COMP2402: Mid-term Review Questions

October 20, 2010

1 Java Collections Framework – Interfaces

All of these questions should be considered in the context of the interfaces in the Java Collections Framework

1. Explain the differences and similarities between a `Set` and a `List`
2. Explain the difference between a `Collection` and a `Map`. Could it also make sense to have `Map` be a subclass of `Collection`?
3. Which of the JCF interfaces would be the most useful if we want to store a collection of students enrolled in COMP2402 so that we can quickly check if a student is enrolled in COMP2402?
4. How does your answer to the previous question change if we also want to be able to quickly output a list of students, sorted by `(lastname,firstname)`
5. How does your answer to the previous question change if we also want to store some auxiliary information (e.g., a mark) with each student.
6. A `Bag` is like a set except that elements can be stored more than once. Explain how you could implement a `Bag` using a `Map`.
7. Explain the differences between an `Iterator` and a `ListIterator`.

2 Java Collections Framework – Implementations

1. Explain why it is important that elements that are stored in a `Set` or `Map` aren't modified in a way that affects the outcome of the equals() method.
2. Explain why you would choose a `LinkedHashSet` (or `LinkedHashMap`) over a `HashSet` (or `HashMap`, respectively).
3. Describe the running time of the methods `get(i)` and `set(i,x)` for an `ArrayList` versus a `LinkedList`
4. Describe the running time of the method `add(i,x)` for an `ArrayList` versus a `LinkedList`
5. Explain why it is possible to quickly make a lot of modifications in the interior (not near either end) of a `LinkedList` but this is not possible in an `ArrayList`.
6. For each of the following methods, decide if it is fast or slow when (a) `l` is an `ArrayList` and (b) when `l` is a `LinkedList`.

```java
public static void frontGets(List<Integer> l, int n) {
  for (int i = 0; i < n; i++) {
    l.get(0);
  }
}

public static void backGets(List<Integer> l, int n) {
  for (int i = 0; i < n; i++) {
    l.get(l.size()-1);
  }
}

public static void randomGets(List<Integer> l, int n) {
  Random gen = new Random();
  for (int i = 0; i < n; i++) {
    l.get(gen.nextInt(l.size()));
  }
}

public static void insertAtBack(List<Integer> l, int n) {
  for (int i = 0; i < n; i++) {
    l.add(new Integer(i));
  }
}

public static void insertAtFront(List<Integer> l, int n) {
  for (int i = 0; i < n; i++) {
    l.add(0, new Integer(i));
  }
}

public static void insertInMiddle(List<Integer> l, int n) {
  for (int i = 0; i < n; i++) {
    l.add(new Integer(i));
  }
  for (int i = 0; i < n; i++) {
    l.add(n/2+i, new Integer(i));
  }
}

public static void insertInMiddle2(List<Integer> l, int n) {
  for (int i = 0; i < n; i++) {
    l.add(new Integer(i));
  }
  ListIterator<Integer> li = l.listIterator(n/2);
  for (int i = 0; i < n; i++) {
    li.add(new Integer(i));
  }
}
```

3 Lists as Arrays

These questions are all about implementing the `List` interface using arrays.

3.1 ArrayStacks

Recall that an `ArrayStack` stores `n` elements in a backing array `a` at locations `a[0]`,...,`a[n-1]`:

```
public class ArrayStack<T> extends AbstractList<T> {
  T[] a;
  int n;
  ...
}
```

1. Describe the implementation and running times of the operations `get(i)` and `set(i,x)` in an `ArrayStack`.
2. Recall that the length of the backing array `a` in an `ArrayStack` doubles when we try and add an element and `n+1 > a.length`. Explain, in general terms why we choose to double rather than just add 1 or a constant.
3. Recall that, immediately after an `ArrayStack` is resized by `grow()` or `shrink` it has `a.length = 2*n`.
   (a) If are currently about to grow the backing array `a`, what can you say about the number of `add()` and `remove()` operations since the last time the `ArrayStack` was resized?
   (b) Recall that we shrink the back array `a` when `3*n < a.length`. If are currently about to shrink the backing array `a`, what can you say about the number of `add()` and `remove()` operations since the last time the `ArrayStack` was resized?
4. From the previous question, what can you conclude about the total number of elemnts copied by `grow()` and `shrink()` if we start with an empty `ArrayStack` and perform m `add()` and `remove` operations.
5. What the amortized (or average) running time of the `add(i,x)` and `remove(i)` operations, as a function of `i` and `size()`.
6. Why is the name `ArrayStack` a suitable name for this data structure?

