18 B-Trees

B-trees are balanced search trees designed to work well on disk drives or other direct-access secondary storage devices. B-trees are similar to red-black trees (Chapter 13), but they are better at minimizing the number of operations that access disks. (We often say just "disk" instead of "disk drive.") 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 drive used. B-trees are similar to red-black trees in that every n-node B-tree has height $O(\lg n)$, so that B-trees can implement many dynamic-set operations in $O(\lg n)$ time. But a B-tree has a larger branching factor than a red-black tree, so the base of the logarithm that expresses its height is larger, and hence its height can be considerably lower.

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. A search for a key in a B-tree makes an $(x.n + 1)$-way decision based on comparisons with the $x.n$ keys stored at node x. An internal node contains pointers to its children, but a leaf node does not.

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 drive differently from data structures designed to work in main random-access memory.

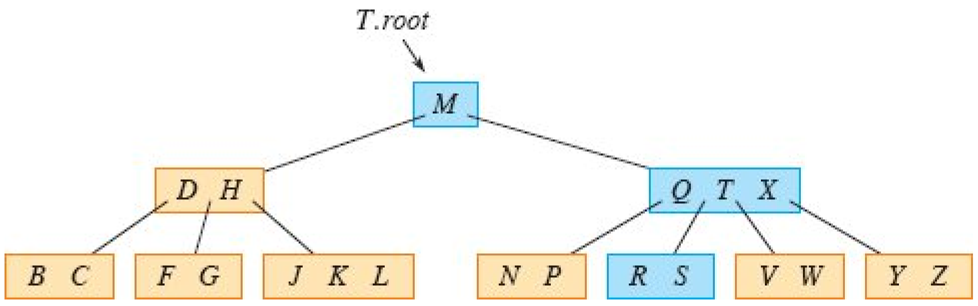


Figure 18.1 A B-tree whose keys are the consonants of English. An internal node x containing $x.n$ keys has $x.n + 1$ children. All leaves are at the same depth in the tree. The blue nodes are examined in a search for the letter R.

Data structures on secondary storage

Computer systems take advantage of various technologies that provide memory capacity. The ***main memory*** of a computer system normally 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 disk drives. Most computer systems also have ***secondary storage*** based on solid-state drives (SSDs) or magnetic disk drives. The amount of such secondary storage often exceeds the amount of primary memory by one to two orders of magnitude. SSDs have faster access times than magnetic disk drives, which are mechanical devices. In recent years, SSD capacities have increased while their prices have decreased. Magnetic disk drives typically have much higher capacities than SSDs, and they remain a more cost-effective means for storing massive amounts of information. Disk drives that store several terabytes[1] can be found for under $100.

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. The surface that passes underneath a given head when it is stationary is called a ***track***.

Although disk drives are cheaper and have higher capacity than main memory, they are much, much slower because they have moving mechanical parts. The mechanical motion has two components: platter rotation and arm movement. As of this writing, commodity disk drives rotate at speeds of 5400–15,000 revolutions per minute (RPM). Typical speeds are 15,000 RPM in server-grade drives, 7200 RPM in drives for desktops, and 5400 RPM 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 main memory. In other words, if a computer waits a full rotation for a particular item to come under the read/write head, it could access main memory more than 100,000 times during that span. The average wait is only half a rotation, but still, the difference in access times for main memory compared with disk drives is enormous. Moving the arms also takes some time. As of this writing, average access times for commodity disk drives are around 4 milliseconds.

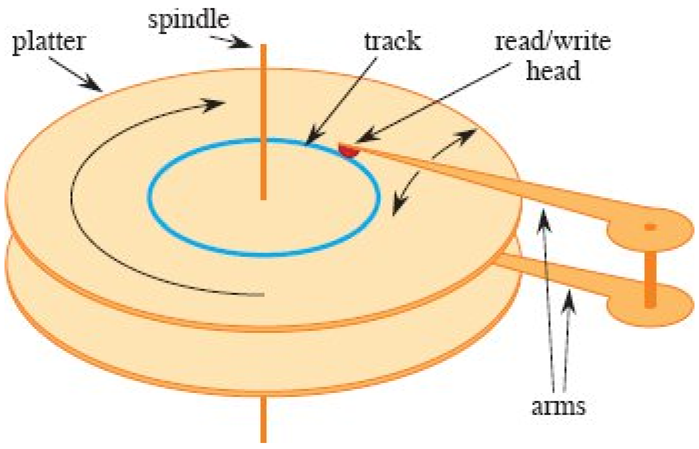


Figure 18.2 A typical magnetic disk drive. It consists of one or more platters covered with a magnetizable material (two platters are shown here) that rotate around a spindle. Each platter is read and written with a head, shown in red, at the end of an arm. Arms rotate around a common pivot axis. A track,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

In order to amortize the time spent waiting for mechanical movements, also known as ***latency***, disk drives access not just one item but several at a time. Information is divided into a number of equal-sized ***blocks*** of bits that appear consecutively within tracks, and each disk read or write is of one or more entire blocks.[2] Typical disk drives have block sizes running from 512 to 4096 bytes. Once the read/write head is positioned correctly and the platter has rotated to the beginning of the desired block, reading or writing a magnetic disk drive is entirely electronic (aside from the rotation of the platter), and the disk drive can quickly read or write large amounts of data.

