Section 04: Solutions

Hashing Problems

1. Hash Table Insertion!

(a) Suppose we have a hash table implemented using separate chaining. This hash table has an internal capacity of 10. Its buckets are implemented using a linked list where new elements are appended to the end. Do not worry about resizing.

Show what this hash table internally looks like after inserting the following key-value pairs in the order given using the hash function $h(x) = 4x$:

$$(1, a), (4, b), (2, c), (17, d), (12, e), (9, e), (19, f), (4, g), (8, c), (12, f)$$

Solution:

See Section slides.

(b) Consider the following scenario:

Suppose we have a hash table with an initial capacity of 12. We resize the hash table by doubling the capacity. Suppose we insert integer keys into this table using the hash function $h(x) = 4x$:

Why is this choice of hash function and initial capacity suboptimal? How can we fix it?

Solution:

Notice that the hash function will initially always cause the keys to be hashed to at most one of three spots: 12 is evenly divided by 4.

This means that the likelihood of a key colliding with another one dramatically increases, decreasing performance.

This situation does not improve as we resize, since the hash function will continue to map to only a fourth of the available indices.

We can fix this by either picking a new hash function that's relatively prime to 12 (e.g. $h(x) = 5x$), by picking a different initial table capacity, or by resizing the table using a strategy other then doubling (such as picking the next prime that's roughly double the initial size).

See Section slides for more details.

2. More Hash Table Insertion!

For each problem, insert the given elements into the described hash table. Do not worry about resizing the internal array.

(a) Suppose we have a hash table that uses separate chaining and has an internal capacity of 12. Assume that each bucket is a linked list where new elements are added to the front of the list.

Insert the following elements in the EXACT order given using the hash function $h(x) = 4x$:

0, 4, 7, 1, 2, 3, 6, 11, 16

Solution:

To make the problem easier for ourselves, we first start by computing the hash values and initial indices:

key	hash	index (pre probing)
0	0	0
4	16	4
7	28	4
1	4	4
2	8	8
3	12	0
6	24	0
11	44	8
16	64	4

The state of the internal array will be

$6 \to 3 \to 0$	/	/	/	$16 \to 1 \to 7 \to 4$	/	/	/	$11 \to 2$	/	/	/

(b) Suppose we have a hash table that uses linear probing and has an internal capacity of 13.

Insert the following elements in the EXACT order given using the hash function $h(x) = 3x$:

2, 4, 6, 7, 15, 13, 19

Solution:

Again, we start by forming the table:

key	hash	index (before probing)
2	6	6
4	12	12
6	18	5
7	21	8
15	45	6
13	39	0
19	57	5

Next, we insert each element into the internal array, one-by-one using linear probing to resolve collisions. The state of the internal array will be:

13	/	/	/	/	6	2	15	7	19	/	/	4

(c) Suppose we have a hash table that uses quadratic probing and has an internal capacity of 10.

Insert the following elements in the EXACT order given using the hash function $h(x) = x$:

0, 1, 2, 5, 15, 25, 35

Solution:

The state of the internal array will be:

0	1	2	/	35	5	15	/	/	25

(d) Suppose we have a hash table with an initial capacity of 8 using quadratic probing. We resize the hash table by doubling the capacity.

Suppose we insert the integer keys $2^{20}, 2 \cdot 2^{20}, 3 \cdot 2^{20}, 4 \cdot 2^{20}, \dots$ using the hash function $h(x) = x$.

Describe what the runtime of the dictionary operations will over time as you keep adding these keys to the table.

Solution:

Initially, for the first few keys, the performance of the table will be fairly reasonable.

However, as we insert each key, they will keep colliding with each other: the keys will all initially mod to index 0.

This means that as we keep inserting, each key ends up colliding with every other previously inserted key, causing all of our dictionary operations to take $\mathcal{O}(n)$ time.

However, once we resize enough times, the capacity of our table will be larger then 2^{20}, which means that our keys no longer necessarily map to the same array index. The performance will suddenly improve at that cutoff point then.

