Carnegie Mellon University
Department of Computer Science
15-415/615 - Database Applications
C. Faloutsos & A. Pavlo, Spring 2015

Homework 3 (by Vinaykumar Vittal Bhat / Jiayu Liu)
Due: **hard and e-copy**, at **1:30pm, 2/19/2015**

IMPORTANT - Deliverables

1. **Hard copy**, in class:
   (a) The **answers** of your code to all questions, as well as
   (b) any new or modified **code** you created.
   Staple everything together, and type the usual info on the top (course#, etc)
2. **Online**, on "blackboard": a tar-file (`[andrew-id]-HW3.tar.gz`) of your code. Running `make` should compile it and pass all the tests.

Reminders:

- *Platform:* We shall run and grade your program on the *andrew linux machines.*
- *Plagiarism*: All homework is to be completed **individually**.
- *Typeset* all of your answers
- *For late homeworks*: email it (a) to all TAs (b) with the subject line exactly `15-415 Homework Submission (HW 3)` and (c) the count of slip-days you are using.
- *Recitations*: there will be two recitations, as announced:
  - Wed 2/11, 01:30-02:20pm, WeH 5302, by Jiayu
  - Wed 2/18, 01:30-02:20pm, WeH 5302, by Vinay

For your information (FYI):

- *Expected time*: 10-20 hours (≈2-4 hours for page counter and familiarization with the code; 3-6h for *k successors*; and 5-10h for *k predecessors.*

Revision : 2015/02/09 20:06

1 Preliminaries - Our B+ Tree Implementation

The goals of this assignment to make you more familiar with the B+ Tree data structure, especially the traversal and search functionalities.

Specifically, you are given a basic B+ Tree implementation and you are asked to extend it by implementing some new operation/functions, that we list later. (see bottom of Table 1).

1.1 Where to Find Makefiles, Code, etc.

The file is at `http://www.cs.cmu.edu/~christos/courses/dbms.S15/hws/HW3/btree.tar.gz`

Quick-start guide:

1. G-unzip and untar the file.
2. `make load` # *compiles everything and loads the data files*
3. `./main` # *to try out the program - e.g.* `S alex`
4. `make` # *like* `load`*, but it also runs tests - only the first test succeeds, on purpose*
5. `make test_search` # *the first test - should always work*

Explanations

- `make load` inserts the entire collection of documents (actually, a dictionary, split into thousands of files). Then, you can search for the key, say "alex", and see the contents of the documents containing the search key.
- `make` runs some tests against the code, and compares (`diff`) the output against the correct output. When your code is implemented correctly, then `make` should report all tests as successful.
- `make test_search` runs the very first test, which should pass out-of-the-box.

1.2 Description of the provided B+ tree package

The specifications of the provided 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.
2. It supports the operations in Table 1 (insert, scan, search, etc).
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, since it is persistent.

The directory structure and contents are as follows:

- `DOC`: contains a very useful documentation of the code.
- `SRC`: the source code.
- `Datafiles`: data documents, to insert to the tree.

Homework 3 continues…

- `Tests`: some sample tests and their solutions.
- Some other useful files, *e.g.*, `README, makefile` etc.
- **IMPORTANT:** Make sure you do *not* delete the files `B-TREE_FILE, POSTINGSFILE, TEXTFILE, parms` - they are created by the B+ tree implementation, they should be in the same directory as `./main`, and they are necessary, to make the B+tree persistent.

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`	Prints 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`	preTty-prints the tree. If the tree is empty, it prints "Tree empty!" instead.
`x`	exit
`#`	[**Not implemented yet**] prints and resets the counter for the number of `FetchPage` calls
`> key k`	[**Not implemented yet**] finds and prints the k successors in the B+ tree for the given `key`
`< key k`	[**Not implemented yet**] finds and prints the k predecessors in the B+ tree for the given `key`

Table 1: B+ tree command interface - the last 3 commands are to be implemented

2 Your tasks

Your task is to implement the last three commands shown in Table 1. Their detailed behavior is as follows:

`#` Prints and resets the number of page fetches (from disk) in the current program. Specifically it means the number of `FetchPage` function calls. (We need it for debugging and grading, to make sure the code avoids needless sequential scans).

