Subject : Data Structure & Algorithms

Class : SYBCA – Semester IV

Lecturer : Ms. Archana R. Oza

Unit II : Sorting and Linked List

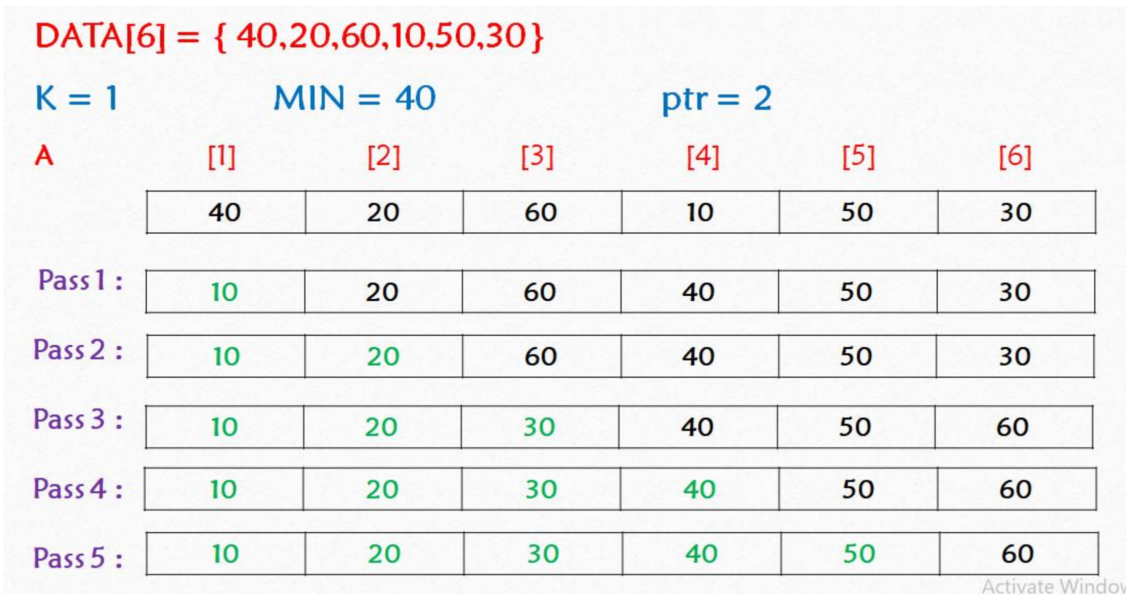
- Selection Sort
- Bubble Sort
- Insertion Sort
- Introduction to Linked List
- Representation of Linked List in Memory
- Searching a Linked List
- Memory Allocation, Garbage Collection
- Insertion and Deletion in Linked List

Sorting Operation is used to arrange the data items in some logical order. Sorting is ordering a list of objects. A Sorting Algorithm is used to rearrange a given array or list elements in ascending or descending order or either in alphanumeric order.

There are various methods for sorting linear arrays, some of them are :

1. Bubble Sort
2. Insertion Sort
3. Selection Sort

- **Selection Sort :** Selection sort is an algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning. First, find the smallest element of the array and place it on the first position. Then, find the second smallest element of the array and place it on the second position. The process continues until we get the sorted array.
  The array with n elements is sorted by using n-1 pass of selection sort algorithm.
  For e.g. Let DATA be array in memory as follows :
  DATA[6] = {40,20,60,10,50,30}
  Sort the given array using Selection sort.



DATA[6] = { 40,20,60,10,50,30 }

K = 1 MIN = 40 ptr = 2

A	[1]	[2]	[3]	[4]	[5]	[6]
	40	20	60	10	50	30
Pass 1 :	10	20	60	40	50	30
Pass 2 :	10	20	60	40	50	30
Pass 3 :	10	20	30	40	50	60
Pass 4 :	10	20	30	40	50	60
Pass 5 :	10	20	30	40	50	60

Algorithm : SelectionSort(DATA, N, k, MIN, TEMP, PTR,LOC)

