Transforming two heaps in linear time, a formal approach

Technical Report 03/14
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

Cristóbal Pareja, Ricardo Peña, Clara Segura
Departamento de Sistemas Informáticos y Computación,
Facultad de Informática, Universidad Complutense de Madrid,
cpareja@sip.ucm.es, ricardo@sip.ucm.es, csegura@sip.ucm.es

Abstract: Heaps are efficient data structures implementing the abstract data type *priority queue*. William's original heaps were implemented by using arrays. In a functional programming setting it is however more common to implement them as binary trees. But then the heap cardinal is needed in order to find the structural position occupied by the element being inserted or deleted. Based on Braun's flexible arrays, Paulson introduced a variation of William's heaps in which the cardinal of the heap was not needed.

In this paper we formally deduce the properties of both kinds of heaps, using as a starting point two simple invariants characterising them. We show that an isomorphism can be defined by mapping structural positions of one of them into structural positions of the other. Then, we explicitly define such an isomorphism by means of a linear time operation *shuffle* which converts William's heaps into Paulson's heaps and the other way around, and which is its own inverse. It can be the basis for building an interface between imperative programs and functional ones at low cost.

Key Words: heap, structural isomorphism

1 Introduction

Heaps are the most common implementations of the abstract data type *priority queue*. The first well-known implementation is the one proposed by J.W.J. Williams in 1964, in which the elements are kept in a binary tree represented by an array, and which is the basis of the heapsort algorithm. Since then, many other people have proposed alternative implementations. For the purpose of this paper, we may classify them in the two following groups:

– *The cardinal operation is injective* with respect to the structure, i.e. there are no two different structures with the same cardinality. Disregarding the value of the concrete elements, William's heaps satisfy this property: given a natural number n, there is only one possible heap structure with this cardinality, namely that one where the array is filled up to the n-th position. Binomial heaps [Cormen et al., 2001, Chapter 19] is another example, in which the bits of n determine which binomial trees are present in the heap.

– *The cardinal operation is not injective.* To this category belong the leftist trees heaps [Horowitz et al., 1995, Chapter 9], and the Fibonacci heaps [Cormen et al., 2001, Chapter 20] among others.

Another useful classification has to do with the structural stability of the insertion operation:

– *Stable insertion heaps* are those where inserting a new element preserves the previous structure and only a new structural position is created. William's heaps satify this criterium but binomial heaps do not: when a new element is added, some subtrees may disappear in the structure and a new subtree may appear. So, many structural positions may change.

– *Non-stable insertion heaps* are most of the other known heaps: binomial, leftist trees and Fibonacci ones.

Paulson proposes in [Paulson, 1996] another heap based on binary trees in which the elements are inserted by following the numbering scheme of Braun trees [Hoogerwoord, 1993]. These were originally proposed as a functional implementation of flexible arrays, in which logarithmic time is needed in order to lookup or update an element, and where elements can be added or deleted at both ends.

Paulson shows that in a functional setting the new heaps —he calls them just 'binary heaps'— are more adequate than Williams ones. The latter, when programmed in a functional language, are binary trees having the property that the root value is the minimum of all the elements, and this property is recursively satisfied by the subtrees. If one wishes to insert a new element, there is no clue on what insertion path to follow, unless the cardinality of the heap is known beforehand. Using the bits of this cardinality, it is possible to locate the new structural position. The same information is needed when deleting the minimum element and reestructuring the tree. In Paulson's heaps, insertion and deletion can be done without needing other information than the tree itself.

In this paper we elaborate on the similarities and differences between Williams heaps (we call them *classic*) and Paulson heaps (we call them *alternate*). We show that the latter have injective cardinal and stable insertion. They enjoy other properties similar to those of Williams' such as having minimum height and all the levels complete except perhaps the last one.

As both heaps are structurally isomorphic to the set of natural numbers, they are isomorphic to each other. We wondered whether it were possible to convert one into the other. We have affirmatively answered the question by building an $O(n)$ *shuffle* operation doing exactly that, and which is its own inverse: when given a classic heap, it produces the alternate one having the same cardinality, and when given an alternate heap, it produces the structurally equivalent classic

one. In fact, we provide a set of transformations, all of them running in linear time, which convert imperative arrays, flexible arrays, classic heaps and alternate heaps to any one of the others. They can be the basis for building an interface between imperative programs and functional ones at a low cost.

The plan of the paper is the following. In Section 2 we provide the formal specifications of classic and alternate heaps by means of structural invariants. In Section 3 we prove some interesting structural properties for both kinds of heaps and conclude that their structure is uniquely determined by their cardinality. We show that bit paths can be used to locate structural positions in these heaps and define several insertion functions for them. Heap ordering is introduced in Section 4. We refine the insertion functions and define new ones for deleting the minimum by using functions to float or sink elements as necessary. Section 5 shows a explicit structural transformation function between classic and alternate heaps which is not very efficient but that is formally proved to be an isomorphism between them. An efficient version is obtained in Section 6 by using some of the structural properties already proved in Section 3. Conclusions are given in Section 7. The functions shown in this paper as well as some auxiliary code, written in Haskell, can be obtained at `http://dalila.sip.ucm.es/heaps`.

2 Classic and alternate heaps

Considering only the structure, both classic and alternate heaps are binary trees

$$\texttt{h}_{(n,k)} \texttt{ = Empty} \mid \texttt{Node h1}_{(n_1,k_1)} \texttt{ () h2}_{(n_2,k_2)}$$

satisfying different structural invariants. Tree *Empty* is both a classic and an alternate heap. In this paper, the notation $h_{(n,k)}$ means that h is a binary tree with cardinality n and height k, considering that the empty tree `Empty` satisfies $n = 0$ and $k = 0$, and the non-empty tree $\texttt{Node h1}_{(n_1,k_1)} \texttt{ x h2}_{(n_2,k_2)}$ satisfies $n = 1 + n_1 + n_2$ and $k = 1 + \max(k_1, k_2)$. A heap $h_{(n,k)}$ has k levels. In the first level the only element is the root of the tree. In level 2, there is a maximum of two elements. In general, the level k consists of the immediate children of nodes in level $k - 1$, and it may contain up to 2^{k-1} elements.

We will refer to the type of heaps as `Heap a`. As part of the paper concentrates on structural properties of heaps, we will use `Heap ()` when the elements are not important. Ordering is introduced in Section 4, where `Ord a` is required.

By *classic* heap we mean the original heap invented by J. W. J. Williams in 1964, which is in the basis of the *heapsort* sorting algorithm. It can be defined as a quasi-complete binary tree where the only possibly incomplete level is the last one, and within this level the elements are grouped to the left. However, in the original formulation, the binary tree was represented on an array so that there were no holes in the range from 1 to the number n of elements. In this section, we define a classic heap as a binary tree satisfying an invariant property.

From this invariant we show (see Section 3) that all the above properties can be deduced.

Figure 1(a) shows a classic heap with 25 elements. The numbers shown in the nodes are not necessarily the elements of the heap, but they indicate the historical order in which positions occur in the structure. These are always the same independently of the heap cardinality.

By *alternate* heap, we mean the binary tree h of cardinality n that is built by the invocation `makeAlt [1..n]`:

```
unshuffle :: [a] -> ([a], [a])
unshuffle = foldr (\ x (ys1,ys2) -> (x:ys2,ys1)) ([],[])

makeAlt :: [a] -> Heap a
makeAlt []     = Empty
makeAlt (x:xs) = Node (makeAlt xs1) x (makeAlt xs2)
                 where (xs1,xs2) = unshuffle xs
```

where function `unshuffle` splits the list argument in a way such that the odd positions are moved to the left list, and the even positions to the right one.

Fig. 1(b) shows an alternate heap with 25 elements. Again, the numbers indicate the historical order in which positions appear in the structure. Alternate heaps have minimum height and the only possibly incomplete level is the last one. We will define however an alternate heap as a binary tree with an invariant property, from which all the above properties can be proved.

Definition 1. Binary tree $h_{(n,k)} = \mathit{Node}\ h_{1(n_1,k_1)}\ ()\ h_{2(n_2,k_2)}$ is a **classic heap** iff it satisfies the following structural invariant

$$I_C \stackrel{\text{def}}{=} \left\{ \begin{array}{c} (k_1 = k_2 \wedge n_1 = 2^{k_1} - 1) \\ \vee\ (k_1 = k_2 + 1 \wedge n_2 = 2^{k_2} - 1) \end{array} \right\}$$

and also both h_1 are h_2 classic heaps.

