Office Use Only		

Monash University

Semester One Examination Period
2008

Faculty Of Information Technology

EXAM CODES:	**FIT 2004**
TITLE OF PAPER:	**Data Structures and Algorithms – Final Exam**
EXAM DURATION:	3 hours writing time
READING TIME:	10 minutes

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

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

AUTHORISED MATERIALS

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

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		
7		

(Extra space for working)

Question 1 [Short Answer] [10 × 2 marks = 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. Explain the meaning of the Ω **notation** in the context of **runtime complexity**.

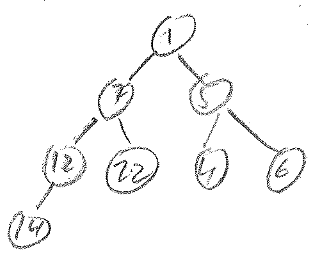
[1.5] Ω provides a lower bound for the runtime.
[0.5] Formally, $\exists c \exists n_0 \forall n > n_0 . f(x) \geq c \cdot g(x) \Leftrightarrow f(x) \in \Omega(g(x))$

2. Outline one condition under which is is possible to give **sorting algorithms** that run in worst case *linear time*.

[2] { If there is only a finite set of values for the individuals (as in bucket sort and radix sort)

3. Is the following a valid embedding of a **min-heap** in an array? If your answer is yes, draw it as a tree; if your answer is no, explain why it is not valid.

1	7	5	12	22	4	6	14



[1] no, [1] it isn't ④ can't be a successor of ⑤ in a min-heap

4. What is the worst case run-time of inserting an element in a **Red-Black tree** with n nodes?

$\mathcal{O}(\log n)$ [2]

5. Assume you are writing a program for a real-time control system that maintains a task list (with unique task numbers as keys). You are required to guarantee that each task lookup can be completed in $c \cdot \log n$ time for some constant c. Your choice for the basic data structure is between a **Splay tree** and a **2-4 tree**. Which one would you choose? Why?

Choose 2-4 tree. [1]
For the splay tree you can only guarantee $\mathcal{O}(n)$ in the worst case [1]

6. Which part of the **heap sort** algorithm dominates its runtime complexity? Explain your answer.

Bottom-up heap building is linear, [1]
so the runtime is dominated by the n deletemin operations [1]

7. Explain what the term "full history" refers to in the context of a **full history recurrence relation**. It refers to the fact that at each step of the recurrence not only one other recursive instance needs to be taken into account, but all other (previous) values.

[2] Eg $T(n) = \frac{1}{N} \sum_{i=0}^{n-1} T(i)$. Here we sum over all "previous" values.

8. What is the complexity of **depth first search** if implemented with an edge list structure?

$\mathcal{O}(n + nm) = \underbrace{\mathcal{O}(nm)}_{[2]}$

9. Name the **algorithm design paradigms** behind the following algorithms: (a) Dijkstra's shortest path algorithm, (b) Bellman-Ford's shortest path algorithm, (c) Floyd-Warshall's shortest path algorithm, (d) Kruskal's minimum spanning tree algorithm.

[0.5] each

a) Greedy
b) Dynamic Programming
c) Dynamic Programming
d) Greedy

10. Consider the **Quicksort** algorithm and its **runtime complexity**. Assume that you had extended the partition function so that it checks whether any of the sublists arising from the partition are already sorted and that your quicksort implementation skips the sorting of such sublists. How would this influence the worst-case and best-case runtime of Quicksort?

We would still have an $\mathcal{O}(n)$ partition operation on each level. It could not run faster.

[1] In the worst case we would still have $\mathcal{O}(n)$ levels, so that the runtime is $\mathcal{O}(n^2)$

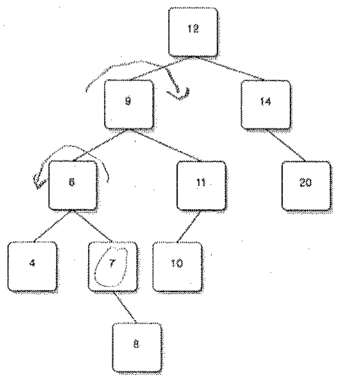
[1] In the best case (we start with a sorted list) there would be no recursive call, so we'd have $\mathcal{O}(n)$.

