Beni-Suif University
College of Computers and AI
Department of Computer Science

Lab Manual
CS211
Data Structures and Algorithms

Preparing the scientific material
A.Prof. Noha Yehia
T.A Osama Hefny T.A. Ahmed Sultan
T.A. Mohamed Gamal T.A. Sahar Hasan
T.A. Hisham Fawzy T.A. Somaya Ahmed

What We'll Build in This Course

Course Objectives

- Provide students with a solid foundation in data structures including linked lists, stacks, queues, trees, and heaps.
- Develop problem-solving skills using appropriate data structures for different scenarios.
- Introduce real-world applications of data structures in software development.
- Prepare students for technical interviews and advanced algorithm courses through hands-on C++ implementation.

Intended Learning Outcomes (ILOs)

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

1. Compare and apply searching algorithms including linear search and binary search on sorted data.
2. Implement sorting algorithms (Selection Sort, Insertion Sort, Merge Sort, Quick Sort) and analyze their performance.
3. Critically evaluate between different data structures based on time complexity, space complexity, and specific use cases.
4. Design and code linked lists with dynamic size.
5. Apply the stack data structure (LIFO principle) using both array-based (fixed size) and linked list-based (dynamic size.
6. Implement the queue data structure (FIFO principle) using both array and linked list representations for applications including job scheduling, breadth-first search (BFS)
7. Construct and traverse binary search trees (BST) for efficient searching, insertion, and deletion operations.
8. Balance AVL trees using rotations (LL, RR, LR, RL) to maintain O(log n) performance.
9. Implement heap data structure and heap sort algorithm for priority queue applications.

Weekly Breakdown

Week 1: Introduction to Data Structures & Algorithms

- **Data Structures: Definition, Advantages, Types, Operations**
- **Algorithms: Definition, Advantages, Characteristics**
- **Relationship between Data Structures and Algorithms**

Week 2: Searching and Sorting Algorithms Part 1

- **Searching Algorithms: Definition**
- **Linear Search: How it works, advantages, disadvantages, complexity (O(n))**
- **Binary Search: How it works, advantages, disadvantages, complexity (O(log n))**
- **Comparison between Linear and Binary Search**
- **Lab: Implement Linear Search and Binary Search (iterative & recursive)**

Week 3: Searching and Sorting Algorithms Part 2

- **Sorting Algorithm: Definition**
- **Selection, Insertion, Bubble, Merge and Quick sort**
- **Complexity Analysis**
- **Lab: Implement different types of sorting algorithms**

Week 4: Linked List

- **Linked List: Definition, Advantages, Disadvantages**
- **Comparison between Arrays and Linked Lists**
- **Types of Linked Lists: Singly, Doubly, Circular**
- **Operations: Insertion, Deletion, Traversal, Search**
- **Lab: Implementing Singly Linked List (all operations)**

Week 5: Stack

- **Stack: Definition, LIFO Principle**
- **Advantages and Disadvantages**
- **Operations: push(), pop(), peek(), isEmpty(), size()**
- **Types: Fixed Size Stack (Array), Dynamic Size Stack (Linked List)**
- **Applications: Undo/Redo, Function Calls, Expression Evaluation, Backtracking**

- **Lab: Implementing Stack using Array and Linked List**

Week 6: Queue

- **Queue: Definition, FIFO Principle**
- **Operations: enQueue(), deQueue(), front(), rear(), isEmpty(), isFull()**
- **Types: Simple Queue, Circular Queue, Priority Queue, Deque**
- **Applications: Job Scheduling, BFS Algorithm, Call Center, Print Spooling**
- **Lab: Implementing Queue using Array and Linked List**

Week 7: Trees & BST

- **Tree Terminology (Root, Parent, Child, Leaf)**
- **BST Property (left < root < right)**
- **BST Operations (Insert, Search)**
- **Tree Traversals (In-order, Pre-order, Post-order, Level-order)**
- **Lab: Implement BST and Traversals**

Week 8: AVL Trees

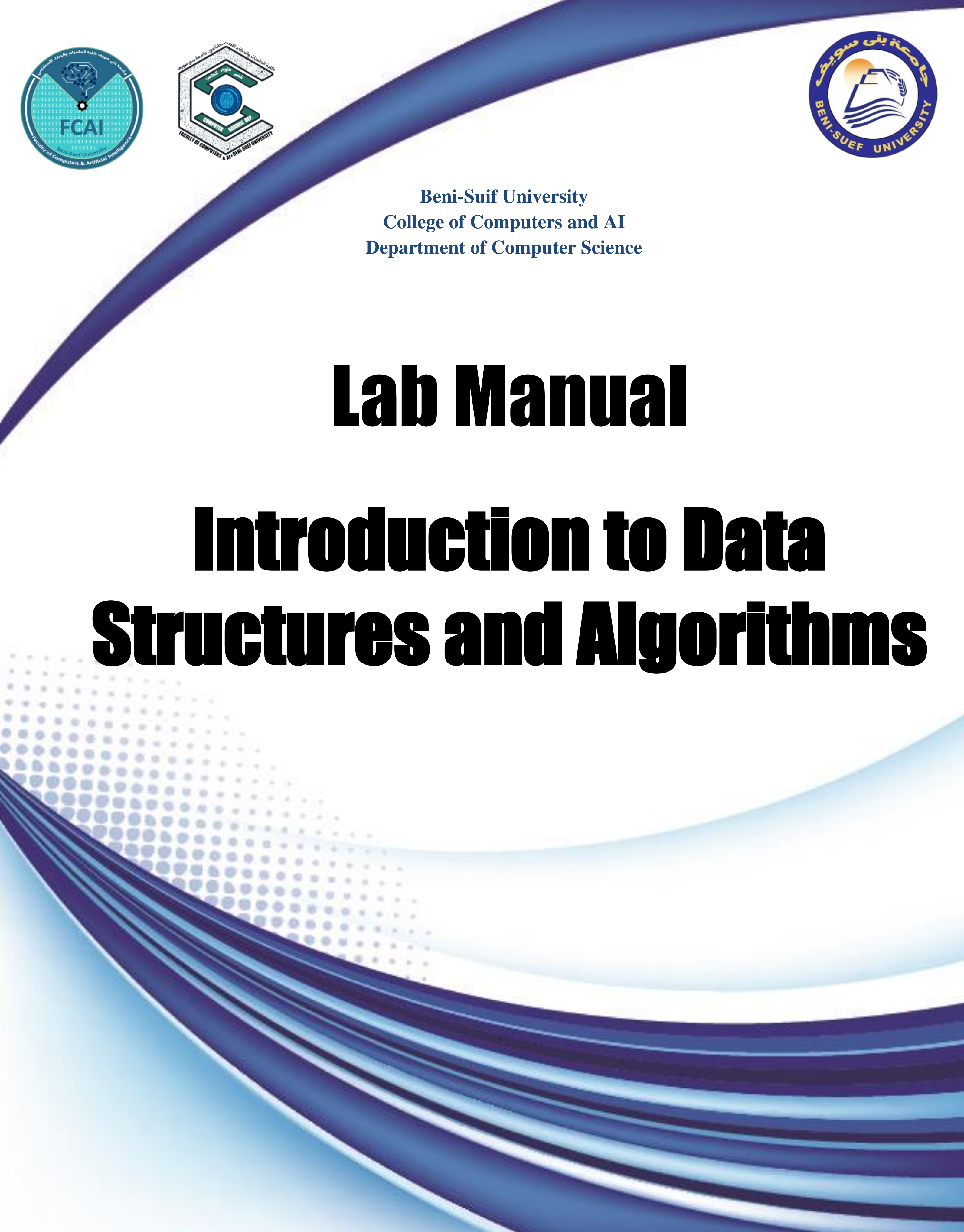
- **Balance Factor**
- **Rotations (LL, RR, LR, RL)**
- **AVL Insertion**
- **Self-balancing Property**
- **Lab: Implement AVL Tree Insertion**

Week 9: Head & Priority Heap

- **Complete Binary Tree**
- **Min-Heap and Max-Heap**
- **Heap Operations (Insert, Extract, Heapify)**
- **Heap Sort**
- **Priority Queue (STL)**
- **Lab: Implement Heap and Priority Queue**

Contents

Lab#	Description
1	Introduction to Data Structures & Algorithms
2	Searching and Sorting Algorithms Part 1
3	Searching and Sorting Algorithms Part 1
4	Linked List
5	Stack
6	Queue
7	Practical Mid Examination
8	Trees & BST
9	AVL Trees
10	Head & Priority Heap
11	Practical Final Examination



Beni-Suif University
College of Computers and AI
Department of Computer Science

Lab Manual

Introduction to Data Structures and Algorithms

Lab - 1
Introduction to Data Structures and Algorithms

What is a Data Structure?

- In simple terms, **data structure** is a way of organizing and storing data in a computer so that it can be accessed and used efficiently.
- The data structure is **not any programming language** like C, C++, java, etc. It is a set of algorithms that we can use in any programming language to structure the data in the memory.
- Think of it as a container — different containers are suited for different purposes.

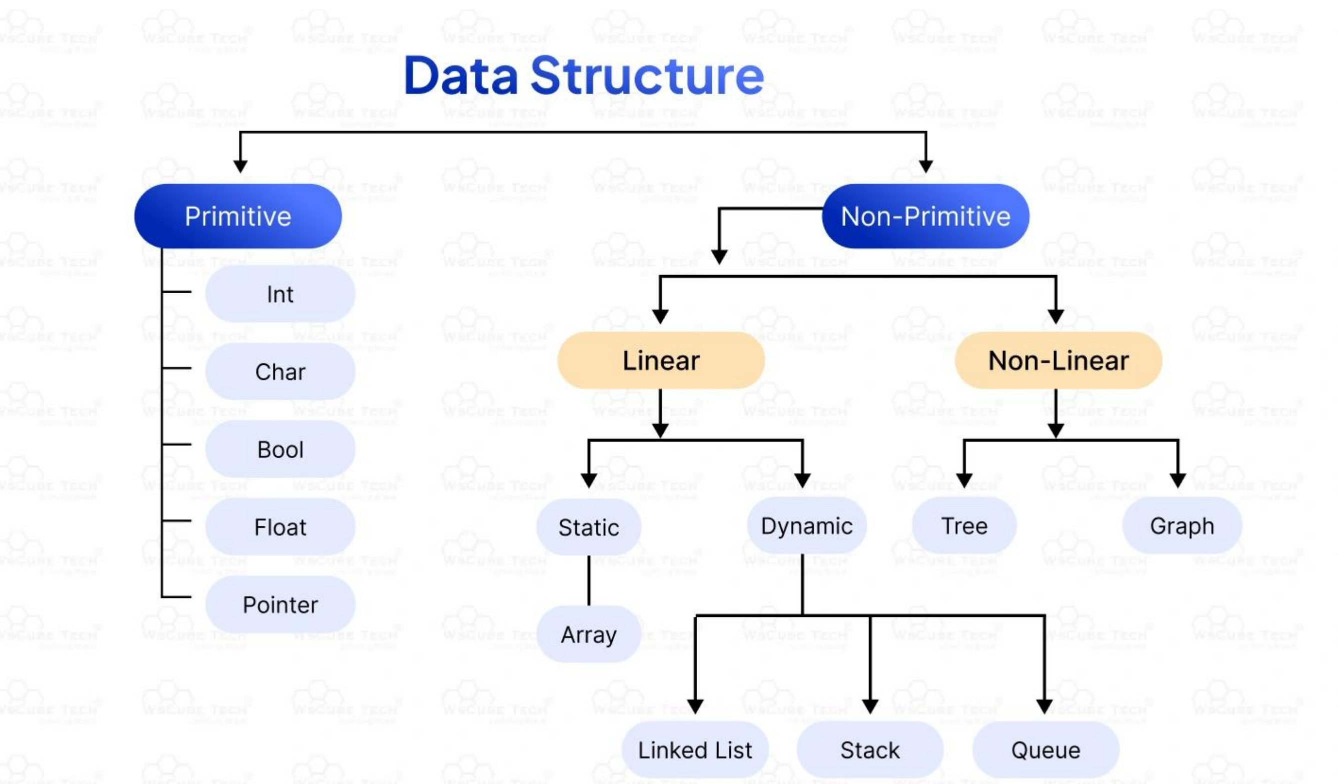
What is an Algorithms?

- **Algorithms** define the set of operations that can be performed on data to produce information. Their efficiency depends on the selected data structures.
- Think of it as a recipe — a sequence of well-defined instructions that takes an input, processes it, and produces an output.
- Data structures and algorithms are interrelated.

Why we use data structures

- The selection of good data structures will help the programmer to design more efficient programs.
- The efficiency of a program depends on two measurements:
  - Space complexity
  - Time complexity
- Time complexity can be expressed as a function of number of operations performed.

Types of Data Structures



1. **Primitive data structure:**

    - The primitive data structures are primitive data types. The int, char, float, double, and pointer are the primitive data structures that can **hold a single value**.

2. **Non-Primitive Data structure:**

    - **Linear Data Structure:** the arrangement of data in a **sequential manner** is known as a linear data structure. The data structures used for this purpose are Arrays, Linked list, Stacks, and Queues. In these data structures, one element is connected to only one another element in a linear form.

    - **Non-linear data structure:** When one element is connected to the 'n' number of elements known as a non-linear data structure. The best example is trees and graphs. In this case, the elements are arranged in **a random manner**.

Data structures can also be classified as:

- **Static data structure:** It is a type of data structure where the size is allocated at the compile time. Therefore, the maximum size is fixed.

- **Dynamic data structure:** It is a type of data structure where the size is allocated at the run time. Therefore, the maximum size is flexible.

Major Operations

- **Searching:** We can search for any element in a data structure.

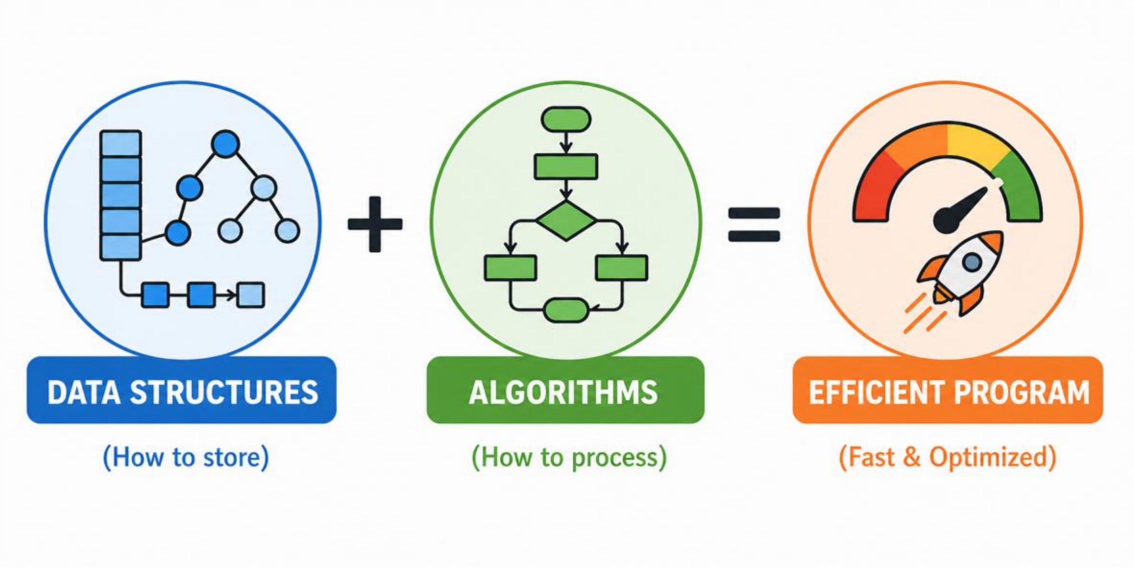
- **Sorting:** We can sort the elements of a data structure either in an ascending or descending order.

- **Insertion:** We can also insert the new element in a data structure.

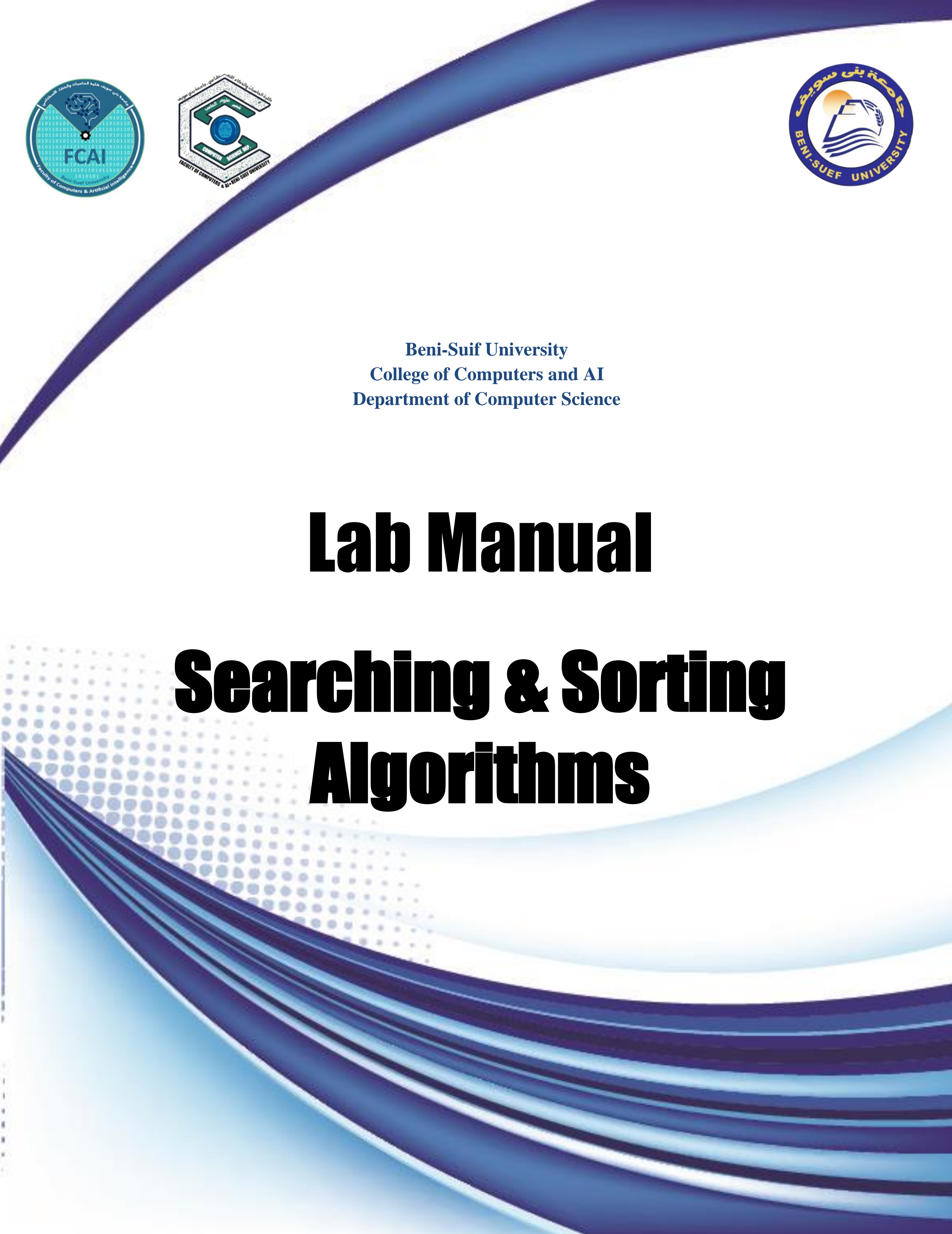
- **Updating:** We can also update the element, i.e., we can replace the element with another element.

- **Deletion:** We can also perform the delete operation to remove the element from the data structure.

The Relationship between data structures and algorithms :



- **Data structures and algorithms** work together as a team. You cannot have one without the other in any real program.
  - **Data structures** organize and store data in memory (the "what" and "where")
  - **Algorithms** process that data step by step (the "how")
- A program becomes efficient only when both are chosen wisely. A great algorithm running on poorly organized data will still be slow. Likewise, perfectly organized data with a bad algorithm will also perform poorly.
- **Key Point:** The right combination of data structure + algorithm = faster execution, less memory usage, and ability to handle larger amounts of data.



Beni-Suif University
College of Computers and AI
Department of Computer Science

Lab Manual

Searching & Sorting Algorithms

Lab - 2
Searching & Sorting Algorithms

1. An Overview?

This lab introduces two fundamental concepts in computer science: **searching algorithms** (finding specific data) and **sorting algorithms** (arranging data in order). By the end of this session, you will understand how different searching and sorting techniques work, their efficiency differences, and when to use each one. You will also implement these algorithms in C++.

