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 are designed so that only a constant number of pages are 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 fields of the object as usual: $key[x]$, for example. If the object referred to by x resides on disk, however, then we must perform the operation DISK-READ(x) to read object x into main memory before we can refer to its fields. (We assume that if x is already in main memory, then DISK-READ(x) requires no disk accesses; it is a "no-op.") Similarly, the operation DISK-WRITE(x) is used to save any changes that have been made to the fields of object x. That is, the typical pattern for working with an object is as follows:

- $x \leftarrow$ a pointer to some object
- DISK-READ(x)
- operations that access and/or modify the fields of x

- DISK-WRITE(x) ▷ Omitted if no fields of x were changed.
- other operations that access but do not modify fields of x

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

Since in most systems the running time of a B-tree algorithm is determined mainly by the number of DISK-READ and DISK-WRITE operations it performs, it is sensible to use these operations efficiently by having them read or write as much information as possible. Thus, a B-tree node is usually as large as a whole disk page. The number of children a B-tree node can have is therefore limited by the size of a disk page.

For a large B-tree stored on a disk, branching factors between 50 and 2000 are often used, 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 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 the root node can be kept permanently in main memory, only *two* disk accesses at most are required to find any key in this tree!
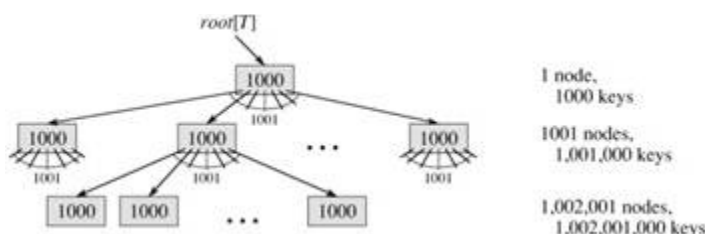

Figure 18.3: A B-tree of height 2 containing over one billion keys. Each internal node and leaf contains 1000 keys. There are 1001 nodes at depth 1 and over one million leaves at depth 2. Shown inside each node x is $n[x]$, the number of keys in x.

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 is stored 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 *root*[T]) having the following properties:

1. Every node x has the following fields:
   a. $n[x]$, the number of keys currently stored in node x,
   b. the $n[x]$ keys themselves, stored in nondecreasing order, so that $key_1[x] \leq key_2[x] \leq \cdots \leq key_{n[x]}[x]$,
   c. *leaf*[x], 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 $n[x]+1$ pointers $c_1[x], c_2[x], ..., c_{n[x]+1}[x]$ to its children. Leaf nodes have no children, so their c_i fields are undefined.
3. The keys $key_i[x]$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $c_i[x]$, then

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

4. All leaves have the same depth, which is the tree's height h.
5. There are lower and upper bounds on the number of keys a node can contain. These bounds can be 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 can contain at most $2t$ - 1 keys. Therefore, an internal node can have at most $2t$ children. We say that a node is **full** if it contains exactly $2t$ - 1 keys. [1]

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 are typically used.

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 $t \geq 2$,

Proof If a B-tree has height h, the root contains at least one key and all other nodes contain at least $t - 1$ keys. Thus, there are 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 there are at least $2t^{h-1}$ Figure 18.4 illustrates such a tree for $h = 3$. Thus, the number n of keys satisfies the inequality

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

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 to 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. Since examining an arbitrary node in a tree usually requires a disk access, the number of disk accesses is substantially reduced.

Exercises 18.1-1

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

Exercises 18.1-2

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

Exercises 18.1-3

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

Exercises 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?

Exercises 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.

[1]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.

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 a DISK-READ on the root is never required; a DISK-WRITE of the root is required, 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 ($n[x] + 1$)-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($root[T]$, 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 $key_i[y] = k$. Otherwise, the value NIL is returned.

```
B-TREE-SEARCH(x, k)
1 i ← 1
2 while i ≤ n[x] and k > key_i[x]
3         do i ← i + 1
4 if i ≤ n[x] and k = key_i[x]
5     then return (x, i)
6 if leaf[x]
7     then return NIL
8     else DISK-READ(c_i[x])
9             return B-TREE-SEARCH(c_i[x], k)
```

Using a linear-search procedure, lines 1-3 find the smallest index i such that $k \leq key_i[x]$, or else they set i to $n[x] + 1$. Lines 4-5 check to see if we have now discovered the key, returning if we have. 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

