MATS UNIVERSITY

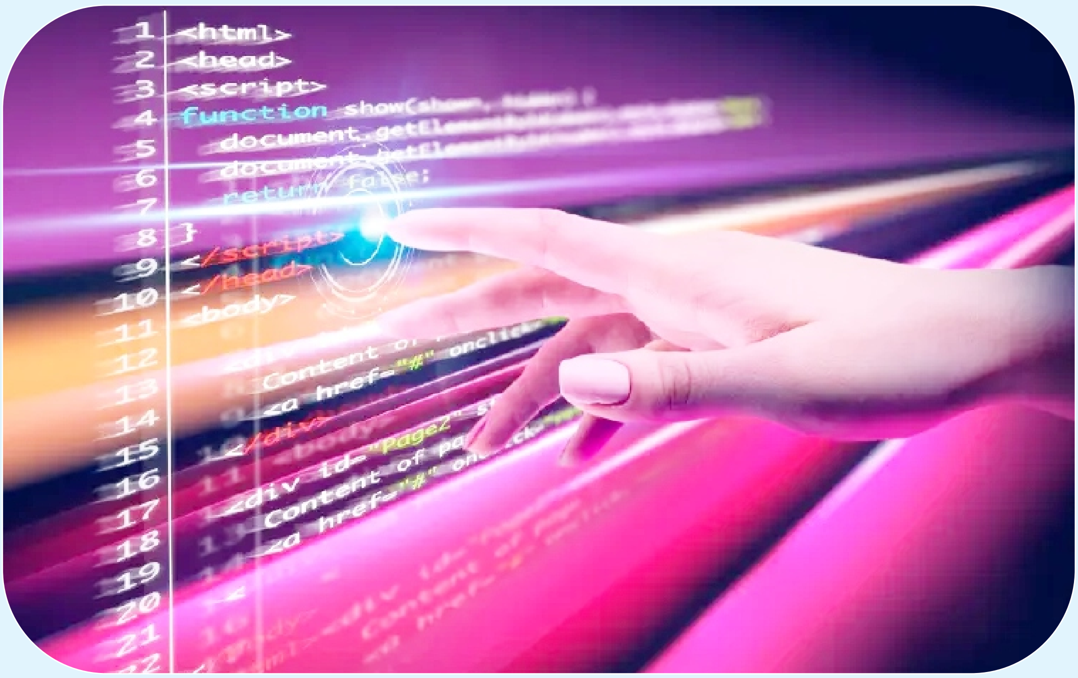
NAAC GRADE A+ ACCREDITED UNIVERSITY

MATS CENTRE FOR OPEN & DISTANCE EDUCATION

Data Structure Concepts

Master of Computer Applications (MCA)

Semester - 1



SELF LEARNING MATERIAL

MATS UNIVERSITY
www.matsuniversity.ac.in
NAAC GRADE A+ ACCREDITED UNIVERSITY

Master of Computer Applications

ODL **MCA 105**

Data Structure Concepts

COURSE DEVELOPMENT EXPERT COMMITTEE

COURSE COORDINATOR

Dr. Poonam Singh, Associate Professor, MATS University, Raipur, Chhattisgarh

COURSE PREPARATION

Dr. Poonam Singh, Associate Professor and Mr. Sanjay Behara, Assistant Professor, MATS University, Raipur, Chhattisgarh

March, 2025

ISBN: 978-93-49916-15-9

@MATS Centre for Distance and Online Education, MATS University, Village- Gullu, Aarang, Raipur-(Chhattisgarh)

Printed & Published on behalf of MATS University, Village-Gullu, Aarang, Raipur by Mr. Meghanadhudu Katabathuni, Facilities & Operations, MATS University, Raipur (C.G.)

Disclaimer-Publisher of this printing material is not responsible for any error or dispute from contents of this course material, this is completely depends on AUTHOR’S MANUSCRIPT.

Printed at: The Digital Press, Krishna Complex, Raipur-492001(Chhattisgarh)

Acknowledgement

The material (pictures and passages) we have used is purely for educational purposes. Every effort has been made to trace the copyright holders of material reproduced in this book. Should any infringement have occurred, the publishers and editors apologize and will be pleased to make the necessary corrections in future editions of this book.

COURSE INTRODUCTION

Data Structures is a fundamental subject in computer science that focuses on organizing, storing, and managing data efficiently. It plays a crucial role in algorithm development and problem-solving. Understanding data structures enables efficient memory usage, quick data retrieval, and optimized computational performance. This course covers various types of data structures, including linear and nonlinear structures, along with their applications in real-world scenarios.

Module 1: Linear Data Structures

This Unit introduces the basic concept of linear data structures, where data elements are arranged sequentially. It covers arrays and linked lists, their operations (insertion, deletion, traversal, searching, and sorting), and their applications. The comparison between static and dynamic memory allocation is also discussed.

Module 2: Stack, Queue, and Recursion

In this Unit, we explore stack and queue, two important linear data structures with different access methods.

- Stack follows the LIFO (Last In, First Out) principle, supporting operations like push, pop, and peek.
- Queue follows the FIFO (First In, First Out) principle, with operations like enqueue and dequeue. Variants such as circular queue, priority queue, and deque are also discussed.
- Recursion, a method where a function calls itself, is introduced along with its applications and differences from iteration.

Module 3: Linked Lists

This Unit focuses on linked lists, a dynamic data structure where elements (nodes) are connected through pointers. Different types of linked lists—singly linked list, doubly linked list, and circular linked list—are discussed in detail, along with operations like insertion, deletion, searching, and traversal. Their advantages over arrays and real-world applications are also covered.

Module 4: Trees and Graphs

This Unit introduces hierarchical and non-linear data structures:

- Trees, including binary trees, binary search trees (BST), and tree traversals (preorder, inorder, postorder). Applications in hierarchical data representation are explored.
- Graphs, including representations (adjacency matrix and adjacency list), traversal techniques (BFS and DFS), and applications in networking and pathfinding.

Module 5: Algorithm Analysis and Design

This Unit focuses on the efficiency of algorithms using asymptotic notations (Big O, Theta, and Omega). Different algorithm design techniques such as divide and conquer, greedy algorithms, dynamic programming, and backtracking are introduced. The importance of selecting appropriate data structures for optimizing algorithm performance is also discussed.

MODULE 1
LINEAR DATA STRUCTURES

LEARNING OUTCOMES

By the end of this Unit, students will be able to:

- Understand data structure concepts, data types, and abstract data types (ADTs) and their role in programming.
- Explain linear data structures using sequential organization, including their operations and applications.
- Learn about arrays, their classification, properties, representation, and memory allocation.
- Implement searching algorithms (Linear Search, Binary Search) for efficient data retrieval.
- Apply sorting algorithms (Insertion Sort, Selection Sort, and Merge Sort) to organize data effectively.

Unit 1: Data structure concepts And Linear data structures

1.1 Data structure concepts, Data type, and Abstract data type

Data structures are essential elements of computer science that facilitate the efficient storage, organization, and management of data. They provide representation of data in memory as well as insertion, deletion, and searching in-memory operations. A data structure defines an algorithm's efficiency, making it the essential concept for optimizing performance. Prevalent data structures encompass arrays, linked lists, stacks, queues, trees, graphs, and hash tables. Each of these architectures has unique advantages and disadvantages depending on particular application.

Data Type

But in a way, it lists resources that a variable can store in a programming language. It delineates the permissible values for a variable and the procedures applicable to those values. Data types can be categorized into primitive kinds and non-primitive types. Primitive data types are types such as integers, floating-point numbers, characters, and booleans that encapsulate a singular value. Non-primitive data types, including arrays, structures, and classes, hold multiple values or complicated data. Also choose the correct data type to make sure of memory and logic correctness in the program.

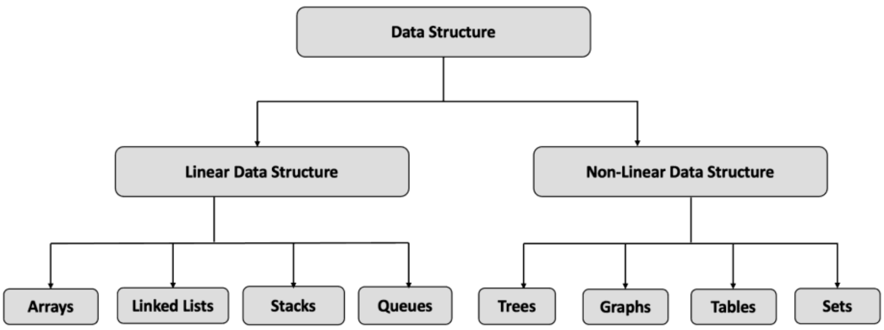


Figure 1.1: Data Structure Type
[Source: https://technologystrive.com/]

Abstract Data Type (ADT)

An abstract data type (ADT) is conceptual model for a data structure characterized by its behavior rather than its implementation. It describes what operations are supported by the data and what result they should produce and how it would be implemented. Lists, stacks, queues and dictionaries are commonly used ADTS. A stack ADT, for example, can utilize operations like push, pop, and peek, irrespective of whether the stack is implemented using an array or linked list. But these classes combined to form ADTs allow you to arrive at a better design that generates modular and reusable code, allowing for more efficient software development.

1.2 Linear data structures using sequential organization, Operations

Key points: Linear data structures are essential elements of computer science and are crucial in the development of algorithms and software. Data elements are maintained in a sequential arrangement, with each element linked to its neighboring element. However, the sequential arrangement of the data renders these structures very natural, and hence easy to implement, manipulate and read. In this investigation we'll cover linear data structures that use the sequential layout, looking more closely at their operations, implementation techniques, performance characteristics, and where each would be applied in practice. An ordered structure indicates the orientation of data components in neighboring memory spots or with specific references to ensure logical proximity. Such an arrangement allows direct access to the elements and performs these operations are insertion, deletion, traversal, search, and modification. Query, Insert, and Delete operations However, these operations may have performance implications based on their respective implementations of linear data structures and memory management techniques. A linear data structure is a structure that has only one dimension. This characteristic makes them a good fit for representing data that has a built-in sequential order which can be natural such as lists, queues, and stacks. The sequential organization can be maintained either by array-based implementations or linked implementations (array based is less flexible while linked can have more complex time requirements).

Arrays: The Fundamental Sequential Structure

Arrays represent the most basic type of linear data structure, as they are organized in a sequential format. An array is data structure comprising a group of elements of same data type, stored in contiguous memory regions. This makes it possible to jump to any element in constant time

Notes

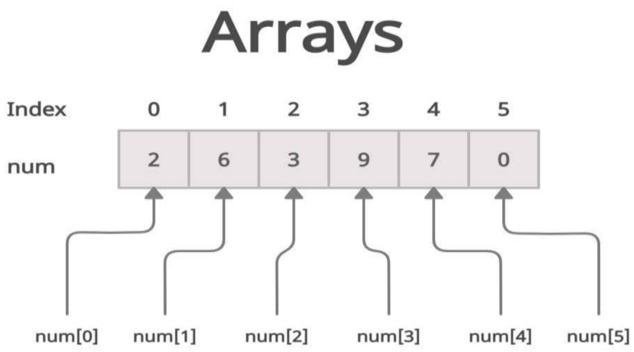


Figure 1.2: Arrays
[Source: https://usemynotes.com/]

(as long as you know the index), so arrays are efficient to use if you do a lot of random access.

Memory Allocation in Arrays

Arrays can be allocated memory in two ways:

1. **Static Allocation:** Memory allocation occurs at compile time, and the array size remains constant during the program's execution. This method is straightforward although deficient in adaptability.
   Static allocation of an array with 100 integer elements
2. **Dynamic Allocation:** Memory is allocated at runtime, permitting flexibility in array dimensions. This approach is more adaptable to varying data sizes but requires explicit memory management.
   int* numbers = (int*) malloc(100 * sizeof(int)); // Dynamic allocation in C
   int* numbers = new int[100]; // Dynamic allocation in C++
   Basic Operations on Arrays

Arrays support several fundamental operations:

1. Accessing Elements

Accessing an element at a specific index is a constant-time operation (O(1)) because arrays provide direct access to elements through indices.

int value = numbers[5]; // Accessing the element at index 5

2. Insertion Operations

Insertion in arrays depends on the position:

- Insertion at the End: If the array has space, inserting at the end is an O(1) operation.

Notes

```
if (currentSize < maxSize) {
   array[currentSize] = newElement;
   currentSize++;
}
```

- Insertion at the Beginning or Middle: Requires shifting elements to make space, resulting in an O(n) time complexity.

```
// Insertion at index 'position'
for (int i = currentSize; i > position; i--) {
   array[i] = array[i-1];
}
array[position] = newElement;
currentSize++;
```

3. Deletion Operations

Similar to insertion, deletion efficiency depends on the position:

- Deletion from the End: O(1) time complexity.

```
if (currentSize > 0) {
   currentSize--;
}
```

- Deletion from the Beginning or Middle: O(n) time complexity due to element shifting.

```
// Deletion at index 'position'
for (int i = position; i < currentSize - 1; i++) {
   array[i] = array[i+1];
}
currentSize--;
```

4. Searching Operations

Arrays support two main search approaches:

- Linear Search: Examines each element sequentially until finding the target or reaching the end, with O(n) time complexity.

```
int linearSearch(int array[], int size, int target) {
   for (int i = 0; i < size; i++) {
      if (array[i] == target) {
         return i; // Return index of found element
      }
   }
   return -1; // Element not found
}
```

- Binary Search: For sorted arrays, offers O(log n) time complexity by repeatedly dividing search space in half.

```
int binarySearch(int array[], int left, int right, int target) {
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (array[mid] == target)
            return mid;
        if (array[mid] < target)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return -1; // Element not found
}
```

5. Traversal Operations

Traversing an array involves visiting each element sequentially, typically using loops:

```
void traverse(int array[], int size) {
    for (int i = 0; i < size; i++) {
        // Process array[i]
        printf("%d ", array[i]);
    }
}
```

Advantages and Limitations of Arrays

Advantages:

- Constant-time random access (O(1))
- Memory efficiency due to lack of overhead for storing relationships
- Cache-friendly due to contiguous memory storage
- Simple implementation
- Limitations:
- Fixed size in static implementations
- Inefficient insertion and deletion operations at arbitrary positions
- Memory wastage when allocated size exceeds actual data size
- Homogeneous data type requirement

Notes

Multi-dimensional Arrays

Arrays can be extended to multiple dimensions to represent more complex data relationships:

```
int matrix[3][4]; // 2D array with 3 rows and 4 columns
// Accessing elements in a 2D array
int value = matrix[1][2]; // Accessing element at row 1, column 2

// Traversing a 2D array
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 4; j++) {
        // Process matrix[i][j]
    }
}
```

Multi-dimensional arrays are stored in memory using either row-major order (C/C++) or column-major order (Fortran), impacting how data is accessed and cached.

Dynamic Arrays: Extending the Basic Array

Unlike static arrays that have a fixed size, dynamic arrays resize themselves when they run out of space. They still have O(1) access time and however they can grow in size.

Implementation of Dynamic Arrays

A typical implementation involves:

1. Initializing with a default capacity
2. Keeping track of the current size
3. Resizing when necessary

```
class DynamicArray {
private:
    int* array;
    int size;
    int capacity;

    void resize() {
        capacity *= 2;
        int* newArray = new int[capacity];
        for (int i = 0; i < size; i++) {
            newArray[i] = array[i];
        }
        delete[] array;
```

```
        array = newArray;
    }

public:
    DynamicArray() {
        capacity = 10;
        size = 0;
        array = new int[capacity];
    }

    void add(int element) {
        if (size == capacity) {
            resize();
        }
        array[size++] = element;
    }

    // Other operations...
};
```

Operations on Dynamic Arrays

Dynamic arrays support the same operations as static arrays but with added resizing capability:

1. Amortized Analysis of Insert Operation

Insertion at the end has an amortized O(1) time complexity. Though individual resize operations are O(n), they are rare enough that the amortized cost of each operation is constantapplied to the underlying array.

```
void add(int element) {
    if (size == capacity) {
        resize(); // O(n) operation but happens rarely
    }
    array[size++] = element; // O(1) operation
}
```

2. Performance Considerations

- Growth Factor: Typically set to 2, meaning the array doubles in size when full
- Shrinking: Some implementations also decrease capacity when utilization falls below a certain threshold

- **Dynamic Arrays in Standard Libraries**
- Various programming languages provide dynamic array implementations:
- std::vector in C++
- ArrayList in Java
- List in C#
- list in Python (with additional functionality)

```
// Using std::vector in C++
#include <vector>
vector<int> numbers;
numbers.push_back(10); // Add element to the end
```

Stacks: LIFO Sequential Structures

A stack is a linear data structure that exhibits a Last-In-First-Out (LIFO) order: in a stack, the last added element is the first removed one. Like a stack of plates, you can only add and remove plates at the top (Last In First Out).

Operations on Stacks

Stacks support two primary operations:

1. Push Operation

Push adds an element to the top of the stack:

```
void push(Stack* stack, int value) {
    if (stack->top == stack->capacity - 1) {
        // Stack overflow
        return;
    }
    stack->array[++stack->top] = value;
}
```

2. Pop Operation

Pop removes and returns the element from the top of the stack:

```
int pop(Stack* stack) {
    if (stack->top == -1) {
        // Stack underflow
        return -1;
    }
    return stack->array[stack->top--];
}
```

3. Additional Stack Operations

- Peek/Top: Returns the top element without removing it

- isEmpty: Checks if the stack is empty
- isFull: Checks if the stack is full (for array implementations)
- Size: Returns the number of elements in the stack

```
int peek(Stack* stack) {
    if (stack->top == -1) {
        // Stack is empty
        return -1;
    }
    return stack->array[stack->top];
}

bool isEmpty(Stack* stack) {
    return stack->top == -1;
}

bool isFull(Stack* stack) {
    return stack->top == stack->capacity - 1;
}

int size(Stack* stack) {
    return stack->top + 1;
}
```

Stack Implementations

Stacks can be implemented using:

1. Array-based Implementation

```
typedef struct {
    int* array;
    int top;
    int capacity;
} Stack;

Stack* createStack(int capacity) {
    Stack* stack = (Stack*)malloc(sizeof(Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = (int*)malloc(stack->capacity * sizeof(int));
    return stack;
}
```

2. Linked List-based Implementation

```
typedef struct Node {
    int data;
    struct Node* next;
} Node;

typedef struct {
    Node* top;
    int size;
} Stack;

Stack* createStack() {
    Stack* stack = (Stack*)malloc(sizeof(Stack));
    stack->top = NULL;
    stack->size = 0;
    return stack;
}

void push(Stack* stack, int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = value;
    newNode->next = stack->top;
    stack->top = newNode;
    stack->size++;
}

int pop(Stack* stack) {
    if (stack->top == NULL) {
        // Stack underflow
        return -1;
    }

    Node* temp = stack->top;
    int value = temp->data;
    stack->top = stack->top->next;
    free(temp);
    stack->size--;
```

```
    return value;
}
```

Applications of Stacks

Stacks have numerous practical applications:

- Function call management (call stack)
- Expression evaluation and conversion (infix to postfix)
- Syntax parsing in compilers
- Undo mechanism in text editors
- Backtracking algorithms
- Browser back button implementation

Example: Checking for Balanced Parentheses

```
bool areParenthesesBalanced(char* expr) {
    Stack* stack = createStack(strlen(expr));
    for (int i = 0; expr[i]; i++) {
        if (expr[i] == '(' || expr[i] == '[' || expr[i] == '{') {
            push(stack, expr[i]);
        } else if (expr[i] == ')' || expr[i] == ']' || expr[i] == '}') {
            if (isEmpty(stack)) {
                return false;
            }
            char top = pop(stack);
            if ((expr[i] == ')' && top != '(') ||
                (expr[i] == ']' && top != '[') ||
                (expr[i] == '}' && top != '{')) {
                return false;
            }
        }
    }
    return isEmpty(stack);
}
```

Queues: FIFO Sequential Structures

A queue is a linear data structure with a First-In-First-Out (FIFO) order, like people in line. First In, First Out (FIFO) — The first element added is the first one to be removed.

