CS 367 - Introduction to Data Structures
Week 5, 2017

Homework h2 graded. Email TA (di3@wisc.edu) by Tuesday, July 18th - 5 pm.
Homework h4 posted, complete as soon as possible; Due by Sunday, July 23rd - 10 pm.
Program 2 posted due July 28th - 10 pm, start working on it as soon as possible.

Last Week

PriorityQueue, array-based heap, recursion, practice, complexity analysis

This Week

Read: Trees, Binary Search Trees

Recursion
- more practice writing/analyzing recursion (from last time)
- execution tree tracing

Searching

Categorizing ADTs Part 1

General Trees
- implementing
- determining tree height

Binary Trees
- implementing

Tree Traversals

Categorizing ADTs Part 2

Binary Search Tree (BST)
- BSTnodes

BST class

Binary Search Tree (BST)
- implementing print
- implementing lookup, insert, delete
- complexities of BST methods

CS Options/Courses

Binary Search Tree (BST)
- BSTnode class
- BST class
- implementing print
- implementing lookup, insert, delete
- complexities of BST methods

Next Week

Red-Black Trees, Graphs

Picking Lottery Numbers

What are your odds of winning the lottery? It depends on the number of possible combinations given how many numbers you have to pick and over what range:

Supercash - choose 6 out of 39 numbers (range 1 – 39)
Megabucks - choose 6 out of 49 numbers (range 1 – 49)

N Choose K: How many combinations of K things can you make from N things?

Recursive Definition:

c(n,k) =

→ Implement the c(n,k) method.

Execution Tree Tracing of c(n,k)

Searching

Linear Search:

Binary Search:

Categorizing ADTs Part 1: based on Layout

LINEAR

HIERARCHICAL

GRAPHICAL

General Tree

-

The Tree Node Class:

```
class Treenode<T> {
   private T data;
   private ListADT<Treenode<T>> children;
   ...
```

→ **Draw a picture** of the memory layout of a Treenode
(assume an ArrayList is used for the ListADT):

The Tree Class:

```
public class Tree<T> {
   private Treenode<T> root;
   private int size;

   public Tree() {
      root = null;
      size = 0;
   }
   ...
```

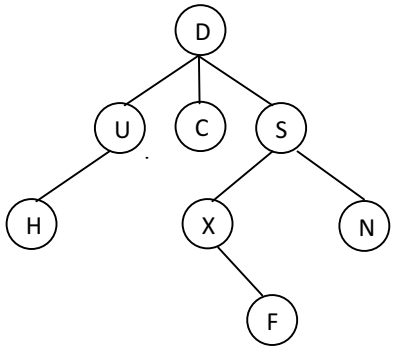
→ **Draw a picture** of the memory layout
of an empty general tree:

→ **Draw a picture** of the memory layout
of a general tree with a root node having 3 children:

Determining Height of a General Tree

Recall the height of a tree is the length of a path from the root to the deepest leaf.

→ **Write a recursive definition** for the height of a general tree.



→ **Complete the recursive height method** based on the recursive definition.
Assume the method is added to a Tree class having a root instance variable.

```
public int height() {
```

Binary Tree

-

The Tree Node Class:

```
class BinaryTreenode<T> {
   private T data;
   private BinaryTreenode<T> leftChild;
   private BinaryTreenode<T> rightChild;

   public BinaryTreenode(T item) {
      data = item;
      leftChild  = null;
      rightChild = null;
   }
   ...
```

→ **Draw a picture** of the memory layout of a BinaryTreenode:

The Tree Class:

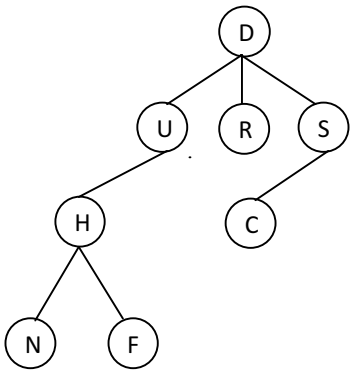
```
public class BinaryTree<T> {
   private BinaryTreenode<T> root;
   private int size;

   public BinaryTree() {
      root = null;
      size = 0;
   }
   ...
```

