Data Structure

DS Core Group

ABES Engineering College
College Code-032
Estd. 2000

Authors	Akhilesh Kumar Srivastava, Amit Pandey, Puneet Kumar Goyal, Manish Srivastava, Amrita Jyoti, Asmita Dixit
Authorized By	
Creation/Revision Date	March 2021
Version	1.0

Chapter 9
Linked List

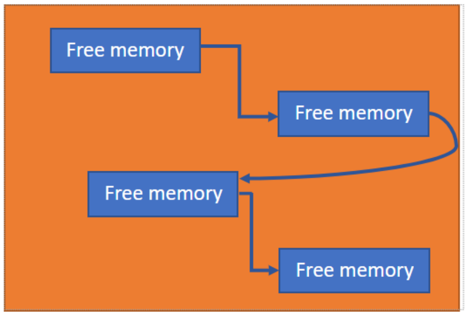
Table of Contents

9.1 Introduction

Suppose a situation wherein a school arranges a movie trip for its students of a class of 100. A teacher has also been assigned for this trip to keep the track students and safely bring them back. Upon reaching to the theater the teacher realizes that the hall is occupied by several people already. The teacher decides to get the students seated one by one wherever the vacant seats are available. When the movie finishes, the students need to be assembled at one place. Teacher has two options: One to note the seat no for each of the students and other one to ask the students to remember the seat no of next student. This way the teacher can keep track of seat no of all the students for assembling them back. The concept is based on the Linked List where teacher knows where the first student is sitting and each student knows the seat number of next students.

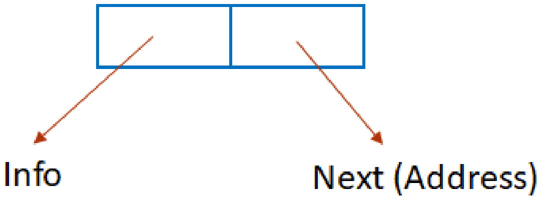
9.1.1 Why Linked List?

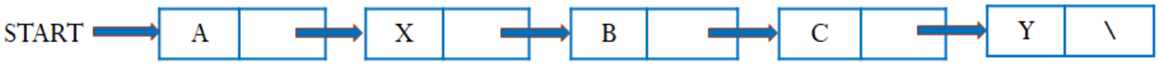
- Consider the RAM where the new process requires the memory. When free memory is not available contiguously, use of linked list for linking of free memory chunks (Holes) will serve the purpose.
- Linked List provides efficient use of memory.
- It's a Dynamic Data Structure, which can change its size (Data Structure can grow or shrink as per the requirement) during the program execution.
- Insertion and deletion operations are easier and efficient as compared to Array.



9.1.2 Definition

A Linked List is a data structure which consists of nodes. Every node contains at least two fields. One of which contains the information and other one contains address of next node.





9.1.2 Properties

- A Linked List is identified through the address of the first node
- To access a node, we need to reach to that node

9.1.3 Linked List Applications:

- Memory Organization
- Web pages (URL Linking)
- Replying to a message after tagging it in WhatsApp, Instagram etc. are linked by linked list.
- Linux File System: Handling big files using the concept of I-Nodes
- Implementation of other Data Structures like Stack, Queue, Tree and Graph (Use of Adjacency List).
- Polynomial Arithmetic
- Arithmetic Computation
- Maintaining directory of names
- Representation of Sparse Matrices
- Maintaining Hash Tables (Chaining)
- Image Viewer software (Use of next and previous buttons for watching images stored in a folder)
- Implementation of Page replacement Algorithms in Operating system

9.1.4 Real life analogy

- Train arrangement: Just like each coach is linked together in a train with a coupler and engine acts as the starting point. In the same pattern Link List has node connections with the starting node's address to START Pointer.
- Music Player Playlist: In a music player playlist, the song which runs currently is proceeded by next song in playlist and the mouse click is the Starting point, just like the address linking in between nodes.
- DNA molecules:
- Roller chain of bicycle: A simple analogy to understand circular link list, that is the supplying of power to wheels in a circular pattern with the help of roller, pin, pin link plate.
- Packet based message delivery on network: Packet delivery via network with linking and acknowledgement back to the origin, in a way Circular Link List.

- Giving travel directions: the existence of direction boards after few kilometers leading the path is in a way explaining the connectivity pattern of Link List.

9.2 Linked List Vs Array

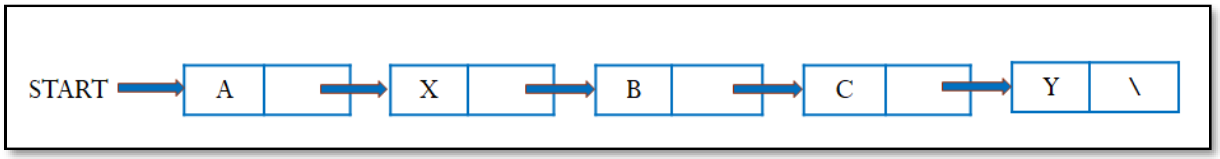
There are some pros and cons with each of the data structure, so too the Array. Array offers the indexed access (fast) but have fixed size. Static array cannot grow or shrink as per the requirement. Array size may fall short or remain under-utilized according the problem in hand. The Dynamic Arrays, e.g. Python List, provide the variable length but the concept is based on reserving surplus memory. Insertion of the data item at the beginning of Static or Dynamic array is costlier than inserting the data at the tail.

Linked List, on the other hand, allocates the memory on demand and insertion and deletions are easier as compared to Arrays.

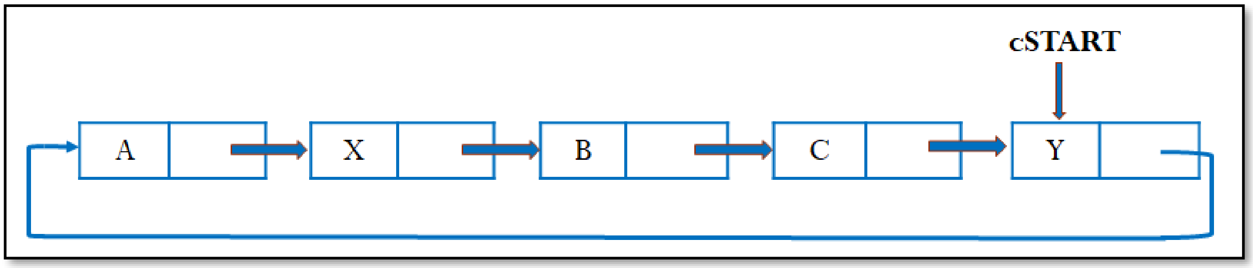
9.3 Types of Linked List

- Linear/Singly Linked List
- Circular Linked List
- Doubly Linked List
- Circular Doubly Linked List

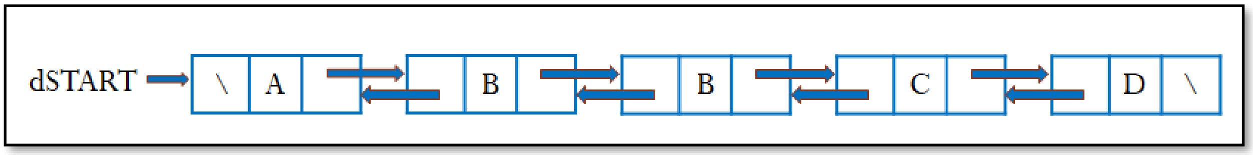
Linear Linked List: The diagram below shows linear Linked List where each node contains information and the address of the next node. The address field of last node contains no address (NULL).



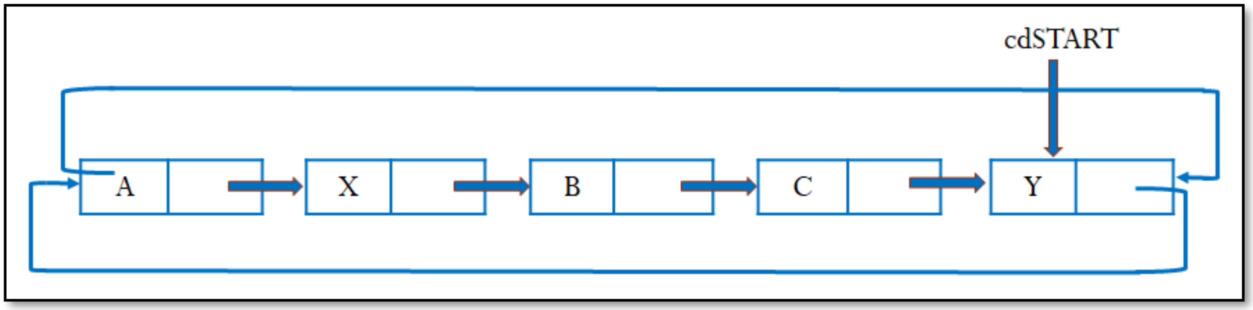
Circular Linked List: Circular Linked List is more like Linear Linked List except that the address field of the last node contains the address of the first node. START keeps the address of the last node in the circular Linked List.



Doubly Linked List: Each node in this type of the Linked List contains at 2 address fields (One for the address of previous node and other one for the address of the next node).



Circular Doubly Linked List: Similar to Doubly Linked List except that the previous field of the first node and next field of the last node is not NULL. Next field of the last keeps the address of the first node and previous field of the first node keeps the address of the last node.

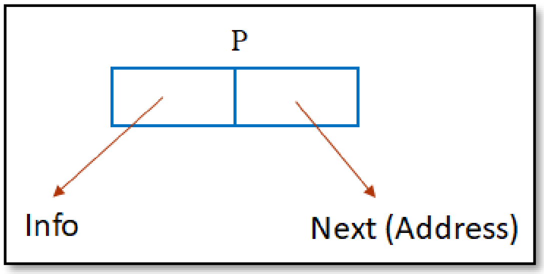


9.4 Computer Science concepts using Linked List as a base

- Blockchain (Sharing the ciphered blocks to next block)
- Computer Network (for breaking the message in packets and linking of packets in order at receiver's end)
- Wireless Sensor Networks (Establishing multi hop connection)
- Time sharing system (Round Robin) in Operating System Process scheduling
- Data base records (Linking of B$^+$Tree Leaf Nodes)

9.5 Notations used in Linked List

- For a node having address P



The fields are accessed as:

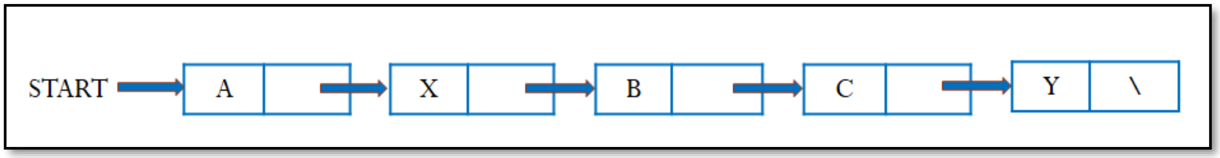
P→ Info

P→Next

- GetNode( ) is used for allocation of memory for new node
- START is used for keeping the address of First node. In case the Linked List is empty, START keeps a NULL.
- Item is used to refer to the element to be inserted.
- When the nodes are deleted, their memory is freed by calling the function FreeNode( )
- For the new node, P is used to keep the address of the newly created node.

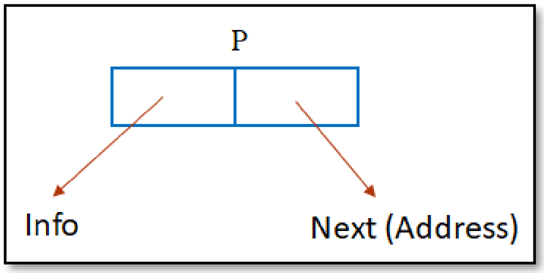
9.6 Single/Singly/Linear Linked List

Single/Linear Linked List: The diagram below shows linear Linked List where each node contains information and the address of the next node. The address field of last node contains no address (NULL).



Notations used in Linear Linked List

For a node having address P



The fields are accessed as:

P→ Info
P→ Next

- **GetNode( )** is used for allocation of memory for new node
- **START** is used for keeping the address of First node. In case the Linked List is empty, START keeps a NULL.
- **Item** is used to refer to the element to be inserted.

9.6.1 Analogy

Single Linked List can be best understood with the concept of treasure hunt. In this, we had to find some items with the help of multiple clues. In this hunt, we have a clue and this clue gives address of next place, thereby we have a starting position and we can search the given item with the help of multiple clues in between.

-

9.6.2 Primitive Operations on Linked List

9.6.2.1 Insertions

1. Insertion at the beginning

In this case a new node is added at the beginning of the Linked List and this new node becomes the starting point of the Linked List.

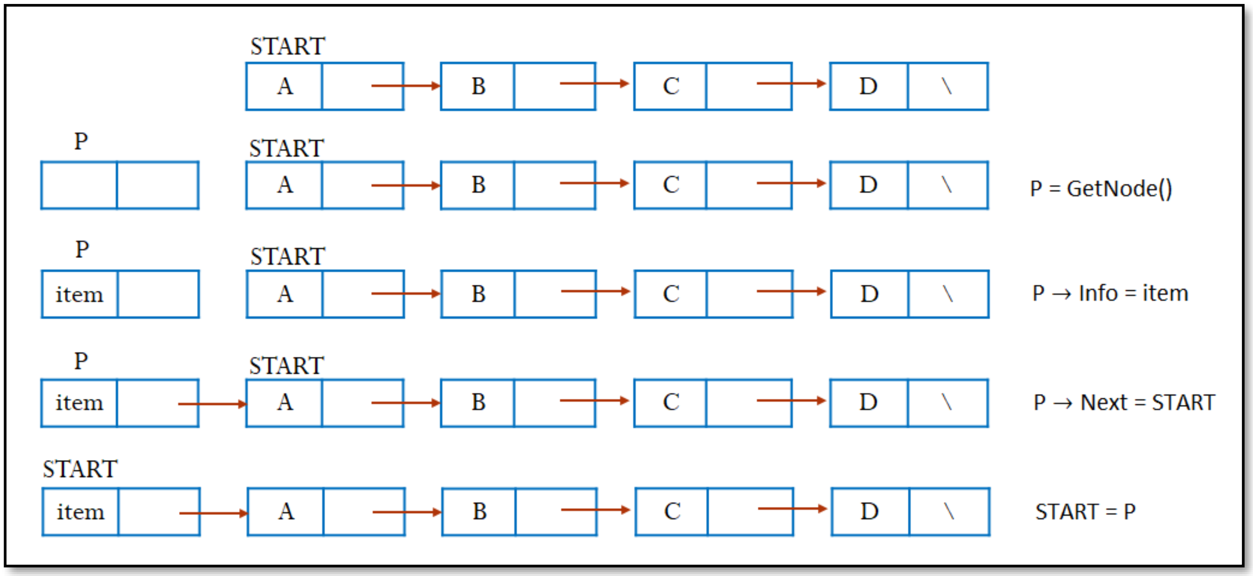
```
ALGORITHM InsBeg(START, item)
BEGIN:
        P = GetNode()
        P → Info = item
        P → Next = START
        START = P
END;
```

No. of Address adjustment = 2

Complexity of Operation

Time Complexity = O(1)

Space Complexity = O(1)



2. Insertion after a given node

In this case, we need to insert a node after a given specific number of nodes. Here, we need to skip the desired numbers of node in list to move the pointer at the point after which the new node will be inserted. Here Q contains the address of the node after which new node will be inserted.

ALGORITHM InsAft(START, Q, item)

// where Q is a pointer containing address of the node after which new node to be inserted and P is the new pointer containing the address of new node to be inserted.

```
BEGIN:
        IF Q == NULL THEN
                WRITE("Void Insertion")
                RETURN
        P = GetNode()
```

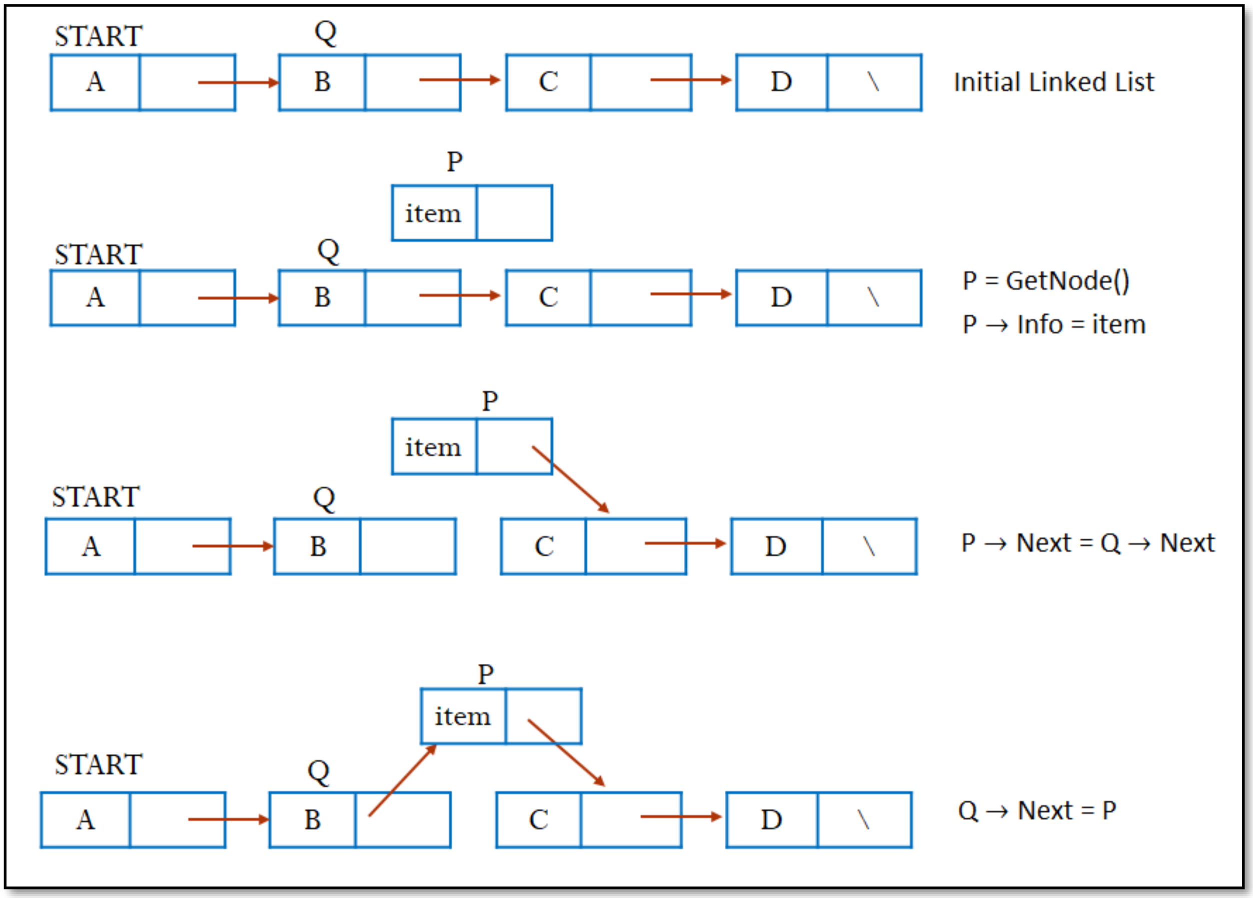
```
          P → Info = item
          P → Next = Q → Next
          Q → Next = P
END;
```

No of link adjustment = 2

Complexity of Operation

Time Complexity = O(1)

Space Complexity = O(1)



3. Insertion at the end

In this case a new node is added at the end of list. For this, we need to traverse the entire list and then change next of last node to the new node.

Prerequisite: - Traversing in Linked List.

ALGORITHM InsEnd(START, item)

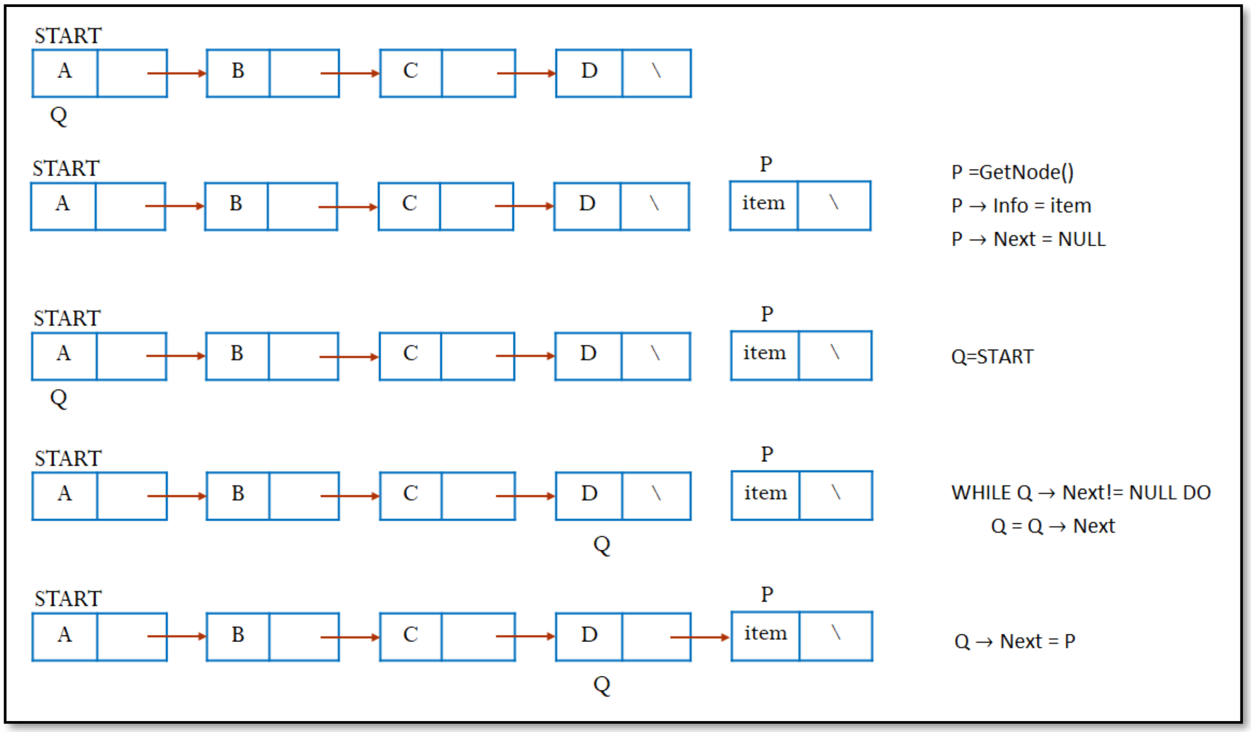
```
BEGIN:
          P =GetNode()
          P → Info = item
          P → Next = NULL
          IF START == NULL THEN
                    START = P
          ELSE
                    Q=START
                    WHILE Q → Next!= NULL DO
                              Q = Q → Next
                              Q → Next = P
END;
```

No of Link adjustment = 2

Complexity of Operation:

Time Complexity = O(n)

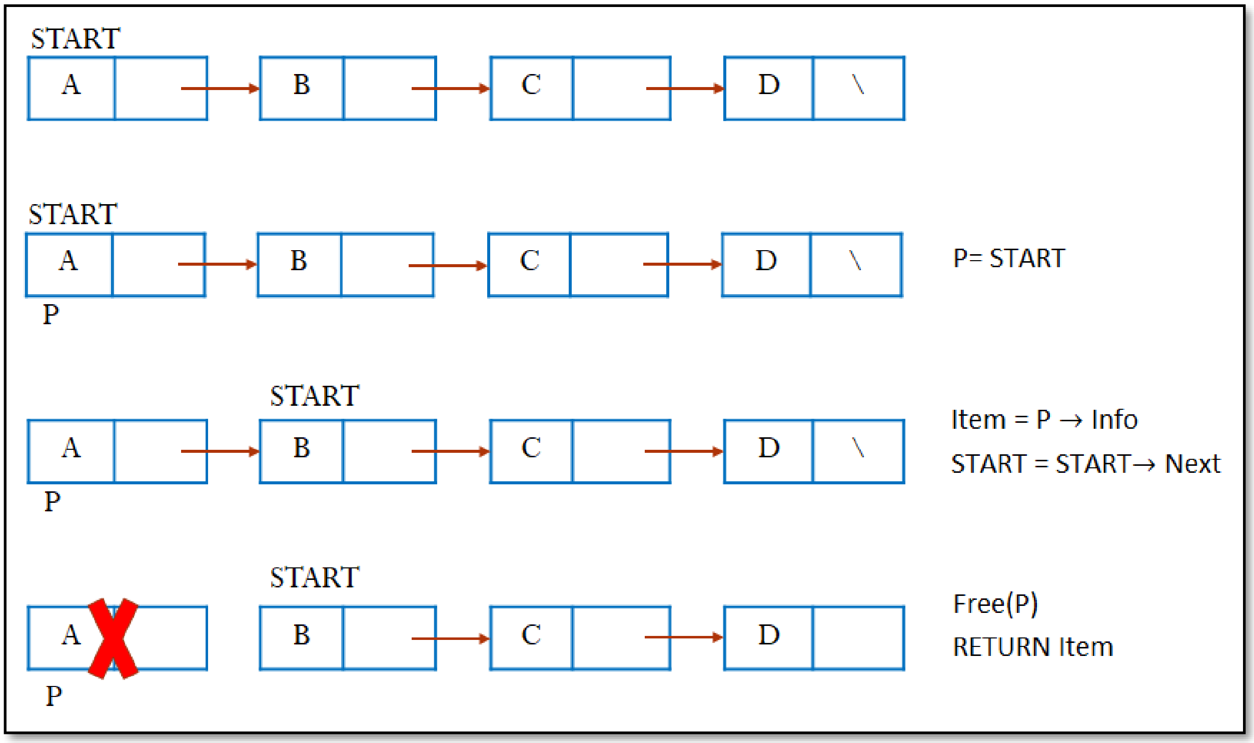
Space Complexity = O(1)



9.6.2.2 Deletion in Linked List

1. Deletion from the beginning

Given a Linked List, our task is to remove the first node of the Linked List and update the head pointer of the Linked List. To remove first node, we need to make the second node as head and delete the memory allocated for first node.



```
ALGORITHM DelBeg(START, item)
BEGIN:
        IF START = = NULL THEN
                WRITE ("Void Deletion")
                RETURN
        ELSE
                P= START
                Item = P → Info
                START = START→ Next
                Free(P)
                RETURN Item
END;
```

No of address Adjustment = 1

Complexity of Operation

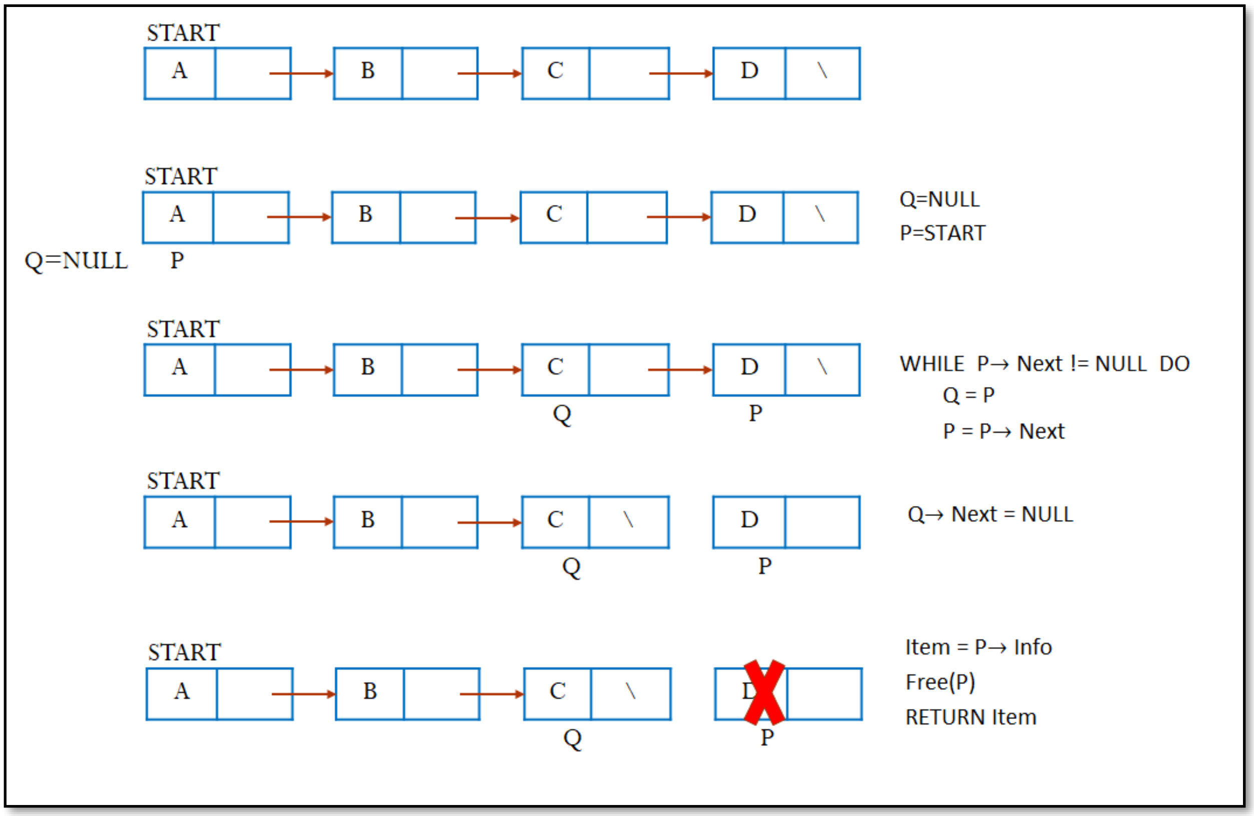
Time Complexity = O(1)

