Lab 16: Binary Search Trees

A. Review Search with Array and Linked Lists

Search in a Collection

An important operation provided by collection classes is finding whether a given item is an element of the collection. You've encountered this operation in several contexts already, for example, the `AmoebaFamily` activity involving locating an amoeba with a given name.

Search Estimates

For each of the questions below, the answer is one of the five options:

- a constant number
- proportional to `k`
- proportional to `n`
- proportional to `log n`
- proportional to `n*n`

Question 1

Suppose that `n` integers are stored in an `IntSequence` object. How many comparisons are necessary in the worst case to determine if a given integer `k` occurs in the sequence?

Question 2

Suppose that `n` comparable objects are stored in an `List` object. How many comparisons are necessary in the worst case to determine if a given object `k` occurs in the list?

Question 3

Suppose that `n` integers are stored in *increasing* order in an `IntSequence` object. How many comparisons are necessary in the worst case to determine if a given integer `k` occurs in the sequence?

Question 4

Suppose that `n` comparable objects are stored in increasing order in a `LinkedList` object. How many comparisons are necessary in the worst case to determine if a given object `k` occurs in the list?

Solutions

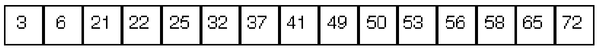
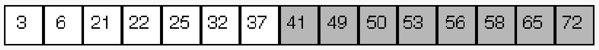
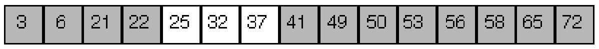
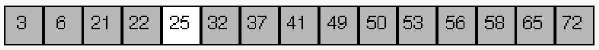
Question 1: proportional to `n` Question 2: proportional to `n` Question 3: proportional to `log n` Question 4: proportional to `n`

Binary Search

We encountered a variation of the binary search algorithm in a guessing game in a lab exercise earlier this semester. Used with an array, it assumes that the elements of the array are in order (say, increasing order). A version that returns the position in the array of the value being looked for works as follows:

1. Set `low` to 0 and `high` to the length of the array, minus 1. The value we're looking for—we'll call it `key`—will be somewhere between position `low` and position `high` if it's in the array.
2. While `low ≤ high`, do the following: (a) Compute `mid`, the *middle* of the range `[low,high]`, and see if that's `key`. If so, return `mid`. (b) Otherwise, we can cut the range of possible positions for `key` in half, by setting `high` to `mid-1` or by setting `low` to `mid+1`, depending on the result of the comparison.
3. If the loop terminates with `low > high`, we know that `key` is not in the array, so we return some indication of failure.

The diagrams below portray a search for the key 25. Elements removed from consideration at each iteration are greyed out.

`low = 0, high = 14, mid = 7`	
`low = 0, high = 6, mid = 3`	
`low = 4, high = 6, mid = 5`	
`low = 4, high = 4, mid = 4`	

Since (roughly) half the elements are removed from consideration at each step, the worst-case running time is proportional of the log base 2 of `N`, where `N` is the number of elements in the array.

Binary Search with Linked Lists

Question

Consider applying the binary search algorithm to a sorted doubly linked list. The variables `low`, `high`, and `med` would then be references to nodes in the list. Which of the steps in the binary search algorithm would execute significantly more slowly for a collection stored in a sorted doubly linked list than for a collection stored in a sorted array?

1. Set `low` to 0 and `high` to the length of the array, minus 1. The value we're looking for—we'll call it `key`—will be somewhere between position `low` and position `high` if it's in the array.
2. While `low ≤ high`, do the following: (a) Compute `mid`, the *middle* of the range `[low,high]`, and see if that's `key`. If so, return `mid`. (b) Otherwise, we can cut the range of possible positions for `key` in half, by setting `high` to `mid-1` or by setting `low` to `mid+1`, depending on the result of the comparison.
3. If the loop terminates with `low > high`, we know that `key` is not in the array, so we return some indication of failure.

A) Step 2a: It takes extra time (using a linked list) to test to see if we have found the key

B) Step 2a: It takes extra time (using a linked list) to calculate the index of the middle node

C) Step 1: It takes extra time (using a linked list) to figure out the length of the linked list

D) Step 2a: It takes extra time (using a linked list) to access the value in the middle node

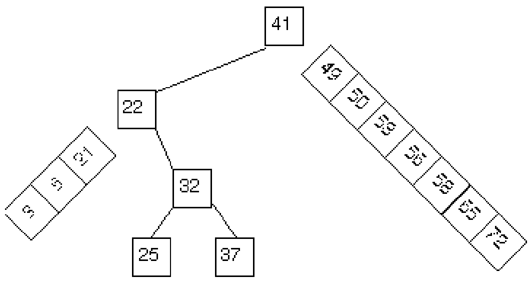
Solution

Answer `D`

B. Work with Binary Search Trees

Binary Search Trees

One might naturally look for a linked structure that combines fast search with fast link manipulation of the found node. The tree of choices in a binary search algorithm suggests a way to organize keys in an explicitly linked tree, as indicated in the diagram below.