Figure 18.1 illustrates the operation of B-TREE-SEARCH; the lightly shaded nodes are 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 path downward from the root of the tree. The number of disk pages accessed by B-TREE-SEARCH is therefore $\Theta(h) = \Theta(\log_t n)$, where h is the height of the B-tree and n is the number of keys in the B-tree. Since $n[x] < 2t$, the time taken by the **while** loop of lines 2-3 within each node is $O(t)$, 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 ← ALLOCATE-NODE()
2   leaf[x] ← TRUE
3   n[x] ← 0
4   DISK-WRITE(x)
5   root[T] ← 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** $key_t[y]$ into two nodes having $t - 1$ 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, it must be split before the new key can be inserted, and thus this need to split full nodes can propagate 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), an index i, and a node y (also assumed to be in main memory) such that $y = c_i[x]$ 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;

Figure 18.5 illustrates this process. The full node y is split about its median key S, which is moved up into y's parent node x. Those keys in y that are greater than the median key are placed in a new node z, which is made a new child of x.
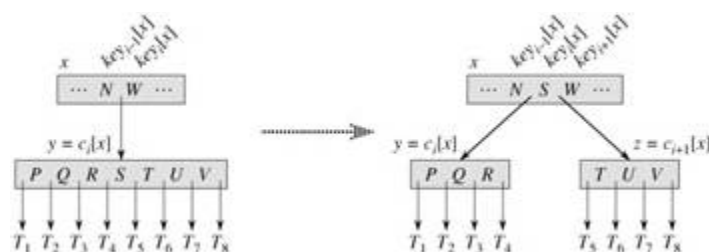

Figure 18.5: Splitting a node with $t = 4$. Node y is split into two nodes, y and z, and the median key S of y is moved up into y's parent.

```
B-TREE-SPLIT-CHILD(x, i, y)
 1   z ← ALLOCATE-NODE()
 2   leaf[z] ← leaf[y]
 3   n[z] ← t - 1
 4   for j ← 1 to t - 1
 5        do keyj[z] ← keyj+t[y]
 6   if not leaf[y]
 7      then for j ← 1 to t
 8              do cj[z] ← cj+t[y]
 9   n[y] ← t - 1
10   for j ← n[x] + 1 downto i + 1
11        do cj+1[x] ← cj[x]
12   ci+1[x] ← z
13   for j ← n[x] downto i
14        do keyj+1[x] ← keyj[x]
15   keyi[x] ← keyt[y]
16   n[x] ← n[x] + 1
17   DISK-WRITE(y)
```

```
18  DISK-WRITE(z)
19  DISK-WRITE(x)
```

B-TREE-SPLIT-CHILD works by straightforward "cutting and pasting." Here, y is the ith child of x and is the node being split. Node y originally has $2t$ children ($2t$ - 1 keys) but is reduced to t children (t - 1 keys) by this operation. Node z "adopts" the t largest children (t - 1 keys) of 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 1-8 create node z and give it the larger t - 1 keys and corresponding t children of y. Line 9 adjusts the key count for y. Finally, lines 10-16 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 17-19 write out all modified disk pages. The CPU time used by B-TREE-SPLIT-CHILD is $\Theta(t)$, due to the loops on lines 4-5 and 7-8. (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 ← root[T]
 2  if n[r] = 2t - 1
 3     then s ← ALLOCATE-NODE()
 4          root[T] ← s
 5          leaf[s] ← FALSE
 6          n[s] ← 0
 7          c₁[s] ← r
 8          B-TREE-SPLIT-CHILD(s, 1, r)
 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 is split and a new node s (having two children) becomes the root. Splitting the root is the only way to increase the 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 perform the insertion of key k in 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.
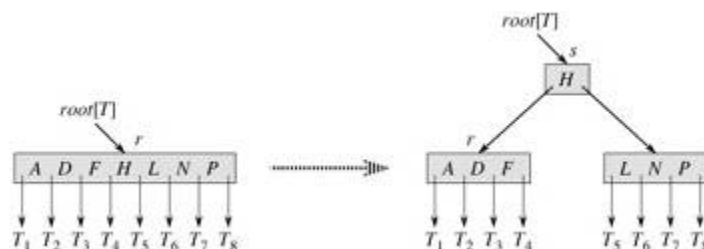

Figure 18.6: Splitting the root with $t = 4$. Root node r is split in two, and a new root node s is created. The new root contains the median key of r and has the two halves of r as children. The B-tree grows in height by one when the root is split.

