Algorithms of several operations:

Traversing:

Traversing is the most common operation that is performed in almost every scenario of singly linked list. Traversing means visiting each node of the list once in order to perform some operation on that. This will be done by using the following statements.

```
1. ptr = head;
2.       while (ptr!=NULL)
3.           {
4.               ptr = ptr -> next;
5.           }
```

Algorithm

- **STEP 1:** SET PTR = HEAD
- **STEP 2:** IF PTR = NULL
- WRITE "EMPTY LIST" GOTO STEP 7 END OF IF**STEP 4:** REPEAT STEP 5 AND 6 UNTIL PTR != NULL
- **STEP 5:** PRINT PTR→ DATA
- **STEP 6:** PTR = PTR → NEXT
- [END OF LOOP]**STEP 7:** EXIT

C function

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. void create(int);
4. void traverse();
5. struct node
```

```
6. {
7.     int data;
8.     struct node *next;
9. };
10.      struct node *head;
11.      void main ()
12.      {
13.          int choice,item;
14.          do
15.          {
16.              printf("\n1.Append
   List\n2.Traverse\n3.Exit\n4.Enter your choice?");
17.              scanf("%d",&choice);
18.              switch(choice)
19.              {
20.                  case 1:
21.                  printf("\nEnter the item\n");
22.                  scanf("%d",&item);
23.                  create(item);
24.                  break;
25.                  case 2:
26.                  traverse();
27.                  break;
28.                  case 3:
29.                  exit(0);
30.                  break;
31.                  default:
32.                  printf("\nPlease enter valid choice\n");
33.              }
34.
35.          }while(choice != 3);
36.      }
37.      void create(int item)
```

```
38.         {
39.             struct node *ptr = (struct node
    *)malloc(sizeof(struct node *));
40.             if(ptr == NULL)
41.             {
42.                 printf("\nOVERFLOW\n");
43.             }
44.             else
45.             {
46.                 ptr->data = item;
47.                 ptr->next = head;
48.                 head = ptr;
49.                 printf("\nNode inserted\n");
50.             }
51.
52.         }
53.       void traverse()
54.         {
55.             struct node *ptr;
56.             ptr = head;
57.             if(ptr == NULL)
58.             {
59.                 printf("Empty list..");
60.             }
61.             else
62.             {
63.                 printf("printing values . . . . .\n");
64.                 while (ptr!=NULL)
65.                 {
66.                     printf("\n%d",ptr->data);
67.                     ptr = ptr -> next;
68.                 }
69.             }
```

```
70.         }
```

Output

```
1.Append List
2.Traverse
3.Exit
4.Enter your choice?1

Enter the item
23

Node inserted

1.Append List
2.Traverse
3.Exit
4.Enter your choice?1

Enter the item
233

Node inserted

1.Append List
2.Traverse
3.Exit
4.Enter your choice?2
printing values . . . . .

233
23
```

Searching:

Searching is performed in order to find the location of a particular element in the list. Searching any element in the list needs traversing through the list and make the comparison of every element of the list with the specified element. If the element is matched with any of the list element then the location of the element is returned from the function.
Algorithm

- **Step 1:** SET PTR = HEAD
- **Step 2:** Set I = 0
- **STEP 3:** IF PTR = NULL
- WRITE "EMPTY LIST"
  GOTO STEP 8
  END OF IF**STEP 4:** REPEAT STEP 5 TO 7 UNTIL PTR != NULL
- **STEP 5:** if ptr → data = item
- write i+1
  End of IF**STEP 6:** I = I + 1
- **STEP 7:** PTR = PTR → NEXT
- [END OF LOOP]**STEP 8:** EXIT

C function

```
1.    #include<stdio.h>
2. #include<stdlib.h>
3. void create(int);
4. void search();
5. struct node
6. {
7.    int data;
8.    struct node *next;
9. };
```