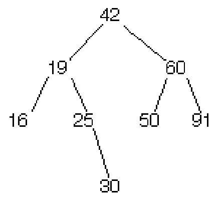
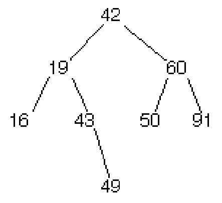
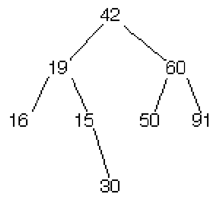


The tree that results is called a *binary search tree* (BST).

- Let the root value (one of the keys to be stored) be `k`.
- Put all the keys that are smaller than `k` into a binary search tree, and let that tree be `k`'s left subtree.
- Put all the keys that are larger than `k` into a binary search tree, and let that tree be `k`'s right subtree.

(This organization assumes that there are no duplicate keys among those to be stored.)

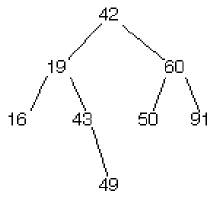
Given below are an example of a binary search tree and two examples of trees that are not binary search trees.

BST	nonBST	nonBST
		

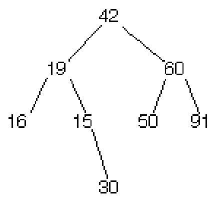
Identify non-binary search trees

Question 1

What nodes (can be multiple) must be removed from this tree to make it a binary search tree?



Question 2



What nodes (can be multiple) must be removed from this tree to make it a binary search tree?

Solutions

Question 1: `43,49`. Both of these are in the left subtree but greater than the root node `42`, and thus violate the constraint of a BST.

Question 2: `15`. `15` is less than its parent `19`. Removing `15` and relinking the parent of `30` to `19` would create a BST.

The `contains` method

Determining if a given value `key` is in the tree takes advantage of the invariants of a Binary Search Tree, where each subtree is also a Binary Search Tree. In pseudocode, here is an outline of the helper method

```
private static boolean contains (TreeNode t, Object key)
```

1. Nothing is in an empty tree, so if `t` is `null`, return `false`.
2. If `key` is equal to `t.myItem`, return `true`.
3. If `key < t.myItem`, `key` must be in the left subtree if it's in the structure at all, so return the result of searching for it in the left subtree.
4. Otherwise return the result of search for `key` in the right subtree.

This algorithm can go all the way down to a leaf to determine its answer. Thus in the worst case, the number of comparisons is proportional to `d`, the depth of the tree.

You will be writing the code for `contains` later on in the lab. For now, just understand how the implementation of the method works.

Use of Comparable objects

As just noted, finding a value in the tree will require "less than", "greater than", and "equals" comparisons. Since the operators < and > don't work with objects, we have to use method calls for comparisons.

The Java convention for this situation is to have the values stored in the tree be objects that implement the `Comparable` interface. This interface prescribes just one method, `int compareTo (Object)`. For `Comparable` objects `obj1` and `obj2`, `obj1.compareTo (obj2)` is

- negative if `obj1` is less than `obj2`,
- positive if `obj1` is greater than `obj2`, and
- zero if the two objects have equal values.

Once again, we will be using this information later on the in lab.

Balance and imbalance

Unfortunately, use of a binary search tree does not guarantee efficient search. For example, the tree

```
            7
           /
          6
         /
        5
       /
      4
     /
    3
   /
  2
 /
1
```

is a binary search tree in which search proceeds the same as in a linked list. We thus are forced to consider the *balance* of a binary search tree. Informally, a balanced tree has subtrees that are roughly equal in size and depth. In a few weeks, we will encounter specific algorithms for maintaining balance in a binary search tree. Until then, we will work under the possibly unwarranted assumption that we don't need to worry much about balance.

One can prove, incidentally, that search in a BST of `N` keys will require only about `2 ln N` comparisons (`ln` is the "natural log" of `N`) if the keys are inserted in random order. Well-balanced trees are common, and degenerate trees are rare.

Count of maximally imbalanced trees

There are 14 different binary search trees with 4 nodes. How many of the 14 are as deep as possible (i.e. they cause the search algorithm to make 4 comparisons in the worst case)?

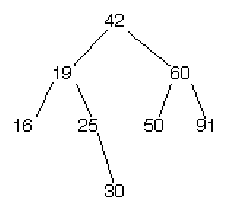
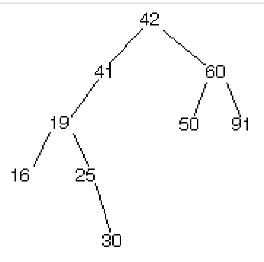
(Luckily these bad trees get rarer when `N` gets larger.)

