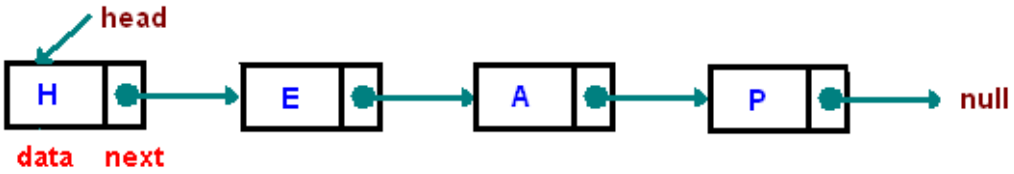
Linked List

A linked list is a set of dynamically allocated linear collection of data elements called nodes, arranged in such a way that each node consist of two parts.

***Info*:** the actual element to be stored in the list. It is also called data field.

Link*:** one or two links (pointer) that point to next and previous node in the list. The last node has a reference to null. ***The NULL value of the next field of the linked list indicates the last node.



The entry point into a linked list is called the **head** of the list. A linked list is a dynamic data structure. The number of nodes in a list is not fixed and can grow and shrink on demand.

Unlike arrays, the nodes of a linked list need not occupy contiguous locations in memory. Instead, they can be stored at discrete memory locations, logically connected with each other through node NEXT.

Memory for each node can be allocated dynamically whenever the need arises.

Operations on linked list:

The basic operations to be performed on the linked list are as follows:

Creation: This operation is used to create a linked list

Insertion: This operation is used to insert a new node in a linked list in a specified position. A new node may be inserted

- ✓ At the beginning of the linked list.
- ✓ At the end of the linked list.
- ✓ At the specified position in a linked list.

Deletion: The deletion operation is used to delete a node from the linked list. A node may be deleted from

- ✓ the beginning of the linked list
- ✓ the end of the linked list
- ✓ the specified position in the linked list.

Traversing: The list traversing is a process of going through all the nodes of the linked list from on end to the other end. The traversing may be either forward or backward.

Searching or find: This operation is used to find an element in a linked list. if the desired element is found then we say operation is successful otherwise unsuccessful.

Concatenation: It is the process of appending second list to the end of the first list.

Defining a Node

```
struct node
        {
                member 1; member 2;
                .....
                struct node *name;
        };
For example,
        struct node
                {
                        int info;
                        struct node *next;
                };
        typedef struct Node NodeType;
```

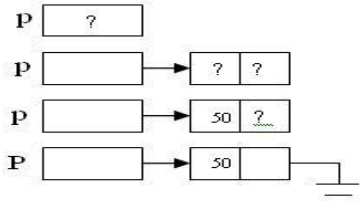
This is a structure of type node. The structure contains two members: a *info* integer member, and a pointer to a structure of the same type (i.e., a pointer to a structure of type node), called next. Therefore this is a ***self-referential*** structure. A structure which contains a reference to itself is called self-referential structure.

Creating a Node:

To create a new node, we use the malloc function to dynamically allocate memory for the new node. After creating the node, we can store new item in the node using a pointer to that node.

The following steps shows the steps required to create a node and storing an item.

```
Nodetype *p;
P=(NodeType*)malloc(sizeof(Nodetype));
p->info=50;
p->next=NULL;
```



Types of Linked List:

Basically we can put linked list into the following Three types:

1. **Singly Linked List**
2. **Doubly Linked List**
3. **Circular Linked List**

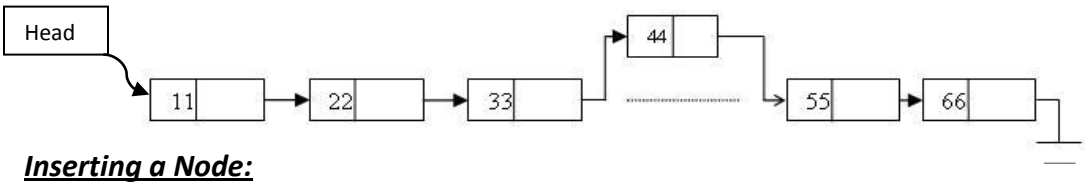
1. ***Singly Linked List:*** A singly linked list is a dynamic data structure which may grow or shrink, and growing and shrinking depends on the operation made. In this type of linked list each node contains two fields one is info field which is used to store the data items and another is link field that is used to point the next node in the list. The last node has a NULL pointer.

   Inserting Nodes:

   To insert an element or a node in a linked list, the following three things have to be done:

   - Allocating a node (Assigning memory to a node)
   - Assigning a data to info field of the node
   - Adjusting a pointer and a new node may be inserted
     - At the beginning of the linked list
     - At the end of the linked list
     - At the specified position in a linked list

   Insertion requires obtaining a new node and changing at most two links



   Inserting a Node:

   Suppose we want to add a new node with data 5 and add it to the list. Then the following changes will be done in the linked list.

   We consider ***head*** as an external pointer. This pointer helps in creating and accessing other nodes in the linked list.

   - The following code defines the structure for linked list and creates a ***head node***. This structure is used in creating a node whenever we need it.

```
struct node
{
        int info;
        struct node *link;
};
struct node *head;
head = NULL;
```

Note: The pointer member variable link points to (i.e. contains address of) a structure variable of the structure type ***node***.

❖ Remember: Whenever ***head==NULL***, the list is empty.

For insertion, we need to do the following three things:

1. ***Allocating a node***: This can be done as,

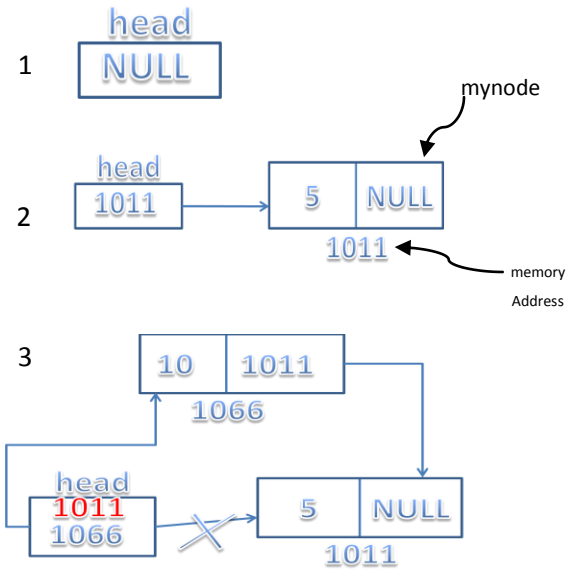
   struct node *mynode;
   mynode=(struct node *)malloc(sizeof(struct node));

2. ***Assigning the data***: This can be done as,

   mynode->info=5;

3. ***Adjusting the pointers***: This depends on where we want to insert the node in the linked list.

