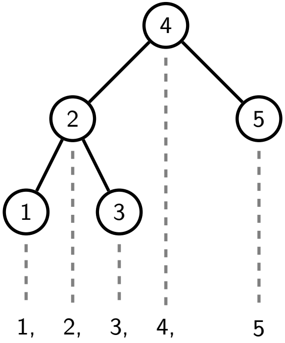
Binary Search Trees

Binary Search Trees

A Binary Search Tree is a tree which is either **empty** or

1. values in the left subtree are **smaller** than in the root
2. values in the right subtree are **larger** than in the root
3. root's left and right subtrees are also Binary Search Trees



$\Longrightarrow$ values in the flattened Binary Search Tree are in order!

A Binary Search Tree is a Binary Tree, so the same constructors and accessors apply. It is just that there is an extra constraint that the node value ordering must be maintained during construction and manipulation.

Binary Search Trees: Insertion

```
insert(v,bst) {
  if ( isEmpty(bst) )
    return MakeTree(v, EmptyTree, EmptyTree)
  elseif ( v < root(bst) )
    return MakeTree(root(bst),
                    insert(v,left(bst)),
                    right(bst))
  elseif ( v > root(bst) )
    return MakeTree(root(bst),
                     left(bst),
                     insert(v,right(bst)))
  else error("Error: value already in tree")
}
```

This creates a new BST which is a copy of the old one but with the extra value inserted correctly. For many applications, this is inefficient because we usually do not need a new copy; we usually want to update the data structure we have in place.

Binary Search Trees: Insertion in Java

We can instead insert the value in place, modifying an existing BST if we use pointers. Here is an implementation in Java (full working program is in Canvas):

```java
public class BSTTree {
  private BSTNode tree = null;

  private static class Node {
    private int     val;
    private Node left, right;

    public BSTNode(int val, Node left, Node right){
      this.val=val; this.left=left; this.right=right;
    }

  // insert methods here
}
```

Binary Search Trees: Insertion in Java

```java
public void insert(int v) {
  if (tree == null) tree = new Node(v, null, null);
  else insert(v, tree);
}

private void insert(int v, Node ptr) {
  if (v < ptr.val) {
    if (ptr.left == null)
      ptr.left = new Node(v, null, null);
    else insert(v, ptr.left);
  }
  else if (v > ptr.val) {
    if (ptr.right == null)
      ptr.right = new Node(v, null, null);
    else insert(v, ptr.right);
  }
  else throw new Error("Value already in tree");
}
```

Searching Binary Search Trees

Starting from the root node, how do we determine whether a value `x` is in the tree?

If the tree is empty, `x` is not in the tree! Otherwise, compare `x` and the value stored in the root. There are three possibilities:

- They are equal $\implies$ we found it!
- `x` is smaller $\implies$ we search the left subtree.
- `x` is larger $\implies$ we search the right subtree.

Searching Binary Search Trees Recursively

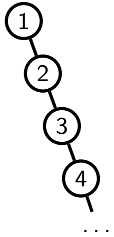
```
isIn(value v, tree t) {
   if ( isEmpty(t) )
      return false
   elseif ( v == root(t) )
      return true
   elseif ( v < root(t) )
      return isIn(v, left(t))
   else
      return isIn(v, right(t))
}
```

Searching Binary Search Trees Iteratively

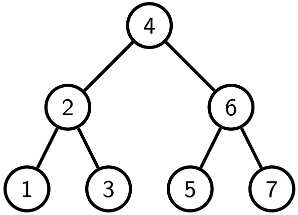
```
isIn(value v, tree t) {
    while ( (not isEmpty(t)) and (v != root(t)) )
        if (v < root(t) )
            t = left(t)
        else
            t = right(t)
    return ( not isEmpty(t) )
}
```

Searching Binary Search Trees

Compare complexities:



vs.



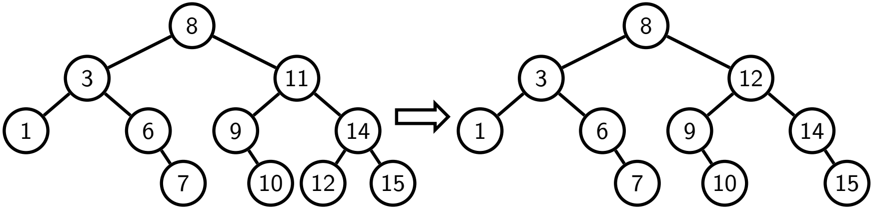
- For the left case, complexity of search is $O(n)$
- For the right case, complexity of search is $O(\log_2 n)$
- For the average case (not proven here), complexity of search is also $O(\log_2 n)$
- On average, complexity of insertion is $O(\log_2 n)$, because it depends on the height of the Binary Search Tree, which is $O(\log_2 n)$
- Average height of a General Binary Tree is $O(\sqrt{n})$

