# Final Review Questions

## 1: JCF & Big-Oh

### 1.1 Big-Oh Notation

1. What is O(n)? What kind of object is it?

   It is a set of all functions..

   O(f(n)) denotes the set of functions g(n) where c•f(n) starts to dominate at large enough n.

   It describes the limiting behaviour of a function. In terms of algorithms, it describes the worst case runtime. It is a set of all functions that are larger than the runtime of an algorithm, at sufficiently large n.

   It describes the growth rate of an algorithm's runtime as the number of elements increases.

2. What does the statement 2n + 3 = O(n^2) mean?

   It means that the function 2n+3 is a member of the Set O(n^2)

   It means that for a given value k, 2n + 3 ≤ k•n^2

3. Recall the basic big-O hierarchy: for any constants a,b > 1:

   ```
   O(a) < O(log n) < O(n) < O(n•log n) < O(n^b) < O(c^n)
   ```

   Where do the functions 2n^2 = O(n^b), 100nlogn = O(n•log n), and log(n^3) = O(n•log n) fit into this hierarchy?

4. What is the running time (in big-o notation) for the following function, assuming the method foo() runs in constant time?

```
for(int i=1; i<n; i++)           // n times =>O(n)
    for(int j=n-i; j<n; j++)     // n/2 times =>
O(n)
        foo();                   // O(1)
```

   Thus, this function has a runtime that is O(n^2).

## 1.2 Java Collections Framework - Interfaces

All of these questions should be considered in the context of the interfaces in the JCF

1. Explain the differences and similarities between a Set and a List.

   A Set has no duplicates, whereas a List may have duplicates. A List has an order to its elements, wheras a set is unordered.

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 store some auxiliary information (e.g., a mark) with each student.

5. 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.

## 1.3 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. Describe the running time of the methods get(i) and set(i,x) for an ArrayList versus a LinkedList.

3. Describe the running time of the method add(i,x) for an ArrayList versus a LinkedList.

4. 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.

```
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));
        }
    }
```

## 2: Lists as Arrays

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

### 2.1 ArrayStacks

Recall that an ArrayStack stores n elements in a backing array a at locatiosn 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 to add an element and n+1> a.length. Explain, in general termswhy we choose to double rather than just add 1 or a constant.

3. Recall that, immediately after an ArrayStack is resized it has a.length = 2*n.

   **a.** If we 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 we 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 elements copied by resize() if we start with an empty ArrayStack and perform m add() and remove() operations?

5. What is 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?

## 2.2 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 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?

## 2.3 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 x0, ... xn-1, describe how these are distributed among front and back and in what order they 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() +/- 1. What does this tell us about the number of add() and remove() operations between two consecutive rebalancing operations?

   This means that we had at least n/2 calls to add() / remove().

   - assume initially balanced arrays
   - each array has n/2
   - after rebalancing, front array has n/4, and back array has 3n/4

- this means that there was n/2 add() / remove(), (where n is the final size)

