**DSA LAB**

**Lab Assignment number 12**

**Name:** Aamir Ansari **Batch:** A **Roll no:** 01

**Aim:** Implementation of Circular Doubly linked list

// code

#include <stdio.h>

#include <stdlib.h>

struct node {

int data;

struct node \*previous;

struct node \*next;

};

struct node \*start = NULL;

int countNodes() {

if (start == NULL) {

printf("\nLIST IS EMPTY!");

return 0;

}

struct node \*ptr = start;

int count = 1;

while (ptr->next != start) {

ptr = ptr->next;

count++;

}

return count;

}

void insertAtBeginning(int toInsert) { // Inserts node at the beginning

struct node \*newNode;

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

newNode->data = toInsert;

if (start == NULL) { // first node of the list is added

newNode->next = newNode;

newNode->previous = newNode;

start = newNode;

} else {

// linking last node with new node

newNode->previous = start->previous;

start->previous->next = newNode;

// linking new node with current first node

newNode->next = start;

start->previous = newNode;

start = newNode;

}

}

void insertAtEnd(int toInsert) { // Inserts node at the end of the list

struct node \*newNode;

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

newNode->data = toInsert;

if (start == NULL) { // first node of the list is added

newNode->next = newNode;

newNode->previous = newNode;

start = newNode;

} else {

// linking newNode with current last node

newNode->previous = start->previous;

start->previous->next = newNode;

// linking newNode with start node

newNode->next = start;

start->previous = newNode;

}

}

void insertBeforeVal(int toInsert, int val) { // Inserts before val

if (start == NULL) {

printf("\nLIST IS EMPTY!");

return;

}

struct node \*newNode;

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

newNode->data = toInsert;

struct node \*ptr = start;

if (ptr->data == val) { // inserting before current first node

insertAtBeginning(toInsert);

} else {

while (ptr->next->data != val) { // traversing

ptr = ptr->next;

}

// linking newNode with ptr->next

newNode->next = ptr->next;

ptr->next->previous = newNode;

// linking newNode with ptr

newNode->previous = ptr;

ptr->next = newNode;

}

}

void insertAfterVal(int toInsert, int val) { // Inserts after val

if (start == NULL) {

printf("\nLIST IS EMPTY!");

return;

}

struct node \*newNode;

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

newNode->data = toInsert;

struct node \*ptr = start;

while (ptr->data != val) {

ptr = ptr->next;

}

if (ptr == start->previous) { // inserting after current last node

insertAtEnd(toInsert);

} else {

// linking newNode with ptr->next

newNode->next = ptr->next;

ptr->next->previous = newNode;

// linking newNode with ptr

newNode->previous = ptr;

ptr->next = newNode;

}

}

void insertAtPosition(int toInsert, int pos) {

if (start == NULL) {

printf("\nLIST IS EMPTY!");

return;

}

struct node \*newNode;

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

newNode->data = toInsert;

struct node \*ptr = start;

int count = 1;

while (count != pos && ptr->next != start) { // traversing

ptr = ptr->next;

count++;

}

if (pos > count+1) { // invalid position

printf("\nList is not that long!");

return;

}

if (count == 1) { // adding new node before first node

insertAtBeginning(toInsert);

} else if (ptr->next == start && count < pos) { // inserting after last node /\* second condition => when the postion is second-last \*/

insertAtEnd(toInsert);

} else { // inserting at any position

insertBeforeVal(toInsert, ptr->data);

}

}

void deleteAtBeginning() { // Deletes elements at the beginning

if (start == NULL) {

printf("\nLIST IS EMPTY!");

return;

}

struct node \*ptr = start;

printf("\nDeleted element is : %d", ptr->data);

if (start->next == start) { // deleting only remaining node

free(start);

start = NULL;

} else {

// linking current second node with the last node

start->next->previous = start->previous;

start->previous->next = start->next;

// shifting start

start = start->next;

// freeing first node

free(ptr);

}

}

void deleteAtEnd() { // deletes element at the end

if (start == NULL) {

printf("\nLIST IS EMPTY!");

return;

