Bachelor of Science
(B.Sc. - PCM)

C++ and DSA
(DBSPDS101T24)

Self-Learning Material
(SEM 1)

Jaipur National University
Centre for Distance and Online Education

Established by Government of Rajasthan
Approved by UGC under Sec 2(f) of UGC ACT 1956
&
NAAC A+ Accredited

PREFACE

Welcome to the world of C++ programming and Data Structures and Algorithms (DSA). This book is designed to provide a comprehensive introduction to the C++ programming language and its application in the context of data structures and algorithms.

C++ is a powerful and versatile programming language widely used in the software industry for developing a wide range of applications, from system software to graphical user interfaces, games, and more. With its rich set of features, including object-oriented programming, templates, and standard libraries, C++ offers both flexibility and efficiency, making it a popular choice among developers.

Additionally, this book will delve into the realm of Data Structures and Algorithms, which form the backbone of computer science and software development. Understanding data structures and algorithms is crucial for building efficient and scalable software systems, as they provide the foundation for organizing, storing, and processing data effectively.

We will explore various data structures, including arrays, linked lists, stacks, queues, trees, and graphs, along with their associated algorithms for operations such as searching, sorting, insertion, and deletion. Through practical examples, exercises, and programming assignments, we will develop a deep understanding of how to choose the right data structure and algorithm for solving specific problems efficiently.

Whether you are a beginner looking to learn programming from scratch or an experienced developer seeking to enhance your skills in C++ and DSA, this book aims to provide you with the knowledge and tools you need to succeed in today's competitive software industry.

So, let's embark on this exciting journey together and unlock the endless possibilities of C++ programming and Data Structures and Algorithms!

Table of Content

Content

Unit-1

Arrays

Objective:

At the end of this session student shall be learning:

1. Understanding Array Basics
2. Applying Array Operations
3. Analyzing Time and Space Complexity
4. Implementing Array-Based Algorithms
5. Understanding Multidimensional Arrays

Structure:

1.1 Arrays
1.2 Types of arrays
1.3 Characteristics of Arrays
1.4 Sparse matrix
1.5 Representation of Sparse Matrix
1.6 Advantages of Sparse Matrix
1.7 Applications of Sparse Matrices
1.8 Stacks
1.9 Types of Stacks
1.10 Simple stack implementation
1.11 Multiple stack implementation
1.12 Infix, Prefix and Postfix Expressions
1.13 Conversion of Infix, Prefix and Postfix Expressions
1.14 Applications of Stack
1.15 Limitations of Array Representation of Stack

1.1 **Arrays**: An array is a data structure used in programming to store a collection of elements, typically of the same type, in a contiguous block of memory. Arrays are fundamental structures in many programming languages, providing efficient storage and retrieval of data.

1.2 Types of Arrays:

1. One-Dimensional Array:
    - Also known as a single-dimensional array, it is a list of elements stored in a linear sequence.
    - Example: int[] numbers = {1, 2, 3, 4, 5};
2. Multi-Dimensional Array:
    - Arrays that have more than one dimension, such as two-dimensional arrays (matrices) or three-dimensional arrays.
    - Example of a two-dimensional array: int[][] matrix = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };

1.3 Characteristics of Arrays:

- Fixed Size: Once an array is created, its size cannot be changed. This means the number of elements it can hold is fixed.
- Index-Based Access: Elements can be accessed directly using their index positions, which provides fast and efficient retrieval.
- Homogeneous Elements: Arrays typically store elements of the same data type, ensuring type consistency.

1.4 Sparse Matrices:

A sparse matrix is a type of matrix in which most of the elements are zero. Due to the large number of zero elements, storing all elements in a standard two-dimensional array would be inefficient in terms of both space and computational resources. Instead, specialized storage methods are used to take advantage of the sparsity.

1.5 Representations of Sparse Matrices

1. **Array Representation**:
    - **Compressed Sparse Row (CSR)**: This method stores the matrix in three separate arrays. One array holds the non-zero values, another holds the column indices of these non-zero values, and a third array keeps track of the starting index of each row in the values array.

        Example: For a matrix:
        [0 0 3]

        [4 0 0]

        [0 5 0]

        - CSR representation would be:
            - Values array: **[3, 4, 5]**
            - Column indices: **[2, 0, 1]**
            - Row pointers: **[0, 1, 2, 3]**

- **Compressed Sparse Column (CSC)**: Similar to CSR, but the columns are compressed instead of the rows. It uses three arrays to store non-zero values, their row indices, and column pointers.

2. **Linked List Representation**:

- In this approach, each non-zero element of the matrix is represented by a node in a linked list. Each node contains information about the value of the element, its row and column indices, and pointers to the next non-zero elements in the row and column.

  Example: For a matrix:
  [0 0 3]

  [4 0 0]

  [0 5 0]

The linked list representation would have nodes:

Node1: Value **3**, Row **0**, Column **2**, Pointer to next in row **null**, Pointer to next in column points to Node 3.

Node2: Value **4**, Row **1**, Column **0**, Pointer to next in row **null**, Pointer to next in column **null**.

Node3: Value **5**, Row **2**, Column **1**, Pointer to next in row **null**, Pointer to next in column **null**.

1.6 Advantages of Sparse Matrix Representations

- **Memory Efficiency**: Sparse matrix representations save memory by storing only non-zero elements, significantly reducing space requirements.

- **Improved Performance**: Operations on sparse matrices, such as matrix-vector multiplication, can be performed more efficiently by skipping zero elements.

1.7 Applications of Sparse Matrices

- **Scientific Computing**: Many problems in scientific computing involve large matrices with few non-zero elements.
- **Graph Algorithms**: Sparse matrices are used to represent adjacency matrices in graphs where the number of edges is much less than the possible number of edges.
- **Machine Learning**: In machine learning, sparse matrices often represent data sets with many features but few non-zero values, such as text data in natural language processing.

1.8 Stacks: A stack is a linear data structure that follows the Last In, First Out (LIFO) principle. This means that the last element added to the stack will be the first one to be removed. Stacks are widely used in various applications, such as expression evaluation, backtracking algorithms, and function call management in programming languages.

1.9 Types of Stacks

1. **Simple Stack**:
    - A basic stack where elements are pushed and popped from one end called the top.
2. **Multiple Stacks**:
    - More than one stack can be implemented within a single array. This is useful in memory-constrained environments where multiple stacks share a common storage space.

