Lab 4 (PA 2)

Programming Assignment 2

Released: 10/2

Due: 10/16

In this assignment you will implement the page replacement procedure in the buffer and a B+ tree index for efficient lookups and range scans. We supply you with all of the low-level code you will need to implement the tree structure. You will implement searching, splitting pages, redistributing tuples between pages, and merging pages(the last two are optional).

The remainder of this document gives some suggestions about how to start coding, describes a set of exercises to help you work through the assignment, and discusses how to hand in your code. This assignment requires you to write a fair amount of code, so we encourage you to start early!

1. Getting started

1.1. Adding skeleton code for assignment 2

You should begin with the code you submitted for assignment 2 (if you did not submit code for assignment 1, or your solution didn't work properly, contact us to discuss options). We have provided you with extra test cases for this assignment that are not in the original code distribution you received. Again, the unit tests we provide are to help guide your implementation along, but they are not intended to be comprehensive or to establish correctness.

You will need to add these new test cases to your release. The easiest way to do this is to untar the new code in the sa directory as your top-level simpledb directory, as follows:

- Make a backup of your assignment1 solution.

- Change to the directory that contains your top-level simpledb code:

```
$ cd CS660-pa1
```

- Download the new tests and skeleton code for assignment2 from **Piazza**
- Extract the new files for assignment2 and add them to your assignment1 folder.
- **Eclipse users** will have to take one more step for their code to compile. (First, make sure the project is in your workspace in Eclipse following similar steps as done in assignment1.) Under the package explorer, right click the project name (probably simpledb), and select **Properties**. Choose **Java Build Path** on the left-hand-side, and click on the **Libraries** tab on the right-hand-side. Push the **Add JARs...** button, select **zql.jar** and **jline-0.9.94.jar**, and push **OK**, followed by **OK**. Your code should now compile.
- If you want to do the extra tests please copy the `BTreeFileDeleteTest.java` in the folder `test extra/simpledb` and `test extra/simpledb/systemtest` into the corresponding place in the `test` folder.

1.2. *Implementation hints*

We encourage you to read through this entire document to get a feel for the high-level design of SimpleDB before you write code.

We also suggest exercises along this document to guide your implementation, but you may find that a different order makes more sense for you. As before, we will grade your assignment by looking at your code and verifying that you have passed the test for the ant targets `test` and `systemtest`. See Section 3.4 for a complete discussion of grading and list of the tests you will need to pass. More details on the steps in this outline, including exercises, are given in Section 2 below.

2. Implementation Guide

2.1. *Page eviction for SimpleDB Buffer*

In assignment 2, you will choose a page eviction policy and instrument any previous code that reads or creates pages to implement your policy.

When more than numPages pages are in the buffer pool, one page should be evicted from the pool before the next is loaded. The choice of eviction policy is up to you; it is not necessary to do something sophisticated. Describe your policy in the writeup. Notice that `BufferPool` asks you to implement a `flushAllPages()` method. This is not something you would ever need in a real implementation of a buffer pool. However, we need this method for testing purposes. You should never call this method from any real code.

Because of the way we have implemented ScanTest.cacheTest, you will need to ensure that your flushPage and fl .llPages methods do no evict pages from the buffer pool to properly pass this test. flushAllPages should call flushPage on all pages in the BufferPool, and flushPage should write any dirty page to disk and mark it as not dirty, while leaving it in the BufferPool. The only method which should remove page from the buffer pool is evictPage, which should call flushPage on any dirty page it evicts.

Exercise 1. Fill in the `flushPage()` method and additional helper methods to implement page eviction in:

- `src/simpledb/BufferPool.java`

At this point, your code should pass the `EvictionTest` system test.

Since we will not be checking for any particular eviction policy, this test works by creating a BufferPool with 16 pages (NOTE: while DEFAULT_PAGES is 50, we are initializing the BufferPool with less!), scanning a file with many more than 16 pages, and seeing if the memory usage of the JVM increases by more than 5 MB. If you do not implement an eviction policy correctly, you will not evict enough pages, and will go over the size limitation, thus failing the test.

