Adam Drozdek, Dušica Vujanović

Duquesne University,
Department of Mathematics and Computer Science,
Pittsburgh, PA 15282, USA

Atomic-key B-trees

Abstract. Atomic-key B-trees are B-trees with keys of different sizes. This note presents two versions of an insertion algorithm and a deletion algorithm for atomic-key B-trees.

Keywords: atomic-key B-tree, data structures, database management.

1. Introduction

In regular B-trees, the number of keys in one node is predefined and customarily the same amount of space is used for each key. A modification introduced in prefix B-trees allows for keys to be of different size; however, this can only be done if these keys share the same prefix. There is thus a need to address the problem of B-trees that can store any keys of variable length. One solution is offered by the *string B-tree* in which the keys are divided and distributed between different nodes of the tree ([3, 5]). The *atomic-key B-tree* offers another solution in which keys are atomic by being stored in their entirety ([4], ch. 5; [1]).

2. The static atomic-key B-tree

In the static version of insertion in the atomic-key B-tree, Hu proposes that a set of keys is divided into $f = \max\left(3, \left\lfloor \frac{B}{\hat{k}} \right\rfloor\right)$ subsets (keys in each set are ordered), where B is the block size and $\hat{k}$ is the average key length. Hu uses the top-down technique to build the tree: all keys are divided up into f sets and from each set, except for the first and the last, shortest key is chosen and all these shortest keys are put in the root. The f–2 keys in the root then separate all keys into f–1 subsets and each of these subsets is put in a descendant of the root. For example, from the four subsets {*a*, *b*, *c*}, {*dd*, *ee*, *f*}, {*gg*, *h*, *ii*}, {*jj*, *k*, *l*}, the root would be {*f*, *h*} and there would be three descendants of the root, {*a*, *b*, *c*, *dd*, *ee*}, {*gg*}, {*ii*, *jj*, *k*, *l*}. If a subset is too large and cannot be accommodated in a node, the subset is divided into f subsets (f is determined anew for each subset according to the average key length in this subset), the shortest key from each subset, except for the first and the last, is put in the node, the shortest keys separate keys in the node into groups that are put in descendants of this node. The process continues until no node splitting is needed. It should be clear that leaves may not be on the same level. Another problem is the appearance of empty leaves. For a block of size 50 and six keys K_1, …, K_6, each of size 10, the average length is 10, f = 50/10 = 5, and thus five sets are created, say, {K_1}, {K_2}, {K_3},{K_4}, {K_5, K_6}, whereby the root becomes {K_2, K_3, K_4}

with two nonempty descendants, $\{K_1\}$ to the left of K_2 and $\{K_5, K_6\}$ to the right of K_4, and two empty descendants, one between K_2 and K_3, and another between K_3 and K_4.Another scenario is illustrated in Figure 2.1. The shortest keys from the three sets of keys are put in the root. The four sets of keys separated by these shortest keys are shown as tall rectangles. They will be used to form the four descendants of the root. However, at least the tall rectangle before A is too large to be accommodated in one node and it will be later processed to acquire its own descendants; the same appears to be the case with the tall rectangle after C. Because of choosing the rightmost key B from its set and the leftmost key C from its set, the tall rectangle between them is empty resulting in an empty leaf.

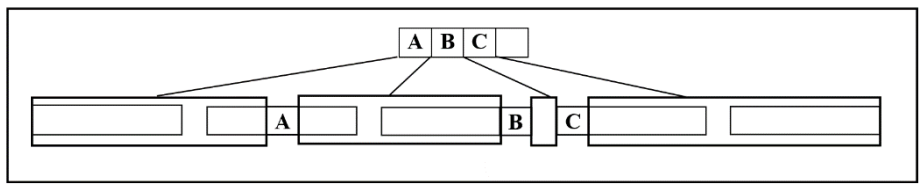

Figure 2.1. Possible positions of shortest keys in their sets

3. The dynamic atomic-key B-tree

The static version requires that all keys are known before the tree is created – to determine the average key length – which limits its applicability. A more interesting situation is a dynamic case when one key is inserted in the tree without knowing which keys will be inserted next. In a B-tree, the height of the tree depends on the number of descendants: the more children each node can have, the smaller the overall height of the tree becomes. For variable length keys, it is thus a good idea to put shorter keys high up in the tree, since more keys can be accommodated in one node and the node can have more children. Each algorithm for insertion in the atomic-key B-tree has to take this consideration into account.

