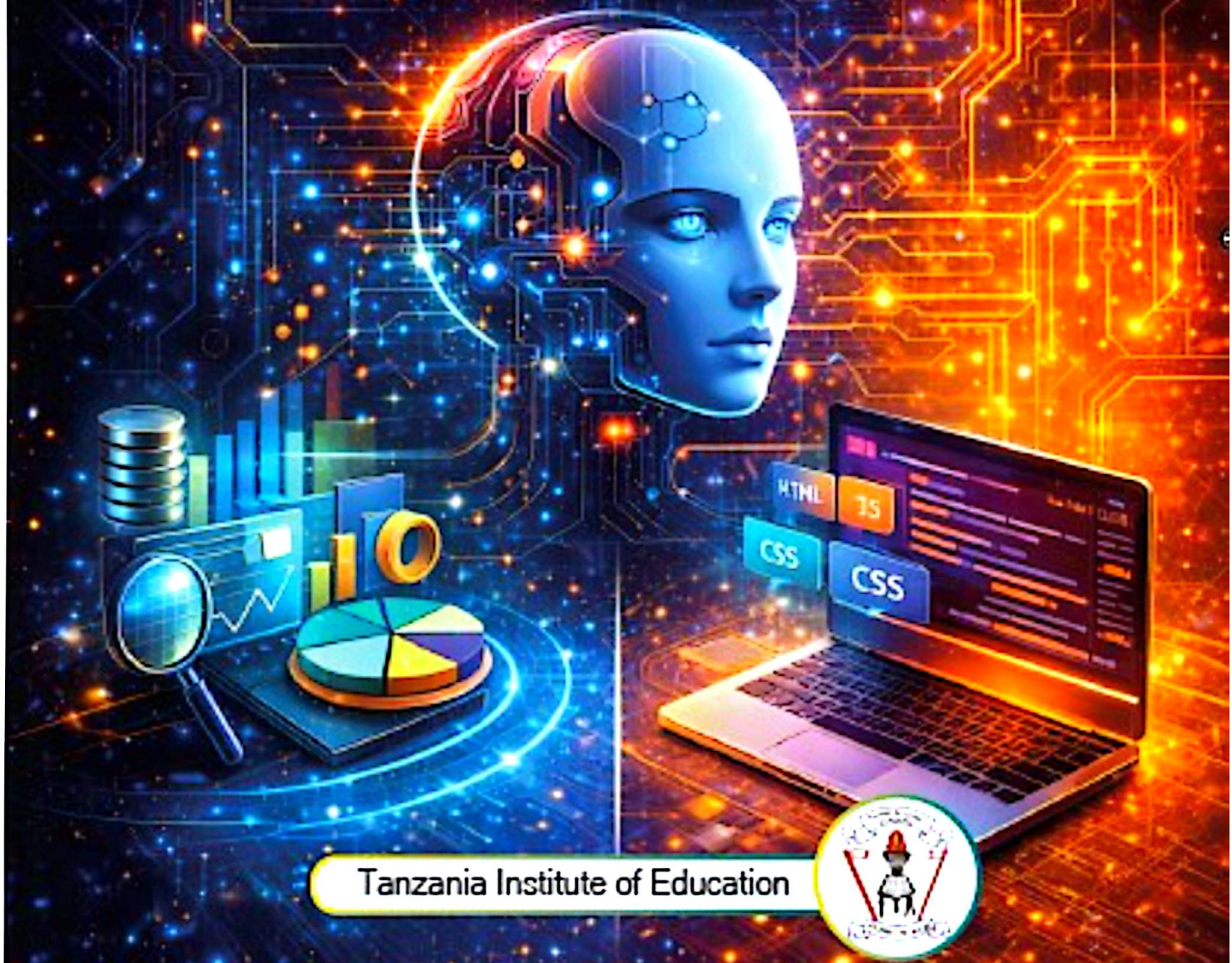
Computer Science

for Advanced Secondary Schools

Student's Book

Form Six



Tanzania Institute of Education

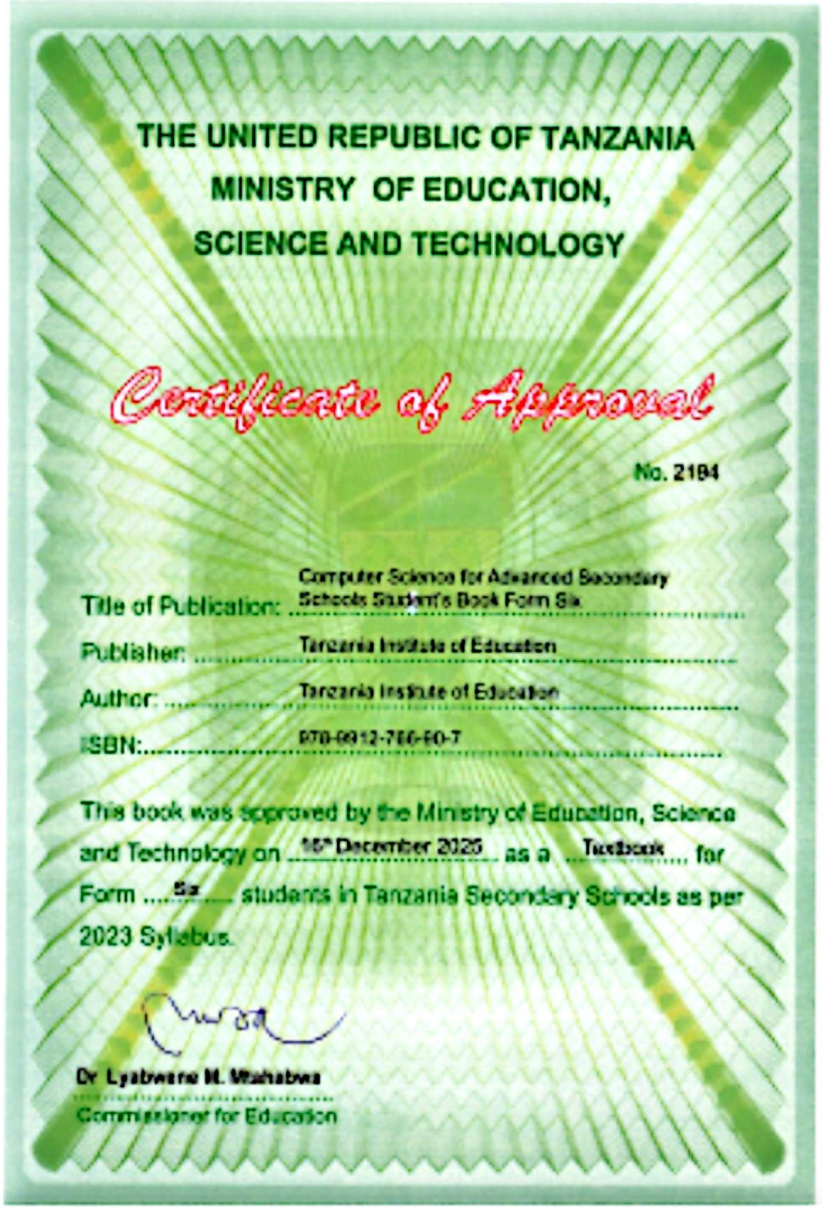
CamScanner

Computer Science

for Advanced Secondary Schools

Student's Book

Form Six



THE UNITED REPUBLIC OF TANZANIA
MINISTRY OF EDUCATION,
SCIENCE AND TECHNOLOGY

Certificate of Approval

No. 2184

Title of Publication: Computer Science for Advanced Secondary Schools Student's Book Form Six

Publisher: Tanzania Institute of Education

Author: Tanzania Institute of Education

ISBN: 978-9912-766-60-7

This book was approved by the Ministry of Education, Science and Technology on 18th December 2023 as a Textbook for Form Six students in Tanzania Secondary Schools as per 2023 Syllabus.

Dr Lyabwene M. Mtahabwa
Commissioner for Education

Tanzania Institute of Education

CamScanner

© Tanzania Institute of Education, 2025

Published 2025

ISBN: 978-9912-766-90-7

Tanzania Institute of Education
Mikocheni area
132 Ali Hassan Mwinyi Road
P.O. Box 35094
14112 Dar es Salaam

Mobile numbers: +255 735 041 170 /+255 735 041 168
E-mail: director.general@tie.go.tz
Website: www.tie.go.tz

FOR ONLINE READING ONLY

All rights reserved. No part of this textbook may be reproduced, stored in any retrieval system, or transmitted in any form or by any means, whether electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of the Tanzania Institute of Education.

CamScanner

Acronyms and abbreviations iv
Acknowledgements v
Preface vi
Chapter One: Data structures and algorithms 1
The concept of data structure 1
Classification of data structures 6
Algorithms 43
Revision Exercise 1 67
Chapter Two: Web application development 71
The Concept of web applications 71
Creating interactive web pages 95
Developing interactive web pages 113
Applying web apis in rich web-based applications 118
Creating data-driven web applications 127
Developing rich-based web applications with desktop-like features 141
Using CSS and modern HTML controls 143
Back-end development 145
Revision Exercise 2 151
Chapter three: Automated systems 152
Understanding systems 152
Concepts of automated systems 162
The core components of an automated system 175
Creating simple automated systems 197
Revision Exercise 3 204
Chapter Four: Foundations of data science 207
Concept of emerging technologies 207
Artificial intelligence and machine learning 217
3D imaging and holography 223
Virtual Reality (VR) and Augmented Reality (AR) 228
Revision Exercise 4 253
Chapter Five: Emerging technologies 255
Overview of data science 255
Types and sources of data 261
Data collection and preparation 333
Exploratory Data Analysis (EDA) 341
Realworld Applications of holography 343
Revision Exercise 5 354
Project work 357
Glossary 358
Bibliography 361
Apendices 365
Indices 367

FOR ONLINE READING ONLY

CamScanner

Acronyms and abbreviations

AI	Artificial Intelligence
ADT	Abstract Data Type
API	Application Programming Interface
AR	Augmented Reality
CSV	Comma-Separated Values
CSS	Cascading Style Sheets
DL	Deep Learning
DS	Data Science
DSA	Data Structures and Algorithms
GraphQL API	Graph Query Language Application Programming Interface
IDE	Integrated Development Environment
IoT	Internet of Things
JSON	JavaScript Object Notation
JS	JavaScript
ML	Machine Learning
NLP	Natural Language Processing
PWA	Progressive Web Application
REST API	Representational State Transfer Application Programming Interface
SOAP API	Simple Object Access Protocol Application Programming Interface
SQL	Structured Query Language
UI	User Interface
VR	Virtual Reality

FOR ONLINE READING ONL

CamScanner

Acknowledgments

The Tanzania Institute of Education (TIE) recognises and values the contributions of all organisations and individuals who participated in the design and development of this textbook. In particular, TIE thanks the University of Dar es Salaam (UDSM), Mzumbe University (MU), Institute of Finance Management (IFM), Projekt Inspire Tanzania for their invaluable support throughout the design and development of this textbook. Besides, the following individuals are highly acknowledged.

Writers: Mr Watson Gibson Mtewele (TIE), Mr Nchahoruri Francis Joel (TIE), and Dr Mahadia Tunga (UDSM)

Editors: Dr Hussein Bakiri(IFM), Dr David T. Makota (IFM), and Mr Joseph Alex (Projekt Inspire)

Designer: Ms Rehema A. Hamisi

Coordinator: Mr Watson Gibson Mtewele (TIE)

TIE also would like to thank the Government of the United Republic of Tanzania for financing the writing and printing of this textbook.



Dr Aneth A. Komba
Director General
Tanzania Institute of Education

FOR ONLINE READING ONLY

COMPUTER SCIENCE F6 FINAL (EVA Feb 3.indd 5 03/02/2026 15:48:44
CamScanner

Preface

This textbook, Computer Science for Advanced Secondary Schools Student's Book, Form six has been written specifically for Form six students in the United Republic of Tanzania.

The book is prepared in accordance with the 2023 Computer Science Syllabus for Advanced Secondary Education, Form V-VI, issued by the Ministry of Education, Science and Technology.

The book consists of five chapters, which are: Data structures and algorithms, Web application development, Automated systems, Foundations of data science and Emerging technologies.

You are encouraged to do all activities and exercises in this book, and other assignments provided by your teacher. You are also required to prepare a portfolio for keeping records of activities performed in different lessons. This will enhance your understanding and development of the intended competencies for this level.

In addition to your studies at this level, you will be required to design and carry out a project in Computer Science and then submit its final report for assessment. For more details refer to the project guide.

Additional learning resources are available in the TIE e-learning at https://ol.tie.go.tz or ol.tie.go.tz

Tanzania Institute of Education

FOR ONLINE READING ONLY

CamScanner



Chapter One

Data structures and algorithms

Introduction

Data structures and algorithms form the backbone of programming and system design, ensuring that applications are not only correct but also scalable, reliable, and optimised for speed and memory usage. In this chapter, you will learn about the concept of data structures and algorithm, how data are structured and instructions sequenced in algorithms, programs, and the relationship between appropriate data and control structures. The competencies developed through data structures and algorithms will help you to write efficient programs to retrieve and store massive data of various types.

FOR ONLINE READING ONLY

Think

Using data structures and algorithms in real-life setting

The concept of data structure

Scenario: ***Using a computer with improper organisation***

Imagine you want to use a navigation app, such as Google Maps, to find directions to a place you have never visited before. The app needs to remember many things, such as roads, places, and how those roads are connected.

Now imagine if the computer "did not organise this information properly". All the road names, places, and distances would be mixed together without any clear order. The computer would struggle to know which road leads where, which place is closer, or which route is faster.

Because of this lack of organisation:

(a) The app would take a very long time to load.

(b) The directions given might be confusing or incorrect.

(c) Sometimes the app might not work at all, especially when handling a lot of information at the same time.

COMPUTER SCIENCE F6 FINAL (EVA Feb 3.indd 1 03/02/2026 15:48:45

CamScanner

This shows that computers, just like people, need "clear ways to organise information" so that tasks can be done quickly and correctly.

Questions

1. Identify three problems a navigation system might face if information is not well organised.
2. Explain why organising roads and places is essential for giving correct directions in a navigation app.
3. Describe how the proper organisation of information helps computers work faster and give accurate results.

Meaning of data structure

Data structures using real-life examples

You always organise items in your daily life, even if you do not realise it. In real life, people store and organise things in different ways, as shown in Figures 1.1 to 1.7:

(a) Books are arranged on a shelf

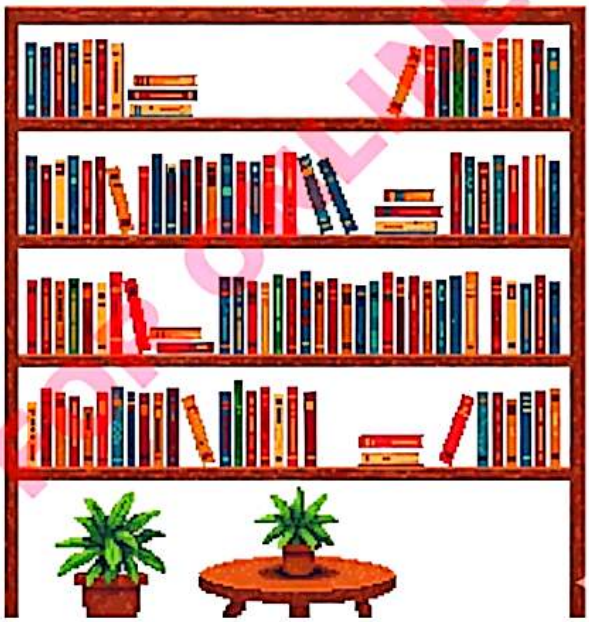


Figure 1.1: *Books arranged on a shelf*

(b) Clothes are kept in a cupboard or wardrobe



Figure 1.2: *Clothes kept in a cupboard*

(c) Phone numbers are saved as contacts in a mobile phone



Figure 1.3: *Phone numbers saved in a mobile phone*

FOR ONLINE READING ONLY

CamScanner

(d) Paper documents are filed in a cabinet



Figure 1.4: *Paper documents in thecabinet*

Each of these methods helps people find, use, and manage items easily. In the same way, computers need methods to store and organise data so that it can be accessed and used efficiently.

A data structure is simply a way of "organising" data in a computer so that it can be used easily.

Everyday organisation patterns and data structures

In daily life, people naturally follow "organised patterns" when arranging or managing things. These patterns help maintain order and make activities run smoothly. Examples of everyday organisation include:

(a) A queue at a bank, where the first person to arrive is the first to be served



Figure 1.5: *A queue at a bank (First-come, First-served)*

(b) A stack of plates, where the last plate placed on top is the first one taken



Figure 1.6: *A stack of plates* (*Last-in, First-out)*

(c) A family hierarchy, which shows relationships from grandparent to parent to child



Figure 1.7: *A family hierachy*

FOR ONLINE READING ONLY

CamScanner

(d) Road connections between towns, which show how different places are linked

These examples demonstrate that structure already exists in everyday life. People use different structures depending on the situation and the goal. Similarly, computers must organise information in clear logical ways to store, process, and retrieve data efficiently.

These organised ways of storing and managing data in a computer are known as "data structures".

A data structure is a method of storing and organising data in a computer so that it can be used efficiently. This means that data is not stored randomly; instead, it is organised in a planned, meaningful way that enables easyaccess, updating, and management.

This concept can be better understood by breaking it into parts. Storing data refers to keeping information in a computer's memory or on storage devices. Organising data means arranging that information in a particular order or structure. Using data efficiently means retrieving it quickly, making changes easily, and managing it without unnecessary delay or complexity.

A simple example of a data structure is a "linear arrangement", similar to books placed in a row on a shelf. This can be represented as:

10→ 20 → 30 → 40

In this arrangement, each item follows the other in a straight line. Such an organisation helps to understand linear data structures, which later lead to concepts such as arrays and lists.

A more advanced example is a "non-linear arrangement", such as a family structure. In a family, one parent may be connected to more than one child, creating a branching pattern. Here, one item can connect to many items rather than following a single straight line.

A data structure is a means for organising, retrieving, processing, and storing data in a computer. A well-designed data structure enables critical operations to be performed with minimal resource consumption, particularly execution time and memory.

The way data is organised affects a program's performance across different tasks.Computer programmers decide which data structures to use based on the nature of the data and the processes to be performed. The data structure organises data items according to the relationship among data elements in a computer program. For example, attributes such as the model, name, car number, engine capacity transmission type, and colour can identify a car. These variables depend on each other to locate a car.

Generally, a data structure links the programmer's view of data with its implementation in computers. It comprises the storage methods and one or more algorithms used to process data.

FOR ONLINE READING ONLY

CamScanner

Importance of data structures

Data structures are essential in programming for producing efficient programs. The following are the importance of data structures:

(i) They make it easy for programmers to access and work with data.

(ii) They offer the means to manage large quantities of data.

(iii) They enable fast searching and sorting of data.

Goals of data structure

During data structure implementation, the following key goals are pursued:

(a) **Efficient data retrieval and processing**

Data structures are designed to organise and store data in ways that enable fast easy access, modification, and processing to solve specific problems within a program. Proper organisation improves the speed and accuracy with which data can be retrieved and processed.

(b) Optimal use of resources

Data structures aim to process data at required speeds while using minimal computer resources. This includes optimising both execution time (speed) and memory usage, leading to effient overall software performance.

Features of data structures

Some of the essential features of data structures are described as follows:

Reusability: A programmer requires many resources to develop any software. However, if the codes and resources used to develop the software are reusable, it can save time and resources of developing a software. Thus, by implementing quality data structures, it is possible to reuse software codes and resources which tends to be cost-effective and time-saving.

Robustness: Computer programs are designed to generates correct output for every possible input provided to them and executes efficiently on all hardware platforms. A robust data structure handles error, invalid inputs or unusual operations without crashing, losing data or producing wrong results.

Adaptability: Developing software such as web browsers, word processors, and Learning Management Systems (LMS) involve bulky software code that has worked or been accomplished appropriately and efficiently for several years. A data structure is adaptable when it works efficiently in different situations and can reorganise itself when conditions change.

