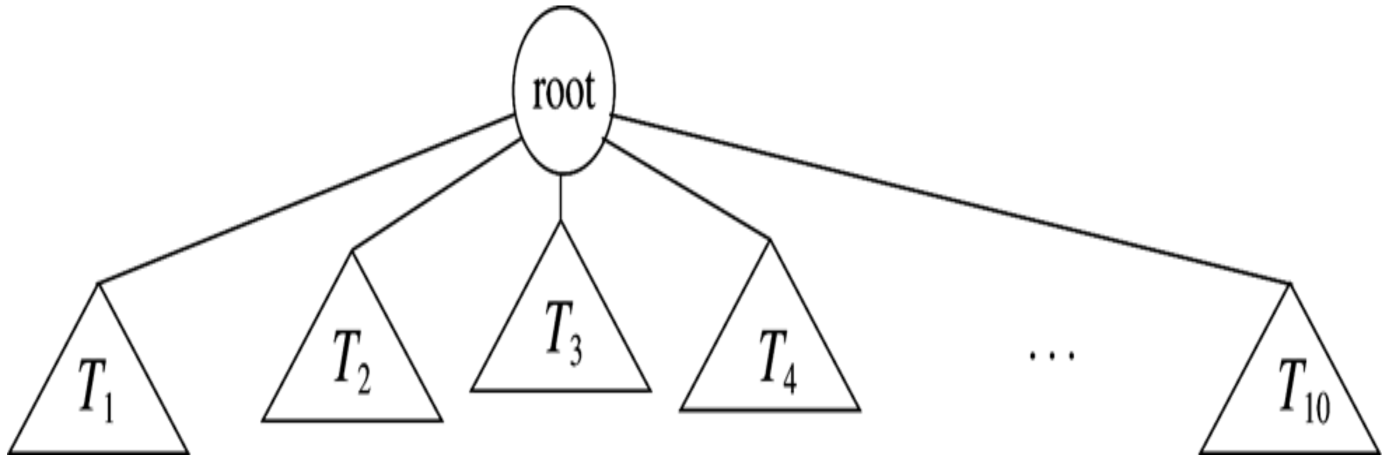
CMSC 341

Binary Search Trees

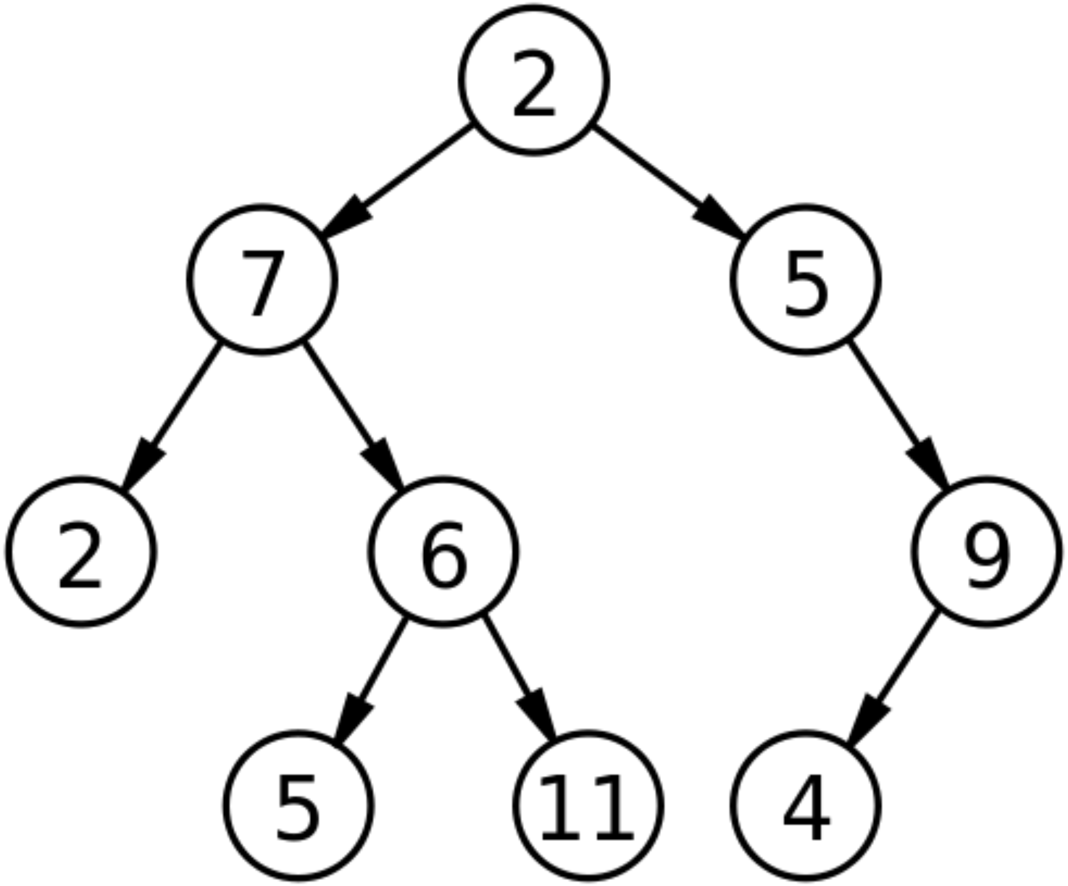
Announcements

- Homework #3 dues Thursday (10/5/2017)
- Exam #1 next Thursday (10/12/2017)

A Generic Tree



Binary Tree



The Binary Node Class

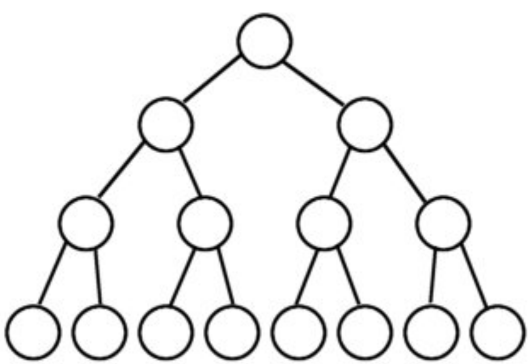
```
private class BinaryNode<AnyType>
{
    // Constructors
    BinaryNode( AnyType theElement )
    {
        this( theElement, null, null );
    }

    BinaryNode( AnyType theElement,
                BinaryNode<AnyType> lt, BinaryNode<AnyType> rt )
    {
        element  = theElement; left = lt; right = rt;
    }

    AnyType element;                  // The data in the node
    BinaryNode<AnyType> left;         // Left child reference
    BinaryNode<AnyType> right;        // Right child reference
}
```

