**Implementation of circular linked list**

// code

#include <stdio.h>

#include <stdlib.h>

// Implementation of circular linked list

// declaration of node of linked list

struct node {

int data;

struct node \*next;

};

// declaration of end

struct node \*end = NULL;

void insertAtBegining(int toInsert) { // Insert at the begining of list

// declaring, inserting value and allocating memory for new node

struct node \*newNode;

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

newNode->data = toInsert;

if (end == NULL) { // if first node is to be added

end = newNode;

newNode->next = end;

} else { // inserting node at the begining

newNode->next = end->next;

end->next = newNode;

}

}

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

// declaring, inserting value and allocating memory for new node

struct node \*newNode;

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

newNode->data = toInsert;

if (end == NULL) { // if first node is to be added

end = newNode;

newNode->next = end;

} else { // inserting node at the end

// traversing pointer

struct node \*ptr;

ptr = end->next;

while (ptr->next != end) {

ptr = ptr->next;

}

newNode->next = end->next;

end->next = newNode;

end = newNode;

}

}

void insertBeforeVal(int toInsert, int val) { // Insert before valule (val) is encountered

// traversing pointer

struct node \*ptr;

ptr = end->next;

struct node \*prePtr;

prePtr = ptr;

// declaring, inserting value and allocating memory for new node

struct node \*newNode;

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

newNode->data = toInsert;

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

printf("\nList is empty!");

return;

}

// traversing upto val in the list

while (ptr->data != val) {

prePtr = ptr;

ptr = ptr->next;

}

if (ptr == end->next) { // adding before first node

newNode->next = end->next;

end->next = newNode;

} else { // adding before any nodes

prePtr->next = newNode;

newNode->next = ptr;

}

}

void insertAfterVal(int toInsert, int val) { // Inserts node after value (val) is encountered

// traversing pointer

struct node \*ptr;

ptr = end->next;

struct node \*prePtr;

prePtr = ptr;

// declaring, inserting value and allocating memory for new node

struct node \*newNode;

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

newNode->data = toInsert;

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

printf("\nList is empty!");

return;

}

// traversing until val is encountered

while (ptr->data != val) {

prePtr = ptr;

ptr = ptr->next;

}

prePtr = ptr;

ptr = ptr->next;

if (prePtr->next == end->next) { // inserting node after last node

newNode->next = end->next;

prePtr->next = newNode;

end = newNode;

} else { // inserting after any node

prePtr->next = newNode;

newNode->next = ptr;

}

}

void insertAtPosition(int toInsert, int pos) { // inserting after given position

// traversing pointer

struct node \*ptr;

ptr = end->next;

struct node \*prePtr;

prePtr = ptr;

// declaring, inserting value and allocating memory for new node

struct node \*newNode;

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

newNode->data = toInsert;

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

printf("\nList is empty!");

return;

}

int count = 1;

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

prePtr = ptr;

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

newNode->next = ptr;

end->next = newNode;

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

newNode->next = end->next;

end->next = newNode;

end = newNode;

} else { // inserting at any position

prePtr->next = newNode;

newNode->next = ptr;

}

}

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

if (end == NULL) { // checks if list is empty

printf("\nList is empty!");

return;

}

// traversing pointer

struct node \*ptr = end->next;

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

// shiftind end->next to second node

end->next = ptr->next;

if (ptr == end) { // when only remaining node is deleted

end = NULL;

}

free(ptr);

}

void deleteAtEnd() { // Deletes node at the end of the linked list

if (end == NULL) { // checks if list is empty

printf("\nList is empty!");

return;

}

// traversing pointer

struct node \*ptr = end->next;

struct node \*prePtr = ptr;

while (ptr->next != end->next) {

prePtr = ptr;

ptr = ptr->next;

}

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

// shifting end to second-last node

prePtr->next = end->next;

end = prePtr;

if (prePtr == ptr) { // when only remaining node is deleted

end = NULL;

}

free(ptr);

}

void deleteBeforeVal(int val) { // Deletes node before given value (val)

if (end == NULL) { // checks if list is empty

printf("\nList is empty!");

return;

}

// traversing pointer

struct node \*ptr = end->next;

struct node \*prePtr = ptr;

if (ptr->data == val) { // if the val is of first node

printf("\nThere is no node before this!");

return;

}

// traversing

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

prePtr = ptr;

ptr = ptr->next;

}

// deleting

prePtr->next = ptr->next;

free(ptr);

}

void deleteAfterVal(int val) { // Deletes node after value (val) is encountered

if (end == NULL) { // checks if list is empty

printf("\nList is empty!");

return;

}

// traversing pointer

struct node \*ptr = end->next;

struct node \*prePtr = ptr;

// traversing the list

while (ptr->data != val) {

prePtr = ptr;

ptr = ptr->next;

}

prePtr = ptr;

ptr = ptr->next;

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

if (ptr->next == end->next) { // last node is deleted

prePtr->next = end->next;

end = prePtr;

free(ptr);