A. at the Beginning



Insert at Beginning

```
mynode->info=5;
if (head==NULL)
        {
            mynode->link=NULL;
           head=mynode;
        }
else
        {
            mynode->link=head;
            head=mynode;
        }
```

Figure below explains steps in inserting a node with an element at the beginning position

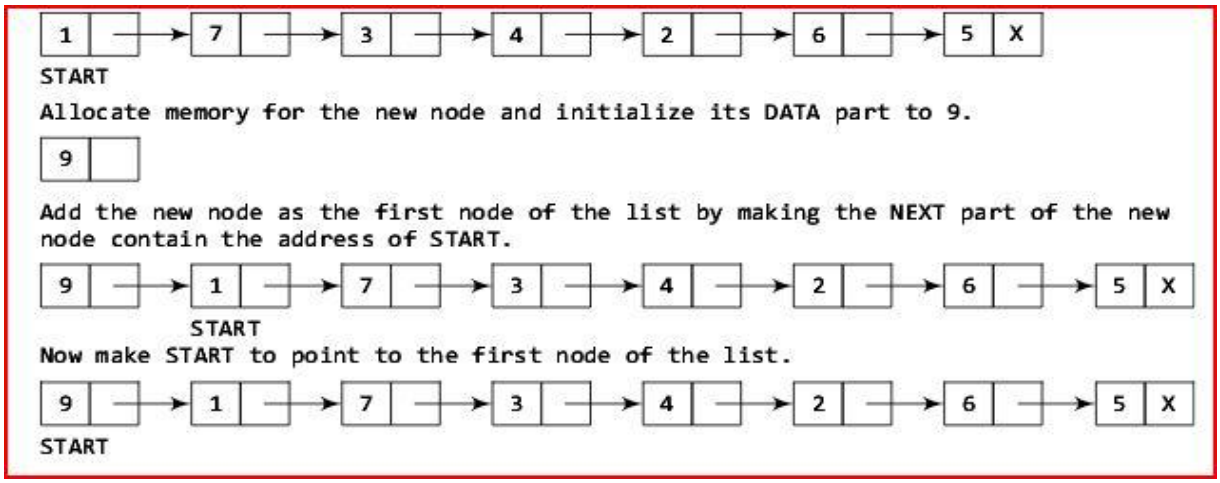
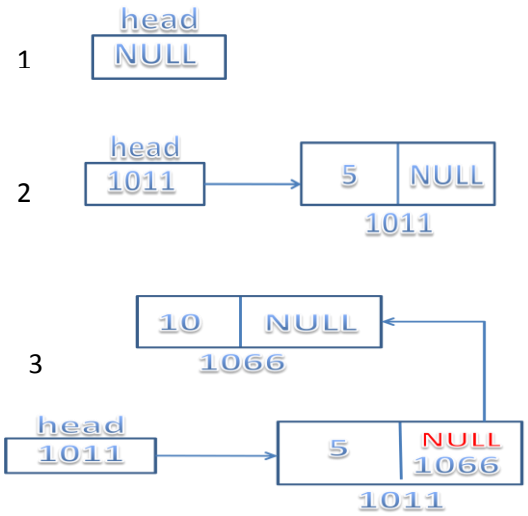


Fig: Inserting a node (element) at the beginning

B. *at the End:*

Suppose we want to add a new node with data 5 as the last node of the list. Then the following changes will be done in the linked list.



Insert at End

```
struct node *temp;
mynode->info=5;
mynode->link=NULL;

if (head==NULL)
        head=mynode;
else
   {
        temp=head;
        while(temp->link!=NULL)
            temp=temp->link
        temp->link=mynode;
   }
```

The code: temp = head; does is

temp	head
1011	1011

Figure below demonstrate steps in inserting a node with a given data element at the last position

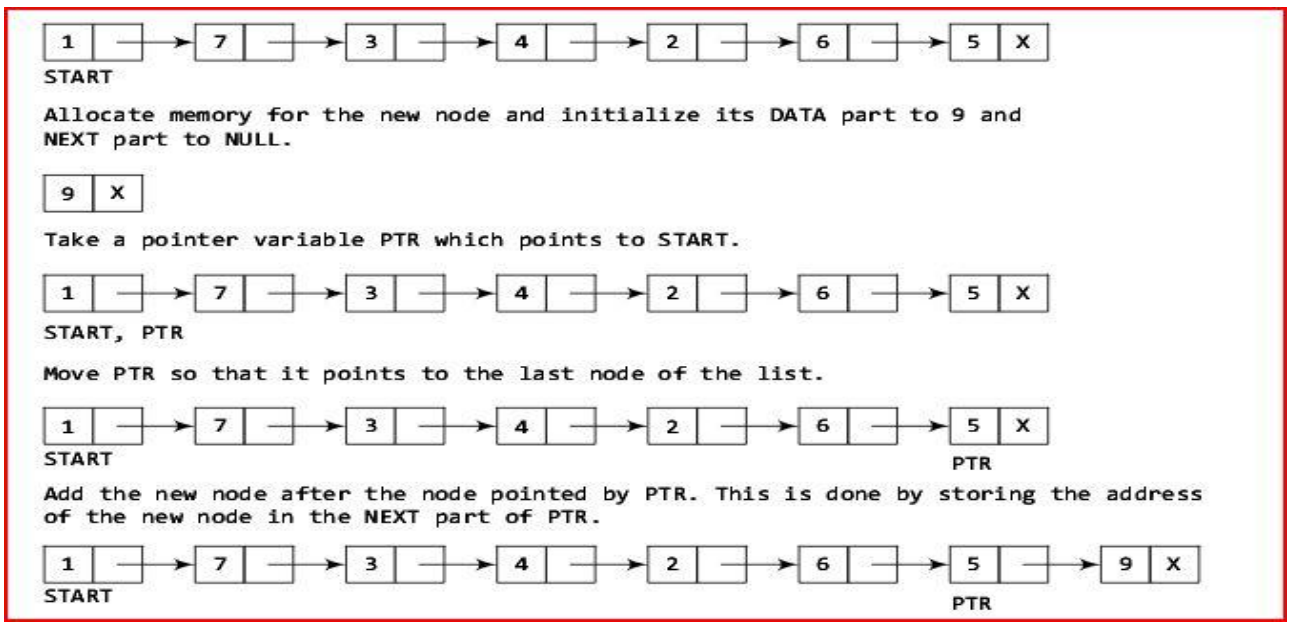
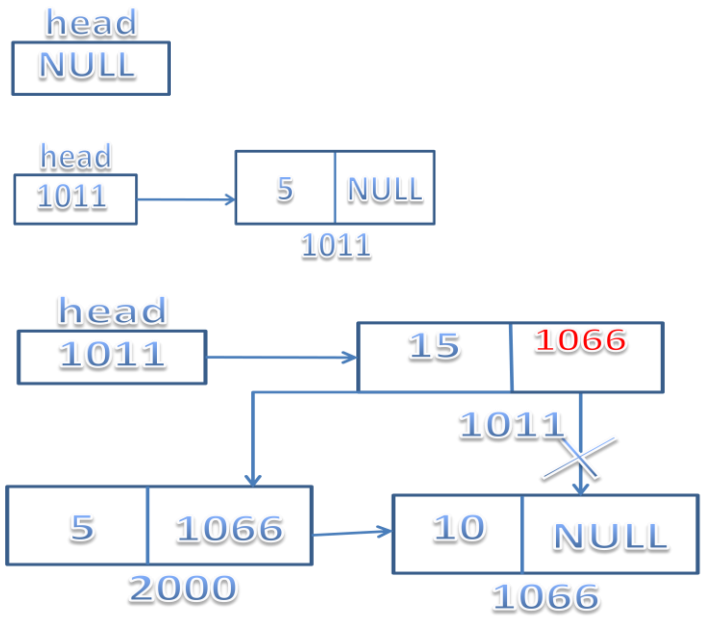


Fig: Inserting an element at the end of the linked list

C. *After a position/given node:*

Suppose we want to add a new node with value 5 after the node containing data 15. Then the following changes will be done in the linked list.



Insert after any position

```
struct nodeType *temp;
if(head==NULL)
{
        mynode->link=NULL;
        head=mynode
}
else
{
        temp=head;
        for(i=0;i<position;i++)
                temp=temp->link;
        mynode->link=temp->link;
        temp->link=mynode;
}
```

In the loop above, we traverse through the linked list to reach to the node that is defined by the position. We need to reach this node because the new node will be inserted after this node. Once we reach this node, we adjust pointers in such a way that new node is inserted after the desired node.