}

struct node \*ptr = start;

printf("\nDeleted element is : %d", ptr->previous->data);

if (start->next == start) { // deleting only remaining node

free(start);

start = NULL;

} else {

// shifting ptr to last node

ptr = ptr->previous;

// linking current second last node to start

ptr->previous->next = start;

start->previous = ptr->previous;

// freeing last node

free(ptr);

}

}

void deleteBeforeVal(int val) { // Deletes before val

if (start == NULL) {

printf("\nLIST IS EMPTY!");

return;

}

struct node \*ptr = start;

if (ptr->data == val) { // deleting before first node

deleteAtEnd();

} else {

while (ptr->next->data != val) { // traversing

ptr = ptr->next;

}

printf("\nDeleted element is : %d", ptr->data);

// linking nodes which are before and after ptr

ptr->previous->next = ptr->next;

ptr->next->previous = ptr->previous;

// freeing ptr

free(ptr);

}

}

void deleteAfterVal(int val) { // deletes after val

if (start == NULL) {

printf("\nLIST IS EMPTY!");

return;

}

struct node \*ptr = start;

while (ptr->data != val) {

ptr = ptr->next;

}

if (ptr->next == start) { // deleting after last node

deleteAtBeginning();

} else if (ptr->next->next == start) { // deleting last node

deleteAtEnd();

} else {

// shifting ptr to node which is to be deleted

ptr = ptr->next;

deleteBeforeVal(ptr->next->data);

}

}

void deleteAtPosition(int pos) { // deletes at position

if (start == NULL) {

printf("\nLIST IS EMPTY!");

return;

}

struct node \*ptr = start;

int count = 1;

while (count!=pos && ptr->next!=start) {

ptr = ptr->next;

count++;

}

if (pos > count) {

printf("\nINVALID POSITION!");

return;

}

if (count == 1) { // deleting first node

deleteAtBeginning();

} else if (ptr->next == start) { // deleting last node

deleteAtEnd();

} else {

deleteAfterVal(ptr->previous->data);

}

if (ptr->next == start && ptr->previous == start) {

start = NULL;

}

}

void updateAtBeginning(int toUpdate) { // updates at the beginning

if (start == NULL) {

printf("\nLIST IS EMPTY!");

return;

}

// updation

start->data = toUpdate;

}

void updateAtEnd(int toUpdate) { // update at the end

if (start == NULL) {

printf("\nLIST IS EMPTY!");

return;

}

// updation

start->previous->data = toUpdate;

}

void updateBeforeVal(int toUpdate, int val) { // updates before val

if (start == NULL) {

printf("\nLIST IS EMPTY!");

return;

}

struct node \*ptr = start;

if (ptr->data == val) { // updating before first node

start->previous->data = toUpdate;

} else {

while (ptr->next->data != val) { // traversing

ptr = ptr->next;

}

// updation

ptr->data = toUpdate;

}

}

void updateAfterVal(int toUpdate, int val) { // update after val is encountered

if (start == NULL) {

printf("\nLIST IS EMPTY!");

return;

}

struct node \*ptr = start;

while (ptr->data != val) {

ptr = ptr->next;

}

ptr = ptr->next;

ptr->data = toUpdate;

}

void updateAtPosition(int toUpdate, int pos) { // updates at posiotion

if (start == NULL) {

printf("\nLIST IS EMPTY!");

return;

}

struct node \*ptr = start;

int count = 1;

while (count!=pos && ptr->next != start) {

ptr = ptr->next;

count++;

}

if (pos>count) {

printf("\nINVALID POSITION!");

return;

}

ptr->data = toUpdate;

}

void search(int val) {

if (start == NULL) {

printf("\nLIST IS EMPTY!");

return;

}

struct node \*ptr = start;

int count = 1;

while (ptr->data != val && count<=countNodes()+1) {

ptr = ptr->next;

count++;

}

// printing

if (count > countNodes()) {

printf("\n%d is not present in the list!", val);

} else {

printf("\nPosition of %d in the list is : %d", val, count);

