Binomial Heaps

This chapter and Chapter 20 present data structures known as mergeable heaps, which support the following five operations.

MAKE-HEAP() creates and returns a new heap containing no elements.

INSERT(H, x) inserts node x, whose key field has already been filled in, into heap H.

MINIMUM(H) returns a pointer to the node in heap H whose key is minimum.

EXTRACT-MIN(H) deletes the node from heap H whose key is minimum, returning a pointer to the node.

UNION(H_1, H_2) creates and returns a new heap that contains all the nodes of heaps H_1 and H_2. Heaps H_1 and H_2 are “destroyed” by this operation.

In addition, the data structures in these chapters also support the following two operations.

DECREASE-KEY(H, x, k) assigns to node x within heap H the new key value k, which is assumed to be no greater than its current key value.¹

DELETE(H, x) deletes node x from heap H.

As the table in Figure 19.1 shows, if we don’t need the UNION operation, ordinary binary heaps, as used in heapsort (Chapter 6), work well. Operations other than UNION run in worst-case time $O(\log n)$ (or better) on a binary heap. If the UNION operation must be supported, however, binary heaps perform poorly. By concatenating the two arrays that hold the binary heaps to be merged and then running MIN-HEAPIFY (see Exercise 6.2-2), the UNION operation takes $\Theta(n)$ time in the worst case.

¹As mentioned in the introduction to Part V, our default mergeable heaps are mergeable min-heaps, and so the operations MINIMUM, EXTRACT-MIN, and DECREASE-KEY apply. Alternatively, we could define a mergeable max-heap with the operations MAXIMUM, EXTRACT-MAX, and INCREASE-KEY.
In this chapter, we examine “binomial heaps,” whose worst-case time bounds are also shown in Figure 19.1. In particular, the UNION operation takes only $O(\lg n)$ time to merge two binomial heaps with a total of $n$ elements.

In Chapter 20, we shall explore Fibonacci heaps, which have even better time bounds for some operations. Note, however, that the running times for Fibonacci heaps in Figure 19.1 are amortized time bounds, not worst-case per-operation time bounds.

This chapter ignores issues of allocating nodes prior to insertion and freeing nodes following deletion. We assume that the code that calls the heap procedures deals with these details.

Binary heaps, binomial heaps, and Fibonacci heaps are all inefficient in their support of the operation SEARCH; it can take a while to find a node with a given key. For this reason, operations such as DECREASE-KEY and DELETE that refer to a given node require a pointer to that node as part of their input. As in our discussion of priority queues in Section 6.5, when we use a mergeable heap in an application, we often store a handle to the corresponding application object in each mergeable-heap element, as well as a handle to corresponding mergeable-heap element in each application object. The exact nature of these handles depends on the application and its implementation.

Section 19.1 defines binomial heaps after first defining their constituent binomial trees. It also introduces a particular representation of binomial heaps. Section 19.2 shows how we can implement operations on binomial heaps in the time bounds given in Figure 19.1.
19.1 Binomial trees and binomial heaps

A binomial heap is a collection of binomial trees, so this section starts by defining binomial trees and proving some key properties. We then define binomial heaps and show how they can be represented.

19.1.1 Binomial trees

The binomial tree $B_k$ is an ordered tree (see Section B.5.2) defined recursively. As shown in Figure 19.2(a), the binomial tree $B_0$ consists of a single node. The binomial tree $B_k$ consists of two binomial trees $B_{k-1}$ that are linked together: the root of one is the leftmost child of the root of the other. Figure 19.2(b) shows the binomial trees $B_0$ through $B_4$.

Some properties of binomial trees are given by the following lemma.

Lemma 19.1 (Properties of binomial trees)
For the binomial tree $B_k$,

1. there are $2^k$ nodes,
2. the height of the tree is $k$,
3. there are exactly $\binom{k}{i}$ nodes at depth $i$ for $i = 0, 1, \ldots, k$, and
4. the root has degree $k$, which is greater than that of any other node; moreover if the children of the root are numbered from left to right by $k - 1, k - 2, \ldots, 0$, child $i$ is the root of a subtree $B_i$.

Proof The proof is by induction on $k$. For each property, the basis is the binomial tree $B_0$. Verifying that each property holds for $B_0$ is trivial.

For the inductive step, we assume that the lemma holds for $B_{k-1}$.

