Implementing Deletion in B^+-Trees

Jan Jannink
Stanford University
Computer Science Dept.
Stanford, CA 94305
e-mail: jan@cs.stanford.edu

Abstract

This paper describes algorithms for key deletion in B^+-trees. There are published algorithms and pseudocode for searching and inserting keys, but deletion, due to its greater complexity and perceived lesser importance, is glossed over completely or left as an exercise to the reader. To remedy this situation, we provide a well documented flowchart, algorithm, and pseudo-code for deletion, their relation to search and insertion algorithms, and a reference to a freely available, complete B^+-tree library written in the C programming language.

1 Motivation

A first offering of a database system implementation course at Stanford University required students to implement indexes to their data files, either in the form of B^+-trees or using extendible hashing. The author, in his capacity as teaching assistant, advised students to search in the literature for pseudocode or descriptions to implement these algorithms.

This paper is motivated by the fact that not a single instance of the B^+-tree deletion algorithm in the form of pseudocode seems to exist in the literature, nor do any elegant implementations of the algorithm exist in the public domain. In fact, of four implementations examined, two in function libraries and two embedded in database programs, two used a weak form of deletion discussed below, another was lengthy and unsuitable for use as a pedagogical tool, and the last was made needlessly complicated by a non-recursive design.

Perhaps since tree structures can, for the most part, be described by a search and an insert method, few authors bother with the more intricate deletion algorithm, and assign it as an exercise to the reader. This omission goes as far back as [Knu73], and is repeated in virtually all of the literature since.

2 Background

B-trees, introduced in [BM72], are a general class of balanced multiway trees which serve as an indexing mechanism for structured data, and are geared in particular towards large paged files. Two classes of B-tree variants were recognized, B^+-trees and B^*-trees, which offer additional properties over the original model. In this lineage it is also worthwhile to cite 2-3 trees, devised in 1970 by Hopcroft [AHU83], since they are in fact the B-tree structure with the smallest fanout, i.e. number of pointers per node, and red-black trees [GS78], which effectively model B-trees by using one or two binary nodes to represent a single 2-3 node. An in depth analysis of most tree variants is found in [Woo93], which presents $2\text{-}3^+$ trees in its examples.

The seminal paper on B-trees [BM72] presents simple flowcharts for the functions to manipulate them, and [Knu73] also describes search and insert algorithms for them. [Com79] provides good general descriptions of B-trees and their variants, as well as relatively detailed descriptions of algorithms to perform search, insertion and deletion on B-trees, although in a dated programming style.

B^+-trees differ from B-trees in one aspect which makes them desirable for database systems, namely that no data resides in the interior nodes of the trees. Since all of the data is contained at the level of the leaves, the leaves can be linked together, allowing sequential access to the data once the leaves are reached. This also means that interior nodes contain only referential data, acting as a guide to the information kept at the leaves. As a result the algorithms for B-trees and their variants are not identical.

Unfortunately, the $^+$ and * notations are not universally accepted, and several good references leave them out. It appears that the B-trees discussed in [AHU83] as well as in [FZ92] and [GR93] are in fact B^+-trees and the algorithms described there could be implemented by filling in the details.

The cause of the deletion gap in the algorithmic record may stem from the fact that there is no single paper introducing the B^+-tree concept. Instead, the notion of maintaining all data in leaf nodes is repeatedly brought up as an interesting variant. As the importance of B^+-trees gained recognition in the database community, a number of textbooks geared towards databases have presented them. In [Sal88] B^+-tree algorithms are presented, though deletion is in fact incomplete and described as "quite a complicated" algorithm. Both [Liv90] and [FZ92] cover them as well, but omit deletion. [FZ92] does present a useful figure depicting the recursive approach to the insertion algorithm for B-trees, which can be applied to any of these tree structures. Finally, [EN94] contains non-recursive pseudocode for search and insertion in B^+-trees, but just an illustrative diagram for deletion.

The only complete deletion algorithms are found, for 2-3 trees in [Oli93] and for B-trees in [Wir76], both of which contain a wealth of pseudocode for many other algorithms. Red-black tree deletion code can found in [CLR90] or at the source [GS78].

3 Definitions

To firmly ground the discussion, we begin by reviewing the definition of the B^+-tree structure and the invariants it must obey. Also, refer to Figure 2 at the end of the paper depicting three representative trees of height four.