Factors to consider when selecting a data structure

It is essential to choose an appropriate data structure for each task. The factors to consider when selecting the proper data structure include the following:

(a) Types of data to be stored

The nature of the data to be stored plays a key role in selecting a data structure. Specific structures are suitable for uniform and straightforward data, such as arrays, while others are better suited for managing complex or hierarchical data, such as trees and graphs.

FOR ONLINE READING ONLY

CamScanner

(b) Operations to be performed on the data

When selecting a data structure, it is important to consider how frequent various operations; such as searching or inserting data will occur. Choosing a structure with appropriate time complexity for these operations improves performance. For instance, hash tables provide fast average access times, whereas linked lists allow efficient insertion and deletion operations.

(c) Nature of the problem and requirements

Selecting an appropriate data structure begins with a clear understanding of the problem to be solved. This includes identifying the required operations, such as searching, inserting, or deleting data and any limitations involved, ensuring that the chosen structure meets the program's objectives.

(d) Space complexity (memory usage)

Data structures differ in their memory requirements. Efficient memory use is essential, particularly in systems with limited resources. For instance, arrays store data in contiguous memory locations, whereas linked lists require additional memory to store pointers.

Classification of data structures

Data structures can be classified based on various aspects, for instance, linear and non-linear data structures based on the arrangement or organisation of their data; or static and dynamic data structures based on whether the size of the data structure changes during program execution or remains fixed. As depicted in Figure 1.8, data structures are classified based on the logical memory layout of their data.

FOR ONLINE READING ONLY

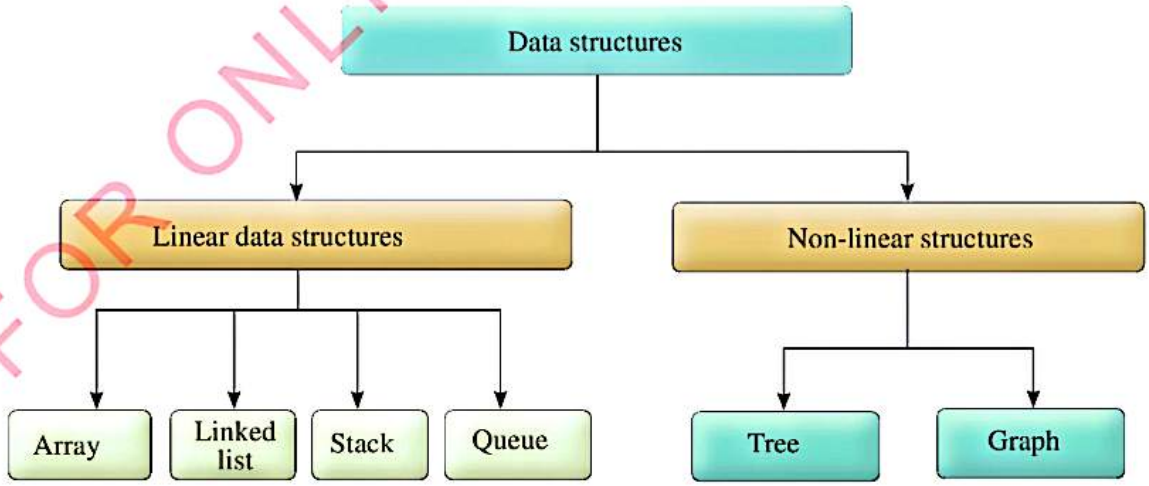


Figure 1.8: *Classification of data structures*

CamScanner

Linear data structures

Linear data structures store data in a sequence, one item after another, and access them sequentially from one location. There are two techniques for implementing linear data structures. The first technique uses arrays, and the second uses pointers or links. Examples of linear data structures are arrays, stacks, linked lists, and queues.

Non-linear data structures

In a non-linear data structure, an item may link to more than one element, this means items are not sequential. This structure primarily represents data with a hierarchical relationship among items. Examples of non-linear data structures are trees and graphs.

Activity 1.1:

Understanding and classifying data structure

Aim: To explore the fundamental concepts of data structures, including their goals and features, the factors that influence their selection, and classification of different data structures based on their characteristics and real-world applications.

Materials: Digital device with Internet access for online search (Options)

Instructions

Step 1: Identify one real-life example where data must be organised efficiently and explain why the choice between a linear or non-linear data structure is important.

Examples: school database, hospital queue, music playlist, road network.

Step 2: Create a classification table of data structures with the categories (linear data structures and non-linear data structures. List at least two examples under each category.

Step 3: Select one linear data structure and one non-linear data structure from your table, then:

(a) Explain how each data structure works.

(b) Describe one real-world application for each.

(c) Explain one possible problem that could occur if an inappropriate data structure were used instead.

Step 4: Write a short paragraph explaining why selecting the appropriate data structure is important for efficiency, performance, and reliability in computing systems.

Deliverables

A short report containing:

(a) One real-life example demonstrating the importance of choosing between linear and non-linear data structures

(b) A completed classification table showing at least two examples under each category

FOR ONLINE READING ONLY

CamScanner

(c) A paragraph explaining the operation, application, and potential problem for one selected linear and one selected non-linear data structure

(d) A reflection paragraph on why correct data structure selection improves system efficiency and performance

Exercise 1.1

1. Kazola Company wants to decide on a data structure to implement the program. How would you assist Kazola Company in selecting the most suitable data structure for their program based on the nature of their data and the operations the system must perform?
2. A school stores student information such as names, subjects, grades, attendance, and disciplinary records. Analyse why primitive data types and simple linear data structures (such as arrays) are insufficient for organising and managing such complex and interrelated data efficiently.
3. Large databases require fast searching, insertion, and deletion of records. Analyse how different non-linear data structures (such as trees and graphs) influence the speed of data retrieval compared to linear data structures like lists and queues. Use relevant examples to support your answer.
4. A navigation system represents cities as points and roads as connections between them. Evaluate why a non-linear data structure is more suitable than a linear data structure for representing this system, and identify the most appropriate data structure for a road network. Justify your choice.
5. A school management system needs to store basic student details, subject enrolments, and prerequisite relationships between courses. Explain, with examples, how primitive data types, linear data structures, and non-linear data structures can be used in a school management system to store student details, subject enrolments, and course prerequisites.
6. Compare linear and non-linear data structures in terms of data organisation, memory usage, and typical applications. Provide one real-world example where a linear data structure is most effective and one where a non-linear data structure is most suitable.

Static data structure



Scenario 1.2: *Packaging and capacity in everyday items*

You walk into a shop and notice several crates of soda arranged neatly on the floor. Each crate has a fixed number of spaces; some hold 12 bottles, others 18 or 24. When you look closely, you see that the bottles fit tightly into their slots, leaving no extra space.

Nearby, you see other items packaged in specific containers: eggs arranged in trays

FOR ONLINE READING ONLY

CamScanner

of 30, juice boxes packed in cartons, biscuits arranged in dozens, and fruit placed neatly in trays. Each container seems to have a clear limit on how many items it can hold.

Questions

1. Based on the scenario, do you think it is possible to add more items to a container than its designed capacity? Explain your answer.
2. What problems might occur if someone forces extra items into a crate, tray, carton, or any container meant for a fixed number?
3. Why do you think manufacturers design containers with specific capacities and fixed arrangements?

Meaning of static data structure

A static data structure is a type of data structure in which the size and memory allocation are fixed at the time of creation and cannot be changed during program execution. This means the number of elements it can hold is predetermined, and it cannot grow or shrink dynamically. Examples include arrays, of which the size is set when declared, and fixed-size records/structures. Key features of a static data structure are that memory is allocated at compile time, it is easier to implement and access, and it can waste memory if not fully used or may run out of space if more data is needed. Static data structures offers benefits and limitations that a programmer should consider when deciding to use them.

Advantages of a static data structure

The following are the advantages of a static data structure:

(a) Ease of implementation: Static data structures are simpler to define, implement, and use compared to dynamic data structures because their size and memory requirements are known in advance.

(b) Fixed size: A static data structure has a predetermined size that is specified at compile time and cannot be changed during program execution.

(c) Faster access and execution: Memory for static data structures is allocated at compile time, eliminating the overhead of memory allocation during program execution. This generally results in faster program performance. In addition, elements in structures such as arrays can be accessed directly using an index, allowing efficient random access.

Disadvantages of a static data structure

The following are the disadvantages of a static data structure:

(a) Memory wastage: Static data structures have a fixed size. If more memory is allocated than required, the unused portion cannot be reused, resulting in wasted memory space.

(b) Fixed capacity limitation: Because the size of a static data structure

FOR ONLINE READING ONLY

COMPUTER SCIENCE F6 FINAL (EVA Feb 3.indd 9 03/02/2026 15:48:50
CamScanner

cannot change during execution, it may become insufficient if the program needs to store more data than initially allocated. This can limit the program's functionality or cause errors.

(c) **Limited flexibility and efficiency:** Static data structures are less flexible than dynamic ones. Operations such as resizing or merging often require creating a new structure and copying existing elements, which increases execution time and reduces efficiency, especially for large data sets.

Types of static data structure

As seen earlier, a static data structure is a method of storing data in which the amount of data stored and the memory used to hold it are fixed. An example of a static data structure is an array.

Arrays

An array is a data structure used to store a collection of elements of the same data type in contiguous memory locations. Each element is identified by an index, which allows fast and direct access. Arrays act as containers with a fixed size, meaning the number of elements must be defined in advance and cannot change during program execution.

Arrays can be one-dimensional, where data is stored as a list, or multidimensional, such as two-dimensional arrays, which are commonly represented as tables or matrices with rows and columns. Most programming languages use arrays as a foundation for implementing other data structures and algorithms.

Example: Consider an array of students sitting in a row, each student has a seat number (index). All students belong to the same class (same data type)

Declaring an array in C++

Example of array declaration is int marks[5] = {60, 70, 80, 75, 90}; , where representation is like that marks→array name, 5→number of elements, 60 → marks[0], 70 → marks[1].

Array indexing in C++ starts from 0, not 1. This means the first element is always accessed using index.

Basic array operations

This includes insertion, deletion, searching, sorting, and merging. Arrays are widely used because they are simple to use and provide fast element access to elements. However, they also have limitations, such as fixed size, difficulty in resizing, and inefficiency when inserting or deleting elements due to shifting.

(i) *Insertion in an array*

Insertion means adding a new element into an array at a specific position. Since arrays have fixed size, we usually shift elements to create space.

Example scenario: A new student joins a class and must be placed between existing students.

FOR ONLINE READING ONLY

CamScanner

Example code

```
#include <iostream>
using namespace std;
int main() {
    int arr[6] = {10, 20, 30, 40, 50};
    int n = 5;      // current size
    int pos = 2;    // insert at index 2
    int value = 25;
    // Shift elements to the right
    for (int i = n; i > pos; i--) {
        arr[i] = arr[i - 1];
    }
    arr[pos] = value;
    n++;
    // Display array
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    return 0;
}
```

Output

```
10 20 25 30 40 50
```

(ii) *Deletion in an array*

Deletion means removing an element from an array. After deletion, elements are shifted left to fill the gap.

Example Scenario: A student transfers to another school and leaves the class. The following is an example code:

```
#include <iostream>
using namespace std;
int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int n = 5;
    int pos = 2; // delete element at index 2
    // Shift elements left
    for (int i = pos; i < n - 1; i++) {
        arr[i] = arr[i + 1];
    }
    n--;
    // Display array
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    return 0;
}
```

Output

```
10 20 40 50
```

FOR ONLINE READING ONLY

CamScanner

Simulating "insertion" in array using VisualGo program

Activity 1.2:

Visualizing array insertion using VisualGo

Aim: To visualize how arrays work, specifically how insertion operates in an array, and to develop problem-solving and analytical skills by observing how data shifts when a new element is inserted.

Materials: Computer, laptop, tablet, or smartphone; Internet connection; and Web browser.

Instructions

Step 1: Access the visual tool

(a) Open a web browser.

(b) Go to https://visualgo.net.

(c) Choose Array and click to view

Step 2: Create the initial array

(a) Locate the option to ***create or input array values***.

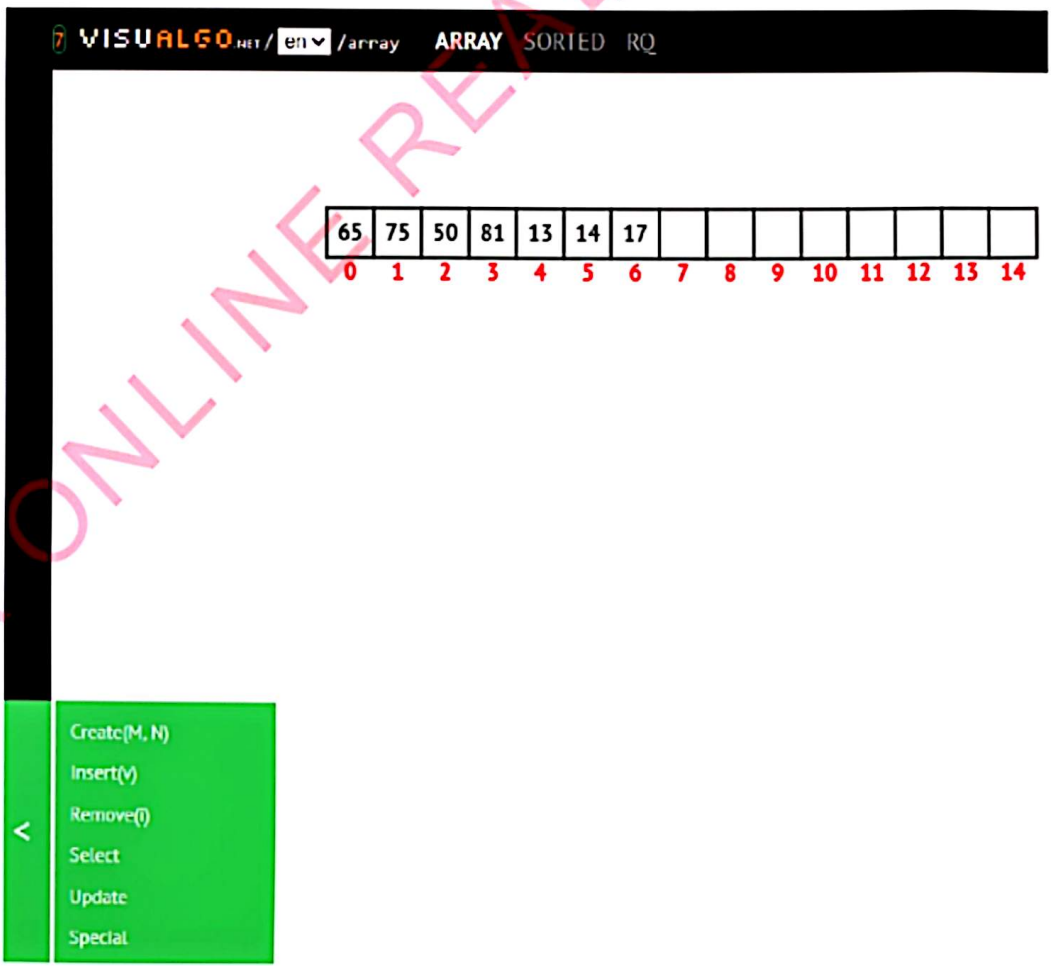


Figure 1.9: *Creating initial array values*

(b) Enter the following array elements in order of 10, 20, 30, 40, 50 as in Figure 1.10.

FOR ONLINE READING ONLY

CamScanner

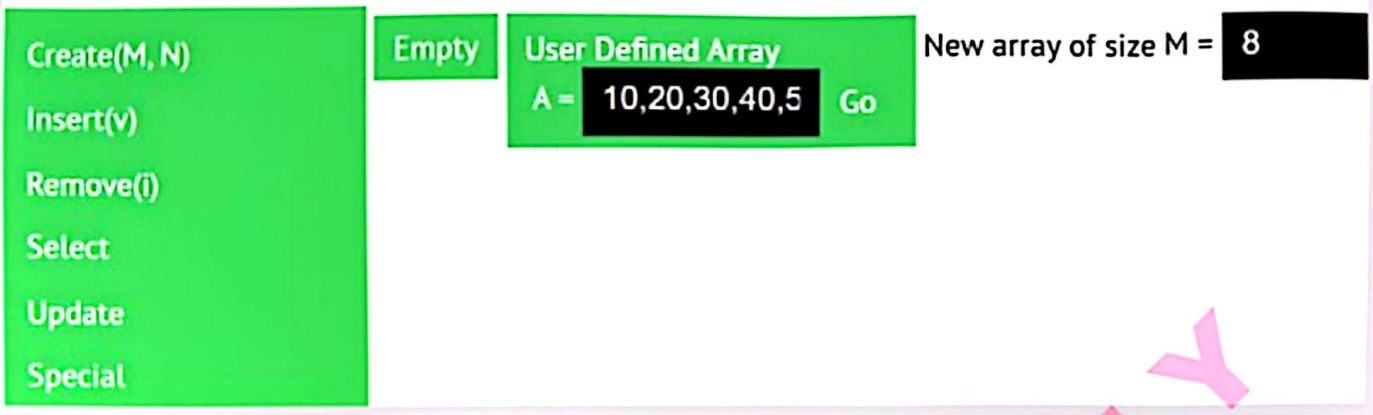


Figure 1.10: *Entering array values*

(c) Confirm or initialize the array.

(d) Observe how the array is displayed visually, with each value stored in a fixed position (index).

Step 3: Perform an insertion operation

(a) Select the *Insert* operation.

(b) Choose a position (for example, insert 25 at index 2).

(c) Start or play the animation.

(d) Carefully observe:

(i) Which elements move (shift) to the right.

(ii) How space is created for the new value.

(iii) The final order of elements in the array is as shown in Figure 1.11.

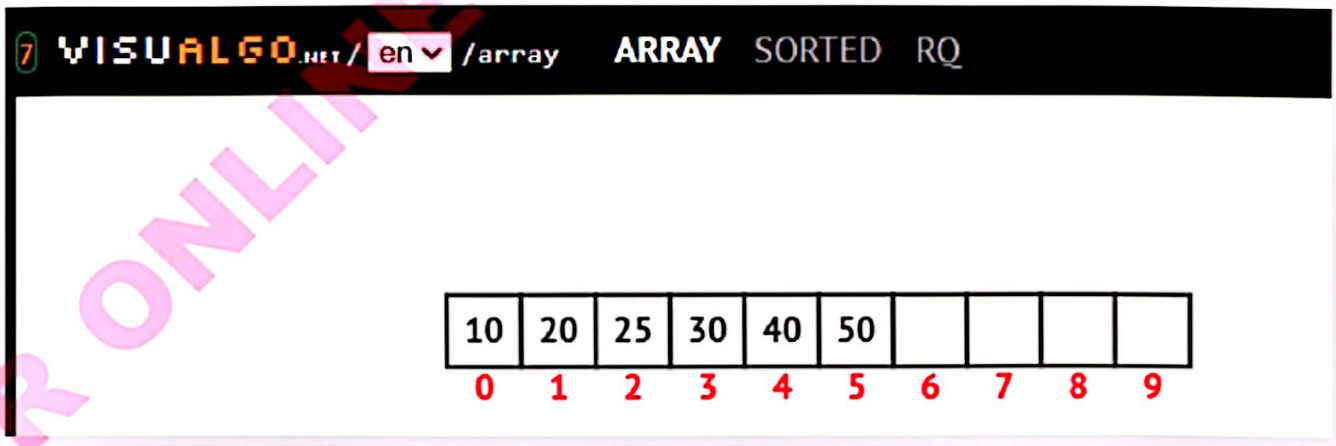


Figure 1.11: *Final order of elements*

Step 4: Analyse the process

(a) Replay the animation if necessary.

(b) Identify how many elements were shifted.

(c) Discuss what would happen if the array were already full.

(d) Relate the process to real-life examples (such as inserting a student into a class list).

Step 5: Repeat and experiment

(a) Try inserting a value at:

FOR ONLINE READING ONLY

CamScanner

(iii) The beginning of the array.
(iv) The middle of the array.
(v) The end of the array.
(b) Compare how the number of shifts changes in each case.

Deliverables

(a) Screenshot of initial array representation before insertion:
(b) Screenshot of final array after insertion
(c) A short explanation (about 3 to 5 sentences) describing:
(i) What happens to existing elements during insertion.
(ii) Why array insertion can be time-consuming.

