CS211, LECTURE 20
SEARCH TREES

ANNOUNCEMENTS:

OVERVIEW:

- motivation
- naive tree search
- sorting for trees and binary trees
- new tree classes
- search
- insert
- delete

1. Motivation

1.1 Search Structure

- continuing
- did linear; now want hierarchical
- so, use trees

1.2 Search Trees

- special trees organized for searching
- focus on ***binary search trees*** because of binary tree focus
- advanced topics later: balanced trees

1.3 Interface to implement

```
interface SearchStructure {
   void insert(Object o);
   void delete(Object o);
   boolean search(Object o);
   int size();
}
```

2. Searching A Tree

2.1 Assume built “at random” (no sorting)

2.2 Tree Traversal

- visit each node and do something with it
- depth-first: pre, post, in (for binary trees)
- breadth-first (also, level-order): each level at a time

2.3 Searching

- could do a traversal for search
- expensive: might have to search every node

2.4 Example of “complete” traversal

- Algorithm:
  - if current node is null, return false
  - else if current node has data, return true
  - else search left branch and search right branch; return result of checking if data is at any node
- Use CONDITIONAL OR (**||**) to help
- Code:

```
// In BinaryNode class:
public boolean naivefind(Object o,BinaryNode n) {
   if (n==null)
      return false;
   else if (o.equals(n.item))
      return true;
   else
      return naivefind(o,n.left) ||
             naivefind(o,n.right);
   }
```

2.5 Improvements

- use parent links (no recursion needed)
- sort the tree, as we did with arrays (rest of lecture!)

2.6 More info on traversal:

- free DS&A!
  http://www.brpreiss.com/books/opus5/html/
- see Chap 9 (Tree Traversal)

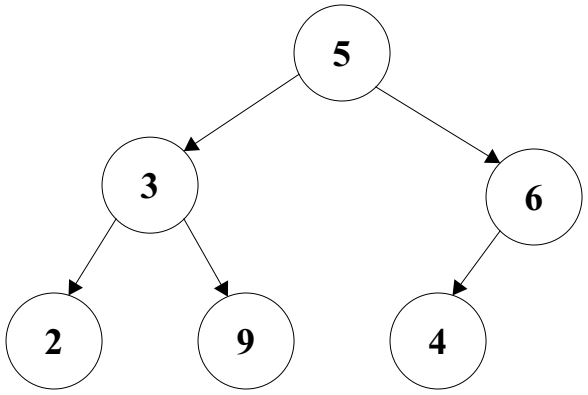
2.7 General Trees?

- using binary trees (see Lecture 18)
- searching general tree anyway? (Preiss: M-Way search tree in 10.6)

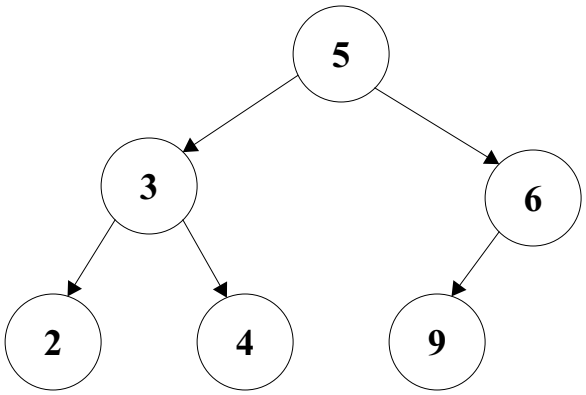
3. Binary Search Tree

3.1 Properties of BST:

- For given **`data`** at a node (root, root of subtree),
  - all nodes with data *smaller* than **`data`** are stored in the left subtree
  - all nodes with data *bigger* than **`data`** are stored in the right subtree
- What about nodes with data equal to data? no repeats



BT ?



BT ?