2.2. Search in B+ Tree

Take a look at `BTreeFile.java`. This is the core file for the implementation of the B+Tree and where you will write all your code for this assignment. Unlike the HeapFile, the BTreeFile consists of four different kinds of pages. As you would expect, there are two different kinds of pages for the nodes of the tree: internal pages and leaf pages. Internal pages are implemented in `BTreeInternalPage.java`, and leaf pages are implemented in `BTreeLeafPage.java`. For convenience, we have created an abstract class in `BTreePage.java` which contains code that is common to both leaf and internal pages. In addition, header pages are implemented in `BTreeHeaderPage.java` and keep track of which pages in the file are in use. Lastly, there is one page at the beginning of every BTreeFile which points to the root page of the tree and the first header page. This singleton page is implemented in `BTreeRootPtrPage.java`. Familiarize yourself with the interfaces of these classes, especially `BTreePage`, `BTreeInternalPage` and `BTreeLeafPage`. You will need to use these classes in your implementation of the B+Tree.

Your first job is to implement the `findLeafPage()` function in `BTreeFile.java`. This function is used to find the appropriate leaf page given a particular key value, and is used for both searches and inserts. For example, suppose we have a B+Tree with two leaf pages (See Figure 1). The root node is an internal page with one entry containing one key (6, in this case) and two child pointers. Given a value of 1, this function should return the first leaf page. Likewise, given a value of 8, this function should return the second page. The less obvious case is if we are given a key value of 6. There may be duplicate keys, so there could be 6's on both leaf pages. In this case, the function should return the first (left) leaf page.

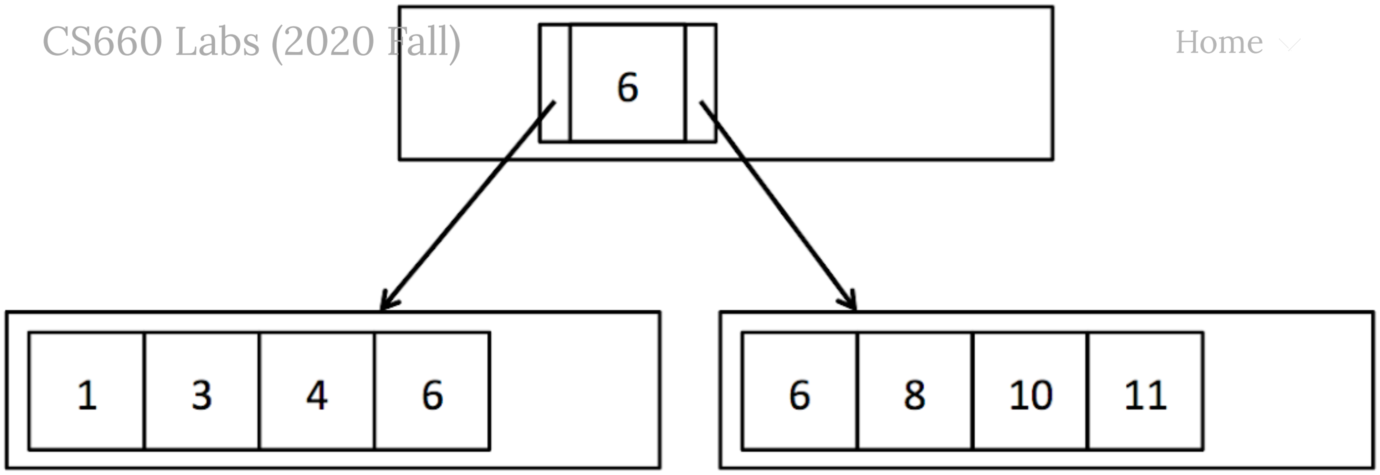


Figure 1: A simple B+ Tree with duplicate keys

Your `findLeafPage()` function should recursively search through internal nodes until it reaches the leaf page corresponding to the provided key value. In order to find the appropriate child page at each step, you should iterate through the entries in the internal page and compare the entry value to the provided key value. `BTreeInternalPage.iterator()` provides access to the entries in the internal page using the interface defined in `BTreeEntry.java`. This iterator allows you to iterate through the key values in the internal page and access the left and right child page ids for each key. The base case of your recursion happens when the passed-in BTreePageId has `pgcateg()` equal to `BTreePageId.LEAF`, indicating that it is a leaf page. In this case, you should just fetch the page from the buffer pool and return it. You do not need to confirm that it actually contains the provided key value f.

Your `findLeafPage()` code must also handle the case when the provided key value f is null. If the provided value is null, recurse on the left-most child every time in order to find the left-most leaf page. Finding the left-most leaf page is useful for scanning the entire file. Once the correct leaf page is found, you should return it. As mentioned above, you can check the type of page using the `pgcateg()` function in `BTreePageId.java`. You can assume that only leaf and internal pages will be passed to this function.

