

CISC-235  
Assignment 2  
February 5, 2017

In this assignment you will explore some properties of Red-Black trees and compare them to “plain” Binary Search Trees.

**Part 1:**

Implement a **BinaryTreeVertex** class in which each object has at least these attributes:

- value: a value
- left (or left\_child): a pointer to another BinaryTreeVertex object
- right (or right\_child): a pointer to another BinaryTreeVertex

and other attributes as needed for your implementation of Red-Black trees.

(Note: if you are completing your assignment in Python it is not acceptable to implement your vertices as Python dictionaries. Part of the intended benefit of this assignment is to consolidate your programming skills with objects and explicit pointers.)

Implement a **BinarySearchTree** class in which each object has at least this attribute:

- root: a pointer to a BinaryTreeVertex object

and any other attributes that will help with the completion of this assignment.

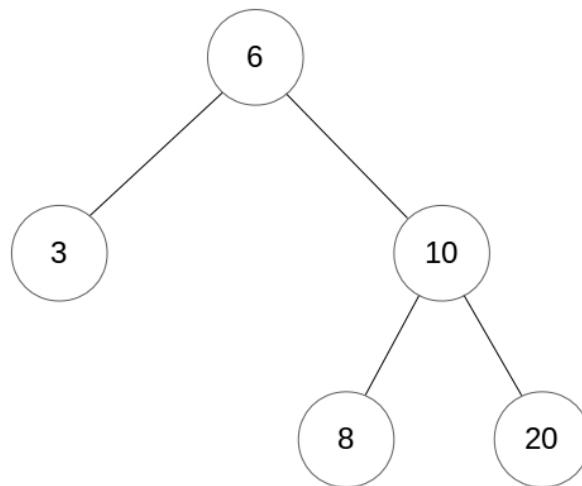
Each BST object must also have an **Insert** method or function. You may assume that the values to be stored in the tree are integers.

Your BST objects must also have a **SearchPath** method that returns a list of all the values visited on the search path. For example, on a particular tree T, T.SearchPath(12) might return

8, 20, 15, 14, 10, 12

The purpose of this modified Search method is to allow checks to determine that the tree is properly arranged.

Your BST objects must also have a **Total\_Depth** method or function that computes the sum of the depths of all vertices in the tree, where the **depth** of a vertex is its level in the tree (the root has depth 0). For example, in this tree



the total depth is  $0 + 1 + 1 + 2 + 2 = 6$

The Total Depth of a tree is important because if we divide it by  $n$  (the number of vertices) we get the average depth, which is 1 less than the average number of vertices visited on a search of the tree (assuming all values in the tree are equally likely to be the target of a search). Thus by comparing the Total Depth of two trees that store the same set of values, we can determine which tree will have better search performance.

Your **Total\_Depth** method or function should run in  $O(n)$  time. Hint: the root is at depth 0. If a vertex knows its own depth, it can tell its children what their depth is. Each vertex can add its own depth to the total depth reported by its children, and return that sum. The value returned by the root will be the total depth of the tree.

All of your methods can be recursive or iterative.

Test your methods by starting with an empty tree and inserting the values 6, 10, 20, 8, 3 (which should produce the tree shown above), then search for each of the values and confirm that the search paths are correct (for example the search path for 8 should be 6, 10, 8)

Also run your **Total\_Depth** method on this tree and confirm that it gives the correct result.

## Part 2:

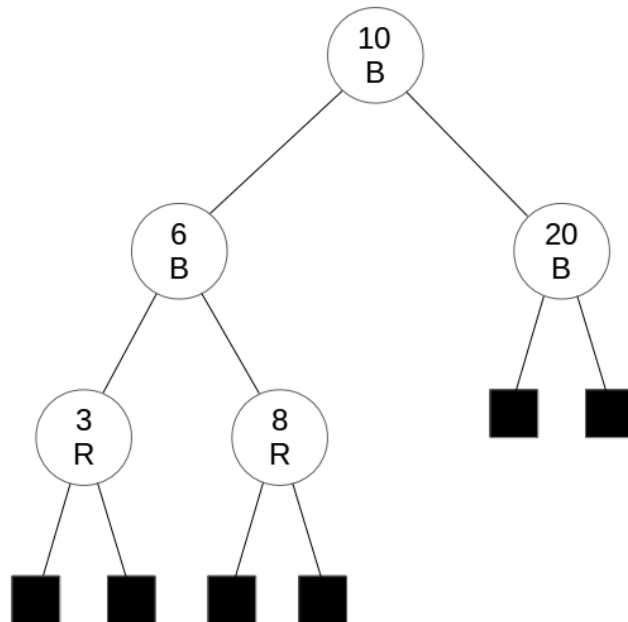
Repeat Part 1, but this time with Red-Black Trees. Each vertex will need an additional field to store the vertex's colour (in real world applications this is usually done with a single bit, but you can use any data type you choose for this field).

Your `Total_Depth` method will be unchanged from Part 1, but your `Search` method should be modified to also display the colour of each vertex encountered (this will give evidence that the tree is properly coloured). Naturally your `Insert` method will need to incorporate the Red-Black colouring and balancing operations. In addition, you will need some way to distinguish leaf vertices from internal vertices. Please refer to the text and the course notes for pseudocode versions of the `Insert` operations for Red-Black trees. It is important to note that the text uses vertex objects that contain pointers to their parents. If you choose to follow the pseudocode given in the text you will need to add this attribute to your vertex objects.

As in Part 1, you have the option of using recursion or not.

Test your methods by starting with an empty tree and inserting the values 6, 10, 20, 8, 3

This should produce the tree



I have drawn the leaves of the tree as black squares. In your implementation you should not include them in the `Total Depth` calculation.

Use your `SearchPath` method to confirm that the tree is correctly structured.

To test your single rotation, start with an empty tree and insert the values 6, 1, 10, 20, 30. I will let you work out what the tree should look like. Use your SearchPath method to confirm that the tree is correctly structured.

To test your double rotation, start with an empty tree and insert the values 6, 10, 8, 20, 3. Confirm that the tree is correctly structured.

### Part 3:

#### Experiment 1:

Your first experiment is intended to explore the frequency of rotation operations as a Red-Black tree grows.

For this experiment, modify your code to count the number of rotations performed. Then build a Red-Black tree from a random permutation of the set  $\{1, 2, \dots, 1000\}$ . After each 100 insertions (call these checkpoints), compute the number of rotations performed **since the last checkpoint**. Repeat this process 500 times and compute the average number of rotations for each checkpoint.

Before doing this experiment, record your expectation: Will the number of rotations between checkpoints increase, decrease, or remain fairly constant? After your experiment, tabulate or graph your results, and discuss whether your results confirm your expectation.

Your table might look something like this:

Checkpoint	Average Number of Rotations
100	18.3
200	21.2
...	...

## Experiment 2:

The *raison d'être* for Red-Black trees is that they guarantee  $O(\log n)$  height. In this experiment you will compare Red-Black trees to simple Binary Search trees. The measurement for comparison will be the Total Depth, as explained in the implementation section. The hypothesis to be tested is that for a random sequence of values, the Total Depth of the Red-Black tree should be less than the Total Depth of the simple Binary Search tree.

To conduct your experiments, use the following procedure:

For  $n$  in {1000, 2000, 4000, 8000, 16000}:

    # conduct 500 trials

    For  $i = 1 \dots 500$ :

        Generate a random permutation of the set  $\{1, 2, \dots, n\}$

        Build a Binary Search Tree by inserting the values in the permutation

        Build a Red-Black Tree by inserting the values in the permutation

        Compute the Total Depth of the two trees

This will generate a lot of data – you will need to summarize. For each value of  $n$ , you should determine the approximate distribution of the Total Depth differences. One way to do this is to compute the ratio  $R = \frac{BST \text{ Total Depth}}{RB \text{ Total Depth}}$  for each trial. If  $R$  is small the

BST is more compact than the RB tree, and if  $R$  is large the RB tree is more compact than the BST. On the other hand, if  $R$  is close to 1 then the two trees are similar in structure.

Then your summary might include a table such as this:

$n$	$R < 0.5$	$0.5 \leq R < 0.75$	$0.75 \leq R \leq 1.25$	$1.25 < R \leq 1.5$	$R > 1.5$
1000	2 %	8 %	15 %	50 %	25 %
2000	1 %	10 %	10 %	54%	25 %
...					

The numbers in the table shown here are **completely made up** – your results will not necessarily show the same pattern as these.

Discuss the results of your comparisons. Do they support the claim that Red-Black Trees are superior to simple Binary Search Trees? Does the difference (if any) become more or less significant as  $n$  becomes larger?

For example, in the table shown above both values of  $n$  indicate that the BST Total Depth is at least 25% greater than the RB Total Depth in at least 75% of the trials. If that continues to be true for larger values of  $n$ , it is evidence that RB Trees are usually more

compact than simple Binary Search Trees.

**Logistics:**

You may complete the programming part of this assignment in Python, Java, C or C++.

Your implementation must include classes as specified. If you choose Python, you may not use the native Dictionary type to implement your vertex and tree objects.

You must submit your source code, properly documented according to standards established in CISC-121 and CISC-124, and taking into consideration any feedback you received on Assignment 1. You must also submit a PDF file summarizing the results of your experiments and containing your conclusions regarding the comparison of Red-Black Trees and simple Binary Search Trees. Both files must contain your name and student number, and must contain the following statement: "I confirm that this submission is my own work and is consistent with the Queen's regulations on Academic Integrity."

You are required to work individually on this assignment. You may discuss the problem in general terms with others and brainstorm ideas, but you may not share code. This includes letting others read your code or your written conclusions. The course TAs will be available during office hours to advise and assist you regarding this assignment.

The due date for this assignment is 11:59 PM, February 19, 2017.