2013 Haskell January Test
Binomial Heaps

This test comprises three parts and the maximum mark is 25. Parts I and II are worth 22 of the 25 marks available. The **2013 Haskell Programming Prize** will be awarded for the most elegant solution overall.

Credit will be awarded throughout for clarity, conciseness, *useful* commenting and the appropriate use of Haskell's various language features and predefined functions.

WARNING: The examples and test cases here are not guaranteed to exercise all aspects of your code. You may therefore wish to define your own tests to complement the ones provided.

1 Introduction

A *binomial heap* is an elegant data structure for supporting ordered collections of values efficiently. In this exercise you're going to implement, along with some other functions, three of the operations usually associated with a binomial heap, namely the insertion of new values, the extraction of the minimum value and the deletion of the minimum value. The crucial property of a binomial heap is that operations such as these execute in a time that is proportional to the *logarithm* of the size of the collection *in the worst case*. It thus avoids the problems associated with some other branching data structures, e.g. binary trees, where an imbalance in the structure can cause it to behave more like a linked list. When this happens operations like the above can end up scaling *linearly* with the size of the collection, in the worst case, instead of logarithmically.

The ordering on values will be determined by the functions/operators of Haskell's `Ord` class (`<`, `<=`, `>`, etc.), i.e. the types of the values we can place in a binomial heap will be required to be instances of the `Ord` class.

2 Binomial Trees

A binomial heap will be implemented as a list of *binomial trees* – see below for the details – where a binomial tree is defined recursively as follows:

1. A binomial tree of *rank 0* is a single node comprising an item of some type `a`. In the examples used here, `a` will be `Int`, but in general it can be any type that is a member of `Ord`.

2. A binomial tree of *rank r* comprises a root node containing an item of type `a` and a list of subtrees, which are binomial trees of rank $r-1$, $r-2$, ..., 1, 0, *in that order*. For the purposes of this exercise each binomial tree will satisfy the *minimal-heap property*, which means that the value stored at a root of a tree is smaller than or equal to all the values stored within its subtrees. We'll call these *ordered* trees. From hereon all trees will be assumed to be ordered, so we will occasionally omit the qualifier 'ordered' when the meaning is clear.

Figure 1 shows examples of ordered trees of `Int`s of rank 0, 1, 2 and 3. Note that the `Int` values are in ascending order – ascending *numerical* order in the case of `Int`s – on each path from the root of each tree to its leaves – this is the minimal-heap property.

Notice that a binomial tree of rank r has exactly 2^r nodes and that the number of nodes at depth d, $0 \leq d \leq r$, is given by

$$\left(\begin{array}{c} r \\ d \end{array} \right) = \frac{r!}{d!\,(r-d)!}$$

hence the name "binomial tree". Note also that the root of a tree is at depth 0.

In this exercise, binomial trees will be represented by the following Haskell data type:

```
data BinTree a = Node a Int [BinTree a]
               deriving (Eq, Ord, Show)
```

The `BinTree` type has a single node constructor, `Node` with three arguments: the value stored at the corresponding node (of type `a`), the rank of the tree rooted at the node (of type `Int`) and a list of children, which are subtrees of type `[BinTree a]`. Recall that each child is itself a binomial tree. For example, the rank-2 tree in Figure 1 will be represented as follows:

```
Node 2 2 [Node 8 1 [Node 9 0 []], Node 7 0 []]
```

The representations of all four trees shown in Figure 1 are defined in the template as `t1`–`t4`.