(iii) *Merging arrays*

Merging means combining two arrays into one larger array. The example Scenario in this may be combining marks from Class A and Class B.

Example code

```
#include <iostream>
using namespace std;
int main() {
    int arr1[3] = {10, 20, 30};
    int arr2[3] = {40, 50, 60};
    int arr3[6];
    for (int i = 0; i < 3; i++)
        arr3[i] = arr1[i];
    for (int i = 0; i < 3; i++)
        arr3[i + 3] = arr2[i];
    for (int i = 0; i < 6; i++)
        cout << arr3[i] << " ";

    return 0;
}
```

Output

```
10 20 30 40 50 60
```

Pointers

A pointer is a variable whose value is the address of another variable, that is, the direct address of the memory location. It is a user-defined data type that creates variables to hold the memory address of other variables. It is denoted by the "*" operator. It represents storage space in memory (RAM) partitioned into a small units called cells, which store values. Each cell has a unique address and occupies one byte in memory, whose address is always an unassigned integer. If one variable carries the address of another variable, the first variable is said to be the pointer of another. Thus, a pointer is a variable whose value is also an address where each variable has two attributes: address and value, as shown in Figure 1.12.

FOR ONLINE READING ONLY

CamScanner



Figure 1.12: *Structure of a pointer*

The syntax of a pointer is *type *ptr_name;* . The type is any data type of the pointer, and; ptr_name is the pointer's name.

Declaring pointers in C++

The pointer can be declared as follows:

Datatype *variable_name;

(a) The datatype is the base type of the pointer, which has to be a valid C++ data type in the case of C++.

(b) The variable_name should be the name of the pointer variable.

(c) An asterisk (*) used above in the pointer declaration is the same as the asterisk used to perform multiplication operations. It marks the variable as a pointer.

The C++ uses the '&' (reference) operator to return a variable's address. For example, if x is a variable, &x returns the address of the variable. The following example shows the application of a pointer in a C++ program.

Example 1: How to apply pointer in C++

```
#include<iostream>
using namespace std;

int main ()
{
    int y = 12;
    int *b;    // declaration of pointer variable

    b = &y;    // the address of y variable is assigned to a pointer b

    cout << "Value at b = " << b << endl;
    // display address stored in pointer b

    cout << "Value at variable y = " << y << endl;
    // display value stored in variable y

    cout << "Value at *b = " << *b;
    // display value stored in address of pointer b
}
```

The output of the program is shown in Figure 1.13:

```
Value at b = 0x7ffc390cbbc4
Value at variable y = 12
Value at *b = 12
```

Figure 1.13: *Output of a program to demonstrate application of pointers*

FOR ONLINE READING ONLY

CamScanner

```
#include<stdlib.h>
using namespace std;
int main()
{
   int i,*ptr;
   ptr =(int*) malloc(3*sizeof(int));
   if(ptr == NULL)
{
      cout<<"Error! memory not allocated.";
      exit(0);
   }
   *(ptr+0)=10;
   *(ptr+1)=20;
   *(ptr+2)=30;
   cout<<"Elements are:";
   for(i =0; i <3; i++){
      cout<<*(ptr + i);
   }
   free(ptr);
   return 0;
}
```

The output for the program in example 3 is "Elements are:10 20 30" is shown in Figure 1.15:

```
Elements are:102030
```

Figure 1.15: *Output of a program for dynamic memory allocation using pointers*

3. The pointers are used to pass arguments by reference in functions to increase efficiency.

   Example 4, the program code in C++ to pass arguments by reference in functions as follows:

FOR ONLINE READING ONLY

CamScanner

```
#include<iostream>
 using namespace std;
 void swap(int* x, int* y){
   int t=*x;
   *x=*y;
   *y = t;
 }
 int main(){
   int r =3, s=9;
   swap(&r,&s);
   cout<< r<<s;
   return 0;
 }
```

The output is 93 as shown in Figure 1.16:

```
93
```

Figure 1.16: *Output of the program to pass arguments by reference*

4. The pointer can be used to implement data structures like linked lists and trees as described in the next sections.

Pointer operations

In C++, the pointer operators are of two kinds:

(a) Reference operator (&): The reference operator (&) returns the variable's address, and

(b) Dereference operator (*): The dereference operator (*) helps us to get the value that has been stored in a memory address.

For example, if you have a variable given the name num1, stored in the address 0x236 and storing the value 27, then

(i) The reference operator (&) will return 0x236.

(ii) The dereference operator (*) will return 27.

Example 5: A C++ program to demonstrate pointer operators (reference and dereference operators) Try this code in C++ and display the output;

FOR ONLINE READING ONLY

CamScanner

```
#include <iostream>
 using namespace std;
 int main () {
 int  x = 26;
 int  *ip;
 ip = &x;
 cout << "Value of x is : ";
 cout << x << endl;
 cout << "Value of ip is : ";
 cout << ip<< endl;
 cout << "Value of *ip is : ";
 cout << *ip << endl;
 return 0;
 }
```

The output of the program shown in Figure 1.17:

```
Value of x is : 26
Value of ip is : 0x7fff8887d654
Value of *ip is : 26
```

Figure 1.17: *Output of the program to demonstrate reference & dereference operators OR pointer operations*

Pointers in arrays

Arrays and pointers work based on a related idea. There are different things to consider when working with arrays having pointers. The array name denotes the base address of the array. This means that, when assigning an array's address to a pointer, do not use an ampersand (&). For example:

```
p = arr; is correct since arr represents the arrays' address but
p = &arr; is incorrect.
```

You can implicitly convert an array into a pointer. For example:

```
int arr [20];
int * y;
//This is a valid operation:
 y = arr;
```

After the declaration, y and arr will be equivalent and share properties. However, a different address can be assigned to y, but you cannot assign anything to arr. For example, you can traverse an array using pointers. Example of a C++ program to traverse the array given as arr[] = {11, 45, 63, 26, 5, 24}.

FOR ONLINE READING ONLY

CamScanner

```
#include <iostream>
using namespace std;
  int main () {
      int *y;
      int arr[] = { 11, 45, 63, 26, 5, 24 };
      y = arr;

      for (int x = 0; x < 6; x++) {
          cout << *y << endl;
          y++;
      }

    return 0;
}
```

The output of the program as shown in Figure 1.18.

```
11
45
63
26
5
24
```

Figure 1.18: Output of a program to traverse an array using a pointer

Records

Records are composite data types formed by several related items of different data types. This enables a programmer to refer to these items using the same identifier, enabling a structured approach to use related items. A record will contain a fixed number of items. For example, a record for a book may include the title, author, publisher, number of pages, and whether it is literature or fiction.

Records store a collection of related data items, where all items have different data types. For example, you might set up a record called book, which stores the book's title, author name, ISBN, and PublicationDate. Title and author are text, whereas the PublicationDate is set as a Date data type. You can write it as follows:

```
Book = Record
Title, Author As Text * 50
ISBN As Text * 13
PublicationDate As Date.
```

When the program runs, every time data is entered for the book, the user will type in up to 50 characters of text for the title, author, and up to 13 of text for ISBN. A variable

FOR ONLINE READING ONLY

CamScanner

```
#recordname {fieldName1 = value1, fieldName2 = value2, fieldName3 = value3 . . . . fieldNamex = valuex}
```

The following is an example of a program to create a record using the C++ programming language.

```
#include <iostream>
#include <string>

using namespace std;

class Student {
public:
    string student_name;
    int student_id;
};

int main() {
    Student S;
    S.student_name = "Maro Sambayeti";
    S.student_id = 7880;

    // Optionally, output the values to verify
    cout << "Student Name: " << S.student_name << endl;
    cout << "Student ID: " << S.student_id << endl;

    return 0;
}
```

The following is an example of a program to access record values in the C++ programming language:

```
#include <iostream>
#include <string>

using namespace std;

class Student {
public:
    string student_name;
    int student_id;
};

int main() {
    Student S;
    S.student_name = "Maro Sambayeti";
    S.student_id = 7880;

    // Output the student details
    cout << S.student_id << "\n" << S.student_name;

    return 0;
}
```

FOR ONLINE READING ONLY

CamScanner

Output is shown in Figure 1.19.

```
7880
Maro Sambayeti
```

Figure 1.19: *Output of a program to access record values.*

Note: Starting the number with zero (0), such as 07880 for student_id will make it octal; therefore, the program will display an error.

Activity 1.3:

Practical implementation of arrays and pointers in c++

Aim: To implement and test static data structures in C++, including one-dimensional arrays, multidimensional arrays, and pointer-based array manipulation.

Materials: Computer installed with a C++ compiler or online IDEs.

Instructions

Step 1: Create a C++ program (array_ops.cpp) that declares a fixed-size static array, such as int arr[100].

(a) Implement functions to:
- (i) insert an element at the end.
- (ii) delete an element at a specified index.
- (iii) display all elements.

(b) Test the program with:
- (i) Initial array: 10 20 30 40 50.
- (ii) Insert 60.
- (iii) Delete element at index 2.

(c) Display the array after each operation.

Step 2: Create a C++ program (matrix_ops.cpp) that declares two static 2×2 matrices.

(a) Perform matrix addition.

(b) Perform matrix transpose.

(c) Use the test case:
- (i) A = [[1, 2], [3, 4]]
- (ii) B = [[5, 6], [7, 8]]

(d) Display the results in proper matrix format.

Step 3: Create a C++ program (matrix_ops.cpp) that declares a static integer array

(a) Use a pointer to traverse the array using pointer arithmetic.

(b) Print the value and memory address of each element.

(c) Implement a function to reverse the array using pointers only (no indexing).

(d) Test using the array:[1, 2, 3, 4, 5]

(e) Display the array before and after reversal.

Step 4: Write a short paragraph explaining:

(a) Why arrays and pointers are considered static data structures, highlighting how their fixed size and memory allocation influence program behaviour.

FOR ONLINE READING ONLY

COMPUTER SCIENCE F6 FINAL (EVA Feb 3.indd 23 03/02/2026 15:48:52
CamScanner

(b) One advantage of static data structures in terms of efficiency or predictability.

(c) One limitation you may encounter during implementation.

(d) A real-life context where static data structures are appropriate due to stable data size or performance requirements.

Deliverables

A comprehensive report comprising of the following:

(a) Three C++ source files:
   (i) array_ops.cpp
   (ii) matrix_ops.cpp
   (iii) pointers_with_arrays.cpp

(b) Screenshots or printed outputs showing:
   (i) Array operations
   (ii) Matrix addition and transpose
   (iii) Pointer-based traversal and reversal

(c) Reflection paragraph on static data structures in C++ (From Step 4 above).

Exercise: 1.2:

1. Analyse four (4) advantages and four (4) disadvantages of static data structures, and briefly explain how each point influence program performance, memory efficiency, or system behaviour.
2. With examples, describe the types of arrays (e.g., one-dimensional, two-dimensional, and multi-dimensional), and illustrate how each type can be applied to solve a specific real-world problem.
3. Evaluate any three (3) advantages and three (3) disadvantages of arrays, and relate each point to realistic practical programming scenarios where these strengths or limitations directly affect implementation.
4. Analyse four (4) real-world applications of arrays, showing how each array structure supports efficient access, storage, or processing of data in each context.
5. Analyse the main operations performed on arrays (such as traversal, insertion, deletion, searching, and updating) and critically explain how each each operation impacts on time efficiency and overall program performance in practical programming scenarios.
6. Demonstrate how to declare a pointer in C++, and critically explain what happens in memory when the pointer is declared, including how it references or stores the address of a variable.
7. Describe four (4) advantages of using pointers in programming and explain how and when each advantage becomes particularly useful in real-world applications or practical programs.
8. Analyse three (3) key applications of pointers in C++, such as dynamic

FOR ONLINE READING ONLY

CamScanner

The pointer in each node allows the program to move from one node to the next, forming a chain. The last node in a linked list contains a null pointer, indicating the end of the list.

Linked lists are one of the most commonly used data structures after arrays, especially when efficient insertion and deletion of elements are required. The following terms are essential to understanding the concept of a linked list.

(i) Link: Each link of a linked list can store data called an element.

(ii) Next: Each link of a linked list contains a link to the following link called Next.

(iii) Linked list: A linked list contains the connection link to the first link called First.

(iv) Head: This is the first node and

(v) Tail is the last node.

Structure of linked lists

The structure of the linked list comprises two parts in each node. These parts are the one which represents data, and the other contains the address of the next node, as shown in Figure 1.20.

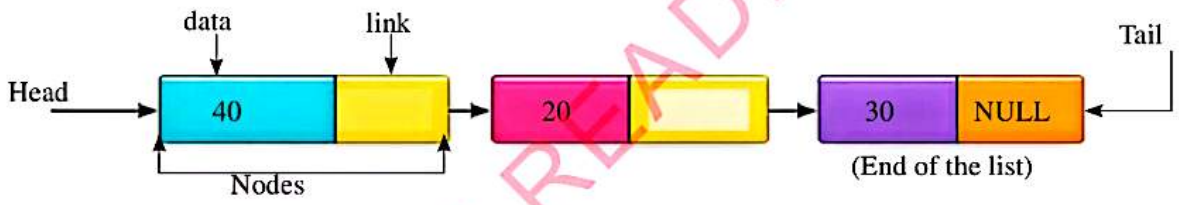


Figure 1.20: *Linked list*

Advantages of linked list

(a) Easy insertion and deletion of elements: Insertion and deletion are efficient and take constant time as the links are manipulated, hence, there is no need to give the initial size of the list.

(b) No memory wastage: the size of the linked list increases and decreases during runtime. This utilises memory efficiently, hence, no memory wastage.

(c) They can be expanded constantly: The linked list can grow and shrink depending on the data to be inserted. With a linked list, you can start with space for just one allocated element and then add on new elements easily without the need to do any copying and reallocating.

(d) Implementation: The implementation of a stack and a queue is very easy using a linked list.

(e) Ease of moving pointer: In the linked list, it is very easy to insert and delete a node. After insertion and deletion, there is no need to shift every element; instead, it surffices to update the next pointer.

Disadvantages of linked lists

(a) Memory usage: The node in a linked list occupies more memory than the array as each node occupies two types of variables, i.e., one is a simple variable, and another is a pointer variable that occupies 4 bytes in memory.

(b) Traversal: In a linked list, traversal is not

FOR ONLINE READING ONLY

COMPUTER SCIENCE F6 FINAL (EVA Feb 3.indd 26 03/02/2026 15:48:52
CamScanner

easy. If you want to access the element in a linked list, you cannot access the element randomly, but in the case of an array, you can randomly access the element by index. For example, if you want to access the 3rd node, you must traverse all the nodes before it. So, the time required to access a particular node is extensive.

(c) Reverse traversing: Backtracking or reverse traversing is difficult in a linked list. It is easier but requires more memory to store the back pointer.

Application of linked lists

The following are some applications of linked lists:

(a) Implement dynamic memory management functions of the operating system.

(b) A circular linked list is used to implement operating system or application functions requiring round-robin tasks execution.

(c) The linked list is used in a circular queue to maintain multiple players' playing sequences in a game.

(d) Implement stacks, queues, binary trees and graphs of pre-defined size.

(e) The circular linked list is used in a slide show where a user wants to go back to the first slide after displaying the last one.

(f) A linked list is used for the polynomial implementation of mathematical operations.

(g) A doubly linked list is used to implement forward and backward buttons in a browser to move backwards and forward in the opened pages of a website.

Types of linked list

Linked lists can be divided into three types:

(a) A singly-linked list is a linked list traversed in one direction from head to tail.

(b) The doubly linked list is a type of linked list traversed in both directions, front and back.

(c) A circular linked list is a list in which the last node points to the first node.

Linked lists operations

The basic operations performed on the linked list are creation, insertion, deletion, traversing, searching, and concatenation.

Creation operation: This is used to create a linked list. Once a linked list is created with one node, the insertion operation can add more elements to a node.

Insertion operation: This is the type of operation used to insert a new node at any specified location in the linked list. A new node may be inserted:

(a) At the beginning of the linked list.

(b) At the end of the linked list.

(c) At any specified position in between in a linked list. For example, Figure 1.21 shows the insertion operation of inserting a new node at the beginning of an existing linked list where:

(i) Figure 1.21 (a) shows an existing linked list of the regions in both Tanzania mainland and Zanzibar island.

FOR ONLINE READING ONLY

CamScanner

(ii) Figure 1.21 (b) shows the insertion of a new region called Tabora, and

(iii) Figure 1.21 (c) shows the resulting new liked list after the insertion operation.

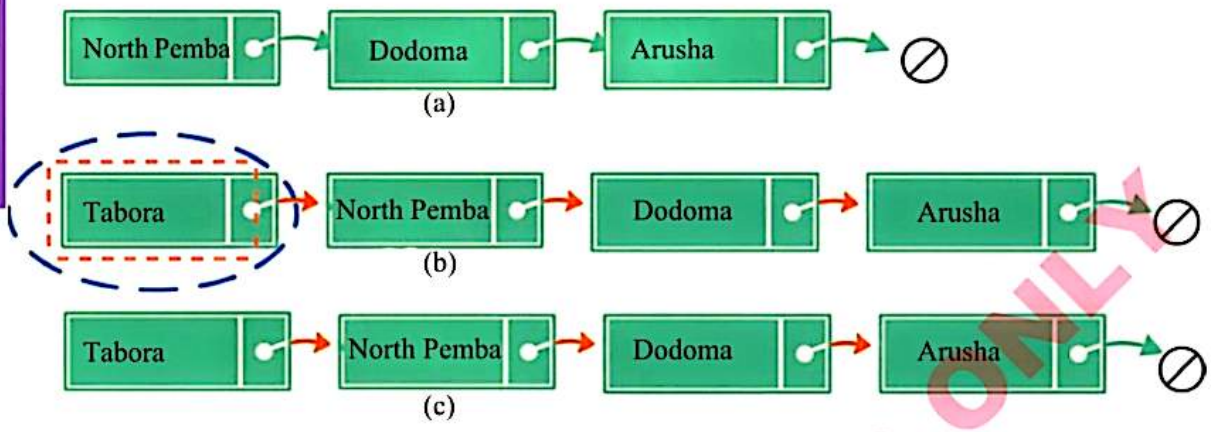


Figure 1.21 (a): *Insertion operation*

The deletion operation: This is used to delete an item (a node) from the linked list. A node may be deleted from the;

(a) Beginning of a linked list

(b) End of a linked list and

(c) Specified location of the linked list. (See Figure 1.22).

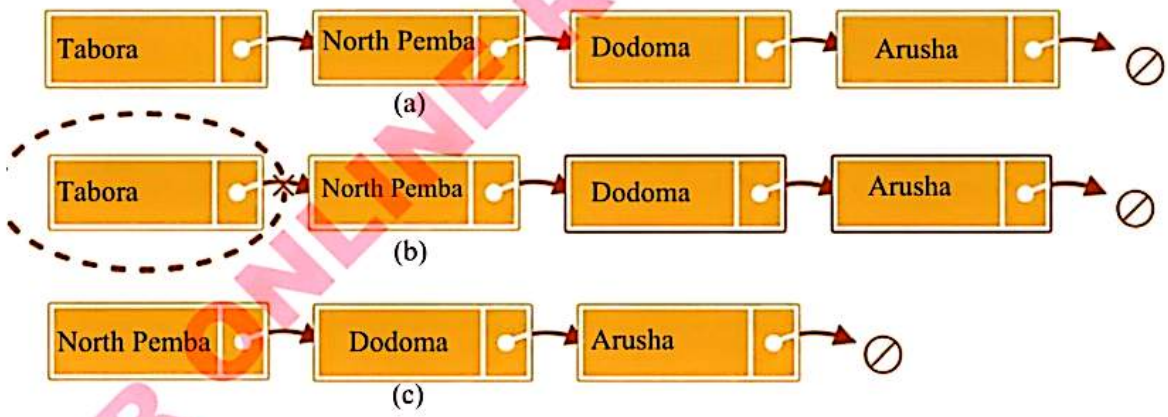


Figure 1. 22 (b): *Deletion operation*

Simulating insertion in linked list using VisualGo program

Activity 1.4:

Visualising linked list insertion using Visualgo

Aim: To visualise the structure and behaviour of a linked list, and to develop analytical and problem-solving skills by observing how insertion at the beginning of a linked list is performed.

