Unit 1: Introduction to Data Structures

Learning Outcomes:

- Students will be able to understand the classification of data structures and their respective operations.
- Students will be able to analyze the time and space complexity of algorithms using Big O notation.
- Students will be able to comprehend abstract data types and their role in software development.
- Students will be able to proficient in implementing basic operations on arrays and strings in C++.
- Students will be able to develop the ability to implement and analyze common sorting and searching algorithms in C++.

Structure:

1.1. Introduction to Data Structures

1.1.1. Classification of Data Structures

1.1.2. Complexity of Algorithms

1.1.3. Abstract Data Types

1.1.4. Arrays

1.1.5. Representation of Arrays in Memory

1.1.6. Operations on Array

1.1.7. Strings and its Representation in Memory

1.1.8. Operations on Strings

1.1.9. Pointers

1.1Introduction to Data Structures

In the realm of computing, a data structure refers to a specific method for organizing and storing data in a computer's memory, facilitating efficient retrieval and utilization of said data at a later time. The arrangement of data can take various forms, with each conforming to a logical or mathematical model known as

a data structure. The effectiveness of a particular data model hinges on two key factors:

1. Reflecting Real-World Relationships: The structure must be sufficiently rich to accurately represent the relationships between the data and real-world objects.
2. Simplicity for Processing: Simultaneously, the structure should maintain simplicity to ensure efficient data processing whenever required.

1.1.1 Classification of Data Structures

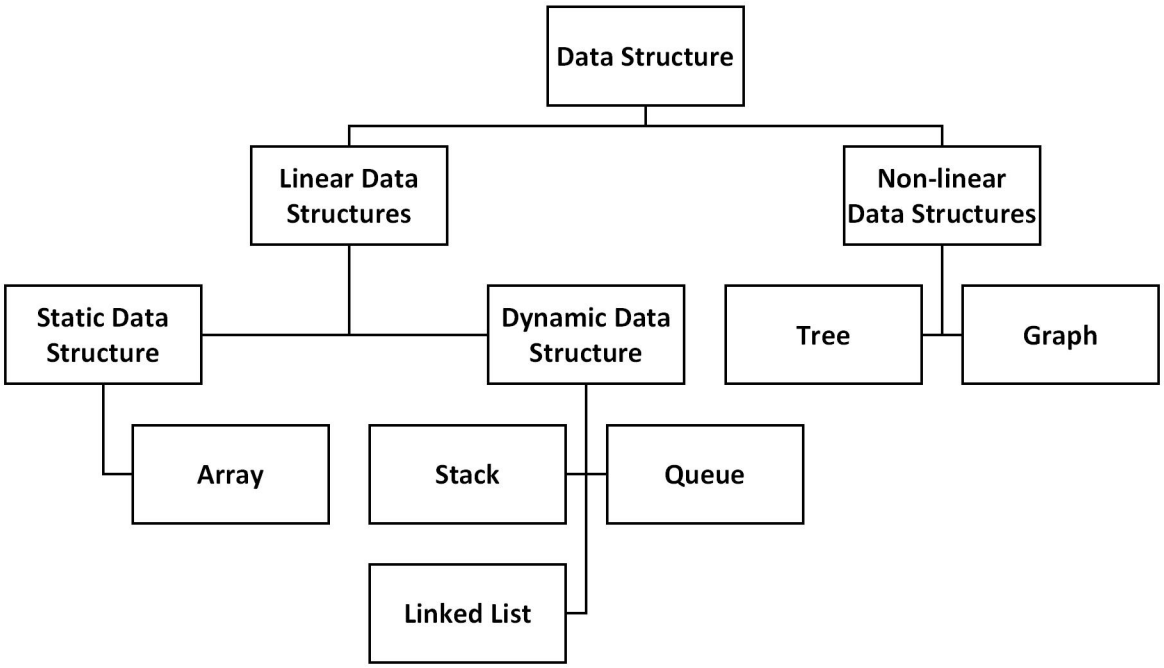


Figure 1.1: Classification of Data Structures

Data structures are broadly categorized into two main types:

1. Linear Data Structures: These structures organize elements in a linear fashion without any inherent hierarchy. They are typically represented in memory using two primary techniques:

Arrays: Linear relationships among elements are established through contiguous memory locations.

Linked Lists: Linear relationships among elements are established using pointers or links.

Common examples of linear data structures include arrays, queues, stacks, and linked lists.

Static Data Structure:

Static data structures are characterized by a fixed memory size, making it straightforward to access elements within them. An array is a prime example of a static data structure.

Dynamic Data Structure:

On the other hand, dynamic data structures have variable sizes that can be altered during runtime, offering flexibility and potentially more efficient memory usage. Examples of dynamic data structures include queues, stacks, and others.

2. Non-linear Data Structures: This category is employed to represent data with hierarchical relationships among its elements. Examples include:

Trees: Reflect hierarchical relationships among elements, often termed as rooted tree graphs.

Graphs: Represent relationships between pairs of elements, not necessarily following a hierarchical structure.

Non-linear data structures serve to capture complex relationships among data elements, providing a flexible framework for organizing diverse datasets.

1.1.2 Complexity of Algorithms

Algorithm

An algorithm is a step-by-step procedure or set of instructions designed to solve a specific problem or perform a particular task. It acts as a blueprint for solving problems computationally, providing a clear sequence of actions that must be followed to achieve the desired outcome. Algorithms are fundamental to computer science and programming, serving as the foundation for developing software and solving complex computational problems.

Advantages of Algorithms:

1. Precision: Algorithms provide precise, unambiguous instructions for solving problems, ensuring consistency and accuracy in their execution.
2. Efficiency: Well-designed algorithms are crafted to optimize performance, aiming to achieve the desired outcome using the least amount of computational resources, such as time and memory.
3. Reusability: Algorithms are modular, allowing for their reuse across different applications and scenarios, promoting code efficiency and reducing redundancy.
4. Scalability: Algorithms can be scaled to handle varying input sizes and complexities, making them adaptable to different problem instances.

Characteristics of Algorithms:

1. Finiteness: To prevent running endlessly, an algorithm must end after a finite number of steps.
2. Input: The data or parameters required for an algorithm to operate are represented by zero or more inputs.
3. Output: The answers or solutions to the given problem are the one or more outputs that an algorithm generates.
4. Determinism: Every step in the algorithm needs to be well-defined and predictable, which means that each time it is run, it should yield the same result for a given input.
5. Effectiveness: An algorithm needs to be able to solve the problem for which it was created in a fair amount of time in order to be considered effective.

Example:

Algorithmic example to find the sum of numbers from 1 to a given input number without using an array:

Algorithm: Sum of Numbers

1. Input: Receive a positive integer n as input.
2. Initialize: Set a variable sum to zero, which will accumulate the sum of numbers.
3. Iterate Through Numbers: - Begin by putting 1 through n in a loop.
   - Increase the sum variable by the current amount at each iteration.
4. Output: Once all numbers are processed, the variable sum holds the total sum of numbers from 1 to n.

Pseudocode:

```
function findSum(n):
   // Step 2
   sum = 0
      // Step 3
   for i from 1 to n:
      // Step 3a
      sum = sum + i
   // Step 4
   return sum
```

Real-life Examples of Algorithm Use

1. Search Engine Algorithms: To crawl and index web pages and to rank search results according to relevancy and other criteria, search engines such as Google employ sophisticated algorithms. These algorithms allow users to

rapidly and effectively sift through the massive amounts of online data to identify pertinent information.

2. GPS Navigation Algorithms: GPS navigation systems use algorithms to determine the fastest or shortest path between two sites while accounting for user preferences, traffic, and road closures. These algorithms make sure that consumers can travel as quickly and efficiently as possible to their destinations.
3. Social Media Algorithms: Users' feeds on social media sites like Facebook and Instagram are personalized by these algorithms, which are based on their interactions, browsing history, and interests. Large volumes of data are analyzed by these algorithms to provide consumers with relevant content that improves their overall platform experience.
4. Algorithms for Online Shopping Recommendations: Online retailers such as Amazon employ algorithms to suggest things to users based on their past browsing and purchasing activity, as well as the actions of other users who are similar to them. By recommending items that are likely to be of interest to customers, these algorithms contribute to higher levels of customer satisfaction.
5. Algorithms for Image and Speech Recognition: In order to evaluate and comprehend visual and aural input, image and speech recognition systems rely on algorithms. These algorithms improve user experiences across a range of sectors by enabling applications like voice assistants, automated image labelling, and facial recognition.

Types of Algorithms

1. Searching techniques: These techniques are employed to find particular objects in a dataset. Binary and linear searches are two examples.
2. Sorting Algorithms: Algorithms for sorting put items in a particular order, such as numerical or alphabetical. Quicksort, bubble sort, and merge sort are a few examples.
3. Graph Algorithms: To tackle issues like determining the shortest path or identifying cycles, graph algorithms work on graphs made up of nodes and edges.
4. Dynamic Programming Algorithms: These algorithms save duplication of computations by decomposing complicated issues into smaller subproblems and storing the answers to these subproblems.
5. Greedy Algorithms: In an attempt to reach a global optimum, greedy algorithms select the options that are locally optimal at each stage. One example is the Dijkstra algorithm, which determines the shortest.

