CSE 2050 – Exam 3 Review

Hey y'all, it's Akeva! I'm going to be going over the topics applicable for Exam 3 which may also encompass past topics such as recursion. This study guide will be focused on Divide-and-Conquer Sorting Algorithms, Mappings and Hashing, as well as Trees. Let's get into the topics at hand!

As a quick note for my guide styling, underlined words are general headings and words that are italicized are subheadings.

<u>Divide-and-Conquer Algorithms</u>

The idea behind divide-and-Conquer Algorithms are that you take a complex problem and split them up into smaller sub-problems, solve them with recursion or even iteratively, and then combine the result. These algorithms are O(nlogn). The two applicable sorting algorithms that do this are Mergesort and Quicksort. Let's talk about them more in depth.

Mergesort

Based on the name, you can infer that merging is a huge part of the way this algorithm sorts an array. It essentially splits an array into individual subparts (i.e., elements and hence the n aspect to the time complexity) by going down the middle and working on either side (logn component) and then two by two combines the results such that the first element is the smallest value, or largest depending on what is asked. Mergesort often uses a helper function as it makes things much easier. As a rule of thumb, the helper function in the case of mergesort should be the actual merging because the functionality is a bit different from mergesort. As I discussed before, we split the array and then build it back together, so we two separate functions that do just that!

You might be thinking that we may need a base case when dealing with mergesort so that we don't hit any errors with splitting and you would be correct! Before I go out and say it, what should the base case for splitting be (hint hint, I mentioned this when described the splitting)…

It should be that when the length of the array, or when the difference of the pointers for a sub-array equals 1 (this depends on the strategy you ultimately use, either creating subarrays using splicing or pointers). Just a note, you may be asked to do either, however, generally, the pointer method is better because it takes less space on your computer: with splicing, you create a new list each time and then when combining, you make a new list again.

Before writing code, I like to think about what my code needs to do. For the sake of the mergesort and not the helper function, let's think about. I need:

- My base case for one element – probably just going to return the element
- I need to split my array in half
    - How can I do this?

        - Need to find out what the middle is
        - Take everything from the beginning to the midway point for one part and from the midway point + 1 to the end of the array for another part
- Send these sides to be merged

Okay, now what about the helper function? It needs:

- To compare the left and right and put the least or greatest number first and continue
    - How can I do this?
        - Have a pointer in both the left and right arrays and compare the elements
        - Increase the one that had the smallest integer
        - Stop when the pointer extends past the length of the left and right array
- Return the combined lists

But let's think about merge more deeply. What if the list is already sorted? If we just compare, eventually the left or right will be put in the result but there will be missing numbers still not accounted for. Okay, so also need to add the extras.

Now that we have accounted for how the code should work, writing the code has become simpler. If you're anything like me and get confused by when return statements should be used in recursion problems, use the following

- 1000% use it during your base case (the easy part)
- Think about whether the result of a recursive call needs to be shown or will need to be saved. Think about this in the last step, which will be half the original array in left and the other in right.
    - Will I need to return the mergesort of a splitting? No, but I will need it later when I feed it into merge?
    - But I will need to return the result of merge. Therefore, this is where the return function should go

I don't think it would be helpful for me to go through mergesort when there are many good resources, but if you want to see my code mergesort, feel free to come in during my hours! To see examples of mergesort and more information regarding it, look at the following videos:

- Michael Sambol (goes over mergesort and has pseudo code)
- Greg Hogg (no helper function)
    - Note, the helper function would be where he defined little l and r for the pointers in the left and right array
- W3 Schools (for my readers and those who want to see a helper version)

Quicksort

Let's move on to Quicksort! So, with this algorithm, it's often done in-place. Generally, the way it works if to the find a pivot, start at the beginning of the array, and have a second pivot that determine is the element is less than that of the pivot. If it is, the first and second pointers values switch. The first pointer increases by one and the second pointer continues until it reaches the pivot. Then the first pointer value and the pivot value switch. In terms of visuals, I think quicksort demands it a lot. So, look at the animations available on W3 Schools. There are many ways to choose a pivot. However, most of the time, you just use the first or last value as the pivot. Similarly to the merge sort, an efficient way to implement the algorithm is to use a helper function.