`> key k` Search for *up to k* successors of the given key in the B+ Tree, sorted lexicographically. The `key` should *not* be included in the result, regardless whether the `key` is in the tree or not. If there are less than k successors in the B+ tree, the result should return them, even if they are less than k.

`< key k` Search for *up to k* predecessors of the given key in the B+ Tree, sorted lexicographically. Again, the `key` should *not* be included in the result; and a shorter-than-k list should be returned, when there are not enough predecessors.

2.1 Details

- **Efficiency**: Your code should *not* resort to sequential scanning - that is, it should require way less than L leaf accesses, where L is the number of leaves of the B+ tree ($\approx 70,000$, in our setting).
- **More examples:** The correct responses for some additional queries are in Table 2.
- **Rudimentary testing:** As can be seen from the provided `makefile`, running `make test_sanity` will do a minimal "sanity check" of your code on a few queries, and `diff` its results with the correct ones.
- **Additional testing:** Passing the few supplied tests of `make test_sanity`, is *necessary, but not sufficient*, for a good grade - please make sure you do your own, additional testing, for as many corner cases as you can think: empty tree, search key out of range, etc.
- **Page count**: we will allow for *small variations* for the *page count* results - as long as the code is faster than linear ($O(N)$), that is, it does *not* do needless sequential scans.

Next, we give you the list of questions; run your code and hand in its responses on the hard copy. The first four questions have the same format (same six parts), and they only differ by the search key (*hercules*, *onomatopoeia*, etc). Please make sure you use *the exact dataset and `parms` file* as in the provided tar-file.

Homework 3 continues...

Question 1: Successors and Predecessors search in a B+ Tree[100 points]

(a) For the key *hercules* answer the following questions:
   i. [**1 point**] Does the word exist in the document (using the command `s key`)?
   ii. [**1 point**] How many pages are read in order to see if the word is in the tree?
   iii. [**4 points**] What are the 5 successor of the key in the tree, sorted lexicographically?
   iv. [**2 points**] How many pages are read in order to fetch them?
   v. [**5 points**] What are the 5 predecessors of the key in the tree, sorted lexicographically?
   vi. [**2 points**] How many pages are read in order to fetch them?

(b) For the key *onomatopoeia* answer the following questions:
   i. [**1 point**] Does the word exist in the document (using the command `s key`)?
   ii. [**1 point**] How many pages are read in order to see if the word is in the tree?
   iii. [**4 points**] What are the 5 successor of the key in the tree, sorted lexicographically?
   iv. [**2 points**] How many pages are read in order to fetch them?
   v. [**5 points**] What are the 5 predecessors of the key in the tree, sorted lexicographically?
   vi. [**2 points**] How many pages are read in order to fetch them?

(c) For the key *sesquipedalian* answer the following questions:
   i. [**1 point**] Does the word exist in the document (using the command `s key`)?
   ii. [**1 point**] How many pages are read in order to see if the word is in the tree?
   iii. [**4 points**] What are the 5 successor of the key in the tree, sorted lexicographically?
   iv. [**2 points**] How many pages are read in order to fetch them?
   v. [**5 points**] What are the 5 predecessors of the key in the tree, sorted lexicographically?
   vi. [**2 points**] How many pages are read in order to fetch them?

(d) For the key *malloc* answer the following questions:
   i. [**1 point**] Does the word exist in the document (using the command `s key`)?
   ii. [**1 point**] How many pages are read in order to see if the word is in the tree?
   iii. [**4 points**] What are the 5 successor of the key in the tree, sorted lexicographically?
   iv. [**2 points**] How many pages are read in order to fetch them?
   v. [**5 points**] What are the 5 predecessors of the key in the tree, sorted lexicographically?
   vi. [**2 points**] How many pages are read in order to fetch them?

(e) [**40 points**] We will test your code on several "secret" settings, which we will publish *after* the due date. Test for as many corner cases as you can, to get full points here.

Question 1 continues. . .