Algorithm complexity

Algorithm complexity refers to the resources, such as time and memory, needed to solve a problem or execute a task. It is commonly measured through time complexity, which indicates the time an algorithm takes to complete relative to the input size, and memory complexity, which measures the memory used by the algorithm. Reducing algorithm complexity is crucial for efficiency and scalability.

Time complexity

Time complexity assesses how long an algorithm takes to solve a task; the number of iterations, comparisons, or other operations frequently calculates this. It is a function describing the algorithm's time consumption in relation to the input size.

For instance, frequent searches may benefit from algorithms like binary search trees due to their efficient time complexity.

Key points regarding time complexity:

Definition: Time complexity refers to the computational resources, particularly time, required by an algorithm to solve a specific problem. It provides an estimate of how the runtime of an algorithm grows with increasing input size.

Notation: Time complexity is typically expressed using Big O notation (O notation), which provides an upper bound on the growth rate of the algorithm's running time as the input size increases. Other notations like Big Omega (Ω) and Big Theta (Θ) are also used for different analysis purposes.

Factors Affecting Time Complexity:

Input Size: Time complexity is analyzed based on the input data size. As the input size increases, the time taken by the algorithm may also increase, but at different rates depending on the algorithm's complexity.

Hardware and Environment: The actual runtime of an algorithm may vary based on the hardware on which it is executed and the environment in which it operates. However, time complexity analysis focuses on the theoretical performance of the algorithm, independent of specific hardware or environment factors.

Types of Time Complexity:

Constant Time (O(1)): Regardless of the size of the input, algorithms with constant time complexity always require the same amount of time to run. Examples include using an array's index to retrieve a particular element or carrying out simple arithmetic operations.

Linear Time (O(n)): The runtime of algorithms with linear time complexity is a function of the input data size. The algorithm's runtime grows linearly with the size of the input. Simple array iteration and linear search are two examples.

The runtime of algorithms with logarithmic time complexity increases logarithmically with the size of the input data. This is known as O(log n) time. These methods are frequently more efficient for high input sizes than linear time techniques. Binary search and some divide-and-conquer strategies are two examples.

Higher Polynomial Time, Cubic Time, and Quadratic Time: Algorithms with polynomial time complexity have runtimes that increase polynomially with the amount of the input. The inefficiency of these algorithms increases with larger input sizes. Some sorting algorithms, such as bubble sort (O(n^2)), and specific nested loop algorithms, are examples.

Exponential Time (O(2^n)) and Factorial Time (O(n!)): The runtimes of algorithms with factorial or exponential time complexity increase factorially or exponentially with the size of the input, respectively. Large input sizes are typically unfeasible for these algorithms due to their extreme inefficiency.

Analyzing Time Complexity:

Counting Operations: Counting the number of fundamental operations (such as comparisons, assignments, and arithmetic operations) carried out by the algorithm as a function of the input size is a common method of analyzing temporal complexity.

Time complexity study focuses on the algorithm's asymptotic behaviour, taking into account how its runtime scales with huge input sizes. It concentrates on the

dominating term that controls the pace of increase in the algorithm's runtime, ignoring constant factors and lower-order terms.

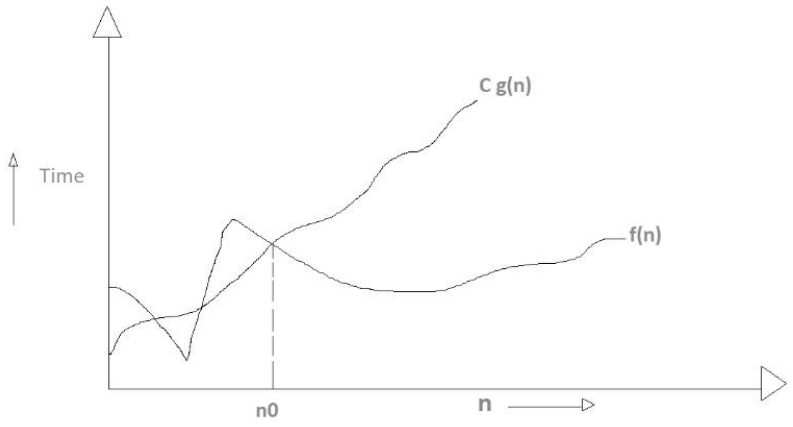
Best Case, Worst Case, and Average Case Time Complexity:

1. Best Case Time Complexity :

In particular, understanding time complexity's upper bounds is essential to algorithm analysis. An algorithm's upper bound is the longest duration it can run for, indicating its worst-case efficiency. This asymptotic upper bound is expressed mathematically using Big O notation, which is widely used.

When we write an algorithm's running time as f(n), we say that it is O(g(n)) if, for every n >= n0, there exist positive constants C and n0 such that 0 \= f(n) \= Cg(n). This notation indicates that when the input size rises, the algorithm's execution time increases no more quickly than a particular function g(n).

A method with an O(n) time complexity, for example, inevitably implies its O(n^2) time complexity because every function that is O(n) is therefore O(n^2). We can more effectively express an algorithm's upper bound thanks to this feature.



In graphical terms, the concept of Big O can be illustrated using a graph where the x-axis represents the input size (n) and the y-axis represents the running time (f(n))

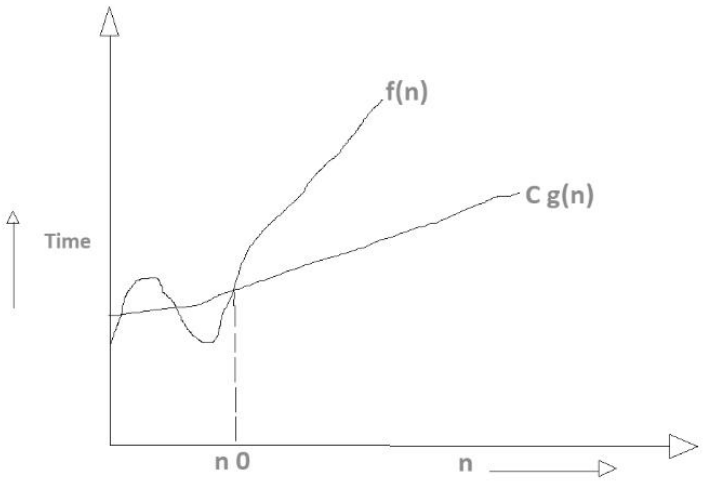
of the algorithm. The Big O notation signifies an upper limit on the growth rate of the algorithm's running time as the input size increases. This provides a visual representation of how the algorithm's performance scales with varying input sizes and facilitates comparisons between different algorithms.

2. Worst Case Time Complexity:

Time complexity also involves understanding lower bounds, which represent the shortest amount of time that an algorithm needs, indicating its most efficient performance or best-case scenario. Ω (Omega) notation is employed to express asymptotic lower bounds in algorithm analysis, just as Big O notation signifies asymptotic upper bounds.

When we specify the running time of an algorithm as f(n), we write Ω(g(n)) if, for every n >= n0, there exist positive constants C and n0 such that 0 \= Cg(n) \= f(n). This notation suggests that as the input size rises, the algorithm's execution time is at least as quick as a particular function g(n).

Because any function that is Ω(n^2) is also Ω(n), for example, an algorithm with a time complexity of Ω(n^2) obviously implies that its time complexity is Ω(n). Thanks to this characteristic, we can more effectively express an algorithm's bottom bound.



Visually, Ω notation can be demonstrated with a graph in which the input size is represented by the x-axis (n) and the y-axis represents the running time (f(n)) of the algorithm. Ω notation signifies a lower limit on the growth rate of the algorithm's running time as the input size increases, providing a visual representation of the algorithm's most efficient performance. This facilitates comparisons between different algorithms and helps in understanding their behaviour under different scenarios.

3. Average Case Time Complexity:

The tightest bound, known as Big Theta (Θ), represents the most accurate estimation of an algorithm's performance across all scenarios, encompassing both the best and worst case times it can take.

If we define the running time of an algorithm as f(n), then in mathematics, if f(n) is both O(g(n)) and Ω(g(n)), then the running time is represented as Θ(g(n)). This shows that as the input size increases, the algorithm's execution time is constrained by the same function g(n) both above and below.

Mathematically,

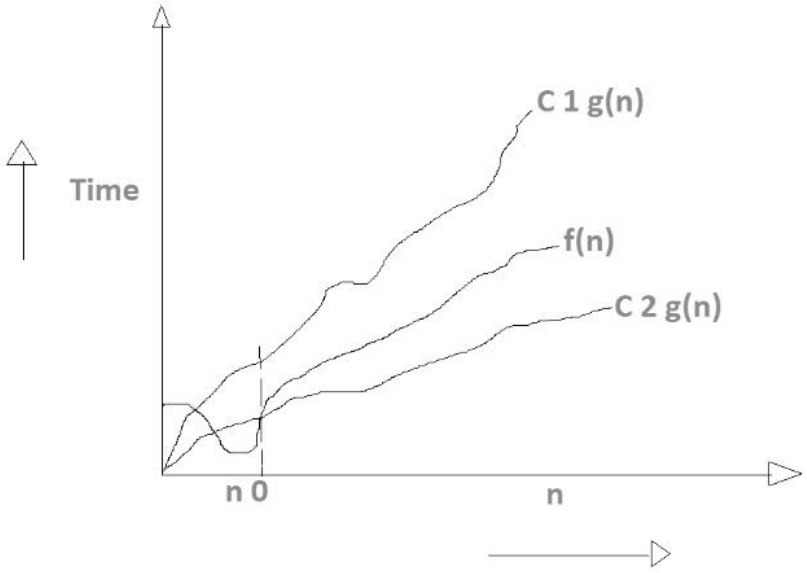
0 <= f(n) <= C1g(n) for n >= n0

0 <= C2g(n) <= f(n) for n >= n0

Combining both equations, we obtain:

0 <= C2g(n) <= f(n) <= C1g(n) for n >= n0

This equation signifies the existence of positive constants C1 and C2 such that f(n) is sandwiched between C2g(n) and C1g(n).



