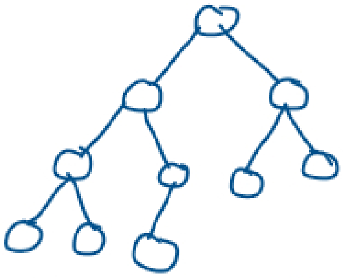
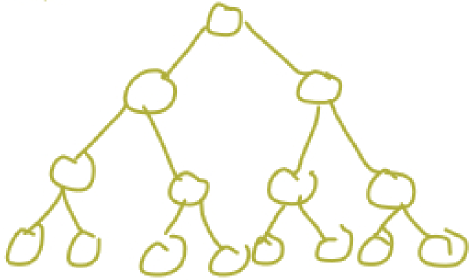
Complete



- Every non leaf has two children except for the last row
- The last row is filled from left → right

Full

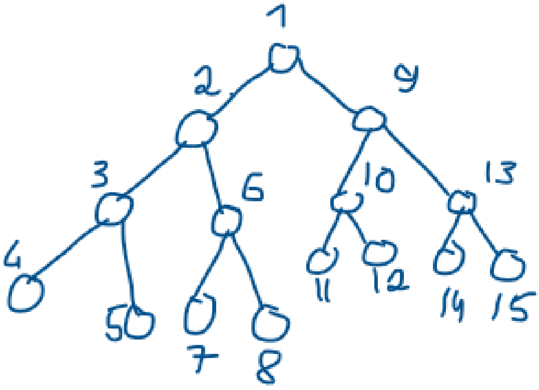


- Every non-leaf has two children
- all the leaves are on the same table

Traversal:

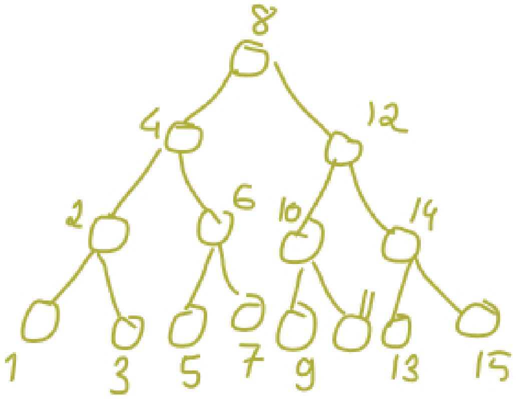
Pre-order traversal:

- visit root node
- visit left child
- visit right child



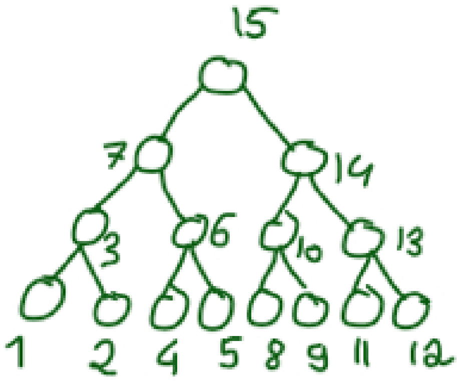
In order traversal

- visit left child
- visit root
- visit right child

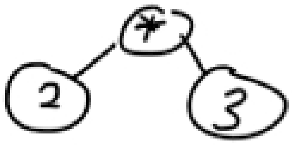


post order traversal:

- left
- right
- root



Expression trees:

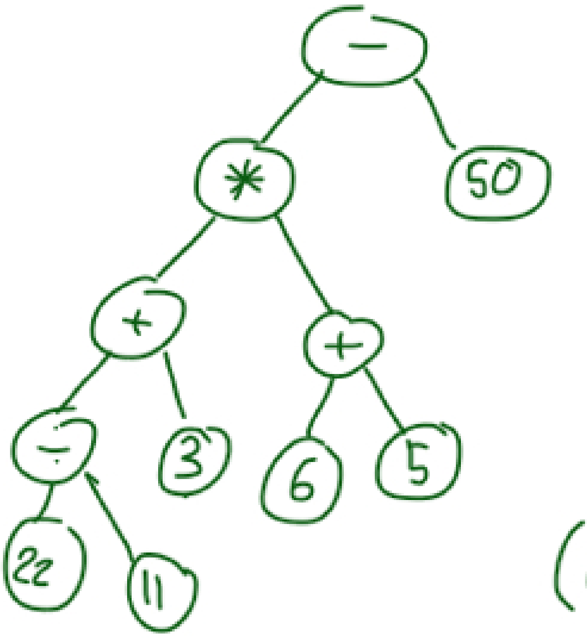


In order: 2 * 3

pre order: * 2 3

post order: 2 3 *

↑
post fix



post order:

22 11 - 3 + 6 5 + * 50 -

In order:

((22 ÷ 11) + 3) * (6 + 5)) - 50

Trees:

Smaller things are on the left

same concept as in linked list

L	data	R

```
class  Node<E>{
    E data;
    Node<E> left, right;

    public Node(E obj){
        E data= obj;
        left = nll ;
        right = null;
    }
}
```

Add: go from root down:

```
    public void add ( E obj ){
        if ( root == null)
                root = new Node (obj);
        else        add( Obj, root);
        current size ++;
    }
 private void add (E obj, Node<E> node){
  If (((Comparable <E>) obj). compareTo (node. Data) > ∅){
      // go right
      if ( node . right==null) {
              node. right = new Node (obj);
              return ;
      }
      return add ( E obj, node . right);
  }
  If (node. left==null ) {
        node. left = new Node (obj);
        return;
  }  return      add( E obj , node. left);
  }
```

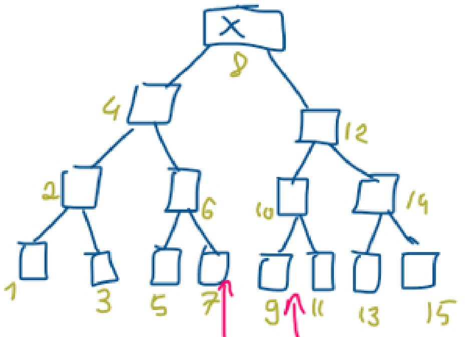
Contains:

```
public boolean contains (E obj){
        return contains(obj, root);
}

private  boolean  contains(E obj, Node<E> node){
        if( node ==null)  return  false;
        if(((  obj . compareTo(node . data)==0)
              return  true;
        if((( obj. compareTo (node. data) >0
              return  contains (obj, node. right);
        else
              return  contains (obj, node. left);
```

Remove:

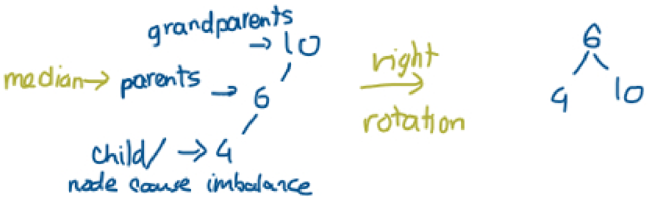
- If remove a leaf node: parents point to null
- If deleting a node with one child: set parents pointers to the child.
- If deleting a node with 2 childs: swap him with in order successor or in order predessor and delete that

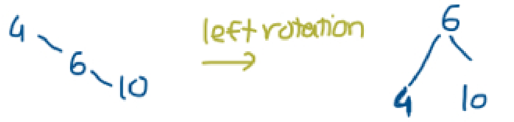


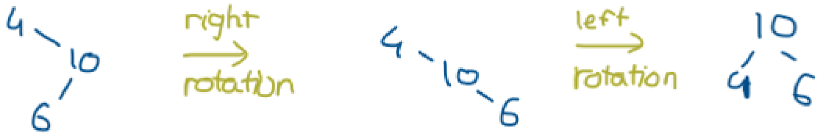
In order Successor : largest thing smaller than the node

