**SINGLY LINKED LIST**

#include<stdio.h>

#include<stdlib.h>

#include<malloc.h>

struct node

{

int data;

struct node\*next;

};

struct node\*head=NULL;

void insertfirst()

{

struct node\*newnode;

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

{

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

scanf("%d",&newnode->data);

newnode->next=NULL;

if(head==NULL)

head=newnode;

else

{

newnode->next=head;

head=newnode;

}

printf("\n %d inserted into the list",newnode->data);

}

}

void insertlast()

{

struct node\*temp,\*newnode;

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

newnode->next=NULL;

temp=head;

{

while(temp->next!=NULL)

temp=temp->next;

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

scanf("%d",&newnode->data);

temp->next=newnode;

printf("\n %d inserted into the list",newnode->data);

}

}

void insertlocation()

{

int key;

struct node\*temp,\*newnode;

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

newnode->next=NULL;

printf("\n enter the value of node after new node to be inserted :");

scanf("%d",&key);

temp=head;

while(temp->data!=key)

{

temp=temp->next;

if(temp==NULL)

break;

}

if(temp==NULL)

{

printf("The %d value not exist",key);

return;

}

if(temp->data==key)

{

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

scanf("%d",&newnode->data);

newnode->next=temp->next;

temp->next=newnode;

printf("\n %d inserted after %d",newnode->data,key);

}

}

void deletefirst()

{

struct node\*temp;

temp=head;

head=temp->next;

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

free(temp);

}

void deletelast()

{

struct node\*temp,\*p;

temp=head;

while(temp->next!=NULL)

{

p=temp;

temp=temp->next;

}

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

free(temp);

p->next=NULL;

}

void deletelocation()

{

struct node\*temp,\*p;

int key;

printf("\n enter the value of node to be deleted :");

scanf("%d",&key);

temp=head;

while(temp->data!=key)

{

p=temp;

temp=temp->next;

if(temp==NULL)

break;

}

if(temp==NULL)

{

printf("The %d value not exist",key);

return;

}

if(temp->data==key)

{

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

if(temp==head)

{

p=head;

head=temp->next;

free(p);

}

else

{

p->next=temp->next;

free(temp);

}

}

}

void search()

{

struct node\*temp;

int key,pos=0;

temp=head;

printf("\n enter the element to be searched :");

scanf("%d",&key);

while(temp->data!=key)

{

temp=temp->next;

pos++;

if(temp==NULL)

break;

}

if(temp==NULL)

{

printf("\n The %d value not exist",key);

return;

}

if(temp->data==key)

{

printf("\n %d value available in location %d",key,pos+1);

}

}

void display()

{

struct node\*p;

if(head==NULL)

printf("list is empty");

else

{

printf("\n elements in linked list are:");

p=head;

while(p!=NULL)

{

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

p=p->next;

}

}

}

void main()

{

int choice;

printf("\n SINGLY LINKED LIST \n ");

do

{

printf("\n\n 1.insert in beggining \n2. insert at last\n3. insert at any random location \n4. delete from beggining \n5. delete from last\n6. delete node after specified location \n7. search for an element \n 8. display \n 9. exit" );

printf("\n enter a choice :");

scanf("%d",&choice);

switch(choice)

{

case 1:insertfirst();

break;

case 2:insertlast();

break;

case 3:insertlocation();

break;

case 4:deletefirst();

break;

case 5:deletelast();

break;

case 6:deletelocation();

break;

case 7:search();

break;

case 8:display();

break;

case 9:exit(0);

printf("exiting the program ");

return;

default:printf("\n invalid option");

}

}

while(choice!=9);

}

**Circular queue**

#include <stdio.h>

#define max 20

int queue[max],front=-1,rear=-1,choice=1,x, i;

void enqueue(int element)

{

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

{

front=0;

rear=0;

queue[rear]=element;

}

else if((rear+1)%max==front)

{

printf("Queue is overflow..");

}

else

{

rear=(rear+1)%max;

queue[rear]=element;