In this case, the helper function is used to do the swapping of higher and lower values as that is a repeated function and a distinct part of quicksort. As this algorithm is often done in-place, we have to keep track of a low pointer for where to start looking for lower numbers and high for where the high number ends. It is similar to mergesort in the fact that we have a left and right array, however, it is not necessarily split in half. Doing something similar to what we did for mergesort, let's begin to think about what functionalities belongs to quicksort and the helper function, which I'll refer to as partition.

In quicksort:

- Define what high is if it's the first time, we have passed the array in the function
- Call the partition function if low is less than high (remember low and high are just pointers or the indices)
    - If they are equal, there is no need to further sort
- Run quicksort on the left side of the pivot index based on where it ended up after partitioning and on the right side

In partition:

- Define the pivot (typically the first or last thing in the array/sub-array)
    - For the sake of the argument, let's say the pivot is just the last element in the sub-array
- Define a floating pointer as low, where low = i - 1
- For each number, let's say j, from the range of low to high, if the number at index j is less than the pivot, switch the values of the number at indexes i and j
    - Increase i by 1
- Finally, switch the values of the pivot and i + 1
- Return the new index of the pivot aka i + 1

To put this list of pseudo code together and to rehash what we went over, refer to the following resources:

- Michael Sambol (pseudo code)
- FelixTechTips
- W3 School

Mappings and Hashing

Dictionaries aka Maps in Python

Alrighty, new section! I am sure y'all are familiar with dictionaries or maps, regardless of the language you have used in the past. In Python, they are defined by dict(), {}, or defaultdict(). You may be more familiar with the first two, however, the last one is a module you can import in Python that allows you define what should be in the list and initializes it. For example, defaultdict(list) initializes new keys to an empty list so that you could do defaultdict.append(whatever) without having to input an empty list and then append what you want. You may not be able to use this in an exam, so this is more for your reference. Dictionaries are extremely helpful due to their key-value pairs. Based on the key, you can find associated values, hence the name. Some important notes about dictionaries you should know and may be tested on:

- Keys must be immutable
    - Do you remember what data types are immutable? List them before looking below
        - Integers
        - Strings
        - Frozensets (don't get them confused with regular sets)
        - Tuples
- Keys can't be doubled within a given dictionary aka they are unique
    - If it's called again, the second key-value pair is saved
- Values can be immutable or mutable
- Dictionaries are ordered
- They used hash maps for quick lookups (next sub-section)

Furthermore, below are some important methods of a dictionary:

- .update(key): can change the values associated with the key
- .keys(): can get a list of all the keys
- .values(): returns all the values
- .items(): returns key-value pairs are tuples (particularly helpful for looping through a dictionary)

- del key: removes a key-value pair from a dictionary

I believe that summarizes what you should know about dictionaries and some key applications of them. Now, we are going to move on to hashing which is a huge chunk of what professors like to test on in CSE 2050.

Hashing – What's underlying dictionaries?

What is hashing? Hashing is a technique that transforms any input data into a has value (fixed length of characters) using mathematics. Now, knowing the mathematics is not important. However, it's imperative you know how to find a has value. In python, the function is hash(whatever). I should mention that input values to hash values are not one-to-one (meaning, per input you could ever make up, it has its own unique has value). There are and can be many keys that map to the same hash value. This is why in dictionaries, lookup is nearly O(1) or O(1) amortized. Now, how does this correlate to a dictionary? We can model a dictionary with a list. A given key in the dictionary will be given a fixed index using the following equation:

$$index = hash(key) \ \% \ n_{buckets}$$

Or the hash of the key modulo the number of buckets in the list.

