Office Use Only

Monash University

Semester One Examination Period
2007

Faculty of Information Technology

EXAM CODES: **FIT 2004**

TITLE OF PAPER: **Data Structures & Algorithms – Final Exam**

EXAM DURATION: 3 hours writing time

READING TIME: 10 minutes

THIS PAPER IS FOR STUDENTS STUDYING AT:(tick where applicable)

☐ Berwick	☐ Clayton	☐ Malaysia	☐ Off Campus Learning	☐ Open Learning
☐ Caulfield	☐ Gippsland	☐ Peninsula	☐ Enhancement Studies	☐ Sth Africa
☐ Pharmacy	☐ Other (specify)			

During an exam, you must not have in your possession, a book, notes, paper, calculator, pencil case, mobile phone or other material/item which has not been authorised for the exam or specifically permitted as noted below. Any material or item on your desk, chair or person will be deemed to be in your possession. You are reminded that possession of unauthorised materials in an exam is a discipline offence under Monash Statute 4.1.

AUTHORISED MATERIALS

CALCULATORS **NO**

OPEN BOOK **NO**

SPECIFICALLY PERMITTED ITEMS **NO**
if yes, items permitted are:

Candidates must complete this section if required to write answers within this paper

STUDENT ID __ __ __ __ __ __ __ __ DESK NUMBER __ __ __ __

(Intentionally left blank)

Additional Instructions

1. All answers must be given in the exam booklet. If possible write your answer in the space directly below the question. You may use extra pages to develop your answers, but these pages will not be marked.
2. You must tick the boxes for all questions that you have attempted in the table below on this page.
3. Unless stated otherwise, you only need to give procedural pseudo-code for algorithms. Concrete program code is only required where this is explicitly stated.
4. Where the questions ask for pseudocode, you are allowed to use concrete program code if you wish, but this is more difficult and not recommended.
5. Where program code is requested, the questions ask for Java code. You are allowed to use 'C' or SML code instead.
6. 'C' and SML programmers please note that where the questions ask for *methods*, you have to substitute *functions*, where they ask for *classes* you have to substitute *structures* or *datatypes*, respectively.

Good Luck!

I declare that I have attempted the questions marked in the table below:

Question	attempted	Marks (office use only)
1		
2		
3		
4		
5		
6		

(Extra space for working)

Question 1 [Short Answer] [10 × 2 mark = 20 marks]

For each of the following question give a concise answer (in two sentences or less). Where required draw a diagram to explain your answer.

1. What is the advantage of **Mergesort** over Quicksort when considering runtime complexity?

2. Explain what is wrong in the following statement: "It is not possible to write a **sorting algorithm** that has a better runtime bound than $O(n \cdot log\ n)$ steps for sorting n elements".

3. Draw a valid **heap** containing the key elements $\{7, 12, 1, 3, 22, 5, 11\}$. Draw your heap as a tree, not as an array.

4. What is the worst case run-time of searching for an element in a **Splay tree** with n nodes?

5. Name at least one advantage of **Splay trees** over **AVL trees**.

6. After which operations does a **Splay tree** perform a re-balancing?

7. Explain what an **amortized runtime analysis** is.

8. Which graph implementation would you use to implement the **transitive closure** algorithm if runtime is the most important consideration and the graph is not sparse?

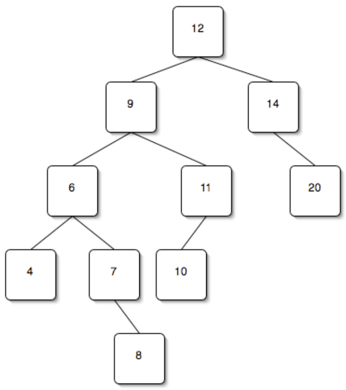
9. Name one well-known standard algorithm for each of the following **algorithm design paradigms**: (a) Greedy, (b) Divide&Conquer, (c) Dynamic Programming. Identify which of your examples use which paradigm.

10. Consider the Quicksort algorithm and its **runtime complexity**. Assume that you had a "magic" implementation of the *partition* function that partitions a list of length n in constant time $O(1)$. What would the average case runtime of quicksort be using this implementation of *partition*?

