CSCB63 Midterm
Data Structure & Algorithm Analysis

Duration – 100 minutes

Examination Aids: No Aids Allowed

Student #: |__|__|__|__|__|__|__|__|__|

Last Name:____________________ **First Name:**_______________

Do **not** turn this page until you have received the signal to start. In the meantime, please fill out the identification section above, and read the instructions below carefully.

This term test consists of 13 questions on 9 pages (including this one), printed on one side of the paper. When you receive the signal to start, please make sure that your copy of the examination is complete.

Answer each question directly on the examination paper, in the space provided, and use the reverse side of the pages for rough work. If you need more space for one of your solutions, use the reverse side of the page and indicate clearly the part of your work that should marked. Be aware that concise, well thought-out answers will be rewarded over long rambling ones. Also, unreadable answers will be given zero (0) so write legibly.

#1: Algorithm Analysis	**_______/30**
#2: Dictionary	**_______/10**
#3: Priority Queue	**_______/28**
#4: Ordered Dictionary	**_______/16**
#5: Data Structure Comparison	**_______/16**
***Total*:**	**______/100**

Algorithm Analysis [*30 marks total*]

1) [2 marks] Algorithm A runs in $O(n \log n)$ and algorithm B runs in $\Omega(n^2)$. Is it true that given the same input, A always takes less time to run than B? If true, explain why. If false, explain and give a counterexample.

Answer:
False. Asymptotic notation, as the name implies, only holds in reference to the asymptotic behavior of a function, that is to say, how it behaves for particularly large inputs. Take for example the functions $8675309 \cdot n \log n$ and $0.0000000000000001n^2$ For a value such as say, $n = 2$, the second function's value is clearly smaller, despite it having a larger asymptotic bound.

2) [2 marks] Is it true that $n/100 = \Omega(n)$? If it's true find a value for c and N, otherwise provide a counter example.

Answer:
True. $n/100 < c * n$ for $c = 1/200$.

3) [2 marks] Is it true that $\log n$ is $O(\log(2n))$? If it's true find a value for c and N, otherwise provide a counter example.

Answer:
We know: $\log n < 1 + \log n = \log 2 + \log n = \log(2n)$ and that this holds for all values of $n>0$. Thus, by definition $\log n$ is $O(\log(2n))$ for witness pair $(c=1, N=0)$.

4) [4 marks] Is it that if given a hash table with more slots than keys, and collision resolution by chaining, the worst case running time of a lookup would be a constant time? Justify your answer.

Answer:
False. In the worst case we get unlucky and all the keys hash to the same slot, for $\Theta(n)$ time.

5) [4 marks] In the following method, let x be the length of xArray, y be the length of yArray, and z be the length of zArray. This method's running time is $\Theta(\underline{x^2z + xy})$.

```
public void futz(int[] xArray, int[] yArray, int[] zArray) {

      for (int i = 0; i < xArray.length; i++) {

            for (int j = 0; j < zArray.length; j++) {

                  for (int k = 0; k < i; k++) {

                        yArray[j] = yArray[j] + zArray[j] *
                          (xArray[j] * (xArray[i] - xArray[k]);
                  }
            }

            for (int j = 0; j < yArray.length; j++) {

                  yArray[j] = 2 * yArray[j];
            }
      }
}
```

6) [8 marks] Consider the following sorting algorithm:

Step 1: Loop through the list of input items, inserting each item in turn into an ordinary binary search tree.

Step2: Then, perform an inorder traversal of the tree, and output the items in inorder.

Answer:

The worst-case running time of this algorithm is in $\Theta(\underline{\quad n^2 \quad})$.

The best-case running time of this algorithm is in $\Theta(\underline{\quad n \log n \quad})$.

7) [8 marks] A student claims to have come up with a sorting algorithm that can take a valid binary heap, and sort its items in $O(n)$ time - which if it true will make him famous! since sorting algorithm mostly perform in $O(n \log n)$ bounds. The student claims that by taking advantage of the fact that the items in a binary heap are already "partly sorted", his algorithm performs in $O(n)$. The sorting step involves comparing two items to find which is smaller (or larger). Give a convincing argument that the student's claim is bogus!

Note: the student's claim isn't completely far-fetched though. Indeed, we can merge two sorted lists of keys into a single sorted list in $O(n)$ time; that's an example of beating the $O(n \log n)$ bound with keys that are "partly sorted". So, the claim requires a thoughtful argument to debunk it.

Hint: Think of the binary heap operations discussed in class and their cost.

The following illustration of a heap is provided for your convenience;

☒	2	4	3	4	7	6	4	5	6	7	9	

Answer:

Here are two different correct answers.

- A key observation is that `bottomUpHeap` is a comparison-based algorithm that transforms any array into a binary heap in $O(n)$ time. Therefore, if the student is correct, then we can create an $O(n)$-time comparison-based sorting algorithm (for *any* list of keys, not just a binary heap) by putting `bottomUpHeap` and the student's algorithm together. But an $O(n)$-time comparison-based sorting algorithm cannot exist. (Note that this argument would not work if `bottomUpHeap` had $\Theta(n \log n)$ worst-case running time; its linear running time is central to the argument.)

- Although the items in a heap are partially sorted, the *leaves* of a heap are not. For example, if all the internal nodes of a heap are zeros, and all the leaves are unique positive numbers, we can permute the leaves any way we like, and it's still a valid binary heap. At least half the nodes in a binary tree are leaves, so there are at least $(n/2)!$ such permutations. So sorting the keys in an n-key binary heap is at least as hard as sorting $n/2$ keys from scratch, which takes $\Omega((n/2) \log (n/2)) = \Omega(n \log n)$ time.

