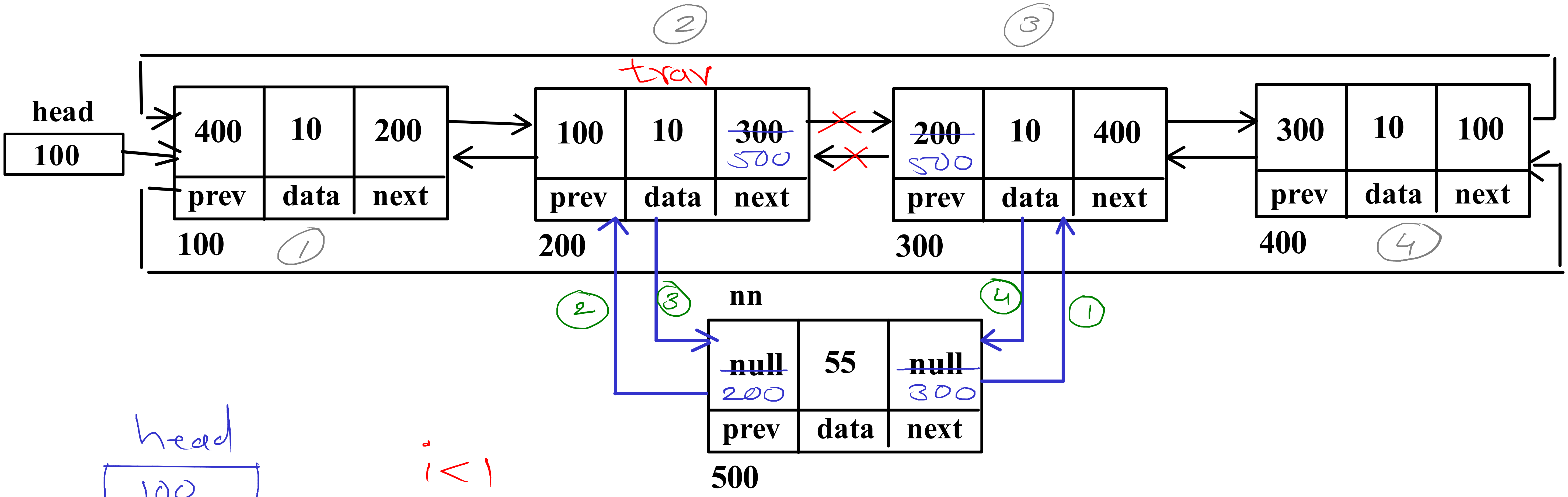
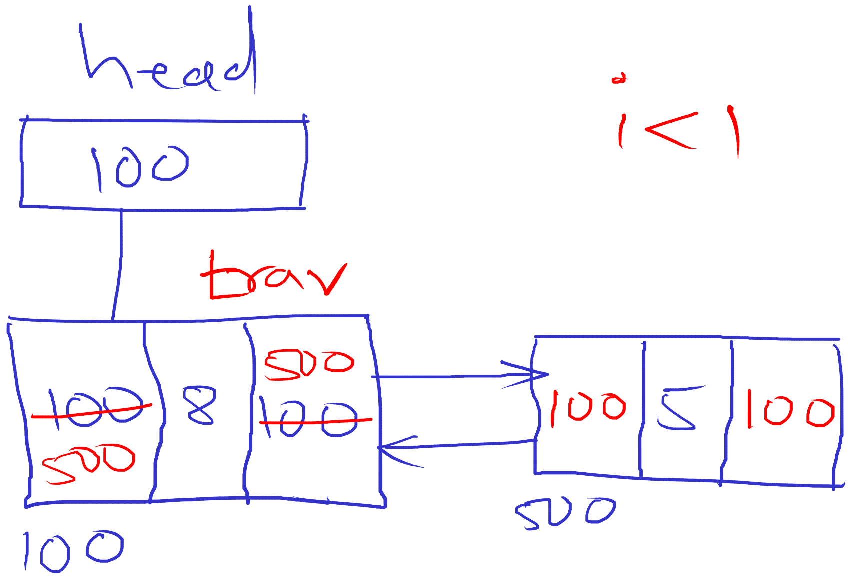
Time Complexity Analysis of Linked List

