

**DEPARTMENT OF COMPUTER SCIENCE AND APPLICATIONS**

**Practical Record**

**on**

**Data Structures**

***Submitted in the partial fulfilment for the award of degree in***

## **MASTER OF COMPUTER APPLICATIONS**

## **Submitted by**

**Yashwanth kumar J**

**P03ZW24S126028**

Under the guidance of

PROF. Annie Christella

Assistant Professor

DEPARTMENT OF COMPUTER SCIENCE AND APPLICATIONS

March - 2025



**Certificate**

This is to certify that Smt/Sri **YASHWANTH KUMAR J** Register Number **P03ZW24S126028** Class **MCA 1ST** semester have successfully completed **Data Structures Lab** Practical Record as prescribed by the College for the Academic year **2024-2025**

**Teacher In Charge Signature of HOD**

**Internal Examiner External Examiner**

**Date: Seal**

**Index**

|  |  |
| --- | --- |
| Sl. No | Program Name |
| 1 | **Linear Search and Binary Search** |
| 2 | **Sorting Procedures (Selection, Bubble, Insertion Sort)** |
| 3 | **Polynomial Addition using Arrays** |
| 4 | **Sparse Matrix Manipulation using Arrays** |
| 5 | **Stack using Arrays** |
| 6 | **Queue using Arrays** |
| 7 | **Circular Queue using Arrays** |
| 8 | **Singly Linked List** |
| 9 | **Polynomial Addition using Linked Lists** |
| 10 | **Queue using Linked Lists** |
| 11 | **Binary Search Tree Traversal using Recursion** |
| 12 | **Graph Representation using Arrays** |
| 13 | **Infix to Postfix Conversion** |
| 14 | **Evaluation of Postfix Expressions** |
| 15 | **Doubly Linked List** |
| 16 | **Circular Linked List** |
| 17 | **Graph using Linked List** |
| 18 | **2D Array allocation Dynamically** |
| 19 | **Demonstrating Realloc Function** |
| 20 | **Binary Search Tree Traversal Without Recursion** |

1. **Program to represent Linear Search and Binary Search.**

**Program:**

#include <stdio.h>

void linearSearch(int arr[], int n, int key)

{

int i;

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

{

if (arr[i] == key)

{

printf("\nThe element found at %d:\n", i);

return;

}

}

printf("Element not found!!!\n");

}

void binarySearch(int arr[], int n, int key)

{

int start, end, mid;

start = 0, end = n - 1;

while (start <= end)

{

mid = (start + end) / 2;

if (arr[mid] == key)

{

printf("The Element is found at: %d\n", mid);

return;

}

else if (arr[mid] > key)

{

end = mid - 1;

} else

{

start = mid + 1;

}

}

printf("Element not found!!!");

}

int main()

{

int choice, n, key,i;

printf("Enter the number of elements in the array:");

scanf("%d", &n);

int arr[n];

printf("Enter the Elements of the Array

(in Sorted Manner):”);

for (i = 0; i < n; i++) {

scanf("%d", &arr[i]);

}

printf("Enter the element to be searched:");

scanf("%d", &key);

printf("\nEnter 1 for Linear Search\n

Enter 2 for Binary Search\n");

scanf("%d", &choice);

switch (choice) {

case 1:

linearSearch(arr, n, key);

break;

case 2:

binarySearch(arr, n, key);

break;

default:

printf("Invalid Choice!!!\n");

}

return 0;

}

**Output:**

Enter the number of elements in the array:5

Enter the Elements of the Array (in Sorted Manner)

1

2

3

4

5

Enter the element to be searched:6

Enter 1 for Linear Search

Enter 2 for Binary Search

1

Element not found!!!

Enter the number of elements in the array:5

Enter the Elements of the Array (in Sorted Manner):

1

2

3

4

5

Enter the element to be searched:5

Enter 1 for Linear Search

Enter 2 for Binary Search

2

The Element is found at: 4

1. **Program to represent sorting procedures (Selection Sort, Bubble Sort, and Insertion Sort).**

**Program:**

#include <stdio.h>

void bubbleSort(int array[], int n)

{

int i, j, temp;

for (i = 0; i < n - 1; i++)

{

for (j = 0; j < n - i - 1; j++)

{

if (array[j] > array[j + 1])

{

temp = array[j];

array[j] = array[j + 1];

array[j + 1] = temp;

}

}

}

}

void selectionSort(int array[], int n)

{

int i, j, minimumIndex, temp;

for (i = 0; i < n - 1; i++)

{

minimumIndex = i;

for (j = i + 1; j < n; j++)

{

if (array[j] < array[minimumIndex])

{

minimumIndex = j;

}

}

temp = array[minimumIndex];

array[minimumIndex] = array[i];

array[i] = temp;

}

}

void insertionSort(int array[], int n)

{

int i, key, j;

for (i = 1; i < n; i++)

