THE UNIVERSITY OF HONG KONG

FACULTY OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE

COMP2119 Introduction to Data Structures and Algorithms

Date: 16 Dec, 2019 Time: 9:30am - 12:30pm

Write Your **University Number here**: _____

Please read the following before you begin.

1. Only approved calculators as announced by the Examinations Secretary can be used in this examination. It is your responsibility to ensure that your calculator operates satisfactorily, and you must record the **name and type of the calculator** used in the first page of your answer script, or here:

   _____

2. You are advised to spend around 5 minutes to go through all the questions before you begin writing. Allocate the time for each part of the question carefully.
3. If you have a printer or plan to annotate the given PDF file, you may write your answer in the designated space. Alternatively, if you are going to first write your answer on blank pieces of paper, the designated space will give you an idea of the length of the solution.
4. Attempt ALL questions. The full mark for the exam is 100 points.

For marking use:

Question 1

Question 2

Question 3

Question 4

Question 5

1 Asymptotic Notation (20 points)

(a) (12 pt) Solve the following recurrence relations and give the simplest expression using Big-Theta notation. You do not need to give a full proof.

(i) $T(n) = T(\lfloor\sqrt{n}\rfloor) + \log n$, $T(0) = T(1) = 1$.

(ii) $T(n) = T(n-3) + n^2$, $T(0) = T(1) = T(2) = 1$.

(iii) $T(n) = T(\lfloor\sqrt{n}\rfloor) + 1$, $T(1) = 1$.

(iv) $T(n) = T(\lfloor\frac{n}{2}\rfloor) + T(\lfloor\frac{n}{3}\rfloor) + T(\lfloor\frac{n}{6}\rfloor) + n$, $T(0) = 0$.

(b) (8 pt) For each of the following statements, state whether it is true or false.

(i) $n^2 = O(n)$.

(ii) $n! = O(2^n)$

(iii) $n^{\frac{1}{\log n}} = O(1)$.

(iv) $n^n = O(2^{n^{\log \log n}})$.

2 Dictionary Abstract Datatype (15 points)

Recall that the dictionary abstract data type stores a collection of keys, and supports the operations *insert*, *delete*, and *search*. If the keys are totally ordered, then it can also support *FindMin* and *FindMax*. Each key can be stored using $O(1)$ space.

In each of the following scenarios, describe briefly how the specifications can be achieved, or explain why it is not possible.

(a) (3 pt) Suppose all the elements are given at the beginning, and there is no insertion or deletion afterwards. Only the search operation will be used later, and a good guarantee on the worst case search time is required.

(b) (3 pt) Elements will be inserted and deleted frequently. The average running time for insert, delete and search is $O(1)$.

(c) (3 pt) All operations insert, delete and search are used frequently. A guarantee on the worst case running time is needed.

(d) (3 pt) Insertions will be done frequently. However, only *FindMin* (i.e. returning the element with minimum key) and *DeleteMin* (i.e., deleting the element with minimum key) are needed. Moreover, only average $O(1)$ time is needed for each operation.

(e) (3 pt) Each key has value in $\{1, 2, \ldots, n\}$. Each insertion or deletion has worst-case $O(1)$ time. Moreover, *FindRange*$[a..b]$ needs to return all elements whose keys are in $[a..b]$, where the running time is proportional to the number of elements returned.

3 Binary Search Tree (15 points)

(a) (2 pt) What is the property that needs to be satisfied by a binary search tree?

(b) (2 pt) In addition to being a binary search tree, what property needs to be maintained for an AVL tree?

(c) (6 pt) Suppose in a binary tree with n nodes, at every node, the difference of heights of the left and the right sub-tree is at most b, where b is some parameter that is at least 1. Prove that the height of the tree is at most $O(b \log n)$.

(d) (5 pt) Recall that in a binary search tree, each node is defined as follows.

```
struct node {
   int key;
   node *left;
   node *right;
   node *parent;
}
```

Write a procedure in a C-like language that takes a pointer to the root of a binary tree, and returns true *if and only if* it is a binary search tree that also satisfies the AVL property; otherwise, it returns false.

