Singly Linear Linked List - Reverse Display

head
↓
10 → 20 → 30 → 40

Non-tail Recursion

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

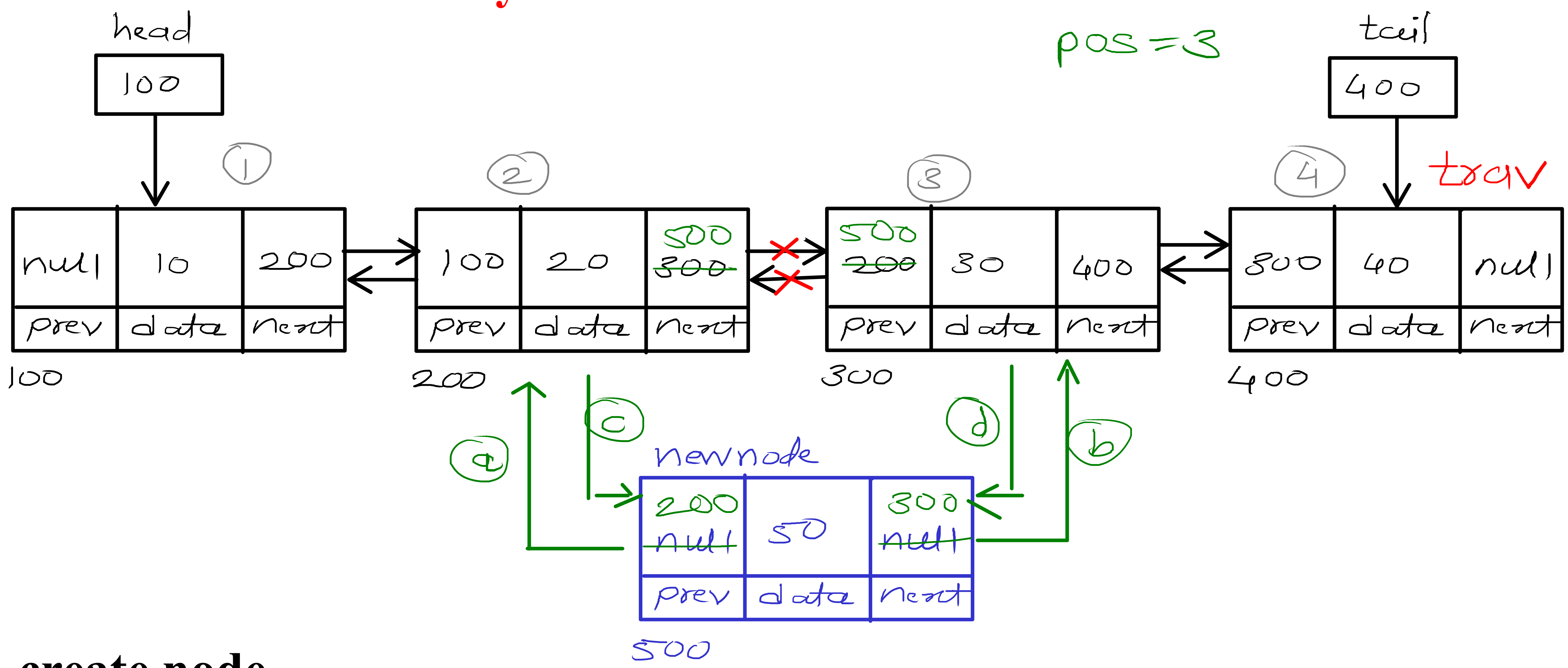
```
rDisplay(&10)
rDisplay(&20)
rDisplay(&30)
rDisplay(&40)
rDisplay(null)
```

Tail Recursion

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

```
fDisplay(&10)
fDisplay(&20)
fDisplay(&30)
fDisplay(&40)
fDisplay(null)
```