{

key = array[i];

j = i - 1;

while (j >= 0 && array[j] > key)

{

array[j + 1] = array[j];

j = j - 1;

}

array[j + 1] = key;

}

}

void printArray(int array[], int n)

{

int i;

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

{

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

}

printf("\n");

}

int main() {

int n, i;

int choice;

printf("Enter the number of elements: ");

scanf("%d", &n);

int array[n];

printf("Enter the elements:\n");

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

{

scanf("%d", &array[i]);

}

printf("\nUnsorted array: \n");

printArray(array, n);

printf("\nChoose a sorting algorithm:\n");

printf("1. Bubble Sort\n");

printf("2. Selection Sort\n");

printf("3. Insertion Sort\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

bubbleSort(array, n);

printf("\nSorted array (Bubble Sort): \n");

printArray(array, n);

break;

case 2:

selectionSort(array, n);

printf("\nSorted array (Selection Sort): \n");

printArray(array, n);

break;

case 3:

insertionSort(array, n);

printf("\nSorted array (Insertion Sort): \n");

printArray(array, n);

break;

default:

printf("\nInvalid choice.\n");

}

return 0;

}

**Output:**

Enter the number of elements: 5

Enter the elements:

5

9

3

0

1

Unsorted array:

5 9 3 0 1

Choose a sorting algorithm:

1. Bubble Sort

2. Selection Sort

3. Insertion Sort

Enter your choice: 1

Sorted array (Bubble Sort):

0 1 3 5 9

1. **Polynomial addition using arrays.**

**Program:**

#include <stdio.h>

void addPolynomials(int poly1[], int poly2[], int result[], int n) {

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

result[i] = poly1[i] + poly2[i];

}

}

void printPolynomial(int poly[], int n) {

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

printf("%dx^%d", poly[i], i);

if (i != n - 1) {

printf(" + ");

}

}

printf("\n");

}

int main() {

int n = 3;

int poly1[] = {1, 2, 3};

int poly2[] = {3, 4, 5};

int result[n];

addPolynomials(poly1, poly2, result, n);

printf("Polynomial 1: ");

printPolynomial(poly1, n);

printf("Polynomial 2: ");

printPolynomial(poly2, n);

printf("Sum: ");

printPolynomial(result, n);

return 0;

}

**Output:**

Polynomial 1: 1x^0 + 2x^1 + 3x^2

Polynomial 2: 3x^0 + 4x^1 + 5x^2

Sum: 4x^0 + 6x^1 + 8x^2

1. **Sparse matrix manipulation using arrays.**

**Program:**

#include <stdio.h>

int main()

{

int sparse\_matrix[4][5] =

{

{0 , 0 , 6 , 0 , 9 },

{0 , 0 , 4 , 6 , 0 },

{0 , 0 , 0 , 0 , 0 },

{0 , 1 , 2 , 0 , 0 }

};

int size = 0;

for(int i=0; i<4; i++)

{

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

{

if(sparse\_matrix[i][j]!=0)

{

size++;

}

}

}

int matrix[3][size];

int k=0;

for(int i=0; i<4; i++)

{

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

{

if(sparse\_matrix[i][j]!=0)

{

matrix[0][k] = i;

matrix[1][k] = j;

matrix[2][k] = sparse\_matrix[i][j];

k++;

}

}

}

for(int i=0 ;i<3; i++)

{

for(int j=0; j<size; j++)

{

printf("%d ", matrix[i][j]);

printf("\t");

}

printf("\n");

}

return 0;

}

**Output:**

0 0 1 1 3 3

2 4 2 3 1 2

6 9 4 6 1 2

1. **Stack using arrays.**

**Program:**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 10

int A[MAX\_SIZE];

int top = -1;

void Push(int x)

{

if (top == MAX\_SIZE - 1)

{

printf("Error: stack overflow\n");

return;

}

A[++top] = x;

}

void Pop()

{

if (top == -1)

{

printf("Error: No element to pop\n");

return;

}

top--;

}

void Print()

{

int i;

printf("Stack: ");

for (i = 0; i <= top; i++)

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

printf("\n");

}

int IsEmpty()

{

if (top == -1)

return 1;

else

return 0;

}

int main()

{

Push(2);

Print();

Push(5);

Print();

Push(10);

Print();

Pop();

Print();

Push(12);

Print();

return 0;

}

**Output:**

Stack: 2

Stack: 2 5

Stack: 2 5 10

Stack: 2 5

Stack: 2 5 12

1. **Queue using arrays.**

**Program:**

#include <stdio.h>

#define MAX\_SIZE 10

int A[MAX\_SIZE];

int front = -1;

int rear = -1;

void Enqueue(int x)

{

if (rear == MAX\_SIZE - 1)

{

printf("Error: Queue is full\n");

return;

}

if (front == -1 && rear == -1)

{

front = rear = 0;

}

else

{

rear++;

}

A[rear] = x;

}

void Dequeue()

