UNIVERSITY OF DUBLIN

TRINITY COLLEGE

Faculty of Engineering and Systems Sciences
Department of Computer Science

B.A.Computer Science
Senior Freshman Examination

Trinity Term 2002

2BA2 - Programming Techniques

Saturday 25th May | Exam Hall | 14.00 - 17.00

Dr. Hugh Gibbons

Attempt FOUR questions

(In presenting programs explain clearly the design of the Eiffel code)

Qs 1.

The following routine, ins_sort, sorts an array using the algorithm 'sorting by insertion'.

```
ins_sort (a:ARRAY[G]; low,high:INTEGER) is
        require
                a /= void and then low <= high
                a.lower <= low and high <= a.upper;
        local
                k: INTEGER;
        do
                from
                        k := low + 1
                until
                        k > high
                loop
                        search(a, low, k-1, a.item(k));
                        insert(a.item(k), index+1, a, low, k-1);
                        k := k + 1
                end
        ensure
                sorted: is_ordered(a, low, high)
        end ; -- ins_sort
```

The routine call,

```
search(a, low, k-1, a.item(k))
```

searches where to insert the item, a.item(k), into the ordered array segment a[low .. k-1] using 'a binary search technique'.

The routine call,

```
insert(a.item(k), index+1, a, low, k-1)
```

inserts the item, a.item(k) at position, index+1, in the array segment, a[low .. k-1].

i) Present an Eiffel routine

```
Search(a:ARRAY[G]; low,high:INTEGER; x:G) is
        require
                Ordered:  Is_Ordered(a,low,high)
        ensure
        --(found → x = a.item(index)) & (¬found → x ∉ a[low..high]
```

that 'binary searches' an array segment a[low .. high] for an item x.
If there is more than one occurrence of x, it returns the position of the 'rightmost' one, i.e. if a.item(index) = x, then a.item(index+1) > x.

ii) Present an Eiffel routine

```
insert(x:G; i:INTEGER; a:ARRAY[G]; low,high:INTEGER)
        require
                a /= void and low <= high
                a.lower <= low and high < a.upper
                low <= i and  i <= high
        ensure
                        equal(a.item(i), x)
```

that inserts x at position i in array, a, by moving items to the right.

iii) Given an array attribute, a:ARRAY[G], present an Eiffel routine

reverse(low, high : INTEGER)

that reverses the array segment a[low .. high].

If the array, a, was in ascending order then the call

reverse(a.lower, a.upper)

changes the array, a, into descending order.

Qs. 2

a) Implement boolean function

is_equal(a, b : ARRAY[G]) : BOOLEAN

which determines whether or not the arrays, a and b, are equal item by item.

b) Present an Eiffel procedure,

partition(L0, R0 : INTEGER; p : G)

that will partition an array attribute, A:ARRAY[G], about a pivot, p, such that after partition we will have

A[L0 .. R] ≤ p ≤ A[L .. R0]

where L and R are integer attributes.

c) Using a procedure for partitioning an array, present a procedure that will Quicksort an array, A:ARRAY[G].

Qs 3

Present Eiffel routines that will

a) Generate all n! permuations of {1..n} in order starting with the 'largest' n, n-1, ..., 2,1 and ending with the 'smallest' 1,2,3,...,n

b) Generate all 2^n subsets of {1..n} starting with the set {1,2,...,n} and finishing with the empty set, {}.

Q 4

Assume we are given the classes, LIST_BAG and NODE with the following short forms (repeated items are allowed),

```
class interface LIST_BAG [G]

    add_first(x : G)
    -- add x, in new first node

    add_last(x:G)
    --add x in new last node

    count : INTEGER

    empty : BOOLEAN

    has (x : G) : BOOLEAN

    remove_first_node
    -- remove first node, if any.

    remove_last_node
    -- remove last node, if any

    reverse
    -- reverse the list

    <other routines>

end -- class LIST_BAG
```

```
class interface NODE[G]
    item : G
    next : NODE[G]

    set_item(x : G)
    set_next(n : NODE[G])

end -- NODE
```

Implement, using linked nodes, the routines,

add_first(x : G), add_last(x:G), remove_first_node, remove_last_node

and the routine

reverse, which reverses the linked list.

Use linked list diagrams in explaining the routines.

Qs. 5.

Assume that a Directed Acyclic Graph (DAG), D, is stored as an adjacency list.

Present Eiffel routines that will

a) Breadth First Traverse the DAG, D.

b) Toplogical Sort the DAG, D.

Qs 6

Assume class interfaces for a Binary Search Tree class, BST, and a BIN_NODE class as follows: (Repeated items in BST.)

```
class interface
        BST [G -> COMPARABLE]
        -- Binary Search Tree
feature

        empty -- make empty

        is_empty: BOOLEAN

        add (x: G)
        -- add x to BST;
        -- (repeated items allowed)

        array2bst(a:ARRAY[G])
        -- add all items from array, a, to
        -- BST.

        bst2array : ARRAY[G]
        --puts items in BST into an array
        -- so that the array is ordered.

        root: BIN_NODE [G]

        count: INTEGER -- # items in tree

end -- class BST
```

```
class interface
        BIN_NODE [G]
feature

        value: G;

        left: BIN_NODE [G];

        right: BIN_NODE [G];

        value_set (v: G)

        left_set (n: BIN_NODE [G])

        right_set (n: BIN_NODE [G])

        build (v: G; l, r: BIN_NODE [G])

end -- class BIN_NODE
```

Implement the routines

add, array2bst, bst2array

in the class BST

Suggestion:

The routine, add(x : G), may use an auxiliary routine, insert, as follows:

```
Add(x:G) is
        do
                if root /= void then
                        insert(x,root)
                else
                        !!root
                        root.build(x,void,void)
                        count := 1
                end
        end -- Add
```

Implement the routine,

insert(x : G; t : BIN_NODE[G])

that inserts an item, x, into a tree with root, t.

The routine, bst2array, may use an auxillary routine

inord(t:BIN_NODE[G])

as follows:

```
bst2array : ARRAY[G] is
        require
                not is_empty
        do
                !!arr_ord.make(1,size)
                index:= 1
                inord(root)
                result := clone(arr_ord)
        end -- bst2array
```

The array, arr_ord, and integer, index, are hidden attributes of the class. The counter, index and array, arr_ord, may be updated during the execution of inord. The routine, inord, inorder traverses the tree, starting at the root, and adds the items of the tree to the array, arr_ord. Implement the routine, inord.