Doubly Linear Linked List - Add Position



//1. create node

//2. if list is empty

// add newnode into head and tail

//3. if list is not empty

// traverse till pos-1 node

//a. add pos-1 node into prev of newnode

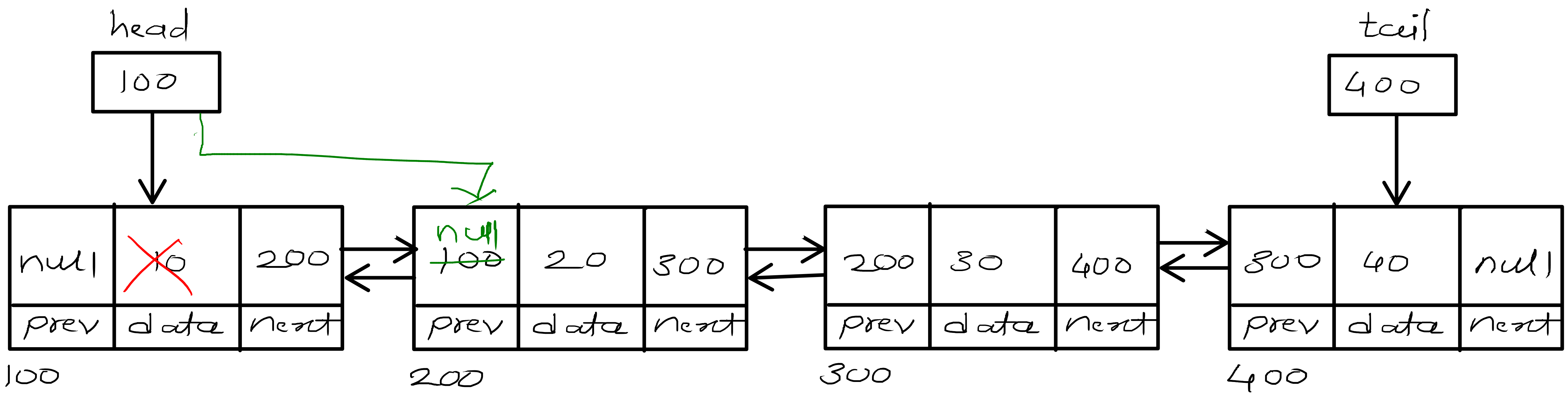
//b. add pos node into next of newnode

//c. add newnode into next of pos-1 node

//d. add newnode into prev of pos node

Time Complexity : O(n)

Doubly Linear Linked List - Delete First



//1. if list is empty

return;

//2. if list has single node

head = tail = null;

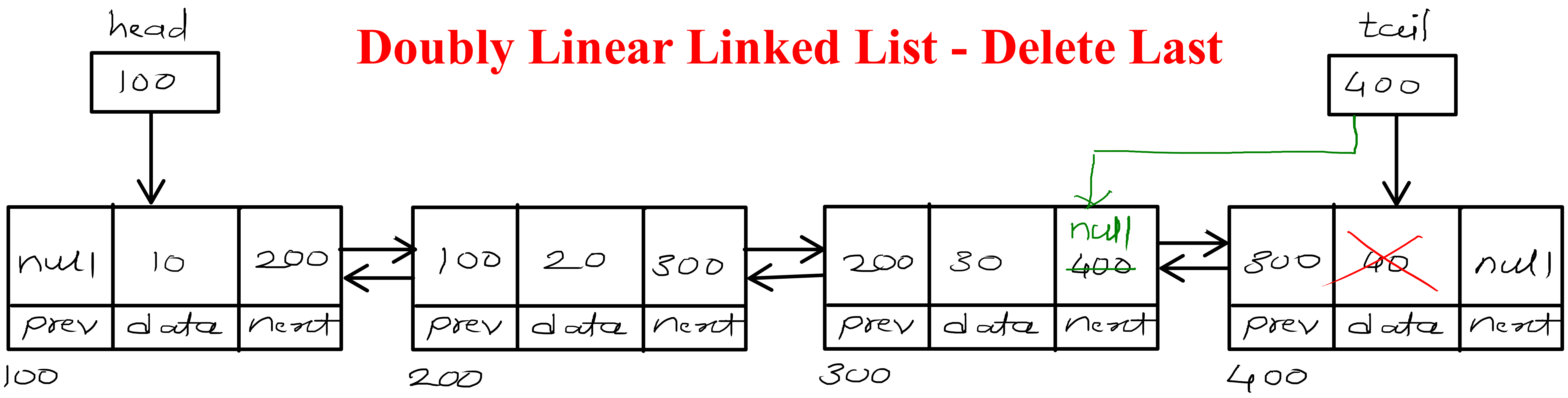
//3. if list has multiple nodes

a. move head on second node

//b. add null into prev of second node

Time Complexity : O(1)

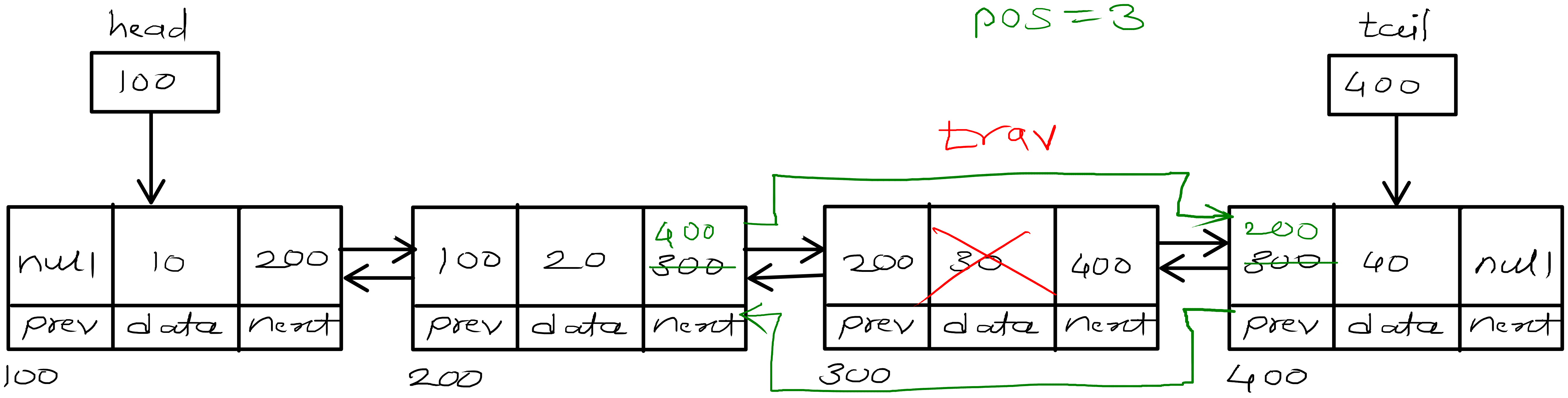
Doubly Linear Linked List - Delete Last



//1. if list is empty
return;
//2. if list has single node
head = tail = null;
//3. if list has multiple nodes
//a. move tail on second last node
//b. add null into next of second last node

