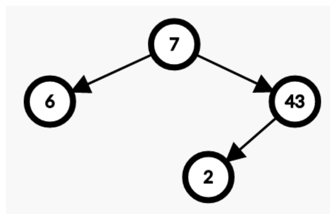
# Section 04: Midterm Review

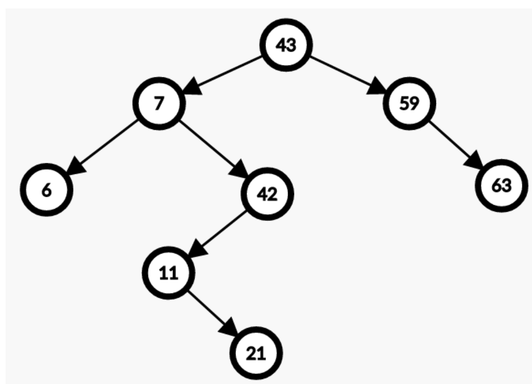
## 1. Valid BSTs and AVL Trees

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

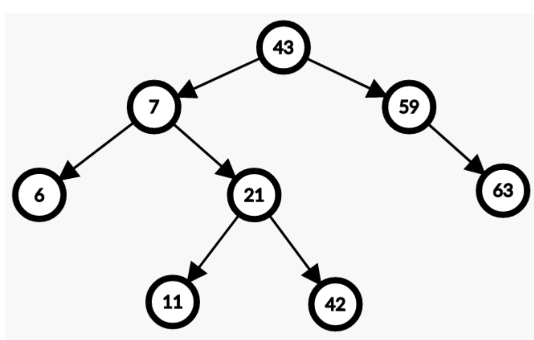
(a)




(b)




(c)



;

## 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

(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

(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

(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 $h(x) = x$:

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

## 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?

(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.

## 4. Eyeballing Big-$\Theta$ bounds

For each of the following code blocks, what is the worst-case runtime? Give a big-$\Theta$ 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;
    }
}
```

(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);
            }
        }
    }
}
```

(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;
}
```

(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));
    }
}
```

## 5. Best case and worst case runtimes

For the following code snippet give the big-$\Theta$ bound on the worst case runtime as well the big-$\Theta$ 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++;
    }
}
```

## 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 $\mathcal{O}(n^3)$

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

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

(d) $1$ is $\Omega(n)$

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

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

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

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

## 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.

```
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.

## 9. Hash tables

(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.

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

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