

# ECS 32B: Conceptual Homework #3

Instructor: Aaron Kaloti

Summer Session #2 2020

## 1 Changelog

You should always refer to the latest version of this document.

- v.1: Initial version.
- v.2: For the first problem, you should specify the height and balance factor of each node in the final tree of each subproblem.
- v.3: Clarified about duplicates (no duplicate keys allowed) for problem #3.
- v.4: Fixed initial tree in problem #1. (Or, rather, unbroke it, since I think it was fine at least in the initial version but somehow got messed up along the way.)

## 2 Grading

- **Due date: 5:00 PM on Wednesday, 09/09. There is no “grace period.”**
- A subset of the problems will be graded for correctness. The rest will be graded on completion.

## 3 Submission Requirements

- Your submission must be typed with LaTeX. **Handwritten/scanned solutions, or solutions not typed with LaTeX, will earn no credit.**
  - You should avoid doing this assignment at the last minute, because it will take a bit of time to get used to LaTeX and to type your answers up into a LaTeX document.
- Your submission must consist of one file, the PDF generated by your LaTeX file (`hw3_answers.pdf`). Since Gradescope does not let you submit files that are neither PDFs nor images, I will not have you submit the `.tex` file. However, if it is clear that your submission was not made with LaTeX, we may email you to ask that you send us your `.tex` file, and if you cannot (i.e. if it turns out that your PDF was in fact not created with LaTeX), you will get a zero on this assignment.
- You may be penalized if, when submitting on Gradescope, you mark the wrong page for a given homework problem, because dealing with this slows down grading. In the middle of the 08/06 lecture, I talk about how to mark the pages of your submission on Gradescope, starting at around 1:03:55 in the video.
- When using LaTeX, you should make use of the math notation/mode where sensible. (The math mode refers to when you put mathematical equations, etc. between dollar signs so that LaTeX makes them look nice.) Repeated failures to do this, to the point that the purpose of using LaTeX is defeated or the readability of your answers is impeded, may result in a penalty on this assignment.

## 4 Regarding Collaboration

- You may not copy answers from any sources, including online sources such as Chegg, StackOverflow, or any solutions manual of any textbook.
- You may partner with **at most one** other student on this assignment. In other words, you can work in *pairs*. You do not have to partner with anyone. (In fact, I think it is better that you do not.) If you partner with someone, it must be a committed partnership; that is, you two will have the same submission, and you must mark on Gradescope that you have partnered for this assignment by following the directions [here](#).

---

\*This content is protected and may not be shared, uploaded, or distributed.

- If students that were not in the same pair seem to have excessively similar answers, they will be reported to the OSSJA for suspicion of academic misconduct. Do not copy answers from (or share answers with) any student who you are not partnered with.

## 5 Identification

Enter the members of your pair. (You can partner with at most one other student.) If you are not partnered with anyone, then leave the second box empty. You can remove the use of `\vspace` in the `.tex` file.

Pair member #1:

Pair member #2:

## 6 Problems

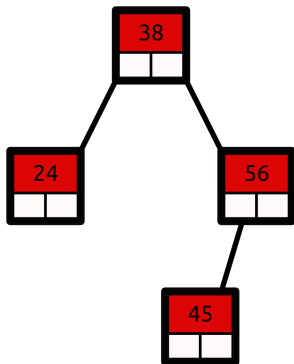
For the problems in which you have to draw something, you can insert an image of your drawing with `\includegraphics`; see more details in the “AVL Tree Insertion” problem section.

Whenever I say that you must draw a tree, you don’t have to handwrite it, i.e. you could use some software for drawing if you want. However, you must actually draw a tree; don’t alternatively provide some list of the nodes and edges, or you won’t get credit.

