

> Lab Program 5:

display singly linked list delete and
display implementation

```
#include <iostream.h>
#include <stdlib.h>
Struct node {
    int data;
    struct node* next;
};

Void display();
Void insert_begin();
Void insert_end();
Void insert_pos();
Void begin_delete();
Struct node *head = NULL;
Void display();
Printf ("Elements are : \n");
Struct node *ptr;
If (head == NULL)
{
    Printf ("list is empty");
    return;
}
Else {
    ptr = head;
    While (ptr != NULL)
```

```

    {
        printf("%d\n", *ptr->data);
        ptr = ptr->next;
    }

}

void insert_begin()
{
    struct node *temp;
    temp = (struct node*) malloc(sizeof(struct node));
    printf("enter the value to be inserted\n");
    scanf("%d", &temp->data);
    temp->next = NULL;
    if (head == NULL)
        head = temp;
    else
        temp->next = head;
    head = temp;
}

void insert_end()
{
    struct node *temp, *ptr;
    temp = (struct node*) malloc(sizeof(struct node));
    printf("enter the value to be inserted\n");
    temp->next = NULL;
    if (head == NULL)
        head = temp;
    else
        {
            ptr = head;
            while (ptr->next != NULL)
                ptr = ptr->next;
            ptr->next = temp;
        }
}

```

```

ptr = head;
while (ptr->next != NULL) {
    ptr = ptr->next;
}
ptr->next = temp;
}

void insert_pos() {
    int pos;
    if (pos < 0 || pos >= size_of_struct_node) {
        printf("Enter the Position");
        scanf("%d", &pos);
    }
    struct node *temp, *ptr;
    printf("Enter the value to be inserted");
    scanf("%d", &temp->data);
    temp->next = NULL;
    if (pos == 0) {
        temp->next = head;
        head = temp;
    } else {
        for (i = 0, ptr = head; i < pos - 1; i++) {
            ptr = ptr->next;
        }
        temp->next = ptr->next;
        ptr->next = temp;
    }
}

```

```
void begin_delete(C) { // begin delete function
```

```
{ struct node * Ptr;
```

```
if (head == NULL)
```

```
{ printf ("In list is empty\n"); }
```

```
else { head = NULL; }
```

```
printf ("Node deleted from the
```

```
beginning...\n"); }
```

```
head = Ptr -> next;
```

```
free (Ptr);
```

```
printf ("In Node deleted from the
```

```
beginning...\n"); }
```

```
else { printf ("list is not empty\n"); }
```

```
void last_delete(C) { // last delete
```

```
{ struct node * ptr, * Ptr;
```

```
if (head == NULL)
```

```
{ printf ("list is empty\n"); }
```

```
else if (head -> next == NULL)
```

```
{ head = NULL; }
```

```
free (head); }
```

```
printf ("In Only node of the list
```

```
deleted...\n"); }
```

```
else { printf ("list is not empty\n"); }
```

```
if (ptr == head, last, (head->next) == NULL)
```



```

while (ptr->next != NULL)
{
    ptr1 = ptr;
    ptr = ptr->next; // start from head
}

ptr1->next = NULL; // free the last node
free (ptr);
printf ("\n Deleted Node from the last. \n");
}

void random_delete()
{
    struct node *ptr, *ptr1;
    int loc, i;
    printf ("\nEnter the location of the node after which you want to perform deletion\n");
    scanf ("%d", &loc);
    ptr = head;
    for (i=0; i<loc; i++)
    {
        if (ptr->next == NULL)
        {
            printf ("can't delete");
            return;
        }
        ptr1 = ptr;
        ptr = ptr->next; // start from head
    }

    if (ptr == NULL)
    {
        printf ("can't delete");
        return;
    }

    ptr1->next = ptr->next;
    free (ptr);
    printf ("\n Deleted node l.d %d, loc %d", loc+1);
}

```



```

void main()
{
    int choice;
    while(1)
    {
        printf ("1. to insert at beginning \n"
                "2. to insert at end \n"
                "3. to insert at Position \n"
                "4. to display \n"
                "5. delete from beginning \n"
                "6. delete from end \n"
                "7. random delete \n"
                "e. exit \n");
    }
}

```

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

scanf ("%d", &choice);

switch (choice)

{

case 1:

insert_begin();
break;

case 2:

insert_end();
break; // word ends

case 3:

insert_pos();

case 4:

display();

break; // if condition not met
// otherwise add entry and return

case 5:

begin_delete();
break; // if condition not met
// otherwise add entry and return



```
case 6;
    last_delete();
    break;

case 7;
    random_delete();
    break; // If user enters 7, it will delete a random node from the list

case 8;
    exit(0);
    break; // If user enters 8, it will exit the program

default;
    printf("invalid choice\n");
    break; // If user enters anything other than 1-8, it will print "invalid choice" and loop back to the beginning
```

Output:

1. to insert at beginning
2. to insert at end
3. to insert at Position
4. to display
5. delete from beginning
6. delete from end
7. delete random delete
8. exit

Enter Your choice: 1

Enter the value to be inserted : 10

Enter your choice: 2
Enter the value to be inserted: 20

Enter your choice : 3
Enter the position : 1
Enter the value to be inserted : 15

elements are 10, 15, 20, 25, 30, 35, 40, 45, 50.

