and p_{leaf}, we should reduce the block size by the amount of space needed for all such information. The next example illustrates how we can calculate the number of entries in a B$^+$-tree.

Example 7. Suppose that we construct a B$^+$-tree on the field in Example 6. To calculate the approximate number of entries in the B$^+$-tree, we assume that each node is 69% full. On the average, each internal node will have 34 * 0.69 or approximately 23 pointers, and hence 22 values. Each leaf node, on the average, will hold 0.69 * p_{leaf} = 0.69 * 31 or approximately 21 data record pointers. A B$^+$-tree will have the following average number of entries at each level:

Root:	1 node	22 key entries	23 pointers
Level 1:	23 nodes	506 key entries	529 pointers
Level 2:	529 nodes	11,638 key entries	12,167 pointers
Leaf level:	12,167 nodes	255,507 data record pointers	

For the block size, pointer size, and search field size as in Example 6, a three-level B$^+$-tree holds up to 255,507 record pointers, with the average 69% occupancy of nodes. Note that we considered the leaf node differently from the nonleaf nodes and computed the data pointers in the leaf node to be 12,167 * 21 based on 69% occupancy of the leaf node, which can hold 31 keys with data pointers. Compare this to the 65,535 entries for the corresponding B-tree in Example 5. Because a B-tree includes a data/record pointer along with each search key at all levels of the tree, it tends to accommodate less number of keys for a given number of index levels. This is the main reason that B$^+$-trees are preferred to B-trees as indexes to database files. Most DBMSs, such as Oracle, are creating all indexes as B$^+$-trees.

Search, Insertion, and Deletion with B$^+$-Trees. Algorithm 17.2 outlines the procedure using the B$^+$-tree as the access structure to search for a record. Algorithm 17.3 illustrates the procedure for inserting a record in a file with a B$^+$-tree access structure. These algorithms assume the existence of a key search field, and they must be modified appropriately for the case of a B$^+$-tree on a nonkey field. We illustrate insertion and deletion with an example.

Algorithm 17.2. Searching for a Record with Search Key Field Value *K*, Using a B$^+$-Tree

```
n ← block containing root node of B+-tree;
read block n;
while (n is not a leaf node of the B+-tree) do
    begin
    q ← number of tree pointers in node n;
    if K ≤ n.K1 (*n.Ki refers to the ith search field value in node n*)
        then n ← n.P1 (*n.Pi refers to the ith tree pointer in node n*)
        else if K > n.Kq−1
            then n ← n.Pq
```

else **begin**
search node n for an entry i such that $n.K_{i-1} < K \leq n.K_i$;
$n \leftarrow n.P_i$
end;
read block n
end;
search block n for entry (K_i, Pr_i) with $K = K_i$; (* search leaf node *)
if found
then read data file block with address Pr_i and retrieve record
else the record with search field value K is not in the data file;

