

CptS 223 - Spring 2020

Take-home Final Exam

Your Name: _Ryan Rochleau_____

ID Number: __011577056_____

READ THE FOLLOWING INSTRUCTIONS:

1. When your finished, create a branch called FinalExam. Add, commit, and push to remote origin on GitLab.
2. You may use your book and online resources. However, you may not work with others.

	Points Possible	Points Earned
Total	120	

~~ General Stuff ~~

1. [2] Assume that we have two functions: $f(n)$ and $g(n)$. On a benchmark where $n = 10,000$, the execution time of $f(n)$ was 15 seconds and the execution time of $g(n)$ was 2.3 seconds. What can we conclude about these two functions?

Answer: That $g(n)$ bounds $f(n)$

2. [2] If $f(n)$ took 30 seconds for an $n = 100$ then how long would it take to process $n = 400$ if $f(n)$ was of order:

1. $O(N)$

Answer: $400/100 = 4$

$$4 * 30 = 120 \text{ seconds}$$

2. $O(N^2)$

Answer: $400^2/100^2 = 16$

$$16 * 30 = 480 \text{ seconds}$$

3. [4] Big-O Dating Game. Match each data structure with the correct desired behavior so that each algorithm achieves its best Big-O (runtime efficiency). Feel free to justify each match.

Structure A (2): I'm the kind of algorithm who wants a data structure that isn't slow minded. When I ask it a question, I expect my data structure to be able to find the answer quickly and efficiently, but I don't need to keep my data in any particular order. (What data structure would be appropriate in a find-heavy algorithm?)

Answer: The appropriate data structure would be an unordered hash map as it doesn't have any particular order but finding values is $O(1)$.

Structure B (2): I like to keep things tidy. I like it best when everything is in its rightful place and would like to meet a data structure that isn't going to make that difficult. (What data structure would be appropriate for an algorithm that needs to maintain the ordering of my data?)

Answer: AVL tree would be appropriate as the rules required for constructing one create order and rotations keep the tree tidy and removes the potential untidiness that can arise from a standard BST with a bad root value.

4. [5] Supply both the names of these interfaces and code to complete the Big Five in this partial class implementation. Note helper functions at the end:

```
class AvlTree {
private:
    AvlNode *root;

public:
    ~AvlTree( ) {                                     // ← Interface name: __Destructor_____

        makeEmpty(root);

    }

    AvlTree( const AvlTree &other ) : root( NULL ) // ← Interface name: __Copy Constructor_____
    {
        root = clone(other.root);

    }

    AvlTree( AvlTree &&other ) : root( NULL )      // ← Interface name: __Move constructor_____
    {
        root = other.root;

        Other.root = nullptr;

    }

    const AvlTree & operator= ( const AvlTree & other ) // ← Interface name: __Copy Assignment operator_____
    {
        AvlTree temp(other);

        Swap(root,temp.root);

        return *this;
    }

    const AvlTree & operator= ( AvlTree && other )      // ← Interface name: __Move assignment operator_____
    {
        if(this!=&other)
        {
            Root = nullptr;

            Root = other.root;

            Other.root = nullptr;

        }
    }
}
```

```

return *this;          // Return ourselves to caller
}
AvlNode * clone( AvlNode *t ) {          // Clone the whole tree
    if( t == NULL ) { return NULL; }
    else { return new AvlNode( t->element, clone( t->left ), clone( t->right ), t->height ); }
}
void makeEmpty( AvlNode * &t ) {          // Delete the whole tree
    if( t != NULL ) { makeEmpty( t->left ); makeEmpty( t->right ); delete t; }
    t = NULL; }
};

```

~~ Hashing ~~

5. [10] Suppose we want to insert the keys (k) 7, 10, 17, 20, 27 (in this order) into an initially empty hash table of size $TS = 7$ and using the primary hash function $h(k) = k \% TS$. Show the final hash table for each of the following scenarios.

a. (4) Collision resolution by separate chaining.

0	1	2	3	4	5	6
7			10			20
			17			27

b. (4) Collision resolution with a linear probing hash function for i in range $(0..TS-1)$:

$$\text{probe}(i) = (\text{hash}(k) + (i + 1)) \% TS$$

0	1	2	3	4	5	6
7	27		10	17		20

c. (2) What is lazy deletion and why is it important for linear and quadratic probing?