}

}

void display() {

if (start == NULL) { // check if list is empty

printf("\nList is empty!");

return;

}

// traversing pointer

struct node \*ptr = start;

printf("FORWARD : ");

while (ptr->next != start) {

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

ptr = ptr->next;

}

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

printf("\nREVERSE : ");

while (ptr->previous != start->previous) {

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

ptr = ptr->previous;

}

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

}

int main() {

int choice, toInsert, toUpdate, val, pos;

while (1) {

printf("\n\*1 INSERT At END ");

printf("\n\*2 INSERT At BEGINING ");

printf("\n\*3 INSERT BEFORE VAL ");

printf("\n\*4 INSERT AFTER VAL ");

printf("\n\*5 INSERT At POSITION ");

printf("\n\*6 DELETE At END ");

printf("\n\*7 DELETE At BEGINING ");

printf("\n\*8 DELETE BEFORE VAL ");

printf("\n\*9 DELETE AFTER VAL ");

printf("\n\*10 DELETE At POSITION ");

printf("\n\*11 UPDATE At END ");

printf("\n\*12 UPDATE At BEGINING ");

printf("\n\*13 UPDATE BEFORE VAL ");

printf("\n\*14 UPDATE AFTER VAL ");

printf("\n\*15 UPDATE At POSITION ");

printf("\n\*16 SEARCH in the list ");

printf("\n\*17 COUNT NODE in the list ");

printf("\n\*18 DISPLAY elements of the list ");

printf("\n\*19 EXIT ");

printf("\nEnter your choice : ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("\nEnter element to insert : ");

scanf("%d", &toInsert);

insertAtEnd(toInsert);

break;

case 2:

printf("\nEnter element to insert : ");

scanf("%d", &toInsert);

insertAtBeginning(toInsert);

break;

case 3:

printf("\nEnter element to insert : ");

scanf("%d", &toInsert);

printf("\nEnter value BEFORE which to insert : ");

scanf("%d", &val);

insertBeforeVal(toInsert, val);

break;

case 4:

printf("\nEnter element to insert : ");

scanf("%d", &toInsert);

printf("\nEnter value AFTER which to insert : ");

scanf("%d", &val);

insertAfterVal(toInsert, val);

break;

case 5:

printf("\nEnter element to insert : ");

scanf("%d", &toInsert);

printf("\nEnter POSITION AT which to insert : ");

scanf("%d", &pos);

insertAtPosition(toInsert, pos);

break;

case 6:

deleteAtEnd();

break;

case 7:

deleteAtBeginning();

break;

case 8:

printf("\nEnter value BEFORE which to DELETE : ");

scanf("%d", &val);

deleteBeforeVal(val);

break;

case 9:

printf("\nEnter value AFTER which to DELETE : ");

scanf("%d", &val);

deleteAfterVal(val);

break;

case 10:

printf("\nEnter POSITION AT which to DELETE : ");

scanf("%d", &pos);

deleteAtPosition(pos);

break;

case 11:

printf("\nEnter element to UPDATE : ");

scanf("%d", &toUpdate);

updateAtEnd(toUpdate);

break;

case 12:

printf("\nEnter element to UPDATE : ");

scanf("%d", &toUpdate);

updateAtBeginning(toUpdate);

break;

case 13:

printf("\nEnter element to UPDATE : ");

scanf("%d", &toUpdate);

printf("\nEnter value BEFORE which to UPDATE : ");

scanf("%d", &val);

updateBeforeVal(toUpdate, val);

break;

case 14:

printf("\nEnter element to UPDATE : ");

scanf("%d", &toUpdate);

printf("\nEnter value AFTER which to UPDATE : ");

scanf("%d", &val);

updateBeforeVal(toUpdate, val);

break;

case 15:

printf("\nEnter element to UPDATE : ");

scanf("%d", &toUpdate);

printf("\nEnter POSITION AT which to UPDATE : ");

scanf("%d", &pos);

updateAtPosition(toUpdate, pos);

break;

case 16:

printf("\nEnter a value to SEARCH : ");

scanf("%d", &val);

search(val);

break;

case 17:

printf("\nList contains %d elements", countNodes());

break;

case 18:

printf("\nElements in the list are : ");

display();

break;

case 19:

printf("\*\*\* E X I T I N G \*\*\*");

exit(1);

break;

default:

printf("INVALID INPUT");

