CISC 3130 Final Exam (Section MY9)

December 16, 2024

Name: ______________________________

Question 1 (12.5 points)

(a) For the following array, write the state of the array after each of the first three passes of selection sort. (That is, write what the array looks like immediately after the first pass, then write what it looks like immediately after the second pass, and then write what it looks like immediately after the third pass.)

```
{32,  87,  94,  12,  72,  25,  65,  73}
```

(b) Repeat, using bubble sort. (Again, only the first three passes.)

```
{32,  87,  94,  12,  72,  25,  65,  73}
```

(c) Repeat, using insertion sort. (Again, only the first three passes.)

```
{32,  87,  94,  12,  72,  25,  65,  73}
```

(d) Trace the complete execution of merge sort. (Continue until the array is fully sorted, not just three passes.)

```
{32,  87,  94,  12,  72,  25,  65,  73}
```

(e) Trace the complete execution of quick sort. (Continue until the array is fully sorted, not just three passes.) When choosing a pivot, always choose the first element of the relevant subarray.

```
{32,  87,  94,  12,  72,  25,  65,  73}
```

Question 2 (8 points)

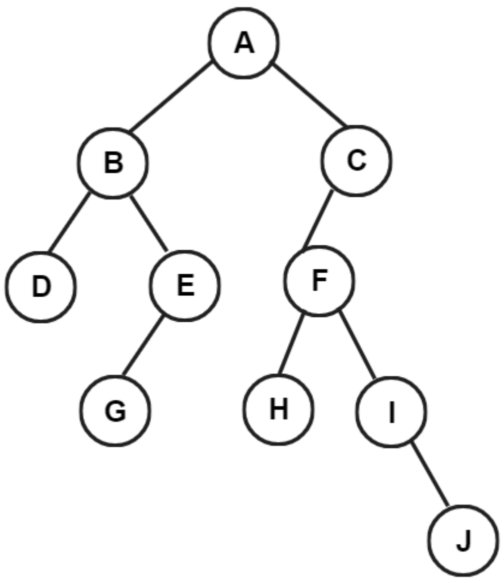
(a) Here is an array that stores the elements of a heap: [15, 8, 13, 7, 5, 3, 4, 6, 1]. Write these elements in tree form. (1 point)

(b) Continuing from part a, suppose we add 12 to the heap. Show what the tree looks like now. (3 points)

(c) Continuing from part b, suppose we remove the root (the max) from the heap. Show what the tree looks like now. (3 points)

(d) Continuing from part c, now write the elements of the heap as an array. (1 point)

Question 3 (8 points)



For the above tree, write the nodes:

(a) in preorder.

(b) in postorder.

(c) in inorder.

(d) in level-order.

Question 4 (4 points)

Suppose we start with an empty binary search tree and add the following elements, in the following order: 48, 58, 92, 51, 77, 17, 10, 23. Draw a diagram showing what the tree looks like now.

Question 5 (5 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.addFirst(10);
deque.addLast(20);
deque.addLast(30);
deque.addFirst(40);
deque.addFirst(50);
// Point A:






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






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

Question 6 (5 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(-26, 7);
map.remove(26);
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 > **0.5**, and that we **double** the capacity (not 2 * capacity + 1).

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 7 (19.5 points)

Consider the following interface:

```
public interface GrabBag<E> {
    void insert(E item);
    E sample();
    E remove();
}
```

A GrabBag represents a bag from which you can grab a random item.

- insert(item): adds item to the grab bag.
- sample(): returns, but does not remove, a random item from the grab bag. The user should not be able to determine ahead of time which item will be returned.
- remove(): removes and returns a random item from the grab bag. The user should not be able to determine ahead of time which item will be removed.

Write a concrete class that implements the above interface. Use a List to store the elements.

For full credit, the running time of each method should be $O(1)$. If your methods are correct but less efficient than $O(1)$, you will still get most of the credit.

Note: The Random class has a no-arg constructor, `Random()`. It has an instance method, `nextInt(int max)`, that returns a random int between 0 (inclusive) and `max` (exclusive). For example, if `rand` refers to a Random object, then `rand.nextInt(4)` returns 0, 1, 2, or 3.

Note: You may omit all import statements.

Question 8 (6 points)

For each of the following, write a line of code that creates a JCF collection. Make sure to use an appropriate interface as the variable type.

