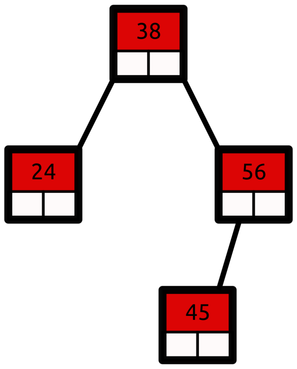
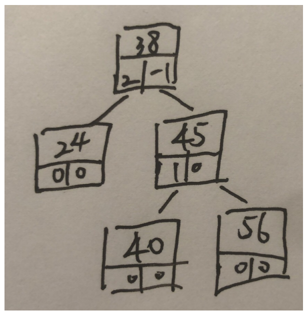
0.1 AVL Tree Insertion

0.1.1 Insert 40

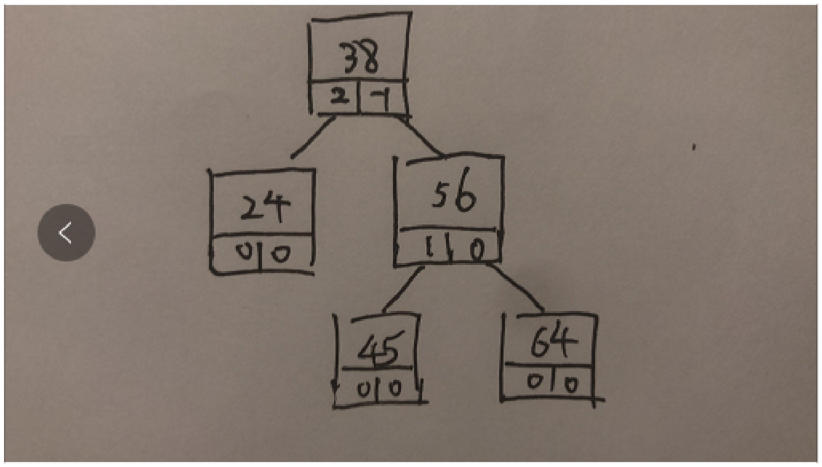
Show the AVL tree that results from inserting 40 into the below AVL tree. If you wish, you can draw the tree by hand and, after uploading an image of, include said image with `includegraphics`; see how it's used in the LaTeX file to include the image of the original tree, and note that the file extension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





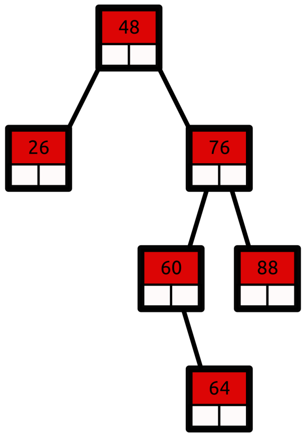
0.1.2 Insert 64

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.



0.2 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.

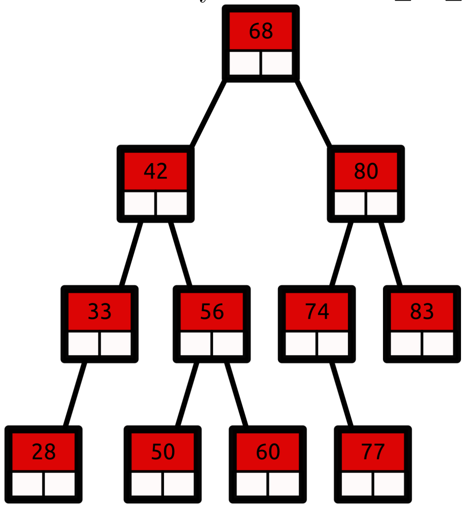


Unfortunately, after inserting 80, Professor Mean Dude is not sure how to rebalance the tree. Following the rules on slide #9 of slide deck #7, the balance factor of 80 is 0, the balance factor of 88 becomes 1, and the balance factor of 76 becomes 0. Since a node whose balance factor became 0 is encountered, 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.

