

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 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 items 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 bag  
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 LinkedListStack where the top of the stack is the tail of the linked list (assume the list is NOT doubly linked):

```
// 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(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
}
    
```

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.

Examples:

input: ( ) [ ]	( [ ] [ ] ) ( )	( ( ) [ ] [ ] ) ( )	] ( ) [ ]	( ) ( [ ] )	( ) [ ] [ ] ( )
output: FTFT	FFFFFTFT	FFFFFFFFFT	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
    
```

What are iterators? Why are they useful? In terms of code, what is required to make use of them?

Iterators are objects used to traverse and sometimes modify a collection.

They are useful because they allow us to write code that is generic with regards to the collection being operated upon. For instance, we could write a method that takes an iterator & prints off the contents of a collection – this method would then work with any collection (bag, linked list, array based sequence, Java ArrayList) that is designed to provide iterators to itself.

In terms of code, two things are required. First, the collection class of interest must implement the Iterable interface, indicating that it will provide an iterator when requested to do so. Second, there needs to be a companion iterator class which implements the Iterator interface, indicating that it can return an element from the collection & check to see if there is currently an element to return. It also has a method to remove the current element, though this is not always safe to call.

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.

Entries in a stack are "ordered". What is the meaning of this statement?

- A. A collection of Stacks can be sorted.
- B. Stack entries may be compared with the '<' operation.
- C. The entries must be stored in a linked list.
- D. There is a first entry, a second entry, and so on.
- E. The entries rely on a Comparator class
- F. None of the above

Which of the following stack operations could result in stack underflow?

- A. isEmpty
- B. pop
- C. push
- D. clone
- E. Two or more of the above answers
- F. None of the above

When using a doubly linked list for a stack, what is the run-time behavior for the pop method?

- A. linear - but only if the head is used as the stack top
- B. linear - but only if the tail is used as the stack top
- C. linear - regardless of whether stack top is the tail or the head
- D. constant - but only if the tail is used as the stack top
- E. constant - regardless of whether stack top is the tail or the head
- F. none of the above

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