Project 1: B+Tree Implementation

Team: Abhiram Vempati, Sankarshana Harish Rao

# Overall Status

In this project, we have implemented B+ tree insert(), \_insert() and Naivedelete() functions in Java. Both insert() and \_insert() functions are used in combination to insert into the tree and Naivedelete() function is used for deleting elements from the tree.

We have used Visual Studio code for implementing and testing the functions. Later, the same implementation/code was tested on Omega server. We have used ‘Pulse Secure’ to connect to Omega server and “File Zilla” to upload files to Omega server.

# Methods in detail

insert() method will used to when we want to insert first record and create a new first leaf page and updating header when split has occurred in an index page.

\_insert() will called when we want to insert a leaf node into a tree and update index and leaf pages in case of split.

## Insert()

Insert method will take key and record Id as input and insert it at correct place in the Tree.

This method is a public method exposed as an interface to delete records from the BTree.

The function algorithm is divided into two parts. The first part will handle insertion when tree is empty, and the second part will handle insertions when tree is not empty. If the insertion results in page splits and formation of new pages, the insert methods includes it in the root page and maintains it in the header page of the BTree class. The algorithm is as follows.

Initially when tree is empty, the following operations are done.

1. Create a new leaf page.
2. Update its new Next and Previous pointers as it is the only leaf page.
3. Insert the record onto new leaf page.
4. Unpin the new leaf page, as data is written.
5. Update the header to point to the new leaf page.

When Tree is not empty, the following operations are done.

1. First, a call to \_insert() is made so that the record can inserted into the tree.
2. Based on the returned value from \_insert(), we determine if a split has occurred or not.
3. if no split has occurred, then the function terminates here
4. if split has occurred, then we create a new index page and insert the returned value into it and update it so that it will point to index page, which previously Header page was pointing. Also, update the header page so that it will now point to new Index page.

## \_insert()

\_insert method is a private method used internally by the BTree class. The insert method delegates index traversal and and insertions resulting in splits to the \_insert function. If a split occurs the \_insert method returns the key to be inserted the upper level.

1. Check if the Page sent using PageId is Leaf page or Index page.
2. If it is an index page then,
3. \_insert() function is called recursively until a leaf page is reached.
4. Once a leaf page is reached, the <key,Rid> is inserted into the leaf page and a value is returned.
5. If the value returned is Null, then return Null.(as leaf page did not split)
6. If the value returned is not Null, then check if space is available in the current index page for inserting the returned key.
7. If there is space for inserting, then insert the returned key.
8. If there is no space, then a new index page needs to be created.
   1. Transfer all the records from the current Index page to new index page.
   2. Transfer few records from the new index page to current index page so that both new and current index page has equal number of keys.
   3. If the current index page has more entries than new index page, then transfer one element back to new index page.
   4. Now determine where the upEntry key has to be placed using keycompare function.
   5. Compare key in upentry and first key in new index page. If the keycompare()>=0, then insert the upentry key into new index page otherwise in the current index page.
9. If the page is a leaf, then -
10. Check if space is available in leaf page for the key.
11. If space is available, then insert the new key and return Null.
12. If space is not available, then the leaf page needs to split to accommodate the new key/record.
    1. Create a new leaf page and adjust its previous and next pointers.
    2. Transfer all the records from current leaf page to new leaf page.
    3. Then, transfer few records from new leaf page to current leaf page…so that the number of records in both the leaf is same.
    4. Undo one entry
    5. Now, decide where the new key is to be inserted using keycompare() function.
    6. If keycompare()>=0, then the key goes to new leaf page otherwise it goes to current leaf page.
    7. Copy the first key in the new leaf page into upentry and return the upentry value.

## NaiveDelete()

1. Find the leaf page where the key exists using the leafPage = findRunStart(key, currentRid).
2. If leaf page is not found, return null.
3. Else get record from the leafPage
4. While looping through the leaf pages indefinitely –
   * 1. Loop though the pages until reaching the page containing the record
     2. If end of pages is reached, return false for unsuccessful delete
5. If currentRecord’s key is greater than the key to delete, then the page has been found.
6. Delete the key from the leafPage.
   * 1. If delete from page was successful. Unpin the page –since it is dirty.
     2. Return true for successful delete.

# Logical Errors

1. Leaf page data cannot be cast into Index page data – In the \_insert() function whenever a new leaf page was created, we were returning the key and leaf data after inserting a new record into new leaf page. This caused an error in insert() function as insert() was supposed to receive the key and page number of the new leaf page so that it can create a new index page and update header. Because of this error, tree structure was not updating properly.
2. Using Null instead of INVALID\_PAGE for checking if a page is present or not while updating the Next and Previous pointers.