Lab 2: B+-Tree Index

Start Assignment

Due Sunday by 11:59pm **Points** 100 **Submitting** a text entry box **Available** after Feb 8 at 12am

Getting Started

This lab will be built upon your code for lab 1. We provide a tarball with some new starter code which you need to merge with your lab0.

1. Make sure you commit every change to Git and `git status` is clean
2. Download lab2.tar.xz (https://psu.instructure.com/courses/2240486/files/145730636?wrap=1) (https://psu.instructure.com/courses/2240486/files/145730636/download?download_frd=1) and untar it inside your repo. You should now have a directory called lab2 inside your repo
3. `cd lab2` and `./import_supplemental_files.sh`
4. Add a new function definition `Varlen::ComputeFreeSpace()` **(listed below in Task 2)** as a public interface in `include/storage/VarlenDataPage.h`, and its empty implementation (simply return something like `-1`) in `src/storage/VarlenDataPage.cpp`.
5. `cd path_to_your_repo` and `git commit -m "Start lab2"`

The code should build without compilation errors once the supplemental files are imported, but most of the additional tests are likely to fail. You may list all the tests in your build directory using the `ctest -N` command.

A few useful hints:

1. Some parts of this lab will heavily rely on your variable-length data page implementation from lab1. So, you need to make sure the contract and structure of `VarlenDataPage` clearly before your coding procedure. Note that this lab also requires you to write a fair amount of code, so **START EARLY**.
2. The description below is meant to be a brief overview of the tasks. **Always** refer to the special document blocks in the header and source files for a more detailed specification of how the functions should behave.
3. You may add any definition/declaration to the classes or functions you are implementing, but do not change the signature of any

public functions. Also, **DO NOT** edit those source files not listed in *source files* to modify below, as they will be replaced during tests.

4. You may ignore any function/class that's related to multi-threading and concurrency control since we are building a single-threaded DBMS this semester. Thus, your implementation doesn't have to be thread-safe.
5. We provide a list of tasks throughout this document, which is a suggestion of the order of implementation such that your code may pass a few more tests after you complete a task. It, however, is not the only possible order and you may find an alternative order that makes more sense to you. It is also possible that your code still could pass certain tests, which is supposed to because of some unexpected dependencies on later tasks, in which case you might want to refer to the test code implementation in the `test/` directory.

Overview

In this lab, you will work on the major index structure used in Taco-DB, namely B+-Tree. More specifically, your goal is to understand the structure of the B+-Tree and implement core pieces of insert/delete/search functions. This B+-Tree supports variable-length keys, thus does not have a stable fanout. Splitting and merging are based on the actual load of internal and leaf pages rather than the number of entries. To simplify our implementation and maximize code reuse, TacoDB's B+-Tree utilizes `VarlenDataPage` to organize all its nodes. In the lab handout, we will find basic tests for various B+-Tree utility functions and some small-scale end-to-end tests. When grading, there will be larger-scale system tests (which are hidden to you) to further evaluate the correctness and efficiency of your implementation.

Index Interface

Source files to READ:

- `include/index/Index.h`
- `include/index/IndexKey.h`
- `src/index/Index.cpp`

Source files to modify: **None**

As a general and important interface in an RDBMS, `Index` should be the base class of all different types of indexes. Its basic public interface is defined in `Index.h`. The metadata information, including its base table, index type, virtual file FID, etc., is stored in the catalog and represented as `IndexDesc` in memory. Static function `Index::Initialize()` will do some essential checks and initialize basic physical storage layout on the virtual file specified in `IndexDesc` allocated for the index, while static function `Index::Create()` will initialize an `Index` class instance so that other components of DBMS can make calls on it.

An index should override public interfaces for inserting/deleting a record and scanning through all the index entries. All indexes in TacoDB are supposed to be organized as Alternative 2, where each index entry is a `(key, rid)` pair. Note that `rid` is unique for each record. Thus, all index entries are unique even though keys in the records are not. `Index::StartScan()` will initialize an iterator that will scan through all index entries falling into a particular key range. When scanning through the iterator, you need to ensure that each index entry will be touched **EXACTLY ONCE**. Besides, since all indexes of the current version of TacoDB are range indexes, your index iterators should scan through index entries in the ascending order of their keys.

