DEMO : Purchase from www.A-PDF.com to remove the watermark

7

Binomial, Fibonacci, and Pairing Heaps

Michael L. Fredman
Rutgers University at New Brunswick

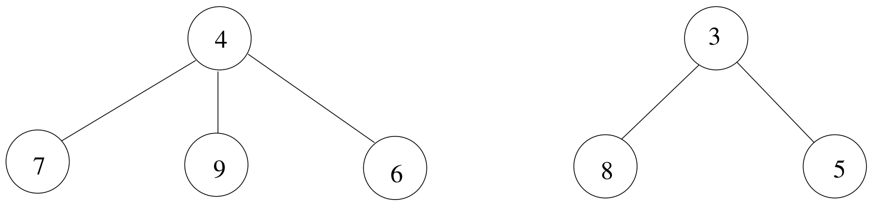
7.1 Introduction

This chapter presents three algorithmically related data structures for implementing meldable priority queues: binomial heaps, Fibonacci heaps, and pairing heaps. What these three structures have in common is that (a) they are comprised of heap-ordered trees, (b) the comparisons performed to execute extractmin operations exclusively involve keys stored in the roots of trees, and (c) a common side effect of a comparison between two root keys is the linking of the respective roots: one tree becomes a new subtree joined to the other root.

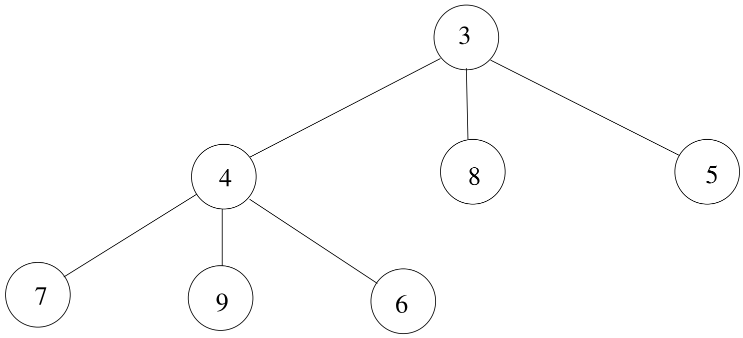
A tree is considered heap-ordered provided that each node contains one item, and the key of the item stored in the parent $p(x)$ of a node x never exceeds the key of the item stored in x. Thus, when two roots get linked, the root storing the larger key becomes a child of the other root. By convention, a linking operation positions the new child of a node as its leftmost child. Figure 7.1 illustrates these notions.

Of the three data structures, the binomial heap structure was the first to be invented (Vuillemin [13]), designed to efficiently support the operations insert, extractmin, delete, and meld. The binomial heap has been highly appreciated as an elegant and conceptually simple data structure, particularly given its ability to support the meld operation. The Fibonacci heap data structure (Fredman and Tarjan [6]) was inspired by and can be viewed as a generalization of the binomial heap structure. The raison d'être of the Fibonacci heap structure is its ability to efficiently execute decrease-key operations. A decrease-key operation replaces the key of an item, specified by location, by a smaller value: e.g. decrease-key(P,k_{new},H). (The arguments specify that the item is located in node P of the priority queue H, and that its new key value is k_{new}.) Decrease-key operations are prevalent in many network optimization algorithms, including minimum spanning tree, and shortest path. The pairing heap data structure (Fredman, Sedgewick, Sleator, and Tarjan [5]) was

© 2005 by Chapman & Hall/CRC



(a) before linking.



(b) after linking.

FIGURE 7.1: Two heap-ordered trees and the result of their linking.

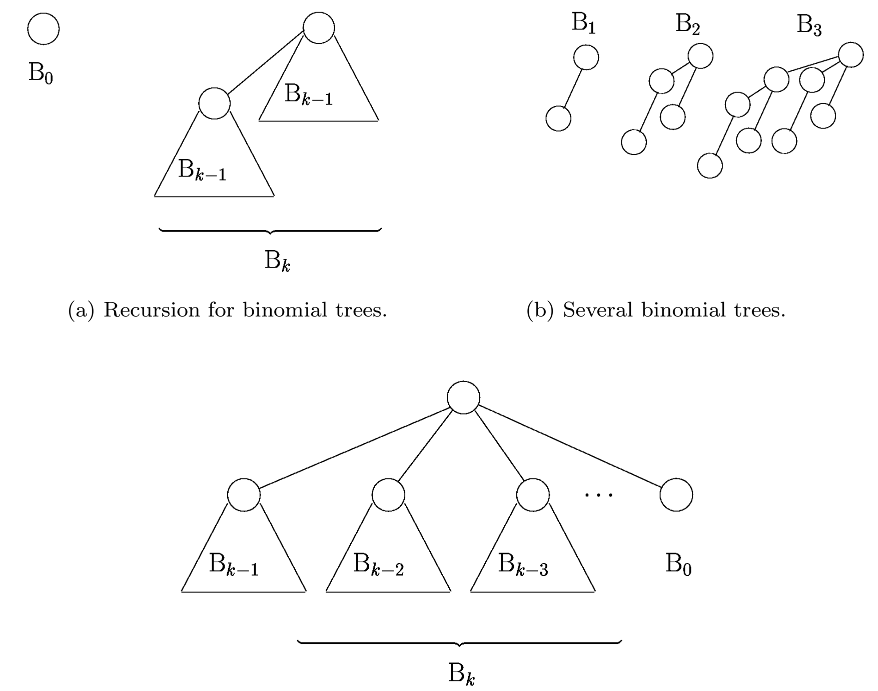
devised as a self-adjusting analogue of the Fibonacci heap, and has proved to be more efficient in practice [11].

Binomial heaps and Fibonacci heaps are primarily of theoretical and historical interest. The pairing heap is the more efficient and versatile data structure from a practical standpoint. The following three sections describe the respective data structures. Summaries of the various algorithms in the form of pseudocode are provided in section 7.5.

7.2 Binomial Heaps

We begin with an informal overview. A single binomial heap structure consists of a forest of specially structured trees, referred to as binomial trees. The number of nodes in a binomial tree is always a power of two. Defined recursively, the binomial tree B_0 consists of a single node. The binomial tree B_k, for $k > 0$, is obtained by linking two trees B_{k-1} together; one tree becomes the leftmost subtree of the other. In general B_k has 2^k nodes. Figures 7.2(a-b) illustrate the recursion and show several trees in the series. An alternative and useful way to view the structure of B_k is depicted in Figure 7.2(c): B_k consists of a root and subtrees (in order from left to right) $B_{k-1}, B_{k-2}, \cdots, B_0$. The root of the binomial tree B_k has k children, and the tree is said to have rank k. We also observe that the height of B_k (maximum number of edges on any path directed away from the root) is k. The name "binomial heap" is inspired by the fact that the root of B_k has $\binom{k}{j}$ descendants at distance j.

