Project3 Report

1. Meta-data page in an index file
- Show your meta-data page of an index design if you have any.
When creating a file, we create a hidden page (Page #: 0) that stores readPageCounter, writePageCounter, and appendPageCounter. Every counter is an integer, which occupies 4KBs. Whenever a user writes/reads/appends a page, the corresponding counter + 1. The counters will be stored before the file is closed.

R: ixReadPageCounter
W: ixWritePageCounter
A: ixAppendPageCounter

Besides, the hidden page also stores general information about the B+ tree, that is, the root page number (int) and tree height (int). Both are used to facilitate search.

Root: Root page number
H: tree height

Actual pages (Root, Non-leaf, Leaf Nodes) start from Page 1.

```
 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
|__R__|__W__|__A__|__Root__|__H__|                     |

|               HIDDEN PAGE                            |

|______________________________________________________|
|                                                      |
|                                                      |
|        STORE RECORD FROM THIS PAGE                   |
|                                                      |
|______________________________________________________|
```

2. Index Entry Format
- Show your index entry design (structure).
In Root or Non-leaf nodes, an index entry is in the form of <key, Page #>, where Page # (short) points to the page containing keys ≥ the key in the pair.

```
 __________________________________________________________
|__Page#__|__Key__|__Page#__|__Key__|__Page#__|    …    |
```

In Leaf nodes, we use Alternative (3) for data entries. Specifically, a data entry is in the form of <key, # of RIDs = N, list of RIDs (RID_1, RID_2, ... , RID_N)>, where # of RIDs (short) indicates the number of RIDs which are associated with the key, and the list of RIDs (<Page Num, Slot Num>) is in the increasing order of the Page Num first and then in the increasing order of Slot Num.

```
 ___________________________________________________________
|__Key__|__# of RIDs __|__Rid#1__|__Rid#2__|    …    |__Rid#N__|
```

Both of them are illustrated in Question 4.

3. Page Format
- Show your internal-page (non-leaf node) design.
It looks like below:

```
Insert direction

 ————————>

 __________________________________________
|_Flag_|_Page#_|_Key1_|_Page#_|_Key2_|_Page#_|
|_Key3_|_Page#_|_Key4_|_Page#_|_Key5_|_Page#_|
|_Key6_|_Page#_|       …                     |
|                                            |
|          …                                 |
|                    …                       |
|                                            |
|     …     |_Page#N-1_|_KeyN_|_Page#N_|     |
|                                            |
|                                        ____|
|                                        |N|F|
 ————————————————————————-————————————————-
```

In the beginning of every non-leaf node, the "Flag" (occupies 2KBs) could be 0 (Root) or 1 (Non-Leaf node).

F: KBs that are used in this page (occupies 2KBs)
N: number of keys in this page (occupies 2KBs)

- Show your leaf-page (leaf node) design.
It looks like below:

```
Key, RID Insert direction
 ———————>
 __________________________________________________________
|_Flag_|_Next#_|_Key1_|_#ofRids_|_Rid#1_|_Rid#2_|_Rid#3_|_Rid#4_|
|_Key2_|_#ofRids_|_Rid#5_|_Rid#6_|_Key3_|_#ofRids_|_Rid#7_|     |
|                                                               |
|                   …                                           |
|                               …                               |
|                                           …                   |
|                                                               |
|           …  |_KeyN_|_#ofRids_|_Rid#i_|_Rid#j_|_Rid#k_|   …   |
|                                                               |
|                                                               |
|                                                               |
|                                                           ____|
|                                                          |N|F|
 __________________________________________________________
```

In the beginning of every leaf node, the “Flag” (occupies 2KBs) should be 2.

F: KBs that are used in this page (occupies 2KBs)
N: number of keys in this page (occupies 2KBs)
Next #: page number that points to the right sibling of this leaf node (occupies 2KB)

4. Implementation Detail
- Have you added your own source file (.cc or .h)?
No. We put our own unit test codes also in ix.cc and ix.h.

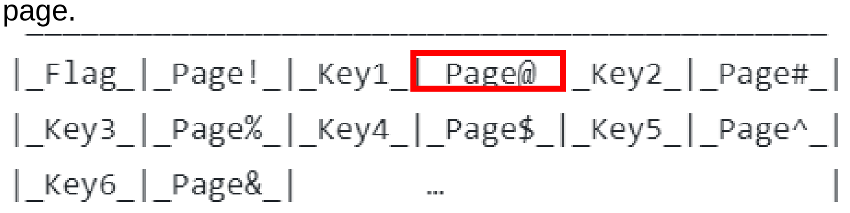
- Have you implemented non-lazy deletion? Choose Yes or No:
No.

