GH raisoni COLLEGE
Engineering and Management Pune

G H Raisoni College of Engineering & Management
An Empowered Autonomous Institute, Affiliated to Savitribai Phule Pune University
Approved by AICTE, New Delhi and Recognized by Govt. of Maharashtra, NAAC Accredited "A+" Grade
Navin Gat No.1200, Domkhel Road, Wagholi, Pune – 412 207 (India)
T: +91 - 9604787185/186 | E: ghrcem.pune@raisoni.net | W: www.ghrcemp.raisoni.net

Department of Computer Engineering

D-19

Lab Manual (2024-25)

Pattern-2023

Class: SY Computer Term: III

Data Structure and Algorithms II
23UCOPCP2407

G H Raisoni College of Engineering and Management, Wagholi, Pune 412207
Department: Computer Engineering

Course Details

Subject: Data Structure and Algorithms (23UCOPCP2407)

CLASS: SY BTECH **DIVISION: B**
External Marks: 25

COURSE OUTCOME	
CO1	Identify file handling methods and apply them to optimize data access and retrieval.
CO2	Describe advanced data structure such as AVL trees, B-trees, Heaps and their applications to solve real-world problems.
CO3	Apply searching and sorting algorithms on tree data structure.
CO4	Implement and analyze the efficiency of graph algorithms for different graph representations.

List of Experiments

Sr. No	List of Laboratory Assignments	Relevance to CO	Software Required
1.	Study of different types of Network cables anWrite a sorting program with the following features: Write a sorting program with the following features: a) Reads data from a text file and sorts it alphabetically by default. b) If the file has data in rows and columns (separated by space or tab) then allows sorting on a particular column.	CO1	Dev C++, Code Block

	c) implement any sort using numeric or alphabetical ordering and compare stable and unstable sort (VLAB:https://ds2-iiith.vlabs.ac.in/exp/radix-sort/analysis/stability-of-selection-sort.html).		
2	Develop C program to build a) MAX_HEAP b) MIN_HEAP c) Implement heap sort using MAX_HEAP (VLAB: https://ds1-iiith.vlabs.ac.in/exp/heap-sort/heap-sort/index.html)	CO2	Dev C++, Code Block
3	Study of different types of Network cables and Practically implement the cross-wired cable and straight through cConstruct binary tree by inserting the values in the order given. After constructing a binary Construct Binary tree perform following traversing using recursive and non-recursive functions. a) Preorder b) Postorder c) Inorder	CO2	Dev C++, Code Block
4.	Implement AVL trees with following operations a) Create b) Insert c) Search d) Delete	CO2, CO3	Dev C++, Code Block
5	Write a program to perform the following operations: a) Insertion into a B-tree b) Searching in a B-tree	CO2, CO3	Dev C++, Code Block
6	Develop C function to implement graph using a) Adjacency lists b) Adjacency matrices c) Demonstrate Djkstra's algorithm on a weighted graph (VLAB: https://ds2-iiith.vlabs.ac.in/exp/dijkstra-algorithm/dijkstras-algorithm/index.html)	CO4	Dev C++, Code Block
7	The given directed graph contains N nodes and M edges. Each edge connects two nodes, Write a program to find the sequence of nodes in the order they are visited using a) DFS	CO4	Dev C++, Code Block

	b) BFS c) Graph traversal concepts understanding (VLAB: https://ds1-iiith.vlabs.ac.in/exp/depth-first-search/graph-traversals.html).		
8	Write a program to read a map stored in a text file with (city1, city2, distance) comma separated values. a) Build a graph using this data. b) Print all pair shortest paths information between all pairs of cities (using Warshall's Algorithm).	CO1, CO4	Dev C++, Code Block
9	Write a program to read a map stored in a text file with (city1, city2, distance) comma separated values. Create Minimum Spanning Tree for this graph using a) Prim's Algorithm b) Kruskal's Algorithm Compare Kruskal's and Prim's with other algorithms to find MST (VLAB : https://ds2-iiith.vlabs.ac.in/exp/min-spanning-trees/index.html)	CO1, CO4	Dev C++, Code Block
10	Given a directed acyclic graph of numbers, generate a topological ordering of numbers using a) DFS method b) Kahn's algorithm c) Analyze the time complexity of both methods by counting the number of operations involved in each iteration (VLAB: https://ds2-iiith.vlabs.ac.in/exp/topo-sort/analysis/time-and-space-complexity.html)	CO4	Dev C++, Code Block
Content Beyond Syllabus			
11	Implement Djkstra's algorithm using Fibonacci Heap.	CO4	Dev C++, Code Block
12	Write a program to implement Splay trees.	CO3	Dev C++, Code Block

`

Experiment No. 1

Aim: Write a sorting program with the following features:

a) Reads data from a text file and sorts it alphabetically by default.
b) If the file has data in rows and columns (separated by space or tab) then allows sorting on a particular column.
c) Implement any sort using numeric or alphabetical ordering and compare stable and unstable sort. (VLAB:https://ds2-iiith.vlabs.ac.in/exp/radix-sort/analysis/stability-of-selection-sort.html).

Theory:
Sorting is a fundamental operation in computer science that involves rearranging the elements of an array or list in a specific order, typically ascending or descending. Sorting algorithms are essential because they optimize the performance of other algorithms that require sorted data, such as search algorithms and database operations. Sorting refers to rearrangement of a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements. in the respective data structure.

Stable and Unstable Sorting Algorithms

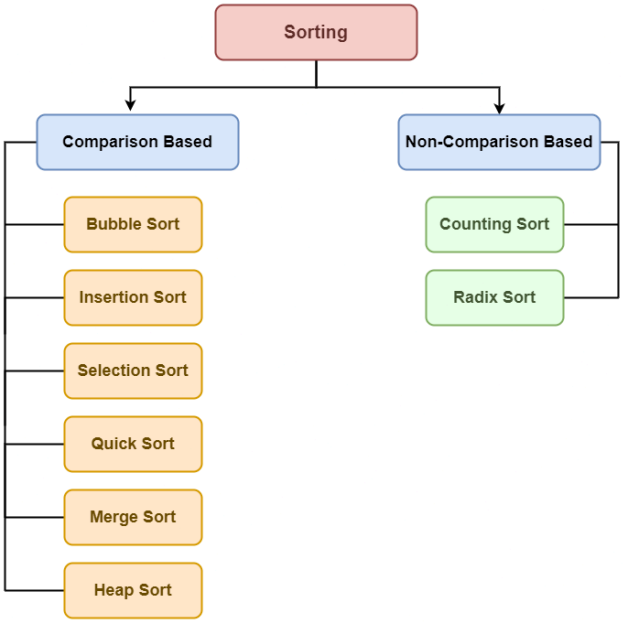
Sorting algorithms can be classified into two categories: stable and unstable. The stability of a sorting algorithm is determined by how it handles equal elements.

Stable Sorting Algorithms

A sorting algorithm is considered stable if it preserves the relative order of equal elements in the sorted output as they appear in the input. This means that if two elements are equal, their order will remain the same after sorting. Bubble Sort, Insertion Sort, Merge Sort, Counting Sort are examples of Stable Sorting.

Unstable Sorting Algorithms

An **unstable** sorting algorithm does not guarantee the preservation of the relative order of equal elements. This means that the order of equal elements may change after sorting. Quick Sort, Heap Sort, Selection Sort are the examples of Unstable Sorting Algorithms.



File handling

File handling is used to store data permanently in a computer. It is used to store data in secondary

memory (Hard disk).
For achieving file handling we need to follow the following steps:-
STEP 1-Naming a file
STEP 2-Opening a file
STEP 3-Writing data into the file
STEP 4-Reading data from the file
STEP 5-Closing a file.

Functions for file Handling

There are many functions in the C library to open, read, write, search and close the file. Lists of file functions are given below:

- fopen() opens new or existing file
- fprintf() write data into the file
- fscanf() reads data from the file
- fputc() writes a character into the file
- fgetc() reads a character from file
- fclose() closes the file

C Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_ROWS 100
#define MAX_COLS 10
#define MAX_LEN 100

// Function prototypes
void readDataFromFile(const char *filename, char data[MAX_ROWS][MAX_COLS][MAX_LEN], int *rows, int *cols);
void sortRows(char data[MAX_ROWS][MAX_COLS][MAX_LEN], int rows, int cols, int colToSort, int numeric);
void printData(char data[MAX_ROWS][MAX_COLS][MAX_LEN], int rows, int cols);
int compareStrings(const char *a, const char *b, int numeric);

int main() {
    char data[MAX_ROWS][MAX_COLS][MAX_LEN];
```

```
    int rows = 0, cols = 0, colToSort;
    char filename[50];
    int numeric;

    // Input the filename
    printf("Enter the file name: ");
    scanf("%s", filename);

    // Read data from file
    readDataFromFile(filename, data, &rows, &cols);

    // Print original data
    printf("\nOriginal Data:\n");
    printData(data, rows, cols);

    // Input column to sort and order type
    printf("\nEnter the column number to sort (0-based index): ");
    scanf("%d", &colToSort);
    printf("Sort numerically? (1 for Yes, 0 for No): ");
    scanf("%d", &numeric);