Deleting from a Binary Search Tree

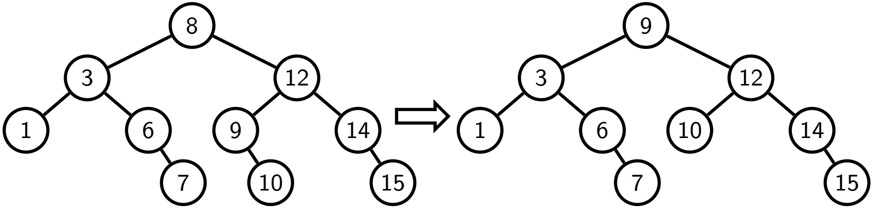
- First Option:
  - Insert all items except the one to be deleted into a new BST
  - n inserts of complexity $O(\log_2 n)$, results in complexity $O(n \log_2 n)$
  - Worse than deleting from an array: $O(n)$
- Second Option:
  1. Find the node containing the element to be deleted:
  2. If it is a leaf, just remove it
  3. Else, if only one of the node's children is not empty, replace the node with the root of the non-empty subtree
  4. Else,
     - 4.1 find the left-most node in the right sub-tree (this contains the smallest value in the right sub-tree)
     - 4.2 replace the value to be deleted with that of the left-most node
     - 4.3 replace the left-most node with its right child (may be empty)
  - Complexity $O(\log_2 n)$

Deleting from a Binary Search Tree

Delete 11:



Delete 8:



Deleting from a Binary Search Tree

```
delete(value v, tree t) {
 if ( isEmpty(t) )
   error("Error:␣given␣item␣is␣not␣in␣given␣tree")
 else
   if ( v < root(t) )
     return MakeTree(root(t), delete(v,left(t)), right(t))
   else if ( v > root(t) )
     return MakeTree(root(t), left(t), delete(v,right(t)))
   else
     if ( isEmpty(left(t)) )
       return right(t)
     elseif ( isEmpty(right(t)) )
       return left(t)
     else return
       MakeTree(smallestNode(right(t)), left(t),
                                   removeSmallestNode(right(t))
}
```

Deleting from a Binary Search Tree

```
smallestNode(tree t) {
  // Precondition: t is a non-empty binary search tree
  if ( isEmpty(left(t) )
      return root(t)
  else
      return smallestNode(left(t));
}

removeSmallestNode(tree t) {
  // Precondition: t is a non-empty binary search tree
  if ( isEmpty(left(t) )
      return right(t)
  else
      return MakeTree(root(t),
                      removeSmallestNode(left(t)),
                      right(t))
}
```

Checking whether a Binary Tree is a BST

Simple algorithm:

1. If the tree is empty, it is a valid BST
2. Else, it is a valid BST if:
   - 2.1 all values in the left branch are less than the root and
   - 2.2 all values in the right branch are greater than the root and
   - 2.3 the left branch is a valid BST and
   - 2.4 the right branch is a valid BST

```
isbst(tree t) {
   if ( isEmpty(t) )
     return true
   else
     return ( allsmaller(left(t),root(t)) and
              isbst(left(t)) and
              allbigger(right(t),root(t)) and
              isbst(right(t)) )
}
```

Checking whether a Binary Tree is a BST

```
allsmaller(tree t, value v) {
    if ( isEmpty(t) )
        return true
    else
        return ( (root(t) < v) and
                 allsmaller(left(t),v) and
                 allsmaller(right(t),v) )
}

allbigger(tree t, value v) {
   if ( isEmpty(t) )
     return true
   else
     return ( (root(t) > v) and
              allbigger(left(t),v) and
              allbigger(right(t),v) )
}
```

Checking whether a Binary Tree is a BST

That was a simple algorithm, but inefficient. Exercise:

1. Count the number of comparisons that occur during the execution of this algorithm on a BST of size 7 and height 2
2. Repeat the exercise on a BST of size 15 and height 3.
3. What conclusions can you draw about the complexity of this algorithm?
4. A perfect Binary Tree is one where every internal node has two children and all the leaf nodes are at the same level (which means that the leaf nodes fill that level). What is the minimum number of comparisons, as a function of the size of the tree, that are truly necessary for any algorithm to check whether a perfect Binary Tree is a BST?
5. Figure out an algorithm that matches the best complexity possible for checking that a Binary Tree is a BST.

Sorting using BSTs

```
printInOrder(tree t) {
  if ( not isEmpty(t) ) {
    printInOrder(left(t))
    print(root(t))
    printInOrder(right(t))
  }
}

sort(array a of size n) {
  t = EmptyTree
  for i = 0,1,...,n−1
    t = insert(a[i],t)
  printInOrder(t)
}
```

Exercise: Modify the above code to put the values back in the array in order instead of printing them out. What is the complexity of sorting an array this way?