**Program 1**

Objective: WAP to implement Queue data structure using linked list

Code:

#include <stdio.h>

#include <stdlib.h>

struct node

{

int info;

struct node \*ptr;

}\*front,\*rear,\*temp,\*front1;

int frontelement();

void enq(int data);

void deq();

void empty();

void display();

void create();

void queuesize();

int count = 0;

void main()

{

int no, ch, e;

printf("\n 1 - Enque");

printf("\n 2 - Deque");

printf("\n 3 - Front element");

printf("\n 4 - Empty");

printf("\n 5 - Exit");

printf("\n 6 - Display");

printf("\n 7 - Queue size");

create();

while (1)

{

printf("\n Enter choice : ");

scanf("%d", &ch);

switch (ch)

{

case 1:

printf("Enter data : ");

scanf("%d", &no);

enq(no);

break;

case 2:

deq();

break;

case 3:

e = frontelement();

if (e != 0)

printf("Front element : %d", e);

else

printf("\n No front element in Queue as queue is empty");

break;

case 4:

empty();

break;

case 5:

exit(0);

case 6:

display();

break;

case 7:

queuesize();

break;

default:

printf("Wrong choice, Please enter correct choice ");

break;

}

}

}

/\* Create an empty queue \*/

void create()

{

front = rear = NULL;

}

/\* Returns queue size \*/

void queuesize()

{

printf("\n Queue size : %d", count);

}

/\* Enqueing the queue \*/

void enq(int data)

{

if (rear == NULL)

{

rear = (struct node \*)malloc(1\*sizeof(struct node));

rear->ptr = NULL;

rear->info = data;

front = rear;

}

else

{

temp=(struct node \*)malloc(1\*sizeof(struct node));

rear->ptr = temp;

temp->info = data;

temp->ptr = NULL;

rear = temp;

}

count++;

}

/\* Displaying the queue elements \*/

void display()

{

front1 = front;

if ((front1 == NULL) && (rear == NULL))

{

printf("Queue is empty");

return;

}

while (front1 != rear)

{

printf("%d ", front1->info);

front1 = front1->ptr;

}

if (front1 == rear)

printf("%d", front1->info);

}

/\* Dequeing the queue \*/

void deq()

{

front1 = front;

if (front1 == NULL)

{

printf("\n Error: Trying to display elements from empty queue");

return;

}

else

if (front1->ptr != NULL)

{

front1 = front1->ptr;

printf("\n Dequed value : %d", front->info);

free(front);

front = front1;

}

else

{

printf("\n Dequed value : %d", front->info);

free(front);

front = NULL;

rear = NULL;

}

count--;

}

/\* Returns the front element of queue \*/

int frontelement()

{

if ((front != NULL) && (rear != NULL))

return(front->info);

else

return 0;

}

/\* Display if queue is empty or not \*/

void empty()

{

if ((front == NULL) && (rear == NULL))

printf("\n Queue empty");

else

printf("Queue not empty");

}

**Program 2**

Objective: WAP to implement Stack data structure using linked list

Code:

#include <stdio.h>

#include <stdlib.h>

struct node

{

int info;

struct node \*ptr;

}\*top,\*top1,\*temp;

int topelement();

void push(int data);

void pop();

void empty();

void display();

void destroy();

void stack\_count();

void create();

int count = 0;

void main()

{

int no, ch, e;

printf("\n 1 - Push");

printf("\n 2 - Pop");

printf("\n 3 - Top");

printf("\n 4 - Empty");

printf("\n 5 - Exit");

printf("\n 6 - Dipslay");

printf("\n 7 - Stack Count");

printf("\n 8 - Destroy stack");

create();

while (1)

{

printf("\n Enter choice : ");

scanf("%d", &ch);

switch (ch)

{

case 1:

printf("Enter data : ");

scanf("%d", &no);

push(no);

break;

case 2:

pop();

break;

case 3:

if (top == NULL)

printf("No elements in stack");

else

{

e = topelement();

printf("\n Top element : %d", e);

}

break;

case 4:

empty();

break;

case 5:

exit(0);

case 6:

display();

break;

case 7:

stack\_count();

break;

case 8:

destroy();

break;

default :

printf(" Wrong choice, Please enter correct choice ");

break;

}

}

}

/\* Create empty stack \*/

void create()

{

top = NULL;

}

/\* Count stack elements \*/

void stack\_count()

{

printf("\n No. of elements in stack : %d", count);

}

/\* Push data into stack \*/

void push(int data)

{

if (top == NULL)

{

top =(struct node \*)malloc(1\*sizeof(struct node));

top->ptr = NULL;

top->info = data;

