CSC263 – Problem Set 1

Remember to write your **full name** and **student number** prominently on your submission. To avoid suspicions of plagiarism: at the beginning of your submission, **clearly state any resources (people, print, electronic) outside of your group, the course notes, and the course staff, that you consulted**.

Remember that you are required to submit your problem sets as both LaTeX `.tex` source files and `.pdf` files. There is a 10% penalty on the assignment for failing to submit both the `.tex` and `.pdf` files, or submitting inconsistent versions of these files.

Due Feb 9, 2022, 22:00; required files: ps1soln.pdf, ps1soln.tex, csc263_ps1.py

Answer each question completely, always justifying your claims and reasoning. Your solution will be graded not only on correctness, but also on clarity. Answers that are technically correct but are hard to understand will not receive full marks. Mark values for each question are contained in the [square brackets].

Please see the course syllabus for the late submission policy.

1. [**10 points**] Consider the following algorithm, where L is a linked list.

```
1 def search(L):
2   n = L.size # the number of elements in "L", excluding the "None" element
3   z = L.head
4   print("Starting search.")
5   while z != None and z.key != 10:
6     print("Keep searching.")
7     z = z.next
8   return z
```

Suppose that the input list L has n (≥ 20) elements (excluding the `None` element). The list is constructed randomly as follows, with all choices are made independently of each other.

- The first number in the list is the number 1
- The second number in the list is picked uniformly randomly from the set $\{1, 2\}$
- The $k-th$ number in the list is picked uniformly randomly from the set $\{1, 2, \ldots, k\}$

Let us analyze the number of times a "print" statement is executed. Answer the following questions in **exact form** and **not** in asymptotic notation. Show your work!

(a) Since the while loop stop at Z.key equals to 10 or running the whole string, and the length of the string is larger than 20. So, the smallest number of times that Print statements are executed should be 10, at this case, the Z.key = 10. So the smallest number of times that print statements are executed is 10 times. And The $k-th$ number in the list is picked uniformly randomly from the set $\{1, 2, \ldots, k\}$. So when k.KEY = 10, $k \geq 10$, and min(k) = 10. The probability for achieving this situation is $\frac{1}{10}$.

(b) The largest number of times that print statements happens when the while loop do not stop until the whole list is run, which is z = None. So the largest number of times that print statements are executed is n times. The probability for achieving this is (1 - $\frac{1}{10}$) × (1 - $\frac{1}{11}$) × (1 - $\frac{1}{12}$) × (1 - $\frac{1}{13}$)×...× (1 - $\frac{1}{n-1}$) × (1 - $\frac{1}{n}$) = $\prod_{i=10}^{n} \frac{i-1}{i} = \frac{9}{n}$

(c) The probability with 10 times = $\frac{1}{10}$
The probability with 11 times = $\frac{9}{10} \times \frac{1}{11} = \frac{9}{10\times11}$
The probability with 12 times = $\frac{9}{10} \times \frac{10}{11} \times \frac{1}{12} = \frac{9}{11\times12}$
and so on...
The probability with n - 1 time = $\frac{9}{(n-2)\times(n-1)}$
Thus, the expected number of times = $10 \times \frac{1}{10} + (\sum_{i=11}^{n-1} i \times \frac{9}{(i-1)\times i}) + \text{n} \times \frac{9}{n} = 1 + (\sum_{i=11}^{n-1} \frac{9}{i-1}) + 9 = 10 + \sum_{i=10}^{n-2} \frac{9}{i}$

(d) Since the expected number of times that print statements are executed is $10 + \sum_{i=10}^{n-2} \frac{9}{i}$,when n approaching to infinity, $\lim_{n\to\infty} 10 + \sum_{i=10}^{n-2} \frac{9}{i} = \lim_{n\to\infty} \sum_{i=10}^{n-2} \frac{9}{i} = \lim_{n\to\infty} \sum_{i=10}^{n} \frac{9}{i} = 9 \lim_{n\to\infty} \sum_{i=10}^{n} \frac{1}{i} = 9 \lim_{n\to\infty} \sum_{i=1}^{n} \frac{1}{i}$
$= 9 \lim_{n\to\infty} \int_{n}^{i} \frac{1}{i}\, di = 9 \ln(n)$
thus, we can conclude that the the expected number of times that print statements are executed is $O(\log(n))$.
Thus $\Theta(\log n)$

