Section 06: Solutions

1. Ternary Heaps

Consider the following sequence of numbers:

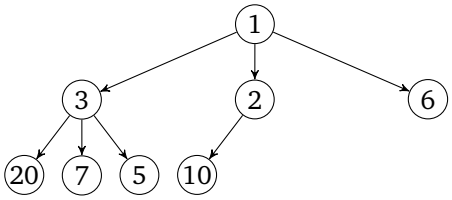
5, 20, 10, 6, 7, 3, 1, 2

(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:

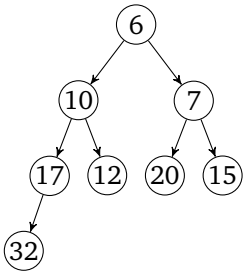
$$\text{child}(i, j) = 3i + j + 1$$

2. Heaps – More Basics

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

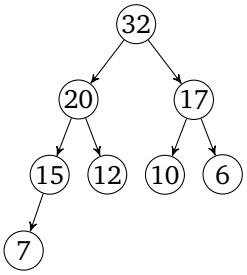
[10, 7, 15, 17, 12, 20, 6, 32]

Solution:



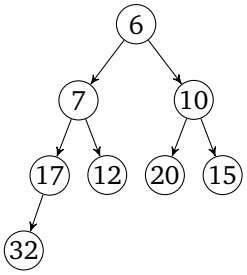
(b) Now, insert the same values into a *max heap*.

Solution:



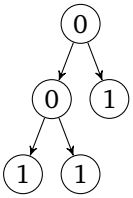
(c) Now, insert the same values into a *min heap*, but use Floyd's `buildHeap` algorithm.

Solution:



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

Solution:



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

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

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

3. 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;
    }
}
```

4. Sorting: Mystery

Consider the following sorting algorithm in pseudocode. Note that, in this case, the upper bound of each for loop is *inclusive*, so they run up to and including $i = A.\text{length} - 1$ and $j = i - 1$.