1.10 Simple Stack Implementation in C++

Here's an implementation of a simple stack using an array in C++:

```
 #include <iostream>
#define MAX 1000
class Stack {
    int top;
public:
    int arr[MAX]; // Maximum size of Stack
    Stack() { top = -1; }
    bool push(int x);
     int pop();
    int peek();
    bool isEmpty();
};
bool Stack::push(int x) {
    if (top >= (MAX - 1)) {
        std::cout << "Stack Overflow\n";
        return false;
    } else {
        arr[++top] = x;
        std::cout << x << " pushed into stack\n";
        return true;
    }
```

```
}
      int Stack::pop() {
   if (top < 0) {
      std::cout << "Stack Underflow\n";
      return 0;
   } else {
      int x = arr[top--];
      return x;
   }
}
int Stack::peek() {
   if (top < 0) {
      std::cout << "Stack is Empty\n";
      return 0;
   } else {
      int x = arr[top];
      return x;
   }
}
bool Stack::isEmpty() {
   return (top < 0);
}
int main() {
```

```
    Stack s;
    s.push(10);
    s.push(20);
    s.push(30);
    std::cout << s.pop() << " popped from stack\n";
    std::cout << "Top element is " << s.peek() << std::endl;
    std::cout << "Stack is empty: " << (s.isEmpty() ? "true" : "false") << std::endl;
    return 0;
}
```

1.11 Multiple Stacks Implementation in C++

Here's an implementation of multiple stacks using a single array in C++:

```
 #include <iostream>
#define MAX 1000
class kStacks {
    int *arr;
    int *top;
    int *next;
    int n, k;
    int free;
public:
    kStacks(int k, int n);
    bool isFull() { return (free == -1); }
```

```
    bool isEmpty(int sn) { return (top[sn] == -1); }
    void push(int item, int sn);
    int pop(int sn);
};
kStacks::kStacks(int k, int n) {
    this->k = k;
    this->n = n;
    arr = new int[n];
    top = new int[k];
    next = new int[n];
    for (int i = 0; i < k; i++)
        top[i] = -1;
    free = 0;
    for (int i = 0; i < n - 1; i++)
        next[i] = i + 1;
    next[n - 1] = -1;
}
void kStacks::push(int item, int sn) {
    if (isFull()) {
        std::cout << "Stack Overflow\n";
        return;
    }
    int i = free;
```

```
    free = next[i];
    next[i] = top[sn];
    top[sn] = i;
    arr[i] = item;
}
int kStacks::pop(int sn) {
    if (isEmpty(sn)) {
        std::cout << "Stack Underflow\n";
        return -1;
    }
    int i = top[sn];
    top[sn] = next[i];
    next[i] = free;
    free = i;
    return arr[i];
}
int main() {
    int k = 3, n = 10;
    kStacks ks(k, n);
    ks.push(15, 2);
    ks.push(45, 2);
    ks.push(17, 1);
    ks.push(49, 1);
```

```
    ks.push(39, 1);
    ks.push(11, 0);
    ks.push(9, 0);
    ks.push(7, 0);
    std::cout << "Popped element from stack 2 is " << ks.pop(2) << std::endl;
    std::cout << "Popped element from stack 1 is " << ks.pop(1) << std::endl;
    std::cout << "Popped element from stack 0 is " << ks.pop(0) << std::endl;
    return 0;
     }
```

In the above implementation:

- **kStacks** class handles multiple stacks in a single array.
- **arr** is the array that stores the elements of all stacks.
- **top** is an array that stores the index of the top elements of all stacks.
- **next** is an array to maintain the next entry in all stacks and manage a free list.
- **free** is an index of the beginning of the free list.

1.12 Prefix, Infix, and Postfix:

In the context of arithmetic and algebraic expressions, **prefix**, **infix**, and **postfix** are different notations used to write expressions. These notations determine the order of operators and operands in an expression.

Infix Expression:

Infix notation is the most common and familiar way of writing expressions. In infix notation, operators are placed between the operands they operate on. This is the standard way humans write mathematical expressions.

- **Example**: **A + B**
- **More Complex Example**: **(A + B) * (C - D)**

Infix expressions require rules of precedence (order of operations) and parentheses to determine the order of evaluation.

Prefix Expression:

Prefix notation, also known as **Polish notation**, places the operator before its operands. This notation eliminates the need for parentheses to indicate the order of operations since the position of the operator determines the order.

- **Example**: **+ A B**
- **More Complex Example**: *** + A B - C D**

In this notation, the expression is evaluated from right to left.

Postfix Expression:

Postfix notation, also known as **Reverse Polish notation (RPN)**, places the operator after its operands. Like prefix notation, postfix notation does not require parentheses for determining the order of operations.

- **Example**: **A B +**
- **More Complex Example**: **A B + C D - ***

In this notation, the expression is evaluated from left to right.

1.13 Conversion and Evaluation:

Conversions between these notations involve understanding the order of operations and the use of stacks for managing operators and operands. Here's a brief overview of how each notation can be converted:

- **Infix to Prefix**: Move the operator before the operands while maintaining the order of operations.
- **Infix to Postfix**: Move the operator after the operands while maintaining the order of operations.
- **Prefix to Infix**: Insert the operator between the operands and add parentheses as needed.
- **Postfix to Infix**: Insert the operator between the operands and add parentheses as needed.

Evaluation Example

Let's evaluate an example for better understanding:

Infix: **(A + B) * (C - D)**

Prefix: *** + A B - C D**

1. **+ A B** → evaluate addition of A and B.
2. **- C D** → evaluate subtraction of D from C.

3. ***** → multiply results of step 1 and step 2.

Postfix: **A B + C D - ***

- **A B +** → evaluate addition of A and B.
- **C D -** → evaluate subtraction of D from C.
- ***** → multiply results of step 1 and step 2.

1.14 Applications of stacks:

1. Expression Evaluation
2. Expression Parsing
3. Function Call Management
4. Undo Mechanisms
5. Depth-First Search (DFS)
6. Backtracking Algorithms
7. Memory Management
8. Browser History Management
9. Expression Conversion
10. Tree Traversals
11. Balanced Parentheses
12. Runtime Stack

1.15 The limitations of Array representation of a Stack:

1. Fixed Size
2. Memory Inefficiency
3. Inflexibility
4. Limited Capacity
5. Inefficient Space Utilization
6. Potential Wastage
7. Complexity of Expansion
8. Contiguous Memory Requirement

Summary

While array-based stacks are straightforward and efficient for managing a fixed number of elements, their limitations make them less suitable for applications requiring dynamic resizing, unpredictable stack sizes, or efficient memory utilization. For these scenarios, alternative implementations such as linked-list-based stacks or dynamic array structures (e.g., using **std::vector** in C++) might be more appropriate.