Learning Objectives

After completing this lab, you should be able to:

- **Define** what searching and sorting algorithms are and why they matter.
- **Implement** Linear Search on both sorted and unsorted data.
- **Implement** Binary Search on sorted arrays (iterative and recursive).
- **Implement** basic sorting algorithms: Selection Sort, Insertion Sort, Bubble Sort
- **Implement** advanced sorting algorithms: Merge Sort and Quick Sort.
- **Compare** the time complexity of different searching and sorting algorithms.
- **Choose** the appropriate algorithm for a given scenario based on data size and order.
- Analyze the advantages and disadvantages of each algorithm.

2. Theory

What are Searching Algorithms?

Searching is the process of finding a specific element (called the key) in a collection of data. Think of it like looking for a specific book in a library.

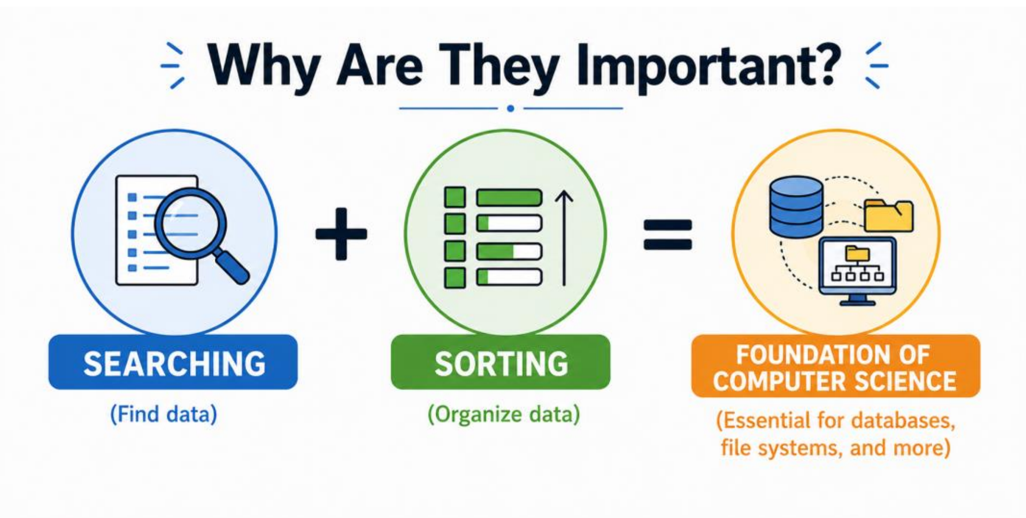
Simple definition: Searching algorithms tell the computer how to find what you're looking for.

What are Sorting Algorithms?

Sorting is the process of arranging elements in a specific order (ascending or descending). Think of it like arranging books on a shelf by their titles.

Simple definition: Sorting algorithms tell the computer how to organize data.

Why Are They Important?

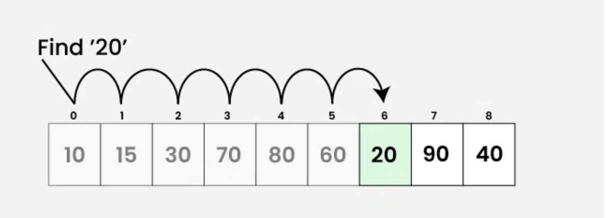


Part 1: Searching Algorithms

1.1 Linear Search

Definition:
Linear Searching Algorithms is the simplest searching algorithm that is used to find an element in the given collection. It simply compares the element to find with each element in the collection one by one till the matching element is found or there are no elements left to compare.

How It Works:



Step	Action
1	Start from the first element (index = 0)
2	Compare current element with the key
3	If equal → return current index
4	If not equal → move to next element
5	Repeat steps 2-4 until element found or end of array
6	If end reached without finding → return -1 (not found)

Complexity Analysis:

Case	Time Complexity	Explanation
Best	O(1)	Element found at first position
Average	O(n)	Element found in the middle
Worst	O(n)	Element at end or not present
Space	O(1)	No extra memory needed

Advantages:

1. Works on unsorted data
2. Simple to understand and implement
3. No preprocessing required
4. Good for small datasets

Disadvantages:

1. Slow for large datasets (O(n) time)
2. Not efficient for repeated searches

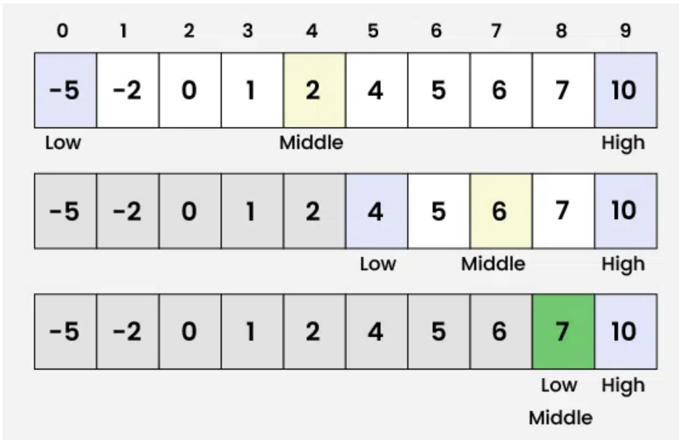
1.2 Binary Search

Definition:

Binary Searching Algorithms is an efficient searching algorithm used to find an element in a sorted array.

Note:- The array must be sorted (ascending or descending).

How It Works:



Step	Action
1	Set low = 0, high = n-1
2	While low ≤ high:
3	mid = low + (high - low) / 2
4	If arr[mid] == key → return mid
5	If key < arr[mid] → high = mid - 1 (search left)
6	If key > arr[mid] → low = mid + 1 (search right)
7	Return -1 (element not found)

Complexity Analysis:

Case	Time Complexity	Explanation
Best	O(1)	Element found at middle
Average	O(log n)	Search space halves each time
Worst	O(log n)	Element at end or not present
Space (Iterative)	O(1)	No extra memory needed
Space (Recursive)	O(log n)	Call stack space

Advantages:

1. Fast for large datasets (O(log n) time)
2. Reduces search space by half each step
3. Efficient for repeated searches

Disadvantages:

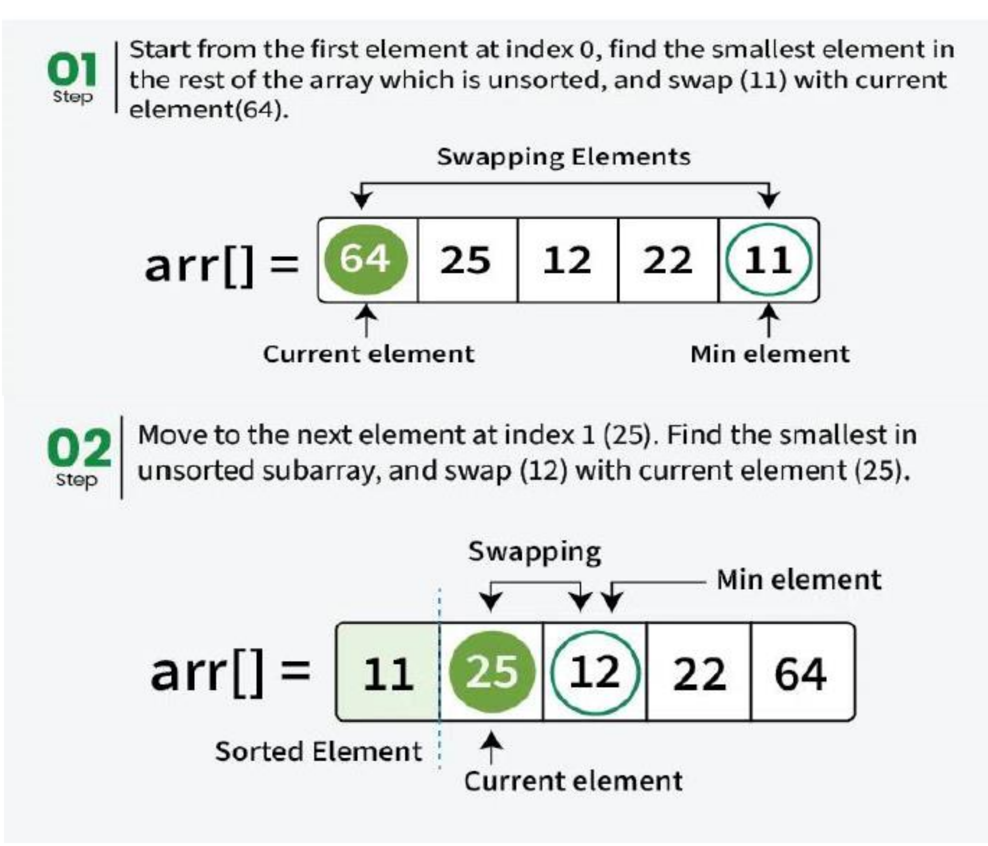
1. Requires sorted data (preprocessing needed)
2. More complex than linear search
3. Sorting adds extra cost if data is not already sorted

Part 2: Sorting Algorithms

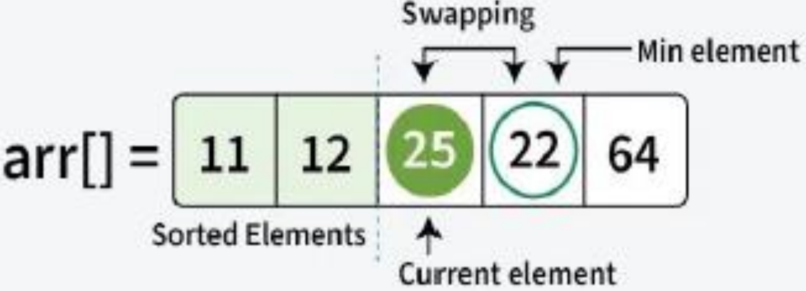
2.1 Selection Sort

Definition:
is a comparison-based sorting algorithm. It sorts an array by repeatedly selecting the smallest (or largest) element from the unsorted portion and swapping it with the first unsorted element. This process continues until the entire array is sorted.

How It Works:



Move to element at index 2 (25). Find the minimum element from unsorted subarray, Swap (22) with current element (25).



Selection Sort Algorithm

04
Step

Move to element at index 3 (25), find the minimum from unsorted subarray and swap (25) with current element (25).

Min element
arr[] = 11 12 22 25 64
Sorted Elements
Current element

Selection Sort Algorithm

05
Step

Move to element at index 4 (64), find the minimum from unsorted subarray and swap (64) with current element (64).

Min element
arr[] = 11 12 22 25 64
Sorted Elements
Current element

Step	Action
1	Find the smallest element in the array
2	Swap it with the first element
3	Find the smallest among remaining elements
4	Swap it with the second element
5	Repeat until the entire array is sorted

Complexity Analysis:

- **Time Complexity:** O(n2) ,as there are two nested loops:
  - One loop to select an element of Array one by one = O(n)
  - Another loop to compare that element with every other Array element = O(n)
  - Therefore overall complexity = O(n) * O(n) = O(n*n) = O(n2)
- **Auxiliary Space:** O(1) as the only extra memory used is for temporary variables.

Advantages:

1. Easy to understand and implement, making it ideal for teaching basic sorting concepts.
2. Requires only a constant O(1) extra memory space.
3. It requires less number of swaps (or memory writes) compared to many other standard algorithms. Only cycle sort beats it in terms of memory writes. Therefore it can be simple algorithm choice when memory writes are costly.

Disadvantages:

1. Selection sort has a time complexity of O(n2) makes it slower compared to algorithms like Quick Sort or Merge Sort.
2. Does not maintain the relative order of equal elements which means it is not stable.

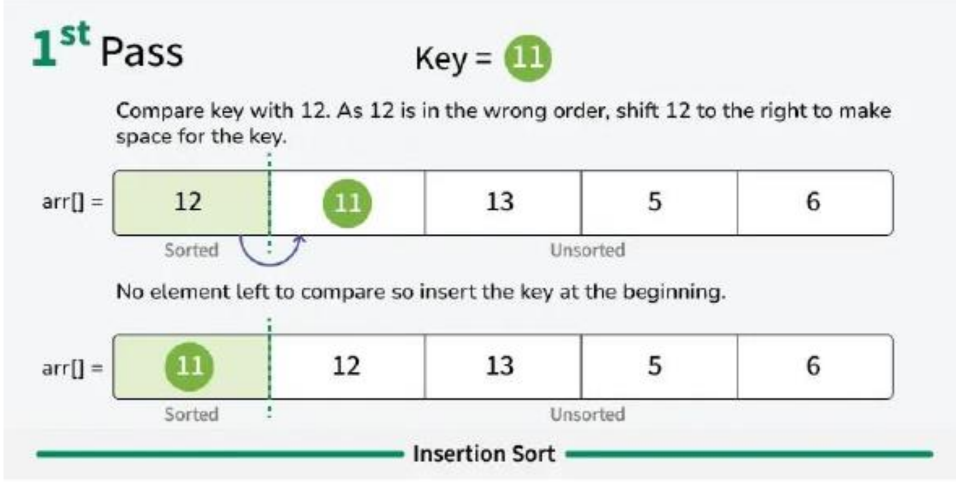
Applications:

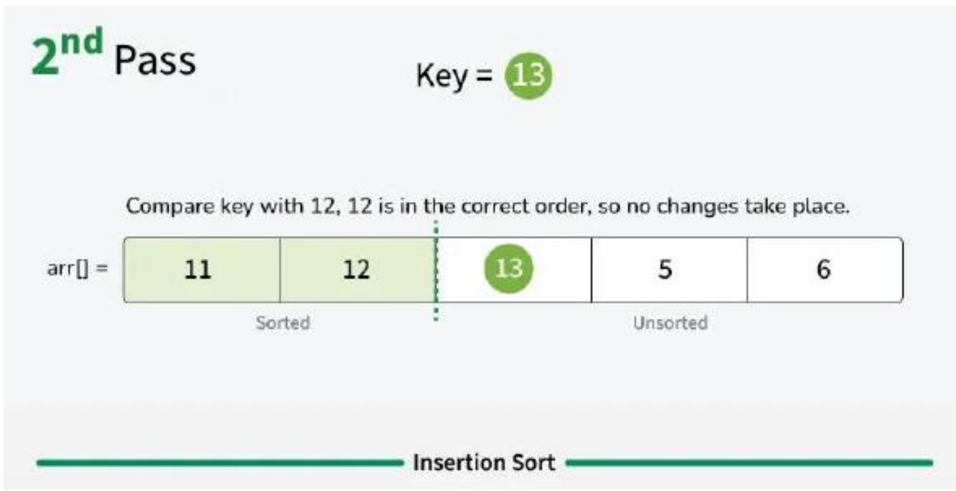
1. Perfect for teaching fundamental sorting mechanisms and algorithm design.
2. Suitable for small lists where the overhead of more complex algorithms isn't justified and memory writing is costly as it requires less memory writes compared to other standard sorting algorithms.
3. Heap Sort algorithm is based on Selection Sort.

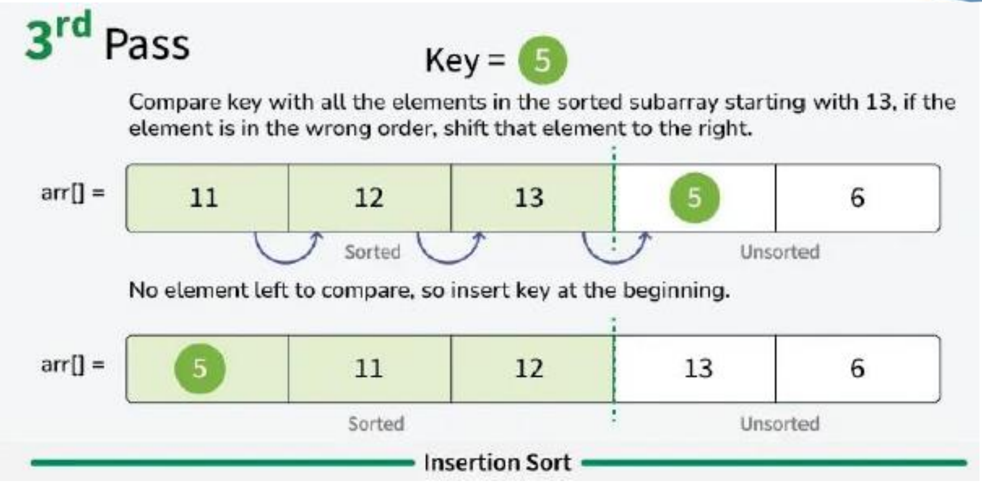
2.2 Insertion Sort

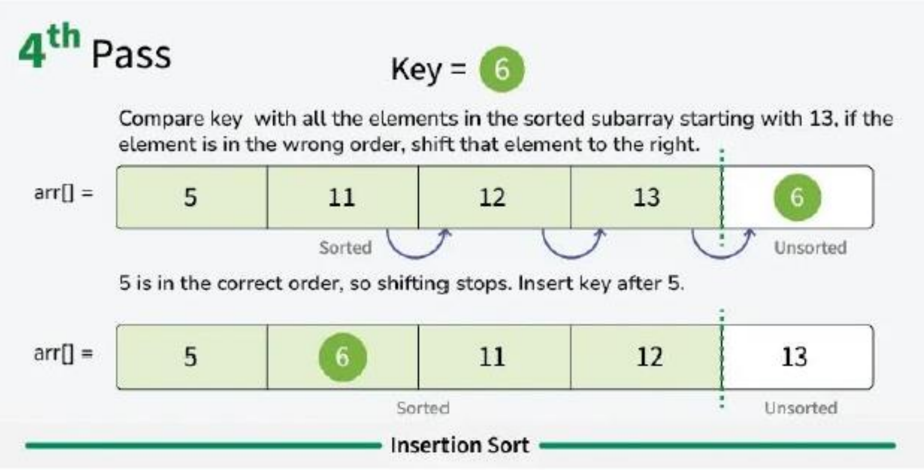
Definition:
Is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is like sorting playing cards in your hands. You split the cards into two groups: the sorted cards and the unsorted cards. Then, you pick a card from the unsorted group and put it in the right place in the sorted group.

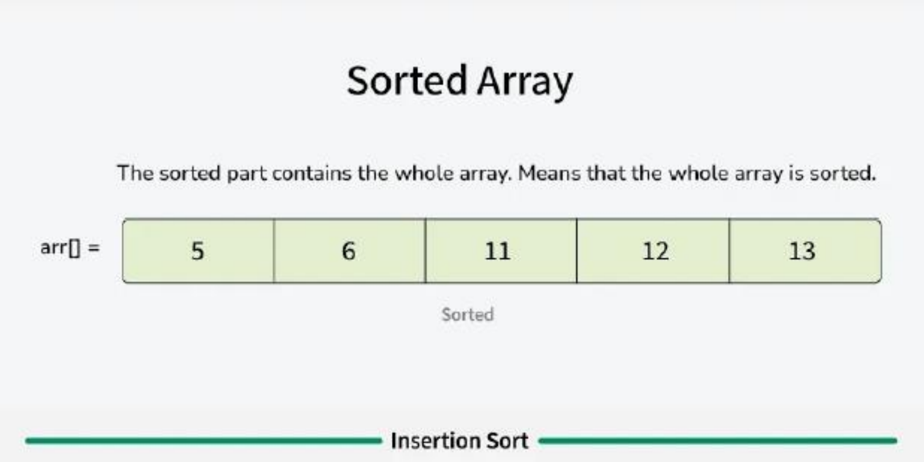
How It Works:











Step	Action
1	Assume first element is already sorted
2	Pick the next element (key)
3	Compare key with elements in sorted portion
4	Shift larger elements one position to the right
5	Insert key into its correct position
6	Repeat until all elements are sorted

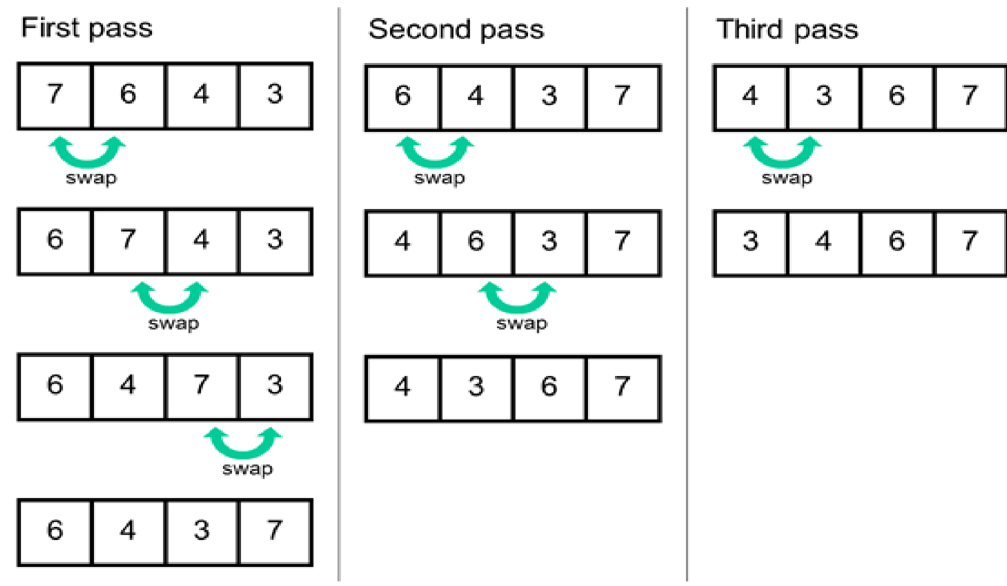
Complexity Analysis:

Case	Time Complexity	Explanation
Best	O(n)	Array already sorted
Average	$O(n^2)$	Random order
Worst	$O(n^2)$	Array reverse sorted
Space	O(1)	In-place sorting

2.3 Bubble Sort

Definition:
is the simplest sorting algorithm that works by **repeatedly** swapping the **adjacent** elements if they are in the wrong order.

