LINKED LIST OPERATIONS

**MANIKANTA BUDATI**

**RA1911030010036**

I have done the program using ONLINE GDB compiler and got the desired outputs.

# **CODE :**

#include <stdio.h>

#include <stdlib.h>

struct node

{

int data;

struct node \*next;

};

int is\_empty(struct node \*head)

{

if(head==NULL)

return 1;

return 0;

}

void display(struct node \*head)

{

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

curr=head;

int i;

if(!is\_empty(head))

{

printf("THE LIST :");

for(i=0;curr!=NULL;i++)

{

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

curr=curr->next;

}

printf(" NULL.\n");

}

else

printf("THE LIST IS EMPTY..\n");

}

struct node \*insert\_front(struct node \*head,int dat)

{

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

temp->data=dat;

temp->next=head;

head=temp;

printf("ELEMENT IS INSERTED SUCCESSFULLY...\n");

int d;

printf("Press 1 to display the updated list.\n");

scanf("%d",&d);

if(d==1)

display(head);

return head;

}

void insert(struct node \*head,int dat,int ps)

{

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

temp->data=dat;

struct node \*cur=head;

int i;

for(i=1;i<ps-1;i++)

cur=cur->next;

temp->next=cur->next;

cur->next=temp;

printf("ELEMENT IS INSERTED SUCCESSFULLY...\n");

int d;

printf("Press 1 to display the updated list.\n");

scanf("%d",&d);

if(d==1)

display(head);

return;

}

struct node \*delete\_front(struct node \*head)

{

struct node \*temp=head;

head=head->next;

free(temp);

printf("ELEMENT IS DELETED SUCCESSFULLY...\n");

int d;

printf("Press 1 to display the updated list.\n");

scanf("%d",&d);

if(d==1)

display(head);

return head;

}

void delete\_pos(struct node \*head,int ps)

{

struct node \*cur=head;

int i;

for(i=1;i<ps-1;i++)

cur=cur->next;

struct node \*temp=cur->next;

cur->next=temp->next;

free(temp);

printf("ELEMENT IS DELETED SUCCESSFULLY...\n");

int d;

printf("Press 1 to display the updated list.\n");

scanf("%d",&d);

if(d==1)

display(head);

return;

}

void sort\_list(struct node \*he)

{

struct node \*current = he, \*tmp = NULL;

int temp;

while(current != NULL)

{

tmp = current->next;

while(tmp != NULL)

{

if(current->data > tmp->data)

{

temp = current->data;

current->data = tmp->data;

tmp->data = temp;

}

tmp =tmp->next;

}

current = current->next;

}

}

struct node \*merge\_lists(int x1,int x2)

{

int i;

struct node \*h1=NULL,\*h2=NULL,\*t1=NULL,\*t2=NULL;

printf("Enter the elements in the first list.\n");

for(i=0;i<x1;i++)

{

struct node \*p1=(struct node \*)malloc(sizeof(struct node));

p1->next=NULL;

scanf("%d",&p1->data);

if(h1==NULL)

{

h1=p1;

t1=p1;

}

else

{

t1->next=p1;

t1=p1;

}

}

printf("Enter the elements in the second list.\n");

for(i=0;i<x2;i++)

{

struct node \*p2=(struct node \*)malloc(sizeof(struct node));

p2->next=NULL;

scanf("%d",&p2->data);

if(h2==NULL)

{

h2=p2;

t2=p2;

}

else

{

t2->next=p2;

t2=p2;

}

}

t1->next=h2;

sort\_list(h1);

printf("THE TWO LISTS ARE MERGED SUCCESSFULLY...\n");

int d;

printf("Press 1 to display the updated list.\n");

scanf("%d",&d);

if(d==1)

display(h1);

return h1;

}

int main()

{

int i,x,op,ins,pos,ae,del,n=0,n1,n2;

struct node \*head=NULL;

do

{

printf("Choose the operation to perform.\n");

printf("Press 1 to check whether th list is empty.\n");

printf("Press 2 to insert a new element.\nPress 3 to delete an element.\n");

printf("Press 4 to display the list.\n");

printf("Press 5 to merge two linked lists.\nPress 6 to exit.\n");

scanf("%d",&op);

switch(op)

{

case 1:

{

int r=is\_empty(head);

if(r==1)

printf("THE LIST IS EMPTY.\n");

else

printf("THE LIST IS NOT EMPTY.\n");

break;

}

case 2:

{

printf("Press 1 to insert w.r.t position.\n");

printf("Press 2 to insert after a particular element.\n");

scanf("%d",&ins);

if(head==NULL && ins==2)

{

printf("NO ELEMENTS ARE PRESENT IN THE LIST YET.\n");

break;