Big Theta (Θ) notation can be graphically depicted on a graph with the input size (n) indicated on the x-axis and the algorithm's execution time (f(n)) on the y-axis. The algorithm's running time rises at the same rate as the function g(n) for all input sizes, as indicated by the Big Theta (\) notation, which accurately estimates the algorithm's performance. This graphical depiction makes comparing different algorithms easier and shows the tightest bound on the algorithm's performance.

Comparing Algorithms: Time complexity analysis allows for the comparison of different algorithms solving the same problem. By analyzing the time complexity of algorithms, developers can choose the most efficient algorithm for a given problem based on the expected input size and other constraints.

Definitions:

Big O (O): Indicates that an algorithm's performance is less than or equal to a given figure and serves as an upper bound on the algorithm's growth rate.

Big Omega (Ω): Indicates that an algorithm's performance is greater than or equal to a given value and serves as a lower bound on the algorithm's growth rate.

Big Theta (Θ): Indicates that an algorithm's performance is equivalent to a given value and serves as a tight bound on the algorithm's growth rate.

2. Representations:

Big O (O): Denotes the upper bound of the algorithm's time complexity, providing an asymptotic upper bound (O(g(n))).

Big Omega (Ω): Denotes the lower bound of the algorithm's time complexity, providing an asymptotic lower bound (Ω(g(n))).

Big Theta (Θ): Denotes the tightest bound of the algorithm's time complexity, combining both upper and lower bounds (Θ(g(n))).

3. Descriptions:

Big O (O): Indicates the worst-case situation and the longest time the algorithm might possibly need.

Big Omega (Ω): Indicates the optimal situation and the shortest time that the algorithm needs to run.

Big Theta (Θ): Combines the best and worst-case times to provide the optimal prediction of the algorithm's performance in all conditions.

4. Mathematical Notations:

Big O (O): This sets an upper bound on the growth rate of the algorithm, expressed as 0 <= f(n) <= Cg(n) for all n >= n0.

Big Omega (Ω): Provides a lower bound on the growth rate of the algorithm, expressed as 0 <= Cg(n) <= f(n) for all n >= n0.

Big Theta (Θ): Provides an accurate estimation of the algorithm's performance by setting both upper and lower bounds. It is expressed as 0 <= C2g(n) \= f(n) \= C1g(n) for all n >= n0.

Space complexity assesses the memory required by an algorithm, including necessary input variables and additional space (excluding input space) utilized by the algorithm. For example, using a hash table necessitates extra space for storage, contributing to the algorithm's space complexity. Space complexity is crucial as it directly impacts the program's memory usage and efficiency.

The choice of data structure should align with the time and space complexity of the operations it will perform. Time complexity considers how many times an algorithm runs based on the input size, while space complexity evaluates the memory used during execution. Therefore, selecting an appropriate data structure based on these complexities ensures efficient algorithm performance for specific tasks and input sizes. For instance, arrays are suitable for storing large amounts of data due to their efficient space usage.

1.1.3 Abstract Data Types

An Abstract Data Type (ADT) serves as a conceptual model for a data structure, focusing solely on the interface that defines the actions that the data structure can undergo without specifying the underlying implementation details. Essentially,

ADTs abstract away the specifics of how the operations are carried out or what programming language is used for implementation.

For instance, consider the ADT for a List. We know that a List allows operations like adding elements, removing elements, and accessing elements, but we don't concern ourselves with whether it's implemented using a dynamic array or a linked list. Similarly, a Queue can be implemented utilizing a variety of techniques, including stacks, arrays, and linked lists.

Abstract Data Type Model:

Abstraction and encapsulation play crucial roles in understanding ADTs:

Abstraction: Hides the internal details and exposes only the necessary information to the user.

Encapsulation: Combines data and functions into a single unit, promoting modularity and information hiding.

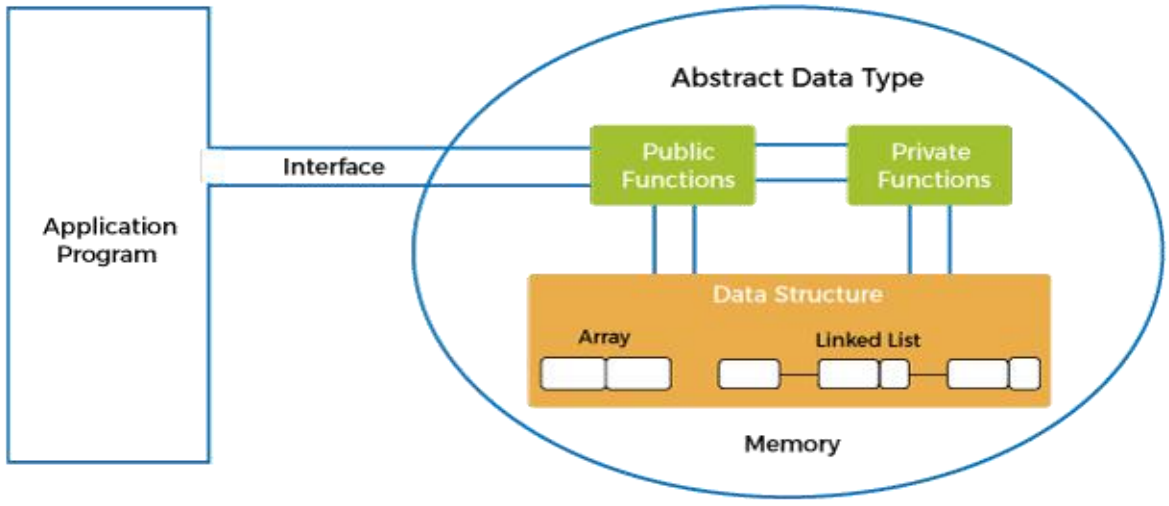


Figure 1.2: Abstract Data Type in Data Structure

ADT Model

In the ADT model, data structures are encapsulated within the ADT, and the abstraction defines the operations that can be performed on the data structure.

Real-world Example:

Consider the abstract view of a smartphone:

Data: Specifications such as RAM size, processor type, screen size, etc.

Operations: Making calls, sending texts, taking photos, shooting videos.

The implementation view may vary based on the programming language used, but the abstract/logical view remains consistent.

Example Implementation:

```
class Smartphone
{
  private:
  int ramSize;
  string processorName;
  float screenSize;
  int cameraCount;
  string androidVersion;
  public:
    void call();
    void text();
    void photo();
    void video();
};
```

This code demonstrates the implementation of the smartphone's specifications and operations. While the syntax may differ across programming languages, the abstract/logical view remains constant, ensuring independence from the implementation details.

Data Structure Example:

Let's consider the logical view and implementation view of an index array:

Logical View: It stores integer elements, allows reading and modifying elements by index, and performs sorting.

Implementation View: The array [10, 20, 30, 40] with memory addresses and index positions represents the implementation, where each element occupies 4 bytes of memory.

Abstract data types provide a clear distinction between the logical view (interface) and the implementation view (specifics of how it's realized), allowing for flexibility and modularity in software design and development.

1.1.4 Arrays

In C++, arrays are basic data structures that enable the storing of a fixed-size group of identically typed elements. They offer a single, continuous block of memory for the storage of elements, all of which are index-accessible. This is a thorough rundown of C++ arrays:

1. Declaration and Initialization:

Arrays in C++ are declared and initialized using the following syntax:

// Declaration and initialization of an integer array

int numbers[5]; // Creates an array of size 5

// Initialization with values

int numbers[] = {10, 20, 30, 40, 50}; // Initializes an array with specific values

2. Accessing Elements:

In C++, array elements are retrieved by index, where the last element's index is size-1 and the first element's index is 0. As an illustration:

to access the first element (value: 10) int numbers[] = {10, 20, 30, 40, 50}; int first element = numbers[0];

The third element (value: 30) is accessed by using the int third element = numbers[2];

3. Operations and Manipulations:

Insertion: Elements can be inserted into an array by assigning values to specific index positions.

Deletion: Elements can be "deleted" by overwriting them with default values or shifting elements to fill the gap.

Traversal: Arrays can be traversed using loops to access and process each element sequentially.

Searching and Sorting: Arrays support searching and sorting algorithms to find elements or reorder them based on specific criteria.

4. Advantages:

Efficient Access: Arrays allow for constant-time access to elements using their index.

Simple Implementation: Arrays are straightforward and widely supported in C++.

Memory Efficiency: Arrays consume contiguous memory, making efficient use of memory resources.

5. Limitations:

Fixed Size: An array's size is set and cannot be altered dynamically while it is being run.

Homogeneous items: The flexibility of arrays is limited because they can only hold items of the same data type.