© 2005 by Chapman & Hall/CRC



(a) Recursion for binomial trees.

(b) Several binomial trees.

(c) An alternative recursion.

FIGURE 7.2: Binomial trees and their recursions.

Because binomial trees have restricted sizes, a forest of trees is required to represent a priority queue of arbitrary size. A key observation, indeed a motivation for having tree sizes being powers of two, is that a priority queue of arbitrary size can be represented as a union of trees of *distinct* sizes. (In fact, the sizes of the constituent trees are uniquely determined and correspond to the powers of two that define the binary expansion of n, the size of the priority queue.) Moreover, because the tree sizes are unique, the number of trees in the forest of a priority queue of size n is at most $\lg(n+1)$. Thus, finding the minimum key in the priority queue, which clearly lies in the root of one of its constituent trees (due to the heap-order condition), requires searching among at most $\lg(n+1)$ tree roots. Figure 7.3 gives an example of binomial heap.

Now let's consider, from a high-level perspective, how the various heap operations are performed. As with leftist heaps (cf. Chapter 6), the various priority queue operations are to a large extent comprised of melding operations, and so we consider first the melding of two heaps.

The melding of two heaps proceeds as follows: (a) the trees of the respective forests are combined into a single forest, and then (b) *consolidation* takes place: pairs of trees having common rank are linked together until all remaining trees have distinct ranks. Figure 7.4 illustrates the process. An actual implementation mimics binary addition and proceeds in much the same was as merging two sorted lists in ascending order. We note that insertion is a special case of melding.

© 2005 by Chapman & Hall/CRC

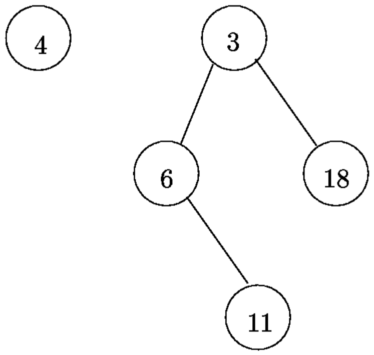
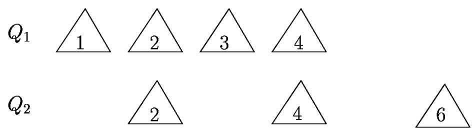
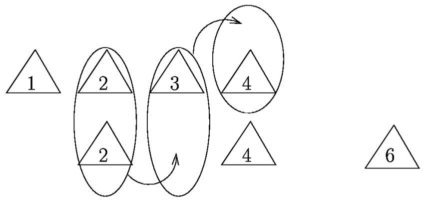


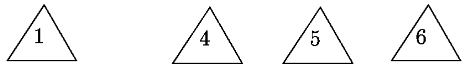
FIGURE 7.3: A binomial heap (showing placement of keys among forest nodes).



(a) Forests of two heaps Q_1 and Q_2 to be melded.



(b) Linkings among trees in the combined forest.



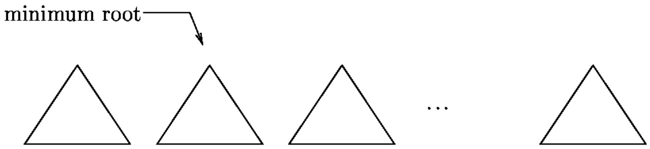
(c) Forest of meld(Q_1,Q_2).

FIGURE 7.4: Melding of two binomial heaps. The encircled objects reflect trees of common rank being linked. (Ranks are shown as numerals positioned within triangles which in turn represent individual trees.) Once linking takes place, the resulting tree becomes eligible for participation in further linkings, as indicated by the arrows that identify these linking results with participants of other linkings.

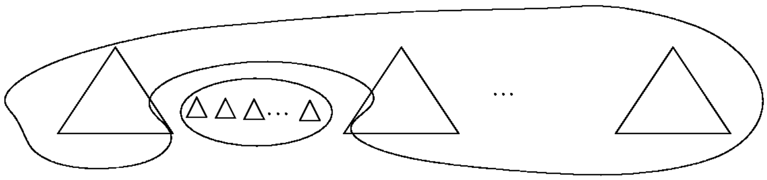
The extractmin operation is performed in two stages. First, the *minimum root*, the node containing the minimum key in the data structure, is found by examining the tree roots of the appropriate forest, and this node is removed. Next, the forest consisting of the subtrees of this removed root, whose ranks are distinct (see Figure 7.2(c)) and thus viewable as

© 2005 by Chapman & Hall/CRC

constituting a binomial heap, is melded with the forest consisting of the trees that remain from the original forest. Figure 7.5 illustrates the process.



(a) Initial forest.



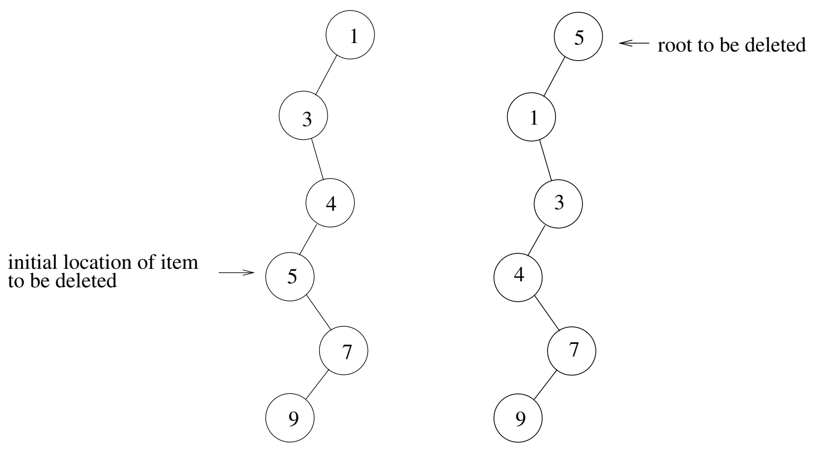
(b) Forests to be melded.

FIGURE 7.5: Extractmin Operation: The location of the minimum key is indicated in (a). The two encircled sets of trees shown in (b) represent forests to be melded. The smaller trees were initially subtrees of the root of the tree referenced in (a).