Obviously, any improvement based on "being lucky" can never change the worst case.

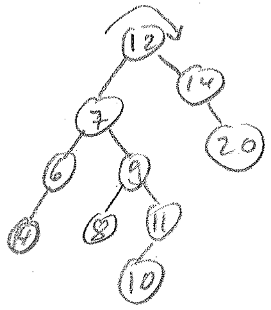
Question 2 [Search Tree Structures] [20 marks]

All of these questions concern different forms of search tree structures. Refer to the "Additional Instructions" on page 3 for notes on coding requirements.

1. Below you see an instance of a **Splay tree**. Assume you execute a search for the node with key 7 in this tree. Illustrate how the splaying takes place (draw the tree after each zig, zig-zig or zig-zag rotation that you perform). **[4 marks]**



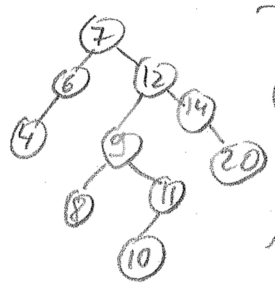
① rl = zig-zag



[2]

A least 1 per tree for identifying correct operation. -0.5 per tree for minor mistakes.

② l = zig



[2]

2. Complete the Java class skeleton given below for a **2-4 tree** storing integer keys. You must specify all attributes, and the signatures for the *find*, *insert*, and *delete* methods (but not their method bodies). **[3 marks]**

```
public class Tree24 extends BinaryTree {
   public ...            // give all attributes as required

   ...                   // specify method signatures for
                         // find, insert, and delete
                         // but omit the method bodies for these

   public int used;       // number of keys stored
   public int[] keys;     // the keys themselves          } [0.5] each
   public Tree24[] children; // the successors

   public boolean find(int x, Tree 24 T)
   public Tree24 insert(int x, Tree 24 T)    } [0.5] each
   public Tree 24 delete(int x, Tree 24 T)
   }

}
```

3. For the class Tree24 that you defined above, complete the skeleton for the *fusion* method given below, including the full method body code. Your method is not required to check whether a fusion is applicable, you can assume this will be checked before the method will be called. **[5 marks]**

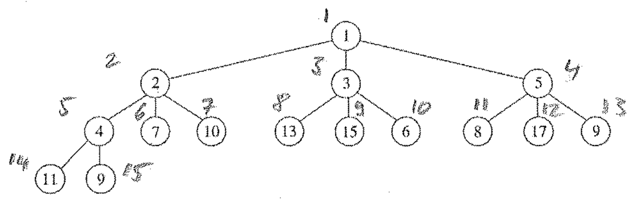
```
public .... fusion(Tree24 P, Tree24 E) {     // fill in the return type

...                         // fully define the method body
                            // the method executes a fusion
                            // for the empty tree node E with given Parent node P

}
```

irrelevant
Note that the type of
relatively complex coding in the
question is not asked in the
exam any more. This is now
answered in the pracs.

4. For the class Tree24 that you defined above, complete the skeleton for the *find* method below, including the full method body code. **[4 marks]**

```
       boolean       int x, Tree 24 T
public .... find(...) {  // complete method signature and method body

...
  { if (T.used==0) return false;
    else { for (int i=0; i<T.used &&
                    T.keys[i]<x; i++);

           if (T.keys[i]==x) return true;
           find (x, T.children[i]);

         }

}
  // This assumes empty leaf nodes
```

5. There are many different variations of **Heap** data structures. One such variation is the *d-heap* which stores d children per node instead of just two. Below is an example of a d-heap. Draw an array embedding for this d-heap and give a general formula to calculate the position of the first child of any node that is stored at array index i for your embedding. **[4 marks]**



1	2	3	5	4	7	10	13	15	6	8	17	9	11	9

[2]

child $(i) = d \cdot i - 1$

[2]

only [1] for "$d \cdot i$"

(Extra space for working)

Question 3 [Graphs and Graph Algorithms] [24 marks]

