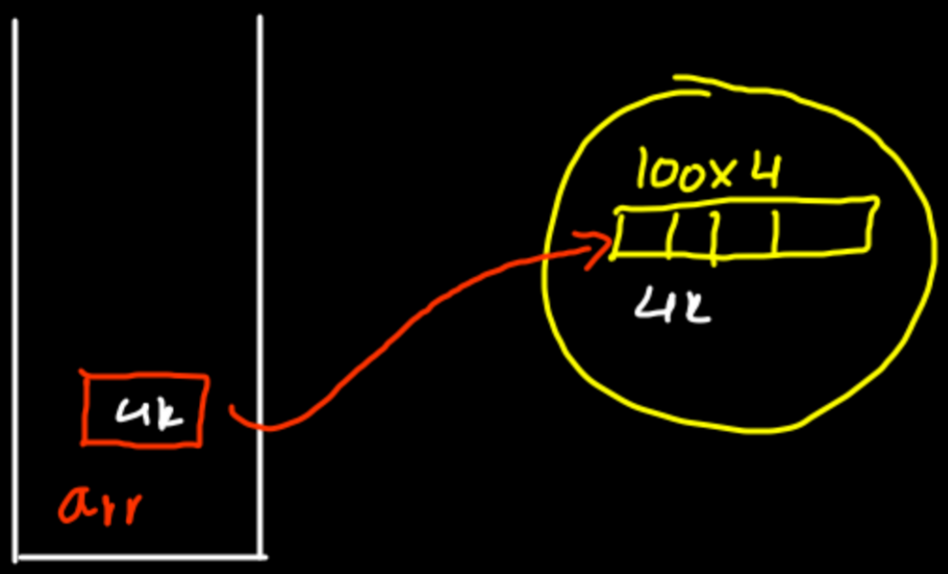
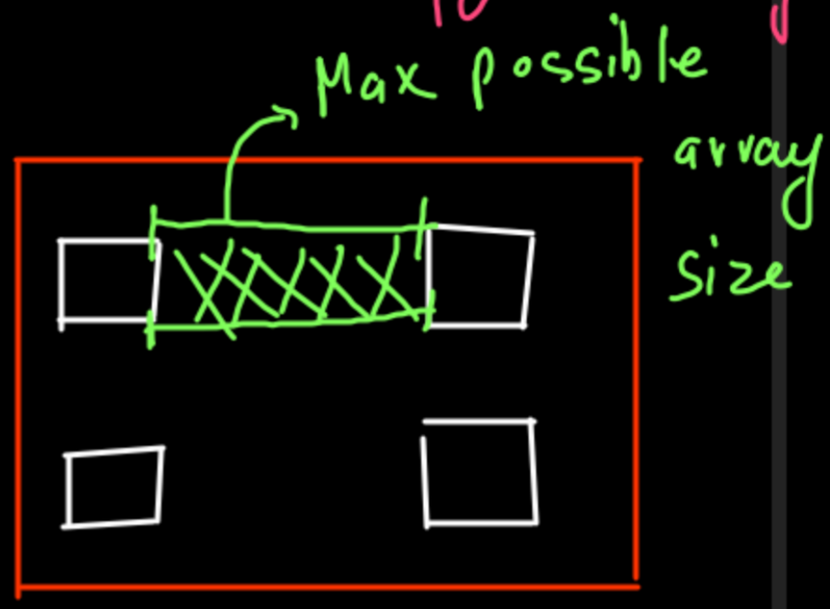
Linked List (Level 1+2)

Arrays

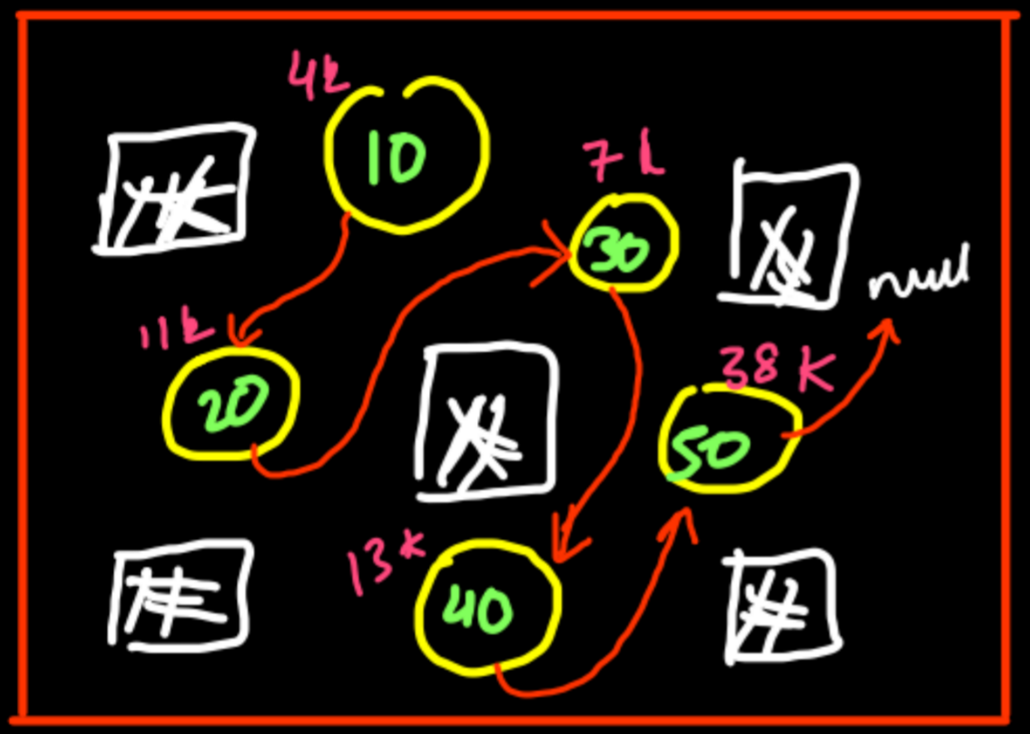
```
int [] arr = new int [100];
```





fragmentation issue (Memory got divided into fragments & enough Mem not avail)

Linked List: Non contiguous memory allocation.



We need links to connect data.

10, 20, 30, 40, 50

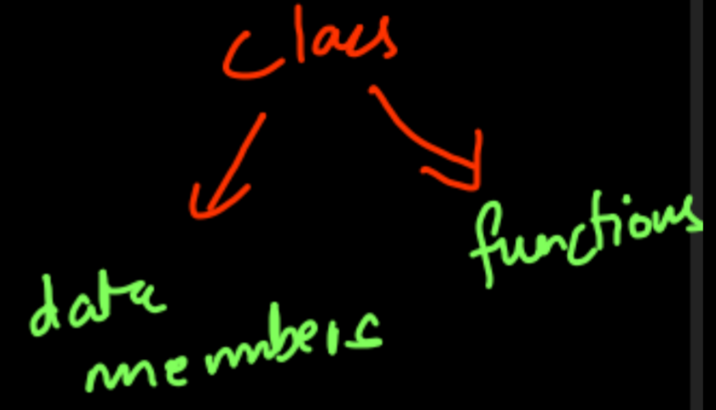
4k 11k 7k 13K 38k

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

```
Node{object}{
  int data;
  Node next;   → Address of Next Node
}
```

Inner class ko static banana hai.

```java
public static class Node {
    int data;
    Node next; //self referential data member
}
```



We only need one node (first Node) to get all the elements of the linked list because second ele will get from first's next & so on.

```java
public static class LinkedList {

    Node head;

    public void display() {

    }

    public void addFirst(int data) {

    }

    public void addLast(int data) {

    }

    public int get(int index) {

    }

}
```

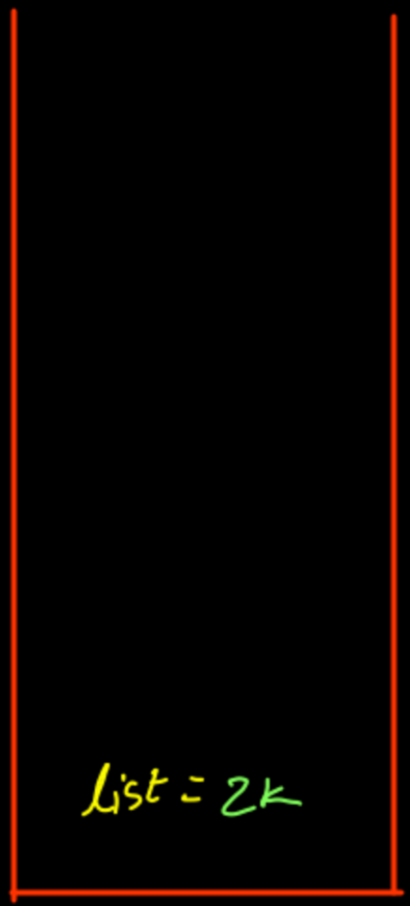
```java
public static class Node {
    int data;
    Node next; //self referential data member
}
```

```java
public static class LinkedList {

    Node head;
    Node tail;
    int size;

    public void display() {

    }

    public void addFirst(int data) {

    }

    public void addLast(int data) {

    }

    public int get(int index) {

    }

}
```