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.

```
B-TREE-INSERT-NONFULL(x, k)
 1  i ← n[x]
 2  if leaf[x]
 3      then while i ≥ 1 and k < key_i[x]
 4              do key_{i+1}[x] ← key_i[x]
 5                  i ← i - 1
 6              key_{i+1}[x] ← k
 7              n[x] ← n[x] + 1
 8              DISK-WRITE(x)
 9      else while i ≥ 1 and k < key_i[x]
10              do i ← i - 1
11              i ← i + 1
12              DISK-READ(c_i[x])
13              if n[c_i[x]] = 2t - 1
14                  then B-TREE-SPLIT-CHILD(x, i, c_i[x])
15                      if k> key_i[x]
16                          then i ← i + 1
17              B-TREE-INSERT-NONFULL(c_i[x], 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 DISK-READ($c_i[x]$) after line 16 increments i, since the recursion will descend 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

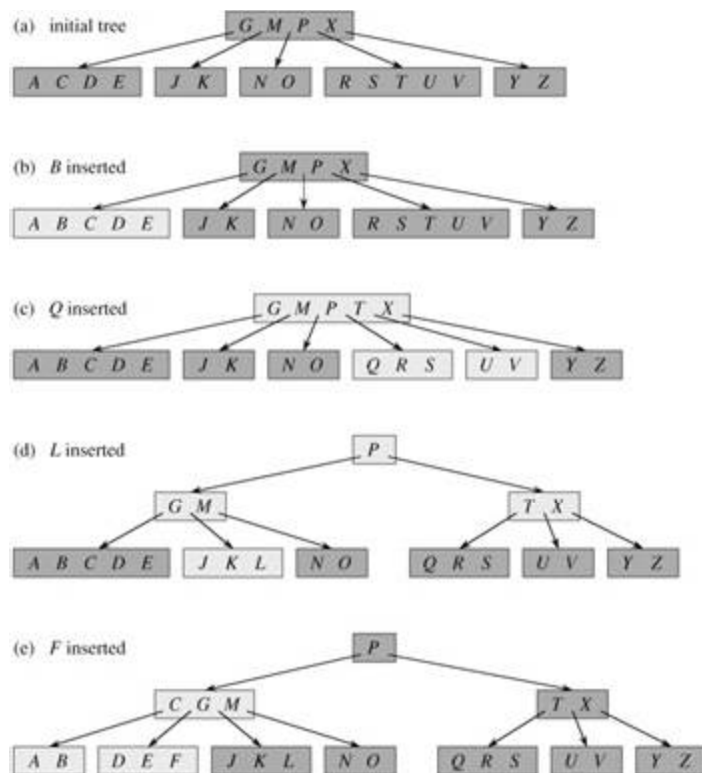Figure 18.7 illustrates the various cases of inserting into a B-tree.

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* is split into two nodes containing *RS* and *UV*, the key T is moved 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 is split 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* is split before F is inserted into the rightmost of the two halves (the *DE* node).

The number of disk accesses performed by B-TREE-INSERT is $O(h)$ for a B-tree of height h, since only $O(1)$ DISK-READ and DISK-WRITE operations are performed between calls to B-TREE-INSERT-NONFULL. The total CPU time used is $O(th) = O(t \log_t n)$. Since B-TREE-INSERT-NONFULL is tail-recursive, it can be alternatively implemented as a **while** loop, 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. Only draw the configurations of the tree just before some node must split, and also draw the final configuration.

Exercises 18.2-2

Explain under what circumstances, if any, redundant DISK-READ or DISK-WRITE operations are performed 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.)

Exercises 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.

Exercises 18.2-4: ★

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

Exercises 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.

Exercises 18.2-6

Suppose that B-TREE-SEARCH is implemented 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.

