

**LABORATORY RECORD NOTEBOOK**

# AMITY UNIVERSITY CHHATTISGARH

**LABORATORY MANUAL**

*A Lab Manual Submitted to*

**Amity University Chhattisgarh, Raipur (AUC)**



In

**Computer Science Engineering**

By Niladitya Sen

(A80105222017)

*Submitted to*

**Mrs Nisha Rathore**

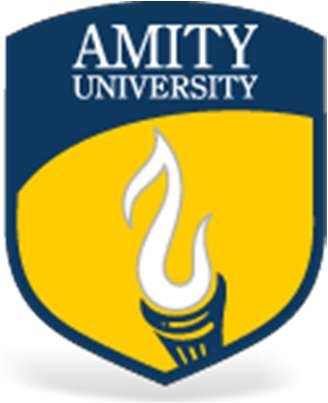
**Amity University Chhattisgarh, Raipur**

# Amity School of Engineering & Technology

**Amity University Chhattisgarh, Raipur (AUC)**

**Manth / Kharora, (Opp.-ITBP) SH-09,**

**Raipur, Chhattisgarh - 493225 (2023-24)**

**Amity University Chhattisgarh**

**Amity School of Engineering & Technology**

**Batch: 2022- 2026**

**Enrolment Number**: A80105222017

This is to certify that this is a Bonafede record of the work done by **Niladitya Sen** bearing enrolment number **A80105222017** of **B. Tech Computer Science & Engineering** Semester **III**, from Amity School of Engineering & Technology, Amity University Chhattisgarh in the **DATA STRUCTURES LAB** with Course Code **CSE3306.**

University Examination held on .

Faculty In-charge Director-ASET

Examiner-1 Examiner-2

INDEX

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sl. No** | **Experiment** | **Date of Experiment** | **Date of Submission** | **Signature** |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

# Experiment 1

**Aim: Write a C program that uses functions to perform the following:**

1. **Create a singly linked list of integers.**
2. **Delete a given integer from the above linked list.**
3. **Display the contents of the above list after deletion.**

**Solution:**

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

struct Node {

int data;

struct Node \*next;

};

struct Node \*HEAD = NULL;

void insertAtBeginning(int data) {

struct Node \*newNode = (struct Node\*) malloc(sizeof(struct Node)); newNode->data = data;

newNode->next = HEAD; HEAD = newNode;

}

void insertAtEnd(int data) {

struct Node \*newNode = (struct Node\*) malloc(sizeof(struct Node)); newNode->data = data;

newNode->next = NULL;

struct Node \*temp = HEAD; while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

void insertAtIndex(int index, int data) { if (index == 0) {

insertAtBeginning(data); return;

}

struct Node \*newNode = (struct Node\*) malloc(sizeof(struct Node)); newNode->data = data;

struct Node \*temp = HEAD; struct Node \*prev = HEAD; int counter = 0;

while (counter <= index - 1 && temp != NULL) { prev = temp;

temp = temp->next; counter++;

}

newNode->next = temp; prev->next = newNode;

}

void deleteAtEnd() {

struct Node \*temp = HEAD;

while (temp->next->next != NULL) { temp = temp->next;

}

temp->next = NULL;

}

void deleteAtBeginning() { HEAD = HEAD->next;

}

void deleteAtIndex(int index) { if (index == 0) {

deleteAtBeginning(); return;

}

struct Node \*temp = HEAD; struct Node \*prev = HEAD;

while (index != 0 && temp->next->next != NULL) { prev = temp;

temp = temp->next; index--;

}

prev->next = temp->next;

}

void printList() {

struct Node \*temp = HEAD; printf("\n[");

while (temp != NULL) { printf("%d ", temp->data); temp = temp->next;

}

printf("]");

}

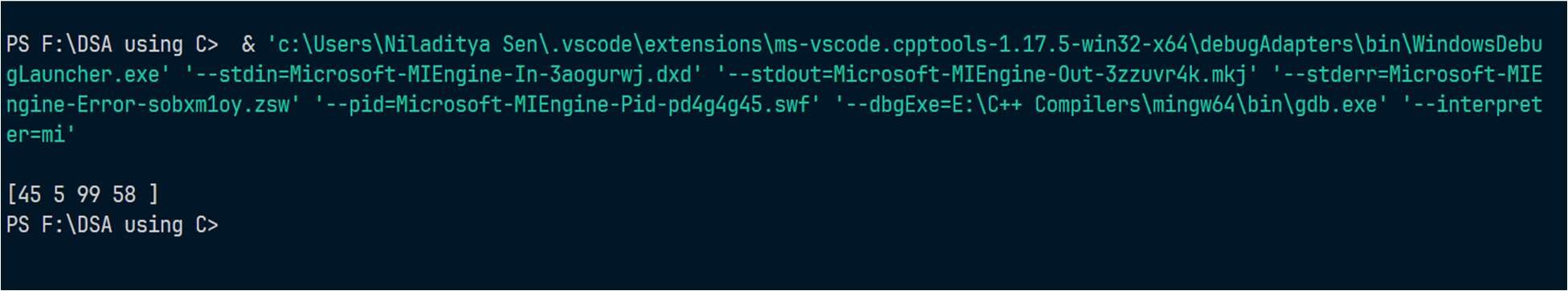
int main() {

insertAtBeginning(12); insertAtBeginning(5); insertAtBeginning(45); insertAtBeginning(10); insertAtEnd(99); insertAtEnd(58); insertAtIndex(90, 3); deleteAtIndex(3); deleteAtBeginning(); deleteAtEnd(); printList();

return 0;

