, *(p -> s));
 }
 else
  push (p, *(p -> s));
 p -> s++;
}

if (*(p -> s) == ‘)’)
{
 opr = pop (p);
 while ((opr) != ‘(’)
 {
*(p -> t) = opr;
  p -> t++;
  opr = pop (p);
```

```
   }
   p -> s++;
  }
 }

 while (p -> top != -1)
 {
  char opr = pop (p);
  *(p -> t) = opr;
  p -> t++;
 }

 *(p -> t) = ‘\0’;
}

/* returns the priority of an operator */
int priority (char c)
{
 if (c == ‘$’)
  return 3;
 if (c == ‘*’ || c == ‘/’ || c == ‘%’)
  return 2;
 else
 {
  if (c == ‘+’ || c == ‘-’)
   return 1;
  else
   return 0;
```

```
  }
}

/* displays the postfix form of given expr. */
void show (struct infix p)
{
  printf ("%s", p.target);
}
```

Output:

```
Enter an expression in infix form:
4 $ 2 * 3 - 3 + 8 / 4 / (1 + 1)
Stack is empty
The postfix expression is:
42$3*3-84/11+/+
```

This program contains a structure called **infix**. The elements **target** and **stack** are used to store the postfix string and to maintain the stack respectively. The **char** pointers **s** and **t** are used to store intermediate results while converting an infix expression to a postfix form. The variable **top** points to the top of the stack.

During program execution when user enters an arithmetic expression the function **setexpr()** assigns the base address of the string to **char** pointer **s**.

Next, the function **convert()** gets called. This function converts the given infix expression to postfix expression. This function scans every character of the string in a **while** loop and performs one of the following operation depending on the type of character scanned.

(a) If the character scanned happens to be a space then that character is skipped.

(b) If character scanned is a digit or an alphabet, it is added to the target string pointed to by **t**.

(c) If the character scanned is a closing parentheses then it is pushed to the stack by calling **push()** function.

(d) If the character scanned is an operator, then firstly, the topmost element from the stack is retrieved. Through a **while** loop, the priorities of the character scanned (i.e. ***(p -> s)**) and the character popped **opr** are compared. Then following steps are performed:

(i) If **opr** has higher or same priority as the character scanned, then **opr** is added to the target string.

(ii) If **opr** has lower precedence than the character scanned, then the loop is terminated. **opr** is pushed back to the stack. Then, the character scanned (***(p -> s)**) is also pushed to the stack.

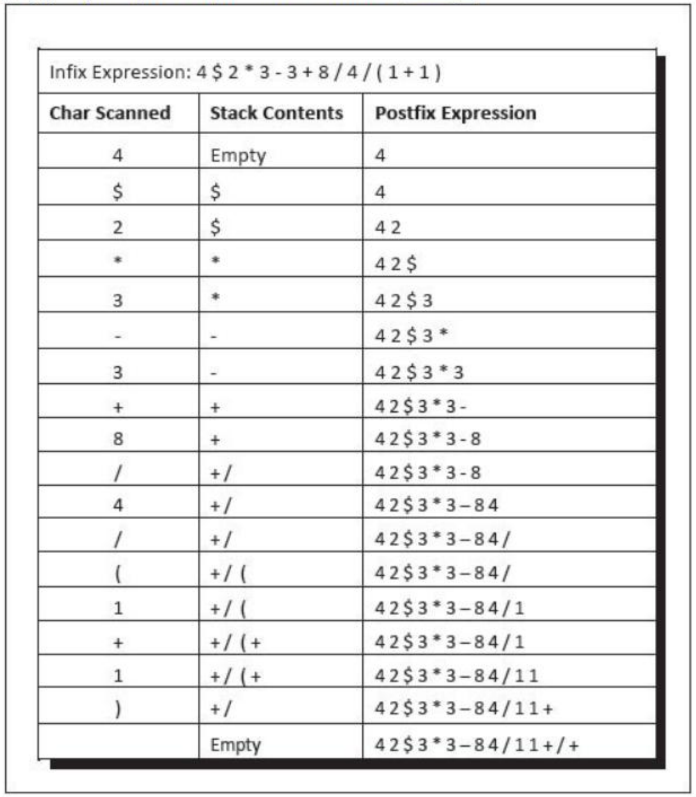
(e) If the character scanned is an opening parenthesis, then the operators present in the stack are popped through a loop. The loop continues till it does not encounter a closing parenthesis. The popped operators are added to the target string pointed to by **t**.

In the **convert()** function we have called functions **push()**, **pop()**, **priority()**. The **push()** function adds a character to the stack, whereas the **pop()** function removes the topmost item from the stack. The **priority()** function returns the priority of operators used in the infix expression. $ (exponentiation) has the highest precedence, followed by *, / and +, -. The function returns integer 3 for $, 2 for * or /, 1 for + or - and 0 for any other character.

The **while** loop in **convert()** gets terminated if the string **s** is exhausted. By then some operators may still be in the stack. These operators should get added to the postfix string. This is done once the control reaches outside the **while** loop in the **convert()** function. Lastly, the converted expression is displayed using the **show()** function.

The steps performed in the conversion of a sample infix expression 4 $ 2 * 3 - 3 + 8

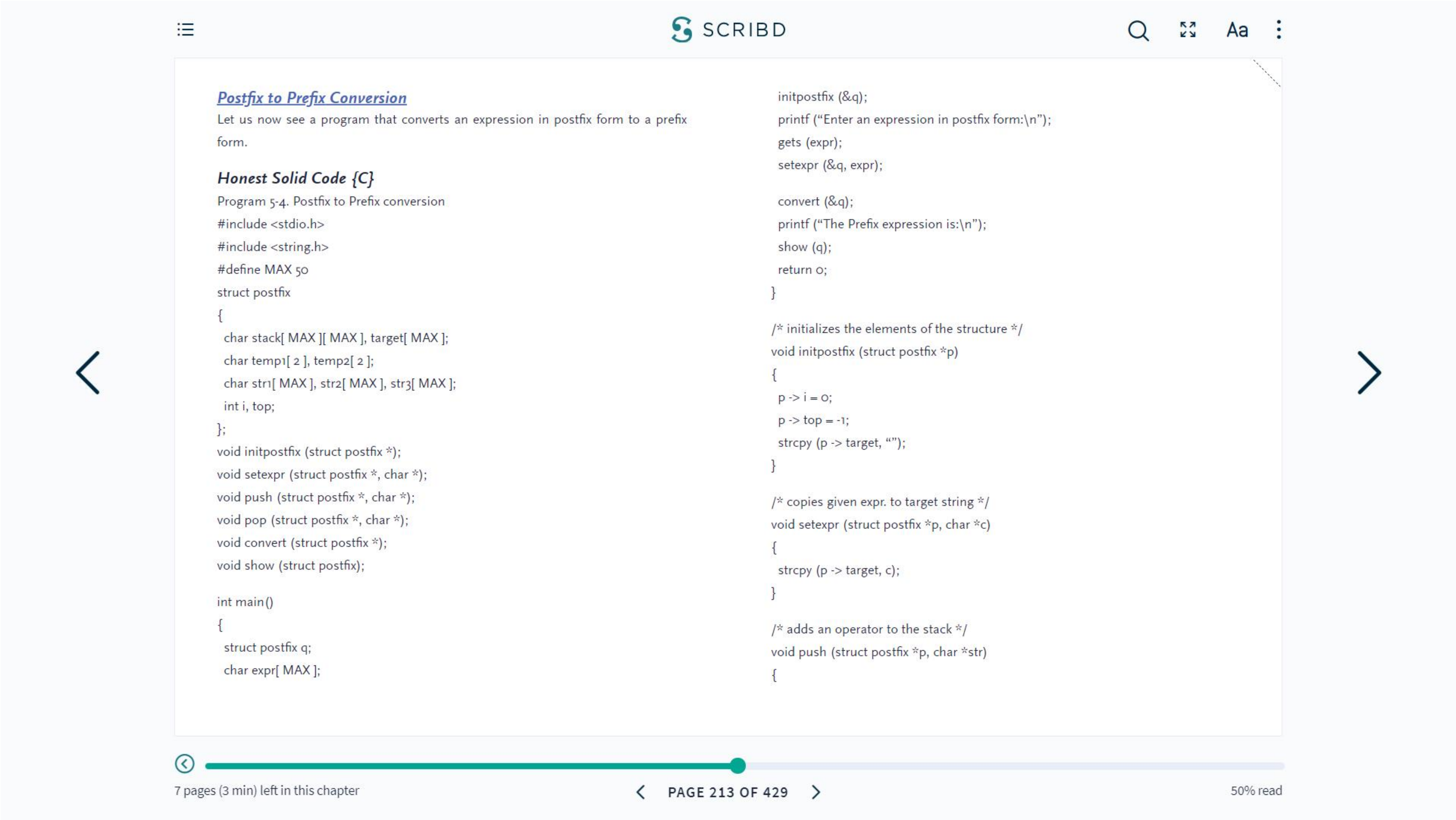
/ 4 / (1 + 1) to a postfix expression are shown in Table 5-1.



Infix Expression: 4 $ 2 * 3 - 3 + 8 / 4 / (1 + 1)

Char Scanned	Stack Contents	Postfix Expression
4	Empty	4
$	$	4
2	$	4 2
*	*	4 2 $
3	*	4 2 $ 3
-	-	4 2 $ 3 *
3	-	4 2 $ 3 * 3
+	+	4 2 $ 3 * 3 -
8	+	4 2 $ 3 * 3 - 8
/	+ /	4 2 $ 3 * 3 - 8
4	+ /	4 2 $ 3 * 3 – 8 4
/	+ /	4 2 $ 3 * 3 – 8 4 /
(	+ / (	4 2 $ 3 * 3 – 8 4 /
1	+ / (	4 2 $ 3 * 3 – 8 4 / 1
+	+ / (+	4 2 $ 3 * 3 – 8 4 / 1
1	+ / (+	4 2 $ 3 * 3 – 8 4 / 1 1
)	+ /	4 2 $ 3 * 3 – 8 4 / 1 1 +
	Empty	4 2 $ 3 * 3 – 8 4 / 1 1 + / +

Table 5-1. *Conversion of Infix to Postfix form.*



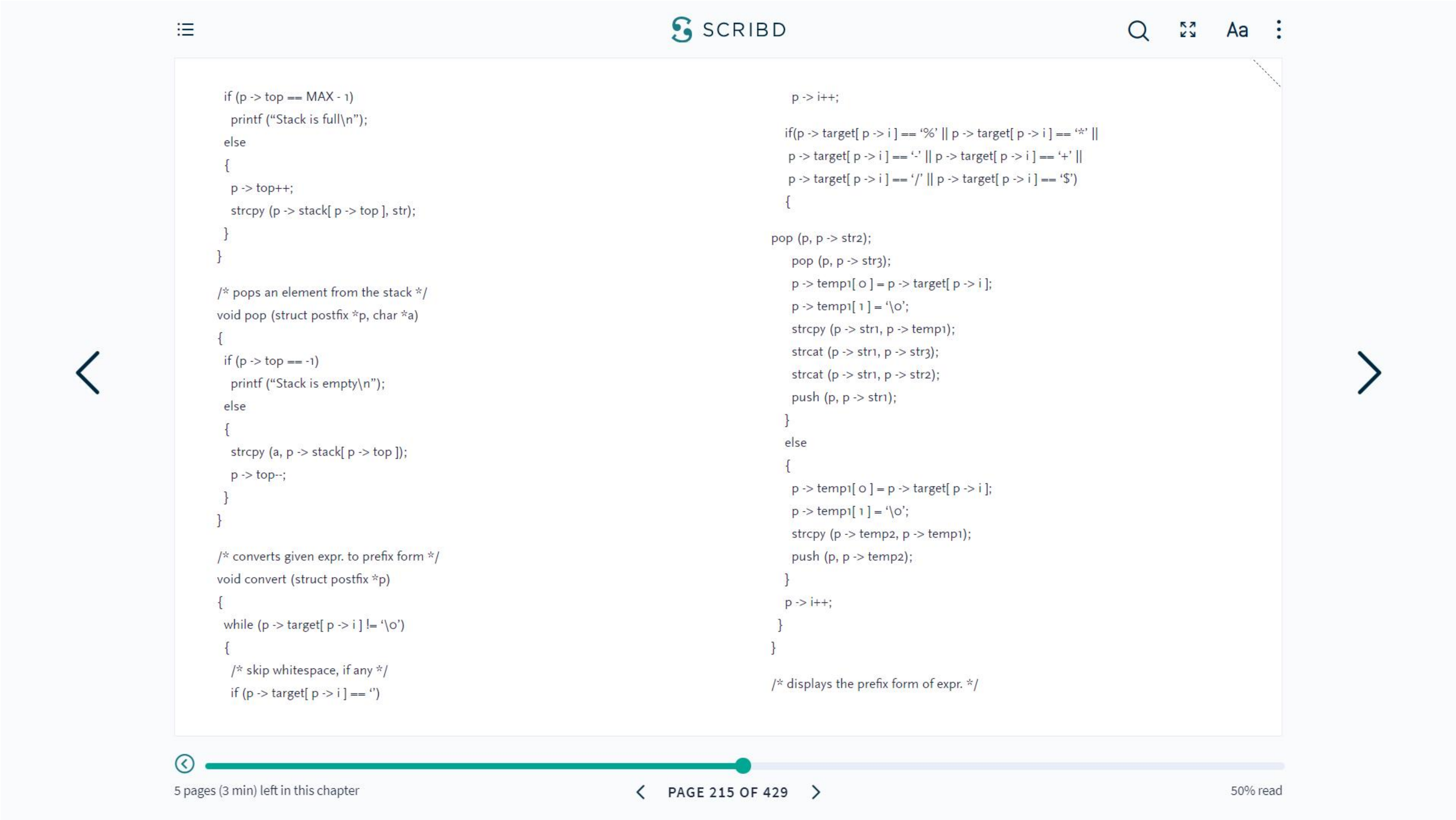
Postfix to Prefix Conversion

Let us now see a program that converts an expression in postfix form to a prefix form.

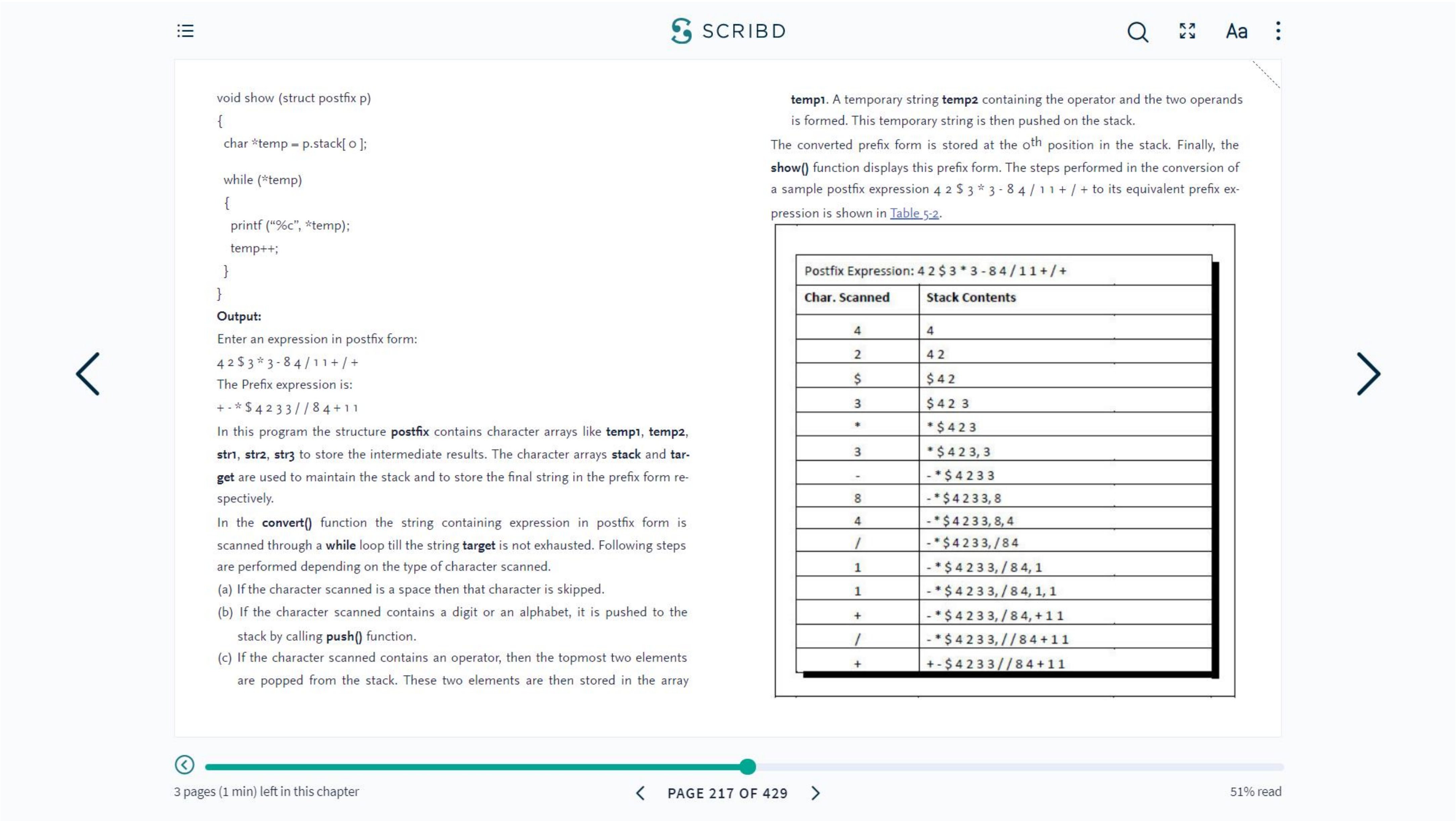
Honest Solid Code {C}

Program 5-4. Postfix to Prefix conversion

```
#include <stdio.h>
#include <string.h>
#define MAX 50
struct postfix
{
  char stack[ MAX ][ MAX ], target[ MAX ];
  char temp1[ 2 ], temp2[ 2 ];
  char str1[ MAX ], str2[ MAX ], str3[ MAX ];
  int i, top;
};
void initpostfix (struct postfix *);
void setexpr (struct postfix *, char *);
void push (struct postfix *, char *);
void pop (struct postfix *, char *);
void convert (struct postfix *);
void show (struct postfix);

int main()
{
  struct postfix q;
  char expr[ MAX ];
  initpostfix (&q);
  printf (“Enter an expression in postfix form:\n”);
  gets (expr);
  setexpr (&q, expr);

  convert (&q);
  printf (“The Prefix expression is:\n”);
  show (q);
  return 0;
}

/* initializes the elements of the structure */
void initpostfix (struct postfix *p)
{
  p -> i = 0;
  p -> top = -1;
  strcpy (p -> target, “”);
}

/* copies given expr. to target string */
void setexpr (struct postfix *p, char *c)
{
  strcpy (p -> target, c);
}

/* adds an operator to the stack */
void push (struct postfix *p, char *str)
{
```



```
  if (p -> top == MAX - 1)
    printf (“Stack is full\n”);
  else
  {
    p -> top++;
    strcpy (p -> stack[ p -> top ], str);
  }
}

/* pops an element from the stack */
void pop (struct postfix *p, char *a)
{
  if (p -> top == -1)
    printf (“Stack is empty\n”);
  else
  {
    strcpy (a, p -> stack[ p -> top ]);
    p -> top--;
  }
}

/* converts given expr. to prefix form */
void convert (struct postfix *p)
{
  while (p -> target[ p -> i ] != ‘\0’)
  {
    /* skip whitespace, if any */
    if (p -> target[ p -> i ] == ‘’)
        p -> i++;

      if(p -> target[ p -> i ] == ‘%’ || p -> target[ p -> i ] == ‘*’ ||
       p -> target[ p -> i ] == ‘-’ || p -> target[ p -> i ] == ‘+’ ||
       p -> target[ p -> i ] == ‘/’ || p -> target[ p -> i ] == ‘$’)
      {
pop (p, p -> str2);
        pop (p, p -> str3);
        p -> temp1[ 0 ] = p -> target[ p -> i ];
        p -> temp1[ 1 ] = ‘\0’;
        strcpy (p -> str1, p -> temp1);
        strcat (p -> str1, p -> str3);
        strcat (p -> str1, p -> str2);
        push (p, p -> str1);
      }
      else
      {
        p -> temp1[ 0 ] = p -> target[ p -> i ];
        p -> temp1[ 1 ] = ‘\0’;
        strcpy (p -> temp2, p -> temp1);
        push (p, p -> temp2);
      }
      p -> i++;
    }
}

/* displays the prefix form of expr. */
```



```
void show (struct postfix p)
{
 char *temp = p.stack[ 0 ];

 while (*temp)
 {
  printf ("%c", *temp);
  temp++;
 }
}
```

Output:

```
Enter an expression in postfix form:
4 2 $ 3 * 3 - 8 4 / 1 1 + / +
The Prefix expression is:
+ - * $ 4 2 3 3 / / 8 4 + 1 1
```

In this program the structure **postfix** contains character arrays like **temp1**, **temp2**, **str1**, **str2**, **str3** to store the intermediate results. The character arrays **stack** and **target** are used to maintain the stack and to store the final string in the prefix form respectively.

In the **convert()** function the string containing expression in postfix form is scanned through a **while** loop till the string **target** is not exhausted. Following steps are performed depending on the type of character scanned.

(a) If the character scanned is a space then that character is skipped.

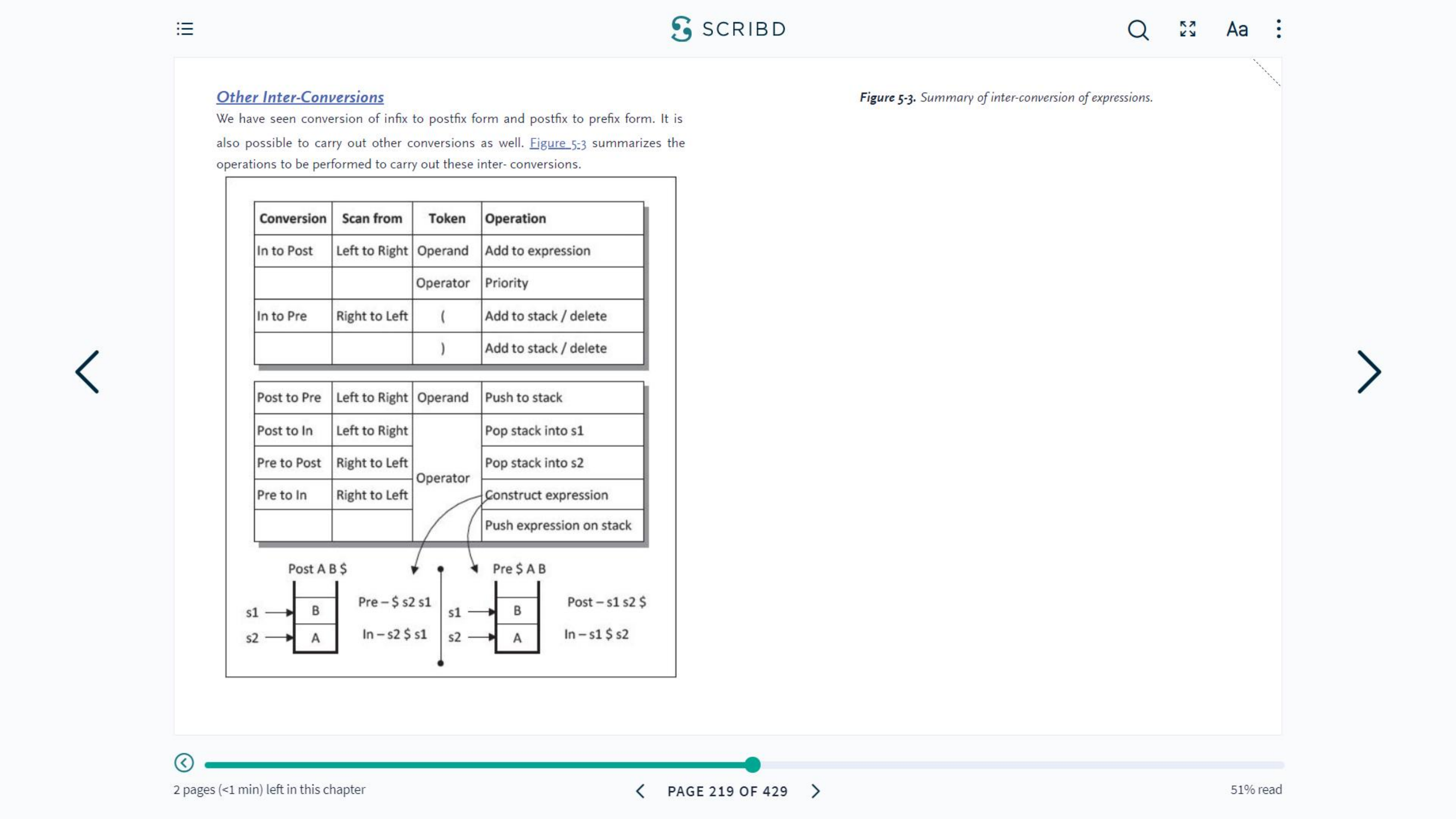
(b) If the character scanned contains a digit or an alphabet, it is pushed to the stack by calling **push()** function.

(c) If the character scanned contains an operator, then the topmost two elements are popped from the stack. These two elements are then stored in the array **temp1**. A temporary string **temp2** containing the operator and the two operands is formed. This temporary string is then pushed on the stack.

The converted prefix form is stored at the 0th position in the stack. Finally, the **show()** function displays this prefix form. The steps performed in the conversion of a sample postfix expression 4 2 $ 3 * 3 - 8 4 / 1 1 + / + to its equivalent prefix expression is shown in Table 5-2.

Postfix Expression: 4 2 $ 3 * 3 - 8 4 / 1 1 + / +	
Char. Scanned	**Stack Contents**
4	4
2	4 2
$	$ 4 2
3	$ 4 2 3
*	* $ 4 2 3
3	* $ 4 2 3, 3
-	- * $ 4 2 3 3
8	- * $ 4 2 3 3, 8
4	- * $ 4 2 3 3, 8, 4
/	- * $ 4 2 3 3, / 8 4
1	- * $ 4 2 3 3, / 8 4, 1
1	- * $ 4 2 3 3, / 8 4, 1, 1
+	- * $ 4 2 3 3, / 8 4, + 1 1
/	- * $ 4 2 3 3, / / 8 4 + 1 1
+	+ - $ 4 2 3 3 / / 8 4 + 1 1

Table 5-2. Conversion of Infix to Postfix form.

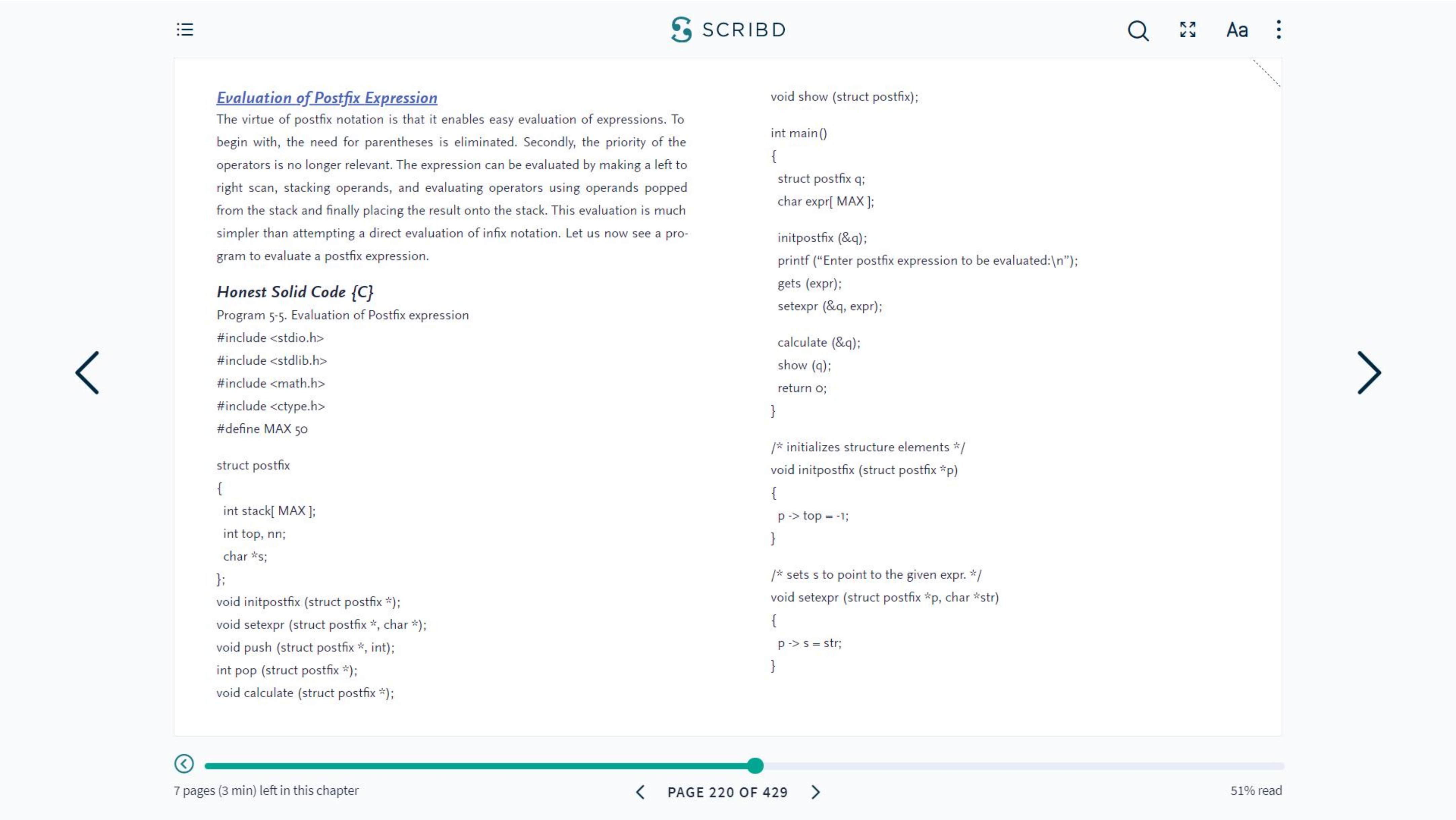


Other Inter-Conversions

We have seen conversion of infix to postfix form and postfix to prefix form. It is also possible to carry out other conversions as well. Figure 5-3 summarizes the operations to be performed to carry out these inter- conversions.

Conversion	Scan from	Token	Operation
In to Post	Left to Right	Operand	Add to expression
		Operator	Priority
In to Pre	Right to Left	(	Add to stack / delete
		)	Add to stack / delete
Post to Pre	Left to Right	Operand	Push to stack
Post to In	Left to Right	Operator	Pop stack into s1
Pre to Post	Right to Left		Pop stack into s2
Pre to In	Right to Left		Construct expression
			Push expression on stack

Figure 5-3. *Summary of inter-conversion of expressions.*



Evaluation of Postfix Expression

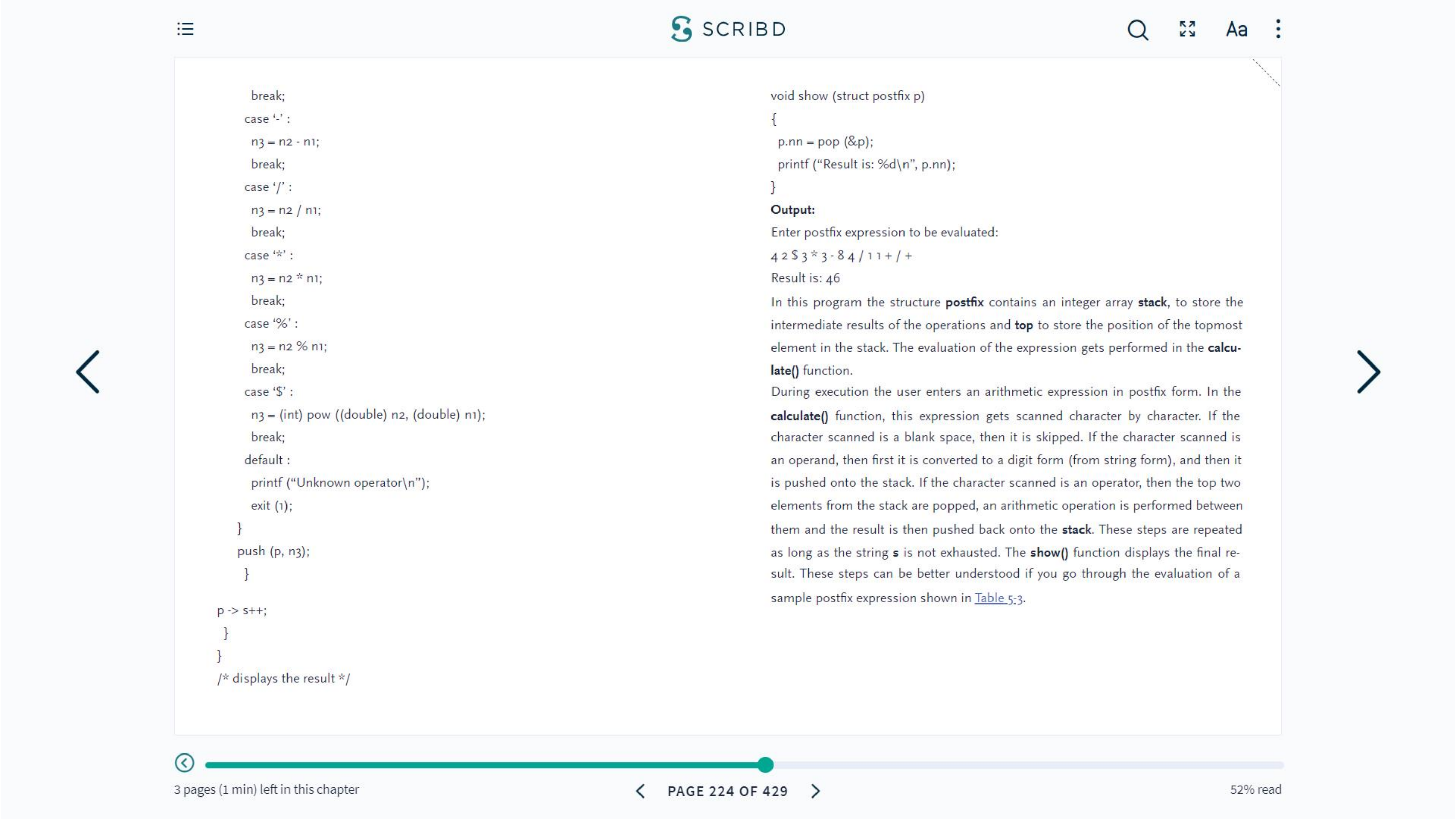
The virtue of postfix notation is that it enables easy evaluation of expressions. To begin with, the need for parentheses is eliminated. Secondly, the priority of the operators is no longer relevant. The expression can be evaluated by making a left to right scan, stacking operands, and evaluating operators using operands popped from the stack and finally placing the result onto the stack. This evaluation is much simpler than attempting a direct evaluation of infix notation. Let us now see a program to evaluate a postfix expression.

Honest Solid Code {C}

Program 5-5. Evaluation of Postfix expression

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>
#define MAX 50

struct postfix
{
  int stack[ MAX ];
  int top, nn;
  char *s;
};
void initpostfix (struct postfix *);
void setexpr (struct postfix *, char *);
void push (struct postfix *, int);
int pop (struct postfix *);
void calculate (struct postfix *);
void show (struct postfix);

int main()
{
  struct postfix q;
  char expr[ MAX ];

  initpostfix (&q);
  printf ("Enter postfix expression to be evaluated:\n");
  gets (expr);
  setexpr (&q, expr);

  calculate (&q);
  show (q);
  return 0;
}

/* initializes structure elements */
void initpostfix (struct postfix *p)
{
  p -> top = -1;
}

/* sets s to point to the given expr. */
void setexpr (struct postfix *p, char *str)
{
  p -> s = str;
}
```

```
/* adds digit to the stack */
void push (struct postfix *p, int item)
{
  if (p -> top == MAX - 1)
    printf (“Stack is full\n”);
  else
  {
    p -> top++;
    p -> stack[ p -> top ] = item;
  }
}

/* pops digit from the stack */
int pop (struct postfix *p)
{
  int data;
  if (p -> top == -1)
  {
    printf (“Stack is empty\n”);
    return NULL;
  }

  data = p -> stack[ p -> top ];
  p -> top--;
  return data;
}

/* evaluates the postfix expression */
void calculate(struct postfix *p)
{
  int n1, n2, n3;
  while (*(p -> s))
  {
    /* skip whitespace, if any */
    if (*(p -> s) == ‘’ || *(p -> s) == ‘\t’)
    {

p -> s++;
      continue;
    }
    /* if digit is encountered */
    if (isdigit (*(p -> s)))
    {
      p -> nn = *(p -> s) - ‘0’;
      push (p, p -> nn);
    }
    else
    {
      /* if operator is encountered */
      n1 = pop (p);
      n2 = pop (p);
      switch (*(p -> s))
      {
        case ‘+’ :
          n3 = n2 + n1;
```



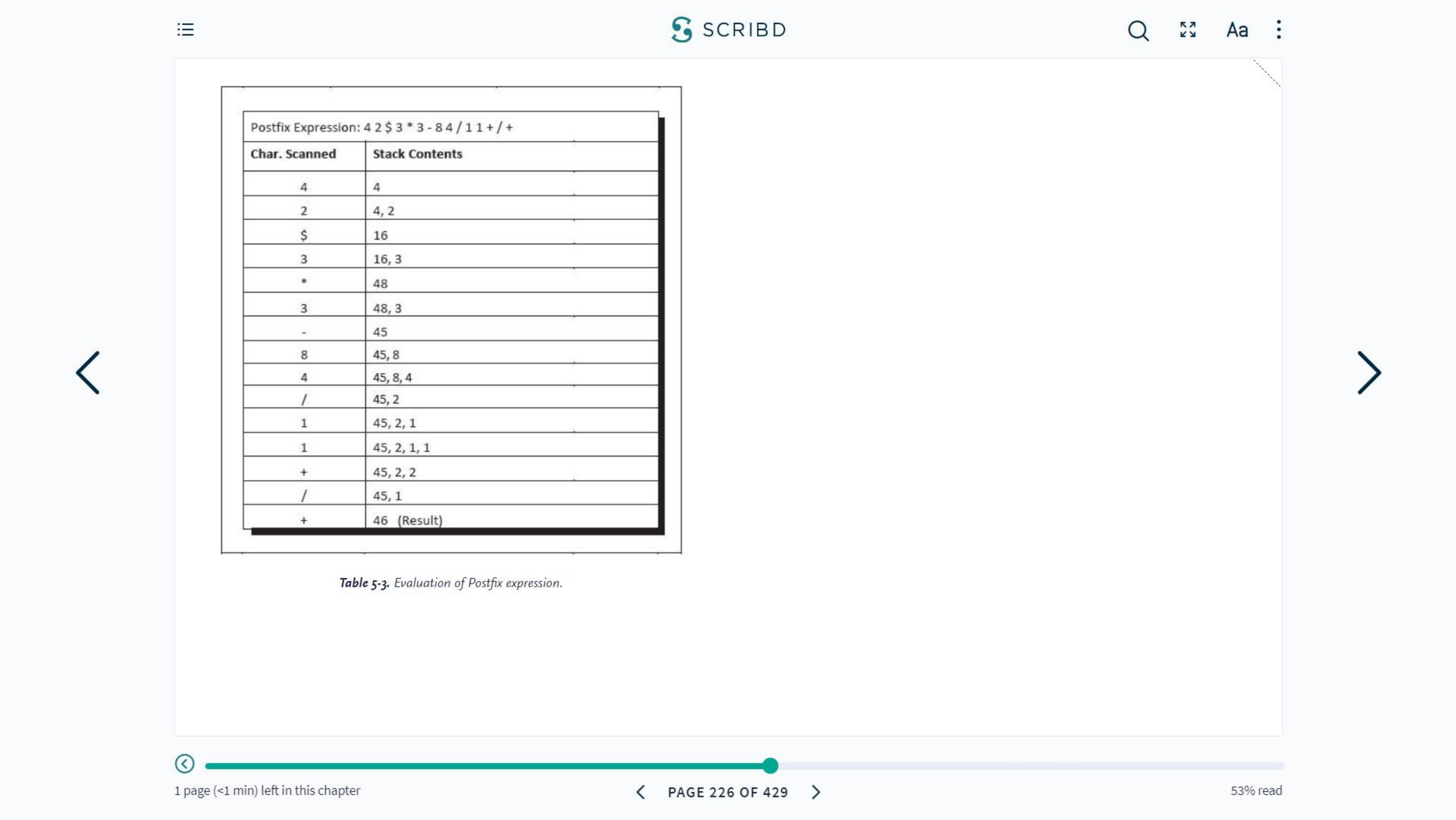
```
          break;
        case '-' :
          n3 = n2 - n1;
          break;
        case '/' :
          n3 = n2 / n1;
          break;
        case '*' :
          n3 = n2 * n1;
          break;
        case '%' :
          n3 = n2 % n1;
          break;
        case '$' :
          n3 = (int) pow ((double) n2, (double) n1);
          break;
        default :
          printf ("Unknown operator\n");
          exit (1);
      }
      push (p, n3);
        }

p -> s++;
 }
}
/* displays the result */
void show (struct postfix p)
{
  p.nn = pop (&p);
  printf ("Result is: %d\n", p.nn);
}
```

Output:

```
Enter postfix expression to be evaluated:
4 2 $ 3 * 3 - 8 4 / 1 1 + / +
Result is: 46
```

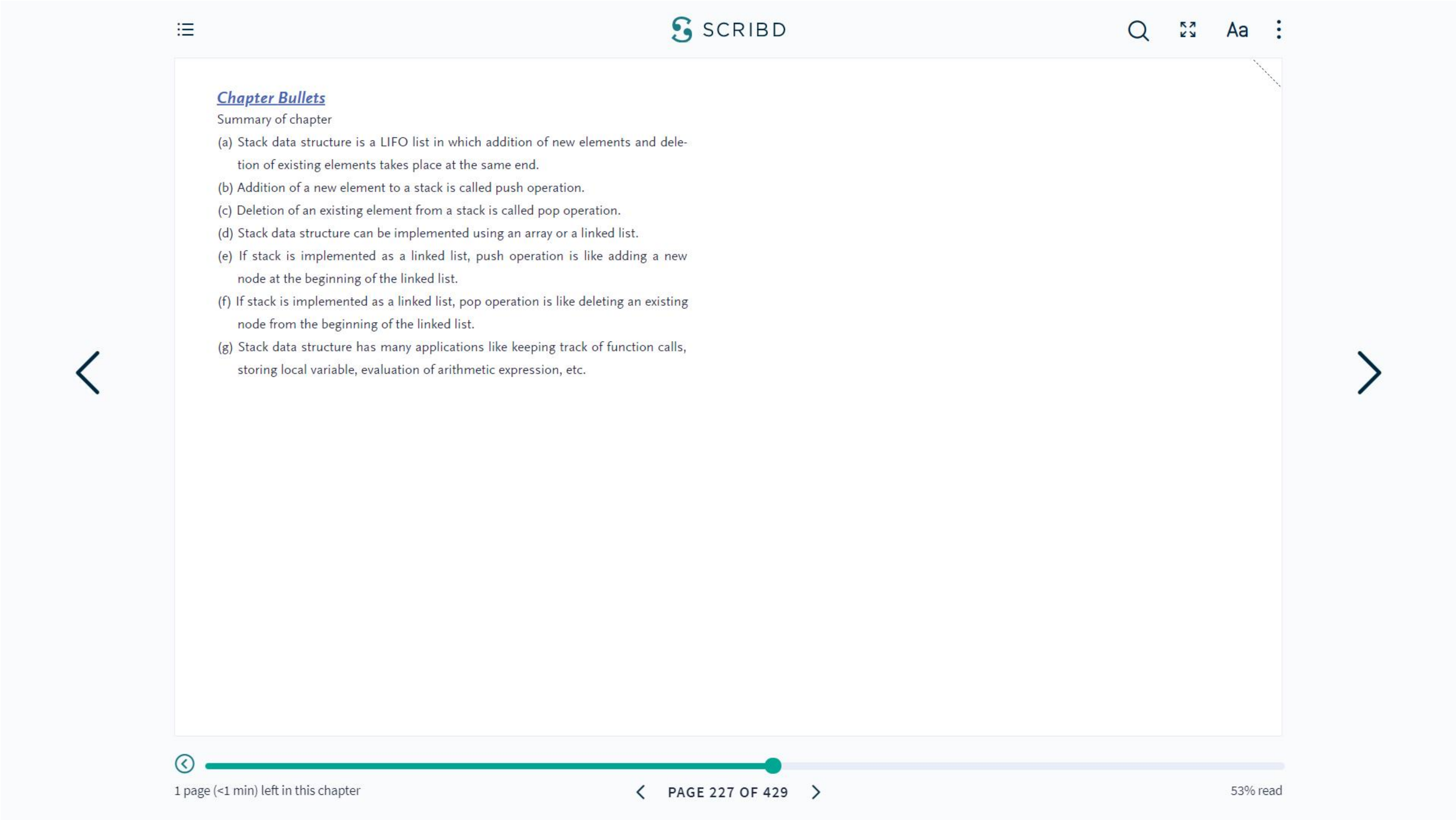
In this program the structure **postfix** contains an integer array **stack**, to store the intermediate results of the operations and **top** to store the position of the topmost element in the stack. The evaluation of the expression gets performed in the **calculate()** function.

During execution the user enters an arithmetic expression in postfix form. In the **calculate()** function, this expression gets scanned character by character. If the character scanned is a blank space, then it is skipped. If the character scanned is an operand, then first it is converted to a digit form (from string form), and then it is pushed onto the stack. If the character scanned is an operator, then the top two elements from the stack are popped, an arithmetic operation is performed between them and the result is then pushed back onto the **stack**. These steps are repeated as long as the string **s** is not exhausted. The **show()** function displays the final result. These steps can be better understood if you go through the evaluation of a sample postfix expression shown in Table 5-3.



Postfix Expression: 4 2 $ 3 * 3 - 8 4 / 1 1 + / +	
Char. Scanned	**Stack Contents**
4	4
2	4, 2
$	16
3	16, 3
*	48
3	48, 3
-	45
8	45, 8
4	45, 8, 4
/	45, 2
1	45, 2, 1
1	45, 2, 1, 1
+	45, 2, 2
/	45, 1
+	46 (Result)

Table 5-3. *Evaluation of Postfix expression.*



Chapter Bullets

Summary of chapter

(a) Stack data structure is a LIFO list in which addition of new elements and deletion of existing elements takes place at the same end.

(b) Addition of a new element to a stack is called push operation.

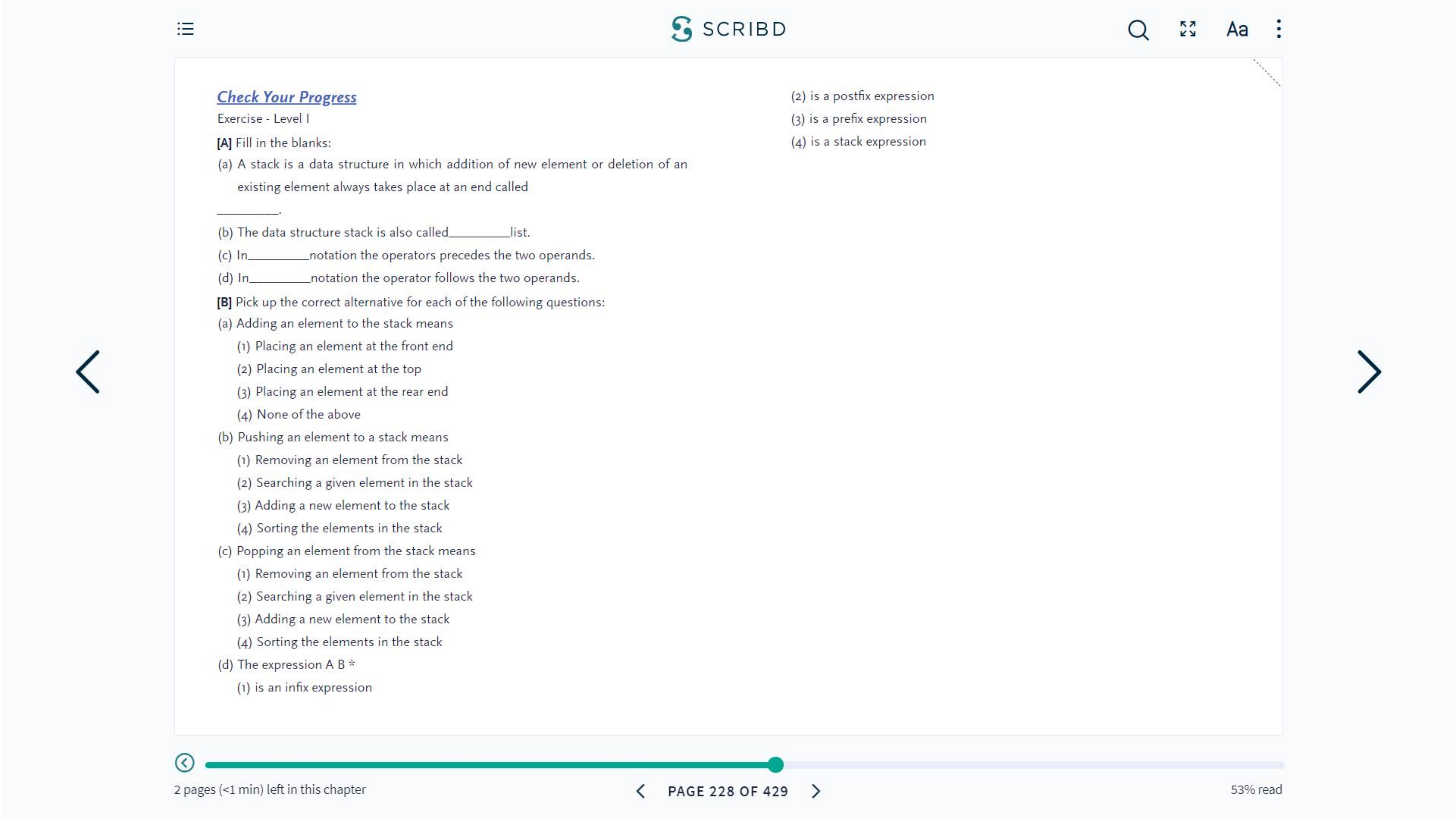
(c) Deletion of an existing element from a stack is called pop operation.

(d) Stack data structure can be implemented using an array or a linked list.

(e) If stack is implemented as a linked list, push operation is like adding a new node at the beginning of the linked list.

(f) If stack is implemented as a linked list, pop operation is like deleting an existing node from the beginning of the linked list.

(g) Stack data structure has many applications like keeping track of function calls, storing local variable, evaluation of arithmetic expression, etc.



Check Your Progress

Exercise - Level I

[A] Fill in the blanks:

(a) A stack is a data structure in which addition of new element or deletion of an existing element always takes place at an end called

__________.

(b) The data structure stack is also called__________list.

(c) In__________notation the operators precedes the two operands.

(d) In__________notation the operator follows the two operands.

[B] Pick up the correct alternative for each of the following questions:

(a) Adding an element to the stack means

(1) Placing an element at the front end

(2) Placing an element at the top

(3) Placing an element at the rear end

(4) None of the above

(b) Pushing an element to a stack means

(1) Removing an element from the stack

(2) Searching a given element in the stack

(3) Adding a new element to the stack

(4) Sorting the elements in the stack

(c) Popping an element from the stack means

(1) Removing an element from the stack

(2) Searching a given element in the stack

(3) Adding a new element to the stack

(4) Sorting the elements in the stack

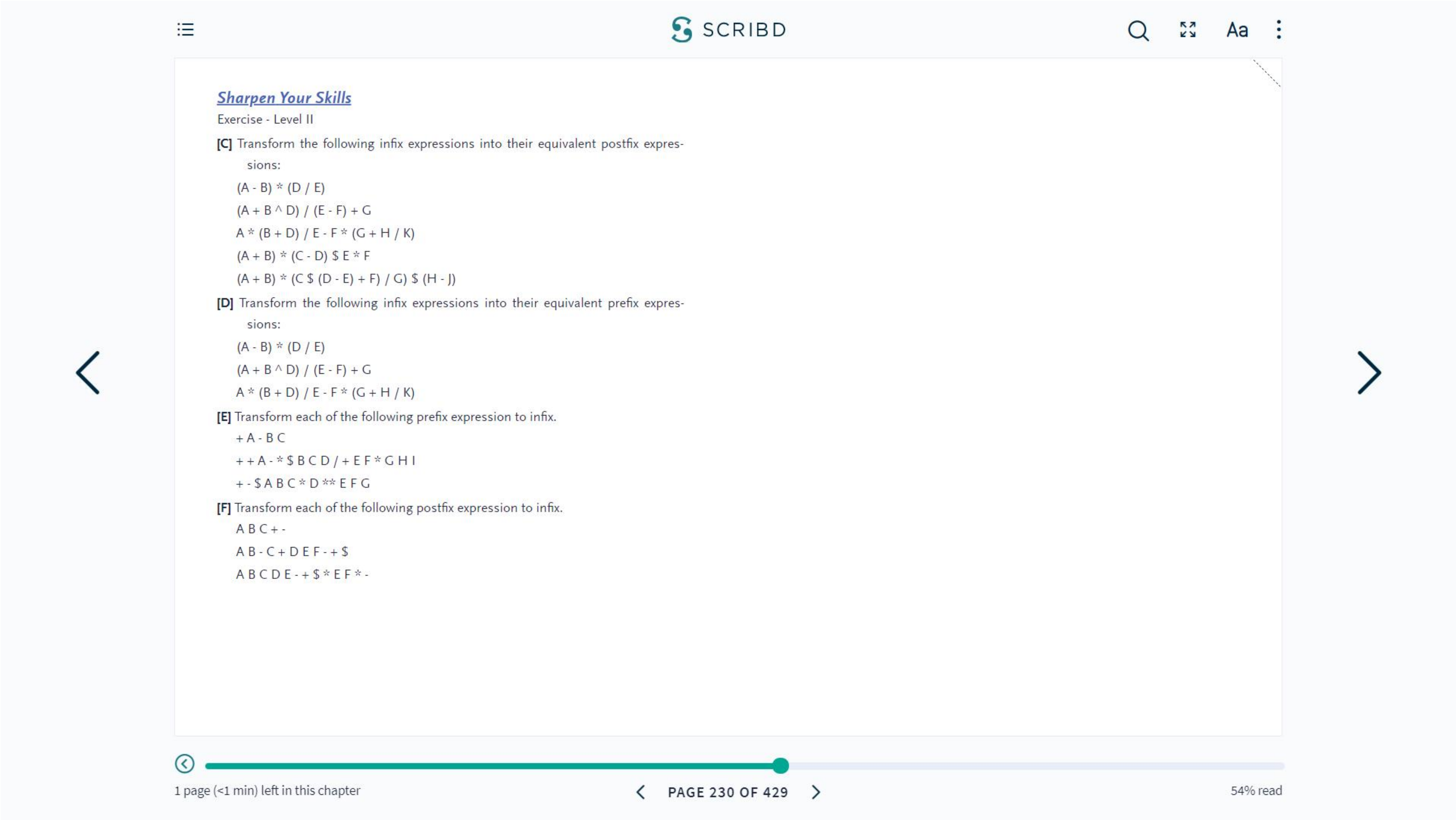
(d) The expression A B *

(1) is an infix expression

(2) is a postfix expression

(3) is a prefix expression

(4) is a stack expression



Sharpen Your Skills

Exercise - Level II

[C] Transform the following infix expressions into their equivalent postfix expressions:

(A - B) * (D / E)

(A + B ^ D) / (E - F) + G

A * (B + D) / E - F * (G + H / K)

(A + B) * (C - D) $ E * F

(A + B) * (C $ (D - E) + F) / G) $ (H - J)

[D] Transform the following infix expressions into their equivalent prefix expressions:

(A - B) * (D / E)

(A + B ^ D) / (E - F) + G

A * (B + D) / E - F * (G + H / K)

[E] Transform each of the following prefix expression to infix.

+ A - B C

+ + A - * $ B C D / + E F * G H I

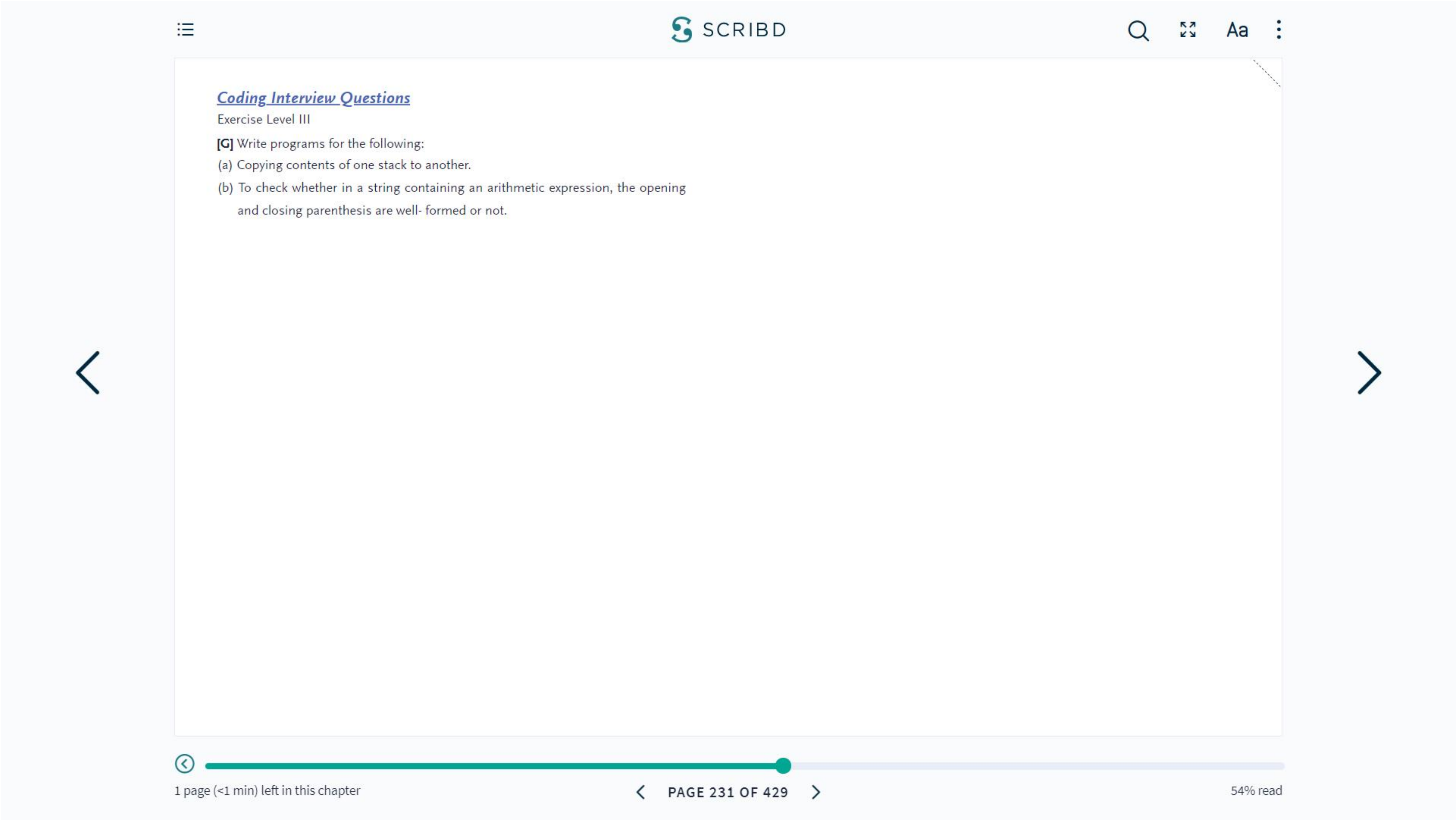
+ - $ A B C * D ** E F G

[F] Transform each of the following postfix expression to infix.

A B C + -

A B - C + D E F - + $

A B C D E - + $ * E F * -



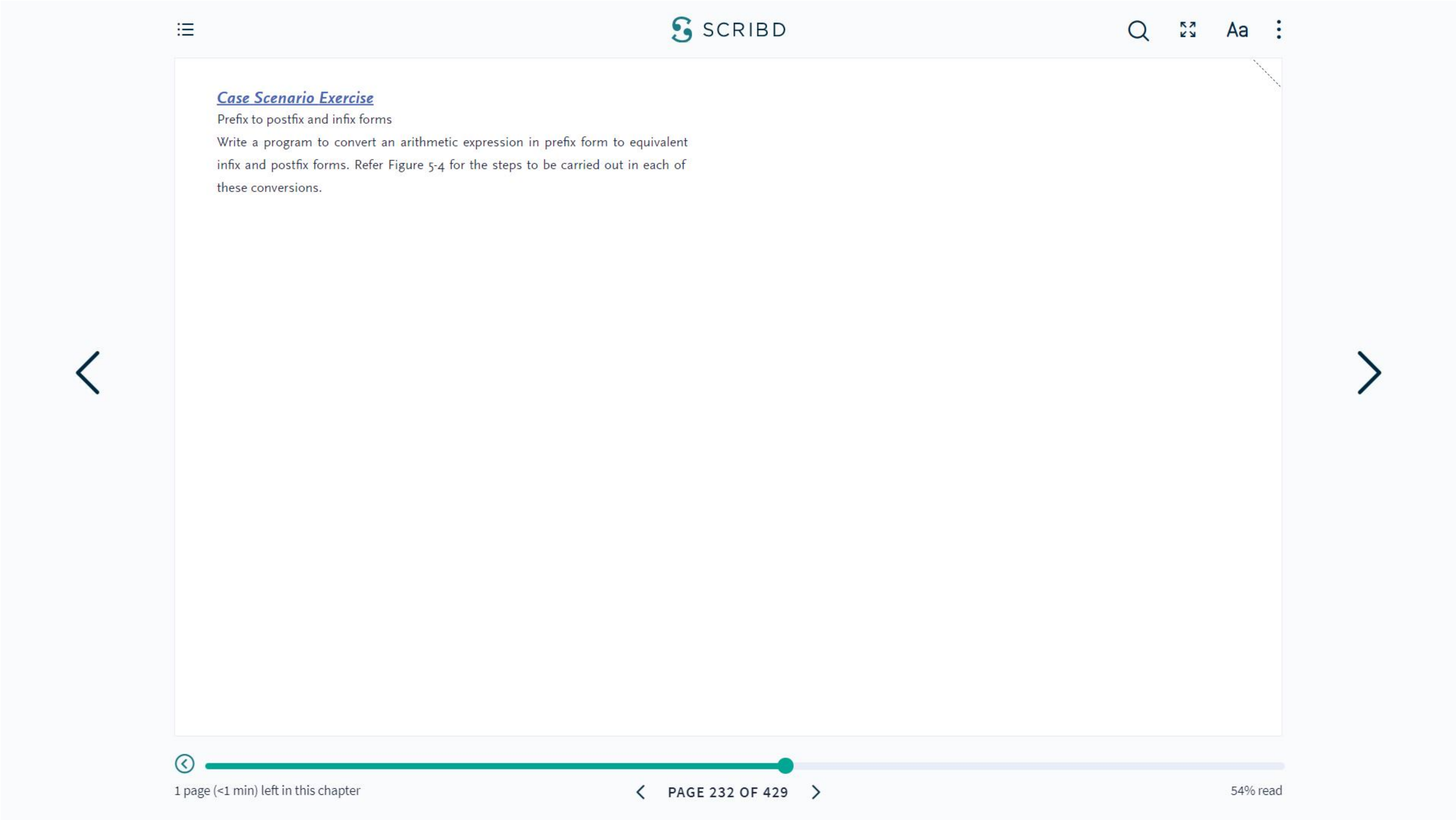
Coding Interview Questions

Exercise Level III

[G] Write programs for the following:

(a) Copying contents of one stack to another.

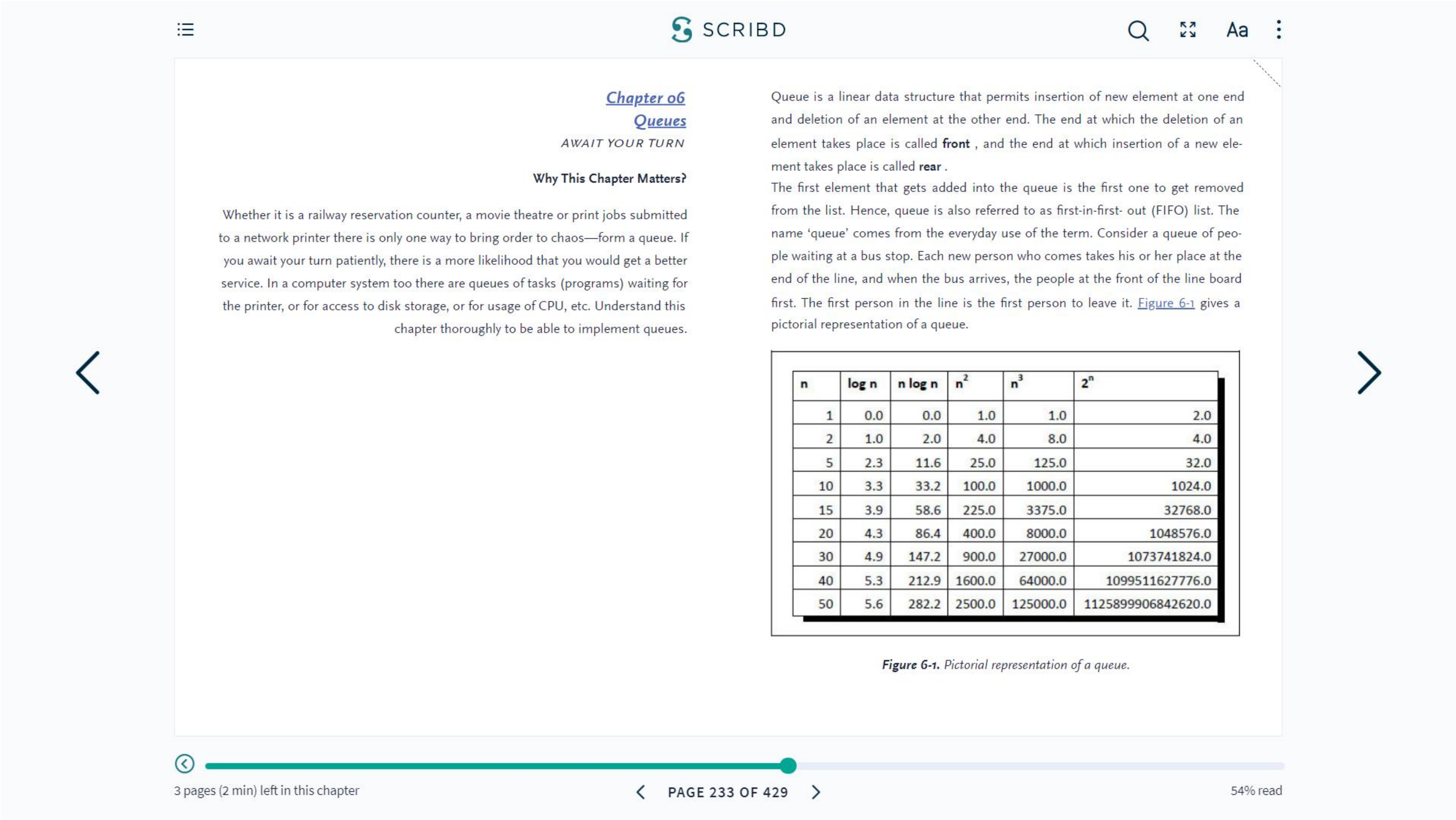
(b) To check whether in a string containing an arithmetic expression, the opening and closing parenthesis are well- formed or not.



Case Scenario Exercise

Prefix to postfix and infix forms

Write a program to convert an arithmetic expression in prefix form to equivalent infix and postfix forms. Refer Figure 5-4 for the steps to be carried out in each of these conversions.



Chapter 06
Queues

AWAIT YOUR TURN

Why This Chapter Matters?

Whether it is a railway reservation counter, a movie theatre or print jobs submitted to a network printer there is only one way to bring order to chaos—form a queue. If you await your turn patiently, there is a more likelihood that you would get a better service. In a computer system too there are queues of tasks (programs) waiting for the printer, or for access to disk storage, or for usage of CPU, etc. Understand this chapter thoroughly to be able to implement queues.

Queue is a linear data structure that permits insertion of new element at one end and deletion of an element at the other end. The end at which the deletion of an element takes place is called **front** , and the end at which insertion of a new element takes place is called **rear** .

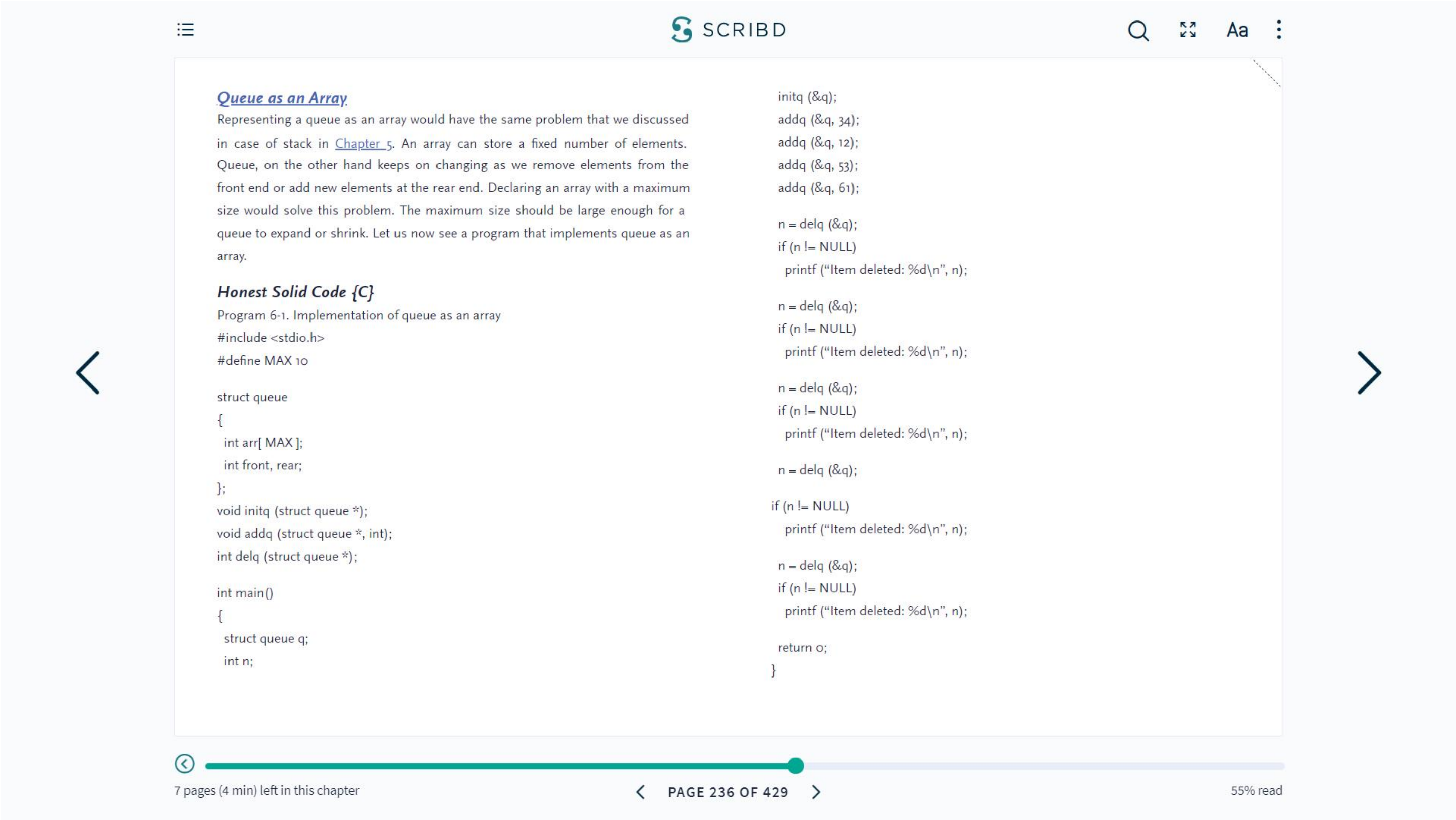
The first element that gets added into the queue is the first one to get removed from the list. Hence, queue is also referred to as first-in-first- out (FIFO) list. The name 'queue' comes from the everyday use of the term. Consider a queue of people waiting at a bus stop. Each new person who comes takes his or her place at the end of the line, and when the bus arrives, the people at the front of the line board first. The first person in the line is the first person to leave it. Figure 6-1 gives a pictorial representation of a queue.

n	log n	n log n	n^2	n^3	2^n
1	0.0	0.0	1.0	1.0	2.0
2	1.0	2.0	4.0	8.0	4.0
5	2.3	11.6	25.0	125.0	32.0
10	3.3	33.2	100.0	1000.0	1024.0
15	3.9	58.6	225.0	3375.0	32768.0
20	4.3	86.4	400.0	8000.0	1048576.0
30	4.9	147.2	900.0	27000.0	1073741824.0
40	5.3	212.9	1600.0	64000.0	1099511627776.0
50	5.6	282.2	2500.0	125000.0	1125899906842620.0

Figure 6-1. *Pictorial representation of a queue.*

In Figure 6-1, 34 is the first element and 42 is the last element added to the queue. Similarly, 34 will be the first element to get removed and 42 will be the last element to get removed from the queue.

Queue, being a linear data structure can be represented using either an array or a linked list. These implementations are discussed in following sections.



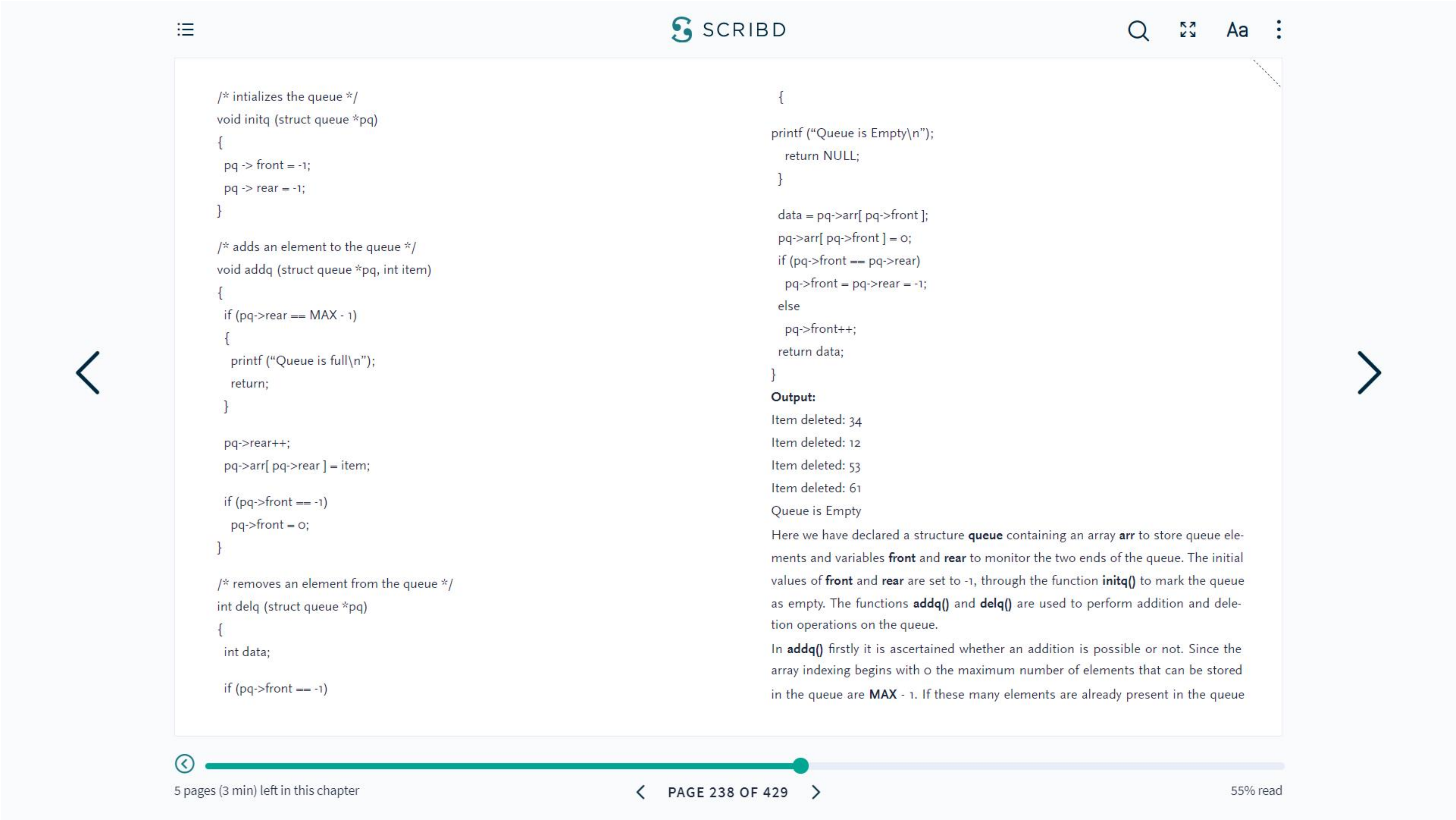
Queue as an Array

Representing a queue as an array would have the same problem that we discussed in case of stack in Chapter 5. An array can store a fixed number of elements. Queue, on the other hand keeps on changing as we remove elements from the front end or add new elements at the rear end. Declaring an array with a maximum size would solve this problem. The maximum size should be large enough for a queue to expand or shrink. Let us now see a program that implements queue as an array.

