Computer Science CSC263H
St. George Campus

February 28, 2019
University of Toronto

Homework Assignment #5

Due: March 14, 2019, by 5:30 pm

- You must submit your assignment as a PDF file, named **a5.pdf**, of a typed (**not** handwritten) document through the MarkUs system by logging in with your CDF account at:

  `https://markus.teach.cs.toronto.edu/csc263-2019-01`

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

- The **a5.pdf** 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 **a5.pdf** PDF file that you submit should contain for each assignment question:

  1. The name(s) of the student(s) 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 **a5.pdf** submission should be no more than 5 pages long in a 10pt font.

Question 1. (1 marks) We want a data structure that maintains a set I of integers and supports the following two operations:

1. INSERT(x), insert a new integer x into I (assume that x is not already in I).
2. SEARCH(x), which returns TRUE if integer x is currently in I, and returns FALSE otherwise.

If we keep I in a sorted array A, then, using binary search, the worst-case time for a SEARCH(x) is $O(\log n)$, where n is the size of I. But the worst-case time for an INSERT(x) is $O(n)$, and the *amortized* insertion time (i.e., the worst-case total time to execute n INSERTs divided by n) is also $O(n)$. Our goal is to decrease the *amortized* insertion time, may be at the cost of increasing the worst-case time of SEARCH.

To do so, we keep I in a *doubly linked list L of sorted arrays* as follows. Let n be the number of elements in I, and $< b_{k-1}, b_{k-2}, \ldots, b_0 >$ be the binary representation of n; note that $\Sigma_{i=0}^{i=k-1} b_i 2^i = n$ and $k = O(\log_2 n)$. For every $i = 0, 1, \ldots, k-1$ such that $b_i = 1$, the doubly-linked list L contains a sorted array A_i of size 2^i; A_i stores 2^i elements of I in increasing sorted order. The arrays of L are listed in order of increasing size. Each integer of I is in exactly one of the sorted arrays of L. Note that although each array of L is sorted, there is no particular relationship between the elements in different arrays of L.

a. Draw two instances of the data structure L, one for set $I = \{6, 8, 4, 13, 9\}$ and one for set $I = \{21, 12, 7, 14, 5, 16, 10\}$.

In the questions below, you should give as efficient algorithms as possible. Describe your algorithms in clear and concise English, there is no need to use pseudocode.

b. Describe an algorithm to perform a SEARCH(x) operation with this data structure. Give a good upper bound on the worst-case time complexity of your algorithm (using the O notation) and justify your answer.

c. Describe an algorithm to perform an INSERT(x) operation with this data structure. Give a good upper bound on the worst-case time complexity of your algorithm (using the O notation) and justify your answer.

HINT: Think about how we insert an element in a Binomial Heap, and use the Merge part of MergeSort.

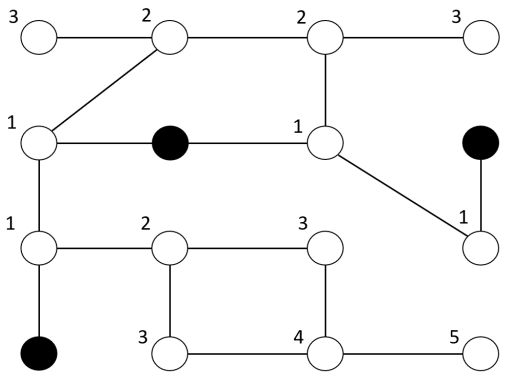
d. Suppose we execute a sequence of n INSERTs starting from an empty set I. Determine a good upper bound on the *amortized* time of an INSERT (i.e., the worst-case total time to execute these n INSERTs divided by n). Justify your answer in two different ways, i.e., give two separate proofs, each proof using a different argument (e.g., use aggregate analysis and the accounting method).

e. Describe an algorithm to perform a DELETE(x) operation, i.e., given a pointer to integer x in one of the arrays of L, remove x, in $O(n)$ time in the worst case. Explain why the worst-case time complexity of your algorithm is $O(n)$.

Question 2. (1 marks) You are given an *undirected, unweighted, and connected* graph $G = (V, E)$ that represents a set of houses and hospitals connected by roads as follows: each vertex in V is colored white if it represents a house, or black if it represents a hospital, and each edge (u, v) in E represents a road between vertices u and v that can be traversed in one unit of time.

