Linked List.

⇒ Arrays

↳ Limitations

→ ① Fixed Size. → int[] arr = new int[3]; → size is fixed.

→ ② Insertion / Deletion → | 10 | 20 | 30 | 40 |

Insertion Not Possible.

| 10 | 20 | 30 | 40 | 50 | |

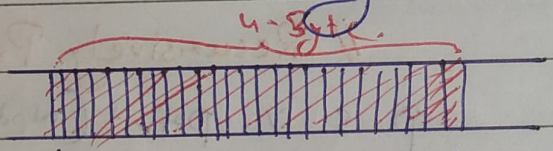
35

| 10 | 20 | 30 | 35 | 40 | 50 |

Insertion is Not easy.

→ ③ Continuous Memory Allocation.

int x = 9 ⇓ x will take 4 bytes in memory



Memory will be allocated continuously.

⇒ Array Lists → Not Truly unlimited size.

list = | 10 | 20 | 30 | 40 |

list = | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 |

→ Size & Capacity

→ list.add(50)

→ list.add(60)

[Overcomes this limitations]

Linked List.

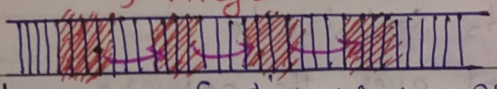
→ Continuous memory Allocation ✗

→ Insertion, Deletion with O(1) space.

→ Truly Unlimited Size.

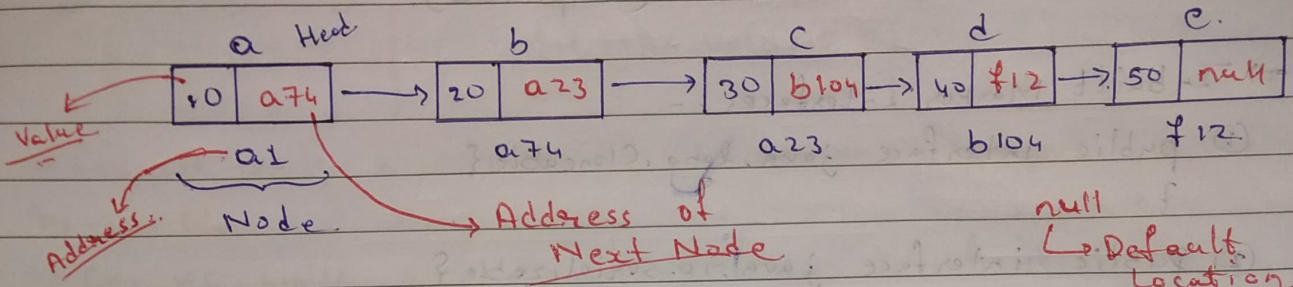
⇒ Intro to {Linked List} → One sided love

5 integer to store.



Linking two Non-Contiguous memory Locations (Nodes).

↳ Node → Value & Address of Next Node.



⇒ Code:

Creation / Printing.

```
Class Node{
    int val;
    Node next;
    Node(int val){
        this.val = val;
    }
}
public Class ListNode{
Public static void main( ){
    Node a = new Node(10);
    Node b = new Node(20);
    Node c = new Node(30);
    Node d = new Node(40);
    a.next = b; // 10 → 20
    b.next = c; // 10 → 20 → 30
    c.next = d; // 10 → 20 → 30 → 40.
    print(a);

    //Sout(a.val); // Normal Printing
    //Sout(a.next.val);
    //Sout(a.next.next.val);
    //Sout(a.next.next.next.val);
}
```

Creating Node

Linking

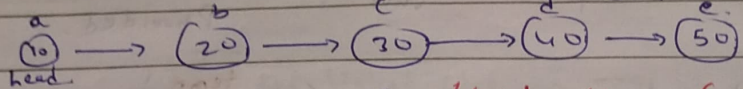
```
// Printing Using Loop.
public static void print(Node head){
    Node temp = head;
    while(temp != null){
        Sout(temp.val);
        temp = temp.next;
    }
}
```

```
// Recursively Printing.
p s v print(Node head){
    if(head == null) return;
    Sout(head.val);
    print(head.next);
}
```

↳ Rearrange for Reverse printing

Displaying a Linked List.