B^+-tree

- is a structure of nodes linked by pointers
- is anchored by a special node called the root, and bounded by leaves
- has a unique path to each leaf, and all paths are equal length
- stores keys only at leaves, and stores reference values in other, internal, nodes
- guides key search, via the reference values, from the root to the leaves

node

- is either internal or a leaf, including the root node
- contains at most n entries and one extra pointer for some fixed n
- has no fewer than $\lfloor n/2 \rfloor$ entries, the root excepted

root node

- is a leaf when it is the only node in the tree and will then contain at least one entry
- must have at least 2 pointers to other nodes when it is internal

internal node

- contains entries consisting of a reference value and a pointer towards the leaves
- its entries point to data classified as greater than or equal to the corresponding reference value
- its extra pointer references data classified as less than the node's smallest reference value

leaf node

- contains entries consisting of a key value and a pointer to the storage location of data matching the key
- its extra pointer references the next leaf node in the tree ordering; leaves linked in this manner are neighbors

In all B-tree type structures, key search proceeds from the root downwards, following pointers to the nodes which contain the appropriate range of keys, as indicated by the reference values. Likewise, all B-trees grow from the leaves up. After obtaining the appropriate location for the new entry, it is inserted. If the node becomes overfull it splits in half and a pointer to the new half is returned for insertion in the parent node, which if full will in turn split, and so on.

B^+-trees distinguish internal and leaf nodes, keeping data only at the leaves, whereas ordinary B-trees would also store keys in the interior. B^+-tree insertion, therefore, requires managing the interior node reference values in addition to simply finding a spot for the data, as in the simpler B-tree algorithm.

B^*-tree algorithms incorporate an insertion overflow mechanism to enforce higher node utilization levels. B^*-tree insertion at full nodes may avoid splitting by first checking neighboring nodes. Keys from the full node are redistributed to a less full neighbor. If both neighbors are full, however, the split must take place.

Deletion in B^+-trees, as in B^*-trees, is precisely the converse of B^*-tree insertion. If a node falls below its minimum number of entries after the deletion, its neighboring nodes are checked. If they have more than the minimum number of keys, a fraction of the surplus keys from the larger neighbor are redistributed to the node. Only if both neighbors are minimal in size are nodes merged together.

4 Lazy Deletion

There has been some research on the acceptability of relaxing the constraint of minimum node size to reduce the number of so-called unsafe tree operations, i.e., those which contain node splitting and merging [ZH89].

The debate has culminated in analysis of a weaker form of the deletion algorithm which we call *lazy deletion*, that imposes no constraints on the number of entries left in the nodes, allowing them to empty completely before simply removing them. According to [GR93], most database system implementations of B^+-trees have adopted this approach. Its most effective use is when it is vital to allow concurrent access to the tree [JS93b], and excessive splitting and merging of nodes would restrict concurrency.

[JS89] derives some analytic solutions calculating memory utilization for B^+-trees under a mix of insertions and lazy deletions, based on previous research which considered insertions only [BY89]. The simulations in [JS89] support its analysis to show that in typical situations, where deletions don't outnumber insertions in the mix of operations, the tree nodes will contain acceptable percentages of entries.

One of the work's assumptions [JS93a] is that the keys and tree operations are chosen uniformly from a random distribution. This assumption is unreasonable in certain realistic situations such as one described below. Allowing interior nodes with only a single pointer to exist in a B^+-tree creates the possibility for arbitrarily unbalanced trees of any height, which are virtually empty, and in which access times have degenerated from the logarithmic bound B^+-trees are meant to guarantee to a worst case unbounded access time. Since nodes are not removed until they are completely empty, the lazy deletion algorithm does not regulate tree height effectively.

5 Example

In our example, the keys on which data are inserted increase monotonically, such as a time stamp, and old data is deleted soon after insertion. Mr. Hapless of Half-Baked pastry shop keeps information about orders as they come in, summarizes them every day, and deletes all but the summaries at the beginning of every month.

In an actual B^+-tree, this activity can correspond to the following operations. First, fill a node until it splits, then delete all but one entry in the node containing the smaller keys. Likewise, every time an internal node splits, delete all but one of the keys pointed to by the node referencing the smaller keys. Since the tree is growing in one direction, the deletions of smaller keys don't change the rate of growth of the tree, but they do make it virtually empty.

In such a scenario, the resulting tree contains n paths from the root, $n-1$ of which are of some fixed height, let's say h, each with exactly one key at the leaf. The nth path from the root leads to a subtree of height $h-1$ with the same structure, as shown in figure 2(c) at the end of the paper.