    // Sort the data
    sortRows(data, rows, cols, colToSort, numeric);

    // Print sorted data
    printf("\nSorted Data:\n");
    printData(data, rows, cols);

    return 0;
}

// Function to read data from a file
void readDataFromFile(const char *filename, char data[MAX_ROWS][MAX_COLS][MAX_LEN], int
*rows, int *cols) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        printf("Error: Unable to open file %s\n", filename);
        exit(1);
    }

    char line[MAX_LEN];
    *rows = 0;
```

```
    while (fgets(line, sizeof(line), file) && *rows < MAX_ROWS) {
        char *token = strtok(line, " \t\n");
        *cols = 0;

        while (token && *cols < MAX_COLS) {
            strcpy(data[*rows][*cols], token);
            token = strtok(NULL, " \t\n");
            (*cols)++;
        }
        (*rows)++;
    }

    fclose(file);
}

// Function to sort rows based on a specific column
void sortRows(char data[MAX_ROWS][MAX_COLS][MAX_LEN], int rows, int cols, int colToSort, int
numeric) {
    for (int i = 0; i < rows - 1; i++) {
        for (int j = 0; j < rows - i - 1; j++) {
            if (compareStrings(data[j][colToSort], data[j + 1][colToSort], numeric) > 0) {
                // Swap entire rows
                for (int k = 0; k < cols; k++) {
                    char temp[MAX_LEN];
                    strcpy(temp, data[j][k]);
                    strcpy(data[j][k], data[j + 1][k]);
                    strcpy(data[j + 1][k], temp);
                }
            }
        }
    }
}

// Function to compare strings (numerically or alphabetically)
int compareStrings(const char *a, const char *b, int numeric) {
    if (numeric) {
        return atoi(a) - atoi(b);
    }
    return strcmp(a, b);
}
```

```
// Function to print the data
void printData(char data[MAX_ROWS][MAX_COLS][MAX_LEN], int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%s\t", data[i][j]);
        }
        printf("\n");
    }
}
```

Output:

```
Enter the file name: data.txt

Original Data:
Alice   90
Bob     75
Charlie 85
David   80

Enter the column number to sort (0-based index): 0
Sort numerically? (1 for Yes, 0 for No): 0

Sorted Data:
Alice   90
Bob     75
Charlie 85
David   80

--------------------------------
Process exited after 16.32 seconds with return value 0
Press any key to continue . . .
```

```
Enter the file name: data.txt

Original Data:
Alice   85      90
Bob     70      75
Charlie 95      85
David   65      80

Enter the column number to sort (0-based index): 1
Sort numerically? (1 for Yes, 0 for No): 1

Sorted Data:
David   65      80
Bob     70      75
Alice   85      90
Charlie 95      85

--------------------------------
Process exited after 21.62 seconds with return value 0
Press any key to continue . . .
```

Conclusions:
We learned the fundamental concept of file I/O and implemented various file handling methods. The file handing concept has been used to sort a text file alphabetically. Additionally, we studied the concept of stable and unstable sorting

Experiment 1 viva Question

1. How would you modify your sorting program to handle large datasets?
2. How would you decide which sorting algorithm to use in a given situation?
3. What is the difference between a stable and unstable sorting algorithm?
4. How do you check for errors while working with files in C?
5. How do you read from a file in C++?
6. Can we directly read and write binary files in C++?
7. What is the role of the `flush()` function in C++ file handling?
8. How do you write to a file in C++?

9. Explain the `fopen()` function in C.
10. What are the different file modes in C?

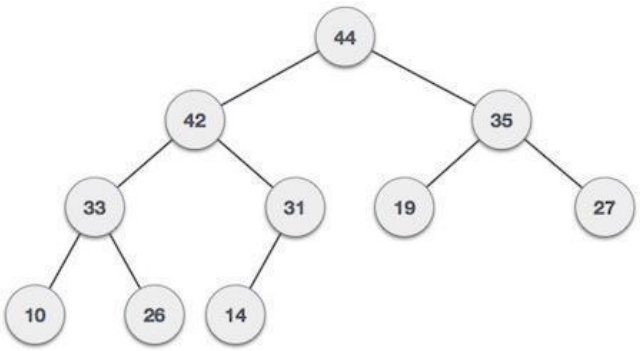
Experiment No. 2

Aim:
Develop C program to build
a) MAX_HEAP
b) MIN_HEAP
c) Implement heap sort using MAX_HEAP (VLAB: https://ds1-iiith.vlabs.ac.in/exp/heap-sort/heap-sort/index.html)

Theory:
A Heap is Tree based data structure in which the tree is a complete binary tree. Since a heap is a complete binary tree, a heap with N nodes has log N height. It is useful to remove the highest or lowest priority element. It is typically represented as an array. There are two types of Heaps in the data structure.

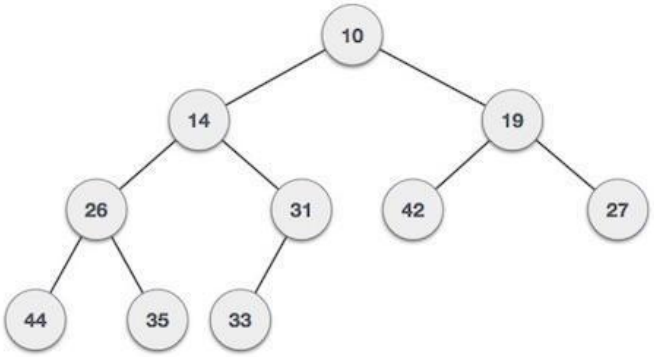
Max-Heap:

In a Max-Heap the key present at the root node must be greater than or equal among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree. In a Max-Heap the maximum key element present at the root. Below is the Binary Tree that satisfies all the property of Max Heap.



Min-Heap:

In a Min-Heap the key present at the root node must be less than or equal among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree. In a Min-Heap the minimum key element present at the root. Below is the Binary Tree that satisfies all the property of Min Heap.



MaxHeapify algorithm

MaxHeapify algorithm can be used to transform the current complete binary tree to a max heap. The basic idea is to start the process from the root. Here, the key value of the root is swapped with the child with the maximum key value. The process is repeated with the node involved in the swapping of keys unless the said node is already a leaf node.

```
1.  MaxHeapify(arr,i)
2.  L = left(i)
3.  R = right(i)
4.  if L ? heap_size[arr] and arr[L] > arr[i]
5.  largest = L
6.  else
7.  largest = i
8.  if R ? heap_size[arr] and arr[R] > arr[largest]
9.  largest = R
10. if largest != i
11. swap arr[i] with arr[largest]
12. MaxHeapify(arr,largest)
13. End
```

Build-Max-Heap Algorithm

Any given array A can be transformed to a max heap by repeatedly using the Max-Heapify algorithm. Let's call this algorithm as the Build-Max-Heap algorithm. The implementation uses the Max-Heapify algorithm starting from the last node with at least one child up to the root node.

```
BuildMaxHeap(arr)
   heap_size(arr) = length(arr)
   for i = length(arr)/2 to 1
MaxHeapify(arr,i)
End
```

Heapsort:

Heapsort is a popular and efficient sorting algorithm. The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list.
Once the heap is built, the root element (the largest element in a Max-Heap) is swapped with the last element of the array. This places the maximum element in its correct sorted position.
After the swap, the heap is reduced by one, and heapify is called on the new root to restore the heap property. This swap and heapify process continues until all elements are moved to their correct positions, and the array becomes sorted.

```
HeapSort(arr)
BuildMaxHeap(arr)
for i = length(arr) to 2
   swap arr[1] with arr[i]
      heap_size[arr] = heap_size[arr] ? 1
      MaxHeapify(arr,1)
End
```

C Code

```
#include <stdio.h>
#include <stdlib.h>
void swap(int *a, int *b)
{
   int tmp = *a;
     *a = *b;
     *b = tmp;
}
 void heapify(int arr[], int N, int i)
{
   int largest = i;
```

```
    int l = 2 * i + 1;
    int r = 2 * i + 2;

    // If left child is larger than root
    if (l < N && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest so far
    if (r < N && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i) {
        swap(&arr[i], &arr[largest]);

        // Recursively heapify the affected sub-tree
        heapify(arr, N, largest);
    }
}

// Function to build a Max-Heap from the given array
void buildHeap(int arr[], int N)
{
    // Index of last non-leaf node
    int startIdx = (N / 2) - 1;

      for (int i = startIdx; i >= 0; i--) {
        heapify(arr, N, i);
    }
}

void printHeap(int arr[], int N)
{
    printf("Array representation of Heap is:\n");

    for (int i = 0; i < N; ++i)
        printf("%d ",arr[i]);
    printf("\n");
}
 // HeapSorting function
void heapsort(int arr[], int n)
{
```

