Algorithms

Exercise 4: Data structures II

Data structures are often explained in an imperative setting. In this exercise, we will look at binomial heaps and red-black trees in a purely functional setting[1]. For this exercise, we will use a pure subset of ML[2]. In other words, we are not allowed to use references/mutable values.

1 Purely functional binomial heaps

1. Binomial heaps are composed of more primitive objects known as binomial trees. Binomial trees are inductively defined as:

   - A binomial tree of rank 0 is a singleton node.
   - A binomial tree of rank $r + 1$ is formed by *linking* two binomial trees of rank r, marking one tree the leftmost child of the other.

   (a) How many nodes does a binomial tree of rank r contain? (1 mark)

   (b) We represent a node in a binomial tree using the following data type:

   datatype $tree = Node$ **of** $int \times int \times tree\ list$

   Each *Node* represents a rank (of type *int*), an element (of type *int*), and a list of children (of type *tree list*). The list of children is maintained in decreasing order of rank, and elements are stored in heap order.

   Given two nodes, explain how you would link them so that heap order is maintained. (2 marks)

   (c) Define a function $link : tree \to tree \to tree$ based on your answer for the previous question. (2 marks)

   (d) A binomial heap is a collection of heap-ordered binomial trees in which no two trees have the same rank. This collection is represented as a list of trees in increasing order of rank:

   type $heap = tree\ list$

   Suppose that you have a binomial heap of size n. How many trees will this heap contain at most? (1 mark)

   (e) Define a function $rank : tree \to int$ which returns the rank of a tree and a function $root : tree \to int$ which returns the value of a tree. (2 marks)

   (f) We wish to define a recursive function whose typing is $insTree : tree \times heap \to heap$. It should insert a binomial tree (the first argument) into a heap (the second argument) and return the updated heap. Describe an algorithm for this task. (3 marks)

   (g) Implement the *insTree* function. (3 marks)

   (h) In order to insert an element (a value of type *int*) into a heap, we need to define a $insert : int \times heap \to heap$ function. Explain how this can be done with the help of *insTree* and implement your answer. (2 marks)

[1] This exercise sheet is largely based on *Purely Functional Data Structures* by Chris Okasaki.

[2] You may use other languages, as long as you impose the same restrictions there.

(i) What is the worst-case time complexity of *insert* for a binomial heap of size n? Justify your answer. (2 marks)

(j) In order to define functions which find or remove the minimum element from the binomial heap, we require two auxiliary functions. The first of these helper function is *removeMinTree* : $heap \to tree \times heap$. In other words, *removeMinTree* is a function which, given a binomial heap, returns the binomial tree which contains the minimum root element and an updated heap in which the tree containing the minimum root element has been removed. Describe how you would implement such a function. (2 marks)

(k) Implement the recursive *removeMinTree* function. (2 marks)

(l) Using *removeMinTree*, we can now define a *findMin* : $heap \to int$ function which returns the smallest root element in the heap. Implement this function. (2 marks)

(m) Implementing a *deleteMin* : $heap \to heap$ function which removes only the smallest element from the heap and returns the updated heap is a little trickier. With reference to *removeMinTree*, briefly explain how you would go about implementing such a function and where the trickiness comes from. (3 marks)

(n) (CSTs only) Define a *merge* : $heap \times heap \to heap$ function which merges two binomial heaps. (5 marks)

(o) (CSTs only) Implement *deleteMin* using *merge*. (2 marks)

(p) The worst-case time complexity of *findMin* can be improved to $\mathcal{O}(1)$ by modifying the *heap* type. Discuss how this could be done. (4 marks)

(q) (CSTs only) Based on your answer for the previous question, re-implement binomial heaps so that finding the minimum element takes $\mathcal{O}(1)$ time. (8 marks)

2 Purely functional red-black trees

2. We define the following two types for red-black trees:

$$\begin{array}{lcl} \textbf{datatype } colour & = & R \mid B \\ \textbf{datatype } tree & = & E \mid T \textbf{ of } colour \times tree \times int \times tree \end{array}$$

A tree is either empty (the E constructor) or a node (the T constructor) consisting of a colour, a left sub-tree, a value, and a right sub-tree. We insist that every red-black tree satisfies the following two invariants:

1. No red node has a red child.
2. Every path from the root to an empty node contains the same number of black nodes.

The algorithm for looking up elements in a red-black tree is the same as for any other binary search tree.

(a) Suppose that you have a *balance* : $colour \times tree \times int \times tree \to tree$ function which, given the values of a node in the tree, balances the node's sub-trees (note: *balance* assumes that the sub-trees are already mostly sorted and it will not traverse them and runs in $\mathcal{O}(1)$ time – it will only ensure that the immediate children of the given node do not violate any of the invariants). Using *balance*, discuss how you would implement a *insert* : $tree \times int \to tree$ function which inserts an element of type *int* into a red-black tree. (3 marks)

(b) Implementing *balance* (typing is given above) in a functional language is surprisingly easy. There are only *four* cases in which the tree needs to be balanced. What are they? If you can think of more than four, try the next few questions first and then see if some cases are redundant. (4 marks)

(c) It is easy to test for the four cases using pattern matching. Implement the *balance* function just by pattern matching on the arguments. (6 marks)

(d) Implement *insert*. (3 marks)

(e) What are the worst-case time and space complexities of *insert*? Explain your answer. (6 marks)

(f) Explain how you would write a function *fromOrdList* : *int list* $\rightarrow$ *tree* that converts a sorted list with no duplicates into a red-black tree. Your function should run in $\mathcal{O}(n)$ time. Also, you do not have an infinite amount of memory. (8 marks)

(g) Your *balance* function is likely going to perform some unnecessary tests in some cases. Suggest how you could split *balance* into several, smaller functions and how you would modify *insert* so that no unnecessary tests are performed. (8 marks)