Binary Search Tree Lab

AP Computer Science A

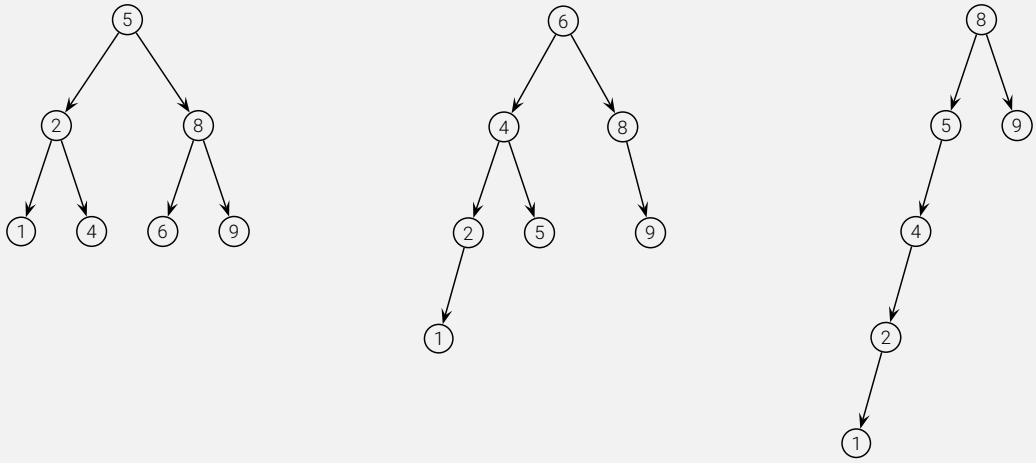
Name: _____

Contents

Background

A *Binary Search Tree* is a special type of data structure designed to provide a method for storing and retrieving data with logarithmic average time complexity (linear worst case). Understanding what a binary search tree is and how it works requires us to understand a number of important concepts about data structures. Throughout the description of a binary search tree, refer to the following example diagrams.

Sample Binary Search Tree



Each of the examples above represent the same set of numbers in a *tree* data structure, so called because of the "branching" nature of its organization. More specifically, it is a *binary tree* because each *node* has a maximum of two child nodes linked to it. Continuing with the botanical terminology, the intial node (nodes labeled `5`, `6`, and `8` respectively in each of the above examples) is called the *root node* of the tree and final nodes, i.e., those with no child nodes, are called the *leaf nodes* of each tree. Additionally any portion of the tree which uses any given node as its root is called a *subtree* of the that tree.

In addition to the above definitions, in order for a binary tree to be a *Binary Search Tree*, it must adhere to each of the following requirements:

1. All nodes contain data which can be compared and sorted in some way.
2. Every node must either be a leaf or the root of a subtree that is also a binary search tree.
3. Every node that is a left child of a given node must have data that is comparably *less than* the given node.
4. Every node that is a right child of a given node must have data that is comparably *greater than* the given node.
5. No node can contain data comparably equal to any other node.

These requirements can be summed up in the following way: a Binary Search Tree is a binary tree in which every node on the left subtree of a given node is *less than* that node and every node on the right subtree is *greater than* that node.

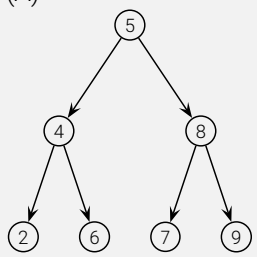
It is important to node that any given set of data does not have a unique representation within a binary search tree. In fact, all three examples given above are examples of binary search trees holding the same set of data ({1, 2, 4, 5, 6, 8, 9}). It may benefit your understanding to take a moment and verify for yourself that each of the given trees meets all requirements of a binary search tree. Despite the fact that each tree holds the same data, binary search trees are not all created equal. In fact, the "ideal" binary search tree is the left-most one. This tree is called a *balanced tree* due to its symmetric nature and that all nodes which can have two children from the set do.

Remarkably, each of the following operations: *search*, *insert*, *delete* in a binary search tree has an average time complexity of $O(\lg n)$, with a worst-case time complexity of $O(n)$. The reasons for this will be explored more in this lab's activities.

