

B-trees are balanced search trees designed to work well on disks or other direct-access secondary storage devices. B-trees are similar to red-black trees (Chapter 13), but they are better at minimizing disk I/O operations. Many database systems use B-trees, or variants of B-trees, to store information.

B-trees differ from red-black trees in that B-tree nodes may have many children, from a few to thousands. That is, the “branching factor” of a B-tree can be quite large, although it usually depends on characteristics of the disk unit used. B-trees are similar to red-black trees in that every n -node B-tree has height $O(\lg n)$. The exact height of a B-tree can be considerably less than that of a red-black tree, however, because its branching factor, and hence the base of the logarithm that expresses its height, can be much larger. Therefore, we can also use B-trees to implement many dynamic-set operations in time $O(\lg n)$.

B-trees generalize binary search trees in a natural manner. Figure 18.1 shows a simple B-tree. If an internal B-tree node x contains $x.n$ keys, then x has $x.n + 1$ children. The keys in node x serve as dividing points separating the range of keys handled by x into $x.n + 1$ subranges, each handled by one child of x . When searching for a key in a B-tree, we make an $(x.n + 1)$ -way decision based on comparisons with the $x.n$ keys stored at node x . The structure of leaf nodes differs from that of internal nodes; we will examine these differences in Section 18.1.

Section 18.1 gives a precise definition of B-trees and proves that the height of a B-tree grows only logarithmically with the number of nodes it contains. Section 18.2 describes how to search for a key and insert a key into a B-tree, and Section 18.3 discusses deletion. Before proceeding, however, we need to ask why we evaluate data structures designed to work on a disk differently from data structures designed to work in main random-access memory.

Data structures on secondary storage

Computer systems take advantage of various technologies that provide memory capacity. The *primary memory* (or *main memory*) of a computer system normally

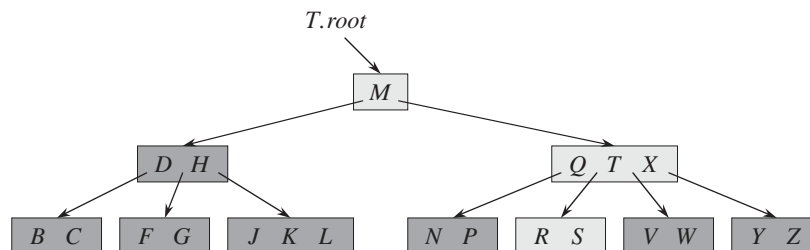


Figure 18.1 A B-tree whose keys are the consonants of English. An internal node x containing $x.n$ keys has $x.n + 1$ children. All leaves are at the same depth in the tree. The lightly shaded nodes are examined in a search for the letter R .

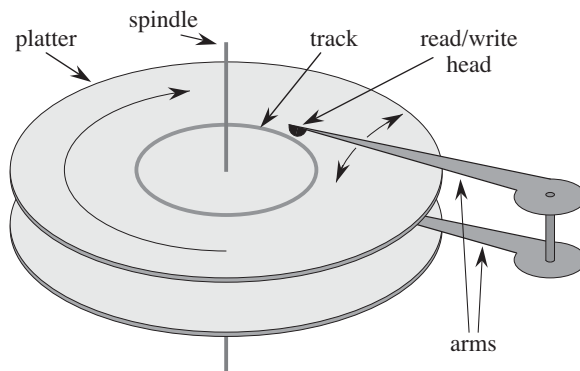


Figure 18.2 A typical disk drive. It comprises one or more platters (two platters are shown here) that rotate around a spindle. Each platter is read and written with a head at the end of an arm. Arms rotate around a common pivot axis. A track is the surface that passes beneath the read/write head when the head is stationary.

consists of silicon memory chips. This technology is typically more than an order of magnitude more expensive per bit stored than magnetic storage technology, such as tapes or disks. Most computer systems also have *secondary storage* based on magnetic disks; the amount of such secondary storage often exceeds the amount of primary memory by at least two orders of magnitude.

Figure 18.2 shows a typical disk drive. The drive consists of one or more *platters*, which rotate at a constant speed around a common *spindle*. A magnetizable material covers the surface of each platter. The drive reads and writes each platter by a *head* at the end of an *arm*. The arms can move their heads toward or away

from the spindle. When a given head is stationary, the surface that passes underneath it is called a *track*. Multiple platters increase only the disk drive's capacity and not its performance.

Although disks are cheaper and have higher capacity than main memory, they are much, much slower because they have moving mechanical parts.¹ The mechanical motion has two components: platter rotation and arm movement. As of this writing, commodity disks rotate at speeds of 5400–15,000 revolutions per minute (RPM). We typically see 15,000 RPM speeds in server-grade drives, 7200 RPM speeds in drives for desktops, and 5400 RPM speeds in drives for laptops. Although 7200 RPM may seem fast, one rotation takes 8.33 milliseconds, which is over 5 orders of magnitude longer than the 50 nanosecond access times (more or less) commonly found for silicon memory. In other words, if we have to wait a full rotation for a particular item to come under the read/write head, we could access main memory more than 100,000 times during that span. On average we have to wait for only half a rotation, but still, the difference in access times for silicon memory compared with disks is enormous. Moving the arms also takes some time. As of this writing, average access times for commodity disks are in the range of 8 to 11 milliseconds.

In order to amortize the time spent waiting for mechanical movements, disks access not just one item but several at a time. Information is divided into a number of equal-sized *pages* of bits that appear consecutively within tracks, and each disk read or write is of one or more entire pages. For a typical disk, a page might be 2^{11} to 2^{14} bytes in length. Once the read/write head is positioned correctly and the disk has rotated to the beginning of the desired page, reading or writing a magnetic disk is entirely electronic (aside from the rotation of the disk), and the disk can quickly read or write large amounts of data.

Often, accessing a page of information and reading it from a disk takes longer than examining all the information read. For this reason, in this chapter we shall look separately at the two principal components of the running time:

- the number of disk accesses, and
- the CPU (computing) time.

We measure the number of disk accesses in terms of the number of pages of information that need to be read from or written to the disk. We note that disk-access time is not constant—it depends on the distance between the current track and the desired track and also on the initial rotational position of the disk. We shall

¹As of this writing, solid-state drives have recently come onto the consumer market. Although they are faster than mechanical disk drives, they cost more per gigabyte and have lower capacities than mechanical disk drives.

nonetheless use the number of pages read or written as a first-order approximation of the total time spent accessing the disk.

In a typical B-tree application, the amount of data handled is so large that all the data do not fit into main memory at once. The B-tree algorithms copy selected pages from disk into main memory as needed and write back onto disk the pages that have changed. B-tree algorithms keep only a constant number of pages in main memory at any time; thus, the size of main memory does not limit the size of B-trees that can be handled.