Answer: Let's say we hard deleted 7 in part b and now we need to find 27. Without having to go through the whole table, we would start at $\text{hash}(27)$ and linearly or quadratically proceed through the table. If we hard

deleted 7, we might stop where 7 was deleted as it's empty and say that 27 isn't in the table while it is actually one index ahead. If we lazy delete, we have knowledge there was once a value there and 27 might still be ahead so we can continue searching until we find 27 or not. In the case of hard deletion, if we ignore empty indexes and just continue searching, we might go through the entire table even if 27 isn't in it and sacrifice efficiency.

6. [4] Which of the following hashing algorithms is likely to produce a faster HashTable implementation? Why?

Hashing Algorithm A	Hashing Algorithm B
<pre>int hash(string text): int result = 0; for(char ch : text) result += ch; return result;</pre>	<pre>int hash(string text): int result = 1; for(char ch : text) result *= ch * ch + (11 * ch) + 7; return result;</pre>

Answer: Neither algorithms take into account the position of the character meaning that anagrams would hash to the same location in both algorithms. Thus, computation of the result is the only major differences between the two algorithms meaning Algorithm A would likely be faster as it takes less processing power to compute the result. However, Algorithm B isn't very hard to compute so the difference is very small.

~~ Heaps ~~

7. [6] Binary Heaps (Priority Queues!). Enqueue (insert) the value 6 to the following binary heap, then do a Dequeue (deleteMin):

Starting heap:

3	7	5	9	7	12	8	14		
---	---	---	---	---	----	---	----	--	--

After enqueue (insert):

3	6	5	7	7	12	8	14	9	
---	---	---	---	---	----	---	----	---	--

After dequeue (deleteMin):

5	6	8	7	7	12	9	14		
---	---	---	---	---	----	---	----	--	--

I suggest you draw it out:

8. [4] List the fundamental structural rules of a binary heap (priority queue) and also let me know what the average height of a binary heap is:

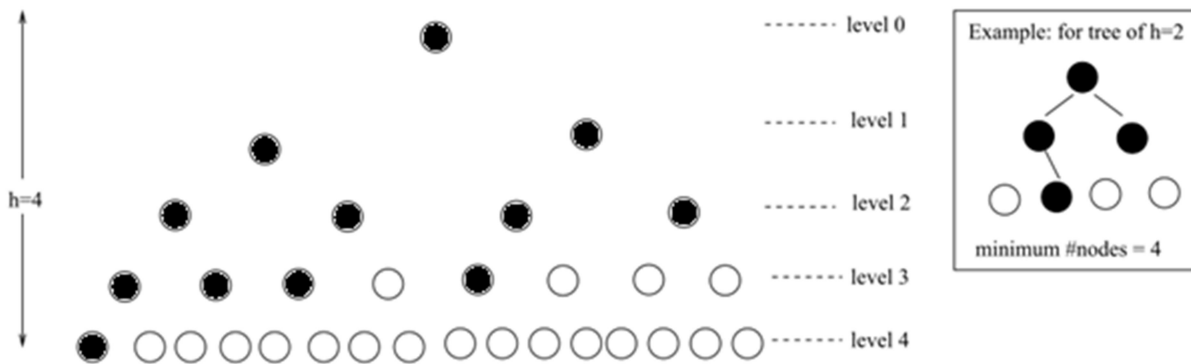
Answer: Binary heaps must be complete trees where each level must be completely filled except for the final level in which all values must fill in from left to right. They are also either min heaps or max heaps where min heaps have every child be a greater value than their parents and max heaps have every child be a smaller value than their parent. Height of a binary heap is $\log(n)$ where n is the number of nodes.

~~ Trees ~~

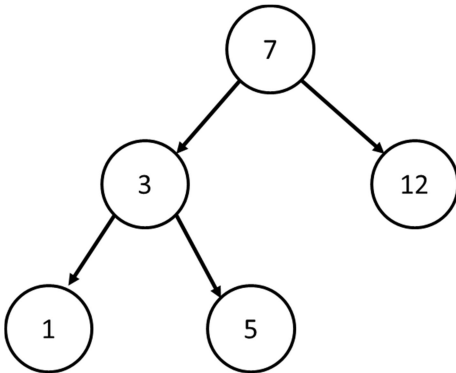
9. [2] For Depth-first (pre, post, & in-order) tree traversals, it is often convenient to use a BST data structure to help process the tree nodes.

