British Informatics Olympiad Final

1–3 April, 2004

Sponsored by Lionhead Studios

Garbage Collection

This question is quite long — do not worry if you do not get to the end

Many modern computer languages, including Java, Haskell, OCAML and C#, provide a facility known as garbage collection for freeing memory. In this question, we will analyse a two algorithms for garbage collection in the context of some simple data structures.

Consider a data structure built out of two kinds of node: *terminal nodes*, which hold a single letter, and *internal nodes* which consist of two pointers, called *left* and *right*. Each pointer can either point to nothing (which we shall write as ∅) or to another node. There is no limit on the number of different pointers that can point to a given node. Finally, there is an extra pointer, called the *root*, which is not part of any node.

Here are two example data structures:

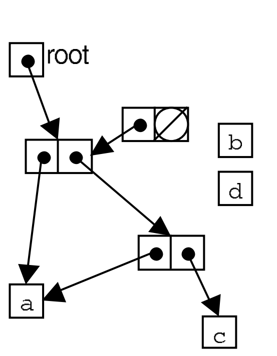


Figure 1

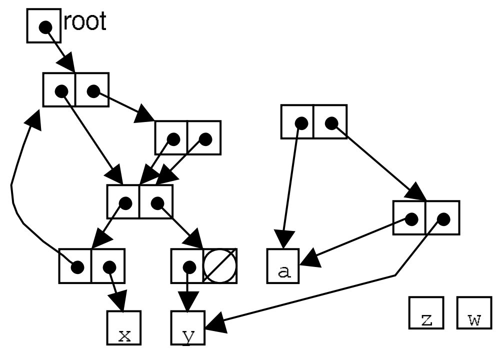
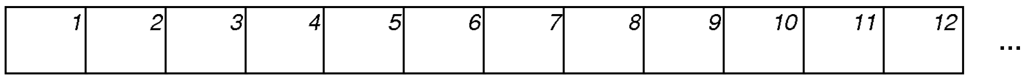


Figure 2

We store data structures in *memory*. Memory consists of a number of *cells*, which are numbered. We picture the cells as a long array, as follows:



Each cell can hold a node, or be *empty*. The *root* is a pointer into memory, which is not part of any cell. These cells can store a data structure. For example, the following memory-diagram stores the data structure from figure 1:

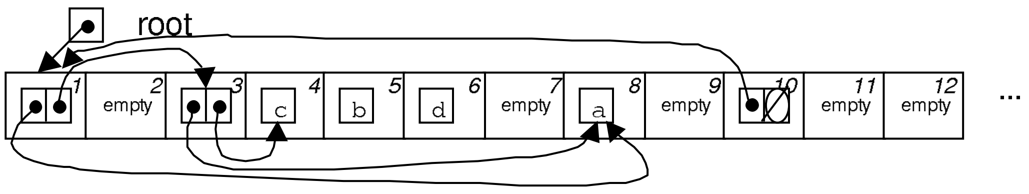


Figure 3

Question 1

Draw a memory-diagram containing the data structure in figure 2.

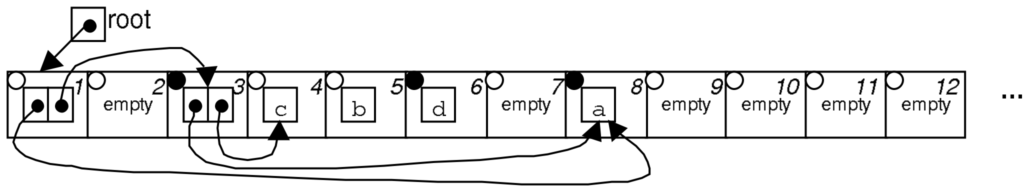
Users can only access the data in memory by starting at the root and following pointers. The user cannot 'see' the actual memory-diagram, but only the parts of the data structure it contains which can be reached in this way. Any nodes which cannot be accessed in this way are called *garbage*. We can safely remove them from memory without the users ever noticing, and replace them with empty cells which can then be used for storing new nodes. In the memory-diagram of figure 3, cells 5, 6 and 10 are garbage.

Question 2
Which cells in your solution to question 1 are garbage? (You should refer to the cells by their numbers.)

The user may at any time change a pointer in an accessible cell to point to any other accessible cell, or to ∅. When this happens, new garbage may be created; so, without a scheme for turning garbage back into empty space, the amount of garbage would increase over time.