We model disk operations in our pseudocode as follows. Let x be a pointer to an object. If the object is currently in the computer's main memory, then we can refer to the attributes of the object as usual: $x.key$, for example. If the object referred to by x resides on disk, however, then we must perform the operation $\text{DISK-READ}(x)$ to read object x into main memory before we can refer to its attributes. (We assume that if x is already in main memory, then $\text{DISK-READ}(x)$ requires no disk accesses; it is a “no-op.”) Similarly, the operation $\text{DISK-WRITE}(x)$ is used to save any changes that have been made to the attributes of object x . That is, the typical pattern for working with an object is as follows:

```

 $x$  = a pointer to some object
DISK-READ( $x$ )
operations that access and/or modify the attributes of  $x$ 
DISK-WRITE( $x$ )           // omitted if no attributes of  $x$  were changed
other operations that access but do not modify attributes of  $x$ 

```

The system can keep only a limited number of pages in main memory at any one time. We shall assume that the system flushes from main memory pages no longer in use; our B-tree algorithms will ignore this issue.

Since in most systems the running time of a B-tree algorithm depends primarily on the number of DISK-READ and DISK-WRITE operations it performs, we typically want each of these operations to read or write as much information as possible. Thus, a B-tree node is usually as large as a whole disk page, and this size limits the number of children a B-tree node can have.

For a large B-tree stored on a disk, we often see branching factors between 50 and 2000, depending on the size of a key relative to the size of a page. A large branching factor dramatically reduces both the height of the tree and the number of disk accesses required to find any key. Figure 18.3 shows a B-tree with a branching factor of 1001 and height 2 that can store over one billion keys; nevertheless, since we can keep the root node permanently in main memory, we can find any key in this tree by making at most only two disk accesses.

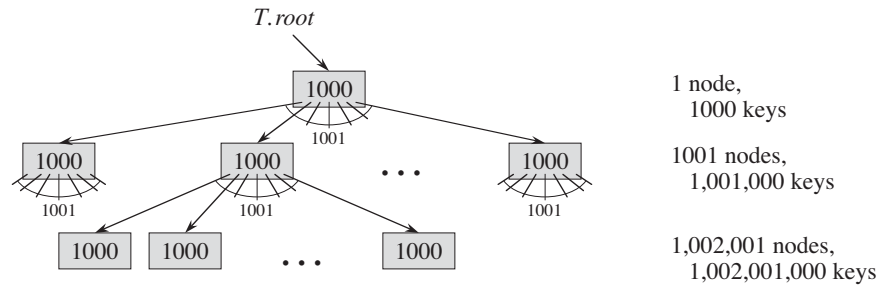


Figure 18.3 A B-tree of height 2 containing over one billion keys. Shown inside each node x is $x.n$, the number of keys in x . Each internal node and leaf contains 1000 keys. This B-tree has 1001 nodes at depth 1 and over one million leaves at depth 2.

18.1 Definition of B-trees

To keep things simple, we assume, as we have for binary search trees and red-black trees, that any “satellite information” associated with a key resides in the same node as the key. In practice, one might actually store with each key just a pointer to another disk page containing the satellite information for that key. The pseudocode in this chapter implicitly assumes that the satellite information associated with a key, or the pointer to such satellite information, travels with the key whenever the key is moved from node to node. A common variant on a B-tree, known as a **B^+ -tree**, stores all the satellite information in the leaves and stores only keys and child pointers in the internal nodes, thus maximizing the branching factor of the internal nodes.

A **B-tree** T is a rooted tree (whose root is $T.root$) having the following properties:

1. Every node x has the following attributes:
 - a. $x.n$, the number of keys currently stored in node x ,
 - b. the $x.n$ keys themselves, $x.key_1, x.key_2, \dots, x.key_{x.n}$, stored in nondecreasing order, so that $x.key_1 \leq x.key_2 \leq \dots \leq x.key_{x.n}$,
 - c. $x.leaf$, a boolean value that is TRUE if x is a leaf and FALSE if x is an internal node.
2. Each internal node x also contains $x.n + 1$ pointers $x.c_1, x.c_2, \dots, x.c_{x.n+1}$ to its children. Leaf nodes have no children, and so their c_i attributes are undefined.

3. The keys $x.key_i$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $x.c_i$, then

$$k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq \cdots \leq x.key_{x.n} \leq k_{x.n+1} .$$

4. All leaves have the same depth, which is the tree's height h .
5. Nodes have lower and upper bounds on the number of keys they can contain. We express these bounds in terms of a fixed integer $t \geq 2$ called the **minimum degree** of the B-tree:
- a. Every node other than the root must have at least $t - 1$ keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.
 - b. Every node may contain at most $2t - 1$ keys. Therefore, an internal node may have at most $2t$ children. We say that a node is **full** if it contains exactly $2t - 1$ keys.²

The simplest B-tree occurs when $t = 2$. Every internal node then has either 2, 3, or 4 children, and we have a **2-3-4 tree**. In practice, however, much larger values of t yield B-trees with smaller height.

The height of a B-tree

The number of disk accesses required for most operations on a B-tree is proportional to the height of the B-tree. We now analyze the worst-case height of a B-tree.

Theorem 18.1

If $n \geq 1$, then for any n -key B-tree T of height h and minimum degree $t \geq 2$,

$$h \leq \log_t \frac{n + 1}{2} .$$

Proof The root of a B-tree T contains at least one key, and all other nodes contain at least $t - 1$ keys. Thus, T , whose height is h , has at least 2 nodes at depth 1, at least $2t$ nodes at depth 2, at least $2t^2$ nodes at depth 3, and so on, until at depth h it has at least $2t^{h-1}$ nodes. Figure 18.4 illustrates such a tree for $h = 3$. Thus, the

²Another common variant on a B-tree, known as a **B*-tree**, requires each internal node to be at least $2/3$ full, rather than at least half full, as a B-tree requires.

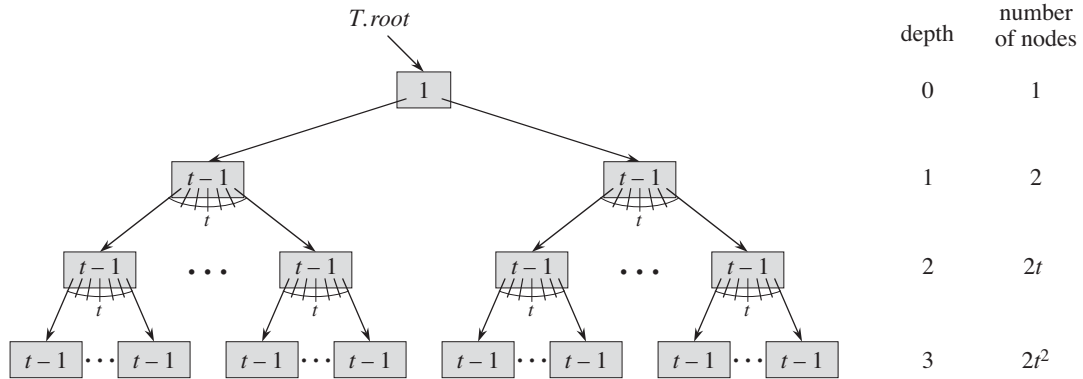


Figure 18.4 A B-tree of height 3 containing a minimum possible number of keys. Shown inside each node x is $x.n$.

number n of keys satisfies the inequality

$$\begin{aligned}
 n &\geq 1 + (t-1) \sum_{i=1}^h 2t^{i-1} \\
 &= 1 + 2(t-1) \left(\frac{t^h - 1}{t-1} \right) \\
 &= 2t^h - 1.
 \end{aligned}$$

By simple algebra, we get $t^h \leq (n+1)/2$. Taking base- t logarithms of both sides proves the theorem. ■

Here we see the power of B-trees, as compared with red-black trees. Although the height of the tree grows as $O(\lg n)$ in both cases (recall that t is a constant), for B-trees the base of the logarithm can be many times larger. Thus, B-trees save a factor of about $\lg t$ over red-black trees in the number of nodes examined for most tree operations. Because we usually have to access the disk to examine an arbitrary node in a tree, B-trees avoid a substantial number of disk accesses.

