Design Report

1. Additional function implemented:

insert into();

```
/**
     * This function will search a position for a new entry.
     * If current page is a non-leaf page, it will find a position and call
insertEntryRecursive.
     * If current page is leaf page, it will try to insert new entry.
     * @param ridKeyPair            The new entry that need to insert.
     * @param pageId                Current page number.
     * @param isLeaf                Whether current leaf is leaf page.
     * @param LEAFARRAYMAX          Length of leaf page array.
     * @param NONLEAFARRAYMAX       Length of non-leaf page array.
     * @param newValue              Need value pushed up by the child.
     * @param newPage                       Page number of the new child page.
    **/
```

reArrangeRoot();

```
/**
     * This function is called when root node got split.
     * We need to create a new root page and link it to the old root page and new
page.
          * @param newValue                Key value pushed up bi child.
          * @param newPageId               Page id of new child page.
          * @param ARRAYMAX                Length of array of non-leaf page.
    **/
```

leafSplitHelper();

```
/**
     * Helper function when a leaf node need to split.
     * @param pos                          Insert position.
     * @param last                 last position for this node.
     * @param LEAFARRAYMAX         Length of leaf page array.
     * @param NONLEAFARRAYMAX      Length of non-leaf page array.
     * @param ridKeyPair           The new entry that need to insert.
     * @param leafNode             Pointer to current page.
     * @param newPageId            Need value pushed up
     * @param newValue             Page number of the new page.
    **/
```

nonLeafSplitHelper();

```
/**
     * Helper function when a nonleaf node need to split.
     * @param pos                          Insert position.
     * @param NONLEAFARRAYMAX      Length of non-leaf page array.
```

```
     * @param nonLeafNode          Pointer to current page.
     * @param newPageId            Need value pushed up
     * @param newValue             Page number of the new page.
     * @param newChildValue        Need value pushed up by the child.
     * @param newChildPageId       Page number of the new child page.
    **/

startScanHelper();
/**
         * This function helps the startScan function.
          * T is the data type.
          * T1 is the non-leaf struct.
          * This function is called by the startScan and do the work.
          * @param lowValParm              Low value of the search range.
          * @param highValParm             High value of the search range.
          * @param ARRAYMAX                      Length of the non-leaf array.
         * @throws BadScanrangeException If lowVal > highval
         **/

scanNextHelper();
/**
           * Fetch the record id of the next index entry that matches the scan.
           * Return the next record from current page being scanned. If current page has been scanned to its entirety, move on to the right sibling of current page, if any exists, to start scanning that page. Make sure to unpin any pages that are no longer required.
           * @param outRid    RecordId of next record found that satisfies the scan criteria returned in this
           * @throws ScanNotInitializedException If no scan has been initialized.
           * @throws IndexScanCompletedException If no more records, satisfying the scan criteria, are left to be scanned.
          **/
```

2. How often do you keep pages pinned? How efficient is your implementation?
For insertion:
We will unpin the page whenever we copy the data to tree.
The insertion time complexity is linear, i.e. O(N), N is the number of nodes that need to be inserted.

For Scan:
We only pin one page at a time.
Our implementation for startScan(), scanNext() and endScan() is efficient. Complexity for startScan() is $O(\log_k N)$, where k is the number of child nodes per non-leaf node and N is the total number of nodes in B+ tree. There's no tree traversing in scanNext() and endScan() so each of these two has O(1) complexity.

3. How our design/implementation would change if you were to allow duplicate keys in the B+ Tree?
Solution 1:
Change the node to a array list, the value would be added to the same entry every time a duplicate key is inserted. When scan for the value, it should get the key value and go to the correspond entry, then search the list to get the value.
Solution 2:
Add a counter to the node. When a new key is inserted, then counter = 1. If a duplicate key is inserted, the counter will increment. Besides, the duplicate key would be added to the left of the previous key. When scan for the specific keyValuePair, searching could traverse down until counter ==1.

4. Test cases:
I added a test case to do a newIntTest in **main.cpp** . If testNum == 4, then it would start testing.

```
void test4() {

  //Test for new index Tests

  std::cout << "--------------------" << std::endl;
  std::cout << "createRelationRandom" << std::endl;
  createRelationForward();
  indexNewTests();
  deleteRelation();
  std::cout << "Test 7 passed" << std::endl;
}

void intNewTests()
{
  std::cout << "Create a B+ Tree index on the integer field" << std::endl;

  BTreeIndex index(relationName, intIndexName, bufMgr, offsetof(tuple,i),
INTEGER);

  // run some tests
  checkPassFail(intScan(&index,5000,GT,5500,LT), 0)

  checkPassFail(intScan(&index,4999,GTE,5500,LTE), 1)

  checkPassFail(intScan(&index,-1000,GT,-1,LT), 0)
}
```