CISC 3130 Exam 2 (Section MY9)

November 25, 2024

Name: _____

Question 1 (10 points)

Complete the `count` method of the following SinglyLinkedList class. The method should return the number of times that e occurs in the list. Assume that no element is null and that e is not null.

Additionally, state the big-Oh running time of your method.

```
public class SinglyLinkedList<E> {
    private static class Node<E> {
        E data;
        Node<E> next;
    }
    private Node<E> head, tail;
    private int size;
    // Methods addFirst, addLast as usual (don't write)

    public int count(E e) {









    }
}
```

Question 2 (8 points)

Suppose we have a SinglyLinkedList<E> class with the following methods: addFirst(e), addLast(e), removeFirst(), removeLast(), getFirst(), and getLast().

Additionally, suppose we have a DoublyLinkedList<E> class with the same methods.

Each method behaves in the obvious manner and has the obvious running time.

Now, suppose we have the following interface:

```
public interface Stack<E> {
    void push(E e);
    E pop();
    E peek();
}
```

We would like to implement the Stack interface, using a linked list to store the elements.

The table below contains eight proposed implementations. The table indicates what kind of linked list is used to store the elements, and how the Stack methods are implemented in terms of the linked list methods. For example, the top left empty cell represents an implementation where the elements are stored in a SinglyLinkedList, push(e) calls addFirst(e), pop() calls removeFirst(), and peek() calls getFirst().

For each proposed implementation, write "correct" if it is correct; otherwise, write "incorrect." An implementation is correct if each method produces the behavior expected by users, regardless of how it works under the hood. For example, peek() should return the most recently added element; we don't care where it is stored.

Additionally, for each correct implementation, write "efficient" if its big-Oh running time is efficient; otherwise, write "inefficient." (We don't care about the efficiency of incorrect implementations.)

	SinglyLinkedList	DoublyLinkedList
push(e): addFirst(e), pop(): removeFirst(), peek(): getFirst()		
push(e): addLast(e), pop(): removeLast(), peek(): getLast()		
push(e): addLast(e), pop(): removeFirst(), peek(): getFirst()		
push(e): addFirst(e), pop(): removeLast(), peek(): getLast()		

Question 3 (8 points)

Use a stack to determine whether the delimiters in the following expression are balanced, that is, each opening delimiter is matched with a closing delimiter of the same type, and the delimiters are nested properly. Follow the algorithm that was covered in class. You will be graded on your tracing of the stack, as well as on your final determination of "balanced" or "not balanced."

```
[ ( ) ] { } { [ [ ] ( ) ( ) ] }
```

Question 4 (6 points)

Consider the following method:

```
public static void mystery(Deque<Integer> stack) {
    Queue<Integer> queue = new LinkedList<>();

    while (!stack.isEmpty()) {
        queue.add(stack.peek());
        queue.add(stack.pop());
    }

    while (!queue.isEmpty()) {
        stack.push(queue.remove());
    }

    while (!stack.isEmpty()) {
        System.out.print(stack.pop() + " ");
    }
}
```

(a) Write the method's running time in big-Oh notation.

(b) Suppose the method is passed the following stack:

```
top [10, 20, 30] bottom
```

Write the method's output in this case.

Question 5 (10 points)

Complete the following method named `mirror` that accepts a queue of strings as a parameter and appends the queue's contents to itself in reverse order. For example, if a queue named `q` stores `["a", "b", "c"]`, the call of `mirror(q)` should change it to store `["a", "b", "c", "c", "b", "a"]`.

Rules:

- Do not use any auxiliary collections (arrays, ArrayLists, etc.) as storage, except for a single stack or queue (not both).
- You may only use the following Queue methods: add(e), remove(), peek(), size(), and isEmpty().
- If you use a stack, represent it with a Deque. You may only use the following Deque methods: push(e), pop(), peek(), size(), and isEmpty().
- Do not use an enhanced for loop (aka for-each loop).

```
public static void mirror(Queue<String> queue) {

}
```

Question 6 (10 points)

Suppose that a deque is implemented using a circular array, aka a ring buffer. Assume that the deque starts with capacity **5** and that it **doubles** its capacity when an element is added to a full deque. Assume that when an element is removed from the deque, the array element where it resided is set to null.

