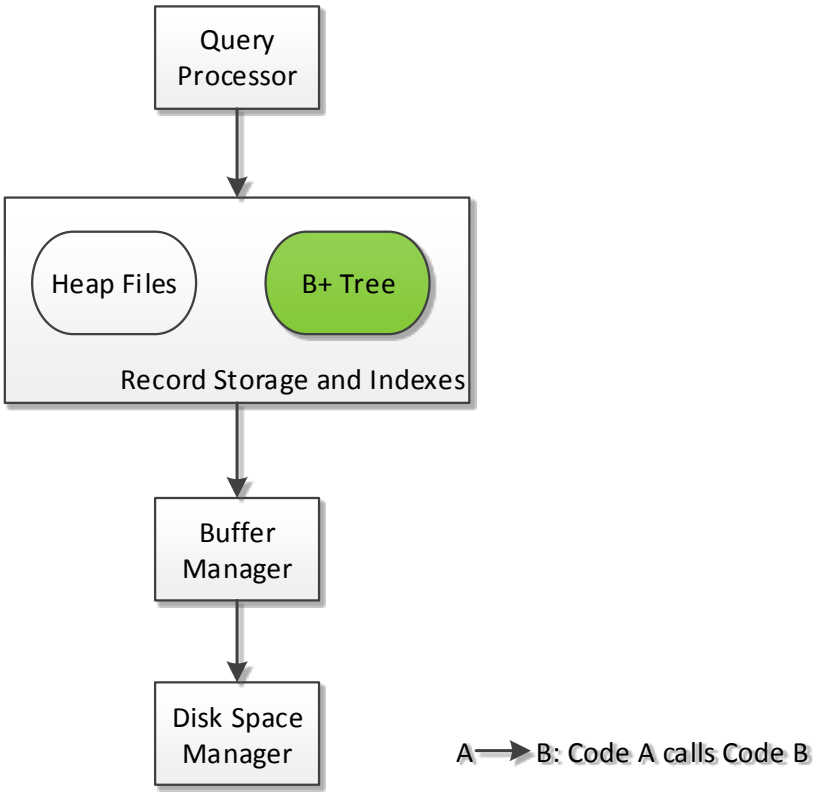
CS4321 Fall 2014
Project 4: B+ Tree

In this assignment, you will implement a B+ tree. Please read the whole assignment carefully and review the relevant material in the textbook before you start implementing. This assignment is due on **Nov 17 at 23:59 pm** via CMS. It is worth a total of 100 points and counts for 35% of your grade. Because this assignment is significantly more complex than previous assignments, there will also be a **midpoint check** on **Nov 5 at 23:59 pm**; details can be found at the very end of the handout.



Important Advice

This assignment is ***significantly more difficult*** than the previous assignments. It requires you to implement a rather intricate data structure.

- Start early. Even if you were able to finish the previous assignments in a few days, you should not plan on finishing the B+ Tree the week before it is due.
- Do this assignment in stages. Start by getting B+ Tree to work on a single leaf, and work from there.
- Once you have your code working for a single leaf, you should take a snapshot of the code and set it aside so that you can submit it to us for the midpoint check. Please see the *Midpoint Check* section at the end of the document for details.

Good luck!

Introduction

In this assignment, you will implement a B+ Tree index in MINIBASE. As you learned in class, the B+ Tree provides efficient ordered access to a large number of records, and is the most common index structure in commercial databases.

Read the textbook carefully and familiarize yourself with the B+ Tree indexing structure and algorithm.

Review of MINIBASE components

In the previous two assignments, you were introduced to the Database (DB), Page, HeapPage, HeapFile and Buffer Manager (BufMgr) abstractions. As you may recall, the DB abstraction is responsible for maintaining file entries in the catalog, allocation/deallocation of pages and performing I/Os on the pages. The Page abstraction is just a dummy class of size MAX_SPACE, and HeapPage is a special type of Page. Finally, the BufMgr is responsible for maintaining "frequently" used pages in memory. In this assignment you will have to build a B+ Tree on top of these abstractions.

type field in HeapPage

The class HeapPage has a member called type, which we did not emphasize before. The type of a page is used in this assignment to indicate whether the page is a leaf page or an index page of the B+ Tree.

SortedPage

A SortedPage is a special type of HeapPage that maintains records in a page in increasing sorted order based on a key. Insertion sort is used to insert the records, so the records in a HeapPage may be moved whenever insertion occurs. This may change the RecordID of the records. For this reason, the RecordID of records in a SortedPage should never be exposed to users.

