CS1Ah Question Sheet 4

Programming with Recursion, Stacks, and Trees

The following questions are designed to promote understanding of the material in CS1Ah. Your tutor may discuss some of the questions, and some may be set as homework. Work on these questions is not formally assessed, so you may discuss answers with your fellow students (remember that this does not apply to practical exercises). When problems on this sheet are set as homework, there is no reason to copy solutions from somebody else; you should think about the questions yourself, so you can discuss answers at a tutorial. If you have difficulty, you may ask a lab demonstrator for help or consult the solution sheet on the web, when it is issued.

1. Dynamically resizing arrays
Consider the start of the definition of a class for an ordered collection of objects which uses an array in a similar way to the ClassOfStudents classes from Lecture Note 17.

```java
public class MyArrayList {

    // static fields.
    public static intMaxSize = 10;
    public static scaling = 2;

    // instance fields.
    private int num;
    private Object[] elements; // elements[0] ... elements[num-1] in use

    // Constructor
    public MyArrayList() {
        elements = new Object[initMaxSize];
        num = 0;
    }
}
```

a) Add to this class methods isFull(), addLast() and removeLast(), where the latter two methods work on the array around the position given by the index number.
b) Modify `addLast()` so that, if trying to add to a full array, the array contents is first copied to a new array of size `scaling` times larger. Use the `System.arraycopy()` method to do the copy.

2. **Sierpinski Gasket**

The Sierpinski Gasket is defined by the following procedure.

1. Draw a triangle.
2. Find the midpoints of each of the triangle’s side.
3. Repeat for each smaller triangle formed by one corner of the original triangle and the two closest midpoints.

![Sierpinski Gasket at different depths of recursion](image)

Figure 4.1: The Sierpinski Gasket at different depths of recursion

Write a program as a method of a class `Sierpinski` to draw the gasket up to a depth given by a command line argument. For your convenience, a class `DrawArea` is provided that you can use for drawing lines in a window. A summary of the parts of the class you could use is:

```java
public class DrawArea {
    public DrawArea(String title, int width, int height)
    public void drawLine(Point p, Point q)
    public void repaint()
}
```

Its `main` method illustrates how it can be used:

```java
public static void main(String [] args) {
    DrawArea a = new DrawArea("Sierpinski Gasket", 400,300);
    a.drawLine(new Point(100,100), new Point(300,100));
    a.drawLine(new Point(300,100), new Point(200,250));
    a.drawLine(new Point(200,250), new Point(100,100));
    a.repaint();
}
```

You will want to include the line

```java
import java.awt.*;
```

at the top of your `Sierpinski.java` file in order to bring in the definition of the `Point` class from the Java library.
3. Memoizing Functions
The common recursive version of the Fibonacci function (see question 6 of Question Sheet 3: Further Java Programs) is very inefficient: the number of calls grows exponentially with argument value. A standard technique to improve the efficiency of recursive programs like this is to maintain a cache of previously calculated values. Each time one starts the body of the recursive function one first checks in the cache if the value has already been calculated, and if it has, the cached value is returned. If the cache entry is empty, the body of the function is executed as normal, and, just before returning, the return value is stored in the cache.

Write a recursive version of Fibonacci that uses caching. Use an `int[]` array with the `ith` entry in the array being either `0` to indicate the cache entry is missing, or the value of `F(i)` (always greater than `0`).

Add a counter to your implementation to count the number of times the body of the Fibonacci function is executed, and a flag to enable or disable caching.

Write a `main` method to print out the number of calls with caching enabled and disabled for a range of input arguments.

4. Tail Recursion and Iteration
The common recursive version of factorial:

```java
public static int recFact(int n) {
    if (n == 1) {
        return 1;
    } else {
        return recFact(n-1) * n;
    }
}
```

can be rewritten in what’s called a **tail recursive** style:

```java
private static int tailRecFactAux(int k, int kFact, int n) {
    if (k == n) {
        return kFact;
    } else {
        return tailRecFactAux(k+1, kFact*(k+1), n);
    }
}

public static int tailRecFact(int n) {
    return tailRecFactAux(1,1,n);
}
```

The argument `kFact` is so called because it is always equal to `k` factorial.

