UNIVERSITY OF WISCONSIN–MADISON
COMPUTER SCIENCES DEPARTMENT

CS 564: DATABASE MANAGEMENT SYSTEMS

Assignment 3: B+ Trees
Due: Wednesday, November 6, 1996

Instructors: Jeff Naughton and Raghu Ramakrishnan

1 Introduction

In this assignment, you will implement a B+ tree in which leaf level pages contain entries of the form ⟨*key, rid of a data record*⟩ (Alternative 2 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 can deal with deletes by simply marking the corresponding leaf entry as 'deleted'; you do not have to implement merging.

You will be given HFPage and SortedPage. SortedPage is 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. The documentation available in the header files is sufficient to understand what operation each function performs.

You 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 open scans that return all data entries (from the leaf pages) which satisfy some range selection on the keys.

You will carry out this assignment in teams of two, continuing with the same partner as for the previous assignment. If you want to change partners, you must let us know first.

2 Getting Started

Please copy all the files from ~cs564-1/fall96/proj3/src into your own local directory. The files are:

- *Makefile:* A sample Makefile for you to compile your project. You will have to set up any dependencies by editing this file. You can also design your own Makefile.
- *Partial templates for btindex_page.h, btleaf_page.h, btfile.h, and btreefilescan.h :* You should complete all these .h files (if needed) and also implement the methods in the corresponding .C files.
- *main.C*,*btree_driver.C*,*sample_keys*: B+ tree test driver program.

You can find other useful include files *bt.h*, *hfpage.h*, *sorted_page.h*, *index.h*,*test_driver.h*, *btree_driver.h*, *minirel.h* and *new_error.h* in ~cs564-1/fall96/proj3/include.

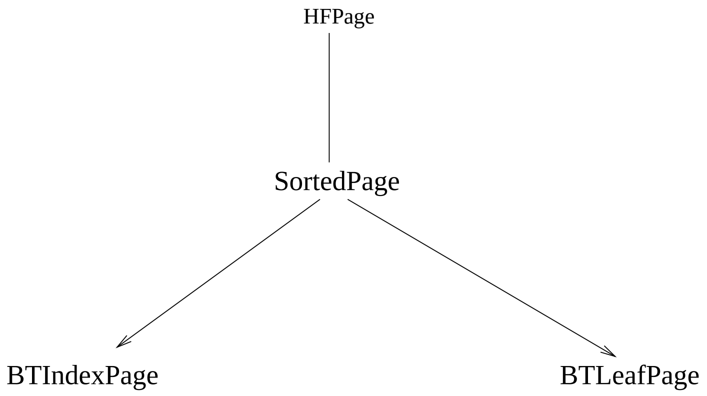

Figure 1: The C++ Classes used for the B+Tree pages

3 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. There is also information about the B+ tree layer in the HTML documentation.

3.1 A Note on Keys for this Assignment

You should note that key values are passed to functions using `void *` pointers (pointing to the key values). The contents of a key should be interpreted using the `AttrType` variable. The key can be either a string(attrString) or an integer(attrInteger), as per the definition of AttrType in `minirel.h`. We just implement these two kinds of keys in this assignment. If the key is a string, it has a fixed maximum length, MAX_KEY_SIZE1, defined in `bt.h`.

Although the specifications for some methods (e.g., the constructor of BTreeFile) suggest that keys can be of (the more general enumerated) type AttrType, you can return an error message if the keys are not of type attrString or attrInteger.

The 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.

3.2 B+ Tree Page-Level Classes

These classes are summarized in Figure 1. Note again that you must not add any private data members to *BTIndexPage* or *BTLeafPage*.

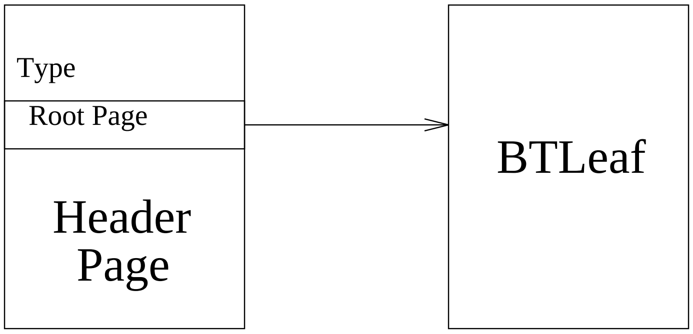

Figure 2: Layout of a BTree with one BTLeafPage

- *HFPage:* This is the base class, you can look at hfpage.h to get more details.
- *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.
- *BTLeafPage:* This class is derived from SortedPage. It inserts records of the type ⟨*key, dataRid*⟩ on the SortedPage. *dataRid* is the rid of the data record. The records are sorted by the key. Further, leaf pages must be maintained in a doubly-linked list.

For further details about the individual methods in these classes, look at the header pages for the class.