```
    int i, temp;

     for (i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }
    // the current array is changed to max heap

    for (i = n - 1; i > 0; i--) {
        temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;
        heapify(arr, i, 0);
    }
}
int main()
{
     //  int arr[] = {1, 3, 5, 4, 6, 13, 10, 9, 8, 15, 17};
       int arr[20], N, i;
       printf("enter array size");
       scanf("%d",&N);
        printf("enter array elements");
       for(i=0;i<N; i++)
      scanf("%d",&arr[i]);
       buildHeap(arr, N);
    printHeap(arr, N);
     heapsort(arr,N);
      printf("Array after performing heap sort: ");
       printHeap(arr, N);

    return 0;
}
```

Output

```
enter array size: 7
enter array elements: 2
4
1
7
11
23
6
Array representation of MAX_HEAP is:
23 11 6 7 4 1 2
Array after performing heap sort: 1 2 4 6 7 11 23
```

Conclusion:

This program implemented MAX heap using Heapify algorithm. Therafter, Heapify algorithm and BuildHeap algorithm are used to perform heap sort on a given array.

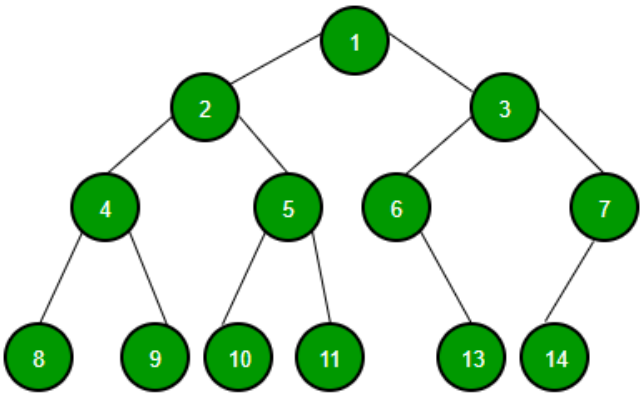
Experiment 2 viva Question

1. Explain the process of heapifying a subtree.
2. What is the space complexity of heap sort?
3. Is heap sort a stable sorting algorithm?
4. How does heap sort compare to merge sort or quicksort in terms of performance?
5. Explain why the time complexity of building a heap is O(n).
6. What is the time complexity of heap sort?
7. Why is heap sort an in-place algorithm?
8. What is the time complexity for deleting the root element (or extracting the maximum/minimum) from a heap?
9. What is the difference between a Min-Heap and a Max-Heap?
10. What is a "heap," and how is it used in Heap Sort?

Experiment 3

Aim:
Construct binary tree by inserting the values in the order given. After constructing a binary tree perform following traversing using recursive and non-recursive functions.
a) Preorder
b) Postorder
c) Inorder

Binary Tree:
In general, tree nodes can have any number of children. In a binary tree, each node can have at most two children. A binary tree is either empty or consists of a node called the root together with two binary trees called the left subtree and the right subtree. Example of Binary Tree is shown below.



Binary Tree Traversal Techniques:
A tree traversal is a method of visiting every node in the tree. By visit, we mean that some type of operation is performed. For example, you may wish to print the contents
of the nodes.
There are four common ways to traverse a binary tree:
1. Preorder
2. Inorder
3. Postorder

Inorder Traversal:

In the case of inorder traversal, the root of each subtree is visited after its left subtree has been traversed but before the traversal of its right subtree begins. The steps for traversing a binary tree in inorder traversal are:

1. Visit the left subtree, using inorder.
2. Visit the root.
3. Visit the right subtree, using inorder.

The algorithm for inorder traversal is as follows:

```
void inorder(node *root)
{
if(root != NULL)
{
inorder(root->lchild);
print root -> data;
inorder(root->rchild);
}
}
```

Preorder Traversal:

In a preorder traversal, each root node is visited before its left and right subtrees are traversed. Preorder search is also called backtracking. The steps for traversing a binary tree in preorder traversal are:

1. Visit the root.
2. Visit the left subtree, using preorder.
3. Visit the right subtree, using preorder.

The algorithm for preorder traversal is as follows:

```
void preorder(node *root)
{
if( root != NULL )
{
print root -> data;
preorder (root -> lchild);
preorder (root -> rchild);
}
}
```

Postorder Traversal:

In a postorder traversal, each root is visited after its left and right subtrees have been

traversed. The steps for traversing a binary tree in postorder traversal are:
1. Visit the left subtree, using postorder.
2. Visit the right subtree, using postorder
3. Visit the root.
The algorithm for postorder traversal is as follows:

```
void postorder(node *root)
{
if( root != NULL )
{
postorder (root -> lchild);
postorder (root -> rchild);
print (root -> data);
}
}
```

Code:

```
#include <stdio.h>
#include <stdlib.h>

// Definition of a node in the binary tree
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Insert node into binary tree
struct Node* insertNode(struct Node* root, int data) {
```

```
    if (root == NULL) {
        return createNode(data);
    }
    if (data < root->data) {
        root->left = insertNode(root->left, data);
    } else {
        root->right = insertNode(root->right, data);
    }
    return root;
}

// Recursive traversals
void preorderRecursive(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorderRecursive(root->left);
        preorderRecursive(root->right);
    }
}

void inorderRecursive(struct Node* root) {
    if (root != NULL) {
        inorderRecursive(root->left);
        printf("%d ", root->data);
        inorderRecursive(root->right);
    }
}

void postorderRecursive(struct Node* root) {
    if (root != NULL) {
        postorderRecursive(root->left);
        postorderRecursive(root->right);
        printf("%d ", root->data);
    }
}

// Non-recursive traversals
```

```
void preorderNonRecursive(struct Node* root) {
    struct Node* stack[100];
    int top = -1;
    struct Node* current = root;

    while (current != NULL || top >= 0) {
        while (current != NULL) {
            printf("%d ", current->data);
            stack[++top] = current;
            current = current->left;
        }
        current = stack[top--];
        current = current->right;
    }
}

void inorderNonRecursive(struct Node* root) {
    struct Node* stack[100];
    int top = -1;
    struct Node* current = root;

    while (current != NULL || top >= 0) {
        while (current != NULL) {
            stack[++top] = current;
            current = current->left;
        }
        current = stack[top--];
        printf("%d ", current->data);
        current = current->right;
    }
}

void postorderNonRecursive(struct Node* root) {
    struct Node* stack1[100], *stack2[100];
    int top1 = -1, top2 = -1;

    if (root == NULL) return;
```

```
    stack1[++top1] = root;

    while (top1 >= 0) {
        struct Node* current = stack1[top1--];
        stack2[++top2] = current;

        if (current->left != NULL) stack1[++top1] = current->left;
        if (current->right != NULL) stack1[++top1] = current->right;
    }

    while (top2 >= 0) {
        printf("%d ", stack2[top2--]->data);
    }
}

// Main function
int main() {
    struct Node* root = NULL;
    int values[] = {50, 30, 20, 40, 70, 60, 80};
    int n = sizeof(values) / sizeof(values[0]);

    // Construct the binary tree
    for (int i = 0; i < n; i++) {
        root = insertNode(root, values[i]);
    }

    printf("Preorder Recursive: ");
    preorderRecursive(root);
    printf("\n");

    printf("Preorder Non-Recursive: ");
    preorderNonRecursive(root);
    printf("\n");

    printf("Inorder Recursive: ");
    inorderRecursive(root);
    printf("\n");
```

```
    printf("Inorder Non-Recursive: ");
    inorderNonRecursive(root);
    printf("\n");

    printf("Postorder Recursive: ");
    postorderRecursive(root);
    printf("\n");

    printf("Postorder Non-Recursive: ");
    postorderNonRecursive(root);
    printf("\n");

    return 0;
}
```

Output:

```
Preorder Recursive: 50 30 20 40 70 60 80
Preorder Non-Recursive: 50 30 20 40 70 60 80
Inorder Recursive: 20 30 40 50 60 70 80
Inorder Non-Recursive: 20 30 40 50 60 70 80
Postorder Recursive: 20 40 30 60 80 70 50
Postorder Non-Recursive: 20 40 30 60 80 70 50