```
10.     struct node *head;
11.     void main ()
12.     {
13.        int choice,item,loc;
14.        do
15.        {
16.           printf("\n1.Create\n2.Search\n3.Exit\n4.Enter
   your choice?");
17.           scanf("%d",&choice);
18.           switch(choice)
19.           {
20.              case 1:
21.              printf("\nEnter the item\n");
22.              scanf("%d",&item);
23.              create(item);
24.              break;
25.              case 2:
26.              search();
27.              case 3:
28.              exit(0);
29.              break;
30.              default:
31.              printf("\nPlease enter valid choice\n");
32.           }
33.
34.        }while(choice != 3);
35.     }
36.        void create(int item)
37.        {
38.           struct node *ptr = (struct node
   *)malloc(sizeof(struct node *));
39.           if(ptr == NULL)
40.           {
```

```
41.             printf("\nOVERFLOW\n");
42.          }
43.          else
44.          {
45.             ptr->data = item;
46.             ptr->next = head;
47.             head = ptr;
48.             printf("\nNode inserted\n");
49.          }
50.
51.       }
52.    void search()
53.    {
54.       struct node *ptr;
55.       int item,i=0,flag;
56.       ptr = head;
57.       if(ptr == NULL)
58.       {
59.          printf("\nEmpty List\n");
60.       }
61.       else
62.       {
63.          printf("\nEnter item which you want to
   search?\n");
64.          scanf("%d",&item);
65.          while (ptr!=NULL)
66.          {
67.             if(ptr->data == item)
68.             {
69.                printf("item found at location %d ",i+1);
70.                flag=0;
71.             }
72.             else
```

```
73.                 {
74.                     flag=1;
75.                 }
76.                 i++;
77.                 ptr = ptr -> next;
78.             }
79.             if(flag==1)
80.             {
81.                 printf("Item not found\n");
82.             }
83.         }
84.
85.     }
```

Output

```
1.Create
2.Search
3.Exit
4.Enter your choice?1

Enter the item
23

Node inserted

1.Create
2.Search
3.Exit
4.Enter your choice?1

Enter the item
34

Node inserted
```

1.Create
2.Search
3.Exit
4.Enter your choice?2

Enter item which you want to search?
34
item found at location 1

Insertion into Linked List:

Inserting a new element into a singly linked list at beginning is quite simple. We just need to make a few adjustments in the node links. There are the following steps which need to be followed in order to inser a new node in the list at beginning.

- Allocate the space for the new node and store data into the data part of the node. This will be done by the following statements.

1. ptr = (struct node *) malloc(sizeof(struct node *));
2. ptr → data = item

- Make the link part of the new node pointing to the existing first node of the list. This will be done by using the following statement.

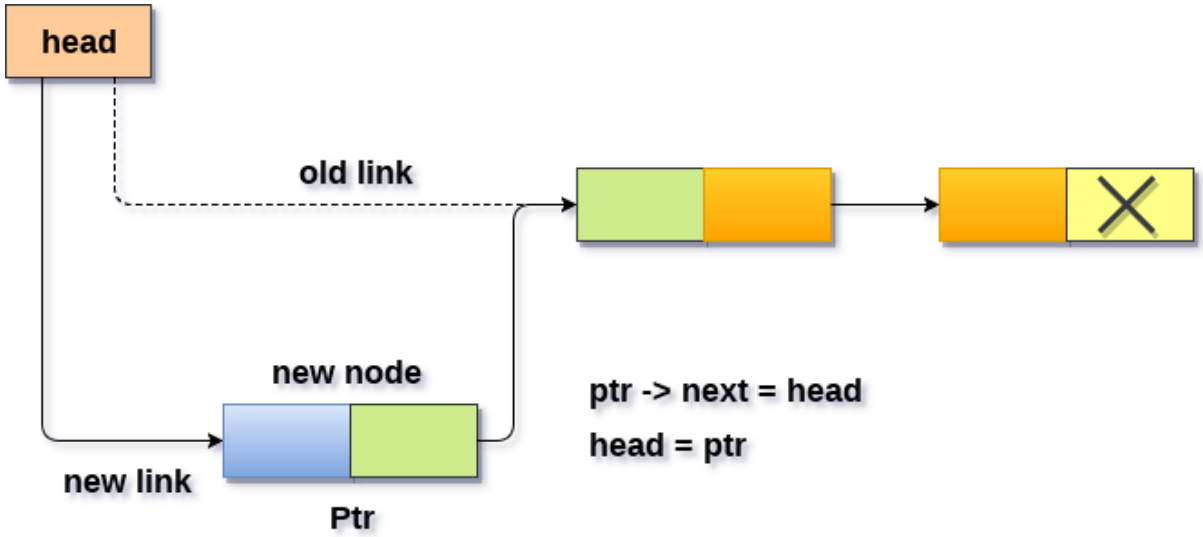
1. ptr->next = head;

- At the last, we need to make the new node as the first node of the list this will be done by using the following statement.

1. head = ptr;

Algorithm

- Step 1: IF PTR = NULL
- Write OVERFLOW
  Go to Step 7
  [END OF IF]Step 2: SET NEW_NODE = PTR
- Step 3: SET PTR = PTR → NEXT
- Step 4: SET NEW_NODE → DATA = VAL
- Step 5: SET NEW_NODE → NEXT = HEAD
- Step 6: SET HEAD = NEW_NODE
- Step 7: EXIT



C Function

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. void beginsert(int);
4. struct node
5. {
6.     int data;
7.     struct node *next;
8. };
9. struct node *head;
10.       void main ()
11.       {
```

```
12.         int choice,item;
13.         do
14.         {
15.             printf("\nEnter the item which you want to
    insert?\n");
16.             scanf("%d",&item);
17.             beginsert(item);
18.             printf("\nPress 0 to insert more ?\n");
19.             scanf("%d",&choice);
20.         }while(choice == 0);
21.     }
22.     void beginsert(int item)
23.         {
24.             struct node *ptr = (struct node
    *)malloc(sizeof(struct node *));
25.             if(ptr == NULL)
26.             {
27.                 printf("\nOVERFLOW\n");
28.             }
29.             else
30.             {
31.                 ptr->data = item;
32.                 ptr->next = head;
33.                 head = ptr;
34.                 printf("\nNode inserted\n");
35.             }
36.
37.         }
```