Finally, we consider arbitrary deletion. We assume that the node ν containing the item to be deleted is specified. Proceeding up the path to the root of the tree containing ν, we permute the items among the nodes on this path, placing in the root the item x originally in ν, and shifting each of the other items down one position (away from the root) along the path. This is accomplished through a sequence of exchange operations that move x towards the root. The process is referred to as a *sift-up* operation. Upon reaching the root r, r is then removed from the forest as though an extractmin operation is underway. Observe that the re-positioning of items in the ancestors of ν serves to maintain the heap-order property among the remaining nodes of the forest. Figure 7.6 illustrates the re-positioning of the item being deleted to the root.

This completes our high-level descriptions of the heap operations. For navigational purposes, each node contains a leftmost child pointer and a sibling pointer that points to the next sibling to its right. The children of a node are thus stored in the linked list defined by sibling pointers among these children, and the head of this list can be accessed by the leftmost child pointer of the parent. This provides the required access to the children of

© 2005 by Chapman & Hall/CRC



(a) initial item placement. (b) after movement to root.

FIGURE 7.6: Initial phase of deletion – sift-up operation.

a node for the purpose of implementing extractmin operations. Note that when a node obtains a new child as a consequence of a linking operation, the new child is positioned at the head of its list of siblings. To facilitate arbitrary deletions, we need a third pointer in each node pointing to its parent. To facilitate access to the ranks of trees, we maintain in each node the number of children it has, and refer to this quantity as the node rank. Node ranks are readily maintained under linking operations; the node rank of the root gaining a child gets incremented. Figure 7.7 depicts these structural features.

As seen in Figure 7.2(c), the ranks of the children of a node form a descending sequence in the children's linked list. However, since the melding operation is implemented by accessing the tree roots in ascending rank order, when deleting a root we first reverse the list order of its children before proceeding with the melding.

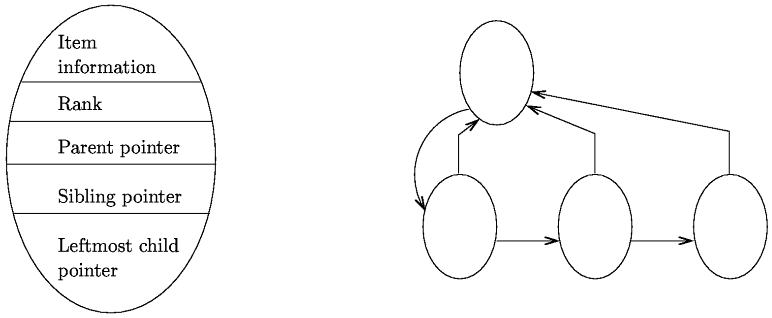
Each of the priority queue operations requires in the worst case $O(\log n)$ time, where n is the size of the heap that results from the operation. This follows, for melding, from the fact that its execution time is proportional to the combined lengths of the forest lists being merged. For extractmin, this follows from the time for melding, along with the fact that a root node has only $O(\log n)$ children. For arbitrary deletion, the time required for the sift-up operation is bounded by an amount proportional to the height of the tree containing the item. Including the time required for extractmin, it follows that the time required for arbitrary deletion is $O(\log n)$.

Detailed code for manipulating binomial heaps can be found in Weiss [14].

7.3 Fibonacci Heaps

Fibonacci heaps were specifically designed to efficiently support decrease-key operations. For this purpose, the binomial heap can be regarded as a natural starting point. Why? Consider the class of priority queue data structures that are implemented as forests of heap-ordered trees, as will be the case for Fibonacci heaps. One way to immediately execute a

© 2005 by Chapman & Hall/CRC



(a) fields of a node.

(b) a node and its three children.

FIGURE 7.7: Structure associated with a binomial heap node. Figure (b) illustrates the positioning of pointers among a node and its three children.

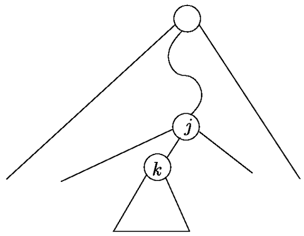
decrease-key operation, remaining within the framework of heap-ordered forests, is to simply change the key of the specified data item and sever its link to its parent, inserting the severed subtree as a new tree in the forest. Figure 7.8 illustrates the process. (Observe that the link to the parent only needs to be cut if the new key value is smaller than the key in the parent node, violating heap-order.) Fibonacci heaps accomplish this without degrading the asymptotic efficiency with which other priority queue operations can be supported. Observe that to accommodate node cuts, the list of children of a node needs to be doubly linked. Hence the nodes of a Fibonacci heap require two sibling pointers.

Fibonacci heaps support findmin, insertion, meld, and decrease-key operations in constant amortized time, and deletion operations in $O(\log n)$ amortized time. For many applications, the distinction between worst-case times versus amortized times are of little significance. A Fibonacci heap consists of a forest of heap-ordered trees. As we shall see, Fibonacci heaps differ from binomial heaps in that there may be many trees in a forest of the same rank, and there is no constraint on the ordering of the trees in the forest list. The heap also includes a pointer to the tree root containing the minimum item, referred to as the *min-pointer*, that facilitates findmin operations. Figure 7.9 provides an example of a Fibonacci heap and illustrates certain structural aspects.

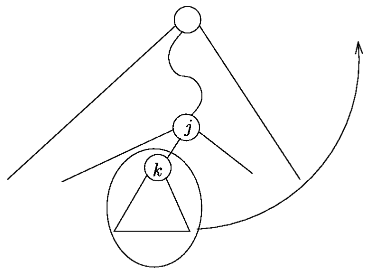
The impact of severing subtrees is clearly incompatible with the pristine structure of the binomial tree that is the hallmark of the binomial heap. Nevertheless, the tree structures that can appear in the Fibonacci heap data structure must sufficiently approximate binomial trees in order to satisfy the performance bounds we seek. The linking constraint imposed by binomial heaps, that trees being linked must have the same size, ensures that the number of children a node has (its rank), grows no faster than the logarithm of the size of the subtree rooted at the node. This *rank versus subtree size* relation is key to obtaining the $O(\log n)$ deletion time bound. Fibonacci heap manipulations are designed with this in mind.

Fibonacci heaps utilize a protocol referred to as *cascading cuts* to enforce the required rank versus subtree size relation. Once a node ν has had two of its children removed as a result of cuts, ν's contribution to the rank of its parent is then considered suspect in terms of rank versus subtree size. The cascading cut protocol requires that the link to ν's parent

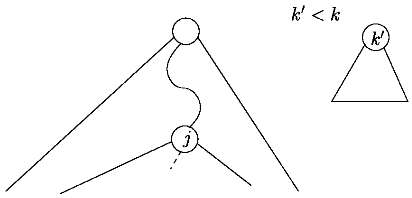
© 2005 by Chapman & Hall/CRC



(a) Initial tree.



(b) Subtree to be severed.



