J Comput Virol (2007) 3:211–219
DOI 10.1007/s11416-007-0035-3

ORIGINAL PAPER

B-tree with fixed node size and no minimum degree

Chandra Prakash

Received: 3 December 2006 / Accepted: 3 January 2007 / Published online: 23 February 2007

Abstract In anti-virus and anti-spyware applications, due to multiplicative increase in processing times with increasing complexity of detection logic and fast growing number of signatures, there is a necessity for data structures for quick retrieval and efficient storage of large collection of signatures. This paper presents a variant of B-tree data structure, where the minimum degree constraint is relaxed while maintaining the order of worst case performance bounds for primitive search, insert and delete operations of the B-tree. It presents a detailed case study of the impact of key (signature) size on storage utilization, given fixed sized nodes and also derives a maximum optimal key size with respect to node size. This variant of B-tree is found to be specifically very useful for storage of large number of keys where size of keys exhibit a wide variation and node size remains fixed.

Keywords Byte · B+-tree · Prefix B-tree · String B-tree

1 Introduction

A B-tree is a data structure for efficient external searching and sorting. As per Cormen et al. [1], a B-tree is characterized by its fixed *minimum degree t*, which is an integer, such that $t \geq 2$, and:

C. Prakash (✉)
Sunbelt-Software, Clearwater,
Florida 33755, USA
e-mail: chandraprakash7@gmail.com

1. Every node other than the root must have at least $t - 1$ keys. Every internal node other than the root thus has at least t children. If the tree is non-empty, the root must have at least one key.
2. Every node can contain at most $2t-1$ keys. Therefore, an internal node can have at most $2t$ children.

With these constraints, it is very inefficient to store a large number of keys which exhibit a wide variation in size, given that the node size is fixed. For example, consider a B-tree implementation where node size is fixed to 2,048 bytes. The key size can vary from 1 to 1,500 bytes. If the minimum degree t is chosen as 100, consider a scenario where all keys have size of 1 byte. In this scenario, if we go by the traditional B-tree and apply constraint 2, 1,849 $[2,048 - (2 \times 100 - 1)]$ bytes in every node will be wasted. On the other extreme if all keys are of size $1,500$ bytes constraint 1 will be violated.

The issues pointed out with the traditional B-tree can be circumvented by choosing a variable sized node. But, having a variable sized node has its own limitations. A variable sized node lends itself to issues of fragmentation and compaction as new keys are added or old keys are removed. For example, consider a scenario of new keys being added in a variable sized node B-tree. One of the new keys being added belongs to node n_1, next to which lies node n_2. To make room for the new key, node n_2 and all subsequent node that follow n_2 need to be moved up. If all the nodes are persisted in one file, the cost of making room for a new key is inversely proportional to its destination offset from the beginning of the file. On the other hand, deleting keys leads to holes between adjacent nodes which are expensive to fill.

In contrast to prefix B-trees by Bayer and Unterauer [2] and string B-trees by Ferragina and Grossi [3], which are variations of B+-tree (where some or part of the keys are stored in the internal nodes [4,5]), this implementation has more kinship to the basic B-tree (Cormen et. al. [1]) with all keys stored in the internal nodes and worst cast performance same as the basic B-tree. Also unlike the string B-tree, which allows keys of arbitrary length, key sizes in this implementation are bounded by node size. One would ask: What are the advantages of this implementation with this size restriction? The answer is: this implementation presents a detailed case study of impact of key size on the basic B-tree and presents an algorithm that addresses issues presented earlier in this section. Further, this implementation enunciates an upper bound on the optimal key size (with respect to node size) that circumvents creating empty nodes.

2 Definition of B-tree with no minimum degree

Due to the issues discussed in previous section with a B-tree where minimum degree is based on the number of keys, a new variant of B-tree, abbreviated as $B^{-}MD$-*tree* (B-tree minus minimum degree), is proposed where node size is fixed. A $B^{-}MD$-*tree* is a rooted tree (with root *root* $[T]$) having following properties:

1. Every node x has the following fields:
   (a) $n[x]$, the number of keys currently stored in node x,
   (b) the $n[x]$ keys themselves, stored in non-decreasing order:
   $\text{key}_1[x] \leq \text{key}_2[x] \leq \cdots \text{key}_{n[x]}[x]$,
   and,
   (c) *leaf[x]*, a Boolean value that is *TRUE* if x is a leaf and *FALSE* if x is an internal node.
   (d) *sizeof* (a), is the length of space occupied in bytes by the parameter a. The parameter a is either a node or a key.
2. If x is an internal node, it also contains $n[x]+1$ pointers $c_1[x], c_2[x], \ldots, c_{n[x]+1}[x]$ to its children. Leaf nodes have no children, hence their c_i fields are *NIL*.
3. The keys $keys_i[x]$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $c_i[x]$, then

$$k_1 \leq \text{key}_1[x] \leq k_2 \leq \text{key}_2[x] \\ \leq \cdots \text{key}_{n[x]}[x] \leq k_{n[x]+1}.$$