	SLLL		SCLL		DLLL		DCLL
1. Add First	O(1)	O(1)	O(n)	O(1)	O(1)	O(1)	O(1)
2. Add Last	O(n)	O(1)	O(n)	O(1)	O(n)	O(1)	O(1)
3. Add Position	O(n)	O(n)	O(n)	O(n)	O(n)	O(n)	O(n)
4. Del First	O(1)	O(1)	O(n)	O(1)	O(1)	O(1)	O(1)
5. Del Last	O(n)	O(n)	O(n)	O(n)	O(n)	O(1)	O(1)
6. Del Position	O(n)	O(n)	O(n)	O(n)	O(n)	O(n)	O(n)
7. Display	O(n)	O(n)	O(n)	O(n)	O(n)	O(n)	O(n)

pos = 3



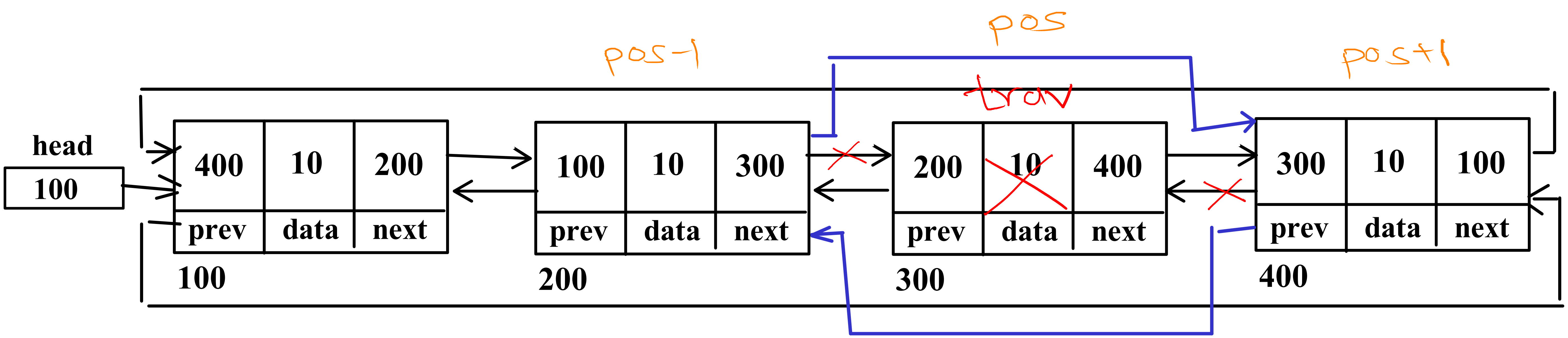


i<1

```
Node trav = head;
for(i=1; i<pos-1; i++)
        trav=trav.next;
```

1. nn.next = trav.next;
2. nn.prev = trav;
3. nn.prev.next = nn
4. nn.next.prev = nn

pos = 3




```
Node trav = head;
for(i=1; i<pos; i++)
    trav = trav.next;

trav.prev.next = trav.next;
trav.next.prev = trav.prev;
```

Linked List Applications

- dynamic data structure - grow / shrink at runtime
- due to this dynamic nature, it is used to implement other data structures
    1. Stack
    2. Queue
    3. Hash Table (Seperate chaining)
    4. Graph (Adjacency list)

- Operating system - job queue, ready queue, waiting queues
  (Doubly circular linked list)

