

QUEEN MARY, UNIVERSITY OF LONDON

DCS128 ALGORITHMS AND DATA STRUCTURES

Class Test Monday 4th April 2005 10.35-11.55

Please fill in your Examination Number here:

Student Number here:

All answers to this test should be written on the test sheet, but you may use spare paper for rough working. Answer as many of the questions as you can.

- 1) Explain in the space below what an **ordered binary tree** is, and explain informally how it is used to maintain a set of objects.

A binary tree is a data structure which is either empty or consists of a data item (referred to as the "root item") and two subsidiary binary trees (referred to as the "left branch" and "right branch").

A binary tree is ordered if it is empty, or if the left branch is ordered and the right branch is ordered, and any item in the left branch is less than the root item and any item in the right branch is greater than the root item in some ordering. An item is in a binary tree if it is the root item, or if it is in the left branch or if it is in the right branch.

To represent a set, an ordered binary tree must represent the operations of membership test, adding and deleting an item.

To test whether an item is a member of the set represented by a tree, if the tree is empty, it is not. If the root item is equal to the item whose membership is being tested, then it is a member. If the item being tested is less than the root item, then it is a member of the set if and only if it is a member of the set represented by the left branch of the tree. If the item being tested is greater than the root item, it is a member of the set represented by the tree if and only if it is a member of the set represented by the right branch.

To add an item to a set represented by a tree, if the tree is empty, it is replaced by a tree whose root item is the item added and whose left and right branches are empty. Otherwise, if the item is less than the root item, the left branch is changed to represent the addition, and if the item is greater than the root item, the right branch is changed to represent the addition. If the item is equal to the root, the tree is left unchanged, keeping the set property that an item occurs at most once in a set.

To delete an item from a set represented by a tree, if the item is less than the root item, the left branch is changed to represent the deletion. If the item is greater than the root item, the right branch is changed to represent the deletion. If the tree is empty it is left empty, representing deletion having no effect if the item being deleted is not in the tree. If the item being deleted is the root of the tree, then if the left branch is empty the tree is replaced by its right branch, and if the right branch is empty the tree is replaced by its left branch. If the item being deleted is the root of the tree and neither the left branch nor the right branch are empty, the rightmost item of the left branch is deleted from its place and the root item is replaced by this item.

It can be seen that all of these operations work by recursively considering either the left branch or the right branch if the item being checked/added/deleted is less than or greater than the root item. The addition and deletion algorithms maintain the ordered property of the tree.

- 2) In this question you can assume the implementation in Java of an abstract data type called `List`, which stores a collection of integers and has the following operations given by non-static methods:
- `head` – returns the first item of the list
 - `tail` – returns a new list consisting of all the items from the list except the first item
 - `cons` – takes an integer argument and returns a new list whose `head` is that integer and whose `tail` is the list the `cons` operation was called on
 - `isEmpty` – returns `true` if the list it is called on is the empty list, `false` otherwise.
- There is also a static method called `empty` which returns a new `List` object representing the empty list.

The code you write should not assume anything about `List` objects except what is given above.

- a) Write a Java static method `replace` which takes as its argument a `List` of integers and two integers and returns a `List` of integers in which all occurrences of the first integer argument have been replaced by the second. For example, if the textual representation of the `List` is `[5,7,9,3,7,4,1,2,8,7,3]` and the integer arguments are 3 and 6, the `List` constructed has the textual representation `[5,7,9,6,7,4,1,2,8,7,6]`.

```
public static List replace(List list, int m, int n)
{
    if(list.isEmpty())
        return List.empty();
    else
    {
        List temp = replace(list.tail(),m,n);
        if(list.head()==m)
            return temp.cons(n);
        else
            return temp.cons(list.head());
    }
}
```

- b) Write in the space below a static method which takes two `List` objects as arguments and returns a `List` object which contains all the integers of the first `List` argument which also occur in the second `List` argument. The order of the integers in the `List` returned does not matter. You may define and use an auxiliary method if you need to.
- For example, if the two `List` arguments have textual representations `[3,10,4,7,8,12,6]` and `[5,4,12,9,8,13,3]` the `List` object returned will have the textual representation `[3,4,8,12]` or something with the same integers in a different order.

```
public static List intersect(List list1, List list2)
{
    List list3 = List.empty();
    for(; !list1.isEmpty(); list1=list1.tail())
        if(member(list1.head(),list2))
            list3 = list3.cons(list1.head());
    return list3;
}

private static boolean member(int n, List list)
{
    for(; !list.isEmpty() && list.head() != n; list=list.tail())
        {}
    return !list.isEmpty();
}
```

