

Linked list

```
#ifndef _SLL_H_
#define _SLL_H_
#include <stdio.h>
#include <stdlib.h>
/* declaration of node in Singly linked list */
```

```
typedef struct node{
    int value;
    struct node * next;
}NODE;
```

```
/* declaration of head (pointer to node) */
extern NODE * head;
```

First header file "sll.h":

```
/* declaration for functions working on sll */
/*creating a list, freeing previously used memory */
void create_list(NODE ** head);
```

```
/* adding node at the end of sll */
int add_at_end(NODE **another, int val);
```

```
/*printing all nodes in list one after another */
void print_list(NODE * another);
```

```
/* counting nodes in list */
int count_nodes(NODE * another);
```

```
/* locating value in node in sll */
NODE * locate(NODE * another, int val);
```

```
/* adding node in growing order of valued */
int add_at_order(NODE **another, int val);
```

```
/* remove node with given value */
void remove_node(NODE **another, int val);
```

```
#endif
```

"singly.c" containing all functions definitions used by the list:
#include "sll.h"

```
/* returns 1 if node is created, 0 if not */
```

```
int add_at_end(NODE **another, int val)
{
    while(*another!=NULL)
        another=&(*another)->next;
    *another=malloc(sizeof(NODE));
    if(*another==NULL)
        return 0;
    (*another)->value=val;
    (*another)->next=NULL;
    return 1;
}
```

```
/* adds not as list was ordered */
```

```
int add_at_order(NODE ** another, int val)
{
    NODE * new;
    while(*another!=NULL && (*another)->value<val)
        another=&(*another)->next;
    new=malloc(sizeof(NODE));
    if(new==NULL)
        return 0;
    new->value=val;
    new->next=(*another);
    *another=new;
}
```

```
/* counts nodes in the list */
```

```
int count_nodes(NODE * another)
{
    int count=0;
    while(another!=NULL)
```

```

    {
        count++;
        another=another->next;
    }

    return count;
}

/* sets head of list to NULL, frees memory if there was previous list */
void create_sll(NODE **node)
{
    NODE * prev;
    NODE * nxt;
    nxt=*node;
    while(nxt!=NULL)
    {
        prev=nxt;
        nxt=prev->next;
        printf("Removing node with val: %d\n\n", prev->value);
        free(prev);
    }
    *node=NULL;
}

/* checks if nodes is in the list, returns pointer to the node */
NODE * locate(NODE * another, int val)
{
    while(another!=NULL)
    {
        if(another->value==val)
            return another;
        another=another->next;
    }

    return another;
}

/* prints out all nodes in the list */
void print_list(NODE * another)
{

```

```

if(another!=NULL)
{
    while(another!=NULL)
    {
        printf("node val: %d\n", another->value);
        another=another->next;
    }
    putchar('\n');
}
else
    printf("List is currently empty.\n\n");
}

/* removes node from list */
void remove_node(NODE **another, int val)
{
    NODE *temp;
    while(*another!=NULL && (*another)->value!=val)
        another=&(*another)->next;
    if(*another==NULL)
        printf("Node not found or empty list.\n\n");
    else
    {
        temp=*another;
        *another=(*another)->next;
        printf("Removing node with value : %d\n\n", temp->value);
        free(temp);
    }
}

```

.....

```

#include "sll.h"

```

```

void print_menu(void); /* this will show menu */
int get_menu(void); /* this will get answer from user */
void clear_input(void); /* clears input buffer */

```

```

int main(void)
{
    NODE * head=NULL;
    int option;
    int node_val;

    while((option=get_menu())!='8')
    {
        switch (option)
        {
            case '1':
                printf("Creating new list, removing previous list, if existed\n\n");
                create_sll(&head);
                break;
            case '2':
                printf("Input integer value for node: ");
                scanf("%d", &node_val);
                add_at_order(&head, node_val);
                clear_input();
                break;
            case '3':
                printf("Input integer value for node: ");
                scanf("%d", &node_val);
                add_at_end(&head, node_val);
                clear_input();
                break;
            case '4':
                printf("Input integer value for node to be removed: ");
                scanf("%d", &node_val);
                remove_node(&head, node_val);
                clear_input();
                break;
            case '5':
                print_list(head);
                break;
            case '6':
                printf("Number of nodes in the list: %d\n\n",count_nodes(head));
                break;
            case '7':
                printf("Input integer value for node to be located in the list: ");