Full Binary Tree

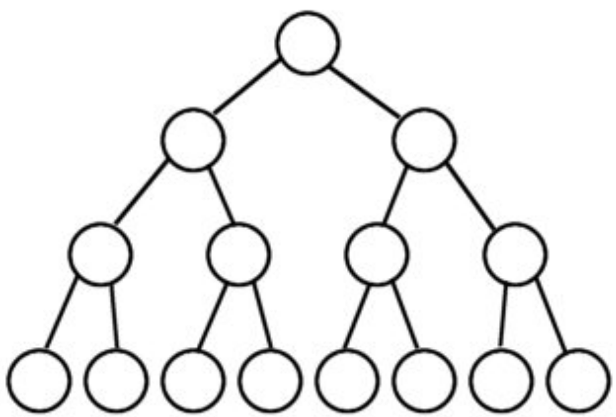
A full binary tree is a binary tree in which every node is a leaf or has exactly two children.



Theorem: A FBT with n internal nodes has n + 1 leaves (external nodes).

Perfect Binary Tree

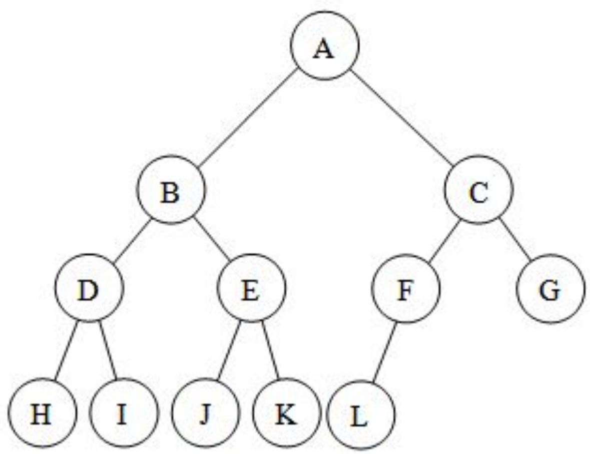
- A *Perfect Binary Tree* is a Full Binary Tree in which all leaves have the same depth.



Theorem:
The number of nodes in a PBT is 2^{h+1}-1, where h is height.

Complete Binary Tree

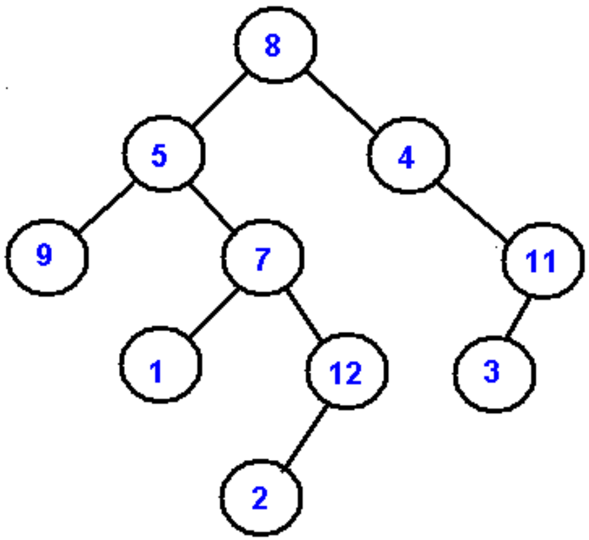
- A Complete Binary Tree is a binary tree in which every level is completed filled, except possibly the bottom level which is filled from left to right.



Tree Traversals

- Inorder (left, root, right)
- Preorder (root, left, right)
- Postorder (left, right, root)
- Levelorder (per-level)

Example



PreOrder - 8, 5, 9, 7, 1, 12, 2, 4, 11, 3

InOrder - 9, 5, 1, 7, 2, 12, 8, 4, 3, 11

PostOrder - 9, 1, 2, 12, 7, 5, 3, 11, 4, 8

LevelOrder - 8, 5, 4, 9, 7, 11, 1, 12, 3, 2

Binary Tree Construction

- Suppose that the elements in a binary tree are distinct.
- Can you construct the binary tree from which a given traversal sequence came?
- When a traversal sequence has more than one element, the binary tree is not uniquely defined.
- Therefore, the tree from which the sequence was obtained cannot be reconstructed uniquely.

Binary Tree Construction

Can you construct the binary tree, given two traversal sequences?

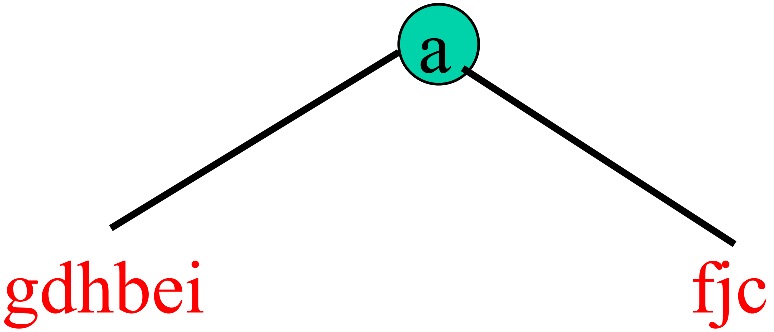
Depends on which two sequences are given.