When the pure top-down technique from the static atomic-key B-tree is applied in the dynamic tree, then upon a node overflow a split results in the shortest key staying in the node being split with the remaining keys along with the new arrival being distributed between two newly created leaves that become descendants of this node. In this way, nonterminal nodes would contain only one key. The problem can be addressed in at least two ways, and in this article we propose two insertion techniques to be used in the dynamic atomic-key B-tree, one that adopts the traditional way of building B-trees bottom-up to build atomic-key B-trees and another which combines bottom-up and top-down ways of handling splits.

4. A bottom-up version of the dynamic atomic-key B-tree

Inserting a key into a leaf that has some room would proceed as in a regular B-tree with updates of relevant fields in the leaf to reflect the new situation. The keys have to be ordered – for strings this would be alphanumeric order – which most of the time requires that the positions of existing keys are reconfigured. Otherwise, the leaf has to be split to promote a key, which may cascade into splits of its parent, grandparent, etc.

There are two problems that need to be addressed when a split occurs while

inserting a key using the bottom-up approach. One problem is the choice of a key to be promoted to the parent.

When a leaf is full, it needs to be split. In a B-tree, the middle key is promoted to the parent and two siblings get a half of keys each. In the atomic-key B-tree, the idea is to promote the shortest key. However, it should not be done unconditionally. If a new key is shortest and, at the same time, it precedes all other keys in the node, then by promoting it to the parent one of the sibling nodes would be empty and another sibling would have the same keys as before splitting, minus the shortest key. This might be acceptable since the empty leaf would not be included in the tree. However, if a new key is the shortest, but it precedes most (but not all) of the keys – or follows most of them – then, after splitting the node and promoting the shortest key, one sibling would have very few keys and another sibling would have most of them. That could result in a tree with some – possibly many – nodes almost empty and consequently in a large number of nodes in the tree, and also in a high imbalance of the tree. To avoid this problem, the shortest key is used for promotion if at least 25% of keys in the node follow this key and when 25% precede it. If this is not the case, then a middle key – like in a regular B-tree – is used for promotion. Thus, in the worst case, when keys are arriving randomly, a node (except for the root) can be only 25% full.

Here is an algorithm for inserting a new key:

```
insertBottomUp(key K)
   if root is null
      root = a new node that includes K;
      return;
   else node = a leaf in which K should be inserted;
   left = right = null;
   while (true)
      if there is enough room for K in node
         incorporate K in node along with pointers to left and right;
         return;
      else if the shortest key is among the first fourth or among
                  the last fourth of keys in node plus K
           K = the middle key among keys in node plus K;
      else K= the shortest key among keys in node plus K;
      left = node with keys from node that precede K;
      right = a new node with keys from node that follow K;
      if node is the root
         root = a new node that includes K and pointers
                to left and right;
         return;
      else node = the parent of node;  //to try to insert K in it;
```

If a key *K* to be inserted is the smallest and falls among the first fourth of keys in a node (Figure 3.1a), or in the last fourth, then a middle key *M* from this node is elected for promotion, the node is split into two, the keys are divided up evenly between the two resulting nodes, and *M* is inserted in the parent of the two nodes (Figure 3.1b) if it was a

parent of the node before splitting; when there was no parent, a new parent is created. It may happen, however, that there is no room in the parent, then the splitting process would be applied to this parent and so on up the tree, possibly to the root. Shaded areas in figures to follow indicate areas occupied by keys.

