3. C Functions

We can write a deletion algorithm with overall structure similar to that used for insertion. Again we shall use recursion, with a separate main function to start the recursion. Rather than attempting to pull an entry down from a parent node during an inner recursive call, we shall allow the recursive function to return even though there are too few entries in its root node. The outer call will then detect this occurrence and move entries as required.

The main function is:

B-tree deletion

```
/* DeleteTree: deletes target from the B-tree.
   Pre:  target is the key of some entry in the B-tree to which root points.
   Post: This entry has been deleted from the B-tree.
   Uses: RecDeleteTree. */
Treenode *DeleteTree(Key target, Treenode *root)
{
   Treenode *oldroot;          /* used to dispose of an empty root          */
   RecDeleteTree(target, root);
   if (root->count == 0) {     /* Root is empty.                            */
      oldroot = root;
      root = root->branch[0];
      free(oldroot);
   }
   return root;
}
```

4. Recursive Deletion

Most of the work is done in the recursive function. It first searches the current node for the target. If it is found and the node is not a leaf, then the immediate successor of the key is found and is placed in the current node, and the successor is deleted. Deletion from a leaf is straightforward, and otherwise the process continues by recursion. When a recursive call returns, the function checks to see if enough entries remain in the appropriate node, and, if not, moves entries as required. Auxiliary functions are used in several of these steps.

```
/* RecDeleteTree: look for target to delete.
   Pre:  target is the key of some entry in the subtree of a B-tree to which current points.
   Post: This entry has been deleted from the B-tree.
   Uses: RecDeleteTree recursively, SearchNode, Successor, Remove, Restore. */
void RecDeleteTree(Key target, Treenode *current)
{
   int pos;                          /* location of target or of branch on which to search */
   if (!current) {
      Warning("Target was not in the B-tree.");
      return;                        /* Hitting an empty tree is an error.                 */
   } else {
      if (SearchNode(target, current, &pos))
         if (current->branch[pos-1]) {
            Successor(current, pos);  /* replaces entry[pos] by its successor              */
            RecDeleteTree(current->entry[pos].key, current->branch[pos]);
         } else
            Remove(current, pos);     /* removes key from pos of *current                  */
      else                           /* Target was not found in the current node.          */
         RecDeleteTree(target, current->branch[pos]);
      if (current->branch[pos])
         if (current->branch[pos]->count < MIN)
            Restore(current, pos);
   }
}
```

```
/* Restore: restore the minimum number of entries.
   Pre:  current points to a node in a B-tree with an entry in index pos; the branch to
         the right of pos has one too few entries.
   Post: An entry taken from elsewhere is to restore the minimum number of entries by
         entering it at current->branch[pos].
   Uses: MoveLeft, MoveRight, Combine. */

void Restore(Treenode *current, int pos)
{
   if (pos == 0)                    /* case: leftmost key                    */
      if (current->branch[1]->count > MIN)
         MoveLeft(current, 1);
      else
         Combine(current, 1);

   else if (pos == current->count)  /* case: rightmost key                   */
      if (current->branch[pos-1]->count > MIN)
         MoveRight(current, pos);
      else
         Combine(current, pos);

                                    /* remaining cases */
   else if (current->branch[pos-1]->count > MIN)
      MoveRight(current, pos);

   else if (current->branch[pos + 1]->count > MIN)
      MoveLeft(current, pos + 1);

   else
      Combine(current, pos);
}
```