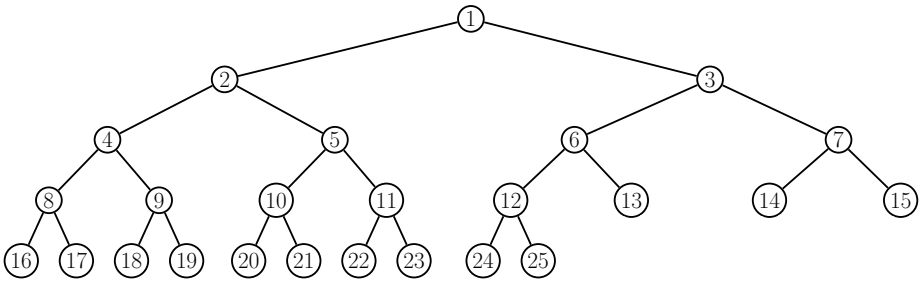
We make note that a complete tree, which satisfies $n = 2^k - 1$ in all its subtrees including itself, is a classic heap, since the invariant above trivially holds. We will prove later that n and k are not independent variables, but at this point it is useful to show both in order to define the invariant.

Definition 2. Binary tree $h_{(n,k)} = \mathit{Node}\ h_{1(n_1,k_1)}\ ()\ h_{2(n_2,k_2)}$ is an **alternate heap** iff it satisfies the following structural invariant

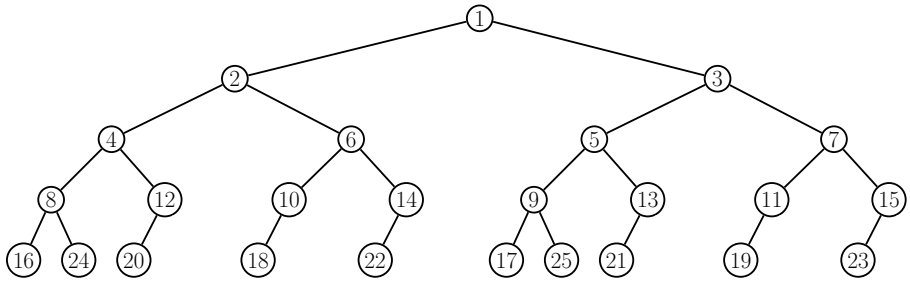
$$I_A \stackrel{\text{def}}{=} \{\, n_2 + 1 \geq n_1 \geq n_2 \,\}$$

and also both h_1 are h_2 alternate heaps.

We make note that a complete tree is an alternate heap, since the invariant above trivially holds. As in the case of classic heaps, we will prove that n and k are not independent variables.



(a)



(b)

Figure 1: A classic heap (a) and an alternate heap (b) with $n = 25$.

3 Structure and cardinality

In this section we prove some properties concerning the structure of classic and alternate heaps. We prove that cardinality uniquely determines their structure, and consequently they are isomorphic.

One of the most interesting properties in both heaps is that the historical positions of the elements in a heap are stable, i.e. they do not change from one cardinality $n-1$ to the next one n, only a new position appears in the heap with n elements. We prove that such position can be identified by using in different ways the bit path corresponding to n. On the basis of these structural properties, we define insertion functions for both kind of heaps which do not consider order.

3.1 Classic heaps

Lemma 3. *The height k of a non-empty classic heap is $k_1 + 1$, being k_1 the height of its left child.*

Proof. Trivial, because $k = 1 + max(k_1, k_2) = 1 + k_1$ by the definition of height, and by $k_1 \geq k_2$.

Lemma 4. *In a classic heap of height $k \geq 1$, its first $k-1$ levels are complete.*

Proof. By induction on k:

$k = 1$ The singleton tree $h_{(1,1)}$ trivially satisfies the property.

$k > 1$ Let us assume that every classic heap with height $k-1$ satifies the property, i. e. its $k-2$ levels are complete. We consider the two possible cases described by I_C:

- $k_1 = k_2$, $n_1 = 2^{k_1} - 1$ and h_2 is a classic heap. Because $n_1 = 2^{k_1} - 1$, the left subtree $h_{1(n_1,k_1)}$ is complete and its height is $k_1 = k - 1$. By being complete, it has $k-1$ levels complete, and in particular $k-2$. Regarding $h_{2(n_2,k_2)}$, it is a classic heap with $k_2 = k - 1$ levels. By induction hypothesis (i.h.) its first $k-2$ levels are complete.

- $k_1 = k_2 + 1$, $n_2 = 2^{k_2} - 1$ and h_1 is a classic heap. Because $n_2 = 2^{k_2} - 1$, the right subtree $h_{2(n_2,k_2)}$ is complete and its height is $k_2 = k_1 - 1 = k - 2$. By being complete, it has $k-2$ complete levels. Regarding $h_{1(n_1,k_1)}$, it is a classic heap with $k_1 = k - 1$ levels. By i.h. its first $k-2$ levels are complete.

In both cases, both subtrees have their $k-2$ first levels complete, so the tree h has $k-1$ complete levels.

Corollary 5. *A classic heap has minimal height.*

As a consequence, given the cardinality $n > 0$ of a classic heap, its height is fully determined: it is the only k satisfying $2^{k-1} \leq n < 2^k$, or equivalently $k = \lfloor \log_2 n \rfloor + 1$.

Lemma 6. *In a classic heap of height k, the elements of the k-th level are fully grouped to the left.*

Proof. By induction on k:

$k = 1$ The singleton tree $h_{(1,1)}$ satisfies the property.

$k > 1$ Let us assume that any heap of height $k-1$ satisfies the property. Consider the two possibilities described by I_C:

- $k_1 = k_2$, $n_1 = 2^{k_1} - 1$, and h_2 a classic heap. Subtree h_1 is complete with height $k-1$, so its $(k-1)$-th level is complete. Subtree h_2 is a classic heap with height $k-1$. By i.h. it satisfies the property. So, the whole tree has the elements of the k-th level fully grouped to the left.

– $k_1 = k_2 + 1$, $n_2 = 2^{k_2} - 1$, and h_1 a classic heap. Subtree h_2 is complete with $k - 2$ levels, so its $(k-1)$-th level is empty. Then, it is enough to consider the left subtree h_1. As it is a classic heap with $k - 1$ levels, by i.h. its $(k-1)$-th level is fully grouped to the left.

Lemma 7. *Given $n \in \mathbb{N}$, there exists an only structural classic heap with this cardinality.*

Proof. By induction on n:

$n = 0$ `Empty` is the only classic heap with cardinality 0.

$n > 0$ There exist $n_1, n_2 \in \mathbb{N}$ such that $n = 1 + n_1 + n_2$. Additionally, the heap has the form $h = \mathit{Node}\ h_1\ ()\ h_2$, with $n_1 =\mid h_1 \mid$ and $n_2 =\mid h_2 \mid$. Let $k_i = \mathit{height}(h_i), k = \mathit{height}(h)$. There are two possibilities which exclude each other:

– $k_2 = k_1$ and $n_1 = 2^{k_1} - 1$. Then, $k_2 = k_1 = k - 1$ holds, and both n_1 and n_2 are uniquely fixed to $n_1 = 2^{k-1} - 1$ and $n_2 = n - n_1 - 1$.

– $k_1 = k_2 + 1$ and $n_2 = 2^{k_2} - 1$. Then, $k_1 = k - 1$ and $k_2 = k - 2$ holds, and both n_1 and n_2 are uniquely fixed to $n_2 = 2^{k-2} - 1$ and $n_1 = n - n_2 - 1$.

By i.h., h_1 and h_2 are the only classic heaps with cardinalities n_1 and n_2. Then h is the only classic heap with cardinality n.

Every natural number $n > 0$ can be expressed in binary form as $n = \langle b_{k-1}, \ldots, b_0 \rangle$, being $k = \lfloor \log_2 n \rfloor + 1$, $b_i \in \{0, 1\}$. As the most significant bit is always $b_{k-1} = 1$, we can uniquely associate to n the list $[b_{k-2}, \ldots, b_0]$ of its non-trivial bits. We will consider the following notations as equivalent:

$$n = \sum_{i=0}^{k-1} b_i 2^i = \langle b_{k-1}, ..., b_0 \rangle = [b_{k-2}, \ldots, b_0]$$

with $b_{k-1} = 1$. For reasons that will become clear later, we call the list $[b_{k-2}, \ldots, b_0]$ the *bit path* determined by n.

