DATA STRUCTURES

UNIT 2 & 3

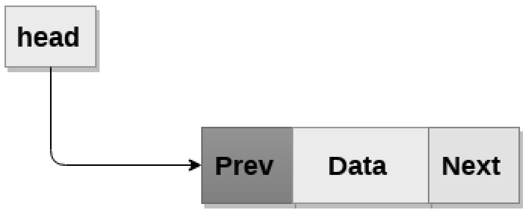
2 - MARKS

1. What is Doubly Linked List?

Doubly linked list is a type of linked list in which a node contains a pointer to the previous as well as a pointer to the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts:

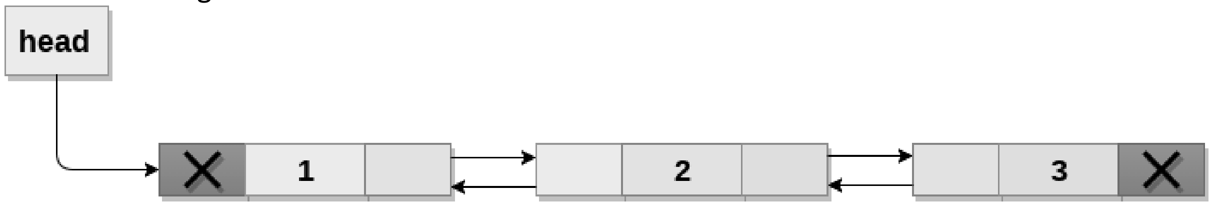
1. Pointer to the previous node.
2. Data.
3. Pointer to the next node.

A sample node in a doubly linked list is shown in the figure.



Node

A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the figure.



Doubly Linked List

11 - MARKS

1. Explain about queue operations in detail with algorithm.

A Queue is an ordered collection of elements in which insertions are made at one end and deletions are made at the other end.

The end at which insertions are made is referred to as the rear end and the end from which deletions are made is referred to as the front end.

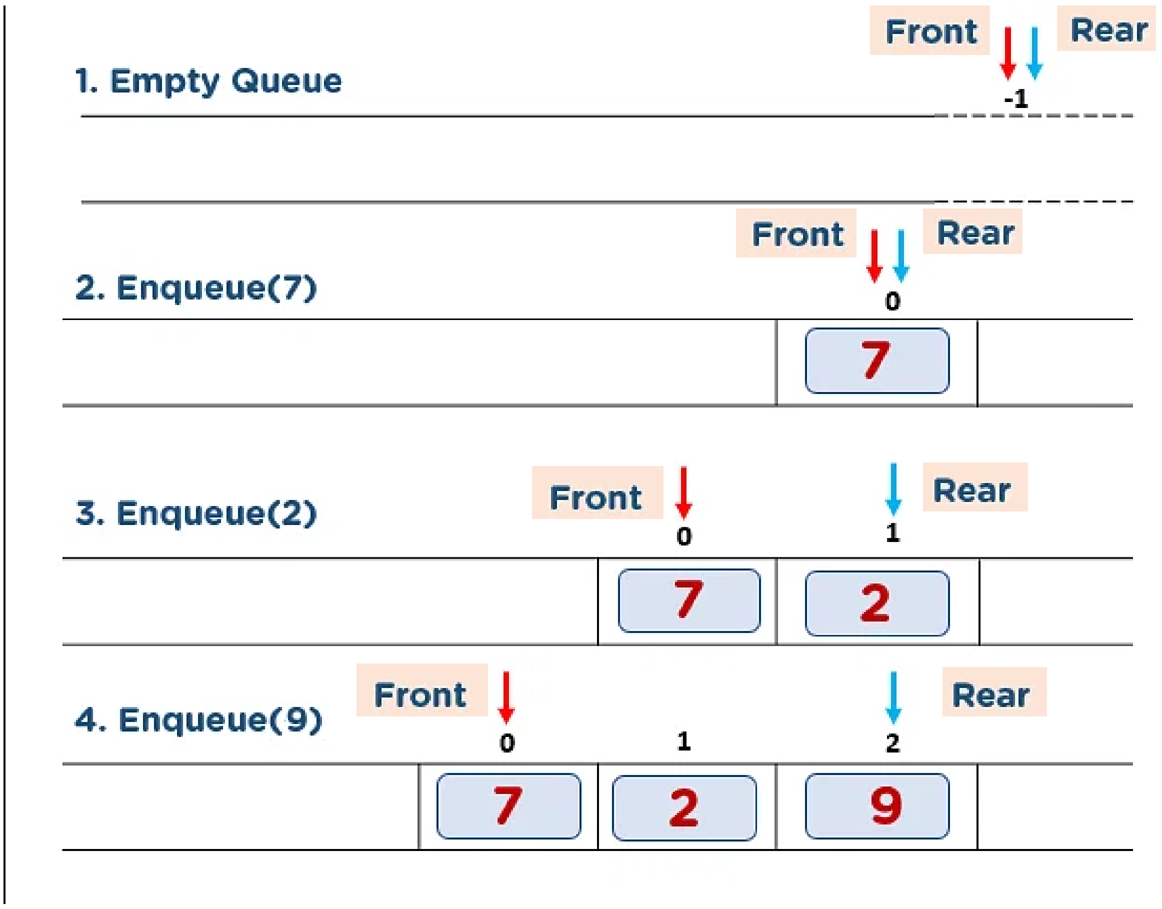
The first element inserted will be the first element to be removed. So, a queue is referred to as First- In- First-Out (FIFO).