(c) Resulting changes

FIGURE 7.8: Immediate decrease-key operation. The subtree severing (Figures (b) and (c)) is necessary only when $k' < j$.

be cut, with the subtree rooted at ν then being inserted into the forest as a new tree. If ν's parent has, as a result, had a second child removed, then it in turn needs to be cut, and the cuts may thus cascade. Cascading cuts ensure that no non-root node has had more than one child removed subsequent to being linked to its parent.

We keep track of the removal of children by marking a node if one of its children has been cut. A marked node that has another child removed is then subject to being cut from its parent. When a marked node becomes linked as a child to another node, or when it gets cut from its parent, it gets unmarked. Figure 7.10 illustrates the protocol of cascading cuts.

Now the induced node cuts under the cascading cuts protocol, in contrast with those primary cuts immediately triggered by decrease-key operations, are bounded in number by the number of primary cuts. (This follows from consideration of a potential function defined to be the total number of marked nodes.) Therefore, the burden imposed by cascading cuts can be viewed as effectively only doubling the number of cuts taking place in the absence of the protocol. One can therefore expect that the performance asymptotics are not degraded as a consequence of proliferating cuts. As with binomial heaps, two trees in a Fibonacci heap can only be linked if they have equal rank. With the cascading cuts protocol in place,

© 2005 by Chapman & Hall/CRC

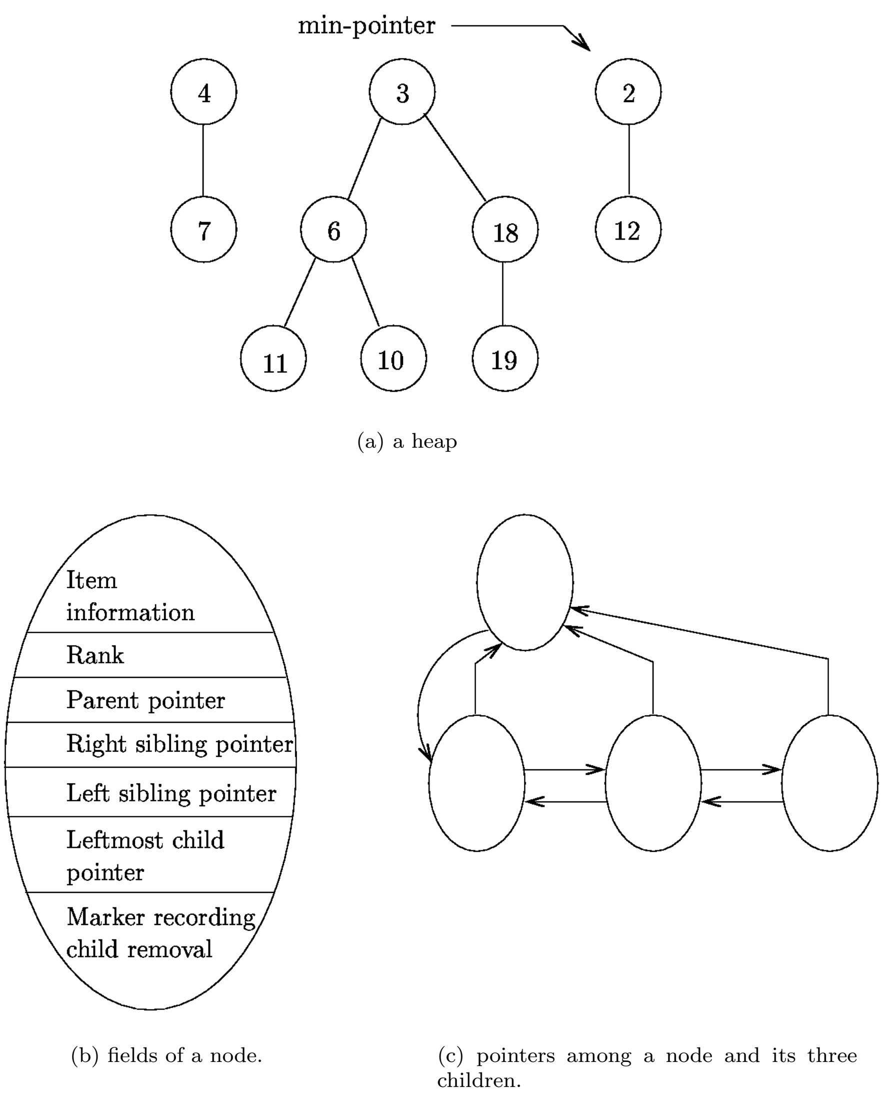
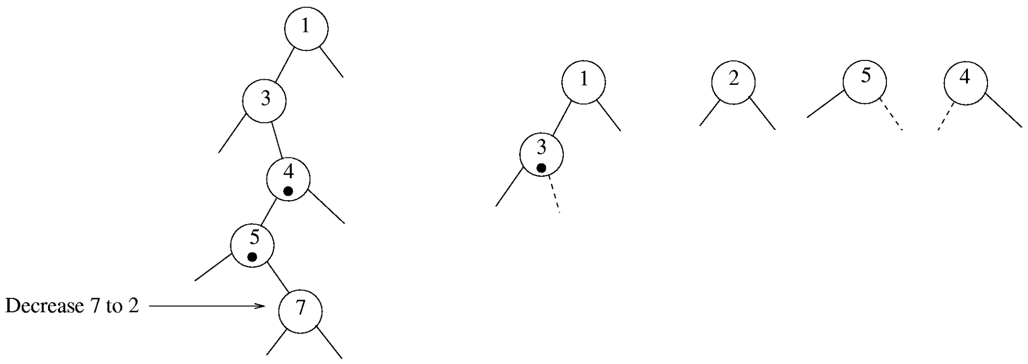


FIGURE 7.9: A Fibonacci heap and associated structure.

we claim that the required rank versus subtree size relation holds, a matter which we address next.

Let's consider how *small* the subtree rooted at a node ν having rank k can be. Let ω be the mth child of ν from the right. At the time it was linked to ν, ν had at least $m-1$ other children (those currently to the right of ω were certainly present). Therefore ω had rank at least $m-1$ when it was linked to ν. Under the cascading cuts protocol, the rank of ω could have decreased by at most one after its linking to ν; otherwise it would have been removed as a child. Therefore, the current rank of ω is at least $m-2$. We minimize the size of the subtree rooted at ν by minimizing the sizes (and ranks) of the subtrees rooted at

© 2005 by Chapman & Hall/CRC



(a) Before decrease-key. (b) After decrease-key.

FIGURE 7.10: Illustration of cascading cuts. In (b) the dashed lines reflect cuts that have taken place, two nodes marked in (a) get unmarked, and a third node gets marked.

