UNIVERSITY OF LONDON

IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

EXAMINATIONS 1999

BEng Honours Degree in Computing Part I
MEng Honours Degrees in Computing Part I
BSc Honours Degree in Mathematics and Computer Science Part I
MSci Honours Degree in Mathematics and Computer Science Part I
for Internal Students of the Imperial College of Science, Technology and Medicine

This paper is also taken for the relevant examinations for the Associateship of the City and Guilds of London Institute

PAPER 1.7 / MC 1.7

SOFTWARE ENGINEERING – TURING, ABSTRACT DATA TYPES AND OBJECTS

Wednesday, May 5th 1999, 2.00 – 4.00

Answer FOUR questions

For admin. only:
paper contains 6 questions

Section A *(Use a separate answer book for this Section)*

Both questions in Section A are about a game called **Go Fish**. The card game **Go Fish** is played between two players as follows:

start: Each player is dealt a hand of five cards from the deck. The value (A,2,3,..,J,Q,K) is important but not the suit. If a player has any pairs (say two tens) in his hand then he puts them into his pile of pairs. When it is a player's turn he will be called the *current* and his opponent will be called the *opponent*. The first current player is the player who was dealt the first card.

turn: A turn starts with current player asking opponent for a card. The current player must hold a card of the same value. If the opponent has a card of the same value it is given to the current player. The pair of cards (the card from current player's hand and the matching card from the opponent player's hand) is added to current player's pile of pairs. Current player continues asking opponent for cards until the opponent cannot provide a matching card. At that time the opponent says "**Go Fish**". Current player takes a card from the deck and it becomes the opponent's turn.

Any time in current player's turn if he is holding no cards, he takes a card from the deck before asking his opponent for a card.

end: The game is over when there are no more cards in the deck and neither player has any cards. The winner is the player that has collected the most pairs.

For questions one and two you must use the following declarations:

```
type Pack : array 1..13 of int
%Pack is the type of variables that contain the
%number of each card held, so if var p:Pack and
%there are 2 Aces, a 5 and 3 8s in p then
% p = [2,0,0,0,1,0,0,3,0,0,0,0,0]

type Cards : record
               num : int   %no. of cards
               vals : Pack %the cards
             end record

type Player : record
                name : string(10) %of player
                num : int    %no. of cards in hand
                pairs : int  %no. of matched pairs
                vals : Pack  %cards in hand
              end record
```

1 This question is about processing data for the game **Go Fish**. Throughout this question you must use the type declarations for `Pack`, `Cards` and `Player` at the end of the **Go Fish** description.

a A deck of cards and two players could be declared and initialised as follows:

```
var deck : Cards := init
  (52,init(4,4,4,4,4,4,4,4,4,4,4,4,4))

var my, computer : Player := init
  ((" "),0,0,init(0,0,0,0,0,0,0,0,0,0,0,0,0))

put "What is your name?" ..
get my.name
computer.name = "Computer"
```

i) What does this code do? Why does the deck initialisation have thirteen 4s whereas the initialisation for `my` and `computer` has only 0s?

ii) The keyword `init` allows initialisation within a declaration statement. Rewrite the declarations and code above removing all the explicit `init`s and replacing them with code that performs the same functionality.

b Throughout the game there is a need to read from the keyboard a single character that is to represent a card. Within the program the cards are represented by the numbers 1 to 13. From the keyboard, however, the player should type a (or A), 2, 3, 4, 5, 6, 7, 8, 9, t (or T), j(or J), q (or Q), or k(or K).

i) Write a procedure

```
getCard( var c : int)
```

which reads in a card from the keyboard and returns its internal representation in the parameter. You can assume that your input data is a single legal character.

ii) How would you have to alter your code for part b i to deal with both white space and illegal characters.

c Write a procedure `printHand`, which prints out a player's hand. For example `printHand` might produce the following output:

```
Computer's hand contains 5 cards and 6 pairs.
1   3
2   5
1   7
1   T
```

You may find the declaration `Cnames` useful.

```
const Cnames := "A23456789TJQK"
```

The three parts carry, respectively, 40%, 40% and 20% of the marks.

[Turn over

2 This question is about writing code for the game **Go Fish**. Throughout this question you must use the rules and type declarations for `Pack`, `Cards` and `Player` that are in the **Go Fish** description that precedes question 1.

a Turing has a predefined procedure `randint(n,start,finish)`, which produces a new number n each time it is called such that `start` ≤ `n`≤`finish`.

i) Using `randint` or otherwise, write a procedure

```
dealCard(var deck:Cards, var p:Player)
```

that transfers a card from the deck to the player. Why are both parameters `var` parameters?

ii) Using `dealCard` or otherwise, write a procedure `dealHands` which deals two hands of five cards each. Do not use global data; pass all data as parameters.

b i) Write a predicate `cardIsInHand` that takes as input parameters a number `n` between 1 and 13, representing a card value and a Player `p` and returns `true` if and only if the card is held by `p`.

ii) When the computer plays a human player, the program must choose the card for the program to ask for from the human player. Write a function

```
function chooseCard(p:Player):int
pre p.name = "Computer" and p.num>0
%post 1 <= result <= 13
```

that returns the value of the card that is required. You may call `cardIsInHand` if needed. Why must the pre-condition `p.num>0` hold?

c i) Given two players `p1` and `p2`, give the condition for the game between them to be over.

ii) Write a procedure `gameOver` that should be called when the game is over. It should congratulate the winner or announce a tie. It should print out the number of pairs the winner is holding.

The three parts carry, respectively, 35%, 35% and 30% of the marks.