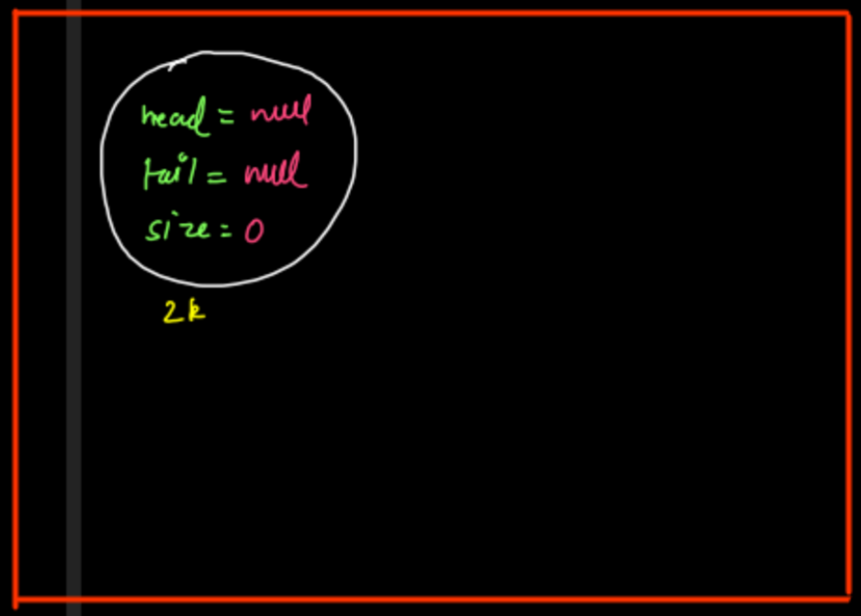
psv main() {

LinkedList list = new LinkedList();

}



Stack



Heap

```java
public static class Node {
    int data;
    Node next; //self referential data member
}
```

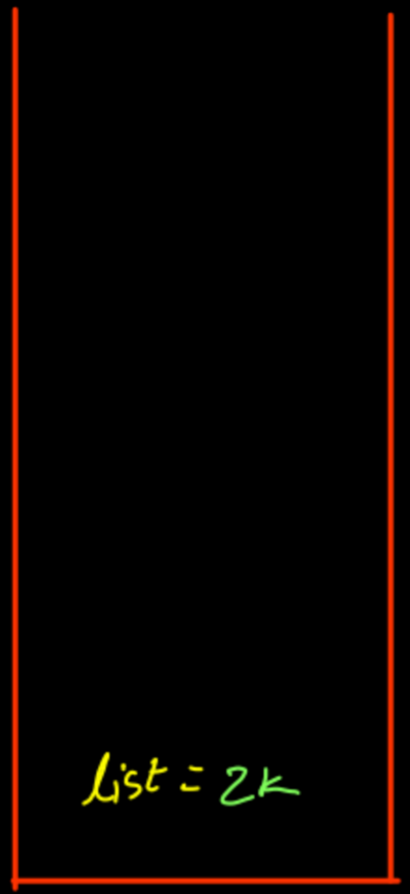
```java
public static class LinkedList {

    Node head;
    Node tail;
    int size;

    public void display() {

    }

    public void addFirst(int data) {

    }

    public void addLast(int data) {

    }

    public int get(int index) {

    }
}
```

psv main() {

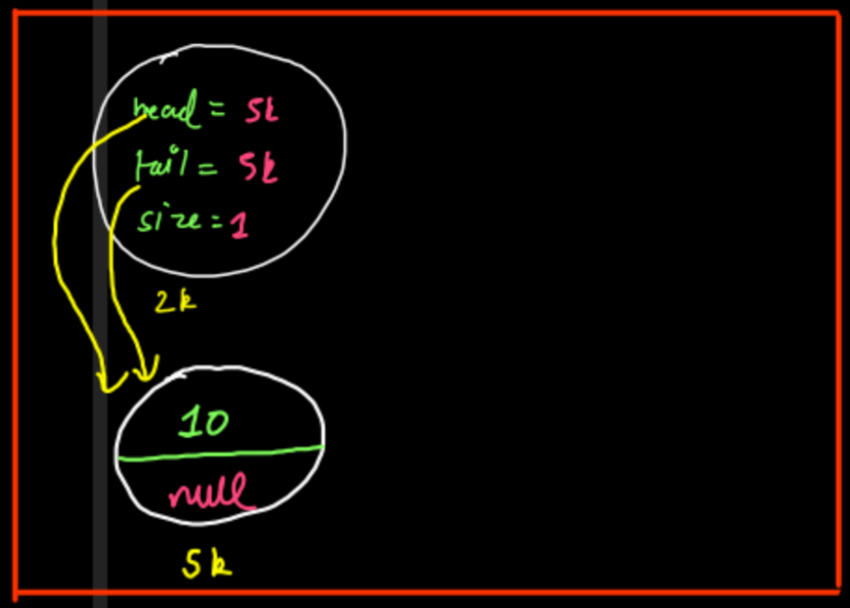
LinkedList list = new LinkedList();

list.addLast(10);

}



Stack



Heap

```java
public static class Node {
    int data;
    Node next; //self referential data member
}
```

```java
public static class LinkedList {

    Node head;
    Node tail;
    int size;

    public void display() {

    }

    public void addFirst(int data) {

    }

    public void addLast(int data) {

    }

    public int get(int index) {

    }

}
```

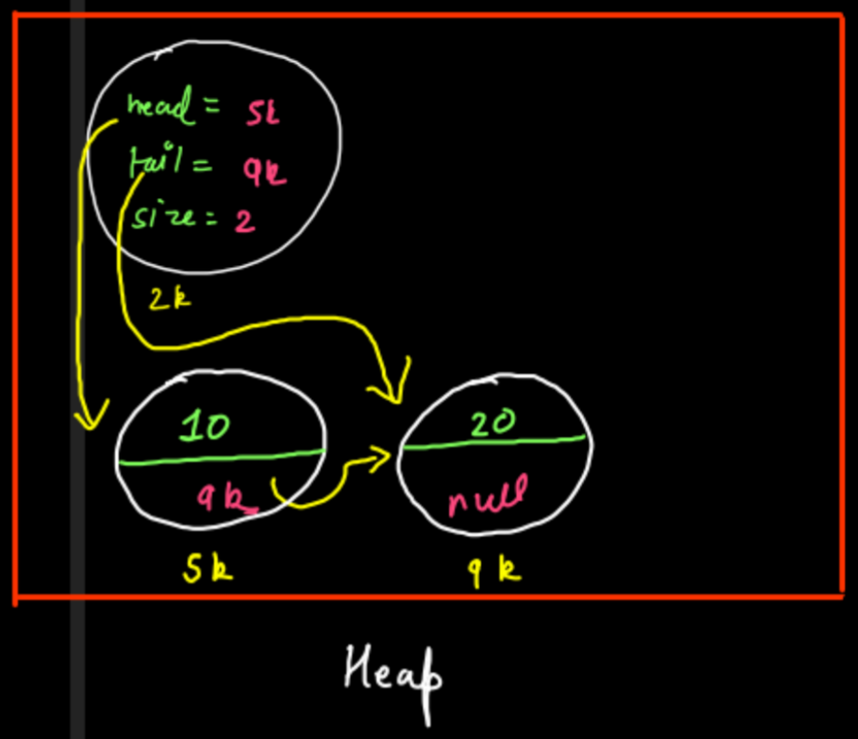
psv main() {

LinkedList list = new LinkedList();

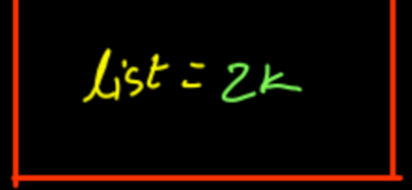
list.addLast(10);

list.addLast(20);

}



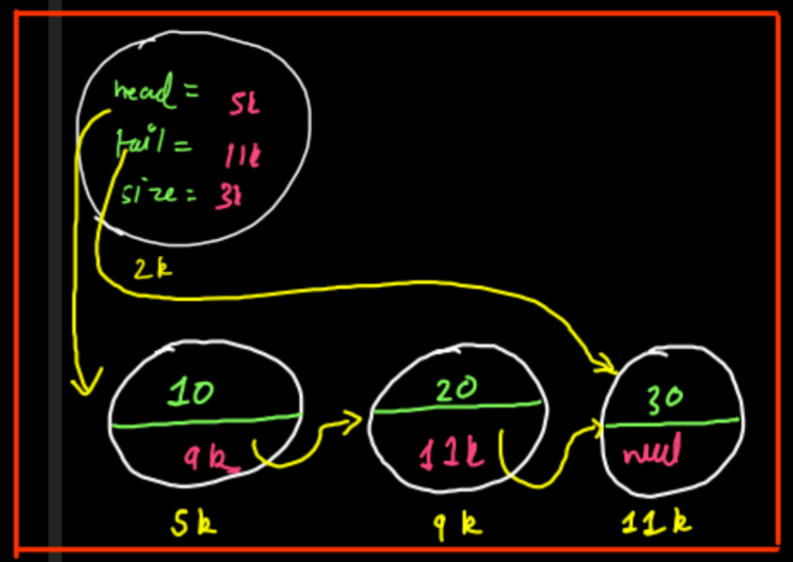
Heap



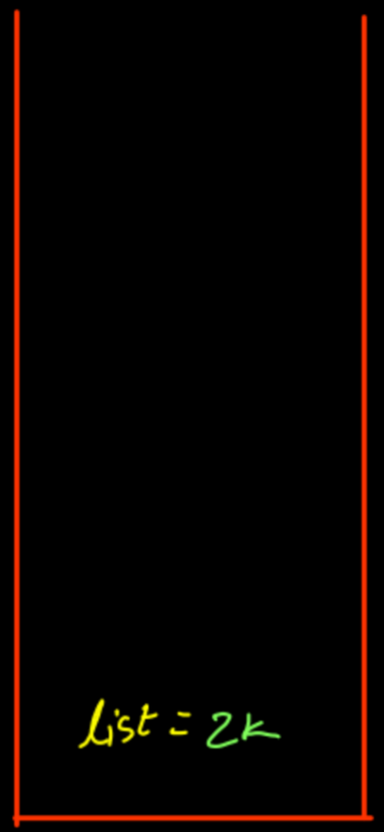
Stack

```java
public static class Node {
    int data;
    Node next; //self referential data member
}
```

```java
public static class LinkedList {

    Node head;
    Node tail;
    int size;

    public void display() {

    }

    public void addFirst(int data) {

    }

    public void addLast(int data) {

    }

    public int get(int index) {

    }

}
```



