to the node that roots the subtree, and do not declare or pass any extra variables. The more compact the code, the less likely that a silly bug will turn up. A fourth, less often used, traversal (which we have not seen yet) is **level-order traversal**. In a level-order traversal, all nodes at depth $d$ are processed before any node at depth $d + 1$. Level-order traversal differs from the other traversals in that it is not done recursively; a queue is used, instead of the implied stack of recursion.

## 4.7 B-Trees

So far, we have assumed that we can store an entire data structure in the main memory of a computer. Suppose, however, that we have more data than can fit in main memory, and, as a result, must have the data structure reside on disk. When this happens, the rules of the game change, because the Big-Oh model is no longer meaningful.

The problem is that a Big-Oh analysis assumes that all operations are equal. However, this is not true, especially when disk I/O is involved. Modern computers execute billions of instructions per second. That is pretty fast, mainly because the speed depends largely on electrical properties. On the other hand, a disk is mechanical. Its speed depends largely on the time it takes to spin the disk and to move a disk head. Many disks spin at 7,200 RPM. Thus, in 1 min it makes 7,200 revolutions; hence, one revolution occurs in 1/120 of a second, or 8.3 ms. On average, we might expect that we have to spin a disk halfway to find what we are looking for, but this is compensated by the time to move the disk head, so we get an access time of 8.3 ms. (This is a very charitable estimate; 9–11 ms access times are more common.) Consequently, we can do approximately 120 disk accesses per second. This sounds pretty good, until we compare it with the processor speed. What we have is billions instructions equal to 120 disk accesses. Of course, everything here is a rough calculation, but the relative speeds are pretty clear: Disk accesses are incredibly expensive. Furthermore, processor speeds are increasing at a much faster rate than disk speeds (it is disk *sizes* that are increasing quite quickly). So we are willing to do lots of calculations just to save a disk access. In almost all cases, it is the number of disk accesses that will dominate the running time. Thus, if we halve the number of disk accesses, the running time will halve.

Here is how the typical search tree performs on disk: Suppose we want to access the driving records for citizens in the state of Florida. We assume that we have 10,000,000 items, that each key is 32 bytes (representing a name), and that a record is 256 bytes. We assume this does not fit in main memory and that we are 1 of 20 users on a system (so we have 1/20 of the resources). Thus, in 1 sec we can execute many millions of instructions or perform six disk accesses.

The unbalanced binary search tree is a disaster. In the worst case, it has linear depth and thus could require 10,000,000 disk accesses. On average, a successful search would require $1.38 \log N$ disk accesses, and since $\log 10000000 \approx 24$, an average search would require 32 disk accesses, or 5 sec. In a typical randomly constructed tree, we would expect that a few nodes are three times deeper; these would require about 100 disk accesses, or 16 sec. An AVL tree is somewhat better. The worst case of $1.44 \log N$ is unlikely to occur, and the typical case is very close to $\log N$. Thus an AVL tree would use about 25 disk accesses on average, requiring 4 sec.

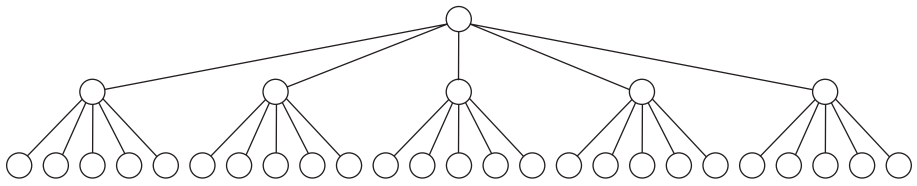

**Figure 4.62** 5-ary tree of 31 nodes has only three levels

We want to reduce the number of disk accesses to a very small constant, such as three or four. We are willing to write complicated code to do this, because machine instructions are essentially free, as long as we are not ridiculously unreasonable. It should probably be clear that a binary search tree will not work, since the typical AVL tree is close to optimal height. We cannot go below $\log N$ using a binary search tree. The solution is intuitively simple: If we have more branching, we have less height. Thus, while a perfect binary tree of 31 nodes has five levels, a 5-ary tree of 31 nodes has only three levels, as shown in Figure 4.62. An **M-ary search tree** allows $M$-way branching. As branching increases, the depth decreases. Whereas a complete binary tree has height that is roughly $\log_2 N$, a complete $M$-ary tree has height that is roughly $\log_M N$.

We can create an $M$-ary search tree in much the same way as a binary search tree. In a binary search tree, we need one key to decide which of two branches to take. In an $M$-ary search tree, we need $M - 1$ keys to decide which branch to take. To make this scheme efficient in the worst case, we need to ensure that the $M$-ary search tree is balanced in some way. Otherwise, like a binary search tree, it could degenerate into a linked list. Actually, we want an even more restrictive balancing condition. That is, we do not want an $M$-ary search tree to degenerate to even a binary search tree, because then we would be stuck with $\log N$ accesses.