Instead of directly calling `BufferPool.getPage()` to get each internal page and leaf page, we recommend calling the wrapper function we have provided, `BTreeFile.getPage()`. It works exactly like `BufferPool.getPage()`, but takes an extra argument to track the list of dirty pages. This function will be important for the next two exercises in which you will actually update the data and therefore need to keep track of dirty pages.

Every internal (non-leaf) page your `findLeafPage()` implementation visits should be fetched with READ_ONLY permission, except the returned leaf page, which should be fetched with the permission provided as an argument to the function. These permission levels will not matter for this assignment, but they will be important for the code to function correctly in future works.

Also note that as part of this implementation, you will need to implement `IndexPredicate.java` and `P[illegible]icate.java` in order to support comparison operations.

Exercise 2: BTreeFile.findLeafPage()

Fill in the `BTreeFile.findLeafPage()` method in:

- `src/simpledb/BTreeFile.java`

After completing this exercise, you should be able to pass all the unit tests in `PredicateTest.java`, `BTreeFileReadTest.java` and the system tests in `BTreeScanTest.java`.

2.3. Insert in B+ Tree

In order to keep the tuples of the B+Tree in sorted order and maintain the integrity of the tree, we must insert tuples into the leaf page with the enclosing key range. As was mentioned above, `findLeafPage()` can be used to find the correct leaf page into which we should insert the tuple. However, each page has a limited number of slots and we need to be able to insert tuples even if the corresponding leaf page is full.

Attempting to insert a tuple into a full leaf page should cause that page to split so that the tuples are evenly distributed between the two new pages. Each time a leaf page splits, a new entry corresponding to the first tuple in the second page will need to be added to the parent node. Occasionally, the internal node may also be full and unable to accept new entries. In that case, the parent should split and add a new entry to its parent. This may cause recursive splits and ultimately the creation of a new root node.

In this exercise you will implement `splitLeafPage()` and `splitInternalPage()` in `BTreeFile.java`. If the page being split is the root page, you will need to create a new internal node to become the new root page, and update the BTreeRootPtrPage. Otherwise, you will need to fetch the parent page with READ_WRITE permissions, recursively split it if necessary, and add a new entry. You will find the function `getParentWithEmptySlots()` extremely useful for handling these different cases. In `splitLeafPage()` you should "copy" the key up to the parent page, while in `splitInternalPage()` you should "push" the key up to the parent page. See Figure 2 and review lecture slides and section 10.5 in the text book if this is confusing. Remember to update the parent pointers of the new pages as needed (for simplicity, we do not show parent pointers in the figures). When an internal node is split, you will need to update the parent pointers of all the children that were moved. You may find the function `updateParentPointers()` useful for this task. Additionally, remember to update the sibling pointers of any leaf pages that were split. Finally, return the page into which the new tuple or entry should be inserted, as indicated by the provided key field. (Hint: You do not need to worry about the fact that the provided key may actually fall in the exact center of the tuples/entries to be split. You should ignore the key during the split, and only use it to determine which of the two pages to return.)

