Computer Science CSC263H
St. George Campus

February 1, 2018
University of Toronto

Homework Assignment #3

Due: February 15, 2018, by 5:30 pm

- You must submit your assignment as a PDF file of a typed (**not** handwritten) document through the MarkUs system by logging in with your CDF account at:

  `https://markus.teach.cs.toronto.edu/csc263-2018-01`

  To work with one or two partners, you and your partner(s) must form a group on MarkUs.

- The PDF file that you submit must be clearly legible. To this end, we encourage you to learn and use the LaTeX typesetting system, which is designed to produce high-quality documents that contain mathematical notation. You can use other typesetting systems if you prefer, but handwritten documents are not accepted.

- If this assignment is submitted by a group of two or three students, the PDF file that you submit should contain for each assignment question:

  1. The name of the student who *wrote* the solution to this question, and
  2. The name(s) of the student(s) who *read* this solution to verify its clarity and correctness.

- By virtue of submitting this assignment you (and your partners, if you have any) acknowledge that you are aware of the homework collaboration policy that is stated in the csc263 course web page: `http://www.cs.toronto.edu/~sam/teaching/263/#HomeworkCollaboration`.

- For any question, you may use data structures and algorithms previously described in class, or in prerequisites of this course, without describing them. You may also use any result that we covered in class, or is in the assigned sections of the official course textbook, by referring to it.

- Unless we explicitly state otherwise, you should justify your answers. Your paper will be marked based on the correctness and completeness of your answers, and the clarity, precision, and conciseness of your presentation.

- Your submission should be no more than 5 pages long in a 10pt font.

Question 1. (1 marks) In this question, you must use the insertion and deletion algorithms as described in the "Balanced Search Trees: AVL trees" handout posted on the course web site.

a. Insert keys 4, 18, 15, 8, 3, 20, 11, 1, 5, 9, 13, 10 (in this order) into an initially empty AVL tree, and show the resulting AVL tree T, including the balance factor of each node.

b. Delete key 4 from the above AVL tree T, and show the resulting AVL tree, including the balance factor of each node.

In each of the above questions, only the final tree should be shown: intermediate trees will be disregarded, and not given partial credit.

Question 2. (1 marks) Consider an abstract data type that consists of a *set* S of integers on which the following operations can be performed:

Add(i) : Adds the integer i to S. If this integer already is in S, then S does not change

Average(t) : Returns the average of all elements of S that are less than or equal to the integer t. If all the elements of S are greater than t, then return 0.

Describe how to implement this abstract data type using an augmented AVL tree T. Each operation should run in $O(\log n)$ worst-case time, where $n = |S|$. Since this implementation is based on a data structure and algorithms described in class and in the AVL handout, you should focus on describing the extensions and modifications needed here.

a. Give a precise and full description of your data structure. In particular, specify what data is associated with each node, specify what the key is at each node, and specify what the auxiliary information is at each node. In particular, what is (are) the augmented field(s), and what identity should this (these) fields satisfy. Illustrate this data structure by giving an example of it (with a small set of your own choice).

b. Describe the algorithm that implements each one of the two operations above, and explain why each one takes $O(\log n)$ time in the worst-case. *Your description and explanation should be in clear and concise English.* For the operation AVERAGE, you should also give the algorithm's high-level pseudocode.

Question 3. (1 marks) You are given a list L of n, not necessarily distinct, integers. Your task is to devise an algorithm that outputs a list of the distinct integers in L, in order of non-increasing frequency (the frequency of an element in L is the number of times it occurs in L). For example, if L is $2, 5, 4, 4, 2, 3, 4, 3$, then a suitable output is $4, 3, 2, 5$; the only other suitable output for this list is $4, 2, 3, 5$. In this example, L contains only small integers, but this is not always the case. You cannot make any assumption on the integers in L (e.g., their maximum size, or distribution). In particular, some integers of L can very large, and so it is not feasible to use tables of size proportional to these integers.

a. Give a Hash Table-based algorithm for this problem whose *expected* time complexity is $O(n)$, under some Hash Table assumptions that you must clearly state. Explain why, your algorithm achieves this time complexity under these assumptions.
What is the *worst-case* time complexity of your algorithm? Use the Θ notation and justify your answer.
HINT: As part of your algorithm, you may find useful to discover a way to sort n numbers in the range 0 to n in $O(n)$ worst-case time.

b. Give an algorithm for this problem whose *worst-case* time complexity is $O(n \log n)$. Explain why your algorithm achieves this time complexity.

[***The questions below will not be corrected/graded. They are given here as interesting problems that use material that you learned in class.***]

Question 4. (0 marks) [Augmented AVL for Dynamic Storage Allocation]

Dynamic storage allocation is a resource management task that is required at several levels of computer systems: for example, an operating system must allocate chunks of main memory to processes, and a file system must allocate chunks of disk space to files. We can describe the problem of storage management in general terms as follows. There is a ***store*** S of N equal-size ***cells***, indexed as $S[i]$ for $i \in \{1, 2, \ldots, N\}$. In practice, the cells of the store are individually addressable units of some type of memory. For example, they may be bytes, words or pages of main memory; or they may be blocks or sectors of disk space.