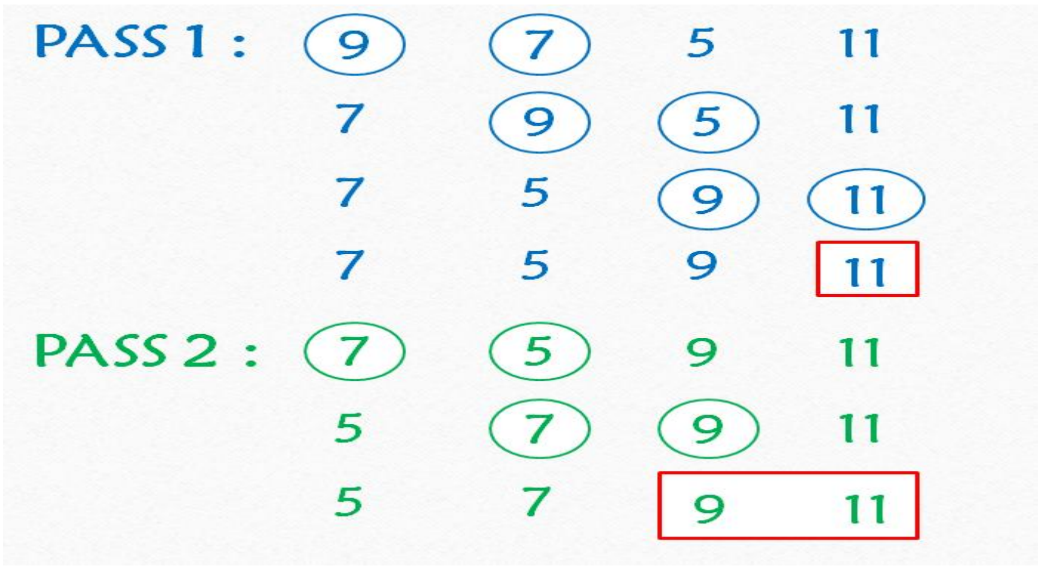
Description : Here, DATA is the linear array with N elements. PTR is the pointer variable to execute the passes and TEMP is the temporary variable that stores the value of elements. LOC stores the location of value to be swapped.

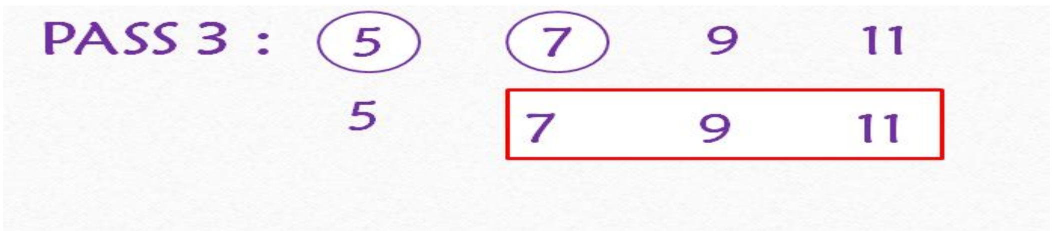
This algorithm sorts the array DATA by repeatedly finding the minimum element (considering ascending order) from the list and putting it at the beginning.

```
Step 1 : Repeat Steps 2,3 and 4 for k=1 to N
Step 2 : Set MIN := DATA[k]
Step 3 : Repeat for ptr = k+1,k+2, _ _ _ _, k+N
             If DATA[ptr] < MIN then
                 MIN = DATA[ptr] & LOC=ptr

Step 4 : Set TEMP := DATA[k]
             DATA[k] := DATA[LOC]
             DATA[LOC] := TEMP
Step 5 : Exit
```

- **Bubble Sort :** Bubble Sort is the simplest sorting algorithm for small list.
  It compares the adjacent elements and swaps their positions if they are not in the desired order.
  Starting from the first index, it compares the first and the second element. If the first element is greater than the second element, they are swapped. If we have total N elements, we need to repeat this process for N-1 times.
  It is known as bubble sort, because with every complete iteration the largest element in the given array, bubbles up towards the last place or the highest index, just like a water bubble rises up to the water surface.
  For e.g. Let DATA be array in memory as follows :
  DATA[4] = {9,5,7,11}
  Sort the given array using Bubble sort.





Thus the sorted array is :

DATA[4] = {5,7,9,11}

Algorithm : BubbleSort(DATA, PTR, K, N)

Description : Here, DATA is the linear array with N elements. PTR is the pointer variable that executes the passes and K is the counter variable that iterates the loop from lower bound to upper bound of the array.
This algorithm sorts the given array in increasing numerical order.

```
Step 1 : [Initialize variable] Set k = LB and PTR = k
Step 2 : Repeat step 3 while K < N
Step 3 : [Execute Pass] Repeat while PTR<=N-k
            a) If DATA[PTR]>DATA[PTR+1] then
                TEMP = DATA[PTR]
                DATA[PTR] = DATA[PTR + 1]
                DATA[PTR + 1] = TEMP
            b) Set PTR :=PTR+1 [End of step 3 loop]
         K = K + 1 [End of step 2 loop]
Step 4 : Exit
```

- **Insertion Sort :** Insertion sort works similarly as we sort cards in our hand in a card game. We assume that the first card is already sorted then we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left.