Often, accessing a block of information and reading it from a disk drive takes longer than processing all the information read. For this reason, in this chapter we'll 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 blocks of information that need to be read from or written to the disk drive. Although 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 platters—the number of blocks read or written provides a good first-order approximation of the total time spent accessing the disk drive.

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 blocks from disk into main memory as needed and write back onto disk the blocks that have changed. B-tree algorithms keep only a constant number of blocks in main memory at any time, and thus the size of main memory does not limit the size of B-trees that can be handled.

B-tree procedures need to be able to read information from disk into main memory and write information from main memory to disk. Consider some object x. If x is currently in the computer's main memory, then the code can refer to the attributes of x as usual: $x.key$, for example. If x resides on disk, however, then the procedure must perform the operation DISK-READ(x) to read the block containing object x into main memory before it can refer to x's attributes. (Assume that if x is already in main memory, then DISK-READ(x) requires no disk accesses: it is a "no-op.") Similarly, procedures call DISK-WRITE(x) to save any changes that have been made to the attributes of object x by writing to disk the block containing x. Thus, 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 blocks in main memory at any one time. Our B-tree algorithms assume that the system automatically flushes from main memory blocks that are no longer in use.

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 block, and this size limits the number of children a B-tree node can have.

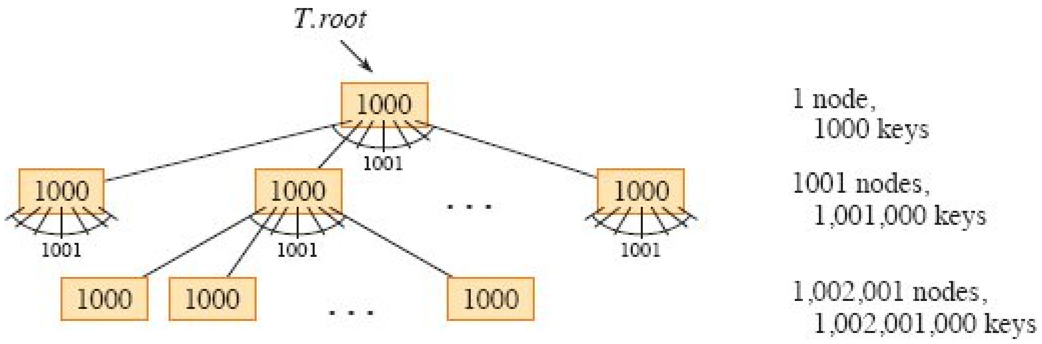


Figure 18.3 A B-tree of height 2 containing over one billion keys. Shown inside each node x is $x.n$, the number of keys in x. Each internal node and leaf contains 1000 keys. This B-tree has 1001 nodes at depth 1 and over one million leaves at depth 2.

Large B-trees stored on disk drives often have branching factors between 50 and 2000, depending on the size of a key relative to the size of a block. 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, if the root node is kept permanently in main memory, at most two disk accesses suffice to find any key in this tree.

18.1 Definition of B-trees

To keep things simple, let's 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, you might actually store with each key just a pointer to another disk block 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 with root $T.root$ 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, $x.key_1$, $x.key_2$, … , $x.key_{x.n}$, stored in monotonically increasing order, so that $x.key_1 \leq x.key_2 \leq \cdots \leq x.key_{x.n}$,

   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 $x.c_1$, $x.c_2$, … , $x.c_{x.n+1}$ 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, expressed 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.[3]

The simplest B-tree occurs when $t = 2$. Every internal node then has either 2, 3, or 4 children, and it is 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. The following theorem bounds the worst-case height of a B-tree.

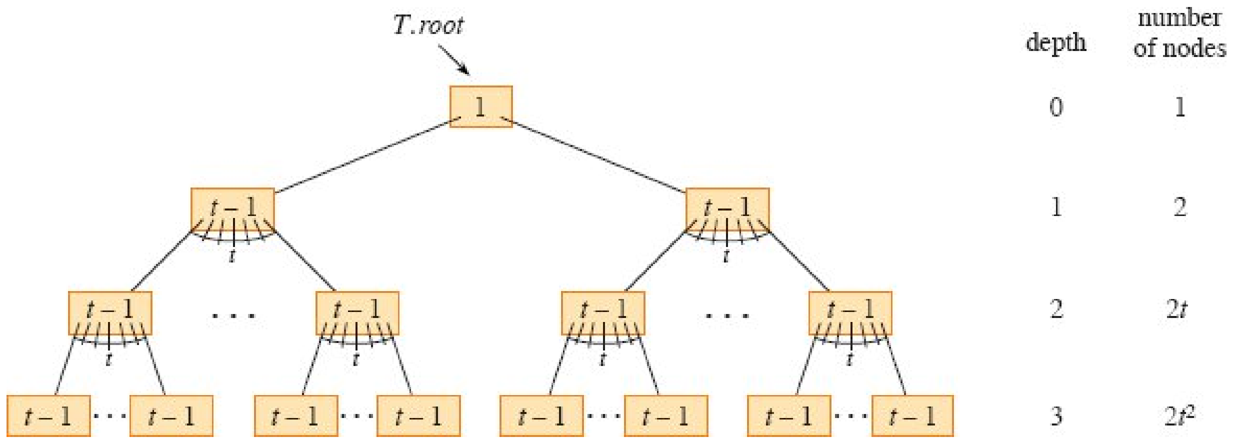


Figure 18.4 A B-tree of height 3 containing a minimum possible number of keys. Shown inside each node x is $x.n$.