If $n > 1$, then the decomposition $\langle b_{k-1}, ..., b_0 \rangle$ has at least two bits. Paying attention to the next significant bit b_{k-2}, if its value is 1, then the left subtree of a classic heap with cardinality n is a complete tree of height $k - 1$. If its value is 0, then the right subtree is a complete tree of height $k - 2$. The remaining subtree has a cardinality n' obtained by 'deleting' the bit b_{k-2} from n, i.e. $n' = \langle b_{k-1}, b_{k-3}, \ldots, b_0 \rangle$. These observations lead to the following function definition computing the only classic heap with cardinality n:

```
heapC :: Int -> Heap ()
heapC 0            = Empty
heapC 1            = Node Empty () Empty
heapC n  | b_{k-2} = 1 = Node (fullT (k − 1)) () (heapC n')
         | b_{k-2} = 0 = Node (heapC n') () (fullT (k − 2))
                   where (b_{k-2}, k, n') = decompose n
```

Given $n > 1$, function `decompose` returns its bit $b_{k-2} = (n \textbf{ div } 2^{k-2}) \textbf{ mod } 2$, its number of bits $k = \lfloor \log_2 n \rfloor + 1$, and the natural number $n' = (n \textbf{ mod } 2^{k-2}) + 2^{k-2}$. The invocation *fullT* k returns a complete tree of height k and cardinality $2^k - 1$.

Lemma 8. *Given $n \in \mathbb{N}$, the invocation heapC n returns the only structural classic heap with this cardinality.*

Proof. By induction on n:

$n \leq 1$ An empty (resp. singleton) heap is returned, which has cardinality 0 (resp. 1).

$n > 1$ We make a case distinction according to the bit b_{k-2}:

- $b_{k-2} = 1$. The heap returned satisfies the invariant, since by i.h. the right subtree is a classic heap with cardinality n' and height $k - 1$, and the left one is a complete tree of height $k - 1$. The cardinality of the total heap is $1 + 2^{k-1} - 1 + (n \textbf{ mod } 2^{k-2}) + 2^{k-2}$, which is equivalent to n.
- $b_{k-2} = 0$. The heap returned satisfies the invariant, since by i.h. the left subtree is a classic heap with cardinality n' and height $k - 1$, and the right one is a complete tree of height $k - 2$. The cardinality of the total heap is $1 + (n \textbf{ mod } 2^{k-2}) + 2^{k-2} + 2^{k-2} - 1 = 2^{k-1} + n \textbf{ mod } 2^{k-2} = n$.

By Lemma 7, there is only one structural classic heap with cardinality n, so `heapC` builds it.

We can use bit paths to locate structural positions in a heap, provided they exist, by adopting the convention that a bit 0 implies a move to the left subtree, and a bit 1 implies a move to the right one. The path [] locates the root element.

Lemma 9. *Two classic heaps of cardinality $n - 1$ and $n = \langle b_{k-1}, ..., b_0 \rangle$ have the same structure, except by one element located at the position given by the bit path $[b_{k-2}, ..., b_0]$.*

Proof. From lemmas 4 and 6, we know that the structure of classic heaps grows level by level, filling them from left to right. Let k be the height of the classic heap with cardinality n. By Lemma 5 we know $2^{k-1} \leq n \leq 2^k - 1$. The position of the n-th element is located as follows:

- It is at the last level, that is the k-th one.
- Inside this level, its position is $n - 2^{k-1}$, in the range from the 0 to $2^{k-1} - 1$. It can be at the left half of the level (position between 0 and $2^{k-2} - 1$), or at the right one (position between 2^{k-2} and $2^{k-1} - 1$).

Now, we proceed by induction on the number k of binary digits of n, or equivalently on the level k of the n-th node:

$k = 1$, $n = 1 = \langle 1 \rangle = [\,]$ The path is the position of the root element, so the lemma is true in this case.

$k \geq 2$, $n = \langle b_{k-1}, \ldots, b_0 \rangle = [b_{k-2}, \ldots, b_0]$ By i.h. we assume that the lemma is true for heaps of height $k-1$. We consider two cases, attending the possible values of b_{k-2}:

- $b_{k-2} = 0$. Then $n = \langle 1, 0, b_{k-3}, \ldots, b_0 \rangle$, and the n-th element is at position $n - 2^{k-1}$ of level k:

$$\begin{aligned} n - 2^{k-1} &= \langle 1, 0, b_{k-3}, \ldots, b_0 \rangle - 2^{k-1} \\ &= \langle 1, b_{k-3}, \ldots, b_0 \rangle \leq 2^{k-2} - 1 \end{aligned}$$

  This position is at the first half of level k, in the left subtree, within this tree at level $k-1$, and at the same position than in the whole tree.

  The n-th element is, within the left subtree, the element $n' = n - 2^{k-2}$, because we need to subtract the root and the $2^{k-2} - 1$ elements of the right subtree, which is a complete tree of height $k-2$. Then,

$$\begin{aligned} n' = n - 2^{k-2} &= \langle 1, 0, b_{k-3}, \ldots, b_0 \rangle - 2^{k-2} \\ &= \langle 1, b_{k-3}, \ldots, b_0 \rangle \end{aligned}$$

  By i.h., the position of n' in the left subtree is given by the path $[b_{k-3}, \ldots, b_0]$. Then the path of n is $[0, b_{k-3}, \ldots, b_0]$, i.e. $[b_{k-2}, \ldots, b_0]$.

- $b_{k-2} = 1$. In this case, position

$$\begin{aligned} n - 2^{k-1} &= \langle 1, 1, b_{k-3}, \ldots, b_0 \rangle - 2^{k-1} \\ &= \langle 1, 0, b_{k-3}, \ldots, b_0 \rangle > 2^{k-2} \end{aligned}$$

  is located at the second half of level k, i.e. within the right subtree at position $n' = n - 2^{k-1}$, as the left subtree is a complete tree of height $k-1$. The rest of the reasoning is similar.

As a consequence of the lemma, we can provide another version of `heapC` which builds a structural classic heap from its cardinality. In this case, it uses an insertion function creating the new position, which is driven by the bit path (see Fig. 2):

```
path :: Int -> [Int]
path n          = reverse (path' n)
path' 1         = []
path' n | n > 1 = n `mod` 2 : path' (n `div` 2)

insert' :: Heap () -> [Int] -> Heap ()
insert' Empty        []     = Node Empty () Empty
insert' (Node l () r) (0:bs) = Node (insert' l bs) () r
insert' (Node l () r) (1:bs) = Node l () (insert' r bs)

insertC h n = insert' h (path n)
heapC n     = foldl insertC Empty [1..n]
```

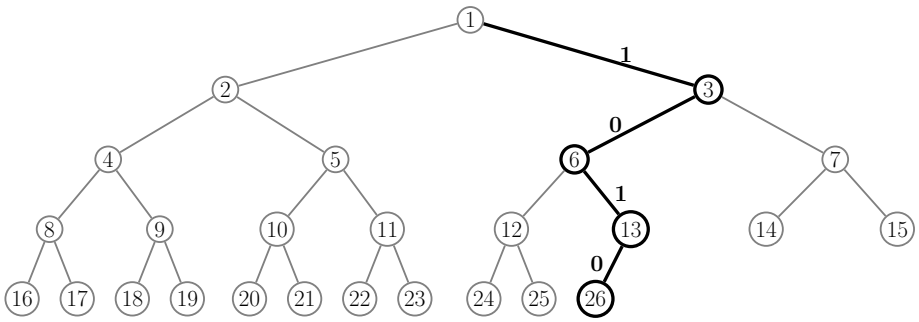


Figure 2: Classic heap, after element $26 = 1\mathbf{1010}_{(2}$ is added

In Fig. 2 we show what happens when the 26-th element is inserted by an invocation to `insertC h 26`.

Likewise, it is straightforward to delete the last structural position provided we have its cardinality:

```
delete' :: Heap () -> [Int] -> Heap ()
delete' (Node Empty () Empty) [] = Empty
delete' (Node l () r) (0:bs) = Node (delete' l bs) () r
delete' (Node l () r) (1:bs) = Node l () (delete' r bs)

deleteC h n = delete' h (path n)
```

3.2 Alternate heaps

Lemma 10. *The height k of an alternate heap with cardinality $n \geq 1$ is $\lfloor \log_2 n \rfloor + 1$.*

Proof. By induction on n:

$n \leq 2$ In this cases, trivially $k = n = \lfloor \log_2 n \rfloor + 1$.

$n > 2$ Both children are non-empty, with cardinalities n_1 and n_2. By i.h. $k_1 = \lfloor \log_2 n_1 \rfloor + 1$ and $k_2 = \lfloor \log_2 n_2 \rfloor + 1$. As $n_2 + 1 \geq n_1 \geq n_2$ and $n = 1 + n_1 + n_2$, then either $n = 1 + 2n_1$ or $n = 2n_1$. Then, in both cases $\lfloor \log_2 n \rfloor = \lfloor \log_2 n_1 \rfloor + 1 = k_1$. As $n_1 \geq n_2$ then $k_1 \geq k_2$, so $k = 1 + k_1 = 1 + \lfloor \log_2 n \rfloor$.