Time Complexity : O(1)

Doubly Linear Linked List - Delete Position



//1. if list is empty

return;

//2. if list has single node

head = tail = null;

//3. if list has multiple nodes

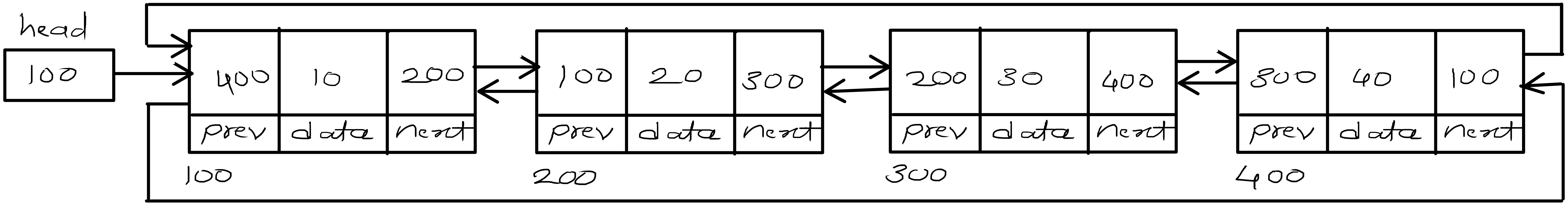
//a. traverse till pos node

//b. add pos+1 node into next of pos-1 node

//c. add pos-1 node inot prev of pos+1 node

Time Complexity : O(n)

Doubly Circular Linked List - Display

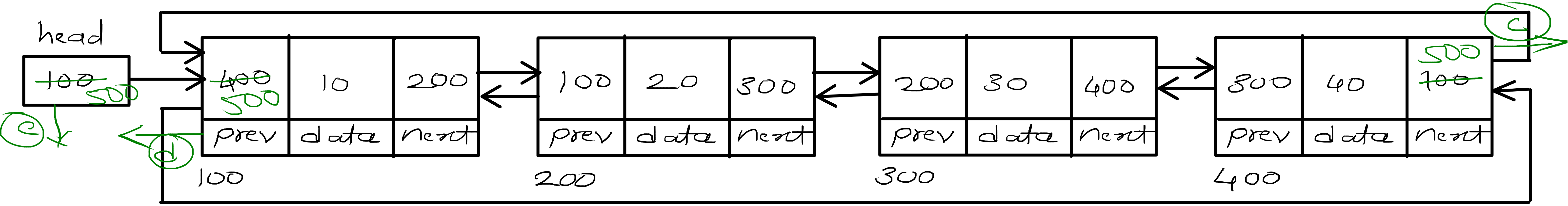


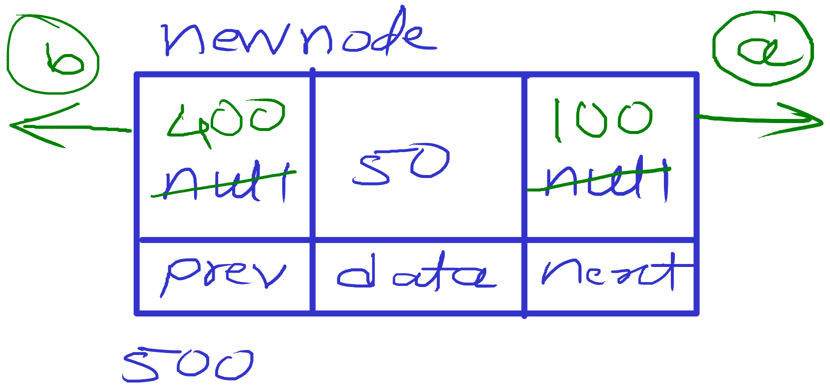

```
void fDisplay() {
    if(isEmpty())
        return;
    Node trav = head;
    do {
        sysout(trav.data);
        trav = trav.next;
    } while(trav != head);
}
```

```
void rDisplay() {
    if(isEmpty())
        return;
    Node trav = head.prev;
    do {
        sysout(trav.data);
        trav = trav.prev;
    } while(trav != head.prev);
}
```

Time Complexity : O(n)