FOR ONLINE READING ONLY

CamScanner

Materials: Computer, laptop, tablet, or smartphone; Internet connection; and web browser

Instructions

Step 1: Access Visualgo

(a) Open a web browser.

(b) Visit https://visualgo.net.

(c) From the main menu, select and Click on Linked List, and will show up as in Figure 1.23.

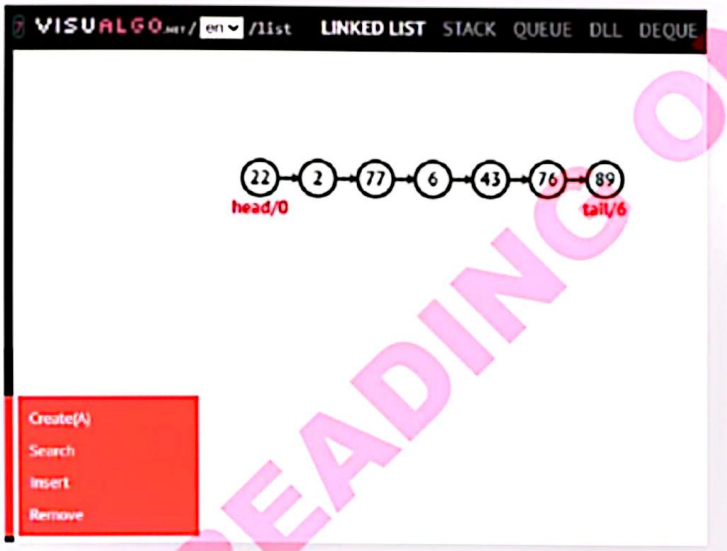


Figure 1. 23: *Linked list interface*

Step 2: Create the Initial Linked List as shown in Figure 1.24

(a) Choose the option to **create or input a linked list**.

(b) Enter the following elements in order: 25 → 35 → 45 .

(c) Initialise the list.

(d) Observe how:

(i) Each node contains **data** and a **pointer (link)**

(ii) Each node points to the next node in the list

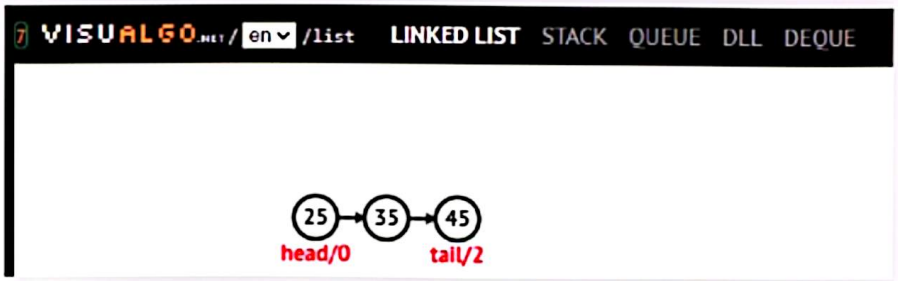


Figure 1. 24: *Initial linked list*

Step 3: Insert a Node at the Beginning

(a) Select the **Insert at Head / Insert at Beginning** operation, as shown in Figure 1.25.

FOR ONLINE READING ONLY

CamScanner



Figure 1. 25: *Inserting a node*

(b) Enter the new value: 59

(c) Start the animation.

(d) Observe carefully: A new node is created for 59; The new node's pointer links to 25; and 59 becomes the new head of the linked list as shown in Figure

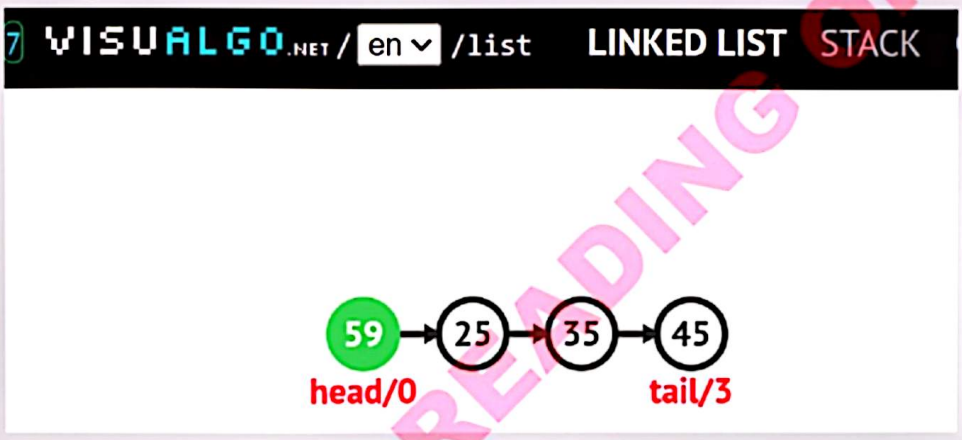


Figure 1. 26: *Inserted value*

Step 4: Observe the final structure

(a) Pause or replay the animation.

(b) Confirm the new order of the linked list:

(c) 59 → 25 → 35 → 45

(d) Compare this process with array insertion at the beginning.

Deliverables

(a) A written or sketched diagram showing initial a linked list

(b) A written or sketched diagram showing new linked list

(c) A short explanation of about (3 to 5 sentences) describing:

(i) What happens to pointers during insertion

(ii) Why linked list insertion does not require shifting elements

Traversing: This is the process of going through all the nodes from one end to another end of a linked list. You can visit from left to right in a singly linked list, forward traversing, nodes only. In a doubly linked list, forward and backwards traversing is also possible. You may observe this during simulations of these operations.

Concatenation: This is the process of appending the second list to the end of the first list. For example, consider a list A having x nodes, and list B with y nodes; then the concatenation operation will place the 1st node of B in the $(x+1)^{th}$ node in A. After concatenation, A will contain (x+y) nodes, as shown in Figure 1.27.

FOR ONLINE READING ONLY

CamScanner

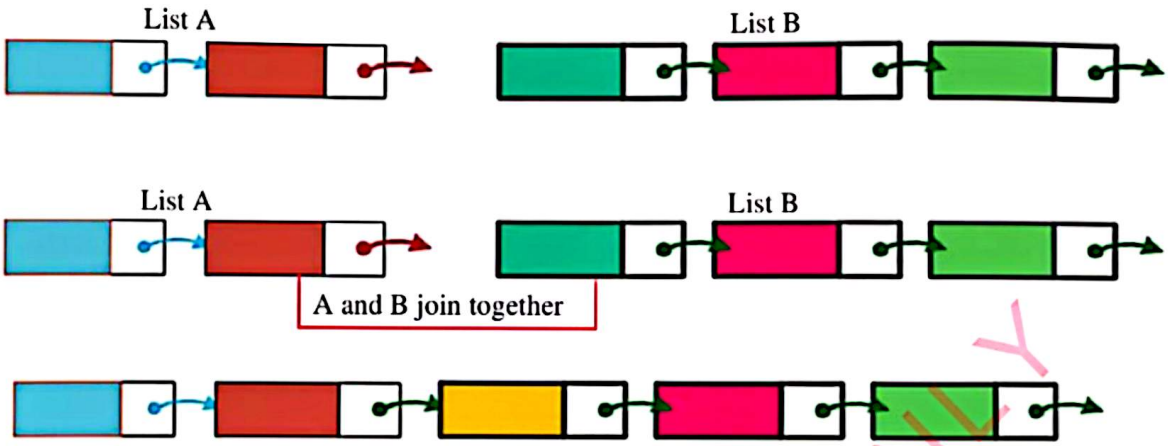


Figure 1.27: *Concatenation in a linked list*

Stacks

A stack is an Abstract Data Type (ADT) and is a linear data structure that can be accessed only at the top for storing and retrieving data. It is analogous to a stack of trays in a cafeteria whereby new trays are put on the top of the stack and taken out from the top. The last tray placed on the stack is the first tray removed from the stack and based on this principle; a stack is called a Last-In-First-Out (LIFO) structure. Here, the element placed (inserted or added) last is accessed first. In stack terminology, the insertion operation is called the push operation and the removal operation is called the pop operation. . Therefore, a stack contains operations that change its status and operations that check this status. The operations are as follows;

(i) clear(): Clear the stack.

(ii) IsEmpty(): determines whether the stack is empty. It returns 1 (true) if the stack is empty otherwise, it returns 0 (false).

(iii) push(): Put an element on the top of the stack.

(iv) pop(): Take the topmost element from the stack.

(v) topEl(): Return the topmost element in the stack without removing it.

(vi) Size(): Returns the number of elements in the stack.

Structure of stacks.

The stack can be demonstrated as in Figure 1.28.

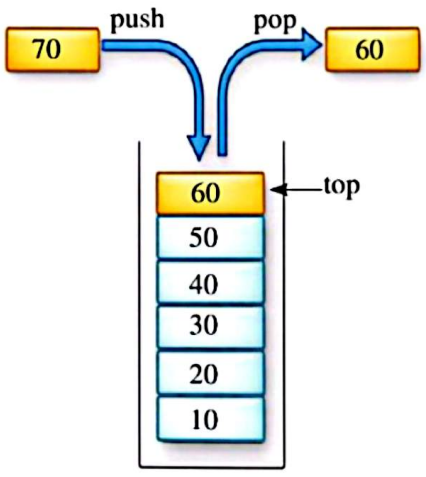


Figure 1. 28: *The stack structure and its operation*

FOR ONLINE READING ONLY

CamScanner

Application of stacks

Generally, stacks is beneficial when data are stored and retrieved in reverse order. One application of stacks is in matching delimiters in a program. This is an example because delimiter matching is part of any compiler: No program is considered correct if the delimiters are mismatched. In C++ programs, we have the following delimiters: parentheses "(" and ")", square brackets "[" and "]", curly brackets "{" and "}", and comment delimiters "/*" and " */". It is also used to undo, redo, and in navigation bars.

Stacks operations

PUSH operation

The mechanism of PUSH operation in a stack is conducted as follows:
Before inserting an element in a stack, you must check whether the stack is full. If you attempt to add an element in a stack that is already full, the overflow condition happens. When you create a stack, set the top pointer as - 1 to indicate that the stack is empty. When the new element is pushed onto the stack, the top pointer gets incremented by 1 (i.e., top=top+1), and the element will be placed at the new position of the top pointer. The elements can be pushed onto the stack until the top pointer reaches the maximum allowed index (such as top = 3 for a stack of size 4), as shown in Figure 1.29.

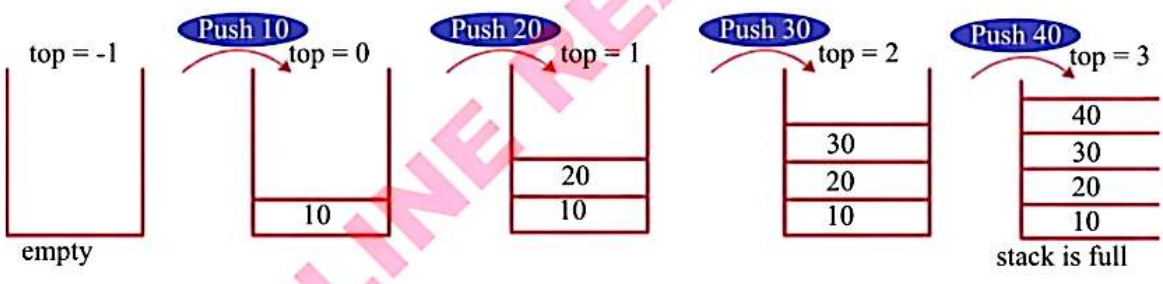


Figure 1.29: *Push operation*

POP operation

The mechanism for POP operation in a stack is conducted as follows:

(a) Before removing an element from the stack, you must check whether the stack is empty.

(b) If you attempt to remove an element from an empty stack, an underflow condition happens.

(c) If the stack is not empty, you first access the element at the top of the stack.

(d) Once the pop operation is performed, the top pointer is decremented by 1 that is, top = -1.

Queues

The queue is an abstract data structure, somewhat similar to stacks. Unlike stacks, a queue is open at both ends. The queue follows the First-In-First-Out (FIFO) methodology; that is, the data item stored first will be accessed first. Queues are what you usually refer to as lines. Its essential features are, that it is ordered and that

FOR ONLINE READING ONLY

COMPUTER SCIENCE E6 FINAL (EVA Feb 3.indd 32 03/02/2026 15:48:53
CamScanner

access is restricted to its ends: things can enter a queue only at the rear and leave the queue only at the front.

The queue holds a sequence of elements that allows insertions (enqueuer) only at one end, called the back or rear, and deletions (dequeuer) and access to elements at the other end, called the front. In computing, queues are essential because resources provide service on a first-come-first-served basis, such as jobs sent to a printer or processes waiting for the CPU in an operating system.

A real-world example of a queue can be a one-way road or single-lane bridge, where the vehicle first enters and exits. More real-world examples are queues at the ticket windows and bus stops, banks, queuing for voting, taking food, and hospitals. Figure 1.30 shows the structure of the queues.

Structure of queues

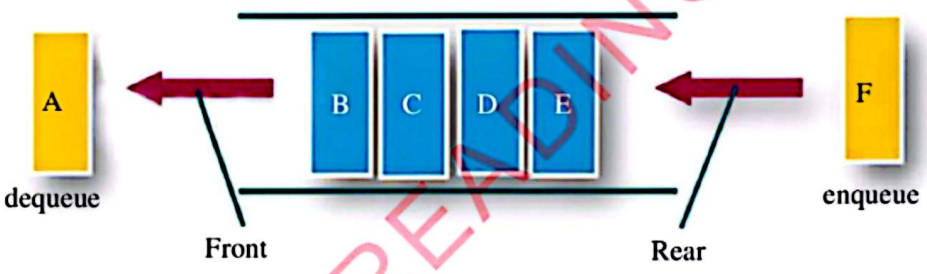


Figure 1.30: *Structure of the queue*

Application of queues

The real-life queue is realised in what you all participate in from time to time. The following are some of the applications that use queues:

(a) You wait in line at railway and bus stations for reservation counters and tickets.

(b) You wait in the dining halls and cafeteria line.

(c) You wait in a queue in the hospital when you want to present your illness case to a doctor.

(d) You issue a print command from your office to a shared networked printer; the print task is lined up in a printer queue. The print command issued first will be printed before the subsequent commands in the line (in the queue).

(e) An operating system is also using different queues to control process scheduling. Processes are added to the processing queue and used by an operating system for various scheduling algorithms.

(f) The first-served scenario requires a queue to simulate real-world queues such as lines at a bank counter or any other first-come.

Queue operations

Similar to stacks, queues support several operations, including the following:

FOR ONLINE READING ONLY

CamScanner

(a) Insert (or add) an element to the queue (enqueue)
(b) Delete (or remove) an element from a queue (dequeue)
(c) Getting the size of a ques (size)
(d) Clearing a queue (clear)

Enqueue operation will insert (or add) an element to the queue at the rear end by incrementing the array index. The dequeue operation will delete (or remove) from the front end by decrementing the array index and will allocate the deleted value to a variable.

When implemented using arrays, the total number of elements present in the queue is front − rear + 1. Figure 1.31 illustrates the basic operations of the queue.

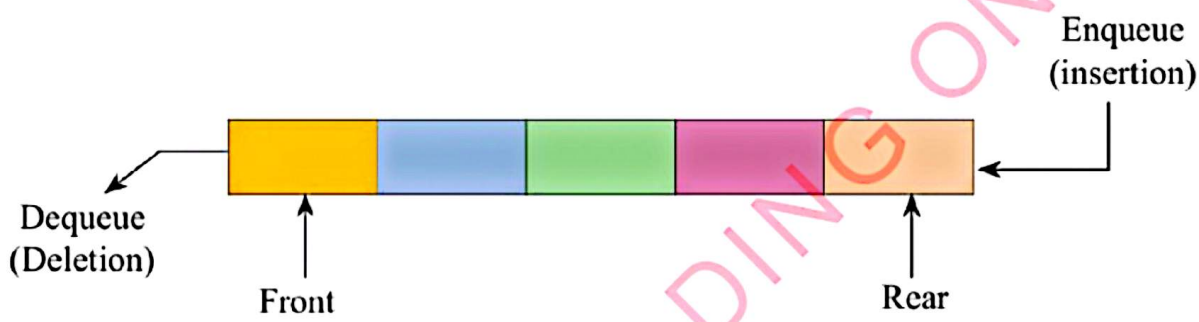


Figure 1.31: *Queues operation*

Simulating a Queue in Scratch program (FIFO) using a List

Activity 1.5:

Simulating a queue (FIFO) using a list in Scratch

Aim: To simulate and visualise the behaviour of a queue (First-In, First-Out) using a Scratch list, and to develop problem-solving and logical-thinking skills by applying enqueue and dequeue operations in a simple program.

Materials: Computer or laptop (or tablet with keyboard); Scratch programming environment (Online: https://scratch.mit.edu, Offline: Scratch Desktop (installed); and Internet connection (for online Scratch only)

Step 1: Create the Queue List in Scratch

(a) Open ***Scratch***

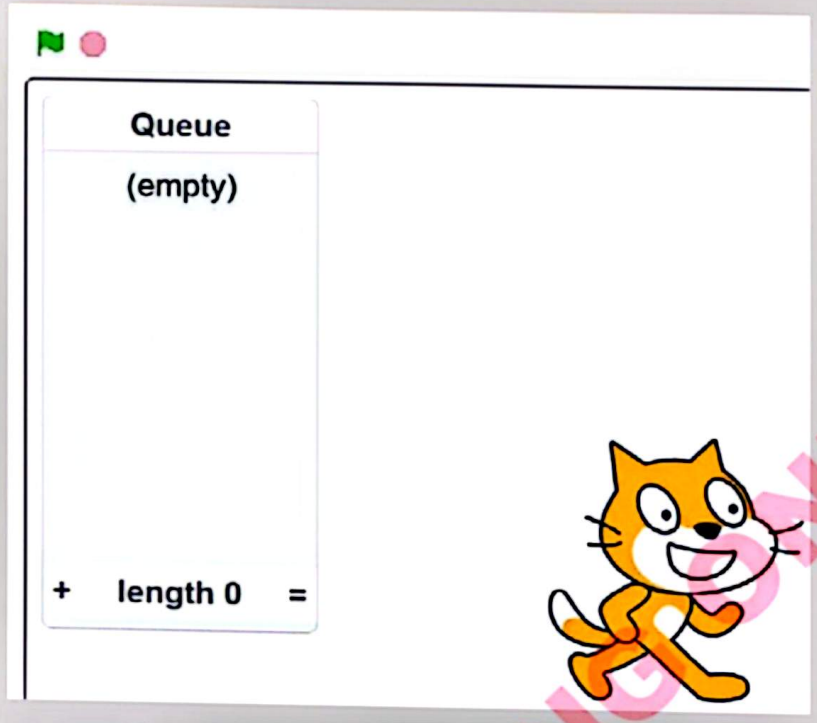
(b) Go to ***Variables → Make a List***

(c) Name the list: ***Queue***

(d) Choose For all sprites, the output will look as shown in Figure

FOR ONLINE READING ONLY

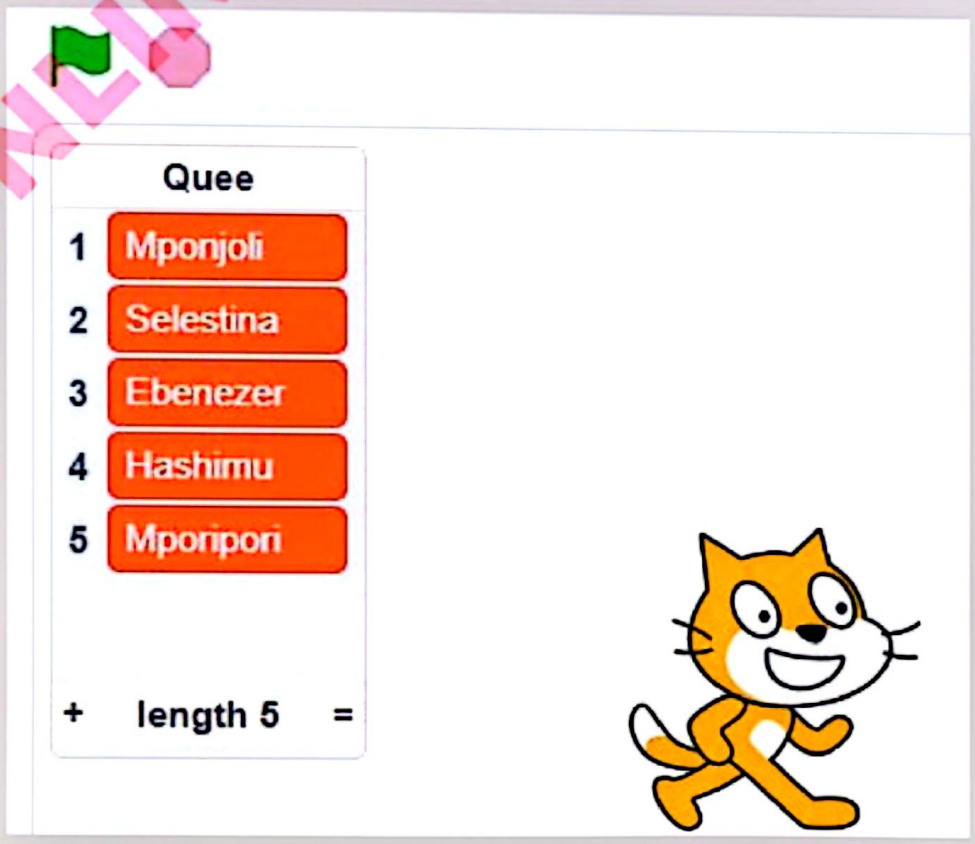
CamScanner



Step 2: Enqueue operation.
Add [PersonName] to [Queue]