Operations on Queues

Queues support two primary operations:

1. Enqueue Operation

Adds an element to the rear of the queue:

```
void enqueue(Queue* queue, int value) {
    if ((queue->rear + 1) % queue->capacity == queue->front) {
        // Queue is full
        return;
    }

    if (queue->front == -1) {
        queue->front = 0;
    }

    queue->rear = (queue->rear + 1) % queue->capacity;
    queue->array[queue->rear] = value;
}
```

2. Dequeue Operation

Removes and returns the element from the front of the queue:

```
int dequeue(Queue* queue) {
    if (queue->front == -1) {
        // Queue is empty
        return -1;
    }

    int value = queue->array[queue->front];
    if (queue->front == queue->rear) {
        // Last element being dequeued
        queue->front = queue->rear = -1;
    } else {
        queue->front = (queue->front + 1) % queue->capacity;
    }

    return value;
}
```

3. Additional Queue Operations

- Front: Returns the front element without removing it
- isEmpty: Checks if the queue is empty
- isFull: Checks if the queue is full
- Size: Returns the number of elements in the queue

```
int front(Queue* queue) {
    if (queue->front == -1) {
```

```
        // Queue is empty
        return -1;
    }
    return queue->array[queue->front];
}
bool isEmpty(Queue* queue) {
    return queue->front == -1;
}
bool isFull(Queue* queue) {
    return (queue->rear + 1) % queue->capacity == queue->front;
}
int size(Queue* queue) {
    if (queue->front == -1) {
        return 0;
    }
    return (queue->rear - queue->front + queue->capacity) % queue-
>capacity + 1;
}
```

Queue Implementations

Queues can be implemented using:

1. Array-based Implementation (Circular Queue)

A circular queue efficiently uses array space by wrapping around when reaching the end:

```
typedef struct {
    int* array;
    int front;
    int rear;
    int capacity;
} Queue;
Queue* createQueue(int capacity) {
    Queue* queue = (Queue*)malloc(sizeof(Queue));
    queue->capacity = capacity;
    queue->front = queue->rear = -1;
    queue->array = (int*)malloc(queue->capacity * sizeof(int));
    return queue;
}
```

2. Linked List-based Implementation

```
typedef struct Node {
```

```
    int data;
    struct Node* next;
} Node;
typedef struct {
    Node* front;
    Node* rear;
    int size;
} Queue;
Queue* createQueue() {
    Queue* queue = (Queue*)malloc(sizeof(Queue));
    queue->front = queue->rear = NULL;
    queue->size = 0;
    return queue;
}
void enqueue(Queue* queue, int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = value;
    newNode->next = NULL;
    if (queue->rear == NULL) {
        queue->front = queue->rear = newNode;
    } else {
        queue->rear->next = newNode;
        queue->rear = newNode;
    }
    queue->size++;
}
int dequeue(Queue* queue) {
    if (queue->front == NULL) {
        // Queue is empty
        return -1;
    }
    Node* temp = queue->front;
    int value = temp->data;
    queue->front = queue->front->next;
    if (queue->front == NULL) {
        queue->rear = NULL;
    }
    free(temp);
```

```
    queue->size--;
    return value;
}
```

Variations of Queues

Several specialized queue variations exist:

1. Double-ended Queue (Deque)

A deque allows insertion and deletion at both ends:

```
typedef struct {
    int* array;
    int front;
    int rear;
    int capacity;
} Deque;
void insertFront(Deque* deque, int value) {
    if (isFull(deque)) {
        return;
    }
    if (deque->front == -1) {
        deque->front = deque->rear = 0;
    } else {
        deque->front = (deque->front - 1 + deque->capacity) % deque-
>capacity;
    }
    deque->array[deque->front] = value;
}
void insertRear(Deque* deque, int value) {
    if (isFull(deque)) {
        return;
    }
    if (deque->front == -1) {
        deque->front = deque->rear = 0;
    } else {
        deque->rear = (deque->rear + 1) % deque->capacity;
    }
    deque->array[deque->rear] = value;
}

int deleteFront(Deque* deque) {
```

```
    if (isEmpty(deque)) {
        return -1;
    }
    int value = deque->array[deque->front];
    if (deque->front == deque->rear) {
        deque->front = deque->rear = -1;
    } else {
        deque->front = (deque->front + 1) % deque->capacity;
    }
    return value;
}
int deleteRear(Deque* deque) {
    if (isEmpty(deque)) {
        return -1;
    }
    int value = deque->array[deque->rear];
    if (deque->front == deque->rear) {
        deque->front = deque->rear = -1;
    } else {
        deque->rear = (deque->rear - 1 + deque->capacity) % deque->capacity;
    }
    return value;
}
```

2. Priority Queue

A priority queue serves elements based on their priority rather than insertion order.

3. Circular Queue

A circular queue optimizes array space usage by connecting the end to the beginning:

// Circular queue was covered in the basic queue implementation

Applications of Queues

Queues are used in various applications:

- Process scheduling in operating systems
- Breadth-first search in graphs
- Print job spooling
- Handling of interrupts in real-time systems
- Buffering in various applications (keyboard buffer, web servers)

- Message queues in distributed systems

Example: Level Order Traversal of a Binary Tree

```
void levelOrderTraversal(TreeNode* root) {
    if (root == NULL) {
        return;
    }
    Queue* queue = createQueue();
    enqueue(queue, root);
    while (!isEmpty(queue)) {
        TreeNode* current = dequeue(queue);
        printf("%d ", current->data);
        if (current->left) {
            enqueue(queue, current->left);
        }
        if (current->right) {
            enqueue(queue, current->right);
        }
    }
}
```

Linked Lists: Dynamic Sequential Structures

A linked list is a collection with a linear structure where each element is stored in a node that consists of a value and a reference to the next element. They do not need to allocate memory contiguously, unlike arrays, which allows them to grow dynamically and have efficient insertions/deletions in between.

Types of Linked Lists

Linked lists come in several variations:

1. Singly Linked List

Each node contains data and a pointer to the next node:

```
typedef struct Node {
    int data;
    struct Node* next;
} Node;
```

2. Doubly Linked List

Each node contains data and pointers to both the next and previous nodes:

```
typedef struct Node {
    int data;
```

Notes

```
    struct Node* next;
    struct Node* prev;
} Node;
```

3. Circular Linked List

The last node points back to the first node, creating a circle:

```
// For a circular singly linked list
// The last node's next points to the head
```

Operations on Linked Lists

Linked lists support various operations:

1. Insertion Operations

- Insertion at the Beginning:

```
void insertAtBeginning(Node** head, int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = value;
    newNode->next = *head;
    *head = newNode;
}
```

- Insertion at the End:

```
void insertAtEnd(Node** head, int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = value;
    newNode->next = NULL;

    if (*head == NULL) {
        *head = newNode;
        return;
    }
    Node* current = *head;
    while (current->next != NULL) {
        current = current->next;
    }
    current->next = newNode;
}
```

- Insertion at a Specific Position:

```
void insertAtPosition(Node** head, int value, int position) {
    if (position < 0) {
        return;
    }
```

```
    if (position == 0 || *head == NULL) {
        insertAtBeginning(head, value);
        return;
    }
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = value;

    Node* current = *head;
    for (int i = 0; i < position - 1 && current->next != NULL; i++) {
        current = current->next;
    }
    newNode->next = current->next;
    current->next = newNode;
}
```

2. Deletion Operations

- Deletion from the Beginning:

```
void deleteFromBeginning(Node** head) {
    if (*head == NULL) {
        return;
    }
    Node* temp = *head;
    *head = (*head)->next;
    free(temp);
}
```

- Deletion from the End:

```
void deleteFromEnd(Node** head) {
    if (*head == NULL) {
        return;
    }
    if ((*head)->next == NULL) {
        free(*head);
        *head = NULL;
        return;
    }
    Node* current = *head;
    while (current->next->next != NULL) {
        current = current->next;
    }
```

```
    free(current->next);
    current->next = NULL;
}
```

- Deletion at a Specific Position:

```
void deleteAtPosition(Node** head, int position) {
    if (*head == NULL || position < 0) {
        return;
    }
    if (position == 0) {
        deleteFromBeginning(head);
        return;
    }
    Node* current = *head;
    for (int i = 0; i < position - 1 && current->next != NULL; i++) {
        current = current->next;
    }
    if (current->next == NULL) {
        return;
    }
    Node* temp = current->next;
    current->next = current->next->next;
    free(temp);
}
```

3. Search Operation

```
Node* search(Node* head, int value) {
    Node* current = head;
    while (current != NULL) {
        if (current->data == value) {
            return current;
        }
        current = current->next;
    }
    return NULL;
}
```

4. Traversal Operation

```
void traverse(Node* head) {
    Node* current = head;
    while (current != NULL) {
```

```
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}
```

Doubly Linked List Operations

Doubly linked lists offer bidirectional traversal but require more complex operations:

1. Insertion in a Doubly Linked List

```
void insertAtBeginning(Node** head, int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = value;
    newNode->next = *head;
    newNode->prev = NULL;
    if (*head != NULL) {
        (*head)->prev = newNode;
    }
    *head = newNode;
}
void insertAtEnd(Node** head, int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = value;
    newNode->next = NULL;
    if (*head == NULL) {
        newNode->prev = NULL;
        *head = newNode;
        return;
    }
    Node* current = *head;
    while (current->next != NULL) {
        current = current->next;
    }
    current->next = newNode;
    newNode->prev = current;
}
```

2. Deletion in a Doubly Linked List

```
void deleteNode(Node** head, Node* toDelete) {
    if (*head == NULL || toDelete == NULL) {
```

```
        return;
    }
    if (*head == toDelete) {
        *head = toDelete->next;
    }
    if (toDelete->next != NULL) {
        toDelete->next->prev = toDelete->prev;
    }
    if (toDelete->prev != NULL) {
        toDelete->prev->next = toDelete->next;
    }
    free(toDelete);
}
```

Circular Linked List Operations

Circular linked lists require special handling of the last node:

```
void insertIntoEmpty(Node** head, int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = value;
    *head = newNode;
    newNode->next = *head;
}
void insertAtBeginning(Node** head, int value) {
    if (*head == NULL) {
        insertIntoEmpty(head, value);
        return;
    }
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = value;
    Node* current = *head;
    while (current->next != *head) {
        current = current->next;
    }
    newNode->next = *head;
    current->next = newNode;
    *head = newNode;
}
```

Advantages and Limitations of Linked Lists

Advantages:

- Dynamic size
- Efficient insertions and deletions
- No memory wastage
- Flexible memory management

Limitations:

- Random access is not supported (O(n) time complexity)
- Extra memory required for pointers
- Not cache-friendly due to non-contiguous memory
- Reverse traversal is difficult in singly linked lists
- Applications of Linked Lists
- Linked lists are used in various applications:
- Implementation of stacks and queues
- Dynamic memory allocation
- Representation of sparse matrices
- Polynomial manipulation
- Hash tables (chaining)
- Adjacency lists for graphs

Example: Reversing a Linked List

```
Node* reverseList(Node* head) {
   Node* prev = NULL;
   Node* current = head;
   Node* next = NULL;
   while (current != NULL) {
      next = current->next;
      current->next = prev;
      prev = current;
      current = next;
   }
   return prev;
}
```

Specialized Linear Data Structures

Sparse Arrays

Sparse arrays efficiently store arrays with many default values by only storing non-default entries.

```
typedef struct {
   int row;
   int col;
   int value;
```

```
} Element;
typedef struct {
    int rows;
    int cols;
    int numElements;
    Element* elements;
} SparseArray;
```

Skip Lists

Skip lists provide probabilistic alternatives to balanced trees with O(log n) average search time.

```
typedef struct SkipListNode {
    int value;
    int level;
    struct SkipListNode** forward;
} SkipListNode;
typedef struct {
    int level;
    int size;
    SkipListNode* header;
} SkipList;
```

Memory-Efficient Linked Lists (XOR Linked Lists)

XOR linked lists combine both addressing (previous and next) with bitwise XOR operation to compress Memory.

```
typedef struct Node {
    int data;
    struct Node* npx; // XOR of next and previous node addresses
} Node;
// Helper functions to get next and previous nodes
Node* XOR(Node* a, Node* b) {
    return (Node*)((uintptr_t)a ^ (uintptr_t)b);
}
```

Performance Comparison and Selection Criteria

Time Complexity Comparison

Operation	Array	Dynamic Array	Linked List	Stack	Queue
Access	O(1)	O(1)	O(n)	O(1)*	O(1)*
Insert (Start)	O(n)	O(n)	O(1)	N/A	N/A

Insert (End)	O(1)**	Amortized O(1)	O(n)/O(1)***	O(1)	O(1)
Insert (Middle)	O(n)	O(n)	O(n)	N/A	N/A
Delete (Start)	O(n)	O(n)	O(1)	N/A	O(1)
Delete (End)	O(1)**	O(1)	O(n)/O(1)***	O(1)	N/A
Delete (Middle)	O(n)	O(n)	O(n)	N/A	N/A
Search	O(n)/O(log n)****	O(n)/O(log n)****	O(n)	N/A	N/A

* For top/front elements only ** If size is tracked *** O(1) if tail pointer is maintained **** O(log n) with binary search if sorted

Space Complexity Comparison

Data Structure	Space Complexity
Array (Static)	O(n)

Selection Criteria

Choosing the appropriate data structure depends on:

Data Structure	Space Complexity
Array (Static)	O(n)
Dynamic Array	O(n)
Linked List	O(n)
Stack	O(n)
Queue	O(n)

1. Access Pattern: Random access vs. sequential access
2. Modification Frequency: Frequent insertions/deletions vs. static data
3. Size Constraints: Fixed size vs. dynamic growth
4. Memory Constraints: Overhead acceptability
5. Operation Types: LIFO, FIFO, or random operations

Unit 2: Linear Array

1.3 Linear Array in data structure and its classification, Properties

Linear array is one of the basic data structure of computer science. It is a group of information saved in the successive memory location and can be accessed conveniently by indexing.

Classification of Linear Arrays

Linear arrays can be classified in several ways:

Based on dimension:

- One-dimensional arrays (vectors)
- Multi-dimensional arrays (matrices, tensors)

Based on size flexibility:

- Static arrays (fixed size, determined at compile time)
- Dynamic arrays (variable size, can grow or shrink at runtime)

Based on the type of elements:

- Homogeneous arrays (all elements have the same data type)
- Heterogeneous arrays (elements can have different data types, like structs or objects)

Properties of Linear Arrays

Linear arrays have several important properties:

1. Random Access

- Elements can be accessed directly using their index in O(1) time
- Formula: address = base_address + (index * size_of_each_element)

2. Memory Allocation

- Elements are stored in contiguous memory locations
- Static arrays have a fixed size allocation
- Dynamic arrays may reallocate memory when resizing

3. Time Complexity

- Access: O(1)
- Search: O(n) for unsorted arrays, O(log n) for sorted arrays using binary search
- Insertion/Deletion:
- At the end: O(1) amortized for dynamic arrays
- At arbitrary positions: O(n) due to shifting elements

4. Space Complexity

- O(n) where n is the number of elements
- Requires extra space for potential growth in dynamic arrays

5. Cache Friendly

- Due to contiguous memory allocation, arrays benefit from spatial locality
- This makes them efficient for CPU cache utilization

6. Limitations

- Static arrays cannot change size once allocated
- Dynamic arrays have overhead for resizing operations
- Insertion/deletion in the middle is inefficient due to shifting

1.4 representations of an array, Operation and Memory location

Wherein arrays are one of the most basic and common data structures in computer science. From their elegant simplicity stems their immense utility across almost all domains of programming. Like at heart, an array is a collection of elements, all of which are specified by an index or a key. These elements are stored sequentially in memory, which enables fast access and manipulation. Arrays are not only useful for storing elements, but they are also the basic building blocks for many algorithms and higher-level data structures. Arrays serve as the backbone of many operations, from sorting and searching algorithms to image processing and numerical calculations. Join us as we take a deep dive into the workings of arrays, from how they're structured in memory to the various operations are supported and what makes them so efficient. We'll explore everything from abstract fundamentals to programming distinctions and low-level details of different implementations of arrays.

Basic Array Representation

An array can be thought of as an enumerated list of cells, each of which can contain a single data type. In this sequence, every cell is assigned an integer index, each one unique, with a typical base value (0 or 1 depending on language) that acts as the first index.

For a one-dimensional array A with n elements, we can represent it as: A = [A[0], A[1], A[2], ..., A[n-1]] (for 0-indexed arrays) A = [A[1], A[2], A[3], ..., A[n]] (for 1-indexed arrays)

This indexed access pattern defines arrays in contrast to other collection data types, including linked lists or sets. This sort of direct mapping means that you can access any element in constant-time.

Mathematical Representation

Mathematically, an array can be viewed as a mapping function from indices to values:

Notes

A: I → V

Where:

- I is the set of valid indices (typically a contiguous range of integers)
- V is the set of possible values the array can store

For a one-dimensional array of size n, the index set I = {0, 1, 2, ..., n-1} for 0-indexed arrays, or I = {1, 2, 3, ..., n} for 1-indexed arrays.

Physical Representation in Memory

Array performance characteristics are dictated by its physical representation in memory. In arrays, the elements are stored as a contiguous block of memory, where each element has a fixed amount of space based on its data type.

Memory Addressing and Location Calculation

Linear Addressing for One-Dimensional Arrays

The memory address of an element in a one-dimensional array can be calculated using a simple formula:

Address of A[i] = Base_Address + (i - Lower_Bound) × Size_of_Each_Element

Where:

- Base_Address is the memory address of the first element of the array
- Lower_Bound is the starting index of the array (typically 0 or 1)
- Size_of_Each_Element is the number of bytes each element occupies

For example, in a 0-indexed array of integers (assuming 4 bytes per integer), the address of the element at index 5 would be: Address of A[5] = Base_Address + (5 - 0) × 4 = Base_Address + 20

This direct calculation is what enables O(1) time complexity for array element access.

Row-Major vs. Column-Major Ordering

For multi-dimensional arrays, two primary memory layout strategies exist:

- Row-Major Ordered: Same Row Elements are Stored Together Used in C, C++, Python and other languages.
- Elements in the same column are stored contiguously. This is prevalent in Fortran, R, MATLAB, etc.

Depending on how ordering is done, it can affect address calculation for reading elements and its impact on performance on some operations especially with respect to cache efficiency.

Memory Location Calculation for Multi-Dimensional Arrays

Row-Major Ordering

For a two-dimensional array A[m][n] in row-major ordering, the address of element A[i][j] is calculated as:

Address of A[i][j] = Base_Address + ((i - Row_Lower_Bound) × n + (j - Column_Lower_Bound)) × Size_of_Each_Element

For a three-dimensional array A[m][n][p], the formula extends to:

Address of A[i][j][k] = Base_Address + (((i - Row_Lower_Bound) × n + (j - Column_Lower_Bound)) × p + (k - Depth_Lower_Bound)) × Size_of_Each_Element

Column-Major Ordering

For a two-dimensional array A[m][n] in column-major ordering:

Address of A[i][j] = Base_Address + ((j - Column_Lower_Bound) × m + (i - Row_Lower_Bound)) × Size_of_Each_Element

The patterns don't stop in single dimension, higher dimensions are basically adding the coordinates for the different dimensions into the address calculation.

Memory Allocation Mechanisms

Static Allocation

Static arrays are arrays with a size determined at compile time. Generally, the memory is allocated in the stack segment of the program memory space. It once set the size which can never be changed during program execution.

In languages like C, static allocation looks like:

```
int array[100]; // Allocates 400 bytes (assuming 4 bytes per int)
```

Therefore, the compiler knows exactly how much memory to allocate, and the memory is automatically deal located when the variable gets out of scope.

Dynamic Allocation

Dynamic arrays are created during runtime and stored in the heap memory segment. This means you can determine the size more flexibly based on the conditions at runtime.

In C, dynamic allocation can be done using:

```
int* array = (int*)malloc(n * sizeof(int)); // Allocates n*4 bytes
```

In C++, the equivalent would be:

Notes

```
int* array = new int[n]; // Allocates n*4 bytes
```

Dynamic allocation requires explicit deallocation to prevent memory leaks:

```
free(array); // C
delete[] array; // C++
```

Automatic Resizing and Growth Strategies

Many modern programming languages include a dynamically resizable array implementation, like C++'s std::vector, Java's ArrayList, or built-in lists in Python. Such implementations often employ the following growth strategies:

1. Amortized Doubling Once the capacity is reached, we allocate a new array with double capacity, copy over all elements, then deal locate the old array.
2. Doubling——basically doubling the unit scale when buffer reaches certain thresholds or Growth Factor similar to doubling but different multiplication factor e.g. 1.5x in some implementations
3. Add Constant Space: Add a fixed amount at a time.

Dynamic Array Performance Characteristics Dynamic arrays can achieve performance characteristics similar to classical arrays, except for the cost of an occasional copying operation. The benefit of Jochen Hoenicke's trick prevents exponentially growing memory consumption. The perfect hash entailed exponential growth, which K&R prevented, but Jochen Hoenicke's trick made dynamic arrays possible for performance in real code..

Basic Array Operations

Access Operation

Accessing an array element is performed by using its index:

```
value = array[index]
```

Time Complexity: O(1) - Constant time, as it involves a direct memory address calculation.

Traversal Operation

Traversal involves visiting each element of the array exactly once:

```
for i = 0 to length(array) - 1
   process array[i]
```

Time Complexity: O(n) - Linear time, where n is the number of elements.

Search Operation

Linear Search

Linear search scans elements one by one:

```
function linearSearch(array, target)
   for i = 0 to length(array) - 1
      if array[i] equals target
         return i
   return -1 // Not found
```

Time Complexity: O(n) - Linear time, where n is the number of elements.

Binary Search (for sorted arrays)

Binary search divides the search interval in half repeatedly:

```
function binarySearch(array, target)
   left = 0
   right = length(array) - 1
   while left <= right
      mid = (left + right) / 2
      if array[mid] equals target
         return mid
      else if array[mid] < target
         left = mid + 1
      else
         right = mid - 1
   return -1 // Not found
```

Time complexity: O(log n) - Logarithmic time and its much better than linear search for large array.