Stack

1. Add First
   Del First

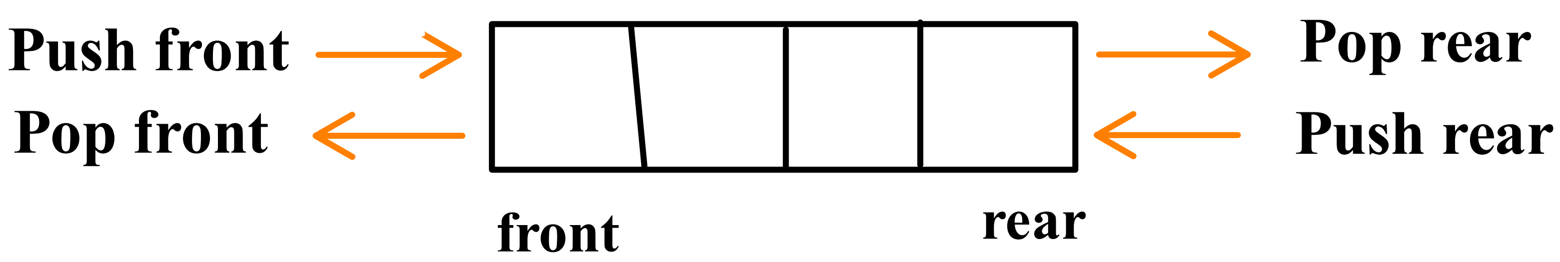
2. Add Last
   Del Last

Queue

1. Add First
   Del Last

2. Add Last
   Del First

Deque (Double Ended Queue)



1. Push front
   Add First
2. Pop front
   Del First

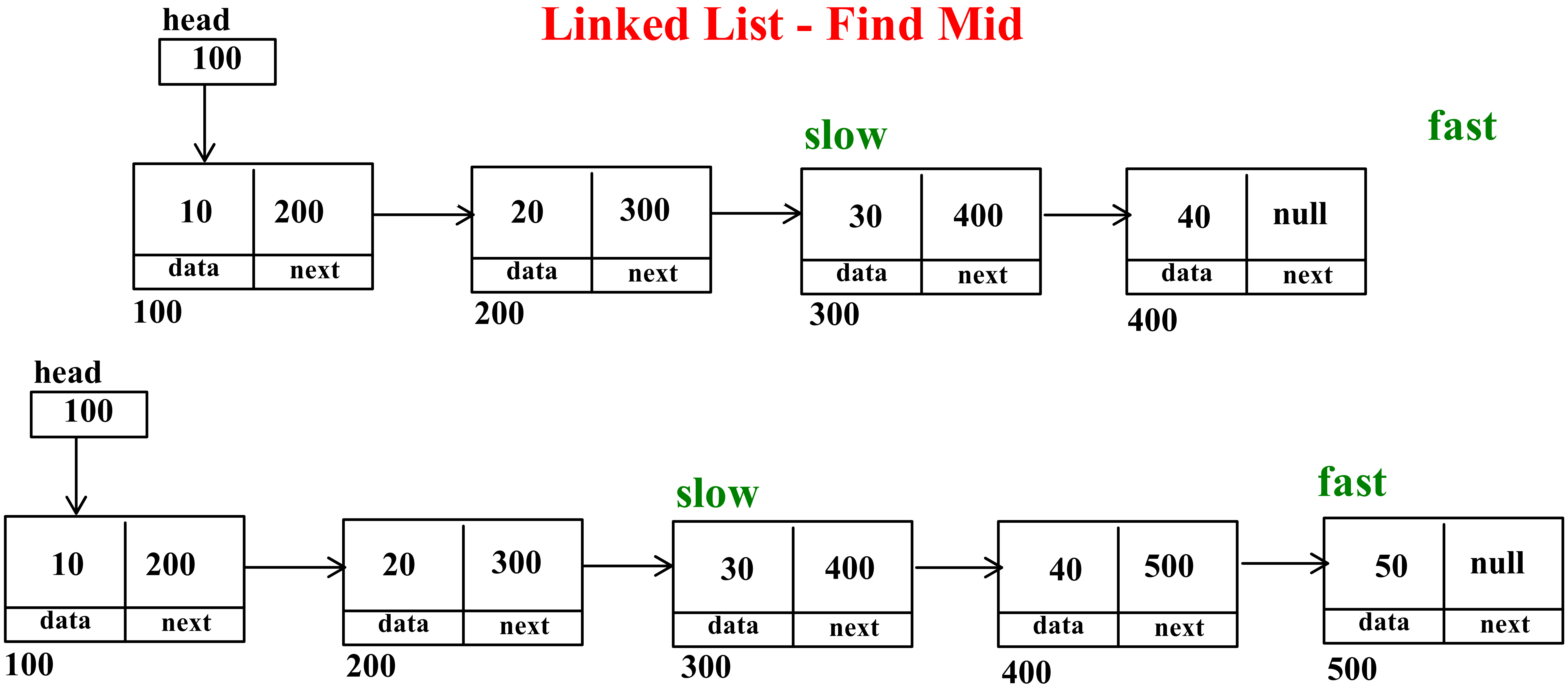
1. Push rear
   Add Last
2. Pop rear
   Del Last

