Binary Search Trees

Project 7. Deadline is Thursday: May 25th, 11:59pm.

1 milestone: Monday, May 22nd, 11:59pm.

This Project has a milestone submission. Refer to the submission section for more details

100 points

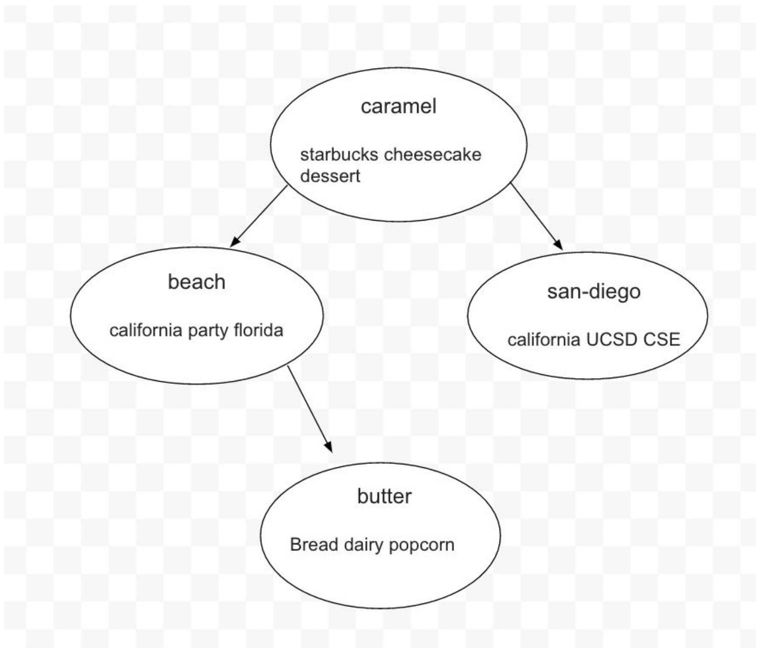
Brief idea: Your ADT will represent a Binary Search Tree where each node contains a `key(T)` and a `ArrayList<T>`

In Part 1, you will implement a Binary Search Tree ADT. The list of methods you must implement is mentioned below.

In Part 2, you will implement a very VERY basic version of a Search Engine (your own little Google :)). You will first build a BSTree based on the document file provided. Once the BSTree is built, you will be given a query string and you must return all the documents that contain that query string. If there are multiple words in a query, then you must find the intersection of documents that contain all the query words.

Part 1. Binary Search Tree

For a BSTree of type String, the tree will have the following format:



Where *caramel*, *butter*, *san-diego* and *beach* are the keys in the BST(keys are the values on which the BST is ordered).

- *starbucks, cheesecake, dessert* are the contents of the ArrayList at node 'caramel'
- *california, CSE, UCSD* are the contents of the ArrayList at node 'san-diego'
- *Bread, dairy, popcorn* are the contents of the ArrayList at node 'butter'
- *california, party, florida* are the contents of the ArrayList at node 'beach'

Implementation Details

1. Now implement three classes: BSTNode, BSTree_Iterator and BSTree where **BSTNode and BSTree_Iterator are inner classes of BSTree.** Refer to the details of each method given in the table below.
2. You are allowed to add helper methods but make sure they are private.
3. Write a JUnit tester BSTreeTester that tests **each and every public method in BSTree.** You need not test the constructors and the methods of BSTNode as they are implicitly tested. Keep testing each method as you implement it.
4. Follow the method signatures given below **strictly.**
5. Please comment all your classes and methods and add a brief description of what each method does. Remember that nothing is obvious.
6. Provide brief descriptions of what each class and method does in the form of header comments. Utilize inline comments when necessary to explain logic.
7. You may assume that no duplicate keys will be inserted in this part
8. **Important: At least three methods must be done recursively. We will check that.**
9. You may use additional helper methods for implementing the recursion