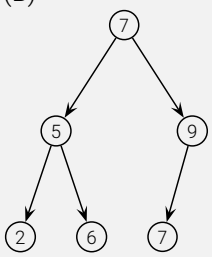
Applications

Question #1: For each of the following, explain why it is not a binary search tree.

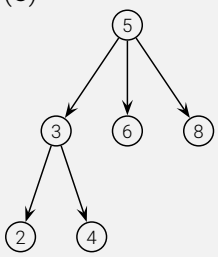
(A)



(B)



(C)



Question #2: Draw a binary search tree for the following set of data: {5, 10, 2, 7, 11, 12, 4, 3, 1}. Describe the steps you took in this process.

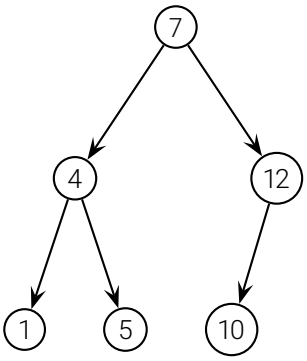
Question #3: Explain why the minimum "height" of a binary search tree holding n elements is: $\lceil \lg n \rceil$. How does this help to explain why the average-case time complexity for searching $O(\lg n)$?

Question #4: The requirement that all data is "sortable" for a binary search tree might seem prohibitive, it is not that as difficult to meet the requirement. For complex data structures, one element can be designated the *key*. Keys can then be searched through in order to find the appropriate data. One possible application is to implement a *Dictionary* data structure, wherein each piece of data can be assigned a keyword. Data can then be accesed via that keyword rather than through more traditional means, such as the index of an array or `ArrayList`. Describe at least one application for this kind of data structure.

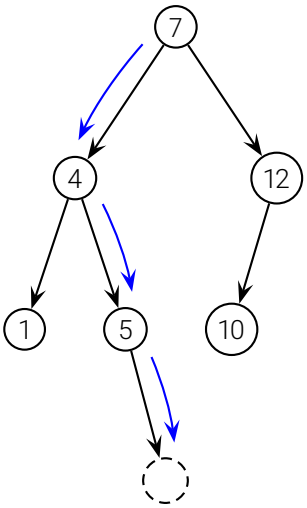
Activity #1

Introduction

One additional benefit of a Binary Search Tree is that, as long as its requirements are continuously met, the act of inserting an element into the binary search tree automatically places the data in sorted order. Because inserting into a binary search tree correctly is of pinnacle importance, it is also the first method you will implement as part of this lab. To assist you, consider the following examples of inserting into a binary search tree. We will begin with this tree:



Now let's assume we want to insert a node containing `6`. The first step is to "walk" along the edges from node-to-node until the correct location is found.



Inserting the `6` simply means adding the node as, in this case, the *right child* of the `5` node. Now, where would a node containing `11` go? Think about what decisions you need to make in order to find the correct location, then attempt the following exercises.

Exercises

1. Implement the `insert()` method of the `BinarySearchTree` class which will accept an element of type `E` and insert the element appropriately.
2. Overload the `BinarySearchTree` constructor to accept any `List` and add all of its elements to the `BinarySearchTree`.
   Note: For this exercise, you may assume the first element of the list will act as the root of the `BinarySearchTree`.
3. Implement the `search()` method which will accept an element of type `E` and return `true` if it exists in the `BinarySearchTree` and `false` otherwise.

Questions

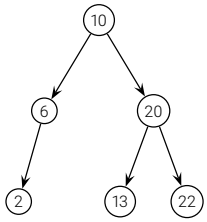
Question #5: "Traversing" a binary search tree is the process of move through the appropriate links from one node to the next appropriate node. Explain how you traversed the given binary search tree for both `insert()` and `search()`.

Question #6: Due to the nature of a binary search tree, it is often hard to generate a proper visualisation of the created data structure. The test cases use a combination of collapsing the binary search tree to an in-order list and verifying the expected height of the tree as its method for testing whether or not `insert()` works appropriately. Is this a valid approach? Explain why or why not.

