Binary Search Trees

Hussein Suleman <hussein@cs.uct.ac.za>

Department of Computer Science
School of IT
University of Cape Town

2020

Binary Search Trees

- Binary Tree with additional properties:
  - nodes can be compared by some value.
  - all nodes in the left subtree are < node.
  - all nodes in the right subtree are > node.
    - can also define one side as equal.

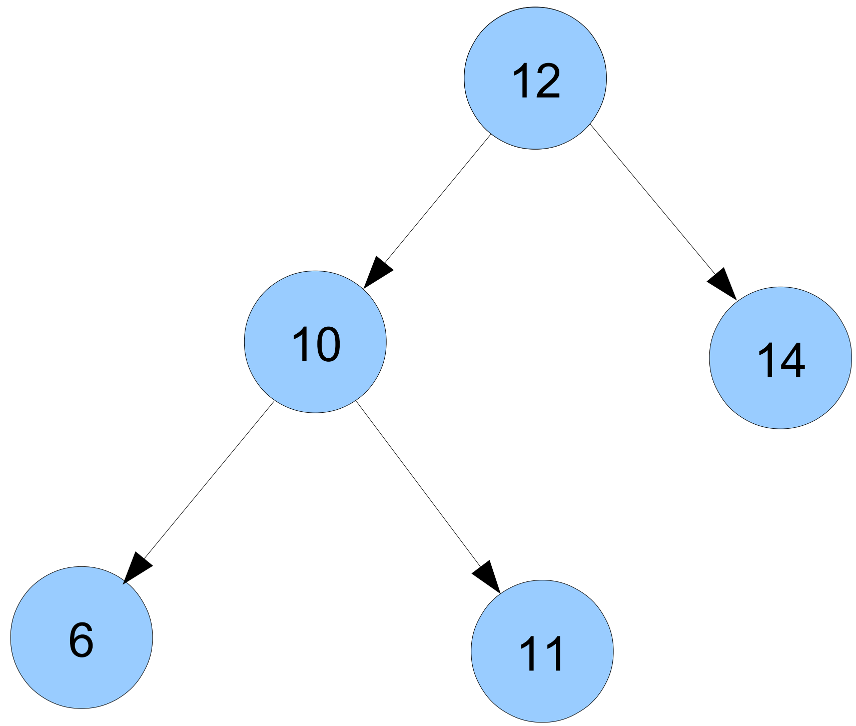
- Advantage: Faster search.
- Disadvantage: More work for adding/removing nodes.

Example Binary Search Tree

- {6, 10, 11} < 12
- 14 > 12

- 6 < 10
- 11 > 10

- InOrder traversal?



Basic Operations

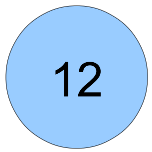
- insert
- find
- delete

insert Algorithm

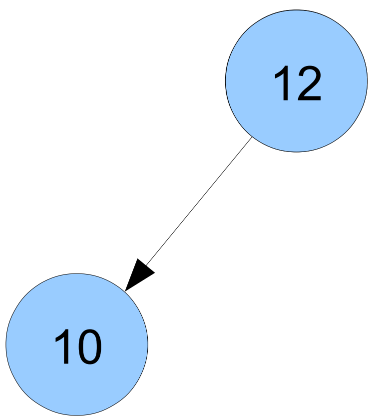
- If the root is null, the new node becomes the entire tree.
- If value<node, add new node to left.
- If value>node, add new node to right.
- Recurse until the node gets added as a new leaf node. Nodes are only added as new leaf nodes.