2. **[10 points]** Suppose the keys 1, 2, 3, 4, 5, 6, 7 are inserted in random order into a binary search tree, where each order of insertion is equally likely to occur. We will be computing the average height of the resultant tree.

   To that end, let $H(n)$ be the average height of a BST with keys from $\{1, \ldots, n\}$ inserted in uniformly random order. So (trivially) $H(1) = 0$, $H(2) = 1$.

   (a) Show that $H(3) = \frac{5}{3}$.
   The average height of 1 being root is 2
   The average height of 2 being root is 1
   The average height of 3 being root is 2
   Thus, $H(3) = \frac{1}{3} \times 2 + \frac{1}{3} \times 1 + \frac{1}{3} \times 2 = \frac{2}{3} + \frac{1}{3} + \frac{2}{3} = \frac{5}{3}$

   (b) Show that $H(4) = \frac{7}{3}$.
   When 1 being its root, all other integers is more than 1, so its right side is a BST with three elements; thus, The average height of 1 being root is $1 + \mathrm{H}(3) = 1 + \frac{5}{3} = \frac{8}{3}$
   When 2 being its root, 1 is less than 2, and 3, 4 is larger than 2, so it left side is a single BST, and the right side is a BST with 2 integers; thus, The average height of 2 being root is $1 + \max(H(2), H(1)) = 1 + H(2) = 1 + 1 = 2$
   When 3 being its root, 1, 3 is less than 3, and 4 is larger than 3, so it left side is a a BST with 2 integers, and its right side is a single BST; thus, The average height of 3 being root is $1 + \max(H(2), H(1)) = 1 + H(2) = 1 + 1 = 2$
   When 4 being its root, all integers are less than 4, so its left side is a BST with three elements; thus The average height of 4 being root is $1 + \mathrm{H}(3) = 1 + \frac{5}{3} = \frac{8}{3}$
   In conclusion, $H(4) = \frac{1}{4} \times \frac{8}{3} + \frac{1}{4} \times 2 + \frac{1}{4} \times 2 + \frac{1}{4} \times \frac{8}{3} = \frac{7}{3}$

   (c) $H(5) = \frac{14}{5}$.
   The average height of 1 being root is $1 + \mathrm{H}(4) = 1 + \frac{7}{3} = \frac{10}{3}$
   The average height of 2 being root is $1 + \max(H(3), H(1)) = 1 + H(3) = 1 + \frac{5}{3} = \frac{8}{3}$
   The average height of 3 being root is $1 + \max(H(2), H(2)) = 1 + H(2) = 1 + 1 = 2$
   The average height of 4 being root is $1 + \max(H(3), H(1)) = 1 + H(3) = 1 + \frac{5}{3} = \frac{8}{3}$
   The average height of 5 being root is $1 + \mathrm{H}(4) = 1 + \frac{7}{3} = \frac{10}{3}$
   Thus, $H(5) = \frac{1}{5} \times \frac{10}{3} + \frac{1}{5} \times \frac{8}{3} + \frac{1}{5} \times 2 + \frac{1}{5} \times \frac{8}{3} + \frac{1}{5} \times \frac{10}{3} = \frac{14}{5}$