Mark-sweep collection

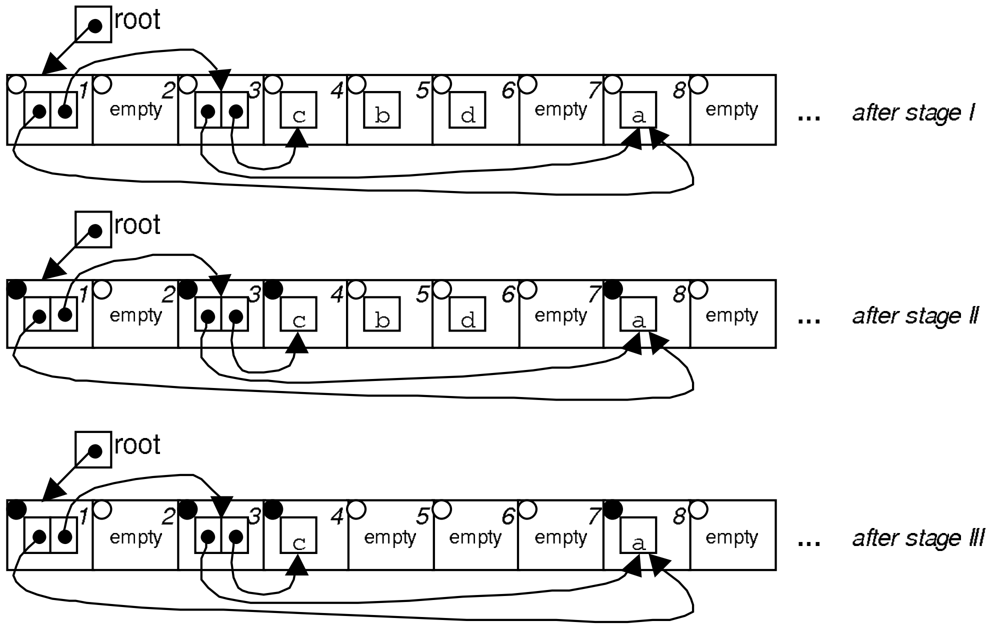
One possible garbage collection scheme is called *mark-sweep*. In this scheme each cell additionally contains a *mark bit* which can be either *set* or *unset*. (Empty memory-cells must always have an unset mark bit.) We shall depict a set mark bit by • and an unset mark bit by ∘. Here is a memory-diagram for figure 1 showing mark bits:



The algorithm for mark-sweep consists of three stages:

I Loop through each memory cell in turn. Unset the mark bit of each cell.
II Set the mark bits of all cells which can be reached by following pointers from the root.
III Loop through each memory cell in turn. If the mark bit of the cell is clear, empty the cell.

For example, this is the effect of the algorithm on the above diagram;



Question 3
Briefly outline a recursive algorithm one could use to do step II. Explain why your algorithm always finishes in a finite number of steps.

Normally, the part of the system which is responsible for garbage collection is also responsible for providing an *allocator*. When the user wants to modify the data structure stored in memory by adding a new node, they call the allocator. It returns an empty cell, into which the user can place the new node. A simple way of implementing the allocator is as follows. It uses a special extra pointer into the memory, called the *runner*.

- We start with the runner pointing at the first cell in memory.
- Whenever we need to allocate a memory cell, we check to see if the runner points at an empty cell. If it does, we return that cell. Otherwise, we repeatedly increment the runner to the next memory cell and try again.
- If we get to the end of the memory cells without finding an empty cell, then all cells must be full. We then run the garbage collection algorithm, reset the runner to the first cell in memory, and try again.

It is possible for us to modify this simple allocator so that we can simplify the garbage collector by removing steps I and III.

Question 4
Briefly explain how this can be done. Why might you want to do this?

Copying collection

An alternative garbage collection scheme is called *copying garbage collection*. In this scheme, we divide the available memory into two halves. Except during garbage collection, we only ever use one half at a time. We make sure that at all times, the non-empty cells form a contiguous block at the beginning of the used half. We keep a pointer to the first empty cell; to allocate, we simply return this empty cell (which the user then fills), and update our 'first empty cell' pointer to point to the next cell.

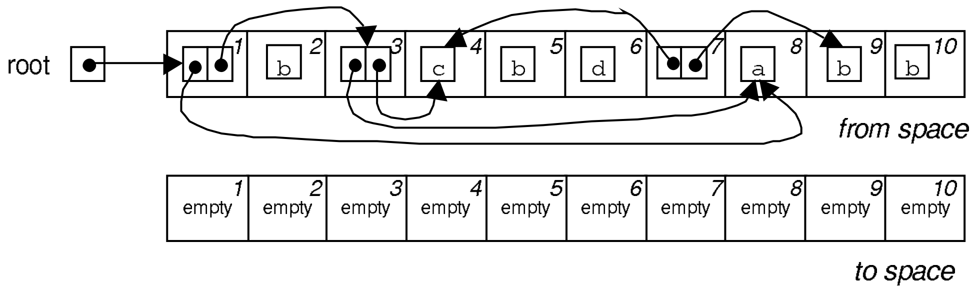
When this pointer reaches the end of the half we are using, we enter garbage collection mode. We then copy everything which is reachable from the first half (which we call the *from-half*) to the beginning of the other half (the *to-half*). (We can then safely empty the entire from-half.)