Assume that you are given G by its adjacency list. We are seeking an algorithm to solve the following problem $\mathcal{P}$: compute for each house vertex, the shortest time to reach some hospital vertex.

An example of a graph G and the shortest time to hospitals from each house in G is shown below:



a. Furio, an infamous csc263 alumnus, claims that *if the number of houses is constant*, then a graph algorithm that he learned in csc263 can be used in a very simple way to solve problem $\mathcal{P}$ in $O(|V| + |E|)$ time in the worst-case.

Briefly explain why Furio is right; in particular, state which algorithm he has in mind and how it can be used to solve $\mathcal{P}$.

In the questions below, the number of houses is *not* constant and the number of hospitals is $k \geq 1$ (also *not* a constant).

b. Paulie, another infamous csc263 alumnus, claims the csc263 algorithm that Furio used in part (a) can be used to solve problem $\mathcal{P}$ in $O(k(|V| + |E|))$ time in the worst-case. Briefly explain why Paulie is right.

c. Tony, their csc263 teacher, claims he can do better than Paulie: he claims that there is an algorithm that can solve problem $\mathcal{P}$ in only $O(|V| + |E|)$ time in the worst-case.

Is Tony right or is he just bragging? If he is right, describe Tony's algorithm in *clear and concise* English. You should also briefly argue why its worst-case time complexity is $O(|V| + |E|)$. If Tony is just bragging, please ignore this question.

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

Question 3. (0 marks) A *multi-set* is like a set where repeated elements matter. For example, $\{1, 8, 3\}$ and $\{3, 3, 8, 1, 8\}$ represent the same *set* but different *multi-sets*. Consider the abstract data type that consists of a multi-set of integers S and supports the following operations:

- INSERT(S, x): insert integer x into multi-set S;
- DIMINISH(S): delete the $\lceil |S|/2 \rceil$ largest integers in S.
  (For example, if $S = \{3, 3, 8, 1, 8\}$, then after DIMINISH(S) is performed, $S = \{3, 1\}$.)

A simple data structure for this ADT, and the corresponding algorithms for each operation, are as follows: The elements of S are stored in an unsorted linked list, and the size of S is stored in a variable n.

- INSERT(S, x): Append x at the end of the list; increment n.
- DIMINISH(S):
  i. $m \leftarrow$ MED(S) find the median of the elements in S
  ii. loop over all elements in S: keep all those $< m$, delete all those $\geq m$
  iii. add as many copies of m as required to have exactly $\lfloor n/2 \rfloor$ elements remaining
  iv. $n \leftarrow \lfloor n/2 \rfloor$

The above uses an algorithm MED that returns the "median" of any list of integers. To simplify this question, we use a slightly non-standard definition of "median": MED returns the $\lceil n/2 \rceil$-th largest integer in the list (including duplicates); for example MED$([3, 2, 1, 1, 3, 3, 3])$ returns 3. Also for this question, assume that MED performs at most $5n$ pairwise comparisons between integers during its execution on any list of length n.

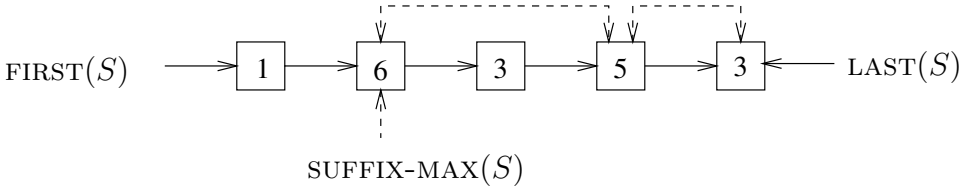
Prove that when we execute an arbitrary sequence of INSERT and DIMINISH operations starting from an empty set S, the *amortized* cost of an operation, *in terms of the number of pairwise comparisons between the elements of the set*, is *constant*. To prove this, use the accounting method to analyse the amortized complexity of the INSERT and DIMINISH algorithms: (a) state clearly what charge you assign to each operation, (b) state and prove an explicit credit invariant, and (c) use your credit invariant to justify that the amortized cost of an operation is constant.

Question 4. (0 marks) Consider the abstract data type MAXQUEUE that maintains a sequence S of integers and supports the following three operations:

- DEQUEUE(S): removes the first element of S and returns its value.
- ENQUEUE(S, x): appends the integer x to the end of S.
- MAXIMUM(S): returns the largest integer in S, but does not delete it.

