Group 31:
Yuhan Xie
Jiahui Zhou
Wei Shi

PP3: B+ Tree Index Design Report

- ## Implementation choices for different functionality:
  - ### Constructor:

    This is the constructor for the whole tree implementation. The primary thing it should do is to set up the status variables of the tree. But to start the tree, we are facing two choices: "graft" a tree that already exists or "seed" a brand new tree. So, we first try to find the existing target index file. If it exists, we set the header and root page. Otherwise, we create a new index file and set the necessary meta information for it. Additionally, the root will be treated as leaf node when we initialize a new tree.

  - ### Tree Construction:

    For inserting nodes, we implemented multiple auxiliary functions. We have two node allocation functions **allocNonLeaf()** and **allocLeaf()**, which allocate pages in bufMgr for nodes and initialize the page for use.

    The main function **insertEntry()** handles insertions with two cases: root is leaf and root is not leaf. When the root is leaf, **insertToLeaf()** is called, we handle possible splits with **splitLeafNode()**, and push up midKey as the only entry of the new root node. When the root is not leaf, **insertToNode()** is called on the root node. The **insertToNode()** function handles the recursive search, insert, split and push up process. The split of leaf node and nonLeaf node are handled separately in **splitLeafNode()** and **splitNonLeafNode()**. Both functions will pass back the midKey to be pushed up and the pageId for the newly splitted node. The outer recursion then handles the insertion of the pushed up key to the current node with **insertToLeaf()** and **insertToNonLeaf()**.

    For time complexity, every insertion in a specific node would cost at most *INTARRAYLEAFSIZE* or *INTARRAYNONLEAFSIZE* time, as we need to traverse the array to find the position for new entry and move all succeeding entries

backward. Finding the node takes $O(2d\log_d N)$ time, as in the worst scenario we need to traverse the entry arrays of each level, and we will have $\log_d$ levels. No repeated traversal required, we always recursively access the node that is either the parent of the target leaf node or the target leaf node.

For page pinning, we only keep one page pinned in every function other than a split function. Only the page where the currently used node is located is being pinned. The page will be unpinned as soon as another function is called. In the split functions, only the pages for the current node and the newly splitted node are being pinned. Therefore, only up to 2 pages are pinned during the whole tree construction process.

- ## Scan

During the Scanning step, we first use the **startScan()** function to begin a filter scan of the index. Within the given range of index, we start from the root to find out the leaf page that contains the first RecordID that satisfies the scan parameters. Here we use a recursive function **recursiveScanPageId()** to recursively find the Page ID of the first element that is larger or equal to the given lower bound. Inside the recursive function, we implemented two other helper functions as an auxiliary tool to improve the efficiency, which are also used in the insert parts. The first helper function is **isLeaf()**, which we pass the page node in and return a boolean value that determines whether it's a leaf node. The second helper function is **findNonleafIndex()**, in which we traverse through the key array to find the index of the key we need. After we find the leaf node, we continuously search for the data entries that are in the right range by linearly fetching the next page that is linked to the original leaf node and stop once we find the target.

For time complexity, **recursiveScanPageId()** only accesses the parent of the target leaf node and the leaf node. No unnecessary traverse used. The time complexity of **startScan()** is $O(2d\log_d N)$. The time complexity of **scanNext()** is $O(1)$ as we keep track of the current page and entry index, each step forward only requires an advance of the index or to the next sibling.

In **startScan(),** pages are unpinned right before a recursive call of **recursiveScanPageId()**. During the scan, **scanNext()** keeps only the page with page number of *currentPageNum* pinned. Before the ending of a scan, the current page is unpinned. Therefore, during the whole scan process, there is only one page pinned.

• Test Case:

- Test 4: This is testing for the functionality of an empty tree. By creating a relation size of 0 and doing scanning operations, we are able to check if the value after operation matches the expected value. One thing to mention is that the relation is created in forward order.
- Test 5: This is testing for a very large tree. The goal for this test is to test the functionality of splitting the non leaf node. Based on the calculation in btree.h, *INTARRAYLEAFSIZE* is 682. After our calculation, we found that only when tree size is larger than (341*681+682 = 232903) a split in a non leaf node will take place. So we set up the tree size to 300000. By creating a relation size of 300000 and doing scanning operations, we can check to see if the value after operations matches the expected values.