UNIVERSITY OF SASKATCHEWAN
Department of Computer Science
176 Thorvaldson Building
110 Science Place, Saskatoon, SK, S7N 5C9, Canada
Telephine: (306) 966-4886, Facimile: (306) 966-4884

Assignment 10 – Solutions and Grading

Object-Oriented Binary Search Tree ADT

Date Due: Tuesday, 17 August 2021, 11:59pm **Total Marks: 50**

Question 0 (5 points):

Purpose: To force the use of Version Control

Degree of Difficulty: Easy

Version Control is a tool that we want you to become comfortable using in the future, so we'll require you to use it in simple ways first, even if those uses don't seem very useful. Do the following steps.

1. Create a new PyCharm project for this assignment
2. Use `Enable Version Control Integration...` to **initialize** Git for your project.
3. Download the Python and text files provided for you with the Assignment, and **add** them to your project.
4. Before you do any coding or start any other questions, make an initial **commit**.
5. As you work on the assignment, use Version Control frequently at various times when you have implemented an initial design, fixed a bug, completed the question, or want to try something different. Make your most professional attempt to use the software appropriately.
6. When you are finished your assignment, open PyCharm's Terminal in your Assignment 4 project folder, and enter the command:

   ```
   git --no-pager log
   ```

   WARNING: Copy/Paste from PDFs can often add unwanted spaces. There are no spaces between dashes (-) in the above command!
7. Copy/Paste the output of the above command into a text file

Notes:

- Several Version Control videos are available on Moodle via the Lecture Videos link.
- No excuses. If you system does not have Git, or if you can't print the log as required, you will not get these marks. Installation of Git on Windows 10 machines is outlined in the "*Step-by-Step Windows 10 - PyCharm UNIX command-line*" video. Git is included in base installations of Linux & MacOS.
- If you are not using PyCharm as your IDE, you are still required to use Git. See the Version Control videos on Moodle that cover using Git on the command line & via GUI.

What Makes a Good Commit Message?

A good commit message should have a **subject line** that describes *what* changed, a **body** that explains *why* the change was made, and any other *relevant* information. Read "*Writing a Good Commit Message*" linked on Moodle for more guidance.

Examples of GOOD commit messages:

```
Initial commit for Assignment

* copied starter files over from Moodle
```

```
Completed Question 2

* full functionality complete after testing with generated files
```

```
Fixed a bug causing duplicate print messages

* extra print statements were being produce in loop
* updated logical condition to fix
```

Examples of BAD commit messages:

```
fixes
```

```
add tests
```

```
updates
```

What to Hand In

After completing and submitting your work for the Assignment, **follow steps 6 & 7 above** to create a text file named `a10-git.txt`. Be sure to include your name, NSID, student number, course number, and lecture section at the top of your `a10-git.txt` file.

Evaluation

- 5 marks: The log file shows that you used Git as part of your work for this assignment.

  For full marks, your log file contains
  - Meaningful commit messages.
  - A minimum of 4 separate commits.

Question 1 (20 points):

Purpose: To study the Binary Search Tree operations.

Degree of Difficulty: Easy to Moderate.

Restrictions: This question is homework assigned to students and will be graded. This question shall not be distributed to any person except by the instructors of CMPT 145. Solutions will be made available to students registered in CMPT 145 after the due date. There is no educational or pedagogical reason for tutors or experts outside the CMPT 145 instructional team to provide solutions to this question to a student registered in the course. Students who solicit such solutions are committing an act of Academic Misconduct, according to the University of Saskatchewan Policy on Academic Misconduct.

Task

On Moodle, you will find 2 files:

- `a10q1.py`
- `a10q1_test.py`

The first file is a partial implementation (with a few bugs thrown in) of the Binary Search Tree operations. The second file is a test script to help you complete and fix the implementation of these operations. Currently, of 100 test cases, only 5 test cases are successful.

Your task is to complete and correct the operations, by studying how they should work, and by fixing the given code.

You are allowed to use the course readings and the lecture notes to complete this question. However, you should not consult any other student, nor any other expert or tutor for assistance in this exercise.