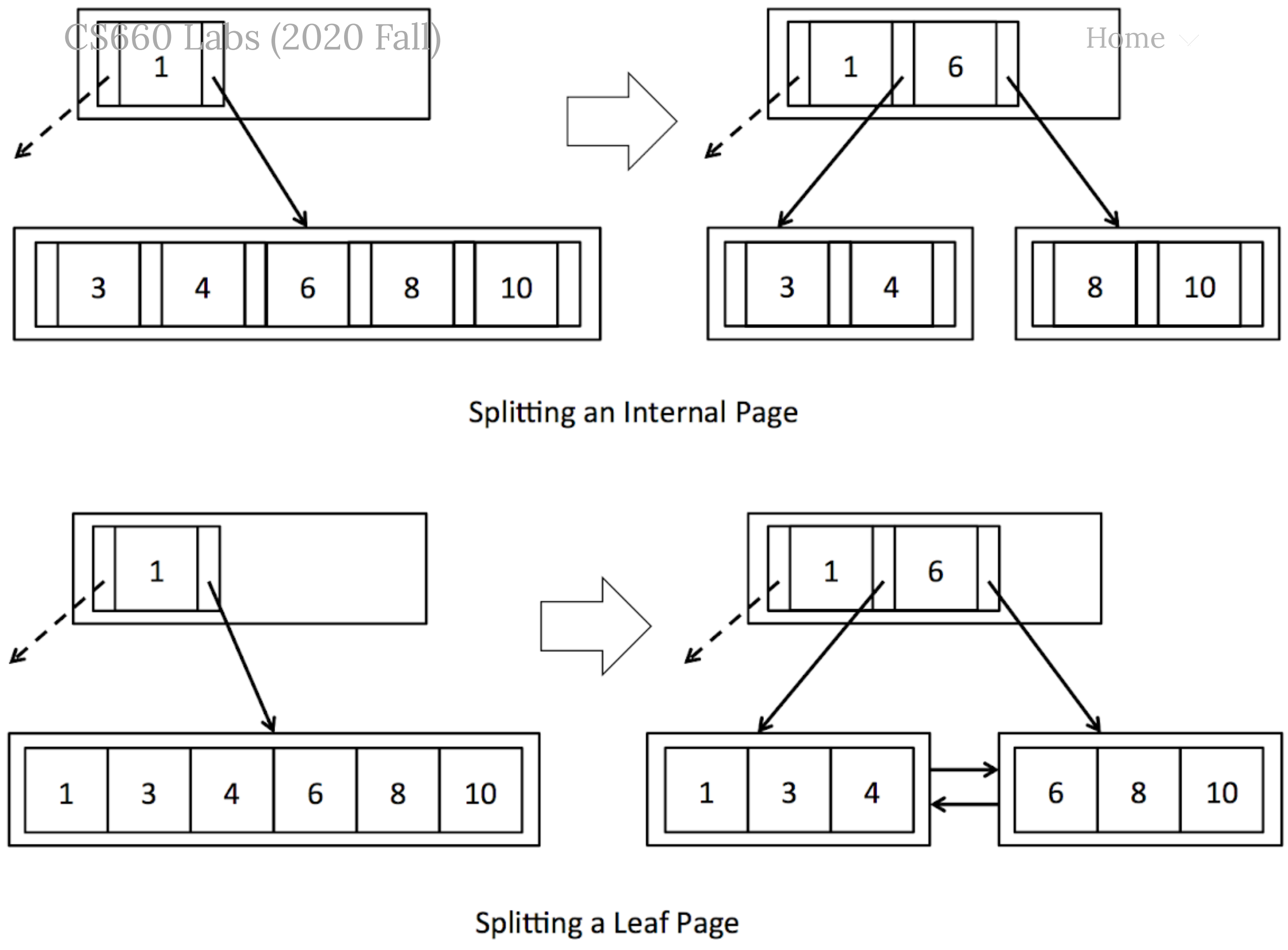


Figure 2: Splitting pages

Whenever you create a new page, either because of splitting a page or creating a new root page, call `getEmptyPage()` to get the new page. This function is an abstraction which will allow us to reuse pages that have been deleted due to merging (covered in the next section).

We expect that you will interact with leaf and internal pages using `BTreeLeafPage.iterator()` and `BTreeInternalPage.iterator()` to iterate through the tuples/entries in each page. For convenience, we have also provided reverse iterators for both types of pages: `BTreeLeafPage.reverseIterator()` and `BTreeInternalPage.reverseIterator()`. These reverse iterators will be especially useful for moving a subset of tuples/entries from a page to its right sibling.

As mentioned above, the internal page iterators use the interface defined in `BTreeEntry.java`, which has one key and two child pointers. It also has a recordId, which identifies the location of the key and child pointers on the underlying page. We think working with one entry at a time is a natural way to interact with internal pages, but it is important to keep in mind that the underlying page does not actually store a list of entries, but stores ordered lists of *ı eys and *m*+1 child pointers. Since the `BTreeEntry` is just an interface and not an object actually stored on the page, updating the fields of `BTreeEntry` will not modify the underlying page. In order to change the data on the page, you need to call `BTreeInternalPage.updateEntry()`. Furthermore, deleting an entry actually deletes only a key and a single child pointer, so we provide the funtions `BTreeInternalPage.deleteKeyAndLeftChild()` and `BTreeInternalPage.deleteKeyAndRightChild()` to make this explicit. The entry's recordId is used to find the key and child pointer to be deleted. Inserting an entry also only inserts a key and single child pointer (unless it's the first entry), so `BTreeInternalPage.insertEntry()` checks that one of the child pointers in the provided entry overlaps an existing child pointer on the page, and that inserting the entry at that location will keep the keys in sorted order.

