CISC 3130

Data Structures
Spring '24
Midterm

Part I. Answer all questions (40 points). Place your answer in the box.

These questions are straight from CodeLab … you can get the correct answers by completing them there.

1. An array is an example of a

a. primitive type
b. class
c. homogeneous container
d. heterogeneous container

2. What is a possible opportunity for leveraging in a `Set` class?

a. using `contains` in `add`
b. using `add` in `contains`
c. using `get` to code `set`
d. using `set` to code `get`

3. Regarding bounds checking for a built-in array and a vector class

a. the legal upper bound for an array is the capacity, while for vector the size
b. the legal upper bound for both containers is the size
c. the legal upper bound for an array is the size, while for vector the capacity
d. the legal upper bound for both containers is the capacity

4. A typical constructor for an exception class

a. a `String` argument corresponding to a descriptive message
b. no argument ... exception classes only have default constructors
c. a `boolean` argument indicating whether the app should be terminated
d. has an `int` argument corresponding to the error code

5. In `checkCapacity`, the elements that are copied are

a. `0 .. size-1`
b. `0 .. size`
c. `0 .. capacity-1`
d. `0 .. capacity`

6. When should `checkCapacity` be called?

a. before any operation that increases the size of the container
b. before any operation that decreases the size of the container
c. before every element access
d. before printing the container

7. Given the following, which of the following might result in a runtime `ClassCastException`?

```
Object object;
String string;
Integer integer;
```

a. `object = integer;`
b. `object = (Integer)string;`
c. `string = object;`
d. `string = (String) object;`

8. Inserting an object into a JCF container involves a(n)

a. upcast
b. downcast
c. could be a downcast or an upcast
d. neither downcast nor upcast is involved

9. Given the following code, which of the following is legal?

```
class A {}
class B extends A {}
class C extends A {}
A a;
B b;
C c;
```

a. `a = new B();`
b. `b = new A();`
c. `b = new C();`
d. `c = new B();`

10. In the following, `E` is

```
class Vector<E> {...}
```

a. the capacity
b. the element type
c. the generic value
d. the type parameter

11. Programming to the interface

a. uses methods of the implementing class that are not in the interface
b. only uses methods specified in the interface
c. only creates objects of the interface type
d. specifies multiple interfaces for the same implementation type

12. The standard algorithmic complexity orders from best to worst (i.e., most efficient to least efficient) is

a. loglinear, quadratic, linear, constant, logarithmic
b. constant, linear, logarithmic, loglinear (n log n), quadratic
c. linear, constant, logarithmic, loglinear, quadratic
d. constant, logarithmic, linear, loglinear, quadratic

13. The following code:

```
for (int i = 0; i < arr.length)
       System.out.println(arr[i]);
```

a. is O(1)
b. is O(n)
c. is O(n log n)
d. is O(n^2)

14. The following code:

```
for (int i = 0; i < n; i++) {
   for (int j = 0; j <= i; j++)
      System.out.print(".");
   System.out.println();
}
```

a. is linear
b. is quadratic
c. is logarithmic
d. is constant

15. Which of the following statements is true

a. both sequences and associatives are containers based on content
b. sequences are containers based on position; associatives are based on content
c. sequences are containers based on content; associatives are based on position
d. sequence and associative categories have nothing to do with position or content

16. A queue implemented using an array with the front always at location 0 and a single index at the rear would have

a. constant insertion and constant deletion
b. linear insertion and linear deletion
c. constant insertion and linear deletion
d. linear insertion and constant deletion

17. Which of the following JCF classes would NOT be reasonably used as a stack?

a. `ArrayDeque`
b. `HashSet`
c. `ArrayList`
d. `LinkedList`

18. Which of the following is NOT an associative (value-based) container?

a. `Deque`
b. `TreeMap`
c. `HashSet`
d. `Bag`

19. Which implementation approach is most likely to result in an efficient (timewise) container?

a. independent
b. composition
c. inheritance
d. any of the above is as likely to be as efficient as any of the others

20. Which implementation approach is most likely to result in delegation methods?

a. composition
b. inheritance
c. independent
d. any of the above

Part II. Answer all questions (60 points)

21. **(20 points)** Here is the interface for the vector discussed in Lectures 1 and 2:

```
interface Vector<E> {
      E get(int index);
      void set(int index, E e);
      void add(E e);          // adds e to end of the vector
      int size();
}
```

Code a class `ArrayVector` that implements the above interface, using a built-in array as the underlying container. The class should:

- Be generic (just like the interface)
  - If you don't know how to code a generic class, you can make the element type an `int`
- Be resizeable
  - If don't know how to make it resizeable, you can either have the size be specified at runtime (in the constructor -- more partial credit) or compile-time (in the declaration – less partial credit).
- Throw appropriate exceptions
  - `throw new Exception("appropriate message");` is fine
- The class should also have a `toString` method

