TUT ALL ASSIGNMENTS

Name: Adit Doke

Rollno.36

AIDS-A batch 2

**Problem statement :**

**1.Fibonacci Search**

#include <stdio.h>

void insertionSort(int array[], int size) {

for (int step = 1; step < size; step++) {

int key = array[step];

int j = step - 1;

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

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

--j;

}

array[j + 1] = key;

}

}

int min(int x, int y){

if (x <= y)

return x;

else

return y;

}

int fibSearch(int arr[], int x, int n){

int fib2 = 0;

int fib1 = 1;

int fib = fib2 + fib1;

while (fib < n) {

fib2 = fib1;

fib1 = fib;

fib = fib2 + fib1;

}

int offset = -1;

while (fib > 1) {

int i = min(offset + fib2, n - 1);

if (arr[i] < x) {

fib = fib1;

fib1 = fib2;

fib2 = fib - fib1;

offset = i;

}

else if (arr[i] > x) {

fib = fib2;

fib1 = fib1 - fib2;

fib2 = fib - fib1;

}

else

return i;

}

if (fib1 && arr[offset + 1] == x)

return offset + 1;

return -1;

}

int main()

{

int arr[100];

int num, pos, count;

printf("\nENTER NUMBER OF ELEMENTS: ");

scanf("%d", &count);

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

printf("\nENTER ELEMENT %d: ",i);

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

}

insertionSort(arr, count);

printf("\nTHE SORTED ARRAY IS: \n");

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

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

printf("\n");

printf("\nENTER ELEMENT TO BE SEARCHED: ");

scanf("%d", &num);

pos = fibSearch(arr, num, count);

if ( pos != -1)

printf("\nELEMENT FOUND AT INDEX %dIN SORTED ARRAY\n",pos);

else

printf("\nELEMENT NOT FOUND\n");

}

**2.Infix to Postfix**

#include<stdio.h>

#include<string.h>

#include<limits.h>

#include<stdlib.h>

# define MAX 100

int top = -1;

char stack[MAX];

// checking if stack is full

int isFull() {

return top == MAX - 1;

}

// checking is stack is empty

int isEmpty() {

return top == -1;

}

// Push function here, inserts value in stack and increments stack top by 1

void push(char item) {

if (isFull())

return;

top++;

stack[top] = item;

}

// Function to remove an item from stack. It decreases top by 1

int pop() {

if (isEmpty())

return INT\_MIN;

// decrements top and returns what has been popped

return stack[top--];

}

// Function to return the top from stack without removing it

int peek(){

if (isEmpty())

return INT\_MIN;

return stack[top];

}

// A utility function to check if the given character is operand

int checkIfOperand(char ch) {

return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');

}

// Fucntion to compare precedence

// If we return larger value means higher precedence

int precedence(char ch)

{

switch (ch)

{

case '+':

case '-':

return 1;

case '\*':

case '/':

return 2;

case '^':

return 3;

}

return -1;

}

// The driver function for infix to postfix conversion

int getPostfix(char\* expression)

{

int i, j;

for (i = 0, j = -1; expression[i]; ++i)

{

// Here we are checking is the character we scanned is operand or not

// and this adding to output.

if (checkIfOperand(expression[i]))

expression[++j] = expression[i];

// Here, if we scan character ‘(‘, we need push it to the stack.

else if (expression[i] == '(')

push(expression[i]);

// Here, if we scan character is an ‘)’, we need to pop and print from the stack

// do this until an ‘(‘ is encountered in the stack.

else if (expression[i] == ')')

{

while (!isEmpty(stack) && peek(stack) != '(')

expression[++j] = pop(stack);

if (!isEmpty(stack) && peek(stack) != '(')

return -1; // invalid expression

else

pop(stack);

}

else // if an opertor

{

while (!isEmpty(stack) && precedence(expression[i]) <= precedence(peek(stack)))

expression[++j] = pop(stack);

push(expression[i]);

}

}

// Once all inital expression characters are traversed

// adding all left elements from stack to exp

while (!isEmpty(stack))

expression[++j] = pop(stack);

expression[++j] = '\0';

