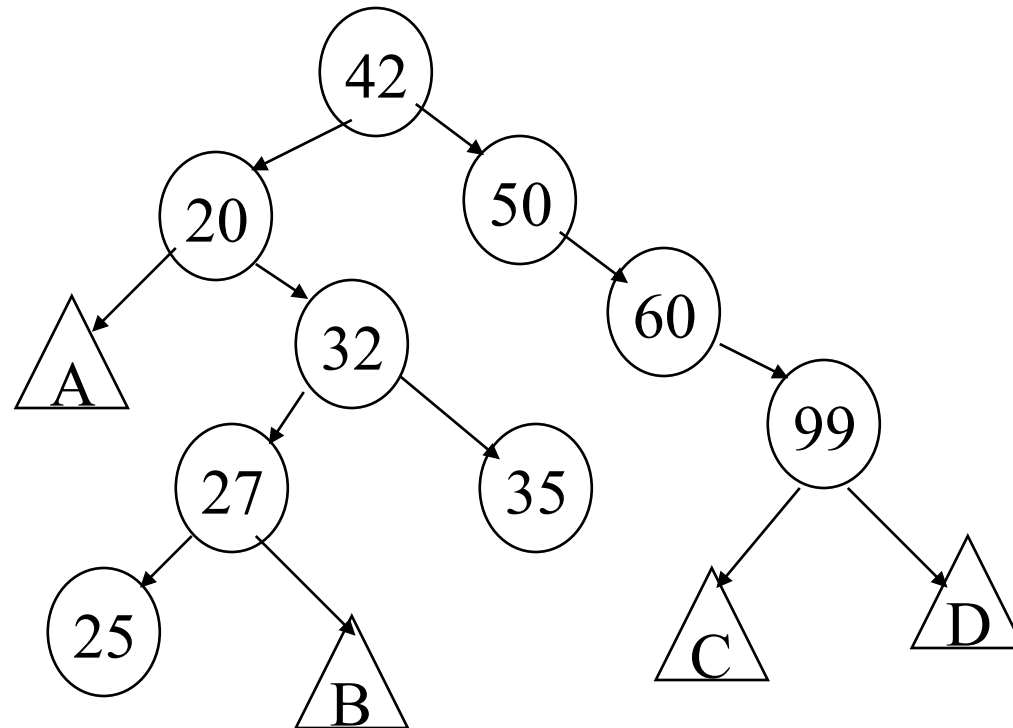# CMSC 341

Binary Search Trees

Textbook section: 4.3

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

SearchTree ADT

- ## The SearchTree ADT

  - A *search tree* is a binary search tree which stores homogeneous elements with no duplicates.

  - It is dynamic.

  - The elements are ordered in the following ways

    - inorder -- as dictated by compareTo( )

    - preorder, postorder, levelorder -- as dictated by the structure of the tree

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)

```java
private BinaryNode<AnyType> root;

public BinarySearchTree( )
{
    root = null;
}

public void makeEmpty( )
{
    root = null;
}

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

BST "contains" Method

```java
 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 $O(\text{height})$ in all cases

Implementation of printTree

```java
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)

```java
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

```java
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;
}
```

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			

Building a BST

- Given an array of elements, what is the performance (best/worst/average) of building a BST from scratch?

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`**`();`

Using Tree Iterator

```
public static void main (String args[] )
{
        BinarySearchTree<Integer> tree = new
                        BinarySearchTree<Integer>();

        // store some ints into the tree

        InOrderIterator<Integer> itr =
                        tree.inOrderIterator( );
        while ( itr.hasNext( ) )
        {
                Object x = itr.next();
                // do something with x
        }
}
```

The InOrderIterator is a Disguised List Iterator

```java
// An InOrderIterator that uses a list to store
// the complete in-order traversal
import java.util.*;
class InOrderIterator<T>
{
    Iterator<T> _listIter;
    List<T> _theList;

    T next()
    {    /*TBD*/              }

    boolean hasNext()
    {    /*TBD*/              }

    InOrderIterator(BinarySearchTree.BinaryNode<T> root)
    {    /*TBD*/              }
}
```

List-Based InOrderIterator Methods

```
//constructor
InOrderIterator( BinarySearchTree.BinaryNode<T> root )
{
   fillListInorder( _theList, root );
   _listIter = _theList.iterator( );
}


// constructor helper function
void fillListInorder (List<T> list,
                     BinarySearchTree.BinaryNode<T> node)
{
   if (node == null) return;
   fillListInorder( list, node.left );
   list.add( node.element );
   fillListInorder( list, node.right );
}
```

List-based InOrderIterator Methods Call List Iterator Methods

```
T next()
{
    return _listIter.next();
}


boolean hasNext()
{
    return _listIter.hasNext();
}
```

InOrderIterator Class with a Stack

```
// An InOrderIterator that uses a stack to mimic recursive traversal
class InOrderIterator
{
        Stack<BinarySearchTree.BinaryNode<T>> _theStack;

        //constructor
        InOrderIterator(BinarySearchTree.BinaryNode<T> root){
                _theStack = new Stack();
                fillStack( root );
        }

        // constructor helper function
        void fillStack(BinarySearchTree.BinaryNode<T> node){
                while(node != null){
                        _theStack.push(node);
                        node = node.left;
                }
        }
```

Stack-Based InOrderIterator

```
T next(){
        BinarySearchTree.BinaryNode<T> topNode = null;
        try {
                topNode = _theStack.pop();
        }catch (EmptyStackException e)
        {
                return null;
        }
        if(topNode.right != null){
                fillStack(topNode.right);
        }
        return topNode.element;
}

boolean hasNext(){
        return !_theStack.empty();
}
}
```

More Recursive BST Methods

- `static <T> bool` **`isBST`** `( BinaryNode<T> t )`
  returns true if the Binary tree is a BST

- `static <T> T` **`findMin`**`( BinaryNode<T> t )`
  returns the minimum value in a BST

- `static <T>`
  `int` **`countFullNodes`** `( BinaryNode<T> t )`
  returns the number of full nodes (those with 2 children) in a binary tree

- `static <T>`
  `int` **`countLeaves`**`( BinaryNode<T> t )`
  counts the number of leaves in a Binary Tree