}

**Output:**



# Experiment 2

**Aim: Write a C program that uses functions to perform the following:**

1. **Create a doubly linked list of integers.**
2. **Delete a given integer from the above doubly linked list.**
3. **Display the contents of the above list after deletion.**

**Solution:**

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

typedef struct NodeType { int data;

struct NodeType \*prev; struct NodeType \*next;

} Node;

Node \*HEAD, \*TAIL = NULL;

void insertAtBeginning(int data) {

Node \*newNode = (Node\*) malloc(sizeof(Node)); newNode->data = data;

newNode->prev = NULL; newNode->next = HEAD;

if (HEAD == NULL) {

HEAD = TAIL = newNode;

} else {

HEAD->prev = newNode;

HEAD = newNode;

}

}

void insertAtEnd(int data) {

Node \*newNode = (Node\*) malloc(sizeof(Node)); newNode->data = data;

newNode->next = NULL;

if (HEAD == NULL) {

insertAtBeginning(data);

} else {

newNode->prev = TAIL; TAIL->next = newNode; TAIL = newNode;

}

}

void insertAtIndex(int data, int index) {

Node \*newNode = (Node\*) malloc(sizeof(Node)); newNode->data = data;

Node \*current = HEAD;

while (index > 0) {

current = current->next; index--;

}

if (current->next == NULL) { newNode->next = NULL; current->next = newNode; current->prev = TAIL; TAIL = newNode;

} else {

newNode->prev = current; newNode->next = current->next; current->next->prev = newNode; current->next = newNode;

}

}

void deleteAtBeginning() { Node \*temp = HEAD;

HEAD->next->prev = NULL; HEAD = HEAD->next;

free(temp);

}

void deleteAtEnd() { Node \*temp = TAIL;

TAIL->prev->next = NULL; TAIL = TAIL->prev;

free(temp);

}

void deleteAtIndex(int index) { Node \*current = HEAD;

while (index > 0) {

current = current->next; index--;

}

current->prev->next = current->next; current->next->prev = current->prev; free(current);

}

void printList() {

Node \*current = HEAD;

printf("\n[");

while (current != NULL) { printf("%d ", current->data); current = current->next;

}

printf("]");

}

int main() {

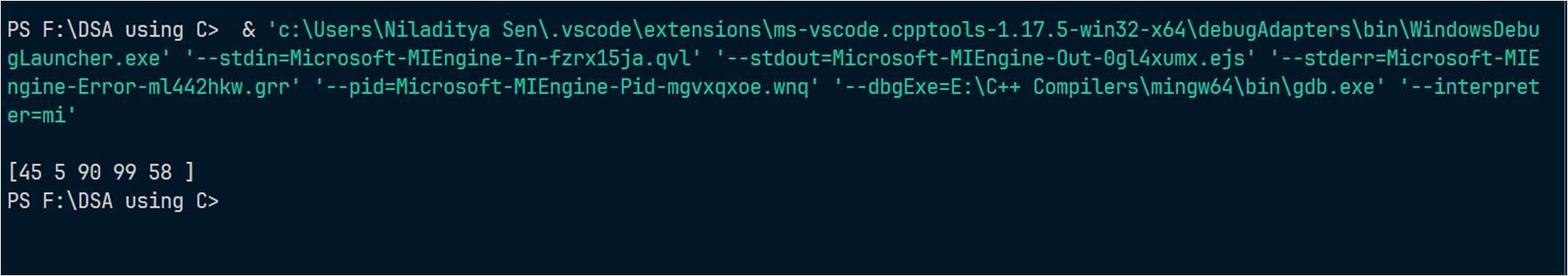
insertAtBeginning(12); insertAtBeginning(5); insertAtBeginning(45); insertAtBeginning(10);

insertAtEnd(99); insertAtEnd(58); insertAtEnd(65); insertAtIndex(90, 3); deleteAtIndex(3); deleteAtBeginning(); deleteAtEnd(); printList();

return 0;

}

**Output:**



# Experiment 3

**Aim: Write C programs to implement a double ended queue ADT using**

1. **Array**
2. **Doubly linked list respectively**

**Solution:**

1. **Array** #include <stdio.h> #define MAX 10

int front = 0, rear = 0; int queue[MAX];

int isFull() {

return MAX == rear;

}

int isEmpty() {

return front == rear;

}

void enqueue(int data) { if (isFull()) {

printf("Overflow"); return;

}

queue[rear] = data;

rear++;

}

void dequeue() {

if (isEmpty()) { printf("Empty Queue\n"); return;

}

front++;

}

int peek() {

if (isEmpty()) { printf("Empty Queue\n"); return -999;

}

return queue[front];

}

void printQueue() { if (isEmpty()) {

printf("Empty Queue\n"); return;

}

printf("\n[");

for (int i = front; i < rear; i++) { if (i == rear - 1) {

printf("%d", queue[i]);

} else {

printf("%d, ", queue[i]);

}

}

printf("]\n");

}

int main() {

for (int i = 1; i <= 10; i++) enqueue(i); printQueue();

for (int i = 1; i <= 11; i++) dequeue(); printQueue();

return 0;

