CS349: B+ Tree Assignment
Scott D. Anderson

1 Introduction

In this assignment, you will implement a B+ tree in which leaf level pages contain complete data records (Alternative 1 for data entries, in terms of the textbook.) You must implement the full search and insert algorithms as discussed in class. In particular, your insert routine must be capable of dealing with overflows (at any level of the tree) by splitting pages; as per the algorithm discussed in class, you will not consider re-distribution. For this assignment, you don't have to deal with deletes.

You will be given `HFPage`. You should implement `SortedPage` as a class derived from `HFPage`, and it augments the `insertRecord` method of `HFPage` by storing records on the `HFPage` in sorted order by a specified *key* value. The key value must be included as the initial part of each record, to enable easy comparison of the key value of a new record with the key values of existing records on a page. I think the documentation available in the `HFPage` header file is sufficient to understand what operation each function performs.

You also need to implement two page-level classes, **BTIndexPage** and **BTLeafPage**, both of which are derived from SortedPage. These page classes are used to build the B+ tree index; you will write code to create, destroy, open and close a B+ tree index, and to find the `RID` a particular record. The `nextRecord` method should work, that we can scan forward from a given `RID`.

Please copy all the files from `dudley:~anderson/btree/` into your own local directory.

2 Design Overview

You should begin by (re-)reading the chapter *Tree Structured Indexing* of the textbook to get an overview of the B+ tree layer.

2.1 A Note on Keys for this Assignment

Keys will work the same way that they work in two important example programs: `heapfile-build` and `heapfile-scan`. Essentially, your code should allow you to create analogous programs called `btree-build` and `btree-scan`, as well as an additional program called `btree-search` that allows a user to provide keys and get records. The scanning program is nice, but the searching program is paramount.

Your `SortedPage` class, which augments the `insertRecord` method of `HFPage` by storing records on a page in sorted order according to a specified *key* value, assumes that the key value is included as the initial part of each record, to enable easy comparison of the key value of a new record with the key values of existing records on a page.

2.2 B+ Tree Page-Level Classes

These classes are summarized in Figure 1. Note again that you must not add any data members (instance variables) to `BTIndexPage` or `BTLeafPage`, because that would change their size. For further details about the individual methods in these classes, look at the header pages for the class.

- *HFPage:* This is the base class, you can look at hfpage.h to get more details.

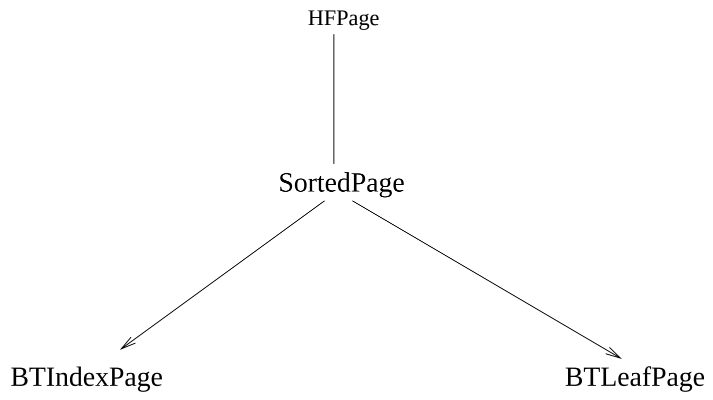

Figure 1: The C++ Classes used for the B+Tree pages

- *SortedPage:* This class is derived from the class HFPage. Its only function is to maintain records on a HFPage in a sorted order. Only the slot directory is re-arranged. The data records remain in the same positions on the page. This exploits the fact that the rids of index entries are not important: index entries (unlike data records) are never 'pointed to' directly, and are only accessed by searching the index page.
- *BTIndexPage:* This class is derived from SortedPage. It inserts records of the type ⟨*key, pageNo*⟩ on the SortedPage. The records are sorted by the key.

  Remember, though, that index pages need to have one more page pointer than key. What do we do? We're going to employ a trick: the `PrevPage` pointer is not needed on an index page, so we can use that pointer to point to the leftmost child of this index page.
- *BTLeafPage:* This class is derived from SortedPage. It inserts records of the type ⟨*key, data*⟩ on the SortedPage. The records are sorted by the key. Further, leaf pages must be maintained in a doubly-linked list.

2.3 Other B+ Tree Classes

We will assume here that everyone understands the concept of B+ trees, and the basic algorithms, and concentrate on explaining the design of the C++ classes that you will implement.

A BTreeFile will contain a header page and a number of BTIndexPages and BTLeafPages. The header page is used to hold information about the tree as a whole, such as the page id of the root page, the type of the search key, the length of the key field(s) (which has a fixed maximum size in this assignment), etc. When a B+ tree index is opened, you should read the header page first, and keep it pinned until the file is closed. Given the name of the B+ tree index file, how can you locate the header page? The DB class has a method

```
Status add_file_entry(const char* fname,
                      PageId_t start_page_num,
                      int key_type,
                      int key_length);
```

that lets you register this information when a file `fname` is created. There are methods for deleting and reading these file entries as well, which can be used when the file is destroyed or opened. The header page contains the page id of the root of the tree, and every other page in the tree is accessed