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 By definition, the root of a nonempty B-tree T contains at least one key, and all other nodes contain at least $t - 1$ keys. Let h be the height of T. Then T contains 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. The number n of keys therefore satisfies the inequality

$$\begin{aligned} n &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} \\ &= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) \quad \text{(by equation (A.6) on page 1142)} \\ &= 2t^h - 1\,, \end{aligned}$$

so that $t^h \leq (n + 1)/2$. Taking base-t logarithms of both sides proves the theorem.

■

You can see the power of B-trees as compared with red-black trees. Although the height of the tree grows as $O(\log n)$ 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 examining an arbitrary node in a tree usually entails accessing the disk, B-trees avoid a substantial number of disk accesses.

Exercises

18.1-1
Why isn't a minimum degree of t = 1 allowed?

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 store the keys 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 results if each black node in a red-black tree absorbs its red children, incorporating their children with its own.

18.2 Basic operations on B-trees

This section presents the details of the operations B-TREE-SEARCH, B-TREE-CREATE, and B-TREE-INSERT. These procedures observe two conventions:

- The root of the B-tree is always in main memory, so that no procedure ever needs to perform a DISK-READ on the root. If any changes to the root node occur, however, then DISK-WRITE must be called on the root.
- Any nodes that are passed as parameters must already have had a DISK-READ operation performed on them.

The procedures 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, the search makes a multiway branching decision according to the number of the node's children. More precisely, at each internal node x, the search makes an $(x.n + 1)$-way branching decision.

The procedure B-TREE-SEARCH generalizes the TREE-SEARCH procedure defined for binary search trees on page 316. It 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, then 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 $i \le x.n$ and $k > x.key_i$
3     $i = i + 1$
4 if $i \le x.n$ and $k == x.key_i$
5     return $(x, i)$
6 elseif $x.leaf$
7     returnNIL
8 else DISK-READ($x.c_i$)
9     return B-TREE-SEARCH($x.c_i$, $k$)
```

Using a linear-search procedure, lines 1–3 of B-TREE-SEARCH find the smallest index i such that $k \le x.key_i$, or else they set i to $x.n + 1$. Lines 4–5 check to see whether the search has discovered the key, returning if it has. Otherwise, if x is a leaf, then line 7 terminates the search unsuccessfully, and if x is an internal node, lines 8–9 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 blue nodes are those examined 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 blocks, 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, first use the B-TREE-CREATE procedure on the next page to create an empty root node and then call the B-TREE-INSERT procedure on page 508 to add new keys. Both of these procedures use an auxiliary procedure ALLOCATE-NODE, whose pseudocode we omit and which allocates one disk block to be used as a new node in $O(1)$ time. 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 requires $O(1)$ disk operations and $O(1)$ CPU time.

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

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, you search for the leaf position at which to insert the new key. With a B-tree, however, you cannot simply create a new leaf node and insert it, as the resulting tree would fail to be a valid B-tree. Instead, you insert the new key into an existing leaf node. Since you cannot insert a key into a leaf node that is full, you need 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, you must split it before you can insert the new key, and thus you could end up splitting full nodes all the way up the tree.

To avoid having to go back up the tree, just split every full node you encounter as you go down the tree. In this way, whenever you need to split a full node, you are assured that its parent is not full. Inserting a key into a B-tree then requires only a single pass down the tree from the root to a leaf.

Splitting a node in a B-tree

The procedure B-TREE-SPLIT-CHILD on the facing page takes as input a *nonfull* internal node x (assumed to reside in main memory) and an index i such that $x.c_i$ (also assumed to reside in main memory) is a *full* child of x. The procedure splits this child in two and adjusts x so that it has an additional child. To split a full root, you first need to make the root a child of a new empty root node, so that you can use B-TREE-SPLIT-CHILD. The tree thus grows in height by 1: splitting is the only means by which the tree grows taller.