B+-Tree Structure

Source file to READ:

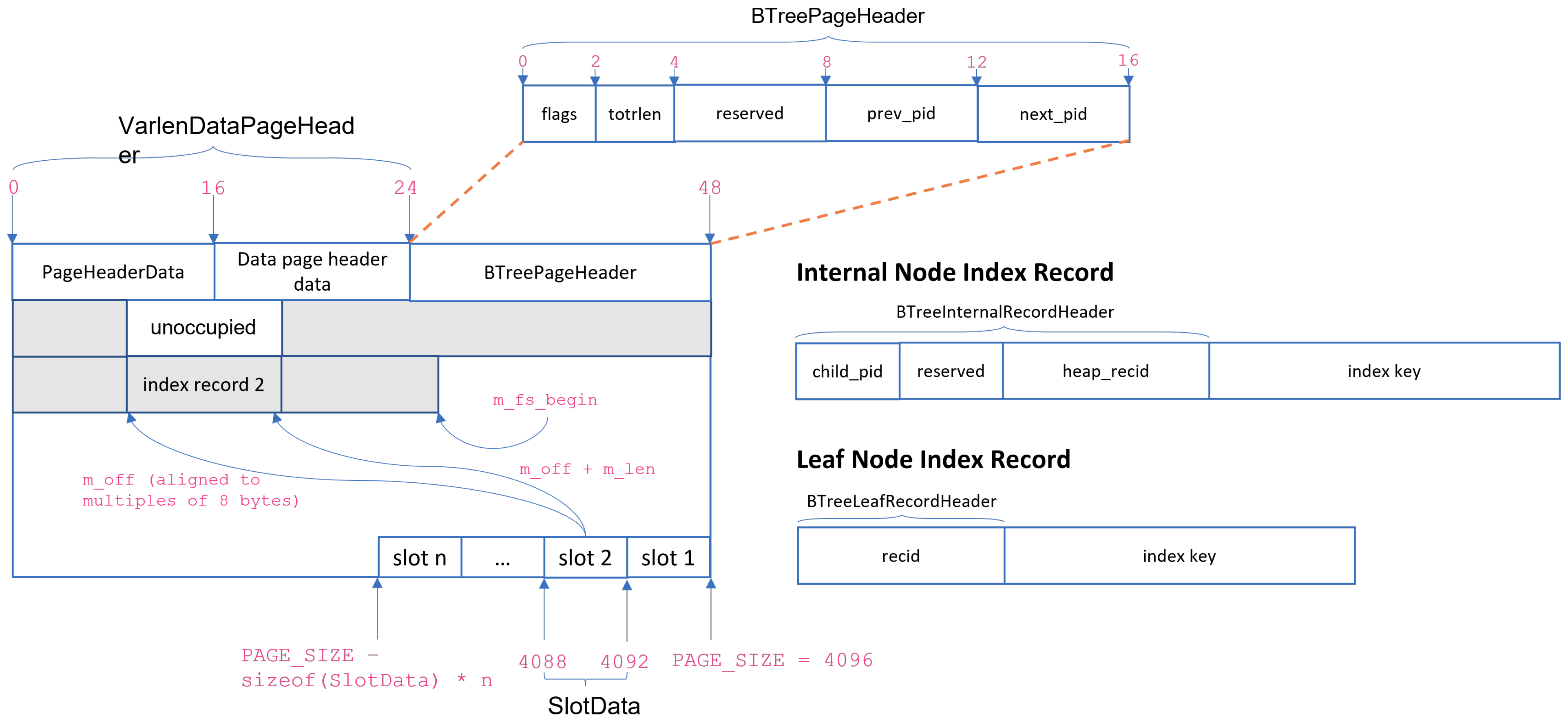
- `include/index/btree/BTree.h`

Source files to modify:

- `src/index/btree/BTree.cpp`
- `src/index/btree/BtreeUtils.cpp`
- `src/storage/VarlenDataPage.h`
- `src/storage/VarlenDataPage.cpp`

The core physical structure of the B+-Tree in TacoDB is as explained during our lectures, which contains internal nodes and leaf nodes. All the structures of a B+-Tree live in a virtual `File` managed by `FileManager`, whose first page is a meta page containing a page number pointing to the root. Note the root node of a B+-Tree can be either a leaf or internal node, depending on when there is enough space for a leaf page to fit all index entries.