insert Example

- root = null, insert 12

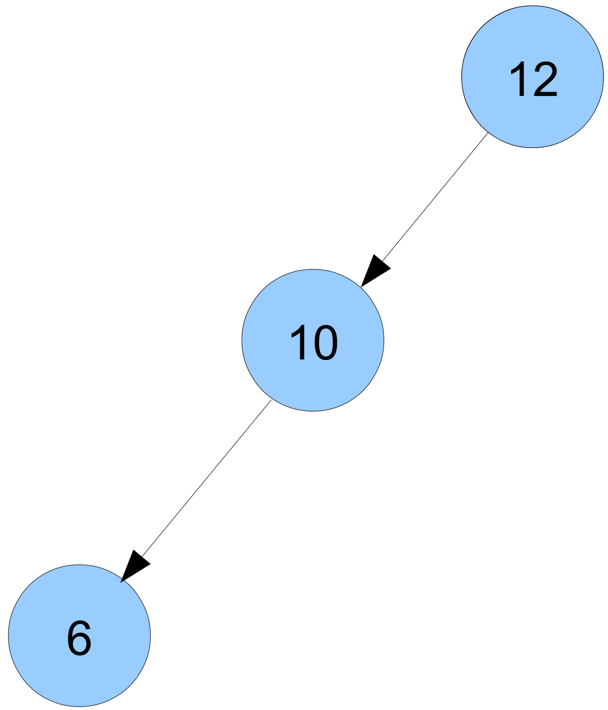


- insert 10



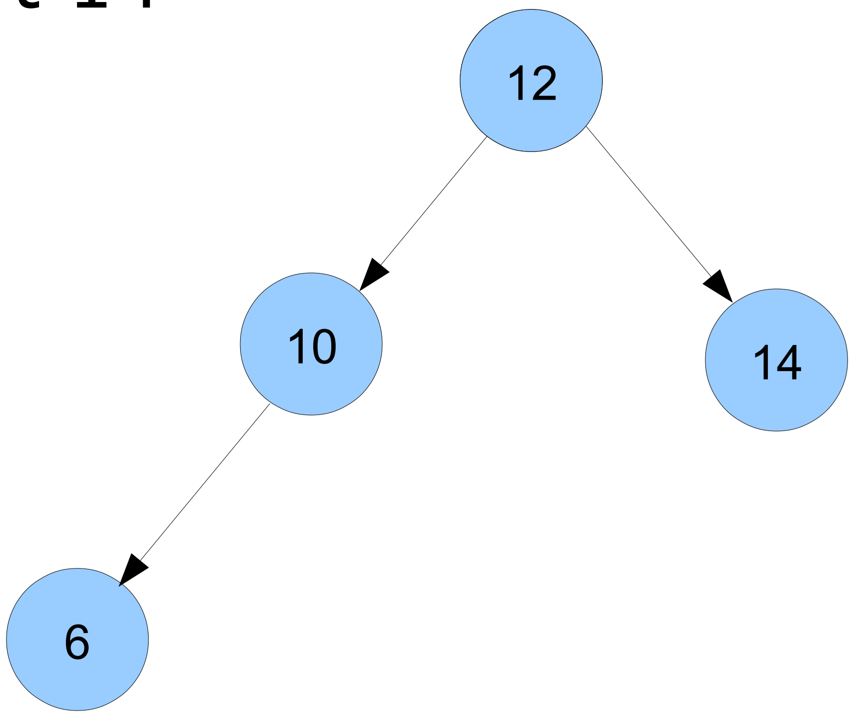
insert Example

- insert 6



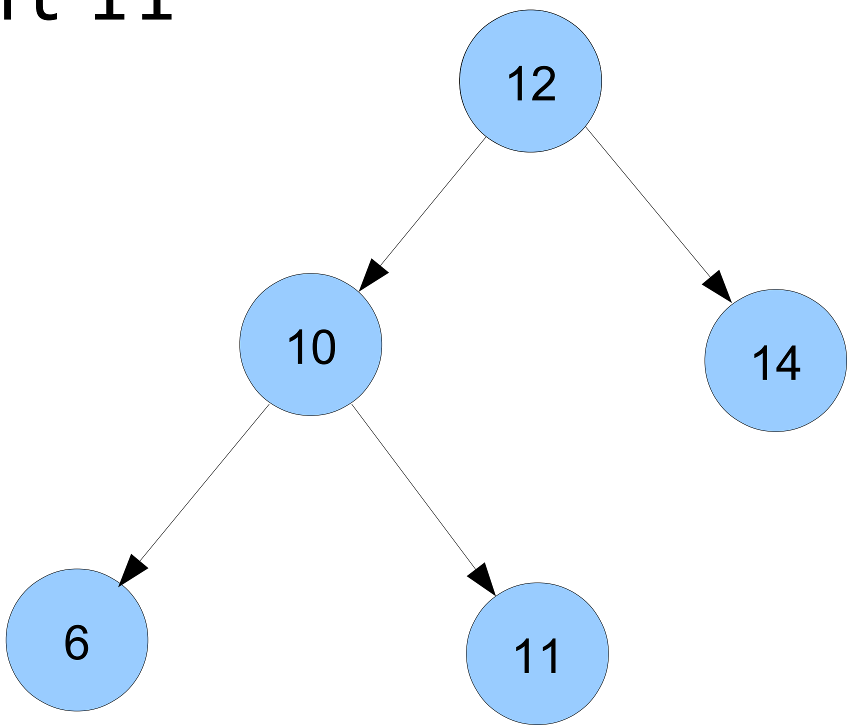
insert Example

- insert 14



insert Example

- insert 11



insert Code

```
public void insert ( dataType d )
{
   if (root == null)
      root = new BinaryTreeNode<dataType> (d, null, null);
   else
      insert (d, root);
}
```