Heap



Stack

psv main() {

LinkedList list = new LinkedList();

list.addLast(10);

list.addLast(20);

list.addLast(30);

}

** Size of reference is unknown in LL.

LL is taking lot of extra space

Random Access is not allowed in LL

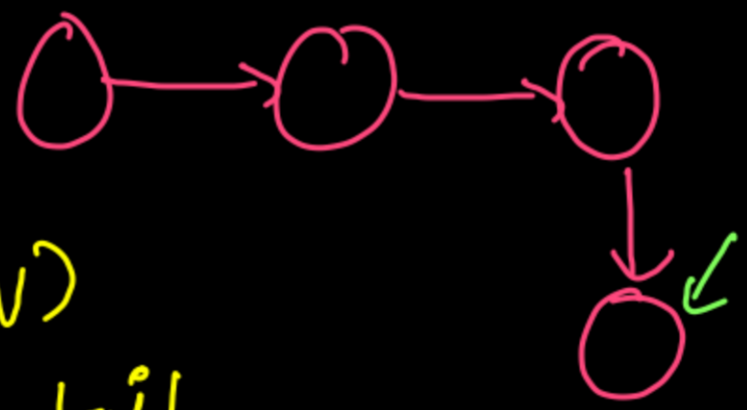
Add Last Code

```
void addLast(int val) {
  // Write your code here
  Node temp = new Node();
  temp.data = val;

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

  size++;
}
```

TC = O(1) → using tail

addLast is a constant operation

#If tail is not given

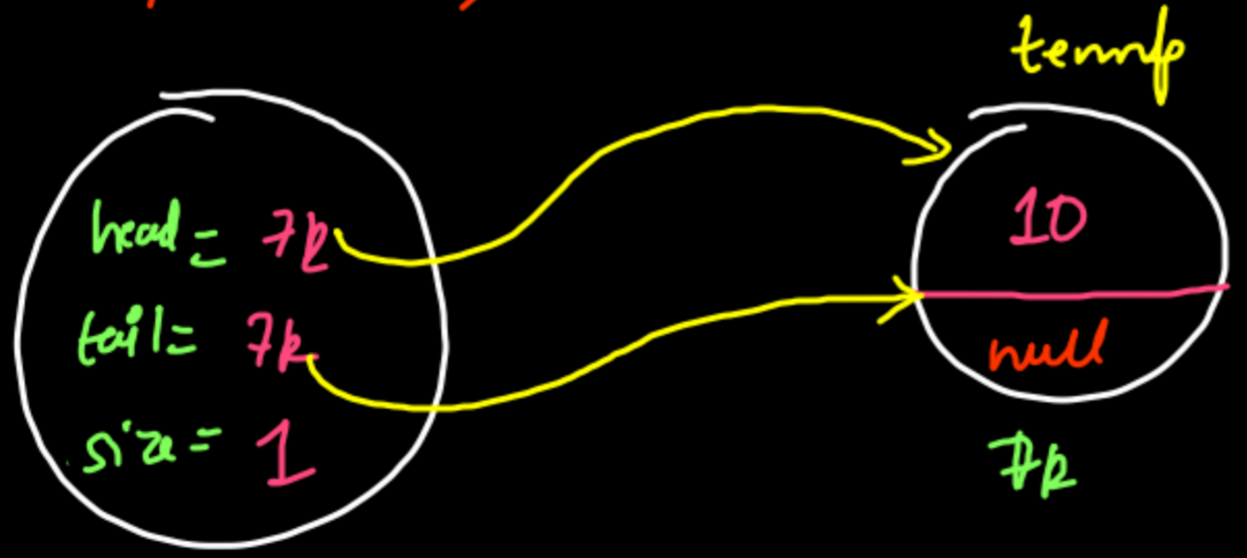


TC = O(N)
without tail

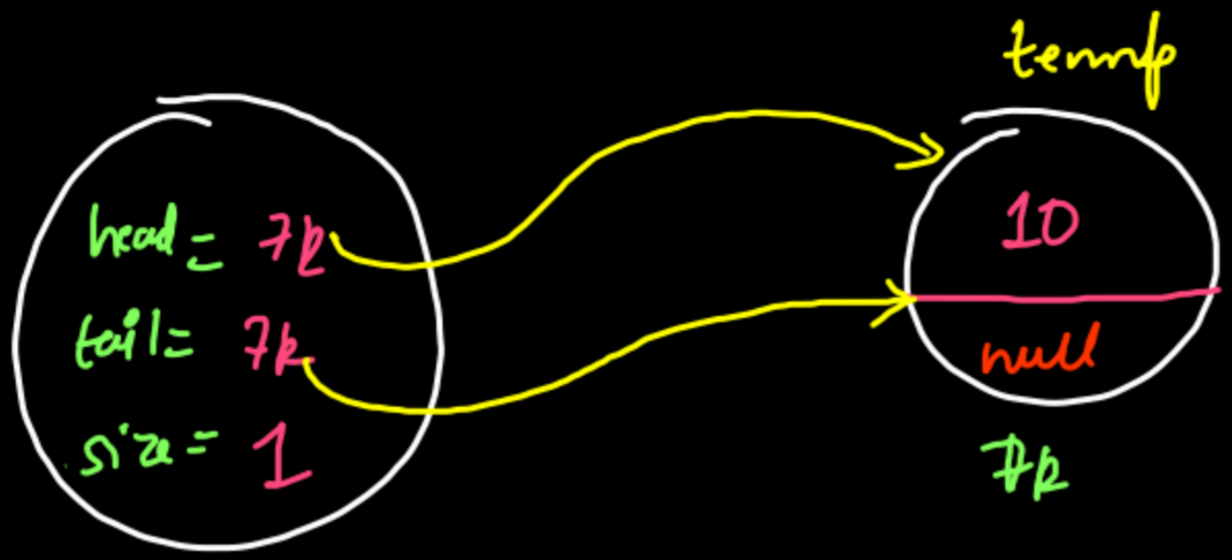
Add first

```
LinkedList list = new LinkedList();
```

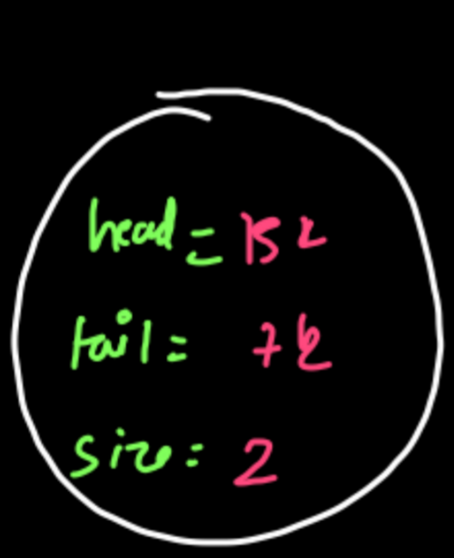
add first(10);

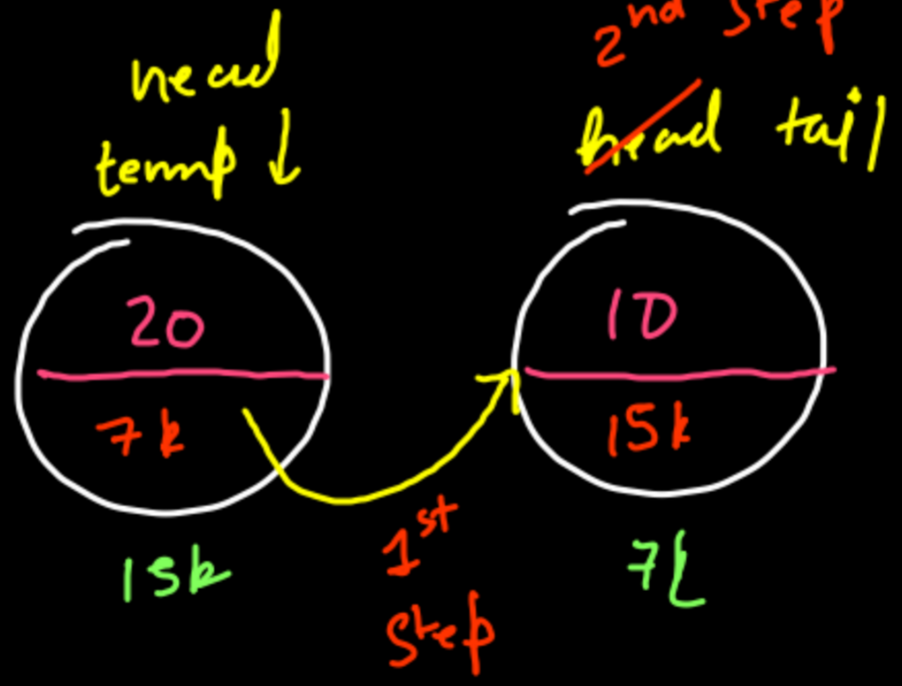



```
if (size == 0)
{ head = temp;
  tail = temp;
}
```



addfirst(20);





if size != 0

1 temp.next = head;

2 head = temp;