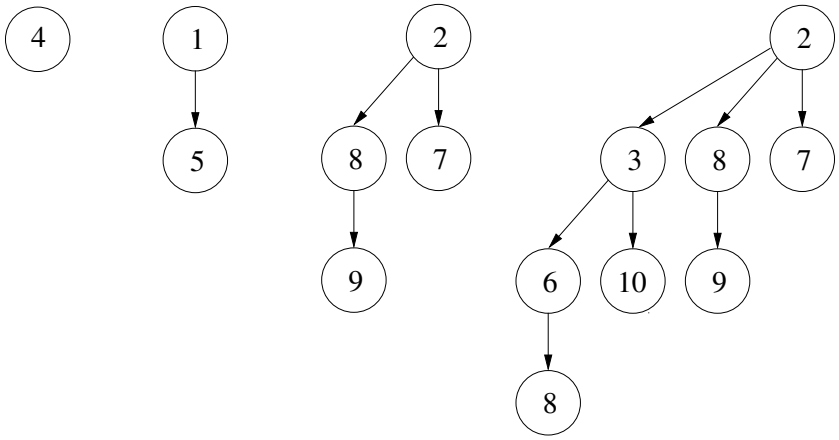


Figure 1: Examples of ordered binomial trees of rank 0, 1, 2 and 3.

2.1 Combining trees

The key operation on binomial trees is that which *combines* two ordered trees *of the same rank*, r say, into a single ordered tree of rank $r+1$. The rank is $r+1$ because the combined tree contains twice as many nodes as the original trees.

Combining works by first determining the tree, t say, with the minimum root value. Clearly, the root of this tree must become the root of the combined tree in order that the combined tree is itself ordered. The combine operation is completed by adding the other tree as an additional child of t. Note that it must be added to the *front* of the list of t's subtrees in order to satisfy the second recursive property above: because the combined tree has rank $r+1$ the first child of the root node of the combined tree must be a tree of rank r.

Note that binomial trees can contain several instances of the same value (they are not sets). If the root values are the same when combining trees then we'll make the arbitrary decision to proceed as if the *second* tree contained the minimum root value, i.e., the first tree should end up as the first child of the second tree.

Figure 2 shows an example of two trees of rank 2 being combined to form a tree of rank 3.

3 Binomial Heaps

A binomial heap is a collection of binomial trees, stored in increasing order of their rank. Importantly, no two trees in the heap may have the same rank. Thus, for a particular rank $r \geq 0$, a heap will contain either zero or one occurrences of a binomial tree of rank r.

In Haskell, we can represent heaps as *lists* of binary trees, although we'll have to maintain the rank order ourselves. Figure 3 shows an example of a binomial heap comprising trees of rank 0, 1 and 3. Here, the tree structures are shown with bold lines, as in the examples above, whilst the elements of the Haskell list making up a heap are shown linked by dashed lines. For example, the heap shown corresponds to the three-element Haskell list `[t1, t2, t4]` where `t1`, `t2` and `t4` are sample binomial trees defined in the template. For convenience the type synonym `BinHeap` has been defined in the template as a list of `BinTrees`:

```
type BinHeap a = [BinTree a]
```

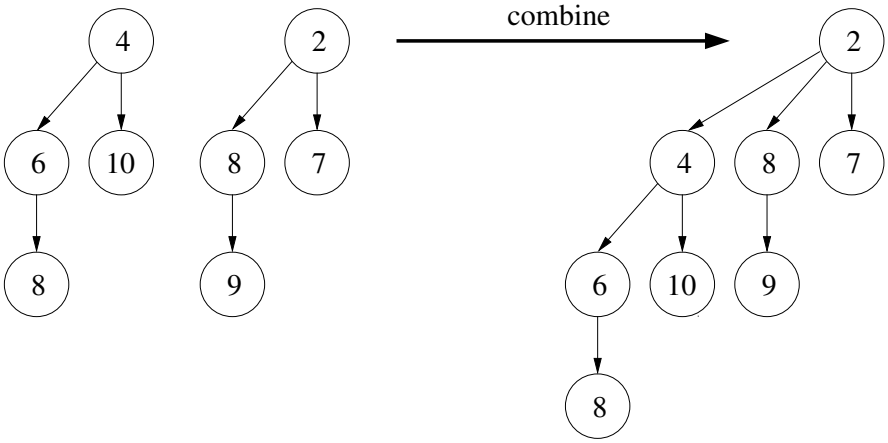


Figure 2: Combining two ordered trees of rank 2, making a tree of rank 3.

