

Read each question carefully. When writing code, note that not all questions require a complete class, or even a complete method. Keep in mind that this review is not intended to be comprehensive. While studying for the exam, it would be prudent to consider variations on the questions presented here.

Under what circumstances would it be preferable to implement a sequence using a doubly linked list rather than an array? Specifically which operations would be more efficient & under what circumstances.

A linked list is more efficient when there are frequent capacities changes as the cost for insertion & deletion at the cursor, head or tail of the list are  $O(1)$ . Read/write at the cursor are also efficient,  $O(1)$ . A linked list is ill-suited to scenarios where the majority of operations call for random access as such operations are on average linear.

Write a generic method called **reverse** that takes an array and reverses it.

```
/**
 * reverses an array
 * @param array - the array to reverse
 * @return an array that is has the reverse order contents of array
 * note: object references are shallow copies, note deep clones!
 */
public <G> G[] reverse(G[] array)
{
    G[] reversedArray = (G[])(new Object[array.length]);
    for(int a=0; a<array.length; a++)
    {
        reversedArray[(array.length-1)-a] = array[a];
    }
    return reversedArray;
}
```

```
// sneaky alternate version that avoids compiler warnings
public <G> G[] reverse(G[] array)
{
    G[] reversedArray = array.clone();
    for(int a=0; a<array.length; a++)
    {
        reversedArray[(array.length-1)-a] = array[a];
    }
    return reversedArray;
}
```

In big O notation indicate the performance of the following operations of IntLinkedListBag:

```
// add a new item to the bag  
add(int item)
```

O(1)

```
// remove the all occurrences of the item from the bag  
// return success (if it was found) or failure (if it wasn't there)  
remove(int item)
```

O(n)

```
// count the number of times item is in the bag  
countOccurrences(int item)
```

O(n)

```
// get the total number of items in the bag  
size()
```

O(1) if stored as an instance variable, O(n) if calculated dynamically

```
// make a new bag identical to the combination of b1 & b2  
union(IntArrayBag b1, IntArrayBag b2)
```

$O(n_1 + n_2)$  where  $n_1$  is the size of bag 1 &  $n_2$  is the size of bag 2

In big O notation indicate the performance of the following operations of IntLinkedListSequence (assume the list is NOT doubly linked)

```
// adds an item to the sequence after the cursor  
addAfter(int item)
```

O(1)

```
// adds an item to the sequence before the cursor  
addBefore(int item)
```

O(1) assuming there is a precursor, O(n) otherwise

```
// count the number of times item is in the list  
countOccurrences(int item)
```

O(n)

```
// create a new duplicate sequence & return it  
clone()
```

O(n)

```
// get the current number of items in the list  
size()
```

O(1) if stored as an instance variable, O(n) if calculated dynamically

In big O notation indicate the performance of the following operations of IntDoublyLinkedListSequence (assume the list IS doubly linked)

```
// adds an item to the sequence after the cursor  
addAfter(int item)
```

O(1)

```
// adds an item to the sequence before the cursor  
addBefore(int item)
```

O(1)

```
// count the number of times item is in the list  
countOccurrences(int item)
```

O(n)

```
// get the current number of items in the list  
size()
```

O(1) if stored as an instance variable, O(n) if calculated dynamically

In big O notation indicate the performance of the following operations of IntArrayStack where the top of the stack is the zero element of the array:

```
// look at the top element
peek()
```

O(1)

```
// remove the top most element
pop()
```

O(n)

```
// place a new element onto the stack
push()
```

O(n)

In big O notation indicate the performance of the following operations of IntArrayStack where the bottom of the stack is the zero element of the array:

```
// look at the top element
peek()
```

O(1)

```
// remove the top most element
pop()
```

O(1)

```
// place a new element onto the stack
push()
```

O(1)

Assume you were reading in a series of paren ( ) and square bracket [ ] characters one at a time. Describe the steps of an algorithm which uses a stack to determine at each step if the string is balanced.

Below are 6 example groups of input & output:

```
input:  ( )[]      ( [ ] ] ) ( )      ( ( ) [ ] ( ) ]      ] ( ) [ ]      ( ) ( [ ] ]      ( ) [ ] [ ( )
output: FTFT      FFFFFTFT      FFFFFFFFFFT      FFFFF      FTFFFF      FTFTFFF
```

let s be the stack of characters

1. get character c

```
if((s.peek( ) == '(' and c == ')') or (s.peek( ) == '[' and c == ']'))
{
```

```
    s.pop( )
    discard c
```

```
}
```

```
else
```

```
{
    s.push( c )
}
```

2.

```
if(s.size() == 0)
    print "T"
```

```
else
```

```
    print "F"
```

3. goto step 1

Assume you had a doubly linked list. Describe the steps necessary to add a new node before the cursor.

```

let c be the cursor node & x be the new node to add to the sequence

if( c.backwardLink != null)
{
    let p = c.backwardLink()
    p.forwardLink = x
    x.backwardLink = p
    x.forwardLink = c
    c.backwardLink = x
}
else
{
    x.backwardLink = null
    x.forwardLink = c
    c.backwardLink = x
    head = x
}

```

initial links ←——

final links ←-----

Assume you had a singly linked list. Describe the steps necessary to move the cursor back 1 element (towards the head)

```

let h be the head node, c be the cursor

if( c == h )
    throw an exception!

Let node x = c;
c = head;
while(c.getLink() != x)
{
    c = c.getLink();
}

```

What are comparators? Why are they useful? Under what conditions are they a better choice than using the Comparable interface?

Comparators are objects (or possibly classes) that implement the Comparator interface & are used to compare two things of a given type.

They are useful because they allow us to write code that is generic with regards to the comparison. For instance, sorting depends on comparing two things & determining which of the two is greater – how this decision is made can be separated out using a Comparator. This can reduce the amount of code we need to write.

Using Comparators is preferable to using the Comparable interface if we can't change the code of the things being compared or if there is more than one way we wish to compare the given items.

What is the advantage of using a generic Bag class instead of a Bag class designed to hold Objects?

The main advantage is that it provides us with a way to code flexible yet type safe collections (in this case, a Bag). Thus, when using a generic bag, the compiler will stop us from placing unrelated heterogeneous elements into the bag. For example, it prevents us from directly adding a String to Bag<Double>. The only way we could get the String into Bag<Double> would be with casting.

Comparators are best suited to situations where:

- A. There is a natural ordering of elements.
- B. The ordering of elements is strict I.E.  $a < b$  is allowed, but  $a \leq b$  is not
- C. You have access to the source code for the elements being compared
- D. The ordering of elements is unimportant
- E. You need to deviate from the natural ordering of elements.
- F. None of the above

In linked lists, dummy nodes are:

- A. Nodes that serve as place holders for head & tail
- B. Nodes that are abandoned by the remove method
- C. Nodes that serve as place holders for cursor & precursor
- D. Nodes that only occur in a doubly linked list
- E. Nodes that link to null
- F. None of the above

To make full use of an external iterator, a container class must implement interface(s)

- A. iterator
- B. iterable
- C. both iterator & iterable are required
- E. either iterator or iterable will work
- D. no interface implementation is required
- F. none of the above

To make use of an external iterator, a class designed to print the contents of a container class must implement interface(s)

- A. iterator
- B. iterable
- C. both iterator & iterable are required
- E. either iterator or iterable will work
- D. no interface implementation is required
- F. none of the above

Compared to a linked list, an array based list is uniquely suited for:

- A. lots of insertions & removals at or after the cursor
- B. lots of insertions & removals throughout the list
- C. lots of read operations at or after the cursor
- D. lots of read operations throughout the list
- F. lots of operations of any type at the head of the list
- F. none of the above