University of Waterloo
CS240E, Winter 2021
Assignment 4

Due Date: Wednesday, Mar 24, 2021 at 5pm

The integrity of the grade you receive in this course is very important to you and the University of Waterloo. As part of every assessment in this course you must read and sign an Academic Integrity Declaration (AID) before you start working on the assessment and submit it **before the deadline of March 24th** along with your answers to the assignment; i.e. **read, sign and submit A04-AID.txt now or as soon as possible**. The agreement will indicate what you must do to ensure the integrity of your grade. If you are having difficulties with the assignment, course staff are there to help (provided it isn't last minute).

The Academic Integrity Declaration must be signed and submitted on time or the assessment will not be marked.

Please read `http://www.student.cs.uwaterloo.ca/~cs240e/w21/guidelines/guidelines.pdf` for guidelines on submission. **Each written question solution must be submitted individually to MarkUs as a PDF** with the corresponding file names: a4q1.pdf, a4q2.pdf, ... , a4q6.pdf.

It is a good idea to submit questions as you go so you aren't trying to create several PDF files at the last minute. **Remember, late assignments will not be marked but can be submitted to MarkUs after the deadline for feedback if you email cs240e@uwaterloo.ca and let the ISAs know to look for it.**

1. (6 marks) A *MultiMap* is an ADT that allows to store multiple values associated with keys. Thus, `Insert`(k, v) may be called even if key k already existed in data structure, and it adds the value v as one of the values associated with k. (You may assume that v was not previously associated with k.) `Search`(k) returns *all* values that are associated with key k, and `Delete`(k) deletes all of them.

   Assume that the keys are uniformly distributed integers. Let N be the number of keys and n be the total number of values that are stored in the dictionary, noting that $n \gg N$ is possible. Describe a realization of ADT MultiMap with expected space $O(n)$, where the operations have the following run-time: *insert*(k, v) has expected time $O(1)$ *search*(k) and *delete*(k) both have worst-case run-time $O(1 + s)$, where s is the number of values associated with k. You do not need to give pseudo-code for the methods, but describe the idea in detail.

2. (6 marks) Recall our hash-function for strings: map string w (with characters in ASCII) to its list $\langle c_0, c_1, \ldots, c_{s-1} \rangle$ of ASCII-codes and then set

$$h(w) := (c_0 R^{s-1} + c_1 R^{s-2} + \cdots + c_{s-1}) \bmod M$$

for some positive integers R and M. Suppose we choose $R = M+1$. What undesirable properties would the resulting hash function have? (Hint: compute the code for "stop", "pots", and "post".) Justify your answer. See Appendix B in the textbook for modular arithmetic rules that might be useful.

3. (2+4(+5)+2 = 8(+5) marks) We have seen one method of obtaining a universal family of hash-functions in class. This assignment discusses another one. Let us assume that all keys come from some universe $\{0, \ldots, U-1\}$, where $U = 2^m$. Therefore any key k can be viewed as bitstring x_k of length m by taking its base-2 representation.

   Let us assume further that the hash-table-size M is $M = 2^b$ for some integer b, with $b < m$. To choose a hash-function, we now randomly choose each entry in a $b \times m$-matrix H to be 0 or 1 (equally likely). Then compute $h_k = (Hx_k)\%2$, where x_k is now viewed as a vector and '%2' is applied to each entry. The output is a b-dimensional vector with entries in $\{0, 1\}$; interpreting it as a length-b bitstring gives a number $\{0, ..., M-1\}$ that we use as hash-value $h(k)$. For example, if $k = 18$, $m = 5$, $b = 3$ and H is as shown below, then $h(k) = 1$ since

$$\underbrace{\begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}}_{H} \quad \underbrace{\begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}}_{\text{18 as length-5 bitstring}} \%2 = \underbrace{\begin{pmatrix} 0 \\ 2 \\ 1 \end{pmatrix}}_{Hx_k} \%2 = \quad \underbrace{\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}}_{\text{1 as length-3 bistring}}$$

   (a) Let H be the above matrix, $m = 5$ and $b = 3$. Consider the keys 9 and 13. What are their hash-values? Show your work.