Exercises

18.1-1

Why don't we allow a minimum degree of $t = 1$?

18.1-2

For what values of t is the tree of Figure 18.1 a legal B-tree?

18.1-3

Show all legal B-trees of minimum degree 2 that represent $\{1, 2, 3, 4, 5\}$.

18.1-4

As a function of the minimum degree t , what is the maximum number of keys that can be stored in a B-tree of height h ?

18.1-5

Describe the data structure that would result if each black node in a red-black tree were to absorb its red children, incorporating their children with its own.

18.2 Basic operations on B-trees

In this section, we present the details of the operations B-TREE-SEARCH, B-TREE-CREATE, and B-TREE-INSERT. In these procedures, we adopt two conventions:

- The root of the B-tree is always in main memory, so that we never need to perform a DISK-READ on the root; we do have to perform a DISK-WRITE of the root, however, whenever the root node is changed.
- Any nodes that are passed as parameters must already have had a DISK-READ operation performed on them.

The procedures we present are all “one-pass” algorithms that proceed downward from the root of the tree, without having to back up.

Searching a B-tree

Searching a B-tree is much like searching a binary search tree, except that instead of making a binary, or “two-way,” branching decision at each node, we make a multiway branching decision according to the number of the node’s children. More precisely, at each internal node x , we make an $(x.n + 1)$ -way branching decision.

B-TREE-SEARCH is a straightforward generalization of the TREE-SEARCH procedure defined for binary search trees. B-TREE-SEARCH takes as input a pointer to the root node x of a subtree and a key k to be searched for in that subtree. The top-level call is thus of the form B-TREE-SEARCH($T.root, k$). If k is in the B-tree, B-TREE-SEARCH returns the ordered pair (y, i) consisting of a node y and an index i such that $y.key_i = k$. Otherwise, the procedure returns NIL.


```

B-TREE-SEARCH( $x, k$ )
1   $i = 1$ 
2  while  $i \leq x.n$  and  $k > x.key_i$ 
3       $i = i + 1$ 
4  if  $i \leq x.n$  and  $k == x.key_i$ 
5      return  $(x, i)$ 
6  elseif  $x.leaf$ 
7      return NIL
8  else DISK-READ( $x.c_i$ )
9      return B-TREE-SEARCH( $x.c_i, k$ )

```

Using a linear-search procedure, lines 1–3 find the smallest index i such that $k \leq x.key_i$, or else they set i to $x.n + 1$. Lines 4–5 check to see whether we have now discovered the key, returning if we have. Otherwise, lines 6–9 either terminate the search unsuccessfully (if x is a leaf) or recurse to search the appropriate subtree of x , after performing the necessary DISK-READ on that child.

Figure 18.1 illustrates the operation of B-TREE-SEARCH. The procedure examines the lightly shaded nodes during a search for the key R .

As in the TREE-SEARCH procedure for binary search trees, the nodes encountered during the recursion form a simple path downward from the root of the tree. The B-TREE-SEARCH procedure therefore accesses $O(h) = O(\log_t n)$ disk pages, where h is the height of the B-tree and n is the number of keys in the B-tree. Since $x.n < 2t$, the **while** loop of lines 2–3 takes $O(t)$ time within each node, and the total CPU time is $O(th) = O(t \log_t n)$.

Creating an empty B-tree

To build a B-tree T , we first use B-TREE-CREATE to create an empty root node and then call B-TREE-INSERT to add new keys. Both of these procedures use an auxiliary procedure ALLOCATE-NODE, which allocates one disk page to be used as a new node in $O(1)$ time. We can assume that a node created by ALLOCATE-NODE requires no DISK-READ, since there is as yet no useful information stored on the disk for that node.

```

B-TREE-CREATE( $T$ )
1   $x = \text{ALLOCATE-NODE}()$ 
2   $x.leaf = \text{TRUE}$ 
3   $x.n = 0$ 
4  DISK-WRITE( $x$ )
5   $T.root = x$ 

```

B-TREE-CREATE requires $O(1)$ disk operations and $O(1)$ CPU time.

Inserting a key into a B-tree

Inserting a key into a B-tree is significantly more complicated than inserting a key into a binary search tree. As with binary search trees, we search for the leaf position at which to insert the new key. With a B-tree, however, we cannot simply create a new leaf node and insert it, as the resulting tree would fail to be a valid B-tree. Instead, we insert the new key into an existing leaf node. Since we cannot insert a key into a leaf node that is full, we introduce an operation that *splits* a full node y (having $2t - 1$ keys) around its *median key* $y.key_t$ into two nodes having only $t - 1$ keys each. The median key moves up into y 's parent to identify the dividing point between the two new trees. But if y 's parent is also full, we must split it before we can insert the new key, and thus we could end up splitting full nodes all the way up the tree.

As with a binary search tree, we can insert a key into a B-tree in a single pass down the tree from the root to a leaf. To do so, we do not wait to find out whether we will actually need to split a full node in order to do the insertion. Instead, as we travel down the tree searching for the position where the new key belongs, we split each full node we come to along the way (including the leaf itself). Thus whenever we want to split a full node y , we are assured that its parent is not full.

Splitting a node in a B-tree

The procedure B-TREE-SPLIT-CHILD takes as input a *nonfull* internal node x (assumed to be in main memory) and an index i such that $x.c_i$ (also assumed to be in main memory) is a *full* child of x . The procedure then splits this child in two and adjusts x so that it has an additional child. To split a full root, we will first make the root a child of a new empty root node, so that we can use B-TREE-SPLIT-CHILD. The tree thus grows in height by one; splitting is the only means by which the tree grows.

Figure 18.5 illustrates this process. We split the full node $y = x.c_i$ about its median key S , which moves up into y 's parent node x . Those keys in y that are greater than the median key move into a new node z , which becomes a new child of x .

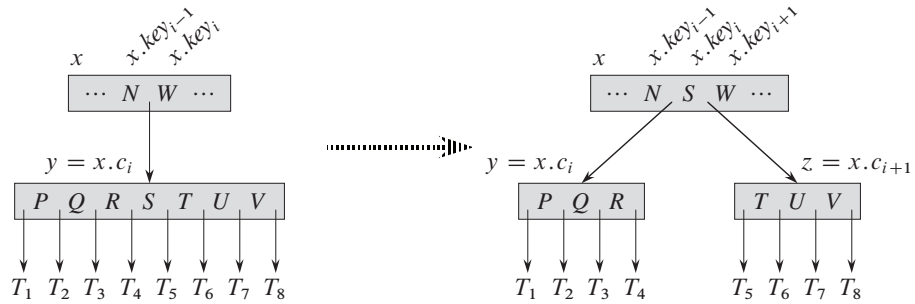


Figure 18.5 Splitting a node with $t = 4$. Node $y = x.c_i$ splits into two nodes, y and z , and the median key S of y moves up into y 's parent.