→ **Draw a picture** of the memory layout
of an empty binary tree:

→ **Draw a picture** of the memory layout
of a binary tree with a root node having 2 children:

Tree Traversals

Goal: visit every node in the tree exactly once



Level-order

General Tree Binary Tree

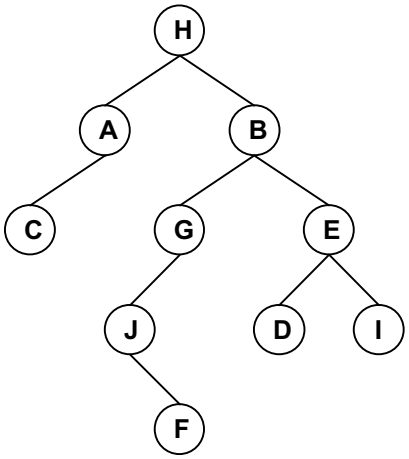
Pre-order

Post-order

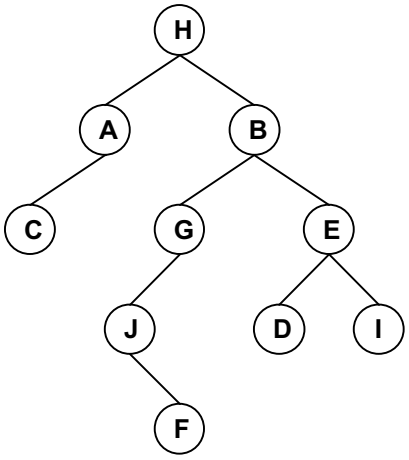
In-order

Practice – Binary Tree Traversals

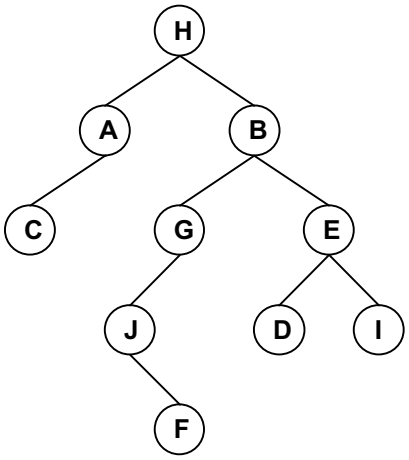
→ **List the nodes** using a pre-order traversal.



→ **List the nodes** using a post-order traversal.



→ **List the nodes** using an in-order traversal.



Categorizing ADTs Part 2

Binary Search Tree (BST)

-

Goal

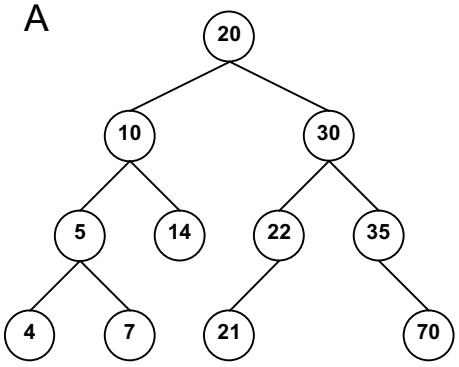
Example **2 3 6 7 10 12 13 15 17 19 22 24 26 27 30**

Ordering Constraint

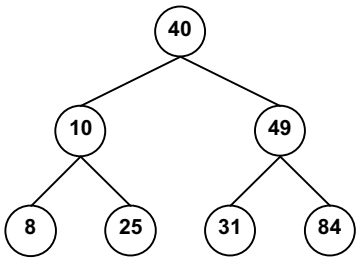
Practice - Identifying Binary Search Trees

→ Identify which trees below are valid BSTs.