{

if (front == -1 && rear == -1)

{

printf("Error: Queue is empty\n");

return;

}

else if (front == rear)

{

front = rear = -1;

}

else

{

front++;

}

}

}

void Print()

{

int i;

printf("Queue: ");

for (i = front; i <= rear; i++)

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

printf("\n");

}

int IsEmpty()

{

if (front == -1 && rear == -1)

return 1;

else

return 0;

}

int main()

{

Enqueue(2);

Print();

Enqueue(5);

Print();

Enqueue(10);

Print();

Dequeue();

Print();

Enqueue(12);

Print();

return 0;

}

**Output:**

Queue: 2

Queue: 2 5

Queue: 2 5 10

Queue: 5 10

Queue: 5 10 12

1. **Circular Queue using arrays.**

**Program:**

#include <stdio.h>

#define MAX\_SIZE 5

int A[MAX\_SIZE];

int front = -1;

int rear = -1;

void Enqueue(int x)

{

if ((rear + 1) % MAX\_SIZE == front)

{

printf("Error: Queue is full\n");

return;

}

else if (front == -1 && rear == -1)

{

front = rear = 0;

}

else

{

rear = (rear + 1) % MAX\_SIZE;

}

A[rear] = x;

}

void Dequeue()

{

if (front == -1 && rear == -1)

{

printf("Error: Queue is empty\n");

return;

}

else if (front == rear)

{

front = rear = -1;

}

else

{

front = (front + 1) % MAX\_SIZE;

}

}

int Front()

{

if (front == -1 && rear == -1)

{

printf("Error: Queue is empty\n");

return -1;

}

return A[front];

}

void Print()

{

int i = front;

if (front == -1 && rear == -1)

{

printf("Queue is empty\n");

return;

}

printf("Queue: ");

while (i != rear)

{

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

i = (i + 1) % MAX\_SIZE;

}

printf("%d\n", A[rear]);

}

int IsEmpty()

{

if (front == -1 && rear == -1)

return 1;

else

return 0;

}

int main()

{

Enqueue(2);

Print();

Enqueue(5);

Print();

Enqueue(10);

Print();

Dequeue();

Print();

Enqueue(12);

Print();

Enqueue(15);

Print();

return 0;

}

**Output:**

Queue: 2

Queue: 2 5

Queue: 2 5 10

Queue: 5 10

Queue: 5 10 12

Queue: 5 10 12 15

1. **Program to represent Singly Linked List.**

**Program:**

#include <stdio.h>

#include <stdlib.h>

struct node {

int INFO;

struct node \*LINK;

};

typedef struct node NODE;

NODE \*start = NULL;

void create() {

char ch;

int i = 0;

NODE \*CPTR, \*NEWNODE;

CPTR = (NODE \*)malloc(sizeof(NODE));

if (CPTR == NULL) {

printf("Memory allocation failed.\n");

exit(1);

}

start = CPTR;

while (1) {

printf("\nEnter the node %d: ", i + 1);

scanf("%d", &CPTR->INFO);

printf("\nDo you wish to add one more node (Y/N): ");

scanf(" %c", &ch);

if (ch == 'Y' || ch == 'y') {

NEWNODE = (NODE \*)malloc(sizeof(NODE));

if (NEWNODE == NULL) {

printf("Memory allocation failed.\n");

exit(1);

}

CPTR->LINK = NEWNODE;

CPTR = NEWNODE;

} else {

CPTR->LINK = NULL;

break;

}

i++;

}

}

void display() {

NODE \*CPTR = start;

if (start == NULL) {

printf("\nLinked list is empty\n");

return;

}

while (CPTR != NULL) {

printf("%d -> ", CPTR->INFO);

CPTR = CPTR->LINK;

}

printf("NULL\n");

}

int length() {

int len = 0;

NODE \*CPTR = start;

if (start == NULL) {

printf("The linked list is empty\n");

return 0;

}

while (CPTR != NULL) {

len++;

CPTR = CPTR->LINK;

}

return len;

}

void search(int ITEM) {

int i = 0;

NODE \*CPTR = start;

while (CPTR != NULL) {

i++;

if (ITEM == CPTR->INFO) {

printf("\nThe item %d is found at position no: %d\n", ITEM, i);

return;

}

CPTR = CPTR->LINK;

}

printf("\nThe item does not exist in the linked list.\n");

}

int main() {

int ITEM;

printf("\n\tCreation of linked list\n");

create();

printf("\nThe linked list created is:\n");

display();

printf("\nThe number of nodes in the linked list is: %d\n", length());

printf("\nEnter the item to be searched: ");

scanf("%d", &ITEM);

search(ITEM);

return 0;

}

}

**Output:**

Creation of linked list

Enter the node 1: 10

Do you wish to add one more node (Y/N): y

Enter the node 2: 19

Do you wish to add one more node (Y/N): y

Enter the node 3: 1

Do you wish to add one more node (Y/N): n

The linked list created is:

10 -> 19 -> 1 -> NULL

The number of nodes in the linked list is: 3

Enter the item to be searched: 1

The item 1 is found at position no: 3

1. **Polynomial addition using linked lists.**

**Program:**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int coeff;

int pow;

struct Node\* next;

};

struct Node\* createNode(int c, int p);

struct Node\* addPolynomial(struct Node\* head1, struct Node\* head2) {

if (head1 == NULL) return head2;

if (head2 == NULL) return head1;

.

if (head1->pow > head2->pow) {

struct Node\* nextPtr =

addPolynomial(head1->next, head2);

head1->next = nextPtr;

return head1;

}

else if (head1->pow < head2->pow) {

struct Node\* nextPtr =

addPolynomial(head1, head2->next);

head2->next = nextPtr;

return head2;

}

struct Node\* nextPtr =

addPolynomial(head1->next, head2->next);

head1->coeff += head2->coeff;

head1->next = nextPtr;

return head1;

}

void printList(struct Node\* head) {

struct Node\* curr = head;

while (curr != NULL) {

printf("%d X ^ %d ", curr->coeff, curr->pow);

curr = curr->next;

}

printf("\n");

}

struct Node\* createNode(int c, int p) {

struct Node\* newNode =

(struct Node\*)malloc(sizeof(struct Node));

newNode->coeff = c;

newNode->pow = p;

newNode->next = NULL;

return newNode;

}

int main() {

struct Node\* head1 = createNode(5, 2);

head1->next = createNode(4, 1);

head1->next->next = createNode(2, 0);

struct Node\* head2 = createNode(-5, 1);

head2->next = createNode(-5, 0);

struct Node\* head = addPolynomial(head1, head2);

printList(head);

return 0;

}

**Output:**

5 X ^ 2 -1 X ^ 1 -3 X ^ 0

1. **Program to represent a Queue using linked lists.**

**Program:**

#include<stdio.h>

#include<stdlib.h>

struct node

{

int data;

struct node \*next;

};

struct node \*front;

struct node \*rear;

void insert();

void delete\_q();

void display();

void main ()

{

int choice;

while(choice != 4)

{

printf("\n\*\*Main Menu\n");

printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");

printf("\nEnter your choice ?");

scanf("%d",& choice);

switch(choice)

{

case 1:

insert();

break;

case 2:

delete\_q();

break;

case 3:

display();

break;

case 4:

exit(0);

break;

default:

printf("\nEnter valid choice??\n");

}

}

}

void insert()

{

struct node \*ptr;

int item;

ptr = (struct node \*) malloc (sizeof(struct node));

if(ptr == NULL)

{

printf("\nOVERFLOW\n");

return;

}

else

{

printf("\nEnter value?\n");

scanf("%d",&item);

ptr -> data = item;

if(front == NULL)

{

front = ptr;

rear = ptr;

front -> next = NULL;

rear -> next = NULL;

}

else

{

rear -> next = ptr;

rear = ptr;

rear->next = NULL;

}

}

}

void delete\_q ()

{

struct node \*ptr;

if(front == NULL)

{

printf("\nUNDERFLOW\n");

return;

}

else

{

ptr = front;

front = front -> next;

free(ptr);

}

}

void display()

{

struct node \*ptr;

ptr = front;

if(front == NULL)

{

printf("\nEmpty queue\n");

}

else

{ printf("\nprinting values .....\n");

while(ptr != NULL)

{

printf("\n%d\n",ptr -> data);

ptr = ptr -> next;

}

}

}

**Output:**

\*\*Main Menu

1.insert an element

2.Delete an element

3.Display the queue

4.Exit

Enter your choice ?1

Enter value?

14

\*\*Main Menu

1.insert an element

2.Delete an element

3.Display the queue

4.Exit

Enter your choice ?1

Enter value?

20

\*\*Main Menu

1.insert an element

2.Delete an element

3.Display the queue

4.Exit

Enter your choice ?1

Enter value?

23

\*\*Main Menu

1.insert an element

2.Delete an element

3.Display the queue

4.Exit

Enter your choice ?3

printing values .....

14

20

23

\*\*Main Menu

1.insert an element

2.Delete an element

3.Display the queue

4.Exit

Enter your choice ?1

Enter value?

33

\*\*Main Menu

1.insert an element

2.Delete an element

3.Display the queue

4.Exit

Enter your choice ?3

printing values .....

14

20

23

33

\*\*Main Menu

1.insert an element

2.Delete an element

3.Display the queue

4.Exit

Enter your choice ?2

\*\*Main Menu

1.insert an element

2.Delete an element

3.Display the queue

4.Exit

Enter your choice ?3

printing values .....

