

# Section 04: Trees

---

## Hashing Problems

### 1. More Hash Table Insertion!

For each problem, insert the given elements into the described hash table. Do not worry about resizing the internal array.

- (a) Suppose we have a hash table that uses linear probing and has an internal capacity of 13.

Insert the following elements in the EXACT order given using the hash function  $h(x) = 3x$ :

2, 4, 6, 7, 15, 13, 19

- (b) Suppose we have a hash table that uses quadratic probing and has an internal capacity of 10.

Insert the following elements in the EXACT order given using the hash function  $h(x) = x$ :

0, 1, 2, 5, 15, 25, 35

- (c) Suppose we have a hash table with an initial capacity of 8 using quadratic probing. We resize the hash table by doubling the capacity.

Suppose we insert the integer keys  $2^{20}, 2 \cdot 2^{20}, 3 \cdot 2^{20}, 4 \cdot 2^{20}, \dots$  using the hash function  $h(x) = x$ .

Describe what the runtime of the dictionary operations will over time as you keep adding these keys to the table.

### 2. Even More Hash Table Insertion!

- (a) Consider the following key-value pairs.

(6, a), (29, b), (41, d), (34, e), (10, f), (64, g), (50, h)

Suppose each key has a hash function  $h(k) = 2k$ . So, the key 6 would have a hash code of 12. Insert each key-value pair into the following hash tables and draw what their internal state looks like:

- (i) A hash table that uses separate chaining. The table has an internal capacity of 10. Assume each bucket is a linked list, where new pairs are appended to the end. Do not worry about resizing.
- (ii) A hash table that uses linear probing, with internal capacity 10. Do not worry about resizing.
- (iii) A hash table that uses quadratic probing, with internal capacity 10. Do not worry about resizing.

### 3. Hashing and Mutation

For the following problems, assume that:

1. IntList is a list of integers.
2. The hash code of an IntList is the sum of the integers in the list.
3. IntLists are considered equal only if they have the same size and the same values in the same order.
4. FourBucketHashMap uses separate chaining and the new items are added to the back of each bucket.
5. FourBucketHashMap always has four buckets and never resizes.

Consider the following code:

```
FourBucketHashMap<IntList, String> fbhm = new FourBucketHashMap<>();
IntList list1 = IntList.of(1, 2);
fbhm.put(list1, "dog");
// Part i
list1.add(3);
// Part ii
```

(a) At Part i (line 4), what will be returned from the following statement?

```
fbhm.get(IntList.of(1, 2));
```

(b) At Part II (line 6), what will be returned from the following statements?

```
fbhm.get(IntList.of(1, 2));
```

```
fbhm.get(IntList.of(1, 2, 3));
```

(c) Is there a problem with the code? If so, explain.

## 4. Debugging a Hash Table

Suppose we are in the process of implementing a hash map that uses open addressing and quadratic probing and want to implement the delete method.

- (a) Consider the following implementation of delete. List every bug you can find.

**Note:** You can assume that the given code compiles. Focus on finding run-time bugs, not compile-time bugs.

```
1      public class QuadraticProbingHashTable<K, V> {
2          private Pair<K, V>[] array;
3          private int size;
4
5          private static class Pair<K, V> {
6              public K key;
7              public V value;
8          }
9
10         // Other methods are omitted, but functional.
11
12         /**
13          * Deletes the key-value pair associated with the key, and
14          * returns the old value.
15          *
16          * @throws NoSuchElementException if the key-value pair does not exist in the method.
17          */
18         public V delete(K key) {
19             int index = key.hashCode() % this.array.length;
20
21             int i = 0;
22             while (this.array[index] != null && !this.array[index].key.equals(key)) {
23                 i += 1;
24                 index = (index + i * i) % this.array.length;
25             }
26
27             if (this.array[index] == null) {
28                 throw new NoSuchElementException("Key-value pair not in dictionary");
29             }
30
31             this.array[index] = null;
32
33             return this.array[index].value;
34         }
35     }
```

- (b) Let's suppose the Pair array has the following elements (pretend the array fit on one line):

["lily", V <sub>2</sub> ]	["castle", V <sub>6</sub> ]	["resource", V <sub>1</sub> ]	["hard", V <sub>9</sub> ]	["bathtub", V <sub>0</sub> ]
["wage", V <sub>4</sub> ]	["refund", V <sub>7</sub> ]	["satisfied", V <sub>6</sub> ]	["spring", V <sub>8</sub> ]	["spill", V <sub>3</sub> ]

And, that the following keys have the following hash codes:

Key	Hash Code
"bathtub"	9744
"resource"	4452
"lily"	7410
"spill"	2269
"wage"	8714
"castle"	2900
"satisfied"	9251
"refund"	8105
"spring"	6494
"hard"	9821

What happens when we call `delete` with the following inputs? Be sure write out the resultant array, and to do these method calls *in order*. (**Note:** If a call results in an infinite loop or an error, explain what happened, but don't change the array contents for the next question.)