Output

Enter the item which you want to insert?

12

Node inserted

Press 0 to insert more ?
0

Enter the item which you want to insert?
23

Node inserted

Press 0 to insert more ?
2

Deletion from linked list:

Delete from a Linked List:-
You can delete an element in a list from:

- Beginning
- End
- Middle

1) **Delete from Beginning:**

Point head to the next node i.e. second node

```
temp = head
head = head->next
```

Make sure to free unused memory

```
free(temp); or delete temp;
```

2) **Delete from End:**

Point head to the previous element i.e. last second element

Change next pointer to null

```
struct node *end = head;
struct node *prev = NULL;
while(end->next)
{
   prev = end;
   end = end->next;
}
prev->next = NULL;
```

Make sure to free unused memory

```
free(end); or delete end;
```

3) **Delete from Middle:**

Keeps track of pointer before node to delete and pointer to node to delete

```
temp = head;
prev = head;
for(int i = 0; i < position; i++)
{
   if(i == 0 && position == 1)
      head = head->next;
      free(temp)
   else
```

```
    {
        if (i == position - 1 && temp)
        {
            prev->next = temp->next;
            free(temp);
        }
        else
        {
            prev = temp;
            if(prev == NULL) // position was greater than number of nodes in the list
                break;
            temp = temp->next;
        }
    }
}
```

Iterative Method to delete an element from the linked list:

To delete a node from the linked list, we need to do the following steps:

- Find the previous node of the node to be deleted.
- Change the **next** of the previous node.
- Free memory for the node to be deleted.