ν's children. Now let F_j denote the minimum possible size of the subtree rooted at a node of rank j, so that the size of the subtree rooted at ν is F_k. We conclude that (for $k \geq 2$)

$$F_k = \underbrace{F_{k-2} + F_{k-3} + \cdots + F_0 + 1}_{k \text{ terms}} + 1$$

where the final term, 1, reflects the contribution of ν to the subtree size. Clearly, $F_0 = 1$ and $F_1 = 2$. See Figure 7.11 for an illustration of this construction. Based on the preceding recurrence, it is readily shown that F_k is given by the $(k + 2)$th Fibonacci number (from whence the name "Fibonacci heap" was inspired). Moreover, since the Fibonacci numbers grow exponentially fast, we conclude that the rank of a node is indeed bounded by the logarithm of the size of the subtree rooted at the node.

We proceed next to describe how the various operations are performed.

Since we are not seeking worst-case bounds, there are economies to be exploited that could also be applied to obtain a variant of Binomial heaps. (In the absence of cuts, the individual trees generated by Fibonacci heap manipulations would all be binomial trees.) In particular we shall adopt a lazy approach to melding operations: the respective forests are simply combined by concatenating their tree lists and retaining the appropriate min-pointer. This requires only constant time.

An item is deleted from a Fibonacci heap by deleting the node that originally contains it, in contrast with Binomial heaps. This is accomplished by (a) cutting the link to the node's parent (as in decrease-key) if the node is not a tree root, and (b) appending the list of children of the node to the forest. Now if the deleted node happens to be referenced by the min-pointer, considerable work is required to restore the min-pointer – the work previously deferred by the lazy approach to the operations. In the course of searching among the roots

© 2005 by Chapman & Hall/CRC

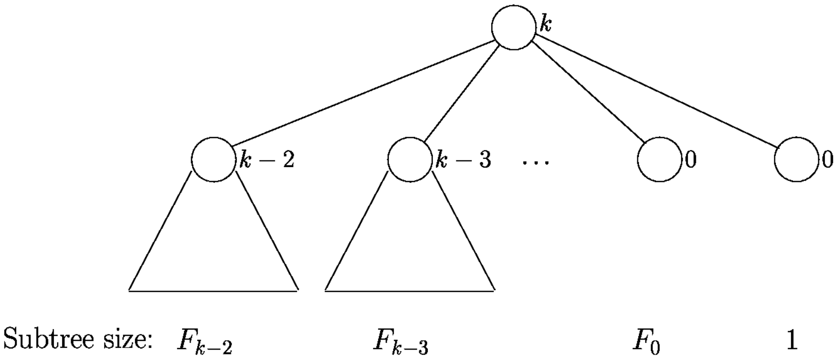


FIGURE 7.11: Minimal tree of rank k. Node ranks are shown adjacent to each node.

of the forest to discover the new minimum key, we also link trees together in a *consolidation* process.

Consolidation processes the trees in the forest, linking them in pairs until there are no longer two trees having the same rank, and then places the remaining trees in a new forest list (naturally extending the melding process employed by binomial heaps). This can be accomplished in time proportional to the number of trees in forest plus the maximum possible node rank. Let max-rank denote the maximum possible node rank. (The preceding discussion implies that max-rank = $O(\log \text{heap-size})$.) Consolidation is initialized by setting up an array A of trees (initially empty) indexed by the range [0,max-rank]. A non-empty position A[d] of A contains a tree of rank d. The trees of the forest are then processed using the array A as follows. To process a tree T of rank d, we insert T into A[d] if this position of A is empty, completing the processing of T. However, if A[d] already contains a tree U, then T and U are linked together to form a tree W, and the processing continues as before, but with W in place of T, until eventually an empty location of A is accessed, completing the processing associated with T. After all of the trees have been processed in this manner, the array A is scanned, placing each of its stored trees in a new forest. Apart from the final scanning step, the total time necessary to consolidate a forest is proportional to its number of trees, since the total number of tree pairings that can take place is bounded by this number (each pairing reduces by one the total number of trees present). The time required for the final scanning step is given by max-rank = log(heap-size).

The amortized timing analysis of Fibonacci heaps considers a potential function defined as the total number of trees in the forests of the various heaps being maintained. Ignoring consolidation, each operation takes constant actual time, apart from an amount of time proportional to the number of subtree cuts due to cascading (which, as noted above, is only constant in amortized terms). These cuts also contribute to the potential. The children of a deleted node increase the potential by $O(\log \text{heap-size})$. Deletion of a minimum heap node additionally incurs the cost of consolidation. However, consolidation reduces our potential, so that the amortized time it requires is only $O(\log \text{heap-size})$. We conclude therefore that all non-deletion operations require constant amortized time, and deletion requires $O(\log n)$ amortized time.

An interesting and unresolved issue concerns the protocol of cascading cuts. How would the performance of Fibonacci heaps be affected by the absence of this protocol?

Detailed code for manipulating Fibonacci heaps can found in Knuth [9].

© 2005 by Chapman & Hall/CRC

7.4 Pairing Heaps

The pairing heap was designed to be a self-adjusting analogue of the Fibonacci heap, in much the same way that the skew heap is a self-adjusting analogue of the leftist heap (See Chapters 5 and 6). The only structure maintained in a pairing heap node, besides item information, consists of three pointers: leftmost child, and two sibling pointers. (The leftmost child of a node uses it left sibling pointer to point to its parent, to facilitate updating the leftmost child pointer its parent.) See Figure 7.12 for an illustration of pointer structure.

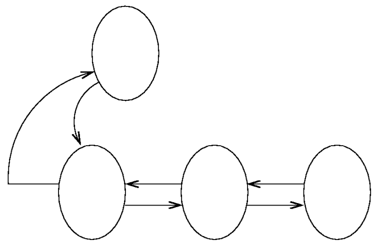


FIGURE 7.12: Pointers among a pairing heap node and its three children.

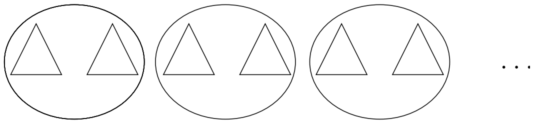
The are no cascading cuts – only simple cuts for decrease-key and deletion operations. With the absence of parent pointers, decrease-key operations uniformly require a single cut (removal from the sibling list, in actuality), as there is no efficient way to check whether heap-order would otherwise be violated. Although there are several varieties of pairing heaps, our discussion presents the two-pass version (the simplest), for which a given heap consists of only a single tree. The minimum element is thus uniquely located, and melding requires only a single linking operation. Similarly, a decrease-key operation consists of a subtree cut followed by a linking operation. Extractmin is implemented by removing the tree root and then linking the root's subtrees in a manner described below. Other deletions involve (a) a subtree cut, (b) an extractmin operation on the cut subtree, and (c) linking the remnant of the cut subtree with the original root.

