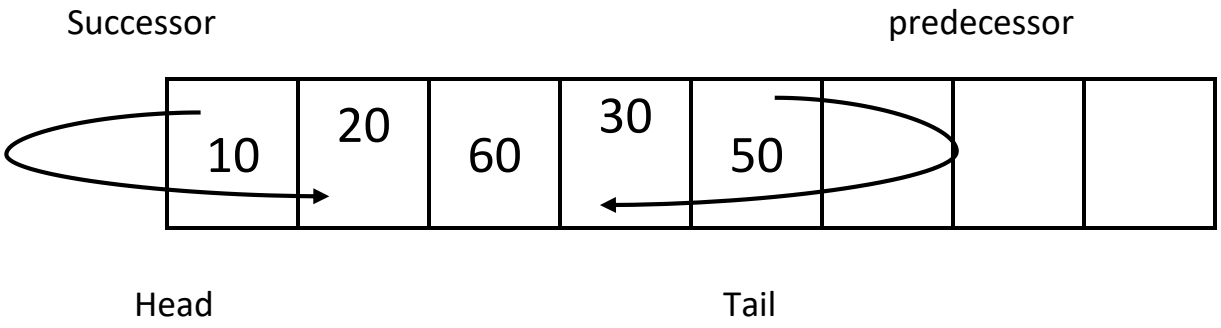
Chapter 3 List

Definition

- List means collection of elements in sequential order to which addition & deletion can be made.
- The first element of a list is called the head of list & the last is called the tail of the list.
- The next element of the head of the list is called its successor. The previous element to the tail (if it is not head of the list) is called its predecessor. Clearly a head doesn't have as predecessor & a tail doesn't have a successor. Any other element of the list has both one successor & one predecessor.

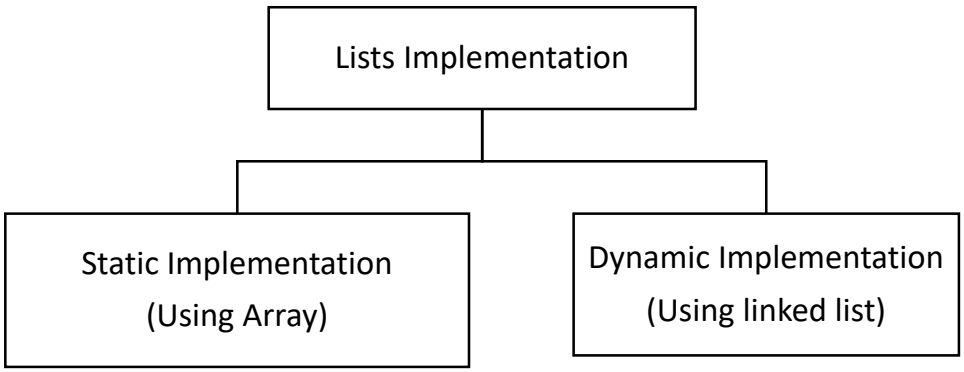


Operations perform in list:-

1. Traversing an array list
2. Searching an element in the list
3. Insertion of an element in the list
4. Deletion of an element in the list.

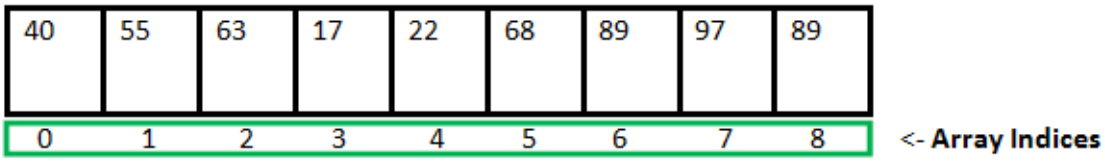
- In memory we can store the list in two ways.
- One way to store the elements at sequential memory location. This implementation is called **static implementation** and is done using array.
- The other way is we can use pointers or links to attach the elements sequentially. This is called **dynamic implementation**.

Chapter 3 List



Storing a list in a static data structure:

- This implementation stores the list in an array.
- The position of each element is given by an index from 0 to n-1, where n is the number of elements.
- Given any index, the element with that index can be accessed in constant time – i.e. the time to access does not depend on the size of the list.
- To add an element at the end of the list, the time taken does not depend on the size of the list. However, the time taken to add an element at any other point in the list does depend on the size of the list, as all subsequent elements must be shifted up. Additions near the start of the list take longer than additions near the middle or end.
- When an element is removed, subsequent elements must be shifted down, so removals near the start of the list take longer than removals near the middle or end.

