THE UNIVERSITY OF WESTERN AUSTRALIA

FACULTY OF
Engineering, Computing
and Mathematics

School of Computer Science & Software Engineering

SEMESTER 1 EXAM 2014

DATA STRUCTURES AND ALGORITHMS 2200 (CITS2200)

SURNAME: _____ STUDENT NO: _____

GIVEN NAMES: _____ SIGNATURE: _____

This paper contains **5 questions** and **8 pages**. Candidates should attempt all questions.
5 Questions 20 marks each
TOTAL MARKS: 100 marks

All questions are to be answered in the space provided in the question paper.
The space provided indicates the expected length of answer.
You may use point form to answer the questions.
If extra space is required, you may request an additional answer book.
Calculators are permitted, though not necessary.

This page is deliberately left blank

1.

(a) Explain the following concepts and give an example of each:

(i) Generics (in Java)

(ii) Dynamic Programming

(iii) Expected Case Complexity

(6)

(b) Describe how amortized case analysis differs from worst case analysis and provide an example.

(3)

(c) Describe as clearly as possible what it means for $f(n)$ to be $O(g(n))$ but $g(n)$ is not $O(f(n))$ and give an example of how they differ.

(3)

(d) Give pseudo-code for the Merge-Sort algorithm.

(4)

(e) Show that Merge-Sort runs in time $O(n \lg n)$ where n is the number of elements being sorted.

(4)

2.

(a) A Priority Queue is specified as follows

□ **Constructors**

(i) *PQueue(max)*: create a priority queue, with maximum size *max*

□ **Checkers**

(ii) *isEmpty()*: returns *true* if the queue is empty, *false* otherwise.

(iii) *isFull()*: returns *true* if the queue is full, *false* otherwise.

(iv) *examine()*: returns the element at the front of the priority queue.

□ **Manipulators**

(v) *enqueue(e, p)*: Inserts the element e into the priority queue with priority p. The element e is placed in front of all elements with priority q where $q > p$, but behind all other elements with priority q where $q <\leq p$. It throws an Overflow exception if the table is full.

(vi) *dequeue()*: removes and returns the element e, at the front of the queue. If the queue is empty, it throws an Underflow exception.

Write a block implementation of the PQueue ADT. You do not have to give lg time implementations of the methods.

(15)

(b) Describe an alternative implementation of a priority queue that can guarantee $O(\lg n)$ performance for enqueue and dequeue operations and justify the $O(\lg n)$ complexity of these methods.

(5)

3.

(a) Describe, as clearly as possible, the problem that the Bellman-Ford Algorithm is designed to solve.

(4)

(b) Give the pseudo-code for the Bellman-Ford Algorithm.

(5)

(c) Show that the Bellman-Ford Algorithm is correct.

(5)

(d) A `Map` class may be implemented in a number of ways. Given a map with comparable keys `K`, mapping to values `V`, suppose that we want to rank implementations, so that one implementation is better than another if its worst case performance is better, or if its worst case performance is the same, but its expected case performance is the same. Rank the following implementations from best to worst, explaining your reasoning. (It is possible that two implementations may have equal rank).

(i) A hashmap using separate chaining.

(ii) A hashmap using dynamic tables.

(iii) A self-balancing tree map, such as a red-black tree, ordered by `K`

(iv) A binary search tree of pairs, ordered by `K`.

(6)

4.

A recursive implementation of an immutable binary tree, `BTree`, with an inner classes, `MyIterator` and `Node`, is defined with the following declarations, member variables, and constructors:

```
public class BTree<E> {
  private Node<E> root;

     //empty constructor. makes an empty tree
  public BTree() {root = null;}

    //constructor for non-empty tree
  public BTree(E item, BTree<E> left, BTree<E> right){
     root = new Node<E>(item, left.root, right.root);
  }

  public Boolean isEmpty(){
    return root == null;
  }

  public E getItem() {
    return root.item;
  }

  public Iterator<E> iterator() {return new MyIterator();}

  private class MyIterator implements Iterator<E> {
    //missing code
  }

  private class Node<E> {
    E item; Node<E> left; Node<E> right;

    public Node(E item, Node<E> left, Node<E> right){
      this.item = item; this.left = left; this.right = right;
    }
  }
}
```

(a) Provide an Iterator for the `BTree` class. The iterator should provide methods:

```
//returns true if and only there are elements left to iterate
public boolean hasNext();

//returns the next element, or throws a NoSuchElementException if there is none
public E next() throws NoSuchElementException;
```

The iterator should traverse the elements of the tree in a level-order traversal. You may assume java.util classes or similar data structures are available.