1. Binomial tree $B_k$ consists of two copies of $B_{k-1}$, and so $B_k$ has $2^{k-1} + 2^{k-1} = 2^k$ nodes.
2. Because of the way in which the two copies of $B_{k-1}$ are linked to form $B_k$, the maximum depth of a node in $B_k$ is one greater than the maximum depth in $B_{k-1}$. By the inductive hypothesis, this maximum depth is $(k - 1) + 1 = k$.
3. Let $D(k, i)$ be the number of nodes at depth $i$ of binomial tree $B_k$. Since $B_k$ is composed of two copies of $B_{k-1}$ linked together, a node at depth $i$ in $B_{k-1}$ appears in $B_k$ once at depth $i$ and once at depth $i + 1$. In other words, the number of nodes at depth $i$ in $B_k$ is the number of nodes at depth $i$ in $B_{k-1}$ plus

bounds are

$O(n \ln n)$

for time

for time

freeing

freeing

in their

in their

such that refer

such that refer

the heap in

the heap in

an object

an object

the size

the size

depends

depends

binomial

binomial

Figure 19.2

Figure 19.2

the bounds
the number of nodes at depth $i - 1$ in $B_{k-1}$. Thus,

$$D(k, i) = D(k - 1, i) + D(k - 1, i - 1) \quad \text{(by the inductive hypothesis)}$$

$$= \binom{k - 1}{i} + \binom{k - 1}{i - 1} \quad \text{(by Exercise C.1-7)}$$

$$= \binom{k}{i}.$$  

4. The only node with greater degree in $B_k$ than in $B_{k-1}$ is the root, which has one more child than in $B_{k-1}$. Since the root of $B_{k-1}$ has degree $k - 1$, the root of $B_k$ has degree $k$. Now, by the inductive hypothesis, and as Figure 19.2(c) shows, from left to right, the children of the root of $B_{k-1}$ are roots of $B_{k-2}, B_{k-3}, \ldots, B_0$. When $B_{k-1}$ is linked to $B_{k-1}$, therefore, the children of the resulting root are roots of $B_{k-1}, B_{k-2}, \ldots, B_0$. 

\[ \square \]
Corollary 19.2
The maximum degree of any node in an \( n \)-node binomial tree is \( \lg n \).

**Proof** Immediate from properties 1 and 4 of Lemma 19.1. \( \square \)

The term "binomial tree" comes from property 3 of Lemma 19.1, since the terms \( \binom{k}{j} \) are the binomial coefficients. Exercise 19.1-3 gives further justification for the term.

### 19.1.2 Binomial heaps

A *binomial heap* \( H \) is a set of binomial trees that satisfies the following *binomial-heap properties*.

1. Each binomial tree in \( H \) obeys the *min-heap property*: the key of a node is greater than or equal to the key of its parent. We say that each such tree is *min-heap-ordered*.

2. For any nonnegative integer \( k \), there is at most one binomial tree in \( H \) whose root has degree \( k \).

The first property tells us that the root of a min-heap-ordered tree contains the smallest key in the tree.

The second property implies that an \( n \)-node binomial heap \( H \) consists of at most \( \lfloor \lg n \rfloor + 1 \) binomial trees. To see why, observe that the binary representation of \( n \) has \( \lfloor \lg n \rfloor + 1 \) bits, say \( b_{\lfloor \lg n \rfloor}, b_{\lfloor \lg n \rfloor - 1}, \ldots, b_0 \), so that \( n = \sum_{i=0}^{\lfloor \lg n \rfloor} b_i 2^i \). By property 1 of Lemma 19.1, therefore, binomial tree \( B_i \) appears in \( H \) if and only if bit \( b_i = 1 \). Thus, binomial heap \( H \) contains at most \( \lfloor \lg n \rfloor + 1 \) binomial trees.

Figure 19.3(a) shows a binomial heap \( H \) with 13 nodes. The binary representation of 13 is \( 1101 \), and \( H \) consists of min-heap-ordered binomial trees \( B_3, B_2, \) and \( B_0 \), having 8, 4, and 1 nodes respectively, for a total of 13 nodes.

### Representing binomial heaps

As shown in Figure 19.3(b), each binomial tree within a binomial heap is stored in the left-child, right-sibling representation of Section 10.4. Each node has a *key* field and any other satellite information required by the application. In addition, each node \( x \) contains pointers \( p[x] \) to its parent, \( child[x] \) to its leftmost child, and \( sibling[x] \) to the sibling of \( x \) immediately to its right. If node \( x \) is a root, then \( p[x] = \text{NIL} \). If node \( x \) has no children, then \( child[x] = \text{NIL} \), and if \( x \) is the rightmost child of its parent, then \( sibling[x] = \text{NIL} \). Each node \( x \) also contains the field \( \text{degree}[x] \), which is the number of children of \( x \).