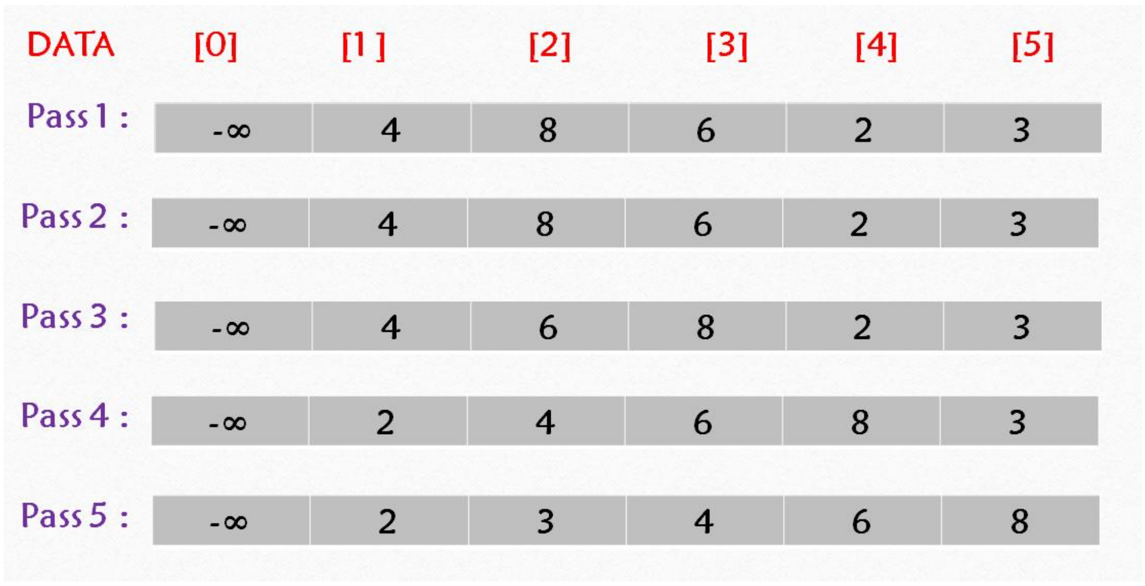
  In the same way, other unsorted cards are taken and put at their right place. A similar approach is used by insertion sort.

  Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration. It is efficient for smaller data sets.

  For e.g. Let LA be array in memory as follows :

  LA[5] = {4,8,6,2,3}

  Sort the given array using Insertion sort.



DATA	[0]	[1]	[2]	[3]	[4]	[5]
Pass 1 :	-∞	4	8	6	2	3
Pass 2 :	-∞	4	8	6	2	3
Pass 3 :	-∞	4	6	8	2	3
Pass 4 :	-∞	2	4	6	8	3
Pass 5 :	-∞	2	3	4	6	8

Algorithm : InsertionSort(LA, N, k, TEMP, PTR)

Description : Here, LA is the linear array with N elements. PTR is the pointer variable to execute the passes and TEMP is the temporary variable that stores the value of elements.

This algorithm sorts the array LA by comparing adjacent elements and shifting their location as required.

```
Step 1 : Set LA[0] = $-\infty$
Step 2 : Repeat steps 3 to 5 for k=2, 3, - , - , N
Step 3 : Set TEMP :=LA [k] & PTR :=k-1
Step 4 : Repeat while TEMP < LA [PTR]
            a) Set LA [PTR+1] := LA [PTR]
            b) Set PTR := PTR – 1
Step 5 : [Insert element at proper position]
            Set LA [PTR+1] :=TEMP
Step 6 : Exit
```

Linked List & its memory representation :

