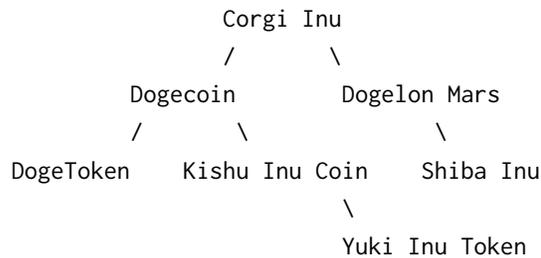


## 1 Coins, Trees, and Dogs?

Tod is obsessed with dog-associated cryptocurrencies. After purchasing an unhealthy volume of volatile assets, Tod decides to use the binary tree below to organize his collection.



- (a) Tod wishes to organize his collection alphabetically. Write out the DFS pre-order, DFS in-order, DFS post-order, and BFS (Level Order) traversals of the following binary tree. Which traversal gives the collection in sorted in alphabetical order?

DFS Pre-order: Corgi Inu, Dogecoin, DogeToken, Kishu Inu Coin, Yuki Inu Token, Dogelon Mars, Shiba Inu

DFS In-order: DogeToken, Dogecoin, Kishu Inu Coin, Yuki Inu Token, Corgi Inu, Dogelon Mars, Shiba Inu

DFS Post-order: DogeToken, Yuki Inu Token, Kishu Inu Coin, Dogecoin, Shiba Inu, Dogelon Mars, Corgi Inu

BFS/Level Order: Corgi Inu, Dogecoin, Dogelon Mars, DogeToken, Kishu Inu Coin, Shiba Inu, Yuki Inu Token

We see that the BFS traversal yields a sorted ordering.

- (b) Tod mistakenly believes that an inorder traversal will yield the collection in alphabetical order. What data structure might Tod have been thinking of? To fix this, draw a tree such that when traversed inorder, it will yield the coins in sorted order.

Tod's tree is not organized according to a true binary search tree, where for every node, all nodes in the tree to the left are smaller and all nodes to the right are greater. Inorder traversals on true or valid binary search trees will yield a sorted ordering of items.

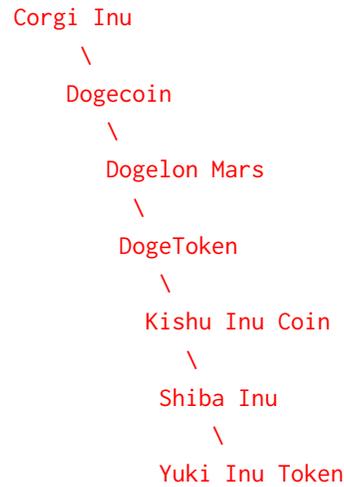
Answers may vary, but here are two examples of a valid binary search trees.

Example 1:

Kishu Inu Coin



Example 2:

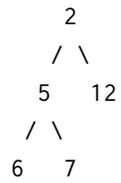


- (c) *Extra:* Provide an example of a tree where the DFS pre-order, DFS in-order, and BFS traversals are the same, and where the DFS post-order traversal is the opposite order of the previous three traversals.

Answers may vary, but see Example 2 from above.

## 2 Mechanical Heap Practice

Consider the following min-heap:

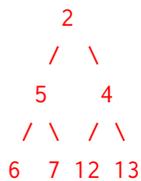


- (a) Draw the heap after inserting the following numbers (in the given order and in succession): 4, 13, 3

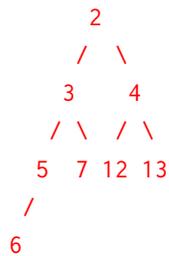
After inserting 4:



After inserting 13:



After inserting 3:

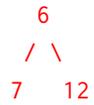


- (b) Now, returning back to the initially given min-heap, draw the heap after removing the minimum element twice. Assume that when bubbling down, the parent will bubble down towards the minimum of the two children if both children have lower values.

After removing 2:



After removing 5:



- (c) *Extra:* What is the runtime of finding a specific element within the heap, assuming we have access to the underlying data structure (e.g. if the heap is represented as an array, we can scan the array)?

Since a heap is not necessarily a valid binary search tree, we can make no guarantees about the structure of elements other than heap properties (every element has higher priority than its two children). Thus, we cannot perform a binary search or logarithmic search on the underlying data structure. Our fastest approach, then, is to scan in linear time. We can upper bound this search time by a  $O(N)$  runtime, for a heap of  $N$  elements.

### 3 Asymptotics Review

Give the tightest bounds (either  $\Omega/O$  or  $\Theta$ ) for the following functions.

- (a) Note that `nextInt(int bound)` returns a random integer between 0 (inclusive) and `bound` (exclusive) and takes constant time.

```

1 void f(int N) {
2     Random rand = new Random();
3     for (int i = 1; i < N; i += rand.nextInt(i) + 1) {
4         for (int j = 0; j < i; j++) {
5             System.out.println(i + j);
6         }
7     }
8 }

```

$\Omega(N)$  and  $O(N^2)$ .

In the best case, `rand.nextInt(i)` always outputs `i - 1`. Then, the outer loop's `i` will take on values `1, 2, 4, ..., N`. The inner loop takes linear time with respects to `i`. So, the runtime is approximately the sum:  $1 + 2 + 4 + \dots + N = \Theta(N)$ .

In the worst case, `rand.nextInt(i)` always outputs 1. Then, the outer loop's `i` will take on values `1, 2, 3, ..., N`. The inner loop takes linear time with respects to `i`. So, the runtime is approximately the sum:  $1 + 2 + 3 + \dots + N = \Theta(N^2)$ .

- (b)

```

1 void g(int N) {
2     if (N < 10000) {
3         return;
4     }
5     for (int i = 0; i < N; i++) {
6         i++;
7     }
8     g(N / 2);
9     g(N / 2);
10 }

```

$\Theta(N \log(N))$ .

With asymptotics, we consider very large values of  $N$  so the base case being  $N < 10000$  instead of  $N < 1$  does not change the overall runtime. Also, because our function always has the same behavior for any input  $N$ , the worst and best case runtimes are the same. So, the bound will be a big Theta bound.

Drawing out the recursive tree, there are still approximately  $\log(N)$  levels. Each level does approximately  $N$  amount of work. Summing over all levels:  $N + N + \dots + N = \Theta(N \log(N))$ .  
 $\log(N)$ times



## 5 Hash Codes and Runtime

Suppose we're given the following Student class definition.

```

1  class Student {
2      public final static String isStudent = "yes";
3
4      public String name;
5      public String major;
6      public String school;
7      public int year;
8      public int id;
9
10     public Student(String name, String major, String school, int year, int id) {
11         ...
12     }
13
14     @Override
15     public int hashCode() {
16         -----
17     }
18
19     @Override
20     public boolean equals(Object o) {
21         if (o == null || this.getClass() != o.getClass()) {
22             return false;
23         }
24         Student other = (Student) o;
25         return school.equals(other.school) && id == other.id;
26     }
27 }

```

- (a) Assume that major, school, and id are never modified after initializing a Student, but year can be modified. For each of the following hash codes, answer whether they are "valid" or "invalid" and justify why.

1. **return** isStudent.hashCode();

**Valid: Returns the same hash code for equal Students.** Because isStudent is defined as a final variable, it is never modified so hashCode consistently returns the same integer.

2. **return** id

**Valid: Returns the same hash code for equal Students.**

3. **return** year + id;

Invalid: A Student's year can be modified, changing the hash code. For example, imagine a Student is placed into a HashSet. Then, their year is changed, modifying the hash code. If we try adding an equal Student to the HashSet, this new hash code might not hash to the same bucket as before.

4. **return** school.hashCode() + id;

Valid: Returns the same hash code for equal Students.

5. **return** 17 \* school.hashCode() + 5 \* major.hashCode() + id;

Invalid: Two Students with the same school and id can be initialized with different majors, resulting in different hash codes even though the Students are equal.

- (b) 1. What is the best case and worst case runtime of the following function, in terms of  $N$  and  $K$ ? Suppose  $N$  is equal to `roster.size()` and the name, major, and school fields for all Students are  $\Theta(K)$  length. Assume that computing hash codes take constant time.

```

1 Set<Student> removeDuplicates(ArrayList<Student> roster) {
2     Set<Student> noDuplicates = new HashSet<>();
3     for (Student s : roster) {
4         if (noDuplicates.contains(s)) {
5             System.out.println("Duplicate found: " + s.name);
6         }
7         noDuplicates.add(s);
8     }
9     return noDuplicates;
10 }
```

Best Case:  $\Theta(\quad)$  Worst Case:  $\Theta(\quad)$

2. Which of the valid hash codes from above would be most likely to cause these runtimes?

During a call to `contains`, the hash code of `s` is computed first. Then, `s.equals` is called on every Student in the corresponding hash bucket to check whether `s` is already in the set. `add` is very similar: `s` is hashed and the corresponding hash bucket is iterated over, checking if `s` is already contained in the set. If `s` is not already in the set, it is added to the end of the hash bucket.

When considering the runtime, let's consider the two steps separately:

1. hashing the Student instance
2. iterating through the hash bucket, calling `equals` on each Student in the bucket.

Best Case:  $\Theta(N)$

1. It takes constant time to compute the hash code and every Student in `roster` is hashed once.

2. With a good hash code like #2, `contains` and `add` never call `equals` if no collisions occur. Note it is only possible for no collisions to occur if the `loadFactor` is less than 1. The overall runtime is therefore  $\Theta(N)$ .

Worst Case:  $\Theta(N^2K)$

1. Like before, computing the hash code for each `Student` instance takes constant time.
2. `isStudent` is a **static** variable and is shared across all `Student` instances. So, a hash code like #1 places all `Students` into the same hash bucket. Calls to `contains` and `add` would search through all `Students` currently in the bucket, each call to `equals` taking  $\Theta(K)$  time. The overall runtime of `removeDuplicates` would be approximately  $2K(1 + 2 + 3 + \dots + N) = \Theta(N^2K)$ .