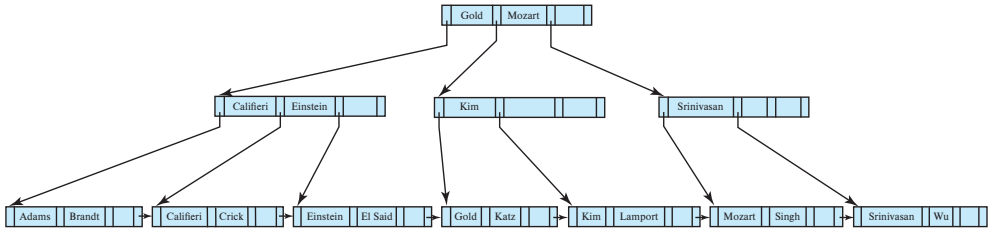

Figure 14.15 Insertion of "Lamport" into the B+-tree of Figure 14.14.

However, the search key value that lies between the pointers that stay on the left, and the pointers that move to the right node is treated differently. In our example, the search key value "Gold" lies between the three pointers that went to the left node, and the two pointers that went to the right node. The value "Gold" is not added to either of the split nodes. Instead, an entry (Gold, $n2$) is added to the parent node, where $n2$ is a pointer to the newly created node that resulted from the split. In this case, the parent node is the root, and it has enough space for the new entry.

The general technique for insertion into a B+-tree is to determine the leaf node l into which insertion must occur. If a split results, insert the new node into the parent

---

procedure *insert*(*value* K, *pointer* P)
  if (tree is empty) create an empty leaf node L, which is also the root
  else Find the leaf node L that should contain key value K
  if (L has less than $n - 1$ key values)
    then insert_in_leaf (L, K, P)
    else begin /* L has $n - 1$ key values already, split it */
      Create node L'
      Copy $L.P_1 \ldots L.K_{n-1}$ to a block of memory T that can
        hold n (pointer, key-value) pairs
      insert_in_leaf (T, K, P)
      Set $L'.P_n = L.P_n$; Set $L.P_n = L'$
      Erase $L.P_1$ through $L.K_{n-1}$ from L
      Copy $T.P_1$ through $T.K_{\lceil n/2 \rceil}$ from T into L starting at $L.P_1$
      Copy $T.P_{\lceil n/2 \rceil + 1}$ through $T.K_n$ from T into L' starting at $L'.P_1$
      Let K' be the smallest key-value in L'
      insert_in_parent(L, K', L')
    end

Figure 14.16 Insertion of entry in a B+-tree.

of node l. If this insertion causes a split, proceed recursively up the tree until either an insertion does not cause a split or a new root is created.

Figure 14.16 outlines the insertion algorithm in pseudocode. The procedure *insert* inserts a key-value pointer pair into the index, using two subsidiary procedures *insert_in_leaf* and *insert_in_parent*, shown in Figure 14.17. In the pseudocode, L, N, P and T denote pointers to nodes, with L being used to denote a leaf node. $L.K_i$ and $L.P_i$ denote the ith value and the ith pointer in node L, respectively; $T.K_i$ and $T.P_i$ are used similarly. The pseudocode also makes use of the function $parent(N)$ to find the parent of a node N. We can compute a list of nodes in the path from the root to the leaf while initially finding the leaf node, and we can use it later to find the parent of any node in the path efficiently.

```
procedure insert_in_leaf (node L, value K, pointer P)
    if (K < L.K_1)
        then insert P, K into L just before L.P_1
        else begin
            Let K_i be the highest value in L that is less than or equal to K
            Insert P, K into L just after L.K_i
        end
procedure insert_in_parent(node N, value K', node N')
    if (N is the root of the tree)
        then begin
            Create a new node R containing N, K', N'    /* N and N' are pointers */
            Make R the root of the tree
            return
        end
    Let P = parent (N)
    if (P has less than n pointers)
        then insert (K', N') in P just after N
        else begin /* Split P */
            Copy P to a block of memory T that can hold P and (K', N')
            Insert (K', N') into T just after N
            Erase all entries from P; Create node P'
            Copy T.P_1 ... T.P_⌈(n+1)/2⌉ into P
            Let K'' = T.K_⌈(n+1)/2⌉
            Copy T.P_⌈(n+1)/2⌉+1 ... T.P_n+1 into P'
            insert_in_parent(P, K'', P')
        end
```

Figure 14.17 Subsidiary procedures for insertion of entry in a B$^+$-tree.