This is because the original tree is a unbalanced tree. He needs to balance the original tree first and then insert the 80 node to the tree.

0.3 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 $0 \le x \le 100$, and you may assume that **no duplicate keys are allowed**.



$[-\infty, 33) \cap [43, 68) \cap [75, 80)$

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):
$[38, 52] \cap [67, 78)$

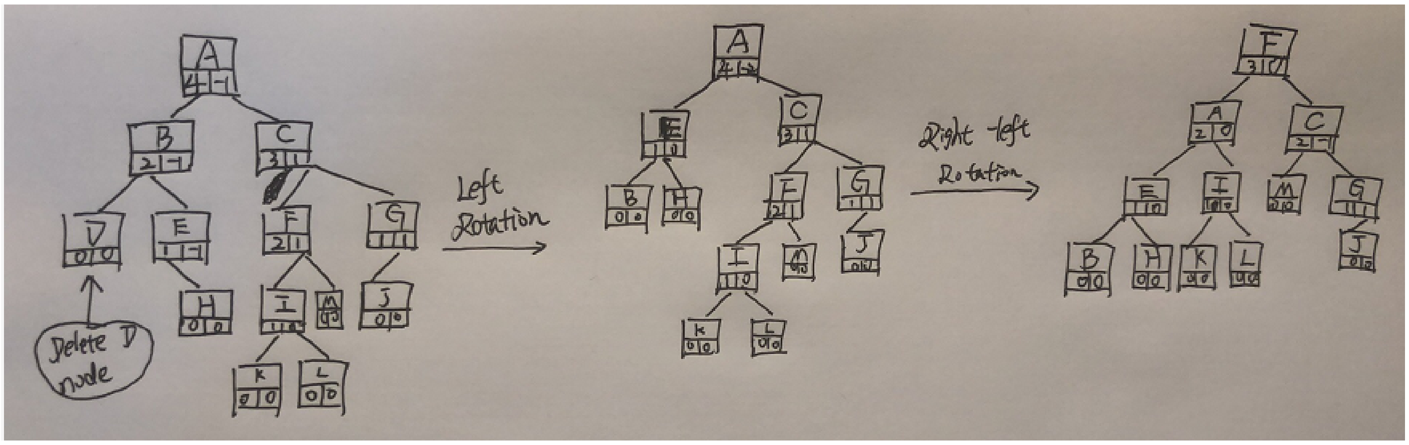
0.4 AVL Trees: Two Deletions

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.

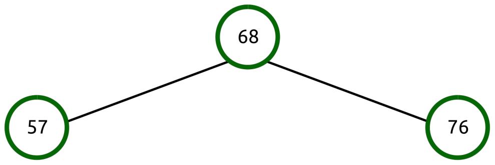
You must provide the following:

- 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.



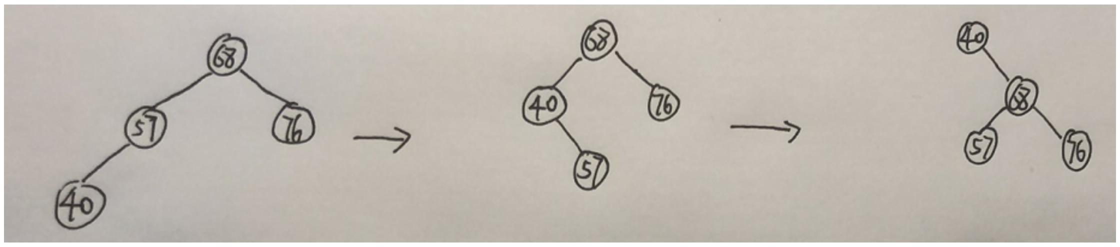
0.5 Splay Tree Insert and Find

Consider the splay tree shown below.