Types :
1. Input restricted Deque
2. Output restricted Deque

Array Vs Linked List

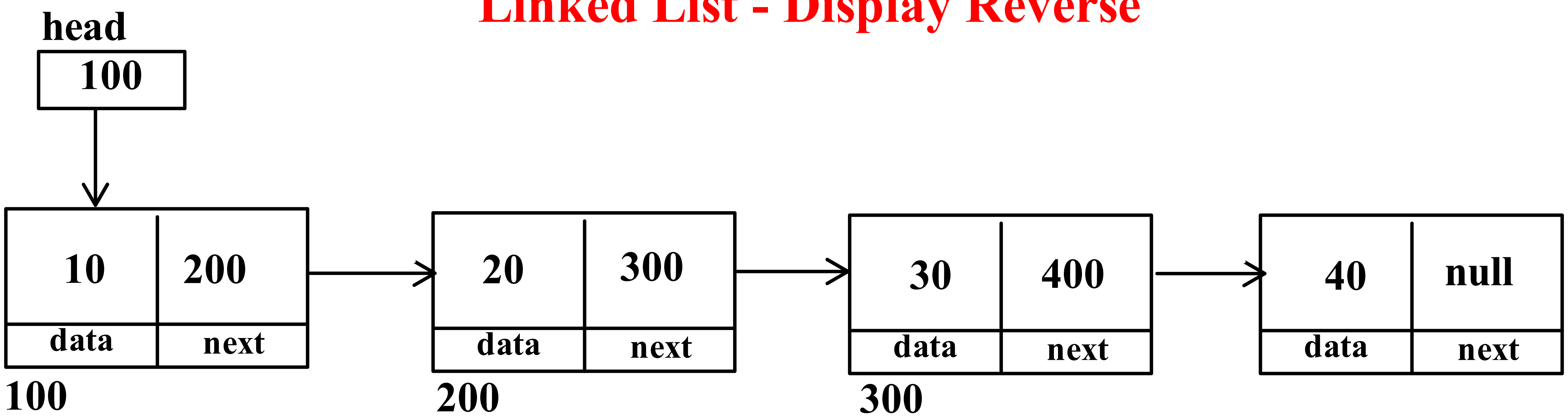
Array	Linked List
1. Array space in memory is contiguous	1. Linked list space in memory is not contiguous
2. Array can not grow or shrink at runtime	2. Linked list can grow or shrink at runtime
3. Random access of elements is allowed	3. Random access of elements is not allowed(sequential)
4. Insert or Delete, needs shifting of array elements	4. Insert or Delete, do not need shifting of nodes
5. Array needs less space	5. Linked lists need more space

Linked List - Find Mid




```
public Node findMid(){
    Node fast = head;
    Node slow = head;
    while(fast != null && fast.next != null){
        fast = fast.next.next;
        slow = slow.next;
    }
    return slow;
}
```