Space Complexity = O(1)

2. Delete from End

Given a Linked List, our task is to remove the last node of this list and set the next part of the second last node as null. To remove last node, we need to traverse up to second last node and set the next part of this node to be null.

```
ALGORITHM  DelEnd(START, item)
BEGIN:
        IF START == NULL THEN
                WRITE (“Void Deletion”)
                RETURN
        ELSE
                Q=NULL
                P=START
                IF START→ Next = NULL THEN          // One node
                START = START→ Next
                ELSE                                //General
                        WHILE  P→ Next != NULL  DO
                                Q = P
                                P = P→ Next
                        Q→ Next = NULL
                        Item = P→ Info
                        Free(P)
                        RETURN Item
END;
```



No of address adjustment = 1

Complexity of Operation:

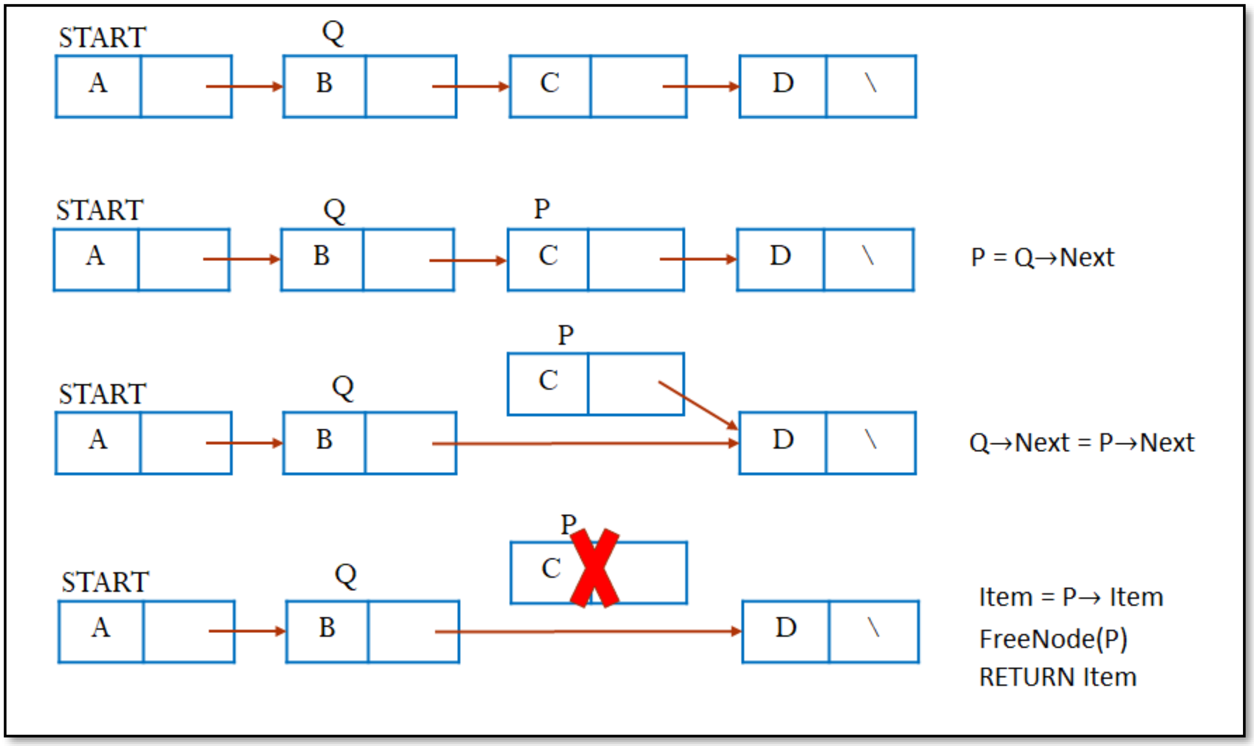
Time Complexity = O(n)

Space Complexity = O(1)

3. Delete node after a given Position

Given a Linked List, our task is to remove the specific node from the list. In this case, we need to traverse up to specific count after which the node is to be deleted. To remove, we need to set the address part of this node to the node after the node to be deleted.

```
ALGORITHM DelAfter(START, Q)
BEGIN:
        IF Q = = NULL AND Q → Next!=NULL THEN
                    WRITE (“Void Deletion”)
        ELSE
          P = Q→Next
          Q→Next = P→Next
          Item = P→ Item
          FreeNode(P)
        RETURN Item
END;
```



No of address adjustment = 1

Complexity of Operation:

Time Complexity = O(1)

Space Complexity = O(1)

9.8 Other Algorithms

9.8.1 Sum of all elements in Linked List

Given a singly Linked List, the task is to find the sum of nodes of the given Linked List.

Solution

The algorithm work by calling the function Add, which helps in adding all the value of data that is present in all nodes by traversing each node from start.

Steps:

- Initialize the node P to start and sum to be 0.
- Traverse the entire list and add the data values present in all nodes of the in variable sum.
- Return this sum value to program.

```
ALGORITHM Add(P)
BEGIN:
        SUM=0
        WHILE P != NULL DO
                SUM = SUM + P→Info
                P = P → Next
        RETURN SUM
END;
```

Complexity of Operation:

Time Complexity: O (n)

Space Complexity: O (1)

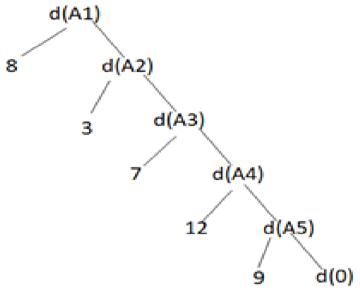
9.8.2 Traversal Algorithm (Recursive)

Traversing in Linked List is determined by visiting all the nodes present in Linked List one by one. In this we visit all the nodes present in Linked List one by one and prints its data value.

Solution:

- Point node P at starting of Linked List. P = Start.
- Visit all the nodes present in Linked List, and prints its data part.

```
ALGORITHM Display(P)
BEGIN:
        IF P ! = NULL THEN
                WRITE  P→Info!=NULL DO
                Display(P→Next)
End;
```



STACK
P =0
P = A5
P = A4
P = A3
P = A2
P = A1

Complexity of Operation:

Time Complexity: O (n)
Space Complexity: O (n)

9.8.3 Concatenation of two Linked List

Analogy: This concept can be best understood with the help of two trains A and B. Train A consists of 8 coaches and Train B consists of 6 coaches. During the time of festival because of increase in rush the railway department concatenate these two trains. It can be done simply by joining the last coach of train A with the first coach of Train B.

Two different Linked List can be concatenated using the same approach. Address part of ending node of list one is linked to first node of second list. This approach only required traversing entire nodes of list one and when we are at last node than we can simply linked its address part with the first node of second list.

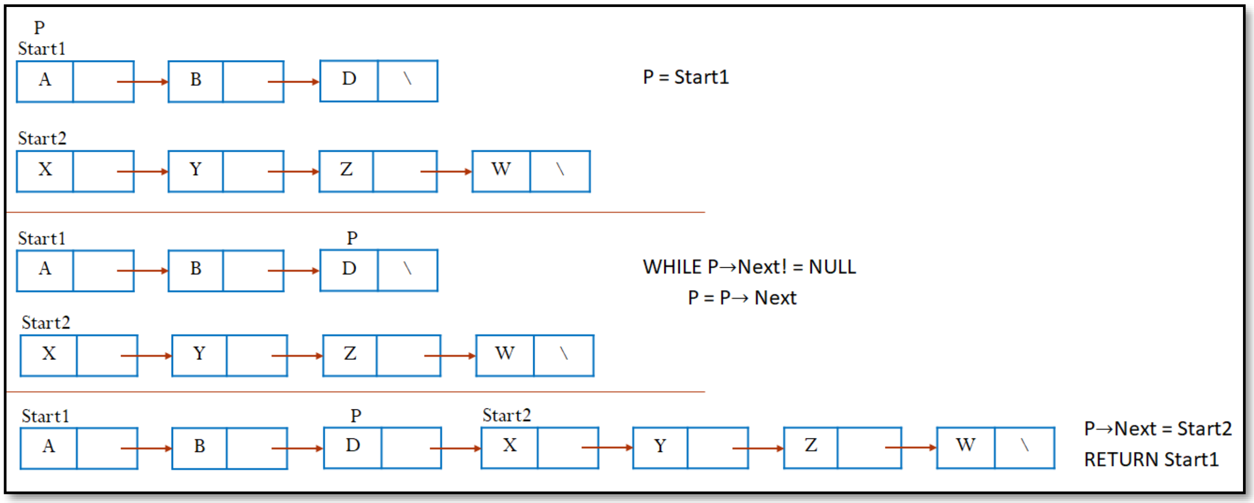
```
ALGORITHM ConcatLinkList(Start1, Start2)
BEGIN:
    IF Start1 == NULL THEN
        RETURN Start2
    ELSE
        IF Start2 == NULL THEN
            RETURN Start1
        ELSE
            P = Start1
        WHILE P→Next! = NULL
            P = P→ Next
            P→Next = Start2
            RETURN Start1
END;
```

Complexity of Operation:

Time Complexity: O (n)
Space Complexity: O (1)

Explanation:

In the above algorithm initially, there are two input Linked List. Start1 has the starting address of first Linked List and Start2 has starting address of second Linked List. A temporary pointer P is taken which traverses till the last node of first Linked List and stops at last node address. Then the Next part of last node is assigned the address of first node of second Linked List via Start2.



9.8.4 Ordered Insertion

It is an operation, which inserts an element at its correct position in a sorted Linked List.

ALGORITHM OrderedInsertion (START, Key)

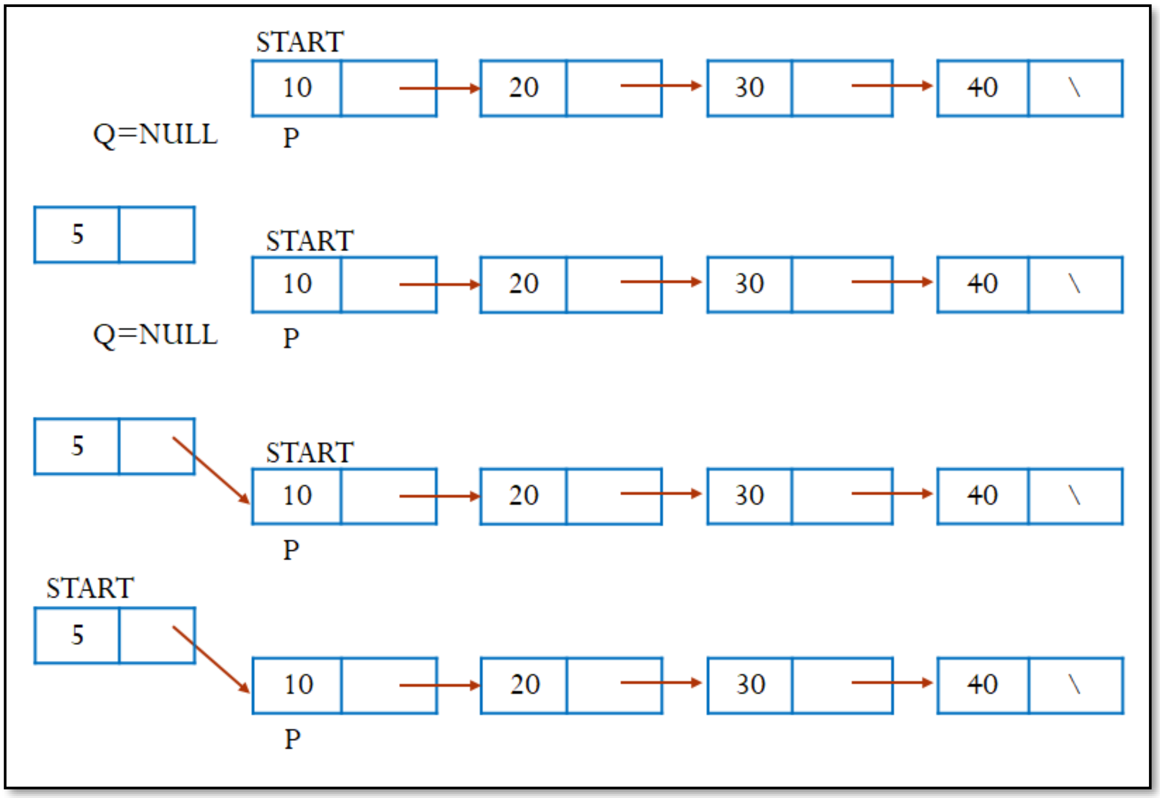
```
BEGIN:
        Q = NULL
        P = START
        WHILE P != NULL AND Key >= P→Info DO
                Q = P
                P = P→Next
        IF Q != NULL THEN
                InsAft(Q, Key)
        ELSE
                InsBeg(START, Key)
END;
```

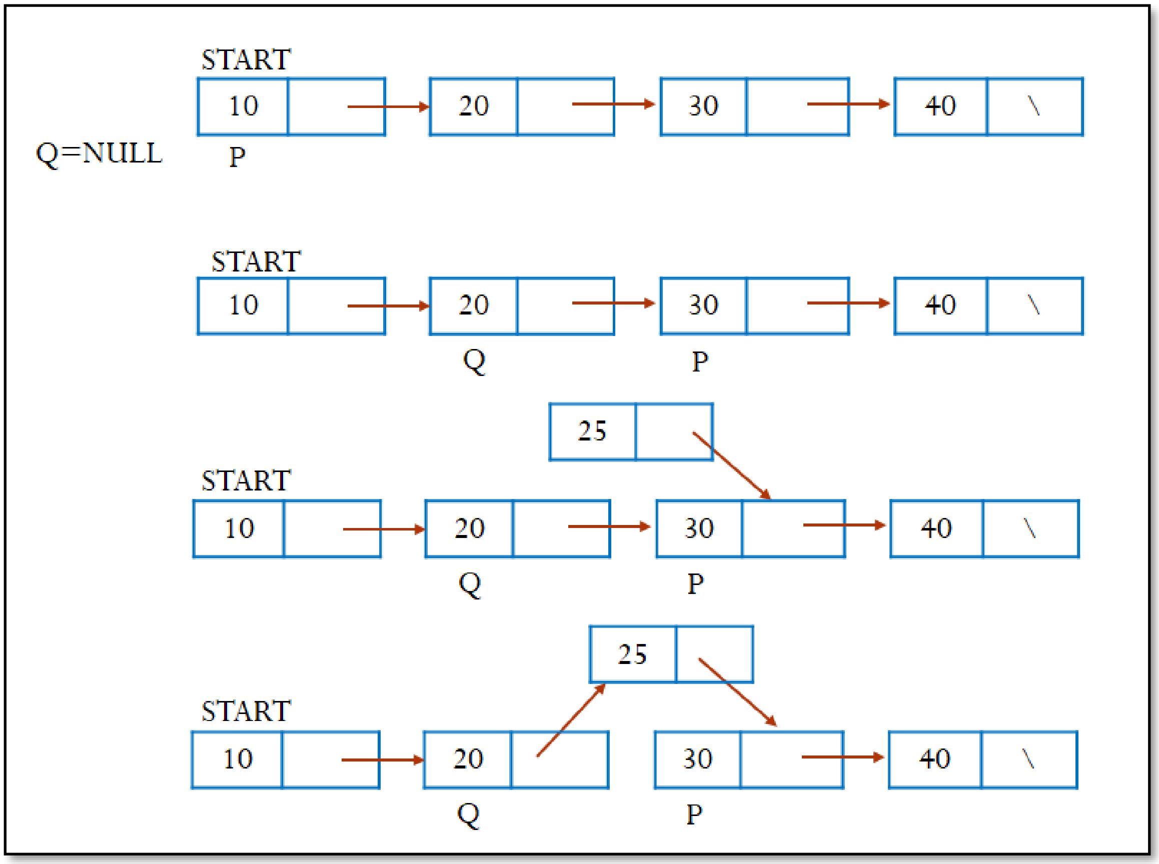
Complexity of Operation

No of Address node Adjustment = 2

Time Complexity = Ω(1), O(n)

Space Complexity = O(1)


START
10
20
30
40
\
Q=NULL
P
5
START
10
20
30
40
\
Q=NULL
P
5
START
10
20
30
40
\
P
START
5
10
20
30
40
\
P


START
10
20
30
40
\
Q=NULL
P
START
10
20
30
40
\
Q
P
25
START
10
20
30
40
\
Q
P
25
START
10
20
30
40
\
Q
P

9.8.5 Merging

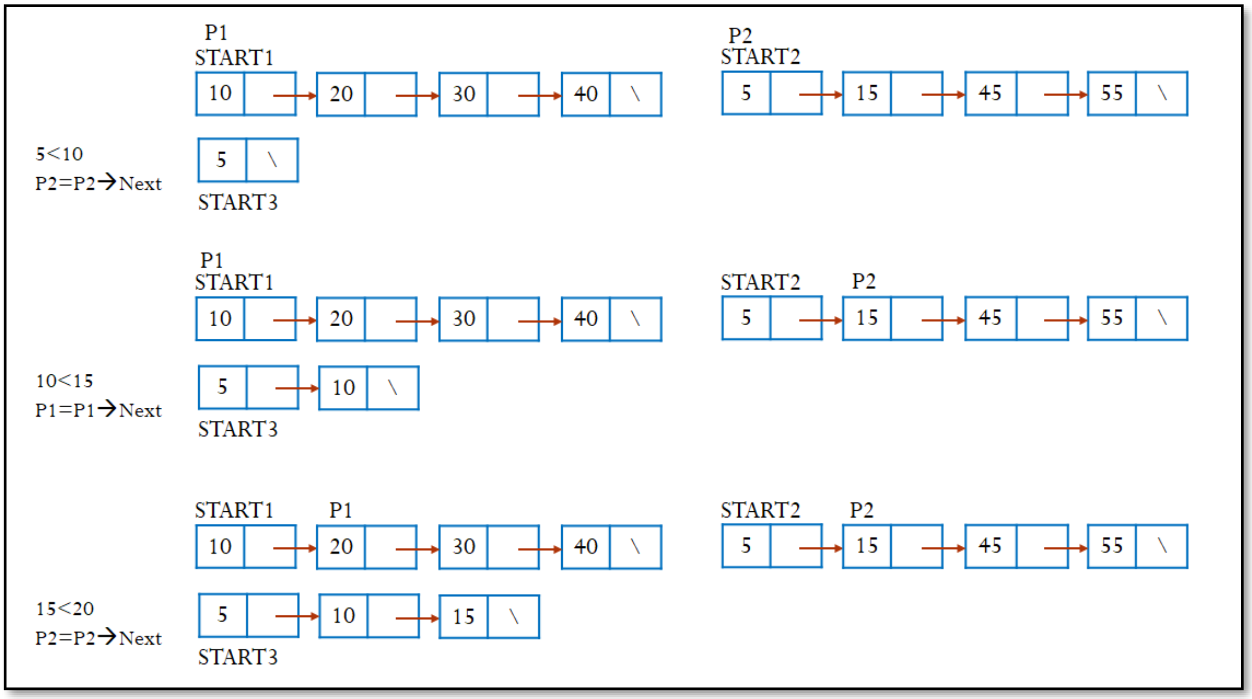
It is an approach in which we combine two sorted Linked List and generates third sorted list. Here first list contains m nodes, second list contains n nodes and the final sorted list contains m+n nodes.

```
ALGORITHM Merging (START1, START2)
BEGIN:
          P1 = START1
          P2 = START2
          START3 = NULL
                   WHILE P1 != NULL AND P2 != NULL DO
                              IF P1 →Info < = P2 →Info THEN
                                        InsEnd(START3, P1→Info)
                                        P1 = P1→Next
                              ELSE
                                        InsEnd(START3, P2→Info)
                                        P2 = P2→Next
          WHILE P1 != NULL DO
                    InsEnd(START3, P1→Info)
                    P1 = P1→Next
          WHILE P2 != NULL DO
                    InsEnd(START3, P2→Info)
                    P2 = P2→Next
RETURN START3
END;
```

Complexity of Operation

Time Complexity = O(m+n)

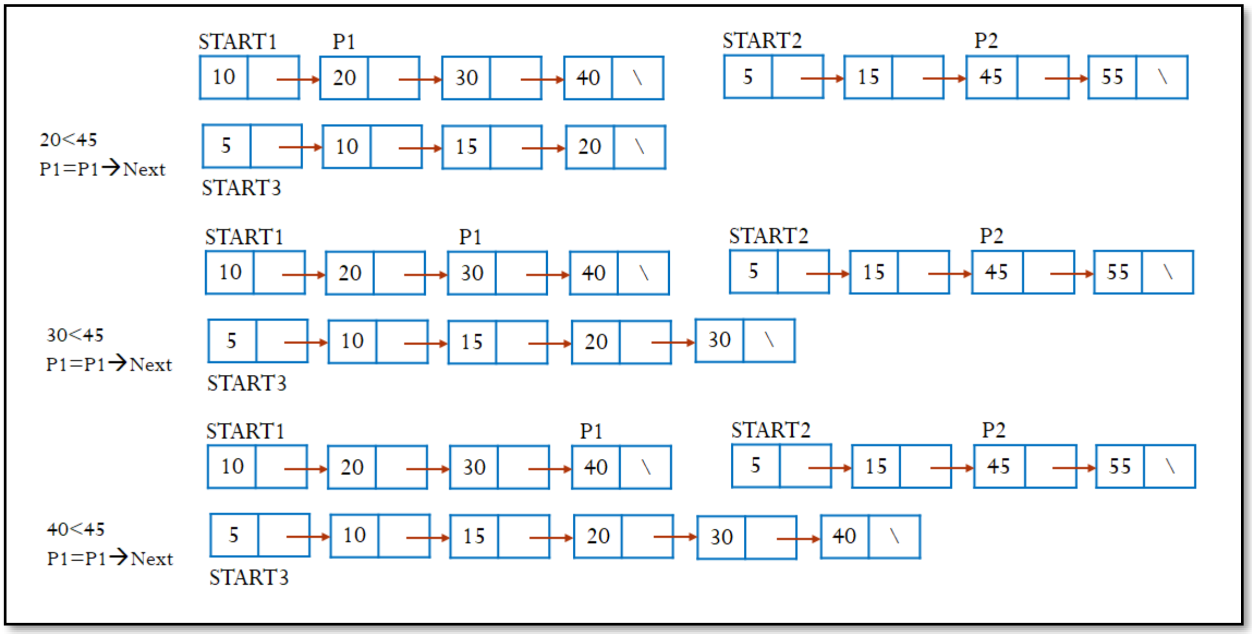
Space Complexity = O(m+n) extra space



5<10
P2=P2→Next

10<15
P1=P1→Next

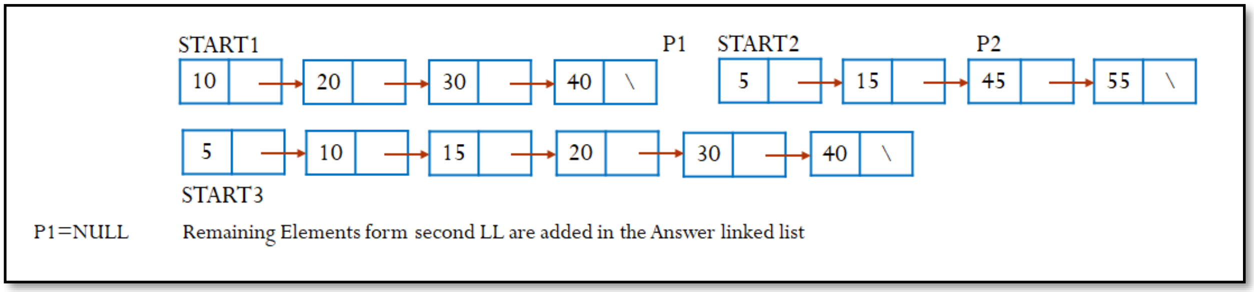
15<20
P2=P2→Next



20<45
P1=P1→Next

30<45
P1=P1→Next

40<45
P1=P1→Next



P1=NULL Remaining Elements form second LL are added in the Answer linked list

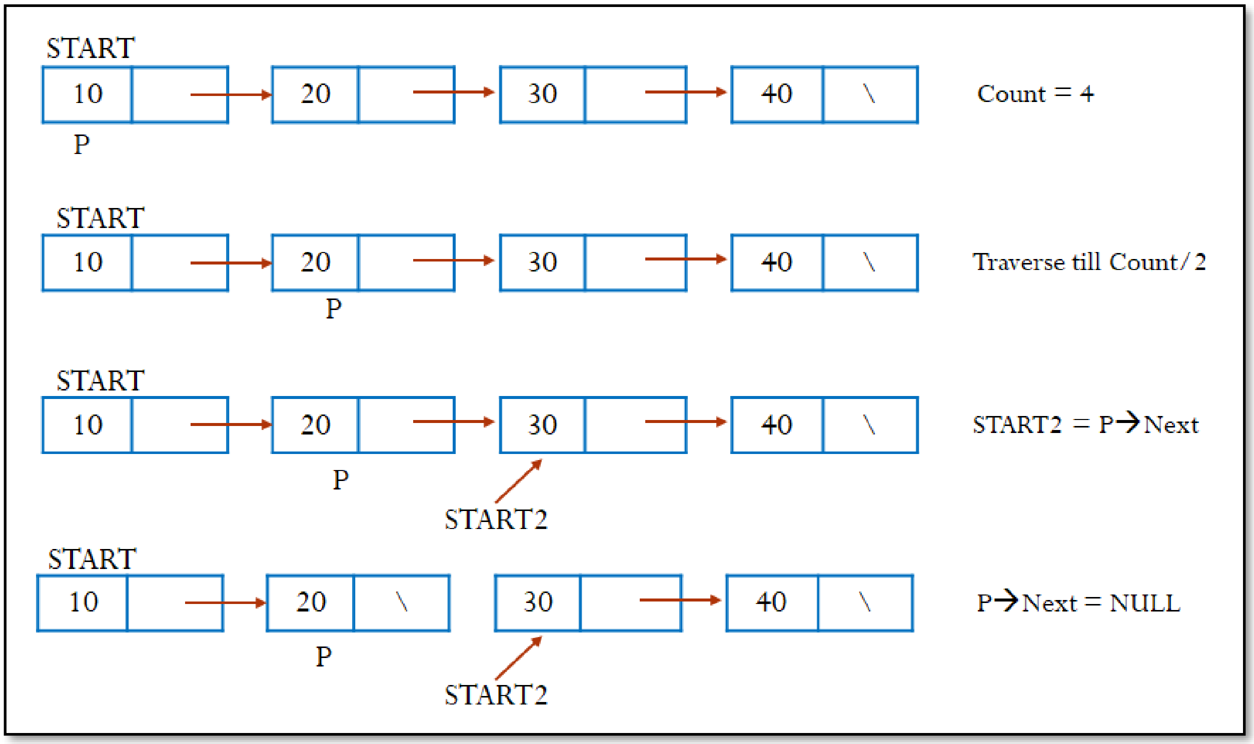
9.8.6 Split List from Mid

It is an operation which splits a given Linked List from its mid node and produces two Linked List where first list ends at mid node and second list starts from the node next to mid node.