   (d) Show that $H(6) = \frac{49}{15}$.
   The average height of 1 being root is $1 + \mathrm{H}(5) = 1 + \frac{14}{5} = \frac{19}{5}$
   The average height of 2 being root is $1 + \max(H(1), H(4)) = 1 + \mathrm{H}(4) = 1 + \frac{7}{3} = \frac{10}{3}$
   The average height of 3 being root is $1 + \max(H(3), H(2)) = 1 + H(3) = 1 + \frac{5}{3} = \frac{8}{3}$
   The average height of 4 being root is $1 + \max(H(3), H(2)) = 1 + H(3) = 1 + \frac{5}{3} = \frac{8}{3}$
   The average height of 5 being root is $1 + \max(H(1), H(4)) = 1 + \mathrm{H}(4) = 1 + \frac{7}{3} = \frac{10}{3}$
   The average height of 6 being root is $1 + \mathrm{H}(5) = 1 + \frac{14}{5} = \frac{19}{5}$
   Thus, $H(6) = \frac{1}{6} \times \frac{19}{5} + \frac{1}{6} \times \frac{10}{3} + \frac{1}{6} \times \frac{8}{3} + \frac{1}{6} \times \frac{8}{3} + \frac{1}{6} \times \frac{10}{3} + \frac{1}{6} \times \frac{19}{5} = \frac{49}{15}$

   (e) Compute $H(7)$.
   The average height of 1 being root is $1 + \mathrm{H}(6) = 1 + \frac{49}{15} = \frac{64}{15}$
   The average height of 2 being root is $1 + \max(H(1), H(5)) = 1 + \mathrm{H}(5) = = 1 + \frac{14}{5} = \frac{19}{5}$
   The average height of 3 being root is $1 + \max(H(2), H(4)) = 1 + \mathrm{H}(4) = 1 + \frac{7}{3} = \frac{10}{3}$
   The average height of 4 being root is $1 + \max(H(3), H(3)) = 1 + \mathrm{H}(3) = 1 + \frac{5}{3} = \frac{8}{3}$
   The average height of 5 being root is $1 + \max(H(2), H(4)) = 1 + \mathrm{H}(4) = 1 + \frac{7}{3} = \frac{10}{3}$
   The average height of 6 being root is $1 + \max(H(1), H(5)) = 1 + \mathrm{H}(5) = = 1 + \frac{14}{5} = \frac{19}{5}$
   The average height of 7 being root is $1 + \mathrm{H}(6) = 1 + \frac{49}{15} = \frac{64}{15}$
   Thus, $H(7) = \frac{1}{7} \times \frac{64}{15} + \frac{1}{7} \times \frac{19}{5} + \frac{1}{7} \times \frac{10}{3} + \frac{1}{7} \times \frac{8}{3} + \frac{1}{7} \times \frac{10}{3} + \frac{1}{7} \times \frac{19}{5} + \frac{1}{7} \times \frac{64}{15} = \frac{382}{105}$

   (f)

```
def H(n):
  if n == 0:    #it is impossible for n = 0,
   return 0     # but if we assume H(0) = 0, it will be easier for recursion
  if n == 1:
   return 0
  elif n == 2:
   return 1
  elif n == 3:
```

```
    return 5/3  # three base case
  else:
    a = 1       #start with 1 being root
    b = n - 1   # there are b node in the right of the root
    c = 0       # there are c node in the left of the root
    total = 0
    while a < n:
      total += (1 + max(H(b), H(c))) / n
      c += 1
      b -= 1
      a += 1    # using while loop to add all conditions
    return total

def max(a, b):
    if a > b:
     return a
    return b
```

This code is how i get c to e.

3. **[10 points]** In this question, we will define a data structure that implements an ADT called `WORD-COST-DICTIONARY` that stores a collection of English words. Each English word is a sequence of lower-case letters a-z, and has at least one vowel. Each English word has an associated cost. The ADT `WORD-COST-DICTIONARY` supports the following operations (where S is a given `WORD-COST-DICTIONARY`)

   - `LookUp(S, w)`: given a word string w, return the cost of that word.
   - `AddCost(S, w, c)`: add the word w to the dictionary, and assign c as its cost.
   - `DeleteCost(S, w)`: remove the word w from the dictionary.
   - `Change(S, w, c)`: change the cost of word w to the new cost c.
   - `CostBetween(S, v, w)`: return the total cost of all words that are lexicographically between v and w, inclusive (i.e. "hey" is lexicographically between "hello" and "hi"). Note that v and w *may or may not be* in the collection. You may assume that v appears earlier than w lexicographically.