As Figure 19.3 also shows, the roots of the binomial trees within a binomial heap are organized in a linked list, which we refer to as the *root list*. The degrees
Figure 19.3 A binomial heap $H$ with $n = 13$ nodes. (a) The heap consists of binomial trees $B_0$, $B_2$, and $B_3$, which have 1, 4, and 8 nodes respectively, totaling $n = 13$ nodes. Since each binomial tree is min-heap-ordered, the key of any node is no less than the key of its parent. Also shown is the root list, which is a linked list of roots in order of increasing degree. (b) A more detailed representation of binomial heap $H$. Each binomial tree is stored in the left-child, right-sibling representation, and each node stores its degree.

of the roots strictly increase as we traverse the root list. By the second binomial-heap property, in an $n$-node binomial heap the degrees of the roots are a subset of $\{0, 1, \ldots, \lfloor \lg n \rfloor \}$. The sibling field has a different meaning for roots than for nonroots. If $x$ is a root, then $\text{sibling}[x]$ points to the next root in the root list. (As usual, $\text{sibling}[x] = \text{NIL}$ if $x$ is the last root in the root list.)

A given binomial heap $H$ is accessed by the field $\text{head}[H]$, which is simply a pointer to the first root in the root list of $H$. If binomial heap $H$ has no elements, then $\text{head}[H] = \text{NIL}$. 
19.2 Operations on binomial heaps

In this section, we show how to perform operations on binomial heaps in the time bounds shown in Figure 19.1. We shall only show the upper bounds; the lower bounds are left as Exercise 19.2-10.

Creating a new binomial heap

To make an empty binomial heap, the MAKE-BINOMIAL-HEAP procedure simply allocates and returns an object $H$, where head[$H$] = NIL. The running time is $\Theta(1)$.
Finding the minimum key

The procedure \textsc{Binomial-Heap-Minimum} returns a pointer to the node with the minimum key in an \( n \)-node binomial heap \( H \). This implementation assumes that there are no keys with value \( \infty \). (See Exercise 19.2-5.)

\begin{verbatim}
BINOMIAL-HEAP-MINIMUM(H)
    1 y ← NIL
    2 x ← head[H]
    3 min ← ∞
    4 while x ≠ NIL
    5       do if key[x] < min
    6          then min ← key[x]
    7                  y ← x
    8          x ← sibling[x]
    9 return y
\end{verbatim}

Since a binomial heap is min-heap-ordered, the minimum key must reside in a root node. The \textsc{Binomial-Heap-Minimum} procedure checks all roots, which number at most \( \lfloor \log n \rfloor + 1 \), saving the current minimum in \( \textit{min} \) and a pointer to the current minimum in \( y \). When called on the binomial heap of Figure 19.3, \textsc{Binomial-Heap-Minimum} returns a pointer to the node with key 1.

Because there are at most \( \lfloor \log n \rfloor + 1 \) roots to check, the running time of \textsc{Binomial-Heap-Minimum} is \( O(\log n) \).

Uniting two binomial heaps

The operation of uniting two binomial heaps is used as a subroutine by most of the remaining operations. The \textsc{Binomial-Heap-Union} procedure repeatedly links binomial trees whose roots have the same degree. The following procedure links the \( B_{k-1} \) tree rooted at node \( y \) to the \( B_{k-1} \) tree rooted at node \( z \); that is, it makes \( z \) the parent of \( y \). Node \( z \) thus becomes the root of a \( B_k \) tree.

\begin{verbatim}
BINOMIAL-LINK(y, z)
    1 p[y] ← z
    2 sibling[y] ← child[z]
    3 child[z] ← y
    4 degree[z] ← degree[z] + 1
\end{verbatim}

The \textsc{Binomial-Link} procedure makes node \( y \) the new head of the linked list of node \( z \)'s children in \( O(1) \) time. It works because the left-child, right-sibling representation of each binomial tree matches the ordering property of the tree: in a \( B_k \) tree, the leftmost child of the root is the root of a \( B_{k-1} \) tree.
The following procedure unites binomial heaps $H_1$ and $H_2$, returning the resulting heap. It destroys the representations of $H_1$ and $H_2$ in the process. Besides \textsc{Binomial-Link}, the procedure uses an auxiliary procedure \textsc{Binomial-Heap-Merge} that merges the root lists of $H_1$ and $H_2$ into a single linked list that is sorted by degree into monotonically increasing order. The \textsc{Binomial-Heap-Merge} procedure, whose pseudocode we leave as Exercise 19.2-1, is similar to the \textsc{Merge} procedure in Section 2.3.1.