The figure below demonstrates inserting a node with data element after a given node. The process shown inserts a new node with data element 50 after the node containing data element 65. Also, 'p' is a temporary nodeType pointer that is used to traverse the list and at the moment p is pointing to the node with element 65.

```
nodeType *newNode, *p;
newNode =(nodeType *)malloc(sizeof(nodeType));
```

newNode -> info = 50;

newNode ->link = p ->link;

p -> link = newNode;

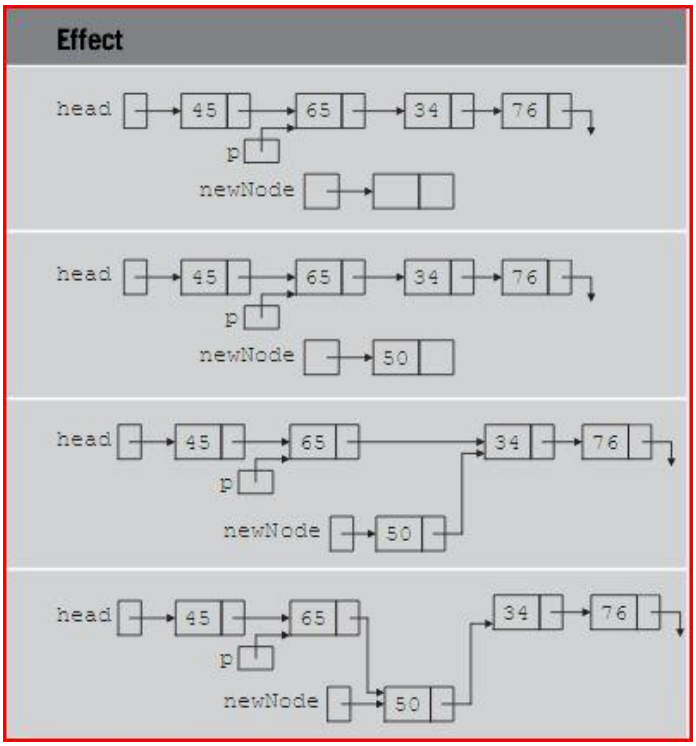


Fig: Inserting a node after a node

Deleting a Node:

The delete operation removes an existing element from the linked list. The following tasks are performed while deleting an existing element:

a) The location of the element is identified
b) The element value is retrieved. In some cases, the element value is simply ignored.
c) The link pointer of the preceding node is reset.

Understand that before deleting a node we must not have an Underflow. It is a condition that occurs when we try to delete a node from a linked list that is empty. This happens when *Head = NULL*

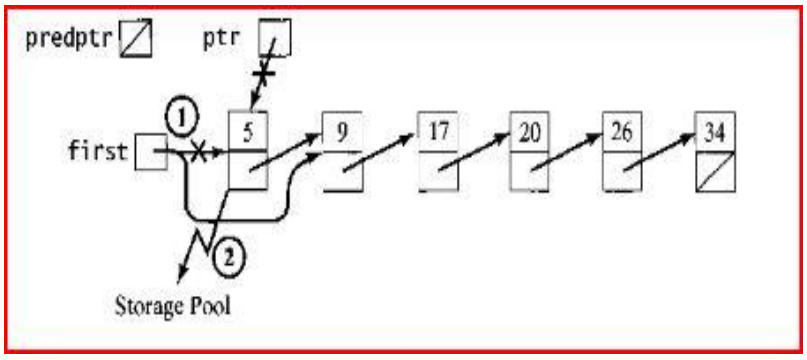
Note that when we delete a node from a linked list, we actually have to ***free*** the memory occupied by that node. The freed memory is then returned to the free pool so that it can be used to store other programs and data.

Depending on the location from where the element is to be deleted, there are three scenarios possible, which are:

(a) Deleting an element from the beginning of the list.
(b) Deleting an element from the end of the list.
(c) Deleting an element somewhere from the middle of the list.

A. Deleting at the beginning:

This case consists of simply resetting *first (head)* to point to the second node in the list and then returning the deleted node to the storage pool of available nodes.

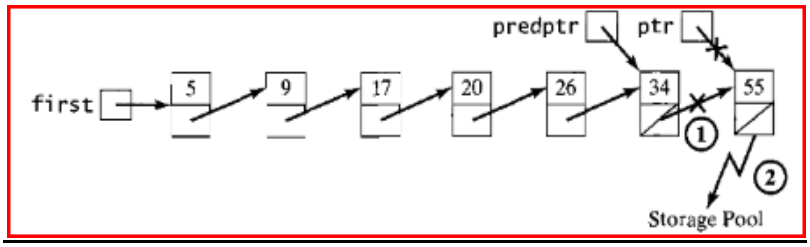


1. Set ***first (head)*** equal to the link part of the node pointed to by ***ptr***
2. Return the node pointed to by ***ptr*** to the storage pool

Deleting at the beginning

```
nodeType *ptr;
int data;
if(head == NULL) {
        printf("Underflow");
        return ;
}
else {
        ptr = first;
        data = ptr->info;
        head = ptr->link;
        free(ptr);
}
return data;
```

B. Deleting at the End:



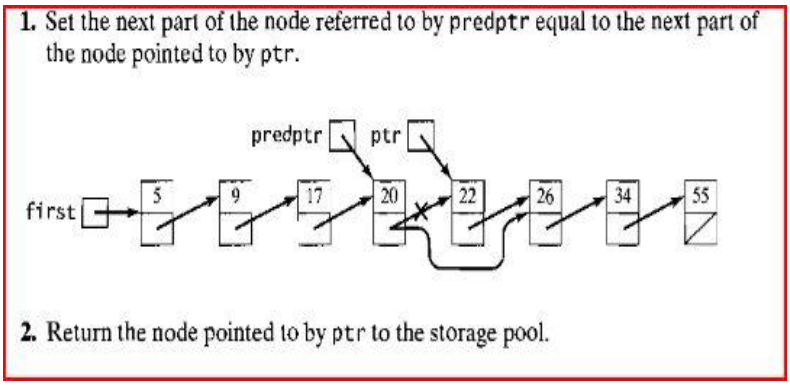
1. Set the ***link*** part of the node referred to by ***predptr*** equal to the ***link*** part of node pointed to by **ptr**.
2. Return the node pointed to by ***ptr*** to the storage pool

Deleting at the end

```
nodeType *ptr, *predptr;
int data;
predptr = head;
if(head == NULL) {
        printf("Underflow");
        return ;
}
else if(predptr->link == NULL){
        data = predptr ->info;
        head = predptr ->link;
}
else {
        while(predptr->link->link !=NULL)
            predptr = predptr->link;
        ptr = predptr->link;
        data = ptr->info;
        predptr->link =ptr->link;
}
free(ptr);
return data;
```

C. Delete after a Node:

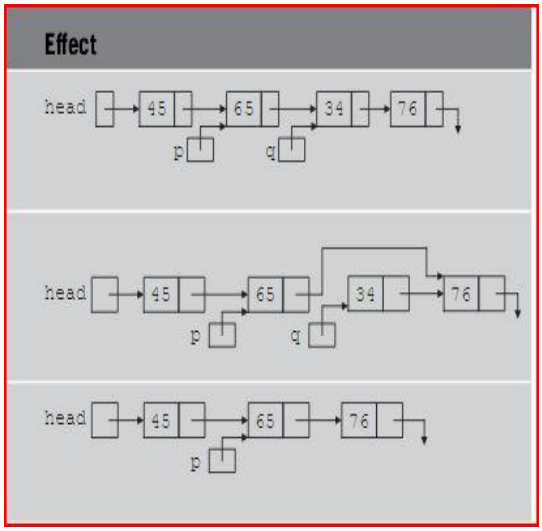
Suppose we want to delete the node containing 22; that ***ptr*** pointed to the node to be deleted; and that the ***predptr*** refers to its predecessor (the node containing 20). We complete this with a bypass operation that sets the ***link*** in the predecessor to point to the successor of the node to be deleted.