Doubly Circular Linked List - Add First

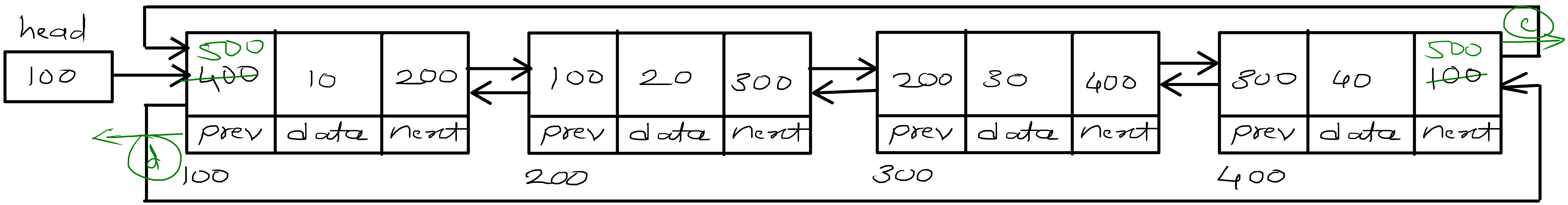


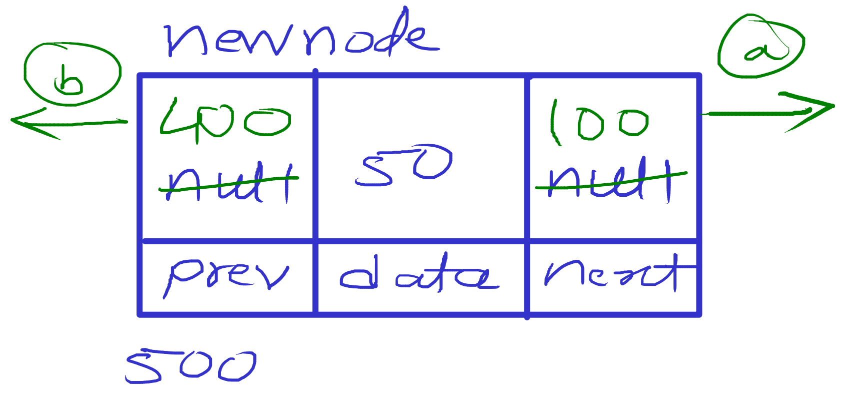



```
void addFirst(int value) {
    Node nn = new Node(value);
    if(isEmpty()) {
        head = nn;
        head.next = nn;
        head.prev = nn;
    }
    else {
        (a) nn.next = head;
        (b) nn.prev = head.prev;
        (c) head.prev.next = nn;
        (d) head.prev = nn;
        (e) head = nn;
    }
}
```

Time Complexity : O(1)

Doubly Circular Linked List - Add Last

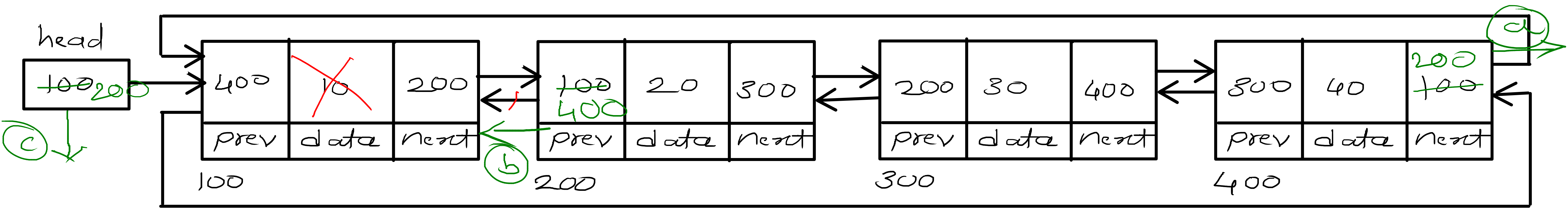





```
void addLast(int value) {
    Node nn = new Node(value);
    if(isEmpty()) {
        head = nn;
        head.next = nn;
        head.prev = nn;
    }
    else {
        nn.next = head;
        nn.prev = head.prev;
        head.prev.next = nn;
        head.prev = nn;
    }
}
```

Time Complexity : O(1)

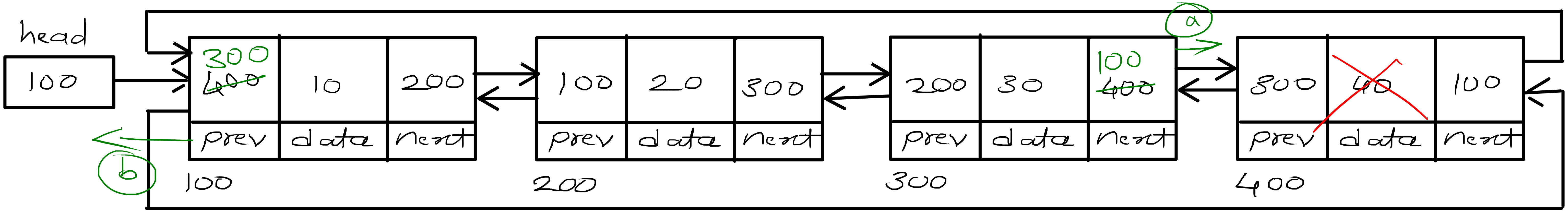
Doubly Circular Linked List - Delete First




```
void deleteFirst() {
    if (Empty())
        return;
    else if (head.next == head)
        head = null;
    else {
        head.prev.next = head.next;
        head.next.prev = head.prev;
        head = head.next;
    }
}
```

Time Complexity : O(1)

Doubly Circular Linked List - Delete Last




```
void deleteLast() {
    if(Empty())
        return;
    else(head.next == head)
        head = null;
    else {
        head.prev.prev.next = head;
        head.prev = head.prev.prev;
    }
}
```

Time Complexity : O(1)

Linked List Applications

1. to implement
    1. stack and queue
    2. Hash Table
    3. Graph

2. OS Data structures - ready queue, job queue, waiting queue (DCLL)

Stack

(push/pop)

