function *find*(v)
/* Assumes no duplicate keys, and returns pointer to the record with
 * search key value v if such a record exists, and null otherwise */
  Set C = root node
  while (C is not a leaf node) **begin**
    Let i = smallest number such that $v \leq C.K_i$
    if there is no such number i **then begin**
      Let P_m = last non-null pointer in the node
      Set $C = C.P_m$
    end
    else if ($v = C.K_i$) **then** Set $C = C.P_{i+1}$
    else Set $C = C.P_i$ /* $v < C.K_i$ */
  end
  /* C is a leaf node */
  if for some i, $K_i = v$
    then return P_i
    else return null ; /* No record with key value v exists*/

Figure 14.11 Querying a B$^+$-tree.

At the leaf node, if there is a search-key value $K_i = v$, pointer P_i directs us to a record with search-key value K_i. The function then returns the pointer to the record, P_i. If no search key with value v is found in the leaf node, no record with key value v exists in the relation, and function *find* returns null, to indicate failure.

B$^+$-trees can also be used to find all records with search key values in a specified range $[lb, ub]$. For example, with a B$^+$-tree on attribute *salary* of *instructor*, we can find all *instructor* records with salary in a specified range such as [50000, 100000] (in other words, all salaries between 50000 and 100000). Such queries are called **range queries**.

To execute such queries, we can create a procedure *findRange*(lb, ub), shown in Figure 14.12. The procedure does the following: it first traverses to a leaf in a manner similar to *find*(lb); the leaf may or may not actually contain value lb. It then steps through records in that and subsequent leaf nodes collecting pointers to all records with key values $C.K_i$ s.t. $lb \leq C.K_i \leq ub$ into a set resultSet. The function stops when $C.K_i > ub$, or there are no more keys in the tree.

A real implementation would provide a version of *findRange* supporting an iterator interface similar to that provided by the JDBC ResultSet, which we saw in Section 5.1.1. Such an iterator interface would provide a method *next*(), which can be called repeatedly to fetch successive records. The *next*() method would step through the entries at the leaf level, in a manner similar to *findRange*, but each call takes only one step and records where it left off, so that successive calls to *next*() step through successive en-

```
function findRange(lb, ub)
/* Returns all records with search key value V such that lb ≤ V ≤ ub. */
    Set resultSet = {};
    Set C = root node
    while (C is not a leaf node) begin
        Let i = smallest number such that lb ≤ C.K_i
        if there is no such number i then begin
            Let P_m = last non-null pointer in the node
            Set C = C.P_m
        end
        else if (lb = C.K_i) then Set C = C.P_{i+1}
        else Set C = C.P_i /* lb < C.K_i */
    end
    /* C is a leaf node */
    Let i be the least value such that K_i ≥ lb
    if there is no such i
        then Set i = 1 + number of keys in C; /* To force move to next leaf */
    Set done = false;
    while (not done) begin
        Let n = number of keys in C.
        if ( i ≤ n and C.K_i ≤ ub) then begin
            Add C.P_i to resultSet
            Set i = i + 1
        end
        else if  (i ≤ n and C.K_i > ub)
            then Set done = true;
        else if (i > n and C.P_{n+1} is not null)
            then Set C = C.P_{n+1}, and i = 1 /* Move to next leaf */
        else Set done = true; /* No more leaves to the right */
    end
    return resultSet;
```

Figure 14.12 Range query on a B$^+$-tree.

tries. We omit details for simplicity, and leave the pseudocode for the iterator interface as an exercise for the interested reader.

We now consider the cost of querying on a B$^+$-tree index. In processing a query, we traverse a path in the tree from the root to some leaf node. If there are N records in the file, the path is no longer than $\lceil \log_{\lceil n/2 \rceil}(N) \rceil$.

Typically, the node size is chosen to be the same as the size of a disk block, which is typically 4 kilobytes. With a search-key size of 12 bytes, and a disk-pointer size of