Honest Solid Code {C}

Program 6-1. Implementation of queue as an array

```
#include <stdio.h>
#define MAX 10

struct queue
{
 int arr[ MAX ];
 int front, rear;
};
void initq (struct queue *);
void addq (struct queue *, int);
int delq (struct queue *);

int main()
{
 struct queue q;
 int n;
 initq (&q);
 addq (&q, 34);
 addq (&q, 12);
 addq (&q, 53);
 addq (&q, 61);

 n = delq (&q);
 if (n != NULL)
  printf (“Item deleted: %d\n”, n);

 n = delq (&q);
 if (n != NULL)
  printf (“Item deleted: %d\n”, n);

 n = delq (&q);
 if (n != NULL)
  printf (“Item deleted: %d\n”, n);

 n = delq (&q);

if (n != NULL)
  printf (“Item deleted: %d\n”, n);

 n = delq (&q);
 if (n != NULL)
  printf (“Item deleted: %d\n”, n);

 return 0;
}
```



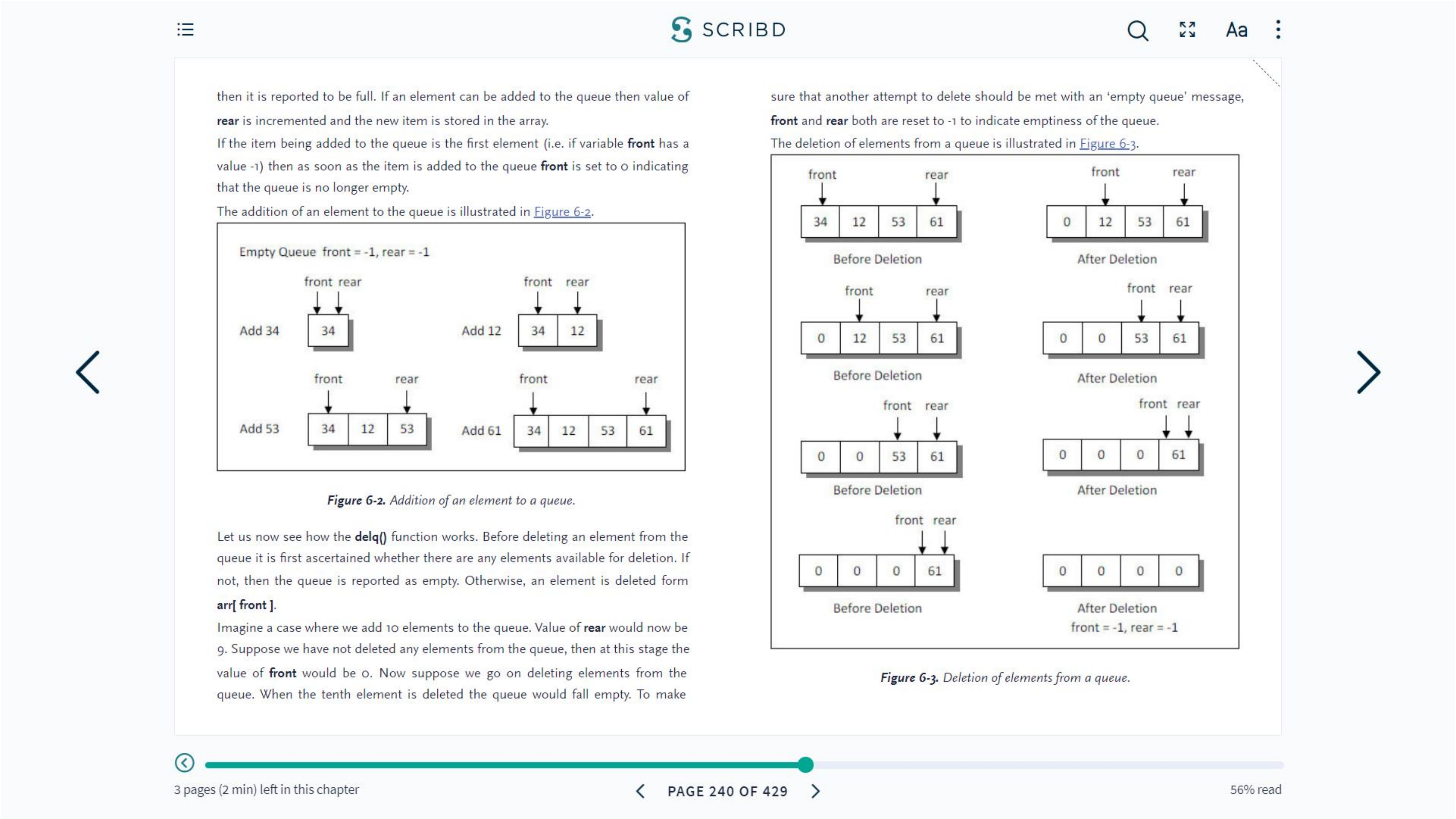
```
/* intializes the queue */
void initq (struct queue *pq)
{
 pq -> front = -1;
 pq -> rear = -1;
}

/* adds an element to the queue */
void addq (struct queue *pq, int item)
{
 if (pq->rear == MAX - 1)
 {
  printf ("Queue is full\n");
  return;
 }

 pq->rear++;
 pq->arr[ pq->rear ] = item;

 if (pq->front == -1)
  pq->front = 0;
}

/* removes an element from the queue */
int delq (struct queue *pq)
{
 int data;

 if (pq->front == -1)
 {

printf ("Queue is Empty\n");
  return NULL;
 }

 data = pq->arr[ pq->front ];
 pq->arr[ pq->front ] = 0;
 if (pq->front == pq->rear)
  pq->front = pq->rear = -1;
 else
  pq->front++;
 return data;
}
```

Output:

```
Item deleted: 34
Item deleted: 12
Item deleted: 53
Item deleted: 61
Queue is Empty
```

Here we have declared a structure **queue** containing an array **arr** to store queue elements and variables **front** and **rear** to monitor the two ends of the queue. The initial values of **front** and **rear** are set to -1, through the function **initq()** to mark the queue as empty. The functions **addq()** and **delq()** are used to perform addition and deletion operations on the queue.

In **addq()** firstly it is ascertained whether an addition is possible or not. Since the array indexing begins with 0 the maximum number of elements that can be stored in the queue are **MAX** - 1. If these many elements are already present in the queue



then it is reported to be full. If an element can be added to the queue then value of **rear** is incremented and the new item is stored in the array.

If the item being added to the queue is the first element (i.e. if variable **front** has a value -1) then as soon as the item is added to the queue **front** is set to 0 indicating that the queue is no longer empty.

The addition of an element to the queue is illustrated in Figure 6-2.

Figure 6-2. Addition of an element to a queue.

Let us now see how the **delq()** function works. Before deleting an element from the queue it is first ascertained whether there are any elements available for deletion. If not, then the queue is reported as empty. Otherwise, an element is deleted form **arr[front]**.

Imagine a case where we add 10 elements to the queue. Value of **rear** would now be 9. Suppose we have not deleted any elements from the queue, then at this stage the value of **front** would be 0. Now suppose we go on deleting elements from the queue. When the tenth element is deleted the queue would fall empty. To make sure that another attempt to delete should be met with an 'empty queue' message, **front** and **rear** both are reset to -1 to indicate emptiness of the queue.

The deletion of elements from a queue is illustrated in Figure 6-3.

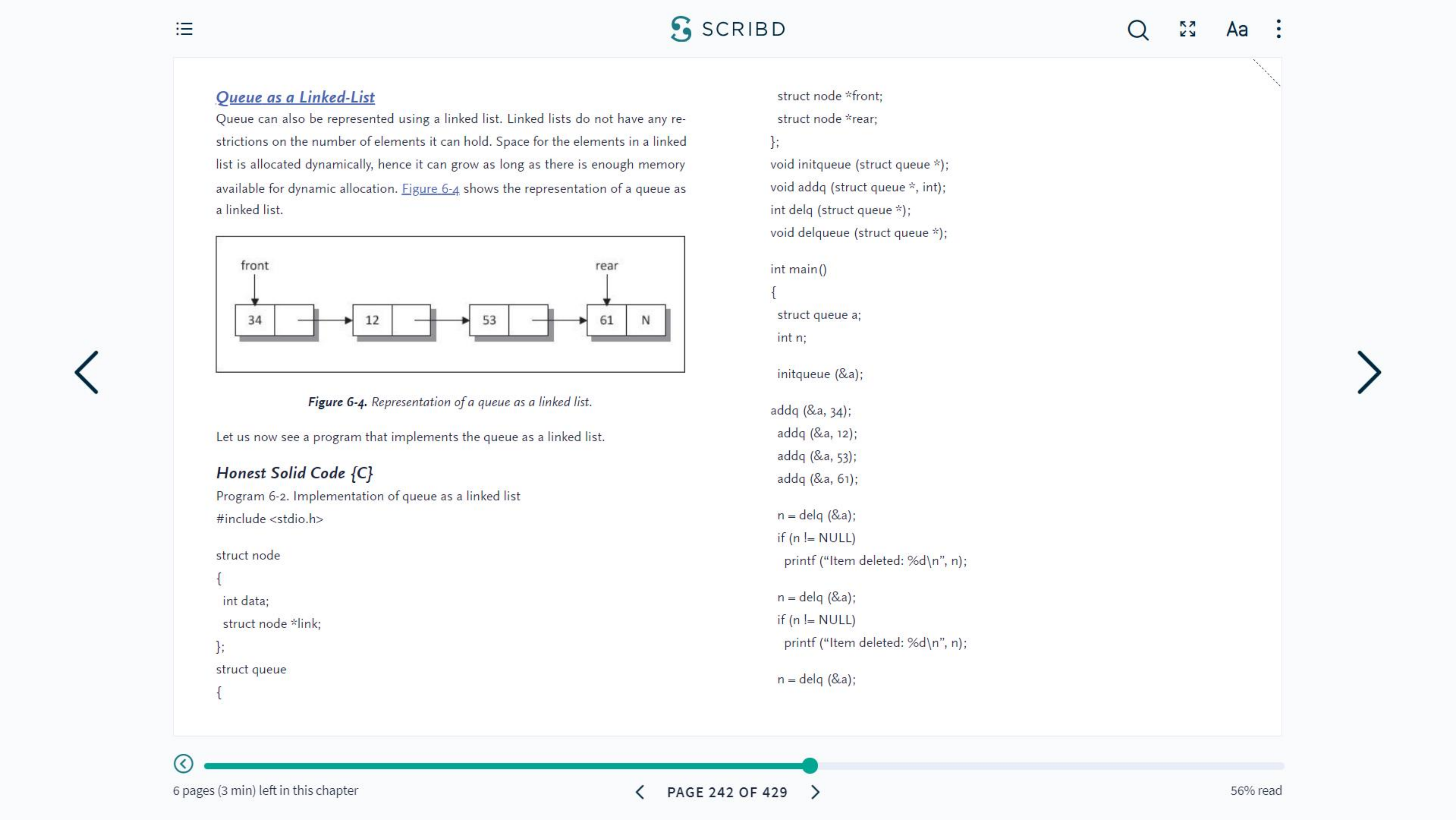
Figure 6-3. Deletion of elements from a queue.

3 pages (2 min) left in this chapter

PAGE 240 OF 429

56% read

Our program has got one limitation. Suppose we go on adding elements to the queue till the entire array gets filled. At this stage the value of **rear** would be **MAX** - 1. Now if we delete 5 elements from the queue, at the end of these deletions the value of **front** would be 5. If now we attempt to add a new element to the queue then it would be reported as full even though in reality the first five slots of the queue are empty. To overcome this situation we can implement a queue as a circular queue, which would be discussed later in this chapter.



Queue as a Linked-List

Queue can also be represented using a linked list. Linked lists do not have any restrictions on the number of elements it can hold. Space for the elements in a linked list is allocated dynamically, hence it can grow as long as there is enough memory available for dynamic allocation. Figure 6-4 shows the representation of a queue as a linked list.

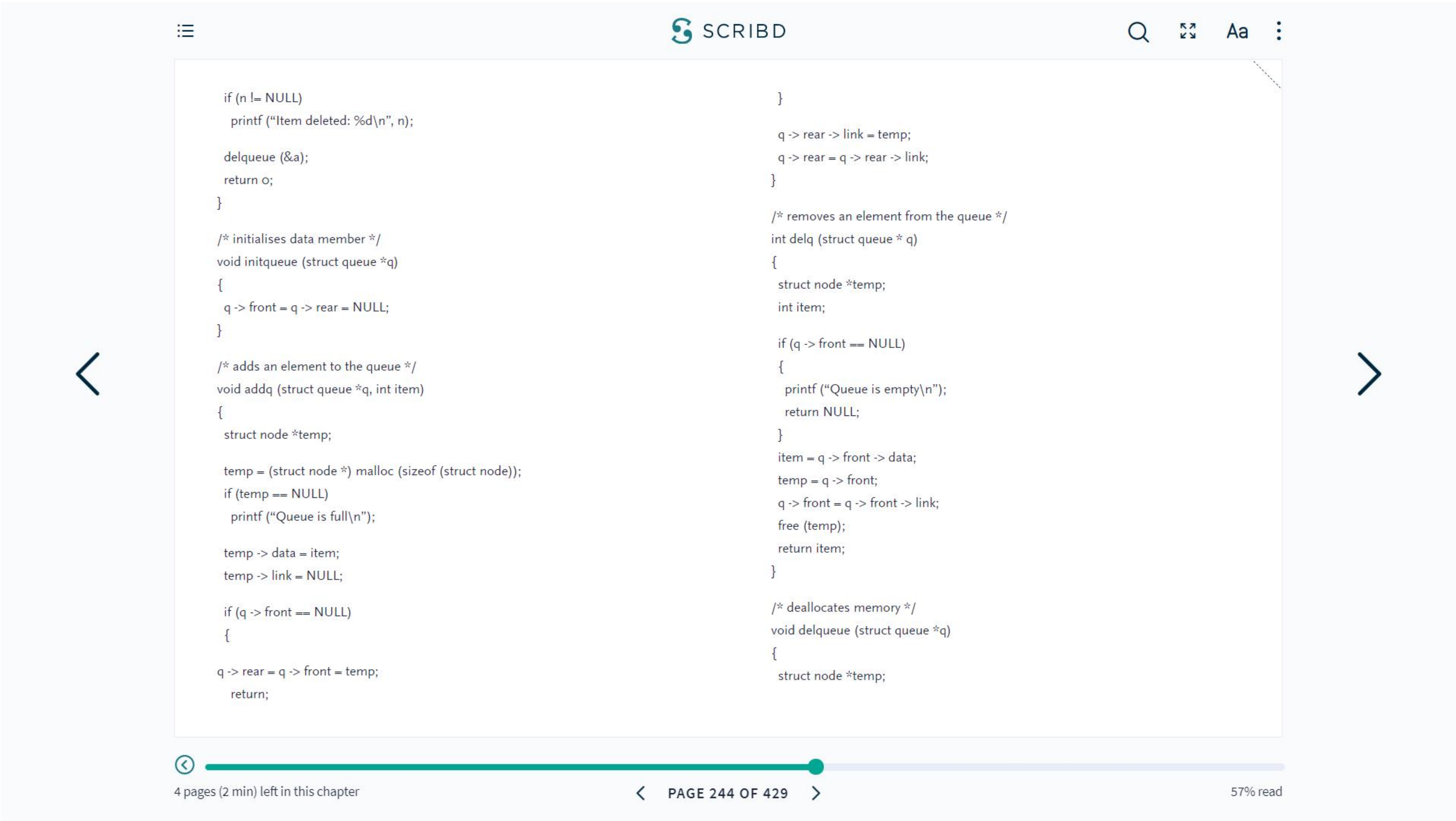
Figure 6-4. Representation of a queue as a linked list.

Let us now see a program that implements the queue as a linked list.

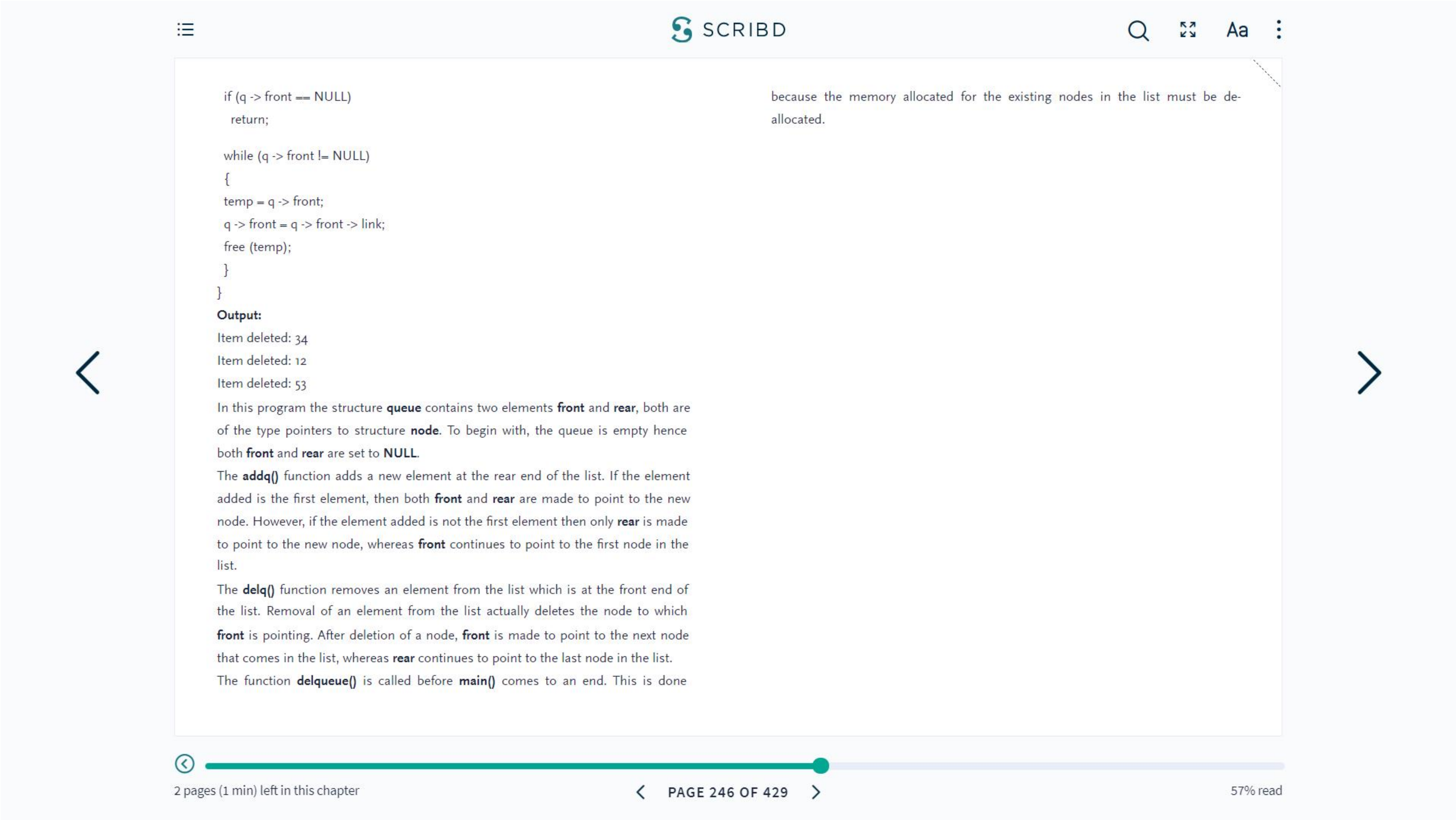
Honest Solid Code {C}

Program 6-2. Implementation of queue as a linked list

```
#include <stdio.h>

struct node
{
 int data;
 struct node *link;
};
struct queue
{
 struct node *front;
 struct node *rear;
};
void initqueue (struct queue *);
void addq (struct queue *, int);
int delq (struct queue *);
void delqueue (struct queue *);

int main()
{
 struct queue a;
 int n;

 initqueue (&a);

addq (&a, 34);
 addq (&a, 12);
 addq (&a, 53);
 addq (&a, 61);

 n = delq (&a);
 if (n != NULL)
  printf ("Item deleted: %d\n", n);

 n = delq (&a);
 if (n != NULL)
  printf ("Item deleted: %d\n", n);

 n = delq (&a);
```



```
  if (n != NULL)
    printf (“Item deleted: %d\n”, n);

  delqueue (&a);
  return 0;
}

/* initialises data member */
void initqueue (struct queue *q)
{
  q -> front = q -> rear = NULL;
}

/* adds an element to the queue */
void addq (struct queue *q, int item)
{
  struct node *temp;

  temp = (struct node *) malloc (sizeof (struct node));
  if (temp == NULL)
    printf (“Queue is full\n”);

  temp -> data = item;
  temp -> link = NULL;

  if (q -> front == NULL)
  {

q -> rear = q -> front = temp;
    return;
  }

  q -> rear -> link = temp;
  q -> rear = q -> rear -> link;
}

/* removes an element from the queue */
int delq (struct queue * q)
{
  struct node *temp;
  int item;

  if (q -> front == NULL)
  {
    printf (“Queue is empty\n”);
    return NULL;
  }
  item = q -> front -> data;
  temp = q -> front;
  q -> front = q -> front -> link;
  free (temp);
  return item;
}

/* deallocates memory */
void delqueue (struct queue *q)
{
  struct node *temp;
```



```
  if (q -> front == NULL)
    return;
  while (q -> front != NULL)
  {
  temp = q -> front;
  q -> front = q -> front -> link;
  free (temp);
  }
}
```

Output:

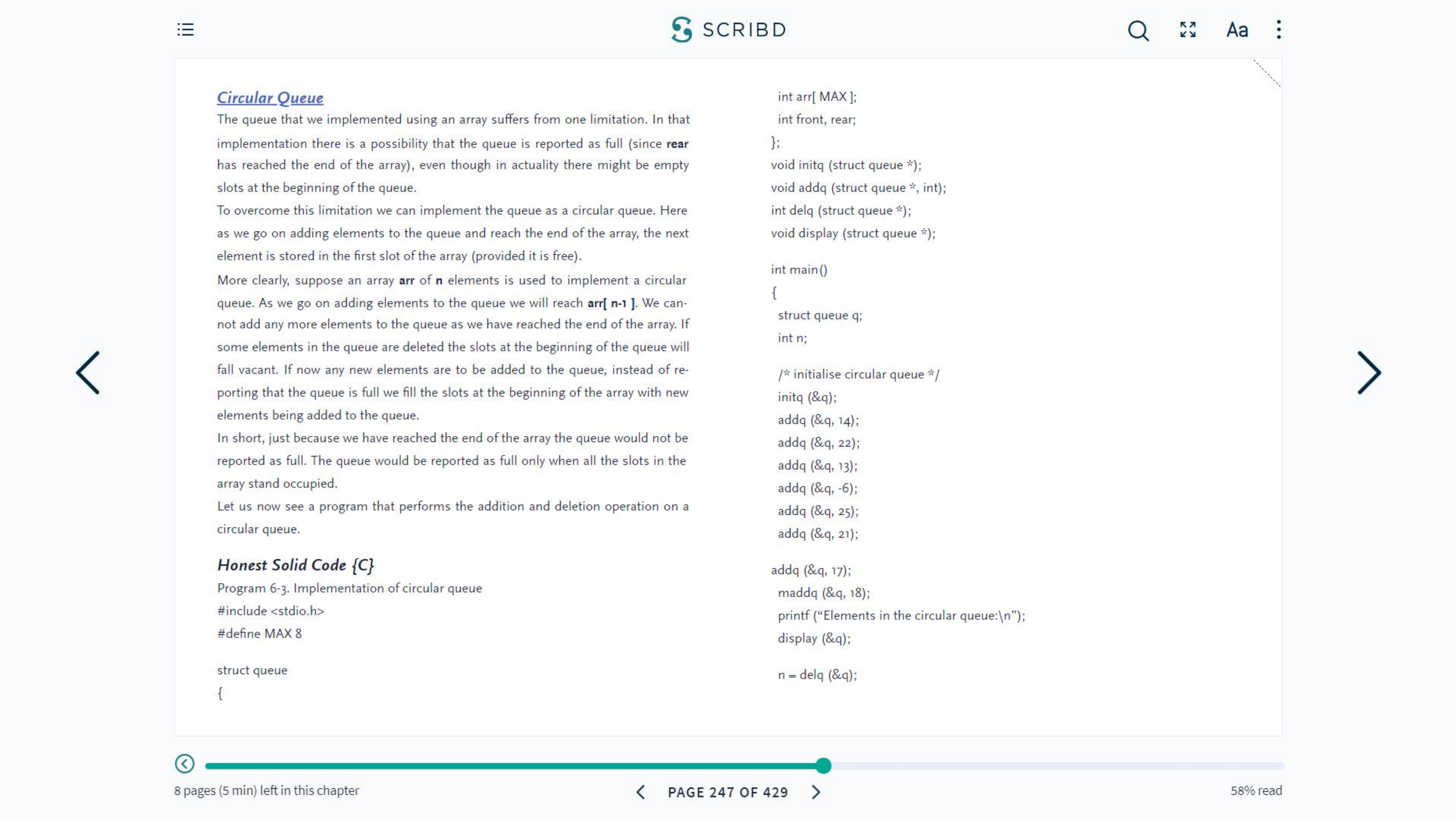
```
Item deleted: 34
Item deleted: 12
Item deleted: 53
```

In this program the structure **queue** contains two elements **front** and **rear**, both are of the type pointers to structure **node**. To begin with, the queue is empty hence both **front** and **rear** are set to **NULL**.

The **addq()** function adds a new element at the rear end of the list. If the element added is the first element, then both **front** and **rear** are made to point to the new node. However, if the element added is not the first element then only **rear** is made to point to the new node, whereas **front** continues to point to the first node in the list.

The **delq()** function removes an element from the list which is at the front end of the list. Removal of an element from the list actually deletes the node to which **front** is pointing. After deletion of a node, **front** is made to point to the next node that comes in the list, whereas **rear** continues to point to the last node in the list.

The function **delqueue()** is called before **main()** comes to an end. This is done because the memory allocated for the existing nodes in the list must be de-allocated.



Circular Queue

The queue that we implemented using an array suffers from one limitation. In that implementation there is a possibility that the queue is reported as full (since **rear** has reached the end of the array), even though in actuality there might be empty slots at the beginning of the queue.

To overcome this limitation we can implement the queue as a circular queue. Here as we go on adding elements to the queue and reach the end of the array, the next element is stored in the first slot of the array (provided it is free).

More clearly, suppose an array **arr** of **n** elements is used to implement a circular queue. As we go on adding elements to the queue we will reach **arr[n-1]**. We cannot add any more elements to the queue as we have reached the end of the array. If some elements in the queue are deleted the slots at the beginning of the queue will fall vacant. If now any new elements are to be added to the queue, instead of reporting that the queue is full we fill the slots at the beginning of the array with new elements being added to the queue.

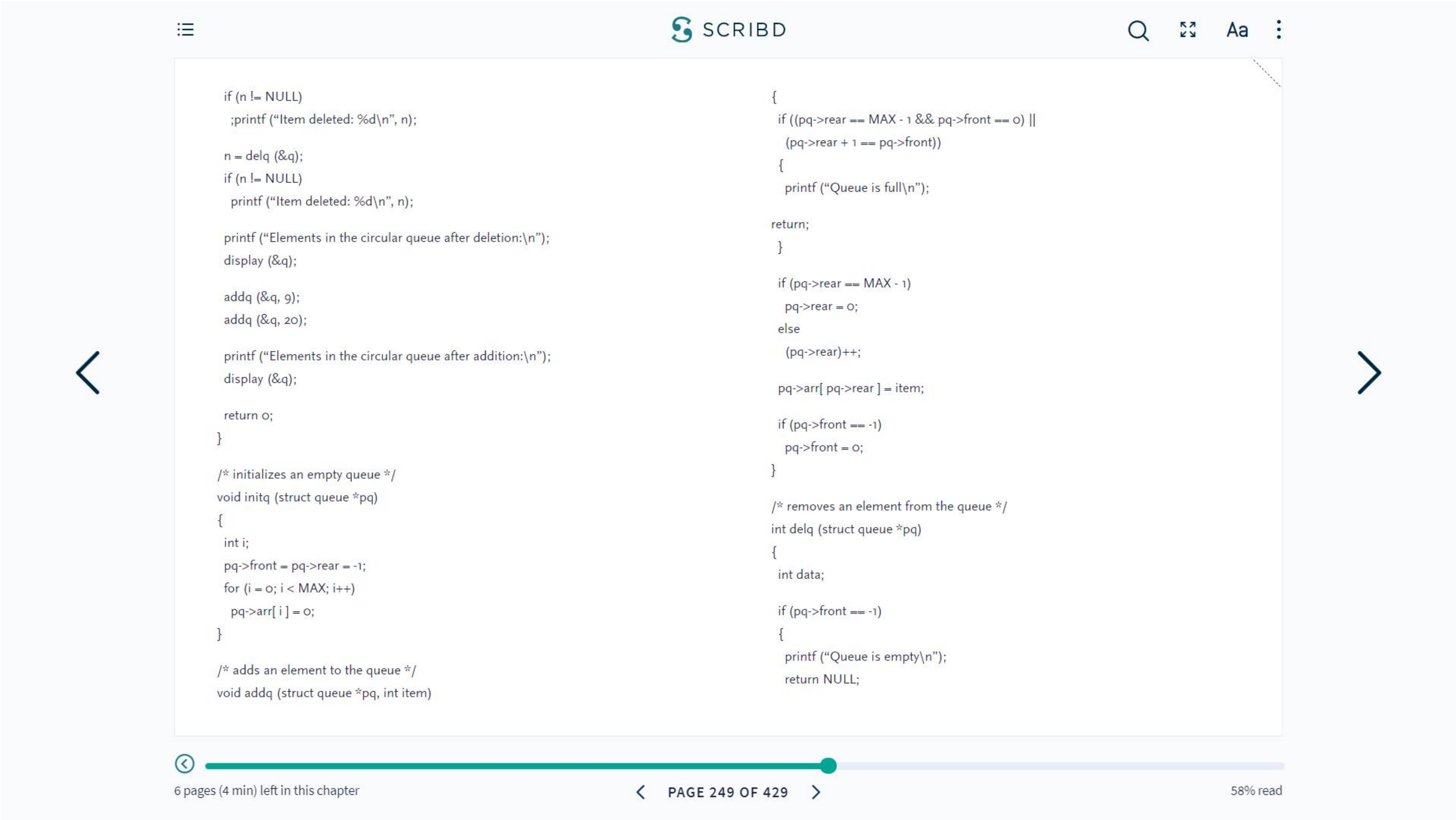
In short, just because we have reached the end of the array the queue would not be reported as full. The queue would be reported as full only when all the slots in the array stand occupied.

Let us now see a program that performs the addition and deletion operation on a circular queue.

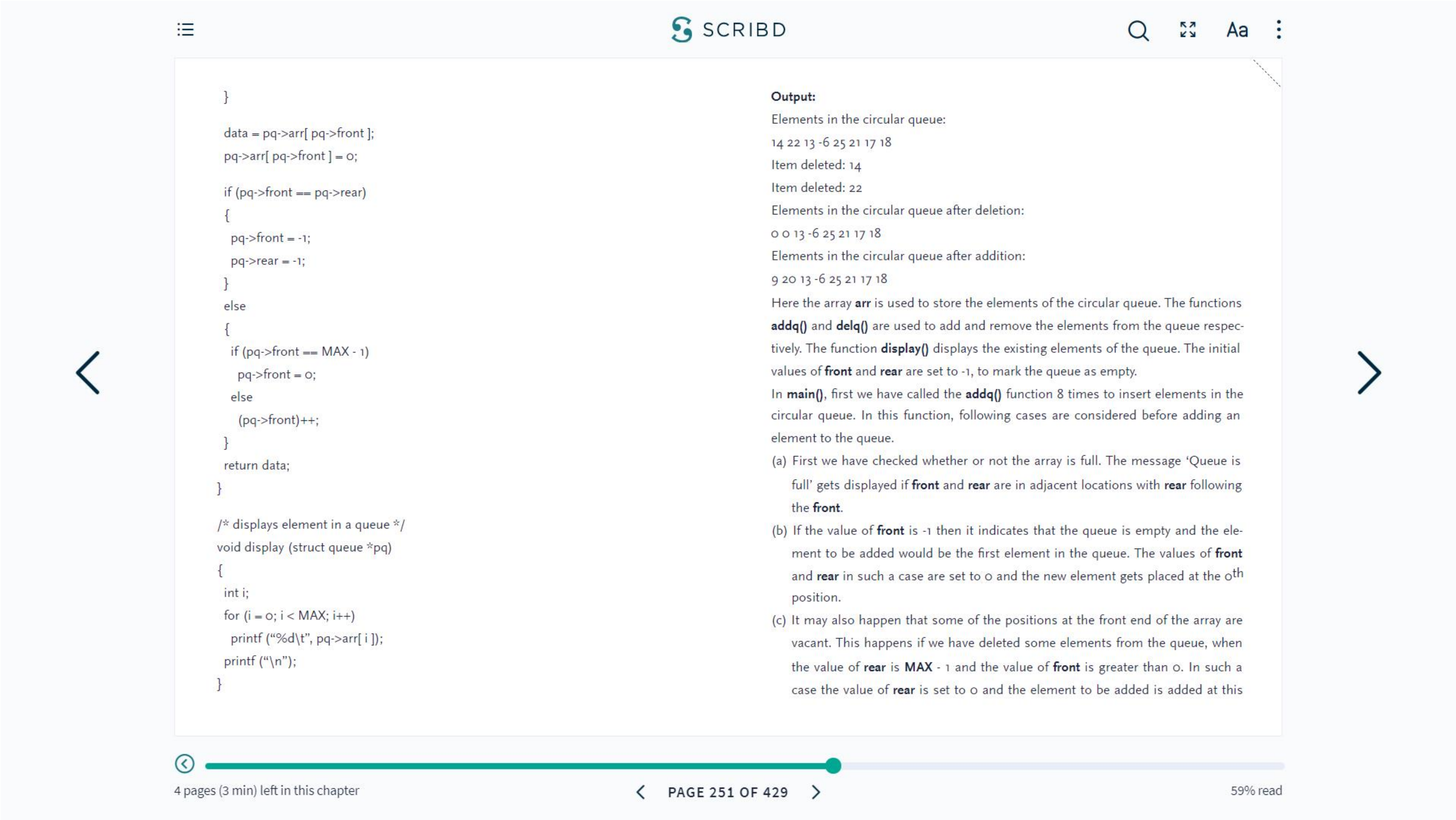
Honest Solid Code {C}

Program 6-3. Implementation of circular queue

```
#include <stdio.h>
#define MAX 8

struct queue
{
 int arr[ MAX ];
 int front, rear;
};
void initq (struct queue *);
void addq (struct queue *, int);
int delq (struct queue *);
void display (struct queue *);

int main()
{
 struct queue q;
 int n;

 /* initialise circular queue */
 initq (&q);
 addq (&q, 14);
 addq (&q, 22);
 addq (&q, 13);
 addq (&q, -6);
 addq (&q, 25);
 addq (&q, 21);

addq (&q, 17);
 maddq (&q, 18);
 printf ("Elements in the circular queue:\n");
 display (&q);

 n = delq (&q);
```



```
 if (n != NULL)
  ;printf ("Item deleted: %d\n", n);

 n = delq (&q);
 if (n != NULL)
  printf ("Item deleted: %d\n", n);

 printf ("Elements in the circular queue after deletion:\n");
 display (&q);

 addq (&q, 9);
 addq (&q, 20);

 printf ("Elements in the circular queue after addition:\n");
 display (&q);

 return 0;
}

/* initializes an empty queue */
void initq (struct queue *pq)
{
 int i;
 pq->front = pq->rear = -1;
 for (i = 0; i < MAX; i++)
  pq->arr[ i ] = 0;
}

/* adds an element to the queue */
void addq (struct queue *pq, int item)
{
 if ((pq->rear == MAX - 1 && pq->front == 0) ||
  (pq->rear + 1 == pq->front))
 {
  printf ("Queue is full\n");

return;
 }

 if (pq->rear == MAX - 1)
  pq->rear = 0;
 else
  (pq->rear)++;

 pq->arr[ pq->rear ] = item;

 if (pq->front == -1)
  pq->front = 0;
}

/* removes an element from the queue */
int delq (struct queue *pq)
{
 int data;

 if (pq->front == -1)
 {
  printf ("Queue is empty\n");
  return NULL;
```



```
    }

    data = pq->arr[ pq->front ];
    pq->arr[ pq->front ] = 0;

    if (pq->front == pq->rear)
    {
        pq->front = -1;
        pq->rear = -1;
    }
    else
    {
        if (pq->front == MAX - 1)
            pq->front = 0;
        else
            (pq->front)++;
    }
    return data;
}

/* displays element in a queue */
void display (struct queue *pq)
{
    int i;
    for (i = 0; i < MAX; i++)
        printf ("%d\t", pq->arr[ i ]);
    printf ("\n");
}
```

Output:

```
Elements in the circular queue:
14 22 13 -6 25 21 17 18
Item deleted: 14
Item deleted: 22
Elements in the circular queue after deletion:
0 0 13 -6 25 21 17 18
Elements in the circular queue after addition:
9 20 13 -6 25 21 17 18
```

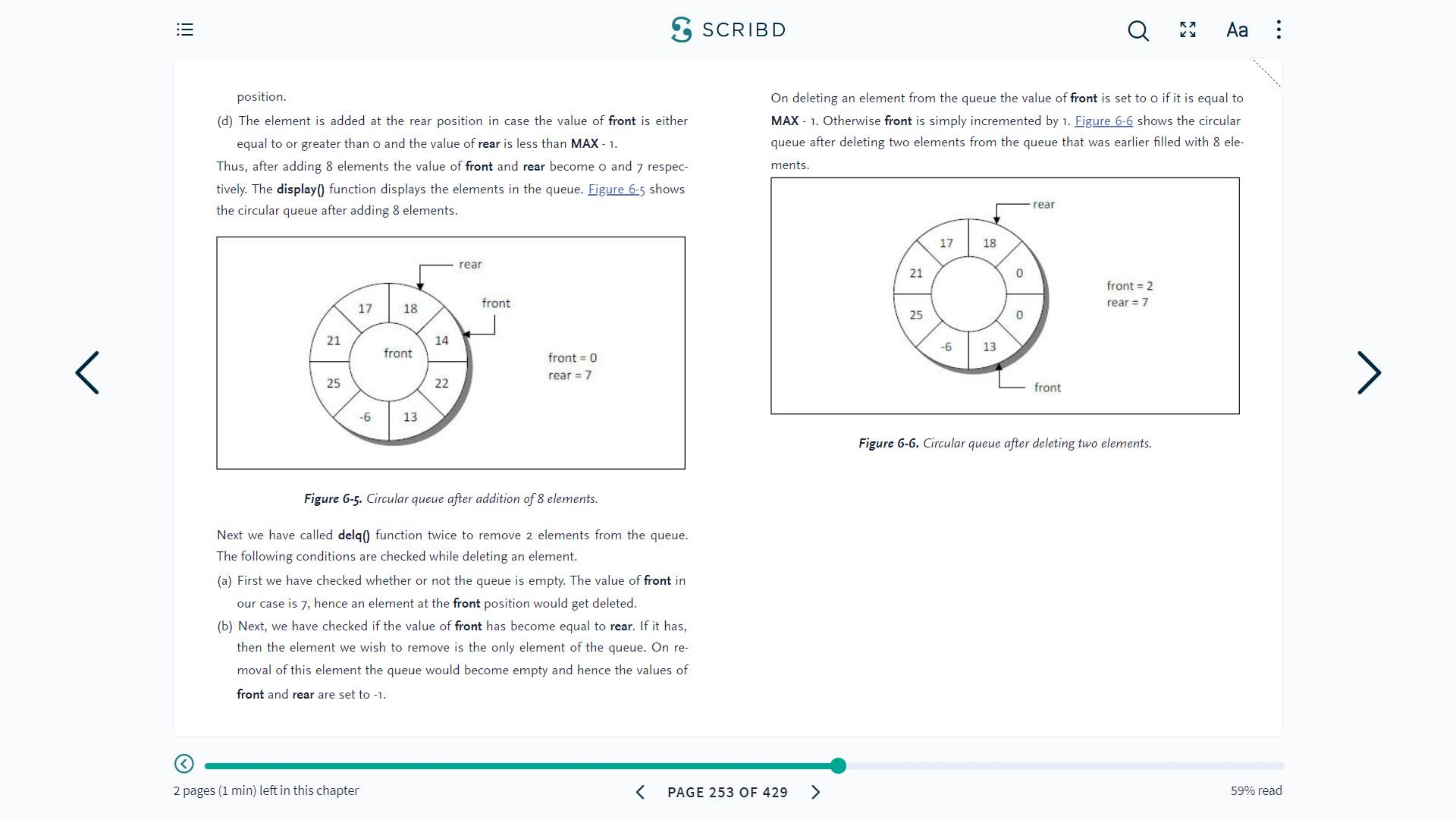
Here the array **arr** is used to store the elements of the circular queue. The functions **addq()** and **delq()** are used to add and remove the elements from the queue respectively. The function **display()** displays the existing elements of the queue. The initial values of **front** and **rear** are set to -1, to mark the queue as empty.

In **main()**, first we have called the **addq()** function 8 times to insert elements in the circular queue. In this function, following cases are considered before adding an element to the queue.

(a) First we have checked whether or not the array is full. The message 'Queue is full' gets displayed if **front** and **rear** are in adjacent locations with **rear** following the **front**.

(b) If the value of **front** is -1 then it indicates that the queue is empty and the element to be added would be the first element in the queue. The values of **front** and **rear** in such a case are set to 0 and the new element gets placed at the 0^{th} position.

(c) It may also happen that some of the positions at the front end of the array are vacant. This happens if we have deleted some elements from the queue, when the value of **rear** is **MAX** - 1 and the value of **front** is greater than 0. In such a case the value of **rear** is set to 0 and the element to be added is added at this



position.

(d) The element is added at the rear position in case the value of **front** is either equal to or greater than 0 and the value of **rear** is less than **MAX** - 1.

Thus, after adding 8 elements the value of **front** and **rear** become 0 and 7 respectively. The **display()** function displays the elements in the queue. Figure 6-5 shows the circular queue after adding 8 elements.

Figure 6-5. *Circular queue after addition of 8 elements.*

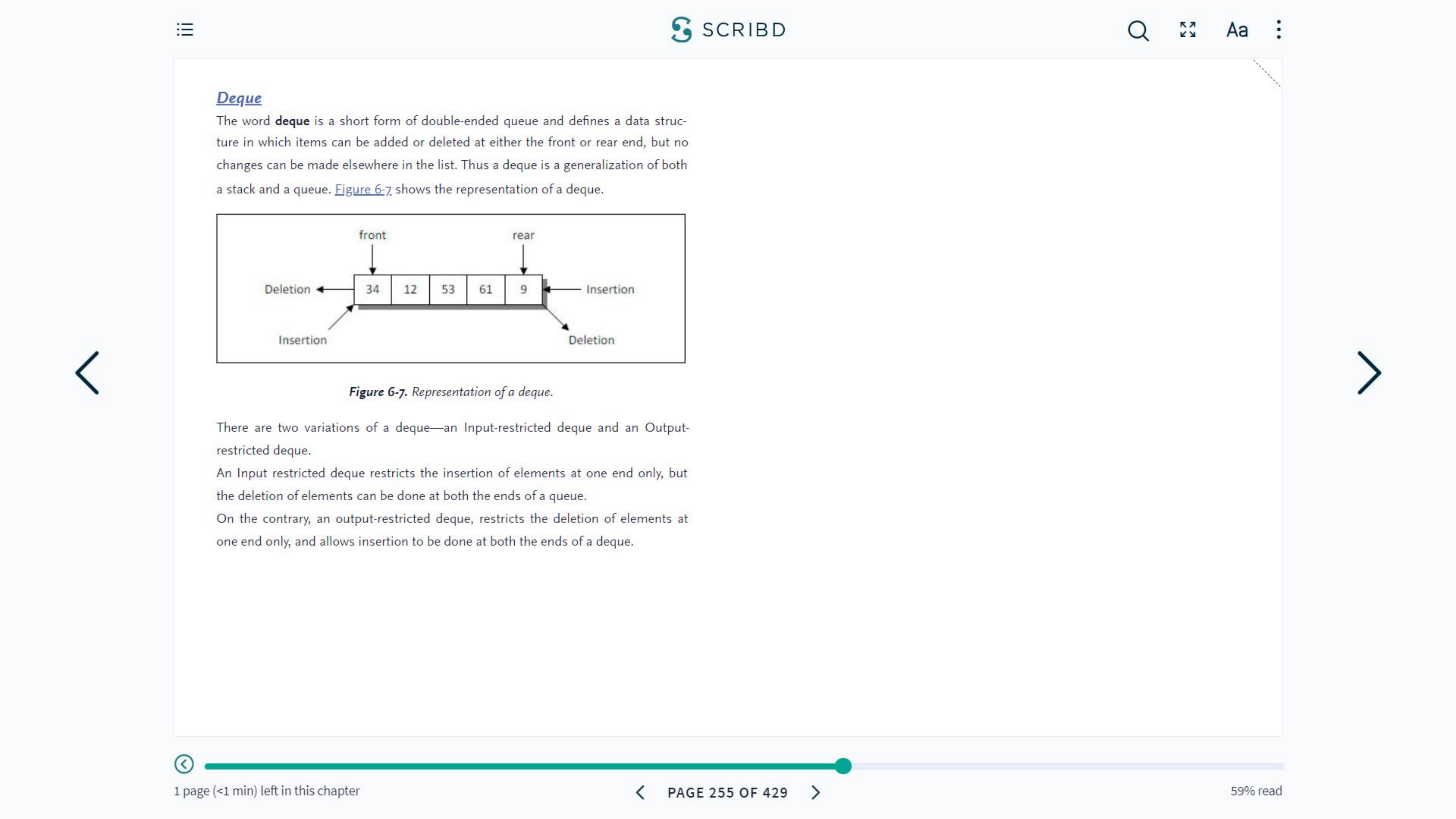
Next we have called **delq()** function twice to remove 2 elements from the queue. The following conditions are checked while deleting an element.

(a) First we have checked whether or not the queue is empty. The value of **front** in our case is 7, hence an element at the **front** position would get deleted.

(b) Next, we have checked if the value of **front** has become equal to **rear**. If it has, then the element we wish to remove is the only element of the queue. On removal of this element the queue would become empty and hence the values of **front** and **rear** are set to -1.

On deleting an element from the queue the value of **front** is set to 0 if it is equal to **MAX** - 1. Otherwise **front** is simply incremented by 1. Figure 6-6 shows the circular queue after deleting two elements from the queue that was earlier filled with 8 elements.

Figure 6-6. *Circular queue after deleting two elements.*



Deque

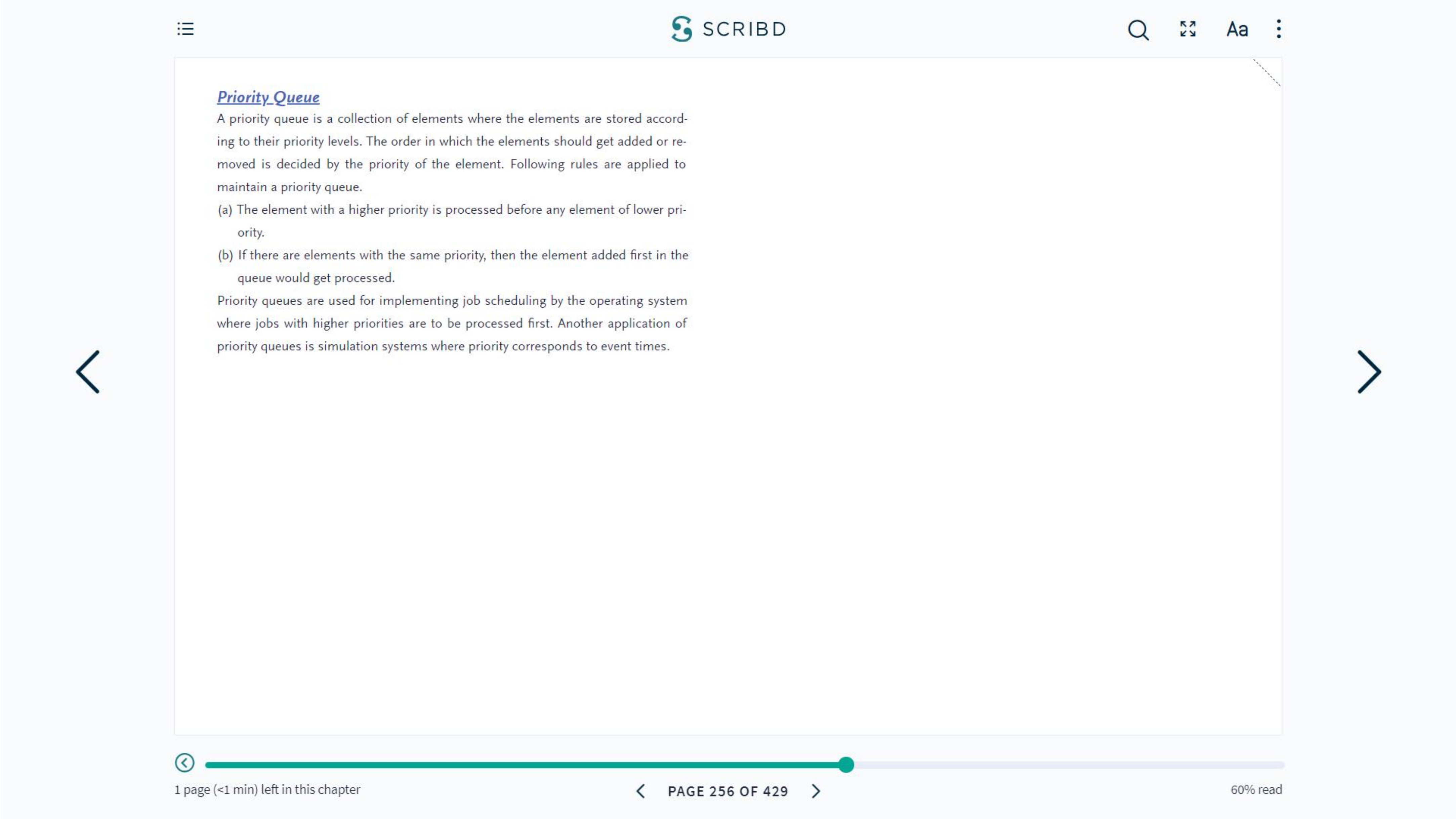
The word **deque** is a short form of double-ended queue and defines a data structure in which items can be added or deleted at either the front or rear end, but no changes can be made elsewhere in the list. Thus a deque is a generalization of both a stack and a queue. Figure 6-7 shows the representation of a deque.

Figure 6-7. *Representation of a deque.*

There are two variations of a deque—an Input-restricted deque and an Output-restricted deque.

An Input restricted deque restricts the insertion of elements at one end only, but the deletion of elements can be done at both the ends of a queue.

On the contrary, an output-restricted deque, restricts the deletion of elements at one end only, and allows insertion to be done at both the ends of a deque.



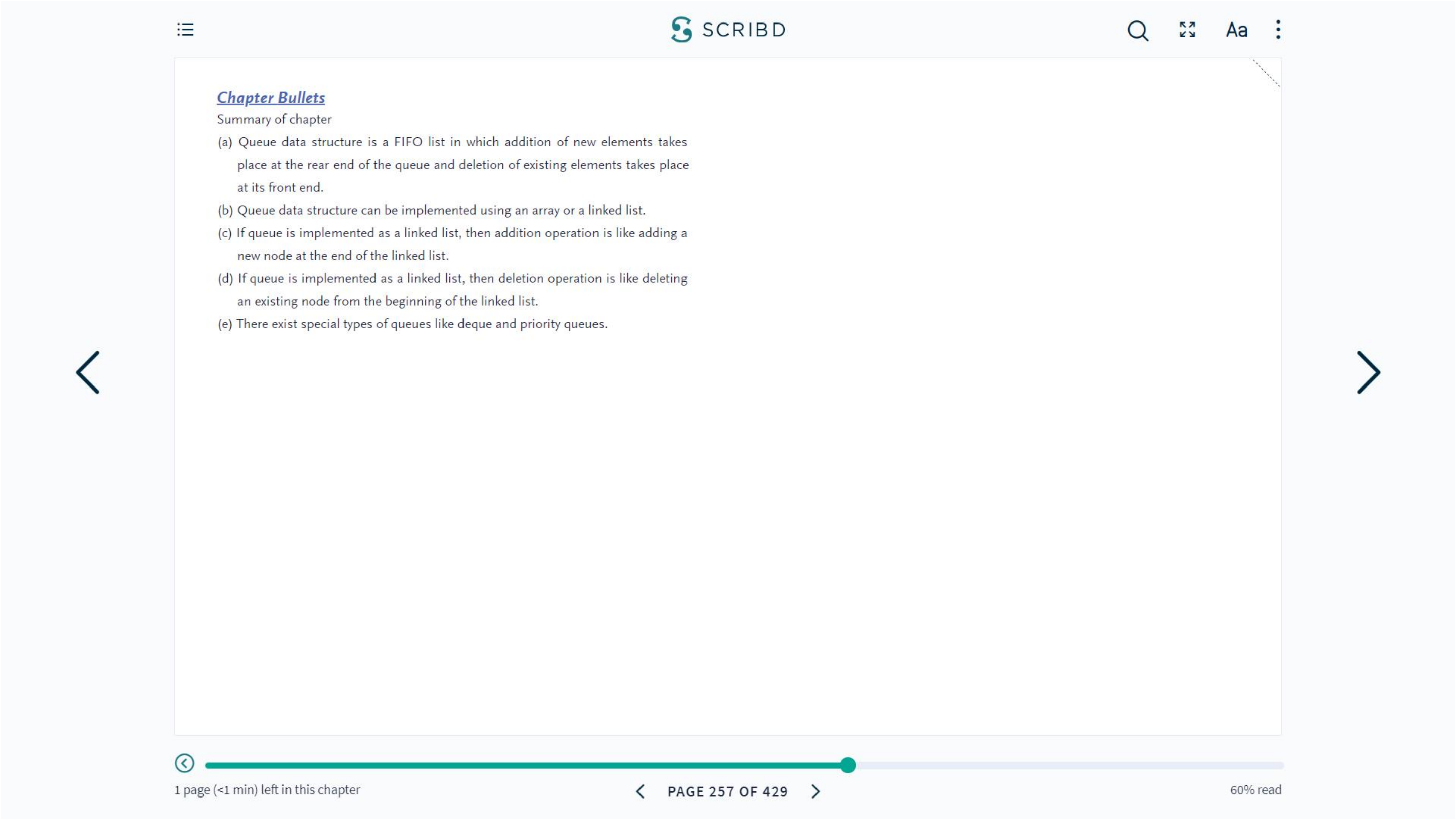
Priority Queue

A priority queue is a collection of elements where the elements are stored according to their priority levels. The order in which the elements should get added or removed is decided by the priority of the element. Following rules are applied to maintain a priority queue.

(a) The element with a higher priority is processed before any element of lower priority.

(b) If there are elements with the same priority, then the element added first in the queue would get processed.

Priority queues are used for implementing job scheduling by the operating system where jobs with higher priorities are to be processed first. Another application of priority queues is simulation systems where priority corresponds to event times.



Chapter Bullets

Summary of chapter

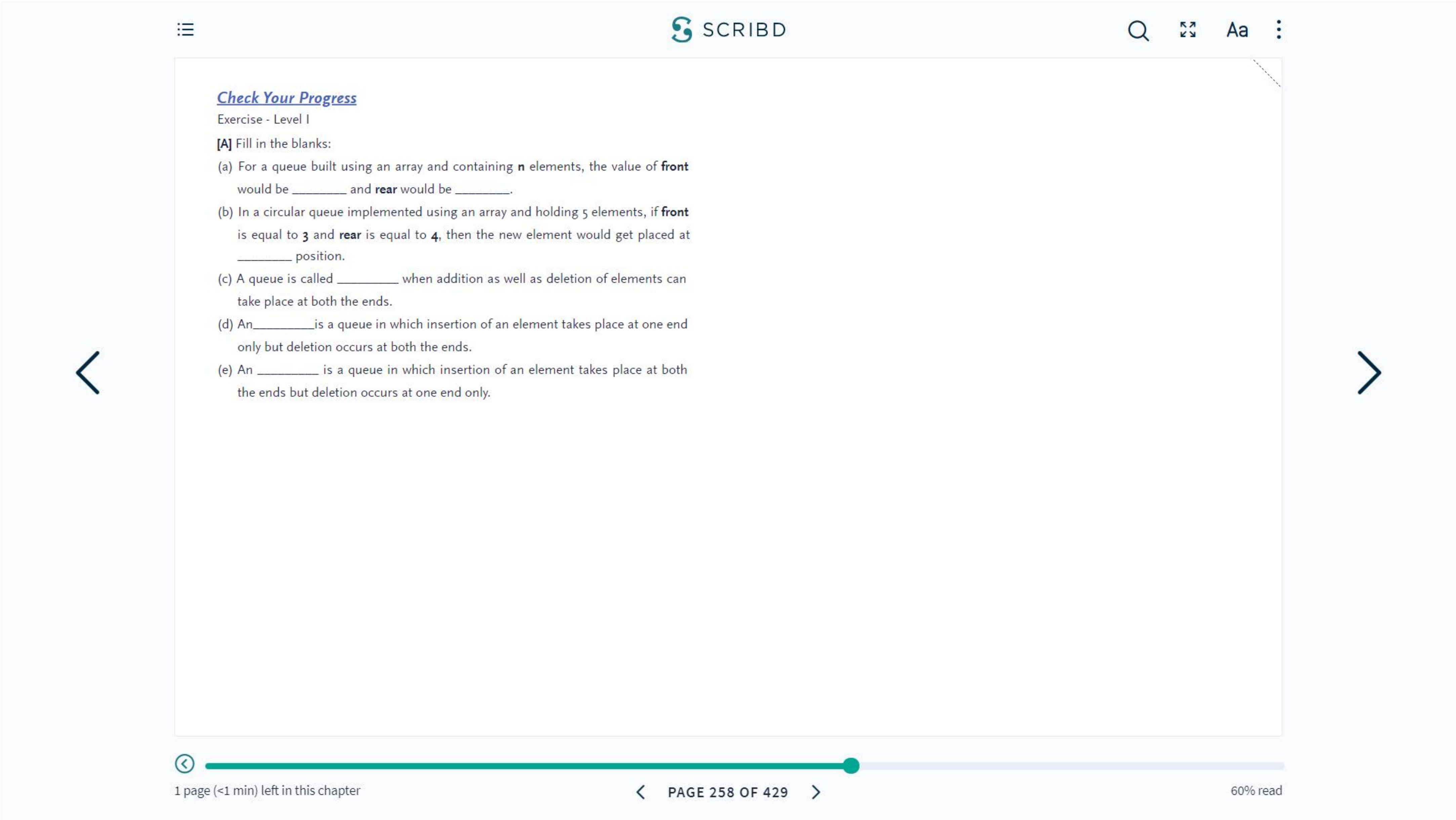
(a) Queue data structure is a FIFO list in which addition of new elements takes place at the rear end of the queue and deletion of existing elements takes place at its front end.

(b) Queue data structure can be implemented using an array or a linked list.

(c) If queue is implemented as a linked list, then addition operation is like adding a new node at the end of the linked list.

(d) If queue is implemented as a linked list, then deletion operation is like deleting an existing node from the beginning of the linked list.

(e) There exist special types of queues like deque and priority queues.



Check Your Progress

Exercise - Level I

[A] Fill in the blanks:

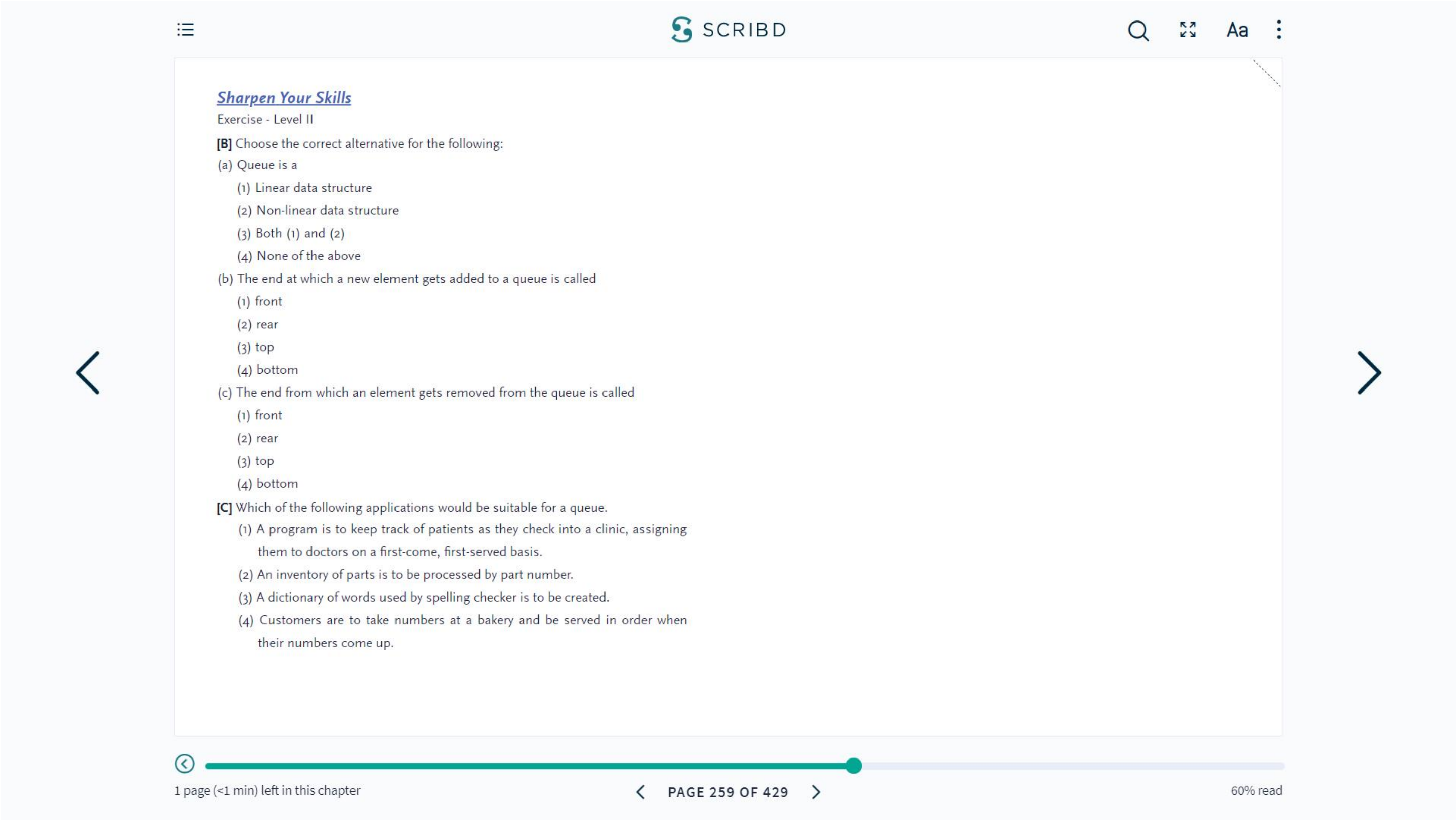
(a) For a queue built using an array and containing **n** elements, the value of **front** would be ________ and **rear** would be ________.

(b) In a circular queue implemented using an array and holding 5 elements, if **front** is equal to **3** and **rear** is equal to **4**, then the new element would get placed at ________ position.

(c) A queue is called _________ when addition as well as deletion of elements can take place at both the ends.

(d) An_________is a queue in which insertion of an element takes place at one end only but deletion occurs at both the ends.

(e) An _________ is a queue in which insertion of an element takes place at both the ends but deletion occurs at one end only.



Sharpen Your Skills

Exercise - Level II

[B] Choose the correct alternative for the following:

(a) Queue is a

(1) Linear data structure

(2) Non-linear data structure

(3) Both (1) and (2)

(4) None of the above

(b) The end at which a new element gets added to a queue is called

(1) front

(2) rear

(3) top

(4) bottom

(c) The end from which an element gets removed from the queue is called

(1) front

(2) rear

(3) top

(4) bottom

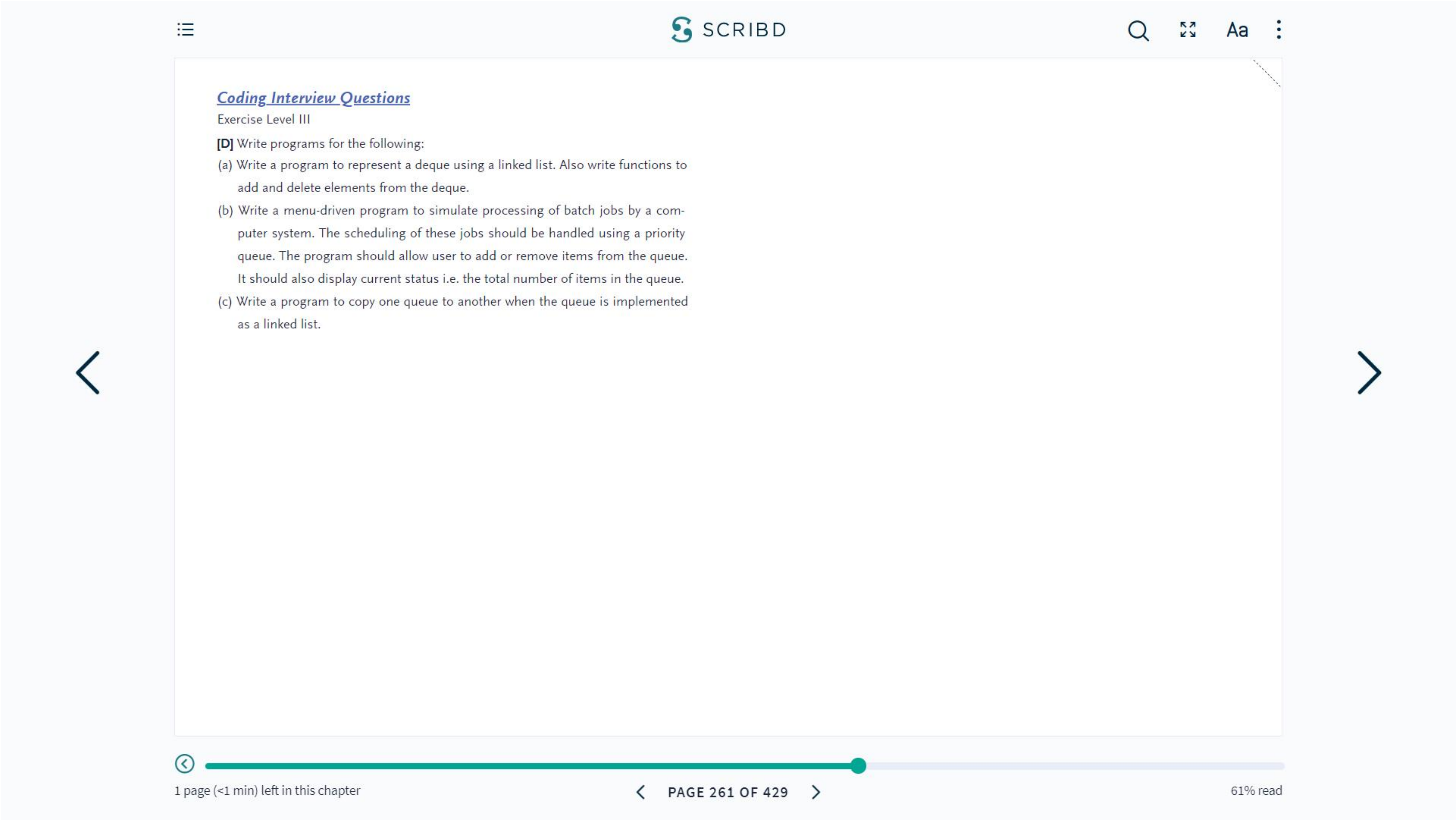
[C] Which of the following applications would be suitable for a queue.

(1) A program is to keep track of patients as they check into a clinic, assigning them to doctors on a first-come, first-served basis.

(2) An inventory of parts is to be processed by part number.

(3) A dictionary of words used by spelling checker is to be created.

(4) Customers are to take numbers at a bakery and be served in order when their numbers come up.



Coding Interview Questions

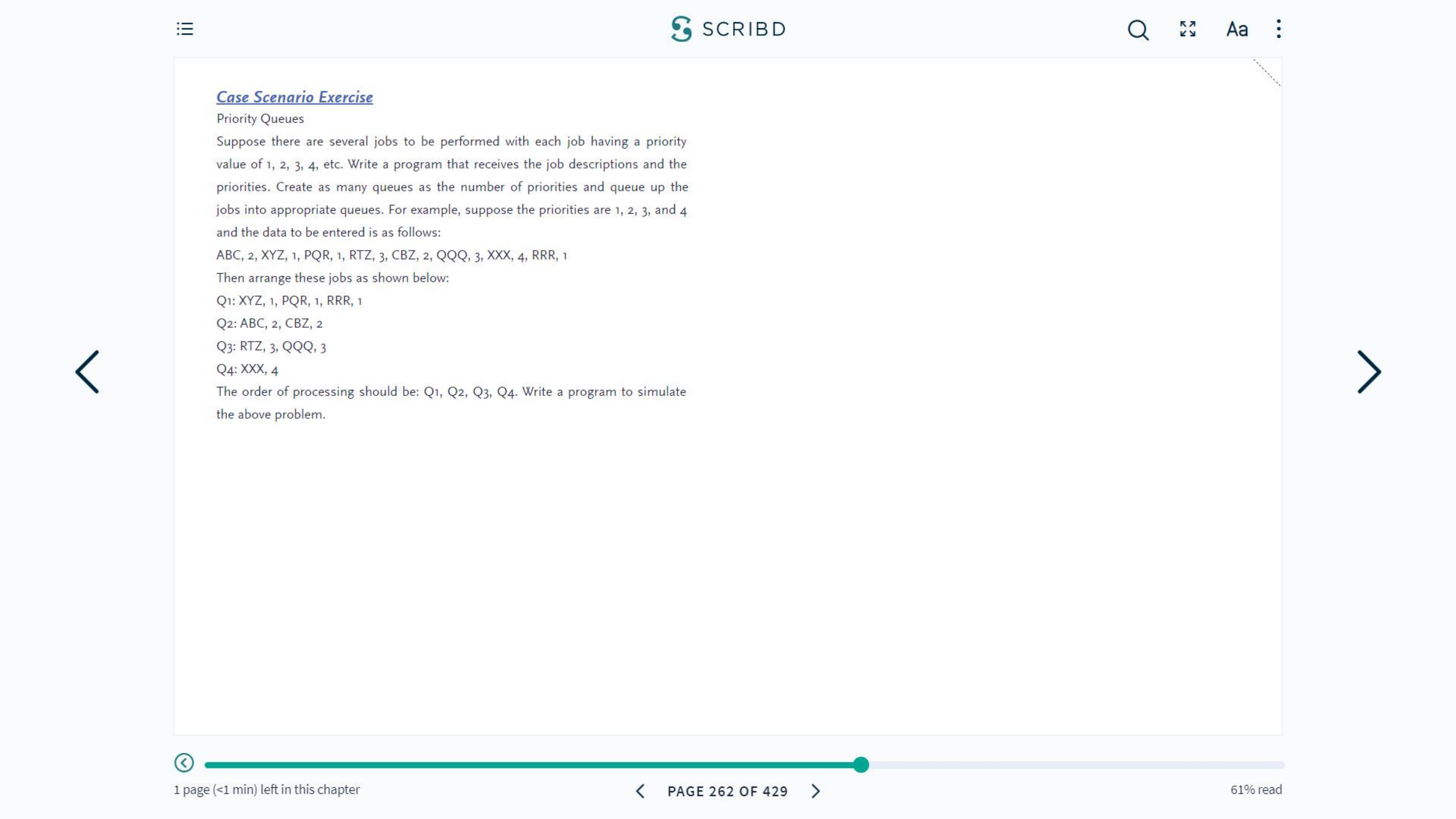
Exercise Level III

[D] Write programs for the following:

(a) Write a program to represent a deque using a linked list. Also write functions to add and delete elements from the deque.

(b) Write a menu-driven program to simulate processing of batch jobs by a computer system. The scheduling of these jobs should be handled using a priority queue. The program should allow user to add or remove items from the queue. It should also display current status i.e. the total number of items in the queue.

(c) Write a program to copy one queue to another when the queue is implemented as a linked list.



Case Scenario Exercise

Priority Queues

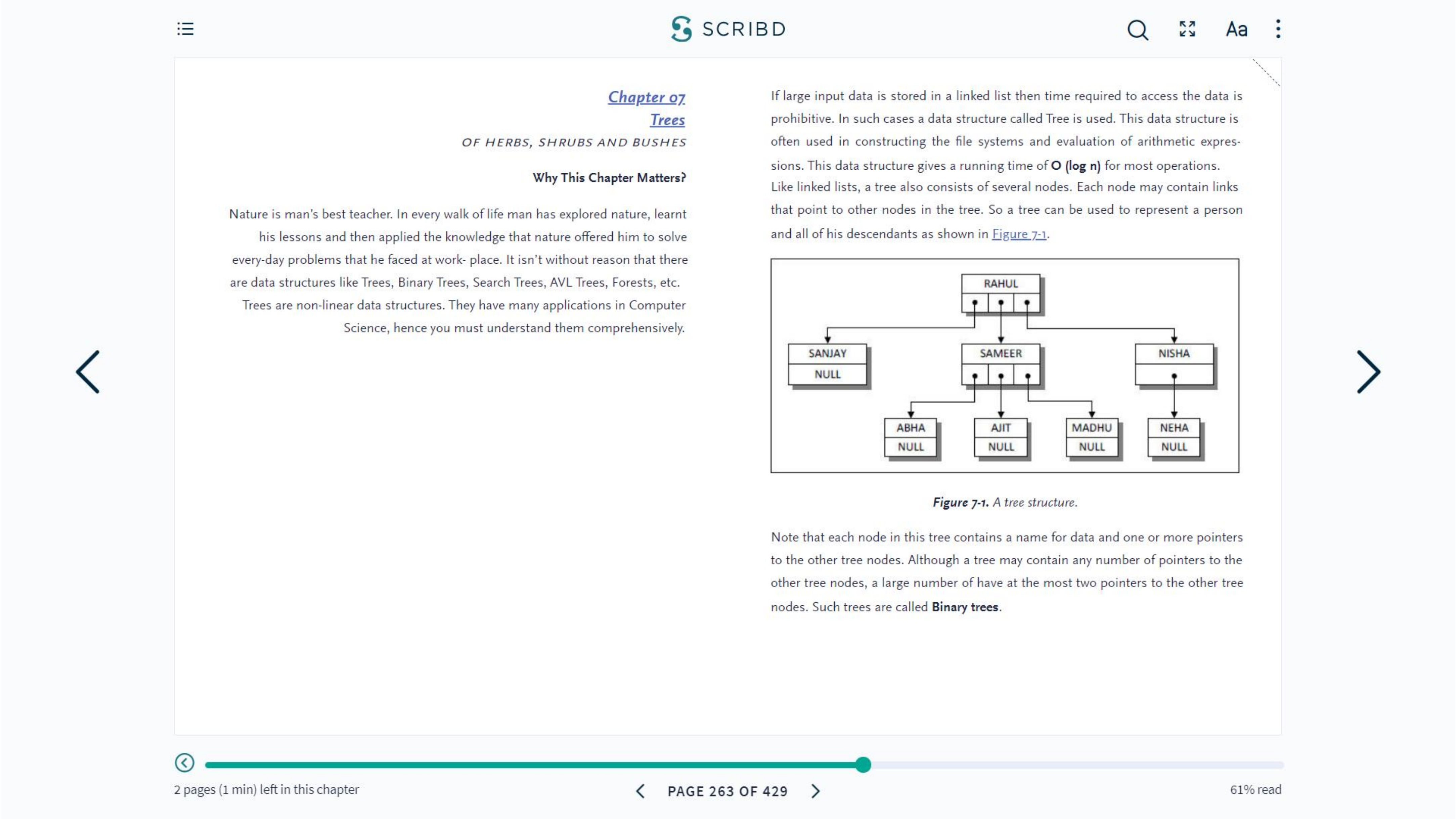
Suppose there are several jobs to be performed with each job having a priority value of 1, 2, 3, 4, etc. Write a program that receives the job descriptions and the priorities. Create as many queues as the number of priorities and queue up the jobs into appropriate queues. For example, suppose the priorities are 1, 2, 3, and 4 and the data to be entered is as follows:

ABC, 2, XYZ, 1, PQR, 1, RTZ, 3, CBZ, 2, QQQ, 3, XXX, 4, RRR, 1

Then arrange these jobs as shown below:

Q1: XYZ, 1, PQR, 1, RRR, 1
Q2: ABC, 2, CBZ, 2
Q3: RTZ, 3, QQQ, 3
Q4: XXX, 4

The order of processing should be: Q1, Q2, Q3, Q4. Write a program to simulate the above problem.



Chapter 07

Trees

OF HERBS, SHRUBS AND BUSHES

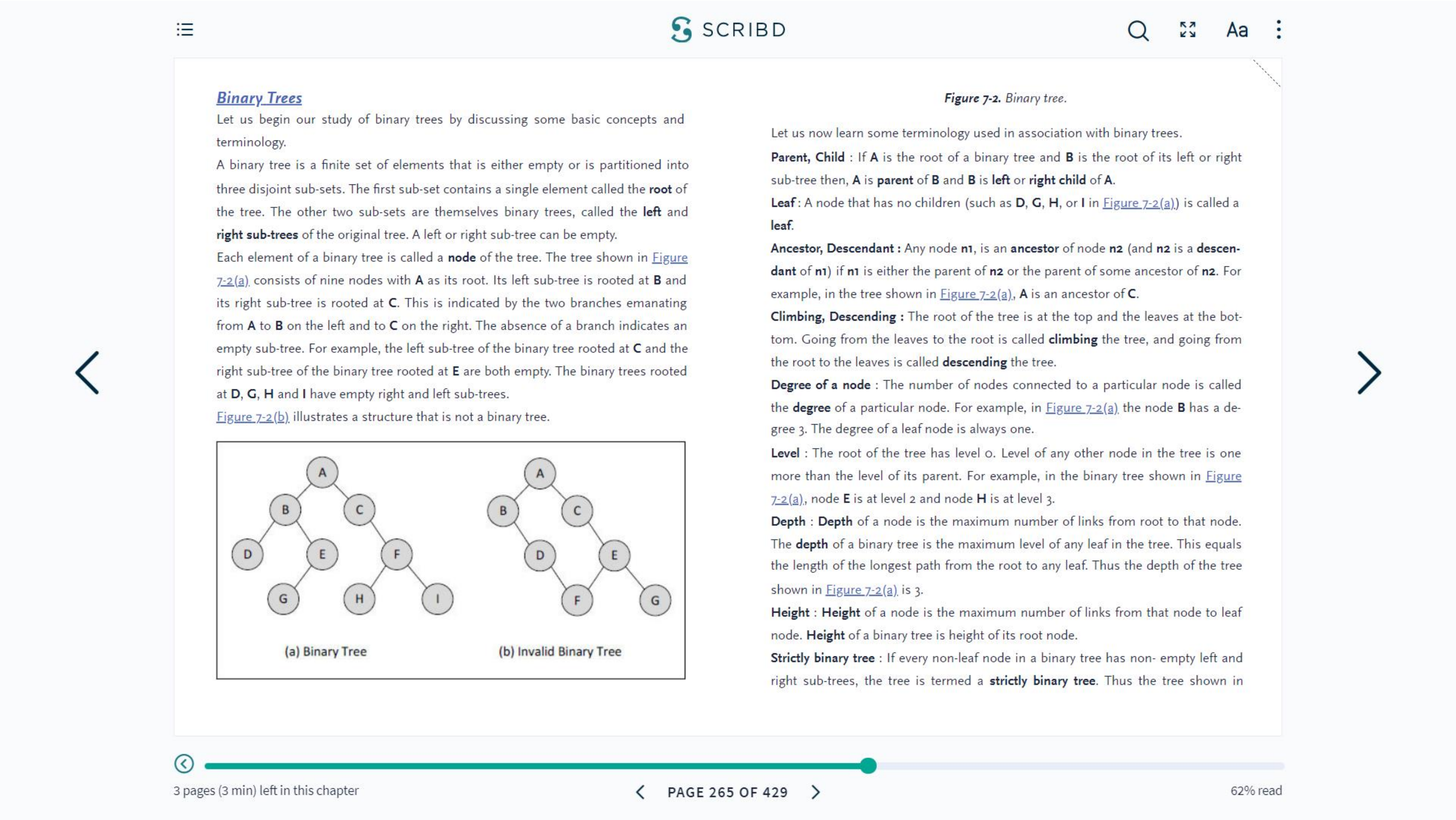
Why This Chapter Matters?