}

else

{

temp =(struct node \*)malloc(1\*sizeof(struct node));

temp->ptr = top;

temp->info = data;

top = temp;

}

count++;

}

/\* Display stack elements \*/

void display()

{

top1 = top;

if (top1 == NULL)

{

printf("Stack is empty");

return;

}

while (top1 != NULL)

{

printf("%d ", top1->info);

top1 = top1->ptr;

}

}

/\* Pop Operation on stack \*/

void pop()

{

top1 = top;

if (top1 == NULL)

{

printf("\n Error : Trying to pop from empty stack");

return;

}

else

top1 = top1->ptr;

printf("\n Popped value : %d", top->info);

free(top);

top = top1;

count--;

}

/\* Return top element \*/

int topelement()

{

return(top->info);

}

/\* Check if stack is empty or not \*/

void empty()

{

if (top == NULL)

printf("\n Stack is empty");

else

printf("\n Stack is not empty with %d elements", count);

}

/\* Destroy entire stack \*/

void destroy()

{

top1 = top;

while (top1 != NULL)

{

top1 = top->ptr;

free(top);

top = top1;

top1 = top1->ptr;

}

free(top1);

top = NULL;

printf("\n All stack elements destroyed");

count = 0;

}

**Program 3**

Objective: WAP to implement Binary search (Divide & Conquer methods)

Code:

#include

int main()

{

int c, first, last, middle, n, search, array[100];

printf("Enter number of elements:\n");

scanf("%d",&n);

printf("Enter %d integers:\n", n);

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

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

printf("Enter the value to find:\n");

scanf("%d", &search);

first = 0;

last = n - 1;

middle = (first+last)/2;

while (first <= last) {

if (array[middle] < search)

first = middle + 1;

else if (array[middle] == search) {

printf("%d is present at index %d.\n", search, middle+1);

break;

}

else

last = middle - 1;

middle = (first + last)/2;

}

if (first > last)

printf("Not found! %d is not present in the list.\n", search);

return 0;

}

**Program 4**

Objective: WAP to implement Quick sorting (Divide & Conquer methods)

Code:

#include<stdio.h>

void quicksort(int number[25],int first,int last){

int i, j, pivot, temp;

if(first<last){

pivot=first;

i=first;

j=last;

while(i<j){

while(number[i]<=number[pivot]&&i<last)

i++;

while(number[j]>number[pivot])

j--;

if(i<j){

temp=number[i];

number[i]=number[j];

number[j]=temp;

}

}

temp=number[pivot];

number[pivot]=number[j];

number[j]=temp;

quicksort(number,first,j-1);

quicksort(number,j+1,last);

}

}

int main(){

int i, count, number[25];

printf("How many elements are u going to enter?: ");

scanf("%d",&count);

printf("Enter %d elements: ", count);

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

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

quicksort(number,0,count-1);

printf("Order of Sorted elements: ");

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

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

return 0;

}

**Program 5**

Objective: WAP to implement Merge sorting (Divide & Conquer methods)

Code:

Live Demo

#include <stdio.h>

#define max 10

int a[11] = { 10, 14, 19, 26, 27, 31, 33, 35, 42, 44, 0 };

int b[10];

void merging(int low, int mid, int high) {

int l1, l2, i;

for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++) {

if(a[l1] <= a[l2])

b[i] = a[l1++];

else

b[i] = a[l2++];

}

while(l1 <= mid)

b[i++] = a[l1++];

while(l2 <= high)

b[i++] = a[l2++];

for(i = low; i <= high; i++)

a[i] = b[i];

}

void sort(int low, int high) {

int mid;

if(low < high) {

mid = (low + high) / 2;

sort(low, mid);

sort(mid+1, high);

merging(low, mid, high);

} else {

return;

}

}

int main() {

int i;

printf("List before sorting\n");

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

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

sort(0, max);

printf("\nList after sorting\n");

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

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

}

**Program 6**

Objective: WAP to implement Multiplication of two matrices using strassen’s method

(Divide & Conquer methods)

Code:

#include<stdio.h>

int main(){

int a[2][2], b[2][2], c[2][2], i, j;

int m1, m2, m3, m4 , m5, m6, m7;

printf("Enter the 4 elements of first matrix: ");

for(i = 0;i < 2; i++)

for(j = 0;j < 2; j++)

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

printf("Enter the 4 elements of second matrix: ");

for(i = 0; i < 2; i++)

for(j = 0;j < 2; j++)

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

printf("\nThe first matrix is\n");

for(i = 0; i < 2; i++){

printf("\n");

for(j = 0; j < 2; j++)