### 6.1 AVL Tree Insertion

#### 6.1.1 Insert 40

Show the AVL tree that results from inserting 40 into the below AVL tree. If you wish, you can draw the tree by hand and, after uploading an image of, include said image with `\includegraphics`; see how it’s used in the LaTeX file to include the image of the original tree, and note that the file extension must be omitted.) **You must specify the height and balance factor of each node in the final tree.**

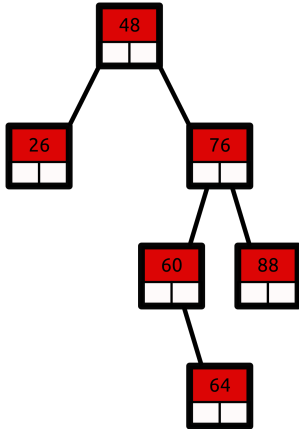


#### 6.1.2 Insert 64

Show the AVL tree that results from inserting 64 into the AVL tree that you have as your answer above. **You must specify the height and balance factor of each node in the final tree.**

## 6.2 AVL Trees: Professor Mean Dude

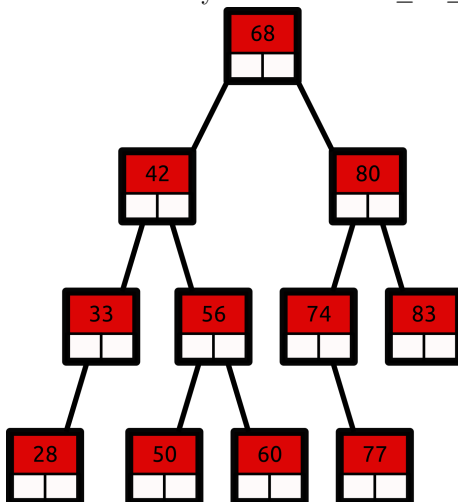
Professor Mean Dude is currently teaching a Python data structures course this summer and is trying to create a very hard final exam. One of the questions that he is working on is meant to test students' knowledge of AVL tree insertion and rebalancing. This question (the one that he created) asks what should happen if the key 80 is inserted into the tree shown below.



Unfortunately, after inserting 80, Professor Mean Dude is not sure how to rebalance the tree. Following the rules on slide #9 of slide deck #7, the balance factor of 80 is 0, the balance factor of 88 becomes 1, and the balance factor of 76 becomes 0. Since a node whose balance factor became 0 is encountered, no more updating should occur, as stated on slide #10. However, Professor Mean Dude notices that the tree is still unbalanced, even though he followed the rules on the lecture slides. What did Professor Mean Dude do wrong? Did Professor Mean Dude perform the insertion wrong? Or is there another issue? Explain.

## 6.3 AVL Trees: Range of Bad Insertions

Consider the AVL tree below. Suppose that we are inserting an *integer*  $x$  into the tree. What is the range of values that  $x$  can take on such that inserting  $x$  will result in a rebalancing (i.e. either a single rotation or double rotation) somewhere in the tree? You may assume that  $0 \leq x \leq 100$ , and you may assume that **no duplicate keys are allowed**.



Below is a tip on how to express the range. If you are trying to express that  $x$  is either in between 38 and 52 (inclusive) or in between 67 and 78 (with 78 excluded), then you can express this range like so (see how it's done in the `.tex` file):  
 $[38, 52] \cap [67, 78)$

## 6.4 AVL Trees: Two Deletions

In this problem, you will create an AVL tree of integers and specify an integer  $x$  to delete from your tree, subject to the following conditions:

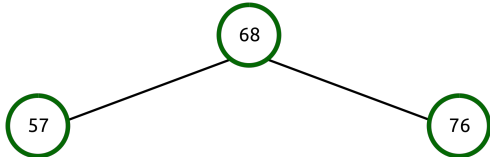
- $x$  is a leaf in the AVL tree.
- Deleting  $x$  results in two rebalances (in the order given):
  - A left single rotation.
  - A right-left double rotation.
- Your AVL tree has as few nodes as is necessary to fulfill the above conditions.

