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

Homework 3 (by Anna Etzel)

Due: **hard and e-copy** at **3:00pm, on Wednesday, Oct. 7**

VERY IMPORTANT: Deposit **hard copy** of your answers and a **hard copy** of any new or modified **code**, and also submit a **tar-file** (`[andrew-id]-HW3.tar.gz`) of your code on Blackboard. For ease of grading, please

1. **Separate** your answers, on different pages for each question (staple additional pages, if needed).
2. **Type** the full info on **each** page: your **name**, **Andrew ID**, **course#**, **Homework#**, **Question#** on each of the 1 pages.
3. **Check** that running `make` compiles the code that you submit and that it passes all the tests.

Reminders:

- *Platform*: We will run and grade your program on the *andrew linux machines*.
- *Plagiarism*: Homework is to be completed *individually*.
- *Typeset* all of your answers whenever possible. Illegible handwriting may get zero points, at the discretion of the graders.
- *Late homeworks*: in that case, please email it
  - to all TAs
  - with the subject line exactly `15-415 Homework Submission (HW 3)`
  - and the count of slip-days you are using.

For your information:

- Graded out of **100** points; **1** questions total
- Rough time estimate: *approx. 15-20 hours* - be sure to start early

Revision : 2015/09/27 12:03

Question	Points	Score
Prefix Bracket Search in a B+ Tree	100	
Total:	100	

1 Preliminaries - Our B+ Tree Implementation

The goal of this assignment is 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 operations/functions, which 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.F15/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 that are not marked unimplemented in Table 1.
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 useful documentation of the code.
- `SRC`: the source code.

- `Datafiles`: data documents, to insert to the tree.
- `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.

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 inserts 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 page 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`	searches 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`	Searches 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`	[**Not implemented yet**] finds and prints the next sequential key in B+ tree which does not have the given `key` as a prefix
`[ key`	[**Not implemented yet**] finds and prints the previous sequential key in the B+ tree which does not have the given `key` as a prefix

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`
: Search for the 'right bracket' of `key`, the first key that follows the given `key` in the B+ tree which does not have `key` as a prefix. If no such key is found (for example, if the last key in the B+ tree has this prefix or this prefix is lexicographically after all keys in the B+ tree), ***NONE*** should be returned.

`[ key`
: Search for the 'left bracket' of `key`, the first key that precedes the given `key` in the B+ tree which does not have `key` as a prefix. If no such key is found (for example, if the first key in the B+ tree has this prefix or this prefix is lexicographically before all keys in the B+ tree), ***NONE*** should be returned.

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:** 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 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.

Question 1: Prefix Bracket Search in a B+ Tree . . . . [100 points]

For the following list of questions, run your code to find the answers and put your responses in the hard copy you turn in. The first four questions have the same six-part format and only differ by the search key (*database*, *comb*, etc). Please make sure you use the same dataset and `parms` file as in the provided tar-file.

Questions about the number of pages read are *not* graded against an exact value. These questions are meant to check that you are not scanning through the B+ tree sequentially to find the correct output.

(a) For the key *database* 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 is the right bracket of the key in the tree, sorted lexicographically?

iv. **[2 points]** How many pages are read in order to fetch this? (There is not just one correct answer for this, a range of values are accepted.)

v. **[5 points]** What is the left bracket of the key in the tree, sorted lexicographically?

vi. **[2 points]** How many pages are read in order to fetch this? (There is not just one correct answer for this, a range of values are accepted.)

(b) For the key *comb* 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 is the right bracket of the key in the tree, sorted lexicographically?

iv. **[2 points]** How many pages are read in order to fetch this? (There is not just one correct answer for this, a range of values are accepted.)

v. **[5 points]** What is the left bracket of the key in the tree, sorted lexicographically?

vi. **[2 points]** How many pages are read in order to fetch this? (There is not just one correct answer for this, a range of values are accepted.)

(c) For the key *h* 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 is the right bracket of the key in the tree, sorted lexicographically?

iv. **[2 points]** How many pages are read in order to fetch this? (There is not just one correct answer for this, a range of values are accepted.)

v. **[5 points]** What is the left bracket of the key in the tree, sorted lexicographically?

Question 1 continues. . .

vi. **[2 points]** How many pages are read in order to fetch this? (There is not just one correct answer for this, a range of values are accepted.)

(d) For the key *zu* 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 is the right bracket of the key in the tree, sorted lexicographically?

iv. **[2 points]** How many pages are read in order to fetch this? (There is not just one correct answer for this, a range of values are accepted.)

v. **[5 points]** What is the left bracket of the key in the tree, sorted lexicographically?

vi. **[2 points]** How many pages are read in order to fetch this? (There is not just one correct answer for this, a range of values are accepted.)

(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.

S dragon	] dragon	[dragon
enter search-word: *** Searching for word dragon found in dragon -------document #1----- colarin hypopepsinia chloroacetic hydnocarpate wickiup dragon spermoviduct simoleon zorilla backbiter drib capsuler coapt preservability certainly dextrotropous agricultural polymorphy ancienty necrobacillary waivery rebringer baillone maigre tenonectomy # # of reads on B-tree: 11	word=? Right bracket of dragon: dragoon # # of reads on B-tree: 18	word=? Left bracket of dragon: dragomanish # # of reads on B-tree: 14
(a)	(b)	(c)

Table 2: Example responses, to additional test queries: (a) 'S' for 'search for word'; (b) '`]`' for 'right bracket of prefix', and (c) '`[`' for 'left bracket of prefix'. The search and bracket queries should give the exact responses shown. However, the number-of-read responses to the left and right bracket queries are not the exact ones required, as the amount is implementation specific.

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 `make test_sanity` should return success. `make test_sanity` tests your implementation for the `]` and `[` commands. 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.

Hints, and optional information:

- For your convenience, we have provided the following place-holder files:
  - `stats.c`
  - `get_rightbracket.c`
  - `get_leftbracket.c`
- *Hint:* Implementing `get_rightbracket.c` may be easier than `get_leftbracket.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 (*database* etc). Within the `Tests/` directory, check `test_rightbracket.inp` and `test_leftbracket.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 would be using `make test_sanity`. Your code should pass this. However, please make sure you test your code on *additional* settings, of your own. Consider corner cases (empty tree, invalid inputs, non-existent words/prefixes, 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).

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.