(10)

(b) Provide Java code to test the iterator. The code should build a tree with at least 5 nodes, use an iterator to print all the elements out, and confirm calling `next()` on an iterator with no next element will throw a `NoSuchElementException`

(6)

(c) Explain the problem that *fail-fast* iterators are designed to address. Why do we not need to consider a fail-fast approach for the `BTree` iterator?

(4)

5.

(a) Describe the expected case and the worst case look-up times for a dictionary implemented as a binary search tree, and give and example of each.

(4)

(b) Select one type of self-balancing search tree, and carefully describe the mechanism that allows the tree to remain balanced. Explain how this impacts the complexity of the *insert* and *remove* methods

(8)

(c) What is a *perfect hash function*? Explain why it is almost always impossible to have a perfect hash function.

(4)

(d) Suppose that we are using linear hashing, and start with an empty table with 2 buckets ($M = 2$), $split = 0$ and a load factor of 0.9. Explain the steps we go through when the following hashes are added (in order):

$$5, 7, 12, 11, 9$$

(4)

End of paper.

Sample Solutions

1.

1. (a)
   (b)
   (c)
   Generics allow a class to be specified with respect to a type variable, eg Map¡K,V¿ Dynamic Programming solves problems by dividing the into smaller sub-problems, recording the results and reusing them later. eg Floyd Warshall Expected Case Complexity measures the complexity of an algorithm with respect to the average, given a probability distribution over the input space, eg Quick-Sort is expected to be n lg n given that all permutations of an array are equally likely.
   (6)
2. (3)
   Amortized analysis allows expensive operation to be amortized against the cheaper operations that must accompany them, where worst case performance just looks at the worst case for any operator. eg in a multipop stack, mpop is amortized cont ant, but worst case linear.
3. (3)
   exists c, N st f(n)¡c. g(n) for all n¿N, but forall c, N, exists n¿N where g(n)¿c. f(n). eg $f(n) = n^2$, $g(n) = n^3$
4. (4)
5. (4)

2.

1.
2.
3.
4.
5.
6.
7. (15)
8. (5)

3.

1. (4)
2. (5)

3. (5)
4.
5.
6.
7.
8. (6)

4.

1.
```
ArrayDeque<Node<E>> q;

public MyIterator(){q = new ArrayDeque<Node<E>>();q.offer(root);}

public boolean hasNext(){return !q.isEmpty();}

public E next(){
  if(isEmpty()) throw new NoSuchElementException();
  Node<E> tmp = q.poll();
  if(tmp.left!=null) q.offer(tmp.left);
  if(tmp.right!=null) q.offer(tmp.right);
  return tmp.item;
}
```
(10)

2.
```
BTree b1 = new BTree();
BTree b2 = new BTree("A",b1,b1);
BTree b3 = new BTree("B",b2,b2);
BTree b4 = new BTree("C",b3,b3);
Iterator i = b4.iterator();
while(i.hasNext())
System.out.println(i.next());
try{i.next();}
catch(Exception e){System.out.println("Success!");}
```
(6)

3. Fail-fast means we require that if the data structure changes during iteration, we disable the iterator.
   It is not required because our tree is immutable.
(4)

5.

1. (4)
2. (8)

3. (4)

4. (4)

*— —5— (0,1) *— —5,7 — - split —*—5,7— — (0,1,2) —12*—5,7— — — - split —12—5—*—7— (0,1,2,3) - split =M, M = 2*M, split = 0 *—12—5— —7— *—12—5— —7,11—-split —*—5— —7,11—12— (0,1,2,3,4) —*—5,9— —7,11—12— - split — —9*— —7,11—12—5— (0,1,2,3,4,5)