   (b) Consider again $m = 5, b = 3$ and keys $k = 9$ and $k' = 13$. Consider the same matrix H, except that the bits in the third column are randomly chosen. What is the probability that $h(k) = h(k')$? Justify your answer.

   (c) (Bonus) Show that (for any m, b) this method of choosing the hash function gives a universal hash function family, or in other words, $P(h(k) = h(k')) \leq \frac{1}{M}$ for any two keys $k \neq k'$.

   (d) This method for obtaining universal hash-functions is much less popular than using the Carter-Wegman functions. Why do you think that that might be the case? (Expected length of answer is 1-3 sentences.)

4. (5 marks) What is the least number of nodes a quadtree on eight points may have? (We assume that every interior node in a quad-tree has exactly four children, i.e., we do not delete nodes that store an empty region.) Give an example and argue why it is minimal. You only have to show the tree-structure; no need to show suitable points.

5. (2 + 5 + 5 + 5(+3) = 17(+3) marks)

A *skewed kd-tree* is a *kd*-tree where the splits are allowed to be less even: If a node v has n_v points in its subtree, then its sibling stores at least $n_v/2$ and at most $2n_v$ points in its subtree.

(a) For $n = 7$ and $n = 9$, show a skewed *kd*-tree that stores n points and has the maximum possible height among all such trees. (You need not prove that it is maximal.) You only have to show the tree-structure; no need to show suitable points.

(b) Give an upper bound on the height of a skewed *kd*-tree that stores n points. Give an exact bound (no asymptotics).

Also state what your bound evaluates to for $n = 7, 9$. For full credit, your bound must be at most one bigger than the height you achieved in part (a) for $n = 7, 9$.

(c) One would *insert* in a skewed *kd*-tree as follows. First insert the point where it needs to be (by splitting at an appropriate leaf). Then check size-balances at the ancestors, and (if needed) re-build a maximal subtree where the size-balance is violated. Consider the following potential function:

$$\Phi(\cdot) := c \sum_{v \in V} \log n_v \cdot \max\{0, |n_{v.left} - n_{v.right}| - 1\}$$

where as before n_v denotes the number of points stored in the subtree rooted at v, and c is a constant. Show that during an *insert* without rebuilding, this potential function increases by $O(\log^2 n)$.

You may use without proof that $\log(x + 1) \leq \log x + \frac{1}{x}$ for $x > 0$.

(Motivation: With this potential function all operations have amortized run-time $O(\log^2 n)$, but to keep the assignment from getting too long you do *not* have to show this.)

(d) We can do a range-search on a skewed *kd*-tree in exactly the same manner as for a *kd*-tree. While the run-time is not as good as for a *kd*-tree, it is better than for quad-trees since at least we do not visit every node. Prove this. Specifically, prove that in a skewed *kd*-tree, the number of boundary nodes in any range-search is in $O(n^c)$ for some $c < 1$.

Any $c < 1$ will give you full credit, but we recommend $c = 0.9$. For up to three **bonus**-marks, make constant c as small as you can.

6. (5+5+2=12 marks) A *range-counting-query* is like a range search, except that you only need to report *how many* items fall into the range, you do not need to list which items they are.

(a) Describe how any balanced binary search tree can be modified such that a range counting query can be performed in $O(\log n)$ time (independent of s, the number of points in the query-interval). Briefly state the changes needed, then describe the algorithm for the range counting query.

(b) Now consider the 2-dimensional-case: Describe an appropriate range-tree based data structure such that you can answer range-counting-queries among 2-dimensional points in time $O((\log n)^2)$. Then describe the algorithm for the range counting query.

(c) Prof. Confused thinks that they can do a similar approach for kd-trees, so report the number of points in $O(\log^2 n)$ time. Why is this not correct?

For all questions concerning points and their data structures, the points are in general position.