\textbf{Binomial-Heap-Union}($H_1$, $H_2$)

1. $H \leftarrow \text{Make-Binomial-Heap}()$
2. $head[H] \leftarrow \text{Binomial-Heap-Merge}(H_1, H_2)$
3. free the objects $H_1$ and $H_2$ but not the lists they point to
4. if $head[H] = \text{NIL}$
5. \hspace{1em} then return $H$
6. prev-$x \leftarrow \text{NIL}$
7. $x \leftarrow head[H]$
8. next-$x \leftarrow \text{Sibling}[x]$
9. while next-$x \neq \text{NIL}$
10. \hspace{1em} do if (degree[$x$] \neq degree[next-$x$]) or
11. \hspace{2em} (\text{sibling}[next-$x$] \neq \text{NIL} and degree[sibling[next-$x$]] = degree[$x$])
12. \hspace{1em} then prev-$x \leftarrow x$
13. \hspace{1em} $x \leftarrow$ next-$x$
14. \hspace{1em} else if key[$x$] \leq key[next-$x$]
15. \hspace{2em} then sibling[$x$] \leftarrow sibling[next-$x$]
16. \hspace{2em} Binomial-Link(next-$x$, $x$)
17. \hspace{1em} else if prev-$x = \text{NIL}$
18. \hspace{2em} then head[H] \leftarrow next-$x$
19. \hspace{2em} else sibling[prev-$x$] \leftarrow next-$x$
20. \hspace{2em} Binomial-Link($x$, next-$x$)
21. \hspace{2em} $x \leftarrow$ next-$x$
22. return $H$

Figure 19.5 shows an example of \textsc{Binomial-Heap-Union} in which all four cases given in the pseudocode occur.

The \textsc{Binomial-Heap-Union} procedure has two phases. The first phase, performed by the call of \textsc{Binomial-Heap-Merge}, merges the root lists of binomial heaps $H_1$ and $H_2$ into a single linked list $H$ that is sorted by degree into monotonically increasing order. There might be as many as two roots (but no more) of each degree, however, so the second phase links roots of equal degree until at most one root remains of each degree. Because the linked list $H$ is sorted by degree, we can perform all the link operations quickly.
Figure 19.5 The execution of \textsc{Binomial-Heap-Union}. (a) Binomial heaps $H_1$ and $H_2$. (b) Binomial heap $H$ is the output of \textsc{Binomial-Heap-Merge} ($H_1$, $H_2$). Initially, $x$ is the first root in the root list of $H$. Because both $x$ and $\text{next-}x$ have degree 0 and $\text{key}(x) < \text{key}(\text{next-}x)$, case 3 applies. (c) After the link occurs, $x$ is the first of three roots with the same degree, so case 2 applies. (d) After all the pointers move down one position in the root list, case 4 applies, since $x$ is the first of two roots of equal degree. (e) After the link occurs, case 3 applies. (f) After another link, case 1 applies, because $x$ has degree 3 and next-x has degree 4. This iteration of the while loop is the last, because after the pointers move down one position in the root list, next-x = NIL.

In detail, the procedure works as follows. Lines 1–3 start by merging the root lists of binomial heaps $H_1$ and $H_2$ into a single root list $H$. The root lists of $H_1$ and $H_2$ are sorted by strictly increasing degree, and \textsc{Binomial-Heap-Merge} returns a root list $H$ that is sorted by monotonically increasing degree. If the root lists of $H_1$ and $H_2$ have $m$ roots altogether, \textsc{Binomial-Heap-Merge} runs in $O(m)$ time by repeatedly examining the roots at the heads of the two root lists and appending the root with the lower degree to the output root list, removing it from its input root list in the process.
The **BINOMIAL-HEAP-UNION** procedure next initializes some pointers into the root list of \( H \). First, it simply returns in lines 4–5 if it happens to be uniting two empty binomial heaps. From line 6 on, therefore, we know that \( H \) has at least one root. Throughout the procedure, we maintain three pointers into the root list:

- \( x \) points to the root currently being examined,
- \( \text{prev-}x \) points to the root preceding \( x \) on the root list: \( \text{ sibling}[\text{prev-}x] = x \) (since initially \( x \) has no predecessor, we start with \( \text{prev-}x \) set to NIL), and
- \( \text{next-}x \) points to the root following \( x \) on the root list: \( \text{ sibling}[x] = \text{next-}x \).