Interestingly, the insertion algorithm must accept this tree as full, that is, ready to acquire a new root at the next appropriate insertion, even though it contains only $h*(n-1)$ keys, less than the expected *minimum* $(\lceil n/2 \rceil)^h$ keys for a tree of this height. More surprisingly, this structure can be pared down to a single path of length h simply by deleting all but one of the tree entries, so that only a single key remains.

Admittedly, the worst case is unlikely, but since plausible scenarios for its occurrence exist, a complete and correct deletion algorithm is preferable.

6 Algorithm with Flowchart

6.1 Deletion

Before presenting pseudocode we provide a basic flowchart and algorithm to illuminate its function. Figure 1 shows how the initial downwards recursive search is followed by an upwards unwinding of the recursion, during which the deletion, and potentially the rebalancing of the tree, takes place. The second phase corresponds to the shaded area of the figure. A set of *immediate neighbors* and *anchors*, defined below, is calculated during the search phase, for use during the tree rebalancing. The algorithm outline is as follows:

1. recurse to a leaf node from root to find deletable entry: for nodes in the search path, calculate immediate neighbors and their anchors
2. *if* entry found at leaf node **continue** *else* **stop**
3. remove appropriate entry from current node
4. *if* there is underflow **continue** *else* **done**
5. *if* current node isn't root, **continue** *else* collapse root: make its only child into the new root so tree height decreases, **done**
6. check number of entries in immediate neighbors
7. *if* both are minimal sized **continue** *else* balance current node: shift over half of a neighbor's surplus keys, adjust anchor, **done**
8. merge with a neighbor whose anchor is the current node's parent, unwind to parent node and **continue** at 3.

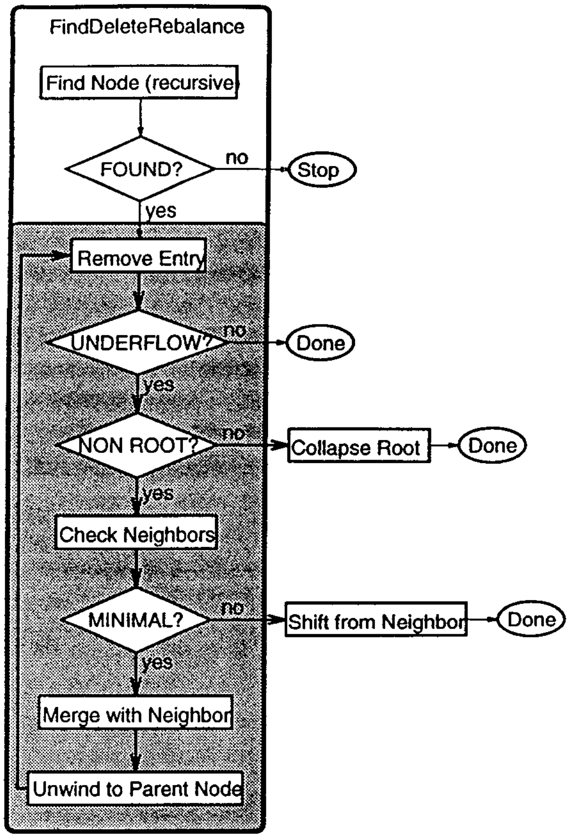

Figure 1: Recursive B^+-tree deletion flowchart

To describe the management of reference values in internal nodes some further definitions are useful. The *parent* of a node is the node immediately preceding it in its search path, thus an *ancestor* is any node in the path to a node. An *immediate neighbor* of a node is a node at the same tree level containing values consecutive to those of the node. The ancestor node at which two other nodes' search paths diverge is called their *anchor*. A single reference value in an anchor determines whether a search continues towards some node or its immediate neighbor. If values shift between nodes after key deletion, the *anchor value* described above must also be updated. Furthermore, when a merge or shift takes place between two internal nodes, their anchor value must also shift to the node receiving entries, in order to maintain correct tree structure. Figure 2(b) shows the anchors of a node's left and right neighbors.

6.2 Notes on Search and Insertion

Key search consists of a recursive descent to the leaves, without any action as the recursion unwinds. The first two steps of the deletion algorithm correspond to a search. Insertion replaces the underflow test with an overflow test, the node merging block with a node splitting block, and the collapse root block with a new root block, while leaving out the minimal neighbor test. B*-tree insertion includes a test for maximal neighbors to determine if overflow rebalancing is possible.

7 Pseudocode Implementation

The pseudocode below is procedural in style, based on a C library implemented by the author. Single line comments are C++ style.