How It Works:



Step	Action
1	Compare first and second elements
2	If first > second, swap them
3	Move to next pair (second and third)
4	Repeat until end of array
5	Largest element is now at the end
6	Repeat steps 1-5 for remaining elements
7	Stop if no swaps were made in a pass

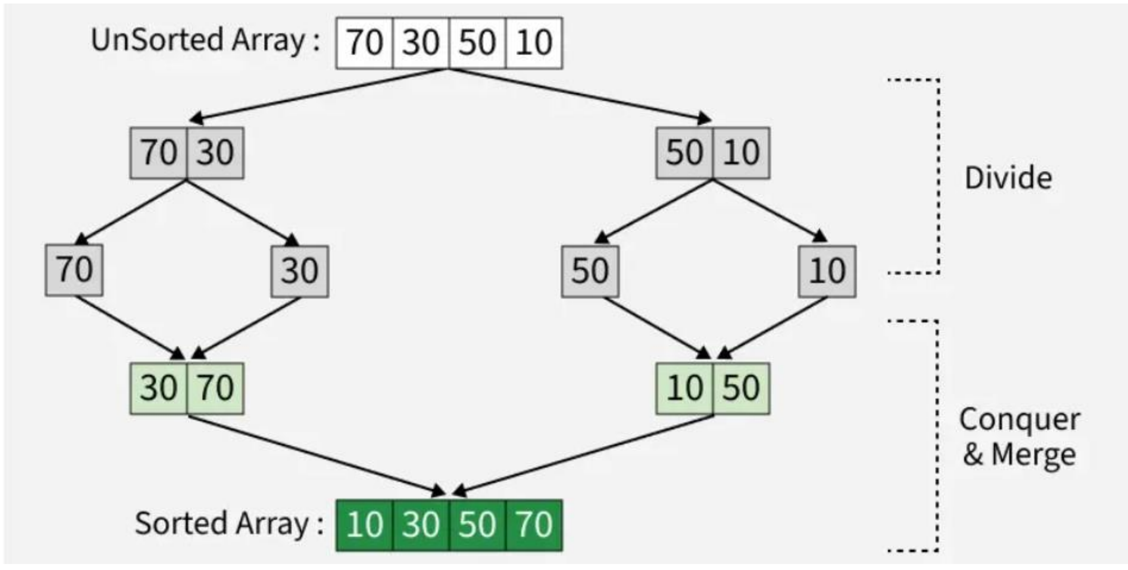
Complexity Analysis:

Case	Time Complexity	Explanation
Best	O(n)	Array already sorted
Average	$O(n^2)$	Random order
Worst	$O(n^2)$	Array reverse sorted
Space	O(1)	In-place sorting

2.4 Merge Sort

Definition:
A divide-and-conquer sorting algorithm that divides the array into two halves, recursively sorts each half, and then merges the two sorted halves into one sorted array.

How It Works:



1. Dividing the array into two halves

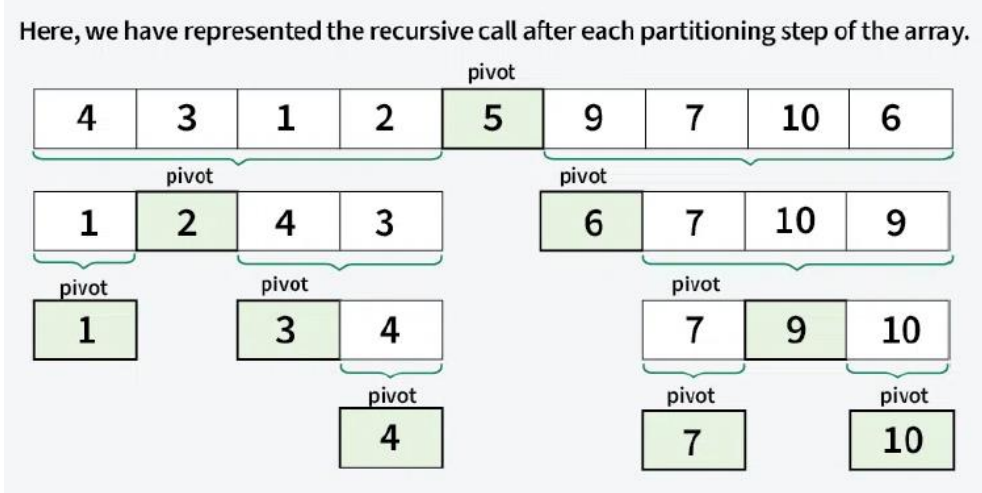
2. Recursively sorting each half

3. Merging the two sorted halves into one sorted array

2.5 Quick Sort

Definition:
Quick Sort is a sorting algorithm based on the Divide and Conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

How It Works:



There are mainly three steps in the algorithm:

1. Choose a Pivot: Select an element from the array as the pivot. The choice of pivot can vary (e.g., first element, last element, random element, or median).

2. Partition the Array: Re arrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on its left, and all elements greater than the pivot will be on its right.

3. Recursively Call: Recursively apply the same process to the two partitioned sub-arrays.

4. Base Case: The recursion stops when there is only one element left in the sub-array, as a single element is already sorted.

3. Lab

Exercise 1: Linear Search Implementation

```
#include <iostream>
#include<vector>
using namespace std;
int linearSearch(vector<int>& v, int key, int i) {
    // If we have traversed the entire vector
    // and the key is not found
    if (i == v.size()) {
        return -1;
    }
    // If the current element matches the key
    if (v[i] == key) {
        return i;
    }
    // Recursive call to check the next element
    return linearSearch(v, key, i + 1);
}

int main() {
    vector<int> v = { 1, 2, 3, 4, 5, 8, 9, 11 };
    // Value to search
    int key = 8;
    // Searching the key in the vector v
    int i = linearSearch(v, key, 0);
    // Checking if element is found or not
    if (i != -1) {
        cout << key << " Found at Position: " << i + 1;
    }
    else {
        cout << key << " NOT found.";
    }
    return 0;
}
```

Exercise 2: Binary Search Implementation

```
#include <iostream>
using namespace std;
int binarySearch(int arr[], int n, int x)
{
    int low = 0;
    int high = n - 1;
    while (low <= high)
    {
        int mid = (high + low) / 2;
        if (arr[mid] == x)
            return mid; // Element found
        else if (x < arr[mid])
            high = mid - 1; // Search left half
        else
            low = mid + 1; // Search right half
    }
    return -1; // Element not found
}
int binarySearch(int arr[], int low, int high, int x) {
    if (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] == x)
            return mid;

        if (x < arr[mid])
            return binarySearch(arr, low, mid - 1, x);

        return binarySearch(arr, mid + 1, high, x);
    }
    return -1;
}
int main()
{
    int arr[] = { 2, 5, 8, 12, 16, 23, 38 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 16;
    int result = binarySearch(arr, n, x);
    if (result != -1)
        cout << "Element found at index: " << result << endl;
    else
        cout << "Element not found" << endl;
    return 0;
```

Exercise 3: Selection Sort Implementation

```cpp
#include <iostream>
using namespace std;
void selectionSort(int arr[], int n) {
for (int i = 0; i < n - 1; i++) {
    int minIndex = i;
    // Find the index of the minimum element in the unsorted part
    for (int j = i + 1; j < n; j++) {
        if (arr[j] < arr[minIndex]) {
        minIndex = j;
        }
    }
    // Swap if a smaller element was found
    if (minIndex != i) {
    swap(arr[i], arr[minIndex]);
    }
    }
}

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

int main() {
    int arr[] = { 64, 25, 12, 22, 11 };
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Original array: ";
    printArray(arr, n);
    selectionSort(arr, n);
    cout << "Sorted array: ";
    printArray(arr, n);
    return 0;
}
```

Exercise 4: Insertion Sort Implementation

```cpp
#include <iostream>
using namespace std;
/* Function to sort array using insertion sort */
void insertionSort(int arr[], int n)
{
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;
        /* Move elements of arr[0..i-1], that are
        greater than key, to one position ahead
        of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
/* A utility function to print array of size n */
void printArray(int arr[], int n)
{
for (int i = 0; i < n; ++i)
cout << arr[i] << " ";
cout << endl;
}
// Driver method
int main()
{
int arr[] = { 12, 11, 13, 5, 6 };
int n = sizeof(arr) / sizeof(arr[0]);
insertionSort(arr, n);
printArray(arr, n);
return 0;
}
```

Exercise 5: Bubble Sort Implementation

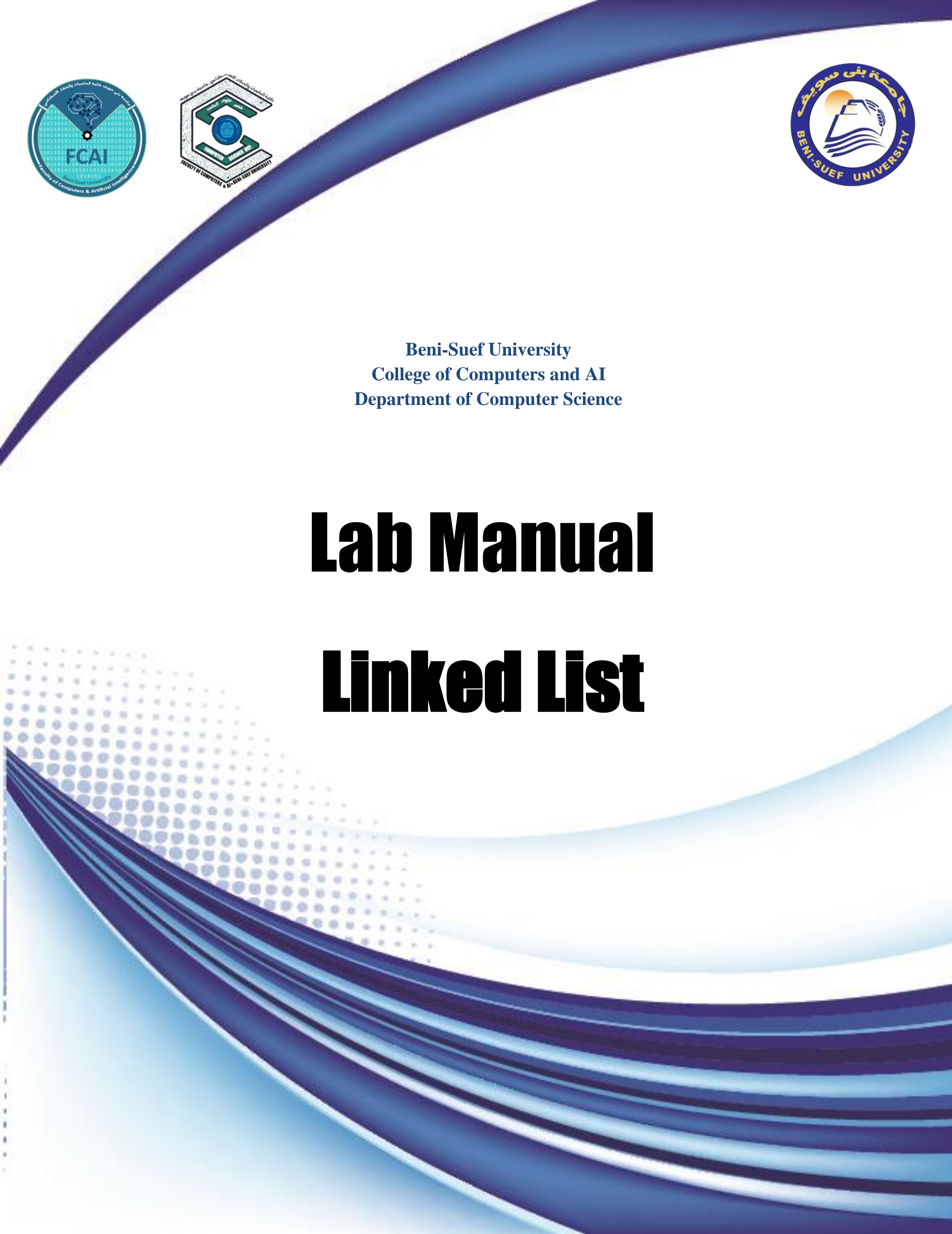
```cpp
#include <iostream>
#include<vector>
using namespace std;
// An optimized version of Bubble Sort
void bubbleSort(vector<int>& arr)
{
    int n = arr.size();
    bool swapped;
    for (int i = 0; i < n - 1; i++)
    {
        swapped = false;
        for (int j = 0; j < n - i - 1; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                swap(arr[j], arr[j + 1]); swapped = true;
            }
        }
        // If no two elements were swapped, then break
        if (!swapped) break;
    }
}

// Function to print a vector
void printVector(const vector<int>& arr)
{
    for (int num : arr)
        cout << " " << num;
}

int main()
{
    vector<int> arr = { 64, 34, 25, 12, 22, 11, 90 };

    bubbleSort(arr);
    cout << "Sorted array: \n";

    printVector(arr);
    return 0;

}
```

Exercise 6: Merge Sort Implementation

```cpp
#include <iostream>
using namespace std;
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int L[n1], R[n2];
    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];
    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j])
            arr[k++] = L[i++];
        else
            arr[k++] = R[j++];
    }

    while (i < n1)
        arr[k++] = L[i++];

    while (j < n2)
        arr[k++] = R[j++];
}

void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
int main() {
    int arr[] = {38, 27, 43, 3, 9, 82, 10};
    int n = sizeof(arr) / sizeof(arr[0]);
    mergeSort(arr, 0, n - 1);
    cout << "Sorted array: ";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";

    return 0;
}
```

Exercise 5: Quick Sort Implementation

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

int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }

    swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quickSort(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int main() {
    vector<int> arr = {10, 7, 8, 9, 1, 5};
    int n = arr.size();

    quickSort(arr, 0, n - 1);

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

    return 0;
}
```



Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

Linked List

Lab - 3
Linked List

1. An overview

This lab introduces one of the most fundamental **dynamic data structures**: the Linked List. By the end of this session, you will understand what makes linked lists different from arrays, how nodes are connected using pointers, and the basic operations you can perform on a singly linked list. You will also implement a complete linked list from scratch in C++.

Learning Objectives

After completing this lab, you should be able to:

- Differentiate between arrays and linked lists.
- Identify the three types of linked lists: Singly, Doubly, and Circular.

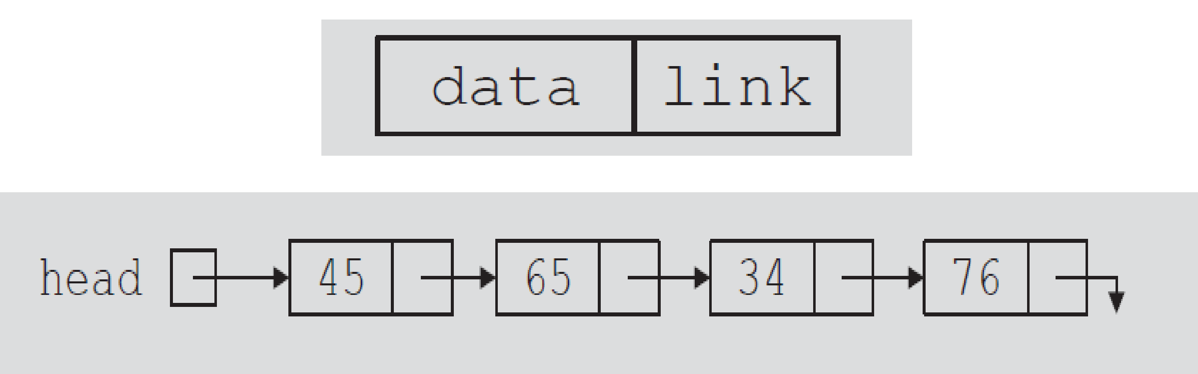
    1. Implement a complete singly linked list with operations:
    2. Insertion at beginning, end, and specific position
    3. Deletion by value and by position
    4. Searching for an element
    5. Traversal and display
    6. Finding length of the list

- Compare the time complexity of linked list operations vs array operations.

2. Theory

What is a Linked List?
A linked list is a collection of components, called nodes. Every node (except the last node) contains the address of the next node. Thus, every node in a linked list has two components: one to store the relevant information (that is, data) and one to store the address, called the link, of the next node in the list. The address of the first node in the list is stored in a separate location, called the head or first.

Simple definition: A linked list is a chain of nodes where each node points to the next node in the sequence.



Advantages of arrays

- In an array, accessing an element is very easy by using the index number.
- The search process can be applied to an array easily.
- 2D Array is used to represent matrices.
- For any reason, a user wishes to store multiple values of similar type then the Array can be used and utilized efficiently.
- For example, in a system, if we maintain a sorted list of IDs in an array id[]. id[] = [1000, 1010, 1050, 2000, 2040].

Disadvantages of arrays

- Array size is fixed.
- Array is homogeneous: The array is homogeneous, i.e., only one type of value can be store in the array.
- Array is Contiguous blocks of memory.
- Insertion and deletion are not easy in Array.

Unlike arrays, the linked list does not need to store data elements in contiguous memory regions or blocks.

Advantages over arrays

1) Dynamic size
2) Ease of insertion/deletion

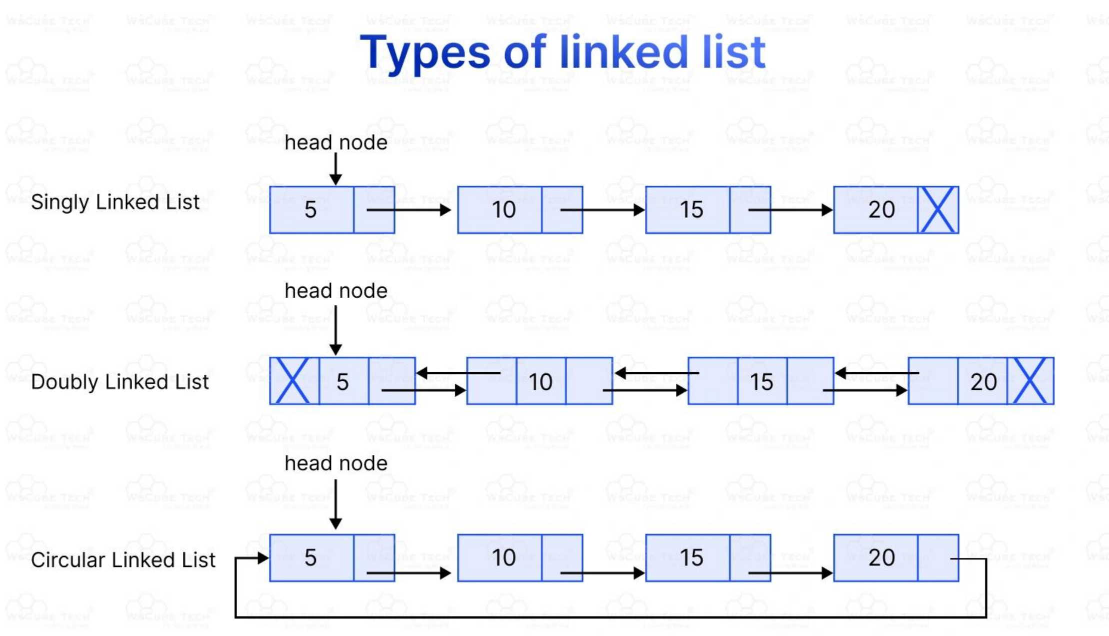
Drawbacks:

1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists efficiently with its default implementation.
2) Extra memory space for a pointer is required with each element of the list.
3) Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

Types of Linked List

Following are the various types of linked list.

- Simple Linked List − Item navigation is forward only.
- Doubly Linked List − Items can be navigated forward and backward.
- Circular Linked List − Last item contains link of the first element as next and the first element has a link to the last element as previous.



Operations on Singly Linked List:

- Traversal
- Searching
- Length
- Insertion:
    - Insert at the beginning
    - Insert at the end
- Insert at a specific position
- Deletion:
    - Delete from the beginning
    - Delete from the end
    - Delete a specific node

3. Lab

Exercise 1: Linked List Implementation

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

// Node structure
class Node {
public:
    int value;
    Node* next;

    Node(int val) {
        value = val;
        next = nullptr;
    }
};

// LinkedList class
class LinkedList {
private:
    Node* head;

public:
    LinkedList() {
        head = nullptr;
    }

    // Insert at beginning
    void insertAtBeginning(int val) {
        Node* newNode = new Node(val);
        newNode->next = head;
        head = newNode;
    }

    // Insert at end
    void insertAtEnd(int val) {
        Node* newNode = new Node(val);

        if (head == nullptr) {
            head = newNode;
            return;
        }
```

```
    Node* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }

    temp->next = newNode;
}

// Insert at specific position (1-based index)
void insertAtPosition(int pos, int val) {
    if (pos == 1) {
        insertAtBeginning(val);
        return;
    }

    Node* newNode = new Node(val);
    Node* temp = head;

    for (int i = 1; temp != nullptr && i < pos - 1; i++) {
        temp = temp->next;
    }

    if (temp == nullptr) {
        cout << "Invalid position!\n";
        return;
    }

    newNode->next = temp->next;
    temp->next = newNode;
}

// Search for a value
Node* search(int val) {
    Node* temp = head;

    while (temp != nullptr) {
        if (temp->value == val) {
            return temp;
        }
        temp = temp->next;
    }

    return nullptr;
}
```

```cpp
    // Display list
    void display() {
        Node* temp = head;

        while (temp != nullptr) {
            cout << temp->value << " -> ";
            temp = temp->next;
        }

        cout << "NULL" << endl;
    }

    // Delete node by value
    void deleteByValue(int val) {
        if (!head) {
            cout << "List is empty, nothing to delete.\n";
            return;
        }

        // If head needs deletion
        if (head->value == val) {
            Node* temp = head;
            head = head->next;
            delete temp;
            return;
        }

        Node* temp = head;

        while (temp->next && temp->next->value != val) {
            temp = temp->next;
        }

        if (!temp->next) {
            cout << "Value not found in the list.\n";
            return;
        }

        Node* toDelete = temp->next;
        temp->next = temp->next->next;
        delete toDelete;
    }
};
```

```cpp
// Main function
int main() {
    LinkedList myList;

    // Inserting elements
    myList.insertAtEnd(10);
    myList.insertAtEnd(20);
    myList.insertAtEnd(30);

    myList.insertAtBeginning(5);
    myList.insertAtPosition(3, 15); // Insert 15 at position 3

    // Display list
    cout << "Linked List: ";
    myList.display();

    // Search
    int searchValue = 20;
    Node* result = myList.search(searchValue);

    if (result != nullptr) {
        cout << "Value " << searchValue << " found in the list." << endl;
    } else {
        cout << "Value " << searchValue << " not found in the list." << endl;
    }

    return 0;
}
```