(a) We want to store Strings and be able remove the minimum element in less than linear time; we also wish to allow duplicates.

(b) We want to store integers in a list with efficient add and remove at front (but we also want to use other list operations).

(c) We want to store associations between keys (unique integers) and values (Strings), and we want to be able to access them in sorted key order.

(d) We want to store Strings without duplicates but we want to be able to access them in insertion order.

Question 9 (5 points)

Consider the following method:

```
public static Set<String> mystery(Map<String, String> map) {
    Set<String> set = new TreeSet<>();

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

    return set;
}
```

(a) Assuming that the type of map passed in is LinkedHashMap, what is the running time of the method in big-Oh notation, where n is `map.size()`?

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

```
{f=z, d=x, e=y, b=y, c=z, a=x}
```

Write the elements of the returned Set in proper order.

Question 10 (7 points)

Write the output of the following code:

```
Deque<Integer> stack = new LinkedList<>();
stack.push(7);
stack.push(10);
System.out.print(stack.peek() + " ");
System.out.print(stack.pop() + " ");
stack.push(3);
stack.push(5);
System.out.print(stack.pop() + " ");
System.out.print(stack.size() + " ");
System.out.print(stack.peek() + " ");
stack.push(8);
System.out.print(stack.pop() + " ");
System.out.print(stack.pop() + " ");
```

Question 11 (8 points)

For each of the following pieces of code, write the running time in big-Oh notation:

(a)
```
int sum = N;
for (int i = 0; i < 1000000; i++) {
    for (int j = 1; j <= i; j++) {
        sum += N;
    }
    for (int j = 1; j <= i; j++) {
        sum += N;
    }
    for (int j = 1; j <= i; j++) {
        sum += N;
    }
}
```

(b)
```
int sum = 0;
for (int i = 1; i <= N - 2; i++) {
    for (int j = 1; j <= i + 4; j++) {
        sum++;
    }
    sum++;
}
```

(c)
```
int sum = 0;
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= N * N; j++) {
        sum++;
    }
    for (int j = 1; j <= 100; j++) {
        sum++;
    }
    for (int j = 1; j <= N; j++) {
        sum++;
    }
    sum++;
}
```

(d)
```
int sum = 0;
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= 100; j++) {
        sum++;
    }
}
```

Question 12 (12 points)

Suppose we have the following records:

```java
record Name(String first, String last) {}
record Person(Name name, int age) {}
```

Recall that a record has public accessor methods with the same names as its fields. For example, the Person record has a method age() that returns the value of the age field. Suppose we have a List<Person> named `people`. Solve the following problems using streams. Do not use loops or recursion.

(a) Print the average age of all people who are above the age of 30. (If it doesn't exist, print nothing.)

(b) Print the number of distinct first names in the list. For example, if the first names in the list are Jane, Adam, Jane, Jane, Adam, print 2.

(c) Print all the people in the list, sorted by age.

(d) Create a Map<Integer, List<Person>> that classifies people by age.

Lambdas and streams reference (read only if needed)

Kinds of lambda expressions:

- function: one input, one output
- predicate: one input, one boolean output
- binary operator: two inputs, one output, all of the same type
- consumer: one input, no returned output
- comparator: two inputs of same type, one int output

Intermediate stream operations of all streams:

- filter(predicate)
- map(function)
- distinct()
- sorted()
- limit(size)

Intermediate operations specific to Stream:

- sorted(comparator)
- mapToInt(function) [returns an IntStream; the provided function must be int-returning]

Terminal operations of all streams:

- count()
- forEach(consumer)
- reduce(initial, binaryOperator)
- findAny() [returns an optional]
- toArray()

Terminal operations specific to Stream:

- toList()
- min(comparator) [returns an optional]
- max(comparator) [returns an optional]
- collect(collector)

Terminal operations specific to IntStream:

- sum()
- average() [returns an optional]
- min() [returns an optional]
- max() [returns an optional]

The collect method takes a Collector specifying how the elements of the stream should be collected. To easily create a Collector, we can use the following methods of the Collectors class:

- toSet()
- joining()
- joining(String delimiter)
- groupingBy(function)

Some terminal operations return an Optional or OptionalInt. Here are some methods of all optionals:

- orElse(otherValue)
- orElseThrow()
- ifPresent(consumer)