10. [2] For Breadth-first (level order) tree traversals, it is often convenient to use a Heap data structure to help process the tree nodes.

11. [2] What are the minimum number of nodes allowed in an AVL tree of height 4? Give your answer both as a number and by shading in the necessary nodes below. Provided is a possible answer to the same question with an AVL tree of height 2. **Answer: 12 nodes**



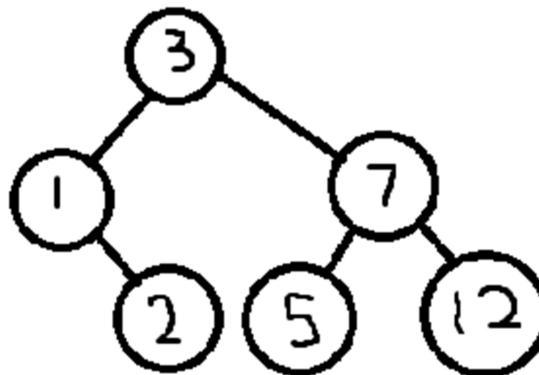
12. [7] Assuming we will be inserting 2 into the following AVL tree:



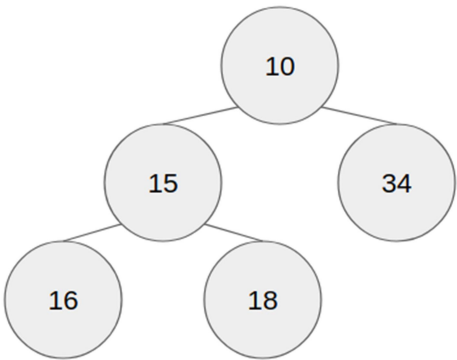
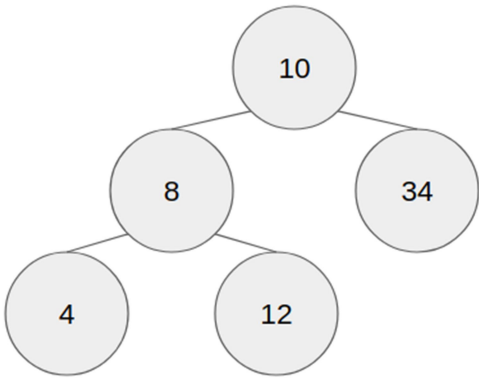
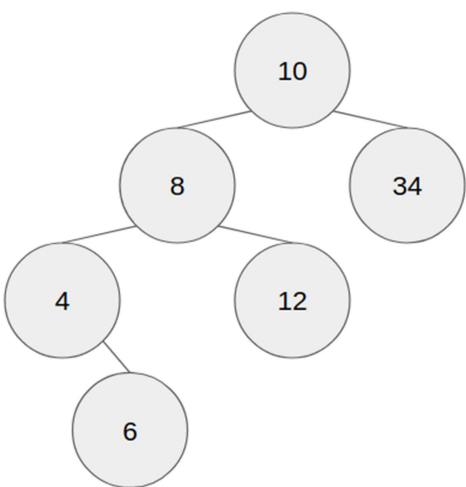
a. (2) What kind of rotation will be performed?

Answer: Right Rotation

b. (5) Draw the resulting tree:

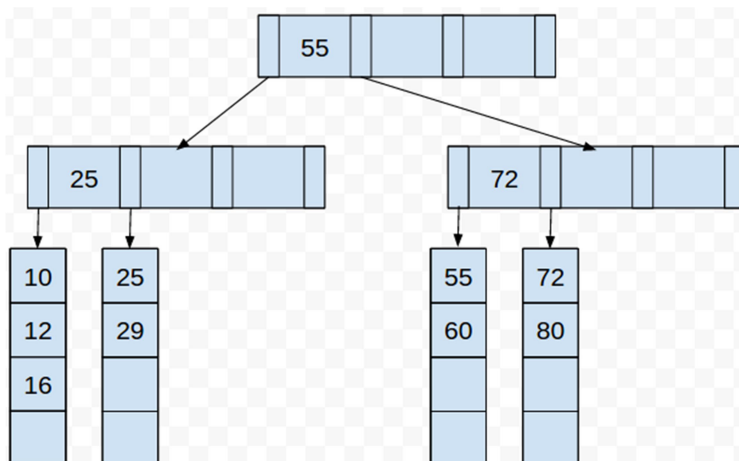


13. [9] Circle whether each tree is a valid Binary Search Tree (BST), AVL tree and/or Binary Heap

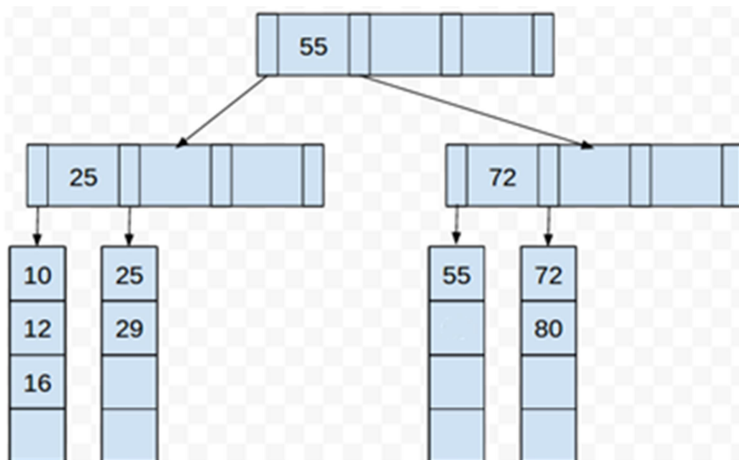
Tree	Valid BST?	Valid AVL?	Valid Heap?
 <pre> graph TD 10((10)) --> 15((15)) 10 --> 34((34)) 15 --> 16((16)) 15 --> 18((18)) </pre>	Yes <input checked="" type="radio"/> No	Yes <input checked="" type="radio"/> No	<input checked="" type="radio"/> Yes No
 <pre> graph TD 10((10)) --> 8((8)) 10 --> 34((34)) 8 --> 4((4)) 8 --> 12((12)) </pre>	Yes <input checked="" type="radio"/> No	Yes <input checked="" type="radio"/> No	Yes <input checked="" type="radio"/> No
 <pre> graph TD 10((10)) --> 8((8)) 10 --> 34((34)) 8 --> 4((4)) 8 --> 12((12)) 4 --> 6((6)) </pre>	Yes <input checked="" type="radio"/> No	Yes <input checked="" type="radio"/> No	Yes <input checked="" type="radio"/> No

14. [4] The following B+ tree has $M = 3$, $L = 4$.

Remove the record with the key 60 from the tree and draw the results.



Result



15. [2] What is the primary reason to use a B+ Tree?

Answer: To limit the number of disk accesses that need to be made. B+ trees help with height reduction as a smaller height allows for less disk seeks.

16. [2] List the four fundamental structural rules of a red-black tree:

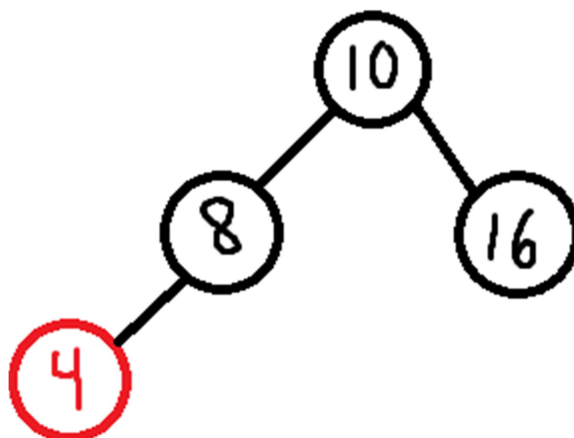
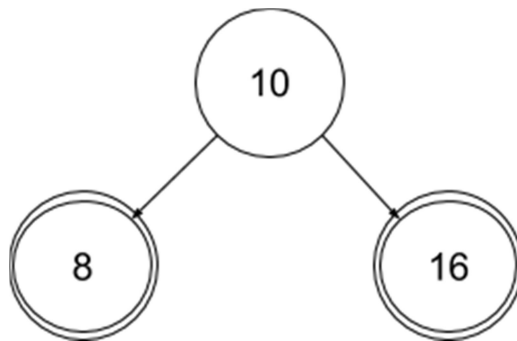
1: Every node is Red or Black

2: The root node is Black

3: Red nodes have no Red children or Red parents

4: Every path from any node to any NULL descendent has the same number of black nodes.

17. [4] Please draw the tree as it should be when we're done with an insert(4) operation:



~~ Parallel Programming ~~

18. [4] What is parallel programming? Explain.

Answer: Parallel programming is when many multiple computations are being performed at the same time or “parallel” to one another. Typically, larger programs can be broken down into smaller tasks which can be performed at the same time as other tasks to reduce runtime and increase time-efficiency.

19. [4] When should we use critical regions in OpenMP? Explain.

Answer: Critical regions means only one thread can be responsible for the execution of the critical region. This is to prevent multiple threads from executing the same piece of code at one time.

20. [4] What is meant by synchronizing tasks? Explain.

Answer: Imposing constraints on the code so that shared data is protected through various methods which include critical, atomic, locks, etc. Locks for example can prevent threads from proceeding until the lock is acquired. In order for a lock to be acquired, it needs to be available and if it isn't, another thread needs to release the lock when it is finished thus preventing threads from interacting with shared data until the other thread is complete.

~~ Sorting ~~

21. [6] For each of the following sorting algorithms, give the best-case and worst-case running times for sorting an array of size N and note the stability of the sort.

Algorithm	Best Case	Worst Case	Is Sort Stable?
InsertionSort	$O(n)$	$O(n^2)$	Yes
HeapSort	$O(n \log(n))$	$O(n \log(n))$	No
QuickSort	$O(n \log(n))$	$O(n^2)$	No
Radix Sort	$O(n * k) \approx O(n)$	$O(n * k) \approx O(n)$	Yes

(k is the number of bits required to store each key but it never gets very large so $O(n)$ is a good approximation)

22. [8] Perform quick sort on the following numbers: [8, 3, 9, 6, 2, 5, 4, 1].

Pick pivot 6 and move it to the end

[8, 3, 9, 1, 2, 5, 4, 6]

Set right pointer at 4 and left at 8. Increase the left and decrease the right pointer until values that are greater and less than 6 respectively are found then swap

[4, 3, 9, 1, 2, 5, 8, 6] (8 is greater than 6 and 4 is less than 6 so swap.

Move pointers and repeat

[4, 3, 5, 1, 2, 9, 8, 6]

No more swaps can be made as pointers collided so swap pivot with right pointer

[4, 3, 5, 1, 2, 6, 8, 9]

Repeat these steps on the left subarray [4, 3, 5, 1, 2] with pivot 3

Swap 3 to end

[4, 2, 5, 1, 3]

4 greater than 3 and 1 less so swap

[1, 2, 5, 4, 3]

Pointers collide so swap 3 with right pointer

[1, 2, 3, 4, 5]

Repeat steps on left subarray [1, 2]

Pick pivot 1 and swap to end

[2, 1]

Pointers collide on 2 so swap 1 with right pointer

[1,2]

Array [2] is already sorted

Pick pivot of 4 in right subarray [4,5]

Swap to end

[5,4]

Pointer collide so swap 4 with right pointer

[4,5]

Array [5] is already sorted

Pick pivot 8 in final right subarray [8,9]

Swap to end

[9,8]

Pointers collide so swap 8 with right pointer

[8,9]

Array [9] is already sorted

Completed sort and ended with [1,2,3,4,5,6,8,9]

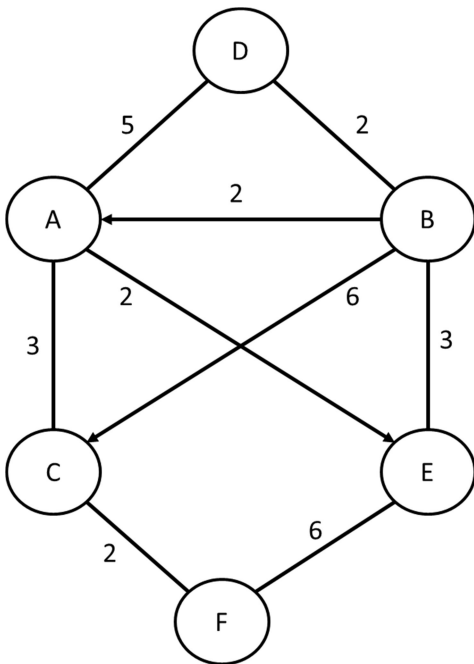
~~ Graphs ~~

23. [1] What are the requirements for a graph to be considered a DAG?

Answer: Nodes are connected to other nodes via a one directional arrow in such a way that there is no way to start at a node X and return to node X by following the arrows.

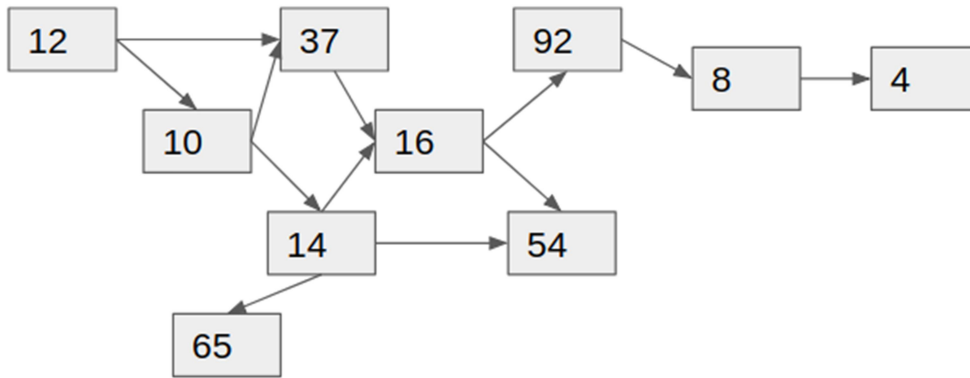
24. [8] Dijkstra's Algorithm. Use Dijkstra's Algorithm to determine the shortest path starting at E. Note that edges without heads are bi-directional. To save time, you do not have to add items to the "priority queue" column after it has been discovered (listed in the "distance" column). Use the table below to show your work.

Not sure I get how to fill the table out. Priority Queue column is how it behaves throughout the algorithm.
Distance Column is the final distances



Node: Distance	Priority Queue
E: 0	E B F
B: 3	B F
F: 6	B F D A C
D: 5	F D A C
A: 5	F A C
C: 8	F C
	C

25. [2] Emit a valid vector of results from topological sort when run on this graph:



[12,10,37,14,65,16,54,92,8,4]

26. [6] Time for some graph definitions in short answer from. What are:

Directed graph: __Vertices connected by edges where each edge has a direction_____

Undirected graph: __Vertices connected by edges where each edge is bidirectional_____

Adjacency: __Vertices are adjacent if there is an edge between them_____

Path: __A sequence of edges that join vertices_____

Cycle: __A path in which the first and last vertices are the same_____

Connected Graph: __A graph in which there is a path between every pair of vertices_____

27. [2] In your own words, describe the purpose behind doing and using Big-O analysis:

Answer: Big O analysis is important for comparing algorithms against each other. It acts sort of as a common language between data structure so that each can be compared to one another in a meaningful way. It is also important to know the Big O of our data structures as efficiency is vital in computer science and using inefficient data structures sacrifices efficiency

28. [2] What is Git and name 3 key things (not commands, but kinds of activities) it enables developers to do:

Answer: Git is commonly known as a version control system meaning it enables developers to control changes in code in an organized manner. Developers can branch off of the master branch to make changes that won't affect the current version on master. They can then merge these changes into the master branch for others to see and use. Git is very useful for large teams as it allows for many individuals to work on separate features and then merge them in later without initially affected the master version.

29. [2] What is your favorite data structure or algorithm? Why?

Answer: Whilst I don't have a perfect understanding of everything about them, I would have to say hash tables for two reasons. The first is that hashing algorithms involve a lot of advanced math which has always interested me and exploring how we can create new hashes that don't collide sounds like a great challenge. Second is how applicable they are to almost every problem. Many coding interviews can be solved using hash tables as their instant lookup proves to be a very useful tool in almost all problems I've run into.