Unit -2

List

Objective:

At the end of this session student shall be learning:

1. Understanding of Data Structures
2. Proficiency in List Operations
3. Application of Lists in Real-World Scenarios
4. Comparison of List Variants
5. Introduction to Advanced List Structures

Structure:

2.1 linked list:

A linked list is a data structure consisting of a sequence of elements, each containing a reference (or link) to the next element in the sequence. Unlike arrays, linked lists do not store elements in contiguous memory locations, which allows for efficient insertion and deletion operations.

2.2 Types of Linked Lists

1. **Singly Linked List:**
    - Structure: Each node contains data and a reference to the next node in the sequence.
    - Example: **Node -> Node -> Node -> NULL**
    - Operations: Insertion, deletion, and traversal are straightforward but can only be done in one direction.
2. **Doubly Linked List:**
    - Structure: Each node contains data, a reference to the next node, and a reference to the previous node.
    - Example: **NULL <- Node <-> Node <-> Node -> NULL**
    - Operations: Supports bidirectional traversal, making it easier to navigate backward and forward.
3. **Circular Linked List:**
    - Structure: The last node in the list points back to the first node, forming a circle.
    - Example: **Node -> Node -> Node -> Node -> ... -> Node**
    - Operations: Useful for applications where the list needs to be accessed in a circular manner.

2.3 Advantages of Linked Lists

- Dynamic Size: Linked lists can grow or shrink dynamically, unlike arrays which have a fixed size.
- Efficient Insertions/Deletions: Insertions and deletions are more efficient, especially for large lists, as they do not require shifting elements as in arrays.

2.4 Disadvantages of Linked Lists

- Memory Usage: Linked lists use extra memory for storing references to the next (and possibly previous) nodes.
- Access Time: Accessing elements takes linear time (**O(n)**) as opposed to constant time (**O(1)**) in arrays, due to sequential traversal.

2.5 Applications of Linked Lists

- Dynamic Memory Allocation: Used in applications requiring frequent insertions and deletions.
- Implementing Data Structures: Used to implement other data structures like stacks, queues, and graphs.
- Handling Polynomial Arithmetic: Efficiently manages polynomial operations where terms are inserted and deleted frequently.

2.6 Example Implementations

Here's a basic implementation of a singly linked list in C++:

```
 #include <iostream>
using namespace std;
class Node {
public:
    int data;
    Node* next;
    Node(int data) {
        this->data = data;
        this->next = nullptr;
    }
};
class LinkedList {
private:
    Node* head;

public:
    LinkedList() {
        head = nullptr;
    }
    void insertAtEnd(int data) {
        Node* newNode = new Node(data);
        if (head == nullptr) {
```

```
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }
    temp->next = newNode;
}
void deleteAtBegin() {
    if (head == nullptr) {
        cout << "List is empty" << endl;
        return;
    }
    Node* temp = head;
    head = head->next;
    delete temp;
}
void display() {
    if (head == nullptr) {
        cout << "List is empty" << endl;
        return;
    }
```

```
        Node* temp = head;
        while (temp != nullptr) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
};
int main() {
    LinkedList list;
    list.insertAtEnd(10);
    list.insertAtEnd(20);
    list.insertAtEnd(30);
    list.display();
    list.deleteAtBegin();
    list.display();
    return 0;
}
```

This example demonstrates basic operations like insertion, deletion, and traversal in a singly linked list. Similar principles can be applied to implement other types of linked lists, with adjustments to handle additional links and structures.

2.7 Stack Representation in Linked Lists:

- **Normal Stack Representation using a Singly Linked List**

  A normal stack implemented with a singly linked list involves having a top pointer that points to the last inserted element, which follows the Last In, First Out (LIFO) principle.

Implementation:

Here is an implementation of a normal stack using a singly linked list in C++:

```
#include <iostream>
class Node {
public:
    int data;
    Node* next;
    Node(int data) {
        this->data = data;
        this->next = nullptr;
    }
};
class Stack {
private:
    Node* top;
public:
    Stack() {
        top = nullptr;
    }
```

```
void push(int data) {
    Node* newNode = new Node(data);
    newNode->next = top;
    top = newNode;
    std::cout << data << " pushed onto stack\n";
}
void pop() {
    if (top == nullptr) {
        std::cout << "Stack Underflow\n";
        return;
    }
    Node* temp = top;
    top = top->next;
    std::cout << temp->data << " popped from stack\n";
    delete temp;
}
void display() {
    if (top == nullptr) {
        std::cout << "Stack is empty\n";
        return;
    }
    Node* temp = top;
    while (temp != nullptr) {
```

```
            std::cout << temp->data << " ";
            temp = temp->next;
        }
        std::cout << std::endl;
    }
};
int main() {
    Stack stack;
    stack.push(10);
    stack.push(20);
    stack.push(30);
    stack.display();
    stack.pop();
    stack.display();
    return 0;
}
```

- ## Circular Stack Representation using a Circular Linked List

  In a circular stack, the last node points back to the first node, creating a circular structure. This can be useful for applications that require circular traversal.

Implementation

Here is an implementation of a circular stack using a singly linked list in C++:

```
 #include <iostream>
class Node {
public:
    int data;
    Node* next;
    Node(int data) {
        this->data = data;
        this->next = nullptr;
    }
};
class CircularStack {
private:
    Node* top;
public:
    CircularStack() {
        top = nullptr;
    }
    void push(int data) {
        Node* newNode = new Node(data);
        if (top == nullptr) {
            top = newNode;
            newNode->next = top; // Point to itself to make it circular
        } else {
```

```
        Node* temp = top;
        while (temp->next != top) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = top;
        top = newNode;
    }
    std::cout << data << " pushed onto circular stack\n";
}
void pop() {
    if (top == nullptr) {
        std::cout << "Circular Stack Underflow\n";
        return;
    }
    if (top->next == top) { // Only one node in the stack
        std::cout << top->data << " popped from circular stack\n";
        delete top;
        top = nullptr;
    } else {
        Node* temp = top;
        while (temp->next != top) {
            temp = temp->next;
```

```
            }
            Node* delNode = top;
            temp->next = top->next;
            top = top->next;
            std::cout << delNode->data << " popped from circular stack\n";
            delete delNode;
        }
    }
    void display() {
        if (top == nullptr) {
            std::cout << "Circular Stack is empty\n";
            return;
        }
        Node* temp = top;
        do {
            std::cout << temp->data << " ";
            temp = temp->next;
        } while (temp != top);
        std::cout << std::endl;
    }
};
int main() {
    CircularStack cStack;
```

```
    cStack.push(10);
    cStack.push(20);
    cStack.push(30);
    cStack.display();
    cStack.pop();
    cStack.display();
    return 0;
}
```

2.8 Key Differences

- **Normal Stack**: Uses a singly linked list where the top pointer points to the most recently added element. The next pointer of the last node is nullptr.
- **Circular Stack**: Uses a circular linked list where the top pointer also points to the most recently added element, but the next pointer of the last node points back to the top, forming a circular structure.