An element x is a *suffix maximum* in a sequence if all elements that occur after x in the sequence are strictly smaller in value. For example, in the sequence 1,6,3,5,3, the suffix maxima are the second, fourth, and fifth elements.

One way to implement the MAXQUEUE abstract data type is to use a singly linked list to represent the sequence S, with additional pointers FIRST(S) and LAST(S) to the first and last elements of that list; and to have a doubly linked list of the suffix maxima (arranged in the same order as they are in the sequence), with an additional pointer SUFFIX-MAX(S) to the first element of that list. For example, the sequence $S = 1, 6, 3, 5, 3$ would be represented as follows:



a. Describe algorithms to implement each of the three operations (namely, DEQUEUE(S), ENQUEUE(S, x), and MAXIMUM(S)) for the MAXQUEUE abstract data type using the above representation. *For each operation, first explain the basic idea of how your algorithm works in a few clear English sentences*; you can then give more details using English (and pseudo-code if necessary).

Your algorithms should be such that the operations have amortised complexity $O(1)$, assuming that initially the MAXQUEUE contains the empty sequence.

b. Prove that your algorithms achieve the desired amortised complexity. *Hint:* Use the accounting method.

c. Determine an asymptotic tight bound for the worst-case cost of an *individual operation* in a sequence of m operations. Justify your bound.

d. Find an alternative implementation of this abstract data type such that the worst-case time needed for an individual operation is $O(\log n)$, where n is the number of elements in the sequence. The amortised complexity in this case need not be $O(1)$.

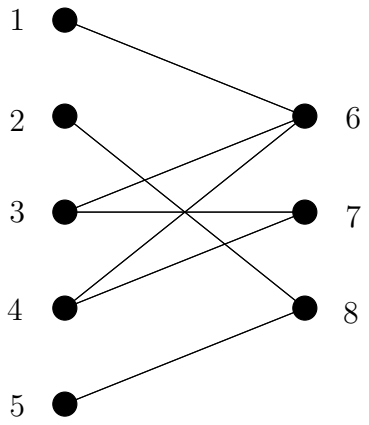
Question 5. (0 marks) Prove that every connected undirected graph G contains at least one vertex whose removal (along with all incident edges) does *not* disconnect the graph. Give an algorithm that for any connected graph $G = (V, E)$, given by its adjacency list, finds such a vertex in $O(|V| + |E|)$ time in the worst case. Justify your algorithm's correctness and worst-case time complexity.

Question 6. (0 marks) Let $G = (V, E)$ be an undirected graph. G is ***bipartite*** if the set of nodes V can be partitioned into two subsets V_0 and V_1 (i.e., $V_0 \cup V_1 = V$ and $V_0 \cap V_1 = \varnothing$), so that every edge in E connects a node in V_0 and a node in V_1. For example, the graph shown below is bipartite; this can be seen by taking V_0 to be the nodes on the left and V_1 to be the nodes on the right.
Note that G is bipartite if and only if every connected component of G is bipartite.

a. Prove that if G is bipartite then it has no *simple cycle* of odd length.[1] Hint: Give a proof by contradiction.

b. Prove that if G has no simple cycle of odd length (i.e., every simple cycle of G has even length) then G is bipartite. (Hint: Suppose every simple cycle of G has even length. Perform a BFS starting at any node s. Assume for now that G is connected, so that the BFS reaches all nodes; you can remove this assumption later on. Use the distance of each node from s to partition the set of nodes into two sets, and prove that no edge connects two nodes placed in the same set.)

[1] Recall that a cycle is simple if it contains each node at most once.



c. Describe an algorithm that takes as input an undirected graph $G = (V, E)$ in adjacency list form. If G is bipartite, then the algorithm returns a pair of sets of nodes (V_0, V_1) so that $V_0 \cup V_1 = V$, $V_0 \cap V_1 = \varnothing$, and every edge in E connects a node in V_0 and a node in V_1; if G is not bipartite, then the algorithm returns the string "not bipartite." Your algorithm should run in $O(n + m)$ time, where $n = |V|$ and $m = |E|$. Explain why your algorithm is correct and justify its running time. (Hint: Use parts (a) and (b).)