Exercise 2: real-life implementation of a linked list could be a task management system, where tasks are stored in a linked list with operations like adding, deleting, searching, updating, and displaying tasks.

Task Management System using Linked List in C++

Operations:

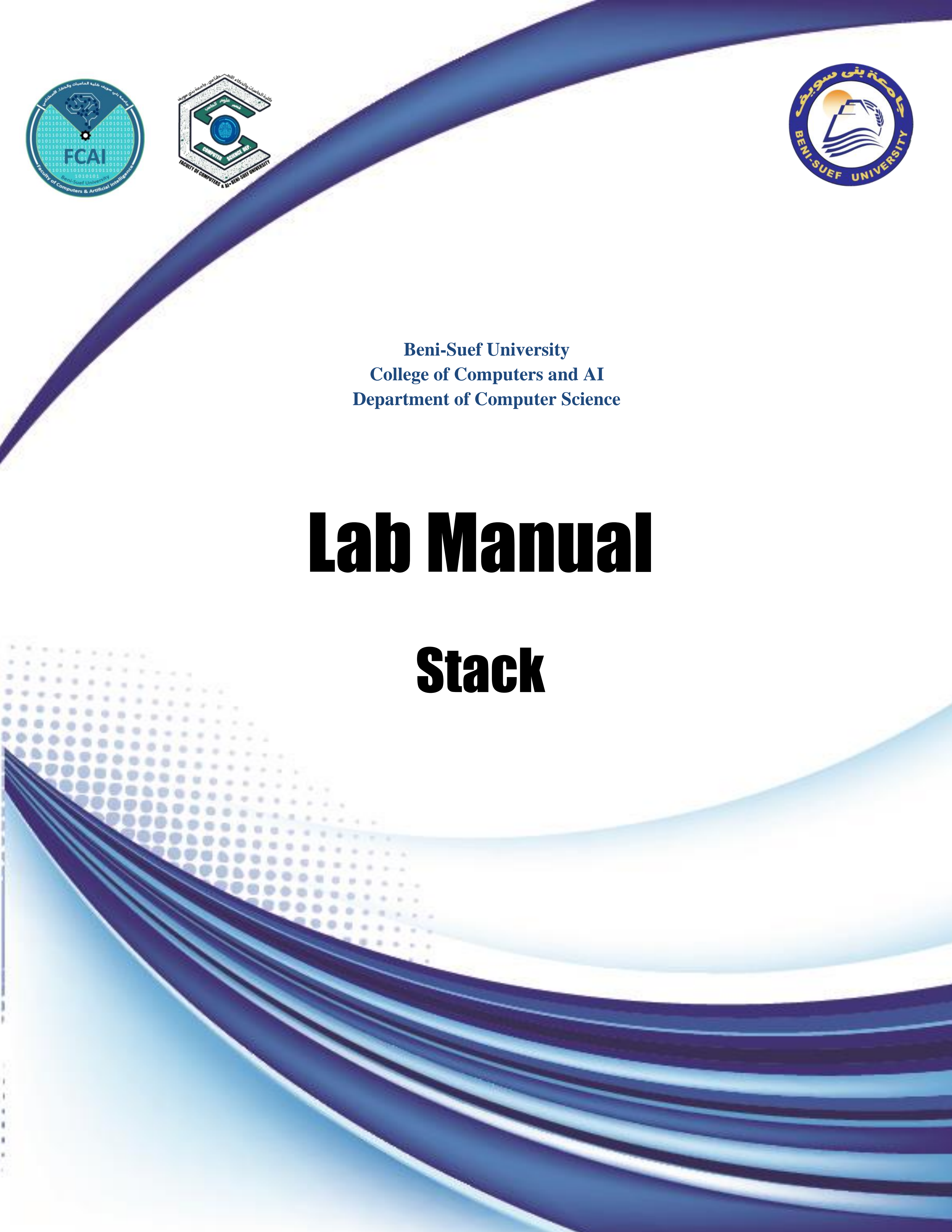
1. **Add Task (at the end)**
2. **Delete Task (by ID)**
3. **Search Task (by ID)**
4. **Update Task (edit details)**
5. **Display Tasks (show all tasks)**

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

// Task Node
class Task {
public:
    int id;
    string description;
    Task* next;

    Task(int i, string desc) {
        id = i;
        description = desc;
        next = nullptr;
    }
};
class TaskManager {
private:
    Task* head;

public:
    TaskManager() {
        head = nullptr;
    }
    // 1. Add Task (at end)
    void addTask(int id, string desc) {
        Task* newTask = new Task(id, desc);

        if (!head) {
            head = newTask;
            return;
        }

        Task* current = head;
```

```
    while (current->next) {
        current = current->next;
    }

    current->next = newTask;
}

// 2. Delete Task by ID
void deleteTask(int id) {
    if (!head) {
        cout << "Task list is empty!\n";
        return;
    }

    // If first node
    if (head->id == id) {
        Task* temp = head;
        head = head->next;
        delete temp;
        cout << "Task " << id << " deleted.\n";
        return;
    }

    Task* current = head;

    while (current->next && current->next->id != id) {
        current = current->next;
    }

    if (current->next) {
        Task* temp = current->next;
        current->next = current->next->next;
        delete temp;
        cout << "Task " << id << " deleted.\n";
    } else {
        cout << "Task " << id << " not found.\n";
    }
}

// 3. Search Task by ID
void searchTask(int id) {
    Task* current = head;

    while (current) {
        if (current->id == id) {
            cout << "Task Found - ID: " << current->id
                 << ", Description: " << current->description << endl;
```

```
                return;
            }
            current = current->next;
        }

        cout << "Task " << id << " not found.\n";
    }

    // 4. Update Task
    void updateTask(int id, string newDesc) {
        Task* current = head;

        while (current) {
            if (current->id == id) {
                current->description = newDesc;
                cout << "Task " << id << " updated.\n";
                return;
            }
            current = current->next;
        }

        cout << "Task " << id << " not found.\n";
    }

    // 5. Display Tasks
    void displayTasks() {
        if (!head) {
            cout << "No tasks available!\n";
            return;
        }
        Task* current = head;

        while (current) {
            cout << "Task ID: " << current->id
                 << " | Description: " << current->description << endl;
            current = current->next;
        }
    }
    // Destructor (free memory)
    ~TaskManager() {
        while (head) {
            Task* temp = head;
            head = head->next;
            delete temp;
        }
    }
};
```

```cpp
// Main Function
int main() {
    TaskManager manager;

    // Adding tasks
    manager.addTask(1, "Complete project report");
    manager.addTask(2, "Prepare presentation");
    manager.addTask(3, "Attend team meeting");

    cout << "\nAll Tasks:\n";
    manager.displayTasks();

    // Search
    cout << "\nSearching for Task ID 2:\n";
    manager.searchTask(2);

    // Update
    cout << "\nUpdating Task ID 2:\n";
    manager.updateTask(2, "Prepare final presentation");

    // Display again
    cout << "\nAfter Update:\n";
    manager.displayTasks();

    // Delete
    cout << "\nDeleting Task ID 1:\n";
    manager.deleteTask(1);

    // Final display
    cout << "\nFinal Tasks:\n";
    manager.displayTasks();

    return 0;
}
```



Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

Stack

Lab - 4
Stack

1. An overview

This lab introduces one of the most fundamental data structures in computer science: the Stack. By the end of this session, you will understand the LIFO (Last In First Out) principle, how stacks work, and how to implement them using both arrays (fixed size) and linked lists (dynamic size). You will also explore real-world applications of stacks.

Learning Objectives

After completing this lab, you should be able to:

- Define what a stack is and explain the LIFO (Last In First Out) principle.
- Identify the basic stack operations: push, pop, peek, isEmpty, and size.
- Implement a stack using arrays (fixed size) with overflow and underflow handling.
- Implement a stack using linked lists (dynamic size) with no size limitations.
- Differentiate between array-based and linked list-based stack implementations.

2. Theory

What is a Stack?

A stack is a linear data structure in which the insertion of a new element and removal of an existing element takes place at the same end represented as the top of the stack.

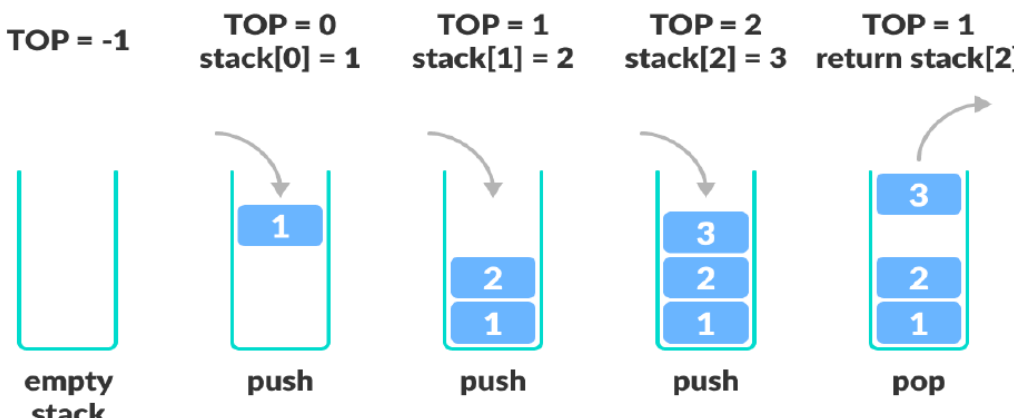
To implement the stack, it is required to maintain the **pointer** to the top of the stack, which is the last element to be inserted because **we can access the elements only on the top of the stack.**

LIFO(Last In First Out):

This strategy states that the element that is inserted last will come out first. You can take a pile of plates kept on top of each other as a real-life example. The plate which we put last is on the top and since we remove the plate that is at the top, we can say that the plate that was put last comes out first. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

Basic Operations on Stack
In order to make manipulations in a stack, there are certain operations provided to us.

1. push() to insert an element into the stack O(1)
2. pop() to remove an element from the stack O(1)
3. top() Returns the top element of the stack. O(1)
4. isEmpty() returns true if stack is empty else false. O(1)
5. size() returns the size of stack.



Types Of Stack:

a. **Fixed Size Stack:** As the name suggests, a fixed size stack has a fixed size and cannot grow or shrink dynamically. If the stack is full and an attempt is made to add an element to it, an overflow error occurs. If the stack is empty and an attempt is made to remove an element from it, an underflow error occurs.

b. **Dynamic Size Stack:** A dynamic size stack can grow or shrink dynamically. When the stack is full, it automatically increases its size to accommodate the new element, and when the stack is empty, it decreases its size. This type of stack is implemented using a linked list, as it allows for easy resizing of the stack.

Applications of the stack:

1. Infix to Postfix /Prefix conversion
2. Redo-undo features at many places like editors, photoshop.
3. Forward and backward features in web browsers
4. Used in many algorithms like Tower of Hanoi, tree traversals, stock span problems, and histogram problems.
5. Backtracking is used to solve problems like the Knight-Tour problem, N-Queen problem, maze problems, and game-like chess or checkers in all these problems we dive into someway if that way is inefficient, we come back to the previous state and go into some another path. To get back from a current state we need to store the previous state in a stack.
6. In Graph Algorithms like Topological Sorting and Strongly Connected Components
7. In Memory management, any modern computer uses a stack as the primary management for a running purpose. Each program that is running in a computer system has its own memory allocations.
8. Stack also helps in implementing function call in computers. The last called function is always completed first.

Advantages of array implementation:

- Easy to implement.
- Memory is saved as pointers are not involved.

Disadvantages of array implementation:

- It is not dynamic i.e., it doesn’t grow and shrink depending on needs at runtime. [But in case of dynamic sized arrays like vector in C++, list in Python, ArrayList in Java, stacks can grow and shrink with array implementation as well].
- The total size of the stack must be defined beforehand.

3. Lab

Exercise 1: Implementing Stack using Arrays:

```
#include <iostream>
using namespace std;
#define size 5
int a[5], top = -1;
int number;
int pop();
void push(int[], int);
int main(){
    // Pushing Numbers into stack
    for (int i = 0; i < size; i++)
    {
        cin >> number;
        push(a, number);
    }
    // Popping Numbers from stack
    for (int i = 0; i < size; i++)
    {
        cout << pop() << endl;
    }
    cout << "/////////////////////\n";
    for (int i = 0; i < size; i++)
        cout << a[i] << endl;
    return 0;
}
// Fixed push
void push(int a[], int number){
    if (top == size - 1)
        cout << "Full Stack or Stack OverFlow\n";
    else
        a[++top] = number;
}
// Fixed pop
int pop(){
    if (top < 0)
    {
        cout << "Empty Stack\n";
        return -1;
    }
    else
        return a[top--];
}
```

Exercise 2: Write a program to insert n numbers into a stack and print them using OOP (classes & Objects)

```
#include <iostream>
using namespace std;
#define MAX 1000
class Stack
{
    int top;
public:
    int a[MAX]; // Maximum size of Stack
    Stack()
    {
        top = -1;
    }

    bool push(int x);
    int pop();
    int peek();
    void display();
    bool isEmpty();
};
bool Stack::push(int x)
{
    if (top >= (MAX - 1))
    {
        cout << "Stack Overflow";
        return false;
    }
    else
    {
        a[++top] = x;
        cout << x << " pushed into stack\n";
        return true;
    }
}
int Stack::pop()
{
    if (top < 0)
    {
        cout << "Stack Underflow";
        return 0;
    }
    else
    {
        int x = a[top--];
        return x;
```

```
    }
}
int Stack::peek()
{
    if (top < 0)
    {
        cout << "Stack is Empty";
        return 0;
    }
    else
    {
        int x = a[top];
        return x;
    }
}
bool Stack::isEmpty()
{
    return (top < 0);
}
void Stack::display()
{
    for (int i = 0; i < top; i++)
    {
        cout << a[i] << endl;
    }
}
// Driver program
int main()
{
    class Stack s;

    s.push(10);
    s.push(20);
    s.push(30);

    cout << s.pop() << " Popped from stack\n";

    s.display();

    system("pause");
    return 0;
}
```

Exercise 3: Write a program to find the maximum number in stack and print it out

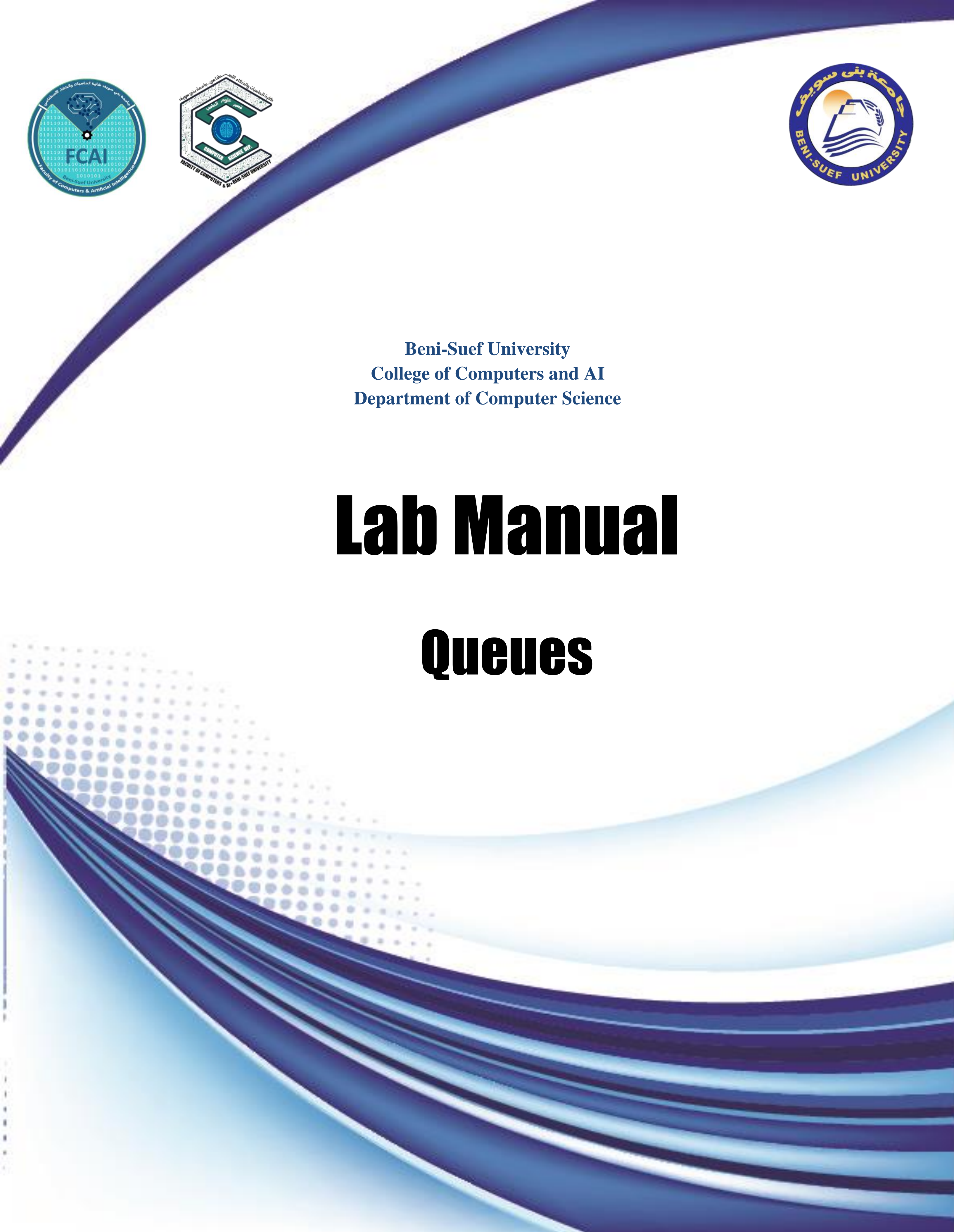
The same first program except modification of pop function as following:

```
int pop()
{
    int max = a[top--];
    for (;;)
    {
        if (top < 0)
            break;
        else
        {
            if (max < a[top])
                max = a[top];
            top--;
        }
    }
    return max;
}
```

Exercise 4: Implementing Stack using Linked List

```
#include <bits/stdc++.h>
using namespace std;

// A structure to represent a stack
class StackNode
{
public:
    int data;
    StackNode* next;
};

// Create new node
StackNode* newNode(int data)
{
    StackNode* stackNode = new StackNode();
    stackNode->data = data;
    stackNode->next = NULL;
    return stackNode;
}

// Check if empty
int isEmpty(StackNode* root)
{
    return !root;
}

// Push
void push(StackNode** root, int data)
{
    StackNode* stackNode = newNode(data);
    stackNode->next = *root;
    *root = stackNode;

    cout << data << " pushed to stack\n";
}

// Pop
int pop(StackNode** root)
{
    if (isEmpty(*root))
        return INT_MIN;

    StackNode* temp = *root;
    *root = (*root)->next;
```

```
    int popped = temp->data;
    free(temp);

    return popped;
}

// Peek
int peek(StackNode* root)
{
    if (isEmpty(root))
        return INT_MIN;

    return root->data;
}

// Driver code
int main()
{
    StackNode* root = NULL;

    push(&root, 10);
    push(&root, 20);
    push(&root, 30);

    cout << pop(&root) << " popped from stack\n";

    cout << "Top element is " << peek(root) << endl;

    cout << "Elements present in stack : ";

    // print all elements
    while (!isEmpty(root))
    {
        cout << peek(root) << " ";
        pop(&root);
    }

    return 0;
}
```



Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

Queues

Lab - 5
Queues

1. An overview

This lab introduces one of the most fundamental data structures in computer science: the **Queue**. By the end of this session, you will understand the FIFO (First In First Out) principle, how queues work, and how to implement them using both arrays (fixed size) and linked lists (dynamic size). You will also explore real-world applications of queues in job scheduling, call centers, and order processing.

Learning Objectives

After completing this lab, you should be able to:

- **Define** what a queue is and explain the FIFO (First In First Out) principle.
- **Identify** the basic queue operations: enQueue, deQueue, front, rear, isEmpty, and size.
- **Implement** a queue using arrays (fixed size) with overflow and underflow handling.
- **Implement** a queue using linked lists (dynamic size) with no size limitations.
- **Differentiate** between array-based and linked list-based queue implementations.
- **Analyze** the time complexity of queue operations (enQueue O(1), deQueue O(1) for linked list, O(n) for array).
- **Apply** queues to solve real-world problems like call center simulations.
- **Understand** the applications of queues in BFS, job scheduling, and print spooling.

2.Theory

What is a Queue?
A queue is an abstract data structure that contains a collection of elements. Queue implements the FIFO mechanism i.e. the element that is inserted first is also deleted first. In other words, the least recently added element is removed first in a queue.

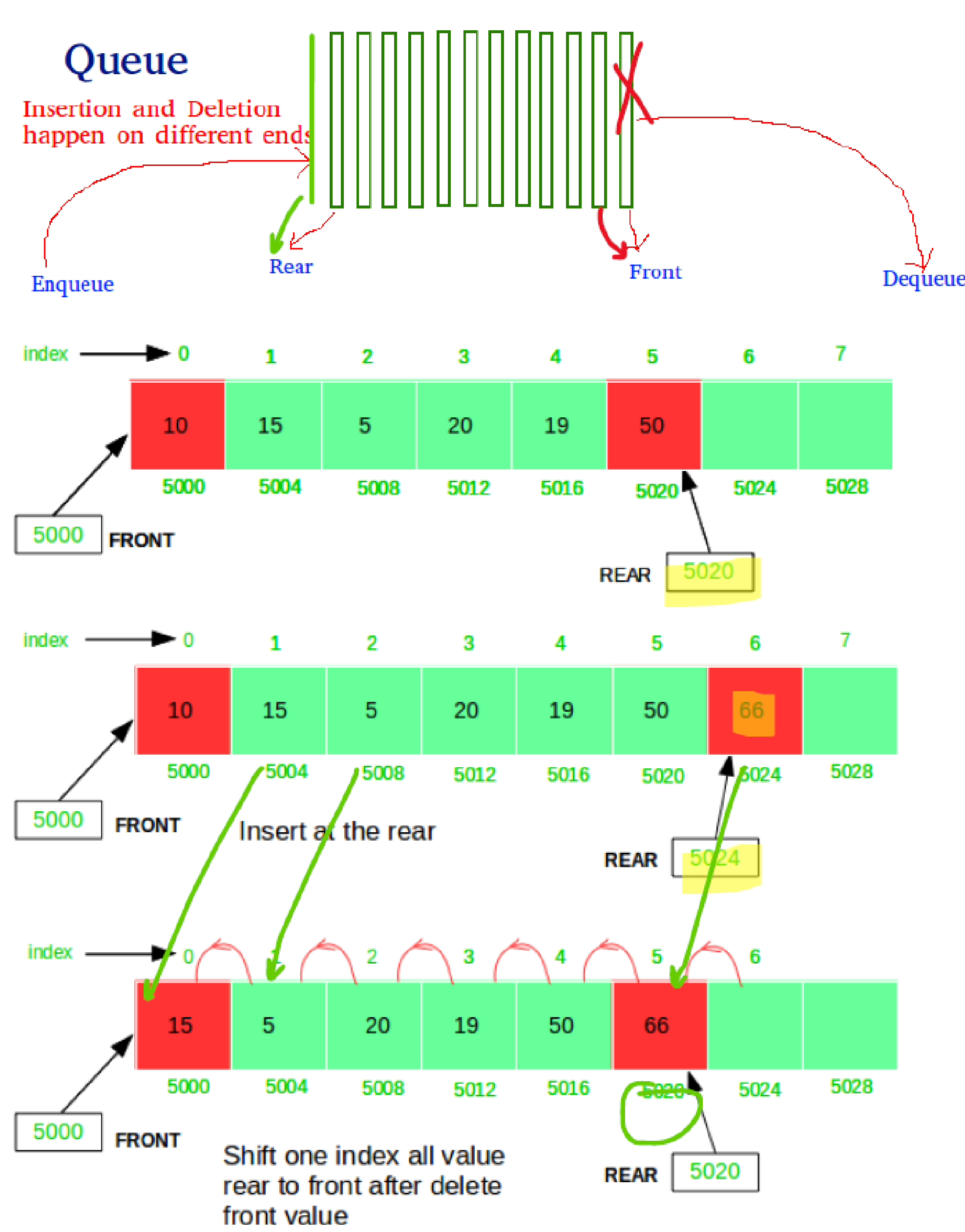
Simple definition: Think of a queue like a line of people waiting at a ticket counter. The person who arrives first is served first.

A program that implements the queue using an array is given as follows –

Another Example:

Basic Operations

1. Front: Get the front item from queue.
2. Rear: Get the last item from queue.
3. enQueue(value)
4. deQueue()
5. peek() − Gets the element at the front of the queue without removing it.
6. isfull() − Checks if the queue is full.
7. isempty() − Checks if the queue is empty.



Applications of Queue Data Structure

1. **Job Scheduling**
   Queues are vital in operating systems for managing processes and jobs. In job scheduling, tasks are kept in a queue, and the CPU or any other resource processes them in the order they arrive. This can be seen in print spooling where print jobs are queued to a printer.

2. **Breadth-First Search (BFS) Algorithm**
   In graph theory, BFS is a traversing algorithm that uses a queue to explore the graph or tree, level by level. This ensures that all vertices or nodes at the current depth (or distance from the start point) are explored before moving on to nodes at the next depth level.

3. **Buffering**
   Queues can be used to manage data packets in networking where data buffering is required. They help in handling asynchronous data transmission between different components of applications, like in I/O buffers.

4. **Call Center Systems**
   In customer service call centers, incoming calls are held in a queue until they can be directed to the next available service representative. This is a practical application of queue which can also be simulated in C++ for training or operational management purposes.

5. **Handling of Interrupts in Real-Time Systems**
   In real-time systems, interrupts may need to be handled in the order they occur, which can be managed by placing them in a queue. This ensures that each interrupt is processed in a timely and systematic manner.

6. **Order Processing**
   In retail or any scenario where orders must be processed in the order they are received, queues can manage and facilitate orderly processing. This is typical in E-commerce systems where transactions are processed in a sequential order.

3. Lab

Exercise 1: Implementing Queue using Array

```
#include <iostream>
using namespace std;
class Queue
{
private:
    int front, rear, capacity;
    int* queue;
public:
    Queue(int c)
    {
        front = rear = 0;
        capacity = c;
        queue = new int[capacity];
    }
    ~Queue()
    {
        delete[] queue;
    }
    void queueEnqueue(int data)
    {
        if (rear == capacity)
        {
            cout << "\nQueue is full\n";
            return;
        }
        else
        {
            queue[rear] = data;
            rear++;
        }
    }
    void queueDequeue(){
        if (front == rear)
        {
            cout << "\nQueue is empty\n";
            return;
        }
        else
        {
            // Shift elements to left
            for (int i = 0; i < rear - 1; i++)
            {
```

```
                queue[i] = queue[i + 1];
            }

            rear--;
        }
    }
    void queueDisplay()
    {
        if (front == rear)
        {
            cout << "\nQueue is Empty\n";
            return;
        }
        for (int i = front; i < rear; i++)
        {
            cout << queue[i] << " <-- ";
        }
        cout << "\n";
    }
    void queueFront()
    {
        if (front == rear)
        {
            cout << "\nQueue is Empty\n";
            return;
        }

        cout << "\nFront Element is: " << queue[front] << "\n";
    }
    void search(int searchItem)
    {
        bool found = false;

        for (int i = front; i < rear; i++)
        {
            if (queue[i] == searchItem)
            {
                found = true;
                break;
            }
        }
        if (found)
            cout << "Item exists\n";
        else
            cout << "Item not exists\n";
    }
};
```

```
int main()
{
    Queue q(4);

    q.queueDisplay();

    q.queueEnqueue(20);
    q.queueEnqueue(30);
    q.queueEnqueue(40);
    q.queueEnqueue(50);

    q.search(20);

    q.queueDisplay();

    q.queueEnqueue(60);

    q.queueDisplay();

    q.queueDequeue();
    q.queueDequeue();

    cout << "\n\nafter two node deletion\n\n";

    q.queueDisplay();

    q.queueFront();

    return 0;
}
```

Exercise 2: Implementing Queue using Linked List

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

// Node structure
struct QNode
{
    int data;
    QNode* next;

    QNode(int d)
    {
        data = d;
        next = NULL;
    }
};

// Queue structure
struct Queue
{
    QNode* front;
    QNode* rear;

    Queue()
    {
        front = rear = NULL;
    }

    // Enqueue
    void enQueue(int x)
    {
        QNode* temp = new QNode(x);

        // If queue is empty
        if (rear == NULL)
        {
            front = rear = temp;
            return;
        }

        rear->next = temp;
        rear = temp;
    }

    // Dequeue
```

```
    void deQueue()
    {
        if (front == NULL)
            return;

        QNode* temp = front;
        front = front->next;

        if (front == NULL)
            rear = NULL;

        delete temp;
    }
};

// Driver code
int main()
{
    Queue q;

    q.enQueue(10);
    q.enQueue(20);

    q.deQueue();
    q.deQueue();

    q.enQueue(30);
    q.enQueue(40);
    q.enQueue(50);

    q.deQueue();

    cout << "Queue Front : "
         << ((q.front != NULL) ? (q.front->data) : -1) << endl;

    cout << "Queue Rear : "
         << ((q.rear != NULL) ? (q.rear->data) : -1);

    return 0;
}
```

Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

Trees & BST

Lab - 5
Trees & BST

1. An overview

This lab introduces one of the most important non-linear data structures in computer science: **Trees** and specifically **Binary Search Trees (BST)**. By the end of this session, you will understand the hierarchical nature of trees, the BST property, and how to perform basic operations like insertion, searching, and different types of traversals. You will also explore real-world applications of trees in databases, file systems, and compilers.

Learning Objectives

After completing this lab, you should be able to:

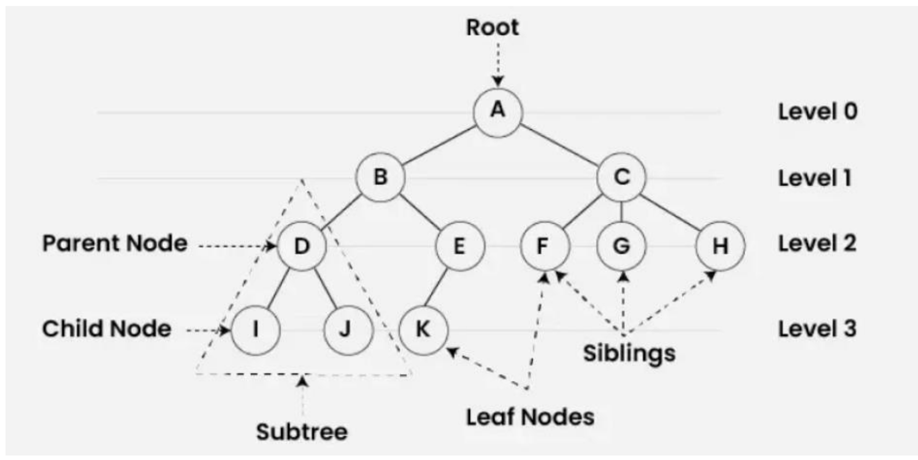
- Define what a tree is and explain its hierarchical structure (nodes, edges, root, leaves).
- Identify the properties of binary trees
- Calculate the size of a tree recursively.
- Define what a Binary Search Tree (BST) is and explain the BST property .
- Implement BST operations: insertion and searching.
- Perform four types of tree traversals:
  - In-Order (LVR) - visits nodes in ascending order
  - Pre-Order (VLR) - useful for creating tree copies
  - Post-Order (LRV) - useful for deleting trees
  - Level-Order (BFS) - visits level by level
- Differentiate between depth-first traversals (In, Pre, Post) and breadth-first traversal (Level-Order).
- Understand the recursive nature of tree structures and operations.

2.Theory

What is a Tree?
Trees are hierarchical data structures that contain nodes connected by edges. They are recursive in nature, which means that they are made up of smaller instances of themselves. Various types such as binary tree, etc. have different characteristics to make them suitable for different applications such as storing sorted data, dictionaries, routing tables, etc.

Simple definition: Think of a tree like a family tree or an organizational chart. Each person (node) has children connected below them.

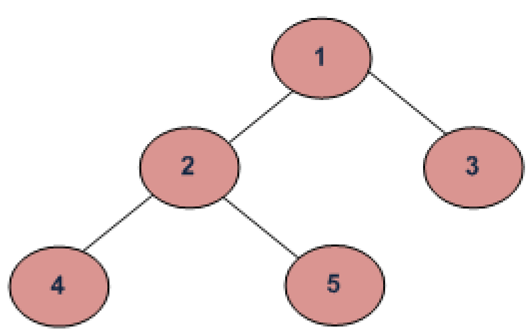
Basic Terminology in Tree:



- Root: Node with no parent (A)
- Parent: Node that has children (A is parent of B, C)
- Child: Node connected to parent (B, C are children of A)
- Sibling: Nodes with same parent (B, C are siblings)
- Leaf: Node with no children (I, J are leaves)
- Edge: Connection between two nodes
- Level: Distance from root (A: level 0, B,C: level 1, E: level 2)
- Height: Number of edges from node to deepest leaf
- Depth: Number of edges from root to node

Prosperities of Binary Trees
A binary tree is a tree where each node has at most two children (left and right).

1. The maximum number of nodes at level ‘l’ of a binary tree is 2l
   For root, l = 0, number of nodes = 20 = 1
2. The Maximum number of nodes in a binary tree of height ‘h’ is 2h – 1



Size of the tree
Size of a tree is the number of elements present in the tree. Size of the above tree is 5. Size() function recursively calculates the size of a tree. It works as follows:
Size of a tree = Size of left subtree + 1 + Size of right subtree.

What is Binary Search Tree?
Binary Search Tree (BST) is a special type of binary tree in which the left child of a node has a value less than the node's value and the right child has a value greater than the node's value. This property is called the BST property and it makes it possible to efficiently search, insert, and delete elements in the tree.

Properties of Binary Search Tree:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- This means everything to the left of the root is less than the value of the root and everything to the right of the root is greater than the value of the root. Due to this performing, a binary search is very easy.
- The left and right subtree each must also be a binary search tree.
- There must be no duplicate nodes(BST may have duplicate values with different handling approaches)

Handling duplicate values in the Binary Search Tree:
We must follow a consistent process throughout i.e. either store duplicate value at the left or store the duplicate value at the right of the root, but be consistent with your approach.

Types of Binary tree traversal:

1. In-Order Traversal (or Infix Traversal): (LVR)

- In in-order traversal, the nodes are traversed in the following order:
- Traverse the left subtree recursively.
- Visit the root node.
- Traverse the right subtree recursively.
- This traversal method is commonly used with binary search trees because it visits nodes in ascending order.

2. Pre-Order Traversal (or Prefix Traversal):VLR

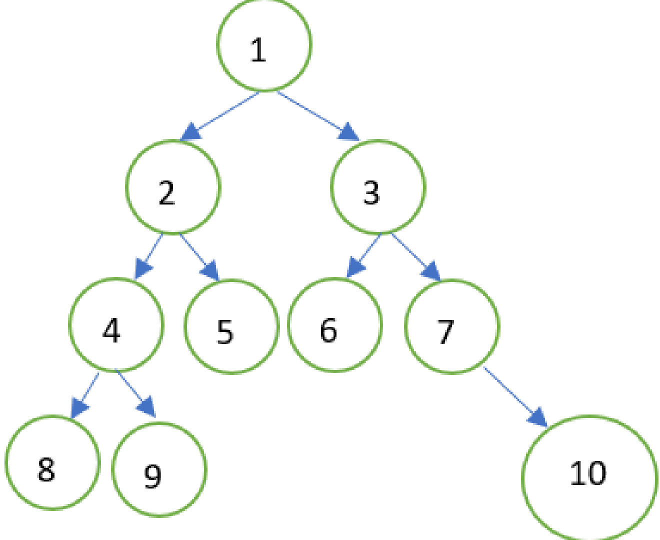
- In pre-order traversal, the traversal order is:
- Visit the root node.
- Traverse the left subtree recursively.
- Traverse the right subtree recursively.
- This method is useful for creating a copy of the tree or for prefix notation expressions (like those used in compilers).

3. Post-Order Traversal (or Postfix Traversal): (LRV)

- In post-order traversal, the nodes are visited in the following sequence:
- Traverse the left subtree recursively.
- Traverse the right subtree recursively.
- Visit the root node.
- This type of traversal is useful for deleting or freeing nodes and space of the tree, calculating the size of the tree, and post-fix expression (used in expression trees).

4. Level-Order Traversal (or Breadth-First Traversal):

- In level-order traversal, nodes are visited level by level from top to bottom, and from left to right on each level. This traversal is different from the previous ones as it is not a depth-first approach.
- It requires additional memory, typically a queue, to track nodes at each level before visiting child nodes.
- This traversal is particularly useful for scenarios where you need to visit nodes in a tiered manner, such as in many tree algorithms including decision trees.