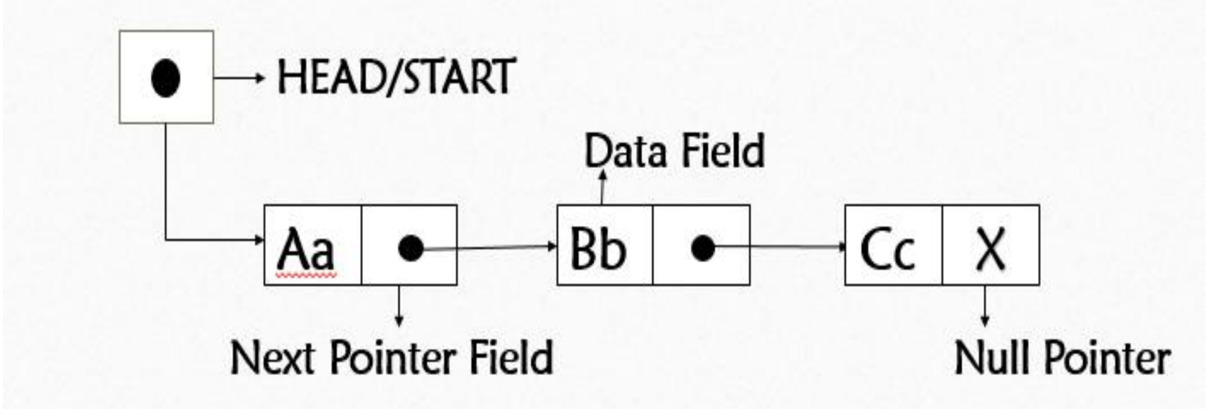
A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations.

It is a collection of data called Nodes, where the linear order is given by means of pointers. Each node is divided into two parts :

First part contains the data element i.e. Information field and second part contains the address of next node i.e. Next Pointer Field. The node that contains the Null Pointer indicates the end of list.

The last node has a reference to null. The entry point into a linked list is called the head/start of the list. Head is not a separate node, but the pointer to the first node.

For Example :



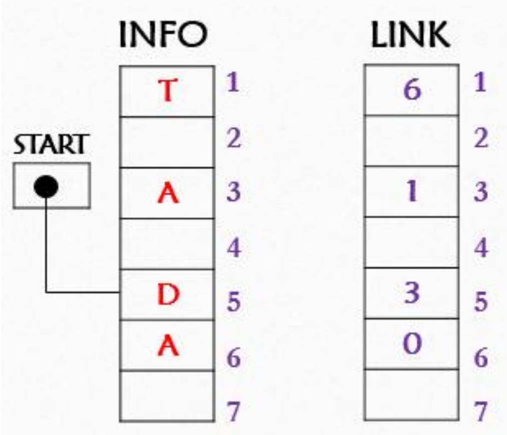
Memory Representation :

Let LIST be linked list in memory, as :

LIST requires two linear arrays.

We call them **INFO** and **LINK**, such as INFO[k] contains the information part & LINK[k] contains the next pointer field of the node.

LIST requires a variable START that contains the location of beginning of the list & NULL[0] indicates the end of list.



Here, START = 5, INFO[5]=D
LINK[5] = 3 INFO[3]=A
LINK[3] = 1 INFO[1]=T
LINK[1] = 6 INFO[6]=A
LINK[6] = 0

Thus, the information is DATA.

Traversing a Linked List :

Traversing refers to visiting or processing each element in the list atleast once. We start from the beginning and visit one node at a time until the end of the list.

Algorithm :TraverseLL(LIST, INFO, LINK, PTR)

Let LIST be linked list in memory with two linear arrays INFO & LINK. The variable PTR points to the node currently being processed.

This algorithm traverses LIST by applying an operation PROCESS to each element in the list.

```
Step 1 : [Initialize pointer PTR] Set PTR := START
Step 2 : Repeat steps 3 & 4 while PTR != NULL
Step 3 : Apply PROCESS to INFO[PTR]
Step 4 : [PTR now points to next node]
          Set PTR := LINK[PTR]
Step 5 : EXIT
```

Searching a Linked List :

Searching an element in a given linked list will happen sequentially starting from the head node of the list until the element has been found.

Searching any element in the list needs traversing through the list and make the comparison of every element of the list with the specified element.

If the element is matched with any of the list element then the location of the element is returned else search is said to be unsuccessful.

