15-415: Database Applications

Project 2: Extending the Functionality of a Basic B+ Tree

School of Computer Science
Carnegie Mellon University, Qatar
Spring 2014

Assigned date: February 27, 2014
Due date: March 15, 2014 by 11:59PM

I. Project Objectives:

DBMSs heavily use and rely on B^+ trees to speed-up operations like equality searches, range searches, grouping, and ordering, among others. This assignment is designed to make you familiar with the implementation of a B^+ tree data structure. You will extend a basic B^+ tree implementation by incorporating additional functionalities described later in the document.

II. The B^+ Tree Package:

You will be provided with a B^+ Tree Package (posted on the course web-page) which contains an implementation of a B^+ tree with Alternative 3. This package will be the basis of this project. In the forthcoming paragraphs, we describe the package content and how to compile it.

a. A Brief Primer :

The package encompasses four folders alongside a README and a make files. The directories and their contents are as follows:

▪ DOC:	useful documentation of the code.
▪ SRC:	source code.
▪ Datafiles:	sample data to load into the tree.
▪ Tests:	sample tests and their solutions.

The README file contains information about using the provided makefile for compiling, cleaning, and testing the code, in addition to a description of the commands supported by the basic B+ tree.

b. Compilation :

To compile the source code, *unzip* the package and type *make*. This compiles the code and creates an executable called ***main*** in addition to four files, namely B-TREE_FILE, POSTINGSFILE, TEXTFILE, and parms. All information in the B+ tree is stored in these files as follows:

▪ B-TREE_FILE:	stores the tree nodes including keys and pointers
▪ POSTINSFILE:	stores the list of pointers to data records
▪ TEXTFILE:	stores the actual data records
▪ Parms:	stores configuration parameters for the tree

To allow the program to access a tree and its data records across multiple executions, do not delete these files and ensure that they reside in the same directory as the executable *main*. Conversely, delete those files when you wish to create a new tree.

c. High-Level View :

To this end, **Figure 1** presents a bird's eye view of the structure of our B$^+$ tree with Alternative 3. Each non-leaf node stores a set of keys and pointers to other nodes (as illustrated by arrow 1). In a leaf node, however, a pointer associated with a particular key refers to a list of pointers called the *postings list* (as illustrated by arrow 2). Each element or *posting* in the postings list is a pointer referring to a text document that contains the key (as illustrated by arrow 3).

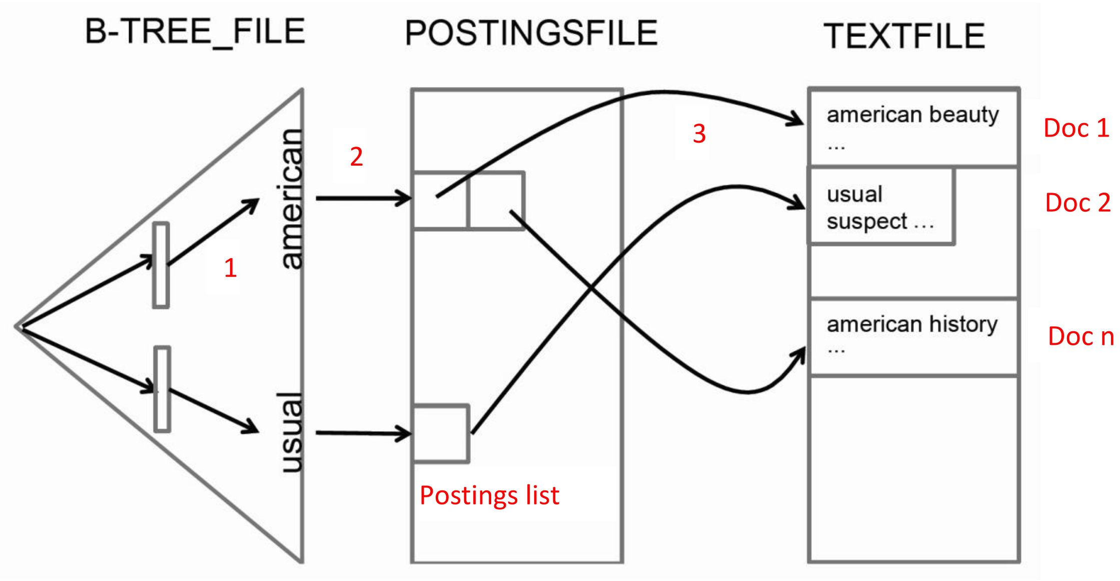

Figure 1: A High-Level View of the Structure of the B$^+$ tree

Each node or a posting is a C struct whose binary representations are stored in -TREE_FILE and POSTINGSFILE, respectively. TEXTFILE is a file of concatenated text documents. When *main* is executed and the aforementioned files are present, the program loads all the data in those files into the tree. Otherwise, the program simply creates a new empty tree.

d. Functionalities :

As mentioned above, our B+ tree implementation stores words as keys. Since the tree enables us to retrieve documents given words (as opposed to finding words in given documents), the tree is referred to as an *inverted index*.