1.
    Add First()
    Delete First()

2.
    Add Last()
    Delete Last()

Queue

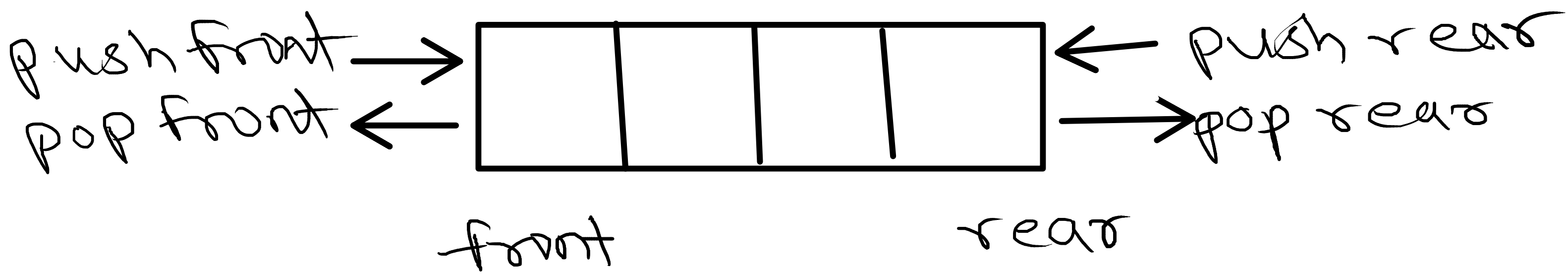
(push/pop)

1.
    Add First()
    Delete Last()

2.
    Add Last()
    Delete First()

Deque

(Double Ended Queue)



Types:

1. Input Restricted deque
    - insert/push is allowed from only one end
2. Output Restricted deque
    - remove/pop is alloed from only one end

```
class List {
    class Node {}
    List() { }
    isEmpty( ) {}
    addFirst(value){}
    deleteFirst( ) { }
    getData ( ) {
        return head.data;
    }
}
```

```
class Stack {
    List list;
    Stack() {
        list = new List()}
    }
    push(value) {
        list.addFirst(value);
    }
    pop ( ) {
        list.deleteFirst();
    }
    peek() {
        return list.getData();
    }
    isEmpty {
        return list.isEmpty();
    }
}
```