Binomial heaps satisfy the following properties:

- The minimum value in the heap is contained in one of the root nodes of the list of trees. In Figure 3, for example, the minimum value is 1.
- If the total number of nodes is N then the heap contains at most $\lfloor log_2(N) \rfloor + 1$ trees. Thus, the minimum value in the heap can be found in at most $\lfloor log_2(N) \rfloor + 1$ steps.
- If the heap contains trees of rank $r_1, r_2, \ldots, r_k$ for some $k \geq 0$, then the number of nodes in the tree is:

$$\sum_{i=0}^{k} 2^{r_i}$$

  For example, the heap in Figure 3 contains trees of rank 0, 1 and 3 so the total number of nodes is $2^0 + 2^1 + 2^3 = 11$.

3.1 Merging Heaps

The key operation on binomial heaps is *merging*, which merges two binomial heaps into one. Each node in the two original heaps occurs somewhere in the merged heap and the merged heap itself satisfies all the properties of a binomial heap. In particular, the minimum element in the merged heap is the smallest of the minimum elements in the original heaps and thus appears in the root node of one of the trees making up the merged heap. As an example, Figure 4 shows two heaps and the result of merging them together.

The merging of two heaps `h` and `h'` works recursively according to the following rules:

- If either heap is empty, i.e. `[]`, then the result is the other heap.
- If the two heaps respectively contain trees `t` and `t'` at the heads of the corresponding lists, then the merge proceeds on the basis of the ranks of those trees:
  - If the rank of `t` is less than that of `t'` then `t` forms the head of the resulting heap and the remainder of the heap is formed by recursively merging the remaining elements of `h` (i.e. the tail of `h`) with `h'`.

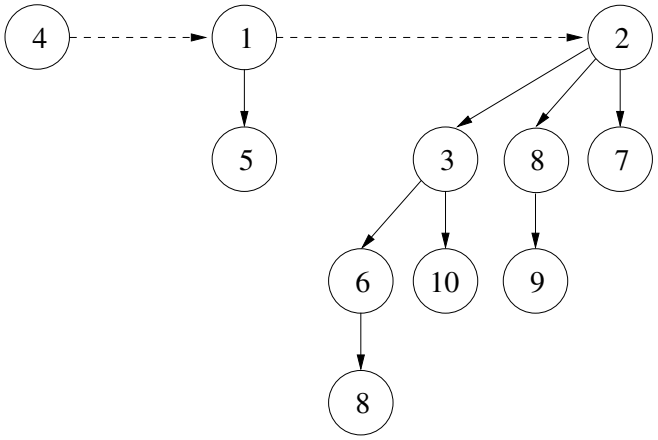
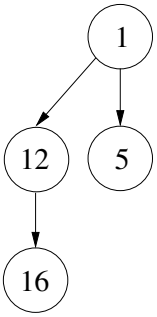


Figure 3: A binomial heap corresponding to `h3 = [t1, t2, t4]`, as given in the template file.

- The symmetric rule applies if the rank of `t'` is less than that of `t`.
- If the two ranks are the same, then `t` and `t'` are *combined*, as decribed above, and the resulting combined tree is *added* to the result of recursively merging the remaining elements of `h` and the remaining elements of `h'`, i.e. the tails of both lists. The simplest way to add the combined tree is to form a singleton heap, i.e. a singleton list, from it and then merge this with the heap that is returned from the recursive call.

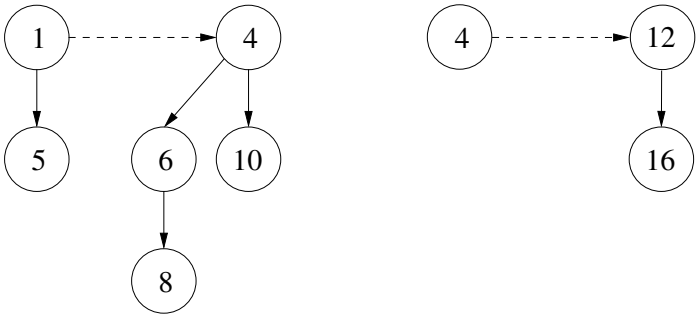
Referring back to Figure 4, the tree with root value 4 appears first in the merged heap because it has the smallest rank (0). The rest of the heap is formed by merging the original heap on the left of Figure 4(a) with the tail of the heap on the right, i.e. the heap comprising the single tree containing values 12 and 16.

