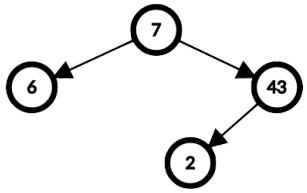
Section 05: Solutions

AVL Tree Problems

1. Valid BSTs and AVL Trees

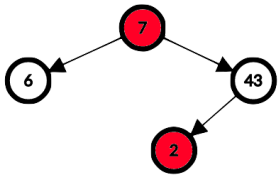
For each of the following trees, state whether the tree is (i) a valid BST and (ii) a valid AVL tree. Justify your answer.

(a)



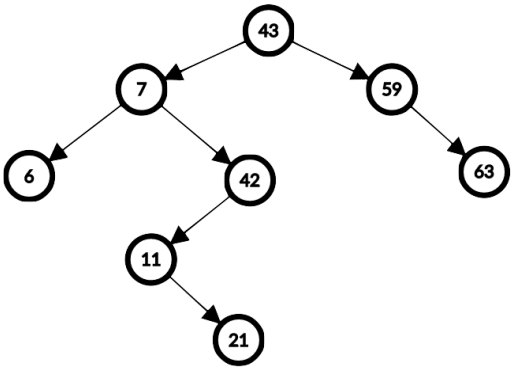
Solution:

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.

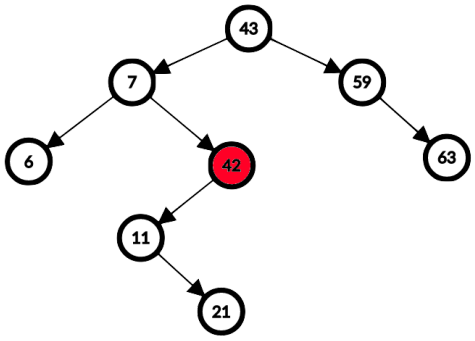
(b)



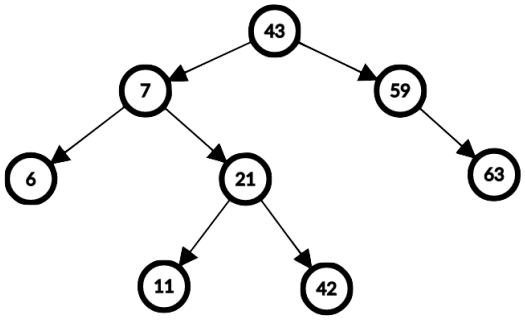
Solution:

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.

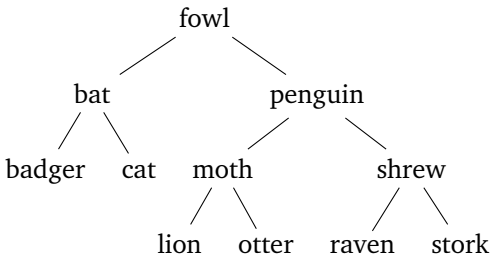
2. Constructing AVL trees

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:

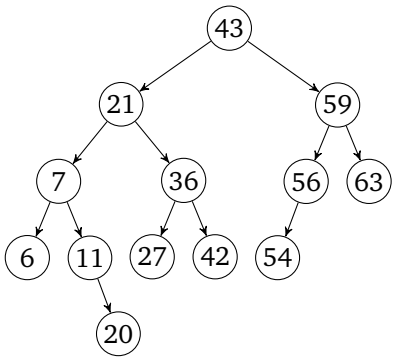


(b)

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

Solution:

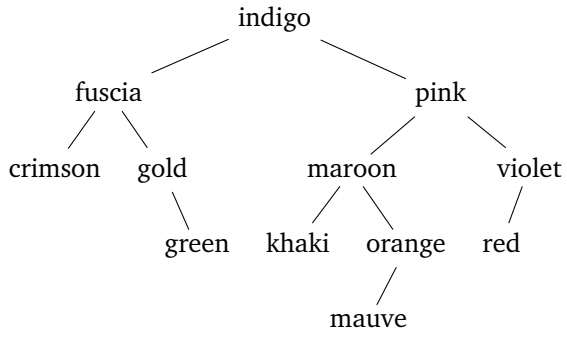
Note: The Section slides have a step-by-step walkthough of this one!