Method-1

In this method, firstly count the total number of nodes in the list and find the address of mid node by dividing total number of nodes by 2. Also consider the case if there is only one node then split list from mid function cannot be performed.

```
ALGORITHM SplitMid(START1)
BEGIN:
        START2 = NULL
        Count = NodeCount(START1)
        P = START1
        i = 1
        WHILE i < Count/2 DO
                P= P→Next
                i = i + 1
        IF Count != 1 THEN
                START2 = P→Next
                P→Next = NULL
                RETURN START2
        ELSE
                WRITE (“One node in List”)
                RETURN NULL
END;
```



Complexity of Operation:

No of address node Adjustment = 1

Time Complexity = O(n)

Space Complexity = O(1)

Method-2

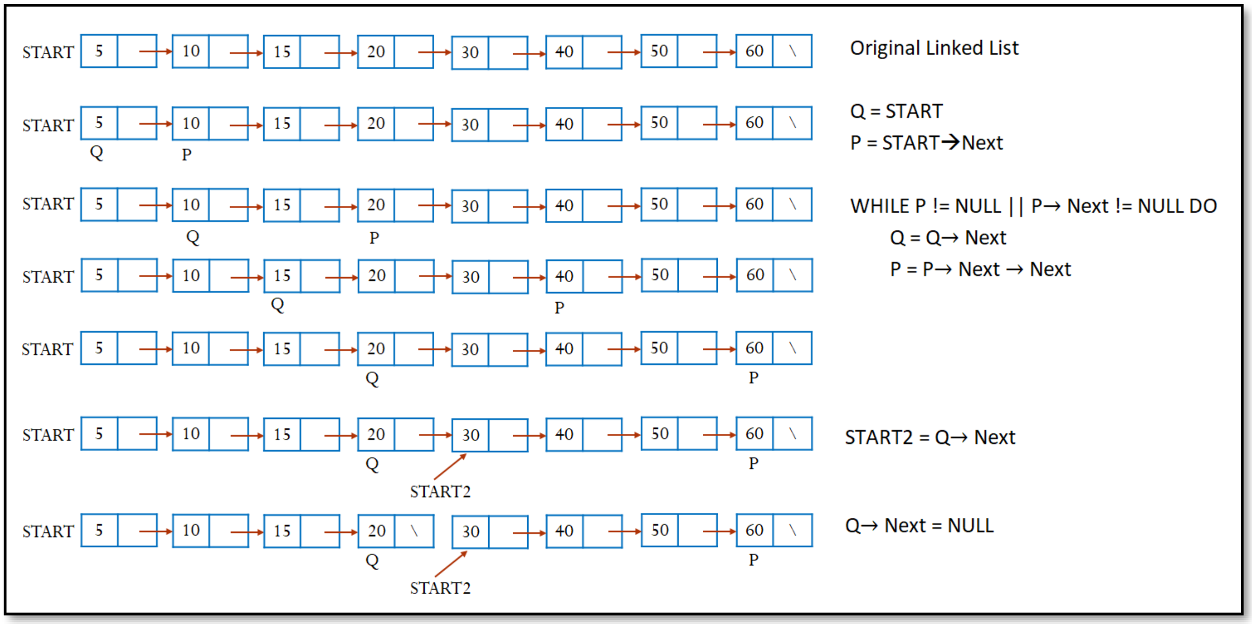
In this method, we will traverse the list using two pointers slow pointer Q and fast pointer P. Slow pointer will be moved by one position and fast pointer will be moved by two position.

ALGORITHM SplitMid(START)

BEGIN:

```
        Q = START
        P = START→Next
        WHILE P != NULL || P→ Next != NULL DO  /* IF fast pointer P points to either last
                                                node or second last node (Q lies in the
                                        middle)*/
                Q = Q→ Next
                P = P→ Next → Next

        START2 = Q→ Next
        Q→ Next = NULL
        RETURN START2
END;
```



Complexity of Operation:

No of address node Adjustment = 1

Time Complexity = O(n/2) = O(n)

Space Complexity = O(1)

9.8.7 Union of two sorted lists

The Union of two sorted Linked List is the new Linked List which contains all the elements which are present in at least one of the Linked Lists.

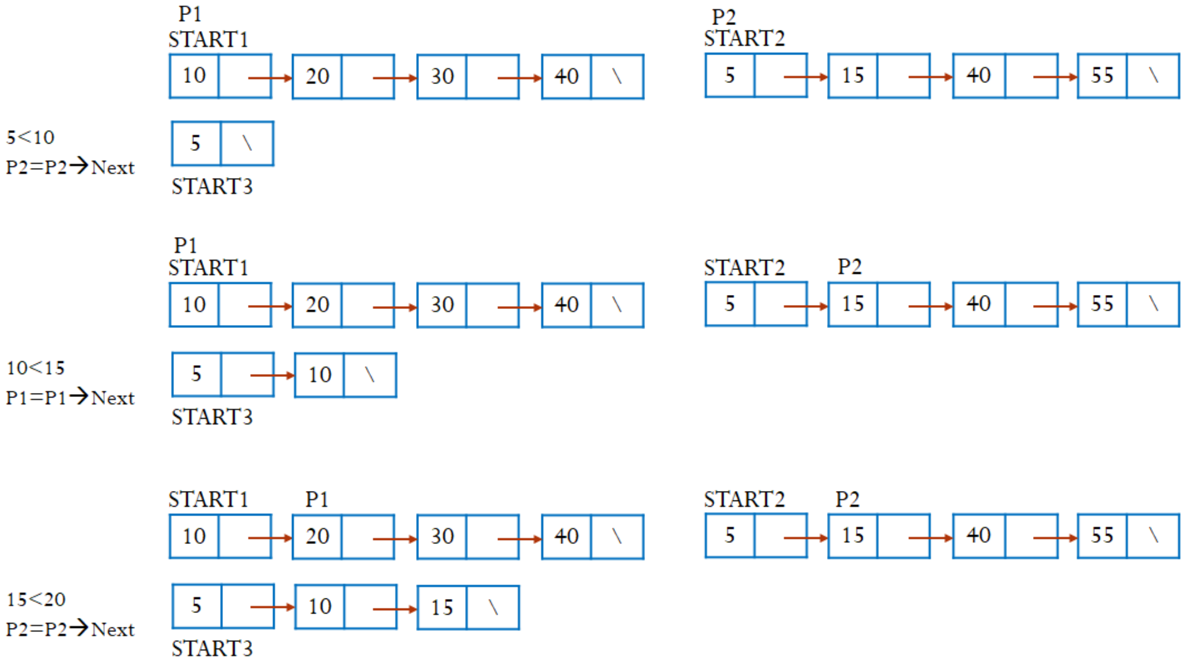
```
ALGORITHM Union (START1, START2)
BEGIN:
        P1 = START1
        P2 = START2
        START3 = NULL
        WHILE P1 != NULL AND P2 != NULL DO
                IF P1 →Info <  P2 →Info THEN
                        InsEnd(START3, P1→Info)
                        P1 = P1→Next
                ELSE
```

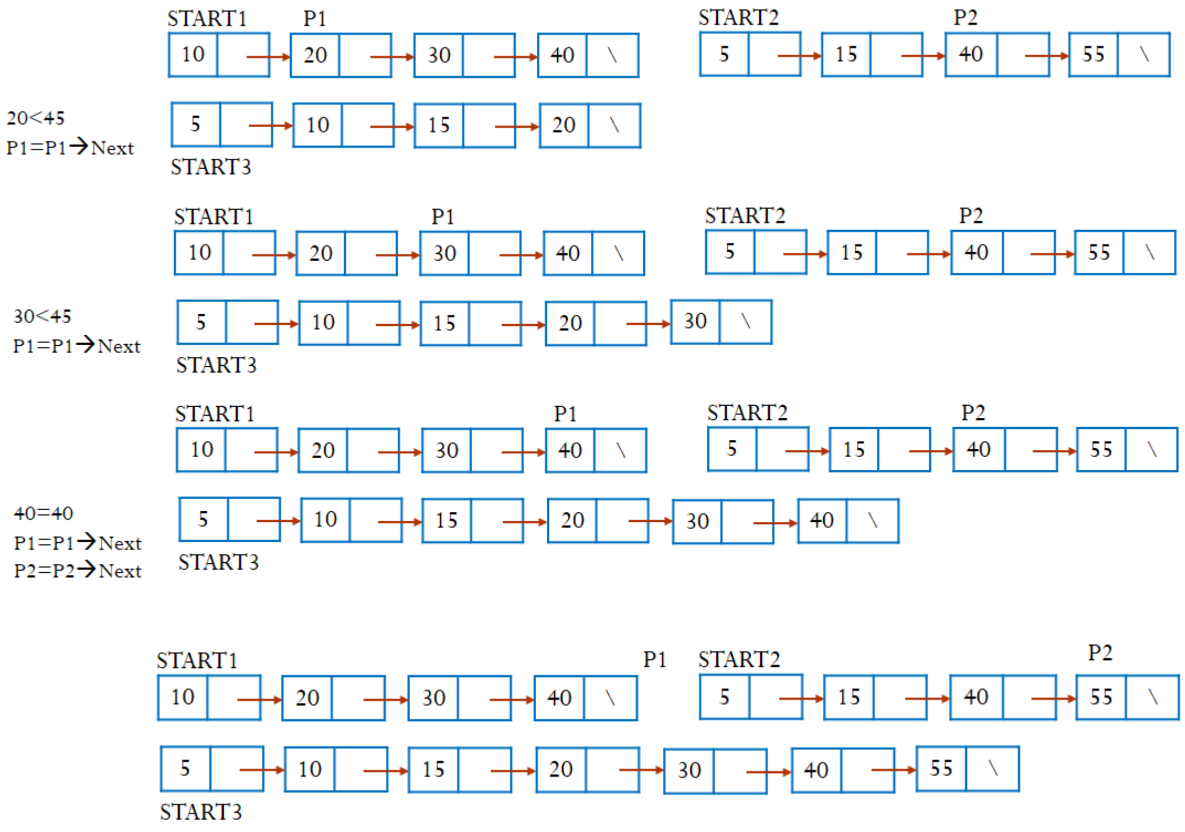
```
            IF P2 →Info < P1→Info THEN
                InsEnd(START3, P2→Info)
                P2 = P2→Next
            ELSE
                InsEnd(START3, P1→Info)
                P1 = P1→Next
                P2 = P2→Next
        WHILE P1 != NULL DO
            InsEnd(START3, P1→Info)
            P1 = P1→Next
        WHILE P2 != NULL DO
            InsEnd(START3, P2→Info)
            P2 = P2→Next
END;
```

Complexity of Operation

Time Complexity = O(m+n)

Space Complexity = O(m+n) extra space





P1=NULL Remaining Elements form second LL are added in the Answer linked list

9.8.8 Intersection of two sorted Linked List

This algorithm performs set intersection operation on given two Linked List.

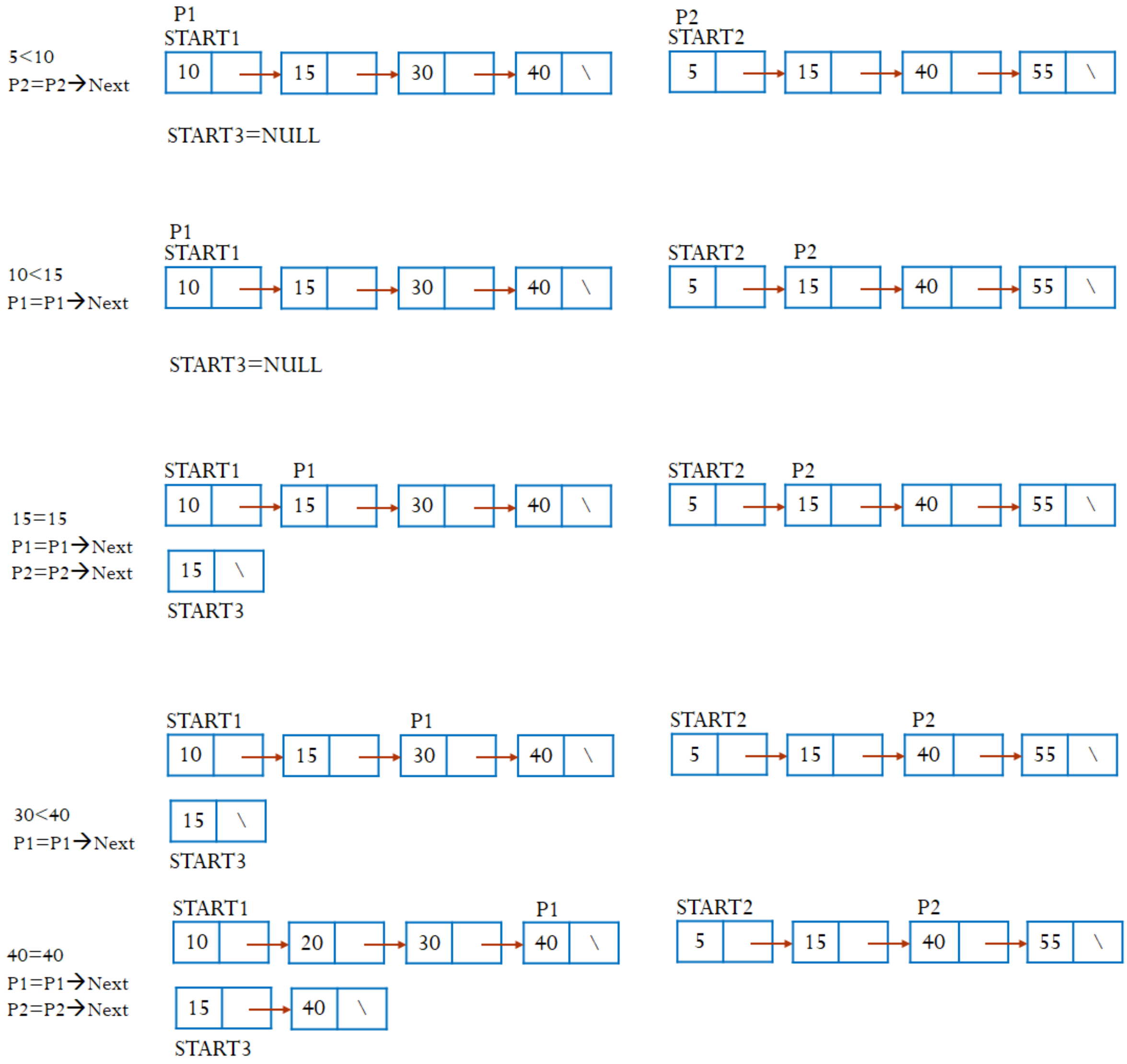
```
ALGORITHM    SetIntersectionLL(START1, START2)
BEGIN:
    P1 = START1
    P2 = START2
    START3 = NULL
    WHILE P1! = NULL AND P2! = NULL DO
        IF P1→ Info! = P2 → Info THEN
            IF P1 → Info < P2 → Info THEN
                P1 = P1→Next
            ELSE
                P2=P2 → Next
        ELSE
            Insertend(START3, P1→Info)
            P1=P1→Next
            P2=P2→Next
        RETURN START3
END;
```

Explanation: In the above algorithm initially, there are two input Linked List. START1 has the starting address of first Linked List and START2 has starting address of second Linked List. A temporary pointer P1 is taken which traverses till the last node of first Linked List and P2 is taken which traverses till the last node of second Linked List. Every Information field of first list is compared with second and the common Information field is picked. The common Information field is then added in third Linked List with START3 pointer using Insertend( ).

Complexity:

Time Complexity: O(m + n)

Space Complexity: O(x) where x is maximum between m and n



9.8.9 Difference (Set Difference) of two sorted Linked List

Considering A – B

```
ALGORITHM SetDifference(START1, START2)
BEGIN:
    P1 = START1
    P2 = START2
    START3 = NULL
    WHILE P1 != NULL AND P2 != NULL DO
        IF P1 →Info < P2 →Info THEN
            InsEnd(START3, P1→Info)
            P1 = P1→Next
        ELSE
            IF P2 →Info < P1→Info THEN
                P2 = P2→Next
            ELSE
                P1 = P1→Next
                P2 = P2→Next
    WHILE P1 != NULL DO
        InsEnd(START3, P1→Info)
        P1 = P1→Next
    RETURN START3
END;
```

Complexity of Operation

Time Complexity = O(m+n)

Space Complexity = O(x), x is smallest of m and n

9.8.10 Reverse the Linked List (In Place)

Problem: Given address to the head node of a Linked List, the task is to reverse the Linked List. (We need to reverse the list by changing the links between nodes).

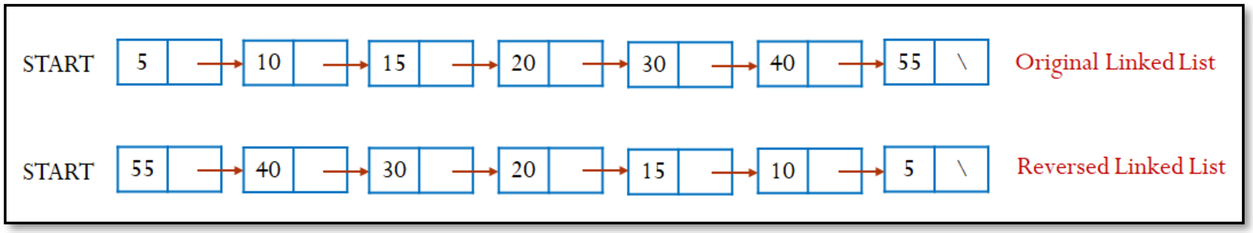
Analogy

The weblink can be reversed using browser cache which allow you to hit the back button. By pressing Back button we can reach to the starting web page. In this case when we start to reverse than the last node will be the first node and link direction will be reversed.

Solution

Given address of the head node of a Linked List, the task is to reverse the Linked List. We need to reverse the list by changing the links between nodes.

1. Initialize three nodes R as NULL, P as start and Q as NULL.
2. Iterate through the Linked List.



```
ALGORITHM ListReverse(START)
BEGIN:
    IF START= NULL THEN
            WRITE (“List is empty”)
    ELSE
            P = START, R = NULL
            WHILE P! = NULL DO
                  Q = P→ Next
                  P→ Next = R
                  R = P
                  P = Q
            START = R
END;
```

9.8.11 SEARCHING IN LINKED LIST

Consider a situation in which Ten students having roll number from 1 to 10 are standing in line in increasing order holding hands of each other. In other words, student one has held the hand of second one, second student had held the hand of third one and so on. The first student can be considered as the base or starting point. Let we have to search student name Rahul among them, then student here can be regarded as node of list while hand can be considered as link for searching.

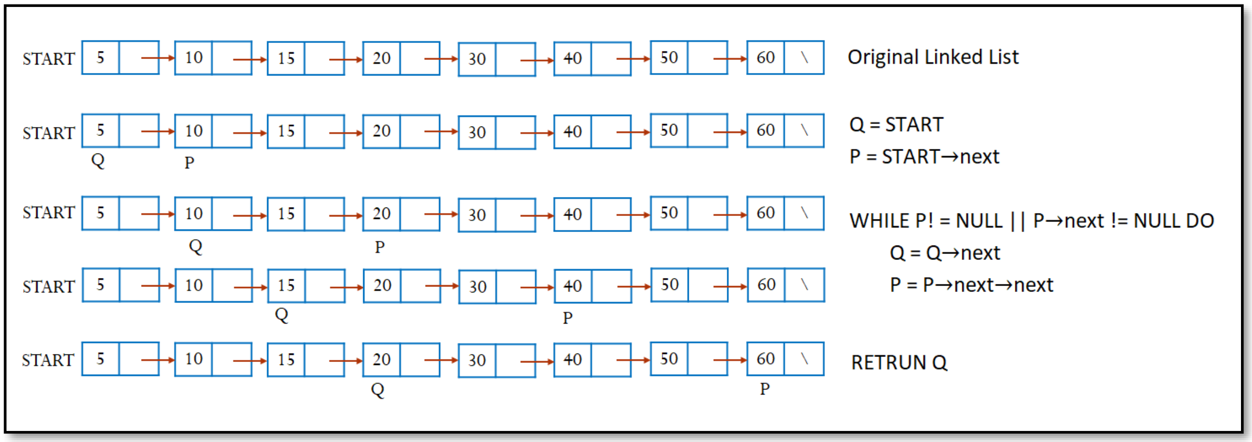
```
ALGORITHM Searching (START, item)
BEGIN:
    P = START
    WHILE P! = NULL DO
        IF P→ info == item THEN
        WRITE (“Search successful”)
        RETURN P
    ELSE
        P = P→ next;
    RETURN NULL
END;
```

Above function performs the Linear Search on the Linked List. We can also perform the Binary Search on the Linked List. For this we require the address of the Middle element in the linked list. Subsequent methods are written to find the Middle element in the linked list.

9.8.12 Finding Middle Element in the Linked List

```
ALGORITHM FindMiddle (START)
BEGIN:
    P = START
    count = 0
    WHILE P!= NULL DO
        count++
        P = P →next;
    count = count/2
    Q = START
    i = 1
    WHILE  i != count DO
        Q = Q →next
        i = i + 1
        RETURN Q
END;
```

Finding Middle using slow and fast pointer



Original Linked List

```
Q = START
P = START→next
```

```
WHILE P! = NULL || P→next != NULL DO
    Q = Q→next
    P = P→next→next
```

```
RETRUN Q
```

ALGORITHM FindMiddle(START)

```
BEGIN:
    Q = START
    P = START→next
    WHILE P! = NULL || P→next != NULL || P→next→next!=NULL DO
        Q = Q→next
        P = P→next→next
    RETRUN Q
END;
```

9.8.13 Binary Search on Linked List

ALGORITHM BinarySearch(START, item)

```
BEGIN:
    IF START!=NULL THEN
        Mid=FindMiddle(START)
        IF Mid →info == item THEN
            RETURN Mid
        ELSE
            IF item < Mid →info THEN
                Mid →Next = NULL
```

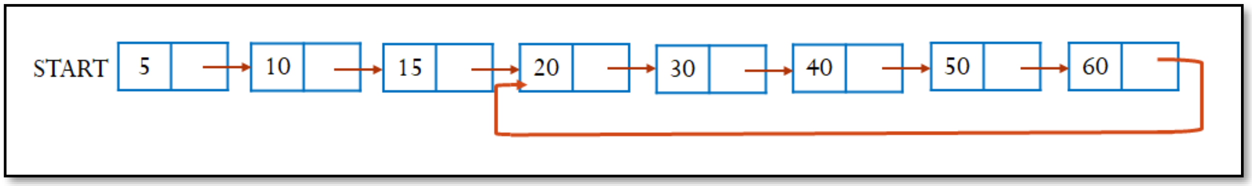
```
                        BinarySearch(START, item)
                ELSE
                        START=Mid →Next
                        BinarySearch(START, item)
        ELSE
                RETURN NULL
END;
```

For Performing Binary Search on Linked List, we need to find the middle element in the Linked List. In case the information to be searched matches with the information of mid node, the search is successful. Otherwise, the search area is reduced to left half or right half of the Linked List (wrt Mid node) depending on if the item to be searched is less than or greater than the information of Mid node respectively. In case the START becomes NULL while in search, the search becomes unsuccessful.
The Time Complexity of This Algorithm is O(N) and not O(Log N). This is because for each call of Binary search, we need to find the Mid Element, that requires O(N) Time.
Space Complexity of the Algorithm is O(Log N) for Recursive nature of this function, and O(1) for Iterative nature.

9.8.14 LOOP DETECTION

Problem: Write an Algorithm

- To check if the link list has a loop or not.
- If link list has a loop then find a length of loop.
- Find start point of loop.



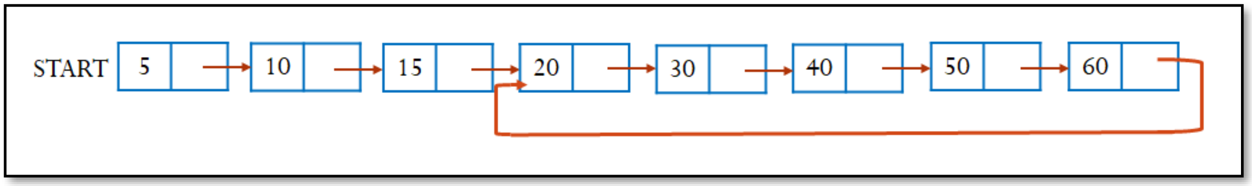
It is already understood that if there is loop or cycle in the link list then there is no end. It means in this case normal traversal does not work because there is no idea where to stop.

Method1- Marking Visited Node

It is considered that along with the information and next address field, the nodes in the linked list have an additional field named as "visited". It is considered that while creating the linked list, the visited field of all nodes have been set as false.

LOGIC-

- First initialize visited field of all nodes is FALSE.
- After that traverse every node and set visited field of node is true.
- If again comes true then it means loop exist otherwise not.



ALGORITHM LoopFind(START)

```
BEGIN:
    P = START
    WHILE P! =NULL AND P→Next! =NULL AND P→visited==FALSE DO
        P→visited=TRUE
        P=P→Next
    IF P→visited==TRUE THEN
        WRITE (loop found)
    ELSE
        WRITE (loop not found)
END;
```

Here we traverse the one time the entire link list so complexity of link list is O(N) but initialize visited field of all nodes so it takes O(N) extra space if variable consider otherwise O(1).

Space complexity: O (N) extra space
Time complexity O(N)

Method-2 (Reversing the List)

LOGIC-

1-Take a pointer P which points START of node.
2- Reverse the entire list
3-If there is a loop in the list, again P points START of node (because cycle exist)
4-If P not points START of node then there is no loop/cycle exist
5-Again reverse the list to get original list

ALGORITHM LoopFind(START)

```
BEGIN:
    P = START
```

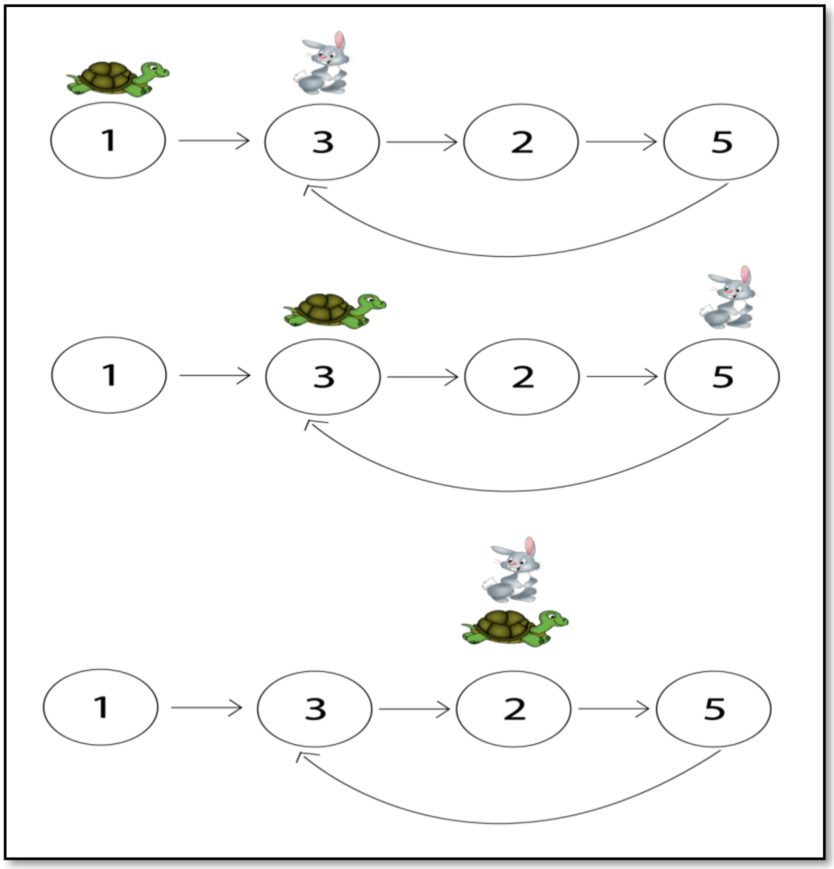
```
          Reverse(P)
          IF P==START THEN
                    WRITE ("loop found")
          ELSE
                    WRITE ("loop not found")
END;
```

This algorithm again takes O (N) time and constant space because just traverse the list single time.