In this lab, you will need to rely on your `VarlenDataPage` implementation in lab 1 to construct both leaf and internal nodes. The figure below shows the physical layout of the page representing a B+-Tree node. You can see that its first 24 bytes are exactly the same as those of `VarlenDataPage` for general heap file data. The only differences are: (1) Index entries for leaf nodes and separator keys with links to corresponding children for internal nodes are treated as index “records” in this specialized `VarlenDataPage`. (2) B+-Tree nodes store their specialized meta information in the user-defined data region. You can reserve enough space for that region through passing `usr_data_sz` when calling `VarlenDataPage::Initialize`. You can find all the fields of that region in `struct BTreePageHeaderData` defined in `include/index/btree/BTreeInternal.h`. Specifically, it includes flags indicating if this is the root node and if this is a leaf node, the pointers to its two neighbor siblings on the same level, and the total number of bytes used by index “records” in the page.



Task 1: Implement functions for laying out B+-Tree pages and some other basic utilities:

- `BTree::CreateNewBTreePage()`
- `BTree::GetBTreeMetaPage()`
- `BTree::CreateLeafRecord()`

- `BTree::CreateInternalRecord()`
- `BTree::Initialize()`
- `BTree::IsEmpty()`

Your code should allocate a new page and pin it after `BTree::CreateNewBTreePage()` is called, then return the location of pinned buffer frame. `BTree::GetBTreeMetaPage()` should pin the meta page of this B+-Tree and return the pinned frame location.`BTree::CreateLeafRecord()` and `BTree::CreateInternalRecord()` are in charge of creating "records" in B+-Tree pages. In particular, records for internal nodes should be contiguous bytes containing `BTreeInternalRecordHeaderData` and the index key (strictly in this order with alignment). You can see the `m_heap_rec_id` is also the part of `BTreeInternalRecordHeaderData` as `(key, rec_id)` will always be unique even if this is not a unique index. **The first record in an internal node should only contain `BTreeInternalRecordHeaderData` since its starting key is stored in its parent.** The key of all records stored in the subtree between two neighboring keys `k1` and `k2` of an internal node falls in a left-closed right-open range `[k1, k2)`. The leaf node is much simpler, where each "record" is contiguous bytes laying out `BTreeLeafRecordHeaderData` (which is effectively just a RID) and the index key (in this order with alignment).

The next step is to finish the implementation of `BTree::Initialize()` by allocating the root node and initializing the B+-Tree meta page. You can find the virtual file ID allocated for the current index by calling `idxdesc->GetIndexEntry()->idxfid()`. `IsEmpty()` is used to check if an **initialized** B+-Tree contains any index entry.

Your code is likely to pass the following tests at this point:

- `BasicTestBTreeUtils.TestCreateNewBTreePage`
- `BasicTestBTreeUtils.TestGetBTreeMetaPage`
- `BasicTestBTreeUtils.TestCreateBTreeRecords`
- `BasicTestBTreeUtils.TestBTreeIsEmpty`

Task 2: Implement additional utility functions for variable-length data page:

- `VarlenDataPage::ComputeFreeSpace()`
- `VarlenDataPage::InsertRecordAt()`
- `VarlenDataPage::RemoveSlot()`

These utility functions are generally useful, especially in handling B+-Tree insertion and deletion. You can find the function contract for `VarlenDataPage::InsertRecordAt()` and `VarlenDataPage::RemoveSlot()` in `include/storage/VarlenDataPage.h`. The contract for `VarlenDataPage::ComputeFreeSpace()` is listed below. Please manually add it into `VarlenDataPage` definition.

```
    /*!
     * Computes the size of the free space on the page if there are \p num_recs
     * records inserted to an empty VarlenDataPage with a user data area of
     * size \p usr_data_sz and the total length of the records is \p
     * total_reclen.  The total length of the records should be the sum of the
     * length of the individual records after 8-byte alignment adjustments.
     *
     * Returns the remaining size if the imaginary insertions would fit on the
     * page. Otherwise, return -1.
     */
    static FieldOffset ComputeFreeSpace(FieldOffset usr_data_sz,
                                        SlotId num_recs,
                                        FieldOffset total_reclen);
```

Your code is likely to pass the following tests at this point:

- `BasicTestSortedDataPage.TestComputeFreeSpace`
- `BasicTestSortedDataPage.TestInsertRecordDeterministic`
- `BasicTestSortedDataPage.TestInsertRecordRandom`
- `BasicTestSortedDataPage.TestInsertRecordOutOfBound`
- `BasicTestSortedDataPage.TestRemoveAllRecordDeterministic`
- `BasicTestSortedDataPage.TestRemoveAllRecordRandomly`
- `BasicTestSortedDataPage.TestRemoveAllRecordRandomlyOutOfBound`
- `BasicTestSortedDataPage.TestInsertionWithCompationExistingSlot`