4. Every leaf has the same level, which is the tree's height h.
5. Every node size is fixed to B, even though it can contain variable sized keys. Each node must have at least one key. There is no upper bound on the number of keys contained in a node as long it has space available to accommodate new keys. But, there is a limitation on the maximum size of a key. The sum of size of a key and the size of two child pointers surrounding it cannot exceed the size of the node.

Adding keys whose sizes are greater than a certain threshold (see Sect. 4) can create empty internal or leaf nodes in a $B^{-}MD$-*tree*. In practical implementations the size of child pointer fields must also be taken into consideration as they are intrinsic part of the node. But, for simplicity sake from now on child pointer sizes will be ignored with respect to key and node size.

Although algorithms presented in this paper assume key sizes to be less than or equal to *optimal* maximum key size (see Sect. 4), to prevent creating empty nodes, yet it is practicable to hypothesize that these algorithms can be made amenable to key sizes up to B and hence to existence of empty nodes. For the purpose of discussion now, assume the key size to be less than node size and let's defer the significance of key size, until we reach Sect. 4.

3 Basic operations in $B^{-}MD$-*tree*

The basic operation set includes search, insertion and deletion. It is assumed that $B^{-}MD$-*tree* is *created* at least once, which is simply having an empty root node with zero keys and no children.

3.1 Search

A non-recursive search algorithm for a $B^{-}MD$-*tree* is described in Algorithm A.1 given in Table 1. This version of search algorithm is very similar to B-tree search described in Cormen et al. [1].

Table 1 Algorithm A.1—Non-recursive B⁻MD-TREE-SEARCH

```
B-MD-TREE-SEARCH(x, k)
1     w ← x
2     while forever
3     do
4             i ← 1
5             while i ≤ n[w] and k > key_i[w]
6             do
7                     i ← i +1
8             if i ≤ n[w] and k = key_i[w]
9             then
10                    return (w, i)
11            if leaf[w]
12            then
13                    return NIL
14            else
15                    w = DISK-READ(c_i[x])
```

Table 2 Algorithm A.2—non-recursive B⁻MD-TREE-INSERT

```
B⁻MD-TREE-INSERT($T$, $k$)
1   $w \leftarrow root[T]$
2   $l \leftarrow 0$
3   while forever
4   do
5           $i \leftarrow 1$
6           $a[l] \leftarrow address(w)$
7           while $i \le n[w]$ and $k > key_i[w]$
8           do
9                   $i \leftarrow i + 1$
10          if $leaf[w]$
11          then
12                  break
13          else
14                  $w$ = DISK-READ($c_i[x]$)
15                  $l \leftarrow l + 1$
16  B⁻MD-TREE-INSERT-AT-LEVEL($T$, $w$, $k$, $NIL$, $a$, $l$)
```

3.2 Insertion

Insertion algorithm in a *B^-MD-tree* is described in Algorithm A.2 given in Table 2. It is assumed that the key to be inserted in the *B^-MD-tree*, is new. This algorithm does not always finish in one downward pass and can back up. This is one of the main differences between this algorithm and the insertion algorithm for a B-tree described in Cormen et al. [1] The backing up manifests due to absence of internal node splits during downward pass. The pre-emptive node splits during downward pass are untenable because of variable sized keys, specifically the median key in the child node may not fit in the parent node. In the B⁻ MD-TREE-INSERT algorithm, between lines 3 and 15, the leaf node where the new key k belongs is located. During this lookup, the address of node traversed at each level is stored in an array a indexed by the level l. After leaf node is reached, B⁻MD-TREE-INSERT-AT-LEVEL is called (see Algorithm A.3 in Table 3 in the Appendix), which inserts the new key at a specified level.

At line 5 of routine B⁻MD-TREE-INSERT-AT-LEVEL, a check is made to see if the destination node has enough space for the new key. At line 18, routine B⁻MD-TREE-SPLIT-NODE (see Algorithm A.4 in Table 4 in the Appendix) is called to split the destination node into two nodes, if it does not have enough room for the new key. In routine B⁻MD-TREE-SPLIT-NODE, a median key (see Sect. 4) is found in the node being split. In lines 13–26, new key is inserted in one of the appropriate node among the two split nodes. At line 25 the address of child node c to the right of new key, is set. The median key k' at line 40 is set to be moved up in the parent, which starts another cycle of inserting a node at specified level at line 4 in routine B⁻MD-TREE-INSERT-AT-LEVEL, after B⁻MD-TREE-SPLIT-NODE returns. Lines 31–36 in routine B⁻MD-TREE-SPLIT-NODE account for increase in height of the tree, when the root node gets split (see Fig. 1).