}

}

int dequeue()

{

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

{

printf("\nQueue is underflow..");

}

else if(front==rear)

{

printf("\n The dequeued element is %d", queue[front]);

front=-1;

rear=-1;

}

else

{

printf("\n The dequeued element is %d", queue[front]);

front=(front+1)%max;

}

}

void search()

{

int f=0, i, k;

if(front==-1)

{

printf("queue is empty ");

}

printf("enter element to search");

scanf("%d", &k);

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

{

if(queue[i]==k)

{

f=1;

break;

}}

if(f==1)

{

printf("element found at location %d \n", i+1);

}

else

{

printf("element not found\n.");

}

}

void display()

{

int i=front;

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

{

printf("\n Queue is empty");

}

else

{

printf("\n Elements in a Queue are :");

while(i<=rear)

{

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

i=(i+1)%max;

}

}

}

int main()

{

while(choice<5 && choice!=0)

{

printf("\n Circular Queue \n");

printf(" --------------");

printf("\n 1. Enqueue ");

printf("\n 2. Dequeue ");

printf("\n 3. Display ");

printf("\n 4. search ");

printf("\n Enter your choice ");

scanf("%d", &choice);

switch(choice)

{

case 1:

printf("Enter the element to be inserted -");

scanf("%d", &x);

enqueue(x);

break;

case 2:

dequeue();

break;

case 3:

display();

break;

case 4:

search();

break;

}}

return 0;