One way to implement this is to use a **B-tree.** The basic B-tree[3] is described here. Many variations and improvements are known, and an implementation is somewhat complex because there are quite a few cases. However, it is easy to see that, in principle, a B-tree guarantees only a few disk accesses.

A B-tree of order $M$ is an $M$-ary tree with the following properties:[4]

1. The data items are stored at leaves.
2. The nonleaf nodes store up to $M - 1$ keys to guide the searching; key $i$ represents the smallest key in subtree $i + 1$.
3. The root is either a leaf or has between two and $M$ children.
4. All nonleaf nodes (except the root) have between $\lceil M/2 \rceil$ and $M$ children.
5. All leaves are at the same depth and have between $\lceil L/2 \rceil$ and $L$ data items, for some $L$ (the determination of $L$ is described shortly).

[3] What is described is popularly known as a $B^+$ tree.

[4] Rules 3 and 5 must be relaxed for the first $L$ insertions.

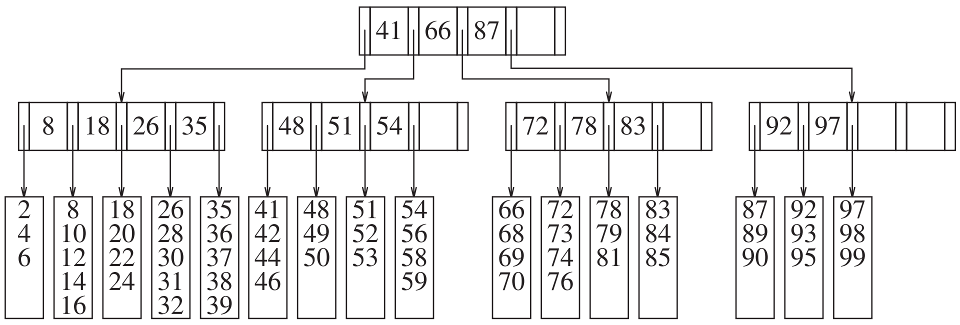

**Figure 4.63** B-tree of order 5

An example of a B-tree of order 5 is shown in Figure 4.63. Notice that all nonleaf nodes have between three and five children (and thus between two and four keys); the root could possibly have only two children. Here, we have $L = 5$. It happens that $L$ and $M$ are the same in this example, but this is not necessary. Since $L$ is 5, each leaf has between three and five data items. Requiring nodes to be half full guarantees that the B-tree does not degenerate into a simple binary tree. Although there are various definitions of B-trees that change this structure, mostly in minor ways, this definition is one of the popular forms.

Each node represents a disk block, so we choose $M$ and $L$ on the basis of the size of the items that are being stored. As an example, suppose one block holds 8,192 bytes. In our Florida example, each key uses 32 bytes. In a B-tree of order $M$, we would have $M-1$ keys, for a total of $32M - 32$ bytes, plus $M$ branches. Since each branch is essentially a number of another disk block, we can assume that a branch is 4 bytes. Thus the branches use $4M$ bytes. The total memory requirement for a nonleaf node is thus $36M-32$. The largest value of $M$ for which this is no more than 8,192 is 228. Thus we would choose $M = 228$. Since each data record is 256 bytes, we would be able to fit 32 records in a block. Thus we would choose $L = 32$. We are guaranteed that each leaf has between 16 and 32 data records and that each internal node (except the root) branches in at least 114 ways. Since there are 10,000,000 records, there are, at most, 625,000 leaves. Consequently, in the worst case, leaves would be on level 4. In more concrete terms, the worst-case number of accesses is given by approximately $\log_{M/2} N$, give or take 1. (For example, the root and the next level could be cached in main memory, so that over the long run, disk accesses would be needed only for level 3 and deeper.)

The remaining issue is how to add and remove items from the B-tree. The ideas involved are sketched next. Note that many of the themes seen before recur.

We begin by examining insertion. Suppose we want to insert 57 into the B-tree in Figure 4.63. A search down the tree reveals that it is not already in the tree. We can add it to the leaf as a fifth item. Note that we may have to reorganize all the data in the leaf to do this. However, the cost of doing this is negligible when compared to that of the disk access, which in this case also includes a disk write.