```

```

scanf("%d", &node_val);
if(locate(head, node_val))
    printf("Node found\n\n");
else
    printf("Node not in the list.\n\n");
clear_input();
break;
default: printf("I should never get in here!\n"); break;
}
}

return EXIT_SUCCESS;
}

void clear_input(void)
{
    while(getchar()!='\n')
        continue;
}

void print_menu(void)
{
    printf("*****\n");
    printf("* 1) create new list      *\n");
    printf("* 2) add ordered node      *\n");
    printf("* 3) add unordered node    *\n");
    printf("* 4) remove node          *\n");
    printf("* 5) print all nodes       *\n");
    printf("* 6) count all nodes       *\n");
    printf("* 7) locate node          *\n");
    printf("* 8) quit                  *\n");
    printf("*****\n");
}

int get_menu(void)
{
    int answer;
    while(print_menu(), (answer=getchar())!=EOF && (answer>'8' || answer<'1'))
    {
        clear_input();
    }
}

```

```

    printf("I didn't understand, select option 1-8\n\n");
}
clear_input();

return answer;
}

```

How linked lists are arranged in memory?

Linked list basically consists of memory blocks that are located at random memory locations. Now, one would ask how are they connected or how they can be traversed? Well, they are connected through pointers. Usually a block in a linked list is represented through a structure like this :

```

struct test_struct
{
    int val;

    struct test_struct *next;
};

```

So as you can see here, this structure contains a value 'val' and a pointer to a structure of same type. The value 'val' can be any value (depending upon the data that the linked list is holding) while the pointer 'next' contains the address of next block of this linked list. So linked list traversal is made possible through these 'next' pointers that contain address of the next node. The 'next' pointer of the last node (or for a single node linked list) would contain a NULL.

___*****

How a node is created?

A node is created by allocating memory to a structure (as shown in above point) in the following way :

```
struct test_struct *ptr = (struct test_struct*)malloc(sizeof(struct test_struct));
```

.....

So, as we can see above, the pointer 'ptr' now contains address of a newly created node. If the linked list is empty and first node is created then it is also known as head node.

Once a node is created, then it can be assigned the value (that it is created to hold) and its next pointer is assigned the address of next node. If no next node exists (or if its the last node) then as already discussed, a NULL is assigned. This can be done in following way :

...

...

```
ptr->val = val;
```

```
ptr->next = NULL;
```

...

...

How to search a node in a linked list?

Searching a node means finding the node that contains the value being searched. This is in fact a very simple task if we talk about linear search (Note that there can be many search algorithms). One just needs to start with the first node and then compare the value which is being searched with the value contained in this node. If the value does not match then through the 'next' pointer (which contains the address of next node) the next node is accessed and same value comparison is done there. The search goes on until last node is accessed or node is found whose value is equal to the value being searched. A code snippet for this may look like :

...

...

...

```
while(ptr != NULL)
```



```

{
    if(ptr->val == val)
    {
        found = true;
        break;
    }
    else
    {
        ptr = ptr->next;
    }
}
...
...
...
.....

```

How a node is deleted?

A node is deleted by first finding it in the linked list and then calling free() on the pointer containing its address. If the deleted node is any node other than the first and last node then the 'next' pointer of the node previous to the deleted node needs to be pointed to the address of the node that is just after the deleted node. Its just like if a person breaks away from a human chain then the two persons (between whom the person was) needs to join hand together to maintain the chain.