B-TREE-SPLIT-CHILD(x, i)

```

1   $z = \text{ALLOCATE-NODE}()$ 
2   $y = x.c_i$ 
3   $z.\text{leaf} = y.\text{leaf}$ 
4   $z.n = t - 1$ 
5  for  $j = 1$  to  $t - 1$ 
6       $z.\text{key}_j = y.\text{key}_{j+t}$ 
7  if not  $y.\text{leaf}$ 
8      for  $j = 1$  to  $t$ 
9           $z.c_j = y.c_{j+t}$ 
10  $y.n = t - 1$ 
11 for  $j = x.n + 1$  downto  $i + 1$ 
12      $x.c_{j+1} = x.c_j$ 
13  $x.c_{i+1} = z$ 
14 for  $j = x.n$  downto  $i$ 
15      $x.\text{key}_{j+1} = x.\text{key}_j$ 
16  $x.\text{key}_i = y.\text{key}_t$ 
17  $x.n = x.n + 1$ 
18 DISK-WRITE( $y$ )
19 DISK-WRITE( $z$ )
20 DISK-WRITE( $x$ )

```

B-TREE-SPLIT-CHILD works by straightforward “cutting and pasting.” Here, x is the node being split, and y is x 's i th child (set in line 2). Node y originally has $2t$ children ($2t - 1$ keys) but is reduced to t children ($t - 1$ keys) by this operation. Node z takes the t largest children ($t - 1$ keys) from y , and z becomes a new child

of x , positioned just after y in x 's table of children. The median key of y moves up to become the key in x that separates y and z .

Lines 1–9 create node z and give it the largest $t - 1$ keys and corresponding t children of y . Line 10 adjusts the key count for y . Finally, lines 11–17 insert z as a child of x , move the median key from y up to x in order to separate y from z , and adjust x 's key count. Lines 18–20 write out all modified disk pages. The CPU time used by B-TREE-SPLIT-CHILD is $\Theta(t)$, due to the loops on lines 5–6 and 8–9. (The other loops run for $O(t)$ iterations.) The procedure performs $O(1)$ disk operations.

Inserting a key into a B-tree in a single pass down the tree

We insert a key k into a B-tree T of height h in a single pass down the tree, requiring $O(h)$ disk accesses. The CPU time required is $O(th) = O(t \log_t n)$. The B-TREE-INSERT procedure uses B-TREE-SPLIT-CHILD to guarantee that the recursion never descends to a full node.

B-TREE-INSERT(T, k)

```

1   $r = T.root$ 
2  if  $r.n == 2t - 1$ 
3       $s = \text{ALLOCATE-NODE}()$ 
4       $T.root = s$ 
5       $s.leaf = \text{FALSE}$ 
6       $s.n = 0$ 
7       $s.c_1 = r$ 
8      B-TREE-SPLIT-CHILD( $s, 1$ )
9      B-TREE-INSERT-NONFULL( $s, k$ )
10 else B-TREE-INSERT-NONFULL( $r, k$ )
```

Lines 3–9 handle the case in which the root node r is full: the root splits and a new node s (having two children) becomes the root. Splitting the root is the only way to increase the height of a B-tree. Figure 18.6 illustrates this case. Unlike a binary search tree, a B-tree increases in height at the top instead of at the bottom. The procedure finishes by calling B-TREE-INSERT-NONFULL to insert key k into the tree rooted at the nonfull root node. B-TREE-INSERT-NONFULL recurses as necessary down the tree, at all times guaranteeing that the node to which it recurses is not full by calling B-TREE-SPLIT-CHILD as necessary.

The auxiliary recursive procedure B-TREE-INSERT-NONFULL inserts key k into node x , which is assumed to be nonfull when the procedure is called. The operation of B-TREE-INSERT and the recursive operation of B-TREE-INSERT-NONFULL guarantee that this assumption is true.

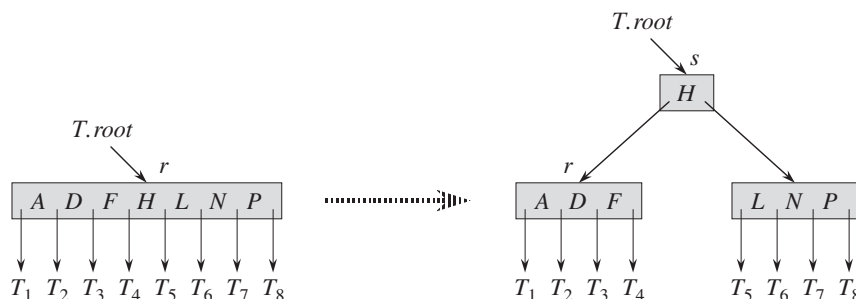


Figure 18.6 Splitting the root with $t = 4$. Root node r splits in two, and a new root node s is created. The new root contains the median key of r and has the two halves of r as children. The B-tree grows in height by one when the root is split.

B-TREE-INSERT-NONFULL(x, k)

```

1   $i = x.n$ 
2  if  $x.leaf$ 
3      while  $i \geq 1$  and  $k < x.key_i$ 
4           $x.key_{i+1} = x.key_i$ 
5           $i = i - 1$ 
6       $x.key_{i+1} = k$ 
7       $x.n = x.n + 1$ 
8      DISK-WRITE( $x$ )
9  else while  $i \geq 1$  and  $k < x.key_i$ 
10      $i = i - 1$ 
11      $i = i + 1$ 
12     DISK-READ( $x.c_i$ )
13     if  $x.c_i.n == 2t - 1$ 
14         B-TREE-SPLIT-CHILD( $x, i$ )
15         if  $k > x.key_i$ 
16              $i = i + 1$ 
17     B-TREE-INSERT-NONFULL( $x.c_i, k$ )

```

The B-TREE-INSERT-NONFULL procedure works as follows. Lines 3–8 handle the case in which x is a leaf node by inserting key k into x . If x is not a leaf node, then we must insert k into the appropriate leaf node in the subtree rooted at internal node x . In this case, lines 9–11 determine the child of x to which the recursion descends. Line 13 detects whether the recursion would descend to a full child, in which case line 14 uses B-TREE-SPLIT-CHILD to split that child into two nonfull children, and lines 15–16 determine which of the two children is now the

correct one to descend to. (Note that there is no need for a $\text{DISK-READ}(x.c_i)$ after line 16 increments i , since the recursion will descend in this case to a child that was just created by $\text{B-TREE-SPLIT-CHILD}$.) The net effect of lines 13–16 is thus to guarantee that the procedure never recurses to a full node. Line 17 then recurses to insert k into the appropriate subtree. Figure 18.7 illustrates the various cases of inserting into a B-tree.