insert Code

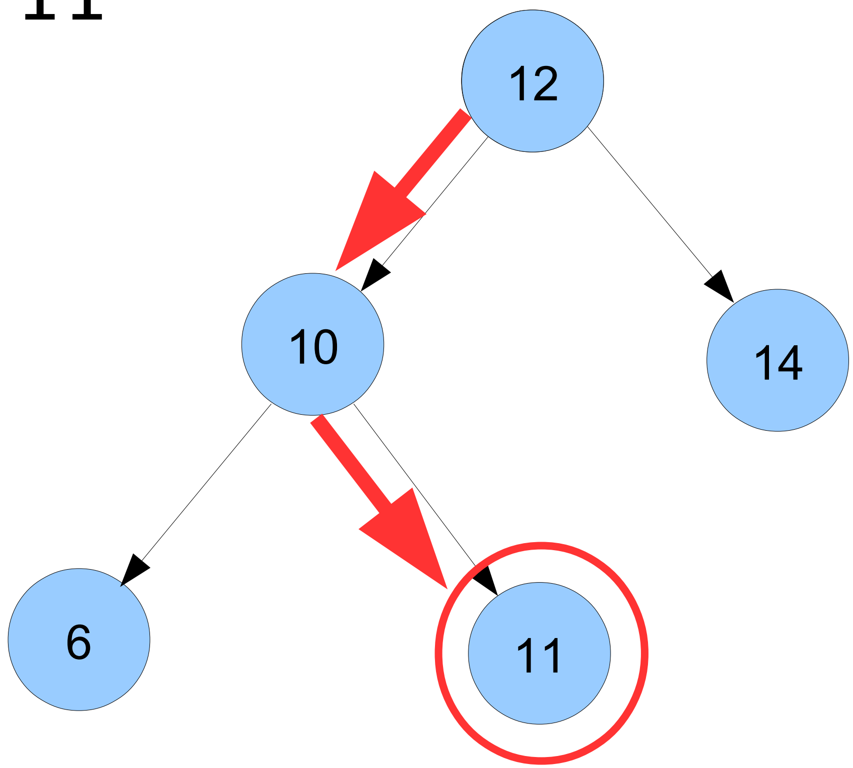
```
public void insert ( dataType d, BinaryTreeNode<dataType> node )
{
   if (d.compareTo (node.data) <= 0)
   {
      if (node.left == null)
         node.left = new BinaryTreeNode<dataType> (d, null, null);
      else
         insert (d, node.left);
   }
   else
   {
      if (node.right == null)
         node.right = new BinaryTreeNode<dataType> (d, null, null);
      else
         insert (d, node.right);
   }
}
```

find Algorithm

- start at root
- if search==root
  - result is root element
- if search<root
  - recurse into left sub-tree
- if search>root
  - recurse into right sub-tree