Linked List - Display Reverse




```
void forwardDisplay(Node trav){
        if(trav == null)
                return;
        sysout(trav.data);
        forwardDisplay(trav.next);
}
```

10, 20, 30, 40

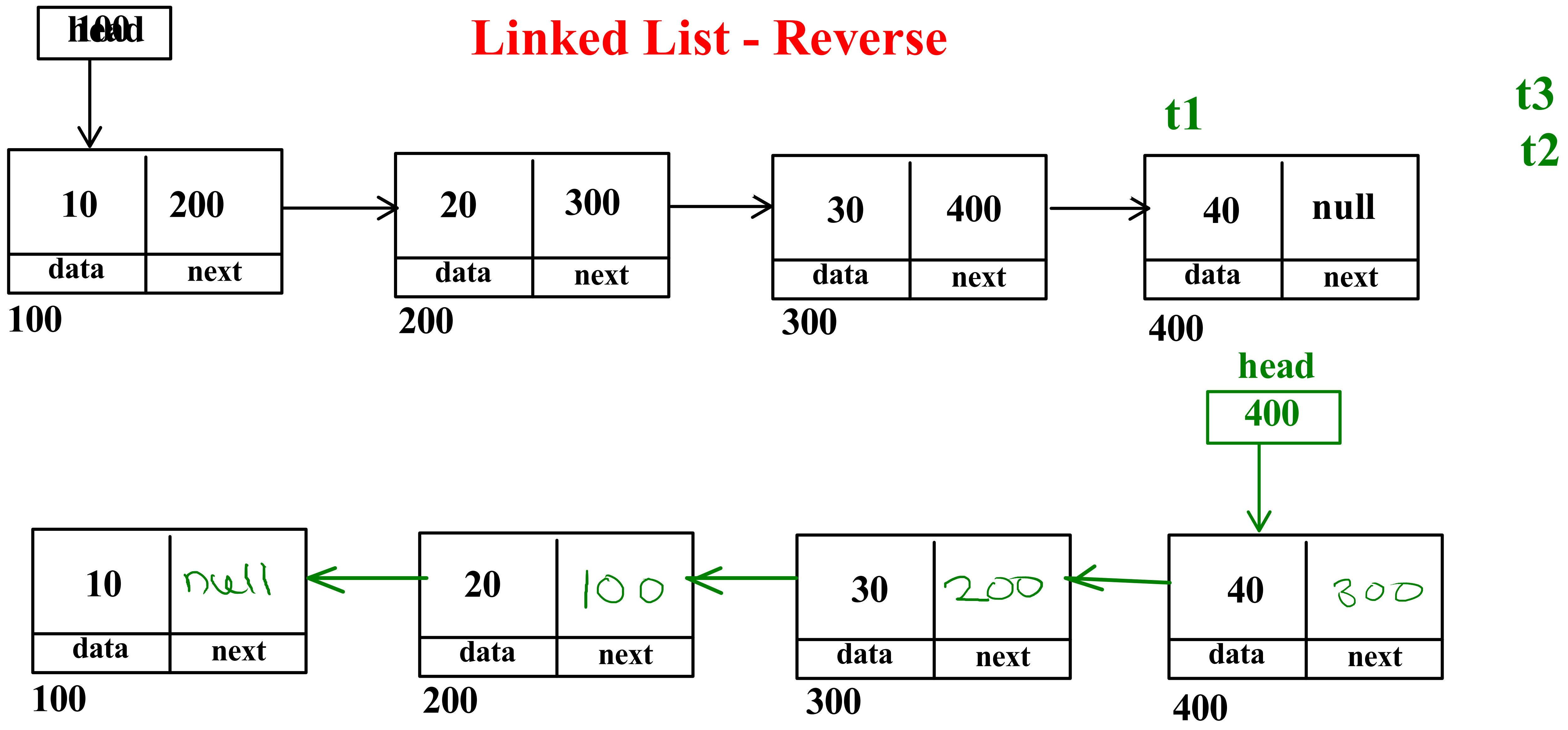
forwardDisplay(100)
forwardDisplay(200)
forwardDisplay(300)
forwardDisplay(400)
forwardDisplay(null)

```
void backwardDisplay(Node trav){
        if(trav == null)
                return;
        backwardDisplay(trav.next);
        sysout(trav.data);
}
```