--------------------------------
Process exited after 0.3193 seconds with return value 0
Press any key to continue . . .
```

Conclusion:
This experiment implemented the tree traversal methods pre-order, post-order and in-order traversing using recursive and non-recursive approach.

Experiment 3 viva Question

1. Explain the types of binary tree traversals.
2. How do you find the height of a binary tree?
3. What is the space complexity of a binary tree?
4. What is the maximum number of nodes in a binary tree of height *h*?
5. How do you perform a Pre-order traversal of a binary tree?
6. How do you perform a Post-order traversal of a binary tree?
7. What is the time complexity of searching for an element in a binary tree?
8. What are the applications of binary trees?
9. What is a degenerate binary tree?
10. What is the time complexity of in-order, pre-order, and post-order traversals?

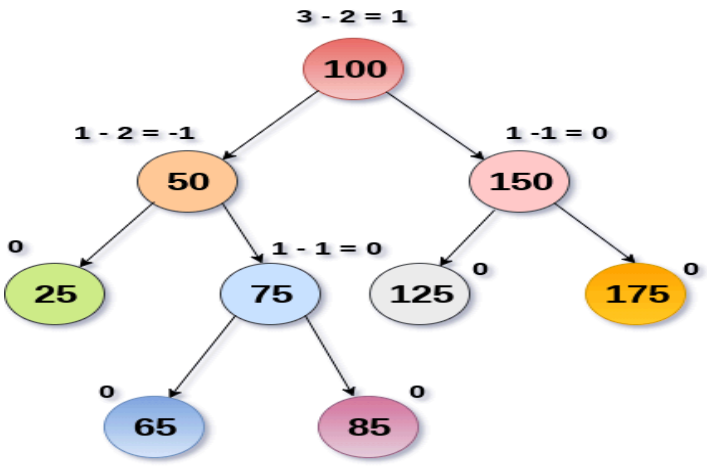
Experiment 4

Aim:
Implement AVL trees with following operations
a) Create
b) Insert
c) Search
d) Delete

AVL Tree

AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors. It is defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced. Below is the example of AVL tree.



AVL Tree

Insertion in AVL

Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.

Steps to be followed:

Let the newly inserted node be w

1. Perform standard BST insert for w.
2. Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.

3. Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that need to be handled as x, y and z can be arranged in 4 ways.

Deletion in AVL

Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore; various types of rotations are used to rebalance the tree.

Steps to be followed:

Let w be the node to be deleted

1. Perform standard BST delete for w.
2. Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the larger height child of z, and x be the larger height child of y. Note that the definitions of x and y are different from insertion here.
3. Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways.

AVL Rotations

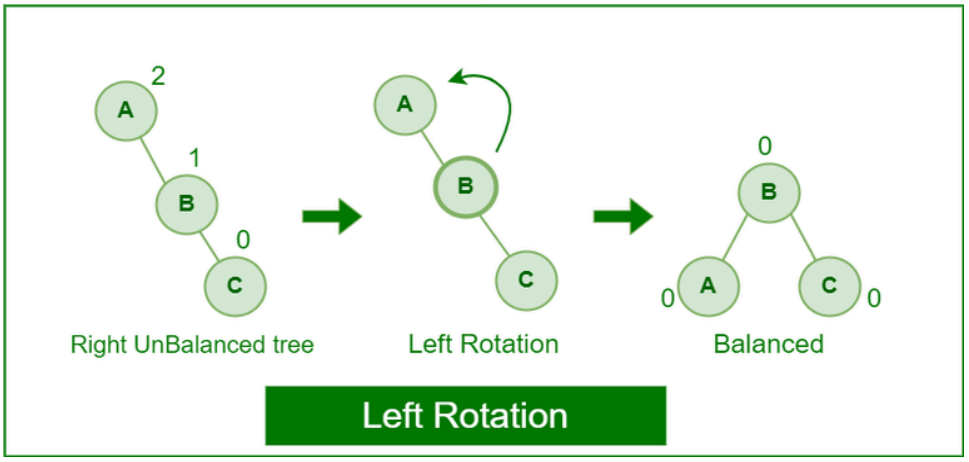
We perform Rotation in AVL tree only in case if Balance Factor is other than -1, 0, and 1. There are basically four types of rotations which are as follows:

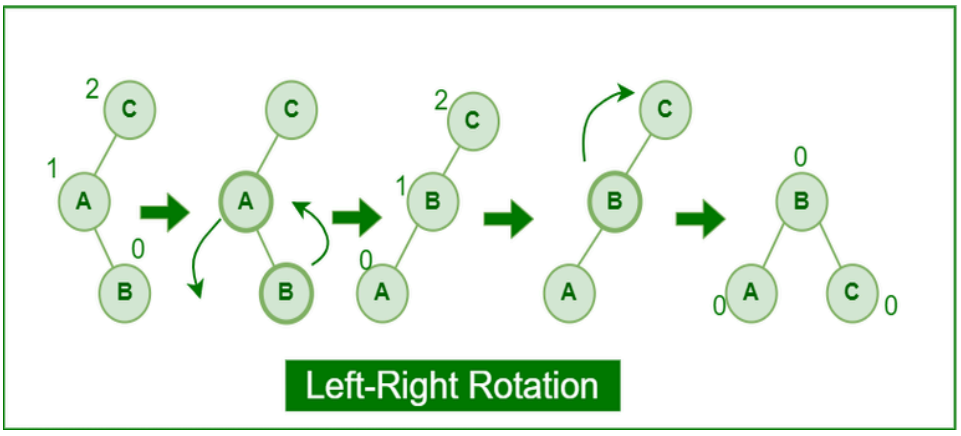
- **L L rotation:** Inserted node is in the left subtree of left subtree of A
- **R R rotation :** Inserted node is in the right subtree of right subtree of A
- **L R rotation :** Inserted node is in the right subtree of left subtree of A
- **R L rotation :** Inserted node is in the left subtree of right subtree of A

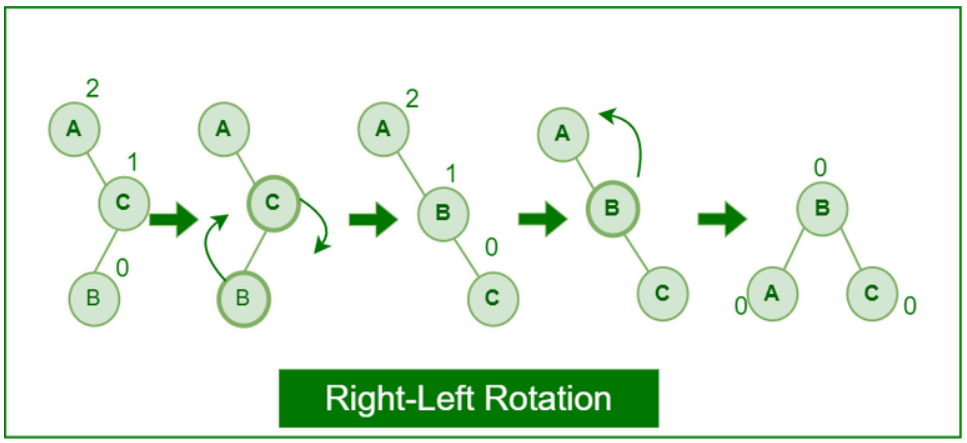
Where node A is the node whose balance Factor is other than -1, 0, 1

The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations. For a tree to be unbalanced, minimum height must be at least 2, Let us understand each rotation.

`


A
B
C
2
1
0
Right UnBalanced tree
Left Rotation
Balanced
Left Rotation


C
A
B
2
1
0
Left-Right Rotation



Code

```
#include <stdio.h>
#include <stdlib.h>

// Definition of a node in the AVL tree
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
    int height;
};

// Function to get the height of a node
int height(struct Node* node) {
    if (node == NULL)
        return 0;
    return node->height;
}

// Function to get the balance factor of a node
int getBalanceFactor(struct Node* node) {
    if (node == NULL)
        return 0;
```

G H Raisoni College of Engineering & Management
An Empowered Autonomous Institute, Affiliated to Savitribai Phule Pune University
Approved by AICTE, New Delhi and Recognized by Govt. of Maharashtra, NAAC Accredited "A+" Grade
Navin Gat No.1200, Domkhel Road, Wagholi, Pune – 412 207 (India)
T: +91 - 9604787185/186 | E: ghrcem.pune@raisoni.net | W: www.ghrcemp.raisoni.net

```
    return height(node->left) - height(node->right);
}

// Function to perform a right rotation
struct Node* rightRotate(struct Node* y) {
    struct Node* x = y->left;
    struct Node* T2 = x->right;

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

    // Update heights
    y->height = (height(y->left) > height(y->right)) ? height(y->left) + 1 : height(y->right) + 1;
    x->height = (height(x->left) > height(x->right)) ? height(x->left) + 1 : height(x->right) + 1;

    // Return new root
    return x;
}

// Function to perform a left rotation
struct Node* leftRotate(struct Node* x) {
    struct Node* y = x->right;
    struct Node* T2 = y->left;

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

    // Update heights
    x->height = (height(x->left) > height(x->right)) ? height(x->left) + 1 : height(x->right) + 1;
    y->height = (height(y->left) > height(y->right)) ? height(y->left) + 1 : height(y->right) + 1;

    // Return new root
    return y;
}
```

```
// Function to insert a node in the AVL tree
struct Node* insert(struct Node* node, int data) {
    // 1. Perform the normal BST insertion
    if (node == NULL) {
        struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->data = data;
        newNode->left = newNode->right = NULL;
        newNode->height = 1;  // New node is initially at height 1
        return newNode;
    }

    if (data < node->data)
        node->left = insert(node->left, data);
    else if (data > node->data)
        node->right = insert(node->right, data);
    else // Duplicate values are not allowed
        return node;

    // 2. Update the height of this ancestor node
    node->height = 1 + (height(node->left) > height(node->right) ? height(node->left) :
height(node->right));

    // 3. Get the balance factor to check if the node is unbalanced
    int balance = getBalanceFactor(node);