4. Binary Search Tree Intuition

- sort a list
- “grab middle” of list (search “halves” instead of whole)
- middle becomes root of tree
- sublists to left and right of root “flop” down
- detach the sublists and repeat process of grab & lift
- hook roots of the sublists to the previous root
- repeat until no more elements

5. Stubbed Out BST

```
abstract public class BinarySearchTree implements
                                          SearchStructure {
   private BinaryNode root;
   // more fields?

   public BinarySearchTree( ) { root = null; }
   public void insert(Object o);
   public void delete(Object o);
   public boolean search(Object o);
   public int size();

   // helper methods?
   // more methods?

}
```

6. Search

6.1 Algorithm:

- if tree is empty, return false
- if root object == search object, return true
- if root object < search object, search right subtree
- if root object > search object, search left subtree

6.2 Code:

```
public boolean search(Object o) {
   BinaryNode n = mySearch(o,root);
   // no empty tree, double check data:
   return ( n!=null && n.getItem().equals(o) );
}

// search helper:
private BinaryNode mySearch(Object o, BinaryNode r) {
   if (r == null) return null;
   int test = ((Comparable) r.getItem()).compareTo(o);
   if (test > 0)
      return mySearch(o, r.getLeft());
   else if (test < 0)
      return mySearch(o, r.getRight());
   else
      return r;
   }
```

7. Search for Max

7.1 Algorithm:

- If empty, return null
- Check right: if empty, return root value; otherwise, find rightmost leaf on the rightmost subtree

7.2 Code:

```
public Comparable findMax() {
   return findMax(root);
}

public Comparable findMax(BinaryNode r) {
   if (r==null)
      return null;
   if (r.getRight()==null)
      return (Comparable) r.getItem();
   else
      return findMax(r.getRight());
}
```

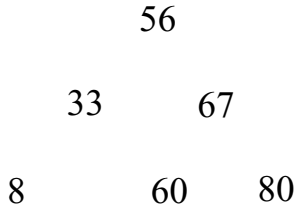
8. Checking if Tree is BST

8.1 Algorithm:

- If tree is empty or leaf -> BST
- Else, internal node:
  - compute smallest and largest values in left and right subtrees
  - check if subtrees are also BSTs
  - BST if both subtress are BSTs and
    largest object in left subtree is < root object and
    smallest object in right subtree is > root object

8.2 Advice

- perform post-order walk of tree:



9. Insertion

9.1 Algorithm:

- Search for object o in BST (assume no dups!)
- if at leaf (and empty root) make a new node with o
- else, if o < subtree root item, insert into left subtree
- else, if o > subtree root item, insert into right subtree

9.2 Code:

```
public void insert (Object o)  {
   root = insert(o,root);
   size++;
}

private BinaryNode insert(Object o, BinaryNode r) {
   if (r == null)
      r = new BinaryNode(o);
   else if (((Comparable) r.getItem()).compareTo(o) > 0)
      r.setLeft(insert(o,r.getLeft()));
   else if (((Comparable) r.getItem()).compareTo(o) < 0)
      r.setRight(insert(o,r.getRight()));
   return r; // actually, not good (no dups!)
}
```

9.3 Alternative Algorithm

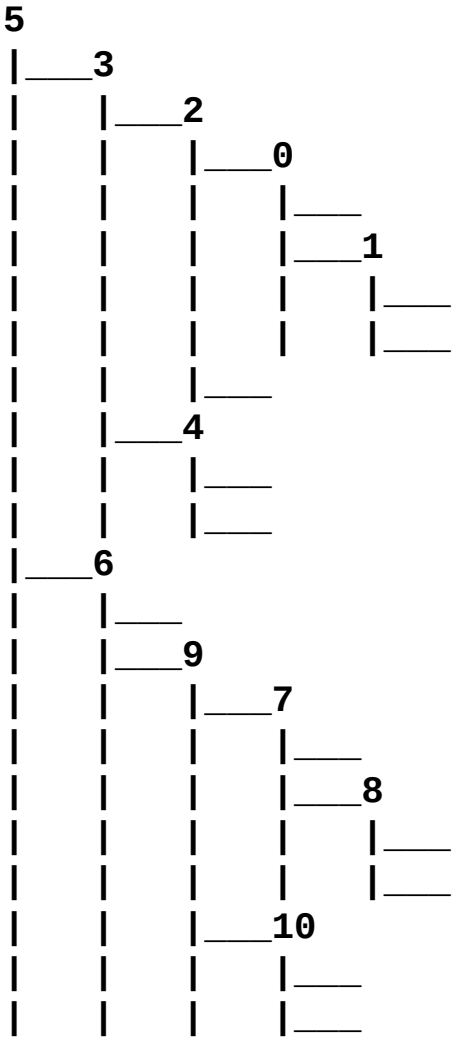
- Search for o
- during search, set these cursors/fingers:
  - current: node where search ends
  - previous: parent of current node
- if at leaf or empty tree, make new node with o
- insert at previous node: based on o and root's item,
  - insert left or
  - insert right
- see **`testBST.java`** for original version