The key is assumed to be of some type keyT, and the node variables are of a node pointer type Nptr.

```
delete(key)
begin
  balanceNode = NO_BALANCE
  root = findRebalance(rootNode, NO_NODE, NO_NODE,
                                  NO_NODE, NO_NODE, key)
end


findRebalance(thisNode, leftNode, rightNode,
                                  lAnchor, rAnchor, key)
begin
  var removeNode, nextNode, nextLeft, nextRight,
                                  nextAncL, nextAncR

// PART 1: recursive descent from root to leaf node

        // find the nodes needing rebalancing
  if thisNode is not minimal sized
      balanceNode = NO_BALANCE
  else if balanceNode == NO_BALANCE
      balanceNode = currentNode

        // node location best matching key value
  nextNode = entry pointer for key

  if thisNode is not a leaf         // continue search

        // calculate neighbor & anchor nodes
    if nextNode is least entry in thisNode
      nextLeft = greatest entry pointer of leftNode
      nextAncL = lAnchor
    else
      nextLeft = preceding entry pointer
      nextAncL = thisNode
    if nextNode is greatest entry in thisNode
      nextRight = least entry pointer of rightNode
      nextAncR = rAnchor
    else
      nextRight = following entry pointer
      nextAncR = thisNode

        // recursive call
    removeNode =
      findRebalance(nextNode, nextLeft, nextRight,
                                nextAncL, nextAncR, key)

  else          // key was found or not
    if entry pointer for key exists
      removeNode = nextNode
```

```
    else
      removeNode = NO_NODE

// PART 2: delete key, unwind recursion, rebalance tree

        // remove entry from current node
  if removeNode == nextNode
    clear removeNode entry in thisNode
    free removeNode memory

        // check which rebalancing actions are needed
  if balanceNode == NO_BALANCE
    done = NO_NODE
  else if thisNode is root
    done = collapseRoot(thisNode)
  else
    done = rebalance(thisNode, leftNode, rightNode,
                                    lAnchor, rAnchor)

  return done
end


collapseRoot(oldRoot)
begin
  if oldRoot is leaf
    newRoot = NO_NODE          // tree is empty
  else
    newRoot = entry pointer to root's sole child
  free oldRoot memory

  return newRoot
end


rebalance(thisNode, leftNode, rightNode, lAnchor, rAnchor)
begin
        // find a neighbor & anchor for rebalancing
  balanceNode = more full of {leftNode, rightNode}

        // select shift or merge operation
  if size(balanceNode) is not minimal
    anchorNode = balanceNode anchor in {lAnchor, rAnchor}
    done = shift(thisNode, balanceNode, anchorNode)
  else
        // at least one anchor is thisNode's parent
    anchorNode = thisNode parent in {lAnchor, rAnchor}
    mergeNode = anchorNode child in {leftNode, rightNode}
    done = merge(thisNode, mergeNode, anchorNode)

  return done
end


shift(thisNode, neighborNode, anchorNode)
begin
        // reference value separates the nodes' data
  if thisNode is an internal node
    copy anchorNode separator value to thisNode

        // equalize the nodes' sizes
  repeat
    shift neighborNode entries to thisNodo
  until size(neighborNode) == size(thisNode)

        // new reference value reflects shifted data
  copy new separator value to anchorNode
        // no more nodes need removal
  balanceNode = NO_BALANCE

  return NO_NODE
end


merge(thisNode, neighborNode, anchorNode)
begin
        // reference value separates the nodes' data
  if thisNode is an internal node
    copy anchorNode separator value to neighborNode

        // empty one of the two nodes
  repeat
    shift thisNode entries to neighborNode
  until size(thisNode) == 0

        // adjust node pointer value in leaf node
  if thisNode is leaf
    set thisNode's extra pointer to be neighborNode's

        // set empty node up for later removal
  return thisNode
end
```

8 Conclusion

We hope that the the information presented here, while hardly revolutionary, fills an unexpected gap. Our purpose has been to show that a straightforward implementation of B$^+$-tree deletion fits well in a common framework with search and insertion methods, and that its correctness is vital to maintain the tree invariants. A complete, commented and fully parametrized library of B$^+$-tree algorithms in the C programming language is available by anonymous ftp from `db.stanford.edu` in the directory `~ftp/pub/jannink/btree/`, or over the web at `http://www-db.stanford.edu:80/pub/jannink/btree/`.

9 Acknowledgements

Professor Jennifer Widom's database implementation course is the primary source of inspiration for this work. Furthermore, professor Widom's encouragement and suggestions improved its quality immeasurably.

References

[AHU83] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison–Wesley, Reading MA, 1983.