} else { // andy other node is deleted

prePtr->next = ptr->next;

free(ptr);

}

}

void deleteAtPosition(int pos) { // Deletes node at given position

if (end == NULL) { // checks if list is empty

printf("\nList is empty!");

return;

}

// traversing pointer

struct node \*ptr = end->next;

struct node \*prePtr = ptr;

int count = 1;

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

prePtr = ptr;

ptr = ptr->next;

count++;

}

if (pos > count) { // invalid pos

printf("\nThere is no node at this position");

return;

}

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

if (end->next == ptr) { // deleting at first position

end->next = ptr->next;

free(ptr);

} else if (ptr->next == end->next) { // deleting at last position

prePtr->next = end->next;

end = prePtr;

end->next = prePtr->next;

free(ptr);

} else { // deleting at any position

prePtr->next = ptr->next;

free(ptr);

}

if (ptr->next == end->next) { // only remaining node is deleted

end = NULL;

}

}

void updateAtBeginning(int toUpdate) { // Updates the element at the beginning

if (end == NULL) { // check if the list is empty

printf("\nList is Empty!");

return;

}

// updation

end->next->data = toUpdate;

}

void updateAtEnd(int toUpdate) { // Updates the element at the end

if (end == NULL) { // check if the list is empty

printf("\nList is Empty!");

return;

}

// updation

end->data = toUpdate;

}

void updateBeforeVal(int toUpdate, int val) { // Updates element before a given val

if (end == NULL) { // check if the list is empty

printf("\nList is Empty!");

return;

}

if (end->next->data == val) { // if the value is of first node

printf("\nThere are no elements before this!");

return;

}

// traversing pointer

struct node \*ptr = end->next;

// traversing

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

ptr = ptr->next;

}

// updation

ptr->data = toUpdate;

}

void updateAfterVal(int toUpdate, int val) { // Updates the element after value (val) is encountered

if (end == NULL) { // check if the list is empty

printf("\nList is Empty!");

return;

}

// traversing pointer

struct node \*ptr = end->next;

// traversing

while (ptr->data != val) {

ptr = ptr->next;

}

// updation

ptr->next->data = toUpdate;

}

void updateAtPosition(int toUpdate, int pos) { // Updates value at given position

if (end == NULL) { // check if the list is empty

printf("\nList is Empty!");

return;

}

// traversing pointer

struct node \*ptr = end->next;

int count = 1;

// traversing

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

ptr = ptr->next;

count++;

}

if (pos > count) { // checks for valid position

printf("\nNo node at the given position!");

return;

}

// updation

ptr->data = toUpdate;

}

int countNodes() { // Counts number of nodes in the list

if (end == NULL) { // if the list is empty

return 0;

}

// traversing pointer

struct node \*ptr = end->next;

int count = 1;

// traversing

while (ptr->next!=end->next) {

ptr = ptr->next;

count++;

}

return count;

}

void search(int val) { // Search weather the val is present in the list and prints its position

if (end == NULL) { // check if the list is empty

printf("\nList is Empty!");

return;

}

// traversing pointer

struct node \*ptr = end->next;

int count = 1;

// traversing

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() { // Displays content of linked list

// traversing pointer

struct node \*ptr;

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

printf("\nList is empty!");

return;

}

// initializing traversing pointer

ptr = end->next;

// printing

while (ptr->next != end->next) {

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

ptr = ptr->next;

}

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);

insertAtBegining(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");