1. Write down the **adjacency matrix** representation of the graph given in Figure 1. **[2 marks]**

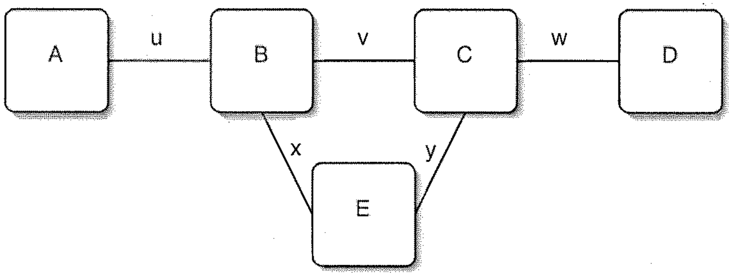


Figure 1: Example for Adjacency List Representation

	A	B	C	D	E
A	0	1	0	0	0
B		0	1	0	1
C			0	1	1
D				0	0
E					0

[0.5] for correct matrix structure

[1.5] for correct values

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]**

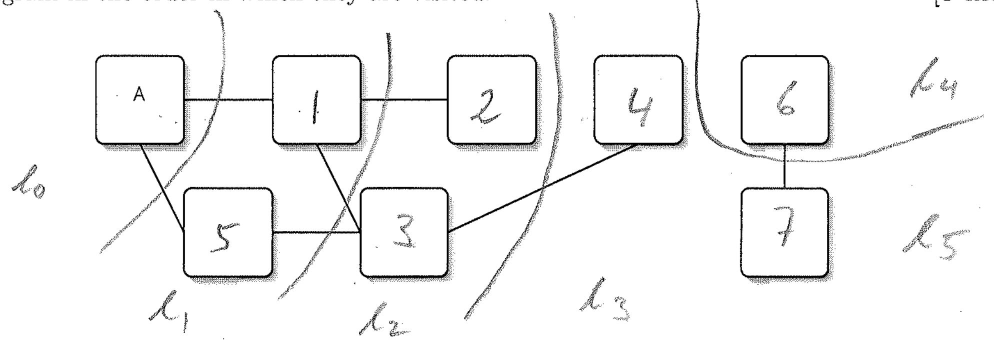


Figure 2: Example for DFS and BFS

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 lines that separate the different levels. Number the levels in the order in which they are visited.

[3 marks]

4. You know (at least) three different shortest path algorithms: **Dijkstra**, **Bellman-Ford**, and **Floyd**. Briefly discuss the criteria on which you would base a decision which one to use for a given application. **[3 marks]**

Floyd is an all-pair shortest path Algorithm [1]
if shortest paths are required only from a given start point
Dijkstra or Bellman-Ford should be used [1]
in that case Dijkstra is preferable if there are no negative
edges, otherwise BF must be used [1].

5. Explain the meaning of **Path compression** for a union-find structure. What is the amortized runtime complexity of a find using a path-compressing tree structure? **[2 marks]**

At each find operation a shortcut from all nodes along the
path to the root is added.
The asymptotic runtime is $\log^*(n)$

6. Consider the following **application problem**: We want to write a program that allows students to determine in which order they can take their chosen units such that all unit prerequisites are taken into account. For example, a student may want to take FIT1002, FIT2004, FIT1008, FIT2999. The unit FIT1002 has no prerequisites, FIT1008 has FIT1002 as prerequisite, FIT2004 has FIT1008 as prerequisite, and FIT2999 also has FIT1008 as a prerequisite. A possible sequence then is FIT1002→FIT1008→FIT2004→FIT2999. The other possible sequence is FIT1002→FIT1008→FIT2999→FIT2004. Each unit should be taken as early as possible, but if there is no direct or indirect prerequisite relation between two units, the student is free to take them in any order.

Write a program to find an order for a given set of units that obeys the prerequisite structure. The input to your program is a directed graph that represents the chosen units and their prerequisite structure. Each vertex in this graph stands for one chosen unit and each directed edge (u, v) models a prerequisite relation, i.e. unit u is a prerequisite for unit v. The task of your program is to assign labels to the vertices such that each vertex is labelled with its position in an admissible sequence.