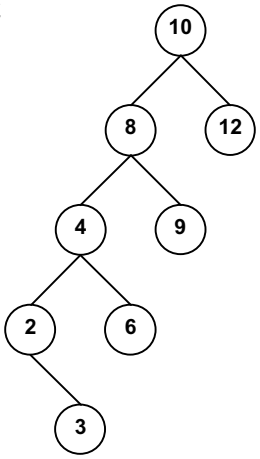
A



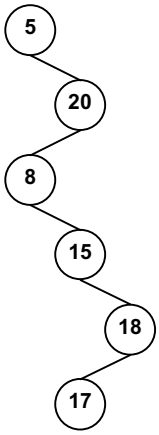
B



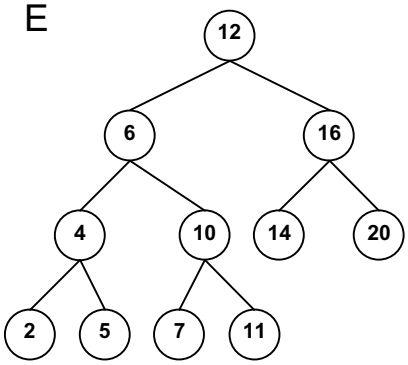
C



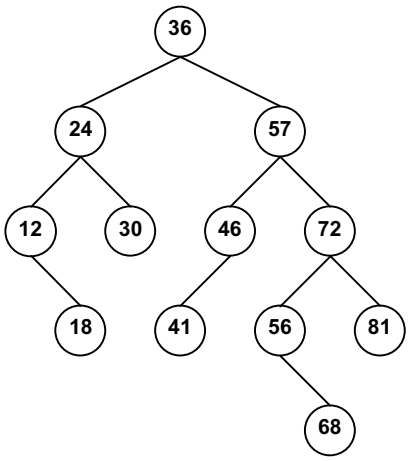
D



E



F



BSTnodes

→ Draw a picture of the memory layout of a Treenode:

```
class BSTnode<K> {

   private K key;
   private BSTnode<K> left, right;

   public BSTnode(K key, BSTnode<K> left, BSTnode<K> right) {
      this.key = key;
      this.left = left;
      this.right = right;
   }

   public K getKey() { return key; }
   public BSTnode<K> getLeft() { return left; }
   public BSTnode<K> getRight() { return right; }

   public void setKey(K newK) { key = newK; }
   public void setLeft(BSTnode<K> newL) { left = newL; }
   public void setRight(BSTnode<K> newR) { right = newR; }
}
```

BST Class

```
import java.io.*;  //for PrintStream

public class BST<K extends Comparable<K>> {

   private BSTnode<K> root;

   public BST() { root = null; }

   public void insert(K key)
                throws DuplicateException {


   }

   public void delete(K key) {


   }

   public boolean lookup(K key) {


   }

   public void print(PrintStream p) {


   }

   //add helpers ...




}
```

BSTs and `BSTnodes`

BST – Binary Search Tree

- is a key-value oriented collection of items where duplicate keys are not allowed
- **goal:** combine speed of binary search for access in an array O( )
  with speed of linking/unlinking in a chain of nodes O( )
- **shape constraint:** binary tree structure
- **order constraint:**

- in lecture, we'll explore BSTs with items having only a key value
  see readings for items having a key value and an associated value

BSTnode Class

```
class BSTnode<K> {

   private K key;
   private BSTnode<K> left, right;

   public BSTnode(K key, BSTnode<K> left, BSTnode<K> right) {
      this.key   = key;
      this.left  = left;
      this.right = right;
   }

   public K getKey() { return key; }
   public BSTnode<K> getLeft()  { return left; }
   public BSTnode<K> getRight() { return right; }

   public void setKey(K newK)             { key = newK; }
   public void setLeft(BSTnode<K> newL)   { left = newL; }
   public void setRight(BSTnode<K> newR) { right = newR; }
}
```

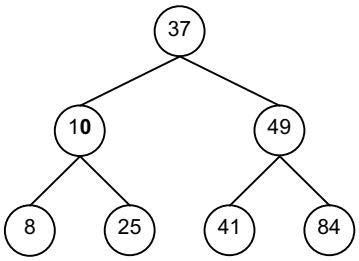
→ Draw a picture of the memory layout of a BSTnode.