Inorder And Preorder

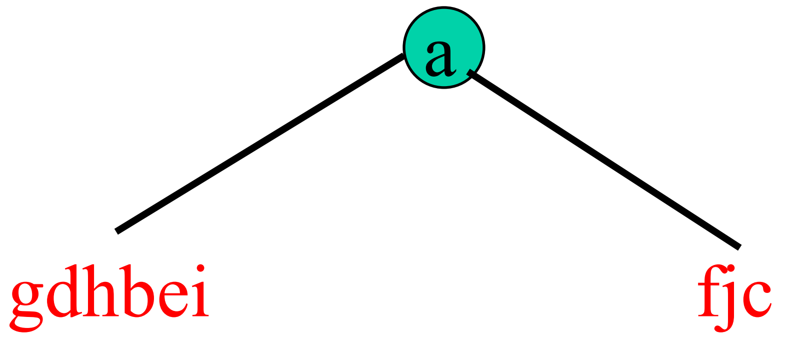
inorder = g d h b e i a f j c

preorder = a b d g h e i c f j

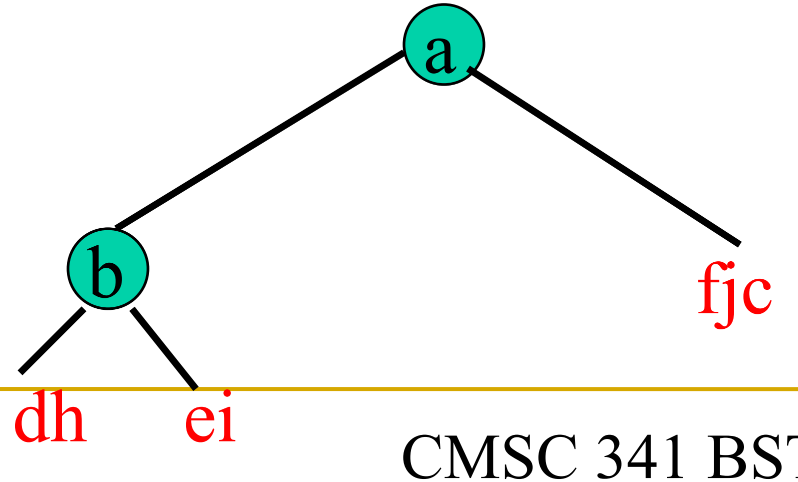
- Scan the preorder left to right using the inorder to separate left and right subtrees.
- a is the root of the tree; gdhbei are in the left subtree; fjc are in the right subtree.



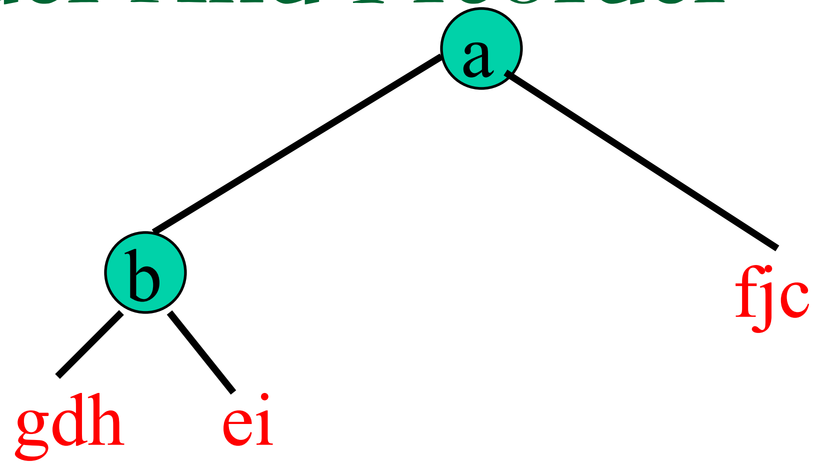
Inorder And Preorder



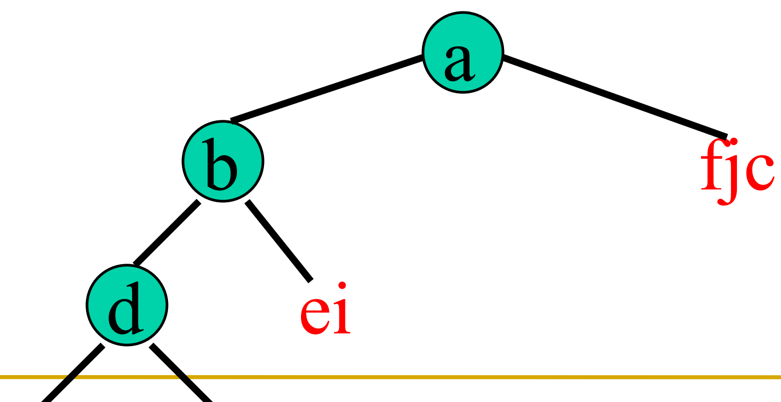
- preorder = b d g h e i c f j
- b is the next root; gdh are in the left subtree; ei are in the right subtree.



Inorder And Preorder



- preorder = a d g h e i c f j
- d is the next root; g is in the left subtree; h is in the right subtree.



Inorder And Postorder

- Scan postorder from right to left using inorder to separate left and right subtrees.
- inorder = g d h b e i a f j c
- postorder = g h d i e b j f c a
- Tree root is a; gdhbei are in left subtree; fjc are in right subtree.

Inorder And Level Order

- Scan level order from left to right using inorder to separate left and right subtrees.
- inorder = g d h b e i a f j c
- level order = a b c d e f g h i j
- Tree root is a; gdhbei are in left subtree; fjc are in right subtree.

Finding an element in a Binary Tree?

- Return a reference to node containing x, return null if x is not found

```
public BinaryNode<AnyType> find(AnyType x)
{
    return find(root, x);
}
private BinaryNode<AnyType> find( BinaryNode<AnyType> node, AnyType x)
{
   BinaryNode<AnyType> t = null;           // in case we don’t find it
   if ( node.element.equals(x) )           // found it here??
        return node;

   // not here, look in the left subtree
   if(node.left != null)
        t = find(node.left,x);

   // if not in the left subtree, look in the right subtree
   if ( t == null)
        t = find(node.right,x);

   // return reference, null if not found
   return t;
}
```

Is this a full binary tree?

```
boolean  isFBT (BinaryNode<AnyType> t)
{
   // base case – an empty tee is a FBT
   if (t == null) return true;

   // determine if this node is "full"
   // if just one child, return – the tree is not full
   if ((t.left == null && t.right != null)
   ||  (t.right == null && t.left != null))
        return false;

   // if this node is full, "ask" its subtrees if they are full
   // if both are FBTs, then the entire tree is an FBT
   // if either of the subtrees is not FBT, then the tree is not
   return isFBT( t.right ) && isFBT( t.left );

}
```

Other Recursive Binary Tree Functions

- Count number of interior nodes

```
int countInteriorNodes( BinaryNode<AnyType> t);
```

- Determine the height of a binary tree. By convention (and for ease of coding) the height of an empty tree is -1

```
int height( BinaryNode<AnyType> t);
```