A trivial consequence of this lemma is that an alternate heap has minimal height. Other consequences are the following ones:

Lemma 11. *Given a non-empty alternate heap of height k with children of heights k_1 (left) and k_2 (right), then $k = k_1 + 1$, and either $k_1 = k_2$ or $k_1 = k_2 + 1$.*

Proof. Let n be the cardinality of the heap. We distinguish three cases:

- $n = 1$. Trivial, because $k = 1$ and $k_1 = k_2 = 0$.
- $n = 2$. Trivial, because $k = 2$, $k_1 = 1$ and $k_2 = 0$.
- $n \geq 3$. Then both children are non-empty. By Lemma 10 and the invariant: $k_1 = \lfloor \log_2 n_1 \rfloor + 1 \geq k_2 = \lfloor \log_2 n_2 \rfloor + 1$, so $k = 1 + max(k_1, k_2) = 1 + k_1$. If $n_1 = n_2$ then $k_1 = k_2$. Otherwise $n_1 = n_2 + 1$. In that case, by properties of the logarithms either $k_1 = k_2$ or $k_1 = k_2 + 1$.

Lemma 12. *Given an alternate heap $h_{(n,k)}$ ($n \geq 2$) with children $h1_{(n_1,k_1)}$ (left) and $h2_{(n_2,k_2)}$ (right) such that $k_1 = k_2 + 1$. Then $n_2 = 2^{k_2} - 1$, i.e. the right child is complete.*

Proof. We distinguish two cases:

- $n = 2$. Necessarily $n_2 = 0$ and $k_2 = 0$, so it holds.
- $n \geq 3$. Then $n_1, n_2 \geq 1$. Notice first that $k_1 = k_2 + 1$ necessarily means that $n_1 = n_2 + 1$ (otherwise $n_1 = n_2$ and by Lemma 10 $k_1 = k_2$). We reason by contradiction. Let us assume that $n_2 \neq 2^{k_2} - 1$. By Lemma 10, $k_2 = \lfloor \log_2 n_2 \rfloor + 1$, i.e. $2^{k_2 - 1} \leq n_2 < 2^{k_2}$. This means that $n_2 < 2^{k_2} - 1$. Then

$$\begin{aligned} (1)\quad & n = 1 + n_1 + n_2 = 1 + (n_2 + 1) + n_2 = 2 + 2n_2 \\ & < 2 + 2(2^{k_2} - 1) = 2^{k_2 + 1} \end{aligned}$$

  But also by Lemmas 10 and 11:

$$\begin{aligned} (2)\quad & \lfloor \log_2 n \rfloor + 1 = k = 1 + k_1 = 2 + k_2 \Rightarrow \\ & \lfloor \log_2 n \rfloor = 1 + k_2 \Rightarrow n \geq 2^{k_2 + 1} \end{aligned}$$

  Facts (1) and (2) are contradictory, so $n_2 = 2^{k_2} - 1$ is true.

Lemma 13. *In any alternate heap of height k, its first $k-1$ levels are complete.*

Proof. By induction on k:

$k < 2$ The property holds trivially.

$k \geq 2$ By Lemma 11, $k = k_1 + 1$ and either $k_1 = k_2$ or $k_1 = k_2 + 1$. By i.h. the left child has its $k_1 - 1$ first levels complete, while the right child has its $k_2 - 1$ first levels complete. The level of the root is always complete. We have two cases:

- $k_1 = k_2$. Then the first $(k_1 - 1) + 1 = k_1 = k - 1$ levels are complete.
- $k_1 = k_2 + 1$. Then by Lemma 12 ($k \geq 2 \Rightarrow n \geq 2$) the right child has all its $k_2 = k_1 - 1$ levels complete. So the first $(k_1 - 1) + 1 = k_1 = k - 1$ levels are complete.

Lemma 14. *For each $n \in \mathbb{N}$, there exists a unique structural alternate heap whose cardinality is n.*

Proof. By induction on n:

$n = 0$ Heap *Empty* is the only one whose cardinality is 0.

$n > 0$ In this case there exist unique n_1 and $n_2 \in \mathbb{N}$ such that $n = n_1 + n_2 + 1$ and $n_1 \geq n_2 \geq n_1 - 1$:

- n is odd. Then $n_1 = n_2 = (n-1)/2$.
- n is even. Then $n_1 = n/2$. In this case $n_1 = n_2 + 1$.

By i.h. there exist unique structural alternate heaps h_1 and h_2 whose corresponding cardinalities are n_1 and n_2. Then the unique heap we are looking for is *Node* h_1 () h_2.

From the previous proof we can write a function *heapA* which given a natural number n, builds the structural alternate heap with cardinality n:

```
heapA :: Int -> Heap ()
heapA 0                 = Empty
heapA n | odd  n        = Node (heapA n') () (heapA n')
                           where n' = (n-1)/2
        | even n        = Node (heapA (n'+1)) () (heapA n')
                           where n' = (n-2)/2
```

Lemma 15. *Two alternate heaps of cardinalities n and $m = n+1 = \langle b_{k-1}, ..., b_0 \rangle$ have the same structure, except for one element located at the position given by the bit path $[b_0, ..., b_{k-2}]$.*

Proof. We denote the alternate heaps by h_n and h_m for simplification. We proceed by induction on the height k of h_n.

$k = 0$ In this case, the property is trivial because h_n = *Empty*, h_m = *Node Empty* () *Empty*, $m = 1 = \langle 1 \rangle$, and the bit path of the element is [ ], as expected.

$k > 0$ In this case h_n = *Node* h_{n_1} () h_{n_2}, where $n = n_1 + 1 + n_2$ and $n_1 \geq n_2 \geq n_1 - 1$. As we have seen in Lemma 14 we can distinguish the following cases:

- n is odd. Then m is even and consequently:

$$\begin{aligned}
(m_1, m_2) &= (m/2, (m-2)/2) \\
&= ((n+1)/2, (n-1)/2) && \{m = n+1\} \\
&= ((n-1)/2 + 1, (n-1)/2) \\
&= (n_1 + 1, n_1) && \{n \text{ is odd}\} \\
&= (n_1 + 1, n_2) && \{n \text{ is odd}\}
\end{aligned}$$

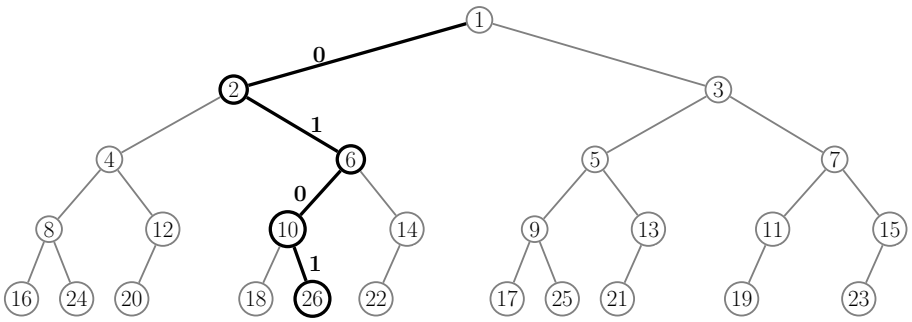


Figure 3: Alternate heap, after element $1\mathbf{1010}_{(2}$ is added

So the left child of h_m has one element more than the left child of h_n, while the right children have the same number of elements. By Lemma 14, h_{n_2} and h_{m_2} are the same structural alternate heap. Additionally, if $m = \langle b_{k-1}, ..., b_0 \rangle$ then $b_0 = 0$ and $m_1 = m/2 = \langle b_{k-1}, ..., b_1 \rangle$. By i.h., h_{n_1} and h_{m_1} have the same structure except by one element located in h_{m_1} at the position given by the bit path $[b_1, \ldots, b_{k-1}]$, which is located in h_m at the position given by the path $[0, b_1, \ldots, b_{k-1}] = [b_0, b_1, \ldots, b_{k-1}]$.

– n is even. Then m is odd, $m_1 = m_2 = (m-1)/2$, and:

$$\begin{aligned}(m_1, m_2) &= ((m-1)/2, (m-1)/2) \\ &= (n/2, n/2) && \{m = n+1\} \\ &= (n_1, n_1) && \{n \text{ is even}\} \\ &= (n_1, n_2 + 1) && \{n \text{ is even}\}\end{aligned}$$

The reasoning is symmetric to the previous case.