- Have you implemented duplicated key handling that can span multiple pages? Choose Yes or No:
If yes, please explain briefly how you have implemented this feature.
No. If we detect duplicate keys in Non-Leaf Nodes, the program will throw an error message "Error: duplicate keys in the parent node" and terminates.

- Other implementation details:
Scan
a) set all properties that are needed for IX_ScanIterator by calling IndexManager::scan()

b) get root page: use the root page number stored in the hidden page to identify the root page

c) compare the key with the keys in this page. If the key satisfies key1 ≤ lowKey < key2, then we go to the page pointer to the left of key2. Take the picture below for instance, we have to go to the "Page@" since it satisfies the condition. Iterate this step until we reach the leaf node and get a leaf node page.


Flag	_Page!_	_Key1_	_Page@_	_Key2_	_Page#_
Key3	_Page%_	_Key4_	_Page$_	_Key5_	_Page^_
Key6	_Page&_	…

Note that if the key is smaller than all keys on the root page, then we go to the first page pointer on that page; if the key is larger than all keys on the root page, then we go to the last page pointer on that page. Similarly, we compare keys on the non-leaf pages and follow page pointers until we reach the leaf page.

d) read leaf page and store information in IX_ScanIterator (e.g., used byte, offset, matchedKey, matchedRid, siblingPointer, etc)

e) read a key and check if the key satisfies the lowKey condition
  i. if yes, check if the key satisfies the highKey condition, if yes, go to step f), if no, return -1
  ii. if no, read next key and iterate step e)

f) read next rid based on current offset and increase offset for next iteration. If all the rids in this key are returned (the next data will be a key), go to step e).

g) if we reach the end of this node page, move to the sibling page and start from d). However, if the sibling page pointer is -1, return IX_EOF, which indicates we are at the end of the file.

Insertion
Inserting an entry can be divided into three steps:

a) traverse from root to leaf node based on the given key

b) read keys in the leaf page. If the key to insert does not exist, insert both the key and the RID. If the key has existed, then find the appropriate position to insert only the RID and increment the #ofRids.

Say we want to insert Key2/Rid#5

Before Insertion:

```
 ____________________________________________________________
|_Flag_|_Next#_|_Key1_|_#ofRids_|_Rid#1_|_Rid#2_|_Rid#3_|_Rid#4_|
|_Key3_|_#ofRids_|_Rid#7_|                                      |
|                                                               |
|                  …                                            |
|                              …                                |
|                                                               |
|          …   |_KeyN_|_#ofRids_|_Rid#i_|_Rid#j_|_Rid#k_|   …   |
|                                                               |
|                                                           ____|
|                                                           |N|F|
 ____________________________________________________________
```

Because Key2 does not exist in the leaf page, we insert both the key and the RID.

After Insertion:

```
 ____________________________________________________________
|_Flag_|_Next#_|_Key1_|_#ofRids_|_Rid#1_|_Rid#2_|_Rid#3_|_Rid#4_|
| Key2 | #ofRids | Rid#5 |_Key3_|_#ofRids_|_Rid#7_|              |
|                                                               |
|                  …                                            |
|                              …                                |
|                                                               |
|          …   |_KeyN_|_#ofRids_|_Rid#i_|_Rid#j_|_Rid#k_|   …   |
|                                                               |
|                                                           ____|
|                                                           |N|F|
 ____________________________________________________________
```

If the key to insert already exists in the leaf page, we insert the RID only.

Let's use the "After Insertion" example above:

Say we want to insert Key2/Rid#6

Before Insertion:

```
 ____________________________________________________________
|_Flag_|_Next#_|_Key1_|_#ofRids_|_Rid#1_|_Rid#2_|_Rid#3_|_Rid#4_|
|_Key2_|_#ofRids_|_Rid#5_|_Key3_|_#ofRids_|_Rid#7_|            |
|                                                              |
|                  …                                           |
|                              …                               |
|                                                              |
|          …   |_KeyN_|_#ofRids_|_Rid#i_|_Rid#j_|_Rid#k_|   …  |
|                                                              |
|                                                          ____|
|                                                         |N|F|
 ____________________________________________________________
```

Because Key2 already exists, we insert Rid#6 only.

```
 ____________________________________________________________
|_Flag_|_Next#_|_Key1_|_#ofRids_|_Rid#1_|_Rid#2_|_Rid#3_|_Rid#4_|
|_Key2_|_#ofRids_|_Rid#5_|  Rid#6  _Key3_|_#ofRids_|_Rid#7_|     |
|                                                              |
|                  …                                           |
|                              …                               |
|                                                              |
|          …   |_KeyN_|_#ofRids_|_Rid#i_|_Rid#j_|_Rid#k_|   …  |
|                                                              |
|                                                          ____|
|                                                         |N|F|
 ____________________________________________________________
```

c) Update #ofRids (1 for newly inserted key, incremented by 1 for existing key), F, N (incremented by 1 if key is inserted).