Searching in Linked List can be done on two types of lists, as :

1. Unsorted List
2. Sorted List

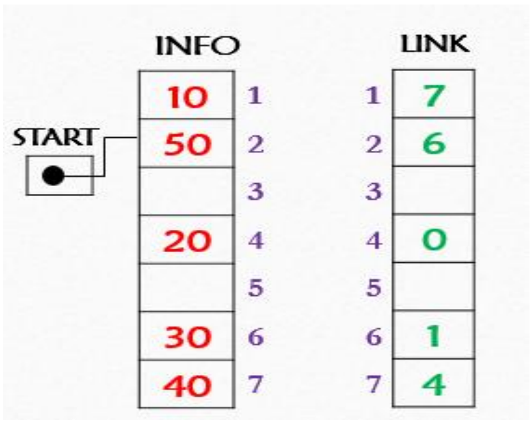
Algorithm for Searching List when the list is unsorted :

Algorithm : SearchLL(INFO, LINK, PTR, ITEM, LOC)

Description : Let LIST be linked list in memory with two linear arrays INFO & LINK. The variable PTR points to the node currently being processed.

This algorithm finds the location LOC of the node where ITEM first appears in the LIST or sets LOC=NULL.

```
Step 1 : [Initialize pointer PTR] Set PTR := START
Step 2 : Repeat Step 3 while PTR != NULL
Step 3 : If ITEM=INFO[PTR] then
            Set LOC :=PTR & Exit
         Else
            Set PTR :=LINK[PTR]
Step 4 : [Search Unsuccessful] Set LOC := NULL
Step 5 : Exit
```



For e.g. ITEM to be searched – 40

Search Successful at Location 7

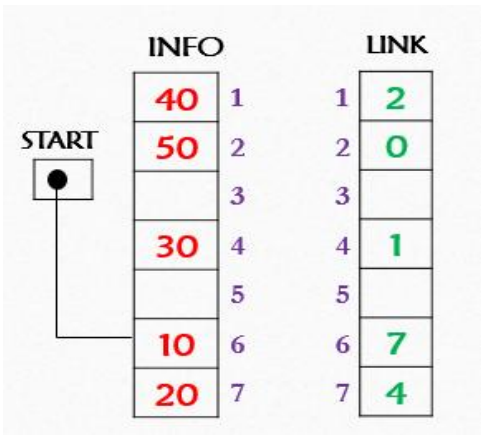
Algorithm for Searching List when the list is sorted :

Algorithm : SearchLL(INFO, LINK, PTR, ITEM, LOC)

Description : Let LIST be linked list in memory with two linear arrays INFO & LINK. The variable PTR points to the node currently being processed.

This algorithm finds the location LOC of the node where ITEM first appears in the LIST or sets LOC=NULL.

```
Step 1 : [Initialize pointer PTR] Set PTR := START
Step 2 : Repeat Step 3 while PTR != NULL
Step 3 : If ITEM<INFO[PTR] then
             Set LOC := NULL & Exit
         Else if ITEM=INFO[PTR] then
             Set LOC := PTR & Exit
         Else
             Set PTR := LINK[PTR]
Step 4 : [Search Unsuccessful] Set LOC := NULL
Step 5 : Exit
```



For e.g. ITEM to be searched – 40

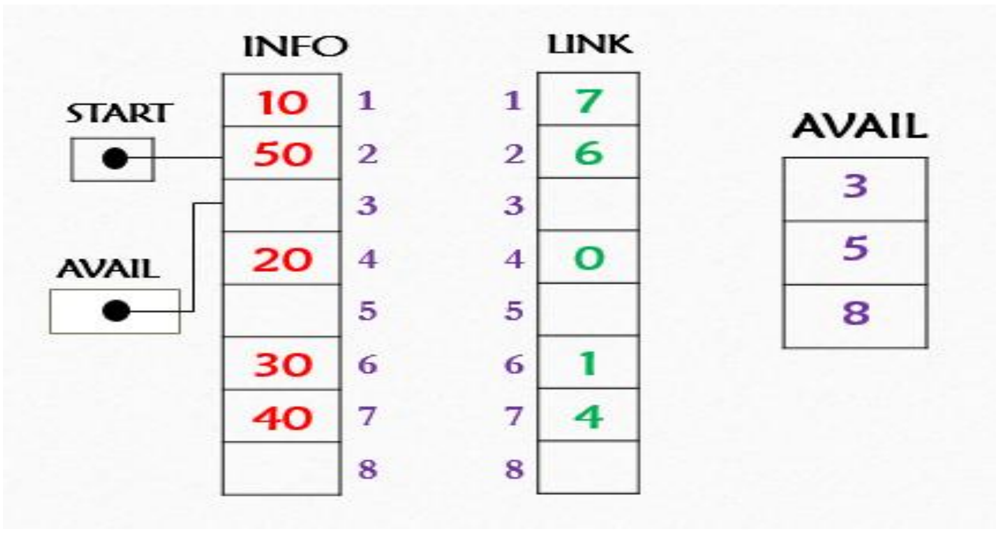
Search Successful at Location 1

Memory Allocation :

Whenever a new node is created, memory is allocated by the system. This memory is taken from list of those memory locations which are free i.e. not allocated. This list is called AVAIL List.