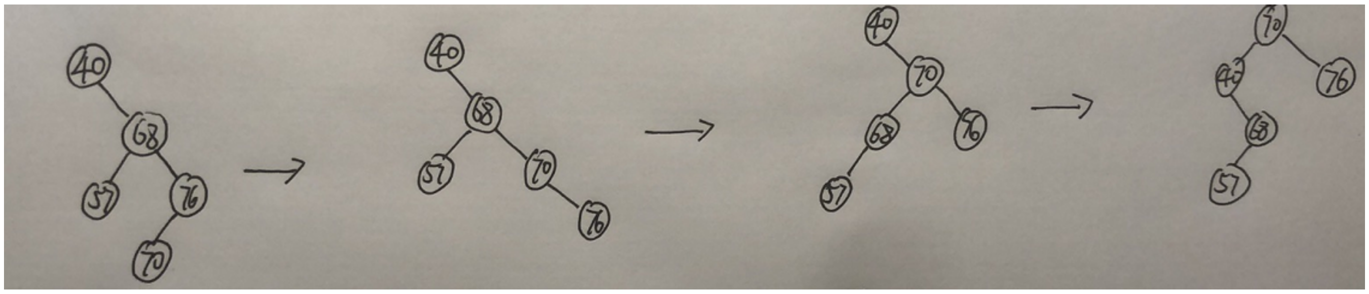


For each problem below, a splay tree operation is given. Draw the splay tree that results from doing that operation on the splay tree. The operations are cumulative, e.g. the third operation is performed on the tree that you got from the second operation. As with the earlier AVL tree problem, you may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

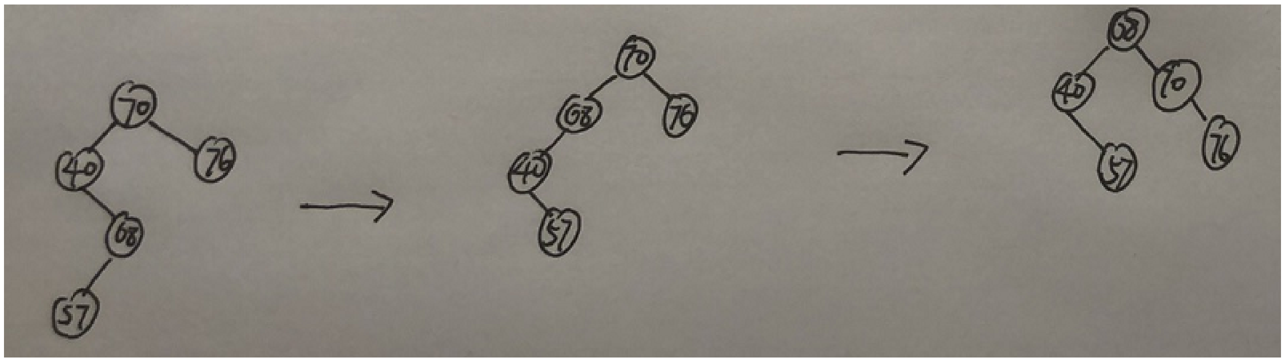
0.5.1 Insert 40



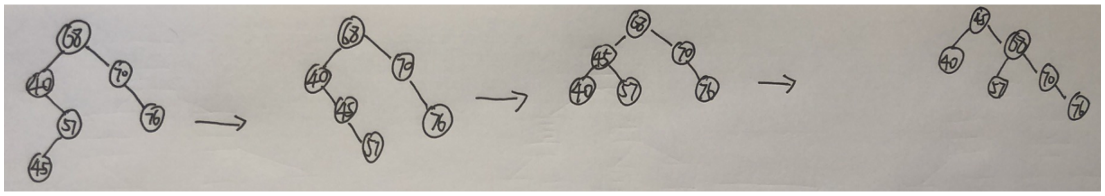
0.5.2 Insert 70



0.5.3 **Find** 68



0.5.4 **Insert** 45



0.6 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.

0.6.1 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.

If we implement 3-heap, for node at index i, parent is at $i//3$(Round down), left child is at $3i-1$, medium child is at $3i$ and right child is at $3i+1$.