3.3 Deletion

Deletion algorithm in a *B^-MD-tree* is described in Algorithm A.5 given in Table 5 in the Appendix. Contrary to B-tree deletion algorithm in Cormen et al. [1], in *B^-MD-tree* deletion, there can be two passes, one definite downward pass and two likely upward passes. The B-tree deletion algorithm in Cormen et al. [1], safeguards the condition that every node traversed in the downward pass has at least t keys. The B⁻MD-TREE-DELETE algorithm has no such provisions whatsoever.

The deletion cases generally fall under the same broad categories as described in Cormen et al. [1], but with several alterations that take into account the sizes of nodes and keys involved. It is assumed that the key k to be deleted is already present in *B^-MD-tree*.

1. If key k is present in a leaf node, delete it from the node.
2. If key k is present in an internal node, then
   (a) If the sum of node sizes of the left child w_{LC} and right child w_{RC} surrounding the key k, combined with size of the key is less than or equal to B, combine the left and right child, splicing the key in between the left and right child in that order. The parent node looses address to right child and the key. Recursively delete key k from the combined node.
   (b) If the size of left child node w_{LC} is larger than the size of the right child node w_{RC}, find the predecessor k' of the key k in the tree rooted at the left child w_{LC}. Replace k with k' in the current node and recursively delete k' from the tree rooted at the left child w_{LC}. Symmetrically, if the size of right child node w_{RC} is larger than the size of the left child node w_{LC}, find the successor k' of the key k in the tree rooted at the right child w_{RC}. Replace k with k' in the current node and recursively delete k' from the tree rooted at the right child w_{RC}.
3. If k is not present in the internal node w but is present in the subtree rooted at one of its child node called the destination node w_{DS}, then
   (a) In the current node w, if there exist both left sibling node w_{LS} and the right sibling node w_{RS} surrounding the destination node w_{DS}, pick the smallest of the two, and call it just the sibling

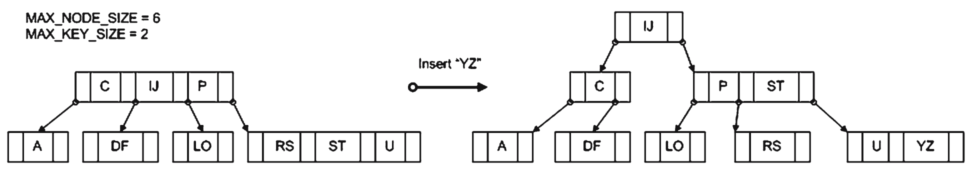

Fig. 1. Insertion showing node split from leaf node to root

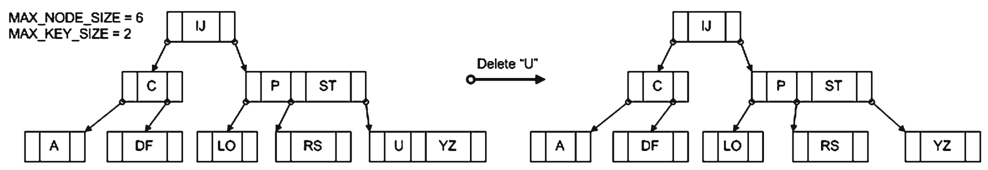

Fig. 2. Deletion case *i* in a leaf node

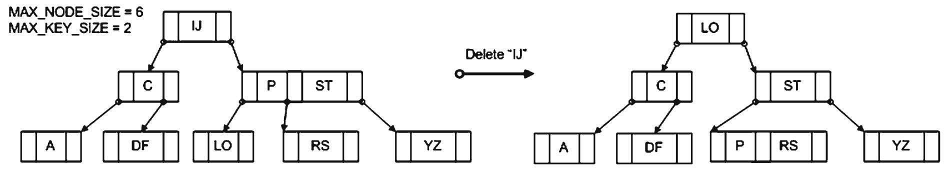

Fig. 3. Deletion case *iib* in internal node. "IJ" is replaced by its successor "LO"

