**B+ Trees – Storage Management and Indexing**

Submitted by: Team 15 - Deepak Thipeswamy (1001235195)

Under Supervision of: Prof. Sharma Chakravarthy and GTA, Mr. Jay Bodra

Department of Computer Science,

University of Texas at Arlington

**Acknowledgements**

My sincere thanks to Prof. Sharma Chakravarthy who gave us brilliant lectures on Storage Management and Indexing before the project and throughout and without his guidance, implementing the DELETE algorithm for B+ Trees would have been a daunting task.

I’m also grateful to the GTA, Mr. Jay Bodra for solving all our queries as soon as possible on Blackboard and in person. A wonderful presentation on Minibase which made things crystal clear and made me think in terms of the algorithm keeping Minibase implementation in mind.

Deepak Thipeswamy (1001235195)

**Overview**

A **B+ tree** is an n-ary tree with a variable but often large number of children per node. A B+ tree consists of a root, internal nodes and leaves. The root may be either a leaf or a node with two or more children.

A B+ tree can be viewed as a [B-tree](https://en.wikipedia.org/wiki/B-tree) in which each node contains only keys (not key-value pairs), and to which an additional level is added at the bottom with linked leaves.

The primary value of a B+ tree is in storing data for efficient retrieval in a [block oriented](https://en.wikipedia.org/wiki/Block_(data_storage)) storage context — in particular, [filesystems](https://en.wikipedia.org/wiki/Filesystems). This is primarily because unlike [binary search trees](https://en.wikipedia.org/wiki/Binary_search_tree), B+ trees have very high fan-out (number of pointers to child nodes in a node, typically on the order of 100 or more), which reduces the number of I/O operations required to find an element in the tree.

**Objective**

In this project, Minibase is used in place of traditional databases for academic purposes. We had to implement a B+ tree in which leaf level pages contain entries of the form (Alternative 2 for data entries, in terms of the textbook on page). We had to implement the full deletion algorithms with the tree balanced at the end of the delete operation. We had to implement delete i.e. remove the record and perform any merging or redistribution as required as per minimum occupancy of 50% in each Leaf and Index Page.

**Methodology**

The algorithm used for the DELETE functionality is as per the prescribed book. Using recursion to traverse through the tree and balance it using merge and redistribute as and when a DELETE operation is performed.

**Overall Status**

The DELETE algorithm is implemented **completely** in the method skeleton provided.

The major components of the implementation and their implementation details are as follows:

1. **Search**

The tree has to be traversed from top to bottom to find the key that needs to be deleted. Initially, I made use of the INSERT’s part of code which is already doing the search to find the place where it needs to insert. Upon getting the logic of SEARCH in place, I continued with the implementation of the algorithm for the DELETE.

*BTIndexPage.getPageNoByKey(key)* methodis used for finding the subtree which might contain the key being deleted.

1. **Delete**

This part of the code is the main part of this project which makes sure that the requested key will be deleted from the tree. The important part of this routine is that, the tree needs to be balanced at the end of every delete operation performed on the tree as this a B+ tree. The balancing of the tree can be performed by using either of the two prominent techniques namely **Redistribute** and **Merging**.

The following are the main parts of the algorithm/code that is implemented to accomplish a full-fledged DELETE operation on a B+ tree.

1. **Recursion**

As per the algorithm, the code implemented uses recursion for the traversal of the tree and also to maintain occupancy states of the B+ tree. *\_Delete* is designed a recursive function which will recurs from the header page through the Index subtrees till it finds a Leaf page holding the key to be deleted.

Ex: // choose a subtree until the data is found in a leaf page

PageId nextPageId = curIndexPage.getPageNoByKey(key);

// unpin the current page

unpinPage(curIndexPageId);

//recurse and get the key to be deleted after merge

upEntry = \_Delete(key, rid, nextPageId, curIndexPageId);

* *curIndexPageId* is the page id of the current index page
* *nextPageId* is the page id of the subtree to be traversed.
* *upEntry* is the variable will be returned from the subroutines (subtrees) containing the new index key in case a balancing of tree has to be performed.

1. **Redistribute**

Redistribution scenario occurs when a key is deleted from one of the leaf pages and that page underflows in terms of occupancy of the size of the page (50% of the total page size). Same scenario can occur for the Index pages also.

*BTIndexPage.getSibling()* is used to find a sibling page through the parent, the current page should try and redistribute with, in case of an underflow. The getSibling() method return the page no of the chosen sibling and direction at which it is found (-1 = Left, 1 = Right and 0 = No eligible sibling found)

*BTIndexPage.redistribute()* is used to perform the redistribution of the keys between the current page and the chosen sibling ensuring that both the pages are at minimum occupancy after redistribution.

1. **Merge**

Merge scenario occurs when the redistribution of the keys was unsuccessful. According to the algorithm, we have to try to redistribute first to avoid propagation of change in structure across many levels of the tree. There are scenarios where the redistribute might not always be possible. In such situations, the chosen direction and the sibling page is used to check where the current page and the chosen sibling page can be merged by ensuring that the combined sizes of both the pages does not cross the limit of the maximum space of any given page of that type. At the end of merge, the empty page will be freed and the sibling pointers are updated as applies.