Operations in Queue:

1. Enqueue operation
2. Dequeue operation
3. Peek operation
4. Empty queue operation

1. Enqueue operation:

Enqueue is an operation used to add or insert a new element into a queue at the rear end. When implementation the Enqueue operation overflow condition of the queue is to be checked.



Algorithm:

```
Enqueue()

Step 1: if (REAR==QSIZE-1) then
                Print, "queue is overflow on enqueue"
                Return
        End of if structure

Step 2: if (FRONT==-1 && REAR==-1) then
                Set FRONT=0 and REAR =0
        Else
                REAR=REAR+2
        End of if structure
```

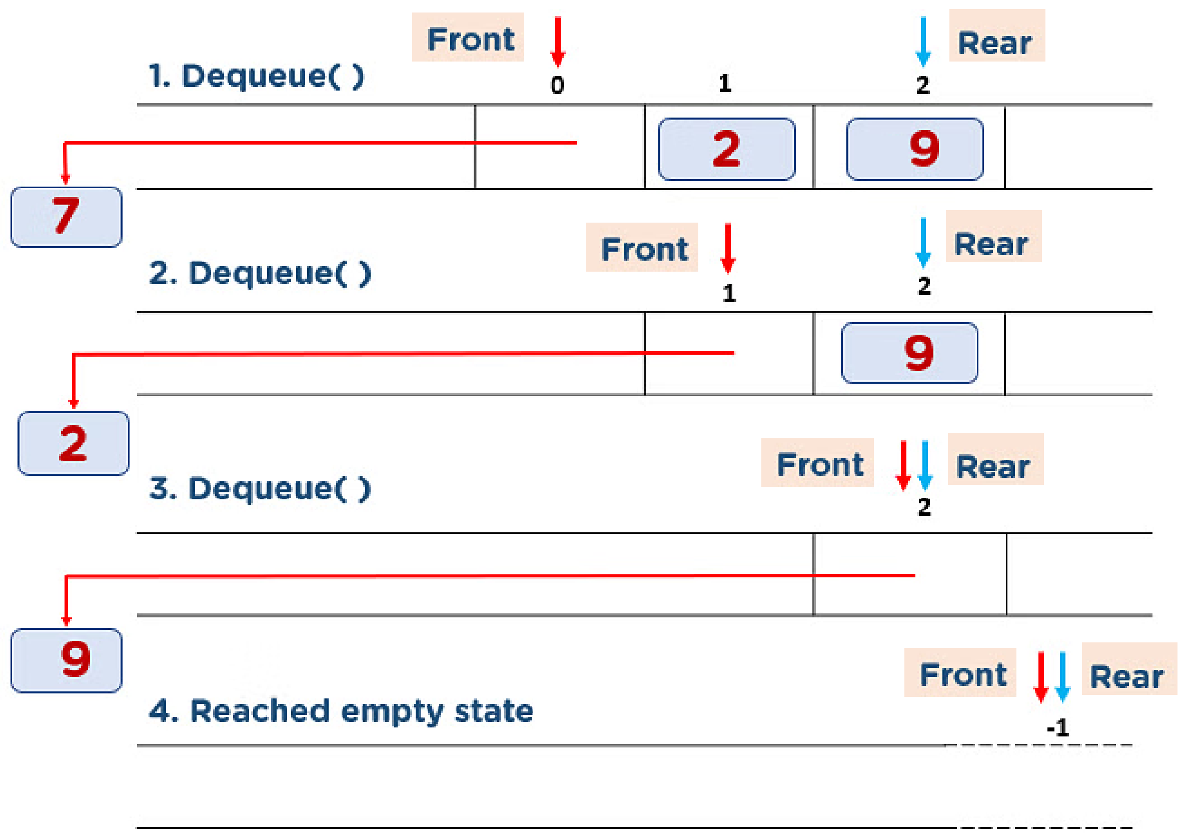
Step 3: print, enter the element

Step 4: read, QUEUE(REAR)

End ENQUEUE()

2. Dequeue operation:

Dequeue is an operation used to remove or delete an element from the front end of the queue. When implementing the dequeue operation, underflow condition of a queue is to be checked.

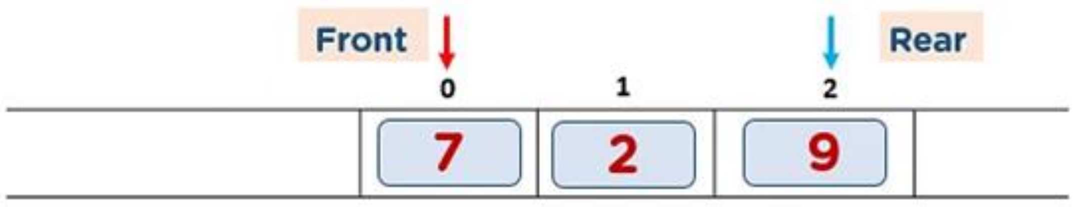


Algorithm:

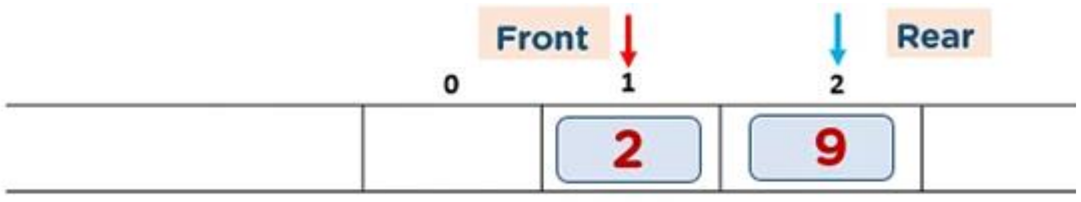
```
DEQUEUE()

Step 1: if (FRONT==-1 && REAR==-1) then
                Print, "queue is underflow on enqueue"
                Return
        End of if structure

Step 2: print, "the dequeued value is, QUEUE(FRONT)
        If (FRONT==REAR) then
                Set FRONT =-1 and REAR =-1
        Else
                FRONT= FRONT+1
        End of if structure

End Dequeue()
```

3. PEEK operation:

Peek operation is used to display the element from the front of the queue, pointed by the FRONT pointer without actually removing it.



PEEK is 7



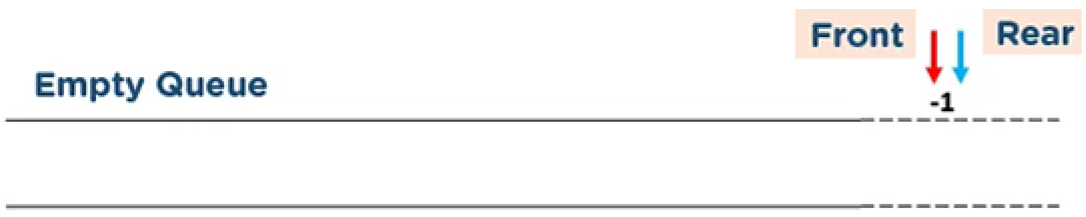
PEEK is 2

Algorithm:

```
PEEK()

Step 1: if (FRONT==-1 && REAR==-1) then
                Print, "queue is Empty"
        Return
        End of if structure

Step 2: print, "the top most value is," QUEUE[FRONT]

End PEEK()
```

4. EMPTY queue:

If a queue contains no elements, it is referred as an empty queue.



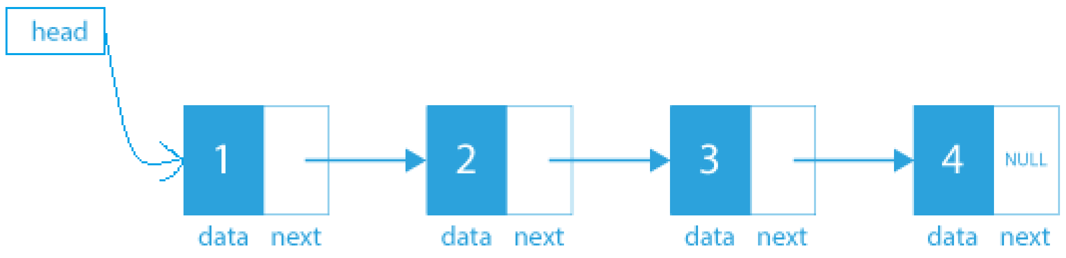
Algorithm:

```
ISEMPTY()

Step 1: if (FRONT==-1 && REAR==-1) then
                Print, "queue is Empty"
        Else
        Print," queue is not empty"
    End of if structure

End ISEMPTY()
```

2. What are the various operations that can be performed on Linked List? Explain it with suitable example.

A linked list is a collection of "nodes" connected together with the links called the pointer. These nodes consist of the data to be stored and a pointer to the address of the next node within the linked list.

In the case of arrays, the size is limited to the definition, but in linked lists, there is no defined size. Any amount of data can be stored in it and can be deleted from it. So, it is dynamic.



Each node consists of two fields,

- information field called DATA that holds the actual element on the list and
- a pointer pointing to next element of the list called LINK or NEXT, ie. It holds the address of the next element or node.

The name of a typical element is denoted by NODE. Pictorially, the node structure is given as follows:

NODE		
	DATA	LINK

OPERATIONS ON LINKED LIST:

1. Insertion : This operation is used to add an element to the linked list.

Insertion of a node of a linked list can be on three positions
- Insertion at the beginning
- Insertion at the end and
- Insertion in the middle of the list.

2. Deletion : Deletion operations are used to remove an element from the beginning of the linked list. Deletion of a node in the linked list can be done in three ways
- Deletion at the beginning
- Deletion at the end and
- Deleting at a specific position.

3. Search : A search operation is used to search an element using the given key. The search operation is done to find a particular element in the linked list. If the element is found in any location, then it returns. Else, it will return null.