3. Even More Hash Table Insertion!

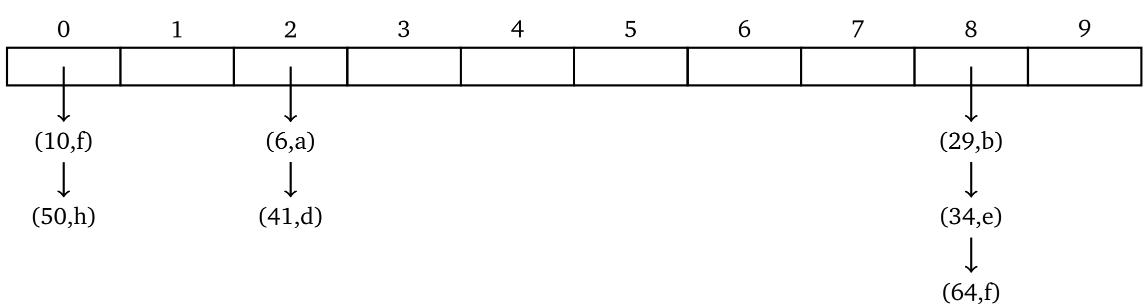
(a) Consider the following key-value pairs.

(6, a), (29, b), (41, d). (34, e), (10, f), (64, g), (50, h)

Suppose each key has a hash function $h(k) = 2k$. So, the key 6 would have a hash code of 12. Insert each key-value pair into the following hash tables and draw what their internal state looks like:

(i) A hash table that uses separate chaining. The table has an internal capacity of 10. Assume each bucket is a linked list, where new pairs are appended to the end. Do not worry about resizing.

Solution:



(ii) A hash table that uses linear probing, with internal capacity 10. Do not worry about resizing.

Solution:

0	1	2	3	4	5	6	7	8	9
(10, f)	(64, g)	(6, a)	(41, d)	(50, h)				(29, b)	(34, e)

(iii) A hash table that uses quadratic probing, with internal capacity 10. Do not worry about resizing.

Solution:

0	1	2	3	4	5	6	7	8	9
(10,f)	(50, h)	(6,a)	(41,d)				(64, g)	(29,b)	(34,e)

4. Hashing and Mutation

For the following problems, assume that:

1. IntList is a list of integers.
2. The hash code of an IntList is the sum of the integers in the list.
3. IntLists are considered equal only if they have the same size and the same values in the same order.
4. FourBucketHashMap uses separate chaining and the new items are added to the back of each bucket.
5. FourBucketHashMap always has four buckets and never resizes.

Consider the following code:

```
FourBucketHashMap<IntList, String> fbhm = new FourBucketHashMap<>();
IntList list1 = IntList.of(1, 2);
fbhm.put(list1, ”dog”);
// Part i
list1.add(3);
// Part ii
```

(a) At Part i (line 4), what will be returned from the following statement?

```
fbhm.get(IntList.of(1, 2));
```

Solution:

“dog”

This will look up the bucket $(1 + 2) \bmod 4 = 3$. In the bucket 3, IntList.of(1, 2) is equivalent to [1, 2], so “dog” which is the stored value is returned.

See Section slides for more details.

(b) At Part II (line 6), what will be returned from the following statements?

`fbhm.get(IntList.of(1, 2));`

`fbhm.get(IntList.of(1, 2, 3));`

Solution:

"`null`"

"`null`"

The first get function will look up the bucket (1 + 2) mod 4 = 3. In the bucket 3, IntList.of(1, 2) is NOT equivalent to [1, 2, 3], so we cannot find the matched key. Hence, return null.

The second get function will loop up the bucket (1 + 2 + 3) mod 4 = 2. Since the bucket 2 is empty, we definitely cannot find the matched key. Hence, return null.

See Section slides for more details.

(c) Is there a problem with the code? If so, explain.

Solution:

Adding 3 into `list1` changes its hash code, causing `list1` to live in the wrong bucket.

See Section slides for more details.

5. Debugging a Hash Table

Suppose we are in the process of implementing a hash map that uses open addressing and quadratic probing and want to implement the `delete` method.

(a) Consider the following implementation of `delete`. List every bug you can find.

Note: You can assume that the given code compiles. Focus on finding run-time bugs, not compile-time bugs.

```java
public class QuadraticProbingHashTable<K, V> {
    private Pair<K, V>[] array;
    private int size;

    private static class Pair<K, V> {
        public K key;
        public V value;
    }

    // Other methods are omitted, but functional.

    /**
     * Deletes the key-value pair associated with the key, and
     * returns the old value.
     *
     * @throws NoSuchKeyException if the key-value pair does not exist in the method.
     */
    public V delete(K key) {
        int index = key.hashCode() % this.array.length;

        int i = 0;
        while (this.array[index] != null && !this.array[index].key.equals(key)) {
```

```java
                i += 1;
                index = (index + i * i) % this.array.length;
            }

            if (this.array[index] == null) {
                throw new NoSuchKeyException("Key-value pair not in dictionary");
            }

            this.array[index] = null;

            return this.array[index].value;
        }
    }
```

Solution:

The full list of all bugs:

(i) If the dictionary contains any null keys, this code will crash. (See the call to `.equals(...)` in the while loop condition.)