You must provide the following:

- The original AVL tree.
  - The integer to delete.
  - The AVL tree after performing the first rebalance, i.e. the left single rotation.
  - The AVL tree after performing the second rebalance, i.e. the right-left double rotation.
- 

## 6.5 Splay Tree Insert and Find

Consider the splay tree shown below.



For each problem below, a splay tree operation is given. Draw the splay tree that results from doing that operation on the splay tree. The operations are cumulative, e.g. the third operation is performed on the tree that you got from the second operation. As with the earlier AVL tree problem, you may want to use `\includegraphics` to insert an images of your drawings.

### 6.5.1 Insert 40

### 6.5.2 Insert 70

### 6.5.3 Find 68

#### 6.5.4 Insert 45

## 6.6 $d$ -Heaps

A  $d$ -heap is a generalization of a binary heap that is exactly the same as a binary heap, except that each node has  $d$  children. Thus, a binary heap is a 2-heap, and a ternary heap is a 3-heap.

### 6.6.1 One-Based Indexing

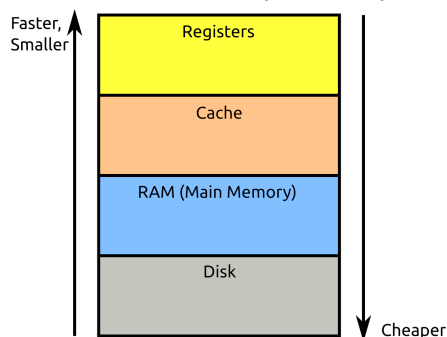
Recall that a binary heap can be represented as a Python list. Given a node at index  $i$  in this Python list, the formulas for finding the index of the parent and left/right children of this node (assuming one-based indexing) were given during lecture. Suppose we are implementing a Python list representation of a ternary heap. Given a node at index  $i$  in this Python list, give the formula for finding the index of the parent, and give the formulas for finding each of the children of this node.

### 6.6.2 Zero-Based Indexing

Repeat the previous problem, except with zero-based indexing instead (i.e. the root of the ternary heap is at index 0 in the Python list).

## 6.7 Caching

Computer architecture, and more specifically computer memory, is a complex and interesting subject in which, as in software-oriented areas like data structures and algorithms, there are many common problems to solve and many tradeoffs to make for speed and other considerations. One such tradeoff is between speed and cost (of making the hardware) and involves what is called the *memory hierarchy*. An abbreviated image of the memory hierarchy is shown below.



Certain properties can make one kind of memory faster (and more expensive) than another kind of memory. Consequently, the registers and cache are the fastest kinds of memory in your computer, whereas the disk is the most plentiful. When a program needs to access a certain piece of data from memory, it will first try to access that data from the cache (pronounced “cash”)<sup>1</sup>. If that fails, then main memory will be checked next, and if that fails, then the disk will be checked. It would be ideal if any data needed by our program happened to be in the cache, but since the cache has a limited size (and since our Python program is not the only software running on our computer), this is impractical. Thus, a cache must be careful about

<sup>1</sup>I don't think explaining registers will make much sense until you've used a compiled programming language such as C or C++

which data is stored. One common rule is to store, in the cache, any data (from main memory or disk) that was recently accessed. The idea behind this is the same as the motivation behind a splay tree: data that was recently accessed is more likely to be accessed again. Cache designers took this idea even further by observing the principle of *spatial locality*: when a program such as a Python program accesses index  $i$  in an array  $A$  (or a “list” or “Python list”, as we call it in Python), where the elements are contiguous, it is likely that indices  $i + 1$ ,  $i + 2$ ,  $i + 3$ , and some more beyond that are likely to be accessed soon (think of a `for` loop iterating through a list like `[5, 8, 14, 2, 9]`; the chance that you access 8 after having accessed 5 is quite high). Thus, in this scenario, a cache that takes advantage of spatial locality would store not only the recently accessed  $A[i]$  in the cache but also store  $A[i + 1]$ ,  $A[i + 2]$ ,  $A[i + 3]$ , and perhaps several more elements along with it<sup>2</sup>, since those elements are right after  $A[i]$  anyways and do not take significant additional time to grab. Importantly, spatial locality can be taken advantage of here *because*  $A[i], A[i + 1], A[i + 2], A[i + 3], \dots$  are contiguous in memory; if they were not, then spatial locality would be irrelevant.