In both `splitLeafPage()` and `splitInternalPage()`, you will need to update the set of `dirtypages` with any newly created pages as well as any pages modified due to new pointers or new data. This is where `BTreeFile.getPage()` will come in handy. Each time you fetch a page, `BTreeFile.getPage()` will check to see if the page is already stored in the local cache (`dirtypages`), and if it can't find the requested page there, it fetches the page from the buffer pool. `BTreeFile.getPage()` also adds pages to the `dirtypages` cache if they are fetched with read-write permission, since presumably they will soon be dirtied.

Note that in a major departure from `HeapFile.insertTuple()`, `BTreeFile.insertTuple()` could return a large set of dirty pages, especially if any internal pages are split. As you may remember from previous assignment, the set of dirty pages is returned to prevent the buffer pool from evicting dirty pages before they have been flushed.

Warning: as the B+Tree is a complex data structure, it is helpful to understand the properties necessary of every legal B+Tree before modifying it. Here is an informal list:

1. If a parent node points to a child node, the child nodes must point back to those same parents.
2. If a leaf node points to a right sibling, then the right sibling points back to that leaf node as a left sibling.
3. The first and last leaves must point to null left and right siblings respectively.
4. Record Id's must match the page they are actually in.
5. A `key` in a node with non-leaf children must be larger than any key in the left child, and smaller than any key in the right child.
6. A `key` in a node with leaf children must be larger or equal than any key in the left child, and smaller or equal than any key in the right child.
7. A node has either all non-leaf children, or all leaf children.
8. A non-root node cannot be less than half full.

We have implemented a mechanized check for all these properties in the file `BTreeChecker.java`. This method is also used to test your B+Tree implementation in the `systemtest/BTreeFileDeleteTest.java`. Feel free to add calls to this function to help debug your implementation, like we did in BTreeFileDeleteTest.java.

** **

1. The checker method should always pass after initialization of the tree and before starting and after completing a full call to key insertion or deletion, but not necessarily within internal methods.

2. A tree may be well formed (and therefore pass `checkRep()`) but still incorrect. For example, the empty tree will always pass `checkRep()`, but may not always be correct (if you just inserted a tuple, the tree should not be empty).

Exercise 3: Splitting Pages

Fill in the `BTreeFile.splitLeafPage()` and `BTreeFile.splitInternalPage()` methods in:

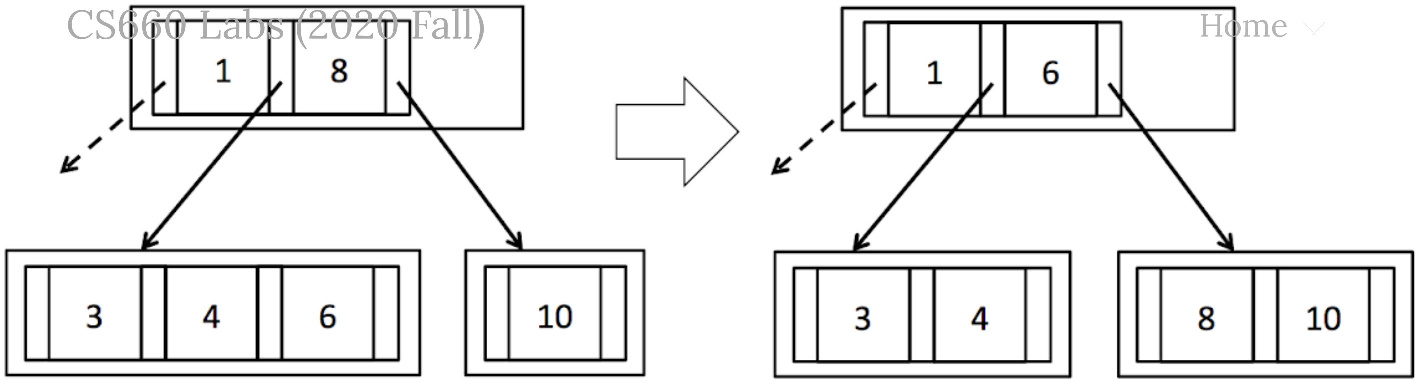
- src/simpledb/BTreeFile.java

After completing this exercise, you should be able to pass the unit tests in `BTreeFileInsertTest.java`. You should also be able to pass the system tests in `systemtest/BTreeFileInsertTest.java`. Some of the system test cases may take a few seconds to complete. These files will test that your code inserts tuples and splits pages correcty, and also handles duplicate tuples.