(ii) If the key parameter is null, the code will crash. (See the call to `.hashCode(...)` at the top of the method.)

(iii) If the key's hashCode is negative, this code will crash. (We try indexing a negative element).

(iv) We probe the array incorrectly. If s is the initial position we check, we ought to be checking s, $s+1$, $s+4$, $s+9$, $s+16$...

Instead, we check s, $s+1$, $s+5$, $s+14$, $s+30$...

(v) Nulling out the array index will break all subsequent deletes. Suppose we have a collision, and our algorithm ends up checking index locations 0, 1, 5, 14, and 30 respectively.

If we null out index 5, then all subsequent probes starting at index 0 will be unable to find whatever's located at 14 or 30.

(vi) The final return has a null pointer exception – we null out that pair before fetching the value.

(b) Let's suppose the `Pair` array has the following elements (pretend the array fit on one line):

["lily", V_2]	["castle", V_6]	["resource", V_1]	["hard", V_9]	["bathtub", V_0]
["wage", V_4]	["refund", V_7]	["satisfied", V_6]	["spring", V_8]	["spill", V_3]

And, that the following keys have the following hash codes:

Key	Hash Code
"bathtub"	9744
"resource"	4452
"lily"	7410
"spill"	2269
"wage"	8714
"castle"	2900
"satisfied"	9251
"refund"	8105
"spring"	6494
"hard"	9821

What happens when we call `delete` with the following inputs? Be sure write out the resultant array, and to do these method calls *in order*. (**Note**: If a call results in an infinite loop or an error, explain what happened, but don't change the array contents for the next question.)

(i) `delete("lily")`

Solution:

Nothing bad happens:

`null`	[“castle”, V_6]	[“resource”, V_1]	[“hard”, V_9]	[“bathtub”, V_0]
[“wage”, V_4]	[“refund”, V_7]	[“satisfied”, V_6]	[“spring”, V_8]	[“spill”, V_3]

(ii) `delete(“spring”)`

Solution:

We don’t probe correctly in general (see bug above), but we *do* happen to loop around eventually to remove “spring”. This code is inefficient, and won’t always work out like this, but in this case, we managed a successful delete:

`null`	[“castle”, V_6]	[“resource”, V_1]	[“hard”, V_9]	[“bathtub”, V_0]
[“wage”, V_4]	[“refund”, V_7]	[“satisfied”, V_6]	`null`	[“spill”, V_3]

(iii) `delete(“castle”)`

Solution:

We stop after we see `array[0]` is null, and we throw a `NoSuchKeyException`, without continuing to probe.

(iv) `delete(“bananas”)`

Solution:

`NoSuchKeyException`, but that’s what we wanted, so no bugs caught.

(v) `delete(null)`

Solution:

`NullPointerException`. Note, this is *not* what we wanted, because `Pairs` support null keys. This code should have returned a `NoSuchKeyException`.

(c) List four different test cases you would write to test this method. For each test case, be sure to either describe or draw out what the table’s internal fields look like, as well as the expected outcome (assuming the `delete` method was implemented correctly). **Hint**: You may use the inputs previously given to help you identify tests, but it’s up to you to describe what kind of input they are testing generally.

Solution:

Some test cases include:

- Picking a key not present in the dictionary. This should trigger an exception (and not change the size).
- Picking a key present in the dictionary. This should succeed, and return the old value (and decrease the size by 1).
- Inserting and attempting to delete a null key. This should succeed (and decrease the size by 1).

- Deleting a key that forces us to probe a few times. This should succeed (and decrease the size, etc).
- Deleting a key in the middle of some probe sequence. All subsequent calls to delete/get/etc should correctly.
- Using a key with a negative hashcode should behave as expected.

Math Review

6. Tree Method

Find a summation for the total work of the following expressions using the Tree Method.