The storage manager maintains at all times a ***pool of available fragments***. Intuitively, a fragment is a maximal sequence of consecutive cells that are available to be allocated to a process that needs them. When a process requests a fragment of size ℓ, the storage manager determines if the pool of available fragments contains a fragment of size $k \geq \ell$. If it does not, the request is rejected. If the pool contains a fragment of size $k \geq \ell$, the storage manager removes it from the pool, splits it into one fragment of size ℓ and one fragment of size $k - \ell$, allocates the former to the requesting process, and returns the latter (if $k - \ell \neq 0$) to the pool. When a process is done with a fragment that it was previously allocated, it returns that fragment to the storage manager. The storage manager returns the fragment to the storage pool after coalescing it with its adjacent fragments, if such fragments are presently in the pool.

More precisely, a fragment is a pair (a, k), where a, k are positive integers such that $a + k - 1 \leq N$; a is the fragment's ***start address*** (the index of the first cell in the fragment) and k is its ***size*** (the number of cells in the fragment). Thus, fragment (a, k) corresponds to the subarray $S[a..a + k - 1]$ of the store S. Two fragments $F = (a, k)$ and $F' = (a', k')$ are ***adjacent*** if the start address of one is immediately after the last cell of the other (i.e., $a = a' + k'$ or $a' = a + k$); F and F' are ***overlapping*** if some cell belongs to both of them (i.e., the intervals $[a, a + k - 1]$ and $[a', a' + k' - 1]$ are not disjoint). The pool of available fragments is a set of fragments that does not contain any adjacent or overlapping fragments. Initially, the pool of available fragments consists of a single fragment, namely $(1, N)$.

We can formulate storage management as an abstract data type. The object manipulated by this ADT is the pool of available fragments, and there are two operations:

- ALLOCATE(P, ℓ): Intuitively, this is a request to allocate a fragment of size ℓ.

  P is a pool of available fragments, and ℓ is an integer such that $1 \leq \ell \leq N$. If P contains no fragment whose size is at least ℓ, the operation returns "reject". If P contains some fragment, say (a, k), such that $k \geq \ell$, the operation returns (a, ℓ), and adjusts P by replacing (a, k) with $(a + \ell, k - \ell)$, if $k > \ell$ — or replacing it with nothing, if $k = \ell$.

- RELEASE$(P, (a, \ell))$: Intuitively, this returns a previously allocated fragment to the pool.

  P is a pool of available fragments, and (a, ℓ) is a fragment that does not overlap any fragment in P. The operation adjusts P by returning the fragment (a, ℓ) to it, after coalescing it with any adjacent fragments.

When the operation ALLOCATE(P, ℓ) is invoked and the pool P contains several fragments of size ℓ or more, there are various strategies that the storage manager could use to decide which fragment to use. Following are three common strategies:

- ***First-fit:*** choose the fragment of adequate size that has the smallest possible start address.
- ***Best-fit:*** choose a smallest fragment of adequate size.
- ***Worst-fit:*** choose a largest fragment of adequate size.

In this question, you must describe how to implement the storage management ADT described above using an *augmented AVL tree* (which represents the pool P of currently available fragments). Your implementation should have the property that the ALLOCATE operation uses the ***first-fit strategy***. Furthermore, each operation ALLOCATE and RELEASE should take $O(\log n)$ time in the worst-case, where n is the number of fragments currently in the pool P.

Since this implementation of P is based on a data structure and algorithms described in class and in the AVL handout, you should focus on describing the extensions and modifications needed here.

1. Give a precise and full description of your data structure. In particular, specify what data is associated with each node, specify what the key is at each node, and specify what the auxiliary information is at each node. Illustrate this data structure by giving an example of it (on a small pool P of available fragments of your own choice).

2. Describe the algorithms for ALLOCATE and RELEASE, and explain why each one takes $O(\log n)$ time in the worst-case.

Question 5. (0 marks) [Computing the join of two relations]

In this question you will explore different algorithms to perform the *join* of two relations, a very important operation in relational databases. We consider a simplified setting where we wish to compute the join of relation R with attributes A and C, and relation S with attributes B and C. Notice that the two relations have a common attribute, C. Each relation is a set of pairs. For each pair (a, c) of R, a is a value of attribute A and c is a value of attribute C. Similarly, for each pair (b, c) of S, b is a value of attribute B and c is a value of attribute C. It may be useful to think of R as a two-dimensional array, with two columns, the first corresponding to A and the other corresponding to C: each row of the two-dimensional array corresponds to a pair (a, c) of R; so the number of rows of the array is equal to the number of pairs in R. Similar comments apply to S.