Question 2 [Balanced Trees] [22 marks]

All of these questions concern different forms of self-balancing trees. Refer to the "Additional Instructions" on page 3 for notes on coding requirements.

1. Perform a standard *AVL re-balancing* for the unbalanced **AVL tree** given below and draw the re-balanced tree next to it. **[3 marks]**



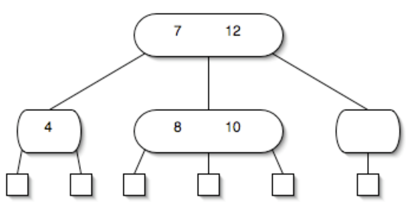
2. Assuming the following Java class skeleton for a **Splay tree**, give *pseudocode* for the `insert` method. You can assume the method `splayAtX` as given. It takes a `SplayTree` node x as its parameter and splays x by performing a rotation for x. Note that `splayAtX` automatically chooses the correct rotation to execute (zig, zig-zig or zig-zag as required). **[5 marks]**

```
public class SplayTree extends BinaryTree {
   public int key;
   public SplayTree left, right;
   ...

   public void splayAtX(SplayTree x) {
      ...  // assume this method as given
   }

   public void insert(SplayTree T, int x)
     // you need to define this.
     // the method inserts a new node with key-value x into T
   }
}
```

3. Which additional information would you have to store in the attributes of the class **SplayTree** to be able to implement the method `public void splayAtX(SplayTree x)`? Give correct Java declarations for the new attributes, briefly explain why you need these and how they would be used in `splayAtX`. **[3 marks]**

4. Consider the method signature of `splayAtX` and your answer to Part 3. Is this an efficient way to implement **splaying**? Explain your answer. **[2 marks]**

5. Perform a standard *transfer* for the underflow in the **(2,4)-tree** node given below and draw the re-balanced tree next to it. **[2 marks]**



6. Which of the following operations in a (2,4)-tree can propagate through the tree: *split, fusion, transfer*? **[2 mark]**

7. Give a Java class named `Tree24` for a **(2,4)-tree** that stores integer keys. You only need to declare the attributes, methods are not required for this question. [**3 marks**]

8. The following is a possible signature for the insert method for a **(2,4)-tree**:

```
public Tree24 insert(Tree24 t, int key) { ... }
```

   Briefly explain why the return value is required and what would be returned in it. [**2 marks**]

(Extra space for working)

Question 3 [Graphs and Graph Algorithms] [26 marks]

1. Draw a diagram for the **adjacency list** representation of the graph given in Figure 1. [**3 marks**]

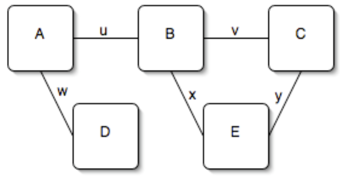


Figure 1: Example for Adjacency List Representation

2. Perform a **Depth First Search** for the graph given in Figure 2 starting from node A. Whenever you have a choice between two nodes, choose the top-most one first. Number the nodes in the diagram in the order in which they are visited. [**3 marks**]

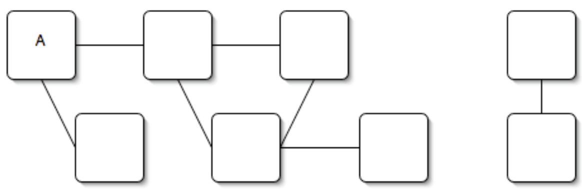


Figure 2: Example for DFS and BFS

3. Perform a **Breadth First Search** on the graph given in Figure 2 starting from node A. Clearly mark the levels of the breadth first search in the diagram by drawing a line around each set of nodes that belong to the same level. Number the levels in the order in which they are visited. [**3 marks**]

4. Consider the graph in Figure 3. Does **Dijkstra**'s shortest path algorithm work for this graph? [**3 marks**]

   - If your answer is "No" briefly explain what the problem is. Mark all nodes in the diagram with the distance labels at the point where the problem is encountered if the algorithm starts at node A.
   - If your answer is "Yes", execute the algorithm for this graph starting at node A and mark all nodes in the diagram with the distance labels that they have at the time when the algorithm takes node B from the priority queue (i.e. "adds it to the cloud").