Nature is man's best teacher. In every walk of life man has explored nature, learnt his lessons and then applied the knowledge that nature offered him to solve every-day problems that he faced at work- place. It isn't without reason that there are data structures like Trees, Binary Trees, Search Trees, AVL Trees, Forests, etc. Trees are non-linear data structures. They have many applications in Computer Science, hence you must understand them comprehensively.

If large input data is stored in a linked list then time required to access the data is prohibitive. In such cases a data structure called Tree is used. This data structure is often used in constructing the file systems and evaluation of arithmetic expressions. This data structure gives a running time of **O (log n)** for most operations. Like linked lists, a tree also consists of several nodes. Each node may contain links that point to other nodes in the tree. So a tree can be used to represent a person and all of his descendants as shown in Figure 7-1.

Figure 7-1. *A tree structure.*

Note that each node in this tree contains a name for data and one or more pointers to the other tree nodes. Although a tree may contain any number of pointers to the other tree nodes, a large number of have at the most two pointers to the other tree nodes. Such trees are called **Binary trees**.



Binary Trees

Let us begin our study of binary trees by discussing some basic concepts and terminology.

A binary tree is a finite set of elements that is either empty or is partitioned into three disjoint sub-sets. The first sub-set contains a single element called the **root** of the tree. The other two sub-sets are themselves binary trees, called the **left** and **right sub-trees** of the original tree. A left or right sub-tree can be empty.

Each element of a binary tree is called a **node** of the tree. The tree shown in Figure 7-2(a) consists of nine nodes with **A** as its root. Its left sub-tree is rooted at **B** and its right sub-tree is rooted at **C**. This is indicated by the two branches emanating from **A** to **B** on the left and to **C** on the right. The absence of a branch indicates an empty sub-tree. For example, the left sub-tree of the binary tree rooted at **C** and the right sub-tree of the binary tree rooted at **E** are both empty. The binary trees rooted at **D**, **G**, **H** and **I** have empty right and left sub-trees.

Figure 7-2(b) illustrates a structure that is not a binary tree.

Figure 7-2. Binary tree.

Let us now learn some terminology used in association with binary trees.

Parent, Child : If **A** is the root of a binary tree and **B** is the root of its left or right sub-tree then, **A** is **parent** of **B** and **B** is **left** or **right child** of **A**.

Leaf : A node that has no children (such as **D**, **G**, **H**, or **I** in Figure 7-2(a)) is called a **leaf**.

Ancestor, Descendant : Any node **n1**, is an **ancestor** of node **n2** (and **n2** is a **descendant** of **n1**) if **n1** is either the parent of **n2** or the parent of some ancestor of **n2**. For example, in the tree shown in Figure 7-2(a), **A** is an ancestor of **C**.

Climbing, Descending : The root of the tree is at the top and the leaves at the bottom. Going from the leaves to the root is called **climbing** the tree, and going from the root to the leaves is called **descending** the tree.

Degree of a node : The number of nodes connected to a particular node is called the **degree** of a particular node. For example, in Figure 7-2(a) the node **B** has a degree 3. The degree of a leaf node is always one.

Level : The root of the tree has level 0. Level of any other node in the tree is one more than the level of its parent. For example, in the binary tree shown in Figure 7-2(a), node **E** is at level 2 and node **H** is at level 3.

Depth : **Depth** of a node is the maximum number of links from root to that node. The **depth** of a binary tree is the maximum level of any leaf in the tree. This equals the length of the longest path from the root to any leaf. Thus the depth of the tree shown in Figure 7-2(a) is 3.

Height : **Height** of a node is the maximum number of links from that node to leaf node. **Height** of a binary tree is height of its root node.

Strictly binary tree : If every non-leaf node in a binary tree has non- empty left and right sub-trees, the tree is termed a **strictly binary tree**. Thus the tree shown in

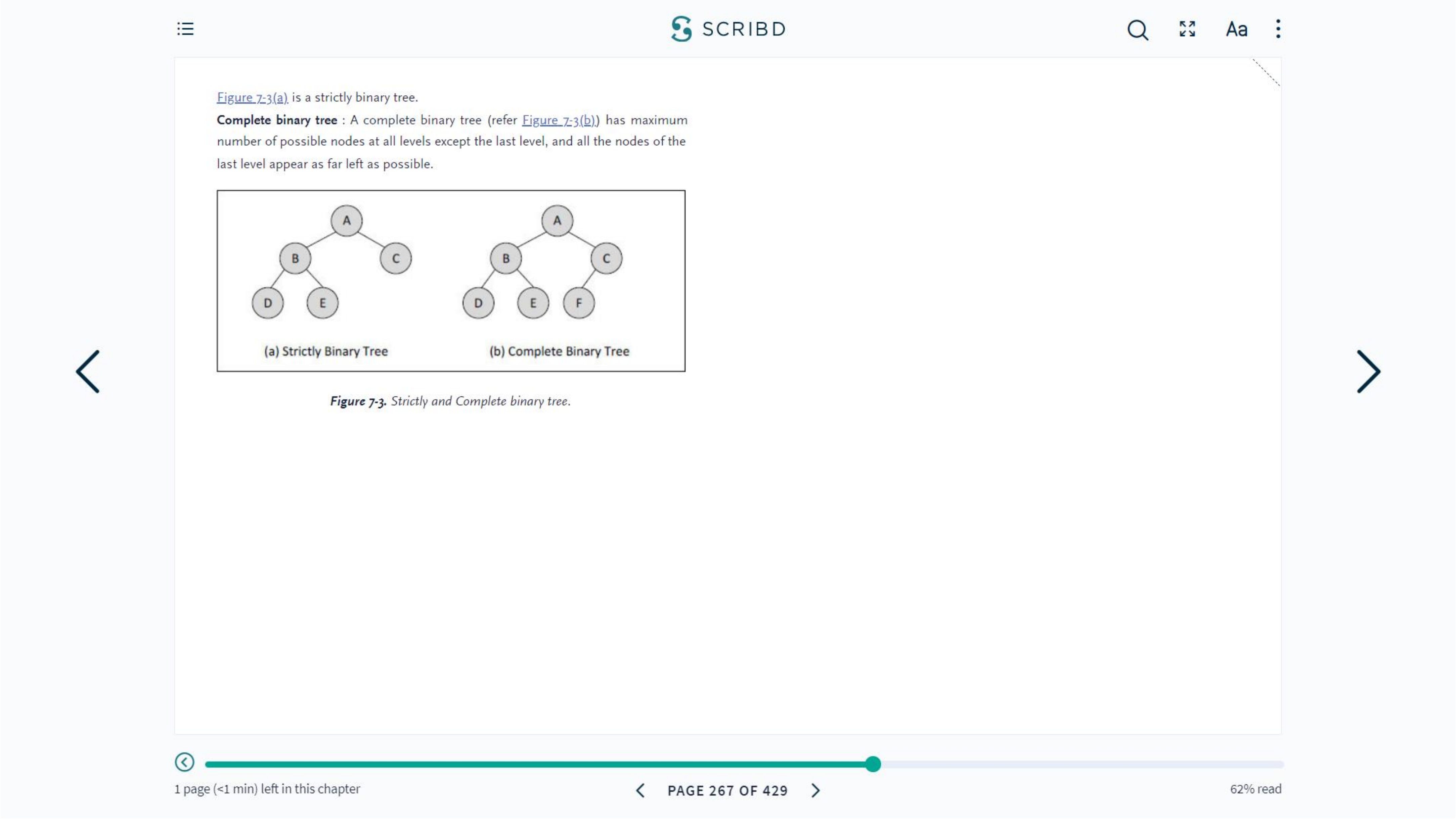


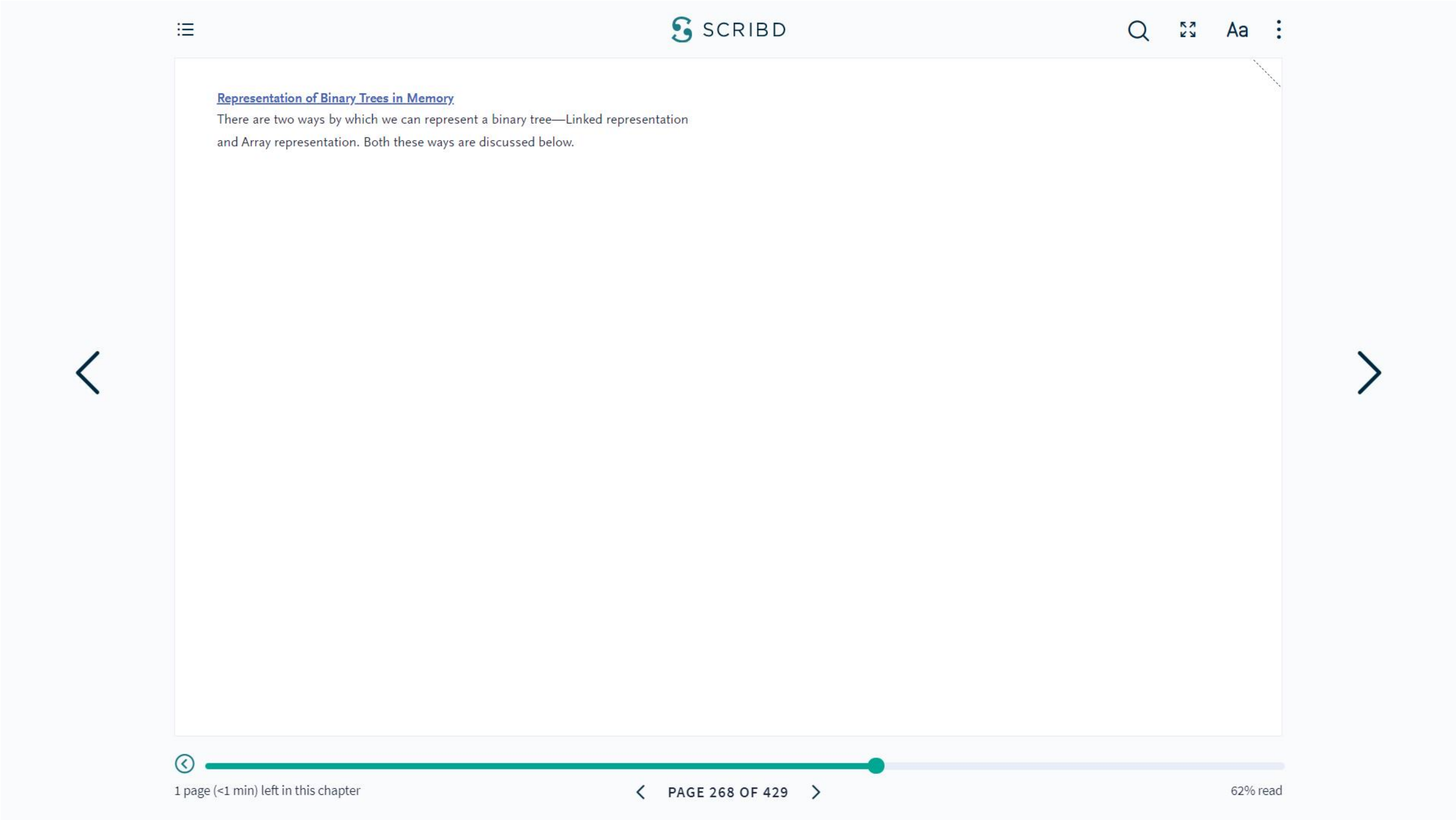
Figure 7-3(a) is a strictly binary tree.

Complete binary tree : A complete binary tree (refer Figure 7-3(b)) has maximum number of possible nodes at all levels except the last level, and all the nodes of the last level appear as far left as possible.

(a) Strictly Binary Tree

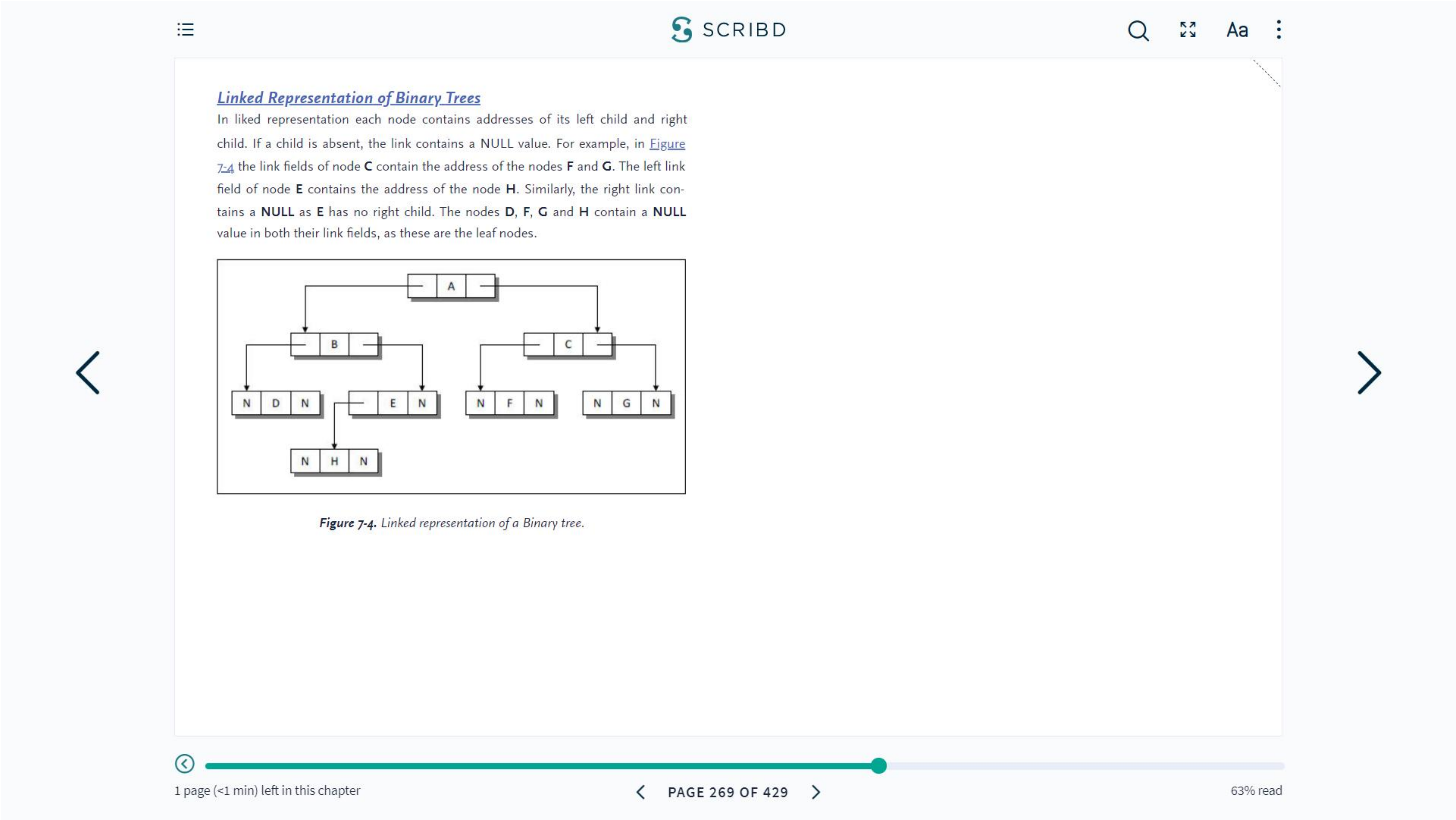
(b) Complete Binary Tree

Figure 7-3. *Strictly and Complete binary tree.*



Representation of Binary Trees in Memory

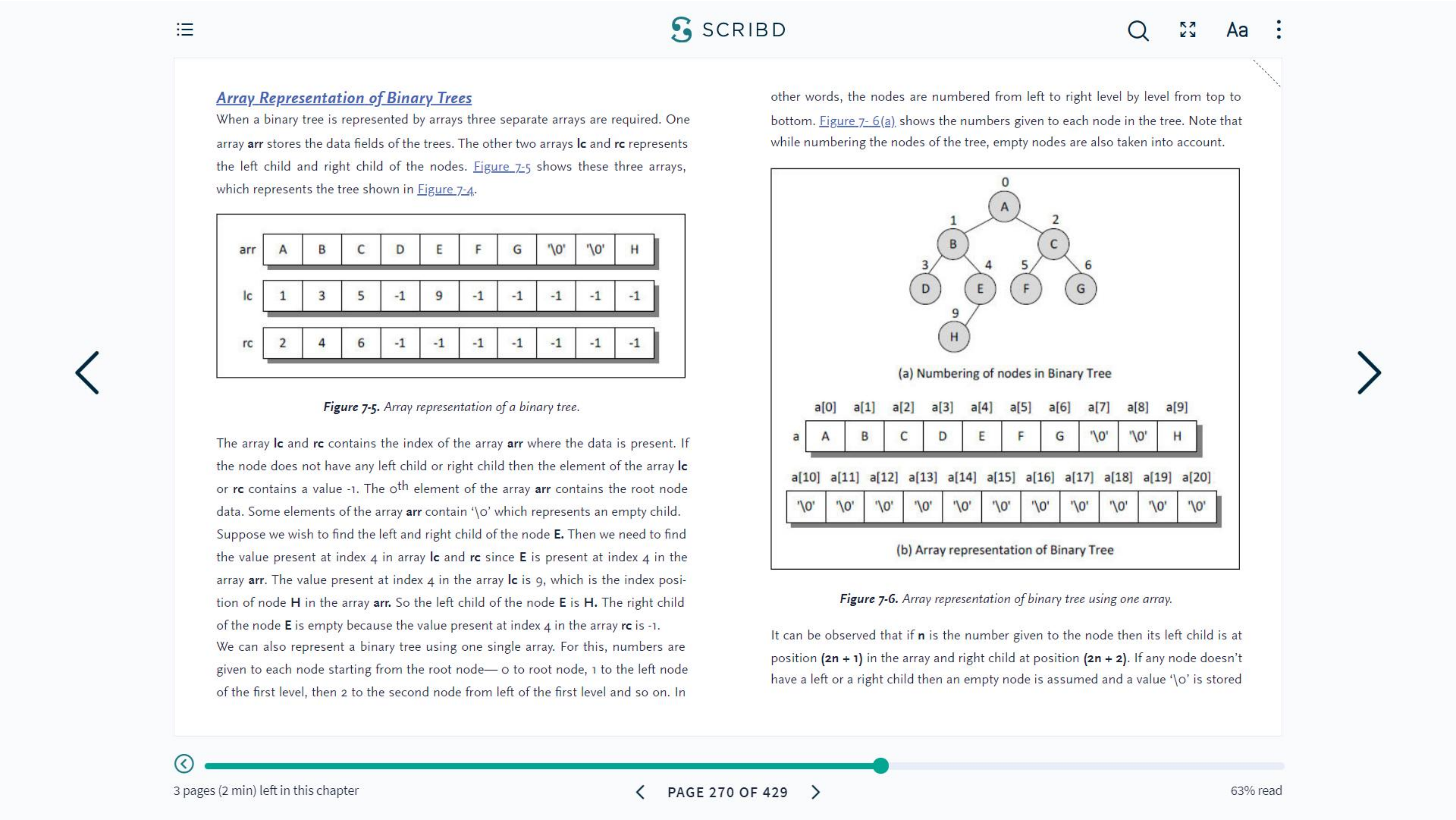
There are two ways by which we can represent a binary tree—Linked representation and Array representation. Both these ways are discussed below.



Linked Representation of Binary Trees

In liked representation each node contains addresses of its left child and right child. If a child is absent, the link contains a NULL value. For example, in Figure 7-4 the link fields of node **C** contain the address of the nodes **F** and **G**. The left link field of node **E** contains the address of the node **H**. Similarly, the right link contains a **NULL** as **E** has no right child. The nodes **D**, **F**, **G** and **H** contain a **NULL** value in both their link fields, as these are the leaf nodes.

Figure 7-4. *Linked representation of a Binary tree.*



Array Representation of Binary Trees

When a binary tree is represented by arrays three separate arrays are required. One array **arr** stores the data fields of the trees. The other two arrays **lc** and **rc** represents the left child and right child of the nodes. Figure 7-5 shows these three arrays, which represents the tree shown in Figure 7-4.

arr	A	B	C	D	E	F	G	'\0'	'\0'	H
lc	1	3	5	-1	9	-1	-1	-1	-1	-1
rc	2	4	6	-1	-1	-1	-1	-1	-1	-1

Figure 7-5. *Array representation of a binary tree.*

The array **lc** and **rc** contains the index of the array **arr** where the data is present. If the node does not have any left child or right child then the element of the array **lc** or **rc** contains a value -1. The 0^{th} element of the array **arr** contains the root node data. Some elements of the array **arr** contain '\0' which represents an empty child. Suppose we wish to find the left and right child of the node **E.** Then we need to find the value present at index 4 in array **lc** and **rc** since **E** is present at index 4 in the array **arr**. The value present at index 4 in the array **lc** is 9, which is the index position of node **H** in the array **arr.** So the left child of the node **E** is **H.** The right child of the node **E** is empty because the value present at index 4 in the array **rc** is -1.

We can also represent a binary tree using one single array. For this, numbers are given to each node starting from the root node— 0 to root node, 1 to the left node of the first level, then 2 to the second node from left of the first level and so on. In other words, the nodes are numbered from left to right level by level from top to bottom. Figure 7- 6(a) shows the numbers given to each node in the tree. Note that while numbering the nodes of the tree, empty nodes are also taken into account.

(a) Numbering of nodes in Binary Tree

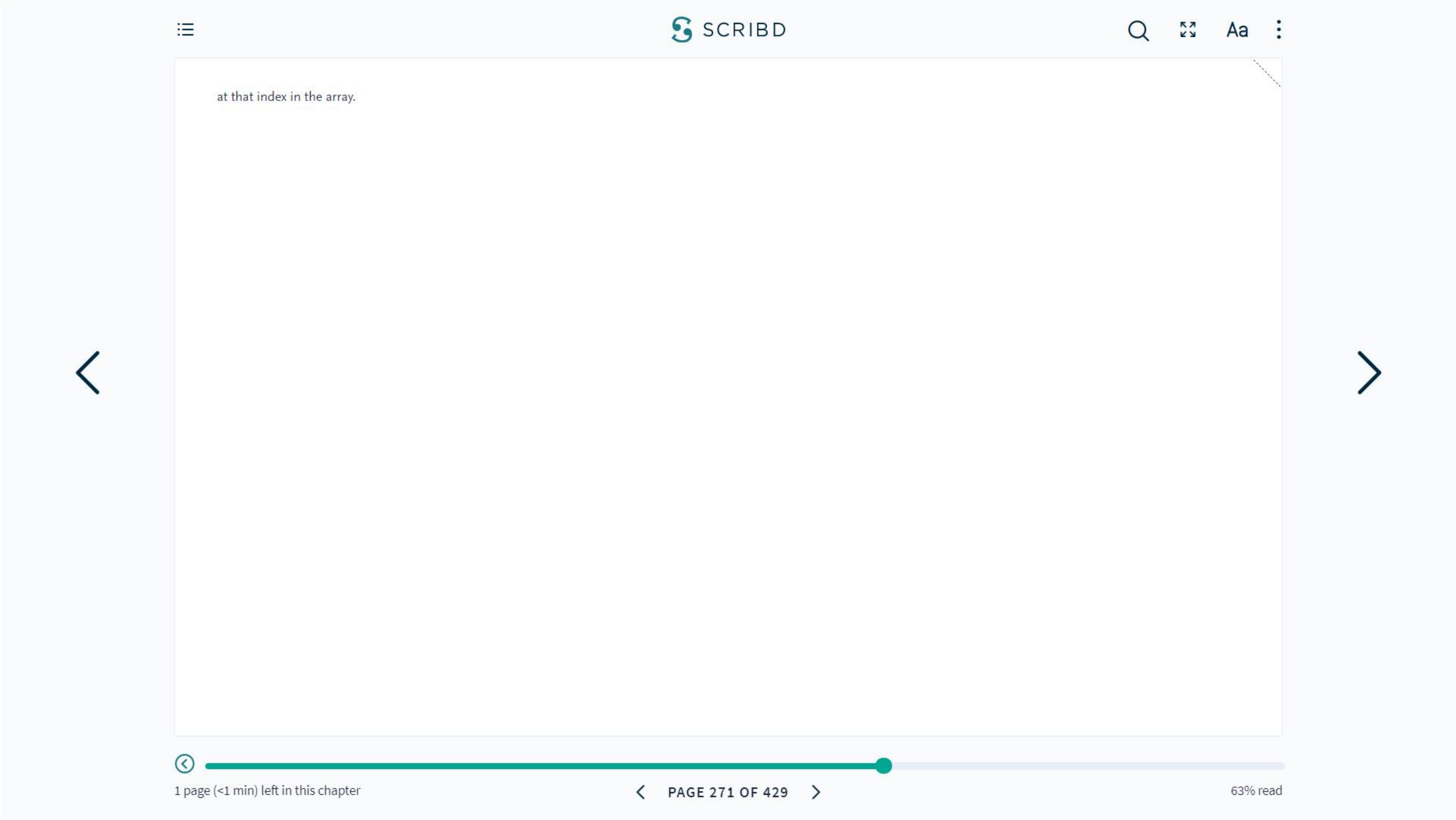
	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a	A	B	C	D	E	F	G	'\0'	'\0'	H

a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]	a[20]
'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'

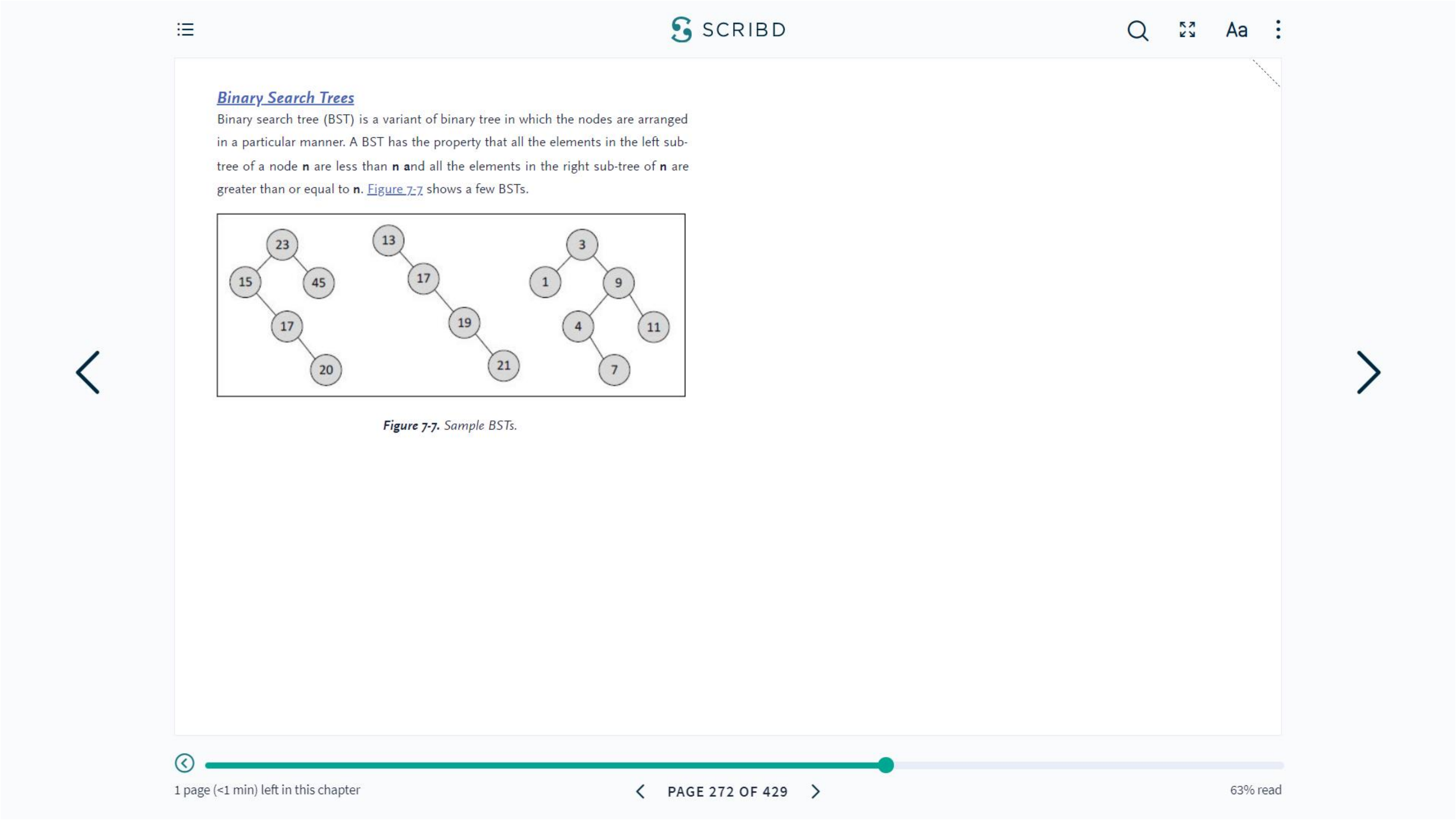
(b) Array representation of Binary Tree

Figure 7-6. *Array representation of binary tree using one array.*

It can be observed that if **n** is the number given to the node then its left child is at position **(2n + 1)** in the array and right child at position **(2n + 2)**. If any node doesn't have a left or a right child then an empty node is assumed and a value '\0' is stored



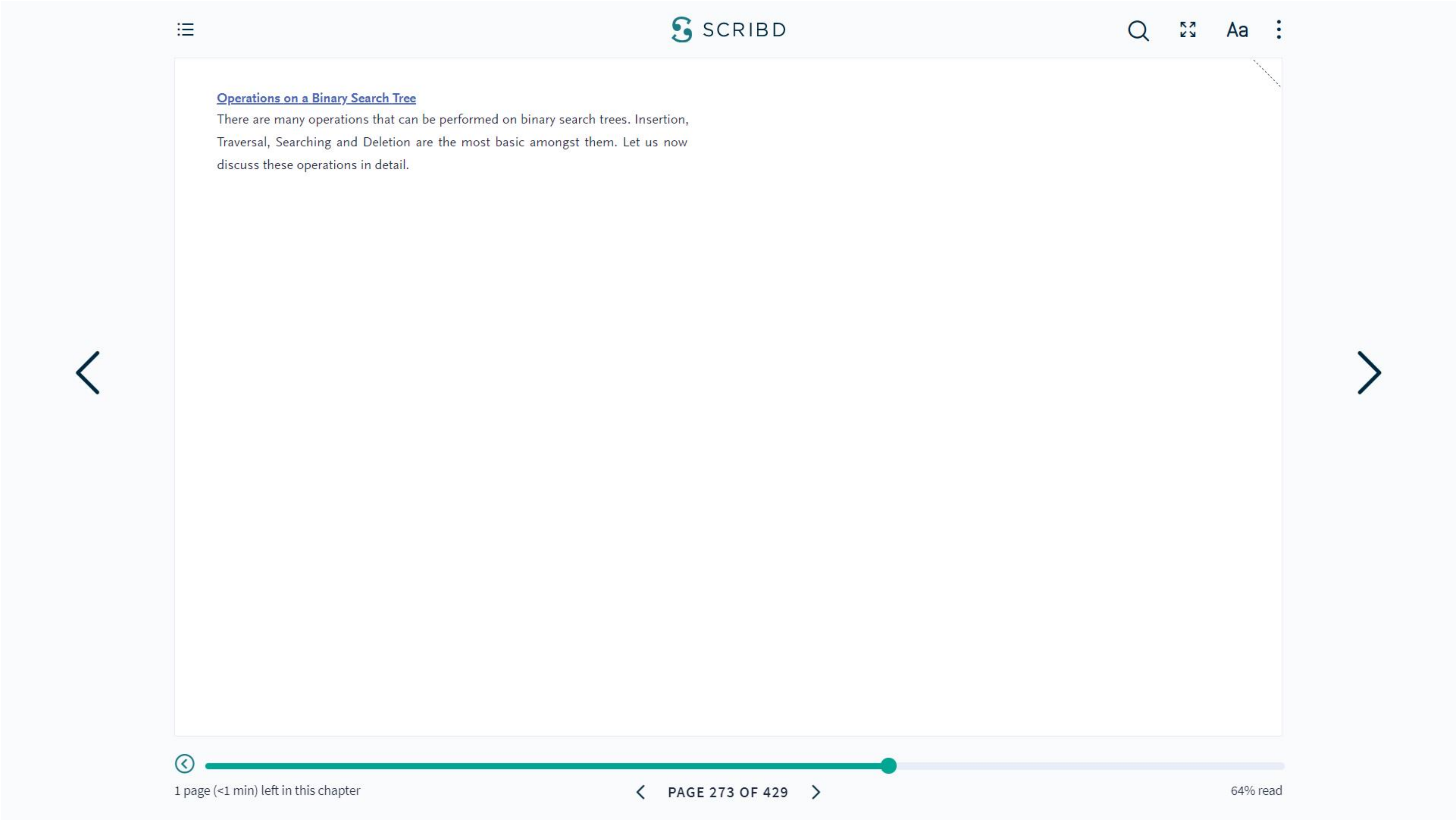
at that index in the array.



Binary Search Trees

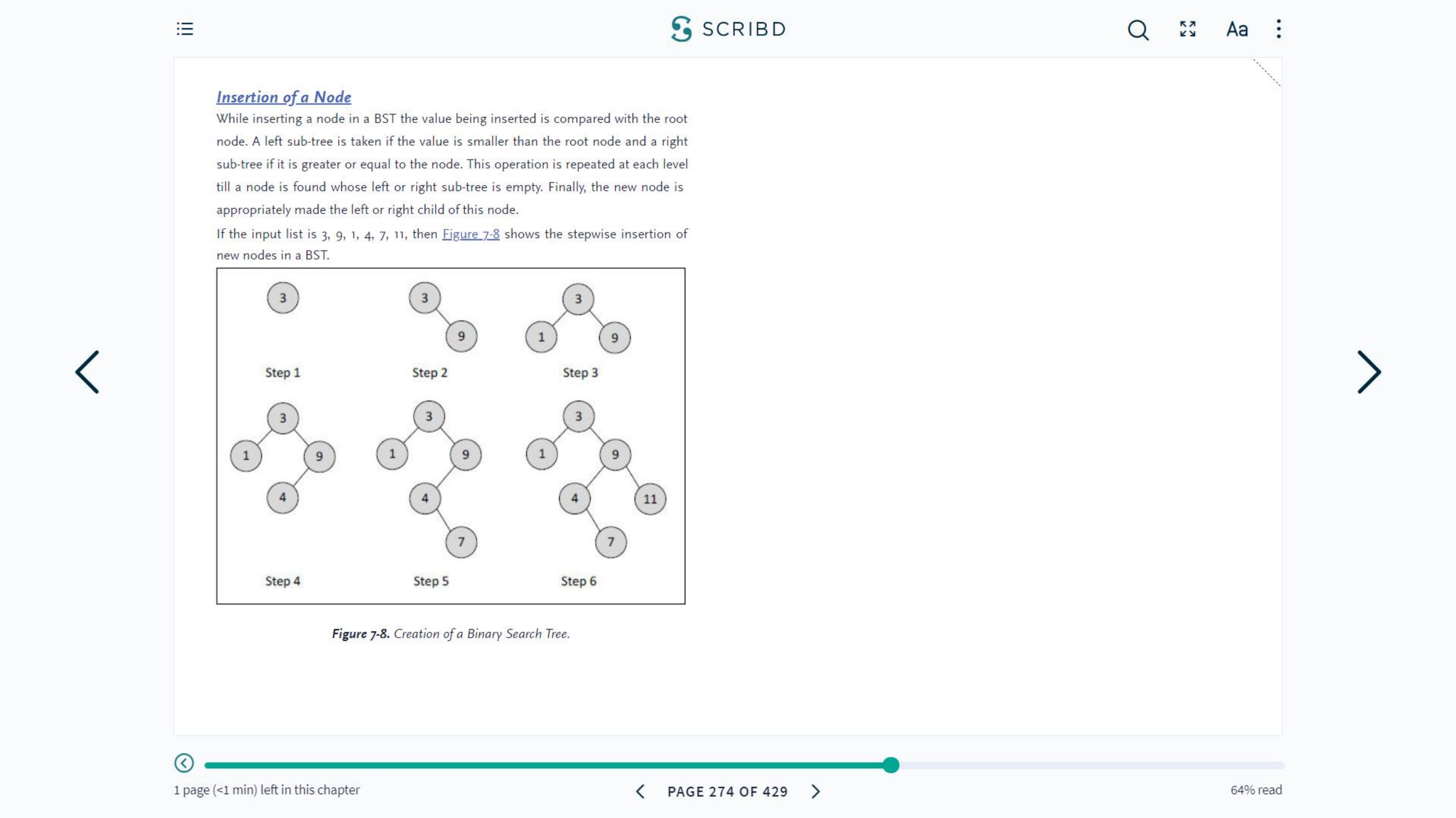
Binary search tree (BST) is a variant of binary tree in which the nodes are arranged in a particular manner. A BST has the property that all the elements in the left sub-tree of a node **n** are less than **n a**nd all the elements in the right sub-tree of **n** are greater than or equal to **n**. Figure 7-7 shows a few BSTs.

Figure 7-7. *Sample BSTs.*



Operations on a Binary Search Tree

There are many operations that can be performed on binary search trees. Insertion, Traversal, Searching and Deletion are the most basic amongst them. Let us now discuss these operations in detail.

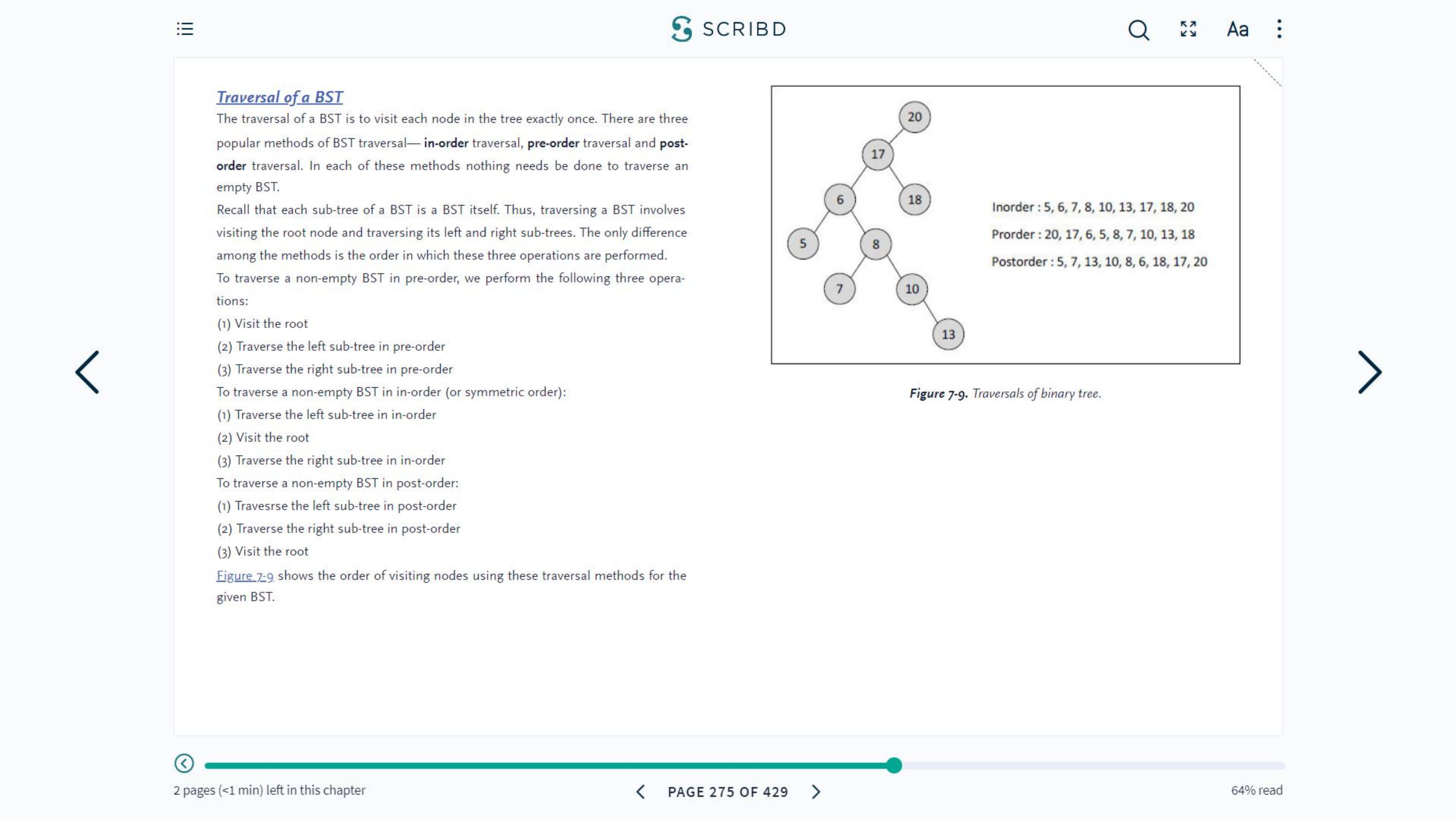


Insertion of a Node

While inserting a node in a BST the value being inserted is compared with the root node. A left sub-tree is taken if the value is smaller than the root node and a right sub-tree if it is greater or equal to the node. This operation is repeated at each level till a node is found whose left or right sub-tree is empty. Finally, the new node is appropriately made the left or right child of this node.

If the input list is 3, 9, 1, 4, 7, 11, then Figure 7-8 shows the stepwise insertion of new nodes in a BST.

Figure 7-8. *Creation of a Binary Search Tree.*



Traversal of a BST

The traversal of a BST is to visit each node in the tree exactly once. There are three popular methods of BST traversal— **in-order** traversal, **pre-order** traversal and **post-order** traversal. In each of these methods nothing needs be done to traverse an empty BST.

Recall that each sub-tree of a BST is a BST itself. Thus, traversing a BST involves visiting the root node and traversing its left and right sub-trees. The only difference among the methods is the order in which these three operations are performed.

To traverse a non-empty BST in pre-order, we perform the following three operations:

(1) Visit the root

(2) Traverse the left sub-tree in pre-order

(3) Traverse the right sub-tree in pre-order

To traverse a non-empty BST in in-order (or symmetric order):

(1) Traverse the left sub-tree in in-order

(2) Visit the root

(3) Traverse the right sub-tree in in-order

To traverse a non-empty BST in post-order:

(1) Travesrse the left sub-tree in post-order

(2) Traverse the right sub-tree in post-order

(3) Visit the root

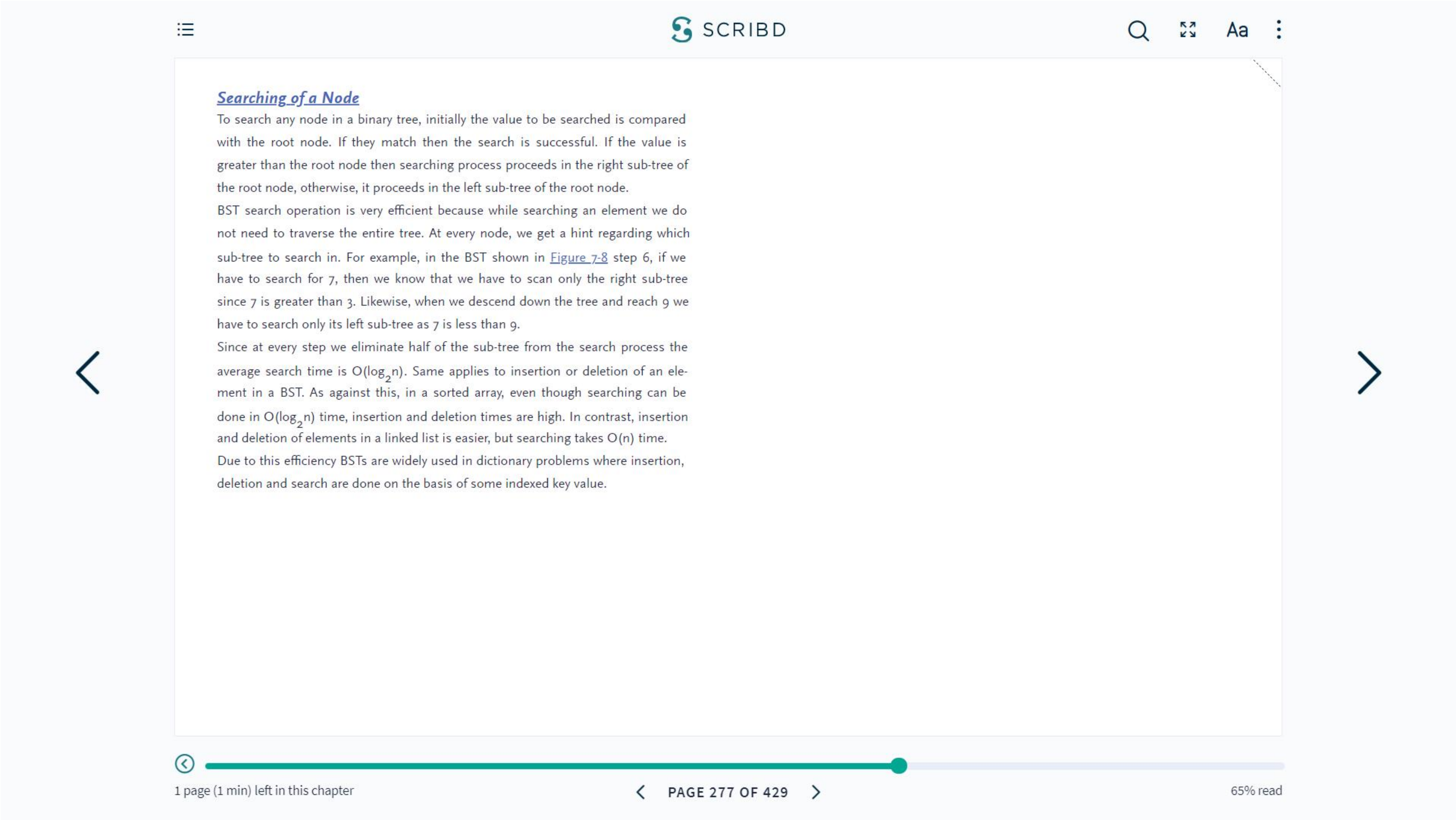
Figure 7-9 shows the order of visiting nodes using these traversal methods for the given BST.

Inorder : 5, 6, 7, 8, 10, 13, 17, 18, 20

Prorder : 20, 17, 6, 5, 8, 7, 10, 13, 18

Postorder : 5, 7, 13, 10, 8, 6, 18, 17, 20

Figure 7-9. *Traversals of binary tree.*



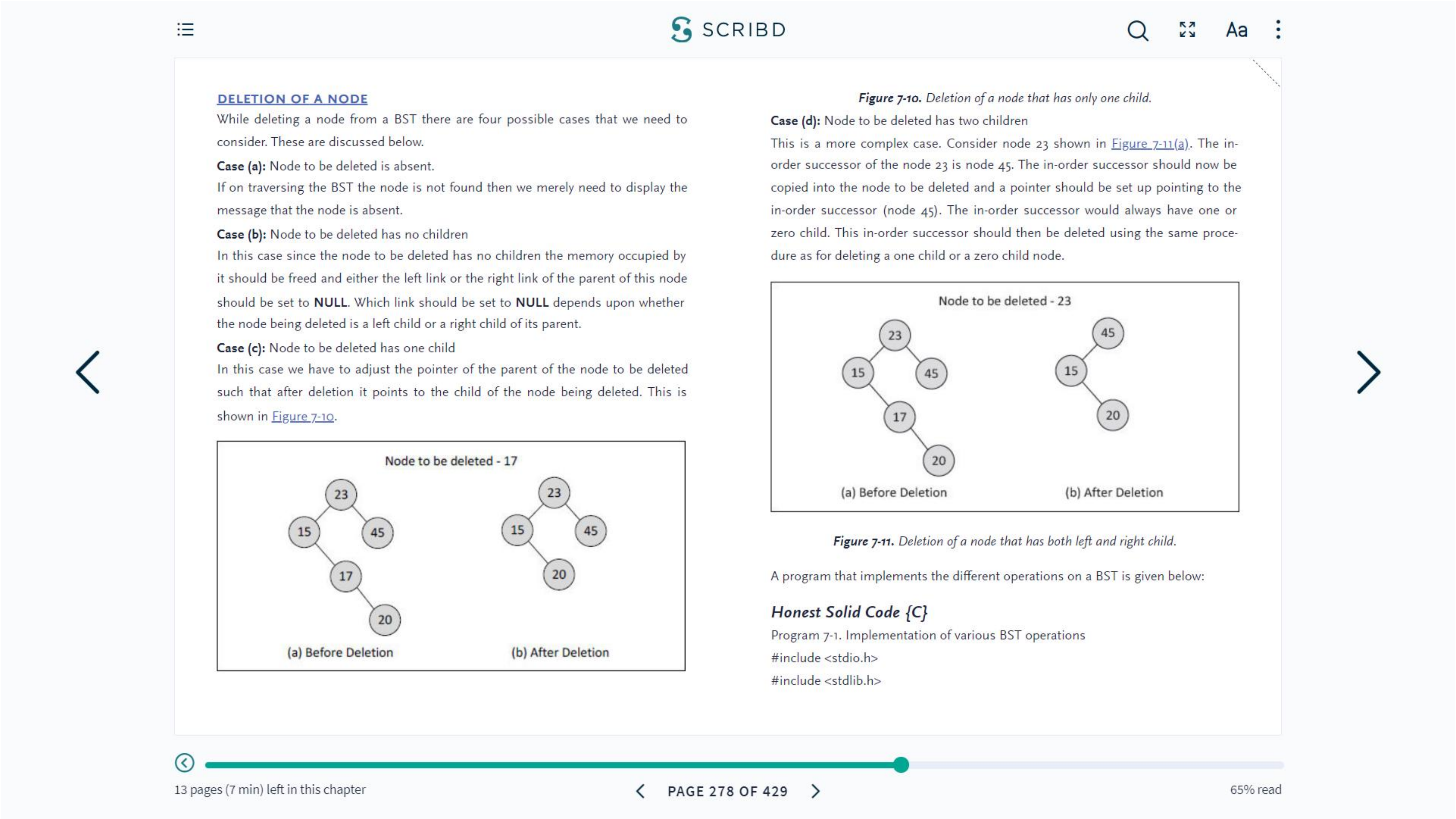
Searching of a Node

To search any node in a binary tree, initially the value to be searched is compared with the root node. If they match then the search is successful. If the value is greater than the root node then searching process proceeds in the right sub-tree of the root node, otherwise, it proceeds in the left sub-tree of the root node.

BST search operation is very efficient because while searching an element we do not need to traverse the entire tree. At every node, we get a hint regarding which sub-tree to search in. For example, in the BST shown in Figure 7-8 step 6, if we have to search for 7, then we know that we have to scan only the right sub-tree since 7 is greater than 3. Likewise, when we descend down the tree and reach 9 we have to search only its left sub-tree as 7 is less than 9.

Since at every step we eliminate half of the sub-tree from the search process the average search time is $O(\log_2 n)$. Same applies to insertion or deletion of an element in a BST. As against this, in a sorted array, even though searching can be done in $O(\log_2 n)$ time, insertion and deletion times are high. In contrast, insertion and deletion of elements in a linked list is easier, but searching takes $O(n)$ time.

Due to this efficiency BSTs are widely used in dictionary problems where insertion, deletion and search are done on the basis of some indexed key value.



DELETION OF A NODE

While deleting a node from a BST there are four possible cases that we need to consider. These are discussed below.

Case (a): Node to be deleted is absent.

If on traversing the BST the node is not found then we merely need to display the message that the node is absent.

Case (b): Node to be deleted has no children

In this case since the node to be deleted has no children the memory occupied by it should be freed and either the left link or the right link of the parent of this node should be set to **NULL**. Which link should be set to **NULL** depends upon whether the node being deleted is a left child or a right child of its parent.

Case (c): Node to be deleted has one child

In this case we have to adjust the pointer of the parent of the node to be deleted such that after deletion it points to the child of the node being deleted. This is shown in Figure 7-10.

Figure 7-10. *Deletion of a node that has only one child.*

Case (d): Node to be deleted has two children

This is a more complex case. Consider node 23 shown in Figure 7-11(a). The in-order successor of the node 23 is node 45. The in-order successor should now be copied into the node to be deleted and a pointer should be set up pointing to the in-order successor (node 45). The in-order successor would always have one or zero child. This in-order successor should then be deleted using the same procedure as for deleting a one child or a zero child node.

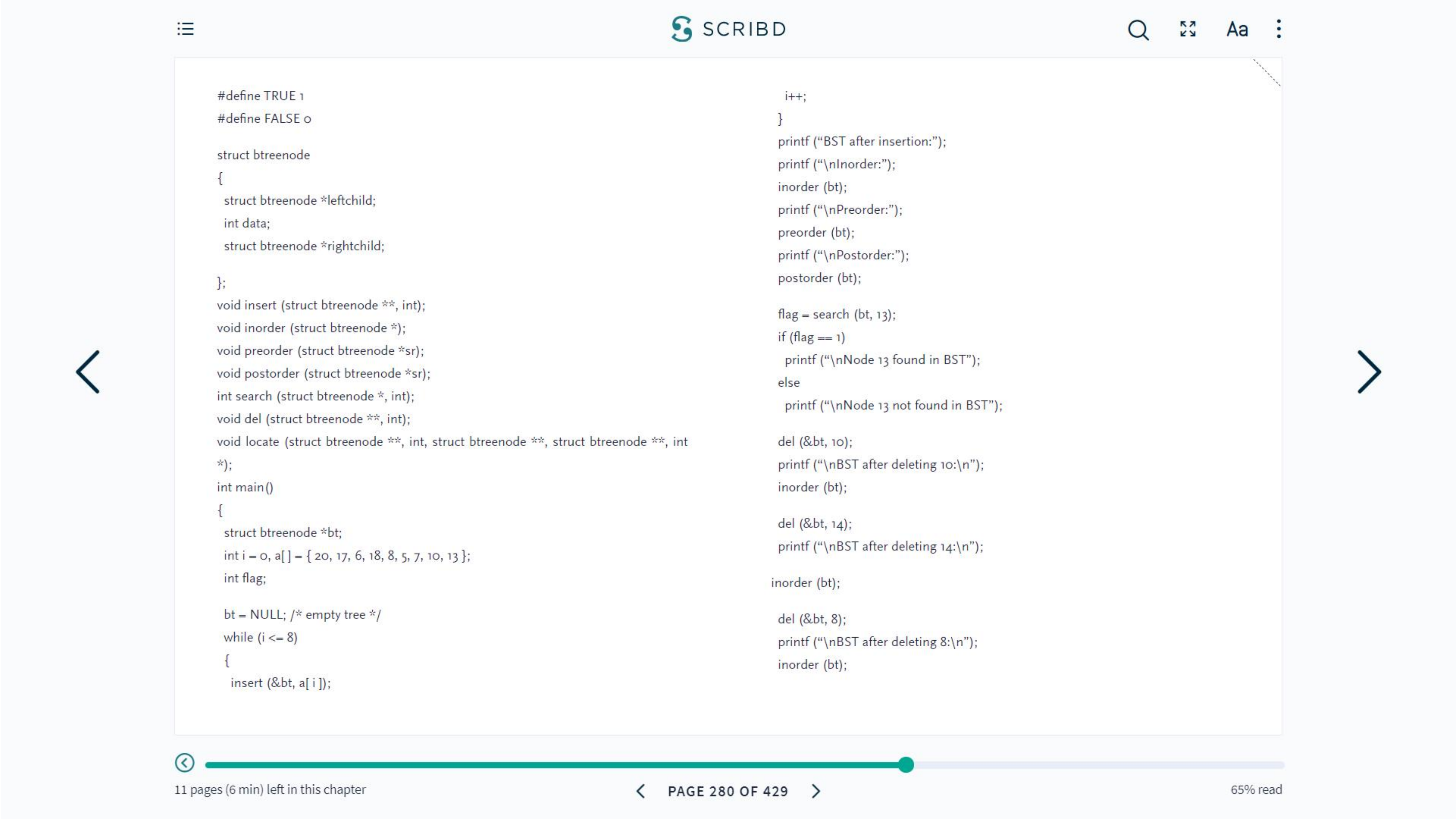
Figure 7-11. *Deletion of a node that has both left and right child.*

A program that implements the different operations on a BST is given below:

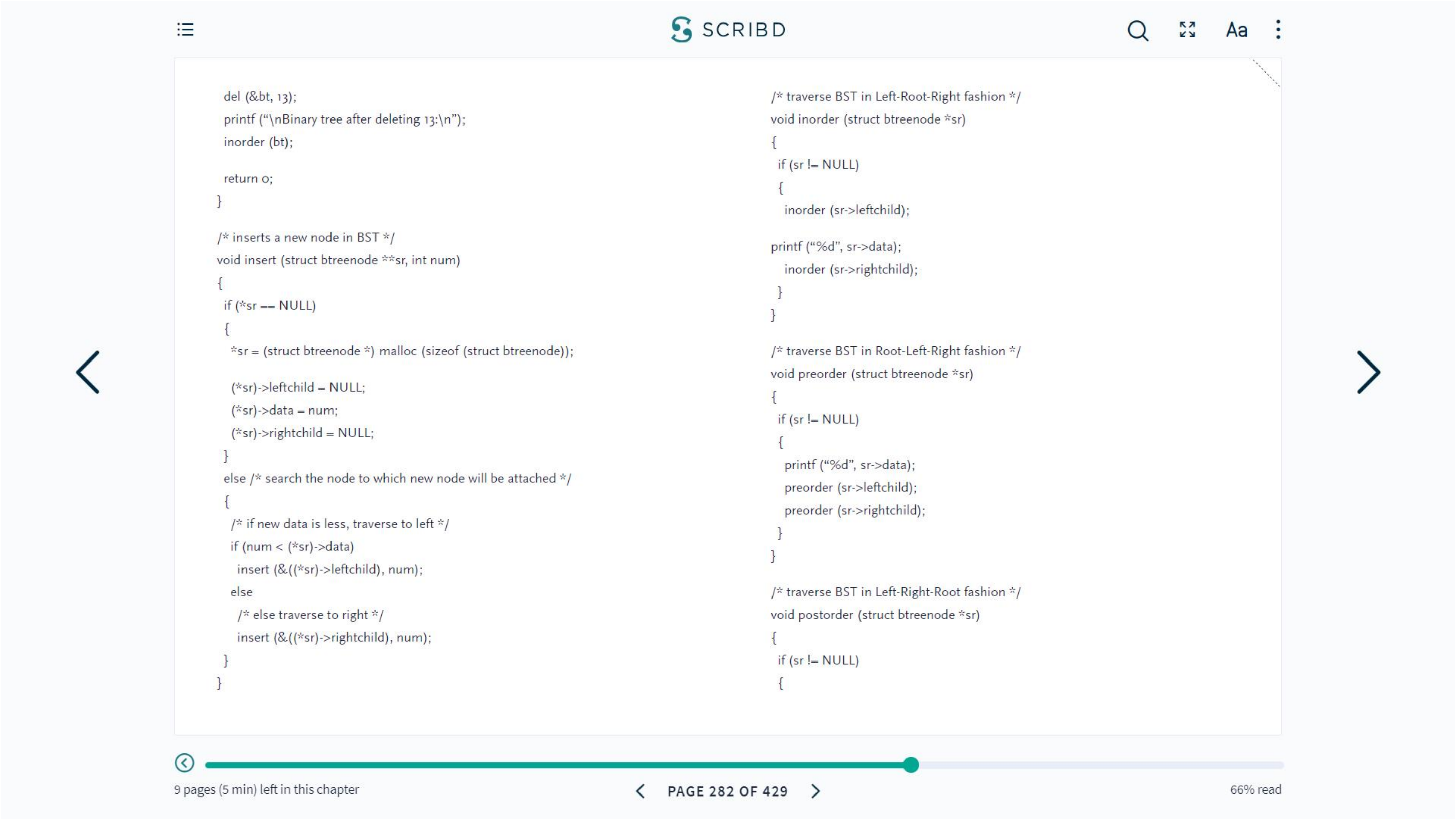
Honest Solid Code {C}

Program 7-1. Implementation of various BST operations

```
#include <stdio.h>
#include <stdlib.h>
```

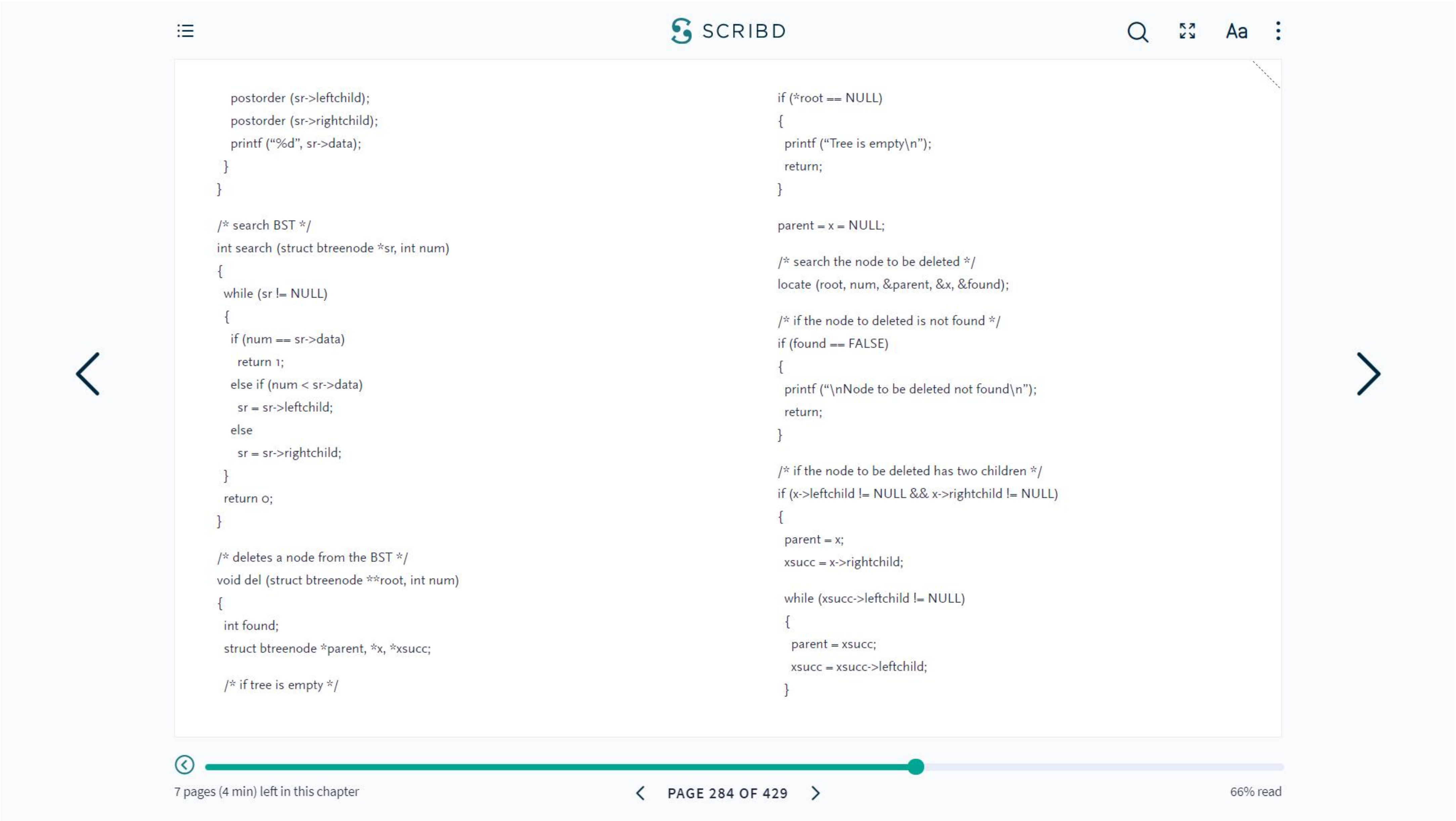


```
#define TRUE 1
#define FALSE 0

struct btreenode
{
 struct btreenode *leftchild;
 int data;
 struct btreenode *rightchild;

};
void insert (struct btreenode **, int);
void inorder (struct btreenode *);
void preorder (struct btreenode *sr);
void postorder (struct btreenode *sr);
int search (struct btreenode *, int);
void del (struct btreenode **, int);
void locate (struct btreenode **, int, struct btreenode **, struct btreenode **, int
*);
int main()
{
 struct btreenode *bt;
 int i = 0, a[ ] = { 20, 17, 6, 18, 8, 5, 7, 10, 13 };
 int flag;

 bt = NULL; /* empty tree */
 while (i <= 8)
 {
  insert (&bt, a[ i ]);
  i++;
 }
 printf (“BST after insertion:”);
 printf (“\nInorder:”);
 inorder (bt);
 printf (“\nPreorder:”);
 preorder (bt);
 printf (“\nPostorder:”);
 postorder (bt);

 flag = search (bt, 13);
 if (flag == 1)
  printf (“\nNode 13 found in BST”);
 else
  printf (“\nNode 13 not found in BST”);

 del (&bt, 10);
 printf (“\nBST after deleting 10:\n”);
 inorder (bt);

 del (&bt, 14);
 printf (“\nBST after deleting 14:\n”);

inorder (bt);

 del (&bt, 8);
 printf (“\nBST after deleting 8:\n”);
 inorder (bt);
```

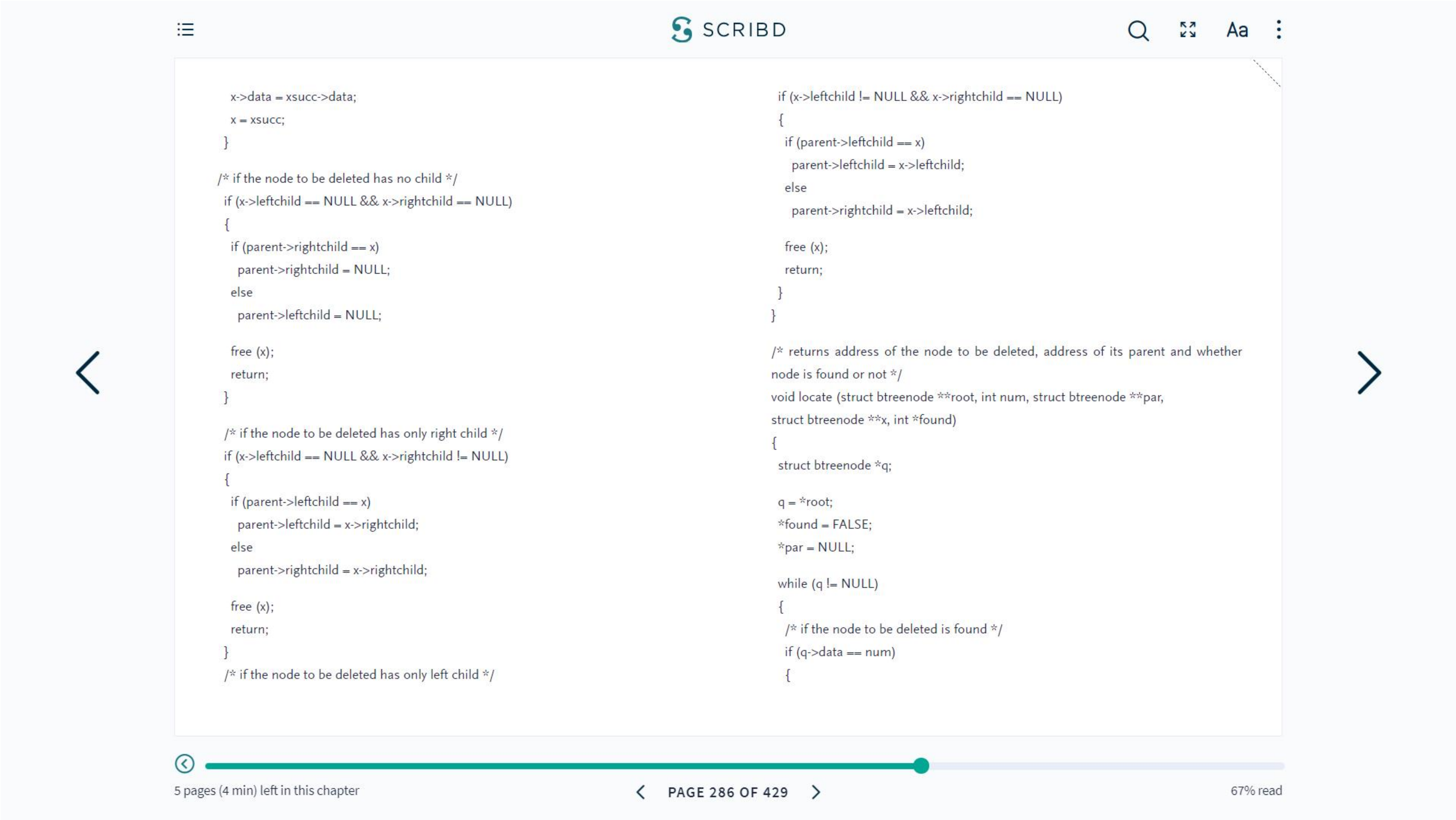


```
 del (&bt, 13);
 printf (“\nBinary tree after deleting 13:\n”);
 inorder (bt);

 return 0;
}

/* inserts a new node in BST */
void insert (struct btreenode **sr, int num)
{
 if (*sr == NULL)
 {
  *sr = (struct btreenode *) malloc (sizeof (struct btreenode));

  (*sr)->leftchild = NULL;
  (*sr)->data = num;
  (*sr)->rightchild = NULL;
 }
 else /* search the node to which new node will be attached */
 {
  /* if new data is less, traverse to left */
  if (num < (*sr)->data)
   insert (&((*sr)->leftchild), num);
  else
   /* else traverse to right */
   insert (&((*sr)->rightchild), num);
 }
}
```

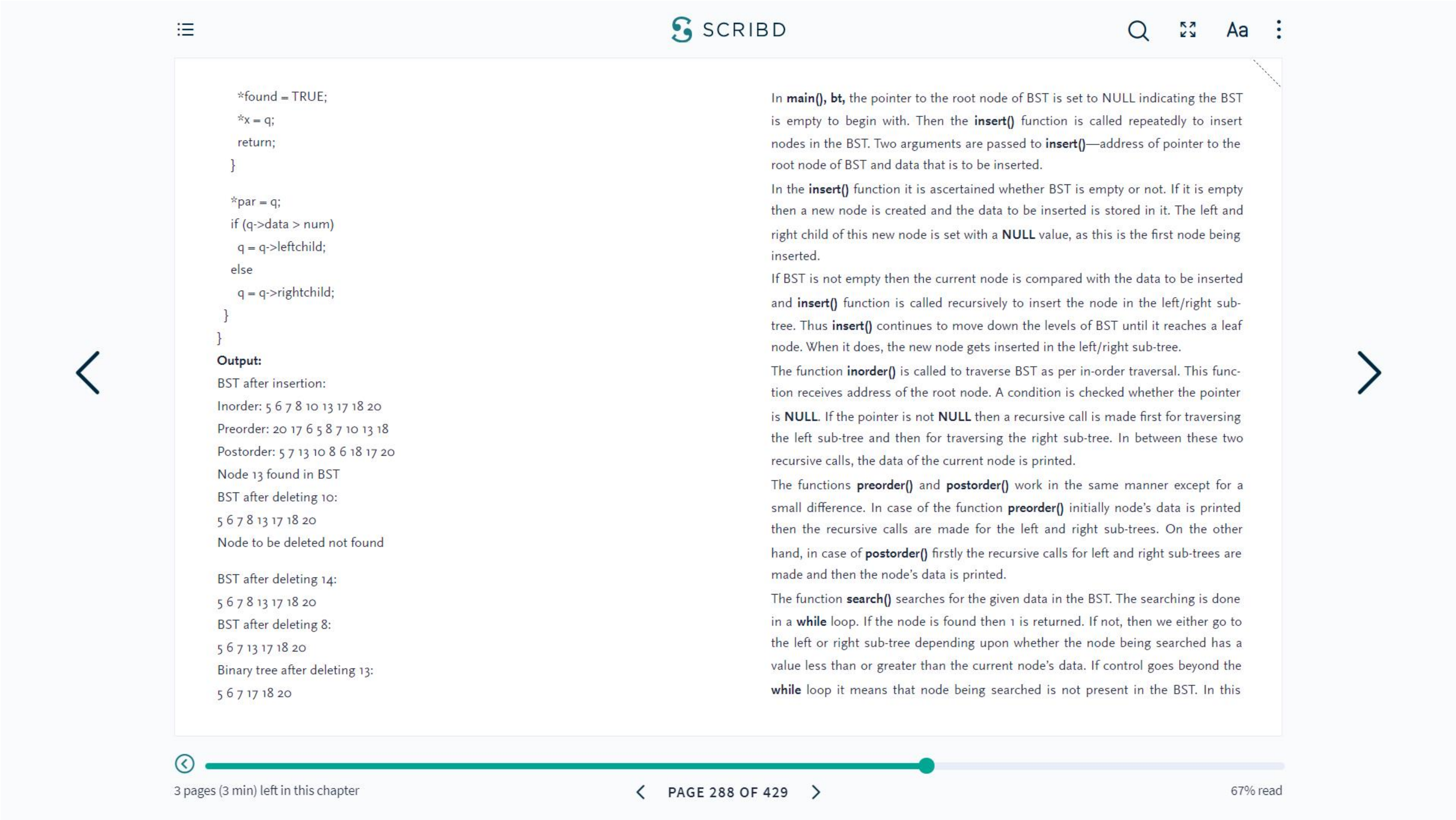
```
/* traverse BST in Left-Root-Right fashion */
void inorder (struct btreenode *sr)
{
 if (sr != NULL)
 {
  inorder (sr->leftchild);

printf (“%d”, sr->data);
  inorder (sr->rightchild);
 }
}

/* traverse BST in Root-Left-Right fashion */
void preorder (struct btreenode *sr)
{
 if (sr != NULL)
 {
  printf (“%d”, sr->data);
  preorder (sr->leftchild);
  preorder (sr->rightchild);
 }
}

/* traverse BST in Left-Right-Root fashion */
void postorder (struct btreenode *sr)
{
 if (sr != NULL)
 {
```



```
    postorder (sr->leftchild);
    postorder (sr->rightchild);
    printf (“%d”, sr->data);
  }
}

/* search BST */
int search (struct btreenode *sr, int num)
{
  while (sr != NULL)
  {
    if (num == sr->data)
      return 1;
    else if (num < sr->data)
      sr = sr->leftchild;
    else
      sr = sr->rightchild;
  }
  return 0;
}

/* deletes a node from the BST */
void del (struct btreenode **root, int num)
{
  int found;
  struct btreenode *parent, *x, *xsucc;

  /* if tree is empty */
  if (*root == NULL)
  {
    printf (“Tree is empty\n”);
    return;
  }

  parent = x = NULL;

  /* search the node to be deleted */
  locate (root, num, &parent, &x, &found);

  /* if the node to deleted is not found */
  if (found == FALSE)
  {
    printf (“\nNode to be deleted not found\n”);
    return;
  }

  /* if the node to be deleted has two children */
  if (x->leftchild != NULL && x->rightchild != NULL)
  {
    parent = x;
    xsucc = x->rightchild;

    while (xsucc->leftchild != NULL)
    {
      parent = xsucc;
      xsucc = xsucc->leftchild;
    }
```



```
  x->data = xsucc->data;
  x = xsucc;
 }

/* if the node to be deleted has no child */
 if (x->leftchild == NULL && x->rightchild == NULL)
 {
  if (parent->rightchild == x)
   parent->rightchild = NULL;
  else
   parent->leftchild = NULL;

  free (x);
  return;
 }

 /* if the node to be deleted has only right child */
 if (x->leftchild == NULL && x->rightchild != NULL)
 {
  if (parent->leftchild == x)
   parent->leftchild = x->rightchild;
  else
   parent->rightchild = x->rightchild;

  free (x);
  return;
 }
 /* if the node to be deleted has only left child */
 if (x->leftchild != NULL && x->rightchild == NULL)
 {
  if (parent->leftchild == x)
   parent->leftchild = x->leftchild;
  else
   parent->rightchild = x->leftchild;

  free (x);
  return;
 }
}

/* returns address of the node to be deleted, address of its parent and whether
node is found or not */
void locate (struct btreenode **root, int num, struct btreenode **par,
struct btreenode **x, int *found)
{
 struct btreenode *q;

 q = *root;
 *found = FALSE;
 *par = NULL;

 while (q != NULL)
 {
  /* if the node to be deleted is found */
  if (q->data == num)
  {
```



```
            *found = TRUE;
            *x = q;
            return;
        }

        *par = q;
        if (q->data > num)
            q = q->leftchild;
        else
            q = q->rightchild;
    }
}
```

Output:

```
BST after insertion:
Inorder: 5 6 7 8 10 13 17 18 20
Preorder: 20 17 6 5 8 7 10 13 18
Postorder: 5 7 13 10 8 6 18 17 20
Node 13 found in BST
BST after deleting 10:
5 6 7 8 13 17 18 20
Node to be deleted not found

BST after deleting 14:
5 6 7 8 13 17 18 20
BST after deleting 8:
5 6 7 13 17 18 20
Binary tree after deleting 13:
5 6 7 17 18 20
```

In **main(), bt,** the pointer to the root node of BST is set to NULL indicating the BST is empty to begin with. Then the **insert()** function is called repeatedly to insert nodes in the BST. Two arguments are passed to **insert()**—address of pointer to the root node of BST and data that is to be inserted.