}

**Output:**



1. **Doubly linked list respectively**

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

typedef struct NodeType { int data;

struct NodeType \*next;

} Node;

Node \*HEAD = NULL;

void enqueue(int data) {

Node \*newNode = (Node\*) malloc(sizeof(Node)); newNode->data = data;

newNode->next = HEAD; HEAD = newNode;

}

void dequeue() {

Node \*temp = HEAD; HEAD = HEAD->next;

free(temp);

}

int peek() {

return HEAD->data;

}

int isEmpty() {

return HEAD == NULL;

}

int isFull() {

Node \*newNode = (Node\*) malloc(sizeof(Node));

return newNode == NULL;

}

void printQueue() {

Node \*current = HEAD;

printf("[");

while (current != NULL) { printf("%d ", current->data); current = current->next;

}

printf("]");

}

int main() {

printf("%d", isEmpty());

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

}

printf("\n"); printQueue();

printf("\n"); dequeue(); printQueue(); printf("\n"); printf("%d", peek());

}

# Output:



**Experiment 4**

**Write a C program that uses functions to perform the following:**

* 1. **Create a binary search tree of characters.**
  2. **Traverse the above Binary search tree recursively in postorder.**

**Solution:**

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

typedef struct NodeType

{

char data;

struct NodeType \*left; struct NodeType \*right;

} Node;

Node \*create(char item)

{

Node \*node = (Node \*)malloc(sizeof(Node \*)); node->data = item;

node->left = node->right = NULL;

}

Node \*insertion(Node \*node, char item)

{

if (node == NULL)

return create(item);

if (item < node->data)

node->left = insertion(node->left, item);

else

node->right = insertion(node->right, item);

return node;

}

void postorder(Node \*node)

{

if (node == NULL)

{

return;

}

postorder(node->left); postorder(node->right); printf("%c, ", node->data);

}

int main()

{

Node \*root = NULL;

for (char i = 'A'; i <= 'E'; i++)

{

root = insertion(root, i);

}

postorder(root);

return 0;

}

**Output:**



# Experiment 5

**Write a C program that uses functions to perform the following:**

1. **Create a binary search tree of integers.**
2. **Traverse the above Binary search tree recursively in inorder.**

**Solution:**

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

typedef struct NodeType

{

int data;

struct NodeType \*left; struct NodeType \*right;

} Node;

Node \*create(int item)

{

Node \*node = (Node \*)malloc(sizeof(Node \*)); node->data = item;

node->left = node->right = NULL;

}

Node \*insertion(Node \*node, int item)

{

if (node == NULL)

return create(item);

if (item < node->data)

node->left = insertion(node->left, item);

else

node->right = insertion(node->right, item);

return node;

}

void inorder(Node \*node)

{

if (node == NULL)

{

return;

}

inorder(node->left); printf("%d, ", node->data); inorder(node->right);

}

int main()

{

Node \*root = NULL;

for (int i = 1; i <= 5; i++)

{

root = insertion(root, i);

}

inorder(root);

return 0;

}

**Output:**



# Experiment 6

**Write C programs for implementing the following sorting methods to arrange a list of integers in ascending order:**

1. **Insertion sort**
2. **Merge sort.**

**Solution:**

* 1. **Insertion Sort**

#include <stdio.h>

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

int key = arr[i]; int j = i - 1;

while (j >= 0 && arr[j] > key) { arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

}

int main() {

int arr[] = {5, 2, 4, 6, 1, 3};

int length = sizeof(arr) / sizeof(arr[0]); insertionSort(arr, length);

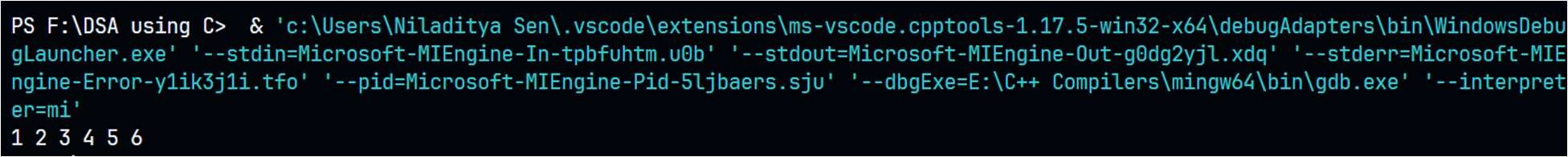
for (int i = 0; i < length; i++) { printf("%d ", arr[i]);

}

return 0;

}

**Output:**



* 1. **Merge Sort**

#include <stdio.h>

void printArray(int arr[], int length);

void merge(int arr[], int left, int middle, int right); void mergeSort(int arr[], int left, int right);

void mergeSort(int arr[], int left, int right) { if (left < right) {

int middle = (left + right) / 2; mergeSort(arr, left, middle); mergeSort(arr, middle + 1, right); merge(arr, left, middle, right);

}

}

void merge(int arr[], int left, int middle, int right) { int leftArrLen = middle - left + 1;

int rightArrLen = right - middle;

int leftArr[leftArrLen], rightArr[rightArrLen]; for (int i = 0; i < leftArrLen; i++) {

leftArr[i] = arr[left + i];

