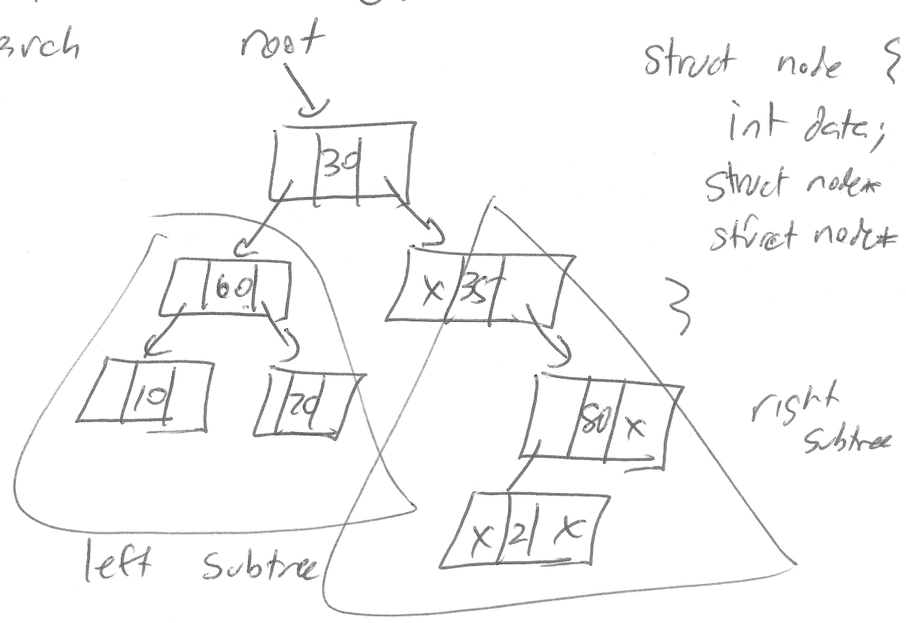
COP 3502 3/10/26

Binary Trees - traversals (preorder, inorder, postorder)
Binary Search Trees - Insertion
Ceiling Function
Standard function - sum, height

functions to support

add/insert
delete
search

Goal better than O(n)



struct node {
int data;
struct node* left;
struct node* right;
}

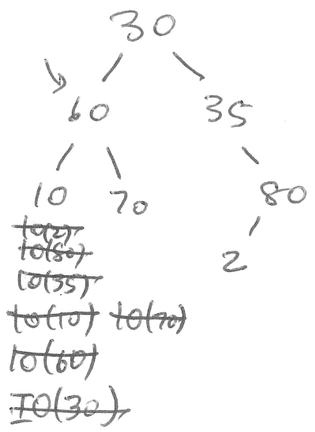
~~p(70)~~
~~p(80)~~
~~p(35)~~
~~p(10)~~ ~~p(70)~~
~~p(60)~~
~~p(30)~~

preorder

1) root
2) preorder(root->left)
3) preorder(root->right)

30, 60, 10, 70, 35, 80, 2

Run-time O(n) n=# nodes



~~to(2)~~
~~to(80)~~
~~to(35)~~
~~to(10)~~ ~~to(70)~~
~~to(60)~~
~~to(30)~~

~~po(2)~~
~~po(80)~~
~~po(35)~~
~~po(10)~~ ~~po(70)~~
~~po(60)~~
~~po(30)~~

stack

inorder

1. inorder (root → left)
2. root
3. inorder (root → right)

left sub | root | right sub

10, 60, 70 | 30 | 35, 2, 80

postorder

1. postorder (root → left)
2. postorder (root → right)
3. root

10, 70, 60 | 2, 80, 35 | 30

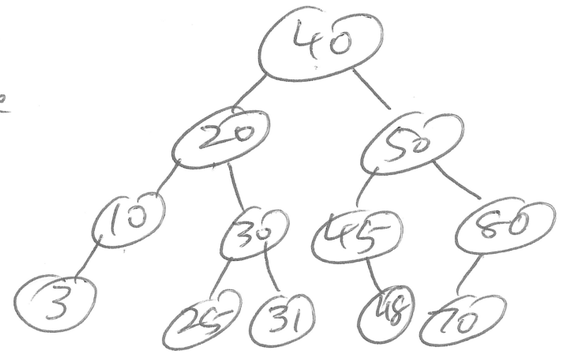
left sub | right sub | root

Binary Serch Tree

for each node, all values in left subtree are $<=$ root node

all values in right subtree $>$ root value

leaf node → left, right are null



Insertion

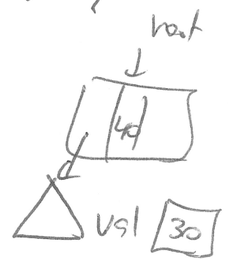
48

3

Insert Recursively

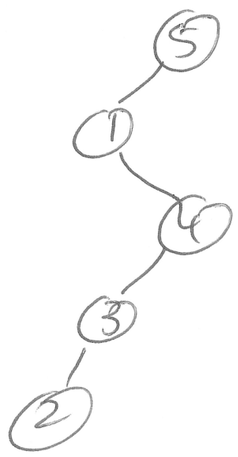
```
node* insert(node* root, int val) {
    if (root == NULL)
        return createNode(val);

    if (val < root->data)
        root->left = insert(root->left, val);
    else
        root->right = insert(root->right, val);
    return root;
}
```