Hint: Just as a reminder, here are the steps you should go through for **any** Tree Method Problem:

i. Draw the recurrence tree.

ii. What is the size of the **input** to each node at level i? As in class, we call the root level $i = 0$. This means that at $i = 0$, your expression for the input should equal n.

iii. What is the amount of **work** done by each node at the i-th *recursive* level?

iv. What is the total number of nodes at level i? As in class, we call the root level $i = 0$. This means that at $i = 0$, your expression for the total number of nodes should equal 1.

v. What is the total work done across the i-th *recursive* level?

vi. What value of i does the last level of the tree occur at?

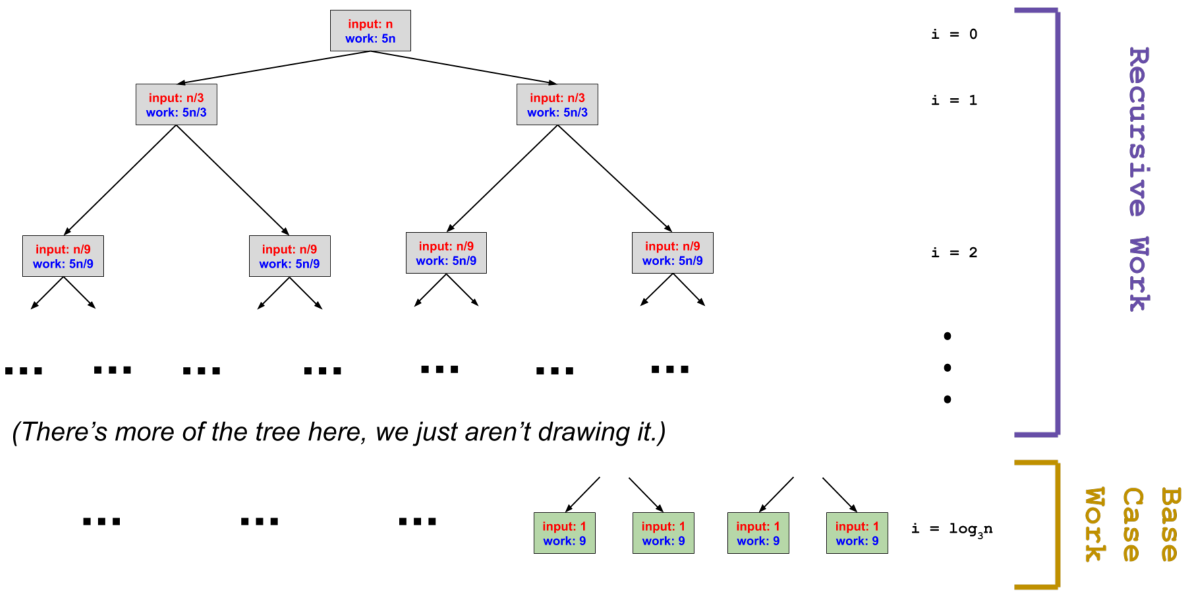
vii. What is the total work done across the base case level of the tree (i.e. the last level)?

viii. Combine your answers from previous parts to get an expression for the total work.

(a) $T(n) = \begin{cases} 2T(n/3) + 5n & \text{if } n > 1 \\ 9 & \text{otherwise} \end{cases}$

Solution:

(i) Here's a drawing of the tree:



(ii) The input size at level i is $n/3^i$ (since we divide the input by 3 at each level).

(iii) The previous answer makes the work in each recursive case node $5 \cdot (n/3^i)$, since each recursive node does five times its input as work.

(iv) The number of nodes at level i is 2^i (since each node has two children).

(v) Multiplying the work per recursive case node by the recursive nodes per level, we get: $5 \cdot \frac{n}{3^i} \cdot 2^i = 5n \cdot \left(\frac{2}{3}\right)^i$.

(vi) The last level of the tree is when $n/3^i = 1$. Solving for i gives us $i = \log_3(n)$.

(vii) Work across the base case level: The number of nodes (from previous parts) is $2^{\log_3(n)}$, and each node does 9 work, so the total work is $9 \cdot 2^{\log_3(n)}$.