Overview of B+ tree

Logically, a B+ Tree stores a collection of key-value pairs **(key, rid)**, where **key** is a search key, and **rid** is the record id of the record being indexed. (It is an index based on what is described as Alternative 2 in the textbook.). For this assignment, your B+ Tree will support one search key type: **NULL-terminated character strings**. Thus the length of a key will be one larger than the number of characters in the string, and the last byte will always be '\0'. Keys are sorted lexicographically, so that "aa" < "ab" < "b". For those of you who don't have much experience working with `NULL`-terminated C-style strings, you may find the methods in the header `<cstring>` to be useful. For instance, you can use `strcmp` to find the relative order of two strings.

A B+ tree index is stored in MINIBASE as a database file. The class IndexFile is an abstract class for indices. BTreeFile is a subclass of IndexFile, and implements the B+ tree in MINIBASE.

The class BTreeFile provides methods for inserting, deleting, and searching the tree. The BTreeFile class is responsible for maintaining the integrity of the tree. (i.e. when an insertion is made, it has to ensure that the new record is inserted in the correct place. If the current node does not have enough space to hold the record, it should perform node split.)

The nodes in a B+ tree can be divided into two categories: internal index nodes and leaf nodes. The leaf nodes store the actual **(key, rid)** pairs, while the internal nodes store **(key, pid)** pairs, where **key** and **pid** denote the key values and the page identifiers (pageIDs) of their children.

The leaf and index nodes in a B+ tree are implemented by the classes BTLeafPage and BTIndexPage, respectively. They are both subclasses of SortedPage and are responsible for inserting into and deleting from leaf node and index node, respectively. These two classes are not responsible for the integrity of the tree.

In MINIBASE, each B+ tree has a **header page**. 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 (maximum) length of the key field(s). In this assignment, you will implement a B+ tree that indexes over a single string attribute. The header page will therefore only store the page id of the root page. When a B+ tree index is opened, you should read the header page first, and keep it pinned until the file is closed. 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.

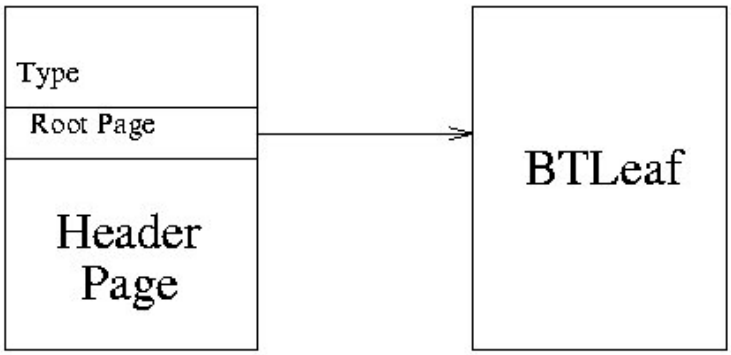


Figure 1: Layout of a BTree with one BTLeafPage

Figure 1 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.