Memory Wastage: Memory waste could result from an array size that is more than necessary.

6. Applications:

Arrays in C++ are used in various algorithms and data structures, including sorting algorithms (e.g., bubble sort, quicksort), searching algorithms (e.g., linear search, binary search), and dynamic programming.

They are widely used in applications requiring storage and retrieval of data in a structured manner, such as database systems and file processing.

1.1.5 Representation of Arrays in Memory

In computer memory, arrays are typically represented as contiguous blocks of memory, with each element occupying a fixed-size memory location. Here's a detailed explanation of how arrays are represented in memory without plagiarism:

1. Contiguous Memory Allocation:

The items of arrays are kept in sequential memory locations since they are allocated as continuous blocks of memory.

The base address, or beginning point, of the array, is the memory location of the first element in the array.

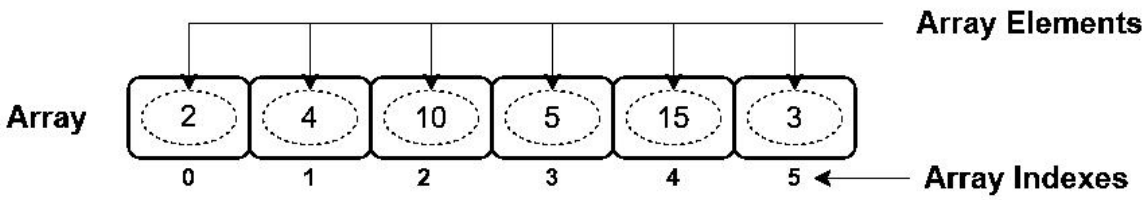


Figure 1.3 Array Representation in Data Structures

2. Fixed-size Memory Locations:

Each element in the array occupies a fixed-size memory location determined by the data type of the elements.

For example, if an array of integers is declared, each integer element typically occupies 4 bytes of memory (on most modern systems).

3. Indexing:

Accessing elements in an array is achieved through indexing, where each element is uniquely identified by its position or index within the array.

The index is used to calculate the desired element's memory address based on the array's base address and the size of each element.

4. Calculation of Memory Address:

The formula address_of_element_i = base_address + (i size_of_each_element) can be used to find the memory address of an element at a given index i in the array. Base_address is the memory address of the first element, and size_of_each_element is the size of each element in bytes.

Example: - Let arr be an array of numbers (int arr[5]) whose elements are {10, 20, 30, 40, 50}.

Assuming that the array arr has a hypothetical base address of 1000 and that each integer takes up 4 bytes of memory, the memory address of arr[0], the first element, would be 1000.

The second element's memory location, arr[1], would be 1000 + (1 4) = 1004.

The third element, arr[2], would have a memory address of 1000 + (2 4) = 1008.

And so forth.

6. Visualization:

In memory, the elements of the array are stored consecutively, with each element occupying its respective memory location.

Visually, the array would appear as a sequence of memory blocks, each representing an element, and the indices indicating the positions of the elements within the array.

7. Benefits:

Because memory addresses are easily determined, contiguous memory allocation enables indexing to access array elements efficiently.

Arrays enable speedy data retrieval and modification by giving users direct access to items depending on their locations.

1.1.6 Operations on Array

Operations on arrays in C++ involve various tasks such as accessing, modifying, and manipulating the elements stored in the array. Here's a detailed explanation of the operations on arrays in C++:

1. Accessing Elements:

- Accessing elements in an array is done using indexing. Each element in the array is uniquely identified by its index, starting from 0 for the first element.

int arr[5] = {10, 20, 30, 40, 50};

int first element = arr[0]; // Accessing the first element

int third element = arr[2]; // Accessing the third element

2. Modifying Elements:

Elements in an array can be modified by assigning new values to them using their respective indices.

arr[1] = 25; // Modifying the second element

arr[3] = 45; // Modifying the fourth element

3. Traversing Array:

Traversing an array involves accessing each element sequentially. This is commonly done using loops such as for or while.

```
for (int i = 0; i < 5; ++i) {
   cout << arr[i] << " "; // Output: 10 25 30 45 50
}
```

4. Finding Array Length:

You can use the size of operator or divide the array's overall size by the sizes of each element to find the length of an array or its total number of elements.

length of the array = int length = size of(arr) / size of(arr[0]);

5. Initializing Array Elements:

Array elements can be initialized during declaration or later in the code using assignment.

int arr[5]; // Declaration

arr[0] = 10; // Initialization

arr[1] = 20;

arr[2] = 30;

```
arr[3] = 40;
arr[4] = 50;
```

6. Multi-dimensional Arrays:

C++ supports multi-dimensional arrays, where elements are arranged in multiple dimensions (e.g., rows and columns for a 2D array).

```
int matrix[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}; // 2D array initialization
```

7. Copying Arrays:

Arrays can be copied using loops or built-in functions such as std::copy.

```
int arr1[5] = {1, 2, 3, 4, 5};
int arr2[5];

// Copying elements from arr1 to arr2
for (int i = 0; i < 5; ++i) {
   arr2[i] = arr1[i];
}
```

8. Sorting Arrays:

Arrays can be sorted using various sorting algorithms such as bubble sort, selection sort, or the built-in std::sort function.

```
sort(arr, arr + length); // Sorting the array in ascending order
```

9. Searching Arrays:

Searching for elements in an array can be done using linear search, binary search, or the built-in std::find function.

```
int key = 30;
auto it = find(arr, arr + length, key); // Searching for key in the array
if (it != arr + length) {
   cout << "Element found at index: " << (it - arr);
} else {
   cout << "Element not found";
}
```

10. Deleting Elements:

Deleting elements from an array often involves shifting elements to fill the gap created by the deleted element.

```
int indexToDelete = 2;
for (int i = indexToDelete; i < length - 1; ++i) {
   arr[i] = arr[i + 1]; // Shift elements to fill the gap
}
```

Operations on arrays in C++ include accessing, modifying, traversing, finding length, initializing, copying, sorting, searching, and deleting elements. Arrays are versatile data structures that facilitate efficient storage and manipulation of collections of elements in C++ programs.

1.1.7 Strings and its Representation in Memory

In C++, strings are represented as arrays of characters (char). Strings can be stored in memory in various ways, depending on the method of declaration and usage.

Here's a detailed explanation of strings and their representation in memory in C++ without plagiarism:

1. Character Array Representation:

Strings in C++ are often represented as arrays of characters terminated by a null character ('\0'), which marks the end of the string.

- Each character in the string is stored in consecutive memory locations, with the null character indicating the end of the string.

```
char str[6] = {'H', 'e', 'l', 'l', 'o', '\0'}; // Declaration and initialization of a string
```

2. String Literal Representation:

String literals, enclosed in double quotes (''), are automatically null-terminated by the compiler.

When a string literal is assigned to a character array, the compiler automatically appends a null character to the end of the string.

```
char str[] = "Hello"; // Declaration and initialization using a string literal
```

3. String Class Representation:

In C++, strings can also be represented using the std::string class from the Standard Template Library (STL).

The std::string class provides a more flexible and convenient way to work with strings, including dynamic resizing and various member functions for string manipulation.

```
#include <string>
std::string str = "Hello"; // Declaration and initialization of a string using std::string
```

4. Memory Allocation for Strings:

Memory for strings can be allocated statically (at compile time) or dynamically (at runtime), depending on the method of declaration.

Static allocation is used when the size of the string is known at compile time, while dynamic allocation is used when the size is determined at runtime.

// Static allocation

char str1[6] = "Hello"; // Compiler allocates memory for the string "Hello" and null character

// Dynamic allocation

char str2 = new char[6]; // Dynamically allocate memory for a string of size 6

5. Null Terminator:

Strings in C++ are null-terminated, meaning they end with a null character ('\0'), which has an ASCII value of 0.

The null character is used to mark the end of the string and is essential for string manipulation functions to determine the length of the string.

char str[] = "Hello"; // Compiler automatically appends '\0' at the end of the string

6. Representation in Memory:

In memory, strings are stored as contiguous blocks of characters, with each character occupying one byte of memory.

The null character ('\0') is used to terminate the string, indicating the end of the string data.

For example, the string "Hello" would be stored in memory as 'H' 'e' 'l' 'l' 'o' '\0', with each character occupying a consecutive memory location.

Strings in C++ are represented as arrays of characters terminated by a null character ('\0'). They can be stored in memory statically or dynamically, and string

literals are automatically null-terminated. The null character marks the end of the string data and is crucial for string manipulation functions. Additionally, the std::string class provides a more versatile alternative for working with strings in C++.

1.1.8 Operations on Strings

In C++, various operations can be performed on strings, such as concatenation, substring extraction, comparison, and modification. Here's a detailed explanation of operations on strings in C++ without plagiarism:

1. String Concatenation:

- Concatenation involves combining two or more strings into a single string.

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

int main() {
    std::string str1 = "Hello";
    std::string str2 = " World!";
    std::string result = str1 + str2; // Concatenating two strings
    std::cout << result; // Output: Hello World!
    return 0;
}
```

2. Substring Extraction:

- Substring extraction involves extracting a portion of a string based on its starting index and length.