}

for (int i = 0; i < rightArrLen; i++) { rightArr[i] = arr[middle + 1 + i];

}

int i = 0, j = 0, k = left;

while (i < leftArrLen && j < rightArrLen) { if (leftArr[i] <= rightArr[j]) {

arr[k] = leftArr[i]; i++;

}

else {

arr[k] = rightArr[j]; j++;

} k++;

}

while (i < leftArrLen) { arr[k] = leftArr[i]; i++;

k++;

}

while (j < rightArrLen) { arr[k] = rightArr[j]; j++;

k++;

}

}

void printArray(int arr[], int length) { for (int i = 0; i < length; i++) {

printf("%d ", arr[i]);

}

printf("\n");

}

int main() {

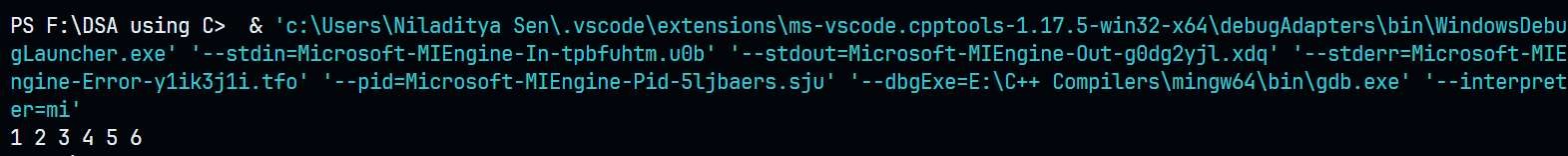
int arr[] = {5, 2, 4, 6, 1, 3};

int length = sizeof(arr) / sizeof(arr[0]); mergeSort(arr, 0, length - 1);

printArray(arr, length); return 0;

}

**Output:**



# Experiment 7

**Write C programs for implementing the following sorting methods to arrange a list of integers in ascending order:**

1. **Quick sort**
2. **Selection sort.**

**Solution:**

* 1. **Quick Sort**

#include <stdio.h>

int partition(int arr[], int left, int right) { int pivot = arr[right];

int i = left - 1;

for (int j = left; j < right; j++) { if (arr[j] < pivot) {

i++;

int temp = arr[j]; arr[j] = arr[i]; arr[i] = temp;

}

}

int temp = arr[right]; arr[right] = arr[i + 1]; arr[i + 1] = temp; return i + 1;

}

void quickSort(int arr[], int left, int right) { if (left < right) {

int pivot = partition(arr, left, right);

quickSort(arr, left, pivot - 1); quickSort(arr, pivot + 1, right);

}

}

int main() {

int arr[] = {5, 4, 3, 2, 1};

int length = sizeof(arr) / sizeof(arr[0]); printf("Unsorted array: ");

for (int i = 0; i < length; i++) { printf("%d ", arr[i]);

}

printf("\n");

quickSort(arr, 0, length - 1); printf("Sorted array: ");

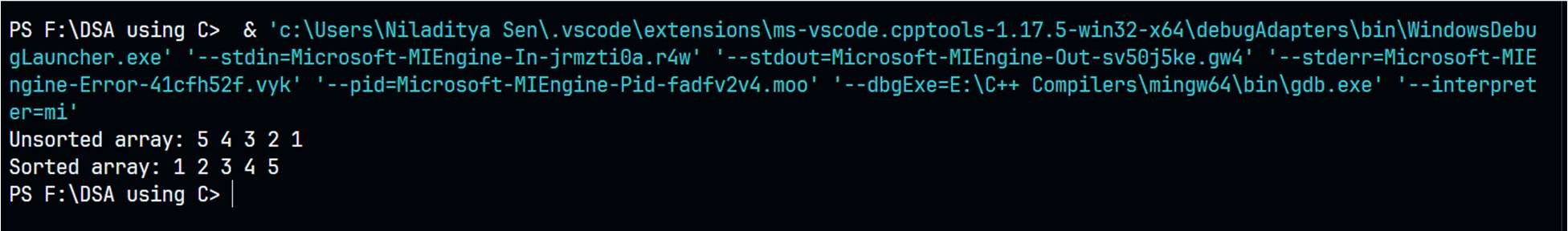
for (int i = 0; i < length; i++) { printf("%d ", arr[i]);

}

printf("\n"); return 0;

}

**Output:**



* 1. **Selection Sort**

#include <stdio.h>

void selectionSort(int arr[], int length) { for (int i = 0; i < length - 1; i++) {

int minIdx = i;

for (int j = i + 1; j < length; j++) { if (arr[j] < arr[minIdx]) {

minIdx = j;

}

}

int temp = arr[i]; arr[i] = arr[minIdx]; arr[minIdx] = temp;

}

}

void printArray(int arr[], int length) { for (int i = 0; i < length; i++) {

printf("%d ", arr[i]);

}

printf("\n");

}

int main() {

int arr[] = { 5, 4, 3, 2, 1 };

int length = sizeof(arr) / sizeof(arr[0]); printf("Original array: "); printArray(arr, length); selectionSort(arr, length);

printf("Sorted array: ");

printArray(arr, length); return 0;

}

**Output:**



# Experiment 8

**Aim: Write a program to perform the following operation:**

1. **Insertion into a B-tree**
2. **Write a C program for implementing Heap sort algorithm for sorting a given list of integers in ascending order.**