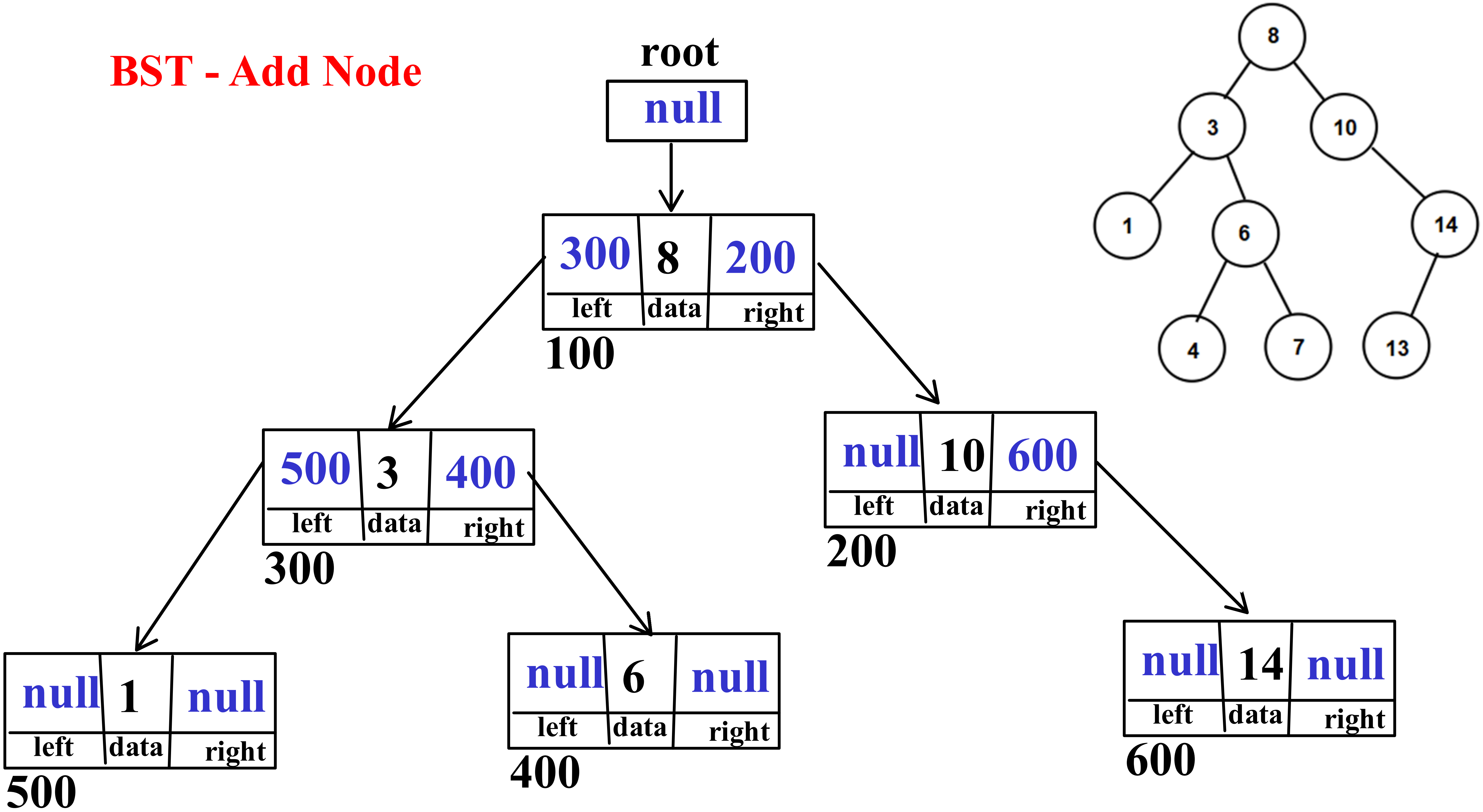
Array Vs Linked List

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

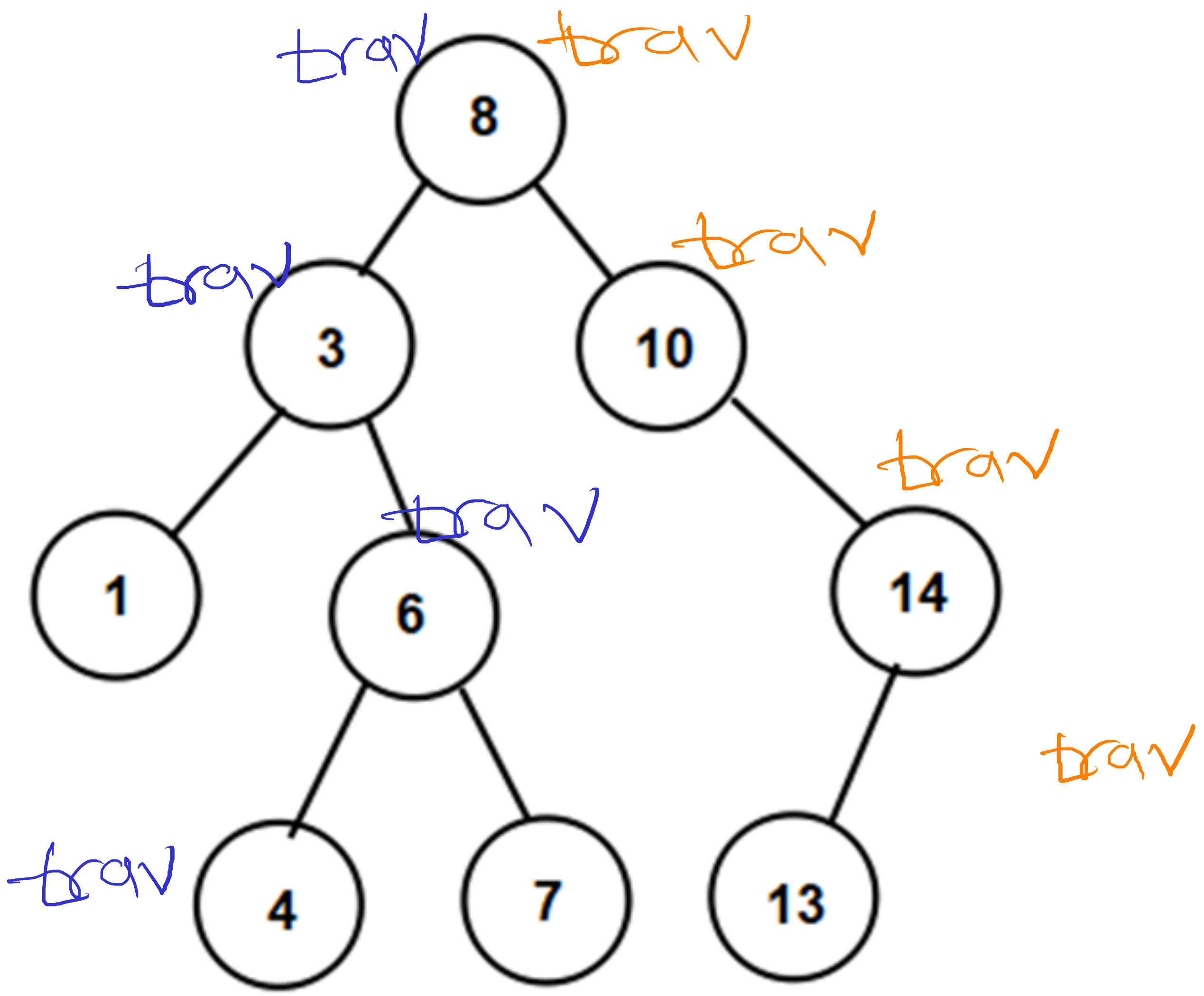
BST - Add Node

//1. create node with given data
//2. if tree is empty
  //a. add newnode into root
//3. if tree is not empty
  //3.1 create trave and start at root
  //3.2 if value is less than current node data
    //3.2.1 if left is empty
      // add newnode into left of current node
    //3.2.2 if left is not empty
      // go into left
  //3.3 if value is greater than current node data
    //3.2.1 if right is empty
      // add newnode into right of current node
    //3.2.2 if right is not empty
      // go into right
  //3.4 repeat step 3.2 and 3.3 till node is not added into tree

BST - Add Node


root
null
300
8
200
left
data
right
100
500
3
400
left
data
right
300
null
10
600
left
data
right
200
null
1
null
left
data
right
500
null
6
null
left
data
right
400
null
14
null
left
data
right
600
8
3
10
1
6
14
4
7
13