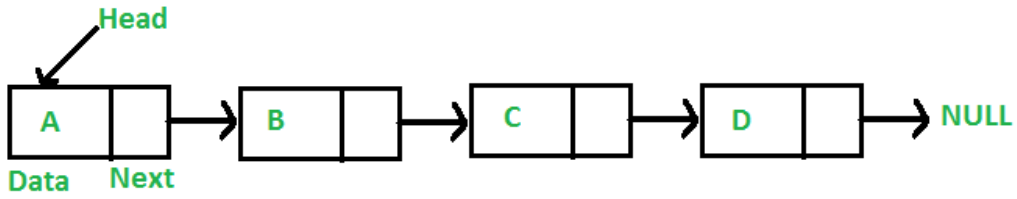


Array Length = 9
First Index = 0
Last Index = 8

Chapter 3 List

Storing a list in a dynamic data structure (Linked List):

- The Link List is stored as a sequence of linked nodes. As in the case of the stack and the queue, each node in a linked list contains data and a reference to the next node.
- The list can grow and shrink as needed the position of each element is given by an index from 0 to n-1, where n is the number of elements.
- Given any index, the time taken to access an element with that index depends on the index. This is because each element of the list must be traversed until the required index is found.
- The time taken to add an element at any point in the list does not depend on the size of the list, as no shifts are required. It does, however, depend on the index. Additions near the end of the list take longer than additions near the middle or start. The same applies to the time taken to remove an element.
- The first node is accessed using the name LinkedList.Head
- Its data is accessed using LinkedList.Head.DataItem



Static Data Structure vs Dynamic Data Structure:

- Static Data structure has fixed memory size whereas in Dynamic Data Structure, the size can be randomly updated during run time which may be considered efficient with respect to memory complexity of the code. Static Data Structure provides easier access to elements with respect to dynamic data structure. Unlike static data structures, dynamic data structures are flexible.

Chapter 3 List

Array Implementation of List:

- This implementation stores the list in an array.

```
#define NUMNODES 500
struct nodetype{
    int info, next;
};
struct nodetype node[NUMNODES];
```

Insertion:

- Input the array elements, the position of the new element to be inserted and the new element.
- I nsert the new element at that position and shift the rest of the elements to right by one position.

Algorithm

1. Get the **element value** which needs to be inserted.
2. Get the **position** value.
3. Check whether the position value is valid or not.
4. If it is **valid**,

   Shift all the elements from the last index to position index by 1 position to the **right**.

   insert the new element in **arr[position]**
5. Otherwise,

   Invalid Position

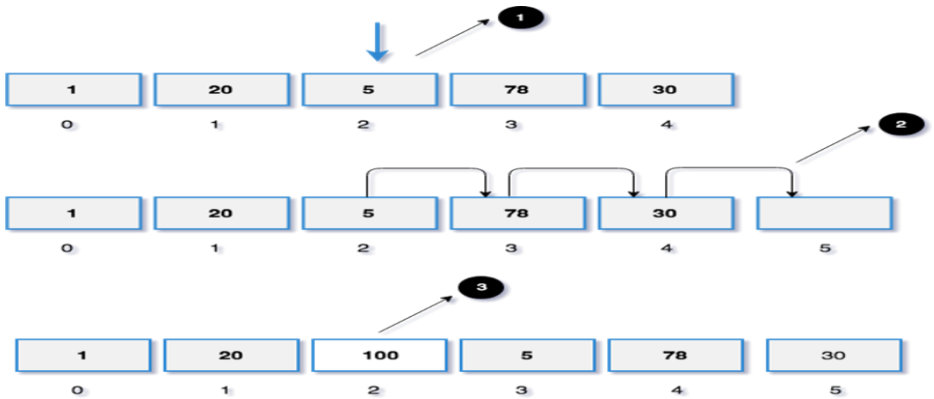
```
int getnode()
{
    int p;
    if(avail == -1){
        printf("overflow \n");
        exit(1);
    }
    p= avail;
    avail= node[avail].next;
    return(p);
}
```

Example1:

Let's take an array of 5 integers.

1, 20, 5, 78, 30.