After repeating addition, the output will look like the one in Figure



FOR ONLINE READING ONLY

CamScanner

Note:

Scratch lists are numbered like this:

1st item → position 1

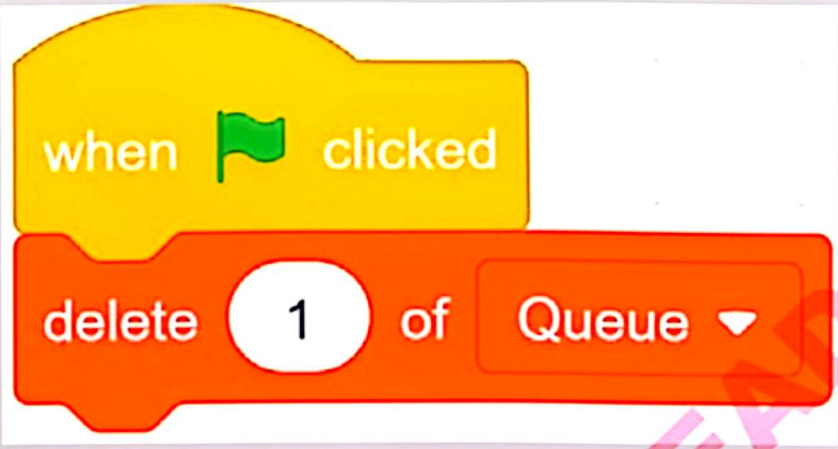
2nd item → position 2

3rd item → position 3

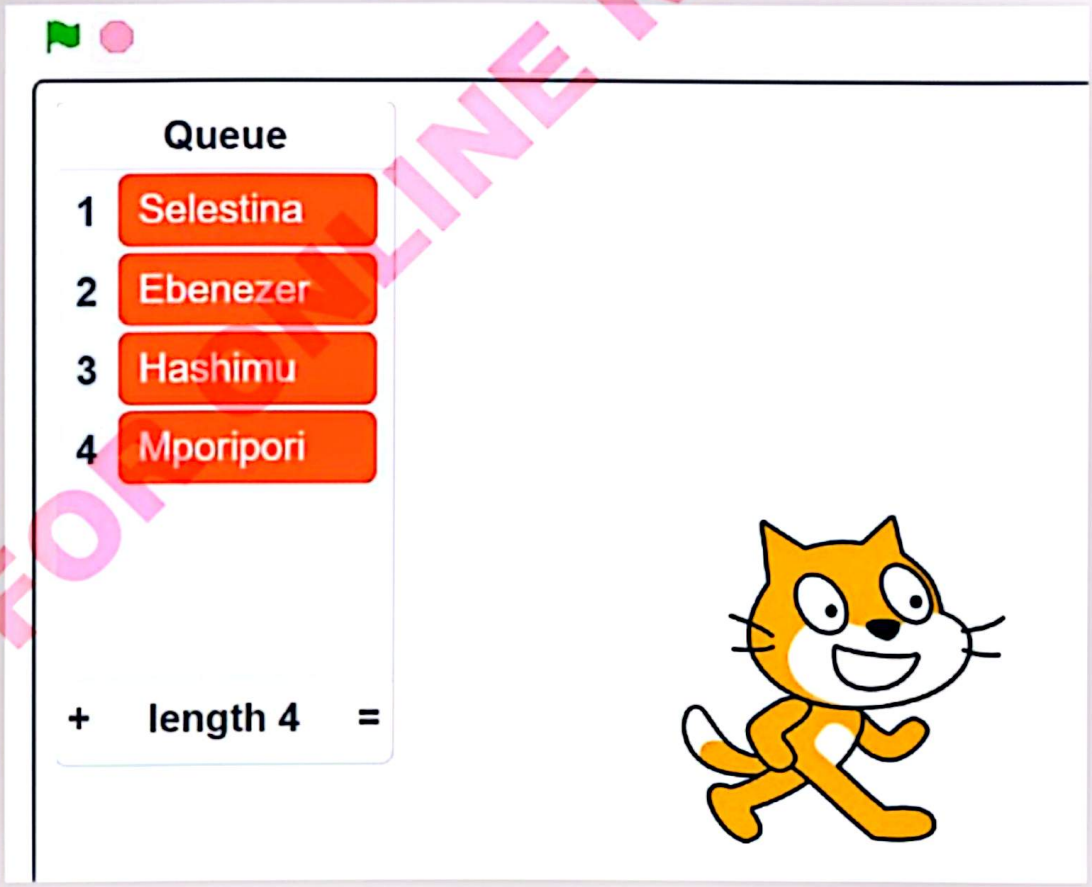
So the front of the queue is item 1

Step 3: Dequeue operation

Remove the first person, Mponjoli leaves first → FIFO



After the deletion operation, the output will look like the one in Figure

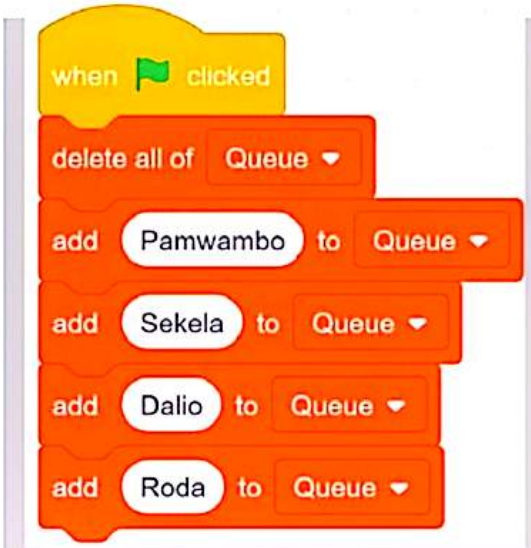


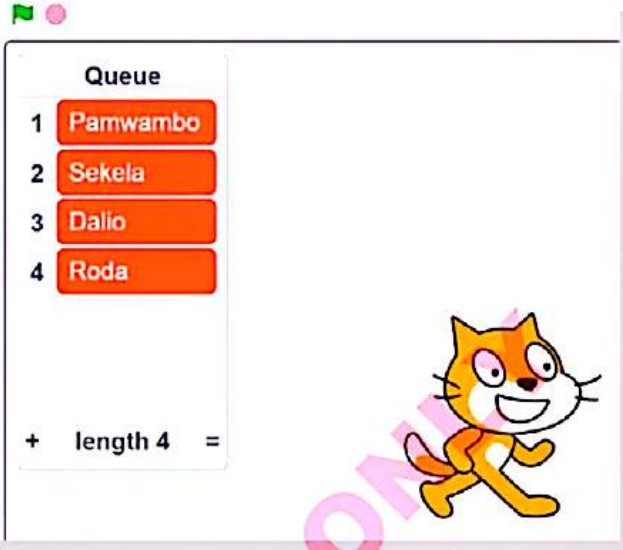
Step 4: Putting enqueue and dequeue together

Combine both operations.

FOR ONLINE READING ONLY

CamScanner





Deliverables

(a) A working Scratch project that creates a list named *Queue,* uses the list to represent a FIFO queue, allows adding items (enqueue), and allows removing items from the front (dequeue).

(b) Evidence (screenshot or observation) showing items added to the Queue list, items removed in the same order they were added.

(c) A short explanation of about (3 to 5 sentences) describing how the Scratch list behaves like a queue, and why the first item added is the first one removed.

FOR ONLINE READING ONLY

Non-linear data structures

A non-linear data structure organises data elements in a non-sequential, hierarchical, or networked manner, allowing for complex relationships between elements rather than a single, straight line. Unlike linear structures, non-linear structures don't have a single level, making traversal more complex and often requiring specific algorithms like Depth-First Search (DFS) or Breadth-First Search (BFS). Common examples include trees, which show parent-child relationships, and graphs, which represent interconnected networks.

Tree

A tree is a non-linear data structure that is an abstract and typically hierarchical structure consisting of nodes with a parent-child relationship. It is used in organisation charts, programming environments, and file systems. There are four basic things connected with any tree: the distinction between nodes, orientation structure, the value of nodes, and the number of levels. Examples of tree data structures include binary tree, heap, and splay.

Binary tree

A binary tree is a tree for which each internal node is allowed to have at most two child nodes. The children of an internal node are called the left child and the right child, depending on the relationship with the parent. The recursive definition of a binary tree is either a tree consisting of a

CamScanner

single node or a tree whose root has an ordered pair of children, each of which is a binary tree. The applications of binary trees represent arithmetic expressions, create decision processes and search. Figure 1.32 illustrates the structure of the binary tree.

Structure of the binary tree

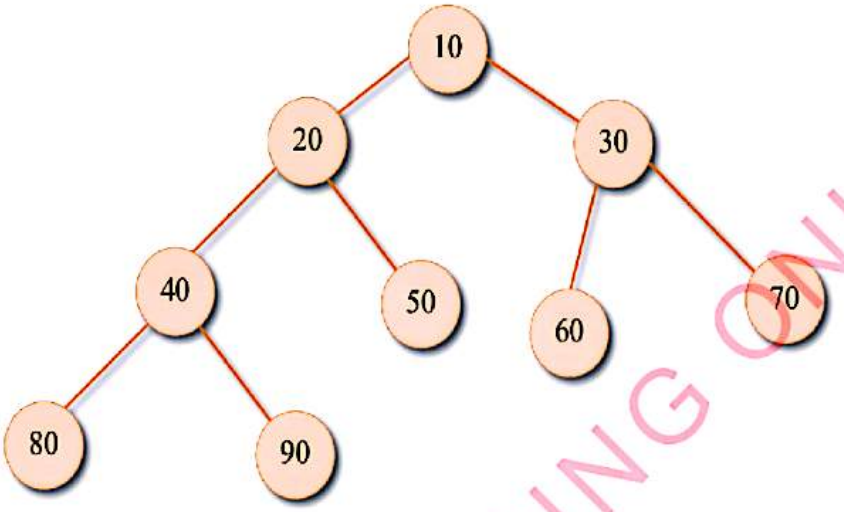


Figure 1.32: *Binary tree*

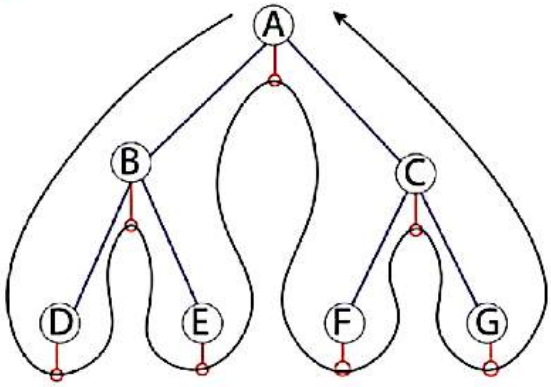
FOR ONLINE READING ONLY

Binary tree operations

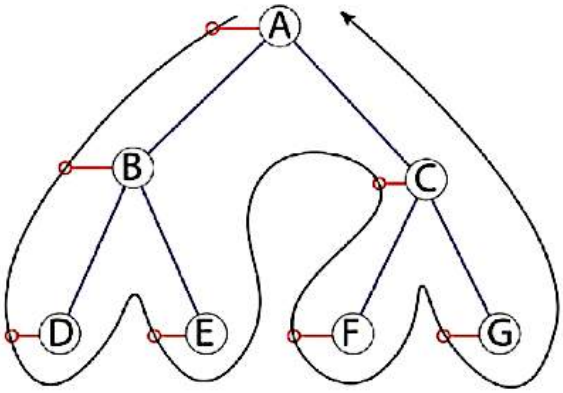
(i) Preorder traversal: A node is visited before its descendants (in the example, the Preorder traversal sequence is 10,20,40,80,90,50,30,60,70).

(ii) Inorder traversal: A node is visited between the traversal of its left and right descendants. In the example, the Inorder traversal sequence is 80,40,90,20,50,10,60,30,70.

(iii) Postorder traversal: A node is visited after its descendants (in the example, the Postorder traversal sequence is 80,90,40,50,20,60,70,30,10). Figure 1.33(a) to 1.33(c) shows the binary tree traversal which will help you to understand the binary tree traversal quickly.

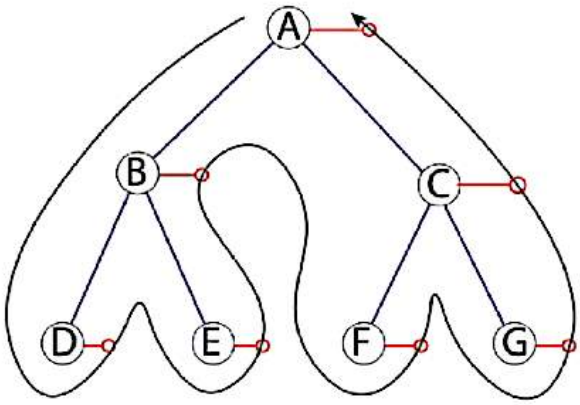


(a) Inorder D, B, E, A, F, C, G



(b) Preorder A, B, D, E, C, F, G

COMPUTER SCIENCE F6 FINAL (EVA Feb 3.indd 38 03/02/2026 15:48:55
CamScanner



(c) Postorder A, B, D, E, C, F, G

Figure 1.33: *Binary tree traversing*

Note: If n is the number of nodes, x is the number of external nodes, i is the number of internal nodes, and h is the height of the binary tree, then the following are the essential properties of binary trees:

$x = i + 1$

$n = 2 * x - 1$

$h >= \log_2 x$

$h >= \log_2 (n + 1) - 1$

In order operation in a binary tree

(i) Traverse the left subtree, such as call Inorder (left subtree).

(ii) Visit the root.

(iii) Traverse the right subtree, such as call Inorder (right-subtree).

Uses of Inorder traversal

For example, in binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal can be reversed. For example, Inorder traversal for Figure 1.34 given tree is 14, 26, 58, 10,30.

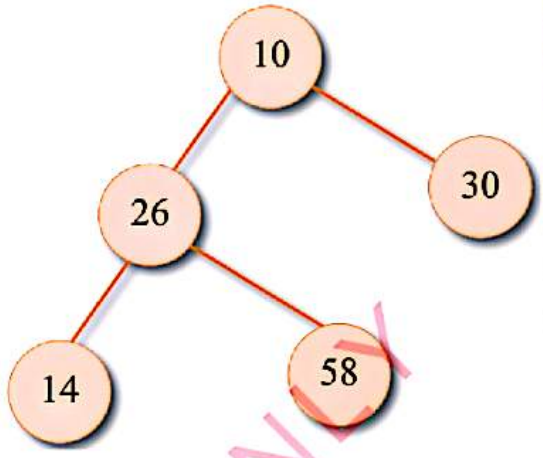


Figure 1.34: *The Inorder traversal*

Preorder operation in a binary tree

The following is an algorithm for Preorder traversal:

(i) Visit the root.

(ii) Traverse the left subtree, that is call Preorder (left-subtree).

(iii) Traverse the right subtree, that is call Preorder (right-subtree).

Uses of Preorder traversal

(i) Preorder traversal is used to create a copy of the tree.

(ii) Preorder traversal is also used to get prefix expressions on an expression tree.

Example 1: The Preorder traversal for the given Figure 1.35 is 11, 23, 41, 15, 13.

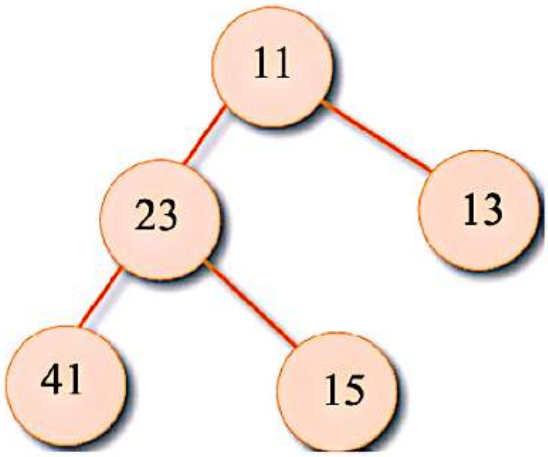


Figure 1.35: *Preorder traversal*

FOR ONLINE READING ONLY

COMPUTER SCIENCE F6 FINAL (EVA Feb 3.indd 39 03/02/2026 15:48:55
CamScanner

Example 2: Study the input in Figure 1.36. Then, write its corresponding output by using preorder traversal.

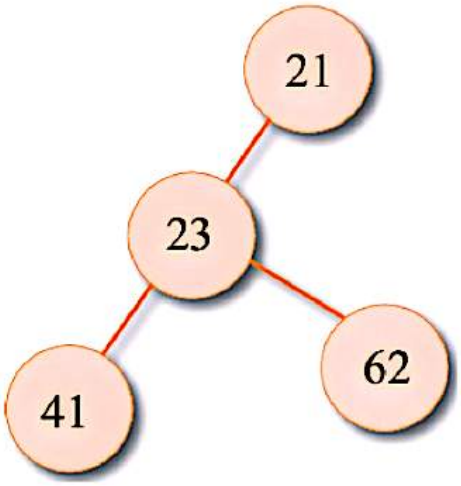


Figure 1.36: *Preorder traversal*

Output: 21, 23, 41, 62

Postorder traversal

The algorithm for Postorder (binary tree):

(i) Traverse the left subtree, that is call Postorder (left-subtree).

(ii) Traverse the right subtree, that is call Postorder (right-subtree).

(iii) Visit the root.

Uses of Postorder traversal

(i) Postorder traversal is used to delete the tree.

(ii) Postorder traversal is also helpful for getting the postfix expression of an expression tree.

Example 3: Postorder traversal for the given Figure 1.37 is 4, 5, 2, 3, 1.

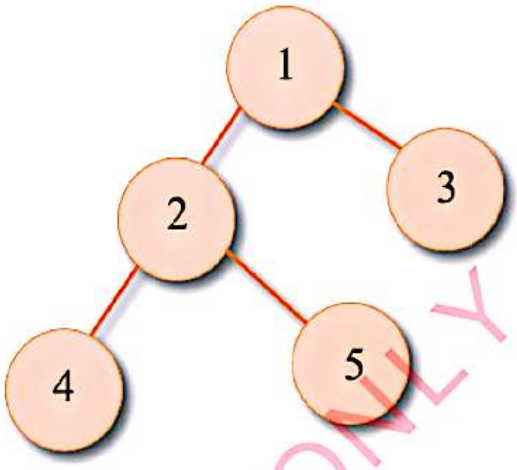


Figure 1.37: *The postorder traversal tree example*

Example 4: Consider input as given in Figure 1.38 and its corresponding output using postorder traversal:

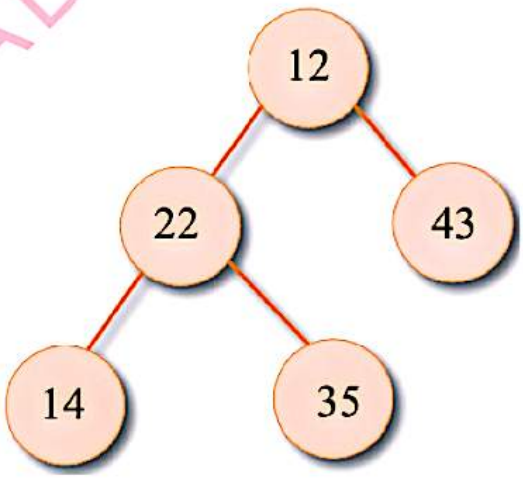


Figure 1.38: *Postorder traversal*

Output: 14, 35, 22, 43, 12

Application of a binary tree

The following are the applications of binary tree:

(a) The binary tree is used to implement the indexing of the segmented database.

