DISJOINT SET:

#include <stdio.h>

#include <stdlib.h>

struct node {

struct node \*rep;

struct node \*next;

int data;

} \*heads[50], \*tails[50];

static int countRoot = 0;

void makeSet(int x) {

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

new->rep = new;

new->next = NULL;

new->data = x;

heads[countRoot] = new;

tails[countRoot] = new;

countRoot++;

}

struct node\* find(int a) {

int i;

struct node \*tmp;

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

tmp = heads[i];

while (tmp != NULL) {

if (tmp->data == a)

return tmp->rep;

tmp = tmp->next;

}

}

return NULL;

}

void unionSets(int a, int b) {

int i, j, pos, flag = 0;

struct node \*tail2;

struct node \*rep1 = find(a);

struct node \*rep2 = find(b);

if (rep1 == NULL || rep2 == NULL) {

printf("\nElement(s) not present in the DS\n");

return;

}

if (rep1 != rep2) {

for (j = 0; j < countRoot; j++) {

if (heads[j] == rep2) {

pos = j;

flag = 1;

countRoot -= 1;

tail2 = tails[j];

for (i = pos; i < countRoot; i++) {

heads[i] = heads[i+1];

tails[i] = tails[i+1];

}

break;

}

}

for (j = 0; j < countRoot; j++) {

if (heads[j] == rep1) {

tails[j]->next = rep2;

tails[j] = tail2;

break;

}

}

while (rep2 != NULL) {

rep2->rep = rep1;

rep2 = rep2->next;

}

}

}

int search(int x) {

int i;

struct node \*tmp;

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

tmp = heads[i];

while (tmp != NULL) {

if (tmp->data == x)

return 1;

tmp = tmp->next;

}

}

return 0;

}

void displayRepresentatives() {

printf("\nSet Representatives: ");

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

printf("%d ", heads[i]->data);

}

printf("\n");

}

void displaySets() {

int i, j;

struct node \*temp;

printf("\nDisjoint Sets:\n");

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

temp = heads[i];

printf("{ ");

int first = 1;

while (temp != NULL) {

if (!first) printf(", ");

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

first = 0;

temp = temp->next;

}

printf(" }\n");

}

}

int main() {

int choice, x, y, setSize,temp=0;

printf("Enter the size of the set (1 to 50): ");

scanf("%d", &setSize);

while (setSize <= 0 || setSize > 50) {

printf("Invalid set size. Please enter a size between 1 and 50: ");

scanf("%d", &setSize);

}

do {

printf("\n1. Make Set");

printf("\n2. Display set representatives");

printf("\n3. Union");

printf("\n4. Find Set");

printf("\n5. Display all sets");

printf("\n6. Exit");

printf("\nEnter your choice: ");

scanf("%d", &choice);

switch(choice) {

case 1:

if(temp>setSize-1){

printf("Elements reached at it's Maxsize\n");

break;

}

printf("Enter the Element to Make a Set: ");

scanf("%d",&x);

if(search(x)){

printf("\nElement %d is already Exist In the Set, Enter the Unique Element.\n",x);

}

else{

makeSet(x);

temp++;

}

break;

case 2:

displayRepresentatives();

break;

case 3:

printf("\nEnter first element: ");

scanf("%d", &x);

printf("Enter second element: ");

scanf("%d", &y);

unionSets(x, y);

temp--;

if(temp<0){

temp=0;

}

//printf("\nSlot: %d\n",temp);

break;

case 4:

printf("\nEnter the element to find: ");

scanf("%d", &x);

struct node \*rep = find(x);

if (rep == NULL) {

printf("\nElement not present in the DS\n");

} else {

printf("\nThe representative of %d is %d\n", x, rep->data);

}

break;

case 5:

displaySets();

break;

case 6:

printf("\nExiting program...\n");

exit(0);

default:

printf("\nInvalid choice! Please try again.\n");

break;

}

} while (1);

return 0;

}

BIT VECTOR:

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 20

int superSet[MAX\_SIZE], superSetSize = 0;

int setA[MAX\_SIZE], setASize = 0;

int setB[MAX\_SIZE], setBSize = 0;

int bitStringA[MAX\_SIZE], bitStringB[MAX\_SIZE];

// Function prototypes

void getUniversalSet();

void getSet(int arr[], int \*size);

int checkSetInUniversal(int arr[], int size);

void generateBitStrings();

void setUnion();

void setIntersection();

void setDifferenceAminusB();

void setDifferenceBminusA();

void printBitString(int arr[], int size);

void printSetFromBitString(int arr[], int size);

void getUniversalSet() {

printf("Enter Universal Set Size (max %d): ", MAX\_SIZE);

scanf("%d", &superSetSize);

if (superSetSize > MAX\_SIZE) {

printf("Error: Size exceeds maximum limit.\n");

exit(1);

}

printf("Enter %d elements for the Universal Set:\n", superSetSize);

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

printf("Element %d: ", i + 1);

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

}

}

void getSet(int arr[], int \*size) {

printf("Enter %d elements (must be in the Universal Set):\n", \*size);

for (int i = 0; i < \*size; i++) {

printf("Element %d: ", i + 1);

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

}

}

int checkSetInUniversal(int arr[], int size) {

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

int found = 0;

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

if (arr[i] == superSet[j]) {

found = 1;

break;

}

}

if (!found) {

printf("Error: Element %d is not in the Universal Set. Please enter the set again.\n", arr[i]);

return 0;

}

}

return 1;

}

void generateBitStrings() {

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

bitStringA[i] = 0;

bitStringB[i] = 0;

}

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

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

if (setA[i] == superSet[j]) {

bitStringA[j] = 1;

break;

}

}

}

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

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

if (setB[i] == superSet[j]) {

bitStringB[j] = 1;

break;

}

}

}

printf("Set A Bit String: ");

printBitString(bitStringA, superSetSize);

printf("Set B Bit String: ");

printBitString(bitStringB, superSetSize);

}

void setUnion() {

int bitStringUnion[MAX\_SIZE];

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

bitStringUnion[i] = bitStringA[i] | bitStringB[i];

}

printf("Union: ");

printSetFromBitString(bitStringUnion, superSetSize);

printf("Union Bit String");

printBitString(bitStringUnion,superSetSize);

}

void setIntersection() {

int bitStringIntersection[MAX\_SIZE];

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

bitStringIntersection[i] = bitStringA[i] & bitStringB[i];

}

printf("Intersection: ");

printSetFromBitString(bitStringIntersection, superSetSize);

printBitString(bitStringIntersection,superSetSize);

}

void setDifferenceAminusB() {

int bitStringDifferenceAminusB[MAX\_SIZE];

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

bitStringDifferenceAminusB[i] = bitStringA[i] & (1 - bitStringB[i]);

}

printf("Difference (A - B): ");

printSetFromBitString(bitStringDifferenceAminusB, superSetSize);

printBitString(bitStringDifferenceAminusB,superSetSize);

}

void setDifferenceBminusA() {

int bitStringDifferenceBminusA[MAX\_SIZE];

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

bitStringDifferenceBminusA[i] = bitStringB[i] & (1 - bitStringA[i]);

}

printf("Difference (B - A): ");

printSetFromBitString(bitStringDifferenceBminusA, superSetSize);

printBitString(bitStringDifferenceBminusA,superSetSize);

}

void printBitString(int arr[], int size) {

printf("{");

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

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

if (i < size - 1) {

printf(", ");

}

}

printf("}\n");

}

void printSetFromBitString(int arr[], int size) {

int first = 1;

printf("{");

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

if (arr[i] == 1) {

if (!first) {

printf(", ");

}

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

first = 0;

}

}

printf("}\n");

}

int main() {

int choice;

getUniversalSet();

do {

printf("Enter Set A Size (max %d): ", superSetSize);

scanf("%d", &setASize);

if (setASize > superSetSize) {

printf("Error: Set A size cannot exceed Universal Set size.\n");

}

} while (setASize > superSetSize);

do {

getSet(setA, &setASize);