→ Shallow Copy of node (temp).
↳ Used to transverse the LL.



① Node temp = a; ✓ // Shallow Copy.

② Node temp = new Node(10); ✗ // Deep Copy.

=> Code :-

```
Public Class ShallowCopy {
    psvm() {
        Node a = new Node(100); // Node Creation
        Node temp = a ; // Shallow Copy
        // Node temp = new Node(a.val); // Deep Copy
        temp.val = 12;
        Sout(a.val);
        Sout(temp.val);
        a.val = 100;
        Sout = (tem.val);
    }
}
```

Reference.

Output :-
12
12
100

InsertAtEnd / Tail Method. → Node Class is already created in previous section

=> Code :-

```
Class SimplyLL{
    private Node head;
    Node tail;
    private int size; // list.size will Not work due to private

    Void insertAtTail(int val){
        Node temp = new Node(val);
        if(head == null) head = tail = temp;
        else{
            tail.next = temp;
            tail = temp;
        }
        size++; // Good practice No need to create size method. if it is public
    }

    Void display(){
        Node temp = head;
        while(temp != null){
            Sout(temp.val + " ");
            temp = nex.tep;
        }
        Sout();
    }
}

Public Class Implementation(){
    psvm(){
        SimplyLL list = new SimplyLL();
        list.insertAtTail(10);
        list.insertAtTail(20);
        list.display();
        list.sizemethod();
    }
}
```

Output :-
10 20.
2.

↳ It is in Next Code

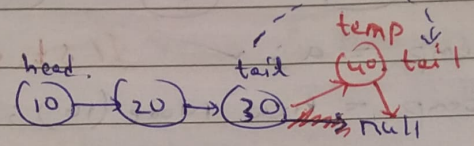
Note :-

① if LL is empty.
↳ head = tail = temp.

T.C = O(1)
S.C = O(1)
↳ when we have tail

T.C = O(n)
↳ if tail is Not Provided.

② if LL is Not empty
↳ tail.next = temp
↳ tail = temp.



Insert At Begining / Head

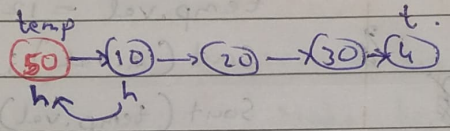
Method ~~Function~~.

Note :-

T.C = O(1)
S.C = O(1).

① if LL is empty
↳ head = tail = temp.

② LL is not empty.
↳ temp.next = head
↳ head = temp.



⇒ Code :-

```
Class SimpLL{
    Node head;
    Node tail;
    int size;

    void insertAtHead(int val){
        Node temp = new Node(val);
        if(head == null) head = tail = temp;
        else {
            temp.next = head;
            head = temp;
        }
        size++;
    }

    void sizemethod(){
        size = 0;
        Node temp = head;
        while(temp != null){
            size++;
            temp = temp.next;
        }
        Sout(size);
    }
}
```

```
public Class Implementation{
    psvm(){
        SimpLL List = new SimpLL();
        list.insertAtHead(70);
        ~~list.insertAtHead(80);~~
        list.display();
        list.sizemethod();
        list.insertAtHead(80);
        list.display();
        list.sizemethod();
        Sout(list.size);
    }
}
```

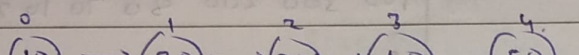
Output :-

```
70
1
80 70
2
2
```

Display Method
- T.C ⇒ O(n).
- S.C ⇒ O(1)

Size Method
- Implemented Linked List → T.C ⇒ O(1).
- If only head is printed → T.C ⇒ O(n).

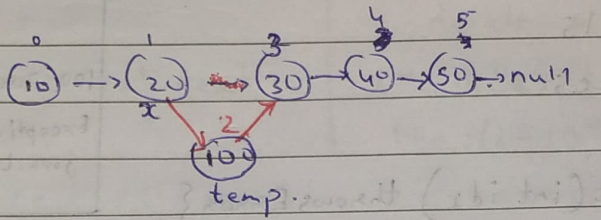
Insert Method [0-Based Indexing]



list.insert(2, 100).

temp.next = x.next.

x.next = temp.



