Section 05: Solutions

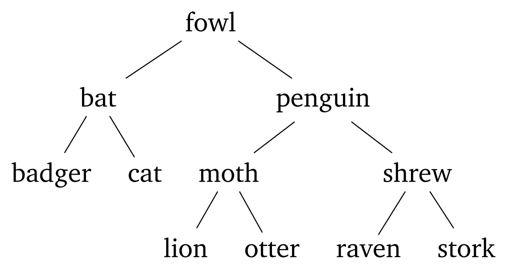
1. Constructing AVL trees

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:

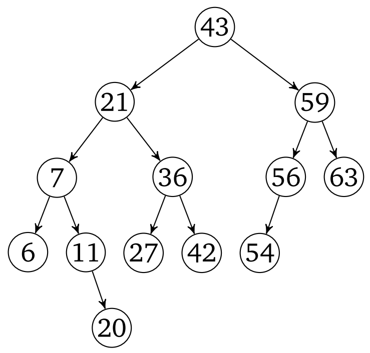


(b)

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

Solution:

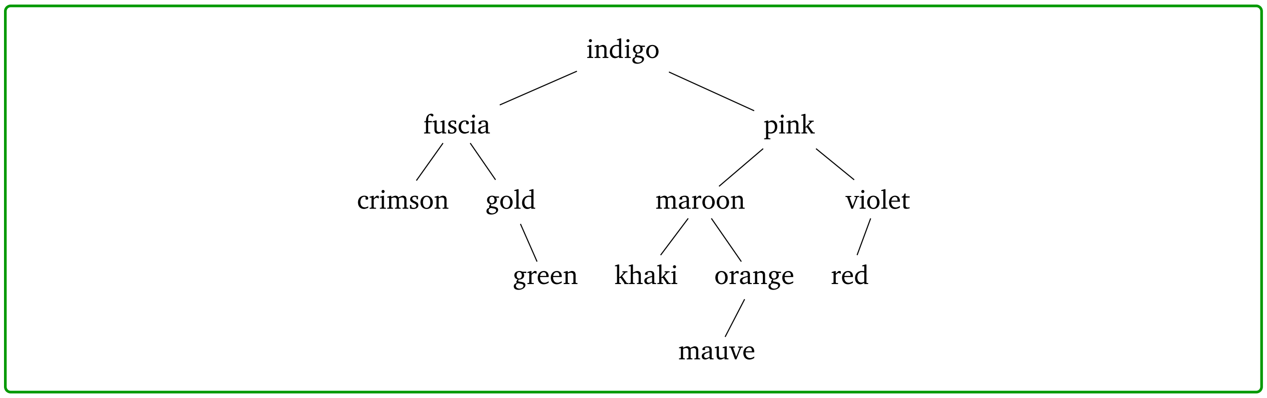
Note: The Section slides have a step-by-step walkthough of this one!



(c)

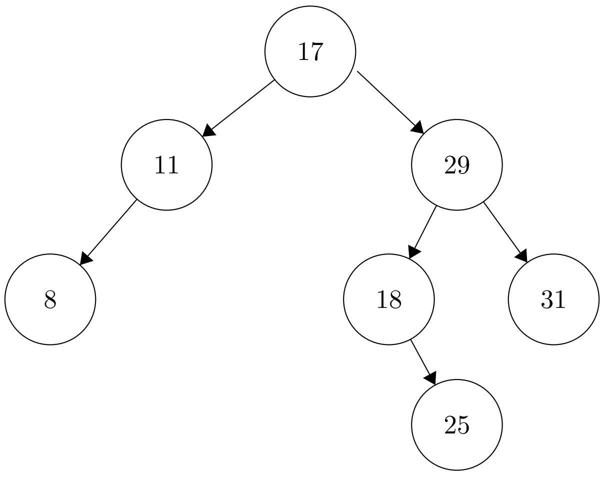
{“indigo”, “fuscia”, “pink”, “goldenrod”, “violet”, “khaki”, “red”, “orange”, “maroon”, “crimson”, “green”, “mauve”}

Solution:



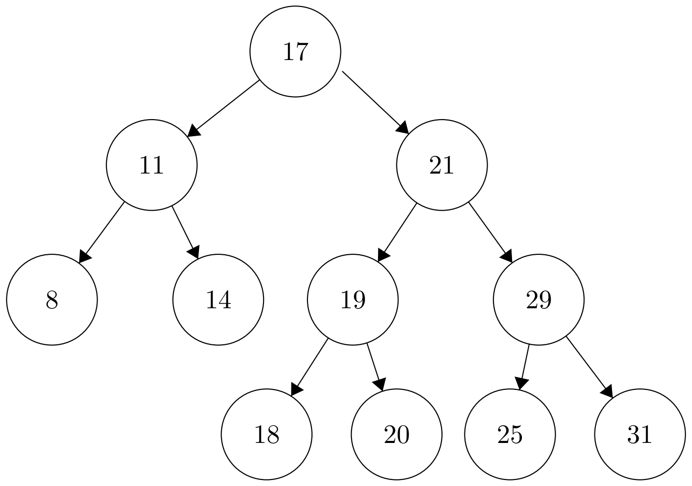
2. Inserting keys

Consider the following AVL tree:



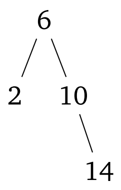
We now add the keys $\{21, 14, 20, 19\}$ (in that order). Show where these keys are added to the AVL tree. Show your intermediate steps.

Solution:



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. True or false?

(a) An insertion in an AVL tree with n nodes requires $\Theta(\log(n))$ rotations.

Solution:

False. Each insertion will require either no rotations, a single rotation, or a double rotation. So, the total number of rotations is in $\Theta(1)$.

(b) A set of numbers are inserted into an empty BST in sorted order and inserted into an empty AVL tree in random order. Listing all elements in sorted order from the BST is $\mathcal{O}(n)$, while listing them in sorted order from the AVL tree is $\mathcal{O}(\log(n))$.

Solution:

False. Although it is true that listing all elements in sorted order from a BST is $\mathcal{O}(n)$, the statement as a whole is false because an AVL tree traversal is also $\mathcal{O}(n)$, since we still have to look at every node once to traverse the tree.

(c) If items are inserted into an empty BST in sorted order, then the BST's `get()` is just as asymptotically efficient as an AVL tree whose elements were inserted in unsorted order.

Solution:

False. If items are inserted into a BST in sorted order, it produces a linked list.

In that case, `get()` would take $\mathcal{O}(n)$ time.

(d) An AVL tree will always do a maximum of two rotations in an insert.

Solution:

True. For an intuition on why this is true, notice that an insertion causes an imbalance because the problem node has one subtree of height k, and the other subtree had height $k+1$, and the insertion occured on the subtree with height $k+1$ to make it height $k+2$.

Then, our rotation will rebalance the tree rooted from the problem node's position such that each subtree is height $k+1$. But since the tree prior to insertion was balanced when the problem node had height $k+2$, and the problem node after rotation still has height $k+2$, and the problem node is also now balanced, the tree must now be completely balanced.

(We know that rotations do not introduce more imbalances below them, since they are a method of fixing imbalances.)

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.

7. Ternary Heaps

Consider the following sequence of numbers:

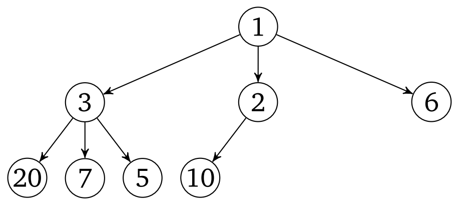
5, 20, 10, 6, 7, 3, 1, 2

(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:

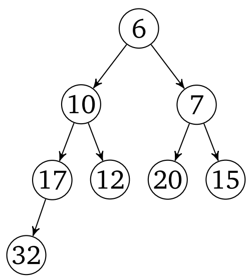
$$\text{child}(i, j) = 3i + j + 1$$

8. Heaps – More Basics

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

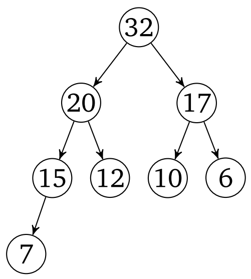
[10, 7, 15, 17, 12, 20, 6, 32]

Solution:



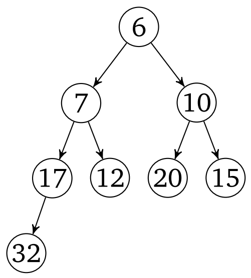
(b) Now, insert the same values into a *max heap*.

Solution:



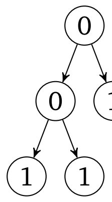
(c) Now, insert the same values into a *min heap*, but use Floyd's `buildHeap` algorithm.

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!)

```java
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:**

```java
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. Running Times

Let's think about the best and worst case for inserting into heaps.

