Section 04: Solutions

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

Solution:

Again, we start by forming the table:

key	hash	index (before probing)
2	6	6
4	12	12
6	18	5
7	21	8
15	45	6
13	39	0
19	57	5

Next, we insert each element into the internal array, one-by-one using linear probing to resolve collisions. The state of the internal array will be:

13	/	/	/	/	6	2	15	7	19	/	/	4

(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

Solution:

The state of the internal array will be:

0	1	2	/	35	5	15	/	/	25

(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 $2^{20}, 2 \cdot 2^{20}, 3 \cdot 2^{20}, 4 \cdot 2^{20}, \ldots$ 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.

Solution:

Initially, for the first few keys, the performance of the table will be fairly reasonable.

However, as we insert each key, they will keep colliding with each other: the keys will all initially mod to index 0.

This means that as we keep inserting, each key ends up colliding with every other previously inserted key, causing all of our dictionary operations to take $\mathcal{O}(n)$ time.

However, once we resize enough times, the capacity of our table will be larger then 2^{20}, which means that our keys no longer necessarily map to the same array index. The performance will suddenly improve at that cutoff point then.

2. Even More Hash Table Insertion!

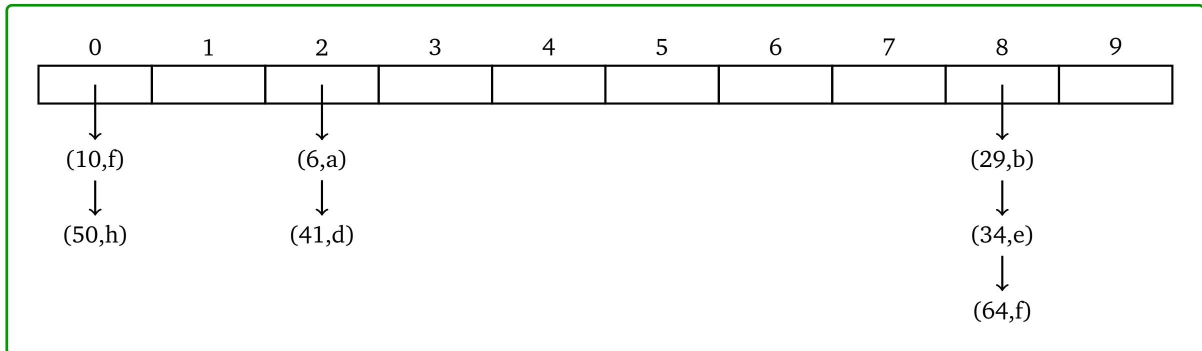
(a) Consider the following key-value pairs.

(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:

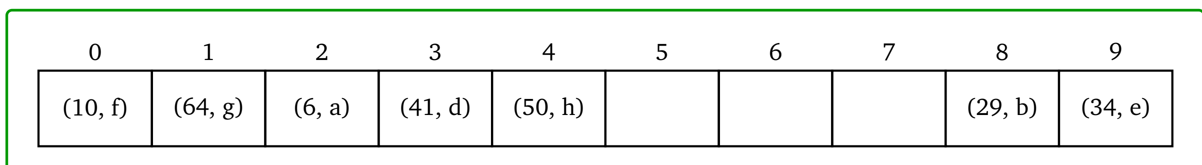
(i) A hash table that uses separate chaining. The table has an internal capacity of 10. Assume each bucket is a linked list, where new pairs are appended to the end. Do not worry about resizing.

Solution:



(ii) A hash table that uses linear probing, with internal capacity 10. Do not worry about resizing.

Solution:



0	1	2	3	4	5	6	7	8	9
(10, f)	(64, g)	(6, a)	(41, d)	(50, h)				(29, b)	(34, e)

(iii) A hash table that uses quadratic probing, with internal capacity 10. Do not worry about resizing.

Solution:

0	1	2	3	4	5	6	7	8	9
(10,f)	(50, h)	(6,a)	(41,d)				(64, g)	(29,b)	(34,e)

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));
```

Solution:

"dog"

This will look up the bucket $(1 + 2) \bmod 4 = 3$. In the bucket 3, IntList.of(1, 2) is equivalent to [1, 2], so "dog" which is the stored value is returned.

See Section slides for more details.

(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));
```

Solution:

"null"

"null"

The first get function will look up the bucket $(1 + 2) \bmod 4 = 3$. In the bucket 3, IntList.of(1, 2) is NOT equivalent to [1, 2, 3], so we cannot find the matched key. Hence, return null.

The second get function will loop up the bucket $(1 + 2 + 3) \bmod 4 = 2$. Since the bucket 2 is empty, we definitely cannot find the matched key. Hence, return null.

See Section slides for more details.

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

Solution:

Adding 3 into `list1` changes its hash code, causing `list1` to live in the wrong bucket.

See Section slides for more details.