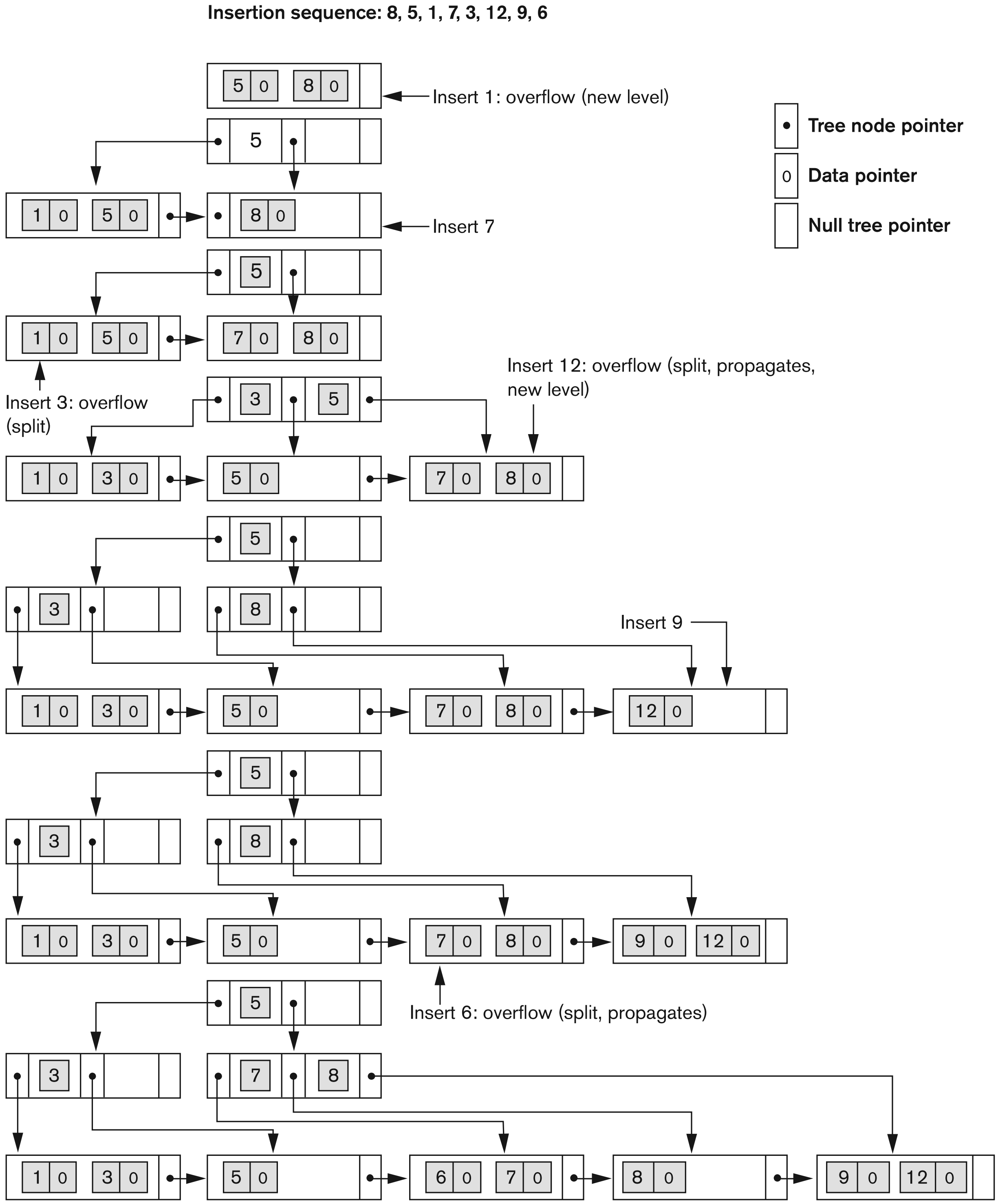
Algorithm 17.3. Inserting a Record with Search Key Field Value K in a B^+-Tree of Order p

$n \leftarrow$ block containing root node of B^+-tree;
read block n; set stack S to empty;
while (n is not a leaf node of the B^+-tree) do
begin
push address of n on stack S;
(*stack S holds parent nodes that are needed in case of split*)
$q \leftarrow$ number of tree pointers in node n;
if $K \leq n.K_1$ (*$n.K_i$ refers to the ith search field value in node n*)
then $n \leftarrow n.P_1$ (*$n.P_i$ refers to the ith tree pointer in node n*)
else if $K \leftarrow n.K_{q-1}$
then $n \leftarrow n.P_q$
else **begin**
search node n for an entry i such that $n.K_{i-1} < K \leq n.K_i$;
$n \leftarrow n.P_i$
end;
read block n
end;
search block n for entry (K_i, Pr_i) with $K = K_i$; (*search leaf node n*)
if found
then record already in file; cannot insert
else (*insert entry in B^+-tree to point to record*)
begin
create entry (K, Pr) where Pr points to the new record;
if leaf node n is not full
then insert entry (K, Pr) in correct position in leaf node n
else **begin** (*leaf node n is full with p_{leaf} record pointers; is split*)
copy n to $temp$ (*$temp$ is an oversize leaf node to hold extra entries*);
insert entry (K, Pr) in $temp$ in correct position;
(*$temp$ now holds $p_{leaf} + 1$ entries of the form (K_i, Pr_i)*)
$new \leftarrow$ a new empty leaf node for the tree; $new.P_{next} \leftarrow n.P_{next}$;
$j \leftarrow \lceil (p_{leaf} + 1)/2 \rceil$;
$n \leftarrow$ first j entries in $temp$ (up to entry (K_j, Pr_j)); $n.P_{next} \leftarrow new$;

```
        new ← remaining entries in temp; K ← K_j ;
        (*now we must move (K, new) and insert in parent internal node;
          however, if parent is full, split may propagate*)
        finished ← false;
        repeat
        if stack S is empty
            then (←no parent node; new root node is created for the tree*)
                begin
                root ← a new empty internal node for the tree;
                root ← <n, K, new>; finished ← true;
                end
            else begin
                n ← pop stack S;
                if internal node n is not full
                    then
                        begin (*parent node not full; no split*)
                        insert (K, new) in correct position in internal node n;
                        finished ← true
                        end
                    else begin (*internal node n is full with p tree pointers;
                                overflow condition; node is split*)
                    copy n to temp (*temp is an oversize internal node*);
                    insert (K, new) in temp in correct position;
                    (*temp now has p + 1 tree pointers*)
                    new ← a new empty internal node for the tree;
                    j ← ⌊((p + 1)/2⌋ ;
                    n ← entries up to tree pointer P_j in temp;
                    (*n contains <P_1, K_1, P_2, K_2, ... , P_{j−1}, K_{j−1}, P_j >*)
                    new ← entries from tree pointer P_{j+1} in temp;
                    (*new contains < P_{j+1}, K_{j+1}, ... , K_{p−1}, P_p, K_p, P_{p+1} >*)
                    K ← K_j
                    (*now we must move (K, new) and insert in
                      parentinternal node*)
            end
        end
    until finished
    end;
end;
```