(c)

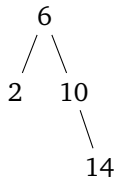
{“indigo”, “fuscia”, “pink”, “goldenrod”, “violet”, “khaki”, “red”, “orange”, “maroon”, “crimson”, “green”, “mauve”}

Solution:



3. 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).

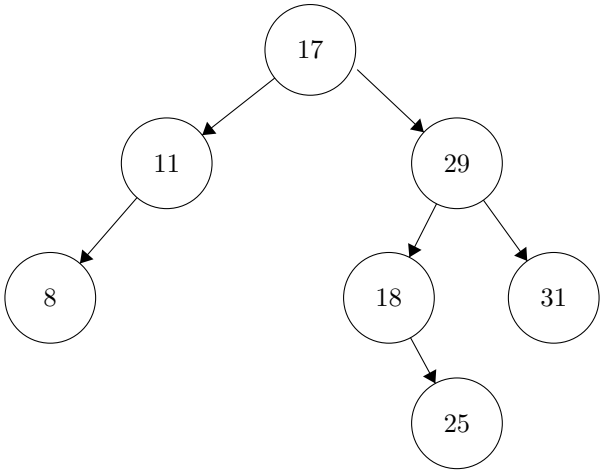
(c) No rotation

Solution:

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

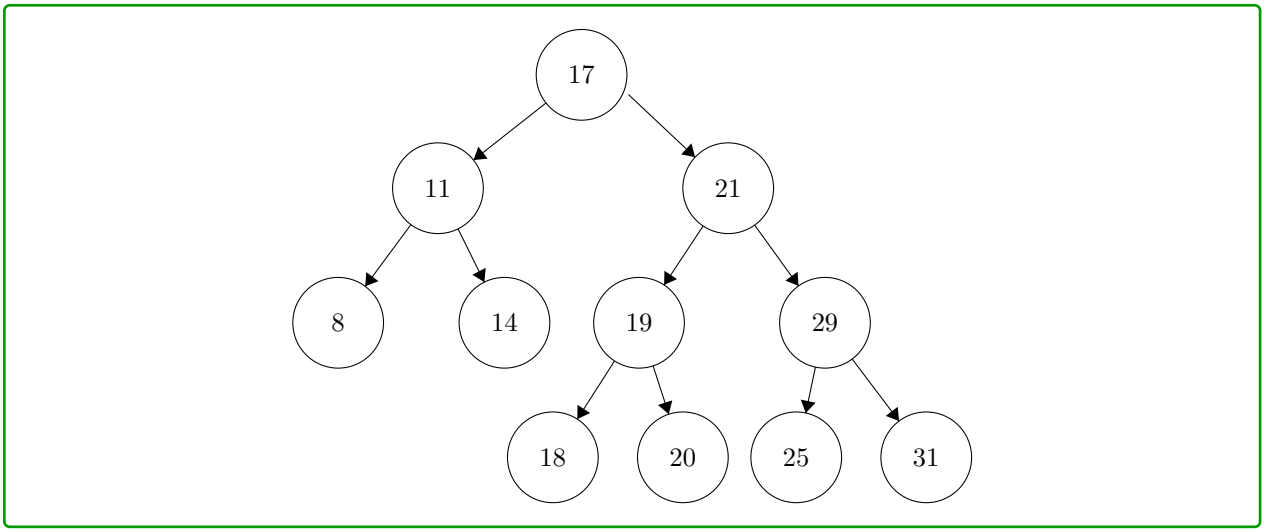
4. Inserting keys and computing statistics

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?