Below is the implementation to delete a node from the list at some position:

- C

```
// C code to delete a node from linked list
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int number;
    struct Node* next;
} Node;

void Push(Node** head, int A)
{
    Node* n = malloc(sizeof(Node));
    n->number = A;
    n->next = *head;
    *head = n;
}

void deleteN(Node** head, int position)
{
    Node* temp;
    Node* prev;
    temp = *head;
    prev = *head;
    for (int i = 0; i < position; i++) {
```

```
        if (i == 0 && position == 1) {
            *head = (*head)->next;
            free(temp);
        }
        else {
            if (i == position - 1 && temp) {
                prev->next = temp->next;
                free(temp);
            }
            else {
                prev = temp;

                // Position was greater than
                // number of nodes in the list
                if (prev == NULL)
                    break;
                temp = temp->next;
            }
        }
    }
}

void printList(Node* head)
{
```

```
    while (head) {
        printf("[%i] [%p]->%p\n", head->number, head,
              head->next);
        head = head->next;
    }
    printf("\n\n");
}

// Drivers code
int main()
{
    Node* list = malloc(sizeof(Node));
    list->next = NULL;
    Push(&list, 1);
    Push(&list, 2);
    Push(&list, 3);

    printList(list);

    // Delete any position from list
    deleteN(&list, 1);
    printList(list);
    return 0;
}
```

Output

[3] [0x1b212c0]->0x1b212a0

[2] [0x1b212a0]->0x1b21280

[1] [0x1b21280]->0x1b21260

[0] [0x1b21260]->(nil)

[2] [0x1b212a0]->0x1b21280

[1] [0x1b21280]->0x1b21260

[0] [0x1b21260]->(nil)

Recommended Problem

Delete a Node in Single Linked List

Delete the first node in a linked list where data == key:
Since every node of the linked list is dynamically allocated using malloc() in C, we need to call free() for freeing memory allocated for the node to be deleted.

- C++
- C
- Java
- Python3
- C#
- Javascript

```
// A complete working C++ program to
// demonstrate deletion in singly
// linked list with class
#include <bits/stdc++.h>
using namespace std;

// A linked list node
class Node {
public:
    int data;
    Node* next;
};

// Given a reference (pointer to pointer)
// to the head of a list and an int,
// inserts a new node on the front of the
// list.
void push(Node** head_ref, int new_data)
{
    Node* new_node = new Node();
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}
```

```
// Given a reference (pointer to pointer)
// to the head of a list and a key, deletes
// the first occurrence of key in linked list
void deleteNode(Node** head_ref, int key)
{

    // Store head node
    Node* temp = *head_ref;
    Node* prev = NULL;

    // If head node itself holds
    // the key to be deleted
    if (temp != NULL && temp->data == key) {

      // Changed head
        *head_ref = temp->next;

      // free old head
        delete temp;
        return;
    }

    // Else Search for the key to be
```

```
    // deleted, keep track of the
    // previous node as we need to
    // change 'prev->next'
    else {
        while (temp != NULL && temp->data != key) {
            prev = temp;
            temp = temp->next;
        }

        // If key was not present in linked list
        if (temp == NULL)
            return;

        // Unlink the node from linked list
        prev->next = temp->next;

        // Free memory
        delete temp;
    }
}

// This function prints contents of
// linked list starting from the
// given node
```

```
void printList(Node* node)
{
    while (node != NULL) {
        cout << node->data << " ";
        node = node->next;
    }
}

// Driver code
int main()
{

    // Start with the empty list
    Node* head = NULL;

    // Add elements in linked list
    push(&head, 7);
    push(&head, 1);
    push(&head, 3);
    push(&head, 2);

    puts("Created Linked List: ");
    printList(head);
```

```
    deleteNode(&head, 1);
    puts("\nLinked List after Deletion of 1: ");

    printList(head);

    return 0;
}

// This code is contributed by ac121102
```

Output

Created Linked List:

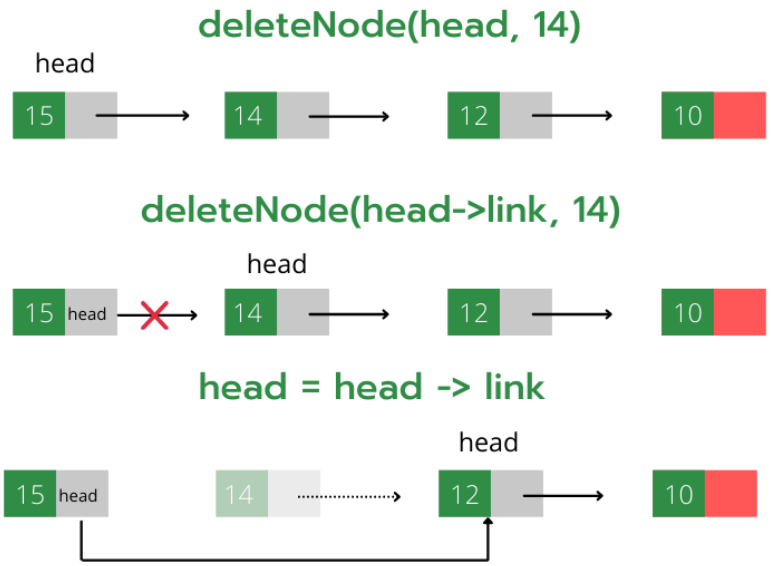
2 3 1 7

Linked List after Deletion of 1:

2 3 7

Time Complexity: O(n)
Auxiliary Space: O(1)
Recursive Method to delete a node from linked list:
To delete a node of a linked list recursively we need to do the following steps:

- We pass node* (node pointer) as a reference to the function (as in node* &head)
- Now since the current node pointer is derived from the previous node's next (which is passed by reference) so now if the value of the current node pointer is changed, the previous next node's value also gets changed which is the required operation while deleting a node (i.e points previous node's next to current node's (containing key) next).
- Find the node containing the given value.
- Store this node to deallocate it later using the free() function.
- Change this node pointer so that it points to its next and by performing this previous node's next also gets properly linked.



Below is the implementation of the above approach.

- C++

```
// C++ program to delete a node in
// singly linked list recursively

#include <bits/stdc++.h>
using namespace std;

struct node {
    int info;
    node* link = NULL;
    node() {}
    node(int a)
        : info(a)
    {
    }
};

// Deletes the node containing 'info'
// part as val and alter the head of
// the linked list (recursive method)
void deleteNode(node*& head, int val)
{

    // Check if list is empty or we
    // reach at the end of the
```

```
    // list.
    if (head == NULL) {
        cout << "Element not present in the list\n";
        return;
    }

    // If current node is the
    // node to be deleted
    if (head->info == val) {
        node* t = head;

        // If it's start of the node head
        // node points to second node
        head = head->link;

        // Else changes previous node's
        // link to current node's link
        delete (t);
        return;
    }
    deleteNode(head->link, val);
}

// Utility function to add a
```

```
// node in the linked list
// Here we are passing head by
// reference thus no need to
// return it to the main function
void push(node*& head, int data)
{
    node* newNode = new node(data);
    newNode->link = head;
    head = newNode;
}

// Utility function to print
// the linked list (recursive
// method)
void print(node* head)
{

    // cout<<endl gets implicitly
    // typecasted to bool value
    // 'true'
    if (head == NULL and cout << endl)
        return;
    cout << head->info << ' ';
    print(head->link);
```

```
}

int main()
{

    // Starting with an empty linked list
    node* head = NULL;

    // Adds new element at the
    // beginning of the list
    push(head, 10);
    push(head, 12);
    push(head, 14);
    push(head, 15);

    // original list
    print(head);

    // Call to delete function
    deleteNode(head, 20);

    // 20 is not present thus no change
    // in the list
    print(head);
```

```
    deleteNode(head, 10);
    print(head);

    deleteNode(head, 14);
    print(head);

    return 0;
}
```

Output

15 14 12 10

Element not present in the list

15 14 12 10

15 14 12

15 12

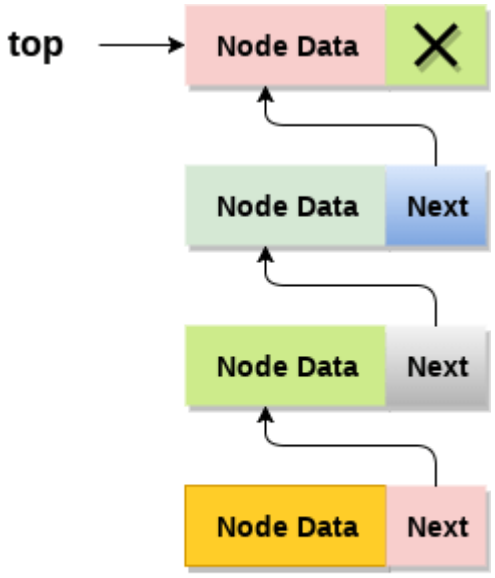
Time Complexity: O(n)
Auxiliary Space: O(n) *(due to recursion call stack)*

Linked representation of Stack and Queue:

Linked list implementation of stack

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.