(a) Name the standard graph algorithm that you would use to compute the year levels. **[2 marks]**

BFS

(b) Give pseudocode to compute this information. **[5 marks]**

```
Topsort(G)
  n = number of vertices
  while (G is not empty)
  { let v be a vertex in ~~G~~ with no
        outgoing edges;
    label v with n;
    delete v from G;
     n = n - 1;
  }
```

(c) Give the runtime of your algorithm. **[2 marks]**

$O(n+m)$ with adjacency list

(d) Give the fastest possible runtime for this problem. You are allowed to change the underlying data structure to achieve a faster runtime. You do not need to give pseudo-code for the new faster implementation if it is different, but you must state which data structure you would use. **[2 marks]**

- can't be done faster
- requires adjacency list

(Extra space for working)

Question 4 [Dynamic Programming] [12 marks]

You have to design an algorithm to help optimize the allocation of workers to the subtasks of a time-critical project. The goal is to maximize the chances of meeting the deadline for the project.

The project consist of N independent subtasks and it only counts as successfully completed if all its subtasks are completed by the deadline. w workers can be assigned to the subtasks. Each worker is qualified to work on any of the subtasks, and each worker can only be assigned to a single subtask. For each subtask $k = 1, \ldots, N$ you know the probability $p_k(j)$ that it will be completed by the deadline if j workers are assigned to it. Assume $p_k(j)$ is given as an array of fixed values.

You have to write a program to compute the highest possible probability to complete all N subtasks with w workers by the deadline. Formally, this can be specified as

$$\begin{array}{ll} \text{maximize} & p_1(x_1) \times p_2(x_2) \times \ldots \times p_N(x_N) \\ \text{subject to} & w \geq \sum_{k=1}^{N} x_k \end{array}$$

where x_k is the number of workers assigned to subtask k.

Let $V_k(i)$ stand for the highest possible probability of completing subtasks $k, ..., N$ with i workers. Then the answer to our problem is given by $V_1(w)$ and we can defined a recursive equation for $V_k(i)$ as follows:

$$V_k(i) = \underset{1 \leq x_k \leq i-N+k}{maximum} \; (p_k(x_k) \times V_{k+1}(i - x_k))$$

This can be understood as follows: You first make a decision how many workers x_k to assign to the subtask k under consideration. Each task must be assigned at least 1 worker and you can assign at most $i - N + k$ workers to task k, since you only have i workers available and you will need at least $(N - k)$ workers for the remaining subtasks. The best probability of success under this choice of x_k then is $p_k(X_k)$ multiplied by the highest possible probability to complete the remaining subtasks. This computation can be performed by using dynamic programming.

1. Fill in the algorithm skeleton below to give a *Dynamic Programming* algorithm that computes $V_1(w)$. **[8 marks]**

```
Algorithm V

    for all i=1...~~N~~ i+(k-N)
        v[N,i] = p[N,i]

    ...

    for k=N-1 to 1 do
        for i = 1 to i+(k-N) do
            let max=0
            for xk = 1 to i+(k-N) do
                if $p_k(x_k)$ * V[k+1, i-$x_k$] > max then
                    max = $p_k(x_k)$ * V[k+1, i-$x_k$];
    return v[1,w]   V[k,i] = max;
```

...continued overleaf

2. Explain briefly why a direct recursive implementation of the function definition is not an efficient way to calculate $V_1(w)$. **[2 marks]**

in a direct recursive computation every V[i] would be computed several times.
("overlapping subproblems")

3. What is the runtime complexity of your dynamic programming algorithm? **[2 marks]**

$\mathcal{O}(Nw^2)$

Question 5 [Runtime Recurrence Relations] [5 marks]

The *undup* operation given below takes two lists without duplicates as arguments and returns a single list that contains all elements of both lists but no duplicate elements. *undup* 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.

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

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

$T(0) = 1$

$T(n) = n + T(n-1)$

$$\begin{aligned} T(n) &= n + T(n-1) \\ &= n + (n-1) + T(n-2) \\ &= n + (n-1) + (n-2) + T(n-3) \\ &= \underbrace{n + (n-1) + \dots +}_{(n-1) \text{ times}} T(0) \\ &= \sum_{i=1}^{n} i \\ &= \Theta(n^2) \end{aligned}$$

(Extra space for working)

Question 6 [Algebras and Datatypes] [9 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 *zip* operation for two lists. *zip* takes two lists and and joins them into a single list by taking the first element from the first list, then the first element from the second list, then the second element from the first list, then the second element from the second list, and so on until one of the lists is exhausted. If there are any elements left in the other list they are appended to the result. For example,

$$zip([1,2,3,4,5],[9,8,7]) = [1,9,2,8,3,7,4,5]$$

1. Extend the algebra by specifying the signature for *zip*. **[2 marks]**

   zip: intlist × intlist → intlist

2. State one reasonable axiom for *zip*. You do *not* need to give a proof for this axiom! **[2 marks]**

   any number of axioms is possible, but zip(nil, L) = zip(L, nil) = L is good enough. Give [1] if axiom structure correct even if axiom has a minor mistake