4 ArrayDeques

Recall that an `ArrayDeque` stores `n` elements at locations `a[j]`, `a[(j+1)%a.length]`,...,`a[(j+n-1)%a.length]`:

```
public class ArrayDeque<T> extends AbstractQueue<T> {
  T[] a;
  int j;
  int n;
  ...
}
```

1. Describe, in words, how to perform an `add(i,x)` operation (a) if `i < n/2` and (b) if `i >= n/2`
2. What is the running time of the `add(i,x)` and `remove(i)` operations, as a function of `i` and `size()`?
3. Describe, in words, why using `System.arraycopy()` to perform shifting of elements in the `add(i,x)` and `remove(i)` operations is so much more complicated for an `ArrayDeque` than an `ArrayStack`.
4. Explain why, using an example, if `a.length` is a power of 2 then `x mod a.length == x & (a.length-1)`. Why is this relevant when discussing `ArrayDeques`

4.1 DualArrayDeques

Recall that a `DualArrayDeque` implements the `List` interface using two `ArrayStacks`:

```
public class DualArrayDeque<T> extends AbstractList<T> {
  ArrayStack<T> front;
  ArrayStack<T> back;
  ...
}
```

1. If the elements of the list are $x_0, \ldots, x_{n-1}$, describe how these are distributed among `front` and `back` and in what order the appear.
2. Recall that we rebalance the elements among `front` and `back` when `front.size()*3 < back.size()` or vice versa. After we rebalance, we have `front.size() == back.size()` $\pm$ `1`. What does this tell us about the number of `add()` and `remove()` operations between two consecutive rebalancing operations. (See page 39 of arrays-ii.pdf).

4.2 RootishArrayStacks

Recall that a `RootishArrayStack` stores a list in a sequence of arrays (blocks) of sizes 1, 2, 3, 4,....

```
public class RootishArrayStack<T> extends AbstractList<T> {
  List<T[]> blocks;
  int n;
  ...
}
```

1. If a `RootishArrayStack` has r blocks, then how many elements can it store?
2. Explain how this leads to the equation

$$b(b+1)/2 \leq i+1 \leq (b+1)(b+2)/2$$

   that tells us the index of the block b that contains list element i.
3. In a `RootishArrayStack` that contains `n` elements, what is the maximum amount of space used that is not dedicated to storing data?

5 Linked Lists

5.1 Singly-Linked Lists

Recall our implementation of a singly-linked list (`SLList`):

```
protected class Node {
  T x;
  Node next;
}
public class SLList<T> extends AbstractQueue<T> {
  Node head;
  Node tail;
  int n;
  ...
}
```

1. Draw a picture of an `SLList` containing the values `a`,`b`,`c`, and `d`. Be sure to show the `head` and `tail` pointers.
2. Consider how to implement a `Queue` as an `SLList`. When we enqueu (`add(x)`) an element, where does it go? When we dequeue (`remove()`) an element, where does it come from?
3. Consider how to implement a `Stack` as an `SLList`. When we push an element where does it go? When we pop an element where does it come from?
4. How quickly can we find the ith node in an SLList?
5. Explain why we can't have an efficient implementation of a `Deque` as an `SLList`.

5.2 Doubly-Linked Lists

Recall our implementation of a doubly-linked list (`DLList`):

```
protected class Node {
  Node next, prev;
  T x;
}
public class DLList<T> extends AbstractSequentialList<T> {
  protected Node dummy;
  protected int n;
  ...
}
```

1. Explain the role of the `dummy` node. In particular, what are `dummy.next` and `dummy.prev`?
2. One of the following two functions correctly adds a node `u` before the node `p` in `DLList`, the other one is incorrect. Which one is correct?

```
protected Node add(Node u, Node p) {
  u.next = p;
  u.prev = p.prev;
  u.next.prev = u;
  u.prev.next = u;
  n++;
  return u;
}
protected Node add(Node u, Node p) {
  u.next = p;
  u.next.prev = u;
  u.prev = p.prev;
  u.prev.next = u;
  n++;
  return u;
}
```

3. What is the running-time of `add(i,x)` and `remove(i)` in a `DLList`? Hint: It depends on `i` and `size()`.

5.3 Memory-Efficient Doubly-Linked-Lists

Recall that a memory efficient doubly-linked list implements the `List` interface by storing a sequence of blocks (arrays) each containing `b`±`1` elements.

1. What is the running-time of `get(i)` and `set(i)` in a memory-efficient doubly-linked list? (Hint: It's a function of `i`, `size()`, and `b`.)
2. What is the amortized (or average) running time of the `add(i)` operation in a memory-efficient doubly-linked list?
3. In a memory-efficient doubly-linked list containing `n` elements, what is the maximum amount of space that is not devoted to storing data?

6 Hash tables

1. If we place n distinct elements into a hash table of size m using a good hash function, how many elements do we expect to find in each table position?
2. Recall the multiplicative hash function `hash(x) = (x.hashCode() * z) >>> w-d`.
   (a) In 32-bit Java, what is the value of `w`?
   (b) How large is the table that is used with this hash function? (In other words, what is the *range* of this hash function?)
   (c) Write this function in more standard mathematical notation using the mod and div (integer division) operators.
3. Explain the relationship between an class' `hashCode()` method and its `equals(o)` method.
4. Explain, in words, what is wrong with the following `hashCode()` method:

```
public class Point2D {
  Double x, y;
  ...
  public int hashCode() {
    return x.hashCode() ^ y.hashCode();
  }
}
```

   Give an example of many points that all have the same `hashCode()`.

5. Explain, in words, what is wrong with the following `hashCode()` method:

```
public class Point2D {
  Double x, y;
  ...
  public int hashCode() {
    return x.hashCode() + y.hashCode();
  }
}
```

   Give an example of 2 points that have the same `hashCode()`.