2.9 Self-Organizing Lists

A self-organizing list is a type of list that reorders its elements based on access patterns to improve search efficiency over time. The primary goal is to reduce the average time complexity for searching elements that are accessed frequently.

Types of Self-Organizing Lists

1. **Move-to-Front (MTF)**
    - Mechanism: Whenever an element is accessed, it is moved to the front of the list.
    - Advantages: Frequently accessed elements are quickly found, as they tend to accumulate at the front of the list.
    - Disadvantages: Can be inefficient for elements that are accessed less frequently or accessed in a more uniform manner.

2. **Transpose**
    - Mechanism: When an element is accessed, it is swapped with the previous element.
    - Advantages: Provides a moderate reordering approach where frequently accessed elements gradually **move** to the front.
    - Disadvantages: Less aggressive than MTF, so it may take longer for frequently accessed elements to reach the front.
3. **Count**
    - Mechanism: Each element has a counter that tracks how many times it has been accessed. The list is periodically reordered based on these counters.
    - Advantages: Ensures elements are sorted based on their access frequency.
    - Disadvantages: Requires additional memory for counters and periodic reordering, which can be computationally expensive.

Applications

- Cache Management: Used in caching algorithms where frequently accessed items need to be quickly retrievable.
- Data Compression: Used in adaptive compression algorithms where certain symbols are accessed more frequently.
- Dynamic Search Optimization: In scenarios where the access patterns of data change over time, self-organizing lists can adapt to improve efficiency.

2.10 Skip Lists:

Skip lists are a probabilistic data structure that allows fast search, insertion, and deletion operations. They are an alternative to balanced trees and provide logarithmic time complexity for these operations.

Structure

- **Levels:** Skip lists consist of multiple levels of linked lists. The bottom level is an ordinary linked list, and each higher level acts as an "express lane" to skip over multiple elements.

- **Nodes:** Each node contains a key and multiple forward pointers, one for each level. The number of levels for each node is determined probabilistically.

Operations

1. **Search**
   - Mechanism: Starts at the highest level and moves forward until the next node's key is greater than the target key or the end of the list is reached. Then, it drops down a level and continues the search.
     - Time Complexity: Average case O(log n).
2. **Insertion**
   - Mechanism: Similar to search, but nodes are inserted at the appropriate positions at each level.
   - Time Complexity: Average case O(log n).
3. **Deletion**
   - Mechanism: Similar to search, but nodes are removed from each level.
   - Time Complexity: Average case O(log n).

Advantages

- Simplicity: Easier to implement and understand compared to balanced trees.
- Flexibility: The probabilistic nature allows for simple adjustments and dynamic updates without complex rebalancing.

Disadvantages

- Space Usage: Requires additional pointers, leading to higher memory consumption compared to simple linked lists.
- Randomness: Performance depends on randomization, which can occasionally lead to less optimal configurations.

Applications

- Databases: Used in databases for indexing to allow fast search, insertion, and deletion operations.
- Networking: Applied in routing algorithms where efficient path finding is crucial.
- Concurrent Computing: Adapted for concurrent data structures where multiple processes need to access and modify the list efficiently.

Unit 3

Queues

Objective:

At the end of this session students shall be learning:

- " Introduction of Principle of OOP's concept
- Understand Object Oriented Methodology
- Overview of procedure Oriented programming
- Definition of Object Oriented Programming.
- Object Oriented Languages"

Structure:

3.1 Queues: Array and Linked Representation

3.2 Linked List Representation of Queues

3.3 Priority Queue: Concepts and Implementations

3.4 Recursion

3.5 Internal Stack Implementation in Recursion

3.6 Benefits of "OOP"

3.7 Object Oriented Language

3.8 Application of "OOP"

3.1 Queues: Array and Linked Representation

Queues are a fundamental data structure in computer science, operating on a First-In-First-Out (FIFO) principle. This means that the first element added to the queue will be the first one to be removed. There are two primary ways to implement queues: using arrays and using linked lists. Each implementation has its advantages and disadvantages.

3.1.1 Array Representation of Queues

In an array-based queue, the queue is typically implemented using a fixed-size array along with two pointers or indices: **front** and **rear**.

Structure:

1. **Array:** A fixed-size array to store the elements.
2. **Front:** An index to the front of the queue.
3. **Rear:** An index to the rear of the queue.

Operations:

1. **Enqueue (Insertion):**
    - Check if the queue is full. If the rear is at the last index, the queue is full.
    - If not full, increment rear and insert the new element at the rear position.

Example:

```
  def enqueue(queue, element):
if queue['rear'] == len(queue['array']) - 1:
 print("Queue is full")
else:
   if queue['front'] == -1:  # If queue is empty
   queue['front'] = 0
   queue['rear'] += 1
   queue['array'][queue['rear']] = elemen
```

3.1.2 Dequeue (Removal):

- Check if the queue is empty. If the front is -1 or front is greater than rear, the queue is empty.
- If not empty, return the element at front and increment front.

Example:

```
def dequeue(queue):
  if queue['front'] == -1 or queue['front'] > queue['rear']:
     print("Queue is empty")
     return None
  else:
     element = queue['array'][queue['front']]
     queue['front'] += 1
     return element
```

3.2 Linked List Representation of Queues

In a linked list-based queue, the queue is implemented using nodes where each node has a value and a reference to the next node.

Structure:

1. **Node:** Each node contains an element and a pointer to the next node.
2. **Front:** A pointer to the front node of the queue.
3. **Rear:** A pointer to the rear node of the queue.

Example:

```
class Node:
  def __init__(self, value):
     self.value = value
     self.next = None
```

3.3 Priority Queue: Concepts and Implementations

A priority queue is an advanced type of queue in which each element is associated with a priority. Elements are dequeued based on their priority rather than the order they were enqueued. Elements with higher priority are dequeued before those with lower priority. In order to preserve the First-In-First-Out (FIFO) order for items with equal priority, two elements with the same priority are dequeued in accordance with their arrival order.

3.3.1 Array-based Implementation

In an array-based priority queue, the array is used to store elements along with their priorities. The array can be sorted or unsorted.

1. Unsorted Array:

- **Enqueue:** O(1) time complexity. Simply add the element to the end of the array.
- **Dequeue:** O(n) time complexity. Search through the array to find the element with the highest priority and remove it.