   Design a data structure that implements `WORD-COST-DICTIONARY` using an **augmented AVL tree**. The worst case runtime of each of these operations should be $O(\lg n)$.

   For the below questions, do **not** repeat algorithms or runtime analyses from the class or the course notes, and refer to known results as needed.

   (a) The key of the AVL tree should be a string which represents the word. And it has 2 attribute, which is the cost of the word, and another one is the sum of the costs of its subtree.

   (b)

```
  def LookUp(S, w):
    x = S.root   # set x be the root of S
    while x.key != w:
      if x.key > w:   # if x.key is greater than w
        x = S.left.key #  try to find w in the left of current x
      else:           # if x.key is less than w
        x = S.right.key # try to find w in the right of current x
    return x.cost
    # in the worst case, it is finding the minimum or the maximum element in
    # the AVL, which will run as much time as its height, which is log(n).
    # thus O(log(n)).

   def AddCost(S, w, c):
     x = S.root # set x be the root of x
     while x.key is not None:
```

```
        if x.key > w:    # if x.key is greater than w
         x.sum += c
         x = S.left.key # check if there is an empty space in the left
        else:            # if x.key is less than w
         x.sum += c
         x = S.right.key # check if there is an empty space in the right
      x.key = w
      x.cost = c
      x.sum += c
      # in the worst case, it is adding its cost to the minimum or the maximum
      #element in the AVL, which will run as much time as its height, which is
      # log(n).
      # thus O(log(n)).

    def DeleteCost(S, w):
      x = S.root # set x be the root of x
      m = Lookup(S, w)
      while x.key ! = w:
        if x.key > w:    # if x.key is greater than w
         x.sum -= m
         x = S.left.key # check if there is an empty space in the left
        else:            # if x.key is less than w
         x.sum -= m
         x = S.right.key # check if there is an empty space in the right
      x.key = None
      x.cost = None
      x.sum -= m
     # Firstly, it runs a Lookup, which is O($\log(n)$). And then,
     # for the while loop, in the worst case, it will run as much
     # time as its height, which is #$\log(n)$.
     #thus this function is O(log(n)) + O(log(n)) = O(2log(n)) = #O(log(n))
```

(c)

```
def Change(s,w,c):
  x = S.root
  m = Lookup(S, w)
  while x.key ! = w:
     if x.key > w:    # if x.key is greater than w
      x.sum = x.sum + c - m
      x = S.left.root # check if there is an empty space in the left
     else:            # if x.key is less than w
      x = S.right.root # check if there is an empty space in the right
      x.sum = x.sum + c - m
  x.cost = c
  x.sum = x.sum + c - m
```

Firstly, it runs a Lookup, which is O(log(n)). At the worst case, we run the string w is the key of a node in the bottom of this binary search tree, and this function's while loop runs the time which is same as the height of the tree, which is log(n), thus, this function is $O(\log(n))$.

(d)

```
def minimum(S):
  x = S.root
  while x.left.key != None:
     x = x.left
 return x

```

```
7   def maximum(S):
8     x = S.root
9     while x.right.key != None:
10      x = x.right
11    return x
12
13  def CostBetween(S, v, w):
14    min = minimum(S)
15    max = maximum(S)
16    if v < min.key && w > max.key:
17     return S.root.sum
18    elif v > max.key || w < min.key:
19     return 0
20    elif v >= min.key && w > max.key:
21     if S.root.key < v:
22      subtree = S.root.right_subtree()
23      return CostBetween(subtree, v, max.key)
24     else:
25      subtree = S.root.left_subtree()
26      return CostBetween(subtree, v, S.root.key) + S.root.right.sum
27    elif v < min.key && w <= max.key:
28     if S.root.key > w:
29       subtree = S.root.left_subtree()
30       return CostBetween(subtree, min.key, w)
31     else:
32       subtree = S.root.right_subtree()
33       return S.root.left.sum + CostBetween(subtree, S.root.key, w)
34    else:     # when  v > min.key && w < max.key
35     if S.root.key < v:
36       subtree = S.root.right_subtree()
37       return CostBetween(subtree, v, w)
38     elif S.root.key > w:
39       subtree = S.root.left_subtree()
40       return CostBetween(subtree, v, w)
41     else:
42       subtree_a = S.root.right_subtree()
43       subtree_b = S.root.left_subtree()
44       return CostBetween(subtree_a, v, w)
45         + CostBetween(subtree_b, v, w)
```