B+-Tree Search

Source files to modify:

- `src/index/btree/BTreeSearch.cpp`
- `src/index/btree/BTreeIterator.cpp`

Task 3: Implement the utility function to compare two keys and complete the recursive B+-Tree point lookup logic:

- `BTree::BTreeTupleCompare()`
- `BTree::BinarySearchOnPage()`
- `BTree::FindLeafPage(bufid, key, recid, p_search_apth)` – the second overload

To compare two index records, you should compare not only the index key but also the record ID. This is because we want to make sure all index records are unique to each other to enforce a total order in the index. You can compare two index keys by calling `TupleCompare(key1, key2, key_schema, lt_func, eq_func)`. You should compare on record ID further if the index key comparison turns out to be equal. Further, we will enforce dictionary order for partial key matches. This means a key like `(1, 2)` is always greater than its prefix `(1, phi)` (where `phi` means unspecified). You can compare a full key with its prefix using `TupleCompare`, but it will turn out to be equal. So you have to resolve that in `BTree::BTreeTupleCompare()` as well.

You should write a simple binary search to look up the key in a B+-Tree Node. The second overload of `BTree::FindLeafPage()` is the core recursion logic of traversing the tree. Please be careful about pinning and unpinning the pages during the traversal. At any time during a search, there should be **ONLY ONE** B+-Tree node pinned in the buffer.

Your code is likely to pass the following tests at this point:

- `BasicTestBtreeSearch.TestBTreeTupleCompareFullKey`
- `BasicTestBtreeSearch.TestBTreeTupleComparePrefixKey`
- `BasicTestBtreeSearch.TestBinarySearchOnLeafPageFixedlen`
- `BasicTestBtreeSearch.TestBinarySearchOnInternalPageFixedlen`
- `BasicTestBtreeSearch.TestBinarySearchOnLeafPageVarlen`
- `BasicTestBtreeSearch.TestBinarySearchOnInternalPageVarlen`

Task 4: Finish up the B+-Tree iterator scan logic:

- `BTree::Iterator::Next()`

You should use the sibling pointers on the leaf node to iterate through index entries in order. You should also check if the scan is completed (either you are running out of index entries or the next index entry falls out of query range).

Your code is likely to pass the following tests at this point:

- `BasicTestBtreeSearch1.TestBTreeFullScanFixedlen`

- `BasicTestBtreeSearch2.TestBTreeRangeScanFixedlen`
- `BasicTestBtreeSearch3.TestBTreeFullScanVarlen`
- `BasicTestBtreeSearch4.TestBTreeRangeScanVarlen`

B+-Tree Insertion

Source files to modify:

- `src/index/btree/BTreeInsert.cpp`

Task 5: Implement the functions to find the correct slot on a leaf node for insertion, and insert an index record on a B-Tree page:

- `BTree::FindInsertionSlotIdOnLeaf()`
- `BTree::InsertRecordOnPage()`

Finding the correct leaf has roughly the same logic as B+-Tree search. In addition, you should remember the path down by keeping track of which B+-Tree nodes (a.k.a B+-Tree page ID) you have touched and which slot number you took during the traversal. This information is useful in later split procedures.

Your code is likely to pass the following tests at this point:

- `BasicTestBTreeInsert.TestBTreeInsertFindSlotOnLeaf`
- `BasicTestBTreeInsert.TestBTreeInsertOnPageNoSplit`
- `BasicTestBTreeInsert.TestBTreeInsertNoSplit`

You should first implement the index record insertion logic without splitting to make it work in easy cases.

Task 6: Implement node splitting and complete `BTree::InsertRecordOnPage()` by adding triggers for recursive splitting:

- `BTree::CreateNewRoot()`
- `BTree::SplitPage()`
- `BTree::InsertRecordOnPage()`

The trigger of splitting a page should be added in `BTree:InsertRecordOnPage()` when there is no space left to insert a new index record.

Consider a local case of splitting a node into two. There will be two nodes originally in the tree being affected, namely the node you are splitting and its parent. During this procedure, `BTree::SplitPage()` will trigger another `BTree::InsertRecordOnPage()` at the parent node. This procedure can trigger another split on the parent node, thus can be recursive and go all the way to the root. You will need to handle this special case of root node splitting with `BTree::CreateNewRoot()`.

