Setting Modification

For convenience, we add `keyNum` filed in `LeafNodeInt` and `NonLeafNodeInt` to store the keys in the leaf or non leaf node.
We also add `initialRootPageNum` in `BTreeIndex`. Because the root node is a leaf node at the very beginning but non leaf node after the first root split. We can use `rootPageNum == initialRootPageNum` to signal if root node is a leaf node.

Constructor & Destructor

```
BTreeIndex(const std::string & relationName, std::string & outIndexName,
           BufMgr *bufMgrIn, const int attrByteOffset,
           const Datatype attrType);
```

The constructor first tries to open `outIndexName` and read `IndexMetaInfo` from the first page. If the information in `IndexMetaInfo` matches with given ones, it will sets the `BTreeIndex`. If `FileNotFoundException` or `BadIndexInfoException` are caught in this process, a new `BlobFile` will be created and a new `IndexMetaInfo` struct will be initialized and wrote to header page. Finally, records in `relationName` will be inserted into B+ tree.

```
~BTreeIndex();
```

If `scanExecuting` is true, it will call `endScan()` first. Then it will flush the file and free the space of file in memory.

Insert

We designed insert as a recursive call that can search the right location for the given key in B+ tree level by level and insert it by country order.The insert function is divided as follows. As for the `unPinPage()`, the recursive insert call will keep the page in memory during deeper calls. The largest number of pages kept in memory is the depth of recursion. All other function will unpin all pages read or allocated in it.

```
template<class T>
T *allocNode(PageId &newPageId);
```

It allocates a page with a given type: LeafNodeInt, NonLeafNodeInt or IndexMetaInfo.

```
bool isLevelOneNode(const NonLeafNodeInt *page);
```

It checks whether the level of given non leaf node is 1, i.e., whether its children are leaves.

```
int findIndexInNonLeaf(const NonLeafNodeInt *nonLeafNode, const void * key);
```

It takes one non leaf node and a key as parameters and finds the location where the key should be inserted using `lower_bound()`.

```
int findIndexInLeaf(const LeafNodeInt * LeafNode, const void * key);
```

It takes one leaf node and a key as parameters and finds the location where the key should be inserted using `lower_bound()`.

```
void recursiveInsert(const PageId currPageNo, const void *key,
                     const RecordId rid, PageId &newPageNo, int & newIndex, bool isLeaf);
```

The core function of the insert. It controls the whole logic flow. `currPageNo` is the root page number of this call. `key` and `rid` is the record to insert. `newPageNo` and `newIndex` are used to store whether a split occurs in this level and the new page number and key. `isLeaf` is used to signal if `currPageNo` is a leaf node.

- First, it checks whether `currPageNo` is a leaf node, if so, call `handleLeafInsertion()` to handle this situation.
- Second, if it’s not a leaf node, we should find the page number of the page where the given record should be inserted and call `recursiveInsert()` to handle the insertion in the next level. We use the found page number as this call’s root.
- Then, assume the low level recursive call has finished. If we find that there is no split in lower level, then we don't need to modify this and above index and just `return`.
- If we find a split occurs in it's children and this non leaf node is not full, we need to insert the new page and new key into this non leaf node by calling `insertToNonLeaf()`.
- If we find a split occurs in it's children and this non leaf node is full, we need to allocate a new page the move half of entries into this new non leaf node. Then insert the new page and new key from children level into corresponding location.

Here, because the pointers in non leaf node are more 1 than the number of keys, we should adjust the structure of new non leaf node by moving the smallest key in it to parent level.

- Finally, due to occurrence of split, we need to set `newIndex` and `newPageNo` to tell parent level to insert new entry.

```
void handleLeafInsertion(const PageId currPageNo, const void *key, const RecordId rid,
                         PageId &newPageNo, int & newIndex);
```

To handle the end of the recursive insertion call. It inserts the `key` and `rid` into `currPageNo` and stores the split information in `newPageNo` and `newIndex`.
If the leaf node is not full, it can just insert record into it.
If the leaf node is full, it needs to allocate a new page, adjust the records between them and return split related information.

```
void splitLeaf(LeafNodeInt *node, LeafNodeInt *newNode, PageId newPageNo,
               const int leftLen);
```

It takes two leaf nodes as arguments and move `leftLen` records from `node` to `newNode` whose page number is `newPageNo`. Then, it adjusts the `rightSibPageNo` in them. `memcpy()` is used here for efficient memory operation.

```
void splitNonLeaf(NonLeafNodeInt *node, NonLeafNodeInt *newNode, const int leftLen);
```