(b) It is used in implementing efficient cache in hardware and software systems.

FOR ONLINE READING ONLY

COMPUTER SCIENCE F6 FINAL (EVA Feb 3.indd 40 03/02/2026 15:48:55
CamScanner

(c) A binary Search Tree is used to search elements efficiently and as a collision handling technique in hash map implementations.

(d) The binary tree is the primary data structure in Microsoft Excel and spreadsheets.

(e) A binary heap (Binary tree variant of the heap) is used to implement a Priority Queue efficiently, which is used in the Heap sort algorithm.

(f) A binary tree with nodes as operations is used to compute arithmetic expressions in some compilers.

(g) A balanced Binary Search Tree is used to represent memory to enable fast memory allocation.

(h) A self-balancing Binary Search Tree is used in implementing games to model a faulty search process.

(i) Binary trees in which each node has two child nodes can be used to represent routing data that changes frequently, allowing for efficient traversal and updates.

Activity 1.6:

Working with user-defined types and constructing dynamic data structures

Aim: To apply user-defined data types through simple C++ programming and to construct visual representations of dynamic data structures using drawing and diagram-based tasks.

Materials: Computer installed with a C++ compiler or online IDEs, textbook for reference, exercise book for notes, pen or pencil.

Instructions

Step 1: Prepare drawing tools to draw a linked list presenting sequnces.

Step 2: On paper, draw a singly linked list representing the sequence: $10 \rightarrow 20 \rightarrow 30 \rightarrow 40$. Your diagram must include:

(a) Each node (data + pointer field).

(b) Pointer arrows connecting the nodes.

(c) A NULL pointer at the end.

Step 3: Using your linked list from Step 2:

(a) Redraw the linked list after inserting 25 between 20 and 30.

(b) Redraw the linked list after deleting 30.

(c) Redraw the linked list after adding 50 at the end.

Step 4: Draw a stack showing the sequence of operations "Push 5, Push 10, Push 15, Pop, Push 20". Show the stack diagram after each operation (5 diagrams total).

Step 5: Draw a queue showing "Enqueue 7, Enqueue 14, Enqueue 21, Dequeue, Enqueue 28". Show the queue after each step (5 diagrams total).

Step 6: Draw the following binary tree, as shown in Figure 1.39.

FOR ONLINE READING ONLY

CamScanner

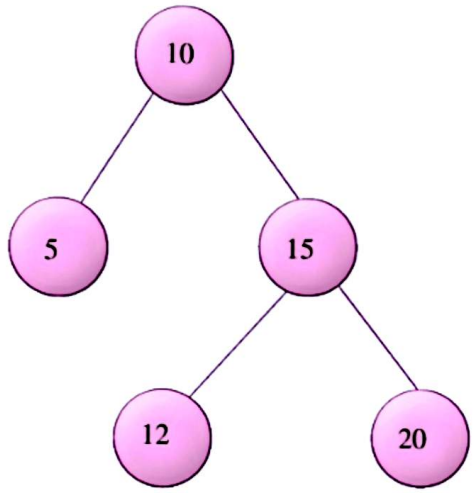


Figure 1.39: *Binary tree*

Step 7: Using the tree from Step 6, trace and write the sequence of nodes visited during:

(a) Preorder traversal.
(b) Inorder traversal.
(c) Postorder traversal.

Step 8: On one page, side-by-side, draw:

(a) One static array of 5 elements.
(b) One linked list of 5 nodes.
(c) Use labels and arrows to visually highlight the difference in structure.

Deliverables

A report comprising of the following:

(a) Coding Output: user_defined_types.cpp (compiled and run successfully)
(b) Drawing-based deliverables:
  (i) Linked list diagrams (initial + modified versions).
  (ii) Stack operation diagrams.
  (iii) Queue operation diagrams.
  (iv) Binary tree diagram.
  (v) Sequence outputs for tree traversals.
  (vi) Visual comparison page (array vs. linked list)

Document all your work in your portfolio.

Exercise 1.3

1. How does using a structure or class help organise related data in a program?
2. In what situation would an enumeration (enum) make code easier to understand?
3. Why are dynamic data structures useful when the amount of data is not known in advance?
4. Analyse one challenge that may occur when using pointers in dynamic data structures?
5. Explain why a linked list is more flexible than an array when inserting elements and illustrate with an example of a scenario where this flexibility improves program efficiency.
6. Compare the operational rule of a stack and a queue while explaining how these rules affect the way data is processed in practical applications.
7. Analyse how the structure of a tree enables hierarchical organisation of data, and provide an example of a real-world application where this hierarchy is beneficial.
8. Justify why traversal is necessary in a binary tree before processing or searching for a node, and then explain how different traversal

FOR ONLINE READING ONLY

CamScanner

methods (in-order, pre-order, post-order) influence the order and efficiency of data access.

Algorithms

Concept of algorithm

One of the significant steps to solving any problem is to analyse the problem. Analysing a problem involves understanding the problem and its requirement. Problem requirements include users, input data, output data and general system functionality. For example, when one wants to develop a school management system, the requirements might be the number of students, teachers, and the school administrator who will be responsible for updating data. After analysing a problem, the next step is to design an algorithm.

An algorithm involves step-by-step analysed procedures to solve a particular problem. It is a set of instructions written to accomplish the defined task. The algorithm is not a program, so you do not need to use a programming language to design it. It is just a way of solving a problem systematically, which can be represented by either a flowchart or pseudocode. The algorithm should define the beginning and end of the problem with a specific number of steps. The steps of the algorithm normally involve decisions, including logic and comparison. All programs designed within the computer system are the implementation of the algorithm. Algorithms are used for calculation, data processing and automatic reasoning. However, the concept of the algorithm can be applied even in solving typical daily life tasks, such as cooking food, studying, teaching, and travelling.

Types of algorithms

In algorithms, similar problem-solving approaches are grouped together so that different types of algorithms can be classified and analysed more easily. This helps in understanding the problem clearly and finding an efficient solution. There are many types of algorithms, but this book focuses on four fundamental categories, recursive, iterative.

(a) Recursive algorithms

This is an algorithm that solves a problem by repeatedly calling itself with smaller or simpler input values. Each recursive call works on a reduced version of the original problem, where the results of one call becomes part of the solution for the next call. This continues, gradually building the final answer, until a base condition is reached. Suppose you want to arrange three books labelled A, B, and C, the following are the possible ways: ABC, ACB, BAC, BCA, CAB and CBA. This means there are six ways of arranging any three objects. Mathematically, that scenario can be solved by using the concept of factorial of n-object, for example, a 3! = 3*2*1 = 6 ways. This signifies that if five students are sitting on the horizontal bench, the ways of arranging them will be 5! = 5*4*3*2*1 = 120 ways. Generally, methods of organising any given n-objects will be as:

$$n! = n \times (n-1) \times (n-2) \times (n-3) \times (n-4) \times \cdots \times 2 \times 1$$

FOR ONLINE READING ONLY

CamScanner

This system of breaking a problem into more minor problems using the solution applied to solve the main problem is called recursion.

The algorithm that applies recursion to solve the problem is known as a recursive algorithm. Recursive algorithms solve a problem by breaking it into a simple problem using the same method by repeatedly calling themselves.

Implementation of a recursive algorithm

You can use a recursive function to implement recursive algorithms. The recursive function contains a statement that causes the same function to execute again before completing the current call. It is the function that calls itself until the given condition is met. The result from one execution in a recursive function is used as the input of the next recursive operation.

The following example presents a C++ program code to demonstrate the recursive function:

```
int factorial (int n) {
 if(n==0)
 return 1;
 else
 return(n*factorial(n-1));
 }
```

From the given demonstration code above:

factorial (3) = 3*factorial(2), factorial (2) = 2*factorila(1), factorial(1) = 1* factorial (0), and factorial (0) =1. Then, factorial(3) = 3*2*1*1. Therefore factorial(3) = 6.

The given solution explains that a problem is split into more minor instances of the same problem.

Note: The recursive function divides a problem into the base case and the recursive step, as shown in Figure 1.40. The base case is the most straightforward instance of the problem that stops the recursion function. It provides the result of the function when the condition is met. Also, the recursive step is responsible for providing solutions by making recursive calls and decreasing the size of inputs.

Consider some of the positive integers:

$$1+2+3+4+5+\cdots n+1=$$

FOR ONLINE READING ONLY

CamScanner

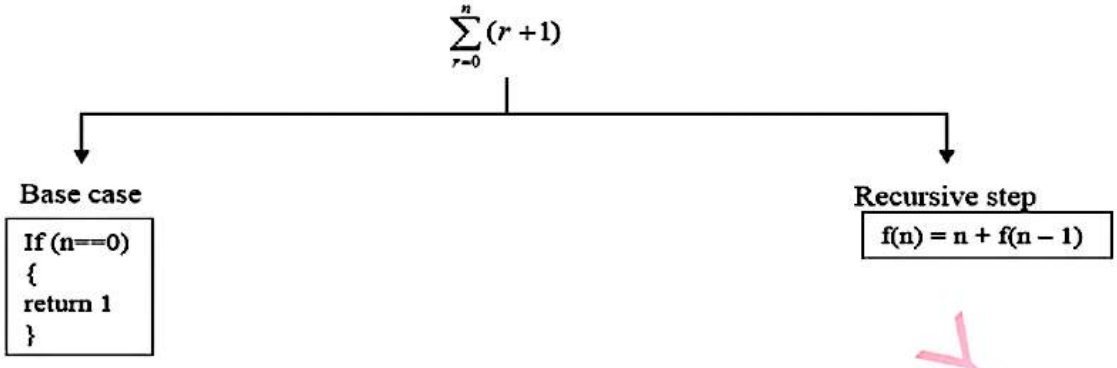


Figure 1.40: *Recursive diagram for base and function*

Note: If there is no base case in the recursive function, the recursive function will continue to repeat.

Recursive execution can be presented by using a data flow model. The box represents procedure invocation through data flow into and out of the box shown in Figure 1.41. Figure 1.42 presents the recursive data flow model and the use of the data flow model to solve the recursive problem, respectively.

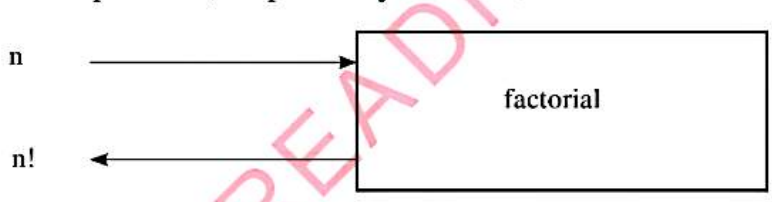


Figure 1.41: *Data flow model for recursive execution*

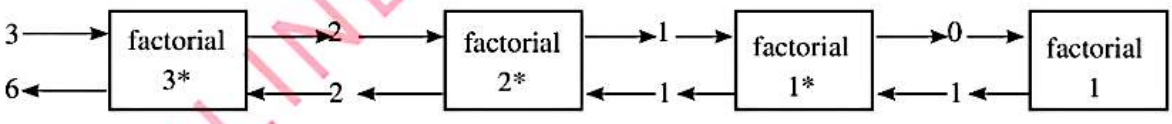


Figure 1.42: *Example of a recursive data flow model during execution*

Example 5: Write a recursive algorithm to find the sum of n-integers. The general algorithm for the recursive algorithm will be given as:

Step 1: Start

Step 2: Assign sum = 0 and count = 0

Step 3: Enter the range, n

Step 4: Compute count = count + n(count)

Step 5: Compute sum = sum + i

Step 6: Repeat step 5 and 4 until count = n

Step 7: Print sum

Step 8: Stop

Example 6: Implement a recursive algorithm to find the sum of n-consecutive integers in C++.

FOR ONLINE READING ONLY

CamScanner

Given syntax:

```
#include <iostream>
using namespace std;

int sum(int n)
{
    if (n != 0)
    {
        return n + sum(n - 1);
    }
    return 0;
}

int main()
{
    int n;
    cout << "Enter the number: ";
    cin >> n;
    cout << "Sum is: " << sum(n);
    return 0;
}
```

Output: 1

The program asks the user to input any number to proceed, as shown in Figure 1.43, so enter any number and press enter.

```
Enter the number: 10
sum is:55
```

Figure 1.43: *Output for a recursive algorithm to find the sum of n-consecutive integers in C++.*

Advantages of a recursive algorithm

The following are the advantages of a recursive algorithm:

(a) It reduces the number of lines of code in a program.

(b) It is clear and straight for the programmer to understand.

(c) Recursive algorithms simply solve complex problems.

Disadvantages of a recursive algorithm

The following are the disadvantages of a recursive algorithm:

(a) A recursive algorithm must include condition statements to terminate the recursion, otherwise, the recursion will continue.

FOR ONLINE READING ONLY

CamScanner

(b) Since the recursive algorithm is repetitive, it requires enough processor time to complete the task.

(c) It simplifies a solution to the problem when using a recursive algorithm.

Activity 1.7:

Generating Fibonacci numbers using C++

Aim: To construct the Fibonacci sequence, develop a recursive algorithm, and implement a C++ program that generates Fibonacci numbers based on user input.

Materials: Computer installed with a C++ compiler or online IDEs, textbook for reference, exercise book for notes, pen or pencil.

Instructions

Step 1: On paper, write the first ten Fibonacci numbers in order, starting from 0 and 1.

Step 2: On paper, write a recursive algorithm that generates the Fibonacci number at position n. Include the base case(s) and recursive case(s) as part of the algorithm structure.

Step 3: Create a C++ program named fibonacci.cpp that:

(a) Prompts the user to enter the range (e.g., first *n* Fibonacci numbers).

(b) Uses a recursive function to compute Fibonacci numbers.

(c) Displays the generated Fibonacci sequence on the screen.

Step 4: Compile and run your program using sample inputs such as: n = 5, n = 10, and n = 15

Step 5: Record and submit the outputs displayed by your program.

Deliverables

A report comprising of the following:

(a) The list of the first ten Fibonacci numbers.

(b) A handwritten recursive algorithm for generating Fibonacci numbers.

(c) A working C++ source file (fibonacci.cpp) implementing recursion.

(d) Program outputs for the given sample inputs.

Exercise: 1.4

1. Algorithms and programs serve different roles in problem-solving.
2. Clearly differentiate between an algorithm and a program, and explain how each contributes to solving a computing problem.
3. You need to obtain a pen from a shopkeeper to complete your homework. Design a step-by-step algorithm that you would follow to get the pen. Your algorithm should show decision steps, conditions, and sequence of actions, just like a real computational process.

FOR ONLINE READING ONLY

COMPUTER SCIENCE F6 FINAL (EVA Feb 3.indd 47 03/02/2026 15:48:56
CamScanner

4. Recursive thinking allows a problem to be solved by breaking it into smaller versions of itself.
   (a) Explain, in your own words, what a recursive algorithm is and when it is suitable to use.
   (b) Give two real or computing examples of problems that can be solved using recursion, and justify why recursion fits those problems.
5. A recursive algorithm must have a stopping condition.
   (a) Explain the difference between a base case and a recursive step in a recursive function.
   (b) Describe the practical consequences (such as memory use, program behaviour, infinite loops) that occur when a recursive function does not define a base case

(b) Iterative algorithm

When a recursive algorithm produces many repetitions, it is the best time to use an iterative algorithm. An iterative algorithm involves the repetition of some steps during problem-solving. The iterative algorithm breaks a problem into multiple iterative by adding some features to get the correct solution. It consists of iterative control structures such as For loop, While loop, and Do.... while loop. Any iterative event should have a condition to accomplish. For example, if you want to score a goal, you will kick the ball to the goalkeeper and repeat it more than once until you score the goal. So, achieving the plan will be your condition to stop kicking the ball.

There are two types of iterations: definite iteration, which provides a restricted number of times when instructions should be repeated, and indefinite iteration, where instructions are repeated within the loop until another event stops.

An example of the iterative algorithm is the conversion of binary to decimal numbers as follows:

Step 1: Start the program

Step 2: Read the binary value, n

Step 3: Initialise dec=0, i=0

Step 4: Check whether (n>0)

Step 5: Calculate r = n%10, dec += r * pow(2, i), n=n/10, ++i

Step 6: Print the value of the dec

Step 7: Stop

An example of a C++ program to present the implementation of an iterative algorithm activity is as follows:

FOR ONLINE READING ONLY

COMPUTER SCIENCE F6 FINAL (EVA Feb 3.indd 48 03/02/2026 15:48:56
CamScanner

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
    int n, dec = 0, i = 0, r;

    cout << "Enter any binary number: ";
    cin >> n;

    while (n > 0)
    {
        r = n % 10;
        n /= 10;
        dec += r * pow(2, i);
        ++i;
    }

    cout << "Equivalent Decimal number: " << dec << endl;
    return 0;
}
```

The output of the program is as in Figure 1.44.

