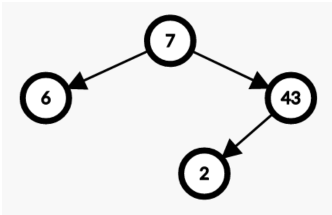
Section 04: Solutions

1. Valid BSTs and AVL Trees

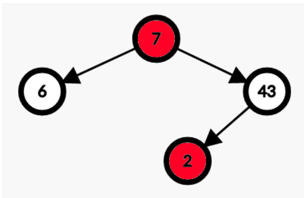
For each of the following trees, state whether the tree is (i) a valid BST and (ii) a valid AVL tree. Justify your answer.

(a)



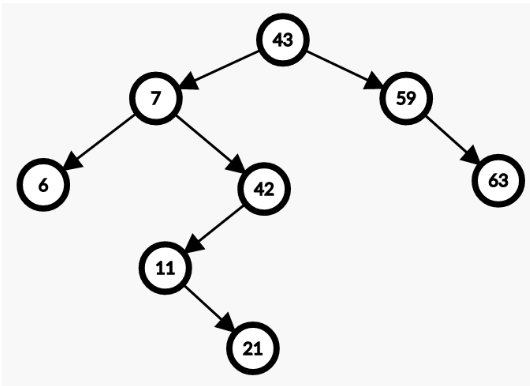
Solution:

This is not a valid BST! The 2 is located in the right sub-tree of 7, which breaks the BST property. Remember that the BST property applies to **every** node in the left and right sub-trees, not just the immediate child!



All AVL trees are BSTs. Because of this, this tree can't be a valid AVL tree either.

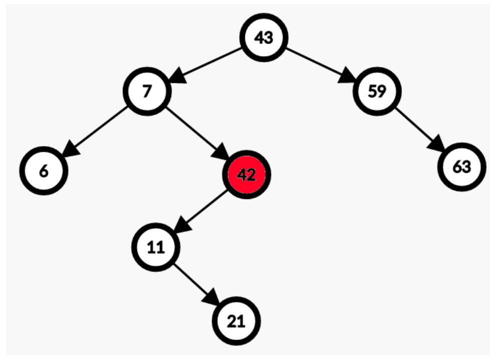
(b)



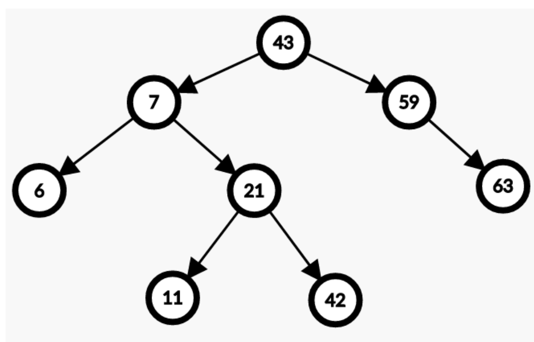
Solution:

This tree is a valid BST! If we check every node, we see that the BST property holds at each of them.

However, this is not a valid AVL tree. We see that some nodes (for example, the 42) violate the balance condition, which is an extra requirement compared to BSTs. Because the heights of 42's left and right sub-trees differ by more than one, this violates the condition.



(c)



Solution:

This tree is a valid BST! If we check every node, we see that the BST property holds at each of them.

This tree is also a valid AVL tree! If we check every node, we see that the balance condition also holds at each of them.

;

2. 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:

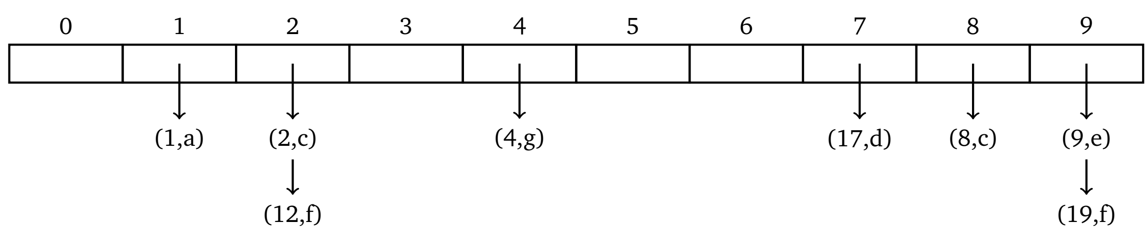
0	1	2	/	35	5	15	/	/	25

(d) Suppose we have a hash table 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 $h(x) = x$:

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

Solution:



3. Evaluating hash functions

Consider the following scenarios.

(a) 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).

(b) 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 $2^{20}, 2 \cdot 2^{20}, 3 \cdot 2^{20}, 4 \cdot 2^{20}, \ldots$ 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.

4. 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)
```java
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)
```java
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 $\Theta(1)$.

5. 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.

6. 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

7. 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?