Section B *(Use a separate answer book for this Section)*

3a Define the Abstract Data Type *B-Tree of order n.*

For a B-Tree of order 4, give the Turing function headers and pre- and post-conditions for the access procedures.

b Write the Turing code for the type declarations for a dynamic implementation of BTree4, the ADT B-Tree of order 4, where the data items are integers.

c Write the Turing code for the following high-level functions; you may assume the standard ADT List and its access procedures, and the ADT BTree4, but you should give any additional functions in full:

```
function enlist(T4:BTree4):List
% pre: takes a B-tree of order 4
% post: returns a list containing the  items of T4 in increasing order

function RangeQuery(T4:BTree4,low,high:int):boolean
% pre: takes a B-Tree of order 4 and two integer values, low<=high
% post: returns True if there are any values i in T4 such that
%            low<=i<=high
```

The three parts carry, respectively, 40%, 20%, 40% of the marks.

[Turn over

4a This question is about the implementation of a queuing system for a service organisation. An abstract data type CQueue (Customer Queue) is to be used to hold the data about all the customers. Each customer is identified by name and has an associated priority, which is an integer value, with higher value representing higher priority.
Access Procedures are:

```
function EmptyCQ() :CQ
%post: returns an empty CQueue

function IsEmptyCQ(C:CQueue):boolean
%pre: takes a customer queue, C
%post: returns True if C is empty, False otherwise

function AddCustomer(P:Customer,C:CQueue):CQueue
%pre: takes a customer, P and a customer queue, C
%post: returns a customer queue with P added to the customers in C

function ServeCustomer(C:CQueue):CQueue
%pre: takes a customer queue, C
%post: returns a customer queue without the highest priority customer in C

function NextCustomer(C:CQueue):Customer
%pre: takes a customer queue, C
%post: returns the highest priority customer in C; C is unchanged
```

Compare alternative approaches to the implementation of CQueue using the ADTs List, Queue, PriorityQueue and Heap, all with dynamic implementation. You should include the operations of AddCustomer and ServeCustomer and their speed when applied to a CQueue of n customers.
State which data structure you choose and give the type declarations for it in Turing code.

b Using the data structure you have chosen in your answer to part a, draw a diagram to illustrate your implementation with the following data, given as (name, priority), in your CQueue:
(Smith, 6), (Brown, 4), (Green,12), (Black,8), (Jones,5), (Pink,9), (Blue,14), (French,11), (Welsh,10), (German,5).

Describe the operations which take place when customer Williams with priority 8 is added to this customer queue.

c Write the Turing code for two versions of the following function:
i) as a high-level function independant of the data structure
ii) as an additional access procedure for the data structure in your answer to parts a and b.

```
function MergeQueues(C,B:CQueue):CQueue
%pre: takes two Customer Queues
%post: returns a single Customer Queue containing all the items of B and C.
```

The three parts carry, respectively, 40%, 30%, 30% of the marks.

Section C *(Use a separate answer book for this Section)*

5 a Describe briefly the conceptual differences between objects and data structures, classes and modules. Compare and contrast the basic principles of object-oriented programming and procedure-oriented programming.

b An **account** class has the following methods:

setaccount(X:Int, Y:Int)	:	set an account with number X and the initial value of Y
getbalance(var X : int)	:	assign the balance of the account to X and print it
deposit(X:Int)	:	add amount of X into the account
withdraw(X:Int,)	:	take out amount of X from the account if the account balance is more than X otherwise report error

i) write an OOT program *account.tu* to implement the class

ii) write an OOT program to set up two accounts (account 1 and account 2) with different amounts of money and then make transactions between the accounts to make them have equal amounts of money.

iii) use inheritance to implement a new class **accountnew** which has the same behaviour as that of the class **account** except that each withdrawal is charged a fee of 1% of the amount of money withdrawn by the bank.

The two parts carry, respectively, 25% and 75% (25% for each subpart) of the marks.

Turn Over

6a Given a class hierarchy as in the following diagram:

$$\begin{tikzcd}
 & & A \arrow[dl] \arrow[dr] & & \\
 & A1 \arrow[dl] \arrow[dr] & & A2 \arrow[dl] \arrow[dr] & \\
A11 & & A12 & & A21
\end{tikzcd}$$

i) Given a pointer p declared as var p : ^A, write down the possible dynamic classes of p.

ii) Given a pointer q, which at run time locates the objects of class A2, what type declarations for a pointer s would definitely make the assignment q : = s legal ?

b Given a class **queue**, implemented by the OOT program *queue.tu*, with the following methods:

Empty:	:	generate an empty queue
IsEmpty	:	test if a queue is empty
Enqueue (X: int)	:	add an integer into a queue
Dequeue (var X: int)	:	remove the integer at the front of a queue and assign the removed integer to X

The implementation uses the array *content* to store the elements of a queue and the number of elements in a queue is stored in the variable *count.* A **priorityqueue** is just like a queue, except that each time an Enqueue occurs, the element is inserted into the queue at the location determined by its priority. In the case of a queue of integers, the priority is determined as adding a new integer element into the queue in front of all other elements in the queue whose values are greater than it. Write an implementation of the class **priorityqueue** by inheriting the class **queue** and overriding the appropriate operation.

c The generic queue **gqueue** containing pointers to objects of any class. It is implemented by the OOT program *gqueue.tu.* and has the same 4 methods as in class **queue** with the Enqueue and Dequeue changed as follows:

Enqueue (X: ^anyclass)	:	add an element into a queue
Dequeue (var X: ^anyclass)	:	remove the element at the front of a queue and assign the removed element to X

The implementation uses the array *content* to store the elements of a queue and the number of elements in a queue is stored in the variable *count.* Implement a generic priority queue **priorityqueue** by inheriting the class **gqueue** and overwriting the Enqueue method. (You need to define the proper data structure where priority is specified using integer.)

The three parts carry, respectively, 25% (12.5% each subpart), 40%, and 35% of the marks.

End of paper