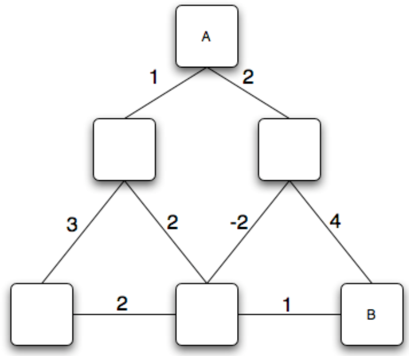


Figure 3: Example for Dijkstra's Algorithm

5. The **Bellman-Ford** dynamic programming algorithm for shortest paths does not work if there is a negative cycle in the graph. Briefly explain what the problem with a negative cycle is.
[2 marks]

6. A simple extension of the **Bellman-Ford** algorithm allows us to test for negative cycles. Given the final distances $d(v)$ that Bellman-Ford has computed for all vertices, we only need to check whether there is some edge e from a vertex u to a vertex v that is inconsistent with the distances $d(u)$ and $d(v)$. Write down pseudocode for the Bellman-Ford algorithm with this extension.
[5 marks]

7. Consider the following **application problem**: The university has an automatic unit enrolment system. Every time a student wants to enrol in a unit, the system checks whether the student has passed all prerequisite units. The university now wants to extend this system to answer questions of the type "which units do I still need to complete before I can enrol in FIT2004". For some student the answer may, for example, be that she still has to complete "FIT1008", but that she also must still complete "FIT 1002" because this is a pre-requisite for "FIT1008". To answer such queries for all units as quickly as possible, it is important that this information is pre-computed as far as possible.

   The prerequisite structure is stored in the form of a directed graph G where the vertices are unit codes and an edge from u to v means that u is a prerequisite for v. This graph is stored in an *adjacency matrix* $A[i, j]$.

   Your task is to write an algorithm that pre-computes and stores as much of this information as possible to support such queries.

   (a) Name the standard graph algorithm that you would use to do this. **[2 marks]**

   (b) Give pseudocode to pre-compute the information. **[5 marks]**

(Extra space for working)

Question 4 [Dynamic Programming] [12 marks]

Two pool players, A and B, compete by playing a series of games. It has been decided that the player who first wins n games is declared as the winner. We have an interest in calculating the chances of the players to win the competition based on how many games each of them has won so far. Let us say, this is because we can bet on A and B after every game in the series.

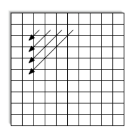
We want to compute $p(i,j)$ which we define as the probability of A to win the competition if A still needs to win i games and B still needs to win j games (or equivalently, A has won $(n-i)$ games so far and B has won $(n-j)$ games so far). To simplify, we assume that A and B are equally good players so that their chances to win any single game are equal. We can define $p(i,j)$ by a simple recursive equation:

$$p(i,j) = \begin{cases} 0 & \text{if} \quad j = 0 \wedge i > 0 \\ 1 & \text{if} \quad i = 0 \wedge j > 0 \\ \frac{p(i-1,j)+p(i,j-1)}{2} & \text{if} \quad \mathrm{i} > 0 \wedge \mathrm{j} > 0 \end{cases}$$

We can understand this as follows: In the first two cases either A or B has clearly won already. In the third case a still needs to win i games and has a 50% chance of winning the next game. In this case, the chances of A to win the competition are given by the average of A's chances to win the competition for the case where A wins the next game and the case where A loses the next game.

- Explain briefly why a direct recursive implementation of the function definition is not an efficient way to calculate $p(i,j)$. **[2 mark]**

- Define a *Dynamic Programming* algorithm to compute $p(i,j)$. **[8 mark]**
  Hint: Your computation needs to progress along diagonals of the matrix as illustrated below.



- What is the runtime complexity of your dynamic programming algorithm? **[2 mark]**

(Extra space for working)

Question 5 [Runtime Complexity] [10 marks]

For each of the following methods, determine their runtime.