Add first Code

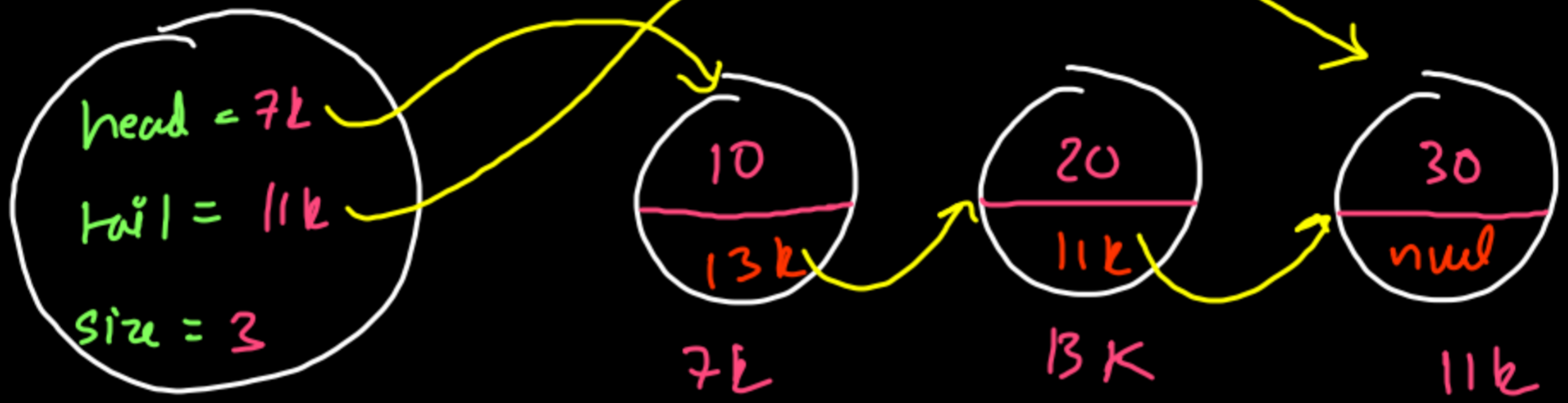
```java
public void addFirst(int val) {
  // write your code here

  Node temp = new Node();
  temp.data = val;

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

  size++;
 }
}
```

TC = O(1)

⇓

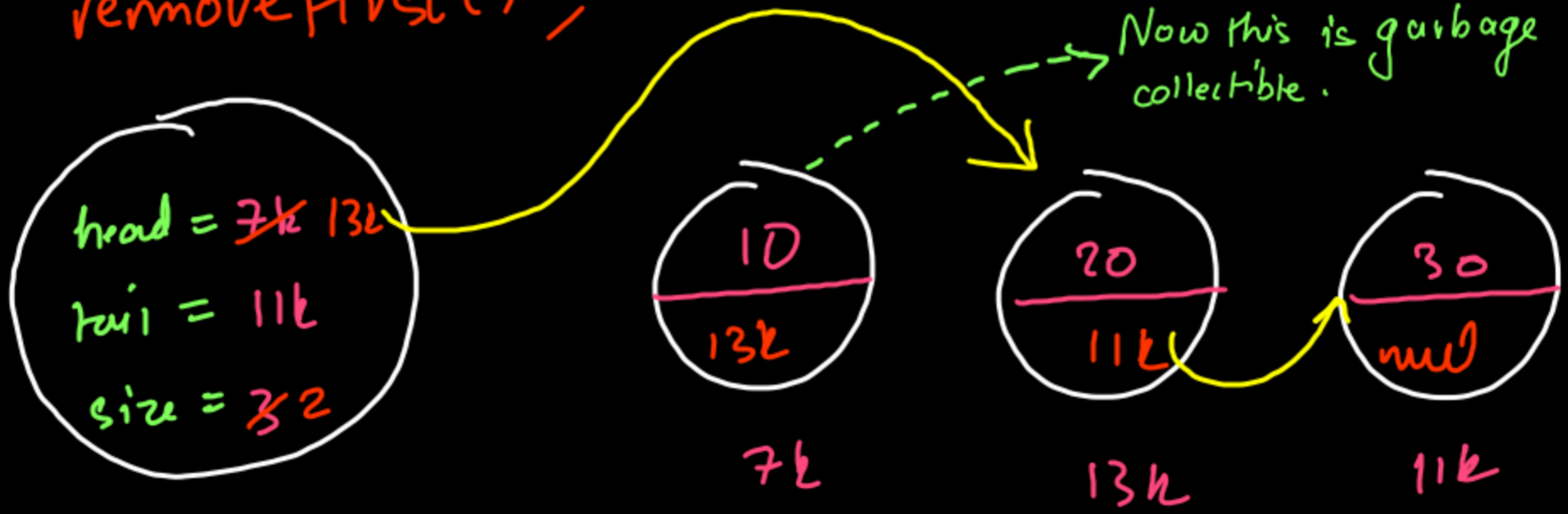
always constant because head will always be there.

Remove first



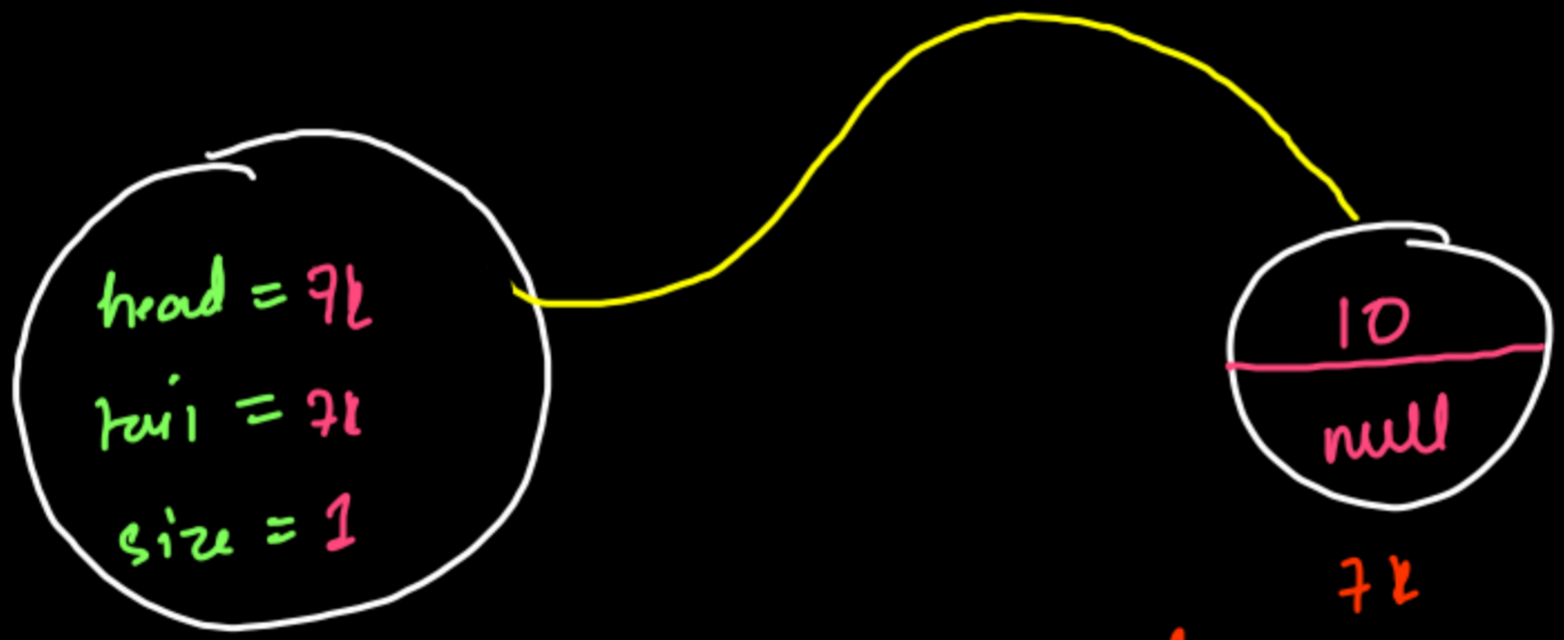
removeFirst();

if (size>1) head = head.next;



if size is 1 then deleting node will empty the LL.

When LL is empty, head = tail = null



head = tail = null

Removefirst Code

```
public void removeFirst(){
  // write your code here
  if(size == 0) {
      System.out.println("List is empty");
      return;
  }

  if(size == 1) {
      head = tail = null;
  } else {
      head = head.next;
  }

  size--;
}
```

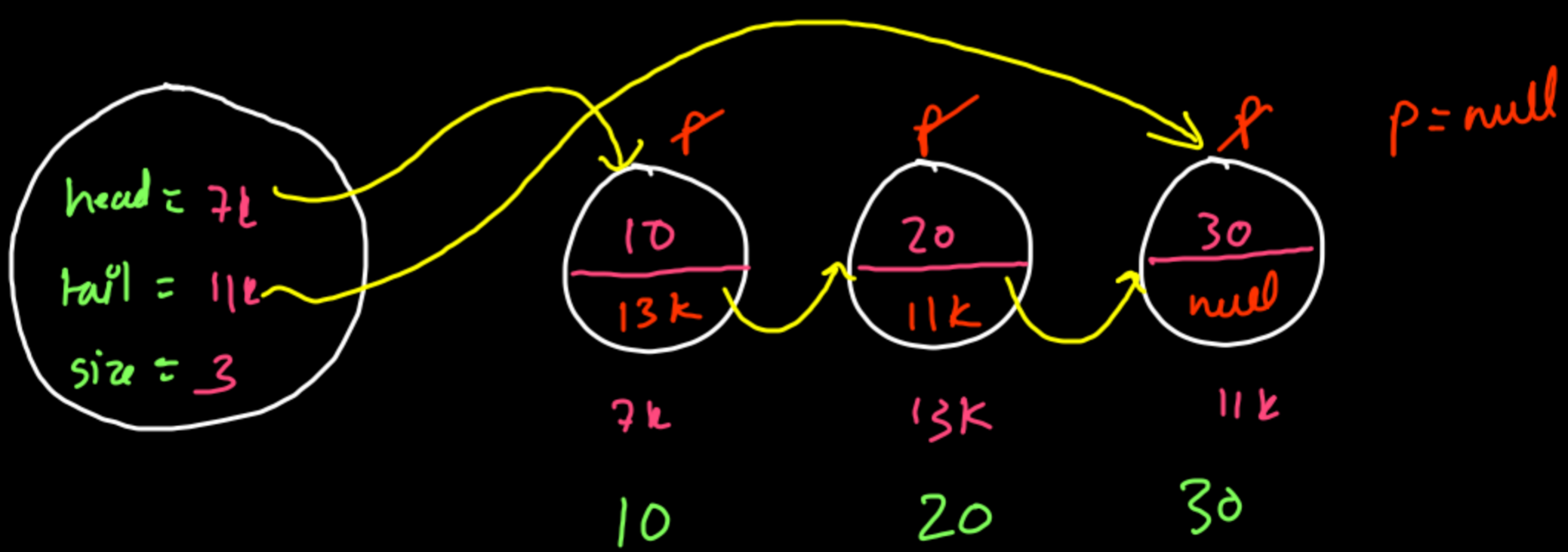
Display List

↗ Using Size

```java
public void display(){
  // write code here

  if(size == 0) return;

  Node p = head;

  for(int i=1;i<=size;i++) {
      System.out.print(p.data + " ");
      p = p.next;
  }
  System.out.println();
}
```