In linked list implementation of stack, the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in the stack. Stack is said to be overflown if the space left in the memory heap is not enough to create a node.



Stack

The top most node in the stack always contains null in its address field. Lets discuss the way in which, each operation is performed in linked list implementation of stack.

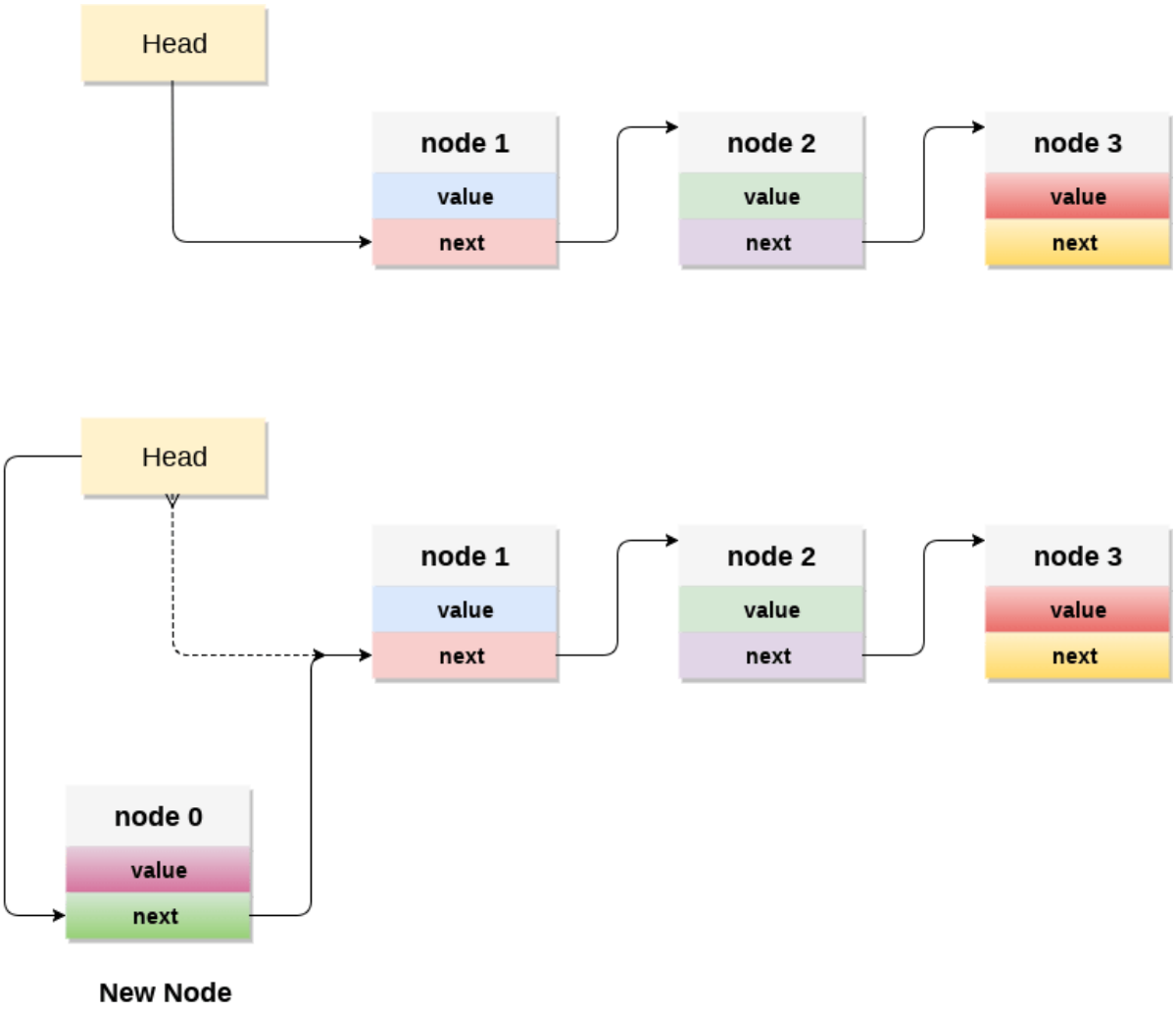
Adding a node to the stack (Push operation)

