ADT Sepcification for a "List".

List

└→ Operation : ADD :

12, 15, 17

* Pre Cond : A List should be exist.

Input : The item to be added. — 25

Process : The item will be added to the end of the list.

Output : Success of addition / None.

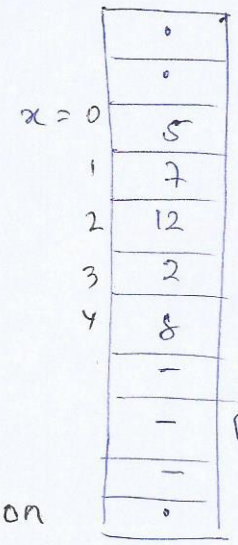
Post Cond : List with an added item. — 12, 15, 17, 25

As ADT Specifications we should understand,

* ① How Data is stored.

* ② Operations.

Arrays



Row major ordering.	Column major ordering.
[1][1]	[1][1]
[1][2]	[2][1]
[1][3]	[3][1]
[1][4]	[4][1]
[1][5]	[5][1]
[2][1]	[1][2]
[2][2]	[2][2]
[2][3]	[3][2]

Student Index No	S_1M_1	S_2M	S_3M	. . .
1	25	75	98	58
2	75	25	72	73
3	50	65	83	28
4	55	* 85	44	22
5	100	100	86	31

2 dimention

A [i][j]

$= x + (i-1) *$ Size of a row

$+ (j-1) *$ Size of an element.

row major ordering.

A [1][2]

Row Colomum.

ex: If I want ~~to~~ access * value,

$= x + (4-1) *$ Size of a row

$+$ ~~2~~ $(3-1) *$ Size of an element

3 ways we can store data.

- 2D Array of String
- Array of Records.
- Different arrays for different records.

ADT - Linked List

2 Structure implementing a LinkedList.

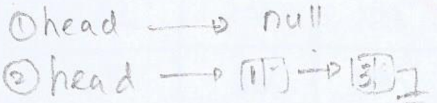
```
① Record /Struct   Node {
          int : data.
        * Node : next.
  }.

② Record /Struct   LinkedList {
        * Node head
  }.
```

ex; LinkedList

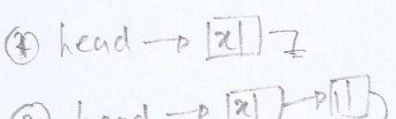
```
LIST_CREATE ( ) {
    LinkedList l = new LinkedList ( );
    l ⟶ head = NULL
    return l.
}.
```

```
                              Type of the Para.     Name of the Para.
Void  LIST_INSERT  ( LinkedList L,  int x ) {
method    method name          Parameters.

    Node  newNode = newNode ( )
    newNode -> data = x
    newNode ⟶ next = head;
    L ⟶ head = newNode. }
```

This Linked List L can have 2 scenarios

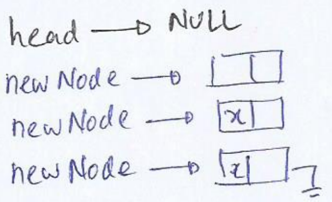


After adding x.

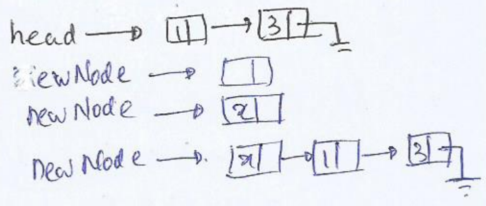


Above Process for,

①st Scenario.



②nd Scenario.



```
Boolean LIST_SEARCH (LinkedList L, int x){
        Node temp = L -> head
        while (Temp != NULL){
            if (temp -> data == x)
                return TRUE
            temp = temp -> next;
        }
        return FALSE
};
```

If the 1st element we have to delete

```
Void LIST-DELETE (LinkedList L, int x){
        if L -> head == NULL
            return.
        if L -> head -> data == x {
            Node temp = L -> head
            L -> head = L -> head -> next
            delete (temp)
            return.
        }.
```

If its in the middle.

```
        Node Previous = head.
        while (prev -> next != NULL){.
            if (Prev -> next -> data == x){
                Node temp = prev -> next
                    Prev -> next = temp -> next
                    delete (temp)
                    return.
            }.
          Prev = prev -> next
        }
}
```

Doubly linked list

Slide

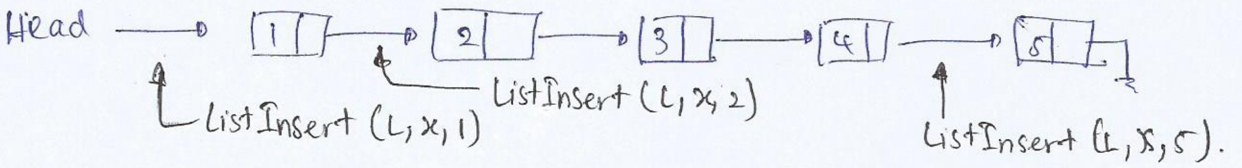
```
List_ CREATE() {.
    DoublyLinkedList l = new LinkedList ();
    l → head = NULL
    return l.
}.

Void LIST_INSERT (DoublyLinkedList L, int x) {.
        Node newNode = newNode( )
        newNode → data = x.
        NewNode → next = head;
        NewNode → Prev. = Null

        head → Prev = NewNode
        L → head = newNode.
    }.
```

Searching is Same.

In Deletion,

```
        ⋮
    If L → head, data == x {
        L → head → nex → prev = NULL.
        L → head = L → head → next .
          ⋮
          Node temp = Prev → next
          temp → next → prev = temp → Prer.
          Prev → next = temp → next.
            ⋮
```

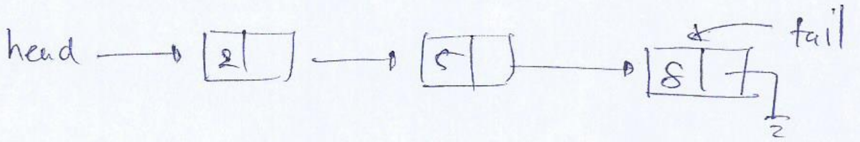
List Insertion for ith.



```
List_Insert (LinkedList L, int x, int location)
else Node' newNode = newNode ();  → if (loc == 1)
                                       List Insert (L, x)
    newNode → data = x
    int i=2;
    Curr = head
    while (i < loc && Curr → next != NULL)
        Curr = Curr → next;
    NewN → next = Curr → next
    Curr → next = NewN
```

Queue

```
Struc Node{                    Struck Queue{
    dat                            * Node head
    next                           * Node fail
}.                             }.
```



Stacks

```
Struct Stack{
        Node top
}.
```