It takes two non leaf nodes as arguments and move `leftLen` keys and `leftLen` pageNo(entries) from `node` to `newNode`. Then it adjusts the `newNode` 's level according to `node` 's. `memcpy()` is used here for efficient memory operation.

```
void splitRoot(const int key, const PageId left, const PageId right);
```

It is called by `insertEntry()` when a new root node needs to be generated. It allocates a new non leaf page and assign `key` to it. Then it sets the `left` and `right` as new root's two children. Then it sets the level of new root node. Then it modifies the meta information like `rootPageNum` and `rootPageNo` in first page(`IndexMetaInfo` object).

```
void insertToLeaf(LeafNodeInt *leafNode, const int index, const void * key,
                  const RecordId rid);
```

It inserts `key` and `rid` to location `index` in `leafNode`. `memmove` is used here for efficient memory movement with overlap. Then it adjusts the `keyNum`.

```
void insertToNonLeaf(NonLeafNodeInt * currNonLeafNode, const int index,
                     const int newIndex, const PageId newPageNo);
```

It inserts `newIndex` to location `index` and `newPageNo` to location `index+1` in `leafNode`. `memmove` is used here for efficient memory movement with overlap. Then is adjusts the `keyNum`.

```
void insertToNewNonLeaf(NonLeafNodeInt * currNonLeafNode, const int index,
                        const int newIndex, const PageId newPageNo);
```

After split of non leaf node, the right half has the same number of `key`s and `pageNo`'s.So the insertion into it is slightly different from normal non leaf node. It inserts `newIndex` to location `index` and `newPageNo` to location `index` in `leafNode`. `memmove` is used here for efficient memory movement with overlap. Then is adjusts the `keyNum`.

```
int deleteNewKeyNonLeaf(NonLeafNodeInt * currNonLeafNode);
```

After split and insertion of non leaf node, this function will be called to adjust the new non leaf node. Specifically, the smallest key in the new node will be move to parent level.

Scanner

During the process of scan, the maximum number of pages pinned is one. During the `startScan()`, it reads pages from the root to the leaf. Before it reads page at next level, page at current level is unpinned with clean state. After it finds the leaf page, it keeps the leaf page pinned. The leaf page is unpinned when the end of the page is reached. Then, the next page is read into the Buffer Manager. Upon throwing `IndexScanCompletedException()`, we leave the responsibility to unpin the current page to `endScan()` function. startScan takes time of `O(h)` where h is the height of the tree. The entire scan process takes linear time because we advance the entry one by one until we hit the upper bound. Details of design of each function are stated below:

```
startScan()
```

The `startScan()` function takes parameters, does sanity checks, and stores them into corresponding class variables. It then calls the private helper function `getLeafPage()` to get the corresponding leaf page number for the starting position of this scan. Then, it reads this leaf page and keeps this page pinned until we reach the end of this page or the upper bound is reached. Note that there is a possibility that the leaf page we got through the getLeafPage() helper function may not contain the desired starting entry due to the design that each partition is closed on the left, open on the right. For example, if our data looks like this: `{1, 2, 3} {6, 7, 8}`, and the search lower bound is set to `5`. The `getLeafPage()` will give us the page number of the first partition. This kind of situation will be handled in the scanNext() function, to either move to the next leaf page or throw `IndexScanCompletedException()`.

```
getLeafPage()
```

This helper function first calls the `getLevelOnePage()` function to get the PageId of the corresponding internal page with level of one. It then reads this internal page into bufMgr, gets the pageId for the leaf page, and unpins the internal page. This function keeps the internal page pinned during execution, and unpins it before return.

```
getLevelOnePage()
```

This helper function searches for the corresponding internal page recursively. On each recurse, it reads one page into Buffer Manager, reads data, and unpins it before calling the next recurse. We store the useful data onto stack, so that we can unpin the page before calling next resurse, but still able to use this data as the parameter for next recurse. This allows us to keep minimum possible pages pinned during the searching process.

```
scanNext():
```

This function is responsible to check if we have reached the end of the current leaf page. The edge case that is discussed in `startScan()` is also handled here. If it indicates that we have reached the end of the current page, it unpins the current page and reads the next page into

Buffer Manager and keeps it pinned. It then checks if the upper bound is reached, if so, it throws `IndexScanCompletedException()` without unpinning the current page. The `endScan()` function is responsible for unpinning the current page.

```
endScan():
```

It checks if `this->scanExecuting` is set to true. If not, simply throw an exception. Otherwise, it unpins the current page and sets all the class fields related to scanning procedure to default values.