Insertion Operation

Insertion at the End

For arrays with available space at the end:

```
function insertAtEnd(array, value)
   array[size] = value
   size = size + 1
```

Time Complexity: O(1) - Constant time, assuming space is available. For dynamic arrays that might need resizing: O(1) amortized time.

Insertion at a Specific Position

To insert an element at position pos:

```
function insertAt(array, pos, value)
   for i = size downto pos + 1
      array[i] = array[i-1]
```

```
    array[pos] = value
    size = size + 1
```

Time Complexity: O(n) - Linear time, since elements need to be shifted.

Deletion Operation

Deletion from the End

```
function deleteFromEnd(array)
    size = size - 1
```

Time Complexity: O(1) - Constant time.

Deletion from a Specific Position

To delete an element at position pos:

```
function deleteAt(array, pos)
    for i = pos to size - 2
        array[i] = array[i+1]
    size = size - 1
```

Time Complexity: O(n) - Linear time, since elements need to be shifted.

Update Operation

Updating an element at a specific index:

```
function update(array, index, newValue)
    array[index] = newValue
```

Time Complexity: O(1) - Constant time.

Advanced Array Operations

Sorting Operations

Arrays are commonly used with various sorting algorithms, each with different performance characteristics:

Bubble Sort

```
function bubbleSort(array)
    for i = 0 to length(array) - 1
        for j = 0 to length(array) - i - 2
            if array[j] > array[j+1]
                swap(array[j], array[j+1])
```

Time Complexity: $O(n^2)$ - Quadratic time.

Selection Sort

```
function selectionSort(array)
    for i = 0 to length(array) - 2
        minIndex = i
        for j = i + 1 to length(array) - 1
            if array[j] < array[minIndex]
                minIndex = j
```

```
    swap(array[i], array[minIndex])
```

Time Complexity: $O(n^2)$ - Quadratic time.

Insertion Sort

```
function insertionSort(array)
  for i = 1 to length(array) - 1
    key = array[i]
    j = i - 1
    while j >= 0 and array[j] > key
      array[j+1] = array[j]
      j = j - 1
    array[j+1] = key
```

Time Complexity: $O(n^2)$ - Quadratic time, but performs well on almost-sorted arrays.

Merge Sort

```
function mergeSort(array, left, right)
  if left < right
    mid = (left + right) / 2
    mergeSort(array, left, mid)
    mergeSort(array, mid + 1, right)
    merge(array, left, mid, right)
```

Time Complexity: $O(n \log n)$ - Linearithmic time.

Quick Sort

```
function quickSort(array, low, high)
  if low < high
    pivotIndex = partition(array, low, high)
    quickSort(array, low, pivotIndex - 1)
    quickSort(array, pivotIndex + 1, high)
```

Time Complexity: $O(n \log n)$ average case, $O(n^2)$ worst case.

Heap Sort

```
function heapSort(array)
  buildMaxHeap(array)
  for i = length(array) - 1 downto 1
    swap(array[0], array[i])
    heapify(array, 0, i)
```

Time Complexity: $O(n \log n)$ - Linearithmic time.

Mathematical Operations

Array Sum

```
function arraySum(array)
```

```
    sum = 0
    for i = 0 to length(array) - 1
        sum = sum + array[i]
    return sum
```

Time Complexity: O(n) - Linear time.

Array Product

```
function arrayProduct(array)
    product = 1
    for i = 0 to length(array) - 1
        product = product * array[i]
    return product
```

Time Complexity: O(n) - Linear time.

Array Mean (Average)

```
function arrayMean(array)
    sum = arraySum(array)
    return sum / length(array)
```

Time Complexity: O(n) - Linear time.

Finding Maximum and Minimum

```
function findMax(array)
    max = array[0]
    for i = 1 to length(array) - 1
        if array[i] > max
            max = array[i]
    return max
function findMin(array)
    min = array[0]
    for i = 1 to length(array) - 1
        if array[i] < min
            min = array[i]
    return min
```

Time Complexity: O(n) - Linear time.

Array Transformation Operations

Mapping

Applying a function to each element:

```
function map(array, func)
    result = new array of same size
    for i = 0 to length(array) - 1
        result[i] = func(array[i])
```

```
    return result
```

Time Complexity: O(n) - Linear time.

Filtering

Creating a new array with elements that pass a test:

```
function filter(array, predicate)
    result = new empty array
    for i = 0 to length(array) - 1
        if predicate(array[i]) is true
            append array[i] to result
    return result
```

Time Complexity: O(n) - Linear time.

Reducing

Combining array elements into a single value:

```
function reduce(array, callback, initialValue)
    accumulator = initialValue
    for i = 0 to length(array) - 1
        accumulator = callback(accumulator, array[i])
 return accumulator
```

Time Complexity: O(n) - Linear time.

Multi-Dimensional Arrays

Two-Dimensional Array Representation

In fact, a two-dimensional array would look just like a table with rows and columns. An m×n array has m rows and n columns.

Mathematically, a 2D array A can be represented as:

A = [[A[0,0], A[0,1], ..., A[0,n-1]], [A[1,0], A[1,1], ..., A[1,n-1]], ... [A[m-1,0], A[m-1,1], ..., A[m-1,n-1]]]

Memory Representation of Multi-Dimensional Arrays

Contiguous Allocation

In such languages as C and C++, multidimensional arrays are laid out in contiguous segments of memory in row-major order or column-major order (depending on the language, as detailed in a previous section).

For example, a 3×4 array in row-major ordering would have elements stored in the following sequence: A[0,0], A[0,1], A[0,2], A[0,3], A[1,0], A[1,1], A[1,2], A[1,3], A[2,0], A[2,1], A[2,2], A[2,3]

Array of Arrays

For example in some languages and implementations, multi-dimensional arrays are implemented as arrays of arrays. Pretty common in languages such as JavaScript and some implementations in Java:

javascript

```
let matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];
```

Here, each element of the outer array is an array, that may or may not be contiguous in memory.

Operations on Multi-Dimensional Arrays

Accessing Elements

```
value = array[row][column]
```

Time Complexity: O(1) - Constant time.

Row and Column Traversal

Row traversal:

```
for i = 0 to rows - 1
  for j = 0 to columns - 1
    process array[i][j]
```

Column traversal:

```
for j = 0 to columns - 1
  for i = 0 to rows - 1
    process array[i][j]
```

Time Complexity: O(m×n) - Where m is the number of rows and n is the number of columns.

Matrix Addition

```
function matrixAdd(A, B)
  if A.rows != B.rows or A.columns != B.columns
    return error
  C = new matrix of size A.rows × A.columns
  for i = 0 to A.rows - 1
    for j = 0 to A.columns - 1
      C[i][j] = A[i][j] + B[i][j]
  return C
```

Time Complexity: O(m×n) - Where m is the number of rows and n is the number of columns.

Matrix Multiplication

```
function matrixMultiply(A, B)
```

```
if A.columns != B.rows
    return error
C = new matrix of size A.rows × B.columns
for i = 0 to A.rows - 1
    for j = 0 to B.columns - 1
        C[i][j] = 0
        for k = 0 to A.columns - 1
            C[i][j] += A[i][k] * B[k][j]
return C
```

Time Complexity: O(m×n×p) - Where A is an m×n matrix and B is an n×p matrix.

Matrix Transpose

```
function matrixTranspose(A)
    B = new matrix of size A.columns × A.rows
    for i = 0 to A.rows - 1
        for j = 0 to A.columns - 1
            B[j][i] = A[i][j]
return B
```

Time Complexity: O(m×n) - Where m is the number of rows and n is the number of columns.

Jagged Arrays

Definition and Representation

Jagged array: An array of arrays in which each array can have a different length. Unlike in the case of multi-dimensional arrays where each dimension has a fixed size.

For example, in C#:

```
int[][] jaggedArray = new int[3][];
jaggedArray[0] = new int[4];
jaggedArray[1] = new int[2];
jaggedArray[2] = new int[5];
```

This would result in a jagged array with 3 rows, the first row contains 4 elements, the second row contains 2 elements, the third row contains 5 elements.

Memory Representation

In a typical implementation of jagged arrays, the first array is an array of pointers to separate array. This is unlike the multi dimensional arrays that have a single block of memory allocation.

The memory structure would look like:

jaggedArray -> [ptr1, ptr2, ptr3]
ptr1 -> [element1, element2, element3, element4]
ptr2 -> [element5, element6]
ptr3 -> [element7, element8, element9, element10, element11]

Operations on Jagged Arrays

Jagged arrays are similar to regular arrays regarding operations performed on them — however, you must keep in mind the lengths of the arrays:

```
// Accessing elements
value = jaggedArray[row][column]
// Traversal
for i = 0 to length(jaggedArray) - 1
   for j = 0 to length(jaggedArray[i]) - 1
      process jaggedArray[i][j]
```

Time complexity for access is still O(1), and traversal is O(total number of elements).

Array Implementation in Different Programming Languages

C/C++ Arrays

"An array is a fixed-size sequence of elements of the same type, stored in contiguous memory" in C and C++ They are zero-indexed and have no bounds checking.

```
c
int array[5] = {1, 2, 3, 4, 5}; // Static array
int* dynamicArray = (int*)malloc(5 * sizeof(int)); // Dynamic array in C
int* dynamicArray = new int[5]; // Dynamic array in C++
```

C++ also provides the std::array and std::vector container classes:

```
std::array<int, 5> arr = {1, 2, 3, 4, 5}; // Fixed-size array
std::vector<int> vec = {1, 2, 3, 4, 5}; // Dynamic array
```

Java Arrays

Arrays are objects in Java that store one type of element. They are zero-indexed and have automatic bounds checking.

```
int[] array = new int[5]; // Declaration and allocation
int[] array = {1, 2, 3, 4, 5}; // Initialization with values
```

Java also provides the ArrayList class for dynamic arrays:

```
ArrayList<Integer> list = new ArrayList<>();
list.add(1);
list.add(2);
```

Python Lists

Python's lists are dynamic arrays that can contain elements of different types:

```
my_list = [1, "string", 3.14, True] # Mixed types
my_list.append(5) # Dynamic resizing
```

JavaScript Arrays

JavaScript arrays are also dynamic and heterogeneous:

```
let array = [1, "string", 3.14, true];
array.push(5); // Dynamic resizing
```

C# Arrays

C# arrays are similar to those in Java, being reference types with automatic bounds checking:

```
int[] array = new int[5]; // Declaration and allocation
int[] array = {1, 2, 3, 4, 5}; // Initialization with values
```

C# also provides the List<T> class for dynamic arrays:

```
List<int> list = new List<int>();
list.Add(1);
list.Add(2);
```

Memory Management Considerations

Memory Alignment

Memory alignment is the way data is arranged in memory (as per the data type). Modern architectures often benefit from, or require, data to be aligned at specific boundaries.

For instance, a 4-byte integer may need to be stored at an address that is a multiple of 4 bytes. The alignment requirement has implications in the way arrays are arranged into memory and will sometimes cause some padding in structures that hold arrays.

Cache Considerations

Arrays take advantage of spatial locality, which means when elements are stored close together in memory, they will tend to be accessed temporally close as well. This property gives arrays very high cache friendliness:

1. **Cache Lines:** When accessing some element, you also fetch the neighboring elements, all of which load up into cache resulting in faster accesses.
2. **Cache Misses:** Linear scans of an array tend to have fewer cache misses vs. random access patterns.

Notes

3. **Row-major vs. Column-major:** Row-major and column-major storage order can have a big effect on cache performance, based on the access order

Memory Fragmentation

Dynamic arrays that expand and contract can lead to memory fragmentation especially if they need to change their size often:

1. **External Fragmentation:** Happens when free memory is technically available but can not be allocated due to fragmentation and inability to get contigous large arrays of memory.
2. **Internal Fragmentation:** This occurs when more memory is allocated than is requested to satisfy alignment requirements or growth strategies

Memory Leaks in Dynamic Arrays

Dynamic arrays require careful management to prevent memory leaks:

1. **Memory Management:** Array created dynamically needs to be properly deallocated when not in use.
2. **Python:** The winner must learn both the basics of Python syntax (where every machine learning program, model, etc.
3. **Garbage Collection:** Languages like Java, Python, and JavaScript utilize garbage collection to automatically free up memory occupied by arrays that are no longer referenced

Performance Analysis of Array Operations

Time Complexity Analysis

Operation	Average Case	Worst Case
Access	O(1)	O(1)
Search (Unsorted)	O(n)	O(n)
Search (Sorted)	O(log n)	O(log n)
Insertion (End)	O(1)*	O(n)*
Insertion (Middle)	O(n)	O(n)
Deletion (End)	O(1)	O(1)
Deletion (Middle)	O(n)	O(n)
Traversal	O(n)	O(n)
Sort	O(n log n)	$O(n^2)$

*Amortized time complexity for dynamic arrays

Space Complexity Analysis

Arrays typically have a space complexity of O(n) — where n is the number of elements. However, dynamic arrays might allocate some additional space for future growth.

Space Complexity: Depending on the implementation of the data structure, it can use up to O(2n) in the worst case in case of unused space at times, if we use a doubling strategy for dynamic arrays, which means doubling the size whenever required..

Performance Comparison with Other Data Structures

Arrays vs. Linked Lists

Feature	Arrays	Linked Lists
Random Access	O(1)	O(n)
Insertion/Deletion at Beginning	O(n)	O(1)
Insertion/Deletion at End	O(1)*	O(1)**
Insertion/Deletion in Middle	O(n)	O(n)***
Memory Usage	Contiguous block	Non-contiguous nodes
Cache Performance	Excellent	Poor

*Amortized for dynamic arrays **Assuming tail pointer ***O(1) after finding the position, but finding takes O(n)

Arrays vs. Hash Tables

Feature	Arrays	Hash Tables
Access by Index	O(1)	N/A
Access by Key	O(n)	O(1) average
Insertion	O(n)	O(1) average
Deletion	O(n)	O(1) average
Ordered Data	Yes	No
Memory Usage	Low	Moderate to high

Arrays vs. Trees

Feature	Arrays	Binary Search Trees
Access	O(1)	O(log n)
Search	O(n) or O(log n)	O(log n)
Insertion	O(n)	O(log n)

Deletion	O(n)	O(log n)
Ordered Operations	No	Yes
Memory Usage	Low	Moderate

Specialized Array Types

Sparse Arrays

Sparse arrays are arrays in which the majority of the entries have the same value (typically zero). Sparse arrays only hold the non-zero elements along with their indices instead of storing all the elements.

Representation Methods

1. Dictionary/Map Representation: Store only non-zero values with their indices as keys.

sparse_array = {1: 5, 10: 3, 100: 8} # Elements at indices 1, 10, and 100

2. Coordinate List (COO): Store pairs of (index, value) for non-zero elements.

[(1, 5), (10, 3), (100, 8)]

3. Compressed Sparse Row (CSR): Used primarily for sparse matrices, storing row pointers, column indices, and values.

Operations on Sparse Arrays

Operations on sparse arrays are modified to work efficiently with the sparse representation:

```
// Access
function access(sparseArray, index)
  if index exists in sparseArray
    return sparseArray[index]
  else
    return defaultValue
```

One big reason why sparse arrays are so powerful is that they help save storage in a data structure designed for sparse matrices in scientific computing, graph algorithms, and large-scale data processing where data is naturally sparse.

Circular Arrays

This means we can treat each element of an array like we are in space where the end of the array becomes part of the start of the array (called circular arrays (or ring buffers)).

Implementation

Circular arrays are typically implemented using modular arithmetic to wrap around the array indices:

```
function get(circularArray, index)
   return array[index % length(array)]
```

For a fixed-size circular array used as a queue:

```
front = 0
rear = 0
function enqueue(value)
   if isFull()
      return error
   array[rear] = value
   rear = (rear + 1) % capacity
function dequeue()
   if isEmpty()
      return error
   value = array[front]
   front = (front + 1) % capacity
   return value
```

Applications of Circular Arrays

1. Circular Buffers: Used in producer-consumer scenarios, streaming data processing, and I/O operations.
2. Real-time Systems: Used in scheduling algorithms and event handling.
3. Memory-efficient Queues: Implementing queues without the need to shift elements.

Dynamic Arrays with Custom Growth Strategies

Different applications may benefit from different growth strategies for dynamic arrays:

1. Geometric Growth (e.g., doubling): Provides good amortized performance but may waste memory.
2. Arithmetic Growth (e.g., adding fixed chunks): More memory-efficient but with higher frequency of resizing operations.
3. Custom Predictive Growth: Adjusting growth based on usage patterns and application-specific knowledge.

Advanced Memory Management Techniques

Memory Pools for Array Allocation

Notes

Memory pools preallocate a big chunk of memory upfront, then distributes it for array allocations. This helps with fragmentation and allocation overhead:

```
function initializeMemoryPool(poolSize)
   pool = allocate(poolSize)
   freeList = initialize linked list of all blocks
function allocateFromPool(size)
   block = find suitable block in freeList
   if block is found
      remove block from freeList
      return block
   else
      return null // Out of memory
```

Custom Allocators

Custom allocators provide application-specific memory management for arrays:

1. Stack Allocators: Fast allocation/deallocation in LIFO order.
2. Buddy Allocators: Efficient handling of varying-sized allocations with minimal fragmentation.
3. Slab Allocators: Optimized for fixed-size allocations, common in operating system kernels.

Memory-Mapped Arrays

Memory-mapped arrays leverage the virtual memory capabilities of the operating system and map the content of an array to a disk file:

```
array = mmap(fileDescriptor, length, protectionFlags, flags, offset)
```

Benefits include:

1. Arrays larger than physical memory
2. Persistence between program executions
3. Efficient sharing between processes

Optimizing Array Operations

SIMD Vectorization

SIMD (Single Instruction, Multiple Data) instructions let you (in one go) perform the same operation over multiple array elements:

```
// Scalar addition
for (int i = 0; i < n; i++)
   c[i] = a[i] + b[i];
// SIMD addition (abstract pseudocode)
for (int i = 0; i < n; i += 4)
```

```
    c[i:i+3] = a[i:i+3] + b[i:i+3];  // Process 4 elements at once
```

Most modern compilers will automatically vectorize array operations, though for peak performance you may still need to manually optimize.

Loop Unrolling

Loop unrolling reduces loop overhead by processing multiple elements in each iteration:

```
// Original loop
for (int i = 0; i < n; i++)
    array[i] = process(array[i]);
// Unrolled loop
for (int i = 0; i < n; i += 4) {
    array[i] = process(array[i]);
    array[i+1] = process(array[i+1]);
    array[i+2] = process(array[i+2]);
    array[i+3] = process(array[i+3]);
}
```

Cache-Aware Algorithms

Optimizing array algorithms for cache performance:

1. Blocking/Tiling: Processing data in chunks that fit in cache.

```
// Matrix multiplication with blocking
for (int i = 0; i < n; i += blockSize)
    for (int j = 0; j < n; j += blockSize)
        for (int k = 0; k < n; k += blockSize)
            // Process block
```

2. Cache-Oblivious Algorithms: Algorithms that inherently perform well on any cache hierarchy without explicit tuning.
3. Array of Structures vs. Structure of Arrays: Choosing the right layout based on access patterns.

```
// Array of Structures
struct Point { float x, y, z; };
Point points[1000];

// Structure of Arrays
struct Points {
    float x[1000];
    float y[1000];
    float z[1000];
};
```

Array Applications and Use Cases

Arrays are fundamental in data processing applications:

1. Time Series Analysis: Sequential data points stored in arrays.
2. Statistical Computations: Calculating means, medians, standard deviations.
3. Signal Processing: Fast Fourier Transforms and other signal processing algorithms.

Unit 3: Searching And Sorting Algorithm

1.5 Searching Algorithms: Linear, Binary

Searching: Searching is one of the basic operations in computer science, which is used to search for a particular element in a data structure like array or list Linear Search and Binary Search are two of the most commonly used searching methods with different efficiency levels depending on the nature of the dataset. How and what search algorithm to pickdepends on the dataset size, ordering of elements, and time complexity.

Linear Search

It is the simplest searching algorithm. In this algorithm checks for the target element sequentially in the list until the target element is found or traversed the whole list. This algorithm can be applied to sorted as well as unsorted data sets. It begins at the first element and progresses towards the last element, comparing each value with the target. If the element is found, return the index of the element otherwise the failure indication (like -1 or Not Found). Working of Linear Search

1. Start from the first element of the array.
2. Compare the current element with the target element.
3. If they match, return the index (position) of the element.
4. If they don't match, move to the next element.
5. Repeat the process until the element is found or the entire list is traversed.
6. If the end of the list is reached without finding the element, return "Not Found".

Time Complexity of Linear Search

Case	Time Complexity	Explanation
Best Case	O(1)	The target element is found at the first position.
Average Case	O(n)	The target element is somewhere in the middle.
Worst Case	O(n)	The target element is at the last position or not present.

Example of Linear Search (Array Implementation in Python)

python
CopyEdit

```python
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i  # Return index if found
    return -1  # Return -1 if not found
arr = [10, 20, 30, 40, 50]
target = 30
result = linear_search(arr, target)
print(f"Element found at index {result}" if result != -1 else "Element not found")
```

Advantages of Linear Search

- Works on both sorted and unsorted lists.
- Simple and easy to implement.
- Requires no additional memory.