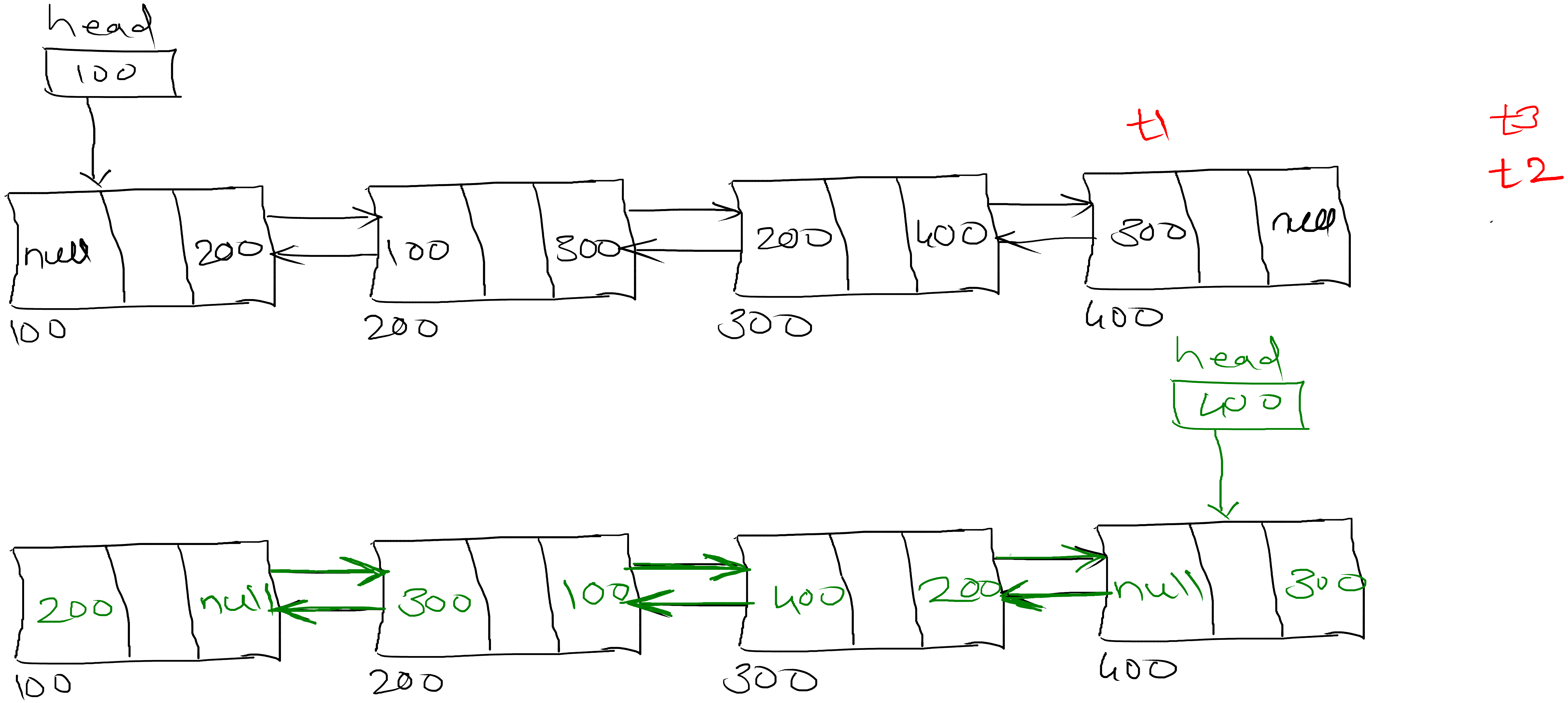
40, 30, 20, 10

backwardDisplay(100)
backwardDisplay(200)
backwardDisplay(300)
backwardDisplay(400)
backwardDisplay(null)