Method-3- (Tortoise and Hare Approach or concept of slow and fast pointer) –

Analogy

Once upon a time there live a hare who runs very fast due to which he was very proud of this, there was a tortoise who was very hard worker. One day due to some reason they both quarreled on some issue and it was decided one who will cover a distance of 50 meter first will won the race. Hare, being proud of his ability to run fast agreed at once and Tortoise also agreed at last. They both start the race but hare was over confident due to which he started to take rest under the tree. At average, it was taken that tortoise was able to take two steps and hare was only able to take only one step. So, at last hard work wins.



LOGIC-

- Here take two pointers slow and fast pointers and both pointers point to START node.
- Move slow pointer to normal speed but fast pointer to double speed.
- If fast pointers reach to NULL then it means there is no cycle because fast pointers move to double speed and that time slow pointer points middle of the list.
- If slow and fast pointers are equal then there is a loop.

```
ALGORITHM LoopFind (START)
BEGIN:
        P = START
        Q = START
        WHILE P!= NULL AND P→link != NULL DO
                Q = Q→link
                P = P→link→link
        IF  P == Q THEN
                RETURN TRUE

         RETURN FALSE
END;
```

Complexity- here is the complexity is O(N) because traverse the list only once by slow pointer.
Time Complexity O(N)
Space Complexity O(1)

Length of Loop

First method-

If loop exists then the address containing in P and Q are same. Now move P two steps, Q one step and increment count. Repeat this step till P does not become equal to Q and return count.

Second Method- Check until Q → Next not equal to P and increment count.

```
ALGORITHM LoopLength(START)
BEGIN:
        P = START
        Q = START
        Count = 0
        WHILE P != NULL and P→Next != NULL DO
            Q = Q→ Next
            P = P→ Next → Next
        IF P==Q THEN
            BREAK

        P=P → Next → Next    //first Method
        Q=Q → Next
        Count = Count +1
        WHILE P != Q DO
            P=P → Next → Next
        Q=Q → Next
        Count++
        RETURN count
```

END;

SECOND METHOD
ALGORITHM LoopLength(START)
BEGIN:

```
    P = START
    Q = START
    Count = 0
    WHILE P != NULL and P → Next != NULL DO
        Q = Q → Next
        P = P → Next → Next
        IF P==Q THEN
            BREAK
    Q=Q->Next
    Count++
    WHILE Q!=P//second method
        Q = Q → Next
        Count = Count + 1
    RETURN Count
END;
```

Time Complexity O(N)
Takes O(N) time because traversal of the list linearly and constant space.
Space Complexity O(1)

<u>Start Point of loop</u>

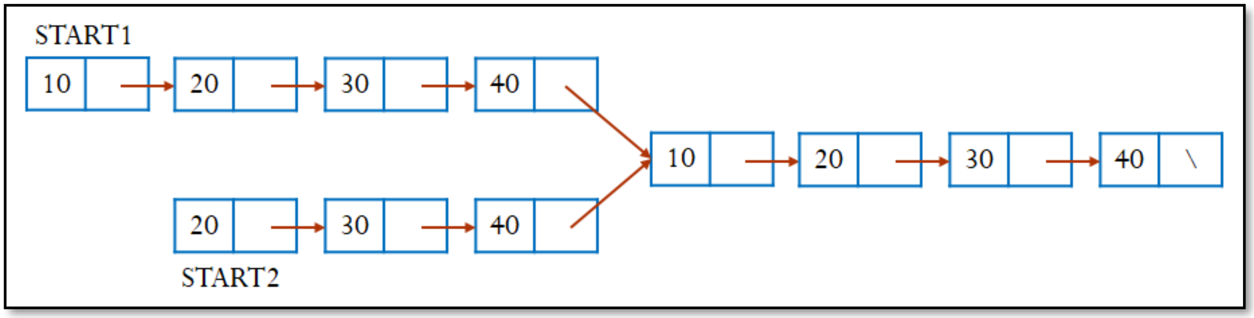
```
ALGORITHM StartPoint(START)
BEGIN:
        P = START
        Q = START
        Initialize count = 0
        While P != NULL and P → Next != NULL DO
                Q = Q → Next
                P = P → Next → Next
        IF P == Q THEN
        BREAK

        P = START
        WHILE P != Q DO
                Q = Q → Next
                P = P → Next
        RETURN P
END;
```

9.8.15 Merging Point of Two link list

Let us suppose that given two link list that merge at certain point that is shown in diagram below.

1. Find the number of unique elements in each list.
2. Find number of nodes common in each link list.



Solution

1. let us suppose that there are (m + c) nodes in first list and (n + c) nodes in second list.
2. c denotes last nodes common in each list.
3. calculate the number of nodes in each list by traversing them linearly. They have m + c and n + c nodes respectively.
4. take difference of two list and this gives excess node in larger list.

5. Move START1 by this difference.
6. now move START1 and START2 forward in a loop and compare the address of nodes they are pointing.

```
ALGORITHM merging Point (START1, START2)
BEGIN:
    P = START1
    Q=START2
    C1=0
    C2=0
    WHILE P! =NULL DO
        P=P → Next
        C1 = C1+1
    WHILE Q! =NULL DO
        Q=Q → Next
        C2 = C2+1
    P = START1
    Q=START2
    IF C1>C2 THEN
        FOR i= 0 TO c1-c2 DO
            P=P → Next
        FOR i= 0 TO c2-c1 DO
            Q=Q → Next
    WHILE P! = Q AND P! =NULL AND Q! =NULL DO
        P =P → Next
        Q = Q → Next
    RETURN P
END;
```

Complexity- takes O(N) time because travers the list linearly and constant space.

9.8.16 Palindrome

Write an algorithm to check if a link list is palindrome or not.

Following list are palindrome

1 → 2 → 3 → 3 → 3 → 2 → 1

M → A→L→A→Y→A→L→A→M

Constraints – Function should take O(N)Time

Solution

The simplest solution is that to take reverse function.

Solution

1. first find middle of link list
2. then compare first half and second half of link list.
3. finally, again reverse the second half of list to store original one.
4. Total complexity O(N)

```
ALGORITHM Palindrome(START)
BEGIN:
P = START
Ispalindrome = TRUE
mid = middle(P)
Q=mid
reverse(mid)
//if number of elements is odd then let first half have the extra element
//check if corresponding elements of the list pointed by START and mid are equal
WHILE mid != NULL DO
        IF mid→info != P→info THEN
                Ispalindrome= FALSE

        P =P → Next
        mid=mid → Next
  reverse(Q)
END;
```

Complexity- this algorithm takes O(N) time complexity because

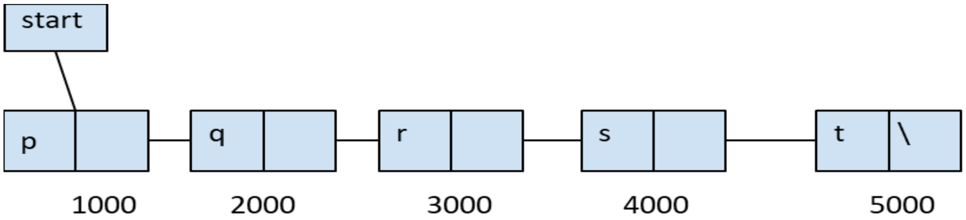
1. first find middle of link list in O(N/2) time
2. then compare first half and second half of link list O(N/2)
3. finally, again reverse the list to store original one O(N/2),

Total complexity O(N)

Multiple Choice Questions

1	What will be the complexity for finding the 7th element from starting in a single linked list? Let n be the number of nodes in Linked List and n > 12 ?
A	O(1)
B	O(n)
C	O(log n)

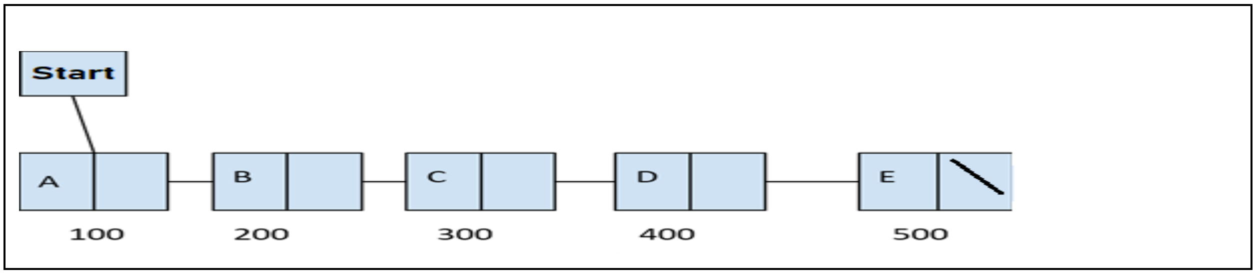
D	None of these
AN	**A**
DL	**M**
TP	

2	What will be the complexity for finding the 8th element from the end in a singly linked list? Let n be the number of nodes in Linked List and n > 20?
A	O(1)
B	O(n)
C	O(log n)
D	None of these
AN	**B**
DL	**M**
TP	
3	How many pointer updates will be required if we are inserting a new node after the second node?  start p q r s t \ 1000 2000 3000 4000 5000
A	One pointer
B	Two pointers
C	Three pointers
D	None of these
AN	**B**
DL	**E**

Scenario

Consider a linked list given below, based upon the pseudocode given, answer the following questions from 5 to 10?

```
Line 1    struct node *p;
Line 2    p = start-> link -> link ;
Line 3    p-> link -> link = p ;
Line 4    start -> link = p-> link ;
```



4	Which node does the node “p” refer to after executing Line 2 of pseudocode?
A	A
B	C
C	B
D	D
AN	**B**
DL	**M**
TP	

5	What will be the address that the node “C” refers to?
A	3000
B	4000
C	5000
D	None of these
AN	**B**
DL	**E**
TP	

6	Which node link part will contain address 3000 after the end of pseudo code?
A	A
B	D
C	E
D	None of these
AN	**E**
DL	**M**
TP	

7	What address will be pointed by the node A after the overall changes made by pseudo code?
A	2000

B	3000
C	4000
D	5000
AN	**C**
DL	**M**
TP	

8	Which node does the node “p” refer to after executing the whole pseudocode?
A	C
B	B
C	D
D	E
AN	**A**
DL	**M**
TP	

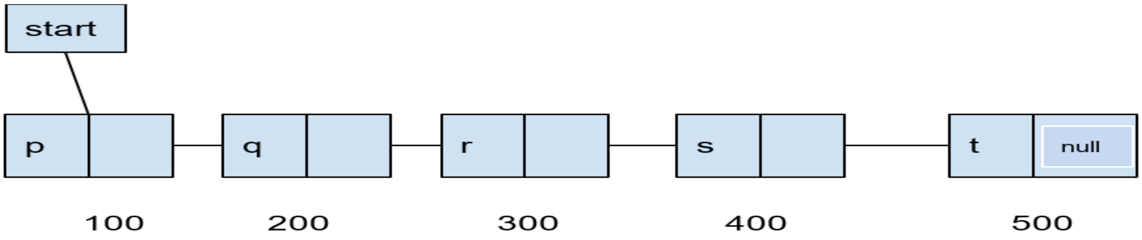
9	What will be the output of the following code? printf("%c" , start -> link ->link->link->link->data);
A	C
B	D
C	B
D	E
AN	**B**
DL	**D**
TP	

10	What will be the output of the following code? printf("%c" , p -> link ->link->link->link->data);
A	E
B	B
C	D
D	C
AN	**C**
DL	**M**

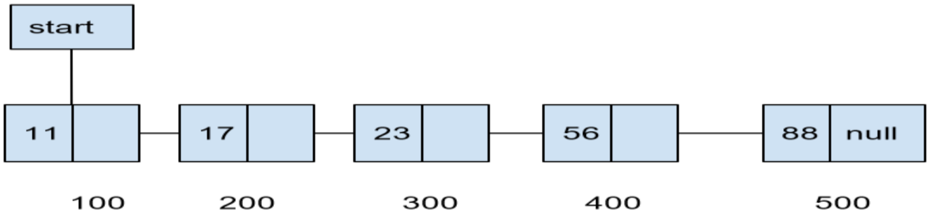
11	Let X be a single Linked List, let ‘p ‘ be a pointer to an intermediate node “Y” in a linked list , What would be the complexity to delete the node “Y” in the linked list ?

A	O (log(n))
B	O log(log (n))
C	O (n^3)
D	None of these
AN	**D**
DL	**M**

12	How many pointer updates will be required to delete the first node of the linked List in the best scenario?
A	One pointer
B	Two pointers
C	Three pointers
D	None of these
AN	**A**
DL	**M**

How many pointer updates will be required if we are inserting a new node after the second node?  start p q r s t null 100 200 300 400 500
One pointer
Two pointers
Three pointers
None of these
B
E

Scenario	Consider a Linked List given below, based upon the pseudocode given, answer the following questions? Line 1 struct node *p; Line 2 p = start→ link → link ; Line 3 p→ link → link = p ;

	Line 4 start → link = p→ link ;  start 11 17 23 56 88 null 100 200 300 400 500
2	Which node does the node “p” refer to after executing Line 2 of pseudocode
A	11
B	23
C	17
D	56
AN	**B**
DL	**M**
TP	
3	What will be the address that the node “23” refers to?
A	300
B	400
C	500
D	None of these
AN	**B**
DL	**E**
TP	

1	What is the time complexity to count the number of elements in the linked list?
A	O(1)
B	O(n^2)
C	O (log(n))
D	None of these
AN	**D**
DL	**M**
TP	

2	What is the output of the following function for starting pointing to the first node of the following linked list? 11 -> 12-> 13-> 14-> 15-> 16 void fun(struct node* start) {

	if(start == NULL)
	   return;
	   printf("%d ", start->data);
	   if(start->next != NULL)
	   fun(start->next->next);
	   printf("%d ", start->data);
	}

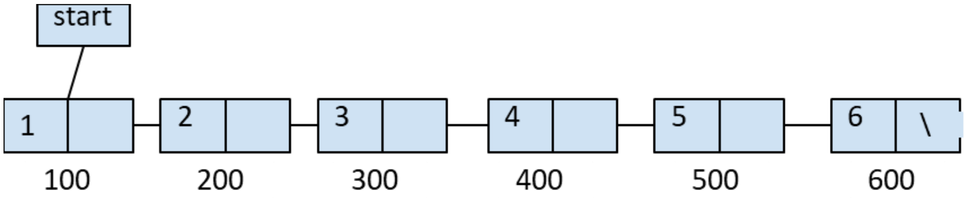
A	12 14 16
B	11 13 15 11 13 15
C	11 13 15 15 13 11
D	None of these
AN	**C**
DL	**M**

3	What does the following function do for a given Linked List with the first node as head?
	void fun1(struct node* head)
	{
	   if(head == NULL)
	   return;
	   fun1(head->next);
	   printf("%d ", head->data);
	}

A	Print all nodes
B	Print only even position nodes
C	Print only odd position nodes
D	None of these
AN	**D**
DL	**M**
TP	

4	What does the following function do for a given Linked List with the first node as head?
	void fun1(struct node* p)
	{
	   if(p)
	  {
	      printf("%d ", p -> data);
	      fun1(p -> next);

	} }
A	1 2 3 4
B	1 3 4
C	4 3 2 1
D	4 3 1
AN	**A**
DL	**M**

5	What does the following function do for a given Linked List with the first node as start and pointer p points to start only ?
	 start
	1
	2
	3
	4
	5
	6
	\
	100
	200
	300
	400
	500
	600
	void fun1(struct node* p)
	{
	  if(p)
	  {
	    fun1(p -> next);
	    printf("%d ", p -> data);
	  }
	}

A	6 6 6 6 6 6
B	1 3 5 6
C	6 5 4 3 2 1
D	4 3 1
AN	**C**
DL	**M**

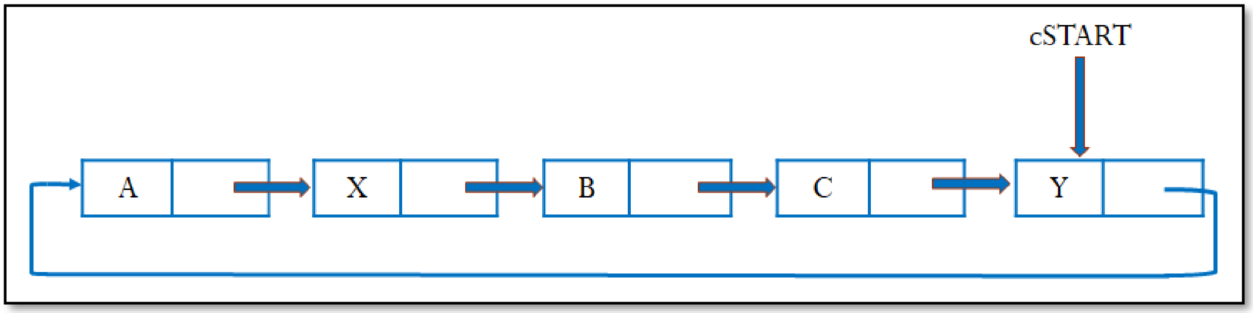
6	What does the following code do ?
	mystery(struct node *start)
	{
	 struct node *p , *q , *r;
	 p = start;
	 q = NULL;

	while (p! = NULL)
	{
	r = q;
	q = p;
	p = p-> next;
	q->next = r;
	}
	start = q;
	}

A	Swapping the even number of node
B	Swapping odd number of node
C	Swapping consecutive position node
D	None of these
AN	**D**
DL	**D (reversing)**

9.9 Circular Linked List

Circular Linked List is more like Linear Linked List except that the address field of the last node contains the address of the first node. START keeps the address of the last node in the circular Linked List.



9.9.1 Circular Linked List Operations

9.9.1.1 Insertions

Insertion at the beginning

```
ALGORITHM InsBeg(cSTART, item)
BEGIN:
        P=GetNode()
        P→info = item
        IF cSTART!=NULL THEN
                P→Next=cSTART→Next
                cSTART→Next=P
        ELSE
                P→Next=P
                cSTART=P
END;
```

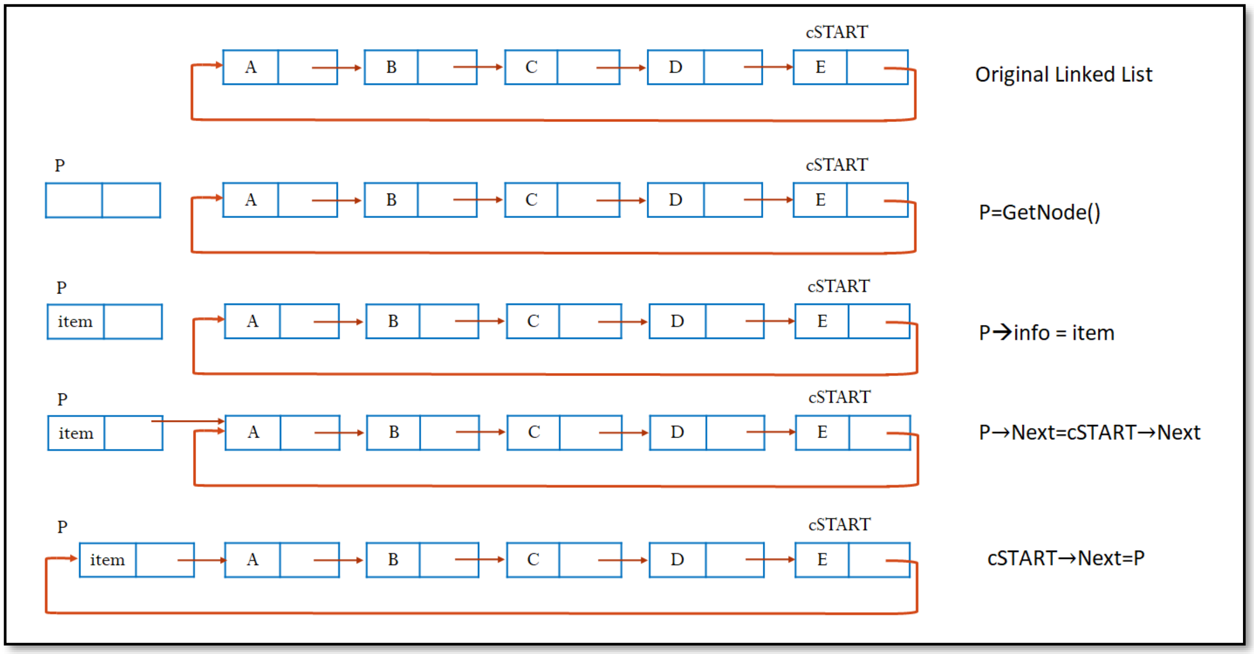
Time Complexity: O(1)

Here, in any case 4 statements are executed

Space Complexity: O(1)

An extra variable **P** is used and the space is allocated in the memory for the new node. Total extra space allocated is constant

No of Link adjustments: 2



Insertion at the End

```
ALGORITHM InsEnd(cSTART, item)
BEGIN:
        P=GetNode()
        P→info = item
        IF cSTART!=NULL THEN
                P→Next=cSTART→Next
                cSTART→Next=P
```

```
          ELSE
                    P→Next=P
                    cSTART=P
END;
```

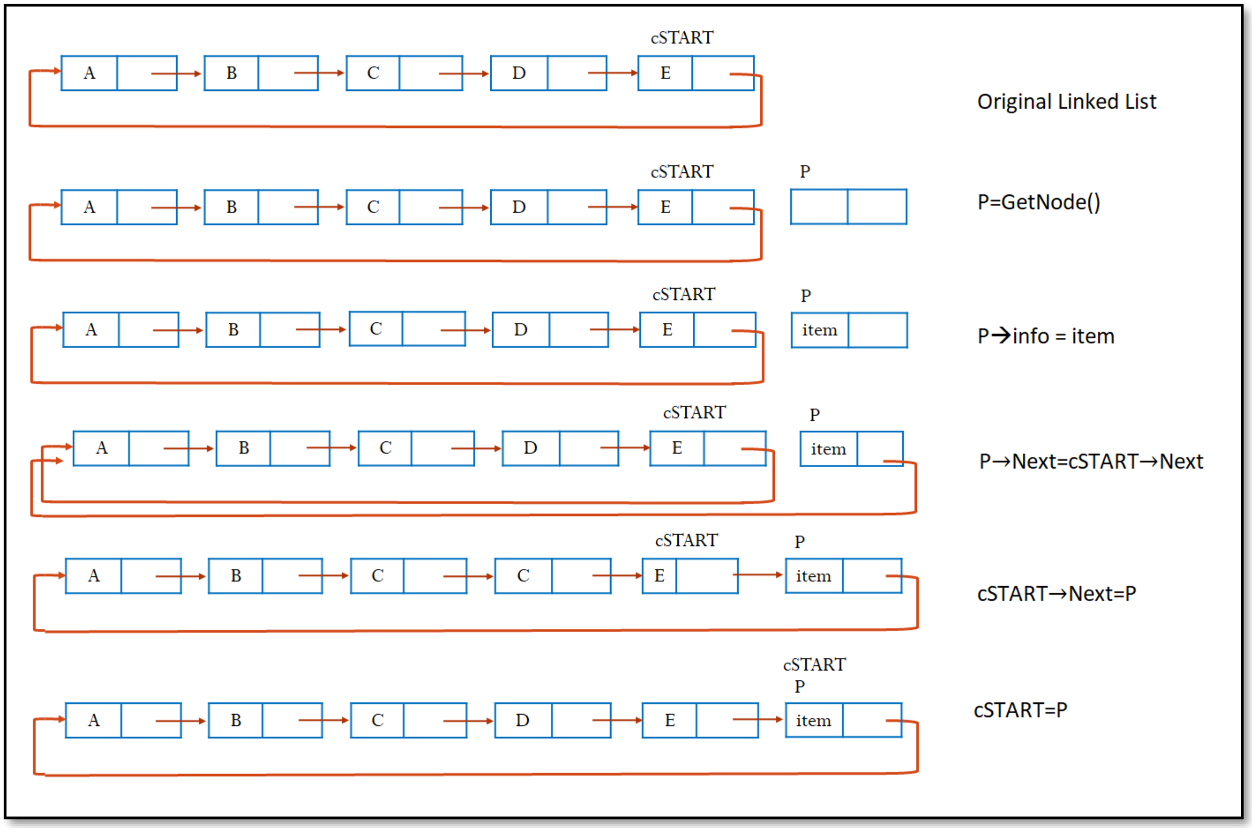
Time Complexity: O(1)

Here, in any case 5 statements are executed

Space Complexity: O(1)

An extra variable **P** is used and the space is allocated in the memory for the new node. Total extra space allocated is constant

No of Link adjustments: 2



Insertion after the given Node

```
ALGORITHM InsAfter(Q, item)
BEGIN:
          IF Q==NULL THEN
                    WRITE("Void Insertion")
          ELSE
                    P = GetNode()
```

```
                P→info = Item
                P→Next=Q→Next
                Q→Next=P
END;
```

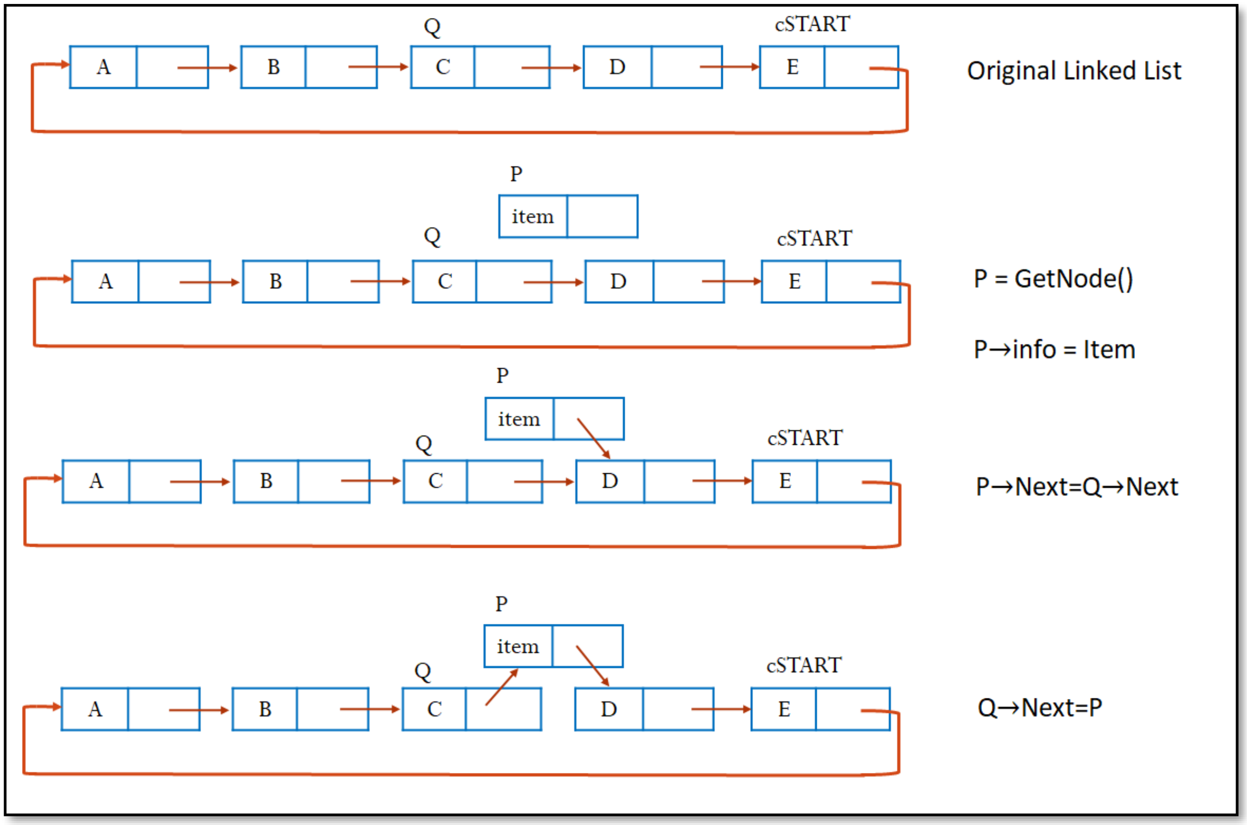
Time Complexity: O(1)

Here, in any case 4 statements are executed

Space Complexity: O(1)

An extra variable **P** is used and the space is allocated in the memory for the new node. Total extra space allocated is constant

No of Link adjustments: 2



9.1.1.2 Deletion Operations

Deletion at the Beginning

```
ALGORITHM DelBeg(cSTART)
BEGIN:
        IF cSTART==NULL THEN
                WRITE("Void Deletion")
        ELSE
                P=cSTART→Next
```

```
                    IF P→Next == P THEN
                              cSTART=NULL
                    ELSE
                              cSTART→Next=P→Next
                    item=P→Info
                    FreeNode(P)
                    RETURN item
END;
```

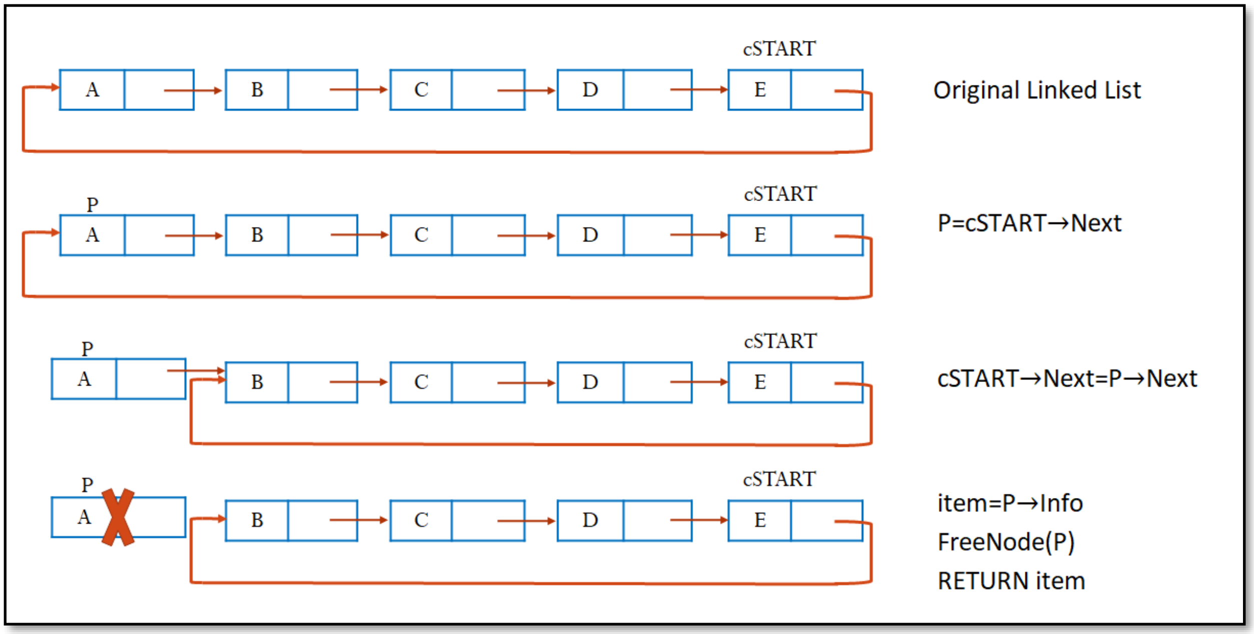
Time Complexity: Θ(1)

It takes only two statements when the Circular Linked List is empty and 6 otherwise. In both of the above cases, the number of statement execution is fixed i.e. constant

Space Complexity: Θ(1)

Two extra variables **P** & **item** are utilized, which is a constant number.