find Example

- find 11



find Code

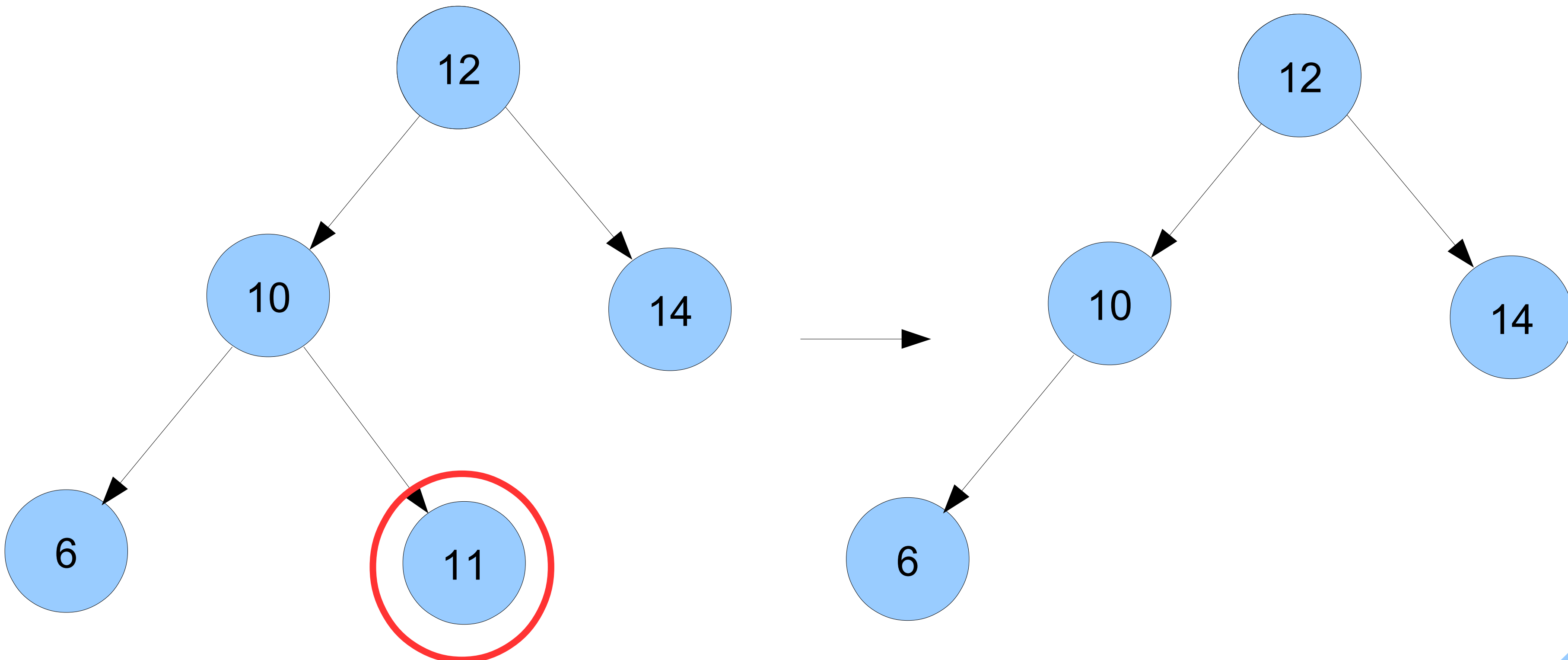
```
public BinaryTreeNode<dataType> find ( dataType d )
{
   if (root == null)
      return null;
   else
      return find (d, root);
}
public BinaryTreeNode<dataType> find ( dataType d, BinaryTreeNode<dataType> node )
{
   if (d.compareTo (node.data) == 0)
      return node;
   else if (d.compareTo (node.data) < 0)
      return (node.left == null) ? null : find (d, node.left);
   else
      return (node.right == null) ? null : find (d, node.right);
}
```

delete Algorithm

- find the element to delete
- if leaf, just delete it
- if one child, replace node with child
- if 2 children,
  - find X = minimum node in right subtree
  - remove X from right subtree
  - replace deleted element with X

