

<b>Experiment No.6</b>
Implement Singly Linked List ADT
Name:Nitish Jha
Roll No:18
Date of Performance:
Date of Submission:
Marks:
Sign:

### **Experiment No. 6: Singly Linked List Operations**

**Aim: Implementation of Singly Linked List**

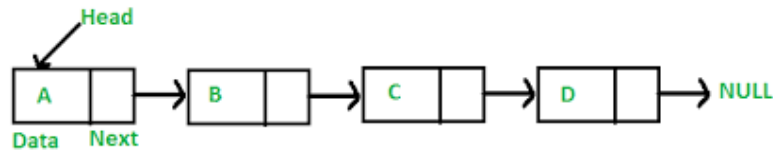
**Objective:**

It is used to implement stacks and queues which are like fundamental needs throughout computer science. To prevent the collision between the data in the hash map, we use a singly linked list.

**Theory:**

A linked list is an ordered collection of elements, known as nodes. Each node has two fields: one for data (information) and another to store the address of the next element in the list. The address field of the last node is null, indicating the end of the list. Unlike arrays, linked list elements are not stored in contiguous memory locations; instead, they are connected by explicit links, allowing for dynamic and non-contiguous memory allocation.

The structure of linked list is as shown below



Header is a node containing null in its information field and an next address field contains the address of the first data node in the list. Various operations can be performed on singly linked lists like insertion at front, end, after a given node, before a given node deletion at front, at end and after a given node.

### Algorithm

Algorithm to insert a new node at the beginning

Step 1: IF AVAIL = NULL

Write OVERFLOW

Go to Step 7 [END OF IF]

Step 2: SET NEW\_NODE = AVAIL

Step 3: SET AVAIL = AVAIL NEXT

Step 4: SET DATA = VAL

Step 5: SET NEW\_NODE -->NEXT = START

Step 6: SET START = NEW\_NODE

Step 7: EXIT

Algorithm to insert a new node at the end

Step 1: IF AVAIL = NULL

Write OVERFLOW

Go to Step 1 [END OF IF]

Step 2: SET = AVAIL

Step 3: SET AVAIL = AVAIL NEXT  
Step 4: SET DATA = VAL  
Step 5: SET NEW\_NODE = NULL  
Step 6: SET PTR = START  
Step 7: Repeat Step 8 while PTR NEXT != NULL  
Step 8: SET PTR = PTR NEXT [END OF LOOP]  
Step 9: SET PTR--> NEXT = New\_Node  
Step 10: EXIT

Algorithm to insert a new node after a node that has value NUM

Step 1: IF AVAIL = NULL  
    Write OVERFLOW  
    Go to Step 12 [END OF IF]  
Step 2: SET = AVAIL  
Step 3: SET AVAIL = AVAIL-->NEXT  
Step 4: SET DATA = VAL  
Step 5: SET PTR = START  
Step 6: SET PREPTR = PTR  
Step 7: Repeat Steps 8 and 9 while != NUM  
Step 8: SET PREPTR = PTR  
Step 9: SET PTR = PTR -->NEXT  
[END OF LOOP]  
Step 10 : PREPTR--> NEXT = NEW\_NODE  
Step 11: SET NEW\_NODE NEXT = PTR  
Step 12: EXIT

Algorithm to insert a new node before a node that has value NUM

Step 1: IF AVAIL = NULL  
    Write OVERFLOW

Go to Step 12 [END OF IF]

Step 2: SET = AVAIL

Step 3: SET AVAIL = AVAIL-->NEXT

Step 4: SET DATA = VAL

Step 5: SET PTR = START

Step 6: SET PREPTR = PTR

Step 7: Repeat Steps 8 and 9 while PTR DATA != NUM

Step 8: SET PREPTR = PTR

Step 9: SET PTR = PTR -->NEXT

[END OF LOOP]

Step 10: PREPTR-->NEXT = NEW\_NODE

Step 11: SET NEXT = PTR

Step 12: EXIT

Algorithm to delete the first node

Step 1: IF START = NULL

Write UNDERFLOW

Go to Step 5 [END OF IF]

Step 2: SET PTR = START

Step 3: SET START = START -->NEXT

Step 4: FREE PTR

Step 5: EXIT

Algorithm to delete the last node

Step 1: IF START = NULL

Write UNDERFLOW

Go to Step 8 [END OF IF]

Step 2: SET PTR = START

Step 3: Repeat Steps 4 and 5 while PTR NEXT != NULL

Step 4: SET PREPTR = PTR

Step 5: SET PTR = PTR -->NEXT [END OF LOOP]

Step 6: SET PREPTR-->NEXT = NULL

Step 7: FREE PTR

Step 8: EXIT

Algorithm to delete the node after a given node

Step 1: IF START = NULL

    Write UNDERFLOW

    Go to Step 1 [END OF IF]

Step 2: SET PTR = START

Step 3: SET PREPTR = PTR

Step 4: Repeat Steps 5 and 6 while PREPTR DATA != NUM

Step 5: SET PREPTR = PTR

Step 6: SET PTR = PTR--> NEXT

[END OF LOOP]