You have elements of priority $1, 2, \ldots, n$. You're going to insert the elements into a min heap one at a time (by calling `insert` not `buildHeap`) in an order that you can control.

(a) Give an insertion order where the total running time of all insertions is $\Theta(n)$. Briefly justify why the total time is $\Theta(n)$. **Solution:**

Insert in increasing order (i.e. $1, 2, 3, ..., n$). For each insertion, it is the new largest element in the heap, so `percolateUp` only needs to do one comparison and no swaps. Since we only need to do those (constant) operations at each `insert`, we do $n \cdot \Theta(1) = \Theta(n)$ operations.

(b) Give an insertion order where the total running time of all insertions is $\Theta(n \log n)$. **Solution:**

Insert in decreasing order. First let's show that this order requires at most $\mathcal{O}(n \log n)$ operations – we have n insertions, each takes at most $\mathcal{O}(\text{height})$ operations. The heap is always height at most $\mathcal{O}(\log n)$, so the total is $\mathcal{O}(n \log n)$.

Now let's show the number of operations is at least $\Omega(n \log n)$. For each insertion, the new element is the new smalliest thing in the heap, so `percolateUp` needs to swap it to the top. For the last $n/2$ elements, the heap is height $\Omega(\log n/2) = \Omega(\log n)$, so there are $\Omega(\log n)$ operations for each of the last $n/2$ insertions. That causes $\Omega(n \log n)$ operations.

Since the number of operations is both $\mathcal{O}(n \log n)$ and $\Omega(n \log n)$ is $\Theta(n \log n)$ by definition.

Remark: it's tempting to say something like "there are n `inserts` and they each have $\Theta(logn)$ operations, but that's not true. The number of operations for the first few inserts is a constant, since the tree isn't that tall yet.

12. Design Decisions

Finally! You've garnered a coveted interview for an internship at Kelp. Determine which data structure(s) are best suited for the scenarios below in terms of **typical** performance, taking into account the specific types of inputs listed in each problem. Your descriptions of the data structure(s) chosen and your algorithm should be brief but sufficiently detailed so that the runtime is unambiguous. Give a simplified, tight Θ bound for the **worstcase** runtime of your solution.

(a) Someone has loaded all the reviews stored on Kelp into a text document of n words. Print out the number of occurrences of each unique word. You **do** need to count the time required to build your data structure, but don't worry about resizing for any of the data structures you pick **Solution:**

Data Structure: `HashMap<String, Integer>`
Usage: Iterate through the words, maintaining a mapping from word to number of times encountered.
Worst Case Runtime: $\Theta(n^2)$ (But usually much better!)

(b) Answer the first question, but assume now that we care more about optimizing for memory usage than runtime. **Solution:**

Data Structure: `ArrayMap<String, Integer>`
Usage: Same as previous question.
Worst Case Runtime: $\Theta\left(n^2\right)$

(c) Answer the first question, but assume now that we also want the ability to print the unique words in alphabetical order. **Solution:**

Data Structure: `AVLMap<String, Integer>`
Usage: Same as previous question. Compare keys alphabetically, and do an in-order traversal for the extra feature.
Worst Case Runtime: $\Theta\left(n \cdot log(n)\right)$

(d) Kelp has a collection of n reviews. Each review has a unique date, author name, number of stars and the text contents of the review. We want to query the number of reviews within a certain datetime range. You don't have to count the time required to construct your data structure this time. **Solution:**

Data Structure: `ArrayList<Review>`
Usage: Build an `ArrayList` sorted on the date of the review. At query time, binary search for the indices corresponding to the endpoints: the difference is the number of reviews in the range.
Worst Case Runtime: $\Theta\left(log(n)\right)$

(e) Kelp is now trying to get into the field of visual computing. Your boss gives you n images of size 256x256, each represented as an `int[][]` array, that you'll need to store in a collection. Each image has associated with it a saturation value, which can be calculated from the pixels. Support the following three operations: `add(int[][] img)`, `getAllImgWithSaturation(int saturation)`, and `remove(int[][] img)`. When providing worst-case runtimes for each operation, don't worry about resizing for any of the data structures you pick, or the time required to construct those data structures. **Solution:**

Data Structure: `HashMap< Integer, HashSet< ImageContainer > >`
Usage: Maintain a mapping from saturation to a set of images with that saturation for easy `add` and `get`. Construct a helper class, `ImageContainer`, that computes the hashcode of the image and stores it. `remove` computes the saturation and then attempts to remove the image from the `HashSet`.
Worst Case Runtime: $\Theta\left(n\right)$