Create a temporary node(x).
↓
Transverse x till "idx-1" pass.

T.C = O(n)
S.C = O(1)

⇒ Code:-

```
Class SinglyLL{
    Node head;
    Node tail;
    int size;
    void insert(int idx, int val){
        Node x = head;
        Node temp = new Node(val);
        if(idx == 0){
            insertAtHead(val);
            return;
        }
        if(idx == size){
            insertAtTail(val);
            return;
        }
        if(idx > size || idx < 0){
            Sout("Invalid Index");
            return;
        }
        for(int i = 1; i < idx; i++){
            x = x.next;
        }
        temp.next = x.next;
        x.next = temp;
        size++;
    }
}
```

```
public Class Implementation{
    psvm(){
        SinglyLL list = new SinglyLL();
        list.insert(0, 100);
        list.display();
        list.insert(1, 200);
        list.display();
        list.insert(1, 300);
        list.display();
    }
}
```

Output :-

100

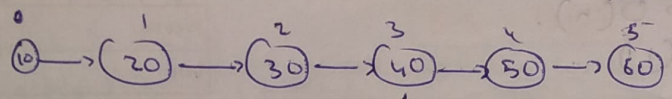
100 200

100 300 200

Note 1-
- if(idx == 0) → insert at Head.
- if(idx == size) → insert at Tail.
- if(idx > size || idx < 0) → Throw Error. ↳ Invalid Index.

Get Element Method

T.C = O(n)



sout(list.get(idx(3)));

3 Base Cases → idx→0, idx→size-1 | idx >= size || idx < 0

↓ extras

Error

=> Code :-

```
Class SimpLL{
    Node head;
    Node tail;
    int size;

    int get(int idx) throws Error{
        if(idx >= size || idx < 0){
            throw new Error("Bhai Error Hai Index Mai");
        }
        if(idx == size-1) return tail.val;
        Node temp = head;
        for(int i=0; i<idx; i++){
            temp = temp.next;
        }
        return temp.val;
    }
    int set(int idx, int val) throws Error{
        if(idx >= size || idx < 0){
            throw new Error("Bhai.....");
        }
        if(idx == size-1){
            tail.val = val;
        }
        Node temp = head;
        for(int i=0; i<idx; i++){
            temp = temp.next;
        }
        temp.val = val;
    }
}
```

```
Public Class Implementation{
    psvm(){
        SimpLL list = new SimpLL();
        list.display();
        Sout(list.get(10));
        list.set(2, 90);
        list.display();
        Sout(list.get(11));
    }
}
```

Output :-

100 300 200 80 70 10 20 30 40 50 60

60.

100 300 90 80 70 10 20 30 40 50 60.

Exception in thread "main".
java.lang.Error: Bhai Error Hai Index Mai.

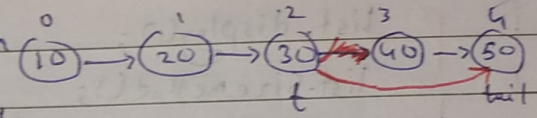
An Evident Limitation of LinkedList.

- To get, the T.C is O(n).
  - In Array, it is O(1).
- To set, the T.C = O(n).

---

deleteAtIndex Method.

deleteAtHead() ⟶ h = h.next.
size--.



deleteAtIndex(3)
temp.next = temp.next.next.
size--.

To delete a node at i^{th} index; we need the: $(i-1)^{th}$ node

=> Code :-

```
Class SimplyLL{
    Node head;
    Node tail;
    int size;

    void deleteAtHead() throws Error{
        if(head == null) throw new Error("Bhai List Khali Hai");
        head = head.next;
        size--;
        Sout();
    }

    Void delete(int idx) throws Error{
        if(idx == 0){ deleteAtHead();
            return;
        }
        if(idx < 0 || idx >= size){
            throw new Error("Bhai Index Invalid Hai");
        }
        Node temp = head;
        for(int i = 1; i < idx; i++){
            temp = temp.next;
        }
        if(temp.next == tail) { tail = temp;
        temp.next = temp.next.next;
        size--;
    }
}
```

```
Public Class Implementation {
    psvm() {
        SimplyLL list = new SimplyLL();
        list.disply();
        list.deleteAtHead();
        list.display();
        list.SizeMethod();
        list.delete(5);
        list.display();
        list.SizeMethod();
        list.delete(8);
        list.display();
        list.SizeMethod();
        sout(list.tail.val);
    }
}
```

