CSE 332 : 21AU Simulated Midterm Solutions

Instructions

- This is a simulated exam. It is designed to be done closed-book/closed-note, and with no access to electronic devices.
- The allotted time is 50 minutes.
- Read directions carefully, especially for which problems require you to show work or provide an explanation.
- Don't forget to answer the reflection questions after you finish the exam (you may spend as much time on those as you wish).
- On a non-simulated exam, we can only give partial credit for work that you've written down.
- If you run out of room on a page, indicate that the answer continues use the back of that page. Try to avoid writing on the very edges of the pages, as on non-simulated exams we scan your answers.
- The last two pages of the exam are a blank sheet for scratch work and a formula sheet.

Advice

- If you feel like you're stuck on a problem, you may want to skip it and come back at the end if you have time.
- Look at the question titles on the cover page to see if you want to start somewhere other than problem 1.
- We recommend ripping off the last two pages from the exam right away.
- Remember to take deep breaths.

Question	Max points
1. Big-O	18
2. Code Analysis	16
3. $\mathcal{O}, \Omega,$ and Θ, oh, my!	12
4. Write a recurrence	9
5. Solve a recurrence	10
6. B-Tree	11
7. PIVOT! (AVL rotations)	6
8. Remove Many	6
9. Priority AVLs	12
Total	**100**

1. Big-O [18 points]

For each of the following operations or functions given below, give a simplified, tight big-$\mathcal{O}$ bound. This means that, for example, $\mathcal{O}\left(2^{n!}\right)$ or $\mathcal{O}\left(5n^2 + 7n + 3\right)$ are unlikely to get points. Unless otherwise specified, all logs are base 2. Explanation or justification are not required.

For questions that ask for the running time of operations, assume the most efficient implementation of those data structures is used.

For array-based structures (including hash tables), assume that you do not have to resize.

(a) Best case for insertion into a heap with n elements. **Solution:**

$O(1)$.

(b) Worst case for `removeMin` from a heap with n^3 elements. **Solution:**

$O(\log n)$. Recall that $\log(n^3) = 3\log(n)$, so the exponent is a constant factor and shouldn't be included.

(c) **Best** case running time to find and remove an element from a BST with n elements. **Solution:**

$O(1)$. If most of the tree is a "stick" and you're removing a leaf on the not-stick-part (for example), you could have very fast deletion.

(d) $f(n) = \log_3\left(2^n\right)$ **Solution:**

$O(n)$

(e) Worst case for n insertions into an initially empty heap. **Solution:**

$O(n \log n)$

(f) $T(n) = \begin{cases} T(n/2) + 5 & \text{if } n \geq 32 \\ 17 & \text{otherwise} \end{cases}$ **Solution:**

$O(\log n)$ This is (except for changes in the constants) the recurrence for binary search.

(g) Removing the first k elements from an array-based queue with n elements (assume that k is much less than n). **Solution:**

$O(n)$

(h) Find the median element of an AVL tree (assume that there are n elements in the AVL tree, and n is odd). **Solution:**

$O(n)$

(i) The average case for an insertion into a separate chaining hash table where there are currently $n = \sqrt{\texttt{TableSize}}$ elements (give your answer in terms of n, not λ or `TableSize`). **Solution:**

$O(1)$

2. Code Analysis [16 points]

Describe the worst-case running time for the following pseudocode functions in Big-$\mathcal{O}$ notation in terms of the variable `n`. Your answer must be tight and simplified. **You do not have to show work or justify your answers for this problem.**

(a)
```
void sand(int n) {
    for(int 1 = 1; i < Math.pow(2, n); i *= 2) {
        if(n < 200000) {
            for(int j = 0; j < n * n * i; j += 2) {
                System.out.println("shells");
            }
        }
        else{
            for(int i = 0; i < n; i += 3) {
                System.out.println("crabs");
            }
        }
    }
}
```

(a) ____________________

Solution:

$O(n^2)$. The outer loop runs n times (because i*= 2) and the inner loop runs n/3 times but when we remove constant factors we get n^2