1. The *merge* operation used as part of mergesort takes two sorted lists as arguments and returns a list that contains all elements of both lists in sorted order. For this question we will use ascending order. *merge* can be expressed in procedural pseudocode as follows, where the operations *length*, *first*, *rest*, and *insert* correspond to the ones given in the algebra in the appendix.

```
merge(l1, l2)
    begin
       if (length(l1)=0) return l2;
       else if (length(l2)=0) return l1;
       else
          if (first(l1) <= first(l2))
                return insert(first(l1), merge(rest(l1), l2));
          else  return insert(first(l2), merge(l1, rest(l2)));
    end.
```

   State the runtime of *merge* formally by giving a recurrence equation for the worst-case runtime and its solution. **Hint:** It is easier to solve this by looking at the combined length of both lists: $n = length(l1) + length(l2)$. **[5 marks]**

2. The *reverse* operation takes a single list as its argument and returns a list with the same items in reversed order. The list algebra in the appendix contains a (somewhat unusual) definition of *reverse*. It uses two other operations: *first_half(l)*, which returns a list with the first $\lfloor n/2 \rfloor$ of l in the same order as in l and *second_half(l)* which returns a list with the last $\lceil n/2 \rceil$ elements of l in the same order as in l.

   In procedural pseudocode, this *reverse* operation can be specified as follows:

```
reverse(l)
    begin
       if (length(l)<2) return l
       else begin
          l1 := reverse(first_half(l));
          l2 := reverse(second_half(l));
          return append( l2, l1 );
       end
    end.
```

   Assume that the operations *first_half*, and *second_half* as well as *append* run in linear time, ie. they need $O(n)$ steps for argument lists with n elements.

   State the runtime of *reverse* formally by giving a recurrence equation for the worst-case runtime and its solution. **[5 marks]**

(Extra space for working)

Question 6 [Algebras and Datatypes] [10 marks]

Appendix A gives an algebra for integer lists very similar to the one discussed in the lectures. Your task is to extend this Algebra with a *merge* operation similar to the one used in mergesort. This *merge* was outlined in Question 5.

1. Extend the algebra by specifying the signature for *merge*. **[2 mark]**

2. State one reasonable axiom regarding the length of the result list of *merge* in relation to its argument lists. You do *not* need to give a proof for this axiom! **[3 mark]**

3. Extend the algebra by specifying the function *merge* in the `functions` part. You can use the operations already defined in the algebra. **[5 marks]**

(Extra space for working)

Appendix: Algebra *integer list*

```
ALGEBRA integer list

sorts intlist, int, bool;

ops
        empty:                                -> intlist;
           (* returns an empty new list *)
        length:      intlist                  -> int;
           (* returns the number of elements in the list *)
        insert:      int x intlist            -> intlist;
           (* inserts an element at the front *)
        delete:      int x intlist            -> intlist;
           (* deletes an element *)
        first:       intlist                  -> int;
           (* returns the first element in the list *)
        rest:        intlist                  -> intlist;
           (* returns the list with the first element removed *)
        contains:    int x intlist            -> bool;
           (* tests whether an element is contained in the list *)
        isempty:     intlist                  -> bool;
           (* tests whether the list is empty *)
        append:      intlist x intlist        -> intlist;
           (* appends the second list to the first *)
        first_half:  intlist                  -> intlist;
           (* returns the first n/2 elements of the list *)
        second_half: intlist                  -> intlist;
           (* returns the second n/2 elements of the list *)
sets
        bool    = {true, false};
        int     = Z;
        intlist = nil | cons(e:int, s:intlist);
functions
        append(nil, l)              = l
        append(cons(e,l1), l2)      = cons(e, append(l1, l2))

        isempty(nil)                = true
        isempty(cons(e,s))          = false

        insert(e, s)                = cons(e,s)

        delete(e, cons(e,s))        = s
        delete(e, cons(f, s))       = cons(f, delete(e, s)) if not (e=f)
        delete(e, nil)              = nil

        contains(e, nil)            = false
        contains(e, cons(e,s))      = true
        contains(e, cons(f,s))      = contains(e,s) if not (e=f)

        reverse(l)                  = l  if  length(l)<2
        reverse(l)                  = append(
                                             reverse(second_half(l)),
                                             reverse(first_half(l))
                                           )  if   length(l)>=2
...
```