} while (checkSetInUniversal(setA, setASize) == 0);

do {

printf("Enter Set B Size (max %d): ", superSetSize);

scanf("%d", &setBSize);

if (setBSize > superSetSize) {

printf("Error: Set B size cannot exceed Universal Set size.\n");

}

} while (setBSize > superSetSize);

do {

getSet(setB, &setBSize);

} while (checkSetInUniversal(setB, setBSize) == 0);

generateBitStrings();

do {

printf("\nChoose an operation:\n");

printf("1. Union of A and B\n");

printf("2. Intersection of A and B\n");

printf("3. Difference (A - B)\n");

printf("4. Difference (B - A)\n");

printf("5. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

setUnion();

break;

case 2:

setIntersection();

break;

case 3:

setDifferenceAminusB();

break;

case 4:

setDifferenceBminusA();

break;

case 5:

printf("Exiting program.\n");

break;

default:

printf("Invalid choice. Please try again.\n");

}

} while (choice != 5);

return 0;

}

BINARY SEARCH TREE:

#include<stdio.h>

#include<stdlib.h>

struct Node{

int data;

struct Node\* left;

struct Node\* right;

};

//struct Node\* root=NULL;

//functn to create a new node

struct Node\* createnode(int data)

{

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

newNode->data=data;

newNode->left=NULL;

newNode->right=NULL;

return newNode;

}

struct Node\* insert(struct Node\* root,int data)

{

if(root==NULL)

{

root=createnode(data);

}

else if(data<root->data)

{

root->left=insert(root->left,data);

}

else if(data>root->data)

{

root->right=insert(root->right,data);

}

return root;

}

struct Node\* findMin(struct Node \*root)

{

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

{

root=root->left;

}

return root;

}

//dlt a node from bst

struct Node\* deleteNode(struct Node\* root,int data){

if(root==NULL){

printf("The value to be deleted is not present in the tree\n");

return root;

}

if(data<root->data){

root->left=deleteNode(root->left,data);

}else if(data>root->data){

root->right=deleteNode(root->right,data);

}else{

//node with one child or no child

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;

}

//node with 2 children

struct Node\* temp=findMin(root->right);

root->data=temp->data;

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

}

return root;

}

//search a node in bst

struct Node\* search(struct Node\* root,int data){

if(root==NULL||root->data==data){

return root;

}

if(data<root->data){

return search(root->left,data);

}

else{

return search(root->right,data);

}

}

//preorder traversal

void preorder(struct Node\* root){

if(root!=NULL){

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

preorder(root->left);

preorder(root->right);

}

}

//inorder traversal

void inorder(struct Node\* root){

if(root!=NULL){

inorder(root->left);

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

inorder(root->right);

}

}

//postorder traversal

void postorder(struct Node\* root){

if(root!=NULL){

postorder(root->left);

postorder(root->right);

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

}

}

int main(){

struct Node\* root=NULL;

int choice,value;

struct Node\* foundNode;

while(1){

printf("1.INSERT NODE\n2.DELETE NODE\n3.SEARCH NODE\n4.PREORDER TRAVERSAL\n5.INORDER TRAVERSAL\n6.POSTORDER TRAVERSAL\n7.EXIT\n");

printf("Enter your choice:");

scanf("%d",&choice);

switch(choice){

case 1:

printf("enter the value to be inserted :");

scanf("%d",&value);

root=insert(root,value);

break;

case 2:

if(root==NULL){

printf("tree is empty \n");

}

else{

printf("enter the value to delete:");

scanf("%d",&value);

root=deleteNode(root,value);

}

break;

case 3:

if(root==NULL){

printf("tree is empty");

}

else{

printf("enter value to search:");

scanf("%d",&value);

foundNode=search(root,value);

if(foundNode!=NULL){

printf("value %d found in the tree ",value);

}else{

printf("value %d not found in the tree",value);

}

}

break;

case 4:

if(root==NULL){

printf("tree is empty");

}

else{

printf("preorder traversal:\n");

preorder(root);

printf("\n");