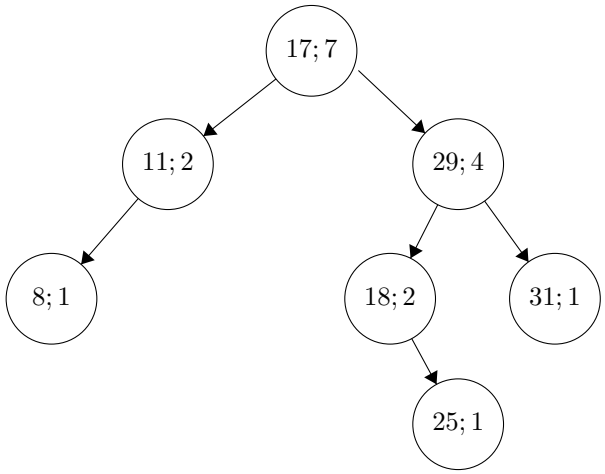
Solution:

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.

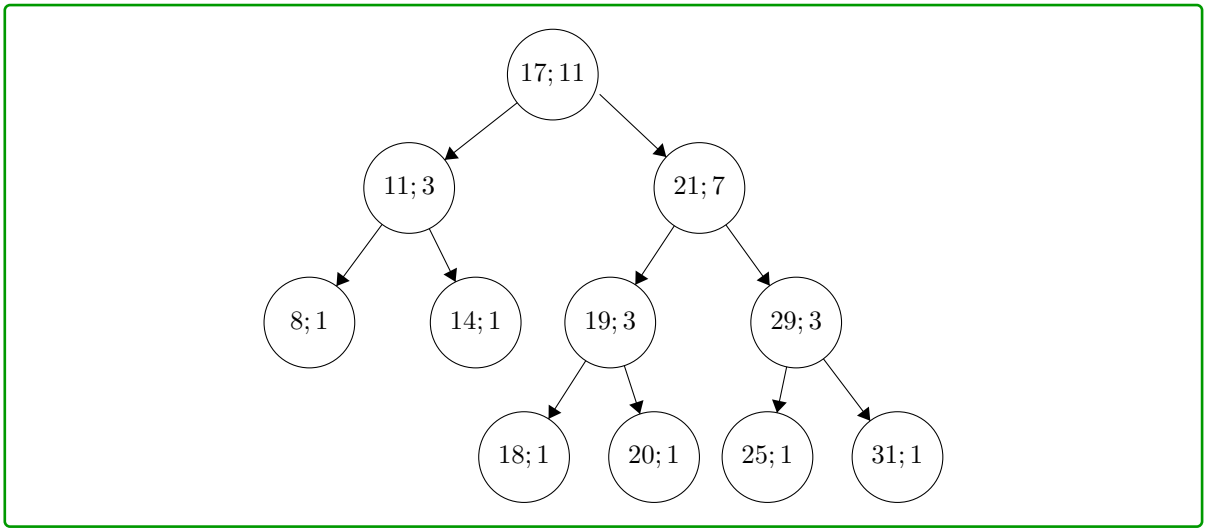
If we were to simply store n integers in an array, one way to compute the median element would be to first sort the array and then look up the element at the $\lceil n/2 \rceil$-th position in the sorted array. This procedure has a runtime of $O(n \log n)$, even when we use a clever sorting algorithm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

5. 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.

6. 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.

Heap Problems

7. Ternary Heaps

Consider the following sequence of numbers:

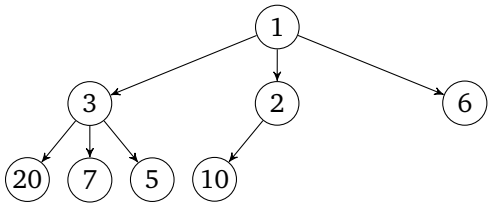
5, 20, 10, 6, 7, 3, 1, 2

(a) Insert these numbers into a min-heap where each node has up to *three* children, instead of two.

(So, instead of inserting into a binary heap, we're inserting into a ternary heap.)

Draw out the tree representation of your completed ternary heap.

Solution:



(b) Draw out the array representation of the above tree. In your array representation, you should start at index 0 (not index 1).

Solution:

1, 3, 2, 6, 20, 7, 5, 10

(c) Given a node at index i, write a formula to find the index of the parent.