*BTTreeFile.mergeIndexPages()* is the method I have implemented to perform the concrete merging operation of the index pages which is a little different from the merges of the leaf page i.e., *BTTreeFile.mergeLeafPages()*.

The major difference between the merge of leaf pages when compared with the merge of the index pages is that, the merge of the index pages pulls down a key from the parent as part of the page being merged but in the leaf page, the value in the parent page is replaced. So the space for the extra key that is pulled down in an index merge needs to be considered. As the subroutines of the recursion returns to the caller, the adjusting of the keys are taken care of.

1. **Boundary Conditions**

As important as the Merge/Redistribute scenarios are, the boundary conditions were the tricky parts where, if there is a single leaf page, there is no need of an index page as the traversal is straight forward. The index page would anyway be empty so that page needs to be freed and the pointer from the header page needs to redirect to the leaf page. Upon delete of all keys, all index and leaf pages should be freed leaving an empty tree.

Ex: **if** (headerPage.get\_rootId().pid == curIndexPage.getCurPage().pid) {

// if no records in it, free page

**if** (curIndexPage.numberOfRecords() > 0) {

// records found, return

unpinPage(curIndexPage.getCurPage(),**true**);

**return** **null**;

} **else** {

// no records, free the empty page and update header

BTSortedPage sortedIndexPage = **new** BTSortedPage(

curIndexPage.getPrevPage(),

headerPage.get\_keyType());

unpinPage(sortedIndexPage.getCurPage());

updateHeader(curIndexPage.getPrevPage());

freePage(curIndexPageId);

**return** **null**;

}

}

**File Descriptions**

**Major Functions**:

1. *mergeOrRedistributeLeafPages() and mergeOrRedistributeIndexPages ()* concerns with the logic where the occupancy is below minimum of 50% and the process of choosing a sibling and trying to redistribute of merge is taken care of in this function.
2. *mergeLeafPages() and mergeIndexPages()* concerns with the logic where the redistribution was not successful and the occupancy of the sibling pages are verified. Here the actual merging takes place.

**Test Cases**

1. Delete individual keys.
2. Delete a range of keys.
3. Delete all the values in the tree.
4. Delete all the values except the boundary keys (first key and last key)

**Additional Test Cases**

1. Delete key ranges other than the actual range of the tree.
2. Delete from an empty tree.
3. Delete duplicates.
4. Insert negative values and delete negative value by absolute delete.
5. Insert range of negative values and positive values and use the range delete.
6. Delete from lower the negative most value and positive values.
7. Insert 0 and delete using absolute and range delete.

All the above test cases are tested and verified against the code.

**Division of Labor**

I have completed the project by myself as I’m a one resource team. The algorithm is straight forward but the implementation in a Minibase environment took a lot of thinking in different angles which was the challenging part in this project. To be honest, I started late thinking that it is straight forward, hence I had to put in at least 7 hours of quality coding time for almost a week. Hence the total number of hours spent in implementation consumed:

**45hrs – 50 hrs**.

**Logical Errors**

1. During the implementation of the major parts, I used to encounter PAGE\_PINNED and PAGE\_UNPINNED exception. Meaning, I had lost the track of how many pages were pinned and how many were edited, how many were pinned more than once, how many were being freed without being pinned.

The way I handled them is by dividing the major chunks of code into smaller parts so that I would know the logical flow in an organized manner from a bird’s eye view. Hence the introduction of 5-6 new methods which perform concrete tasks divided functionally.

My 2 cents: We can have a data structure which holds the information about all the pinned pages so when a page needs to be freed or unpinned, the custom method would unpin the pages internally and free them without having to go through exceptions. But, this can cause a serious problem of concurrency also.

1. Adjusting of the links from sibling pages after a page is being freed was also one of the logical error I faced. The final tree would print only a small subtree from top because the links to the lower levels are ambiguous.

The way I handled it is simple, and in 3 steps.

1. Check if the page is the left most, if yes, then link the right sibling to INVALID (-1)
2. Check if the page is the right most, if yes, then link the right sibling to INVALID (-1)
3. Check if the page is neither of the above, then do a two way linking.
4. Check if the current page is the only page left, then update both links to INVALID.
5. Merge of index pages fails because I did not consider the space required for the pull down key from the parent and the size for the Slot and PageId which take int = 4 each.

I handled it by calculating the merge evaluation by considering the space for one extra key. Hence this was also resolved.

1. Data Inconsistency: I noticed that the keys that were previously deleted were again seen during the next cycle.

This was a straightforward situation where I had to keep in mind all the pages I was currently editing and also unpin the pages by setting the dirty flag to TRUE before exiting any given recursion level.

**Conclusion**

B+ tree project made me realized that implementing algorithms officially defined for any particular problem encountered is probably the fastest way to completion of the task. At first, I tried to build upon the code of *NaiveDelete* which was a fool’s errand. After a frustrating 2 days of coding I decided to follow the algorithm to the word and also use *\_insert()* code for the SEARCH operations and basic skeleton of the Recursion to be implemented.

The code is neatly formatted and modularized for good readability. This project was one of the most action oriented tasks I have gone through since my 5 years of industry experience in Development. Hoping for more challenging projects in the course.