    // 4. If the node becomes unbalanced, then there are 4 cases

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

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

    // Left Right Case
    if (balance > 1 && data > node->left->data) {
```

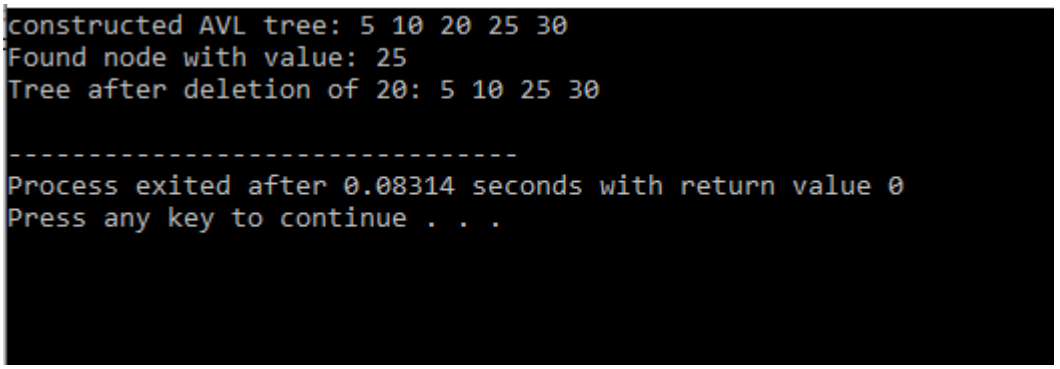
```
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && data < node->right->data) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}

// Function to search for a value in the AVL tree
struct Node* search(struct Node* root, int key) {
    if (root == NULL || root->data == key)
        return root;
    if (root->data < key)
        return search(root->right, key);
    return search(root->left, key);
}

// Function to find the node with the minimum value
struct Node* minNode(struct Node* node) {
    struct Node* current = node;
    while (current->left != NULL)
        current = current->left;
    return current;
}

// Function to delete a node from the AVL tree
struct Node* deleteNode(struct Node* root, int key) {
    // STEP 1: Perform standard BST delete
    if (root == NULL)
        return root;

    if (key < root->data)
```

```
    root->left = deleteNode(root->left, key);
  else if (key > root->data)
    root->right = deleteNode(root->right, key);
  else {
    if (root->left == NULL || root->right == NULL) {
      struct Node* temp = root->left ? root->left : root->right;
      if (temp == NULL) {
        temp = root;
        root = NULL;
      } else
        *root = *temp;
      free(temp);
    } else {
      struct Node* temp = minNode(root->right);
      root->data = temp->data;
      root->right = deleteNode(root->right, temp->data);
    }
  }

  if (root == NULL)
    return root;

  // STEP 2: Update height of the current node
  root->height = 1 + (height(root->left) > height(root->right) ? height(root->left) : height(root->right));

  // STEP 3: Get the balance factor to check whether the node became unbalanced
  int balance = getBalanceFactor(root);

  // STEP 4: Balance the node if it's unbalanced
  if (balance > 1 && getBalanceFactor(root->left) >= 0)
    return rightRotate(root);

  if (balance > 1 && getBalanceFactor(root->left) < 0) {
    root->left = leftRotate(root->left);
    return rightRotate(root);
  }
```

```
    if (balance < -1 && getBalanceFactor(root->right) <= 0)
        return leftRotate(root);

    if (balance < -1 && getBalanceFactor(root->right) > 0) {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}

// Function to print the tree (Inorder Traversal)
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

// Main function
int main() {
    struct Node* root = NULL;
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 25);
    root = insert(root, 5);

    printf("constructed AVL tree: ");
    inorder(root);
    printf("\n");

    struct Node* searchResult = search(root, 25);
    if (searchResult != NULL)
        printf("Found node with value: %d\n", searchResult->data);
    else
```

```
        printf("Node with value 25 not found.\n");

    root = deleteNode(root, 20);
    printf("Tree after deletion of 20: ");
    inorder(root);
    printf("\n");

    return 0;
}
```

Output



Conclusion

The aim of this experiment is to explain AVL tree and properties of AVL tree. It implemented operations like insertion, searching and deletion on the AVL tree.

G H Raisoni College of Engineering & Management
An Empowered Autonomous Institute, Affiliated to Savitribai Phule Pune University
Approved by AICTE, New Delhi and Recognized by Govt. of Maharashtra, NAAC Accredited "A+" Grade
Navin Gat No.1200, Domkhel Road, Wagholi, Pune – 412 207 (India)
T: +91 - 9604787185/186 | E: ghrcem.pune@raisoni.net | W: www.ghrcemp.raisoni.net

`

Experiment 4 viva Question

1. What is the balance factor of an AVL tree?
2. Explain different types of rotations in an AVL tree.
3. What is the time complexity for inserting a node in an AVL tree?
4. What is the time complexity for deleting a node from an AVL tree?
5. How is the height of a node in an AVL tree determined?
6. Can an AVL tree be implemented using a doubly linked list?
7. What is the maximum height of an AVL tree with ***n*** nodes?
8. Can an AVL tree become unbalanced after a delete operation?
9. Give an example where an AVL tree provides better performance than a regular binary search tree.
10. What is the advantage of using an AVL tree over a regular binary search tree?

Experiment 5

Aim:

Write a program to perform the following operations:
a) Insertion into a B-tree
b) Searching in a B-tree

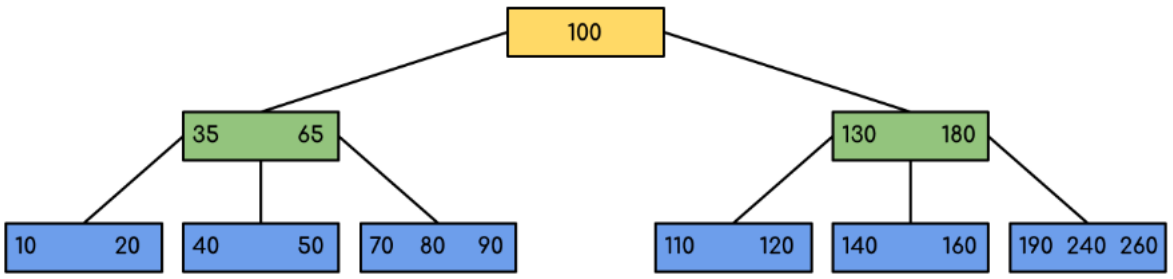
Theory:

A **B-tree** is a self-balancing search tree that maintains sorted data and allows efficient insertion, deletion, and searching operations. It is used in databases and file systems for indexing because it keeps data in sorted order and allows for logarithmic time complexity for search, insertion, and deletion operations.

Key properties of a B-tree:

- Each node can have multiple keys.
- The tree is balanced, and all leaf nodes are at the same depth.
- It is a generalization of a binary search tree where each node can have more than two children.
- It maintains the order property, where keys are ordered within nodes and among the children of each node.

Example of B Tree



B-Tree Insertion: Insertion into a B-tree involves:

- Traversing the tree from the root to find the correct position for the new key.
- If the node where the key is to be inserted is full, it is split into two nodes, and the middle key is moved up into the parent node.
- If the parent node is full, this process continues recursively.

When inserting a new key into a B-tree following are the sequence of steps

1. **Find the correct leaf node**: Start from the root and find the appropriate leaf node where the key should be inserted. You traverse down the tree by comparing the key to be inserted with the keys in the current node.
2. **Insert the key**: Insert the key in the appropriate position within the leaf node.
3. **Handle overflow**: If the leaf node exceeds its maximum capacity (`m-1` keys), split the node:
    - Split the node into two nodes.
    - Promote the middle key to the parent node.
4. **Recursive promotion**: If the parent node also overflows, repeat the process recursively up the tree. If the root overflows, create a new root.

B-Tree Search: Search in a B-tree is similar to a binary search. For each node, the algorithm:

- Compares the search key with the keys in the node.
- If the key is found, the search is successful.
- If the key is not found, the search continues in the appropriate child node.

Code:

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 3  // Maximum number of keys in a node
#define MIN 2  // Minimum number of keys in a node

// Structure to represent a B-tree node
struct BTreeNode {
    int keys[MAX + 1];
    struct BTreeNode* children[MAX + 2];
    int numKeys;
    int leaf;
};

// Function to create a new B-tree node
```

```
struct BTreeNode* createNode(int leaf) {
    struct BTreeNode* newNode = (struct BTreeNode*)malloc(sizeof(struct BTreeNode));
    newNode->leaf = leaf;
    newNode->numKeys = 0;
    for (int i = 0; i < MAX + 2; i++) {
        newNode->children[i] = NULL;
    }
    return newNode;
}

// Function to split a full node
void splitChild(struct BTreeNode* parent, int index, struct BTreeNode* fullChild) {
    struct BTreeNode* newChild = createNode(fullChild->leaf);
    newChild->numKeys = MIN;

    // Copy the last MIN keys from the full child to the new child
    for (int i = 0; i < MIN; i++) {
        newChild->keys[i] = fullChild->keys[i + MIN + 1];
    }

    // If it's not a leaf, copy the last MIN + 1 children
    if (!fullChild->leaf) {
        for (int i = 0; i < MIN + 1; i++) {
            newChild->children[i] = fullChild->children[i + MIN + 1];
        }
    }

    fullChild->numKeys = MIN;

    // Shift the children and keys of the parent node to make room for the new child
    for (int i = parent->numKeys; i > index; i--) {
        parent->children[i + 1] = parent->children[i];
    }

    parent->children[index + 1] = newChild;

    // Shift the keys of the parent to make room for the middle key of the full child
    for (int i = parent->numKeys - 1; i >= index; i--) {
        parent->keys[i + 1] = parent->keys[i];
    }

    // Move the middle key of the full child to the parent node
```

```
    parent->keys[index] = fullChild->keys[MIN];
    parent->numKeys++;
}

// Function to insert a key into a non-full node
void insertNonFull(struct BTreeNode* node, int key) {
    int i = node->numKeys - 1;

    if (node->leaf) {
        // Find the position where the new key should be inserted
        while (i >= 0 && key < node->keys[i]) {
            node->keys[i + 1] = node->keys[i];
            i--;
        }
        node->keys[i + 1] = key;
        node->numKeys++;
    } else {
        // Find the child to insert into
        while (i >= 0 && key < node->keys[i]) {
            i--;
        }
        i++;

        // If the child is full, split it
        if (node->children[i]->numKeys == MAX) {
            splitChild(node, i, node->children[i]);

            // Determine which of the two children to insert into
            if (key > node->keys[i]) {
                i++;
            }
        }
        insertNonFull(node->children[i], key);
    }
}

// Function to insert a key into the B-tree
void insert(struct BTreeNode** root, int key) {
    // If the root is full, split it and create a new root
    if ((*root)->numKeys == MAX) {
        struct BTreeNode* newRoot = createNode(0);
        newRoot->children[0] = *root;
```

```
        splitChild(newRoot, 0, *root);
        *root = newRoot;
    }

    // Insert the key into the non-full root
    insertNonFull(*root, key);
}

// Function to search for a key in the B-tree
struct BTreeNode* search(struct BTreeNode* root, int key) {
    int i = 0;
    while (i < root->numKeys && key > root->keys[i]) {
        i++;
    }

    if (i < root->numKeys && key == root->keys[i]) {
        return root; // Key found
    } else if (root->leaf) {
        return NULL; // Key not found
    } else {
        return search(root->children[i], key); // Search the appropriate child
    }
}

// Function to print the B-tree (Inorder Traversal)
void inorder(struct BTreeNode* root) {
    if (root != NULL) {
        for (int i = 0; i < root->numKeys; i++) {
            inorder(root->children[i]);
            printf("%d ", root->keys[i]);
        }
        inorder(root->children[root->numKeys]);
    }
}

// Main function
int main() {
    struct BTreeNode* root = createNode(1);

    // Insert keys into the B-tree
    insert(&root, 10);
    insert(&root, 20);
```

```
insert(&root, 5);
insert(&root, 6);
insert(&root, 12);
insert(&root, 30);
insert(&root, 7);

// Print the B-tree in Inorder
printf("Insertion in B-tree: ");
inorder(root);
printf("\n");

// Search for a key
int searchKey = 12;
struct BTreeNode* searchResult = search(root, searchKey);
if (searchResult != NULL) {
    printf("Key %d found in the B-tree.\n", searchKey);
} else {
    printf("Key %d not found in the B-tree.\n", searchKey);
}

return 0;
  }
```