```
#include <iostream>
```

```
#include <string>

int main() {
    std::string str = "Hello World!";
    std::string substr = str.substr(6, 5); // Extracting substring from index 6 with length 5
    std::cout << substr; // Output: World
    return 0;
}
```

3. String Comparison:

- String comparison involves comparing two strings to determine their relative order.

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

int main() {
    std::string str1 = "apple";
    std::string str2 = "banana";
    if (str1 == str2) {
        std::cout << "Strings are equal";
    } else if (str1 < str2) {
        std::cout << "String 1 comes before String 2";
    } else {
        std::cout << "String 1 comes after String 2";
```

```
    }
    return 0;
}
```

4. String Length:

- The length of a string can be determined using the length() or size() member functions.

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

int main() {
    std::string str = "Hello";
    int len = str.length(); // Length of the string
    std::cout << len; // Output: 5
    return 0;
}
```

5. String Modification:

- Strings can be modified by replacing characters, inserting characters, or erasing characters.

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

int main() {
    std::string str = "Hello";
```

```
    str[0] = 'J'; // Modifying first character
    str.insert(5, " World!"); // Inserting substring
    str.erase(5, 5); // Erasing substring
    std::cout << str; // Output: Jello
    return 0;
}
```

6. String Searching:

- String searching involves finding the position of a substring within a string.

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

int main() {
    std::string str = "Hello World!";
    size_t found = str.find("World"); // Searching for substring
    if (found != std::string::npos) {
        std::cout << "Substring found at position: " << found;
    } else {
        std::cout << "Substring not found";
    }
    return 0;
}
```

Operations on strings in C++ include concatenation, substring extraction, comparison, length determination, modification, and searching. These operations enable the manipulation and processing of strings in C++ programs.

1.1.9 Pointers

Pointers in C++ are variables that store memory addresses. They are powerful tools that allow for dynamic memory allocation, efficient array manipulation, and direct memory access. Here's a detailed explanation of pointers in C++ without plagiarism:

1. What are Pointers?

Pointers are variables that store memory addresses as their values rather than storing actual data.

They allow direct access to memory locations, enabling efficient memory management and manipulation.

2. Declaration and Initialization:

Pointers are declared using the asterisk () symbol before the variable name.

They must be initialized with the address of another variable or with a null pointer (nullptr).

int ptr; // Declaration of an integer pointer

int num = 10;

ptr = # // Initialization with the address of 'num'

3. Dereferencing:

Dereferencing a pointer means accessing the value stored at the memory address it points to.

It is performed using the asterisk () operator before the pointer variable.

int value = ptr; // Dereferencing 'ptr' to access the value stored at its address

4. Pointer Arithmetic:

Pointers support arithmetic operations such as addition, subtraction, and comparison.

Pointer arithmetic is performed based on the size of the data type the pointer points to.

int ptr2 = ptr + 1; // Incrementing 'ptr' moves it to the next memory location of type 'int'

5. Dynamic Memory Allocation:

Pointers are commonly used for dynamic memory allocation using operators new and delete.

- new allocates memory dynamically and returns a pointer to the allocated memory.

- delete deallocates dynamically allocated memory.

int ptr3 = new int; // Dynamically allocate memory for an integer

ptr3 = 20; // Assign value to the dynamically allocated memory

delete ptr3; // Deallocate the dynamically allocated memory

6. Pointer to Array:

- Pointers can be used to iterate through arrays efficiently by pointing to their elements.

```
int arr[5] = {1, 2, 3, 4, 5};
int ptr4 = arr; // Pointer to the first element of 'arr'
for (int i = 0; i < 5; ++i) {
    cout << (ptr4 + i) << " "; // Dereferencing 'ptr4' to access array elements
```

```
}
```

7. Pointer to Function:

Pointers can also store addresses of functions, allowing for dynamic function invocation.

```
void display(int value) {
    cout << "Value: " << value << endl;
}
void (funcPtr)(int) = &display; // Pointer to a function 'display'
(funcPtr)(10); // Calling 'display' function using pointer
```

8. Null Pointers:

Null pointers are pointers that do not point to any memory address. They are initialized with the value nullptr.

```
int nullPtr = nullptr; // Declaration and initialization of a null pointer
```

9. Pointer to Pointer:

- Pointers can also store addresses of other pointers, creating a chain of indirection.