Similarly, whenever a node is deleted, the deleted space becomes reusable and is added to the list of unused space i.e. to AVAIL List. This unused space can be used in future for memory allocation.

Thus, Memory allocation is the process of setting aside sections of memory in a program to be used to store variables, and instances of structures and classes.



AVAIL is a list of free nodes in the memory. It contains unused memory cells and these memory cells can be used in future. It is also known as ‘List of Available Space’ or ‘Free Storage List’.

It is used along with linked list. Whenever a new node is to be inserted in the linked list, a free node is taken from the AVAIL List and is inserted in the linked list.

Similarly, whenever a node is deleted from the linked list, it is inserted in the AVAIL List. So that it can be used in future.

Memory allocation is of two types :

i. **Static Memory Allocation :**

When memory is allocated during compilation time, it is called ‘Static Memory Allocation’. This memory is fixed and cannot be increased or decreased after allocation.

If more memory is allocated than requirement, then memory is wasted. If less memory is allocated than requirement, then program will not run successfully. So exact memory requirements must be known in advance.

ii. Dynamic Memory Allocation :

When memory is allocated during run/execution time, it is called 'Dynamic Memory Allocation'. This memory is not fixed and is allocated according to our requirements.

Thus in it there is no wastage of memory. So there is no need to know exact memory requirements in advance.

Garbage Collection :

Whenever a node is deleted, some memory space becomes reusable. This memory space should be available for future use.

One way to do this is to immediately insert the free space into availability list. But this method may be time consuming or inefficient for the operating system. So another method is used which is called 'Garbage Collection'.

The special technique of operating system which searches unused memory space and makes it free is called as garbage collection.

In this method the OS collects the deleted space time to time onto the availability list. This process happens in two steps.

In first step, the OS goes through all the lists and tags all those cells which are currently being used.

In the second step, the OS goes through all the lists again and collects untagged space and adds this collected space to availability list.

Garbage collection is an automated process executed when small amount of free space is left in the system or no free space is left in the system or when CPU is idle and has time to do the garbage collection.

Overflow condition :

If we have to insert new space into the data structure, but there is no free space i.e. availability list is empty, then this situation is called 'Overflow'.

The programmer can handle this situation by printing the message of OVERFLOW. Overflow happens at the time of insertion, when AVAIL = NULL.

Underflow condition :

If we have to delete data from the data structure, but there is no data in the data structure i.e. data structure is empty, then this situation is called 'Underflow'.

The programmer can handle this situation by printing the message of UNDERFLOW. Underflow happens at the time of deletion, when START=NULL.

Insertion in Linked List :

Linked List is a linear & dynamic data structure that have list of data elements. All the data elements are linked with each other with the help of memory space address.

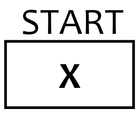
Such organization helps to perform Insertion & Deletion efficiently. Insertion operation refers to adding a new node which contains data as well as address.

Linked List for its insertion uses a START pointer which always points to first node in the list.

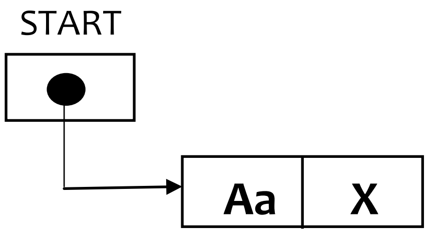
While Inserting a new node, following possibilities are considered :

i. Inserting first node :

If START pointer contains NULL address that means the new node is the first node of the list.