Output:

```
Insertion in B-tree: 5 6 7 10 0 0 12 20 0 0 30
Key 12 found in the B-tree.

--------------------------------
Process exited after 0.08592 seconds with return value 0
Press any key to continue . . .
```

Conclusion: B Tree is an essential data structure. This experiment discussed insertion and searching in B Tree and implemented the same.

Experiment 5 viva Question

1. Explain B-tree, and how does it differ from a binary search tree?
2. What are the properties of a B-tree?
3. What does the "order" of a B-tree refer to?
4. Why is it important for B-trees to remain balanced?
5. What is the maximum number of children a node in a B-tree can have?
6. How do B-trees maintain balance while performing insertions or deletions?
7. Explain the process of insertion in a B-tree.
8. What happens when a node in a B-tree overflows during insertion?
9. How does searching for a key in a B-tree differ from searching in a binary search tree?
10. What is the time complexity for searching in a B-tree?

Experiment 6

Aim:

Develop C function to implement graph using
a) Adjacency lists
b) Adjacency matrices
c) Demonstrate Djkstra's algorithm on a weighted graph (VLAB: https://ds2-iiith.vlabs.ac.in/exp/dijkstra-algorithm/dijkstras-algorithm/index.html)

Theory:

Graph

A graph is a collection of nodes (vertices) and edges (connections between nodes). In graph theory, we often need to represent graphs in a way that allows efficient operations such as searching, adding/removing edges, and calculating shortest paths.

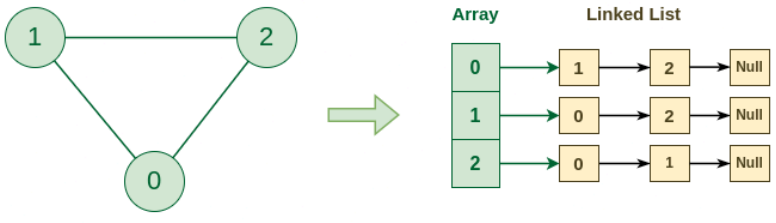
A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices (V) and a set of edges (E). The graph is denoted by G (V, E).

There are two main ways to represent a graph:

- Adjacency List
- Adjacency Matrix

Adjacency List:

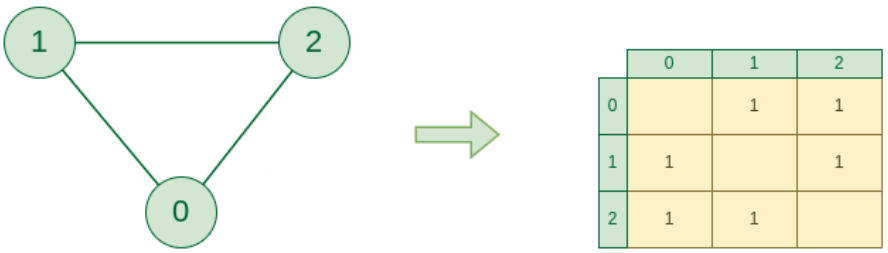
An adjacency list is a collection of lists or arrays where each list corresponds to a vertex in the graph, and the elements of the list are the vertices that are adjacent to it (connected by an edge).



Graph Representation of Undirected graph to Adjacency List

Adjacency Matrix:
An adjacency matrix is a 2D matrix where the element at position (i, j) represents the presence or weight of an edge between vertex i and vertex j. This is a way of representing a graph as a matrix of boolean (0's and 1's)

Let's assume there are n vertices in the graph So, create a 2D matrix adjMat[n][n] having dimension n x n. Below is the example.



Undirected Graph Adjacency Matrix

Graph Representation of Undirected graph to Adjacency Matrix

Dijkstra's Algorithm
Dijkstra's algorithm is used for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It works by iteratively selecting the vertex with the smallest tentative distance and exploring its neighbors.
Steps of Dijkstra's Algorithm:

1. Initialize distances to all vertices as infinite, except for the source node which has a distance of 0
2. Use a priority queue to select the node with the smallest tentative distance.
3. Update the distances of its adjacent nodes.
4. Repeat until all nodes have been visited.

How it works:

- Set initial distances for all vertices: 0 for the source vertex, and infinity for all the other.
- Choose the unvisited vertex with the shortest distance from the start to be the current vertex. So the algorithm will always start with the source as the current vertex.
- For each of the current vertex's unvisited neighbor vertices, calculate the distance from the source and update the distance if the new, calculated, distance is lower.
- We are now done with the current vertex, so we mark it as visited. A visited vertex is not checked again.

- Go back to step 2 to choose a new current vertex, and keep repeating these steps until all vertices are visited.
- In the end we are left with the shortest path from the source vertex to every other vertex in the graph.