```
Enter any binary number: 1001
Equivalent Decimal number: 9
```

Figure 1.44: *Output of the program*

Note:

(a) The iterative algorithm is more efficient than the recursive algorithm in terms of speed of execution and memory usage.

(b) The iterative algorithm can also be used to write any program that can be written using a recursive algorithm

Exercise 1.5

1. (a) Analyse the real-life example where iteration is applied.
   (b) Write an iterative algorithm to express the example.
2. Critically explain why is it beneficial using iterative over recursive algorithms?
3. (a) Design an algorithm to find the sum of the digits made from a positive number.
   (b) Create the C++ program to implement the algorithm.
4. (a) Change the following recursive algorithm to find the reverse of the number in iterative form
   Step 1: Start
   Step 2: Enter the value of n
   Step 3: Assign r = 0, sum = 0
   Step 4: If n>0
   r = n mod 10
   sum = sum *10 + r
   n = n div 10
   Step 5: Repeat step 4 until the condition fail
   Step 6: Print sum
   Step 7: Stop
   (b) Develop a program of the given algorithm to display the output.

FOR ONLINE READING ONLY

CamScanner

(c) Searching algorithm

You have studied how data is organised in different data structures such as an array, pointer, stack, and linked list. The most important operation conducted in a linked list is a search algorithm. The searching for the item within the list, involves the following:

(i) Determine if the data is within the list or not.

(ii) Check if the data are arranged for quickly finding the location of data.

(iii) Determine where the deleted data or item can be found.

(iv) To simplify searching for data or items within the list, each data set should have a uniquely identified key.

For example, if you want to search for details of a particular student from a school database, student ID as a uniquely identified key should be used to search student details. You will compare that key with the targeted records to find the required data.

Generally, the search algorithm is designed to find or retrieve an item from the list of data. There are two basic types of search algorithms, which are:

(i) Linear search

(ii) Binary search

Linear search

Suppose you want to find a particular car number from a list of car numbers. The easiest way is to compare your car number (target) with all numbers within the list by starting from the first number to the last number. Searching is over when you reach the point where the number equals the target. A system of finding data among several elements within the list by comparing each value with the targeted value is known as a linear search. Sometimes the linear search is known as a sequential search. A linear search can be performed either within a sorted or unsorted list. The linear search algorithm derived from the given scenario is:

Step 1: Start

Step 2: Compare each number with the targeted number

Step 3: If the present number matches the target, declare victory and stop

Step 4: If the present number is not equal to the target, move to the next number and repeat step 2.

Step 5: Stop

Example of a C++ program to implement the linear search algorithm activity derived from the given scenario

FOR ONLINE READING ONLY

CamScanner

```
#include<iostream>
using namespace std;

int a[50], item, n, i;

void linearSearch() {
    for (i = 0; i < n; i++) {
        if (item == a[i]) {
            cout << "The item is found at location " << i + 1 << endl;  // Adjusted for 1-based index
            return;
        }
    }
    cout << "Item not found!" << endl;  // If the item isn't found
}

int main() {
    cout << "Enter number of cars within the list: ";
    cin >> n;
    cout << "Enter all numbers within the list:" << endl;
    for (i = 0; i < n; i++) {
        cin >> a[i];
    }

    cout << "Enter number to search: ";
    cin >> item;

    linearSearch();
    return 0;
}
```

The output of the program will be in Figure 1.45:

```
Enter number of cars within the list: 4
Enter all numbers within the list
56
12
35
78
Enter number to search: 12
the item is found at location 2
```

Figure 1.45: *Output of the program*

Advantages of the linear search algorithm

(a) A linear search is performed very fast if a list contains few elements.

(b) Linear search does not need to be sorted; anyhow data arranged, searching is possible.

(c) Any operation within a list, including insertion and deletion of the items, will not affect the linear search.

Disadvantages of the linear search algorithm is in time to complete searching. Generally, the linear search seems to be quietly inefficient. Suppose you have 1000 items, and the searched item is at the last position. Then, you will pass through 999 items without getting the value. However, the linear search can work on any sorted or unsorted data set.

FOR ONLINE READING ONLY

CamScanner

Exercise 1.6

1. Critically analyse how the linear search algorithm sequentially locates elements in a data set, focusing on its efficiency, limitations, and the scenarios in which it may be more suitable than other search methods.
2. Analyse the significance of using a linear search in real-life scenarios, illustrating situations where it is particularly useful despite its simplicity.
3. Consider a list of numbers with their positions as presented in Table 1.2

Table 1.2: *List of numbers and their respective position*

Positions in a list	Value
0	3
1	5
2	2
3	9
4	6
1	5
8	6
7	7

(i) Write an algorithm to search number '2' from the list of the data value.

(ii) Develop a C++ program for the algorithm you have designed in part (i).

Binary search

You have realised that sequential or linear search is inefficient for an extensive list. Therefore, if you want to search data from the list that contains many data, use binary search. The binary search uses the divide and conquers approach to search data within the list. Binary search divides the list into two portions when the list is sorted from the smallest value to the biggest value. After dividing the list into two halves, compare the target with the middle item, then decide which half to take up or down the middle value. For example, suppose you want to search for a number from the phone book, where names are arranged in ascending order. Instead of looking for a number from the first to the last, you can start from the middle and see whether to go up or down depending on the position of a targeted name. The following is the algorithm and C++ program to explain the binary search algorithm.

Algorithm

Step 1: Start

Step 2: Sort the list from small to the large value

Step 3: Find the middle value of the list

Step 4: If the middle value of the list and the targeted value are equal, declare the victory and stop

Step 5: If the middle item is less than the target, then remain with only upper items of the middle value, and from the list and repeat step 2 using the new list.

Step 6: If the middle item is greater than the target, then continue with the lower items list of the middle value, repeat step 2 using the new list.

Step 7: Stop

FOR ONLINE READING ONLY

CamScanner

Example of a C++ program to show the implementation of a binary search algorithm

```
#include <iostream>
using namespace std;

int a[50], item, n, loc, beg, mid, end, i;

void binarySearch() {
    beg = 0;
    end = n - 1;
    mid = (beg + end) / 2;

    while ((beg <= end) && (a[mid] != item)) {
        if (item < a[mid])
            end = mid - 1;
        else
            beg = mid + 1;

        mid = (beg + end) / 2;
    }

    if (item == a[mid])
        cout << "The item is found at location " << mid + 1 << endl;
    else
        cout << "Item does not exist" << endl;
}
int main() {
    cout << "Enter number of items in the list: ";
    cin >> n;

    cout << "Enter all numbers within the list" << endl;
    for (i = 0; i < n; i++) {
        cin >> a[i];
    }

    cout << "Enter number to search: ";
    cin >> item;

    binarySearch();

    return 0;
}
```

The output of the C++ program is shown in Figure 1.46.

```
Enter number of items in the list: 5
Enter all numbers within the list
56
71
87
91
99
Enter number to search: 100
Item does not exist
```

Figure 1.46: *Output of the program*

FOR ONLINE READING ONLY

CamScanner

Advantages of binary search

(a) Binary search narrowed the long distance searching by dividing the list into two halves.

(b) When the list of data is extensive, binary search works more efficiently than linear search.

Disadvantages of binary search

(a) Programming a binary search algorithm is complicated compared to a linear search

(b) It is time-consuming to solve, especially if the binary search has many items.

Activity 1.8:

Implementing linear and binary search in C++

Aim: To collect student marks, store them in a program, and apply linear and binary search techniques in C++ to determine the number of students who passed or failed.

Materials: Computer installed with a C++ compiler or online IDEs.

Instructions

Step 1: Record current marks for computer science subject from all students

Step 2: Set 50% to be a pass mark

Step 3: Write a linear search algorithm to find the number of students who passed the exam

Step 4: Write a binary search algorithm to find the number of students who failed the exam

Step 5: Use C++ programming language to develop a linear and binary search program

Step 6: Compile and run the program

Step 7: Use the program to search the number of students who have average pass marks and above in a class

Step 8: Take the snapshot of the output and save it in a word processor file

Deliverables

A comprehensive report comprising of the following:

(a) C++ source file search_marks.cpp containing:

(i) Linear search implementation

(ii) Sorted list

(iii) Binary search procedure

(iv) Output statements

(b) Screenshot or printout of the program output showing pass and fail counts

Exercise 1.7

1. Explain the necessity of search algorithms in computing, and analyse how they enhance efficiency, ensure accurate results, and optimise data retrieval in practical problem-solving
2. Analyse the advantages of using a binary search algorithm over a sequential (linear) search algorithm when solving problems that involve large datasets.
3. Assume that you are required to develop a program using the search algorithm, binary or linear. Which search is best for you? Give reasons to support your answer.

FOR ONLINE READING ONLY

COMPUTER SCIENCE F6 FINAL (EVA Feb 3.indd 54 02/02/2026 15:48:57

CamScanner

(d) Sorting algorithm

Data should be arranged in ascending or descending order due to different reasons. You can organise data for easy searching, reading, retrieving, or otherwise. For example, a class teacher finds each student's position in a class after placing the students' marks in ascending order. The algorithm responsible for putting data within a list in a required order is known as the sorting algorithm. The sorting algorithm uses a comparison operator or the divide and conquer technique to decide the new order of elements within the data structure.

Types of sorting algorithms

There are various types of sorting algorithms, including selection-sort, insertion-sort, bubble-sort, heap sort, quick-sort, merge sort.

Selection-sort

Selection-sort involves sorting by selecting elements in the list and shifting them to their required position. The algorithm locates the smallest element at the top of the unsorted portion of the list. The selection sort is most applied in an array. Figure 1.47 shows a selection sort algorithm.

FOR ONLINE READING ONLY

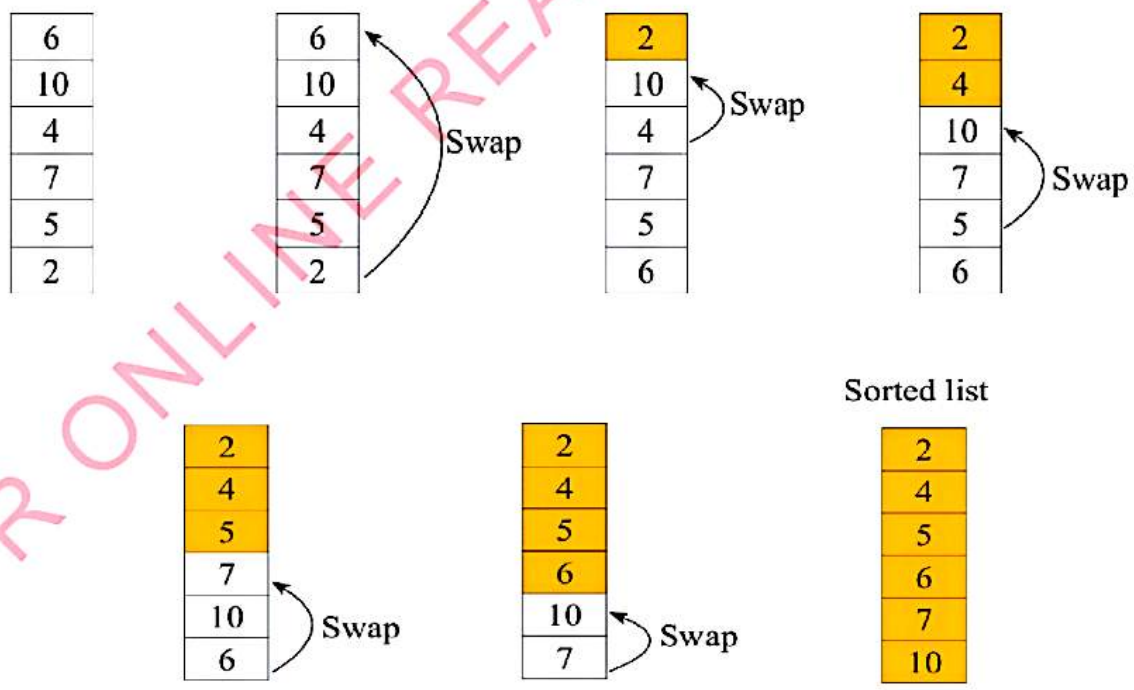


Figure 1.47: *Selection sort*

Consider the following example for implementing the selection sort algorithm in C++.

CamScanner

```
#include <iostream>
using namespace std;

int findSmallest(int[], int);

int main() {
    int school[6] = {20, 8, 16, 4, 1, 10};
    int pos, temp, pass = 0;

    cout << "\nInput list of elements to be Sorted:\n";
    for (int i = 0; i < 6; i++) {
        cout << school[i] << "\t";
    }

    for (int i = 0; i < 6; i++) {
        pos = findSmallest(school, i);
        temp = school[i];
        school[i] = school[pos];
        school[pos] = temp;
        pass++;
    }

    cout << "\nSorted list of elements is:\n";
    for (int i = 0; i < 6; i++) {
        cout << school[i] << "\t";
    }

    cout << "\nNumber of passes required to sort the array: " << pass;

    return 0;
}

int findSmallest(int school[], int i) {
    int ele_small, position, j;
    ele_small = school[i];
    position = i;

    for (j = i + 1; j < 6; j++) {
        if (school[j] < ele_small) {
            ele_small = school[j];
            position = j;
        }
    }
    return position;
}
```

FOR ONLINE READING ONLY

Insertion-sort

Insertion-sort puts an element in its proper place by comparing the value of data and considering the position of a particular data. Insertion-sort divides a list into sorted and unsorted sublists. Then, elements from the unsorted sublist are placed in the correct position in the sorted sublist. Insertion sort can be applied to sort data in linked data, as seen in Figure 1.48.

CamScanner

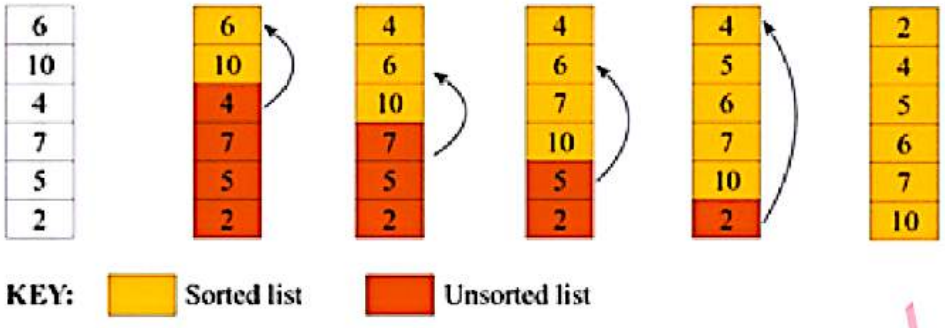


Figure 1.48: *Insertion sort*

The general algorithm for writing insertion sort will be:

Step 1: Start

Step 2: set temp = y [z]

Step 3: set i = z – 1

Step 4: Repeat

while temp <= y [i]

set y [i + 1] = y[i]

set i = i – 1

[end of inner loop]

Set y [i + 1] = temp

[end of loop]

Step 5: Exit

Insertion sort implementation using C++ language. Example of a program using C++ language which to sort the marks of 12 students is as follows.

```
#include <iostream>
using namespace std;

int main() {
    // Array of student marks
    int studentMarks[12] = {20, 45, 28, 24, 91, 55, 73, 80, 58, 99, 49, 45};

    // Display the input list
    cout << "\nInput list is:\n";
    for (int i = 0; i < 12; i++) {
        cout << studentMarks[i] << "\t";
    }

    // Insertion sort algorithm
    for (int k = 1; k < 12; k++) {
        int temp = studentMarks[k];
        int j = k - 1;

        while (j >= 0 && temp <= studentMarks[j]) {
            studentMarks[j + 1] = studentMarks[j];
            j = j - 1;
        }
        studentMarks[j + 1] = temp;
    }

    // Display the sorted list
    cout << "\nSorted list is:\n";
    for (int i = 0; i < 12; i++) {
        cout << studentMarks[i] << "\t";
    }

    return 0;
}
```

FOR ONLINE READING ONLY

CamScanner

The output of the program is shown in Figure 1.49.

```
Input list is
20      45      28      24      91      55      73      80      58      99      49      45
Sorted list is
20      24      28      45      45      49      55      58      73      80      91      99
```

Figure 1.49: *Output for insertion sort implementation using C++*

Bubble sort

Bubble sort is the comparison-based type of sorting algorithm. The bubble sort algorithm sorts data by comparing and swapping elements if they are not in the required order. It continues with the swapping of data until all data is sorted. For the data, which is arranged horizontally, such as data in an array, bubble sort starts by comparing the first two elements and then swapping them if they are not arranged correctly, then arranging the second and third elements. Bubble sort is a stable sorting algorithm but inefficient for large data volumes. Figure 1.50 shows an example of the bubble sort algorithm.

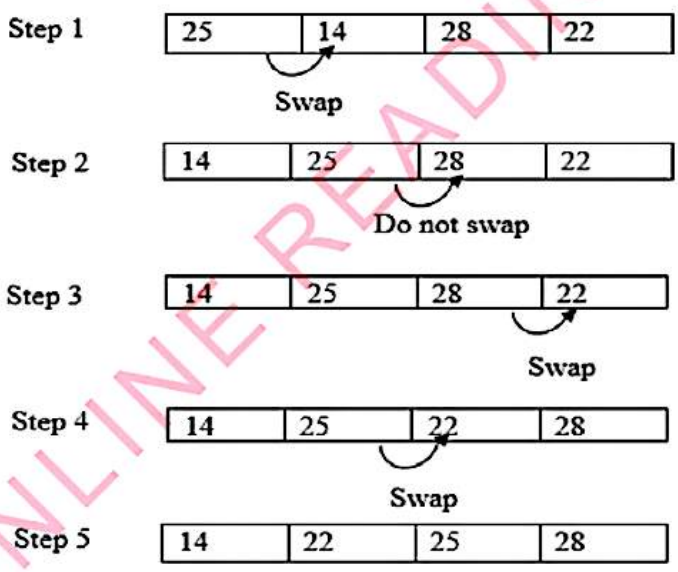


Figure 1.50: *Bubble sort algorithm*

Implementation of bubble sort

The following is algorithms and C++ program to implement the bubble sort algorithm

Algorithm

Step 1: Start

Step 2: Compare the first and second element

Step 3: If the first element is greater than the second element of an array, swap them

Step 4: If the first element is less than the second element of an array, move to the next elements

Step 5: Repeat step 1 until all data is sorted within the list

Example of a C++ program for bubble sort algorithm implementation:

FOR ONLINE READING ONLY

CamScanner

```
#include <iostream>
using namespace std;

int main() {
    int n, i, arr[50], j, temp;

    // Input size of array
    cout << "Enter the size of array: ";
    cin >> n;

    // Input the array elements
    cout << "Enter " << n << " numbers:" << endl;
    for (i = 0; i < n; i++) {
        cin >> arr[i];
    }

    // Bubble sort algorithm
    for (i = 0; i < (n - 1); i++) {
        for (j = 0; j < (n - i - 1); j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap elements if they are in the wrong order
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }

    // Output the sorted array
    cout << "\nThe sorted array is: \n";
    for (i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

FOR ONLINE READING ONLY

The output of the C++ program to implement bubble sort is as shown in Figure 1.51.

```
Enter the size of array: 5
Enter 5 Numbers
45
26
89
12
50

The sorted array is:
12 26 45 50 89
```

Figure 1.51: *Output of the program*

CamScanner

Example 7: Sort (5, 1, 12, 6, 23) using the bubble sort algorithm solution:

Step 1: (5, 1, 12, 6, 23) → (1, 5, 12, 6,23)

Step 2: (1, 5, 12, 6, 23) → (1, 5, 12, 6,23)

Step 3: (1, 5, 12, 6, 23) → (1, 5, 6, 12,23)

Hence, all elements are sorted.

Advantages of Bubble sort algorithm

1. Simple and easy to understand: It is one of the most basic sorting algorithms, suitable for teaching beginners.
2. Easy to implement: The algorithm requires only a few lines of code.
3. Works well for small datasets: Performance is acceptable when the input size is very small.
4. In-place sorting: It requires no additional memory, as sorting happens within the original array.
5. Can detect already sorted lists: With an optimized version (using a swap flag), the algorithm stops early if no swaps occur.

Limitations of Bubble sort

1. Very slow for large datasets: Its time complexity ($O(n^2)$) makes it impractical for big lists.
2. Inefficient compared to other algorithms: Algorithms like Quick sort, Merge sort, and Heap sort perform much faster.
3. Too many comparisons and swaps: It repeatedly processes the list even when nearly sorted.
4. Not suitable for real-world applications: Rarely used in production systems due to poor efficiency.
5. Unstable performance: Small changes in input size greatly affect execution time because of its quadratic behavior.

Heap sort

Heap sort is a sorting method based on comparing elements. It is like a selection sort where you first find the smallest element in the list and position it at the beginning of the list. This process should be repeated to all elements within the list. Unlike selection sort, heapsort does not use much time with a horizontal arrangement of the unsorted element; instead, it has a way of quickly sorting the largest elements.

Figure 1.52 shows an example of the heapsort algorithm.