We wrote our own unit test to check the resulting B+ tree is valid. The check includes:
(1) all keys are in the increasing order in each page and all RIDs are in the increasing order of (Page Num, Slot Num) in each leaf page
(2) the shape of the tree looks good (print out and eyeball check)
(3) next page pointers in leaf pages work well (can point to correct sibling pages)

Split

If a page has overflow, we split the page and then insert the entry based on its key value. We try to maintain the old page to be at least half full and distribute the remaining entries to the new page. If a leaf page has overflow and is split, we copy up the first key in the new leaf page to the parent page:

- In the case of a non-leaf page, we push up the middle key to the parent page.
- In the case of a root page, we split the root page and re-define the two pages to be non-leaf pages, create a new root page, and then push up the middle key to that page.

To facilitate the copy-up and push-up work, we use an array of size of height to store the traverse path and then use the array to identify which parent page should be inserted.

Deletion

We implemented lazy deletion. Deleting an entry can be divided into four steps:

a) traverse from root to leaf node based on the given key

b) read the KEY and number of rids in this KEY, check if this key matches the given key
 i. if yes, go through each rid until the read rid matches the given rid, then goes to step c)
 ii. if no, offset to next KEY by increasing offset by KEY length and (# of Rids)*8, then repeat step b)
 *Note: If the program goes through the whole leaf page but does not find any Key/Rid which matches the given one, it indicates no such Key/Rid pair in the tree. Return -1 (failed).

c) Key/Rid is found! Offset all keys and rids after this Key/Rid pair. For instance:

Say we want to delete Key2/Rid#6

Before Deletion:

```
 ____________________________________________________________
|_Flag_|_Next#_|_Key1_|_#ofRids_|_Rid#1_|_Rid#2_|_Rid#3_|_Rid#4_|
|_Key2_|_#ofRids_|_Rid#5_ _Rid#6_ _Key3_|_#ofRids_|_Rid#7_|     |
|                                                               |
|                  …                                            |
|                              …                                |
|                                                               |
|          …  |_KeyN_|_#ofRids_|_Rid#i_|_Rid#j_|_Rid#k_|   …    |
|                                                               |
|                                                           ____|
|                                                          |N|F|
 ____________________________________________________________
```

After Deletion:

```
 ____________________________________________________________
|_Flag_|_Next#_|_Key1_|_#ofRids_|_Rid#1_|_Rid#2_|_Rid#3_|_Rid#4_|
|_Key2_|_#ofRids_|_Rid#5_|_Key3_|_#ofRids_|_Rid#7_|             |
|                                                               |
|                  …                                            |
|                              …                                |
|                                                               |
|          …  |_KeyN_|_#ofRids_|_Rid#i_|_Rid#j_|_Rid#k_|   …    |
|                                                               |
|                                                           ____|
|                                                          |N|F|
 ____________________________________________________________
```

If the number of rids of certain key is zero after deleting, we have to delete the key as well.

Let's use the "After Deletion" example above:

Say we want to delete Key2/Rid#5

Before Deletion:

```
 ______________________________________________________________
|_Flag_|_Next#_|_Key1_|_#ofRids_|_Rid#1_|_Rid#2_|_Rid#3_|_Rid#4_|
  Key2 | #ofRids | Rid#5   _Key3_|_#ofRids_|_Rid#7_|            |
|                                                               |
|                   …                                           |
|                               …                               |
|                                                               |
|           …  |_KeyN_|_#ofRids_|_Rid#i_|_Rid#j_|_Rid#k_|   …   |
|                                                               |
|                                                           ____|
|                                                           |N|F|
 ______________________________________________________________
```

After Deletion:

```
 ______________________________________________________________
|_Flag_|_Next#_|_Key1_|_#ofRids_|_Rid#1_|_Rid#2_|_Rid#3_|_Rid#4_|
|_Key3_|_#ofRids_|_Rid#7_|                                      |
|                                                               |
|                   …                                           |
|                               …                               |
|                                                               |
|           …  |_KeyN_|_#ofRids_|_Rid#i_|_Rid#j_|_Rid#k_|   …   |
|                                                               |
|                                                           ____|
|                                                           |N|F|
 ______________________________________________________________
```