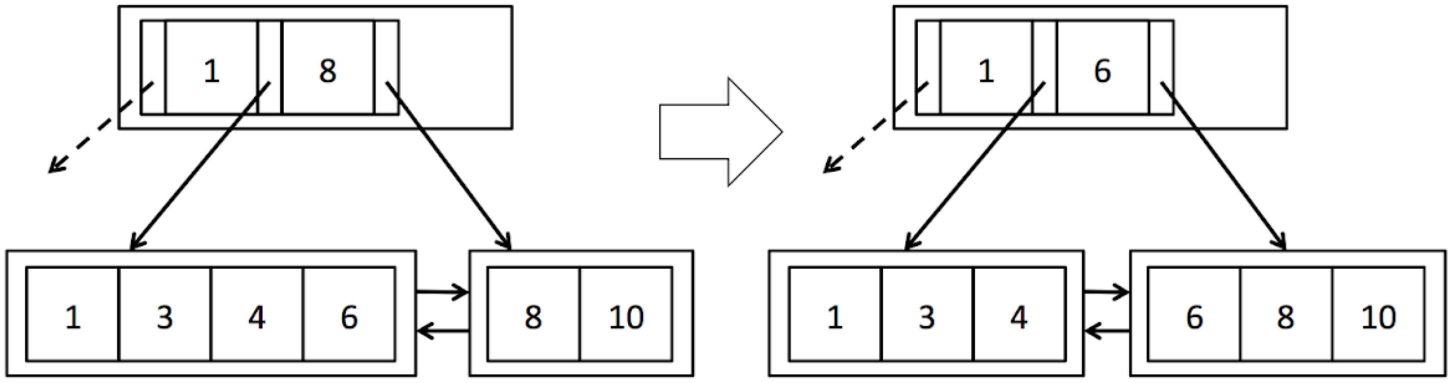
Extra: Delete in B+ Tree (30pts)

This part is not required but if you can pass the tests you will earn some extra credits!

In order to keep the tree balanced and not waste unnecessary space, deletions in a B+Tree may cause pages to redistribute tuples (Figure 3) or, eventually, to merge (see Figure 4).