Notice that both cases share a same structural growing pattern, transforming the sizes of its subtrees as follows:

$$\mathit{Node}\ h_{n_1}\ ()\ h_{n_2} \rightsquigarrow \mathit{Node}\ h_{n_2+1}\ ()\ h_{n_1}$$

As a consequence of this lemma, we can write another version of `heapA` which builds a structural alternate heap from its cardinality. It uses the same insertion function as the second version of `heapC`, but now it is driven by the reverse of the bit path given by function `path'` (see Fig. 3). We include a deletion function also driven by the reverse of the bit path.

```
insertA, deleteA :: Heap () -> Int -> Heap ()
insertA h n = insert' h (path' n)
heapA n     = foldl insertA Empty [1..n]
deleteA h n = delete' h (path' n)
```

The last remark in the proof of this lemma leads us to the definition of an insertion function where the children rotate in order to preserve the heap structural invariant. Such insertion function, unlike that of the classic heaps, needs neither the bit path nor the heap cardinality:

```
insertRotA :: Heap () -> Heap ()
insertRotA Empty = Node Empty () Empty
insertRotA (Node h1 () h2) = Node (insertRotA h2) () h1
```

As a consequence, given an alternate heap h with cardinality $n-1$:

$$insertA\ h\ n = insertRotA\ h$$

Analogously, we can define a deletion function with rotation:

```
deleteRotA (Node Empty () Empty) = Empty
deleteRotA (Node h1 () h2) = Node h2 () (deleteRotA h1)
```

4 Considering elements and ordering

Heaps are useful for storing elements because they satisfy a particular ordering property: the minimum element is at the trees's root and the property is recursively satisfied by all the subtrees. It is straightforward to convert the above structural versions for insertion and deletion into their corresponding non-structural ones, without changing their asymptotic costs. For insertion in a classic heap we just propagate the maximum element downwards the insertion path:

```
insertCH' :: Ord a => Heap a -> a -> [Int] -> Heap a
insertCH' Empty x [] = Node Empty x Empty
insertCH' (Node l y r) x (0:bs)
   = Node (insertCH' l max bs) min r
    where (min, max) = if x <= y then (x,y) else (y,x)
insertCH' (Node l y r) x (1:bs)
   = Node l min (insertCH' r max bs)
    where (min, max) = if x <= y then (x,y) else (y,x)
insertCH h x n = insertCH' h x (path n)
```

For deletion in a classic heap we first modify function `delete'` of Sec. 3.1 in such a way that it, not only deletes the last structural position, but also returns the element stored there:

```
extractLastC :: Heap a -> [Int] -> (a,Heap a)
extractLastC (Node Empty x Empty) [] = (x, Empty)
extractLastC (Node l y r) (0:bs)
   = (x, Node l' y r) where (x, l') = extractLastC l bs
extractLastC (Node l y r) (1:bs)
   = (x, Node l y r') where (x, r') = extractLastC r bs
```

The next step is replacing the root element by the one just extracted, and sinking it down the heap in order to re-establish the heap property:

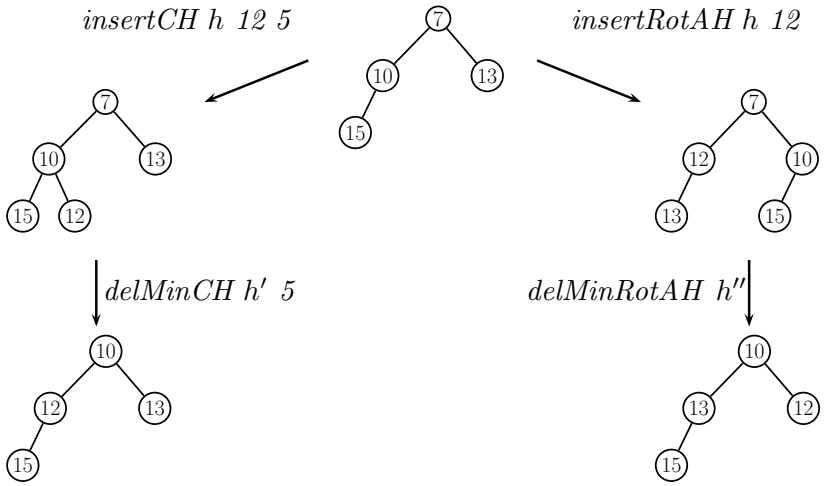


Figure 4: Insertion and deletion with order

```
delMinCH :: Ord a => Heap a -> Int -> Heap a
delMinCH h 1 = Empty
delMinCH h n = sinkRoot (Node l x r)
    where (x, Node l _ r) = extractLastC h (path n)
```

Function `sinkRoot` runs in logarithmic time and ensures that the minimum element is always at the root of each subtree:

```
sinkRoot :: Ord a => Heap a -> Heap a
sinkRoot t@(Node Empty x Empty) = t
sinkRoot t@(Node (Node Empty y Empty) x Empty
   | x <= y = t
   | otherwise = Node (Node Empty x Empty) y Empty
sinkRoot t@(Node l@(Node l1 x1 r1) x r@(Node l2 x2 r2))
   | x <= min x1 x2 = t
   | x1 <= min x x2 = Node (sinkRoot (Node l1 x r1)) x1 r
   | x2 <= min x x1 = Node l x2 (sinkRoot (Node l2 x r2))
```

Notice that the definition takes advantage of Lemma 6 of classic heaps in the sense that the execution may eventually run into the second equation, and in that there is no need for a symmetric equation to this.

Insertion and deletion in alternate heaps have been studied in [Paulson, 1996, Sec. 4.16]. With different names and syntax the following functions are essentially the same presented there. Insertion is a slight modification of function `insertRotA` of Sec. 3.2:

```
insertRotAH :: Ord a => Heap a -> a -> Heap a
insertRotAH Empty      x = Node Empty x Empty
insertRotAH (Node l y r) x = Node (insertRotAH r max) min l
    where (min, max) = if x <= y then (x,y) else (y,x)
```

Deletion of the minimum element first requires extracting the last structural element. This is done by extracting the leftmost one and then re-structuring the heap:

```
extractLastA :: Heap a -> (a, Heap a)
extractLastA (Node Empty x Empty) = (x, Empty)
extractLastA (Node l y r) = (x, Node r y l')
                     where (x, l') = extractLastA l
```

The rest is identical to the classic heap deletion:

```
delMinRotAH :: Ord a => Heap a -> Heap a
delMinRotAH (Node Empty _ Empty) = Empty
delMinRotAH h = sinkRoot (Node l x r)
          where (x, Node l _ r) = extractLastA h
```

Differently to the structural case, in general a pair of insertion and deletion operations starting from the same initial heap produces different heaps in the classic and in the alternate cases, even if the final structure is the same, as Fig. 4 shows.

5 Explicit structural isomorphism between heaps

In this section we define a explicit structural isomorphism between classic and alternate heaps, called *shuffle*:

```
shuffle :: Heap a -> Heap a
shuffle Empty          = Empty
shuffle (Node h1 x h2) = Node h1' x h2'
       where (h1',h2') = comb (shuffle h1, shuffle h2)
```

Its definition uses another function *comb* which given two heaps re-distributes their elements by levels:

```
comb :: (Heap a, Heap a) -> (Heap a, Heap a)
comb (Node l1 x r1, Node l2 y r2)
     = (Node l1' x r1', Node l2' y r2')
       where (l1', r1') = comb (l1, l2)
             (l2', r2') = comb (r1, r2)
comb pair = pair
```

It is easy to see that the cost of *shuffle* is in $O(n \log n)$. Proving that it is an isomorphism needs some previous lemmas of *comb* and *shuffle*. The following ones may be applied to a class of binary trees including but not restricted to alternate and classic heaps. Detailed proofs can be found in the Appendix.

The following lemma asserts that function *comb* acts by levels, i.e. it may change the position of an element but not its level. This is illustrated in Figure 5.

Let p denote a (possibly empty) sequence of bits, and $b : p$, $p\#b$ non-empty sequences of bits respectively starting or ending with bit b. The empty sequence is denoted by $[\,]$. Let us denote the multiset of elements at level i in the tree h by $els\ i\ h_{(n,k)}$. If $i > k$, we consider that $els\ i\ h_{(n,k)} = \emptyset$.