Of course, that was relatively painless, because the leaf was not already full. Suppose we now want to insert 55. Figure 4.64 shows a problem: The leaf where 55 wants to go is already full. The solution is simple: Since we now have $L + 1$ items, we split them into two

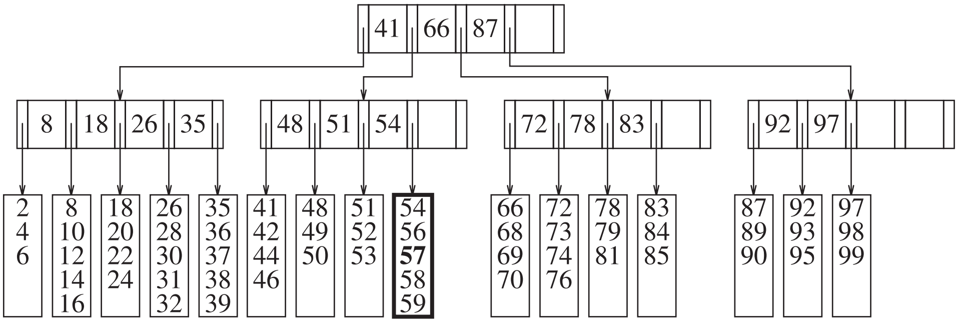

**Figure 4.64** B-tree after insertion of 57 into the tree in Figure 4.63

leaves, both guaranteed to have the minimum number of data records needed. We form two leaves with three items each. Two disk accesses are required to write these leaves, and a third disk access is required to update the parent. Note that in the parent, both keys and branches change, but they do so in a controlled way that is easily calculated. The resulting B-tree is shown in Figure 4.65. Although splitting nodes is time-consuming because it requires at least two additional disk writes, it is a relatively rare occurrence. If $L$ is 32, for example, then when a node is split, two leaves with 16 and 17 items, respectively, are created. For the leaf with 17 items, we can perform 15 more insertions without another split. Put another way, for every split, there are roughly $L/2$ nonsplits.

The node splitting in the previous example worked because the parent did not have its full complement of children. But what would happen if it did? Suppose, for example, that we insert 40 into the B-tree in Figure 4.65. We must split the leaf containing the keys 35 through 39, and now 40, into two leaves. But doing this would give the parent six children, and it is allowed only five. The solution is to split the parent. The result of this is shown in Figure 4.66. When the parent is split, we must update the values of the keys and also the parent's parent, thus incurring an additional two disk writes (so this insertion costs five disk writes). However, once again, the keys change in a very controlled manner, although the code is certainly not simple because of a host of cases.

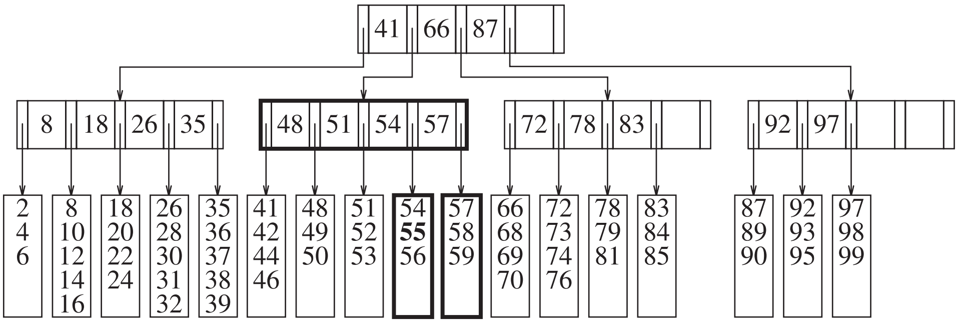

**Figure 4.65** Insertion of 55 into the B-tree in Figure 4.64 causes a split into two leaves

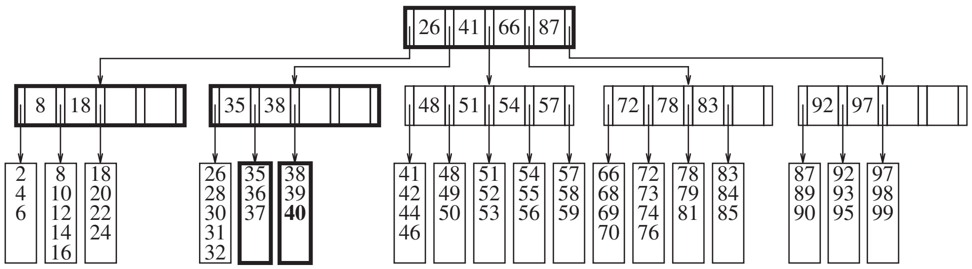

**Figure 4.66** Insertion of 40 into the B-tree in Figure 4.65 causes a split into two leaves and then a split of the parent node

When a nonleaf node is split, as is the case here, its parent gains a child. What if the parent already has reached its limit of children? Then we continue splitting nodes up the tree until either we find a parent that does not need to be split or we reach the root. If we split the root, then we have two roots. Obviously, this is unacceptable, but we can create a new root that has the split roots as its two children. This is why the root is granted the special two-child minimum exemption. It also is the only way that a B-tree gains height. Needless to say, splitting all the way up to the root is an exceptionally rare event. This is because a tree with four levels indicates that the root has been split three times throughout the entire sequence of insertions (assuming no deletions have occurred). In fact, the splitting of any nonleaf node is also quite rare.