}

break;

case 5:

if(root==NULL){

printf("tree is empty");

}

else{

printf("inorder traversal:\n");

inorder(root);

printf("\n");

}

break;

case 6:

if(root==NULL){

printf("tree is empty");

}

else{

printf("postorder traversal:\n");

postorder(root);

printf("\n");

}

break;

case 7:

exit(0);

default:

printf("invalid choice!please try again\n");

}

}

return 0;

}

CIRCULAR QUEUE:

#include <stdio.h>

#include <stdlib.h>

int \*queue;

int size;

int front = -1, rear = -1;

void initializeQueue() {

queue = (int \*)malloc(size \* sizeof(int));

}

void enqueue(int element) {

if (front == (rear + 1) % size) {

printf("\nQUEUE IS FULL\n");

return;

}

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

front = rear = 0;

} else {

rear = (rear + 1) % size;

}

queue[rear] = element;

printf("\n%d is Inserted\n",element);

}

int dequeue() {

int element;

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

printf("\nQUEUE IS EMPTY\n");

return -1;

}

element = queue[front];

if (front == rear) {

front = rear = -1;

} else {

front = (front + 1) % size;

}

printf("\n%d ELEMENT IS DELETED FROM THE QUEUE\n", element);

return element;

}

int searchElement(int element) {

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

printf("\nQUEUE IS EMPTY\n");

return -1;

}

int current = front;

int position = 1;

do {

if (queue[current] == element) {

return position;

}

current = (current + 1) % size;

position++;

} while (current != (rear + 1) % size);

return -1;

}

void displayQueue() {

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

printf("\nQUEUE IS EMPTY\n");

return;

}

printf("QUEUE ELEMENTS ARE: ");

int current = front;

do {

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

current = (current + 1) % size;

} while (current != (rear + 1) % size);

printf("\n");

}

int main() {

int choice, searchResult, element;

printf("ENTER THE SIZR OF THE QUEUE: ");

scanf("%d", &size);

initializeQueue();

do {

printf("\nCIRCULAR QUEUE MENU\n");

printf("1. Enqueue\n");

printf("2. Dequeue\n");

printf("3. Search Element\n");

printf("4. Display\n");

printf("5. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the element to enqueue: ");

scanf("%d", &element);

enqueue(element);

break;

case 2:

dequeue();

break;

case 3:

printf("Enter the element to search: ");

scanf("%d", &element);

searchResult = searchElement(element);

if (searchResult != -1) {

printf("%d found at position %d\n", element, searchResult);

} else {

printf("%d not found in the queue\n", element);

}

break;

case 4:

displayQueue();

break;

case 5:

printf("Exiting!!!.\n");

break;

default:

printf("Invalid choice. Please enter a valid option.\n");

break;

}

} while (choice != 5);

free(queue);

return 0;

}

DOUBLY LINKED LIST:

#include<stdio.h>

#include<stdlib.h>

struct node{

int data;

struct node\* next;

struct node\* prev;

};

struct node\*head=NULL;

void display(){

if (head == NULL) {

printf("\nList is empty\n");

return;

}

struct node\*temp=head;

while (temp->next!=NULL){

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

temp=temp->next;

}

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

}

void ins\_beg(int item){

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

newnode->data = item;

newnode->next = head;

newnode->prev = NULL;

if (head != NULL){

head->prev=newnode;

}

head=newnode;

}

void ins\_end(int item){

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

newnode->next=NULL;

newnode->data=item;

if (head==NULL){

head=newnode;

return;

}

struct node\*temp=head;

while (temp->next!=NULL){

temp=temp->next;

}

newnode->prev=temp;

temp->next=newnode;

}

void ins\_pos(int item,int pos){

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

newnode->data = item;

if (pos == 1) {

ins\_beg(item);

return;

}

struct node\* temp = head;

while(pos!=2){

temp=temp->next;

pos--;

}

newnode->next=temp->next;

newnode->prev=temp;

temp->next->prev=newnode;

temp->next=newnode;

}

void del\_beg(){

struct node\*temp=head;

if(head==NULL){

printf("\nList is empty\n");

return;