Delete after a node

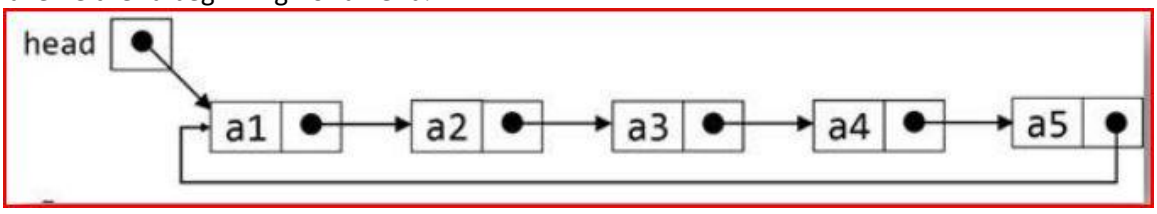
```
nodeType *ptr, *predptr;
int data,i = 1, pos;
predptr = head;
if(head == NULL) {
        printf("Underflow");
        return ;
}
else if (head->link == NULL) {
        data = ptr->info;
        head = ptr->link;
}
else {
        while(i < pos){
            predptr = predptr->link;
             i++;
        }
        ptr = predptr->link;
        data = ptr->info;
        predptr->link =ptr->link;
}
free(ptr);
return data;
```

```
p=head;
while(p->info != 65)
      p= p->link;
q = p->link;

p->link = q->link;

Free(q);
```



2.Circular Linked List:

A linked list in which the last node points to the first node is called a circular linked list. The circular linked lists have neither a beginning nor an end.



The circular linked list is quite similar to linear linked list (singly linked list); the only difference being that the link (next) field of the last node contains the pointer back to the first node. Therefore structure defined for the circular linked list is same as the linear linked list.

```
struct node
{
        int info;
        struct node *link;
};
```

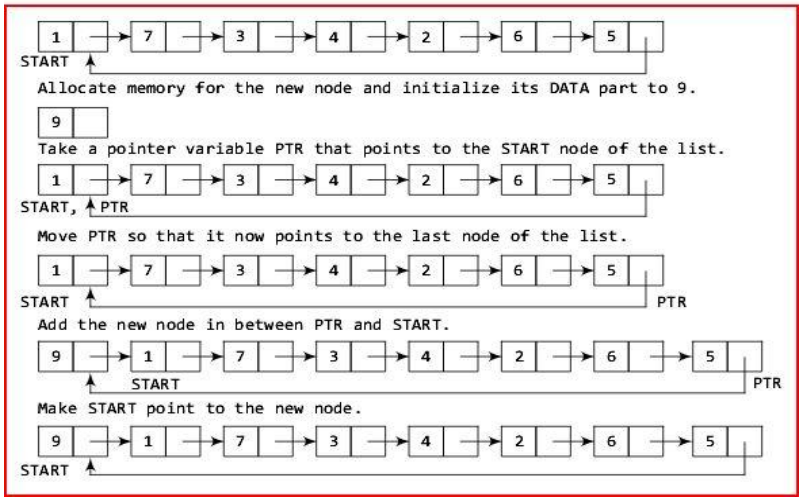
Note that there are no NULL values in the NEXT part of any of the nodes of circular linked list.

A shortcoming of linear linked list (singly linked list) is that with a given pointer to node in linked list we cannot reach any of the nodes that precede the node which the given pointer variable is pointing to. This small drawback is overcome by making the list a circular list.

Inserting a Node in Circular linked list:

- ***At the Beginning:***

  Suppose we want to add a new node with data 9 as the first node of the list. Then the following changes will be done in the linked.



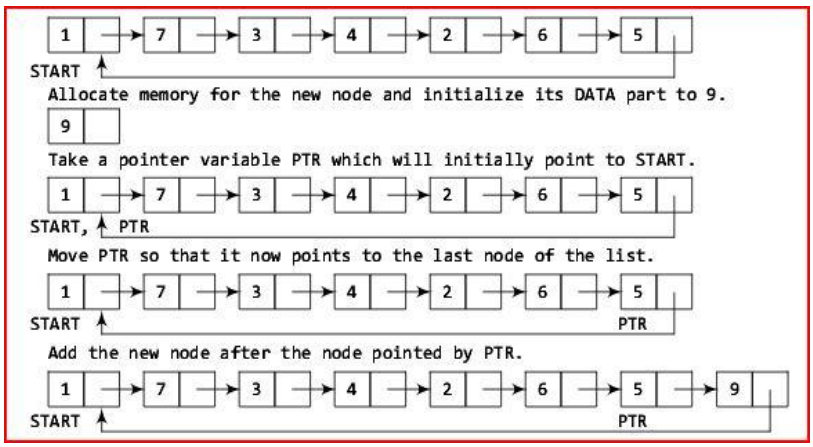
While inserting a node in a circular linked list, we have to use a ***while*** loop to traverse to the last node of the list. Because the last node contains a pointer to START, its NEXT field is updated so that after insertion it points to the new node which will be now known as START

Insert at the beginning

```
mynode = (nodeType*) malloc(sizeof(nodeType));
mynode->info = data;
if(head == NULL)
{
   head = mynode;
   mynode->link = mynode;
}
else
{
   nodeType *ptr = head;
   while(ptr->link ! = head)
         ptr = ptr->link;
   mynode->link = head;
   head = mynode;
   ptr ->link = head.
}
```

- ***At the end:***

  Suppose we want to add a new node with data 9 as the last node of the list. Then the following changes will be done in the linked list.



In the `while` loop, we traverse through the linked list to reach the last node. Once we reach the last node, we change the `NEXT` pointer of the last node to store the address of the new node. Remember that the `NEXT` field of the new node contains the address of the first node which is denoted by `START (head)`.

Note: Inserting after a node is similar to that of the linear linked list.

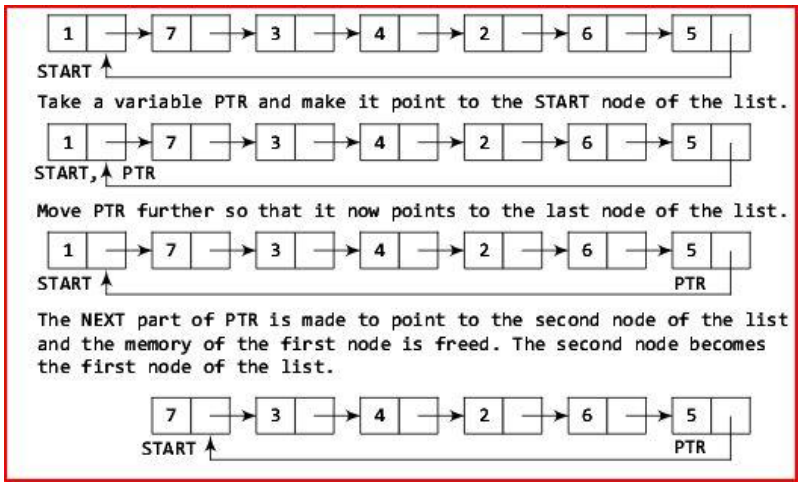
Inserting at the end

```
nodeType *mynode;
mynode = (nodeType*)malloc(sizeof(nodeType));
mynode->info = data;
if(head == NULL)
{
   head = mynode;
   mynode->link = mynode;
}
else
{
   nodeType *ptr = head;
   while(ptr->link ! = head)
         ptr = ptr->link;
   mynode->link = head;
   ptr ->link = mynode.
}
```

Deleting a node in circular linked list:

A. *At the beginning:*

When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list:



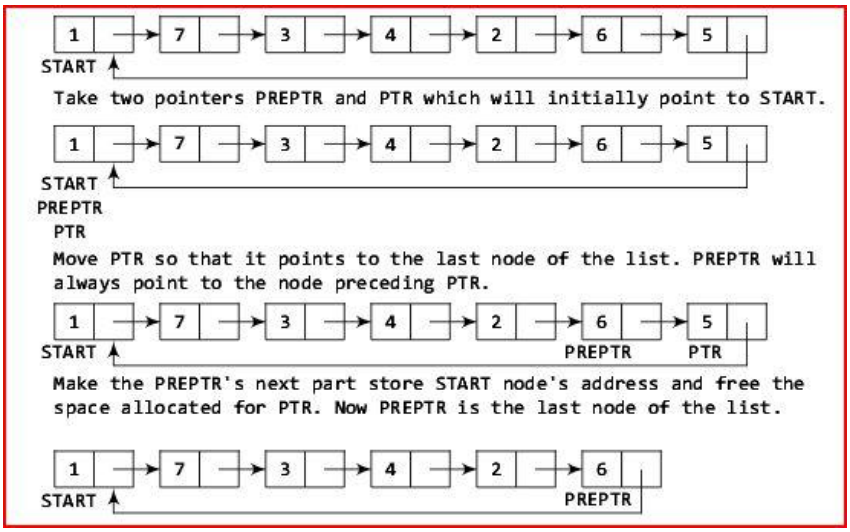
If `START (head) = NULL`, then it signifies that there are no nodes in the list and the control is transferred back to where it is called from. if `START(head)->link = head`, then the list is of single node only. In this case, `START (head)` is set to `NULL` and the only node is freed.
However, if there are nodes in the linked list, then we use a pointer variable `PTR` to traverse the list to ultimately reach the last node. We change the `link` pointer of the last node to point to the second node of the circular linked list. First is node is freed and start (head) is to the second last node of list.

Delete a node at the beginning

```
nodeType *ptr;
ptr = head;
if(ptr== Null)
{
  printf("Underflow");
  return;
}
else if(head->link == head)
{
   data = ptr->info;
   head =NULL;
   free(ptr;)
}
else
{
   while(ptr->link != head)
            Ptr = ptr->link
   ptr->link = head->link;
   data = head->info;
   free(head);
   head =ptr->link;
}
return data;
```

B. *At the End:*

Suppose we want to delete the last node from the linked list, then the following changes will be done in the linked list.



We take a pointer variable `PTR` and initialize it with `START`. That is, `PTR` now points to the first node of the linked list. In the `while` loop, we take another pointer variable `PREPTR` such that `PREPTR` always points to one node before `PTR`. Once we reach the last node and the second last node, we set the next pointer of the second last node to `START`, so that it now becomes the (new) last node of the linked list. The memory of the previous last node is freed and returned to the free pool

Delete a node at the End

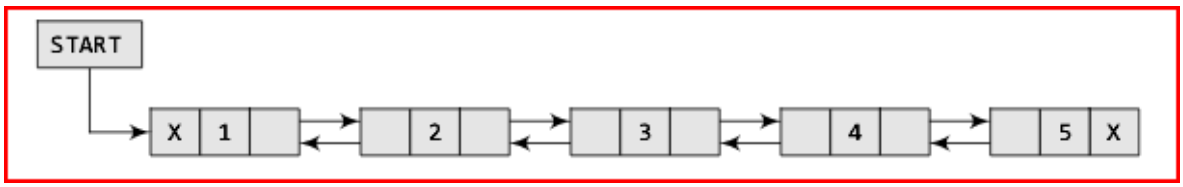
```
nodeType *ptr,predptr;
ptr = head;
if(ptr== Null)
{
  printf("Underflow");
  return;
}
else if(head->link == head)
{
   data = ptr->info;
   head =NULL;
}
else
{
   while(ptr->link != head)
            {
                        predptr = ptr;
                        ptr = ptr->link;
            }
   predptr->link = head;
   data = ptr->info;
}
free(ptr);
return data;
```

Note: Deleting a node from the middle of circular linked list is similar to that of linear linked list.

3. ***Doubly Linked List:***

A doubly linked list is a linked list in which every node has a ***next*** pointer and a ***back*** pointer. In other words, every node contains the address of the next node (except the last node), and every node contains the address of the previous node (except the first node).

A doubly linked list can be traversed in either direction. That is, we can traverse the list starting at the first node or, if a pointer to the last node is given, we can traverse the list starting at the last node.



In C programming, the structure of a doubly linked list can be given as,

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

The PREV field of the first node and the NEXT field of the last node will contain NULL. The PREV field is used to store the address of the preceding node, which enables us to traverse the list in the backward direction.

Doubly linked lists may call for more memory space and maintain rather operations, they, however, are quite efficient in searching elements for they maintain pointers to nodes on both directions.

Inserting a node in Doubly Linked List:

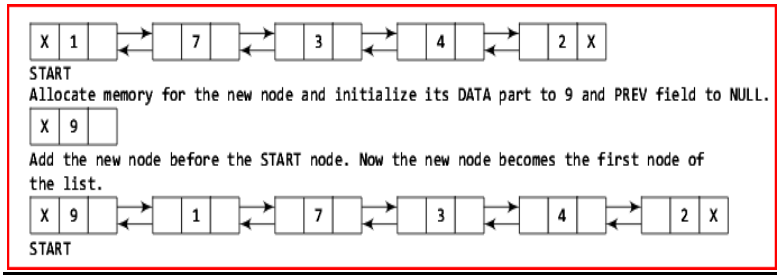
The insertion of a node in the list requires the adjustment of two pointers in certain nodes. As before, we find the place where the new item is supposed to be inserted, create the node, store the new item, and adjust the link fields of the new node and other particular nodes in the list.

The insert operation in a doubly linked list is performed in the same manner as a singly linked list. The only exception is that the additional node pointer PREVIOUS is also required to be updated for the new node at the time of insertion.

Algorithms for the basic doubly linked list operations are similar to those for the singly-linked case, the main difference being the need to set some additional links.

A. *At the Beginning*

Suppose we want to add a new node with data 9 as the first node of the list. Then the following changes will be done in the linked list.



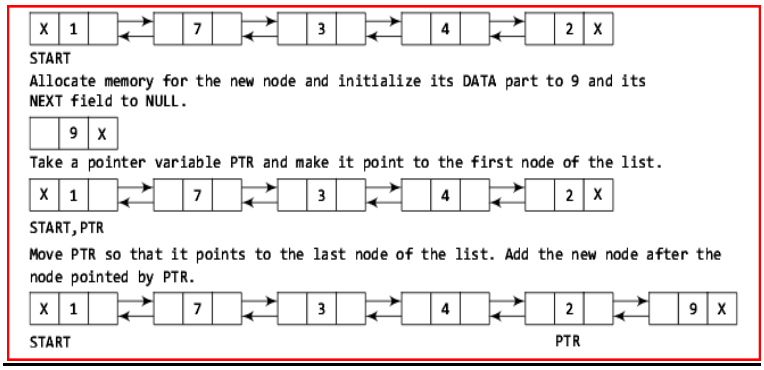
When we get a new node (mynode) we set its `info` part with the given `data value` and the `NEXT` part is initialized with the address of the first node of the list, which is stored in `START(head)`. Now, since the new node is added as the first node of the list, it will now be known as the `START(head)` node, that is, the `START(head)` pointer variable will now hold the address of `mynode`.

Insert at the beginning

```
nodeType *mynode;
mynode = (nodeType*)malloc(sizeof(nodeType));
mynode->info = data;
mynode>prev = NULL;
if(head == NULL){
   mynode->next = NULL;
   head = mynode;
}
else{
   mynode->next =head;
   head->prev = mynode;
   head = mynode;
}
```

B. *At the End:*

Suppose we want to add a new node with data 9 as the last node of the list. Then the following changes will be done in the linked list.