For a B-tree of height h , B-TREE-INSERT performs $O(h)$ disk accesses, since only $O(1)$ DISK-READ and DISK-WRITE operations occur between calls to $\text{B-TREE-INSERT-NONFULL}$. The total CPU time used is $O(th) = O(t \log_t n)$. Since $\text{B-TREE-INSERT-NONFULL}$ is tail-recursive, we can alternatively implement it as a **while** loop, thereby demonstrating that the number of pages that need to be in main memory at any time is $O(1)$.

Exercises

18.2-1

Show the results of inserting the keys

$F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E$

in order into an empty B-tree with minimum degree 2. Draw only the configurations of the tree just before some node must split, and also draw the final configuration.

18.2-2

Explain under what circumstances, if any, redundant DISK-READ or DISK-WRITE operations occur during the course of executing a call to B-TREE-INSERT . (A redundant DISK-READ is a DISK-READ for a page that is already in memory. A redundant DISK-WRITE writes to disk a page of information that is identical to what is already stored there.)

18.2-3

Explain how to find the minimum key stored in a B-tree and how to find the predecessor of a given key stored in a B-tree.

18.2-4 ★

Suppose that we insert the keys $\{1, 2, \dots, n\}$ into an empty B-tree with minimum degree 2. How many nodes does the final B-tree have?

18.2-5

Since leaf nodes require no pointers to children, they could conceivably use a different (larger) t value than internal nodes for the same disk page size. Show how to modify the procedures for creating and inserting into a B-tree to handle this variation.

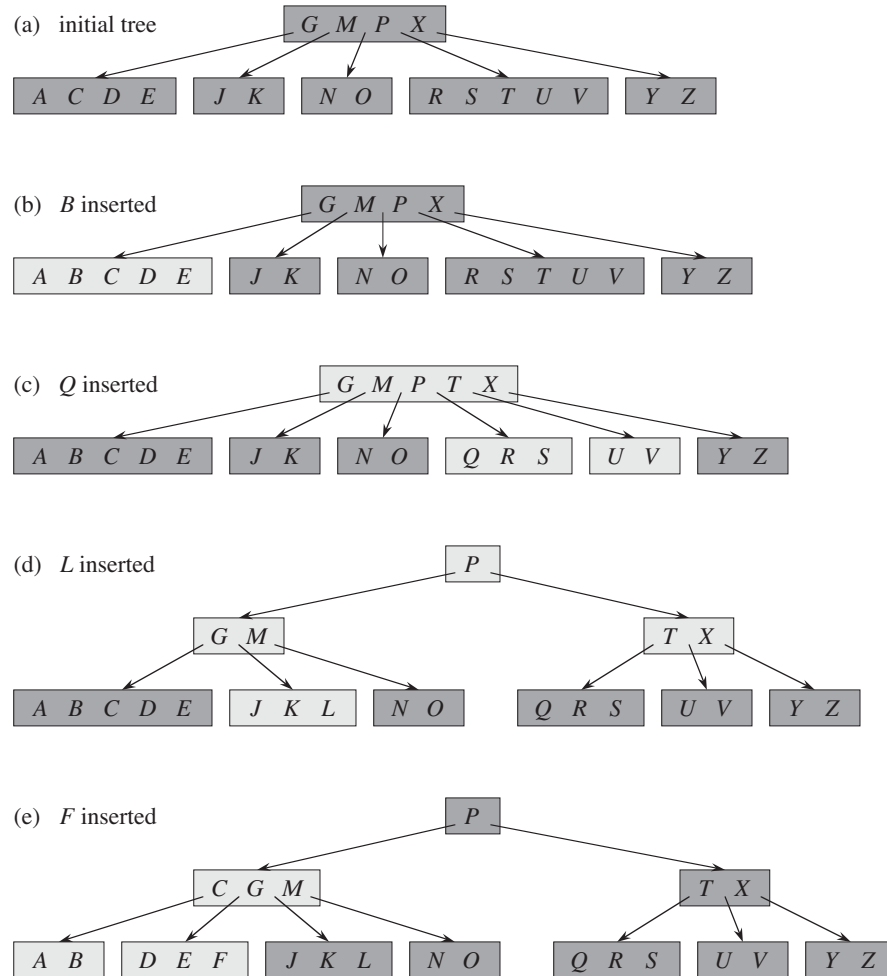


Figure 18.7 Inserting keys into a B-tree. The minimum degree t for this B-tree is 3, so a node can hold at most 5 keys. Nodes that are modified by the insertion process are lightly shaded. (a) The initial tree for this example. (b) The result of inserting *B* into the initial tree; this is a simple insertion into a leaf node. (c) The result of inserting *Q* into the previous tree. The node *RSTUV* splits into two nodes containing *RS* and *UV*, the key *T* moves up to the root, and *Q* is inserted in the leftmost of the two halves (the *RS* node). (d) The result of inserting *L* into the previous tree. The root splits right away, since it is full, and the B-tree grows in height by one. Then *L* is inserted into the leaf containing *JK*. (e) The result of inserting *F* into the previous tree. The node *ABCDE* splits before *F* is inserted into the rightmost of the two halves (the *DE* node).

18.2-6

Suppose that we were to implement B-TREE-SEARCH to use binary search rather than linear search within each node. Show that this change makes the CPU time required $O(\lg n)$, independently of how t might be chosen as a function of n .

18.2-7

Suppose that disk hardware allows us to choose the size of a disk page arbitrarily, but that the time it takes to read the disk page is $a + bt$, where a and b are specified constants and t is the minimum degree for a B-tree using pages of the selected size. Describe how to choose t so as to minimize (approximately) the B-tree search time. Suggest an optimal value of t for the case in which $a = 5$ milliseconds and $b = 10$ microseconds.

18.3 Deleting a key from a B-tree

Deletion from a B-tree is analogous to insertion but a little more complicated, because we can delete a key from any node—not just a leaf—and when we delete a key from an internal node, we will have to rearrange the node’s children. As in insertion, we must guard against deletion producing a tree whose structure violates the B-tree properties. Just as we had to ensure that a node didn’t get too big due to insertion, we must ensure that a node doesn’t get too small during deletion (except that the root is allowed to have fewer than the minimum number $t - 1$ of keys). Just as a simple insertion algorithm might have to back up if a node on the path to where the key was to be inserted was full, a simple approach to deletion might have to back up if a node (other than the root) along the path to where the key is to be deleted has the minimum number of keys.

The procedure B-TREE-DELETE deletes the key k from the subtree rooted at x . We design this procedure to guarantee that whenever it calls itself recursively on a node x , the number of keys in x is at least the minimum degree t . Note that this condition requires one more key than the minimum required by the usual B-tree conditions, so that sometimes a key may have to be moved into a child node before recursion descends to that child. This strengthened condition allows us to delete a key from the tree in one downward pass without having to “back up” (with one exception, which we’ll explain). You should interpret the following specification for deletion from a B-tree with the understanding that if the root node x ever becomes an internal node having no keys (this situation can occur in cases 2c and 3b on pages 501–502), then we delete x , and x ’s only child $x.c_1$ becomes the new root of the tree, decreasing the height of the tree by one and preserving the property that the root of the tree contains at least one key (unless the tree is empty).

