CMU - SCS
15-415/15-615 Database Applications
Spring 2013, C. Faloutsos
Homework 3: Indexing
Due: 1:30pm on Thursday, 2/21/2013

Reminders - IMPORTANT:

- Like all homework, it has to be done **individually**.
- As elaborated in section 4 please submit your answers
  - **in hard copy, in class**, 1:30pm, on Thursday, 02/21/2013, **and**
  - e-copy of your code through blackboard, under `Assignments / Homework 3`. We will create a thread for this purpose.
- Please make sure your program compiles, and works without any errors or warnings on the **andrew machines**, as it will be tested there.
- **Test** your code for as many corner cases as you can imagine - we will grade it by running it on a set of 'secret' test cases we have created (like empty tree, tree with a single entry, etc etc).

Reminders - FYI:

- Weight: 20% of homework grade.
- The points of this homework add up to 100.
- Homework prepared by Danai Koutra.
- Rough time estimates: 20 hours ($\sim$5 hours to download, compile and understand the code, $\sim$ 15 hours to implement and test the required functions).

1 Preliminaries - Our B+ Tree Implementation

This assignment is designed to make you more familiar with the B+ Tree data structure. You are given a basic B+ Tree implementation and you are asked to extend it by implementing some new

operation/functions, that we list in subsection 2

The specifications of the basic implementation are:

1. It creates an "inverted index" in alphabetical order in the form of a B+ tree over a given corpus of text documents (explained in detail later).
2. It supports the following operations: insert, scan, search and print.
3. No duplicate keys are allowed in the tree. FYI: It uses a variation of "Alternative 3" and stores a postings list for each word that appears many times.
4. It **does not** support deletions.
5. The tree is stored on disk.

1.1 Where to Find Makefiles, Code, etc.

The file is available at `http://www.cs.cmu.edu/~christos/courses/dbms.S13/hws/HW3/db_asn3.zip` or through AFS (`/afs/cs.cmu.edu/user/christos/www/courses/dbms.S13/hws/HW3/db_asn3.zip`).

- Unzip the file.
- Type `make demo` to compile everything and see a demo.

The demo searches the key "american" and shows the contents of the documents containing the key.

1.2 Description of the provided B+ tree package

The directory structure and contents are as follows:

- `DOC`: contains a very useful documentation of the code.
- `SRC`: the source code.
- `Datafiles`: sample data to add to the tree.
- `Tests`: some sample tests and their solutions.
- Some other useful files, *e.g.*, `README, makefile` etc.
- **IMPORTANT:** Files `B-TREE_FILE, POSTINGSFILE, TEXTFILE, parms` are created by our B+ tree implementation, when a tree is created (recall that the implementation is disk-resident). To allow `main` to access this tree (across multiple executions), make sure that these files are not deleted and are present in the same directory as `main`. Conversely, delete these files if you want to create a new tree.

In more detail, the main program file is called "`main.c`." It waits for the user to enter commands and responds to them as shown in Table 1.

ARGUMENT	EFFECT
`C`	`P`rints all the keys that are present in the tree, in ascending lexicographical order.
`i arg`	The program parses the text in `arg` 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 `arg` make the "keys" of the B+ tree, and the value for all these keys is set to `arg`. Since this tree enables us to find which words are present in which documents, it is known as the *inverted index*.
`p arg`	Prints the keys in a particular `p`age of the B+ tree where `arg` is the page number. It also prints some statistics about the page such as the number of bytes occcupied, the number of keys in the page, etc.
`s <key>`	`s`earches the tree for <key> (which is a single word). If the key is found, the program prints "Found the key!". If not, it prints "Key not found!".
`S <key>`	`S`earches the tree for <key>. If the key is found, the program prints the documents in which the key is present, also known as the *posting list* of <key>. If not, it prints "Key not found!".
`T`	pre`T`ty-prints the tree. If the tree is empty, it prints "Tree empty!" instead.
`x`	exit

Table 1: Existing interface

2 Tasks

Our implementation provides each of the described operations in Table 1. In this assignment, you are asked to extend it to support the commands listed in Table 2.

ARGUMENT	EFFECT
`f` <key_1> <key_2>	**[50 pts]** Print in *alphabetical order* (`f`orward) the *distinct keys* that are in the range defined by <key_1> and <key_2> (including the bounds). If <key_1> and <key_2> are not in alphabetical order, it should print "Invalid key order!". If no documents have keys within the given range, it should print "Keys in the given range not found!".
`b` <key_1> <key_2>	**[50 pts]** Print in *reverse alphabetical order* (`b`ackward) the *distinct keys* that are in the range defined by <key_1> and <key_2> (including the bounds). If <key_1> and <key_2> are not in alphabetical order, it should print "Invalid key order!". If no documents have keys within the given range, it should print "Keys in the given range not found!".

Table 2: Operations to be implemented and their weights

Clarifications/Hints

- For your convenience, we have provided you with sample tests and their corresponding outputs in `Tests`. To see if your implementation runs correctly on the test files, type
  - `make test_range`
  - `make test_rangeRev`

  `test_range` and `test_rangeRev` test your implementation for the 'f' and 'b' command, respectively. If `diff` is empty for both `test_range` and `test_rangeRev`, then your implementation passes the provided tests! However, we will use several other (unpublished) test files during grading, so please also **make sure to test your implementation on other test files of your own on the andrew machines**.
- See def.h for important data structure information.

3 Testing Mechanism

We will test your submission for **correctness** using scripts, and also look through your code.

- **Correctness.** An easy test is to run your code against the sample test files provided with the assignment. For each test file, the output from your program should be exactly the same as the solution output (*i.e.*, `diff` is empty). Make sure you test your code on additional datasets of your own. Also, consider corner cases (e.g., invalid inputs, non-existent words etc.). For the command "b", where you are asked to give the keys within a given range in reverse alphabetical order, **you are not allowed** to store the words in alphabetical order in an array and then print them in reverse order. You have to make use of the structure of the B+ tree. **Note:** We will use extra (unpublished) test cases to grade.
- **Format.** We will use scripts to test the output of your code. Therefore, please make sure your output follows the same format as the sample test file solutions. That is, the result of `diff` between your output and the provided outputs, should be empty.
- **Code.** We will check the functions that you modified in order to support the required operations (see below what to hand-in).

4 What to hand-in

In short, a hard copy of the changed functions, and a tar-file with everything we need to run our tests.

1. Create a tar file of your complete source code including **all and only** the necessary files, as well as the `makefile` (i.e., exclude *.o *.out etc files).
2. Submit the tar file via blackboard, under `Assignments / Homework 3`.
3. Submit in class a hard copy (e.g., pdf file) with all the changes that you made to the source code for each operation that you are asked to implement. Please include in the pdf **only** the functions that you changed.