}

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

head=temp->next;

if (head!=NULL){

temp->next->prev=NULL;

}

free(temp);

}

void del\_end(){

struct node\*temp=head;

if(head==NULL){

printf("\nList is empty\n");

return;

}

if (temp->next==NULL) {

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

free(temp);

head = NULL;

return;

}

while(temp->next!=NULL){

temp=temp->next;

}

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

temp->prev->next=NULL;

free(temp);

}

void del\_pos() {

if (head==NULL){

printf("\nList is empty\n");

return;

}

int pos;

printf("\nEnter the position of element to be deleted:");

scanf("%d",&pos);

struct node\*temp=head;

if (pos<1) {

printf("\nInvalid position\n");

return;

}

if (pos==1) {

del\_beg();

return;

}

while(pos!=1)

{

temp=temp->next;

pos--;

}

if (temp->next==NULL){

del\_end();

return;

}

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

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

printf("%d is deleted from the position %d.\n",temp->data,pos);

free(temp);

return;

}

void main(){

int ch;

while (1)

{

printf("\n1.INSERT AT BEG\n2.INSERT AT END\n3.INSERT AT POS\n4.DELETE AT BEG\n5.DELETE AT END\n6.DELETE AT POS\n7.DISPLAY\n8.EXIT\n\nENTER YOUR CHOICE: ");

scanf("%d",&ch);

switch (ch)

{

case 1:

int item;

printf("\nEnter the element to be inserted:");

scanf("%d",&item);

ins\_beg(item);

break;

case 2:

int item2;

printf("\nEnter the element to be inserted:");

scanf("%d",&item2);

ins\_end(item2);

break;

case 3:

int item3,pos2;

printf("\nEnter the element to be inserted:");

scanf("%d",&item3);

printf("\nEnter the position of element to be inserted:");

scanf("%d",&pos2);

ins\_pos(item3,pos2);

break;

case 4:

del\_beg();

break;

case 5:

del\_end();

break;

case 6:

del\_pos();

break;

case 7:

display();

break;

case 8:

printf("\n!!!....Exiting....!!!\n");

exit(0);

break;

default:

printf("\nInvalid choice!");

break;

}

}

}

SINGLY LINKED LIST:

#include<stdio.h>

#include<stdlib.h>

struct node{

int data;

struct node\*next;

};

struct node\*head=NULL;

void ins\_beg(int item){

if(head==NULL){

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

newNode->data=item;

newNode->next=NULL;

head=newNode;

}

else{

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

newNode->data=item;

newNode->next=head;

head=newNode;

printf("\n%d is Inserted in the list",newNode->data);

}

}

void ins\_end(int item){

struct node\*temp=head;

if(head==NULL){

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

newNode->data=item;

newNode->next=NULL;

head=newNode;

}

else{

while(temp->next!=NULL){

temp=temp->next;

}

if(temp->next==NULL){

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

newNode->data=item;

temp->next=newNode;

newNode->next=NULL;

}

}

}

void ins\_pos(int item,int pos){

int count=2;

struct node\*temp=head;

if(pos==1){

ins\_beg(item);

}

else{

while(temp->next!=NULL){

if(count==pos){

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

newNode->data=item;

newNode->next=temp->next;

temp->next=newNode;

}

count=count+1;

temp=temp->next;

}

}

}

void del\_beg(){

if(head==NULL){

printf("\nThe List is Empty\n");

}

else{

struct node\*temp=head;

head=temp->next;

printf("\n %d is Deleted from the list",temp->data);

free(temp);

}

}

void del\_end(){

struct node\*temp=head;

struct node\*prev;

if(head==NULL){

printf("\nThe List is Empty\n");

}

else{

while(temp->next!=NULL){

prev=temp;

temp=temp->next;

}

prev->next=NULL;

printf("\n%d is Deleted from the List",temp->data);

free(temp);

}

}

void del\_pos(int pos){

struct node\*temp=head;

struct node\*prev=NULL;

if (pos < 1) {

printf("\nInvalid position\n");

return;