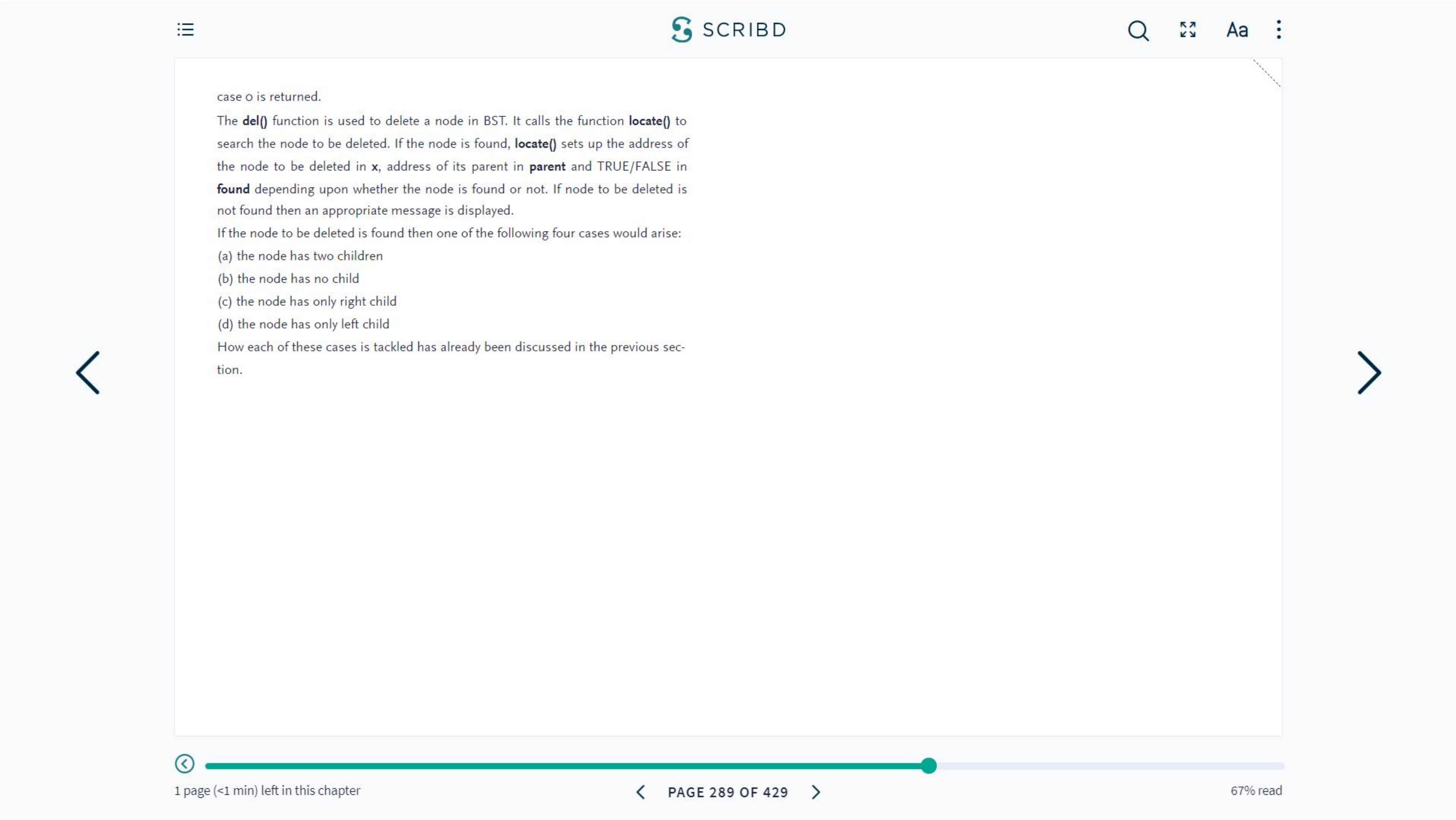
In the **insert()** function it is ascertained whether BST is empty or not. If it is empty then a new node is created and the data to be inserted is stored in it. The left and right child of this new node is set with a **NULL** value, as this is the first node being inserted.

If BST is not empty then the current node is compared with the data to be inserted and **insert()** function is called recursively to insert the node in the left/right sub-tree. Thus **insert()** continues to move down the levels of BST until it reaches a leaf node. When it does, the new node gets inserted in the left/right sub-tree.

The function **inorder()** is called to traverse BST as per in-order traversal. This function receives address of the root node. A condition is checked whether the pointer is **NULL**. If the pointer is not **NULL** then a recursive call is made first for traversing the left sub-tree and then for traversing the right sub-tree. In between these two recursive calls, the data of the current node is printed.

The functions **preorder()** and **postorder()** work in the same manner except for a small difference. In case of the function **preorder()** initially node's data is printed then the recursive calls are made for the left and right sub-trees. On the other hand, in case of **postorder()** firstly the recursive calls for left and right sub-trees are made and then the node's data is printed.

The function **search()** searches for the given data in the BST. The searching is done in a **while** loop. If the node is found then 1 is returned. If not, then we either go to the left or right sub-tree depending upon whether the node being searched has a value less than or greater than the current node's data. If control goes beyond the **while** loop it means that node being searched is not present in the BST. In this



case o is returned.

The **del()** function is used to delete a node in BST. It calls the function **locate()** to search the node to be deleted. If the node is found, **locate()** sets up the address of the node to be deleted in **x**, address of its parent in **parent** and TRUE/FALSE in **found** depending upon whether the node is found or not. If node to be deleted is not found then an appropriate message is displayed.

If the node to be deleted is found then one of the following four cases would arise:

(a) the node has two children

(b) the node has no child

(c) the node has only right child

(d) the node has only left child

How each of these cases is tackled has already been discussed in the previous section.

Reconstruction of a Binary Tree

If we know the sequence of nodes obtained through in-order/pre- order/post-order traversal it may not be feasible to reconstruct the binary tree. This is because two different binary trees may yield same sequence of nodes when traversed using post-order traversal. Similarly, in-order or pre-order traversal of different binary trees may yield the same sequence of nodes. However, we can construct a unique binary tree if the results of in-order and pre-order traversal are available. Let us understand this with the help of following set of in-order and pre-order traversal results:

In-order traversal: 4, 7, 2, 8, 5, 1, 6, 9, 3

Pre-order traversal: 1, 2, 4, 7, 5, 8, 3, 6, 9

We know that the first value in the pre-order traversal gives us the root of the binary tree. So the node with data 1 becomes the root of the binary tree. In in-order traversal, initially the left sub-tree is traversed then the root node and then the right sub-tree. So the data before 1 in the in-order list (i.e. 4, 7, 2, 8, 5) forms the left sub-tree and the data after 1 in the in-order list (i.e. 6, 9, 3) forms the right sub-tree. In Figure 7-12(a), the structure of tree is shown after separating the tree in left and right sub-trees.

Now look at the left sub-tree. The data in pre-order list is 2, so the root node of the left sub-tree is 2. Hence data before 2 in the in-order list (i.e. 4, 7) will form the left sub-tree of the node that contains a value 2. The data that comes to the right of 2 in the in-order list (i.e. 8, 5) forms the right sub-tree of the node with value 2. Figure 7-12(b) shows structure of tree after expanding the left and right sub-tree of the node that contains a value 2.

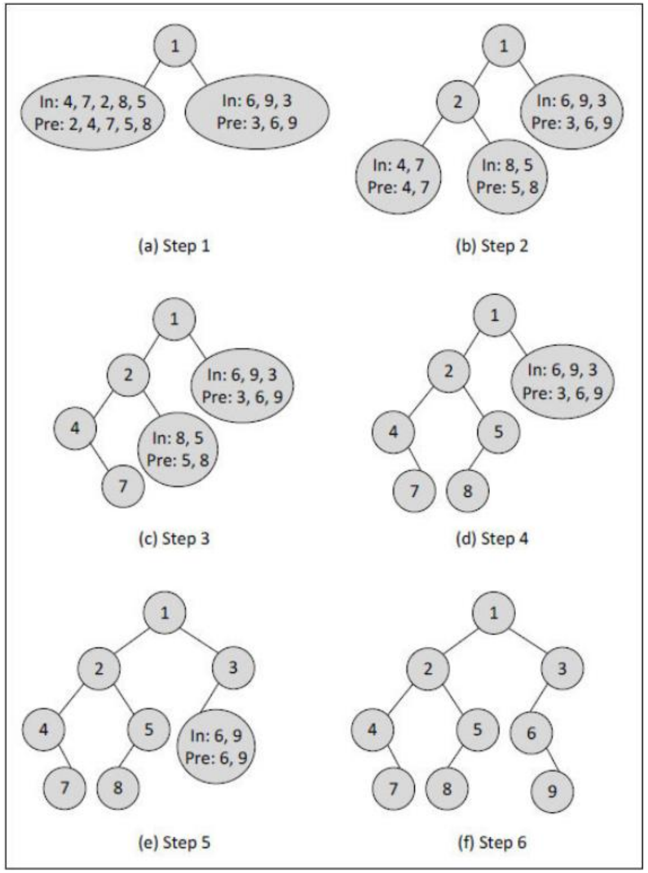
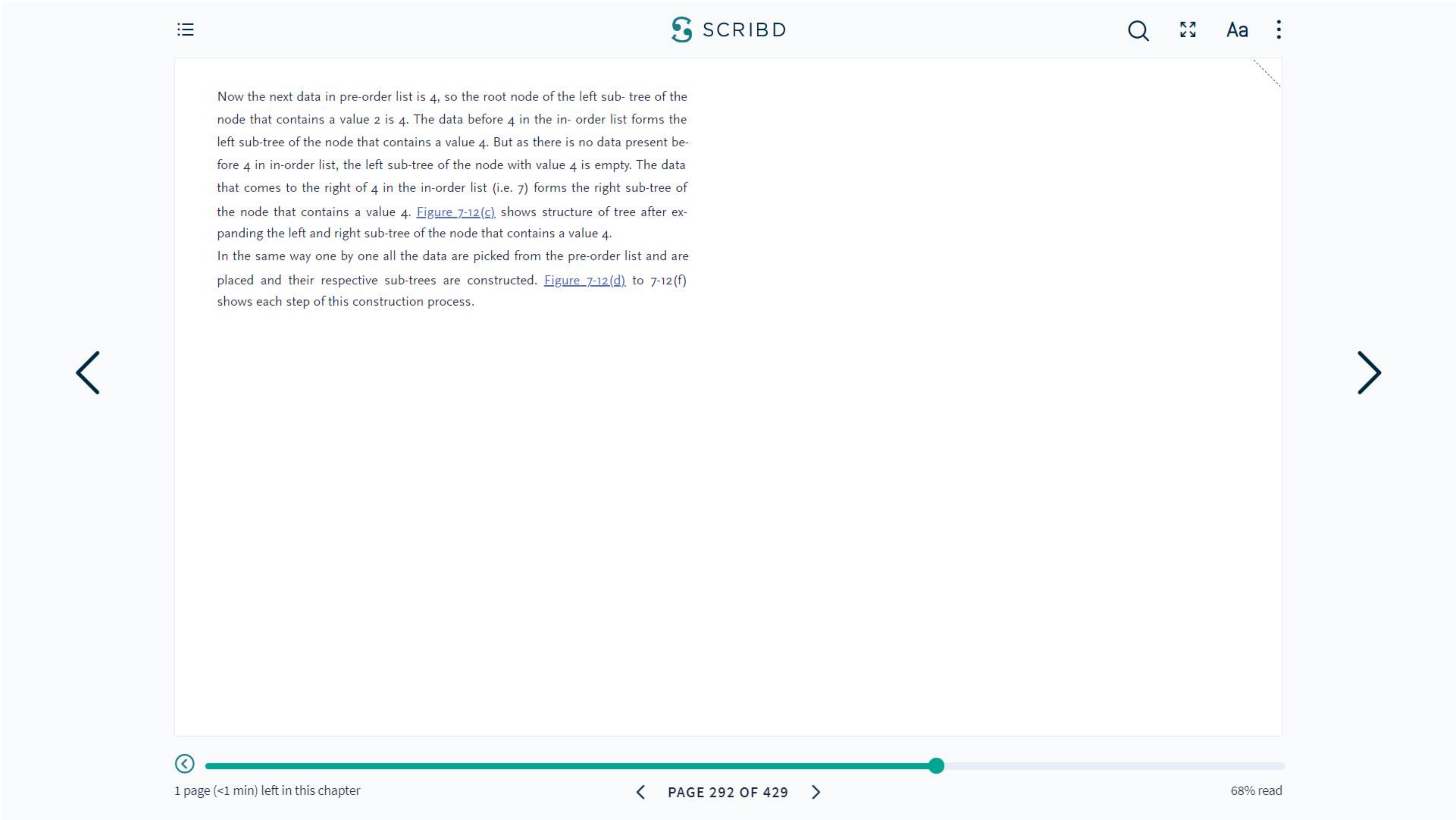
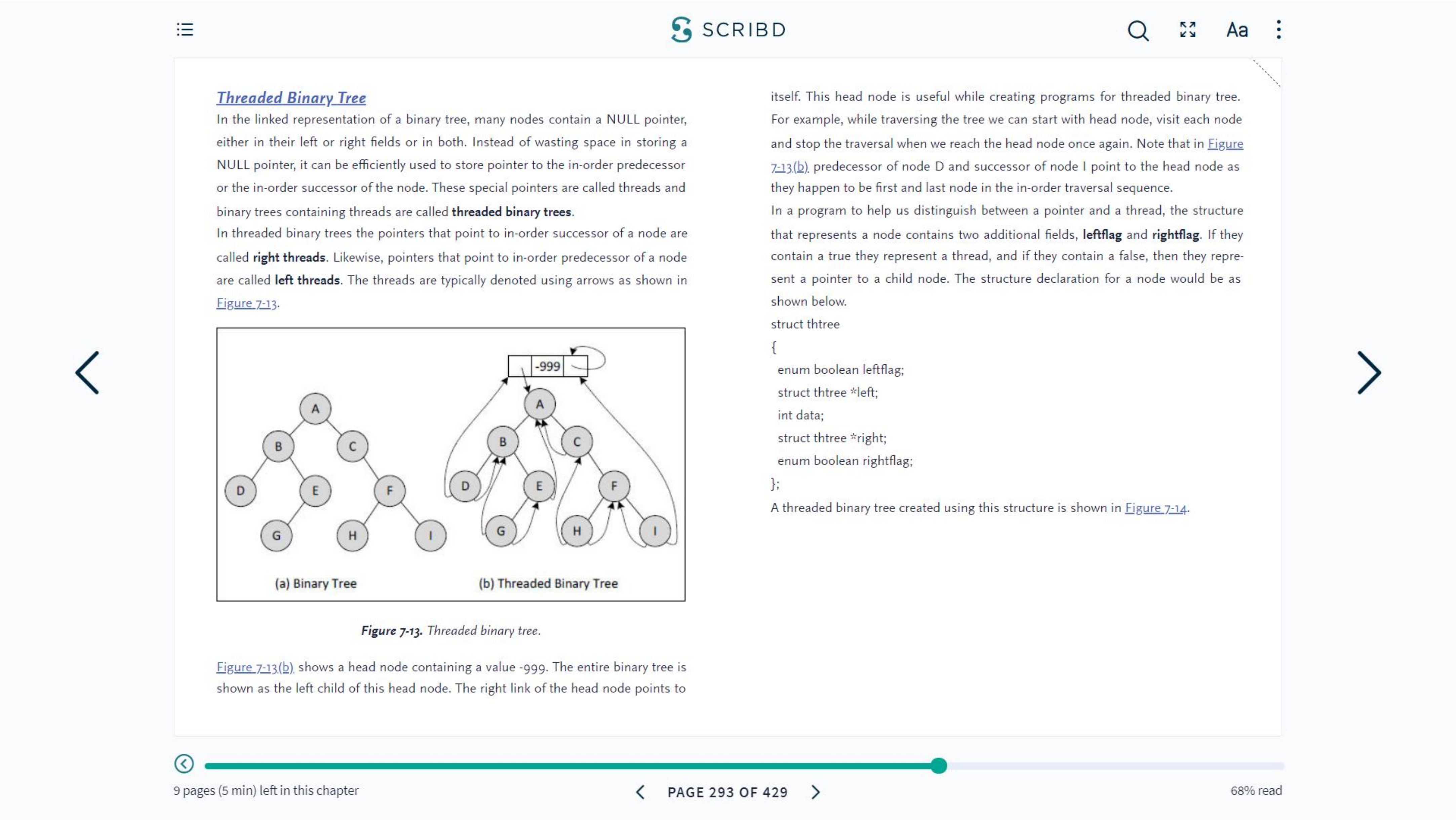


Figure 7-12. *Reconstruction of a binary tree.*



Now the next data in pre-order list is 4, so the root node of the left sub- tree of the node that contains a value 2 is 4. The data before 4 in the in- order list forms the left sub-tree of the node that contains a value 4. But as there is no data present before 4 in in-order list, the left sub-tree of the node with value 4 is empty. The data that comes to the right of 4 in the in-order list (i.e. 7) forms the right sub-tree of the node that contains a value 4. Figure 7-12(c) shows structure of tree after expanding the left and right sub-tree of the node that contains a value 4.

In the same way one by one all the data are picked from the pre-order list and are placed and their respective sub-trees are constructed. Figure 7-12(d) to 7-12(f) shows each step of this construction process.



Threaded Binary Tree

In the linked representation of a binary tree, many nodes contain a NULL pointer, either in their left or right fields or in both. Instead of wasting space in storing a NULL pointer, it can be efficiently used to store pointer to the in-order predecessor or the in-order successor of the node. These special pointers are called threads and binary trees containing threads are called **threaded binary trees**.

In threaded binary trees the pointers that point to in-order successor of a node are called **right threads**. Likewise, pointers that point to in-order predecessor of a node are called **left threads**. The threads are typically denoted using arrows as shown in Figure 7-13.

Figure 7-13. *Threaded binary tree.*

Figure 7-13(b) shows a head node containing a value -999. The entire binary tree is shown as the left child of this head node. The right link of the head node points to itself. This head node is useful while creating programs for threaded binary tree. For example, while traversing the tree we can start with head node, visit each node and stop the traversal when we reach the head node once again. Note that in Figure 7-13(b) predecessor of node D and successor of node I point to the head node as they happen to be first and last node in the in-order traversal sequence.

In a program to help us distinguish between a pointer and a thread, the structure that represents a node contains two additional fields, **leftflag** and **rightflag**. If they contain a true they represent a thread, and if they contain a false, then they represent a pointer to a child node. The structure declaration for a node would be as shown below.

```
struct thtree
{
  enum boolean leftflag;
  struct thtree *left;
  int data;
  struct thtree *right;
  enum boolean rightflag;
};
```

A threaded binary tree created using this structure is shown in Figure 7-14.

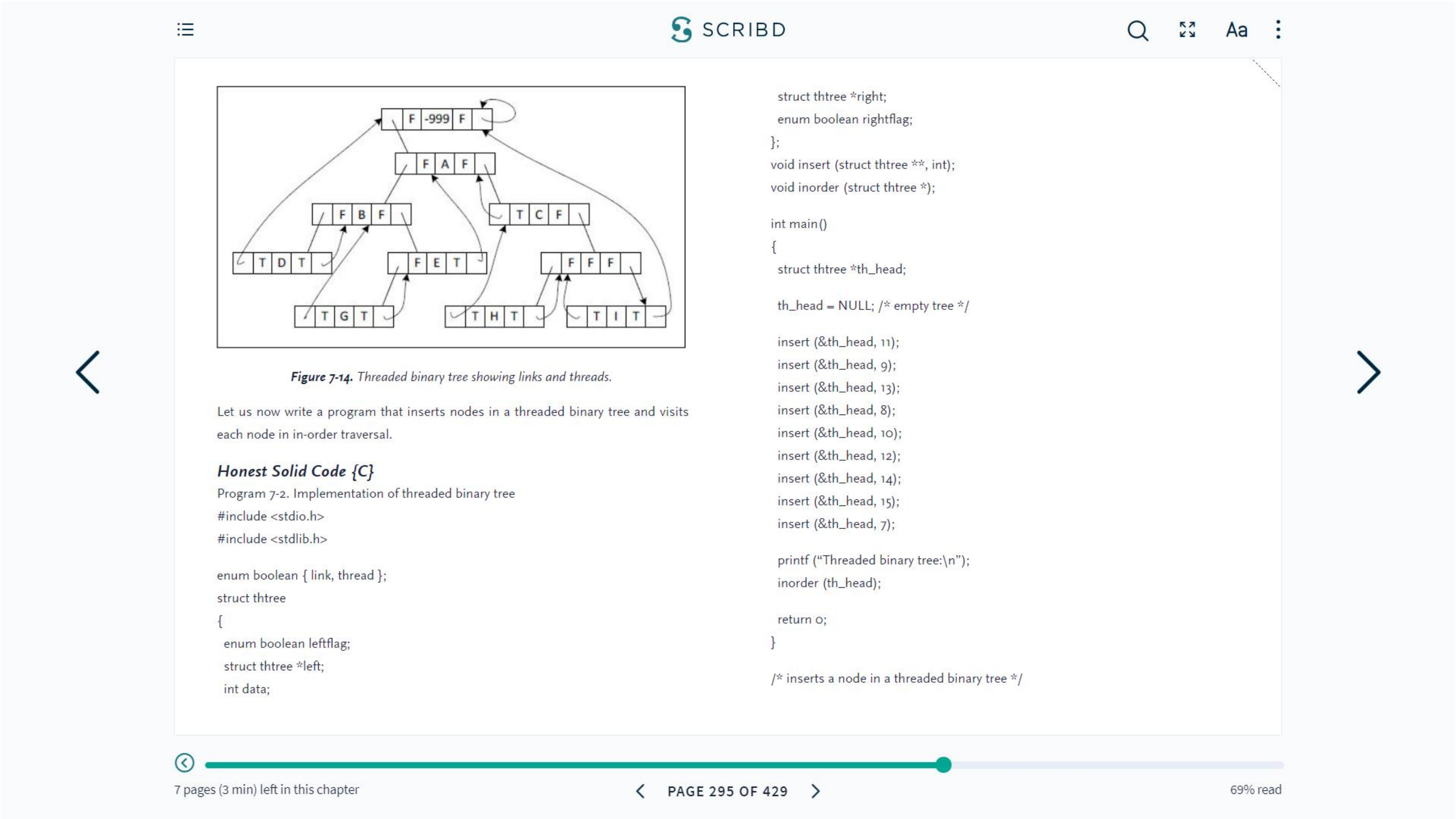


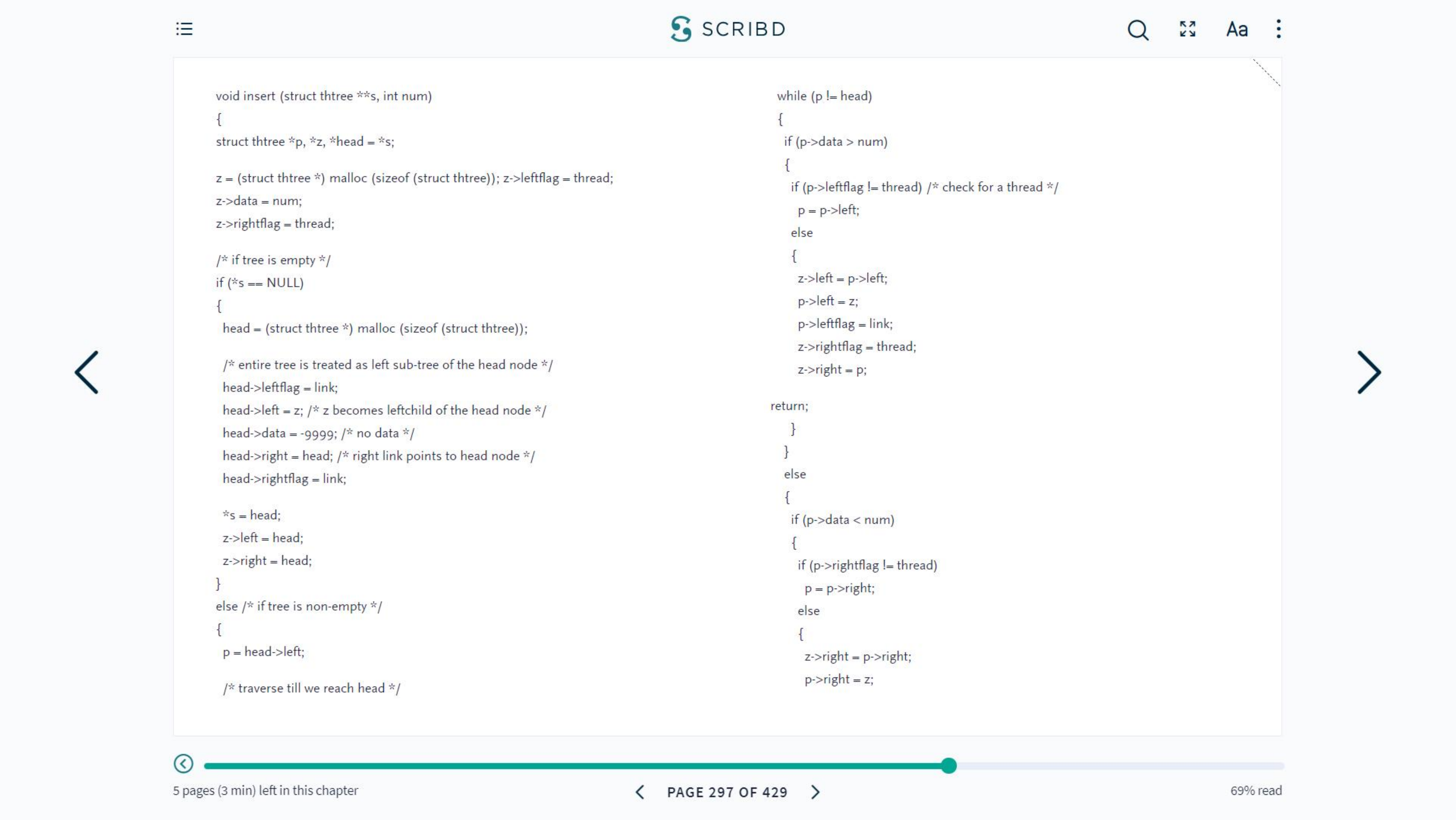
Figure 7-14. Threaded binary tree showing links and threads.

Let us now write a program that inserts nodes in a threaded binary tree and visits each node in in-order traversal.

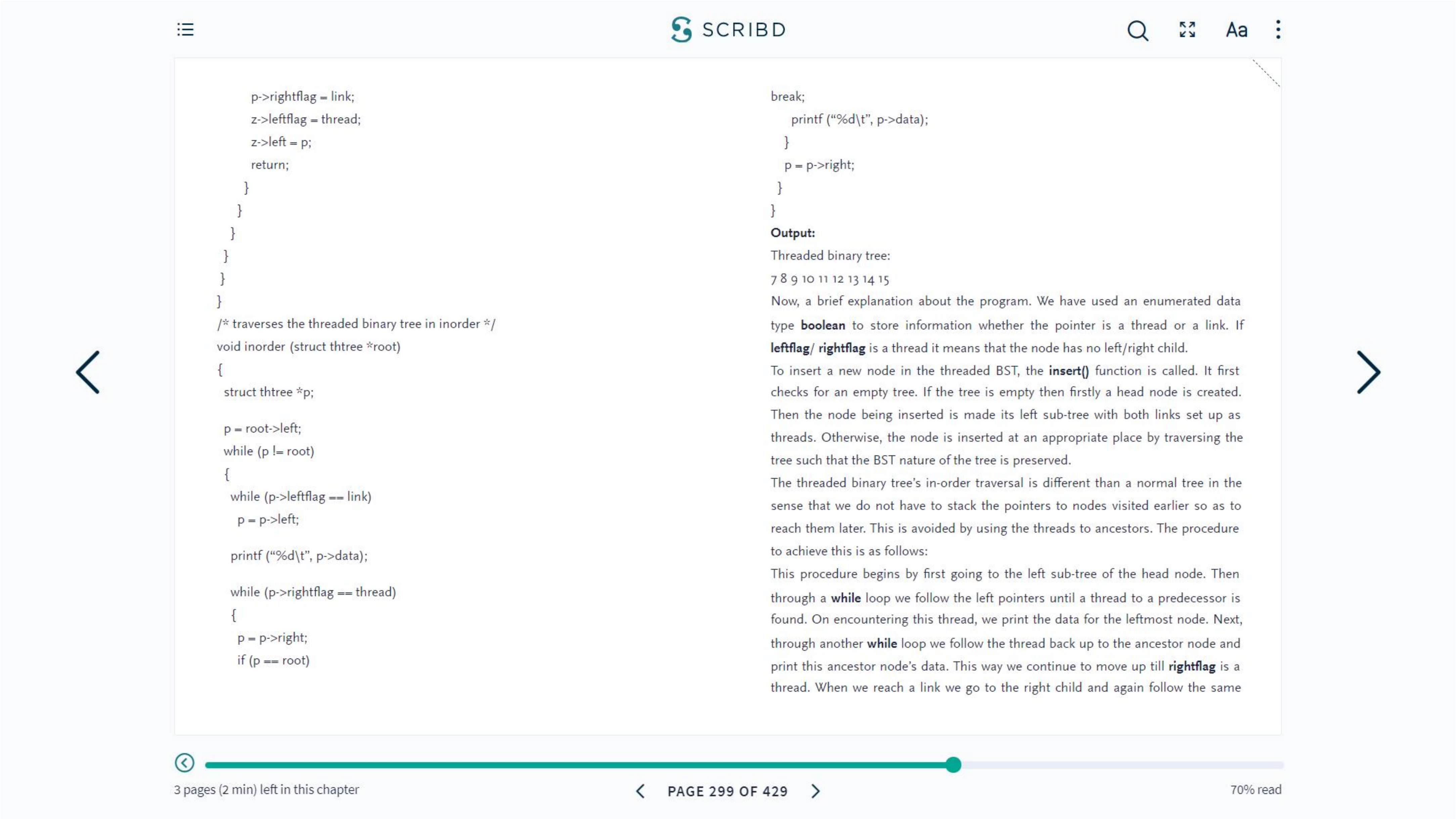
Honest Solid Code {C}

Program 7-2. Implementation of threaded binary tree

```
#include <stdio.h>
#include <stdlib.h>

enum boolean { link, thread };
struct thtree
{
  enum boolean leftflag;
  struct thtree *left;
  int data;
  struct thtree *right;
  enum boolean rightflag;
};
void insert (struct thtree **, int);
void inorder (struct thtree *);

int main()
{
  struct thtree *th_head;

  th_head = NULL; /* empty tree */

  insert (&th_head, 11);
  insert (&th_head, 9);
  insert (&th_head, 13);
  insert (&th_head, 8);
  insert (&th_head, 10);
  insert (&th_head, 12);
  insert (&th_head, 14);
  insert (&th_head, 15);
  insert (&th_head, 7);

  printf ("Threaded binary tree:\n");
  inorder (th_head);

  return 0;
}

/* inserts a node in a threaded binary tree */
```



```
void insert (struct thtree **s, int num)
{
struct thtree *p, *z, *head = *s;

z = (struct thtree *) malloc (sizeof (struct thtree)); z->leftflag = thread;
z->data = num;
z->rightflag = thread;

/* if tree is empty */
if (*s == NULL)
{
  head = (struct thtree *) malloc (sizeof (struct thtree));

  /* entire tree is treated as left sub-tree of the head node */
  head->leftflag = link;
  head->left = z; /* z becomes leftchild of the head node */
  head->data = -9999; /* no data */
  head->right = head; /* right link points to head node */
  head->rightflag = link;

  *s = head;
  z->left = head;
  z->right = head;
}
else /* if tree is non-empty */
{
  p = head->left;

  /* traverse till we reach head */
  while (p != head)
  {
    if (p->data > num)
    {
      if (p->leftflag != thread) /* check for a thread */
        p = p->left;
      else
      {
        z->left = p->left;
        p->left = z;
        p->leftflag = link;
        z->rightflag = thread;
        z->right = p;

return;
      }
    }
    else
    {
      if (p->data < num)
      {
        if (p->rightflag != thread)
          p = p->right;
        else
        {
          z->right = p->right;
          p->right = z;
```



```
        p->rightflag = link;
        z->leftflag = thread;
        z->left = p;
        return;
       }
      }
     }
    }
   }
  }
/* traverses the threaded binary tree in inorder */
void inorder (struct thtree *root)
{
 struct thtree *p;

 p = root->left;
 while (p != root)
 {
  while (p->leftflag == link)
   p = p->left;

  printf ("%d\t", p->data);

  while (p->rightflag == thread)
  {
   p = p->right;
   if (p == root)
break;
   printf ("%d\t", p->data);
  }
  p = p->right;
 }
}
```

Output:

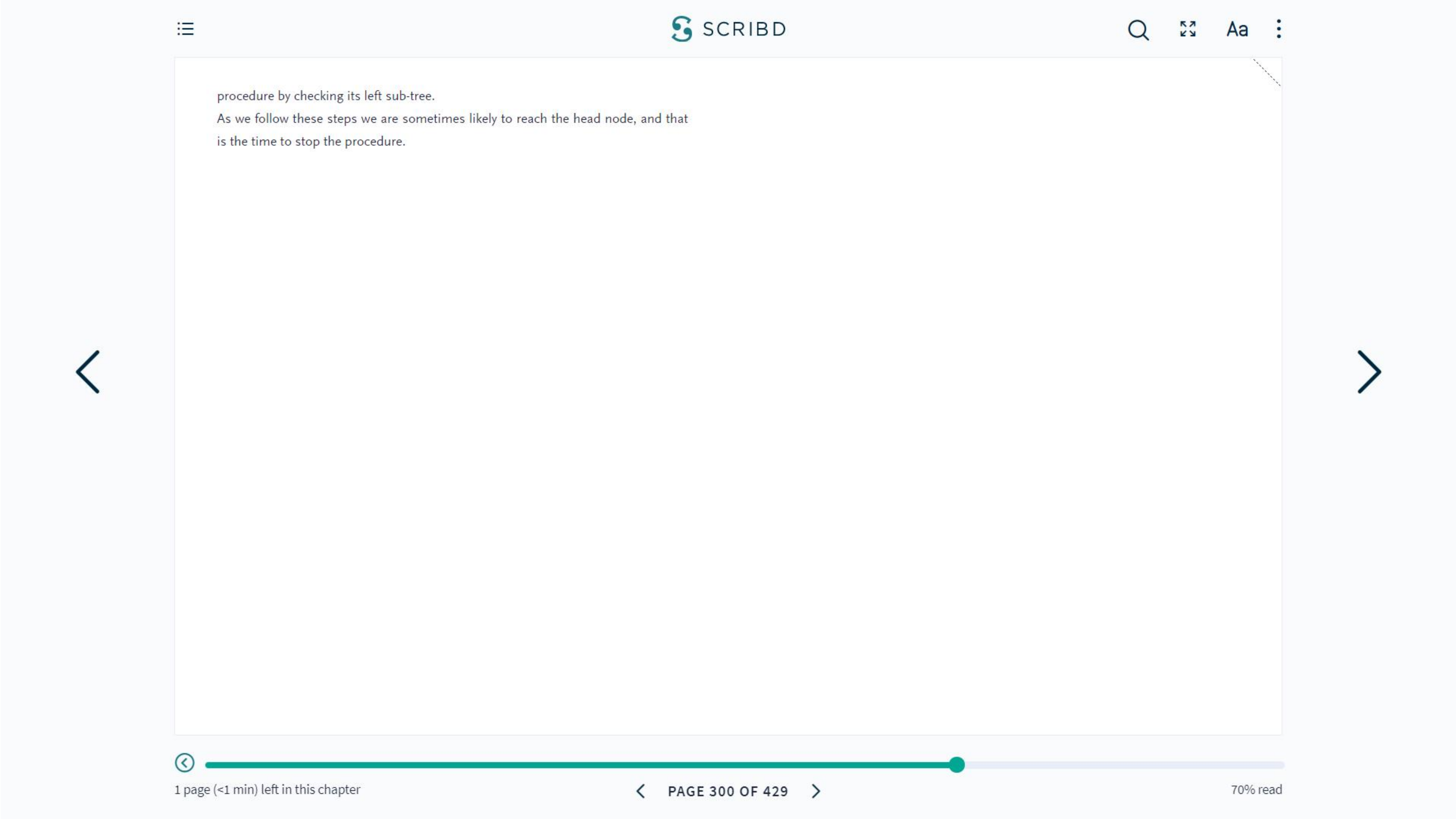
```
Threaded binary tree:
7 8 9 10 11 12 13 14 15
```

Now, a brief explanation about the program. We have used an enumerated data type **boolean** to store information whether the pointer is a thread or a link. If **leftflag**/ **rightflag** is a thread it means that the node has no left/right child.

To insert a new node in the threaded BST, the **insert()** function is called. It first checks for an empty tree. If the tree is empty then firstly a head node is created. Then the node being inserted is made its left sub-tree with both links set up as threads. Otherwise, the node is inserted at an appropriate place by traversing the tree such that the BST nature of the tree is preserved.

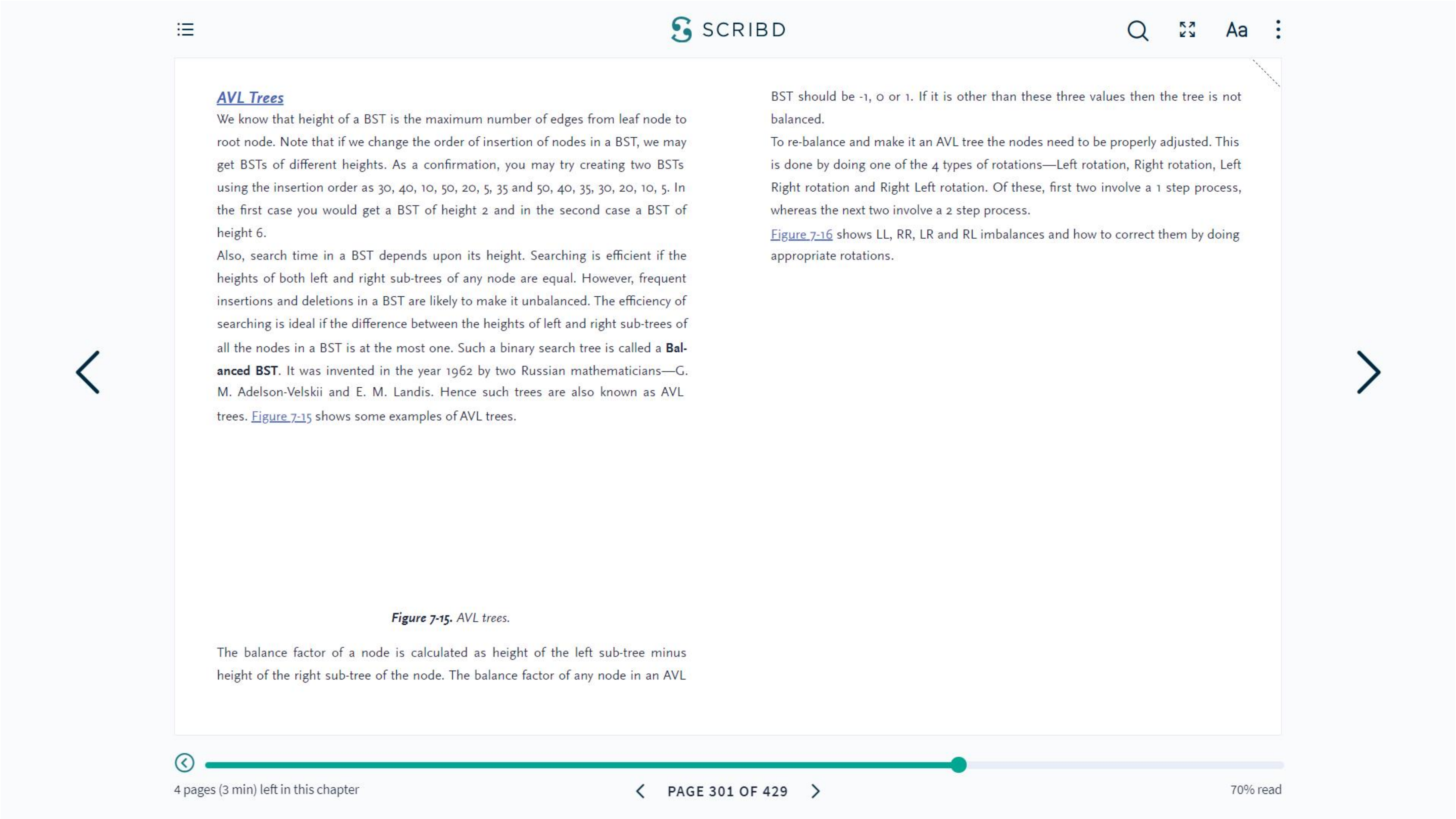
The threaded binary tree's in-order traversal is different than a normal tree in the sense that we do not have to stack the pointers to nodes visited earlier so as to reach them later. This is avoided by using the threads to ancestors. The procedure to achieve this is as follows:

This procedure begins by first going to the left sub-tree of the head node. Then through a **while** loop we follow the left pointers until a thread to a predecessor is found. On encountering this thread, we print the data for the leftmost node. Next, through another **while** loop we follow the thread back up to the ancestor node and print this ancestor node's data. This way we continue to move up till **rightflag** is a thread. When we reach a link we go to the right child and again follow the same



procedure by checking its left sub-tree.

As we follow these steps we are sometimes likely to reach the head node, and that is the time to stop the procedure.



AVL Trees

We know that height of a BST is the maximum number of edges from leaf node to root node. Note that if we change the order of insertion of nodes in a BST, we may get BSTs of different heights. As a confirmation, you may try creating two BSTs using the insertion order as 30, 40, 10, 50, 20, 5, 35 and 50, 40, 35, 30, 20, 10, 5. In the first case you would get a BST of height 2 and in the second case a BST of height 6.

Also, search time in a BST depends upon its height. Searching is efficient if the heights of both left and right sub-trees of any node are equal. However, frequent insertions and deletions in a BST are likely to make it unbalanced. The efficiency of searching is ideal if the difference between the heights of left and right sub-trees of all the nodes in a BST is at the most one. Such a binary search tree is called a **Balanced BST**. It was invented in the year 1962 by two Russian mathematicians—G. M. Adelson-Velskii and E. M. Landis. Hence such trees are also known as AVL trees. Figure 7-15 shows some examples of AVL trees.

Figure 7-15. *AVL trees.*

The balance factor of a node is calculated as height of the left sub-tree minus height of the right sub-tree of the node. The balance factor of any node in an AVL BST should be -1, 0 or 1. If it is other than these three values then the tree is not balanced.

To re-balance and make it an AVL tree the nodes need to be properly adjusted. This is done by doing one of the 4 types of rotations—Left rotation, Right rotation, Left Right rotation and Right Left rotation. Of these, first two involve a 1 step process, whereas the next two involve a 2 step process.

Figure 7-16 shows LL, RR, LR and RL imbalances and how to correct them by doing appropriate rotations.

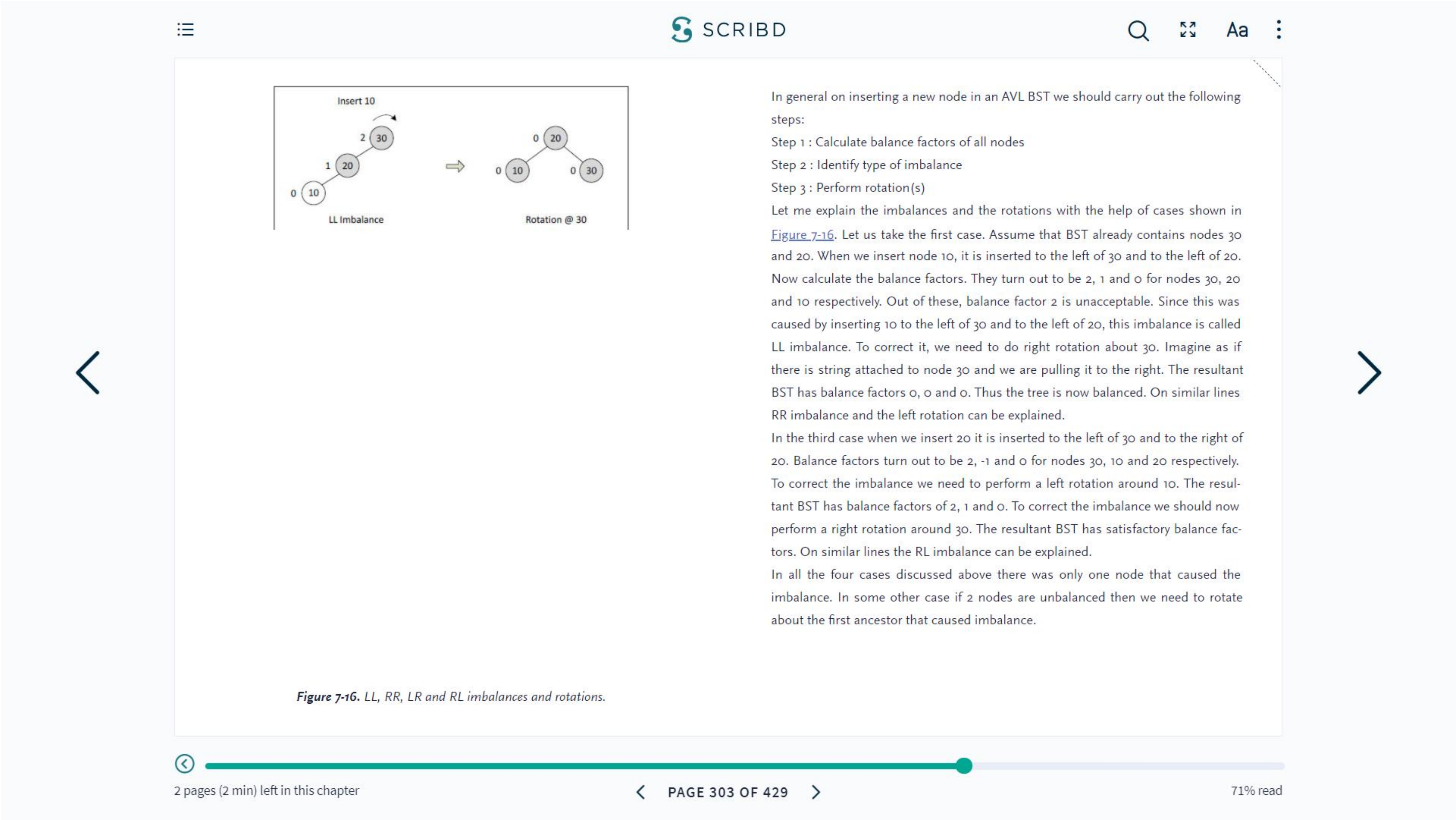


Figure 7-16. LL, RR, LR and RL imbalances and rotations.

In general on inserting a new node in an AVL BST we should carry out the following steps:

Step 1 : Calculate balance factors of all nodes

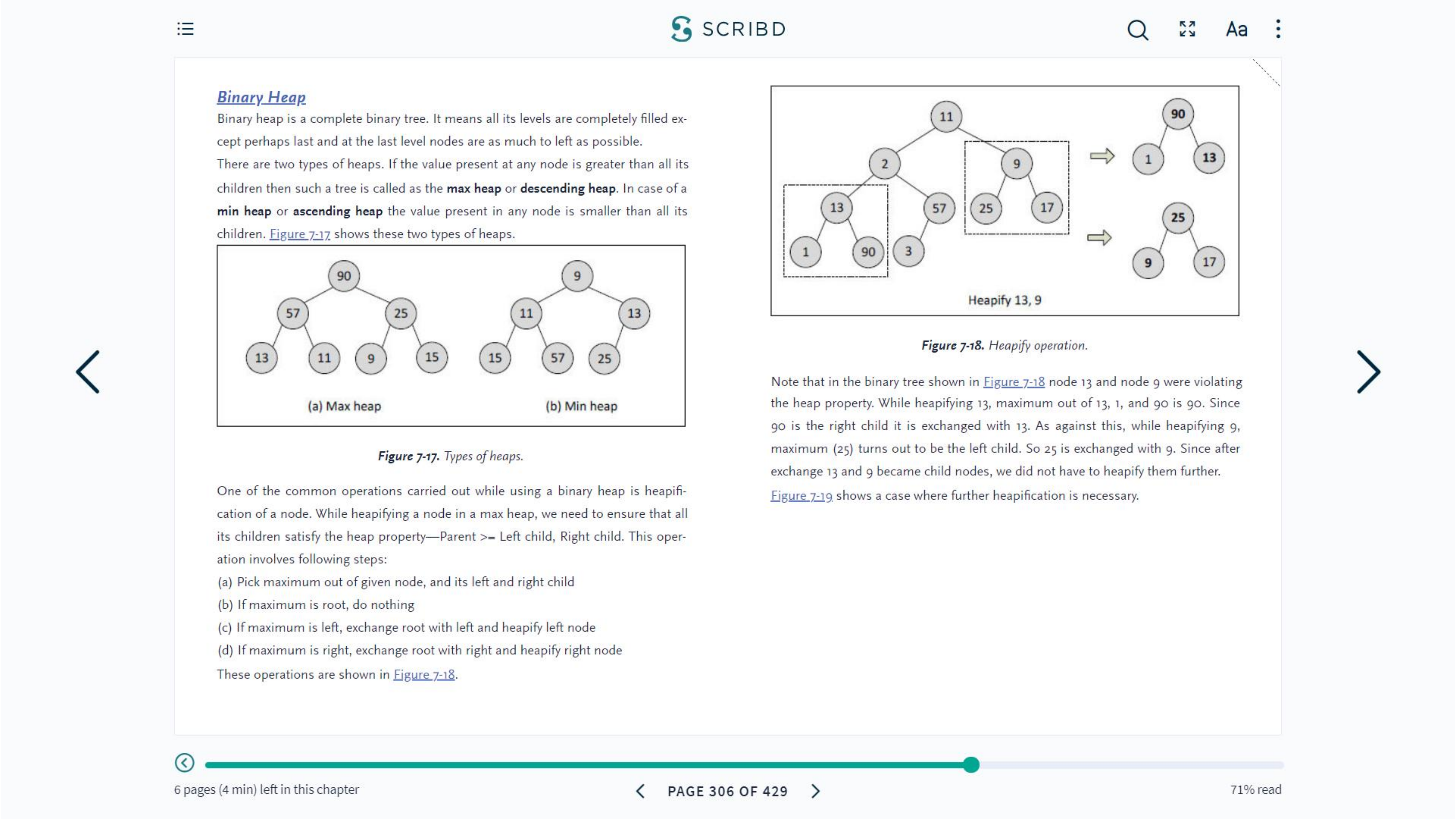
Step 2 : Identify type of imbalance

Step 3 : Perform rotation(s)

Let me explain the imbalances and the rotations with the help of cases shown in Figure 7-16. Let us take the first case. Assume that BST already contains nodes 30 and 20. When we insert node 10, it is inserted to the left of 30 and to the left of 20. Now calculate the balance factors. They turn out to be 2, 1 and 0 for nodes 30, 20 and 10 respectively. Out of these, balance factor 2 is unacceptable. Since this was caused by inserting 10 to the left of 30 and to the left of 20, this imbalance is called LL imbalance. To correct it, we need to do right rotation about 30. Imagine as if there is string attached to node 30 and we are pulling it to the right. The resultant BST has balance factors 0, 0 and 0. Thus the tree is now balanced. On similar lines RR imbalance and the left rotation can be explained.

In the third case when we insert 20 it is inserted to the left of 30 and to the right of 20. Balance factors turn out to be 2, -1 and 0 for nodes 30, 10 and 20 respectively. To correct the imbalance we need to perform a left rotation around 10. The resultant BST has balance factors of 2, 1 and 0. To correct the imbalance we should now perform a right rotation around 30. The resultant BST has satisfactory balance factors. On similar lines the RL imbalance can be explained.

In all the four cases discussed above there was only one node that caused the imbalance. In some other case if 2 nodes are unbalanced then we need to rotate about the first ancestor that caused imbalance.



Binary Heap

Binary heap is a complete binary tree. It means all its levels are completely filled except perhaps last and at the last level nodes are as much to left as possible.

There are two types of heaps. If the value present at any node is greater than all its children then such a tree is called as the **max heap** or **descending heap**. In case of a **min heap** or **ascending heap** the value present in any node is smaller than all its children. Figure 7-17 shows these two types of heaps.

Figure 7-17. *Types of heaps.*

One of the common operations carried out while using a binary heap is heapification of a node. While heapifying a node in a max heap, we need to ensure that all its children satisfy the heap property—Parent >= Left child, Right child. This operation involves following steps:

(a) Pick maximum out of given node, and its left and right child

(b) If maximum is root, do nothing

(c) If maximum is left, exchange root with left and heapify left node

(d) If maximum is right, exchange root with right and heapify right node

These operations are shown in Figure 7-18.

Figure 7-18. *Heapify operation.*

Note that in the binary tree shown in Figure 7-18 node 13 and node 9 were violating the heap property. While heapifying 13, maximum out of 13, 1, and 90 is 90. Since 90 is the right child it is exchanged with 13. As against this, while heapifying 9, maximum (25) turns out to be the left child. So 25 is exchanged with 9. Since after exchange 13 and 9 became child nodes, we did not have to heapify them further.

Figure 7-19 shows a case where further heapification is necessary.

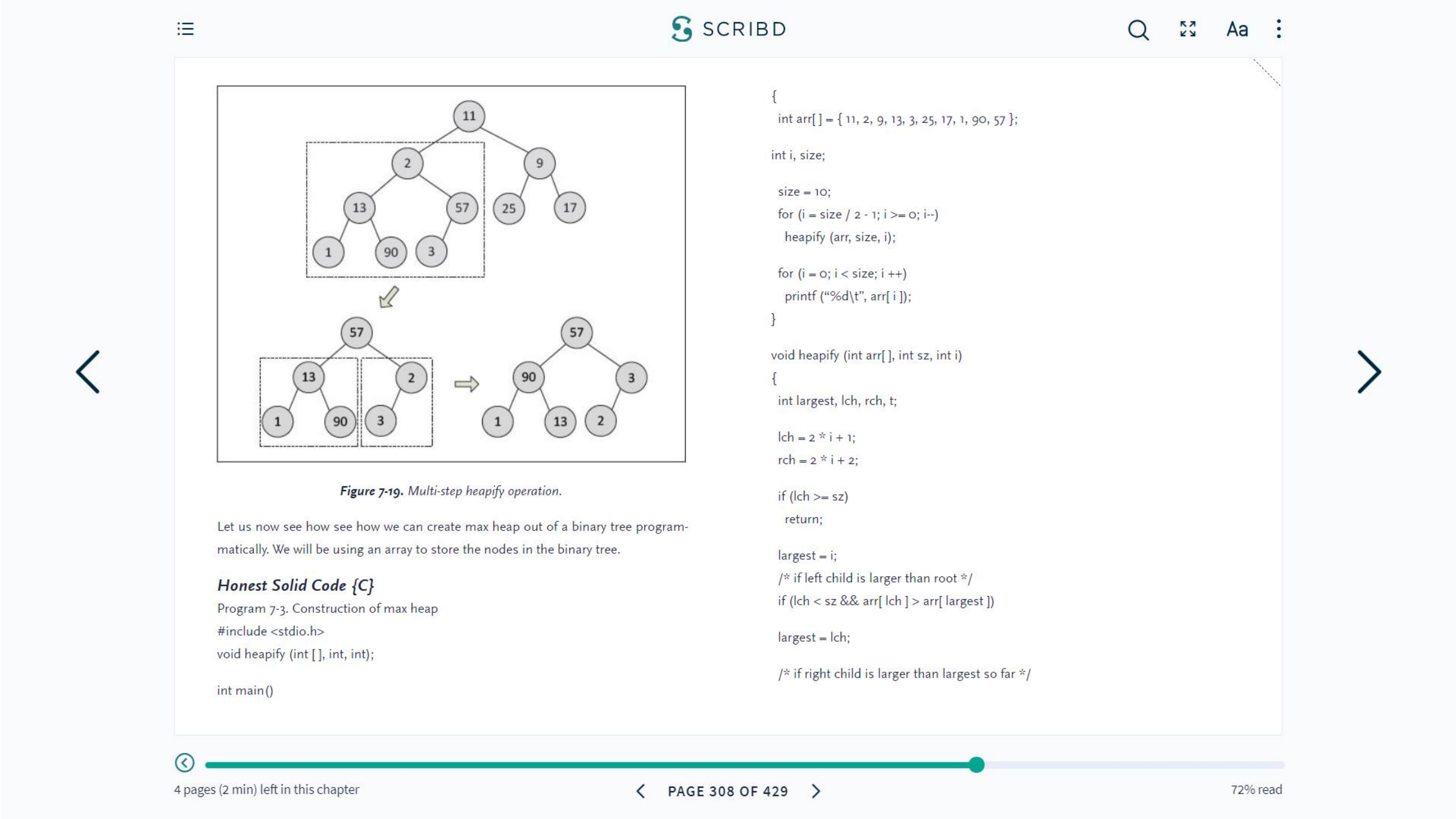


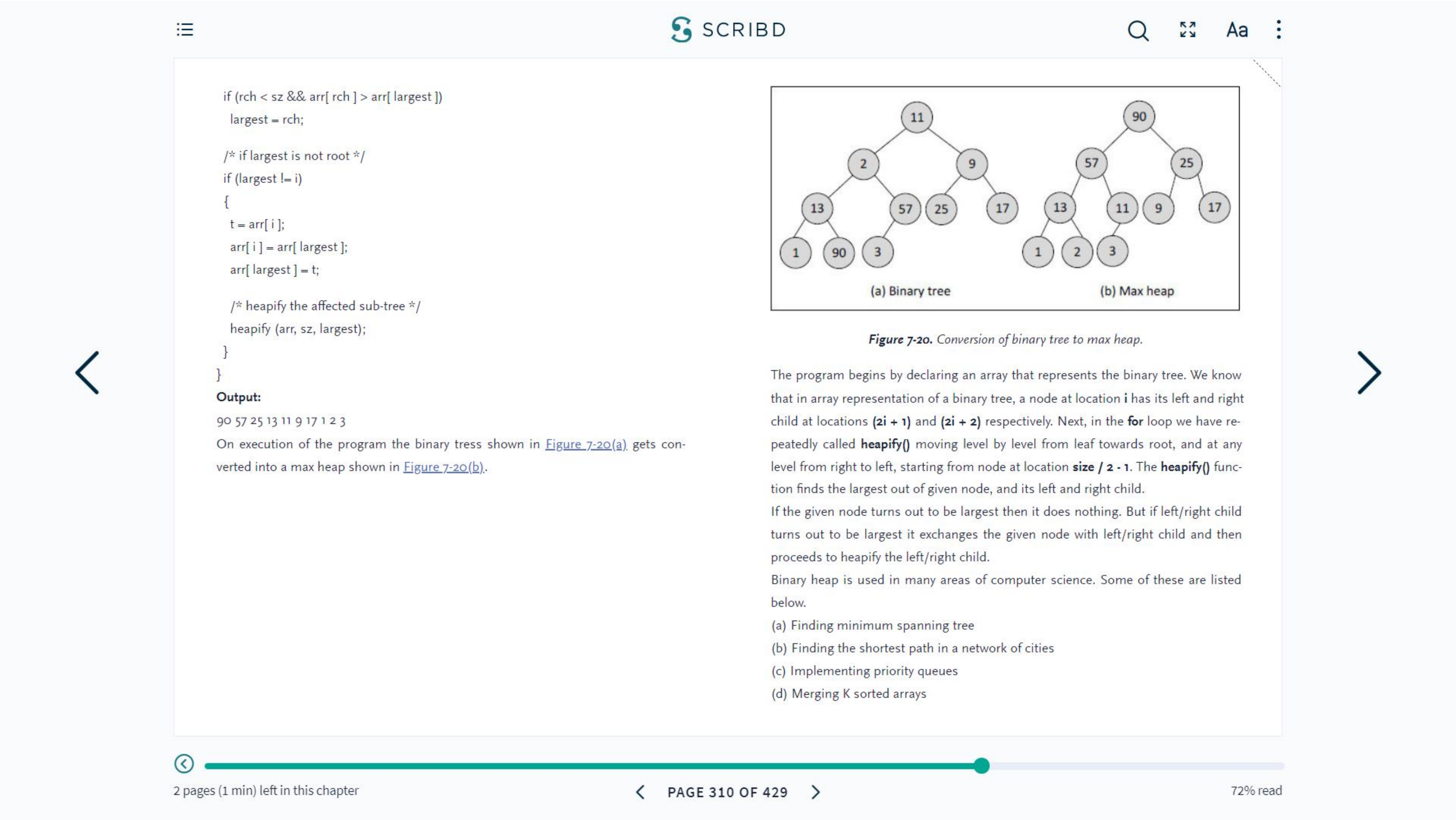
Figure 7-19. Multi-step heapify operation.

Let us now see how see how we can create max heap out of a binary tree programmatically. We will be using an array to store the nodes in the binary tree.

Honest Solid Code {C}

Program 7-3. Construction of max heap

```
#include <stdio.h>
void heapify (int [ ], int, int);

int main()
{
 int arr[ ] = { 11, 2, 9, 13, 3, 25, 17, 1, 90, 57 };

int i, size;

 size = 10;
 for (i = size / 2 - 1; i >= 0; i--)
   heapify (arr, size, i);

 for (i = 0; i < size; i ++)
   printf ("%d\t", arr[ i ]);
}

void heapify (int arr[ ], int sz, int i)
{
 int largest, lch, rch, t;

 lch = 2 * i + 1;
 rch = 2 * i + 2;

 if (lch >= sz)
   return;

 largest = i;
 /* if left child is larger than root */
 if (lch < sz && arr[ lch ] > arr[ largest ])

 largest = lch;

 /* if right child is larger than largest so far */
```



```
 if (rch < sz && arr[ rch ] > arr[ largest ])
   largest = rch;

 /* if largest is not root */
 if (largest != i)
 {
   t = arr[ i ];
   arr[ i ] = arr[ largest ];
   arr[ largest ] = t;

   /* heapify the affected sub-tree */
   heapify (arr, sz, largest);
 }
}
```

Output:

```
90 57 25 13 11 9 17 1 2 3
```

On execution of the program the binary tress shown in Figure 7-20(a) gets converted into a max heap shown in Figure 7-20(b).

Figure 7-20. *Conversion of binary tree to max heap.*

The program begins by declaring an array that represents the binary tree. We know that in array representation of a binary tree, a node at location **i** has its left and right child at locations **(2i + 1)** and **(2i + 2)** respectively. Next, in the **for** loop we have repeatedly called **heapify()** moving level by level from leaf towards root, and at any level from right to left, starting from node at location **size / 2 - 1**. The **heapify()** function finds the largest out of given node, and its left and right child.

If the given node turns out to be largest then it does nothing. But if left/right child turns out to be largest it exchanges the given node with left/right child and then proceeds to heapify the left/right child.

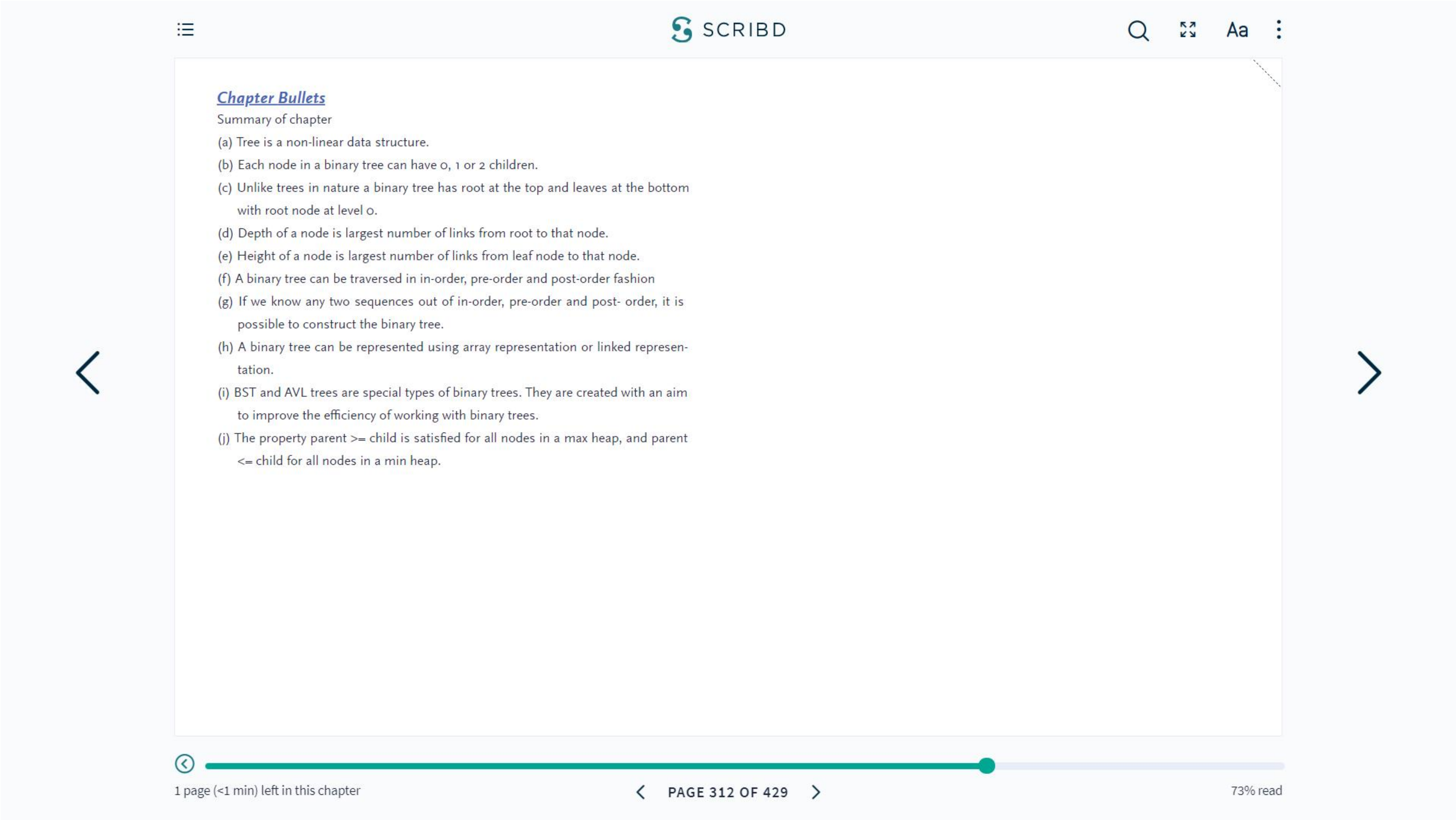
Binary heap is used in many areas of computer science. Some of these are listed below.

(a) Finding minimum spanning tree

(b) Finding the shortest path in a network of cities

(c) Implementing priority queues

(d) Merging K sorted arrays



Chapter Bullets

Summary of chapter

(a) Tree is a non-linear data structure.

(b) Each node in a binary tree can have 0, 1 or 2 children.

(c) Unlike trees in nature a binary tree has root at the top and leaves at the bottom with root node at level 0.

(d) Depth of a node is largest number of links from root to that node.

(e) Height of a node is largest number of links from leaf node to that node.

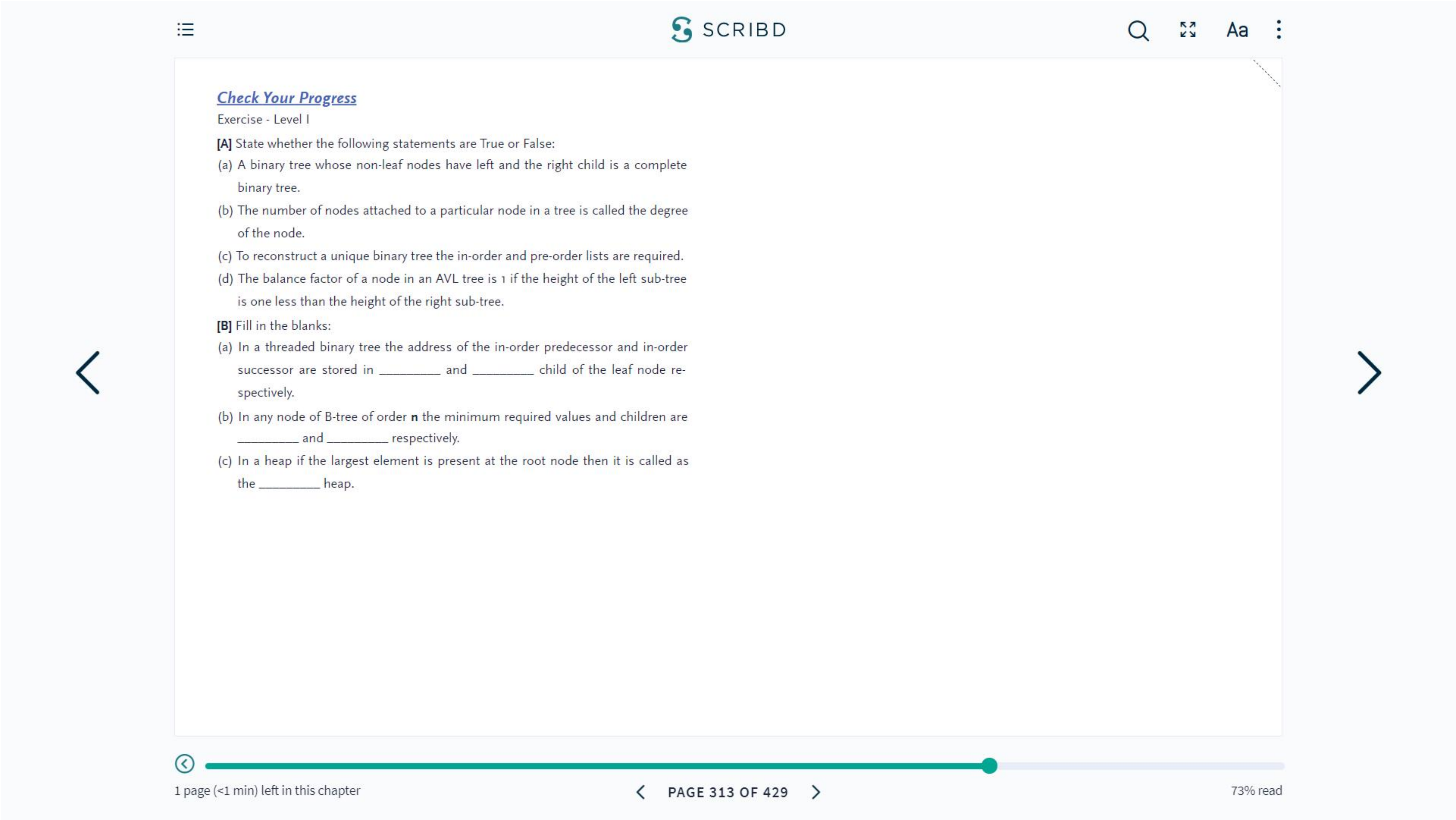
(f) A binary tree can be traversed in in-order, pre-order and post-order fashion

(g) If we know any two sequences out of in-order, pre-order and post- order, it is possible to construct the binary tree.

(h) A binary tree can be represented using array representation or linked representation.

(i) BST and AVL trees are special types of binary trees. They are created with an aim to improve the efficiency of working with binary trees.

(j) The property parent >= child is satisfied for all nodes in a max heap, and parent <= child for all nodes in a min heap.



Check Your Progress

Exercise - Level I

[A] State whether the following statements are True or False:

(a) A binary tree whose non-leaf nodes have left and the right child is a complete binary tree.

(b) The number of nodes attached to a particular node in a tree is called the degree of the node.

(c) To reconstruct a unique binary tree the in-order and pre-order lists are required.

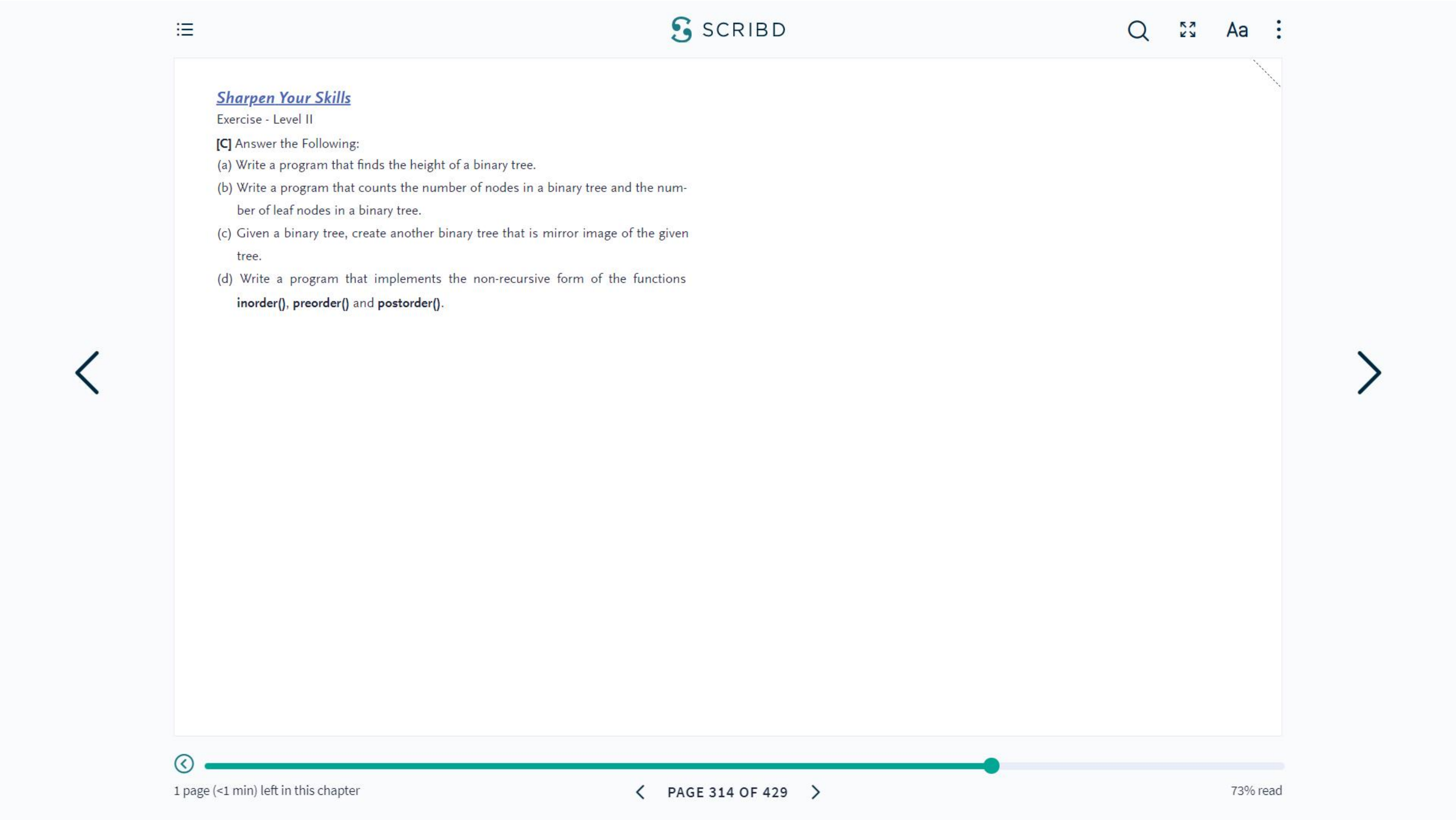
(d) The balance factor of a node in an AVL tree is 1 if the height of the left sub-tree is one less than the height of the right sub-tree.

[B] Fill in the blanks:

(a) In a threaded binary tree the address of the in-order predecessor and in-order successor are stored in _________ and _________ child of the leaf node respectively.

(b) In any node of B-tree of order **n** the minimum required values and children are _________ and _________ respectively.

(c) In a heap if the largest element is present at the root node then it is called as the _________ heap.



Sharpen Your Skills

Exercise - Level II

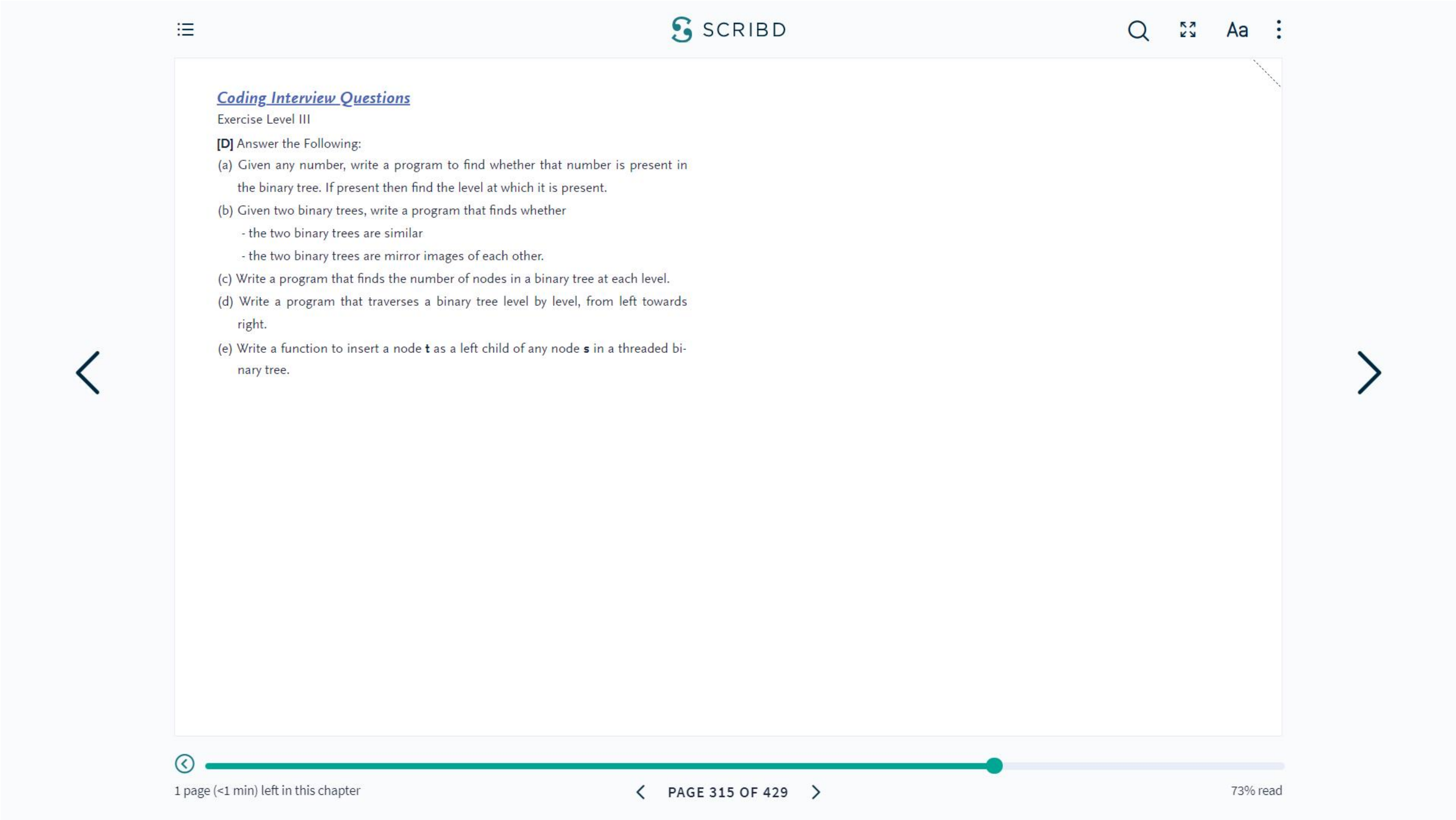
[C] Answer the Following:

(a) Write a program that finds the height of a binary tree.

(b) Write a program that counts the number of nodes in a binary tree and the number of leaf nodes in a binary tree.

(c) Given a binary tree, create another binary tree that is mirror image of the given tree.

(d) Write a program that implements the non-recursive form of the functions **inorder()**, **preorder()** and **postorder()**.



Coding Interview Questions

Exercise Level III

[D] Answer the Following:

(a) Given any number, write a program to find whether that number is present in the binary tree. If present then find the level at which it is present.