The *join* of R and S, denoted $R \bowtie S$, is a relation with attributes A, B and C (i.e., the union of the attributes of R and S). It consists of the set of all triples (a, b, c) such that (a, c) is a pair in R and (b, c) is a pair in S. Notice that the two pairs that are being joined to produce the triple (a, b, c) of $R \bowtie S$ must have *the same value* in the common attribute C.

Another way of thinking about $R \bowtie S$ is as follows: We consider every possible combination of a pair (a, c) from R and a pair (b, c') from S. If $c = c'$ — i.e., if the two pairs have the same value in the common attribute — then this combination contributes the triple (a, b, c) to the join. If $c \neq c'$ then this combination contributes nothing to the join.

For example, suppose that R is the relation *Teaches* with attributes *ProfName* and *CourseId*; and S is the relation *Takes* with attributes *StudName* and *CourseId*. A pair (p, c) is in *Teaches* if and only if professor p teaches course c. A pair (s, c) is in *Takes* if and only if student s takes course c. The join *Teaches* $\bowtie$ *Takes* is the relation with attributes *ProfName*, *StudName*, *CourseId*; it contains a triple (p, s, c) if and only if professor p teaches student s in course c.

The obvious way to compute $R \bowtie S$ is to consider each combination of pairs from R and S and create a triple from each such combination where the two pairs have the same value in the common attribute. Obviously this algorithm takes time $\Theta(mn)$, where m is the number of pairs in R and n is the number of pairs in S.

In a way, this is the best we can hope for: If all pairs in R and all pairs in S have the same value in the common attribute C, then there will be mn triples in $R \bowtie S$, and we can't create a set of mn triples faster than in $\Omega(mn)$ time! In practice, however, it is very unlikely that every pair in R joins with every pair in S; in most cases, the size of $R \bowtie S$ is much smaller than mn. For instance, in our example of the *Teaches* and *Takes* relations, if these happened to describe, respectively, the courses taught by U of T faculty and the courses taken by U of T students in a particular semester, the first relation would

contain about 4,000 pairs (approximately 2,000 professors each teaching 2 courses) and the second relation would contain about 250,000 pairs (approximately 50,000 students, each taking 5 courses). In this case the product mn is approximately one billion, while the actual size of *Teaches* $\bowtie$ *Takes* will be approximately 250,000 — more than three orders of magnitude smaller! (There could be more records in *Teaches* $\bowtie$ *Takes* than in *Takes*, in case a course is co-taught by several professors, but such instances are rare, so typically the size of *Teaches* $\bowtie$ *Takes* will be only a little bit larger than that of *Takes*.)

In view of this example, we would like to have an algorithm that computes $R \bowtie S$ in time $\Theta(k+f(m,n))$, where k is the actual size of $R \bowtie S$ and $f(m,n)$ is as small a function as possible.

In the following two questions you are asked to devise and describe such algorithms. You should describe each algorithm in *clear and precise* English, and *illustrate the data structure(s)* that you use with simple example(s); no pseudocode is necessary (but you can use pseudo-code as an additional explanation). To the extent your algorithm makes use of algorithms or data structures discussed in this course or in its prerequisites, you can simply state what algorithm(s) and data structure(s) you use without describing them in detail.

In the following, assume that each of R and S is given as a two dimensional array, where each column corresponds to an attribute and each row corresponds to a pair of the relation. You can **not** make any assumption on the size or distribution of values contained in R or S (i.e., the values associated with attributes A, B, or C). In particular, some of these values may be very large (e.g., they may be 128-bit integers), and so it is **not** feasible to use arrays of size proportional to these values.

Let k be the size of the join $R \bowtie S$, and N be the size of the larger of the two given relations R and S, i.e., $N = max(n,m)$.

a. Describe an algorithm that computes $R \bowtie S$ in *worst-case* time $O(k + N \log N)$. In other words, the worst-case running time of your algorithm, *with respect to all the inputs R and S that have $N = max(n,m)$ and whose join is of size k*, is $O(k + N \log N)$. Your should describe your algorithm as we explained above (*start with a few English sentences explaining the high-level idea of your algorithm*).

Explain why the worst-case time complexity of your algorithm is $O(k + N \log N)$.

b. Describe an algorithm that computes $R \bowtie S$ in *expected* time $\Theta(k + N)$ *under some reasonable assumptions that you should state*. Your should describe your algorithm as we explained above (*start with a few English sentences explaining the high-level idea of your algorithm*).

Explain why the expected running time of your algorithm is $\Theta(k + N)$.

What is the *worst-case* time complexity of this algorithm? Justify your answer.

Hint: Note that R or S may have many pairs with the *same* value c in attribute C. You should think about this case carefully when you formulate your assumptions, and when you analyse the time complexity of your algorithm.