Figure 17.12 illustrates insertion of records in a B^+-tree of order $p = 3$ and $p_{leaf} = 2$. First, we observe that the root is the only node in the tree, so it is also a leaf node. As soon as more than one level is created, the tree is divided into internal nodes and leaf nodes. Notice that *every key value must exist at the leaf level*, because all data pointers are at the leaf level. However, only some values exist in internal nodes to guide the search. Notice also that every value appearing in an internal node also appears as *the rightmost value* in the leaf level of the subtree pointed at by the tree pointer to the left of the value.

Figure 17.12
An example of insertion in a B^+-tree with $p = 3$ and $p_{leaf} = 2$.



When a *leaf node* is full and a new entry is inserted there, the node *overflows* and must be split. The first $j = \lceil((p_{\text{leaf}} + 1)/2)\rceil$ entries in the original node are kept there, and the remaining entries are moved to a new leaf node. The *j*th search value is replicated in the parent internal node, and an extra pointer to the new node is created in the parent. These must be inserted in the parent node in their correct sequence. If the parent internal node is full, the new value will cause it to overflow also, so it must be split. The entries in the internal node up to P_j—the *j*th tree pointer after inserting the new value and pointer, where $j = \lfloor((p + 1)/2)\rfloor$—are kept, whereas the *j*th search value is moved to the parent, not replicated. A new internal node will hold the entries from P_{j+1} to the end of the entries in the node (see Algorithm 17.3). This splitting can propagate all the way up to create a new root node and hence a new level for the B$^+$-tree.

Figure 17.13 illustrates deletion from a B$^+$-tree. When an entry is deleted, it is always removed from the leaf level. If it happens to occur in an internal node, it must also be removed from there. In the latter case, the value to its left in the leaf node must replace it in the internal node because that value is now the rightmost entry in the subtree. Deletion may cause **underflow** by reducing the number of entries in the leaf node to below the minimum required. In this case, we try to find a sibling leaf node—a leaf node directly to the left or to the right of the node with underflow—and redistribute the entries among the node and its **sibling** so that both are at least half full; otherwise, the node is merged with its siblings and the number of leaf nodes is reduced. A common method is to try to **redistribute** entries with the left sibling; if this is not possible, an attempt to redistribute with the right sibling is made. If this is also not possible, the three nodes are merged into two leaf nodes. In such a case, underflow may propagate to **internal** nodes because one fewer tree pointer and search value are needed. This can propagate and reduce the tree levels.

Notice that implementing the insertion and deletion algorithms may require parent and sibling pointers for each node, or the use of a stack as in Algorithm 17.3. Each node should also include the number of entries in it and its type (leaf or internal). Another alternative is to implement insertion and deletion as recursive procedures.[13]

Variations of B-Trees and B$^+$-Trees. To conclude this section, we briefly mention some variations of B-trees and B$^+$-trees. In some cases, constraint 5 on the B-tree (or for the internal nodes of the B$^+$–tree, except the root node), which requires each node to be at least half full, can be changed to require each node to be at least two-thirds full. In this case the B-tree has been called a **B*-tree**. In general, some systems allow the user to choose a **fill factor** between 0.5 and 1.0, where the latter means that the B-tree (index) nodes are to be completely full. It is also possible to specify two fill factors for a B$^+$-tree: one for the leaf level and one for the internal nodes of the tree. When the index is first constructed, each node is filled up

[13]For more details on insertion and deletion algorithms for B$^+$-trees, consult Ramakrishnan and Gehrke (2003).