Linked List - Reverse

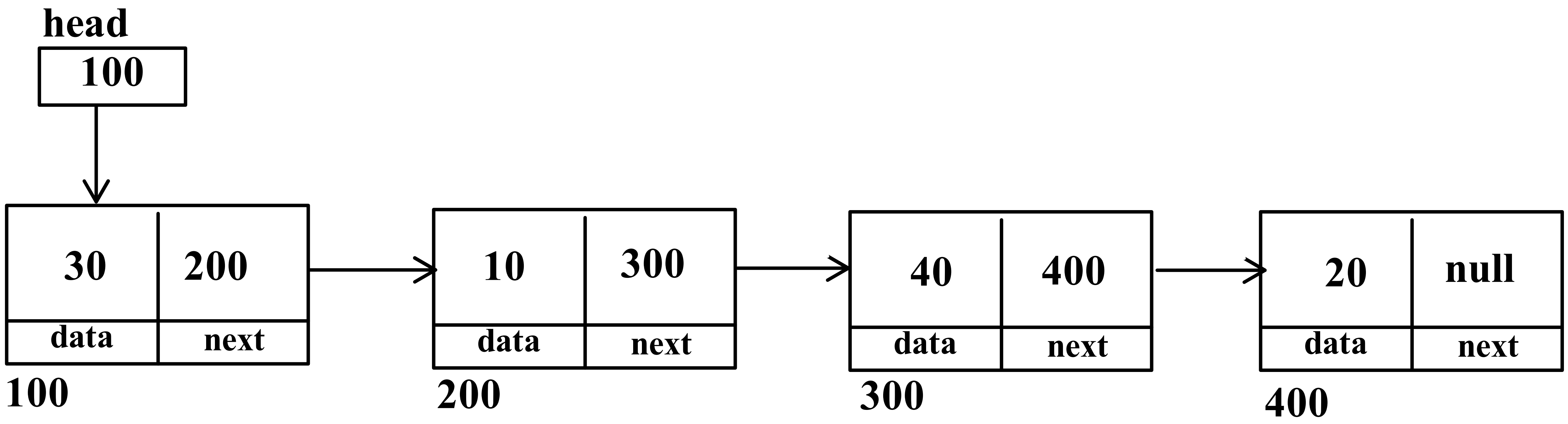



```
public void reverseList(){
        Node t1 = head; Node t2 = head.next; Node t3;
        while(t2 != null){
                t3 = t2.next;
                t2.next = t1;
                t1 = t2;
                t2 = t3;
        }
        head.next = null;  head = t1;
}
```




```
Node t1 = head; Node t3;
Node t2 = head.next;
while(t2 != null) {
    t3 = t2.next
    t1.prev = t2
    t2.next = t1
    t1 = t2; t2 = t3;
}
head.next = null; t1.prev = null;
head = t1;
```

Linked List - sort




```
for(Node i=head; i.next != null ; i=i.next)
{
    for(Node j=i.next; j!=null; j=j.next)
    {
        if(i.data > j.data) {
            int temp=i.data ;
            i.data=j.data;
            j.data=temp;
        }
    }
}
```

ADD Node into BST

//1. create node with given data
//2. if BStree is empty
    // add newnode into root
//3. if BSTree is not empty
    //3.1 create one trav pointer and start from root
    //3.2 if data is less than current node data
        //3.2.1 if current node do not have left child
            // add newnode into left of current node
        //3.2.2 if current node have left child
            // go on left child
    //3.3 if data is greater than current node data
        //3.3.1 if current node do not have right child
            // add newnode into right of current node
        //3.3.2 if current node has right child
            // go on right child
    //3.4 repeat step 3.2 and 3.3 till node is not added into BST

ADD Node into BST