Initially, there are at most two roots on the root list \( H \) of a given degree: because \( H_1 \) and \( H_2 \) were binomial heaps, they each had at most one root of a given degree. Moreover, **BINOMIAL-HEAP-MERGE** guarantees us that if two roots in \( H \) have the same degree, they are adjacent in the root list.

In fact, during the execution of **BINOMIAL-HEAP-UNION**, there may be three roots of a given degree appearing on the root list \( H \) at some time. We shall see
in a moment how this situation could occur. At each iteration of the while loop of lines 9–21, therefore, we decide whether to link \( x \) and \( \text{next-x} \) based on their degrees and possibly the degree of \( \text{siblings[next-x]} \). An invariant of the loop is that each time we start the body of the loop, both \( x \) and \( \text{next-x} \) are non-nil. (See Exercise 19.2.4 for a precise loop invariant.)

Case 1, shown in Figure 19.6(a), occurs when \( \text{degree}[x] \neq \text{degree[\text{next-x}]} \), that is, when \( x \) is the root of a \( B_k \)-tree and \( \text{next-x} \) is the root of a \( B_l \)-tree for some \( l > k \). Lines 11–12 handle this case. We don’t link \( x \) and \( \text{next-x} \), so we simply march the pointers one position farther down the list. Updating \( \text{next-x} \) to point to the node following the new node \( x \) is handled in line 21, which is common to every case.

Case 2, shown in Figure 19.6(b), occurs when \( x \) is the first of three roots of equal degree, that is, when

\[
\text{degree}[x] = \text{degree[\text{next-x}]} = \text{degree[siblings[\text{next-x}]]}.
\]

We handle this case in the same manner as case 1: we just march the pointers one position farther down the list. The next iteration will execute either case 3 or case 4 to combine the second and third of the three equal-degree roots. Line 10 tests for both cases 1 and 2, and lines 11–12 handle both cases.

Cases 3 and 4 occur when \( x \) is the first of two roots of equal degree, that is, when

\[
\text{degree}[x] = \text{degree[\text{next-x}]} \neq \text{degree[siblings[\text{next-x}]]}.
\]

These cases may occur in any iteration, but one of them always occurs immediately following case 2. In cases 3 and 4, we link \( x \) and \( \text{next-x} \). The two cases are distinguished by whether \( x \) or \( \text{next-x} \) has the smaller key, which determines the node that will be the root after the two are linked.

In case 3, shown in Figure 19.6(c), \( \text{key}[x] \leq \text{key[\text{next-x}]} \), so \( \text{next-x} \) is linked to \( x \). Line 14 removes \( \text{next-x} \) from the root list, and line 15 makes \( \text{next-x} \) the leftmost child of \( x \).

In case 4, shown in Figure 19.6(d), \( \text{next-x} \) has the smaller key, so \( x \) is linked to \( \text{next-x} \). Lines 16–18 remove \( x \) from the root list; there are two cases depending on whether \( x \) is the first root on the list (line 17) or is not (line 18). Line 19 then makes \( x \) the leftmost child of \( \text{next-x} \), and line 20 updates \( x \) for the next iteration.

Following either case 3 or case 4, the setup for the next iteration of the while loop is the same. We have just linked two \( B_k \)-trees to form a \( B_{k+1} \)-tree, which \( x \) now points to. There were already zero, one, or two other \( B_{k+1} \)-trees on the root list resulting from \( \text{BINOMIAL-heap-merge} \), so \( x \) is now the first of either one, two, or three \( B_{k+1} \)-trees on the root list. If \( x \) is the only one, then we enter case 1 in the next iteration: \( \text{degree}[x] \neq \text{degree[\text{next-x}]} \). If \( x \) is the first of two, then we enter either case 3 or case 4 in the next iteration. If \( x \) is the first of three that we enter case 2 in the next iteration.