- 3) In the space below give a Java implementation of the abstract data type Queue , using a linked list representation with an extra pointer to the last cell.

```
class Queue
{
    private Cell front,back;

    public Object first()
    {
        return front.first;
    }

    public void leave()
    {
        front=front.next;
        if(front==null)
            back=null;
    }

    public void join(Object obj)
    {
        if(front==null)
        {
            front = new Cell(obj,null);
            back=front;
        }
        else
        {
            back.next = new Cell(obj,null);
            back=back.next;
        }
    }

    public boolean isEmpty()
    {
        return front==null;
    }

    private static class Cell
    {
        Object first;
        Cell next;

        Cell(Object f,Cell n)
        {
            first=f;
            next=n;
        }
    }
}
```

- 4) a) Describe briefly the difference between **pre-order** and **post-order** tree traversal.

These are both traversals of the tree in which the traversal is broken into three steps: a recursive traversal of the left branch, a recursive traversal of the right branch, and an operation on the root. The difference between the two is that a pre-order traversal does the operation on the root before traversing either of the branches, while a post-order traversal does the operation on the root after traversing both branches.

- b) Explain briefly the technique which is used to obtain **breadth-first** tree traversal.

The technique used to traverse a tree breadth-first is to use a queue of trees. The queue is initially set to contain the whole tree being traversed. Then, iteratively, the first tree is taken from the front of the queue, an operation is done on its root and its left and right branches put on the back of the queue. This continues until the queue is empty.

- c) Explain, without using code, the technique used to **join** one linked list to the end of another **destructively**.

If the first list is null, then the join of the two lists is equal to the second list. Otherwise, a pointer is set to the first list. It is then repeatedly reset to the 'next' field of the cell it is pointing to until it points to a cell whose 'next' field is null (which must be the last cell in the list). Then the 'next' field of that cell is set to the second list.

- d) Explain, without using code, the “**stack and reverse**” technique for copying a linked list.

A pointer is set to the list being copied, and a variable representing a stack is set to null. Repeatedly, the stack variable is set to refer to a new cell whose data item is the data item of the cell the pointer points to and whose 'next' field is the previous value of the stack variable; then the pointer is then reset to the 'next' field of the cell it points to. This is done until the pointer has the value 'null'. The stack will then be a copy of the original list, but with the data in reverse order. To produce a copy in the correct order this whole process is repeated with the pointer running down the stack and a new stack being created by copying the data items as before. When the pointer becomes null, the new stack will be a copy of the original list.

- 5) Describe **three** different data structures that may be used to implement the abstract data type “sequence” (that is, a list with an insertion point). In each case explain briefly how the data structure represents the abstract data type.

A sequence is a collection of data which has the concept of being in a particular order and of there being an ‘insertion point’ in that order. The operations on it are to delete the item at the current insertion point, to add a new item at the current insertion point, and to move the insertion point backwards and forwards.

One way of representing this is as an array of a fixed size (the maximum size the sequence is assumed to need) with two integers. One integer gives the size of the sequence with the array up to but not including the cell indexed by this array containing the current sequence data in order. The other integer indexes the current insertion point. Moving the insertion point is represented by incrementing or decrementing the integer representing it. Adding a new item is represented by moving all the items above the insertion point one space up in the array and putting the new item in the space cleared. Deleting an item is represented by moving all the items in the array after the insertion point one place down.

A second way of representing a sequence is as a doubly-linked list. This is an ordered list of cells each of which has a pointer to two cells, one (the ‘backwards’ pointer) to the cell before it in the list, the other (the ‘forward’ pointer) to the cell after it in the list. The sequence is then represented by a pointer to a cell in this structure, the cell it points to being the current insertion point. Moving the insertion point backwards or forwards is represented by resetting the pointer to the backwards or forwards pointer of the cell it points to. Adding a new item is done by creating a new cell to store the item and resetting the links so that this cell is inserted into the doubly-linked list at the insertion point. Deleting an item is done by resetting the links so the cell the pointer originally pointed to is cut out of the list and the pointer rest to the cell its forward pointer pointed to.

A third way of representing a sequence is as two singly linked lists. The cells in one store the data of the sequence after the insertion point in the order it occurs in the sequence. The cells in the other store the data at the insertion point in its first cell and the data before the insertion point after that in reverse order to the order it occurs in the sequence. Adding a new item to the sequence is represented by adding a new cell storing the item to the front of the ‘before’ list. Deleting an item is represented by setting the ‘before’ list to the ‘next’ field of its first cell. Moving forward is represented by taking the first cell off the ‘after’ list and putting it at the front of the ‘before’ list. Moving backward is represented by taking the first cell off the ‘before’ list and putting it at the front of the ‘after’ list.