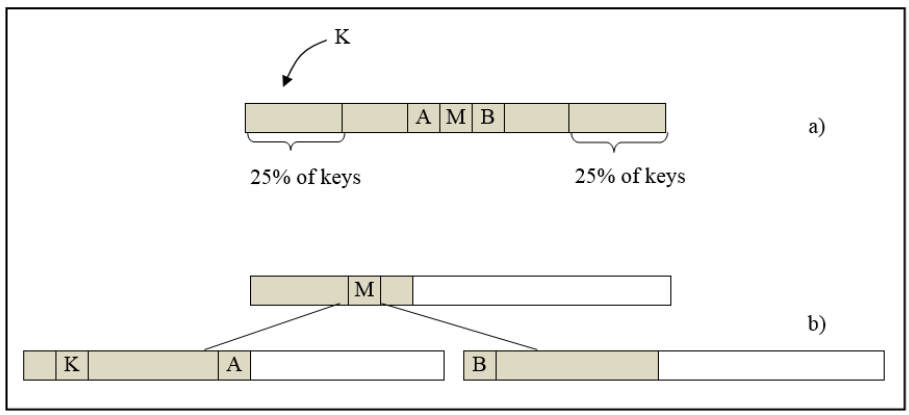

Figure 3.1. Inserting a new key *K* that is shorter than all keys in a full node and belongs among the keys in the first fourth of keys; (a) before insertion, (b) after insertion

If a key *K* to be inserted is larger than the smallest key *S* in a node, but *S* is in the first fourth among keys (Figure 3.2a), then a middle key *M* is promoted to the parent of the node (if any; otherwise, a new parent is created) and the node is split and the nodes obtained from splitting become descendants of the same parent (Figure 3.2b).

If a key *K* to be inserted is the smallest and falls in the second or third fourth of the keys in a node (Figure 3.3a), *K* is moved to the parent of the node and the node itself is split (Figure 3.3b).

If *K* is larger than the smallest key *S* and *S* falls in the second or third fourth of the keys (Figure 3.4a), *S* is promoted and the node is split integrating *K* along the way (Figure 3.4b).

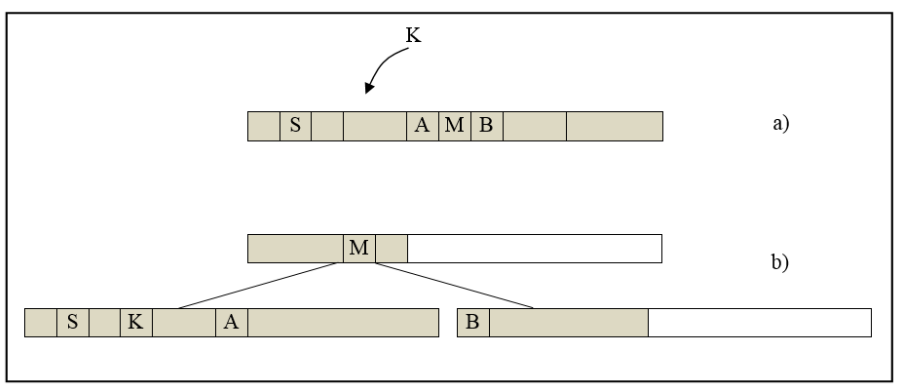

Figure 3.2. Inserting a new key K in a full node when the shortest key S is among the first fourth of keys; (a) before insertion, (b) after insertion.

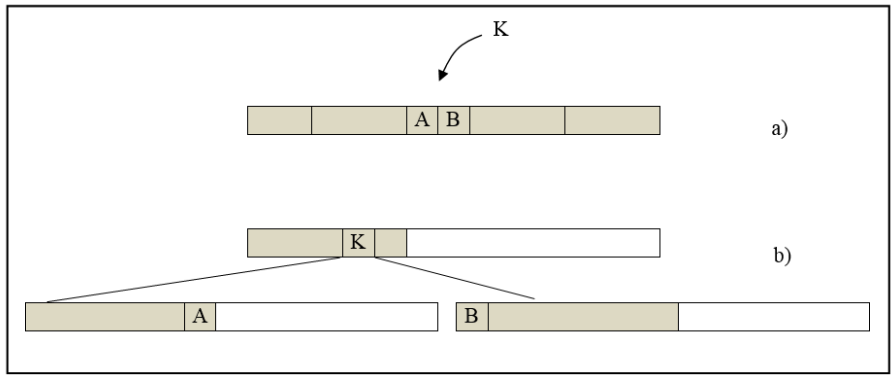

Figure 3.3. Inserting a new key K that is shorter than all keys in a full node and belongs among the keys in the mid-part of the node; (a) before insertion, (b) after insertion.

