Class - ①

→ LL Basics

→ Display, GetAt

→ Add → Last, First, At

→ Remove → Last, First, At

Class - ②

→ Reverse Linked list
- Data iterative
- Pointer iterative
- Data recursive
- Pointer recursive

→ Display Reverse

→ k Reverse

→ Palindrome LL

Linked list

Why LL?

- Arrays need contiguous memory allocation in heap.

So, maximum array size can only be the maximum available contiguous block of memory ($\leq 10^6$)

∴ Arrays are not practical when large data needs to be stored.

This causes an issue of fragmentation

So, Linked List (LL) is a solution to the above problem because it is a non-contiguous linear data structure

To keep track of these non-contiguous blocks of data, we use links.

Hence, the name Linked List.

For a linked list of integers, foll. class (blueprint of node) is established

```
public static class Node {
    int data;
    Node next;
}
```

Self referential data member

```
public static LinkedList {
    Node head;
    int size;
}
```

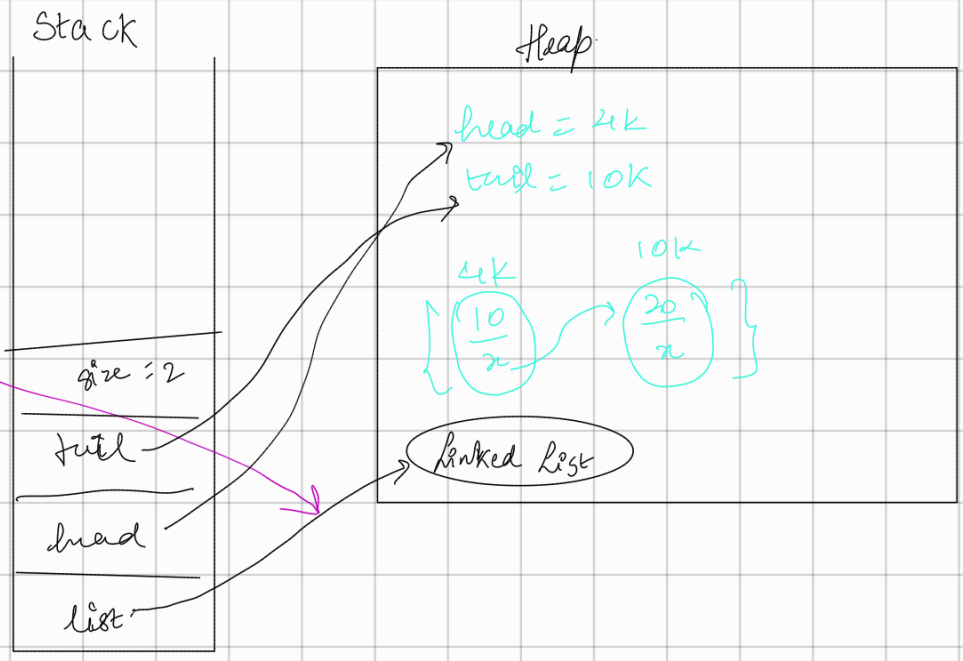
LinkedList list; → points to null

list = new LinkedList();

makes this line

list.add(10)

list.add(20)



To store 4 node LL: Space of 4 nodes,
Space of 4 pointers
Space of LL class.

Disadvantages of linked lists over arrays

1) Extra space for storing 1 node
2) Random access not available (in O(1) instead takes O(N))

Advantages of linked lists over arrays

1) Non-contiguous memory allocation
2) No problem of fragmentation

CODE of Linked list class:

```
public static class LinkedList {
    Node head;
    Node tail;
    int size;

    public void addFirst(int val) {
        Node temp = new Node();
        temp.data = val;

        if (size == 0) {
            // New node is first & last node
            head = temp;
            tail = temp;
        } else {
            tail.next = temp;
            tail = temp;
        }

        size++;
    }
```

```
public void addFirst(int val){
    Node temp = new Node();
    temp.data = val;

    if(size == 0){
      head = temp;
      tail = temp;
    } else {
       temp.next = head;
       head = temp;
    }
    size ++;
}

public void removeFirst(){
    if(size == 0){
        System.out.println("List is empty");
        return;
    }
    if (size > 1){
        head = head.next;
    } else {
        head = tail = null;
    }
    size --;
}

public void display(){
    if(size == 0)
        return;
```

```
        Node curr= head ;
        while(curr != null) {
            System.out.println(curr.data + " ");
            curr = curr.next;
        }
        System.out.println ();
    }

    public int getFirst () {
        if(size==0) {
            System.out.println("List is empty");
            return -1;
        }
        return head.data;
    }

    public int getAt (int idx){
        if(size == 0){
            System.out.println("List is empty");
            return -1;
        }
        if(idx<0  || idx >= size) {
            System.out.println("Invalid arguments");
            return -1;
        }
        Node curr = head;
        for(int i=0; i< idx; i++)
            curr. = curr.next;
        return curr.data;
    }
```

```
public void removeLast(){
    if(size==0){
        System.out.println("List is empty");
        return;
    }
    if(size==1)
        head=tail=null;
    else{
        Node prevTail = head;
        for(int i=0; i<size-2; i++)
            prevTail= prevTail.next;
        prevTail.next=null;
        tail= prevTail;
    }
    size--;
  }
}
```