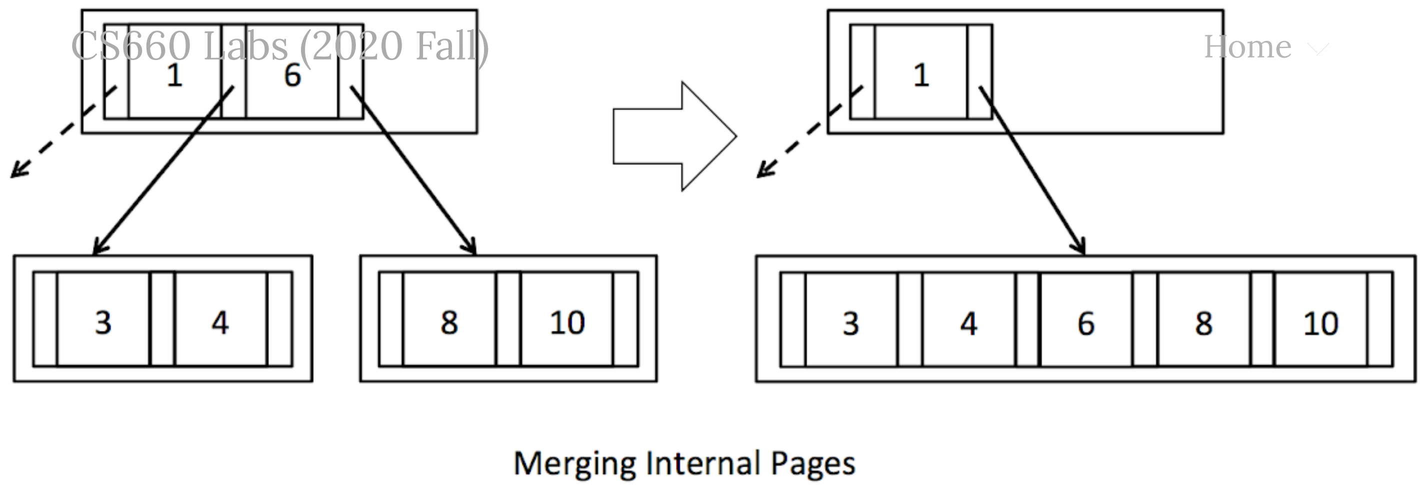


Redistributing Internal Pages



Redistributing Leaf Pages

Figure 3: Redistributing pages



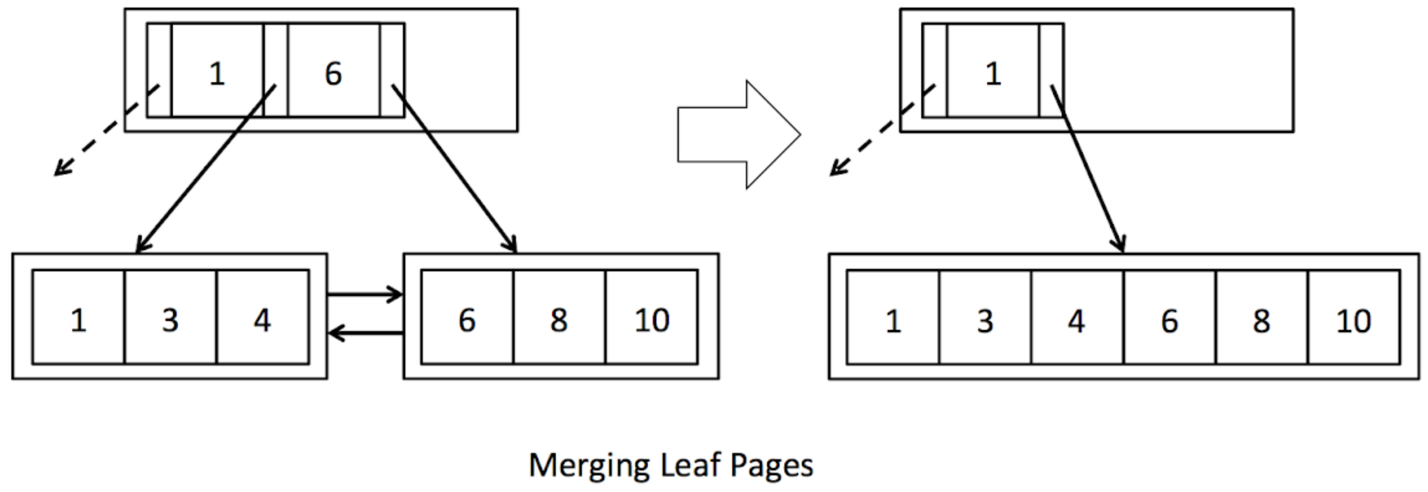


Figure 4: Merging pages

Attempting to delete a tuple from a leaf page that is less than half full should cause that page to either steal tuples from one of its siblings or merge with one of its siblings. If one of the page's siblings has tuples to spare, the tuples should be evenly distributed between the two pages, and the parent's entry should be updated accordingly (see Figure 3). However, if the sibling is also at minimum occupancy, then the two pages should merge and the entry deleted from the parent (Figure 4). In turn, deleting an entry from the parent may cause the parent to become less than half full. In this case, the parent should steal entries from its siblings or merge with a sibling. This may cause recursive merges or even deletion of the root node if the last entry is deleted from the root node.

In this exercise you will implement `stealFromLeafPage()`, `stealFromLeftInternalPage()`, `stealFromRightInternalPage()`, `mergeLeafPages()` and `mergeInternalPages()` in `BTreeFile.java`. In the first three functions you will implement code to evenly redistribute tuples/entries if the siblings have tuples/entries to spare. Remember to update the corresponding key field in the parent (look carefully at how this is done in Figure 3 - keys are effectively "rotated" through the parent). In `stealFromLeftInternalPage()/stealFromRightInternalPage()`, you will also need to update the parent pointers of the children that were moved. You should be able to reuse the function updateParentPointers() for this purpose.

In mergeLeafPages() and mergeInternalPages() you will implement code to merge pages, effectively performing the inverse of splitLeafPage() and splitInternalPage(). You will find the function deleteParentEntry() extremely useful for handling all the different recursive cases. Be sure to call setEmptyPage() on deleted pages to make them available for reuse. As with the previous exercises, we recommend using BTreeFile.getPage() to encapsulate the process of fetching pages and keeping the list of dirty pages up to date.