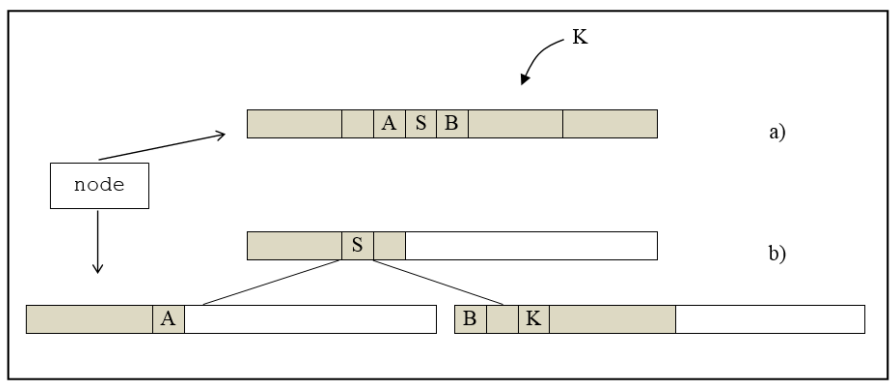

Figure 3.4. Inserting a new key K in a full node when the shortest key S is in the mid-part of the node; (a) before insertion, (b) after insertion.

There is a problem with long keys, longer than the maximum length that could be accommodated in one node. To solve the problem for such a key *K*, a new key can be created with the maximum permitted length extracted from *K* with a special character appended to its end to indicate the truncation and a number that would indicate the position of the cut-off part in an overflow file. Preferably, this number would be in binary form to assure that the same number of bytes is always used for its representation. Another possibility is that the trailing part of an overlong key would be placed in a descendant node. Admittedly, such a situation would be extremely rare considering the size of nodes (normally, at least half a kilobyte).

Although the primary criterion for choosing a key for promotion is the size of the key, the relative order of keys remains the same in an atomic-key B-tree as in the regular B-tree; therefore, the search proceeds in the same way in both trees: when a key *K* is not found in a particular node, the search continues in a descendant accessible through the pointer shared by keys K_1 and K_2 for which $K_1<K \leq K_2$, unless *K* precedes all keys in the node, thus the descendant accessible through the leftmost pointer is chosen, or *K* follows

all keys in the node, in which case the search continues in a descendant accessible through the rightmost pointer.

Several variations of the atomic-key B-tree are possible. Here are some possibilities:

1. Instead of choosing the middle key to avoid imbalance of the tree, the shortest key among the second and third fourths of keys in a node can be found for promotion.
2. Instead of choosing a middle key, choose the key that is the closest to the middle of the array of keys; this would divide the key area almost evenly, but it could lead to an imbalance; for example, if a node has four keys, *abcd efghijklmno*, and a split must be made, then the last key *ghijklmno* would be chosen for promotion since it begins in the middle of the array, resulting in high imbalance. However, for large number of keys in a node, such a situation would be quite unlikely.
3. In the case of a tie, when there are multiple keys of the same shortest length, choose the key that is closest to the middle key (or, alternatively, closest to the middle of the key area).
4. An imbalance, however, will always be possible, unlike in a regular B-tree. If this imbalance becomes too worrisome, then instead of allowing the shortest key to be among the second and third fourths, the key would be required to be among the middle third of keys. On the other hand, the restriction on imbalance can be relaxed if the shortest key for promotion can be chosen among the second, third, and fourth fifths of keys. To be sure, other numbers can also be used.
5. Interpret percentages in terms of the amount of space rather than in terms of the number of nodes. So, in the node of size 400, the first fourth of keys (25%) would be the keys occupying the first 100 positions of the node (with possible spillover of the last key to the next positions) and the last fourth would be keys in the last 100 positions.

