Section 05: Solutions

Heap Problems

1. Ternary Heaps

Consider the following sequence of numbers:

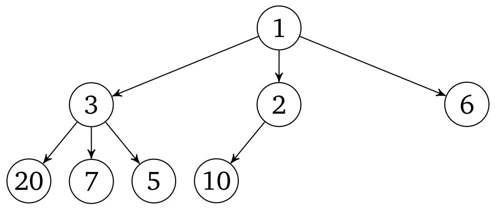
5, 20, 10, 6, 7, 3, 1, 2

(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:

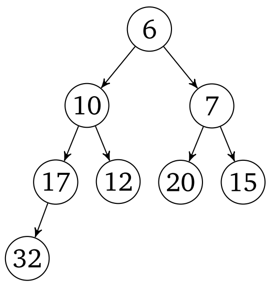
$$\text{child}(i, j) = 3i + j + 1$$

2. Heaps – More Basics

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

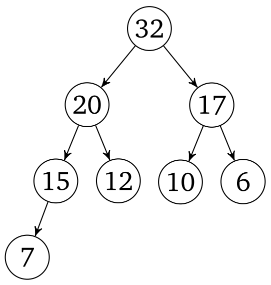
[10, 7, 15, 17, 12, 20, 6, 32]

Solution:



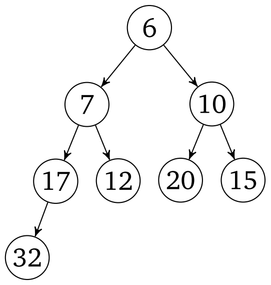
(b) Now, insert the same values into a *max heap*.

Solution:



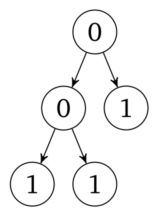
(c) Now, insert the same values into a *min heap*, but use Floyd's `buildHeap` algorithm.

Solution:



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

Solution:



(e) Call `removeMin` 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

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

4. 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;
    }
}
```

5. 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.

```java
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).

Exam 1 Review

6. Selecting ADTs and data structures

For each of the following scenarios, choose:

(a) An ADT: `Stack` or `Queue`

(b) A data structure: `array list` or `linked list with front` or `linked list with front and back`

Justify your choice.

(a) You're designing a tool that checks code to verify all opening brackets, braces, parenthesis, etc... have closing counterparts.

Solution:

We'd use the Stack ADT, because we want to match the *most recent* bracket we've seen first.

Since Stacks push and pop on the same end, there is no reason to use an implementation with two pointers. (We don't need access to the "back" ever.)

Asymptotically (i.e. in big-$\mathcal{O}$ terms), there is no difference between the `LinkedList` with a `front` pointer and the `Array` implementation. In practice, the `Array` implementation is almost certain to be faster due to how computer caches work. Later in the quarter we will use the term *spatial locality* to explain this behavior.

(b) Disneyland has hired you to find a way to improve the processing efficiency of their long lines at attractions. There is no way to forecast how long the lines will be.

Solution:

We'd use the Queue ADT here, because we're dealing with…a line.

The important thing to note here is that if we try to use the implementation of a `LinkedList` with *only a* `front` *pointer*, either *add* or *next* will be very slow. That is clearly not a good choice.

Arguably, the `LinkedList` implementation with both pointers is better than the array implementation because we will never have to resize it.

(c) A sandwich shop wants to serve customers in the order that they arrived, but also wants to look ahead to know what people have ordered and how many times to maximize efficiency in the kitchen.

Solution:

This is still clearly the Queue ADT, but it's unclear that any of these implementations are a good choice!

One of the cool things about data structures is that if only one isn't good enough, you can use *two*. If we only care about the "normal queue features", then we would probably use the `LinkedList` implementation with one pointer. However, we can **ALSO** simultaneously use a *Map* to store the "number of times a food item appears in the queue".

7. Best case and worst case runtimes

For the following code snippet give the big-Θ bound on the worst case runtime as well the big-Θ bound on the best case runtime, in terms of n the size of the input array.

```java
void print(int[] input) {
    int i = 0;
    while (i < input.length - 1) {
        if (input[i] > input[i + 1]) {
            for (int j = 0; j < input.length; j++) {
                System.out.println("uh I don't think this is sorted plz help");
            }
        } else {
            System.out.println("input[i] <= input[i + 1] is true");
        }
        i++;
    }
}
```

Solution:

worst case: $\Theta(n^2)$ consider if the input is reverse sorted order – for each elemet we'd enter the inner for loop that loops over all n elements.

best case: $\Theta(n)$ consider if the input is already sorted and the check for if `(input[i] > input[i + 1])` is never true. Then the runtime's main contributor is just the outer while loop which will run n times.

8. Big-O, Big-Omega True/False Statements

For each of the statements determine if the statement is true or false. You do not need to justify your answer.

(a) $n^3 + 30n^2 + 300n$ is $\mathcal{O}(n^3)$ **Solution:**

T

(b) $nlog(n)$ is $\mathcal{O}(log(n))$ **Solution:**

F

(c) $n^3 - 3n + 3n^2$ is $\mathcal{O}(n^2)$ **Solution:**

F

(d) 1 is $\Omega(n)$ **Solution:**

F

(e) $.5n^3$ is $\Omega(n^3)$ **Solution:**

T

9. Eyeballing Big-Θ bounds

For each of the following code blocks, what is the worst-case runtime? Give a big-Θ bound. You do not need to justify your answer.

(a)
```
void f1(int n) {
    int i = 1;
    int j;
    while(i < n*n*n*n) {
        j = n;
        while (j > 1) {
            j -= 1;
        }
        i += n;
    }
}
```

Solution:

$\Theta(n^4)$

One thing to note that the while loop has increments of $i+=n$. This causes the outer loop to repeat n^3 times, not n^4 times.

(b)
```
int f2(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            System.out.println("j = " + j);
        }
        for (int k = 0; k < i; k++) {
            System.out.println("k = " + k);
            for (int m = 0; m < 100000; m++) {
                System.out.println("m = " + m);
            }
        }
    }
}
```

Solution:

$\Theta(n^2)$

Notice that the last inner loop repeats a small constant number of times – only 100000 times.

(c)
```
int f3(n) {
    count = 0;
    if (n < 1000) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                for (int k = 0; k < i; k++) {
                    count++;
                }
            }
        }
    } else {
        for (int i = 0; i < n; i++) {
            count++;
        }
    }
    return count;
}
```

Solution:

$\Theta(n)$

Notice that once n is large enough, we always execute the 'else' branch. In asymptotic analysis, we only care about behavior as the input grows large.

(d)
```
void f4(int n) {
    // NOTE: This is your data structure from the first project.
    LinkedDeque<Integer> deque = new LinkedDeque<>();
    for (int i = 0; i < n; i++) {
            if (deque.size() > 20) {
                deque.removeFirst();
            }
            deque.addLast(i);
        }
        for (int i = 0; i < deque.size(); i++) {
            System.out.println(deque.get(i));
        }
    }
```

Solution:

$\Theta(n)$

Note that deque would have a constant size of 20 after the first loop. Since this is a `LinkedDeque`, `addLast` and `removeFirst` would both be $\Theta(1)$.

10. 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.