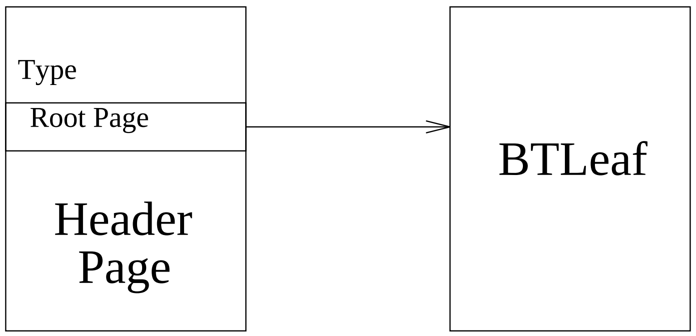

Figure 2: Layout of a BTree with one BTLeafPage

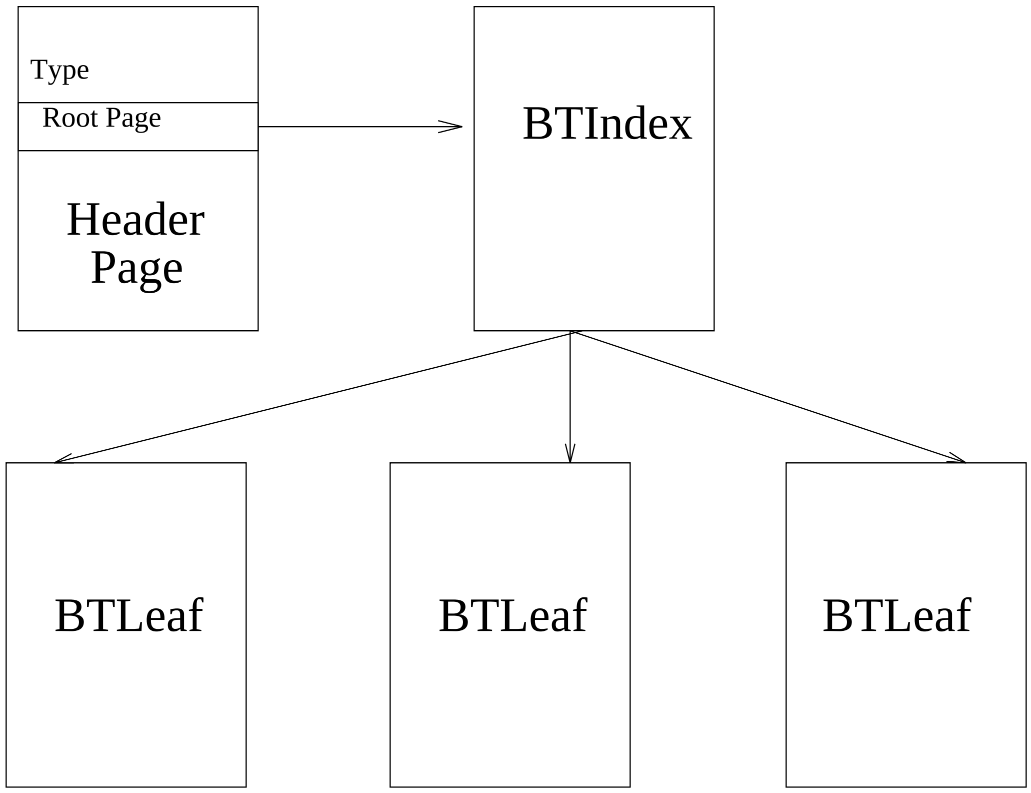

Figure 3: Layout of a BTree with more than one BTLeafPage

through the root page. (If you can see a way to avoid having a header page, that would be very good. If you can see why a header page is useful, that would also be good.)

Figure 2 shows what a BTreeFile with only one BTLeafPage looks like; the single leaf page is also the root. Note that there is *no* BTIndexPage in this case. Figure 3 shows a tree with a few BTLeafPages, and this can easily be extended to contain multiple levels of BTIndexPages as well.

2.3.1 BTreeFile

The main class to be implemented for this assignment is `BTreeFile`, which is analogous to `HeapFile`.

The methods to be implemented include:

BTreeFile::BTreeFile There are two constructors for BTreeFile: one that will only open an index, and another that will create a new index on disk, with a given type and key size. These are analogous to the ones for `HeapFile`.

BTreeFile::insertRecord This method takes two arguments: (a pointer to) a record and its length. Unlike `HeapFile`, it won't set a RID. Why?

If a page overflows (i.e., no space for the new entry), you should split the page. You may have to insert additional entries of the form ⟨key, id of child page⟩ into the higher level index pages as part of a split. Note that this could recursively go all the way up to the root, possibly resulting in a split of the root node of the B+ tree. However, the user will be unaware of all this complexity.

BTreeFile::findRecord This method takes one input argument — (a pointer to) a key, and sets two output arguments: a pointer to a record and its length. This should search down the btree until it finds the appropriate record.

3 Testing

You need not only to implement the solution, but you need to convince me that your code is indeed a solution. You will do this by writing test programs that puts your code through its paces. These are primarily `btree-build`, `btree-search`, and `btree-scan`. However, any other important cases that those don't cover should be covered by a `btree-test` program. Think of everything you can, and test it.

You must document your testing, so that I understand it. You can't just turn in a bunch of code (I've not been a good example in this). You have to describe (in English) the various test cases and what the correct result is, and then show (with print statements in your code and so forth), that your code performs correctly.

4 What to Turn In, and When

You should put your working code on Dudley and email me its location, so that I can copy it to my own directory to run and read. The assignment is due by midnight on Wednesday, March 14th.