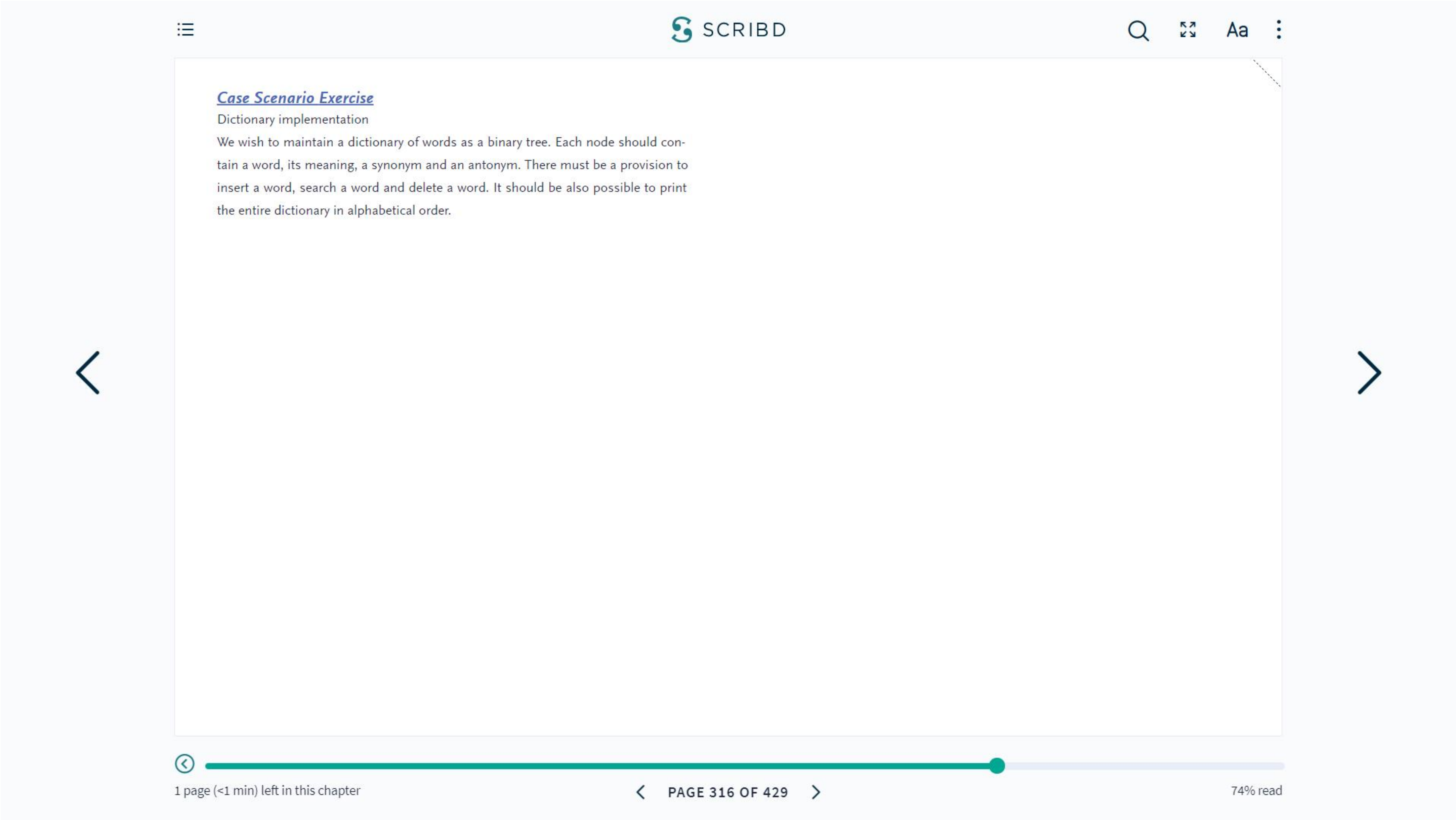
(b) Given two binary trees, write a program that finds whether

- the two binary trees are similar
- the two binary trees are mirror images of each other.

(c) Write a program that finds the number of nodes in a binary tree at each level.

(d) Write a program that traverses a binary tree level by level, from left towards right.

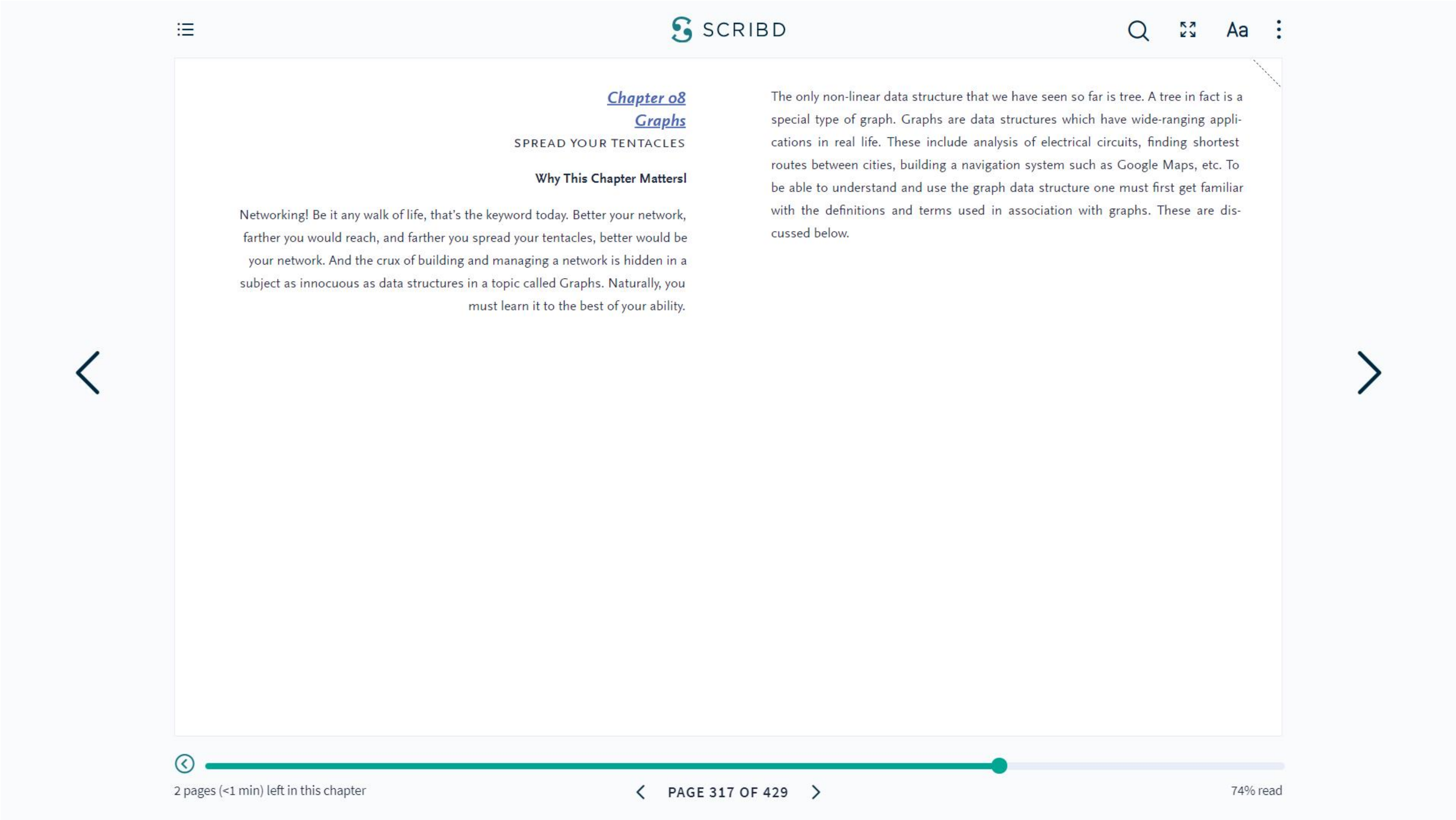
(e) Write a function to insert a node **t** as a left child of any node **s** in a threaded binary tree.



Case Scenario Exercise

Dictionary implementation

We wish to maintain a dictionary of words as a binary tree. Each node should contain a word, its meaning, a synonym and an antonym. There must be a provision to insert a word, search a word and delete a word. It should be also possible to print the entire dictionary in alphabetical order.



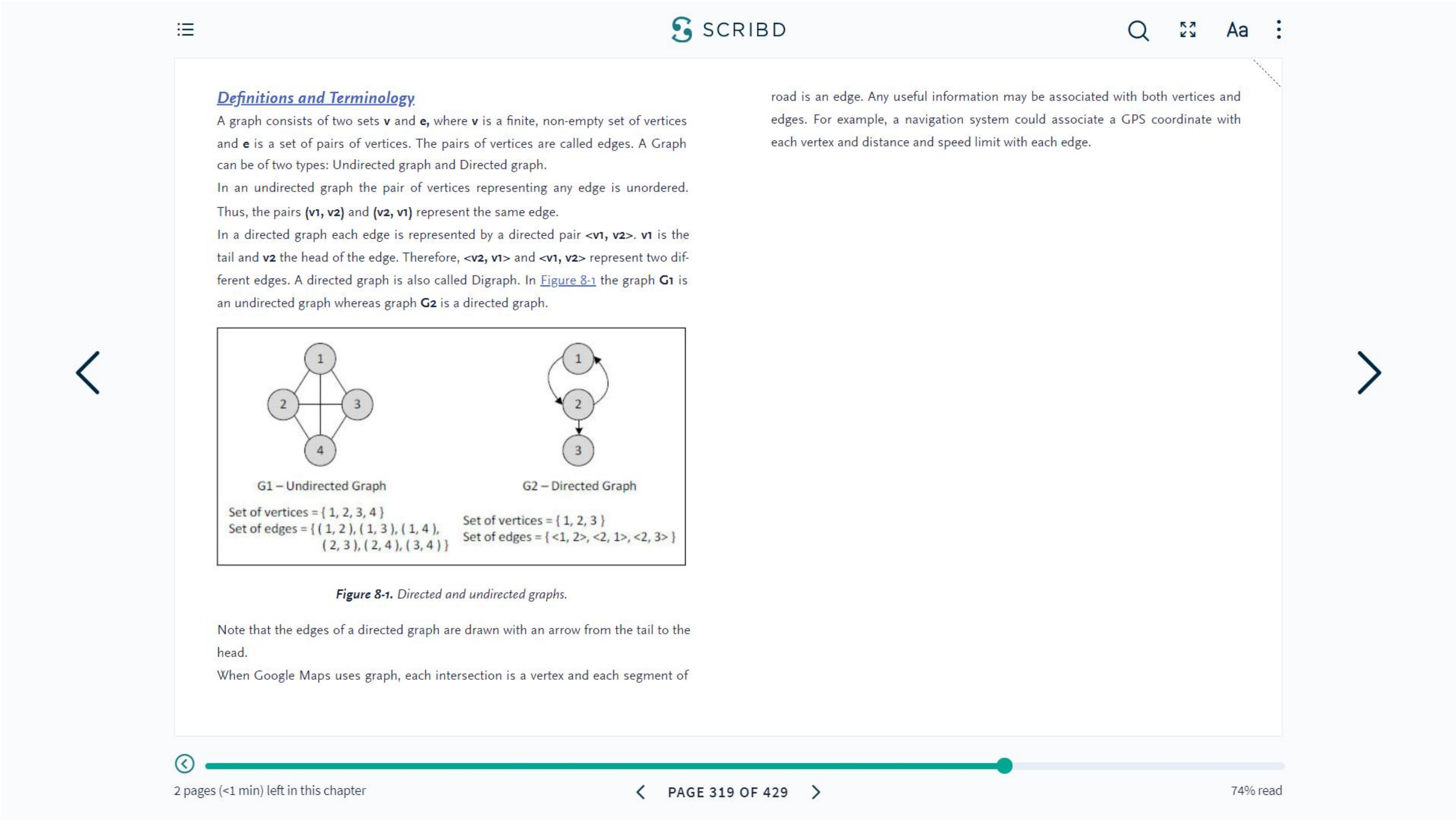
Chapter 08
Graphs

SPREAD YOUR TENTACLES

Why This Chapter Mattersl

Networking! Be it any walk of life, that's the keyword today. Better your network, farther you would reach, and farther you spread your tentacles, better would be your network. And the crux of building and managing a network is hidden in a subject as innocuous as data structures in a topic called Graphs. Naturally, you must learn it to the best of your ability.

The only non-linear data structure that we have seen so far is tree. A tree in fact is a special type of graph. Graphs are data structures which have wide-ranging applications in real life. These include analysis of electrical circuits, finding shortest routes between cities, building a navigation system such as Google Maps, etc. To be able to understand and use the graph data structure one must first get familiar with the definitions and terms used in association with graphs. These are discussed below.



Definitions and Terminology

A graph consists of two sets **v** and **e,** where **v** is a finite, non-empty set of vertices and **e** is a set of pairs of vertices. The pairs of vertices are called edges. A Graph can be of two types: Undirected graph and Directed graph.

In an undirected graph the pair of vertices representing any edge is unordered. Thus, the pairs **(v1, v2)** and **(v2, v1)** represent the same edge.

In a directed graph each edge is represented by a directed pair **<v1, v2>**. **v1** is the tail and **v2** the head of the edge. Therefore, **<v2, v1>** and **<v1, v2>** represent two different edges. A directed graph is also called Digraph. In Figure 8-1 the graph **G1** is an undirected graph whereas graph **G2** is a directed graph.

Set of vertices = { 1, 2, 3, 4 }
Set of edges = { (1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4) }

Set of vertices = { 1, 2, 3 }
Set of edges = { <1, 2>, <2, 1>, <2, 3> }

Figure 8-1. *Directed and undirected graphs.*

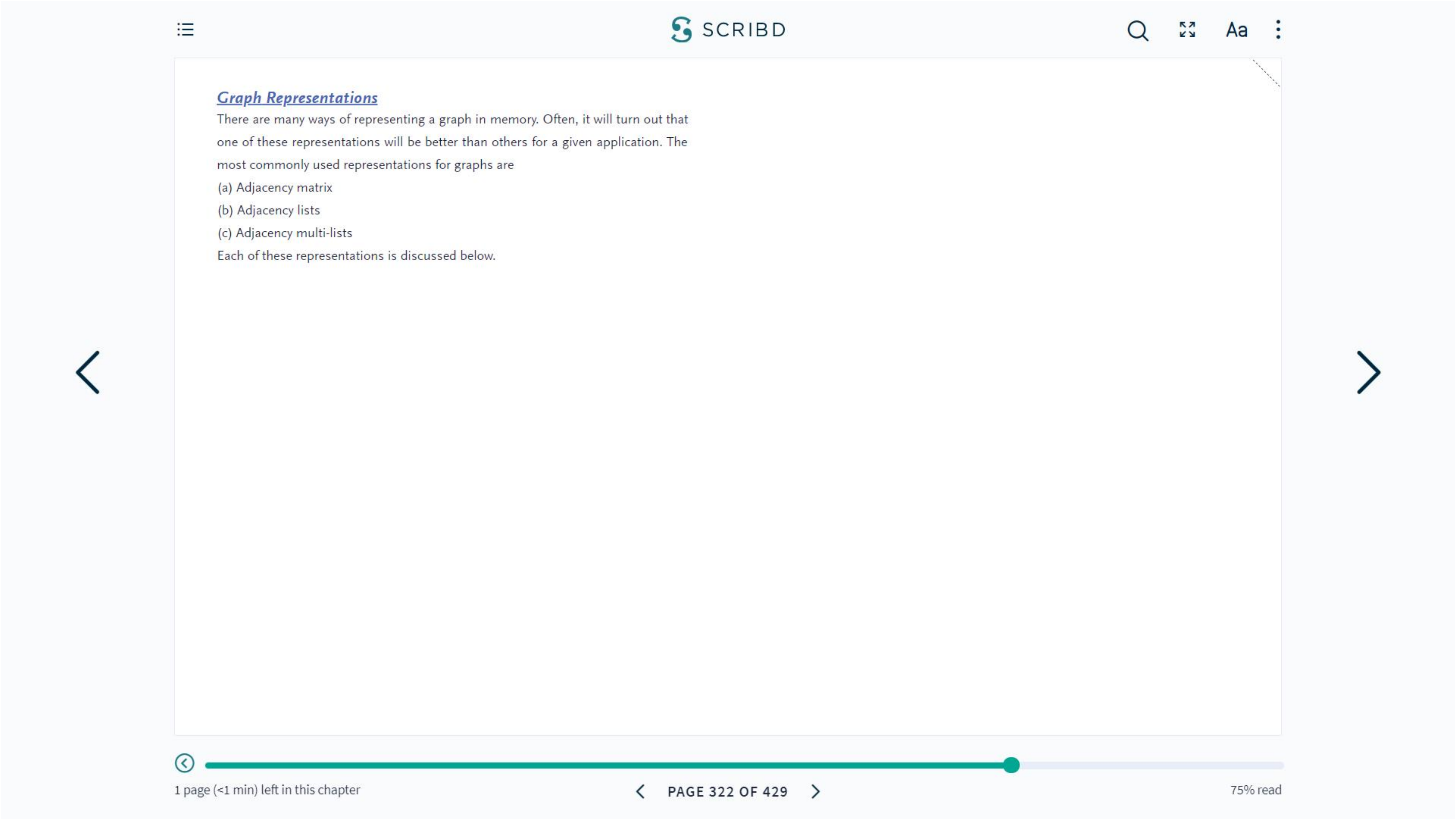
Note that the edges of a directed graph are drawn with an arrow from the tail to the head.

When Google Maps uses graph, each intersection is a vertex and each segment of road is an edge. Any useful information may be associated with both vertices and edges. For example, a navigation system could associate a GPS coordinate with each vertex and distance and speed limit with each edge.

Adjacent Vertices and Incident Edges

In an undirected graph if **(v1, v2)** is an edge in the set of edges, then the vertices **v1** and **v2** are said to be adjacent and the edge **(v1, v2)** is **incident on** vertices **v1** and **v2**. In Figure 8-2, vertex 2 in **G1** is **adjacent to** vertices 1, 3, and 4. The edges **incident on** vertex 3 in **G1** are **(1, 3)**, **(2, 3)** and **(4, 3)**.

If **<v1, v2>** is a directed edge, then vertex **v1** is said to be **adjacent to v2** while **v2** is **adjacent from v1**. The edge **<v1, v2>** is **incident on v1** and **v2**. In Figure 8-2, in **G2,** vertices 1 and 3 are **adjacent to** vertex 2, whereas, vertex 2 is **adjacent from** vertex 1. Also, the edges **incident on** vertex 2 are **<1, 2>**, **< 2, 1 >** and **< 2, 3 >**.



Graph Representations

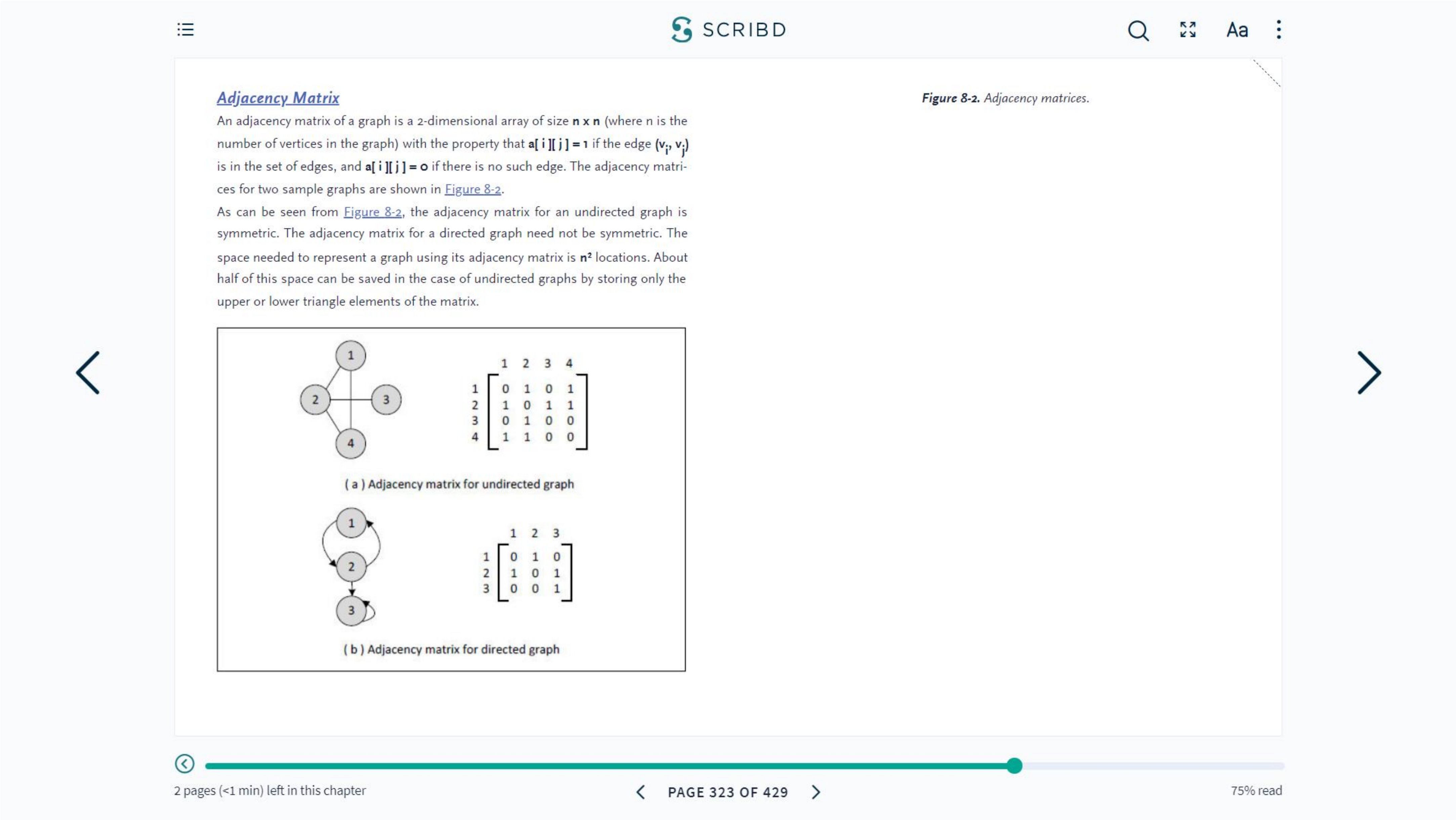
There are many ways of representing a graph in memory. Often, it will turn out that one of these representations will be better than others for a given application. The most commonly used representations for graphs are

(a) Adjacency matrix

(b) Adjacency lists

(c) Adjacency multi-lists

Each of these representations is discussed below.



Adjacency Matrix

An adjacency matrix of a graph is a 2-dimensional array of size **n x n** (where n is the number of vertices in the graph) with the property that **a[i][j] = 1** if the edge **(v_i, v_j)** is in the set of edges, and **a[i][j] = 0** if there is no such edge. The adjacency matrices for two sample graphs are shown in Figure 8-2.

As can be seen from Figure 8-2, the adjacency matrix for an undirected graph is symmetric. The adjacency matrix for a directed graph need not be symmetric. The space needed to represent a graph using its adjacency matrix is **n^2** locations. About half of this space can be saved in the case of undirected graphs by storing only the upper or lower triangle elements of the matrix.

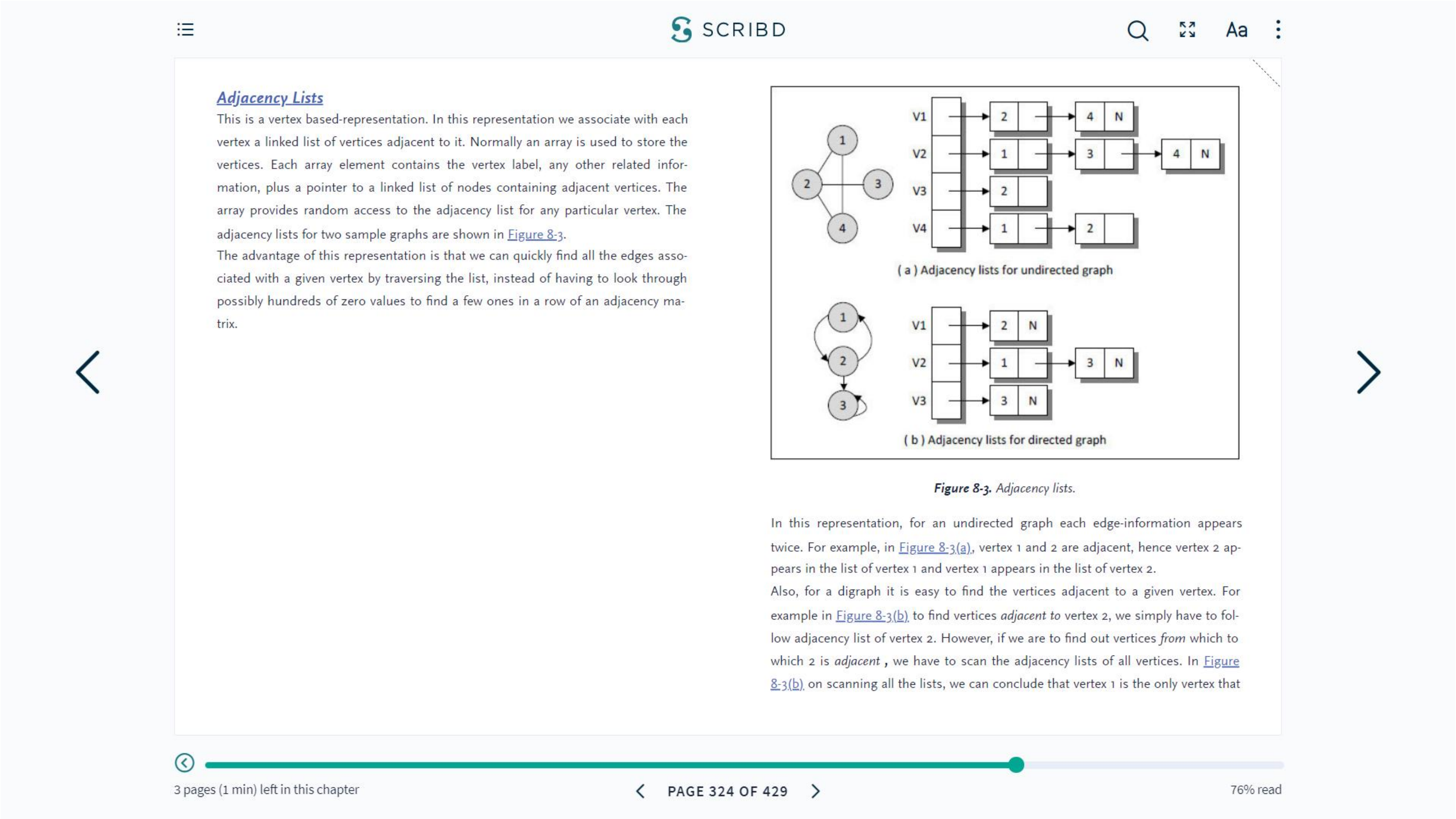
$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 \\ 3 & 0 & 1 & 0 & 0 \\ 4 & 1 & 1 & 0 & 0 \end{array}$$

(a) Adjacency matrix for undirected graph

$$\begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 1 \\ 3 & 0 & 0 & 1 \end{array}$$

(b) Adjacency matrix for directed graph

Figure 8-2. *Adjacency matrices.*



Adjacency Lists

This is a vertex based-representation. In this representation we associate with each vertex a linked list of vertices adjacent to it. Normally an array is used to store the vertices. Each array element contains the vertex label, any other related information, plus a pointer to a linked list of nodes containing adjacent vertices. The array provides random access to the adjacency list for any particular vertex. The adjacency lists for two sample graphs are shown in Figure 8-3.

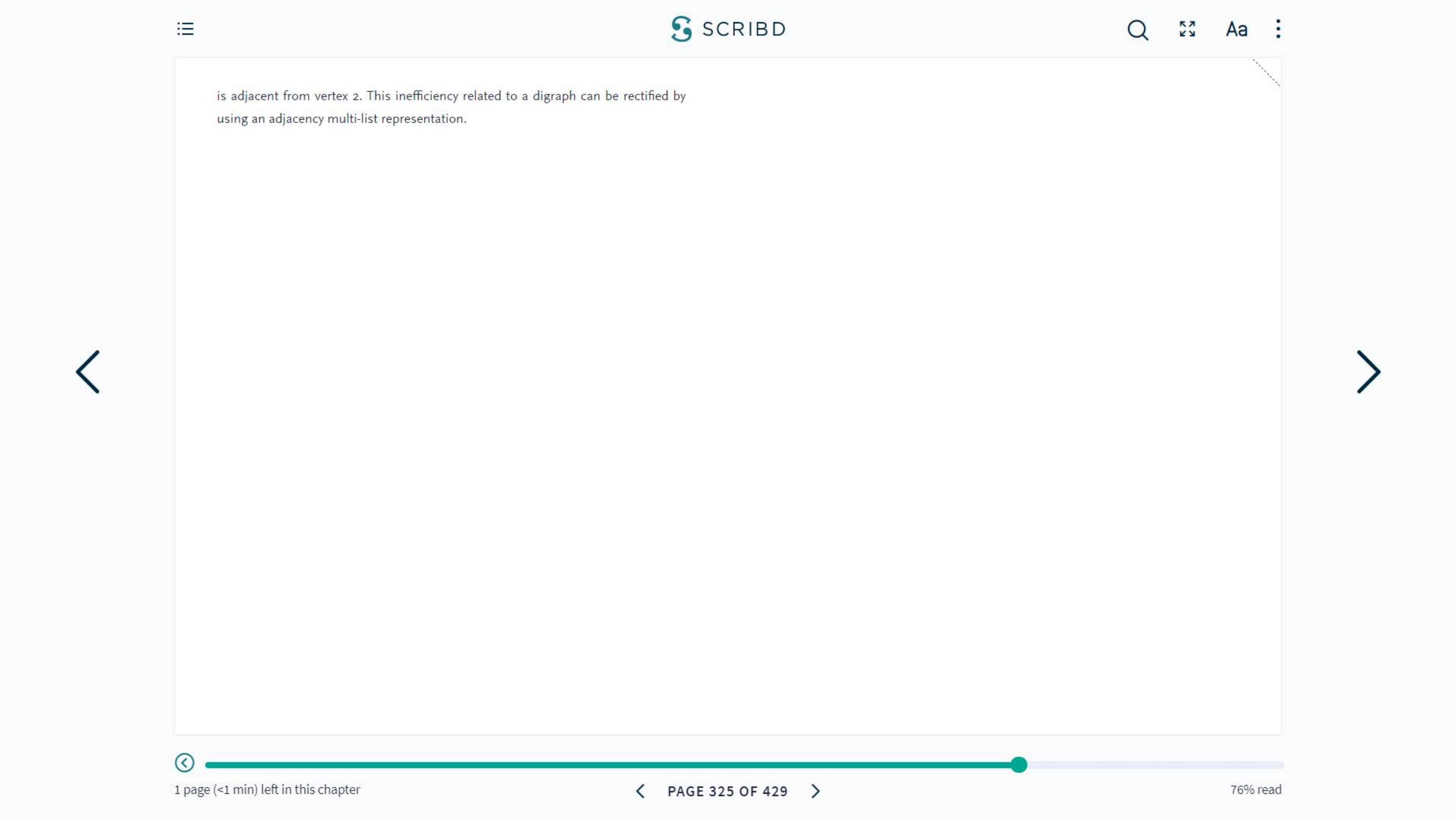
The advantage of this representation is that we can quickly find all the edges associated with a given vertex by traversing the list, instead of having to look through possibly hundreds of zero values to find a few ones in a row of an adjacency matrix.

1
2
3
4
V1
V2
V3
V4
2
4
N
1
3
4
N
2
1
2
(a) Adjacency lists for undirected graph
1
2
3
V1
V2
V3
2
N
1
3
N
3
N
(b) Adjacency lists for directed graph

Figure 8-3. *Adjacency lists.*

In this representation, for an undirected graph each edge-information appears twice. For example, in Figure 8-3(a), vertex 1 and 2 are adjacent, hence vertex 2 appears in the list of vertex 1 and vertex 1 appears in the list of vertex 2.

Also, for a digraph it is easy to find the vertices adjacent to a given vertex. For example in Figure 8-3(b) to find vertices *adjacent to* vertex 2, we simply have to follow adjacency list of vertex 2. However, if we are to find out vertices *from* which to which 2 is *adjacent* **,** we have to scan the adjacency lists of all vertices. In Figure 8-3(b) on scanning all the lists, we can conclude that vertex 1 is the only vertex that



is adjacent from vertex 2. This inefficiency related to a digraph can be rectified by using an adjacency multi-list representation.

Adjacency Multi-lists

An adjacency multi-list is an edge-based representation rather than a vertex-based representation. Each node that represents an edge consists of 5 fields. Of these, 2nd and 4th field are related and 3rd and 5th field are related. We would soon see the relationship.

Like adjacency list, an array of vertices is also maintained. Each array element points to a suitable edge node.

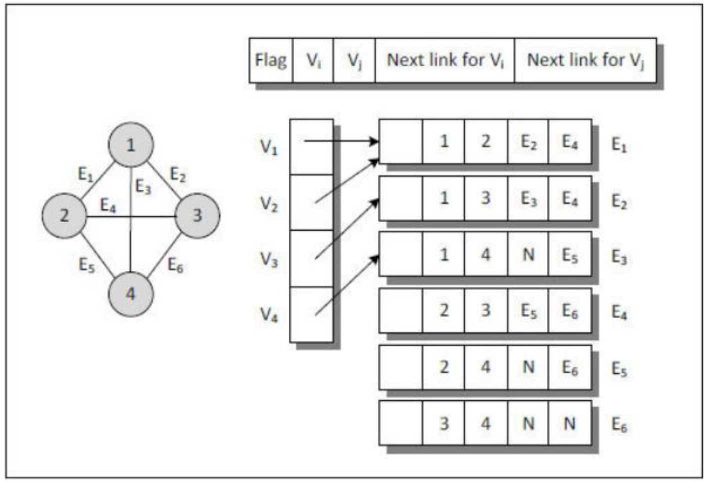


Figure 8-4. *Adjacency multi-lists for undirected graph.*

While constructing the multi-lists for graph shown in Figure 8-4 firstly the fields V_i and V_j are filled in the 6 edge nodes, E_1 to E_6. Then we start with vertex 1. This vertex has 3 incident edges E_1, E_2 and E_3. Hence the 1st element of vertices array is made to point to edge E_1. Then the edge node for E_1 is searched for vertex 1. It is found in V_i field of E_1. Since the next incident edge for vertex 1 is E_2 the fourth field of node E_1 is set up with pointer to edge node E_2. Then node E_2 is examined for vertex 1. Here also 1 is found in field V_i. Hence pointer to node E_3 is set up in fourth field of node E_2. Then E_3 is searched for vertex 1. It is found in field V_i. Since there are no more edges incident on vertex 1 hence fourth field of node E_3 is set with NULL.

Let us understand this process for vertex 2 as well. Vertex 2 has 3 incident edges E_1, E_4 and E_5. So to begin with, the 2nd element of vertices array is made to point to edge E_1. Then E_1 is searched for vertex 2. 1 is found in V_j field of E_1. Since the next incident edge for vertex 2 is E_4 the fifth field of node E_1 is set up with pointer to edge node E_4. Then node E_4 is examined for vertex 2. Here 2 is found in field V_i. Hence the fourth field of node E_4 is setup with pointer to edge E_5. Then E_5 is searched for vertex 2. It is found in field V_i. Since there are no more edges incident on vertex 2 hence fourth field of node E_5 is set with NULL.

If this procedure is carried out systematically for all other vertices then the adjacency multi-lists shown in Figure 8-4 would get created. If we traverse these lists for each element of the vertex array then we can find out the sequence of incident edges for each vertex. These sequences are given below.

Vertex 1 : E_1, E_2, E_3
Vertex 2 : E_1, E_4, E_5
Vertex 3 : E_2, E_4, E_6
Vertex 4 : E_3, E_5, E_6

On similar lines we can also create adjacency multi-lists for a directed graph. Only difference being, there would be two elements for each vertex in the array of vertices—one when the vertex is head of an edge and another when it is a tail. This is shown in Figure 8-5.

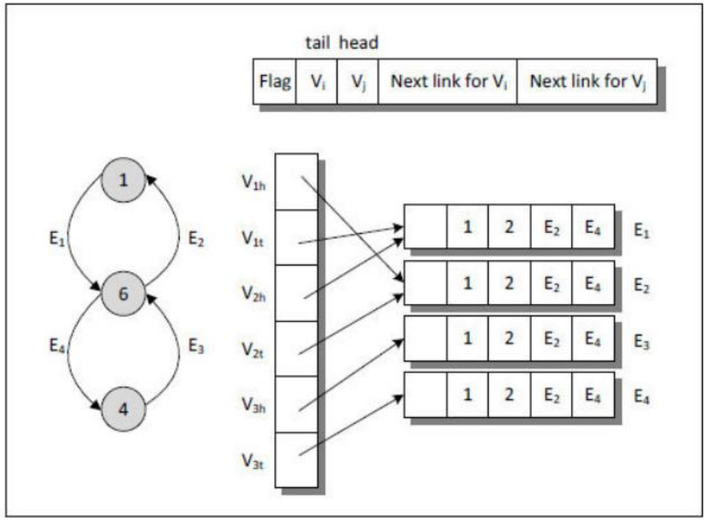
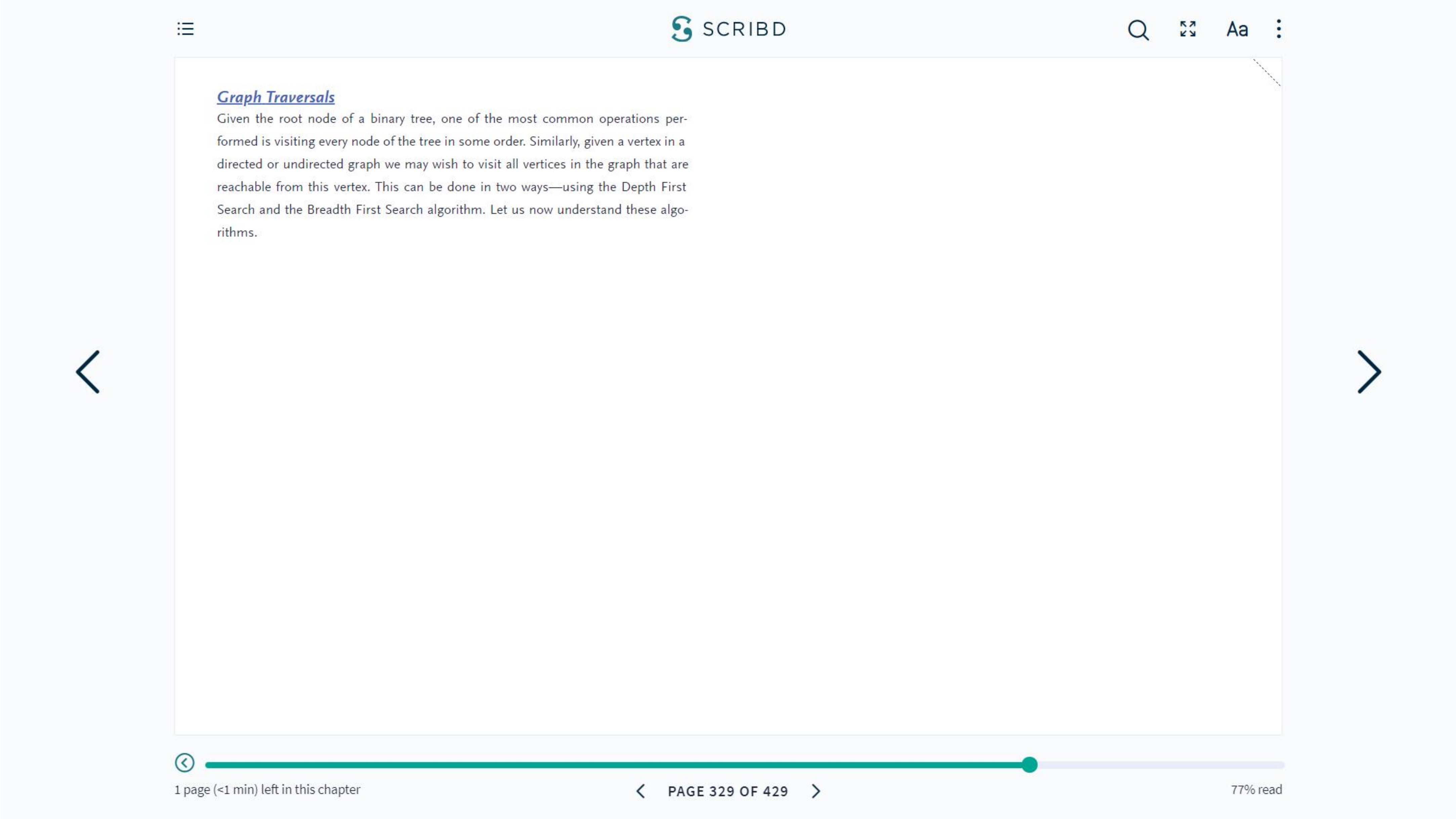


Figure 8-5. *Adjacency multi-lists for directed graph.*

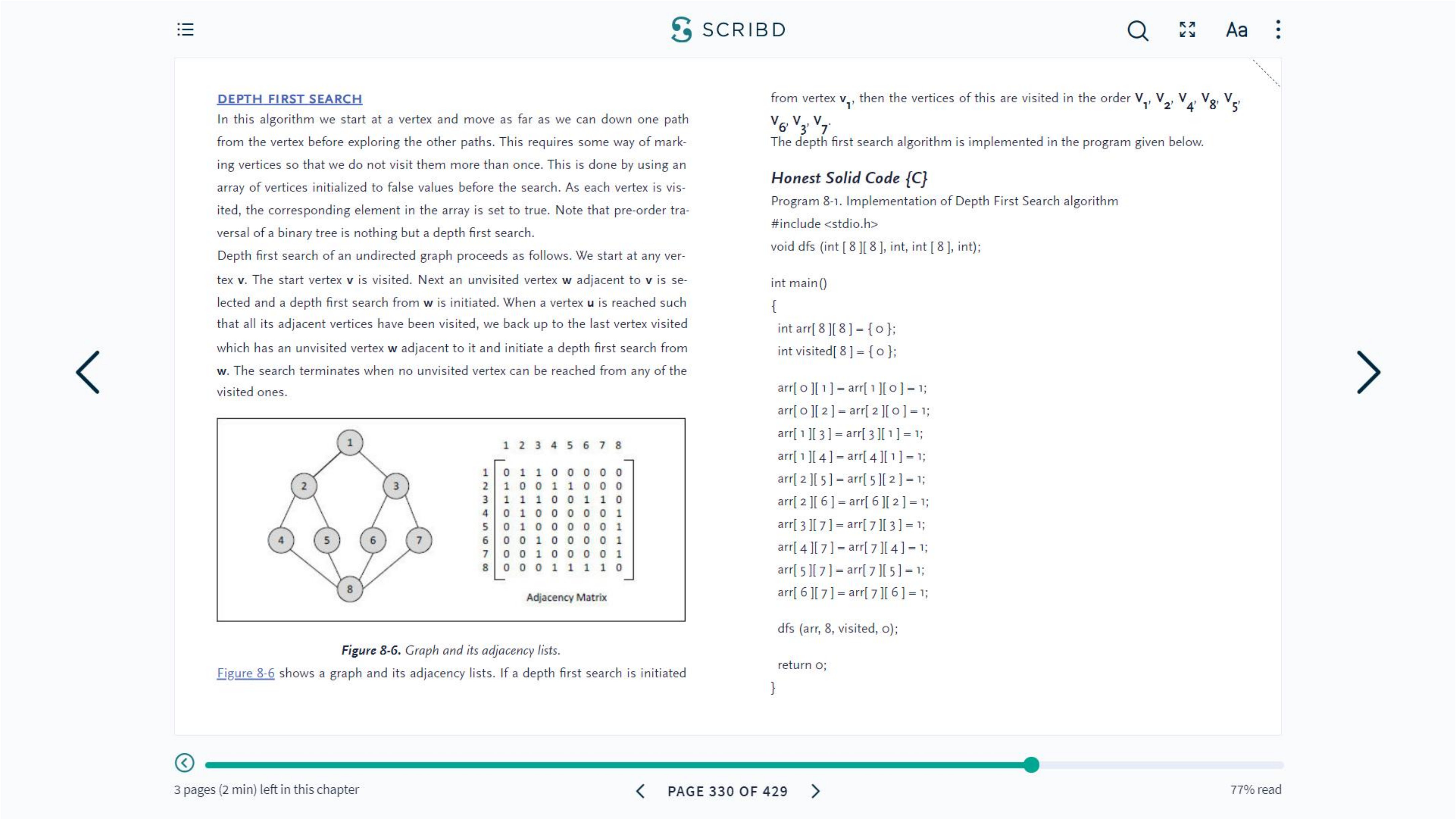
If we traverse the lists shows in Figure 8-5 for each element of the vertex array, then we the sequence of incident edges for each vertex would be as follows. These sequences are given below.

V_{1h} : E_2
V_{1t} : E_1
V_{2h}: E_1, E_4
V_{2t} : E_2, E_3
V_{3h}: E_3
V_{3t} : E_4



Graph Traversals

Given the root node of a binary tree, one of the most common operations performed is visiting every node of the tree in some order. Similarly, given a vertex in a directed or undirected graph we may wish to visit all vertices in the graph that are reachable from this vertex. This can be done in two ways—using the Depth First Search and the Breadth First Search algorithm. Let us now understand these algorithms.



DEPTH FIRST SEARCH

In this algorithm we start at a vertex and move as far as we can down one path from the vertex before exploring the other paths. This requires some way of marking vertices so that we do not visit them more than once. This is done by using an array of vertices initialized to false values before the search. As each vertex is visited, the corresponding element in the array is set to true. Note that pre-order traversal of a binary tree is nothing but a depth first search.

Depth first search of an undirected graph proceeds as follows. We start at any vertex **v**. The start vertex **v** is visited. Next an unvisited vertex **w** adjacent to **v** is selected and a depth first search from **w** is initiated. When a vertex **u** is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited which has an unvisited vertex **w** adjacent to it and initiate a depth first search from **w**. The search terminates when no unvisited vertex can be reached from any of the visited ones.

$$\begin{array}{c|cccccccc|} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 3 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 4 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 5 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 6 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 7 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 8 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{array}$$

Adjacency Matrix

Figure 8-6. *Graph and its adjacency lists.*

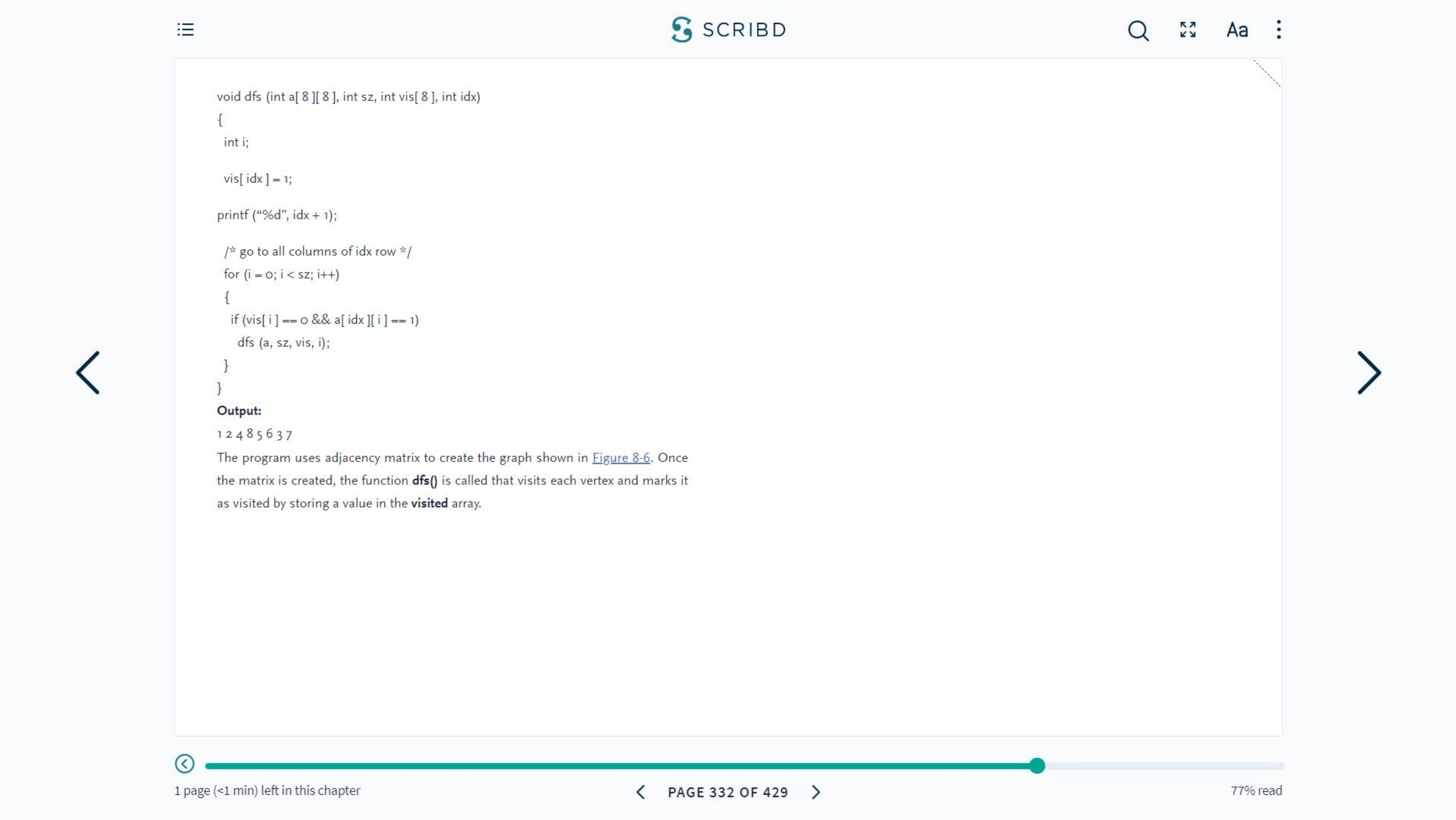
Figure 8-6 shows a graph and its adjacency lists. If a depth first search is initiated from vertex $\mathbf{v_1}$, then the vertices of this are visited in the order $\mathbf{V_1}$, $\mathbf{V_2}$, $\mathbf{V_4}$, $\mathbf{V_8}$, $\mathbf{V_5}$, $\mathbf{V_6}$, $\mathbf{V_3}$, $\mathbf{V_7}$.

The depth first search algorithm is implemented in the program given below.

Honest Solid Code {C}

Program 8-1. Implementation of Depth First Search algorithm

```
#include <stdio.h>
void dfs (int [ 8 ][ 8 ], int, int [ 8 ], int);

int main()
{
 int arr[ 8 ][ 8 ] = { 0 };
 int visited[ 8 ] = { 0 };

 arr[ 0 ][ 1 ] = arr[ 1 ][ 0 ] = 1;
 arr[ 0 ][ 2 ] = arr[ 2 ][ 0 ] = 1;
 arr[ 1 ][ 3 ] = arr[ 3 ][ 1 ] = 1;
 arr[ 1 ][ 4 ] = arr[ 4 ][ 1 ] = 1;
 arr[ 2 ][ 5 ] = arr[ 5 ][ 2 ] = 1;
 arr[ 2 ][ 6 ] = arr[ 6 ][ 2 ] = 1;
 arr[ 3 ][ 7 ] = arr[ 7 ][ 3 ] = 1;
 arr[ 4 ][ 7 ] = arr[ 7 ][ 4 ] = 1;
 arr[ 5 ][ 7 ] = arr[ 7 ][ 5 ] = 1;
 arr[ 6 ][ 7 ] = arr[ 7 ][ 6 ] = 1;

 dfs (arr, 8, visited, 0);

 return 0;
}
```

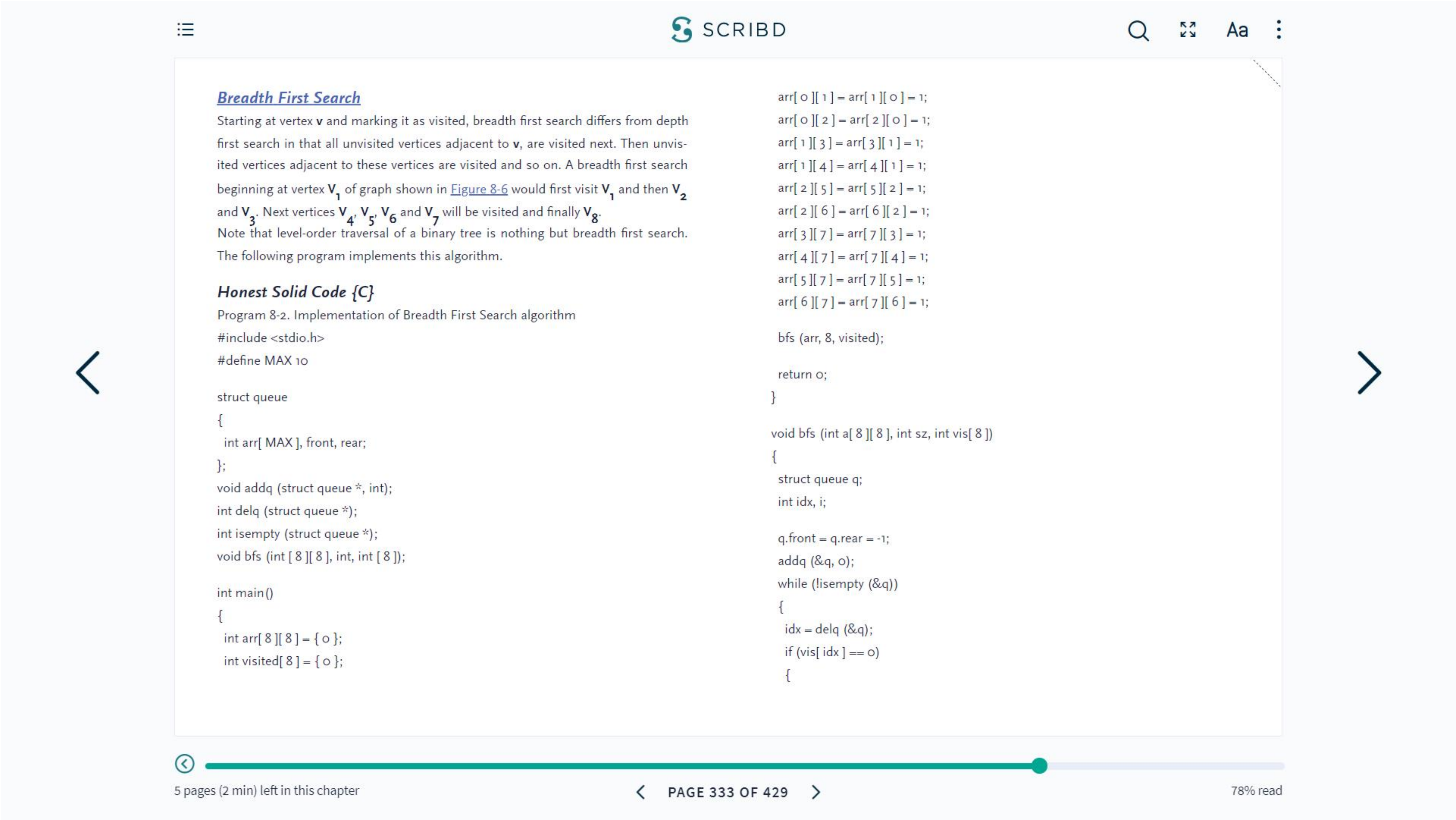


```
void dfs (int a[ 8 ][ 8 ], int sz, int vis[ 8 ], int idx)
{
 int i;

 vis[ idx ] = 1;

printf (“%d”, idx + 1);

 /* go to all columns of idx row */
 for (i = 0; i < sz; i++)
 {
  if (vis[ i ] == 0 && a[ idx ][ i ] == 1)
    dfs (a, sz, vis, i);
 }
}
```

Output:

```
1 2 4 8 5 6 3 7
```

The program uses adjacency matrix to create the graph shown in Figure 8-6. Once the matrix is created, the function **dfs()** is called that visits each vertex and marks it as visited by storing a value in the **visited** array.



Breadth First Search

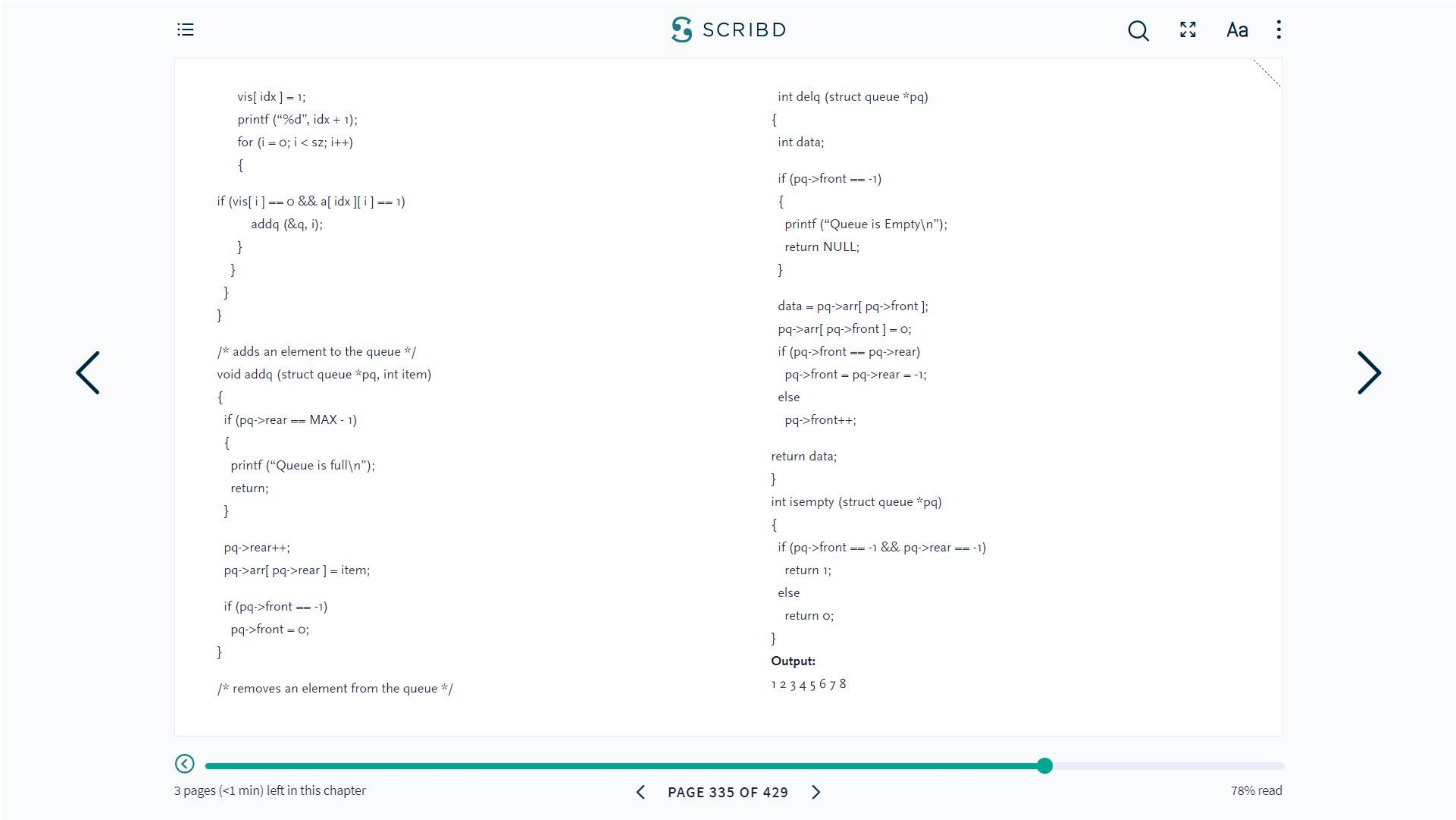
Starting at vertex **v** and marking it as visited, breadth first search differs from depth first search in that all unvisited vertices adjacent to **v**, are visited next. Then unvisited vertices adjacent to these vertices are visited and so on. A breadth first search beginning at vertex V_1 of graph shown in Figure 8-6 would first visit V_1 and then V_2 and V_3. Next vertices V_4, V_5, V_6 and V_7 will be visited and finally V_8.

Note that level-order traversal of a binary tree is nothing but breadth first search. The following program implements this algorithm.

Honest Solid Code {C}

Program 8-2. Implementation of Breadth First Search algorithm

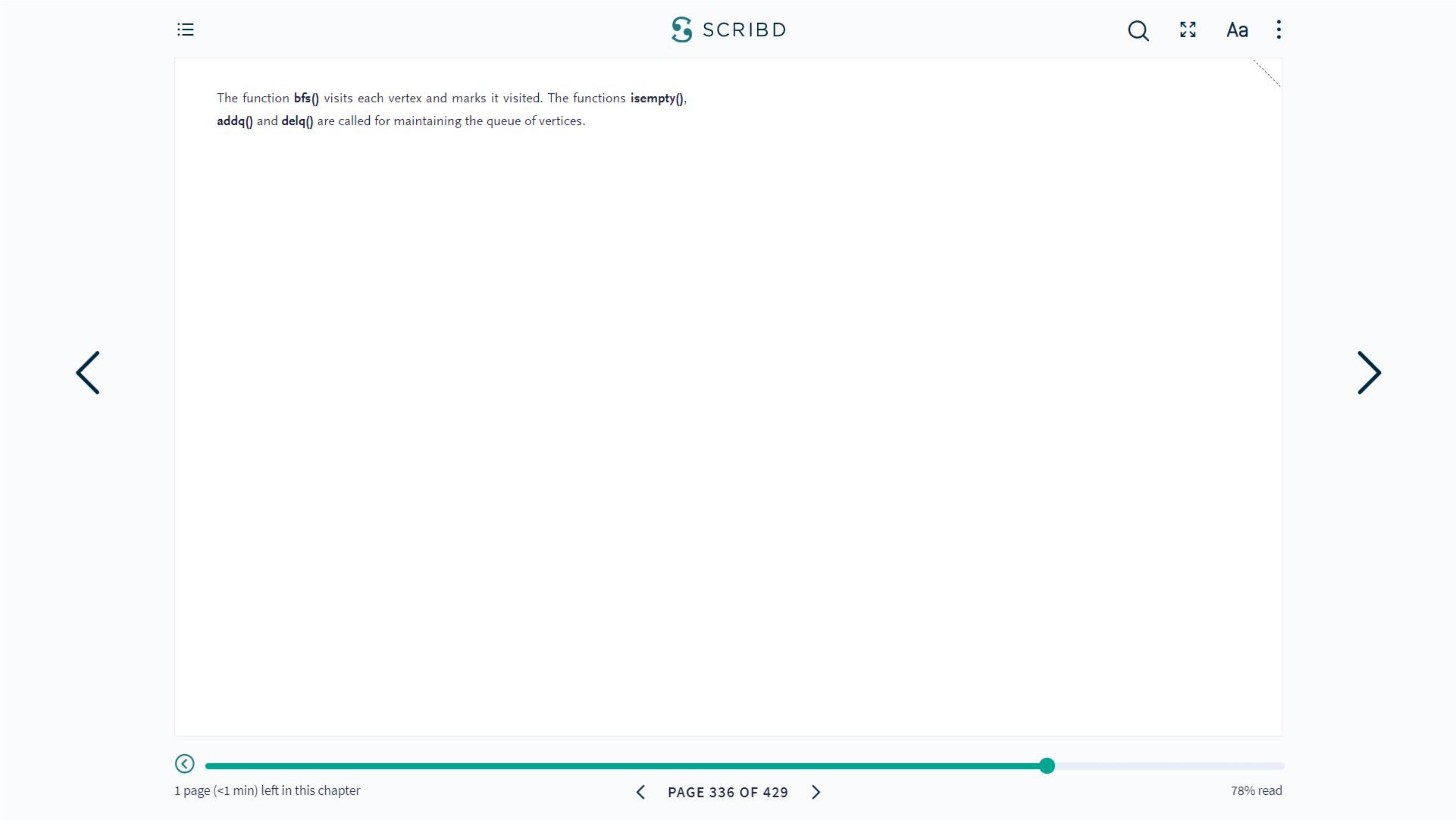
```
#include <stdio.h>
#define MAX 10

struct queue
{
 int arr[ MAX ], front, rear;
};
void addq (struct queue *, int);
int delq (struct queue *);
int isempty (struct queue *);
void bfs (int [ 8 ][ 8 ], int, int [ 8 ]);

int main()
{
 int arr[ 8 ][ 8 ] = { 0 };
 int visited[ 8 ] = { 0 };
 arr[ 0 ][ 1 ] = arr[ 1 ][ 0 ] = 1;
 arr[ 0 ][ 2 ] = arr[ 2 ][ 0 ] = 1;
 arr[ 1 ][ 3 ] = arr[ 3 ][ 1 ] = 1;
 arr[ 1 ][ 4 ] = arr[ 4 ][ 1 ] = 1;
 arr[ 2 ][ 5 ] = arr[ 5 ][ 2 ] = 1;
 arr[ 2 ][ 6 ] = arr[ 6 ][ 2 ] = 1;
 arr[ 3 ][ 7 ] = arr[ 7 ][ 3 ] = 1;
 arr[ 4 ][ 7 ] = arr[ 7 ][ 4 ] = 1;
 arr[ 5 ][ 7 ] = arr[ 7 ][ 5 ] = 1;
 arr[ 6 ][ 7 ] = arr[ 7 ][ 6 ] = 1;

 bfs (arr, 8, visited);

 return 0;
}

void bfs (int a[ 8 ][ 8 ], int sz, int vis[ 8 ])
{
 struct queue q;
 int idx, i;

 q.front = q.rear = -1;
 addq (&q, 0);
 while (!isempty (&q))
 {
  idx = delq (&q);
  if (vis[ idx ] == 0)
  {
```



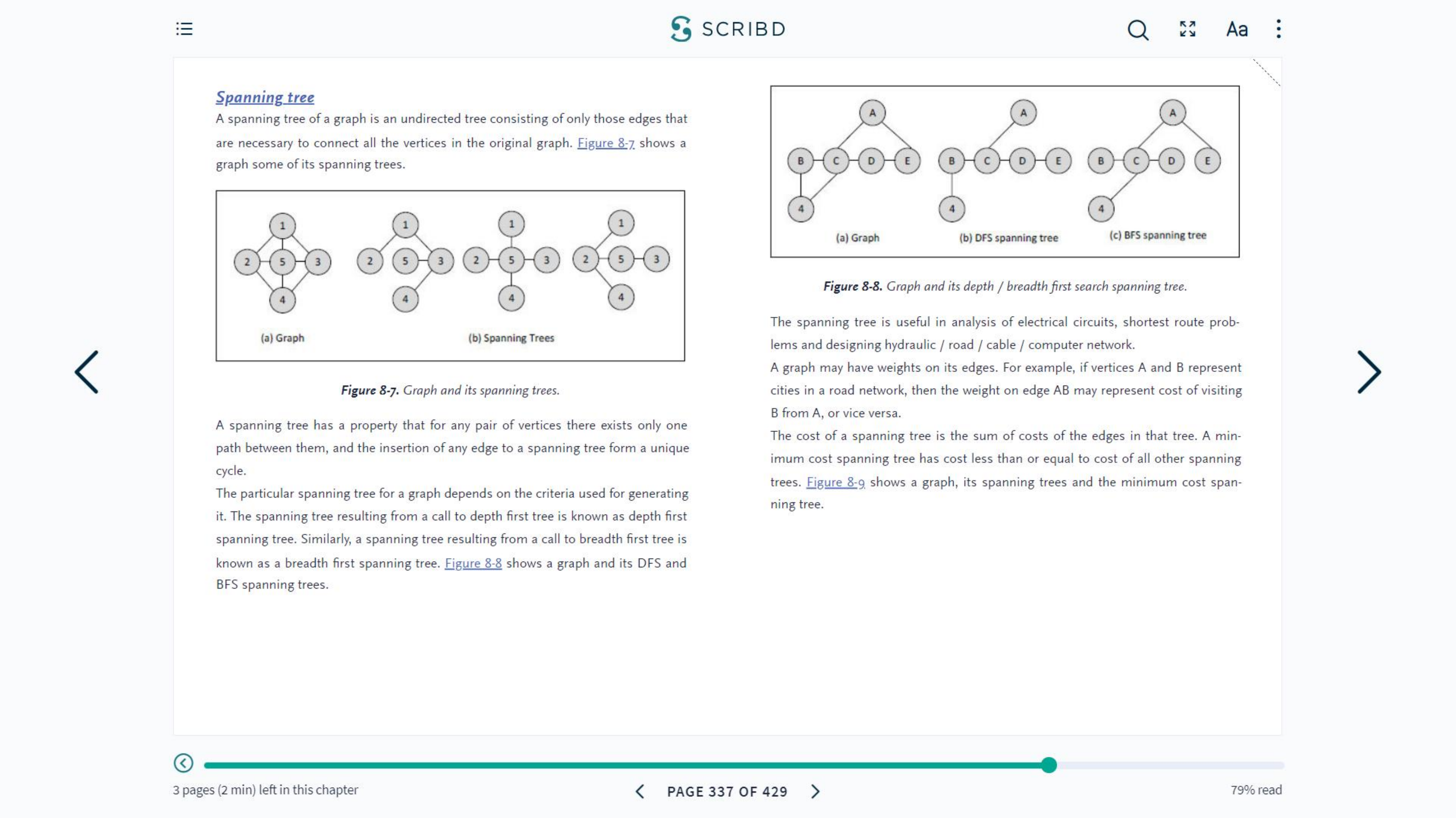
```
    vis[ idx ] = 1;
    printf (“%d”, idx + 1);
    for (i = 0; i < sz; i++)
    {
if (vis[ i ] == 0 && a[ idx ][ i ] == 1)
      addq (&q, i);
    }
   }
  }
}

/* adds an element to the queue */
void addq (struct queue *pq, int item)
{
 if (pq->rear == MAX - 1)
 {
  printf (“Queue is full\n”);
  return;
 }

 pq->rear++;
 pq->arr[ pq->rear ] = item;

 if (pq->front == -1)
  pq->front = 0;
}

/* removes an element from the queue */
 int delq (struct queue *pq)
{
 int data;

 if (pq->front == -1)
 {
  printf (“Queue is Empty\n”);
  return NULL;
 }

 data = pq->arr[ pq->front ];
 pq->arr[ pq->front ] = 0;
 if (pq->front == pq->rear)
  pq->front = pq->rear = -1;
 else
  pq->front++;

return data;
}
int isempty (struct queue *pq)
{
 if (pq->front == -1 && pq->rear == -1)
  return 1;
 else
  return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8
```



The function **bfs()** visits each vertex and marks it visited. The functions **isempty()**, **addq()** and **delq()** are called for maintaining the queue of vertices.



Spanning tree

A spanning tree of a graph is an undirected tree consisting of only those edges that are necessary to connect all the vertices in the original graph. Figure 8-7 shows a graph some of its spanning trees.

Figure 8-7. Graph and its spanning trees.

A spanning tree has a property that for any pair of vertices there exists only one path between them, and the insertion of any edge to a spanning tree form a unique cycle.

The particular spanning tree for a graph depends on the criteria used for generating it. The spanning tree resulting from a call to depth first tree is known as depth first spanning tree. Similarly, a spanning tree resulting from a call to breadth first tree is known as a breadth first spanning tree. Figure 8-8 shows a graph and its DFS and BFS spanning trees.

Figure 8-8. Graph and its depth / breadth first search spanning tree.

The spanning tree is useful in analysis of electrical circuits, shortest route problems and designing hydraulic / road / cable / computer network.

A graph may have weights on its edges. For example, if vertices A and B represent cities in a road network, then the weight on edge AB may represent cost of visiting B from A, or vice versa.

The cost of a spanning tree is the sum of costs of the edges in that tree. A minimum cost spanning tree has cost less than or equal to cost of all other spanning trees. Figure 8-9 shows a graph, its spanning trees and the minimum cost spanning tree.

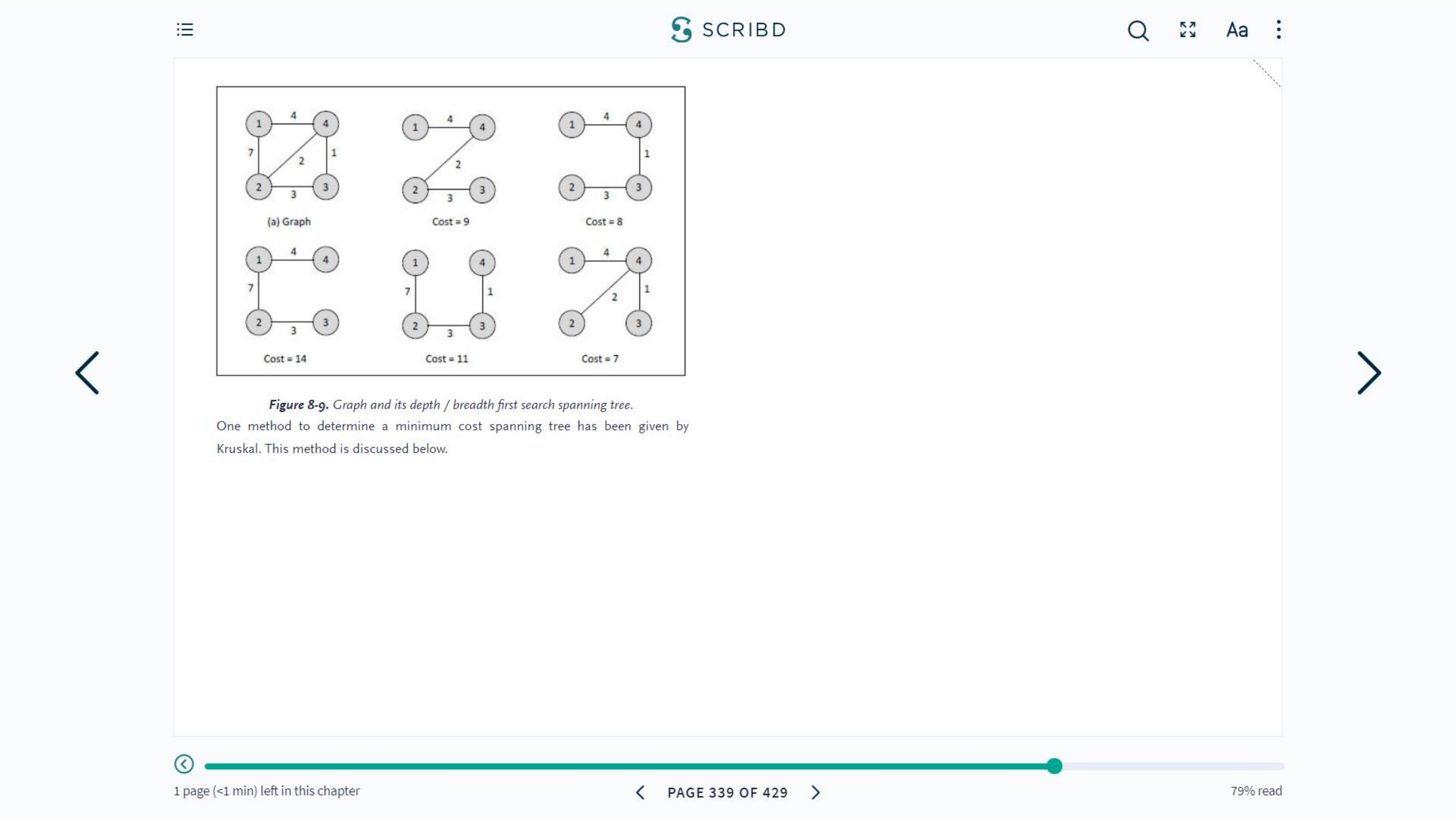
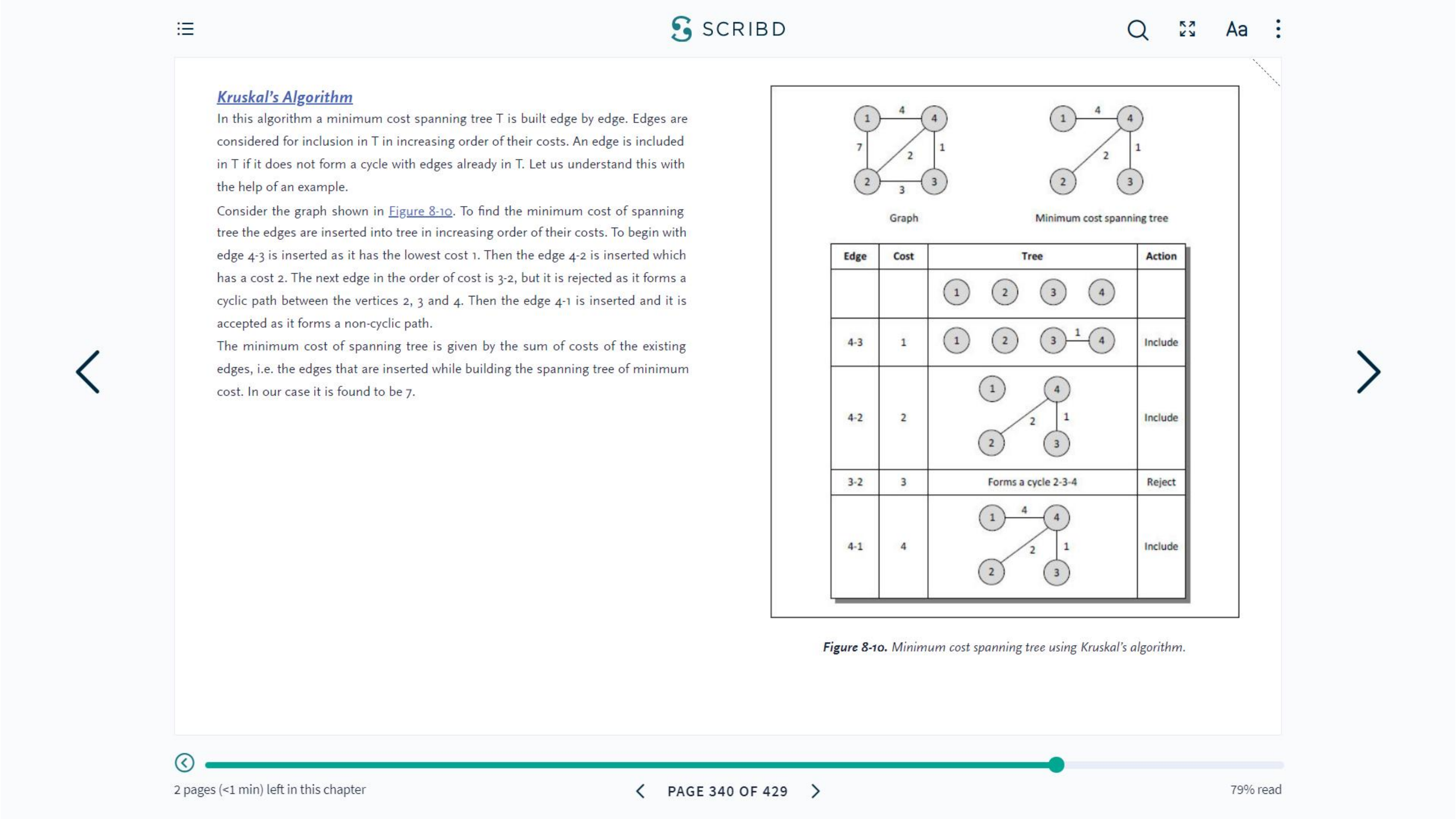


Figure 8-9. Graph and its depth / breadth first search spanning tree.

