**Design Report(Group 14)**

**Intro**

For our project 3 we implemented a B+ tree index to further the Buffer Manager program. Our B+ tree uses a search key of type integer to traverse the tree to find corresponding record id pairs. This allows us to efficiently travers the database, since it is self-balancing, in O(log n) for search, insert, and delete.

**Traversing the B+ tree**

The advantage of using a B+ tree is we can traverse trough the tree much like a linked list to find a page/key pair. The nodes contain a pointer to the next node and the key value. Using these nodes, we can traverse the tree in an ordered manner. Since the B+ tree is ordered by left child nodes being less than the parent, right child nodes being grater than the parent, and the middle in-between the two values, we can use this information to efficiently travers the tree.

**Design Choices**

We implemented the B+ tree by consistently calling InserEntry() to insert any new pages. From this point we determine where the entry should go based on the depth of the tree and the balance.

**Unpin Page**

We decided to unpin the pages as soon as possible. This allows us to keep the buffer pool from filling up. This allows us to speed up the program since the I/O would be a bottle neck, and we are now replacing that with CPU cycles, which are must faster.

**Efficiency**

Search: searching the tree will result in a I/O cost of 2\*height, since we need to read in and write out through the pages.

Insert: Inserting will be a simple cost of 2, since we are writing directly to the root node of the tree.

Scanning: Scanning the tree will have a worst-case scenario of O(log n) if we need to read every leaf node, or the same as Search.

Splitting: Splitting results in a I/O cost of 4. This is due to the need of reading in and writing both the current node and the new node we are creating in the split.

**Extra Tests**

**Test 4:**

We are testing negative values

**Test 5:**

Description