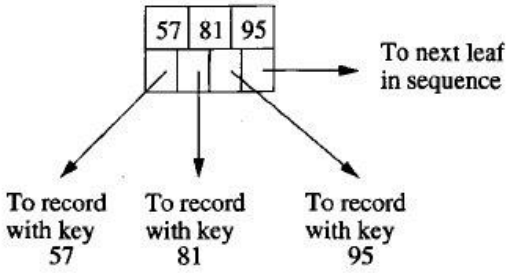

Figure 14.11 A typical leaf node of a B-tree

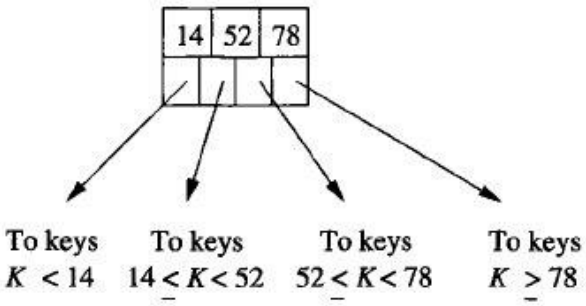

Figure 14.12 A typical interior node of a B-tree

14.2.3 Lookup in B-Trees

We now revert to our original assumption that there are no duplicate keys at the leaves. We also suppose that the B-tree is a dense index, so every search-key value that appears in the data file will also appear at a leaf. These assumptions make the discussion of B-tree operations simpler, but are not essential for these operations. In particular, modifications for sparse indexes are similar to the changes we introduced in Section 14.1.3 for indexes on sequential files.
Suppose we have a B-tree index and we want to find a record with search key value K. We search for K recursively, starting at the root and ending at a leaf. The search procedure is:

BASIS: If we are at a leaf, look among the keys there. If the ith key is K, then the ith pointer will take us to the desired record.

INDUCTION: If we are at an interior node with keys $K_1, K_2, \ldots, K_n$, follow the rules given in Section 14.2.1 to decide which of the children of this node should next be examined. That is, there is only one child that could lead to a leaf with key K. If $K < K_1$, then it is the first child, if $K_1 < K <= K_2$, it is the second child, and so on. Recursively apply the search procedure at this child.

Example 14.14: Suppose we have the B-tree of Fig. 14.13, and we want to find a record with search key 40. We start at the root, where there is one key, 13. Since 13 < 40, we follow the second pointer, which leads us to the second-level node with keys 23, 31, and 43. At that node, we find 31 < 40 < 43, so we follow the third pointer. We are thus led to the leaf with keys 31, 37, and 41. If there had been a record in the data file with key 40, we would have found key 40

at this leaf. Since we do not find 40, we conclude that there is no record with key 40 in the underlying data.
Note that had we been looking for a record with key 37, we would have taken exactly the same decisions, but when we got to the leaf we would find key 37. Since it is the second key in the leaf, we follow the second pointer, which will lead us to the data record with key 37. □

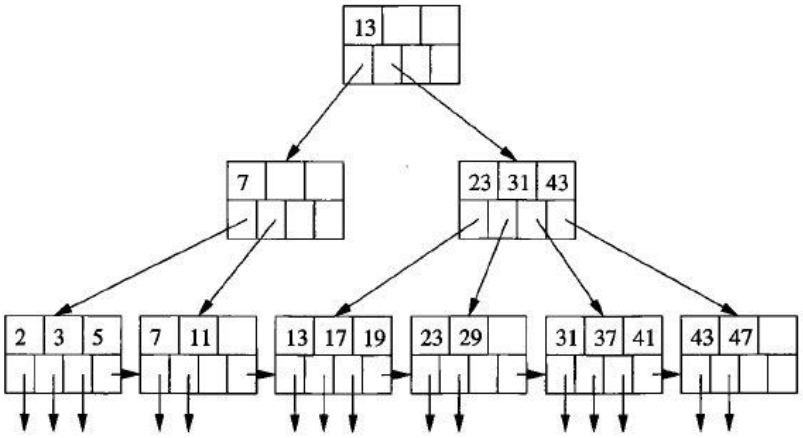

Figure 14.13 A B-tree

14.2.4 Range Queries

B-trees are useful not only for queries in which a single value of the search key is sought, but for queries in which a range of values are asked for. Typically, *range queries* have a term in the WHERE-clause that compares the search key with a value or values, using one of the comparison operators other than = or <>. Examples of range queries using a search-key attribute *k* are:

```
SELECT * FROM R WHERE R.k > 40;
SELECT * FROM R WHERE R.k >= 10 AND R.k <= 25;
```