Exercises 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 a key may be deleted from any node-not just a leaf-and deletion from an internal node requires that the node's children be rearranged. 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, though it is not allowed to have more than the maximum number $2t - 1$ 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.

Assume that procedure B-TREE-DELETE is asked to delete the key k from the subtree rooted at x. This procedure is structured to guarantee that whenever B-TREE-DELETE is called 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). The following specification for deletion from a B-tree should be interpreted with the understanding that if it ever happens that the root node x becomes an internal node having no keys (this situation can occur in cases 2c and 3b, below), then x is deleted and x's only child $c_1[x]$ 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).

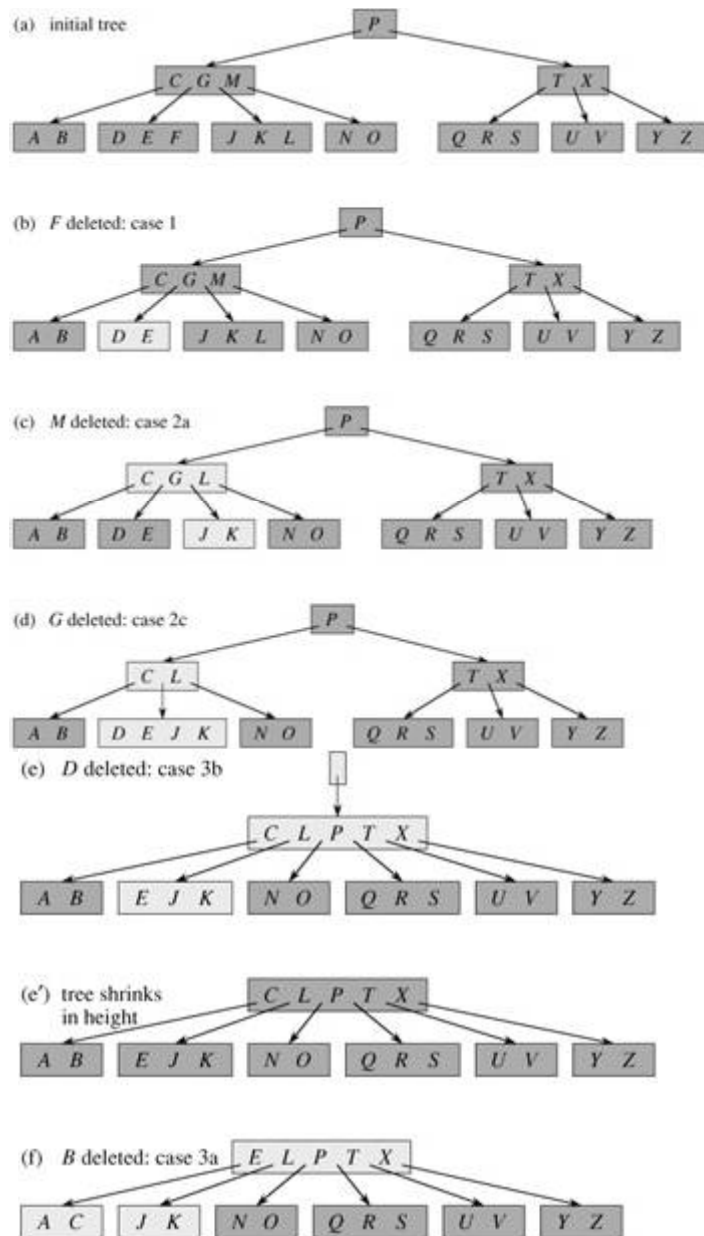illustrates the various cases of deleting keys from a B-tree.

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 is moved up to take M's position. *(d)* Deletion of G. This is case 2c: G is pushed down to make node $DEGJK$, and then G is deleted from this leaf (case 1). *(e)* Deletion of D. This is case 3b: the recursion can't descend to node CL because it has only 2 keys, so P is pushed down and merged with CL and TX to form $CLPTX$; then, D is deleted from a leaf (case 1). *(e')* After (d), the root is deleted and the tree shrinks in height by one. *(f)* Deletion of B. This is case 3a: C is moved to fill B's position and E is moved to fill C's position.

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.
   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. (Finding k' and deleting it can be performed in a single downward pass.)

b. Symmetrically, if the child z that follows k in node x 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. (Finding k' and deleting it can be performed 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 $c_i[x]$ of the appropriate subtree that must contain k, if k is in the tree at all. If $c_i[x]$ 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 $c_i[x]$ has only t - 1 keys but has an immediate sibling with at least t keys, give $c_i[x]$ an extra key by moving a key from x down into $c_i[x]$, moving a key from $c_i[x]$'s immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into $c_i[x]$.
   b. If $c_i[x]$ and both of $c_i[x]$'s immediate siblings have t - 1 keys, merge $c_i[x]$ 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 Figure 18.8(f).

Exercises 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