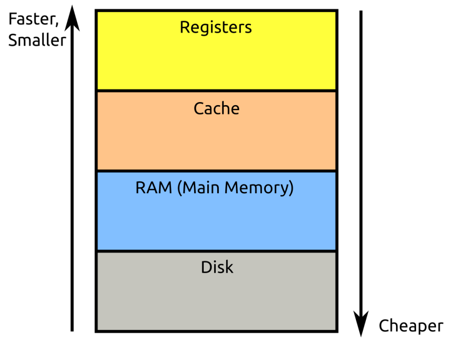
0.6.2 Zero-Based Indexing

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).

If we implement 3-heap, for node at index i, parent is at $i//3$(Round down) if $i\%3 \neq 0$, else parent is at $(i/3)-1$. left child is at $3i+1$, medium child is at $3i+2$ and right child is at $3i+3$.

0.7 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 (prounounced "cash")[1]. 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 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[2], 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* $A[i], A[i+1], A[i+2], A[i+3], \ldots$ *are contiguous in memory*; if they were not, then spatial locality would be irrelevant.

0.7.1 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.

Python list. Because the elements in python list are contiguous in memory. However, Link list is not.

0.7.2 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.

Open addressing (Linear probing). This is because linear probing will have a primary clustering situation. The probability of next position of Open addressing is not empty is much bigger than the probability of next position of separate chaining. This means elements are contiguous in memory.

0.7.3 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.

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

[2] 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.

Linear Probing. This is because linear probing will have a primary clustering situation. However, Quadratic Probing would not. Even though quadratic probing is vulnerable to secondary clustering but it is not contiguous with this node.

0.8 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.

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

0.8.1 Nested Loops

```python
def HasDuplicate(vals):
    for i in range(len(vals)):
        for j in range(i + 1, len(vals)):
            if vals[i] == vals[j]:
                return True
    return False
```

The worst-case time complexity of this function is $\Theta(n^2)$. The worst-case scenario is that there is no duplicate element in this list. We can use $\sum_{i=1}^{n} i = \Theta(n^2)$ to describe it.

0.8.2 Binary Search Tree

```python
def HasDuplicate(vals):
    bst = BST()  # create instance of binary search tree
    for val in vals:
        if val in bst:
            return True
        bst.insert(val)
    return False
```

The worst-case time complexity of this function is $\Theta(n^2)$. The worst-case scenario is that there is no duplicate element in this list. The find operation is $\Theta(n)$ and insert operation is $\Theta(n)$. Find and insert n times. We can use $\sum_{i=1}^{n} i = \Theta(n^2)$ to describe it.

0.8.3 AVL Tree

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

The worst-case time complexity of this function is $\Theta(n \log n)$. The worst-case scenario is that there is no duplicate element in this list. The find operation is $\Theta(n \log n)$ and insert operation is $\Theta(n \log n)$. Find and insert n times. We can use $\sum_{i=1}^{n} \lg i = \Theta(n \lg n)$. to describe it.

0.8.4 Hash Table

```python
def HasDuplicate(vals):
    h = HashTable()
    for val in vals:
        if val in h:
            return True
        h.insert(val)
    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.

The worst-case time complexity of this function is amortized $\Theta(n)$. The worst-case scenario is that there is no duplicate element in this list. The find operation is amortized $\Theta(1)$ and insert operation is amortized $\Theta(1)$. Find and insert n times. Because this function concern huge find and insert operation, so it is fine to use amortized $\Theta(1)$ to describe those operations. we can use $\sum_{i=1}^{n} 1 = \Theta(n)$ to describe it.

0.9 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.

0.9.1 Bubble Sort

It is meaningful distinction between the best case and the worst case. The best-case time complexity is $O(n)$. There is no adjacent element needs to exchange. The best-case is this list is sorted.

0.9.2 Selection Sort

No difference.

0.9.3 Insertion Sort

No difference.

0.9.4 Mergesort

No difference.

UCDAVIS
COMPUTER SCIENCE