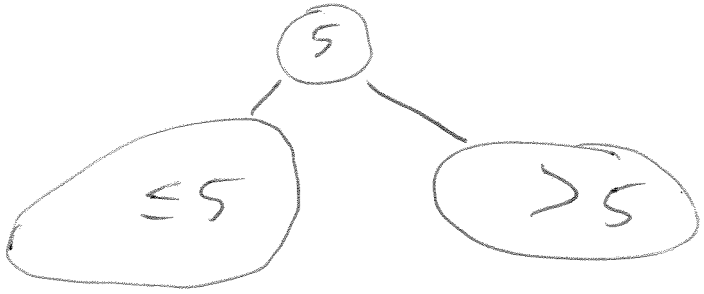
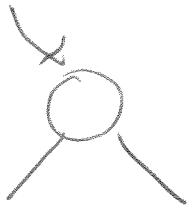
Binary Search trees



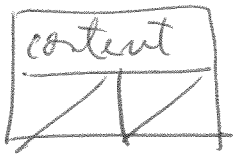
At all nodes!

```
public class BinarySearchTree {
    BinarySearchTreeNode root;
    public BinarySearchTree(){
        root = null;
    }
    public boolean search(Object val) {
    return root.search(val); // interesting stuff
    }
    public void insert(Object val) {
        root.insert(val);
    }
    // remove
```

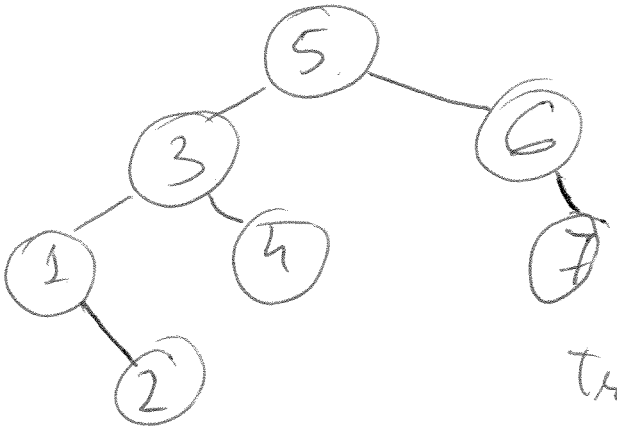


comparable type; see code

```
public class BinarySearchTreeNode {
    Object content; // Comparable
    BinarySearchTreeNode left, right;
    public BinarySearchTreeNode (Object val) {
        content = val;
        left = null;
        right = null;
    }
// Search: Binary search-style
    public boolean Search (Object val) {
        if (val.equals (content)) {
                ↑ not ==!
            return true;
        }
        if (val.compareTo (content) == -1) {   // val is smaller so look on left
            if (left == null) { // no more tree!
                return false;
            }
            return left.search (val); // looking under
                                      // left node
// Trees are a recursive data structure
// Grammar notation: BSTNode = null | Obj BSTNode BSTNode
                                       or ↑
// ListNode = null | Obj ListNode
```



Inserting in a Binary Search Tree:



Insert 2

Put at leaf, go "recursively" down the tree to find appropriate spot

Insert 7

Navigate down tree left or right as needed

Inside BinarySearchTree class:

```
public void insert(val) {
    if (root != null) {
        root.insert(val);
    } else {
        root = new BinarySearchTreeNode(val); // tree was empty
    }
}
```

Inside BinarySearchTreeNode:

```
public void insert(Object val){
  // Are we at a leaf? Base case
  if((left == null) && val.compareTo(content)==-1)
     // make a node and insert it
     BinarySearchTreeNode newNode = new BSTNode(val);
     left = newNode;
  }
  if((right == null) && val.compareTo(content)==1){
     // similar
  }
  if(val.compareTo(content)==-1){
     // value is smaller; there is a left subtree
     left.insert(val);
  } else if(val.compareTo(content)==1){
     right.insert(val);
  }
  // val at node; do nothing; alternative logic
                                  is possible
}
```

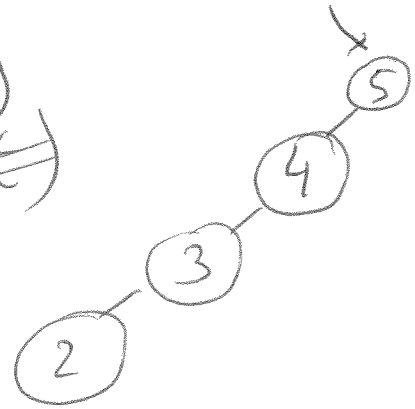
$T(n) = c + T(k)$

If tree is balanced $O(\log_2 n)$

```
// looking on right
if (right == null)
    return false;
return right.search(val);
}
```