At each of the points indicated below,

- Write the state of the deque's internal array.
- State the values of indexOfFirst and indexOfLast.

```
Deque<Integer> deque = new ArrayDeque<>();
deque.addLast(10);
deque.addFirst(20);
deque.addFirst(30);
deque.addLast(40);
deque.addLast(50);
// Point A:




deque.addFirst(60);
deque.addFirst(70);
// Point B:




for (int i = 0; i < 6; i++) {
    deque.removeLast();
}
// Point C:
```

Question 7 (8 points)

Draw a diagram illustrating the internal state of the HashMap immediately after the following code runs.

```
Map<Integer, Integer> map = new HashMap<>();
map.put(13, 10);
map.put(15, 10);
map.put(13, 5);
map.put(18, 15);
map.put(23, 6);
map.put(-3, 7);
map.remove(6);
map.remove(15);
```

Additionally, write the final size, capacity, and load factor.

Assume that the initial capacity is **5**. Assume that we increase the capacity when the load factor is $> \mathbf{0.5}$, and that we **double** the capacity (not 2 * capacity + 1). Note: the order of the entries within each "bucket" doesn't matter.

Assume that the hash function takes a key and returns `Math.abs(key) % capacity`.

Question 8 (6 points)

Consider the following method:

```
public static void mystery(SequencedMap<String, Integer> map) {
    SequencedSet<Integer> set = new LinkedHashSet<>();

    for (String s : map.keySet()) {
        set.add(map.get(s));
    }

    System.out.println(set);
}
```

(a) Write the method's running time in big-Oh notation.

(b) Suppose the method is passed the following map:

```
{f=40, d=20, c=40, e=30, b=30,  a=20}
```

Write the method's output in this case.

Question 9 (6 points)

Consider the following method:

```
public static void mystery(int n) {
    SequencedSet<Integer> set = new LinkedHashSet<>();

    for (int i = 1; i <= n * 2; i++) {
        set.add(i);
    }

    for (Iterator<Integer> iter = set.iterator(); iter.hasNext(); ) {
        System.out.print(iter.next() + " ");

        if (iter.hasNext()) {
            iter.next();
        }
    }
}
```

(a) Write the method's running time in big-Oh notation.

(b) Suppose the method is passed the value 5. Write the method's output in this case.

Question 10 (12 points)

Complete the following method. The method should determine whether the provided Collection contains an element with exactly the provided frequency. For example, the call

```
containsFrequency(List.of("a", "c", "b", "a", "a"), 3)
```

should return true, since `"a"` has a frequency of 3. If the second argument would be 1, the method would also return true, since `"c"` (as well as `"b"`) has a frequency of 1. But if the second argument would be 2, the method would return false.

Your method must run in $O(n)$ time. Tip: use a Map.

```
public static boolean containsFrequency(Collection<String> coll,
                                        int frequency) {









}
```

Question 11 (8 points)

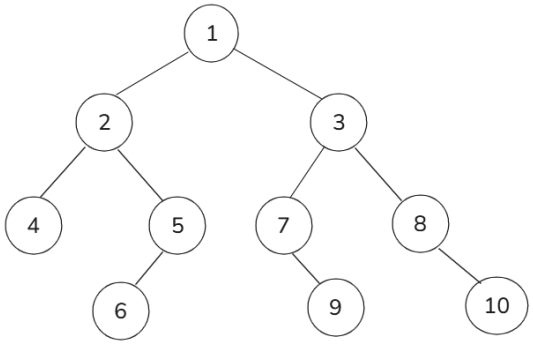
Suppose we have the following class:

```
public class BinaryTreeNode<E> {
    public E data;
    public BinaryTreeNode<E> left, right;
}
```

Complete the following method. The method should return the number of leaf nodes in the binary tree that is rooted at the given node. A leaf node is a node that has no children. You may use the provided helper method.

```
public static int countLeaves(BinaryTreeNode<?> root) {









}

private static boolean isLeaf(BinaryTreeNode<?> node) {
    return node.left == null && node.right == null;
}
```

Question 12 (8 points)



For the above tree, write the following traversals:

(a) preorder

(b) inorder

(c) postorder

(d) level-order