13. Challenge: Recurrences and Heaps

Suppose we have a min heap implemented as a tree, based on the following classes:

```
class HeapNode {
    HeapNode left;
    HeapNode right;
    int priority;

    // constructors and methods omitted.
}

class Heap {
    HeapNode root;
    int size;

    // constructors and methods omitted.
}
```

You just finished implementing your min heap and want to test it, so you write the following code to test whether the heap property is satisfied.

```
boolean verify(Heap h) {
    return verifyHelper(h.root);
}

boolean verifyHelper(HeapNode curr) {
    if (curr == null)
        return true;
    if (curr.left != null && curr.priority > curr.left.priority)
        return false;
    if (curr.right != null && curr.priority > curr.right.priority)
        return false;
    return verifyHelper(curr.left) && verifyHelper(curr.right);
}
```

In this problem, we will use a recurrence to analyze the worst-case running time of `verify`.

(a) Write a recurrence to describe the worst-case running time of the function above. **Hint:** our recurrences need an input integer, use the height of the subtree rooted at `curr`.

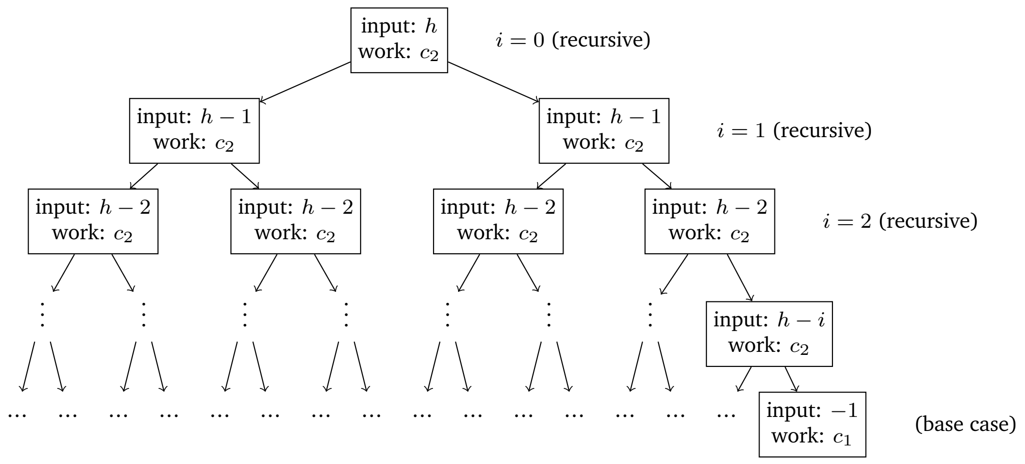
Solution:

$$T(h) = \begin{cases} c_1 & \text{if } h = -1 \\ 2T(h-1) + c_2 & \text{otherwise} \end{cases}$$

Instead of writing explicit numbers, we've written the recurrence with c_1, c_2 to represent those constants. You will get the same big-$\mathcal{O}$ at the end regardless of what actual numbers you plug in there. Notice that even when a node doesn't exist, we still make a recursive call and do constant work to realize we can stop recursing, so our base case really is when $h = -1$. Since we're doing worst case analysis, both of the subtrees could have $h - 1$ (i.e. the heap has all possible nodes at every level)

(b) Find an expression (using summations but no recursion) to describe the running time using the tree method. Leave the overall height of the tree h as a variable in your expression.

Solution:



There are 2^i nodes at the ith level. Each recursive node does c_2 work, so the total work on the ith recursive level is $2^i c_2$. The last recursive level is at $i = h$ (where the input is $h - i = h - h = 0$), so the total recursive work is $\sum_{i=0}^{h} 2^i c_2$. Similarly, the total base case work is $2^{h+1}c_1$. Thus

$$T(n) = 2^{h+1}c_1 + \sum_{i=0}^{h} 2^i c_2$$

(c) Simplify to a closed form.

Solution:

$$\begin{aligned}
2^{h+1}c_1 + \sum_{i=0}^{h} 2^i c_2 &= 2^{h+1}c_1 + c_2 \sum_{i=0}^{h} 2^i \\
&= 2^{h+1}c_1 + c_2 \frac{2^{h+1}-1}{2-1} \\
&= 2^{h+1}c_1 + c_2\left(2^{h+1}-1\right) \\
&= (c_1+c_2)2^{h+1} - c_2
\end{aligned}$$

(d) If a complete tree has height h, how many nodes could it have? Use this to determine a formula for the height of a complete tree on n nodes.