- Many others

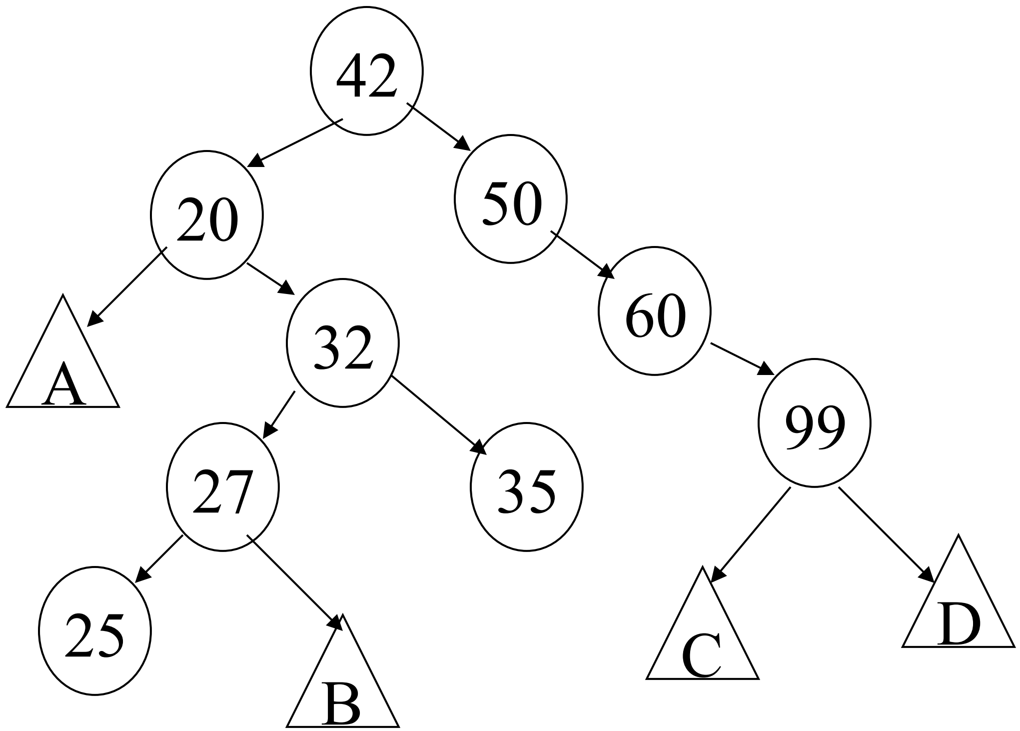
Other Binary Tree Operations

- How do we insert a new element into a binary tree?
- How do we remove an element from a binary tree?

Binary Search Tree

- **A *Binary Search Tree*** is a Binary Tree in which, at every node v, the values stored in the left subtree of v are less than the value at v and the values stored in the right subtree are greater.
- The elements in the BST must be comparable.
- Duplicates are not allowed in our discussion.
- Note that each subtree of a BST is also a BST.

A BST of integers



Describe the values which might appear in the subtrees labeled A, B, C, and D

BST Implementation

```
public class
BinarySearchTree<AnyType extends Comparable<? super AnyType>>
{
    private static class BinaryNode<AnyType>
   {
          // Constructors
          BinaryNode( AnyType theElement )
          { this( theElement, null, null ); }

          BinaryNode( AnyType theElement,
               BinaryNode<AnyType> lt, BinaryNode<AnyType> rt )
          { element  = theElement; left = lt; right = rt; }

          AnyType element;            // The data in the node
          BinaryNode<AnyType> left;   // Left child reference
          BinaryNode<AnyType> right;  // Right child reference
    }
```

BST Implementation (2)

```
private BinaryNode<AnyType> root;

public BinarySearchTree( )
{
    root = null;
}

public void makeEmpty( )
{
    root = null;
}

public boolean isEmpty( )
{
    return root == null;
}
```

BST "contains" Method

```
 public boolean contains( AnyType x )
 {
    return contains( x, root );
 }

private boolean contains( AnyType x, BinaryNode<AnyType> t )
{
    if( t == null )
        return false;

    int compareResult = x.compareTo( t.element );

    if( compareResult < 0 )
        return contains( x, t.left );
    else if( compareResult > 0 )
        return contains( x, t.right );
    else
        return true;    // Match
}
```

Performance of “contains”

- Searching in randomly built BST is *O*(lg n) on average
  - but generally, a BST is not randomly built
- Asymptotic performance is *O*(height) in all cases

Implementation of printTree

```
public void printTree()
{
    printTree(root);
}

private void printTree( BinaryNode<AnyType> t )
{
        if( t != null )
        {
            printTree( t.left );
            System.out.println( t.element );
            printTree( t.right );
        }
}
```

BST Implementation (3)

```
public AnyType findMin( )
{
    if( isEmpty( ) ) throw new UnderflowException( );
            return findMin( root ).element;
}
public AnyType findMax( )
{
    if( isEmpty( ) ) throw new UnderflowException( );
            return findMax( root ).element;
}
public void insert( AnyType x )
{
    root = insert( x, root );
}
public void remove( AnyType x )
{
    root = remove( x, root );
}
```

The insert Operation

```
private BinaryNode<AnyType>
insert( AnyType x,  BinaryNode<AnyType> t )
 {
     if( t == null )
         return new BinaryNode<AnyType>( x, null, null );

     int compareResult = x.compareTo( t.element );

     if( compareResult < 0 )
         t.left = insert( x, t.left );
     else if( compareResult > 0 )
         t.right = insert( x, t.right );
     else
         ;  // Duplicate; do nothing
     return t;
 }
```

The remove Operation

```
private BinaryNode<AnyType>
remove( AnyType x,  BinaryNode<AnyType> t )
{
  if( t == null )
       return t;   // Item not found; do nothing
  int compareResult = x.compareTo( t.element );
  if( compareResult < 0 )
       t.left = remove( x, t.left );
  else if( compareResult > 0 )
       t.right = remove( x, t.right );
  else if( t.left != null && t.right != null ){ // 2 children
       t.element = findMin( t.right ).element;
       t.right = remove( t.element, t.right );
  }
  else  // one child or leaf
       t = ( t.left != null ) ? t.left : t.right;
  return t;
}
```

Implementations of find Max and Min

```
private BinaryNode<AnyType> findMin( BinaryNode<AnyType> t )
{
    if( t == null )
       return null;
    else if( t.left == null )
        return t;
    return findMin( t.left );
}

private BinaryNode<AnyType> findMax( BinaryNode<AnyType> t )
{
    if( t != null )
        while( t.right != null )
            t = t.right;

    return t;
}
```

Remove a node

Constraint: delete a node, maintain BST property

1. Deleting a leaf. Easy. Just do it. BST property is not affected.

2. Deleting a non-leaf node v
   a. v has no left child -- replace v by its right child
   b. v has no right child -- replace v by its left child
   c. v has both left and right children, either:
      1. Replace data in v by data of predecessor and delete predecessor
      2. Replace data in v by data in successor and delete successor

Performance of BST methods

- What is the asymptotic performance of each of the BST methods?