```
B-TREE-SPLIT-CHILD($x$, $i$)
$y = x.c_i$                          // full node to split
$z$ = ALLOCATE-NODE()                // $z$ will take half of $y$
$z.leaf = y.leaf$
$z.n = t - 1$
for $j = 1$ to $t - 1$               // $z$ gets $y$'s greatest keys …
    $z.key_j = y.key_{j+t}$
if not $y.leaf$
    for $j = 1$ to $t$               // … and its corresponding children
        $z.c_j = y.c_{j+t}$
$y.n = t - 1$                        // $y$ keeps $t - 1$ keys
for $j = x.n + 1$ downto $i + 1$     // shift $x$'s children to the right …
    $x.c_{j+1} = x.c_j$
$x.c_{i+1} = z$                      // … to make room for $z$ as a child
for $j = x.n$ downto $i$             // shift the corresponding keys in $x$
    $x.key_{j+1} = x.key_j$
$x.key_i = y.key_t$                  // insert $y$'s median key
$x.n = x.n + 1$                      // $x$ has gained a child
DISK-WRITE($y$)
DISK-WRITE($z$)
DISK-WRITE($x$)
```

Figure 18.5 illustrates how a node splits. B-TREE-SPLIT-CHILD splits the full node $y = x.c_i$ about its median key (S in the figure), 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.

B-TREE-SPLIT-CHILD works by straightforward cutting and pasting. Node x is the parent of the node y being split, which is x's ith child (set in line 1). Node y originally has $2t$ children and $2t - 1$ keys, but splitting reduces y to t children and $t - 1$ keys. The t largest children and $t - 1$ keys of node y move over to node z, which 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 node x that separates the pointers to nodes y and z.

Lines 2–9 create node z and give it the largest $t - 1$ keys and, if y and z are internal nodes, the corresponding t children of y. Line 10 adjusts the key count for y. Then, lines 11–17 shift keys and child pointers in x to the right in order to make room for x's new child, insert z as a new 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 blocks. The CPU time used by B-TREE-SPLIT-CHILD is $\Theta(t)$, due to the **for** loops in lines 5–6 and 8–9. (The **for** loops in lines 11–12 and 14–15 also run for $O(t)$ iterations.) The procedure performs $O(1)$ disk operations.

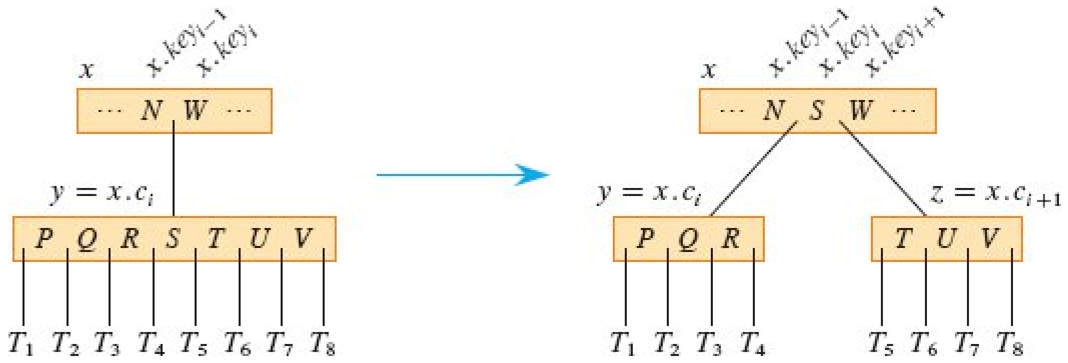


Figure 18.5 Splitting a node with $t = 4$. Node $y = x.c_i$ splits into two nodes, y and z, and the median key S of y moves up into y's parent.

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

Inserting a key k into a B-tree T of height h requires just a single pass down the tree and $O(h)$ 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. If the root is full, B-TREE-INSERT splits it by calling the procedure B-TREE-SPLIT-ROOT on the facing page.

```
B-TREE-INSERT($T$, $k$)

1 $r = T.root$
2 **if** $r.n == 2t - 1$
3     $s$ = B-TREE-SPLIT-ROOT($T$)
4     B-TREE-INSERT-NONFULL($s$, $k$)
5 **else** B-TREE-INSERT-NONFULL($r$, $k$)
```

B-TREE-INSERT works as follows. If the root is full, then line 3 calls B-TREE-SPLIT-ROOT in line 3 to split it. A new node s (with two children) becomes the root and is returned by B-TREE-SPLIT-ROOT. Splitting the root, illustrated in Figure 18.6, is the only way to increase the height of a B-tree. Unlike a binary search tree, a B-tree increases in height at the top instead of at the bottom. Regardless of whether the root split, B-TREE-INSERT finishes by calling B-TREE-INSERT-NONFULL to insert key k into the tree rooted at the nonfull root node, which is either the new root (the call in line 4) or the original root (the call in line 5).

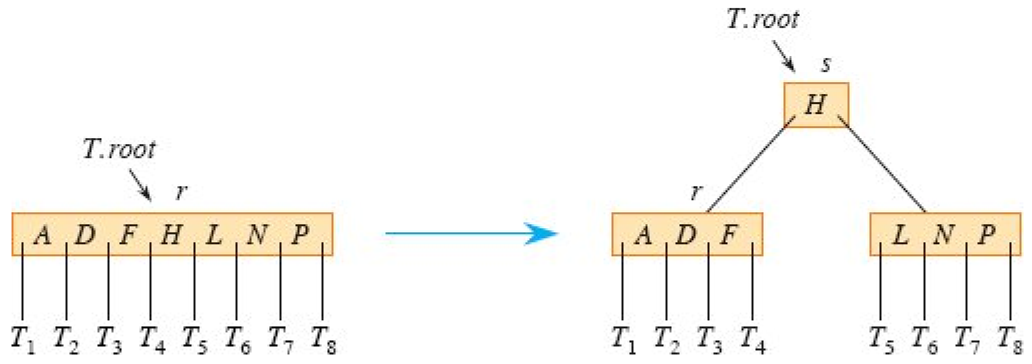


Figure 18.6 Splitting the root with $t = 4$. Root node r splits in two, and a new root node s is created. The new root contains the median key of r and has the two halves of r as children. The B-tree grows in height by one when the root is split. A B-tree's height increases only when the root splits.