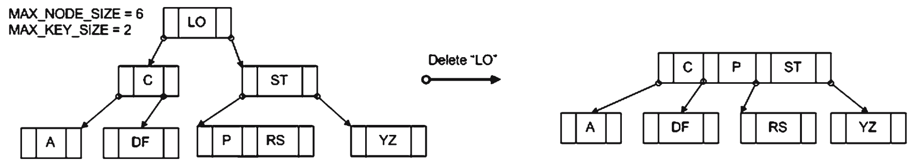

Fig. 4. Deletion case *iia* in internal node. "LO" is first spliced in between two combined child nodes and then deleted

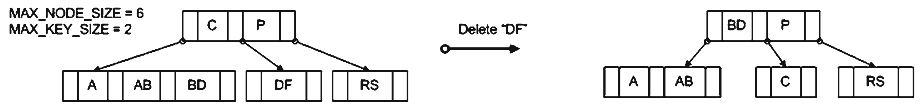

Fig. 5. Deletion case *iiib*. "BD" is shuffled up in the parent node and "C" is shuffled down in child node

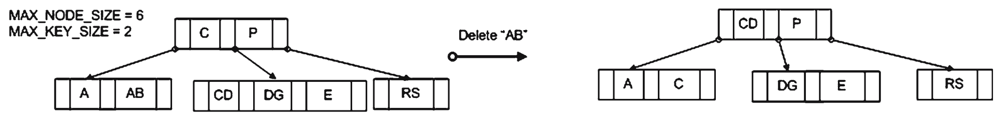

Fig. 6. Another deletion case *iiib*. "CD" is shuffled up in the parent node and "C" is shuffled down in child node

Figs. 1–6 Examples of insertion and deletion cases

node w_{SB}. If sum of the size of the destination node w_{DS}, and its sibling node, w_{SB}, combined with the size of key k' that sits in parent node w, between destination and sibling node, is less than or equal to B, combine w_{DS}, w_{SB}, and k', into w_{DS}, freeing w_{SB}. Recursively delete k from subtree rooted at w_{DS}.

(b) In the current node w, if there exist both left sibling node w_{LS} and the right sibling node w_{RS}, surrounding the destination node w_{DS}, pick the largest of the two, and call it just the sibling node w_{SB}. If the w_{DS} node size is smaller than w_{SB} or if w_{DS} is a leaf node with a single key, then shuffle condition is *TRUE*. If the shuffle condition is TRUE, see if keys can be shuffled between w_{DS}, w_{SB} and w. Shuffling involves moving a key down from parent node w into w_{DS} and a key up from w_{SB} to w. If the shuffle condition is TRUE and shuffling is not possible due to size constraints and w_{DS} is a leaf node with a single key, do a forced shuffle, causing a split of node w.

In cases 3a and 3b, there can be a possible scenario where destination node w has only one sibling. This happens when the destination node is one of the two extreme child nodes, i.e., the leftmost or the rightmost one.

The check for shuffle in case 3b guarantees that space exists in the node where the key is moved. For example, when a key is moved down from the parent node w to child node w_{DS}, the child node must have space to accommodate the parent key.

The forced shuffle in case 3b prevents creating empty leaf nodes. If forced shuffle is not applied, leaf nodes can become empty during deletion, even if key sizes are bounded (see Sect. 4). Forced shuffle is not applied to internal nodes, because leaf nodes stay non-empty during deletion, provided the key sizes are bounded. This is a corollary of prompt replacement of a deleted internal node key by its successor or predecessor from leaf node (case 2b).

Lines 20 and 21 in routine B$^-$MD-TREE-DELETE implement case 1. Lines 27–30 implement case 2a. Routine B-TREE-COMBINE-NODES takes one of the input parameters as level l of parent node w. Depending on whether the tree height is reduced or not, it returns the level of next node where deletion will progress. In addition, routine B-TREE-COMBINE-NODES returns the combined node. Lines 32–42 implement case 2b. Notice calls to B$^-$MD-TREE-INSERT-AT-LEVEL at lines 35 and 39 where the deletion algorithm can ensue an upward pass. Lines 65–67, and 78–90 implement cases 3a and 3b, respectively. Routine B$^-$MD-TREE-FORCE-SHUFFLE-KEYS at line 86 is assumed to call B$^-$MD-TREE-INSERT-AT-LEVEL. This causes split of the parent node to accommodate shuffled key from child node and prevents creating empty leaf nodes. The routine FREE-NODE releases all the resources for a node and delinks it from the *B$^-$MD-tree* node set. In one possible implementation of FREE-NODE, the freed up node can be added to a linked list pool, which can later be reused in inserts.

Figure 2 through Fig. 7 show various cases of *B$^-$MD-tree* deletion. Figure 7 shows the forced shuffle scenario of case 3b.