Again, during this recursive procedure, you should be very careful about pinning and unpinning B+-Tree pages. At any time during insertion, there should be **AT MOST FOUR** pages pinned at the same time. *(Think about which four pages?)*

Your code is likely to pass the following tests at this point:

- `BasicTestBTreeSplitPage.TestSplitLeafPageWithUniformReclenSmallKey`
- `BasicTestBTreeSplitPage.TestSplitLeafPageWithUniformReclenLargeKey`
- `BasicTestBTreeSplitPage.TestSplitInternalPageWithUniformReclenSmallKey`
- `BasicTestBTreeSplitPage.TestSplitInternalPageWithUniformReclenLargeKey`
- `BasicTestBTreeInsert.TestBTreeInsertSingleSplit`
- `BasicTestBTreeInsert.TestBTreeInsertRecursiveSplit`
- `BasicTestBTreeInsert.TestBTreeInsertUnique`
- `BasicTestBTreeInsert.TestBTreeInsertVarlenKey`

B+-Tree Deletion

Source files to modify:

- `src/index/btree/BTreeDelete.cpp`
- `src/storage/VarlenDataPage.h` *(bonus)*
- `src/storage/VarlenDataPage.cpp` *(bonus)*

Task 7: Implement the function to find the location of the deleting record and the function for erasing an index record on a B-Tree page:

- `BTree::FindDeletionSlotIdOnLeaf()`
- `BTree::DeleteSlotOnPage()`

Locating the record for deletion in the leaf level should also remember the traversal path the same as what you did in insertion. Note that, when we are searching for a leaf page to locate a key to delete without specifying a valid record ID, it might end up on a leaf page that is to the left of any matching keys, because an invalid record id is treated as smaller than any valid record id. In this case, you may have to move to the right page (`m_next_page`) to find a matching key for deletion (if there is any, it must be in the first slot of the right page). One tricky issue is that the parent of the original leaf page may be different from the leaf page on the right. If that happens, we ask you to simply add one to the slot ID of the last `PathItem` of the search path in the `BTree::FindDeletionSlotIdOnLeaf()` – `BTree::HandleMinPageUsage()` will handle the logic of moving to the right to find the correct parent. Then, you should first implement erasing the index record without considering merging or rebalancing in `BTree::DeleteSlotOnPage()`.

Your code is likely to pass the following tests at this point:

- `BasicTestBTreeDelete.TestBTreeDeleteFindKey`
- `BasicTestBTreeDelete.TestBTreeDeleteFindKeyOverflow`
- `BasicTestBTreeDelete.TestBTreeDeleteSlot`
- `BasicTestBTreeDelete.TestBTreeDeleteSlotRoot`

Task 8: Implement node merging and complete `BTree::DeleteSlotOnPage()`:

- `BTree::MergePages()`

Merging should happen when a B+-Tree node is under-utilized (it has free space of more than 60% of the page size), which is triggered by `BTree::HandleMinPageUsage()` called in `BTree::DeleteKey()`. *Since node rebalancing is assigned as a bonus, you don't have to implement it to get the full score. The test we use will dynamically specify which test enables rebalancing.*

Node merging should be conducted between two sibling pages next to each other, and delete an index record on the parent. This may recursively trigger another `BTree::MergePages()` on the parent level, which is already handled recursively in `BTree::HandleMinPageUsage()`. The merge may go all the way up to the root, and eventually change the root node PID stored in B+-Tree meta page. You don't have to deal with this as well, since we have already handled it for you in `BTree::HandleMinPageUsage()`. During this recursive deletion and merging process, there should be **AT MOST FOUR** pages pinned at the same time. *(which four pages?)*

Your code is likely to pass the following tests at this point:

- `BasicTestBTreeMergePages.TestMergeLeafPageWithUniformReclenSmallKey`
- `BasicTestBTreeMergePages.TestMergeLeafPageWithUniformReclenLargeKey`
- `BasicTestBTreeMergePages.TestMergeInternalPageWithUniformReclenSmallKey`
- `BasicTestBTreeMergePages.TestMergeInternalPageWithUniformReclenLargeKey`
- `BasicTestBTreeMergePages.TestMergeLeafPageWithExactFill`
- `BasicTestBTreeMergePages.TestMergeInternalPageWithExactFill`
- `BasicTestBTreeDelete1.TestBTreeDeleteSingleMerge`
- `BasicTestBTreeDelete1.TestBTreeDeleteRecursiveMerge`
- `BasicTestBTreeDelete2.TestBTreeDeleteVarlenKey`