For this function, firstly, we find the minimum and maximum node in the tree, which will spend a log(n).
Secondly, we sperate this function into four different conditions
In the first condition, when v < min.key and w > max.key. In this situation, all words in the tree has been counted. thus, the total cost of v to w should be equal to the total cost of the whole tree, which is equal to the sum at root of the tree. And the complexity is 0(1) in this case, so the total complexity is O(log(n)) under this condition.
In the second condition, when v > max.key or w < min.key. In this situation, since v to w is out of range, none of the word in tree has been count. thus, the cost of v to w is 0. And the complexity is O(1), so the total complexity is O(log(n)) under this condition.
In condition 3, when v ≥ min.key and w > max.key. In this condition, we separate into two conditions.
3.1 When the key of the root is less than v, that is means that v to w is all in the right side of the root. So, we can shrink the searching area from whole tree to only the right subtree of the whole AVL. In this case, for the worst case, when v = min, it will runs as much time as the height of tree, which is log(n). Thus, the complexity is O(log(n)).
3.2 When the key of the root is larger than v, that means that the whole right subtree has been counted,and only the part from v to the root has to be calculate. In this situation, it is equal to the total cost between v to the root add the total sum of the right side of the root, which is the sum of the right son of the root. In this case, the worst

case will run as much time as the heights of the tree, which is $\log(n) + 1$. Thus, the complexity is $O(\log(n))$.
In condition 4, when v < min.key and w ≤ max.key. In this condition, we separate into two conditions.
4.1 When the key of the root is greater than w,that is means that v to w is all in the left side of the root. So, we can shrink the searching area from whole tree to only the left subtree of the whole AVL. In this case, for the worst case, when w = max, it will runs as much time as the height of tree, which is $\log(n)$. Thus, the complexity is $O(\log(n))$.
4.2 When the key of the root is less than w, that means that the whole left subtree has been counted,and only the part from root to the w has to be calculate. In this situation, it is equal to the total cost between the root to w add the total sum of the left side of the root, which is the sum of the left son of the root. In this case, the worst case will run as much time as the heights of the tree, which is $\log(n) + 1$. Thus, the complexity is $O(\log(n))$.
In condition 5, when v > min.key or w < max.key, which means v to w is in the tree. we separate into three conditions. 5.1 When the key of the root is less than v, which means that v to w is all in the right side of the root. So we can shrink the searching area from whole tree to only the right subtree. In this case, in the worst condition, we will run as much time as its height, which is $\log(n)$. Thus, the complexity is $O(\log(n))$. 5.2 When the key of the root is greater than w, which means that v to w is all in the left side of the root. So we can shrink the searching area from whole tree to only the left subtree. In this case, in the worst condition, we will run as much time as its height, which is $\log(n)$. Thus, the complexity is $O(\log(n))$. 5.3 When the key of the root is between v to w, we must add the total cost from v to w in the left subtree and v to w in the right subtree. After separate, CostBetween(leftsubtree, v, w) is condition 3 , and CostBetween(rightsubtree, v, w) is condition 4. So it is $O(\log(n)) + O(\log(n))$, and the complexity is $O(\log(n))$.