A Practical C Linked List Example

Here is a practical example that creates a linked list, adds some nodes to it, searches and deletes nodes from it.

```
#include<stdio.h>
```

```
#include<stdlib.h>
```

```
#include<stdbool.h>
```

```
struct test_struct
```

```
{
```

```
    int val;
```

```
    struct test_struct *next;
```

```
};
```

```
struct test_struct *head = NULL;
```

```
struct test_struct *curr = NULL;
```

```
struct test_struct* create_list(int val)
```

```
{
```

```
    printf("\n creating list with headnode as [%d]\n",val);
```

```
    struct test_struct *ptr = (struct test_struct*)malloc(sizeof(struct test_struct));
```

```
    if(NULL == ptr)
```

```
    {
```

```
        printf("\n Node creation failed \n");
```

```
        return NULL;
```

```
    }
```

```
    ptr->val = val;
```

```
    ptr->next = NULL;
```

```
    head = curr = ptr;
```

```

    return ptr;
}

struct test_struct* add_to_list(int val, bool add_to_end)
{
    if(NULL == head)
    {
        return (create_list(val));
    }

    if(add_to_end)
        printf("\n Adding node to end of list with value [%d]\n",val);
    else
        printf("\n Adding node to beginning of list with value [%d]\n",val);

    struct test_struct *ptr = (struct test_struct*)malloc(sizeof(struct test_struct));
    if(NULL == ptr)
    {
        printf("\n Node creation failed \n");
        return NULL;
    }
    ptr->val = val;
    ptr->next = NULL;

    if(add_to_end)

```

```

{
    curr->next = ptr;
    curr = ptr;
}
else
{
    ptr->next = head;
    head = ptr;
}
return ptr;
}

```

```

struct test_struct* search_in_list(int val, struct test_struct **prev)

```

```

{
    struct test_struct *ptr = head;
    struct test_struct *tmp = NULL;
    bool found = false;

    printf("\n Searching the list for value [%d] \n",val);

    while(ptr != NULL)
    {
        if(ptr->val == val)
        {
            found = true;

```

```
        break;
    }
    else
    {
        tmp = ptr;
        ptr = ptr->next;
    }
}
```

```
if(true == found)
{
    if(prev)
        *prev = tmp;
    return ptr;
}
else
{
    return NULL;
}
}
```

```
int delete_from_list(int val)
{
    struct test_struct *prev = NULL;
    struct test_struct *del = NULL;
```

```
printf("\n Deleting value [%d] from list\n",val);
```

```
del = search_in_list(val,&prev);
```

```
if(del == NULL)
```

```
{
```

```
    return -1;
```

```
}
```

```
else
```

```
{
```

```
    if(prev != NULL)
```

```
        prev->next = del->next;
```

```
    if(del == curr)
```

```
    {
```

```
        curr = prev;
```

```
    }
```

```
    else if(del == head)
```

```
    {
```

```
        head = del->next;
```

```
    }
```

```
}
```

```
free(del);
```

```
del = NULL;
```

```

    return 0;
}

void print_list(void)
{
    struct test_struct *ptr = head;

    printf("\n -----Printing list Start----- \n");
    while(ptr != NULL)
    {
        printf("\n [%d] \n",ptr->val);
        ptr = ptr->next;
    }
    printf("\n -----Printing list End----- \n");

    return;
}

int main(void)
{
    int i = 0, ret = 0;
    struct test_struct *ptr = NULL;

    print_list();

```

```
for(i = 5; i<10; i++)  
    add_to_list(i,true);
```

```
print_list();
```

```
for(i = 4; i>0; i--)  
    add_to_list(i,false);
```

```
print_list();
```

```
for(i = 1; i<10; i += 4)  
{  
    ptr = search_in_list(i, NULL);  
    if(NULL == ptr)  
    {  
        printf("\n Search [val = %d] failed, no such element found\n",i);  
    }  
    else  
    {  
        printf("\n Search passed [val = %d]\n",ptr->val);  
    }  
}
```

```
print_list();
```



```

    ret = delete_from_list(i);
    if(ret != 0)
    {
        printf("\n delete [val = %d] failed, no such element found\n",i);
    }
    else
    {
        printf("\n delete [val = %d] passed \n",i);
    }

    print_list();
}

return 0;
}

```

In the code above :

The first node is always made accessible through a global ‘head’ pointer. This pointer is adjusted when first node is deleted.

