COMP 530 Assignment 4: B+-Tree

Due Monday, March 12th at 11:55 PM.

1. The Task

In this assignment, your task is to use all of the infrastructure that we have built so far in order to develop a B+-Tree (in `MyDB_BPlusTreeReaderWriter.h`). You are required to implement each of the public methods in this class. I have also included method headers (and some bodies) for a number of private methods that I use in my implementation. You can choose to implement these private methods or not; I only include them here in order to give you an idea of how you might go about implementing this project.

2. Implementation Overview

Your B+-Tree implementation will have two kinds of pages; internal pages and leaf pages. Use the `MyDB_PageReaderWriter.setType ()` method to set the type of the page; you can use the analogous method to get the type of the page.

Leaf pages store actual data records, just like any other file. They are probably not sorted (see below). But internal pages store special `MyDB_INRecord` records. They are probably sorted on the key. I have provided an implementation of these records for you. Basically, these records hold (key, ptr) pairs, rather than storing normal data.

One question you will need to answer is: what does an empty tree look like? Does it have two nodes, a root and a leaf? This is what my empty tree looks like. Or does it look differently?

2. The Public Methods

These are the public methods that you are required to implement.

2.1 `MyDB_BPlusTreeReaderWriter ()`

This accepts the name of the attribute to use as the sorting attribute for the B+-Tree (we only accept a single sorting attribute), the table corresponding to the B+-Tree, and the buffer manager used to serve up pages for the tree. Make sure that when you change the location of the root in the B+-Tree, you call `MyDB_Table.setRootLocation ()` in order to save it! The you can call `getRootLocation ()` to retrieve it.

2.2 `getRangeIteratorAlt ()`

This method accepts two attribute values (a lower bound and an upper bound) and returns an iterator that will list all of the records in this range, from low to high, inclusive. Note however that the records **do not** need to be returned in sorted order. This method is probably going to rely heavily on the helper method `discoverPages ()` below, which it can use to get a list of pages that it can build an iterator over. It will then create an iterator over those pages.

2.3 `getSortedRangeIteratorAlt ()`

This is almost exactly like the above method, except that the records must be returned in sorted order. Since records are typically not stored sorted on a page, this means that before a page is iterated over, its contents must be sorted. Fortunately, once you've implemented `discoverPages ()`, you can then call it and use the provided class `MyDB_PageListIteratorSelfSortingAlt` to perform the iteration (rather than the standard `MyDB_PageListIteratorAlt` class). This class will sort the contents of the pages as it iterates over them.

2.4 append ()

Just adds a new record to the B+-Tree. It is going to rely on the `append ()` private helper function (see below) on the root of the tree. Note that if the helper function indicates that a split has happened then this method needs to handle this by creating a new root that contains pointers to both the old root and the result of the split.

2.5 printTree ()

Prints the contents of the tree to the screen. For debugging. This method will never be tested, but you still have to implement it.

3. The Private Methods

Again, you are not required to implement any of these private methods, but I have included prototypes for them, since I include them in my own implementation. You may find each of them useful. They are as follows.

3.1 `discoverPages ()`

`discoverPages ()` accepts the identity of a page in the file corresponding to the B+-Tree, and then it recursively finds all leaf pages reachable from that initial page, which could possibly have and records that fall in a specified range (the range is specified by `MyDB_AttValPtr low, MyDB_AttValPtr high`). Any pages found are then returned to the caller by putting them in the parameter `list`. The return value from this method is a boolean indicating whether the page pointed to by `whichPage` was at the leaf level (this can be used to perform an optimization to avoid repeatedly descending to all of the leaf pages that are children of an internal node, causing extra I/Os, once you have found a single leaf page). `discoverPages ()` is used to implement iterators over the B+-Tree file, that efficiently look for all of the records with keys falling in a specified range.

3.2 `append ()`

`append ()` accepts the identity of a page `whichPage` in the file corresponding to the B+-Tree, as well as a record to append, and then it recursively finds the appropriate leaf node for the record and appends the record to that node. This is all done using the classical B+-Tree insertion algorithm. (In this sense, "append" is something of a misnomer, because we are actually doing a classic B+-Tree insert; I used the name "append" to be consistent with the regular file reader/writer). If the page `whichPage` splits due to the insertion (a split can occur at an internal node level or at a leaf node level) then the `append ()` method returns a new `MyDB_INRecordPtr` object that points to an appropriate internal node record. As described subsequently, internal node records are special records that live only in the internal nodes (pages) in the B+-Tree and they are different because they have only a key and a pointer, and no data. All leaf nodes in our B+-Tree (those whose type is `MyDB_PageType :: RegularPage`) will have only "regular" records. All others (directory or internal node pages) will hold these special internal node records.

3.3 `split ()`

This method accepts a page to split, and one more record to add onto the end of the page (which cannot fit). It then splits the contents of the page, incorporating the new record, leaving all of the records with big key values in place, and creating a new page with all of the small key values at the end of the file. A smart pointer to an internal node record whose key value and pointer is produced by this method as a return value (so that it can be inserted into the parent of the splitting node). This internal node record has been set up to point to the new page.