At this point the heads of both heaps contain trees of rank 1 and these are therefore combined to form the rank-2 tree below:

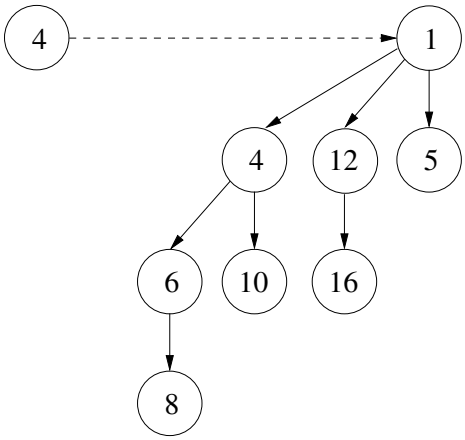


which we'll call `t''`. This is then added to the heap, `h''` say, that results from recursively merging the tails of the two heaps: the first tail comprises the singleton tree with values 4, 6, 8 and 10 and the second is the empty heap (`[]`); this invokes one of the base cases and `h''` is thus the heap comprising values 4, 6, 8 and 10 in a single rank-2 tree, t''', say.

The addition of `t''` to `h''` is implemented by merging the singleton heap `[t'']` with `h''`. Applying the merging rules again in turn, `t''` is then combined with `t'''` (they are both of rank 2) forming a rank-3 tree comprising values 1, 4, 6, 8, 10, 12, 16 and 5. This is then added to the



a. Two heaps



b. The merged heap

Figure 4: Merging two heaps.

result of recursively merging the two heap tails which, in this case, are both empty. We thus end up with the heap shown in Figure 4(b).

3.2 New value insertion

Once the merging function has been defined it is easy to implement a function for inserting a new value into a heap: simply form a singleton heap comprising a singleton tree of rank 0, made using the given value, and merge this with the given heap.

3.3 Exracting and deleting minimal values

Extraction of the mimimum value in a heap simply involves inspecting the value of the root nodes of the trees representing the heap and returning the smallest. For example, the minimum value in the heap shown in Figure 3 is 1.

Deleting the root node with the smallest value is a little more involved. The trick is to observe

that the children of the root node form a list of binomial trees that is itself a perfectly valid binomial *heap*, but with the elements in the *reverse* order. For example, the minimum value of the merged heap of Figure 4(b) is 1, as we have just seen. The children of the corresponding node are represented by a list of binomial trees of rank 2, 1 and 0 respectively. If you reverse this, you then have a list of trees with rank 0, 1 and 2 – a valid binomial heap, `h'`, say. So, to effect the deletion you simply remove the tree rooted at 1 from the original heap, in this example leaving the singleton heap comprising the rank-0 tree with root value 4, and then merge this with `h'`. The result is shown in Figure 5.

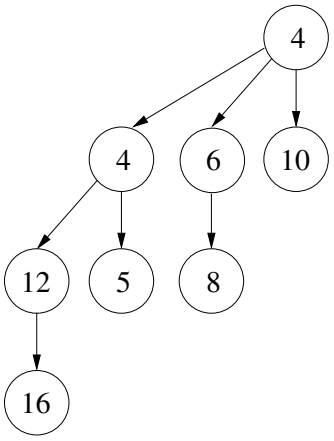


Figure 5: The result of deleting the minimum value from Figure 4(b).

Note that there may be several root nodes with the same minimal value, in which case the *leftmost* one, i.e. the one with the lowest rank, is arbitrarily chosen for deletion.

4 What To Do

Parts I and II require you to implement the various functions described in the text above and one additional one. Part III contains a separate, rather fiddly, problem that you are required to work out largely for yourselves. This carries only 3 of the 25 marks, so you are advised to complete Parts I and II before attempting Part III.