Code:

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 10

// Structure to represent a node in the adjacency list
struct Node {
    int vertex;
    int weight;
    struct Node* next;
};

// Structure to represent a graph
struct Graph {
    struct Node* adjList[MAX];
};

// Function to initialize the graph
void initGraph(struct Graph* graph) {
    for (int i = 0; i < MAX; i++) {
        graph->adjList[i] = NULL;
    }
}

// Function to add an edge to the graph
void addEdge(struct Graph* graph, int src, int dest, int weight) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->vertex = dest;
    newNode->weight = weight;
    newNode->next = graph->adjList[src];
    graph->adjList[src] = newNode;

    // Since it's an undirected graph, add an edge from dest to src
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->vertex = src;
    newNode->weight = weight;
```

```
    newNode->next = graph->adjList[dest];
    graph->adjList[dest] = newNode;
}

// Function to print the adjacency list
void printGraph(struct Graph* graph) {
    for (int i = 0; i < MAX; i++) {
        struct Node* temp = graph->adjList[i];
        printf("\nVertex %d:", i);
        while (temp) {
            printf(" -> %d (Weight: %d)", temp->vertex, temp->weight);
            temp = temp->next;
        }
    }
    printf("\n");
}

int main() {
    struct Graph graph;
    initGraph(&graph);

    addEdge(&graph, 0, 1, 4);
    addEdge(&graph, 0, 7, 8);
    addEdge(&graph, 1, 2, 8);
    addEdge(&graph, 1, 7, 11);
    addEdge(&graph, 2, 3, 7);
    addEdge(&graph, 2, 5, 4);
    addEdge(&graph, 2, 8, 2);
    addEdge(&graph, 3, 4, 9);
    addEdge(&graph, 3, 5, 14);
    addEdge(&graph, 4, 5, 10);
    addEdge(&graph, 5, 6, 2);
    addEdge(&graph, 6, 7, 1);
    addEdge(&graph, 6, 8, 6);
    addEdge(&graph, 7, 8, 7);

    printGraph(&graph);
    return 0;
}
```

Output

```
Vertex 0: -> 7 (Weight: 8) -> 1 (Weight: 4)
Vertex 1: -> 7 (Weight: 11) -> 2 (Weight: 8) -> 0 (Weight: 4)
Vertex 2: -> 8 (Weight: 2) -> 5 (Weight: 4) -> 3 (Weight: 7) -> 1 (Weight: 8)
Vertex 3: -> 5 (Weight: 14) -> 4 (Weight: 9) -> 2 (Weight: 7)
Vertex 4: -> 5 (Weight: 10) -> 3 (Weight: 9)
Vertex 5: -> 6 (Weight: 2) -> 4 (Weight: 10) -> 3 (Weight: 14) -> 2 (Weight: 4)
Vertex 6: -> 8 (Weight: 6) -> 7 (Weight: 1) -> 5 (Weight: 2)
Vertex 7: -> 8 (Weight: 7) -> 6 (Weight: 1) -> 1 (Weight: 11) -> 0 (Weight: 8)
Vertex 8: -> 7 (Weight: 7) -> 6 (Weight: 6) -> 2 (Weight: 2)
Vertex 9:

--------------------------------
Process exited after 0.0906 seconds with return value 0
Press any key to continue . . .
```

```c
#include <stdio.h>

#define MAX 10
#define INF 999999

// Structure to represent a graph
struct Graph {
    int adjMatrix[MAX][MAX];
};

// Function to initialize the graph
void initGraph(struct Graph* graph) {
    for (int i = 0; i < MAX; i++) {
        for (int j = 0; j < MAX; j++) {
            graph->adjMatrix[i][j] = INF;  // Use INF to represent no edge
        }
    }
}

// Function to add an edge to the graph
void addEdge(struct Graph* graph, int src, int dest, int weight) {
    graph->adjMatrix[src][dest] = weight;
    graph->adjMatrix[dest][src] = weight;  // For undirected graph
}
```

```
// Function to print the adjacency matrix
void printGraph(struct Graph* graph) {
    printf("\nAdjacency Matrix:\n");
    for (int i = 0; i < MAX; i++) {
        for (int j = 0; j < MAX; j++) {
            if (graph->adjMatrix[i][j] == INF)
                printf("INF ");
            else
                printf("%d ", graph->adjMatrix[i][j]);
        }
        printf("\n");
    }
}

int main() {
    struct Graph graph;
    initGraph(&graph);

    addEdge(&graph, 0, 1, 4);
    addEdge(&graph, 0, 7, 8);
    addEdge(&graph, 1, 2, 8);
    addEdge(&graph, 1, 7, 11);
    addEdge(&graph, 2, 3, 7);
    addEdge(&graph, 2, 5, 4);
    addEdge(&graph, 2, 8, 2);
    addEdge(&graph, 3, 4, 9);
    addEdge(&graph, 3, 5, 14);
    addEdge(&graph, 4, 5, 10);
    addEdge(&graph, 5, 6, 2);
    addEdge(&graph, 6, 7, 1);
    addEdge(&graph, 6, 8, 6);
    addEdge(&graph, 7, 8, 7);

    printGraph(&graph);
    return 0;
}
```

Output:

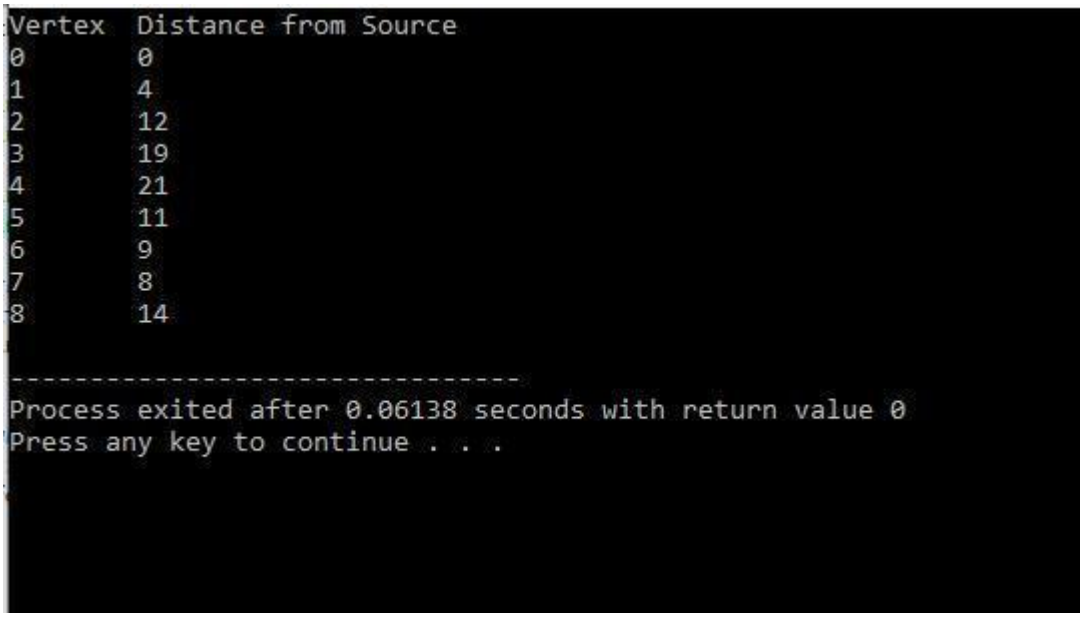
```
Adjacency Matrix:
INF 4 INF INF INF INF INF 8 INF INF
4 INF 8 INF INF INF INF 11 INF INF
INF 8 INF 7 INF 4 INF INF 2 INF
INF INF 7 INF 9 14 INF INF INF INF
INF INF INF 9 INF 10 INF INF INF INF
INF INF 4 14 10 INF 2 INF INF INF
INF INF INF INF INF 2 INF 1 6 INF
8 11 INF INF INF INF 1 INF 7 INF
INF INF 2 INF INF INF 6 7 INF INF
INF INF INF INF INF INF INF INF INF INF

--------------------------------
Process exited after 0.06245 seconds with return value 0
Press any key to continue . . .
```

```
#include <stdio.h>
#include <limits.h>

#define MAX 10
#define INF 999999

// Structure to represent a graph
struct Graph {
    int adjMatrix[MAX][MAX];
};

// Function to initialize the graph
void initGraph(struct Graph* graph) {
    for (int i = 0; i < MAX; i++) {
        for (int j = 0; j < MAX; j++) {
            graph->adjMatrix[i][j] = INF;  // Use INF to represent no edge
        }
    }
}

// Function to add an edge to the graph
void addEdge(struct Graph* graph, int src, int dest, int weight) {
```

```
    graph->adjMatrix[src][dest] = weight;
    graph->adjMatrix[dest][src] = weight;  // For undirected graph
}

// Function to find the vertex with the minimum distance
int minDistance(int dist[], int sptSet[], int V) {
    int min = INF, min_index;
    for (int v = 0; v < V; v++) {
        if (sptSet[v] == 0 && dist[v] <= min) {
            min = dist[v];
            min_index = v;
        }
    }
    return min_index;
}

// Function to implement Dijkstra's algorithm
void dijkstra(struct Graph* graph, int src, int V) {
    int dist[V];
    int sptSet[V];  // Shortest path tree set

    // Initialize all distances as INF and sptSet[] as 0
    for (int i = 0; i < V; i++) {
        dist[i] = INF;
        sptSet[i] = 0;
    }

    dist[src] = 0;

    // Find the shortest path for all vertices
    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, sptSet, V);
        sptSet[u] = 1;

        // Update dist[] for adjacent vertices of the picked vertex
        for (int v = 0; v < V; v++) {
            if (!sptSet[v] && graph->adjMatrix[u][v] != INF && dist[u] != INF &&
                dist[u] + graph->adjMatrix[u][v] < dist[v]) {
                dist[v] = dist[u] + graph->adjMatrix[u][v];
            }
        }
    }
```

```
    // Print the constructed distance array
    printf("Vertex\tDistance from Source\n");
    for (int i = 0; i < V; i++) {
        printf("%d\t%d\n", i, dist[i]);
    }
}

int main() {
    struct Graph graph;
    initGraph(&graph);