	Best Case	Worst Case	Average Case
contains			
insert			
remove			
findMin/ Max			
makeEmpty			

Predecessor in BST

- Predecessor of a node v in a BST is the node that holds the data value that immediately precedes the data at v in order.

- Finding predecessor
  - v has a left subtree
    - then predecessor must be the largest value in the left subtree (the rightmost node in the left subtree)
  - v does not have a left subtree
    - predecessor is the first node on path back to root that does not have v in its left subtree

Successor in BST

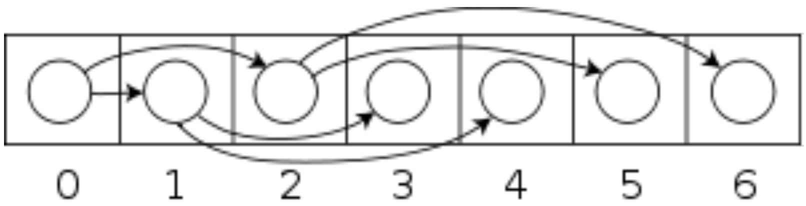
- Successor of a node v in a BST is the node that holds the data value that immediately follows the data at v in order.

- Finding Successor
  - v has right subtree
    - successor is smallest value in right subtree (the leftmost node in the right subtree)
  - v does not have right subtree
    - successor is first node on path back to root that does not have v in its right subtree

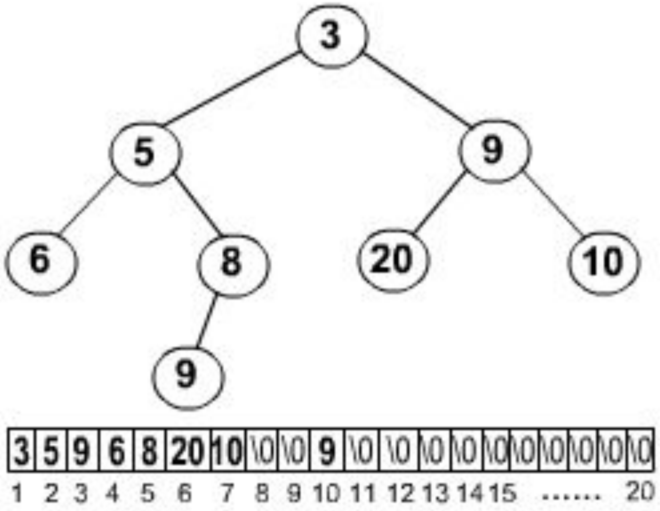
Tree Iterators

- As we know there are several ways to traverse through a BST. For the user to do so, we must supply different kind of iterators. The iterator type defines how the elements are traversed.
  - `InOrderIterator<T>` **`inOrderIterator`**`();`
  - `PreOrderIterator<T>` **`preOrderIterator`**`();`
  - `PostOrderIterator<T>` **`postOrderIterator`**`();`
  - `LevelOrderIterator<T>` **`levelOrderIterator`**`();`

BST as Arrays



Parent: $\left\lfloor \frac{i-1}{2} \right\rfloor$



Binary Search Tree Operations Review

Basic BST Operations

- (BST Setup) → set up a BST
- (Node Setup) → set up a BST Node
- **void insert(x)** → insert x into the BST
- **void remove(x)** → remove x from the BST
- **<type> findMin()** → find min value in the BST
- **<type> findMax()** → find max value in the BST
- **boolean contains(x)** → is x in the BST?
- **boolean isEmpty()** → is the BST empty?
- **void makeEmtpy()** → make the BST empty
- **void PrintTree()** → print the BST

Public and Private Functions

- Many of the operations we want to use will have two (overloaded) versions
- Public function takes in zero or one arguments
  - Calls the private function
- Private function takes in one or two arguments
  - Additional argument is the “root” of the subtree
  - Private function recursively calls itself
    - Changes the “root” each time to go further down the tree

Insert

```
void insert( x )
```

Inserting a Node

- Insertion will always create a new leaf node

- In determining what to do, there are 4 choices
  - Go down the <u>left</u> subtree (visit the left child)
    - Value we want to insert is smaller than current
  - Go down the <u>right</u> subtree (visit the right child)
    - Value we want to insert is greater than current
  - <u>Insert</u> the node at the current spot
    - The current “node” is NULL (we’ve reached a leaf)
  - Do <u>nothing</u> (if we’ve found a duplicate)

Insert Functions

- Two versions of insert
  - Public version (one argument)
  - Private version (two arguments, recursive)
- Public version immediately calls private one

```
void insert( const Comparable & x )
{
    // calls the overloaded private insert()
    insert( x, root );
}
```

Starting at the Root of a (Sub)tree

- First check if the "root" of the tree is NULL
  - If it is, create and insert the new node
  - Send left and right children to NULL

```
// overloaded function that allows recursive calls
void insert( const Comparable & x, BinaryNode * & t )
{
    if( t == NULL ) // no node here (make a leaf)
        t = new BinaryNode( x, NULL, NULL );
    // rest of function…
}
```

Insert New Node (Left or Right)

- If the "root" we have is not NULL
  - Traverse down another level via its children
  - Call **`insert()`** with new sub-root (recursive)

```
// value in CURRENT root 't' < new value
else if( x < t->element ) {
    insert( x, t->left ); }

// value in CURRENT root 't' > new value
else if( t->element < x ) {
    insert( x, t->right ); }

else;  // Duplicate; do nothing
```

Full Insert() Function

- Remember, this function is recursive!

```
// overloaded function that allows recursive calls
void insert( const Comparable & x, BinaryNode * & t )
{
    if( t == NULL ) // no node here (make a new leaf)
        t = new BinaryNode( x, NULL, NULL );

   // value in CURRENT root 't' < new value
    else if( x < t->element ) { insert( x, t->left ); }

    // value in CURRENT root 't' > new value
    else if( t->element < x ) { insert( x, t->right ); }

    else;  // Duplicate; do nothing
}
```

What's Up With `BinaryNode * & t`?