### 6.7.1 Python List vs. Linked List

Which of a normal list (i.e. a Python list) vs. a linked list would you expect to benefit more from a cache taking advantage of spatial locality? Explain.

### 6.7.2 Separate Chaining vs. Open Addressing

With a hash table, with which of separate chaining (with a linked list) vs. open addressing (with linear probing) would you expect to benefit more from a cache taking advantage of spatial locality? Explain.

### 6.7.3 Linear Probing vs. Quadratic Probing

With a hash table using open addressing, with which of linear probing vs. quadratic probing would you expect to benefit more from a cache taking advantage of spatial locality? Explain.

## 6.8 Check Duplicates

Suppose that you are writing a function that checks if a list of values contains any duplicate values. That is, the function should return `True` if the list does and `False` otherwise. **What would be the worst-case time complexity of each of the approaches to this function? Explain. Where it would help your explanation, you might want to explain what a worst-case scenario would be for that specific approach and/or use one of the two summations mentioned below.** In some cases, I use classes that you may assume do what the names suggest, e.g. `HashTable`, and fake method names that also do what their names suggest, e.g. `insert()`; this is not valid Python, since those classes are not built-in. You should make sure to not assume the `in` operator always has the same worst-case time complexity (e.g. linear time, constant time) no matter what the data structure is. Although the hash table operations `find`, `insert`, and `delete` take *amortized* constant time, you will not be penalized for omitting the word “amortized” in your analysis.

You may find these summations helpful. The first one was described during the lecture on selection sort and insertion sort. You can take these as fact, i.e. you don’t have to prove them. *You may find it appropriate to mention these summations in your analysis when appropriate.* Consequently, you should understand *why* the first summation was used in the worst-case analysis of selection sort.

- $\sum_{i=1}^n i = \Theta(n^2)$ .
- $\sum_{i=1}^n \lg i = \Theta(n \lg n)$ .

<sup>2</sup>The exact number depends on factors we will not get into, but certainly, the cache cannot grab all the elements that it wants to; there are limits.

### 6.8.1 Nested Loops

```
1 def HasDuplicate(vals):
2     for i in range(len(vals)):
3         for j in range(i + 1, len(vals)):
4             if vals[i] == vals[j]:
5                 return True
6     return False
```

### 6.8.2 Binary Search Tree

```
1 def HasDuplicate(vals):
2     bst = BST() # create instance of binary search tree
3     for val in vals:
4         if val in bst:
5             return True
6         bst.insert(val)
7     return False
```

### 6.8.3 AVL Tree

Same as the previous part, except that `bst` is an instance of `AVLTree` instead of `BST`.

### 6.8.4 Hash Table

```
1 def HasDuplicate(vals):
2     h = HashTable()
3     for val in vals:
4         if val in h:
5             return True
6         h.insert(val)
7     return False
```

Specify the worst-case time complexity in terms of the length of the input list only, not the size of the hash table.

## 6.9 Sorting Algorithms: Best-Case Analysis

For each of the following sorting algorithms, is there a meaningful distinction between the “best case” and the “worst case”. That is, would the best-case time complexity differ from the worst-case time complexity? If it does differ from the worst-case time complexity, then explain why and, if it helps your explanation, give an example of an input that would cause the best case to occur. If the best-case time complexity *does not* differ, then just say that it does not differ (e.g. “No difference.”) without any elaboration.

### 6.9.1 Bubble Sort

### 6.9.2 Selection Sort

### 6.9.3 Insertion Sort

### 6.9.4 Mergesort