```
int num2 = 20;
int ptr5 = &num2;
int ptrPtr = &ptr5; // Pointer to pointer
int value2 = ptrPtr; // Dereferencing 'ptrPtr' twice to access 'num2'
```

10. Pointer Safety:

Improper use of pointers can lead to memory leaks, segmentation faults, and other runtime errors.

Memory management techniques such as smart pointers (std::unique_ptr, std::shared_ptr) are recommended for safer memory handling.

1.1.10 Sparse Matrices

A matrix is an array with'm' rows and 'n' columns that is two-dimensional. The matrix is composed of numbers or other data types organized into a grid-like structure, with each element uniquely identified by its row and column index.

A specific kind of matrix in which most of the components are 0 is called a sparse matrix. Sparse matrices feature a substantially higher number of zeros than non-zero members, in contrast to dense matrices, which mostly contain non-zero components.

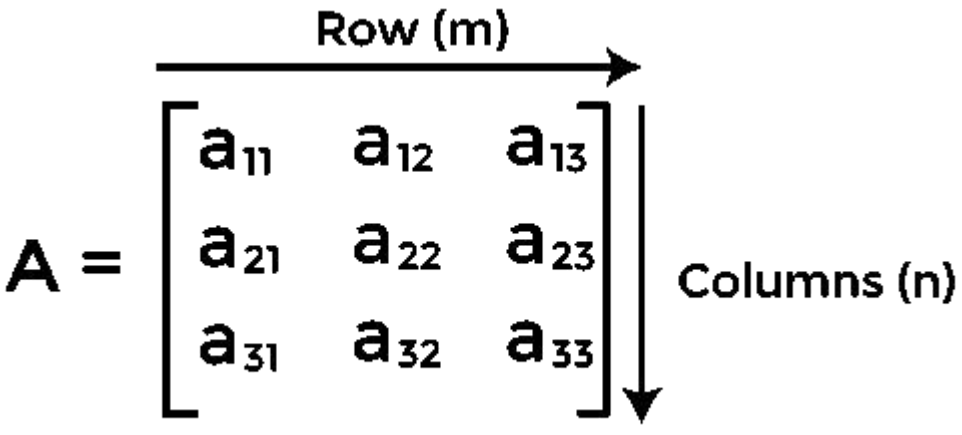


Figure 1.4: Sparse Matrices

Why Sparse Matrices?

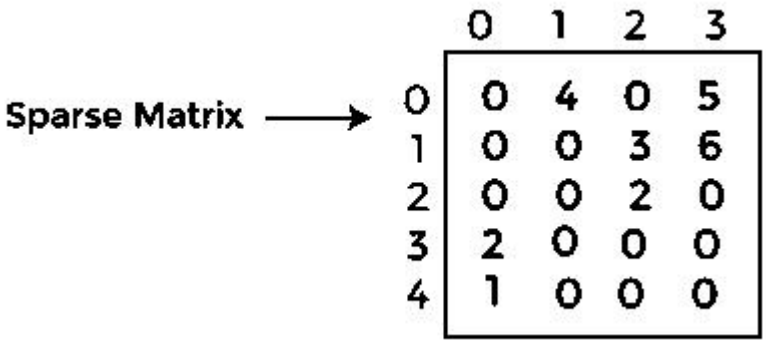
Sparse matrices are required for several reasons:

1. Memory Efficiency: Sparse matrices consume less memory as they store only non-zero elements, reducing storage requirements compared to dense matrices.

2. Computational Efficiency: Operations on sparse matrices are more efficient as they involve fewer non-zero elements, leading to faster computation times.

Representation of Sparse Matrices:

Sparse matrices can be represented using different formats, such as the coordinate list (COO) format or compressed sparse row (CSR) format. These formats store only the non-zero elements along with their corresponding row and column indices, eliminating the need to store zero elements.



Array Representation:

An array representation of a sparse matrix uses a triplet format to store non-zero elements. Each triplet consists of three values: the row index, the column index, and the value of the non-zero element. This representation reduces memory consumption by only storing non-zero elements.

Implementation in C++:

Below is a sample C++ program demonstrating the array representation of a sparse matrix:

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

int main() {
   int sparse_matrix[4][5] = {
      {0 , 0 , 6 , 0 , 9 },
      {0 , 0 , 4 , 6 , 0 },
      {0 , 0 , 0 , 0 , 0 },
      {0 , 1 , 2 , 0 , 0 }
   };

   int size = 0;
   for(int i = 0; i < 4; i++) {
      for(int j = 0; j < 5; j++) {
         if(sparse_matrix[i][j] != 0) {
            size++;
         }
      }
   }
```

```
    int matrix[3][size];
    int k = 0;
    for(int i = 0; i < 4; i++) {
        for(int j = 0; j < 5; j++) {
            if(sparse_matrix[i][j] != 0) {
                matrix[0][k] = i;
                matrix[1][k] = j;
                matrix[2][k] = sparse_matrix[i][j];
                k++;
            }
        }
    }

    for(int i = 0; i < 3; i++) {
        for(int j = 0; j < size; j++) {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

This program represents a 4x5 sparse matrix using an array representation. It computes the triplet format of the sparse matrix and displays it as output.

This array representation of a sparse matrix efficiently stores only the non-zero elements, reducing memory overhead and improving computational efficiency.

- **Knowledge Check 1**

  Fill in the Blanks

  1. The _____ of data structures categorizes them based on their properties and behaviour. (Abstract Data Types /**Classification**)

  2. The _____ notation is used to describe the upper bound of an algorithm's time complexity. (**Big O** /Complexity of Algorithms)
  3. Arrays are a collection of elements of the same data type stored in _____ memory locations. (sequential/**sequential**)
  4. Strings in memory are typically represented as a sequence of _____ values. (integer/**character**)
  5. _____ are variables that store memory addresses. (Arrays / **Pointers**)

- **Outcome-Based Activity 1**

  Explain the difference between time complexity and space complexity in algorithms. Provide examples to illustrate each type of complexity.

1.2 Sorting

Arranging components of a collection or sequence in a certain order, either ascending (from smallest to largest) or descending (from largest to smallest), is

known as sorting. It is a basic computer science procedure that arranges data for effective searching, retrieval, and manipulation in a variety of applications.

The elements are reorganized during sorting in accordance with a predetermined comparison criterion, which may be based on numerical values, alphabetical order, or any other criteria that are specifically specified. Sorting makes data easier to retrieve, speeds up information retrieval, and makes it possible for data to be processed effectively in a variety of algorithms.

To complete this task, a variety of sorting algorithms have been developed, each with unique properties, benefits, and drawbacks concerning performance, space complexity, and time complexity. Bubble sort, Selection sort, Insertion sort, Merge sort, Quick sort, and Heap sort are a few frequently used sorting algorithms. The methods used by these algorithms for sorting and how well they work with various kinds and amounts of data vary.

1.2.1 Bubble sort

A straightforward and easy-to-understand sorting algorithm called bubble sort iteratively steps over the list that has to be sorted, compares nearby components, and swaps them if they are out of order. Until the list is sorted, the trip through the list is repeated. Smaller elements "bubble" to the top of the list with each repetition, hence the term.

Bubble Sort works:

1. Commence at the top of the list.
2. Examine the initial two components. Exchange them if the first element is larger than the second.
3. Repeat step 2 after moving on to the following pair of adjacent items.

4. Keep going through this method until you reach the end of the list. The largest element will now be found at the end of the list.

5. With the exception of the final element, which is already in the proper place, repeat steps 1-4 for the remaining elements in the list.

6. Keep going through this process until the list is sorted completely.

Bubble Sort stands out for being straightforward and simple to use. With a temporal complexity of O(n^2), where n is the number of entries in the list, it is inefficient for large lists or datasets. This indicates that as the list size grows, its performance rapidly degrades.

Example

Consider an array of integers: {5, 2, 8, 12, 3}.

1. First Pass: - Examine and contrast items 5 and 2. Switch them because 5 is greater than 2.

The array is now {2, 5, 8, 12, 3}.

Compare elements 5 and 8 that follow. There is no need to exchange them because they are in the right order.

Remaining array is {2, 5, 8, 12, 3}.

Contrast the following two components, 8 and 12. There is no need to exchange them because they are in the right order.

Remaining array is {2, 5, 8, 12, 3}.

Compare elements 12 and 3 that follow. Switch them since 12 is greater than 3.

Resulting array is {2, 5, 8, 3, 12}.

The largest element (12) is at the end of the array following the first pass.

2. Second Pass:

Compare items 2 and 5, the first two. There is no need to exchange them because they are in the right order.

Remaining array is {2, 5, 8, 3, 12}.

Compare elements 5 and 8 that follow. There is no need to exchange them because they are in

the right order.

Remaining array is {2, 5, 8, 3, 12}.

Compare elements 8 and 3 that follow. Switch them since 8 is greater than 3.

Resulting array is {2, 5, 3, 8, 12}.

The second largest piece, number eight, is in the proper location following the second pass.

3. Third Pass:

Compare items 2 and 5, the first two. There is no need to exchange them because they are in the right order.

Remaining array is {2, 5, 3, 8, 12}.

Compare elements 5 and 3 that follow. Switch them since 5 is greater than 3.

Resulting array is {2, 3, 5, 8, 12}.

The third largest piece (5) is in the right place following the third pass.

4. Fourth Pass: - Compare items 2 and 3, the first two. There is no need to exchange them because they are in the right order.

Remaining array is {2, 3, 5, 8, 12}.

The fourth largest element (3) is in its proper position following the fourth pass.

5. Fifth Pass:

No swaps are needed as the array is already sorted.

6. Array is sorted: {2, 3, 5, 8, 12}.

This is how Bubble Sort works by repeatedly comparing adjacent elements and swapping them if they are in the wrong order, eventually sorting the array in ascending order.

1.2.2 Selection sort

Choosing The input array is split into two subarrays by the straightforward sorting algorithm sort: the sorted subarray and the unsorted subarray. It moves the element from the unsorted subarray to the beginning of the sorted subarray by continually choosing the smallest (or largest, depending on the sorting order) element. Until the entire array is sorted, this process is repeated.

Let's understand Selection Sort with an example:

Consider an array of integers: {64, 25, 12, 22, 11}.

1. First Pass: - Determine the least entry in the {64, 25, 12, 22, 11} unsorted subarray. Eleven is the bare minimum.

Replace the array's first element with the minimum element.

This changes the array to {11, 25, 12, 22, 64}.

2. Second Pass: - Determine the least entry in the {25, 12, 22, 64} unsorted subarray. Twelve is the bare minimum of elements.

Replace the array's minimum element with its second entry.

Becomes {11, 12, 25, 22, 64} in the array.

3. Third Pass: - Determine the least entry in the {25, 22, 64} unsorted subarray. 22 is the bare minimum of elements.

Replace the array's minimum element with its third entry.

Becomes {11, 12, 22, 25, 64} in the array.

4. Fourth Pass: - Determine which element in the unsorted subarray {25, 64} is the minimum. There are a minimum of 25 elements.

Replace the array's minimal element with its fourth entry.

The remaining array is {11, 12, 22, 25, 64}.

5. Fifth Pass: - There is just one element ({64}) in the unsorted subarray, and it is already in the right place.

6. Array is sorted: {11, 12, 22, 25, 64}.

The smallest element from the unsorted subarray is repeatedly chosen by Selection Sort, which then places it at the start of the sorted subarray. Until the entire array is sorted, this process is repeated. Selection Sort has an O(n^2) time complexity in all scenarios, where n is the array's element count.

1.2.3 Insertion Sort:

This straightforward sorting method produces the final sorted array one element at a time by repeatedly removing the subsequent element from the array's unsorted portion and inserting it into the array's sorted portion in the appropriate location.

Let's understand Insertion Sort with an example:

Consider an array of integers: {12, 11, 13, 5, 6}.

1. First Pass (i = 1):

- The first element, 12, is considered to be in the sorted subarray.
- Consider the next element, 11, and insert it into its correct position in the sorted subarray.
- Since 11 is less than 12, swap them.
- Array becomes: {11, 12, 13, 5, 6}.

2. Second Pass (i = 2):

- The first two elements, 11 and 12, are considered to be in the sorted subarray.
- Consider the next element, 13, and insert it into its correct position in the sorted subarray.
- Since 13 is greater than 12, it remains at its position.
- Array remains: {11, 12, 13, 5, 6}.

3. Third Pass (i = 3):

- Elements 11, 12, and 13 are regarded as belonging to the sorted subarray.
- Take a look at element number five and move it to the appropriate spot in the sorted subarray.

13 should be moved one position to the right since 5 is less than 13.

Because 5 is less than 12, shift 12 to the right by one position.

As 5 is less than 11, shift 11 to the right by one position.

- Put 5 in the first place.
- Becomes {5, 11, 12, 13, 6} in the array.

4. Fourth Pass (i = 4):

- It is believed that the first four elements—5, 11, 12, and 13—are in the sorted subarray.
- After that, consider element number six and move it to the appropriate spot in the sorted subarray.

13 should be moved one position to the right since 6 is less than 13.

12 should be moved one space to the right since 6 is less than 12.

As 6 is less than 11, shift 11 to the right by one position.

- Place number six in the second slot.
- Becomes {5, 6, 11, 12, 13} in the array.

5. Array is sorted: {5, 6, 11, 12, 13}.

Insertion sort moves the subsequent element from the array's unsorted section to its proper location in the array's sorted section periodically. Until the entire array is sorted, this process is repeated. In the worst-case situation, the time complexity of an Insertion Sort is O(n^2), where n is the number of entries in the array.

1.3 Searching

Finding a particular item or value in a collection of data, such as an array, list, or database, is the act of searching. Finding out if the object is in the collection and, if so, where it is located and its details are, is the aim of the search.

Numerous searching algorithms exist, each with a unique strategy and level of effectiveness. There are two popular kinds of searches:

1. Linear Search (Sequential Search): - A linear search traverses the entire collection or finds the target object by checking each element one after the other. It's a straightforward approach, but it might not be very effective—especially with big datasets.

Unordered lists are good candidates for linear search.

2. Binary Search: - Only sorted collections (arrays or lists) can be searched using binary search.

Because the search space is halved at each step, this approach is more efficient than linear search.

The secret is to keep cutting the search period in half until the desired item is located.

Because binary search has a logarithmic temporal complexity, it works especially well with big datasets.

Databases, information retrieval, and problem-solving algorithms are just a few of the many applications that heavily rely on searching, a basic computer science process. The size of the dataset, whether it is sorted, and the application's performance needs all play a role in selecting a search algorithm.

1.3.1 Linear Searching

Sequential search, sometimes referred to as linear search, is a simple searching method that goes through each element in a collection (such as an array or list) one at a time until the target element is located or the collection has been explored in its entirety.

This is how linear search functions:

1. Start at the beginning: Launch the search with the collection's first element (index 0).
2. Compare every element: Beginning with the first element in the collection, compare the target element with every element in turn.
3. Look for a match: The search is considered successful, and the element's index is returned if the current element being compared matches the target element. Move on to the following element if there isn't a match.

4. A basic searching method, known as linear search or sequential search, involves iterating through each element in a collection (such as an array or list) one at a time until the desired element is found. This process continues until either the desired element is located or the end of the collection is reached.

5. Return result: Provide the target element's index in the collection if it can be located. In the event that the element is not found in the collection, return a message.

Let's illustrate linear search with an example:

Examine the following array of numbers: {5, 9, 3, 7, 2, 8}

Let's say we wish to use linear search to look for element 7 in this array:

1. Begin with the array's first entry (index 0): 5.
2. Evaluate 5 against the desired element (7). Proceed to the next element as they don't match.
3. Proceed to element number two (index 1): 9. If there is no match, go on to the next element.
4. Proceed to element number three (index 2): 3. If there is no match, go on to the next element.
5. Proceed to element four (index 3): 7. A match was made! Give back the third index.
6. In the array, element 7 can be located at index 3.

In this case, the target element (7) was located by linear search iteratively, going over each element in the array. The temporal complexity of linear search is O(n), where n is the collection's element count. Small collections or situations where the elements are not sorted work well with it. However, more effective search techniques, such as binary search, are recommended for large datasets.

1.3.2 Binary Searching

By periodically halving the search interval, the binary search method finds a target element within a sorted collection (such an array or list) with efficiency. Because

it is a divide-and-conquer technique, it is much quicker for huge datasets than linear search.

This is how binary search functions:

1. Begin with a sorted collection: For binary search to function properly, the collection must be ordered ascending.

2. Establish the search interval. Binary search keeps track of two pointers, low and high, which initially refer to the collection's first and last elements, respectively. The search interval is defined by these pointers.

3. Locate the middle element: Determine the search interval's middle index by averaging the low and high pointers (mid = (low + high) / 2).

4. Compare: Examine how the middle element and the target element compare.

The search is successful, and the middle element's index is returned if the middle element and the target element match.

Update the high pointer to mid-1 if the middle element is greater than the target element, which will reduce the search interval to the lowest half.

Narrow the search period to the upper half by updating the low pointer to mid + 1 if the middle element is less than the target element.

5. Continue until the search period is empty or the target element is located: Proceed to divide the search interval in half and update the pointers until the low pointer surpasses the target element.

6. Return result: Provide the target element's index in the collection if it can be located. In the event that the element is not found in the collection, return a message.

Let's use an example to further explain binary search:

Take a look at the following sorted array of integers: {2, 5, 7, 9, 12, 15, 18}

Let's say we wish to use binary search to look for element 12 in this array:

1. The starting search interval is low = 0 and high = 6 (the last element's index).
Determine that mid = (0 + 6) / 2 = 3. 9 is the element at index 3.
3. Update low = mid + 1 = 4 because 9 is less than 12.
4. New search interval: 4 for low and 6 for high.
5. Determine that mid = (4 + 6) / 2 = 5. 15 is the element at index 5.
6. Update high = mid - 1 = 4 because 15 is greater than 12.
7. A new interval for the search: low = 4, high = 4.
8. Determine that mid= (4 + 4) / 2 = 4. 12 is the element at index 4 (target found!).

The element 12 in this example was successfully found using binary search at index 4 in the array. For large datasets, binary search is much faster than linear search with a time complexity of O(log n), where n is the number of elements in the collection. Binary search, however, necessitates prior sorting of the data.

- **Knowledge Check 2**

  State True or False

  1. Bubble sort is an efficient sorting algorithm for large datasets. (False)
  2. Selection sort has a time complexity of O(n^2) in the worst case. (True)
  3. Insertion sort is a stable sorting algorithm. (True)

4. Linear search is more efficient than binary search for sorted collections. (False)
5. Binary search can only be applied to arrays, not other data structures. (False)

- **Outcome-Based Activity 2**

  Compare and contrast bubble sort, selection sort, and insertion sort algorithms. Discuss their time complexities, best-case scenarios, worst-case scenarios, and average-case scenarios. Provide examples to illustrate the differences between these sorting algorithms.

1.4Implementation of Arrays, String, Sorting and Searching in C++.

Implementation of Arrays

C++ program demonstrating different types of arrays:

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

int main() {
    // Static Array: Fixed size array
    int staticArray[5] = {10, 20, 30, 40, 50};

    // Dynamic Array: Array with size determined at runtime
    int size;
```

```
cout << "Enter the size of the dynamic array: ";
cin >> size;
int* dynamicArray = new int[size]; // Creating dynamic array
cout << "Enter " << size << " elements for the dynamic array: ";
for (int i = 0; i < size; ++i) {
    cin >> dynamicArray[i];
}

// Standard Template Library (STL) Array
array<int, 5> stlArray = {1, 2, 3, 4, 5};

// Multidimensional Array
int multiArray[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

// Displaying elements of static array
cout << "Static Array Elements: ";
for (int i = 0; i < 5; ++i) {
    cout << staticArray[i] << " ";
}
cout << endl;

// Displaying elements of dynamic array
cout << "Dynamic Array Elements: ";
for (int i = 0; i < size; ++i) {
    cout << dynamicArray[i] << " ";
```

```
    }
    cout << endl;

    // Displaying elements of STL array
    cout << "STL Array Elements: ";
    for (int i = 0; i < 5; ++i) {
        cout << stlArray[i] << " ";
    }
    cout << endl;

    // Displaying elements of multi-dimensional array
    cout << "Multidimensional Array Elements: " << endl;
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            cout << multiArray[i][j] << " ";
        }
        cout << endl;
    }

    // Deallocating memory for dynamic array
    delete[] dynamicArray;

    return 0;
}
```