Solution:

$$\text{parent}(i) = \left\lfloor \frac{i-1}{3} \right\rfloor$$

(d) Given a node at index i, write a formula to find the j-th child. Assume that $0 \leq j < 3$.

Solution:

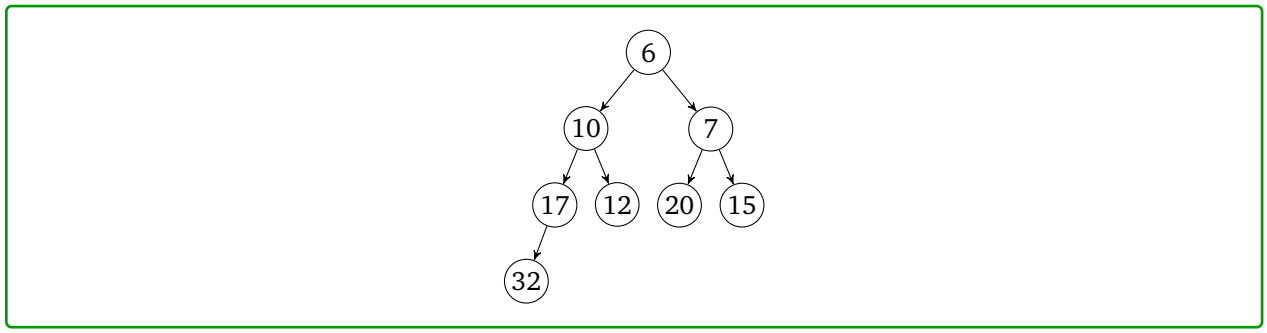
$$\text{child}(i, j) = 3i + j + 1$$

8. Heaps – More Basics

(a) Insert the following sequence of numbers into a *min heap*:

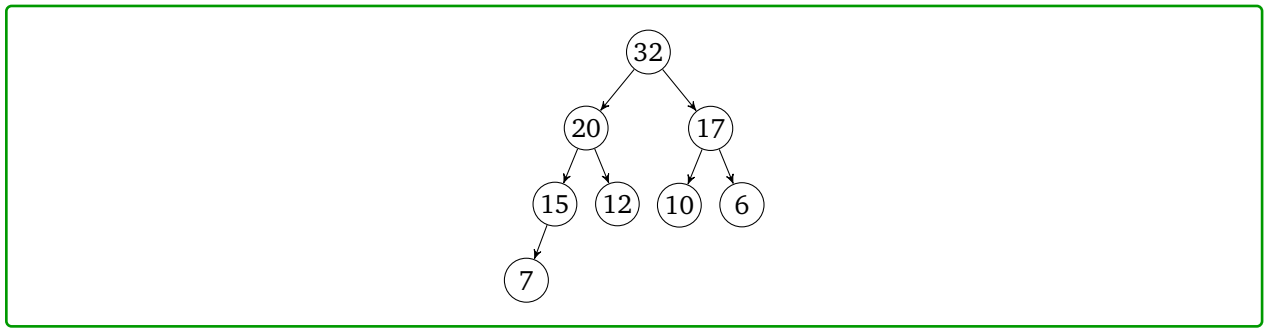
[10, 7, 15, 17, 12, 20, 6, 32]

Solution:



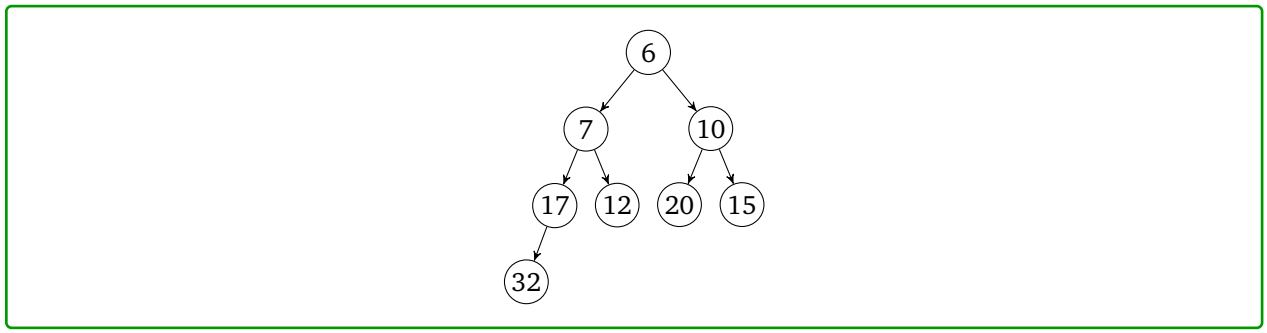
(b) Now, insert the same values into a *max heap*.