4 Sorting (30 points)

For each of the following scenarios, describe a method (briefly in words) to satisfy the specifications. Please provide brief explanation. You may use pseudocode too, if you prefer.

(a) (6 pt) Suppose the elements are given in an array of length n, where each key is an arbitrary number. Only $O(1)$ extra space should be used, and elements in the array should be rearranged into a sorted order. Worst-case running time should be $O(n \log n)$.

(b) (6 pt) Suppose the elements are given in a singly-linked list, where only the pointer to the head is available to the algorithm. The algorithm should return the sorted elements in a singly-linked list, by restructuring the links of the list without creating or destroying nodes. Moreover, extra memory used should be as little as possible. If n is the number of elements in the list, the algorithm should run in $O(n \log n)$ time.

(c) (6 pt) Suppose you are given a collection of n elements, each of which is an integer in the range $\{1, 2, \ldots, n\}$. Give an algorithm with optimal asymptotic running time that returns the set of elements that appears at least once in the collection in sorted order.

(d) (6 pt) Suppose you are given an array of n elements, where each element is an integer in $\{1, 2, 3, \ldots, n^k\}$ for some small constant $k \geq 2$. The task is to sort the elements with asymptotic running time strictly better than $O(n \log n)$. (You may use extra $O(n)$ space.)

(e) (6 pt) Suppose the elements are given in an array of length n, where each key is an arbitrary number. However, you are guaranteed that for each element, its initial index position and its final index position in sorted order differ by at most B. The value B is known to the algorithm. The task is to rearrange the elements in the array into sorted order, with running time $O(n \log B)$. You may use $O(B)$ extra space.

(In the next question, we describe a data structure to solve this problem using only $O(1)$ extra space.)

5 Shifting Wrapped-Around Heap (20 points)

Recall that an array $H[1..K]$ can be used to represent a binary tree, where the children of $H[i]$ are $H[2i]$ and $H[2i+1]$. For $K = 6$, below is an example of a min-heap, where the root contains the minimum value.

i	1	2	3	4	5	6
$H[i]$	3	5	4	8	7	6

(a) (3 pt) Draw the binary tree represented in the above example.

(b) In order to solve question 4(e) using only $O(1)$ extra space, we need to design a heap data structure that is "shifting" within some other array $A[0..n-1]$, where $K \leq n$. We assume that the size K of the heap does not change. There are two variables s and r with the following invariants.

- The elements of the heap are stored in $A[s..s+K-1]$.
- The heap elements are "wrapped around" such that the root of the heap is stored at $A[r]$.

Study the following example carefully, where the above heap H is stored inside A with $s = 2$ and $r = 4$.

i	0	1	2	3	4	5	6	7	8	9	$\cdots$
$A[i]$	?	?	7	6	3	5	4	8	?	?	$\cdots$

(3 pt) In terms of i, s, r and K, define an expression $h(i)$ such that the element at $H[i]$ is stored at $A[h(i)]$. In the above example, $h(1) = 4$ and $h(6) = 3$.

(c) (4 pt) Implement the operation `ShiftHeap()` that is called when $s + K < n$. The operation overwrites the entry $A[s + K]$ with $A[s]$. To avoid losing data, the operation should return the original value of $A[s + K]$. After calling `Shift()` to the above example, the result is as follows.

i	0	1	2	3	4	5	6	7	8	9	$\cdots$
$A[i]$	?	?	?	6	3	5	4	8	7	?	$\cdots$

(d) (4 pt) Suppose the root of the heap $A[r]$ is overwritten with some potentially large value that might cause the heap property to be violated. Write an operation `FixRoot()` to maintain the heap property again. (You may use $h(i)$ from above.)

(e) (6 pt) Write an algorithm to solve 4(e) using only $O(1)$ extra space. In addition to the above subroutines, you may assume the following subroutines are already defined for you.

- `BuildHeap()`: It constructs a heap from elements in $A[0..K-1]$ and sets $s = r = 0$; the running time is $O(K)$, using $O(1)$ extra space.
- `SortHeap()`: It rearranges the elements in $A[s..s+K-1]$ into sorted order in time $O(K \log K)$ using $O(1)$ extra space.

END OF PAPER