Enter your choice (1:15 min + 1 hour) if
node deleted from the beginning . . .

Enter Your choice : - {
Enter the Allocation of nodes after which you want
to Perform (Deletion + Insert) Job & ID }
Delete

Deleted node 2 (last node processed)

Enter your choice : 6 & show flight found
only node of the flight is deleted in fig for
Explain why = 1.89

Enter your choice : 4
list is empty.

Enter your choice : 8. Exit

~~for Cite & Triple -> it's good - I like it~~
Hence forward - don't & about this
it's been - taken care of
inches - 18/12/2019
inches - 18/12/2019
inches - 18/12/2019

1966 - presents - this is the standard form.

Date: 11/1/24

leetCode - 1 week to train
Week 3 3D Boxes

#include <stdlib.h>

```

typedef struct {
    int *Stack;
    int *minStack;
    int top;
} MinStack;

MinStack* minStackCreate() {
    MinStack* Stack = (MinStack*) malloc(sizeof(MinStack));
    Stack->Stack = (int*) malloc(sizeof(int)*10000);
    Stack->minStack = (int*) malloc(sizeof(int)*10000);
    Stack->top = -1;
    return Stack;
}

void minStackPush(MinStack* Obj, int val) {
    Obj->top++;
    Obj->Stack[Obj->top] = val;
    if (Obj->top == 0 || val <= Obj->minStack[Obj->top-1]) {
        Obj->minStack[Obj->top] = val;
    } else {
        Obj->minStack[Obj->top] = Obj->minStack[Obj->top-1];
    }
}

```

```

void minStackPop (MinStack * Obj) {
    Obj->top = ;
}

int minStackTop (MinStack * Obj) {
    return Obj->stack [Obj->top];
}

int minStackGetMin (MinStack * Obj) {
    return Obj->minStack [Obj->top];
}

void minStackFree (MinStack * Obj) {
    free (Obj->stack);
    free (Obj->minStack);
    free (Obj);
}

```

Output: ["MinStack", "Push", "Push", "Push",
 "Push", "GetMin", "Pop", "Pop", "GetMin"]

[[], [-2], [0], [-3], [1], [1], [1], [1]]

Output: [null, null, null, null, -3, null, 0, -2]

(This is not the right answer)
 (off by one error)
 (Index out of bound)

(Program is bad?) I think



```
#include<stdio.h>
#include<stdlib.h>
struct node{
int data;
struct node*next;
};
void display();
void insert_begin();
void insert_end();
void insert_pos();
void begin_delete();
struct node *head=NULL;
void display()
{
    printf("elements are :\n");
    struct node *ptr;
    if(head==NULL)
    {
        printf("list is empty");
        return;
    }
    else{
        ptr=head;
        while(ptr !=NULL)
        {
            printf("%d\n", ptr->data);
            ptr=ptr->next;
        }
    }
}
void insert_begin()
{
    struct node*temp;
    temp =(struct node*)malloc(sizeof(struct node));
    printf("enter the value to be inserted\n");
    scanf("%d",&temp->data);
    temp->next=NULL;
    if(head==NULL)
        head=temp;
    else{
        temp->next=head;
        head=temp;
    }
}
```

```

temp->next=NULL;
if(head==NULL)
    head=temp;
else{
    temp->next=head;
    head=temp;
}

void insert_end()

struct node *temp,*ptr;
temp=(struct node*)malloc(sizeof(struct node));
printf("enter the value to be inserted \n");
scanf("%d",&temp->data);
temp->next=NULL;
if(head==NULL)
{
    head=temp;
}
else
{
    ptr=head;
    while(ptr->next != NULL)
    {

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

void insert_pos()

int pos,i;
struct node*temp,*ptr;
printf("enter the position");
scanf("%d",&pos);
temp=(struct node*)malloc(sizeof(struct node));
printf("enter the value to be inserted\n");
scanf("%d",&temp->data);
temp->next=NULL;
if(pos==0)

