Exercise 3: AVL Trees, Design Decisions, Hashing, Code Modeling

Due date: Friday August 2, 2019 at 11:59 pm

Instructions:

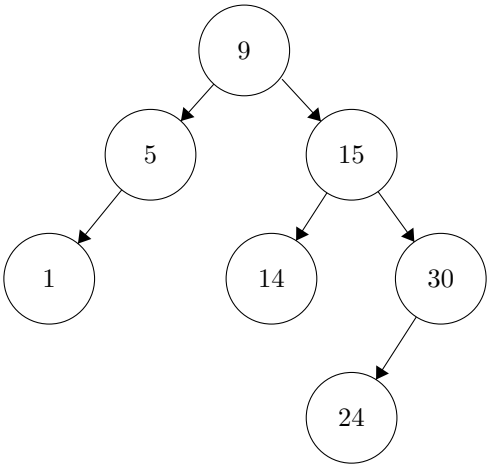
Submit a typed or neatly handwritten scan of your responses to the "Exercise 3" assignment on Gradescope here: https://www.gradescope.com/courses/52323. Make sure to log in to your Gradescope account using your UW email to access our course.

For more details on how to submit, see https://courses.cs.washington.edu/courses/cse401/18au/hw/submitting_hw_guide.pdf.

These problems are meant to be done **individually**. If you do want to discuss problems with a partner or group, make sure that you're writing your answers individually later on. Check our course's collaboration policy if you have questions.

1. AVL trees

Consider the following AVL Tree:



(a) Give an example of a value you could insert into this tree to cause no rotations. Draw the tree after inserting this value.

(b) Give an example of a value you could insert into the original tree to cause a single rotation. Draw the tree after inserting this value.

(c) Give an example of a value you could insert into the original tree to cause a double rotation. Draw the tree after inserting this value.

(d) Draw the resulting tree after inserting all of the following values (in order) into the original tree: 0, 8, 51, 19. Make sure to label your final answer if you choose to draw intermediate trees.

2. Design decisions

Imagine you've been tasked with writing a program to process all the files in your Downloads folder. Who knows what you'll be using it for - it's your own Downloads folder, after all. Which ADT and data structure implementation would help you represent and solve this problem the best? This design decision question is open-ended and multiple correct answers exist, given reasonable justification. To answer this general question, you must explicitly answer each of the following sub-questions to form your answer and receive credit.

For this problem, assume your Downloads folder has no subfolders. If you did have folders within folders within folders and so on ... you would probably consider some type of recursive representation like a tree. We are considering only **simple** array and node implementations (like those from the first week of the quarter), and you should not use a hash table or AVL tree for your dictionaries (for example).

(a) Out of a List, Stack, Queue, or Dictionary, which ADT do you choose and why does the ADT fit? Specify one use case for your Downloads folder and what method(s) of the ADT are involved to perform the use case.

(b) Assume runtime is the most important concern in this implementation. Our two main data structure implementations are an array-based structure or a node-based structure, like a linked list. Which of those two implementations will you choose and why does the implementation fit? Specify one operation (a specific method call or the use case from above, which might be multiple method calls) that this implementation is optimized for, and state what the runtime is in terms of a simplified tight big-Θ.

(c) Now choose a **different** ADT from the options provided to represent and solve this same problem.

(i) Which different ADT do you choose and why does the ADT fit? Specify a different use case for the Downloads folder than earlier, and what method(s) of the ADT that performing this use case involves. For this use case, explain whether or not it makes sense to use the ADT you selected in part a.

(ii) For your new ADT, again choose between a basic array-based or node-based implementation. Give one reason that you would choose this implementation for this new ADT. You may explain for a general use case of the ADT instead of the specific use case of the Downloads folder mentioned above.

3. Hashing

Consider the following hash tables implementing the dictionary ADT. Each table has an initial size of 10. Assume the hash tables **do not resize** in this problem.

(a) This hash table uses the hash function $h(x) = 2x$, and **linear probing** to resolve collisions. Show the table after inserting these keys:

$$4,\ 3,\ 24,\ 4,\ 14,\ 73,\ 74$$

(the corresponding values are omitted to make writing the results easier).

0	1	2	3	4	5	6	7	8	9

(b) This hash table uses the hash function $h(x) = x$, and **quadratic probing** to resolve collisions. Show the table after inserting these keys:

$$7,\ 27,\ 17,\ 37,\ 47,\ 22,\ 1$$

(the corresponding values are omitted to make writing the results easier).

0	1	2	3	4	5	6	7	8	9

(c) What is the load factor for each of the two hash tables after the final insertions?

(d) In this problem, we will see that the speed of your hash table methods can depend not just on the keys themselves, but also on the order of your insertions. Define a **failed probe** to be a time when your `put` call tries a location in the array, but it is already full. Suppose you have a hash table of size 10, with hash function $h(x) = x$ and your collision resolution strategy is quadratic probing. Your keys to insert are $0, 4, 5, 10, 11, 39$.

Give an ordering of the keys such that the total number of failed probes is at least ~~3~~ 4. (Potential hint: one possibility is to make one key have a large number of failed probes)

Give another ordering such that the total number of failed probes is at most 2.

4. More Code Modeling

In this problem we will analyze the code in `indexOfKth` method below. You may assume that all strings in `input` and the string `target` have constant length.

```
1  public int indexOfKth(String[] input, String target, int k) {
2      int seen = 0;
3      for (int i = 0; i < input.length; i++) {
4          if (input[i].equals(target)) {
5              seen++;
6              if (seen == k) {
7                  return i;
8              }
9          }
10     }
11     return -1;
12 }
```

(a) Describe how `input` and `target` could look to achieve the best-case runtime for `indexOfKth` method.

(b) Describe how `input` and `target` could look to achieve the worst-case runtime for `indexOfKth` method.

(c) Does a big-Θ running time exist for worst-case scenario for `indexOfKth` method? If so, state it. If not, give the tight big-$\mathcal{O}$. Your answer may depend on $n =$ `input.length` and/or the parameter k.

(d) Does a big-Θ running time exist for best-case scenario for `indexOfKth` method? If so, state it. If not, give the tight big-$\mathcal{O}$. Your answer may depend on $n =$ `input.length` and/or the parameter k.