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

}

printf("\nThe second matrix is\n");

for(i = 0;i < 2; i++){

printf("\n");

for(j = 0;j < 2; j++)

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

}

m1= (a[0][0] + a[1][1]) \* (b[0][0] + b[1][1]);

m2= (a[1][0] + a[1][1]) \* b[0][0];

m3= a[0][0] \* (b[0][1] - b[1][1]);

m4= a[1][1] \* (b[1][0] - b[0][0]);

m5= (a[0][0] + a[0][1]) \* b[1][1];

m6= (a[1][0] - a[0][0]) \* (b[0][0]+b[0][1]);

m7= (a[0][1] - a[1][1]) \* (b[1][0]+b[1][1]);

c[0][0] = m1 + m4- m5 + m7;

c[0][1] = m3 + m5;

c[1][0] = m2 + m4;

c[1][1] = m1 - m2 + m3 + m6;

printf("\nAfter multiplication using Strassen's algorithm \n");

for(i = 0; i < 2 ; i++){

printf("\n");

for(j = 0;j < 2; j++)

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

}

return 0;

}

**Program 7**

Objective: WAP to implement BFS(using adjacency matrix)

Code:

#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;

};

struct Graph\* createGraph(int vertices);

void addEdge(struct Graph\* graph, int src, int dest);

void printGraph(struct Graph\* graph);

void bfs(struct Graph\* graph, int startVertex);

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;

}

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)

{

// Add edge from src to dest

struct node\* newNode = createNode(dest);

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

graph->adjLists[src] = newNode;

// Add edge from dest to src

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]);

}

}

}

**Program 8**

Objective: WAP to implement DFS (using adjacency matrix)

Code:

#include<stdio.h>

void DFS(int);

int G[10][10],visited[10],n; //n is no of vertices and graph is sorted in array G[10][10]

void main()

{

int i,j;

printf("Enter number of vertices:");

scanf("%d",&n);

//read the adjecency matrix

printf("\nEnter adjecency matrix of the graph:");

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

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

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

//visited is initialized to zero

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

visited[i]=0;

DFS(0);

}

void DFS(int i)

{

int j;

printf("\n%d",i);

visited[i]=1;

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

if(!visited[j]&&G[i][j]==1)

DFS(j);

}

**Program 9**

Objective: WAP to implement krushkal’s algorithm to find minimum spanning tree.

Code: #include<stdio.h>

#include<stdlib.h>

#define infinity 9999

#define MAX 20

int G[MAX][MAX],spanning[MAX][MAX],n;

int prims();

int main()

{

int i,j,total\_cost;

printf("Enter no. of vertices:");

scanf("%d",&n);

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

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

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

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

total\_cost=prims();

printf("\nspanning tree matrix:\n");

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

{

printf("\n");

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

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

}

printf("\n\nTotal cost of spanning tree=%d",total\_cost);

return 0;

}

int prims()

{

int cost[MAX][MAX];

int u,v,min\_distance,distance[MAX],from[MAX];

int visited[MAX],no\_of\_edges,i,min\_cost,j;

//create cost[][] matrix,spanning[][]

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

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

{

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

cost[i][j]=infinity;

else

cost[i][j]=G[i][j];

spanning[i][j]=0;

}

//initialise visited[],distance[] and from[]

distance[0]=0;

visited[0]=1;

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

{

distance[i]=cost[0][i];

from[i]=0;

visited[i]=0;

}

min\_cost=0; //cost of spanning tree

no\_of\_edges=n-1; //no. of edges to be added

while(no\_of\_edges>0)

{

//find the vertex at minimum distance from the tree

min\_distance=infinity;

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

if(visited[i]==0&&distance[i]<min\_distance)

{

v=i;

min\_distance=distance[i];

}

u=from[v];

//insert the edge in spanning tree

spanning[u][v]=distance[v];

spanning[v][u]=distance[v];

no\_of\_edges--;

visited[v]=1;

//updated the distance[] array

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

if(visited[i]==0&&cost[i][v]<distance[i])

{

distance[i]=cost[i][v];

from[i]=v;

}

min\_cost=min\_cost+cost[u][v];

}

return(min\_cost);

}

**Program 10**

Objective: WAP to implement krushkal’s algorithm to find minimum spanning tree.