Disadvantages of Linear Search

- Slow for large datasets.
- Inefficient compared to other search algorithms.

Binary Search

Binary Search is a faster searching algorithm that applies only on sorted data. Returning to the algorithm tracking how many elements to check, it does not check half of the elements every step, so it divides the dataset by two and removes half elements. A divide and conquer approach, which means its much faster than Linear Search for larger datasets.

Working of Binary Search

1. Sort the array (if not already sorted).
2. Find the middle element of the array.
3. Compare the middle element with the target element.
   - If it matches, return the index.
   - If the target is less than the middle element, repeat the search in the left half.
   - If the target is greater than the middle element, repeat the search in the right half.
4. Continue until the target element is found or the search space reduces to zero.

Time Complexity of Binary Search

Case	Time Complexity	Explanation
Best Case	O(1)	The middle element is the target.
Average Case	O(log n)	The search space is divided in each step.
Worst Case	O(log n)	The target element is at the last level of recursion.

Example of Binary Search (Array Implementation in Python)

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
arr = [10, 20, 30, 40, 50]
target = 30
result = binary_search(arr, target)
print(f"Element found at index {result}" if result != -1 else "Element not found")
```

Advantages of Binary Search

- Much faster than Linear Search for large datasets.
- Reduces the number of comparisons by dividing the dataset.

Disadvantages of Binary Search

- Works only on sorted data.
- More complex than Linear Search to implement.
- Comparison of Linear Search vs. Binary Search

Feature	Linear Search	Binary Search
Efficiency	O(n) (slower)	O(log n) (faster)
Data Requirement	Works on any data	Works only on sorted data
Implementation	Simple and easy	More complex

Use Case	Small datasets, unordered lists	Large datasets, ordered lists
Memory Usage	No extra space needed	No extra space needed

Linear Search and Binary Search are important searching techniques, having their own pros and cons. Hence Linear Search is easy but time-consuming for large numbers of data; Binary Search, on the other hand, is complex but fast, and you need to have the data sorted. Linear Search is preferable if you search through an unordered dataset, meanwhile Binary Search is best if the dataset is already sorted as it has a logarithmic time complexity. To solve a problem, you need to know which algorithm works best for your problem constraints and the dataset type.

1.6 Sorting Algorithm—Insertion, Selection, Merge sort

Sorting is a basic operation in computer science that arranges elements in a required order (usually ascending or descending). Since searching, retrieving, and organizing data is a need in many applications, from databases to files, sorting is one of the fundamental things in computer science. There are numerous sorting algorithms, some which are more efficient than others depending on things like time complexity, space complexity, and stability. There are various sorting Algorithms like Insertion Sort, Selection Sort, Merge Sort, etc.

1. Insertion Sort

The simple, comparison-based Insertion Sort algorithm builds the end sorted sequence one element at a time. It works a bit like sorting playing cards in a hand — every new card gets added to where it belongs in relation to cards that are already in order.

Working Mechanism

1. Start with the second element (since a single element is already sorted).
2. Compare the element with the previous elements and shift them if necessary.
3. Insert the element in its correct position.
4. Repeat the process for all elements until the list is sorted.

Example

Unsorted Array: [7, 3, 5, 2]

Pass	Array State
1st	[3, 7, 5, 2]
2nd	[3, 5, 7, 2]
3rd	[2, 3, 5, 7]

Time Complexity

Case	Complexity	Explanation
Best Case	$O(n)$	Already sorted array, only one comparison per element.
Average Case	$O(n^2)$	Elements inserted at different positions with shifting required.
Worst Case	$O(n^2)$	Reverse sorted array, maximum shifting required.

Python 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
arr = [7, 3, 5, 2]
print("Sorted Array:", insertion_sort(arr))
```

Advantages & Disadvantages

Efficient for small datasets

Stable sorting algorithm (preserves order of duplicate elements)

Inefficient for large datasets

Slower compared to advanced sorting techniques

2. Selection Sort

Selection Sort algorithm: algorithm explains Selection Sort : Sort by repeatedly selecting the smallest element in the unsorted array and swapping it with the first unsorted element. It keeps two subarrays in a single array: the subarray which is sorted is left and the remaining is unsorted, and is kept reducing the unsorted subarray.

Working Mechanism

1. Find the smallest element in the unsorted part.
2. Swap it with the first unsorted element.
3. Move to the next element and repeat the process.

Example

Unsorted Array: [29, 10, 14, 37, 13]

Pass	Array State
1st	[10, 29, 14, 37, 13]
2nd	[10, 13, 14, 37, 29]
3rd	[10, 13, 14, 37, 29]
4th	[10, 13, 14, 29, 37]

Time Complexity

Case	Complexity	Explanation
Best Case	$O(n^2)$	Comparisons are always required.
Average Case	$O(n^2)$	Nested loops make it inefficient for large datasets.
Worst Case	$O(n^2)$	Even in the worst case, the number of comparisons remains $O(n^2)$.

Python Implementation

```
def selection_sort(arr):
    for i in range(len(arr)):
        min_idx = i
        for j in range(i + 1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr
arr = [29, 10, 14, 37, 13]
print("Sorted Array:", selection_sort(arr))
```

Advantages & Disadvantages

Simple and easy to implement

Performs well with small lists

Inefficient for large datasets

Not a stable sorting algorithm

3. Merge Sort

Merge Sort Merge Sort is a divide and conquer algorithm. It is very efficient and is used in applications where stability and efficiency are important.

Working Mechanism

1. Divide the array into two halves.
2. Recursively sort each half.
3. Merge the sorted halves to form the final sorted array.

Example

Unsorted Array: [38, 27, 43, 3, 9, 82, 10]

4. Divide into [38, 27, 43] and [3, 9, 82, 10]
5. Further divide: [38, 27], [43], [3, 9], [82, 10]
6. Merge step-by-step until sorted: [3, 9, 10, 27, 38, 43, 82]

Time Complexity

Case	Complexity	Explanation
Best Case	O(n log n)	Always divides the array into two equal halves.
Average Case	O(n log n)	Consistently efficient for large datasets.
Worst Case	O(n log n)	Even in the worst case, maintains O(n log n).

Python Implementation

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]
        merge_sort(left_half)
        merge_sort(right_half)
        i = j = k = 0
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
```

```
            j += 1
        k += 1
    while i < len(left_half):
        arr[k] = left_half[i]
        i += 1
        k += 1
    while j < len(right_half):
        arr[k] = right_half[j]
        j += 1
        k += 1
arr = [38, 27, 43, 3, 9, 82, 10]
merge_sort(arr)
print("Sorted Array:", arr)
```

Advantages & Disadvantages

Efficient for large datasets

Stable sorting algorithm

Consistent O(n log n) performance

Requires extra memory (O(n) space complexity)

Slower than Quick Sort for small datasets

Comparison of Sorting Algorithms

Algorithm	Best Case	Average Case	Worst Case	Space Complexity	Stable?
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes

Out of these sorting algorithms, Insertion sort is best for small datasets, Selection sort is simple but inefficient, and Merge sort is quite efficient on large datasets. Different sorting algorithms have different time and space complexities depending on the size of the dataset and if you need a sort that preserves the order of elements with equal values.

Multiple-Choice Questions (MCQs)

1. **Which of the following best defines an Abstract Data Type (ADT)?**

a) A data type defined by its implementation details
b) A data type defined by its behavior and operations
c) A data type with no defined operations
d) A data type only used in object-oriented programming
(Answer: b)

2. **Which of the following is a linear data structure?**
a) Tree
b) Graph
c) Queue
d) Hash Table
(Answer: c)

3. **Which of the following is a characteristic of an array?**
a) Elements can be inserted dynamically anywhere
b) Elements are stored in contiguous memory locations
c) Elements are always sorted
d) The size of the array increases automatically
(Answer: b)

4. **Which searching algorithm works efficiently with sorted arrays?**
a) Linear Search
b) Binary Search
c) Breadth-First Search
d) Depth-First Search
(Answer: b)

5. **What is the worst-case time complexity of Linear Search?**
a) O(1)
b) O(log n)
c) O(n)
d) $O(n^2)$
(Answer: c)

6. **Which sorting algorithm repeatedly finds the smallest element and moves it to the front?**
a) Merge Sort
b) Insertion Sort
c) Selection Sort
d) Quick Sort
(Answer: c)

7. **Which sorting algorithm has a worst-case time complexity of O(n log n)?**

a) Bubble Sort
b) Merge Sort
c) Selection Sort
d) Insertion Sort
(Answer: b)

8. **What is the primary advantage of Merge Sort over Insertion Sort?**
a) It is easier to implement
b) It performs better for large datasets
c) It requires no extra space
d) It works best on nearly sorted arrays
(Answer: b)

9. **In Binary Search, what happens if the middle element is smaller than the target value?**
a) The left half of the array is searched
b) The right half of the array is searched
c) The algorithm terminates immediately
d) The entire array is searched again
(Answer: b)

10. **Which data structure is best suited for implementing a queue?**
a) Stack
b) Array
c) Linked List
d) Graph
(Answer: c)

Short Questions

1. What is the difference between data types and abstract data types (ADTs)?
2. List two advantages and disadvantages of using arrays.
3. How does Linear Search work, and when is it useful?
4. What is the difference between Linear Search and Binary Search?
5. Explain the basic concept of sorting and why it is important in data structures.
6. What is the main difference between Selection Sort and Insertion Sort?
7. Why is Merge Sort considered more efficient than Selection Sort?
8. Define the worst-case time complexity of Binary Search.

9. What is the primary difference between static and dynamic arrays?
10. How does memory allocation work in sequential data structures?

Long Questions

1. Explain the concept of Abstract Data Types (ADTs) and their importance in programming.
2. Discuss arrays in detail, including their properties, classification, and memory allocation.
3. Explain the working of Linear Search and Binary Search, and compare their time complexities.
4. Describe Insertion Sort, Selection Sort, and Merge Sort, comparing their advantages and disadvantages.
5. Write a C or Python program to implement Binary Search and explain how it works.
6. Analyze the time complexity of different sorting algorithms and compare their performances.
7. Explain the significance of data structures in programming and how they improve efficiency.
8. How does the divide-and-conquer strategy apply to sorting algorithms like Merge Sort?
9. Discuss real-world applications of searching and sorting algorithms in software development.
10. Implement Selection Sort in Python/C, and provide a step-by-step explanation of its working.

MODULE 2
STACK, QUEUE AND RECURSION

LEARNING OUTCOMES

By the end of this Unit, students will be able to:

- Understand the sequential representation of stacks, their operations, and applications such as expression evaluation and function calls.
- Explain recursion, its working mechanism, and its applications in algorithm design.
- Learn about queues, their sequential representation, and different variations such as Dequeue (Double-ended Queue) and Priority Queue.
- Implement and analyze stack, queue, and recursion-based algorithms for efficient problem-solving.

Unit 4: Stack

2.1 Representation of Stacks using sequential organization, Applications

Stack is a linear data structure which follows Last In First Out order (LIFO). That is, the last element added (the top of the stack) is the first element to be removed. The knowledge of stacks is widely used in programming, memory management and real application such as undo-redo, function calls, etc.

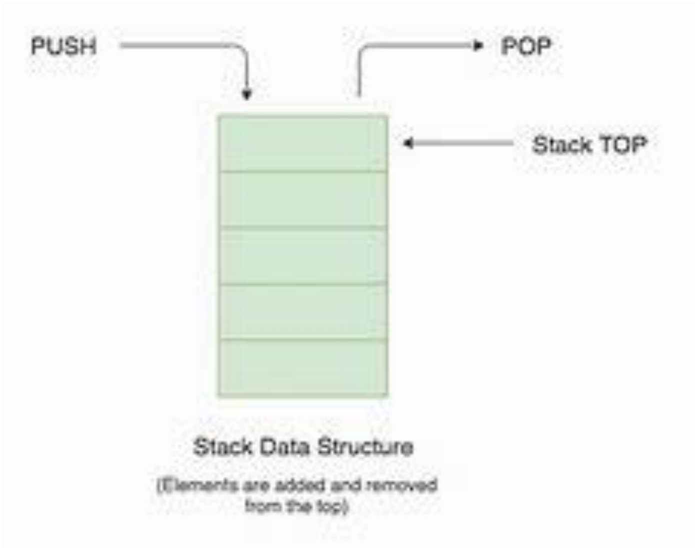


Figure 2.1: Stack Data Structure
[Source: https://th.bing.com/]

1. Representation of Stacks Using Sequential Organization

And we can implement stacks using arrays, which mean that elements are in adjacent memory locations (sequential memory). This method offers quick access but has a defined size in that the stack can't expand beyond the size allocated for it..

Structure of Stack Using an Array

A stack consists of the following:

1. An array to store elements.
2. A variable top, which indicates the index of the top element in the stack.
3. Stack operations such as push, pop, peek, and isEmpty.

Stack Operations Using Sequential Organization (Array)

Operation	Description	Time Complexity
Push (Insertion)	Adds an element to the top of the stack.	O(1)
Pop (Deletion)	Removes the top element from the stack.	O(1)
Peek (Top Element)	Retrieves the top element without removing it.	O(1)
isEmpty	Checks if the stack is empty.	O(1)

Stack Representation Using an Array

Example (Stack of Size 5)

Index	Stack Content
0	10
1	20
2	30
3	40
4 (Top)	50

2. Implementation of Stack Using an Array in Python

```
class Stack:
    def __init__(self, size):
        self.size = size
        self.stack = [None] * size  # Fixed-size array
        self.top = -1  # Stack is empty initially

    def push(self, value):
        if self.top == self.size - 1:
            print("Stack Overflow! Cannot push", value)
        else:
            self.top += 1
            self.stack[self.top] = value
            print(value, "pushed to stack")
    def pop(self):
```

```
        if self.top == -1:
            print("Stack Underflow! Cannot pop")
        else:
            popped_value = self.stack[self.top]
            self.top -= 1
            print(popped_value, "popped from stack")
            return popped_value
    def peek(self):
        if self.top == -1:
            print("Stack is empty")
        else:
            return self.stack[self.top]
    def is_empty(self):
        return self.top == -1
# Example Usage
s = Stack(5)
s.push(10)
s.push(20)
s.push(30)
print("Top Element:", s.peek())  # Output: 30
s.pop()
print("Stack Empty?", s.is_empty())  # Output: False
```

Advantages & Disadvantages of Sequential Stack Representation

Fast operations (O(1) time complexity for push/pop).

Simple to implement using an array.

Fixed size (cannot grow dynamically).

Wasted memory if the stack is not fully utilized.

3. Applications of Stacks

Real-Life Applications of Stacks in Programming, OS, and Daily)).

1. Function Call Management in Programming

- Function calls in programming follow a stack structure.
- When a function is called, it is pushed onto the call stack.
- When the function completes execution, it is popped from the stack.
- This is used for recursive function calls.

2. Undo & Redo Functionality

- In text editors, the undo feature works using a stack.
- When an action is performed, it is pushed onto the stack.

- Undoing an action pops the last operation and restores the previous state.

3. Expression Evaluation (Infix to Postfix/Prefix Conversion)

- Mathematical expressions like (A + B) * C are evaluated using stacks.
- Operators and operands are pushed and popped from the stack during conversion.

4. Backtracking (Maze Solving, Pathfinding, Game Moves)

- Stacks help in solving mazes by storing visited paths.
- In chess, moves are stored in a stack, allowing undoing moves.

5. Parentheses Matching in Compilers

- Stacks are used in syntax checking of expressions like {[()()]}.
- Each opening bracket is pushed onto the stack.
- When a closing bracket is found, the stack is popped to match them.

6. Browser Back & Forward Navigation

- Browsers use two stacks for navigation.
- When moving back, the current page is pushed onto a forward stack.
- When moving forward, the page is popped from the forward stack.

Stack Abstract Data Type Stack abstract data type are typically used using sequential organization (arrays). Simple to implement and offer fast operations, but they are limited by their fixed size. Stacks are an important data structure in computing, used extensively for programming, undo-redo features, function calls, compiler and browser navigation, and much more.

Unit 5: Recursion

2.2 Recursion and its applications

Recursion is a method of trying to solve a problem by calling a function that calls itself. Recursion uses sub-sub problems until a base condition is met instead of using loops. Its main use is in various algorithms such as divide and conquer, backtracking or tree traversal (including depth-first search).

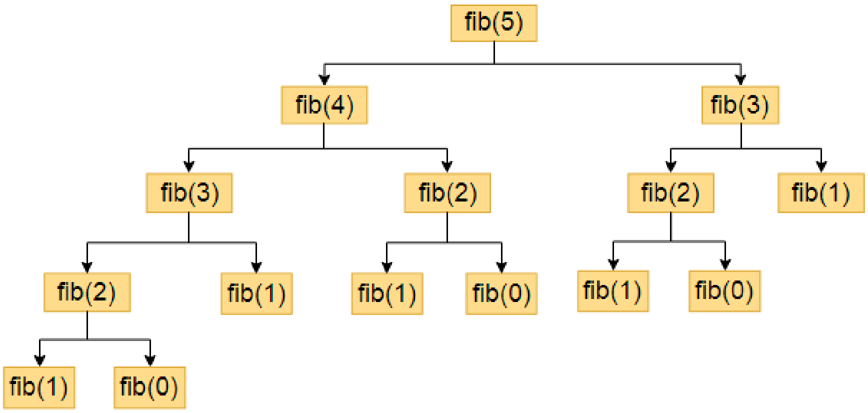


Figure 2.2: Recursion
[Source: https://www.studyfame.com/]

Key Components of Recursion

1. Base Case – The stopping condition that ends the recursion.
2. Recursive Case – The function calls itself with a modified parameter to approach the base case.

Example: Factorial Calculation Using Recursion

Factorial of n (n!) is defined as:

n!=n×(n−1)×(n−2)×...×1n! = n \times (n-1) \times (n-2) \times ... \times 1n!=n×(n−1)×(n−2)×...×1

Using recursion:

factorial(n)=n×factorial(n−1)factorial(n) = n \times factorial(n-1)factorial(n)=n×factorial(n−1)

```
def factorial(n):
    if n == 0:  # Base case
        return 1
    return n * factorial(n - 1)  # Recursive case

print(factorial(5))  # Output: 120
```

Types of Recursion

1. Direct Recursion

- A function directly calls itself.
- Example: Factorial calculation.

2. Indirect Recursion

- A function calls another function, which in turn calls the first function.

```
def functionA(n):
    if n > 0:
        print(n, end=" ")
        functionB(n - 1)

def functionB(n):
    if n > 0:
        print(n, end=" ")
        functionA(n - 1)
functionA(5)  # Output: 5 4 3 2 1 1 2 3 4
```

3. Tail Recursion

- The recursive call is the last statement in the function.
- Optimized by compilers to avoid excessive function calls.

```
def tail_recursive_factorial(n, result=1):
    if n == 0:
        return result
    return tail_recursive_factorial(n - 1, result * n)

print(tail_recursive_factorial(5))  # Output: 120
```

4. Non-Tail Recursion

- The function performs operations after the recursive call.

```
def non_tail_recursive_factorial(n):
    if n == 0:
        return 1
    return n * non_tail_recursive_factorial(n - 1)
print(non_tail_recursive_factorial(5))  # Output: 120
```

Applications of Recursion

1. Mathematical Computations

Factorial Calculation

Recursion is commonly used to compute factorials, as shown above.

Fibonacci Sequence

The Fibonacci sequence follows a recursive pattern:

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

```
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)
print(fibonacci(6))  # Output: 8
```

2. Data Structure Traversals

Tree Traversal

Recursion is used to traverse trees efficiently.

- Preorder Traversal (Root → Left → Right)
- Inorder Traversal (Left → Root → Right)
- Postorder Traversal (Left → Right → Root)

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
def inorder_traversal(root):
    if root:
        inorder_traversal(root.left)
        print(root.data, end=" ")
        inorder_traversal(root.right)
root = Node(1)
root.left = Node(2)
root.right = Node(3)
inorder_traversal(root)  # Output: 2 1 3
```

Graph Traversal (DFS - Depth First Search)

Recursion helps in graph traversal using Depth First Search (DFS).

```
def dfs(graph, node, visited=set()):
    if node not in visited:
        print(node, end=" ")
        visited.add(node)
        for neighbor in graph[node]:
            dfs(graph, neighbor, visited)
graph = {
    'A': ['B', 'C'],
```

```
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': [],
    'F': []
}
dfs(graph, 'A')  # Output: A B D E C F
```

3. Divide and Conquer Algorithms

Recursion is one of the methods that fall in the category of divide and conquer algorithms, where a larger problem is [divided into smaller subproblems..

Merge Sort

- Divide the array into two halves.
- Recursively sort each half.
- Merge the sorted halves.