This is Vector Version 5 from Lecture 2

22. (**7 points**) Here are the some of the methods for the JCF `List` interface:

```
interface List<E> {
      E get(int index)
      void set(int index, E e);
      void add(E e);
      int size();
      boolean isEmpty();
}
```

and here is a method that performs a bubble sort on a built-in array

```
void bubble(E [] arr) {
     for (int last = arr.length-1; last > 0; last--)
          for (j = 0; j < last; j++)
               if (arr[j] > arr[j+1}) {
                    E temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
               }
}
```

a. Rewrite this method so it sorts a `List`, i.e., the method header is now:

```
void bubble(List list)
```

(you may not need all the methods specified in the above interface).

```
void bubble(List<E> list) {
   for (int last = list.size()-1; last > 0; last--)
         for (j = 0; j < last; j++)
              if (list.get(j) > list.get(j+1)) {
                   E temp = list.get(j);
                   List.set(j, list.get(j+1));
                   List.set(j+1, temp);
              }
}
```

I notice I didn't place the type parameter in the method header in the question (I inserted it in the solution); as a result any mistakes in generics are ignored. Also, any issues with > and .compareTo are ignored as well.

b. (**3 points**) Which JCF containers (i.e., concrete classes) can sorted using the above method? What do they have in common that allows them to be sorted by this method?

- **Any class that implements the `List` interface**
- **In the JCF, these are `ArrayList` and `LinkedList`.**

Either one of the above bullets – or both – is fine

23. (**5 points**) Explain why it makes sense to start a fixed-size, linear (not resizeable, and not circular) deque off in the middle of an array, while a FIFO queue should start off at the left of the array (think of the way they each grow and shrink). Pictures help; so does speaking in big-O notation!

Deques can grow from either end, therefore you want to start the container off with room to grow on both sides. FIFO queues on the other hand only grow at the end, so to maximize that, you want them starting off at the far left end.

Deque

Queue

In terms of big-O notation, if the end of the container (deque or FIFO queue) is up against the 'wall' of the underlying container (the array), the insertion becomes an O(n) operation (because of the necessary shift), but it there's room to insert and element, the operation is an O(1).

24. (**5 points**) Explain why a stack implemented using a 0-based array should be written to push/pop at the end of the array rather than the beginning. Your answer should be short (1 or two sentences) and employ algorithmic analysis (complexity) terminology, in particular big-O notation.

Inserting and removing elements from the end of an array is an inexpensive O(1) operation, while inserting an removing elements from the beginning (position) is a more expensive O(n) operation

25. (**10 points**) One can search a stack by having a second (temporary) stack hold the elements while one goes through the stack looking for the desired value). Once the value has been found (or not), the elements are restored to the original stack. Write a method that accepts a stack and a value and return whether the value is contained in the stack (i.e., `boolean search(stack, value)`). Upon return, the stack should remain unchanged. Pseudo-code or Java is fine. (If you can't do this, for partial credit write a method that accepts a stack as a parameter and empties the stack, i.e., `void clear(stack)`).

```
search(stack, val) {
      temp  = new Stack();

      found = false;
      while (!stack.isEmpty() && !found) {
            top = stack.pop();
            if (val == top) found = true;
            temp.push(top);
      }

      while (!temp.isEmpty())
            stack.push(temp.pop());
      return found;
}
```

Slightly shorter and more readable code if you used peek

26. (**5 points**) Here are subsets of the methods of the JCF `ArrayList` and `ArrayDeque` classes.

```
interface ArrayList {
   void add(int index, E e);
   E remove(int index);
   Boolean isEmpty();
}
```

```
interface ArrayDeque {
   void addFirst(E e);
   E removeFirst ();
   void addLast(E e);
   E removeLast();
   boolean isEmpty();
}
```

In Lab 5 you demo'd a FIFO queue using both an `ArrayDeque` and an `ArrayList` Which was easier to work with? Explain your answer.

Two possible answers, both basically the same:

- **If you followed the interface above, the idea is that `add` / `remove` of FIFO are nothing more than delegations to `addLast` / `removeFirst` (or vice versa) of `ArrayDeque`, but require a bit of logic (the addition of a position argument, size, and 0) when using an `ArrayList` – `add(size, …)` / `remove(0)`**
- **If you happened to actually do the Lab and/or rembered that `ArrayDeque` also has the FIFO `add`/`remove` implemented (it implements the `Queue` interface, but you were not expected to know that), then the delegation does not even require a name change to the methods.**

27. (**5 points**) We've when searching an array, using binary search is O(logn) while linear search is… well, linear (O(n)). Why are we not always using binary search then? Again, as much as possible, use the terminology of complexity, i.e., big-O notation.

Binary search requires the array be maintained in sorted order. If one does not plan on inserting many items compared to how often one searches, the overhead of inserting an item into the sorted list (an O(n) operation) may make sense in order to achieve an O(logn) lookup through binary search. If there will be many insertions relative to lookups, then it may not make sense to take the O(n) cost, and instead simply append the new item to the end (an O(1) operation) and take the hit on the lookup.