(b)
```
int leaves(int n) {
    if (n < 150) {
        return n;
    }
    return n * 2 * leaves(n / 2);
}
```

(b) ____________________

Solution:

$O(\log n)$

(c)
```
void snow(int n) {
    int sum = 0;
    for(int i = 0; i < n*n; i++) {
        sum++;
    }
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < sum; j++) {
            System.out.println(”winter wonderland”);
        }
    }
}
```

(c) ____________________ **Solution:**

$O(n^3)$. The first loop runs n^2 time and sets sum $= n^2$. The second loop runs n (outer loop) * sum (inner loop) times, and since sum $= n^2$, this means it runs n^3 times.

(d)
```
void sun(int n) {
    int k = 0;
    for(int i = 1; i < n; i *= 2) {
        k++;
    }
    for(int i = 0; i < n; i++) {
        if(k % 5 == 0) {
            k--;
        }
        else {
            for(int j = 0; j < k; j++) {
                System.out.println(”let’s go to the beach”);
            }
        }
    }
}
```

(d) ____________________

Solution:

$O(n \log n)$ Regardless of whether we do the k- or not, we will only decrease k by at most one (no matter how many times we go through the loop), and every time except possibly the first we'll need to do the else branch. We'll do that else branch $\Theta(k) = \Theta(\log(n))$ times, for each of the n times through the outer loop, giving us the claimed $O(n \log n)$

3. $\mathcal{O}, \Omega,$ and Θ, oh my! [12 points]

For each of the following statements, indicate whether it is always true, sometimes true, or never true. You do not need to include an explanation. Assume that the domain and codomain of all functions in this problem are natural numbers.

(a) If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$ then $f(n)$ is $O(h(n))$.

Always True **Sometimes True** **Never True**

Solution:

Always True

(b) If $\log(f(n))$ is $\Theta(\log(g(n)))$ then $f(n)$ is $\Theta(g(n))$.

Always True **Sometimes True** **Never True**

Solution:

Sometimes True – the claim is true for $f(n) = n^2$ and $f(n) = n^3$, for example, but fails when $f(n) = 2^n$ and $g(n) = 3^n$.

(c) $f(n)$ is $O(f(n) \cdot \log n)$, but not $\Theta(f(n) \cdot \log(n))$.

Always True **Sometimes True** **Never True**

Solution:

Always True, $f(n)$ cannot be in $\Omega(f(n) \cdot \log n)$

(d) Let $f(n)$ be the worst-case running time of an insertion into a separate chaining hash table. $f(n)$ is $\Omega(1)$.

Always True **Sometimes True** **Never True**

Solution:

Always True

4. Write a recurrence [9 points]

Give a base case and a recurrence for the runtime of the following function. Use variables appropriately for constants (e.g. c1, c2, etc.) in your recurrence (you do not need to attempt to count the exact number of operations). **YOU DO NOT NEED TO SOLVE this recurrence**

```
int seattleWeather(int n) {
    if (n < 50) {
        for (int i = 0; i < 100; i++) {
            print ("It's still raining");
        }
        return 332;
    } else if (seattleWeather(n / 2) < 332) {
        for (int i = 0; i < n * n * n; i++) {
            print("It's over 100 degrees!");
        }
    }
    for (int i = 1; i < n; i *= 2) {
        print("It's a little cloudy");
    }
    print("It's sunny!");
    return 3 * n * seattleWeather(n / 3) + 10 * seattleWeather(n / 2);
}
```

$$T(n) = \begin{cases} \underline{\hspace{8cm}} & \text{for } n < 50 \\ \underline{\hspace{8cm}} & \text{for } n \geq 50 \end{cases}$$

Yay!! You do NOT need to solve *this* recurrence...

Solution:

$$T(n) = \begin{cases} c_0 & \text{for } n < 50 \\ T(\frac{n}{3}) + 2T(\frac{n}{2}) + c_1 \cdot \log_2 n + c_2 & \text{for } n \geq 50 \end{cases}$$

Notes: For the recursive case, we see that seattleWeather(n / 2) gets called twice since it is also called in the else-if condition. Notice that the return is always greater than or equal to 332 because the base case returns 332 and any other value also returns a value that is larger since it only adds to the previous return. So, the for loop that runs to n^3 is never executed. (So it works out that it's never over 100 degrees in Seattle!) For the $\log n$ component, notice that the i is multiplied by 2 at each step of the iteration.

5. Solve a Recurrence [10 points]

Suppose the running time of an algorithm satisfies the recurrence given below. Find the closed form for $T(N)$. **You may assume N is a large power of 3.** Your answer should *not* be in Big-$\mathcal{O}$ notation. Show the exact constants and bases of logarithms in your answer (e.g. do NOT use c_1, c_2 in your answer).
Your final answer must not have any summation symbols or recursion – you may find the list of summations and logarithm identities on the last page of the exam to be useful.
You must show your work to receive *any* credit.

$$T(n) = \begin{cases} 3T\left(\frac{n}{3}\right) + n^2 & \text{if n > 3} \\ 2 & \text{otherwise} \end{cases}$$

Solution:

$$\begin{aligned}
T(n) &= 3T\left(\frac{n}{3}\right) + n^2 \\
&= 3\left[3T\left(\frac{n}{3^2}\right) + \left(\frac{n}{3}\right)^2\right] + n^2 \\
&= 3^2T\left(\frac{n}{3^2}\right) + \frac{n^2}{3} + n^2 \\
&= 3^2\left[3T\left(\frac{n}{3^3}\right) + \left(\frac{n}{3^2}\right)^2\right] + \frac{n^2}{3} + n^2 \\
&= 3^3T\left(\frac{n}{3^3}\right) + \frac{n^2}{3^2} + \frac{n^2}{3} + n^2 \\
&\dots \\
&= 3^iT\left(\frac{n}{3^i}\right) + \sum_{j=0}^{i-1} \frac{n^2}{3^j}
\end{aligned}$$

We will need the i that hits the base case, in this case that's when $n = 3$, so we solve for:

$$\frac{n}{3^i} = 3 \implies n = 3^{i+1} \implies i + 1 = \log_3(n) \implies i = \log_3(n) - 1$$

Plugging in, we get:

$$3^{\log_3(n)-1}T(3) + \sum_{j=0}^{\log_3(n)-2} \frac{n^2}{3^j} = 2 \cdot \frac{n}{3} + n^2 \sum_{j=0}^{\log_3(n)-2} \left(\frac{1}{3}\right)^j = \frac{2n}{3} + n^2 \cdot \frac{\left(\frac{1}{3}\right)^{\log_3(n)-1} - 1}{1/9 - 1}$$

From here, we have no more summations, so you could stop here. You could simplify further, as follows:

$$\begin{aligned}
\frac{2n}{3} + n^2 \cdot \frac{\left(\frac{1}{3}\right)^{\log_3(n)-1} - 1}{1/3 - 1} &= \frac{2n}{3} + n^2 \cdot \frac{(3)^{-\log_3(n)+1} - 1}{1/3 - 1} \\
&= \frac{2n}{3} + n^2 \cdot \frac{(3)^{\log_3(n^{-1})+1} - 1}{1/3 - 1} \\
&= \frac{2n}{3} + n^2 \cdot \frac{3n^{-1} - 1}{-2/3} \\
&= \frac{2n}{3} + \frac{3n - n^2}{-2/3} \\
&= \frac{2n}{3} + \frac{3n^2}{2} - \frac{23n}{6} \\
&= \frac{3}{2}n^2 - \frac{7}{2}n
\end{aligned}$$

6. B-tree [11 points]

(a) Given the following parameters for a B-Tree and assuming M and L were chosen appropriately, what are M and L?
Page Size: 512 Bytes
Key Size: 8 Bytes
Pointer Size: 4 Bytes
Value Size: 16 Bytes per Record (does **NOT** include the key) [6 points]
Solution:

We start by defining the following variables.

- 1 page on disk is b bytes
- Keys are k bytes
- Pointers are t bytes
- Key/Value pairs are v bytes

We know that the amount of memory used by one leaf node is vL and the amount of memory used by one internal node is $tM + k(M-1)$. We want select values for M and L such that both equations are $\leq b$.

If we solve both equations for M and L, we obtain $M = \left\lfloor \dfrac{b+k}{t+k} \right\rfloor$ and $L = \left\lfloor \dfrac{b}{v} \right\rfloor$

Plugging in the given values, we get $M = \left\lfloor \dfrac{512+8}{4+8} \right\rfloor = 43$ and $L = \left\lfloor \dfrac{512}{16+8} \right\rfloor = 21$

Since you didn't have a calculator, an answer of the form $\left\lfloor \frac{512+8}{4+8} \right\rfloor$ would be considered simplified enough and get full credit.

(b) Now consider an alternative to the B-Tree called the C-Tree. The C-Tree is similar to the B-Tree except rather than having internal nodes only store the key, internal nodes now store both the key and the value (note that this means that keys and values that appear in internal nodes are no longer stored in the leaf nodes as well like they are in regular B-Trees). This way, we do not have to iterate all the way down to the leaf nodes every find!
Suppose you have an extremely large data set (where the relevant analysis for your code is the number of times you have to bring in a page from memory, instead of counting operations). If you care about best-case behavior, would you prefer a C-tree or a B-tree? Why? [3 points] **Solution:**

In the best case scenario, the key we are looking for will be close to the root. In a B-Tree, the location of the key relative to the root did not matter since we always had to iterate down to the leaf to get the corresponding value. However, in a C-Tree, the internal nodes store both the key and the value. As such, if a key is located close to the root, we will not have to iterate all the way down to the leaf node, saving us disk accesses. Thus, if we care about best-case behavior, we would prefer a C-Tree.

(c) If you care about worst-case behavior, why should you use a B-Tree? [2 points] **Solution:**

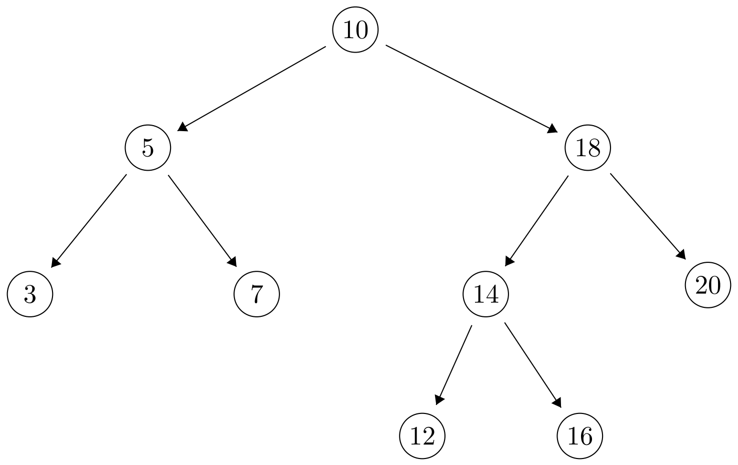
As a C-Tree stores both the keys and the values in the internal nodes, the M values will be less than those of a B-Tree with identical parameters. This results in the height of a C-Tree being larger than the height of a B-Tree as the number of elements grows. In the worst case, the key we are looking for will be stored in a leaf node. If the height of the B-Tree is less than the height of the C-Tree, we will have to make fewer disk accesses to reach the leaf node for the B-Tree than for the C-Tree. Thus, if we care about worst-case behavior, we should use a B-Tree.

7. PIVOT! [6 points]

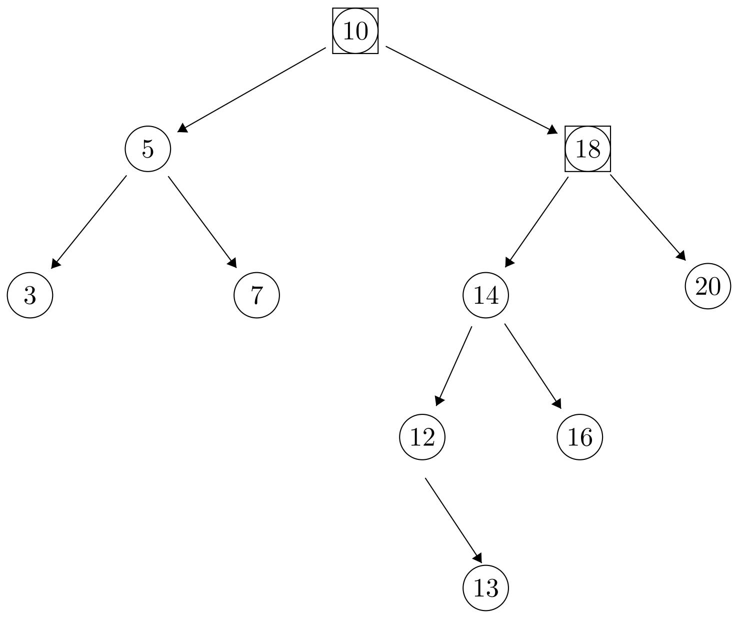
For the following AVL trees and key to insert:

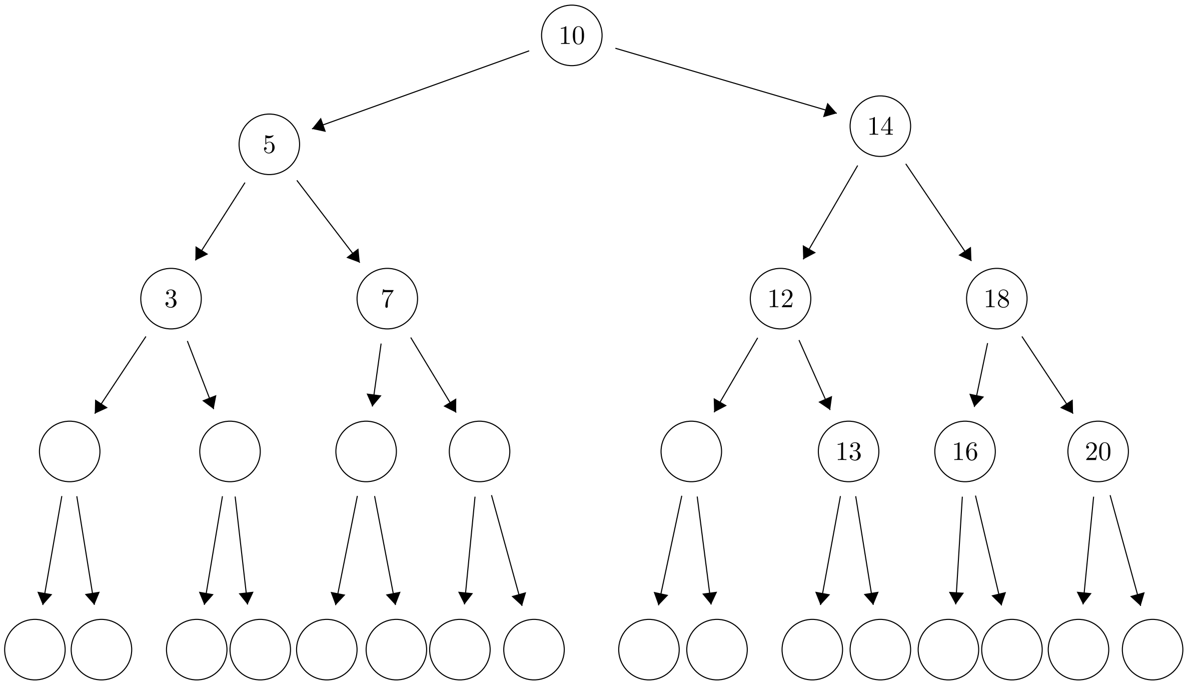
- On the top (given) tree, draw the new node where it will be inserted into the tree (before any rotations occur).
- On the top (given) tree, put a box around **every** imbalanced node in the tree (if any).
- In the bottom (empty) tree, draw the final tree after any rotations are performed. If no rotations are needed, you may write "no rotations" instead of filling in the tree. If a node does not exist in the final tree, leave that circle blank.

Item to insert: 13



Solution:




10
5
14
3
7
12
18
13
16
20

8. Remove Many [6 points]

You have an array-based implementation of a 2-heap, implementing a minimum priority queue. Suppose that you want all of the k smallest elements of the heap, where k is a variable, but k is much smaller than n.

(a) Your first idea is to scan through the array based implementation from index 1 (where the root is stored) to index x, where x is the last location where the k^{th} smallest element could be. What value of x should you choose? Briefly (1-2 sentences) explain why you chose your value of x. (4 points) **Solution:**

Heaps keep the smallest element at the top, but they're not fully sorted! The k smallest elements could all be in a "line" with each other with larger elements around them. So you have to check through the end of level $k-1$ level of the tree. We therefore need to check that $\sum_{i=0}^{k-1} 2^i = 2^k - 1$.

(b) Your second idea is to call `removeMin` k times, then re-`insert` those k elements. What is the worst-case big-$\mathcal{O}$ running time of this version? No justification required. (2 points) **Solution:**

$O(k \log(n))$. Both calls are $\Theta(\log n)$ in the worst-case. Since k is much less than n, repeating the calls won't make the heap small enough to make a difference in the efficiency.

9. Priority AVLs [12 points]

Recall that priority queues had three operations: `insert`, `removeMin`, and `peekMin`. Having just finished your implementation of an AVL tree (and not your implementation of a heap), you decide to see if you can skip heaps entirely and just use your AVL tree code for a Priority Queue as well.

(a) Give a brief (1-2 sentence) description of pseudocode for a method `getMinNode`, this will be a private helper method that finds the node containing the smallest priority in the tree. (2 points) **Solution:**

Starting from the root, while current node has a left child, go left. When the current node no longer has a left child, return the current node.

(b) What is the worst-case running time of `getMinNode` in a tree with n elements? Give a 1 sentence justification. (2 points) **Solution:**

$O(\log(n))$. The height of an AVL tree is always $O(\log n)$.

(c) Describe how to implement `insert`. Your description can simply describe what alterations (if any) are needed from the standard AVL insertion. (2 points) **Solution:**

We would give full credit for either "no changes needed" or "after insertion use `getMinNode` to update minimum field" (the answer here depends on whether you've figured out part d already, so we'd accept the answer if you hadn't gotten there yet)

(d) Describe how to implement `peekMin` in $O(1)$ time. Be careful! You can't run `getMinNode` in this method, so you'll have to use it in (two) other places... (2 points) **Solution:**

Add a field `currMinVal` and update it by calling `getMinNode` after insertions and deletions. `peekMin` then simply returns that field value.

(e) You've (at least) matched the big-O running times for heap-based implementations of priority queues. Give at least two reasons why one might prefer the heap implementation discussed in class instead of AVL trees. (4 points) **Solution:**

- Better constant factors (no pointers)
- Better memory footprint (less wasted space with fewer pointers)
- Better cache behavior *possibly* (heaps have better spatial locality, but for sufficiently tall heaps, that may not help much since we bounce very far away in the array structure).
- Easier to implement.
- There might be others we haven't thought of!

Extra piece of paper for scratch work

Reference Sheet

Geometric series identities

$$\sum_{i=0}^{k} c^i = \frac{c^{k+1}-1}{c-1} \qquad \sum_{i=0}^{\infty} c^i = \frac{1}{1-c} \text{ if } |c| < 1$$

Sums of polynomials

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \qquad \sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \qquad \sum_{i=0}^{n} i^3 = \frac{n^2(n+1)^2}{4}$$

Log identities

$$b^{\log_b(a)} = a \qquad \log_b(x^y) = y \cdot \log_b(x) \qquad a^{\log_b(c)} = c^{\log_b(a)} \qquad \log_b(a) = \frac{\log_d(a)}{\log_d(b)}$$