↗ Without using size.

```java
public void display(){
  // write code here

  if(size == 0) return;

  Node curr = head;

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



Get Value in LL

GetFirst()

return head.data;
else return -1;
} In case of invalid index print "Invalid arguments"

If list is empty print "List is empty"

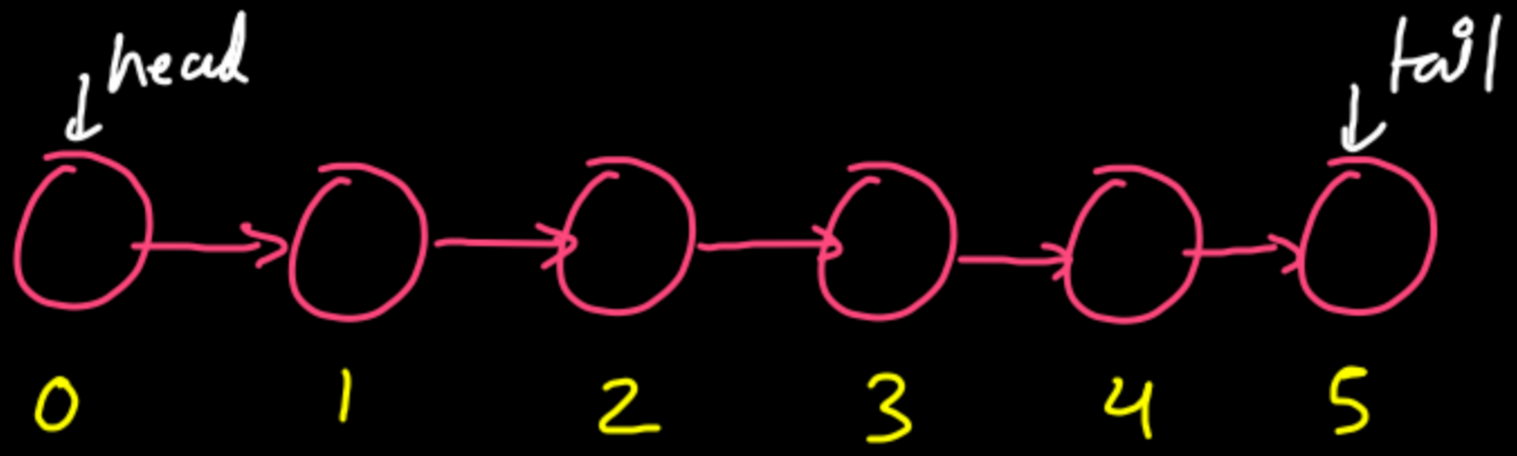
GetLast()

return tail.data;
else return -1;
} In case of invalid index print "Invalid arguments"

If list is empty print "List is empty"

GetAt(Index)

Indexing is 0 based



Apply a Loop now starting from 0 to idx-1
& return curr.data.

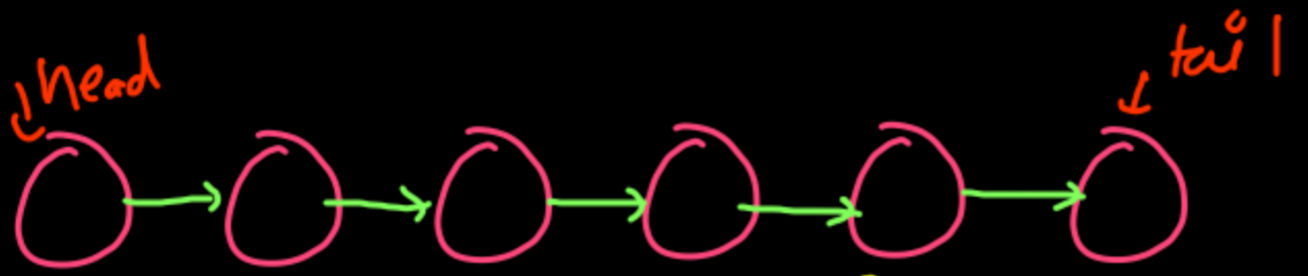
All Get Codes

```java
public int getFirst(){
  // write your code here
  if(size == 0) {
      System.out.println("List is empty");
      return -1;
  } else {
      return head.data;
  }
}
```

```java
public int getLast(){
  // write your code here
  if(size == 0) {
      System.out.println("List is empty");
      return -1;
  } else {
      return tail.data;
  }
}
```

```java
public int getAt(int idx){
  // write your code here
  if(size == 0) {
      System.out.println("List is empty");
      return -1;
  }

  if(idx < 0 || idx >= size) {
      System.out.println("Invalid arguments");
      return -1;
  }

  if(idx == 0) return getFirst();

  if(idx == size - 1) return getLast();

  Node curr = head;

  for(int i=0;i<idx;i++) {
    curr = curr.next;
  }

  return curr.data;
 }
}
```

Removelast



we need previous of tail to delete tail.

Hence, even if we have tail, we have to traverse the LL to delete the lastNode.

Hence, the Time Complexity will always be O(N).

① Go till prev of tail.

```
Node prevTail = get(size-2);
prevTail.next = null;
tail = prevTail;
```

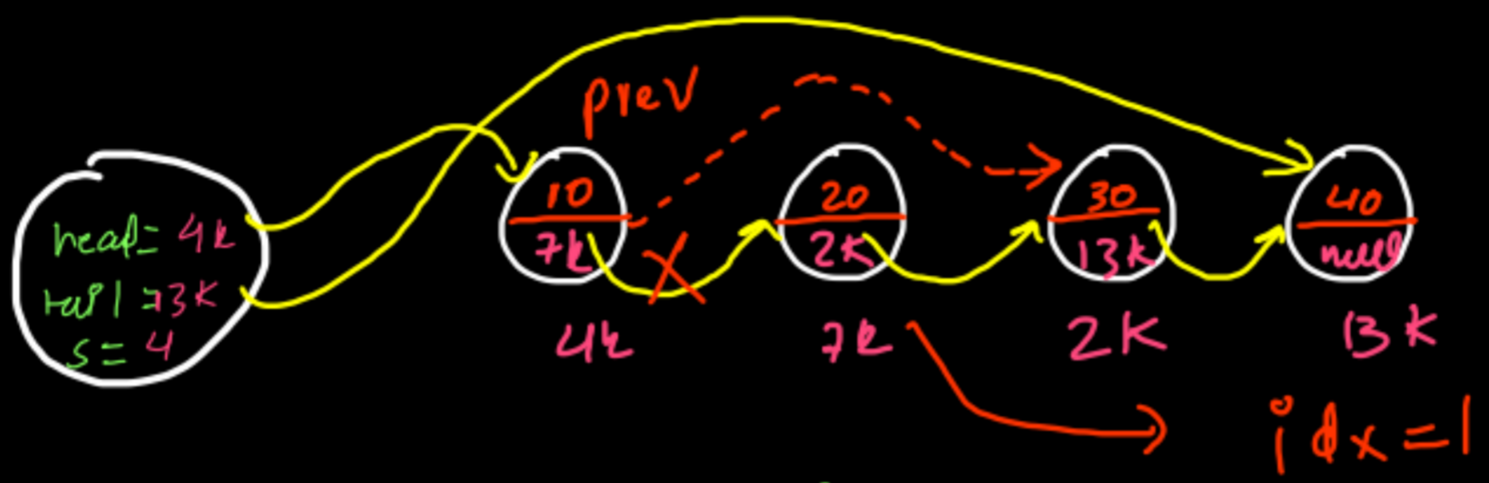
} size > 0

```
if (size == 1)
    head = tail = null;
```

Remove Last Code

```java
public void removeLast(){
  // write your code here
  if(size == 0) {
      System.out.println("List is empty");
      size--;
      return;
  }

  if(size == 1) {
      head = tail = null;
      size--;
      return;
  }

  Node curr = head;
  for(int i=0;i<size-2;i++) {
      curr = curr.next;
  }

  curr.next = null;
  tail = curr;

