```
func find (search key value K) returns nodepointer
// Given a search key value, finds its leaf node
return tree_search(root, K);                          // searches from root
endfunc

func tree_search (nodepointer, search key value K) returns nodepointer
// Searches tree for entry
if *nodepointer is a leaf, return nodepointer;
else,
    if K < K_1 then return tree_search(P_0, K);
    else,
        if K ≥ K_m then return tree_search(P_m, K);        // m = # entries
        else,
            find i such that K_i ≤ K < K_{i+1};
            return tree_search(P_i, K)
endfunc
```

Figure 9.9 Algorithm for Search

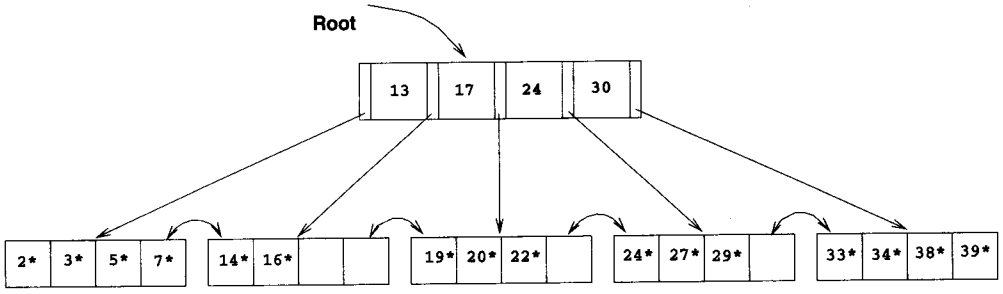

Figure 9.10 Example of a B+ Tree, Order d=2

```
proc insert (nodepointer, entry, newchildentry)
// Inserts entry into subtree with root '*nodepointer'; degree is d;
// 'newchildentry' is null initially, and null upon return unless child is split

if *nodepointer is a non-leaf node, say N,
    find i such that K_i ≤ entry's key value < K_{i+1};          // choose subtree
    insert(P_i, entry, newchildentry);                  // recursively, insert entry
    if newchildentry is null, return;                // usual case; didn't split child
    else,                          // we split child, must insert *newchildentry in N
        if N has space,                                                  // usual case
            put *newchildentry on it, set newchildentry to null, return;
        else,                                // note difference wrt splitting of leaf page!
            split N:                  // 2d + 1 key values and 2d + 2 nodepointers
            first d key values and d + 1 nodepointers stay,
            last d keys and d + 1 pointers move to new node, N2;
            // *newchildentry set to guide searches between N and N2
            newchildentry = & (⟨smallest key value on N2, pointer to N2⟩);
            if N is the root,                                // root node was just split
                create new node with ⟨pointer to N, *newchildentry⟩;
                make the tree's root-node pointer point to the new node;
            return;

if *nodepointer is a leaf node, say L,
    if L has space,                                                      // usual case
    put entry on it, set newchildentry to null, and return;
    else,                                          // once in a while, the leaf is full
        split L: first d entries stay, rest move to brand new node L2;
        newchildentry = & (⟨smallest key value on L2, pointer to L2⟩);
        set sibling pointers in L and L2;
        return;
endproc
```

Figure 9.11 Algorithm for Insertion into B+ Tree of Order d

```
proc delete (parentpointer, nodepointer, entry, oldchildentry)
// Deletes entry from subtree with root '*nodepointer'; degree is d;
// 'oldchildentry' null initially, and null upon return unless child deleted
if *nodepointer is a non-leaf node, say N,
    find i such that K_i ≤ entry's key value < K_{i+1};          // choose subtree
    delete(nodepointer, P_i, entry, oldchildentry);              // recursive delete
    if oldchildentry is null, return;              // usual case: child not deleted
    else,                                  // we discarded child node (see discussion)
        remove *oldchildentry from N,      // next, check minimum occupancy
        if N has entries to spare,                                    // usual case
            set oldchildentry to null, return;          // delete doesn't go further
        else,                          // note difference wrt merging of leaf pages!
            get a sibling S of N:                  // parentpointer arg used to find S
            if S has extra entries,
                redistribute evenly between N and S through parent;
                set oldchildentry to null, return;
            else, merge N and S                                // call node on rhs M
                oldchildentry = & (current entry in parent for M);
                pull splitting key from parent down into node on left;
                move all entries from M to node on left;
                discard empty node M, return;

if *nodepointer is a leaf node, say L,
    if L has entries to spare,                                        // usual case
        remove entry, set oldchildentry to null, and return;
    else,                          // once in a while, the leaf becomes underfull
        get a sibling S of L;                      // parentpointer used to find S
        if S has extra entries,
            redistribute evenly between L and S;
            find entry in parent for node on right;                   // call it M
            replace key value in parent entry by new low-key value in M;
            set oldchildentry to null, return;
        else, merge L and S                                // call node on rhs M
            oldchildentry = & (current entry in parent for M);
            move all entries from M to node on left;
            discard empty node M, adjust sibling pointers, return;
endproc
```

Figure 9.16 Algorithm for Deletion from B+ Tree of Order d