

HW 5 BST, RB Tree, and Dynamic Programming (LCS)

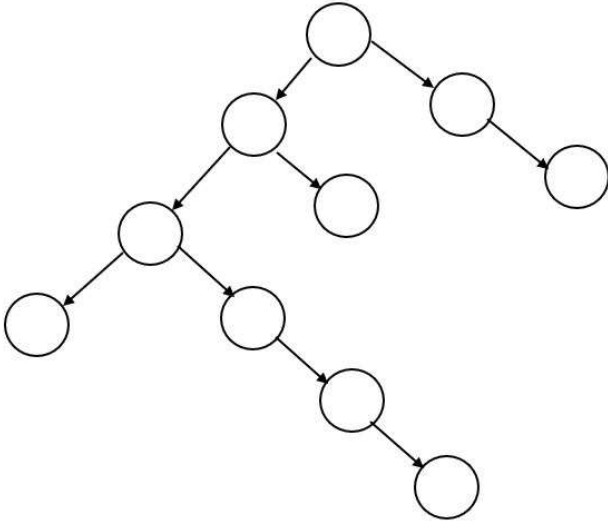
Start Assignment

Q1) (10 points) Answer the following questions in a separate document.

- a). (2 points) Assign the keys, **59, 140, 30, 41, 68, 115, 70, 155, 99, 92** to the nodes of the **binary search tree** shown below. Feel free to do trials and errors to fill the tree with the given list of keys.
- b). (2 points) There are a list of random keys and a binary search tree. The tree shape is fixed like the example below and you can't change it. The number of keys in the list is the same as the number of available nodes of the tree. We want to **systematically** fill in all the nodes of the tree with the randomly ordered list of the keys. Describe an algorithm to fill the tree completely.
- c). (2 points) **Starting from the root node**, we want to populate a BST with the keys from a given list without altering its order. Using the keys and the tree in question a) as an example, show the ordered list of the keys which can be sequentially inserted into the tree. Also, explain and demonstrate how you would get such a list **systematically** using the tree from question b). Please limit your description to a sentence or two.
- d). (2 points) Describe why this specific instance of binary search tree below can't be colored to form a valid red-black tree even without populating keys.
- e). (2 points) Now let's try to do two rotations to make it an RB tree. Describe how rotations are done and show the final picture of an RB tree showing each node with a key and its color.

Q1) Deliverable:

Upload a PDF document with the answers to 5 questions with a picture of an RB tree from the last question (e). Feel free to print or draw the picture below and fill the nodes in the tree with pens.



Q2 (10 points) Write a Python code implementing naïve LCS recursive, LCS DP, and LCS bottom-up versions. Then write a driver program that

1. Reads all the numbers in the "rand1000000.txt" file **as strings**. You may use "rand1000.txt" instead for your testing purpose.
2. For each 6-digit integer (100,000 - 999,999) string in the file, run an LCS function against "0123456789" string
3. Using your **timeEfficiency()** function from HW2, compare the time efficiency of three versions (recursive vs. DP vs. bottom-up) for the entire 1M integer strings

My test result was around 30 minutes (recursive) vs 40 seconds (DP).

Q2 Deliverable:

Both Python codes of your main driver program and the output of your timeEfficiency() calls with your comments

Q3) (25 points) In this question, you will also develop your own simple hash function and a hash table of 6-digit integer strings from the 1M data file.

1. Modify the LCS recursive version to count the number of recursive calls for any 6-digit integer string (100,000 - 999,999) against "0123456789" string. The function returns a tuple of two elements (the returned LCS number, the number of recursive calls to find the LCS number)
2. We will use this recursive LCS function as a very slow **hash function of integer strings**. This recursive LCS function will map the integer strings into the indexes of the hash table. **You are not allowed to use Python dict data type for this problem.** You must develop your own hash table (**table size: 10,000**). The number of recursive calls from the LCS function is used as the index of the hash table.
3. Now, to evaluate how well-spread this hash function would be when mapping the integer string keys to the indexes, we want to count the number of collisions for each index of the hash table built from the 1 M data file. It would be ideal if the average collision number is close to 100 for 10,000 slots out of 1M numbers. We can get an idea of its uniformity by displaying the number of collisions, but in this question,

you are asked to visualize the distribution of the collisions across all the indexes using a Python plotting library.

4. Feel free to choose any plotting library available (an example: <https://plot.ly/python/> (<https://plot.ly/python/>)) to visualize the distribution of the collision of the LCS (hash) function:

```
import matplotlib.pyplot as plt
from plotly.graph_objs import Bar, Layout
from plotly import offline
```

or any other library of your choice such as panda, if you prefer

- Draw an **offline bar graph** to show the relationship between hashed indexes (the number of recursive calls to compute the LCS against "0123456789") and the number of collisions for each index to the hash table. You may refer to the chart that I came up with using Python dict{} class (I could have used collections or **Counter** class for that matter) but again, for this homework exercise, you should use your own hash table which maps each key to the corresponding list of collisions. In any case, the final chart should look very similar, if not identical, to << [lcsUniformity.html](https://ucdenver.instructure.com/courses/471001/files/14040026/download?download_frd=1) ↓
(https://ucdenver.instructure.com/courses/471001/files/14040026/download?download_frd=1) >>

NOTE: my sample chart is based on numbers ranging 0 - 999,999

5. From my personal experience, I believe that no meaningful data analysis can be completed without interjecting human intuition in the data interpretation process. Let's exercise our own. Look at my chart mentioned above. Although it appears fairly uniformly distributed across the indexes (LCS numbers), it's not an ideal distribution. Moreover, if you look at it closely, after passing a certain threshold, the collision rate tapering off rather rapidly, which means that the bigger the number of recursions is, the smaller the collisions (frequencies of hash keys generated in the range) get and they are thinly spread out. Also, note that these numbers would take a much longer time to return hash results because they make more recursive calls. So I would like to limit the number of recursive calls in a function when it passes a certain threshold.