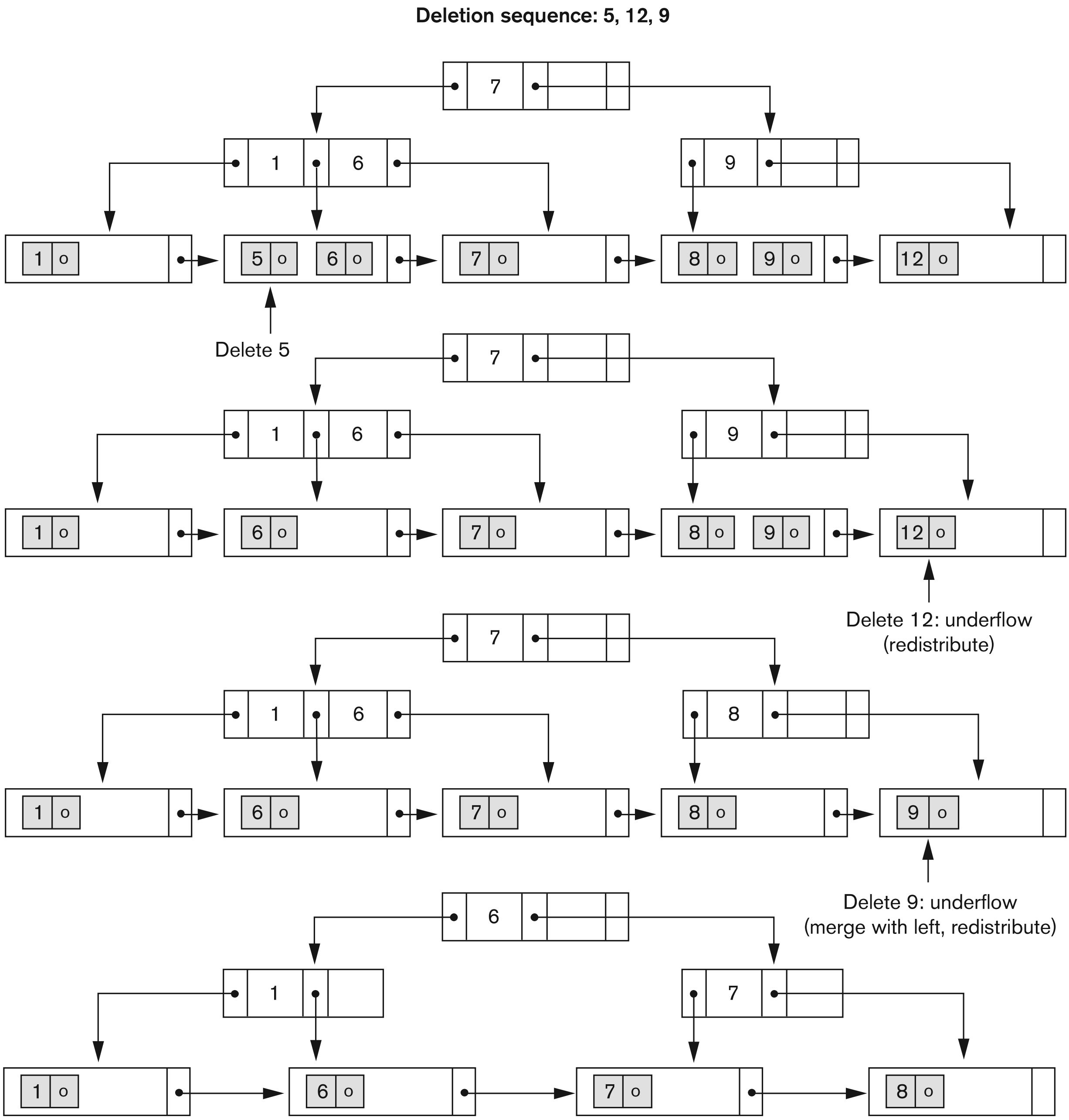


Figure 17.13
An example of deletion from a B+-tree.

to approximately the fill factors specified. Some investigators have suggested relaxing the requirement that a node be half full, and instead allow a node to become completely empty before merging, to simplify the deletion algorithm. Simulation studies show that this does not waste too much additional space under randomly distributed insertions and deletions.