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]
        merge_sort(left_half)
        merge_sort(right_half)
        i = j = k = 0
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1
        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1
        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
```

```
        k += 1
arr = [38, 27, 43, 3, 9, 82, 10]
merge_sort(arr)
print(arr)  # Output: [3, 9, 10, 27, 38, 43, 82]
```

Backtracking Algorithms

Backtracking is a technique for solving problems.

Solving the N-Queens Problem

```
def print_solution(board):
    for row in board:
        print(" ".join(row))
    print()
def is_safe(board, row, col, n):
    for i in range(col):
        if board[row][i] == 'Q':
            return False
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 'Q':
            return False
    for i, j in zip(range(row, n, 1), range(col, -1, -1)):
        if board[i][j] == 'Q':
            return False
    return True
def solve_n_queens(board, col, n):
    if col >= n:
        print_solution(board)
        return True
    res = False
    for i in range(n):
        if is_safe(board, i, col, n):
            board[i][col] = 'Q'
            res = solve_n_queens(board, col + 1, n) or res
            board[i][col] = '.'
    return res
n = 4
board = [['.' for _ in range(n)] for _ in range(n)]
solve_n_queens(board, 0, n)
```

Advantages & Disadvantages of Recursion

- Simplifies complex problems like tree traversal, graphs, and backtracking.
- Reduces code complexity, making it easier to read.
- Useful for divide and conquer problems like sorting.

Disadvantages

- High memory consumption due to function call stack.
- Slower execution due to repeated function calls.
- May cause stack overflow if the base case is not defined properly.

Recursion is an elegant way of solving problems used in mathematic problems, traversing data structures, divide-and-conquer techniques, and backtracking techniques. Its benefits notwithstanding, it has to be used judiciously to prevent performance problems.

Unit 6: Queue

2.3 Queue, Representation of Queues using sequential organization, Dequeue

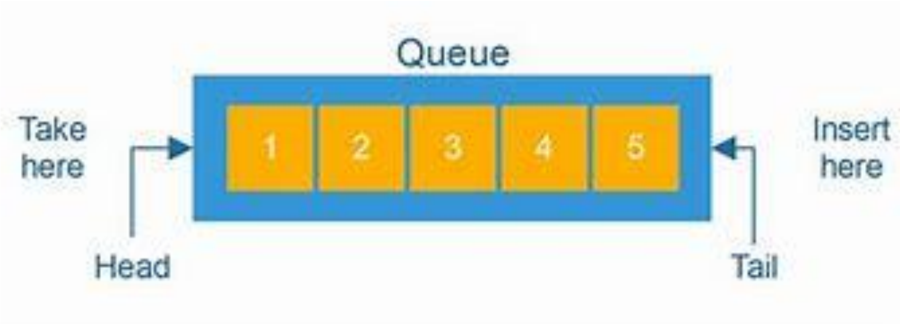


Figure 2.3: Queue
[Source: https://th.bing.com/]

A queue is a linear data structure which follows the First In, First Out (FIFO) order. So we insert elements at the back and recover them from the front. Queues have numerous applications, from scheduling and buffering to real-world scenarios such as printer queues, process scheduling, and customer service queues.

Basic Queue Operations

Operation	Description	Time Complexity
Enqueue (Insertion)	Adds an element at the rear of the queue.	O(1)
Dequeue (Deletion)	Removes an element from the front of the queue.	O(1)
Peek (Front Element)	Retrieves the front element without removing it.	O(1)
isEmpty	Checks if the queue is empty.	O(1)

1. Representation of Queues Using Sequential Organization (Arrays)

Another example of abstract data types: Queues, which are implemented on arrays, which is a collection of an area of memory. This is called sequential organization; that is, elements are in hard, physical order, and the memory is allocated in such a way that they are in contiguously located memory.

Structure of a Queue Using an Array

A queue contains:

- An array to store elements.
- Two pointers:
- front – Indicates the first element of the queue.
- rear – Indicates the last inserted element.

Example: Queue Representation Using an Array (Size = 5)

Index	**0**	**1**	**2**	**3**	**4**
Queue Content	10	20	30	40	50
Front	yes				
Rear					yes

Implementation of Queue Using an Array in Python

```
class Queue:
    def __init__(self, size):
        self.size = size
        self.queue = [None] * size  # Fixed-size array
        self.front = -1  # Indicates the front element
        self.rear = -1  # Indicates the rear element
    def enqueue(self, value):
        if self.rear == self.size - 1:
            print("Queue Overflow! Cannot enqueue", value)
        else:
            if self.front == -1:  # First element inserted
                self.front = 0
            self.rear += 1
            self.queue[self.rear] = value
            print(value, "added to queue")
    def dequeue(self):
        if self.front == -1 or self.front > self.rear:
            print("Queue Underflow! Cannot dequeue")
        else:
            print(self.queue[self.front], "removed from queue")
            self.front += 1  # Move front pointer
    def peek(self):
```

```
        if self.front == -1 or self.front > self.rear:
            print("Queue is empty")
        else:
            return self.queue[self.front]
    def is_empty(self):
        return self.front == -1 or self.front > self.rear
# Example Usage
q = Queue(5)
q.enqueue(10)
q.enqueue(20)
q.enqueue(30)
print("Front Element:", q.peek())  # Output: 10
q.dequeue()
print("Queue Empty?", q.is_empty())  # Output: False
```

Advantages & Disadvantages of Sequential Queue Representation

- Fast operations (O(1) time complexity for enqueue and dequeue).
- Simple to implement using arrays.
- Fixed size (cannot dynamically grow).
- Wasted memory due to unused spaces after deletion.

2. Circular Queue (Optimized Sequential Queue Representation)

In a simple queue, after several dequeues, the unused spaces cannot be reused. Circular queues consider this problem and make the queue circular such that the rear reaches the end, it wraps to the front In a straightforward queue, unused spaces cannot be reused after multiple dequeues. To solve this problem, circular queues make the queue circular, so, whenever the rear reaches the end of the queue, it is circularly wrapped around to the front end of the queue.

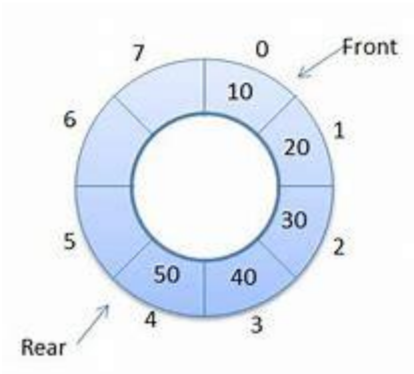


Figure 2.4: Circular Queue [Source: https://th.bing.com/]

Notes

Implementation of Circular Queue Using an Array in Python

```
class Circular Queue:
    def __init__(self, size):
        self.size = size
        self.queue = [None] * size
        self.front = -1
        self.rear = -1
    def enqueue(self, value):
        if (self.rear + 1) % self.size == self.front:
            print("Queue Overflow!")
        else:
            if self.front == -1:
                self.front = 0
            self.rear = (self.rear + 1) % self.size
            self.queue[self.rear] = value
            print(value, "added to circular queue")
    def dequeue(self):
        if self.front == -1:
            print("Queue Underflow!")
        else:
            print(self.queue[self.front], "removed from circular queue")
            if self.front == self.rear:  # Only one element left
                self.front = self.rear = -1
            else:
                self.front = (self.front + 1) % self.size
cq = CircularQueue(5)
cq.enqueue(10)
cq.enqueue(20)
cq.enqueue(30)
cq.dequeue()
cq.enqueue(40)
cq.enqueue(50)
cq.enqueue(60)  # Wraps around
```

3. Dequeue (Double-Ended Queue)

A Dequeue (Double-Ended Queue) is a linear queue where we can add and delete the elements from both ends, front and the rear. It supports two types:

1. Input-Restricted Dequeue – Insertion is not allowed at one end only, but deletion goes at both ends.
2. String Parse from String-to-String queue dequeue deque d Queue Stack Q Stack S Stack parse Stack S S Table S dequeue D Stack parse Stack S Table S Stack parse Stack S

Operations in a Dequeue

Operation	Description	Time Complexity
Insert at Front	Adds an element at the front.	O(1)
Insert at Rear	Adds an element at the rear.	O(1)
Delete from Front	Removes an element from the front.	O(1)
Delete from Rear	Removes an element from the rear.	O(1)

Implementation of Dequeue Using an Array in Python

```
from collections import deque
dq = deque()
# Insert at rear
dq.append(10)
dq.append(20)
print("Dequeue:", dq)
# Insert at front
dq.appendleft(5)
print("Dequeue after front insertion:", dq)
# Delete from front
dq.popleft()
print("Dequeue after front deletion:", dq)
# Delete from rear
dq.pop()
print("Dequeue after rear deletion:", dq)
```

Applications of Dequeue

Sliding Window Problems – Used in maximum/minimum sliding window calculations.

Job Scheduling – Tasks are processed from both ends based on priority.

Palindrome Checking – Characters can be compared from both ends.

4. Applications of Queues

1. Scheduling in Operating Systems
   - CPU process scheduling follows FIFO queues.
   - Disk scheduling algorithms use priority queues.
2. Print Queue in Printers
   - Print jobs are handled using FIFO queues, ensuring first-come, first-served.
3. Network & Data Buffering
   - Packets are queued before transmission in routers and switches.
   - Video streaming buffers use queues for smooth playback.
4. Call Center and Customer Service
   - Customer support calls follow FIFO queues for fair handling.
   - Queues and deques are one of the important data structures used in scheduling, buffering and in many other real world applications. They are sequential with arrays that provide faster operation time but circular queues and deques allow more flexibility in insertion and deletion. They are used to solve efficient algorithmic problems such as process scheduling, buffering in computing, and task management, which makes understanding these structures important

2.4 Priority Queue

A Priority Queue is a special type of queue where each element has some priority associated with it. A priority queue is a special type of queue that is different from a normal queue where elements are processed in a FIFO (First In, First Out) order.

Key Features of a Priority Queue

Each element has a priority value.

Higher-priority elements are dequeued before lower-priority elements.

If two elements have the same priority, they follow FIFO order.

Example of a Priority Queue

Think of a hospital emergency room that treats patients according to how serious their condition is, not the order they arrived.

Patient Name	Condition	Priority Level
Alice	Mild fever	3 (Low)
Bob	Fracture	2 (Medium)
Charlie	Heart Attack	1 (High)

Types of Priority Queues

1. **Min-Priority Queue**
   - The lowest-priority element is dequeued first.
   - Example: Dijkstra's Algorithm (finding shortest paths).
2. **Max-Priority Queue**
   - The highest-priority element is dequeued first.
   - Example: Task scheduling, emergency services.

Implementation of Priority Queue

1. Using a List (Naïve Approach)

The elements are stored in an unordered list and the element with the highest/lowest priority is found by the time of deletion (O(n) time complexity).

```
class PriorityQueue:
    def __init__(self):
        self.queue = []
    def enqueue(self, item, priority):
        self.queue.append((item, priority))
    def dequeue(self):
        if not self.queue:
            return "Queue is empty"
        self.queue.sort(key=lambda x: x[1])  # Sort by priority (min first)
        return self.queue.pop(0)[0]  # Remove the highest priority element
pq = PriorityQueue()
pq.enqueue("Alice", 3)
pq.enqueue("Bob", 2)
pq.enqueue("Charlie", 1)
print(pq.dequeue())  # Output: Charlie (highest priority)
```

2. Using a Heap (Efficient Approach)

A binary heap (Min-Heap or Max-Heap) is used to insert and delete in O(log n) time complexity.

```
import heapq
class PriorityQueue:
    def __init__(self):
        self.queue = []
    def enqueue(self, item, priority):
        heapq.heappush(self.queue, (priority, item))  # Min-Heap (lowest
priority first)
    def dequeue(self):
        if not self.queue:
```

```
            return "Queue is empty"
        return heapq.heappop(self.queue)[1]  # Remove highest priority item
pq = PriorityQueue()
pq.enqueue("Alice", 3)
pq.enqueue("Bob", 2)
pq.enqueue("Charlie", 1)
print(pq.dequeue())  # Output: Charlie
```

4. Applications of Priority Queue

CPU Scheduling – Processes with higher priority execute first.

Graph Algorithms – Used in Dijkstra's and *A Algorithm** for shortest path.

Data Compression (Huffman Coding) – Nodes with lower frequency get higher priority.

Network Packet Scheduling – Important packets (like VoIP) are sent first.

Event-Driven Simulations – Events with higher importance are processed first.

5. Comparison of Priority Queue Implementations

Implementation Method	Enqueue Time Complexity	Dequeue Time Complexity	Space Complexity
Unsorted List	O(1)	O(n)	O(n)
Sorted List	O(n)	O(1)	O(n)
Binary Heap (Min/Max-Heap)	O(log n)	O(log n)	O(n)

A priority queue is a data structure that enables retrieval of highest priority elements first, rather than insertion order. It has its applications in CPU scheduling, graph algorithms, network routing, and simulations. A heap-based implementation has the added benefit of decent performance for real-world applications.

Multiple-Choice Questions (MCQs)

1. Which data structure follows the Last-In, First-Out (LIFO) principle?
   a) Queue
   b) Stack
   c) Linked List
   d) Priority Queue

(Answer: b)

2. Which operation removes the top element from a stack?
   a) Enqueue
   b) Pop
   c) Push
   d) Peek

   (Answer: b)
3. What is a common application of stacks in programming?
   a) Managing function calls
   b) Scheduling processes
   c) Searching in an unordered list
   d) Sorting data

   (Answer: a)

4. Which of the following problems is best solved using recursion?
   a) Fibonacci sequence
   b) Tower of Hanoi
   c) Tree traversal
   d) All of the above

   (Answer: d)
5. What differentiates a queue from a stack?
   a) A queue follows LIFO, while a stack follows FIFO
   b) A stack follows FIFO, while a queue follows LIFO
   c) A queue follows FIFO, while a stack follows LIFO
   d) Both follow the LIFO principle

   (Answer: c)
6. Which of the following is NOT a type of queue?
   a) Circular Queue
   b) Dequeue
   c) Priority Queue
   d) Hash Queue

   (Answer: d)
7. What happens when a recursive function lacks a base case?
   a) It executes once and terminates
   b) It results in an infinite recursion, causing a stack overflow
   c) It returns a NULL value
   d) The compiler automatically adds a base case

   (Answer: b)

8. Which of the following operations is performed at both ends in a dequeue?
   a) Insert
   b) Delete
   c) Both Insert and Delete
   d) None of the above
   (Answer: c)
9. Which queue variation assigns priorities to elements for processing?
   a) Circular Queue
   b) Dequeue
   c) Priority Queue
   d) Stack Queue
   (Answer: c)
10. Which data structure is commonly used for backtracking problems?
   a) Queue
   b) Stack
   c) Hash Table
   d) Tree
   (Answer: b)

Short Questions

1. Define stack and list its primary operations.
2. Explain recursion with an example.
3. What is the difference between iteration and recursion?
4. Describe the FIFO principle in queues.
5. What is a priority queue, and how is it different from a normal queue?
6. Explain how stacks are used for function calls in programming.
7. What is a circular queue, and why is it useful?
8. List two real-world applications of recursion.
9. What is the difference between push and pop operations in a stack?
10. How can recursion be converted into iteration?

Long Questions

1. Explain stack operations with a detailed example, including push, pop, and peek operations.

2. Discuss recursion in-depth, including base cases and recursive function execution flow.
3. Write a program to implement stack operations using an array.
4. Describe queues and their variations, such as circular queues, deques, and priority queues.
5. Compare and contrast stacks and queues, highlighting their use cases.
6. Implement a recursive algorithm to compute the Fibonacci sequence and explain its execution.
7. Explain how recursion works in the Tower of Hanoi problem and provide a solution.
8. Describe expression evaluation using stacks, including infix, prefix, and postfix notations.
9. Write a program to implement a queue using an array, including enqueue and dequeue operations.
10. Discuss how recursion can be optimized using memorization or iterative approaches.

MODULE 3
LINKED LIST

3.0 LEARNING OUTCOMES

By the end of this chapter, students will be able to:

- Understand the concept of linked lists, their representation, and advantages over arrays.
- Perform operations on linked lists, including traversing, searching, insertion, and deletion.
- Learn about memory allocation in linked lists and how dynamic memory is managed using pointers.

Unit 7: Linked list

3.1 Linked list and its representation

The linked list is a linear data structure in which the elements are not stored at contiguous memory locations but are linked using pointers. A linked list node consists of:

1. **Data** – The actual value stored in the node.
2. **Pointer (Next)** – A reference to the next node in the list.

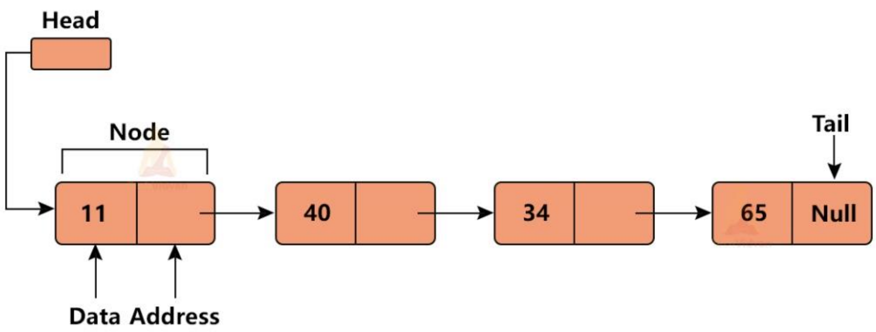


Figure 3.1: Linked List
[Source: https://techvidvan.com/]

Comparison of Linked List vs. Array

Feature	Linked List	Array
Memory Allocation	Dynamic	Fixed Size
Insertion/Deletion	O(1) (at beginning), O(n) (at middle/end)	O(n) (requires shifting)
Access Time	O(n) (sequential access)	O(1) (direct access)
Extra Space	Requires extra space for pointers	No extra space needed

2. Types of Linked Lists

1. Singly Linked List – Each node points to the next node.
2. Doubly Linked List – Each node has two pointers (next and previous).
3. Circular Linked List – The last node points back to the first node.

3. Representation of Linked List

Structure of a Node (Singly Linked List)

python

CopyEdit

```
class Node:
    def __init__(self, data):
        self.data = data  # Store the data
        self.next = None  # Pointer to the next node
```

Basic Operations in Linked List

Operation	**Description**
Insertion	Add a new node at the beginning, end, or middle.
Deletion	Remove a node from the list.
Traversal	Move through the list to access elements.

Linked List Implementation in Python

python

CopyEdit

```
class Node:
```

```
    def __init__(self, data):
        self.data = data
        self.next = None
class LinkedList:
    def __init__(self):
        self.head = None
    def insert_at_end(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        temp = self.head
        while temp.next:
            temp = temp.next
        temp.next = new_node
    def display(self):
        temp = self.head
        while temp:
            print(temp.data, end=" -> ")
            temp = temp.next
        print("None")
# Example Usage
ll = LinkedList()
ll.insert_at_end(10)
```

```
ll.insert_at_end(20)

ll.insert_at_end(30)

ll.display()  # Output: 10 -> 20 -> 30 -> None
```

4. Advantages & Disadvantages of Linked List

- Dynamic size allocation (efficient memory utilization).
- Efficient insertions and deletions compared to arrays.
- Extra memory required for pointers.
- Slower access time (O(n) vs. O(1) for arrays).

5. Applications of Linked Lists

- Memory management (Dynamic Allocation).
- Implementation of stacks and queues.
- Undo-Redo functionality in text editors.

Graph representation (Adjacency List).

Linked List: It is a mutable and useful data structure for dynamic memory allocation and efficient insertions and deletions. It is often used in data structures (stack, queue, graph), although it needs additional memory for pointers.

Unit 8: Operations on Linked list

3.2 Operations on Linked list: Traversing, Searching, Insertion, Deletion

A linked list is a data structure made up of nodes wherein each node is linked through pointers. Linked lists are often used because of the relatively high number of operations that can be performed on them, such as traversing, searching, inserting and deleting. Operations are helpful to optimize the linked list elements operation.

1. Traversing a Linked List

Traversing the Linked List means going through the linked list one by one and getting its data. And since linked lists does not contain arrays with contiguous memory, following next pointer of each node make it necessary to traverse them one by one.

Algorithm for Traversing

1. Start from the head node.
2. Print or process the data of the current node.
3. Move to the next node using the next pointer.
4. Repeat until the end of the list (None) is reached.

Implementation in Python

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
class LinkedList:
    def __init__(self):
        self.head = None
```

```python
    def insert_at_end(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        temp = self.head
        while temp.next:
            temp = temp.next
        temp.next = new_node
    def traverse(self):
        temp = self.head
        while temp:
            print(temp.data, end=" -> ")
            temp = temp.next
        print("None")
# Example Usage
ll = LinkedList()
ll.insert_at_end(10)
ll.insert_at_end(20)
ll.insert_at_end(30)
ll.traverse()  # Output: 10 -> 20 -> 30 -> None
```

Time Complexity:

O(n) – Each node is visited once.

2. Searching in a Linked List

Notes

It involves getting whether a specific value exists in the linked list and retracing steps to the position (index) if it does. Because linked lists do not allow direct indexing, a search is performed by traversing through each of the nodes sequentially.

Algorithm for Searching

1. Start from the head node.
2. Compare the data of the current node with the target value.
3. If found, return the position of the node.
4. If not found, move to the next node.
5. Repeat until the end of the list is reached.