Example:

```
class UnsortedPriorityQueue:
  def __init__(self):
    self.queue = []

  def enqueue(self, element, priority):
    self.queue.append((element, priority))

  def dequeue(self):
    if not self.queue:
      print("Queue is empty")
      return None
    max_priority_index = 0
    for i in range(1, len(self.queue)):
      if self.queue[i][1] > self.queue[max_priority_index][1]:
        max_priority_index = i
    return self.queue.pop(max_priority_index)

  def peek(self):
```

```
        if not self.queue:
            print("Queue is empty")
            return None
        max_priority_index = 0
        for i in range(1, len(self.queue)):
            if self.queue[i][1] > self.queue[max_priority_index][1]:
                max_priority_index = i
        return self.queue[max_priority_index]

    def is_empty(self):
        return len(self.queue) == 0
```

3.3.2 Linked List-based Implementation

In a linked list-based priority queue, each node contains an element and its priority.

1. Unsorted Linked List:

- Enqueue: O(1) time complexity. Insert the new node at the head of the list.
- Dequeue: O(n) time complexity. Traverse the list to find and remove the node with the highest priority.

Example:

```
class Node:
    def __init__(self, element, priority):
        self.element = element
        self.priority = priority
        self.next = None

class UnsortedLinkedListPriorityQueue:
    def __init__(self):
        self.head = None

    def enqueue(self, element, priority):
        new_node = Node(element, priority)
        new_node.next = self.head
        self.head = new_node

    def dequeue(self):
        if not self.head:
            print("Queue is empty")
```

```
            return None
        max_priority_node = self.head
        max_priority_node_prev = None
        current = self.head
        prev = None
        while current:
            if current.priority > max_priority_node.priority:
                max_priority_node = current
                max_priority_node_prev = prev
            prev = current
            current = current.next
        if max_priority_node_prev:
            max_priority_node_prev.next = max_priority_node.next
        else:
            self.head = max_priority_node.next
        return max_priority_node.element

    def peek(self):
        if not self.head:
            print("Queue is empty")
            return None
        max_priority_node = self.head
        current = self.head
        while current:
            if current.priority > max_priority_node.priority:
                max_priority_node = current
            current = current.next
        return max_priority_node.element

    def is_empty(self):
        return self.head is None
```

3.4 Recursion:

Programming recursion is the process by which a function calls itself to address progressively smaller versions of the same problem. This approach can simplify the code for problems that have a natural recursive structure, such as those involving hierarchical or repetitive processes.

3.4.1 How Recursion Works

In a recursive function, there are two main components:

1. Base Case: This is the condition under which the recursion ends. Without a base case, the function would call itself indefinitely, leading to infinite recursion and eventually a stack overflow.
2. Recursive Case: This is where the function calls itself with a smaller or simpler argument, moving towards the base case.

3.4.2 Example of a Recursive Function: Factorial

The factorial of a non-negative integer nnn is the product of all positive integers less than or equal to nnn. It is denoted as n!n!n!.

n!=n×(n−1)×(n−2)×⋯×1n! = n ₩times (n-1) ₩times (n-2) ₩times ₩cdots ₩times 1n!=n×(n−1)×(n−2)×⋯×1

For instance:

- 5!=5×4×3×2×1=1205! = 5 \times 4 \times 3 \times 2 \times 1 = 1205!=5×4×3×2×1=120
- 3!=3×2×1=63! = 3 \times 2 \times 1 = 63!=3×2×1=6

Using recursion, the factorial can be defined as:

- **Base Case:** 0!=10! = 10!=1
- **Recursive Case:** n!=n×(n−1)!n! = n ₩times (n-1)!n!=n×(n−1)!

3.4.3 Advantages of Recursion

1. **Simplified Code:** Recursive solutions can be more concise and easier to understand, especially for problems that have a natural recursive structure, such as tree traversals, factorial calculation, and the Fibonacci sequence.
    - Example: Calculating the factorial using recursion is straightforward and mirrors the mathematical definition.
2. **Elegant Solutions:** Recursive functions can often provide elegant solutions to complex problems. This is particularly true for problems involving divide-and-conquer strategies, like quicksort and mergesort.

3. **Reduced Code Complexity:** For problems like directory traversal or solving puzzles (e.g., Towers of Hanoi), recursion can reduce the complexity of the code compared to iterative solutions.

3.4.4 Disadvantages of Recursion

1. **Performance Overhead:** Recursive calls involve function call overhead and use more stack space. Each recursive call adds a new frame to the call stack, which can lead to higher memory usage and slower execution.
   - Example: Calculating large factorials recursively can quickly exhaust the stack, leading to a stack overflow error.
2. **Stack Overflow:** If the recursion depth is too large (i.e., too many recursive calls), it can lead to a stack overflow, where the program runs out of stack memory.
   - Example: Deeply nested recursive calls without an adequate base case can cause the program to crash.
3. **Difficult to Debug:** Recursive functions can be harder to debug and understand, especially when the recursion depth is deep. It can be challenging to track the flow of recursive calls and their respective states.

3.4.5 Example of Recursion: Fibonacci Sequence

The Fibonacci sequence is another classic example of recursion. Each number in the sequence is the sum of the two preceding ones, usually starting with 0 and 1.

$$F(n)=F(n-1)+F(n-2)F(n) = F(n-1) + F(n-2)F(n)=F(n-1)+F(n-2)$$

With the base cases:

- $F(0)=0F(0) = 0F(0)=0$
- $F(1)=1F(1) = 1F(1)=1$

3.5 Internal Stack Implementation in Recursion

When a function is called, the system uses a stack data structure to manage the function calls. This stack is known as the **call stack**. Each time a function is invoked, a **stack frame** is created

to store the function's local variables, arguments, return address, and the state of the function. When the function completes, its stack frame is popped off the stack.

3.5.1 How the Call Stack Works in Recursion

In recursion, multiple stack frames are created as the function calls itself. Understanding how the call stack operates helps in grasping the internal workings of recursion.

1. **Function Call:**
   - A new stack frame is added to the call stack whenever a recursive function is invoked.
   - The parameters of the function, local variables, and the return address are all listed in this stack frame.
2. **Base Case:**
   - The recursion progresses until it reaches the base case.
   - Once the base case is met, the function begins to return, and stack frames start to be popped off.
3. **Returning from Recursion:**
   - As each recursive call returns, its corresponding stack frame is popped off the stack.
   - The function resumes execution from the point where it was paused, using the saved return address.