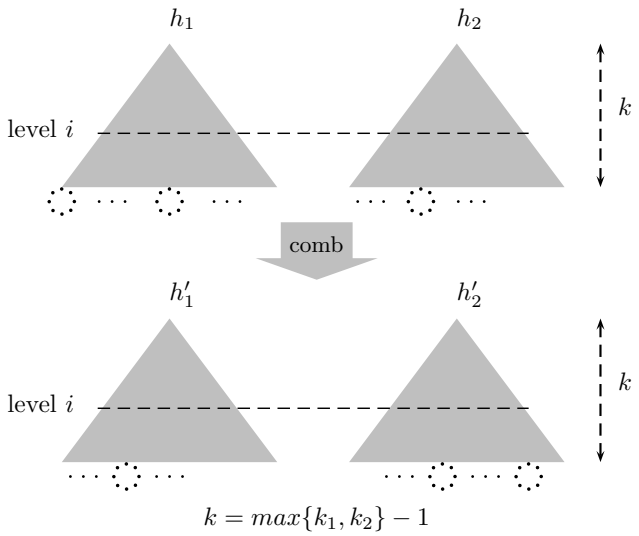


Figure 5: Lemma 16: *comb* acts by levels

Lemma 16. *Let $h_{1(n_1,k_1)}, h_{2(n_2,k_2)}$ be binary trees such that*

- $|k_1 - k_2| \leq 1$, *and*
- *whose first $max\{k_1, k_2\} - 1$ levels are complete (at least).*

Let $h'_{1(n'_1,k'_1)}$, $h'_{2(n'_2,k'_2)}$ be binary trees such that $(h'_1, h'_2) = comb\ (h_1, h_2)$, then:

1. *$n_1 + n_2 = n'_1 + n'_2$ and $max\{k_1, k_2\} = max\{k'_1, k'_2\}$.*
2. *If there is an element x located in h_1 (resp. h_2) at path $[\,]$, then x is located in h'_1 (resp. h'_2) at path $[\,]$.*
3. *If there is an element x located in h_1 at path $0 : p$ (resp. $1 : p$), then x is located in h'_1 (resp. h'_2) at path $p\#0$.*
4. *If there is an element x located in h_2 at path $0 : p$ (resp. $1 : p$), then x is located in h'_1 (resp. h'_2) at path $p\#1$.*
5. *For each $i \geq 1$, els i $h_1 \cup$ els i h_2 = els i $h'_1 \cup$ els i h'_2.*

Proof. By structural induction on the argument trees (see Appendix).

The following theorem establishes that *shuffle* reverses the paths of the elements. This is illustrated in Figure 6. Let $\hat{p}$ denote the reverse of sequence p.

Theorem 17. *Let $h_{(n,k)}$ be a tree such that*

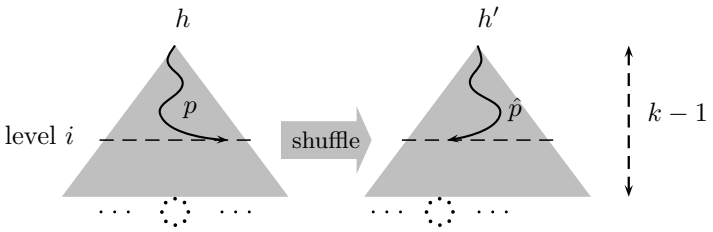


Figure 6: Theorem 17: *shuffle* reverses the paths of the elements

- *it is Empty, or*
- *if* $h_{(n,k)} = Node\ h_{1(n_1,k_1)}\ y\ h_{2(n_2,k_2)}$*, then at least its first* $k-1$ *levels are complete.*

Let $h'_{(n',k')}$ *such that* $h' = shuffle\ h$*. Then*

1. $n' = n$ *and* $k' = k$
2. h' *has complete the same levels as* h
3. *For any element* x *in* h *located at path* p*,* x *is located in* h' *at path* $\hat{p}$

Proof. By structural induction on h and Lemma 16 (see Appendix).

A direct corollary from Theorem 17 and Lemmas 4, 13, 9 and 15 is the following one:

Corollary 18. *Given a heap* h *of cardinality* n*, and* $h' = shuffle\ h$*, then if* h *is a structural classic (resp. alternate) heap, then* h' *is a structural alternate (resp. classic) heap of cardinality* n*.*

As classic and alternate structural heaps are uniquely determined by their cardinalities, *shuffle* is its own inverse, i.e. it is a structural isomorphism between classic and alternate heaps. Consequently, if h is an alternate heap with cardinality n:

$$insertC\ (shuffle\ h)\ (n+1) = shuffle\ (insertRotA\ h)$$
$$deleteC\ (shuffle\ h)\ n = shuffle\ (deleteRotA\ h)$$

6 Transformations between data structures

We have seen that classic heaps and arrays are convenient data structures when programming in an imperative setting, while flexible arrays and alternate heaps

are more convenient in a functional one. In particular, the second ones do not need the cardinality or its bits in order to look up, update, add or remove elements. The `shuffle` operation of Sec. 5 provides a way of transforming the former into the latter and the other way around. Unfortunately, its cost $O(n \log n)$ is not as convenient as one would like. After some unsuccessful attempts, we have found a linear algorithm to implement this operation, by making an intensive use of the lemmas and theorems proved in Sec. 3. The main idea is first decomposing a binary tree into the list of its levels, then permuting each individual level, and finally composing the new binary tree. It is clear that the initial and final transformations can be easily programmed with a linear cost. The key for permuting the elements of each level with a linear cost is having the permutation for the elements pre-computed as a list of indices. By lemmas 9 and 15, two corresponding positions n and n' of a given level in a classic and its equivalent alternate heaps are such that the bits of n are the inverse of the bits of n'. Then, the only thing needed to pre-compute the permutation of level k is to reverse the bits of the numbers $0, 1, \ldots, 2^k - 1$. In principle, reversing k bits needs a time in $O(k)$, it does not matter whether we do it in the binary or in the decimal system. The second idea for doing it in constant time is reusing the permutation computed for the previous level. The following linear time function generates the list of the permutations of all the levels of an infinite binary tree:

```
genLevels :: [[Int]]
genLevels          = genLevels' 1 [0]
genLevels' k level = level : genLevels' (2*k) nextLevel
   where nextLevel = concat (map addBit level)
         addBit i  = [i,k+i]
```

For instance, `take 4 genLevels` generates the list:

```
[[0],[0,1],[0,2,1,3],[0,4,2,6,1,5,3,7]]
```

The linear time `shuffle` operation is defined as follows:

```
shuffle :: BinTree a -> BinTree a
shuffle = mount . permuteLevels . list2array . unmount
```

Where `unmount :: BinTree a -> [[Maybe a]]` (not shown) uses a FIFO queue to compute the list of levels of the input tree. The type `Maybe a` is needed to mark with `Nothing` the holes of the last level corresponding to empty subtrees. If a queue with amortized time $O(1)$ for all its operations is used (see, for example [Okasaki, 1998, Chapter 5]), then the total cost of `unmount` is linear in the worst case. Operation `mount` does the opposite and can be programmed with a simple `foldr`.

The operation `list2array`, shown below, converts (always in linear time) each level into a Haskell array. This is essential to have constant access to each element in order to move it to its final destination in the level. Finally, `permuteLevels` uses the arrays and the pre-computed indices to permute the elements:

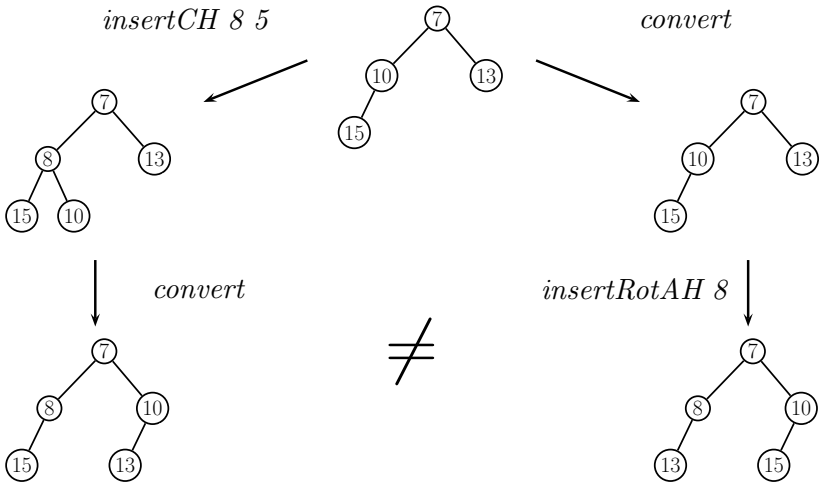


Figure 7: Only structural isomorphism holds

```
list2array :: [[Maybe a]] -> [Array Int (Maybe a)]
list2array = snd . mapAccumL f 1
             where f k xs = (2*k, listArray (0,k-1) xs)
permuteLevels :: [Array Int (Maybe a)] -> [[Maybe a]]
permuteLevels = zipWith permute genLevels
                where permute is arr = map (arr!) is
```

If the starting point is an imperative array (or a classic heap implemented as an array), a simple linear function `toLists` can convert it into a list of levels `[[Maybe a]]`. From that point, the above transformations can translate it into a binary tree representing a flexible array. The opposite path can be followed if the starting point is an alternate heap, or a flexible array, and the aim is producing an imperative array (or a classic heap).

Finally, if one wishes to preserve the heap property, it is obvious that no version of `shuffle` preserves it, since the elements of each level are permuted, and the order relation of a particular element with either its parent or its children is lost. Then, we can use the well-known linear time algorithm that sinks the elements starting from the last level (see, e.g. [Paulson, 1996, Section 4.16]) in order to re-establish the property:

```
heapify :: Ord a => Heap a -> Heap a
heapify Empty        = Empty
heapify (Node l x r) = sinkRoot (heapify l) x (heapify r)

convert :: Ord a => Heap a -> Heap a
convert = heapify . shuffle
```

In general, if `h` is a classic heap we should expect `convert (insertCH h x n)` to be different from `insertRotAH (convert h) x`, as Fig 7 shows.