**Solution:**

1. **Insertion into a B-tree** #include <stdio.h> #include <stdlib.h> struct BTreeNode

{

int \*keys; int t;

struct BTreeNode \*\*C; int n;

int leaf;

};

struct BTreeNode \*createNode(int t, int leaf)

{

struct BTreeNode \*newNode = (struct BTreeNode

\*)malloc(sizeof(struct BTreeNode)); newNode->t = t;

newNode->leaf = leaf;

newNode->keys = (int \*)malloc(sizeof(int) \* (2 \* t - 1)); newNode->C = (struct BTreeNode \*\*)malloc(sizeof(struct BTreeNode

\*) \* (2 \* t)); newNode->n = 0; return newNode;

}

void splitChild(struct BTreeNode \*x, int i, struct BTreeNode \*y)

{

struct BTreeNode \*z = createNode(y->t, y->leaf); z->n = x->t - 1;

for (int j = 0; j < x->t - 1; j++) z->keys[j] = y->keys[j + x->t];

if (y->leaf == 0)

{

for (int j = 0; j < x->t; j++) z->C[j] = y->C[j + x->t];

}

y->n = x->t - 1;

for (int j = x->n; j >= i + 1; j--) x->C[j + 1] = x->C[j];

x->C[i + 1] = z;

for (int j = x->n - 1; j >= i; j--) x->keys[j + 1] = x->keys[j];

x->keys[i] = y->keys[x->t - 1]; x->n = x->n + 1;

}

void insertNonFull(struct BTreeNode \*x, int k)

{

int i = x->n - 1; if (x->leaf == 1)

{

while (i >= 0 && x->keys[i] > k)

{

x->keys[i + 1] = x->keys[i]; i--;

}

}

else

{

x->keys[i + 1] = k; x->n = x->n + 1;

while (i >= 0 && x->keys[i] > k) i--;

if (x->C[i + 1]->n == 2 \* x->t - 1)

{

splitChild(x, i + 1, x->C[i + 1]); if (x->keys[i + 1] < k)

i++;

}

insertNonFull(x->C[i + 1], k);

}

}

void insert(struct BTreeNode \*\*root, int k)

{

if (\*root == NULL)

{

}

else

\*root = createNode(2, 1); (\*root)->keys[0] = k; (\*root)->n = 1;

{

if ((\*root)->n == 2 \* (\*root)->t - 1)

{

}

else

}

}

struct BTreeNode \*s = createNode((\*root)->t, 0); s->C[0] = \*root;

splitChild(s, 0, \*root); int i = 0;

if (s->keys[0] < k) i++;

insertNonFull(s->C[i], k);

\*root = s;

insertNonFull(\*root, k);

void traverse(struct BTreeNode \*root)

{

int i;

for (i = 0; i < root->n; i++)

{

if (root->leaf == 0) traverse(root->C[i]);

printf(" %d", root->keys[i]);

}

if (root->leaf == 0) traverse(root->C[i]);

}

int main()

{

struct BTreeNode \*root = NULL; int n, i, x;

printf("Enter the number of elements to be inserted: "); scanf("%d", &n);

printf("Enter the elements to be inserted: "); for (i = 0; i < n; i++)

{

scanf("%d", &x); insert(&root, x);

}

printf("Traversal of the constucted tree is "); traverse(root);

return 0;

}

Output:



1. **Heap Sort**

#include <stdio.h>

void swap(int \*a, int \*b)

{

int temp = \*a;

\*a = \*b;

\*b = temp;

}

void heapify(int arr[], int length, int i)

{

int largest = i;

int left = 2 \* i + 1; int right = 2 \* i + 2;

if (left < length && arr[left] > arr[largest])

{

largest = left;

}

if (right < length && arr[right] > arr[largest])

{

largest = right;

}

if (largest != i)

{

swap(&arr[i], &arr[largest]); heapify(arr, length, largest);

}

}

void heapSort(int arr[], int length)

{

for (int i = length / 2 - 1; i >= 0; i--)

{

heapify(arr, length, i);

}

for (int i = length - 1; i >= 0; i--)

{

swap(&arr[0], &arr[i]); heapify(arr, i, 0);

}

}

void printArray(int arr[], int length)

{

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

{

printf("%d ", arr[i]);

}

printf("\n");

}

int main()

{

int arr[] = {5, 4, 3, 2, 1};

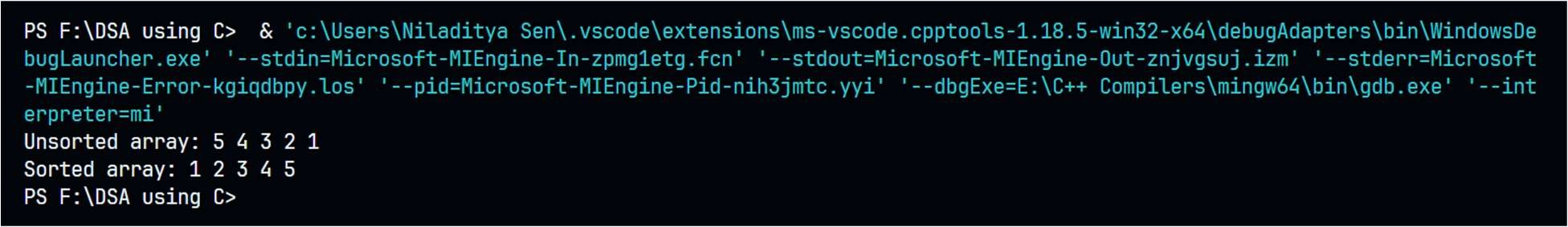
int length = sizeof(arr) / sizeof(arr[0]); printf("Unsorted array: "); printArray(arr, length);

heapSort(arr, length); printf("Sorted array: ");

printArray(arr, length); return 0;