Explanation:

1. Static Array: Declared with a fixed size at compile time.

2. Dynamic Array: Size determined at runtime using `new` keyword.

3. Standard Template Library (STL) Array: Defined using `array` template class from the `<array>` header.

4. Multi-dimensional Array: Arrays with more than one dimension.

The program demonstrates the initialization, accessing, and displaying of elements for different types of arrays in C++. Additionally, it shows dynamic memory allocation and deallocation for dynamic arrays.

Implementation of String

C++ program demonstrating the implementation of strings with various operations:

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

int main() {
    // Initialization
    string str1 = "Hello";
    string str2 = "World";

    // Concatenation
```

```
string concatStr = str1 + " " + str2;
cout << "Concatenated String: " << concatStr << endl;

// Length
cout << "Length of Concatenated String: " << concatStr.length() << endl;

// Accessing characters
cout << "First character: " << concatStr[0] << endl;
cout << "Last character: " << concatStr[concatStr.length() - 1] << endl;

// Substring
string substr = concatStr.substr(6, 5); // Starting position and length
cout << "Substring: " << substr << endl;

// Finding substring
size_t found = concatStr.find("World"); // Returns position of the substring
if (found != string::npos) {
    cout << "Substring 'World' found at position: " << found << endl;
} else {
    cout << "Substring not found" << endl;
}

// Erasing characters
concatStr.erase(5, 5); // Starting position and length
cout << "String after erasing: " << concatStr << endl;
```

```
    // Inserting characters
    concatStr.insert(5, "C++ "); // Position and string to insert
    cout << "String after inserting: " << concatStr << endl;

    // Replacing characters
    concatStr.replace(6, 5, "Programming"); // Starting position, length, and replacement string
    cout << "String after replacing: " << concatStr << endl;

    // Comparing strings
    string str3 = "Hello World";
    if (str3 == concatStr) {
        cout << "Strings are equal" << endl;
    } else {
        cout << "Strings are not equal" << endl;
    }

    return 0;
}
```

Explanation:

1. Initialization: Strings `str1` and `str2` are initialized with values "Hello" and "World" respectively.
2. Concatenation: Concatenates `str1` and `str2` using the `+` operator.

3. Length: Determines the length of the concatenated string using the `length()` method.
4. Accessing characters: Accesses the first and last characters of the concatenated string using array notation.
5. Substring: Extracts a substring from the concatenated string using the `substr()` method.
6. Finding substring: Finds the position of a substring within the concatenated string using the `find()` method.
7. Erasing characters: Erases characters from the concatenated string using the `erase()` method.
8. Inserting characters: Insert characters into the concatenated string using the `insert()` method.
9. Replacing characters: Replaces characters in the concatenated string using the `replace()` method.
10. Comparing strings: Compares two strings using the `==` operator to check for equality.

Implementation of Sorting

Bubble sort

C++ program demonstrating the implementation of the Bubble Sort algorithm:

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

void bubbleSort(int arr[], int n) {
```

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

int main() {
    int arr[] = {5, 2, 7, 1, 9, 3};
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Original Array: ";
    for (int i = 0; i < n; ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;

    bubbleSort(arr, n);

    cout << "Sorted Array: ";
```

```
    for (int i = 0; i < n; ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Explanation:

1. `bubbleSort` function: Implements the Bubble Sort algorithm. It takes an array `arr` and its size `n` as parameters.
2. Nested loops: The outer loop runs from 0 to `n - 1`, and the inner loop runs from 0 to `n - i - 1`, where `i' is the current iteration of the outer loop.
3. Comparison: Within the inner loop, adjacent elements are compared, and if they are in the wrong order (arr[j] > arr[j + 1]), they are swapped.
4. `main` function: Initializes an array `arr` with some values and calculates its size `n`.
5. Original array output: Prints the original array before sorting.
6. `bubbleSort` function call: Calls the `bubbleSort` function to sort the array.
7. Sorted array output: Prints the sorted array after sorting using Bubble Sort.