}

void reverse(char \*exp){

int size = strlen(exp);

int j = size, i=0;

char temp[size];

temp[j--]='\0';

while(exp[i]!='\0')

{

temp[j] = exp[i];

j--;

i++;

}

strcpy(exp,temp);

}

void brackets(char\* exp){

int i = 0;

while(exp[i]!='\0')

{

if(exp[i]=='(')

exp[i]=')';

else if(exp[i]==')')

exp[i]='(';

i++;

}

}

void InfixtoPrefix(char \*exp){

int size = strlen(exp);

// reverse string

reverse(exp);

//change brackets

brackets(exp);

//get postfix

getPostfix(exp);

// reverse string again

reverse(exp);

}

int main()

{

printf("The infix is: ");

char expression[] = "((a/b)+c)-(d+(e\*f))";

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

InfixtoPrefix(expression);

printf("The prefix is: ");

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

return 0;

}

**3.Doubly Linked list**

#include<stdio.h>

#include<stdlib.h>

struct node

{

struct node \*prev;

struct node \*next;

int data;

};

struct node \*head;

void insertion\_beginning();

void insertion\_last();

void insertion\_specified();

void deletion\_beginning();

void deletion\_last();

void deletion\_specified();

void display();

void search();

int main () {

int choice, i;

main\_menu:

printf("\n1. INSERT\n2. DELETE\n3. DISPLAY\n4. SEARCH\n5. EXIT\n");

printf("\nENTER YOUR CHOICE: ");

scanf("%d",&choice);

switch(choice){

case 1:

insert:

printf("\n1. INSERT AT BEGINNING\n2. INSERT AT END\n3. INSERT IN MIDDLE\n4. EXIT\n");

printf("\nENTER YOUR CHOICE: ");

scanf("%d",&i);

switch(i){

case 1:

insertion\_beginning();

goto insert;

break;

case 2:

insertion\_last();

goto insert;

break;

case 3:

insertion\_specified();

goto insert;

break;

case 4:

goto main\_menu;

break;

default:

printf("\nINVALID CHOICE\n");

break;

}

break;

case 2:

del:

printf("\n1. DELETE NODE AT BEGINNING\n2. DELETE NODE AT END\n3. DELETE NODE IN MIDDLE\n4. EXIT\n");

printf("\nENTER YOUR CHOICE: ");

scanf("%d",&i);

switch(i){

case 1:

deletion\_beginning();

goto insert;

break;

case 2:

deletion\_last();

goto insert;

break;

case 3:

deletion\_specified();

goto insert;

break;

case 4:

goto main\_menu;

break;

default:

printf("\nINVALID CHOICE\n");

break;

}

break;

case 3:

display();

goto main\_menu;

break;

case 4:

search();

goto main\_menu;

break;

case 5:

return 0;

default:

printf("\nINVALID CHOICE\n");

break;

}

}

void insertion\_beginning(){

struct node \*ptr;

int item;

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

if(ptr == NULL){

printf("\nOVERFLOW");

}

else{

printf("\nENTER ELEMENT: ");

scanf("%d",&item);

if(head==NULL){

ptr->next = NULL;

ptr->prev=NULL;

ptr->data=item;

head=ptr;

}

else{

ptr->data=item;

ptr->prev=NULL;

ptr->next = head;

head->prev=ptr;

head=ptr;

}

printf("\nNODE INSERTED\n");

}

}

void insertion\_last(){

struct node \*ptr,\*temp;

int item;

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

if(ptr == NULL){

printf("\nOVERFLOW");

}

else{

printf("\nENTER ELEMENT: ");

scanf("%d",&item);

ptr->data=item;

if(head == NULL){

ptr->next = NULL;

ptr->prev = NULL;

head = ptr;

}

else{

temp = head;

while(temp->next!=NULL){

temp = temp->next;

}

temp->next = ptr;

ptr ->prev=temp;

ptr->next = NULL;

}

}

printf("\nNODE INSERTED\n");

