Homework Assignment 4 Searching and Hashing

Due May 14 11:59 pm

Homework template and test script can be downloaded from Canvas. Be sure to test with *hw4_test.py*, and submit both *hw4.py* and *HashTable.py* to Gradescope. Problems 1 and 2 are in *hw4.py* and both parts of Problem 3 are in *HashTable.py*.

Problem 1 Sequential Search (20 points)

We presented a recursive sequential search function in lecture 14 slide 43 that used slicing to pass a smaller list to the recursive function call. We showed in PythonTutor that Python creates a new list for every function call, resulting in unnecessary overhead. For this problem, you will rewrite the recursive procedure so that it doesn't use slicing, but instead passes an index value for the beginning of the list. You should use the definition line below which we provided for this new function in hw4.py.

```
def sequentialSearchRec(alist, item, index=0):
```

The first call to binarySearchRec will not include the index parameter, and Python will set it to 0 by default. Index is the position of the first element in the (remaining) list being searched.

Problem 2 Binary Search (30 points)

We presented a recursive binary search function in lecture 17 slide 38 that used slicing to pass a smaller list to the recursive function call. We also showed in PythonTutor slicing resulted in a new list for every function call. Your goal is to eliminate that overhead by re-writing the function so that it doesn't use slicing. You will do this by passing the first and last indices along with the unmodified list to recursive calls. This new version of recursive binary search will therefore use the midpoint calculation used in iterative binary search. You should use the definition line below which we provided for this new function in hw4.py.

```
def binarySearchRec(alist, item, first=None, last=None):
```

The first call to binarySearchRec will not include the first and last parameters, and Python will set those to None by default. You should test for this case and set them to the appropriate initial values for the first and last index of the list.

Problem 3 Hashing

The HashTable class implemented in chapter 5 of your online textbook exhibits undesirable behavior if you use put(key,val) to add a new key-value pair when the table is full. Re-implement

the put method so that the table will automatically increase in size when the method detects that the table is full.

The new size should be a prime number that approximately twice the current size of the table. For example, if the original size is 11, the new size would be 23. We are providing you with a good sequence of hash table sizes up to a maximum size of 1543. You may assume that maximum size you will need to implement for the table is 1543.

good_sizes = [11, 23, 53, 97, 193, 769, 1543]

We will divide this task into two parts.

Problem 3 Part 1 Counting empty slots (20 points)

In the first part implement a method called empty_slots() that returns the number of empty slots in the hash table. A slot is empty if it contains None. The number of empty slots decreases by one when a key is inserted into a slot containing ‘None’

To keep the O(1) complexity of insertion into the hash table, we need empty_slots() method to be O(1). To achieve this, you will need to implement empty_slots() using a variable to count the number of empty slots, instead of scanning the entire table each time empty_slots() is called.

Problem 3 Part 2 Increasing the table size (30 points)

After empty_slots() is implemented, you will use it to test if the table is full before a value is inserted into the table. If the table is full you will create a new table using the next value in the good_sizes list the and re-insert each key from the previous table into the new table. You may want to create a helper method for this, but it’s not necessary.