Outputs:

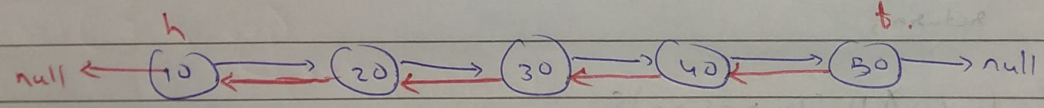
```
100 300 90 80 70 10 20 30 40 50 60
300 90 80 70 10 20 30 40 50 60
10
300 90 80 70 10 30 40 50 60
9
300 90 80 70 10 30 40 50
8
60
```

Limitations of Singly Linked List

get → O(n)
set → O(n)
delete tail → O(n)

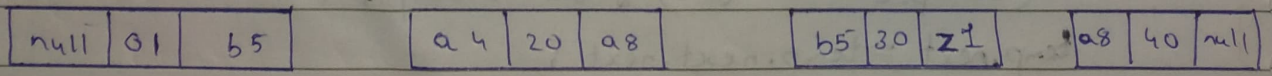
=> We cannot go back & we always need to traverse entire list.

Introducing Doubly Linked List



next →
Prev. ←

```
Class Node {
    int val;
    Node next;
    Node prev; // Extra
}
```

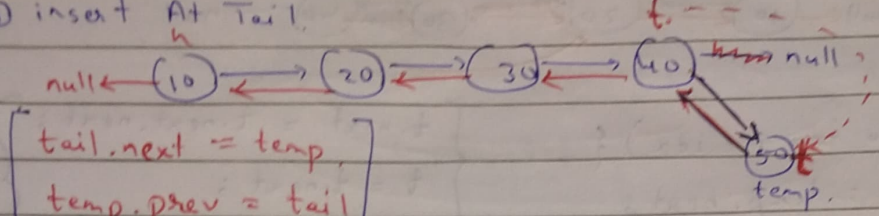


Note :- We Take up more memory

Implementation

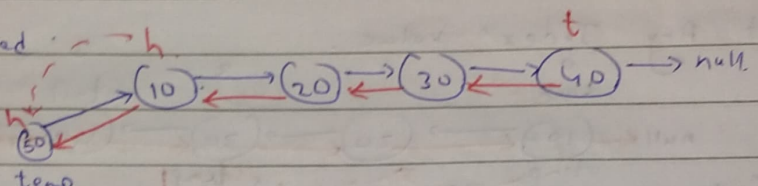
Insertion.

(1) insert At Tail.



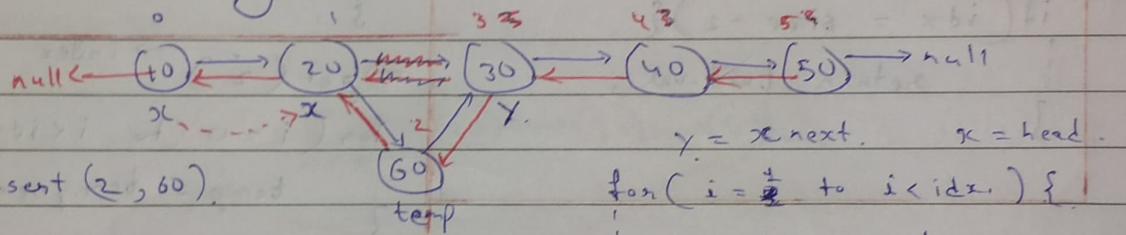

```
tail.next = temp.
temp.prev = tail
tail = temp.
```

(2) insert At Head :-




```
temp.next = head.
head.prev = temp.
head = temp.
```

(3) insert At Any index



insert (2, 60).