4. Display : Display operation is used to display the linked list display() will display the nodes present in the list.

INSERTION IN A LIST:

Inserting a new item, say ‘x’ into the list has three situations:

1. Insertion at the beginning or front of the list
2. Insertion at the end of the list
3. Insertion in the middle of the list

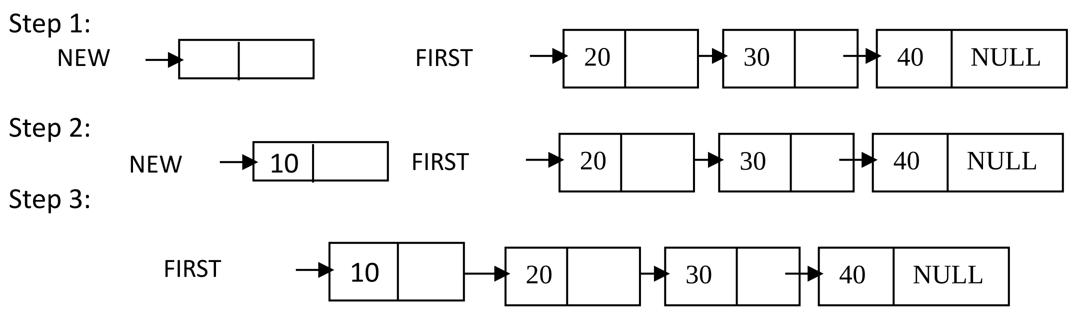
(i) INSERTION AT THE BEGINNING :

Steps for placing the new item at the beginning of a linked list:

1. Obtain space for new node
2. Assign data to the item field of new node
3. Set the next field of the new node to point to the start of the list
4. Change the head pointer to point to the new node.

ALGORITHM :

```
Function INSERT(X, FIRST)
    Variables used:
        X  ← New element to be inserted
    FIRST ← Pointer to the first element whose node contains
                INFO and LINK fields.
    AVAIL← Pointer to the top element of the availability
     NEW  ← Temporary pointer variable.

     1.  [Underflow?]
            If AVAIL = NULL
            Then Write(‘AVAILABILITY UNDERFLOW’)
                    Return(FIRST)
     2.  [Obtain address of next free node]
            NEW ← AVAIL
     3.  [Remove free node from availability]
            AVAIL ← LINK(AVAIL)
     4.  [Initialize fields of new node and its link to the list]
            INFO(NEW) ← X
            LINK(NEW) ← FIRST

     5.  [Return address of new node]
            Return(NEW)
```



(ii) INSERTION AT THE END :

Steps for inserting an item at the end of the list:

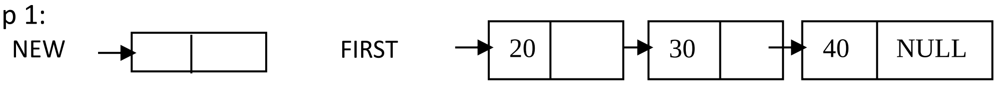
1.Set space for new node x
2.Assign value to the item field of x
3.Set the next field of x to NULL
4.Set the next field of N2 to point to x

ALGORITHM :

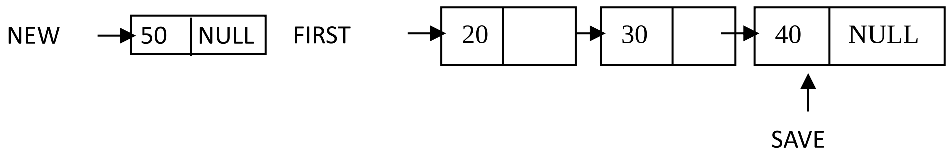
Function INSEND(X,FIRST)
Variables used:

```
          X← new element
      FIRST← Pointer to the first element whose node contains
              INFO and LINK fields.
     AVAIL← pointer to the top element of the availability stack
     NEW,SAVE← Temporary pointer variables
  1.   [Underflow?]
         If AVAIL = NULL
         Then  Write('AVAILABILITY UNDERFLOW')
               Return(FIRST)
  2.   [Obtain address of next free node]
         NEW ← AVAIL
  3.   [Remove free node from availability stack]
         AVAIL ← LINK(AVAIL)
  4.   [Initialize fields of new node]
         INFO(NEW) ← X
         LINK(NEW) ← NULL
  5.   [Is the list empty?]
         If FIRST  = NULL
         Then  Return(NEW)
  6.   [Initiate search for the last node]
         SAVE ← FIRST
  7.   [Search for end of list]
          Repeat while LINK(SAVE) ≠ NULL
                SAVE ← LINK(SAVE)
  8.   [Set LINK field of last node to NEW]
         LINK(SAVE) ← NEW
  9.   [Return first node pointer]
         Return(FIRST)
```

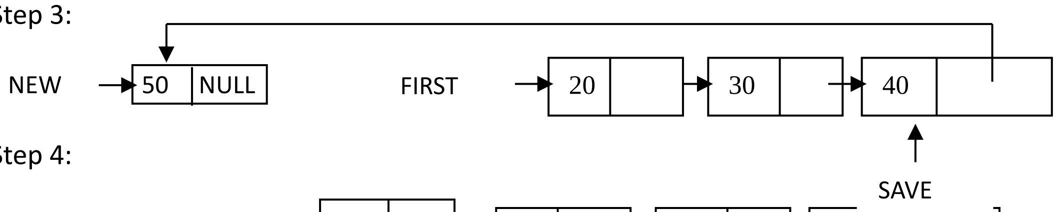
Step 1:



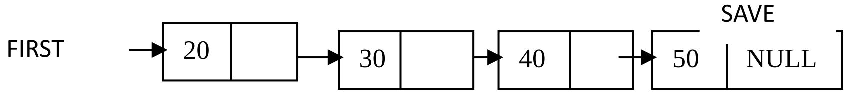
Step 2:



Step 3:



Step 4:



Note that no data is physically moved, as we had seen in array implementation. Only the pointers are readjusted.