5. A hybrid version of the dynamic atomic-key B-tree

To reduce underflow, another algorithm can be used in which an overflow during insertion still leads to a split, but there are two possibilities: 1. if an overflowed node's parent has room for the shortest key from this node, then this shortest key is moved there and the node itself is split and all keys are properly redistributed, as in Figure3.4. This is a bottom-up insertions the way it is done in the previous algorithm and, generally, in regular B-trees. 2. Another possibility arises when the parent of an overflowed node has no room for the shortest key from this node, in which case two nodes are created for the keys from this node and the newly arrived key; only the shortest key remains in this node and the newly created nodes become its two descendants (Figure 5.1). One of them may be overflowed, so the splitting process has to be applied to it. This is a top-down split performed in the way proposed by Hu for his static atomic-key B-tree. Here is a pseudocode of this algorithm:

```
insertHybrid(key K)
    if root is null
        root = a new node that includes K;
        return;
    else node = a leaf in which K should be inserted;
    while (true)
        if there is enough room for K in node
```

```
        incorporate K in node;
        return;
    else if node ≠ root and there is enough room in node's parent
            to include the shortest key SK from node plus K
        move SK to the parent;
        split node and distribute its keys plus K among
            node and its newly created sibling;
        set pointer to the left of SK to node and pointer to
            the right of SK to node's sibling;
    else SK = the shortest key among keys in node plus K;
        left = a node with keys from node that precede SK;
        right = a node with keys from node that follow SK;
        put SK in node;
        attach left and right as descendants of node;
        if left or right is overflowed
            node = left or right;
            K = the shortest key from left or right;
        else return;
```

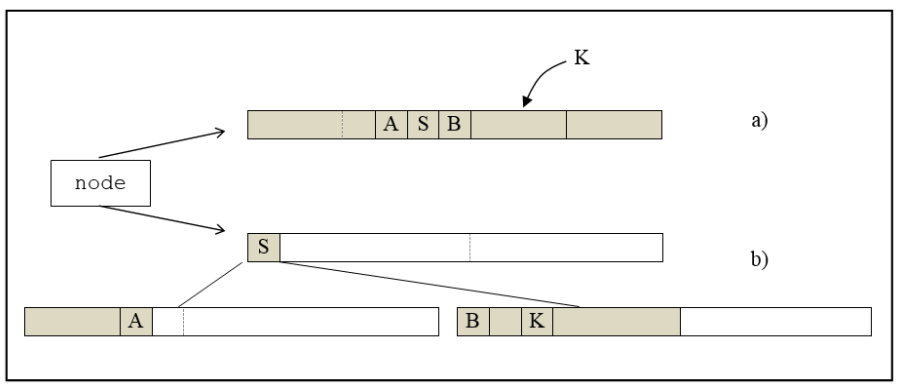

Figure 5.1. Inserting a new key *K* in a full node when there is no room for the shortest key *S* is in the parent of the node; (a) before insertion, (b) after insertion.

6. Comparison of insertion algorithms

The height of the B-tree $h \leq \log_q \frac{N+1}{2} + 1$, where $q = \frac{m}{2}$ and m is the order of the tree([2], p. 314). This gives the number of node accesses for insertion of a new key and the worst case for the number of accesses when searching for a key in the tree. For the atomic-key B-tree, expected search for insertion and search is given by $O\left(\left\lceil\frac{\hat{k}}{B}\right\rceil \log_{1+\lfloor\frac{B}{\hat{k}}\rfloor} N\right)$ ([4], p. 123; [1], p. 306). To see how these two logarithmic formulas compare to one another, consider processing keys with the maximum length = 10 (which will be the space allocated for one key in the B-tree), the average length $\hat{k}$ = 7 and to assure the same size of nodes for both types of trees, the order of the B-tree $m = \frac{B}{\text{maximum key length+size of reference/pointer}}$:

Table 6.1. Comparison of expected numbers of accesses for regular and atomic-key B-trees

N	B	$\left\lceil \frac{\hat{k}}{B} \right\rceil \log_{1+\lfloor \frac{B}{\hat{k}} \rfloor} N$	$\log_q \frac{N+1}{2} + 1$
1000	100	2.49	6.66
1000	500	1.61	3.36
1000	4000	1.09	2.32
10000	100	3.32	8.75
10000	500	2.15	4.23
10000	4000	1.45	2.81
100000	100	4.15	10.85
100000	500	2.68	5.10
100000	4000	1.81	3.29
1000000	100	4.98	12.94
1000000	500	3.22	5.97
1000000	4000	2.18	3.78

This table indicates that the average number of node accesses for search/insertion in the B-tree is always larger than in the corresponding atomic-key B-tree; however, the difference decreases with the increase of the size of one block/node. These are just estimates and it is interesting to see to what extent experimental runs reflect these differences.