}

void insertion\_specified(){

struct node \*ptr,\*temp;

int item,loc,i;

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

if(ptr == NULL){

printf("\nOVERFLOW");

}

else{

temp = head;

printf("\nENTER POSITION: ");

scanf("%d",&loc);

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

temp = temp->next;

if(temp == NULL){

printf("\nTHERE ARE LESS THAN %d ELEMENTS", loc);

return;

}

}

printf("\nENTER ELEMENT: ");

scanf("%d",&item);

ptr->data = item;

ptr->next = temp->next;

ptr -> prev = temp;

temp->next = ptr;

temp->next->prev=ptr;

printf("\nNODE INSERTED\n");

}

}

void deletion\_beginning(){

struct node \*ptr;

if(head == NULL){

printf("\nUNDERFLOW");

}

else if(head->next == NULL){

head = NULL;

free(head);

printf("\nNODE DELETED\n");

}

else{

ptr = head;

head = head -> next;

head -> prev = NULL;

free(ptr);

printf("\nNODE DELETED\n");

}

}

void deletion\_last(){

struct node \*ptr;

if(head == NULL){

printf("\n UNDERFLOW");

}

else if(head->next == NULL){

head = NULL;

free(head);

printf("\nNODE DELETED\n");

}

else{

ptr = head;

while(ptr->next != NULL){

ptr = ptr -> next;

}

ptr -> prev -> next = NULL;

free(ptr);

printf("\nNODE DELETED\n");

}

}

void deletion\_specified(){

struct node \*ptr, \*temp;

int val;

printf("\nENTER DATA AFTER WHICH NODE IS TO BE DELETED: ");

scanf("%d", &val);

ptr = head;

while(ptr -> data != val)

ptr = ptr -> next;

if(ptr -> next == NULL){

printf("\nCANNOT DELETE\n");

}

else if(ptr -> next -> next == NULL){

ptr ->next = NULL;

}

else{

temp = ptr -> next;

ptr -> next = temp -> next;

temp -> next -> prev = ptr;

free(temp);

printf("\nNODE DELETED\n");

}

}

void display(){

struct node \*ptr;

printf("\nTHE DOUBLY LINKED LIST IS: \n");

ptr = head;

while(ptr != NULL){

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

ptr=ptr->next;

}

}

void search(){

struct node \*ptr;

int item,i=0,flag;

ptr = head;

if(ptr == NULL){

printf("\nEMPTY LIST\n");

}

else{

printf("\nENTER ELEMENT TO BE SEARCHED: \n");

scanf("%d",&item);

while (ptr!=NULL){

if(ptr->data == item){

printf("\nELEMENT FOUND AT LOCATION %d ",i+1);

flag=0;

break;

}

else{

flag=1;

}

i++;

ptr = ptr -> next;

}

if(flag==1){

printf("\nELEMENT NOT FOUND\n");

}

}

}

**4.Tree Operations**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

int count;

struct node{

int data;

struct node \*left;

struct node \*right;

};

struct node \*root = NULL;

struct node \*create\_node(int data){

struct node \*new\_node = (struct node \*)malloc(sizeof(struct node));

if (new\_node == NULL){

printf("\nMEMORY NOT ALLOCATED");

return NULL;

}

new\_node->data = data;

new\_node->left = NULL;

new\_node->right = NULL;

return new\_node;

}

void insert(int data){

struct node \*new\_node = create\_node(data);

if (new\_node != NULL){

if (root == NULL){

root = new\_node;

printf("\nNODE INSERTED\n");

return;

}

struct node \*temp = root;

struct node \*prev = NULL;

while (temp != NULL){

prev = temp;

if (data > temp->data)

temp = temp->right;

else

temp = temp->left;

}

if (data > prev->data)

prev->right = new\_node;

else

prev->left = new\_node;

printf("\nNODE INSERTED\n");

}

}

struct node \*smallest\_node(struct node \*root){

struct node \*curr = root;

while (curr != NULL && curr->left != NULL)

curr = curr->left;

return curr;