When executed, *main* loops indefinitely, accepting and processing commands from the user. **Table 1** summarizes all the supported commands and their corresponding outputs.

COMMAND	**OUTPUT**
c	Prints all the keys that are present in the tree, in as**c**ending lexicographical order.
i <doc>	Parses the text in <doc> which is a text file, and **i**nserts the uncommon words (i.e., words not present in “comwords.h”) into the B+ tree. More specifically, the uncommon words of <doc> make the “keys” of the B+ tree, and the value for all the keys is set to the text of <doc>.
p <num>	Prints the keys in a particular **p**age of the B+ tree where <num> is the page number. It also prints some statistics about the page such as the number of bytes occupied, the number of keys in the page, etc.
s <key>	**s**earches the tree for <key> where <key> is a single word. If found, the program prints “Found the key!”. If not, it prints “Key not found!”.
S <key>	**S**earches the tree for <key>. If found, the program prints the text of all documents in which the key is present, also known as the posting list of <key>. If not, it prints “Key not found!”.
T	Prints the tree in a pre**T**ty format! If the tree is empty, it prints “Tree empty!” instead.
x	E**x**its the program.

Table 1: Supported Commands

To see a demo type *make demo*. The demo inserts some text documents into the tree (using the command *i*) and then searches for the word "american" (using the command *S*). As described in Table 1, the command *S* will print the text of each document containing the word "american".

III. Additional Functionalities:

In this project, you will implement two additional commands as described in **Table 2**.

COMMAND	**OUTPUT**
f <key1> <keys2>	Print in *alphabetical order* (**f**orward) the distinct keys that are in the range defined by <key1> and <key2> (including the bounds). If <key1> and <key2> are not in alphabetical order, print "Invalid key order!". If no documents have keys within the given range, print "Keys in the given range not found!".
b <key1> <keys2>	Print in *reverse alphabetical order* (**b**ackward) the distinct keys that are in the range defined by <key1> and <key2> (including the bounds). If <key1> and <key2> are not in alphabetical order, print "Invalid key order!". If no documents have keys within the given range, print "Keys in the given range not found!".

Table 2: Operations to Implement

Implement the commands shown in **Table 2** in the files *keysRange_ToImplement.c* and *keysRangeRev_toImplement.c,* respectively (found the SRC directory). Note that for the command *b*, you are not allowed to store the output of the command *f* in an array and subsequently print it.

IV. Testing:

For your convenience, we have provided you with sample tests and their corresponding outputs in the directory Tests. To see if your implementation of the commands *f* and *b* run correctly on the test files, type:

- Make test_range
- Make test_rangeRev

The test shall match your solution output with our reference output and if the difference is none, your implementation will pass the test. In addition to the provided test cases, you must devise tests (of your own) that run on different datasets (i.e., documents) of your choice and different argument values. Also, consider corner cases like invalid inputs, non-existent words etc.

V. Getting Started:

In order to jump-start your implementation, you should:

- Run the demo and make sure you understand the tree structure
- Study the important data structures defined in *def.h*
- Understand how the basic search (in *search.c*) works

VI. Deliverables:

Zip an archive (named P2_<your_andrew_id>.zip) containing a complete implementation of the files *keysRange_ToImplement.c* and *keysRangeRev_ToImplement.c* in addition to any other supplementary c files required to run the commands. If you will alter the makefile, please make sure to include your updated makefile as well. Submit this archive to the Andrew File System at /afs/qatar.cmu.edu/usr10/mhhammou/www/15415-s14/handin/p2/<your_andrew_id>

VII. Getting Help:

You can get help by visiting the professor and the TA during their office hours or by appointment. You can also post your questions on Piazza. The link for the course home page on Piazza is: http://piazza.com/qatar.cmu/spring2014/15415/home

VIII. Late Policy:

- If you hand in on time, there is no penalty (duh!).
- 0 -24 hours late = 25% penalty.
- 2 4 -48 hours late = 50% penalty.
- More than 48 hours late = you lose all the points for this project.

NOTE: You can use your grace-days quota. For details about the quota, please refer to the course syllabus.