}

if (pos == 1) {

del\_beg();

return;

}

while(pos!=1){

prev=temp;

temp=temp->next;

pos--;

}

prev->next=temp->next;

printf("%d is Deleted from the position %d.\n",temp->data,pos+1);

free(temp);

temp=NULL;

return;

if(head!=NULL)

prev->next = temp->next;

free(temp);

}

void trav(){

struct node\*temp=head;

if(head==NULL){

printf("\nThe List is Empty\n\n");

}

else{

printf("\nThe List Elements are:\n\n");

while(temp->next!=NULL){

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

temp=temp->next;

}

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

}

}

int main(){

while(1){

int ch;

int item;

int cpos;

printf("\n\n\_\_\_\_\_\_\_\_\_\_\_\_\_Singly Linked List Operations\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");

printf("1.Insertion at Begining\n2.Insertion at the End\n3.Insertion at Specific Position\n4.Deletion at Begining\n5.Deletion at the End\n6.Deletion at Specific Position\n7.Display the List\n8.Quit\n");

printf("\nEnter your Choice: ");

scanf("%d",&ch);

switch(ch){

case 1:

printf("\nEnter the Element to be Inserted: ");

scanf("%d",&item);

ins\_beg(item);

break;

case 2:

printf("\nEnter the Element to be Inserted: ");

scanf("%d",&item);

ins\_end(item);

break;

case 3:

printf("\nEnter the Element to be Inserted: ");

scanf("%d",&item);

printf("\nEnter the Position:");

scanf("%d",&cpos);

ins\_pos(item,cpos);

break;

case 4:

del\_beg();

break;

case 5:

del\_end();

break;

case 6:

printf("Enter the Position: ");

scanf("%d",&cpos);

del\_pos(cpos);

break;

case 7:

trav();

break;

case 8:

printf("\nExiting...............\n");

exit(1);

default:

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

break;

}

}

return 0;

}

STACK USING LINKED LIST:

#include<stdio.h>

#include<stdlib.h>

struct node

{

int data;

struct node\*link;

};

struct node \*top = NULL;

int isEmpty()

{

if(top==NULL)

{

return 1;

}

else{

return 0;

}

}

void push(int data)

{

struct node \*newNode;

newNode = malloc(sizeof(newNode));

if(newNode==NULL)

{

printf("Stack underflow\n");

}

newNode->data=data;

newNode->link=NULL;

newNode->link=top;

top=newNode;

}

int pop()

{

struct node\*temp;

int val;

if(isEmpty())

{

printf("Stack Underflow");

}

temp=top;

val=temp->data;

top=temp->link;

free(temp);

return val;

}

int peek()

{

if(isEmpty())

{

printf("Stack underflow");

}

return top->data;

}

void search(int item)

{

struct node \*temp=top;

int flag=0;

while(temp!=NULL)

{

if(item==temp->data)

{

printf("%d is Included in the stack\n",temp->data);

flag=1;

}

temp=temp->link;

}

if(flag!=1){

printf("\nElement not found\n");

}

}

void print()

{

struct node\*temp=top;

if(isEmpty())

{

printf("Stack Underflow\n");

}

printf("The Stack Elements are\n\n");

while(temp)

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

temp=temp->link;

}

printf("\n");

}

int main()

{

int choice,data,sitem;

while(1)

{

printf("\n");

printf("1.Push\n2.Pop\n3.Print the Top element\n4.Print all the Stack Element\n5.Search Element\n6.Exit");

printf("\n\nPlease Enter your choice\n");

scanf("%d",&choice);

switch(choice)

{

case 1:

printf("Enter the element to be Pushed\n");

scanf("%d",&data);

push(data);

break;

case 2:

data=pop();

printf("%d Element Removed\n",data);

break;

case 3:

printf("The Top most Element of the Stack is %d\n",peek());

break;

case 4:

print();

break;

case 5:

printf("Enter the item to be searched\n");

scanf("%d",&sitem);

search(sitem);

break;

case 6:

exit(1);

default:

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

}

}

return 0;

}