What to Hand In

- A file named `a10q1.py` containing the corrected and completed implementation. **Note: If you do not name the script appropriately, you will get zero marks.**

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- Your solution to this question must be named `a10q1.py`, or you will receive a zero on this question.
- 20 marks: When our scoring script runs using your implementation of `a10q1.py`, no errors are reported. Partial credit will be allocated as follows:

Number of tests passed	Marks	Comment
0-5	0	the given a10q1 script already passes 5 tests total
5-45	5	
46-69	10	
70-79	12	Studying the lecture notes gets you here at least
80-89	15	
90-98	18	this is actually pretty good
99-100	20	

For example, if your implementation passes 78 tests, you'll get 12/20. If your implementation passes 85 tests, you'll get 15/20. Our test script is based on the test scripts distributed with the assignment, but may not be exactly the same.

Solution: The given script has the following errors and partial implementations (all line numbers refer to the given script):

- `member_prim()`

  Line 50: Return value

  Line 51: Base case condition

  Line 52: Return value

  Line 53: Recursive case condition

  Line 54: Not an error! Unneeded, but harmless.

  Line 57: Mutates the tree in a corrupting way.

- `insert_prim()`

  Line 79: Return value is the class not an object.

  Line 82: Return value

  Line 87: Return value

  Line 88: Return value

  Line 92: Tree needs to be updated

  Line 93: Return value

  Line 94: Return value

- `delete_prim()`

  Line 124: Recursive case condition

  Line 125: tuple assignment clobbers tnode

  Line 130: Incomplete case

  Line 160: Return value

Marking guidelines:

- Students were given an implementation with some errors introduced, and some parts incomplete.
- To grade the implementation, drop the files `treenode.py`, `test145.py`, `a10q1_test.py` into the student's folder, and run `python` as follows:

```
UNIX$ python a10q1_test.py
----------------- Summary ------------------
A10Q1: 20/20
Tests passed: 100/100
```

  Transfer the score to Moodle.

Question 2 (10 points):

Purpose: To implement an Object Oriented ADT based on the Primitive binary search tree operations from Question 1.

Degree of Difficulty: Easy to Moderate.

Restrictions: This question is homework assigned to students and will be graded. This question shall not be distributed to any person except by the instructors of CMPT 145. Solutions will be made available to students registered in CMPT 145 after the due date. There is no educational or pedagogical reason for tutors or experts outside the CMPT 145 instructional team to provide solutions to this question to a student registered in the course. Students who solicit such solutions are committing an act of Academic Misconduct, according to the University of Saskatchewan Policy on Academic Misconduct.

Task

This question assumes you have completed A10Q1. You can get started on this even if a few test cases fail for A10Q1.

On Moodle, you will find 2 files:

- `a10q2.py`
- `a10q2_test.py`

The first file is a partial implementation of the TBase ADT. The second file is a test script to help you complete the implementation of these operations. Currently, of 20 test cases, only 1 test case is successful.

Your task is to complete and correct the operations, by studying how they should work, and by fixing the given code.

You are allowed to use the course readings and the lecture notes to complete this question. However, you should not consult any other student, nor any other expert or tutor for assistance in this exercise.

TBase ADT Description

If you remember how we worked with node chains in Assignment 5, and LLists in Assignment 6, this will feel very familiar. The TBase ADT is simply a primitive binary search tree wrapped in a nice object with an interface that's a bit simpler the the functions from A10Q1. The good news is that because those functions are now all working, all you have to do is get the TBase methods to call those primitive functions properly.

The TBase object stores the root of a binary search tree, in much the same way that the LList record stored the head of a node chain. A more accurate analogy would be to the Stack ADT from Chapter 14, because we only stored the reference to the front of the node chain (the *top*). The TBase object also stores the number of values currently stored, in a private attribute. To help make this clear, the `__init__()` is given, and there is no need to add to it.

The TBase class in `a10q2.py` has five methods outlined with doc-strings, but the methods do not yet do what they should do. Here's what they should do:

- `is_empty(self)` Is the collection empty? Returns a boolean answer.
- `size(self)` How many unique values are stored? Returns an integer answer.
- `member(self, value)` Is the given value in the collection? Returns a boolean answer.
- `add(self, value)` Add the given value to the collection. Returns True if the value was added, and False only if the value is already there.
- `remove(self, value)` Remove the given value from the collection. Returns True if the value was removed, and False only if the value was not there to be removed.