}

Output:



# Experiment 9

**Aim: Write a C program to implement all the functions of a dictionary (ADT) using hashing.**

**Solution:**

#include <iostream> #include <stdlib.h>

using namespace std;

#define max 10

typedef struct list

{

int data;

struct list \*next;

} node\_type;

node\_type \*ptr[max], \*root[max], \*temp[max];

class Dictionary

{

public:

int index; Dictionary(); void insert(int); void search(int);

void delete\_ele(int);

};

Dictionary::Dictionary()

{

index = -1;

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

{

root[i] = NULL; ptr[i] = NULL; temp[i] = NULL;

}

}

void Dictionary::insert(int key)

{

index = int(key % max);

ptr[index] = (node\_type \*)malloc(sizeof(node\_type)); ptr[index]->data = key;

if (root[index] == NULL)

{

}

else

{

}

root[index] = ptr[index]; root[index]->next = NULL; temp[index] = ptr[index];

temp[index] = root[index];

while (temp[index]->next != NULL) temp[index] = temp[index]->next;

temp[index]->next = ptr[index];

}

void Dictionary::search(int key)

{

int flag = 0;

index = int(key % max); temp[index] = root[index]; while (temp[index] != NULL)

{

if (temp[index]->data == key)

{

}

else

}

cout << "\nSearch key is found!!"; flag = 1;

break;

temp[index] = temp[index]->next;

if (flag == 0)

cout << "\nsearch key not found ";

}

void Dictionary::delete\_ele(int key)

{

index = int(key % max); temp[index] = root[index];

while (temp[index]->data != key && temp[index] != NULL)

{

ptr[index] = temp[index];

temp[index] = temp[index]->next;

}

ptr[index]->next = temp[index]->next; cout << "\n"

<< temp[index]->data << " has been deleted."; temp[index]->data = -1;

temp[index] = NULL; free(temp[index]);

}

int main()

{

int val, ch, n, num; char c;

Dictionary d; do

{

cout << "\nMENU:\n1.Create";

cout << "\n2.Search for a value\n3.Delete an value"; cout << "\nEnter your choice:";

cin >> ch; switch (ch)

{

case 1:

cout << "\nEnter the number of elements to be inserted:"; cin >> n;

cout << "\nEnter the elements to be inserted:"; for (int i = 0; i < n; i++)

{

cin >> num; d.insert(num);

}

break; case 2:

cout << "\nEnter the element to be searched:"; cin >> n;

d.search(n); case 3:

cout << "\nEnter the element to be deleted:"; cin >> n;

d.delete\_ele(n); break;

default:

cout << "\nInvalid Choice.";