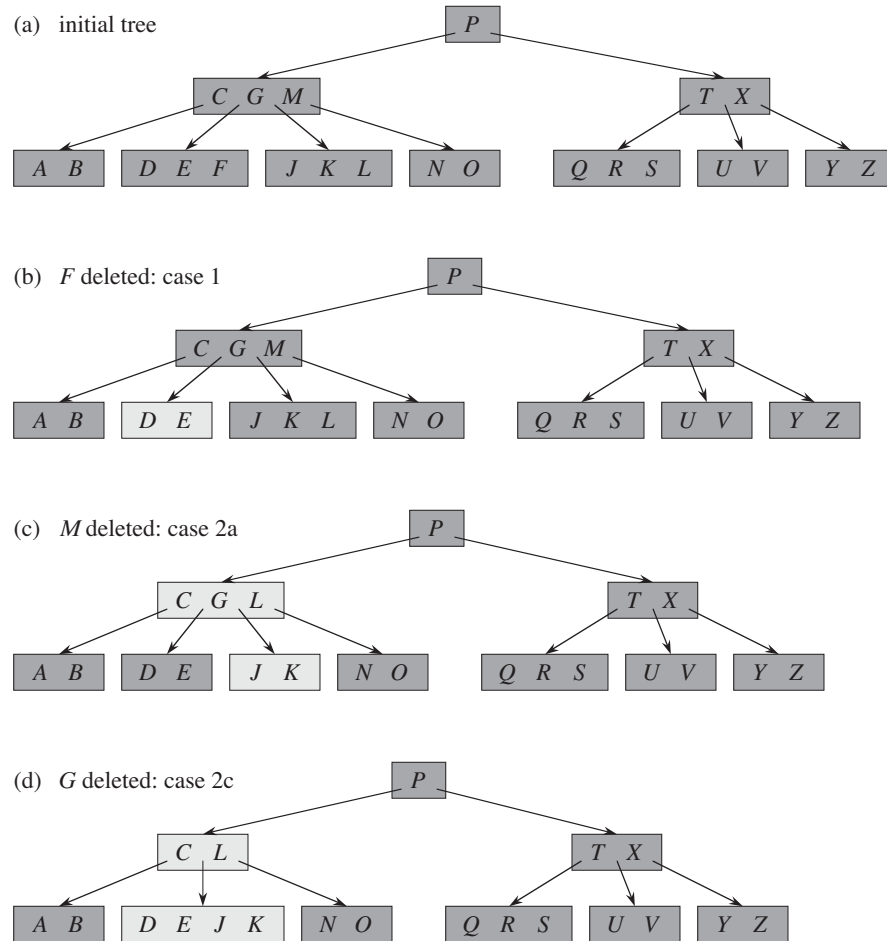
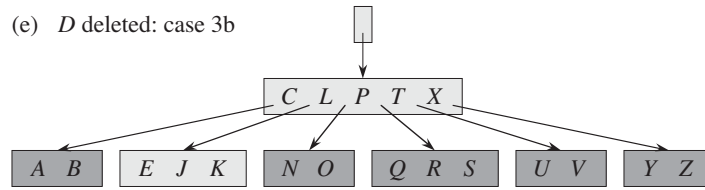


Figure 18.8 Deleting keys from a B-tree. The minimum degree for this B-tree is $t = 3$, so a node (other than the root) cannot have fewer than 2 keys. Nodes that are modified are lightly shaded. (a) The B-tree of Figure 18.7(e). (b) Deletion of *F*. This is case 1: simple deletion from a leaf. (c) Deletion of *M*. This is case 2a: the predecessor *L* of *M* moves up to take *M*'s position. (d) Deletion of *G*. This is case 2c: we push *G* down to make node *DEGJK* and then delete *G* from this leaf (case 1).

We sketch how deletion works instead of presenting the pseudocode. Figure 18.8 illustrates the various cases of deleting keys from a B-tree.

1. If the key k is in node x and x is a leaf, delete the key k from x .
2. If the key k is in node x and x is an internal node, do the following:

(e) D deleted: case 3b

(e') tree shrinks in height

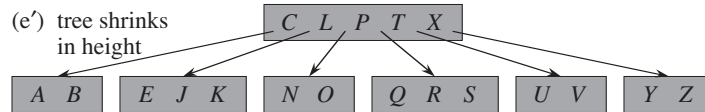
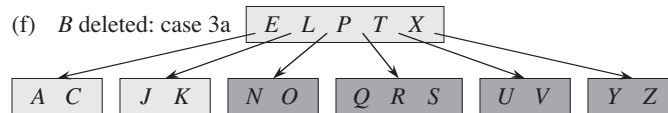
(f) B deleted: case 3a

Figure 18.8, continued (e) Deletion of D . This is case 3b: the recursion cannot descend to node CL because it has only 2 keys, so we push P down and merge it with CL and TX to form $CLPTX$; then we delete D from a leaf (case 1). (e') After (e), we delete the root and the tree shrinks in height by one. (f) Deletion of B . This is case 3a: C moves to fill B 's position and E moves to fill C 's position.

- a. If the child y that precedes k in node x has at least t keys, then find the predecessor k' of k in the subtree rooted at y . Recursively delete k' , and replace k by k' in x . (We can find k' and delete it in a single downward pass.)
 - b. If y has fewer than t keys, then, symmetrically, examine the child z that follows k in node x . If z has at least t keys, then find the successor k' of k in the subtree rooted at z . Recursively delete k' , and replace k by k' in x . (We can find k' and delete it in a single downward pass.)
 - c. Otherwise, if both y and z have only $t - 1$ keys, merge k and all of z into y , so that x loses both k and the pointer to z , and y now contains $2t - 1$ keys. Then free z and recursively delete k from y .
3. If the key k is not present in internal node x , determine the root $x.c_i$ of the appropriate subtree that must contain k , if k is in the tree at all. If $x.c_i$ has only $t - 1$ keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then finish by recursing on the appropriate child of x .

- a. If $x.c_i$ has only $t - 1$ keys but has an immediate sibling with at least t keys, give $x.c_i$ an extra key by moving a key from x down into $x.c_i$, moving a key from $x.c_i$'s immediate left or right sibling up into x , and moving the appropriate child pointer from the sibling into $x.c_i$.
- b. If $x.c_i$ and both of $x.c_i$'s immediate siblings have $t - 1$ keys, merge $x.c_i$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

Since most of the keys in a B-tree are in the leaves, we may expect that in practice, deletion operations are most often used to delete keys from leaves. The B-TREE-DELETE procedure then acts in one downward pass through the tree, without having to back up. When deleting a key in an internal node, however, the procedure makes a downward pass through the tree but may have to return to the node from which the key was deleted to replace the key with its predecessor or successor (cases 2a and 2b).

Although this procedure seems complicated, it involves only $O(h)$ disk operations for a B-tree of height h , since only $O(1)$ calls to DISK-READ and DISK-WRITE are made between recursive invocations of the procedure. The CPU time required is $O(th) = O(t \log_t n)$.

Exercises

18.3-1

Show the results of deleting C , P , and V , in order, from the tree of Figure 18.8(f).

18.3-2

Write pseudocode for B-TREE-DELETE.