Since I mentioned that hashs are not one-to-one, what happens when we encounter an input that maps to a hash value that we have seen? Well, we call this a hash collision and there are two main ways to deal with them:

- Open Addressing: this means that we need to search for the next available slot or bucket and put the value there
- Separate chaining: this means that we tack on the value to the existing one in a list

Okay, but what happens when we are getting full, or we have a bucket with a long chain? Then we need to resize. There is a number usually associated with triggering resizing and that depends on what is given. Sometimes it's when our makeshift dictionary is two-thirds of the way full; just do what is asked of the problem. So, this means in our class of this makeshift dictionary, we need to keep track of the number of key-value pairs we put in it, define the load factor, and the number of buckets, and determine how much the buckets grow.

For examples of fully implementing this, I would highly suggest looking in your textbook and the slides your professor posted. Furthermore, use the following resources:

- Middle Georgia State University
- w3w3w3 (contains logical some errors)
- howCode

Trees

Okay, last section! Trees are data structures that encompass many underlying parts of our daily lives such as genealogy, workplace hierarchies in terms of who people report to, file systems, and AI systems (such as decision trees or Random Forests). Before we dive into how we can apply them and associated algorithms, let's start with some definitions. I want to note that trees start from the root and branch downward which the definitions are based on.

Definitions

- Node: A point within the tree that has an associated value
- Root: the top or starting node within a tree
- Child: A node that comes or descends from one node
- Parent: the node directly above and attached to the child node
- Ancestor: any node that is above the node in question in the lineage. For example, the parent, the parent of the parent, and so on
    - Think of it like your maternal and paternal grandparents are your ancestors but not your aunt or second cousin.
- Descendant: children of a node and any child after them
- Depth: the level of a node within a tree
    - The depth of a root = 0 and subsequently increases by 1 for each level of the tree. You can think of it as how many connections away the node is from the root
- Subtree: A tree that begins at a defined root and encompasses all descendants of the defined root
- Binary tree: A tree where each node can have a maximum of two children
    - If the tree is defined using an array, you can calculate the indices of the children and parents
        - Children

$$left\ child = 2i + 1$$

$$right\ child = 2i + 2$$

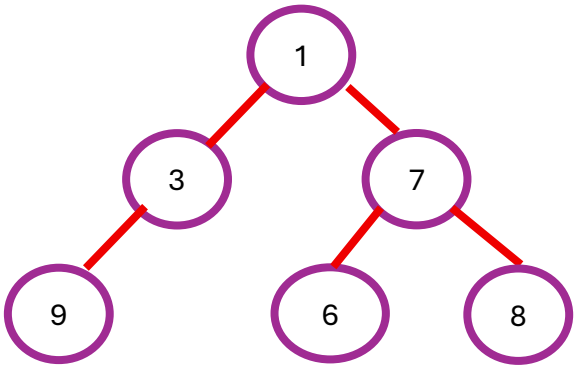
        - Parents (yes, floor division)

$$parent = (i - 1)//2$$

- Internal node: every node that has children
- Leaf node: nodes without any children

- Complete tree: a binary tree where all levels except the last one are completely full (each node has 2 children)
- Perfect tree: a tree where every internal node has two children and all leaf nodes are at the same depth

Ordering of a Tree



If we want to go through the whole tree, there are three different ways to visit the nodes and print out their ordering (try going through the above tree in each order and see if you get it right):

1. Pre-order: you visit the root node first then the left subtree and then the right subtree
    a. 1, 3, 9, 7, 6, 8
2. In-order: you visit the left first, then the root, and then the right
    a. 9, 3, 1, 6, 7, 8
3. Post-order: you visit the left, the right, and then the root
    a. 9, 3, 6, 8, 7, 1

I should note that when I write visit, it's more of acknowledging the node or printing the value because of course you have to visit the node to see if it has a left or right. Try implementing pre-order, in-order, and post-order iteratively and then recursively. For extra resources, look at the following:

- Recursion
    - W3 Schools
- Iteration
    - GeeksforGeeks

Now that we know how we can generally go through a list, let's dive into how we can traverse through a tree.

Tree Traversal (DFS, BFS)

Let's start with some definitions. What does DFS and BFS mean?