The extractmin operation combines the subtrees of the root using a process referred to as *two-pass pairing*. Let $x_1, \cdots, x_k$ be the subtrees of the root in left-to-right order. The first pass begins by linking x_1 and x_2. Then x_3 and x_4 are linked, followed by x_5 and x_6, etc., so that the odd positioned trees are linked with neighboring even positioned trees. Let $y_1, \cdots, y_h$, $h = \lceil k/2 \rceil$, be the resulting trees, respecting left-to-right order. (If k is odd, then $y_{\lceil k/2 \rceil}$ is x_k.) The second pass reduces these to a single tree with linkings that proceed from right-to-left. The rightmost pair of trees, y_h and y_{h-1} are linked first, followed by the linking of y_{h-2} with the result of the preceding linking etc., until finally we link y_1 with the structure formed from the linkings of $y_2, \cdots, y_h$. See Figure 7.13.

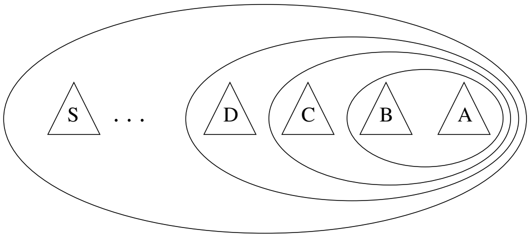
Since two-pass pairing is not particularly intuitive, a few motivating remarks are offered. The first pass is natural enough, and one might consider simply repeating the process on the remaining trees, until, after logarithmically many such passes, only one tree remains. Indeed, this is known as the multi-pass variation. Unfortunately, its behavior is less understood than that of the two-pass pairing variation.

The second (right-to-left) pass is also quite natural. Let H be a binomial heap with exactly 2^k items, so that it consists of a single tree. Now suppose that an extractmin followed by

© 2005 by Chapman & Hall/CRC



(a) first pass.



(b) second pass.

FIGURE 7.13: Two-pass Pairing. The encircled trees get linked. For example, in (b) trees A and B get linked, and the result then gets linked with the tree C, etc.

an insertion operation are executed. The linkings that take place among the subtrees of the deleted root (after the new node is linked with the rightmost of these subtrees) entail the right-to-left processing that characterizes the second pass. So why not simply rely upon a single right-to-left pass, and omit the first? The reason, is that although the second pass preserves existing balance within the structure, it doesn't improve upon poorly balanced situations (manifested when most linkings take place between trees of disparate sizes). For example, using a single-right-to-left-pass version of a pairing heap to sort an increasing sequence of length n (n insertions followed by n extractmin operations), would result in an n^2 sorting algorithm. (Each of the extractmin operations yields a tree of height 1 or less.) See Section 7.6, however, for an interesting twist.

In actuality two-pass pairing was inspired [5] by consideration of splay trees (Chapter 12). If we consider the child, sibling representation that maps a forest of arbitrary trees into a binary tree, then two-pass pairing can be viewed as a splay operation on a search tree path with no bends [5]. The analysis for splay trees then carries over to provide an amortized analysis for pairing heaps.

The asymptotic behavior of pairing heaps is an interesting and unresolved matter. Reflecting upon the tree structures we have encountered in this chapter, if we view the binomial trees that comprise binomial heaps, their structure highly constrained, as likened to perfectly spherical masses of discrete diameter, then the trees that comprise Fibonacci heaps can be viewed as rather rounded masses, but rarely spherical, and of arbitrary (non-discrete) size. Applying this imagery to the trees that arise from pairing heap manipulations, we can aptly liken these trees to chunks of clay subject to repeated tearing and compaction, typically irregular in form. It is not obvious, therefore, that pairing heaps should be asymptotically efficient. On the other hand, since the pairing heap design dispenses with the rather complicated, carefully crafted constructs put in place primarily to facilitate proving the time bounds enjoyed by Fibonacci heaps, we can expect efficiency gains at the level of elemen-

© 2005 by Chapman & Hall/CRC

tary steps such as linking operations. From a practical standpoint the data structure is a success, as seen from the study of Moret and Shapiro [11]. Also, for those applications for which decrease-key operations are highly predominant, pairing heaps provably meet the optimal asymptotic bounds characteristic of Fibonacci heaps [3]. But despite this, as well as empirical evidence consistent with optimal efficiency in general, pairing heaps are in fact asymptotically sub-optimal for certain operation sequences [3]. Although decrease-key requires only constant worst-case time, its execution can asymptotically degrade the efficiency of extractmin operations, even though the effect is not observable in practice. On the positive side, it has been demonstrated [5] that under all circumstances the operations require only $O(\log n)$ amortized time. Additionally, Iacono [7] has shown that insertions require only constant amortized time; significant for those applications that entail many more insertions than deletions.

The reader may wonder whether some alternative to two-pass pairing might provably attain the asymptotic performance bounds satisfied by Fibonacci heaps. However, for information-theoretic reasons *no* such alternative exists. (In fact, this is how we know the two-pass version is sub-optimal.) A precise statement and proof of this result appears in Fredman [3].

Detailed code for manipulating pairing heaps can be found in Weiss [14].

7.5 Pseudocode Summaries of the Algorithms

This section provides pseudocode reflecting the above algorithm descriptions. The procedures, link and insert, are sufficiently common with respect to all three data structures, that we present them first, and then turn to those procedures having implementations specific to a particular data structure.

7.5.1 Link and Insertion Algorithms

```
Function link(x,y){
   // x and y are tree roots. The operation makes the root with the
   // larger key the leftmost child of the other root. For binomial and
   // Fibonacci heaps, the rank field of the prevailing root is
   // incremented.  Also, for Fibonacci heaps, the node becoming the child
   // gets unmarked if it happens to be originally marked.  The function
   // returns a pointer to the node x or y that becomes the root.
}

Algorithm Insert(x,H){
   //Inserts into heap H the item x
   I = Makeheap(x)
      // Creates a single item heap I containing the item x.
   H = Meld(H,I).
}
```