20

23

33

\*\*Main Menu

1.insert an element

2.Delete an element

3.Display the queue

4.Exit

Enter your choice ?4

=== Code Execution Successful ===

1. **Program for a Binary Search Tree using**

**recursion.**

#include <stdio.h>

#include <stdlib.h>

struct BinaryTreeNode {

int key;

struct BinaryTreeNode \*left, \*right;

};

struct BinaryTreeNode\* newNodeCreate(int value)

{

struct BinaryTreeNode\* temp

= (struct BinaryTreeNode\*)malloc(

sizeof(struct BinaryTreeNode));

temp->key = value;

temp->left = temp->right = NULL;

return temp;

}

struct BinaryTreeNode\*

searchNode(struct BinaryTreeNode\* root, int target)

{

if (root == NULL || root->key == target) {

return root;

}

if (root->key < target) {

return searchNode(root->right, target);

}

return searchNode(root->left, target);

}

struct BinaryTreeNode\*

insertNode(struct BinaryTreeNode\* node, int value)

{

if (node == NULL) {

return newNodeCreate(value);

}

if (value < node->key) {

node->left = insertNode(node->left, value);

}

else if (value > node->key) {

node->right = insertNode(node->right, value);

}

return node;

}

void postOrder(struct BinaryTreeNode\* root)

{

if (root != NULL) {

postOrder(root->left);

postOrder(root->right);

printf(" %d ", root->key);

}

}

void inOrder(struct BinaryTreeNode\* root)

{

if (root != NULL) {

inOrder(root->left);

printf(" %d ", root->key);

inOrder(root->right);

}

}

void preOrder(struct BinaryTreeNode\* root)

{

if (root != NULL) {

printf(" %d ", root->key);

preOrder(root->left);

preOrder(root->right);

}

}

struct BinaryTreeNode\* findMin(struct BinaryTreeNode\* root)

{

if (root == NULL) {

return NULL;

}

else if (root->left != NULL) {

return findMin(root->left);

}

return root;

}

struct BinaryTreeNode\* delete (struct BinaryTreeNode\* root,

int x)

{

if (root == NULL)

return NULL;

if (x > root->key) {

root->right = delete (root->right, x);

}

else if (x < root->key) {

root->left = delete (root->left, x);

}

else {

if (root->left == NULL && root->right == NULL) {

free(root);

return NULL;

}

else if (root->left == NULL

|| root->right == NULL) {

struct BinaryTreeNode\* temp;

if (root->left == NULL) {

temp = root->right;

}

else {

temp = root->left;

}

free(root);

return temp;

}

else {

struct BinaryTreeNode\* temp

= findMin(root->right);

root->key = temp->key;

root->right = delete (root->right, temp->key);

}

}

return root;

}

int main()

{

struct BinaryTreeNode\* root = NULL;

root = insertNode(root, 50);

insertNode(root, 30);

insertNode(root, 20);

insertNode(root, 40);

insertNode(root, 70);

insertNode(root, 60);

insertNode(root, 80);

if (searchNode(root, 60) != NULL) {

printf("60 found");

}

else {

printf("60 not found");

}

printf("\n");

postOrder(root);

printf("\n");

preOrder(root);

printf("\n");

inOrder(root);

printf("\n");

struct BinaryTreeNode\* temp = delete (root, 70);

printf("After Delete: \n");

inOrder(root);

return 0;

}

**Output:**

60 found

20 40 30 60 80 70 50

50 30 20 40 70 60 80

20 30 40 50 60 70 80

After Delete:

20 30 40 50 60 80

=== Code Execution Successful ===

1. **Program to represent a Graph using arrays.**

**Program:**

#include <stdio.h>

#include <stdlib.h>

#define MAX 10

#define TRUE 1

#define FALSE 0

void insertEdge(int graph[MAX][MAX], int u, int v);

void deleteEdge(int graph[MAX][MAX], int u, int v);

int searchEdge(int graph[MAX][MAX], int u, int v);

void BFS(int graph[MAX][MAX], int start);

void DFS(int graph[MAX][MAX], int start);

void DFSUtil(int graph[MAX][MAX], int start,

int visited[MAX]);

int main()

{

int graph[MAX][MAX] = { 0 };

insertEdge(graph, 0, 1);

insertEdge(graph, 0, 2);

insertEdge(graph, 1, 2);

insertEdge(graph, 2, 0);

insertEdge(graph, 2, 3);

printf("BFS starting from node 2:\n");

BFS(graph, 2);

printf("DFS starting from node 2:\n");

DFS(graph, 2);

return 0;

}

void insertEdge(int graph[MAX][MAX], int u, int v)

{

graph[u][v] = 1;

graph[v][u] = 1;

}

void deleteEdge(int graph[MAX][MAX], int u, int v)

{

graph[u][v] = 0;

graph[v][u] = 0;

}

int searchEdge(int graph[MAX][MAX], int u, int v)

{

return graph[u][v];

}

void BFS(int graph[MAX][MAX], int start)

{

int visited[MAX] = { 0 };