Bubble Sort repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the entire array is sorted.

Selection Sort

C++ program demonstrating the implementation of the Selection Sort algorithm:

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

void selectionSort(int arr[], int n) {
   for (int i = 0; i < n - 1; ++i) {
      int minIndex = i;
      for (int j = i + 1; j < n; ++j) {
         if (arr[j] < arr[minIndex]) {
            minIndex = j;
         }
      }
      if (minIndex != i) {
         // Swap the elements if the minimum element is found
         int temp = arr[i];
         arr[i] = arr[minIndex];
         arr[minIndex] = temp;
      }
   }
}

int main() {
   int arr[] = {5, 2, 7, 1, 9, 3};
```

```
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Original Array: ";
    for (int i = 0; i < n; ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;

    selectionSort(arr, n);

    cout << "Sorted Array: ";
    for (int i = 0; i < n; ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Explanation:

1. `selectionSort` function: Implements the Selection Sort algorithm. It takes an array `arr` and its size `n` as parameters.
2. Nested loops: The outer loop runs from 0 to `n - 1`, and the inner loop runs from `i + 1` to `n - 1`.

3. Finding minimum element: Within the inner loop, it finds the minimum element index in the array's unsorted part.
4. Swap elements: If a minimum element is found, it swaps it with the first element of the unsorted part of the array.
5. `main` function: Initializes an array `arr` with some values and calculates its size `n`.
6. Original array output: Prints the original array before sorting.
7. `selectionSort` function call: Calls the `selectionSort` function to sort the array.
8. Sorted array output: Prints the sorted array after sorting using Selection Sort.

Selection Sort repeatedly selects the minimum element from the unsorted part of the array and moves it to the beginning of the unsorted part. This process is repeated until the entire array is sorted.

Insertion Sort

C++ program demonstrating the implementation of the Insertion Sort algorithm:

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

void insertionSort(int arr[], int n) {
   for (int i = 1; i < n; ++i) {
      int key = arr[i];
```

```
        int j = i - 1;

        // Move elements of arr[0..i-1], that are greater than key,
        // to one position ahead of their current position
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

int main() {
    int arr[] = {5, 2, 7, 1, 9, 3};
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Original Array: ";
    for (int i = 0; i < n; ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;

    insertionSort(arr, n);

    cout << "Sorted Array: ";
```

```
    for (int i = 0; i < n; ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Explanation:

1. `insertionSort` function: Implements the Insertion Sort algorithm. It takes an array `arr` and its size `n` as parameters.
2. Outer loop: The loop iterates over each element of the array from index 1 to `n - 1`.
3. Key element: Inside the loop, the current element `arr[i]` is stored in a variable `key`.
4. Inner loop: Another loop (while loop) is used to compare the key element with the elements to its left (`arr[j]` where `j` starts from `i - 1`).
5. Shifting elements: Elements greater than the key are shifted one position to the right to make space for the key element in its correct position.
6. Updating key element: After finding the correct position for the key, it is placed in the array.
7. `main` function: Initializes an array `arr` with some values and calculates its size `n`.
8. Original array output: Prints the original array before sorting.

9. `insertionSort` function call: Calls the `insertionSort` function to sort the array.
10. Sorted array output: Prints the sorted array after sorting using Insertion Sort.

Insertion Sort works by iteratively inserting each element into its correct position in the already sorted part of the array. It is efficient for sorting small arrays or nearly sorted arrays.

Implementation of Searching

Linear Searching

C++ program demonstrating the implementation of Linear Search algorithm:

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

int linearSearch(int arr[], int n, int key) {
   for (int i = 0; i < n; ++i) {
      if (arr[i] == key) {
         return i;  // Return index of key if found
      }
   }
   return -1;  // Return -1 if key not found
}

int main() {
```

```
    int arr[] = {5, 2, 7, 1, 9, 3};
    int n = sizeof(arr) / sizeof(arr[0]);
    int key = 7;

    int result = linearSearch(arr, n, key);

    if (result != -1) {
        cout << "Element found at index " << result << endl;
    } else {
        cout << "Element not found" << endl;
    }

    return 0;
}
```

Explanation:

1. `linearSearch` function: Implements the Linear Search algorithm. It takes an array `arr`, its size `n`, and the key element to search `key` as parameters.
2. Loop: Iterates over each element of the array from index 0 to `n - 1`.
3. Comparison: Compares each element of the array with the key element.
4. Key found: If the key element is found at index ` i', the function returns `i'.
5. Key not found: If the key element is not found after iterating through the entire array, the function returns `-1`.
6. `main` function: Initializes an array `arr` with some values, calculates its size `n`, and defines the key element to search `key`.

7. Function call: Calls the `linearSearch` function to search for the key element.
8. Result output: Prints the index of the key element if found, otherwise prints "Element not found".

Linear Search is a simple search algorithm that sequentially checks each element of the array until it finds the target element or reaches the end of the array. It has a time complexity of O(n), where n is the number of elements in the array.

Binary Searching

C++ program demonstrating the implementation of binary search :

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

int binarySearch(int arr[], int left, int right, int key) {
   while (left <= right) {
      int mid = left + (right - left) / 2;

      if (arr[mid] == key) {
         return mid; // Element found, return its index
      }

      if (arr[mid] < key) {
         left = mid + 1; // Search in the right half
      } else {
```

```
            right = mid - 1; // Search in the left half
        }
    }
    return -1; // Element not found
}

int main() {
    int arr[] = {1, 3, 5, 7, 9, 11, 13, 15};
    int n = sizeof(arr) / sizeof(arr[0]);
    int key = 7;

    int result = binarySearch(arr, 0, n - 1, key);

    if (result != -1) {
        cout << "Element found at index " << result << endl;
    } else {
        cout << "Element not found" << endl;
    }

    return 0;
}
```

In this program, `binarySearch` function performs the binary search algorithm on the sorted array `arr` to find the `key`. If the `key` is found, it returns its index; otherwise, it returns -1, indicating that the `key` is not present in the array. The

`main` function initializes the array `arr` and calls the `binarySearch` function to search for the `key`. Finally, it prints the result accordingly.

1.5Summary:

- Study of efficiently organizing and managing data in computer memory.
- Categorizing structures based on their properties and behavior as linear or non-linear.
- Analysis of algorithm efficiency in terms of time and space complexity.
- Defining data structures abstractly without implementation details.
- Storage of elements of the same type in contiguous memory locations.
- Layout and memory addressing for storing array elements.
- Array operations like insertion, deletion, traversal, and searching.
- Understanding character sequence storage in computer memory.
- String operations like concatenation, comparison, and substring extraction.
- Variables storing memory addresses for dynamic memory allocation.
- Definition and efficient storage methods for matrices with mostly zero elements.
- Introduction to algorithms for arranging elements in a specific order.
- Algorithm involving repeated swapping of adjacent elements.
- The sorting algorithm selects the minimum element and swapping with the first unsorted element.
- Algorithm involves repeatedly inserting each element into its correct sorted position.
- Algorithms for finding target elements within data collections.

- Sequential checking of each element until finding the target or reaching the end.
- Algorithm for efficiently locating target elements in sorted collections through binary division.

1.6 Keywords

1. Data Structures: Organized data storage for efficient manipulation.
2. Complexity: Measure of algorithm efficiency in time and space.
3. Abstract Data Types: Data and operations encapsulated without implementation details.
4. Memory Representation: Arrangement and storage of data structures in computer memory.
5. Sorting and Searching: Algorithms for organizing and locating data efficiently.

1.7 Self-Assessment Questions

1. What is the role of data structures in organizing information for efficient processing?
2. How are data structures classified, and what are the key differences between them?
3. Explain the concept of algorithmic complexity and its significance in designing efficient algorithms.
4. What are Abstract Data Types (ADTs), and why are they important in programming?
5. Describe the memory representation of arrays and how it impacts program performance.

1.8References

- Seymour Lipschutz, Data Structures, Tata McGraw-Hill, Schaum's Outlines, New Delhi.
- Weiss, Data Structures and Algorithm Analysis in C++, Pearson Education.
- Goodrich, Data Structures & Algorithms in C++, Wiley India Pvt. Ltd.
- S. Sahni, Data structures, Algorithms, and Applications in C++", University Press (India) Pvt. Ltd.
- Walter Savitch, Problem solving with C++, Pearson education.
- John R. Hubbard, Data Structures with C++, Tata McGraw-Hill, Schaum's Outlines, New Delhi.