There are other ways to handle the overflowing of children. One technique is to put a child up for adoption should a neighbor have room. To insert 29 into the B-tree in Figure 4.66, for example, we could make room by moving 32 to the next leaf. This technique requires a modification of the parent, because the keys are affected. However, it tends to keep nodes fuller and saves space in the long run.

We can perform deletion by finding the item that needs to be removed and then removing it. The problem is that if the leaf it was in had the minimum number of data items, then it is now below the minimum. We can rectify this situation by adopting a neighboring item, if the neighbor is not itself at its minimum. If it is, then we can combine with the neighbor to form a full leaf. Unfortunately, this means that the parent has lost a child. If this causes the parent to fall below its minimum, then it follows the same strategy. This process could

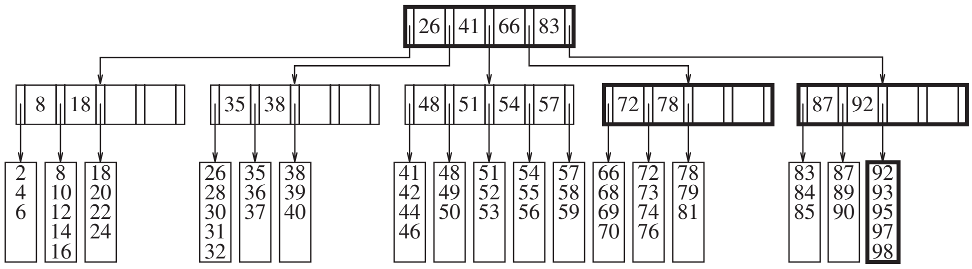

**Figure 4.67** B-tree after the deletion of 99 from the B-tree in Figure 4.66

percolate all the way up to the root. The root cannot have just one child (and even if this were allowed, it would be silly). If a root is left with one child as a result of the adoption process, then we remove the root and make its child the new root of the tree. This is the only way for a B-tree to lose height. For example, suppose we want to remove 99 from the B-tree in Figure 4.66. Since the leaf has only two items and its neighbor is already at its minimum of three, we combine the items into a new leaf of five items. As a result, the parent has only two children. However, it can adopt from a neighbor, because the neighbor has four children. As a result, both have three children. The result is shown in Figure 4.67.

## 4.8 Sets and Maps in the Standard Library

The STL containers discussed in Chapter 3—namely, `vector` and `list`—are inefficient for searching. Consequently, the STL provides two additional containers, `set` and `map`, that guarantee logarithmic cost for basic operations such as insertion, deletion, and searching.

### 4.8.1 Sets

The `set` is an ordered container that does not allow duplicates. Many of the idioms used to access items in `vector` and `list` also work for a `set`. Specifically, nested in the `set` are `iterator` and `const_iterator` types that allow traversal of the `set`, and several methods from `vector` and `list` are identically named in `set`, including `begin`, `end`, `size`, and `empty`. The `print` function template described in Figure 3.6 will work if passed a `set`.

The unique operations required by the `set` are the abilities to insert, remove, and perform a basic search (efficiently).

The insert routine is aptly named `insert`. However, because a `set` does not allow duplicates, it is possible for the `insert` to fail. As a result, we want the return type to be able to indicate this with a Boolean variable. However, `insert` has a more complicated return type than a `bool`. This is because `insert` also returns an `iterator` that represents where `x` is when `insert` returns. This `iterator` represents either the newly inserted item or the existing item that caused the `insert` to fail, and it is useful, because knowing the position of the item can make removing it more efficient by avoiding the search and getting directly to the node containing the item.

The STL defines a class template called `pair` that is little more than a `struct` with members `first` and `second` to access the two items in the `pair`. There are two different `insert` routines:

```
pair<iterator,bool> insert( const Object & x );
pair<iterator,bool> insert( iterator hint, const Object & x );
```

The one-parameter `insert` behaves as described above. The two-parameter `insert` allows the specification of a hint, which represents the position where `x` should go. If the hint is accurate, the insertion is fast, often $O(1)$. If not, the insertion is done using the normal insertion algorithm and performs comparably with the one-parameter `insert`. For instance, the following code might be faster using the two-parameter `insert` rather than the one-parameter `insert`: