CISC-235
Assignment 3
Due 11:59 PM, March 8, 2020

In this assignment you will explore some properties of Binary Search Trees.

Your trees will be used to store and operate on sets of integers. In accordance with the formal definition of a set, your tree will never contain duplicate values. However, your program must be prepared for situations in which the user will attempt to add a value that is already present, or remove a value that is not present.

Clarification: how you handle these situations is up to you. You can print a message, return a particular value, raise an exception, or any other reasonable action.

Part 1:

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

- value: integer
- left (or left_child): pointer to another BinaryTreeVertex object
- right (or right_child): pointer to another BinaryTreeVertex object

plus other attributes that you may choose to add to help you complete this assignment.

(Note: if you are completing your assignment in Python it is **not** acceptable to implement your vertices as Python dictionaries. One of the learning outcomes of this assignment is to consolidate your programming skills with objects (classes) 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.

Your BinarySearchTree object must have the following methods or functions:

Search(x) The search function must return a pointer to the vertex containing the search value. If the search value is not in the tree, the function must return a null pointer.

Insert(x) The insert function must maintain proper binary search tree ordering of the values stored in the tree.

Delete(x) The delete function must remove the specified value from the tree. The tree

must maintain proper binary search tree ordering of the values remaining after the deletion.

Clarification: these functions can be recursive or iterative.

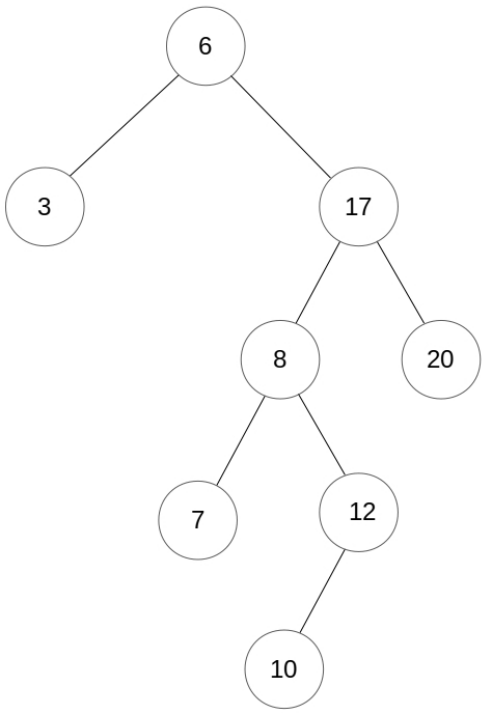
Your BST objects must also have a **SearchPath(x)** method that returns a list of all the values visited on the search path. For example, on a particular tree T, T.SearchPath(12) might return

8, 20, 15, 14, 10, 12

This would indicate that 8 is the value stored in the root of the tree, 20 is the root's right child, and so on down to 12

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

Your BST objects must also have an **Average_Depth** method or function that computes the average 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 1). For example, in this tree



the average depth is $\frac{1+2+2+3+3+4+4+5}{8} = \frac{24}{8} = 3$

The average depth of a tree is important because it is the average number of vertices visited on a search of the tree (assuming we only search for values in the tree and all values in the tree are equally likely to be the target of a search). Thus the average depth of a tree gives us a measure that is directly related to the average search time for that tree.

Your Average_Depth method or function should run in O(n) time. Hint: write a method to compute sum of the depths of the vertices. The root is at depth 1. 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 sum of all depths in the tree.

Your BST objects must also have a **Max_Depth** method or function that computes the number of levels in the tree. For example, in the tree shown above the maximum depth is 5.

The Max Depth of a tree is important because it tells us the worst-case search time for the tree.

We can imagine a tree where the Total Depth is low but the Max Depth is much higher due to one long path. This is significant because it tells us that the worst-case search time for that tree is much higher than the average-case.

Your Max_Depth method or function should run in O(n) time.

All of your methods can be recursive or iterative.

Demonstrate the correctness of your methods by starting with an empty tree and inserting the values 6, 3, 17, 8, 20, 7, 12, 10 in that order (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, 17, 8.

Also run your Average_Depth and Max_Depth methods on this tree and confirm that they give the correct result.

Now delete the value 17, and show that the search paths are correct after the deletion.

Part 2:

Experiment 1: Binary Search Trees Without Deletion

One recognized problem with Binary Search Trees is that in the worst case they can have O(n) levels. In this experiment you will measure the Average Depth and Max Depth of randomly generated Binary Search Trees.

To conduct your experiments, use the following procedure:

```
# for different sizes of the data set
For  n in {250, 500, 1000, 2000, 4000, 8000, 16000}:
    # conduct 500 trials
    For i = 1 .. 500:
        Generate a random permutation of the set {1,2, ..., n}
        Build a Binary Search Tree by inserting the values in the order given
            by the permutation
        Compute the Average Depth and Max Depth of the tree
    Compute the average Average Depth and average Max Depth for n
```

Clarification: the average Average Depth =

$$\frac{\text{sum of the Average Depth for the 500 trees with n vertices}}{500}$$

Compute the maximum Average Depth and maximum Max Depth for n

Consider these hypotheses:

Hypothesis 1: The average Average Depth of the trees on n vertices generated in this way is in O(log n)

Hypothesis 2: The maximum Average Depth of the trees on n vertices generated in this way is in O(log n)

Hypothesis 3: The average Max Depth of the trees on n vertices generated in this way is in O(log n)

Hypothesis 4: The maximum Max Depth of the trees on n vertices generated in this way is in O(log n)

Do the results of your experiments support these hypotheses?

Clarification: After running this experiment you will have computed four values for each value of n. You may want to organize these data in a table like this:

n	Average Average Depth	Maximum Average Depth	Average Max Depth	Maximum Max Depth
250	AA_{250}	MA_{250}	AM_{250}	MM_{250}
500	AA_{500}	Etc.		
1000	AA_{1000}			
...	...			
16000	AA_{16000}			

where the AA_{250}, MA_{250} (etc) values are the values you computed during your experiment.

This is not a requirement but it is a convenient way to organize your results.

You are not required to do sophisticated statistical analyses on your collected data. To conclude that your experimental data support the hypothesis that some observed quantity $X(n)$ is in $O(\log n)$, it is sufficient (in this assignment) to find a value k such that $k * \log n$ is within 10% of $X(n)$ - that is,

$$0.9 * X(n) \leq k * \log n \leq 1.1 * X(n) \text{ for all the values of } n.$$

(**Clarification: X(n) represents the relevant measure for each of the hypotheses. The purpose of these computations is to determine if there is a function of the form k*log n**

that is "pretty close" to all of your measurements for this quantity.)

That simplifies to "Can we find a value of k that satisfies

$$\frac{0.9 * X(n)}{\log n} \leq k \leq \frac{1.1 * X(n)}{\log n}$$

for all the values of n?"

If no such value exists, you can (in this assignment) conclude that your experimental data does not support the hypothesis.

Clarification: Consider the first hypothesis. for each value of n, plugging in your computed values for AA_n will give a set of constraints on k:

$$\frac{0.9 * AA_{250}}{\log 250} \leq k \leq \frac{1.1 * AA_{250}}{\log 250}$$

$$\frac{0.9 * AA_{500}}{\log 500} \leq k \leq \frac{1.1 * AA_{500}}{\log 500}$$

…

$$\frac{0.9 * AA_{16000}}{\log 16000} \leq k \leq \frac{1.1 * AA_{16000}}{\log 16000}$$

The hypothesis is supported if there is a value of k that satisfies all of these constraints. If there is no such k, then your experimental results do not support this hypothesis.

Also, if you find that your computer is not able to deal with trees with 16000 vertices in a reasonable amount of time, feel free to cap your trees at a smaller size.

Clarification: There is not necessarily a "right answer" in the sense that everyone must get exactly the same result. Because you are all generating random trees, the outcomes could be quite variable. Your answer must be based on your own experimental results.

Experiment 2: Binary Search Trees With Deletion

In this experiment you will measure the Total Depth and Max Depth of randomly generated Binary Search Trees, with some values being deleted during the construction sequence.

To conduct your experiments, use the following procedure:

for different sizes of the data set
For n in {250, 500, 1000, 2000, 4000, 8000, 16000}:
    # conduct 500 trials
    For i = 1 .. 500:
        Let x = $\frac{11 * n}{10}$
        Generate a random permutation of the set {1,2, ..., x}
        Build a Binary Search Tree by inserting the values in the order given by the permutation
        Generate a random sample of $\frac{n}{10}$ values from {1, …, x}
        Delete the values in the random sample from the tree
        Compute the Average Depth and Max Depth of the final tree
    Compute the average Average Depth and average Max Depth for n
    Compute the maximum Average Depth and maximum Max Depth for n

Clarification: the set of values to be deleted from the tree should all be distinct. This guarantees that your final trees for each value of n will all have the same number of vertices. One simple way to generate a random set of $\frac{n}{10}$ distinct values to delete is to generate a random permutation of $\{1, \ldots, x\}$ and then use the first $\frac{n}{10}$ values in that permutation.

Consider and discuss the same four hypotheses for the trees created in this experiment.

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 and taking into consideration any feedback you received on Assignments 1 and 2. You must also submit a PDF file summarizing the results of your experiments and containing your conclusions. 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.