    // Example graph with weighted edges
    addEdge(&graph, 0, 1, 4);
    addEdge(&graph, 0, 7, 8);
    addEdge(&graph, 1, 2, 8);
    addEdge(&graph, 1, 7, 11);
    addEdge(&graph, 2, 3, 7);
    addEdge(&graph, 2, 5, 4);
    addEdge(&graph, 2, 8, 2);
    addEdge(&graph, 3, 4, 9);
    addEdge(&graph, 3, 5, 14);
    addEdge(&graph, 4, 5, 10);
    addEdge(&graph, 5, 6, 2);
    addEdge(&graph, 6, 7, 1);
    addEdge(&graph, 6, 8, 6);
    addEdge(&graph, 7, 8, 7);

    // Run Dijkstra's algorithm from vertex 0
    dijkstra(&graph, 0, 9);

    return 0;
}
```

Output:



Experiment 6 viva Question

1. What are the different ways to represent a graph in memory?
2. In an adjacency matrix, how do you check if there is an edge between two vertices?
3. In an adjacency list, how do you check if there is an edge between two vertices?
4. In which applications would you prefer using an adjacency matrix over an adjacency list?
5. Which graph representation is more space-efficient for sparse graphs?
6. What is the primary objective of Dijkstra's algorithm?
7. How does Dijkstra's algorithm find the shortest path in a graph?
8. What types of graphs can Dijkstra's algorithm be applied to?
9. Can Dijkstra's algorithm handle negative edge weights? Why or why not?

10. Explain the step-by-step process of Dijkstra's algorithm.

Experiment 7

Aim:

The given directed graph contains N nodes and M edges. Each edge connects two nodes, Write a program to find the sequence of nodes in the order they are visited using

a) DFS

b) BFS

c) Graph traversal concepts understanding (VLAB: https://ds1-iiith.vlabs.ac.in/exp/depth-first-search/graph-traversals.html).

Graph Traversal

Graph traversal refers to the process of visiting all the nodes or vertices in a graph in a specific order. There are two primary methods for graph traversal:

Breadth-First Search (BFS):
Breadth First Search (BFS) is a fundamental graph traversal algorithm. It begins with a node, then first traverses all its adjacent. Once all adjacent are visited, then their adjacent are traversed.

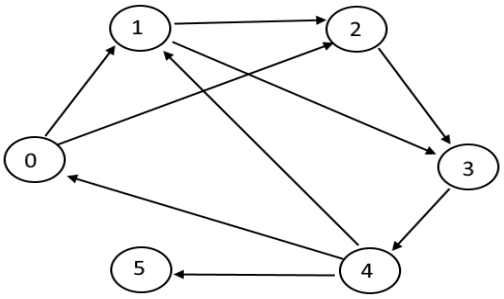
Approach: BFS explores all the neighbors of a node before moving on to the next level neighbors.

Algorithm:

- Start from a source node.
- Explore all unvisited neighbors at the present depth level before moving on to nodes at the next depth level.
- Use a queue to keep track of nodes to visit next.

BFS Example

Graph



BFS starting from Node 0

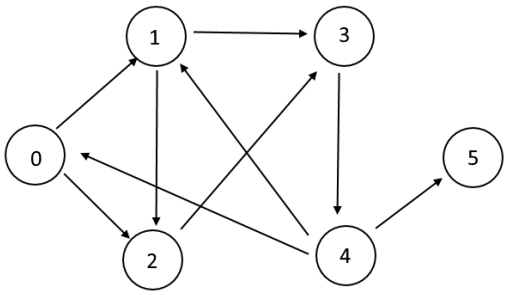
0 2 1 3 4 5

Depth-First Search (DFS):
The algorithm starts from a given source and explores all reachable vertices from the given source

Approach: DFS explores as far as possible along a branch before backtracking.
Algorithm:

- Start from a source node.
- Explore each branch of the graph as deeply as possible before moving to the next branch.
- Use a stack (either explicitly or through recursion) to keep track of the path.

DFS Example



Depth First Traversal - 0 1 3 4 5 2

C Code

```
#include<stdio.h>
int q[20],top=-1,front=-1,rear=-1,a[20][20],vis[20],stack[20];
int del();
void add(int item);
void bfs(int s,int n);
void dfs(int s,int n);
void push(int item);
int pop();

int main()
{
int n,i,s,ch,j;
char c,dummy;
printf("ENTER THE NUMBER VERTICES ");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
printf("ENTER 1 IF %d HAS A NODE WITH %d ELSE 0 ",i,j);
scanf("%d",&a[i][j]);
}
}
printf("THE ADJACENCY MATRIX IS\n");
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
printf(" %d",a[i][j]);
}
printf("\n");
}

do
{
for(i=1;i<=n;i++)
vis[i]=0;
printf("\nMENU");
printf("\n1.B.F.S");
printf("\n2.D.F.S");
printf("\nENTER YOUR CHOICE");
```

```
scanf("%d",&ch);
printf("ENTER THE SOURCE VERTEX :");
scanf("%d",&s);

switch(ch)
{
case 1:bfs(s,n);
break;
case 2:
dfs(s,n);
break;
}
printf("DO U WANT TO CONTINUE(Y/N) ? ");
scanf("%c",&dummy);
scanf("%c",&c);
}while((c=='y')||(c=='Y'));
return 0;
}


//**************BFS(breadth-first search) code**************//
void bfs(int s,int n)
{
int p,i;
add(s);
vis[s]=1;
p=del();
if(p!=0)
printf(" %d",p);
while(p!=0)
{
for(i=1;i<=n;i++)
if((a[p][i]!=0)&&(vis[i]==0))
{
add(i);
vis[i]=1;
}
p=del();
if(p!=0)
printf(" %d ",p);
}
for(i=1;i<=n;i++)
```

```
if(vis[i]==0)
bfs(i,n);
}

void add(int item)
{
if(rear==19)
printf("QUEUE FULL");
else
{
if(rear==-1)
{
q[++rear]=item;
front++;
}
else
q[++rear]=item;
}
}
int del()
{
int k;
if((front>rear)||(front==-1))
return(0);
else
{
k=q[front++];
return(k);
}
}

//***************DFS(depth-first search) code******************//
void dfs(int s,int n)
{
int i,k;
push(s);
vis[s]=1;
k=pop();
if(k!=0)
printf(" %d ",k);
while(k!=0)
{
```

```
for(i=1;i<=n;i++)
if((a[k][i]!=0)&&(vis[i]==0))
{
push(i);
vis[i]=1;
}
k=pop();
if(k!=0)
printf(" %d ",k);
}
for(i=1;i<=n;i++)
if(vis[i]==0)
dfs(i,n);
}
void push(int item)
{
if(top==19)
printf("Stack overflow ");
else
stack[++top]=item;
}
int pop()
{
int k;
if(top==-1)
return(0);
else
{
k=stack[top--];
return(k);
}
}
```

Output:

```
ENTER THE NUMBER VERTICES 4
ENTER 1 IF 1 HAS A NODE WITH 1 ELSE 0 1
ENTER 1 IF 1 HAS A NODE WITH 2 ELSE 0 0
ENTER 1 IF 1 HAS A NODE WITH 3 ELSE 0 1
ENTER 1 IF 1 HAS A NODE WITH 4 ELSE 0 0
ENTER 1 IF 2 HAS A NODE WITH 1 ELSE 0 1
ENTER 1 IF 2 HAS A NODE WITH 2 ELSE 0 1
ENTER 1 IF 2 HAS A NODE WITH 3 ELSE 0 0
ENTER 1 IF 2 HAS A NODE WITH 4 ELSE 0 1
ENTER 1 IF 3 HAS A NODE WITH 1 ELSE 0 1
ENTER 1 IF 3 HAS A NODE WITH 2 ELSE 0 0
ENTER 1 IF 3 HAS A NODE WITH 3 ELSE 0 0
ENTER 1 IF 3 HAS A NODE WITH 4 ELSE 0 0
ENTER 1 IF 4 HAS A NODE WITH 1 ELSE 0 0
ENTER 1 IF 4 HAS A NODE WITH 2 ELSE 0 1
ENTER 1 IF 4 HAS A NODE WITH 3 ELSE 0 1
ENTER 1 IF 4 HAS A NODE WITH 4 ELSE 0 0
THE ADJACENCY MATRIX IS
 1 0 1 0
 1 1 0 1
 1 0 0 0
 0 1 1 0

MENU
1.B.F.S
2.D.F.S
ENTER YOUR CHOICE2
ENTER THE SOURCE VERTEX :3
 3  1  2  4 DO U WANT TO CONTINUE(Y/N) ? y
```

Conclusion:

This program implemented graph traversal algorithm DFS and BFS on a given graph.

Experiment 7 viva Question

1. How does BFS work?
2. How does DFS work?
3. When would you use BFS and when would you use DFS?
4. What data structures are used in BFS and DFS?
5. What is the time complexity of BFS and DFS?
6. What is the space complexity of BFS and DFS?
7. Can DFS and BFS be used for searching in a tree structure?
8. How does DFS handle disconnected graphs?
9. What happens if a graph contains cycles?
10. What are some real-world applications of BFS and DFS?