}

**BFS**

**#include<stdio.h>**

**void BFS(int);**

**int graph[10][10], visited[10],total;**

**int main()**

**{**

**int i,j;**

**printf("\nEnter the total number of vertices in graph\n");**

**scanf("%d",&total);**

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

**for(i=0;i<total;i++)**

**{**

**for(j=0;j<total;j++)**

**{**

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

**}**

**}**

**for(i=0;i<total;i++)**

**{**

**visited[i] = 0;**

**}**

**printf("\nBFS traversal is \n\n");**

**BFS(0);**

**}**

**void BFS(int vertex)**

**{**

**int j;**

**printf("%d\t\n",vertex);**

**visited[vertex] = 1;**

**for(j=0;j<total;j++)**

**{**

**if(!visited[j] && graph[vertex][j] == 1 )**

**{**

**BFS(j);**

**}**

**}}**

**DFS**

**/\*Implementation of DFS traverasal mechanism\*/**

**#include<stdio.h>**

**#include<stdlib.h>**

**int graph[10][10], visited[10],total,arr[30];**

**static int k=0,count=0;**

**void DFS(int);**

**void main()**

**{**

**int i,j;**

**printf("\nEnter the total number of vertices in graph\n");**

**scanf("%d",&total);**

**/\*Adjacency matrix input\*/**

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

**for(i=0;i<total;i++)**

**{**

**for(j=0;j<total;j++)**

**{**

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

**}**

**}**

**for(i=0;i<total;i++)**

**{**

**visited[i] = 0;**

**}**

**printf("\nDFS traversal is \n");**

**DFS(0);**

**}**

**void DFS(int vertex)**

**{**

**int j,c=0;**

**count++;**

**printf("%d\t",vertex);**

**visited[vertex] = 1;**

**for(j=0;j<total;j++)**

**{**

**if(!visited[j] && graph[vertex][j] == 1)**

**{**

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

**c=1;**

**}**

**if(count == total)**

**{**

**exit(0);**

**}**

**}**

**if(c==1)**

**{**

**DFS(arr[k]);**

**}**

**else**

**{**

**k--;**

**DFS(arr[k]);**

**}**

**}**

**BUBBLE SORT**

**#include <stdio.h>**

**int main()**

**{**

**int array[100], n, c, d, swap;**

**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]);**

**for (c = 0 ; c < n - 1; c++)**

**{**

**for (d = 0 ; d < n - c - 1; d++)**

**{**

**if (array[d] > array[d+1])**

**{**

**swap = array[d];**

**array[d] = array[d+1];**

**array[d+1] = swap;**

**}**

**}**

**}**

**printf("Sorted list in ascending order:\n");**

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

**printf("%d\n", array[c]);**

**return 0;**

**}**

**LINEAR SEARCH**

**#include <stdio.h>**

**int main()**

**{**

**int array[100], search, c, n;**

**printf("Enter number of elements in array\n");**

**scanf("%d", &n);**

**printf("Enter %d integer(s)\n", n);**

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

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

**printf("Enter a number to search\n");**

**scanf("%d", &search);**

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

**{**

**if (array[c] == search)**

**{**

**printf("%d is present at location %d.\n", search, c+1);**

**break;**

**}**

**}**

**if (c == n)**

**printf("%d isn't present in the array.\n", search);**

**return 0;**

**}**

**MERGE 2 SORTED ARRAY**

**#include <stdio.h>**

**int main(void)**

**{**

**int i, n, j, k;**

**printf("Enter the size of the first array: ");**

**scanf("%d", &n);**

**int arr1[n];**

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

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

**{**

**scanf("%d", &arr1[i]);**

**}**

**printf("Enter the size of the second array: ");**

**scanf("%d", &k);**

**int arr2[k];**

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

**for (j = 0; j < k; j++)**

**{**

**scanf("%d", &arr2[j]);**

**}**

**int arr3[n + k];**

**i = j = 0;**

**int in;**

**for (in = 0; in < n + k; in ++)**

**{**

**if (i < n && j < k)**

**{**

**if (arr1[i] < arr2[j])**

**{**

**arr3[in] = arr1[i];**

**i++;**

**}**

**else**

**{**

**arr3[in] = arr2[j];**

**j++;**

**}**

**}**

**else if (i < n)**

**{**

**arr3[in] = arr1[i];**

**i++;**

**}**

**else**

**{**

**arr3[in] = arr2[j];**

**j++;**

**}**

**}**

**printf("The merged array is: \n");**

**for (in = 0; in < n + k; in++)**

**{**

**printf("%d ", arr3[in]);**

**}**

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

**return 0;**

**}**

**DOUBLY LINKED LIST**

**#include <stdio.h>**

**#include <stdlib.h>**

**struct node {**

**struct node \*prev;**

**struct node \*next;**

**int data;**

**};**

**struct node \*new=NULL,\*ptr=NULL,\*head=NULL, \*nextptr=NULL, \*prevptr=NULL,\*last=NULL;**

**void insertion\_last();**

**void r\_display();**

**void delete\_last();**

**void insertion\_beginning();**

**void delete\_beginning();**

**void display();**

**void insert\_after\_key();**

**void delete();**

**void create\_node();**

**void search();**

**void main() {**

**int opt;**

**do {**

**printf("Doubly Linked List");**

**printf("\n-------------------\n \n");**

**printf("1.Insertion Beginning \n2.Insert after key \n3.Insertion last\n4.Delete beginning \n5.Delete last\n6.Delete specific Key \n7.Display \n8.Reverse display \n9.search \n10. exit");**