- The code " `* & t` " is a reference to a pointer
- Remember that passing a reference allows us to change the value of a variable in a function
  - And have that change "stick" outside the function
- When we pass a variable, we pass its value
  - It just so happens that a pointer's "value" is the address of something else in memory

Find Minimum

```
Comparable findMin( )
```

Finding the Minimum

- What do we do?
  - Go all the way down to the left

```
Comparable findMin(BinaryNode *t )
{
    // empty tree
    if (t == NULL) { return NULL; }

    // no further nodes to the left
     if (t->left == NULL) {
        return node->value;    }
    else {
        return findMin(t->left);    }
}
```

Find Maximum

```
Comparable findMax( )
```

Finding the Maximum

- What do we do?
  - Go all the way down to the right

```
Comparable findMax(BinaryNode *t )
{
    // empty tree
    if (t == NULL) { return NULL; }

    // no further nodes to the right
     if (t->right == NULL) {
         return node->value;    }
    else {
         return findMin(t->right);    }
}
```

Recursive Finding of Min/Max

- Just like **`insert()`** and other functions, **`findMin()`** and **`findMax()`** have 2 versions

- Public (no arguments):
  - **`Comparable findMin( );`**
  - **`Comparable findMax( );`**
- Private (one argument):
  - **`Comparable findMax (BinaryNode *t);`**
  - **`Comparable findMax (BinaryNode *t);`**

Delete the Entire Tree

```
void makeEmpty ( )
```

Memory Management

- Remember, we don't want to lose any memory by freeing things out of order!
  - Nodes to be carefully deleted

- BST nodes are only deleted when
  - A single node is removed
  - We are finished with the entire tree
    - Call the destructor

Destructor

- The destructor for the tree simply calls the **`makeEmpty()`** function

```
// destructor for the tree
~BinarySearchTree( )
{
    // we call a separate function
    // so that we can use recursion
    makeEmpty( root );
}
```

Make Empty

- A recursive call will make sure we hang onto each node until its children are deleted

```
void makeEmpty( BinaryNode * & t )
{
    if( t != NULL )
    {
        // delete both children, then t
        makeEmpty( t->left );
        makeEmpty( t->right );
        delete t;
        // set t to NULL after deletion
        t = NULL;
    }
}
```

Find a Specific Value

```
boolean contains( x )
```

Finding a Node

- Only want to know <u>if</u> it's in the tree, not <u>where</u>
  - Use recursion to traverse the tree

```
bool contains( const Comparable & x ) const  {
    return contains( x, root ); }

bool contains( const Comparable & x, BinaryNode *t ) const
{
    if( t == NULL ) { return false; }
    // our value is lower than the current node's
    else if( x < t->element ) { return contains( x, t->left ); }
    // our value is higher than the current node's
    else if( t->element < x ) { return contains( x, t->right );}
    else { return true; }   // Match
 }
```

Finding a Node

- Only want to know if it's in the tree, not where
  - Use recursion to traverse the tree

```
bool contains( const Comparable & x ) const  {
    return contains( x, root ); }

bool contains( const Comparable & x, BinaryNode *t ) const
{
    if( t == NULL ) { return
    // our value is lower th
    else if( x < t->element                           ->left ); }
    // our value is higher than the current node's
    else if( t->element < x ) {
    else { return true; }   //
 }
```

We have to have a defined overloaded comparison operator for this to work!

(Both of the **else if** statements use **<** so we only need to write one)

Removing a Node

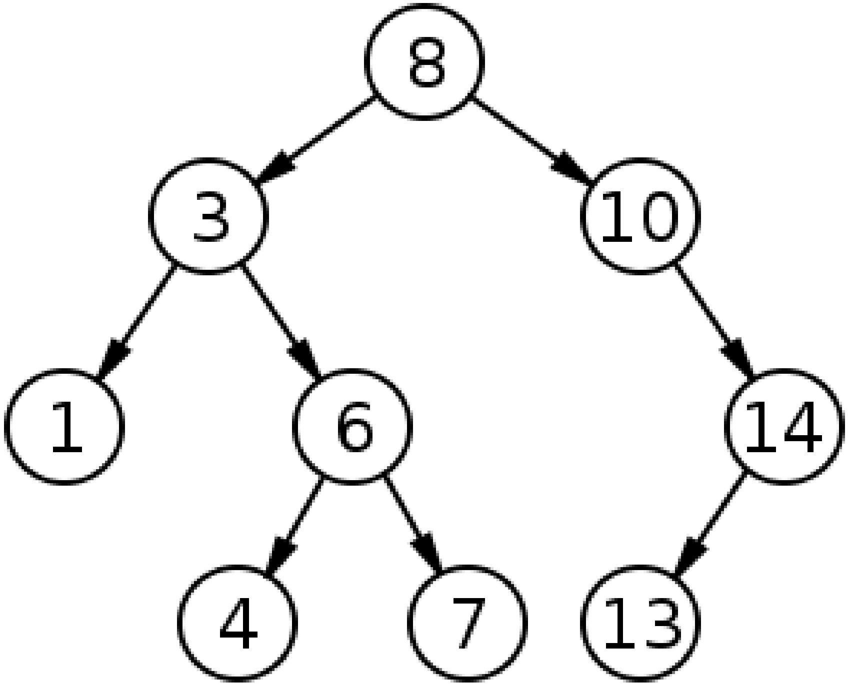
```
void remove( x )
```

Complicated Removal

- Similar to a linked list, removal is often much more complicated than insertion or complete deletion
- We must first traverse the tree to find the target we want to remove
  - If we “disconnect” a link, we need to reestablish
- Possible scenarios
  - No children (leaf)
  - One child
  - Two children

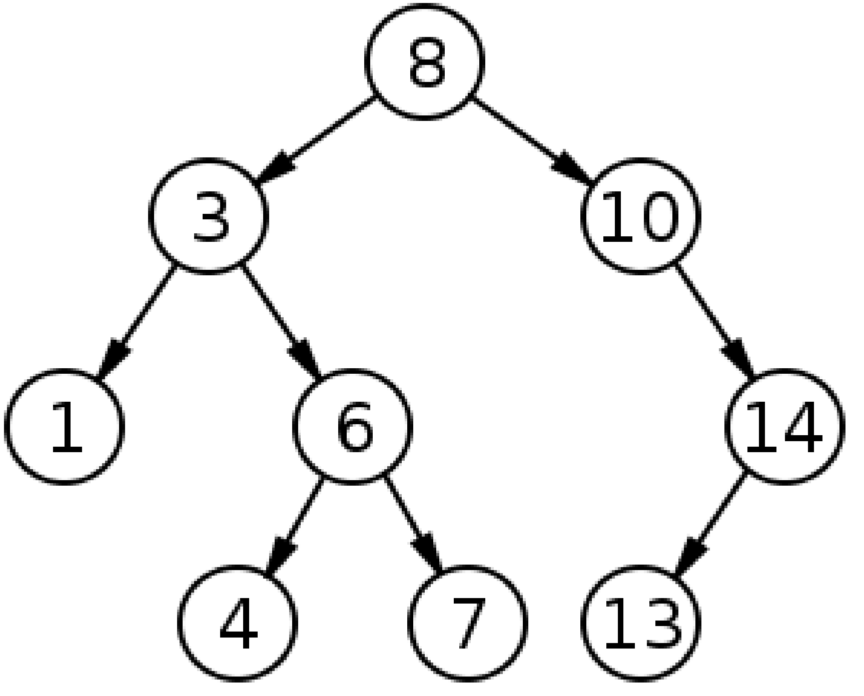
Removing A Node – Example 1

- Remove 4
- Any issues?