This exercise should feel familiar, especially, `is_empty(self)` and `size(self)`. The remaining three methods should call the corresponding primitive functions from A10Q1 correctly. You do not need to implement them separately.

What to Hand In

- A file named `a10q2.py` containing the corrected and completed implementation. **Note: If you do not name the script appropriately, you will get zero marks.**

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- Your solution to this question must be named `a10q2.py`, or you will receive a zero on this question.
- 10 marks: When our scoring script runs using your implementation of `a10q2.py`, no errors are reported. Partial credit will be allocated as follows:

Number of tests passed	Marks	Comment
0-1	0	the given a10q2 script already passes 1 test total
2-9	5	
10-18	8	
19-20	10	

  For example, if your implementation passes 17 tests, you'll get 8/10. Our test script is based on the test scripts distributed with the assignment, but may not be exactly the same.

Solution: A model solution is found in `a10q2.py`. Here are some highlights.

- `is_empty(self)`

```
    def is_empty(self):
        """
        Purpose:
            Check if the TBase object is empty.
        Return
            :return: True if it's empty, False otherwise
        """
        return self.__size == 0
```

- `size(self)`

```
    def size(self):
        """
        Purpose:
            Return the number of elements in the TBase object
        Return
            :return: an integer indicating the number of elements
        """
        return self.__size
```

- `member(self, value)`

```
    def member(self, value):
        """
        Purpose:
            Check if target is stored in the TBase object.
        Pre-Conditions:
            :param value: a value,
             Must be comparable to the other values in the TBase object
        Return
            :return: True if value is in the object
        """
        return prim.member_prim(self.__root, value)
```

- add(self, value)

```
    def add(self, value):
        """
        Purpose:
            Store the given value in the TBase object.
        Pre-Conditions:
            :param value: a value
             Must be comparable to the other values in the TBase object
        Return
            :return: True if value was added
                     False if it was already there
        """
        flag, tn = prim.insert_prim(self.__root, value)
        if flag:
            self.__size += 1
            self.__root = tn
        return flag
```

- remove(self, value)

```
    def remove(self, value):
        """
        Purpose:
            Remove the given value from the TBase object.
        Pre-Conditions:
            :param value: a value
             Must be comparable to the other values in the TBase object
        Return
            :return: True if value was removed
                     False if the value was not there at all
        """
        flag, tn = prim.delete_prim(self.__root, value)
        if flag:
            self.__root = tn
            self.__size -= 1
        return flag
```

Marking guidelines:

- It's certainly possible for students to have poor implementations of these methods. For example, calls to A10Q1 might be repeated, or the return values handled incorrectly or inelegantly.
  - There is no place here to deduct marks for that. A flaw in the grading scheme.
- To grade the implementations:
  1. Drop the files `treenode.py`, `test145.py`,`a10q2_test.py` into the student's folder, and run `python` as follows:

     ```
     UNIX$ python a10q2_test.py
     ----------------- Summary ------------------
     A10Q2: 10/10
     Tests passed: 20/20
     ```

     Transfer the score to Moodle.

Question 3 (10 points):

Purpose: To implement the same behaviour as TBase, but without using A10Q1.

Degree of Difficulty: Easy.

Restrictions: This question is homework assigned to students and will be graded. This question shall not be distributed to any person except by the instructors of CMPT 145. Solutions will be made available to students registered in CMPT 145 after the due date. There is no educational or pedagogical reason for tutors or experts outside the CMPT 145 instructional team to provide solutions to this question to a student registered in the course. Students who solicit such solutions are committing an act of Academic Misconduct, according to the University of Saskatchewan Policy on Academic Misconduct.

Task

This question assumes you have understood what's needed for A10Q2. You can get started on A10Q3 independently from A10Q1 and A10Q2.

You may be wondering if all this work is purely academic, or if there is some value to it. While our TBase objects store simple values, you could easily imagine using this class to store database records, or other application data. We won't build a real application, but we can imagine it.