Solution:



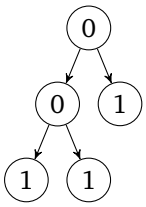
(c) Now, insert the same values into a *min heap*, but use Floyd's `buildHeap` algorithm.

Solution:



(d) Insert 1, 0, 1, 1, 0 into a *min heap*.

Solution:



(e) Call `removeMin` on the min heap stored as the following array: $[2, 5, 7, 8, 10, 9]$ **Solution:**

$[5, 8, 7, 9, 10]$
Credit: https://medium.com/@randerson112358/min-heap-deletion-step-by-step-1e05ff9d3932

9. Sorting and Reversing (with Heaps)

(a) Suppose you have an array representation of a heap. Must the array be sorted? **Solution:**

No, $[1, 2, 5, 4, 3]$ is a valid min-heap, but it isn't sorted.

(b) Suppose you have a sorted array (in increasing order). Must it be the array representation of a valid min-heap? **Solution:**

Yes! Every node appears in the array before its children, so the heap property is satisfied.

(c) You have an array representation of a min-heap. If you reverse the array, does it become an array representation of a max-heap? **Solution:**

No. For example, $[1, 2, 4, 3]$ is a valid min-heap, but if reversed it won't be a valid max-heap, because the maximum element won't appear first.

(d) Describe the most efficient algorithm you can think of to convert the array representation of a min-heap into a max-heap. What is its running time? **Solution:**

You already know an algorithm – just use `buildHeap` (with percolate modified to work for a max-heap instead of a min-heap). The running time is $\mathcal{O}(n)$.

10. Project Prep: Contains

You just finished implementing your heap of `ints` when your boss tells you to add a new method called `contains`. Your solution should not, in general, examine every element in the heap(do it recursively!)

```
public class DankHeap {
    // NOTE: Data starts at index 0!
    private int[] heapArray;
    private int heapSize;

    // Other heap methods here....

    /**
     * examine whether element k exists in the heap
     * @param int k, the element to find.
     * @return true if found, false otherwise
     */
    public boolean contains(int k) {
        // TODO!
    }
}
```

(a) How efficient do you think you can make this method? **Solution:**

> The best you can do in the worst case is $\mathcal{O}(n)$ time. If you start at the top (unlike a binary search tree) the node of priority k could be in either subtree, so you might have to check both. Even if in general we might not need to examine every node, in the worst case, this might lead us to check every node.

(b) Write code for `contains`. Remember that `heapArray` starts at index 0! **Solution:**

```
private boolean contains(int k){
    if(k < heapArray[0]){ //if k is less than the minimum value
        return false;
    }else if (heapSize == 0){
        return false;
    }
    return containsHelper(k, 0);
}
private boolean containsHelper(int k, int index){
    if(index >= heapSize){ //if the index is larger than the heap's size
        return false;
    }
    if(heapArray[index] == k){
        return true;
    }else if(heapArray[index] < k){
        return containsHelper(k, index * 2 + 1) || containsHelper(k, index * 2 + 2);
    }else{
        return false;
    }
}
```

11. Challenge: Debugging Heaps of Problems

For this problem, we will consider a hypothetical hash table that uses linear probing and implements the `IDictionary` interface. Specifically, we will focus on analyzing and testing one potential implementation of the `remove` method.

