Implementation choice I made:

The whole program uses BlobFile class to store all the information, including metadata page and also data page.

At first, a IndexMetaInfo object is created and stored in the metaData Page. Because this page is the first page that Buffer Manager allocated, the page number of metadata page is 1. Therefore, whenever we need to update the information, we can use bufMgr->readPage(file, 1, Page) to read the meta data page and modify it.

Metadata page also stores the root page number. Therefore, the root page can be retrieved from metadata page. The root page is a Leaf page at first. When data entry exceeds the limit, a splitLeafPage() is called to split root page, allocated right leaf page and new root page, and update the pointer.

Then a traverse function is implemented to insert recursively. Here is the logic: suppose we traverse to the Level 3(leaf page), insert the data and then return back to level 2, then I check whether level 3 is filled. If so, the leaf page will split and insert an entry in to level 2, and return to level 1. After return back from level 1, then I check whether level 2 is filled, if so, the non leaf page will split and an entry will be inserted into level 1… This is how I insert entry recursively.

How often keep the page pinned

When a page is being used, the pinCnt will remain nonzero (increment by 1). When operations are done for a page, the pinCnt will decrement to zero. Unless an entry is inserted in to the page, the dirty bit will remain false.

How efficient is my algorithm

I will first do an EMS merge sort, same as what bulk loading did. In this way, we can keep the data clustered and avoid modifying pointers. All the data can be inserted in order. Besides, the buffer memory is larger than data entry, so we don’t include extra I/O.

Extra design choice I made

In order to make it easier to track the size of the Leaf and non Leaf pages. I include another object to both the leaf and non leaf class:

```
struct NonLeafNodeInt{
  /**
   * current size of the node
   */
  int size;

  /**
   * Level of the node in the tree.
   */
  int level;

  /**
   * Stores keys.
   */
  int keyArray[ INTARRAYNONLEAFSIZE ];

  /**
   * Stores page numbers of child pages which them
   */
  PageId pageNoArray[ INTARRAYNONLEAFSIZE + 1 ];
```

In this way, when size is equal to INARRAYNONLEAFSIZE (current example), then I can know the node is full. Remind that don't forget to update the INARRAYNONLEAFSIZE:

```
//                                          size and level     extra pageNo                key       pageNo
const  int INTARRAYNONLEAFSIZE = ( Page::SIZE - 2*sizeof( int ) - sizeof( PageId ) ) / ( sizeof( int ) + sizeof( PageId ) );
```

Also I include a Boolean value in IndexMetaInfo class, which indicates whether the root page is a leaf page or a non-leaf page.

```
struct IndexMetaInfo{
  /**
   * whether it is a leaf node
   */
  bool isLeafPage;
```

If duplicate key and value

In my mind, there are two ways:

1. Use overflow pages. For every leaf page, add an extra bool variable: isDuplicate and an extra PageId: duplicatePageNo. If isDuplicate == true, it indicates that there is duplicate key, we can retrieve it through duplicatePageNo. However, if data is skewed, then the searching efficiency will be affected.
2. For each entry, mark the how many times it appears. Add an extra int variable: timeItAppears. If there is no duplicate key, set timeItAppears to 1. If there are duplicate keys, set timeItAppears to that number.