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

visited[start] = TRUE;

queue[rear++] = start;

while (front < rear) {

int current = queue[front++];

printf("%d ", current);

for (int i = 0; i < MAX; i++) {

if (graph[current][i] == 1 && !visited[i]) {

visited[i] = TRUE;

queue[rear++] = i;

}

}

}

printf("\n");

}

void DFS(int graph[MAX][MAX], int start)

{

int visited[MAX] = { 0 };

DFSUtil(graph, start, visited);

printf("\n");

}

void DFSUtil(int graph[MAX][MAX], int start,

int visited[MAX])

{

visited[start] = TRUE;

printf("%d ", start);

for (int i = 0; i < MAX; i++) {

if (graph[start][i] == 1 && !visited[i]) {

DFSUtil(graph, i, visited);

}

}

}

**Output:**

BFS starting from node 2:

2 0 1 3

DFS starting from node 2:

2 0 1 3

1. **Program for Infix to Postfix conversion.**

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

int prec(char c) {

if (c == '^')

return 3;

else if (c == '/' || c == '\*')

return 2;

else if (c == '+' || c == '-')

return 1;

else

return -1;

}

void infixToPostfix(char\* exp) {

int len = strlen(exp);

char result[len + 1];

char stack[len];

int j = 0;

int top = -1;

for (int i = 0; i < len; i++) {

char c = exp[i];

if (isalnum(c))

result[j++] = c;

else if (c == '(')

stack[++top] = '(';

else if (c == ')') {

while (top != -1 && stack[top] != '(') {

result[j++] = stack[top--];

}

top--;

}

else {

while (top != -1 && (prec(c) < prec(stack[top]) ||

prec(c) == prec(stack[top]))) {

result[j++] = stack[top--];

}

stack[++top] = c;

}

}

while (top != -1) {

result[j++] = stack[top--];

}

result[j] = '\0';

printf("%s\n", result);

}

int main() {

char exp[] = "a+b\*(c^d-e)^(f+g\*h)-i";

infixToPostfix(exp);

return 0;

}

**Output:**

abcd^e-fgh\*+^\*+i-

1. **Program for Evaluation of Postfix Expressions.**

**Program:**

#include <stdio.h>

#include <string.h>

#include <math.h>

#include <ctype.h>

#define MAX 20

int s[MAX], top = -1;

void push(int element) {

if (top == MAX - 1) {

printf("Stack Overflow\n");

return;

}

s[++top] = element;

}

int pop() {

if (top == -1) {

printf("Stack Underflow\n");

return -1;

}

return s[top--];

}

int main() {

char postfix[MAX], ch;

int i, op1, op2, res, len;

printf("\nProgram to Evaluate Postfix Expression\n");

printf("Enter the postfix expression: ");

scanf("%s", postfix);

len = strlen(postfix);

for (i = 0; i < len; i++) {

ch = postfix[i];

if (isdigit(ch))

push(ch - '0');

else {

op2 = pop();

op1 = pop();

switch (ch) {

case '+': res = op1 + op2; break;

case '-': res = op1 - op2; break;

case '\*': res = op1 \* op2; break;

case '/': res = op1 / op2; break;

case '^': res = pow(op1, op2); break;

default: printf("Invalid Character\n");

return 1;

}

push(res);

}

}

printf("Result of the expression: %d\n", pop());

return 0;

}

**Output:**

Program to Evaluate Postfix Expression

Enter the postfix expression: 53+62\*-

Result of the expression: -4

1. **Program to represent Doubly Linked List.**

**Program:**

#include <stdio.h>

#include <stdlib.h>

// defining a node

typedef struct Node {

int data;

struct Node\* next;

struct Node\* prev;

} Node;

Node\* createNode(int data)

{

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

newNode->data = data;

newNode->next = NULL;

newNode->prev = NULL;

return newNode;

}

void insertAtBeginning(Node\*\* head, int data)

{

// creating new node

Node\* newNode = createNode(data);

// check if DLL is empty

if (\*head == NULL) {

\*head = newNode;

return;

}

newNode->next = \*head;

(\*head)->prev = newNode;

\*head = newNode;

}

void insertAtEnd(Node\*\* head, int data)

{

// creating new node

Node\* newNode = createNode(data);

// check if DLL is empty

if (\*head == NULL) {

\*head = newNode;

return;

}

Node\* temp = \*head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

newNode->prev = temp;

}

void insertAtPosition(Node\*\* head, int data, int position)

{

if (position < 1) {

printf("Position should be >= 1.\n");

return;

}

if (position == 1) {

insertAtBeginning(head, data);

return;

}

Node\* newNode = createNode(data);

Node\* temp = \*head;

for (int i = 1; temp != NULL && i < position - 1; i++) {

temp = temp->next;

}

if (temp == NULL) {

printf(

"Position greater than the number of nodes.\n");

return;

}

newNode->next = temp->next;

newNode->prev = temp;

if (temp->next != NULL) {

temp->next->prev = newNode;

}

temp->next = newNode;