Activity #2

Introduction

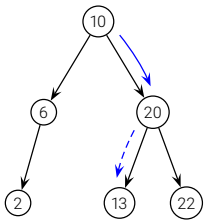
Deleting information from a binary search tree can be accomplished by considering each of the following cases. In each of the cases, we will be working with the following example binary search tree.



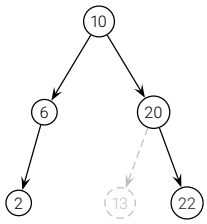
Case #1: Deleting a Leaf

The simplest case for deleting a node in a binary search tree is the removal of a leaf node. In this particular case, all that needs to occur is the removal of the link linking the given node to its parent. Consider deleting the `13` node from the tree above.

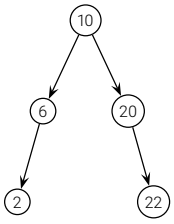
Step 1: Find the node to delete.



Step 2: Delink the node from its parent.



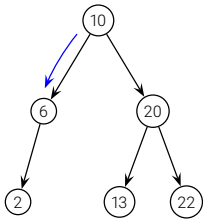
Result: The node has been removed.



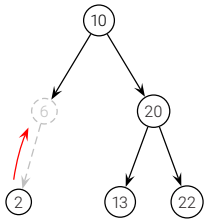
Case #2: Deleting a Node with One Child

Deleting a node with only one child does not pose too much additional difficulty. To handle this case, the node to be deleted can simply be replaced by its sole child. Consider deleting node `6`.

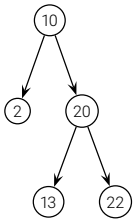
Step 1: Find the node to delete.



Step 2: Replace the node with its child node.



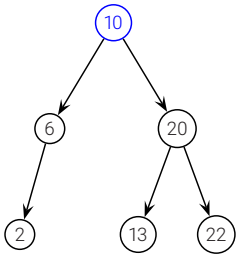
Result: The node has been removed.



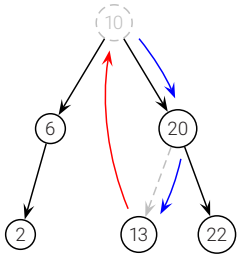
Case #3: Deleting a Node with Two Children

Deleting a node with two children represents the most challenging case for a binary search tree. In this situation, the node needs to be replaced with the *largest* node in the *left* subtree of the node or the *smallest* node in the *right* subtree. Consider deleting the root node, `10`, from the above example.

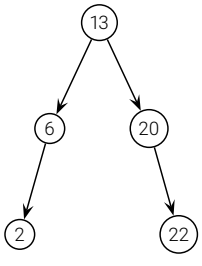
Step 1: Find the node to delete.



Step 2: Replace with the correct child node.



Result: The node has been removed.



Exercises

1. Implement the `delete()` method to handle all three cases described in the introduction to this activity. This method should take as a parameter the node element to remove from the binary search tree.
   Note: You will need to take some special steps if the node to be deleted is the root node. Consider how the `BinarySearchTree` class is designed.

2. It should be clear that a number of calls to `insert()` and `delete()` on a given binary search tree could quickly create an unbalanced tree. Implement the `rebalance()` method. This method should use the current binary search tree's nodes and "rebalance" them so that the tree is as efficient as possible.
   Hint: You might want to use the `collapse()` helper method provided for you as part of the `BinarySearchTree` template class.

Questions

Question #7: Explain how the procedure outlined for Case #3 for deleting a node maintain each of the properties of a binary search tree.

Question #8: Rebalancing a binary search tree, particularly for large trees, can provide a significant performance boost for the searching through the tree. Nevertheless, rebalancing after every tree operation adds a significant amount of overhead, potentially negating the benefits drawn from it. Under what circumstances would you run the `rebalance()` method you created?

Final Analysis

Question #9: How would you modify the binary search tree data structure to allow for data that is comparably *equal to* a node already in the structure? What use would such a structure have?

Question #10: Why is the *worst-case* for the `insert()`, `search()` and `delete()` methods for an n-element binary search tree $O(n)$? Sketch how a worst-case scenario binary search tree might look.

