1 Balanced and Valid BSTs

Given the following binary trees, determine if each is a valid BST, and whether or not it has minimum-BST-height. A minimum height BST has the same height as the optimal binary search tree containing the same elements.

```
         6                        10                       9
       /   \                    /    \                   /   \
      5     9                  6      12               7       11
    /     /   \              /       /               /   \       \
   2     7     10           3       11              5      8       13
     \                     / \                     / \
       3                  1   7                   3   8
         \
          4
      Valid:  T               Valid:      F           Valid:      F
 Min-Height:      F      Min-Height:      F      Min-Height:   T
```

Suppose we know the height `H` and number of nodes `N` of a BST. Can we determine whether or not this BST is minimum-BST-height without having to check the values of each node? Why or why not?

Yes, this is possible. The minimum height *minH* can be found by $minH = \lfloor log_2 N \rfloor$. If your tree's height `H` is the same as `minH`, then you are minimum height. Otherwise, you aren't.

2 Ding Dong Design Discussion: A Better Hash Map?

Ding has a wild idea to improve hash maps: Instead of placing items via modulo, use the faster absolute value operation (`bin = Math.abs(key.hashCode())`), and preemptively expand the size of the hash map as necessary (so the bin exists). Ding's twin brother, Dong, believes Ding's idea is boloney, as a hash map always runs in constant time anyways, and thus doesn't need improvement. Enumerate the pros and cons of Ding's modified implementation, and reject or justify the validity of Dong's reaction.

Almost all the points raised aren't correct.

Modulo is slightly slower than absolute value, but not by much. This has to do with how modulo in general is implemented by arithmetic operations (it abuses floor division, that `(int)/(int)=(int)`). Absolute value can be implemented via flipping all the bits and adding one. It should be noted that in particular, modding by a power of 2 can be implemented much faster than the normal modulus, and thus HashMaps in Java are generally a size that is a power of 2.

Ding potentially expands too much for an insertion, which would be space inefficient. Imagine that the first element you insert into a HashMap has a hashcode output of 9,000,000. Then the HashMap would expand to 9 million just to accomodate one item. Further suppose that we didn't insert anything more into this HashMap. Then we just wasted a ton of time and space.

Hash Maps don't always run in constant time, it depends on the runtime of the hashcode function of the data type being stored and the depth of the bins. While a hash map can't really control the runtime of a datatypes hashcode, it can do some extra math processing on the hashcode output to ensure we are somewhat evenly distributed (see Java's source code and hash(int n) method for an example of this optimization over a naive implementation).

3 Coco's Guerrilla Section Conundrum

Coco doesn't know sign language, but she wants to learn by associating each letter with a number. Help her hash `(Character,Integer)` entry pairs into a `HashMap<Character,Integer>`. Assume that the hash map starts with 10 buckets and resizes with load factor 0.65. Draw the results of inserting four items: `('a',9) ('u',2) ('b',5) ('a',5)`. Determine `C`, the total number of hash collisions, and `F`, the current load factor after the four insertions. The hash code for `'a'` is 97, `'b'` is 98, and so on.

The fourth entry overrides the first, which doesn't count as a hash collision (it's a value update). At the end there are three entries, so F=3/10=0.3. There was 1 hash collision ('a' and 'u'), so C=1.

4 Heaps of fun

(a) Assume that we have a binary min-heap (smallest value on top) data structure called `Heap` that stores integers, and has properly implemented `insert` and `removeMin` methods. Draw the heap and its corresponding array representation after each of the operations below:

```
Heap h = new Heap(5); // Creates a min-heap with 5 as the root
h.insert(7);
h.insert(3);
h.insert(1);
h.insert(2);
h.removeMin();
h.removeMin();
```

```
Heap h = new Heap(5); //Creates a min-heap with 5 as the root
[5]                5
h.insert(7);
[5,7]              5
                  /
                 7
h.insert(3);
[3,7,5]            3
                  / \
                 7   5
h.insert(1);
[1,3,5,7]          1
                  / \
                 3   5
                /
               7
h.insert(2);
[1,2,5,7,3]        1
                  / \
                 2   5
                / \
                7   3
h.removeMin();
[2,3,5,7]           2
                   / \
                  3   5
                 /
                7
h.removeMin();
[3,7,5]             3
                   / \
                  7   5
```

(b) Your friend Sahil Finn-Garng challenges you to quickly implement an integer max-heap data structure - "Hah! I'll just use my min-heap implementation as a template to write max-heap.java", you think to yourself. Unfortunately, two Destroyer Penguins manage to delete your `MinHeap.java` file. You notice that you still have `MinHeap.class`. Can you still complete the challenge before time runs out? Hint: you can still use methods from MinHeap.

Yes. For every insert operation negate the number and add it to the min-heap. For a removeMax operation call removeMin on the min-heap and negate the number returned. Any number negated twice is itself (with one exception in Java, 2^{-31}), and since we store the negation of numbers, the order is now reversed (what used to be the max is now the min).

5 Extra for Experts: LRU Cache

LRU stands for Least Recently Used. When removing from the LRU cache, we always remove the least "recently used" item. An item is "recently used" if it was recently added (`add()`) or accessed (`get() contains()`). For example, if we add three items, A, B, and C, and then check `.contains(B)`, calling remove three times would return A, C, and then B. Describe how to implement an LRU cache as efficiently as possible. Then give the tightest Big-O runtimes of `add() get() remove()` and `contains()`. Hint: You are not limited to one classic data structure.

Use a FastLinkedList, aka a Queue + indexing HashMap.

For a more thorough explanation, see:

`https://www.quora.com/What-is-the-best-way-to-Implement-a-LRU-Cache`

Essentially, we keep a Queue of recently used things. The item at the front of the queue is the least recently used, whereas the item at the back of the queue is the most recently used. Anytime you access something, no matter what position it is in the queue, you can remove it and add it to the back (to properly mark it as recently used). If someone asks to remove the least recently used item, just return the front of the queue.

As of now, the runtime is not perfect. Searching through a queue takes linear time. We can fix this by maintaining essentially an indexer: a HashMap mapping nodes to the corresponding QueueNode. You'll also need to ensure that your Queue is implemented via a Doubly Linked List, and that it supports remove and add operations that take in a QueueNode argument (will have to break the abstraction barrier, but that's ok). Then you can accomplish all the tasks in constant time (assuming good hashing).

An alternative solution involves just using a PriorityQueue, which isn't too bad, but doesn't achieve constant runtimes like the Queue+HashMap solution does.