BST Class

```
import java.io.*;  //for PrintStream

public class BST<K extends Comparable<K>> {

   private BSTnode<K> root;

   public BST() { root = null; }

   public void insert(K key)
                throws DuplicateException {


   }

   public void delete(K key) {


   }

   public boolean lookup(K key) {


   }

   public void print(PrintStream p) {


   }

   //add helpers ...




}
```

Implementing `print`

→ **Write a recursive definition** to print a binary tree given **`n`**, a reference to a BSTnode.



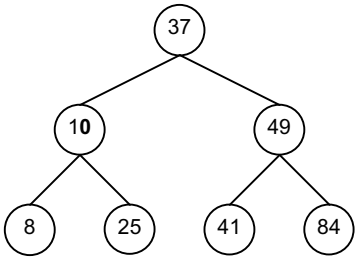
→ **Complete the recursive print method based** on the recursive definition.

```
public void print(PrintStream p) {
   print(root, p);
}

private void print(BSTnode<K> n, PrintStream p) {
```

Implementing `lookup`

Pseudo-Code Algorithm

```
private boolean lookup(BSTnode<K> n, K key) {
```

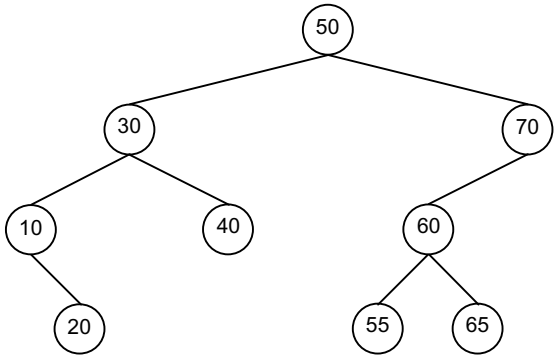


Implementing `insert`

High-Level Algorithm

```
private BSTnode<K> insert(BSTnode<K> n, K key)
                                  throws DuplicateException {
```

Practice - Inserting into a BST



→ Insert 5, 27, 90, 73, 57 into the tree above.

→ What can you conclude about the shape of a BST when values are inserted in sorted order?

→ Will you get a bad shape *only* if values are inserted in sorted order?

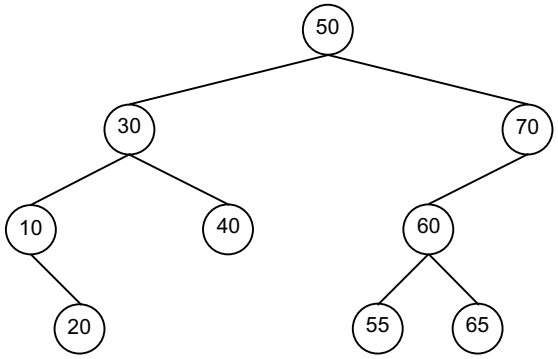
✵

Implementing `delete`

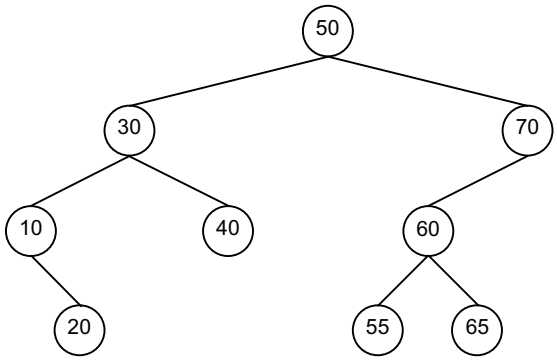
High-Level Algorithm

```
private BSTnode<K> delete(BSTnode<K> n, K key) {
```

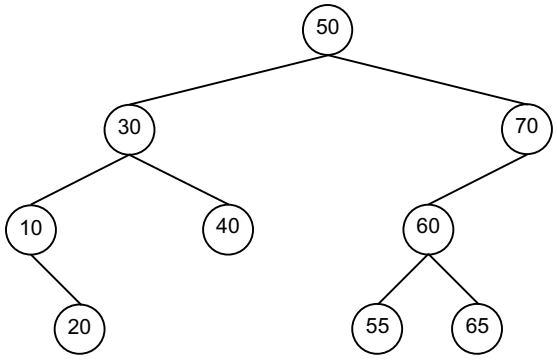
Practice - Deleting from a BST



→ Delete 90 from the tree above.
→ Delete 40 and 65 from the tree above.



