CS1102s Data Structures and Algorithms

14/3/2008

Examination Questions Midterm 2

This examination question booklet has 8 pages, including this cover page, and contains 12 questions.

You have 30 minutes to complete the exam. Use a B2 pencil to fill up the provided MCQ form. Leave Section A blank. Fill up Sections B and C.

After finishing, place the MCQ sheet on top of the question sheet and leave both on the table, when you exit the room.

Question 1: Consider the ADT List and its implementation `LinkedListReferenceBased` (page 245). What is displayed when executing the following program fragment?

```
ListInterface myList = new ListReferenceBased();
for (int i=1; i <= 3; i++) myList.add(1, "object␣"+i);   // corrected
for (int j=1; j <= 3; j++) System.out.println(myList.get(j));
```

1 [A] The order is not deterministic because the ADT List is position-oriented, and not value-oriented.

1 [B]
```
object 1
object 2
object 3
```

1 [C] The order is not deterministic because the list implementation used is reference-based, and not array-based.

1 [D]
```
object 3
object 2
object 1
```

Answer 1:

1 [D] `myList.add(1,...)` adds the new element *in front of* all other elements. Thus the numbers appear in reverse order in the list.

Question 2: Consider the following Java program fragment.
What is the result of executing it?

```
ListInterface myList = new ListReferenceBased();
String myString;
myString = "Some␣String";
myList.add(1,myString);
myList.remove(1);
System.out.println(myString);
```

2 [A] `Some String`

2 [B] `null`

2 [C] `null pointer exception`

2 [D] The program does cannot be compiled.

2 [E] `List index out of bounds on remove`

Answer 2:

2 [A] Strings are immutable. Putting them in a data structure (such as a list) does not change that. Even if the object would be mutable, the list data structure does not change the list item objects.

Question 3: Consider the following Java program fragment that uses the ADT Stack and its implementation `StackReferenceBased` (page 344). Assume that `getStack()` returns a `StackReferenceBased` object without error. What is displayed when executing the following program fragment?

```
StackReferenceBased myStack = getStack();
myStack.popAll();
myStack.push("first");
myStack.push("second");
Object x = myStack.peek();
myStack.pop();
System.out.println(myStack.pop());
```

3 [A] `first`

3 [B] `StackException on pop: stack empty`

3 [C] `second`

3 [D] `null`

3 [E] We cannot know what is printed, because we do not know how many objects the stack, which is returned by `getStack()`, contains.

Answer 3:

3 [A] Stacks are last-in-first-out data structures. Thus the first `pop` pops the string `second`, and the second `pop` pops `first`. Note that `popAll` ensures that the stack is empty at the beginning, and that `peek` does not change the stack.

Question 4: Consider the following Java program fragment that uses the ADT Queue and its implementation `QueueReferenceBased` (page 390). Assume that `getQueue()` returns a `QueueReferenceBased` object without error. What is displayed when executing the following program fragment?

```
QueueReferenceBased myQueue = getQueue();
myQueue.dequeueAll();
myQueue.enqueue("first");
myQueue.enqueue("second");
Object x = myQueue.peek();
myQueue.dequeue();
System.out.println(myQueue.dequeue());
```

4 [A] `first`

4 [B] `QueueException on dequeue: queue empty`

4 [C] `second`

4 [D] `null`

4 [E] We cannot know what is printed, because we do not know how many objects the queue, which is returned by `getQueue()`, contains.

Answer 4:

4 [C] Queues are first-in-first-out data structures. Thus the first `dequeue` returns the string `first`, and the second `dequeue` returns `second`. Note that `dequeueAll` ensures that the queue is empty at the beginning, and that `peek` does not change the queue.

Question 5: Consider the array-based implementation of the ADT Queue, `QueueArrayBased` (page 394). Which statement is true?

5 [A] None of the below.

5 [B] The operations `enqueue(...)`, `dequeue`, and `dequeueAll()` all take $O(1)$ time.

5 [C] The operations `enqueue(...)`, `dequeue`, and `dequeueAll()` all take $O(n)$ time, where n is the number of objects in the queue before the operation is called.

5 [D] The operations `enqueue(...)` and `dequeue` take $O(n)$ time, where n is the number of objects in the queue before the operation is called, and the operation `dequeueAll()` takes $O(1)$ time.

Answer 5:

5 [B] All queue operations in both implementations take $O(1)$ time.

Question 6: Consider the reference-based implementation of the ADT List, `ListReferenceBased` (page 245). Which statement is true?

6 [A] The operations `add(index,item)`, `remove(index,item)`, and `get(index,item)` all take $O(1)$ time.

6 [B] The operations `add(index,item)`, `remove(index,item)`, and `get(index,item)` all take $O(n)$ time, where n is the number of objects in the list before the operation is called.

6 [C] The operations `add(index,item)` and `remove(index,item)` take $O(n)$ time, where n is the number of objects in the list before the operation is called, and the operation `get(index,item)` takes $O(1)$ time.

6 [D] The operation `add(index,item)` takes $O($`index`$)$ time, and the operations `get(index,item)` and `remove(index,item)` take $O(1)$ time.

6 [E] The operations `add(index,item)`, `remove(index,item)`, and `get(index,item)` all take $O($`index`$)$ time.

Answer 6:

6 [E] The `index` determines how far the list is searched in order to apply the operation.

Question 7: Which statement on postfix and infix expressions (page 349) is false?

7 [A] Infix expressions that use addition and multiplication in general require parentheses, to indicate the order in which the operations need to be executed.

7 [B] A common way to evaluate an expression in postfix notation is to first convert it to an infix expression and then evaluate the infix expression.

7 [C] The postfix expression `364*+2*` evaluates to `54`.

7 [D] Postfix expressions do not have any parentheses.

7 [E] The infix expression `(4 + 7) * 3` evaluates to `33`.

Answer 7:

7 [B] Expressions in postfix notation are easier to evaluate (only one stack needed) than expressions in infix notation. Thus a translation from postfix to infix is not helpful.

Question 8: What statement on worst case complexity is false?

8 [A] If an algorithm is $O(5 \cdot n)$, it is also $O(n)$.

8 [B] If an algorithm is $O(n^2 + n)$, it is also $O(n^2)$.

8 [C] If an algorithm is $O(5)$, it is also $O(10)$.

8 [D] If an algorithm is $O(n \cdot (2 \cdot \log_2 n))$, it is also $O(n \cdot \log_2 n)$.

8 [E] If an algorithm is $O(2^n)$, it is also $O(n^2)$.

Answer 8:

8 [E] See page 471 for a discussion of the different classes. Also see page 472.

Question 9: The performance of `Quicksort` in practice depends on the choice of the pivot element (`choosePivot(..)` on page 498). Assume the following implementation:

```
private static void choosePivot(Comparable[] theArray, int first, int last) {
}
```

which means that the first element is the pivot element. Which statement on the performance of the resulting Quicksort algorithm is true?

9 [A] Sorting of unsorted "random" arrays is $O(n \cdot \log_2 n)$, the sorting of already sorted arrays is $O(n^2)$, and the sorting of arrays that are sorted in reverse order is $O(n \cdot \log_2 n)$.

9 [B] Sorting of unsorted "random" arrays is $O(n \cdot \log_2 n)$, the sorting of already sorted arrays is $O(n \cdot \log_2 n)$, and the sorting of arrays that are sorted in reverse order is $O(n^2)$.

9 [C] Sorting of unsorted "random" arrays is $O(n \cdot \log_2 n)$, the sorting of already sorted arrays is $O(n^2)$, and the sorting of arrays that are sorted in reverse order is $O(n^2)$.

9 [D] Sorting of unsorted "random" arrays is $O(n^2)$, the sorting of already sorted arrays is $O(n^2)$, and the sorting of arrays that are sorted in reverse order is $O(n^2)$.

9 [E] Sorting of unsorted "random" arrays is $O(n^2)$, the sorting of already sorted arrays is $O(n \cdot \log_2 n)$, and the sorting of arrays that are sorted in reverse order is $O(n \cdot \log_2 n)$.

Answer 9:

9 [C] "random" and "already sorted" is explained in the book. It is easy to see that the case "sorted in reverse order" is similar to "already sorted".

Question 10: Consider the following algorithm for sorting a given number n of distinct positive integers, whose values range from 1 to m, which is given in an array `theArray`.

```
int[] temp = new int[m+1];    // corrected
for (int i=0; i < n; i++)
   temp[theArray[i]] = theArray[i];
int insert=0;
for (int i=1; i <= m; i++)    // corrected
   if (temp[i] != 0) theArray[insert++] = temp[i];
```

Which of the following statements is true?

10 [A] This algorithm is $O(n)$.

10 [B] This algorithm is $O(m)$. (corrected)

10 [C] This algorithm is $O(n^2)$.

10 [D] This algorithm is $O(m^2)$.

10 [E] This algorithm is $O(m \cdot \log_2 n)$.

Answer 10:

10 [B] The second loop takes $O(m)$ time. Since the given items are distinct, we have $n < m$ and thus overall, $O(m)$ captures the complexity in the best way.

Question 11: The space consumption of sorting algorithms is sometimes an important consideration. In particular, some sorting algorithms require the creation of additional temporary arrays to hold intermediate results. Which statement is correct?

11 [A] Bubble sort requires no additional arrays, whereas selection sort, insertion sort, Merge sort and Quicksort all require additional arrays.

11 [B] Bubble sort and selection sort require no additional arrays, whereas insertion sort, Merge sort and Quicksort all require additional arrays.

11 [C] Bubble sort and insertion sort require no additional arrays, whereas selection sort, Merge sort and Quicksort all require additional arrays.

11 [D] Selection sort, bubble sort, and insertion sort require no additional arrays, whereas Merge sort and Quicksort require additional arrays.

11 [E] Selection sort, bubble sort, insertion sort, and Merge sort require no additional arrays, whereas Quicksort requires additional arrays.

Answer 11:

11 [D] The given implementations for selection sort, bubble sort, and insertion sort do not use `new`, whereas the implementations of Merge sort and Quicksort do.

Question 12: How many calls of `mergesort` (page 489) are performed when sorting an array of size 64?

12 [A] 128

12 [B] 2^{63}

12 [C] 127

12 [D] 63

12 [E] 64

Answer 12:

12 [C] When the given array has the size of a power of 2, the algorithm always splits the array exactly in half. The result is call tree which is a full binary tree, with the leaves represented by the array entries. Thus there are 64 leaves, and 63 inner nodes. (If you want, you can show that any binary tree has one more leaf than it has inner nodes.)