Implementation in Python

```
class LinkedList:
    def __init__(self):
        self.head = None
    def insert_at_end(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        temp = self.head
        while temp.next:
            temp = temp.next
        temp.next = new_node
    def search(self, key):
        temp = self.head
```

```python
        position = 0
        while temp:
            if temp.data == key:
                return f"Element found at index {position}"
            temp = temp.next
            position += 1
        return "Element not found"
# Example Usage
ll = LinkedList()
ll.insert_at_end(10)
ll.insert_at_end(20)
ll.insert_at_end(30)
print(ll.search(20))  # Output: Element found at index 1
print(ll.search(40))  # Output: Element not found
```

Time Complexity:

O(n) – Each node is checked once.

3. Insertion in a Linked List

Insertion is the process of adding a new node at a specific position. There are three common cases:

1. At the beginning (Head Insertion)
2. At the end (Tail Insertion)
3. In the middle (Between two nodes)

Algorithm for Insertion

1. Create a new node with the given data.
2. Adjust pointers based on insertion position.

3. Update the next reference of the previous node.

Implementation in Python

```
class LinkedList:
    def __init__(self):
        self.head = None
    def insert_at_beginning(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node
    def insert_at_end(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        temp = self.head
        while temp.next:
            temp = temp.next
        temp.next = new_node
    def insert_at_position(self, data, position):
        new_node = Node(data)
        if position == 0:  # Insert at the beginning
            new_node.next = self.head
            self.head = new_node
            return
```

```
        temp = self.head
        for _ in range(position - 1):
            if not temp:
                return "Position out of bounds"
            temp = temp.next
        new_node.next = temp.next
        temp.next = new_node

# Example Usage
ll = LinkedList()
ll.insert_at_end(10)
ll.insert_at_end(30)
ll.insert_at_position(20, 1)  # Insert 20 at index 1
ll.traverse()  # Output: 10 -> 20 -> 30 -> None
```

Time Complexity:

O(1) for beginning insertion
O(n) for middle/end insertion

4. Deletion in a Linked List

Deletion involves removing a node from the list. Common cases include:

1. Deleting the first node (Head deletion).
2. Deleting a node in the middle.
3. Deleting the last node (Tail deletion).

Algorithm for Deletion

1. If list is empty, return "Underflow."
2. If deleting the first node, update head.

3. If deleting a middle node, adjust the next pointer of previous node.
4. If deleting last node, set previous node's next to None.

Implementation in Python

```
class LinkedList:
    def __init__(self):
        self.head = None
    def insert_at_end(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        temp = self.head
        while temp.next:
            temp = temp.next
        temp.next = new_node
    def delete_node(self, key):
        temp = self.head
        # Deleting first node
        if temp and temp.data == key:
            self.head = temp.next
            temp = None
            return
        # Deleting middle or last node
        prev = None
```

```
        while temp and temp.data != key:
            prev = temp
            temp = temp.next
        if temp is None:
            return "Element not found"
        prev.next = temp.next
        temp = None

# Example Usage
ll = LinkedList()
ll.insert_at_end(10)
ll.insert_at_end(20)
ll.insert_at_end(30)
ll.delete_node(20)  # Delete node with value 20
ll.traverse()  # Output: 10 -> 30 -> None
```

Time Complexity:

O(1) for deleting first node
O(n) for deleting middle/last node

Insertion/deletion in linked lists is more efficient, and memory can be allocated dynamically while in arrays it cannot, as they have static memory allocation. But, they need to be traversed sequentially for search and access. The methods utilized in basic operations (traversal, searching, insertion, deletion) constitute the basis for complex data structures like stacks, queues, and graphs.

Unit 9: Memory Allocation

3.3 Memory Allocation

This action is typically taken at run time when the program is executed. It protects the overall performance, reduces extra usage of memory, and avoids situations where insufficient memory leads to crashes.

Types of Memory in a Computer System

Memory Type	Description
Primary Memory (RAM)	Temporary, volatile storage used by the CPU for fast access.
Secondary Memory (HDD/SSD)	Non-volatile, used for long-term data storage.
Cache Memory	High-speed memory for frequently accessed data.
Register Memory	Small, fastest memory directly inside the CPU.

1. Types of Memory Allocation

Memory allocation is classified into two main types:

1. Static Memory Allocation
2. Dynamic Memory Allocation

Static Memory Allocation

- Memory is assigned before program execution (at compile time).
- The memory size is immutable and cannot be altered during runtime.
- Faster execution since memory is pre-allocated.

- Uses stack memory for storage.

Example (Static Memory Allocation in C)

```
int arr[5];  // Fixed size array (allocated at compile time)
```

Advantages:
Faster execution
No memory fragmentation
Disadvantages:
Wastage of memory if unused
Cannot allocate memory dynamically

2. Dynamic Memory Allocation

- Memory is allocated during program execution (at runtime).
- Size is flexible, and memory can be allocated or deallocated as needed.
- Uses heap memory for storage.

Example (Dynamic Memory Allocation in C)

```
int *ptr = (int*) malloc(5 * sizeof(int));   // Allocating memory dynamically
```

Advantages:
Efficient memory usage
Can allocate or free memory as needed
Disadvantages:
Slower execution due to runtime allocation
Memory leaks if not properly deallocated

3. Methods of Dynamic Memory Allocation in C/C++

Function	Description	Header File
malloc(size)	Allocates a block of memory but does not initialize it.	<stdlib.h>
calloc(n, size)	Allocates multiple blocks and initializes them to zero.	<stdlib.h>
realloc(ptr, size)	Resizes a previously allocated memory block.	<stdlib.h>
free(ptr)	Deallocates memory to prevent memory leaks.	<stdlib.h>

Example (Dynamic Memory Allocation Using malloc in C)

```
#include <stdio.h>

#include <stdlib.h>

int main() {

   int *ptr = (int*) malloc(5 * sizeof(int));  // Allocating memory for 5 integers

   if (ptr == NULL) {

      printf("Memory allocation failed!");

      return 1;

   }

   for (int i = 0; i < 5; i++)

      ptr[i] = i * 10;  // Assigning values
```

```
    for (int i = 0; i < 5; i++)
        printf("%d ", ptr[i]);  // Output: 0 10 20 30 40
    free(ptr);  // Deallocating memory
    return 0;
}
```

4. Memory Allocation in Data Structures

1. Stack Memory Allocation (Static)

- Stores function calls, local variables, and recursion data.
- Memory is automatically allocated and deallocated.
- Limited size (stack overflow can occur).

2. Heap Memory Allocation (Dynamic)

- Stores dynamically allocated memory (e.g., linked lists, trees).
- Memory must be manually managed (malloc/free).
- Larger than stack memory but slower access.

5. Common Memory Allocation Issues

Issue	Description
Memory Leak	Forgetting to free dynamically allocated memory.
Dangling Pointer	Accessing memory after it has been freed.
Fragmentation	Memory is divided into small unused blocks, reducing efficiency.
Buffer Overflow	Writing more data than allocated, leading to crashes.

Example of a Memory Leak (C)

```
void memory_leak() {
    int *ptr = (int*) malloc(5 * sizeof(int));  // Allocated memory
    // Forgot to free memory -> Memory leak!
}
```

Solution: Always use free(ptr) after allocation.

It is a very crucial concept in programming as memory allocation helps to manage resources efficiently. Static allocation is easy but rigid, and dynamic allocation gives you flexibility but requires properly managing the memory. The correct management of memory will guard against leaks, fragmentation and buffer overflows, which would otherwise make your program less efficient.

Multiple-Choice Questions (MCQs)

1. **Which of the following is an advantage of linked lists over arrays?**
   a) Faster access to elements using indexing
   b) Dynamic memory allocation
   c) Fixed size allocation
   d) Requires less memory per node
   (Answer: b)

2. **Which type of linked list allows traversal in both directions?**
   a) Singly Linked List
   b) Doubly Linked List
   c) Circular Linked List
   d) None of the above
   (Answer: b)

3. **What is the time complexity of inserting an element at the beginning of a linked list?**

   a) O(1)

   b) O(n)

   c) O(log n)

   d) $O(n^2)$

   (Answer: a)

4. **Which operation is most efficient in a linked list compared to an array?**

   a) Accessing an element at a specific index

   b) Deleting an element from the middle

   c) Sorting elements

   d) Merging two lists

   (Answer: b)

5. **What does the 'head' pointer in a linked list represent?**

   a) The last node in the list

   b) The middle node in the list

   c) The first node in the list

   d) A temporary pointer for traversal

   (Answer: c)

6. **Which type of linked list has its last node pointing to the first node?**

   a) Singly Linked List

   b) Doubly Linked List

   c) Circular Linked List

   d) Multi-Level Linked List

   (Answer: c)

7. **What happens when a node is deleted from a singly linked list?**

   a) The previous node's next pointer is updated

   b) The entire list is deleted

   c) Memory for all nodes is freed

d) The previous node becomes the last node

(Answer: a)

8. **Which of the following statements is true about linked lists?**

   a) They have a fixed size

   b) They allow efficient random access

   c) They use dynamic memory allocation

   d) They are always slower than arrays

   (Answer: c)

9. **What is the primary disadvantage of linked lists?**

   a) Fixed memory allocation

   b) Higher memory overhead due to pointers

   c) Inefficient insertion and deletion

   d) Cannot store data dynamically

   (Answer: b)

10. **Which function is used to allocate memory dynamically in a linked list in C?**

    a) malloc()

    b) calloc()

    c) free()

    d) Both a and b

    (Answer: d)

Short Questions

1. What is a linked list, and how does it differ from an array?
2. List the advantages of linked lists over arrays.
3. What are the different types of linked lists, and how do they differ?
4. How is memory allocated dynamically for linked lists?
5. What is a circular linked list, and where is it used?
6. Explain the difference between singly and doubly linked lists.

7. What are the main operations performed on a linked list?
8. How is traversal performed in a linked list?
9. Explain the memory overhead issue in linked lists.
10. How do you delete a node from a singly linked list?

Long Questions

1. Explain the structure of a linked list and how it is represented in memory.
2. Discuss the advantages and disadvantages of linked lists compared to arrays.
3. Write a C program to implement a singly linked list with insertion and deletion operations.
4. Describe the traversal, searching, and insertion operations in linked lists with examples.
5. Explain the concept of dynamic memory allocation in linked lists and how malloc() and free() are used.
6. Compare singly, doubly, and circular linked lists, discussing their applications.
7. Write a C program to implement a doubly linked list with insertion and deletion at different positions.
8. What are the applications of linked lists in real-world computing?
9. Describe how deletion works in a linked list, including edge cases such as deleting the first and last nodes.
10. Implement a circular linked list in C, including insertion, deletion, and traversal operations.

MODULE 4
TREE AND GRAPH

LEARNING OUTCOMES

By the end of this Unit, students will be able to:

- Understand tree concepts, including their structure and applications.
- Learn the representation of binary trees and perform operations such as searching, insertion, and deletion.
- Implement and analyze Binary Search Tree (BST) and AVL tree algorithms for optimized searching and balancing.
- Explore graph representations (adjacency matrix, adjacency list), operations (searching, insertion, deletion), and traversal techniques (BFS, DFS) for efficient graph processing.

Unit 10: Tree concepts And Binary Tree

4.1 Tree concepts

A tree is a type of data structure that is used to represent relationships between elements in a hierarchical manner. It is made up of nodes linked through edges, where each node contains data and pointers to its children nodes. Trees are non-linear data structures (unlike linear data structures like arrays, linked lists) used for efficient searching, sorting, and hierarchical data organization.

Basic Terminology of Trees

Term	Description
Node	A single element in a tree (stores data and references).
Root	The topmost node (starting point of the tree).
Parent	A node that has child nodes.
Child	A node derived from another node (parent).
Sibling	Nodes that share the same parent.
Leaf Node	A node without children (terminal node).
Edge	Connection between two nodes.
Depth	Distance from the root to a node.
Height	Maximum depth of the tree.
Subtree	A section of a tree rooted at a particular node.

1. Properties of a Tree

1. A tree consists of N nodes and (N-1) edges.
2. There is only one root node.
3. A tree is a connected and acyclic structure (no cycles).
4. Each node can have any number of children.

2. Types of Trees

General Tree

- A tree where each node can have any number of children.

Binary Tree

- A tree where each node has at most two children (left and right).

Binary Search Tree (BST)

- A binary tree where:
- Left subtree contains smaller values.
- Right subtree contains larger values.

- Efficient for searching, insertion, and deletion (O(log n) complexity).

Balanced Tree

- A tree where the height difference between left and right subtrees is minimal.
- Example: AVL Tree, Red-Black Tree.

Heap Tree

- A complete binary tree used for priority queues.
- Min Heap: The parent is smaller than its children.
- Max Heap: The parent is greater than its children.

Trie (Prefix Tree)

- Used for searching words in dictionaries and autocomplete suggestions.

3. Representation of Trees

Trees can be represented using:

Linked List Representation

Each node contains data, left child, and right child pointers.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
root = Node(10)  # Root node
root.left = Node(5)  # Left child
root.right = Node(15)  # Right child
```

Array Representation

Trees can be stored in an array (for complete binary trees).

For a node at index i:

- Left Child → 2*i + 1
- Right Child → 2*i + 2
- Parent → (i - 1) // 2

Example for [10, 5, 15, 3, 7]:

Index	Value	Left Child	Right Child	Parent
0	10	5 (1)	15 (2)	-
1	5	3 (3)	7 (4)	10 (0)

4. Tree Traversal

Traversal is the process of visiting nodes in a tree.

Depth-First Search (DFS)

Type	Order
Preorder (NLR)	Root → Left → Right
Inorder (LNR)	Left → Root → Right
Postorder (LRN)	Left → Right → Root

```
def inorder_traversal(root):
    if root:
        inorder_traversal(root.left)
        print(root.data, end=" ")
        inorder_traversal(root.right)
```

Breadth-First Search (BFS) (Level Order Traversal)

- Visits nodes level by level (top to bottom).
- Implemented using a queue.

python

CopyEdit

```
from collections import deque
def level_order_traversal(root):
    if not root:
        return
    queue = deque([root])
    while queue:
        node = queue.popleft()
        print(node.data, end=" ")
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
```

5. Applications of Trees

Database Indexing (B-Trees, B+ Trees)

File System Hierarchies

Network Routing Algorithms

Expression Evaluation (Syntax Trees)

Artificial Intelligence (Decision Trees)
Compiler Design (Abstract Syntax Trees)

Trees are essential hierarchical data structures used for searching, sorting, and managing data. The concept of trees is an important aspect of computer science, it is data structures that sort the data into tree forms.

4.2 Binary Tree-Representation

A Binary Tree is a hierarchical data structure in which each node possesses a maximum of two offspring.

- Left Offspring
- Right Offspring

Binary trees are widely used in searching, sorting, expression evaluation, and hierarchical data representation.

Representation of Binary Tree

1. Linked List Representation (Node-Based Representation)

In this representation, each node has:

- Data (value of node).
- Pointer to left child.
- Pointer to right child.

Python Implementation (Binary Tree Node Structure)

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
# Creating a simple binary tree
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
# Tree Structure:
#       1
#      / \
#     2   3
#    / \
#   4   5
```

Notes

Advantages:

Dynamic size (grows as needed)

Efficient insertions and deletions

Disadvantages: Uses extra memory for pointers

2. Array Representation (Sequential Representation)

A binary tree can also be stored in an array where:

- Root node is at index 0.
- Left child of node at index i is at 2*i + 1.
- Right child of node at index i is at 2*i + 2.
- Parent of node at index i is at (i-1) // 2.

Example: Storing a Binary Tree in an Array

For a binary tree:

markdown

```
    1
   / \
  2   3
 / \
4   5
```

Array representation: [1, 2, 3, 4, 5]

Index	Node	Left Child Index	Right Child Index
0	1	1	2
1	2	3	4
2	3	-	-
3	4	-	-
4	5	-	-

Python Implementation (Binary Tree using an Array)

```
class BinaryTree:
    def __init__(self):
        self.tree = []
    def insert(self, data):
        self.tree.append(data)  # Insert node at the next available position
    def get_left_child(self, index):
        left_index = 2 * index + 1
        return self.tree[left_index] if left_index < len(self.tree) else None
    def get_right_child(self, index):
        right_index = 2 * index + 2
```

```
    return self.tree[right_index] if right_index < len(self.tree) else None
# Example Usage
bt = BinaryTree()
bt.insert(1)
bt.insert(2)
bt.insert(3)
bt.insert(4)
bt.insert(5)
print("Left Child of 1:", bt.get_left_child(0))  # Output: 2
print("Right Child of 1:", bt.get_right_child(0))  # Output: 3
```

Advantages:

Efficient for complete binary trees

Direct access using index

Disadvantages:

Wasted memory if the tree is sparse

Difficult insertions/deletions in the middle

3. Choosing the Right Representation

Feature	Linked List Representation	Array Representation
Memory Usage	Extra space for pointers	Wastes space in sparse trees
Insertion/Deletion	Efficient (O(1) at root)	Inefficient (O(n) shifting)
Traversal	Requires recursion/iteration	Direct access using index
Best Use Case	General trees (BST, AVL)	Complete Binary Trees

Binary trees are implemented by linked list (pointers) or array (indexing). Linked list approche is flexible solution for a dynamic tree, while array solution would be good for complete binary trees. Efficiency in memory and faster operations in applications like searching, parsing, and sorting are provided through the knowledge of both methods.

4.3 Operations: Searching, Insertion, Deletion

A Binary Tree provides fundamental capabilities such as search, insert, and delete. These operations are very prevalent in Binary Search Trees(BSTs), where elements follow sorted order:

- left subtree has values that are inferior to root.

- right subtree contains values that surpass those of root.

1. Searching in a Binary Tree

Searching: Finding one specific value from the tree. searching is also efficient in BST as it is of O(log n) complexity in balanced trees.

Algorithm for Searching in BST

1. Commence with root node.
2. If key matches root, return the node.
3. If key is smaller, perform a search in left subtree.
4. If the key is bigger, perform a search in the right subtree.
5. Continue iterating until the key is located or the tree is depleted.

Python Implementation

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
def search(root, key):
    if root is None or root.data == key:
        return root  # Found the key or reached a leaf node
    if key < root.data:
        return search(root.left, key)
    return search(root.right, key)
# Example Tree
root = Node(10)
root.left = Node(5)
root.right = Node(20)
root.left.left = Node(3)
root.left.right = Node(7)
# Search for a node
result = search(root, 7)
print("Found" if result else "Not Found")  # Output: Found
```

Time Complexity:

Best Case (Balanced Tree): O(log n)

Worst Case (Skewed Tree): O(n)

2. Insertion in a Binary Search Tree (BST)

Insertion adds a new node while maintaining the BST property.

Algorithm for Insertion

1. If tree is empty, create a new node as the root.
2. Compare value with current node:
   - If smaller, insert into the left subtree.
   - If greater, insert into the right subtree.

1. Repeat until an empty position is found.

Python Implementation

```
def insert(root, key):
    if root is None:
        return Node(key)  # Insert new node if tree is empty
    if key < root.data:
        root.left = insert(root.left, key)  # Recur for left subtree
    else:
        root.right = insert(root.right, key)  # Recur for right subtree
    return root
# Example Usage
root = Node(10)
root = insert(root, 5)
root = insert(root, 15)
root = insert(root, 3)
root = insert(root, 7)
```

Time Complexity:

Best Case (Balanced Tree): O(log n)

Worst Case (Skewed Tree): O(n)

3. Deletion in a Binary Search Tree (BST)

Deletion removes a node while maintaining the BST property.

Cases for Deletion:

1. A node devoid of descendants (Leaf Node) - Simply remove it.
2. A node possessing a solitary kid - Remove the node and connect its child to its parent.
3. A node possessing two children necessitates identifying the inorder successor (the smallest node within the right subtree), replacing the node's value with that of inorder successor, and subsequently eliminating the inorder successor.

Algorithm for Deletion

1. Search for the node to delete.
2. If it is terminal node, remove it straight.

3. If it possesses a single offspring, substitute it with that offspring.
4. If it has two children, find the inorder successor, replace the node's value, and delete the successor.

Python Implementation

```
def find_min(node):
    while node.left:
        node = node.left
    return node
def delete(root, key):
    if root is None:
        return root
    # Search for the node to delete
    if key < root.data:
        root.left = delete(root.left, key)
    elif key > root.data:
        root.right = delete(root.right, key)
    else:
        # Case 1: No child (leaf node)
        if root.left is None and root.right is None:
            return None
        # Case 2: One child
        if root.left is None:
            return root.right
        elif root.right is None:
            return root.left
        # Case 3: Two children
        temp = find_min(root.right)  # Find inorder successor
        root.data = temp.data  # Replace node value
        root.right = delete(root.right, temp.data)  # Delete successor
    return root
# Example Usage
root = Node(10)
root = insert(root, 5)
root = insert(root, 15)
root = insert(root, 3)
root = insert(root, 7)
root = delete(root, 5)  # Delete node with value 5
```

Time Complexity:

Best Case (Balanced Tree): O(log n)

Worst Case (Skewed Tree): O(n)

4. Summary of BST Operations

Operation	Best Case Complexity	Worst Case Complexity
Search	O(log n)	O(n)
Insertion	O(log n)	O(n)
Deletion	O(log n)	O(n)

BST is efficient for searching, inserting, and deleting in balanced trees.

In worst-case (skewed trees), performance degrades to O(n).

Binary trees also support important operations such as searching, insertion, and removal, which are the basis for search engines, databases, and file systems. The Binary Search Tree (BST) exhibits logarithmic time complexity for many operations and is an essential data structure in computer science.

4.4 Algorithms: Binary Search Tree and AVL

Binary Search Tree (BST)

A Binary Search Tree (BST) is binary tree in which each node adheres to the principle that :

- The left subtree holds values lesser than root.
- The right subtree comprises values that exceed those of the root.
- Duplicate values are prohibited.

Binary Search Trees facilitate efficient search, insertion, and deletion operations, achieving an average complexity of O(log n) for balanced structures.

BST Operations and Algorithms

Insertion in BST

Algorithm:

1. If tree is empty, create a new node as the root.
2. Compare the key with root:
   - If smaller, insert it into the left subtree.
   - If greater, insert it into right subtree.
3. Recursively find the correct position for new node.

Python Implementation:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
def insert(root, key):
    if root is None:
        return Node(key)
    if key < root.data:
        root.left = insert(root.left, key)
    else:
        root.right = insert(root.right, key)
    return root
# Example Usage
root = Node(10)
root = insert(root, 5)
root = insert(root, 15)
```

Notes

```
root = insert(root, 3)
root = insert(root, 7)
```

Time Complexity:

- Best Case: O(log n) (Balanced Tree)
- Worst Case: O(n) (Skewed Tree)

Searching in BST

Algorithm:

1. Commence at the root node.
2. If key corresponds to root, return the node.
3. If the key is lesser, search in left subtree.
4. If key is larger, conduct a search in right subtree.
5. Continue iterating until the key is located or the tree is depleted.Python Implementation:

```
def search(root, key):
    if root is None or root.data == key:
        return root
    if key < root.data:
        return search(root.left, key)
    return search(root.right, key)
# Example Usage
found = search(root, 7)
print("Found" if found else "Not Found")  # Output: Found
```

Time Complexity:

- Best Case: O(1)
- Worst Case: O(n) (Skewed Tree)

Deletion in BST

Algorithm:

Identify the node designated for deletion.