}

void deleteAtBeginning(Node\*\* head)

{

// checking if the DLL is empty

if (\*head == NULL) {

printf("The list is already empty.\n");

return;

}

Node\* temp = \*head;

\*head = (\*head)->next;

if (\*head != NULL) {

(\*head)->prev = NULL;

}

free(temp);

}

void deleteAtEnd(Node\*\* head)

{

// checking if DLL is empty

if (\*head == NULL) {

printf("The list is already empty.\n");

return;

}

Node\* temp = \*head;

if (temp->next == NULL) {

\*head = NULL;

free(temp);

return;

}

while (temp->next != NULL) {

temp = temp->next;

}

temp->prev->next = NULL;

free(temp);

}

void deleteAtPosition(Node\*\* head, int position)

{

if (\*head == NULL) {

printf("The list is already empty.\n");

return;

}

Node\* temp = \*head;

if (position == 1) {

deleteAtBeginning(head);

return;

}

for (int i = 1; temp != NULL && i < position; i++) {

temp = temp->next;

}

if (temp == NULL) {

printf("Position is greater than the number of "

"nodes.\n");

return;

}

if (temp->next != NULL) {

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

}

if (temp->prev != NULL) {

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

}

free(temp);

}

void printListForward(Node\* head)

{

Node\* temp = head;

printf("Forward List: ");

while (temp != NULL) {

printf("%d ", temp->data);

temp = temp->next;

}

printf("\n");

}

void printListReverse(Node\* head)

{

Node\* temp = head;

if (temp == NULL) {

printf("The list is empty.\n");

return;

}

while (temp->next != NULL) {

temp = temp->next;

}

// Traverse backwards

printf("Reverse List: ");

while (temp != NULL) {

printf("%d ", temp->data);

temp = temp->prev;

}

printf("\n");

}

int main()

{

Node\* head = NULL;

insertAtEnd(&head, 10);

insertAtEnd(&head, 20);

insertAtBeginning(&head, 5);

insertAtPosition(&head, 15, 2); // List: 5 15 10 20

printf("After Insertions:\n");

printListForward(head);

printListReverse(head);

deleteAtBeginning(&head); // List: 15 10 20

deleteAtEnd(&head); // List: 15 10

deleteAtPosition(&head, 2); // List: 15

printf("After Deletions:\n");

printListForward(head);

return 0;

}

**Output:**

After Insertions:

Forward List: 5 15 10 20

Reverse List: 20 10 15 5

After Deletions:

Forward List: 15

=== Code Execution Successful ===

1. **Program to represent Circular Linked List**

**Program:**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node \*next;

};

struct Node\* createNode(int value);

struct Node\* insertAtPosition(struct Node \*last, int data, int pos) {

if (last == NULL) {

// If the list is empty

if (pos != 1) {

printf("Invalid position!\n");

return last;

}

struct Node \*newNode = createNode(data);

last = newNode;

last->next = last;

return last;

}

struct Node \*newNode = createNode(data);

struct Node \*curr = last->next;

if (pos == 1) {

newNode->next = curr;

last->next = newNode;

return last;

}

for (int i = 1; i < pos - 1; ++i) {

curr = curr->next;

if (curr == last->next) {

printf("Invalid position!\n");

return last;

}

}

newNode->next = curr->next;

curr->next = newNode;

if (curr == last) last = newNode;

return last;

}

void printList(struct Node \*last) {

if (last == NULL) return;

struct Node \*head = last->next;

while (1) {

printf("%d ", head->data);

head = head->next;

if (head == last->next) break;

}

printf("\n");

}

struct Node\* createNode(int value) {

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

newNode->data = value;

newNode->next = NULL;

return newNode;

}

int main() {

// Create circular linked list: 2, 3, 4

struct Node \*first = createNode(2);

first->next = createNode(3);

first->next->next = createNode(4);

struct Node \*last = first->next->next;

last->next = first;

printf("Original list: ");

printList(last);

int data = 5, pos = 2;

last = insertAtPosition(last, data, pos);

printf("List after insertions: ");

printList(last);

return 0;

}

**Output:**

Original list: 2 3 4

List after insertions: 2 5 3 4

=== Code Execution Successful ===

1. **Program to Represent Graph Using Linked List**

**Program:**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int vertex;

struct Node\* next;

};

struct Graph {

int numVertices;

struct Node\*\* adjLists;

};

struct Node\* createNode(int vertex) {

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

newNode->vertex = vertex;

newNode->next = NULL;

return newNode;

}

struct Graph\* createGraph(int vertices) {

struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));

graph->numVertices = vertices;

graph->adjLists = (struct Node\*\*)malloc(vertices \* sizeof(struct Node\*));

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

graph->adjLists[i] = NULL;

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) {

for (int i = 0; i < graph->numVertices; i++) {

struct Node\* temp = graph->adjLists[i];

printf("Adjacency list of vertex %d: ", i);

while (temp) {

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

temp = temp->next;