Programming Question

The best way to learn a data structure or an algorithm is to code it up. In each problem set, we will have a programming exercise for which you will be asked to write some code and submit it. You may also be asked to include a write-up about your code in the PDF/TEXfile that you submit. Make sure to **maintain your academic integrity** carefully, and protect your own work. The code you submit will be checked for plagiarism. It is much better to take the hit on a lower mark than risking much worse consequences by committing an academic offence.

4. **[10 points]** In this question, we will solve a problem that we call **Gold Prospecting**. The function `solve_gold_prospecting` takes a list of commands that operate on the current collection of data. Your task is to process the commands in order and return the required list of results. There are two kinds of commands: `insert` commands and `get_gold` commands.

   An `insert` command is a string of the form `insert x`, where `x` is an integer. (Note the space between `insert` and `x`.) This command adds `x` to the collection.

   A `get_gold` command is simply the string `get_gold`. It retrieves the $\lceil \phi \times n \rceil$-th smallest element in the collection, where $\phi = 0.618$ is a variation of the **golden ratio**, and n is the current size of the collection. For this question, use $\phi = 0.618$ and no additional decimal places.

   Your goal is to implement `insert` in $O(\lg n)$ time worst-case, and `get_gold` in $O(1)$ time worst-case. Your algorithm should also have $O(n)$ space complexity. Here, n is the number of elements currently in the collection. The list returned by `solve_gold_prospecting` consists of the results, in order, from each `get_gold` command.

   Let's go through an example. Here is a sample call of `solve_gold_prospecting`:

```
solve_gold_prospecting(
  ['insert 10',
   'get_gold',
   'insert 5',
   'insert 2',
   'insert 8',
   'get_gold',
   'insert -5'
   'get_gold',
  ])
```

These commands corresponds to the following steps:

- The collection begins empty, with no elements.
- We insert 10. The collection contains just the integer 10.
- We then have our first `get_gold` command. The result is the $\lceil \phi \times 1 \rceil = \lceil 0.618 \rceil = 1$st smallest element currently in the collection, which is 10.
- We insert 5. The collection now contains 10 and 5.
- We insert 2. The collection now contains 10, 5, and 2.
- We insert 8. The collection now contains 10, 5, 2, and 8.
- Now we have our second `get_gold` command. The result is the $\lceil \phi \times 4 \rceil = \lceil 2.472 \rceil = 3$rd smallest element currently in the collection, which is 8.
- We insert -5. The collection now contains 10, 5, 2, 8, and -5.
- Now we have our third and final `get_gold` command. The result is the $\lceil \phi \times 5 \rceil = \lceil 3.09 \rceil = 4$th smallest element currently in the collection, which is 8.

So, the above call `solve_gold_prospecting` returns `[10, 8, 8]`, which are the three values produced by the `get_gold` commands.

For this code, i plan to use two heaps to achieve this, one of them is a minimum heap, and the other one is a maximum heap.
when inserting an new integer, we firstly compare it with the root of the minimum integer;
Firstly, if it is adding to a heap, which means that it is the first integer, i will add it into the minimum heap.
if it is smaller than the current minimum integer, we add this element into the maximum heap, and then compare the length of the maximum heap with the length of the whole heap times (1 - golden rate); if the length of the maximum heap is larger, then we will remove the root of the maximum heap(which is the largest integer), and add it into the minimum heap.
if the new integer we are trying to insert is larger than the current minimum integer, we will add it into the minimum heap, and then compare the length of the minimum heap with the length of the whole heap times the golden rate, if the length of the minimum heap is larger, we will remove the smallest integer in the minimum heap(which is the root), and then adding it into the maximum heap.
For function getgold, when there is only 1 element, we just need to get that element, which is in the minimum heap, otherwise, return the root of the max heap.

For function insert, we firstly did a inserting for normal heap, which is $O(\log(n))$, and then, for moving the element, we only move the root of both heaps, and adding it into the new heap, it is another $O(\log(n))$. Thus, new insert method is $O(\log(n)) + O(\log(n)) = O(2\log(n)) = O(\log(n))$.
For function getgold, since we only care about a root of heap, it is always $O(1)$.