```
B-TREE-SPLIT-ROOT(T)
1 s = ALLOCATE-NODE()
2 s.leaf = FALSE
3 s.n = 0
4 s.c_1 = T.root
5 T.root = s
6 B-TREE-SPLIT-CHILD(s, 1)
7 return s
```

The auxiliary procedure B-TREE-INSERT-NONFULL on page 511 inserts key k into node x, which is assumed to be nonfull when the procedure is called. B-TREEINSERT-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 operation of B-TREE-INSERT and the recursive operation of B-TREE-INSERT-NONFULL guarantee that this assumption is true.

Figure 18.7 illustrates the various cases of how B-TREE-INSERT-NONFULL inserts a key into a B-tree. Lines 3–8 handle the case in which x is a leaf node by inserting key k into x, shifting to the right all keys in x that are greater than k. If x is not a leaf node, then k should go into the appropriate leaf node in the subtree rooted at internal node x. Lines 9–11 determine the child $x.c_i$ to which the recursion descends. Line 13 detects whether the recursion would descend to a full child, in which case line 14 calls B-TREE-SPLIT-CHILD to split that child into two nonfull children, and lines 15–16 determine which of the two children is the correct one to descend to. (Note that DISK-READ($x.c_i$) is not needed after line 16 increments i, since the recursion descends in this case to a child that was just created by 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.

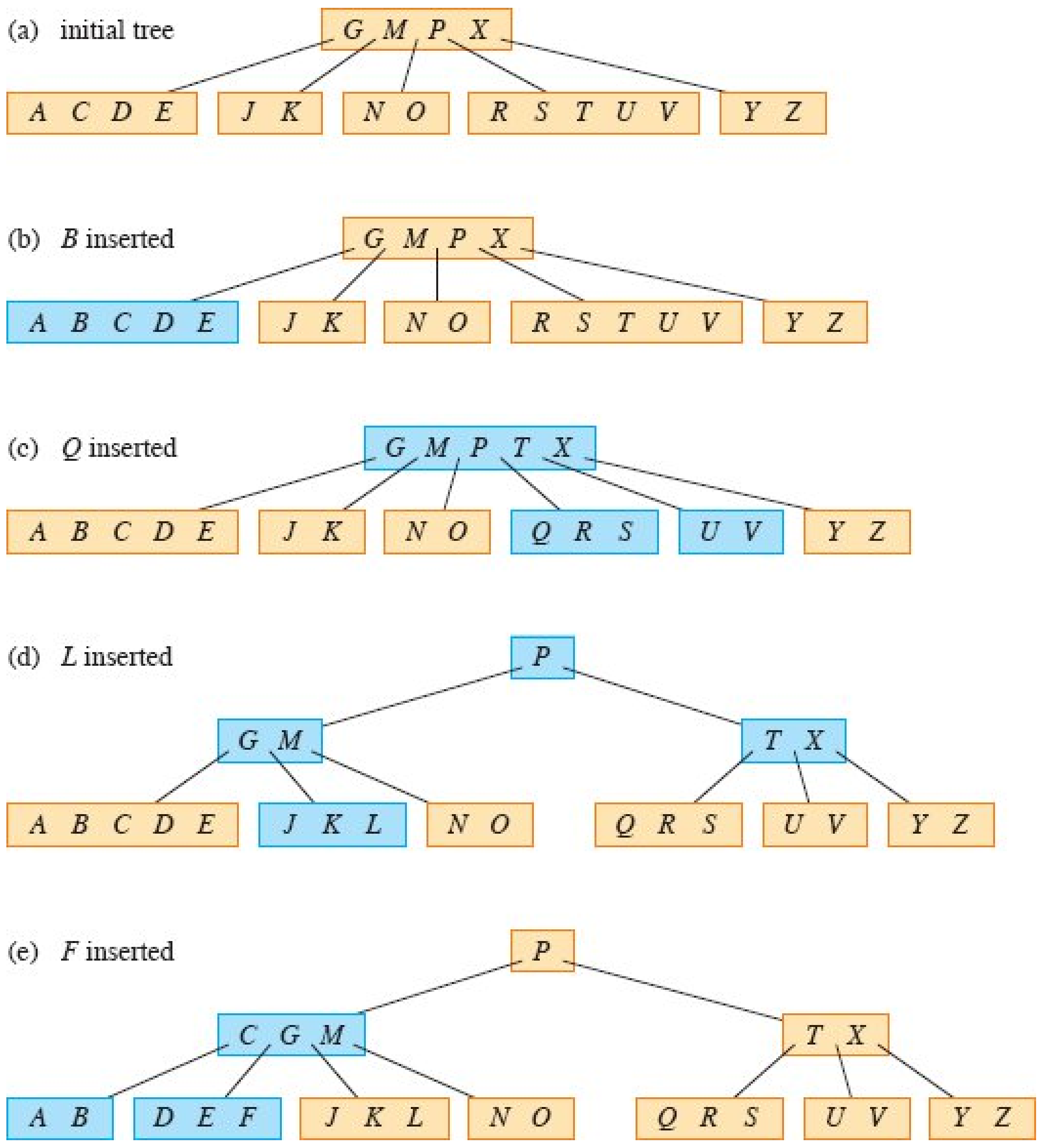


Figure 18.7 Inserting keys into a B-tree. The minimum degree *t* for this B-tree is 3, so that a node can hold at most 5 keys. Blue nodes are modified by the insertion process. **(a)** The initial tree for this example. **(b)** The result of inserting *B* into the initial tree. This case is a simple insertion into a leaf node. **(c)** The result of inserting *Q* into the previous tree. The node *RST U V* splits into two nodes containing *RS* and *U V*, 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).

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