Instances of deletion:

- Remove the leaf node with no children.
- In the case of a single child: Substitute the node with its offspring.
- For a node with two children: Identify the inorder successor (the smallest node in right subtree), substitute the node with successor, and subsequently remove the successor.

Python Implementation:

```
def find_min(node):
    while node.left:
```

```
        node = node.left
    return node
def delete(root, key):
    if root is None:
        return root
    if key < root.data:
        root.left = delete(root.left, key)
    elif key > root.data:
        root.right = delete(root.right, key)
    else:
        if root.left is None:
            return root.right
        elif root.right is None:
            return root.left
        temp = find_min(root.right)
        root.data = temp.data
        root.right = delete(root.right, temp.data)
    return root
# Example Usage
root = delete(root, 5)
```

Time Complexity:

- Best Case: O(log n) (Balanced Tree)
- Worst Case: O(n) (Skewed Tree)

Limitations of BST

Unbalanced BST leads to O(n) operations in the worst case.

Degenerates into a linked list if values are inserted in sorted order.

Solution: Use AVL trees to maintain balance.

AVL Tree (Self-Balancing BST)

An AVL Tree is a self-balancing binary search tree in which the height disparity (balance factor) between the left and right subtrees does not exceed 1.

Balance Factor (BF) = Height of Left Subtree - Height of Right Subtree

If the absolute value of the Balance Factor exceeds 1, the tree undergoes rotation to reestablish equilibrium.

Rotation Type	When it Occurs	Action
Right Rotation (LL Rotation)	Insert in left subtree of left child	Rotate right

Left Rotation (RR Rotation)	Insert in right subtree of right child	Rotate left
Left-Right Rotation (LR Rotation)	Insert in right subtree of left child	Left Rotate first, then Right Rotate
Right-Left Rotation (RL Rotation)	Insert in left subtree of right child	Right Rotate first, then Left Rotate

Rotations in AVL Tree

Insertion in AVL Tree

1. Insert the node as in BST.
2. Update balance factors of all affected nodes.
3. If |Balance Factor| > 1, perform the appropriate rotation.

Python Implementation:

```
class AVLNode:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
        self.height = 1  # Height of the node
def get_height(node):
    return node.height if node else 0
def get_balance(node):
    return get_height(node.left) - get_height(node.right) if node else 0
def right_rotate(y):
    x = y.left
    T2 = x.right
    x.right = y
    y.left = T2
    y.height = 1 + max(get_height(y.left), get_height(y.right))
    x.height = 1 + max(get_height(x.left), get_height(x.right))
    return x
def left_rotate(x):
    y = x.right
    T2 = y.left
    y.left = x
    x.right = T2
    x.height = 1 + max(get_height(x.left), get_height(x.right))
    y.height = 1 + max(get_height(y.left), get_height(y.right))
```

```
    return y
def insert_avl(root, key):
    if root is None:
        return AVLNode(key)
    if key < root.data:
        root.left = insert_avl(root.left, key)
    else:
        root.right = insert_avl(root.right, key)
    root.height = 1 + max(get_height(root.left), get_height(root.right))
    balance = get_balance(root)
    # Perform rotations if unbalanced
    if balance > 1 and key < root.left.data:
        return right_rotate(root)
    if balance < -1 and key > root.right.data:
        return left_rotate(root)
    if balance > 1 and key > root.left.data:
        root.left = left_rotate(root.left)
        return right_rotate(root)
    if balance < -1 and key < root.right.data:
        root.right = right_rotate(root.right)
        return left_rotate(root)
    return root
```

Time Complexity:

- Insertion & Deletion: O(log n) (Always balanced)
- • BST offers fast running time for search and insert, but tends to be unbalanced.
- • After insertion/deletion, AVL Tree gets re-balanced automatically in order to keep O(log n) time take for all cases.
- • AVL trees is used in databases, search engines, and memory indexing when fast looking needed

Unit 12: Graph

4.5 Graph, Graph Representation, Operations: Searching, Insertion, Deletion,

Graph and Its Operations

A graph is a non-linear data structure that consists of:

- Vertices (Nodes) – Represent objects.
- Edges (Connections) – Represent relationships between objects.

Graphs are widely used in networking, social media, shortest path algorithms, and AI.

2. Types of Graphs

Graph Type	Description
Directed Graph (Digraph)	Edges have direction (A → B).
Undirected Graph	Edges do not have direction (A — B).
Weighted Graph	Edges have weights (cost, distance, time).
Unweighted Graph	Edges do not have weights.
Cyclic Graph	Graph contains cycles (A → B → C → A).
Acyclic Graph (DAG)	No cycles, used in scheduling tasks.

3. Graph Representation

Graphs can be represented using:

1. Adjacency Matrix

A 2D array where matrix[i][j] = 1 if there is an edge from i to j.

Example:

```
  A B C
A [0 1 0]
B [1 0 1]
C [0 1 0]
```

Python Implementation:

python

CopyEdit

```
class GraphMatrix:
    def __init__(self, vertices):
        self.vertices = vertices
        self.graph = [[0] * vertices for _ in range(vertices)]
    def add_edge(self, u, v):
        self.graph[u][v] = 1
```

```
        self.graph[v][u] = 1  # Undirected Graph
    def display(self):
        for row in self.graph:
            print(row)
# Example Usage
g = GraphMatrix(3)
g.add_edge(0, 1)
g.add_edge(1, 2)
g.display()
```

Pros: Fast edge lookup O(1).

Cons: Uses O(V²) space even for sparse graphs.

2. Adjacency List (Efficient Representation)

A list of lists where each node stores its neighbors.

Example:

A → B

B → A, C

C → B

Python Implementation:

python

CopyEdit

```
from collections import defaultdict
class GraphList:
    def __init__(self):
        self.graph = defaultdict(list)
    def add_edge(self, u, v):
        self.graph[u].append(v)
        self.graph[v].append(u)  # Undirected Graph
    def display(self):
        for key, values in self.graph.items():
            print(key, "->", values)
# Example Usage
g = GraphList()
g.add_edge("A", "B")
g.add_edge("B", "C")
g.display()
```

Pros: Uses O(V + E) space, efficient for sparse graphs.

Cons: Edge lookup is O(V) in the worst case.

4. Graph Operations

Searching (Graph Traversal)

1. Depth-First Search (DFS)

- Recursive traversal that explores as far as possible before backtracking.
- Used in: Pathfinding, cycle detection, topological sorting.

Python Implementation:

python

CopyEdit

```
def dfs(graph, node, visited=set()):
    if node not in visited:
        print(node, end=" ")
        visited.add(node)
        for neighbor in graph[node]:
            dfs(graph, neighbor, visited)
# Example Usage
graph = {"A": ["B", "C"], "B": ["D"], "C": ["E"], "D": [], "E": []}
dfs(graph, "A")  # Output: A B D C E
```

Time Complexity: O(V + E)

2. Breadth-First Search (BFS)

- Uses queue to explore neighbors level by level.
- Used in: Shortest path (Dijkstra's Algorithm), AI search algorithms.

Python Implementation:

```
from collections import deque
def bfs(graph, start):
    queue = deque([start])
    visited = set([start])
    while queue:
        node = queue.popleft()
        print(node, end=" ")
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
# Example Usage
graph = {"A": ["B", "C"], "B": ["D"], "C": ["E"], "D": [], "E": []}
bfs(graph, "A")  # Output: A B C D E
```

Time Complexity: O(V + E)

Insertion (Adding Nodes and Edges)

- Adding a vertex: Simply add a new key in adjacency list.
- Adding an edge: Update adjacency list/matrix.

Python Implementation (Adding a Node & Edge in Adjacency List):

```
def add_vertex(graph, vertex):
    if vertex not in graph:
        graph[vertex] = []
def add_edge(graph, u, v):
    graph[u].append(v)
    graph[v].append(u)
# Example Usage
graph = {}
add_vertex(graph, "A")
add_vertex(graph, "B")
add_edge(graph, "A", "B")
print(graph)  # Output: {'A': ['B'], 'B': ['A']}
```

Time Complexity: O(1) for adjacency list, $O(V^2)$ for adjacency matrix

Deletion (Removing Nodes and Edges)

- Eliminating an edge: Remove from the adjacency list.
- Eliminate a vertex by first detaching all its edges.

Python Implementation (Deleting a Node & Edge):

```
def remove_edge(graph, u, v):
    graph[u].remove(v)
    graph[v].remove(u)
def remove_vertex(graph, vertex):
    graph.pop(vertex, None)
    for neighbors in graph.values():
        if vertex in neighbors:
            neighbors.remove(vertex)
# Example Usage
graph = {"A": ["B"], "B": ["A", "C"], "C": ["B"]}
remove_edge(graph, "A", "B")
remove_vertex(graph, "C")
print(graph)  # Output: {'B': []}
```

Time Complexity: O(1) for adjacency list, O(V) for adjacency matrix

5. Applications of Graphs

- Shortest Path Algorithms – GPS Navigation (Dijkstra's Algorithm).
- Social Networks – Friend recommendations (Graph traversal).
- Computer Networks – Routing algorithms (BFS, DFS).
- AI & Game Development – Path finding (A* Algorithm).
- Scheduling Problems – Task dependencies (Topological Sorting).

Graphs provide solution to many real world complex problems, ranging from networking to path finding, until AI search. In sparse graphs, an adjacency list is efficient, while in dense graphs, an adjacency matrix is used. Working with graphs involves understanding searching (DFS, BFS), insertion, and deletion operations.

4.6 Traversing

Graph traversal refers to the systematic visitation of all nodes (vertices) and edges within a graph.

It helps in:

Searching for elements

Finding shortest paths

Detecting cycles

Solving AI and network-related problems

2. Types of Graph Traversal

Traversal Type	Description	Data Structure Used
Depth-First Search (DFS)	Explores as far as possible before backtracking	Stack (Recursion)
Breadth-First Search (BFS)	Explores neighbors level by level	Queue

3. Depth-First Search (DFS)

Concept

• Initiates at node and delves as deeply as feasible prior to retracing steps.

- Uses recursion (stack) to keep track of visited nodes.
- Used in maze solving, cycle detection, and topological sorting.

Algorithm

1. Start from a node.

2. Mark it as visited.
3. Visit adjacent unvisited nodes recursively.
4. Backtrack when no unvisited neighbors remain.

Python Implementation

```
def dfs(graph, node, visited=set()):
    if node not in visited:
        print(node, end=" ")
        visited.add(node)
        for neighbor in graph[node]:
            dfs(graph, neighbor, visited)
# Example Usage
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
dfs(graph, 'A')  # Output: A B D E F C
```

Time Complexity: O(V + E) (Vertices + Edges)
Space Complexity: O(V) (For recursive stack in worst case)

4. Breadth-First Search (BFS)

Concept

- Starts at a node and explores all its neighbors before moving deeper.
- Uses a queue to store visited nodes.
- Used in shortest path algorithms (Dijkstra's, A), network broadcasting, and AI*.

Algorithm

1. Start from a node.
2. Mark it as visited and enqueue it.
3. Dequeue a node, process it, and enqueue its unvisited neighbors.
4. Repeat until all nodes are visited.

Python Implementation

```
from collections import deque
def bfs(graph, start):
```

```
    queue = deque([start])
    visited = set([start])
    while queue:
        node = queue.popleft()
        print(node, end=" ")
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
# Example Usage
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
bfs(graph, 'A')  # Output: A B C D E F
```

Time Complexity: O(V + E)

Space Complexity: O(V)

5. DFS vs. BFS Comparison

Feature	DFS	BFS
Data Structure	Stack (Recursion)	Queue
Exploration	Deep before wide	Level-wise
Memory Usage	Less for sparse graphs	More for dense graphs
Best for	Cycle detection, Topological sorting	Shortest path, AI search

6. Applications of Graph Traversal

DFS: Maze solving, Cycle detection, Web crawling

BFS: Shortest path (Google Maps), Social media friend suggestions

both: Network routing, AI decision trees

DFS and BFS are fundamental graph traversal techniques for solving complex problems from networking, AI, and path finding. The selection of the approach is contingent upon the graph structure and the specific use case.

Multiple-Choice Questions (MCQs)

1. What is a tree in data structures?
   a) A linear data structure
   b) A hierarchical data structure
   c) A random-access data structure
   d) A sequential data structure
   (Answer: b)
2. In a binary tree, each node can have at most:
   a) One child
   b) Two children
   c) Three children
   d) Unlimited children
   (Answer: b)
3. Which of the following is a self-balancing binary search tree?
   a) AVL Tree
   b) Binary Search Tree (BST)
   c) Heap
   d) Hash
   e) Tree
   (Answer: a)
4. What is the worst-case time complexity of searching in a Binary Search Tree (BST)?
   a) O(1)
   b) O(log n)
   c) O(n)
   d) O(n log n)
   (Answer: c)
5. Which rotation is NOT used in balancing an AVL tree?
   a) Left Rotation
   b) Right Rotation
   c) Top Rotation
   d) Left-Right Rotation
   (Answer: c)
6. Which of the following is NOT a tree traversal technique?
   a) Inorder
   b) Preorder
   c) Breadth-First Search (BFS)
   d) Depth-First Search (DFS)

Notes

(Answer: c)

7. Which of the following graph representations uses a 2D matrix to store connections?
   a) Adjacency Matrix
   b) Adjacency List
   c) Incidence List
   d) Edge List

   (Answer: a)
8. In which traversal method do we visit the left subtree, then the root, and finally the right subtree?
   a) Preorder
   b) Inorder
   c) Postorder
   d) Level Order

   (Answer: b)
9. Which graph traversal algorithm uses a queue data structure?
   a) Depth-First Search (DFS)
   b) Breadth-First Search (BFS)
   c) Prim's Algorithm
   d) Kruskal's Algorithm

   (Answer: b)
10. Which of the following is NOT a graph traversal algorithm?
    a) BFS
    b) DFS
    c) Dijkstra's Algorithm
    d) Bubble Sort

    (Answer: d)

Short Questions

1. What is a binary tree, and how does it differ from a general tree?
2. Explain the inorder, preorder, and postorder tree traversal methods.
3. What is a Binary Search Tree (BST), and how is it different from a normal binary tree?
4. What are AVL trees, and why are they used?
5. What is tree balancing, and why is it important?
6. What is the difference between BFS and DFS in graph traversal?
7. Describe the adjacency matrix and adjacency list representations of graphs.

8. How does insertion work in a BST?
9. What is the primary advantage of using an AVL tree over a normal BST?
10. What are the real-world applications of graphs in computing?

Long Questions

1. Explain the concept of trees, their structure, and their applications in computing.
2. Discuss the different types of binary tree traversals with examples.
3. Describe the Binary Search Tree (BST), its insertion, deletion, and searching operations.
4. Implement a Binary Search Tree (BST) in C or Python and explain its working.
5. Explain AVL tree rotations (LL, RR, LR, RL) and how they help maintain balance.
6. Write an algorithm to perform insertion in an AVL tree and explain it with an example.
7. Compare Adjacency Matrix and Adjacency List representations in graphs.
8. Explain Depth-First Search (DFS) and Breadth-First Search (BFS) with examples.
9. Implement a graph using an adjacency list and perform DFS traversal.
10. Discuss real-world applications of trees and graphs in computer science.

MODULE 5
ALGORITHM ANALYSIS AND DESIGN

LEARNING OUTCOMES

By the end of this Unit, students will be able to:

- Understand the role of algorithms in computing, their characteristics, and the classification of problems into P and NP categories.
- Analyze algorithms based on time complexity, space complexity, and execution time to measure efficiency.
- Learn about asymptotic notations (Big-O, Omega, Theta) and their significance in evaluating algorithm performance.
- Examine algorithm design methodologies, such as Greedy, Divide and Conquer, and Dynamic Programming, accompanied by practical examples for each methodology.

Unit 13: The Role of Algorithm in Computing

5.1 The Role of Algorithm in Computing, Characteristics of algorithm, P and NP

The Role of Algorithm in Computing

What is an Algorithm? If that was a mouthful, then let me explain what it actually means in a few simple words. It takes in some data, applies logical rules to process it, and gives out the required result.

Importance of Algorithms in Computing

Efficiency – Optimizes computation time and resources.

Automation – Used in AI, automation, and machine learning.

Data Processing – Essential for sorting, searching, and managing large datasets.

Security – Used in encryption, hashing, and cybersecurity.

Artificial Intelligence – Powers decision-making in AI models.

Characteristics of a Good Algorithm

An algorithm must possess the following characteristics:

Characteristic	Description
Input	Takes zero or more inputs.
Output	Produces at least one output.
Definiteness	Each step must be well-defined.
Finiteness	Must terminate after a finite number of steps.
Correctness	Should produce the correct result for all inputs.
Effectiveness	Each step must be simple and computable.
Generality	Should be applicable to a broad class of problems.

3. P and NP Problems

Definition: Problems that can be solved in polynomial time (O(n^k)) using a deterministic algorithm.

Example: Sorting (Merge Sort – O(n log n)), Shortest Path (Dijkstra's Algorithm – $O(V^2)$).

Key Concept: If a problem belongs to P, it means it can be solved efficiently.

NP (Nondeterministic Polynomial Time) Problems

Definition: Problems where a solution can be verified in polynomial time, but finding the solution may take exponential time.

Example: Traveling Salesman Problem (TSP), Integer Factorization, Graph Coloring.

Key Concept: If a problem belongs to NP, it means it is hard to solve but easy to verify.

P vs. NP Complexity Classes

Complexity Class	Definition	Example Problems
P	Solvable in polynomial time.	Sorting, Matrix Multiplication
NP	Verifiable in polynomial time but may take exponential time to solve.	Sudoku, Hamiltonian Path
NP-Hard	As hard as NP problems but not necessarily verifiable in P time.	Halting Problem
NP-Complete	Problems that are both NP and NP-Hard.	Traveling Salesman, 3-SAT

The P vs. NP Problem

The biggest open question in computer science:

Is P = NP?

- If P = NP, then all problems in NP can be solved in polynomial time.
- If P ≠ NP, then some problems remain unsolvable in polynomial time.

Impact:

- Cryptography depends on P ≠ NP (e.g., RSA encryption).
- Optimization & AI would advance if P = NP.

Algorithms are the building blocks of computing, guaranteeing that problems can be solved efficiently. P and NP classification of problems helps in analysis of problems. The P vs. The NP problem is one of the most significant unresolved issues in computer science.

5.2 problems

In computer science, problems are classified to their complexity, solvability, and computational efficiency. The classification of problems aids in understanding if a given problem can be solved in a reasonably efficient way or if we have to resort to heuristics and approximation methods.