Solution

`8` binary search trees. Starting at the root node, we are constrained to select a single child for this root node, and it can be either to the left or to the right of this root node. Similarly for this child node. So we have `2^3` possibilities.

Insertion into a BST

As you may have noticed in the preceding step, there are a large number of binary search trees that contain a given `N` keys. Correspondingly, there are typically a bunch of places in a BST that a key to be inserted might go, anywhere from the root down to a leaf. Given below are two trees; the tree on the right shows one possible way to insert the key 41 into the tree on the left.

To minimize restructuring of the tree and the creation of internal nodes with only one child, we choose to insert a new key only as a new *leaf*. The following pair of methods does this.

```
public void add(Comparable key) {
    myRoot = add(myRoot, key);
}

private static TreeNode add(TreeNode t, Comparable key) {
    if (t == null) {
        return new TreeNode(key); // return the new leaf
    } else if (key.compareTo(t.myItem) < 0) {
        t.myLeft = add(t.myLeft, key);
        return t;
    } else {
        t.myRight = add(t.myRight, key);
        return t;
    }
}
```

A common pattern for methods that modify trees is the following:

```
 Make a recursive call to modify a subtree— this call returns the root of the modified subtree.
Store the reference to the modified subtree into the appropriate TreeNode field (myLeft or myRight),
and return the root of the current tree.
```

(You may have encountered this in methods involving `ListNodes`.) In the code above, this is done in the statements

```
t.myLeft = add (t.myLeft, key);
return t;
```

and

```
t.myRight = add (t.myRight, key);
return t;
```

A BinarySearchTree class, extending Binary Tree

Now it's time to start writing code!

Since binary search trees share many of the characteristics of regular binary trees, we can define the `BinarySearchTree` class using inheritance. Do this as follows.

1. First, redefine relevant `private` methods and instance variables in `BinaryTree.java` as `protected`.
2. Next, create a file `BinarySearchTree.java` with the class definition
3. Lastly, change `myItem` from an `Object` to a `Comparable` type. Change method signatures accordingly.

```
 public class BinarySearchTree extends BinaryTree ...
```

(The `BinarySearchTree class` shouldn't have any instance variables. Why not?)

```
4. Finally, supply two methods for the `BinarySearchTree` class.
```

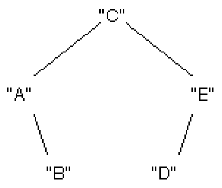
```
public boolean contains (Comparable)
```

takes a `Comparable` object as argument and checks whether the tree contains it. (Recall that `Comparable` objects provide an `int compareTo` method that returns a negative integer if this object is less than the argument, a positive integer if this is greater than the argument, and 0 if the two objects have equal values.)

```
public void add (Comparable)
```

takes a `Comparable` object as argument and adds it to the tree *only if it isn't already there*. The trees you create with the `add` method will thus not contain any duplicate elements.

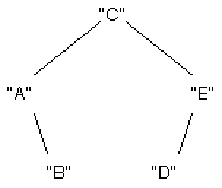
```
5. Test your code. In particular, using only calls to `add` inside JUnit or a `main` method, create the tree shown below.
```



Comparison Counting

Question

How many comparisons *between keys* were necessary to produce the sample tree in the previous step (shown below)? Ignore equality checks.



Solution

6 comparisons.

C. Homework

Readings

Next lab, we'll finish up trees and move into maps.

Programming Exercises

Complete the following two exercises, and submit your solutions in the files `BinaryTree.java` and `BinaryTreeTest.java`. Submit your files as `hw13`. `BinaryTreeTest.java` should test the new methods described below.

1) A textbook used in a previous offering of CS 61BL presented code for an inorder iterator. This code is in the file `lab16/InorderIterator.java` (it should go inside the `BinaryTree` class). The iterator is not optimally coded; in particular, the invariant relation between `currentNode` and `nodeStack` maintained by the next method isn't stated anywhere, and anyway this invariant is more complicated than it should be. Remove the `currentNode` instance variable so that the methods are only dependent on `nodeStack`. Fix the code so that, after each call to the the `next` method, the next item to return will be at the top of `nodeStack`. The corresponding change to `hasNext` will be

```
public boolean hasNext( ) {
    return !nodeStack.isEmpty( );
}
```

Your iterator constructor must run in time proportional to the depth of your tree. Hint: You'll need to import `java.util.Iterator`, `NoSuchElementException`, and `Stack`.

2) Write and test a constructor for the `BinaryTree` class that, given two `ArrayLists` of `Comparables`, constructs the corresponding binary tree. The first `ArrayList` contains the objects in the order they would be enumerated in a *preorder* traversal; the second `ArrayList` contains the objects in the order they would be enumerated in an *inorder* traversal. For example, given

`ArrayLists` "A", "B", "C", "D", "E", "F" and "B", "A", "E", "D", "F", "C", you should initialize the tree as follows: