Data Structures
Spring 2021
Midterm 1 — Solutions

1 Asymptotic Analysis

QUESTION: (a) [5 pts] Rank the following functions in order from lowest to highest big-O complexity (you do not need to show any work or any formal proofs):

$$6n + 1700 \qquad 500 \lg n \qquad 3^n \qquad 0.2n^{15} - 75n^8$$

ANSWER:

1. $500 \lg n$
2. $6n + 1700$
3. $0.2n^{15} - 75n^8$
4. 3^n

COMMENTS: This went fine for most people. The most common error was flipping the order of $0.2n^{15} - 75n^8$ and 3^n; the important thing to be aware of here is that a function with an n in an exponent will *always* grow faster than a function that's a polynomial in n.

QUESTION: (b) [10 pts] Let $f(n) = 4n^2 - 2n + 7$. What is the big-O class of $f(n)$? You should give the smallest class possible. Use the definition of big-O to show that $f(n)$ is in the big-O class that you specified.

ANSWER: We consider $g(n) = n^2$:

$$\begin{aligned} 0 &\leq 4n^2 - 2n + 7 \\ &\leq 4n^2 + 2n + 7 \\ &\leq 4n^2 + 2n^2 + 7n^2 \\ &= 13n^2 \end{aligned}$$

For $n_0 \geq 1$ and $c = 13$, we establish that $f(n) \in O(n^2)$.

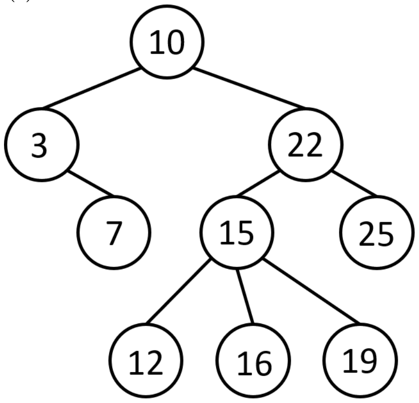
COMMENTS: The most common error was that many people chose an n_0 and plugged in that specific value to show that the desired inequality holds for the chosen n_0. This is not quite enough, because the definition of big-O states that the inequality must hold *for all* values of $n \geq n_0$.

2 Binary Search Trees

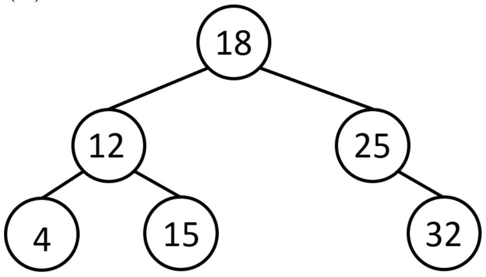
QUESTION: (a) [9 pts] For each of the following structures:

1. State whether or not it is a binary search tree, and
2. If not, state what is wrong with it.

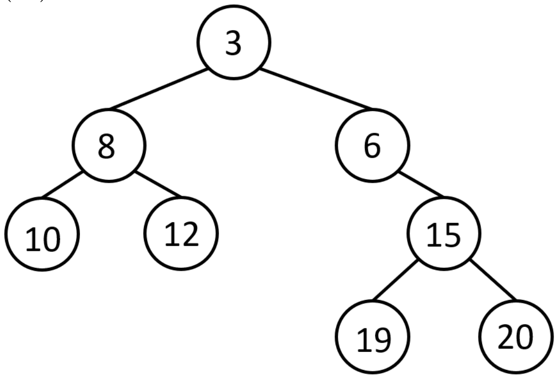
(i)



(ii)



(iii)



ANSWER:

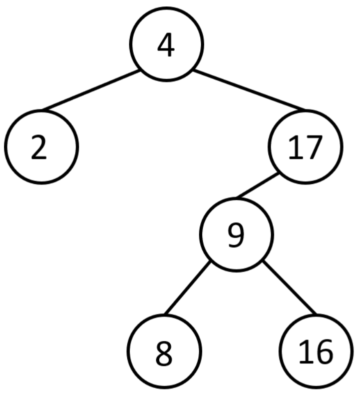
(i) No. The node 15 has three children, which is one too many.

(ii) Yes.

(iii) No. The left children do not have lesser values than their parents.

COMMENTS: Not too many errors here.

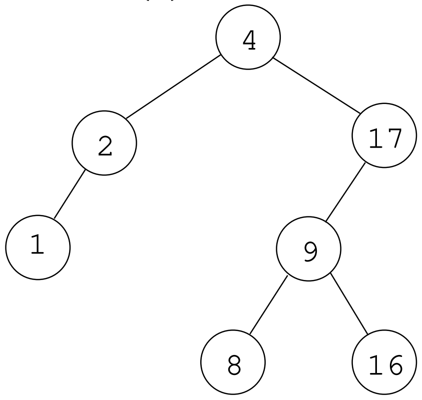
QUESTION: (b) [6 pts] Here is a binary search tree:



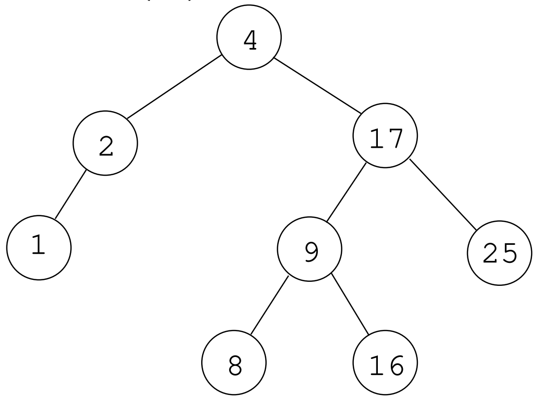
Show how this BST changes as the following operations are performed in sequence: `add(1)`, `add(25)`, `add(19)`. Your answer should include **three** pictures, one after **each** addition.

ANSWER:

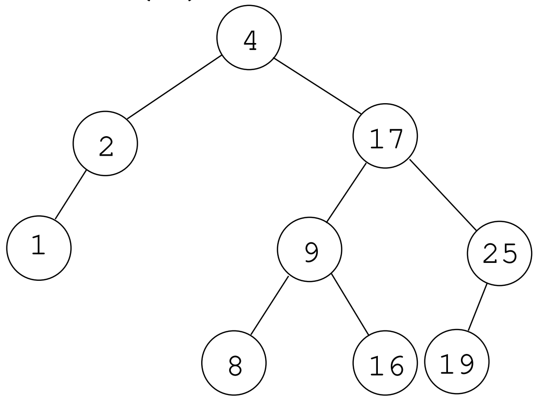
After `add(1)`:



After `add(25)`:



After `add(19)`:



COMMENTS: For this question, the key concept is that keys inserted into to binary trees are **always** leaves. So long as each addition, in sequence, created a new leaf, then the additions were largely performed correctly.

3 Mysterious Methods

Suppose that, somewhere outside the `Queue` class, my program contains the following method (assume my `Queue` stores `int` values):

```
public void mystery (Queue q, int x) {
    if (q.isEmpty()) {
        q.enqueue(x);
        return;
    }
    for (int i = 0; i < q.size(); i++) {
        int y = q.peek();
        if (y == x) return;
        q.dequeue();
        q.enqueue(y);
    }
}
```

Question: (a) [10 pts] What does this do? That is, in what way is the input `q` modified (or not) when `mystery` runs?

Answer: (a) If the queue is empty, `x` is inserted as the sole item; otherwise, the queue's contents are *rotated* until either `x` is at the head or, if `x` is not present, `q` is returned to its original state.

Comments: The most common conceptual problem with this question was to assume that **only** this method affected the queue. A number of people failed to see that the queue could be manipulated outside of this method, and therefore the queue could have arbitrary contents when called.

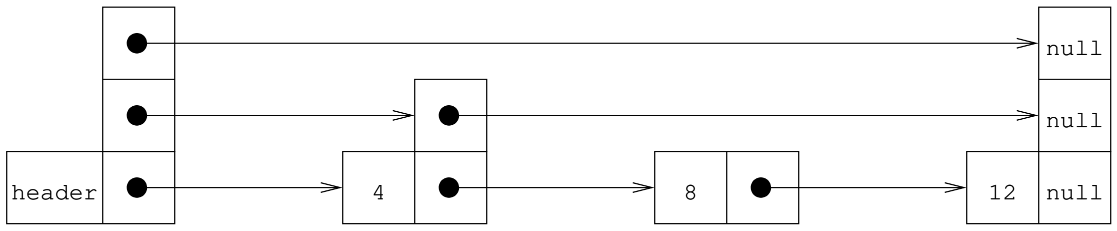
Question: (b) [10 pts] Suppose the `Queue` class is implemented using a doubly linked list with head and tail pointers. Let n be the number of items in the queue. In terms of n, what is the worst case big-O runtime of `mystery`? What's the best case big-O runtime? **Explain your reasoning.**

Answer: Best case is that `x` is already at the head of the queue, which requires $O(1)$ time. In the worse case, `x` is not present, and the entire queue of n items is rotated (removed and re-inserted), requiring $O(n)$ time.

Comments: The concept of a *best case* was often misunderstood, in which many people referred to the time that this method would take when the queue was *empty*. However, big-O expressions refer to bounds **as a function of the input size**. Even for a *best case* analysis, the result must be given in terms of an arbitrarily large queue size. That is, how much time does this method take, in the best case, **even when n is large?** Here, the result is $O(1)$ not because the queue is empty, but because the value sought (`x`) will be, in this best case, at the front of the queue already, no matter how large n is.

4 Skip Lists

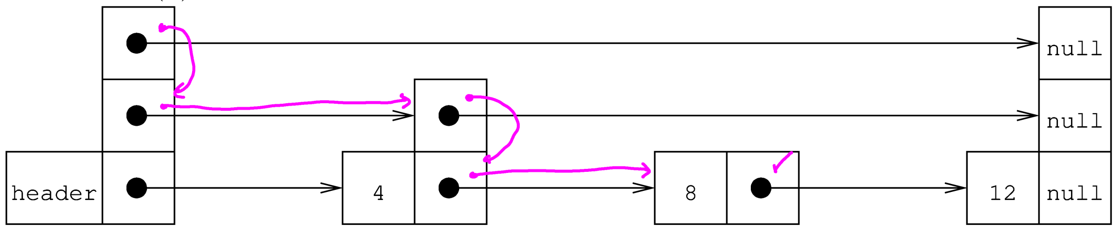
For the following skip list of integers...



...answer the following questions about performing the operation `add(10)`:

QUESTION: (a) [5 pts] Trace the search that would performed by the operation `find(10)`. That is, show where the search would begin, and how it would move through the skip list structure, from beginning to end. (If you draw on this page, please be sure to scan it with what you submit; if answer this question on another piece of paper, redraw the skip list there.)

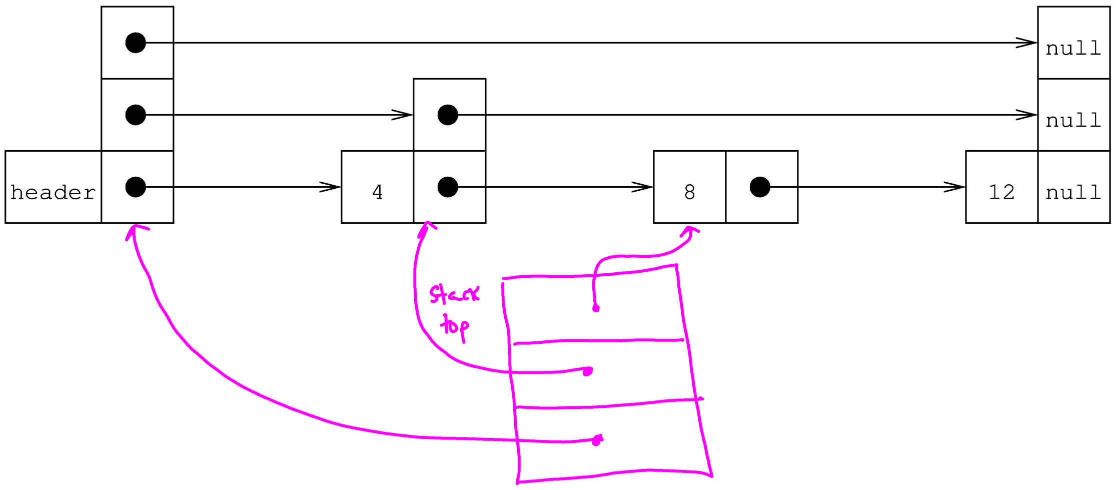
ANSWER: (a)



COMMENTS: Some people's answers did not quite follow the search path all the way to the predecessor node at the lowest level.

QUESTION: (b) [5 pts] Show the resulting stack of pointers that `find(10)` would return.

ANSWER: (b)



COMMENTS: In many cases, the stacks drawn were too small or too large. In the former case, some people forgot that a pointer to the `header` node was required for level 2; in the latter case, some people included separate pointers to some nodes (e.g., the `4` node, the `header` node) for *each* level that those nodes contained. Here, one pointer to the given node was needed, yielding a stack height of 3 to match the list's height.

QUESTION: (c) [5 pts] Assume that the new node is of height 3 (that is, with pointers for levels 0, 1, and 2). Now assuming a fair coin (probability of heads or tails is 1/2 each), what sequence of coin flips yielded this height? And what was the probability of that sequence?

ANSWER: (c) *Heads, heads, tails*,[1] which has a probability of $\frac{1}{2} \times \frac{1}{2} \times \frac{1}{2} = \frac{1}{8}$.

COMMENTS: This question caused a little trouble, because the presentation of it to each section was a little different, and because some elements of that presentation created a little confusion.

Some people confused the presentation of the *expected number of coin flips* for the probabilty of a specific outcome (which is what this question presents). The expected number of flips is 2 (given a sum of probabilities $1 + \frac{1}{2} + \frac{1}{4} + \ldots$), but a probability must be a value between 0 and 1.

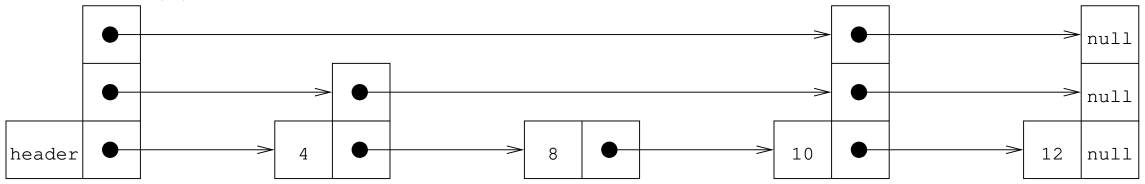
And the probability here depends on whether you consider the final flip, which must terminate the sequence, as one of the events. Some people took, from the *expected number of flips* analysis, that the final flip did not count, in some sense, as part of the probabilistic evaluation, counting only the first two flips. However, the third

[1]That's if you're in Section 01; for Section 02, it would have been *tails, tails, heads*. Both are equivalent.

flip was only the terminating event because of its outcome; it still had a $\frac{1}{2}$ probability of not terminating the sequence, allowing another flip. Many people calculated a probability of $\frac{1}{2} \times \frac{1}{2} = \frac{1}{4}$, which clearly has the concept right, but gets a detail wrong.

QUESTION: (d) [5 pts] Redraw the skip list with this new node added into the structure.

ANSWER: (d)



COMMENTS: Not many problems here.

5 Checking Parentheses

QUESTION: [30 pts] Consider a string consisting of the parenthetical markings `(`, `)`, `<`, and `>`. A string is *properly matched* if: (1) there are the same number of opening symbols— `(`, `<` —and closing symbols— `)`, `>`; (2) every opening symbol is matched with a closing symbol of the same type; and (3) the parenthetical symbols are properly nested. For example, the string `(<><()>)` is properly matched. The strings `(()`, `(<)>`, and `)<>(` are not properly matched.

Write a method that takes a `String` as input and returns `true` if it is properly matched and `false` if not. (You may assume that the input `String` includes only the four allowed symbols). Your method should use stacks and/or queues to accomplish this task; you may declare a few extra variables of type `int`, `char`, `double`, `boolean`, or `String` if you want, but you should NOT use any data structures that can store multiple values other than stacks and queues (that is, do not declare any additional arrays, Lists, skip lists, linked lists, etc.)

The `charAt(int i)` method in the `String` class may be helpful; you can write `s.charAt(i)` to look up the character in position `i` of the string `s` (indexing starts at 0).

ANSWER:

```java
public static boolean matched (String s) {

    Stack<Character> stack = new Stack<Character>();
    for (int i = 0; i < s.length(); i += 1) {
        char c = s.charAt(i);
        if (c == '(' || c == '<') {
            stack.push(c);
        } else if (c == ')') {
            if (stack.isEmpty() || stack.pop() != '(')
                return false;
        } else if (c == '>') {
            if (stack.isEmpty() || stack.pop() != '<')
                return false;
        }
    }

    return stack.isEmpty();

}
```

COMMENTS: Mistakes fell into two basic categories on this challenging question: *big, conceptual errors* and *small, detailed goofs*.

Big errors: The most common was the failure to see that a single stack is needed to keep track of, and to pair, open and closing braces. Each opening brace needed to correspond to a *push* onto the stack; each closing required a *pop* and a matching of the brace type. People often used queues (which don't help), or multiple stacks. In either case, the ability to pair and properly order the openings and closings couldn't be achieved, although some clever ad-hoc approaches handled a number of cases.

Small errors: The most common of these was to fail to test whether the stack was empty. In particular, if the stack is empty when a closing brace is encountered, then there is (at least) one closing brace too many, and the pattern fails. Likewise, if the stack is non-empty at the end, then there are unmatched opening braces, and the pattern again fails. The final common error was to pair and open and a close, but not to compare the *type* of opening brace to its closing brace.