Exercise 4: Redistributing Pages

Implement BTreeFile.stealFromLeafPage(), BTreeFile.stealFromLeftInternalPage(), BTreeFile.stealFromRightInternalPage() in

- src/simpledb/BTreeFile.java

After completing this exercise, you should be able to pass some of the unit tests in BTreeFileDeleteTest.java (such as testStealFromLeftLeafPage and testStealFromRightLeafPage). The system tests may take several seconds to complete since they create a large B+ tree in order to fully test the system.

Exercise 5: Merging Pages

Implement BTreeFile.mergeLeafPages() and BTreeFile.mergeInternalPages() in

- src/simpledb/BTreeFile.java

Now you should be able to pass all unit tests in BTreeFileDeleteTest.java and the system tests in systemtest/BTreeFileDeleteTest.java.

You have now completed this assignment. Good work!

3. Logistics

You must submit your code (see below) as well as a short (2 pages, maximum) writeup describing your approach. This writeup should:

- Describe any design decisions you made, including your choice of page eviction policy. Describe briefly your insertion and deletion methods in B+ tree. Describe your idea in solving the bonus exercise (if it applies).
- Discuss and justify any changes you made to the API.
- Describe any missing or incomplete elements of your code.
- Describe how long you spent on the assignment, and whether there was anything you found particularly difficult or confusing.

3 Collaboration

This assignment should be manageable for a single person. Larger groups beyond 2 are not allowed. Please indicate

This assignment should be manageable for a single person. Larger groups beyond 2 are not allowed. Please indicate clearly who you worked with, if anyone, on your team's writeup.

3.2. *Submitting your assignment*

Follow the same procedure as that in assignment 1. Please submit your writeup as a PDF or plain text file (.txt). Please do not submit a .doc or .docx. Please include pack up your write up with your code in the same zip file.

Make sure all your code is packaged so the instructions outlined in section 3.4 work.

3.3. *Submitting a bug*

Please submit bug reports to the instructor and cc TF. When you do, please try to include:

- A description of the bug.
- A .java file we can drop in the `test/simpledb` directory, compile, and run.
- A .txt file with the data that reproduces the bug. We should be able to convert it to a .dat file using `HeapFileEncoder`.

3.4 *Grading*

85% of your grade will be based on whether or not your code passes the system test suite we will run over it. These tests will be a superset of the tests we have provided. Before handing in your code, you should make sure it produces no errors (passes all of the tests) from both ant test and ant systemtest.

Important: before testing, we will replace your build.xml, BTreeFileEncoder.java, and the entire contents of the test/ directory with our version of these files! This means you cannot change the format of .dat files. You should therefore be careful changing our APIs. This also means you need to test whether your code compiles with our test programs. In other words, we will untar your tarball, replace the files mentioned above, compile it, and then grade it. It will look roughly like this:

```
[replace build.xml, BTreeFileEncoder.java, and test]

$ ant test

$ ant systemtest

[additional tests]
```

If any of these commands fail, we'll be unhappy, and, therefore, so will your grade.

An additional **15%** of your grade will be based on the quality of your writeup and our subjective evaluation of your code.

3.5 ***Late Submission Policy***

5 ıts will be deducted for late submission within one day.

15 points will be deducted for late submission within two days

15 points will be deducted for late submission within two days.

30 points will be deducted for late submission within three days.

We do not accept late submission after 3 days!!!

4. Example on Indexing

4.1. B+ tree

Suppose that we are using B+ tree on a file that contains records with the following search-key values:

(49, 24, 54, 31, 16, 76, 85, 68, 15, 40, 13, 35, 67, 18, 17, 29)

Load these values into a file in the given order using B+ tree. Assume that every page of the B+ tree index can store up to four (4) values (each with one pointer).

4.2. Extensible Hashing

Suppose that we are using extensible hashing on a file that contains records with the search-key above.

Load these values into a file in the given order using extensible hashing. Assume that every block (bucket) of the hash index can store up to four (4) values.

4.3. Linear Hashing

Suppose that we are using Linear hashing on a file that contains records with the search-key above.

Load these values into a file in the given order using linear hashing. Assume that every block (bucket) of the hash index can store up to four (4) values.