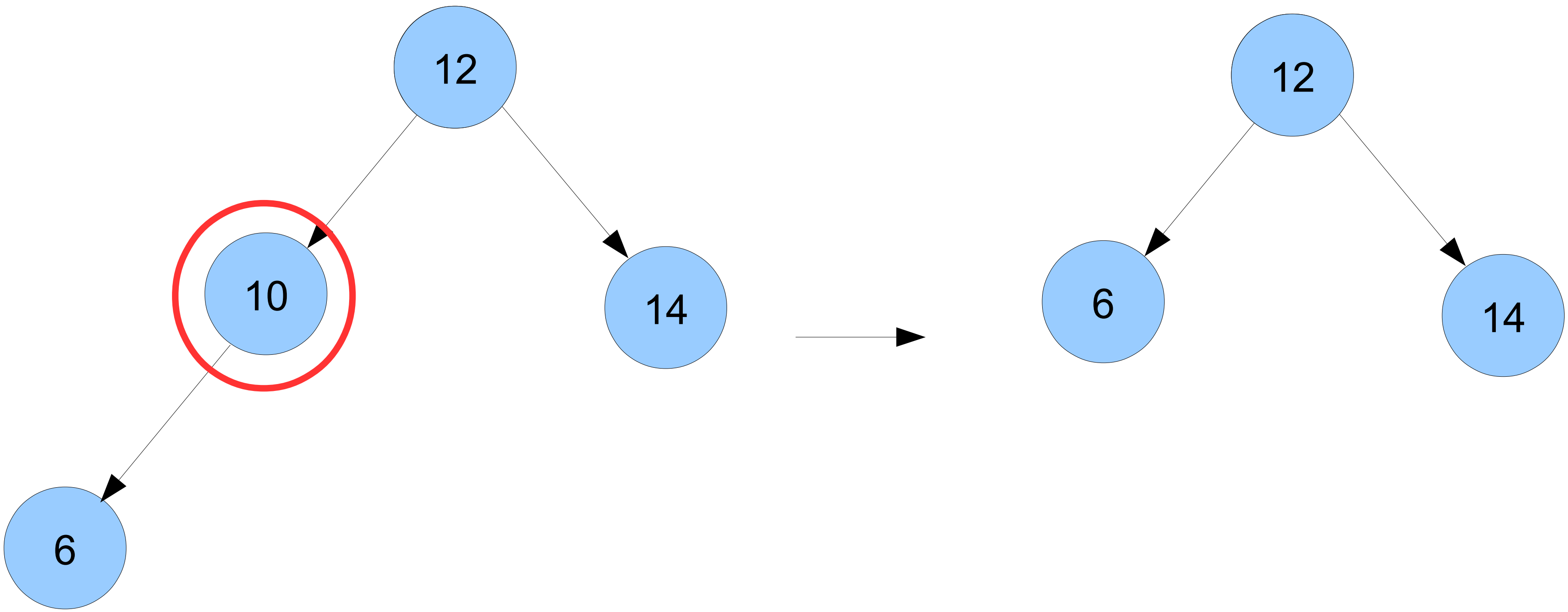
delete Example

- delete 11 - leaf node



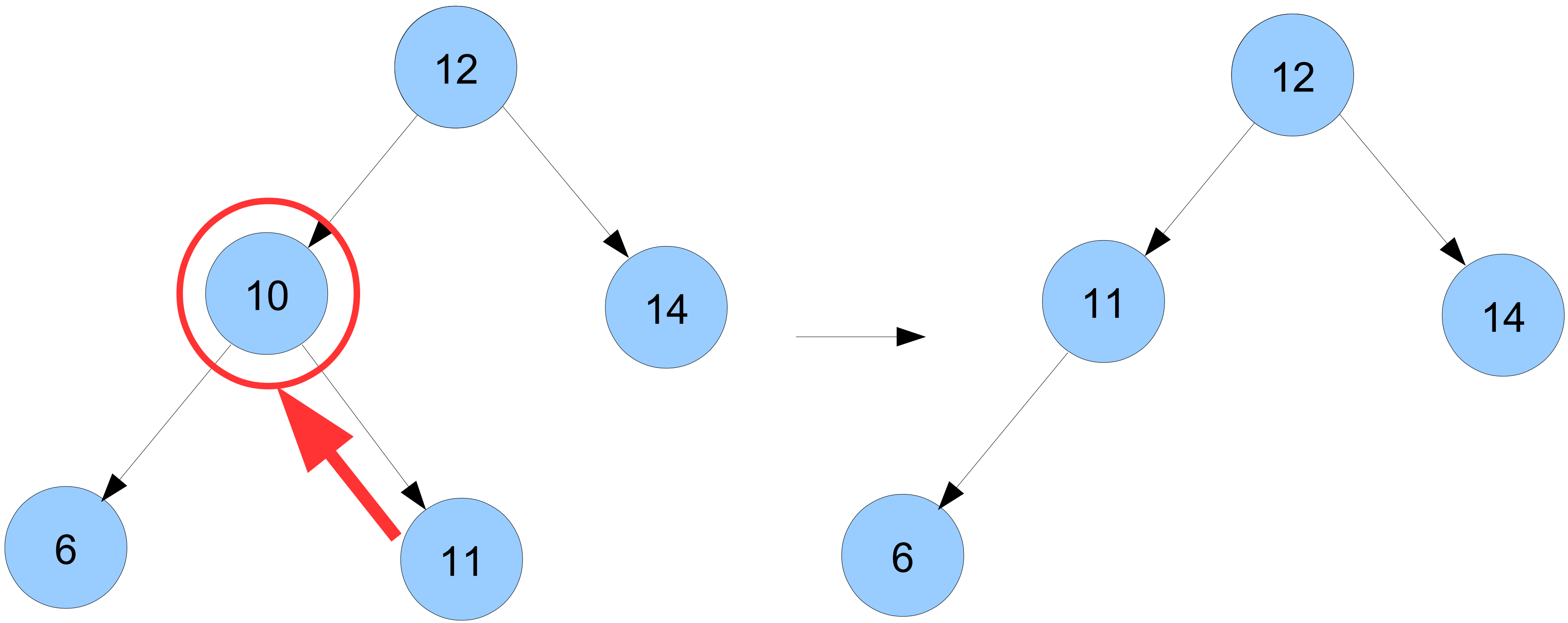
delete Example

- delete 10 - one child node



delete Example

- delete 10 - 2 child nodes



delete Code

```
public void delete ( dataType d )
{
   root = delete (d, root);
}
```

delete Code

```
public BinaryTreeNode<dataType> delete ( dataType d, BinaryTreeNode<dataType> node )
{
   if (node == null) return null;
   if (d.compareTo (node.data) < 0)
      node.left = delete (d, node.left);
   else if (d.compareTo (node.data) > 0)
      node.right = delete (d, node.right);
   else if (node.left != null && node.right != null )
   {
      node.data = findMin (node.right).data;
      node.right = removeMin (node.right);
   }
   else
      if (node.left != null)
         node = node.left;
      else
         node = node.right;
   return node;
}
```

delete Code

```
   public BinaryTreeNode<dataType> findMin ( BinaryTreeNode<dataType>
node )
   {
      if (node != null)
         while (node.left != null)
            node = node.left;
      return node;
   }
```

delete Code

```
   public BinaryTreeNode<dataType> removeMin ( BinaryTreeNode<dataType>
node )
   {
      if (node == null)
         return null;
      else if (node.left != null)
      {
         node.left = removeMin (node.left);
         return node;
      }
      else
         return node.right;
   }
```

How to Handle Duplicate Keys

- Assume all keys are unique!
- Issue:
  - Is it just identical item copies or different items with identical keys
- Options:
  - Duplicates in “adjacent” tree positions:
    - Must modify all operations to handle this
  - Counter for duplicates in each node
  - List of items attached to each node

Complexity Analysis

- Best Case:
  - search - O(1)
  - insert - O(1)
  - delete - O(1)
- Worst Case:
  - search - O(n)
  - insert - O(n)
  - delete - O(n)
  - *Why not O(log n)?*

Order Statistics

- Instead of finding minimum, what if we want Kth value?
- Can do this efficiently by maintaining size of subtree at each node.
- Every function that changes the tree needs to update these sizes.
  - insert, delete, removeMin

