Algorithms

Exercise 3: Data structures I

1 Binary trees

1. In OOP ex1, we implemented *unbalanced* binary trees. Let's talk about them.
   (a) Suppose that we wish to insert natural numbers from a list into an (unbalanced) binary tree, one after another. Give an example of a list which would result in a completely unbalanced tree. (1 mark)
   (b) What is the worst-case time complexity of inserting an item into an unbalanced binary tree and why? (2 marks)
   (c) What is the worst-case time complexity of looking up an item in an unbalanced binary tree and why? (2 marks)
   (d) A space-efficient way of implementing a binary tree is using an array. Each node in the tree is represented by a position in the array. Given the index of a node within the array, it is possible to calculate the index of its parent, left child, and right child. The skeleton for such an implementation in Java is given below:

```
class BinaryTree {
  private int[] values;

  public BinaryTree(int n) {
    values = new int[n];
  }
  private int getLeft(int index) {  /* TODO */  }
  private int getRight(int index) {  /* TODO */  }
  public void insert(int value) {  /* TODO */  }
  public boolean contains(int value) {  /* TODO */  }
}
```

   i. Complete the definitions of the *getLeft* and *getRight* methods so that they return the index of the left or right child for a given node. Suppose that the index of the given node is i (named *index* in the skeleton code), then the left child's position is $2 * i + 1$ and the right child's position is $2 * i + 2$. (2 marks)
   ii. With reference to the *getLeft* and *getRight* methods, explain how you would implement the *insert* and *contains* methods. (4 marks)
   iii. (CSTs only) Implement the *insert* and *contains* methods. (4 marks)
   iv. What is the worst-case space complexity of *insert*? (1 mark)
   v. Currently, this implementation can only contain up to n-many nodes (including the root). Suggest one way to remove this limit. The tree should still be represented using an array. (2 marks)
   vi. (CSTs only) Implement what you suggested in your answer to the previous question. (4 marks)
   vii. Discuss what impact your (suggested) changes have on the time complexity of *insert*. (6 marks)
   viii. Describe how you would modify this implementation of binary trees to make it self-balancing. (6 marks)

2 Bloom filters

2. A Bloom filter is a probabilistic data structure that can be used to represent a set. It consists of an array of m-many bits and it uses k-many hash functions, each of which can map set elements to positions in the array. Primarily, there are two operations which are supported by a Bloom filter: insertions and lookups. In order to insert an element into the Bloom filter, it needs to be run through each of the k-many hash functions, resulting in k-many positions. All these positions in the array will then be set to 1. In order to query whether an element is in the Bloom filter, you also need to run it through each of the k-many hash functions, resulting in k-many positions. If the bits at all of those positions are 1, then the element *may* be in the bloom filter. If one or more of the bits are 0, then the element *definitely* isn't in the set.

   (a) Sketch an ADT based on the specification above. (3 marks)

   (b) Suppose that $k = 1$. What are the disadvantages of a bloom filter when compared with a hashmap? (2 marks)

   (c) Suggest *three* hash functions. (3 marks)

   (d) What are the worst-case time complexities of inserting an element and looking up an element? (2 marks)

   (e) In a language of your choice, implement a data structure or class for bloom filters. For the moment, this should simply consist of a bit array. It should also be possible to specify the size of the bit array. (2 marks)

   (f) (CSTs only) Extend your data structure so that, in addition to the bit array, it can store a list of hash functions. (2 marks)

   (g) Implement an *insert* function (see the specification above). NSTs only: you should use the three hash functions from your answer for part (c). CSTs only: you should run the element that is to be inserted into the set through each of the hash functions in the list. (2 marks)

   (h) Implement a *query* function (see the specification above). NSTs only: you should use the three hash functions from your answer for part (c). CSTs only: you should run the element that is being looked up through each of the hash functions in the list. (2 marks)

   (i) (CSTs only, advanced) Modify your insertion and lookup functions so that all hash functions are run in parallel. Discuss whether it makes sense to do this. Provide evidence for your arguments in the form of benchmarks. (8 marks)