Similarly there is a ‘curr’ pointer that contains the last node in the list. This is also adjusted when last node is deleted.

Whenever a node is added to linked list, it is always checked if the linked list is empty then add it as the first node.

he output of the above code looks like :

\$./11

-----Printing list Start-----

-----Printing list End-----

creating list with headnode as [5]

Adding node to end of list with value [6]

Adding node to end of list with value [7]

Adding node to end of list with value [8]

Adding node to end of list with value [9]

-----Printing list Start-----

[5]

[6]

[7]

[8]

[9]

-----Printing list End-----

Adding node to beginning of list with value [4]

Adding node to beginning of list with value [3]

Adding node to beginning of list with value [2]

Adding node to beginning of list with value [1]

-----Printing list Start-----

[1]

[2]

[3]

[4]

[5]

[6]

[7]

[8]

[9]

-----Printing list End-----

Searching the list for value [1]

Search passed [val = 1]

-----Printing list Start-----

[1]

[2]

[3]

[4]

[5]

[6]

[7]

[8]

[9]

-----Printing list End-----

Deleting value [1] from list

Searching the list for value [1]

delete [val = 1] passed

-----Printing list Start-----

[2]

[3]

[4]

[5]

[6]

[7]

[8]

[9]

-----Printing list End-----

Searching the list for value [5]

Search passed [val = 5]

-----Printing list Start-----

[2]

[3]

[4]

[5]

[6]

[7]

[8]

[9]

-----Printing list End-----

Deleting value [5] from list

Searching the list for value [5]

delete [val = 5] passed

-----Printing list Start-----

[2]

[3]

[4]

[6]

[7]

[8]

[9]

-----Printing list End-----

Searching the list for value [9]

Search passed [val = 9]

-----Printing list Start-----

[2]

[3]

[4]

[6]

[7]

[8]

[9]

-----Printing list End-----

Deleting value [9] from list

Searching the list for value [9]

delete [val = 9] passed

-----Printing list Start-----

[2]

[3]

[4]

[6]

[7]

[8]

-----Printing list End-----

Example of Linked List

Format:[data,address]

Head->[3,1000]->[43,1001]->[21,1002]

In the example, the number 43 is present at location 1000 and the address is present at in the previous node. This is how a linked list is represented.

Basic Linked List Functions

There are multiple functions that can be implemented on the linked list in C. Let's try to understand them with the help of an example program. First, we create a list, display it, insert at any location, delete a location. The following code will show you how to perform operations on the list.

```
#include<stdlib.h>
```

```
#include <stdio.h>
```

```
void create();
```

```
void display();
```

```
void insert_begin();
```

```
void insert_end();
```

```
void insert_pos();
```

```
void delete_begin();
```

```
void delete_end();
```

```
void delete_pos();
```

```

struct node
{
    int info;
    struct node *next;
};
struct node *start=NULL;
int main()
{
    int choice;
    while(1){

        printf("\n\t\t\t\t\t MENU\t\t\t\t\t n");
        printf("\n 1.Create\t\t n");
        printf("\n 2.Display\t\t n");
        printf("\n 3.Insert at the beginning\t\t n");
        printf("\n 4.Insert at the end\t\t n");
        printf("\n 5.Insert at specified position\t\t n");
        printf("\n 6.Delete from beginning\t\t n");
        printf("\n 7.Delete from the end\t\t n");
        printf("\n 8.Delete from specified position\t\t n");
        printf("\n 9.Exit\t\t n");
        printf("\n-----n");
        printf("Enter your choice:t");
        scanf("%d",&choice);
    }
}