BST - Binary Search



//1. start from root
//2. if key is equal to current data
//return current node
//3. if key is less than current data
// search key into left of current node
//4. if key is greater than current data
// search key into right of current node
//5. repeat step 2 to 4 till leaf nodes

Key = 4 , Key is found

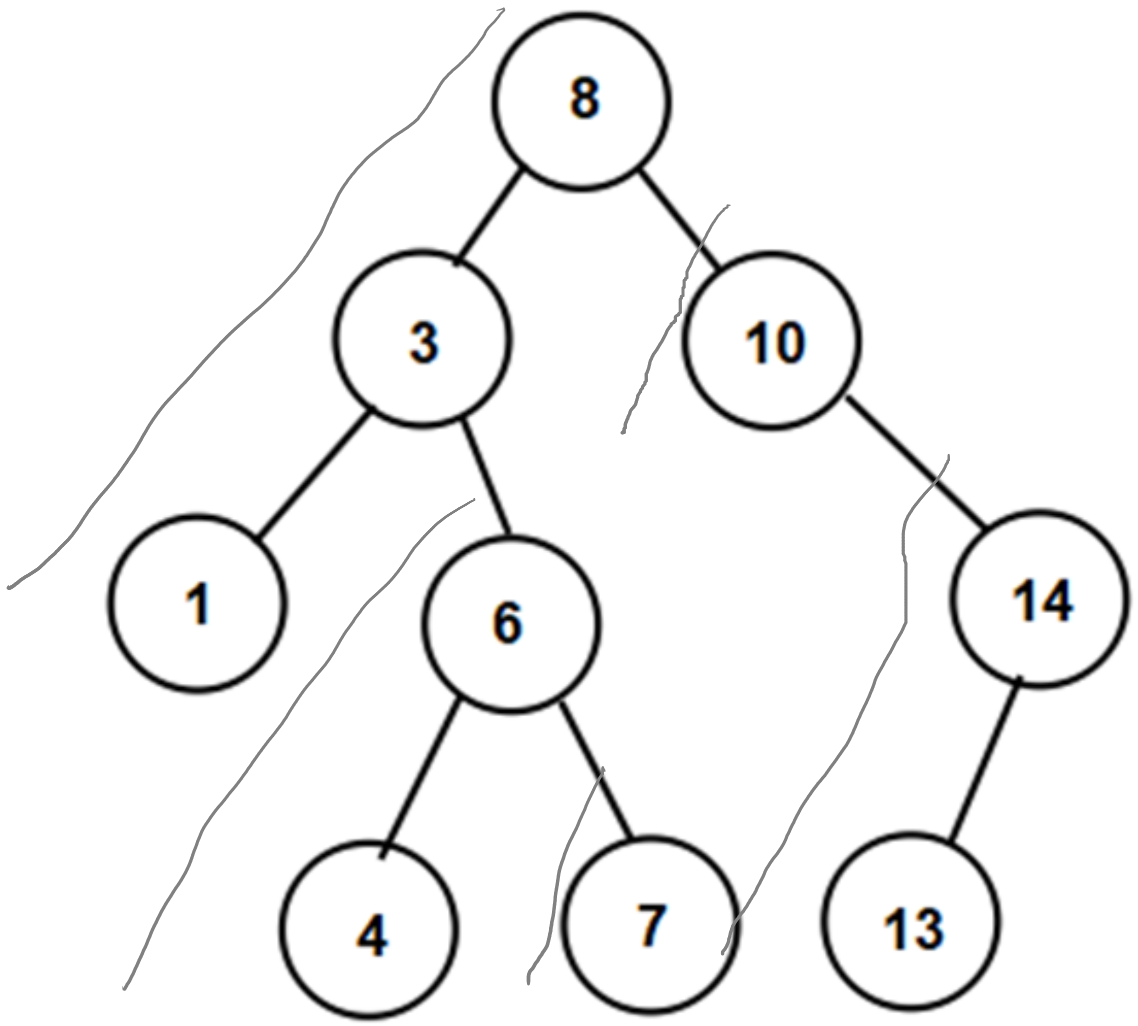
Key = 15

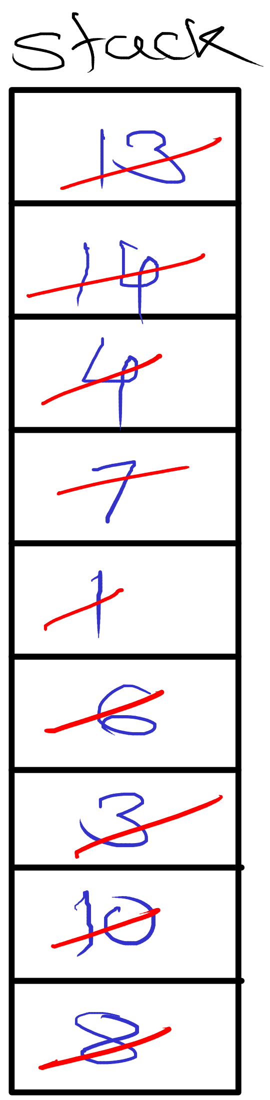
$n = 2^h - 1$

$h = \log n$

Time Complexity $= O(h) = O(\log n)$

BST - DFS (Depth First Search)