- Depth-first search (DFS): You go through a particular branch first before explore going through another branch
- Breadth-first search (BFS): You go through all the nodes in a level first before going to another level

With regards to implementing these algorithms, just generally know that we will have to use lists or Doubly-Linked lists and be able to implement both ways so that you are fully prepared for the exam. My general rule of thumb for thinking about DFS vs BFS is that in DFS, we care about last in first out (LIFO) and therefore can easily implement this in O(n) time with a list. With BFS, however, we need to visit each level, so we use first in first out (FIFO), which, as you may recall, works better with queues which is best done in a Doubly-Linked List (particularly deque in Python).

If you need a refresher on implementing LinkedLists, use the following resources:

- GeeksforGeeks
- Programming and Math Tutorials
- Greg Hogg

For sake of example before I give y'all other resources, I am going to use the above example tree to show what should be going into and out of the list in a DFS and BFS.

For DFS (note, it could also start with the left branch first and still be viable solution):

[]

[1]

[3, 7]

[3, 6, 8]

[3, 6]

[3]

[9]

[]

Your code should take root node, pop it from the list, and process any children it may have and add them to the list. Then you pop the last element and process any children it may have and so on until the list in empty. Each new line is the processing the pop and adding any new children.

For BFS:

[]

[1]

[3, 7]

[7, 9]

[9, 6, 8]

[6, 8]

[8]

[]

This code should take in your root node, pop from the front of the queue, and add all the children to the end of the queue. In the next pass of the loop, it will pop from the front again and continue until the list is empty. If you are allowed to use deque on your exam and not sure how to use it, refer to this resource by GeeksforGeeks.

As I have been hinting at in this section, a while loop with be your friend in implementing these algorithms conditioned on the emptiness of the list or queue (aka while not list or while not queue). This could also be done recursively. For the sake of the exam, know both ways to complete DFS and BFS. For some resources to help put these concepts together and perform DFS and BFS, use the following:

- Greg Hogg (BFS)
- Michael Sambol (BFS w/ pseudo code)
- Art of CSE (BFS w/recursion)
    - Don't worry about the fact that it's a graph. Trees are a type of graph
    - See if you can modify the code for a tree application
- howCode (DFS w/ iteration and recursion)

Binary Search Tree (BST)

As an extension of binary search for an array, we can use the same logic for trees. Like binary search, you cannot use it on any regular tree. It has to be a binary tree where all the values in the left subtree are less than the current node's value and the right have nodes that are all greater than the current node. Based on this, is the above binary tree a binary search tree?

…No, it isn't. However, the left subtree consisting of the node with 7 and its children would be the overall form.

The primary use for binary search tree is the quick look up as to whether or not a number exists in a tree. In a tree that isn't a BST, you would have to perform DFS or BFS to see if the desired value is in the tree, which is O(n) in the worst case scenario because you have to go through the

whole tree to establish that node was the last one checked or not in the tree at all. However, in BSTs, you can search in log(n). We just use the rules I established earlier:

- Left children are less than the parent value
- Right children are greater the parent value

You can imagine though that if we add a node to a or remove a node from a tree, we may have to switch somethings around to make the tree at BST again. This is where functions such as bubble up or bubble down appears where we swap values until all values are located exactly where they need to be. Before moving, we would have to check the parent in terms of bubbling up which can be done by using the equations I mentioned in the definition's subsection of the Trees section. Similarly, we need to check the children when bubbling down, which can be done by accessing .left or .right or indices for each using the formula I have listed above.

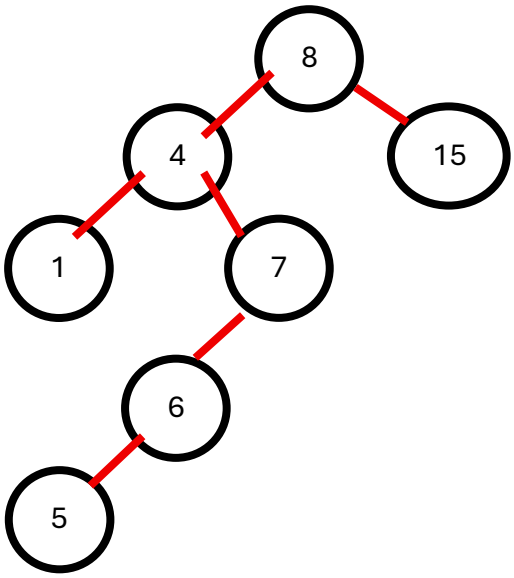
Try implementing the search, delete, and insertion function on your own. If you need more guidance, refer to the following resources:

- GeeksforGeeks
- NeuralNine

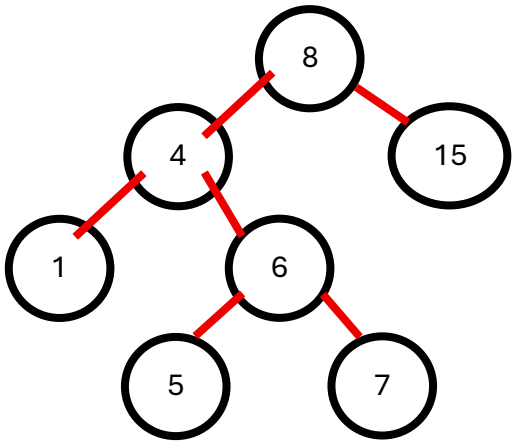
Balancing Trees aka AVL Trees

Okay, let's move to my favorite (😐) topic. Considering how this was one of the harder topics for me to grasp, I will try my best to flesh out everything and provide y'all with ample resources that helped me with learning balancing.

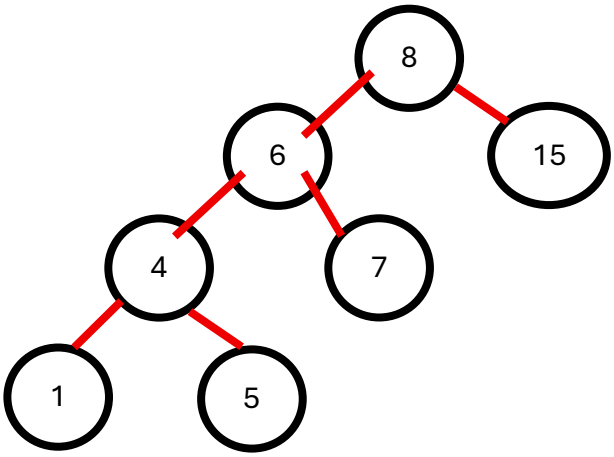
Let's start with the general concept of balancing. Why should be balance? It can greatly help with when we are searching for something and not have us go through all the nodes just to find the value. If we had a BST that was just a long chain, that would defeat the purpose searching for things in O(logn) time. So, when one side is significantly heavier than another, we balance the tree by performing rotations. The way I like to think about how these rotations come into play is that when the left is heavier, we shift the root node (or the base node in question) to the right. Conversely, when the right is heavier, you rotate it left. This would not interfere with the rules of a BST because by definition, if the rotation is on the left subtree, everything is less than the root node anyways and the same can be said for the right subtree but with everything being greater. There are instances where we will have to do two or even more rotation. An example of this is the following tree:



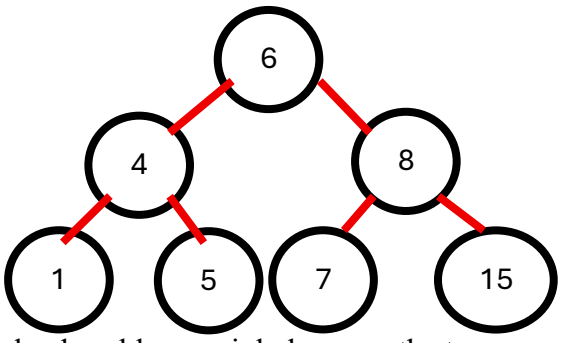
Why? There is a chain of 3 at the bottom (7, 6, and 5), which can be rotated to the right so that the parent attached to 4 is not 7 but rather 6. This change is reflected below:



Sweet, that looks better. But see how there are 5 elements on the left side of the root and only 1 on the right side? So, there's so more work to be done. Let's rotate left at 4. You should get the following:



Note that when we rotate a node with two children, we transfer the left child to the right of the of the node being rotated down. Now that we can rotate 6 which will even out the tree, making us have 3 descendants on the left and right sides. Try it and check it with what I get below:



This was more of an involved problem mainly because the tree was severely unbalanced, when in these trees, we balance as we go. However, the premise stays the same with checking if the right and left sides are balanced in a tree with checking the weights or heights. If there is a difference larger than 1, usually you attempt to balance the tree and rotate it in the direction of the lighter side. With this logic, let's try writing some code workflow for insertion:

class Node:

- Will contain the key, left, right, and height for each node

def insert(node, key):

- Check node aka root exists; if not return the node class with the key
- Check the value and compare it to the left and right
- Update the height
    - Node.height = 1 + max(node.left.height, node.right.height)
- Get the balance calling get_balance()
- If the balance is greater than 1 and the value of node we are inserting is less than the left child of the current node, rotate right
    - The balance tells us that the balance left is heavier and we had to put the key on the left, so we need to rotate right
- If the balance is less than negative one (aka more negative than -1) and the key of the node we put in is greater than the right child of the current node, we need to rotate left
    - This suggests that the right is heavier and be needs to rotate it left to make it weighs more balanced
- If the balance is greater than 1 but the key is greater the left child of the current not, we made the left middle heavier so to account for that, we need to rotate the left child to the left and then rotate everything to the right

- If the balance is more negative than -1 and the key is less than the right child's key (meaning we put the new node in the inner side of the right's child), then we need to rotate the right node to the right and then rotate the node left
- Return node

For the rotations if it makes you think of it better, the balance, when positive signals left, and the key value tells you right or left. If you don't get a double left or double right, you do everything in order of the info presented: balance tells us left and values tells us right so, we rotate left and then right.

Def rotate left:

- right_ = x.right
- outer= y.left
- right_.left = x
- X.right = outer
- Update heights
- Return y

This is exactly what we did in moving from the second diagram to the third diagram in the rotation I did earlier in this section. The same logic would be defined for rotate right. Try it and the flesh out the full code for balancing on your own. Check your responses with the following: Geeks4Geeks.

For extra explanation on balancing, refer to the following resources:

- Michael Sambol (pseudo code)
  - Deletions
  - Insertions
  - Search
- Abdul Bari (insertion and rotations walkthrough)
- Programiz (code)

Heaps

There are two types of heaps: the min heap and max heap. The min heap is a special type of a binary tree where the parent is smaller than or equal to their children. The max heap is where the parent is larger than or equal to the children. It's O(n) space in general because we keep the node values in a list. The key functions you will need to know are:

- Insert: insert a new node
    - The function needs to check that the child nodes are not smaller than the new node
    - May need to swap nodes until the heap is properly satisfied
    - Time is O(logn) since we only need to go through the levels and swap as down as necessary
- Delete: delete a new node and fix the heap as needed
    - Typically, the deleted node is swapped downwards until it is a leaf and then subsequently deleted from the array
    - Time is also O(logn) for the same reason as insert
- Peek: checks the top of the peak
    - O(1) time due to heaps being in an array
- Heapify: turns a list into a min heap
    - Need to determine the smallest or largest out of parent and children pairs (depending on the heap we are dealing with)
    - Time is O(n) because you will need to go through all the whole list
    - Space is O(1) since this is done in-place
- Heapsort: turns an unsorted array into a max heap and then removes the largest element and then does heapify again until the list is sorted
    - Time is O(nlogn) because you must build the heap and then extract the elements
    - Space is O(1) because this is done in-place by keeping track of the array that is sorted already

For a further explanation of heaps, refer to the following resources:

- Min Heap
    - GeeksforGeeks
- Max Heap:
    - Programming and Math Tutorials
    - GeeksforGeeks
- Heapsort
    - Michael Sambol (pseudo-code)
    - CodeSavant

Okay, that's all I have for y'all. Hopefully this is a helpful refresher and answers some of the questions you may have. Good luck on your exam!