}

struct node \*delete (struct node \*root, int key){

if (root == NULL)

return root;

if (key < root->data)

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

else if (key > root->data)

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

else{

if (root->left == NULL){

struct node \*temp = root->right;

free(root);

return temp;

}

else if (root->right == NULL){

struct node \*temp = root->left;

free(root);

return temp;

}

struct node \*temp = smallest\_node(root->right);

root->data = temp->data;

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

}

return root;

}

int search(int key){

struct node \*temp = root;

while (temp != NULL){

if (key == temp->data)

return 1;

else if (key > temp->data)

temp = temp->right;

else

temp = temp->left;

}

return 0;

}

void inorderRecur(struct node \*root){

if (root == NULL)

return;

inorderRecur(root->left);

printf("%d\t", root->data);

inorderRecur(root->right);

}

void preorderRecur(struct node \*root){

if (root == NULL)

return;

printf("%d\t", root->data);

preorderRecur(root->left);

preorderRecur(root->right);

}

void postorderRecur(struct node \*root){

if (root == NULL)

return;

postorderRecur(root->left);

postorderRecur(root->right);

printf("%d\t", root->data);

}

int tree\_height(struct node \*root) {

// Get the height of the tree

if (!root)

return 0;

else {

// Find the height of both subtrees

// and use the larger one

int left\_height = tree\_height(root->left);

int right\_height = tree\_height(root->right);

if (left\_height >= right\_height)

return left\_height + 1;

else

return right\_height + 1;

}

}

int countnodes(struct node \*root){

if(root != NULL){

countnodes(root->left);

count++;

countnodes(root->right);

}

return count;

}

void mirror(struct node\* root){

if (root == NULL)

return;

else {

struct node\* temp;

/\* do the subtrees \*/

mirror(root->left);

mirror(root->right);

/\* swap the pointers in this node \*/

temp = root->left;

root->left = root->right;

root->right = temp;

}

}

int main(){

int choice, i, j, data;

main\_menu:

printf("\n1. INSERT\n2. DELETE\n3. SEARCH\n4. TRAVERSE\n5. TREE HEIGHT\n6. NODE COUNT\n7. MIRROR IMAGE\n8. EXIT\n");

printf("\nENTER YOUR CHOICE: ");

scanf("%d",&choice);

switch(choice){

case 1:

printf("\nENTER VALUE TO BE INSERTED: ");

scanf("%d", &data);

insert(data);

goto main\_menu;

break;

case 2:

printf("\nENTER VALUE TO BE DELETED: ");

scanf("%d", &data);

root = delete(root, data);

goto main\_menu;

break;

case 3:

printf("\nENTER VALUE TO BE SEARCHED: ");

scanf("%d", &data);

if (search(data) == 1)

printf("\nDATA FOUND\n");

else

printf("\nDATA NOT FOUND\n");

goto main\_menu;

break;

case 4:

printf("\nRECURSIVE INORDER TRAVERSAL IS:\n");

inorderRecur(root);

printf("\n");

printf("\nRECURSIVE POSTORDER TRAVERSAL IS:\n");

postorderRecur(root);

printf("\n");

printf("\nRECURSIVE PREORDER TRAVERSAL IS:\n");

preorderRecur(root);

printf("\n");

goto main\_menu;

break;

case 5:

printf("\nHEIGHT OF TREE IS: %d\n", tree\_height(root));

goto main\_menu;

break;

case 6:

printf("\nNUMBER OF NODES IS: %d\n",countnodes(root));

goto main\_menu;

break;

case 7:

mirror(root);

goto main\_menu;

break;

case 8:

return 0;

default:

printf("\nINVALID CHOICE\n");

goto main\_menu;

break;

}

}

**5. Prims Algorithm**

#include <limits.h>

#include <stdbool.h>

#include <stdio.h>

#define V 5

int minKey(int key[], bool mstSet[]){

int min = INT\_MAX, min\_index;

for (int v = 0; v < V; v++)

if (mstSet[v] == false && key[v] < min)

min = key[v], min\_index = v;

return min\_index;