The theoretical value of the TBase class is that it should be more efficient to use a binary search tree than using a Python list to keep track of values as they are added and removed from a collection. That's the theory. But what about in practice?

In Question 4, we will run an experiment to see if it really is more efficient practically. We'll want to compare how TBase performs to the performance on the same task using a Python list. In Question 4, we will build a single script that can easily be adapted, so that we can use TBase or a Python list. But before we get there, we need to make a Python list look like a TBase object. We called this an adaptor. Our code in Question 4 will require an object with the methods `member(self, value)`, `add(self, value)`, and `remove(self, value)`. We have one of those from Question 2.

In this question, we will define an alternative implementation of the TBase class, which uses Python lists instead of binary search trees. By building this adaptor, we can use the same experiment script (Question 4) for both implementations, A10Q2, and A10Q3.

On Moodle, you will find 2 files:

- `a10q3.py`
- `a10q3_test.py`

The first file is a partial implementation of the TBase ADT, very similar to the one from A10Q2. The second file is a test script to help you complete the implementation of these operations (also very similar to A10Q2). Currently, of 20 test cases, only 1 test case is successful.

You are allowed to use the course readings and the lecture notes to complete this question. However, you should not consult any other student, nor any other expert or tutor for assistance in this exercise.

Alternate TBase ADT Description

This file is almost identical to A10Q2. The methods have the same names, and the same interface (as described by the doc-strings). This time, you should implement the methods assuming that the TBase object stores everything in a Python list.

The TBase object stores the list of values (the stuff). The methods described below simply access this list to perform the needed behaviours. To help make this clear, the `__init__()` is given, and there is no need to add to it.

The TBase class in `a10q3.py` has five methods outlined with doc-strings, but the methods do not yet do what they should do. Here's what they should do:

- `is_empty(self)` Is the collection empty? Returns a boolean answer.
- `size(self)` How many unique values are stored? Returns an integer answer.
- `member(self, value)` Is the given value in the collection? Returns a boolean answer.
- `add(self, value)` Add the given value to the collection. Returns True if the value was added, and False only if the value is already there.
- `remove(self, value)` Remove the given value from the collection. Returns True if the value was removed, and False only if the value was not there to be removed.

You have some decisions to make. Which list methods should you use for these? Your decisions here will have some effect on the results from Question 4. Keep it simple, until you have Question 4 working. Then if you have time, revisit these methods.

What to Hand In

- A file named `a10q3.py` containing the corrected and completed implementation. **Note: If you do not name the script appropriately, you will get zero marks.**

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- Your solution to this question must be named `a10q3.py`, or you will receive a zero on this question.
- 10 marks: When our scoring script runs using your implementation of `a10q3.py`, no errors are reported. Partial credit will be allocated as follows:

Number of tests passed	Marks	Comment
0-1	0	the given a10q3 script already passes 1 test total
2-9	5	
10-18	8	
19-20	10	

  For example, if your implementation passes 17 tests, you'll get 8/10. Our test script is based on the test scripts distributed with the assignment, but may not be exactly the same.

Solution: The basic implementation is found in `a10q3.py`. IN this implementation, the Python list is kept unsorted.

- `is_empty(self)`

```
    def is_empty(self):
        """
        Purpose:
            Check if the TBase object is empty.
        Return
            :return: True if it's empty, False otherwise
        """
        return len(self.__stuff) == 0
```

- `size(self)`

```
    def size(self):
        """
        Purpose:
            Return the number of elements in the TBase object
        Return
            :return: an integer indicating the number of elements
        """

        return len(self.__stuff)
```

- `member(self, value)`. For this method, we use Python's `in` operator, which performs linear search on a list. The worst-case time complexity is $O(N)$, where N is the number of elements in the list.

```
    def member(self, value):
        """
        Purpose:
            Check if target is stored in the TBase object.
        Pre-Conditions:
            :param value: a value,
             Must be comparable to the other values in the TBase object
        Return
            :return: True if value is in the object
        """
        return value in self.__stuff
```