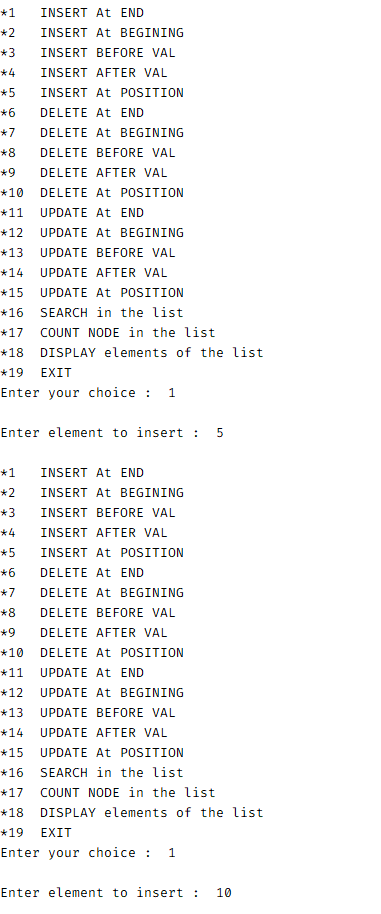
}

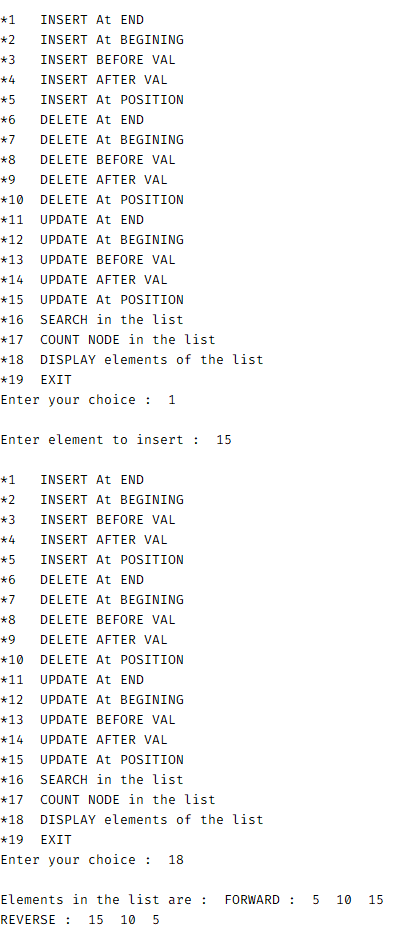
}

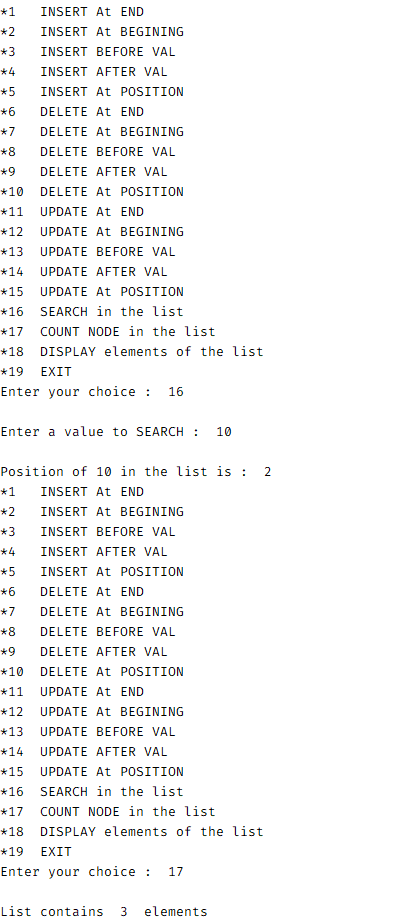
return 0;

}

// output

.

.

.

