**The Implementation of B+ Trees**

**Overview:**

In this project, we were asked to implement B+ Trees using Minibase. We were provided with the skeleton code of Java, a B tree test file and we were supposed to implement the following functions in order to implement a fully functional B+ Trees: -

1. ***insert()***:

In the ‘Insert()’ function, we first checked whether the values to be inserted is an Integer or not. We then checked whether the headerpage is pointing to an INVALID\_PAGE or not. If it is pointing, this means that our tree is empty and we need to create a new leaf page. Upon creating, we must update the headerpage to the new node that we created. If the header page is not pointing to the Invalid page, we call the ‘\_insert()’ method and pass the key value. The ‘\_Insert()’ function returns a value which tells whether the root node is splitted or not. If there was a split, we must create a new Index page and update the headerpage.

1. ***\_insert():***

The ‘\_insert()’ method is called by its parent ‘Insert()’ method. In this method we will first check whether the page is an index page or a leaf page. If it is an index page, we check for available space. If we have enough available space, we insert the key. If we don’t have enough available space, we first create an index page and then unpin it. This is followed by calling the ‘\_insert()’ method recursively. In Case of split we split the page equally, insert the key and redistribute suitably. We will simply push up the key in case of an Index page. If it is a leaf page, we insert the key and split if required. In case of split, we will copy up the middle key and set the pointers accordingly. We then return the copied value to the calling parent function.

1. ***NaiveDelete():***

In this method, we first checked whether the tree is empty or not. If empty, it will return false and we can not delete. If not, We must create a new Leafpage to store the value returned by ‘findrunstart()’ method which gives us the leftmost leaf page. We then traverse all the nodes and look for the key value and simply delete it. While traversing, we use the ‘getNext()’ method to get the pageId of next page along with pinning the next page and unpinning the previous page.

In this Project, we were also provided with a BTtest.java file in which there were 6 Test cases that were given and our goal was to ensure the proper functioning of all the test cases. All our methods are working properly and all the test cases that we were supposed to do are perfectly executing.

**File Descriptions:**

There were no new files added. We just needed to implement the above functions in BTreeFile.java and our goal was to implement those functions and execute all the test cases that were there in the BTTest.java.

**Division of Labour:**

Most of the Implementation in this project is done by both of us. We first tried to understand the basic schema of the B+ Tree algorithm. We studied the java doc for the project and the Minibase functions in our first week.

We then hardcoded the algorithm on paper and used few integer data samples to check if our algorithm is working or not. After that, we started to code in the BTreeFile.java where we encountered numerous difficulties. To get the final code running, we almost utilized 25-30 hours.

The total project took approximately 40-45 hours.

**Logical errors**

We encountered a lot of logical errors in our code. Some of them are as follows: -

**Logical error 1: Pinning the Page**

This error I encountered while coding our ‘insert()’ function. I recursively called the ‘\_insert()’ function without pinning the page that I am passing to the ‘\_insert()’ function.

I handled this error by reading the reference book which stated that we must pin every page before modifying it. And when I pinned the page before calling the ‘\_insert()’ function that error was solved.

**Logical error 2: Constant page size and constant data entry size**

I tried to split the node in two cases, that is, when the node consists of even records and when it consists of the odd records.

But later according to the Minibase, a Leaf Node can have only 62 data entries. This means that we only had to code for the split of the even number of data entries in a full leaf node. The complexity of the code reduced to half, since we didn’t had to code for the odd number of data entries in the leaf node.

**Logical error 3: Splitting the page into two equal halves**

1. In our first algorithm, we used a dummy leaf page called ‘swapLeafPage’, to split the data entries into two halves. But then we faced a lot of reference issues. Sometimes they were still pointing to the dummy page, sometimes to their own page.

We solved this problem using the same two pages, old leaf page and new leaf page, instead of using a 3rd swap leaf page.

1. This became easy as I realized that we had to split only even number of entries. I just used a for loop to count all the entries in the node. Using this count variable, I processed the for loop for ‘count/2’ times to split the records in two equal halves.
2. There was also a case when after inserting the key value, there was a split and the old leaf node had 1 extra element than the new leaf node. So, one of the challenge was to copy the last data entry from the old leaf node to the new leaf node.

For this we used a dummy data entry in the for loop which always had the latest inserted entry value in the old leaf page. So, at the end of the for loop, we could use the last data entry that was inserted in the old leaf page with this dummy variable.

**Logical error 4: The pages got pinned too many times**

We spent approximately 4-5 hours on correcting this logical error. Before calling the ‘\_insert()’ function we pinned the page once and even at the start of the ‘\_insert()’ we pinned the same page.

We took a sample data set and used our algorithm on the paper to insert the elements. We drew the recursive chart and found out that we were pinning the page two times and unpinning it only once.

**Logical error 5: Traversing the leaf page in which the key is there**

Initially, we were not able to understand what actually the ‘findRunStart()’ method is doing. We were confused whether it is giving the leftmost page or the page in which our key is there which was quite challenging and made our code wrong.

In order to overcome this problem, we first understood the whole ‘findRunStart()’ method and get to the concept of it that, it is returning the leftmost leaf page and not the page in which our key was there.