Computer Science 331 — Winter 2016
Assignment #3 Solutions

Instructions

This assignment concerns lecture material that was introduced in this course on or before Wednesday, March 9.

Please **read the entire assignment** before you begin work on any part of it.

This assignment is due by 11:59 pm on Wednesday, March 23.

There are a total of 100 marks available, plus an additional 20 bonus marks.

Questions

A *mapping* is a partial function $f : K \mapsto V$ from a set K of "keys" to a set V of "values." Since this is a partial function, there is *at most* one value $f(k) \in V$ that is defined for each key $k \in K$. If the set of keys k such that $f(k)$ is defined is finite, and there is a useful "linear order" defined for the set K, then a binary search tree is one of several data structures that can be used to represent the mapping f. Note that a mapping is another way to describe the dictionary ADT described in class.

The Java Collections Framework contains an interface `SortedMap` describing such a mapping. The following questions will make use of the `SimpleSortedMap` interface (available on the Assignment 2 web page), a simplified version of `SortedMap`. The file `LinkedListMap.java` consists of a linked list based implementation of `SimpleSortedMap`, and `CountWords.java` contains a program the uses a `SimpleSortedMap` to identify the words and their frequencies from a text file.

The purpose of this assignment is to give you practice working with a binary search tree implementation of a mapping. In particular, you will:

- create an implementation of a binary search tree in which all algorithms are iterative (as opposed to recursive),
- create an Iterator object for your binary search tree class that implements an iterative version of an in-order traversal,
- work with the JCF `TreeMap` class, which contains an implementation of a red-black tree.

You will have to read background material on Java iterators and tree traversals. The file `traversals.pdf`, available on the Assignment 3 web page and the course Schedule page, contains a small amount of material on tree traversals. Further information on-line should be easy to find if you need it.

1. **(15 marks)** Prove that the following iterative search algorithm for a binary search tree is correct (i.e., it is partially correct and terminates). You must use a loop invariant (for partial correctness) and a loop variant (for termination).

```
$V$ search($T$, $key$)
PRECONDITION: {$T$ is a binary search tree with keys of type $E$ and associated values of type
  $V$; $key$ is of type $E$ }
  $v = null$
  $curr = T.root$
  {$P$ : - $T$ and $key$ have not been modified}
  {      - $v = null$ and $curr = T.root$}
  while $curr \neq null$ and $v = null$ do
    {$I$ : - $T$ and $key$ have not been modified}
    {      - ($key = curr.key$ and $v = curr.value$) or ($curr$ is the root of a possibly empty
    subtree $T'$ of $T$ and no node in $T$ but not in $T'$ has key $key$)}
    if $key = curr.key$ then
      $v = curr.value$
    else if $key < curr.key$ then
      $curr = curr.left$
    else
      $curr = curr.right$
    end if
  end while
  {$Q$ : ($v$ is the value with key $key$ in $T$) or ($v = null$ and $key$ is not a key in $T$)}
  if $v = null$ then
    throw KeyNotFoundException
  else
    return $v$
  end if
POSTCONDITION: {$T$ and $key$ are unchanged; if $key$ is a key in $T$ then the associated value
  is returned; if $key$ is not in $T$ then a KeyNotFoundException is thrown }
```

Solution: Note that assertions P, I, and Q have been added to the algorithm. No statements in the algorithm modify T or key, so this part of the assertions holds throughout.

We first prove that I is a loop invariant by induction. Let c denote the loop condition and S_2 denote the loop body.

(a) (Base case) True before first iteration: the first part of I is false (because $v = null$), but the second part is true. $curr$ is equal to T (allowed to be empty), and there are no nodes in T but not in T'

(b) If True before an iteration and the loop guard c holds, then True after the iteration:

- Before the iteration: because c holds, we have $v_{before} = null$, $curr_{before} \neq null$, and no node in T_{before} but not in T'_{before} has key equal to key.
- There are three cases to consider after the iteration:
  - if first if condition is true, then $curr_{after} = curr_{before}$ and the first clause of the or statement in I holds ($key = curr_{after}.key$ and $v = curr_{after}.value$).

  - If the second if statement holds, then $key < curr_{before}.key$ and $curr_{after} = curr_{before}.left$ (T'_{after} is the left subtree of T_{before}). By the binary search tree property, $key < curr_{before}.key$ means that key cannot be in the right subtree of $curr_{before}$, so if key is in T it has to be in T'_{after} and cannot be in any other node of T. Finally, $curr_{after}$ may be null, indicating that T'_{after} is empty. Thus, the second clause in the or of I is satisfied.
  - If the else statement is taken, then $key > curr_{before}.key$ and the argument is analogous to the previous case.

  Thus, in all cases I holds after an iteration of the loop body, and by induction I is a loop invariant.

We now prove partial correctness. Let S_1 denote the two assignment statements before the while loop, and S_3 denote the code after the while loop.

- $\{Precondition\}$ S_1 $\{P\}$: easy (omitted)
- upon termination of the loop, if the first clause of I holds, then v is the value with key key in T (first clause of Q is satsified). Otherwise, $curr = null$ and the loop invariant I implies that key is not a key in the tree T (second clause of Q is satisfied).
- $\{Q\}$ S_3 $\{Postcondition\}$: if $v = null$, then Q implies that key is not in the tree T, and the algorithm correctly throws the exception. If $v \neq null$, then Q implies that it is the value associated to the key key in T, and the algorithm correctly returns v.

Thus, the algorithm is partially correct.

To prove termination, we show that the while loop terminates by showing that $f(curr) = height(curr)+1$ is a loop variant. The function is integer valued and decreases by at least one after each iteration because $curr$ is replaced by the root of one of its subtrees, and the height of a subtree is strictly less than the height of a tree. If $f(curr) = 0$, then $height(curr) = -1$ and $curr$ must be an empty tree (i.e., null) and the while loop terminates.

2. **(10 marks)** Give pseudocode for an *iterative* algorithm for insertion into a binary search tree.

Solution: insert(T, key, $value$)

```
prev = null
curr = T.root
while curr ≠ null do
  if key = curr.key then
    throw KeyFoundException
  else if key < curr.key then
    prev = curr
    curr = curr.left
  else
    prev = curr
    curr = curr.right
  end if
end while
if prev = null then
```

```
    $T.root$ = new Entry($key, value$)
  **else if** $key.compareTo(prev.key) < 0$ **then**
    $prev.left$ = new Entry($key, value$)
  **else**
    $prev.left$ = new Entry($key, value$)
  **end if**
```

3. **(10 marks)** Give pseudocode for an *iterative* algorithm for an in-order traversal of a binary search tree. Your algorithm should make use of a stack.

   Solution: The idea is to use a stack to simulate the recursive calls by storing the nodes encountered during each call. The algorithm begins by following left child references until it reaches null, with each non-null node traversed begin pushed on the stack. Then, the stack is popped, the corresponding node is "traversed," and the process is repeated with the node's right child. This process repeats until the right child is null and the stack is empty.

```
**inOrder**($T$)
  initialize empty stack $S$
  $curr = T.root$
  **while** $curr$ is not null or $S$ is not empty **do**
    **while** $curr$ is not null **do**
      $S.push(curr)$
      $curr = curr.left$
    **end while**
    **if** $S$ is not empty **then**
      $curr = S.pop()$
      process node $curr$
      $curr = curr.right$
    **end if**
  **end while**
```

4. **(30 marks)** Implement a Java class called `BSTMap` that implements the `SimpleSortedMap` interface using a binary search tree. This class should satisfy the following:

   - The `search`, `insert`, `delete`, and `modify` functions must be implemented *iteratively*. You must *not* use a stack for any of these functions.
   - The `Iterator` returned by the `iterator` function must perform an in-order traversal of the binary search tree, accessing the keys in ascending order. You may use any implementation of the Stack ADT you like for your implementation. The `remove` operation, required by `Iterable`, does not need to be supported, and can just throw an `UnsupportedOperationException`.

   Be sure to document your class thoroughly using `javadoc` tags and assertions as appropriate. Both the quality of your implementation and documentation will be taken into account when grading this question. *5 of the 30 available marks for this question will be allocated to documentation.*

   Solution: See the Assignment 3 page on the course web site for a model solution.

5. **(15 marks)** Implement a Java class called `RBTMap` that implements the `SimpleSortedMap` interface using a red-black tree. You should do this by using the `TreeMap` class from the Java API, and using existing functions of `TreeMap` to implement the required functions from the interface. The Java Collections API documentation, available through the course web page, provides all the information you need to know about `TreeMap` to enable you to complete this question.

   Be sure to document your class thoroughly using `javadoc` tags and assertions as appropriate. Both the quality of your implementation and documentation will be taken into account when grading this question. *5 of the 20 available marks for this question will be allocated to documentation.*

   Solution: See the Assignment 3 page on the course web site for a model solution.

Note: The `CountWords.java` program will be used to test your `BSTMap` and `RBTMap` classes. This program accepts an optional 2nd command line parameter that allows you to choose which implementation of `SimpleSortedMap` is used. As long as your `BSTMap` and `RBTMap` correctly use this interface and provide a default constructor, the only required modification to `CountWords.java` should be uncommenting the lines in the program where `SimpleSortedMap` instance is created. The TAs will use the version of this program provided on the web site as-is, and it is your responsibility to make sure that your classes work properly with it.

6. **(15 marks)** Add a function `height` (this function may be recursive or iterative) to the `BSTMap` class that computes the height of the binary search tree. Write a program called `A3Q5.java` that computes some statistics on the height of m different binary search trees created by inserting n random integers into the tree, where n and m are command-line parameters. Run your program for values of n = 100, 1000, 10000, and 100000 and m = 100, and present a table with the following information for each value of n :

   - the minimum height of all 100 random trees
   - the maximum height of all 100 random trees
   - the average height of all 100 random trees
   - an upper bound on the expected height of a random binary search tree of size n, as per the average case analysis presented in class
   - a worst-case upper bound on the maximum height of a random red-black tree of size n.

   Solution: See the Assignment 3 page on the course web site for a model solution.

7. **(5 marks)** Is the data you obtained in the previous question what you would expect? Why or why not?

   Solution: For sufficiently large n, the average height of a random binary search tree of size n is bounded by $3\log_2 n$, so one expects that the obtained average height of all m binary search trees (as long as m is also sufficiently large) with size n would be less than this value.

   The main theorem of Red-Black Trees implies that the height of a Red-Black Tree with n is at most $2\log_2(n+1)$. For sufficiently large n, the average, maximum, and most likely the minimum heights of the binary search trees will be larger than this bound, indicating that even in the worst case a red-black tree will usually have smaller height than a binary search tree. Your height data should confirm this.

Note: Although you are not being asked to provide a test suite for this assignment, you are responsible for testing your classes as thoroughly as possible in order to make them as bug-free as possible. You should feel free to use JUnit or to add any auxiliary functions to your classes that will help with this process.

Bonus Questions

The following bonus question should only be attempted after the previous questions have been completed satisfactorily. While solutions to these questions are not required for full credit on this assignment, correct solutions can result in a grade of more than 100 %.

8. **(20 bonus marks)** Implement a Java class called `MyRBTMap` that implements the Java interface `SimpleSortedMap` using the red-black tree data structure from scratch, i.e., without using `TreeMap`. Modify the program `CountWords.java` to use your class as well.