9.4 Example (see `TestBST.java`)

```
public class TestBST {
   public static void main( String [ ] args ) {

      BST bst = new BST();

      BinaryNode b5 = new BinaryNode(new Integer(5));
      bst.setRoot(b5);

      BinaryNode b2 = new BinaryNode(new Integer(2));
      BinaryNode b3 = new BinaryNode(new Integer(3));
      BinaryNode b4 = new BinaryNode(new Integer(4));
      BinaryNode b6 = new BinaryNode(new Integer(6));
      BinaryNode b9 = new BinaryNode(new Integer(9));

      b5.setLeft(b3);
      b5.setRight(b6);
      b3.setLeft(b2);
      b3.setRight(b4);
      b6.setRight(b9);


      bst.insert(new Integer(0));
      bst.insert(new Integer(1));
      bst.insert(new Integer(7));
      bst.insert(new Integer(8));
      bst.insert(new Integer(10));

      System.out.println(bst.toTree());
   }
}
```

9.5 Session

Note: I modified **`toTree()`** in **`BinaryNode`**!

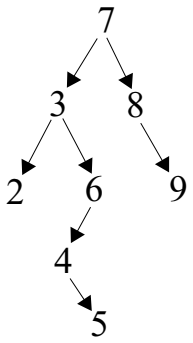


9.6 Modified toTree

- now Tree-i-fying leaves
- can distinguish between left and right subtrees
- code could be cleaned up…
- Code:

```
public String toTree(String blank,String spacing) {
   String s = "" + item + "\n";
   if (left == null)
      s += spacing + "\n";
   else
      s += spacing + left.toTree(blank,
            blank+spacing);
   if (right == null)
      s += spacing + "\n";
   else
      s += spacing + right.toTree(blank,
            blank+spacing);
   return s;
}
```

10. Deleting

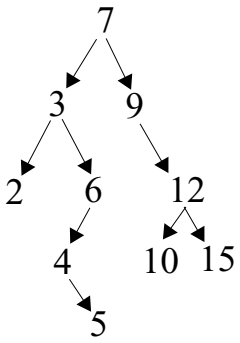


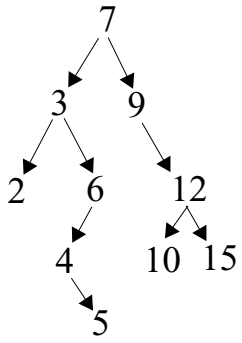
10.1 Easy cases:

- leaf (9): change ref in parent (8) to null
- 1 child (6): change parent's (3) child to child's child (4) (so, 3's child becomes 4)

10.2 Node has two children?

- a bit harder
- more general algorithm for deleting node N with item i:
  - walk down tree until you find N with i
  - let p be parent of N
  - if left subtree of N is null, make right subtree of N into subtree of p
  - if left subtree of N is not empty, extract max value from left subtree of N and stick that into N





11. Exercises

- Write a recursive **`findMin`**.
- Write iterative versions of **`search`**, **`findMax`**, **`findMin`**, and **`insert`**.
- Write method **`checkBST`** to check if a tree is a BST.
- Write **`delete`**…see **`testBST`** and textbooks for course.