No of Link adjustments: 1



Deletion at End

ALGORITHM DelEnd(cSTART)

BEGIN:

```
          IF cSTART==NULL THEN
                    WRITE("Void Deletion")
          ELSE
                    P=cSTART→Next
                    WHILE P→Next!=cSTART DO
                              P=P→Next
                    P→Next=cSTART→Next
                    Q=cSTART
                    IF cSTART→Next!=cSTART
                              cSTART=P
                    ELSE
                              cSTART=NULL
                              item=Q→Info
               FreeNode(Q)
          RETURN item
          END;
```

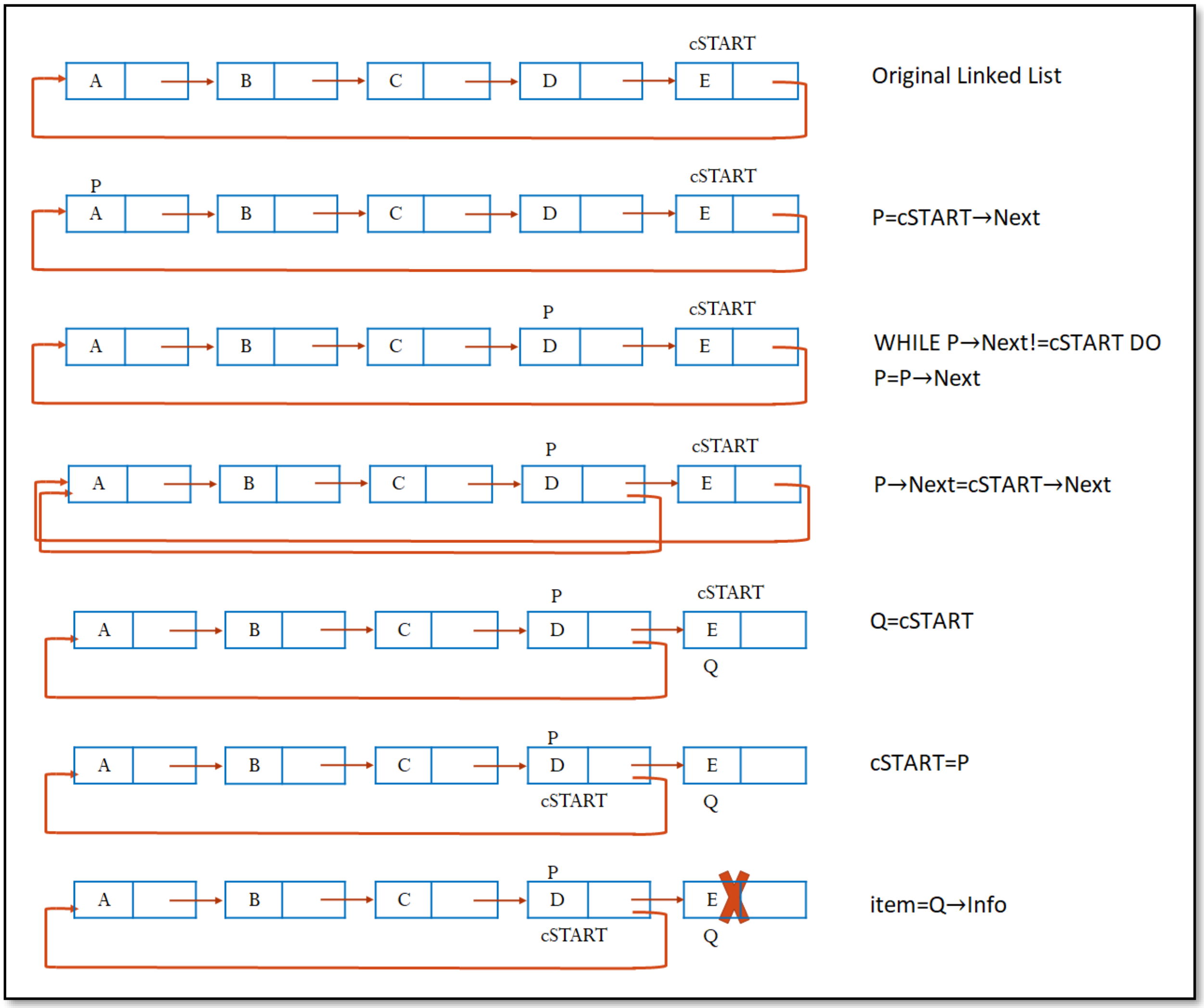
Time Complexity: O(N)

Traversal of circular Linked List requires O(N) Time.

Space Complexity:O(1)

Two extra variables **P** & **item** are utilized, which is a constant number.

No of Link adjustments: 1



Deletion after the given node

```
ALGORITHM DelAfter(P)
BEGIN:
        IF P == NULL OR P→Next == P THEN
                WRITE("Void Deletion")
        ELSE
                Q=P→Next
                P→Next=Q→Next
                item=Q→Info
                FreeNode(Q)
                RETURN item
END;
```

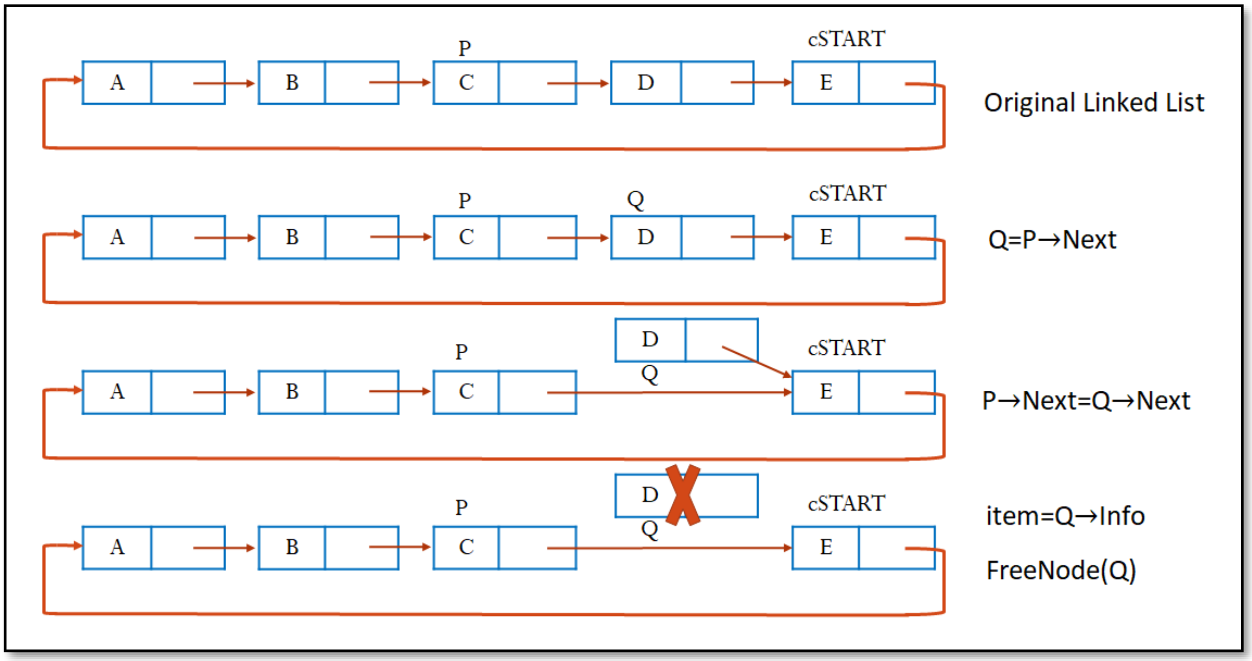
Time Complexity: Θ(1)

It takes only two statements when the Circular Linked List is empty and 5 otherwise. In both of the above cases, the number of statement execution is fixed i.e. constant

Space Complexity:Θ(1)

Two extra variables **P** & **item** are utilized, which is a constant number.

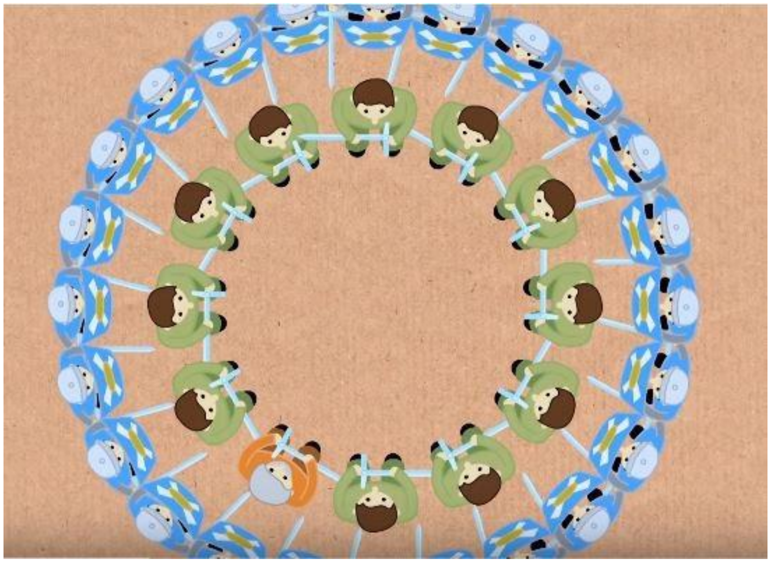
No of Link adjustments: 1



9.10 Application of Circular Linked List

9.10.1 Josephus Problem:

In the first century, Flavius Josephus, a Jewish historian and captain of Jewish Army writes about a scenario. Josephus and his 100 soldiers were trapped in a cave by Roman soldiers. They chose suicide over capture, and decided about a serial method of committing suicide in a specified manner (counting from 1 to 13 and 13th man to commit suicide). Josephus states that by luck or possibly by the grace of god, he survived until the end of this and surrendered to the Romans rather than killing himself.



An arrangement of soldiers (nodes) in a circle (Circular Linked List) and deletion of 13th node continuously until a single node is left out will solve this problem of one who remains till the end. Making it more general, let's consider deletion of kth node in a s

```
ALGORITHM Josephus (cSTART, k)
BEGIN:
    P = cSTART → Next
    WHILE P→Next!=P DO
        Count = 1
        WHILE Count ! = k-1 DO
            Count = Count + 1
            P = P→Next
        Q = P
        P = P→Next
        P = P →Next
        DelAfter(Q)
    RETURN P→Info
END;
```

9.11 Comparison of Linear Linked List and Circular Linked List Operations

Operations	Linear Linked List		Circular Linked List	
	Time Complexity	**Space Complexity**	**Time Complexity**	**Space Complexity**
Insertion at Beginning	O(1)	O(1)	O(1)	O(1)

Insertion in the Mid	Search Time + O(1)	O(1)	Search Time + O(1)	O(1)
Insertion at the End	O(n)	O(1)	O(1)	O(1)
Traversal	O(n)	O(1)	O(n)	O(1)
Deletion at Beginning	O(1)	O(1)	O(1)	O(1)
Deletion in the Mid	Search Time + O(1)	O(1)	Search Time + O(1)	O(1)
Deletion at the End	O(n)	O(1)	O(n)	O(1)

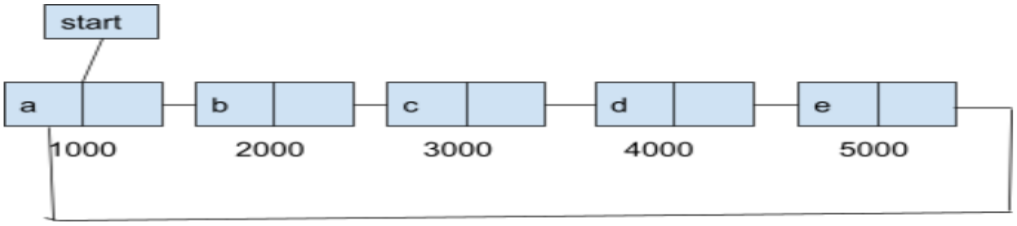
Multiple Choice Questions

1	Suppose we are inserting a node in a circular linked list in the end. How many least pointers updates are needed in this case?
A	1
B	2
C	3
D	4
AN	**B**
DL	**M**

2	Suppose we are inserting a node in a circular linked list in the first position. How many least pointers updates are needed in this case?
A	3
B	1
C	2
D	4
AN	**C**
DL	**M**

3	Which of the following will be the terminating condition while searching in a circular linked list? Assuming a pointer temp points to the starting of the linked list?
A	while (temp! = start)
B	while (temp -> next! = null)
C	while (temp! = null)
D	None of these
AN	**D**
DL	**M**

Scenar	Consider a linked list given below, based upon the pseudocode given, answer the

io	following questions from to? Line 1 struct node *p; Line 2 p = start-> link -> link ; Line 3 p-> link -> link = start-> link ; Line 4 start = start -> link ;  start a b c d e 1000 2000 3000 4000 5000
4	Which node does the “p” point to after the changes made in the linked list by pseudocode?
A	a
B	b
C	c
D	d
AN	
DL	

5	What will be the output of the following code after the changes made in the linked list by pseudocode? printf("%c" , start -> link ->link ->link->data);
A	a
B	b
C	e
D	d
AN	**C**
DL	**D**

6	What will be the address pointed by p-> link ->link -> link?
A	3000
B	2000
C	1000
D	None of these

AN	**B**
DL	**D**

7	How many nodes will be there after the execution of pseudocode in the given linked list?
A	3
B	2
C	4
D	5
AN	**C**
DL	**M**

8	What will be the output of the following code ? printf("%c" , p -> link ->link->link->link->data);
A	a
B	c
C	b
D	d
AN	**B**
DL	**D**

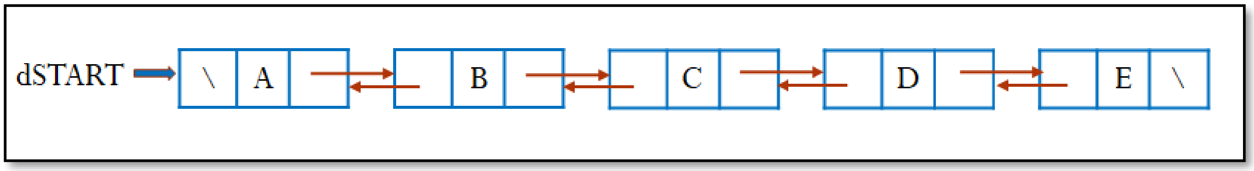
9.12 Doubly Linked List

9.12.1 Introduction

Doubly Linked List can be understood easily by the use case of “**music playlist**”. Here each song is connected to its previous song and to its next song in a playlist. The playlist is created with the first song as a starting point from where the music will be played and the last song next will be set to null as it would be the last song.

04. Deus Ex OST - UNATCO.flac (2:06)
06. Deus Ex OST - NYC Streets.flac (2:16)
07. Deus Ex OST - Lebedev's Airfield.flac (2:03)
11. Deus Ex OST - The Synapse (Hong Kong Streets).flac (2:16)
12. Deus Ex OST - Hong Kong Action.flac (0:59)
13. Deus Ex OST - Majestic 12 Labs.flac (1:49)
14. Deus Ex OST - Versalife.flac (1:35)

Each node in this type of the Linked List contains at 2 address fields (One for the address of previous node and other one for the address of the next node).



<u>**Notations used in Doubly Linked List**</u>

- For a node having address P

Prev	Info	Next

P→ Info
P→Next
P→Prev

9.12.2 Primitive operations

9.12.2.1 Insertion

Insertion at the Beginning

ALGORITHM InsBeg(dSTART,item)

```
BEGIN:
        P=GetNode()
        P→info=item
        P→Prev=NULL
        P→Next=dSTART
        IF dSTART!=NULL
        dSTART→Prev=P
    dSTART=P
END;
```

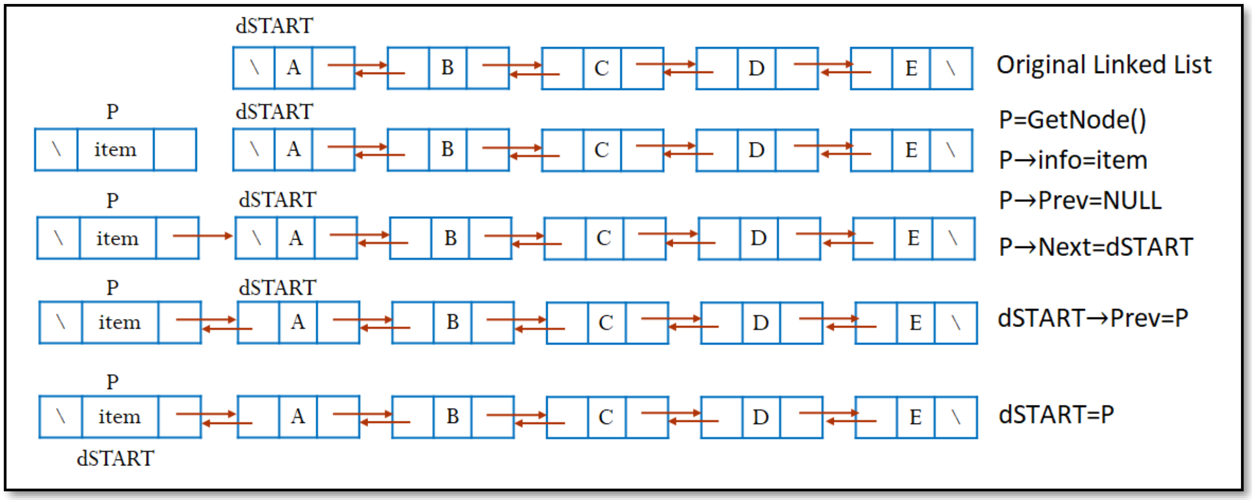
Time Complexity: O(1)

Here, in any case 6 statements are executed

Space Complexity: O(1)

An extra variable **P** is used and the space is allocated in the memory for the new node. Total extra space allocated is constant

No of Link adjustments: 4



Insertion at the End

```
ALGORITHM InsEnd(dSTART,item)
BEGIN:
    P=GetNode()
    P→info=item
    P→Next=NULL
    Q=dSTART
    IF dSTART!=NULL THEN
        WHILE Q→Next !=NULL
            Q=Q→Next
        Q→Next=P
        P→Prev=Q
    ELSE
        P→Prev=NULL
        dSTART=P
END;
```

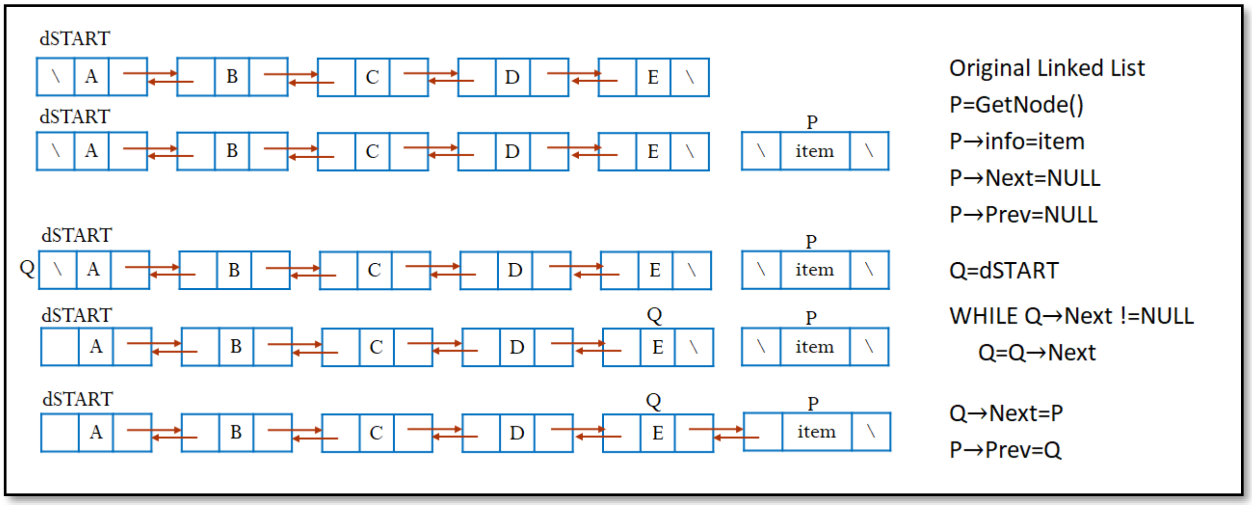
Time Complexity: O(N)

Traversal requires O(N) Time and 5 other statements. Total time required asymptotically is O(N)

Space Complexity: O(1)

An extra variable **P** is used and the space is allocated in the memory for the new node. Total extra space allocated is constant

No of Link adjustments: 3



Insertion before the given node

ALGORITHM InsBefore(Q, item)

BEGIN:

```
        IF Q==NULL THEN
                WRITE(“Void Insertion”);
        ELSE
                P=GetNode()
                P→info=item
                IF Q→ Prev!=NULL THEN
                        R=Q→Prev
                        R→Next=P
                        P→Prev=R
                ELSE
                        P→Prev=NULL
                        P→Next=Q
                        Q→Prev=P
END;
```

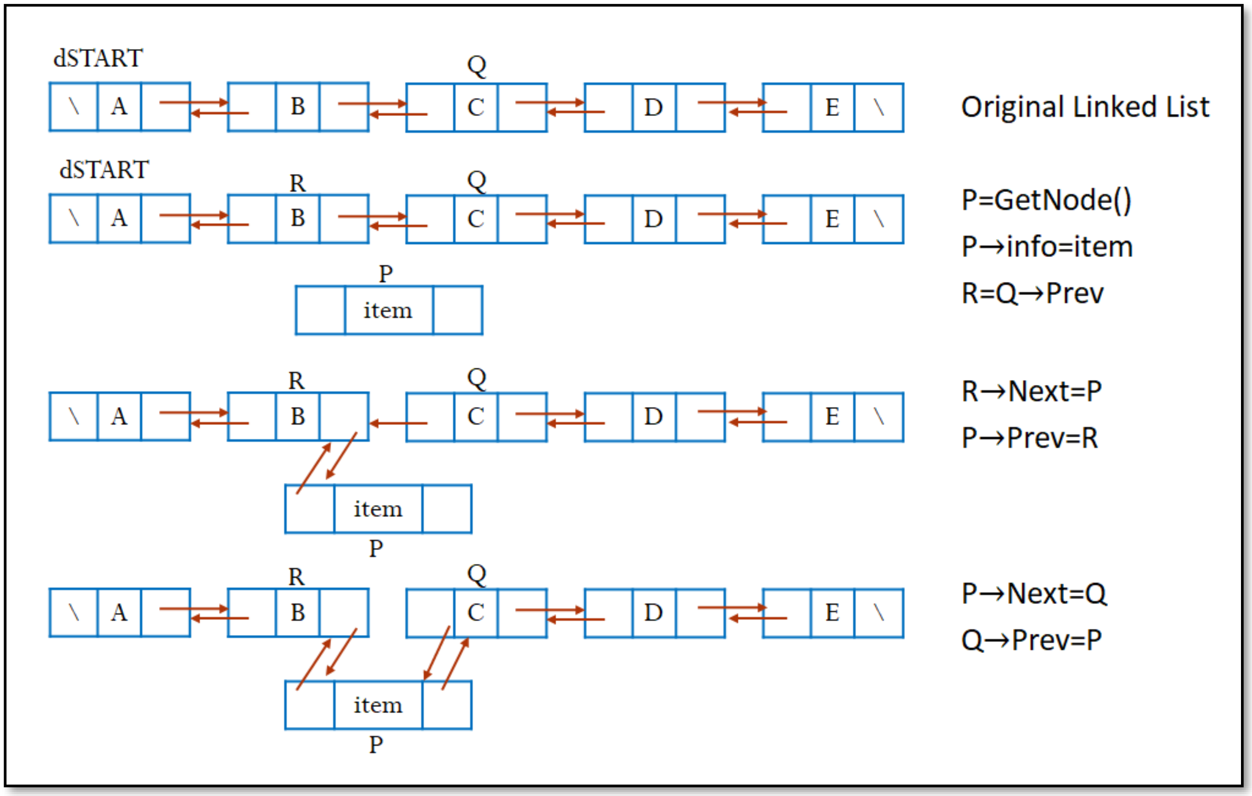
Time Complexity: O(1)

Here, in any case 7 statements are executed, which is a constant

Space Complexity: O(1)

An extra variable **P** is used and the space is allocated in the memory for the new node. Total extra space allocated is constant

No of Link adjustments: 4



Insertion after the given node

ALGORITHM InsAfter(Q, item)

BEGIN:

```
    IF Q==NULL THEN
        WRITE("Void Insertion");
    ELSE
        P=GetNode()
        P→info=item
        IF Q→ Next!=NULL THEN
            R=Q→Next
            R→Prev=P
            P→Next=R
        ELSE
            P→Next=NULL
```

```
                        Q→Next=P
                        P→Prev=Q
END;
```