S alex	> alex 26	< alex 26
enter search-word:	word=?	word=?
*** Searching for word alex	k=?	k=?
found in alex	found 26 successors:	found 26 predecessors:
-------document #1-----	alexander	alethea
rembrandtism	alexanders	alethiology
sinomenine	alexandra	alethopteis
inductorium	alexandreid	alethopteroid
alex	alexandrian	alethoscope
resoothe	alexandrianism	aletocyte
usuary	alexandrina	aletris
sulphatase	alexandrine	alette
eurythmical	alexandrite	aleukemic
buffont	alexas	aleurites
ridgepoled	alexia	aleuritic
salvadoraceous	alexian	aleurobius
cytopathologic	alexic	aleurodes
nonbursting	alexin	aleurodidae
clapping	alexinic	aleuromancy
batholith	alexipharmacon	aleurometer
octachord	alexipharmacum	aleuronat
tautometric	alexipharmic	aleurone
clockroom	alexipharmical	aleuronic
eta	alexipyretic	aleuroscope
ensate	alexis	aleut
spiropentane	alexiteric	aleutian
renomination	alexiterical	aleutic
unsentimentalist	alexius	aleutite
schemeful	aleyard	alevin
remissly	aleyrodes	alewife
#	#	#
# of reads on B-tree: 11	# of reads on B-tree: 19	# of reads on B-tree: 29

(a) (b) (c)

Table 2: Expected responses, to additional, example queries: (a) ‘S’ for ‘search for word’; (b) ‘>’ for ‘successors search’, and (c) ‘<’ for ‘predecessors search’.

Question 1 continues…

2.2 Clarifications/Hints

- Your implementation should be *case insensitive*. All keys are inserted after converting them to lower case.
- Make sure all searches are only for *alphanumeric* strings.
- *Rudimentary testing*: running `make`, or, more detailed
  - `make test_sanity`

  should return success. `make test_sanity` tests your implementation for the `>` and `<` commands, respectively. If `diff` is empty for both of them, then your implementation passes the provided tests! Please refrain from changing these tests as they serve as a check-point for the expected output format.
- *Automatic grading*: FYI, we will do `make grade`. Please leave unchanged the `grade` target in the makefile.

Hints, and optional information:

- For your convenience, we have provided the following place-holder files:
  - `stats.c`
  - `get_successors.c`
  - `get_predecessors.c`
- *Hint:* Implementing `get_successors.c` should be easier than `get_predecessors.c`.
- We recommend the use of source code version control tools, like 'git', 'mercurial', or 'svn'.
- For your convenience, we have also provided you with most of the queries, for the questions above (*hercules* etc). Within the `Tests/` directory, check `test_successors.inp` and `test_predecessors.inp`. Feel free to modify those input files, if you want to automate the generation of your answers for the hard-copy deliverable.

3 Testing and Grading

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

Correctness. As we said earlier, an easy, minimal check is through: `make test_sanity` - your code should pass. However, please make sure you test your code on *additional* settings, of your own. Consider corner cases (e.g., empty tree, invalid inputs, non-existent words etc.). As mentioned, we will use several, *additional*, "secret" test cases to grade your code.

Output Format. If `make test_sanity` is successful, you have the right output format.

Code. We will check the functions that you created/modified to support the required operations (e.g., `stats.c`, etc).

Question 1 continues...

4 What to hand-in

As we said in the front page, we want both a hard copy of the changed functions; and a `tar`-file with everything we need to run our tests.

1. **Hard copy**: in class, please submit
   - (a) your *answers* to the questions listed, and
   - (b) all the *changes* that you made to the source code.

   Please hand-in **only** the functions that you added / changed.
2. **Online:**
   - Create `[your-andrew-id]-HW3.tar.gz`, a (compressed) `tar` file of your complete source code including **only and all** the necessary files, as well as the `makefile` (i.e., exclude *.o *.out etc files);
   - Submit your `tar` file via blackboard, under `Assignments / Homework 3`.

For your convenience, `make handin` automates the collection of deliverables. However, it is **your responsibility** to make sure everything is included properly.