  size--;
 }
}
```

Remove At



1) idx == 0 → removeFirst();

2) idx == size - 1 → removeLast();

3) removeAt idx → ① Get $(idx-1)^{th}$ node

$0 < idx < size - 1$ ② prev.next = prev.next.next

③ size--;

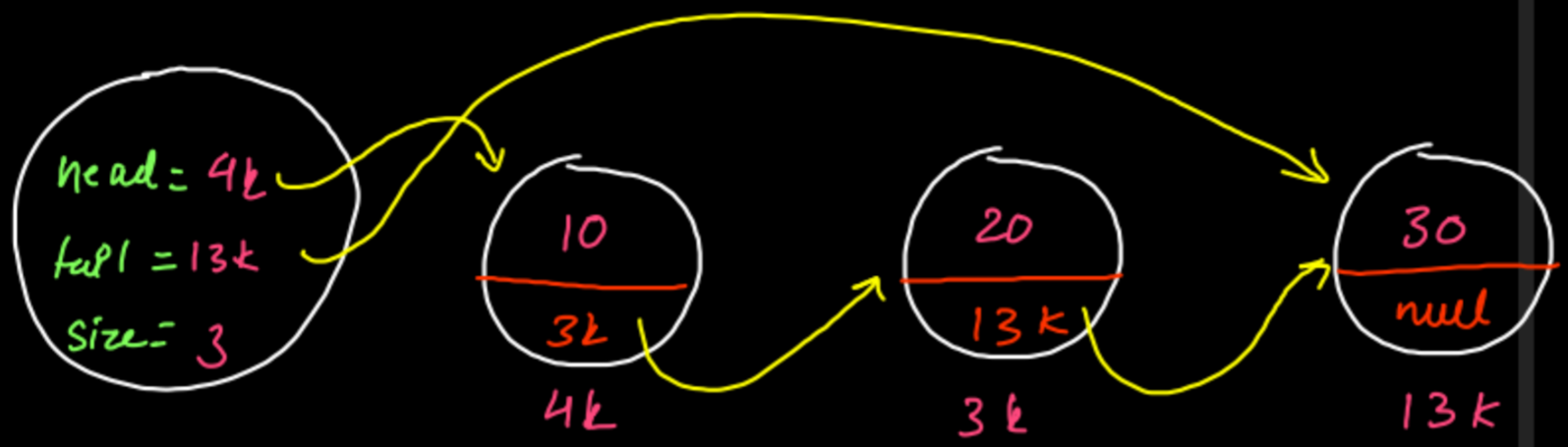
idx < 0 || idx >= size } invalid args

size == 0 } LL is empty

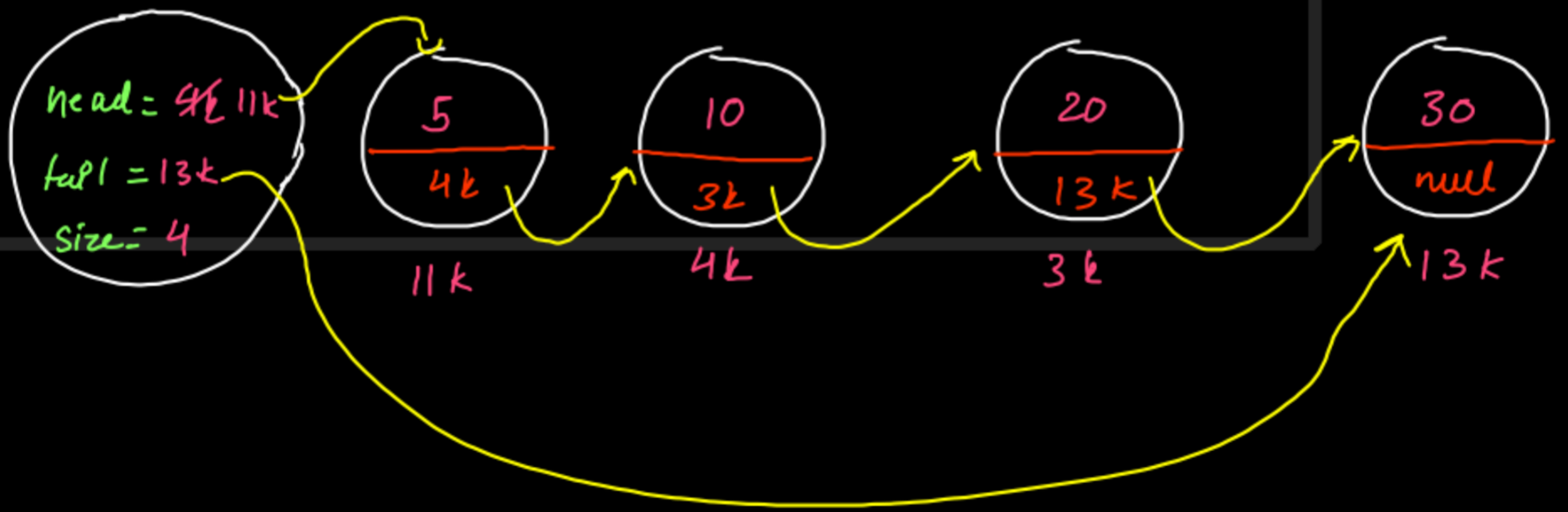
Remove At Code

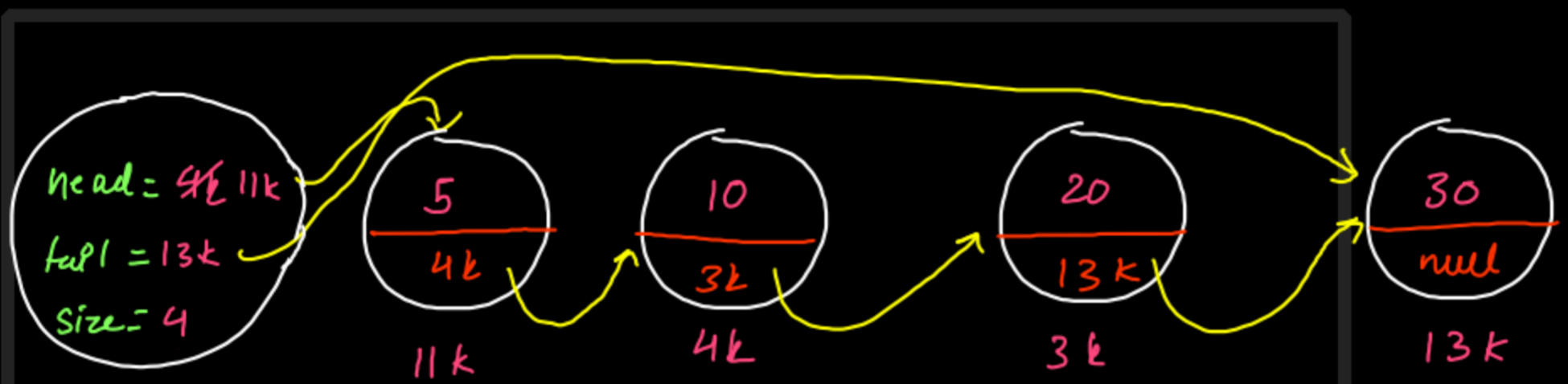
```java
public void removeAt(int index) {
  // write your code here
   if(index < 0 || index >= size) {
        System.out.println("Invalid arguments");
        return;
    }

    if(size == 0) {
        System.out.println("List is empty");
        return;
    }

    if(index == 0) {
        removeFirst();
        return;
    }

    if(index == size - 1) {
        removeLast();
        return;
    }

    Node curr = head;
    for(int i=1;i<index;i++) {
        curr = curr.next;
    }

    curr.next = curr.next.next;
    size--;
}
```

Add At

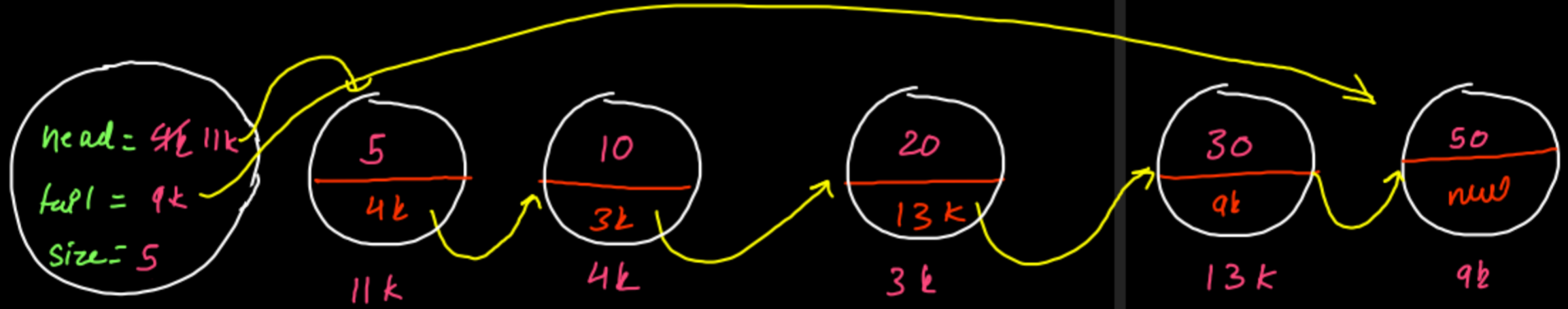


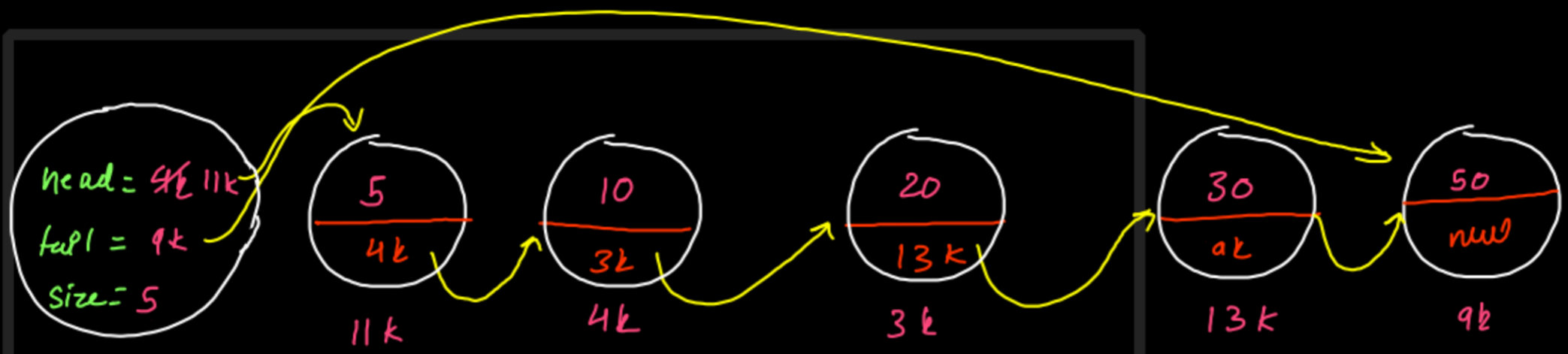
addAt(0,5) -> addFirst();



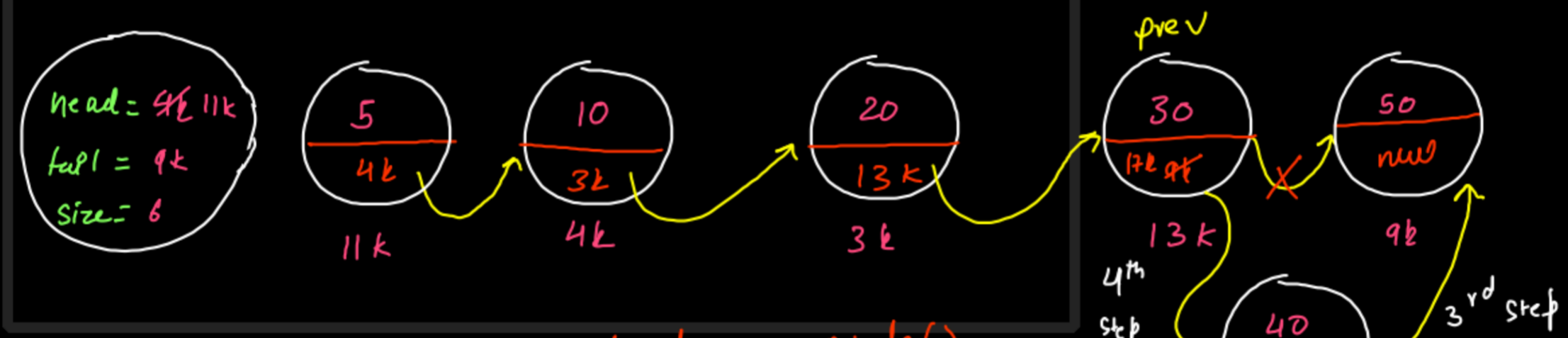


addAt (4, 50) ⇒ addLast ()





addAt(4,40); ⇒ addAt(size-1) is not addAtTail