(iii) INSERTION AT THE MIDDLE :

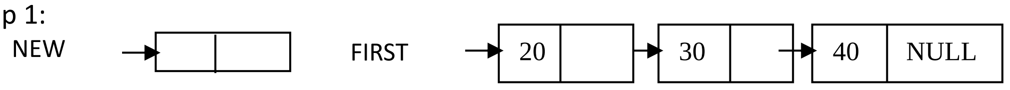
Steps for inserting the new node x between the two existing nodes, say N1 and N2(or in order):

1. Set space for new node x
2. Assign value to the item field of x
3. Search for predecessor node n1 of x
4. Set the next field of x to point to link of node n1(node N2)
5. Set the next field of N1 to point to x.

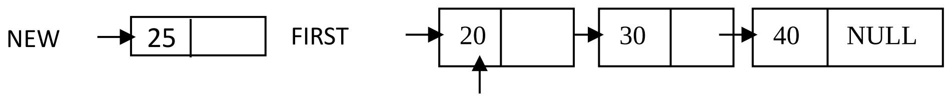
ALGORITHM

```
Function INSORD(X, FIRST)
    Variables used:
      X← new element
  FIRST← Pointer to the first element whose node contains
         INFO and LINK fields.
AVAIL← pointer to the top element of the availability
NEW,SAVE← Temporary pointer variables
1.  [Underflow?]
        If AVAIL = NULL
        Then  Write('AVAILABILITY UNDERFLOW')
              Return(FIRST)
2.  [Obtain address of next free node]
        NEW ← AVAIL
3.  [Remove free node from availability]
        AVAIL ← LINK(AVAIL)
4.  [Copy information contents into new node]
        INFO(NEW) ← X
5.  [Is the list empty?]
        If FIRST = NULL
        Then LINK(NEW) ← FIRST
6.  [Does the new node precede all others in the list?]
        If INFO(NEW) ≤ INFO(FIRST)
        Then LINK(NEW) ← FIRST
             Return(NEW)
7.  [Initialize temporary pointer]
        SAVE ← FIRST
8.  [Search for predecessor of new node]
    Repeat while LINK(SAVE) ≠ NULL and INFO(LINK(SAVE))≤ INFO(NEW)
              SAVE ← LINK(SAVE)
9.  [Set link fields of new node and its predecessor]
         LINK(NEW) ← LINK(SAVE)
         LINK(SAVE)← NEW
10. [Return first node pointer]
         Return(FIRST)
```

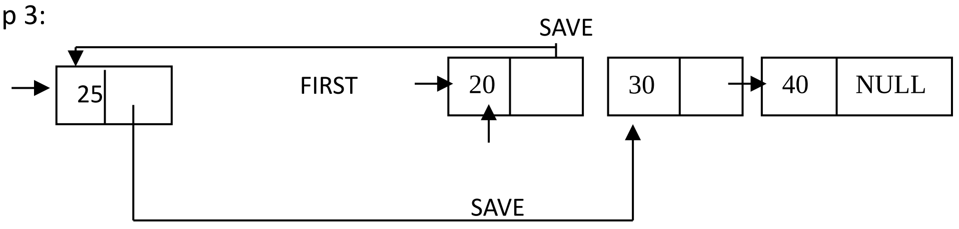
Step 1:



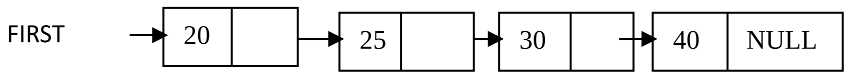
Step 2:



Step 3:



Step 4:



DELETING AN ITEM FROM A LIST:

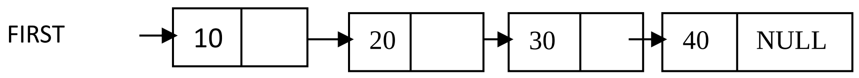
Deleting a node from the list requires only one pointer value to be changed, there we have situations:

1.Deleting the first item
2.Deleting the last item
3.Deleting between two nodes in the middle of the list.

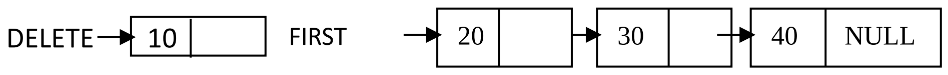
(i) DELETION AT THE BEGINNING :

Steps for deleting the first item:
1.If the element x to be deleted is at first store next field of x in some other variable y.
2.Free the space occupied by x
3.Change the head pointer to point to the address in y.