```

```

temp=(struct node*)malloc(sizeof(struct node));
printf("enter the value to be inserted\n");
scanf("%d",&temp->data);
temp->next=NULL;
if(pos==0)
{
    temp->next=head;
    head=temp;
}
else
{
    for(i=0, ptr=head; i<pos-1;i++)
    {
        ptr=ptr->next;
    }
    temp->next=ptr->next;
    ptr->next=temp;
}

void begin_delete()
{
    struct node *ptr;
    if(head == NULL)
    {
        printf("\nList is empty\n");
    }
    else
    {
        ptr = head;
        head = ptr->next;
        free(ptr);
        printf("\nNode deleted from the begining ... \n");
    }
}

void last_delete()
{
    struct node *ptr,*ptr1;
    if(head == NULL)
    {
        printf("\nlist is empty");
    }
}

```

```
void last_delete()

    struct node *ptr,*ptr1;
    if(head == NULL)
    {
        printf("\nlist is empty");
    }
    else if(head -> next == NULL)
    {
        head = NULL;
        free(head);
        printf("\nOnly node of the list deleted ... \n");
    }

    else
    {
        ptr = head;
        while(ptr->next != NULL)
        {
            ptr1 = ptr;
            ptr = ptr ->next;
        }
        ptr1->next = NULL;
        free(ptr);
        printf("\nDeleted Node from the last ... \n");
    }
}

void random_delete()

    struct node *ptr,*ptr1;
    int loc,i;
    printf("\n Enter the location of the node after which you want to perform deletion \n");
    scanf("%d",&loc);
    ptr=head;
    for(i=0;i<loc;i++)
    {
        ptr1 = ptr;
        ptr = ptr->next;

        if(ptr == NULL)
        {
            printf("\nCan't delete");
            return;
        }
    }
```



```
if(ptr == NULL)
{
    printf("\nCan't delete");
    return;
}
ptr1 ->next = ptr ->next;
free(ptr);
printf("\nDeleted node %d ",loc+1);

void main()

int choice;
while(1)
{

    printf("\n 1.to insert at the beginning\n"
           " 2.to insert at the end\n"
           " 3.to insert at the position\n"
           " 4.to display\n"
           " 5.delete from beginning\n"
           " 6.delete from end\n"
           " 7.random delete\n"
           " 8.exit\n");
    printf("enter your choice:\n");
    scanf("%d",&choice);
    switch(choice)
    {

        case 1:
            insert_begin();
            break;
        case 2:
            insert_end();
            break;
        case 3:
            insert_pos();
            break;
        case 4:
            display();
            break;
        case 5:
            begin_delete();
            break;
    }
}
```

```
printf("\n 1.to insert at the beginning\n"
      " 2.to insert at the end\n"
      "3.to insert at the position\n"
      "4.to display\n"
      "5.delete from beginning\n"
      "6.delete from end\n"
      "7.random delete\n"
      "8.exit\n");
printf("enter you choice:\n");
scanf("%d",&choice);
switch(choice)
{
    case 1:
        insert_begin();
        break;
    case 2:
        insert_end();
        break;
    case 3:
        insert_pos();
        break;
    case 4:
        display();
        break;
    case 5:
        begin_delete();
        break;
    case 6:
        last_delete();
        break;
    case 7:
        random_delete();
        break;
    case 8:
        exit(0);
        break;
    default:
        printf("invalid choice\n");
        break;
}
```

```
1.to insert at the beginning
2.to insert at the end
3.to insert at the position
4.to display
5.delete from beginning
6.delete from end
7.random delete
8.exit
enter you choice:
3
enter the position1
enter the value to be inserted
30

1.to insert at the beginning
2.to insert at the end
3.to insert at the position
4.to display
5.delete from beginning
6.delete from end
7.random delete
8.exit
enter you choice:
4
elements are :
10
30
20

1.to insert at the beginning
2.to insert at the end
3.to insert at the position
4.to display
5.delete from beginning
6.delete from end
7.random delete
8.exit
enter you choice:
5

Node deleted from the begining ...

1.to insert at the beginning
2.to insert at the end
3.to insert at the position
4.to display
5.delete from beginning
6.delete from end
7.random delete
8.exit
enter you choice:
7

Enter the location of the node after which you want to perform deletion
1

Deleted node 2
1.to insert at the beginning
2.to insert at the end
3.to insert at the position
4.to display
5.delete from beginning
6.delete from end
```



```
lements are :  
0  
0  
0  
  
1.to insert at the beginning  
2.to insert at the end  
3.to insert at the position  
4.to display  
5.delete from beginning  
.delete from end  
.random delete  
.exit  
nter you choice:
```

```
ode deleted from the begining ...
```

```
1.to insert at the beginning  
2.to insert at the end  
3.to insert at the position  
4.to display  
5.delete from beginning  
.delete from end  
.random delete  
.exit  
nter you choice:
```

```
Enter the location of the node after which you want to perform deletion
```

```
leted node 2  
1.to insert at the beginning  
2.to insert at the end  
3.to insert at the position  
4.to display  
5.delete from beginning  
.delete from end  
.random delete  
.exit  
nter you choice:
```

```
lements are :  
0  
  
1.to insert at the beginning  
2.to insert at the end  
3.to insert at the position  
4.to display  
5.delete from beginning  
.delete from end  
.random delete  
.exit  
nter you choice:
```