Removing A Node – Example 2

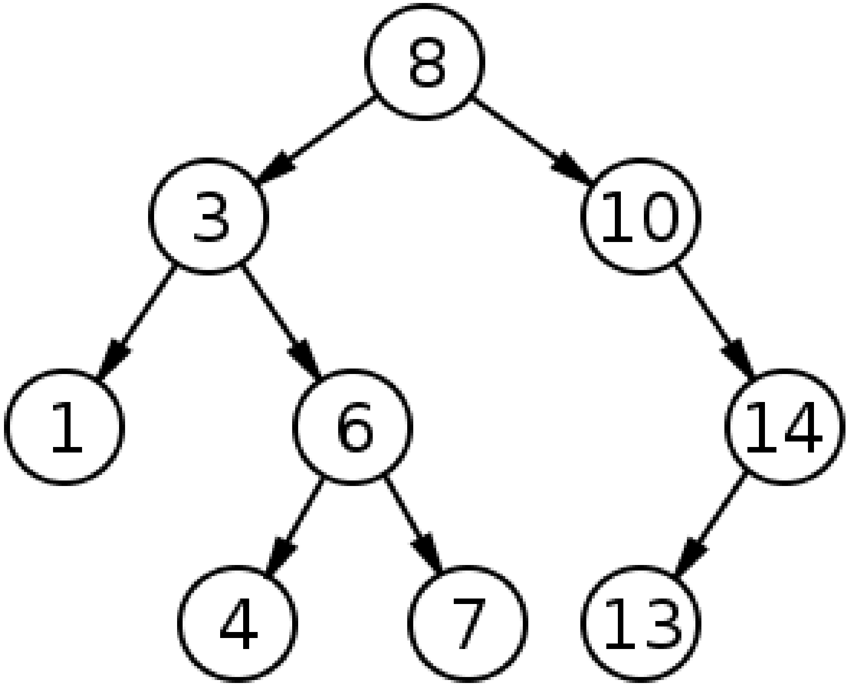
- Remove 6
- Any issues?



Removing A Node – Example 3

- Remove 8

- Any issues?



Removing a Node – No Children

- Simplest scenario for removal
  - No children to worry about managing
- Reminder: nodes with no children are leaves

- We still have to find the target node first
- To remove a node with no children, we need to do the following:
  - Cut the link from the parent node
  - Free the memory

Removing a Node – One Child

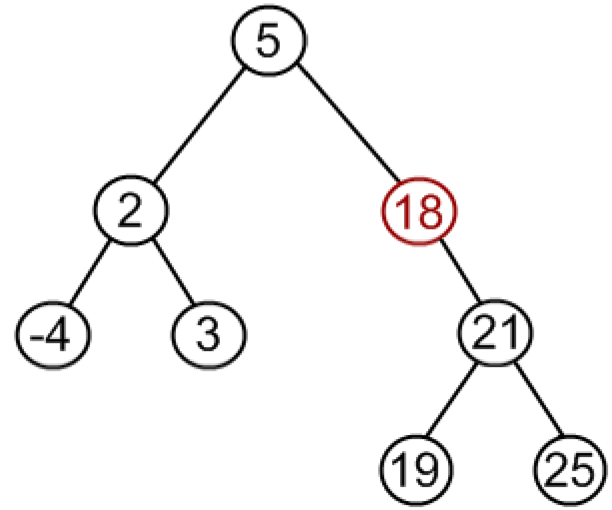
- Second easiest scenario for removal
  - Only one child is linked to the node

- The node can only be deleted after its parent adjusts the link to bypass the node to the child
  - The "grandparent" node takes custody

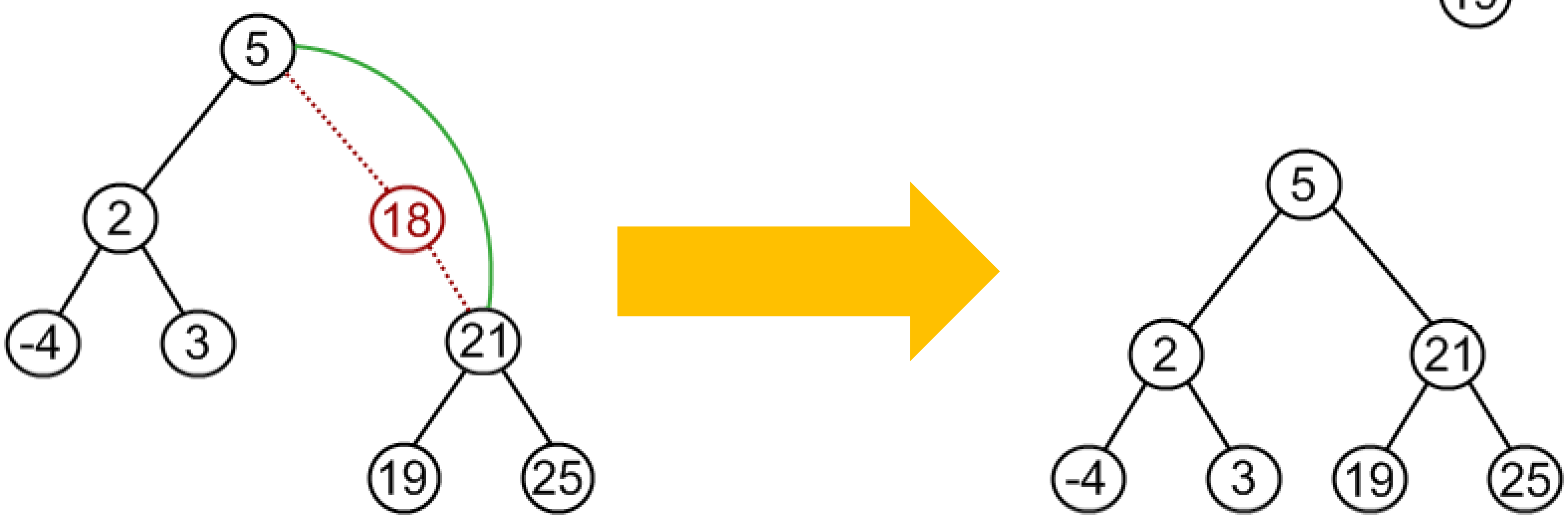
- To remove a node with one child, we need to do the following:
  - Connect node's parent to its child (custody)
  - Free the memory