3.5.2 Example: Factorial Function with Stack Frames

Consider the factorial function implemented recursively. The factorial of n (denoted as n!n!n!) is defined as:

n!=n×(n−1)×(n−2)×⋯×1n! = n ₩times (n-1) ₩times (n-2) ₩times ₩cdots ₩times 1n!=n×(n−1)×(n−2)×⋯×1

For n=5n = 5n=5:

Here is the recursive implementation:

5!=5×4×3×2×1=1205! = 5 \times 4 \times 3 \times 2 \times 1 = 1205!=5×4×3×2×1=120

```
def factorial(n):
    if n == 0:  # Base case
```

```
    return 1
else:  # Recursive case
    return n * factorial(n - 1)
```

Unit 4

Tree

Objective:

At the end of this session students shall be learning:

- " Introduction of Principle of OOP's concept
- Understand Object Oriented Methodology
- Overview of procedure Oriented programming
- Definition of Object Oriented Programming.
- Object Oriented Languages"

Structure:

4.1 Introduction to Trees as a Data Structure

A tree is a widely used data structure that simulates a hierarchical tree structure with a set of connected nodes. Each node contains a value and references to other nodes (its children). Trees are used in various applications, including databases, file systems, and network routing.

Key Terminology

1. Root: The topmost node of a tree.
2. Node: An element of the tree that contains a value.
3. Parent: A node that has one or more child nodes.
4. Child: A node that has a parent node.
5. Leaf: A node with no children.
6. Subtree: A tree formed by a node and its descendants.
7. Height: The number of edges on the longest path from the root to a leaf.
8. Depth: The number of edges from the root to a node.

4.2 Binary Tree

A binary tree is a type of tree where each node has at most two children, referred to as the left child and the right child. This structure is particularly useful for implementing binary search trees, heaps, and other algorithms.

4.1.1 Binary Tree Operations

1. **Insertion**
2. **Deletion**
3. **Traversal**

4.1.2 Insertion in a Binary Tree

Insertion in a binary tree involves adding a new node at the first available position in level order (left to right).

4.1.3 Algorithm for Insertion:

1. Create a new node.
2. Use a queue to perform level order traversal.

3. Find the first node that does not have two children.
4. Insert the new node as a left or right child of the node found in step 3.

4.1.4 Example Implementation:

```
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.value = key
def insert(root, key):
    if root is None:
        return Node(key)
    queue = [root]
    while queue:
        temp = queue.pop(0)
        if not temp.left:
            temp.left = Node(key)
            break
        else:
            queue.append(temp.left)
        if not temp.right:
            temp.right = Node(key)
            break
        else:
            queue.append(temp.right)
```

```
    return root
```

4.1.5 Deletion in a Binary Tree

Deletion in a binary tree involves removing a node and ensuring the tree remains a valid binary tree after the deletion.

4.1.6 Algorithm for Deletion:

1. Find the deepest and rightmost node in the binary tree.
2. Replace the value of the node to be deleted with the deepest node's value.
3. Delete the deepest node.

4.1.7 Example Implementation:

```
def delete_deepest(root, d_node):
    queue = [root]
    while queue:
        temp = queue.pop(0)
        if temp is d_node:
            temp = None
            return
        if temp.right:
            if temp.right is d_node:
                temp.right = None
                return
            else:
                queue.append(temp.right)
        if temp.left:
            if temp.left is d_node:
```

```
            temp.left = None
                return
            else:
                queue.append(temp.left)
def deletion(root, key):
    if root is None:
        return None
    if root.left is None and root.right is None:
        if root.value == key:
            return None
        else:
            return root
    key_node = None
    queue = [root]
    while queue:
        temp = queue.pop(0)
        if temp.value == key:
            key_node = temp
        if temp.left:
            queue.append(temp.left)
        if temp.right:
            queue.append(temp.right)
    if key_node:
```

```
        x = temp.value
        delete_deepest(root, temp)
        key_node.value = x
    return root
```

4.3 Traversal in a Binary Tree

Traversal is the process of visiting all the nodes of a binary tree in a specific order. The most common traversal methods are:

1. In-order Traversal: Left subtree, Root, Right subtree
2. Pre-order Traversal: Root, Left subtree, Right subtree
3. Post-order Traversal: Left subtree, Right subtree, Root
4. Level-order Traversal: Level by level from top to bottom

4.3.1 Example Implementations:

In-order Traversal:

```
def inorder_traversal(root):
    if root:
        inorder_traversal(root.left)
        print(root.value, end=' ')
        inorder_traversal(root.right)
```

Pre-order Traversal:

```
def preorder_traversal(root):
    if root:
        print(root.value, end=' ')
        preorder_traversal(root.left)
        preorder_traversal(root.right)
```

Post-order Traversal:

```
def postorder_traversal(root):
    if root:
        postorder_traversal(root.left)
        postorder_traversal(root.right)
        print(root.value, end=' ')
```

Level-order Traversal:

```
def level_order_traversal(root):
    if root is None:
        return
    queue = [root]
    while queue:
        temp = queue.pop(0)
        print(temp.value, end=' ')
        if temp.left:
            queue.append(temp.left)
        if temp.right:
            queue.append(temp.right)
```

4.4 Threaded Binary Tree

A Threaded Binary Tree is a type of binary tree where nodes are arranged to make in-order traversal faster and more efficiently. In a threaded binary tree, the null pointers are replaced with pointers to the in-order predecessor or successor, thus "threading" through the binary tree. This allows traversal without using a stack or recursion.

4.4.1 Types of Threaded Binary Trees

1. Single Threaded: Every node threads toward either its predecessor or successor in the order it appears.
2. Every node has two threads, one for each of its predecessors and successors in the order they appear in the list.

4.4.2 Insertion in a Threaded Binary Tree

Inserting a node in a threaded binary tree involves updating the existing threads to maintain the structure.

Algorithm for Insertion:

1. Find the correct position for the new node in the binary tree.
2. Update the threads: Adjust the threads of the predecessor and successor nodes to point to the new node.

Example Implementation:

```
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.value = key
        self.lthread = False
        self.rthread = False
def insert(root, key):
    parent = None
    current = root
    while current:
        parent = current
        if key < current.value:
            if current.lthread:
```

```
            break
        current = current.left
    else:
        if current.rthread:
            break
        current = current.right
new_node = Node(key)
if not parent:
    root = new_node
elif key < parent.value:
    new_node.left = parent.left
    new_node.right = parent
    parent.lthread = False
    parent.left = new_node
else:
    new_node.left = parent
    new_node.right = parent.right
    parent.rthread = False
    parent.right = new_node
return root
```

4.4.3 Deletion in a Threaded Binary Tree

Deleting a node in a threaded binary tree requires careful handling of threads to maintain the structure.

Algorithm for Deletion:

1. **Find the node** to be deleted.
2. **Handle three cases**:
    - Node with no children (leaf node)
    - Node with one child
    - Node with two children
3. **Update the threads** of predecessor and successor nodes to bypass the deleted node.