}

int printMST(int parent[], int graph[V][V]){

printf("Edge \tWeight\n");

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

printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);

}

void primMST(int graph[V][V]){

int parent[V];

int key[V];

bool mstSet[V];

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

key[i] = INT\_MAX, mstSet[i] = false;

key[0] = 0;

parent[0] = -1;

for (int count = 0; count < V - 1; count++) {

int u = minKey(key, mstSet);

mstSet[u] = true;

for (int v = 0; v < V; v++)

if (graph[u][v] && mstSet[v] == false

&& graph[u][v] < key[v])

parent[v] = u, key[v] = graph[u][v];

}

printMST(parent, graph);

}

int main()

{

int graph[V][V] = { { 0, 2, 0, 6, 0 },

{ 2, 0, 3, 8, 5 },

{ 0, 3, 0, 0, 7 },

{ 6, 8, 0, 0, 9 },

{ 0, 5, 7, 9, 0 } };

primMST(graph);

return 0;

}

**6.Kruskal Algorithm**

#include <stdio.h>

#include <conio.h>

#include <stdlib.h>

int i, j, k, a, b, u, v, n, ne = 1;

int min, mincost = 0, cost[9][9], parent[9];

int find(int);

int uni(int, int);

int main() {

printf("\n\tImplementation of Kruskal's Algorithm\n");

printf("\nEnter the no. of vertices:");

scanf("%d", & n);

printf("\nEnter the cost adjacency matrix:\n");

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

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

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

if (cost[i][j] == 0)

cost[i][j] = 999;

}

}

printf("The edges of Minimum Cost Spanning Tree are\n");

while (ne < n) {

for (i = 1, min = 999; i <= n; i++) {

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

if (cost[i][j] < min) {

min = cost[i][j];

a = u = i;

b = v = j;

}

}

}

u = find(u);

v = find(v);

if (uni(u, v)) {

printf("%d edge (%d,%d) =%d\n", ne++, a, b, min);

mincost += min;

}

cost[a][b] = cost[b][a] = 999;

}

printf("\n\tMinimum cost = %d\n", mincost);

getch();

}

int find(int i) {

while (parent[i])

i = parent[i];

return i;

}

int uni(int i, int j) {

if (i != j) {

parent[j] = i;

return 1;

}

return 0;

}

**7.Hashing**

#include<stdio.h>

#include<stdlib.h>

#define SIZE 10

int hash[SIZE];

void insert(){

int key, index, i, flag=0, hkey;

printf("\nENETR VALUE: ");

scanf("%d",&key);

hkey = key%SIZE;

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

if(i == SIZE-1)

printf("\nELEMENT CANNOT BE INSERTED\n");

else{

index=(hkey+i)%SIZE;

if(hash[index] == NULL){

hash[index]=key;

break;

}

}

}

}

void search(){

int key, index, i, flag=0, hkey;

printf("\nENETR VALUE TO BE SEARCHED: ");

scanf("%d",&key);

hkey=key%SIZE;

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

index=(hkey+i)%SIZE;

if(hash[index]==key){

printf("\nVALUE FOUND AT INDEX %d",index);

break;

}

}

if(i == SIZE)

printf("\nVALUE NOT FOUND\n");

}

void hash\_display(){

int i;

printf("\nELEMENTS IN THE HASH TABLE ARE \n");

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

printf("\nINDEX %d \t VALUE = %d",i,hash[i]);

}

int main (){

int ch, opt;

struct Graph\* graph;

int ver, start;

main\_menu:

printf("\n1. INSERT\n2. DISPLAY\n3. SEARCH\n4. EXIT\n");

printf("\nENTER YOUR CHOICE: ");

scanf("%d",&opt);

switch(opt){

case 1:

insert();

goto main\_menu;

break;

case 2:

hash\_display();

printf("\n");

goto main\_menu;

break;

case 3:

search();

goto main\_menu;

break;

case 4:

return 0;

}

}

//----------------------------------------------------------- X ----------------------------------------------------------