Step 1: Delete 10



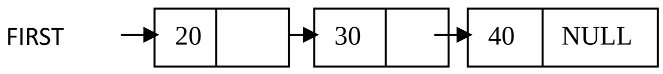
Step 2:



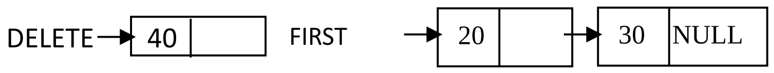
(ii) DELETION AT THE END :

Steps for deleting the last item:
1.Set the next field of the node previous to the node x which is to be deleted as NULL
2.Free the space occupied by x

Step 1: DELETE 40

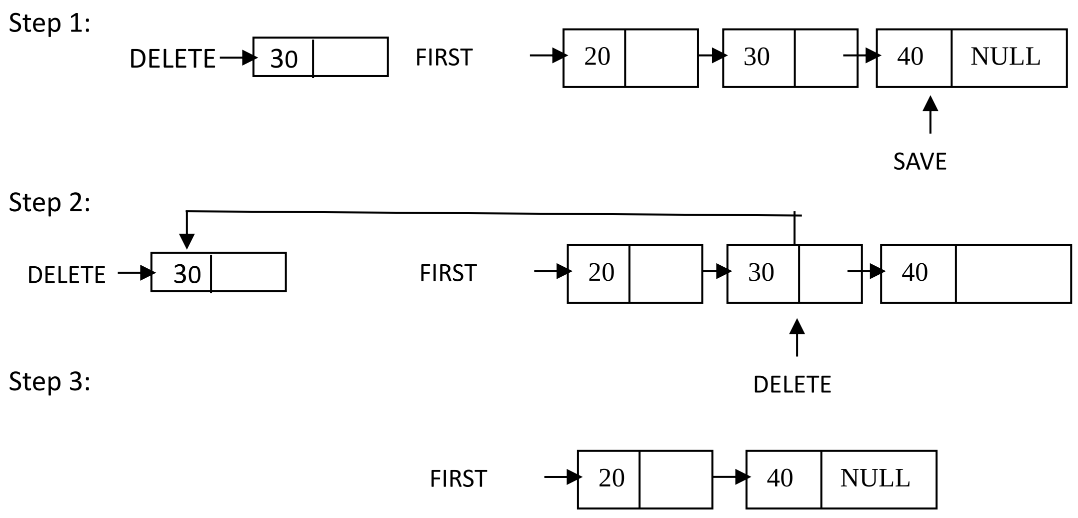


Step 2:



(iii) DELETION AT THE MIDDLE :

Steps for deleting x between two nodes N1 and N2 in the middle of the list:
1.Set the next field of the node N1 previous to x to point to the successor field N2 of the node x.
2.Free the space occupied by x.



ALGORITHM

Procedure DELETE(X, FIRST)
Variables used:

```
      X  ← New element to be inserted
   FIRST ← Pointer to the first element whose node contains
              INFO and LINK fields.
    TEMP← To find the desired node
    PRED← keeps track of the predecessor of TEMP
    1.  [Empty list?]
           If FIRST = NULL
           Then Write('UNDERFLOW")
                 Return
    2.  [Initialize search for X]
           TEMP ← FIRST
    3.  [Find X]
            Repeat thru step 5 while TEMP ≠ X and LINK(TEMP) ≠ NULL
    4.  [Update predecessor marker]
            PRED ← TEMP
    5.  [Move to next node]
            TEMP ← LINK(TEMP)
    6.  [End of the list]
            If TEMP ≠ X
            Then Write('NODE NOT FOUND")
                  Return
    7.  [Delete X]
            If X = FIRST      (Is X the first node?)
            Then FIRST ← LINK(FIRST)
            Else  LINK(PRED) ← LINK(X)
    8.  [Return node to availability area]
            LINK(X) ← AVAIL
```

```
AVAIL ← X
Return
```

3. Describe in detail about doubly linked list operations.

Following are the basic operations in doubly linked list.

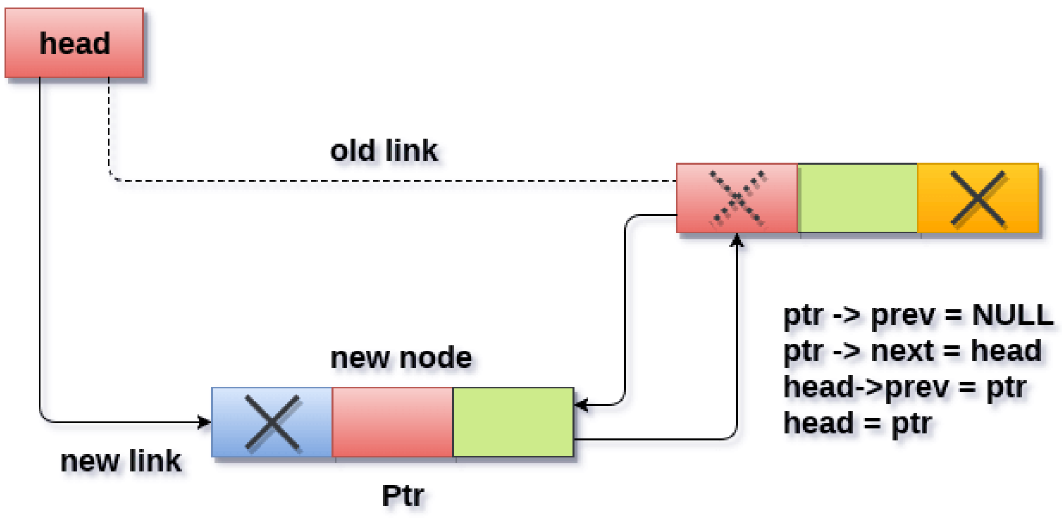
1. Insertion − Adds an element at the beginning of the list.
2. Deletion − Deletes an element at the beginning of the list.
3. Insert Last − Adds an element at the end of the list.
4. Delete Last − Deletes an element from the end of the list.
5. Insert After − Adds an element after an item of the list.
6. Delete − Deletes an element from the list using the key.
7. Display forward − Displays the complete list in a forward manner.
8. Display backward − Displays the complete list in a backward manner.