If we want to find all keys in the range [a, b] at the leaves of a B-tree, we do a lookup to find the key *a*. Whether or not it exists, we are led to a leaf where *a* could be, and we search the leaf for keys that are *a* or greater. Each such key we find has an associated pointer to one of the records whose key is in the desired range. As long as we do not find a key greater than *b* in the current block, we follow the pointer to the next leaf and repeat our search for keys in the range [a, b].
The above search algorithm also works if *b* is infinite; i.e., there is only a lower bound and no upper bound. In that case, we search all the leaves from the one that would hold key *a* to the end of the chain of leaves. If a is -∞ (that is, there is an upper bound on the range but no lower bound), then the search for "minus infinity" as a search key will always take us to the first leaf. The search then proceeds as above, stopping only when we pass the key *b*.

Example 14.15: Suppose we have the B-tree of Fig. 14.13, and we are given the range (10, 25) to search for. We look for key 10, which leads us to the second leaf. The first key is less than 10, but the second, 11, is at least 10. We follow its associated pointer to get the record with key 11.
Since there are no more keys in the second leaf, we follow the chain to the third leaf, where we find keys 13, 17, and 19. All are less than or equal to 25, so we follow their associated pointers

and retrieve the records with these keys. Finally, we move to the fourth leaf, where we find key 23. But the next key of that leaf, 29, exceeds 25, so we are done with our search. Thus, we have retrieved the five records with keys 11 through 23. □

14.2.5 Insertion Into B-Trees

We see some of the advantages of B-trees over simpler multilevel indexes when we consider how to insert a new key into a B-tree. The corresponding record will be inserted into the file being indexed by the B-tree, using any of the methods discussed in Section 14.1; here we consider how the B-tree changes. The insertion is, in principle, recursive:

- We try to find a place for the new key in the appropriate leaf, and we put it there if there is room.
- If there is no room in the proper leaf, we split the leaf into two and divide the keys between the two new nodes, so each is half full or just over half full.
- The splitting of nodes at one level appears to the level above as if a new key-pointer pair needs to be inserted at that higher level. We may thus recursively apply this strategy to insert at the next level: if there is room, insert it; if not, split the parent node and continue up the tree.
- As an exception, if we try to insert into the root, and there is no room, then we split the root into two nodes and create a new root at the next higher level; the new root has the two nodes resulting from the split as its children. Recall that no matter how large n (the number of slots for keys at a node) is, it is always permissible for the root to have only one key and two children.

When we split a node and insert it into its parent, we need to be careful how the keys are managed. First, suppose N *is a* leaf whose capacity is n keys. Also suppose we are trying to insert an $(n + 1)$st key and its associated pointer. We create a new node M, which will be the sibling of N, immediately to its right. The first $\left\lceil \frac{n+1}{2} \right\rceil$ key-pointer pairs, in sorted order of the keys, remain with N, while the other key-pointer pairs move to M. Note that both nodes N and M are left with a sufficient number of key-pointer pairs - at least $\left\lfloor \frac{n+1}{2} \right\rfloor$ pairs.

Now, suppose N is an interior node whose capacity is n keys and $n + 1$ pointers, and N has just been assigned n + 2 pointers because of a node splitting below. We do the following:

1. Create a new node M, which will be the sibling of N, immediately to its right.
2. Leave at N the first $\left\lceil \frac{n+2}{2} \right\rceil$ pointers, in sorted order, and move to M the remaining $\left\lfloor \frac{n+2}{2} \right\rfloor$ pointers.
3. The first $\left\lceil \frac{n}{2} \right\rceil$ keys stay with N, while the last $\left\lfloor \frac{n}{2} \right\rfloor$ keys move to M. Note that there is always one key in the middle left over; it goes with neither N nor M. The leftover key K indicates the smallest key reachable via the first of M's children. Although this key doesn't appear in N or M, it is associated with M, in the sense that it represents the smallest key reachable via M. Therefore K will be inserted into the parent of N and M to divide searches between those two nodes.