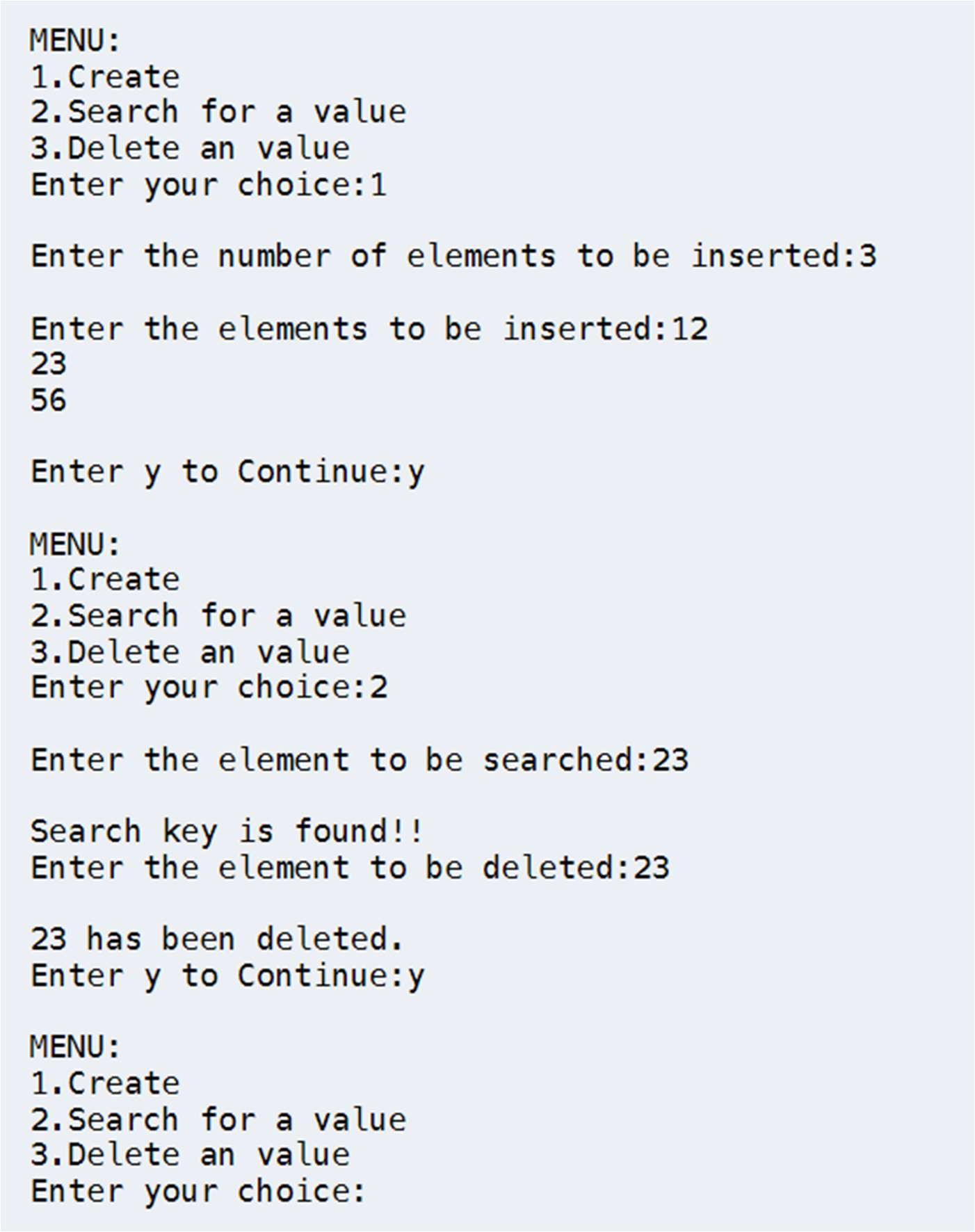
}

cout << "\nEnter y to Continue:"; cin >> c;

} while (c == 'y');

}

**Output:**



# Experiment 10

**Write C programs for implementing the following graph traversal algorithms:**

* 1. **Depth ﬁrst traversal**
  2. **Breadth ﬁrst traversal. Solution:**

1. **Depth ﬁrst traversal** #include <stdio.h> #include <stdlib.h> struct node {

int vertex;

struct node\* next;

};

struct node\* createNode(int v); struct Graph {

int numVertices; int\* visited;

struct node\*\* adjLists;

};

void DFS(struct Graph\* graph, int vertex) { struct node\* adjList = graph->adjLists[vertex]; struct node\* temp = adjList;

graph->visited[vertex] = 1; printf("Visited %d \n", vertex); while (temp != NULL) {

int connectedVertex = temp->vertex;

if (graph->visited[connectedVertex] == 0) { DFS(graph, connectedVertex);

}

temp = temp->next;

}

}

struct node\* createNode(int v) {

struct node\* newNode = malloc(sizeof(struct node)); newNode->vertex = v;

newNode->next = NULL; return newNode;

}

struct Graph\* createGraph(int vertices) {

struct Graph\* graph = malloc(sizeof(struct Graph)); graph->numVertices = vertices;

graph->adjLists = malloc(vertices \* sizeof(struct node\*)); graph->visited = malloc(vertices \* sizeof(int));

int i;

for (i = 0; i < vertices; i++) { graph->adjLists[i] = NULL; graph->visited[i] = 0;

}

return graph;

}

void addEdge(struct Graph\* graph, int src, int dest) { struct node\* newNode = createNode(dest);

newNode->next = graph->adjLists[src]; graph->adjLists[src] = newNode;

newNode = createNode(src);

newNode->next = graph->adjLists[dest]; graph->adjLists[dest] = newNode;

}

void printGraph(struct Graph\* graph) { int v;

for (v = 0; v < graph->numVertices; v++) { struct node\* temp = graph->adjLists[v];

printf("\n Adjacency list of vertex %d\n ", v); while (temp) {

printf("%d -> ", temp->vertex); temp = temp->next;

}

printf("\n");

}

}

int main() {

struct Graph\* graph = createGraph(4); addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