One method to determine a minimum cost spanning tree has been given by Kruskal. This method is discussed below.



Kruskal's Algorithm

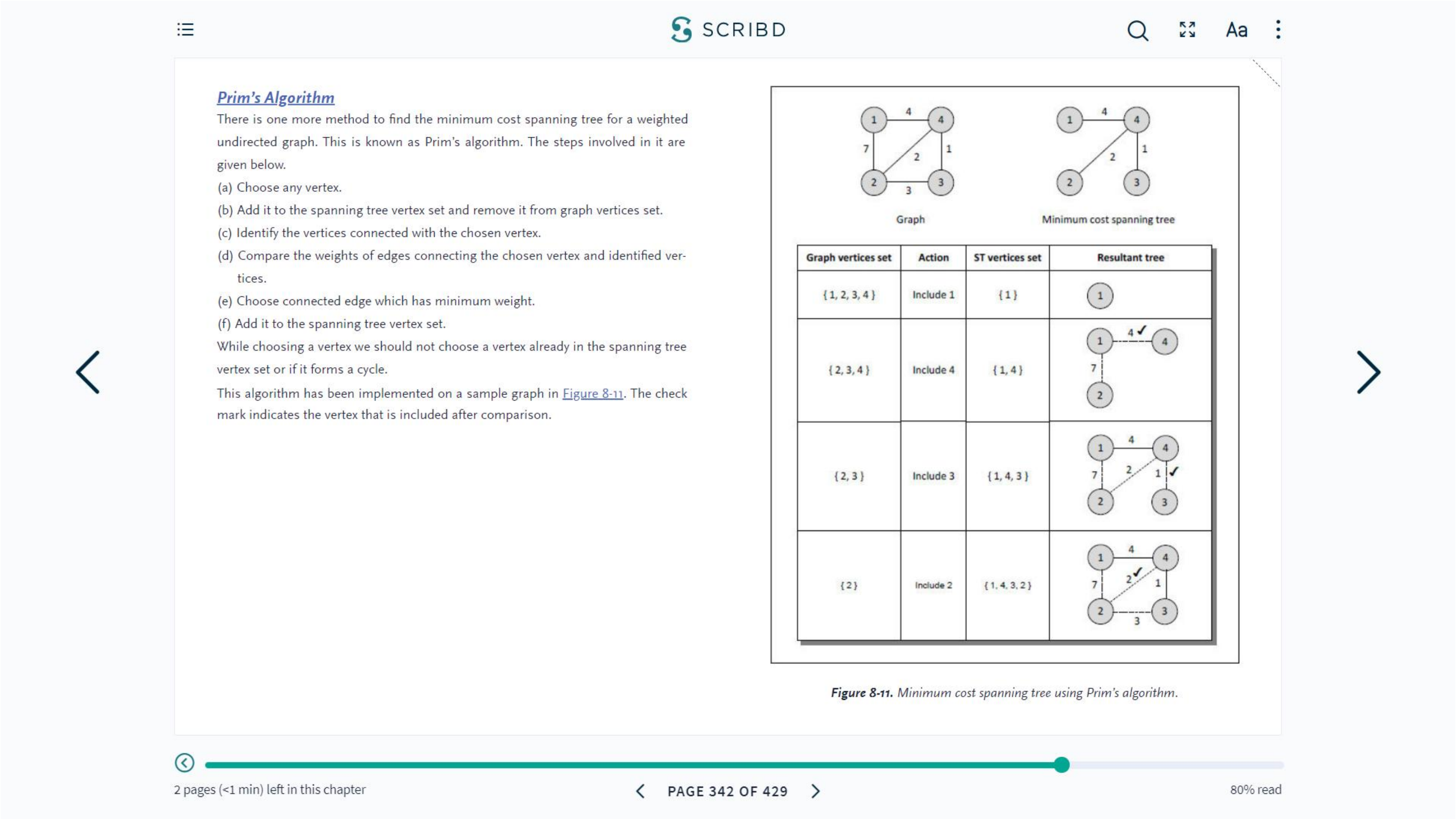
In this algorithm a minimum cost spanning tree T is built edge by edge. Edges are considered for inclusion in T in increasing order of their costs. An edge is included in T if it does not form a cycle with edges already in T. Let us understand this with the help of an example.

Consider the graph shown in Figure 8-10. To find the minimum cost of spanning tree the edges are inserted into tree in increasing order of their costs. To begin with edge 4-3 is inserted as it has the lowest cost 1. Then the edge 4-2 is inserted which has a cost 2. The next edge in the order of cost is 3-2, but it is rejected as it forms a cyclic path between the vertices 2, 3 and 4. Then the edge 4-1 is inserted and it is accepted as it forms a non-cyclic path.

The minimum cost of spanning tree is given by the sum of costs of the existing edges, i.e. the edges that are inserted while building the spanning tree of minimum cost. In our case it is found to be 7.

Edge	Cost	Tree	Action
4-3	1		Include
4-2	2		Include
3-2	3	Forms a cycle 2-3-4	Reject
4-1	4		Include

Figure 8-10. *Minimum cost spanning tree using Kruskal's algorithm.*



Prim's Algorithm

There is one more method to find the minimum cost spanning tree for a weighted undirected graph. This is known as Prim's algorithm. The steps involved in it are given below.

(a) Choose any vertex.

(b) Add it to the spanning tree vertex set and remove it from graph vertices set.

(c) Identify the vertices connected with the chosen vertex.

(d) Compare the weights of edges connecting the chosen vertex and identified vertices.

(e) Choose connected edge which has minimum weight.

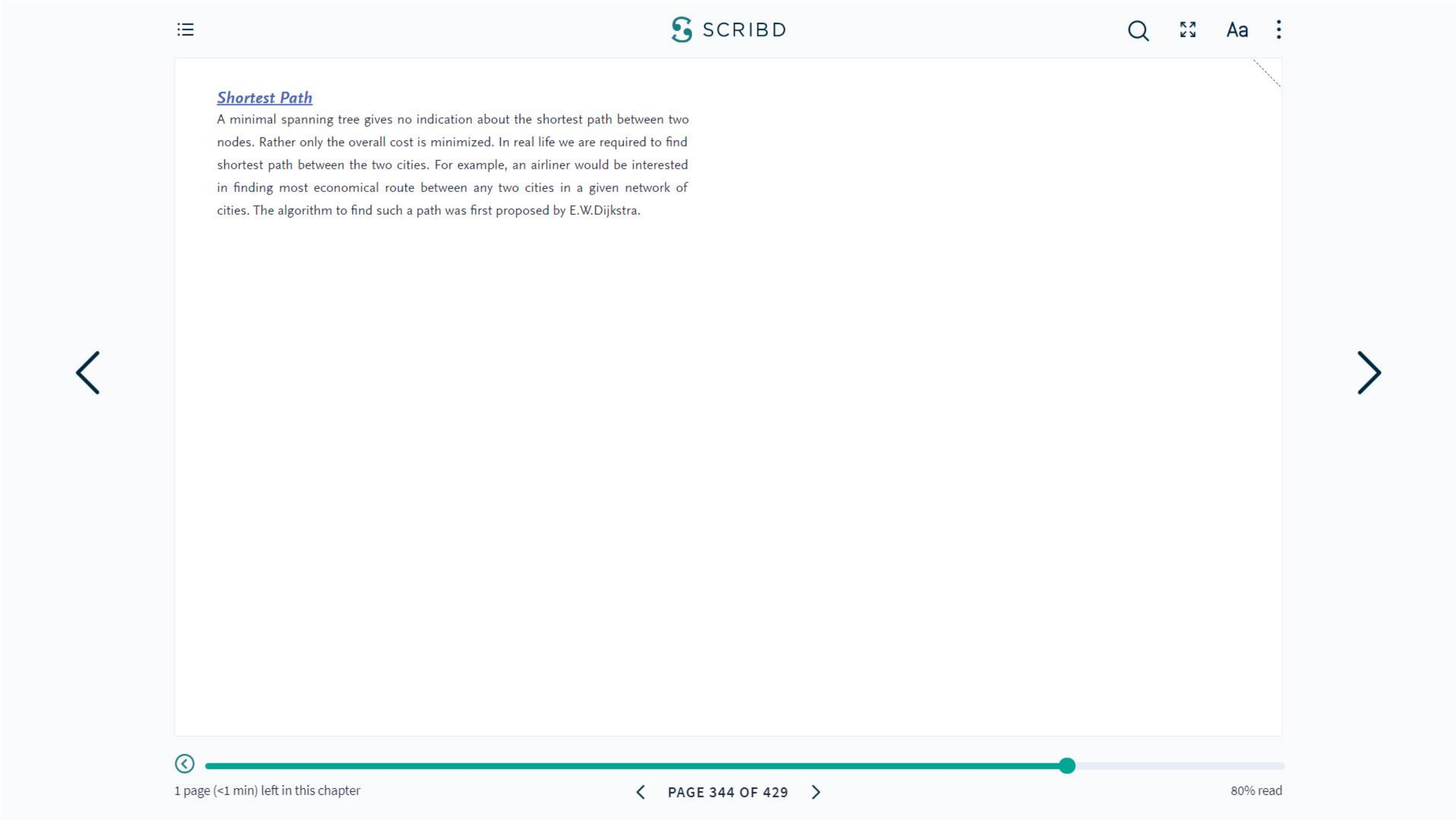
(f) Add it to the spanning tree vertex set.

While choosing a vertex we should not choose a vertex already in the spanning tree vertex set or if it forms a cycle.

This algorithm has been implemented on a sample graph in Figure 8-11. The check mark indicates the vertex that is included after comparison.

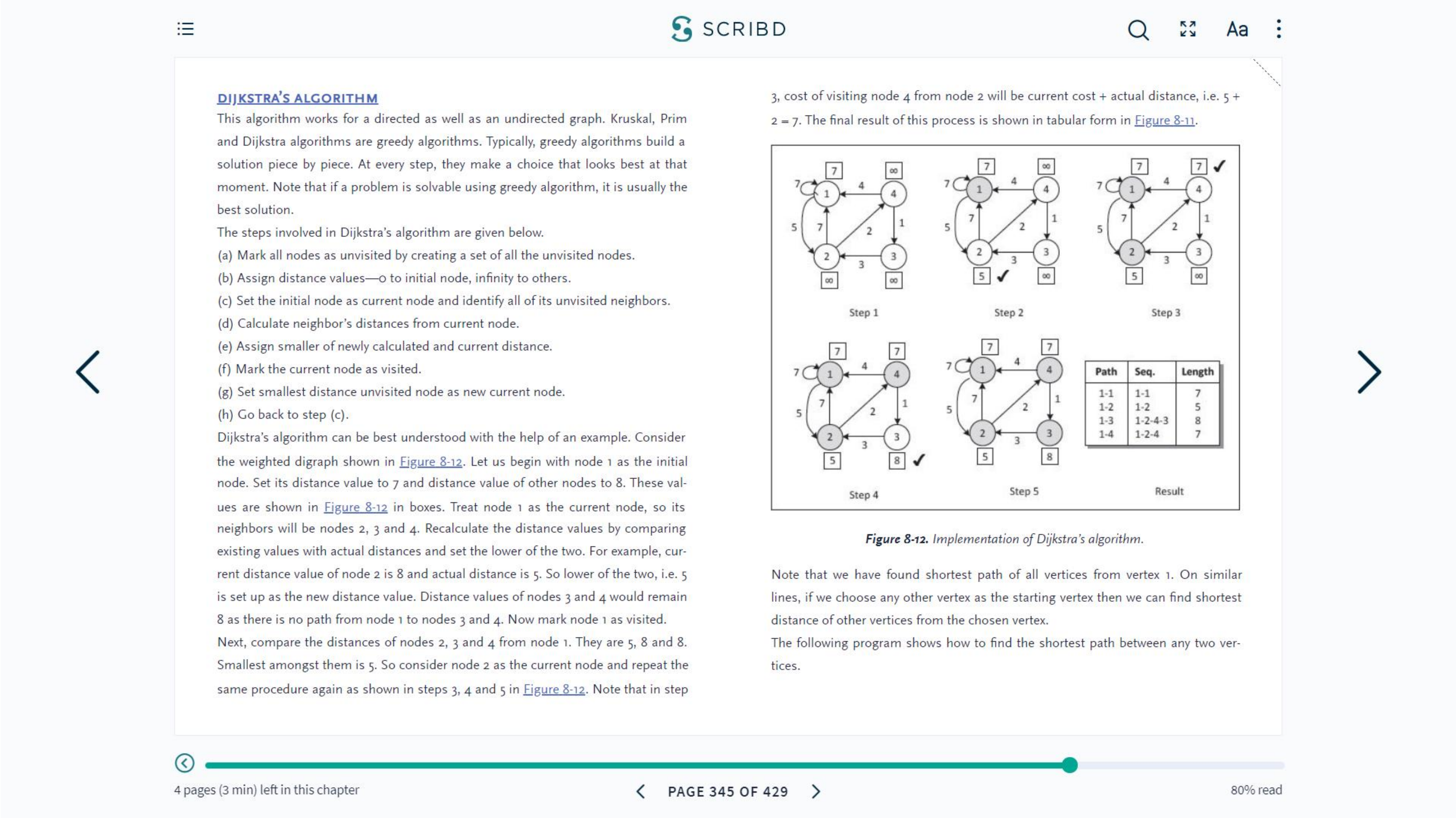
Graph vertices set	Action	ST vertices set	Resultant tree
{ 1, 2, 3, 4 }	Include 1	{ 1 }	1
{ 2, 3, 4 }	Include 4	{ 1, 4 }	1 4✓ 4 7 2
{ 2, 3 }	Include 3	{ 1, 4, 3 }	1 4 4 7 2 1✓ 2 3
{ 2 }	Include 2	{ 1, 4, 3, 2 }	1 4 4 7 2✓ 1 2 3 3

Figure 8-11. *Minimum cost spanning tree using Prim's algorithm.*



Shortest Path

A minimal spanning tree gives no indication about the shortest path between two nodes. Rather only the overall cost is minimized. In real life we are required to find shortest path between the two cities. For example, an airliner would be interested in finding most economical route between any two cities in a given network of cities. The algorithm to find such a path was first proposed by E.W.Dijkstra.



DIJKSTRA'S ALGORITHM

This algorithm works for a directed as well as an undirected graph. Kruskal, Prim and Dijkstra algorithms are greedy algorithms. Typically, greedy algorithms build a solution piece by piece. At every step, they make a choice that looks best at that moment. Note that if a problem is solvable using greedy algorithm, it is usually the best solution.

The steps involved in Dijkstra's algorithm are given below.

(a) Mark all nodes as unvisited by creating a set of all the unvisited nodes.

(b) Assign distance values—0 to initial node, infinity to others.

(c) Set the initial node as current node and identify all of its unvisited neighbors.

(d) Calculate neighbor's distances from current node.

(e) Assign smaller of newly calculated and current distance.

(f) Mark the current node as visited.

(g) Set smallest distance unvisited node as new current node.

(h) Go back to step (c).

Dijkstra's algorithm can be best understood with the help of an example. Consider the weighted digraph shown in Figure 8-12. Let us begin with node 1 as the initial node. Set its distance value to 7 and distance value of other nodes to 8. These values are shown in Figure 8-12 in boxes. Treat node 1 as the current node, so its neighbors will be nodes 2, 3 and 4. Recalculate the distance values by comparing existing values with actual distances and set the lower of the two. For example, current distance value of node 2 is 8 and actual distance is 5. So lower of the two, i.e. 5 is set up as the new distance value. Distance values of nodes 3 and 4 would remain 8 as there is no path from node 1 to nodes 3 and 4. Now mark node 1 as visited.

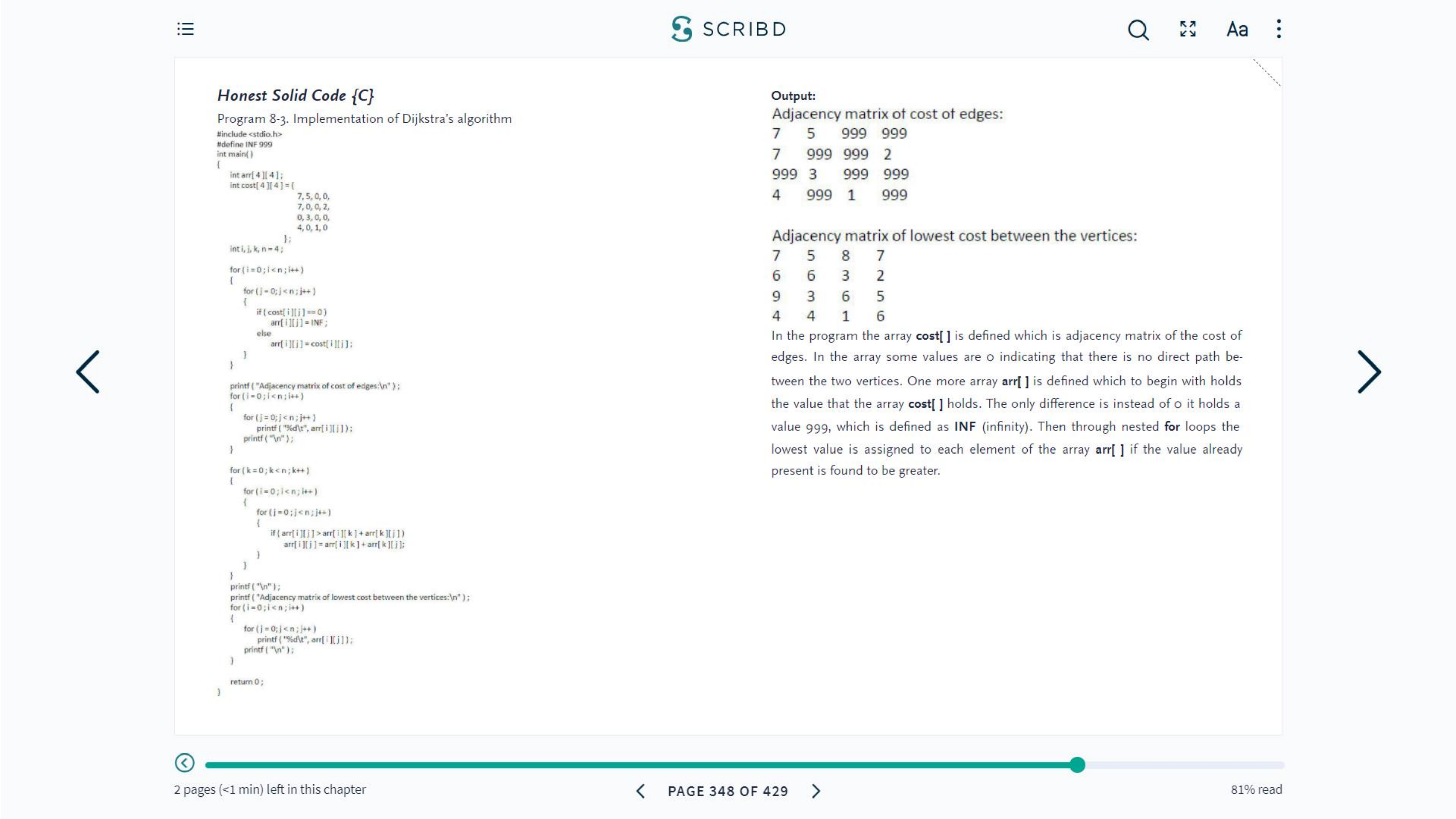
Next, compare the distances of nodes 2, 3 and 4 from node 1. They are 5, 8 and 8. Smallest amongst them is 5. So consider node 2 as the current node and repeat the same procedure again as shown in steps 3, 4 and 5 in Figure 8-12. Note that in step 3, cost of visiting node 4 from node 2 will be current cost + actual distance, i.e. 5 + 2 = 7. The final result of this process is shown in tabular form in Figure 8-11.

Path	Seq.	Length
1-1	1-1	7
1-2	1-2	5
1-3	1-2-4-3	8
1-4	1-2-4	7

Figure 8-12. *Implementation of Dijkstra's algorithm.*

Note that we have found shortest path of all vertices from vertex 1. On similar lines, if we choose any other vertex as the starting vertex then we can find shortest distance of other vertices from the chosen vertex.

The following program shows how to find the shortest path between any two vertices.



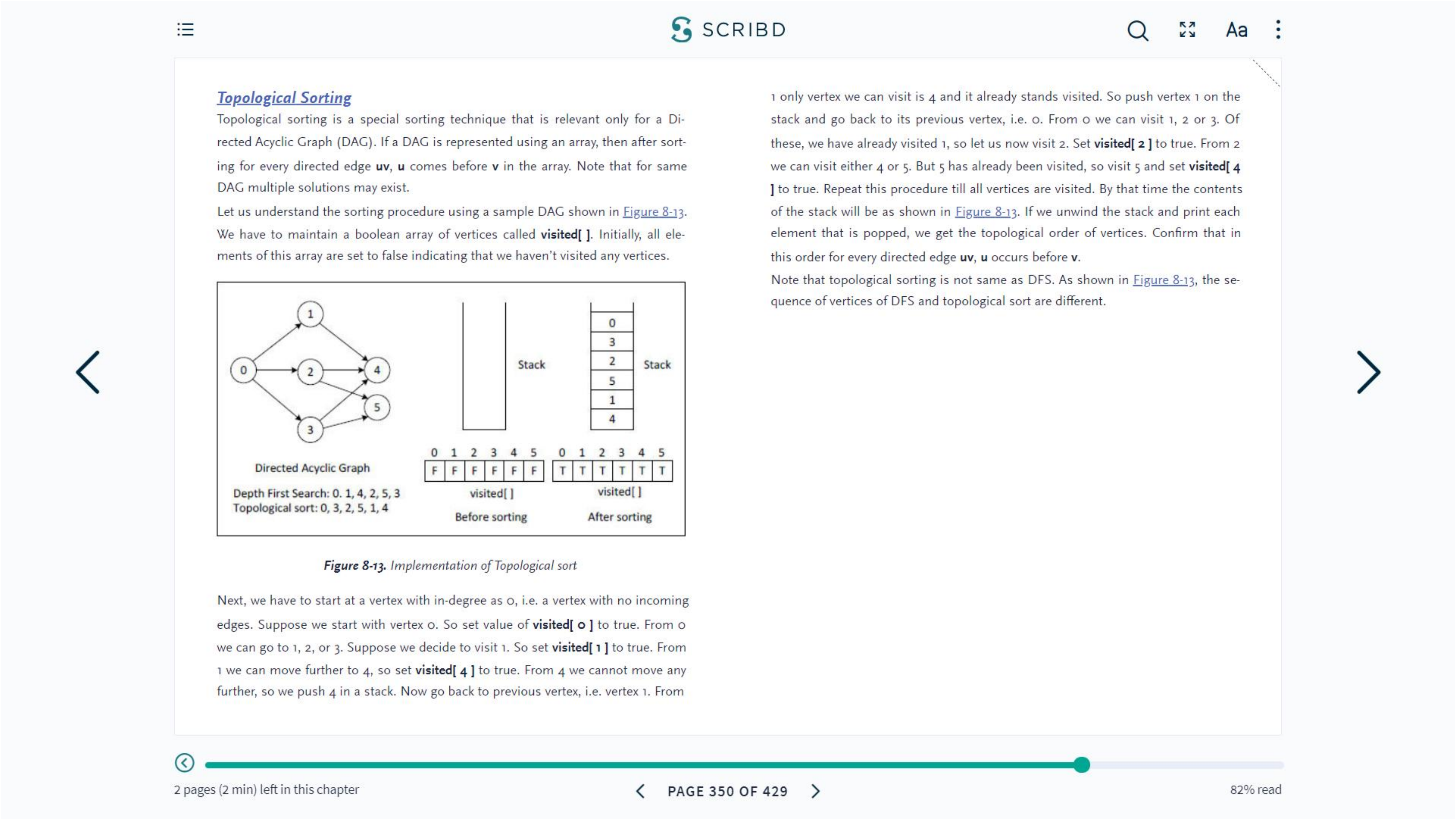
Program 8-3. Implementation of Dijkstra's algorithm

```c
#include <stdio.h>
#define INF 999
int main( )
{
    int arr[ 4 ][ 4 ] ;
    int cost[ 4 ][ 4 ] = {
                    7, 5, 0, 0,
                    7, 0, 0, 2,
                    0, 3, 0, 0,
                    4, 0, 1, 0
                };
    int i, j, k, n = 4 ;

    for ( i = 0 ; i < n ; i++ )
    {
        for ( j = 0; j < n ; j++ )
        {
            if ( cost[ i ][ j ] == 0 )
                arr[ i ][ j ] = INF ;
            else
                arr[ i ][ j ] = cost[ i ][ j ] ;
        }
    }

    printf ( "Adjacency matrix of cost of edges:\n" ) ;
    for ( i = 0 ; i < n ; i++ )
    {
        for ( j = 0; j < n ; j++ )
            printf ( "%d\t", arr[ i ][ j ] ) ;
        printf ( "\n" ) ;
    }

    for ( k = 0 ; k < n ; k++ )
    {
        for ( i = 0 ; i < n ; i++ )
        {
            for ( j = 0 ; j < n ; j++ )
            {
                if ( arr[ i ][ j ] > arr[ i ][ k ] + arr[ k ][ j ] )
                    arr[ i ][ j ] = arr[ i ][ k ] + arr[ k ][ j ];
            }
        }
    }
    printf ( "\n" ) ;
    printf ( "Adjacency matrix of lowest cost between the vertices:\n" ) ;
    for ( i = 0 ; i < n ; i++ )
    {
        for ( j = 0; j < n ; j++ )
            printf ( "%d\t", arr[ i ][ j ] ) ;
        printf ( "\n" ) ;
    }

    return 0 ;
}
```

Output:

```
Adjacency matrix of cost of edges:
7    5    999  999
7    999  999  2
999  3    999  999
4    999  1    999

Adjacency matrix of lowest cost between the vertices:
7    5    8    7
6    6    3    2
9    3    6    5
4    4    1    6
```

In the program the array **cost[ ]** is defined which is adjacency matrix of the cost of edges. In the array some values are 0 indicating that there is no direct path between the two vertices. One more array **arr[ ]** is defined which to begin with holds the value that the array **cost[ ]** holds. The only difference is instead of 0 it holds a value 999, which is defined as **INF** (infinity). Then through nested **for** loops the lowest value is assigned to each element of the array **arr[ ]** if the value already present is found to be greater.



Topological Sorting

Topological sorting is a special sorting technique that is relevant only for a Directed Acyclic Graph (DAG). If a DAG is represented using an array, then after sorting for every directed edge **uv**, **u** comes before **v** in the array. Note that for same DAG multiple solutions may exist.

Let us understand the sorting procedure using a sample DAG shown in Figure 8-13. We have to maintain a boolean array of vertices called **visited[ ]**. Initially, all elements of this array are set to false indicating that we haven't visited any vertices.

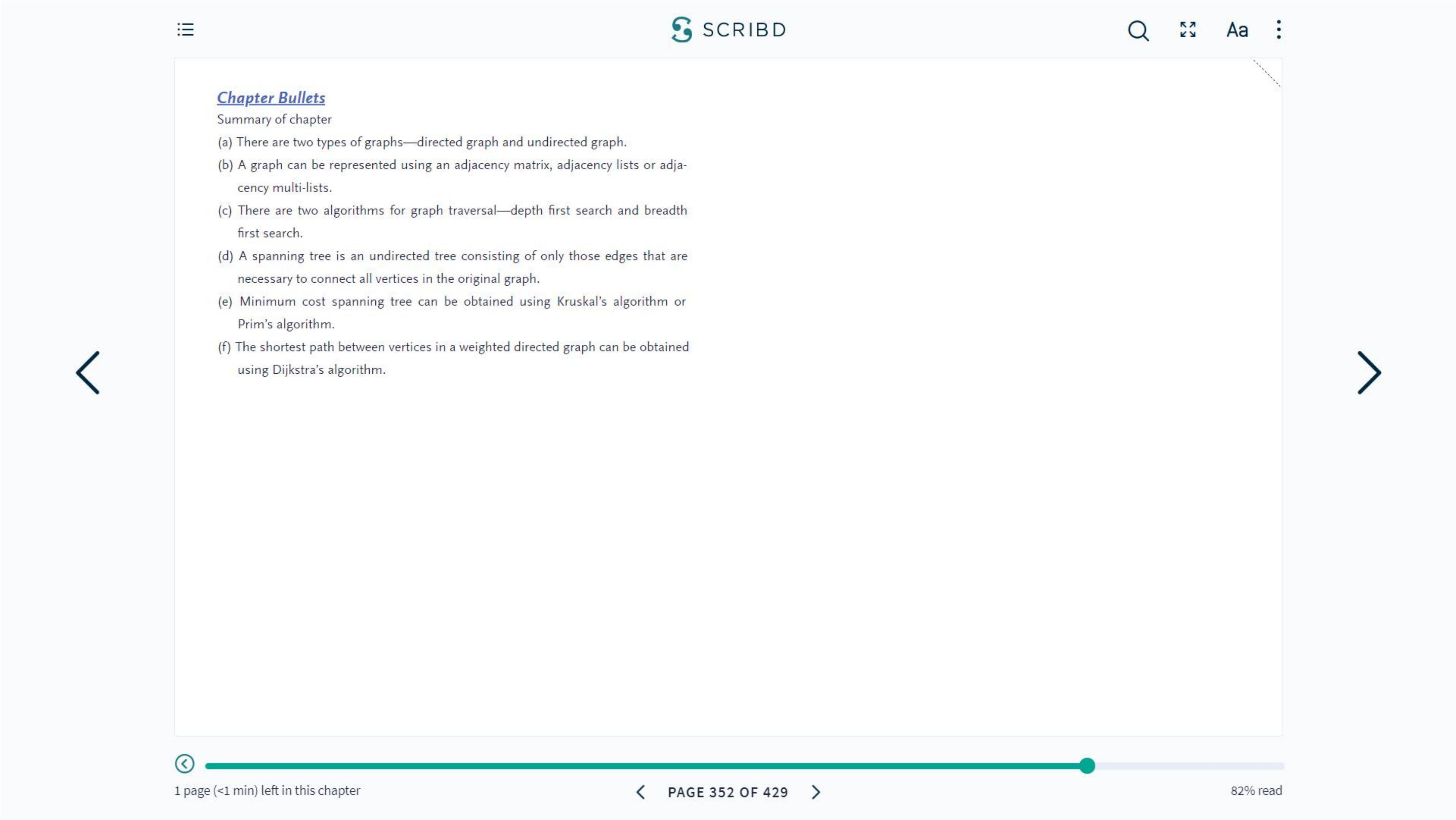
0	1	2	3	4	5
F	F	F	F	F	F

0	1	2	3	4	5
T	T	T	T	T	T

Figure 8-13. *Implementation of Topological sort*

Next, we have to start at a vertex with in-degree as 0, i.e. a vertex with no incoming edges. Suppose we start with vertex 0. So set value of **visited[0]** to true. From 0 we can go to 1, 2, or 3. Suppose we decide to visit 1. So set **visited[1]** to true. From 1 we can move further to 4, so set **visited[4]** to true. From 4 we cannot move any further, so we push 4 in a stack. Now go back to previous vertex, i.e. vertex 1. From 1 only vertex we can visit is 4 and it already stands visited. So push vertex 1 on the stack and go back to its previous vertex, i.e. 0. From 0 we can visit 1, 2 or 3. Of these, we have already visited 1, so let us now visit 2. Set **visited[2]** to true. From 2 we can visit either 4 or 5. But 5 has already been visited, so visit 5 and set **visited[4]** to true. Repeat this procedure till all vertices are visited. By that time the contents of the stack will be as shown in Figure 8-13. If we unwind the stack and print each element that is popped, we get the topological order of vertices. Confirm that in this order for every directed edge **uv**, **u** occurs before **v**.

Note that topological sorting is not same as DFS. As shown in Figure 8-13, the sequence of vertices of DFS and topological sort are different.



Chapter Bullets

Summary of chapter

(a) There are two types of graphs—directed graph and undirected graph.

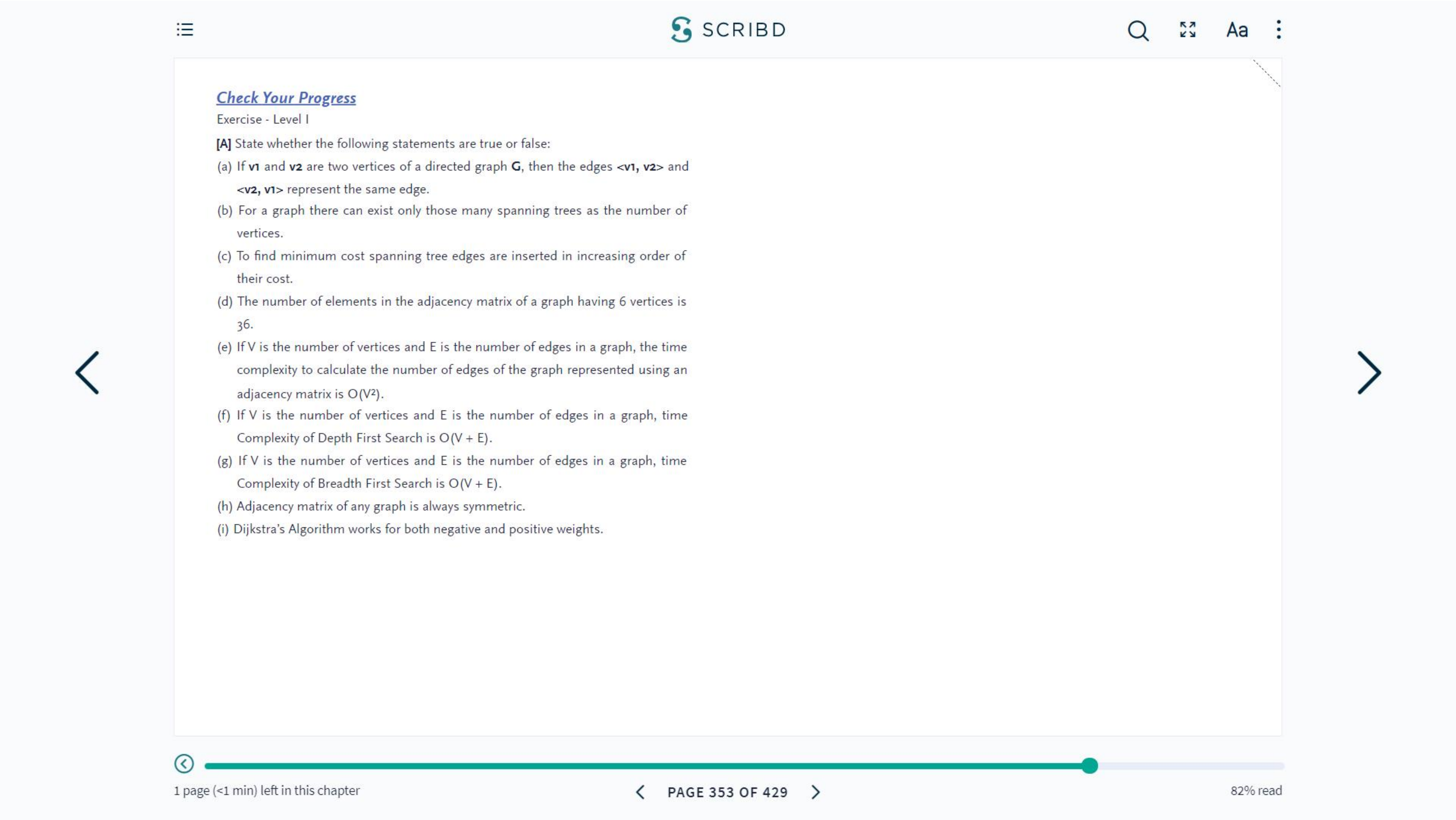
(b) A graph can be represented using an adjacency matrix, adjacency lists or adjacency multi-lists.

(c) There are two algorithms for graph traversal—depth first search and breadth first search.

(d) A spanning tree is an undirected tree consisting of only those edges that are necessary to connect all vertices in the original graph.

(e) Minimum cost spanning tree can be obtained using Kruskal's algorithm or Prim's algorithm.

(f) The shortest path between vertices in a weighted directed graph can be obtained using Dijkstra's algorithm.



Check Your Progress

Exercise - Level I

[A] State whether the following statements are true or false:

(a) If **v1** and **v2** are two vertices of a directed graph **G**, then the edges **<v1, v2>** and **<v2, v1>** represent the same edge.

(b) For a graph there can exist only those many spanning trees as the number of vertices.

(c) To find minimum cost spanning tree edges are inserted in increasing order of their cost.

(d) The number of elements in the adjacency matrix of a graph having 6 vertices is 36.

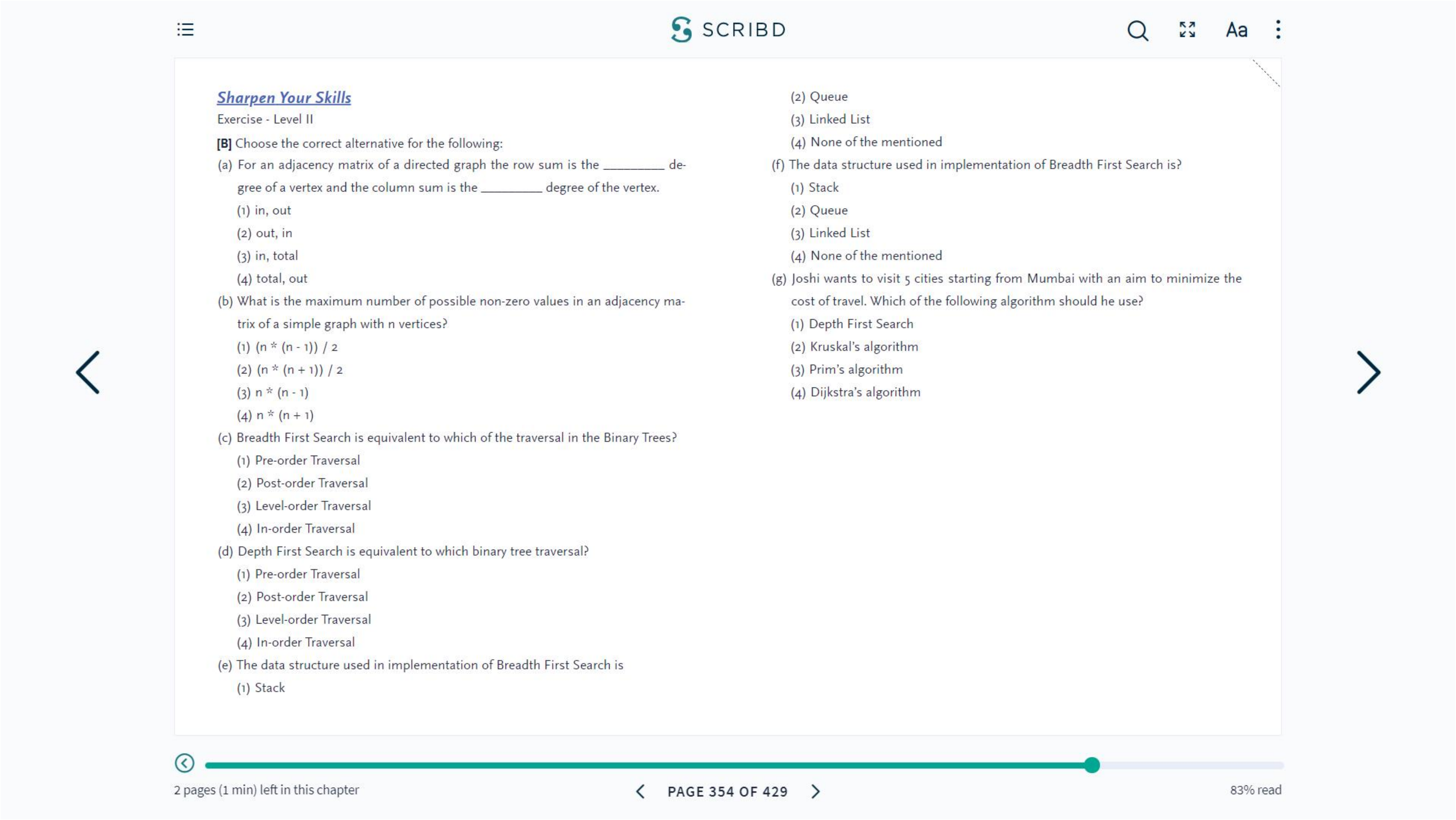
(e) If V is the number of vertices and E is the number of edges in a graph, the time complexity to calculate the number of edges of the graph represented using an adjacency matrix is $O(V^2)$.

(f) If V is the number of vertices and E is the number of edges in a graph, time Complexity of Depth First Search is O(V + E).

(g) If V is the number of vertices and E is the number of edges in a graph, time Complexity of Breadth First Search is O(V + E).

(h) Adjacency matrix of any graph is always symmetric.

(i) Dijkstra's Algorithm works for both negative and positive weights.



Sharpen Your Skills

Exercise - Level II

[B] Choose the correct alternative for the following:

(a) For an adjacency matrix of a directed graph the row sum is the _________ degree of a vertex and the column sum is the _________ degree of the vertex.

(1) in, out

(2) out, in

(3) in, total

(4) total, out

(b) What is the maximum number of possible non-zero values in an adjacency matrix of a simple graph with n vertices?

(1) (n * (n - 1)) / 2

(2) (n * (n + 1)) / 2

(3) n * (n - 1)

(4) n * (n + 1)

(c) Breadth First Search is equivalent to which of the traversal in the Binary Trees?

(1) Pre-order Traversal

(2) Post-order Traversal

(3) Level-order Traversal

(4) In-order Traversal

(d) Depth First Search is equivalent to which binary tree traversal?

(1) Pre-order Traversal

(2) Post-order Traversal

(3) Level-order Traversal

(4) In-order Traversal

(e) The data structure used in implementation of Breadth First Search is

(1) Stack

(2) Queue

(3) Linked List

(4) None of the mentioned

(f) The data structure used in implementation of Breadth First Search is?

(1) Stack

(2) Queue

(3) Linked List

(4) None of the mentioned

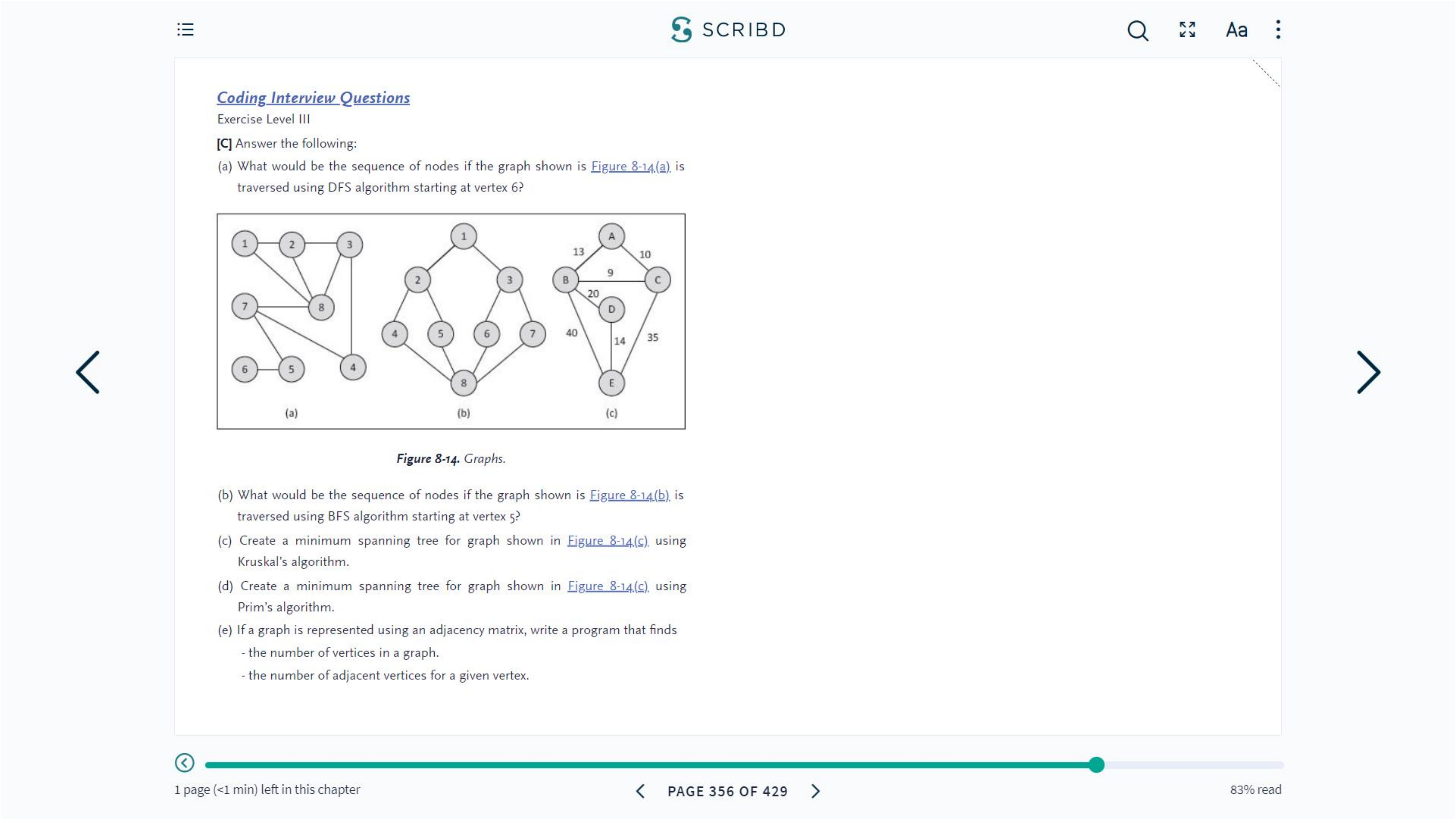
(g) Joshi wants to visit 5 cities starting from Mumbai with an aim to minimize the cost of travel. Which of the following algorithm should he use?

(1) Depth First Search

(2) Kruskal's algorithm

(3) Prim's algorithm

(4) Dijkstra's algorithm



Coding Interview Questions

Exercise Level III

[C] Answer the following:

(a) What would be the sequence of nodes if the graph shown is Figure 8-14(a) is traversed using DFS algorithm starting at vertex 6?

Figure 8-14. *Graphs.*

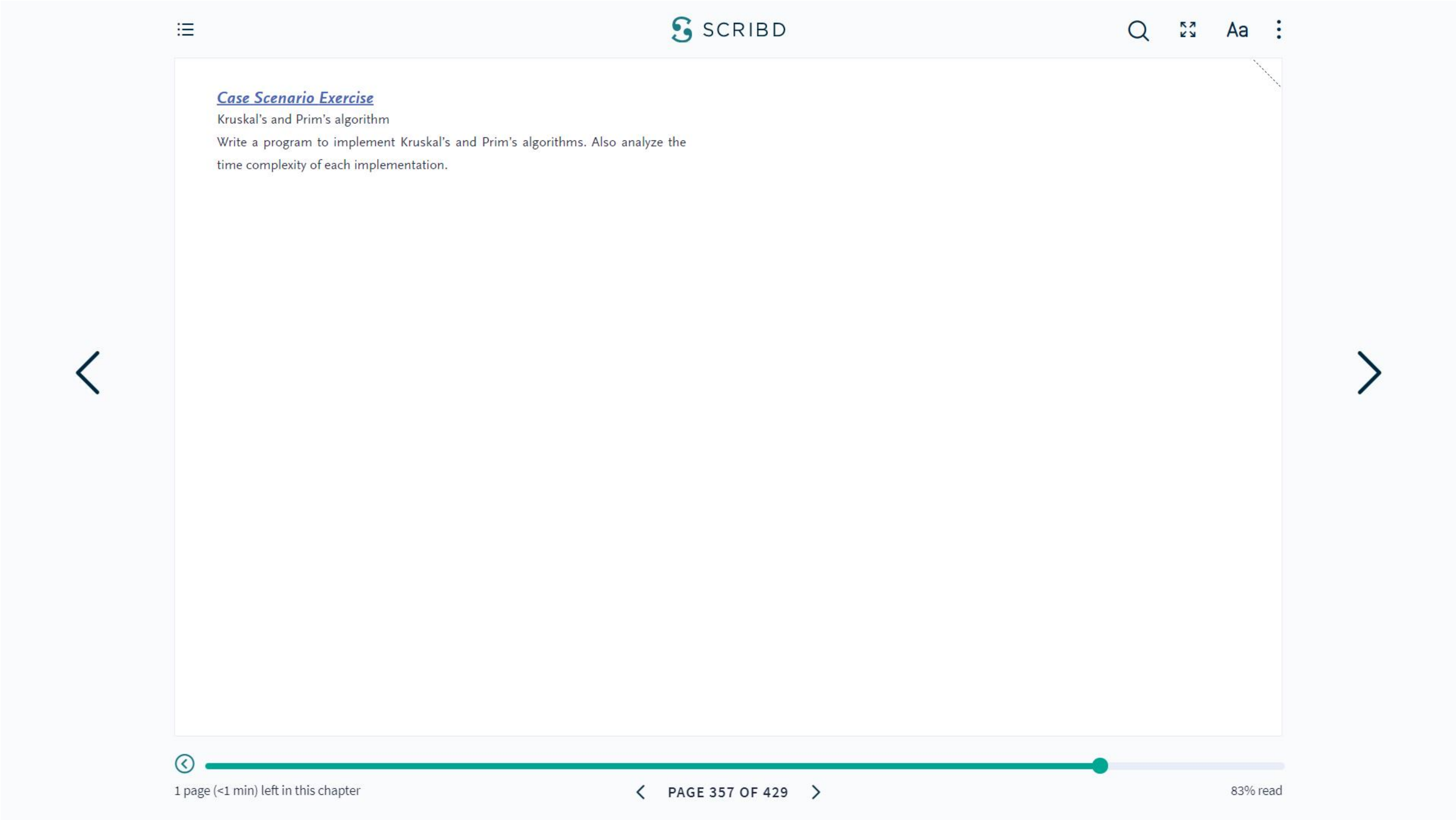
(b) What would be the sequence of nodes if the graph shown is Figure 8-14(b) is traversed using BFS algorithm starting at vertex 5?

(c) Create a minimum spanning tree for graph shown in Figure 8-14(c) using Kruskal's algorithm.

(d) Create a minimum spanning tree for graph shown in Figure 8-14(c) using Prim's algorithm.

(e) If a graph is represented using an adjacency matrix, write a program that finds

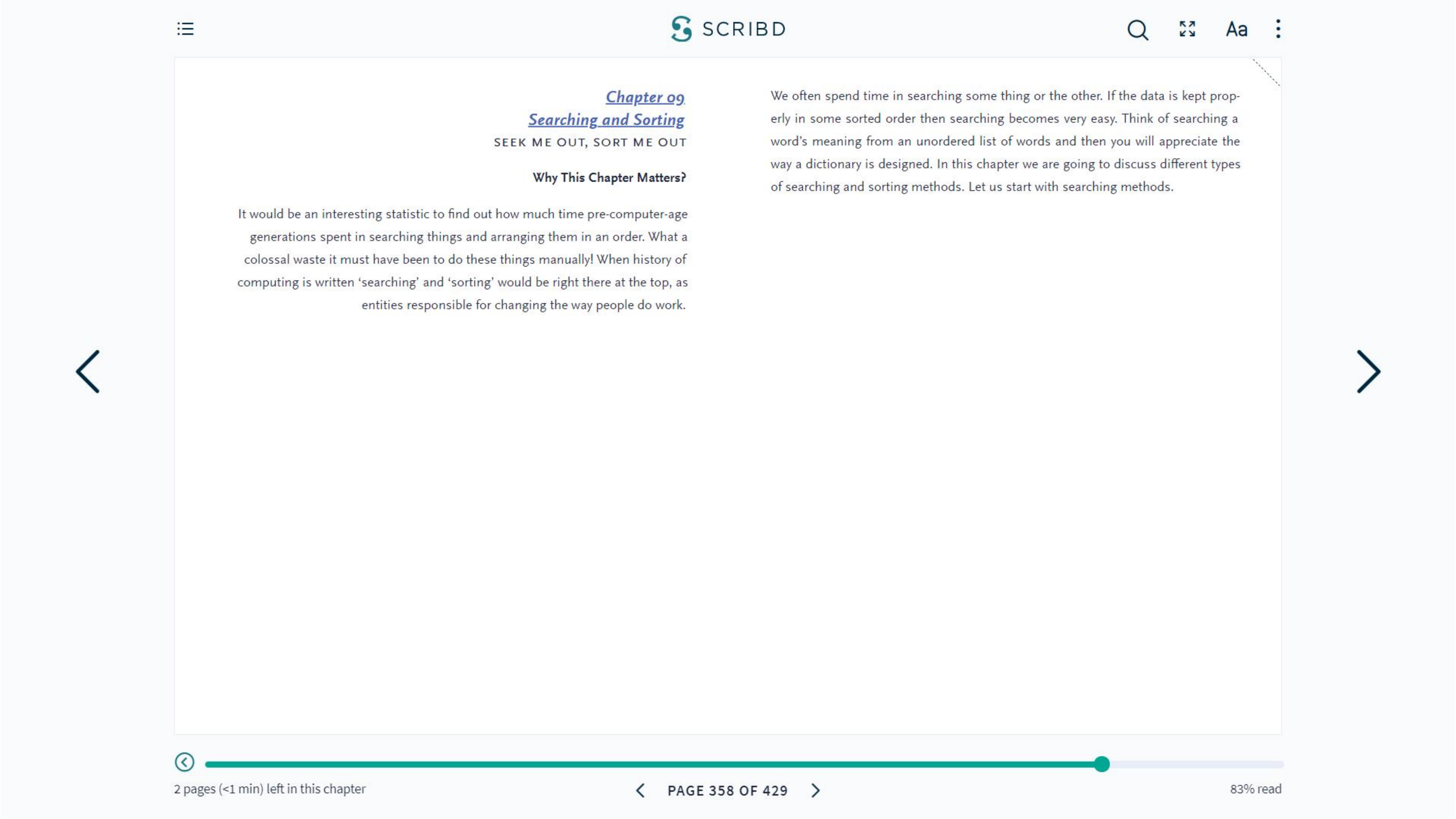
- the number of vertices in a graph.
- the number of adjacent vertices for a given vertex.



Case Scenario Exercise

Kruskal's and Prim's algorithm

Write a program to implement Kruskal's and Prim's algorithms. Also analyze the time complexity of each implementation.



Chapter 09 Searching and Sorting

SEEK ME OUT, SORT ME OUT

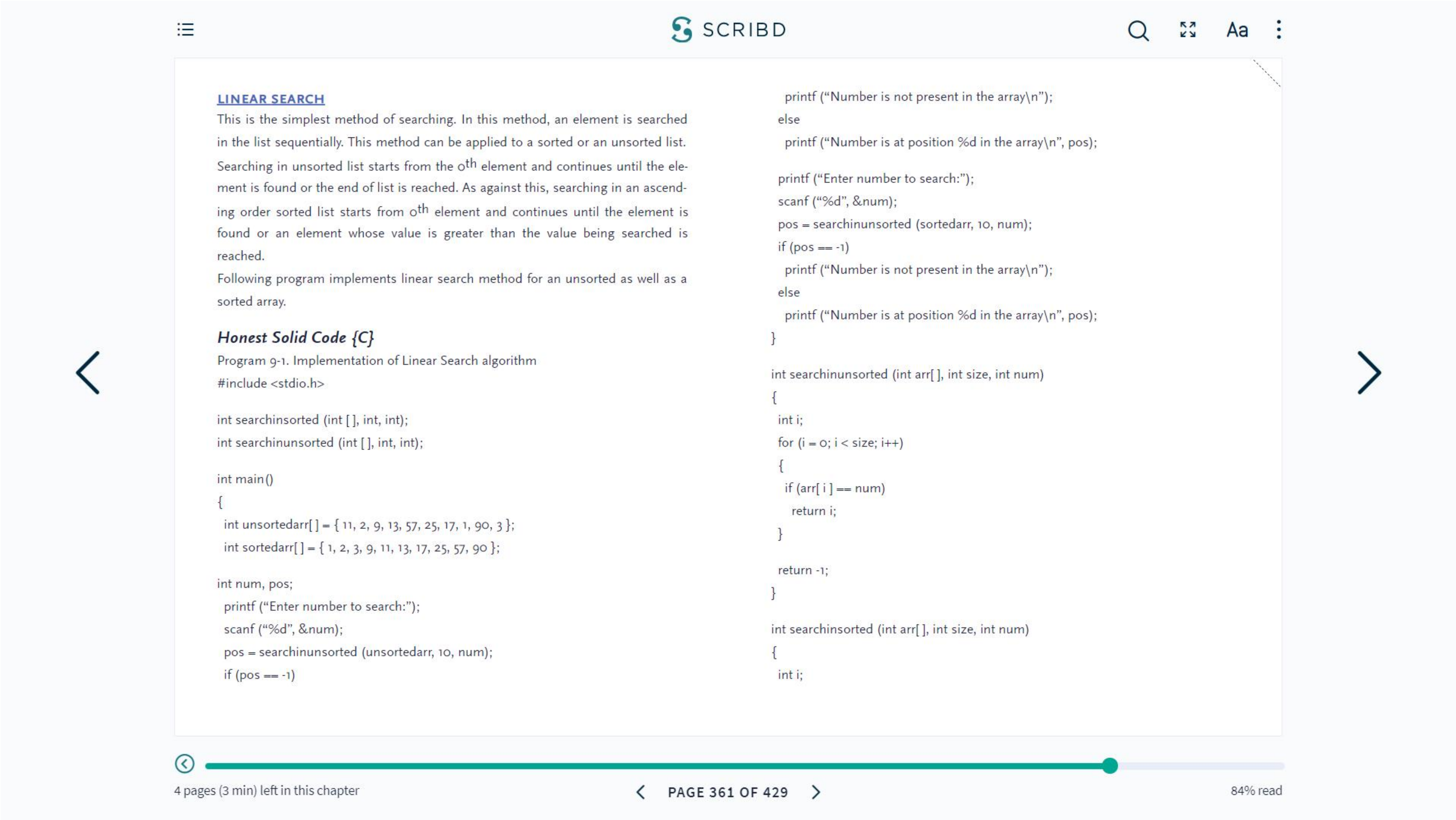
Why This Chapter Matters?

It would be an interesting statistic to find out how much time pre-computer-age generations spent in searching things and arranging them in an order. What a colossal waste it must have been to do these things manually! When history of computing is written 'searching' and 'sorting' would be right there at the top, as entities responsible for changing the way people do work.

We often spend time in searching some thing or the other. If the data is kept properly in some sorted order then searching becomes very easy. Think of searching a word's meaning from an unordered list of words and then you will appreciate the way a dictionary is designed. In this chapter we are going to discuss different types of searching and sorting methods. Let us start with searching methods.

Searching

Searching is an operation that finds the location of a given element in a list. The search is said to be successful or unsuccessful depending on whether the element that is to be searched is found or not. Here, we will discuss two standard searching methods—Linear search and Binary search.



LINEAR SEARCH

This is the simplest method of searching. In this method, an element is searched in the list sequentially. This method can be applied to a sorted or an unsorted list. Searching in unsorted list starts from the 0th element and continues until the element is found or the end of list is reached. As against this, searching in an ascending order sorted list starts from 0th element and continues until the element is found or an element whose value is greater than the value being searched is reached.

Following program implements linear search method for an unsorted as well as a sorted array.

Honest Solid Code {C}

Program 9-1. Implementation of Linear Search algorithm

```
#include <stdio.h>

int searchinsorted (int [ ], int, int);
int searchinunsorted (int [ ], int, int);

int main()
{
 int unsortedarr[ ] = { 11, 2, 9, 13, 57, 25, 17, 1, 90, 3 };
 int sortedarr[ ] = { 1, 2, 3, 9, 11, 13, 17, 25, 57, 90 };

int num, pos;
 printf ("Enter number to search:");
 scanf ("%d", &num);
 pos = searchinunsorted (unsortedarr, 10, num);
 if (pos == -1)
  printf ("Number is not present in the array\n");
 else
  printf ("Number is at position %d in the array\n", pos);

 printf ("Enter number to search:");
 scanf ("%d", &num);
 pos = searchinunsorted (sortedarr, 10, num);
 if (pos == -1)
  printf ("Number is not present in the array\n");
 else
  printf ("Number is at position %d in the array\n", pos);
}

int searchinunsorted (int arr[ ], int size, int num)
{
 int i;
 for (i = 0; i < size; i++)
 {
  if (arr[ i ] == num)
   return i;
 }

 return -1;
}

int searchinsorted (int arr[ ], int size, int num)
{
 int i;
```

```
 if (num > arr[ size - 1 ])
  return -1;

 for (i = 0; i < size; i++)
 {

if (arr[ i ] > num)
   return -1;
  if (arr[ i ] == num)
   return i;
 }

 return -1;
}
```

Output:

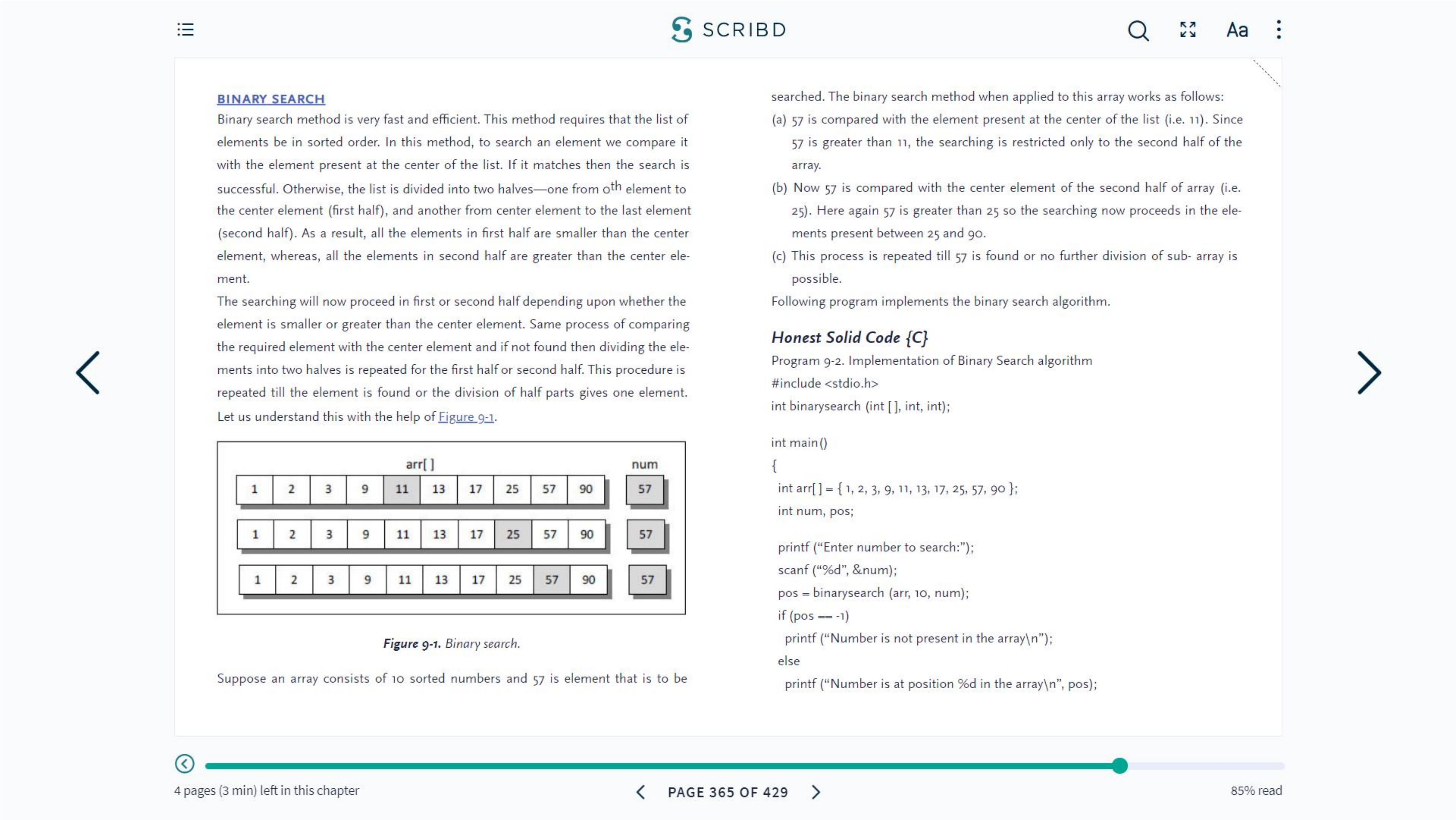
```
Enter number to search: 13
Number is at position 3 in the array
Enter number to search: 100
Number is not present in the array
```

In the program, **num** is the number that is to be searched in the array. While searching in **unsortedarr**, inside the **for** loop each time **arr[i]** is compared with **num**. If any element is equal to **num**, it means that the element is found. Hence its position in the array is returned. If control reaches beyond the **for** loop, it means that the element is not present in the array. In this case -1 is returned. We have returned -1, because no element can be present at position -1 in the array.

While searching in a sorted array, search starts at the 0th element and ends when the element is found or any element of the list is found to be greater than the element to be searched.

The number of comparisons in case of sorted list might be less as compared to the unsorted list because the search may not always continue till the end of the list.

The performance of linear search algorithm can be measured by counting the number of comparisons done to locate an element. In the worst case, in an array of size **n**, this algorithm would carry out **n** comparisons to reach a conclusion whether the element being searched is present in the array or not. Hence worst case time complexity of this algorithm is **O (n)**.



BINARY SEARCH

Binary search method is very fast and efficient. This method requires that the list of elements be in sorted order. In this method, to search an element we compare it with the element present at the center of the list. If it matches then the search is successful. Otherwise, the list is divided into two halves—one from 0th element to the center element (first half), and another from center element to the last element (second half). As a result, all the elements in first half are smaller than the center element, whereas, all the elements in second half are greater than the center element.

The searching will now proceed in first or second half depending upon whether the element is smaller or greater than the center element. Same process of comparing the required element with the center element and if not found then dividing the elements into two halves is repeated for the first half or second half. This procedure is repeated till the element is found or the division of half parts gives one element. Let us understand this with the help of Figure 9-1.

arr[ ]										num
1	2	3	9	11	13	17	25	57	90	57
1	2	3	9	11	13	17	25	57	90	57
1	2	3	9	11	13	17	25	57	90	57

Figure 9-1. *Binary search.*

Suppose an array consists of 10 sorted numbers and 57 is element that is to be searched. The binary search method when applied to this array works as follows:

(a) 57 is compared with the element present at the center of the list (i.e. 11). Since 57 is greater than 11, the searching is restricted only to the second half of the array.

(b) Now 57 is compared with the center element of the second half of array (i.e. 25). Here again 57 is greater than 25 so the searching now proceeds in the elements present between 25 and 90.

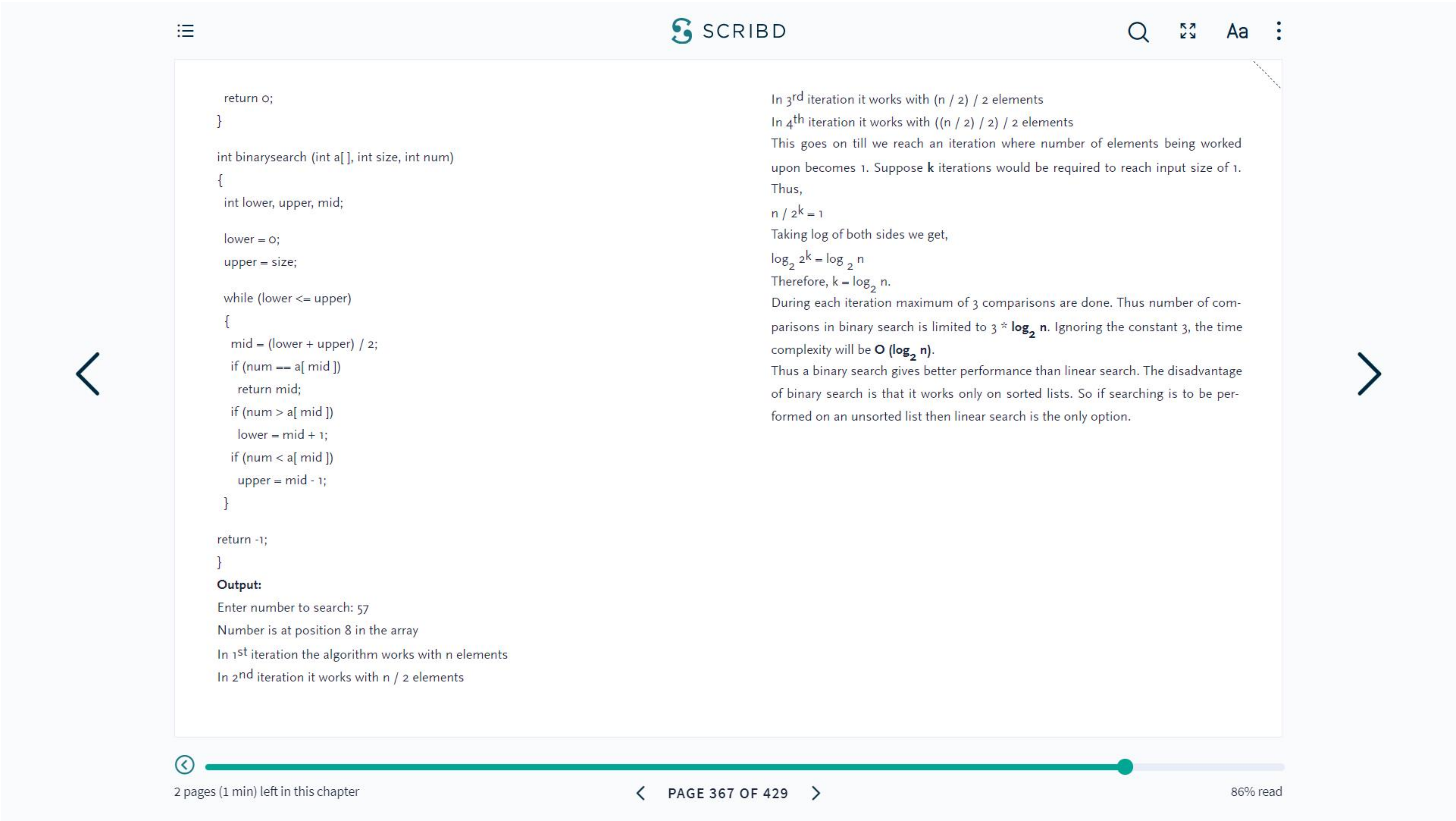
(c) This process is repeated till 57 is found or no further division of sub- array is possible.

Following program implements the binary search algorithm.

Honest Solid Code {C}

Program 9-2. Implementation of Binary Search algorithm

```
#include <stdio.h>
int binarysearch (int [ ], int, int);

int main()
{
 int arr[ ] = { 1, 2, 3, 9, 11, 13, 17, 25, 57, 90 };
 int num, pos;

 printf (“Enter number to search:”);
 scanf (“%d”, &num);
 pos = binarysearch (arr, 10, num);
 if (pos == -1)
  printf (“Number is not present in the array\n”);
 else
  printf (“Number is at position %d in the array\n”, pos);
```



```
 return 0;
}

int binarysearch (int a[ ], int size, int num)
{
 int lower, upper, mid;

 lower = 0;
 upper = size;

 while (lower <= upper)
 {
  mid = (lower + upper) / 2;
  if (num == a[ mid ])
   return mid;
  if (num > a[ mid ])
   lower = mid + 1;
  if (num < a[ mid ])
   upper = mid - 1;
 }

return -1;
}
```

Output:

```
Enter number to search: 57
Number is at position 8 in the array
```

In 1st iteration the algorithm works with n elements

In 2nd iteration it works with n / 2 elements

In 3rd iteration it works with (n / 2) / 2 elements

In 4th iteration it works with ((n / 2) / 2) / 2 elements

This goes on till we reach an iteration where number of elements being worked upon becomes 1. Suppose **k** iterations would be required to reach input size of 1. Thus,

$n / 2^k = 1$

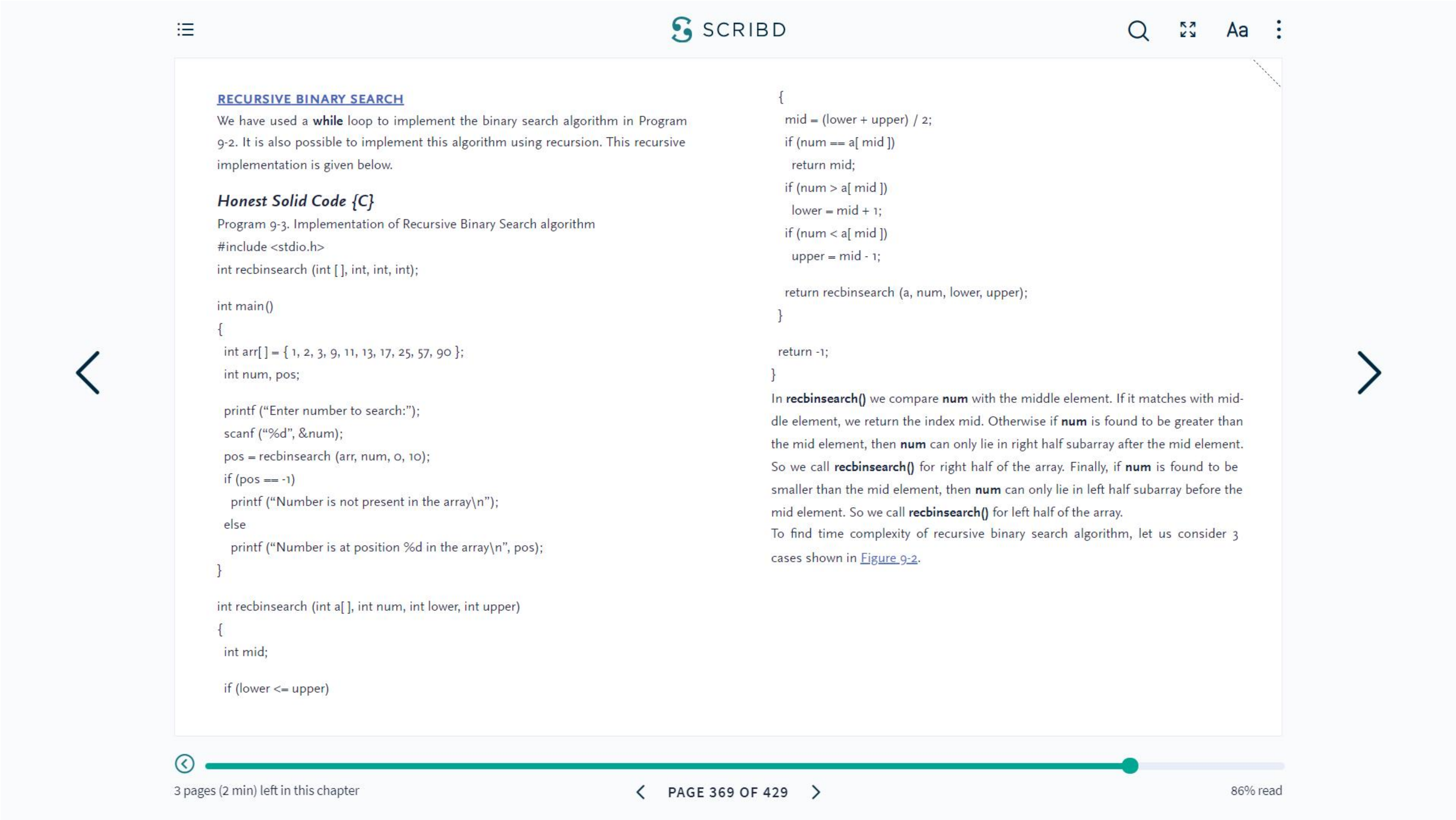
Taking log of both sides we get,

$\log_2 2^k = \log_2 n$

Therefore, $k = \log_2 n$.

During each iteration maximum of 3 comparisons are done. Thus number of comparisons in binary search is limited to 3 * $\mathbf{\log_2 n}$. Ignoring the constant 3, the time complexity will be **O** $(\mathbf{\log_2 n})$.

Thus a binary search gives better performance than linear search. The disadvantage of binary search is that it works only on sorted lists. So if searching is to be performed on an unsorted list then linear search is the only option.



RECURSIVE BINARY SEARCH

We have used a **while** loop to implement the binary search algorithm in Program 9-2. It is also possible to implement this algorithm using recursion. This recursive implementation is given below.

Honest Solid Code {C}

Program 9-3. Implementation of Recursive Binary Search algorithm

```
#include <stdio.h>
int recbinsearch (int [ ], int, int, int);

int main()
{
 int arr[ ] = { 1, 2, 3, 9, 11, 13, 17, 25, 57, 90 };
 int num, pos;

 printf (“Enter number to search:”);
 scanf (“%d”, &num);
 pos = recbinsearch (arr, num, 0, 10);
 if (pos == -1)
  printf (“Number is not present in the array\n”);
 else
  printf (“Number is at position %d in the array\n”, pos);
}

int recbinsearch (int a[ ], int num, int lower, int upper)
{
 int mid;

 if (lower <= upper)
 {
  mid = (lower + upper) / 2;
  if (num == a[ mid ])
   return mid;
  if (num > a[ mid ])
   lower = mid + 1;
  if (num < a[ mid ])
   upper = mid - 1;

  return recbinsearch (a, num, lower, upper);
 }

 return -1;
}
```

In **recbinsearch()** we compare **num** with the middle element. If it matches with middle element, we return the index mid. Otherwise if **num** is found to be greater than the mid element, then **num** can only lie in right half subarray after the mid element. So we call **recbinsearch()** for right half of the array. Finally, if **num** is found to be smaller than the mid element, then **num** can only lie in left half subarray before the mid element. So we call **recbinsearch()** for left half of the array.

To find time complexity of recursive binary search algorithm, let us consider 3 cases shown in Figure 9-2.