```
i = x.n
if x.leaf                              // inserting into a leaf?
    while i ≥ 1 and k < x.key_i        // shift keys in x to make room for k
        x.key_{i+1} = x.key_i
        i = i − 1
    x.key_{i+1} = k                    // insert key k in x
    x.n = x.n + 1                      // now x has 1 more key
    DISK-WRITE(x)
else while i ≥ 1 and k < x.key_i       // find the child where k belongs
        i = i − 1
    i = i + 1
    DISK-READ(x.c_i)
    if x.c_i.n == 2t − 1               // split the child if it's full
        B-TREE-SPLIT-CHILD(x, i)
        if k > x.key_i                 // does k go into x.c_i or x.c_{i+1}?
            i = i + 1
    B-TREE-INSERT-NONFULL(x.c_i, k)
```

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 at each level of the tree. The total CPU time used is $O(t)$ in each level of the tree, or $O(th) = O(t \log_t n)$ overall. Since B-TREE-INSERT-NONFULL is tail-recursive, you can instead implement it with a **while** loop, thereby demonstrating that the number of blocks 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 block that is already in memory. A redundant DISK-WRITE writes to disk a block of information that is identical to what is already stored there.)

18.2-3
Professor Bunyan asserts that the B-TREE-INSERT procedure always results in a B-tree with the minimum possible height. Show that the professor is mistaken by proving that with $t = 2$ and the set of keys $\{1, 2, \ldots, 15\}$, there is no insertion sequence that results in a B-tree with the minimum possible height.

★ ***18.2-4***
If you insert the keys $\{1, 2, \ldots, n\}$ 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 block size. Show how to modify the procedures for creating and inserting into a B-tree to handle this variation.

18.2-6
Suppose that you implement B-TREE-SEARCH to use binary search rather than linear search within each node. Show that this change makes the required CPU time $O(\lg n)$, independent of how t might be chosen as a function of n.

18.2-7
Suppose that disk hardware allows you to choose the size of a disk block arbitrarily, but that the time it takes to read the disk block is $a+bt$, where a and b are specified constants and t is the minimum degree

for a B-tree using blocks 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

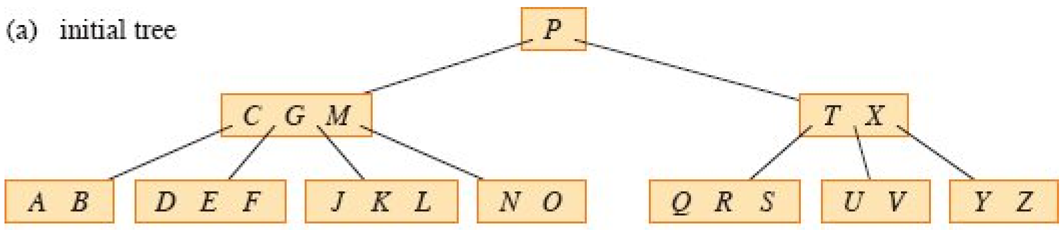
Deletion from a B-tree is analogous to insertion but a little more complicated, because you can delete a key from any node—not just a leaf—and when you delete a key from an internal node, you must rearrange the node's children. As in insertion, you must guard against deletion producing a tree whose structure violates the B-tree properties. Just as a node should not get too big due to insertion, a node must not get too small during deletion (except that the root is allowed to have fewer than the minimum number $t - 1$ of keys). And just as a simple insertion algorithm might have to back up if a node on the path to where the key is to be inserted is 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. Unlike the procedures TREE-DELETE on page 325 and RB-DELETE on page 348, which are given the node to delete—presumably as the result of a prior search—B-TREE-DELETE combines the search for key k with the deletion process. Why do we combine search and deletion in B-TREE-DELETE? Just as B-TREE-INSERT prevents any node from becoming overfull (having more than $2t - 1$ keys) while making a single pass down the tree, B-TREE-DELETE prevents any node from becoming underfull (having fewer than $t - 1$ keys) while also making a single pass down the tree, searching for and ultimately deleting the key.

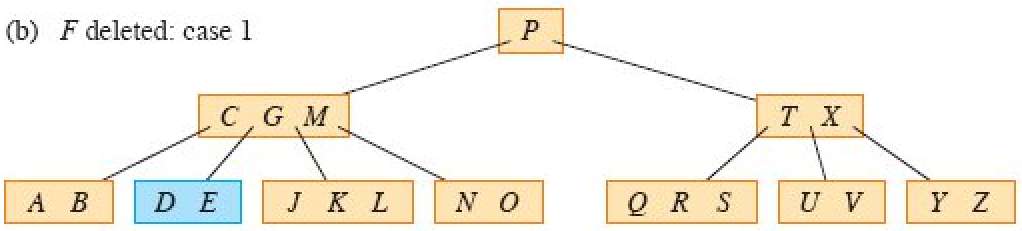