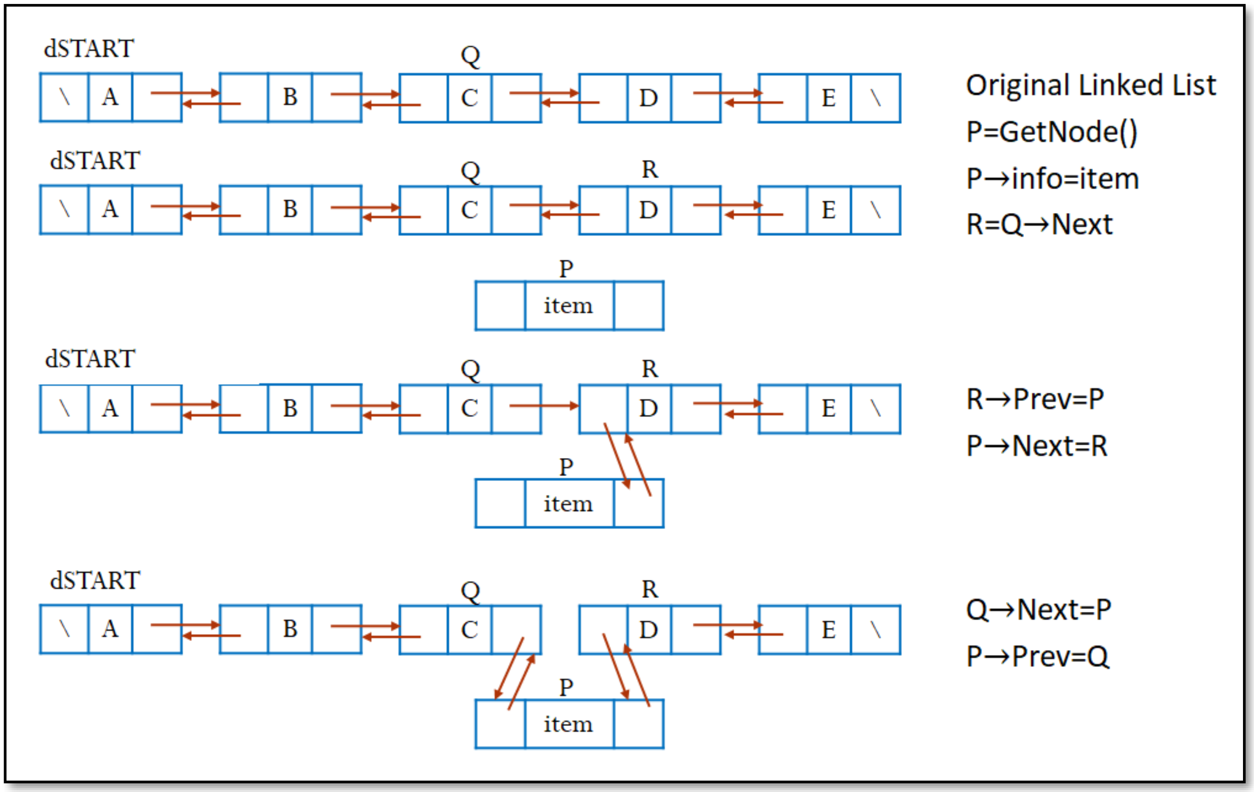
Time Complexity: O(1)

Here, in any case 7 statements are executed, which is a constant

Space Complexity: O(1)

An extra variable **P** is used and the space is allocated in the memory for the new node. Total extra space allocated is constant

No of Link adjustments: 4



9.12.2.1 Deletion

Deletion at the Beginning

```
ALGORITHM DelBeg(dSTART)
BEGIN:
        IF dSTART == NULL THEN
                WRITE("Void Deletion")
        ELSE
                P=dSTART
                Q=dSTART→Next
                IF Q!=NULL THEN
                    Q→Prev=NULL
                    dSTART=Q
                    item=dSTART→Info
                    FreeNode(P)
                    RETURN item
END;
```

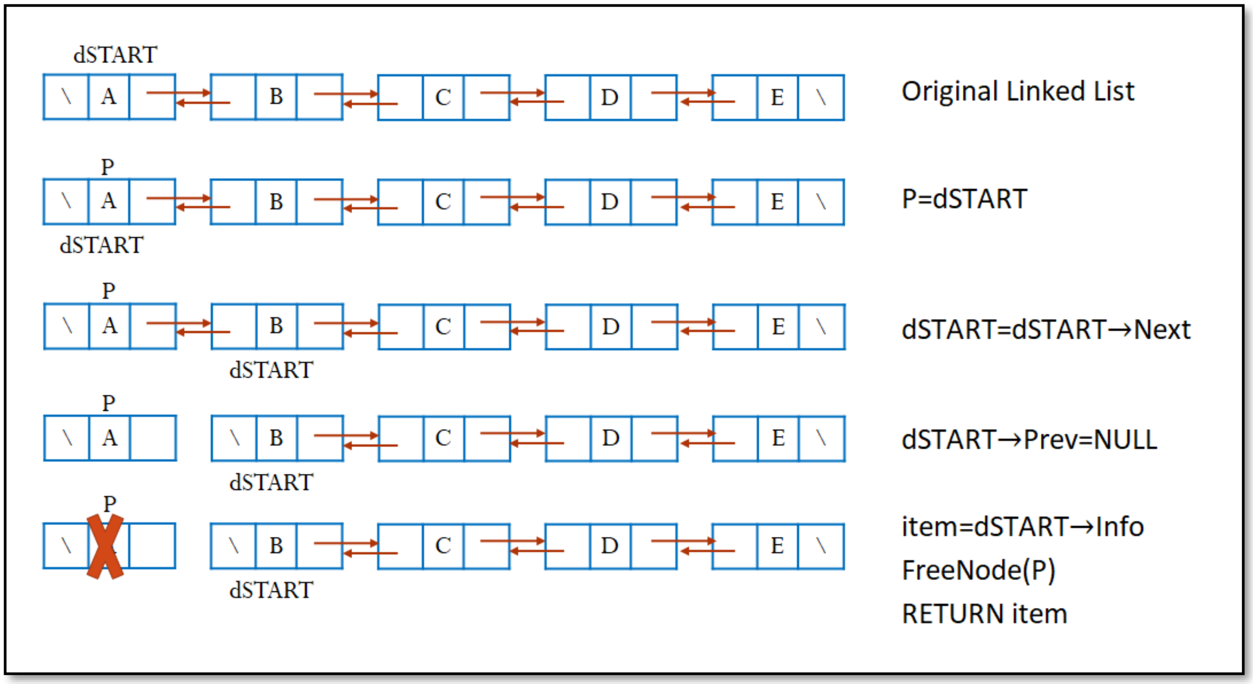
Time Complexity: O(1)

Here, in any case 7 statements are executed, which is a constant

Space Complexity: O(1)

Two extra variable **P, Q** are used. Total extra space allocated is constant

No of Link adjustments: 2



Deletion at the End

ALGORITHM DelEnd(dSTART)

BEGIN:

```
    IF dSTART==NULL THEN
        WRITE(“Void Deletion”)
    P=dSTART
    WHILE P→Next!=NULL THEN
        P=P→Next
        Q=P→Prev
        IF Q!=NULL THEN
                Q→Next=NULL
        ELSE
                dSTART=NULL
        Item=P→Info
        FreeNode(P)
        RETURN item
END;
```

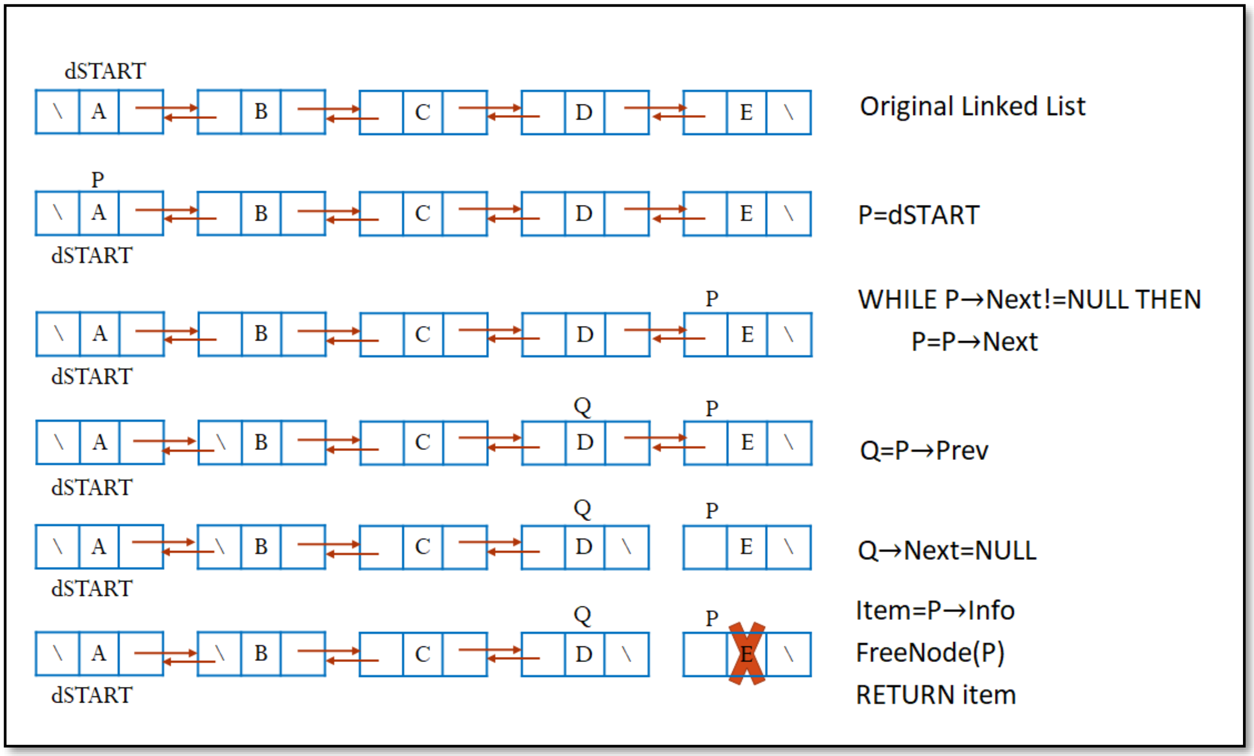
Time Complexity: O(N)

Traversal requires O(N) Time and 5 other statements. Total time required asymptotically is O(N)

Space Complexity: O(1)

Two extra variable **P, Q** are used. Total extra space allocated is constant

No of Link adjustments: 2



Deletion before the given node

ALGORITHM DelBefore(Q)

BEGIN:

```
        IF Q==NULL OR Q→prev==NULL THEN
                WRITE("Void Deletion")
        ELSE
                P=Q→left
                R=P→left
                Q→prev=R
                R→next=Q
                Item=P→Info
                FreeNode(P)
                RETURN item
        END;
```

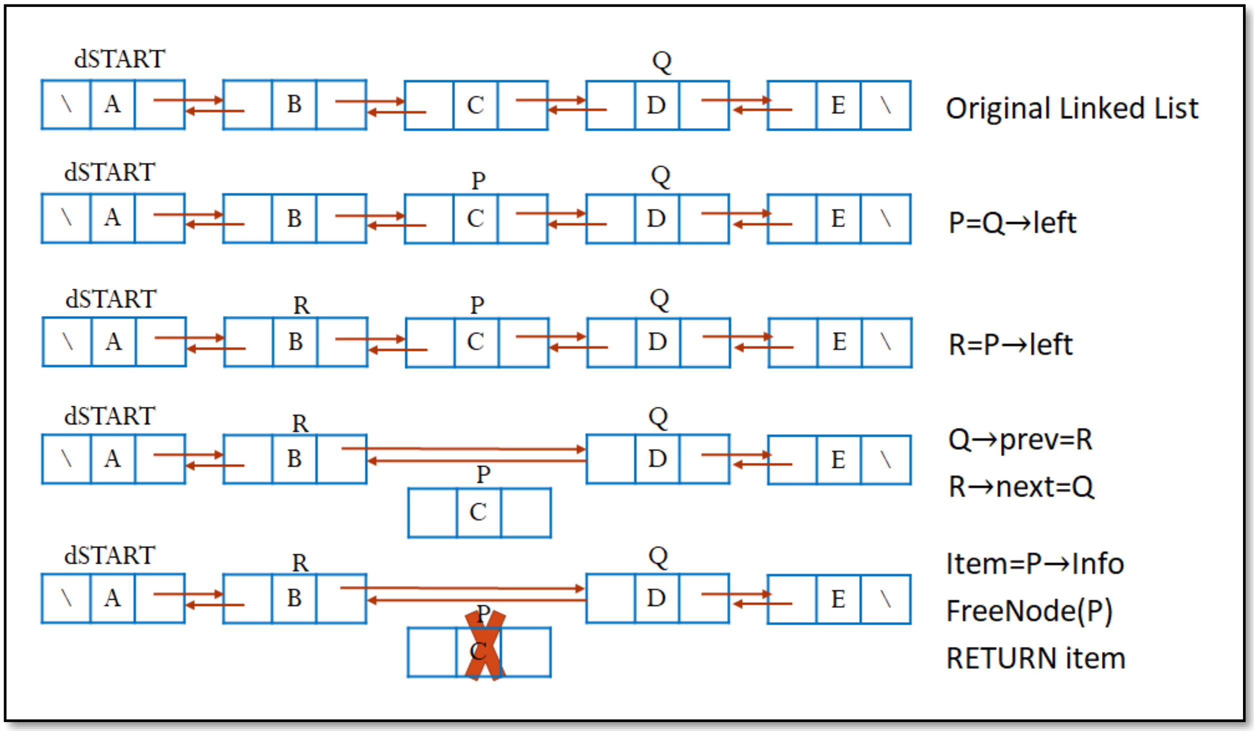
Time Complexity: O(1)

Total of 7 statement execution. Constant.

Space Complexity: O(1)

Two extra variable **P, Q** are used. Total extra space allocated is constant

No of Link adjustments: 2



Deletion after the given node

ALGORITHM DelAfter(Q)

BEGIN:

```
    IF Q==NULL OR Q→Next==NULL THEN
        WRITE("Void Deletion")
    ELSE
        P=Q→Next
        R=P→Next
```

```
            Q→Next=R
            R→Prev=Q
            Item=P→Info
            FreeNode(P)
            RETURN item
END;
```

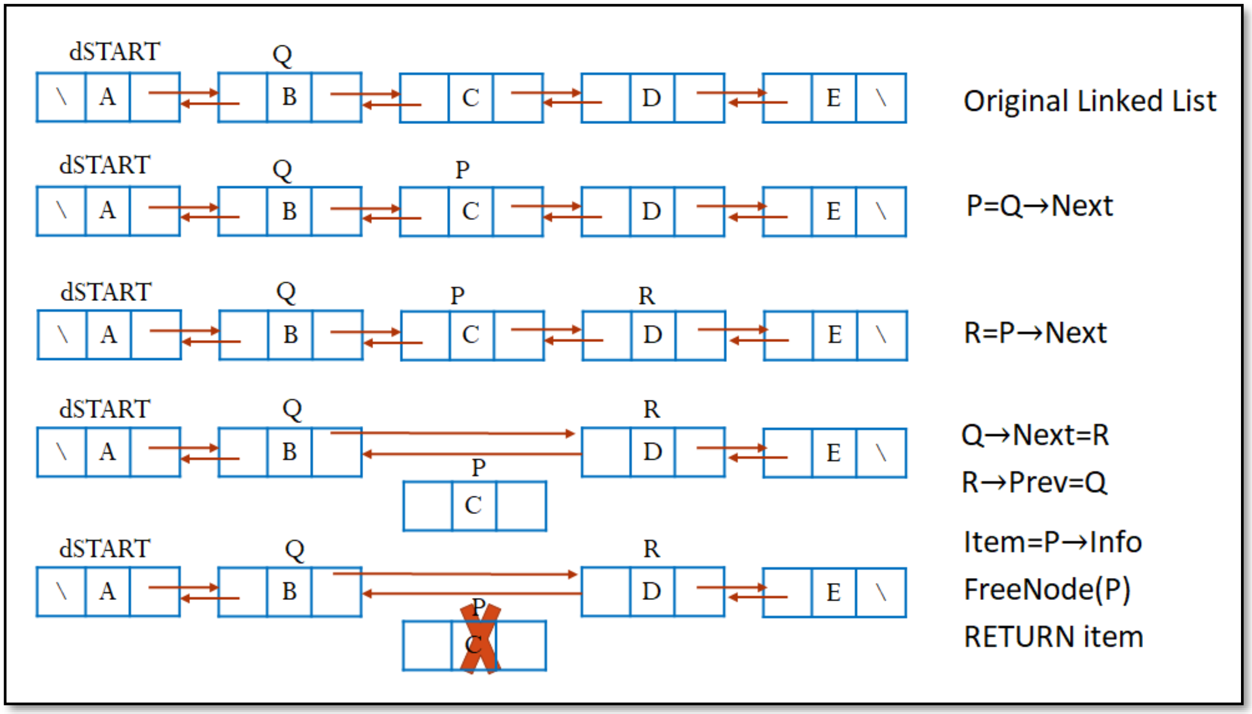
Time Complexity: O(1)

Total of 7 statement execution. Constant.

Space Complexity: O(1)

Two extra variable **P, Q** are used. Total extra space allocated is constant

No of Link adjustments: 2



9.12.2.3 Traversal

ALGORITHM Traverse(dSTART)

BEGIN:

```
      P=dSTART
      WHILE P!=NULL DO
      WRITE (P→Info)
      P=P→Next
END;
```

Time Complexity: O(N)

Traversal requires O(N) Time and 1 other statement. Total time required asymptotically is O(N)

Space Complexity: O(1)

Two extra variable **P** is used. Total extra space allocated is constant

No of Link adjustments: 0

Multiple Choice Questions

1	How many pointer updates will be required, if we are inserting a node in the beginning of some given Doubly singly Linked List?
A	3
B	4
C	5
D	2
AN	**B**
DL	**M**

2	What would be the complexity of inserting a new node after a node with a pointer "p" pointed towards it in a singly doubly linked list?
A	O(n)
B	O(1)
C	Olog(n)
D	None of these
AN	**B**
DL	**M**

3	What would be the complexity of inserting a new node at a starting position in a singly doubly linked list?
A	O(n)
B	O(1)
C	Olog(n)
D	None of these
AN	**B**
DL	**M**

4	What would be the complexity of inserting a new node at the end in a singly doubly linked list?
A	O(n)
B	O(1)

C	O log(n)
D	None of these
AN	**A**
DL	**M**

9.13 Circular Doubly Linked List

9.13.1 Introduction

Doubly Linked List can be understood easily by the use case of “**music playlist”**. Here each song is connected to its previous song and to its next song in a playlist. The playlist is created with the first song as a starting point from where the music will be played and the last song next will be set to take address of first song. Here there exist a cycle where last song is connected to first song.

Nulloy 0.5 Alpha

Alexander Brandon - The Synapse (Hong Kong Streets)

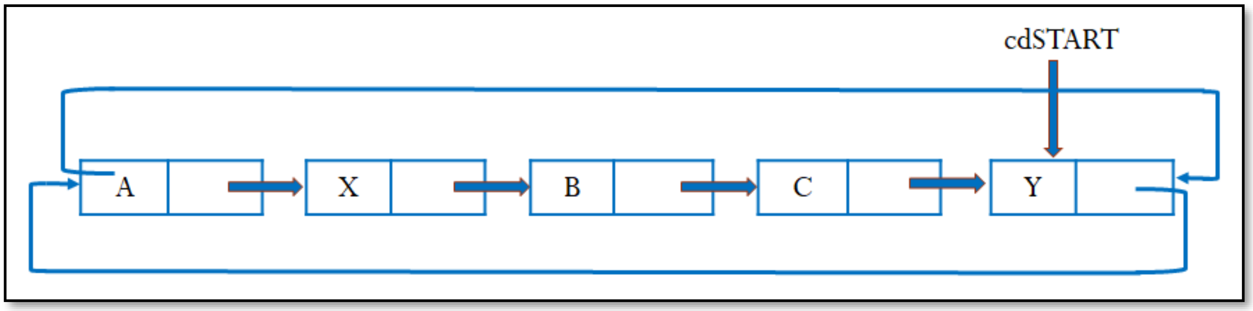
1:01/2:16

04. Deus Ex OST - UNATCO.flac (2:06)
06. Deus Ex OST - NYC Streets.flac (2:16)
07. Deus Ex OST - Lebedev's Airfield.flac (2:03)
11. Deus Ex OST - The Synapse (Hong Kong Streets).flac (2:16)
12. Deus Ex OST - Hong Kong Action.flac (0:59)
13. Deus Ex OST - Majestic 12 Labs.flac (1:49)
14. Deus Ex OST - Versalife.flac (1:35)

Circular Doubly Linked List is the collection of nodes where each node has at least three field

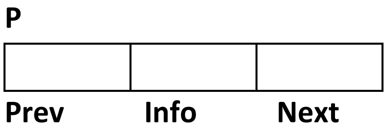
- **Next Address field-** It contains the address of next node in the list.
- **Previous Address field-** It contains the address of previous node in the list.
- **Information Field-** It contains the actual data to be stored.

In this list Next address field of the last node contains the address of the first node and previous address field of the last node contains the address of last node. No node contains NULL. Also START variable contains the address of the first node in the list.



<u>Notations used in Circular Doubly Linked List</u>

For a node having address P



-the fields are accessed as:

P→ Info

P→Next

P →Prev

- **GetNode( )** is used of allocation of memory for new node
- **START** is used for keeping the address of First node. In case the Linked List is empty, START keeps a NULL.

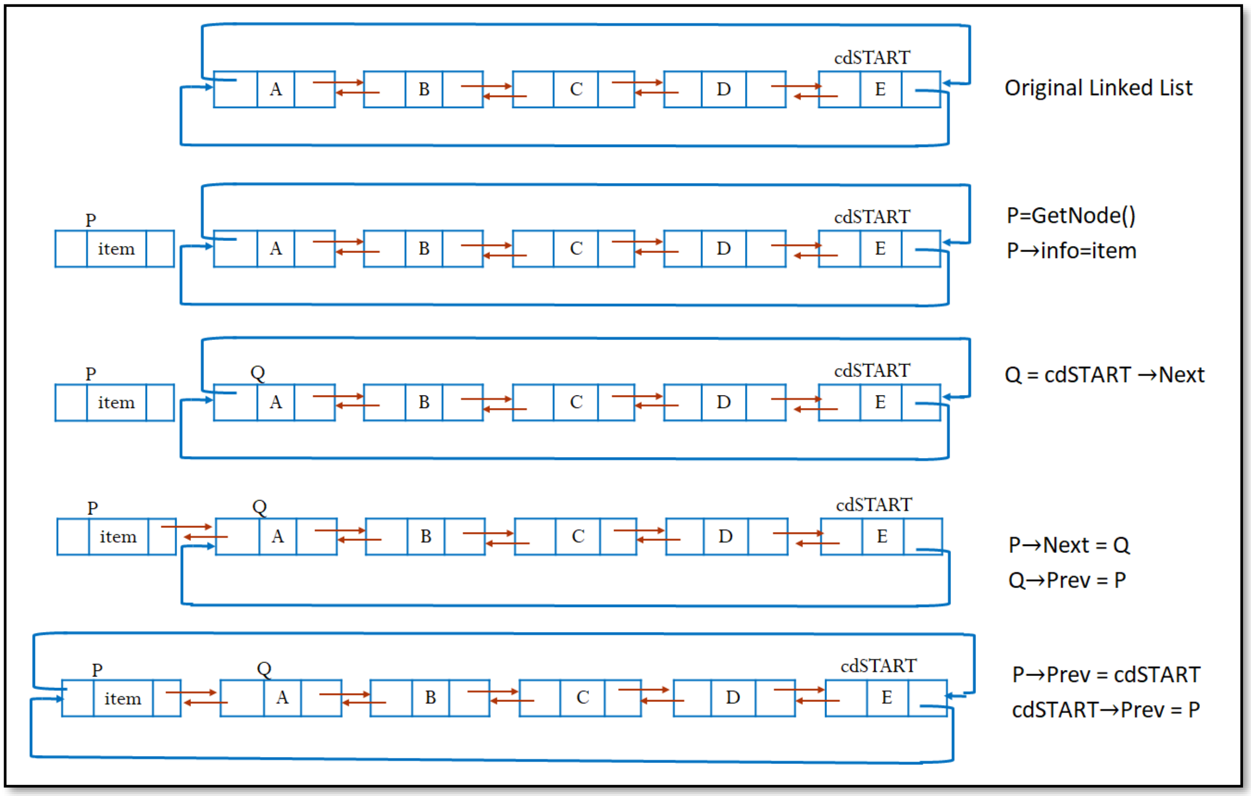
9.13.2 Circular Doubly Linked List Operations

9.13.2.1 Insertion

Insertion at the Beginning

This algorithm inserts an item as first node in the circular doubly Linked List. There are two scenarios of inserting a node in circular doubly Linked List at beginning. Either the list is empty or it contains more than one element in the list. The insertion in these two cases can be done by the following given algorithm.

```
ALGORITHM InsBeg(cdSTART,item)
BEGIN:
        P=GetNode()
        P→info=item
        IF cdSTART == NULL THEN
                P→Prev = P
                P→Next = P
        ELSE
                Q = cdSTART →Next
                P→Next = Q
                Q→Prev = P
                P→Prev = cdSTART
                cdSTART→Prev = P
    END;
```



Time Complexity: O(1)

Here, in any case 6 statements are executed

Space Complexity: O(1)

An extra variable **P** is used and the space is allocated in the memory for the new node. Total extra space allocated is constant

No of Link adjustments: 4

<u>Insert at the End</u>

This algorithm inserts an item as last node in the circular doubly Linked List. There are two scenarios of inserting a node in circular doubly Linked List at beginning. Either the list is empty or it contains more than one element in the list. The insertion in these two cases can be done by the following given algorithm.