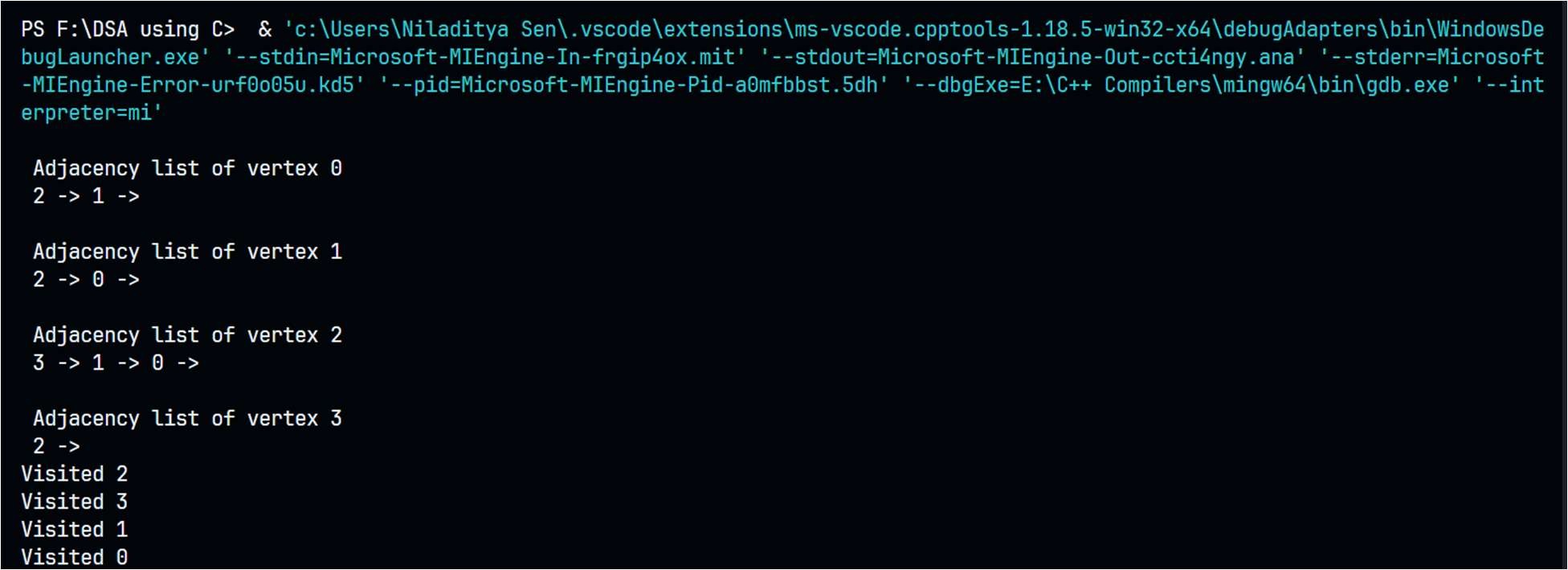
addEdge(graph, 1, 2);

addEdge(graph, 2, 3); printGraph(graph); DFS(graph, 2);

return 0;

}

**Output:**



1. **Breadth First Search** #include <stdio.h> #include <stdlib.h> #define SIZE 40 struct queue {

int items[SIZE]; int front;

int rear;

};

struct queue\* createQueue();

void enqueue(struct queue\* q, int); int dequeue(struct queue\* q);

void display(struct queue\* q); int isEmpty(struct queue\* q); void printQueue(struct queue\* q); struct node {

int vertex;

struct node\* next;

};

struct node\* createNode(int); struct Graph {

int numVertices;

struct node\*\* adjLists; int\* visited;

};

void bfs(struct Graph\* graph, int startVertex) { struct queue\* q = createQueue();

graph->visited[startVertex] = 1; enqueue(q, startVertex);

while (!isEmpty(q)) { printQueue(q);

int currentVertex = dequeue(q); printf("Visited %d\n", currentVertex);

struct node\* temp = graph->adjLists[currentVertex]; while (temp) {

int adjVertex = temp->vertex;

if (graph->visited[adjVertex] == 0) { graph->visited[adjVertex] = 1; enqueue(q, adjVertex);

}

temp = temp->next;

}

}

}

struct node\* createNode(int v) {

struct node\* newNode = malloc(sizeof(struct node)); newNode->vertex = v;

newNode->next = NULL; return newNode;

}

struct Graph\* createGraph(int vertices) {

struct Graph\* graph = malloc(sizeof(struct Graph)); graph->numVertices = vertices;

graph->adjLists = malloc(vertices \* sizeof(struct node\*)); graph->visited = malloc(vertices \* sizeof(int));

int i;

for (i = 0; i < vertices; i++) { graph->adjLists[i] = NULL; graph->visited[i] = 0;

}

return graph;

}

void addEdge(struct Graph\* graph, int src, int dest) { struct node\* newNode = createNode(dest);

newNode->next = graph->adjLists[src]; graph->adjLists[src] = newNode; newNode = createNode(src);

newNode->next = graph->adjLists[dest]; graph->adjLists[dest] = newNode;

}

struct queue\* createQueue() {

struct queue\* q = malloc(sizeof(struct queue)); q->front = -1;

q->rear = -1; return q;

}

int isEmpty(struct queue\* q) { if (q->rear == -1)

return 1; else

return 0;

}

void enqueue(struct queue\* q, int value) {

if (q->rear == SIZE - 1) printf("\nQueue is Full!!");

else {

if (q->front == -1) q->front = 0;

q->rear++;

q->items[q->rear] = value;

}

}

int dequeue(struct queue\* q) { int item;

if (isEmpty(q)) { printf("Queue is empty"); item = -1;

} else {

item = q->items[q->front]; q->front++;

if (q->front > q->rear) { printf("Resetting queue "); q->front = q->rear = -1;

}

}

return item;

}

void printQueue(struct queue\* q) { int i = q->front;

if (isEmpty(q)) { printf("Queue is empty");

} else {

printf("\nQueue contains \n");

for (i = q->front; i < q->rear + 1; i++) { printf("%d ", q->items[i]);

}

}

}

int main() {

struct Graph\* graph = createGraph(6);

|  |  |  |
| --- | --- | --- |
| addEdge(graph, | 0, | 1); |
| addEdge(graph, | 0, | 2); |
| addEdge(graph, | 1, | 2); |
| addEdge(graph, | 1, | 4); |
| addEdge(graph, | 1, | 3); |
| addEdge(graph, | 2, | 4); |
| addEdge(graph, | 3, | 4); |
| bfs(graph, 0); |  |  |
| return 0; |  |  |

}

**Output:**

