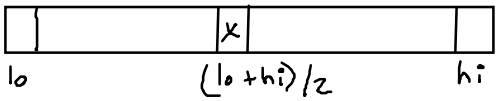
Binary Search Trees

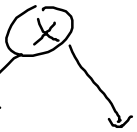
Recall Binary Search on a sorted array:



- inspect value at middle index
- if what we want, done!
- if our search key $< x$, search only left half of array
- if our search key $> x$, search only right half of array

Binary Search Tree

↳ Binary Tree with ordering principle



all values in left subtree of x are less than x

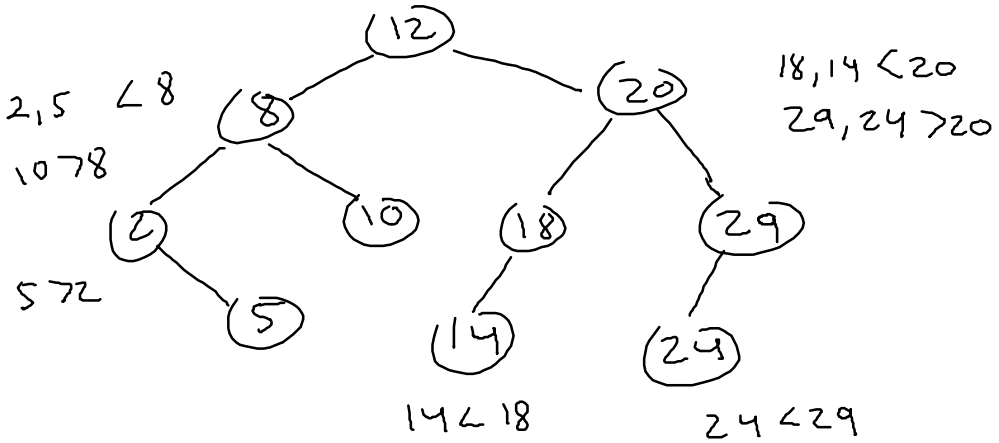
all values in right subtree of x are greater than x

AT EVERY NODE

Example:

8, 2, 10, 5 all < 12

14, 18, 20, 29, 24 all > 12



2, 5 < 8

10 > 8

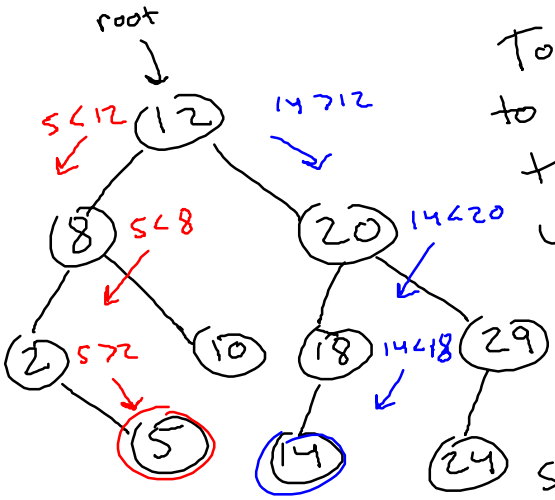
5 > 2

18, 14 < 20

29, 24 > 20

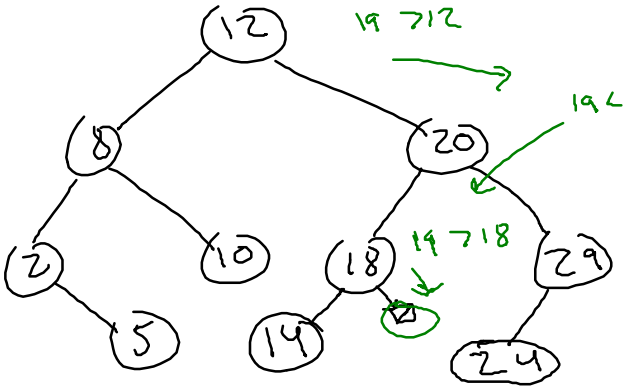
14 < 18

24 < 29



To search, compare to value at root. If that value isn't what you want, choose left or right subtree based on whether search key is less than or greater than value in root node, and proceed recursively.