Adding a node to the stack is referred to as **push** operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation. In order to push an element onto the stack, the following steps are involved.

Java Try Catch

1. Create a node first and allocate memory to it.
2. If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.
3. If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.

4. **Time Complexity : o(1)**



C implementation :

```
1. void push ()
2. {
3.     int val;
4.     struct node *ptr =(struct
   node*)malloc(sizeof(struct node));
5.     if(ptr == NULL)
6.     {
7.         printf("not able to push the element");
8.     }
9.     else
10.         {
```

```
11.             printf("Enter the value");
12.             scanf("%d",&val);
13.             if(head==NULL)
14.             {
15.                 ptr->val = val;
16.                 ptr -> next = NULL;
17.                 head=ptr;
18.             }
19.             else
20.             {
21.                 ptr->val = val;
22.                 ptr->next = head;
23.                 head=ptr;
24.
25.             }
26.             printf("Item pushed");
27.
28.         }
29.     }
```

5. Deleting a node from the stack (POP operation) Deleting a node from the top of stack is referred to as **pop** operation. Deleting a node from the linked list implementation of stack is different from that in the array implementation. In order to pop an element from the stack, we need to follow the following steps :
    1. **Check for the underflow condition:** The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
    2. **Adjust the head pointer accordingly:** In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted

and the node must be freed. The next node of the head node now becomes the head node.

6. **Time Complexity : o(n)**

C implementation

```
1. void pop()
2. {
3.     int item;
4.     struct node *ptr;
5.     if (head == NULL)
6.     {
7.         printf("Underflow");
8.     }
9.     else
10.        {
11.            item = head->val;
12.            ptr = head;
13.            head = head->next;
14.            free(ptr);
15.            printf("Item popped");
16.
17.        }
18.    }
```

7. Display the nodes (Traversing)

Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form of stack. For this purpose, we need to follow the following steps.

1. Copy the head pointer into a temporary pointer.
2. Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

8. **Time Complexity : o(n)**

C Implementation

```
1. void display()
```

```
2. {
3.     int i;
4.     struct node *ptr;
5.     ptr=head;
6.     if(ptr == NULL)
7.     {
8.         printf("Stack is empty\n");
9.     }
10.         else
11.         {
12.             printf("Printing Stack elements \n");
13.             while(ptr!=NULL)
14.             {
15.                 printf("%d\n",ptr->val);
16.                 ptr = ptr->next;
17.             }
18.         }
19.     }
```

9. Menu Driven program in C implementing all the stack operations using linked list :

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. void push();
4. void pop();
5. void display();
6. struct node
7. {
8. int val;
9. struct node *next;
10.     };
11.     struct node *head;
12.
13.     void main ()
```

```
14.     {
15.         int choice=0;
16.         printf("\n**********Stack operations using
    linked list**********\n");
17.         printf("\n----------------------------------------
    ---\n");
18.         while(choice != 4)
19.         {
20.             printf("\n\nChose one from the below
    options...\n");
21.             printf("\n1.Push\n2.Pop\n3.Show\n4.Exit"
    );
22.             printf("\n Enter your choice \n");
23.             scanf("%d",&choice);
24.             switch(choice)
25.             {
26.                 case 1:
27.                 {
28.                     push();
29.                     break;
30.                 }
31.                 case 2:
32.                 {
33.                     pop();
34.                     break;
35.                 }
36.                 case 3:
37.                 {
38.                     display();
39.                     break;
40.                 }
41.                 case 4:
42.                 {
```