LeetCode - 1
Week 3

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

```
typedef struct {
```

```
    int *Stack;
```

```
    int top;
```

```
} MinStack;
```

```
MinStack * minStackCreate ()
```

```
MinStack * Stack = (MinStack *) malloc
```

```
(sizeof(MinStack));
```

```
Stack->Stack = (int *) malloc(sizeof(int)*
```

```
(int(10000));
```

```
Stack->minStack = (int *) malloc(sizeof(int)*10000);
```

```
Stack->top = -1;
```

```
return Stack;
```

```
}
```

```
Void minStackPush (MinStack * Obj, int val)
```

```
Obj->top++;
```

```
Obj->Stack[Obj->top] = val;
```

```
if (Obj->top == 0 || val <= Obj->minStack[Obj->top-1])
```

```
{
```

```
    Obj->minStack[Obj->top] = val;
```

```
} else {
```

```
    Obj->minStack[Obj->top] = Obj->minStack[Obj->
```

```
    top-1];
```

```
}
```

```
}
```



```

void minStackPop (MinStack* obj) {
    obj->top--;
}

int minStackTop (MinStack* obj) {
    return obj->stack [obj->top];
}

int minStackGetMin (MinStack* obj) {
    return obj->minStack [obj->top];
}

void minStackFree (MinStack* obj) {
    free (obj->stack);
    free (obj->minStack);
    free (obj);
}

```

Output: ["MinStack", "Push", "Push", "Push",
 "GetMin", "Pop", "Top", "GetMin"]

[[], [-2], [0], [-3], [3], [], [], []]

Output: [null, null, null, null, -3, null, 0, -2]

~~if (obj->stack == NULL) { return;~~

~~if (obj->minStack == NULL) { return;~~

~~if (obj->top == -1) { return;~~

~~{ "push and a fail" } return;~~

~~{ "return" }~~

~~3 null~~

~~3 -2~~

~~3 0~~

~~3 -3~~



```
1 #include <stdlib.h>
2
3
4 typedef struct {
5     int *stack;
6     int *minStack;
7     int top
8 } MinStack;
9
10
11 MinStack* minStackCreate() {
12     MinStack* stack = (MinStack*)malloc(sizeof(MinStack));
13     stack->stack = (int*)malloc(sizeof(int) * 10000);
14     stack->minStack = (int*)malloc(sizeof(int) * 10000);
15     stack->top = -1;
16     return stack;
17 }
18
19 void minStackPush(MinStack* obj, int val) {
20     obj->top++;
21     obj->stack[obj->top] = val;
22     if (obj->top == 0 || val <= obj->minStack[obj->top - 1]) {
23         obj->minStack[obj->top] = val;
24     } else {
25         obj->minStack[obj->top] = obj->minStack[obj->top - 1];
26     }
27 }
28
29 void minStackPop(MinStack* obj) {
30     obj->top--;
31 }
```



```
20 obj->top++;
21 obj->stack[obj->top] = val;
22 if (obj->top == 0 || val <= obj->minStack[obj->top - 1]) {
23 obj->minStack[obj->top] = val;
24 } else {
25 obj->minStack[obj->top] = obj->minStack[obj->top - 1];
26 }
27 }
28
29 void minStackPop(MinStack* obj) {
30 obj->top--;
31 }
32
33 int minStackTop(MinStack* obj) {
34 return obj->stack[obj->top];
35 }
36
37 int minStackGetMin(MinStack* obj) {
38 return obj->minStack[obj->top];
39 }
40
41 void minStackFree(MinStack* obj) {
42 free(obj->stack);
43 free(obj->minStack);
44 free(obj);
45 }
```



- Case 1

Input

```
["MinStack","push","push","push","getMin","pop","top","getMin"]
```

```
[[],[-2],[0],[-3],[],[],[],[]]
```

Output

```
[null,null,null,null,-3,null,0,-2]
```

Expected

```
[null,null,null,null,-3,null,0,-2]
```



Scanned with OKEN Scanner