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 NoSuchKeyException 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 NoSuchKeyException("Key-value pair not in dictionary");
29         }
30
31         this.array[index] = null;
32
33         return this.array[index].value;
34     }
35 }
```

Solution:

The full list of all bugs:

(i) If the dictionary contains any null keys, this code will crash. (See the call to `.equals(...)` in the while loop condition.)

(ii) If the key parameter is null, the code will crash. (See the call to `.hashCode(...)` at the top of the method.)

(iii) If the key's hashCode is negative, this code will crash. (We try indexing a negative element).

(iv) We probe the array incorrectly. If s is the initial position we check, we ought to be checking s, $s+1$, $s+4$, $s+9$, $s+16$...

Instead, we check s, $s+1$, $s+5$, $s+14$, $s+30$...

(v) Nulling out the array index will break all subsequent deletes. Suppose we have a collision, and our algorithm ends up checking index locations 0, 1, 5, 14, and 30 respectively.

If we null out index 5, then all subsequent probes starting at index 0 will be unable to find whatever's located at 14 or 30.

(vi) The final return has a null pointer exception – we null out that pair before fetching the value.

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

["lily", V_2]	["castle", V_6]	["resource", V_1]	["hard", V_9]	["bathtub", V_0]
["wage", V_4]	["refund", V_7]	["satisfied", V_6]	["spring", V_8]	["spill", V_3]

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")`

Solution:

Nothing bad happens:

`null`	["castle", V_6]	["resource", V_1]	["hard", V_9]	["bathtub", V_0]
["wage", V_4]	["refund", V_7]	["satisfied", V_6]	["spring", V_8]	["spill", V_3]

(ii) `delete("spring")`

Solution:

We don't probe correctly in general (see bug above), but we *do* happen to loop around eventually to remove "spring". This code is inefficient, and won't always work out like this, but in this case, we managed a successful delete:

null	["castle", V_6]	["resource", V_1]	["hard", V_9]	["bathtub", V_0]
["wage", V_4]	["refund", V_7]	["satisfied", V_6]	null	["spill", V_3]

(iii) `delete("castle")`

Solution:

We stop after we see `array[0]` is null, and we throw a `NoSuchKeyException`, without continuing to probe.

(iv) `delete("bananas")`

Solution:

`NoSuchKeyException`, but that's what we wanted, so no bugs caught.

(v) `delete(null)`

Solution:

`NullPointerException`. Note, this is *not* what we wanted, because `Pairs` support null keys. This code should have returned a `NoSuchKeyException`.

(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.

Solution:

Some test cases include:

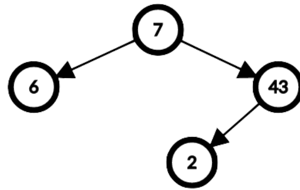
- Picking a key not present in the dictionary. This should trigger an exception (and not change the size).
- Picking a key present in the dictionary. This should succeed, and return the old value (and decrease the size by 1).
- Inserting and attempting to delete a null key. This should succeed (and decrease the size by 1).