Step 7: SET TEMP = PTR

Step 8: SET PREPTR -->NEXT = PTR--> NEXT

Step 9: FREE TEMP

Step 10: EXIT

**Code:**

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

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

```
#include <conio.h>
```

```

#include <malloc.h>

struct node
{
    int data;

    struct node *next;
};

struct node *start = NULL;

struct node *create_ll(struct node *);

struct node *display(struct node *);

struct node *insert_beg(struct node *);

struct node *insert_end(struct node *);

struct node *insert_before(struct node *);

struct node *insert_after(struct node *);

struct node *delete_beg(struct node *);

struct node *delete_end(struct node *);

struct node *delete_node(struct node *);

struct node *delete_after(struct node *);

struct node *delete_list(struct node *);

struct node *sort_list(struct node *);

int main(int argc, char *argv[]) {

    int option;

    do

    {

        printf("\n\n ** IMPEMENRTATION OF SINGLY LINDED LIST **");

        printf("\n 1: Create a list");

        printf("\n 2: Display the list");

        printf("\n 3: Add a node at the beginning");
    }

```

```
printf("\n 4: Add a node at the end");

printf("\n 5: Add a node before a given node");

printf("\n 6: Add a node after a given node");

printf("\n 7: Delete a node from the beginning");

printf("\n 8: Delete a node from the end");

printf("\n 9: Delete a given node");

printf("\n 10: Delete a node after a given node");

printf("\n 11: Delete the entire list");

printf("\n 12: Sort the list");

printf("\n 13: EXIT");

printf("\n\n Enter your option : ");

scanf("%d", &option);

switch(option)

{

case 1: start = create_ll(start);

printf("\n LINKED LIST CREATED");

break;

case 2: start = display(start);

break;

case 3: start = insert_beg(start);

break;

case 4: start = insert_end(start);

break;

case 5: start = insert_before(start);

break;

case 6: start = insert_after(start);

break;
```

```

case 7: start = delete_beg(start);

break;

case 8: start = delete_end(start);

break;

case 9: start = delete_node(start);

break;

case 10: start = delete_after(start);

break;

case 11: start = delete_list(start);

printf("\n LINKED LIST DELETED");

break;

case 12: start = sort_list(start);

break;

}

}while(option !=13);

getch();

return 0;

}

struct node *create_ll(struct node *start)

{

struct node *new_node, *ptr;

int num;

printf("\n Enter -1 to end");

printf("\n Enter the data : ");

scanf("%d", &num);

while(num!=-1)

{

```



```
new_node = (struct node*)malloc(sizeof(struct node));
```

```
new_node -> data=num;
```

```
if(start==NULL)
```

```
{
```

```
new_node -> next = NULL;
```

```
start = new_node;
```

```
}
```

```
else
```

```
{
```

```
ptr=start;
```

```
while(ptr->next!=NULL)
```

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

```
ptr->next = new_node;
```

```
new_node->next=NULL;
```

```
}
```

```
printf("\n Enter the data : ");
```

```
scanf("%d", &num);
```

```
}
```

```
return start;
```

```
}
```

```
struct node *display(struct node *start)
```

```
{
```

```
struct node *ptr;
```

```
ptr = start;
```

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

```
{
```

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

```

    ptr = ptr -> next;
}

return start;
}

struct node *insert_beg(struct node *start)
{
    struct node *new_node;

    int num;

    printf("\n Enter the data : ");

    scanf("%d", &num);

    new_node = (struct node *)malloc(sizeof(struct node));

    new_node -> data = num;

    new_node -> next = start;

    start = new_node;

    return start;
}

struct node *insert_end(struct node *start)
{
    struct node *ptr, *new_node;

    int num;

    printf("\n Enter the data : ");

    scanf("%d", &num);

    new_node = (struct node *)malloc(sizeof(struct node));

    new_node -> data = num;

    new_node -> next = NULL;

    ptr = start;

    while(ptr -> next != NULL)

```

```

ptr = ptr -> next;

ptr -> next = new_node;

return start;
}

struct node *insert_before(struct node *start)
{
    struct node *new_node, *ptr, *preptr;
    int num, val;

    printf("\n Enter the data : ");

    scanf("%d", &num);

    printf("\n Enter the value before which the data has to be inserted : ");

    scanf("%d", &val);

    new_node = (struct node *)malloc(sizeof(struct node));

    new_node -> data = num;

    ptr = start;

    while(ptr -> data != val)
    {
        preptr = ptr;

        ptr = ptr -> next;
    }

    preptr -> next = new_node;

    new_node -> next = ptr;

    return start;
}

struct node *insert_after(struct node *start)
{
    struct node *new_node, *ptr, *preptr;

```

```

int num, val;

printf("\n Enter the data : ");

scanf("%d", &num);

printf("\n Enter the value after which the data has to be inserted : ");

scanf("%d", &val);

new_node = (struct node *)malloc(sizeof(struct node));

new_node -> data = num;

ptr = start;

preptr = ptr;

while(preptr -> data != val)

{

    preptr = ptr;

    ptr = ptr -> next;

}

preptr -> next=new_node;

new_node -> next = ptr;

return start;

}

struct node *delete_beg(struct node *start)

{

    struct node *ptr;

    ptr = start;

    start = start -> next;

    free(ptr);

    return start;

}

struct node *delete_end(struct node *start)