The running time of \( \text{BINOMIAL-heap-union} \) is \( O(\lg n) \), where \( n \) is the total number of nodes in binomial heaps \( H_1 \) and \( H_2 \). We can see this as follows. Let \( H_1 \)
Figure 19.6 The four cases that occur in BINOMIAL-HEAP-UNION. Labels a, b, c, and d serve only to identify the roots involved; they do not indicate the degrees or keys of these roots. In each case, x is the root of a $B_k$-tree and $l > k$. (a) Case 1: $\text{degree}[x] \neq \text{degree}[\text{next-x}]$. The pointers move one position farther down the root list. (b) Case 2: $\text{degree}[x] = \text{degree}[\text{next-x}] = \text{degree}[\text{child-x}]$. Again, the pointers move one position farther down the list, and the next iteration executes either case 3 or case 4. (c) Case 3: $\text{degree}[x] = \text{degree}[\text{next-x}] \neq \text{degree}[\text{child-x}]$ and $\text{key}[x] \leq \text{key}[\text{next-x}]$. We remove next-x from the root list and link it to x, creating a $B_{k+1}$-tree. (d) Case 4: $\text{degree}[x] = \text{degree}[\text{next-x}] \neq \text{degree}[\text{child-x}]$ and $\text{key}[x] > \text{key}[\text{next-x}]$. We remove x from the root list and link it to next-x, again creating a $B_{k+1}$-tree.

contain $n_1$ nodes and $H_2$ contain $n_2$ nodes, so that $n = n_1 + n_2$. Then $H_1$ contains at most $\lceil \log n_1 \rceil + 1$ roots and $H_2$ contains at most $\lceil \log n_2 \rceil + 1$ roots, and so $H$ contains at most $\lceil \log n_1 \rceil + \lceil \log n_2 \rceil + 2 \leq \lceil \log n \rceil + 2 = O(\log n)$ roots immediately after the call of BINOMIAL-HEAP-MERGE. The time to perform BINOMIAL-HEAP-MERGE is thus $O(\log n)$. Each iteration of the while loop takes $O(1)$ time, and there are at most $\lceil \log n_1 \rceil + \lceil \log n_2 \rceil + 2$ iterations because each iteration either advances the
pointers one position down the root list of $H$ or removes a root from the root list. The total time is thus $O(\lg n)$.

**Inserting a node**

The following procedure inserts node $x$ into binomial heap $H$, assuming that $x$ has already been allocated and $key[x]$ has already been filled in.

```
BINOMIAL-HEAP-INSERT($H$, $x$)
1 $H' \leftarrow$ MAKE-BINOMIAL-HEAP()
2 $p[x] \leftarrow$ NIL
3 $\text{child}[x] \leftarrow$ NIL
4 $\text{Sibling}[x] \leftarrow$ NIL
5 $\text{degree}[x] \leftarrow 0$
6 $\text{head}[$H'`] \leftarrow x$
7 $H \leftarrow$ BINOMIAL-HEAP-UNION($H$, $H'$)
```

The procedure simply makes a one-node binomial heap $H'$ in $O(1)$ time and unites it with the $n$-node binomial heap $H$ in $O(\lg n)$ time. The call to BINOMIAL-HEAP-UNION takes care of freeing the temporary binomial heap $H'$. (A direct implementation that does not call BINOMIAL-HEAP-UNION is given as Exercise 19.2-8.)

**Extracting the node with minimum key**

The following procedure extracts the node with the minimum key from binomial heap $H$ and returns a pointer to the extracted node.

```
BINOMIAL-HEAP-EXTRACT-MIN($H$)
1 find the root $x$ with the minimum key in the root list of $H$, and remove $x$ from the root list of $H$
2 $H' \leftarrow$ MAKE-BINOMIAL-HEAP()
3 reverse the order of the linked list of $x$'s children, and set $\text{head}[$H'`] to point to the head of the resulting list
4 $H \leftarrow$ BINOMIAL-HEAP-UNION($H$, $H'$)
5 return $x$
```

This procedure works as shown in Figure 19.7. The input binomial heap $H$ is shown in Figure 19.7(a). Figure 19.7(b) shows the situation after line 1: the root $x$ with the minimum key has been removed from the root list of $H$. If $x$ is the root of a $B_k$-tree, then by property 4 of Lemma 19.1, $x$'s children, from left to right, are roots of $B_{k-1}$, $B_{k-2}$, ..., $B_0$-trees. Figure 19.7(c) shows that by reversing the list of $x$'s children in line 3, we have a binomial heap $H'$ that contains every node
in $x$'s tree except for $x$ itself. Because $x$'s tree was removed from $H$ in line 1, the binomial heap that results from uniting $H$ and $H'$ in line 4, shown in Figure 19.7(d), contains all the nodes originally in $H$ except for $x$. Finally, line 5 returns $x$.