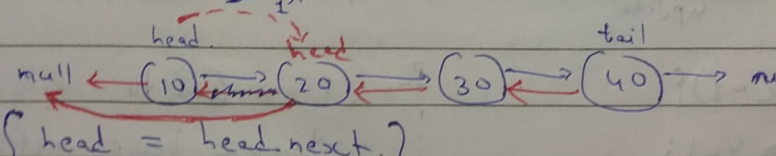
```
y = x.next.        x = head.
for ( i = 1 to i < idx. ) {
    x = x.next
}
```

=> Place x & y at idx-1 & idx respectively.

```
{ => x.next = temp , temp.prev = x.
  => y.prev = temp , temp.next = y // }
```

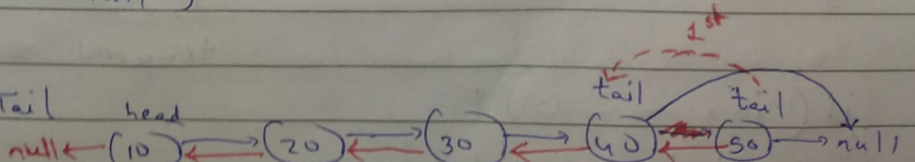
Deletion :-

(1) Delete at Head



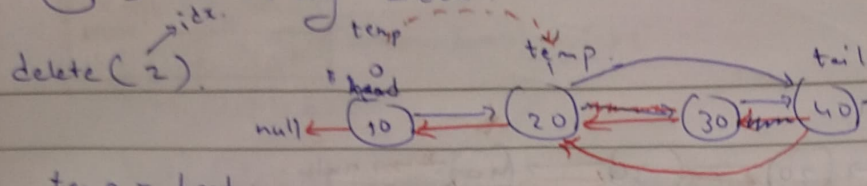

```
{ head = head.next.
  head.prev = null }
```

(2) Delete at Tail




```
{ tail. = tail.prev.
  tail.next = null. }
```

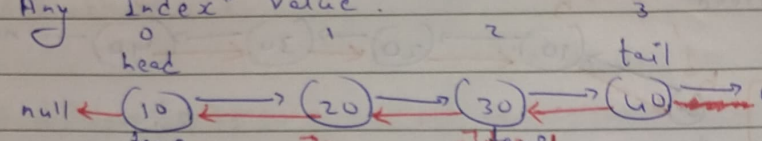
(3) Delete At Any Index.




```
temp = head;
for( i = 1 to ~~idx~~ i < idx) {
    temp = temp.next;
}
```

```
t.next = t.next.next;
t = t.next;
t.prev = t.prev.prev;
```

Get Any Index' value.




```
if(idx >= size || idx < 0) {
    Throw Error
}
```

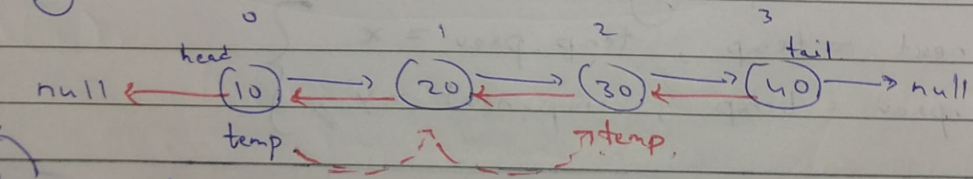
```
if( idx = size - 1){
    return tail.val;
}
```

```
if( idx == 0){
    return head.val;
}
```

⇒ get(2) → index

```
for( i = 0 to i < idx) {
    temp = temp.next;
}
return temp.val;
```

Set Any Index Value.



index

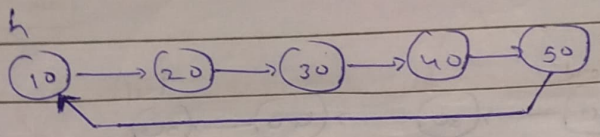
set(2, 60) → Value that need to insert/update

```
if( idx == size - 1){
    head.val = val;
}
```

```
if( idx == 0) {
    tail.val = val;
}
```

```
for( i = 0 ~~= 1~~ to i < idx){
    temp = temp.next;
}
temp.val = val;
```

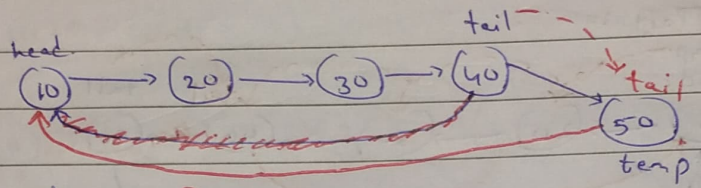
Singly Circular Linked-List.