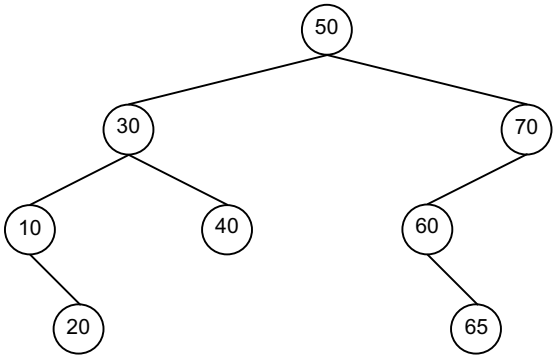
→ Delete 10 and 70 from the tree above and redraw the tree.



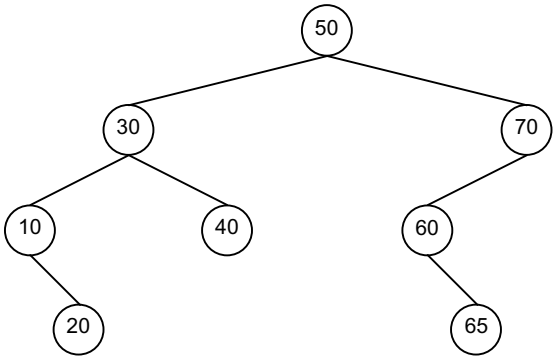
→ How do you delete 50 or 30 from the tree above?

Implementing `delete` (cont.)

Practice - Deleting from a BST



→ Delete 30 from the tree above using the _____.



→ Delete 50 from the tree above using the _____.

Complexities of BST Methods

Problem size: N =

print:

lookup:

insert:

delete:

Classifying Binary Trees

Full

Complete

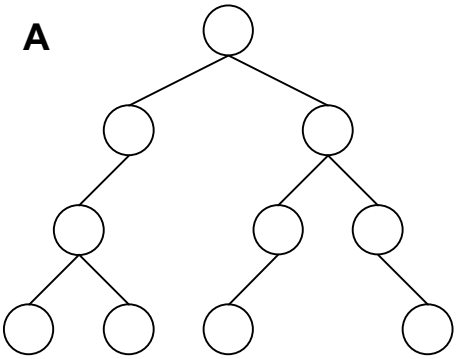
Height-balanced

Balanced

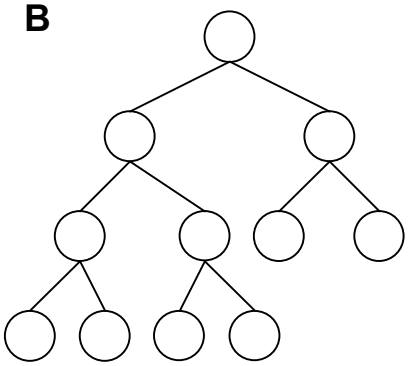
Practice - Classifying Binary Trees

→ Identify which trees below are full, complete and/or height balanced.