}

printf("NULL\n");

}

}

int main() {

int vertices = 5;

struct Graph\* graph = createGraph(vertices);

addEdge(graph, 0, 1);

addEdge(graph, 0, 4);

addEdge(graph, 1, 2);

addEdge(graph, 1, 3);

addEdge(graph, 1, 4);

addEdge(graph, 2, 3);

addEdge(graph, 3, 4);

printGraph(graph);

return 0;

}

**Output:**

Adjacency list of vertex 0: 4 -> 1 -> NULL

Adjacency list of vertex 1: 4 -> 3 -> 2 -> 0 -> NULL

Adjacency list of vertex 2: 3 -> 1 -> NULL

Adjacency list of vertex 3: 4 -> 2 -> 1 -> NULL

Adjacency list of vertex 4: 3 -> 1 -> 0 -> NULL

=== Code Execution Successful ===

1. **Program to Allocate two-dimensional Dynamically**

**Program:**

#include <stdio.h>

#include <stdlib.h>

int main() {

int rows, cols;

printf("Enter the number of rows: ");

scanf("%d", &rows);

printf("Enter the number of columns: ");

scanf("%d", &cols);

int \*\*array = (int \*\*)malloc(rows \* sizeof(int \*));

for (int i = 0; i < rows; i++) {

array[i] = (int \*)malloc(cols \* sizeof(int));

}

if (array == NULL) {

printf("Memory allocation failed!\n");

return 1;

}

printf("Enter elements of the array:\n");

for (int i = 0; i < rows; i++) {

for (int j = 0; j < cols; j++) {

scanf("%d", &array[i][j]);

}

}

printf("Array elements:\n");

for (int i = 0; i < rows; i++) {

for (int j = 0; j < cols; j++) {

printf("%d ", array[i][j]);

}

printf("\n");

}

for (int i = 0; i < rows; i++) {

free(array[i]);

}

free(array);

return 0;

}

**Output:**

Enter the number of rows: 3

Enter the number of columns: 3

Enter elements of the array:

1

2

3

4

5

6

7

8

1

Array elements:

1 2 3

4 5 6

7 8 1

=== Code Execution Successful ===

1. **Program to demonstrate the use of realloc function.**

**Program:**

#include <stdio.h>

#include <stdlib.h>

int main()

{

int \*ptr = (int \*)malloc(sizeof(int)\*2);

int i;

int \*ptr\_new;

\*ptr = 10;

\*(ptr + 1) = 20;

ptr\_new = (int \*)realloc(ptr, sizeof(int)\*3);

\*(ptr\_new + 2) = 30;

for(i = 0; i < 3; i++)

printf("%d ", \*(ptr\_new + i));

getchar();

return 0;

}

**Output:**

10 20 30

1. **Program for Binary Search Tree Traversal without Recursion**

**Program:**

#include <stdio.h>

#include <stdlib.h>

// Structure for a BST Node

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* createNode(int data) {

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

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

void inorderTraversal(struct Node\* root) {

struct Node\* stack[100]; // Stack for storing nodes

int top = -1;

struct Node\* current = root;

while (current != NULL || top != -1) {

while (current != NULL) {

stack[++top] = current;

current = current->left;

}

current = stack[top--];

printf("%d ", current->data);

current = current->right;

}

}

void preorderTraversal(struct Node\* root) {

if (root == NULL) return;

struct Node\* stack[100];

int top = -1;

stack[++top] = root;

while (top != -1) {

struct Node\* current = stack[top--];

printf("%d ", current->data);

if (current->right) stack[++top] = current->right;

if (current->left) stack[++top] = current->left;

}

}

void postorderTraversal(struct Node\* root) {

if (root == NULL) return;

struct Node\* stack1[100], \*stack2[100];

int top1 = -1, top2 = -1;

stack1[++top1] = root;

while (top1 != -1) {

struct Node\* current = stack1[top1--];

stack2[++top2] = current;

if (current->left) stack1[++top1] = current->left;

if (current->right) stack1[++top1] = current->right;

}

while (top2 != -1) {

printf("%d ", stack2[top2--]->data);

}

}

int main() {

struct Node\* root = createNode(10);

root->left = createNode(5);

root->right = createNode(20);

root->left->left = createNode(3);

root->left->right = createNode(7);

root->right->left = createNode(15);

root->right->right = createNode(25);

printf("Inorder Traversal (LNR) : ");

inorderTraversal(root);

printf("\n");

printf("Preorder Traversal (NLR) : ");

preorderTraversal(root);

printf("\n");

printf("Postorder Traversal (LRN) : ");

postorderTraversal(root);

printf("\n");

return 0;

}

**Output:**

Inorder Traversal : 3 5 7 10 15 20 25

Preorder Traversal : 10 5 3 7 20 15 25

Postorder Traversal : 3 7 5 15 25 20 10

=== Code Execution Successful ===