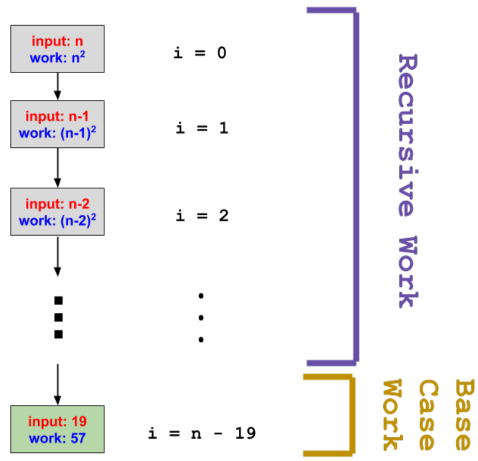
(viii) Summing up work across all recursive levels and then adding in the base case work, we get:

$$\sum_{i=0}^{\log_3(n)-1} 5n \left(\frac{2}{3}\right)^i + 9 \cdot 2^{\log_3(n)}$$

(b) $T(n) = \begin{cases} T(n-1) + n^2 & \text{if } n > 19 \\ 57 & \text{otherwise} \end{cases}$

Solution:

(i) Here's a drawing of the tree:



(ii) The input size at level i is $n - i$ (since we subtract the input by 1 at each level).

(iii) The previous answer makes the work in each recursive case node $(n - i)^2$, since each recursive node does the square of its input as work.

(iv) The number of nodes at any level i is 1 (since each node has a single child).

(v) Multiplying the work per recursive case node by the recursive nodes per level, we get: $1 \cdot (n - i)^2$.

(vi) The last level of the tree is when $n - i = 19$. Solving for i gives us $i = n - 19$.

(vii) The number of nodes in the base case level is 1, and each node does 57 work, so the total work is $1 \cdot 57$.

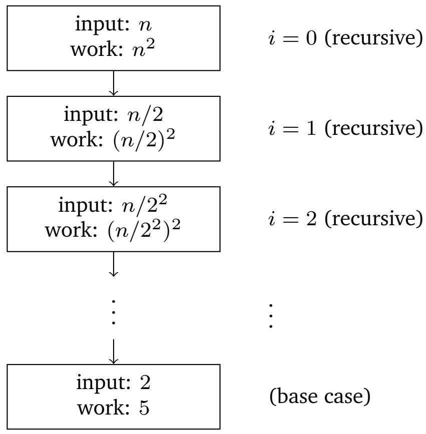
(viii) Summing up work across all recursive levels and then adding in the base case work, we get:

$$57 + \sum_{i=0}^{(n-19)-1} (n - i)^2$$

(c) $T(n) = \begin{cases} T(n/2) + n^2 & \text{if } n \geq 4 \\ 5 & \text{otherwise} \end{cases}$

Solution:

(i)



We assume the input size is a power of 2, so the base case input size is 2 instead of 3.

(ii) The input size is $n/2^i$ since we divide the input by 2 at each level.

(iii) The previous answer makes the work at each recursive node $(n/2^i)^2 = n^2/2^{2i}$, since each recursive node does the square of its input as work.

(iv) There is only 1 node at every level.

(v) The total work at each recursive level is $1 \cdot n^2/2^{2i}$.

(vi) The last level of the tree is when $n/2^i = 2$, so $i = \log_2 n - 1$.

(vii) Since there is only 1 node at the base case level, the total work is $1 \cdot 5$.

(viii)

$$5 + \sum_{i=0}^{\log_2 n - 2} \frac{n^2}{2^{2i}}$$

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. Modeling

Consider the following method. Let n be the integer value of the `n` parameter, and let m be the size of the `LinkedDeque`.

```
    public int mystery(int n, LinkedDeque<Integer> deque) {
        if (n < 7) {
            System.out.println("???");
            int out = 0;
            for (int i = 0; i < n; i++) {
                out += i;
            }
            return out;
        } else {
            System.out.println("???");
            System.out.println("???");
            out = 0;
            // NOTE: Assume LinkedDeque has working, efficient iterator.
            for (int i : deque) {
                out += 1;
                for (int j = 0; j < deque.size(); j++) {
                    System.out.println(deque.get(j));
                }
            }
            return out + 2 * mystery(n - 4, deque) + 3 * mystery(n / 2, deque);
        }
    }
```

Give a recurrence formula for the **worst-case** running time of this code. It's OK to provide a $\mathcal{O}$ for non-recursive terms (for example if the running time is $A(n) = 4A(n/3) + 25n$, you need to get the 4 and the 3 right but you don't have to worry about getting the 25 right). Just show us how you got there.

Solution:

$$T(n, m) = \begin{cases} 1 & \text{when } n < 7 \\ m^3 + T(n-4, m) + T(n/2, m) & \text{otherwise} \end{cases}$$