```

```
switch(choice)
{
    case 1:
        create();
        break;
    case 2:
        display();
        break;
    case 3:
        insert_begin();
        break;
    case 4:
        insert_end();
        break;
    case 5:
        insert_pos();
        break;
    case 6:
        delete_begin();
        break;
    case 7:
        delete_end();
        break;
    case 8:
        delete_pos();
```

```

        break;

    case 9:

        exit(0);
        break;

    default:

        printf("n Wrong Choice:n");
        break;

    }

}

return 0;
}

void create()
{
    struct node *temp,*ptr;
    temp=(struct node *)malloc(sizeof(struct node));
    if(temp==NULL)
    {
        printf("nOut of Memory Space:n");
        exit(0);
    }
    printf("nEnter the data value for the node:t");
    scanf("%d",&temp->info);
    temp->next=NULL;

```

```
    if(start==NULL)
    {
        start=temp;
    }
    else
    {
        ptr=start;
        while(ptr->next!=NULL)
        {
            ptr=ptr->next;
        }
        ptr->next=temp;
    }
}

void display()
{
    struct node *ptr;
    if(start==NULL)
    {
        printf("\nList is empty:\n");
        return;
    }
    else
    {
        ptr=start;
```

```

        printf("\nThe List elements are:\n");
        while(ptr!=NULL)
        {
            printf("%d\t",ptr->info );
            ptr=ptr->next ;
        }
    }
}

void insert_begin()
{
    struct node *temp;
    temp=(struct node *)malloc(sizeof(struct node));
    if(temp==NULL)
    {
        printf("\nOut of Memory Space:\n");
        return;
    }
    printf("\nEnter the data value for the node:t" );
    scanf("%d",&temp->info);
    temp->next =NULL;
    if(start==NULL)
    {
        start=temp;
    }
    else

```

```

    {
        temp->next=start;
        start=temp;
    }
}

void insert_end()
{
    struct node *temp,*ptr;
    temp=(struct node *)malloc(sizeof(struct node));
    if(temp==NULL)
    {
        printf("\nOut of Memory Space:\n");
        return;
    }
    printf("\nEnter the data value for the node:t" );
    scanf("%d",&temp->info );
    temp->next =NULL;
    if(start==NULL)
    {
        start=temp;
    }
    else
    {
        ptr=start;
        while(ptr->next !=NULL)

```



```

        {
            ptr=ptr->next ;
        }
        ptr->next =temp;
    }
}

void insert_pos()
{
    struct node *ptr,*temp;
    int i,pos;
    temp=(struct node *)malloc(sizeof(struct node));
    if(temp==NULL)
    {
        printf("\nOut of Memory Space:\n");
        return;
    }
    printf("\nEnter the position for the new node to be inserted:t");
    scanf("%d",&pos);
    printf("\nEnter the data value of the node:t");
    scanf("%d",&temp->info) ;

    temp->next=NULL;
    if(pos==0)
    {
        temp->next=start;
    }
}

```

```

        start=temp;
    }
else
{
    for(i=0,ptr=start;i<pos-1;i++) { ptr=ptr->next;
        if(ptr==NULL)
        {
            printf("\nPosition not found:[Handle with care]\n");
            return;
        }
    }
    temp->next =ptr->next ;
    ptr->next=temp;
}
}

void delete_begin()
{
    struct node *ptr;
    if(ptr==NULL)
    {
        printf("\nList is Empty:\n");
        return;
    }
else
{

```

```

        ptr=start;
        start=start->next ;
        printf("\nThe deleted element is :%dt",ptr->info);
        free(ptr);
    }
}

void delete_end()
{
    struct node *temp,*ptr;
    if(start==NULL)
    {
        printf("\nList is Empty:");
        exit(0);
    }
    else if(start->next ==NULL)
    {
        ptr=start;
        start=NULL;
        printf("\nThe deleted element is:%dt",ptr->info);
        free(ptr);
    }
    else
    {
        ptr=start;
        while(ptr->next!=NULL)

```

```

        {
            temp=ptr;
            ptr=ptr->next;
        }
        temp->next=NULL;
        printf("\nThe deleted element is:%d",ptr->info);
        free(ptr);
    }
}

void delete_pos()
{
    int i,pos;
    struct node *temp,*ptr;
    if(start==NULL)
    {
        printf("\nThe List is Empty:\n");
        exit(0);
    }
    else
    {
        printf("\nEnter the position of the node to be deleted:t");
        scanf("%d",&pos);
        if(pos==0)
        {
            ptr=start;