Example 14.16: Let us insert key 40 into the B-tree of Fig. 14.13. We find the proper leaf for the insertion by the lookup procedure of Section 14.2.3. As found in Example 14.14, the insertion goes into the fifth leaf. Since this leaf now has four key-pointer pairs — 31, 37, 40, and 41 — we need to split the leaf. Our first step is to create a new node and move the highest two keys, 40 and 41, along with their pointers, to that node. Figure 14.15 shows this split.

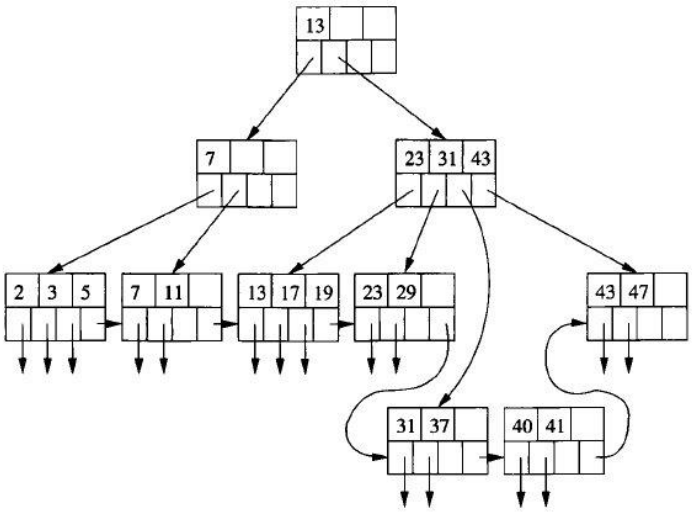

Figure 14.15 Beginning the insertion of key 40

Notice that although we now show the nodes on four ranks to save space, there are still only three levels to the tree. The seven leaves are linked by their last pointers, which still form a chain from left to right. We must now insert a pointer to the new leaf (the one with keys 40 and 41) into the node above it (the node with keys 23, 31, and 43). We must also associate with this pointer the key 40, which is the least key reachable through the new leaf. Unfortunately, the parent of the split node is already full; it has no room for another key or pointer. Thus, it too must be split.

We start with pointers to the last five leaves and the list of keys representing the least keys of the last four of these leaves. That is, we have pointers *Pi, P2, P3, Pi, P5* to the leaves whose least keys are 13, 23, 31, 40, and 43, and we have the key sequence 23, 31, 40, 43 to separate these pointers. The first three pointers and first two keys remain with the split interior node, while the last two pointers and last key go to the new node. The remaining key, 40, represents the least key accessible via the new node.

Figure 14.16 shows the completion of the insert of key 40. The root now has three children; the last two are the split interior node. Notice that the key 40, which marks the lowest of the keys reachable via the second of the split nodes, has been installed in the root to separate the keys of the root's second and third children.

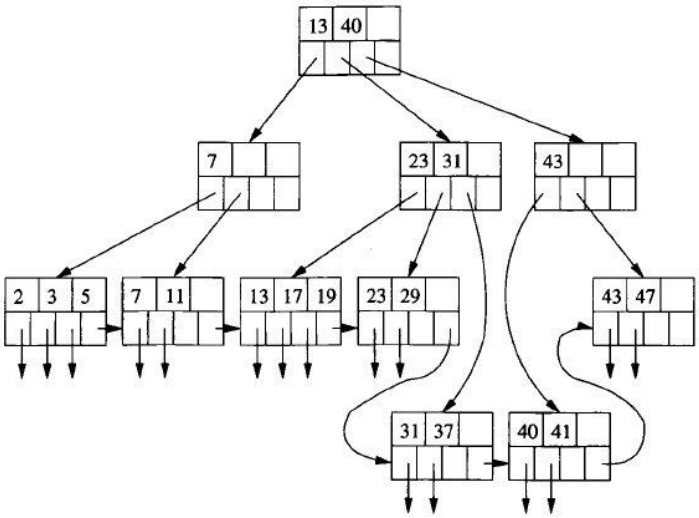

Figure 14.16 Completing the insertion of key 40