### 2.4 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 < i+1 < (b+1)(b+2)/2
   ```

3. In a RootishArrayStack that contains n elements, what is the maximum amount of space used that is not dedicated to storing data?

## 3: Linked Lists

### 3.1 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 enqueue (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 using an SLList.

## 3.2 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 the 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?

## 3.3 Space-Efficient Doubly-Linked-Lists

Recall that a space 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 space-efficient doubly-linked list?

2. What is the amortized running time of the add(i) operation in a space-efficient doubly-linked list?

3. In a space-efficient doubly-linked list containing n elements, what is the maximum amount of space that is not devoted to storing data?

# 4: SkipLists

Recall that a skiplist stores elements in a sequence of smaller and smaller lists L0, ..., Lk. Li is obtained from Li-1 by tossing a coin for each element in Li-1 and including the element in Li if that coin comes up heads.

1. Draw an example of a skiplist select a few elements and show the search paths for these elements

2. Explain how the reverse search path is related to the following experiment: Toss a coin repeatedly until the first time the coin comes up heads.

3. If there are n elements in L0, what is the expected number of elements in L1? What about in Li?

4. If there are n elements in L0, give an upper bound on the expected length of the search path for any particular element.

5. Explain, briefly, how a skiplist can be used to implement the SortedSet interface. What are the running times of operations add(x), remove(x), and contains(x)?

6. Explain, briefly, how a skiplist can be used to implement the List interface. What are the running times of the operations add(i,x), remove(i), get(i,x), and set(i,x).

# 5: 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 expected 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 a class's 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 different points that have the same hashCode().

# 6: Binary Trees

## 6.1 Binary Trees

1. Draw a good sized binary tree.
2. Label the nodes of your drawing with their depth.
3. Label the nodes by the order they are processed in a preorder traversal.
4. Label the nodes by the order they are processed in an inorder traversal.
5. Label the nodes by the order they are processed in a postorder traversal.
6. Label the nodes of your tree with the sizes of their subtrees
7. What kind of traversal would you do to compute the sizes of all subtrees in a tree?

## 6.2 Binary Search Trees

1. Consider an array containing the integers 0,...,15 in sorted order. Illustrate the operation of binary search on a few (a) integer values and a few (b) non-integer values.
2. Draw a binary search tree containing (at least) 0,...,15
3. Show the search path for a value x in the tree and a value x' not in the tree
4. Insert some value x' into the tree
5. Delete some value x from your tree (try this for external nodes, internal nodes with 1 child, and internal nodes with two children).

# 7: Treaps

1. Choose a permutation of the values 0,...15. Draw the binary search tree that results from inserting the elements of your permutation in order. Are there other permutations that could have given the same search tree?

2. Explain the relationship between quicksort and random binary search trees.

3. Define the heap property for how priorities are used to make a binary tree into a Treap.

4. Pick some random numbers p0, ..., p15, and draw the treap that contains 0,...,15 where pi gives the priority for number i

5. Explain the relationship between random binary search trees and Treaps.

6. Explain the relationship between insertion and deletion in a Treap.

# 8: Scapegoat Trees

Recall that a scapegoat node is a node v = u.parent where 2•size(u) > 3•size(u.parent).

1. In a scapegoat tree, is size(u) < size(u.parent) for every non-root node u?

2. Draw an example of a tree that looks surprisingly unbalanced but is still a valid scapegoat tree.

3. Exaplain why, in a scapegoat tree, we keep two separate counters n and q that both -sort of- keep track of the number of nodes?

4. If v is a scapegoat node (for example, because 3*size(v.left)* > *2*size(v)) then explain how many operations (insertion/deletions) have affected v's subtree since the last time the subtree containing v

was rebuilt.

# 9: 2-4 and Red-Black Trees

Advantages of 2-4: log n height **always**; deterministic. Treap have randomness. Scapegoat has amortized cost.

1. Draw an interesting example of a 2-4 tree.
2. Explain what happens in a 2-4 tree when an insertion causes a node to have more than 4 children.
3. Explain what happens in a 2-4 tree when a deletion causes a node to have only one child.
4. Draw a red-black tree that corresponds to your 2-4 tree
5. Explain the relationship between the red-black tree properties and the 2-4 tree properties.

# 10: Heaps

## 10.1 Binary Heaps

These questions are about complete binary heaps represented using the Eytzinger Method.

1. Draw an example of a complete binary heap with key values.
2. Illustrate how your example's values map into an array using the Eytzinger Method.
3. Consider a binary heap represented using the Eytzinger Method. Give the formulas for the parent, left child, and right child of the value stored at position i.
4. When we perform a DeleteMin (remove()) operation on a heap, where do we get the value to replace the root, and what do we do with it?

5. Explain or illustrate the insertion algorithm for a heap

6. Explain the operation of HeapSort

## 10.2 Randomized Meldable Heaps

1. Draw two examples of heaps (not necessarily complete) and show how to meld (merge) them.

2. Explain how the add(x), and remove() operations can be done using the meld() operation.

## 11: Sorting Algorithms

1. Explain the relative advantages of MergeSort, HeapSort, and Quicksort. Things to keep in mind are (a) the number of comparisons performed, (b) the amount of extra memory required by the algorithms, and (c) whether their running time is guaranteed or only probabilistic.

   Stable [if order of duplicates is preserved]? Inplace [if elements are copied to new memory blocks]? Which has lowest number of comparisons?

   **Mergesort:** Stable but not in place. n•log n comparisons. Great unless memory restriction. [best in terms of comparisons]

   **Heapsort:** Not stable, but in place. 2 comparisons

   **Quicksort:** Not stable, and in place. 2•ln8 [4.16] comparions[??]. It has largest worst time, but good average time.

2. Create a random permutation of numbers 0,...,15. Walkthrough the sorting of this permutation using the HeapSort, MergeSort, QuickSort, and CountingSort algorithms.

3. Explain how the Radix sort algorithm works. What key feature of the counting sort algorithm makes it work correctly?

**Radix sort:** Best when you have a spart array, or large range with many zeros.

It uses counting sorts in sections of size *d*; Sort least significant bits first using counting sort, so that next significant bit duplicates are already sorted...

# 12: Graphs

1. Draw an interesting directed graph (with at least 6 nodes).
2. How many entries are in the Adjacency List representation of this graph? (Can you know without listing them?)
3. How many zeroes are in the Adjacency Matrix representation of this graph?
4. Walkthrough the Breadth-first and Depth-first traversals of this graph. Of the two trees that these traversals create, which has a greater height?