Problems

18-1 Stacks on secondary storage

Consider implementing a stack in a computer that has a relatively small amount of fast primary memory and a relatively large amount of slower disk storage. The operations PUSH and POP work on single-word values. The stack we wish to support can grow to be much larger than can fit in memory, and thus most of it must be stored on disk.

A simple, but inefficient, stack implementation keeps the entire stack on disk. We maintain in memory a stack pointer, which is the disk address of the top element on the stack. If the pointer has value p , the top element is the $(p \bmod m)$ th word on page $\lfloor p/m \rfloor$ of the disk, where m is the number of words per page.

To implement the PUSH operation, we increment the stack pointer, read the appropriate page into memory from disk, copy the element to be pushed to the appropriate word on the page, and write the page back to disk. A POP operation is similar. We decrement the stack pointer, read in the appropriate page from disk, and return the top of the stack. We need not write back the page, since it was not modified.

Because disk operations are relatively expensive, we count two costs for any implementation: the total number of disk accesses and the total CPU time. Any disk access to a page of m words incurs charges of one disk access and $\Theta(m)$ CPU time.

- a.* Asymptotically, what is the worst-case number of disk accesses for n stack operations using this simple implementation? What is the CPU time for n stack operations? (Express your answer in terms of m and n for this and subsequent parts.)

Now consider a stack implementation in which we keep one page of the stack in memory. (We also maintain a small amount of memory to keep track of which page is currently in memory.) We can perform a stack operation only if the relevant disk page resides in memory. If necessary, we can write the page currently in memory to the disk and read in the new page from the disk to memory. If the relevant disk page is already in memory, then no disk accesses are required.

- b.* What is the worst-case number of disk accesses required for n PUSH operations? What is the CPU time?
- c.* What is the worst-case number of disk accesses required for n stack operations? What is the CPU time?

Suppose that we now implement the stack by keeping two pages in memory (in addition to a small number of words for bookkeeping).

- d.* Describe how to manage the stack pages so that the amortized number of disk accesses for any stack operation is $O(1/m)$ and the amortized CPU time for any stack operation is $O(1)$.

18-2 Joining and splitting 2-3-4 trees

The *join* operation takes two dynamic sets S' and S'' and an element x such that for any $x' \in S'$ and $x'' \in S''$, we have $x'.key < x.key < x''.key$. It returns a set $S = S' \cup \{x\} \cup S''$. The *split* operation is like an “inverse” join: given a dynamic set S and an element $x \in S$, it creates a set S' that consists of all elements in $S - \{x\}$ whose keys are less than $x.key$ and a set S'' that consists of all elements in $S - \{x\}$ whose keys are greater than $x.key$. In this problem, we investigate

how to implement these operations on 2-3-4 trees. We assume for convenience that elements consist only of keys and that all key values are distinct.

- a. Show how to maintain, for every node x of a 2-3-4 tree, the height of the subtree rooted at x as an attribute $x.height$. Make sure that your implementation does not affect the asymptotic running times of searching, insertion, and deletion.
- b. Show how to implement the join operation. Given two 2-3-4 trees T' and T'' and a key k , the join operation should run in $O(1 + |h' - h''|)$ time, where h' and h'' are the heights of T' and T'' , respectively.
- c. Consider the simple path p from the root of a 2-3-4 tree T to a given key k , the set S' of keys in T that are less than k , and the set S'' of keys in T that are greater than k . Show that p breaks S' into a set of trees $\{T'_0, T'_1, \dots, T'_m\}$ and a set of keys $\{k'_1, k'_2, \dots, k'_m\}$, where, for $i = 1, 2, \dots, m$, we have $y < k'_i < z$ for any keys $y \in T'_{i-1}$ and $z \in T'_i$. What is the relationship between the heights of T'_{i-1} and T'_i ? Describe how p breaks S'' into sets of trees and keys.
- d. Show how to implement the split operation on T . Use the join operation to assemble the keys in S' into a single 2-3-4 tree T' and the keys in S'' into a single 2-3-4 tree T'' . The running time of the split operation should be $O(\lg n)$, where n is the number of keys in T . (*Hint:* The costs for joining should telescope.)

Chapter notes

Knuth [211], Aho, Hopcroft, and Ullman [5], and Sedgewick [306] give further discussions of balanced-tree schemes and B-trees. Comer [74] provides a comprehensive survey of B-trees. Guibas and Sedgewick [155] discuss the relationships among various kinds of balanced-tree schemes, including red-black trees and 2-3-4 trees.

In 1970, J. E. Hopcroft invented 2-3 trees, a precursor to B-trees and 2-3-4 trees, in which every internal node has either two or three children. Bayer and McCreight [35] introduced B-trees in 1972; they did not explain their choice of name.

Bender, Demaine, and Farach-Colton [40] studied how to make B-trees perform well in the presence of memory-hierarchy effects. Their *cache-oblivious* algorithms work efficiently without explicitly knowing the data transfer sizes within the memory hierarchy.

19 Fibonacci Heaps

The Fibonacci heap data structure serves a dual purpose. First, it supports a set of operations that constitutes what is known as a “mergeable heap.” Second, several Fibonacci-heap operations run in constant amortized time, which makes this data structure well suited for applications that invoke these operations frequently.

Mergeable heaps

A *mergeable heap* is any data structure that supports the following five operations, in which each element has a *key*:

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

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

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

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

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

In addition to the mergeable-heap operations above, Fibonacci heaps also support the following two operations:

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

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

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.

Procedure	Binary heap (worst-case)	Fibonacci heap (amortized)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$O(\lg n)$

Figure 19.1 Running times for operations on two implementations of mergeable heaps. The number of items in the heap(s) at the time of an operation is denoted by n .

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 fairly well. Operations other than UNION run in worst-case time $O(\lg n)$ on a binary heap. If we need to support the UNION operation, however, binary heaps perform poorly. By concatenating the two arrays that hold the binary heaps to be merged and then running BUILD-MIN-HEAP (see Section 6.3), the UNION operation takes $\Theta(n)$ time in the worst case.

Fibonacci heaps, on the other hand, have better asymptotic time bounds than binary heaps for the INSERT, UNION, and DECREASE-KEY operations, and they have the same asymptotic running times for the remaining 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. The UNION operation takes only constant amortized time in a Fibonacci heap, which is significantly better than the linear worst-case time required in a binary heap (assuming, of course, that an amortized time bound suffices).

Fibonacci heaps in theory and practice

From a theoretical standpoint, Fibonacci heaps are especially desirable when the number of EXTRACT-MIN and DELETE operations is small relative to the number of other operations performed. This situation arises in many applications. For example, some algorithms for graph problems may call DECREASE-KEY once per edge. For dense graphs, which have many edges, the $\Theta(1)$ amortized time of each call of DECREASE-KEY adds up to a big improvement over the $\Theta(\lg n)$ worst-case time of binary heaps. Fast algorithms for problems such as computing minimum spanning trees (Chapter 23) and finding single-source shortest paths (Chapter 24) make essential use of Fibonacci heaps.