3.4 `getINRecord ()`

I have actually provided an implementation of this particular method which you can use. This returns a new internal node record, which you can then use to add to internal pages in the tree. Note that I have also provided you with a special record class for B+-Tree internal node records (this method returns a pointer to one of these, a `MyDB_INRecordPtr`).

3.5 `getKey ()`

I have also provided an implementation of this. This obtains the key attribute for an actual data record (that is, it obtains the value that the record is sorted on).

3.6 `buildComparator ()`

I have also provided an implementation of this particular method which you can use. What it does is to build a comparator (a lambda) for two records. Both of the records must either be `MyDB_INRecord` object for this tree, or they must be data records for this tree, or some combination. The constructed lambda, when invoked, will return true iff the key of the first record is less than the key of the second record.

4. Additional Details

4.1 The `MyDB_INRecord ()` Class

Internal node records are special records that live only in the internal nodes (pages) in the B+-Tree. They have only two attributes: a key (whose type matches the designated search key attribute for the B+-Tree as a whole) and a pointer, which is the integer identifier of a child page. The B+-Tree reader/writer can automatically create these for you (with the correct key type) as needed, via calls to `getINRecord ()`. There are a number getter and setter methods available on this special record type: `setKey ()` `getPtr ()` and so on (the first one sets the key attribute for the internal node record, and the second one gets the pointer associated with the internal node record). Note that on creation, the internal node record automatically has the largest possible key value inside of it. This is useful, because (as you will see) when I set up my almost-empty B+-Tree with just the root and one leaf, I put a single internal node record on the leaf, whose key value is correctly set to infinity. Note that the internal record class is a subclass of the regular record class.

4.2 The `sortInPlace ()` Method

Finally, I have aded a new method to the `MyDB_PageReaderWriter` class called `sortInPlace ()`. This method is like the `sort ()` method you used in the last assignment, except that it actually changes the contents of the page, so that after the call, the page is correctly sorted.

4.3 Notes on Maintaining a Sorted Order in the Tree

The way that we will implement this is to **not** maintain the records on a leaf node in the tree in sorted order. That is, just append the record to the end of the page. We'll do this to make our lives easier. Maintaining the page in sorted order is not too difficult (just a few extra lines), but it requires some thought to do efficiently and we won't bother in this assignment. That is why the sorted iterator automatically sorts each leaf page before returning its contents: the assumption is that the contents are not sorted (if you choose to have an implementation where you maintain a sorted order, you can change the sorted iterator so that it does not do the sort, and this will be more efficient, particularly on queries).

In contrast, we will maintain the records in internal nodes in sorted order. This is necessary to be able to perform inserts or to do anything with the tree. However (unless you really want to) there is no need to be fancy. Whenever you insert into an internal node, just call `sortInPlace ()` to sort the contents of the page. This isa bit inefficient, but the only time this will happen is after a leaf split, which requires a full sort of some descendent leaf page contents, anyway. So we won't worry about it.

4.4 Make Smaller Test Cases to Help You Debug!

As usual, I am supplying a tough test case. You will want to debug using much, much smaller data sets with much, much smaller pages sizes (1 KB makes sense). I have given a method that can recursively print out the entire contents and structure of the tree. This will be useful as you debug.

4.5 My Code, Your Code, and the Honor Code

As usual, I am giving out my solution to A3 in this assignment. Simply stated, you are not allowed to distribute my code to **anyone** who is not taking the class right now… **ever**! Doing this constitutes a violation of the honor code. I'm particularly concerned about my code falling into the hands of students who might take the class in the future. Thus, a hard and fast prohibition on distributing my source code.

4.6 Testing And Grading

Again, we're using `Qunit`. Again, I have written a few `Qunit` tests. When you build your project using the `SCons` build tool these tests will be complied and an executable will be produced.

Again, if you can make it through my test cases, it is likely that you've worked through the majority of the bugs in your buffer manager, and you've got most of the functionality done. However, you'll probably want to create a few additional test cases of your own—some simpler ones that you can use early on as you develop your code, and some additional, nasty ones just to make sure that everything is working.

When you turn in your code, and it's time for us to grade, we'll run the test cases I've suppled, as well as several others that won't be made public until after the turnin. You'll be graded on your code's success in passing all of the test cases, though we revere the right to browse through your code and take off additional points if it appears you are missing some functionality or have somehow hacked something in a sketchy sort of way. You won't be graded on style and comments. However, I strongly encourage you to take this opportunity to put your best software engineering practices to use.

5. Getting Started

The instructions this time around for getting started are much the same as the instructions last time around, so I won't reproduce them here.

6. Turnin

Simply zip up all of your source code and then turn it in on Canvas (make sure to archive into the zip format, and not some other archiving format. If you choose to use something else, we'll take off a few points!). Please name your archive A4.zip. Please do not change the original directory structure, except for perhaps adding some new files. The root should be a directory called A4, with two subdirectories Build and Main. And so on.

And remember, **to get *any* credit on A4, your code must compile and run on Clear**. That way, we have a common environment for grading and we don't have to spend time getting your code to compile.

Finally, and this is important: **include a README file in the root of your project with any important information**, including the names of the one or two people who worked on the project.

And also: if you work with a partner, **only turn in one copy of your source**. Otherwise, we'll possibly grade your submission twice, and end up getting quite annoyed.