95-771 – Data Structures and Algorithms for Information Processing
Homework #2

Due Monday, February 18, 2019 11:59:59 PM
Name *Mingqi(Ariel) Tan*

Topics: Queues, Red Black Trees and Dynamic Programming

Part 1. Queues and Red Black Trees (90%)

For Part 1, name your project RedBlackTreeSpellCheckerProject.

Write a spell checker that is based on a red black tree. The execution of the program, when run with shortWords.txt as a command line argument, will appear as follows:

java RedBlackTreeSpellChecker shortwords.txt
Loading a tree of English words from shortwords.txt.
Red Black Tree is loaded with 10 words.
Initial tree height is 3.
Never worse than 2 * Lg(n+1) = 6.918804721654216.

Legal commands are:
d display the entire word tree with a level order traversal.
s print the words of the tree in sorted order (using an inorder traversal).
r print the words of the tree in reverse sorted order (reverse inorder traversal).
c <word> to spell check this word.
a <word> add this word to tree.
f <fileName> to check this text file for spelling errors.
i display the diameter of the tree.
m view this menu.
! to quit.

c aaa
aaa Not in dictionary. Perhaps you mean
aa
>c aa
Found aa after 3 comparisons
>a mike
mike was added to dictionary.
>c mike
Found mike after 4 comparisons
>c mik
mik Not in dictionary. Perhaps you mean
mike

```
>a mik
mik was added to dictionary.
>d
Level order traversal
[data = aal:Color = Black:Parent = -1: LC = a: RC = aam]
[data = a:Color = Black:Parent = aal: LC = Aani: RC = aa]
[data = aam:Color = Black:Parent = aal: LC = aalii: RC = aardwolf]
[data = Aani:Color = Black:Parent = a: LC = A: RC = Aaron]
[data = aa:Color = Black:Parent = a: LC = -1: RC = -1]
[data = aalii:Color = Black:Parent = aam: LC = -1: RC = -1]
[data = aardwolf:Color = Red:Parent = aam: LC = aardvark: RC = mike]
[data = A:Color = Red:Parent = Aani: LC = -1: RC = -1]
[data = Aaron:Color = Red:Parent = Aani: LC = -1: RC = -1]
[data = aardvark:Color = Black:Parent = aardwolf: LC = -1: RC = -1]
[data = mike:Color = Black:Parent = aardwolf: LC = mik: RC = -1]
[data = mik:Color = Red:Parent = mike: LC = -1: RC = -1]
>c aalii
Found aalii after 3 comparisons
>!
Bye.
```

Here is another example. In this case we are using the much larger file ‘words.txt’. The file ‘words.txt’ is on the course schedule and consists of all of the words in the English language (in sorted order). We are also using the program’s ability to spell check a text file.

```
java RedBlackTreeSpellChecker words.txt
Loading a tree of English words from shortwords.txt.
Red Black Tree is loaded with 235923 words.
Initial tree height is 32.
Never worse than 2 * Lg( n+1) = 35.69562343012844.

Legal commands are:
d   display the entire word tree with a level order traversal.
s    print the words of the tree in sorted order (using an inorder traversal).
r    print the words of the tree in reverse sorted order (reverse inorder traversal).
c   <word> to spell check this word.
a   <word> add this word to tree.
f   <fileName> to check this text file for spelling errors.
i   display the diameter of the tree.
m  view this menu.
!   to quit.

>f shortwords.txt
No spelling errors found.
>f badwords.txt
```

'This' was not found in dictionary.
'sume' was not found in dictionary.
'mispelled' was not found in dictionary.
'wirds' was not found in dictionary.
'And' was not found in dictionary.
>!
Bye.

Note that the "a" command will not add a word to the dictionary if the word is already present in the dictionary. So, for example, if the user enters >a hello the output will be "The word 'hello' is already in the dictionary.
There will be three data files used for testing. There is a small word list called 'shortwords.txt' and another large one called 'words.txt'. A third file, used for testing the spell checker's 'f' option, is called 'badwords.txt'. The 'badwords.txt' file will contain a few poorly spelled words. The original word list file name is passed to the program on the command line. Be sure to test your solution using the data found in shortwords.txt and words.txt. Along with badwords.txt, these files are found on the schedule.

On the class schedule is the javadoc specifications for three classes: Queue.java, RedBlackNode.java, and RedBlackTree.java. You are required to implement all of the methods described there. If you need additional methods, feel free to add them.

The RedBlackTree class is an implementation of a balanced binary search tree in Java. You have seen in class how a red black search tree works. Following the Javadoc specifications given, you are to write an implementation of the RedBlackTree class. Once you have completed your implementation, you should examine each method you have written and state the worst and best case Big-Theta functions (as you did in the first assignment). Include those big theta functions in your comments.

Note the method levelOrderTraversal().The method levelOrderTraversal() must use a queue to perform the traversal. So, you will need to write a queue class. Your queue class must be implemented as a circular array (as described in class and on the slides.). Be sure to use an array of Object references rather than Strings. The queue class will begin with an array of size 5. If the array overflows (which it will in this homework), your program will automatically double the size to 10. If this too overflows, increase the array size to 20, and so on. This increasing of the array size will be transparent to the caller. After allocating room for the new array, you will need to copy the old array contents to the new array.

An algorithm for a level order traversal is provided on the course slides. Details concerning red black trees may be found in the Javadoc provided. You should carefully follow the pseudocode provided (from CLR) and translate the pseudocode to Java.

Include a class called RedBlackTreeSpellChecker.java that has a main() routine that interacts with a user as shown above. RedBlackTreeSpellChecker.java will create the Red Black tree and fill it with words from the file specified on the command line. It will

then allow the user to check a word (use “c” word) or add a word (use “a” word) or print the tree (use “d”, “s” or “r”) or spell check a text file (use “f” “filename”) or quit (use “!”). The “i” command will be used to compute the current diameter of the tree. The diameter of a tree is the length of the longest path between any two nodes in the tree. The menu itself may be redisplayed by using “m” for “menu”. If a word is not found on a spell check then a word that is near by in the tree should be displayed. How you compute “near by” is up to you. No sophisticated approach is required here – define “nearby” in some reasonable manner. On all successful searches, the number of comparisons will be displayed. Also, note above the initial lines of output showing the height of the tree and the value of 2 * Lg (n + 1) where n is the number of tree entries. This is a base two log and represents the maximum height of a red black tree. You need to compute this value in your program.

Be very careful to use the exact same names as we did so that your RedBlackTree class will work with our test program.

Within the Javadoc of the RedBlackTree class, there is an example main() routine provided as a test case. Your main routine will generate the same style of output, i.e., for each node holding data in the red-black tree, be sure to show its color and value and the data in its parent, left child and right child nodes.

Part 2. Dynamic Programming (10%)

For Part 2, name your project WorldSeriesOddsProject.

Later in the course, we will visit the Floyd-Warshall all pairs shortest path algorithm. That algorithm uses “dynamic programming” to compute shortest paths in a graph. This exercise will introduce you to some of the fundamental ideas behind dynamic programming.

This assignment is based on a discussion of Dynamic Programming found in "Data Structures and Algorithms", by Aho, Hopcroft and Ullman.
ISBN 0-201-00023-7.

In the World Series of Major League Baseball, two teams play and the first to win 4 games is the World Champion team. Let's suppose that the two teams are equally matched and each has a 50% chance of winning any one game. Let's label the two teams as A and B and let P(i,j) be the probability that if A needs i games to win and B needs j games to win that A will eventually win the series.

For example, if the Pirates have won 2 games and the Cardinals have won 1, the probability that the Pirates will win the series is P(2,3) = 11/16. We will see how to compute that probability in a moment.

Note that if j > 0 then P(0,j) = 1. Why? Since the Pirates need 0 games to win, the series is over and the Pirates have won.

Also, if i > 0 then P(i,0) = 0. Why? Since the Cardinals need no games to win, the series is over and the Cardinals have won. The probability that the Pirates will win is 0.

In general, some thinking arrives at the following recurrence relation:

P(i,j) = 1 if i = 0 and j > 0
= 0 if i > 0 and j = 0
= (P(i-1,j) + P(i,j-1))/2 if i > 0 and j > 0

The reasoning behind the bottom part of the recurrence is this: Each team is equally likely to win. So P(i-1,j) is the probability that the Pirates will win the series after winning the next game and P(i,j-1) is the probability that the Pirates will win after losing the next game. Since each team is equally likely to win, we sum these probabilities and compute an average by dividing by 2.

When applying the recurrence, either i or j must go to 0. They both cannot go to 0 at the same time.

Write a recursive routine that computes P(i,j). This recursive routine should be based on the recurrence relation shown above. In other words, given the recurrence, the recursive routine should be a simple translation to Java. Use it to compute the following values and fill in the chart below.

	Value	Time in seconds (approximately)
P(2,3) =	0.6875	0.002
P(4,7) =	0.828125	0.002
P(7,6) =	0.38720703125	0.003
P(10, 12) =	0.6681880950927734	0.006
P(20, 23) =	N/A	N/A
P(30,15) =	N/A	N/A
P(50,40) =	N/A	N/A

You may have had some trouble computing the above probabilities. The problem is that we are computing the solutions to sub-problems over and over again. If you ran out of time, place question marks in the appropriate blanks on the chart.

With a dynamic programming approach, we will build a two-dimensional array that holds the values of all previously computed results. We can use these previously computed values to compute the next probability.

Suppose that table T is a two dimensional array. The top row of this table (assuming the top left corner is at (0,0)) can be used to hold the probabilities when i = 0. Each of these would be 1.0. The leftmost column of this table, can be used to hold the probabilities when j = 0. Each of these would be 0.0. Using the values in the table, one can compute the remaining probabilities in a bottom-up fashion. T(i,j) can be computed by referencing T(i-1,j) and T(i,j-1). Both of these values were computed earlier.

Write a solution that computes P using dynamic programming. Again, complete the following table.

	Value	Time in seconds (approximately)
P(2,3) =	0.6875	0.002
P(4,7) =	0.828125	0.002
P(7,6) =	0.38720703125	0.002
P(10, 12) =	0.6681880950927734	0.002
P(20, 23) =	0.678015521848138	0.002
P(30,15) =	0.011314420602957398	0.002
P(50,40) =	0.1445480403480301	0.002

We computed P in two ways – by using recursion and by dynamic programming. Which of these was faster and why?

The recursive method computes a lot of same probabilities over and over again. However, the dynamic programming method computes the probabilities and store them for next time use.

*Each probability is only computed once that the time complexity is fixed at θ(m*n). Therefore, the second way is much faster.*

Submission

Submit one zipped folder to Canvas. Name it Project2.zip. It will contain RedBlackTreeSpellCheckerProject and WorldSeriesOddsProject. In addition, included within Project2.zip, submit a copy of this Word document. The document will have your name, two completed charts, and a short paragraph describing the run time performance of the two programs used to compute probabilities.

Where it makes sense, document your code with pre- and post-conditions. Use good variable names and good indentation. Provide a Big Theta analysis for your Red Black Tree methods.