Methods of BSTNode class

Method	Description
public BSTNode(BSTNode left, BSTNode right, ArrayList<T> relatedInfo, T key)	A constructor that initializes the BSTNode instance variables.
public BSTNode(BSTNode left, BSTNode right,T key)	A constructor that initializes BSTNode variables. Note: This constructor is used when you want to add a key with no related information yet. In this case, you must create an empty ArrayList for the node.

public T getKey( )	Returns the key
public BSTNode getLeft( )	Returns the left child of the node
public BSTNode getRight( )	Returns the right child of the node
public ArrayList<T> getRelatedInfo( )	Returns the ArrayList of the node
public void setLeft(BSTNode newLeft)	Setter for left pointer of the node
public void setRight(BSTNode newRight)	Setter for right pointer of the node
public void setRelatedInfo(ArrayList<T> newInfo)	Setter for the ArrayList of the node
public void addNewInfo(T info)	Append new info to the end of the existing ArrayList of the node You may use the ArrayList API's add method here.
public boolean removeInfo(T info)	Remove 'info' from the ArrayList of the node and return true. If the ArrayList does not contain the value 'info', return false You may use the ArrayList API's remove method here.

Methods of BSTree class

Method	Description
public BSTree()	A 0-arg constructor that initializes root to null and nelems to 0
public BSTNode getRoot()	Returns the root of BSTree. Returns null if the tree is empty
public int getSize()	Returns nelems
public void insert(T key)	Inserts a key into the BST. Throws NullPointerException if key is null. Note: This method would insert a node with the key and an empty ArrayList into the tree.
public boolean findKey(T key)	Return true if the 'key' is found in the tree, false otherwise. Throw NullPointerException if key is null

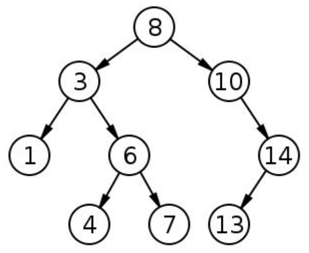
public void insertInformation(T key, T info)	Inserts ‘info’ into the ArrayList of the node whose key is ‘key’. Throw NullPointerException if ‘key’ or ‘info’ is null Throw IllegalArgumentException if ‘key’ is not found in the BST
public ArrayList<T> findMoreInformation(T key)	Return the ArrayList of the node with key value ‘key’ Throw NullPointerException if key is null Throw IllegalArgumentException if ‘key’ is not found in the BST
public int findHeight()	Returns the height of the tree. The **height** of a tree is the length of the longest downward path to a leaf from the root. By convention, height of an empty tree is -1 and the height of the root is 0.
public int leafCount()	Returns the number of leaf nodes in the tree.

BSTree_Iterator

In this section, you will build an Iterator for your BST, that iterates over the BST. In particular, you must implement the next() and hasNext() methods in the iterator. The first call to next() will return the smallest number in BST. Calling next() again will return the next smallest number in the BST, and so on. Sounds familiar?

Yes! You will essentially implement an iterative inorder traversal using your iterator. The basic approach is to use a Stack of BSTNode to keep track of the path from the root to the current node. (**You may use Java’s Stack API here).**

Your constructor in the iterator will populate the stack with all nodes from the root to the leftmost leaf node (let’s call it the *leftPath* of the tree). In each call to the next( ), you will pop a node from the Stack and push the *leftPath* of the tree rooted at its right node. Consider the example below:



The constructor of your Iterator will initialize the Stack with the nodes : [1, 3, 8] where 1 is the top of the Stack.
First call to next( ) returns 1 (top of the stack) with no change to the Stack (as 1 has no right subtree).
Second call to next( ) returns 3 (top of the stack) but now it also pushes 6 and 4 in that order to the Stack as they are in the *leftPath* of the tree rooted at 6.
Third call to next( ) returns 4, with no change to the Stack.
Fourth call to next( ) returns 6, and adds 7 to the Stack.
…...and so on.