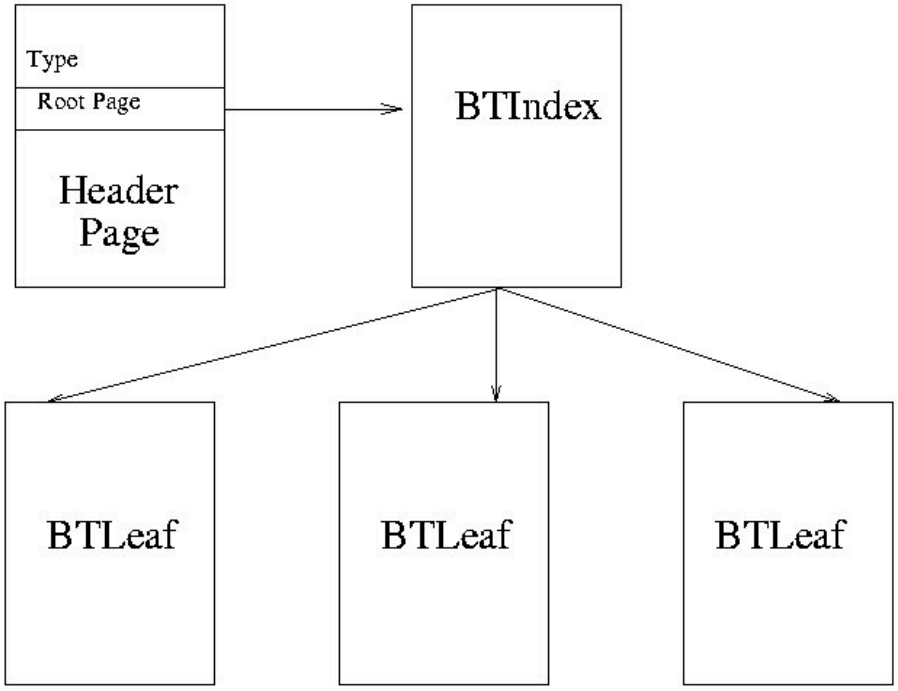


Figure 2: Layout of a BTree with more than one BTLeafPage

Figure 2 shows a tree with a few BTLeafPages, and this can easily be extended to contain multiple levels of BTIndexPages as well.

For this assignment, the implementation of BTLeafPage and BTIndexPage are provided for you. We recommend that you study the source code for these two classes carefully. Their implementation can be found in btleaf.cpp and btindex.cpp. You are free to add any methods to these classes that can help with your assignment (try to keep the methods private wherever possible). **Do not add any member variables to these classes, since they have to map carefully onto the structure of a page**.

BTLeafPage

Each leaf data item that we insert or delete from a BTreeFile is a tuple (key, record id). These tuples are stored as records of type KeyDataEntry in BTLeafPage. To facilitate a scan through the leaf nodes, the leaf nodes are linked together as a doubly linked list. These "pointers" in the link list are actually PageID of the previous page and next page. The two members corresponding to these pointers are called nextPage and prevPage, which should look familiar to you from the HeapPage assignment. They can be set and retrieved with SetNextPage(), SetPrevPage(), GetNextPage() and GetPrevPage().

Other functions provided for BTLeafPage are Insert and Delete, which, as the names suggest, insert and delete a KeyDataEntry to/from the page. GetNext(), GetFirst() and GetCurrent() are also provided to scan through the records in a BTLeafPage.

BTIndexPage

The BTIndexPage contains a sequence <pid_1, key_1, pid_2, key_2, ..., key_k, pid_{k+1}> (the semantics of this sequence are as described in the textbook). The prevPage member in BTIndexPage (also referred to as the leftlink or the leftmost child page) is used to store pid_1 while (key_i, pid_{i+1}) are stored as records of type IndexEntry in the BTIndexPage.

Just like in BTLeafPage, Insert(), Delete(), GetFirst() and GetNext() are provided for this class. It contains some other methods that can be very helpful to your implementation. Read btindex.h and btindex.cpp for their description.

Scanning a B+ Tree

We can also open a scan on a B+ Tree, and retrieve all records within a range of key values. The BTreeFileScan class is a data structure to keep track of the current cursor in the scan. It provides a GetNext() method for retrieving the next record in the BTreeFile. We can also delete the current record at the cursor while we are scanning, using the DeleteCurrent() method.

Assignment requirements

In this assignment, you are required to implement two classes: BTreeFile and BTFileScan, link them with the rest of the code, and make sure that all test programs run

successfully. For simplicity, your implementation does not have to deal with duplicate key values. The details of the functions that you have to implement are given below.

BTreeFile

BTreeFile::BTreeFile, BTreeFile::~BTreeFile

The constructor for the BTreeFile takes in a filename, and checks if a file with that name already exists in the database. If the file exists, we "open" the file. Otherwise we create a new file with that name.

The destructor of BTreeFile just "closes" the index. In a correct implementation of B+ tree, only the header page will stay pinned at this point. Therefore only the header page is unpinned here.

The implementation is provided, and you do not need to modify it. You are recommended to take a look at them.

BTreeFile::DestroyFile

This method deletes the entire index file from the database. You need to free all pages allocated for this index file.