To prevent any node from becoming underfull, the design of B-TREE-DELETE guarantees that whenever it calls itself recursively on a node x, the number of keys in x is at least the minimum degree t at the time of the call. (Although the root may have fewer than t keys and a recursive call may be made *from* the root, no recursive call is made *on* the root.) This condition requires one more key than the minimum required by the usual B-tree conditions, and so a key might have to be moved from x into one of its child nodes (still leaving x with at least the minimum $t - 1$ keys) before a recursive call is made on that child, thus allowing deletion to occur in one downward pass without having to traverse back up the tree.

We describe how the procedure B-TREE-DELETE(T, k) deletes a key k from a B-tree T instead of presenting detailed pseudocode. We examine three cases, illustrated in Figure 18.8. The cases are for when the search arrives at a leaf, at an internal node containing key k, and at an internal node not containing key k. As mentioned above, in all three cases node x has at least t keys (with the possible exception of when x is the root). Cases 2 and 3—when x is an internal node—guarantee this property as the recursion descends through the B-tree.

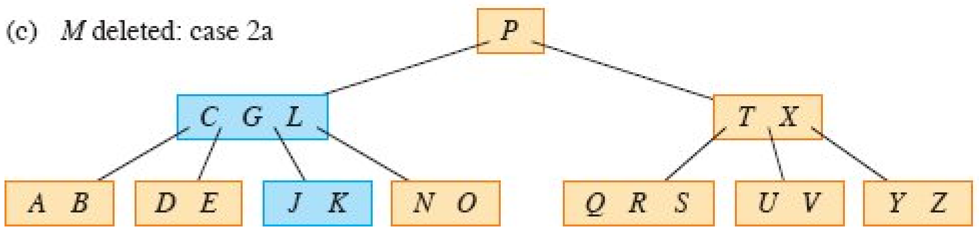
(a) initial tree



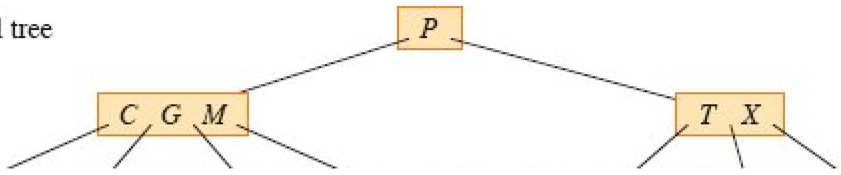
(b) F deleted: case 1



(c) M deleted: case 2a



(a) initial tree



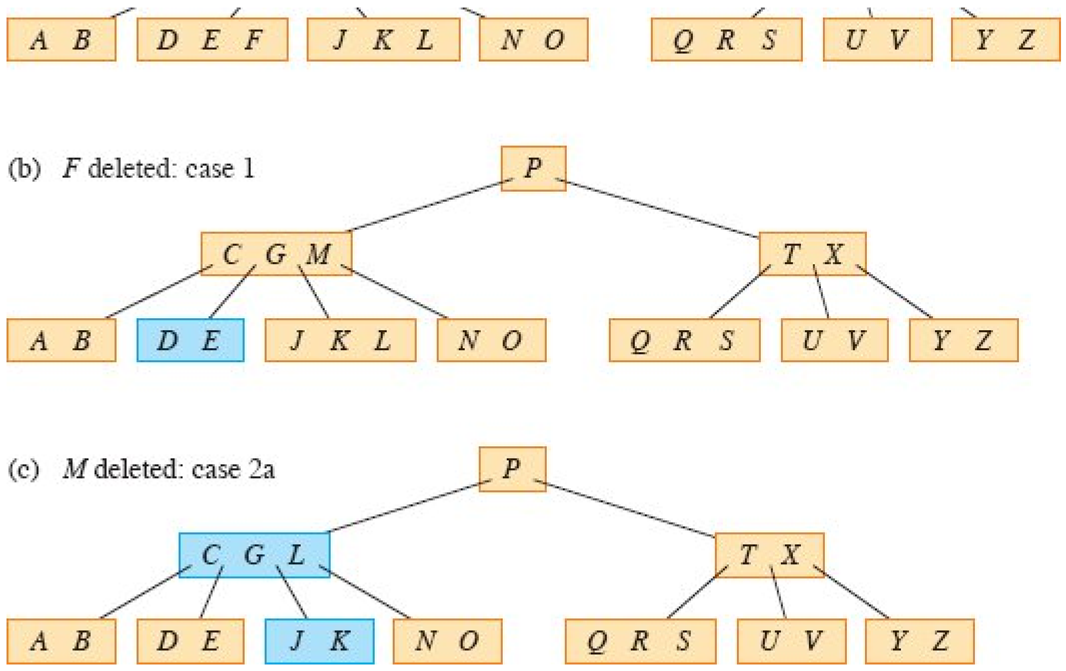


Figure 18.8 Deleting keys from a B-tree. The minimum degree for this B-tree is $t = 3$, so that, other than the root, every node must have at least 2 keys. Blue nodes are those that are modified by the deletion process. **(a)** The B-tree of Figure 18.7(e). **(b)** Deletion of F, which is case 1: simple deletion from a leaf when all nodes visited during the search (other than the root) have at least $t = 3$ keys. **(c)** Deletion of M, which is case 2a: the predecessor L of M moves up to take M's position. **(d)** Deletion of G, which is case 2c: push G down to make node $DEGJK$ and then delete G from this leaf (case 1). **(e)** Deletion of D, which is case 3b: since the recursion cannot descend to node CL because it has only 2 keys, push P down and merge it with CL and TX to form $CLP\ TX$. Then delete D from a leaf (case 1). **(e^0)** After (e), delete the empty root. The tree shrinks in height by 1. **(f)** Deletion of B, which is case 3a: C moves to fill B's position and E moves to fill C's position.

Case 1: *The search arrives at a leaf node x.* If x contains key k, then delete k from x. If x does not contain key k, then k was not in the B-tree and nothing else needs to be done.

Case 2: *The search arrives at an internal node x that contains key k.* Let $k = x.key_i$. One of the following three cases applies, depending on the number of keys in $x.c_i$ (the child of x that precedes k) and $x.c_{i+1}$ (the child of x that follows k).

Case 2a: *$x.c_i$ has at least t keys.* Find the predecessor k' of k in the subtree rooted at $x.c_i$. Recursively delete k' from $x.c_i$, and replace k by k' in x. (Key k' can be found and deleted in a single downward pass.)

Case 2b: *$x.c_i$ has $t - 1$ keys and $x.c_{i+1}$has at least t keys.* This case is symmetric to case 2a. Find the successor k' of k in the subtree rooted at $x.c_{i+1}$. Recursively delete k' from $x.c_{i+1}$, and replace k by k' in x. (Again, finding and deleting k' can be done in a single downward pass.)

Case 2c**:** *Both $x.c_i$ and $x.c_{i+1}$have $t - 1$ keys.* Merge k and all of $x.c_{i+1}$ into $x.c_i$, so that x loses both k and the pointer to $x.c_{i+1}$, and $x.c_i$ now contains $2t - 1$ keys. Then free $x.c_{i+1}$ and recursively delete k from $x.c_i$.

Case 3**:** *The search arrives at an internal node x that does not contain key k.* Continue searching down the tree while ensuring that each node visited has at least t keys. To do so, 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 case 3a or 3b as necessary to guarantee descending to a node containing at least t keys. Then finish by recursing on the appropriate child of x.

Case 3a**:** *$x.c_i$ 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 $x.c_i$.

Case 3b**:** *$x.c_i$ and each of $x.c_i$'s immediate siblings have $t - 1$ keys.* (It is possible for $x.c_i$ to have either one or two siblings.) 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.