Reasoning behind the above algorithm: In each call to next() which asks for the next element, you obviously won't go to the left subtree as all the elements there have already been returned by previous calls to the next( ) method. You would go to the smallest number in the right subtree if the right subtree is not null.

Implementation details

The iterator must implement the Iterator class as follows:
`public class BSTree_Iterator implements Iterator<T>`

Method	Description
public BSTree_Iterator()	Constructor that initializes the Stack with the leftPath of the root
public boolean hasNext()	Returns false if the Stack is empty i.e. no more nodes left to iterate, true otherwise
public T next()	Returns the next item in the BST. Throws NoSuchElementException if there is no next item

Once you implement the BSTree_Iterator class, add the following method to your BSTree class:

```
public Iterator<T> iterator() {
     return new BSTree_Iterator();
}
```

You can now create an iterate over your BSTree by calling the iterator( ) method on it as follows in the tester:

```
Iterator<String> iter = myTree.iterator();
```

Test your BSTree and BSTree_Iterator methods thoroughly. **Remember, all public methods must be tested.**

Part 2. Search Engine

In this part, you will implement a very VERY basic search application. The program starts with reading a text file (**passed as a command-line argument**) containing the documents and their corresponding keywords. For example, consider the following text file:

```
Pineapple
fruit yellow nutrition
Los-Angeles
California sunny warner
San-Diego
California beach UCSD
Nice-Places
California sunny beach NYC
```

Here, Pineapple is a **document** whose **keywords** are fruit, yellow and nutrition. Los-Angeles is a **document** whose **keywords** are California, sunny and warner.

Part 2.1 Populating the BST

- **In a file SearchEngine.java**, write a method **populateSearchTree** that reads the file and populates the BST with nodes. Each **keyword** is a key in the BST and the document name is added to the ArrayList of the keyword. For example, for the above file, your BST will have a node whose key is 'california' and its ArrayList contains 'los-angeles', 'san-diego' and 'nice-places'. Similarly 'sunny' is a key of a node whose ArrayList will contain 'los-angeles' and 'nice-places'.

- You may assume that the type T here is always a String.

- We want our search to be case-insensitive in order to get better search results and so make sure that you convert a string to lower case before inserting it to the BST. You may use the String API method toLowerCase() to do this for you.

- Your BST must not contain duplicate keys. For example, since the keyword ‘california’ is present in both ‘los-angeles’ and ‘san-diego’, you will have only one node with the key ‘california’ whose ArrayList will contain both ‘los-angeles’ and ‘san-diego’. (This is the whole purpose of using a ArrayList :))

Part 2.2 Query Search

In this part you will create a method **searchMyQuery** to use your BST. Given any query string (**passed as a second command line argument**), you must return all documents that contain that query string. If there are multiple words in a query string, then you must return the intersection* of documents that contain all the words.

- For example, if the query string is "california", then your program should print :

```
"Documents related to california are: [los-angeles, nice-places,
san-diego]"
```

- If the query is "sunny california", then the result is:

```
"Documents related to sunny california are: [los-angeles, nice-places]
Documents related to california are: [san-diego]"
```

Results are printed in the order of relevance : “sunny california” first, followed by the results for “sunny” and “california” (if any). **Note that there are no results printed for “sunny” because all its results are already returned by the query “sunny california”. Your search engine must not return repeated results.**

- If there are no matches for a query “california”, then the result is:

```
"The search yielded no results for california”.
```

Note: For queries with more than 2 words, we ideally want to check all possible combination of words in the documents. However, to keep this homework simple, we will follow the following basic rule in query processing:

Query: word1 word2 word3
Output:
<Output of query word1 word2 word3>
<Output of query word1 - if any>
<Output of query word2 - if any>
<Output of query word3 - if any>

You must use the print methods already provided in the starter code. Do not change the print and the main methods.