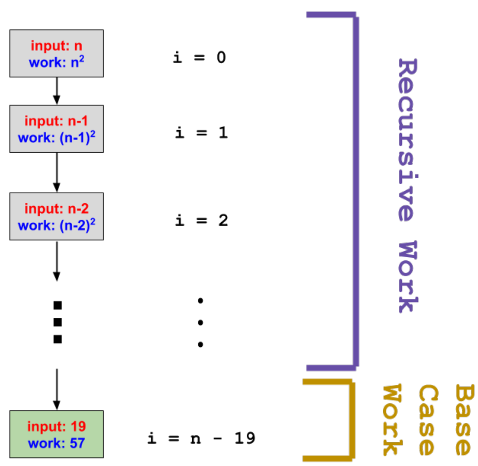
vii. What is the total work done across the base case level of the tree (i.e. the last level)?

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

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

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 $1 \cdot 57$.

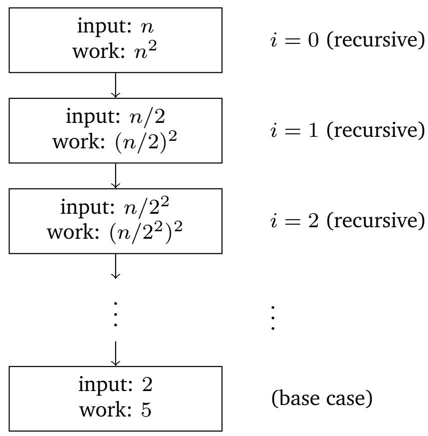
(viii) Summing up work across all recursive levels and then adding in the base case work, we get:

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

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

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 $1 \cdot 5$.

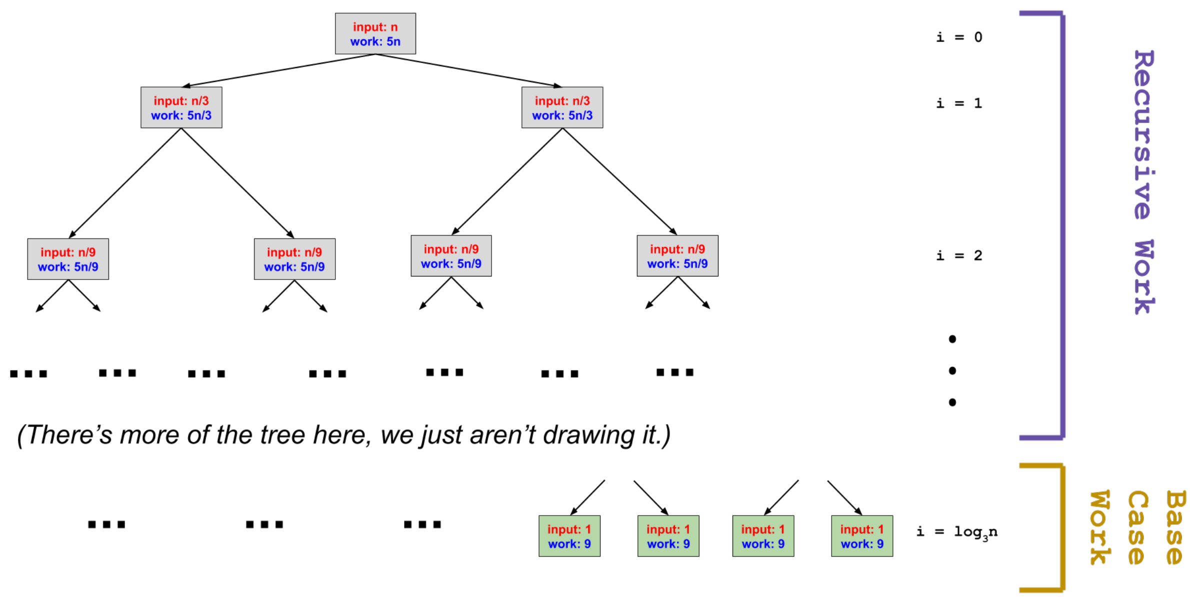
(viii)

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

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

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:

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

8. Modeling

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

```java
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:

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

9. Hash tables

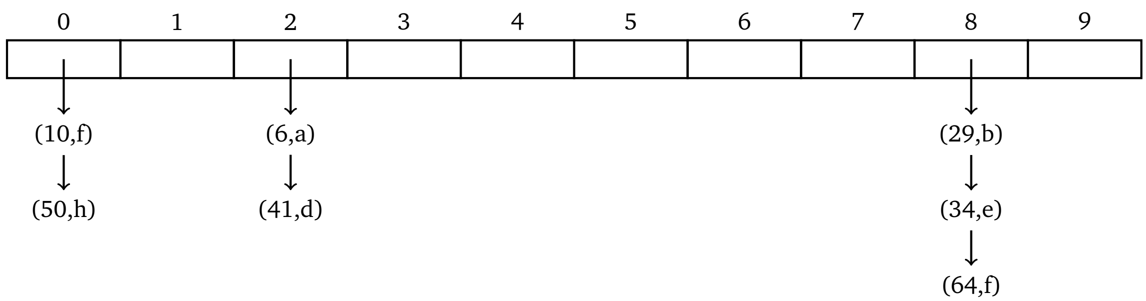
(a) Consider the following key-value pairs.

$$(6, \text{a}), (29, \text{b}), (41, \text{d}). (34, \text{e}), (10, \text{f}), (64, \text{g}), (50, \text{h})$$

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)