Code:

#include<stdio.h>

#define MAX 30

typedef struct edge

{

int u,v,w;

}edge;

typedef struct edgelist

{

edge data[MAX];

int n;

}edgelist;

edgelist elist;

int G[MAX][MAX],n;

edgelist spanlist;

void kruskal();

int find(int belongs[],int vertexno);

void union1(int belongs[],int c1,int c2);

void sort();

void print();

void main()

{

int i,j,total\_cost;

printf("\nEnter number of vertices:");

scanf("%d",&n);

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

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

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

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

kruskal();

print();

}

void kruskal()

{

int belongs[MAX],i,j,cno1,cno2;

elist.n=0;

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

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

{

if(G[i][j]!=0)

{

elist.data[elist.n].u=i;

elist.data[elist.n].v=j;

elist.data[elist.n].w=G[i][j];

elist.n++;

}

}

sort();

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

belongs[i]=i;

spanlist.n=0;

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

{

cno1=find(belongs,elist.data[i].u);

cno2=find(belongs,elist.data[i].v);

if(cno1!=cno2)

{

spanlist.data[spanlist.n]=elist.data[i];

spanlist.n=spanlist.n+1;

union1(belongs,cno1,cno2);

}

}

}

int find(int belongs[],int vertexno)

{

return(belongs[vertexno]);

}

void union1(int belongs[],int c1,int c2)

{

int i;

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

if(belongs[i]==c2)

belongs[i]=c1;

}

void sort()

{

int i,j;

edge temp;

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

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

if(elist.data[j].w>elist.data[j+1].w)

{

temp=elist.data[j];

elist.data[j]=elist.data[j+1];

elist.data[j+1]=temp;

}

}

void print()

{

int i,cost=0;

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

{

printf("\n%d\t%d\t%d",spanlist.data[i].u,spanlist.data[i].v,spanlist.data[i].w);

cost=cost+spanlist.data[i].w;

}

printf("\n\nCost of the spanning tree=%d",cost);

}

**Program 11**

Objective: WAP to implement Fractional knapsack (Dynamic Approach).

Code:

#include <stdio.h>

int n = 5; /\* The number of objects \*/

int c[10] = {12, 1, 2, 1, 4}; /\* c[i] is the \*COST\* of the ith object; i.e. what

YOU PAY to take the object \*/

int v[10] = {4, 2, 2, 1, 10}; /\* v[i] is the \*VALUE\* of the ith object; i.e.

what YOU GET for taking the object \*/

int W = 15; /\* The maximum weight you can take \*/

void simple\_fill() {

int cur\_w;

float tot\_v;

int i, maxi;

int used[10];

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

used[i] = 0; /\* I have not used the ith object yet \*/

cur\_w = W;

while (cur\_w > 0) { /\* while there's still room\*/

/\* Find the best object \*/

maxi = -1;

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

if ((used[i] == 0) &&

((maxi == -1) || ((float)v[i]/c[i] > (float)v[maxi]/c[maxi])))

maxi = i;

used[maxi] = 1; /\* mark the maxi-th object as used \*/

cur\_w -= c[maxi]; /\* with the object in the bag, I can carry less \*/

tot\_v += v[maxi];

if (cur\_w >= 0)

printf("Added object %d (%d$, %dKg) completely in the bag. Space left: %d.\n", maxi + 1, v[maxi], c[maxi], cur\_w);

else {

printf("Added %d%% (%d$, %dKg) of object %d in the bag.\n", (int)((1 + (float)cur\_w/c[maxi]) \* 100), v[maxi], c[maxi], maxi + 1);

tot\_v -= v[maxi];

tot\_v += (1 + (float)cur\_w/c[maxi]) \* v[maxi];

}

}

printf("Filled the bag with objects worth %.2f$.\n", tot\_v);

}

int main(int argc, char \*argv[]) {

simple\_fill();

return 0;

}

**Program 12**

Objective: WAP to implement 0/1 knapsack problem (Dynamic Approach).

Code:

// A Dynamic Programming based solution for 0-1 Knapsack problem

#include<stdio.h>

// A utility function that returns maximum of two integers

int max(int a, int b) { return (a > b)? a : b; }

// Returns the maximum value that can be put in a knapsack of capacity W

int knapSack(int W, int wt[], int val[], int n)

{

int i, w;

int K[n+1][W+1];

// Build table K[][] in bottom up manner

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

{

for (w = 0; w <= W; w++)

{

if (i==0 || w==0)

K[i][w] = 0;

else if (wt[i-1] <= w)

K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w]);

else

K[i][w] = K[i-1][w];

}

}

return K[n][W];

}

int main()

{

int val[] = {60, 100, 120};

int wt[] = {10, 20, 30};

int W = 50;

int n = sizeof(val)/sizeof(val[0]);

printf("%d", knapSack(W, wt, val, n));

return 0;

}