4 Optimal maximum key size and locating median key

Even though a *B$^-$MD-tree* key size can be up to B, it is seen that having a preset maximum value of key size improves the *storage utilization* of a persisted *B$^-$MD-tree*. Storage utilization is efficient, if all nodes are at least half full, i.e., occupied size in each node is at least $\frac{B}{2}$.

Better storage utilization results when key sizes are smaller. For certain large key sizes ($>\frac{B}{3}$), there can be scenarios where some of the nodes end up being empty with no keys. This is shown in Fig. 8, as a new key is inserted in *B$^-$MD-tree*. Location of median key in node split during insertion can affect the storage utilization. The median key can be based on the size or the number of items currently occupied. For example, a median key based on size would be such that it is divides the occupied length of the node in two *approximately* equal sized partitions.

There should be at least three keys in a node so that the two nodes resulting from the split have at least one key when the median key is moved out. This sets the *optimal* value of maximum key size, K_{max} to $\frac{B}{3}$. But, even after having K_{max} preset to the optimal value, choosing a median key merely based on size can still create empty nodes during inserts and may not always result in the best storage utilization. This can be noted in scenarios presented in Fig. 9 and Fig. 11, where some internal and leaf nodes are left empty after insertion. Therefore, having preset K_{max} to its optimal value, we first pick the median key based on size. If that results in first or last key being chosen as the median key, then we pick the median key based on the number of items occupied. This *mixed* technique of choosing the median key is exemplified in Fig. 10, prevents creating empty nodes and results in better storage utilization.

5 *B$^-$MD-tree* performance

The $\mathcal{O}$ and Θ complexity metrics for search, insertion and deletion are same as for a B-tree.

5.1 Maximum height h

Assume all keys have the same size. Let's say B is the fixed node size, K be the key size and N be the total number of keys. Then, root node at level 0 has $\frac{B}{K}$ keys. Level 1 has $(\frac{B}{K})(\frac{B}{K}+1)$ keys. Level 2 has $(\frac{B}{K})(\frac{B}{K}+1)^2$ keys and so on. Hence,

$$N = \frac{B}{K} + \frac{B}{K}\sum_{i=1}^{h}\left(\frac{B}{K}+1\right)^{i}$$

$$N = \left(\frac{B}{K}+1\right)^{h+1} - 1, \quad h \geq 0,$$

$$h = \log_{\frac{B}{K}+1}(N+1) - 1.$$

If node size B is very large as compared to key size K, we have h as $\mathcal{O}(\log_{\frac{B}{K}} N)$. In practical scenarios keys have varying sizes and K can be deemed as size of the largest key.

For a given N and B, the maximum height h will be achieved when all nodes have least possible keys, i.e., key sizes are equal to optimal maximum key size $\frac{B}{3}$.

Figs. 7–10 Examples of insertion and deletion cases (continued)

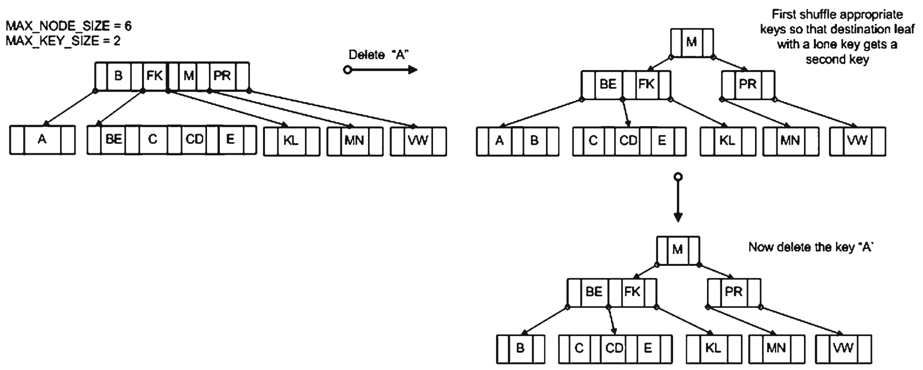
MAX_NODE_SIZE = 6
MAX_KEY_SIZE = 2
Delete "A"
First shuffle appropriate keys so that destination leaf with a lone key gets a second key
Now delete the key "A"

Fig. 7. Deletion special case *iiib*, where single key left in a leaf node is to be deleted and shuffling key up from child node into the parent requires a split of parent node. This necessitates a forced shuffle to prevent creating an empty leaf node