}

else

{

if(ins==1)

{

n++;

printf("Enter the element and position.\n");

scanf("%d %d",&x,&pos);

if(pos>n+1 || pos<1)

{

printf("THE GIVEN POSITION IS INVALID..\n");

break;

}

else

{

if(pos==1)

{

head=insert\_front(head,x);

break;

}

else

{

insert(head,x,pos);

break;

}

}

}

else if(ins==2)

{

n++;

printf("Enter the element to be inserted and the element after which it should be inserted.\n");

scanf("%d %d",&x,&ae);

struct node \*c=head;

int check=0;

for(i=2;c!=NULL;i++)

{

if(c->data==ae)

{

check++;

break;

}

c=c->next;

}

if(check==0)

{printf("The given element does not present in the list..\n");

break;}

else

{

insert(head,x,i);

break;

}

}

else

{

printf("INVALID INPUT.!!\n");

break;

}

}

}

case 3:

{

if(head==NULL)

{

printf("NO ELEMENTS ARE PRESENT IN THE LIST...\n");

break;

}

else

{

printf("Press 1 to delete by specifying position.\n");

printf("Press 2 to delete by specifying the element.\n");

scanf("%d",&del);

if(del==1)

{

printf("Enter the position of element to be deleted.\n");

scanf("%d",&pos);

if(pos>n || pos<1)

{

printf("INVALID POSITION ENTERED..\n");

break;

}

else

{

n--;

if(pos==1)

{

head=delete\_front(head);

break;

}

else

{

delete\_pos(head,pos);

break;

}

}

}

else if(del==2)

{

n--;

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

scanf("%d",&x);

int check=0;

struct node \*d=head;

for(i=1;d!=NULL;i++)

{

if(d->data==x)

{

check++;

break;

}

d=d->next;

}

if(check==0)

{printf("The given element does not present in the list..\n");

break;}

else

{

if(head->data==x)

{

head=delete\_front(head);

break;

}

else

{

delete\_pos(head,i);

break;

}

}

}

else

{

printf("INVALID INPUT.!!\n");

}

}

}

case 4:

{

display(head);

break;

}

case 5:

{

printf("Enter the number of elements in first and second list respectively.\n");

scanf("%d %d",&n1,&n2);

n=n1+n2;

head=merge\_lists(n1,n2);

}

}

}while(op!=6);

return 0;

}

# **EXPLANATION** :

The code is mainly based on the user defined functions present. I have created separate functions to do various operations on the linked list. First I created the node of type structure which contains the data part and the next part which points to the next node in the list. The node is given dynamic memory allocation to expand or reduce to our wish. Then comes the functions which do the following operations.

1. **INT IS\_EMPTY()** **:**

This function takes the head pointer as the parameter to check whether the link is empty or not. The list is empty if the head is null. If empty it returns 1 which indicates true and 0 if not empty which indicates false.

1. **VOID DISPLAY() :**

This function is used to display the elements present in the list. The function takes head pointer as parameter and a current pointer is created to move along the list and print elements. Initially current is pointed to head and a “for loop” is used to move the cur pointer as well as printing the elements till it reaches null. When it reaches null, it prints NULL.

1. **STRUCT NODE \*INSERT\_FRONT() :**

This function is to insert an element at the front of the list. It takes “head” pointer and the element to be inserted as parameters. There I created a temporary node to insert and the element is assigned to the data part. The next part of temp should point to head and the head is shifted on to temp node, such that now the temp node is the first node which is pointed by head. Later it prints the confirmation statement and returns the head pointer to main function from where it is accessed.

1. **VOID INSERT() :**

This function is to insert an element in any other positions. It takes “head”, the element (dat) and the position (pos) of insertion as parameters. Here also a temporary node (temp) is created and the data part is assigned. Another node named “cur” is created to travel across the list. The current pointer is moved by using “for” loop (iterating from 1 to pos-1) to reach the node after which the insertion is to be done. Now the next part of temp should point to where the next part of cur is pointing and later the next part of cur is made to point to temp. So that the temp node is placed after the cur node in the ist. Thus the insertion of element is done.

1. **STRUCT NODE \*DELETE\_FRONT() :**

This function performs deletion of the first element in the list. The head pointer is the parameter for this function. We use a temporary node to store the head node initially and then the head node is shifted to next node. Now the first node is pointed by temp and that is freed to delete from the list. The head node is returned to the main function.

1. **VOID DELETE\_POS() :**

This function is to delete an element given the position. The head pointer and the position of element are passed as parameters. Here also a ”cur” pointer is used to travel across the list. The current pointer is moved by using “for” loop (iterating from 1 to pos-1) to reach the node previous to the node to be deleted. Now a temporary node is created to point to the next of cur, which is to be deleted. The next part of cur is made to point to the node present after the node to be deleted. So the temp node which is to be deleted is separated from the list and is freed to release memory.