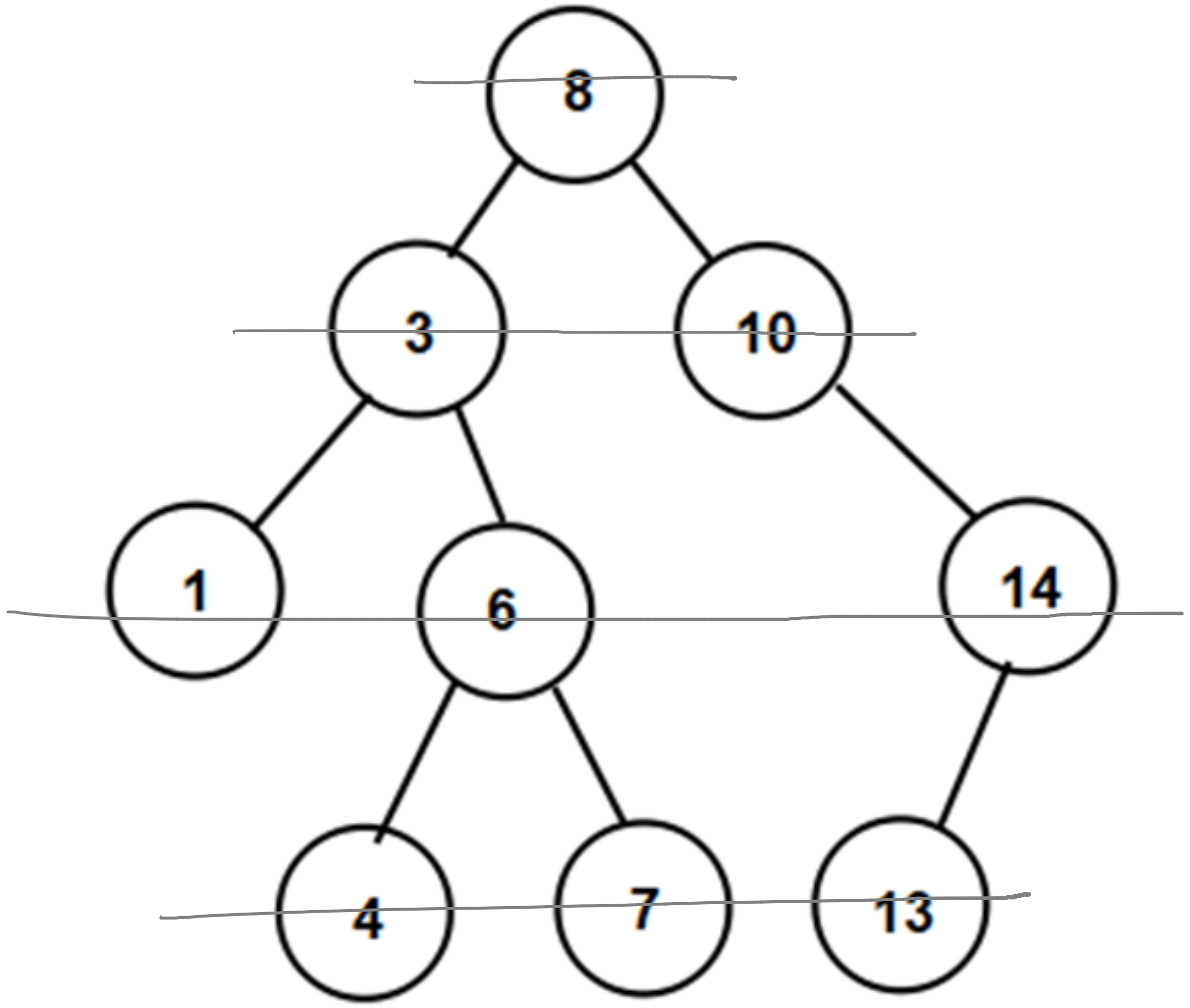


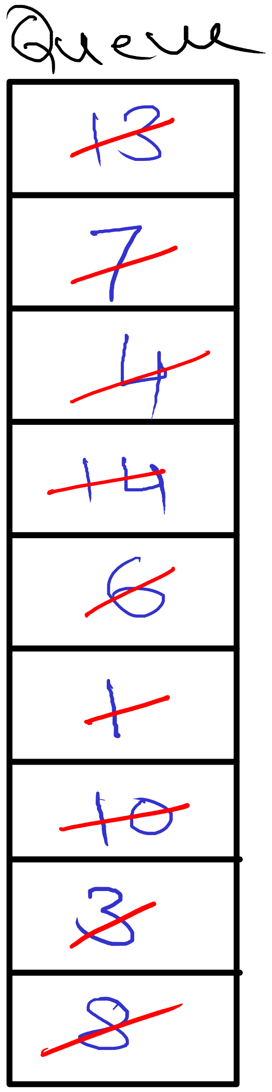


//1. push root on stack
//2. pop one node from stack
//3. visit(print) node
//4. if right exist, push it on stack
//5. if left exist, push it on stack
//6. while stack is not empty
    //repeat ste 2 to 5

DFS Traversal : 8, 3, 1, 6, 4, 7, 10, 14, 13

BST - BFS (Bredth First Search)





//1. push root on queue
//2. pop one node from queue
//3. visit(print) node
//4. if left exist, push it on queue
//5. if right exist, push it on queue
//6. while queue is not empty
//repeat ste 2 to 5

BFS Traversal : 8, 3, 10, 1, 6, 14, 4, 7, 13