```

```

        start=start->next ;
        printf("\nThe deleted element is:%dt",ptr->info );
        free(ptr);
    }
    else
    {
        ptr=start;
        for(i=0;i<pos;i++) { temp=ptr; ptr=ptr->next ;
            if(ptr==NULL)
            {
                printf("\nPosition not Found:n");
                return;
            }
        }
        temp->next =ptr->next ;
        printf("\nThe deleted element is:%dt",ptr->info );
        free(ptr);
    }
}
}

```

The first part of this code is creating a structure. A linked list structure is created so that it can hold the data and address as we need it. This is done to give the compiler an idea of how the node should be.

struct node

```
{
```

```
int info;  
struct node *next;  
};
```

In the structure, we have a data variable called info to hold data and a pointer variable to point at the address. There are various operations that can be done on a linked list, like:

```
create()  
display()  
insert_begin()  
insert_end()  
insert_pos()  
delete_begin()  
delete_end()  
delete_pos()
```

These functions are called by the menu-driven main function. In the main function, we take input from the user based on what operation the user wants to do in the program. The input is then sent to the switch case and based on user input.

Based on what input is provided the function will be called. Next, we have different functions that need to be solved. Let's take a look at each of these functions.

Create Function

First, there is a create function to create the linked list. This is the basic way the linked list is created. Lets us look at the code.

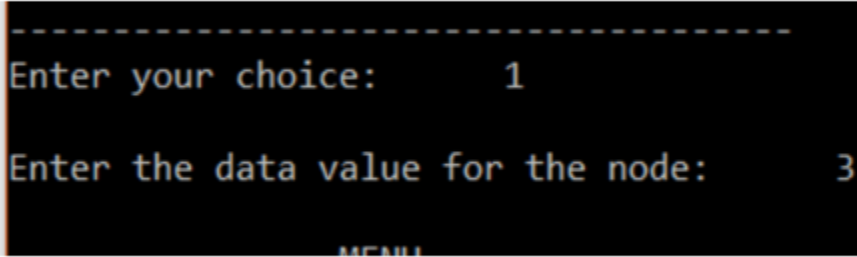
```
void create()  
{
```

```

struct node *temp,*ptr;

printf("\nEnter the data value for the node:t");
scanf("%d",&temp->info);
temp->next=NULL;
if(start==NULL)
{
    start=temp;
}
else
{
    ptr=start;
    while(ptr->next!=NULL)
    {
        ptr=ptr->next;
    }
    ptr->next=temp;
}
}

```



```

-----
Enter your choice:      1
Enter the data value for the node:      3
MENU

```

First, two pointers are created of the type node, ptr, and temp. We take the value that is needed to be added in the linked list from the user and store it in info part of

the temp variable and assign temp of next that is the address part to null. There is a start pointer holding the start of the list. Then we check for the start of the list. If the start of the list is null, then we assign temp to the start pointer. Otherwise, we traverse to the last point where the data has been added.

For this, we assign ptr the start value and traverse till ptr->next= null. We then assign ptr->next the address of temp. In a similar way, there is code given for inserting at the beginning, inserting at the end and inserting at a specified location.

Display Function

Here's the code for display function.

```
void display()
{
    struct node *ptr;
    if(start==NULL)
    {
        printf("\nList is empty:\n");
        return;
    }
    else
    {
        ptr=start;
        printf("\nThe List elements are:\n");
        while(ptr!=NULL)
        {
            printf("%d\t",ptr->info );
            ptr=ptr->next ;
        }
    }
}
```



```
    }  
}
```