1. **VOID SORT\_LIST() :**

This function sorts the list in ascending order by taking head pointer as parameter. Two temporary nodes are created (current and temp) which access two numbers of the list at a time and compare them. Initially current is assigned with head and tmp with the next of head. Using two while loops, one runs till current reaches null and the other till tmp reaches null, the current node is compared with all the other nodes coming next to it. If the data of current is greater than that of tmp, they will be swapped. This goes on till current reaches null i.e., all the nodes are compared with others. Swapping takes place whenever first data is greater than the next. So the whole list gets sorted at the end.

1. **STRUCT NODE \* MERGE\_LISTS() :**

This function is used to merge two lists. It takes the size of two lists to be merged as parameters (x1 and x2). Inside the function, 5 node structures are created. “h1” and “h2” points as heads (initially null) for the two linked lists. “t1” and “t2” as tail pointers and “p1” and “p2” to store the elements (memory -- dynamic). Using “for” loop, iterating x1 times, data part of p1 is assigned. For first iteration, head and are made to point p1 and for next iterations, the tail pointer goes on storing the address of p1 and shifting itself to p1 so that the link is made. Same procedure is followed for the second list too which iterates x2 times. Finally, the next part of tail pointer of first list is made to point onto h2 which points to head of second list. So now, the last node of first list is linked to first node of second list. Thus, the two lists are merged. At last the “sort\_list” function is accessed to sort the elements of merged list and returns the head pointer (h1) to the main function.

## **IN MAIN FUNCTION :**

Main function here mostly handles the variables. The head node from structure node is initialized to null. First all the operations available are printed to get user input. The user input to choose an operation to perform is stored in variable “op“ and it is used in switch case.

The variable “**n**” is used to store the number of elements present in the list. The value of **n**“ is incremented if insertion takes place and decremented if deletion takes place. The loop “do-while” (with exit condition specified) is used to perform the operations multiple times.

1. **CASE 1 (OP==1) :**

This case is used to check whether the list is empty. Uses “is\_empty()” function to check and print the status of the list.

1. **CASE 2 (OP==2)** **:**

This case handles the insertion operation. This again provides two different choices.

**CHOICE 1**: To insert w.r.t. Position given : Takes user input of element and position (pos).

If the given pos is greater than n+1 or less than 1, it means the position is invalid and the program gets out of the switch loop. If the position given is executable and is equal to 1, then it calls the “**insert\_front()**” function to insert at the start and the returned pointer is stored in the head pointer. For all the other positions “**insert()**” function is used.

**CHOICE 2** : To insert after a specified element : This function is executable only if the list contains at least 1 element. If the list is empty, it gets out of there. Else , it gets the data from the user and a temporary pointer “c” is used to traverse along the list. Using “for” loop, we compare the data part of “c node” with the element after which new element should be inserted. “Check” variable is used to verify the element. If it is not found then check is 0 and loop ends. If the element is found, then the iterations done in “for loop” will give the position of the node that contains the element. So we can access “**insert()”** function passing “head” and “variable i” as parameters to insert the new element in “ (i+1) th” position.

If the input is neither 1 nor 2, then it prints “invalid input”.

1. **CASE 3 (OP==3) :**

This case is for the deletion of an element. It again provides 2 choices. It will not be operated if the list is empty. So that is checked first.

**CHOICE 1 :** To delete by specifying position : it gets input from user, the position that should be deleted. If the position is greater than no.of elements (n) present, then it returns “Invalid position” and gets out of the loop. If the position thai is to be deleted is 1, then it calls the function “delete\_front” passing “head” pointer as parameter and stores the new head. If other position is specified then the function “delete\_pos” will be called with parameters “head” and position to be deleted.

**CHOICE 2 :** To delete by specifying the element itself : As explained in the “insertion after an element” choice, here also it checks the presence of element, calls “delete\_pos” function by passing “head” and “no.of iterations” (that gives the position) as parameters.

if the input is neither 1 nor 2, then it prints “invalid input”.

1. **CASE 4 (OP==4) :**

This option just displays the whole list by calling “display()” function.

1. **CASE 5 (OP==5) :**

This is to merge two given linked lists. It takes the sizes of two lists from the user and stores them as “n1” and “n2”. The sum of these two will be assigned to “n”, as it is the total no.of elements present. Then it calls “merge\_lists” function passing “n1” and “n2” as parameters and the returned node is stored in the head pointer so as to enable to perform further operations on the merged list.

1. **CASE 6 (OP==6) :** This is the exit condition for the “do-while” loop. Program ends here.