```

```

{
    struct node *ptr, *preptr;

    ptr = start;

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

    preptr -> next = NULL;

    free(ptr);

    return start;
}

struct node *delete_node(struct node *start)
{
    struct node *ptr, *preptr;

    int val;

    printf("\n Enter the value of the node which has to be deleted : ");

    scanf("%d", &val);

    ptr = start;

    if(ptr -> data == val)
    {
        start = delete_beg(start);

        return start;
    }

    else

    {
        while(ptr -> data != val)

```

```

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

preptr -> next = ptr -> next;
free(ptr);
return start;
}
}

struct node *delete_after(struct node *start)
{
    struct node *ptr, *preptr;
    int val;

    printf("\n Enter the value after which the node has to deleted : ");
    scanf("%d", &val);

    ptr = start;
    preptr = ptr;
    while(preptr -> data != val)
    {
        preptr = ptr;
        ptr = ptr -> next;
    }

    preptr -> next = ptr -> next;
    free(ptr);
    return start;
}

struct node *delete_list(struct node *start)

```

```

{
    struct node *ptr; // Lines 252-254 were modified from original code to fix
    unresponsiveness in output window

    if(start!=NULL){

        ptr=start;

        while(ptr != NULL)

        {

            printf("\n %d is to be deleted next", ptr -> data);

            start = delete_beg(ptr);

            ptr = start;

        }

    }

    return start;

}

struct node *sort_list(struct node *start)

{

    struct node *ptr1, *ptr2;

    int temp;

    ptr1 = start;

    while(ptr1 -> next != NULL)

    {

        ptr2 = ptr1 -> next;

        while(ptr2 != NULL)

        {

            if(ptr1 -> data > ptr2 -> data)

            {

                temp = ptr1 -> data;

                ptr1 -> data = ptr2 -> data;

```

```

ptr2 -> data = temp;

}

ptr2 = ptr2 -> next;

}

ptr1 = ptr1 -> next;

}

return start;

}

```

### Output:

```

** IMPLEMENTATION OF SINGLY LINKED LIST **
1: Create a list
2: Display the list
3: Add a node at the beginning
4: Add a node at the end
5: Add a node before a given node
6: Add a node after a given node
7: Delete a node from the beginning
8: Delete a node from the end
9: Delete a given node
10: Delete a node after a given node
11: Delete the entire list
12: Sort the list
13: EXIT

Enter your option : 1

Enter -1 to end
Enter the data : 23

Enter the data : 14

Enter the data : -1_

```



```

9: Delete a given node
10: Delete a node after a given node
11: Delete the entire list
12: Sort the list
13: EXIT

Enter your option : 2
                23    14

** IMPLEMENTATION OF SINGLY LINKED LIST **
1: Create a list
2: Display the list
3: Add a node at the beginning
4: Add a node at the end
5: Add a node before a given node
6: Add a node after a given node
7: Delete a node from the beginning
8: Delete a node from the end
9: Delete a given node
10: Delete a node after a given node
11: Delete the entire list
12: Sort the list
13: EXIT

Enter your option : _

```

## Conclusion:

1) Write an example of stack and queue implementation using singly linked list?

- The code presented is designed to handle a singly linked list, featuring an array of functions for performing various operations on it. These operations include creating the list, displaying its contents, appending nodes to the beginning or end, inserting nodes before or after a specified node, deleting nodes, sorting the list, and erasing the entire list. The code is written in C and follows a menu-driven approach for user interaction.

However, it's essential to note that the code relies on certain header files such as `` and ``, which are platform-specific and not part of the standard C library. Additionally, it employs the `getch()` function commonly found in older DOS-based compilers. These dependencies could hinder the code's portability to modern systems.

Regarding your inquiry about implementing stacks and queues using a singly linked list, here's a concise explanation:

### Singly Linked List for Stack Implementation:

A stack can be effectively realized using a singly linked list by concentrating operations on one end of the list, typically the head. The most recently added element becomes the new head. Stack operations involve:

- Push: Adding an element to the head of the list.
- Pop: Removing and returning the element at the head.
- Peek: Retrieving the element at the head without removal.
- IsEmpty: Verifying whether the list is empty.

### Singly Linked List for Queue Implementation:

Likewise, a queue can be implemented using a singly linked list. In a queue, elements are enqueued at the rear and dequeued from the front of the list. Queue operations comprise:

- Enqueue: Adding an element to the rear of the list.
- Dequeue: Removing and returning the element at the front.
- Front: Retrieving the element at the front without removal.
- IsEmpty: Checking whether the list is empty.

To create these data structures with a singly linked list, you can adapt the provided code by defining appropriate functions to handle stack and queue operations. The fundamental concept involves manipulating the linked list in a manner that preserves the intended behavior of stacks and queues.