Unsuccessful searches are those that end at a null subtree. Our search down tree tells us search key would have to be in that subtree, and it's NULL, so search key not in tree

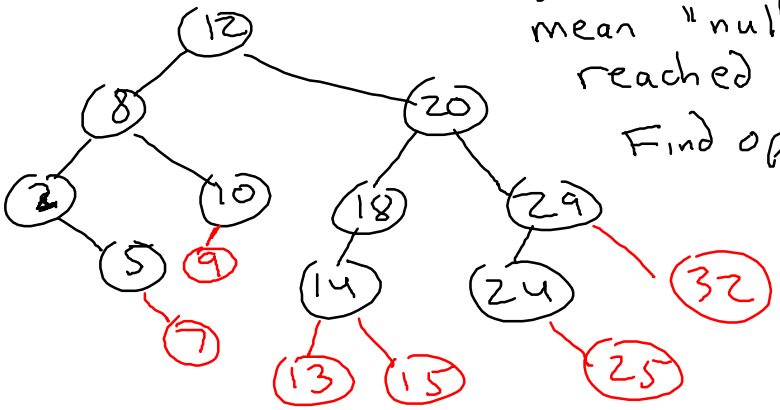


```
int Find(TreeNode* TN, Etype X)
{
  if (TN == NULL)
    return 0;
  else if (TN->elem == X)
    return 1;
  else if (X < TN->elem)
    return Find(TN->left, X);
  else // x > TN->elem
    return Find(TN->right, X);
}
```

must check this case first

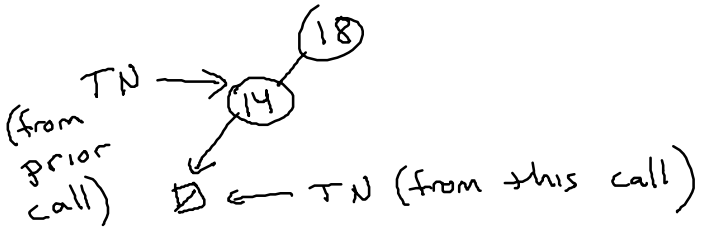
Don't forget returns here!

Insertion → ~~F~~ind where node should be, then put it there (i.e. insert as new leaf). By "should be" we mean "null subtree reached during Find operation"

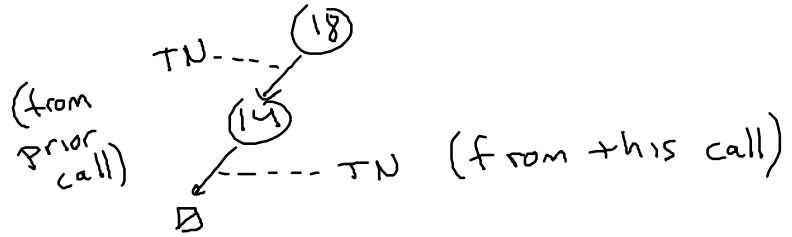


```
//not quite correct
void Insert(TreeNode* TN, EtypeX)
{   if (TN == NULL)
        TN = new TreeNode(x);
    ⋮
    ⋮  more to come
```

Wrong because TN is a copy of parent's ~~left~~ or right pointer, not actual pointer



We want this:



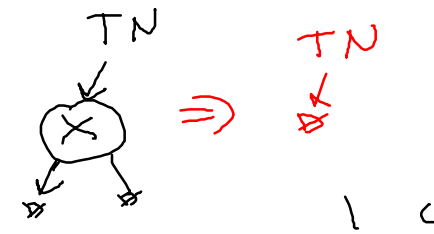
So pass by reference

```
void Insert(TreeNode*& TN, EType X)
{
    if (TN == NULL)
        TN = new TreeNode(X);
    else if (X < TN->elem)
        Insert(TN->left, X);
    else if (X > TN->elem)
        Insert(TN->right, X);
    // else X == TN->elem, and
    // we'll ignore duplicates. We
    // don't have to, but that's
    // how we'll handle duplicates.
}
```

Remove (note → pass by reference again

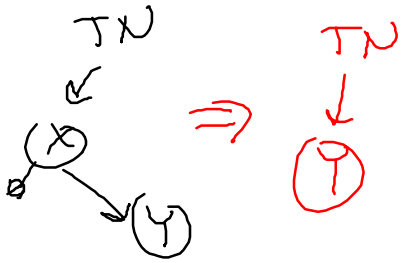
```
void Remove(TreeNode *& TN, EType X)
{   if (TN == NULL)
       return;
    else if (X < TN -> elem)
       Remove (TN -> left, X);
    else if (X > TN -> elem)
       Remove (TN -> right, X);
    else // TN -> elem == X
    {
       // we have found value to
       // be removed; now run
       // remove cases (next page)
    }
}
```