Example Implementation:

```
def delete(root, key):
    parent = None
    current = root
    while current:
        if key == current.value:
            break
        parent = current
        if key < current.value:
            if current.lthread:
                return root
            current = current.left
        else:
            if current.rthread:
                return root
            current = current.right
    if not current.left and not current.right:
        if not parent:
            root = None
```

```
    elif parent.left == current:
        parent.left = current.left
        parent.lthread = True
    else:
        parent.right = current.right
        parent.rthread = True
elif not current.left:
    if not parent:
        root = current.right
    elif parent.left == current:
        parent.left = current.right
    else:
        parent.right = current.right
elif not current.right:
    if not parent:
        root = current.left
    elif parent.left == current:
        parent.left = current.left
    else:
        parent.right = current.left
else:
    succ_parent = current
    succ = current.right
```

```
        while not succ.lthread:
            succ_parent = succ
            succ = succ.left
        current.value = succ.value
        if succ_parent.left == succ:
            succ_parent.left = succ.right
        else:
            succ_parent.right = succ.right
    return root
```

4.5 Height-Balanced Tree: AVL Tree

An AVL Tree, also known as an Adelson-Velsky and Landis Tree, is a self-balancing binary search tree in which there is never more than one height difference between any node's left and right subtrees. This height balance ensures that the tree maintains O(log n) time complexity for insertion, deletion, and lookup operations.

AVL Tree Operations

1. **Insertion**
2. **Deletion**
3. **Rotations**
4. **Traversal**

4.5.1 Insertion in an AVL Tree

Insertion in an AVL tree is similar to insertion in a binary search tree, followed by balancing the tree if it becomes unbalanced.

Algorithm for Insertion:

1. Perform standard BST insertion.
2. Update the height of the current node.
3. Determine the balancing factor by dividing the height of the left and right subtrees.
4. Rotations should be used to rebalance the tree if the balance factor is more than or less than -1.

Example:

```
 class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.value = key
        self.height = 1
def get_height(node):
    if not node:
        return 0
    return node.height
def get_balance(node):
    if not node:
        return 0
    return get_height(node.left) - get_height(node.right)
def right_rotate(y):
    x = y.left
    T2 = x.right
    x.right = y
    y.left = T2
    y.height = max(get_height(y.left), get_height(y.right)) + 1
    x.height = max(get_height(x.left), get_height(x.right)) + 1
    return x
```

```
def left_rotate(x):
    y = x.right
    T2 = y.left
    y.left = x
    x.right = T2
    x.height = max(get_height(x.left), get_height(x.right)) + 1
    y.height = max(get_height(y.left), get_height(y.right)) + 1
    return y
def insert(root, key):
    if not root:
        return Node(key)
    elif key < root.value:
        root.left = insert(root.left, key)
    else:
        root.right = insert(root.right, key)
    root.height = max(get_height(root.left), get_height(root.right)) + 1
    balance = get_balance(root)
    if balance > 1 and key < root.left.value:
        return right_rotate(root)
    if balance < -1 and key > root.right.value:
        return left_rotate(root)
    if balance > 1 and key > root.left.value:
        root.left = left_rotate(root.left)
```

```
        return right_rotate(root)
    if balance < -1 and key < root.right.value:
        root.right = right_rotate(root.right)
        return left_rotate(root)
    return root
```

4.5.2 Deletion in an AVL Tree

Deletion in an AVL tree involves removing the node and then balancing the tree if it becomes unbalanced.

Algorithm for Deletion:

1. Execute a typical BST delete.
2. Adjust the current node's height.
3. Determine the component of balance.
4. Rotations should be used to rebalance the tree if the balance factor is more than or less than -1.

Example:

```
 def min_value_node(node):
    current = node
    while current.left:
        current = current.left
    return current
def delete(root, key):
    if not root:
        return root
    elif key < root.value:
        root.left = delete(root.left, key)
```

```
elif key > root.value:
    root.right = delete(root.right, key)
else:
    if root.left is None:
        temp = root.right
        root = None
        return temp
    elif root.right is None:
        temp = root.left
        root = None
        return temp
    temp = min_value_node(root.right)
    root.value = temp.value
    root.right = delete(root.right, temp.value)
if root is None:
    return root
root.height = max(get_height(root.left), get_height(root.right)) + 1
balance = get_balance(root)
if balance > 1 and get_balance(root.left) >= 0:
    return right_rotate(root)
if balance > 1 and get_balance(root.left) < 0:
    root.left = left_rotate(root.left)
    return right_rotate(root)
```

```
    if balance < -1 and get_balance(root.right) <= 0:
        return left_rotate(root)
    if balance < -1 and get_balance(root.right) > 0:
        root.right = right_rotate(root.right)
        return left_rotate(root)
    return root
```

4.5.3 Rotations in AVL Tree

Rotations are used to rebalance the tree when it becomes unbalanced.

1. Right Rotation: Applied when the left subtree is taller.
2. Left Rotation: Applied when the right subtree is taller.
3. Rotation from the Left to the Right: A combination of left and right rotations.
4. Rotation from the Right to the Left: A combination of rotation from the Right to the Left.

4.5.4 Traversal in an AVL Tree

Traversal methods in an AVL tree are the same as in a binary search tree. The common methods are in-order, pre-order, and post-order traversals.

Unit 5

Searching and Sorting

Objective:

At the end of this session students shall be learning:

- " Introduction of Principle of OOP's concept
- Understand Object Oriented Methodology
- Overview of procedure Oriented programming
- Definition of Object Oriented Programming.
- Object Oriented Languages"

Structure:

5.1 Searching and Sorting with Linear Search

5.2 Binary Search

5.3 Comparison of Linear Search and Binary Search

5.4 Sorting Algorithms: Selection Sort, Insertion Sort, and Shell Sort

5.5 Comparison of Sorting Technique

5.1 Searching and Sorting with Linear Search

Searching and sorting are fundamental operations in computer science that are essential for efficiently managing and retrieving data. Here's an overview of two common searching algorithms: Linear Search and Binary Search, along with a comparison of their characteristics.

Linear Search is the simplest searching algorithm. It sequentially checks each element of the list until the desired element is found or the list ends.

5.1.1 Algorithm for Linear Search:

1. Begin with the list's first entry.
2. Make a comparison between the target and current elements.
3. Return the location of the current element if it is the target.
4. Proceed to the next element if the current element is not the target.
5. Until the target element is located or the list is finished, repeat steps 2-4.