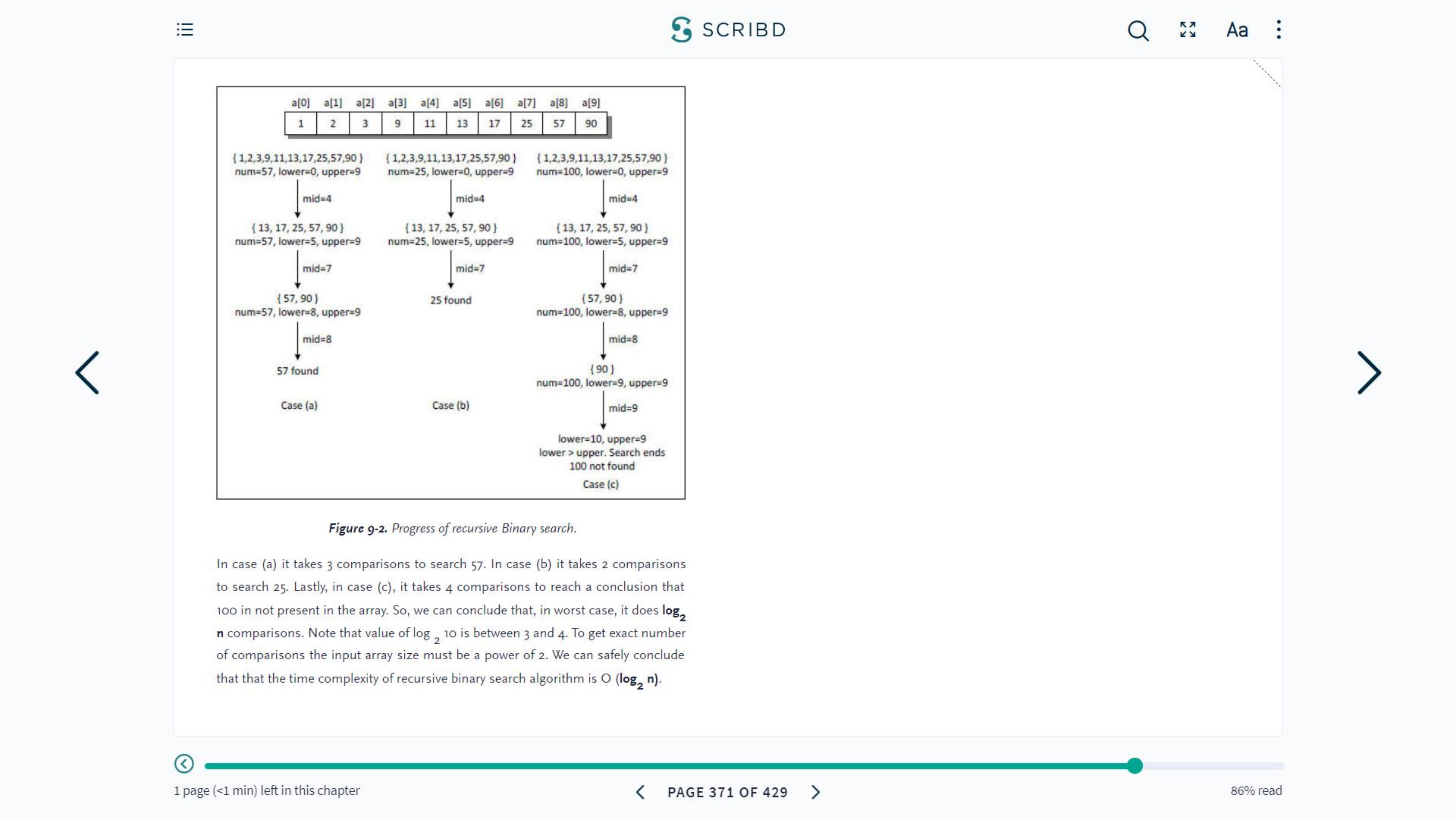


Figure 9-2. *Progress of recursive Binary search.*

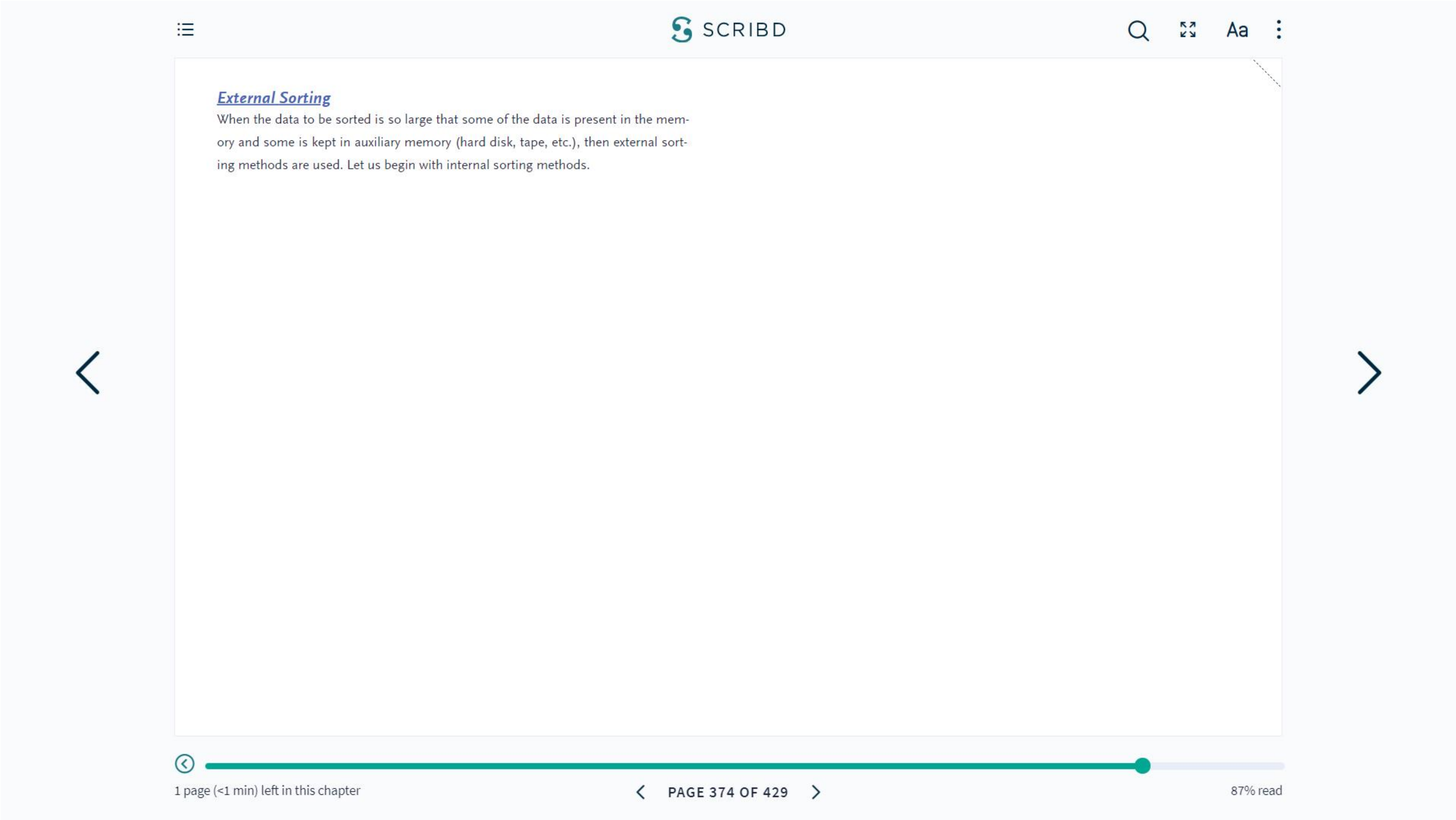
In case (a) it takes 3 comparisons to search 57. In case (b) it takes 2 comparisons to search 25. Lastly, in case (c), it takes 4 comparisons to reach a conclusion that 100 in not present in the array. So, we can conclude that, in worst case, it does **$\log_2$ n** comparisons. Note that value of log $_2$ 10 is between 3 and 4. To get exact number of comparisons the input array size must be a power of 2. We can safely conclude that that the time complexity of recursive binary search algorithm is O (**$\log_2$ n**).

Sorting

Sorting refers to arranging elements of a set in some order. There are different methods that are used to sort the data in ascending or descending order. These methods can be divided into two categories. They are as follows:

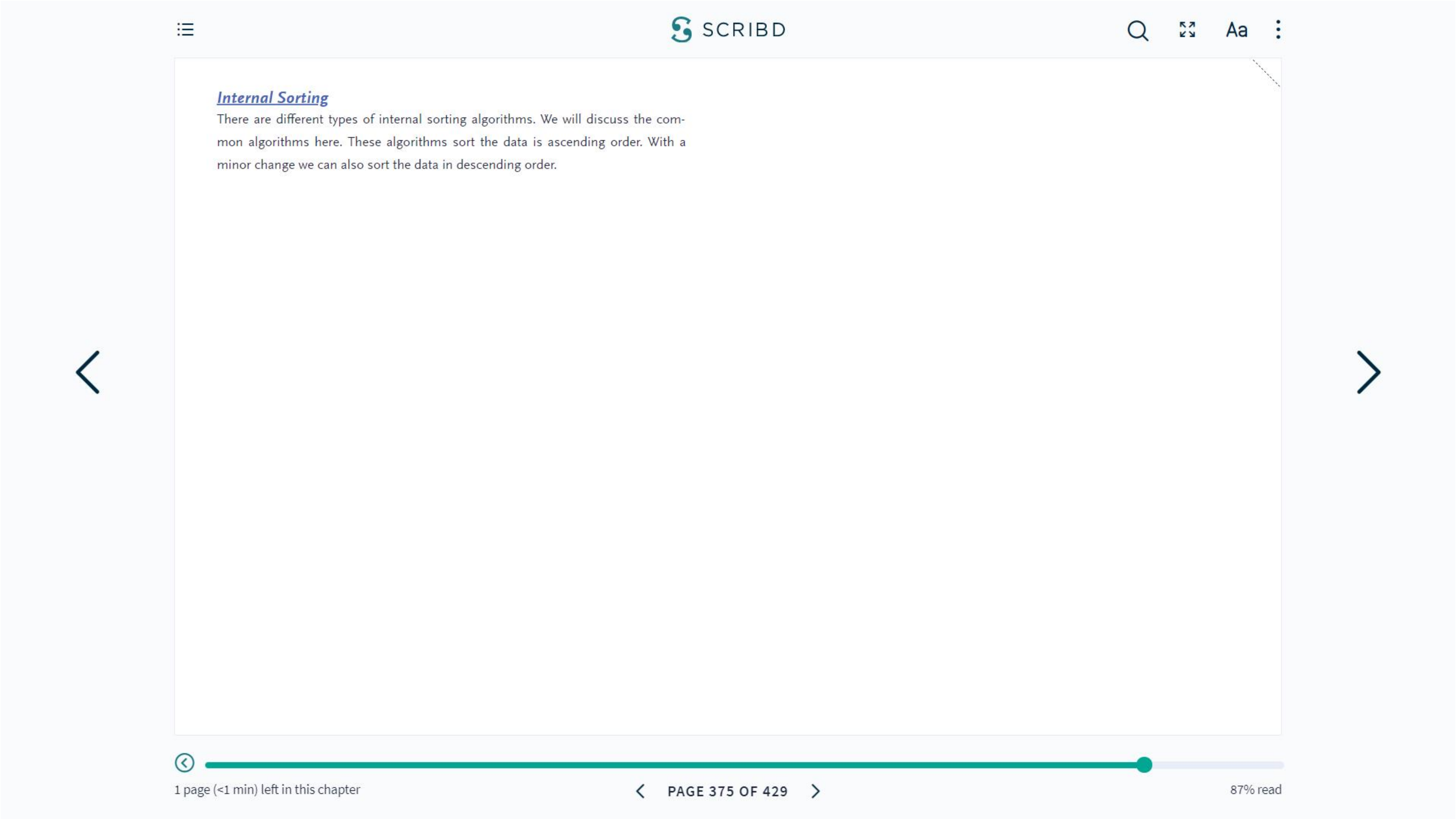
Internal Sorting

If all the data to be sorted can be accommodated at a time in memory then internal sorting methods are used.



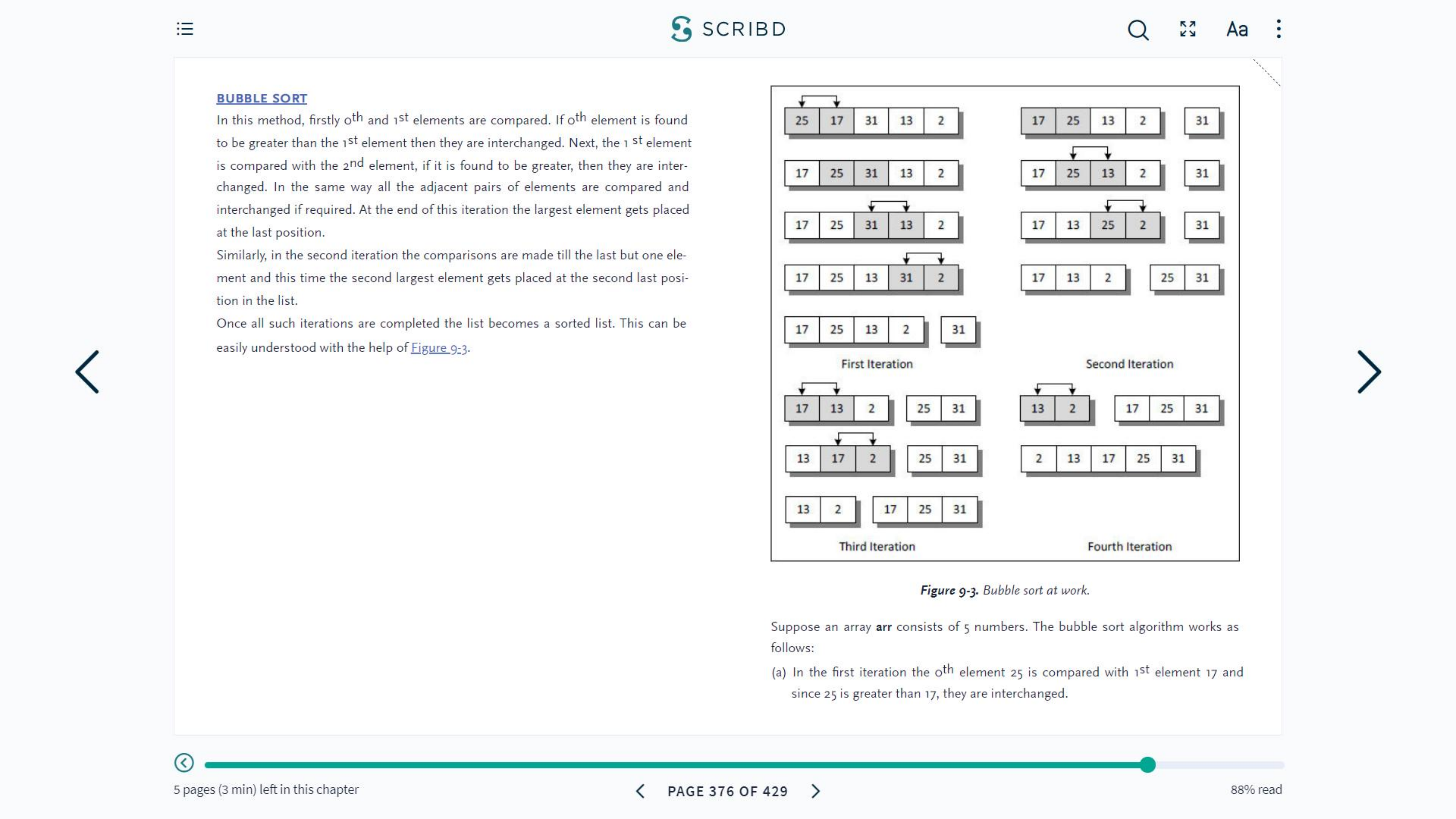
External Sorting

When the data to be sorted is so large that some of the data is present in the memory and some is kept in auxiliary memory (hard disk, tape, etc.), then external sorting methods are used. Let us begin with internal sorting methods.



Internal Sorting

There are different types of internal sorting algorithms. We will discuss the common algorithms here. These algorithms sort the data is ascending order. With a minor change we can also sort the data in descending order.



BUBBLE SORT

In this method, firstly 0th and 1st elements are compared. If 0th element is found to be greater than the 1st element then they are interchanged. Next, the 1 st element is compared with the 2nd element, if it is found to be greater, then they are interchanged. In the same way all the adjacent pairs of elements are compared and interchanged if required. At the end of this iteration the largest element gets placed at the last position.

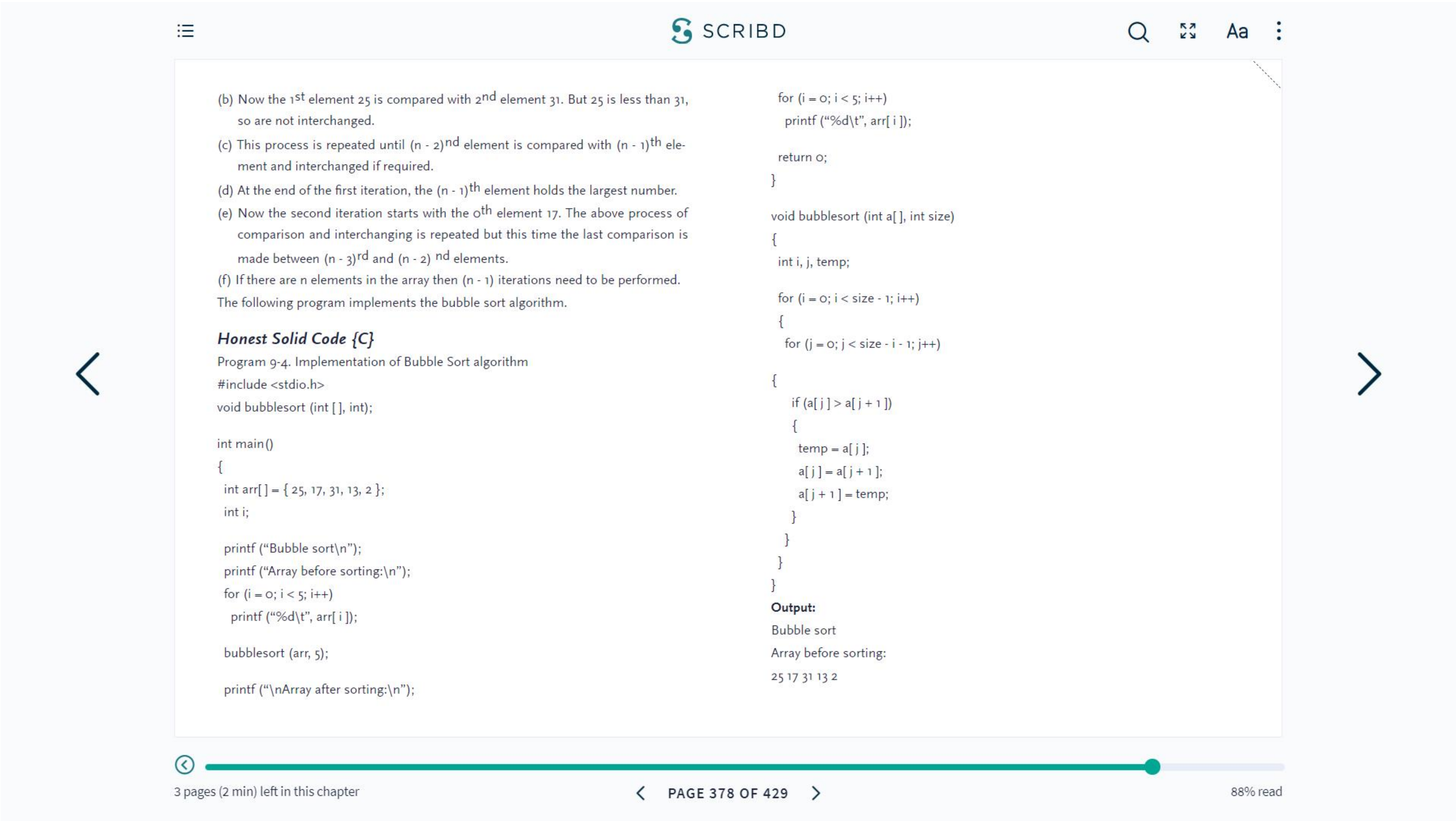
Similarly, in the second iteration the comparisons are made till the last but one element and this time the second largest element gets placed at the second last position in the list.

Once all such iterations are completed the list becomes a sorted list. This can be easily understood with the help of Figure 9-3.

Figure 9-3. *Bubble sort at work.*

Suppose an array **arr** consists of 5 numbers. The bubble sort algorithm works as follows:

(a) In the first iteration the 0th element 25 is compared with 1st element 17 and since 25 is greater than 17, they are interchanged.



(b) Now the 1st element 25 is compared with 2nd element 31. But 25 is less than 31, so are not interchanged.

(c) This process is repeated until (n - 2)nd element is compared with (n - 1)th element and interchanged if required.

(d) At the end of the first iteration, the (n - 1)th element holds the largest number.

(e) Now the second iteration starts with the 0th element 17. The above process of comparison and interchanging is repeated but this time the last comparison is made between (n - 3)rd and (n - 2) nd elements.

(f) If there are n elements in the array then (n - 1) iterations need to be performed.

The following program implements the bubble sort algorithm.

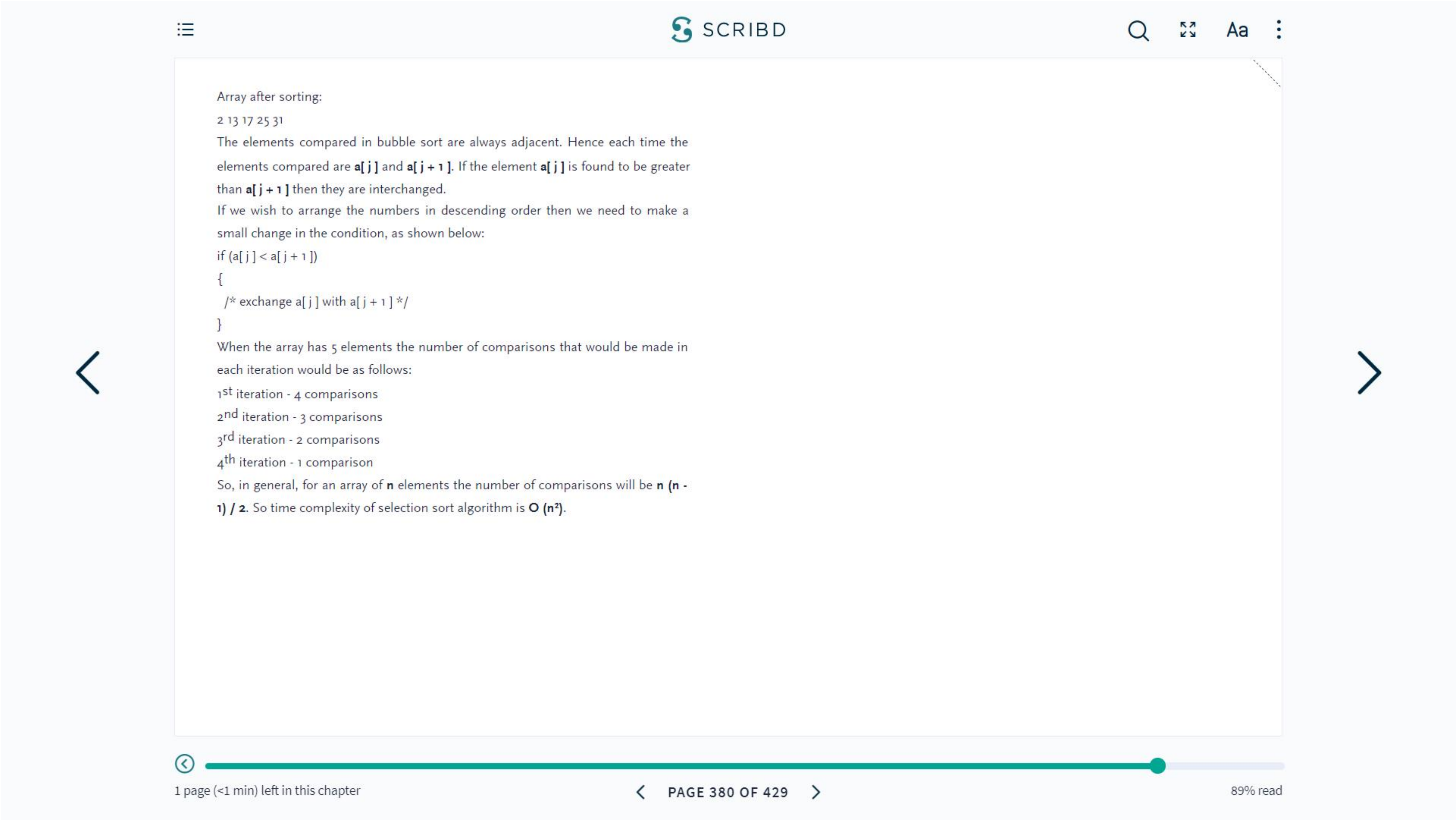
Honest Solid Code {C}

Program 9-4. Implementation of Bubble Sort algorithm

```
#include <stdio.h>
void bubblesort (int [ ], int);

int main()
{
 int arr[ ] = { 25, 17, 31, 13, 2 };
 int i;

 printf ("Bubble sort\n");
 printf ("Array before sorting:\n");
 for (i = 0; i < 5; i++)
  printf ("%d\t", arr[ i ]);

 bubblesort (arr, 5);

 printf ("\nArray after sorting:\n");
 for (i = 0; i < 5; i++)
  printf ("%d\t", arr[ i ]);

 return 0;
}

void bubblesort (int a[ ], int size)
{
 int i, j, temp;

 for (i = 0; i < size - 1; i++)
 {
  for (j = 0; j < size - i - 1; j++)

{
    if (a[ j ] > a[ j + 1 ])
    {
     temp = a[ j ];
     a[ j ] = a[ j + 1 ];
     a[ j + 1 ] = temp;
    }
   }
 }
}
```

Output:

```
Bubble sort
Array before sorting:
25 17 31 13 2
```



Array after sorting:

2 13 17 25 31

The elements compared in bubble sort are always adjacent. Hence each time the elements compared are **a[j]** and **a[j + 1]**. If the element **a[j]** is found to be greater than **a[j + 1]** then they are interchanged.

If we wish to arrange the numbers in descending order then we need to make a small change in the condition, as shown below:

```
if (a[ j ] < a[ j + 1 ])
{
 /* exchange a[ j ] with a[ j + 1 ] */
}
```

When the array has 5 elements the number of comparisons that would be made in each iteration would be as follows:

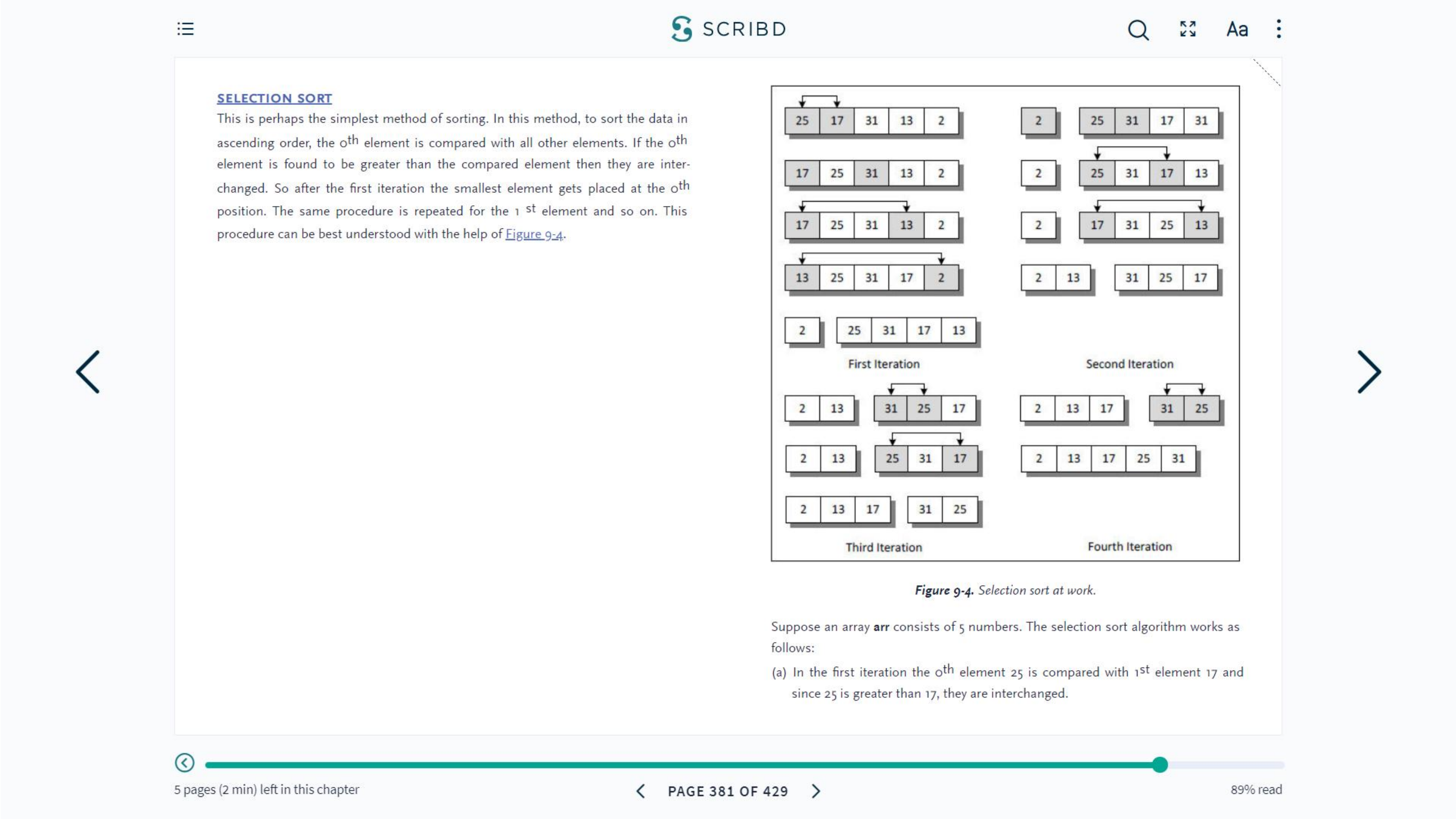
1st iteration - 4 comparisons

2nd iteration - 3 comparisons

3rd iteration - 2 comparisons

4th iteration - 1 comparison

So, in general, for an array of **n** elements the number of comparisons will be **n (n - 1) / 2**. So time complexity of selection sort algorithm is **O (n^2)**.



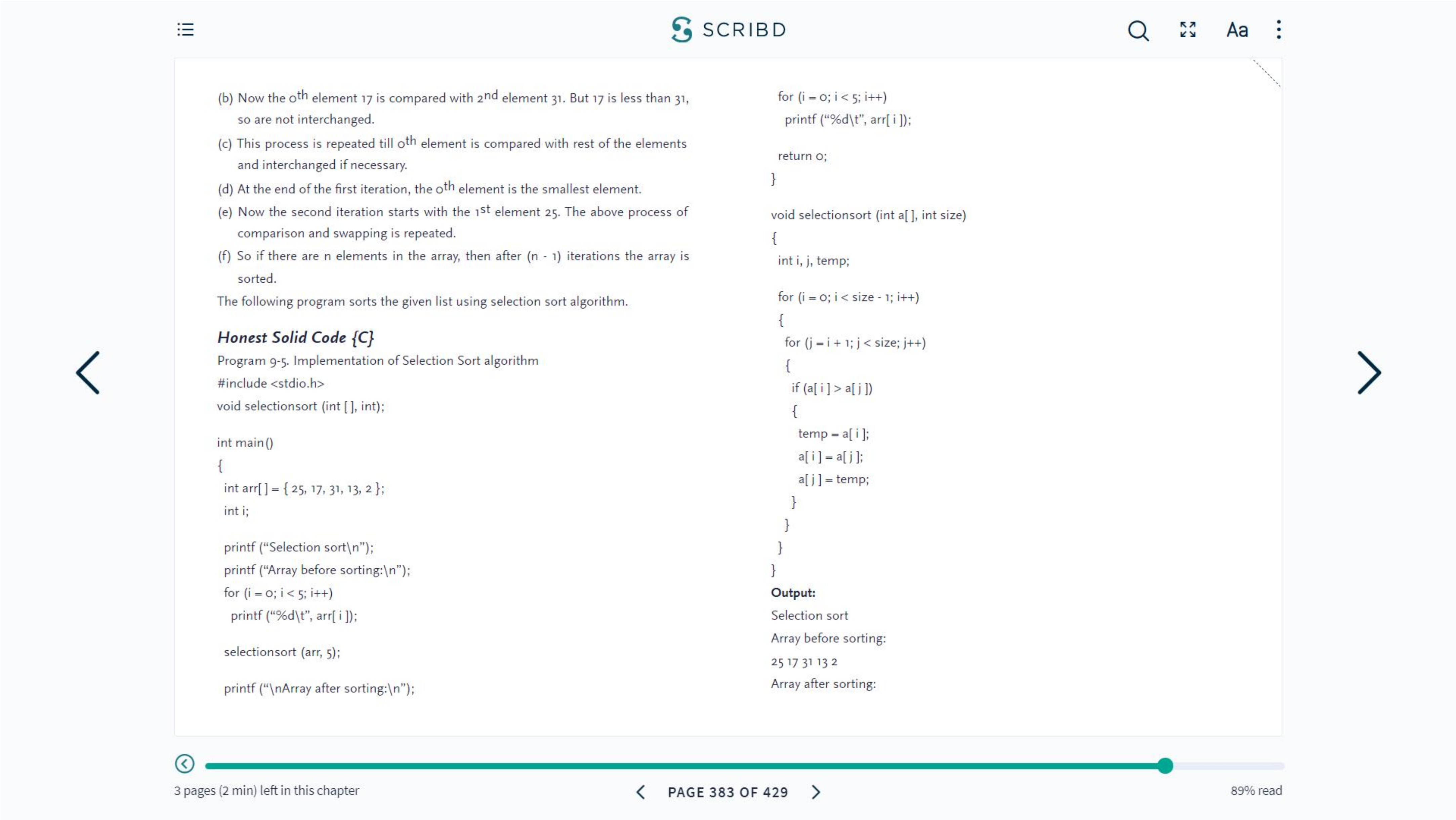
SELECTION SORT

This is perhaps the simplest method of sorting. In this method, to sort the data in ascending order, the 0th element is compared with all other elements. If the 0th element is found to be greater than the compared element then they are interchanged. So after the first iteration the smallest element gets placed at the 0th position. The same procedure is repeated for the 1 st element and so on. This procedure can be best understood with the help of Figure 9-4.

Figure 9-4. *Selection sort at work.*

Suppose an array **arr** consists of 5 numbers. The selection sort algorithm works as follows:

(a) In the first iteration the 0th element 25 is compared with 1st element 17 and since 25 is greater than 17, they are interchanged.



(b) Now the 0th element 17 is compared with 2nd element 31. But 17 is less than 31, so are not interchanged.

(c) This process is repeated till 0th element is compared with rest of the elements and interchanged if necessary.

(d) At the end of the first iteration, the 0th element is the smallest element.

(e) Now the second iteration starts with the 1st element 25. The above process of comparison and swapping is repeated.

(f) So if there are n elements in the array, then after (n - 1) iterations the array is sorted.

The following program sorts the given list using selection sort algorithm.

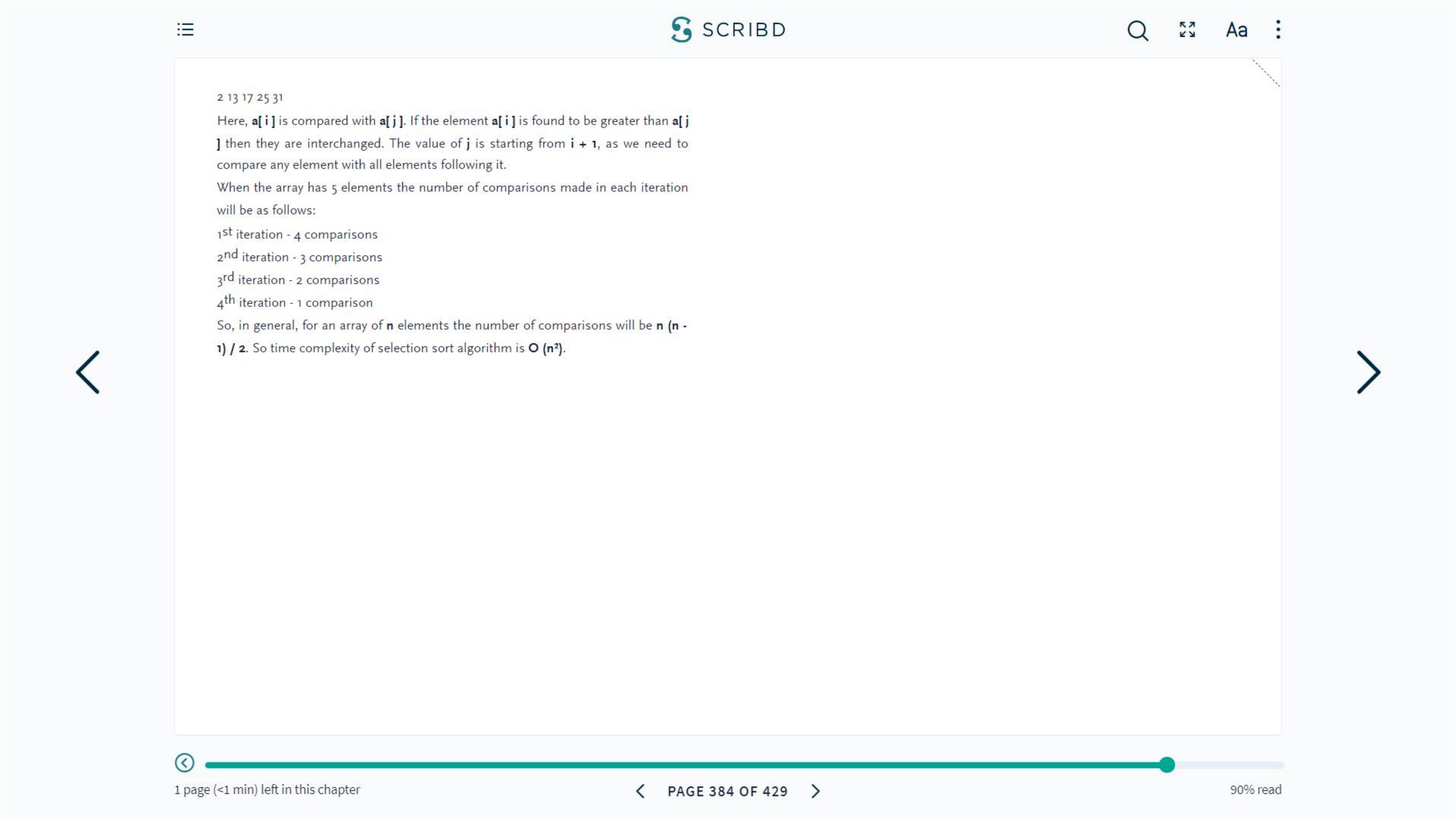
Honest Solid Code {C}

Program 9-5. Implementation of Selection Sort algorithm

```
#include <stdio.h>
void selectionsort (int [ ], int);

int main()
{
 int arr[ ] = { 25, 17, 31, 13, 2 };
 int i;

 printf ("Selection sort\n");
 printf ("Array before sorting:\n");
 for (i = 0; i < 5; i++)
  printf ("%d\t", arr[ i ]);

 selectionsort (arr, 5);

 printf ("\nArray after sorting:\n");
 for (i = 0; i < 5; i++)
  printf ("%d\t", arr[ i ]);

 return 0;
}

void selectionsort (int a[ ], int size)
{
 int i, j, temp;

 for (i = 0; i < size - 1; i++)
 {
  for (j = i + 1; j < size; j++)
  {
   if (a[ i ] > a[ j ])
   {
    temp = a[ i ];
    a[ i ] = a[ j ];
    a[ j ] = temp;
   }
  }
 }
}
```

Output:

```
Selection sort
Array before sorting:
25 17 31 13 2
Array after sorting:
```



2 13 17 25 31

Here, **a[i]** is compared with **a[j]**. If the element **a[i]** is found to be greater than **a[j]** then they are interchanged. The value of **j** is starting from **i + 1**, as we need to compare any element with all elements following it.

When the array has 5 elements the number of comparisons made in each iteration will be as follows:

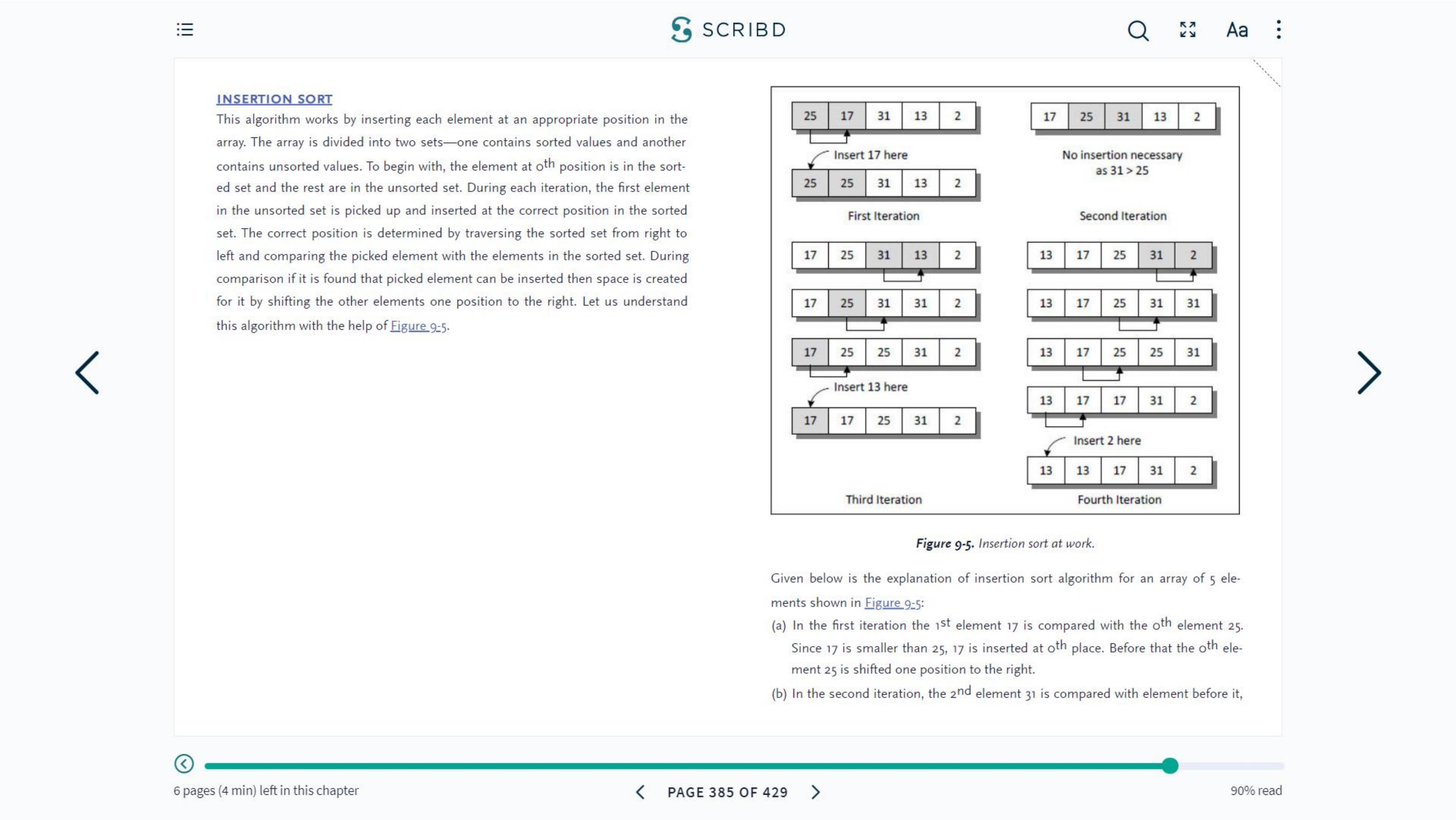
1st iteration - 4 comparisons

2nd iteration - 3 comparisons

3rd iteration - 2 comparisons

4th iteration - 1 comparison

So, in general, for an array of **n** elements the number of comparisons will be **n (n - 1) / 2**. So time complexity of selection sort algorithm is **O (n^2)**.



INSERTION SORT

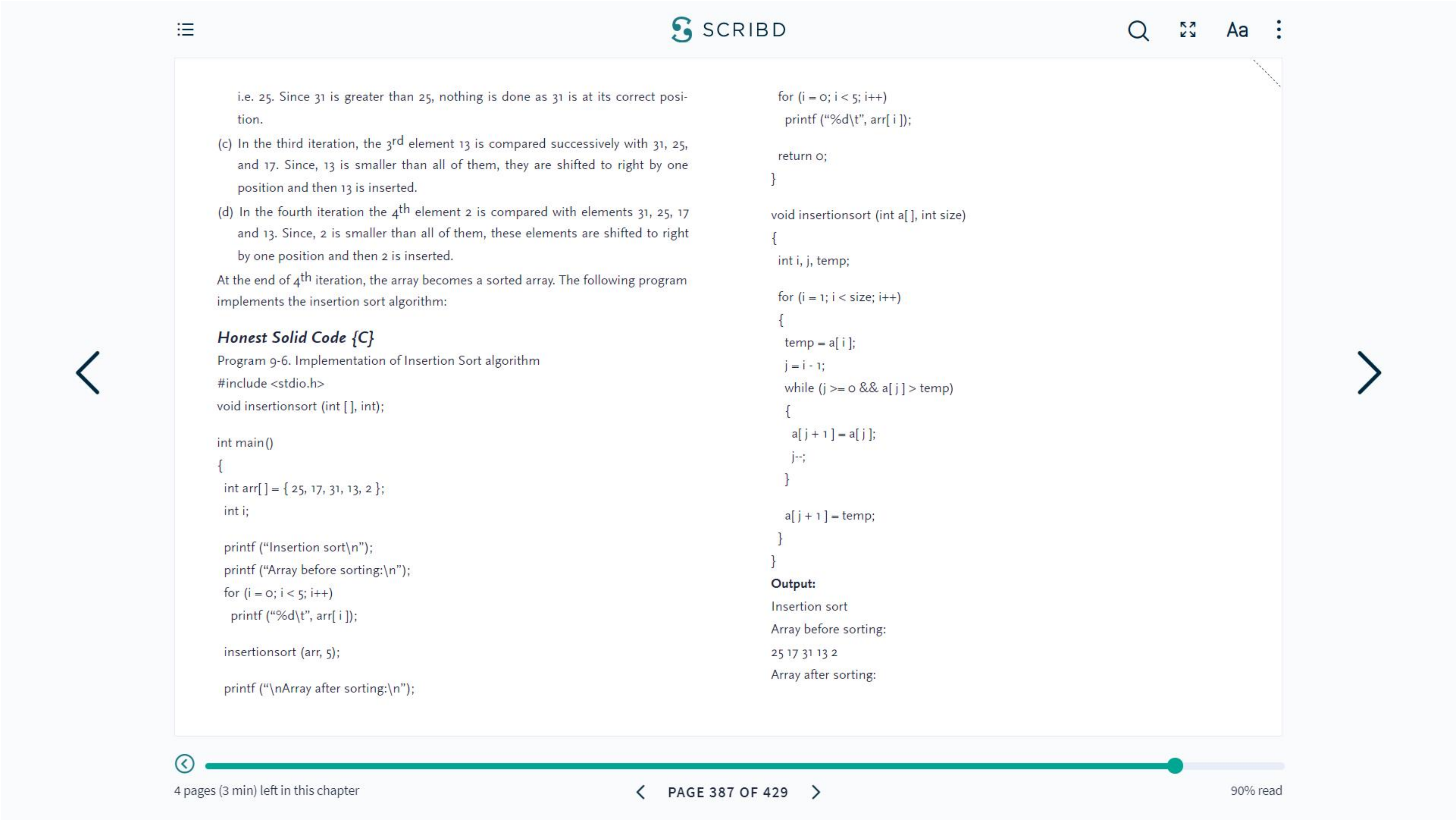
This algorithm works by inserting each element at an appropriate position in the array. The array is divided into two sets—one contains sorted values and another contains unsorted values. To begin with, the element at 0^{th} position is in the sorted set and the rest are in the unsorted set. During each iteration, the first element in the unsorted set is picked up and inserted at the correct position in the sorted set. The correct position is determined by traversing the sorted set from right to left and comparing the picked element with the elements in the sorted set. During comparison if it is found that picked element can be inserted then space is created for it by shifting the other elements one position to the right. Let us understand this algorithm with the help of Figure 9-5.

Figure 9-5. *Insertion sort at work.*

Given below is the explanation of insertion sort algorithm for an array of 5 elements shown in Figure 9-5:

(a) In the first iteration the 1^{st} element 17 is compared with the 0^{th} element 25. Since 17 is smaller than 25, 17 is inserted at 0^{th} place. Before that the 0^{th} element 25 is shifted one position to the right.

(b) In the second iteration, the 2^{nd} element 31 is compared with element before it,



i.e. 25. Since 31 is greater than 25, nothing is done as 31 is at its correct position.

(c) In the third iteration, the 3rd element 13 is compared successively with 31, 25, and 17. Since, 13 is smaller than all of them, they are shifted to right by one position and then 13 is inserted.

(d) In the fourth iteration the 4th element 2 is compared with elements 31, 25, 17 and 13. Since, 2 is smaller than all of them, these elements are shifted to right by one position and then 2 is inserted.

At the end of 4th iteration, the array becomes a sorted array. The following program implements the insertion sort algorithm:

Honest Solid Code {C}

Program 9-6. Implementation of Insertion Sort algorithm

```
#include <stdio.h>
void insertionsort (int [ ], int);

int main()
{
 int arr[ ] = { 25, 17, 31, 13, 2 };
 int i;

 printf (“Insertion sort\n”);
 printf (“Array before sorting:\n”);
 for (i = 0; i < 5; i++)
  printf (“%d\t”, arr[ i ]);

 insertionsort (arr, 5);

 printf (“\nArray after sorting:\n”);
 for (i = 0; i < 5; i++)
  printf (“%d\t”, arr[ i ]);

 return 0;
}

void insertionsort (int a[ ], int size)
{
 int i, j, temp;

 for (i = 1; i < size; i++)
 {
  temp = a[ i ];
  j = i - 1;
  while (j >= 0 && a[ j ] > temp)
  {
   a[ j + 1 ] = a[ j ];
   j--;
  }

  a[ j + 1 ] = temp;
 }
}
```

Output:

```
Insertion sort
Array before sorting:
25 17 31 13 2
Array after sorting:
```

2 13 17 25 31

In the program the outer **for** loop is starting from 1 as the unsorted set starts at 1st position. The inner loop is used for comparison to decide the position where the picked element (**temp**) and for shifting the elements one position to the right to make room for inserting the picked element.

Let us consider best case and worst case for analyzing the time complexity of this algorithm. The best case is when the array is already sorted and the worst case is when the array elements are in descending order. The important operations to be considered in this algorithm are comparison to determine where the element should be inserted and movement to create space for inserting the element.

In the best case the number of comparisons and movements will be as shown below.

for i = 1, 1 comparison + 0 movement = 1

for i = 2, 1 comparison + 0 movement = 1

for i = 3, 1 comparison + 0 movement = 1

for i = 4, 1 comparison + 0 movement = 1

...

...

for i = n, 1 comparison + 0 movement = 1

So total number of operations will be 1 + 1 + 1 + 1.... This sum will be equal to **n**. Thus time complexity in best case will be **O (n)**.

In the worst case the number of comparisons and movements will be as shown below.

for i = 2, 1 comparison + 1 movement = 2

for i = 3, 2 comparisons + 2 movements = 4

for i = 3, 3 comparisons + 3 movements = 6

for i = 4, 3 comparisons + 3 movements = 8

for i = n, n - 1 comparisons + n - 1 movements= 2(n - 1)

If we add all this, we get

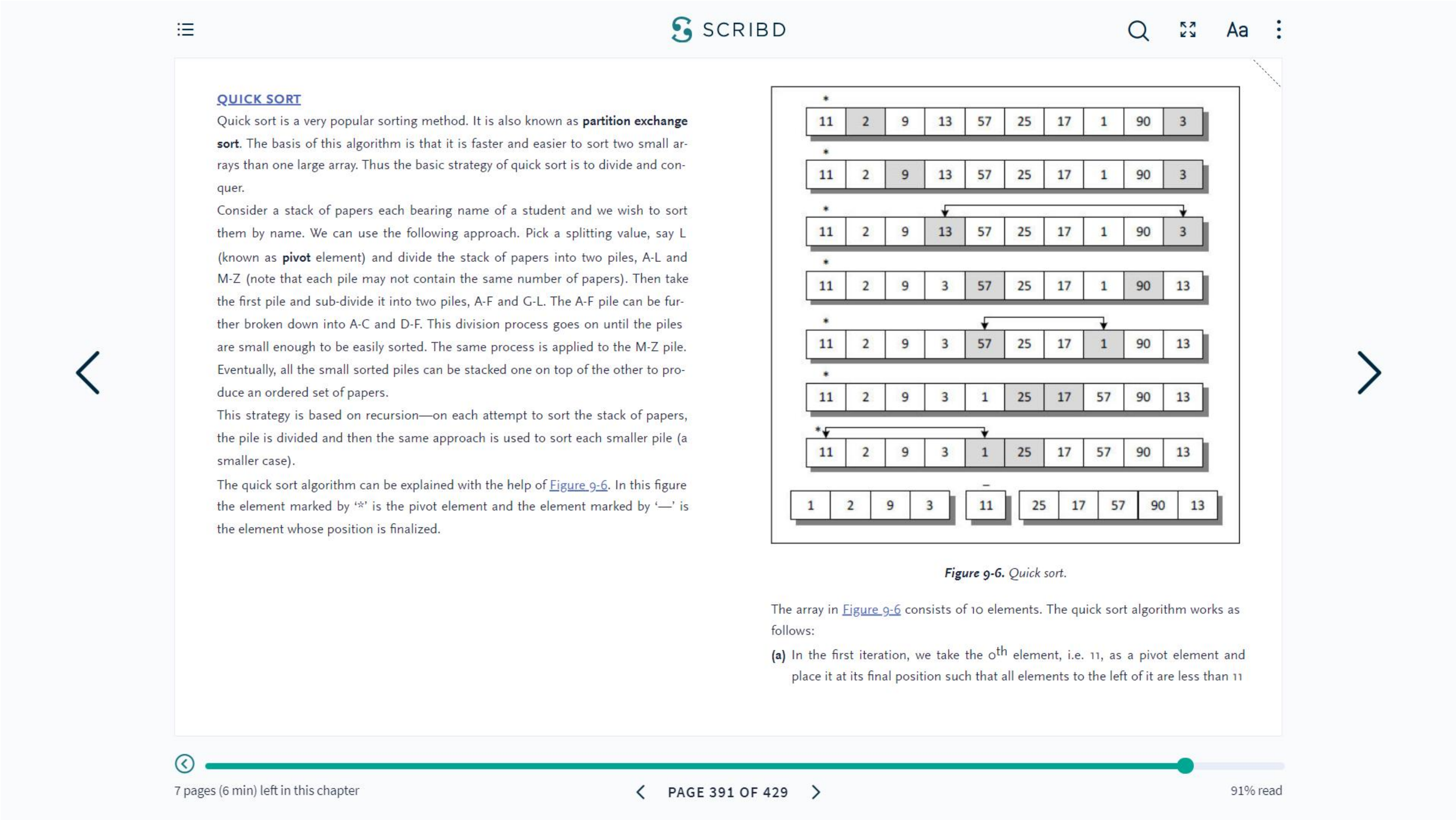
2 + 4 + 6 + 8 + ... + 2 (n - 1)

= 2 (1 + 2 + 3 + + 4... + (n -1))

= 2 (n (n -1) / 2)

= O (n^2)

Thus time complexity in best case will be **O (n^2)**.



QUICK SORT

Quick sort is a very popular sorting method. It is also known as **partition exchange sort**. The basis of this algorithm is that it is faster and easier to sort two small arrays than one large array. Thus the basic strategy of quick sort is to divide and conquer.

Consider a stack of papers each bearing name of a student and we wish to sort them by name. We can use the following approach. Pick a splitting value, say L (known as **pivot** element) and divide the stack of papers into two piles, A-L and M-Z (note that each pile may not contain the same number of papers). Then take the first pile and sub-divide it into two piles, A-F and G-L. The A-F pile can be further broken down into A-C and D-F. This division process goes on until the piles are small enough to be easily sorted. The same process is applied to the M-Z pile. Eventually, all the small sorted piles can be stacked one on top of the other to produce an ordered set of papers.

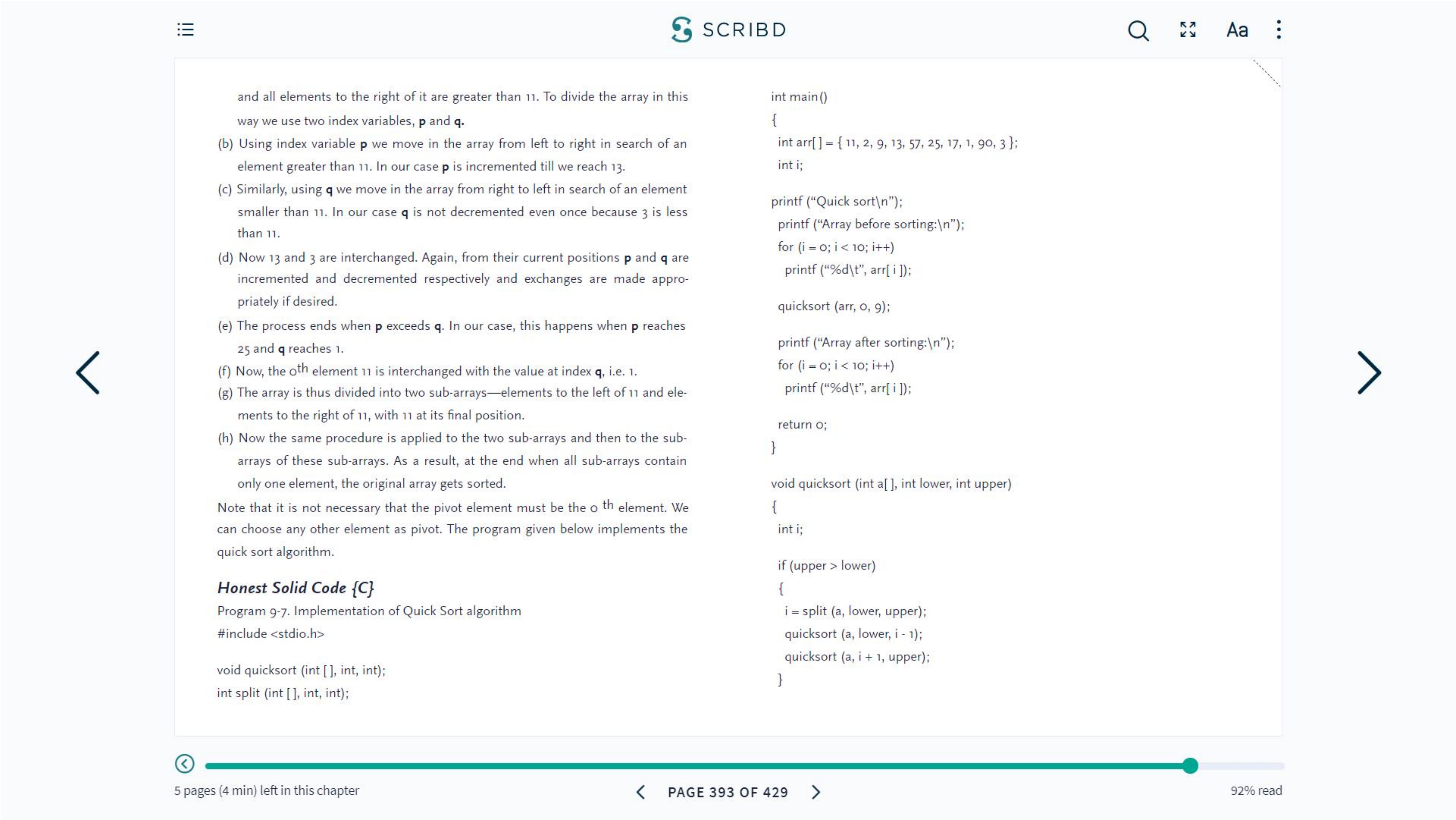
This strategy is based on recursion—on each attempt to sort the stack of papers, the pile is divided and then the same approach is used to sort each smaller pile (a smaller case).

The quick sort algorithm can be explained with the help of Figure 9-6. In this figure the element marked by '*' is the pivot element and the element marked by '—' is the element whose position is finalized.

Figure 9-6. *Quick sort.*

The array in Figure 9-6 consists of 10 elements. The quick sort algorithm works as follows:

(a) In the first iteration, we take the 0th element, i.e. 11, as a pivot element and place it at its final position such that all elements to the left of it are less than 11



and all elements to the right of it are greater than 11. To divide the array in this way we use two index variables, **p** and **q.**

(b) Using index variable **p** we move in the array from left to right in search of an element greater than 11. In our case **p** is incremented till we reach 13.

(c) Similarly, using **q** we move in the array from right to left in search of an element smaller than 11. In our case **q** is not decremented even once because 3 is less than 11.

(d) Now 13 and 3 are interchanged. Again, from their current positions **p** and **q** are incremented and decremented respectively and exchanges are made appropriately if desired.

(e) The process ends when **p** exceeds **q**. In our case, this happens when **p** reaches 25 and **q** reaches 1.

(f) Now, the 0th element 11 is interchanged with the value at index **q**, i.e. 1.

(g) The array is thus divided into two sub-arrays—elements to the left of 11 and elements to the right of 11, with 11 at its final position.

(h) Now the same procedure is applied to the two sub-arrays and then to the sub-arrays of these sub-arrays. As a result, at the end when all sub-arrays contain only one element, the original array gets sorted.

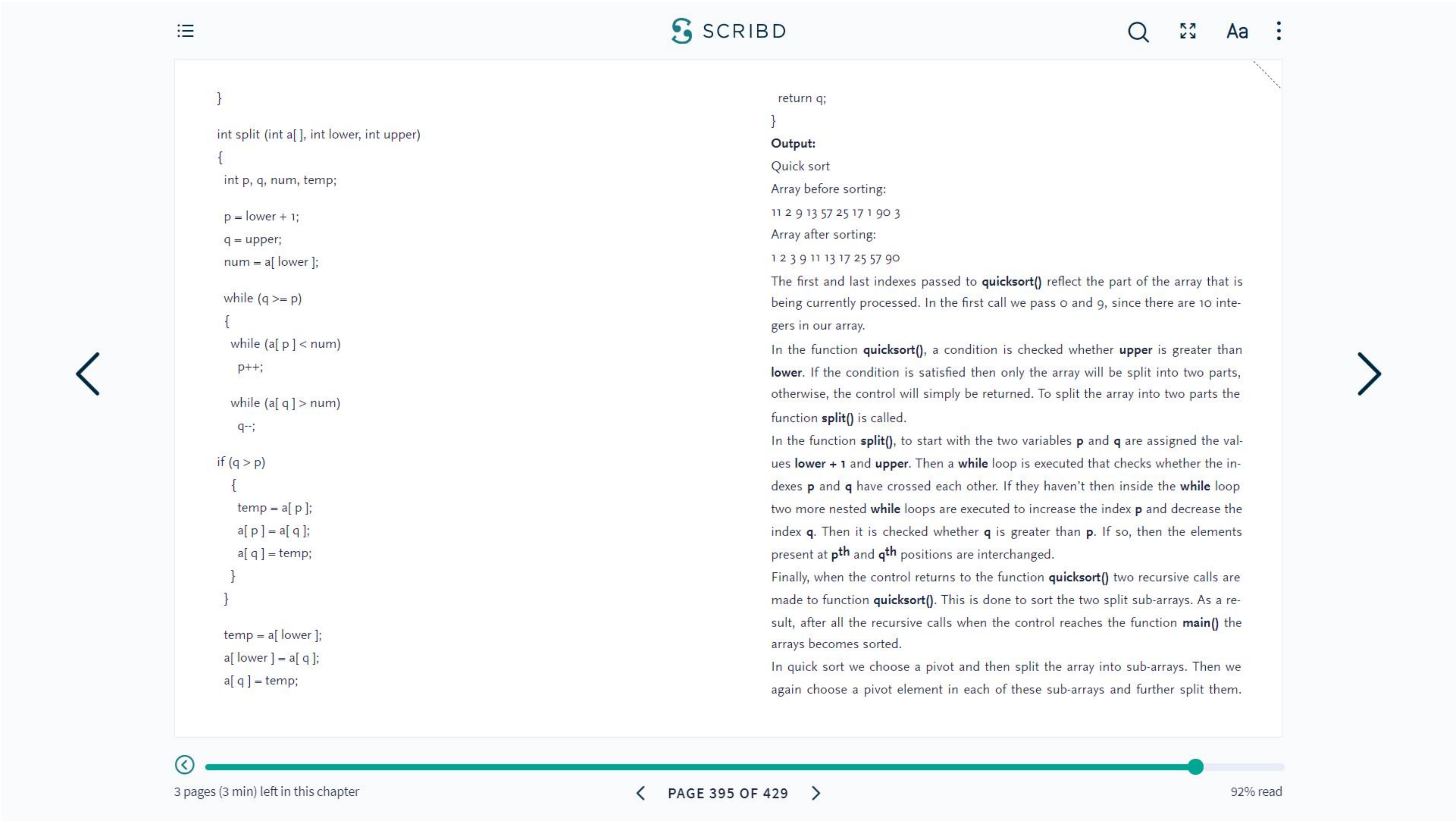
Note that it is not necessary that the pivot element must be the 0 th element. We can choose any other element as pivot. The program given below implements the quick sort algorithm.

Honest Solid Code {C}

Program 9-7. Implementation of Quick Sort algorithm

```
#include <stdio.h>

void quicksort (int [ ], int, int);
int split (int [ ], int, int);
```

```
int main()
{
 int arr[ ] = { 11, 2, 9, 13, 57, 25, 17, 1, 90, 3 };
 int i;

printf (“Quick sort\n”);
 printf (“Array before sorting:\n”);
 for (i = 0; i < 10; i++)
  printf (“%d\t”, arr[ i ]);

 quicksort (arr, 0, 9);

 printf (“Array after sorting:\n”);
 for (i = 0; i < 10; i++)
  printf (“%d\t”, arr[ i ]);

 return 0;
}

void quicksort (int a[ ], int lower, int upper)
{
 int i;

 if (upper > lower)
 {
  i = split (a, lower, upper);
  quicksort (a, lower, i - 1);
  quicksort (a, i + 1, upper);
 }
```



```
}

int split (int a[ ], int lower, int upper)
{
 int p, q, num, temp;

 p = lower + 1;
 q = upper;
 num = a[ lower ];

 while (q >= p)
 {
  while (a[ p ] < num)
   p++;

  while (a[ q ] > num)
   q--;

if (q > p)
  {
   temp = a[ p ];
   a[ p ] = a[ q ];
   a[ q ] = temp;
  }
 }

 temp = a[ lower ];
 a[ lower ] = a[ q ];
 a[ q ] = temp;
 return q;
}
```

Output:

```
Quick sort
Array before sorting:
11 2 9 13 57 25 17 1 90 3
Array after sorting:
1 2 3 9 11 13 17 25 57 90
```

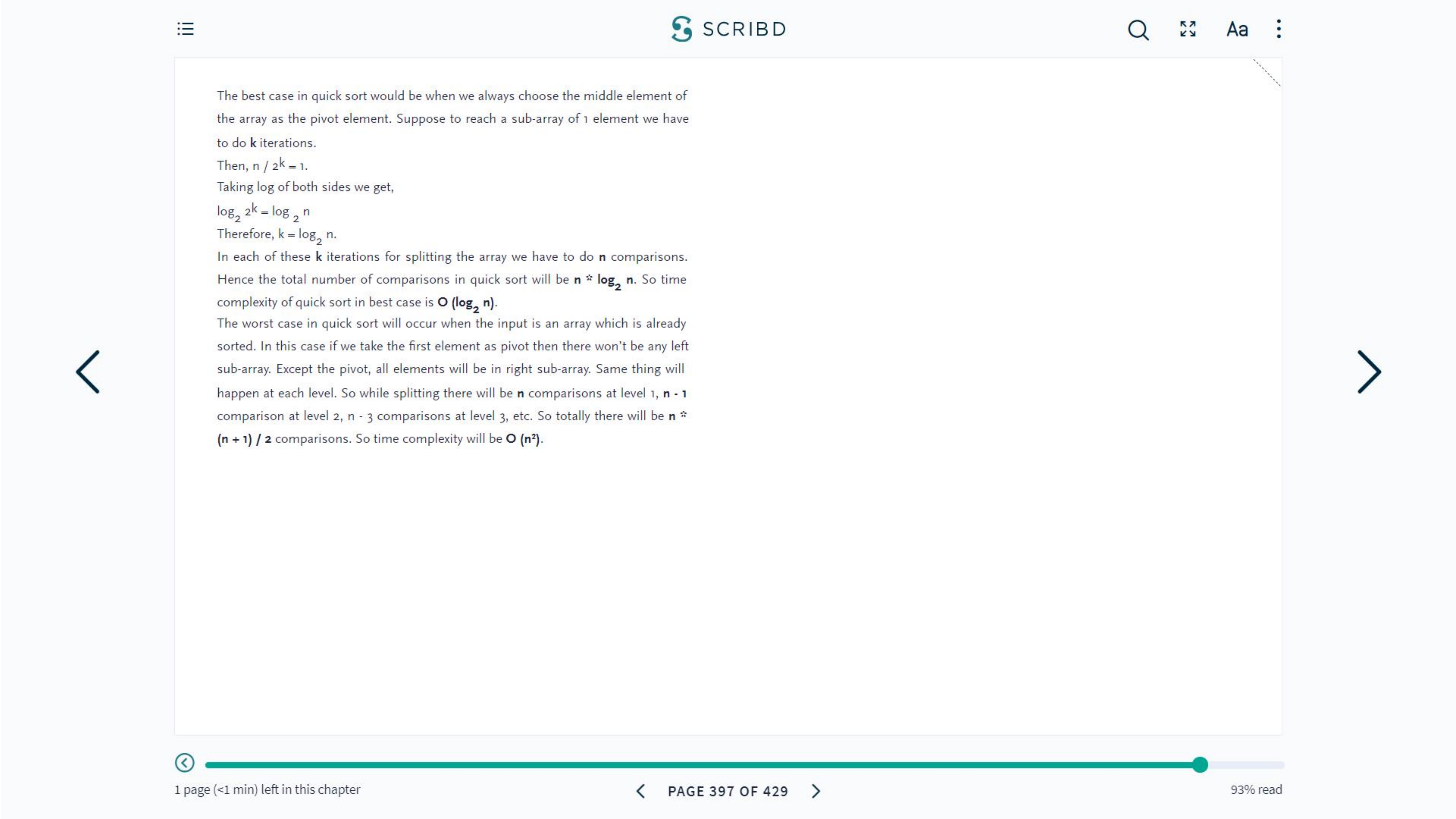
The first and last indexes passed to **quicksort()** reflect the part of the array that is being currently processed. In the first call we pass 0 and 9, since there are 10 integers in our array.

In the function **quicksort()**, a condition is checked whether **upper** is greater than **lower**. If the condition is satisfied then only the array will be split into two parts, otherwise, the control will simply be returned. To split the array into two parts the function **split()** is called.

In the function **split()**, to start with the two variables **p** and **q** are assigned the values **lower + 1** and **upper**. Then a **while** loop is executed that checks whether the indexes **p** and **q** have crossed each other. If they haven't then inside the **while** loop two more nested **while** loops are executed to increase the index **p** and decrease the index **q**. Then it is checked whether **q** is greater than **p**. If so, then the elements present at **p**th and **q**th positions are interchanged.

Finally, when the control returns to the function **quicksort()** two recursive calls are made to function **quicksort()**. This is done to sort the two split sub-arrays. As a result, after all the recursive calls when the control reaches the function **main()** the arrays becomes sorted.

In quick sort we choose a pivot and then split the array into sub-arrays. Then we again choose a pivot element in each of these sub-arrays and further split them.



The best case in quick sort would be when we always choose the middle element of the array as the pivot element. Suppose to reach a sub-array of 1 element we have to do **k** iterations.

Then, $n / 2^k = 1$.

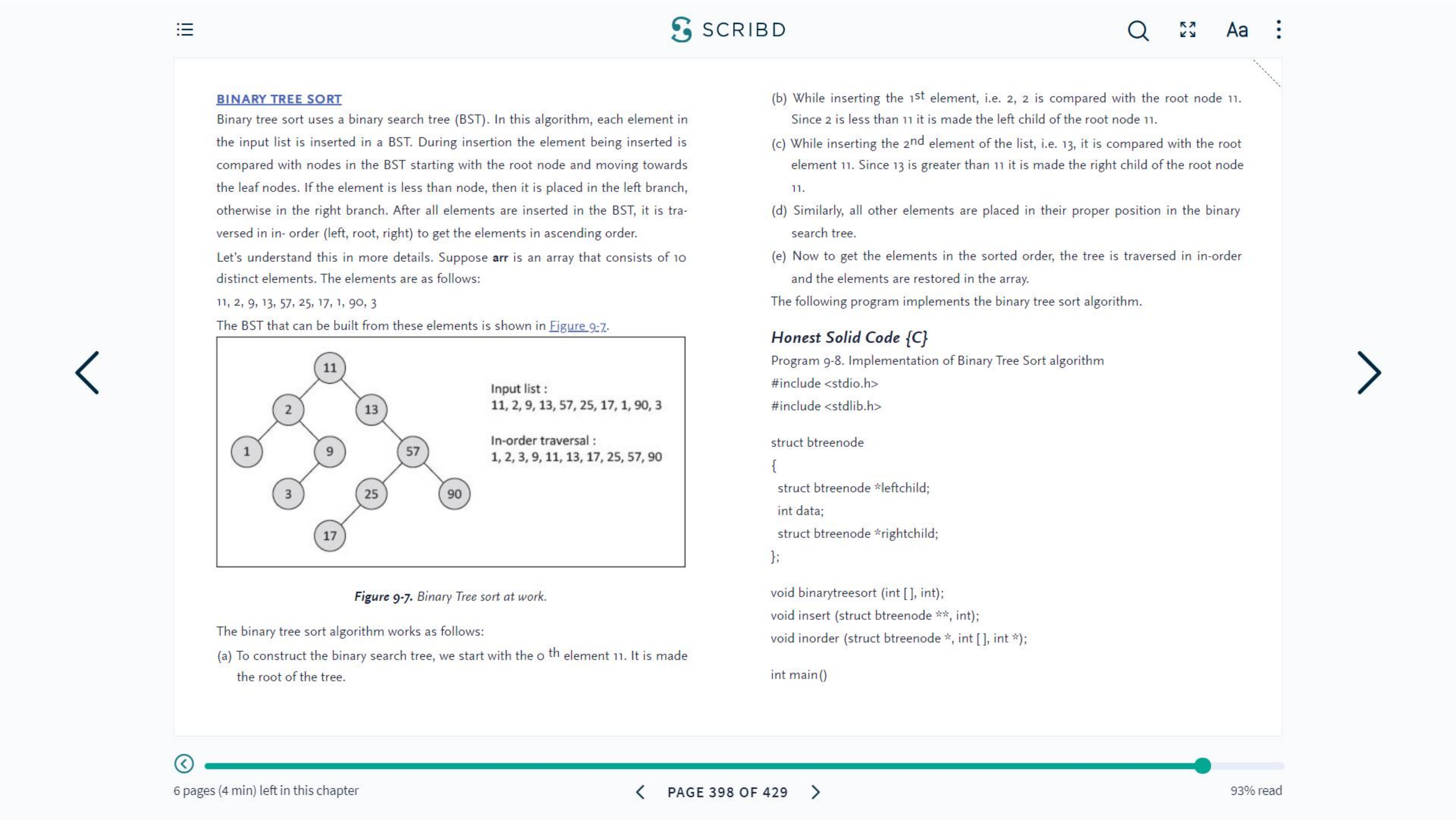
Taking log of both sides we get,

$\log_2 2^k = \log_2 n$

Therefore, $k = \log_2 n$.

In each of these **k** iterations for splitting the array we have to do **n** comparisons. Hence the total number of comparisons in quick sort will be **n * log$_2$ n**. So time complexity of quick sort in best case is **O (log$_2$ n)**.

The worst case in quick sort will occur when the input is an array which is already sorted. In this case if we take the first element as pivot then there won't be any left sub-array. Except the pivot, all elements will be in right sub-array. Same thing will happen at each level. So while splitting there will be **n** comparisons at level 1, **n - 1** comparison at level 2, n - 3 comparisons at level 3, etc. So totally there will be **n * (n + 1) / 2** comparisons. So time complexity will be **O (n^2)**.



BINARY TREE SORT

Binary tree sort uses a binary search tree (BST). In this algorithm, each element in the input list is inserted in a BST. During insertion the element being inserted is compared with nodes in the BST starting with the root node and moving towards the leaf nodes. If the element is less than node, then it is placed in the left branch, otherwise in the right branch. After all elements are inserted in the BST, it is traversed in in- order (left, root, right) to get the elements in ascending order.

Let's understand this in more details. Suppose **arr** is an array that consists of 10 distinct elements. The elements are as follows:

11, 2, 9, 13, 57, 25, 17, 1, 90, 3

The BST that can be built from these elements is shown in Figure 9-7.

Input list :
11, 2, 9, 13, 57, 25, 17, 1, 90, 3

In-order traversal :
1, 2, 3, 9, 11, 13, 17, 25, 57, 90

***Figure** 9-7. Binary Tree sort at work.*

The binary tree sort algorithm works as follows:

(a) To construct the binary search tree, we start with the 0 th element 11. It is made the root of the tree.

(b) While inserting the 1st element, i.e. 2, 2 is compared with the root node 11. Since 2 is less than 11 it is made the left child of the root node 11.

(c) While inserting the 2nd element of the list, i.e. 13, it is compared with the root element 11. Since 13 is greater than 11 it is made the right child of the root node 11.

(d) Similarly, all other elements are placed in their proper position in the binary search tree.

(e) Now to get the elements in the sorted order, the tree is traversed in in-order and the elements are restored in the array.