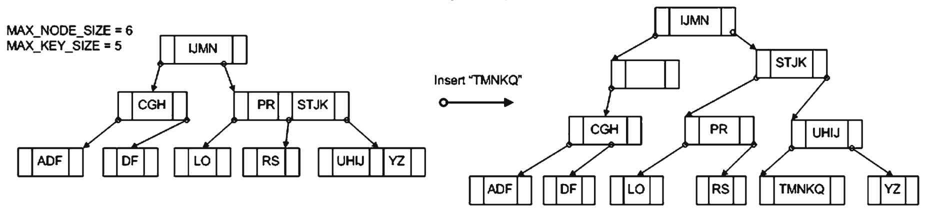

Fig. 8. Inserting a very large key can result in empty internal node

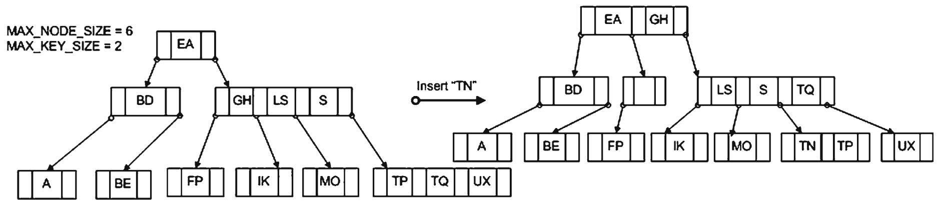

Fig. 9. Median key is chosen such that its size and the sum of sizes of keys preceding it is greater than or equal to half of occupied size in the node. This criteria lends to first key "GH" being chosen as median key, resulting in empty internal nodes

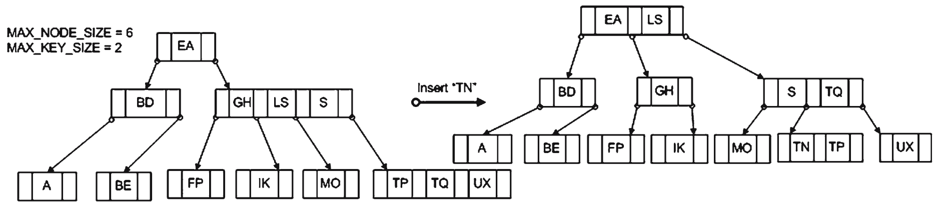

Fig. 10. Choosing median key based on both size and number of occupied keys

Fig. 11 Examples of insertion and deletion cases (end)

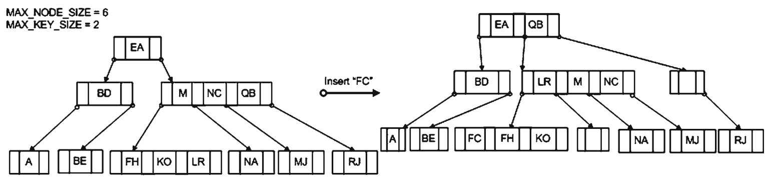

Fig. 11. Median key is chosen such that the sum of sizes of keys preceding it is greater than or equal to half of occupied size in the node. This criteria lends to last keys "LR" and "QB" being chosen as median keys, resulting in empty nodes

Therefore,

$$h \le \log_{3+1}(N+1) - 1,$$

$$h \le \frac{1}{2}\log_2(N+1) - 1,$$

where, $h \ge 0$, $B \ge 3$ and $1 \le K \le \frac{S}{3}$.

Note that when $K = \frac{B}{3}$, maximum height h varies as $\mathcal{O}(\log N)$ and is independent of B and K.

5.2 Search performance

DISK-READ requires $\mathcal{O}(1)$ time and there cannot be no more h disk reads in search. Hence, there cannot be no more than $\mathcal{O}(h)$ disk reads. The CPU usage depends on the number of key comparisons in the while loop at line 5 in B⁻MD-TREE-SEARCH. The number of comparisons are bounded by number of keys, which cannot be more than B, assuming 1 byte of storage for the smallest key. Hence, the CPU performance is, $\mathcal{O}(B \times h)$ or $\mathcal{O}(B \times \log_{\frac{B}{K}} N)$.

5.3 Insert performance

The insert routine can back up from leaf to the root in the worst case. Therefore, number of disk reads and CPU time of key comparisons in a *B^-MD-tree* can be twice as compared to a B-tree. Overall, there are $\mathcal{O}(h)$ DISK-READs and $\mathcal{O}(h)$ DISK-WRITEs. The CPU time in key comparisons is $\mathcal{O}(B \times \log_{\frac{B}{K}} N)$.

5.4 Delete performance

The delete routine can back up from an internal node to the root two times in the worst case. One back up scenario is possible in case 2b and the other in case 3b (see Sect. 3.3). Therefore, number of disk reads, writes and CPU time in key comparisons in a *B^-MD-tree* can be three times more as compared to a B-tree. Overall, there are $\mathcal{O}(h)$ DISK-READs, $\mathcal{O}(h)$ DISK-WRITEs and CPU time in key comparisons is $\mathcal{O}(B \times \log_{\frac{B}{K}} N)$.