3 The concatenate vanishes

3. A well-known problem with linked lists is that many operations require $\mathcal{O}(n)$ time. In many languages, we can improve the time complexity by making the list doubly-linked or adding a pointer to the last element. In a purely functional programming language, such as DNH, these workarounds are not possible. In this section, we will *calculate* more efficient functions. Consider the following definition of a function in DNH which concatenates two lists:

$$\begin{array}{lll} \textbf{fun} \;\; \mathit{concat}\ [\,]\ \mathit{ys} & = & \mathit{ys} \\ \quad | \;\; \mathit{concat}\ (x :: \mathit{xs})\ \mathit{ys} & = & x :: (\mathit{concat}\ \mathit{xs}\ \mathit{ys}) \end{array}$$

   (a) What is the time complexity of *concat*? (1 mark)

(b) A property of concatenation is that it is associative. In other words, the following equation should hold for three lists xs, ys, and zs of the same type:

$$\forall xs : \alpha\ list, \forall ys : \alpha\ list, \forall zs : \alpha\ list, concat\ xs\ (concat\ ys\ zs) = concat\ (concat\ xs\ ys)\ zs \quad (1)$$

i. Prove that $concat\ [\,]\ (concat\ ys\ zs) = concat\ (concat\ [\,]\ ys)\ zs$. (1 mark)

ii. Assuming that $concat\ xs\ (concat\ ys\ zs) = concat\ (concat\ xs\ ys)\ zs$ holds, prove that $concat\ (x :: xs)\ (concat\ ys\ zs) = concat\ (concat\ (x :: xs)\ ys)\ zs$. (3 marks)

(c) Below is a data type for Hutton's razor in DNH:

$$\textbf{datatype}\ expr \quad = \quad VAL\ \textbf{of}\ int \mid PLUS\ \textbf{of}\ expr \times expr$$

Remember that we view expressions as trees. We now wish to calculate the size of an expression. A *VAL* expression has size 1. A *PLUS* expression's size is the size of the left sub-expression plus the size of the right sub-expression plus 1. Complete the following, recursive definition:

$$\begin{array}{lll} \textbf{fun} & size\ (VAL(v)) & = \quad ??? \\ \mid & size\ (PLUS(l,r)) & = \quad ??? \end{array}$$

(2 marks)

(d) What is an invariant of *size e* for all expressions *e*? *I.e.* what is a property of the result of *size e*? (1 mark)

(e) Below is the compilation function for Hutton's razor in DNH:

$$\textbf{datatype}\ instr \quad = \quad PUSH\ \textbf{of}\ int \mid ADD$$

$$\begin{array}{lll} \textbf{fun} & comp\ (VAL(v)) & = \quad PUSH\ v \\ \mid & comp\ (PLUS(l,r)) & = \quad concat\ (concat\ (comp\ r)\ (comp\ l))\ [ADD] \end{array}$$

What is the worst-case time complexity of *comp e* in terms of $\mathcal{O}(g(n))$ for some function $g(n)$ where *e* is of type *expr* and $n = size\ e$? Explain how you have derived your answer. (5 marks)

(f) (CSTs only) *comp* is currently very inefficient. Luckily, since we are working in a purely functional language, we can calculate a faster function. The trick is to begin with a more general version of *comp* which we will call $comp'$. We do not know what the definition of $comp'$ is going to look like, but we want the following equation to hold:

$$\forall e : expr, \forall xs : instr\ list, \qquad comp'\ e\ xs = concat\ (comp\ e)\ xs \quad (2)$$

If we can find a way to define $comp'$, then *comp* can later be defined as:

$$\textbf{fun}\quad comp\ e \quad = \quad comp'\ e\ [\,]$$

So how do we define $comp'$? We could try to define the function by hand so that it satisfies Equation 2, but a better way is to calculate the definition using induction. To demonstrate how this works, consider the following definition of *reverse*:

$$\begin{array}{lll} \textbf{fun} & reverse\ [\,] & = \quad [\,] \\ \mid & reverse\ (x :: xs) & = \quad concat\ (reverse\ xs)\ [x] \end{array}$$

This function runs in $\mathcal{O}(n^2)$ time which is clearly not great for such a simple task. To improve it, we will say that we want to define a $reverse'$ function so that the following equation holds:

$$\forall xs : \alpha\ list, \forall ys : \alpha\ list, \qquad reverse'\ xs\ ys = concat\ (reverse\ xs)\ ys \quad (3)$$