Implementation

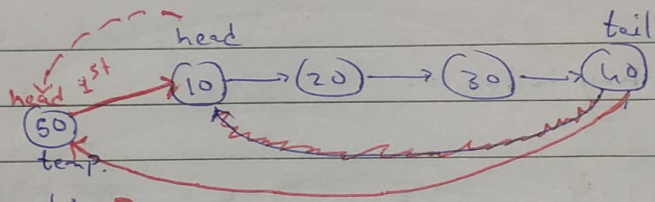
Insertion :-

(1) Insert At Tail. :-



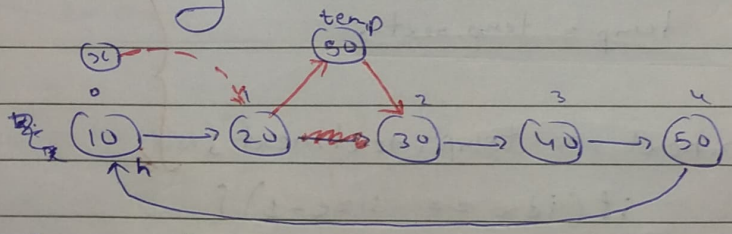

```
tail.next = temp;
tail = temp;
tail.next = head;
```

(2) Insert At Head. :-




```
temp.next = head;
head = temp;
tail.next = head;
```

(3) Insert At any index.



insert (2 , 50)
→ idx → val.

```
x = head.
for ( int i = 1 to i < idx ) {
    x = x.next;
}
```

```
temp.next = x.next;
x.next = temp;
```

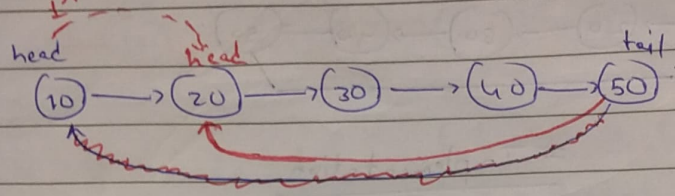
```
if ( idx == 0 ) {
    insertAtHead(val);
}
```

```
if ( idx == size ) {
    insertAtTail(val);
}
```

Deletion :-

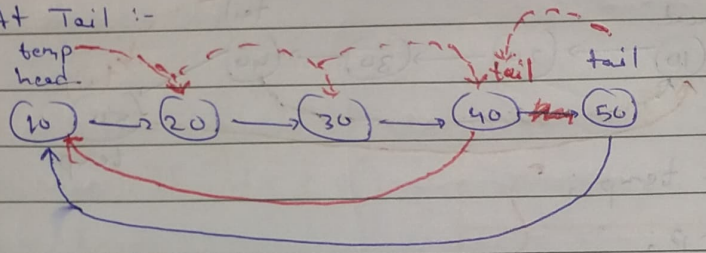
M	T	W	T	F	S	S
Page No.:						YOUVA
Date:						

(1) Delete At Head :-




```
head = head.next;
tail.next = head;
```

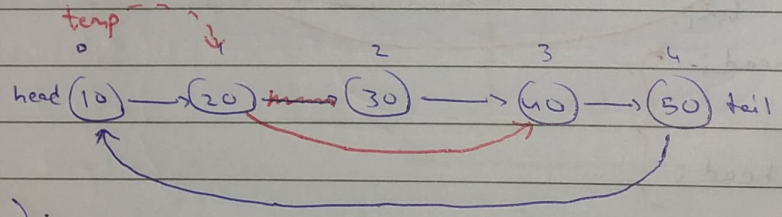
(2) Delete At Tail :-




```
temp = head;
for(int i = 1 to i < size-1){
    temp = temp.next;
}
```

```
temp.next = head;
tail = temp;
```

