**CSE DBMS 5331 Project – 1 Report**

**Team No. <ENTER TEAM NO.>**

**Hemakumar Gokulakannan(1001415255)**

**Overall Status**

The Project has been completed and all the methods that were to be implemented have been completed. Methods **insert()**, \_**insert()** & **NaiveDelete()** have been coded after looking at the pseudo code provided in **Project 1 Demo\_v1.pdf**. Following are briefs about how we implemented the methods in this project.

1. **insert()**

In the method we first check if the header is pointing to an INVALID\_PAGE, this means that there are no nodes (Index or Leaf) attached to the header, in other words if the Tree is empty, we insert a leaf page with the key received from the calling method. Once we come to the second iteration we check if there is already a leaf existing and we invoke \_insert().

**NOTE:** According to our observation a leaf, for this project, is considered full if it has 62 Keys in it and an Index page is considered full if it has 83 Keys in it. A key in Leaf page takes 16 units of space and in an index it takes 12 units of space.

1. **\_insert():**
   1. **Header Points to a Leaf and it has space.**

Once \_insert() is invoked we check if the header is pointing to a leaf node, if yes, we keep inserting all the keys until the leaf is full.

* 1. **Header Points to a Leaf and it does not have space.**

Once we insert an entry that results in a split we create an index page and push the middle value on the BTSorted Leaf page to the index page. For the entries smaller than the middle key, the one we pushed to the newly created index page, we create a new leaf and insert all the keys smaller than index in it. Once done we make the previous page pointer of the index key point to the leaf with all the keys smaller than the index. For keys greater than or equal to the Index key we create a new leaf and make the index key’s next page pointer point to the leaf containing all the keys greater than the index keys.

* 1. **Tree which results in a Leaf Split and then an Index Split.**

In a case where an index results in a split follow the same pattern but here we push the middle value to a new index page, we also create two more index pages that have keys smaller than and greater than and equal to the middle value. We set the previous and next pointer of the index page accordingly. We keep doing this process recursively until we reach a condition where there is no split. In worst case this split would happen from the leaf till the root node.

1. **NaiveDelete():**

This method deletes a particular Key entry from a leaf page. For this we are using **findRunStart(key,rid)** which gives us a leaf page where the deletion has to be made. To delete the key we use the **deleteSortedRecord(rid)** which deletes that particular key from the leaf page. If the Key is not found the function **findRunStart(key,rid)** returns NULL and we throw a custom exception which says “Key not found!!”.

**File Description**

No new files were created for this project.

**Logical Errors and How they were handled**

According to us following were the logical errors we found in the code.

1. **findRunStart(key,rid):** When we delete a key using NaiveDelete(), after deletion if we try to delete the same key the method, **findRunStart(key,rid)**, returns the leaf page of where the key existed. Ideally, it should return NULL in such a scenario.
2. During an Index split deleting a record and then fetching the next one is not feasible as there is no getCurrent() method for the BTIndexPage class. Hence, we had to first store the values in an array data structure and then delete in a separate iteration.
3. Since, a key entry in a leaf page takes 16 units of space and an index takes 12 units of space we could accommodate 62 & 83 key entries in leaf and index page respectively there were 8 units and 4 units of space which were always be unused(or wasted) in Leaf and Index page respectively.
4. When we insert a key into an Index page there are two UpEntry that need to be taken care. The first one is the one which is returned by the Index page and the second one is the one that the Index page receives by its Child. This lead to incorrect values being inserted in the tree, but we handled this by using two separate variables for this – UpEntry & Idx\_UpEntry.