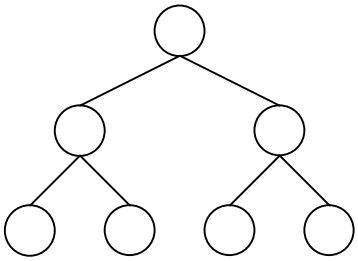
A



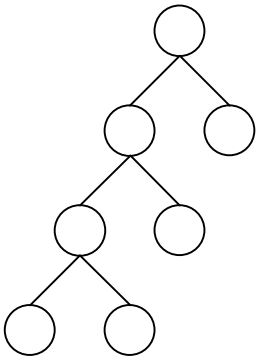
B



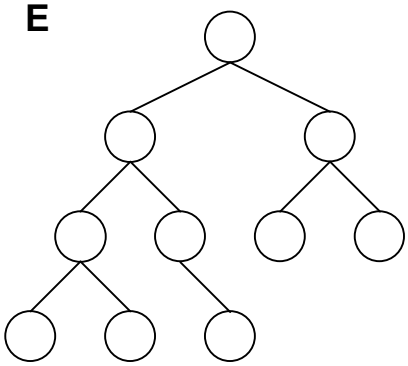
C



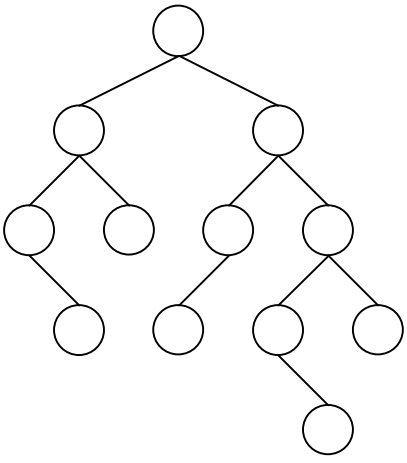
D



E



F



Balanced Search Trees

Goal:

Idea:

AVL

BTrees

CS Options

CS Certificate

5 CS Courses (12 credits minimum)
- Data Structures – CS 367 (& possibly prereq 302)
- 2 CS Courses >=400 level
- 2 Other CS Courses

CS Major (Declare as soon as able – requirements are changing)

Basic CS
- Discrete Math – CS 240
- Programming + Data Structures – CS 302, CS 367
- Basic Systems – CS 252, CS 354

Math
- Calculus – MA 221, MA 222
- 2 Beyond Calc – STATS 324 (intro applied stats), MA 340 (linear algebra)

Group A Theory
- Algorithms – CS 577

Group B Hardware/Software
- OS – CS 537

Group C Applications
- AI – CS 540

Group D Electives
- 2 CS Courses >=400 level

CS Double Major

- Must complete major requirements
- Easy for Computer Engineering Majors

CS Courses

Take Next

- CS 240 Introduction to Discrete Mathematics
- CS/ECE 252 Introduction to Computer Engineering (prereq for CS 354)
- CS/ECE 354 Machine Organization and Basic Systems (prereq for many group B)
- (CS 368) Learning a New Programming Language (C++ for CS 537)

- NOTE: CS/ECE 352 Digital Systems Fundamentals no longer required (is pre-req for CS 552)

>= 400 can take after CS 367

- CS 407 Foundations of Mobile Systems (spring, popular)
- CS 534 Computational Photography
- CS 539 Introduction to Artificial Neural Networks and Fuzzy Systems
- CS 540 Introduction to Artificial Intelligence
- CS 570 Human Computer Interaction (spring)

>= 400 can take after CS 367 + CS354

- CS 536 Introduction to Compilers
- CS 537 Introduction to Operating Systems
- CS 564 Database Management Systems: Design and Implementation
- CS 552 Introduction to Computer Architecture – CS 352, 354, and 367

>= 400 can take after CS 367 + Math

- CS 412 Introduction to Numerical Methods – MA 222 + MA 234 or CS 240
- CS 435 Introduction to Cryptography – MA 320 or MA 340
- CS 475 Introduction to Combinatorics – MA 320, 340, 341, or 375
- CS 514 Numerical Analysis = CS240, CS367 & MA 340
- CS 520 Introduction to Theory of Computing – CS240 & 367
- CS 524 Introduction to Optimization – CS302 and MA 320 or 340
- CS 525 Linear Programming Methods – MA 320 or MA 340 or MA 443
- CS 533 Image Processing – MA 320 or MA 340 (fall)
- CS 559 Computer Graphics – MA 320 or MA 340
- CS 576 Introduction to Bioinformatics – MA 222 (fall)
- CS 577 Introduction to Algorithms – CS 240