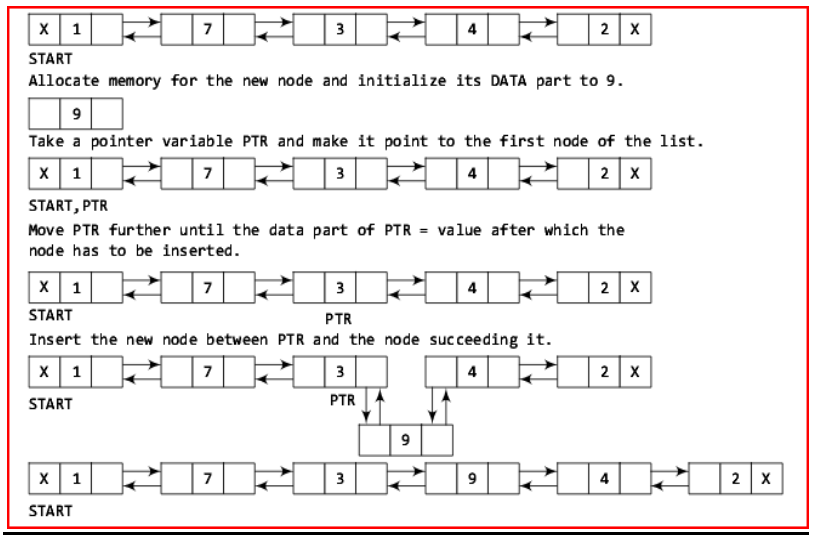
We declare a pointer variable `PTR of type nodeType` and initialize it with `START(head)`. In the `while` loop, we traverse through the linked list to reach the last node. Once we reach the last node, we change the `NEXT` pointer of the last node to store the address of the new node. The `NEXT` field of the new node contains `NULL` which signifies the end of the linked list. The `PREV` field of the `NEW_NODE` will be set so that it points to the node pointed by `PTR` (now the second last node of the list).

Insert at the End

```
nodeType *mynode,*ptr;
mynode = (nodeType*)malloc(sizeof(nodeType));
mynode->info = data;
mynode>next = NULL;
if(head == NULL){
   mynode->next = NULL;
   head = mynode;
}
else{
    ptr = head;
    while(ptr -> next != NULL)
            ptr  = ptr -> next;
     mynode->prev = ptr;
     ptr -> next = mynode;
}
```

C. *After a node:*

Suppose we want to add a new node with value 9 after the node containing 3 (after 3rd position). Then the following changes will be done in the linked list.



We declare a pointer `PTR of type nodeType` and initialize it with `START(head)`. That is, `PTR` now points to the first node of the linked list. In the `while` loop, we traverse through the linked list to reach the node defined by `pos` after which a new node is inserted. We need to reach this node because the new node will be inserted after this node. Once we reach this node, we change the `NEXT` and `PREV` fields in such a way that the new node is inserted after the desired node.

NOTE: It is important that the link changes be done in the correct order.

Insert at the End

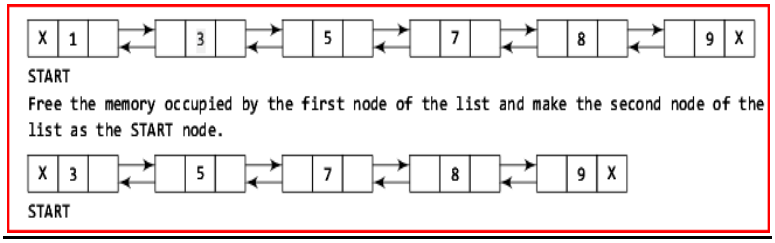
```
nodeType *mynode,*ptr;
mynode = (nodeType*)malloc(sizeof(nodeType));
mynode->info = data;

if(head == NULL)
{
   mynode->next = NULL;
   head = mynode;
}
Else
{
    ptr = head;
    int i = 1, pos;
    while(i < pos)
            ptr  = ptr -> next;
   mynode -> next = ptr->next;
   mynode->prev = ptr;
   ptr->next->prev = mynode;
   ptr->next = mynode;
}
```

Deleting a node from Doubly Linked List:

The delete operation in a doubly linked list is performed in the same manner as a singly linked list. The only exception is that the additional node pointer PREVIOUS of the adjacent node is also required to be updated at the time of deletion.

A. From the Beginning:

If we want to delete a node from the beginning of the list, then the following changes will be done in the linked list.



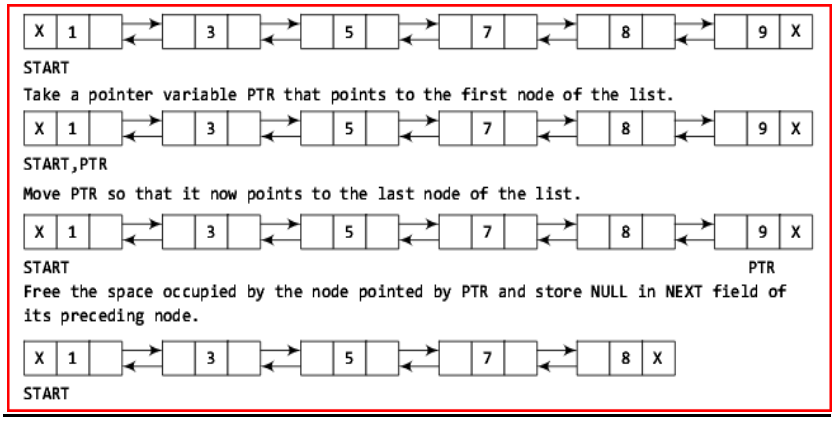
The algorithm here deletes the first node of a doubly linked list. In First, we check if the linked list exists or not. If `START = NULL`, then it signifies that there are no nodes in the list and the control is transferred back to where it was called from. However, if there are nodes in the linked list, then we use a temporary pointer variable `PTR` that is set to point to the first node of the list. For this, we initialize `PTR` with `START` that stores the address of the first node of the list. Now, `START` is made to point to the next node in sequence and finally the memory occupied by `PTR` (initially the first node of the list) is freed and returned to the free pool.

Delete a node at the beginning

```
nodeType *ptr;
ptr = head;
if(ptr== Null){
  printf("Underflow");
  return;
}
else if(ptr->next == NULL && ptr->prev ==NULL){
   data = ptr->info;
   head =NULL;
}
else{
   Ptr->next->prev = NULL;
   head =Ptr->next;
}
free (ptr);
return data
```

B. From the End:

Suppose we want to delete the last node from the linked list, then the following changes will be done in the linked list.



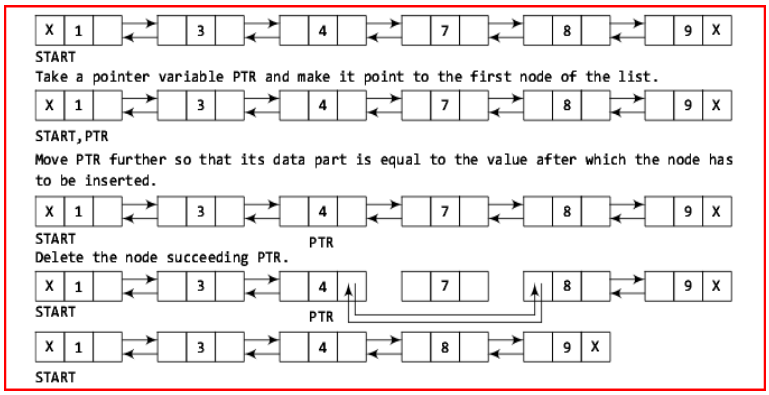
The algorithm here deletes the last node of a doubly linked list. First we declare a pointer variable `PTR` and initialize it with `START`. That is, `PTR` now points to the first node of the linked list. The `while` loop traverses through the list to reach the last node. Once we reach the last node, we can also access the second last node by taking its address from the `PREV` field of the last node. To delete the last node, we simply have to set the next field of second last node to `NULL`, so that it now becomes the (new) last node of the linked list. The memory of the previous last node is freed and returned to the free pool.

Delete a node at the End

```
nodeType *ptr;
ptr = head;
if(ptr== Null){
  printf("Underflow");
  return;
}
else if(ptr->next == NULL && ptr->prev ==NULL){
   data = ptr->info;
   head =NULL;
}
else{
   while(ptr -> next != NULL)
            ptr = ptr -> next;
   Ptr->prev->next = NULL;
   }
free (ptr);
return data;
```

C. Delete from after a node:

Suppose we want to delete the node that succeeds the node which contains data value 4. Then the following changes will be done in the linked list.