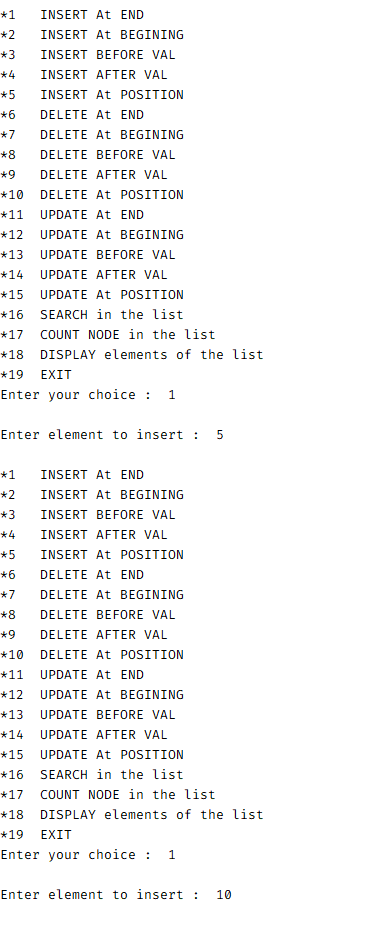
}

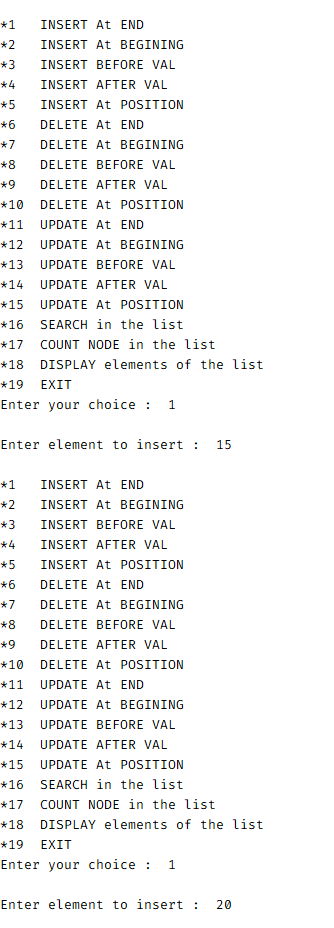
}

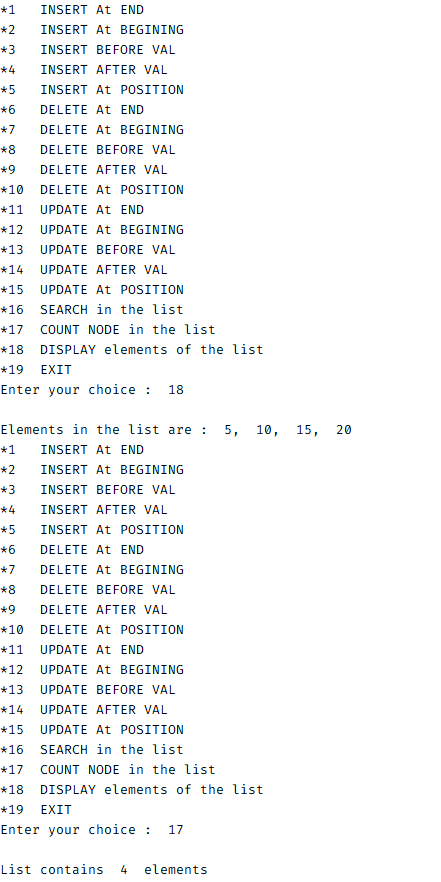
return 0;

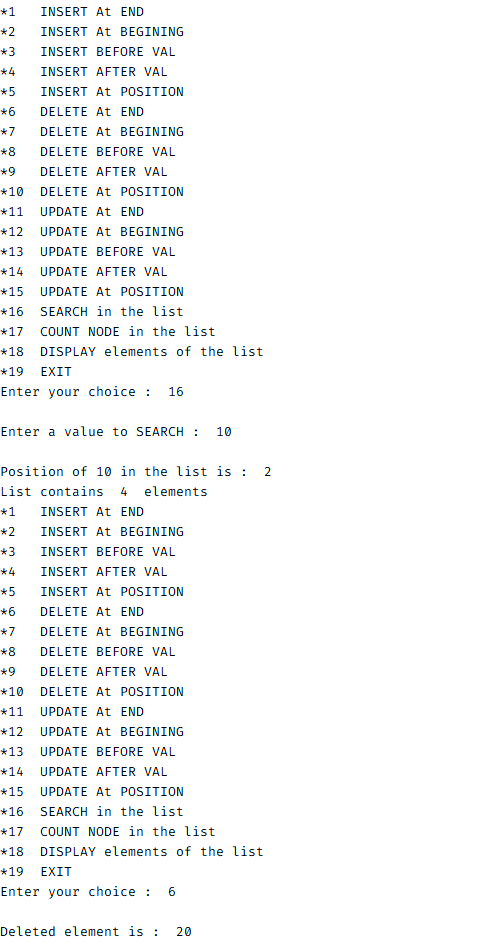
}

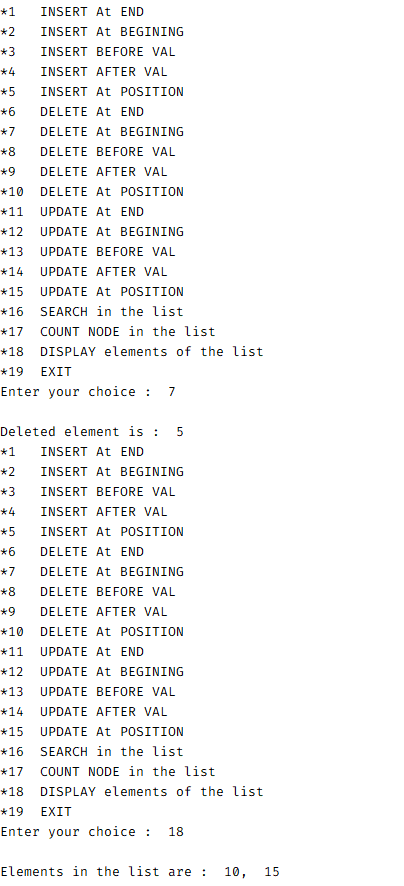
// output

.

.

.

.

.