6 Conclusion

B^-MD-tree has been found to be very useful in persistent storage of keys, which exhibit a wide variation in size. For example, consider an implementation of *B^-MD-tree*, where B is set to 4,000 bytes and optimal K_{max} is set to 1,300 bytes ($\frac{B}{3}$) allowing key sizes from 1 to 1,300 bytes. Applications that need to query on a large collection of persistent data items whose sizes are in such a range, find *B^-MD-tree* very useful. For example, anti-spyware and anti-virus application have large variation of signature sizes and would be a perfect fit for this implementation.

Despite feasible maintenance and better storage utilization, it is to be noted that it is very likely to have some fragmentation in every node because the sum of occupied key sizes may not be exactly same as the fixed node size B.

The backing up in inserts and deletes in the worst case can be addressed by using caching techniques that prevent frequent disk I/O. Also, the CPU usage in key comparisons within a node can be improved by replacing linear search with the *binary* search.

Appendix

A *B^-MD-tree* additional algorithms

Table 3 Algorithm A.3—non-recursive B⁻MD-TREE-INSERT-AT-LEVEL

```
B⁻MD-TREE-INSERT-AT-LEVEL(T, x, k, c, a, l)
1   w ← x
2   while forever
3   do
4       i ← n[w]
5       if (sizeof(w) + sizeof(k)) ≤ B
6       then
7           while i ≥ 1 and k < key_i[w]
8           do
9               key_{i+1}[w] ← key_i[w]
10              c_{i+2}[w] ← c_{i+1}[w]
11              i ← i – 1
12          key_{i+1}[w] ← k
13          c_{i+2}[w] ← c
14          n[w] ← n[w] + 1
15          DISK-WRITE(w)
16          break
17      else
18          (w, k, c, l) ← B⁻MD-TREE-SPLIT-NODE(T, w, k, c, a, l)
```

Table 4 Algorithm A.4—B⁻MD-TREE-SPLIT-NODE

```
B⁻MD-TREE-SPLIT-NODE(T, x, k, c, a, l)
1   j = LOCATE-MEDIAN-POS(x)
2   z = ALLOCATE-NODE()
3   leaf(z) = leaf(x)
4   for i ← j + 1 to n[x]
5       key_{i-j}[z] ← key_i[x]
6   n[z] = n[x] – j
7   if not leaf(x)
8   then
9       for i ← j + 1 to n[x] + 1
10          c_{i-j}[z] ← c_i[x]
11  k' ← key_j[x]
12  n[x] ← j – 1
13  if k > key_1[z]
14  then
15      y ← z
16  else
17      y ← x
18  i ← n[y]
19  while i ≥ 1 and k < key_i[y]
20  do
21      key_{i+1}[y] ← key_i[y]
22      c_{i+2}[y] ← c_{i+1}[y]
23      i ← i – 1
24  key_{i+1}[y] ← k
25  c_{i+2}[y] ← c
26  n[y] ← n[y] + 1
27  DISK-WRITE(x)
28  DISK-WRITE(z)
29  if l = 0
30  then
31      r ← ALLOCATE-NODE()
32      leaf[r] ← FALSE
33      n[r] ← 0
34      c_1[r] ← address(x)
35      root[T] ← r
36      x ← r
37  else
38      l ← l – 1
39      x ← DISK-READ(a[l])
40  k ← k'
41  c ← address(z)
42  return (x, k, c, l)
```