* Here are a few methods that you may find helpful in the searchMyQuery algorithm:

- addAll
- Contains
- containsAll
- retainAll

Part 2.3 Sample Input and Output

Some examples of inputs and their respective outputs are given in the file HW7_InputOutput.txt.
Note 1: The query comes as a command-line argument as well. (for example, java hw7.SearchEngine input.txt “california”)
Note 2: The word “Output” in the file is **not** a part of the actual output.

Part 3. (Optional) Extra credit problems.

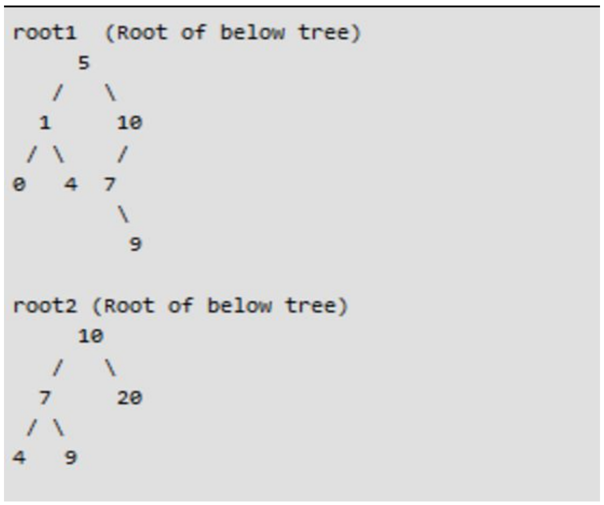
(5 points for each problem): You can improve your grade by working on the following problem:

Sometimes it would be convenient to modify data stored at your nodes. I’d like you to add two more public methods to the BSTree class. (but feel free to create more helper methods).

1) Method that returns the intersection of two BSTrees (Elements that can be found in both BSTs). The method takes two iterators as parameter and returns an ArrayList containing the intersection data of the two trees

```
public ArrayList<T> intersection(Iterator<T> iter1, Iterator<T> iter2)
```

Example: Consider the two trees below:



```
root1  (Root of below tree)
     5
   /   \
  1     10
 / \    /
0   4  7
        \
         9

root2 (Root of below tree)
     10
   /   \
  7     20
 / \
4   9
```

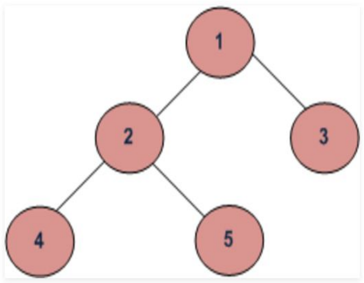
The output ArrayList will be : [4, 7, 9, 10]

2) Method that counts number of nodes at a given level. The method takes a level as a parameter and returns the count of nodes at that level (Given that level of the root is 0).

```
public int levelCount(int level)
```

For example, the tree below will have the following inputs/outputs

- levelCount(0) returns 1
- levelCount(2) returns 2
- levelCount(4) returns -1



3) Check whether two nodes with keys ‘n1’ and ‘n2’ in the given BST are cousins of each other or not. Two nodes are cousins of each other if they are at same level and have different parents. Also, you may assume that we will insert distinct keys of Integer type.

```
public boolean areCousins(T n1, T n2)
```

4) Check whether the given BST is complete. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

```
public boolean isCompleteBST()
```

Submission

1. **Milestone submission (Due on Monday, 22nd May)**

   Files to submit:

   a. BSTree.java (You must implement the class BSTNode, the BSTree_Iterator AND 5 methods of the BSTree class namely: getRoot, getSize, insert, findKey, insertInformation).

b. BSTreeTester.java

2. **Final Submission (Due on Thursday, 25th May)**

   Files to submit:

   a. BSTree.java
   b. BSTreeTester.java
   c. SearchEngine.java
   d. (Optional Part 3) Add the extra credit methods to your BSTree.java file