- `add(self, value)` For this method, we use Python's `List.append()` method, which adds a new value at the end of the list. Python's implementation is pretty clever. The worst-case time complexity is $O(1)$; that is, it's fast no matter how big the list is.

```
    def add(self, value):
        """
        Purpose:
            Store the given value in the TBase object.
        Pre-Conditions:
            :param value: a value
             Must be comparable to the other values in the TBase object
        Return
            :return: True if value was added
                     False if it was already there
        """
        if value in self.__stuff:
            return False
        else:
            self.__stuff.append(value)
            return True
```

- `remove(self, value)` For this method, we use Python's `List.remove()` method, which performs linear search on a list. The worst-case time complexity is $O(N)$, where N is the number of elements in the list.

```
    def remove(self, value):
        """
        Purpose:
            Remove the given value from the TBase object.
        Pre-Conditions:
            :param value: a value
             Must be comparable to the other values in the TBase object
        Return
            :return: True if value was removed
                     False if the value was not there at all
        """
        if value not in self.__stuff:
            return False
        else:
            self.__stuff.remove(value)
            return True
```

Alternate Solutions
This implementation is the simplest, and I guess most students will use that one. However, it is possible to implement a better version using Python lists that are sorted. This kind of implementation was mentioned in the BST lecture. With a sorted list, the `member()` method can do binary search, which is $O(\log N)$; this is a big improvement over the $O(N)$ of the unsorted list. To keep the list sorted, there are two techniques:

1. In `add()`, sort the list after every every `append()`. This is $O(N \log N)$. It's clearly worse than the unsorted list.
2. In `add()`, place the new value exactly where it should go, using the `List.insert()` method. It's possible to find the right place using a variation of binary search, one that returns the index of the position the new value should take. Binary search requires $O(\log N)$ steps. Here's where it gets dicey. To insert a value into the list, Python has to make room for it, by moving all the values in the list back one index. According to the Python Wiki, this takes $O(N)$ time.

Similarly, the `remove()` operation can be implemented by doing a binary search to find the value to be removed, but the `List.remove()` method requires $O(N)$ steps to shuffle data values.

Marking guidelines:

- As with Q2, it's certainly possible for students to have poor implementations of these methods. For example, a loop could be used for `member()`, or values could be added at the 0 index of the list.
  - There is no place here to deduct marks for that. A flaw in the grading scheme.
- To grade the implementations:
  1. Drop the files `test145.py`,`a10q3_test.py` into the student's folder, and run `python` as follows:

     ```
     UNIX$ python a10q3_test.py
     ----------------- Summary ------------------
     A10Q3: 10/10
     Tests passed: 20/20
     ```

     Transfer the score to Moodle.

Question 4 (5 points):

Purpose: To compare the two implementations from Q2 and Q3. To draw many parts of the course together in as single demonstration.

Degree of Difficulty: Easy. There is no coding here, but you may have to read through the code to understand what's going on.

Restrictions: This question is homework assigned to students and will be graded. This question shall not be distributed to any person except by the instructors of CMPT 145. Solutions will be made available to students registered in CMPT 145 after the due date. There is no educational or pedagogical reason for tutors or experts outside the CMPT 145 instructional team to provide solutions to this question to a student registered in the course. Students who solicit such solutions are committing an act of Academic Misconduct, according to the University of Saskatchewan Policy on Academic Misconduct.

Task

In Question 2 we implemented the TBase class using binary search trees. In Question 3 we implemented the TBase class using Python lists. They have the same methods, and the same purposes. But which one is more efficient? A formal algorithm analysis tells us that binary search trees should be very efficient for storing and retrieving and deletions by data values. But does the practice match the theory? This question will answer that.

Ideally, we would have a real application requiring the kinds of methods TBase defines, like a student database, or something like that. Instead, you will use a little simulator that only adds, searches, and removes data from the TBase, with no other useful functionality. The simulator will read a file containing a lot of simple commands, and see how long it takes to execute the whole file.

Because it is so close to the end of semester, I have provided such a simulation for you. You will use it, and your answers to A10Q2 and A10Q3, to explore the efficiency of these two implementation of the the same ADT.