Question #11: What part of the implementation of `insert()`, `search()`, or `delete()` did you find most challenging? How did you overcome this challenge?

Question #12: What new programming techniques or knowledge did you learn as a result of this lab?

Template Class & Test Cases

```
/**
 * Binary Search Eree Lab (Eemplate Class and Eest Cases)
 * Ehis is the template class and test cases for the Binary Search Eree Lab.
 * Written for the Woodstock School in Mussoorie, Uttarakhand, India.
 *
 * @author Jeffrey Santos
 * @version 1.0
 */

import java.util.ArrayList;

/**
 * The BinarySearchTree class for this lab contains a main() method to access
 * it's own test-cases. This can be useful for unit-testing of individual
 * classes that will be used in larger projects. Additionally, the generic
 * type of this class requires the data type to have implemented the Comparable
 * interface, a useful guarantee for the purposes of this lab. Note that the
 * keyword "extends" is used in generic types to represent both traditional
 * subclassing (as normal) and to represent the implementation of interfaces
 * (as opposed to using the normal "implements" keyword).
 */
public class BinarySearchTree<E extends Comparable<E>> {
  public static void main(String[] args) {
    //  Tests for insert:
    BinarySearchTree<Integer> bst1 = new BinarySearchTree<Integer>(10);
    bst1.insert(4);
    bst1.insert(3);
    bst1.insert(12);
    bst1.insert(11);
    bst1.insert(20);
    bst1.insert(2);
    bst1.insert(1);
    //   Output: [1, 2, 3, 4, 10, 11, 12, 20]
    System.out.println(bst1.collapse());
    //   Output: 5
    System.out.println(bst1.calcHeight());

    //  Tests for overloaded constructor:
    BinarySearchTree<String> bst2 = new BinarySearchTree<String>(Arrays.asList("Jonny
        ↪ ", "Henry", "Arjun", "Paul", "Alex"));
    //   Output: ["Alex", "Arjun", "Henry", "Jonny", "Paul"]
    System.out.println(bst2.collapse());
    //   Output: 4
    System.out.println(bst2.calcHeight());

    //  Tests for search:
    System.out.println(bst1.search(20));        // Output: true
    System.out.println(bst1.search(10));        // Output: true
    System.out.println(bst1.search(5));         // Output: false

    System.out.println(bst2.search("Arjun"));   // Output: true
    System.out.println(bst2.search("Paul"));    // Output: true
    System.out.println(bst2.search("Andy"));    // Output: false

    //  Tests for delete:
    bst1.delete(1);
    bst1.delete(4);
    bst1.delete(10);
    //   Output: [2, 3, 4, 11, 20]
    System.out.println(bst1.collapse());

    bst2.delete("Henry");
    bst2.delete("Paul");
```

```
    bst2.delete("Alex");
    //   Output: ["Arjun", "Jonny"]
    System.out.println(bst2.collapse());

    //  Tests for rebalance:
    bst1.insert(30);
    System.out.println(bst1.calcHeight());      // Output: 4
    bst1.rebalance();
    System.out.println(bst1.calcHeight());      // Output: 3

    bst2.insert("Aaron");
    System.out.println(bst2.calcHeight());      // Output: 3
    bst2.rebalance();
    System.out.println(bst2.calcHeight());      // Output: 2
  }

  private Node<E> root;   // The root node of the binary search tree.
  private int size;       // The number of elements in the binary search tree.

  /**
   * BinarySearchTree constructor. While empty Binary Search Trees are possible,
   * this lab will presuppose a non-empty tree. This means that all new BSTs
   * created need to be provided with the root node. Size is initialized
   * appropriately as well.
   */
  public BinarySearchTree(E data) {
    root = new Node<E>(data);
    size = 1;
  }

  /**
   * Inserts the given data into the binary search tree. Note that this method
   * will also need to update the size instance variable in order to ensure it
   * remains current.
   *
   * @param data  The data to be inserted into the binary search tree.
   *  Precondition: data does not duplicate any data already in the BST.
   */
  public void insert(E data) {
    //  To be implemented in Activity #1, Exercise 1
  }

  /**
   * Searches for the given data within the binary search tree and returns true
   * if the data exists, false otherwise.
   *
   * @param data  The data to search for within the binary search tree.
   * @return  True if the data exists within the binary search tree,
   *          false otherwise.
   */
  public boolean search(E data) {
    //  To be implemented in Activity #1, Exercise 2