The algorithm here deletes a node after a given node of a doubly linked list. We declare here a pointer variable `PTR` and initialize it with `START (head)`. That is, `PTR` now points to the first node of the doubly linked list. The `while` loop traverses through the linked list to reach the given node. Once we reach at the node identified by `pos`, the node succeeding it can be easily accessed by using the address stored in its `NEXT` field. The `NEXT` field of the given node is set to contain the contents in the `NEXT` field of the succeeding node. Finally, the memory of the node succeeding the given node is freed and returned to the free pool.

Delete a node after a node

```
nodeType *ptr;
ptr = head;
if(ptr== Null){
  printf("Underflow");
  return;
}
else if(ptr->next == NULL && ptr->prev ==NULL){
   data = ptr->info;
   head =NULL;
   free(ptr);
}
else{
    int i = 1, pos
    nodeType *temp;
    while(i < pos)
            ptr = ptr -> next;
     temp = Ptr->next;
     data = temp -> info;
     ptr->next = temp->next;
      temp->next->prev = ptr;
   }
free (temp);
return data;
```

Linked List Implementation of Stack:

Stack:

A stack is a list of homogenous elements in which the addition and deletion of elements occurs only at one end, called the top of the stack. Stack is also called a Last In First Out (LIFO) data structure because elements are added and removed from one end only.

We use PUSH operation to add elements into stack and POP operation to remove from stack. The TOP elements always return the top most elements for the stack.

An element can be removed from the stack only if there is something in the stack, and an element can be added to the stack only if there is room.

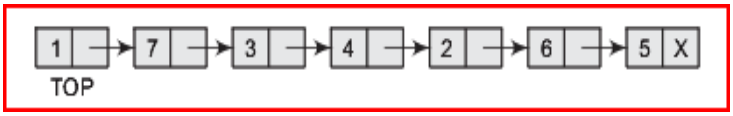
We need at least these six operations to implement a stack successfully, which are described next.

- ***initializeStack***—Initializes the stack to an empty state.
- ***isEmptyStack***—Determines whether the stack is empty. If the stack is empty, it returns the value true; otherwise, it returns the value false.
- ***isFullStack***—Determines whether the stack is full. If the stack is full, it returns the value true; otherwise, it returns the value false.
- ***push***—Adds a new element to the top of the stack. The input to this operation consists of the stack and the new element. Prior to this operation, the stack must exist and must not be full.
- ***top***—Returns the top element of the stack. Prior to this operation, the stack must exist and must not be empty.
- ***pop***—Removes the top element of the stack. Prior to this operation, the stack must exist and must not be empty.

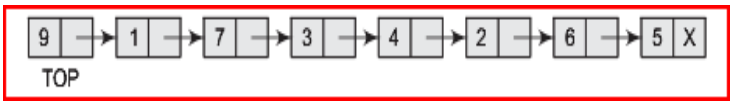
In a linked stack, every node has two parts—one that stores data and another that stores the address of the next node. The HEAD pointer of the linked list is used as TOP. All insertions and deletions are done at the node pointed by TOP. If TOP = NULL, then it indicates that the stack is empty

Push Operation

The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack.



To insert an element with value 9, we first check if stack is empty. If this is the case, then we allocate memory for a new node, store the value in its INFO part and NULL in its NEXT part. The new node will then be called TOP. However, if stack is not empty, then we insert the new node at the beginning of the linked stack and name this new node as TOP. Thus, the updated stack becomes as shown below

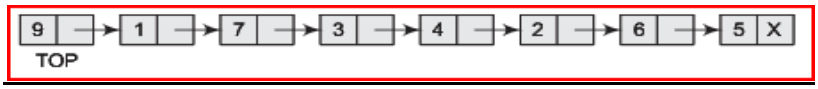


PUSH Operation:

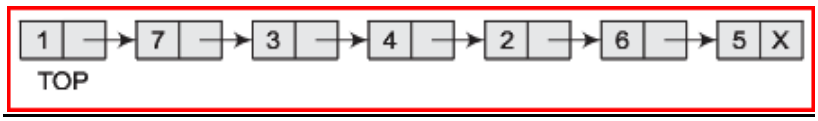
```
nodeType *node;
node = (nodeType *)malloc(sizeof(nodeType));
node -> info = data;
if(isEmptyStack())
{
  node->next = NULL;
  TOP = node;
}
else
{
  node->next = TOP;
  TOP = node;
}
```

POP Operation:

The pop operation is used to remove an element from the stack. An element is removed from the topmost position of the stack.



The pop operation is used to delete the topmost element from a stack. However, before deleting the value, we must first check if stack is empty, because we cannot delete from an empty stack. If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed.
In case TOP!=NULL, then we will delete the node pointed to by TOP, and make TOP point to the second element of the linked stack. Thus, the updated stack becomes as shown below.



PUSH Operation:

```
nodeType *ptr;
if(isEmptyStack())
{
 printf("Underflow");
 return;
}
else
{
 ptr = TOP;
 data = ptr->info;
 TOP = ptr->link;
 free(ptr);
}
return data;
```

```
isEmptyStack()
{
 if(TOP == NULL)
        return true;
 else
        return false;
}
```

```
top()
{
If(!isEmptyStack())
   return TOP->info;
else
  rerutrn -1;
}
```

****

```
/*
 * Linked list implementation of Stack
 * linkedstack.c
 *
 * Copyright 2020 gobinda <gobinda@GOBINDA-PC>
 *
 */
#include <stdio.h>
#include <stdlib.h>

struct node
{
	int info;
	struct node *next;
	};
typedef struct node nodeType;
nodeType *TOP = NULL; // initialization of the linked stack;

void push(int);
int pop();
int top();
int isEmptyStack();
void display();

int main()
{
	int data;
	int choice;
	do
	{
		printf("\n\n 1. PUSH");
		printf("\n 2. POP");
		printf("\n 3. TOP");
		printf("\n 0. exit");
		printf("\n\n Enter your choice: ");
		scanf("%d",&choice);

		switch(choice)
		{
			case 1:
				printf("\n Enter a value to PUSH : ");
				scanf("%d",&data);
				push(data);
				display();
				break;

			case 2:
				data = pop();
				if(data != -1)
					printf("\nThe POPed item is = %d",data);
				else
					printf("\n Stack is EMPTY!!");

				display();
				break;
```

```
                        case 3:
                                data = top();
                                if(data != -1)
                                        printf("the TOP item is : %d",data);
                                else
                                        printf("\n Stack is EMPTY!!");
                                break;

                        case 0:
                                exit(0);

                        default:
                                printf("\nInvalid Input");
                                break;
                        }
                }while(1);

        return 0;
}

//this functions pushes an item into the linked stack
void push(int item)
{
        nodeType *node;
        node = (nodeType *)malloc(sizeof(nodeType));
        node -> info = item;
        if(isEmptyStack())
        {
          node -> next = NULL;
        }
        else
        {
          node -> next = TOP;
        }
        TOP = node;
}

//This function deletes an item from the linked stack
int pop()
{
        nodeType *ptr;
        int data;
        if(isEmptyStack())
        {
         printf("Underflow");
         return -1;
        }
        else
        {
         ptr = TOP;
         data = ptr -> info;
         TOP = ptr -> next;
         free(ptr);
        }
        return data;
}
```

```
//this function displays the items in the linked stack
void display()
{
	nodeType *ptr;
	ptr = TOP;
	if( ptr != NULL)
	{
	printf("\nThe Linked Stack items are: ");
	while(ptr != NULL)
	{
			printf("%d\t",ptr -> info);
			ptr = ptr -> next;
			}
	}
	else
			printf("\nStack is EMPTY!!");
}

//This function checks if the stack is empty or not
int isEmptyStack()
{
	if(TOP == NULL)
	return 1;
	  else
	return 0;
	}

// this function returns the top element
int top()
{
	if(!isEmptyStack())
			return TOP->info;
	else
	return -1;
	}
```

****

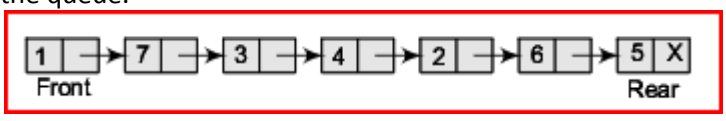
Linked List Implementation of Queue:

The linked implementation of a queue simplifies many of the special cases of the array implementation and, because the memory to store a queue element is allocated dynamically, the queue is never full.
Because elements are added at one end and removed from the other end, we need to know the front of the queue and the rear of the queue. Thus, we need two pointers, `Front` and `Rear`, to maintain the queue.
All insertions will be done at the rear end and all the deletions will be done at the front end. If `FRONT = REAR = NULL`, then it indicates that the queue is empty.