Data

On Moodle, you will find a file `a10q4.zip` containing a bunch of example text files. Each file consists of a number of lines, with each line being a simple command:

```
a 2
a 7
r 2
m 2
m 7
```

Each line consists of 2 tokens, a character (`a`, `r`, or `m`), and an integer. The character indicates the command:

- `a` means to add the integer to the TBase object
- `r` means to remove the integer from the TBase object
- `m` means to check if the integer is in the TBase object

The example above says to add 2 and 7, then remove 2, and finally check if 2 and 7 are in the TBase object. The example files are very long, so that we can get a sense of the efficiency of our class's methods on modern computers.

Simulation

On Moodle you will find a Python script `a10q4.py` which opens the data files, runs the simulations, and then outputs some data to the console, and displays some plots to the screen.

This experiment script should use the TBase object. Once you have the script working, you are to compare the practical efficiency of the two implementations (A10Q2 and A10Q3) using the same script. Because the two implementations have the same methods, they can be exchanged simply by changing the import line.

Reflection Questions

- Provide a table of results, showing the run-time for both methods on all example files. You can copy-/paste from the console, into a document.
- Explain the results of the experiment. If one implementation is faster, why? If they are the same, explain how that could happen. Use what you know about the time complexity of Binary Search Tree operations in your explanation.

Note: You answers could change depending on how you implemented A10Q3!

Here's a template for the table you want to produce. The column for N gives the length of the scripts, in terms of the number of commands. The two empty columns should be filled with execution times.

N	A10Q2	A10Q3
1000		
2000		
3000		
4000		
5000		
6000		
7000		
8000		
10000		
12000		
14000		

What to Hand In

- A file named `a10q4_reflections.TXT` (but DOCX, PDF, and RTF files are all acceptable, especially if you want to get fancy with plots) containing your table of results, the answers to the reflection questions.

Be sure to include your name, NSID, student number, and course number at the top of all documents.

Evaluation

- 7 marks: Your results and explanations.
  - Your Table of results is neat and clear.
  - Your explanation draws together concepts of algorithm analysis, and the study of binary search trees.
  - You made some plausible assumptions about Python list methods' efficiency, or you have used external sources (cite them).

Solution:

Reflection Questions

- I ran the A10Q4 script using my iMac. I collected the time taken to execute the script for the various N. The table is given below.

N	A10Q2	A10Q3	Sorted
1000	0.0026	0.0033	0.0027
2000	0.0063	0.0125	0.0060
3000	0.0103	0.0287	0.0097
4000	0.0139	0.0491	0.0131
5000	0.0193	0.0769	0.0173
6000	0.0217	0.1218	0.0210
7000	0.0281	0.1475	0.0262
8000	0.0293	0.1943	0.0296
10000	0.0400	0.3066	0.0370
12000	0.0484	0.4269	0.0468
14000	0.0592	0.5857	0.0570

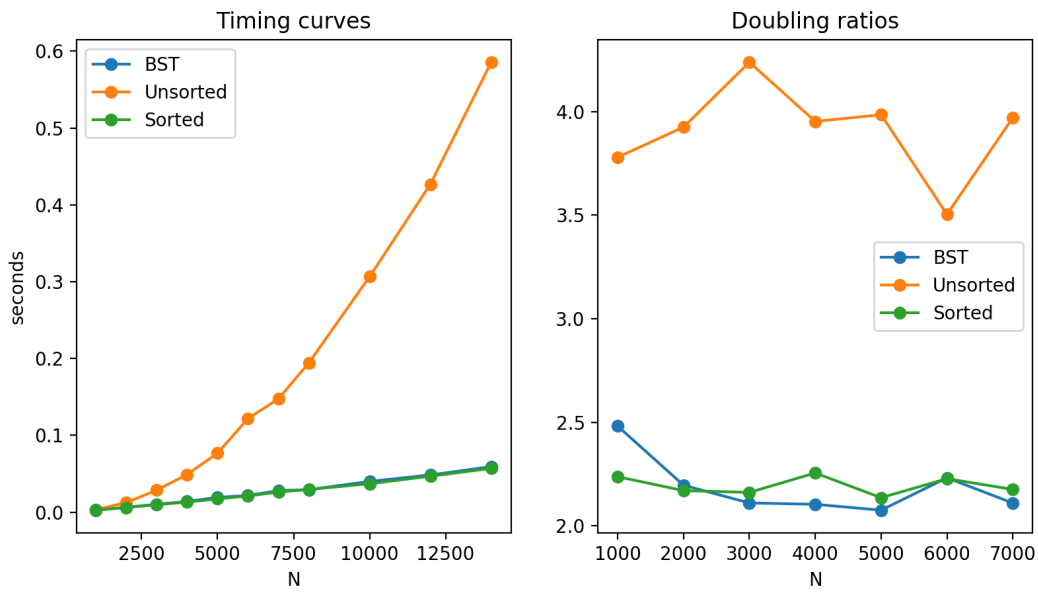
At $N = 1000$, i.e., the smallest example, the two implementations are roughly equally fast. But the times for A10Q3 increase a lot faster than for A10Q2. There is a third column here, which is an alternate implementation that uses a sorted list and binary search. The times for this implementation are virtually identical to A10Q2.