With that in mind,

- From the intuition of your own bar graph, pick up a threshold number and give your reasons explaining why you picked up the number. No right or wrong answer here.
- To handle those numbers that go over the threshold, you may develop a simple secondary hash function of your own choice to collapse those outliers to a shorter range so that the upper bound of the entire hash table is 10,000. Those remapped outliers should result in more uniformly distributed (packed) index values.
- Then run your new hashing algorithm to redraw the distribution chart ranging from 1 - 10,000.

Since running LCS for the million numbers in rand1000000.txt file takes quite a long time, you may first test it with rand1000.txt file (less than 10 seconds)

Q3 Deliverable :

- Python codes of modified LCS recursive and the driver program
- Python codes of your own dictionary library

- The offline HTML files of the hash table collision charts for both before and after the secondary hash function.

Extra credit) (15 - 20 points to your midterm or final exam score to maximize your grade)

This is for the BST and RB Balanced Tree implementation

Read the following postings:

1. <https://stackoverflow.com/questions/2298165/pythons-standard-library-is-there-a-module-for-balanced-binary-tree> (<https://stackoverflow.com/questions/2298165/pythons-standard-library-is-there-a-module-for-balanced-binary-tree>)
2. https://www.reddit.com/r/Python/comments/9allgh/wheres_pythons_damn_binary_search_tree/ (https://www.reddit.com/r/Python/comments/9allgh/wheres_pythons_damn_binary_search_tree/)
3. <https://medium.com/@stephenagrice/how-to-implement-a-binary-search-tree-in-python-e1cdba29c533> (<https://medium.com/@stephenagrice/how-to-implement-a-binary-search-tree-in-python-e1cdba29c533>)
4. <https://github.com/OmkarPathak/Data-Structures-using-Python/blob/master/Trees/BinarySearchTree.py> (<https://github.com/OmkarPathak/Data-Structures-using-Python/blob/master/Trees/BinarySearchTree.py>)

Also, visit:

- <https://pypi.org/project/binarytree/> (<https://pypi.org/project/binarytree/>)

I found the third posting above is useful for the beginners of a BST implementation. Please read the blog in its entirety.

1. Download or copy BST implementation from the third or fourth link or any other source you like.
2. Add the following methods in the downloaded code:
 - **getHeight ()** - returns the height of a given tree
 - **countNodes()** - returns the number of total nodes in a given tree
3. Create and then populate a BST tree with the integer data from 'rand1000000.txt' file and get the height and number of nodes of a given tree using your methods specified above.
 - Make sure add() and delete() work
4. Build a BST from "rand1000.txt" file (or 100 nodes if you prefer but you may need to make your own data file of 100 nodes)
 - Traverse the tree in **descending** order of the keys and print the keys with a ranking number. For example, if a key, 418621, is the 534th largest in the tree, display "[534] 418621"
 - **You must use Python "Generator" pattern using 'yield' to display the number in the specified format.**
 - Feel free to use any Python standard data types available to map a number to English format
5. Now, do the same with an RB BST implementation. You may use the 'pypi' binary tree package (see the link below) or any other RB tree library you can find from the Internet. Then do steps 2 - 3 (don't need step 4) with a Red-Black BST tree.
 - <https://pypi.org/project/bintrees/> (**a stopped project**) (<https://pypi.org/project/bintrees/>)

6. Compare and analyze the results from both trees

Extra credit deliverable:

1. (12 points, or 7 points without RB BST) The analysis of the heights of both trees (BST and RB BST) with 1 Million nodes
2. (8 points) The output of a BST tree traversal displaying each key in descending order with a ranking number.

Some Rubric (5)			
Criteria	Ratings		Pts
Question 1: Correctness of answers (a - e) 2 points for each subquestion	10 pts Full Marks	0 pts No Marks	10 pts
Question 2: naïve LCS recursive, LCS DP, and LS bottom-up versions	10 pts Full Marks	0 pts No Marks	10 pts
Question 3: LCS recursive and Hash table - (5 points) Python codes of modified LCS recursive and the driver program - (10 points) Python codes of your own dictionary implementation - (10 points) The offline HTML files of the hash table collision charts for both before and after modification	25 pts Full Marks	0 pts No Marks	25 pts
Extra Credit to the midterm or final exam - (12 points) for both BST and RB BST, or (7 points) for BST only Implementation of the heights of BST and RB BST with 1 Million nodes - (8 points) Displaying the output of a tree traversal displaying the numbers in descending order with a ranking number.	0 pts Full Marks	0 pts No Marks	0 pts
Total Points: 45			