© 2005 by Chapman & Hall/CRC

7.5.2 Binomial Heap-Specific Algorithms

```
Function Meld(H,I){
   // The forest lists of H and I are combined and consolidated -- trees
   // having common rank are linked together until only trees of distinct
   // ranks remain.  (As described above, the process resembles binary
   // addition.)  A pointer to the resulting list is returned.  The
   // original lists are no longer available.
}

Function Extractmin(H){
   //Returns the item containing the minimum key in the heap H.
   //The root node r containing this item is removed from H.
   r = find-minimum-root(H)
   if(r = null){return "Empty"}
   else{
      x = item in r
      H = remove(H,r)
         // removes the tree rooted at r from the forest of H
      I = reverse(list of children of r)
      H = Meld(H,I)
      return x
   }
}

Algorithm Delete(x,H)
   //Removes from heap H the item in the node referenced by x.
   r = sift-up(x)
      // r is the root of the tree containing x.  As described above,
      // sift-up moves the item information in x to r.
   H = remove(H,r)
      // removes the tree rooted at r from the forest of H
   I = reverse(list of children of r)
   H = Meld(H,I)
}
```

7.5.3 Fibonacci Heap-Specific Algorithms

```
Function Findmin(H){
   //Return the item in the node referenced by the min-pointer of H
   //(or "Empty" if applicable)
}

Function Meld(H,I){
   // The forest lists of H and I are concatenated.  The keys referenced
   // by the respective min-pointers of H and I are compared, and the
   // min-pointer referencing the larger key is discarded.  The concatenation
   // result and the retained min-pointer are returned.  The original
   // forest lists of H and I are no longer available.
}
```

© 2005 by Chapman & Hall/CRC

```
Algorithm Cascade-Cut(x,H){
   //Used in decrease-key and deletion.  Assumes parent(x) != null
   y = parent(x)
   cut(x,H)
      // The subtree rooted at x is removed from parent(x) and inserted into
      // the forest list of H.  The mark-field of x is set to FALSE, and the
      // rank of parent(x) is decremented.
   x = y
   while(x is marked and parent(x) != null){
       y = parent(x)
       cut(x,H)
       x = y
   }
   Set mark-field of x = TRUE
}

Algorithm Decrease-key(x,k,H){
   key(x) = k
   if(key of min-pointer(H) > k){ min-pointer(H) = x}
   if(parent(x) != null and key(parent(x)) > k){ Cascade-Cut(x,H)}
}

Algorithm Delete(x,H){
   If(parent(x) != null){
      Cascade-Cut(x,H)
      forest-list(H) = concatenate(forest-list(H), leftmost-child(x))
      H = remove(H,x)
         // removes the (single node) tree rooted at x from the forest of H
   }
   else{
      forest-list(H) = concatenate(forest-list(H), leftmost-child(x))
      H = remove(H,x)
      if(min-pointer(H) = x){
         consolidate(H)
            // trees of common rank in the forest list of H are linked
            // together until only trees having distinct ranks remain.  The
            // remaining trees then constitute the forest list of H.
            // min-pointer is reset to reference the root with minimum key.
       }
   }
}
```

7.5.4 Pairing Heap-Specific Algorithms

```
Function Findmin(H){
   // Returns the item in the node referenced by H (or "empty" if applicable)
}

Function Meld(H,I){
   return link(H,I)
}
```

© 2005 by Chapman & Hall/CRC

```
Function Decrease-key(x,k,H){
   If(x != H){
      Cut(x)
         // The node x is removed from the child list in which it appears
      key(x) = k
      H = link(H,x)
   }
   else{ key(H) = k}
}

Function Two-Pass-Pairing(x){
   // x is assumed to be a pointer to the first node of a list of tree
   // roots.  The function executes two-pass pairing to combine the trees
   // into a single tree as described above, and returns a pointer to
   // the root of the resulting tree.
}

Algorithm Delete(x,H){
   y = Two-Pass-Pairing(leftmost-child(x))
   if(x = H){ H = y}
   else{
      Cut(x)
         // The subtree rooted at x is removed from its list of siblings.
      H = link(H,y)
   }
}
```

7.6 Related Developments

In this section we describe some results pertinent to the data structures of this chapter. First, we discuss a variation of the pairing heap, referred to as the skew-pairing heap. The skew-pairing heap appears as a form of "missing link" in the landscape occupied by pairing heaps and skew heaps (Chapter 6). Second, we discuss some adaptive properties of pairing heaps. Finally, we take note of soft heaps, a new shoot of activity emanating from the primordial binomial heap structure that has given rise to the topics of this chapter.

Skew-Pairing Heaps

There is a curious variation of the pairing heap which we refer to as a *skew-pairing* heap – the name will become clear. Aside from the linking process used for combining subtrees in the extractmin operation, skew-pairing heaps are identical to two-pass pairing heaps. The skew-pairing heap extractmin linking process places greater emphasis on right-to-left linking than does the pairing heap, and proceeds as follows.

First, a right-to-left linking of the subtrees that fall in odd numbered positions is executed. Let H_{odd} denote the result. Similarly, the subtrees in even numbered positions are linked in right-to-left order. Let H_{even} denote the result. Finally, we link the two trees, H_{odd} and H_{even}. Figure 7.14 illustrates the process.

The skew-pairing heap enjoys $O(\log n)$ time bounds for the usual operations. Moreover, it has the following curious relationship to the skew heap. Suppose a finite sequence S of

© 2005 by Chapman & Hall/CRC

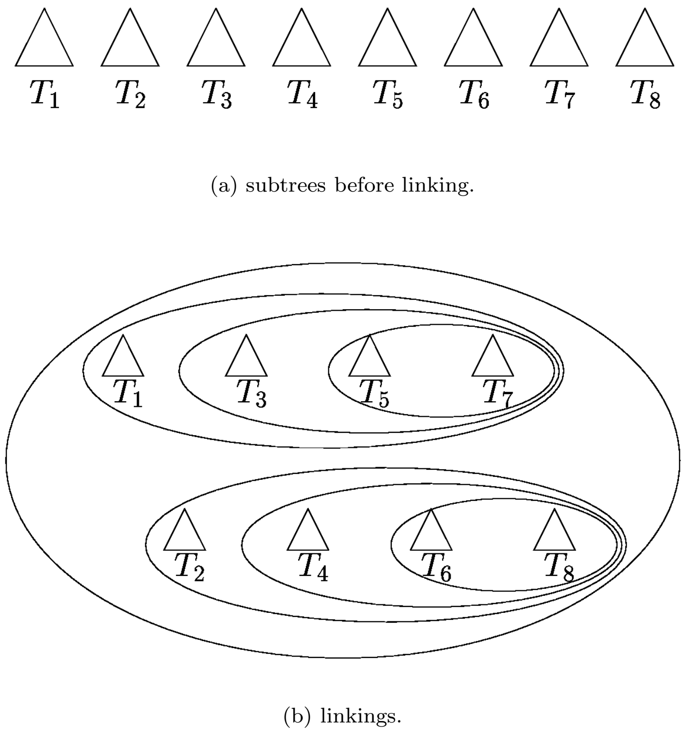


FIGURE 7.14: Skew-pairing heap: linking of subtrees performed by extractmin. As described in Figure 7.13, encircled trees become linked.