Dictionary [*10 marks total*]

8) [5 marks] A `SimpleBoard` object stores an `8 x 8` array grid in which each cell has value 0, 1, or 2. Suppose you want to store lots of `SimpleBoards` in a hash table. The following hash code will not distribute `SimpleBoard` objects evenly among the buckets? Explain why. (Assume the compression function is good.)

```
public class SimpleBoard {

      private int[][] grid;

      public int hashCode() {
            int code = 0;
            for (int i = 0; i < 8; i++) {
                  for (int j = 0; j < 8; j++) {
                        code = code + (i + j) * grid[i][j];
                  }
            }
            return code;
      }
}
```

Answer:
This hash code does not differentiate between two different cells for which $i + j$ is the same, so moving a chip along a diagonal does not change a board's hash code. Thus, collisions are likely.

9) [5 marks] Every word falls into one of eight categories: nouns, verbs, pronouns, adjectives, adverbs, prepositions, conjunctions, and interjections. Suppose you want to be able to look up the category of any English word in $O(1)$ time. You could use a hash table to map each word to a category. The hash table's chains could use ListNodes, wherein each ListNode contains a reference to a word, and an extra field (perhaps a short) that indicates the category of the word.

However, we want to use as little memory as possible. We can save some memory by eliminating the extra field from each ListNode. Explain how we can do this, yet still determine a word's category in $O(1)$ time. Hint: you can adjust the data structure in some other way, use a combination of data structures, or both. But don't increase the memory use by more than a small constant. (That's an added constant, not a constant factor.)

Answer:
Use eight hash tables--one for each category. You can tell what category a word is in (if any) simply by checking for its presence in each of the eight hash tables. Each hash table's size needs to be chosen proportionately to the number of words in each category, so that each table will have the same load factor as one big hash table would. That way, the eight hash tables offer $O(1)$-time operations and take up the same amount of memory as one big table would.

Priority Queue [*28 marks total*]

10) [12 marks] What does the following binary heap look like after each of the operations executed (<u>operations independent from each other</u>).

☒	2	4	3	4	7	6	4	5	6	7	9	

a) `insert(1)`

Answer:

☒	1	4	2	4	7	3	4	5	6	7	9	6

b) `removeMin()`

☒												

11) [16 marks] Let A be an array representation for a heap. Provide pseudocode for a function DELETE-KEY(A,k) that deletes key k from a heap, while preserving the heap property. You may assume that exactly one instance of the key occurs in the heap. The complexity should be $O(n)$, where n is the number of elements in the heap. The length of the heap is specified by a variable A.length. You must write out the pseudocode for any function you wish to use, including functions introduced in class.

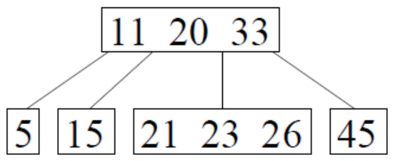
Answer:

```
DELETE-KEY(A,k)
     i = 1 // (Assuming 1-based indexing)
     while A[i] != k
          i = i + 1
     A.length = A.length – 1
     if i <= A.length
          A[i] = A[A.length + 1]
          BUILD-MAX-HEAP(A, i) // Code must also be
                               // supplied
```

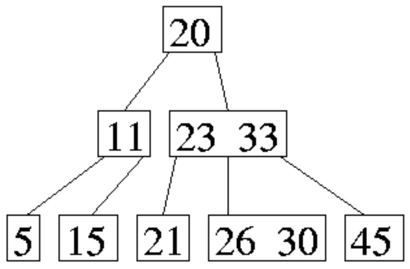
Ordered Dictionary [*16 marks total*]

12) [16 marks] Draw the following 2-3-4 trees after the execution of the specified operation (you can use either top-down or bottom-up)

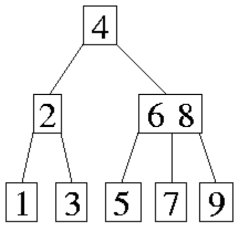
a) `insert(30)`



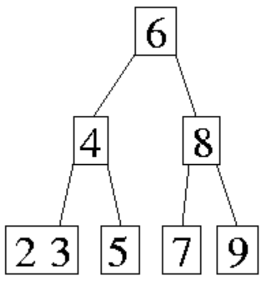
Answer:



b) `remove(1)`



Answer:



Data Structure Comparison [*16 marks total*]

13) [16 marks] For each of the following pairs of data structures, name one advantage that the first data structure has over the second and one advantage that the second has over the first. Your answers should be *brief*. Note that "having more operations" is *not* an acceptable answer since the Abstract Data Type determines the set of operations. Further, note that we are assuming high-quality implementations are readily available, so "easier to implement" is also *not* an acceptable answer.

a) [Heap vs. Sorted Array] used as a Priority Queue.

Answer:
Heap advantage: All PQ operations are O(log n) time.
Sorted Array advantage: Fast if no insertions after initialization or getMax takes O(1) time.

b) [Hashing with Chaining and Table-Doubling vs. Balanced BST] used as a Dictionary.

Answer:
Hashing advantage: O(1) expected time (but bad worst-case)
Balanced BST advantage: O(log n) worst-case time

c) [Array of Lists, one list for each priority - sometimes called a *bounded-height priority queue* - vs. Heap] used as a Priority Queue.

Answer:
Array of Lists advantage: O(1) per operation (but requires small number of priorities) or FIFO for items with tied priorities
Heap advantage: Can allow arbitrary number of different priorities.

d) [Sorted Array vs. Balanced BST] used as a Dictionary.

Answer:
Sorted Array advantage: Uses less space
Balanced BST advantage: Insert and remove are faster (O(log n)).

END OF MIDTERM