Name:	
Student Number:	
Signature:	

UNIVERSITY OF NEW SOUTH WALES

FINAL EXAMINATION

Session 2, 2008

COMP1921

DATA STRUCTURES AND ALGORITHMS

- Time allowed: **3 hours**
- Total number of marks: **60**
- The exam consists of **TWO** parts
  - ALL questions of BOTH parts are to be answered
  - PART 1 – Practical – Consists of 3 questions worth 30 marks
    - Questions *are* of equal value
    - Solutions are to be coded, tested and submitted using the workstation
  - PART 2 – Written – Consists of 4 questions worth 30 marks
    - Questions *are not* of equal value
    - Answers are to be handwritten into the Answer Booklet
    - Start your answer to each question on a new page
- This paper and the *back* of the Answer Booklet may be used for rough working
- Rough working *will not* be marked
- Candidates **MAY NOT** retain this paper
- Candidates must bring their student card for identification
- Candidates **MAY NOT** use any aids apart from the provided
  C Reference Card and writing materials
- Candidates *may not* communicate with any person besides the supervisor
- Mobile phones must be switched off

PART 1 — Practical Part (30 marks)

There are **THREE** questions in this Part. In each case, you are required to modify a given program by implementing a function to achieve a specific outcome, or to transform some data in a specific way. Each question is worth 10 marks.

You will be provided with an executable file implementing the solution to each question. You will need to provide your own test data to ensure your solution behaves correctly. You are also provided with a *C Reference Card* to assist you in programming.

Logging on

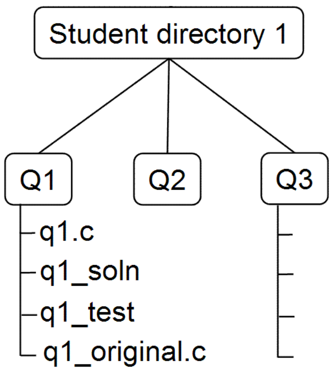
To log on to the examination environment, sit at your allocated workstation and move the mouse. You should see a small login window (different from the usual logon screen). Type your login name and usual password. Leave the window near the top of the screen so the supervisors can check your identity.

The exam environment provides you with a basic window manager that can create xterms and editor windows. Press the *right* mouse button to activate the menu.

The exam environment provides you with access to the usual UNIX commands, but you will not be able to access other accounts or the Internet.

Exam directory structure

The following diagram illustrates the structure of your working area.



When you log on, your HOME directory is set to a workspace with three question subdirectories named `Q1`, `Q2` and `Q3` respectively. Each directory contains four files named according to the following conventions. For example, the `Q1` directory for Question 1 contains the C source you are required to modify `q1.c`, an executable solution `q1_soln`, a sample test file `q1_test`, and in case you need a fresh copy of the C source to work with, a copy of the source `q1_original.c`.

Submission

When you complete each question, enter this command into an xterm:

```
submit questionnum file
```

where *questionnum* is the Question number (1 2 or 3) and *file* is the C source file you wish to have assessed (`q1.c`, `q2.c` or `q3.c`).

You may submit multiple times if you wish; the last submission will be assessed. At the end of the elapsed time you must submit whatever you have completed.

If you have questions

You may put any questions to the supervisors. Raise your hand and wait for a response. You may ask for a clarification of a question if you believe it to be unclear or ambiguous. You may *not* ask for advice on how to solve the task (or if you do the supervisor will ignore the request).

If you need to leave the room temporarily, raise your hand. As you must be accompanied by a supervisor only one person may be absent at a time. Nobody may leave before half an hour has elapsed from the end of reading time and nobody is admitted after this time.

And finally...

Try to relax. This is not meant to be an ordeal, but to reflect the kind of situation where simple but realistic problems can be solved using skills acquired over the last 12 weeks. Just make sure you leave sufficient time to attempt all questions.

Part 1 – Question 1: (10 marks)

Write a function that moves the node containing the largest element of an unordered list to the end of the list.

The C source file `q1.c` contains functions to obtain an unordered list of single ASCII characters from a user. You are required to implement the function `reorder` to move the largest item (greatest ASCII code value) to the tail of the list. You may use any other functions already in the file to implement your function.

The executable file `q1_soln` may be used to check the behaviour of the desired solution. The test file `q1_test` provides a guide to the format of the input. When provided as input to the executable, the format of the output can also be observed. Note that you may need to test other input cases to ensure your solution works as intended.

Enter your solution by modifying the stub for function `reorder` at the end of the given source file.

When you are finished working on the solution, use the command

```
submit 1 q1.c
```

to submit your work for marking. You may *resubmit* your solution after further work on it. We will mark your last submission.

End of Question 1

Part 1 – Question 2: (10 marks)

Write a function that takes as arguments an integer array `A`, an integer `n` that indicates how many elements of `A` may be accessed, and an integer `k`, and returns the value of the `k`-th smallest integer in `A`. That is, the value returned should be the one that would move into `A[k-1]` if array `A` were to be sorted. The array `A` should not be modified, and duplicates should be handled correctly.

Hint: When duplicates are taken into account, the `k`-th smallest element of an array is characterized as having *fewer than* `k` smaller elements and *no more than* `n - k` larger elements.

The file `q2.c` contains a main function that will allow the user to enter and store in array `A` up to 100 integers in random order. This function calls the function `kthsmallest` which you are required to modify.

Reminder: Use the command

```
submit 2 q2.c
```

to submit your work for marking.

End of Question 2

Part 1 – Question 3: (10 marks)

Suppose nodes in a graph represent people and there is an edge from p to q if p knows q. A *celebrity* is a person c such that everyone knows c but c knows nobody else. Note that there can be at most one celebrity.