Example:

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1
```

5.1.2 Characteristics:

- **Time Complexity:** O(n), where n is the number of elements in the list.
- **Space Complexity:** O(1).
- **Use Case:** Works well with small or unsorted lists.

5.2 Binary Search

A more effective technique is Binary Search, however it needs the list to be sorted beforehand. The list is divided in half repeatedly, cutting the search period in half each time.

5.2.1 Algorithm for Binary Search:

1. Start with two pointers: one at the beginning (low) and one at the end (high) of the list.
2. Calculate the middle index.
3. Compare the middle element with the target element.
4. If the middle element is the target, return its position.
5. If the target is smaller than the middle element, search the left half.
6. If the target is larger than the middle element, search the right half.
7. Repeat steps 2-6 until the target element is found or the search interval is empty.

Example Implementation:

```
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

5.2.2 Characteristics:

- **Time Complexity:** O(log n), where n is the number of elements in the list.
- **Space Complexity:** O(1) for iterative, O(log n) for recursive.
- **Use Case:** Suitable for large, sorted lists.

5.3 Comparison of Linear Search and Binary Search

Feature	Linear Search	Binary Search
Complexity	O(n)	O(log n)
Data Requirement	Unsorted or sorted	Sorted
Implementation	Simple	Slightly more complex
Performance	Slower for large datasets	Faster for large datasets
Use Case	Small lists or unsorted data	Large, sorted lists
Memory Usage	O(1)	O(1) iterative, O(log n) recursive
Advantages	Easy to implement, no sorting needed	Efficient for large datasets
Disadvantages	Inefficient for large datasets	Requires sorted data

Fig 5.1 Comparison Chart

5.4 Sorting Algorithms: Selection Sort, Insertion Sort, and Shell Sort

Sorting is a fundamental operation in computer science used to arrange data in a specific order, typically in ascending or descending order. Here, we discuss three common sorting algorithms: Selection Sort, Insertion Sort, and Shell Sort, along with a comparison of their characteristics.

5.4.1 Selection Sort

Choosing Sort is a basic sorting algorithm that relies on comparisons. The input list is split into two sections: the sorted portion on the left and the unsorted portion on the right. The list is initially unsorted and the sorted portion is empty.

Algorithm for Selection Sort:

1. Iterate over the list.
2. Find the minimum element in the unsorted part of the list.
3. Swap the minimum element with the first element of the unsorted part.

4. Move the boundary between the sorted and unsorted parts one element to the right.
5. Repeat steps 1-4 until the entire list is sorted.

Example Implementation:

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr
```

5.4.2 Characteristics:

- Time Complexity: $O(n^2)$
- Space Complexity: O(1)
- Stability: Not stable
- Use Case: Simple to implement, useful for small lists

5.4.3 Insertion Sort

Insertion Sort is a comparison-based sorting algorithm that builds the final sorted array one item at a time. It is much like sorting playing cards in your hands.

5.4.4 Algorithm for Insertion Sort:

1. From the second member to the last, go through the list iteratively.
2. Compare each element with those in the sorted section of the list.
3. Place the element in the sorted section at the appropriate location.
4. Continue steps 2-3 for every element in the list until it is sorted.

Example Implementation:

```
def insertion_sort(arr):
```

```
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr
```

5.4.5 Characteristics:

- Time Complexity: $O(n^2)$ worst case, O(n) best case
- Space Complexity: O(1)
- Stability: Stable
- Use Case: Efficient for small or nearly sorted lists

5.4.5 Shell Sort

Shell Sort is an in-place comparison-based sorting algorithm. It is an extension of insertion sort that makes distant items interchangeable. The goal is to organize the list of items so that a sorted list may be obtained by taking every h-th member starting from any position.

5.4.6 Algorithm for Shell Sort:

1. Choose a sequence of gaps (e.g., n/2, n/4, ..., 1).
2. For each gap, perform a gapped insertion sort.
3. Reduce the gap and repeat step 2 until the gap is 1.
4. The list is sorted when the gap is reduced to 1.

Example

```
def shell_sort(arr):
    n = len(arr)
    gap = n // 2
    while gap > 0:
```

```
    for i in range(gap, n):
        temp = arr[i]
        j = i
        while j >= gap and arr[j - gap] > temp:
            arr[j] = arr[j - gap]
            j -= gap
        arr[j] = temp
    gap //= 2
return arr
```

5.4.7 Characteristics:

- Time Complexity: Depends on the gap sequence, typically O(n log n) to $O(n^2)$
- Space Complexity: O(1)
- Stability: Not stable
- Use Case: More efficient than insertion sort for medium-sized lists

5.5 Comparison of Sorting Techniques

Feature	Selection Sort	Insertion Sort	Shell Sort
Time Complexity	$O(n^2)$	$O(n^2)$ worst, O(n) best	O(n log n) to $O(n^2)$ depending on gap sequence
Space Complexity	O(1)	O(1)	O(1)
Stability	Not stable	Stable	Not stable
In-Place	Yes	Yes	Yes
Use Case	Simple, small lists	Small or nearly sorted lists	Medium-sized lists
Implementation	Easy to implement	Easy to lement	More complex but efficient

Fig 5.2 Comparison Chart

REFERENCES

1. "Accelerated C++: Practical Programming by Example" by Andrew Koenig and Barbara E. Moo
2. "Effective C++: 55 Specific Ways to Improve Your Programs and Designs" by Scott Meyers
3. "C++ Primer" by Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo
4. "The C++ Programming Language" by Bjarne Stroustrup
5. "C++ Concurrency in Action: Practical Multithreading" by Anthony Williams
6. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
7. "Algorithms" by Robert Sedgewick and Kevin Wayne
8. "Data Structures and Algorithm Analysis in C++" by Mark Allen Weiss
9. "Problem Solving with Algorithms and Data Structures Using C++" by Brad Miller and David Ranum
10. "Data Structures and Algorithm Analysis in C++" by Mark Allen Weiss (Book)
11. Journals like IEEE Transactions on Computers, ACM Transactions on Programming Languages and Systems, and Journal of the ACM
12. "C++ Templates: The Complete Guide" by David Vandevoorde, Nicolai M. Josuttis,
13. "C++ Standard Library: A Tutorial and Reference" by Nicolai M. Josuttis
14. "Modern C++ Programming Cookbook: Implement modern C++ features with ease" by Marius Bancila
15. "Programming: Principles and Practice Using C++" by Bjarne Stroustrup
16. "C++ Concurrency in Action, Second Edition: Practical Multithreading" by Anthony Williams
17. "Data Structures and Algorithms in C++" by Michael T. Goodrich, Roberto Tamassia, and David M. Mount
18. "Algorithm Design Manual" by Steven S. Skiena
19. "Competitive Programming 3" by Steven Halim and Felix Halim
20. "Cracking the Coding Interview: 189 Programming Questions and Solutions" by Gayle Laakmann McDowell
21. "Introduction to the Theory of Computation" by Michael Sipser