COMP 530 Assignment 4: B+-Tree

Due Monday, March 14th 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. I have provided you with a B+-Tree implementation in that is perhaps 80% complete (in `MyDB_BPlusTreeReaderWriter.cc`). However, I have left out the "core" of my implementation: the guts of the two methods `discoverPages ()` and `append ()`. You are tasked with filling those in. At a high level, here is what those methods do:

1. `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.
2. `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.

2. Details, Details, Details

Again, I'm providing you with a lot of infrastructure that you can use to make this task easier, but the biggest difficulty associated with this assignment is going to be understanding that infrastructure. I describe some of this infrastructure now.

2.1 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.

2.2 The `buildComparator ()` Method

This method is important. 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.

2.3 The `split ()` Method

I have already implemented a `split ()` method for you. 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.

2.4 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.

2.5 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.

2.6 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.

2.7 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.

2.8 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.

2.9 Project Difficulty

All of the code that you need to write can be put into two functions whose contents I have gutted. I'd expect that you will write 25 to 75 lines of code to complete this assignment. But it'll take some time to figure that code out.

3. 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.

4. Turnin

Simply zip up all of your source code and then turn it in on OwlSpace (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.