To compare the efficiency of insertion algorithms, implementations of the three types of trees – the B-tree, bottom-up atomic-key B-tree, and hybrid atomic-key B-tree – have been run on the same file that contained randomly generated strings of length 1 to 10, with the average length equal to 7. The program was run for three different numbers N of keys – 1000, 10000, and 100000 – which, because of the random nature of these keys, included only small number of duplicates (ca. 10%). The program was also run for different sizes of blocks/nodes – 100, 500 (i.e., ca. 0.5 KB), and 4000 (ca. 4 KB) – assuming that the nodes included not only keys themselves, but also references/pointers. The only optimization used in the implementation of the insertion algorithm in the hybrid atomic-key B-tree was choosing a shortest key closest to the middle of the set of keys in a node when there were at least two keys of the shortest length. The results are summarized in the following tables:

Table 6.2. Processing efficiency of B-trees

N	B	B-tree			
		number of nodes	occupied space %	height	average of node accesses
1000	100	344	62.5	5	4.63
1000	500	53	67.6	3	2.38
1000	4000	9	47.8	2	1.78
10000	100	3343	61.4	7	6.28
10000	500	499	69.2	3	2.93
10000	4000	65	63.5	2	1.98
100000	100	31863	60.9	9	8.01
100000	500	4821	68.0	4	3.86
100000	4000	534	74.1	3	2.63

Table 6.3. Processing efficiency of hybrid atomic-key B-trees

N	*B*	Hybrid atomic-key B-tree			
		number of nodes	occupied space %	height	average of node accesses
1000	100	326	51.8	5	3.72
1000	500	65	48.6	3	1.95
1000	4000	7	55.6	2	1.74
10000	100	3189	51.4	7	4.91
10000	500	551	55.7	3	2.72
10000	4000	54	70.1	2	1.92
100000	100	30824	51.2	9	6.06
100000	500	6233	47.4	4	3.21
100000	4000	882	41.3	3	2.03

Table 6.4. Processing efficiency of bottom-up atomic-key B-trees

N	*B*	Bottom-up atomic-key B-tree			
		number of nodes	occupied space %	height	average of node accesses
1000	100	257	68.4	5	4.35
1000	500	52	61.3	3	2.47
1000	4000	5	77.8	2	1.75
10000	100	2490	68.6	7	6.01
10000	500	459	67.3	4	2.91
10000	4000	56	67.6	2	1.95
100000	100	23918	68.8	9	7.45
100000	500	4388	67.8	4	3.73
100000	4000	516	70.6	3	2.63

The bottom-up atomic-key B-tree consistently required fewer nodes than the regular B-tree although both trees were of the same height and in one case the former was even higher by one level than the latter. Except for one case, the bottom-up atomic-key B-tree also required on the average fewer node accesses than the B-tree (average of node accesses is defined as the ratio of all node accesses both for insertion of new nodes and for the search of duplicates to the number *N* of keys). Consistently, nodes in the bottom-up atomic-key B-tree are at least 60% full, which is similar to the B-tree; however, the B-tree has a problem with internal fragmentation because of assigning the same amount of space to all keys.

The hybrid atomic-key B-tree has the best averages of nodes accesses, better than the other two trees even though it sometimes has more nodes than the other two trees. The reason for this larger number of nodes is that always a shortest key is elected during split and if there is only one key of the shortest length, it can be one of the leftmost or one of the rightmost keys in the node. This means that one of the two nodes resulting from the split will be almost full and will very likely require a split really soon.

Table 6.5. Processing efficiency of the combined atomic-key B-tree

N	*B*	Combined atomic-key B-tree			
		number of nodes	occupied space %	height	average of node accesses
1000	100	297	56.8	5	3.70
1000	500	63	50.1	3	1.96
1000	4000	7	55.6	2	1.74
10000	100	2962	55.4	7	4.91
10000	500	497	61.7	3	2.75
10000	4000	53	71.3	2	1.92
100000	100	28370	55.7	9	6.07
100000	500	5632	52.4	4	3.26
100000	4000	882	41.3	3	2.03

To restrain this occasional overabundance of nodes in the hybrid tree, the cautious choice of the key during split as applied in the bottom-up tree can also be applied in the hybrid tree. The results of the behavior of such a tree are presented in the table 6.5.