With this structure, we'll show how each traversal method operates:

1. In-Order Traversal (Left, Root, Right)
- Starts at the leftmost node, progresses to its root, and continues to the right, recursively.
- Output for this tree: 8, 4, 9, 2, 5, 1, 6, 3, 7, 10

2. Pre-Order Traversal (Root, Left, Right)
- Visits the root first, then the left subtree, and finally the right subtree, recursively.
- Output for this tree: 1, 2, 4, 8, 9, 5, 3, 6, 7, 10

3. Post-Order Traversal (Left, Right, Root)
- Completes visits of left and right subtrees before processing the root, recursively.
- Output for this tree: 8, 9, 4, 5, 2, 6, 10, 7, 3, 1

4. Level-Order Traversal (Breadth-First)
- Processes nodes level by level from top to bottom.
- Output for this tree: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

3. Lab

Exercise 1: Calculate size of tree

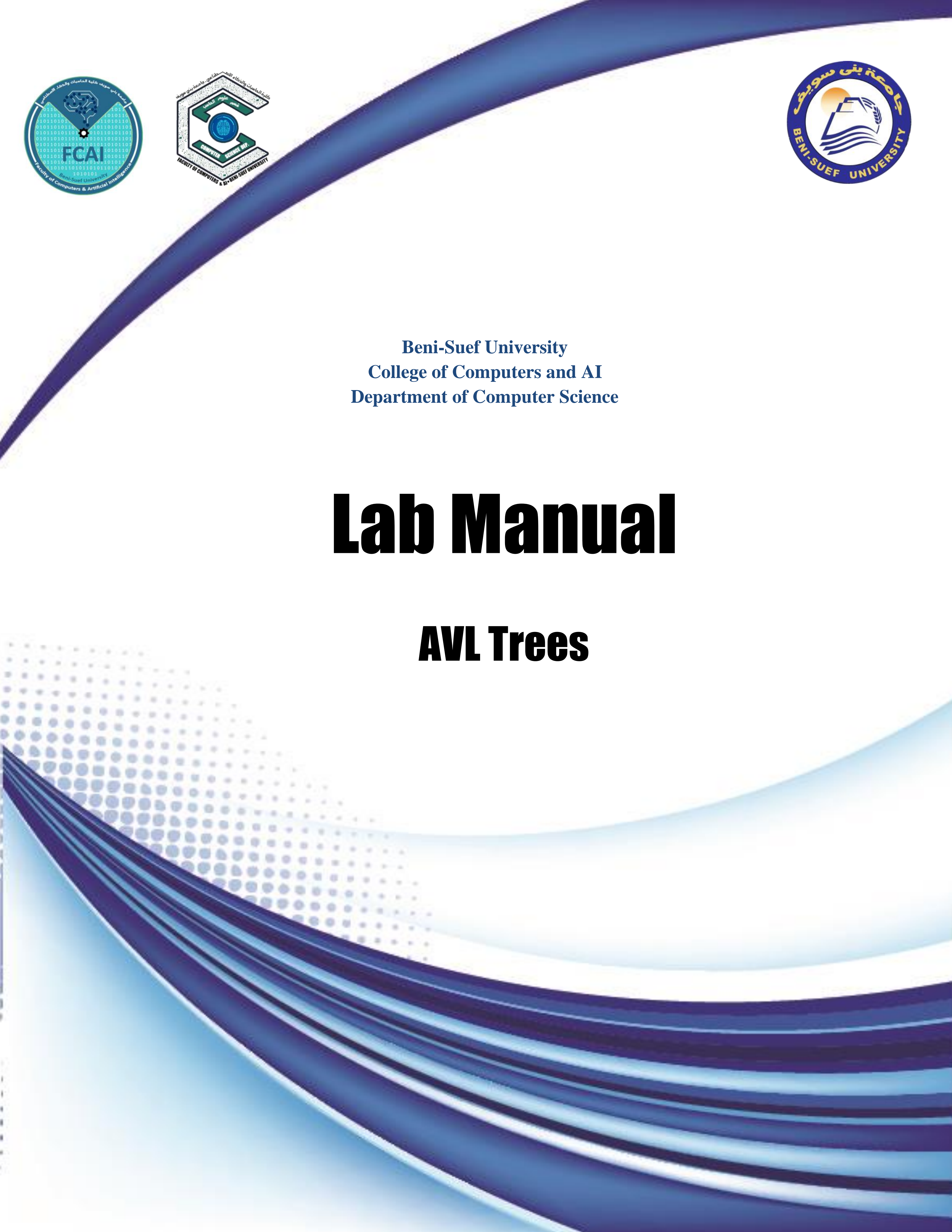
```cpp
#include <iostream>
using namespace std;

class node {
public:
    int data;
    node* left;
    node* right;
};

node* newNode(int data) {
    node* Node = new node();
    Node->data = data;
    Node->left = NULL;
    Node->right = NULL;
    return Node;
}

int size(node* root) {
    if (root == NULL)
        return 0;
    return size(root->left) + 1 + size(root->right);
}

int main() {
    node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    cout << "Size of the tree is " << size(root);
    return 0;
}
```

Exercise 2: BST Implementation

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

struct node {
    int key;
    struct node *left, *right;
};

// A utility function to create a new BST node
struct node* newNode(int item) {
    struct node* temp = new node;
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to insert a new node with given key in BST
struct node* insert(struct node* node, int key) {
    // If the tree is empty, return a new node
    if (node == NULL)
        return newNode(key);

    // Otherwise, recur down the tree
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    // Return the (unchanged) node pointer
    return node;
}

// Utility function to search a key in a BST
struct node* search(struct node* root, int key) {
    // Base Cases: root is null or key is present at root
    if (root == NULL || root->key == key)
        return root;

    // Key is greater than root's key
    if (root->key < key)
        return search(root->right, key);

    // Key is smaller than root's key
    return search(root->left, key);
```

```
}

// In-order traversal function to print BST elements in ascending order
void inOrderTraversal(struct node* root) {
    if (root != NULL) {
        inOrderTraversal(root->left);
        cout << root->key << " ";
        inOrderTraversal(root->right);
    }
}

// Pre-order traversal function to print BST elements in pre-order
void preOrderTraversal(struct node* root) {
    if (root != NULL) {
        cout << root->key << " ";
        preOrderTraversal(root->left);
        preOrderTraversal(root->right);
    }
}

// Post-order traversal function to print BST elements in post-order
void postOrderTraversal(struct node* root) {
    if (root != NULL) {
        postOrderTraversal(root->left);
        postOrderTraversal(root->right);
        cout << root->key << " ";
    }
}

// Driver Code
int main() {
    struct node* root = NULL;

    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    cout << "In-order traversal of the BST: ";
    inOrderTraversal(root);
    cout << endl;

    cout << "Pre-order traversal of the BST: ";
    preOrderTraversal(root);
```

```
    cout << endl;

    cout << "Post-order traversal of the BST: ";
    postOrderTraversal(root);
    cout << endl;

    // Key to be found
    int key = 60;

    // Searching in a BST
    if (search(root, key) == NULL)
        cout << key << " not found" << endl;
    else
        cout << key << " found" << endl;

    system("pause");
    return 0;
}
```



Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

AVL Trees

Lab - 7
AVL Trees

1. An overview

This lab introduces one of the most important self-balancing binary search trees: the AVL Tree. By the end of this session, you will understand why balancing is necessary, how balance factors work, and how to perform rotations to maintain balance after insertions. You will also implement a complete AVL tree with insertion and rotation operations.

Learning Objectives

After completing this lab, you should be able to:

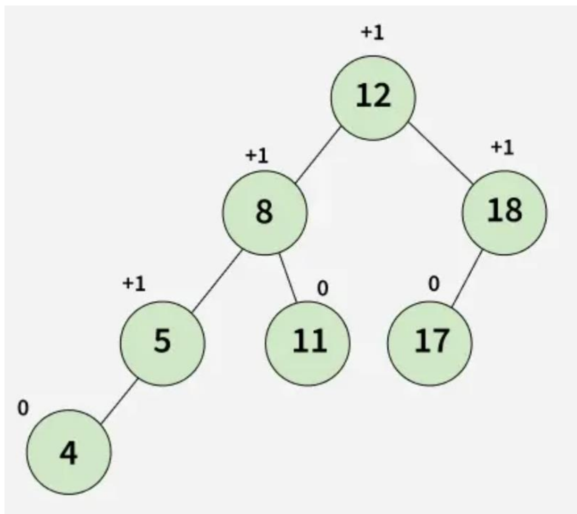
- Define what an AVL tree is and explain why self-balancing is important.
- Calculate the balance factor for any node (left height - right height).
- Identify the four imbalance cases: LL, RR, LR, and RL.
- Implement right rotation and left rotation to fix imbalances.
- Perform AVL tree insertion with automatic rebalancing.
- Understand the time complexity of AVL operations (O(log n)).
- Differentiate between AVL trees and regular BSTs.

2. Theory

What is an AVL Tree?
An AVL tree defined as a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees for any node cannot be more than one.

Balance Factor = left subtree height - right subtree height
For a Balanced Tree(for every node): -1 ≤ Balance Factor ≤ 1 which means it belongs to these values {-1,0,1}.

Example of an AVL Tree:



The balance factors for different nodes are:
- 12 : +1
- 8 :   +1
- 18 : +1
- 5 :   +1
- 11 : 0
- 17 : 0
- 4 :   0.

Since all differences are lies between -1 to +1, so the tree **is an AVL tree**.

Note: Height of the tree means the number of edges from the root node to leaf node, for example the height of the tree from the root node is 3.

Example of a BST which is not an AVL Tree:
The Below Tree is not an AVL Tree as the balance factor for nodes 8 and 12 is more than 1.

Important Points about AVL Tree:

- Rotations: rotations are designed to restore balance in O(1) time while ensuring the overall time complexity remains O(log n). AVL Trees use four cases to rebalance themselves after insertions and deletions:
    - Left-Left (LL)
    - Right-Right (RR)
    - Left-Right (LR)
    - Right-Left (RL)
- insertion and Deletion: While insertion is followed by upward traversals to check balance and apply rotations, deletion can be more complex due to multiple rotations possibly being required. AVL Trees may require multiple rebalancing steps during deletion, unlike Red-Black Trees which limit this better.

- Use Cases: AVL Trees are particularly useful when you need frequent and efficient lookups, like :-
- database indexing
- memory-intensive applications, or where predictable time complexity is crucial.

Operations on an AVL Tree:

- **Searching :** It is same as normal Binary Search Tree (BST) as an AVL Tree is always a BST. So we can use the same implementation as BST. The advantage here is time complexity is O(log n)
- **Insertion :** It does rotations along with normal BST insertion to make sure that the balance factor of the impacted nodes is less than or equal to 1 after insertion
- **Deletion :** It also does rotations along with normal BST deletion to make sure that the balance factor of the impacted nodes is less than or equal to 1 after deletion.

Rotating the subtrees (Used in Insertion and Deletion)

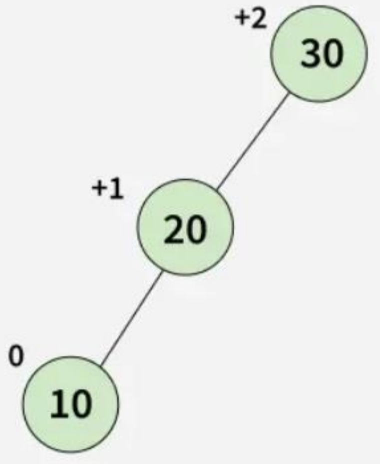
An AVL tree may rotate in one of the following four ways to keep itself balanced while making sure that the BST properties are maintained.

1. Left-Left Case:

Occurs when a node is inserted into the left subtree of the left child, causing the balance factor to become **more than +1.**

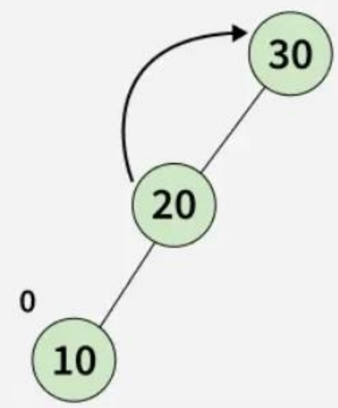
Fix: Perform a single right rotation.

01 Step | Left unbalanced tree



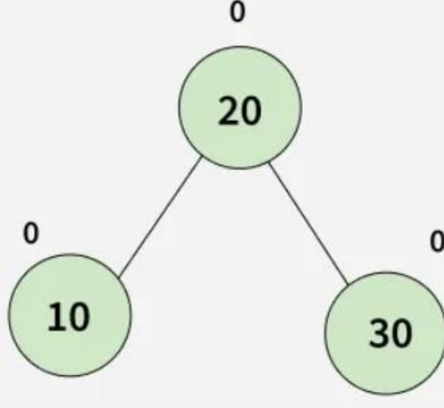
Node '30' has a balance factor of +2, which lies outside the acceptable range of -1 to +1.

02 Step | Performing right rotation



To achieve balance in the tree, a right rotation is performed on nodes 20 and 30

03 Step | Balanced tree

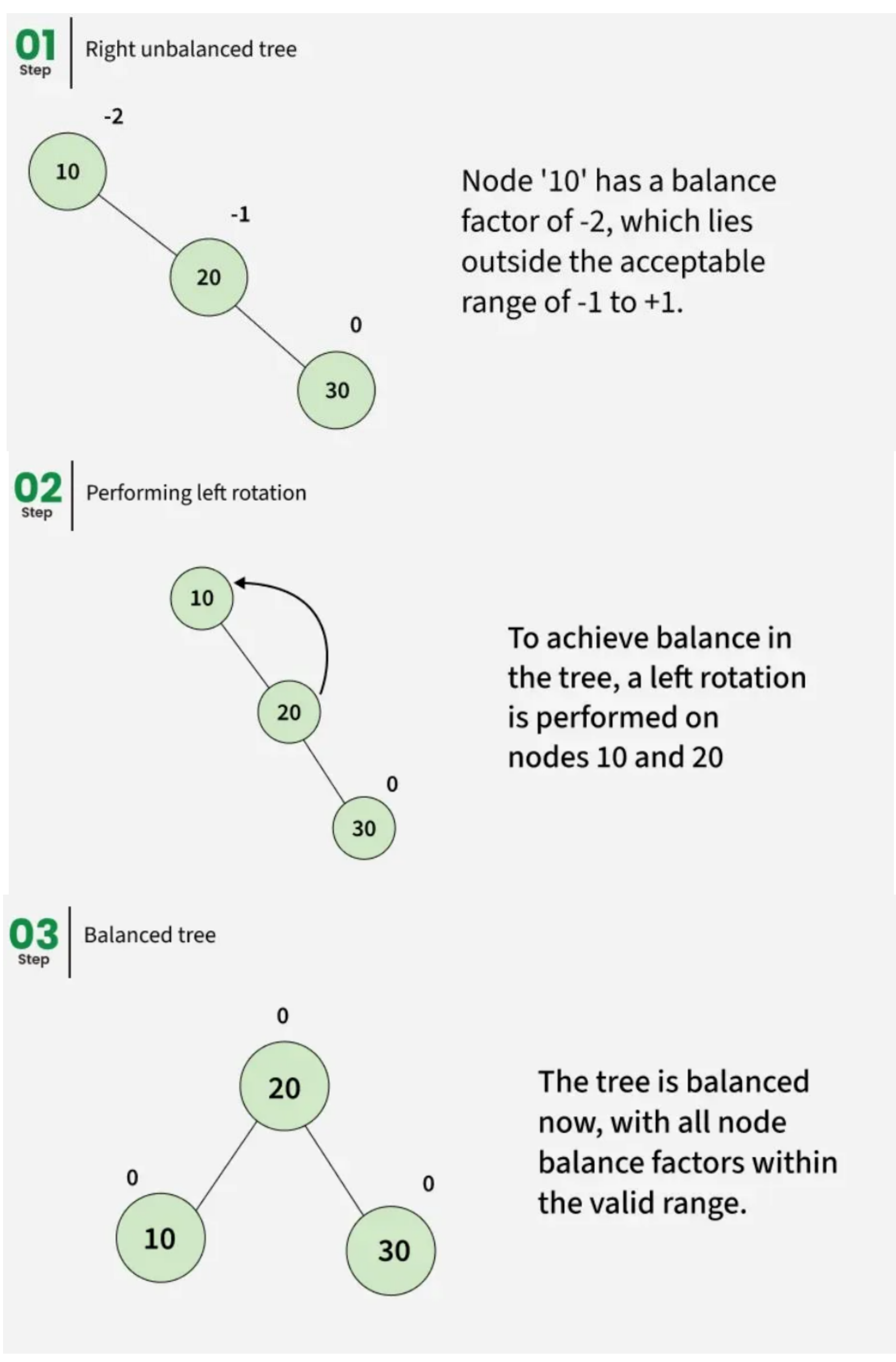


The tree is balanced now, with all node balance factors within the valid range.

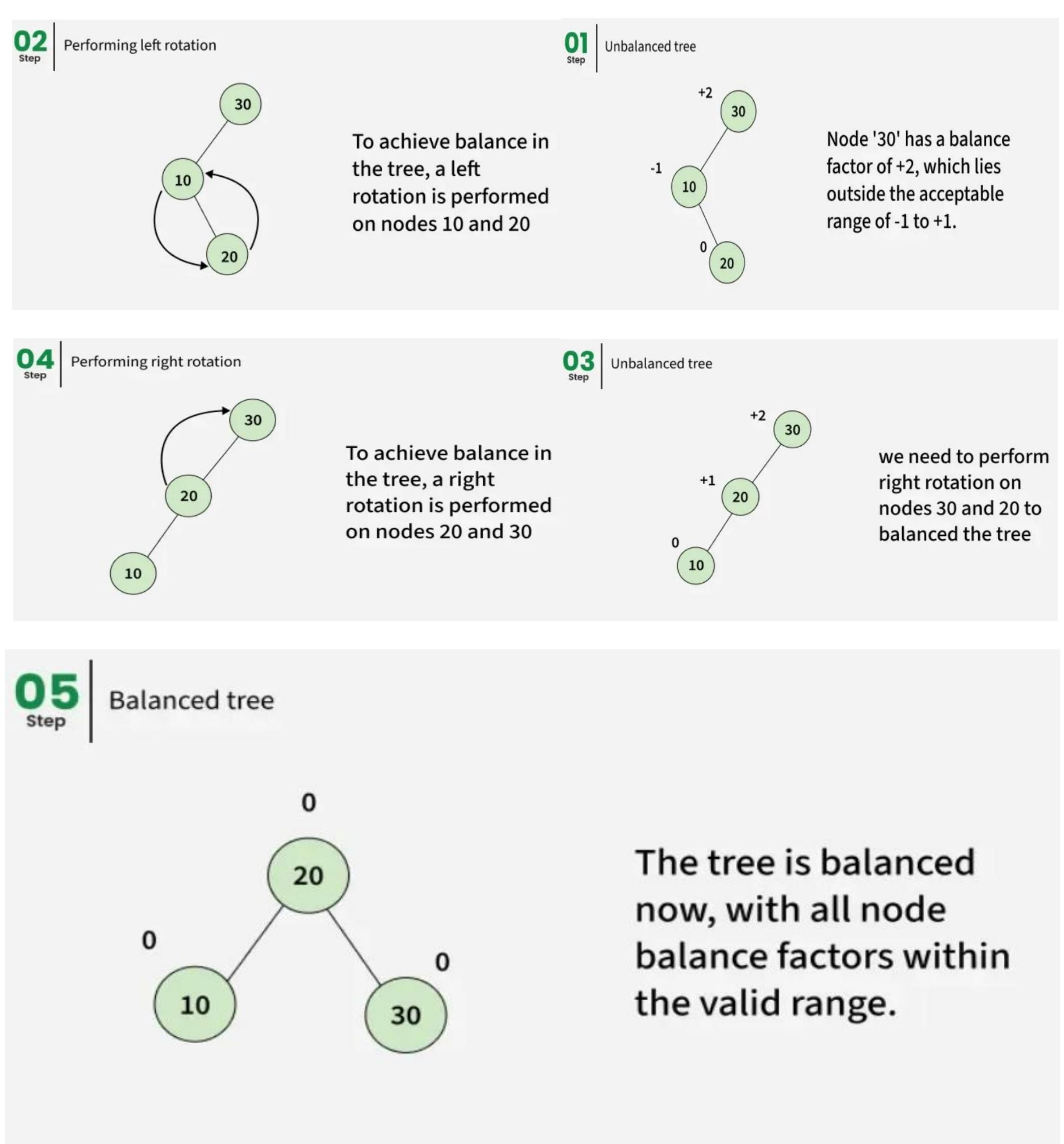
2. Right-Right Case:

Occurs when a node is inserted into the right subtree of the right child, making the balance factor less than -1.

Fix: Perform a single left rotation.



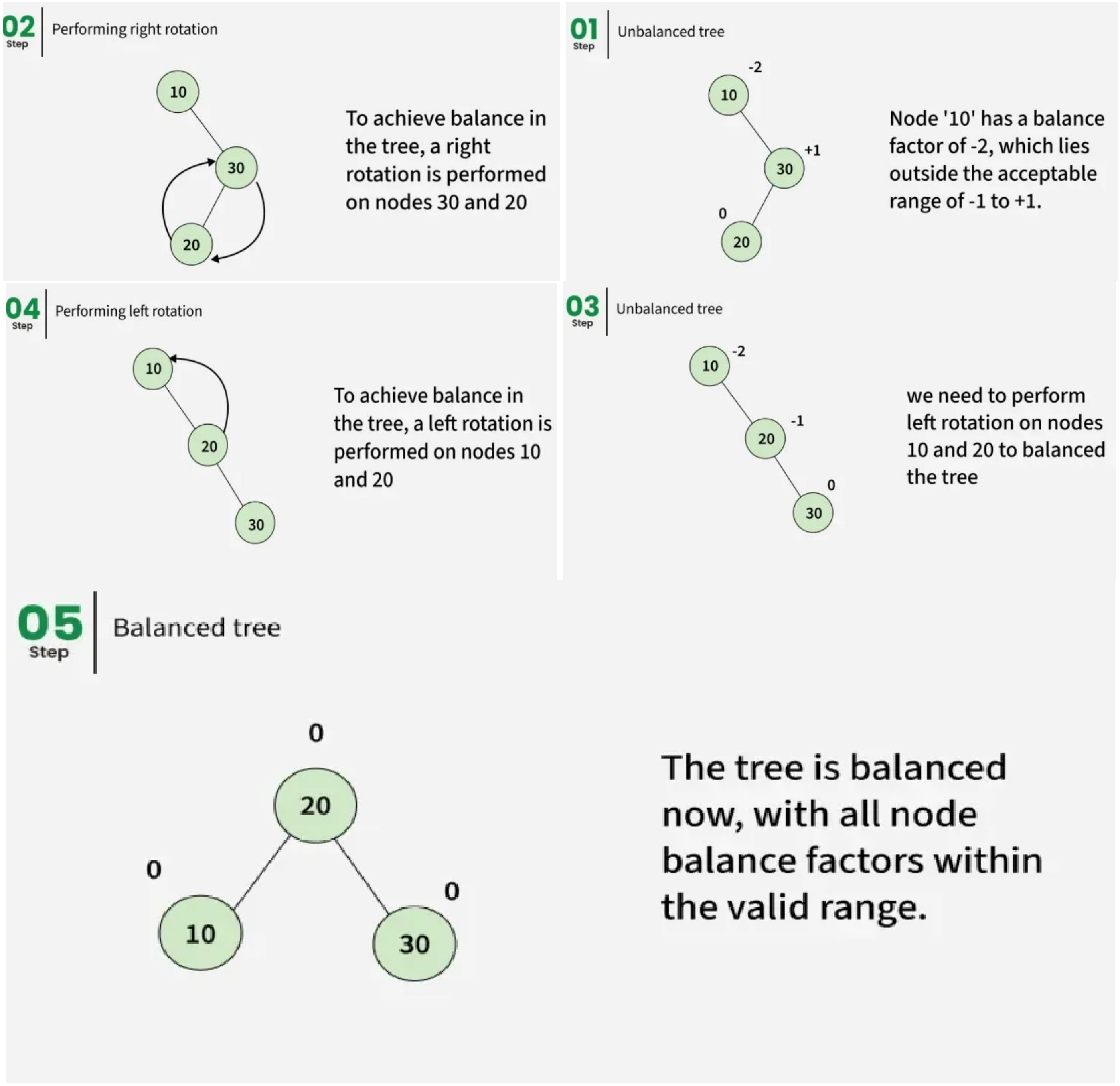
3. Left-Right Case:
Occurs when a node is inserted into the right subtree of the left child, which disturbs the balance factor of an ancestor node, making it left-heavy.
Fix: Perform a left rotation on the left child, followed by a right rotation on the node.



4. Right-Left Case:

Occurs when a node is inserted into the left subtree of the right child, which disturbs the balance factor of an ancestor node, making it right-heavy.

Fix: Perform a right rotation on the right child, followed by a left rotation on the node.



Advantages of AVL Tree:

1. AVL trees can self-balance themselves and therefore provides time complexity as O(log n) for search, insert and delete.
2. As it is a balanced BST, so items can be traversed in sorted order.
3. Since the balancing rules are strict compared to Red Black Tree, AVL trees in general have relatively less height and hence the search is faster.
4. AVL tree is relatively less complex to understand and implement compared to Red Black Trees.

Disadvantages of AVL Tree:

1. It is difficult to implement compared to normal BST.
2. Less used compared to Red-Black trees. Due to its rather strict balance.
3. AVL trees provide complicated insertion and removal operations as more rotations are performed.

Insertion in AVL Tree:

To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).

- Left Rotation
- Right Rotation

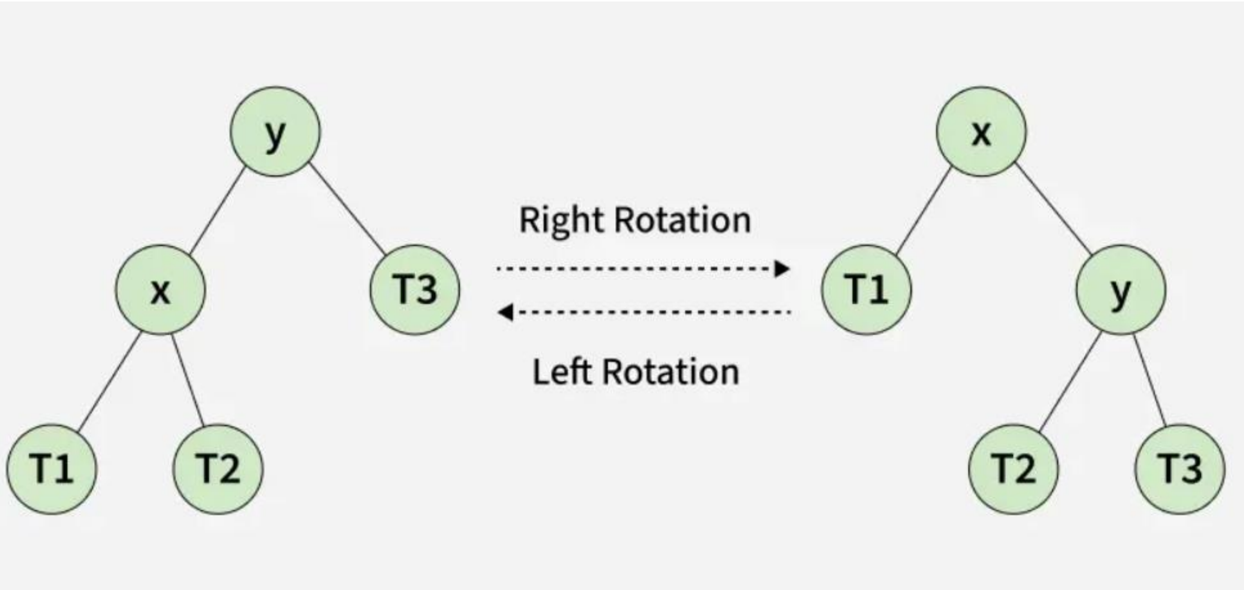
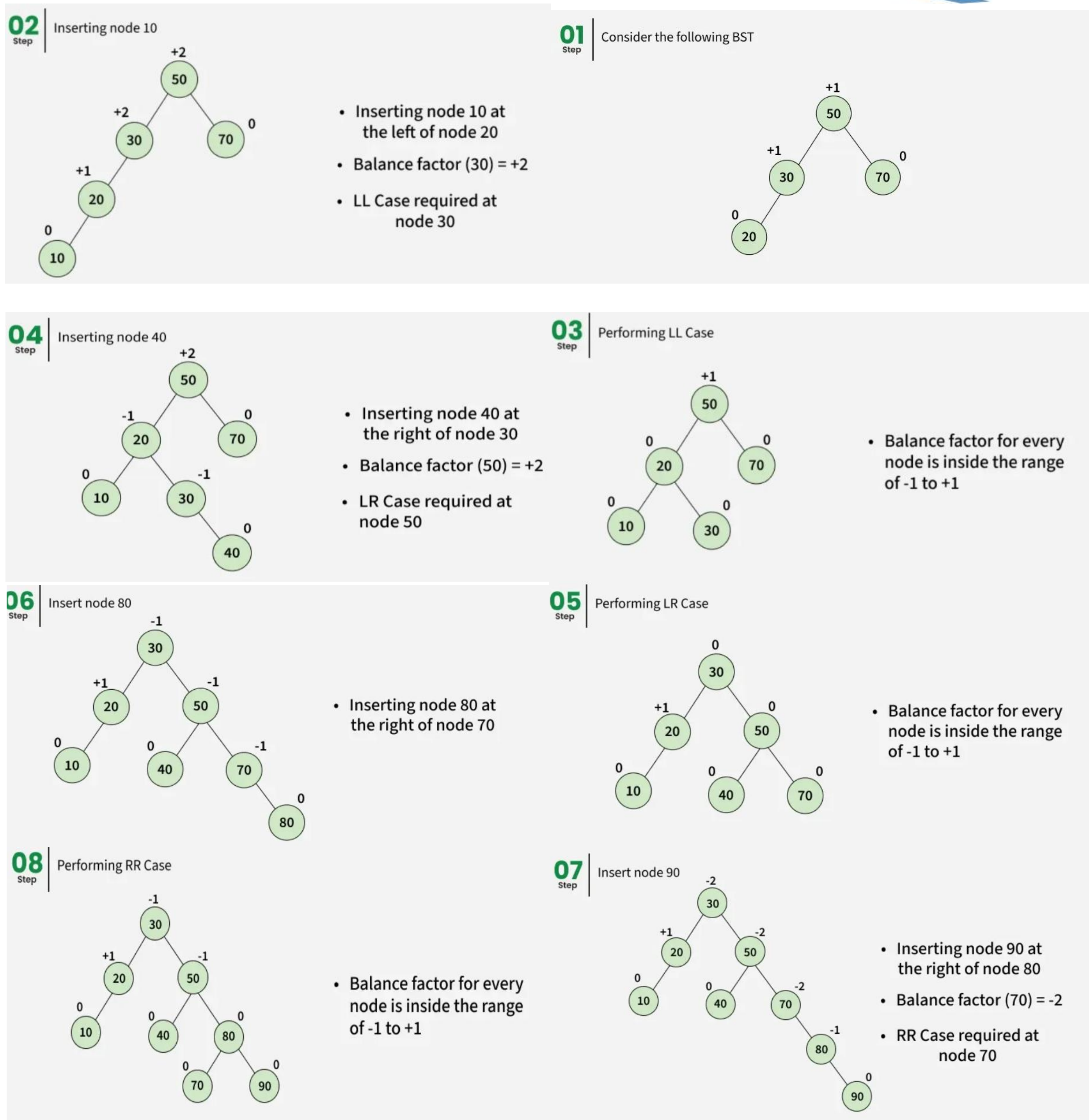
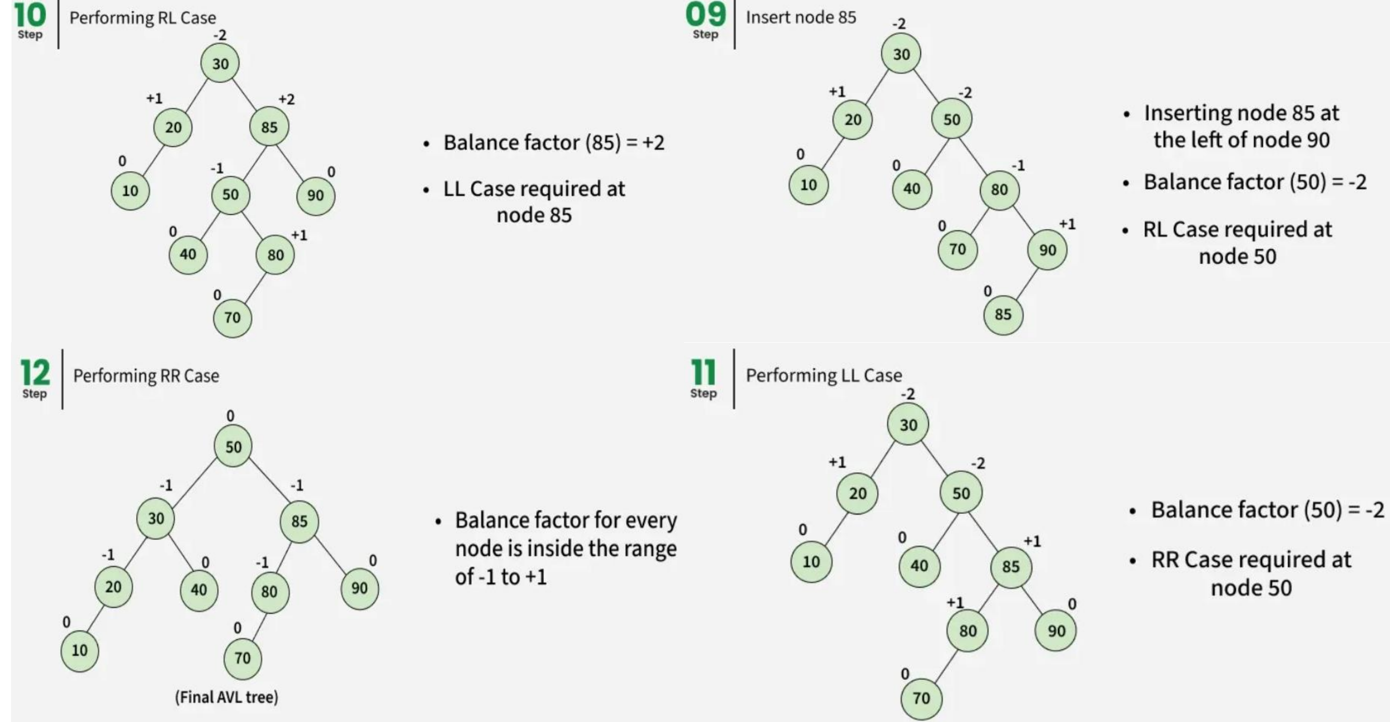


Illustration of Insertion at AVL Tree:


01 Step
Consider the following BST
02 Step
Inserting node 10
• Inserting node 10 at the left of node 20
• Balance factor (30) = +2
• LL Case required at node 30
03 Step
Performing LL Case
• Balance factor for every node is inside the range of -1 to +1
04 Step
Inserting node 40
• Inserting node 40 at the right of node 30
• Balance factor (50) = +2
• LR Case required at node 50
05 Step
Performing LR Case
• Balance factor for every node is inside the range of -1 to +1
06 Step
Insert node 80
• Inserting node 80 at the right of node 70
07 Step
Insert node 90
• Inserting node 90 at the right of node 80
• Balance factor (70) = -2
• RR Case required at node 70
08 Step
Performing RR Case
• Balance factor for every node is inside the range of -1 to +1


10 Step
Performing RL Case
Balance factor (85) = +2
LL Case required at node 85
09 Step
Insert node 85
Inserting node 85 at the left of node 90
Balance factor (50) = -2
RL Case required at node 50
12 Step
Performing RR Case
Balance factor for every node is inside the range of -1 to +1
(Final AVL tree)
11 Step
Performing LL Case
Balance factor (50) = -2
RR Case required at node 50

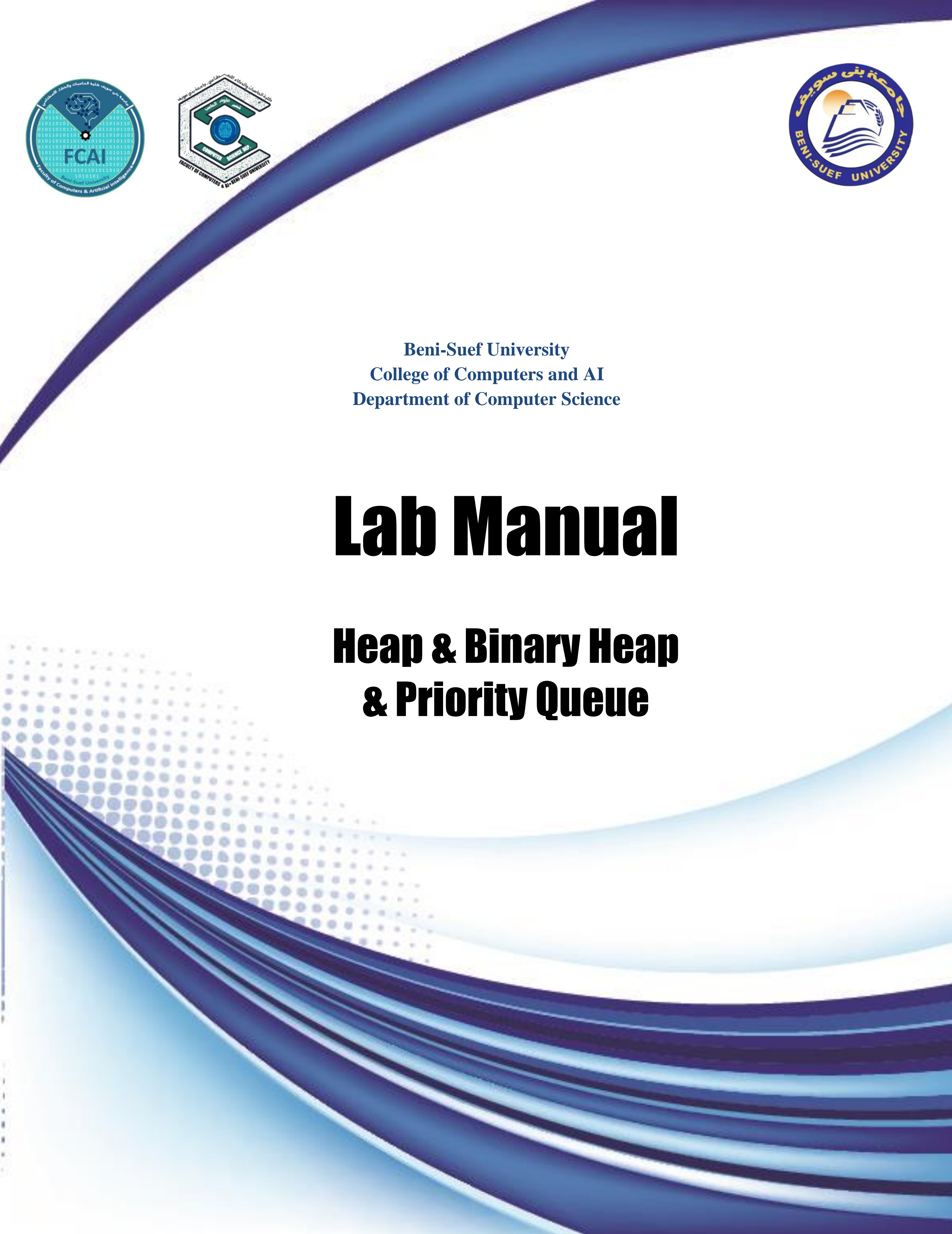
3. Lab

Exercise 1: insert a node in AVL tree:

```
#include <bits/stdc++.h>
using namespace std;
// An AVL tree node
struct Node {
    int key;
    Node* left;
    Node* right;
    int height;
    Node(int k) {
        key = k;
        left = nullptr;
        right = nullptr;
        height = 1;
    }
};
// Get height of node
int height(Node* N) {
    if (N == nullptr)
        return 0;
    return N->height;
}
// Right rotation
Node* rightRotate(Node* y) {
    Node* x = y->left;
    Node* T2 = x->right;
    // Perform rotation
    x->right = y;
    y->left = T2;
    // Update heights
    y->height = 1 + max(height(y->left), height(y->right));
    x->height = 1 + max(height(x->left), height(x->right));
    return x;
}
// Left rotation
Node* leftRotate(Node* x) {
    Node* y = x->right;
    Node* T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
```

```cpp
    x->height = 1 + max(height(x->left), height(x->right));
    y->height = 1 + max(height(y->left), height(y->right));

    return y;
}

// Get balance factor
int getBalance(Node* N) {
    if (N == nullptr)
        return 0;
    return height(N->left) - height(N->right);
}

// Insert into AVL tree
Node* insert(Node* node, int key) {

    // Normal BST insertion
    if (node == nullptr)
        return new Node(key);

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else
        return node; // No duplicates

    // Update height
    node->height = 1 + max(height(node->left),
                           height(node->right));

    // Check balance
    int balance = getBalance(node);

    // LL Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // RR Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // LR Case
    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
```

```
    // RL Case
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}

// Preorder traversal
void preOrder(Node* root) {
    if (root != nullptr) {
        cout << root->key << " ";
        preOrder(root->left);
        preOrder(root->right);
    }
}

// Driver code
int main() {
    Node* root = nullptr;