```
1: function MysterySort(A)
2:     for i = 1 to A.length − 1 do
3:         for j = 0 to i − 1 do
4:             if A[j] ≥ A[i] then
5:                 x = A[i]
6:                 shift every item from j to i − 1 right by one
7:                 A[j] = x
8:                 break
```

(a) Is MysterySort most similar to insertion sort, merge sort, quick sort, or selection sort?

Solution:

MysterySort is most similar to insertion sort. After every iteration of the i for-loop, a new item has been *inserted* somewhere in the sorted section of the array. Unlike selection sort, the new item could be inserted anywhere in the sorted section, not just at the end. This may involve shifting items over to the right to make room.

(b) Is MysterySort a stable sorting algorithm? Why or why not?

Solution:

MysterySort is unstable. This is due to a combination of reasons:

(i) The j for-loop goes over the array from left-to-right;

(ii) The if-condition that inserts the item does not use strict inequality.

This means MysterySort will insert each new item *before* the first item that is greater than or *equal* to it. Since each successive new item came *after* the last one in the original array, this reverses the order of equal items, which means MysterySort is unstable.

Note that insertion sort is normally a stable sorting algorithm. This could be considered a bug in MysterySort, since stability is a desirable property among sorting algorithms. You could make MysterySort stable by changing line 4 to check if $A[j] > A[i]$ instead of $A[j] \geq A[i]$.

(c) What is the best-case runtime (as a tight big-$\mathcal{O}$ bound) for MysterySort? Why is this the best case?

Hint: What happens when MysterySort is given an array that is already sorted?

Solution:

The best-case runtime is $\mathcal{O}(n^2)$.

Consider that for an array A that is already sorted, $A[i] \geq A[j]$ for all $i > j$. This means that the if-condition will never be true (ignoring equal items for now). The j for-loop will then always run i times, which is $\mathcal{O}(n^2)$.

For an array A that is sorted in *reverse* order, $A[i] \leq A[j]$ for all $i > j$. This means that the if-condition will always be true, so the loop runs only once. But when it runs, it shifts every item from j to $i-1$ right by one; since $j = 0$, this takes i operations. So it is also $\mathcal{O}(n^2)$.

Any array will be some combination of these two extremes: the i for-loop will run for k iterations before shifting $i-k$ items over. Either way, it has to visit $k + (i-k) = i$ items every time, so it is always $\mathcal{O}(n^2)$.

Note that insertion sort normally runs in $\mathcal{O}(n)$ time in the best case, which is an already sorted array. You could fix MysterySort by reversing the j for-loop so it goes from $j = i-1$ to 0, and changing the if-condition so it checks if $A[i] \geq A[j]$. This is always true if A is already sorted, so the loop runs once; but this time, it tries to shift over $(i-1) - j = (i-1) - (i-1) = 0$ items! This takes constant time, so MysterySort will run in $\mathcal{O}(n)$.

5. Sorting: Design Decisions

For each of the following scenarios, say which sorting algorithm you think you would use and why. There may be more than one right answer.

(a) Suppose we have an array where we expect the majority of elements to be sorted "almost in order". What would be a good sorting algorithm to use?

Solution:

Merge sort and quick sort are always predictable standbys, but we may be able to get better results if we try using something like insertion sort, which is $\mathcal{O}(n)$ in the best case.

(b) You are writing code to run on the next Mars rover to sort the data gathered each night. (Think about sorting with limited memory and computational power.)

Solution:

Since each memory stick costs thousands (millions?) of dollars to send to Mars, an in-place sort is probably your best bet. Among in-place sorts, heap sort is a great choice (since it is guaranteed $\mathcal{O}(n \log n)$ time and doesn't even use much stack memory). Insertion sort meets memory needs, but wouldn't be fast.

(c) You're writing the backend for the website `SortMyNumbers.com`, which sorts numbers given by users.

Solution:

Do you trust your users? I wouldn't. Because of that, I want a worst-case $\mathcal{O}(n \log n)$ sort. Heap sort or Merge sort would be good choices.

(d) Your artist friend says for a piece she wants to make a computer sort every possible ordering of the numbers $1, 2, \ldots, 15$. Your friend says something special will happen after the last ordering is sorted, and you'd like to see that ASAP.

Solution:

Since you're going to sort all the possible lists, you want to optimize for the average case – Quick sort has the best average case behavior, which makes it a really good choice. Merge sort and heapsort also have average speed of $\mathcal{O}(n \log n)$ but they're usually a little slower on average (depending on the exact implementation).

She didn't appreciate your snarky suggestion to "just print $[1, 2, \ldots, 15]$ $15!$ times." Something about not accurately representing the human struggle.

6. Sorting: Algorithm Practice

(a) Demonstrate how you would use quick sort to sort the following array of integers. Use the first index as the pivot; show each partition and swap.

[6, 3, 2, 5, 1, 7, 4, 0]

Solution:

[solutions omitted]

(b) Show how you would use merge sort to sort the same array of integers.

Solution:

[solutions omitted]

7. Food For Thought: 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`.

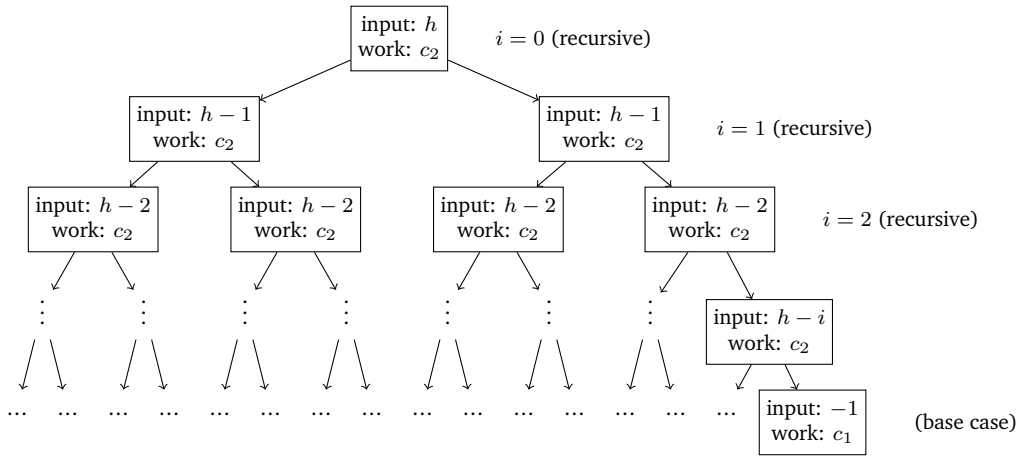
Solution:

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

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 $\mathcal{O}\, n$.

8. 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).

9. Memory: Short Answer

(a) What are the two types of memory locality?

Solution:

Spatial locality is memory that is physically close together in addresses. Temporal locality is the assumption that pages recently accessed will be accessed again.

(b) Does this more benefit arrays or linked lists?

Solution:

This typically benefits arrays. In Java, array elements are forced to be stored together, enforcing spatial locality. Because the elements are stored together, arrays also benefit from temporal locality when iterating over them.

10. Food For Thought: More Heaps

10.1. 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.