I also calculated the doubling ratios for these implementations. I repeated each execution 10 times, and took the minimum value (instead of an average). The data is given below.

N	A10Q2	A10Q3	Sorted
2000	2.48	3.78	2.24
4000	2.20	3.93	2.17
6000	2.11	4.24	2.16
8000	2.10	3.95	2.25
10000	2.07	3.99	2.13
12000	2.23	3.50	2.23
14000	2.11	3.97	2.18

The doubling ratio is basically the ratio $T(2N)/T(N)$, the ratio of the time taken by a problem of twice the size. In the above table, I list the value of $2N$ in the first column.

Because tables of numbers are not as helpful as a graphic, here's a plot of the data. The green data is A10Q2, the orange data is A10Q3, and the blue data refers to the implementation that uses the Sorted list.



- Explanation.

  Theoretically, all the BST operations are worst-case $O(\log N)$, where N is the number of elements in the tree, assuming non-degenerate trees. For us, N is not really known, but because we start with roughly $N//3$ add commands, we can conclude that the number of elements is more or less proportional to the size of the script. If each operation is $O(\log N)$, and there are N operations, then the theoretical time complexity is $O(N \log N)$. The doubling ratio is slightly above 2, but well below 4, so that's consistent.

  For A10Q3, the times increase a lot faster. In my implementation of A10Q3, I used linear search for member and remove. These are both $O(N)$. The add operation was implemented using append, which might be $O(1)$. So if 2/3 of N operations are $O(N)$, then the entire execution will be $O(N^2)$. The doubling ratio, for A10Q2 is around 2.2, and for A10Q3, it's around 3.9.

  We can understand these results as follows:

  - For A10Q2, all operations are $O(\log n)$ where n is the number of values in the tree. This is based on the assumption of non-degenerate tree shapes, of course.
  - For A10Q3, all operations are $O(n)$, where n is the number of values in the list.
  - Each of the experiment test files have N commands for adding, removing or checking values.
  - At the beginning of each script, there are about $N/3$ `add` commands. The BST (Q2) or list (Q3) grows initially from being pretty small, to having about $N/3$ values. Because 3 is constant, we can say that the number of values is $O(N)$, even though it's not exactly N.
  - After the initial sequence of `add` commands, there are roughly equal numbers of `add`, `remove`, and `member`. The total number of these is about $2N/3$, or $O(N)$.
  - The time complexity for processing the N commands in a single command file is $O(N)$ times the complexity of each operation individually.
  - We will over-estimate the worst case if we use $n = N$. In other words, if we pretend that there are N values in the tree (or list) even during the first portion of the command file, then we are over-estimating the true cost.

  - For A10Q2, we perform $O(N)$ operations each having worst case time complexity of $O(\log N)$ where N is the length of the command file. Therefore, the cost to process the whole file is $O(N \log N)$.
  - For A10Q3, we perform $O(N)$ operations each having worst case time complexity of $O(N)$ where N is the length of the command file. Therefore, the cost to process the whole file is $O(N^2)$.
  - The doubling ratio for $O(N^2)$ is theoretically equal to 4. The doubling ratios determined from our experiment for A10Q3 are close to this value.
  - The doubling ratio for $O(N \log N)$ is theoretically just a bit larger than 2. The doubling ratios determined from our experiment for A10Q2 are in the right range. This suggests that our trees are not degenerate. Anecdotally, I checked the height of the trees after processing all the commands, and the height of the BST was not too far from the height of a perfectly complete tree with N values. Not right on, but not anywhere close to N, and so we're not working with degenerate trees.

