CS2321 Lab 12

Lab Instructions:
Save the code you write for each exercise in this lab as a *library* -- that is, a textfile with a .py extension containing only executable python code (i.e. no angle-bracket prompts, etc). Name each file according to the exercise number (e.g. ex1.py, ex2.py, etc.) and save them to a directory containing the report file (in PDF), when completed, compress them together in a single zip file to be submitted on D2L. Title and label axes of all graphs, if graphs are required.

Each function should have a docstring explaining what the function does.
Any Follow-up Questions and their Answers should be included in a **docstring** following the `main()` function.

e.g. the structure for a Python module should look like:

```
'''
   modulename.py
   Doc-string explaining what this module does
'''
# imports, such as math, random, etc., as needed

# Your code, includes definitions of classes, functions, etc.
def ...
def ...
.
.
.
def main():
   # Do what is needed.


if __name__ == "__main__":
      main()

'''
   Doc-string answering follow up questions
'''
```

Lab Deliverable: Once all your programs run correctly, collect their code and the results of their test-cases in a nicely-formatted **PDF** file exported from Word Processing document (e.g. MS Word or LibreOffice) to be included in the submission on D2L.

This **report** should consist of each lab exercise, clearly **labeled** in order, consisting of code, then copy/pasted text output, and figure of x-y plotted graphs properly titled and labelled.

Search, and Hashing

Goals for this lab: to explore the Hashing search algorithm, and compare it to SequentialSearch and BinarySearch. These techniques are O(1), O(n) and O(lg n) respectively. So in completing this lab, I would expect you to generate graphs that look like constant, linear and logarithmic respectively.

Lab Instructions:

1. Implement Hashing storage, and retrieval.
   - write a q*uickLoad* method for your HashTable class
     - interface: {ht}.quickLoad(*listOfValues*, *loadFactor*)
     - quickLoad method: resets *self.slots* to an appropriately-sized list (depending on size of *listOfValues* and *loadFactor*). Then fills it with elements from *listOfValues*.

   - choose your favorite collision-resolution method, except: do not use the *list-of-lists* technique shown in class (unless it is *an extra trial in addition* to a more traditional technique).
     - show/explain the code for your rehash method in your lab report

2. Compare average search-time of a given item using Hashing (with two load-levels: 50% and 95%)
   a. Generate search-timing series for both *present items*, and *non-present items*
   b. This can probably be done in two graphs: one for present, another for non-present items. Each graph contains two series: 50% and 95% timings.

3. Graphically compare HT results to (Unsorted) Sequential Search, and Binary Search (again, using ***average per-item timing*** for both *present* and *nonpresent* items).

4. Come up with two relevant, testable questions (i.e. hypotheses) and test-procedures; write your hypothesis; then run your tests, determine your average results, graph them, revisit your hypothesis and ***explain your results***.

Example question 1: Does search using Hashing, a presumably O(1) (constant-time) algorithm, really execute faster than Binary Search, a O(lg n) technique?

- Your Hypothesis: Don't be ridiculous! How could it possibly?!?

Example question 2: Does search using Hashing, a presumably O(1) (constant-time) algorithm, really execute faster than Sequential Search, a O(n) technique?

- The Everybody Hypothesis: Sounds legit.

Suggested parameters to get you started:

a) Test with 10 list-sizes of size 100,000 - 1,000,000.
b) To ensure all-unique numbers, generate a list of random numbers using: `random.sample()`
c) To properly compare apples to apples, search for the same two lists of numbers (present, non-present) in each case.

Suggested Plan of Attack:

A. Write a HashTable class: The HashTable class is given as a starting point.

- use a simple Python-List implementation, not a list-of-lists.
    - default list-size: 100 name: *self.slots*
- quickLoad method: resets *self.slots* to an appropriately-sized list (depending on size of *listvals* and *loadFactor*). Then fills it with elements from *listVals*.

B. Create a random list of 200,000 integers using random.***sample()*** name: TOTLIST

- divide into two master-lists of size 100,000: INS and NOTINS

For example, to create a list of 20 items with all unique numbers between 0 and 100:

```
lyst = random.sample(range(100), 20)
```

Make sure that the # of unique values is more than 20, and in this example, there are 100 unique numbers to choose from.

```
      For LEN in range(10000, 100001, 10000):
        Create 2 HTs with first LEN elements of INS -- 0.50 and 0.95
loadFactor respectively  (e.g.   ht.QuickLoad(INS(:LEN), 0.50)
                                 myINS = INS(:LEN)
                                 myNotINS = NOTINS(:LEN) )
        Find average search-time of LEN NOTINS items vs. INS items

        Compare to search-times of these sizes of lists using:
            - sequential search
            - binary search
            - hash table search with its get method
```

Hints:

- Never overwrite INS, NOTINS after creation, but you may choose to create new (derived) lists as needed, using the list slicing operator: lyst = INS[:]
- Be sure to explain the necessary preprocessing steps for the sorted search-types in your report
- Beware of *aliasing* when manipulating and/or sorting your lists. If aliasing occurs, explain how it happened, how you discovered it, and how you fixed it.
- To search the INS, randomly pick an item as search target each time it is tested.
- To search the NOTINS, simple search for a non-existent number such as a negative number, -1, assuming all the numbers in NOTINS are non-negative integers.
- Create three functions for the three searches and time each search twice with present/non-present targets.
- Use timeit to measure the timing of the functions.