Insertion at the Beginning

In this operation, we create a new node with three compartments, one containing the data, the others containing the address of its previous and next nodes in the list. This new node is inserted at the beginning of the list.

Steps

1. START
2. Create a new node with three variables: prev, data, next.
3. Store the new data in the data variable
4. If the list is empty, make the new node as head.
5. Otherwise, link the address of the existing first node to the next variable of the new node, and assign null to the prev variable.
6. Point the head to the new node.
7. END

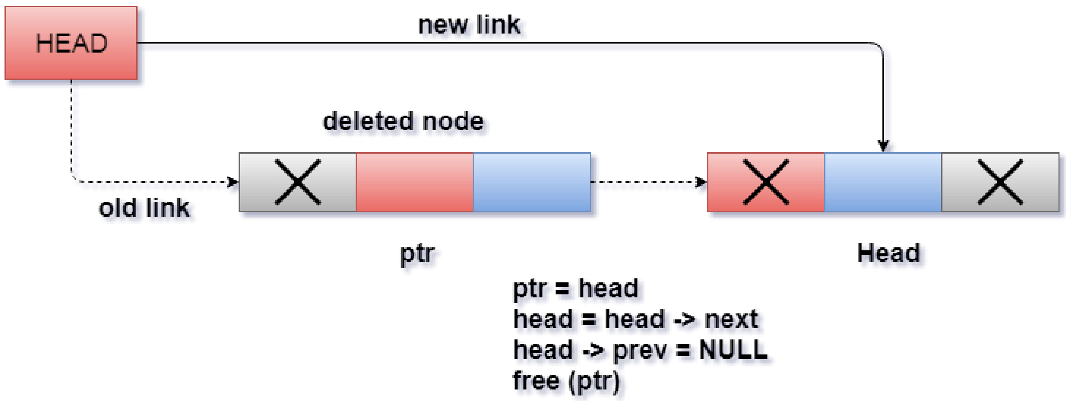


Insertion into doubly linked list at beginning

Deletion at the Beginning
This deletion operation deletes the existing first nodes in the doubly linked list. The head is shifted to the next node and the link is removed.

Steps

1. START
2. Check the status of the doubly linked list
3. If the list is empty, deletion is not possible
4. If the list is not empty, the head pointer is shifted to the next node.
5. END



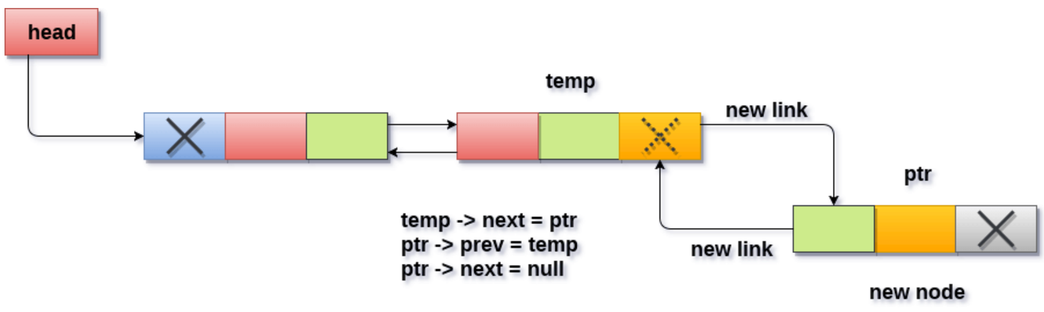
Deletion in doubly linked list from beginning

Insertion at the End

In this insertion operation, the new input node is added at the end of the doubly linked list; if the list is not empty. The head will be pointed to the new node, if the list is empty.

Steps

1. START
2. If the list is empty, add the node to the list and point the head to it.
3. If the list is not empty, find the last node of the list.
4. Create a link between the last node in the list and the new node.
5. The new node will point to NULL as it is the new last node.
6. END



