```
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 $K_i \leq$ 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