(a) Come up with at least 4 different test cases to test this `remove(...)` method. For each test case, describe what the expected outcome is (assuming the method is implemented correctly).

Try and construct test cases that check if the `remove(...)` method is correctly using the key's hash code. (You may assume that you can construct custom key objects that let you customize the behavior of the `equals(...)` and `hashCode()` method.)

Solution:

Some examples of test cases:

- If the dictionary contains null keys, or if we pass in a null key, everything should still work correctly.
- If we try removing a key that doesn't exist, the method should throw an exception.
- If we pass in a key with a large hash value, it should mod and stay within the array.
- If we pass in two different keys that happen to have the same hash value, the method should remove the correct key.
- If we pass in a key where we need to probe in the middle of a cluster, removing that item shouldn't disrupt lookup of anything else in that cluster.

  For example, suppose the table's capacity is 10 and we pass in the integer keys 5, 15, 6, 25, 36 in that order. These keys all collide with each other, forming a primary cluster. If we delete the key 15, we should still successfully be able to look up the values corresponding to the other keys.

(b) Now, consider the following (buggy) implementation of the `remove(...)` method. List all the bugs you can find.

```
public class LinearProbingDictionary<K, V> implements IDictionary<K, V> {
    // Field invariants:
    //
    // 1. Empty, unused slots are null
    // 2. Slots that are actually being used contain an instance of a Pair object

    private Pair<K, V>[] array;

    // ...snip...

    public V remove(K key) {
        int index = key.hashCode();

        while ((this.array[index] != null) && !this.array[index].key.equals(key)) {
            index = (index + 1) % this.array.length;
        }

        if (this.array[index] == null) {
            throw new NoSuchKeyException();
        }
        V returnValue = this.array[index].value;
        this.array[index] = null;
        return returnValue;
    }
}
```

Solution:

The bugs:

- We don't mod the key's hash code at the start
- This implementation doesn't correctly handle null keys
- If the hash table is full, the while loop will never end
- This implementation does not correctly handle the "clustering" test case described up above.

  If we insert 5, 15, 6, 25, and 36 then try deleting 15, future lookups to 6, 25, and 36 will all fail.

Note: The first two bugs are, relatively speaking, trivial ones with easy fixes. The middle bug is not trivial, but we have seen many examples of how to fix this. The last bug is the most critical one and will require some thought to detect and fix.

(c) Briefly describe how you would fix these bug(s).

Solution:

- Mod the key's hash code with the array length at the start.
- Handle null keys in basically the same way we handled them in `ArrayDictionary`
- There should be a size field, with `ensureCapacity()` functionality.
- Ultimately, the problem with the "clustering" bug stems from the fact that breaking a primary cluster into two in any way will inevitably end up disrupting future lookups.

  This means that simply setting the element we want to remove to null is not a viable solution. Here are many different ways we can try and fix this issue, but here are just a few different ideas with some analysis:

  - One potential idea is to "shift" over all the elements on the right over one space to close the gap to try and keep the cluster together. However, this solution also fails.

    Consider an internal array of capacity 10. Now, suppose we try inserting keys with the hash-codes 5, 15, 7. If we remove 15 and shift the "7" over, any future lookups to 7 will end up landing on a null node and fail.

  - Rather then trying to "shift" the entire cluster over, what if we could instead just try and find a single key that could fill the gap. We can search through the cluster and try and find the very last key that, if rehashed, would end up occupying this new slot.

    If no key in the cluster would rehash to the now open slot, we can conclude it's ok to leave it null.

    This would potentially be expensive if the cluster is large, but avoids the issue with the previous solution.

  - Another common solution would be to use lazy deletion. Rather then trying to "fill" the hole, we instead modify each `Pair` object so it contains a third field named `isDeleted`.

    Now, rather then nulling that array entry, we just set that field to true and modify all of our other methods to ignore pairs that have this special flag set. When rehashing, we don't copy over these "ghost" pairs.

    This helps us keep our delete method relatively efficient, since all we need to do is to toggle a flag.

    However, this approach also does complicate the runtime analysis of our other methods (the load factor is no longer as straightforward, for example).