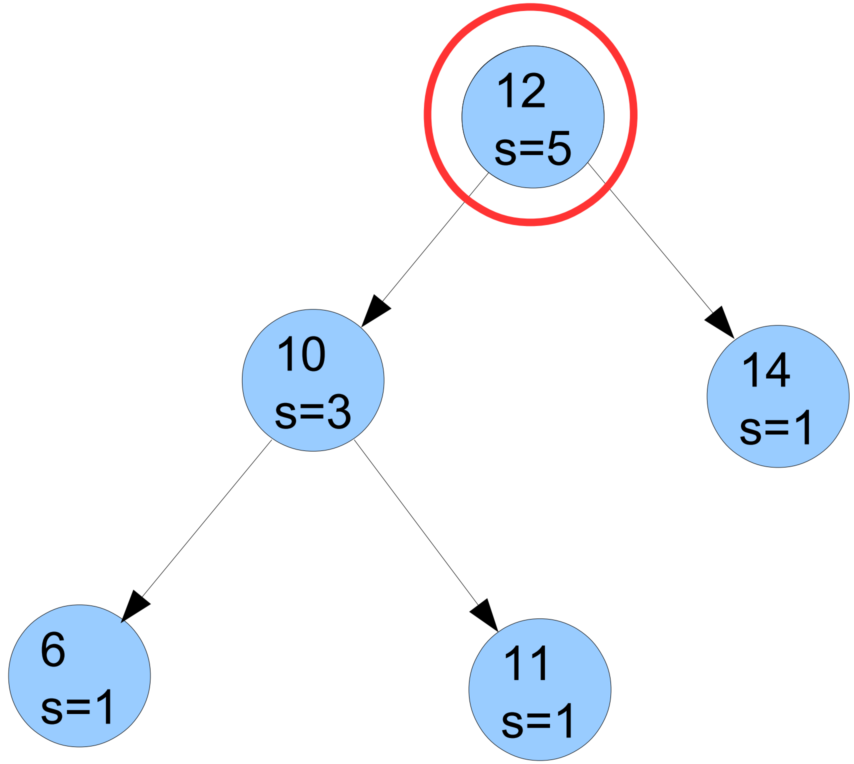
FindK Algorithm

- Suppose SL = size of left subtree
- Suppose SR = size of right subtree

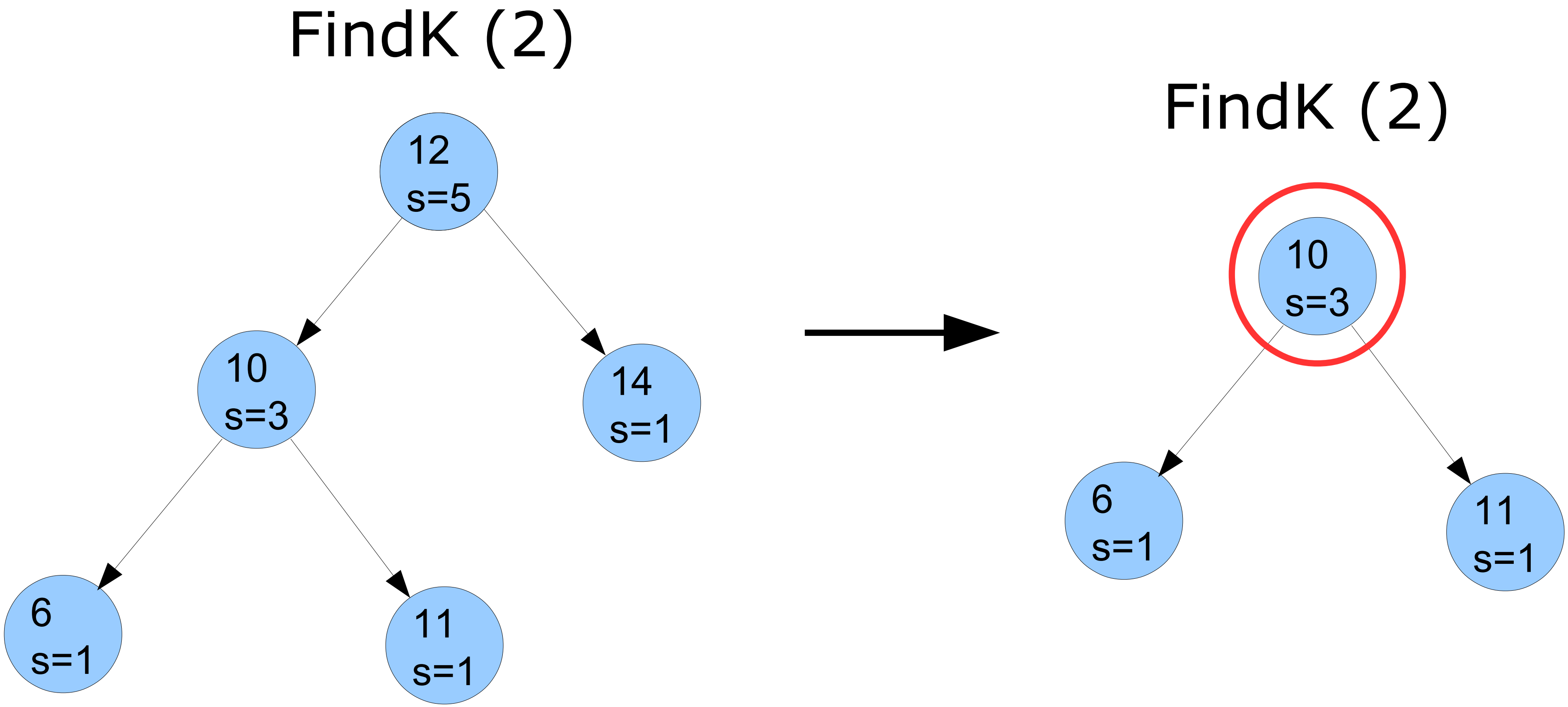
- If K = SL+1, root node is answer
- If K<SL+1, findK(k) in left subtree
- If K>SL+1, findK(K-SL-1) in right subtree

FindK Example 1

- FindK (4)



FindK Example 2



that's all folks!