Algorithms and Data Structures (ECE 345)

Issued on: Feb 3rd, 2023
Due on: Feb 19th, 2023

ASSIGNMENT 2

1. Assignment must be submitted by 11:59 PM EST on the due date through the Quercus submission system as a single PDF file.
2. Assignment must be completed **individually except the programming exercise** for which you can work in group of up to **two** students. Report for the programming exercise must be submitted separately at the same time. Only one report is needed for each group.
3. All pages must be numbered and no more than a single answer for any question.
4. Use LaTeX or Microsoft Word for your assignment writeup. You can find a LaTeX and a Word template for writing your assignment on Quercus.
5. Any problem encountered with submission must be reported to the head TA as soon as possible.
6. Unless otherwise stated, you need to justify the correctness and complexity of algorithms you designed for the problems.
7. Partial marks will be given if your algorithm requires more time or space complexity than specified.

EXERCISE 1 Algorithm Analysis, 15 points

A **ternary** heap is a heap that stores in a ternary tree. It is similar to binary heaps that we learned in class, but each node has at most **three** children. Similarly, every level is full except for the bottom level, and all the nodes at the bottom level are as far to the left as possible.

Consider a **MAX** ternary heap that is represented as an array A with size n. For element at index i, we have:

$$Left(i) = A[3i-1], 3i-1 \leq n$$

$$Mid(i) = A[3i], 3i \leq n$$

$$Right(i) = A[3i+1], 3i+ \leq n$$

$$Parent(i) = A[\lfloor (i+1)/3 \rfloor], i > 1$$

Describe an algorithm for $INSERT(A, key)$ (and $HEAPIFY(A, i)$ if necessary) and give a tight bound worst-case time complexity Θ in n.

EXERCISE 2 Induction, 15 points

We are looking for the number of binary trees with height h, where the height is measured by number of levels. For example, there is only one binary tree -the empty tree- with height 0; for height 1, there is only one binary tree which is the single-node tree; and there are 3 binary trees with with height 2. Give a recurrence formula for a sequence b (i.e., express b_h using $b_0, b_1, \cdots, b_{h-1}$), and prove using induction for all natural numbers h that b_h is the number of binary trees with height h.

$$b_0 = 1$$

$$b_h = ?, h \geq 1$$

EXERCISE 3 Linear Time Sorting, 15 points

You are given an array of strings W with size n and an integer k. Assume all strings have constant length, i.e., for any string $w \in W$, len(w)=$O(1)$. Devise an algorithm to find the k most frequent strings in $O(n)$ time using only $O(n)$ space. Partial credit will be awarded if your algorithm requires more time or space than what is specified above.

EXERCISE 4 AVL Trees, 15 points

Your friend Alice and you established a very unreliable communication protocol: She consistently sends messages to you in the form of (k, v), where k is the timestamp when he sends you the message, and v is the content of the message represented by an integer value. We call two messages (k_1, v_1) and (k_2, v_2) you received **out of order** if $k_1 < k_2$ but $v_1 > v_2$. Assume all messages have distinct k values and distinct v values. For each incoming packet, describe an $O(\log n)$ algorithm to determine if **any two messages** you have received are out-of-order, where n is the number of messages you have already received.

EXERCISE 5 Hashing, 10 points

Demonstrate the insertion of keys 23, 15, 19, 30, 28, 33, 36, 22 into a hashed table where collisions are resolved by open addressing with **quadratic probing**. Let the table have 7 slots, let the auxiliary hash function be $h'(k) = k \mod 7$ and let $c_1 = 1$, $c_2 = 1$. For any keys that can not be placed, list the key and give a reason.

EXERCISE 6 Programming Exercise, 30 points

(In)secure Password Storage/Checking: In this programming assignment, you will experiment with simple and (in)secure password storage/checking, and you will measure some performance statistics.

1. **Outline:**

   - Assume that you will be given a file containing a list of existing passwords. The name of the file will be `passwords.txt`, and the format will be one password per line without comma separators. Assume the file will reside in the same path as your executables. A sample file will be provided to you on Blackboard. Your program will need to parse the file and hash the passwords into a hash table. You may assume that your program will be tested on files with $100 - 10000$ passwords.
   - Each password in `passwords.txt` will be alphanumeric **without** any complex characters such as `!, ?, #, &,` etc. It will only consist of a mix of lower and/or upper case English alphabet letters with digits $\in \{0, \ldots, 9\}$. Each password will be 6 to 12 characters long. For example, `ece345LoVe` is a valid password, whereas `HATEece345HATE` is invalid due to length violation (and probably other reasons...!), and `Pa33word!!??` is invalid due to containing complex characters.
   - You are free to decide how to implement your hashing. Use only methods described in lecture. You are **not** allowed to use cryptographic libraries such as `OpenSSL`. Security is not our concern in this exercise.
   - Your executable should be named as `checkpass`, and it's behavior should be as follows:
     - It should be callable from command line as `checkpass passwords.txt password`, where `passwords.txt` is a file containing passwords and `password` is user provided. For example: `checkpass passwords.txt 23CdB9a13`

For C/C++ please include a `Makefile` to build your sources into the executable. For any other compiled language, please follow the same format as for C/C++.
For Java please create a `Makefile` that compiles your sources and a `checkpass.sh` script that runs your code for the given input, with the correct classpath.
For Python please name your script `checkpass.py`, have `#!/usr/bin/env pythonX` (where `X` is the version of python you are using) as the first line of your file. Additionally please do `chmod +x checkpass.py` so that you can run your file as `./checkpass.py passwords.txt password`. For any other scripting language (supported by the UG machines), please follow the same format as for python, except name your file as per that language (ie for perl use `checkpass.pl`)

- Your program then should print in standard output `VALID` if and only if the password is valid according to the constraints described above AND the password does not already exist in `passwords.txt` AND the reverse of the password does not exist in `passwords.txt`. Think how to make the check for reverse passwords efficient. In any other case your program will print `INVALID`.
- If the password entered is valid then your program should hash it and store the password (not its hash) into `passwords.txt`, and then it should terminate. If the password is not valid then the program should simply terminate after outputting `INVALID`.

2. **Deliverables:**

- Your full source code. Any code that does not build or does not run will receive a mark of 0. All code will be marked on the UG machines.
- A written report with readable graphs (with proper lables, grid ticks, and legends as needed) for each group. The report must address the following points:
  - Implementation details regarding your hashing approach. What is the size of your hash table and which hash function are you using? Are you applying chaining or open addressing? If you are using open addressing state what probing sequence you are implementing.
  - A brief justification of your implementation decisions above.
  - For a fixed number of entries $n = 1000$, vary the size of your table to get various load factors. For each load factor, run your code on **five different** `password.txt` files which you will create. Each file should include 1000 password entries. Plot the **average number of collisions vs. load factor**. You will need several values of load factor to see a trend on your plot. Briefly explain what you observe in the plot and why.
    NOTE: To set up this experiment, you can easily create another version of your program where there is no password input from the command line. You only need to parse the `passwords.txt` files that you will create and insert the entries in your hashtable. However, ***DO NOT*** *submit this version of the code.*