If we need to insert at end (consisting one or more nodes) than firstly, we find the last node by traversing the list, the last node would be one whose next part consists the address of first node. Then we can simply update some pointers for insertion.

```
ALGORITHM InsEnd(cdSTART, item)
BEGIN:
P = GetNode()
P→info = Item

IF cdSTART==NULL THEN
        P→Next = P
        P →Prev = P
        cdSTART = P

ELSE
        Q = cdSTART →Next
        cdSTART→Next = P
        P→Prev = cdSTART
        P → Next = Q
        Q→ Prev = P
        cdSTART = P
END;
```

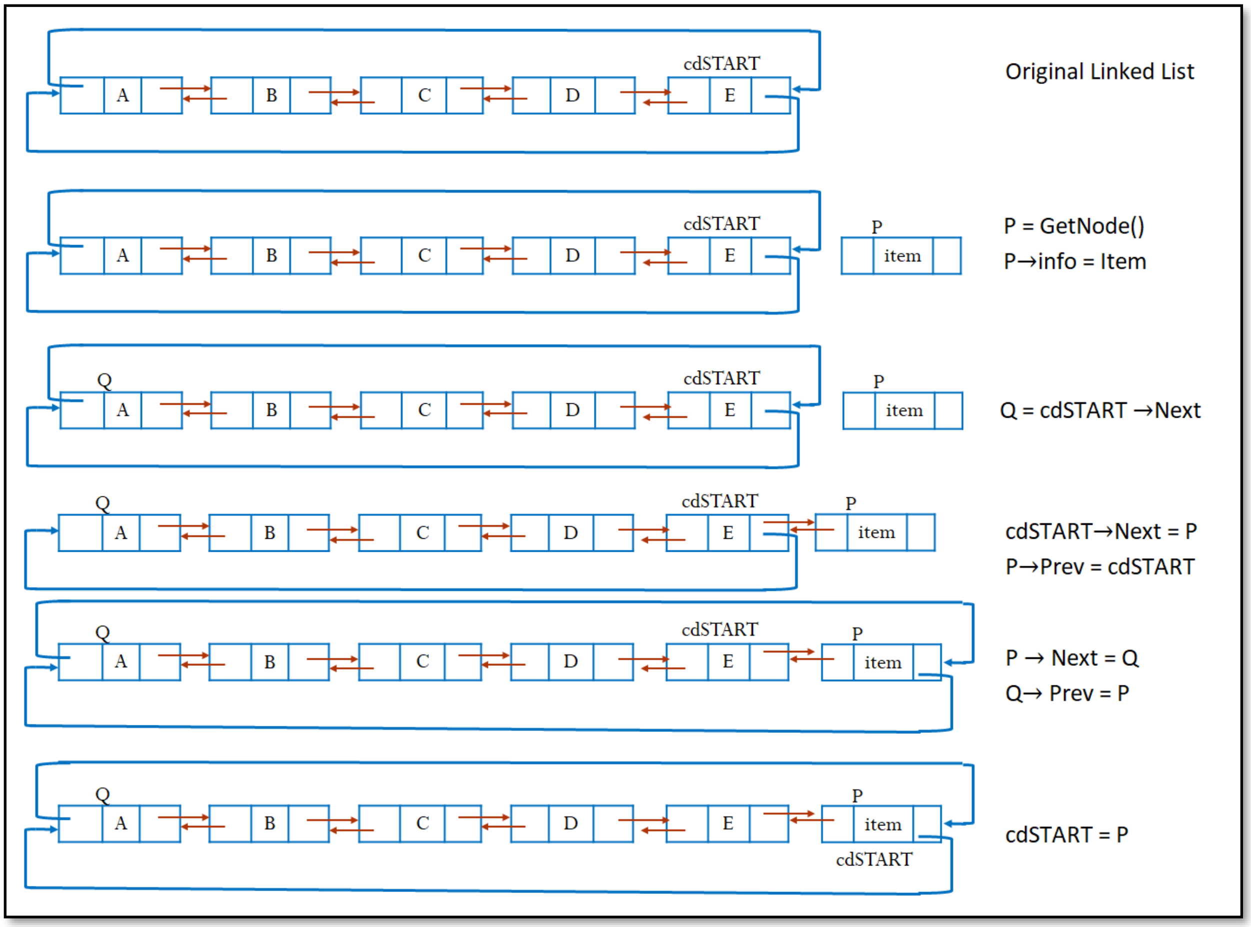
Time Complexity: O(n)

Here, traversal is required to reach to the last node.

Space Complexity: O(1)

An extra variable **P** is used and the space is allocated in the memory for the new node. Total extra space allocated is constant

No of Link adjustments: 4



Original Linked List

P = GetNode()
P→info = Item

Q = cdSTART →Next

cdSTART→Next = P
P→Prev = cdSTART

P → Next = Q
Q→ Prev = P

cdSTART = P

Insertion after the given Node

This algorithm inserts an item at some position in the circular doubly Linked List. There is one scenario of inserting a node in circular doubly Linked List at given position. If we need to insert a node after some specific position (consisting one or more nodes) in the list than firstly, we find the location of that node after which we can perform insertion.

ALGORITHM InsAfter(Q, item)
BEGIN:

```
        IF Q==NULL THEN
                WRITE("Void Insertion")
        ELSE
                P = GetNode()
                P→info = Item
                R=Q→Next
```

```
            Q→Next = P
            P→Prev = Q
            R→ Prev = P
            P→Next = R
END;
```

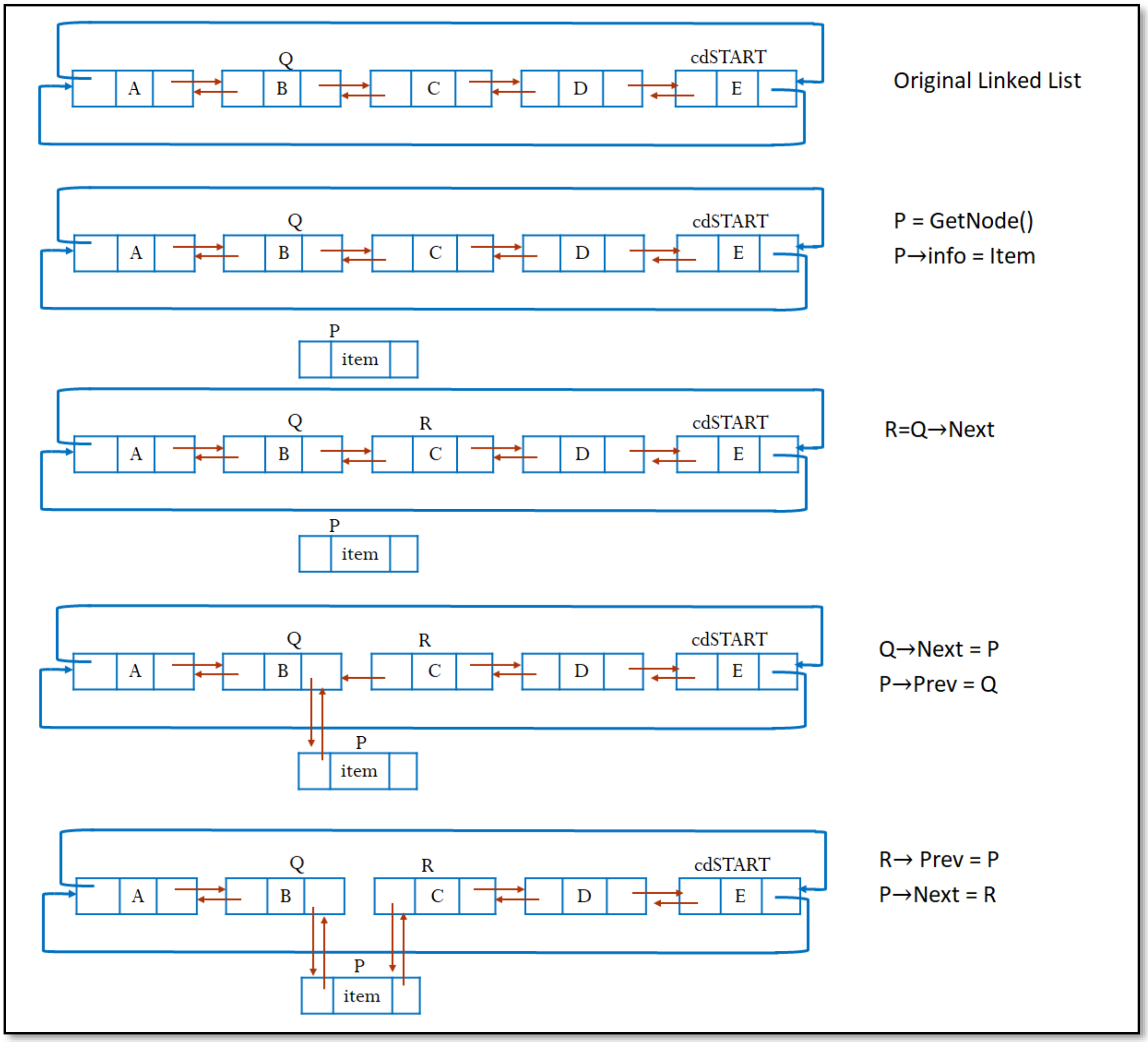
Time Complexity: O(1)

Here, traversal is not required because of first node.

Space Complexity: O(1)

An extra variable **P** is used and the space is allocated in the memory for the new node. Total extra space allocated is constant

No of Link adjustments: 4



Original Linked List

P = GetNode()
P→info = Item

R=Q→Next

Q→Next = P
P→Prev = Q

R→ Prev = P
P→Next = R

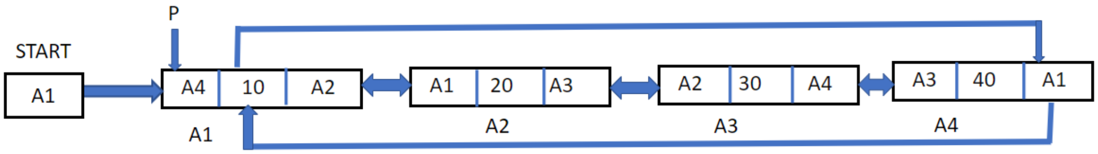
9.13.2.2 Deletions

Deletion at the Beginning

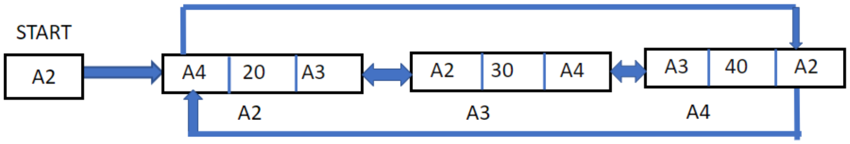
There can be two scenarios of deleting the first node in a circular doubly Linked List. The node which is to be deleted can be the only node present in the Linked List. In this case, the condition start → next == start will become true; therefore, the list needs to be completely deleted. It can be simply done by assigning head pointer of the list to null and free the head pointer.

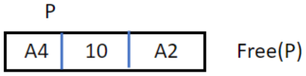
```
ALGORITHM DelBeg(cdSTART)
BEGIN:
IF cdSTART == NULL THEN
WRITE("Void Deletion")
ELSE
P=cdSTART
IF P→Next != cdSTART THEN
P→Next→Prev = P→Prev
P→Prev→Next = P→Next
cdSTART = P→Next
ELSE
cdSTART = NULL
item=cdSTART→Info
FreeNode(P)
RETURN item
END;
```

Case 1:

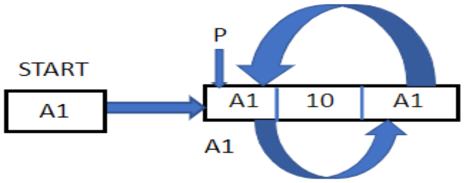


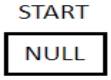
P→Next→Prev = P→Prev
P→Prev→Next = P→Next
cdSTART = P→Next

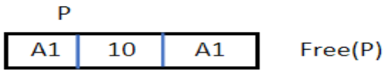




Case 2:







Time Complexity: O(1)

Here, in any case 7 statements are executed, which is a constant

Space Complexity: O(1)

One extra variable **P is** used. Total extra space allocated is constant

No of Link adjustments: 2

b) Deletion at the End

There can be two scenarios of deleting the last node in a circular doubly Linked List.

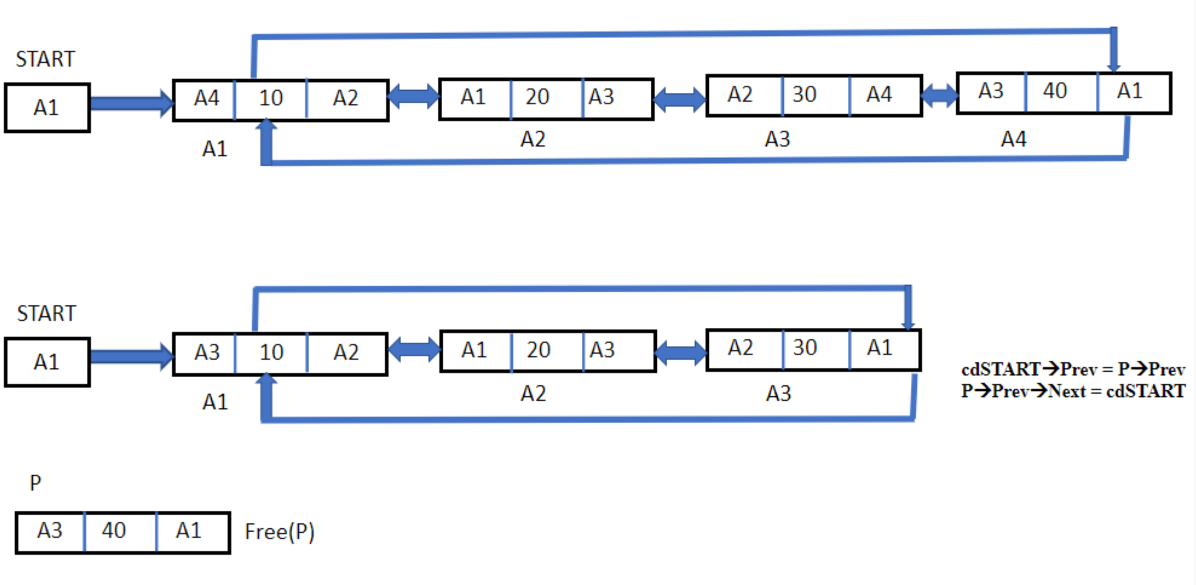
The node which is to be deleted can be the only node present in the Linked List. In this case, the condition start → next == start will become true; therefore, the list needs to be completely deleted.

If list contains more than one element than we need to traverse till the last node and after that we can perform deletion.

ALGORITHM DelEnd(cdSTART)

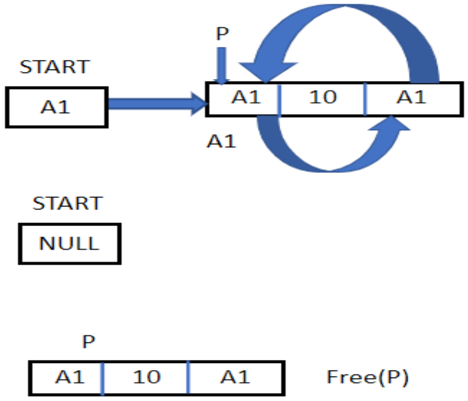
```
BEGIN:
IF cdSTART==NULL THEN
WRITE("Void Deletion")
P=cdSTART
IF P→Next! = cdSTART THEN
cdSTART→Prev = P→Prev
P→Prev→Next = cdSTART
ELSE
cdSTART=NULL
Item=P→Info
FreeNode(P)
Return item
END;
```

Case 1:



cdSTART→Prev = P→Prev
P→Prev→Next = cdSTART

Case 2:



Time Complexity: O(N)

Traversal requires O(N) Time. Total time required asymptotically is O(N)

Space Complexity: O(1)

One extra variable **P** are used. Total extra space allocated is constant

No of Link adjustments: 2

Deletion after the given node

The node which is to be deleted can be determined by finding the location of node to be deleted. If list contains more than one element than we can simply delete the node after reaching to that node.

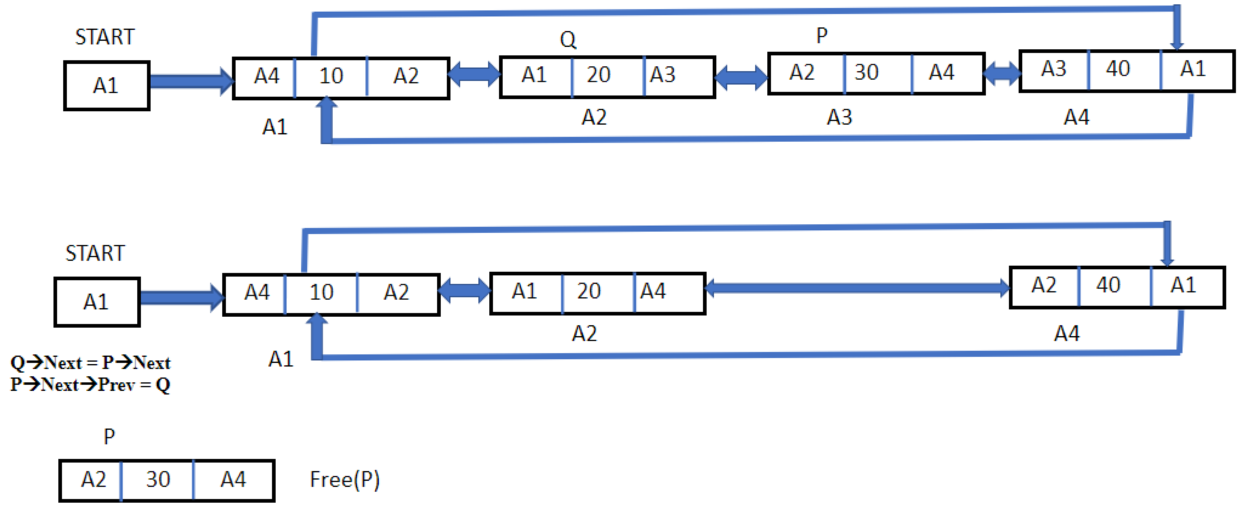
ALGORITHM DelAfter(Q)

BEGIN:

```
IF P == NULL OR P→Next == P THEN
WRITE("Void Deletion")
ELSE
P = Q→Next
Q→Next = P→Next
P→Next→Prev = Q
IF P == cdSTART THEN
cdSTART = Q→Next
item=P→Info
FreeNode(P)
RETURN item
```

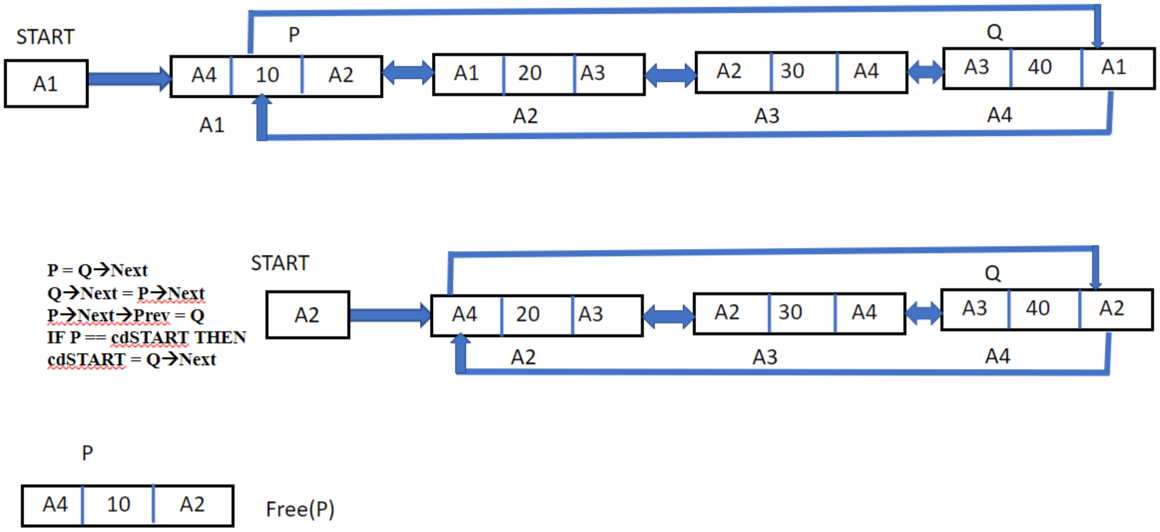
END;

Case 1:



```
Q→Next = P→Next
P→Next→Prev = Q
```

Case 2:



```
P = Q→Next
Q→Next = P→Next
P→Next→Prev = Q
IF P == cdSTART THEN
cdSTART = Q→Next
```

Time Complexity: Θ(1)

It takes only two statements when the Circular Linked List is empty and maximum 5 otherwise. In both of the above cases, the number of statement execution is fixed i.e. constant

Space Complexity: Θ(1)

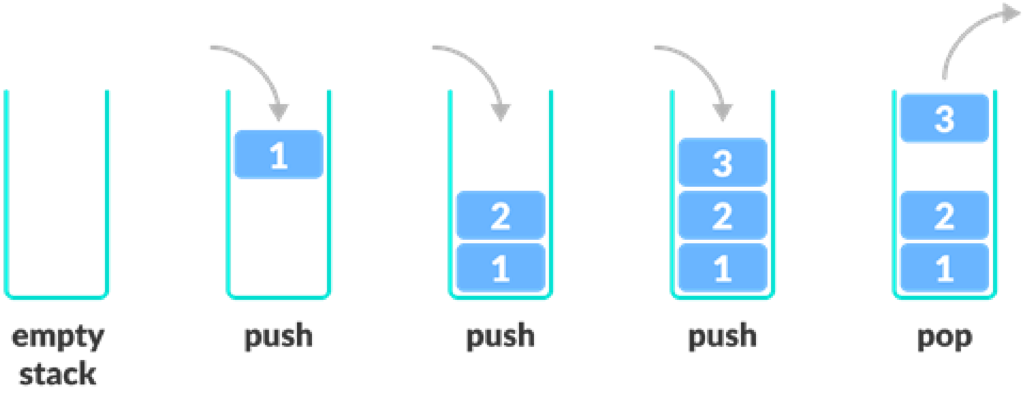
Two extra variables **P** & **item** are utilized, which is a constant number.

No of Link adjustments: 2 or 3

9.14 Implementation of other data structures using Linked List

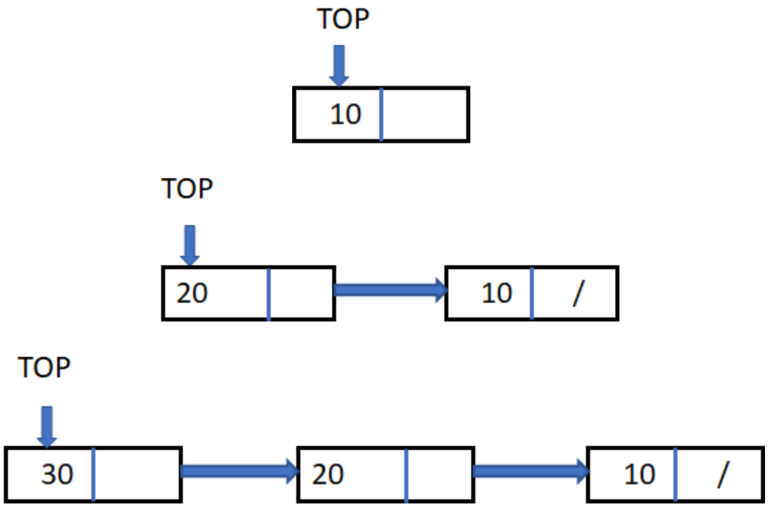
9.14.1 Stack

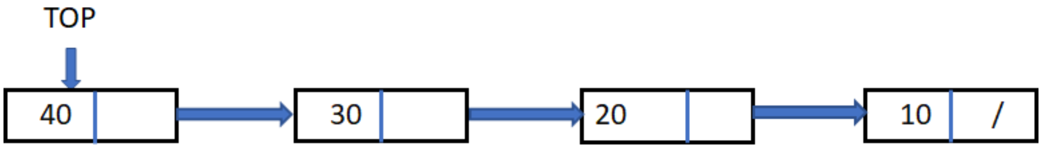
Stacks are the ordered collection of items into which items may be inserted or removed from the same end called TOP of the stack. Stack works on the principal of LIFO – last in first out scheme. To ensure the LIFO behavior we can think of insertion and deletion of items from beginning of the Linked List.



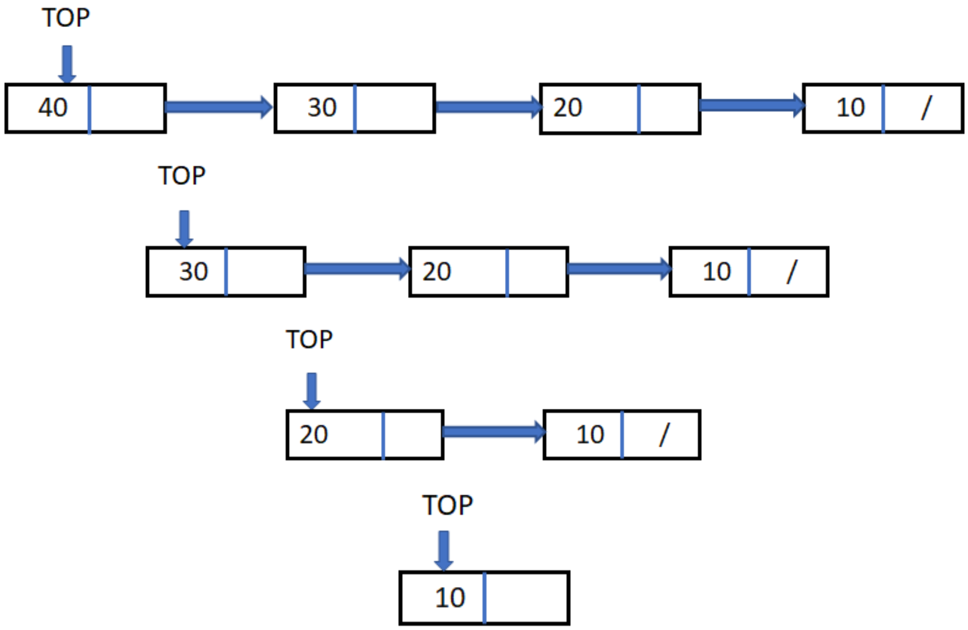
```
ALGORITHM   Initialization (TOP)
BEGIN:
TOP = NULL
END;
```

```
ALGORITHM   PUSH (TOP, item)
BEGIN:
InsBeg(TOP, item)
END;
```





```
ALGORITHM   POP(TOP)
BEGIN:
IF TOP = = NULL THEN
                WRITE (“Stack Underflows”)
                  RETURN
x = Delbeg(TOP)
RETURN x
END;
```



```
ALGORITHM   Display(TOP)
BEGIN:
RETURN TOP → info
END;
```

Explanation: The above algorithm performs STACK implementation using Linked List. For each STACK insertion, deletion and display operation PUSH ( ), POP( ), Display( ) functions are created respectively.

Complexity:

Time Complexity: PUSH operation: O(1)
POP operation: O(1)
Display operation: O(1)
Space Complexity: PUSH operation: O(1)
POP operation: O(1)
Display operation: O(1)

5.10 Merge Stack

Basically, there are two primitive operations on stack

Push(item) **:** In Push() operation, add an item into top of stack is done in O(1) time.

Pop(item) **:** In Pop() operation, delete an item from top of stack is done in O(1) time.

In mergeStack(stack1, stack2) operation, we merge the content of s2 stack into s1 stack in Constant (O(1)) time. In order to implement this, array is not a suitable data structure for Stack implementation as Push() and Pop() operation can easily be implemented in O(1) time but mergeStack() operation cannot be implemented in O(1) time.

MergeStack operation Using an array

Step1- Create a new stack S3 whose size equals the sum of stack S1 and stack S2.

Step2- Copy the item of stack S1 and stack S2 into stack s3.

Step3- Delete stack S1 and S2.

These steps take O(n) time, where n represents the combined size of S1 and S2.

MergeStack() operation Using link list

By using linked list we can implement MergeStack() in O(1) time.

Push(), Pop() and MergeStack() operation

Here we take two pointers **First** and **Top**. Top pointer pointing to Top of the node that is when new item to be inserted or item can be deleted. Second pointer named as first always points to first inserted node when stack is empty. So in order to merge the stack, simply link the top pointer of first stack to the first pointer of second stack.

```
ALGORITHM Push(TOP,START,ITEM)      //TOP is reference of last node to be added
BEGIN:                              //START is reference of first node to be added
    IF TOP==NULL THEN
        TOP→data=ITEM
        TOP→Next=NULL
        START=TOP
        RETURN TOP
    ELSE
        P→data=ITEM                 //P is the reference of new node
        P→Next=NULL
        TOP→Next=P
        TOP=P
        RETURN TOP
END;
```

In the above algorithms Push() operation takes O(1) time as TOP is always pointing to the last node so there is no need to traverse the entire link list.

```
ALGORITHM Pop(TOP)                  //TOP is reference of last node to be added
BEGIN:                              //START is reference of first node to be added
    IF TOP==NULL THEN
        RETURN TOP
    ELSE IF TOP→next==NULL THEN
        Free(TOP)
```

```
                TOP=NULL
                RETURN TOP
        ELSE
                P=TOP
                TOP=TOP→next
                Free(P)
                P=NULL
                RETURN TOP
END;
```

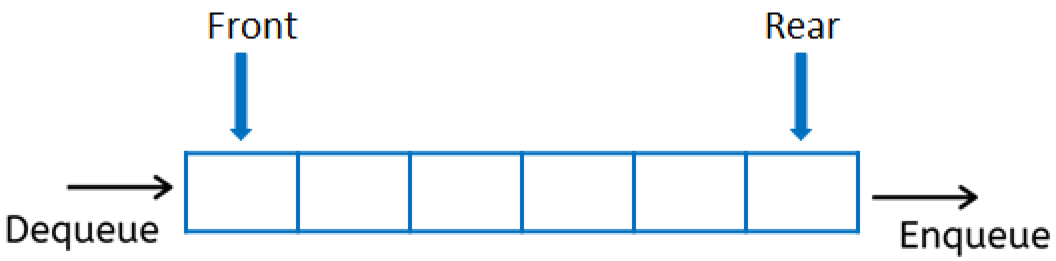
In the above algorithms pop() operation takes O(1) time as TOP is always pointing to the last node so there is no need to traverse the entire link list.

```
ALGORITHM MergeStack(TOP1,TOP2, START2) //TOP1 pointing to the last node of first stack
BEGIN:                                  //START2 pointing to the first node of second stack
        TOP1→next=START2
        RETURN TOP2
END;
```

In the above algorithms MergeStack() operation takes O(1) time as TOP1 is always pointing to the last node of first stack and START2 is pointing to the first node of second link list so here our task is only to link the two stacks.