Since each of lines 1-4 takes $O(\log n)$ time if $H$ has $n$ nodes, BINOMIAL-HEAP-EXTRACT-MIN runs in $O(\log n)$ time.
Decreasing a key

The following procedure decreases the key of a node $x$ in a binomial heap $H$ to a new value $k$. It signals an error if $k$ is greater than $x$'s current key.

\begin{verbatim}
BINOMIAL-HEAP-DECREASE-KEY($H$, $x$, $k$)
  1 if $k > key[x]$
  2     then error "new key is greater than current key"
  3     key[$x$] $\leftarrow k$
  4     y $\leftarrow x$
  5     z $\leftarrow p[y]$
  6     while $z \neq \text{NIL}$ and $key[y] < key[z]$
  7         do exchange $key[y] \leftrightarrow key[z]$
  8         $>$ If $y$ and $z$ have satellite fields, exchange them, too.
  9     y $\leftarrow z$
10     z $\leftarrow p[y]$
\end{verbatim}

As shown in Figure 19.8, this procedure decreases a key in the same manner as in a binary min-heap: by "bubbling up" the key in the heap. After ensuring that the new key is in fact no greater than the current key and then assigning the new key to $x$, the procedure goes up the tree, with $y$ initially pointing to node $x$. In each iteration of the while loop of lines 6–10, $key[y]$ is checked against the key of $y$'s parent $z$. If $y$ is the root or $key[y] \geq key[z]$, the binomial tree is now min-heap-ordered. Otherwise, node $y$ violates min-heap ordering, and so its key is exchanged with the key of its parent $z$, along with any other satellite information. The procedure then sets $y$ to $z$, going up one level in the tree, and continues with the next iteration.

The \texttt{BINOMIAL-HEAP-DECREASE-KEY} procedure takes $O(\lg n)$ time. By property 2 of Lemma 19.1, the maximum depth of $x$ is $\lfloor \lg n \rfloor$, so the while loop of lines 6–10 iterates at most $\lfloor \lg n \rfloor$ times.

Deleting a key

It is easy to delete a node $x$'s key and satellite information from binomial heap $H$ in $O(\lg n)$ time. The following implementation assumes that no node currently in the binomial heap has a key of $-\infty$.

\begin{verbatim}
BINOMIAL-HEAP-DELETE($H$, $x$)
  1 BINOMIAL-HEAP-DECREASE-KEY($H$, $x$, $-\infty$)
  2 BINOMIAL-HEAP-EXTRACT-MIN($H$)
\end{verbatim}

The \texttt{BINOMIAL-HEAP-DELETE} procedure makes node $x$ have the unique minimum key in the entire binomial heap by giving it a key of $-\infty$. (Exercise 19.2-6
Figure 19.8  The action of Binomial-Heap-Decrease-Key. (a) The situation just before line 6 of the first iteration of the while loop. Node y has had its key decreased to 7, which is less than the key of y’s parent z. (b) The keys of the two nodes are exchanged, and the situation just before line 6 of the second iteration is shown. Pointers y and z have moved up one level in the tree, but min-heap order is still violated. (c) After another exchange and moving pointers y and z up one more level, we find that min-heap order is satisfied, so the while loop terminates.

Exercises

19.2-1
Write pseudocode for Binomial-Heap-Merge.
19.2-2
Show the binomial heap that results when a node with key 24 is inserted into the
binomial heap shown in Figure 19.7(d).

19.2-3
Show the binomial heap that results when the node with key 28 is deleted from the
binomial heap shown in Figure 19.8(c).

19.2-4
Argue the correctness of BI NOMIAL-HEAP-UNION using the following loop in-
variant:

At the start of each iteration of the while loop of lines 9–21, x points to a
root that is one of the following:

- the only root of its degree,
- the first of the only two roots of its degree, or
- the first or second of the only three roots of its degree.

Moreover, all roots preceding x’s predecessor on the root list have unique
degrees on the root list, and if x’s predecessor has a degree different from
that of x, its degree on the root list is unique, too. Finally, node degrees
monotonically increase as we traverse the root list.

19.2-5
Explain why the BI NOMIAL-HEAP-MINIMUM procedure might not work correctly
if keys can have the value \( \infty \). Rewrite the pseudocode to make it work correctly in
such cases.

19.2-6
Suppose there is no way to represent the key \(-\infty\). Rewrite the BI NOMIAL-HEAP-
DELETE procedure to work correctly in this situation. It should still take \( O(\lg n) \)
time.

19.2-7
Discuss the relationship between inserting into a binomial heap and incrementing a
binary number and the relationship between uniting two binomial heaps and adding
two binary numbers.