Here is an efficient way to determine a celebrity. The algorithm has an elimination phase and a testing phase. In the elimination phase, all persons except one are eliminated as being non-celebrities. The testing phase checks whether the remaining person is a celebrity. Order the people 0, 1, 2, ... For the elimination phase, examine pairs of people as follows. First look at 0 and 1. If 0 knows 1, then 0 cannot be a celebrity so carry forward 1 to the next step. If 0 does not know 1, then 1 cannot be a celebrity so carry forward 0 to the next step. Whoever remains is compared with 2 in the same way, and so on, until the end of the list is reached. Now in the testing phase, the one remaining person is checked to see whether that person is known by all others and also knows nobody else.

The given C program `q3.c` uses integers for nodes and the adjacency matrix representation for edges (`adjacency[i][j] = 1` if and only if `i` knows `j`). It reads an initial graph from standard input (we can assume the graph is nonempty and everyone knows him/herself).

Implement the above algorithm to determine if the graph contains a celebrity. If there is a celebrity, the function should return the index of the celebrity, otherwise the function should return -1.

Use the command

```
submit 3 q3.c
```

to submit your work for marking.

End of Question 3

PART 2 — Written Part (30 marks)

INSTRUCTIONS: READ CAREFULLY
This Part contains **FOUR questions** *numbered 1 – 4.*
Use a fresh page of the Answer Booklet *to commence your answer to* **each question***.*

Part 2 – Question 1 (10 marks)

The following line of C code was found in a program involving the dynamic creation and manipulation of linked lists:

```
temp = (node *)malloc(sizeof(node));
```

a) What does the function

```
void * malloc(size_t)
```

return when sufficient memory is available? (2 marks)

b) What does it return when sufficient memory is not available? (1 mark)

c) In the first line of code above, what do you expect the type of the variable temp to be? (1 mark)

d) From which region in memory is data dynamically allocated by the malloc function? Illustrate your answer with a diagram. (2 marks)

e) What are the advantages of dynamically allocating memory over static allocation? (2 marks)

f) Describe the concept of a "memory leak" in a program. How are memory leaks avoided? (2 marks)

End of Question 1

Part 2 – Question 2: (7 marks)

a) Give a brief explanation of how the run-time stack supports function invocation. (3 marks)

b) What information is typically stored in a stack frame? (2 marks)

c) What is “stack overflow”? How does it occur? (2 marks)

End of Question 2

Part 2 – Question 3: (4 marks)

Design a suitable data structure for representing and storing the following information contained in the periodic table of the elements.

Each element is described by its name, its chemical symbol (up to 3 characters in length), its atomic number (number of protons in its nucleus), and its atomic weight (a single precision real number). Assume that 118 elements are known.

a) Construct the required definitions and declarations to implement your design. (3 marks)

b) Describe a sensible approach you could use to initialize the data that would be stored in your structure. (1 mark)

End of Question 3

Part 2 – Question 4: (9 marks)

A *connected component* of a graph is a subset of the nodes of the graph such that there is a path between any pair of nodes in the same component, but no path between any two nodes in different components. Intuitively, each component is an "island" in that there is a path beteeen every pair of nodes on the same island but no path between nodes on different islands.

Here is some pseudo-code to find the connected components of a graph: initially every node is in a separate component, then we iterate over the edges in the graph, merging components as we discover they are connected by an edge.

```
//  initialize components: each initial component is a single node
(1) Nodes = the list of all nodes in the graph;
(2) for each node in Nodes
(3)     component[node] = node;
//  now combine components where they are connected by an edge
(4) Edges = the list of all edges in the graph;
(5) while Edges is not empty {
(6)     delete an edge {u,v} from Edges;
(7)     if u and v are currently in different components
(8)          merge components containing u and v
(9)     }
```

Supposing we can use nodes to index an array, a simple representation is to number components 0, 1, 2, ..., and set each array entry to be the number of the component containing the node. For example, if the graph contains cities `sydney`, `melbourne`, `brisbane`, `auckland` and `wellington`, the array `component` could be as follows:

```
component[sydney] = 0
component[melbourne] = 0
component[brisbane] = 1
component[auckland] = 2
component[wellington] = 2
```

Two nodes `u` and `v` are in the same component if and only if the values of `component[u]` and `component[v]` are the same. Thus `sydney` and `melbourne` are in one component, `brisbane` is in another component by itself, and `auckland` and `wellington` are in a third component.

A basic operation is merging two components into one. As an example, suppose we merge components 0 and 1: the new values for `sydney`, `melbourne` and `brisbane` must now be the lower of the two values (so here they are set to 0). Thus `sydney`, `melbourne` and `brisbane` are now in the same component.

```
component[sydney] = 0
component[melbourne] = 0
component[brisbane] = 0
component[auckland] = 2
component[wellington] = 2
```

We now wish to calculate the time complexity of this algorithm as an O-function of n (the number of nodes) and m (the number of edges).

In terms of these parameters:

a) How many times (exact number) is the statement in line 3 executed (initialization of components)? (1 mark)

b) How many times (exact number) is the statement in line 6 executed (deleting an edge from the list of edges)? (1 mark)

c) What is the time complexity (O-function) of testing whether two nodes are in different components? (1 mark)

d) In the worst case, how many times (O-function) is the statement in line 8 executed (merging two components)? (2 marks)

e) What is the worst case time complexity (O-function) of merging two components into a single component? (2 marks)

f) Hence what is an estimate for the overall worst case time complexity (O-function) of this function? Ignore the complexity of lines 1 and 4 that create the list of nodes and edges in the graph. (2 marks)

Explain your answers briefly and state any assumptions you make.

End of Question 4