- Deleting a key that forces us to probe a few times. This should succeed (and decrease the size, etc).
- Deleting a key in the middle of some probe sequence. All subsequent calls to delete/get/etc should correctly.
- Using a key with a negative hashcode should behave as expected.

AVL Tree Problems

5. Valid BSTs and AVL Trees

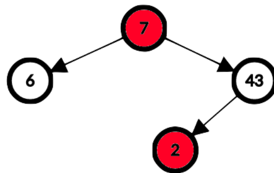
For each of the following trees, state whether the tree is (i) a valid BST and (ii) a valid AVL tree. Justify your answer.

(a)



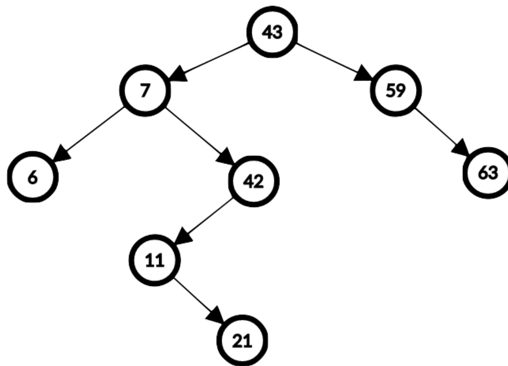
Solution:

This is not a valid BST! The 2 is located in the right sub-tree of 7, which breaks the BST property. Remember that the BST property applies to **every** node in the left and right sub-trees, not just the immediate child!



All AVL trees are BSTs. Because of this, this tree can't be a valid AVL tree either.

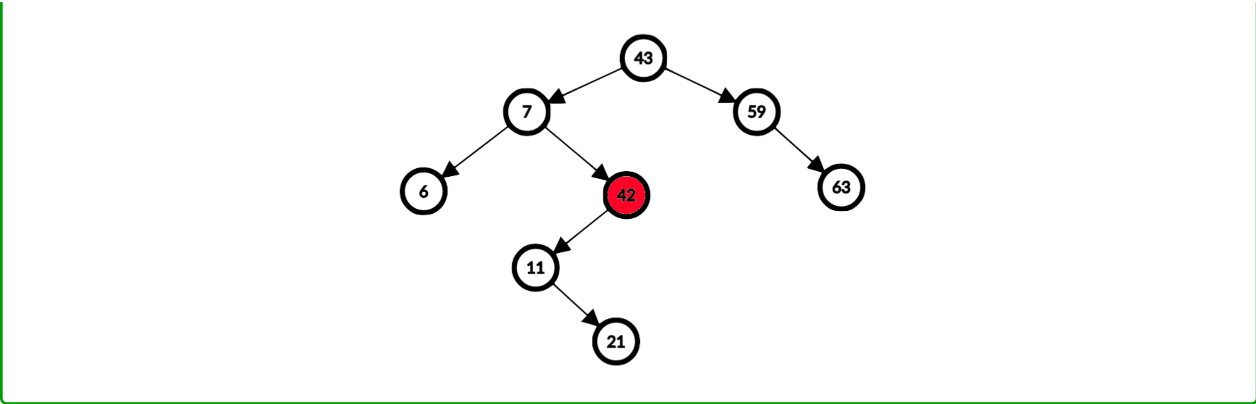
(b)



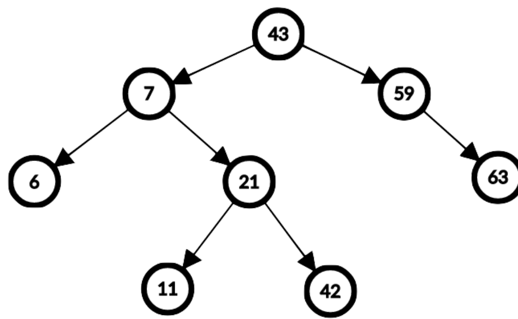
Solution:

This tree is a valid BST! If we check every node, we see that the BST property holds at each of them.

However, this is not a valid AVL tree. We see that some nodes (for example, the 42) violate the balance condition, which is an extra requirement compared to BSTs. Because the heights of 42's left and right sub-trees differ by more than one, this violates the condition.



(c)



Solution:

This tree is a valid BST! If we check every node, we see that the BST property holds at each of them.

This tree is also a valid AVL tree! If we check every node, we see that the balance condition also holds at each of them.

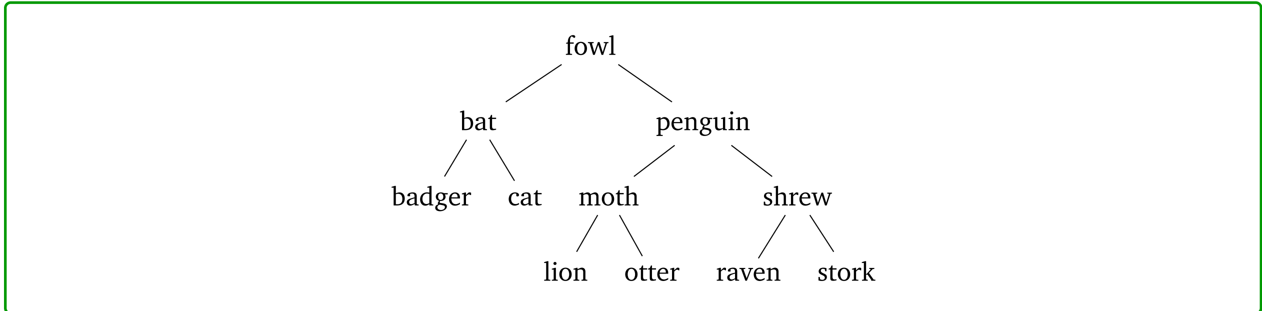
6. Constructing AVL trees

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"}

Solution:

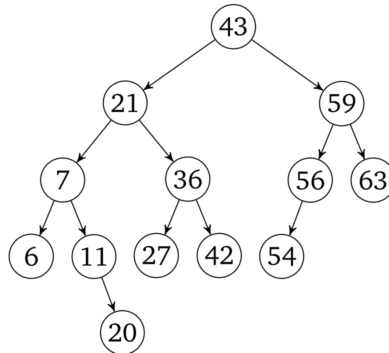


(b)

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

Solution:

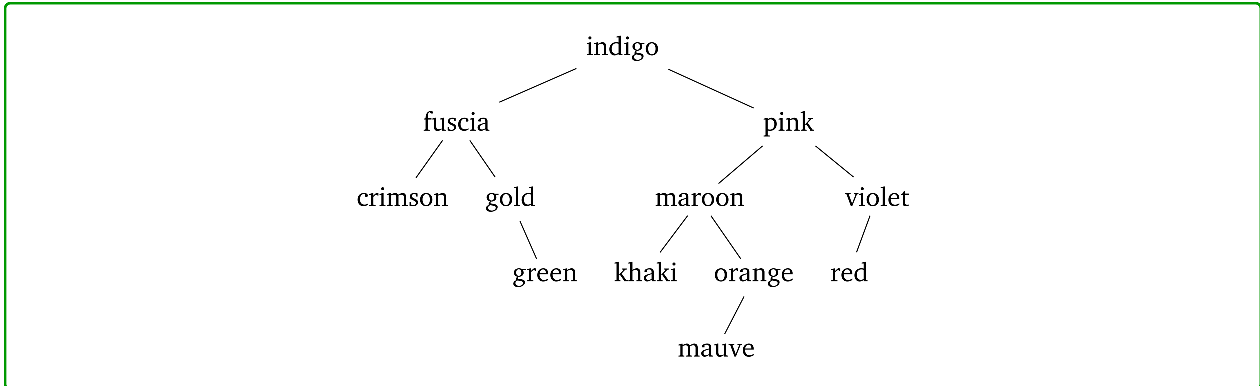
Note: The Section slides have a step-by-step walkthough of this one!



(c)

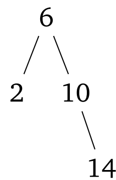
{“indigo”, “fuscia”, “pink”, “goldenrod”, “violet”, “khaki”, “red”, “orange”, “maroon”, “crimson”, “green”, “mauve”}

Solution:



7. AVL tree rotations

Consider this AVL tree:



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

(a) A single rotation

Solution:

Any value greater than 14 will cause a single rotation around 10 (since 10 will become unbalanced, but we'll be in the line case).

(b) A double rotation

Solution:

Any value between 10 and 14 will cause a double rotation around 10 (since 10 will be unbalanced, and we'll be in the kink case).

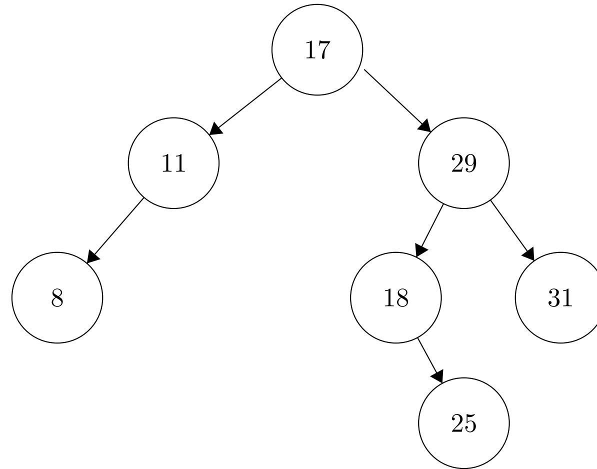
(c) No rotation

Solution:

Any value less than 10 will cause no rotation (since we can't cause any node to become unbalanced with those values).

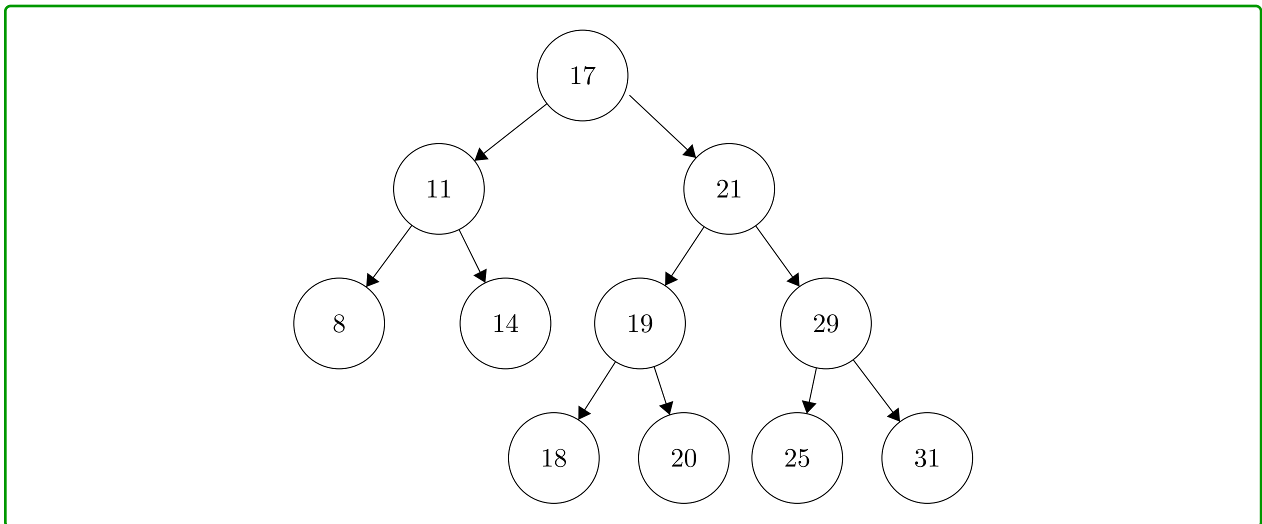
8. Inserting keys and computing statistics

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.

Solution:



(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.

i. Given an AVL tree storing numbers, like the one above, describe a procedure that will return the minimum element stored in the tree.