```
1 temp = new Node();
2 temp.data = val;
3 temp.next = prev.next;
4 prev.next = temp;
```

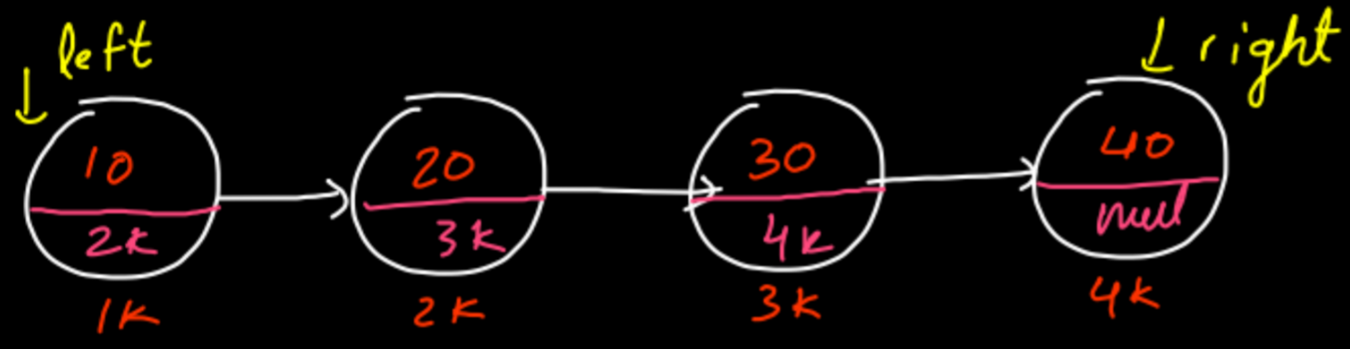
Add At Code

```java
public void addAt(int idx, int val){
  // write your code here

  if(idx < 0 || idx > size) {
      System.out.println("Invalid arguments");
      return;
  }

  if(idx == 0) {
      addFirst(val);
      return;
  }

  if(idx == size) {
      addLast(val);
      return;
  }

  Node prev = head;

  for(int i=1;i<idx;i++) {
      prev = prev.next;
  }

  Node temp = new Node();
  temp.data = val;
  temp.next = prev.next;
  prev.next = temp;
  size++;
 }
}
```

Reverse a Linked List

1. Data Iterative
2. Pointer Iterative
3. Data Recursive
4. Pointer Recursive

Reverse a Linked List Data Iterative



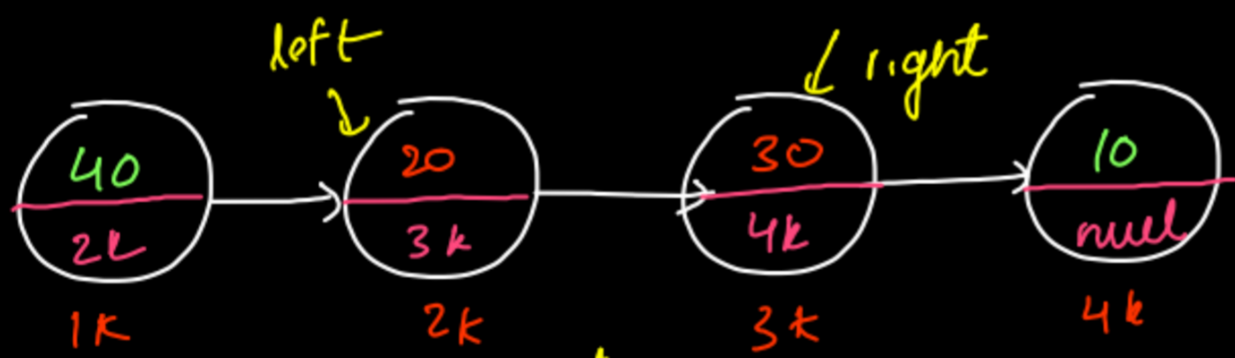
getAt (left) } O(N)
getAt (right) }

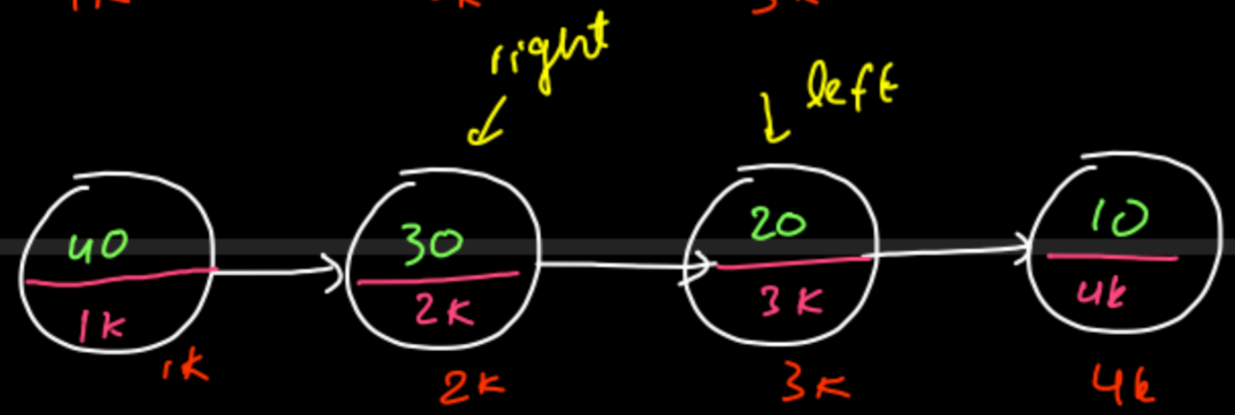
do this for N times

$\frac{N}{2}$

$\frac{N}{2} * O(N)$

$\Rightarrow TC = O(N^2)$



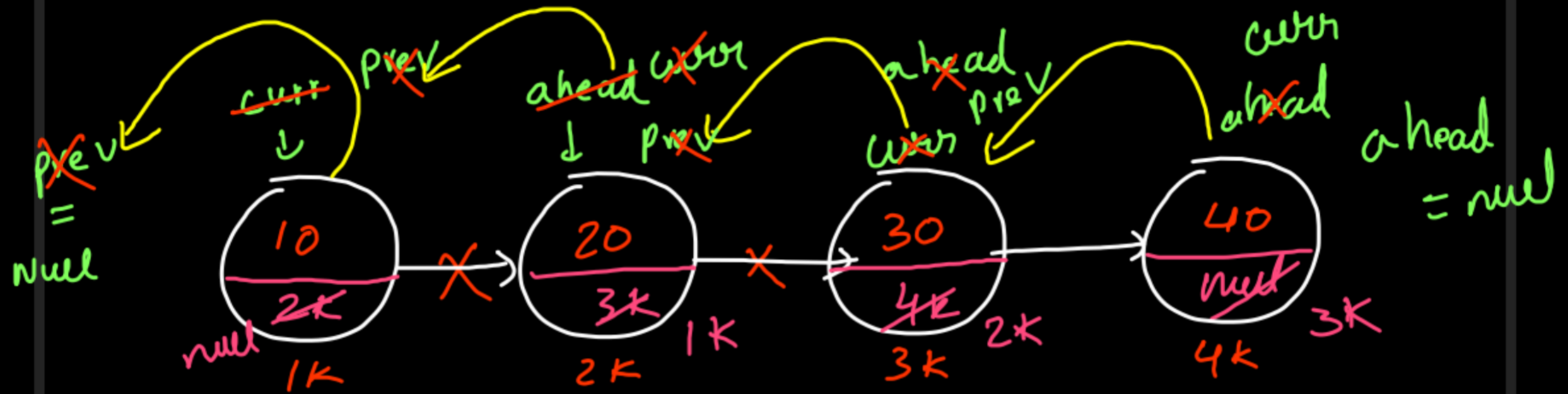


Reverse DI Code