Task 9: (Bonus): Implement node rebalancing and optimize `BTree::DeleteSlotOnPage()` with rebalancing logic:

- `VarlenDataPage::ShiftSlots()`
- `BTree::RebalancePages()`
- `BTree::DeleteSlotOnPage()`

`RebalancePages()` is called when we have an under-utilized page, but it can’t be merged with a sibling page because its page usage will exceed the page size if they are merged. `RebalancePages()` should try to move records from the page with a higher page usage to the one with a lower page usage, such that both have a page usage no less than `BTREE_MIN_PAGE_USAGE`. There are usually many choices of how many tuples to move. Different from `SplitPage()` which can always succeed, `RebalancePages()` may fail because the new separator key after the rebalancing may not fit into the parent page (as it might be longer than the old separator key). You should not make a choice such that the new separator key cannot fit into the parent page. Out of all valid choices, you should choose the one that minimizes the page usage difference between the two pages. If there’s no valid choice, you should not change either of the pages and should return `false` from `RebalancePages()`.

During this rebalancing process, there should be **AT MOST THREE** pages pinned at the same time. *(Which tree pages?)*

You might find it useful to have a utility function in `VarlenDataPage` with a contract like below:

```
    /*!
     * Shifts all the (occupied or unoccupied) slots to the right by n slots to
     * truncate existing slots or to the left by n slots to reserve new slots.
     *
     * If \p truncate is true, the first \p n slots are removed and the
     * remaining slots are moved to occupy consecutive slots from slot ID
     * `GetMinSlotId()`. It is **not** an error if \p n is greater than the
     * number of slots on the page, in which case, all the slots are removed
     * (and the page becomes an empty page).
     *
     * If \p truncate is false, all the existing slots are moved to occupy
     * consecutive slots from slot ID `n + 1`. The first \p n slots should then
     * be reserved as unoccupied slots. ShiftSlots is allowed to compact the
     * page to make room for the new slots when \p truncate is false. It is
     * a fatal error if it cannot make room for new slots (i.e., the caller is
     * responsible for ensuring there's enough room).
     */
    void ShiftSlots(SlotId n, bool truncate);
```

Test for rebalancing can be found with this package (lab2_bonus.tar.xz) (https://psu.instructure.com/courses/2240486/files/145730637?wrap=1) (https://psu.instructure.com/courses/2240486/files/145730637/download?download_frd=1) . Please use the importing commands as above to import new tests.

Your code is likely to pass the following tests at this point:

- `BonusTestSortedDataPage.TestShiftSlotsToRight`
- `BonusTestSortedDataPage.TestShiftSlotsToRightWithInvalidSid`
- `BonusTestSortedDataPage.TestShiftSlotsToRightThenToLeft`
- `BonusTestSortedDataPage.TestShiftSlotsToLeftNoSpace`
- `BonusTestBTreeRebalancePages.TestRebalanceLeafPagesWithUniformReclenLargeKey`
- `BonusTestBTreeRebalancePages.TestRebalanceInternalPagesWithUniformReclenLargeKey`
- `BonusTestBTreeRebalancePages.TestRebalanceLeafPagesBestSepWontFitL2R`
- `BonusTestBTreeRebalancePages.TestRebalanceLeafPagesBestSepWontFitR2L`
- `BonusTestBTreeRebalancePages.TestRebalanceLeafPagesNoFitL2R`
- `BonusTestBTreeRebalancePages.TestRebalanceLeafPagesNoFitR2L`
- `BonusTestBTreeRebalancePages.TestRebalanceInternalPagesBestSepWontFitL2R`
- `BonusTestBTreeRebalancePages.TestRebalanceInternalPagesBestSepWontFitR2L`
- `BonusTestBTreeRebalancePages.` `TestRebalanceInternalPagesNoFitL2R`
- `BonusTestBTreeRebalancePages.` `TestRebalanceInternalPagesNoFitR2L`

Submission Guideline

When you are ready to submit the lab, push your code to Github and find the latest commit hash. Git commit hash can be found on the Github website or through the command git log. Copy and paste the commit hash into the text box for this assignment.