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

[2] for correct use of cons argument matching
[1] for base cases
[2] for correctly alternating

zip (L1, L2) = zip1(L1, L2)

zip1(cons(a, L1), L2) = cons(a, zip2(L1, L2))
zip1(nil, L) = L

zip2(L1, cons(a, L2)) = cons(a, zip1(L1, L2))
zip2(L, nil) = L

(Extra space for working)

Question 7 [Divide and Conquer] [10 marks]

We want to compute the median of a list L of n integer numbers. For simplicity assume that the list length n is odd, so the median is defined to be the middle element ($n/2$-th element) in the sorted version of L. The list is allowed to contain duplicates. For example, the median of the list $[7, 3, 2, 4, 1, 5, 6]$ is 4. The naive approach would be to sort the list and pick the $n/2$-th element. The question is whether we can do this faster.

A possible Divide & Conquer approach to computing the median works as follows: We pick some element v in the list L and partition L into three parts:

$$\left.\begin{array}{lcll} L_{<v} & = & \{x \in L \mid x < v\} & \text{is the list of all elements in L smaller than v} \\ L_{=v} & = & \{x \in L \mid x = v\} & \text{is the list of all elements in L equal to v} \\ L_{>v} & = & \{x \in L \mid x > v\} & \text{is the list of all elements in L greater than v} \end{array}\right\} \mathcal{O}(n)$$

Obviously it is easy to find out in which of the lists the median is. We continue recursively to search in the corresponding sub-list. Thus we find the median of the list L by computing the value of *rank*$(L, n/2)$, where *rank* is defined as:

$$rank(L,k) = \begin{cases} rank(L_{<v}, k) & \text{if} \quad k \leq \mid L_{<v} \mid \\ v & \text{if} \quad \mid L_{<v} \mid < k \leq \mid L_{<v} \mid + \mid L_{=v} \mid \\ rank(L_{>v}, k - \mid L_{<v} \mid - \mid L_{=v} \mid) & \text{if} \quad \mid L_{<v} \mid + \mid L_{=v} \mid < k \end{cases}$$

v should be chosen such that the list size shrinks in general as fast as possible, so ideally each recursive call of *rank* on average reduces the list to half its size (assuming there are few duplicates). In the following you may assume that *rank* always manages to find such an "ideal" v, and that it does so in $O(1)$ time.

1. State the runtime of *rank* formally by giving a recurrence equation for the worst-case runtime under the above assumptions for selecting v. Don't forget to take the partitioning of the list into account!

[4 marks]

$T(1) = 1$ [1]

$T(n) = \underset{\underset{[1]}{\uparrow}}{n} + \underset{\underset{[2]}{\uparrow}}{T\left(\frac{n}{2}\right)}$ [3]

2. Solve the recurrence relation you have given as the answer to the previous question.

[3 marks]

$T(n) = \mathcal{O}(n)$

3. In reality it is not possible to pick a v in constant time that approximately halves the list length. However, on average it is possible to pick a v in *linear* time that shrinks the list to 3/4 of its original size. Explain how this change influences the runtime of your median algorithm.

[3 marks]

$T(1) = 1$

[1] $T(n) = n + n + T\left(\frac{3}{4}n\right) = 2n + \frac{3}{4}n$

use Master Theorem. $T(n) = a\,T\left(\frac{n}{b}\right) + cn^k$

$c = 2,\ K = 1,\ a = 1,\ b = 4/3 \Rightarrow a < b^k \Rightarrow T(n) = \mathcal{O}(n^k) = \mathcal{O}(n)$

[2] runtime becomes linear

(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:      intlist                    -> int;
           (* returns the first element of the list *)
        rest:       intlist                    -> intlist;
           (* returns the list excluding the first element *)
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

        length(nil)              = 0
        length(cons(e,s))        = 1+length(s)

        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)

        first(cons(e,s))         = e

        rest(cons(e,s))          = s
```