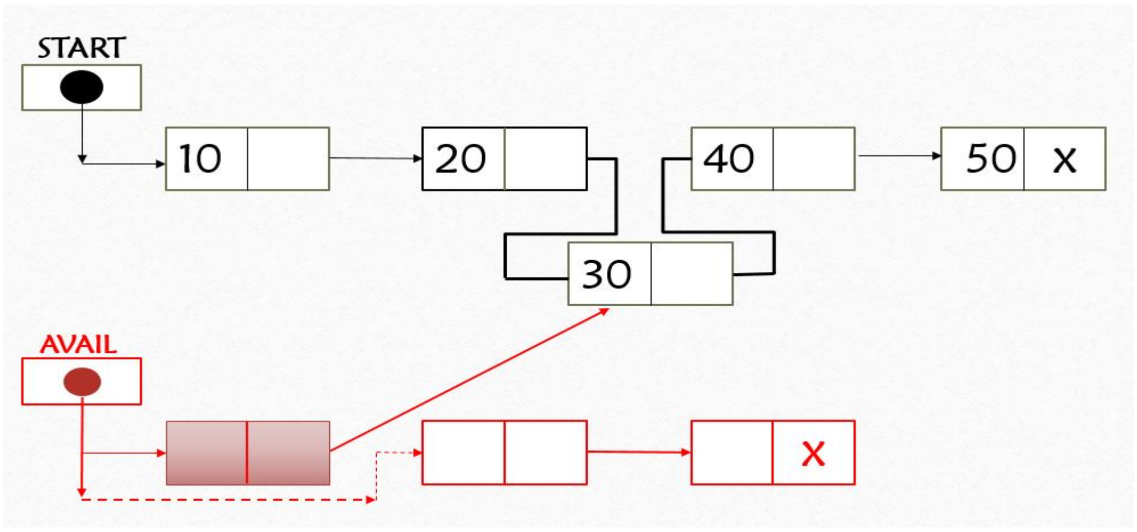


After Insertion :



ii. Inserting after any given location :

If a linked list contains some data nodes then at any position a new node can be inserted. After Insertion it becomes mandatory to properly update the addresses of nodes. So that the linked list maintains its linearity.



The Insertion algorithm must first check whether AVAIL NULL or some address. If AVAIL contains NULL then the condition becomes Overflow i.e. no space to insert new node. In other case, Insertion can be performed.

Algorithm : INSERT_LIST(INFO, LINK, AVAIL, NEW, PTR, ITEM, LOC)

Description : Let LIST be the linked list in memory with INFO, LINK, AVAIL & START pointers. ITEM is the element to be inserted as a new node & **LOC** is the location **after which** new node is to be inserted.

This algorithm inserts new element after the given location.

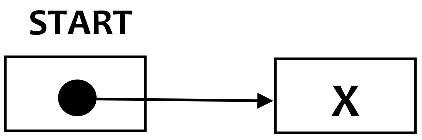
```
Step 1 : [OVERFLOW ?] If AVAIL = NULL then
              Write OVERFLOW & Exit
Step 2 : Set NEW := AVAIL   &   AVAIL := LINK [AVAIL]
Step 3 : Set INFO[NEW] := ITEM
Step 4 : If START = NULL then [Insertion at beginning]
              Set LINK [NEW] :=START &
                  START := NEW
         Else [Insertion after given location]
              Set LINK [NEW] := LINK[LOC] &
                  LINK [LOC] := NEW
Step 5 : EXIT
```

Deletion From Linked List :

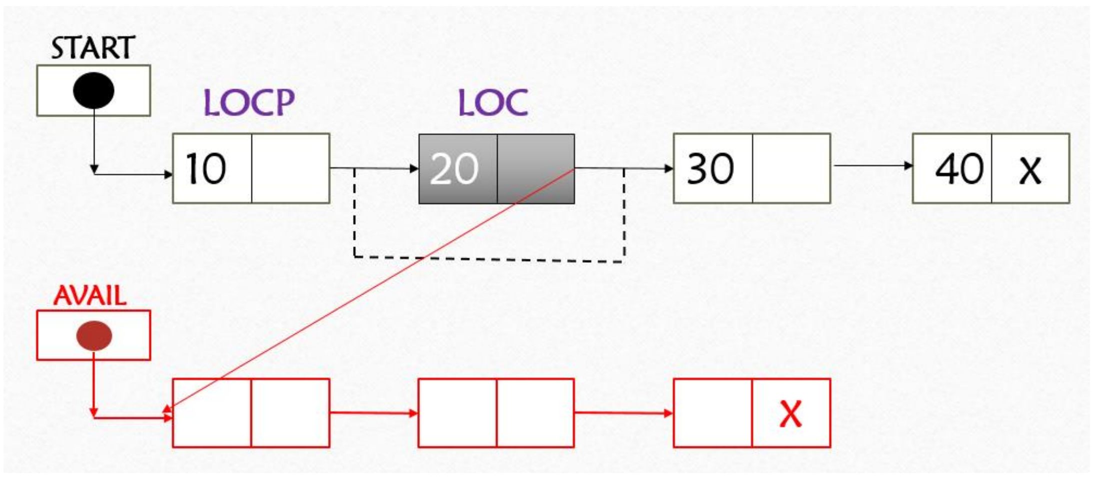
Deletion operation refers to removing existing node from linked list. Once the element is removed, the memory should be reclaimed and the links must be updated.