(3) Delete At ~~---~~ Any index :-



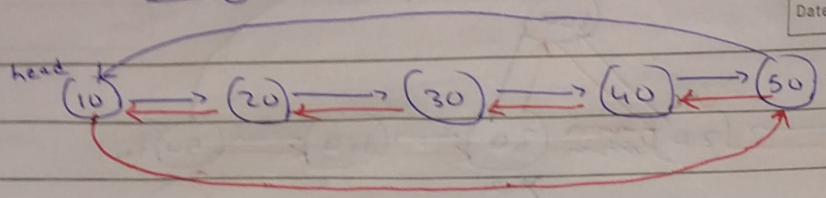

```
delete(2);
temp = head;
for(int i = 1 to i < idx){
    temp = temp.next;
}
temp.next = temp.next.next;
```

```
if(idx == 0){
    deleteAtHead();
}
```

```
if(idx == size-1){
    deleteAtTail();
}
```

Get & set methods are same like ~~to~~ doubly Linked-List

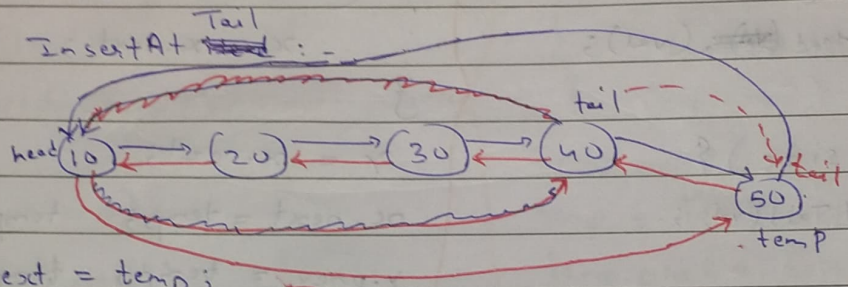
Doubly Circular Linked List



Implementation

Insertion :-

(1) InsertAt ~~Head~~ Tail :-

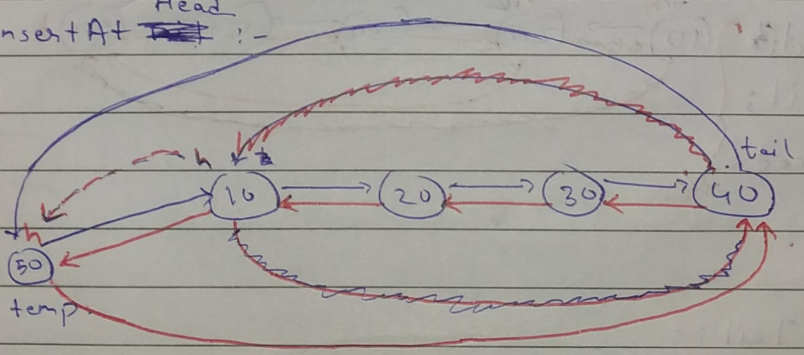



```
tail.next = temp;
temp.prev = tail;
~~tail.next~~ tail = temp;
tail.next = head;
head.prev = tail;
```

//

```
tail = head.prev;
```

(2) InsertAt ~~Head~~ :-

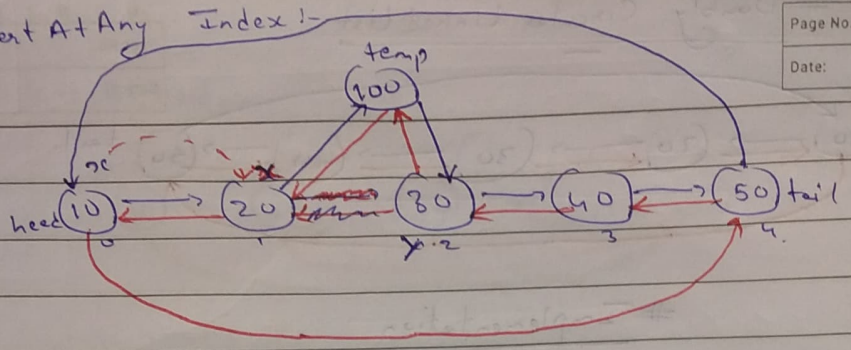



```
if tail is Not
  Given
tail = head.prev;
```

```
temp.next = head;
head.prev = temp;
head = temp;
head.prev = tail;
tail.next = head;
```

//

(3) Insert At Any Index :-

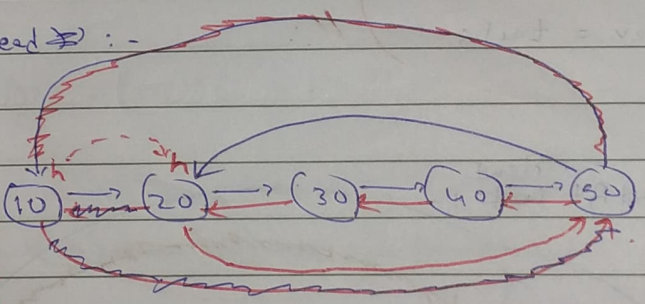



```
insert(2, 100);
if(idx == 0){
    insertAtHead(val);
}
if(idx == size){
    insertAtTail(val);
}
```

```
x = head;
for(int i = 1 to i < idx){
    x = x.next;
}
y = x.next;
x.next = temp;   temp.prev = x;
y.prev = temp;   temp.next = y;
```

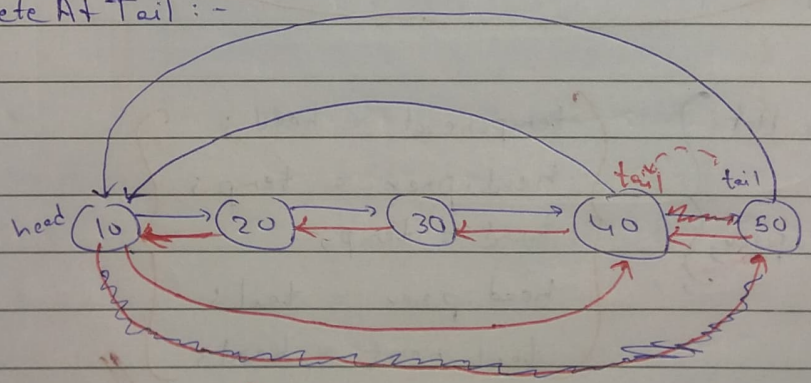
Deletion :-

(1) Delete At Head :-



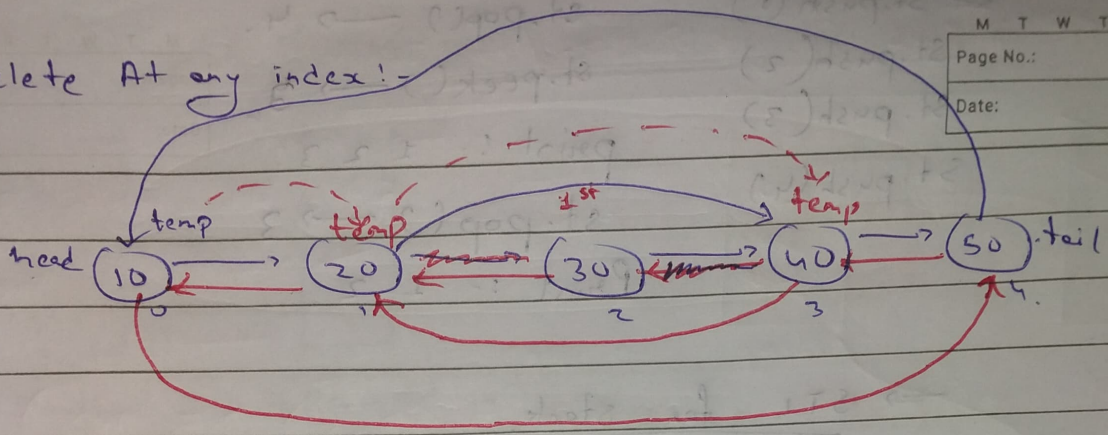

```
head = head.next;
head.prev = tail;
tail.next = head;
```

(2) Delete At Tail :-




```
tail = tail.prev;
tail.next = head;
head.prev = tail;
```

(3) Delete At any index :-




```
delete(2);

if(idx == size-1){
    deleteAtTail();
}
if(idx == 0){
    deleteAtHead();
}
```

```
temp = head.
for(int i = 1; i < idx){
    temp = temp.next;
}
temp.next = temp.next.next;
temp = temp.next;
temp.prev = temp.prev.prev;
```

[Get & set methods are same like doubly linked list]