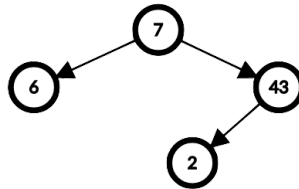
- (i) `delete("lily")`
  - (ii) `delete("spring")`
  - (iii) `delete("castle")`
  - (iv) `delete("bananas")`
  - (v) `delete(null)`
- (c) List four different test cases you would write to test this method. For each test case, be sure to either describe or draw out what the table's internal fields look like, as well as the expected outcome (assuming the `delete` method was implemented correctly). **Hint:** You may use the inputs previously given to help you identify tests, but it's up to you to describe what kind of input they are testing generally.

## AVL Tree Problems

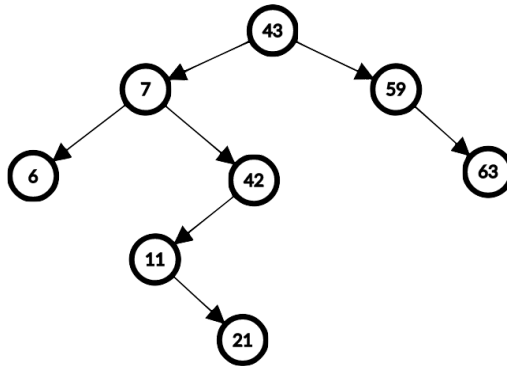
### 5. Valid BSTs and AVL Trees

For each of the following trees, state whether the tree is (i) a valid BST and (ii) a valid AVL tree. Justify your answer.

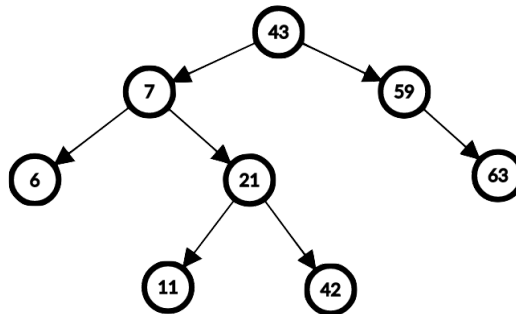
(a)



(b)



(c)



## 6. Constructing AVL trees

Draw an AVL Tree as each of the following keys are added in the order given. Show intermediate steps.

(a)

{“penguin”, “stork”, “cat”, “fowl”, “moth”, “badger”, “otter”, “shrew”, “lion”, “raven”, “bat”}

(b)

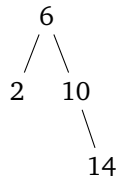
{6, 43, 7, 42, 59, 63, 11, 21, 56, 54, 27, 20, 36}

(c)

{“indigo”, “fuchsia”, “pink”, “goldenrod”, “violet”, “khaki”, “red”, “orange”, “maroon”, “crimson”, “green”, “mauve”}

## 7. AVL tree rotations

Consider this AVL tree:



Give an example of a value you could insert to cause:

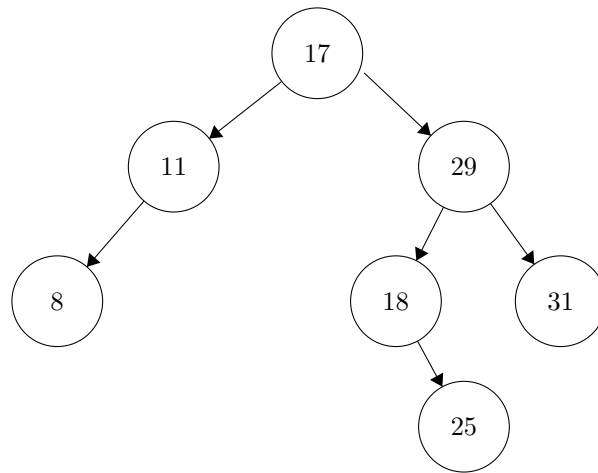
(a) A single rotation

(b) A double rotation

(c) No rotation

## 8. Inserting keys and computing statistics

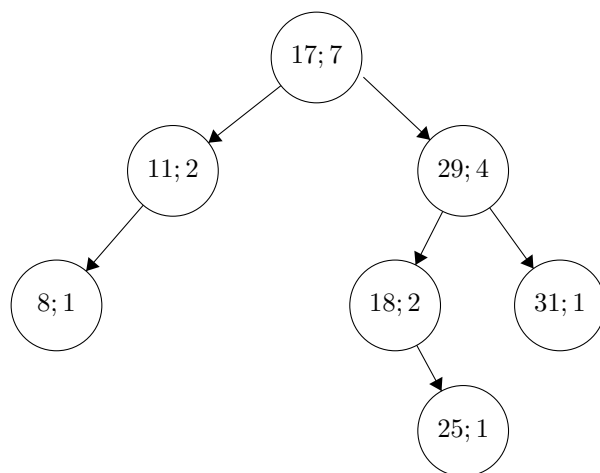
In this problem, we will see how to compute certain statistics of the data, namely, the minimum and the median of a collection of integers stored in an AVL tree. Before we get to that let us recall insertion of keys in an AVL tree. Consider the following AVL tree:



- (a) We now add the keys  $\{21, 14, 20, 19\}$  (in that order). Show where these keys are added to the AVL tree. Show your intermediate steps.
- (b) Recall that if we use an unsorted array to store  $n$  integers, it will take us  $O(n)$  runtime in order to compute the minimum element in the array. This can be done by running a loop that scans the array from the first index to the last index, which keeps track of the minimum element that it has seen so far. Now we will see how to compute the minimum element of a set of integers stored in an AVL tree which runs \*much\* faster than the procedure described above.
- Given an AVL tree storing numbers, like the one above, describe a procedure that will return the minimum element stored in the tree.
  - Supposing an AVL tree has  $n$  elements, what is the runtime of the above procedure in terms of  $n$ ? How does this runtime compare with the  $O(n)$  runtime of the linear scan of the array?
- (c) In the next few problems, we will see how to compute the median element of the set of elements stored in the AVL tree. The median of a set of  $n$  numbers is the element that appears in the  $\lceil n/2 \rceil$ -th position, when this set is written in sorted order. When  $n$  is even,  $\lceil n/2 \rceil = n/2$  and when  $n$  is odd,  $\lceil n/2 \rceil = (n + 1)/2$ . For example, if the set is  $\{3, 2, 1, 4, 6\}$  then the set in sorted order is  $\{1, 2, 3, 4, 6\}$ , and the median is 3.

If we were to simply store  $n$  integers in an array, one way to compute the median element would be to first sort the array and then look up the element at the  $\lceil n/2 \rceil$ -th position in the sorted array. This procedure has a runtime of  $O(n \log n)$ , even when we use a clever sorting algorithm like Mergesort. We will now see how to compute the median, when the data is stored in a rather modified AVL tree \*much\* faster.

For the time being, assume that we have a modified version of the AVL tree that lets us maintain, not just the key but also the number of elements that occur below the node at which the key is stored plus one (for that node). The use of this will become apparent very soon. As an example, the modified version of the AVL tree above, would like so (the number after the semi-colon in each node accounts for the number of nodes below that node plus one).



- i. We now again add the keys  $\{21, 14, 20, 19\}$  (in that order) to the modified AVL tree. How does the modified AVL tree look after the insertions are done?
  
- ii. Given a modified AVL tree, like the one above, describe a procedure that will return the median element stored in the tree. Note that in the modified tree, you can access the number of elements lying below a node in addition to the number stored in that node. Can you use this extra information to find the median more quickly?
  
- iii. Supposing a modified AVL tree has  $n$  elements, what is the runtime of the above procedure in terms of  $n$ ? How does this runtime compare with the  $O(n \log n)$  runtime described earlier?
  
- iv. Bonus: After every insertion, the number of nodes that lie below a given node need not remain the same. For example, after four insertions, the number of nodes below the root increased and the number of nodes below the node where the key "29" was stored, decreased. Describe a procedure that takes as input a modified AVL tree  $T$  with  $n$  nodes, an integer key  $k$  and, returns the modified AVL  $T'$ , that has the key  $k$  inserted in  $T$ . What is the runtime of this procedure?



## 9. Big- $\mathcal{O}$

Write down a tight big- $\mathcal{O}$  for each of the following. Unless otherwise noted, give a bound in the worst case.

- (a) Insert and find in a BST.
- (b) Insert and find in an AVL tree.
- (c) Finding the minimum value in an AVL tree containing  $n$  elements.
- (d) Finding the  $k$ -th largest item in an AVL tree containing  $n$  elements.
- (e) Listing elements of an AVL tree in sorted order

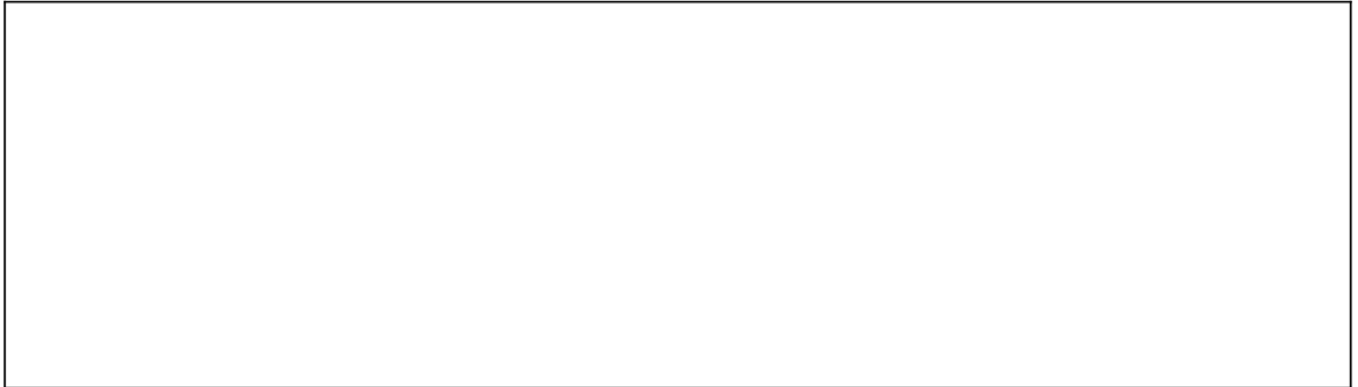
## 10. Analyzing dictionaries

- (a) What are the constraints on the data types you can store in an AVL tree?
- (b) When is using an AVL tree preferred over a hash table?
- (c) When is using a BST preferred over an AVL tree?
- (d) Consider an AVL tree with  $n$  nodes and a height of  $h$ . Now, consider a single call to `get(...)`. What's the maximum possible number of nodes `get(...)` ends up visiting? The minimum possible?
- (e) **Challenge Problem:** Consider an AVL tree with  $n$  nodes and a height of  $h$ . Now, consider a single call to `insert(...)`. What's the maximum possible of nodes `insert(...)` ends up visiting? The minimum possible? Don't count the new node you create or the nodes visited during rotation(s).

## Trie Problems

### 1. Trie to Delete 0's and 1's?


(a) Insert all possible binary strings of lengths 0-3 (i.e. "", "1", "0", "10", ..., "110", "111") into a `Trie`.



(b) From here, remove all binary strings of length 2 (e.g. "00"). How many nodes would disappear? Why?



(c) From here, remove all binary strings of length 3 (e.g. "000"). How many nodes would disappear? Why?



## 2. Call Me Maybe

Suppose you want to transfer someone's phone book to a data structure so that you can call all the phone numbers with a particular area code efficiently.

(a) What data structure would you use? How would you implement it? There are a few answers here.

(b) What is the time complexity of your solution?

(c) What is the space complexity?

### 3. Let's Trie to be Old School

Text on nine keys (T9)'s objective is to make it easier to type text messages with 9 keys. It allows words to be entered by a single keypress for each letter in which several letters are associated with each key. It combines the groups of letters on each phone key with a fast-access dictionary of words. It looks up in the dictionary all words corresponding to the sequence of keypresses and orders them by frequency of use. So for example, the input '2665' could be the words {book, cook, cool}. Describe how you would implement a T9 dictionary for a mobile phone.



T9 Example