Suppose tree has n nodes; taken branch has k nodes

$T(n) = c + \underline{T(k)}$



Worst case: $T(n) = c + T(n-1)$

$\Downarrow$ (~~Sel. sort~~)

$O(n^2)$

~~$T(n) = c + T(n-1) =$~~

~~$= c + c + T(n$~~

$T(n) = c + T(n-1) = c + c + T(n-2)$

$= 2c + T(n-2) = \ldots = \Rightarrow O(n)$

$\uparrow_n$

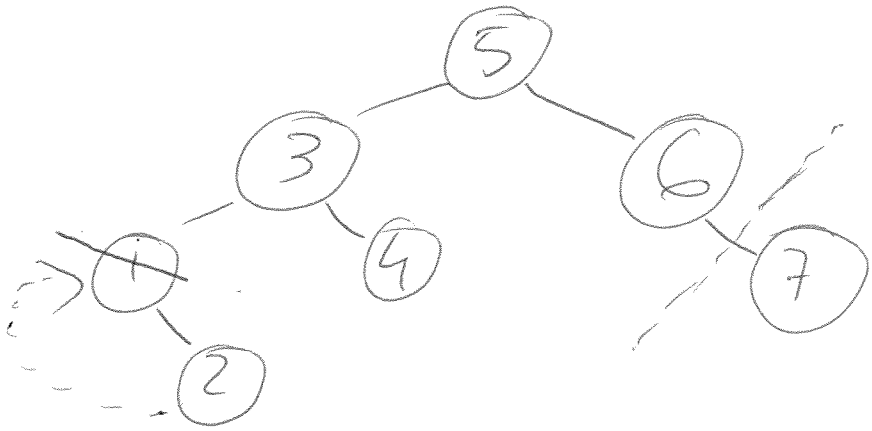
Best case: $T(n) = c + T\left(\frac{n}{2}\right)$

(always half the nodes on one side)

$T(n) = c + T\left(\frac{n}{2}\right) = c + c + T\left(\frac{n}{4}\right) =$

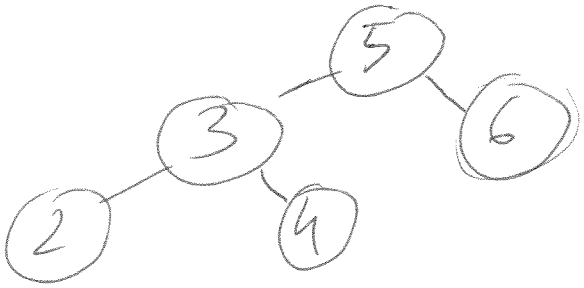
$= 2c + c + T\left(\frac{n}{8}\right) = \ldots = c \log_2 n + c_1$

$\Rightarrow O(\log_2 n)$

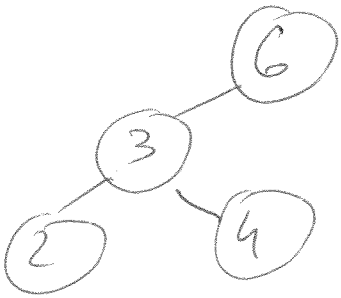


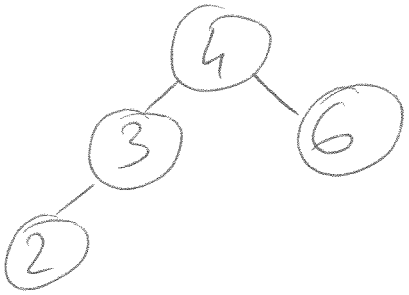
Remove 7

Remove 1 : link the single child in place of its parent



Remove 5





Largest value in left subtree or
Smallest value in right subtree
left → right .... right
right → left ... left

Binary Search Tree class:

Need to check if root itself is removed

Binary Search Tree Node class:

//Need to check if value to be removed is in the tree:

if (!search(val)) return;

// root being null case is taken care of by parent)

//Node for value to replace the node

if (left == null && val.compareTo(content) == -1)

Either: modify at node content

content = right.content

left = right.left.~~content~~;

right = right.right;

Or: if we have a reference to parent node:

parent.left = right