meld and extractmin operations is executed (beginning with heaps of size 1) using (a) a skew heap and (b) a skew-pairing heap. Let C_s and C_{s-p} be the respective sets of comparisons between keys that actually get performed in the course of the respective executions (ignoring the order of the comparison executions). Then $C_{s-p} \subset C_s$ [4]. Moreover, if the sequence S terminates with the heap empty, then $C_{s-p} = C_s$. (This inspires the name "skew-pairing".) The relationship between skew-pairing heaps and splay trees is also interesting. The child, sibling transformation, which for two-pass pairing heaps transforms the extractmin operation into a splay operation on a search tree path having no bends, when applied to the skew-pairing heap, transforms extractmin into a splay operation on a search tree path having a bend at each node. Thus, skew-pairing heaps and two-pass pairing heaps demarcate opposite ends of a spectrum.

Adaptive Properties of Pairing Heaps

Consider the problem of merging k sorted lists of respective lengths $n_1, n_2, \cdots, n_k$, with $\sum n_i = n$. The standard merging strategy that performs $\lg k$ rounds of pairwise list merges requires $n \lg k$ time. However, a merge pattern based upon the binary Huffman tree, having minimal external path length for the weights $n_1, n_2, \cdots, n_k$, is more efficient when the lengths n_i are non-uniform, and provides a near optimal solution. Pairing heaps can be utilized to provide a rather different solution as follows. Treat each sorted list as a

© 2005 by Chapman & Hall/CRC

linearly structured pairing heap. Then (a) meld these k heaps together, and (b) repeatedly execute extractmin operations to retrieve the n items in their sorted order. The number of comparisons that take place is bounded by

$$O(\log \binom{n}{n_1, \cdots, n_k})$$

Since the above multinomial coefficient represents the number of possible merge patterns, the information-theoretic bound implies that this result is optimal to within a constant factor. The pairing heap thus self-organizes the sorted list arrangement to approximate an optimal merge pattern. Iacono has derived a "working-set" theorem that quantifies a similar adaptive property satisfied by pairing heaps. Given a sequence of insertion and extractmin operations initiated with an empty heap, at the time a given item x is deleted we can attribute to x a contribution bounded by $O(\log \mathrm{op}(x))$ to the total running time of the sequence, where $\mathrm{op}(x)$ is the number of heap operations that have taken place since x was inserted (see [8] for a slightly tighter estimate). Iacono has also shown that this same bound applies for skew and skew-pairing heaps [8]. Knuth [10] has observed, at least in qualitative terms, similar behavior for leftist heaps . Quoting Knuth:

> Leftist trees are in fact already obsolete, except for applications with a strong tendency towards last-in-first-out behavior.

Soft Heaps

An interesting development (Chazelle [1]) that builds upon and extends binomial heaps in a different direction is a data structure referred to as a *soft heap*. The soft heap departs from the standard notion of priority queue by allowing for a type of error, referred to as *corruption*, which confers enhanced efficiency. When an item becomes corrupted, its key value gets increased. Findmin returns the minimum current key, which might or might not be corrupted. The user has no control over which items become corrupted, this prerogative belonging to the data structure. But the user does control the overall amount of corruption in the following sense.

The user specifies a parameter, $0 < \epsilon \leq 1/2$, referred to as the error rate, that governs the behavior of the data structure as follows. The operations findmin and deletion are supported in constant amortized time, and insertion is supported in $O(\log 1/\epsilon)$ amortized time. Moreover, no more than an ϵ fraction of the items present in the heap are corrupted at any given time.

To illustrate the concept, let x be an item returned by findmin, from a soft heap of size n. Then there are no more than ϵn items in the heap whose original keys are less than the original key of x.

Soft heaps are rather subtle, and we won't attempt to discuss specifics of their design. Soft heaps have been used to construct an optimal comparison-based minimum spanning tree algorithm (Pettie and Ramachandran [12]), although its actual running time has not been determined. Soft heaps have also been used to construct a comparison-based algorithm with known running time $m\alpha(m, n)$ on a graph with n vertices and m edges (Chazelle [2]), where $\alpha(m, n)$ is a functional inverse of the Ackermann function. Chazelle [1] has also observed that soft heaps can be used to implement median selection in linear time; a significant departure from previous methods.

© 2005 by Chapman & Hall/CRC

References

[1] B. Chazelle, The Soft Heap: An Approximate Priority Queue with Optimal Error Rate, *Journal of the ACM*, 47 (2000), 1012–1027.

[2] B. Chazelle, A Faster Deterministic Algorithm for Minimum Spanning Trees, *Journal of the ACM*, 47 (2000), 1028–1047.

[3] M. L. Fredman, On the Efficiency of Pairing Heaps and Related Data Structures, *Journal of the ACM*, 46 (1999), 473–501.

[4] M. L. Fredman, A Priority Queue Transform, *WAE: International Workshop on Algorithm Engineering* LNCS 1668 (1999), 243–257.

[5] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan, The Pairing Heap: A New Form of Self-adjusting Heap, *Algorithmica*, 1 (1986), 111–129.

[6] M. L. Fredman and R. E. Tarjan, Fibonacci Heaps and Their Uses in Improved Optimization Algorithms, *Journal of the ACM*, 34 (1987), 596–615.

[7] J. Iacono, New upper bounds for pairing heaps, *Scandinavian Workshop on Algorithm Theory*, LNCS 1851 (2000), 35–42.

[8] J. Iacono, Distribution sensitive data structures, *Ph.D. Thesis, Rutgers University*, 2001.

[9] D. E. Knuth, *The Stanford Graph Base*, ACM Press, New York, N.Y., 1994.

[10] D. E. Knuth, *Sorting and Searching* 2d ed., Addison-Wesley, Reading MA., 1998.

[11] B. M. E. Moret and H. D. Shapiro, An Empirical Analysis of Algorithms for Constructing a Minimum Spanning Tree, *Proceedings of the Second Workshop on Algorithms and Data Structures* (1991), 400–411.

[12] S. Pettie and V. Ramachandran, An Optimal Minimum Spanning Tree Algorithm, *Journal of the ACM* 49 (2002), 16–34.

[13] J. Vuillemin, A Data Structure for Manipulating Priority Queues, *Communications of the ACM*, 21 (1978), 309–314.

[14] M. A. Weiss, *Data Structures and Algorithms in C*, 2d ed., Addison-Wesley, Reading MA., 1997.

© 2005 by Chapman & Hall/CRC