Solution:

Remember that an AVL tree satisfies the BST property, i.e. for any node, all keys in the left sub-tree below that node must be smaller than all the keys in the right sub-tree. Since the minimum is the smallest element in the tree, it must lie in the left sub-tree below the root. By the same reasoning, the minimum must also lie in the left sub-tree below the left node connected to the root and so on and so forth.

Proceeding this way, we can set l_0 to be the root of the tree and for all $i \geq 1$, we can set l_i to the left node connected to l_{i-1}. By our reasoning above, the minimum lies in the subtree below l_i for every i. Hence, we can simply start at the root i.e. l_0 and keep following the edge towards the nodes $l_1, l_2, \ldots$ until we hit a leaf! The leaf must be the minimum, as there is no subtree rooted below it.

ii. 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?

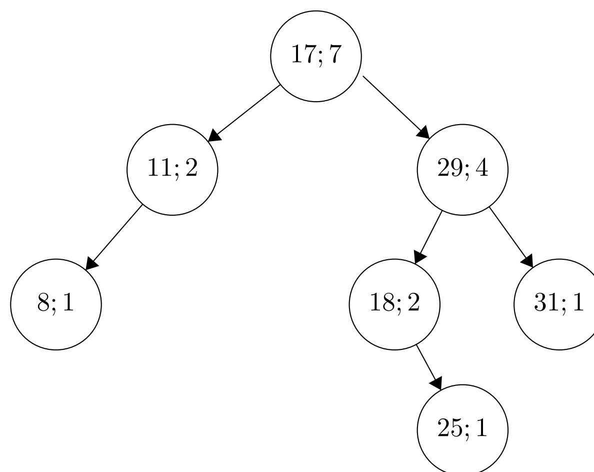
Solution:

The above procedure, is essentially a loop that starts at the root and stops when it reaches a leaf. The length of any path from the root to a leaf in an AVL tree with n elements is a most $O(\log n)$. Hence, the above procedure has runtime $O(\log n)$. This runtime is exponentially better than the linear scan which takes $O(n)$ time!

(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.

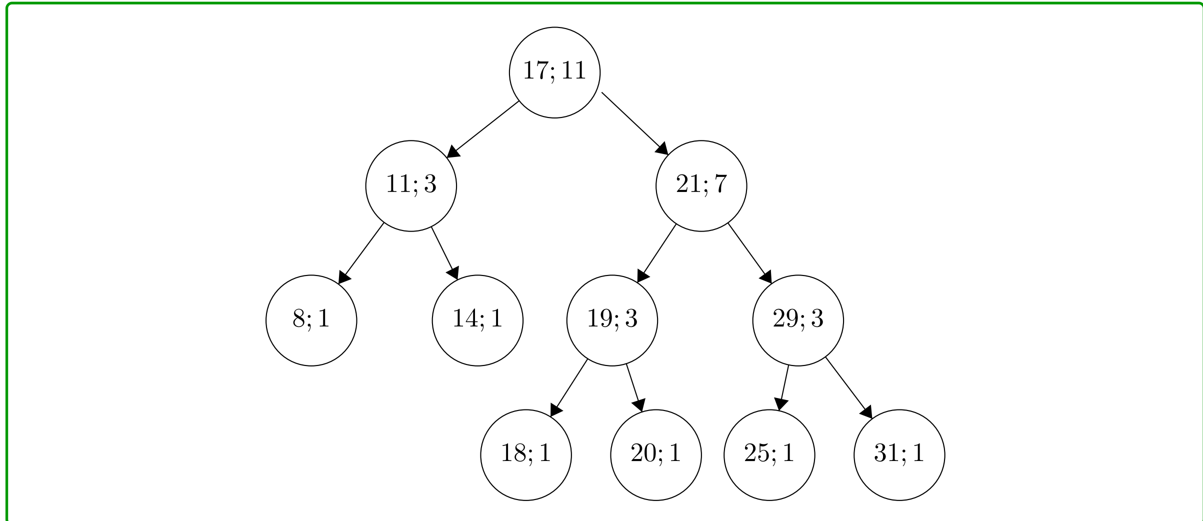
If we were to simply store n integers in an array, one way to compute the median element would be to first sort the array and then look up the element at the $\lceil n/2 \rceil$-th position in the sorted array. This procedure has a runtime of $O(n \log n)$, even when we use a clever sorting algorithm 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?

Solution:



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?

Solution:

We will actually show that using a modified AVL tree, we can compute the k-th smallest element for any k. The k-th smallest element of a set of n numbers is the number at index k when the set is written in sorted order. Note that this problem is more general than computing the median! If we plug $k = \lceil n/2 \rceil$, we can compute the median!

Similar to the strategy that we used to compute the minimum, we start by setting l_0 to be the root of the tree. At this point, we check the number of nodes that lie below the left and right nodes connected to the root. Let these numbers be $x_{l_0,0}$ and $x_{l_0,1}$ respectively. We consider three cases below.

I. Let us suppose for the moment that $x_{l_0,0} = k - 1$. We observe that if the elements in the AVL tree were to be written in sorted order, all the elements in the left subtree below root would appear before the root, which itself would appear before the elements in the right sub-tree. Since there are $k - 1$ elements in the left subtree, the index of the root is k, which is the desired element.

II. Now suppose $x_{l_0,0} < k - 1$. Again, if we were to write the elements in the AVL tree in sorted order, the k-th smallest element would now lie in the subtree below the right node.

III. Finally, if $x_{l_0,0} > k - 1$, the k-th smallest number would lie in subtree below the left node.

The upshot of this is that by checking the number of nodes in the left and right subtrees below a given node, we were able to find out which subtree the k-th smallest element lies in! We can repeat this, recursing in the appropriate subtree. For example, if $x_{l_0,0} < k - 1$ then we recurse in the right subtree. However, the k-th smallest element in the entire tree may not be the k-th smallest element in the right subtree!

We want to find out what element we need to look for in the right subtree in order to find the k-th smallest element in the entire tree. Let us suppose that the k-th smallest element in the entire subtree is in fact the k'-th smallest element in the right subtree. If we were to write out the elements in the AVL tree in sorted order, it follows that the k'-th smallest element in the right subtree is at position $(x_{l_0,0} + 1) + k'$ in the entire tree. But this position is also the position of the k-th smallest element. It follows that

$$(x_{l_0,0} + 1) + k' = k \implies k' = k - (x_{l_0,0} + 1).$$

Therefore, in order to locate the k-th smallest element in the entire tree, it suffices to locate the $(k - (x_{l_0,0} + 1))$-th smallest element in the right subtree, which we can do as detailed above.

We repeat the procedure until, we either find the k-th smallest element to be a node, like in Case **I**, or we hit a leaf.

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?

Solution:

The above procedure, is essentially a loop that starts at the root and stops when it reaches a leaf. The length of any path from the root to a leaf in an AVL tree with n elements is a most $O(\log n)$. Hence, the above procedure has runtime $O(\log n)$. This runtime is far far better than the solution based on sorting which takes $O(n \log n)$ time; we managed a shave off the linear term in the latter expression!

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.

Solution:

$\mathcal{O}(n)$ and $\mathcal{O}(n)$, respectively. This is unintuitive, since we commonly say that `find()` in a BST is "`log(n)`", but we're asking you to think about *worst-case* situations. The worst-case situation for a BST is that the tree is a linked list, which causes `find()` to reach $\mathcal{O}(n)$.

(b) Insert and find in an AVL tree.

Solution:

$\mathcal{O}(\log(n))$ and $\mathcal{O}(\log(n))$, respectively. The worst case is we need to insert or find a node at height 0. However, an AVL tree is always a balanced BST tree, which means we can do that in $\mathcal{O}(\log(n))$.

(c) Finding the minimum value in an AVL tree containing n elements.

Solution:

$\mathcal{O}(\log(n))$. We can find the minimum value by starting from the root and constantly traveling down the left-most branch.

(d) Finding the k-th largest item in an AVL tree containing n elements.

Solution:

With a standard AVL tree implementation, it would take $\mathcal{O}(n)$ time. If we're located at a node, we have no idea how many elements are located on the left vs right and need to do a traversal to find out. We end up potentially needing to visit every node.

If we modify the AVL tree implementation so every node stored the number of children it had at all times (and updated that field every time we insert or delete), we could do this in $\mathcal{O}(\log(n))$ time by performing a binary search style algorithm.

(e) Listing elements of an AVL tree in sorted order

Solution:

$\mathcal{O}(n)$. An AVL tree is always a balanced BST tree, which means we only need to traverse the tree in in-order once.

10. Analyzing dictionaries

(a) What are the constraints on the data types you can store in an AVL tree?

Solution:

The keys need to be orderable because AVL trees (and BSTs too) need to compare keys with each other to decide whether to go left or right at each node. (In Java, this means they need to implement `Comparable`). Unlike a hash table, the keys do *not* need to be hashable. (Note that in Java, every object is technically hashable, but it may not hash to something based on the object's value. The default hash function is based on reference equality.)

The values can be any type because AVL trees are only ordered by keys, not values.

(b) When is using an AVL tree preferred over a hash table?

Solution:

(i) You can iterate over an AVL tree in sorted order in $\mathcal{O}(n)$ time.

(ii) AVL trees never need to resize, so you don't have to worry about insertions occasionally being very slow when the hash table needs to resize.

(iii) In some cases, comparing keys may be faster than hashing them. (But note that AVL trees need to make $\mathcal{O}(\log n)$ comparisons while hash tables only need to hash each key once.)

(iv) AVL trees *may* be faster than hash tables in the worst case since they guarantee $\mathcal{O}(\log n)$, compared to a hash table's $\mathcal{O}(n)$ if every key is added to the same bucket. But remember that this only applies to pathological hash functions. In most cases, hash tables have better asymptotic runtime ($\mathcal{O}(1)$) than AVL trees, and in practice $\mathcal{O}(1)$ and $\mathcal{O}(\log n)$ have roughly the same performance.

(c) When is using a BST preferred over an AVL tree?

Solution:

One of AVL tree's advantages over BST is that it has an asymptotically efficient `find` even in the worst case.

However, if you know that `insert` will be called more often than `find`, or if you know the keys will be inserted in a random enough order that the BST will stay balanced, you may prefer a BST since it avoids the small runtime overhead of checking tree balance properties and performing rotations. (Note that this overhead is a constant factor, so it doesn't matter asymptotically, but may still affect performance in practice.)

BSTs are also easier to implement and debug than AVL trees.

(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?

Solution:

The max number is $h + 1$ (remember that height is the number of edges, so we visit $h + 1$ nodes going from the root to the farthest away leaf); the min number is 1 (when the element we're looking for is just the root).

(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).

Solution:

The max number is $h + 1$. Just like a `get`, we may have to traverse to a leaf to do an insertion.

To find the minimum number, we need to understand which elements of AVL trees we can do an insertion at, i.e. which ones have at least one `null` child.

In a tree of height 0, the root is such a node, so we need only visit the one node.
In an AVL tree of height 1, the root can still have a (single) `null` child, so again, we may be able to do an insertion visiting only one node.

On taller trees, we always start by visiting the root, then we continue the insertion process in either a tree of height $h - 1$ or a tree of height $h - 2$ (this must be the case since the the overall tree is height h and the root is balanced). Let $M(h)$ be the minimum number of nodes we need to visit on an insertion into an AVL tree of height h. The previous sentence lets us write the following recurrence

$$M(h) = 1 + \min\{M(h-1), M(h-2)\}$$

The 1 corresponds to the root, and since we want to describe the minimum needed to visit, we should take the minimum of the two subtrees.

We could simplify this recurrence and try to unroll it, but it's easier to see the pattern if we just look at the first few values:

$$M(0) = 1, M(1) = 1, M(2) = 1 + \min\{1, 1\} = 2, M(3) = 1 + \min\{1, 2\} = 2, M(4) = 1 + \min\{2, 2\} = 3$$

In general, $M()$ increases by one every other time h increases, thus we should guess the closed-form has an $h/2$ in it. Checking against small values, we can get an exactly correct closed-form of:

$$M(h) = \lfloor h/2 \rfloor + 1$$

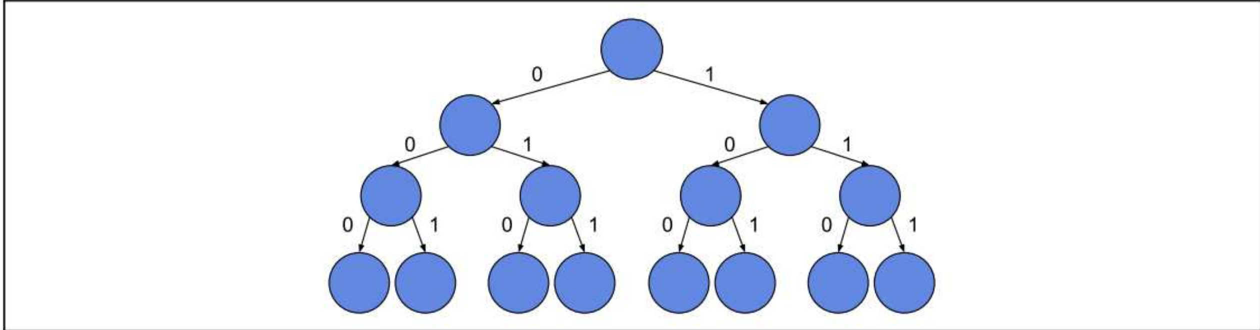
which is our final answer.

Note that we need a very special (as empty as possible) AVL tree to have a possible insertion visiting only $\lfloor h/2 \rfloor + 1$ nodes. In general, an AVL of height h might not have an element we could insert that visits only $\lfloor h/2 \rfloor + 1$. For example, a tree where all the leaves are at depth h is still a valid AVL tree, but any insertion would need to visit $h + 1$ nodes.

Trie Problems

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

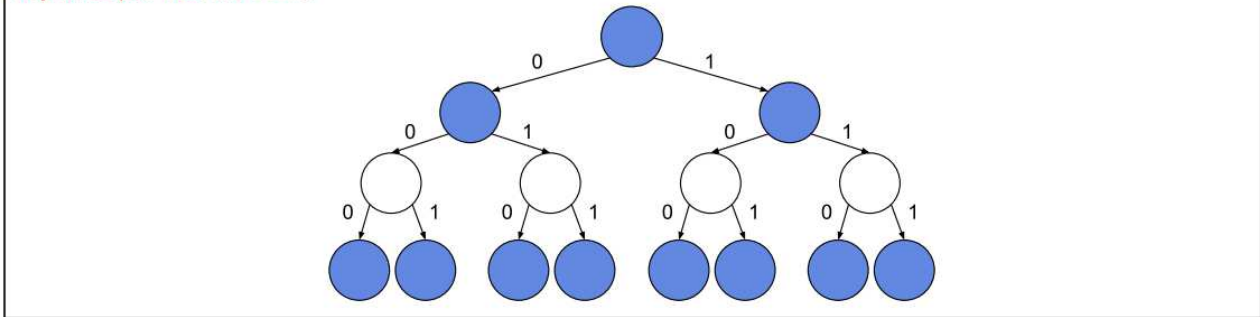
(a) Insert all possible binary strings of lengths 0-3 (i.e. "", "1", "0", "10", ..., "110", "111") into a `Trie`.



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

0 nodes

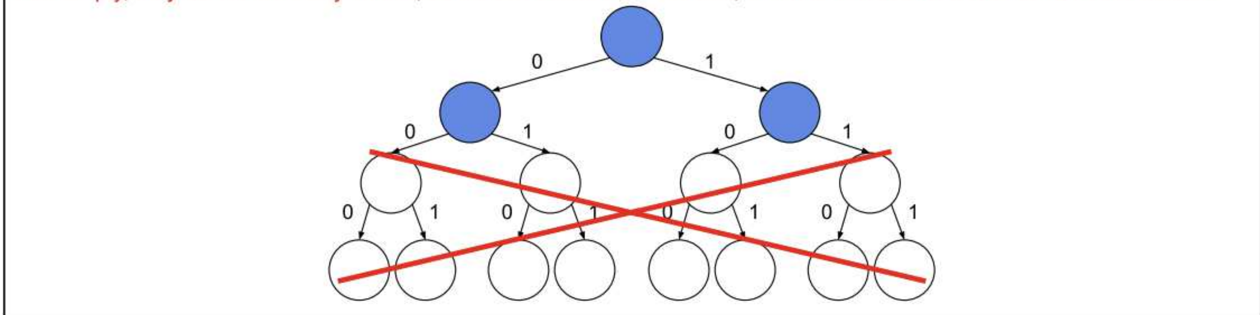
We still need the nodes storing the value for binary strings of length 2 because they have pointers to the nodes for binary strings of length 3, which still exist in the `Trie`. Therefore, we only set the value to `null` in the node to remove the key-value pair from the `Trie`.



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

12 nodes

Since the binary strings of length 3 are all leaf nodes, they do not have any pointers to other relevant nodes in the `Trie`, so we can delete them, which is 8 nodes. However, in part a, the nodes that used to store the binary strings of length 2 are now empty, they do not store any values, so we can delete those as well, which is 4 nodes for a total of 12 nodes deleted.



2. Call Me Maybe

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

One way to solve this would be using a `HashMap` where the keys are the area codes and the values are a list of corresponding phone numbers. We will need to parse the phone number to get the first three numbers.

Another way to solve this is by using a `Trie`. We would use the entire phone number as the "route" and insert all numbers into the `Trie`. Then, to find all the phone numbers to call, we would use the area code to partially travel down the `Trie`, then visit all children nodes to find the phone numbers to print.

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

If we compare the `HashMap` and `TrieMap` approaches, both will have the same runtime efficiency of $\Theta(n)$ to build and $\Theta(e)$ to call all the phone numbers with a particular area code efficiently.

If we let n be the total number of phone numbers and e be the expected number of phone numbers per area code, we can find that it takes $\Theta(n)$ time to build either the `HashMap` or the `Trie`. Likewise, given some area code, it takes $\Theta(e)$ time to visit and call each phone number.

Initially, it may seem like the `Trie` would be slower due to the traversals. However, recall that the depth of the `Trie` is always equal to the length of a phone number, which is a constant value.

(c) What is the space complexity?

Asymptotically, the `Trie` will generally be more space-efficient.

The reason why the `Trie` turns out to be more space-efficient on average is because the `Trie` is capable of storing near-duplicate phone numbers in less space then the `HashMap`. If we have the phone numbers `123-456-7890`, `123-456-7891`, and `123-456-7892`, the map must store each number individually whereas the `Trie` is able to combine them together and only branch for the very last number.

However, in practice, because each of the `Trie` nodes stores a pointer to the next node, it can quickly add up and take up a lot of memory.

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

One way to implement this would be by using a `Trie`. The routes (branches) are represented by the digits and the node's values are a collection of words. So if you typed in 2, 6, 6, 5, you would choose the child representing 2, then 6, then 6, then 5, traveling four layers deep into the `Trie`.

Then, that child node's value would contain a collection of all dictionary words corresponding to this particular sequence of numbers.

To populate the `Trie`, you would iterate through each word in the dictionary, and first convert the word into the appropriate sequence of numbers.

Then, you would use that sequence as the key or "route" to traverse the `Trie` and add the word.