While performing deletion following possibilities must be considered :

i. **Empty List :** If START pointer contains NULL address, there is no node to delete & hence the condition is underflow.



ii. **Deletion at any location :** If list contains multiple nodes, node at any location can be deleted and after the deletion the previous node's address part should be updated.

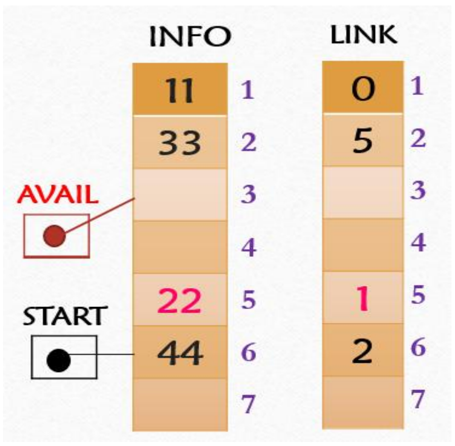


Algorithm : DELETE_LIST(INFO, LINK, AVAIL, PTR, ITEM, LOC, LOCP)

Description : Let LIST be the linked list in memory with INFO, LINK, AVAIL & START pointers. LOC indicates the location from where element is to be deleted & LOCP indicates the previous node. When LOCP=NULL then node to be deleted is the first node.

This algorithm deletes an existing element from given location.

```
Step 1 : [UNDERFLOW ?] If START = NULL then
             Write UNDERFLOW & Exit
Step 2 : [Deleting first node]
         If LOCP = NULL then
             Set START := LINK[START]
         Else
             Set LINK[LOCP] := LINK[LOC]
Step 3 : Set LINK [LOC] := AVAIL
Step 4 : Set AVAIL := LOC
Step 5 : EXIT
```



Item to be deleted : 22

22 is at position 2, thus LOC=5

Previous Element : 33

33 is at position 2, thus LOCP = 2

Before Deletion :

44	33	22	11

After Deletion :

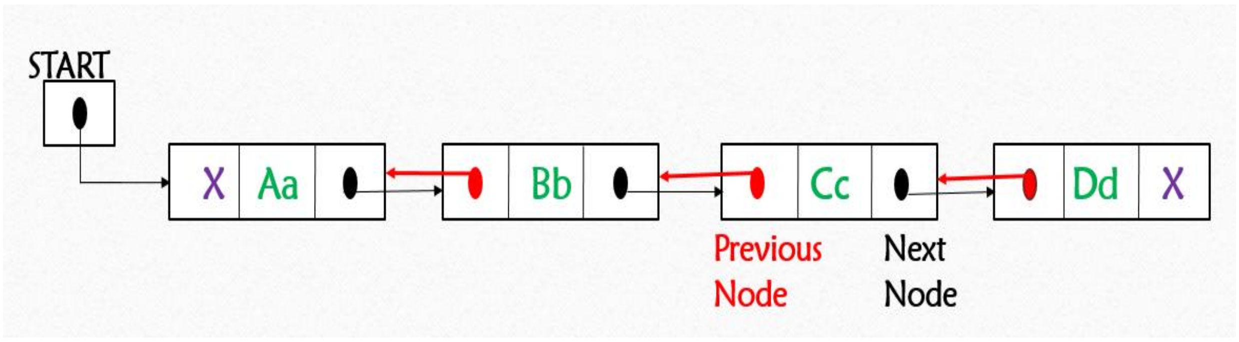
44	33	11

Two – way linked list :

Two – way linked list is a type of linked list in which each node apart from storing its data has two links.

The first link points to the previous node in the list and the second link points to the next node in the list.

The first node of the list has its previous link pointing to NULL similarly the last node of the list has its next node pointing to NULL.



The two links help us to traverse the list in both backward and forward direction. But storing an extra link requires some extra space.

A two way list is a linear collection of data elements, called nodes, where each node N is divided into three parts : Information field, Forward link- which points to the next node and Backward link-which points to the previous node.

Two-way linked list is also called as doubly linked list. Doubly Linked List can also be used as Circular List where the last node points to the first node.

Two-way linked list is used by browsers to implement backward and forward navigation of visited web pages i.e. back and forward button. It is also used by various application to implement Undo and Redo functionality.