BTreeFile::Insert

This method inserts a pair (key, rid) into the B+ tree. The actual pair (key, rid) is inserted into a leaf node. However, this insertion may cause one or more (key, pid) pairs to be inserted into B+ tree index nodes. You should always check to see if the current node has enough space before you insert. If there is not enough space (node overflow), you have to split the current node by creating a new node, and copying some of the data over from the current node to the new node. 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. Splitting will cause a new entry to be added in the parent node. In this assignment, you will implement node splitting for handling overflow on inserts. You do not need to implement sibling redistribution on inserts. Note that since we implement string keys, records in B+ tree may have different sizes. Therefore, instead of trying to move half of the records to a new page during sibling redistribution as was suggested by the text book, what you should really do is to try to minimize the difference in space consumption between the two sibling nodes in the redistribution process. Note that with variable length records, in theory it is possible that we cannot split a node even when it overflows (e.g. there is only one record currently in the node, which almost fills up the entire space). However, you do not have to deal with this case, and this case will not occur in the test cases for this assignment.

Splitting of the root node should be considered separately, since if we have a new root, we need to update the B+ tree header page to reflect the changes. Also, if you split a leaf node, be careful in maintaining the linked list of B+ tree leaf nodes.

Due to the complexity of this function, we recommend that you write separate functions for different cases. For example, it is a good idea to write a function to insert into a leaf node, and a function to insert into an index node. The following shows some simplified code fragment that may be helpful.

Checking if there is enough space to insert a record into a leaf node :

```
if (page->AvailableSpace() ...
```

Inserting a pair (key, rid) into a leaf node with page ID pid can be done with the following code :

```
BTLeafPage *page;
RecordID outRid;
MINIBASE_BM->PinPage(pid, (Page *&)page);
page->Insert(key, rid, outRid);
MINIBASE_BM->UnpinPage(pid, DIRTY);
```

BTreeFile::Delete

This method deletes an entry (key, rid) from a leaf node. Deletion from a leaf node may cause one or more entries in the index node to be deleted. You should always check if a node underflows (less than 50% full) after deletion. If so, you should perform sibling redistribution, and if that does not work, merge sibling nodes. (read and implement the algorithm in the textbook). Note that with variable length records, in theory it is possible that we can neither redistribute nor merge when a node underflows (e.g. the sibling nodes of this node each have only one record, which almost fills up the entire space). However, you do not have to deal with this case, and this case will not occur in the test cases for this assignment.

For implementing redistribution in this assignment, it is sufficient for you to pick one of the two siblings and try to redistribute. For example, you can always pick the left sibling to perform redistribution; if that fails, try merging; if that fails still, do nothing.

You should consider different scenarios separately (perhaps write separate functions for them). For example, deletion from a leaf node and from an index node should be considered separately. Deletion from the root should also be consider separately (what happens if the root becomes empty after some deletion, but there is still a child node?)

The following code fragment may be helpful :

Checking if a node is half full :

```
if (page->AvailableSpace() > HEAPPAGE_DATA_SIZE/2)
{
   // check if redistribution or merge can occur
}
```

BTreeFile::Search, BTreeFile::_Search, BTreeFile::_SearchIndex

These methods together implement the search logic of B+ tree. The implementation is provided, and you do not need to modify it. However, you are recommended to take a look at them.

BTreeFile::OpenScan

This method should create a new BTreeFileScan object and initialize it based on the search range specified. It is useful to find out which leaf node the first record to scan is in.

BTreeFile::DumpStatistics

This function will collect statistics to reflect the performance of your B+ Tree implementation. The following statistics will be printed

1. Total number of nodes in the tree
2. Total number of data entries in the tree
3. Total number of index entries in the tree
4. Average, minimum and maximum fill factor (used space/total space) of leaf nodes and index nodes.
5. Height of tree

These statistics should serve you in understanding whether the tree is in correct shape.

BTreeFile::PrintTree, BTreeFile::PrintWhole

These are helper functions that should help you debug, by showing the tree contents. The implementation has been provided, and you do not need to modify it. However, you are recommended to take a look at them.

BTreeFileScan

First, note that BTreeFileScan only has a default constructor, which you do not need to implement. Also, a BTreeFileScan object will only be created inside BTreeFile::OpenScan.

BTreeFileScan::~BTreeFileScan

The destructor of BTreeFileScan should clean up the necessary things (such as unpinning any pinned pages).

BTreeFileScan::GetNext

GetNext returns the next record in the B+ Tree in increasing order of key. Basically, GetNext traverses through the link list of leaf nodes, and returns all the records in them that are within the scan range, one at a time.