We do the copying as follows: we divide the to-half into three sections. The rightmost section is empty space. The middle section are nodes which have been copied to the to-half, but which may still point to nodes which are in the from-half; these are shaded grey. The leftmost section consists of nodes copied to the to-half whose pointers also now point to nodes in the to-half; these are shaded black.

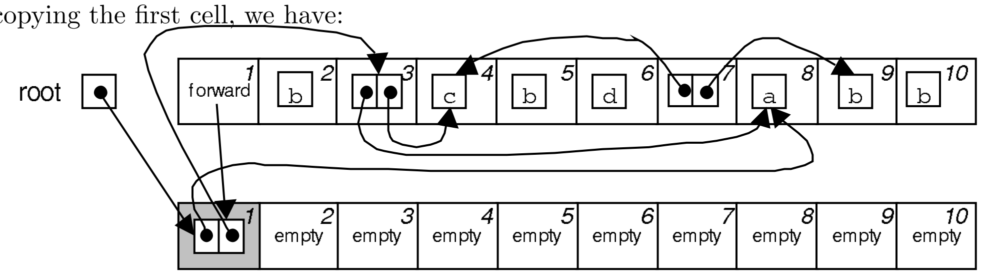
To begin, we copy the node pointed to by the root, and shade it grey. We then repeatedly pick the leftmost grey memory cell. If it contains a terminal node, we shade it black and continue. If it is an internal node I, we first look at its left child L; if L has not been moved to the to-half, we move it, and adjust the pointer in I to point at L's new location in the to-half. Otherwise, we just adjust the pointer. The same is done for the right child. We then colour the cell containing I black.

Finally, whenever we move a node to the to-half, we place a special *forwarding pointer* in its old position (in the from-half) pointing to its new position (in the to-half).

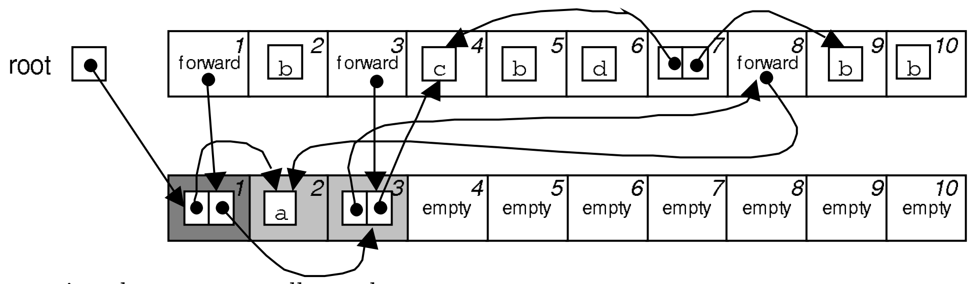
The algorithm finishes whenever there are no more grey nodes. For example, consider the following memory-diagram:



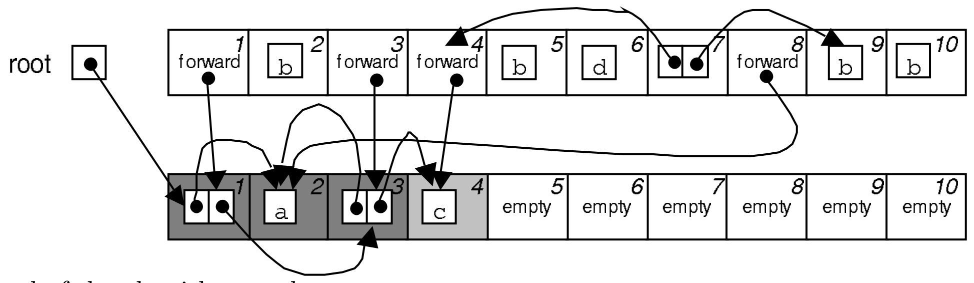
After copying the first cell, we have:



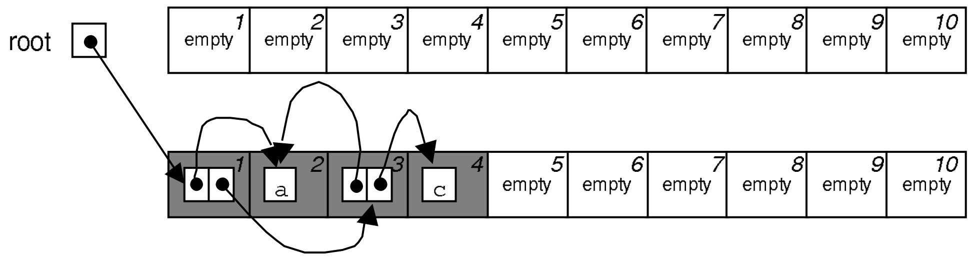
After processing the first cell from grey to black, we have:



After processing the next two cells, we have:



At the end of the algorithm, we have:



Question 5

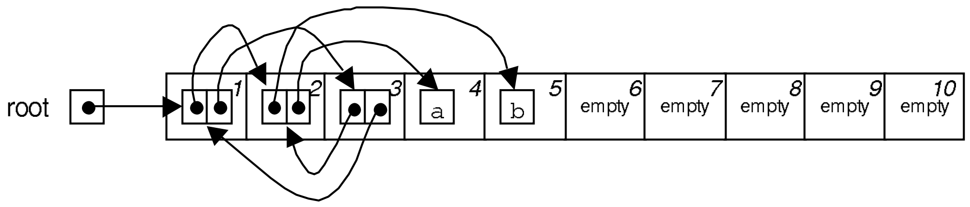
Apply the copying garbage collection algorithm to your solution to question 1. Draw the resulting memory-diagram at the stage where two objects have been coloured black, and at the stage where four objects have been coloured black. You need not continue the algorithm beyond this point.

Question 6

Consider the following part of the above algorithm: 'If L has not been moved to the to-half, we move it, and adjust the pointer in I to point at L's new location in the to-half. Otherwise, we just adjust the pointer.' Give simple pseudocode for an implementation for this step, explaining explicitly how the code tells if a node has already been copied. You may assume that you can tell if a cell contains a forwarding pointer or a node.

Question 7

The following diagram could not possibly have arisen as a result of running the algorithm above on a memory-diagram. (For brevity, only the to-space is shown; the from-space is blank.) Why not?



Question 8

Suppose someone modifies the copying algorithm so that when it processes an internal node, it no longer always processes the left part before the right part, but rather processes the parts in a random order. How many possible memory-diagrams could be produced if the input memory-diagram stores the data structure in figure 4 below? What about figure 5?

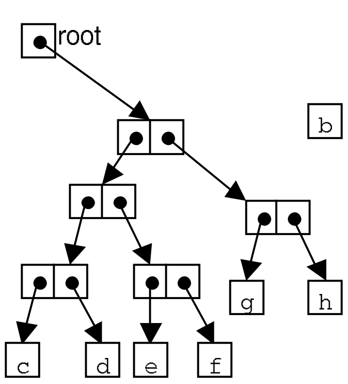


Figure 4

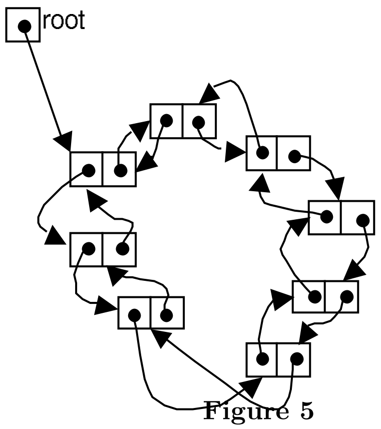


Figure 5

Question 9

Can you give a memory-diagram such that, when we run the modified algorithm from the previous question on it, the number of memory-diagrams that could be produced is not a power of 2? If so, give an example. Otherwise, explain briefly why not.

We now consider the relative merits of the two schemes we have discussed. Neither of the two algorithms we have discussed is better than the other in all respects.

Question 10

Suppose that there were other external pointers (like the root), which the garbage collector could see and read, but (unlike the root) not alter. As for the root, anything reachable from these other pointers would not be considered garbage. Would this pose a problem for mark-sweep? What about copying collection?

Question 11

Which of the two schemes is better in terms of fast allocation? Which is better in terms of needing fewer cells to operate with a data structure of a given size (number of nodes)? Justify your answer.

Question 12

Suppose that we were trying to adapt our chosen scheme to deal with a new kind of data structure, some of whose nodes require more than one adjacent cell in memory. How would this influence your choice between the two techniques? Justify your answer.

Generational systems

Real-life *generational* systems rely on a hybrid of the above schemes. They are based on the observation that 'most objects die young'; if an object, such as one of our nodes, has already survived one round of garbage collection, it will probably last for many more. Objects are split into two types: *old* objects, which have already survived one round of garbage collection, and *new* objects, which are all the rest. Roughly, they use mark-sweep for the old objects and copying collection for the new.

Question 13

Why do you think generational systems are built with the schemes this way round?

Question 14

Describe briefly how you would implement such a generational system. Pay special attention to how you would handle pointers from nodes in the old store to nodes in the new store. (You should be aware that the number of such pointers is normally very small in practice.) Would it help if the user called a special function to warn you whenever such a pointer was created?