7 Conclusion and related work

We have characterised William's and Paulson's heaps by means of two simple invariants from which all their properties can be formally deduced. Given a cardinality n, we have provided simple functions for building the unique structure corresponding to such cardinality, and also for creating and deleting the last structural position. Additionally, we have defined an operation implementing the unique isomorphism between them, which maps their corresponding structural positions. Even taking into account the actual elements stored in the heaps, and eventually restoring the heap property, the operation runs in a worst case linear time. This operation could be the basis for efficiently transforming imperative arrays and/or William's heaps into funcional flexible arrays and/or Paulson's heaps.

Flexible functional arrays were originally proposed by W. Braun in 1983. An account of them can be found at [Hoogerwoord, 1993]. Paulson [Paulson, 1996] had the idea of using them for implementing heaps. Three new algorithms working on Braun trees were proposed in [Okasaki, 1997]. The last one converts a list into a flexible array in time $O(n)$, so it partially competes with our transformation of Sec. 6. It resembles our `mount` algorithm in the sense that it also builds a binary tree from a list of levels, but the placement of elements inside a level is done by an ingenious use of a `zipWith3` function.

In [Sack and Strothotte, 1990] another interesting transformation of heaps is proposed . There, a classic heap is decomposed into a list of complete trees with an additional root, called *pennants*. This decomposition allows two new operations on classic heaps, *split* and *merge*, to be done in a time $O(\log^2 n)$. The merge operation can be done in $O(\log n)$ in leftist and binomial heaps, but previously to this work it had linear cost for a classic heap.

Leftist trees heaps were invented by C. Crane in 1972, and they already appear in Knuth's books of 1973. Its functional implementation is straightforward (e.g. see [Núñez et al., 1995]). Binomial heaps were originally proposed by J. Vuillemin in 1978 and they are explained in any modern text on data structures (see e.g. [Horowitz et al., 1995, Sec. 9.4]). The first functional implementation is due to D. King [King, 1994]. Since then, many other people in the functional programming community have become interested in them [Okasaki, 1998, Hinze, 1999]. Fibonacci heaps are due to M. Fredman and R. Tarjan in 1987 [Fredman and Tarjan, 1987]. We are not aware of functional implementations of this structure. In [Hinze, 2001] the author presents a functional implementation of a data structure *priority search queue*, originally proposed by E. M. McCreight in 1985, which combines the operations of dictionaries with those of priority queues achieving logarithmic costs for all of them.

8 Acknowledgements

This work has been partially funded by the projects TIN2009-14312-C02-01 (TESIS), TIN2008-04103/TSI, TIN2008-06622-C03-01/TIN (STAMP), and S20-09/TIC-1465 (PROMETIDOS).

References

[Cormen et al., 2001] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. (2001). *Introduction to Algorithms*. The MIT Press/McGraw-Hill.

[Fredman and Tarjan, 1987] Fredman, M. and Tarjan, R. (1987). Fibonacci heaps and their use in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615.

[Hinze, 1999] Hinze, R. (1999). Explaining binomial heaps. *FUNCTIONAL PEARL: Journal of Functional Programming*, 9(1):93–104.

[Hinze, 2001] Hinze, R. (2001). A Simple Implementation Technique for Priority Search Queues. In *Int. Conf. on Functional Programming, ICFP'01*, pages 110–121. ACM.

[Hoogerwoord, 1993] Hoogerwoord, R. R. (1993). A logarithmic implementation of flexible arrays. In *Int. Conference on Mathematics of Program Construction, MPC'92*, volume 669 of *LNCS*, pages 191–207. Springer.

[Horowitz et al., 1995] Horowitz, E., Mehta, D., and Sahni, S. (1995). *Fundamentals of Data Structures in C++*. W. H. Freeman & Co.

[King, 1994] King, D. J. (1994). Functional Binomial Queues. In *Procs. of the Glasgow Workshop on Functional Programming*. Springer-Verlag.

[Núñez et al., 1995] Núñez, M., Palao, P., and Peña, R. (1995). A Second Year Course on Data Structures based on Functional Programming. In *FPLE'95. LNCS 1022*, pages 65–84. Springer-Verlag.

[Okasaki, 1997] Okasaki, C. (1997). Three Algorithms on Braun Trees. *FUNCTIONAL PEARL: Journal of Functional Programming*, 7(6):661–666.

[Okasaki, 1998] Okasaki, C. (1998). *Purely Functional Data Structures*. Cambridge University Press.

[Paulson, 1996] Paulson, L. C. (1996). *ML for the working programmer (2nd ed.)*. Cambridge University Press.

[Sack and Strothotte, 1990] Sack, J.-R. and Strothotte, T. (1990). A Characterization of Heaps and Its Applications. *Information and Computation*, 86:69–86.

Appendix: detailed proofs

Lemma 16

Proof. By structural induction on the argument trees. In order to be able to reason by induction we have to prove that assuming the premises hold for h_1 and h_2 then the premises also hold for the tree of the internal calls, i.e. for l_1, l_2 and for r_1, r_2. So, assume $h_{1(n_1,k_1)}, h_{2(n_2,k_2)}$ non-empty trees such that

- $|k_1 - k_2| \leq 1$, and
- whose first $max\{k_1, k_2\} - 1$ levels are complete.

If $h_{1(n_1,k_1)} = Node\ l_{1(n_{11},k_{11})}\ y\ r_{1(n_{12},k_{12})}$ and $h_{2(n_2,k_2)} = Node\ l_{2(n_{21},k_{21})}\ z\ r_{2(n_{22},k_{22})}$, then we have to prove that:

- $|k_{11} - k_{21}| \leq 1$ and $|k_{12} - k_{22}| \leq 1$
- l_1 and l_2 have their first $max\{k_{11}, k_{21}\} - 1$ levels complete
- r_1 and r_2 have their first $max\{k_{12}, k_{22}\} - 1$ levels complete

We distinguish three cases: $k_1 = k_2$, $k_1 = k_2 + 1$ and $k_2 = k_1 + 1$.

- $k_1 = k_2$. Then $max\{k_1, k_2\} = k_1 = k_2$. By hypothesis the first $k_1 - 1$ levels of h_1 and h_2 are complete, i.e. the first $k_1 - 2$ levels of l_1 and l_2 are complete. This means that $k_{11} = k_1 - 2$ or $k_{11} = k_1 - 1$, and $k_{21} = k_1 - 2$ or $k_{21} = k_1 - 1$. For any of the four possible combinations $|k_{11} - k_{21}| \leq 1$. Also, $max\{k_{11}, k_{21}\} - 1 = k_1 - 2$ or $max\{k_{11}, k_{21}\} - 1 = k_1 - 3$, so the first $max\{k_{11}, k_{21}\} - 1$ levels of l_1 and l_2 are complete. The reasoning for the right children is analogous.
- $k_1 = k_2 + 1$. Then $max\{k_1, k_2\} = k_1$. By hypothesis the first $k_1 - 1$ levels of h_1 and h_2 are complete, i.e. the first $k_1 - 2$ levels of l_1 and l_2 are complete. This means that $k_{11} = k_1 - 2$ or $k_{11} = k_1 - 1$, and $k_{21} = k_1 - 2$. For any of the two possible combinations $|k_{11} - k_{21}| \leq 1$. Again, $max\{k_{11}, k_{21}\} - 1 = k_1 - 2$ or $max\{k_{11}, k_{21}\} - 1 = k_1 - 3$, so the first $max\{k_{11}, k_{21}\} - 1$ levels of l_1 and l_2 are complete. The reasoning for the right children is analogous.
- $k_2 = k_1 + 1$. Reasoning is symmetric to the previous one.

Proof of (1)

By structural induction on h_1 and h_2.

We prove first that $n_1 + n_2 = n'_1 + n'_2$ by structural induction on h_1 and h_2:

$h_1 = \mathit{Empty}$ **or** $h_2 = \mathit{Empty}$ In this case $h'_1 = h_1$ and $h'_2 = h_2$, so the property holds trivially.

$h_{1(n_1,k_1)} = \mathit{Node}\ l_{1(n_{11},k_{11})}\ y\ r_{1(n_{12},k_{12})}$, $h_{2(n_2,k_2)} = \mathit{Node}\ l_{2(n_{21},k_{21})}\ \ z\ r_{2(n_{22},k_{22})}$ Let us assume the results of the recursive calls to *comb* are respectively:

$$l'_{1(n'_{11},k'_{11})}, r'_{1(n'_{21},k'_{21})} \text{ and } l'_{2(n'_{12},k'_{12})}, r'_{2(n'_{22},k'_{22})}$$

By definition of *comb*, $n'_1 = n'_{11} + n'_{21} + 1$ and $n'_2 = n'_{12} + n'_{22} + 1$. By i.h. $n_{11} + n_{21} = n'_{11} + n'_{21}$ and $n_{12} + n_{22} = n'_{12} + n'_{22}$, so

$$\begin{aligned} n_1 + n_2 &= (n_{11} + n_{12} + 1) + (n_{21} + n_{22} + 1) \\ &= (n'_{11} + n'_{21} + 1) + (n'_{12} + n'_{22} + 1) \\ &= n'_1 + n'_2 \end{aligned}$$

The fact that $max\{k_1, k_2\} = max\{k'_1, k'_2\}$ is very easy to prove by structural induction. If any of the two trees is *Empty* the property holds trivially by definition. Otherwise:

$$\begin{aligned} max\{k_1, k_2\} &= max\{max\{k_{11}, k_{12}\} + 1, max\{k_{21}, k_{22}\} + 1\} \\ &= max\{k_{11}, k_{12}, k_{21}, k_{22}\} + 1 \quad \{\text{positive heights}\} \\ &= max\{max\{k_{11}, k_{21}\} + 1, max\{k_{12}, k_{22}\} + 1\} \\ &= max\{max\{k'_{11}, k'_{21}\} + 1, max\{k'_{12}, k'_{22}\} + 1\} \quad \{\text{i.h.}\} \\ &= max\{k'_1, k'_2\} \end{aligned}$$

Proofs of (2) to (4)

Property (2) is trivial from the definition of *comb*. We now prove both facts of property (3) simultaneously by structural induction on h_1 and h_2:

$h_1 = \mathit{Empty}$ Trivially true.

$h_1 = \mathit{Node}\ l_1\ y\ r_1$ Assume first, there is an element x located in h_1 at path $0 : p$. This means that x is located in l_1 at path p. We can distinguish three cases:

- $p = [\,]$. By (2), x is located in l'_1 at path $[\,]$, and also located in h'_1 at path $0 : [\,] = p\#0$.
- $p = 0 : p'$. By i.h. x is located in l'_1 at path $p'\#0$, and also located in h'_1 at path $0 : p'\#0 = p\#0$.

– $p = 1 : p'$. By i.h. x is located in r'_1 at path $p'\#0$, and also located in h'_1 at path $1 : p'\#0 = p\#0$.

Assume now, there is an element x located in h_1 at path $1 : p$. Such element is located in r_1 at path p, and the rest of the reasoning is similar to the previous case.

The reasoning is symmetric to prove (4).

Proof of (5)

Property (5) is a consequence of the previous ones. By (1) and the properties of the cardinal of a tree, it is sufficient to prove that for each i, $els\ i\ h_1 \cup els\ i\ h_2 \subseteq els\ i\ h'_1 \cup els\ i\ h'_2$. If there was a level i and an element z such that $z \in els\ i\ h'_1 \cup els\ i\ h'_2$ but $z \notin els\ i\ h_1 \cup els\ i\ h_2$, then it would happen that $n_1+n_2 = |els\ i\ h_1|+|els\ i\ h_2| < |els\ i\ h'_1| + |els\ i\ h'_2| = n'_1 + n'_2$, which is not true, by (1).

Let us prove then that for each i, $els\ i\ h_1 \cup els\ i\ h_2 \subseteq els\ i\ h'_1 \cup els\ i\ h'_2$. If h_1 and h_2 are empty trees, or i is a level where there are no elements in h_1 nor h_2, then the property is trivially true. If $z \in els\ i\ h_1 \cup els\ i\ h_2$, then either $z \in els\ i\ h_1$ or $z \in els\ i\ h_2$ at a given path p.

Notice that the level i where an element is located at a tree is equal to the length of its path p plus one. This means that the path length of all the elements located at the same level of a tree is the same. So, a consequence of properties (2)-(4) is that z is located either in h'_1 or h'_2 at a path p' of the same length as p, and consequently at the same level i, i.e. $z \in els\ i\ h'_1$ or $z \in els\ i\ h'_2$.

□

Theorem 17

Proof. By structural induction on h and Lemma 16.

If $h = Empty$, by definition of *shuffle*, $n' = n = 0$, $k' = k = 0$ and $h' = h$ has no elements.

If $h_{(n,k)} = Node\ h_{1(n_1,k_1)}\ y\ h_{2(n_2,k_2)}$, let $h''_{1(n''_1,k''_1)} = shuffle\ h_1$ and $h''_{2(n''_2,k''_2)} = shuffle\ h_2$. By hypothesis, h has at least its first $k-1$ levels complete, which means that (if they are non-empty) h_1 and h_2 have at least their first $k-2$ levels complete. As $k = max\{k_1, k_2\} + 1$, then $k - 2 = max\{k_1, k_2\} - 1$, i.e. $k - 2 \geq k_1 - 1$ and $k - 2 \geq k_2 - 1$. As a consequence h_1 has at least $k_1 - 1$ levels complete and h_2 has at least $k_2 - 1$ levels complete. This means that we can apply the induction hypothesis to conclude

– $n''_1 = n_1$, $n''_2 = n_2$, $k''_1 = k_1$ and $k''_2 = k_2$

– h''_1 (resp. h''_2) has complete the same levels as h_1 (resp. h_2)

– For any element x in h_1 (resp. h_2) located at path p, x is located in h_1'' (resp. h_2'') at path $\hat{p}$

As h has at least its first $k-1$ levels complete, this means that h_1 and h_2 meet the conditions of Lemma 16, i.e. $|k_1 - k_2| \leq 1$ and their first $max\{k_1, k_2\} - 1$ levels are complete (as $k = max\{k_1, k_2\} + 1$). As by i.h. $k_1'' = k_1$, $k_2'' = k_2$ and h_1'', h_2'' have complete respectively the same levels as h_1, h_2, then h_1'', h_2'' also meet the same conditions.

By applying Lemma 16, if $(h'_{1(n_1', k_1')}, h'_{2(n_2', k_2')}) = comb(h_1'', h_2'')$, then $n_1'' + n_2'' = n_1' + n_2'$. Consequently $n = n_1 + n_2 + 1 = n_1'' + n_2'' + 1 = n_1' + n_2' + 1 = n'$. Also $k = max\{k_1, k_2\} + 1 = max\{k_1'', k_2''\} + 1 = max\{k_1', k_2'\} + 1 = k'$.

Let us prove now that h' has complete the same levels as h. By i.h. we know that h_1'' and h_2'' have complete the same levels as h_1 and h_2, respectively. Trivially, if level $i = 1$ is complete in h, it is also complete in h'. If level $i > 2$ is complete in h then level $i - 1$ is complete both in h_1 and h_2, and by i.h. also in h_1'' and h_2''. By Lemma 16 , $els\ (i-1)\ h_1'' \cup els\ (i-1)\ h_2 = els\ (i-1)\ h_1' \cup els\ (i-1)\ h_2'$, which means that level $i - 1$ is also complete both in h_1' and h_2', i.e. level i is complete in h'. As the number of elements is the same, it is not possible that h' has more complete levels than h.

Assume there is an element x in h located at path p. We proceed by induction on the length l of p. We have three cases:

$(l = 0)$ $p = [\ ]$ By definition of *shuffle*, x is located in h' at path $p = [\ ] = \hat{p}$.

$(l \geq 1)$ $p = 0 : p'$ Then x is located in h_1 at path p'. By i.h., x is located in h_1'' at path $\hat{p'}$. We have again three cases.

- $\hat{p'} = [\ ]$. By Lemma 16 , x is located in h_1' at path $\hat{p'} = [\ ]$. So, it is located in h' at path $0 : [\ ] = \hat{p}$.
- $\hat{p'} = 0 : p''$. By Lemma 16, x is located in h_1' at path $p''\#0$, and also in h' path $0 : p''\#0 = \hat{p'}\#0 = \hat{p}$.
- $\hat{p'} = 1 : p''$. By Lemma 16, x is located in h_2' at path $p''\#0$, and also in h' at path $1 : p''\#0 = \hat{p'}\#0 = \hat{p}$.

$(l \geq 1)$ $p = 1 : p'$ This case is very similar to the previous one.

□

Corollary 18

Proof. From Lemmas 4 and 13, both classic and alternate heaps hold the conditions of Theorem 17. From Lemma 9 we conclude that the elements contained

in a classic heap of cardinal n are located at positions given by the bit paths corresponding to numbers 1 to n. From Lemma 15 we conclude that the elements contained in an alternate heap of cardinal n are located at positions given by the reverse bit paths corresponding to numbers 1 to n. By Theorem 17, function *shuffle* reverses the paths of the elements of a tree and preserves the cardinal, so it transforms a classic heap into an alternate one. As the reverse function is idempotent, it transforms an alternate heap into a classic one.

□