Delete Function

Here's the snippet of code to delete a node from the linked list.

```
void delete_pos()  
{  
    int i,pos;  
    struct node *temp,*ptr;  
    if(start==NULL)  
    {  
        printf("\nThe List is Empty:\n");  
        exit(0);  
    }  
    else  
    {  
        printf("\nEnter the position of the node to be deleted:t");  
        scanf("%d",&pos);  
        if(pos==0)  
        {  
            ptr=start;  
            start=start->next ;  
            printf("\nThe deleted element is:%dt",ptr->info );  
            free(ptr);  
        }  
        else
```

```

    {
        ptr=start;
        for(i=0;i<pos;i++) { temp=ptr; ptr=ptr->next ;
            if(ptr==NULL)
            {
                printf("\nPosition not Found:\n");
                return;
            }
        }
        temp->next =ptr->next ;
        printf("\nThe deleted element is:%dt",ptr->info );
        free(ptr);
    }
}

```

In the deletion process, it first checks if the list is empty, if yes it exists. If it is not empty it asks the user for the position to be deleted. Once the user enters the position, it checks if it is the first position, if yes it assigns ptr to start and moves the start pointer to next location and deletes ptr. If the position is not zero, then it runs a for loop from 0 all the way to the pos entered by the user and stored in the pos variable. There is an if statement to decide if the position entered is not present. If ptr is equal to null, then it is not present.

We assign ptr to temp in the for loop, and ptr then moves on to the next part. After this when the position is found. We make the temp variable to hold the value of ptr->next thus skipping the ptr. Then ptr is deleted. Similarly, it can be done for first and the last element deletion

Doubly Linked List

It is called the doubly linked list because there are two pointers, one point to the next node and other points to the previous node. The operations performed in doubly linked are similar to that of a singly linked list. Here's the code for basic operations.

```
#include<stdio.h>

#include<stdlib.h>

struct Node;

typedef struct Node * PtrToNode;

typedef PtrToNode List;

typedef PtrToNode Position;


struct Node
{
    int e;
    Position previous;
    Position next;
};


void Insert(int x, List l, Position p)
{
    Position TmpCell;
    TmpCell = (struct Node*) malloc(sizeof(struct Node));
    if(TmpCell == NULL)
        printf("Memory out of spacen");
```

```

else
{
    TmpCell->e = x;
    TmpCell->previous = p;
    TmpCell->next = p->next;
    p->next = TmpCell;
}
}

```

```

void Delete(int x, List l)
{
    Position p, p1, p2;
    p = Find(x, l);
    if(p != NULL)
    {
        p1 = p -> previous;
        p2 = p -> next;
        p1 -> next = p -> next;
        if(p2 != NULL)          // if the node is not the last node
            p2 -> previous = p -> previous;
    }
    else
        printf("Element does not exist!!!\n");
}

```

```
}
```

```
void Display(List l)
```

```
{
```

```
    printf("The list element are :: ");
```

```
    Position p = l->next;
```

```
    while(p != NULL)
```

```
    {
```

```
        printf("%d -> ", p->e);
```

```
        p = p->next;
```

```
    }
```

```
}
```

```
int main()
```

```
{
```

```
    int x, pos, ch, i;
```

```
    List l, ll;
```

```
    l = (struct Node *) malloc(sizeof(struct Node));
```

```
    l->previous = NULL;
```

```
    l->next = NULL;
```

```
    List p = l;
```

```
    printf("DOUBLY LINKED LIST IMPLEMENTATION OF LIST ADT\n");
```

```
    do
```

```

{
    printf("\n1. CREATE\n 2. DELETE\n 3. DISPLAY\n 4. QUIT\n\nEnter the
choice :: ");
    scanf("%d", &ch);
    switch(ch)
    {
    case 1:
        p = l;
        printf("Enter the element to be inserted :: ");
        scanf("%d",&x);
        printf("Enter the position of the element :: ");
        scanf("%d",&pos);
        for(i = 1; i < pos; i++) { p = p->next;
        }
        Insert(x,l,p);
        break;

    case 2:
        p = l;
        printf("Enter the element to be deleted :: ");
        scanf("%d",&x);
        Delete(x,p);
        break;

```

```
    case 3:
        Display(1);
        break;
    }
}
while(ch<4);
}
```