Documentation

Since this is a complex assignment, you should also submit a document describing the code that you have written. This document should be about 1-3 pages long, and include a description of your implementation strategy and any problems you encountered. You should also include the following:

- Any assumptions you made to get the code to work.
- A description of any test cases you wrote and why they are useful. You should also include the code of these test cases in the project.
- Any known bugs. If your code fails a test case, explain the symptoms and what you think the problem might be.
- A description of your performance measurements, if you elected to do the extra credit.

Please include this document in your zip directory in PDF format.

Getting Started

The zip file for the project can be downloaded using the course management system. Once you unzip the file, the directory contains the following files (among others):

btfile.cpp, btfile.h	code skeleton for the BTreeFile class
btleaf.cpp, btleaf.h	code for BTLeafPage class
btindex.cpp, btindex.h	code for the BTIndexPage class
btfilescan.cpp, btfilescan.h	code skeleton for BTreeFileScan class
sortedpage.cpp, sortedpage.h	code for SortedPage class
bt.h	general declaration of types
btreetest.cpp, btreetest.h	code for manual test interface to the B+ tree
btreeDriver.cpp, btreeDriver.h	the B+ tree test cases
main.cpp	contains main(), runs tests

You should write most of your code in **BTreeFile** (.h and .cpp) and **BTreeFileScan** (.h and .cpp). You can add any member variables and methods to these two classes.

Again, you can also add new methods to the BTLeafPage (.h and .cpp) and BTIndexPage (.h and .cpp) classes (keep these methods private wherever possible). You **should not** modify any of the other existing methods in BTLeafPage and BTIndexPage. You also **should not** add any member variables to these classes, since they have to map carefully onto the structure of a page.

Manual Test Interface

You are provided with a component BTreeTest with which you can test B+ tree interactively via standard input. Below are the commands accepted.

insert <low> <high>	All the records from [low, high) are inserted sequentially
scan <low> <high>	Scan for <high> <low> records.
delete <low> <high>	Delete records between <high> <low>
deletescan <low> <high>	Scan and deleteCurrent records between <high> <low>.
print	Print B+ tree.
stats	Show stats
quit (or EOF)	Termination

Note that <low> and <high> denote integer values. However, they will be converted to strings before being inserted into the B+ tree.

Note that this manual test interface is only provided for your convenience in debugging and testing your code. You should write your code towards passing the test cases in BTreeDriver.

Extra Credit

You can get up to 5 points of extra credit on this assignment by performing a simple performance evaluation of your B+ Tree. You should produce a graph with the number of records on the x-axis and time on the y-axis. You should measure (separately) the time to build the tree and the time to perform some number of queries/scans. We are leaving the details up to you, but you should describe them clearly in your documentation. To measure the time a block of code takes, you can use the clock function:

```
#include <ctime>
...
clock_t initTime = clock();
```

```
// Code you want to time.
clock_t endTime = clock();
timeInMilliseconds = (endTime - initTime)
                                * (1000.0 / CLOCKS_PER_SEC);
```

You could also discuss some of the statistics collected from BTreeFile::DumpStatistics.

What to Turn In

You should create a zip file containing the Visual Studio project and all source/header files as well as the project documentation. To keep the file size small. Please delete the .sdf file in the root file and the entire ipch directory before zipping up the files. You should also clean the solution (Build->Clean Solution) and turn off breakpoints before submitting.

Grading and Late Policy: The final submission is worth 90% of your score for the assignment. The late policy for this project is the same as for previous assignments: barring extensions provided by the course staff, submitting the project one day late will result in a 25% deduction from your score for the final submission, and submitting the project two days late will result in a 50% deduction. We will not accept submissions that are more than two days late.

Midpoint Check

The goal of the midpoint check is to encourage you to start early on the project. By the deadline for the midpoint check on Nov 5th, you should have a BTree implementation that can handle insertions/deletions and scans over a tree that contains a single leaf node. However, your implementation will not be required to handle operations that involved index pages at that point in the project. The code that you submit for the midpoint check should pass the test case 0-2 that we have provided to you.

When you submit your code for the midpoint check, you should follow the code cleanup procedure described in the *What to Turn In* section of this document. **In the zip directory that contains your code, you should include a short PDF or text file that describes any known bugs in your implementation**. (A longer writeup is *not* required for the midpoint check.)

Grading and Late Policy: The midpoint check is worth 10% of your score for the assignment. Submitting the project one day late will result in a 25% deduction from your score on the midpoint check, and submitting the project two days late will result in a 50% deduction. We will not accept submissions that are more than two days late.