In order predessor : Smallest thing bigger than the node

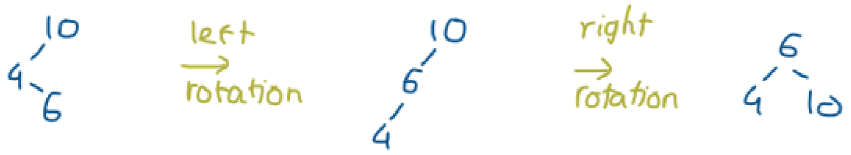
Rotation:







⇒ Right - Left Rotation



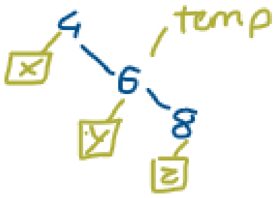
⇒ Left - Right Rotation

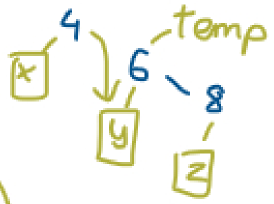
Left:

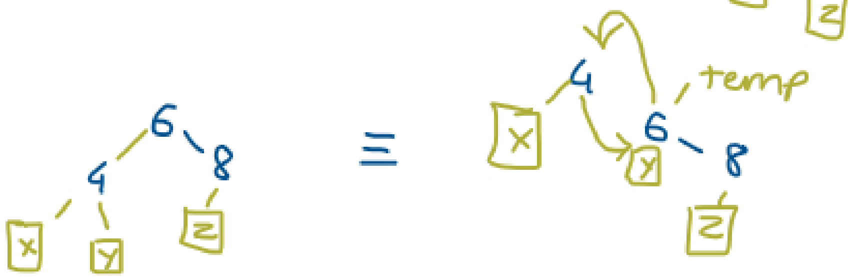
Set temp = grandparents right child

Set grandparents right child = temp left child

set temp left child = grandparent





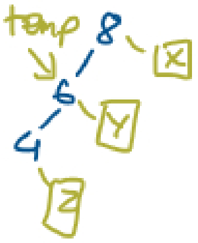


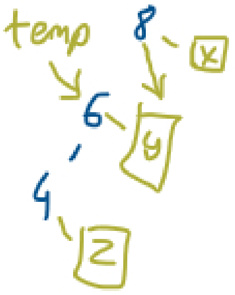
Right:

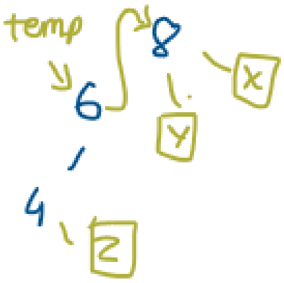
Set temp = grand parents left child

set grandparents left child = temp right

Set temp right child = grand parent



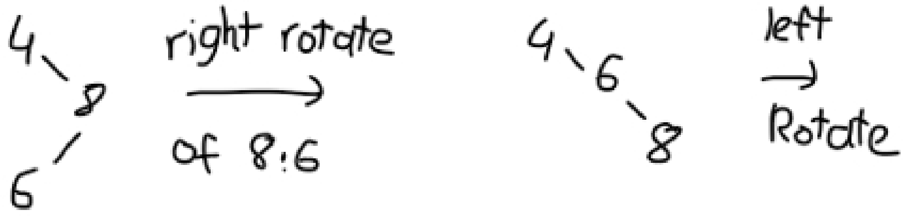




grandparents

```
public Node<E> leftRotate(Node<E> node){
    Node<E> tmp= node.right;
    node.right = temp.left;
    temp.left= node;
    return temp;    → important for Left-Right Rotate
}

public Node<E> rightRotate(Node<E> node){
      Node<E> tmp= node.left;
      node.left = temp.right;
      temp.right= node;
      return temp;
}
```



```
public Node<E> rightLeftRotate(Node<E> node){
        node.right = rightRotate(node.right);
        return lefRotate(node);
}

public Node<E> leftRightRotate(Node<E> node){
    node.left= leftRotate (node.let);
    return rightRotate( node);
}
```