FOR ONLINE READING ONLY

CamScanner

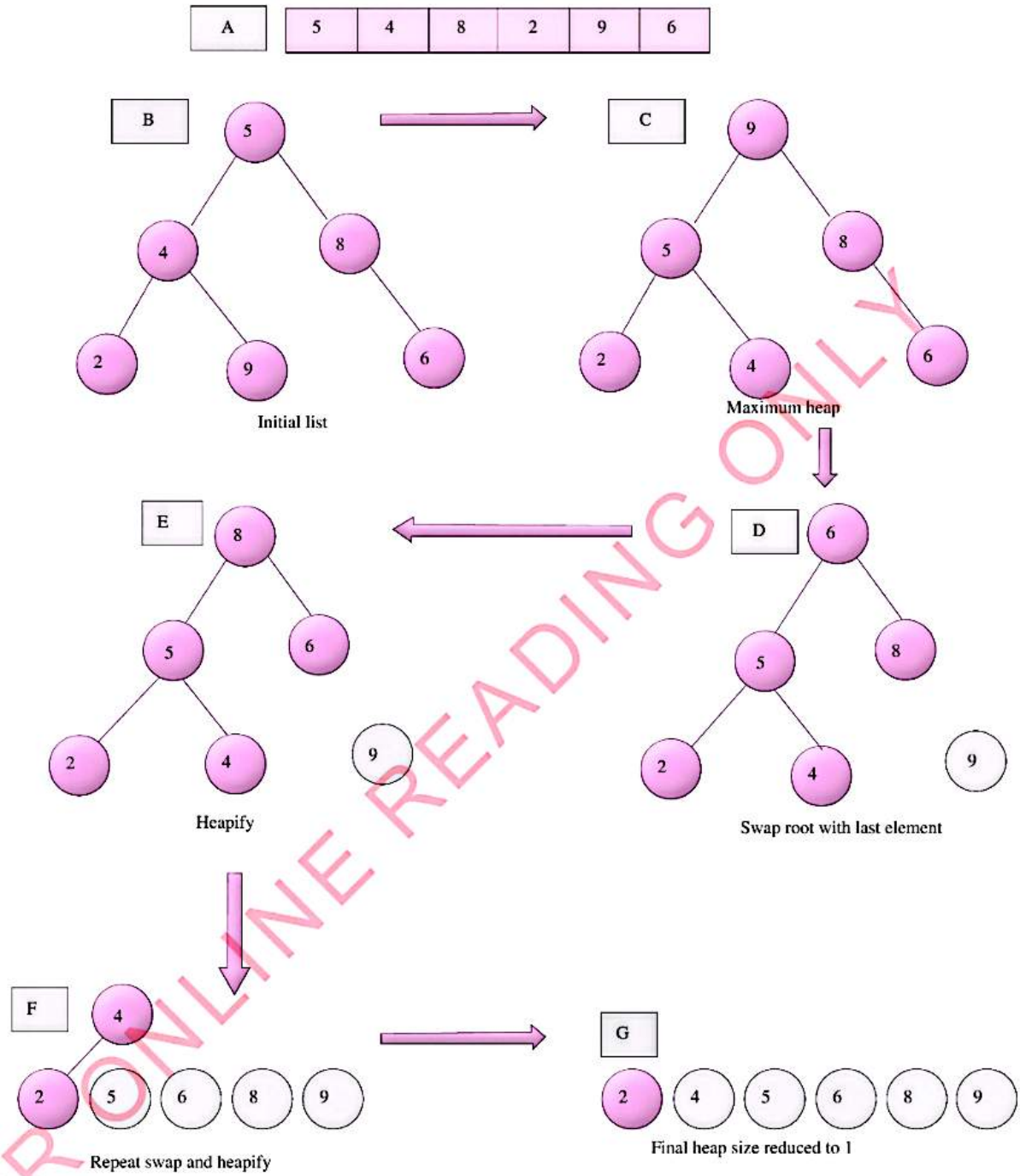


Figure 1.52: *Heapsort*

Using the diagrams in Figure 1.52, you can now generalise writing an algorithm for the heap-sort technique as follows:

Step 1: Start

Step 2: From the given data, build a max heap from the given data such that the root is the highest element of the heap.

Step 3: Remove the root, that is the highest element from the heap, and swap it with the last element of the heap.

Step 4: Then, adjust the max heap not to violate the max heap properties (heapify).

Step 5: The above step reduces the heap size by 1.

FOR ONLINE READING ONLY

COMPUTER SCIENCE F6 FINAL (EVA Feb 3.indd 61 03/02/2026 15:48:58
CamScanner

Step 6: Repeat the above three steps until the heap size is reduced to 1.

Step 7: Stop.

Remember, to sort the given dataset in increasing order, you first construct a maximum heap for the given data as shown in the previous heap-sort algorithm.

Example: The following code demonstrates the implementation of heap-sort using C++.

```
#include <iostream>
using namespace std;

// Function to maintain the heap property
void Load(int arr[], int n, int root) {
    int largest = root;
    int l = 2 * root + 1;
    int r = 2 * root + 2;

    // Check if left child exists and is greater than root
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // Check if right child exists and is greater than root
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // If the largest is not root, swap and continue heapifying
    if (largest != root) {
        swap(arr[root], arr[largest]);
        Load(arr, n, largest); // Recursively heapify the affected subtree
    }
}

// Function to perform heapsort
void loadSort(int arr[], int n) {
    // Build a max heap
    for (int i = n / 2 - 1; i >= 0; i--)
        Load(arr, n, i);
    // One by one extract elements from heap and heapify the reduced heap
    for (int i = n - 1; i >= 0; i--) {
        // Swap the root (maximum element) with the last element
        swap(arr[0], arr[i]);

        // Call Load on the reduced heap
        Load(arr, i, 0);
    }
}

// Function to display the array
void displayArray(int arr[], int n) {
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";
    cout << "\n";
}

int main() {
    int heap_arr[] = {11, 9, 6, 13, 2, 8};
    int n = sizeof(heap_arr) / sizeof(heap_arr[0]);

    cout << "Input array:\n";
    displayArray(heap_arr, n);

    loadSort(heap_arr, n);

    cout << "Sorted array:\n";
    displayArray(heap_arr, n);

    return 0;
}
```

The output of the heap-sort program is as shown in Figure 1.53:

```
Inputed array
11 9 6 13 2 8
Sorted array
2 6 8 9 11 13
```

Figure 1.53: *The output of the heap-sort program*

FOR ONLINE READING ONLY

CamScanner

Advantages of Heap sort

1. Efficient time complexity: Always runs in O(n log n) time, regardless of input order.
2. In-place sorting: Requires only constant extra memory (O(1)).
3. Suitable for large datasets: Performs reliably and consistently even with big lists.

Limitations of Heap Sort

1. Not a stable sort: Equal elements may not keep their original order.
2. More complex to implement: Requires understanding of heap structures and index relationships.
3. Less cache-friendly: Frequent memory jumps make it slower in practice compared to algorithms like Quick Sort.

Quick sort

Quick sort uses the divide and conquer sorting technique. The list is portioned into two sublists, and each will be sorted and joined to form one list again. It uses the recursion method to implement sorting. Quick sort divides a list into lower and upper sublists separated by a pivot. A pivot is a selected element from the list that is more significant than all elements in a lower sublist and less significant in an upper sublist. Quick sort can be applied to sort data within an array. Figure 1.54 shows an example of the quick sort algorithm.

FOR ONLINE READING ONLY

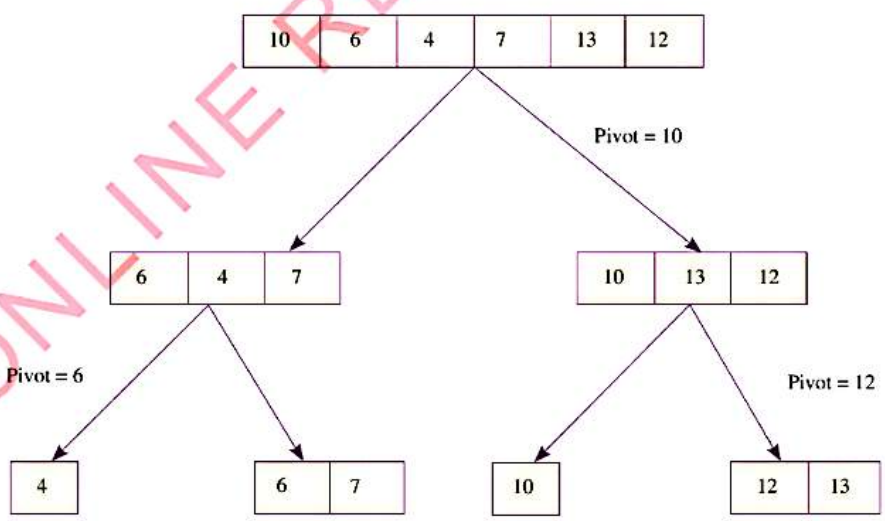


Figure 1.54: *Quick sort*

Advantages of Quick sort

1. Very fast in practice: Quick Sort often outperforms other O(n log n) algorithms due to good cache performance and fewer comparisons.
2. Efficient for large datasets: Works well even with big inputs when the pivot is chosen wisely.
3. Divide-and-conquer approach: Easy to adapt for parallel processing and efficient recursion.

CamScanner

Limitations of Quick Sort

1. Worst-case performance is $O(n^2)$: Happens when poor pivots are selected, especially with already sorted or nearly sorted data.
2. Not stable: Equal elements may change their original order after sorting.
3. Requires additional stack memory: Recursive calls increase memory usage, especially for large lists.

Merge Sort

Merge sort is also a divide and conquer sorting algorithm. It divides the list into two halves and then merges the two sorted lists. Merge sort starts by dividing the list into sublists and continues dividing sublists into two more halves while swapping the elements, then combining swapped elements to form a completely sorted list. Figure 1.55 shows an example of the merge sort algorithm.

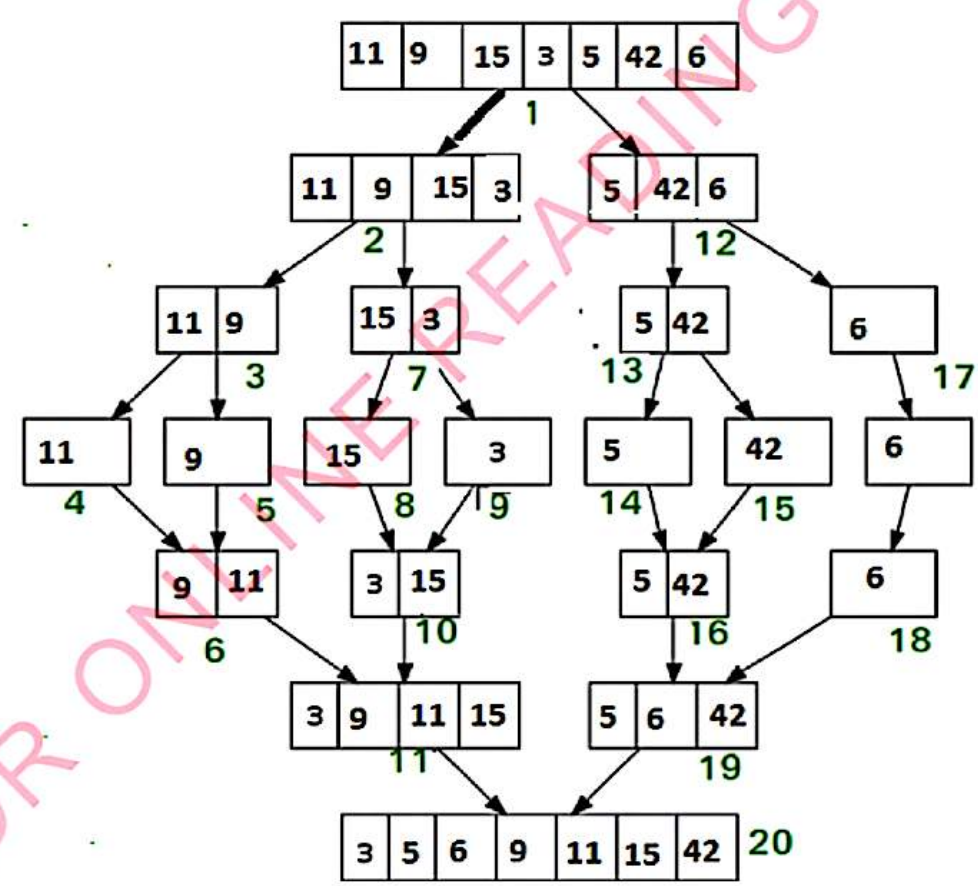


Figure 1.55: *Merge sort*

Advantages of Merge sort

1. Consistent time complexity: Always runs in O(n log n) time, regardless of how sorted the input is.
2. Stable sorting algorithm: Equal elements maintain their original order.
3. Works well with large datasets: Especially effective for linked lists and external sorting such as large files.

FOR ONLINE READING ONLY

CamScanner

Limitations of Merge sort

1. Requires extra memory: Needs additional space (O(n)) for merging, unlike in-place algorithms.
2. Slower in practice for small datasets: Performs more operations compared to algorithms like Quick Sort or Insertion Sort.
3. Usually not as cache-friendly: Accessing divided subarrays may reduce CPU cache efficiency.

The following are things to consider when choosing the suitable sorting algorithms for the application:

Stability of the sorting algorithms

The stability of a sorting algorithm is considered by how it treats identical elements. Stable sorting algorithms maintain the order of equal or repeated elements, which is impossible for unstable sorting. This is to say, stable sorting preserves the order of two similar elements within the list. Suppose you want to sort the students according to their grades after being sorted based on their names; you must use a stable sort algorithm such as merge and insertion sort to sort similar grades without affecting their names. However, it is possible to make unstable sorts like heap and quick sort stable.

Execution time

This explains a running time of an algorithm that can be set for data processing. It explains the average performance of the algorithm on every set of data. Usually, execution time depends on recursive calls, memory usage, and comparisons. Quicksort and merge sort are examples of fast sorting algorithms.

Memory usage

Memory usage depends on the size of the input data to be sorted. An example of a sorting algorithm that does not need extra memory is the insertion and bubble sort. Otherwise, merge sort and quick sort requires more memory to complete sorting.

Note: Depending on some factors, every sorting algorithm can be the best or worst. These factors are based on the problem to be solved and the data to be manipulated.

Activity 1.9:

Working with sorting algorithms

Aim: To implement basic sorting algorithms using C++ and manually execute the steps of advanced sorting algorithms to develop practical competence in sorting techniques.

Materials: Computer installed with a C++ compiler or online IDEs.

Instructions

Step 1: Create a C++ program named selection_sort.cpp.

(a) Create an array with at least 10 unsorted integers.

(b) Implement selection sort to arrange the array in ascending order.

(c) Display the list before and after sorting.

Step 2: Create a C++ program named insertion_sort.cpp.

(a) Use an array with at least 10 unsorted integers.

FOR ONLINE READING ONLY

CamScanner

(b) Implement insertion sort.

(c) Display the list before and after sorting.

Step 3: Create a C++ program named bubble_sort.cpp.

(a) Store at least 10 unsorted integers in the array.

(b) Implement bubble sort.

(c) Display the list before and after sorting.

Step 4: Create a C++ program named heap_sort.cpp.

(a) Store at least 10 unsorted integers in an array.

(b) Implement heap sort (max-heap version):

(i) Build the heap

(ii) Perform heapify

(iii) Extract the maximum repeatedly

(c) Display the list before and after sorting.

Step 5: On paper, write the following list: 28, 12, 5, 20, 17, 30, 9

(a) Use 28 as the pivot.

(b) Draw the list after completing one full partition step.

(c) Choose either the left or right sub-list and draw one more partition step. (Stop here, no full recursion needed.)

Step 6: On paper, write the following list: 32, 15, 7, 20, 5, 18

(a) Draw the split phase until each element stands alone.

(b) Perform the merge phase, showing:

(i) Two-element merges

(ii) Four-element merges

(iii) Final sorted list

(c) Draw each merge step clearly.

Deliverables

A report comprising of the following:

(a) C++ Programs and Outputs

(i) selection_sort.cpp and output screenshot

(ii) insertion_sort.cpp and output screenshot

(iii) bubble_sort.cpp and output screenshot

(iv) heap_sort.cpp and output screenshot

(b) Paper-based tasks

(i) Quick sort partition diagrams

(ii) Merge sort split and merge diagrams

Exercise 1.8

1. Critically analyse importance of sorting during data processing, particularly when handling large datasets. Provide examples of scenarios where sorting significantly improves efficiency and accuracy.
2. Evaluate the key factors that differentiate one sorting algorithm from another, and and how these factors influence the choice of algorithm in practical programming scenarios
3. Analyse how selection sort determines which element to place in the next sorted position?

FOR ONLINE READING ONLY

CamScanner

4. Justify the situation in which insertion sort perform more efficiently than alternative sorting algorithms in practical scenarios.
5. Analyse the condition that causes bubble sort to stop early during sorting.
6. Explain how the heap structure enables heap sort to efficiently locate the next largest or smallest element
7. Assess the role of the pivot in quick sort.
8. Examine why merge sort require additional memory during the merging process?

Chapter Summary

Reflect on what you have learnt from this chapter, summarize and keep it in your portfolio.

Revision Exercise 1

1. By using illustrations, describe operations done in the stack data structure.
2. Analyse with examples four operations that the queue can perform.
3. Sort the numbers: 55, 44, 83, 26, 47, 66, 92 and 72 in a binary tree structure and arrange them in PreOrder, In-order, and Postorder.
4. Explain three (3) features of a suitable algorithm.
5. Design an algorithm for a program that inputs the radius of a circle from a user. The user will be asked by the program to calculate diameter, circumference, or area. The result will be calculated and then displayed.
6. Write an algorithm to find squares of the number.
7. (a) Write an algorithm to find the length of a password.

   (b) Write a C++ program to implement the algorithm made in part (a).
8. (a) Describe the linear search.

   (b) Explain the steps to implement linear search.
9. (a) Analyse how binary search efficiently locates an element in a sorted dataset and outperforms other search methods in terms of speed and resource usage.

   (b) The following form an array: 10, 31, 0, 9, 38, 20, 13, 24, 7 and 200. Explain why it is impossible to locate key "0" using binary search.
10. (a) By showing steps clearly, show how binary search is implemented to find an element in the following Array, and let the searched key be 56 as shown in Table 1.3.

FOR ONLINE READING ONLY

CamScanner

Table 1.3: *Elements of an array*

Elements	Index
10	0
12	1
24	2
29	3
37	4
40	5
51	6
56	7
68	8

(b) Use a C++ program to implement the given algorithm.

11. (a) Analyse the bubble sort algorithm, including its principle, characteristics, and scenarios where it is most appropriate.

(b) Explain how bubble sort is implemented, highlighting each step of the process and how it affects the algorithm's efficiency in practical applications. Read the following C++ code and answer the questions that follows:

```
#include <iostream>
using namespace std;

int main() {
    int a[5] = {2, 5, 4, 1, 3}, i, k, x;

    // Sorting the array in descending order
    for (i = 0; i < 4; i++) {
        for (k = 1 + i; k < 5; k++) {
            if (a[i] < a[k]) {
                // Swap elements
                x = a[i];
                a[i] = a[k];
                a[k] = x;
            }
        }
    }

    // Display the sorted array
    cout << "Sorted array in descending order: ";
    for (i = 0; i < 5; i++) {
        cout << a[i] << " ";
    }
    cout << endl;

    return 0;
}
```

FOR ONLINE READING ONLY

CamScanner

(a) How many elements are included in an array?

(b) Analyse type of sorting algorithm is implemented on the given code?

(c) In which order the array "a" will be sorted?

(d) Which variable will be used temporarily in the given code? Justify your answer.

(e) Write a complete C++ program to display the sorted array "a."



Project: *Design and implementation of the Ketox Service Rendering application system*

Project description

Ketox is a service rendering application system. The application aims to create a bridge between service providers and clients to ease the accessibility of business-related services. The application has two sides, one for the service provider and another for the clients. Each side defines business privileges to the particular side.The client-side has the following privileges:

(a) Search for a service provider and order the service.

(b) Get all the necessary information about any service provider.

(c) Pay for any ordered service through any mobile money service application.

(d) Rate and comment upon the services provider.

The service provider has the following privileges:

(a) Receive requests from clients.

(b) Get a short portfolio of all clients who need service.

(c) Approve payment soon after receiving the message of the money sent by the customer from mobile money application services.

(d) Rate and comment upon the clients.Design the system's algorithm, then use it to implement the system using a C++ programming language.

Project objectives

1. To Analyse and model the functional requirements of a two-sided service-rendering application.
2. To design a detailed algorithm representing the workflow of both client and service-provider interactions.
3. To develop a functional C++ program implementing the designed algorithm.
4. To demonstrate data handling, user interaction, and process flow in a real-world service system context.

FOR ONLINE READING ONLY

CamScanner

Project tasks

1. Identify and document all user privileges for both the client and service-provider sides.
2. Design a comprehensive system algorithm covering:
   (a) Client activities (searching, ordering, paying, rating).
   (b) Service-provider activities (receiving requests, viewing client details, approving payments, rating).
1. Create appropriate data structures to store provider information, client details, service requests, and ratings.
2. Implement the system algorithm using C++ (input/output simulation acceptable).
3. Test all functionalities to ensure correct flow between clients and service providers.
4. Document the implementation process, challenges, and system limitations.

Project deliverables

(a) A written system algorithm.

(b) A complete C++ program implementing the Ketox system.

(c) A project report containing:
   (i) Introduction and system overview
   (ii) Algorithm design
   (iii) Implementation details
   (iv) Sample inputs and outputs
   (v) System limitations and recommendations

FOR ONLINE READING ONLY

COMPUTER SCIENCE F6 FINAL (EVA Feb 3.indd 70 03/02/2026 15:49:01
CamScanner