In cases 2c and 3b, if node x is the root, it could end up having no keys. When this situation occurs, then x is deleted, and x's only child $x.c_1$ becomes the new root of the tree. This action decreases the height of the tree by one and preserves the property that the root of the tree contains at least one key (unless the tree is empty).

Since most of the keys in a B-tree are in the leaves, deletion operations often end up deleting 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 x, however, the procedure might make a downward pass through the tree to find the key's predecessor or successor and then return to node x to replace the key with its predecessor or successor (cases 2a and 2b). Returning to node x does not require a traversal through all the levels between x and the node containing the predecessor or successor, however, since the procedure can just keep a pointer to x and the key position within x and put the predecessor or successor key directly there.

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 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. Maintain in memory a stack pointer, which is the disk address of the top element on the stack. Indexing block numbers and word offsets within blocks from 0, if the pointer has value p, the top element is the (p mod m)th word on block $\lfloor p/m \rfloor$ of the disk, where m is the number of words per block.

To implement the PUSH operation, increment the stack pointer, read the appropriate block into memory from disk, copy the element to be pushed to the appropriate word on the block, and write the block back to disk. A POP operation is similar. Read in the appropriate block from disk, save the top of the stack, decrement the stack pointer, and return the saved value. You need not write back the block, since it was not modified, and the word in the block that contained the popped value is ignored.

As in the analyses of B-tree operations, two costs matter: the total number of disk accesses and the total CPU time. A disk access also incurs a cost in CPU time. In particular, any disk access to a block 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 you keep one block of the stack in memory. (You also maintain a small amount of memory to record which block is currently in memory.) You can perform a stack operation only if the relevant disk block resides in memory. If necessary, you can write the block currently in memory to the disk and read the new block from the disk into memory. If the relevant disk block 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 you now implement the stack by keeping two blocks in memory (in addition to a small number of words for bookkeeping).

d. Describe how to manage the stack blocks 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 $x'.key < x.key < x''.key$ for any $x' \in S'$ and $x'' \in S''$. 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 another set S'' that consists of all elements in $S - \{x\}$ whose keys are greater than $x.key$. This problem investigates how to implement these operations on 2-3-4 trees (B-trees with $t = 2$). 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 $\{T'_0, T'_1, \ldots, T'_m\}$ and a set of keys $\{k'_1, k'_2, \ldots, k'_m\}$ such that $y < k'_i < z$ for $i = 1, 2, \ldots, m$ and 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 [261], Aho, Hopcroft, and Ullman [5], and Sedgewick and Wayne [402] give further discussions of balanced-tree schemes and B-trees. Comer [99] provides a comprehensive survey of B-trees. Guibas and Sedgewick [202] 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 [39] introduced B-trees in 1972 with no explanation of their choice of name.

Bender, Demaine, and Farach-Colton [47] 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.

---

[1] When specifying disk capacities, one terabyte is one trillion bytes, rather than 2_{40} bytes.

[2] SSDs also exhibit greater latency than main memory and access data in blocks.

[3] Another 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.