To construct the definition of *reverse′*, we simultaneously assume that Equation 3 holds and perform induction on *xs* to verify that. Our goal is to simplify the expression as much as possible. The base case of the induction will result in the base case of the definition of *reverse′* and the recursive case of the induction will yield the recursive case of the definition of *reverse′*. The base case for induction on lists is the empty list:

$$
\begin{aligned}
& \mathit{reverse}'\ [\,]\ \mathit{ys} \\
= \quad & \{ \quad \text{Equation 3} \quad \} \\
& \mathit{concat}\ (\mathit{reverse}\ [\,])\ \mathit{ys} \\
= \quad & \{ \quad \text{applying } \mathit{reverse} \quad \} \\
& \mathit{concat}\ [\,]\ \mathit{ys} \\
= \quad & \{ \quad \text{applying } \mathit{concat} \quad \} \\
& \mathit{ys}
\end{aligned}
$$

We cannot evaluate *ys* any further, so that we will use it as the base case for *reverse′*:

$$
\begin{array}{lll}
\textbf{fun} \quad \mathit{reverse}'\ [\,]\ \mathit{ys} & = & \mathit{ys} \\
\quad \mid \quad \mathit{reverse}'\ (x :: \mathit{xs})\ \mathit{ys} & = & ???
\end{array}
$$

Next we assume that $\forall \mathit{ys} : \alpha\ \mathit{list}, \mathit{reverse}'\ \mathit{xs}\ \mathit{ys} = \mathit{concat}\ (\mathit{reverse}\ \mathit{xs})\ \mathit{ys}$ holds (the induction hypothesis) in addition to Equation 3 to prove the inductive case:

$$
\begin{aligned}
& \mathit{reverse}'\ (x :: \mathit{xs})\ \mathit{ys} \\
= \quad & \{ \quad \text{Equation 3} \quad \} \\
& \mathit{concat}\ (\mathit{reverse}\ (x :: \mathit{xs}))\ \mathit{ys} \\
= \quad & \{ \quad \text{applying } \mathit{reverse} \quad \} \\
& \mathit{concat}\ (\mathit{concat}\ (\mathit{reverse}\ \mathit{xs})\ [x])\ \mathit{ys} \\
= \quad & \{ \quad \text{associativity of } \mathit{concat} \quad \} \\
& \mathit{concat}\ (\mathit{reverse}\ \mathit{xs})\ (\mathit{concat}\ [x]\ \mathit{ys}) \\
= \quad & \{ \quad \text{induction hypothesis} \quad \} \\
& \mathit{reverse}'\ \mathit{xs}\ (\mathit{concat}\ [x]\ \mathit{ys}) \\
= \quad & \{ \quad \text{applying } \mathit{concat} \quad \} \\
& \mathit{reverse}'\ \mathit{xs}\ (x :: \mathit{ys})
\end{aligned}
$$

We can now complete the definition of *reverse′*:

$$
\begin{array}{lll}
\textbf{fun} \quad \mathit{reverse}'\ [\,]\ \mathit{ys} & = & \mathit{ys} \\
\quad \mid \quad \mathit{reverse}'\ (x :: \mathit{xs})\ \mathit{ys} & = & \mathit{reverse}'\ \mathit{xs}\ (x :: \mathit{ys})
\end{array}
$$

This, in turn, allows us to give a new definition for *reverse*:

$$
\textbf{fun} \quad \mathit{reverse}\ \mathit{xs} \quad = \quad \mathit{reverse}'\ \mathit{xs}\ [\,]
$$

i. What is the worst-case time complexity of the new *reverse* function? (1 mark)

ii. Calculate the base case of *comp′* by evaluating $\mathit{comp}'\ \mathit{VAL}(v)\ \mathit{xs}$ under the assumption that Equation 2 holds. (2 marks)

iii. Calculate the recursive case of $comp'$ by evaluating $comp'\ PLUS(l,r)\ xs$ under the assumption that Equation 2 and the induction hypothesis $\forall xs : instr\ list, comp'\ e\ xs = concat\ (comp\ e)\ xs$ hold. (5 marks)

iv. What is the worst-case time complexity of $comp'\ e\ [\,]$ in terms of $\mathcal{O}(g(n))$ for some function $g(n)$ where e is of type *expr* and $n = size\ e$? Explain how you have derived your answer. (2 marks)

4. Download the model solutions for OOP ex1[1] and implement a *reverse* method in the *LinkedList* class. (2 marks)

5. (CSTs only) Compare the time complexity of that method with the original and revised definitions of *reverse* in DNH. (2 marks)

[1] `http://www.cl.cam.ac.uk/~mbg28/oop-ex1.zip`