In this style, the recursive call only occurs as a last statement in the function body: nothing further is done with the result value of each recursive call than simply return it. With such a style it is not necessary that a new activation record be created for each recursive invocation, and it is straightforward to transform the recursive definition into an iterative one. For the factorial function, a hand translation yields:

```java
public static int itFact(int n) {
    int k = 1;
    int kFact = 1;
```
```java
int result;
while(true) {
    if (k == n) {
        result = kFact;
        break;
    }
    // Setup of arguments to recursive call.
    int newK = k+1;
    int newKFact = kFact*(k+1);
    int newN = n;
    // effect of recursive call loop back
    k = newK;
    kFact = newKFact;
    n = newN;
}
return result;
}
```

Write a tail recursive version of the Fibonacci function and transform it by hand to an iterative version in a similar way. Hint: have the argument list of each recursive call include the values of 2 adjacent Fibonacci numbers.

This transformation is not surprisingly called *tail recursion elimination* and smart compilers often can do it automatically.

5. **Follow-on from Lecture Note 21 on linked lists.**

a) Obtain copies of the singly-linked-list files from the CS1 webpage. Make a copy of the `SLinkedList.java` file, calling it `SLinkedList1.java` and rename the class inside similarly. Add new method `addLast()` that works from the start of the linked list. Create `Test1.java` from `Test.java` to test this new method.

b) Make another copy of the original `SLinkedList` file and class called `SLinkedList2`. As discussed in lecture, modify the class to include an extra instance field `lastLink` which should hold a reference to the last link of the list. As with `firstLink` it is appropriate to have it set to `null` when the list is empty. Modify the existing methods `addFirst()` and `removeFirst()` to maintain this extra field: you’ll need to handle new special cases for when the list is empty with `addFirst()` and for when it has one element with `removeFirst()`.

c) Add new method `addLast()` to class `SLinkedList2` making use of this new field. Modify `Test.java` to create `Test2.java` to check your method works.

6. **Recursion on Lists**

Add `reverse`, `appendTo`, and `sublistOf` methods to the class `SLinkedList1` of question 5. The methods have the following specifications:

- `t.reverse()` reverses the list (t). For example, the reverse of (1, 2, 3) is (3, 2, 1).
• u.appendTo(t) appends the list u to the list t. For example, if u is (1, 2) and t is (3, 4), then after executing u.appendTo(t) the list u is (3, 4, 1, 2).

• u.sublistOf(t) returns true if the elements of the list u can be found in the list t, in the same order, but possibly with other elements in between. For instance, (1, 2) is a sublist of (3, 1, 4, 5, 2, 6), but (4, 1) is not. The method should have no side-effects, i.e., after execution of u.sublistOf(t) the list u should be the same as before.

Use recursion to implement the methods. You are only allowed to use the methods isEmpty, addFirst, getFirst and removeFirst. You are not allowed to access the fields of Slink1 objects.

If your implementation of the method addLast in question 5 was not recursive, give now a recursive one.

7. Recursion on Trees
Consider the BinTree class given in Lecture Note 24.

a) Add printNodesPostorder and printNodesInorder methods which print the data stored in nodes, traversing the trees in the appropriate order. For the in-order traversal, print parentheses around each subtree to disambiguate the output.

b) Add a method sumIntegers. The expression t.sumIntegers() should return the sum of the integer data objects in the tree t. Other objects should be ignored. (Hint: use instanceof).

c) Write a method evalExpr which evaluates an expression tree stored in a Bin-Tree. Operators are stored in nodes as String objects, operands are stored as Integers.

d) (Harder). Write a method printNodesBreadthFirst which prints the nodes of the tree in a breadth first order. In this ordering, the nodes at each depth level are printed from left to right, starting from the root and working down the tree. (Hint: use a queue data structure.)

8. Sorting using Binary Search Trees
Write a Java program which implements a tree sort algorithm. The algorithm sorts an array by putting the elements one by one into a binary search tree as described in Lecture Note 25, and then repeatedly removes the least element and puts it back into the array. TheSearchTree.java file from the lecture note is available on-line at: http://www.dcs.ed.ac.uk/teaching/csi/CS1/Java/SearchTrees/SearchTree.java.