- Hooray for practice matching the theory. Using a BST is significantly faster than an unsorted list.

It's interesting to consider the third implementation, the one labelled *Sorted*. The data show that it's virtually identical to the implementation using BSTs in A10Q2. In the Timing Curve plot, above, the green data sits right on top of the blue data; it's barely visible. Does that mean Python lists are as fast as binary search trees, even for insert and remove operations? The answer is complicated.

- Every source I have seen indicates that inserting and removing values from a Python list is $O(n)$. This would imply that we should see the green line should have the same general shape as the orange line, not the blue line.

- The data shows that something weird is going on. It could be a hardware optimization, exploited by Python's list methods. Or it could be a clever Python implementation. It's not clear what the true explanation is, but it is clear that this trick only works for lists whose size is below a critical threshold. The data files provided in this assignment resulted in lists whose sizes are below this threshold. I did run the experiment with very large values of N, and when $N > 2^{18}$ or so (that is about $20\times$ bigger than the data files provided in this assignment), the timing curve for Sorted started increasing more like $O(N^2)$ than $O(N \log N)$. It took close to an hour of computer time to collect that data, so most students probably will not have noticed that.

  I am not sure that the same results for Sorted can be obtained on Linux or Windows. I presume so, but I did not check.

  Practically speaking, these kinds of tricks are good news. Many applications will use lists small enough to benefit from dramatically increased speed. On the other hand, it does cloud the issue for novice computer scientists. How can you take algorithm analysis seriously if mysterious hardware optimizations or clever implementations make the analysis seem pointless?

Marking guidelines:
Grade the reflection responses using the following criteria, expressed in bold. Each bold phrase is a rubric category. If more than one applies, use the most severe (min point value).

- 5 marks: Results and explanations.
  - Your Table of results is neat and clear.
    - The table has to be legible. If there are entries that are unreadable, it **could have been better**.
    - If the table is not complete, it **could have been better**.
    - Students will probably not limit precision, and we have to accept that. It's terrible, but acceptable as long as the numbers don't overlap, or aren't obscured in some way.
    - Students are only required to compare 2 implementations.
    - Doubling ratios are nice, but not required. The doubling ratios should be close to the values discussed above, but some variance is expected.
    - There was no requirement to use plots, or MatPlotLib. Excel, while not preferred, has to be allowed because it was not ruled out in the assignment.
  - Your explanation draws together concepts of algorithm analysis, and the study of binary search trees.
    - If A10Q2 does not show $O(N \log N)$ behaviour, then it's **weak understanding of the concepts**. A bug in A10Q2 or A10Q1 could pass all the tests, but still produce degenerate BSTs. Here is where marks are lost for that.
    - If the explanation is based on a single operation (e.g., $O(\log N)$ instead of $O(N \log N)$), then it's **weak understanding of the concepts**.
    - Students are not expected to perform the analysis of the BST operations. They were given in lecture.
    - If the explanation concludes that A10Q2 is "X times faster" than A10Q3, then it's **weak understanding of the concept**.
    - If the explanations are wrong, vague, or uncertain, then it's **weak understanding of the concept**.
    - If the explanations are more or less correct, but expressed using poor language skills, or perhaps poor presentation skills, then **it could have been better**.
  - You made some plausible assumptions about Python list methods' efficiency, or you have used external sources (cite them).
    - A quick Google search pulls up lots of hits for the complexity of the list methods. We discussed `in` in class, but not `List.insert` or `List.remove`. We should presume that students looked it up, and that's fine. But they may not have cited their source, or the citation might be like what I did ("Python Wiki"). Informal citation is fine. We don't need to be strict.
    - If the complexity of List methods is not discussed at all it's **weak understanding of the concept**.