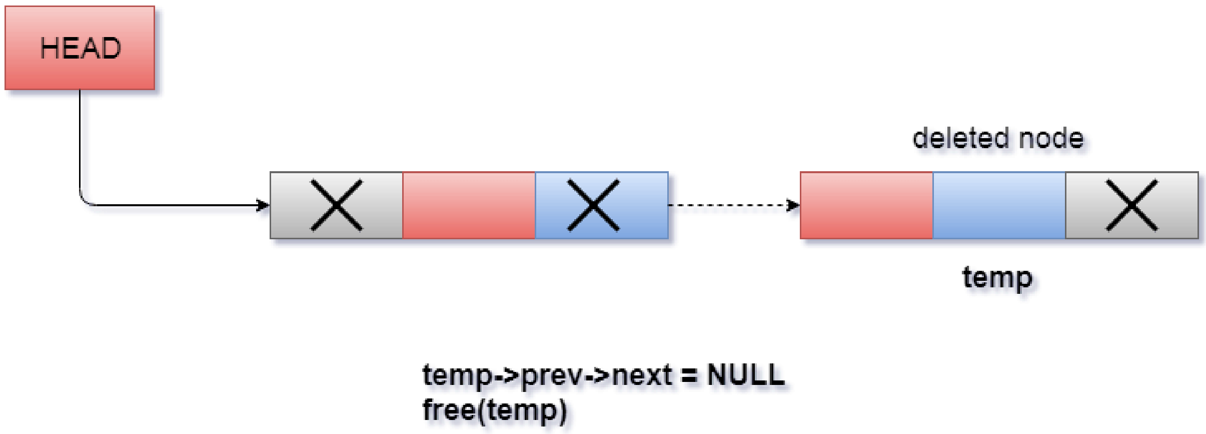
Insertion into doubly linked list at the end

Deletion at the End

In this deletion operation, the last node is deleted at the end of the doubly linked list; if the list is not empty.

Steps

1. START
2. If the list is not empty, delete the last node in the list.
3. Now the last node will point to NULL as it is the new last node.
4. END



Deletion in doubly linked list at the end

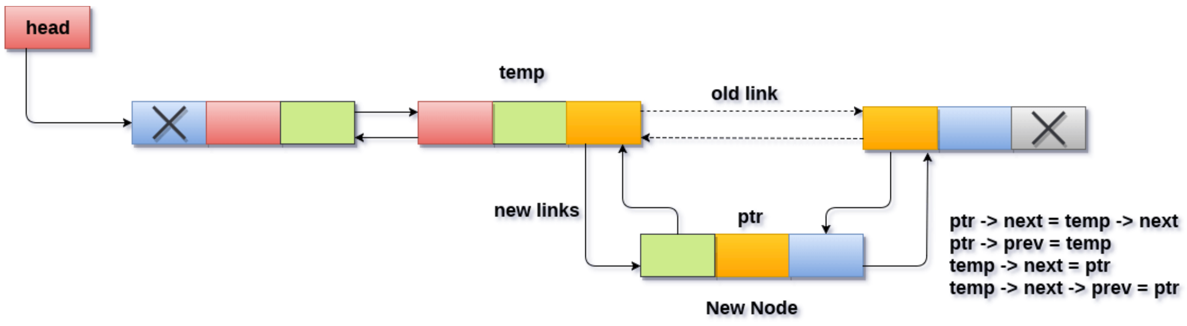
Insertion After a Node

In order to insert a node after the specified node in the list, we need to skip the required number of nodes in order to reach the mentioned node and then make the pointer adjustments as required.

Steps

1. START

2. Allocate the memory for the new node.
3. Traverse the list by using the pointer temp to skip the required number of nodes in order to reach the specified node.
4. The temp would point to the specified node at the end of the for loop.
5. The new node needs to be inserted after this node therefore we need to make a pointer adjustments here.
6. Make the next pointer of ptr point to the next node of temp.
7. Make the prev of the new node ptr point to temp.
8. Make the next pointer of temp point to the new node ptr.
9. Make the previous pointer of the next node of temp point to the new node.
10. END



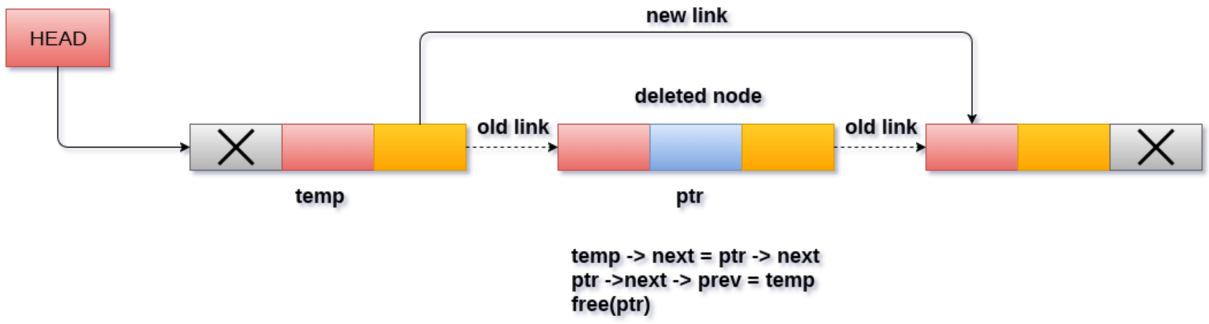
Insertion into doubly linked list after specified node

Deletion After a Node

In order to delete a node after the specified node in the list, we need to skip the required number of nodes in order to reach the mentioned node and then make the pointer adjustments as required.

Steps

1. START
2. Copy the head pointer into a temporary pointer temp..
3. Traverse the list until we find the desired data value.
4. Check if this is the last node of the list. If it is so then we can't perform deletion.
5. Check if the node which is to be deleted, is the last node of the list, if it so then we have to make the next pointer of this node point to null so that it can be the new last node of the list.
6. Otherwise, make the pointer ptr point to the node which is to be deleted. Make the next of temp point to the next of ptr. Make the previous of next node of ptr point to temp. free the ptr.
7. END



Deletion of a specified node in doubly linked list