Remove cases: 0 children



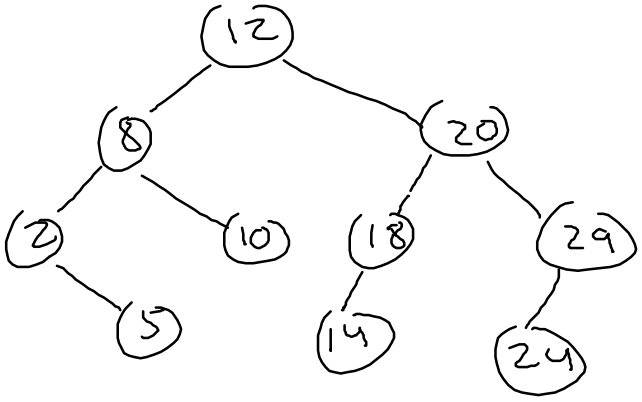
delete TN;
TN = NULL

1 child



temp = TN
TN = x's child
delete temp

2 child case → deleting node would separate subtrees; we'd rather not have to reassemble them so we do something different



If we are deleting 20, let's not delete the node but instead replace the value with another value in the tree.

The only values that would work (i.e. ordering principle would not be violated) would be what comes after or before 20 in an inorder traversal.

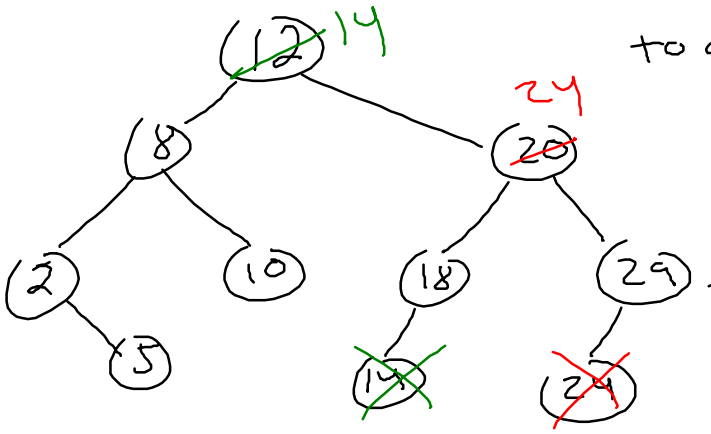
In Order Successor (IOS) of a node is the minimum value (leftmost value) in right subtree of that node.

In Order Predecessor (IOP) of a node is the maximum value (rightmost value) in left subtree of that node.

Note that by definition, IOS has no left child and IOP has no right child, so deleting either one is a 0 or 1 child removal case.

2- child remove
- replace value with IOS
- remove node that originally held IOS

(IOP works too, but in this course we will use IOS)



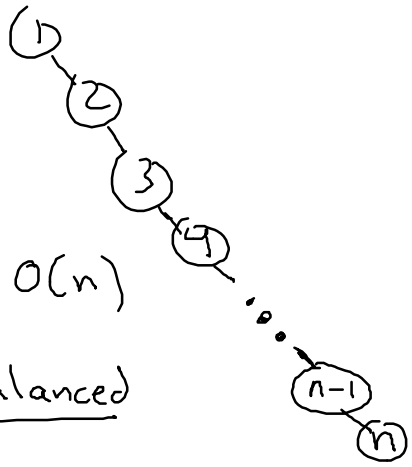
to delete 20, replace w/ 24 and delete 24

to delete 12, replace w/ 14 and delete 14

Consider this code, where T is a BST:

```
for (int i = 1; i <= n; i++)
    T.Insert(i);
```

You get this tree:

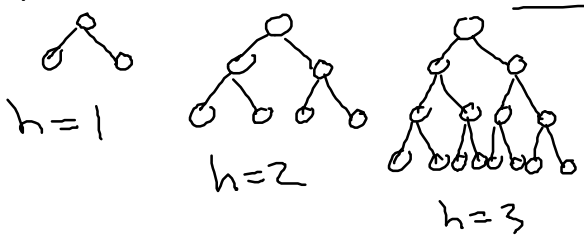


worst case BST!

height is $n-1$

which is $O(n)$

Perfect Trees are balanced



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

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

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

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

$$h = O(\lg n)$$

i.e. height of perfect tree is logarithmic on # of nodes

The average BST is roughly balanced, i.e. $h = O(\lg n)$ [proof in book if you are curious]

Since BST operations (Remove, Insert, Find) all involve traversal of one path from root to leaf, all are $O(h)$ where h is height of tree.

worst case: $h = O(n)$

BST ops are $O(n)$

avg case: $h = O(\lg n)$

BST ops are $O(\lg n)$