Classification of Problems

Problem Type	Description	Example Problems
Decision Problems	Problems with a "Yes" or "No" answer.	Is a number prime? Does a path exist in a graph?
Optimization Problems	Finding the best solution among many possible ones.	Shortest path, Traveling Salesman Problem (TSP)
Search Problems	Finding a specific solution within a large dataset.	Finding an element in a list, Graph search
Counting Problems	Counting the number of valid solutions.	Counting the number of possible paths in a grid

Computational Complexity Classes

Complexity Class	Definition	Example Problems
P (Polynomial Time)	Problems solvable in polynomial time.	Sorting, Shortest Path (Dijkstra's Algorithm)
NP (Nondeterministic Polynomial Time)	Problems verifiable in polynomial time but hard to solve.	Sudoku, Traveling Salesman Problem
NP-Hard	As hard as NP problems, but not necessarily verifiable in polynomial time.	Halting Problem, Chess Problem
NP-Complete (NPC)	Problems that are both NP and NP-Hard.	3-SAT, Hamiltonian Cycle

Key Question: Does P = NP? This remains an open problem in computer science.

Example: The Traveling Salesman Problem (TSP)

- Given: A set of cities and distances between them.
- Goal: Find shortest possible route that visits each city exactly once and returns to start.
- Complexity: NP-Hard (No known polynomial-time solution).
- Real-world Use Cases: Logistics, Circuit Design, Delivery Optimization.

Knowledge of how to classify problems is essential for designing efficient algorithms and selecting an appropriate method. All problems in P have efficient solutions, while NP problems only have heuristics/approximations for larger inputs. P vs. NP is still among the most important unsolved problems in computing.

Unit 14: Analyzing algorithms: Time and space complexity

5.3 Analyzing algorithms: Time and space complexity, Execution time

Analyzing Algorithms: Time Complexity, Space Complexity, and Execution Time

This, in turn, helps understand time and space complexity of respective algorithm with help of algorithm analysis. It enables us to assess various algorithms and select the optimal for a specific issue.

Why Analyze Algorithms?

- To measure performance and scalability.
- To compare different approaches to solve a problem.
- To optimize resource usage (memory, CPU).

Time Complexity

Time complexity is the amount of time an algorithm takes to run based on the input size (n). With Big-O notation it can be expressed.

Common Time Complexities

Complexity	Name	Example Algorithms
O(1)	Constant Time	Accessing an array element
O(log n)	Logarithmic Time	Binary Search
O(n)	Linear Time	Linear Search
O(n log n)	Linearithmic Time	Merge Sort, Quick Sort
$O(n^2)$	Quadratic Time	Bubble Sort, Selection Sort
$O(2^n)$	Exponential Time	Recursive Fibonacci
O(n!)	Factorial Time	Traveling Salesman Problem (TSP)

Example: Comparing Linear and Binary Search

Linear Search (O(n)) – Scans all elements one by one.

python

CopyEdit

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i  # Found
    return -1  # Not found
```

Binary Search (O(log n)) – Divides the list in half at each step.

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

Binary Search is much faster than Linear Search for large datasets.

Space Complexity

The space complexity is a measure of the amount of memory your algorithm will take with respect to the input size. It includes:

- Fixed part (code, constants).
- Variable part (dynamic memory allocation, recursion stack).

Common Space Complexities

Complexity	Description	Example
O(1)	Constant space	Swapping two variables
O(n)	Linear space	Storing an array of size n
$O(n^2)$	Quadratic space	Adjacency matrix for graphs
O(n log n)	Recursive algorithms	Merge Sort

Example: Iterative vs. Recursive Fibonacci

Iterative Fibonacci (O(1) Space)

```
def fibonacci_iter(n):
    a, b = 0, 1
    for _ in range(n):
        a, b = b, a + b
    return a
```

Recursive Fibonacci (O(n) Space - Due to Call Stack)

python

CopyEdit

```
def fibonacci_rec(n):
    if n <= 1:
        return n
```

```
    return fibonacci_rec(n - 1) + fibonacci_rec(n - 2)
```

Iteration is more space-efficient than recursion.

Execution Time Measurement

Execution time measures how long an algorithm takes to run in a real-world scenario.

Using Python's time Module

```
import time
def sample_function(n):
    return sum(range(n))
start_time = time.time()
sample_function(1000000)
end_time = time.time()
print("Execution Time:", end_time - start_time, "seconds")
```

Factors Affecting Execution Time:

- Hardware (CPU, RAM).
- Programming language and compiler optimizations.
- Input size and distribution.
- It analyzes the speed with which operations are performed using time complexity.
- We use space complexity for memory usage analysis.
- Execution time for real-time performance feedback.

So, depending on the requirements of the problem, how the algorithms are optimized are different in terms of time and space.

5.4 Asymptotic notations

When we say the performance(time complexity to be precise) of algorithm is expressed with n(n being input size) then we mean asymptotic notation. It is used to compare algorithms and to estimate scalability.

Why Use Asymptotic Notations?

- Ignore constant factors and lower-order terms.
- Focus on growth rate as input size increases.
- Helps in comparing algorithms efficiently.

Types of Asymptotic Notations

Notation	Meaning	Definition	Example
O (Big-O)	Upper Bound (Worst Case)	$f(n) \leq c * g(n)$ for large n	$O(n^2)$ for Bubble Sort
Ω (Big-Omega)	Lower Bound (Best Case)	$f(n) \geq c * g(n)$ for large n	$\Omega(n)$ for Linear Search

Θ (Theta)	Tight Bound (Average Case)	$c_1 * g(n) \le f(n) \le c_2 * g(n)$	Θ(n log n) for Merge Sort

Big-O Notation (Upper Bound, Worst Case)

- Defines the maximum time taken by an algorithm.
- Example: Worst-case Linear Search takes O(n) comparisons.

Example Code: Linear Search (O(n))

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i  # Found
    return -1  # Not found
```

Best for predicting the worst-case scenario.

Omega (Ω) Notation (Lower Bound, Best Case)

- Defines minimum time an algorithm will take.
- Example: Best-case Linear Search finds the element at Ω(1) (first position).

Example Code: Best Case for Linear Search (Ω(1))

```
def best_case_search(arr, target):
    if arr[0] == target:
        return 0  # Found in first position
    return -1
```

Useful for theoretical analysis but not always practical.

Theta (Θ) Notation (Tight Bound, Average Case)

- Defines the exact time complexity (both upper and lower bounds).
- Example: Merge Sort runs in Θ(n log n) in all cases.

Best notation for accurate complexity analysis.

Asymptotic Complexity Comparison

Complexity	**Name**	**Example Algorithms**
O(1)	Constant Time	Array Access
O(log n)	Logarithmic Time	Binary Search
O(n)	Linear Time	Linear Search
O(n log n)	Linearithmic Time	Merge Sort, Quick Sort
$O(n^2)$	Quadratic Time	Bubble Sort
$O(2^n)$	Exponential Time	Fibonacci (Recursive)

O(n!)	Factorial Time	Traveling Salesman Problem (TSP)

- Big-O is used for worst case analysis.
- Omega (Ω) denotes the optimal scenario.
- Theta (Θ) tightly determines execution time.

Asymptotic notations are one of the fundamental concepts in computers, Understanding that is very important for algorithms and optimising the performance.

Unit 15: Algorithm design techniques: Greedy, Divide and conquer, Dynamic programming

5.5 Algorithm design techniques: Greedy, Divide and conquer, Dynamic

1. Greedy Algorithm

A Greedy Algorithm makes decision making step by step.At every step it picks what looks like the best choice (local optimum) with the hope it would lead to a global optimum.

Key Features:

- No backtracking or re-evaluation.
- Works best for optimization problems.
- Fast and simple but does not guarantee the best solution always.

Example: Fractional Knapsack Problem

- Problem: Given n items with weights and values, maximize the total value in knapsack of capacity W, where fractions of items can be taken.
- Greedy Strategy: Pick items with highest value/weight ratio first.
- Python Implementation:

```
def fractional_knapsack(items, capacity):
    items.sort(key=lambda x: x[1] / x[0], reverse=True)  # Sort by value/weight ratio
    total_value = 0
    for weight, value in items:
        if capacity >= weight:
            capacity -= weight
            total_value += value
        else:
            total_value += value * (capacity / weight)
            break
    return total_value
# Example usage: (weight, value) pairs
items = [(10, 60), (20, 100), (30, 120)]
capacity = 50
print("Maximum value:", fractional_knapsack(items, capacity))  # Output: 240.0
```

Time Complexity: O(n log n) (Sorting dominates).

Other Greedy Algorithm Examples:

- Huffman Coding (Data Compression)
- Prim's & Kruskal's Algorithm (Minimum Spanning Tree)
- Dijkstra's Algorithm (Shortest Path for Weighted Graphs)

Limitations: May fail to find the global optimum (e.g., 0/1 Knapsack).

2. Divide and Conquer Algorithm

The Divide and Conquer approach divides the problem into subproblems, recursively solves the subproblems, then combines the results.

Key Features:

- Recursive approach
- Used in sorting, searching, and computational geometry
- Efficient for large problems

Example: Merge Sort

Problem: Sort an array using Divide and Conquer.

Steps:

1. Divide: Split the array into two halves.
2. Conquer: Recursively sort each half.
3. Combine: Merge two sorted halves.

Python Implementation:

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]
        merge_sort(left_half)
        merge_sort(right_half)
        i = j = k = 0  # Merging process
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1
        while i < len(left_half):
            arr[k] = left_half[i]
```

Notes

```
            i += 1
            k += 1
        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1
arr = [38, 27, 43, 3, 9, 82, 10]
merge_sort(arr)
print(arr)  # Output: [3, 9, 10, 27, 38, 43, 82]
```

Time Complexity: O(n log n)

Other Divide and Conquer Examples:

- Quick Sort (Pivot-based sorting, O(n log n))
- Binary Search (O(log n) Search Algorithm)
- Closest Pair of Points (Computational Geometry)

Limitations: May use extra space (Merge Sort needs O(n) extra space).

3. Dynamic Programming (DP)

Dynamic Programming A top-down (memoization) or bottom-up (tabulation) methodology that addresses intricate issues by partitioning them into overlapping subproblems and retaining the outcomes to prevent redundant calculations.

Key Features:

- Optimal substructure (Problem can be broken into subproblems).
- Overlapping subproblems (Results are reused).
- Uses extra space for memoization or tables.

Example: Fibonacci Series (Using Memoization)

- Problem: Compute Fibonacci numbers efficiently.
- DP Strategy: Store already computed results.
- Python Implementation (Memoization - Top Down)

```
def fibonacci(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2, memo)
    return memo[n]
print(fibonacci(10))  # Output: 55
```

Python Implementation (Tabulation - Bottom Up)

```
def fibonacci_tabulation(n):
    dp = [0, 1]
    for i in range(2, n + 1):
        dp.append(dp[i - 1] + dp[i - 2])
    return dp[n]
print(fibonacci_tabulation(10))  # Output: 55
```

Time Complexity:

- Naïve Recursion: $O(2^n)$
- Memoization: O(n)
- Tabulation: O(n)

Other Dynamic Programming Examples:

- 0/1 Knapsack Problem (Maximize profit in limited capacity)
- Longest Common Subsequence (DNA sequencing, text similarity)
- Matrix Chain Multiplication (Optimization problems)
- Limitations: Requires extra memory, slower for small inputs.

4. Comparison of Greedy, Divide and Conquer, and Dynamic Programming

Feature	Greedy	Divide & Conquer	Dynamic Programming
Approach	Step-by-step choice	Recursion + Merging	Memoization or Tabulation
Optimal Solution	Not always guaranteed	Always for problems with optimal substructure	Always for overlapping subproblems
Efficiency	Fast but may fail	Recursive, efficient for large data	Efficient but uses extra memory
Example	Kruskal's Algorithm, Huffman Coding	Merge Sort, Quick Sort	Fibonacci, Knapsack

- Greedy approaches are fast, but may not yield the optimal solution.
- Divide and Conquer splits problems into independent subproblems and merges results.

- Dynamic programming works the best for problems involving overlapping subproblems and optimal substructure.

Algorithm Design Techniques These are the types of techniques that one can use depending on the type of the problem & goodness of the required optimization.

5.6 programming (one example of each)

Greedy Algorithm Example: Activity Selection Problem

Problem: Given n activities with start and end times, select the maximum number of activities that do not overlap.

Greedy Strategy:

- Sort activities by finish time.
- Select activities that start after the previous selected activity ends.

Python Implementation:

```
def activity_selection(activities):
    activities.sort(key=lambda x: x[1])  # Sort by finish time
    selected = [activities[0]]  # Select first activity
    for i in range(1, len(activities)):
        if activities[i][0] >= selected[-1][1]:  # Non-overlapping condition
            selected.append(activities[i])
    return selected
# Example Usage
activities = [(1, 3), (2, 5), (4, 6), (6, 8), (5, 9)]
print("Selected Activities:", activity_selection(activities))
```

Time Complexity: O(n log n) (Sorting dominates).

2. Divide and Conquer Example: Quick Sort

Problem: Sort an array using Quick Sort (Divide and Conquer).

Steps:

1. Choose a pivot element.
2. Partition the array into two halves:
    - Left: Elements smaller than the pivot.
    - Right: Elements greater than the pivot.
3. Recursively sort both halves.

Python Implementation:

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
```

```python
    pivot = arr[len(arr) // 2]  # Choose pivot
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)
# Example Usage
arr = [10, 7, 8, 9, 1, 5]
print("Sorted Array:", quick_sort(arr))
```

Time Complexity: O(n log n) (Average case).

Dynamic Programming Example: 0/1 Knapsack Problem

Problem: Given n items with weights and values, find the maximum value that can be obtained in a knapsack of capacity W, where items cannot be divided.

DP Strategy:

- Use a 2D table to store maximum values for each weight limit.

Python Implementation:

```python
def knapsack(weights, values, capacity):
    n = len(values)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]
    for i in range(1, n + 1):
        for w in range(capacity + 1):
            if weights[i - 1] <= w:
                dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i - 1]],
dp[i - 1][w])
            else:
                dp[i][w] = dp[i - 1][w]
    return dp[n][capacity]
# Example Usage
weights = [2, 3, 4, 5]
values = [3, 4, 5, 6]
capacity = 5
print("Maximum Value:", knapsack(weights, values, capacity))    #
Output: 7
```

Time Complexity: O(n × W) (Efficient DP solution).

Greedy Algorithm (Activity Selection) – Fast but doesn't always guarantee optimality.

Divide and Conquer (Quick Sort) – Efficient and widely used in sorting.

Notes

Dynamic Programming (0/1 Knapsack) – Optimal but uses extra space.

Multiple-Choice Questions (MCQs)

1. Which of the following statements about algorithms is true?
   a) An algorithm must always have an infinite number of steps
   b) An algorithm must be unambiguous and well-defined
   c) An algorithm must be implemented in a specific programming language
   d) An algorithm does not require an input
   (Answer: b)
2. Which of the following asymptotic notations describes the worst-case time complexity of an algorithm?
   a) Big-O (O)
   b) Omega (Ω)
   c) Theta (Θ)
   d) Small-O (o)
   (Answer: a)
3. What is the time complexity of a linear search algorithm?
   a) O(1)
   b) O(n)
   c) O(log n)
   d) $O(n^2)$
   (Answer: b)
4. Which of the following problems belongs to the P category?
   a) Traveling Salesman Problem
   b) Sorting an array using Merge Sort
   c) Boolean Satisfiability Problem (SAT)
   d) Hamiltonian Cycle Problem
   (Answer: b)
5. Which of the following statements best describes NP-complete problems?
   a) They are solvable in polynomial time
   b) Their solutions can be verified in polynomial time, but solving them may require exponential time
   c) They are always unsolvable
   d) They require logarithmic space complexity
   (Answer: b)

6. Which algorithm design paradigm follows a "divide and conquer" approach?
   a) Greedy
   b) Dynamic Programming
   c) Merge Sort
   d) Backtracking
   (Answer: c)
7. Which of the following is an example of a greedy algorithm?
   a) Quick Sort
   b) Prim's Algorithm
   c) Merge Sort
   d) Binary Search
   (Answer: b)
8. Which algorithm design approach solves subproblems first and then builds up the final solution?
   a) Divide and Conquer
   b) Greedy Algorithm
   c) Dynamic Programming
   d) Brute Force
   (Answer: c)
9. What is the time complexity of the Merge Sort algorithm in the worst case?
   a) O(n)
   b) O(n log n)
   c) $O(n^2)$
   d) O(log n)
   (Answer: b)
10. Which of the following is NOT an example of a dynamic programming problem?
   a) Fibonacci sequence
   b) Knapsack problem
   c) Dijkstra's shortest path
   d) Longest common subsequence
   (Answer: c)

Short Questions

1. Define an algorithm and explain its importance in computing.

2. What is the difference between time complexity and space complexity?
3. Explain the significance of Big-O notation in algorithm analysis.
4. What is the difference between P and NP problems?
5. What is NP-complete problems, and why are they difficult to solve?
6. Compare Greedy algorithms and Dynamic Programming approaches.
7. Describe Divide and Conquer methodology and give an example.
8. Explain why Merge Sort is better than Bubble Sort in terms of complexity.
9. What is memorization, and how is it used in Dynamic Programming?
10. How does the Knapsack Problem utilize dynamic programming?

Long Questions

1. Explain the role of algorithms in computing, their characteristics, and provide real-world examples of their applications.
2. Discuss asymptotic notations (Big-O, Omega, and Theta) with examples.
3. Compare P, NP, and NP-complete problems, and explain their computational significance.
4. Describe and implement Merge Sort using the Divide and Conquer approach.
5. Explain Greedy Algorithm methodology with an example such as Kruskal's Algorithm.
6. Write a C or Python program to compute the Fibonacci sequence using recursion and dynamic programming, and compare their performance.
7. Explain Dynamic Programming, its working principle, and solve a Longest Common Subsequence (LCS) problem.
8. Compare Greedy algorithms vs. Dynamic Programming vs. Divide and Conquer, highlighting their advantages and limitations.
9. Discuss the Traveling Salesman Problem (TSP) and its classification in NP-complete problems.
10. Implement a graph algorithm using BFS (Breadth-First Search) or DFS (Depth-First Search) in Python or C.

References

Linear Data Structure (Chapter 1)

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press.
2. Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley Professional.
3. McDowell, G. L. (2016). Cracking the Coding Interview: 189 Programming Questions and Solutions (6th ed.). CareerCup.
4. Knuth, D. E. (1997). The Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd ed.). Addison-Wesley.
5. Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data Structures and Algorithms in Java (6th ed.). Wiley.

Stack, Queue and Recursion (Chapter 2)

1. Weiss, M. A. (2011). Data Structures and Algorithm Analysis in C++ (4th ed.). Pearson.
2. Karplus, W. J. (1985). The Practical Guide to Structured System Design (2nd ed.). Yourdon Press.
3. Backhouse, R. C. (1986). Program Construction and Verification. Prentice Hall.
4. Horowitz, E., & Sahni, S. (2007). Fundamentals of Data Structures in C++ (2nd ed.). W. H. Freeman.
5. Lafore, R. (1998). Data Structures and Algorithms in Java. Sams Publishing.

Linked List (Chapter 3)

1. Wirth, N. (1976). Algorithms + Data Structures = Programs. Prentice-Hall.
2. Tanenbaum, A. S., Langsam, Y., & Augenstein, M. J. (1996). Data Structures Using C. Prentice Hall.
3. Malik, D. S. (2010). C++ Programming: From Problem Analysis to Program Design (5th ed.). Course Technology.
4. Drozdek, A. (2012). Data Structures and Algorithms in C++ (4th ed.). Cengage Learning.

5. Shaffer, C. A. (2011). Data Structures and Algorithm Analysis in C++ (3rd ed.). Dover Publications.

Tree and Graph (Chapter 4)

1. Skiena, S. S. (2020). The Algorithm Design Manual (3rd ed.). Springer.
2. Tarjan, R. E. (1983). Data Structures and Network Algorithms. Society for Industrial and Applied Mathematics.
3. Kleinberg, J., & Tardos, E. (2005). Algorithm Design. Pearson Education.
4. Even, S. (2011). Graph Algorithms (2nd ed.). Cambridge University Press.
5. Nisan, N., & Schocken, S. (2005). The Elements of Computing Systems: Building a Modern Computer from First Principles. MIT Press.

Algorithm Analysis and Design (Chapter 5)

1. Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. (2006). Algorithms. McGraw-Hill.
2. Manber, U. (1989). Introduction to Algorithms: A Creative Approach. Addison-Wesley.
3. Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1983). Data Structures and Algorithms. Addison-Wesley.
4. Sipser, M. (2012). Introduction to the Theory of Computation (3rd ed.). Cengage Learning.
5. Savage, J. E. (1998). Models of Computation: Exploring the Power of Computing. Addison-Wesley.

MATS UNIVERSITY

MATS CENTER FOR OPEN & DISTANCE EDUCATION

UNIVERSITY CAMPUS : Aarang Kharora Highway, Aarang, Raipur, CG, 493 441

RAIPUR CAMPUS: MATS Tower, Pandri, Raipur, CG, 492 002

T : 0771 4078994, 95, 96, 98 M : 9109951184, 9755199381 Toll Free : 1800 123 819999

eMail : admissions@matsuniversity.ac.in Website : www.matsodl.com