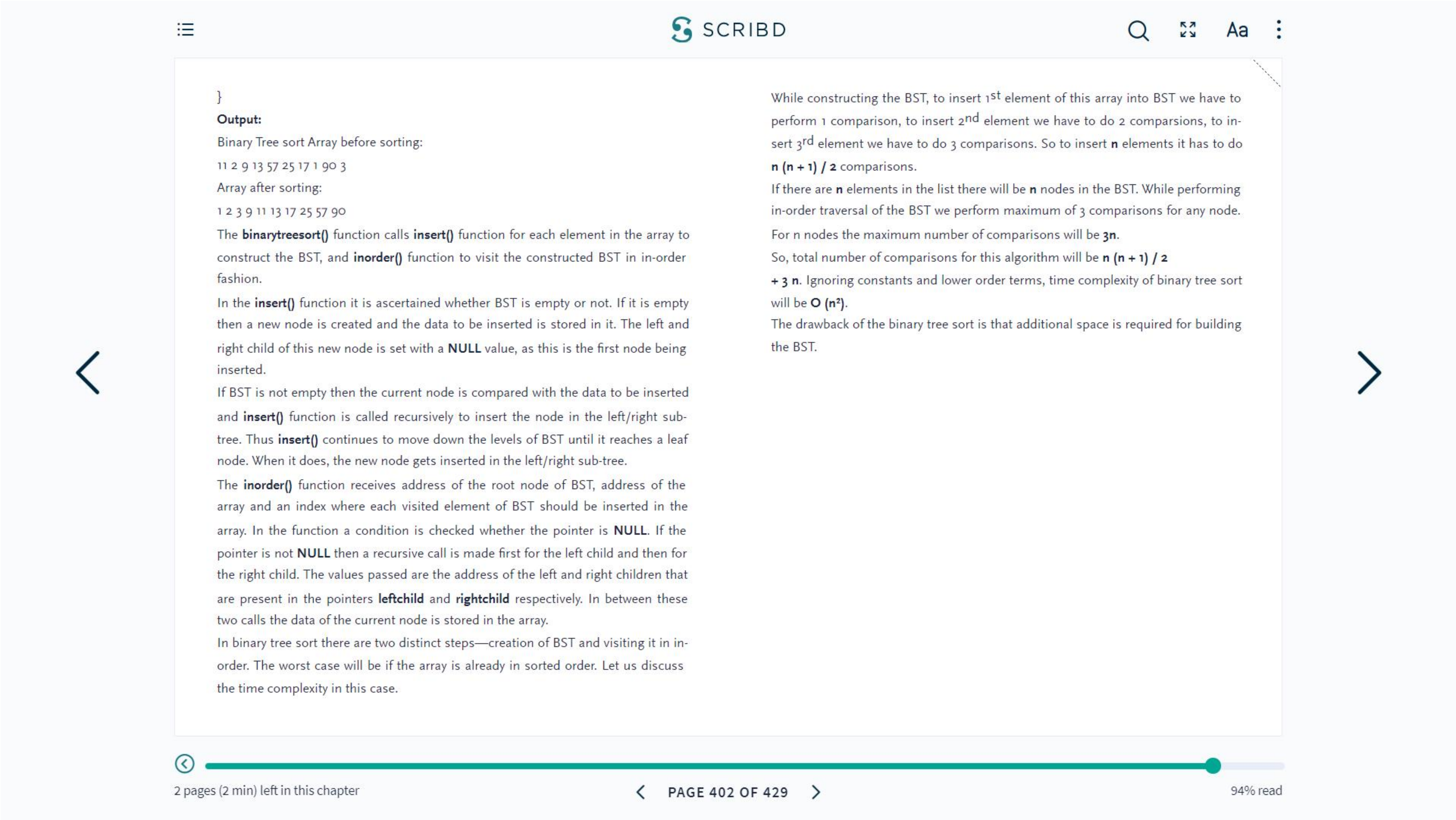
The following program implements the binary tree sort algorithm.

Honest Solid Code {C}

Program 9-8. Implementation of Binary Tree Sort algorithm

```
#include <stdio.h>
#include <stdlib.h>

struct btreenode
{
  struct btreenode *leftchild;
  int data;
  struct btreenode *rightchild;
};

void binarytreesort (int [ ], int);
void insert (struct btreenode **, int);
void inorder (struct btreenode *, int [ ], int *);

int main()
```

```
{
 int arr[ ] = { 11, 2, 9, 13, 57, 25, 17, 1, 90, 3 };
 int i;

 printf (“Binary Tree sort\n”);
 printf (“Array before sorting:\n”);
 for (i = 0; i < 10; i++)
   printf (“%d\t”, arr[ i ]);

 binarytreesort (arr, 10);

 printf (“Array after sorting:\n”);
 for (i = 0; i < 10; i++)
   printf (“%d\t”, arr[ i ]);

 return 0;
}

void binarytreesort (int a[ ], int size)
{
 struct btreenode *bt;
 int i;

 bt = NULL;
 for (i = 0; i < size; i++)
   insert (&bt, a[ i ]);

 i = 0;
 inorder (bt, a, &i);
}

void insert (struct btreenode **pr, int num)
{
 if (*pr == NULL)
 {
   *pr = (struct btreenode *) malloc (sizeof (struct btreenode));

   (*pr)->leftchild = NULL;
   (*pr)->data = num;
   (*pr)->rightchild = NULL;
 }
 else
 {
   if (num < (*pr)->data)
     insert (&((*pr)->leftchild), num);
   else
     insert (&((*pr)->rightchild), num);
 }
}

void inorder (struct btreenode *pr, int a[ ], int *p)
{
 if (pr != NULL)
 {
   inorder (pr->leftchild, a, p);
   a[ *p ] = pr->data;
   *p = *p + 1;
   inorder (pr->rightchild, a, p);
 }
```



}

Output:

```
Binary Tree sort Array before sorting:
11 2 9 13 57 25 17 1 90 3
Array after sorting:
1 2 3 9 11 13 17 25 57 90
```

The **binarytreesort()** function calls **insert()** function for each element in the array to construct the BST, and **inorder()** function to visit the constructed BST in in-order fashion.

In the **insert()** function it is ascertained whether BST is empty or not. If it is empty then a new node is created and the data to be inserted is stored in it. The left and right child of this new node is set with a **NULL** value, as this is the first node being inserted.

If BST is not empty then the current node is compared with the data to be inserted and **insert()** function is called recursively to insert the node in the left/right sub-tree. Thus **insert()** continues to move down the levels of BST until it reaches a leaf node. When it does, the new node gets inserted in the left/right sub-tree.

The **inorder()** function receives address of the root node of BST, address of the array and an index where each visited element of BST should be inserted in the array. In the function a condition is checked whether the pointer is **NULL**. If the pointer is not **NULL** then a recursive call is made first for the left child and then for the right child. The values passed are the address of the left and right children that are present in the pointers **leftchild** and **rightchild** respectively. In between these two calls the data of the current node is stored in the array.

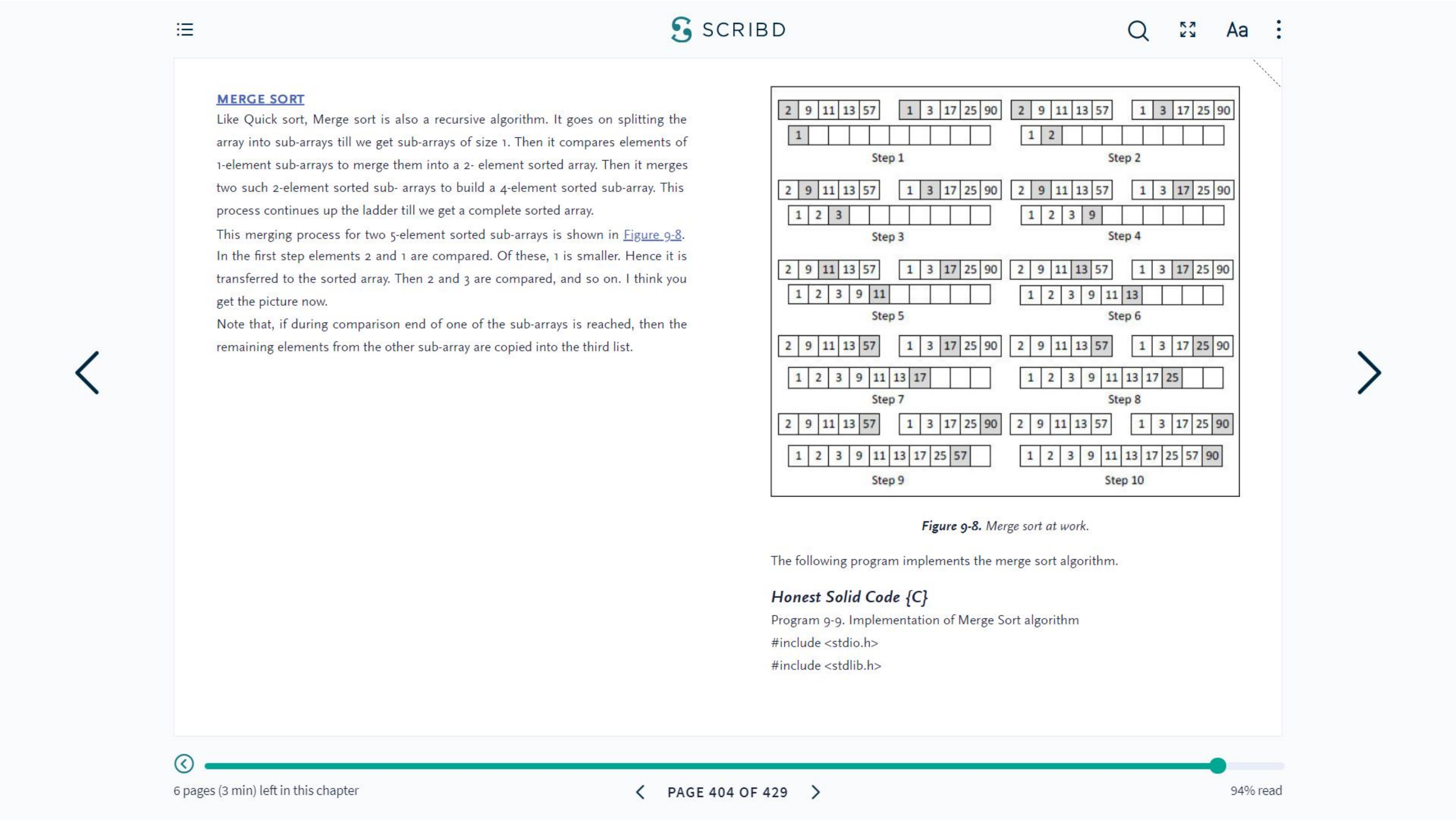
In binary tree sort there are two distinct steps—creation of BST and visiting it in in-order. The worst case will be if the array is already in sorted order. Let us discuss the time complexity in this case.

While constructing the BST, to insert 1st element of this array into BST we have to perform 1 comparison, to insert 2nd element we have to do 2 comparsions, to insert 3rd element we have to do 3 comparisons. So to insert **n** elements it has to do **n (n + 1) / 2** comparisons.

If there are **n** elements in the list there will be **n** nodes in the BST. While performing in-order traversal of the BST we perform maximum of 3 comparisons for any node. For n nodes the maximum number of comparisons will be **3n**.

So, total number of comparisons for this algorithm will be **n (n + 1) / 2 + 3 n**. Ignoring constants and lower order terms, time complexity of binary tree sort will be **O (n^2)**.

The drawback of the binary tree sort is that additional space is required for building the BST.



MERGE SORT

Like Quick sort, Merge sort is also a recursive algorithm. It goes on splitting the array into sub-arrays till we get sub-arrays of size 1. Then it compares elements of 1-element sub-arrays to merge them into a 2- element sorted array. Then it merges two such 2-element sorted sub- arrays to build a 4-element sorted sub-array. This process continues up the ladder till we get a complete sorted array.

This merging process for two 5-element sorted sub-arrays is shown in Figure 9-8. In the first step elements 2 and 1 are compared. Of these, 1 is smaller. Hence it is transferred to the sorted array. Then 2 and 3 are compared, and so on. I think you get the picture now.

Note that, if during comparison end of one of the sub-arrays is reached, then the remaining elements from the other sub-array are copied into the third list.

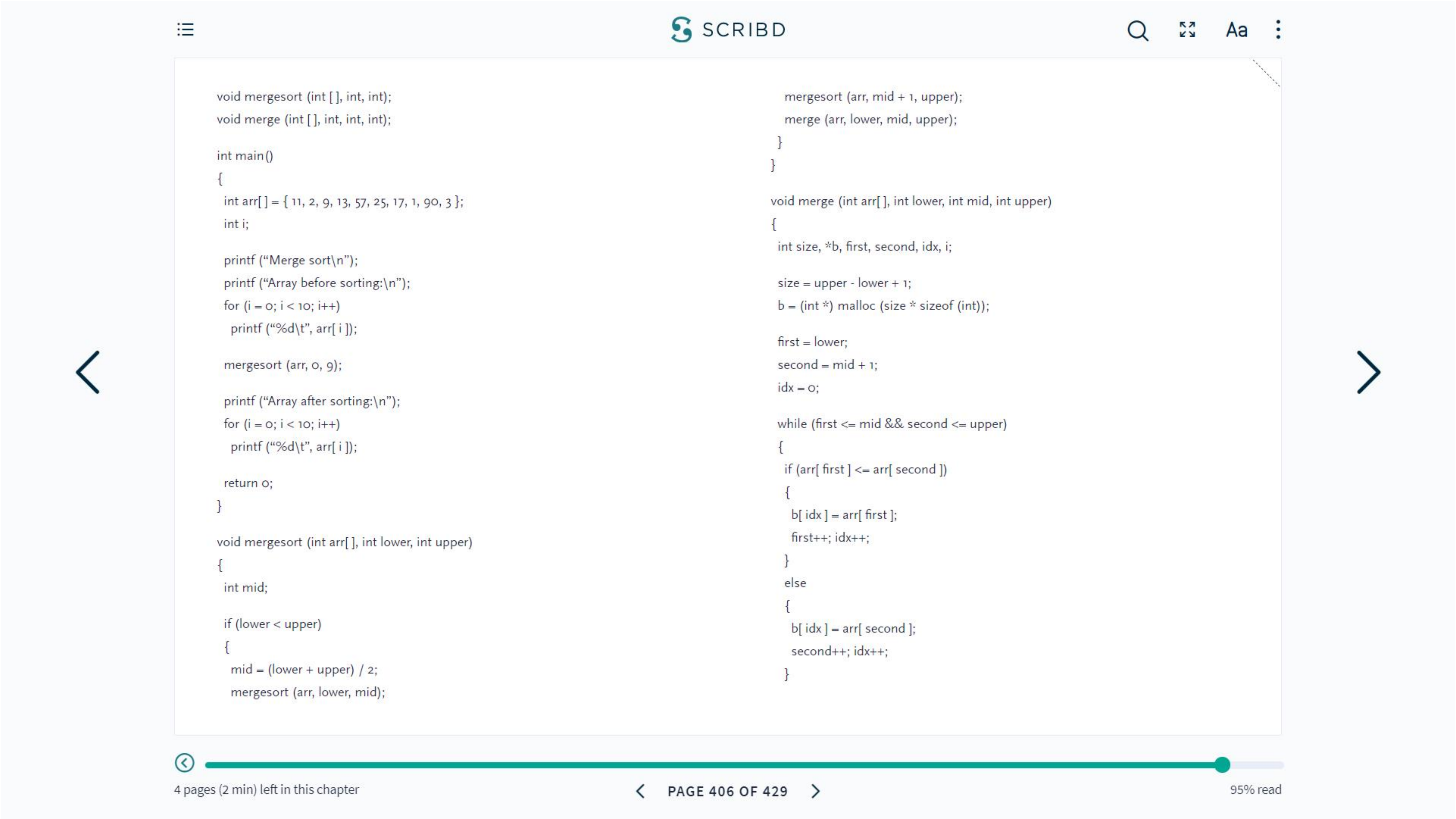
Figure 9-8. *Merge sort at work.*

The following program implements the merge sort algorithm.

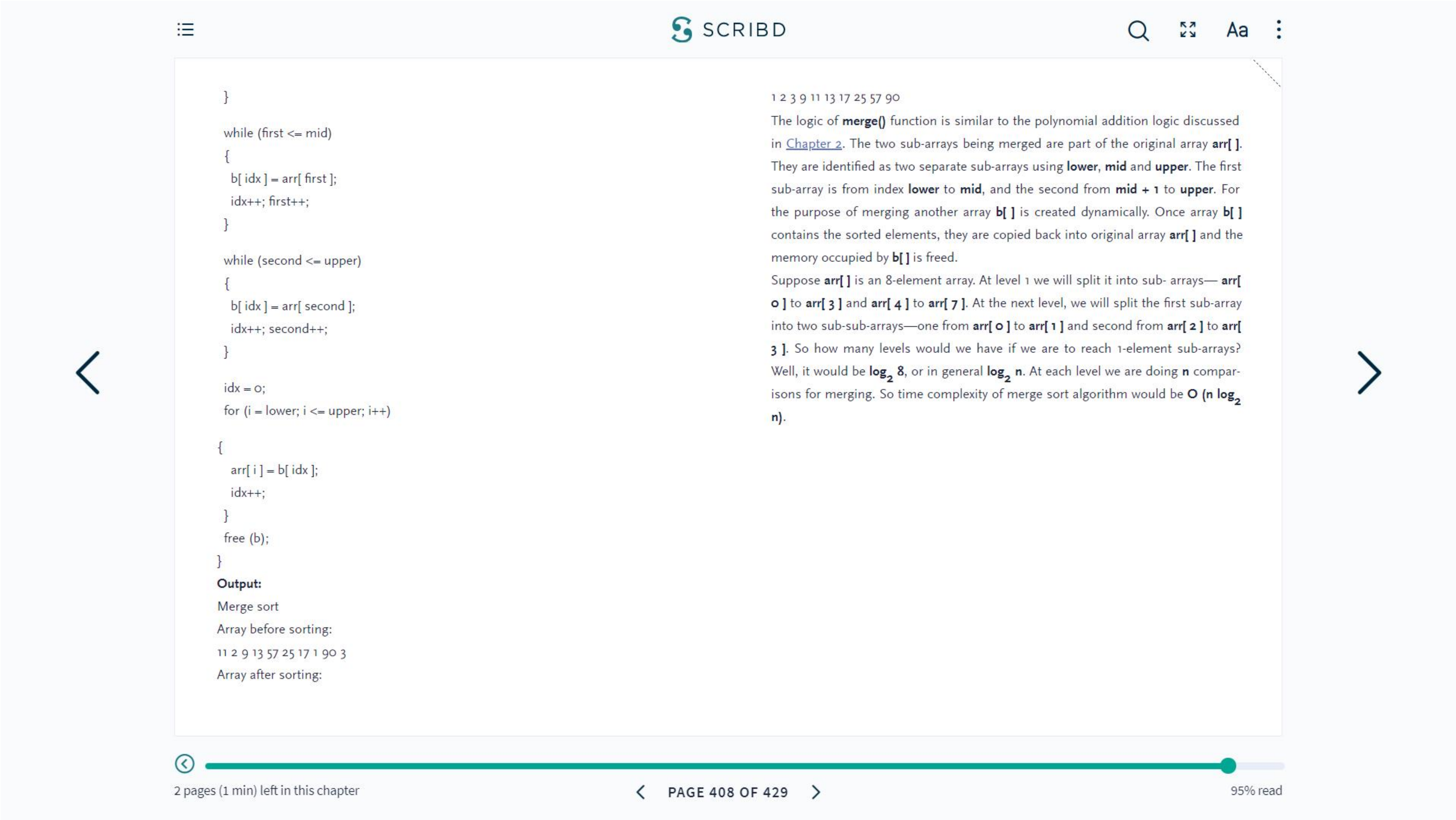
Honest Solid Code {C}

Program 9-9. Implementation of Merge Sort algorithm

```
#include <stdio.h>
#include <stdlib.h>
```



```
void mergesort (int [ ], int, int);
void merge (int [ ], int, int, int);

int main()
{
 int arr[ ] = { 11, 2, 9, 13, 57, 25, 17, 1, 90, 3 };
 int i;

 printf (“Merge sort\n”);
 printf (“Array before sorting:\n”);
 for (i = 0; i < 10; i++)
  printf (“%d\t”, arr[ i ]);

 mergesort (arr, 0, 9);

 printf (“Array after sorting:\n”);
 for (i = 0; i < 10; i++)
  printf (“%d\t”, arr[ i ]);

 return 0;
}

void mergesort (int arr[ ], int lower, int upper)
{
 int mid;

 if (lower < upper)
 {
  mid = (lower + upper) / 2;
  mergesort (arr, lower, mid);
  mergesort (arr, mid + 1, upper);
  merge (arr, lower, mid, upper);
 }
}

void merge (int arr[ ], int lower, int mid, int upper)
{
 int size, *b, first, second, idx, i;

 size = upper - lower + 1;
 b = (int *) malloc (size * sizeof (int));

 first = lower;
 second = mid + 1;
 idx = 0;

 while (first <= mid && second <= upper)
 {
  if (arr[ first ] <= arr[ second ])
  {
   b[ idx ] = arr[ first ];
   first++; idx++;
  }
  else
  {
   b[ idx ] = arr[ second ];
   second++; idx++;
  }
```



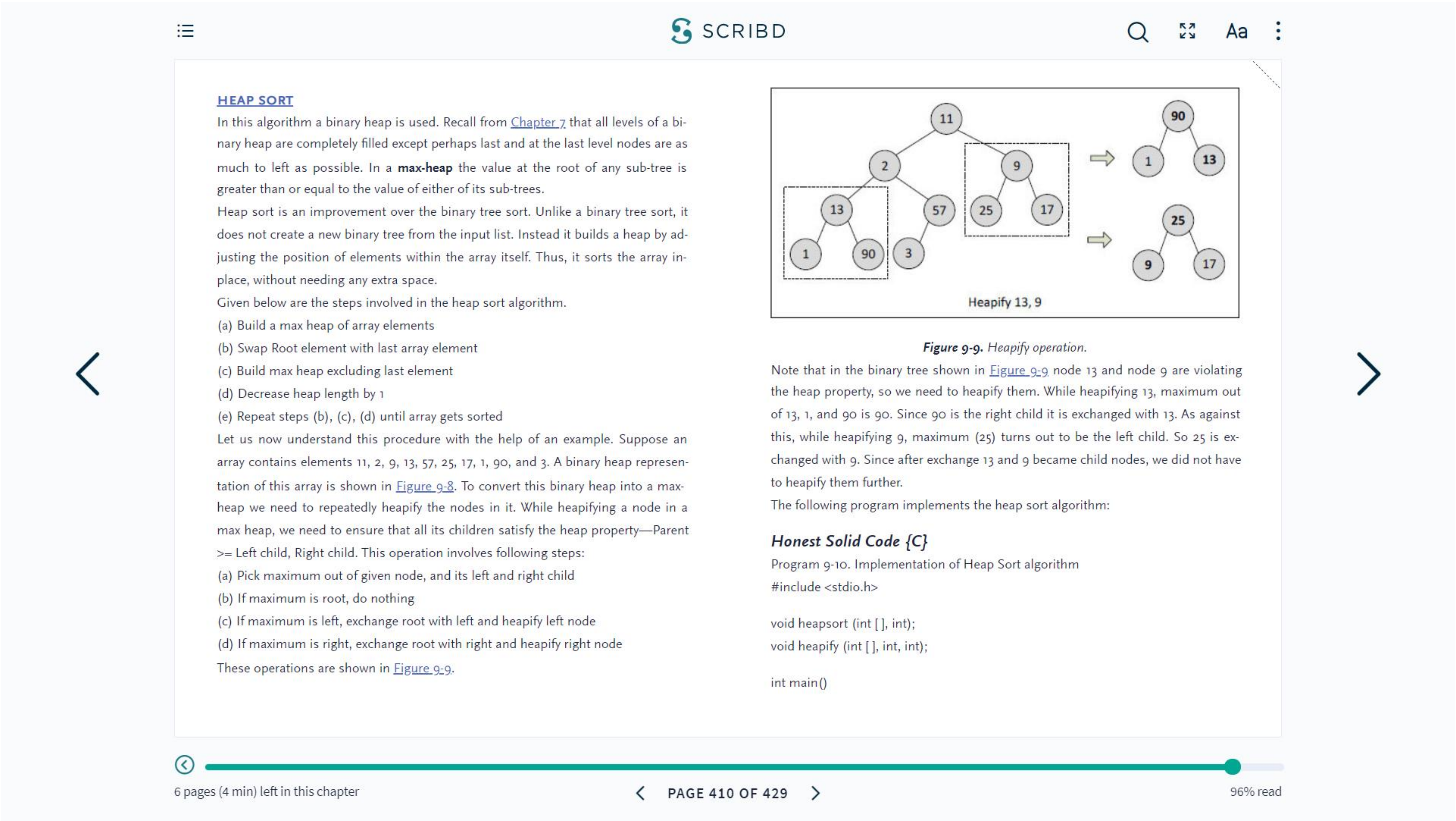
```
  }

  while (first <= mid)
  {
   b[ idx ] = arr[ first ];
   idx++; first++;
  }

  while (second <= upper)
  {
   b[ idx ] = arr[ second ];
   idx++; second++;
  }

  idx = 0;
  for (i = lower; i <= upper; i++)

 {
   arr[ i ] = b[ idx ];
   idx++;
  }
  free (b);
 }
```

Output:

```
Merge sort
Array before sorting:
11 2 9 13 57 25 17 1 90 3
Array after sorting:
1 2 3 9 11 13 17 25 57 90
```

The logic of **merge()** function is similar to the polynomial addition logic discussed in Chapter 2. The two sub-arrays being merged are part of the original array **arr[ ]**. They are identified as two separate sub-arrays using **lower**, **mid** and **upper**. The first sub-array is from index **lower** to **mid**, and the second from **mid + 1** to **upper**. For the purpose of merging another array **b[ ]** is created dynamically. Once array **b[ ]** contains the sorted elements, they are copied back into original array **arr[ ]** and the memory occupied by **b[ ]** is freed.

Suppose **arr[ ]** is an 8-element array. At level 1 we will split it into sub- arrays— **arr[0]** to **arr[3]** and **arr[4]** to **arr[7]**. At the next level, we will split the first sub-array into two sub-sub-arrays—one from **arr[0]** to **arr[1]** and second from **arr[2]** to **arr[3]**. So how many levels would we have if we are to reach 1-element sub-arrays? Well, it would be **$\log_2$ 8**, or in general **$\log_2$ n**. At each level we are doing **n** comparisons for merging. So time complexity of merge sort algorithm would be **O (n $\log_2$ n)**.



HEAP SORT

In this algorithm a binary heap is used. Recall from Chapter 7 that all levels of a binary heap are completely filled except perhaps last and at the last level nodes are as much to left as possible. In a **max-heap** the value at the root of any sub-tree is greater than or equal to the value of either of its sub-trees.

Heap sort is an improvement over the binary tree sort. Unlike a binary tree sort, it does not create a new binary tree from the input list. Instead it builds a heap by adjusting the position of elements within the array itself. Thus, it sorts the array in-place, without needing any extra space.

Given below are the steps involved in the heap sort algorithm.

(a) Build a max heap of array elements

(b) Swap Root element with last array element

(c) Build max heap excluding last element

(d) Decrease heap length by 1

(e) Repeat steps (b), (c), (d) until array gets sorted

Let us now understand this procedure with the help of an example. Suppose an array contains elements 11, 2, 9, 13, 57, 25, 17, 1, 90, and 3. A binary heap representation of this array is shown in Figure 9-8. To convert this binary heap into a max-heap we need to repeatedly heapify the nodes in it. While heapifying a node in a max heap, we need to ensure that all its children satisfy the heap property—Parent >= Left child, Right child. This operation involves following steps:

(a) Pick maximum out of given node, and its left and right child

(b) If maximum is root, do nothing

(c) If maximum is left, exchange root with left and heapify left node

(d) If maximum is right, exchange root with right and heapify right node

These operations are shown in Figure 9-9.

Figure 9-9. *Heapify operation.*

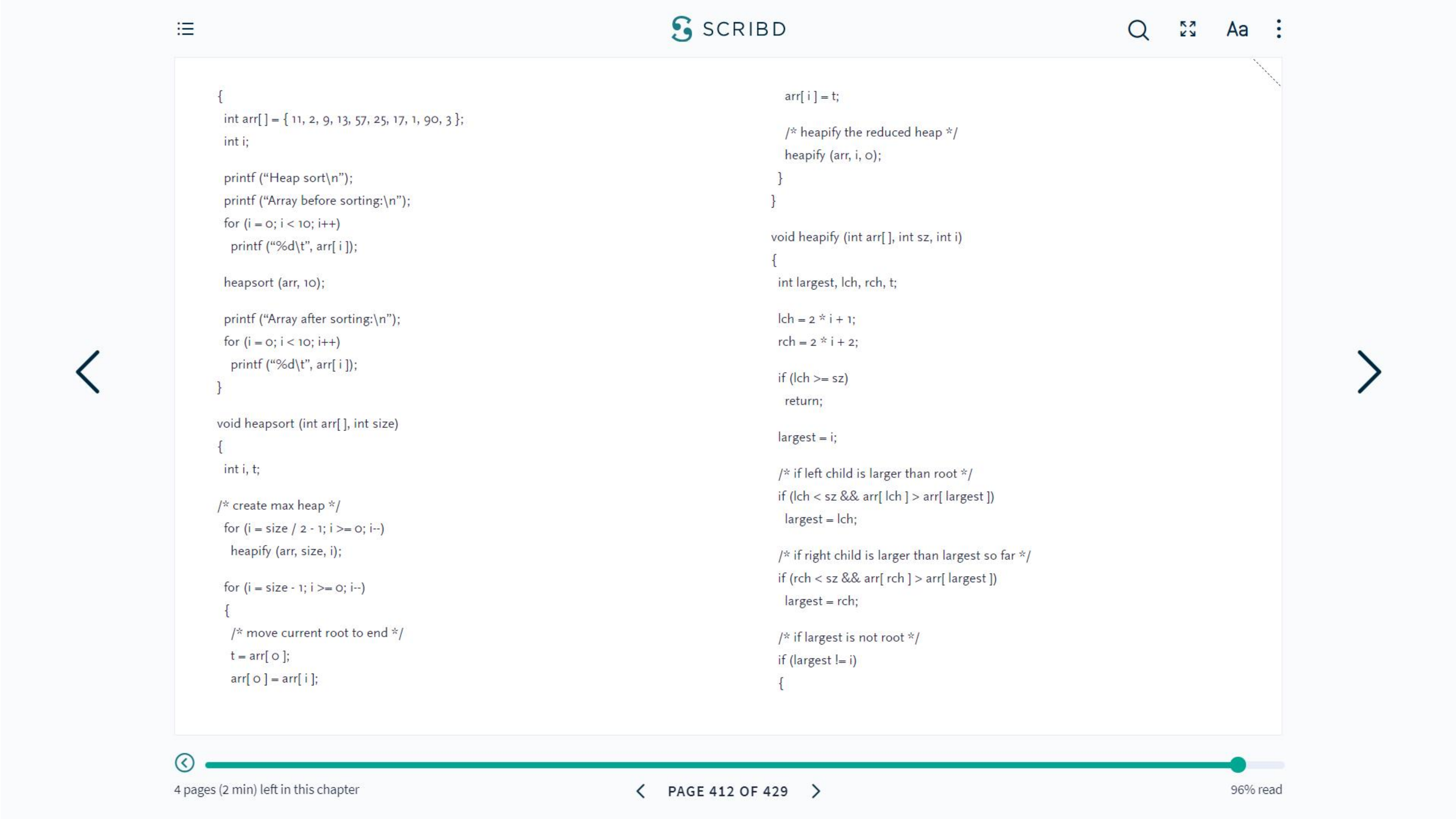
Note that in the binary tree shown in Figure 9-9 node 13 and node 9 are violating the heap property, so we need to heapify them. While heapifying 13, maximum out of 13, 1, and 90 is 90. Since 90 is the right child it is exchanged with 13. As against this, while heapifying 9, maximum (25) turns out to be the left child. So 25 is exchanged with 9. Since after exchange 13 and 9 became child nodes, we did not have to heapify them further.

The following program implements the heap sort algorithm:

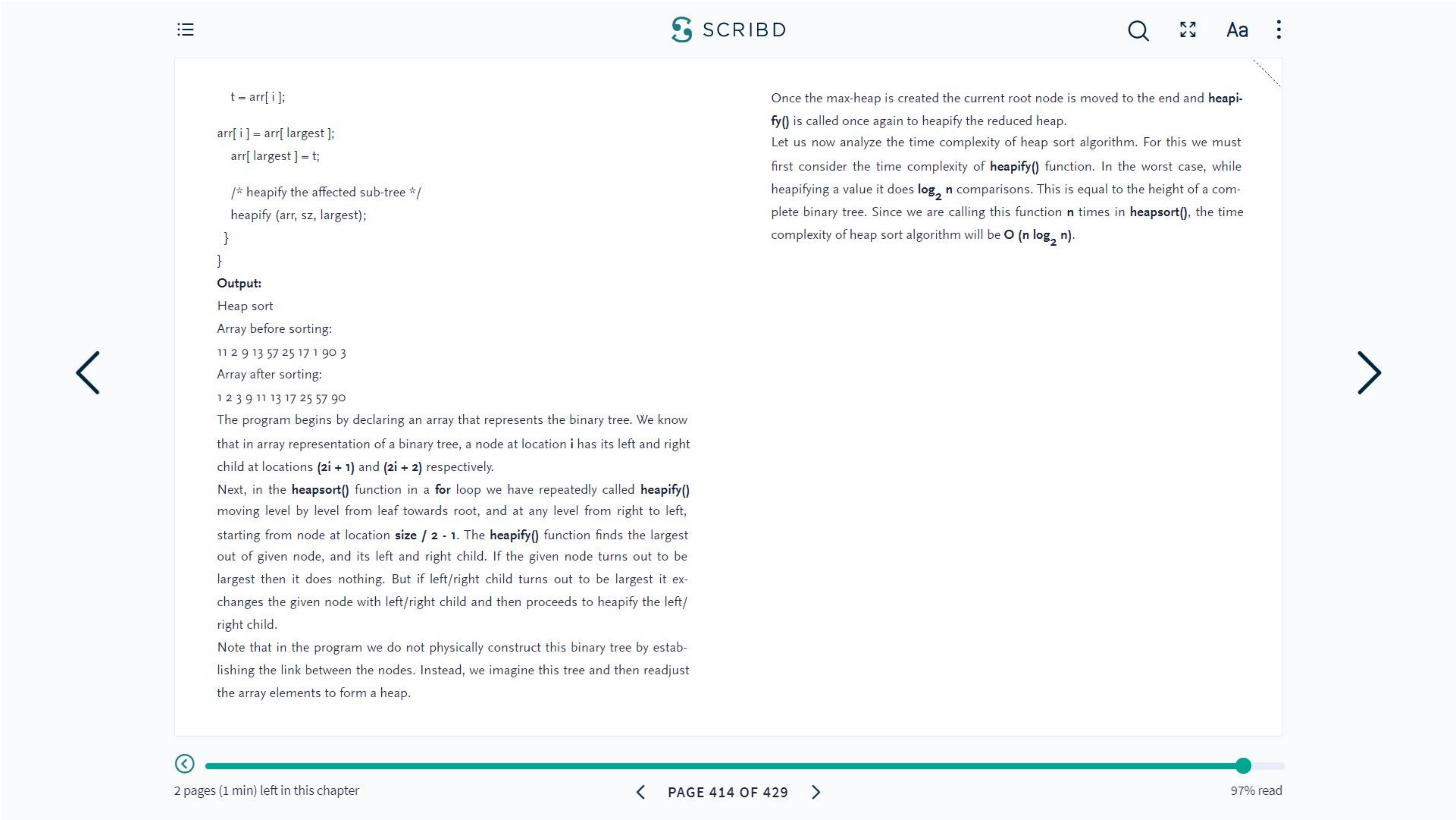
Honest Solid Code {C}

Program 9-10. Implementation of Heap Sort algorithm

```
#include <stdio.h>

void heapsort (int [ ], int);
void heapify (int [ ], int, int);

int main()
```



```
{
 int arr[ ] = { 11, 2, 9, 13, 57, 25, 17, 1, 90, 3 };
 int i;

 printf (“Heap sort\n”);
 printf (“Array before sorting:\n”);
 for (i = 0; i < 10; i++)
  printf (“%d\t”, arr[ i ]);

 heapsort (arr, 10);

 printf (“Array after sorting:\n”);
 for (i = 0; i < 10; i++)
  printf (“%d\t”, arr[ i ]);
}

void heapsort (int arr[ ], int size)
{
 int i, t;

/* create max heap */
 for (i = size / 2 - 1; i >= 0; i--)
  heapify (arr, size, i);

 for (i = size - 1; i >= 0; i--)
 {
  /* move current root to end */
  t = arr[ 0 ];
  arr[ 0 ] = arr[ i ];
  arr[ i ] = t;

  /* heapify the reduced heap */
  heapify (arr, i, 0);
 }
}

void heapify (int arr[ ], int sz, int i)
{
 int largest, lch, rch, t;

 lch = 2 * i + 1;
 rch = 2 * i + 2;

 if (lch >= sz)
  return;

 largest = i;

 /* if left child is larger than root */
 if (lch < sz && arr[ lch ] > arr[ largest ])
  largest = lch;

 /* if right child is larger than largest so far */
 if (rch < sz && arr[ rch ] > arr[ largest ])
  largest = rch;

 /* if largest is not root */
 if (largest != i)
 {
```



```
    t = arr[ i ];

  arr[ i ] = arr[ largest ];
    arr[ largest ] = t;

    /* heapify the affected sub-tree */
    heapify (arr, sz, largest);
  }
}
```

Output:

```
Heap sort
Array before sorting:
11 2 9 13 57 25 17 1 90 3
Array after sorting:
1 2 3 9 11 13 17 25 57 90
```

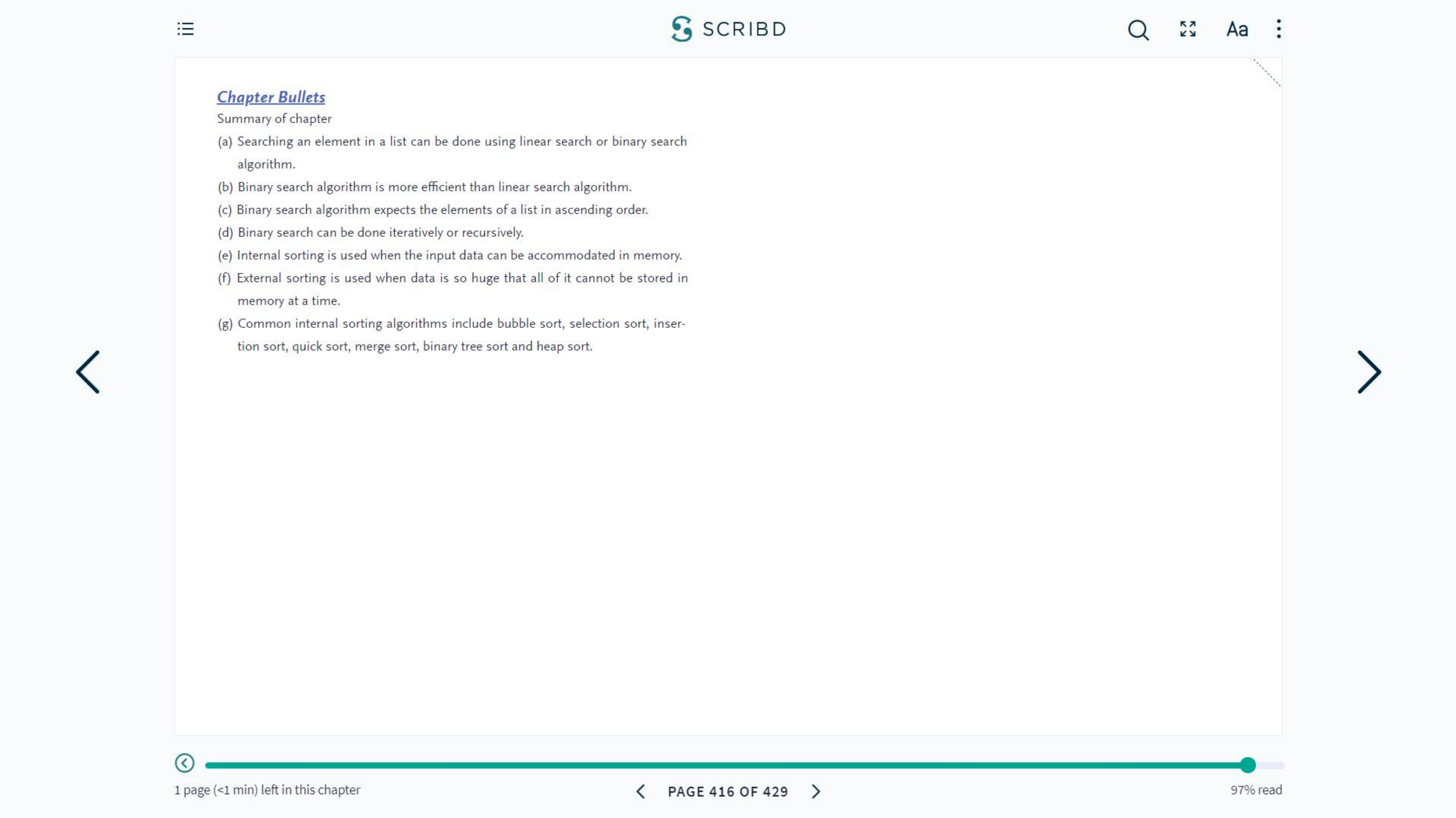
The program begins by declaring an array that represents the binary tree. We know that in array representation of a binary tree, a node at location **i** has its left and right child at locations **(2i + 1)** and **(2i + 2)** respectively.

Next, in the **heapsort()** function in a **for** loop we have repeatedly called **heapify()** moving level by level from leaf towards root, and at any level from right to left, starting from node at location **size / 2 - 1**. The **heapify()** function finds the largest out of given node, and its left and right child. If the given node turns out to be largest then it does nothing. But if left/right child turns out to be largest it exchanges the given node with left/right child and then proceeds to heapify the left/right child.

Note that in the program we do not physically construct this binary tree by establishing the link between the nodes. Instead, we imagine this tree and then readjust the array elements to form a heap.

Once the max-heap is created the current root node is moved to the end and **heapify()** is called once again to heapify the reduced heap.

Let us now analyze the time complexity of heap sort algorithm. For this we must first consider the time complexity of **heapify()** function. In the worst case, while heapifying a value it does $\log_2 n$ comparisons. This is equal to the height of a complete binary tree. Since we are calling this function **n** times in **heapsort()**, the time complexity of heap sort algorithm will be **O** $(n \log_2 n)$.



Chapter Bullets

Summary of chapter

(a) Searching an element in a list can be done using linear search or binary search algorithm.

(b) Binary search algorithm is more efficient than linear search algorithm.

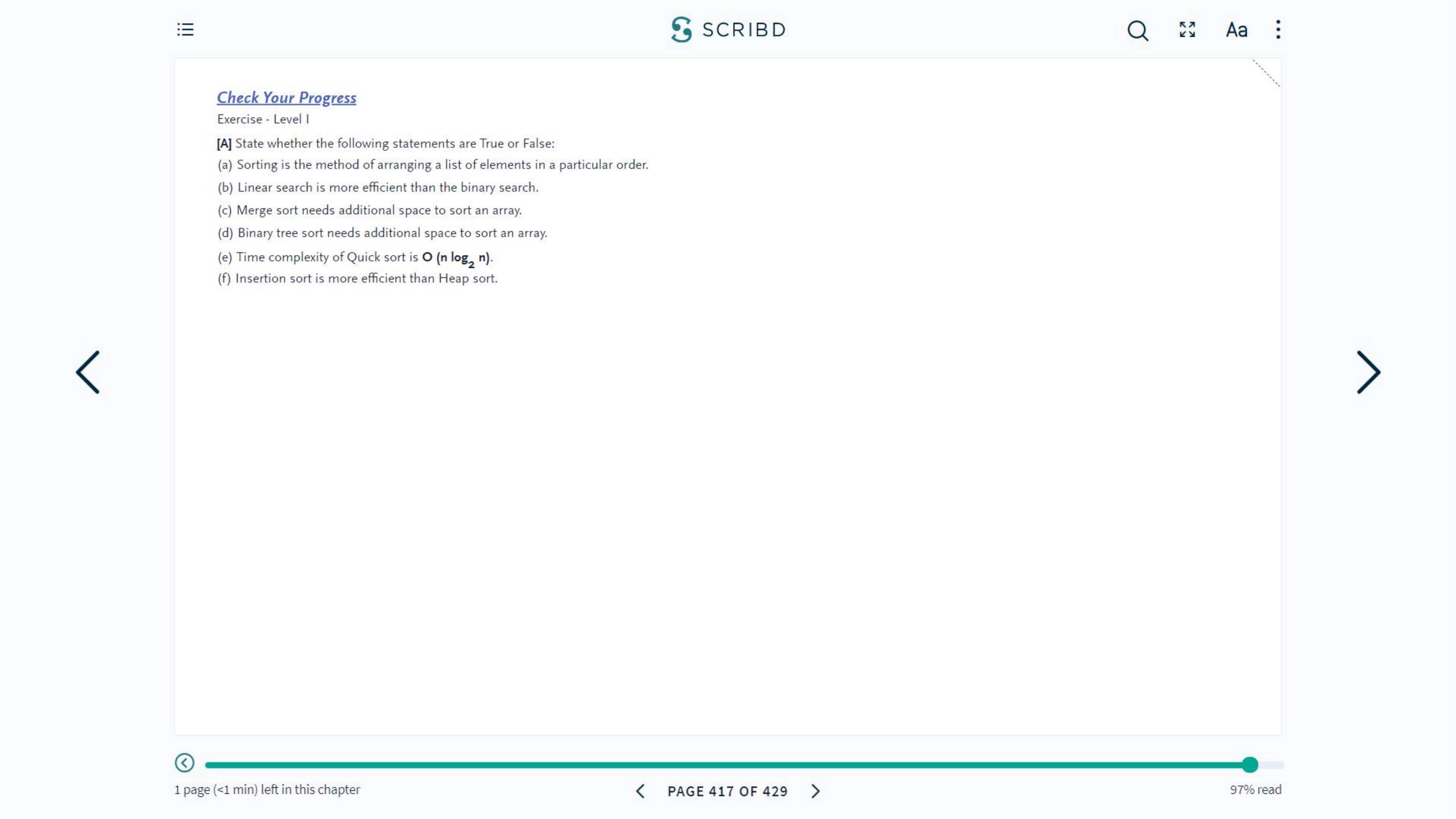
(c) Binary search algorithm expects the elements of a list in ascending order.

(d) Binary search can be done iteratively or recursively.

(e) Internal sorting is used when the input data can be accommodated in memory.

(f) External sorting is used when data is so huge that all of it cannot be stored in memory at a time.

(g) Common internal sorting algorithms include bubble sort, selection sort, insertion sort, quick sort, merge sort, binary tree sort and heap sort.



Check Your Progress

Exercise - Level I

[A] State whether the following statements are True or False:

(a) Sorting is the method of arranging a list of elements in a particular order.

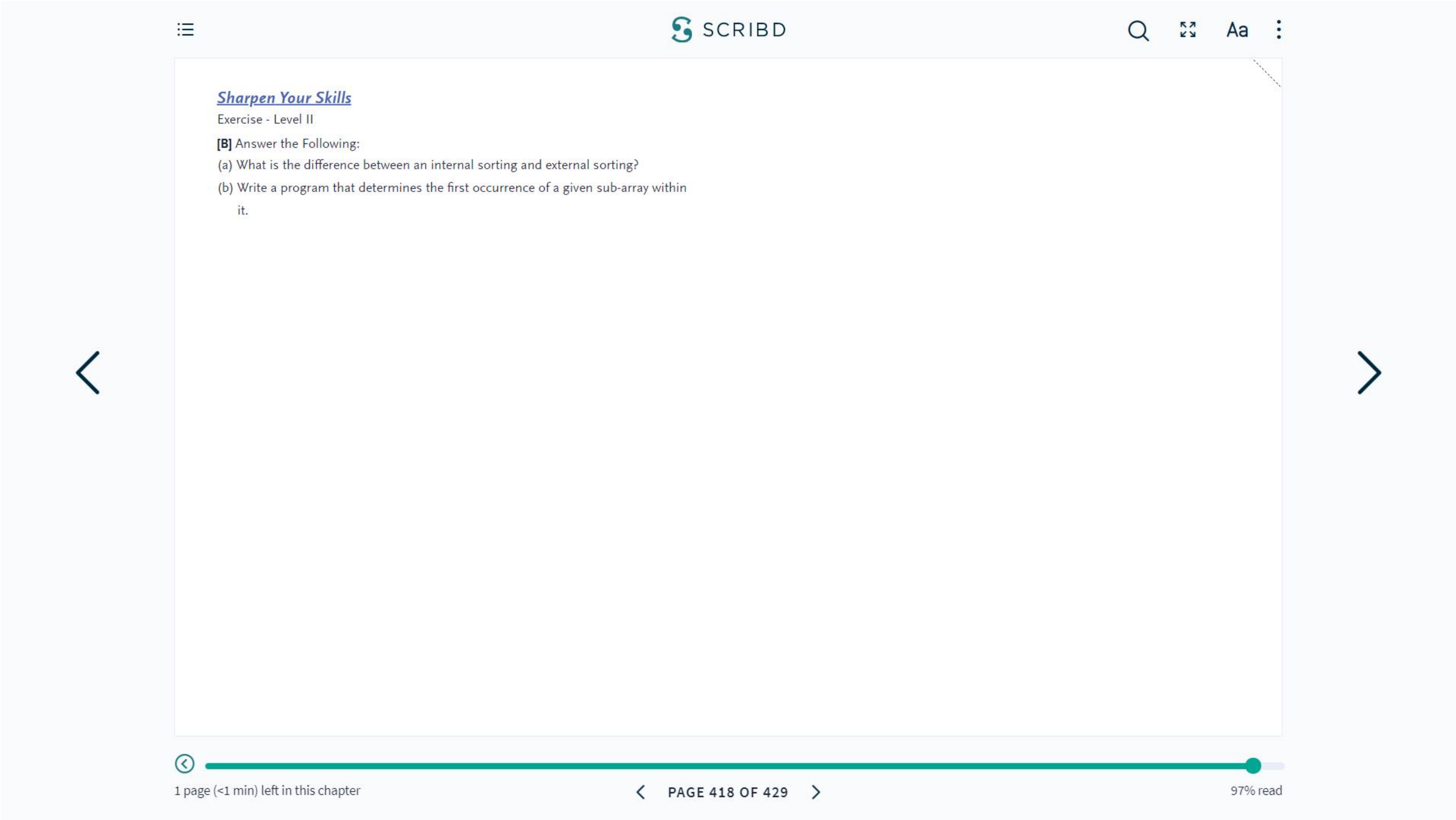
(b) Linear search is more efficient than the binary search.

(c) Merge sort needs additional space to sort an array.

(d) Binary tree sort needs additional space to sort an array.

(e) Time complexity of Quick sort is $\mathbf{O\ (n \log_2 n)}$.

(f) Insertion sort is more efficient than Heap sort.



Sharpen Your Skills

Exercise - Level II

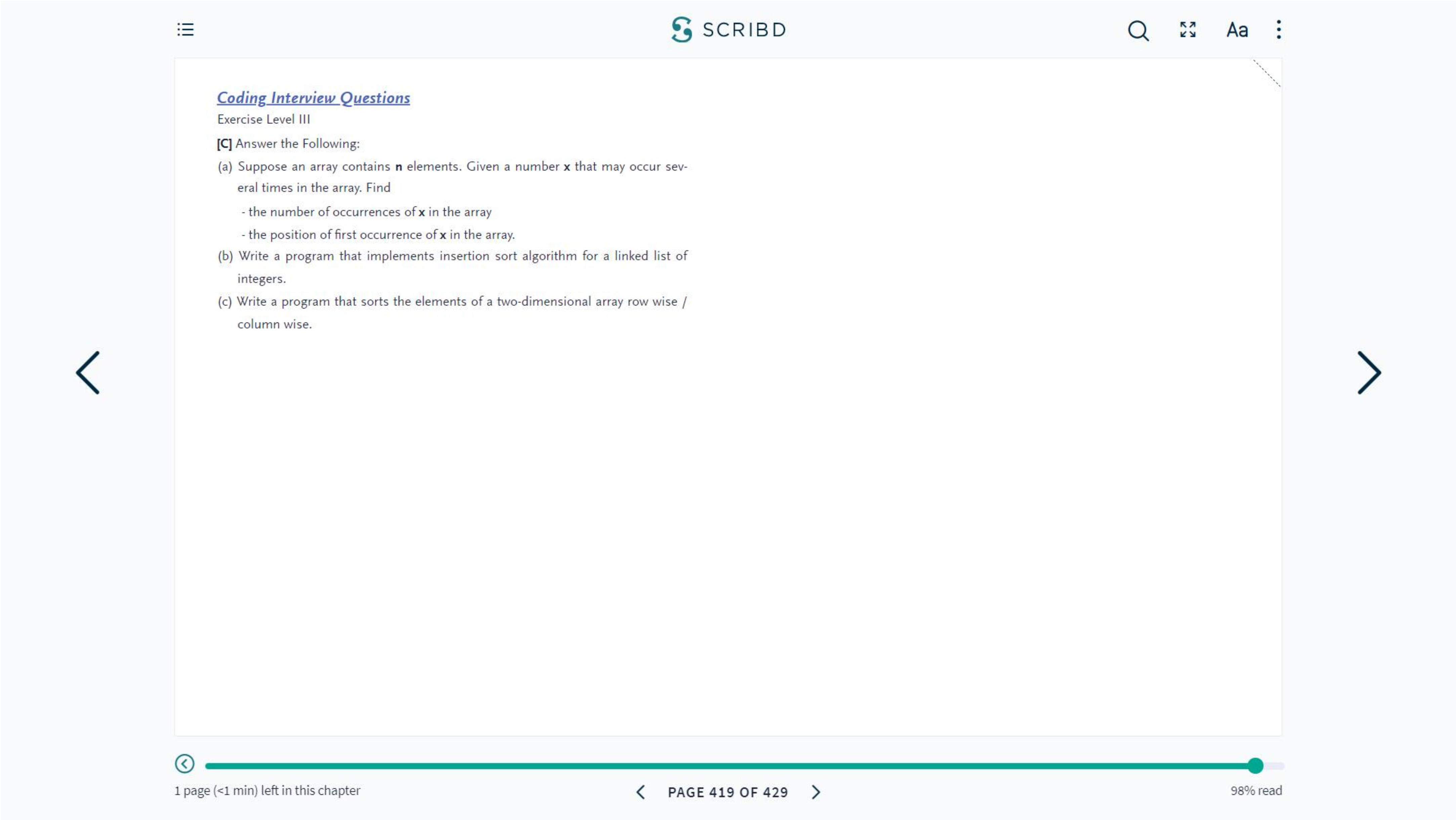
[B] Answer the Following:

(a) What is the difference between an internal sorting and external sorting?

(b) Write a program that determines the first occurrence of a given sub-array within it.

1 page (<1 min) left in this chapter

97% read



Coding Interview Questions

Exercise Level III

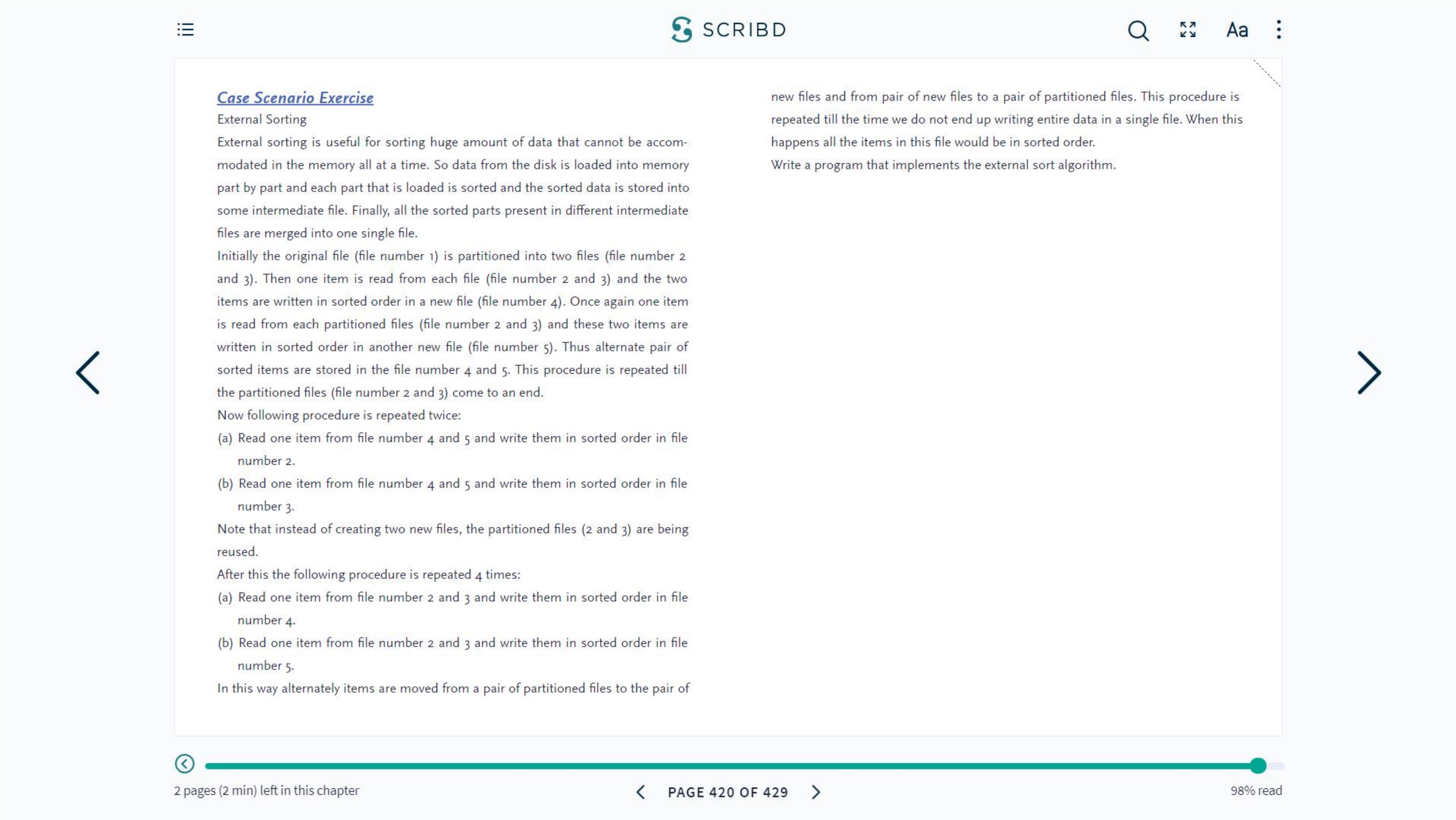
[C] Answer the Following:

(a) Suppose an array contains **n** elements. Given a number **x** that may occur several times in the array. Find

- the number of occurrences of **x** in the array
- the position of first occurrence of **x** in the array.

(b) Write a program that implements insertion sort algorithm for a linked list of integers.

(c) Write a program that sorts the elements of a two-dimensional array row wise / column wise.



Case Scenario Exercise

External Sorting

External sorting is useful for sorting huge amount of data that cannot be accommodated in the memory all at a time. So data from the disk is loaded into memory part by part and each part that is loaded is sorted and the sorted data is stored into some intermediate file. Finally, all the sorted parts present in different intermediate files are merged into one single file.

Initially the original file (file number 1) is partitioned into two files (file number 2 and 3). Then one item is read from each file (file number 2 and 3) and the two items are written in sorted order in a new file (file number 4). Once again one item is read from each partitioned files (file number 2 and 3) and these two items are written in sorted order in another new file (file number 5). Thus alternate pair of sorted items are stored in the file number 4 and 5. This procedure is repeated till the partitioned files (file number 2 and 3) come to an end.

Now following procedure is repeated twice:

(a) Read one item from file number 4 and 5 and write them in sorted order in file number 2.

(b) Read one item from file number 4 and 5 and write them in sorted order in file number 3.

Note that instead of creating two new files, the partitioned files (2 and 3) are being reused.

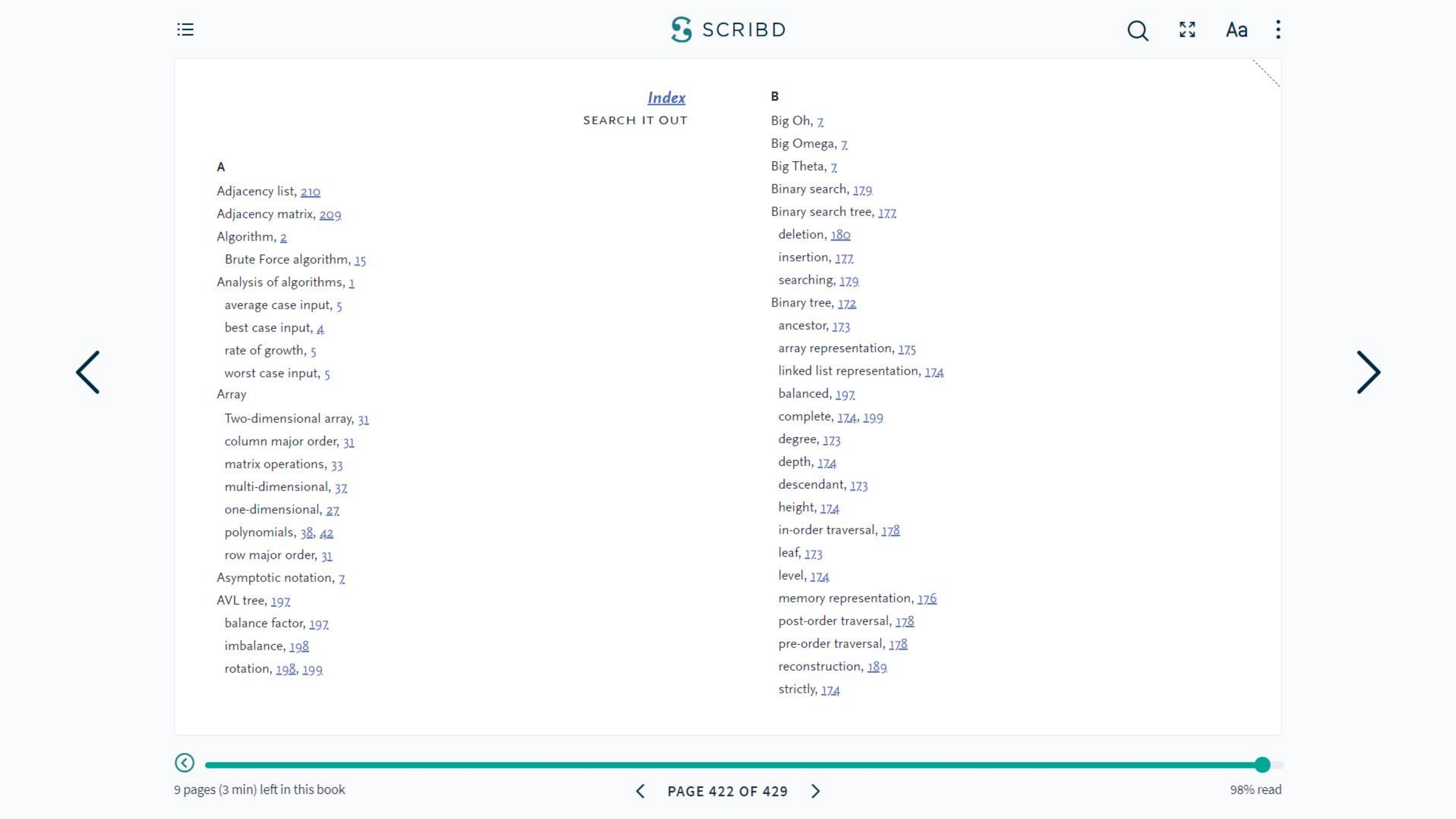
After this the following procedure is repeated 4 times:

(a) Read one item from file number 2 and 3 and write them in sorted order in file number 4.

(b) Read one item from file number 2 and 3 and write them in sorted order in file number 5.

In this way alternately items are moved from a pair of partitioned files to the pair of new files and from pair of new files to a pair of partitioned files. This procedure is repeated till the time we do not end up writing entire data in a single file. When this happens all the items in this file would be in sorted order.

Write a program that implements the external sort algorithm.



Index

SEARCH IT OUT

A

Adjacency list, 210
Adjacency matrix, 209
Algorithm, 2
  Brute Force algorithm, 15
Analysis of algorithms, 1
  average case input, 5
  best case input, 4
  rate of growth, 5
  worst case input, 5
Array
  Two-dimensional array, 31
  column major order, 31
  matrix operations, 33
  multi-dimensional, 37
  one-dimensional, 27
  polynomials, 38, 42
  row major order, 31
Asymptotic notation, 7
AVL tree, 197
  balance factor, 197
  imbalance, 198
  rotation, 198, 199

B

Big Oh, 7
Big Omega, 7
Big Theta, 7
Binary search, 179
Binary search tree, 177
  deletion, 180
  insertion, 177
  searching, 179
Binary tree, 172
  ancestor, 173
  array representation, 175
  linked list representation, 174
  balanced, 197
  complete, 174, 199
  degree, 173
  depth, 174
  descendant, 173
  height, 174
  in-order traversal, 178
  leaf, 173
  level, 174
  memory representation, 176
  post-order traversal, 178
  pre-order traversal, 178
  reconstruction, 189
  strictly, 174

  threaded, 191
Binary tree sort, 257
Breadth frist search, 216
Brute Force algorithm, 15
Bubble sort, 242

C
Circular
  queue, 161

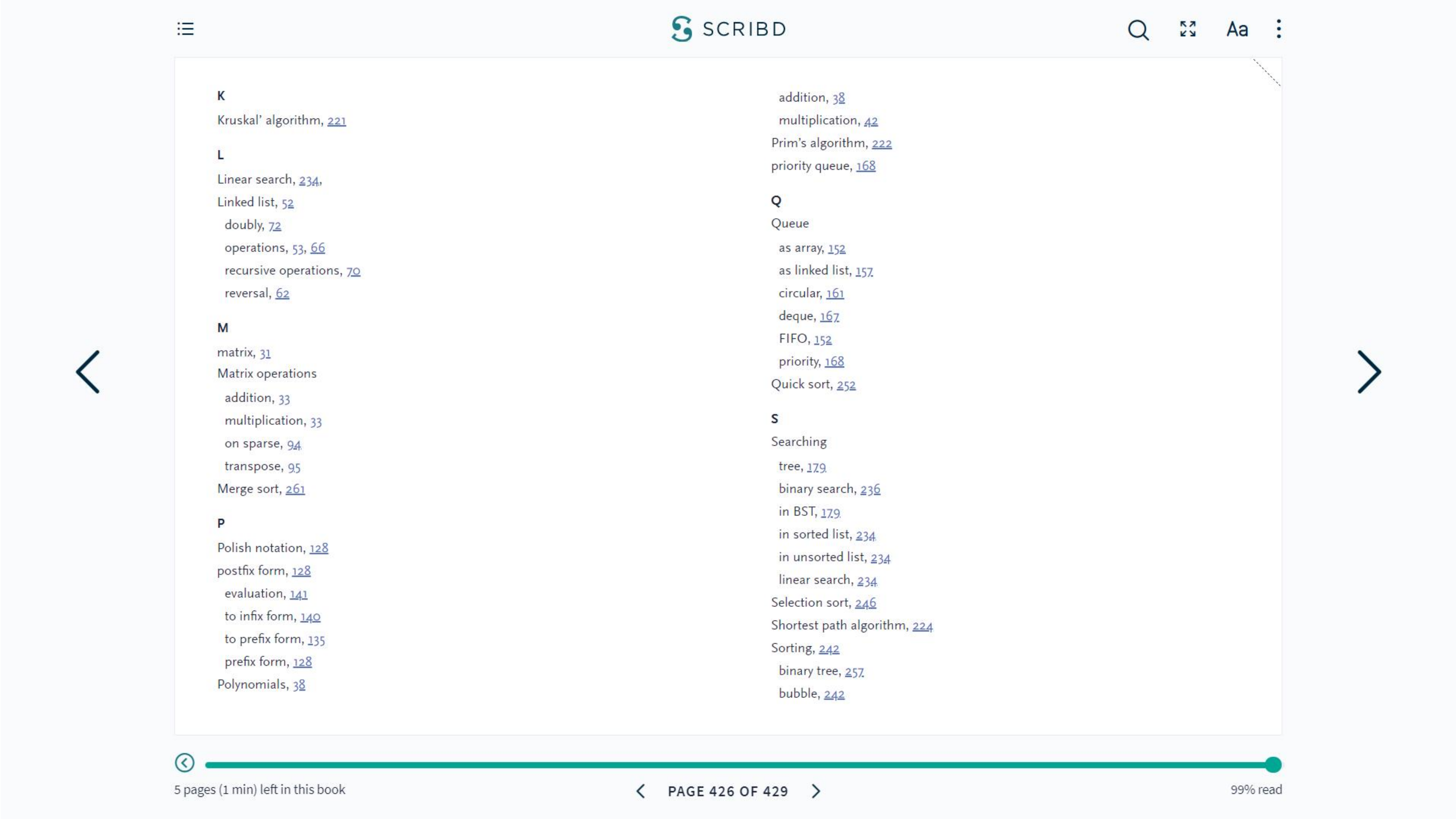
D
Data structures
  linear, 28
  non-linear, 172, 208
Depth first search, 214
  deque, 167
  input-restricted, 167
  output-restricted, 168
Dijkstra's algorithm, 224
Doubly linked list, 72

E
External sorting, 242
  edge, 208

G
Graph
  Dijkstra's algorithm, 224
  Kruskal's algorithm, 221
  Prim's algorithm, 222
  adjacency list, 210
  adjacency matrix, 209
  breadth first search, 216
  depth first search, 214
  digraph, 208
  directed, 208
  edge, 208
  shortest path algorithm, 224
  spanning tree, 219
  undirected, 208
  vertex, 208

H
Heap, 199
  construction, 200
  max-heap, 199
  min-heap, 199
Heap sort, 265

I
infix form, 128
  to postfix form, 128
  to prefix form, 140
Insertion sort, 249
Internal sorting, 242



K

Kruskal' algorithm, 221

L

Linear search, 234,
Linked list, 52
  doubly, 72
  operations, 53, 66
  recursive operations, 70
  reversal, 62

M

matrix, 31
Matrix operations
  addition, 33
  multiplication, 33
  on sparse, 94
  transpose, 95
Merge sort, 261

P

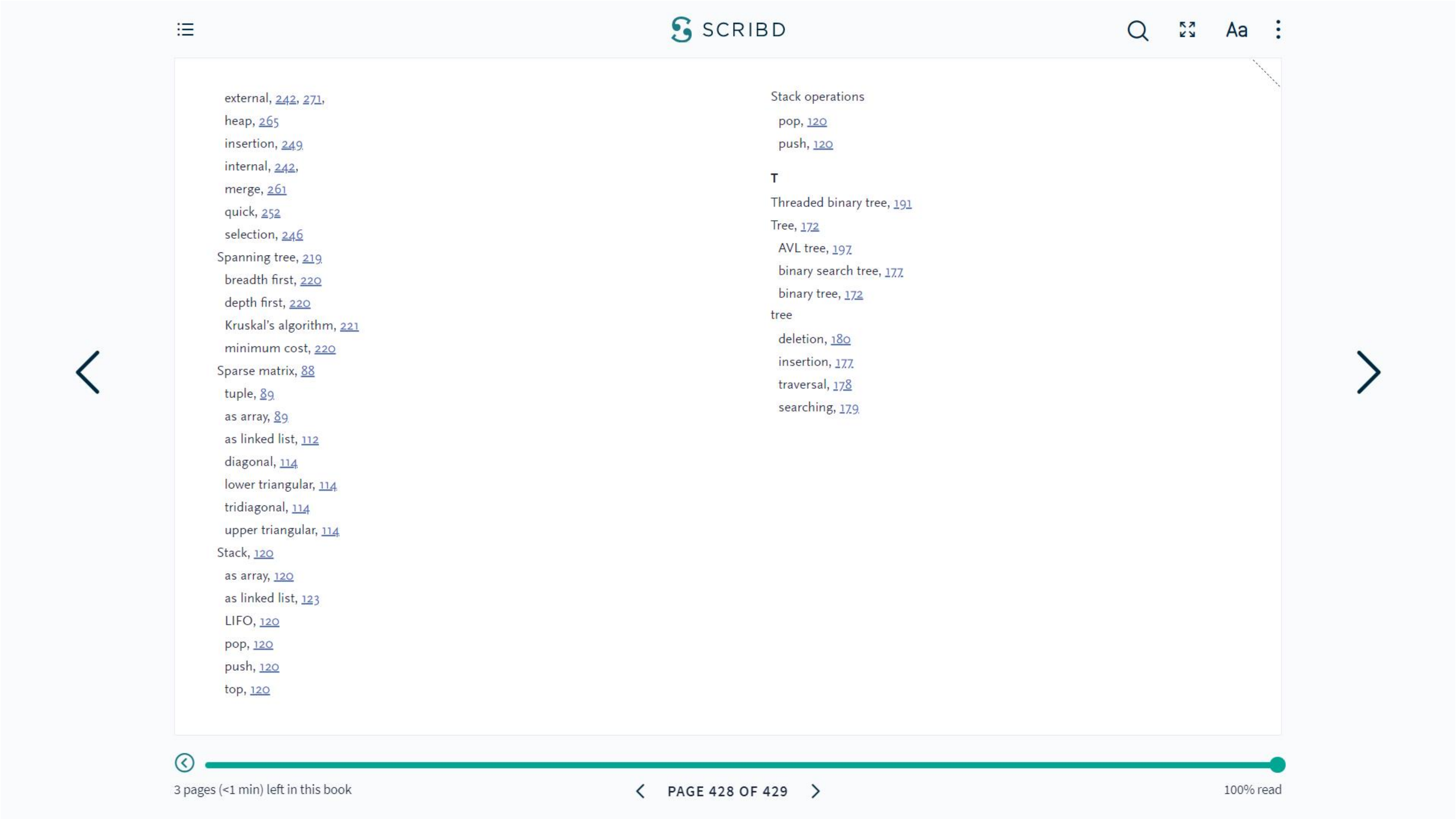
Polish notation, 128
postfix form, 128
  evaluation, 141
  to infix form, 140
  to prefix form, 135
  prefix form, 128
Polynomials, 38
  addition, 38
  multiplication, 42
Prim's algorithm, 222
priority queue, 168

Q

Queue
  as array, 152
  as linked list, 157
  circular, 161
  deque, 167
  FIFO, 152
  priority, 168
Quick sort, 252

S

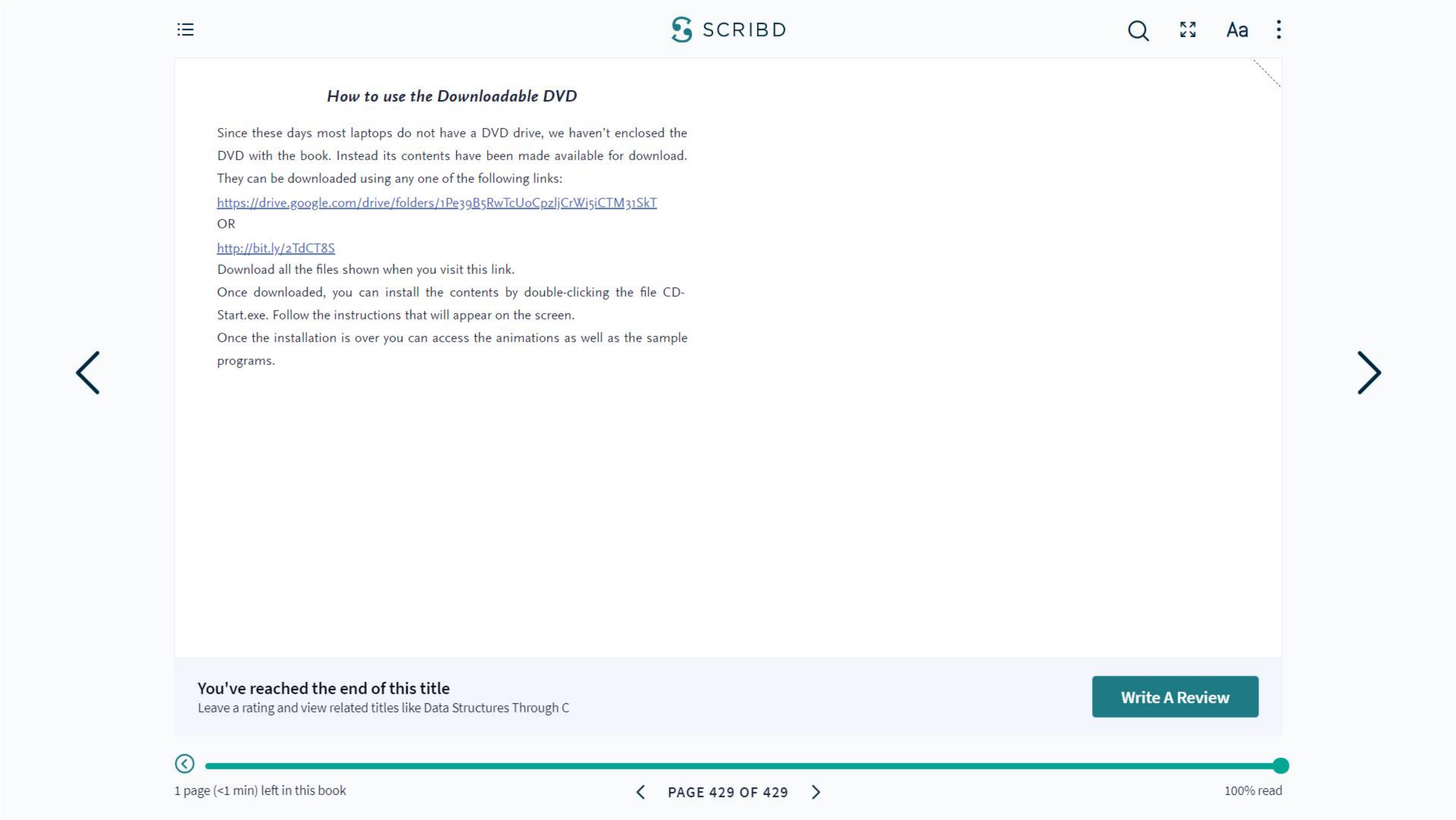
Searching
  tree, 179
  binary search, 236
  in BST, 179
  in sorted list, 234
  in unsorted list, 234
  linear search, 234
Selection sort, 246
Shortest path algorithm, 224
Sorting, 242
  binary tree, 257
  bubble, 242



external, 242, 271,
heap, 265
insertion, 249
internal, 242,
merge, 261
quick, 252
selection, 246
Spanning tree, 219
breadth first, 220
depth first, 220
Kruskal's algorithm, 221
minimum cost, 220
Sparse matrix, 88
tuple, 89
as array, 89
as linked list, 112
diagonal, 114
lower triangular, 114
tridiagonal, 114
upper triangular, 114
Stack, 120
as array, 120
as linked list, 123
LIFO, 120
pop, 120
push, 120
top, 120
Stack operations
pop, 120
push, 120

T

Threaded binary tree, 191
Tree, 172
AVL tree, 197
binary search tree, 177
binary tree, 172
tree
deletion, 180
insertion, 177
traversal, 178
searching, 179



How to use the Downloadable DVD

Since these days most laptops do not have a DVD drive, we haven't enclosed the DVD with the book. Instead its contents have been made available for download. They can be downloaded using any one of the following links:

https://drive.google.com/drive/folders/1Pe39B5RwTcUoCpzljCrWi5jCTM31SkT
OR

http://bit.ly/2TdCT8S
Download all the files shown when you visit this link.
Once downloaded, you can install the contents by double-clicking the file CD-Start.exe. Follow the instructions that will appear on the screen.
Once the installation is over you can access the animations as well as the sample programs.

You've reached the end of this title
Leave a rating and view related titles like Data Structures Through C