19.2-8
In light of Exercise 19.2-7, rewrite BI NOMIAL-HEAP-INSERT to insert a node di-
rectly into a binomial heap without calling BI NOMIAL-HEAP-UNION.
19.2-10
Find inputs that cause \textsc{Binomial-Heap-Extract-Min}, \textsc{Binomial-Heap-Decrease-Key}, and \textsc{Binomial-Heap-Delete} to run in $\Omega(lg n)$ time. Explain why the worst-case running times of \textsc{Binomial-Heap-Insert}, \textsc{Binomial-Heap-Minimum}, and \textsc{Binomial-Heap-Union} are $\tilde{\Omega}(lg n)$ but not $\Omega(lg n)$. (See Problem 3-5.)

Problems

19-1 2-3-4 heaps
Chapter 18 introduced the 2-3-4 tree, in which every internal node (other than possibly the root) has two, three, or four children and all leaves have the same depth. In this problem, we shall implement 2-3-4 heaps, which support the mergeable-heap operations.

The 2-3-4 heaps differ from 2-3-4 trees in the following ways. In 2-3-4 heaps, only leaves store keys, and each leaf $x$ stores exactly one key in the field \textit{key}[x]. There is no particular ordering of the keys in the leaves; that is, from left to right, the keys may be in any order. Each internal node $x$ contains a value \textit{small}[x] that is equal to the smallest key stored in any leaf in the subtree rooted at $x$. The root $r$ contains a field \textit{height}[r] that is the height of the tree. Finally, 2-3-4 heaps are intended to be kept in main memory, so that disk reads and writes are not needed.

Implement the following 2-3-4 heap operations. Each of the operations in parts (a)–(e) should run in $O(lg n)$ time on a 2-3-4 heap with $n$ elements. The UNION operation in part (f) should run in $O(lg n)$ time, where $n$ is the number of elements in the two input heaps.

\begin{enumerate}
  \item \textsc{Minimum}, which returns a pointer to the leaf with the smallest key.
  \item \textsc{Decrease-Key}, which decreases the key of a given leaf $x$ to a given value $k \leq \textit{key}[x]$.
  \item \textsc{Insert}, which inserts leaf $x$ with key $k$.
  \item \textsc{Delete}, which deletes a given leaf $x$.
  \item \textsc{Extract-Min}, which extracts the leaf with the smallest key.
\end{enumerate}
f. UNION, which unites two 2-3-4 heaps, returning a single 2-3-4 heap and destroying the input heaps.

19-2 Minimum-spanning-tree algorithm using binomial heaps

Chapter 23 presents two algorithms to solve the problem of finding a minimum spanning tree of an undirected graph. Here, we shall see how binomial heaps can be used to devise a different minimum-spanning-tree algorithm.

We are given a connected, undirected graph $G = (V, E)$ with a weight function $w : E \rightarrow \mathbb{R}$. We call $w(u, v)$ the weight of edge $(u, v)$. We wish to find a minimum spanning tree for $G$: an acyclic subset $T \subseteq E$ that connects all the vertices in $V$ and whose total weight

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

is minimized.

The following pseudocode, which can be proven correct using techniques from Section 23.1, constructs a minimum spanning tree $T$. It maintains a partition $\{V_i\}$ of the vertices of $V$ and, with each set $V_i$, a set $E_i \subseteq \{(u, v) : u \in V_i \text{ or } v \in V_i\}$ of edges incident on vertices in $V_i$.

\begin{verbatim}
MST(G)
1  T <- Ø
2  for each vertex $v_i \in V[G]$
3      do $V_i$ <- \{v_i\}
4          $E_i$ <- \{(v_i, v) \in E[G]\}
5  while there is more than one set $V_i$
6      do choose any set $V_i$
7          extract the minimum-weight edge $(u, v)$ from $E_i$
8          assume without loss of generality that $u \in V_i$ and $v \in V_j$
9          if $i \neq j$
10             then $T$ <- $T \cup \{(u, v)\}$
11             $V_i$ <- $V_i \cup V_j$, destroying $V_j$
12             $E_i$ <- $E_i \cup E_j$
\end{verbatim}

Describe how to implement this algorithm using binomial heaps to manage the vertex and edge sets. Do you need to change the representation of a binomial heap? Do you need to add operations beyond the mergeable-heap operations given in Figure 19.1? Give the running time of your implementation.
Binomial heaps were introduced in 1978 by Vuillemin [307]. Brown [49, 50] studied their properties in detail.