

# Binary search tree operations

Lecturer: Arran Stewart

# Outline

We discuss:

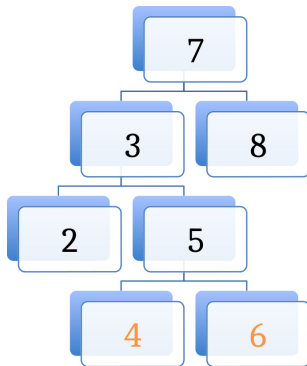
- ▶ What are common operations binary search trees (BSTs) support?
- ▶ How do we implement those in Java?

# Binary search tree operations

- ▶ We have seen that a binary search tree (BST) is an efficient data structure for storing ordered collections of data.
- ▶ We will now look briefly at Java implementations of operations to insert, find minimum and remove the minimum element from a BST.
- ▶ More complex operations exist for maintaining balanced BSTs, but they are beyond the scope of this course
- ▶ However, if you are interested some details are available in the text book and the UWA lecture notes.

# BST insertion

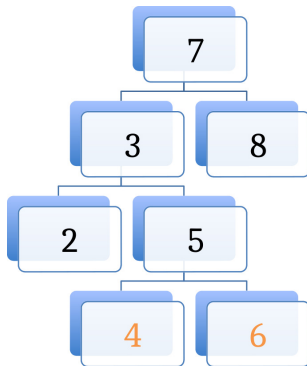
- ▶ We have seen an example of insertion, but let us see it again with an implementation using Java generics, where the tree never contains duplicates.
- ▶ Let us suppose we already have a BST containing the numbers 2, 3, 4, 7, and 8.
- ▶ If we want to insert the numbers 4 and then 6, what do we need to do?



# BST insertion

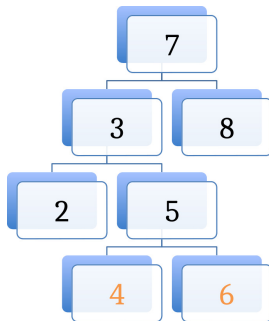
The general steps are:

- ▶ perform a search for the element
- ▶ if the element is found, we will report there is a duplicate item
- ▶ if an empty node is reached, insert a new node containing the element



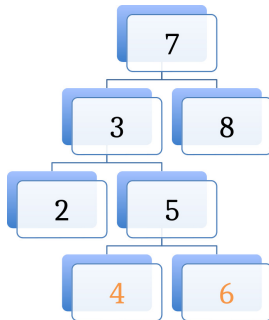
# insert

```
public BinaryTreeNode<T> insert(T a, BinaryTreeNode<T> t) {  
    if (t == null) {  
        t = new BinaryTreeNode<T>(a,null,null);  
    } else if ( t.value.compareTo(a) < 0 ) {  
        t.left = insert(a,t.left);  
    } else if ( t.value.compareTo(a) > 0 ) {  
        t.right = insert(a,t.right);  
    } else { //tried to insert duplicate  
        throw new DuplicateItem( a.toString() );  
    }  
    return t;  
}
```



## findMin

- ▶ The `findMin` operation finds the minimum element by starting at the root of the tree and taking the left node until the left child is `null` – i.e. we reach a node with no children.

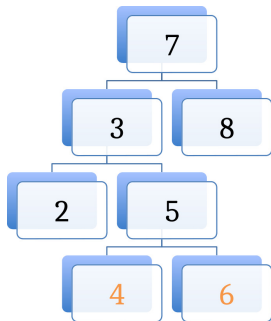


## findMin

Here is a Java implementation of findMin:

```
BinaryTreeNode<T> findMin( BinaryTreeNode<T> n ) {  
    if( n != null ) {  
        while( n.left != null ) {  
            n = n.left;  
        }  
        return n;  
    }  
}
```

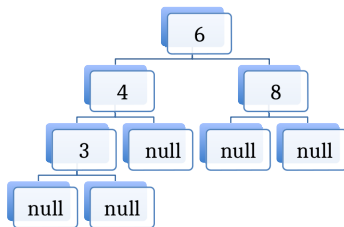
Let us examine the operation of this on the tree to the right.





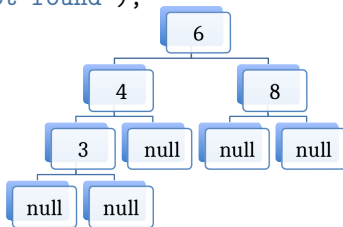
## removeMin

- ▶ The `removeMin` operations removes the minimum element from a tree.
- ▶ A BST will also usually have more complicated methods to remove *any* element from the tree, but we will just examine the simpler `removeMin` method.
- ▶ Like `findMin`, this method keeps searching left subtrees until it reaches the leftmost point.



## removeMin Java code

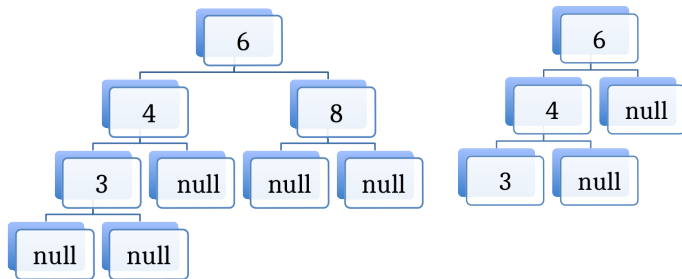
```
private BinaryTreeNode<T> removeMin(BinaryTreeNode<T> t) {  
    if (t==null) {  
        throw new ItemNotFound("Min not found");  
    } else if (t.left != null) {  
        t.left = removeMin( t.left );  
        return t;  
    } else {  
        return t.right;  
    }  
}
```



null nodes are shown in the diagram to help understand how the code works.

## removeMin

Suppose we call removeMin on the left hand tree below; assume we have variables node6, node4, and node3, pointing to the nodes containing those numbers.



## removeMin

1. removeMin (node6) has node6.left != null  
so node6.left = removeMin(node4); then return node6
2. removeMin(node4) has node4.left != null  
So node4.left = removeMin(node3); then return node4;
3. removeMin(node3) has node3.left == null  
So take the else case and return node3.right which is null

Returning to call 2, set node4.left = null (the result of removeMin(node3)) and return node4.