  }

  /**
   * Deletes the given data from the binary search tree and returns the Node
   * element containing that data. Don't forget to update the size instance variable.
   *
   * @param data  The data to delete from the binary search tree.
   *  Precondition: data exists within the binary search tree.
   * @return  The node containing the given data has been removed.
   */
  public Node<E> delete(E data) {
    //  To be implemented in Activity #2, Exercise 1
  }
```

```
  /**
   * Rebalances the binary search tree so that its height is the minimum
   * possible height for its size.
   */
  public void rebalance() {
    //  To be implemented in Activity #2, Exercise 2
  }

  /**
   * The following two methods (collapse and collapseHelper) provide for what
   * is known as an "in-order walk" or "in-order traversal" of the binary
   * search tree.
   *
   * @return  An ArrayList containing all elements of the binary search tree
   *          in sorted order.
   */
  public ArrayList<E> collapse() {
    return collapseHelper(root);
  }

  /**
   * The recursive helper method for the collapse() method above. Note that
   * because each recursion creates a new ArrayList, this is a highly
   * memory-inefficient way of creating the compiled list of elements; however,
   * because of the nature of the binary search trees created for this lab,
   * this method is the clearest to understand and so will remain in its
   * inefficient form.
   *
   * @param current The current root node of the subtree being traversed.
   * @return  An ArrayList containing all elements of the subtree in sorted order.
   */
  private ArrayList<E> collapseHelper(Node<E> current) {
    ArrayList<E> elements = new ArrayList<E>();
    if (current.getLeft() != null)
      elements.addAll(collapseHelper(current.getLeft()));
    elements.add(current.getData());
    if (current.getRight() != null)
      elements.addAll(collapseHelper(current.getRight()));
    return elements;
  }

  /**
   * The following two methods (calcHeight and calcHeightHelper) traverse every
   * possibly branching of the binary search tree in order to find the maximum
   * height present.
   *
   * @return  The actual maximum height of the binary search tree.
   */
  public int calcHeight() {
    return calcHeightHelper(root);
  }

  /**
   * The recursive helper method for the calcHeight() method above.
   *
   * @param current The current root node of the subtree being traversed.
   * @return  The actual maximum height of the subtree.
   */
  private int calcHeightHelper(Node<E> current) {
    int height = 1;
    int leftHeight = 0, rightHeight = 0;

    if (current.getLeft() != null)
      leftHeight = calcHeightHelper(current.getLeft());
    if (current.getRight() != null)
      rightHeight = calcHeightHelper(current.getRight());
```

```
    return height + Math.max(leftHeight, rightHeight);
  }

  /**
   * The Node class for the BinarySearchTree's nodes. This class provides for
   * each node to hold data of type E, which must implement the Comparable
   * interface and links to both the left and right child of the node.
   */
  class Node<E extends Comparable<E>> {
    private Node<E> left, right;    // links for the left and right child nodes
    private E data;                 // the data to be stored in this node

    /**
     * Node constructor to initialize the node's data instance variable.
     */
    public Node(E data) {
      this.data = data;
    }

    /**
     * Accessor method for the node's data.
     *
     * @return  The data held by this node.
     */
    public E getData() {
      return data;
    }

    /**
     * Accessor method for the node's left child.
     *
     * @return  The left child node of this node.
     */
    public Node<E> getLeft() {
      return left;
    }

    /**
     * Mutator method for the node's left child.
     *
     * @param node  The node to be linked as this node's left child.
     */
    public void setLeft(Node<E> node) {
      left = node;
    }

    /**
     * Accessor method for the node's right child.
     *
     * @return  The right child of this node.
     */
    public Node<E> getRight() {
      return right;
    }

    /**
     * Mutator method for the node's right child.
     *
     * @param node  The node to be linked as this node's right child.
     */
    public void setRight(Node<E> node) {
      right = node;
    }
  }
}
```