```
public void reverseDI() {
  // write your code here
  int left = 0;
  int right = size - 1;

  while(left <= right) {
      Node leftNode = getNodeAt(left);
      Node rightNode = getNodeAt(right);
      int temp = leftNode.data;
      leftNode.data = rightNode.data;
      rightNode.data = temp;
      left++;
      right--;
    }
  }
}
```

However it is an $O(n^2)$ approach, it is still the only & best possible approach for reversing LL DI way.

Reverse a Linked List Pointer Iterative



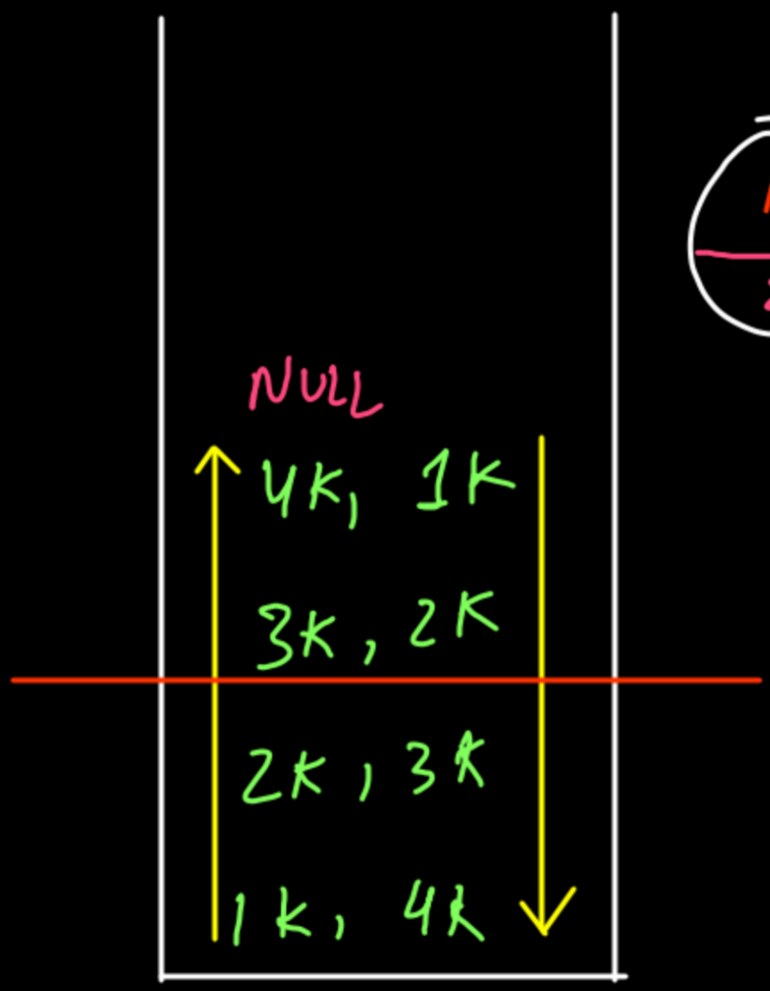
We will need 3 pointers

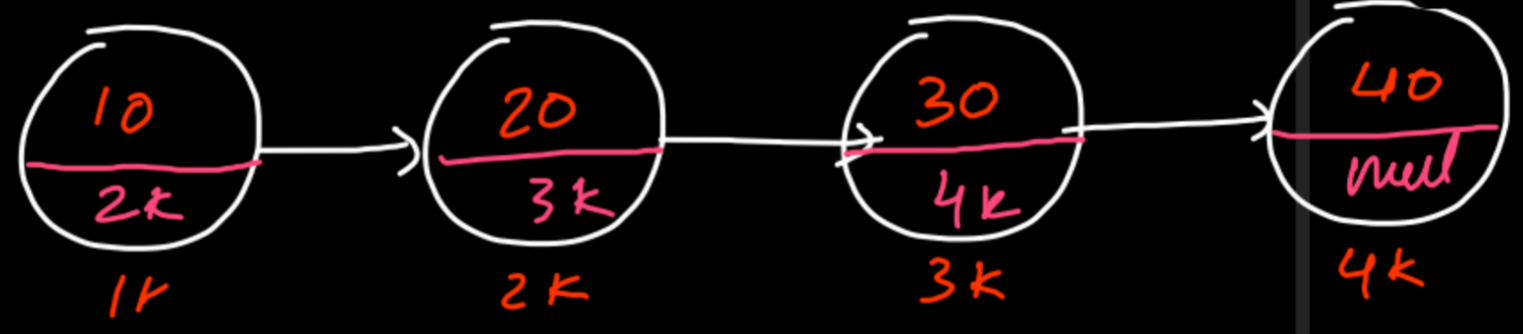
```
curr.next = prev;
prev = curr;
curr = ahead;
```
} while (curr != null)

Reverse a LL Pointer Iterative Code

```
public void reversePI(){
  // write your code here
  if(size == 0 || size == 1) {
      return;
  }
  Node prev = null;
  Node curr = head;

  while(curr != null) {
      Node ahead = curr.next;
      curr.next = prev;
      prev = curr;
      curr = ahead;
  }

//   swap head and tail
  Node temp = head;
  head = tail;
  tail = temp;
}
```

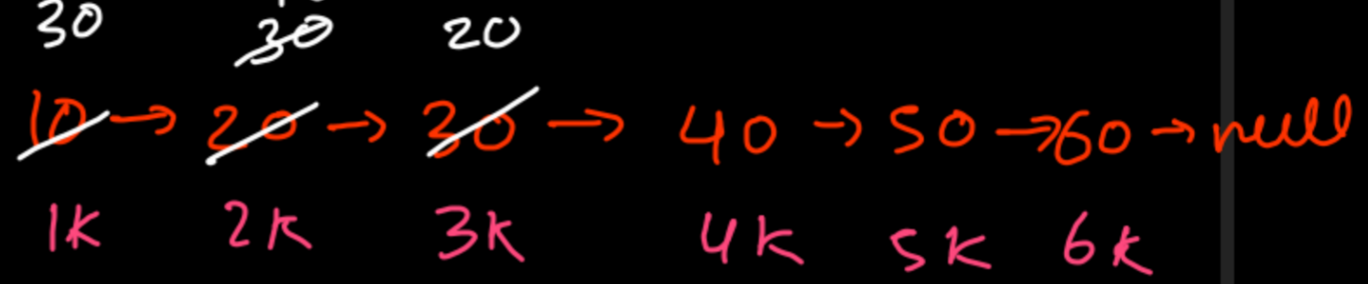
Reverse a LL Data Recursive





Do only 1/2 times because else the LL will remain as it is.

30 10 20



l = null

l = 6k r = 1k c = 5

l = 5k r = 2k c = 4

l = 4k r = 1k c = 3

l = 3k r = 2k c = 2

l = 2k r = 1k c = 1

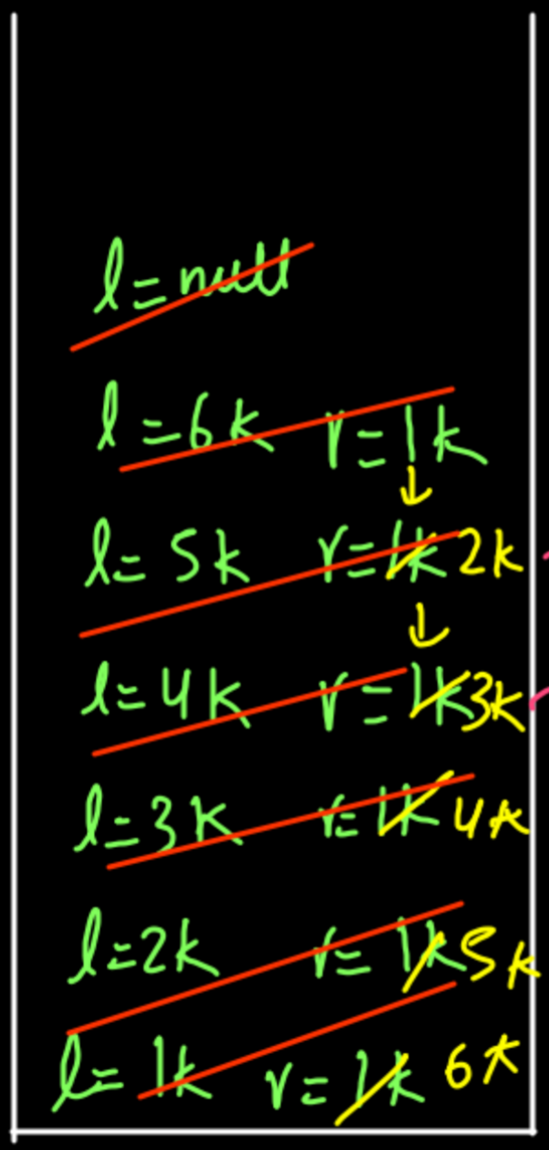
l = 1k r = 1k c = 0

This is wrong as right = right.next. will persist only locally. To persist it in the next call, make right global or return it.

```
static Node right;
public void reverseDR(Node left, int counter){
    if(left == null){
        return;
    }

    reverseDR(left.next, counter + 1);

    if(counter < size/2){
        swap(left, right);
    }

    right = right.next;
}

public void reverseDR() {
    Node left = head;
    right = head;
    reverseDR(left, 0);
}
```

Dry Run for Data Recursive using Return

if (counter > size(2)



r 60 l 10
10 → 20 → 30 → 40 → 50 → 60 → null
1k 2k 3k 4k 5k 6k

r 50 l 20
60 → 20 → 30 → 40 → 50 → 10 → null
1k 2k 3k 4k 5k 6k

r 40 l 30
60 → 50 → 30 → 40 → 20 → 10 → null
1k 2k 3k 4k 5k 6k

60 → 50 → 40 → 30 → 20 → 10 → null
1k 2k 3k 4k 5k 6k