If we need to insert an element 100 at position 2, the execution will be,



Deletion

- Input the array elements, the position of the element to be deleted and the element.
- Delete the element and shift the rest of the elements to left by one position.

Algorithm

1. Find the given element in the given array and note the index.
2. If the element found,

   Shift all the elements from index + 1 by 1 position to the left.

   Reduce the array size by 1.
3. Otherwise, print "Element Not Found"

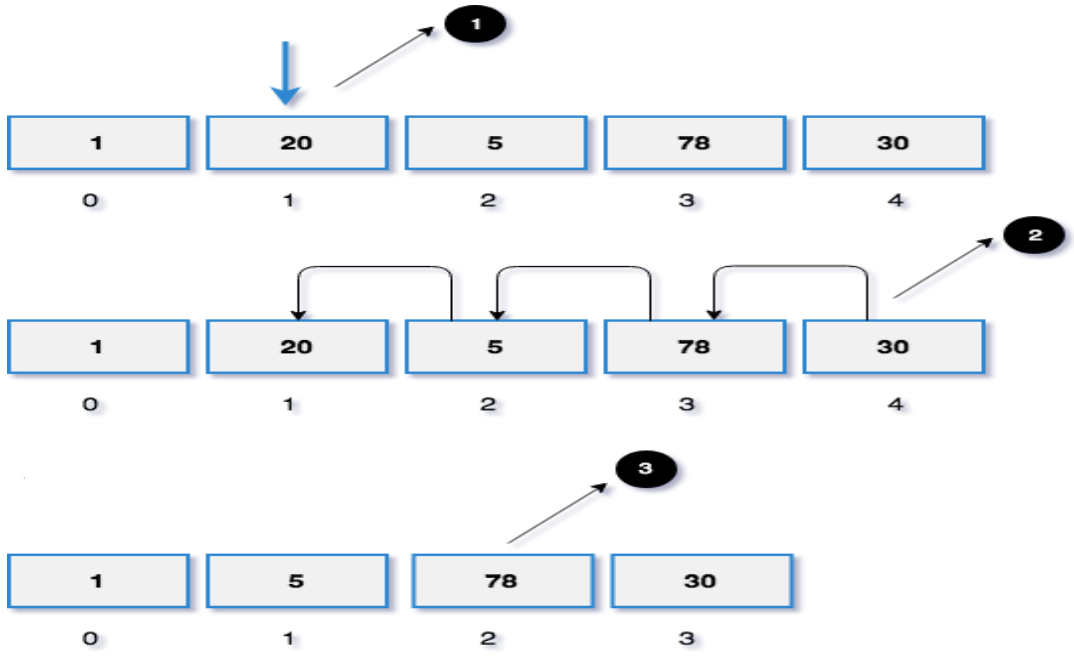
```
void freenode()
{
    node[p].next=avail;
    avail=p;
    return;
}
```

Example1:

Let's take an array of 5 elements.

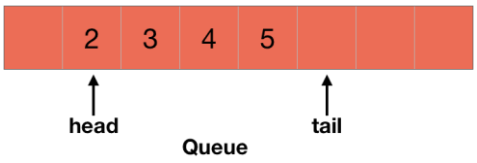
1, 20, 5, 78, 30.

If we remove element 20 from the array, the execution will be,



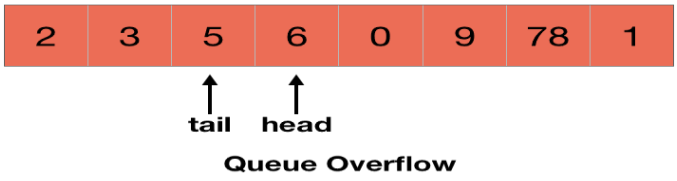
Queue as list:

We will maintain two pointers - *tail* and *head* to represent a queue. *head* will always point to the oldest element which was added and *tail* will point where the new element is going to be added.



Insertion:

```
if Q.head=Q.tail+1
      Error “Queue Overflow”
  else
      Q[Q.tail] = x
      if Q.tail == Q.size
          Q.tail = 1
      else
          Q.tail = Q.tail+1
```



Deletion:

```
if Q.tail=Q.head
    Error “Queue Underflow”
else
    x = Q[Q.head]
    if Q.head == Q.size
        Q.head = 1
    else
        Q.head = Q.head+1
    return x
```

An Empty Queue