3.3 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

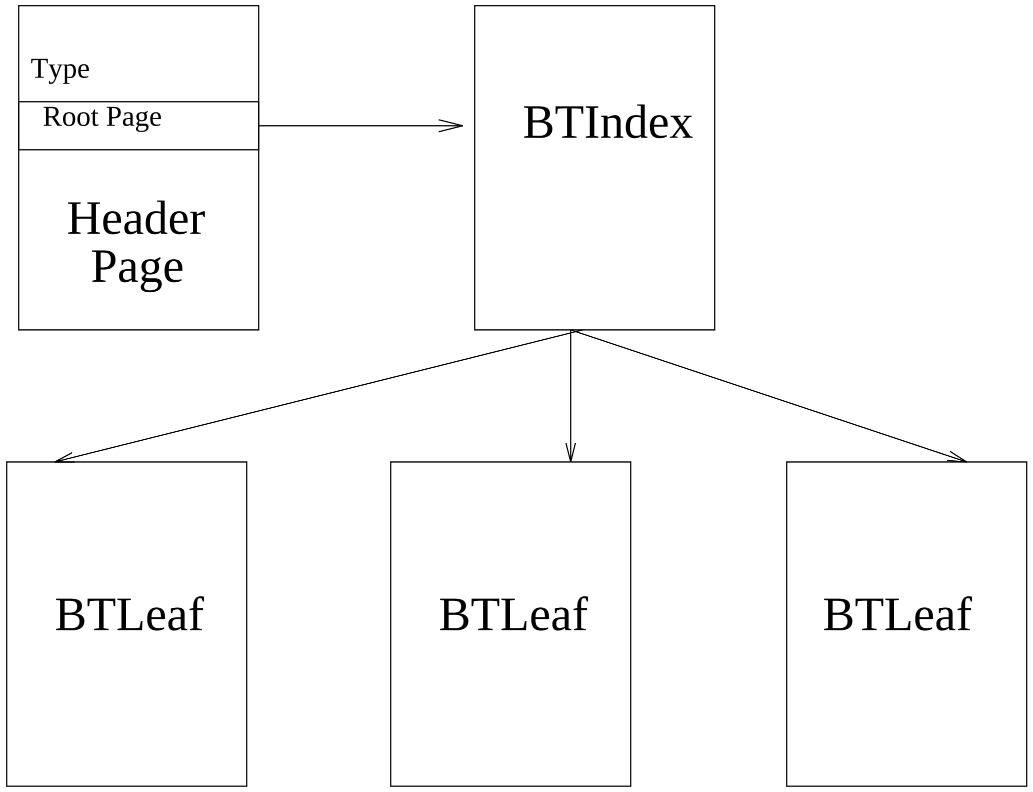

Figure 3: Layout of a BTree with more than one BTLeafPage

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 header_page_num);
```

that lets you register this information when a file fname is created. There are methods for deleting and reading these 'file entries' (⟨file name, header page⟩ pairs) 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 through the root page.

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.

3.3.1 IndexFile and IndexFileScan

A BTree is one particular type of index. There are other types, for example a Hash index. However, all index types have some basic functionality in common. We've taken this basic index functionality and created a *virtual base class* called IndexFile. You won't write any code for IndexFile. However, any class derived from an IndexFile should support `IndexFile()`, `Delete()`, and `insert()`. (IndexFile and IndexFileScan are defined in `~cs564-1/fall96/proj3/include/index.h`).

Likewise, an IndexFileScan is a virtual base class that contains the basic functionality all index file scans should support.

3.3.2 BTreeFile

The main class to be implemented for this assignment is BTreeFile. BTreeFile is a derived class of the IndexFile class, which means a BTreeFile is a kind of IndexFile. However, since IndexFile is a virtual base class *all* of the methods associated with IndexFile must be implemented for BTreeFile. You should have copied `btfile.h` into your directory, as per the instructions in Section 2.

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. Observe that the key type is passed as a value of **AttrType** . For this assignment, you only need to handle keys of type attrString and attrInteger. If there is a call with a key whose type is not one of the two, return an error.

BTreeFile::insert The `BTreeFile::insert` method takes two arguments: (a pointer to) a *key* and the *rid* of a data record. The data entry to be inserted—i.e., a 'record' in the leaf pages of the BTreeFile —consists of the pair ⟨key, rid of data record⟩.

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.

BTreeFile::Delete The `BTreeFile::Delete` routine simply removes the entry from the appropriate BTLeafPage. You are not required to implement redistribution or page merging when the number of entries falls below threshold.

3.3.3 BTreeFileScan

Finally, you will implement scans that return data entries from the leaf pages of the tree. You should create the scan through a member of BTreeFile, so that you can report an error if a BTreeFile is closed before a scan is completed.

Note that BTreeFileScans should support several kinds of range selections. These ranges are described in `btfile.h`.

4 A Note on the Due Date

This project should take you about two weeks to finish. Unfortunately, that would make this due around October 30, which is the date of our midterm exam. Since your instructors are such nice people, they have decided not to have you finishing your assignment at the same time you are preparing for the midterm.

However, we would strongly urge you to begin the assignment as if it were due in two weeks. Then you can take some time out to prepare for the midterm, and will be in good shape to finish in the week following that. If you wait until after the midterm to start, you will be unlikely to finish.

5 Extra Credit

Extra credit of up to 15 percentage points is available. However, the maximum number of points that you can score on this assignment is 110. (So the 15 extra points could be used partially to offset points you lose elsewhere on the assignment.) The main motivation for trying these additional challenges should be the opportunity to write more complete software and understand some of the finer points, rather than to score more points. *Do not start on this until you have completed the basic assignment!*

The tasks are listed below with the extra percentage points they carry in parentheses:

(1) *5 points:* Support duplicate records by allowing records with the same key to exist on more than one page. (You should not use any overflow pages!)

(2) *10 points:* Implement node redistribution and merging during deletion.