[BM72] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[BY89] R. A. Baeza-Yates. Expected behaviour of B$^+$-trees under random insertions. *Acta Informatica*, 26(5):439–471, 1989.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge MA, 1990.

[Com79] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[EN94] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Benjamin / Cummings, Redwood City CA, second edition, 1994.

[FZ92] M. J. Folk and B. Zoellick. *File Structures*. Addison–Wesley, Reading MA, second edition, 1992.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kauffman, 1993.

[GS78] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21. IEEE, 1978.

[JS89] T. Johnson and D. Shasha. Utilization of B-trees with inserts, deletes and modifies. In *Proceedings of the 8th Symposium on Principles of Database Systems*, pages 235–246. ACM, 1989.

[JS93a] T. Johnson and D. Shasha. B-trees with inserts and deletes: Why free-at-empty is better than merge-at-half. *Journal of Computer and System Sciences*, 47(1):45–76, 1993.

[JS93b] T. Johnson and D. Shasha. The performance of current B-tree algorithms. *ACM Transactions on Database Systems*, 18(1):51–101, 1993.

[Knu73] D. E. Knuth. *The Art of Computer Programming: Volume III*. Addison–Wesley, Reading MA, 1973.

[Liv90] P. Lividas. *File Structures: Theory and Practice*. Prentice Hall, Englewood Cliffs NJ, 1990.

[Oli93] I. Oliver. *Programming Classics: Implementing the World's Best Algorithms*. Prentice Hall, Englewood Cliffs NJ, 1993.

[Sal88] B. J. Salzberg. *File Structures: an Analytic Approach*. Prentice Hall, Englewood Cliffs NJ, 1988.

[Wir76] N. Wirth. *Algorithms + Data Structures = Programs*. Prentice Hall, Englewood Cliffs NJ, 1976.

[Woo93] D. Wood. *Data Structures, Algorithms, and Performance*. Addison–Wesley, Reading MA, 1993.

[ZH89] B. Zhang and M. Hsu. Unsafe operations in B-trees. *Acta Informatica*, 26(5):421–438, 1989.

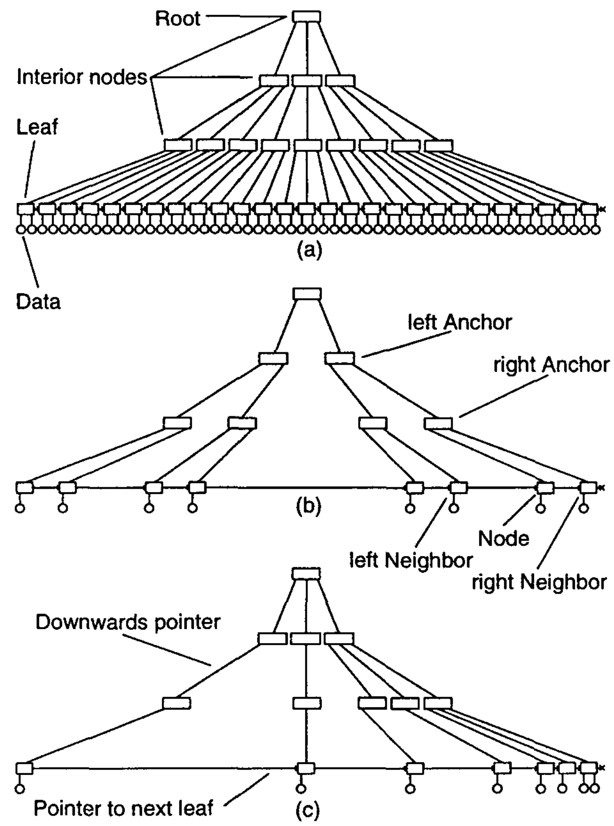

Figure 2: height 4 trees (a) maximum (b) minimum (c) a 'maximal' tree under lazy deletion

A Appendix: 2-3$^+$-trees

Figure 2(a) depicts the maximum 2-3$^+$-trees of height 4, which references 54 keys. Any insertion into this tree will cause a new root to be created. Figure 2(b) is the minimum 2-3$^+$-tree of height 4. Any deletion from it would cause a sequence of merges culminating in the collapse of the root, resulting in a height 3 tree referencing 7 keys.

Trees for which the insertion of some key increases its height are called maximal. Normally, a smallest maximal 2-3$^+$-tree of height 3 references 8 keys, and would appear as Figure 2(b) plus one node after the 9th key is inserted and the new root added. The tree in Figure 2(c), pared down through lazy deletion is still maximal. An insert to its rightmost node will cause it to require a new root, even though the resulting height 5 tree will reference only 9 keys.