Example Removal – One Child

- Remove “18” from this BST:



- Grandparent takes custody



Source: http://www.algolist.net/Data_structures/Binary_search_tree/Removal

Code for Removal

```
void remove( const Comparable & x, BinaryNode * & t )
{
    // code to handle two children prior to this

    else
    {
        // "hold" the position of node we'll delete
        BinaryNode *oldNode = t;

        // ternary operator
        t = ( t->left != NULL ) ? t->left : t->right;
        delete oldNode;
    }
}
```

Ternary Operator – Removal Code

- The ternary operator code for removal

```
// ternary operator
t = ( t->left != NULL ) ? t->left : t->right;
```

- Can also be expressed as

```
// if the left child isn't NULL
if ( t->left != NULL) {
    t = t->left;          // replace t with left child
} else {
    t = t->right;         // else replace with right child
}
```

Actually the same code works for one or zero children! Why?

Removing a Node – One Child

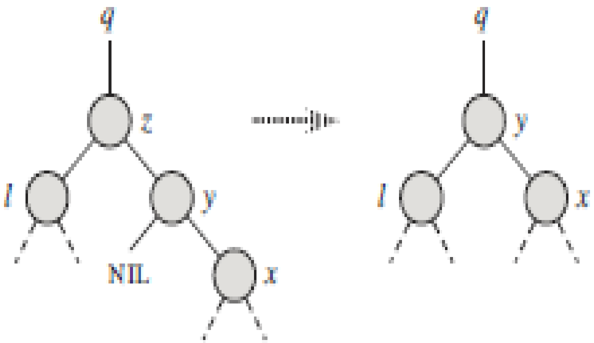
- Second easiest scenario for removal
  - Only one child is linked to the node
- The node can only be deleted after its parent adjusts the link to bypass the node to the child
  - The "grandparent" node takes custody
- To remove a node with one child, we need to do the following:
  - Connect node's parent to its child (custody)
  - Free the memory

Removing a Node – Two Children

- Most difficult scenario for removal
  - Everyone in the subtree will be affected

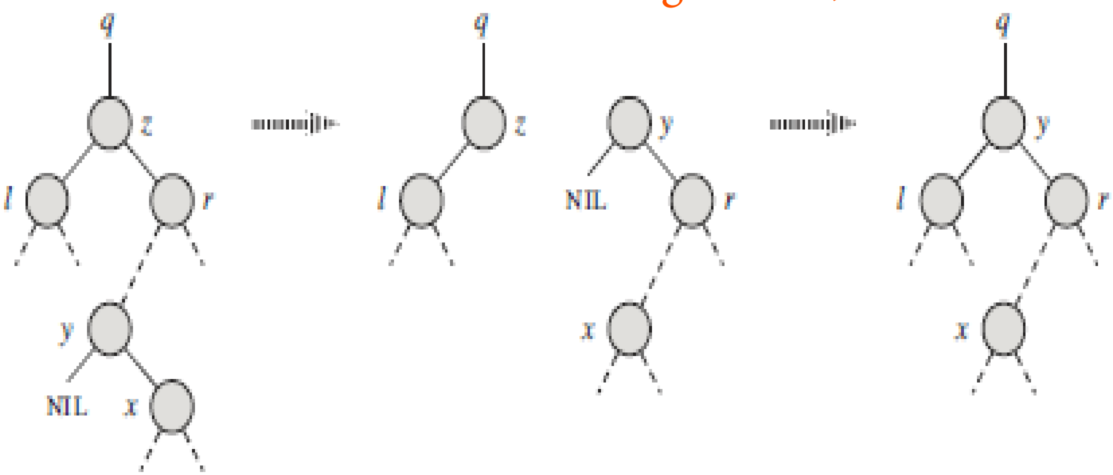
- Instead of completely deleting the node, we will replace its value with another node's
  - The smallest value in the right subtree
  - Use `findMin()` to locate this value
  - Then delete the node whose value we moved

Removing a Node – Two Children



We find z's successor y,which lies in z's right subtree and has no left child. We want to splice y out of its current location and have it replace z in the tree. If y is z's right child then we replace z by y, leaving y's right child alone.

Otherwise, y lies within z's right subtree but is not z's right child. In this case, we first replace y by its own right child, and then we replace z by y.



Remove node z

Remove Function

```
void remove( const Comparable & x, BinaryNode * & t )
{
    if( t == NULL ) { return; }  // item not found; do nothing

    // continue to traverse until we find the element
    if( x < t->element ) { remove( x, t->left ); }
    else if( t->element < x ) { remove( x, t->right ); }

    else if( t->left != NULL && t->right != NULL ) // two children
    {
        // find right's lowest value
        t->element = findMin( t->right )->element;
        // now delete that found value
        remove( t->element, t->right );
     }
     else // zero or one child
     {
          BinaryNode *oldNode = t;
          // ternary operator
          t = ( t->left != NULL ) ? t->left : t->right;
          delete oldNode;
     }
}
```

Printing a Tree

```
void printTree( )
```

Printing a Tree

- Printing is simple – only question is which order we want to traverse the tree in?

```
// ostream &out is the stream we want to print to
// (it maybe cout, it may be a file - our choice)
void printTree( BinaryNode *t, ostream & out ) const
{
    // if the node isn't null
    if( t != NULL )
    {
        // print an in-order traversal
        printTree( t->left, out );
        out << t->element << endl;
        printTree( t->right, out );
    }
}
```

Performance
Run Time of BST Operations

Big O of BST Operations

Operation	Big O
`contains( x )`	*O(log n)*
`insert( x )`	*O(log n)*
`remove( x )`	*O(log n)*
`findMin/findMax( x )`	*O(log n)*
`isEmpty( )`	*O(1)*
`printTree( )`	*O(n)*