Disappointingly, the number of nodes in such a combined tree is the same or fairly close to the number of nodes in the hybrid tree and, appreciably, the average node accesses are the same or very similar to the averages obtained for the hybrid tree. The reason is that, particularly for large blocks, there can be multiple keys of shortest length spread over a node and thus the shortest key elected during a split would be the same as in the hybrid method. Only when there are lone shortest keys close to the left or to the right border of the node will the combined method chose a key guaranteeing an even split of keys.

All these results suggest that when the space is at the premium, the bottom-up version of the tree would be preferable; if the number of accesses should be minimized, then the hybrid version would be the tree of choice. The middle-course applied in the combined atomic-key B-tree does not appear to provide any significant advantage over the two other kinds of the atomic-key B-tree.

7. Deletion from the dynamic atomic-key B-tree

Another operation worth mentioning is deletion. Here is a possible algorithm.

```
delete(key K)
    node = the node that contains K;
    if node is a leaf
        if K is the last key in node
            remove the leaf;
        else delete K from node;
        return;
    if K is the last key in node
        rebuild the tree rooted at the parent of node;
    else left = a child node to the left of K in node;
        right = a child node to the right of K in node;
```

```
remove K from node;
if there is enough room in node for the rightmost key
        in left or the leftmost key in right
    move this key to node;
    if the child from which the key was moved is not a leaf
        if the last key was moved from this child
            rebuild the subtree rooted at node;
        else rebuild the subtree rooted at this child;
    else if the child is a leaf from which the last key was removed
        remove the leaf;
else rebuild the subtree rooted at node;
```

To illustrate deletion, consider the situation in Figure 7.1. If there is enough room in the node after removal of *K* to place either C or D in its place (possibly shifting keys from B onwards to the right), then C or D is placed there. If the child from which the key was transferred is a leaf, then nothing else needs to be done if the key is C or, if it is D, all keys in the leaf have to be shifted to the left to plug the hole. If none of the children is a leaf, then the subtree rooted at the child from which a key was taken to replace *K* has to be rebuilt using the method for static atomic-key B-tree. The deletion operation can be performed on average in $O\left(\left\lceil\frac{k}{B}\right\rceil \log_{1+\left\lceil\frac{B}{k}\right\rceil} N\right)$ node accesses ([4], p. 144; [1], p. 311).

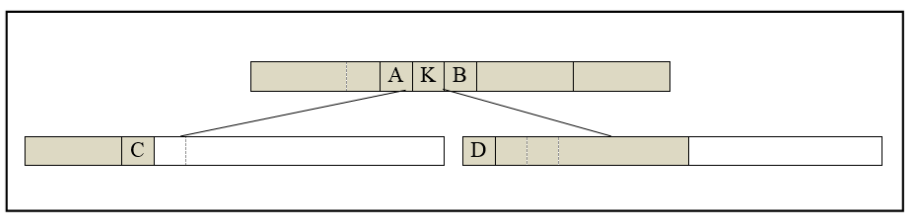

Figure 7.1. A situation before the deletion of *K*. The descendant nodes can be leaves or non-terminal nodes

8. Conclusion

The atomic-key B-tree in some of its manifestations determined by an insertion algorithm is a very attractive alternative to traditional B-trees. Just as B-trees can be fairly easily changed to B^+-trees, so could the atomic-key B-trees be upgraded to atomic-key B^+-trees, and thereby become used a replacement of B^+-trees in implementation of indexes in databases.

References

1. Bender M.A., Hu H., Kuszmaul B.C., Performance guarantees for B-trees with different-sized atomic keys, *Proceedings of the 29th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, New York, ACM 2010, pp. 305-316.
2. Drozdek A., *Data structures and algorithms in C++*, Boston, Cengage Learning 2013.

3. Ferragina P., Grossi R., The string B-tree: A new data structure for string search in external memory and its applications, *Journal of the ACM* 46 (1999), pp. 1-46.
4. Hu H., *Cache-oblivious data structures for massive data sets*, PhD diss., New York, Stony Brook University 2007.
5. Na J.Ch., Park K., Simple implementation of string B-trees, in: Apostolico A., Melucci M. (eds.), *String Processing and Information Retrieval*, Berlin, Springer 2004, pp. 214--215.