From a practical point of view, however, the constant factors and programming complexity of Fibonacci heaps make them less desirable than ordinary binary (or k -ary) heaps for most applications, except for certain applications that manage large amounts of data. Thus, Fibonacci heaps are predominantly of theoretical interest. If a much simpler data structure with the same amortized time bounds as Fibonacci heaps were developed, it would be of practical use as well.

Both binary heaps and Fibonacci heaps are inefficient in how they support the operation SEARCH; it can take a while to find an element with a given key. For this reason, operations such as DECREASE-KEY and DELETE that refer to a given element require a pointer to that element 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 the corresponding mergeable-heap element in each application object. The exact nature of these handles depends on the application and its implementation.

Like several other data structures that we have seen, Fibonacci heaps are based on rooted trees. We represent each element by a node within a tree, and each node has a *key* attribute. For the remainder of this chapter, we shall use the term “node” instead of “element.” We shall also ignore issues of allocating nodes prior to insertion and freeing nodes following deletion, assuming instead that the code calling the heap procedures deals with these details.

Section 19.1 defines Fibonacci heaps, discusses how we represent them, and presents the potential function used for their amortized analysis. Section 19.2 shows how to implement the mergeable-heap operations and achieve the amortized time bounds shown in Figure 19.1. The remaining two operations, DECREASE-KEY and DELETE, form the focus of Section 19.3. Finally, Section 19.4 finishes a key part of the analysis and also explains the curious name of the data structure.

19.1 Structure of Fibonacci heaps

A **Fibonacci heap** is a collection of rooted trees that are *min-heap ordered*. That is, each tree obeys the *min-heap property*: the key of a node is greater than or equal to the key of its parent. Figure 19.2(a) shows an example of a Fibonacci heap.

As Figure 19.2(b) shows, each node x contains a pointer $x.p$ to its parent and a pointer $x.child$ to any one of its children. The children of x are linked together in a circular, doubly linked list, which we call the *child list* of x . Each child y in a child list has pointers $y.left$ and $y.right$ that point to y 's left and right siblings, respectively. If node y is an only child, then $y.left = y.right = y$. Siblings may appear in a child list in any order.

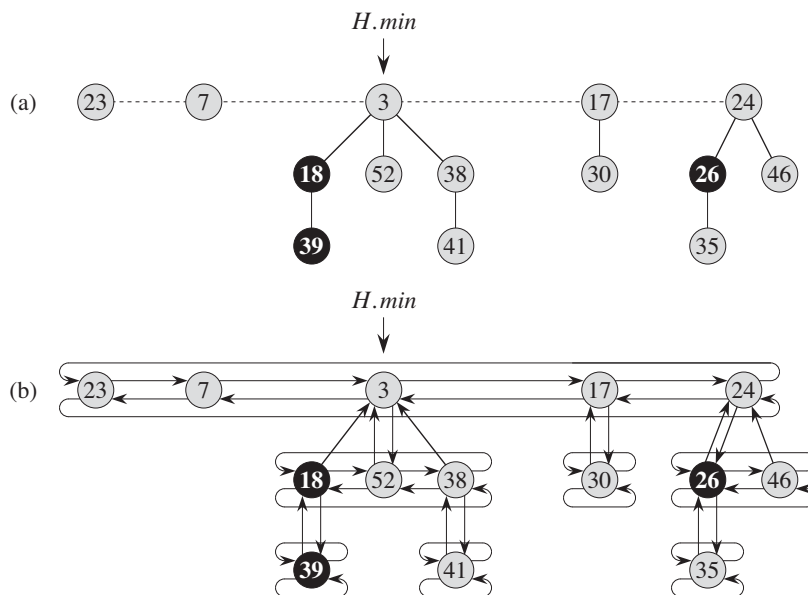


Figure 19.2 (a) A Fibonacci heap consisting of five min-heap-ordered trees and 14 nodes. The dashed line indicates the root list. The minimum node of the heap is the node containing the key 3. Black nodes are marked. The potential of this particular Fibonacci heap is $5 + 2 \cdot 3 = 11$. (b) A more complete representation showing pointers p (up arrows), $child$ (down arrows), and $left$ and $right$ (sideways arrows). The remaining figures in this chapter omit these details, since all the information shown here can be determined from what appears in part (a).

Circular, doubly linked lists (see Section 10.2) have two advantages for use in Fibonacci heaps. First, we can insert a node into any location or remove a node from anywhere in a circular, doubly linked list in $O(1)$ time. Second, given two such lists, we can concatenate them (or “splice” them together) into one circular, doubly linked list in $O(1)$ time. In the descriptions of Fibonacci heap operations, we shall refer to these operations informally, letting you fill in the details of their implementations if you wish.

Each node has two other attributes. We store the number of children in the child list of node x in $x.degree$. The boolean-valued attribute $x.mark$ indicates whether node x has lost a child since the last time x was made the child of another node. Newly created nodes are unmarked, and a node x becomes unmarked whenever it is made the child of another node. Until we look at the DECREASE-KEY operation in Section 19.3, we will just set all $mark$ attributes to FALSE.

We access a given Fibonacci heap H by a pointer $H.min$ to the root of a tree containing the minimum key; we call this node the **minimum node** of the Fibonacci

heap. If more than one root has a key with the minimum value, then any such root may serve as the minimum node. When a Fibonacci heap H is empty, $H.min$ is NIL.

The roots of all the trees in a Fibonacci heap are linked together using their *left* and *right* pointers into a circular, doubly linked list called the **root list** of the Fibonacci heap. The pointer $H.min$ thus points to the node in the root list whose key is minimum. Trees may appear in any order within a root list.

We rely on one other attribute for a Fibonacci heap H : $H.n$, the number of nodes currently in H .

Potential function

As mentioned, we shall use the potential method of Section 17.3 to analyze the performance of Fibonacci heap operations. For a given Fibonacci heap H , we indicate by $t(H)$ the number of trees in the root list of H and by $m(H)$ the number of marked nodes in H . We then define the potential $\Phi(H)$ of Fibonacci heap H by

$$\Phi(H) = t(H) + 2m(H) . \quad (19.1)$$

(We will gain some intuition for this potential function in Section 19.3.) For example, the potential of the Fibonacci heap shown in Figure 19.2 is $5 + 2 \cdot 3 = 11$. The potential of a set of Fibonacci heaps is the sum of the potentials of its constituent Fibonacci heaps. We shall assume that a unit of potential can pay for a constant amount of work, where the constant is sufficiently large to cover the cost of any of the specific constant-time pieces of work that we might encounter.

We assume that a Fibonacci heap application begins with no heaps. The initial potential, therefore, is 0, and by equation (19.1), the potential is nonnegative at all subsequent times. From equation (17.3), an upper bound on the total amortized cost provides an upper bound on the total actual cost for the sequence of operations.

Maximum degree

The amortized analyses we shall perform in the remaining sections of this chapter assume that we know an upper bound $D(n)$ on the maximum degree of any node in an n -node Fibonacci heap. We won't prove it, but when only the mergeable-heap operations are supported, $D(n) \leq \lfloor \lg n \rfloor$. (Problem 19-2(d) asks you to prove this property.) In Sections 19.3 and 19.4, we shall show that when we support DECREASE-KEY and DELETE as well, $D(n) = O(\lg n)$.