Solution:

A complete tree of height h has h completely filled rows, and one partially filled row. The number of nodes in the first h rows is: $\sum_{i=0}^{h-1} 2^i = 2^h - 1$ The final row has between 1 and 2^h nodes, so the total number of nodes is between 2^h and $2^{h+1} - 1$ nodes.

So if we take the $\log_2$ of the number of nodes, we'll get a number between h and $h + 1$, thus to get the height exactly, we should find:

$$\lfloor \log_2 n \rfloor$$

(e) Use the formula from the last part to find the big-$\mathcal{O}$ of the `verify`.

Solution:

Combining the last two parts, we have a formula of:

$$(c_1 + c_2)2^{\lfloor \log_2 n \rfloor + 1} - c_2$$

We'd like to eventually cancel the 2 and the exponent of $\log_2 n$. Let's make that easier by making the $+1$ in the exponent what it really is – multiplying the expression by 2.

$$(c_1 + c_2)2 \cdot 2^{\lfloor \log_2 n \rfloor} - c_2$$

Can we use that exponents and logs are inverses to cancel? Not if we want an exact formula; but all we care about is the big-$\mathcal{O}$. Getting rid of the floor will at most increase the exponent by 1, which is just multiplying that expression by 2, so we are just changing a constant factor and can make the substitution:

$$2(c_1 + c_2)2^{\lfloor \log_2 n \rfloor} - c_2 \approx 2(c_1 + c_2)2^{\log_2 n} - c_2 = 2(c_1 + c_2)n - c_2$$

The expression is now clearly $\mathcal{O}(n)$.

14. 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).

15. Challenge: Design

Imagine a database containing information about all trains leaving the Washington Union station on Monday. Each train is assigned a departure time, a destination, and a unique 8-digit train ID number.

What data structures you would use to solve each of the following scenarios. Depending on scenario, you may need to either (a) use multiple data structures or (b) modify the implementation of some data structure.

Justify your choice.

(a) Suppose the schedule contains 200 trains with 52 destinations. You want to easily list out the trains by destination.

Solution:

One solution would be to use a dictionary where the keys are the destination, and the value is a list of corresponding chains. Any dictionary implementation would work.

Alternatively, we could modify a separate chaining hash table so it has a capacity of exactly 52 and does not perform resizing. If we then make sure each destination maps to a unique integer from 0 to 51 and hash each train based on this number, we could fill the table and simply print out the contents of each bucket.

A third solution would be to use a BST or AVL tree and have each train be compared first based on destination then based on their other attributes. Once we insert the trains into a tree, we can print out the trains sorted by destination by doing an in-order traversal.

(b) In the question above, trains were listed by destination. Now, trains with the same destination should further be sorted by departure time.

Solution:

Regardless of which solution we modify, we would first need to ensure that the train objects are compared first by destination, and second by departure time.

We can modify our first solution by having the dictionary use a sorted set for the value, instead of a list. (The sorted set would be implemented using a BST or an AVL tree).

We can modify our second solution in a similar way by using specifically a BST or an AVL tree as the bucket type.

Our third solution actually requires no modification: if the trains are now compared first by destination and second by departure time, the AVL and BST tree's iterator will naturally print out the trains in the desired order.

(c) A train station wants to create a digital kiosk. The kiosk should be able to efficiently and frequently complete look-ups by train ID number so visitors can purchase tickets or track the location of a train. The kiosk should also be able to list out all the train IDs in ascending order, for visitors who do not know their train ID.

Note that the database of trains is not updated often, so the removal and additions of new trains happen infrequently (aside from when first populating your chosen DS with trains).

Solution:

Here, we would use a dictionary mapping the train ID to the train object.

We would want to use either an AVL tree or a BST, since we can list out the trains in sorted order based on the ID.

Note that while the AVL tree theoretically is the better solution according to asymptotic analysis, since it's guaranteed to have a worst-case runtime of $\mathcal{O}(\log(n))$, a BST would be a reasonable option to investigate as well.

big-O analysis only cares about very large values of n, since we only have 200 trains, big-O analysis might not be the right way to analyze this problem. Even if the binary search tree ends up being degenerate, searching through a linked list of only 200 element is realistically going to be a fast operation.

What's actually best will depend on the libraries you already have written, what hardware you actually run on, and how you want to balance code that will be sustainable if you get more trains vs. code that will be easy to understand and check for bugs right now.