Table 5 Algorithm A.5—non-recursive B$^-$MD-TREE-DELETE(end)

```
B⁻MD-TREE-DELETE($T$, $k$)
    $w \leftarrow root[T]$
    $l \leftarrow 0$
    while forever
    do
        $i \leftarrow 1$
        $a[l] \leftarrow address(w)$
        while $i \leq n[w]$ and $k > key_i[w]$
        do
            $i \leftarrow i + 1$
        if $i \leq n[w]$ and $k = key_i[w]$
        then
            $a_{LC} \leftarrow c_i[w]$
            $a_{RC} \leftarrow c_{i+1}[w]$
            for $j \leftarrow i$ to $n[w] - 1$
            do
                $key_j[w] \leftarrow key_{j+1}[w]$
                $c_{j+1}[w] \leftarrow c_{j+2}[w]$
            if $leaf[w]$
            then
                DISK-WRITE($w$)
                break
            else
                $w_{LC} \leftarrow$ DISK-READ($a_{LC}$)
                $w_{RC} \leftarrow$ DISK-READ($a_{RC}$)
                if (sizeof($w_{LC}$) + sizeof($w_{RC}$) + $sizeof(k)$) $\leq B$
                then
                    ($w_{LC}$, $l$) $\leftarrow$ B-TREE-COMBINE-NODES($w_{LC}$, $w_{RC}$, $k$, $l$)
                    DISK-WRITE($w_{LC}$)
                    FREE-NODE($w_{RC}$)
                    $w \leftarrow w_{LC}$
                else
                    if sizeof($w_{LC}$) > sizeof($w_{RC}$)
                    then
                        $k' \leftarrow$ LOCATE-PREDECESSOR($a_{LC}$)
                        B⁻MD-TREE-INSERT-AT-LEVEL($T$, $w$, $k'$, $a_{RC}$, $a$, $l$)
                        $w \leftarrow w_{LC}$
                    else
                        $k' \leftarrow$ LOCATE-SUCCESSOR($a_{RC}$)
                        B⁻MD-TREE-INSERT-AT-LEVEL($T$, $w$, $k'$, $a_{RC}$, $a$, $l$)
                        $w \leftarrow w_{RC}$
                    $k \leftarrow k'$
                    $l \leftarrow l + 1$
        else
            $w_{DS} \leftarrow$ DISK-READ($c_i[w]$)
            if $i = 1$
            then
                $w_{LS} \leftarrow$ NIL
                $w_{RS} \leftarrow$ DISK-READ($c_{i+1}[w]$)
                $w_{SB} \leftarrow w_{RS}$
            else if $i = n[w] + 1$
            then
                $w_{LS} \leftarrow$ DISK-READ($c_{i-1}[w]$)
                $w_{RS} \leftarrow$ NIL
                $w_{SB} \leftarrow w_{LS}$
            else
                $w_{LS} \leftarrow$ DISK-READ($c_{i-1}[w]$)
                $w_{RS} \leftarrow$ DISK-READ($c_{i+1}[w]$)
                if sizeof($w_{LS}$) < sizeof($w_{RS}$)
                then
                    $w_{SB} \leftarrow w_{LS}$
                else
                    $w_{SB} \leftarrow w_{RS}$
            if (sizeof($w_{DS}$) + sizeof($w_{SB}$) + $sizeof(key_i[w])$) $\leq B$
            then
                ($w_{DS}$, $l$) $\leftarrow$ B⁻MD-TREE-COMBINE-NODES($w_{DS}$, $w_{SB}$, $key_i[w]$, $l$)
                DISK-WRITE($w_{DS}$)
                FREE-NODE($w_{SB}$)
            else
                if (not $w_{LS}$ = NIL) and (not $w_{RS}$ = NIL)
```

Table 5 continued

```
            then
                if sizeof($w_{LS}$) > sizeof($w_{RS}$)
                then
                    $w_{SB} \leftarrow w_{LS}$
                else
                    $w_{SB} \leftarrow w_{RS}$
            if ((sizeof($w_{SB}$) > sizeof($w_{DS}$)) or ($leaf[w_{DS}]$ and $n[w_{DS}] = 1$))
            then
                if $B^-$MD-TREE–CAN-SHUFFLE-KEYS($w_{SB}$, $w$, $w_{DS}$)
                then
                    $B^-$MD-TREE-SHUFFLE-KEYS($w_{SB}$, $w$, $w_{DS}$)
                    DISK-WRITE($w_{SB}$)
                    DISK-WRITE($w$)
                    DISK-WRITE($w_{DS}$)
                else if $leaf[w_{DS}]$ and $n[w_{DS}] = 1$
                then
                    $B^-$MD-TREE-FORCE-SHUFFLE-KEYS($w_{SB}$, $w$, $w_{DS}$)
                    DISK-WRITE($w_{SB}$)
                    DISK-WRITE($w$)
                    DISK-WRITE($w_{DS}$)
            $l \leftarrow l + 1$
        $w \leftarrow w_{DS}$
```

References

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L.: Introduction to Algorithms. MIT Press, Massachusetts (1990)
2. Bayer, R., Unterauer, K.: Prefix B-trees. ACM Trans. Database Syst. **2**(1), 11–26 (1977)
3. Ferragina, P., Grossi, R.: The string B-tree: a new data structure for string search in external memory and its applications. J. ACM **46**(2), 236–280 (1999)
4. Comer, D.: Ubiquitous B-Tree. ACM Comput. Surv. (CSUR). Archive **11**(2), 121–137 (1979)
5. Korth, H.F., Silberschatz, A.: Database System Concepts. McGraw-Hill Inc., New York (1991)