9. Heaps
In this question we consider trees whose nodes are labelled by comparable data, say integers. In the following discussion, we abuse language and identify a node with its label.

A heap is a binary tree which satisfies the heap property: every node is larger than or equal to its children. (Actually, heaps are also assumed to be balanced binary trees,
but we ignore this for the moment: see question 10). In particular, the root is the maximum of all nodes.

Assume we know that the left and right subtrees are heaps. However, the tree itself may be a heap or not. We can transform the tree into a heap by means of the following algorithm, usually called *heapify*:

- Let \( l \) be the largest of the root and its two children (if they exist).
- If \( l \) is the root, do nothing: the tree is a heap.
- If \( l \) is the left (right) child, then exchange the root and the left (right) child; recursively apply *heapify* to the left (right) subtree.

Now, given an arbitrary tree, we can transform it into a heap by *heapifying* its subtrees bottom-up. That is, we heapify the left and right subtrees of of a tree before heapifying the tree itself.

a) Take a full binary tree of depth 3 (i.e., every path from the root to a leaf contains exactly three nodes), and transform it by hand into a heap using the procedure above.

b) Define a class *Heap* using the *BinTree* class of Lecture Note 24. An object of the class should have one single field *root* of type *Bintree*. The class should have methods *heapify*, *buildHeap*, *insert*, *getMax*, and *extractMax* with the following specifications:

- \( t.heapify() \), where \( t \) is of type binary tree, applyes to \( t \) the *heapify* procedure described above.
- \( u = \text{buildHeap}(t) \), where \( t \) and \( u \) are of type binary tree, assigns to \( u \) the result of heapifying all subtrees of \( t \), bottom-up. This method is used to define the only constructor of the class *Heap*:
  \[
  \text{Heap(BinTree n) \{root = \text{buildHeap}(n);\}}
  \]
- \( h.insert(n) \), where \( h \) is a heap and \( n \) is an integer, adds a new node with label \( n \) to the heap \( h \). After execution \( h \) must still satisfy the properties of a heap.
- \( h.getMax() \) returns the maximum integer stored in the heap.
- (Harder.) \( h.extractMax() \) removes the maximum node from the heap, and reorganises the rest into a heap. *Hint:* replace the root by one of the leaves and then use *heapify*. 

c) Write a program which builds a balanced tree using the constructors of *BinTree*, prints it, transforms it into a heap, performs several operations on the heap, and prints the results.

A class implementing the methods *insert*, *getMax*, and *extractMax* is called a *priority queue*, and has many applications in scheduling, event-driven simulation, and as a data structure in many important algorithms.
10. Implementing balanced trees as arrays

A binary tree is balanced if there is a number $k$ such that the depth of all leaves is either $k$ or $k + 1$. Moreover, if a leaf has depth $k + 1$, then all leaves to its left also have depth $k + 1$. In other words, a binary tree is balanced if there is a number $k$ such that the first $k$ levels of the tree are full, and the $k + 1$-th level 'is being filled' starting from the left.

A balanced tree with $n$ elements can be implemented using an array $A[1..n]$, as follows:

- the root of the tree is stored in $A[1]$;
- the left child of $A[i]$ is stored in $A[2i]$;

As a consequence, the parent of $A[i]$ is stored in $A[⌊i/2⌋]$, where $⌊i/2⌋$ denotes the result of rounding $i/2$ down.

a) Show that all the elements of the array store one and exactly one node of the tree.

b) Define a class `BalancedTree` in which balanced trees are implemented as arrays. Implement methods `printNodesPostorder`, `printNodesInorder`, `sumIntegers` and `printNodesBreadthFirst` as in question 7.

c) Reimplement the class `Heap` of question 9 using `BalancedTree` instead of `BinTree`. (This is in fact the usual implementation of a heap).

Paul Jackson, Jim O'Donnell, David Aspinall, Mary Ellen Foster, Javier Esparza, 2003/01/03 17:06:16.