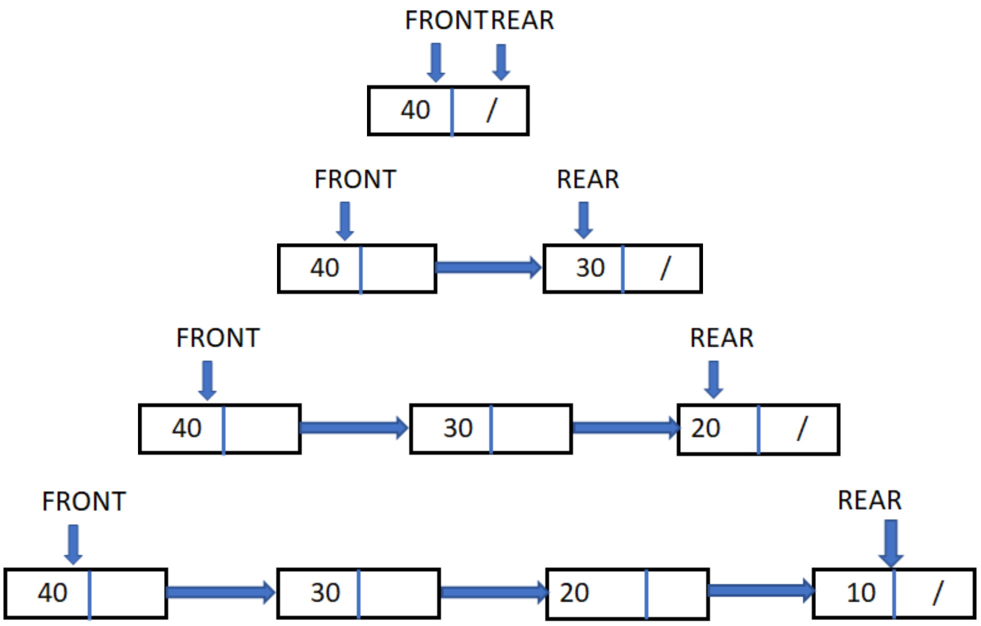
9.14.2 Queue

Queues are the ordered collection of items into which items may be inserted at one end called REAR of the queue and removed from another end called FRONT of the Queue. Queue works on the principal of FIFO – First in first out scheme. To ensure the FIFO behavior we can think of insertion at the end and deletion of items from beginning of the Linked List.

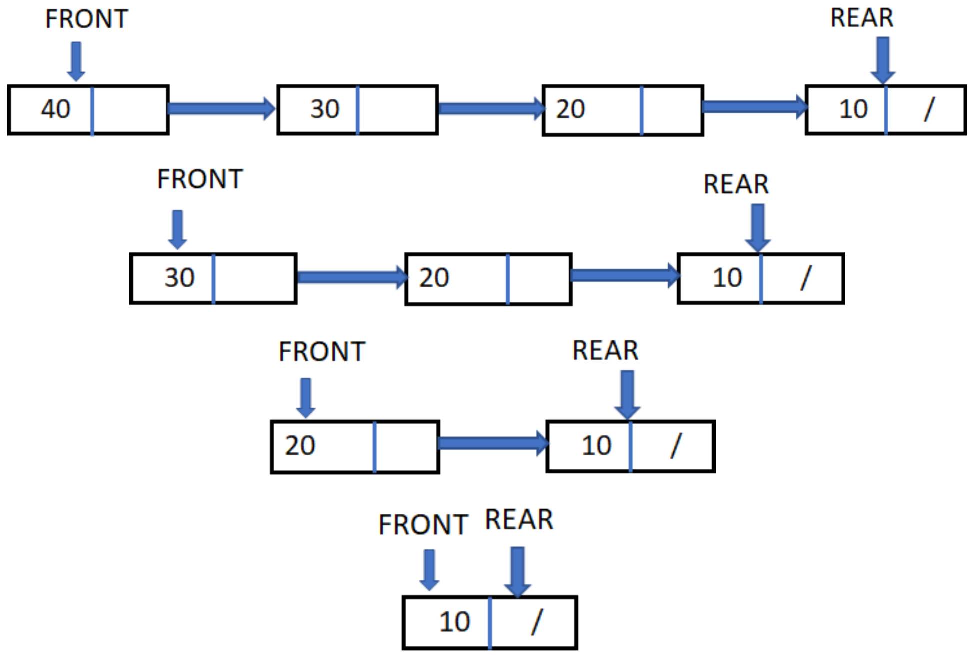


```
ALGORITHM Initialize(FRONT, REAR)
BEGIN:
FRONT = NULL
REAR = NULL
END;
```

```
ALGORITHM  Enqueue(FRONT, REAR, item)
BEGIN:
IF FRONT = = NULL AND REAR = = NULL THEN
InsBeg(REAR, item)
FRONT = REAR
ELSE
InsAfter(REAR, item)
REAR = REAR →NEXT
END;
```



```
ALGORITHM Dequeue (FRONT, REAR)
BEGIN:
IF FRONT = = NULL AND REAR = = NULL THEN
WRITE (“Queue Underflow”)
RETURN
ELSE
x = Delbeg(FRONT)
IF FRONT = = NULL THEN
REAR = NULL
END;
```



Explanation: The above algorithm performs QUEUE implementation using Linked List. For each QUEUE Initialization, Insertion and deletion operation Initialize(), Enqueue( ), Dequeue ( ) functions are created respectively.

Complexity:

Time Complexity: Enqueue operation: O(1)

Dequeue operation: O(1)

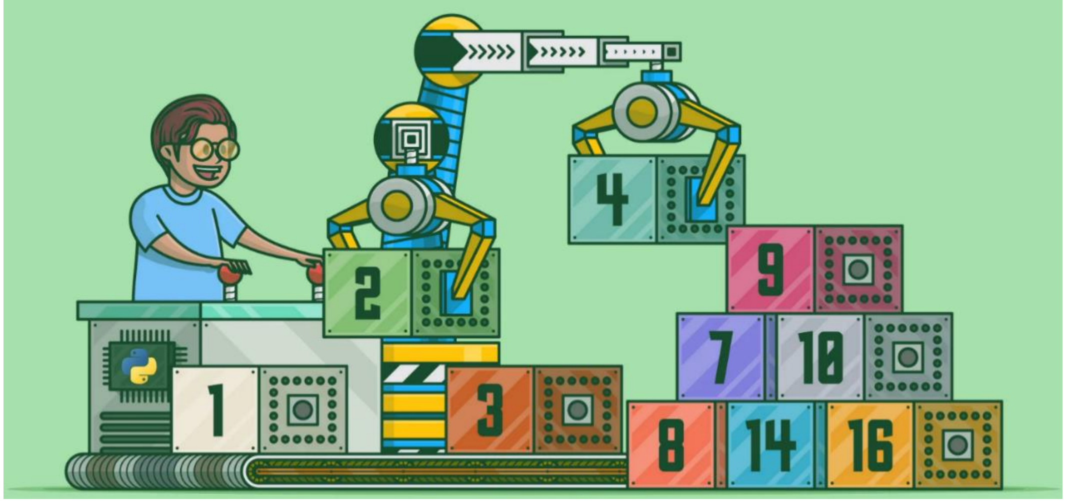
Space Complexity: Enqueue operation: O(1)

Dequeue operation: O(1)

9.14.3 Priority Queue

Consider a situation where students are assembled in a queue to deposit their fees. Meanwhile a faculty member arrives for taking some service from the same window as that of the fee counter. Students may probably give respect to the faculty and will allow him to take the service on priority.

Consider another situation in which an Operating system is required to execute the tasks on the basis on priority. The processes of different priority will keep coming in the system that needs to be executed. Higher priority process should be available readily for execution and incoming processes should be arranged according to the priority.



Priority Queues are the data structure in which the intrinsic ordering of the elements does determine the result of the basic operations.

There are two types of Priority Queue in general: Ascending, Descending.

Ascending PQ: Lower numbers are considered higher priority e.g. 1 means highest priority. Upon subsequent deletions, the ascending sequence of priority numbers will appear.

Descending PQ: Higher numbers are considered higher priority. Upon subsequent deletions, the descending sequence of priority numbers will appear.

Priority Queue can be implemented using Array, Linked List and Heaps.

For priority queue implementation using link list, add extra field priority is used in the node (along with 2 usual fields as Info and Next).

Insertion in Priority Queue

For insertion of an item in the Priority Queue, following steps can be performed.

- If start contains NULL then it means simply insert the item as a first node.
- If start does not contain NULL then first compare priority field and insert according to the priority.
- In case of deletion just free the first node as highest priority.

```
ALGORITHM EnQueue(START, Item, Py)
BEGIN:
        P=GetNode()
        P→info = Item
        P→Priority = Py
Q = NULL
R = START
WHILE R!=NULL AND R→priority >= Py DO
                Q=R
R=R→Next
        IF Q==NULL THEN
                InsBeg(START,Item,Py)
        ELSE
                InsAft(START,Item,Py)
END;
```

Deletion in Priority Queue

```
ALGORITHM DeQueue(START)
BEGIN:
IF START==NULL THEN
WRITE("Void Deletion")
RETURN
ELSE
Item=DelBeg(START)
RETURN Item
END;
```

Complexity-

EnQueue Algorithm takes O(N) time because traversal is required in the list and comparison of the priority in just single pass.

DeQueue Algorithm requires O(1) time.
Both the Algorithms require O(1) Space.

9.14.3 Double Ended Queue

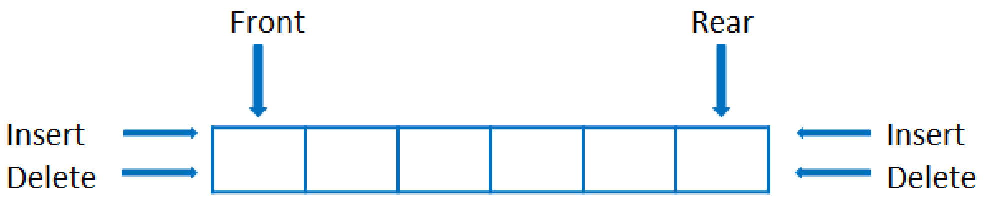
Consider a situation where a software maintains a list of actions performed and upon undo the recent activity gets undone. In case we perform the redo, the same activity comes back. This is possible only if we preserver the undone activity somewhere. This creates the requirement of storing the fresh activity and undo activity as well.

If we are dealing with a limited size buffer for storage of activities, then upon the insertions of the activities beyond the size of the buffer, we need to delete the outdated activities. This creates the requirement of deletion upon undo and again when buffer overflows.

For the above activities, we need the insertion and deletion allowed at both the ends in the buffer.

Consider the Web browser's history. Recently visited URLs are added to the front of the deque, and the URL at the back of the deque is removed after some specified number of insertions at the front.

Definition: Double Ended Queue are Data structure in which insertion and Deletions are possible at both the ends i.e. Front and Rear.



There are two variations of the double ended queue

a) Input Restricted Deque: Deletions are allowed at both the ends but the insertion is restricted to either of the ends

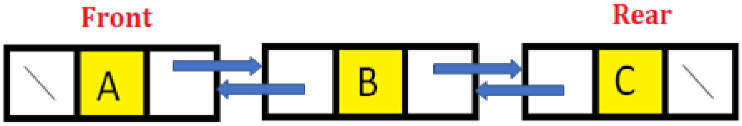
b) Output Restricted Deque: Insertions are allowed at both the ends but the Deletion is restricted to either of the ends

Best way to implement the Double Ended Queue is the Linked Lst

Operations on Double Ended Queue

```
ALGORITHM Initialize(Deque DQ)
BEGIN:
DQ.FRONT=NULL
DQ.REAR=NULL
END;
```

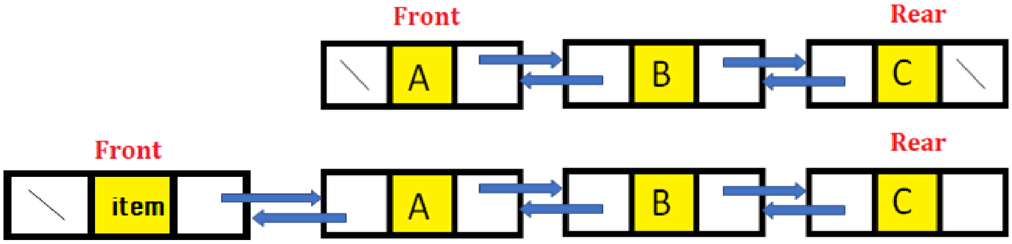
```
ALGORITHM Empty(Deque DQ)
BEGIN:
    IF DQ.FRONT = = NULL THEN
        RETURN TRUE
    ELSE
        RETURN FALSE
END;
```



Insertions

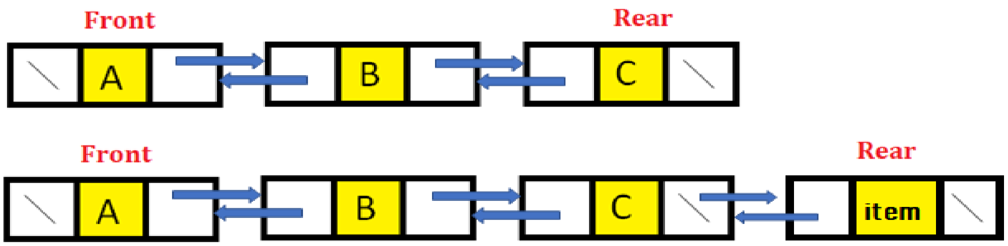
Insertion at Front

```
ALGORITHM InsFront(DQ, item)
BEGIN:
    InsBeg(DQ.FRONT, item)
    IF DQ.REAR == NULL THEN
        DQ.REAR = DQ.FRONT
END;
```



Insertion at Rear

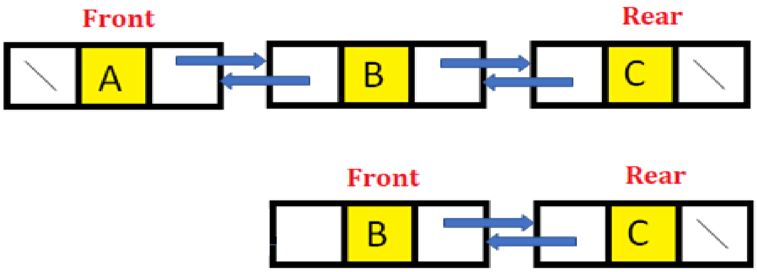
```
ALGORITHM InsRear(DQ, item)
BEGIN:
    IF DQ.REAR == NULL THEN
        InsBeg(DQ.Rear, item)
        DQ.FRONT = DQ.Rear
    ELSE
InsAft(DQ.REAR, item)
END;
```



Deletions

Deletion at Front

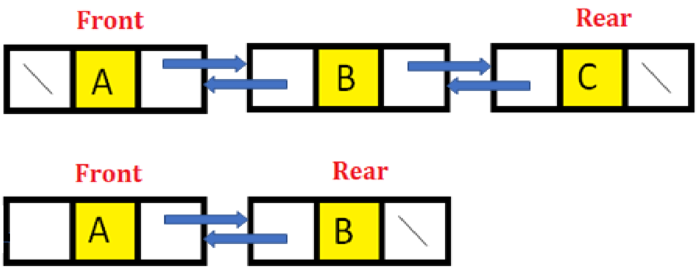
```
ALGORITHM DelFront(DQ)
BEGIN:
          IF EMPTY (DQ) THEN
             WRITE ("Queue Underflows")
             EXIT (1)
          ELSE
          Item = DelBeg(DQ.FRONT)
                    IF DQ.FRONT == NULL THEN
                              DQ.REAR = NULL
                    RETURN item
END;
```



Deletion at Rear

```
ALGORITHM DelRear(DQ)
BEGIN:
          IF EMPTY (DQ) THEN
             WRITE ("Queue Underflows")
             EXIT (1)
          ELSE
          DQ.Rear = DQ.REAR → Left
IF DQ.Rear == NULL
Item = DelBeg(DQ.FRONT)
```

```
            ELSE
                Item = DelAft(DQ.REAR)
        RETURN item
END;
```



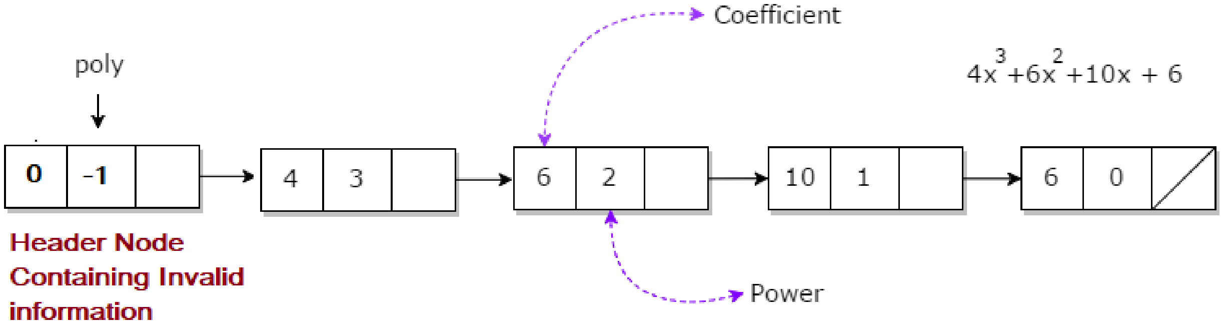
Time and Space Complexity of each of the above operations is O(1).

9.15 Header Linked List

A *header node* is a special node that is found at the *beginning* of the list. A list that contains this type of node, is called the header-Linked List. This type of list is useful when information other than that found in each node is needed. *For example*, suppose there is an application in which the number of items in a list is often calculated. Usually, a list is always traversed to find the length of the list. However, if the current length is maintained in an additional header node that information can be easily obtained.

Similarly, header node can contain maximum or minimum information among all the nodes.

Sometimes header node denotes just the beginning of the Linked List and it contains invalid information. E.g.

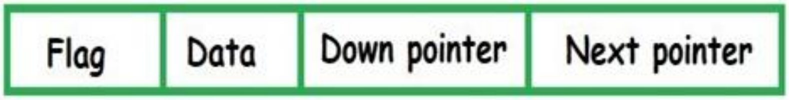


9.16 Generalized Linked List

A Generalized Linked List L, is defined as a finite sequence of n>=0 elements, l_1, l_2, l_3, l_4, …, l_n, such that l_i are either atom or the list of atoms. Thus

L = (l_1, l_2, l_3, l_4, …, l_n)

where n is total number of nodes in the list.



To represent a list of items there are certain assumptions about the node structure.

- Flag = 1 implies that *down pointer* exists
- Flag = 0 implies that *next pointer* exists
- Data means the atom
- Down pointer is the address of node which is down of the current node
- Next pointer is the address of node which is attached as the next node

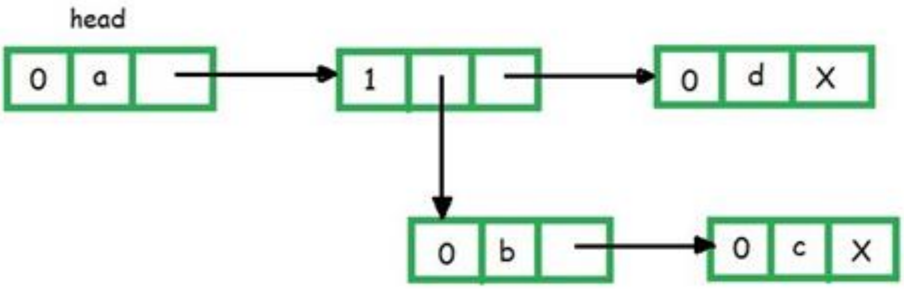
Why Generalized Linked List?

Generalized Linked Lists are used because although the efficiency of polynomial operations using Linked List is good but still, the disadvantage is that the Linked List is unable to use *multiple variable polynomial equation* efficiently. It helps us to represent multi-variable polynomial along with the list of elements.

Typical 'C' structure of Generalized Linked List

Example of GLL {List representation}

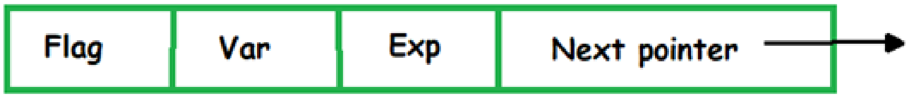
(a, (b, c), d)



When first field is 0, it indicates that the second field is variable. If first field is 1 means the second field is a down pointer, means some list is starting.

Polynomial Representation using Generalized Linked List

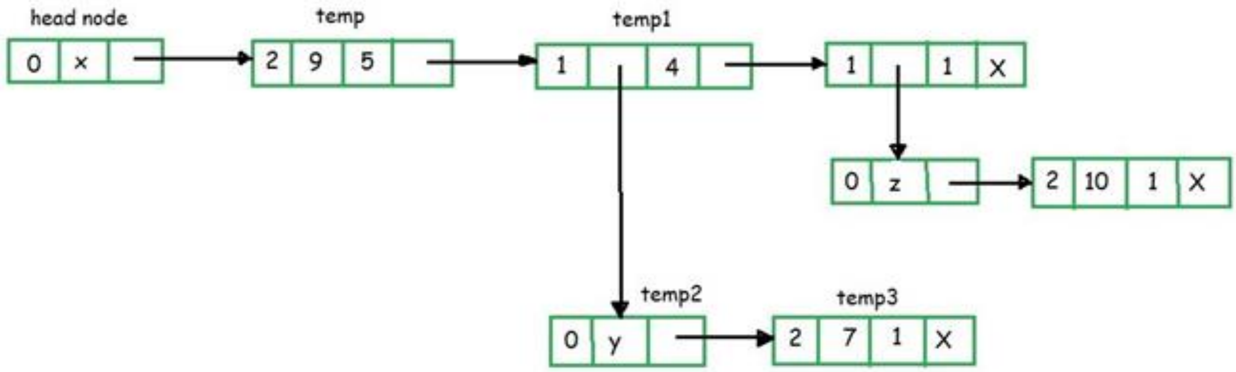
The typical node structure will be:



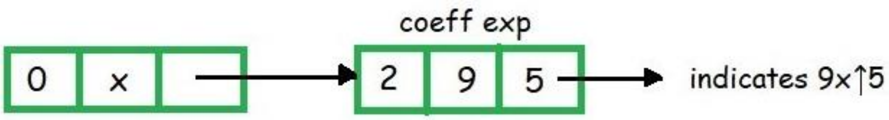
- Here Flag = 0 means *variable* is present
- Flag = 1 means *down pointer* is present
- Flag = 2 means *coefficient* and *exponent* is present

Example:

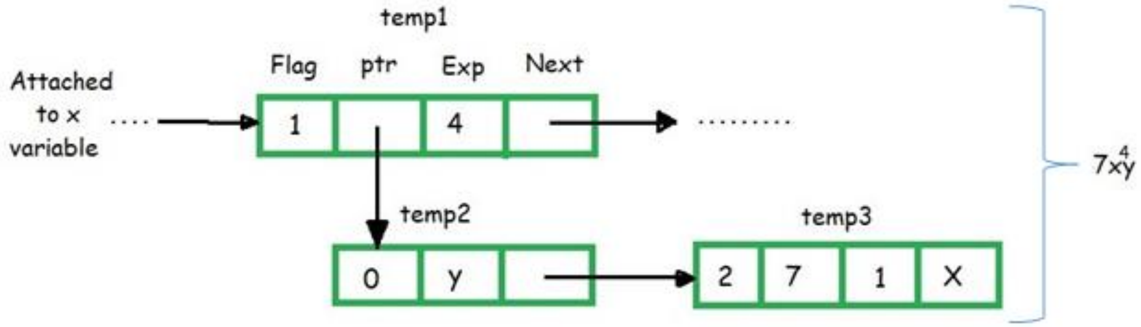
$9x^5 + 7xy^4 + 10xz$



In the above example the head node is of variable x. The temp node shows the first field as 2 means coefficient and exponent are present.



Since temp node is attached to head node and head node is having variable x, temp node having coefficient = 9 and exponent = 5. The above two nodes can be read as $9x^5$. ↑



Similarly, in the above figure, the node temp1 can be read as x^4.

The flag field is 1 means down pointer is there

temp2 = y

temp3 = coefficient = 7

exponent = 1

flag = 2 means the node contains coefficient and exponent values.

temp2 is attached to temp3 this means $7y_1$ and temp2 is also attached to temp1 means

temp1 x temp2

x^4 x $7y^1$ = $7x^4y^1$ value is represented by above figure

9.17. Garbage Collection in Linked List

In case the Linked List were implemented using Array, a pool of nodes are created. Each GetNode function allocates the node from this pool and nodes are collected back to this pool corresponding to Freenode function. The nodes which are freed can be re-used by subsequent calls to Getnode function.

Analogy: - RAM Memory Organization

9.18 Disadvantage of Linked Lists

- The Linked List consume more memory per data item as compared to Arrays as one address field (minimum) is required for storing the address of the next node.
- Random access to the nodes is not possible, each time the traversal has to start from the beginning of the Linked List (Sequential access)
- Back traversal in Linked List either requires the additional address field (as in the doubly Linked List) or Recursion. Both increases the time space complexity further as compared to array.

9.18 Competitive coding Problems

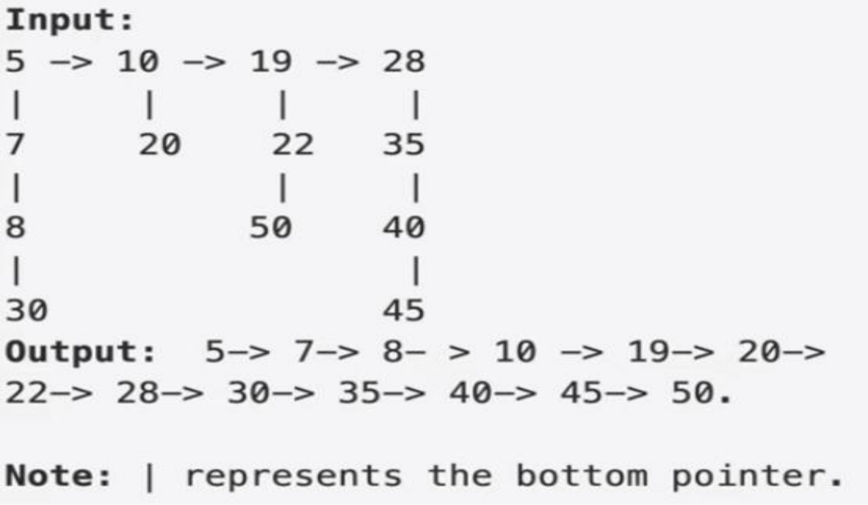
Problem1: Flattening of Linked List

Given a Linked List of size N. Each node of this represents a Linked List. There are two types of the pointer in this Linked List nodes.

Next pointer: points to the next node

Bottom pointer: points to the Linked List where this node is the head

The flattened list will be printed using the bottom pointer instead of the next pointer.



```
Input:
5 -> 10 -> 19 -> 28
|      |      |      |
7      20     22    35
|             |      |
8             50    40
|                    |
30                  45
Output:  5-> 7-> 8- > 10 -> 19-> 20->
22-> 28-> 30-> 35-> 40-> 45-> 50.

Note: | represents the bottom pointer.
```

Problem 2: Arrangement/De-arrangement

Arrange the consonants ad vowel nodes of the linked list it in such a way that all the vowels nodes come before the consonants while maintaining the order of their arrival

Problem 3: Deletion of Nodes from Back

Delete kth node from end of a linked list in a single scan O(n) time

Problem 4: Sum Circular Linked List

Modify a Circular Linked List such that each node stores the sum of all nodes except itself

Problem 5: Non-Fibonacci Circular Linked List

Remove all Fibonacci Nodes from a Circular Singly Linked List