Section 05: Solutions

1. Hash Tables – More Collision Resolution

Suppose you have a hash table of size 10, with initial hash function $h(x) = 2x$. Insert the following keys into the hash table:

$$2, 12, 3, 10, 5, 7$$

Do not resize the hash table while doing these insertions.

(a) Have your hash table use linear probing to resolve collisions. **Solution:**

0	1	2	3	4	5	6	7	8	9
10	5			2	12	3	7		

(b) Have your hash table use quadratic probing to resolve collisions. **Solution:**

0	1	2	3	4	5	6	7	8	9
10	5			2	12	3		7	

(c) Have your hash table use double hashing, with second hash function $f(x) = x + 1$ as the secondary hash function. **Solution:**

0	1	2	3	4	5	6	7	8	9
10		5		2		3	12	7	

2. True or False

(a) An insertion can fail in a hash table using separate chaining when $\lambda \geq 1$. **Solution:**

False. As long as the "buckets" have arbitrary capacity (like all the data structures we've discussed) you can always find a new spot to insert into. It just might not be efficient.

(b) An insertion can fail in a hash table using linear probing when $\lambda = 3/4$. **Solution:**

False. Linear probing will eventually check every spot in the table. Since $\lambda < 1$ there is a spot it will find eventually.

(c) An insertion can fail in a hash table using quadratic probing when $\lambda = 3/4$. **Solution:**

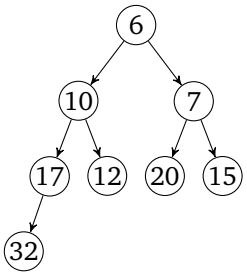
True. Quadratic probing might not check every spot in the hash table, so the insertion could hit only full spots, even if $1/4$ of the spots are open.

3. Heaps – Basics

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

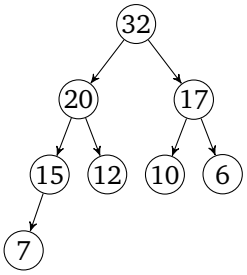
[10, 7, 15, 17, 12, 20, 6, 32]

Solution:



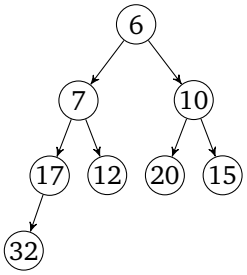
(b) Now, insert the same values into a *max heap*.

Solution:



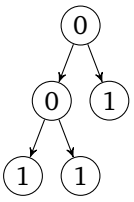
(c) Now, insert the same values into a *min heap*, but use Floyd's `buildHeap` algorithm.

Solution:



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

Solution:



(e) Call `removeMin` three times on the min heap stored as the following array: $[1, 5, 10, 6, 7, 13, 12, 8, 15, 9]$ **Solution:**

$[7, 8, 10, 9, 15, 13, 12]$

4. Food For Thought: Heaps

4.1. 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.

4.2. Sorting and Reversing

(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 $\mathcal{O}(n)$.

5. Food For Thought: Heaps and Dictionaries

You just finished implementing your heap when your boss tells you they need you to add a new method.

```
/**
 * removes the element of priority k from the heap, and restores
 * the heap property
 * @param int k, the priority of the element to remove
 */
public void delete(int k)
```

(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. In the worst case, this leads to checking every node.

(b) Based on your answer to the previous part, should you use a heap to implement a dictionary? **Solution:**

No. By the same logic as the last part, `find` will also take $\mathcal{O}(n)$ operations. AVL trees (in the worst case) and hash tables (in the average case) both have better behavior.

6. Food For Thought: Recurrences

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 closed form. Leave the overall height of the tree h as a variable in your expression. You can use either unrolling or the tree method.

Solution:

This example will use unrolling, tree method will result in exactly the same summation.

$$\begin{aligned} T(h) &= 2T(h-1)+3 \\ &= 2(2T(h-2)+3)+3 \\ &= 2^2T(h-2)+2\cdot 3+3 \\ &= 2^2(2T(h-3)+3)+2\cdot 3+3 \\ &= 2^3T(h-3)+2^2\cdot 3+2\cdot 3+3 \end{aligned}$$

Identifying the pattern, after i steps of unrolling, we have:

$$2^iT(h-i)+\sum_{j=0}^{i-1}2^j\cdot 3$$

We need to plug in $h+1$ to hit the base case, so we get:

$$2^{h+1}\cdot 1+\sum_{j=0}^{h}2^j\cdot 3$$

(c) Simplify to a closed form.

Solution:

$$\begin{aligned} 2^{h+1}+\sum_{j=0}^{h}2^j\cdot 3 &= 2^{h+1}+3\sum_{j=0}^{h}2^j \\ &= 2^{h+1}+3\frac{2^{h+1}-1}{2-1} \\ &= 2^{h+1}+3\left(2^{h+1}-1\right) \\ &= 4\cdot 2^{h+1}-3 \end{aligned}$$

(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: $\sum_{j=0}^{h-1}2^j=2^h-1$ 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:

$$4 \cdot 2^{\lfloor \log_2(n) \rfloor + 1} - 3$$

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.

$$4 \cdot 2 \cdot 2^{\lfloor \log_2(n) \rfloor} - 3$$

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:

$$8 \cdot 2^{\lfloor \log_2(n) \rfloor} - 3 \approx 8 \cdot 2^{\log_2(n)} - 3 = 8n - 3$$

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

7. Debugging

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).