Run-time insert
$O(h)$
h = height of tree

Search is like insert
$O(h)$

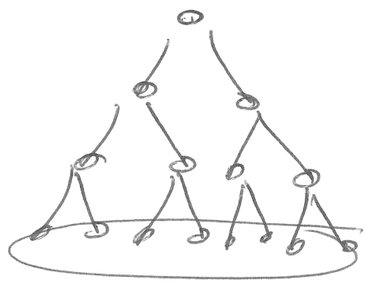


2^{n-1} possible trees that have a height of $n-1$, where tree has n nodes.

Worst case of a binary search tree is $O(n)$

worst case height BST → O(n)

best case height



$n = 1+2+4+8+\ldots 2^h$

$= 2^{h+1} - 1$

h

$n = 2^{h+1} - 1$

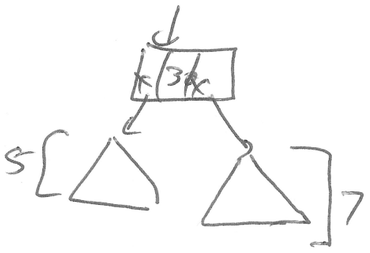
$(n+1) = 2^{h+1}$

$\log_2(n+1) = h+1$

$h = \log_2(n+1) - 1$

insert
search ← O(lgn)

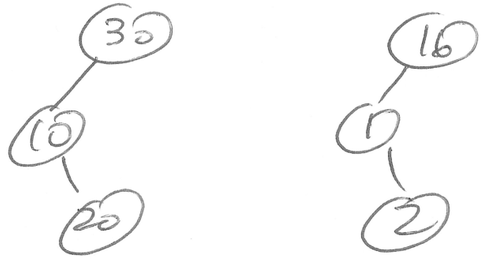
AVG CASE is also O(lgn)




```
int height(node* root) {
    if (root == NULL) return -1;
    int max = height(root->left);
    int myr = height(root->right);
    if (myr > max) max = myr;
    return max+1;
}
//If you need info OFTEN store in
//the node.
```

```
int sum(node* root) {
    if (root == NULL) return 0;
    return root->data + sum(root->left) + sum(root->right);
}
```

Are 2 BST structurally identical

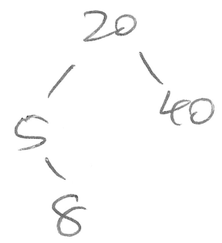



```
int equalStructure(node* t1, node* t2) {
    if (t1==NULL && t2 == NULL)
        return 1;
    if (t1 == NULL || t2 == NULL)
        return 0;
    return equalStructure(t1->left, t2->left) &&
           equalStructure(t1->right, t2->right);
}
```

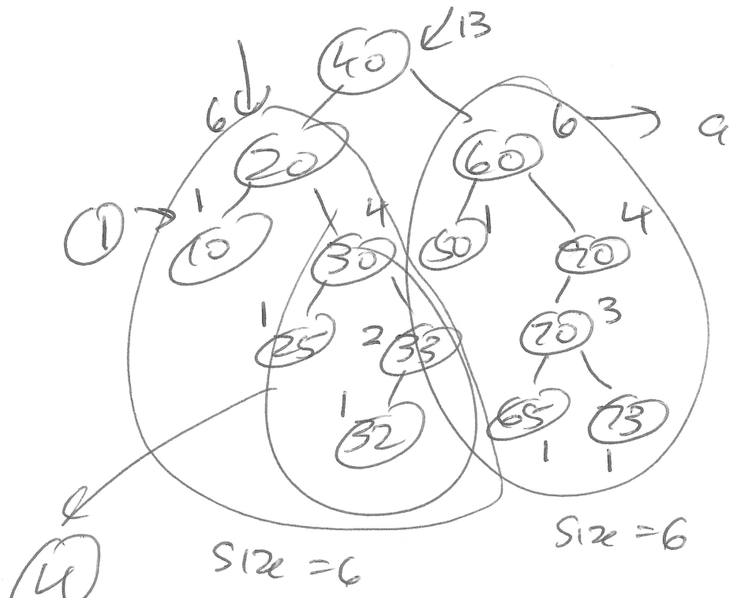
equalTrees add

t1->data == t2->data &&

Sample



k^{th} smallest



5^{th} smallest

also store # nodes in subtree

find (5^{th} smallest, 20)
→ find (3^{rd} smallest, 30)
→ find (1^{st} smallest, 33)
→ find (1^{st} smallest, 32)