**printf("\nChoose operation : ");**

**scanf("%d",&opt);**

**switch(opt) {**

**case 1: insertion\_beginning();**

**break;**

**case 2: insert\_after\_key();**

**break;**

**case 3: insertion\_last();**

**break;**

**case 4: delete\_beginning();**

**break;**

**case 5: delete\_last();**

**break;**

**case 6: delete();**

**break;**

**case 7: display();**

**break;**

**case 8: r\_display();**

**break;**

**case 9: search();**

**break;**

**case 10:**

**default:break;**

**}**

**} while(opt!=10);**

**}**

**void create\_node() {**

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

**printf("Enter the data to be inserted : ");**

**scanf("%d",&new->data);**

**new->next = NULL;**

**new->prev = NULL;**

**}**

**void insertion\_beginning() {**

**create\_node();**

**if(head == NULL){**

**head = new;**

**} else {**

**head->prev=new;**

**new->next = head;**

**head = new;**

**}**

**}**

**void insertion\_last() {**

**create\_node();**

**if(head == NULL) {**

**head = new;**

**}**

**else {**

**ptr=head;**

**while(ptr->next!=NULL){**

**ptr = ptr->next;**

**}**

**ptr->next = new;**

**new->prev = ptr;**

**}**

**}**

**void insert\_after\_key() {**

**int key;**

**create\_node();**

**printf("Enter the key where the node to be inserted : ");**

**scanf("%d",&key);**

**ptr=head;**

**while(ptr->data != key && ptr->next != NULL){**

**ptr = ptr->next;**

**}**

**if(ptr->data == key){**

**new->next = ptr->next;**

**new->prev = ptr;**

**ptr->next->prev = new;**

**ptr->next = new;**

**}**

**else {**

**printf("The key is not found\n ");**

**}**

**}**

**void delete\_beginning() {**

**if(head == NULL){**

**printf("Empty List");**

**}**

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

**head = NULL;**

**free(head);**

**printf("NODE DELETED\n");**

**}**

**else {**

**ptr=head;**

**head = head->next;**

**head->prev = NULL;**

**free(ptr);**

**printf("NODE DELETED\n");**

**}**

**}**

**void delete\_last() {**

**if(head == NULL) {**

**printf("Empty List \n");**

**}**

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

**head = NULL;**

**free(head);**

**printf("\n node deleted \n");**

**}**

**else {**

**ptr = head;**

**while(ptr->next!=NULL){**

**ptr = ptr->next;**

**}**

**ptr->prev->next = NULL;**

**free(ptr);**

**printf("\n node deleted \n");**

**}**

**}**

**void delete() {**

**int key;**

**printf("Enter the key where the node to be deleted : ");**

**scanf("%d",&key);**

**if(head->data == key){**

**delete\_beginning();**

**}**

**else {**

**ptr=head;**

**while(ptr->data != key && ptr->next != NULL){**

**ptr = ptr->next;**

**}**

**if(ptr->data != key) {**

**printf("No data found \n");**

**}**

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

**ptr->prev->next = NULL;**

**free(ptr);**

**printf("NODE DELETED \n");**

**}**

**else {**

**ptr->prev->next=ptr->next;**

**ptr->next->prev=ptr->prev;**

**free(ptr);**

**printf("NODE DELETED \n");**

**}**

**}**

**}**

**void display() {**

**int c=0;**

**if(head == NULL) {**

**printf("Empty List \n");**

**} else {**

**printf("\n----------------------\n");**

**ptr = head;**

**while(ptr!=NULL){**

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

**c++;**

**ptr = ptr->next;**

**}**

**printf("\n----------------------\n");**

**printf("Number nodes: %d \n \n",c);**

**}**

**}**

**void r\_display(){**

**if(head == NULL) {**

**printf("Empty List \n");**

**} else {**

**printf("\n----------------------\n");**

**ptr = head;**

**while(ptr->next!=NULL){**

**ptr = ptr->next;**

**}**

**last =ptr;**

**while(last!=NULL){**

**printf("%d ",last->data);**

**last =last->prev;**

**}**

**printf("\n----------------------\n");**

**}**

**}**

**void search()**

**{**

**struct node \*ptr;**

**int item,i=0,flag;**

**ptr = head;**

**if(ptr == NULL)**

**{**

**printf("\nEmpty List\n");**

**}**

**else**

**{**

**printf("\nEnter item which you want to search?\n");**

**scanf("%d",&item);**

**while (ptr!=NULL)**

**{**

**if(ptr->data == item)**

**{**

**printf("\nitem found at location %d \n \n ",i+1);**

**flag=0;**

**break;**

**}**

**else**

**{**

**flag=1;**

**}**

**i++;**

**ptr = ptr -> next;**

**}**

**if(flag==1)**

**{**

**printf("\nItem not found\n");**

**}**

**}**

**}**