    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    /*
        Final AVL Tree:
              30
            /    \
          20      40
         /  \       \
       10   25      50
    */

    cout << "Preorder traversal: ";
    preOrder(root);

    return 0;
}
```



Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

Heap & Binary Heap & Priority Queue

Lab - 8

Heap & Binary Heap & Priority Queue

1. An overview

This lab introduces two important concepts in computer science: the **Heap** data structure and the **Priority Queue**. By the end of this session, you will understand what a complete binary tree is, the heap property (min-heap and max-heap), how to implement basic heap operations, and how to use priority queues for scheduling problems. You will also implement the efficient Heap Sort algorithm.

Learning Objectives

After completing this lab, you should be able to:

- **Define** what a complete binary tree is and its properties.
- **Differentiate** between perfect binary trees and complete binary trees.
- **Calculate** parent, left child, and right child indices in array representation.
- **Define** the heap property for Min-Heap and Max-Heap.
- **Implement** basic heap operations: insert, extractMin/Max, deleteKey, decreaseKey.
- **Understand** the heapify process (bubble-up and bubble-down).
- **Implement** Heap Sort algorithm with O(n log n) time complexity.
- **Apply** heaps to solve priority queue problems.

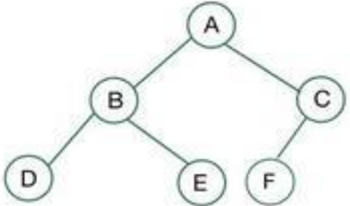
2. Theory

Binary tree

We know a tree is a non-linear data structure. It has no limitation on the number of children. A binary tree has a limitation as any node of the tree has at most two children: a left and a right child.

What is a Complete Binary Tree?

A complete binary tree is a special type of binary tree where all the levels of the tree are filled completely except the **lowest** level nodes which are filled from as left as possible.



Some terminology of Complete Binary Tree:

- **Root** – Node in which no edge is coming from the parent. Example -node A
- **Child** – Node having some incoming edge is called child. Example – nodes B, F are the child of A and C respectively.
- **Sibling** – Nodes having the same parent are sibling. Example- D, E are siblings as they have the same parent B.
- **Degree of a node** – Number of children of a particular parent. Example- Degree of A is 2 and Degree of C is 1. Degree of D is 0.
- **Internal/External nodes** – Leaf nodes are external nodes and non leaf nodes are internal nodes.
- **Level** – Count nodes in a path to reach a destination node. Example- Level of node D is 2 as nodes A and B form the path.
- **Height** – Number of edges to reach the destination node, Root is at height 0. Example – Height of node E is 2 as it has two edges from the root.

Difference between depth and height :

The **depth** of a node is the **number of edges from the root to that node**.
The **depth of the root node is always 0**.
Example:

```
     1        (d=0)
    / \
   2   3      (d=1)
  / \    /
 4   5 6      (d=2)
```

- Depth(1) = 0
- Depth(2) = 1
- Depth(4) = 2

The **height** of a node is the **longest path from that node to a leaf**.
The **height of the tree** is the **height of the root node**.
Leaf nodes always have height = 0.
Example:

```
     1       (h=2)
    / \
   2     3     (h=1)
  / \     /
 4  5   6      (h=0)
```

- Height(1) = 2
- Height(2) = 1
- Height(4) = 0

Properties of Complete Binary Tree:

- A complete binary tree is said to be a proper binary tree where all leaves have the same depth.
- In a complete binary tree number of nodes at depth **d** is $\mathbf{2^d}$

$$2^h \leq N \leq 2^{h+1} - 1$$

- In a complete binary tree with **n** nodes height of the tree is **log(n+1)**.
- All the levels **except the last level** are completely full.
- A **Complete Binary Tree (CBT)** is a binary tree where all levels are completely filled **except possibly the last one**, which is filled from **left to right**.

- **Example of a Complete Binary Tree**:

```
      1
     /  \
    2    3
   / \   /
  4   5 6
```

- **Not a Complete Binary Tree** (node 6 appears after an empty right spot of node 3):

```
      1
     /  \
    2    3
   / \     \
  4   5     6
```

- Indexing in an Array Representation A CBT can be stored efficiently in an array without pointers.For a node at index i:
    - Parent node index = (i - 1) / 2
    - Left child index = 2 * i + 1
    - Right child index = 2 * i + 2

Example (Array Representation of CBT) For [1, 2, 3, 4, 5, 6]:

Index: 0 1 2 3 4 5
Value: 1 2 3 4 5 6

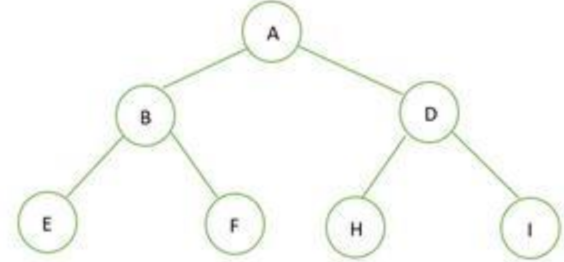
- Parent of 2 (index 1): (1-1)/2 = 0 → 1
- Left child of 2 (index 1): 2*1+1 = 3 → 4
- Right child of 2 (index 1): 2*1+2 = 4 → 5

Perfect Binary Tree vs Complete Binary Tree:

A binary tree of height ‘h’ having the maximum number of nodes is **a perfect binary tree**. For a given height h, the maximum number of nodes is 2^{h+1}-1.

A complete binary tree of height h is a perfect binary tree up to height h-1, and in the last level element are stored in left to right order.

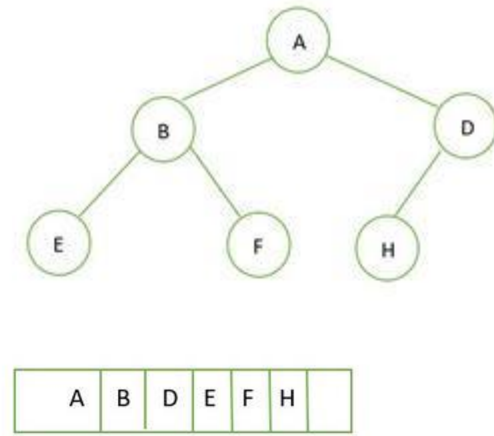
Example 1:
Height= 2 , max number of nodes= 7 , so it is a perfect binary tree



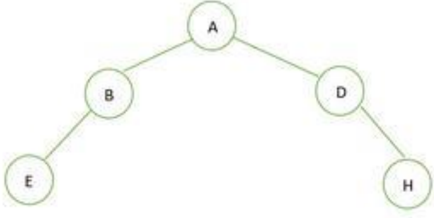
It is also a **CBT, which is presented as :-**

A	B	D	E	F	H	I

Example 2:-
Height= 2 , max number of nodes= 6, so it is not a perfect binary tree. Hence this is a **complete binary tree**



Example 3:
The height of the binary tree is 2 and the maximum number of nodes that can be there is 7, but there are only 5 nodes hence it is **not a perfect binary tree**. In case of a complete binary tree, we see that in the last level elements are not filled from left to right order. So it is **not a complete binary tree**.



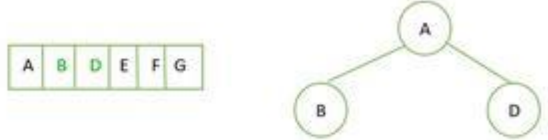
<u>Creation of Complete Binary Tree:</u>

Consider the below array:

1. The 1st element will the root (value at index = 0)

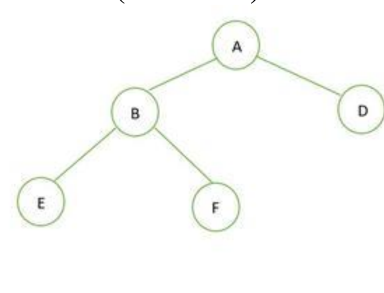
A is taken as root

2. The next element (at index = 1) will be left and third element (index = 2) will be right child of root



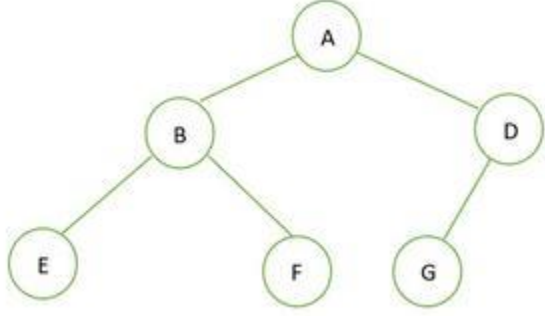
B as left child and D as right child

3. fourth (index = 3) and fifth element (index = 4) will be the left and right child of B node



E and F are left and right child of B

4. Next element (index = **5**) will be left child of the node D



G is made left child of D node.

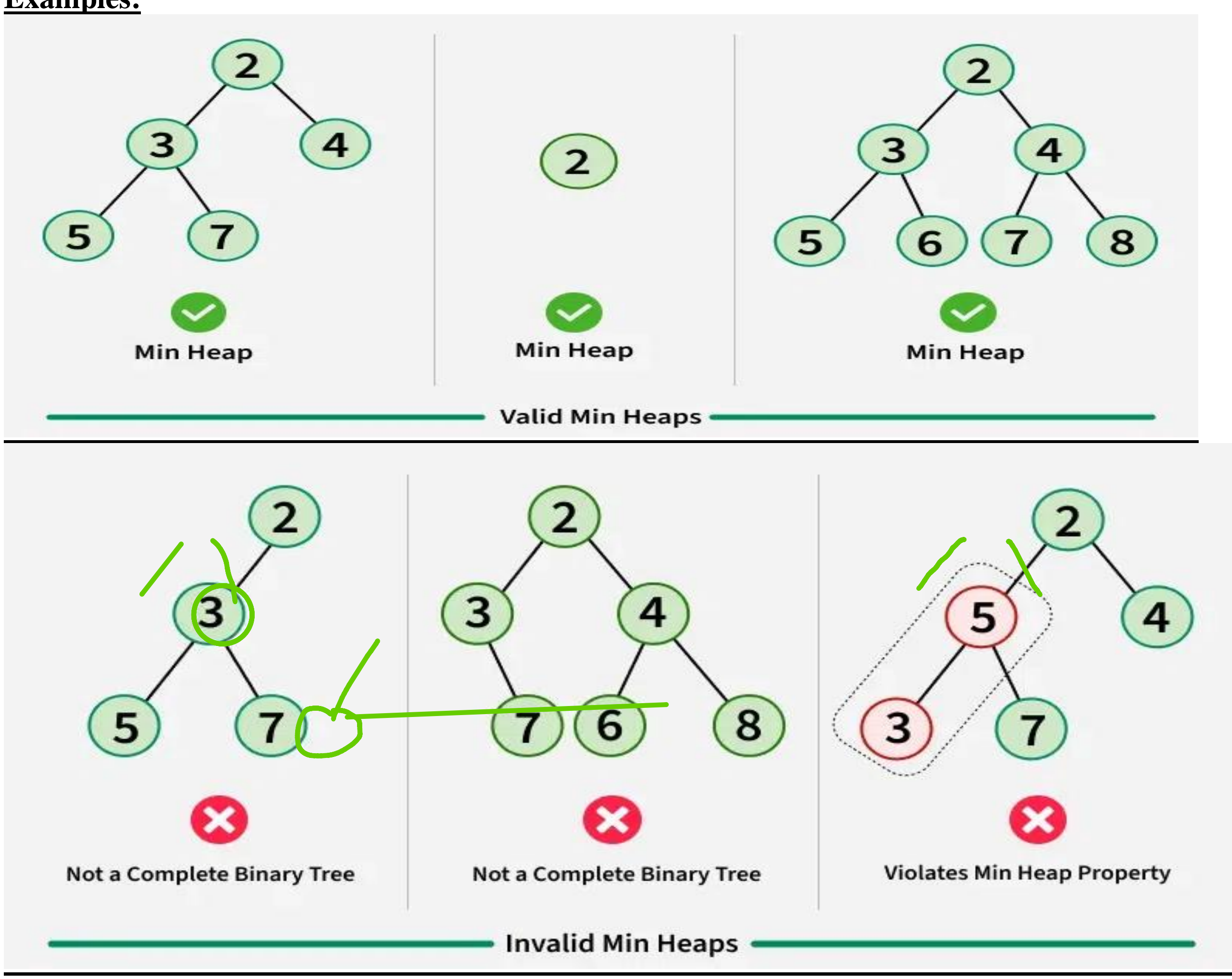
Binary heap

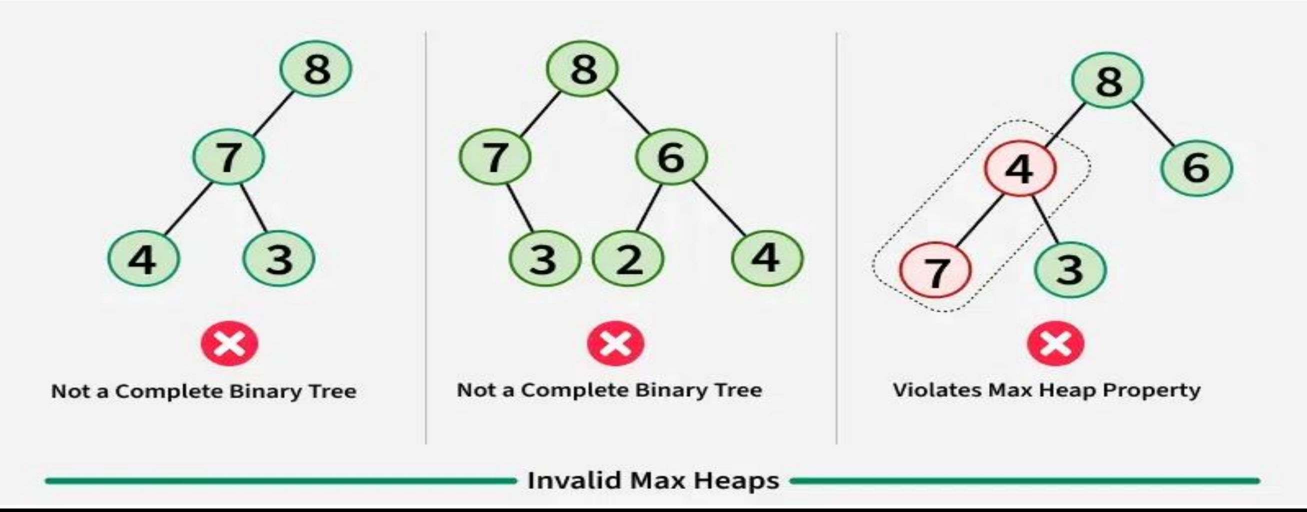
A **Heap** is a complete binary tree data structure that satisfies the heap property: in a min-heap, the value of each child is greater than or equal to its parent, and in a max-heap, the value of each child is less than or equal to its parent. Heaps are commonly used to implement priority queues, where the smallest (or largest) element is always at the root.

A Binary Heap is a **complete Binary Tree** which is used to store data efficiently to get the **max** or **min** element based on its type. A Binary Heap is either **Min Heap** or **Max Heap**.

In a **Min Binary Heap**, the key at the root must be **minimum** among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to Min Heap.

Examples:





How is Binary Heap represented?

A Binary Heap is a Complete Binary Tree. A binary heap is typically represented as an array.

- The root element will be at arr[0].
- The below table shows indices of other nodes for the ith node, i.e., arr[i]:

arr[(i-1)/2]	Returns the parent node
arr[(2*i)+1]	Returns the left child node
arr[(2*i)+2]	Returns the right child node

Priority Q

In C++, priority queue is a type of queue in which there is some **priority** assigned to the elements. According to this priority, elements are removed from the queue. By default, the value of the element being inserted is considered as priority. Higher its value, higher its priority. But this can be changed to any desired priority scheme as per requirement.
Priority queue is defined as
std::priority_queue inside **<queue>** header file.
priority_queue<*T, c, comp*> *pq;*
where,

- **T:** Type of the priority queue
- **pq:** Name assigned to the priority queue.
- **c**: Underlying container. Uses vector as default.
- **comp**: It is a binary predicate function that tells priority queue how to compare two elements. It is used to set the custom priority parameter and scheme. It is optional and if not provided, maximum value gets maximum priority.

All Member Functions

Following is the list of all member functions of **std::priority_queue** class in C++:

Function	Definition
empty()	Returns whether the priority queue is empty.
size()	Returns the size of the priority queue.
top()	Returns top element of the priority queue
push()	Adds the element into the priority queue.
pop()	Deletes the top element of the priority queue.
swap()	Used to swap the contents of two priority queues provided the queues must be of the same type, although sizes may differ.

3. Lab

Exercise 1: Complete Binary Tree – Count Nodes

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

class TreeNode {
public:
    int val;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int value) {
        val = value;
        left = nullptr;
        right = nullptr;
    }
};

int countNodes(TreeNode* root) {
    if (root == nullptr)
        return 0;

    return 1 + countNodes(root->left) + countNodes(root->right);
}

int main() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    root->right->left = new TreeNode(6);

    cout << "Number of nodes: " << countNodes(root) << endl;

    return 0;
}
```

Exercise 2: Iemplementation of basic heap operations

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

// Swap function
void swap(int* x, int* y);

// =========================
// MinHeap Class
// =========================
class MinHeap {
    int* harr;
    int capacity;
    int heap_size;

public:
    MinHeap(int capacity);

    void MinHeapify(int i);

    int parent(int i) { return (i - 1) / 2; }
    int left(int i) { return (2 * i + 1); }
    int right(int i) { return (2 * i + 2); }

    int extractMin();
    void decreaseKey(int i, int new_val);
    int getMin() { return harr[0]; }
    void deleteKey(int i);
    void insertKey(int k);
};

// =========================
// Constructor
// =========================
MinHeap::MinHeap(int cap) {
    heap_size = 0;
    capacity = cap;
    harr = new int[cap];
}

// =========================
// Insert
// =========================
void MinHeap::insertKey(int k) {
    if (heap_size == capacity) {
```

```
        cout << "\nOverflow\n";
        return;
    }

    heap_size++;
    int i = heap_size - 1;
    harr[i] = k;

    while (i != 0 && harr[parent(i)] > harr[i]) {
        swap(&harr[i], &harr[parent(i)]);
        i = parent(i);
    }
}

// ========================
// Decrease Key
// ========================
void MinHeap::decreaseKey(int i, int new_val) {
    harr[i] = new_val;

    while (i != 0 && harr[parent(i)] > harr[i]) {
        swap(&harr[i], &harr[parent(i)]);
        i = parent(i);
    }
}

// ========================
// Extract Min
// ========================
int MinHeap::extractMin() {
    if (heap_size <= 0)
        return INT_MAX;

    if (heap_size == 1) {
        heap_size--;
        return harr[0];
    }

    int root = harr[0];
    harr[0] = harr[heap_size - 1];
    heap_size--;

    MinHeapify(0);

    return root;
}
```

```
// ==========================
// Delete Key
// ==========================
void MinHeap::deleteKey(int i) {
    decreaseKey(i, INT_MIN);
    extractMin();
}

// ==========================
// Heapify
// ==========================
void MinHeap::MinHeapify(int i) {
    int l = left(i);
    int r = right(i);
    int smallest = i;

    if (l < heap_size && harr[l] < harr[smallest])
        smallest = l;

    if (r < heap_size && harr[r] < harr[smallest])
        smallest = r;

    if (smallest != i) {
        swap(&harr[i], &harr[smallest]);
        MinHeapify(smallest);
    }
}

// ==========================
// Swap Function
// ==========================
void swap(int* x, int* y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

// ==========================
// Heap Sort
// ==========================
void heapSort(int arr[], int n) {
    MinHeap h(n);

    for (int i = 0; i < n; i++)
        h.insertKey(arr[i]);

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

```cpp
        arr[i] = h.extractMin();
}

// ========================
// Main
// ========================
int main() {
    MinHeap h(11);

    h.insertKey(3);
    h.insertKey(2);
    h.deleteKey(1);
    h.insertKey(15);
    h.insertKey(5);
    h.insertKey(4);
    h.insertKey(45);

    cout << h.extractMin() << " ";
    cout << h.getMin() << " ";

    h.decreaseKey(2, 1);
    cout << h.getMin();

    // Heap Sort Test
    int arr[] = {12, 3, 5, 7, 19, 1};
    int n = sizeof(arr) / sizeof(arr[0]);

    heapSort(arr, n);

    cout << "\nSorted array: ";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";

    return 0;
}
```

Exercise 3: Iemplementation Priority Queue

```cpp
#include <iostream>
#include <queue>
#include <map>
#include <vector>
#include <string>
using namespace std;

int main() {
    // Example input data
    vector<pair<string, int>> jobs = {
        { "ABC", 2 }, { "XYZ", 1 }, { "PQR", 1 }, { "RTZ", 3 },
        { "CBZ", 2 }, { "QQQ", 3 }, { "XXX", 4 }, { "RRR", 1 }
    };
    // Map of priority -> queue of job names
    map<int, queue<string>> priorityQueues;
    // Enqueue jobs into the appropriate priority queues
    for (const auto& job : jobs) {
        string name = job.first;
        int priority = job.second;
        priorityQueues[priority].push(name);
    }



    // Display the jobs grouped by priority (1 = highest)
    cout << "Jobs organized by priority:\n";
    for (int p = 1; p <= 4; ++p) {
        if (priorityQueues[p].empty()) continue;

        cout << "Priority " << p << ": ";
        while (!priorityQueues[p].empty()) {
            cout << priorityQueues[p].front() << " ";
            priorityQueues[p].pop();
        }
        cout << endl;
    }

    return 0;
}
```