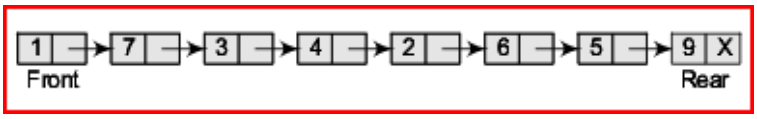
Operations on Linked Queue

Insert (Enqueue) Operation

The `insert` operation is used to insert an element into a queue. The new element is added as the last element of the queue.



To insert an element with, we first check if FRONT=NULL. If the condition holds, then the queue is empty. So, we create a new node, store the value in its INFO part and NULL in its NEXT part. The new node will then be called both FRONT and REAR. However, if FRONT!=NULL, then we will insert the new node at the rear end of the linked queue and name this new node as REAR. Thus, the updated queue becomes as shown below.

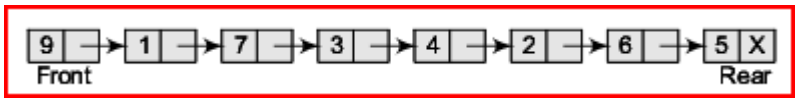


enqueue Operation:

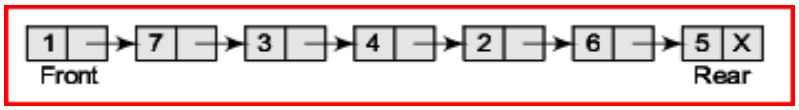
```
nodeType *node;
node = (nodeType *)malloc(sizeof(nodeType));
node -> info = data;
node->next = NULL;
if(isEmptyQueue())
{
  FRONT= node;
  REAR = node;
}
else
{
  REAR->next = node
  REAR = node;
}
```

Delete (Dequeue) Operation:

The delete operation is used to delete first element in a queue, i.e., the element whose address is stored in FRONT. However, before deleting the value, we must first check if FRONT=NULL because if this is the case, then the queue is empty and no more deletions can be done. If an attempt is made to delete a value from a queue that is already empty, an underflow message is printed.



To delete an element, we first check if FRONT=NULL. If the condition is false, then we delete the first node pointed by FRONT. The FRONT will now point to the second element of the linked queue. Thus, the updated queue becomes as shown below.




```
isEmptyQueue()
{
if(FRONT == NULL)
   return 1;
else
   retrun 0;
}
```

Dequeue Operation:

```
nodeType *ptr;
ptr = FRONT;
if(isEmptyQueue())
{
 printf("Underflow");
 return;
}
Else if(ptr -> next == NULL)
{
   data = Ptr -> info;
   FRONT = NULL;
   REAR = NULL
}
else
{
   data = ptr->info;
  FRONT= ptr->link;
 }
free(ptr);
return data;
```

****

```
//Queue_linkedList.c
// Implementation of QUEUE using Singly Linked List
// copyright 2020 Gobinda Subedi.
//This is prepared for B.Ed.ICT 3rd Semester.
#include<stdio.h>
#include<stdlib.h>// This is for exit() and malloc functions

struct LinkedList
{
   int data;
   struct LinkedList *next;
};// this structure creates a node with two parts 1. data, and 2. next.
```

```
typedef struct LinkedList node;//You all should know what typedef does.
node *tail;// Tail (rear)...where new node will be enqueued
node *head;//head (front).. from where a node will be dequeued.

//Function Prototypes;
void Enqueue(int);
int Dequeue();
int Peek();
int isEmpty();
void Display();

int main()
{
   head = NULL;// when linked Queue is empty, top and head should be set to NULL.
   tail = NULL;
   int choice, val;
   do
   {
      printf("\n*************************");
      printf("\n\tMENU");
      printf("\n*************************\n");
      printf("\n 1. TO ENQUEUE");
      printf("\n 2. TO DEQUEUE");
      printf("\n 3. TO PEEK");
      printf("\n 4. TO DISPLAY");
      printf("\n 0. TO EXIT");

      printf("\n\n Enter your choice:  ");
      scanf("%d",&choice);

      switch(choice)
      {
      case 1:
         printf("\n Enter the value to Enqueue: ");
         scanf("%d",&val);// value to push into the stack.
         Enqueue(val);
         Display();
         break;

      case 2:
         val = Dequeue();//Val stores the value after execution of Pop() funciton.
         if(val != -1)
         {
            printf("\n The DEQUEUED item = %d \n",val);
            Display();
         }
         break;

      case 3:
         val = Peek();
         if(val != -1)
            printf("\n The FRONT item = %d \n",val);
         else
            printf("\n Empty! Nothing to Show!\n");
         break;

      case 4:
         Display();
         break;

      case 0:
         exit(0);// Dumps releases all the resource used and Exits the program.
```

```
        default:
          printf("\n Invalid Choice ");
          break;
        }

    }while(1);//Menu displays in loop until user exits pressing 0.
return 0;
}

void Enqueue(int val)
{
    node *mynode;
    mynode = malloc(sizeof(node));
    if(mynode == NULL) //Checking if memory is assigned to new mynode
    {
        printf("Heap Overflow\n");
        exit(1);//If memory for new node could not be assigned, So, we exit.
    }

    mynode -> data = val;
    mynode -> next =  NULL;

    if(isEmpty())
    {
        head = mynode;
        tail = mynode;
    }
    else
    {
        tail -> next = mynode;
        tail = mynode;
    }
}

int Dequeue()
{
    int val;
    node *ptr;

    if(isEmpty())// Checking if there is anything to Dequeue.
    {
        printf("\n Queue Underflow\n");
        return -1;
    }

    else//if there are some nodes in the linked Queue, this part of code is executed.
    {
        if(head == tail)//checking if there is single node in the linked Queue.
        {
                                    ptr = head;
            val = ptr -> data;
            head = NULL;//if there is single node in the Queue, we need to arrange the pointer this way to dequeu.
            tail = NULL;
        }
        else// if there are multiple nodes in the linked queue, this part gets executed
        {
                                    ptr = head;
            val = ptr -> data;
            head  = ptr -> next;
        }
    }
    free(ptr);
```

```
    return val;
}

int Peek()// We need to see what is at the head (front) node
{
    if(head == NULL)
        return -1;
    else
        return (head -> data);
}

int isEmpty()
{
    if(head == NULL && tail == NULL)
        return 1;
    else
        return 0;
}

void Display()
{
    if(head != NULL)
    {
        node *tmp;//tmp is used for parsing the linked list
        tmp = head;
        printf("\n Values in the Linked Queue are:\t");
        while(tmp != NULL)//linked list parsed until the last node whose next part points to the null.
        {
            printf("%d\t",tmp -> data);//print the value at the current node.
            tmp = tmp -> next;// move to next node.
        }
    }
    else
        printf("\n Linked Queue is Empty");
}
```

****