A number of test trees and heaps are included in the template for testing purposes; most of these represent the trees and heaps in the above figures. These are labelled `t1`, ..., `t8` and `h1`, ..., `h7` respectively.

4.1 Part I – Binomial Trees

1. Define three functions: `value :: BinTree a -> a`, `rank :: BinTree a -> Int` and `children :: BinTree a -> [BinTree a]` that respectively return the value, rank and children of the root node of a given binomial tree. For example, `value t4` should return 2, `rank t7` should return 3 and `children t2` should return `[Node 5 0 []]`. You may wish to use these functions in the questions that follow.

[1 mark]

2. Define a function `combineTrees :: Ord a => BinTree a -> BinTree a -> BinTree a` that will combine two binomial trees as described in Section 2.1 above. For example, the application `combineTrees t5 t6` should yield `t7`, as per Figure 2.

[3 marks]

4.2 Part II – Binomial Heaps

1. Define a function `extractMin :: Ord a => BinHeap a -> a` that will return the minimum value stored in a given binomial heap. A precondition is that the heap is non-empty. For example, `extractMin h3` (the heap shown in Figure 3) should return 1.

[2 marks]

2. Define a function `mergeHeaps :: Ord a => BinHeap a -> BinHeap a -> BinHeap a` that will merge two heaps as described in Section 3.1. For example, `mergeHeaps h4 h5` should return `h6`.

[6 marks]

3. Define a function `insert :: Ord a => a -> BinHeap a -> BinHeap a` that will insert a given value into a given heap, as outlined in Section 3.2. For example, `insert 7 [Node 4 0 []]`, equivalent to `insert 7 [t1]`, should yield `[Node 4 1 [Node 7 0 []]]`.

[1 mark]

4. Define a function `deleteMin :: Ord a => BinHeap a -> BinHeap a` that removes the minimum value from a give heap, as described in Section 3.3. For example, `deleteMin h6` should generate the heap shown in Figure 5, defined as `h7` in the template.

[5 marks]

5. Define a "binomial sorting" function `binSort :: Ord a => [a] -> [a]` that sorts the elements of a given list of items in ascending order, using a binomial heap as an intermediate data structure. You should do this by inserting the list items one by one into an initially empty heap. You can extract the sorted list by repeatedly extracting, and then deleting, the minimum element, until the heap becomes empty. For example, `binSort "BinomialHeap"` should return `"BHaaeiilmnop"`.

[4 marks]

4.3 Part III – Binary Heap Counting

Recall from the above that a binomial heap contains at most one binomial tree of a given rank and that a tree of rank r contains 2^r nodes, $r \geq 0$. The number of nodes in the heap can thus be represented in binary as $b_R, b_{R-1}, \ldots, b_1, b_0$, where $b_i, 0 \leq i \leq R$, is 1 if the heap contains a tree of rank i and 0 if not, and where R is the rank of the largest tree in the heap. Assuming there are no leading zeros, we can assume that b_R=1.

1. Define a function `toBinary :: BinHeap a -> [Int]` that generates the binary representation of the number of nodes in a given heap. For example, `toBinary h2` should return `[1,1,0,0]`, the binary representation of 12. Try to avoid using `^`, `div` and `mod`.

2. In order to test whether, for example, the number of elements in a merged heap is the same as the sum of the number of elements in the two original heaps you could work with integers, or you could instead write something like:

```
toBinary (mergeHeaps h h') == binarySum (toBinary h) (toBinary h')
```

where `h` and `h'` are the original heaps. Define the `binarySum` function that sums directly the binary representations of two non-negative integers using *ripple carry addition*, i.e. by recursively traversing the representations of the two numbers from right to left (least significant to most significant bit), applying a *full adder* at each bit position. A full adder takes the two bits that are to be summed and the carry bit from the right and generates a sum bit and a carry bit according the obvious rules:

Inputs			Outputs	
b_1	b_2	Carry in	Sum	Carry out
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

Test your function by verifying that, for example, `binarySum (toBinary h4) (toBinary h5)` generates `[1,0,0,1]`. Did you use the `reverse` function? If so, can you define the function without it?!

[3 marks for the two questions]