Carnegie Mellon University
Department of Computer Science
15-415/615 - Database Applications
C. Faloutsos & A. Pavlo, Fall 2016

Homework 3 (by Huanchen Zhang)

Due: **hard and e-copy** at **3:00pm, on Wednesday, Oct. 5**

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. **Type** the full info on **each** page: your **name**, **Andrew ID**, **course#**, **Homework#**, **Question#** on each page.
2. **Check** that running `make` compiles the code and 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.
- *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; **2** questions total
- Rough time estimate: *approx. 20 hours* - be sure to start early

Revision : 2016/10/07 17:17

Question	Points	Score
Fun with Orders	20	
Count My Range	80	
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. Specifically, you are given a basic B+ Tree implementation and you are asked to extend it by implementing some new operations/functions.

1.1 Where to Find Makefiles, Code, etc.

The file is at `http://15415.courses.cs.cmu.edu/fall2016/hws/HW3/btree.tar.gz`

Quick-start guide:

- `tar xvfz btree.tar.gz; make`

to see a small demo. Moreover:

1. `make load` # *compiles everything and loads the data files*
2. `./main` # *to try out the program - e.g.* `S christos`
3. `make` # *like* `load`*, but it also runs tests - only the first test succeeds, on purpose*
4. `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 a simple search test, which should pass out-of-the-box.
- `make test_sanity` runs some very simple tests on the new functionalities to be implemented, which should *fail* out-of-the-box. Before submitting your code, **make sure it passes these tests.**

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`	s**C**an and 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*.
`o arg`	(For **Question 1**) Inserts the uncommon words in the text file (specified by `arg`) into the B+ tree. Unlike "i arg", the keys are inserted in the same **o**rder as they appear in the text file.
`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.
`#`	Prints and resets B+ tree statistics. For this assignment, only "number of reads (B+ tree page accesses)" is printed and reset.
`n`	Count the **n**umber of pages in the B+ tree.
`x`	E**x**it

Table 1: Existing B+ Tree command interface

2 Your tasks

Your task is to implement the commands listed in Table 2. The detailed behavior and implementation hints of the commands are embedded in the following questions.

ARGUMENT	EFFECT
`c pageNum`	(**Question 2(a)**) counts the number of keys in the subtree rooted at page `pageNum`.
`r leftkey rightkey`	(**Question 2(b)**) Counts the number of keys in the range [`leftkey`, `rightkey`] (boundaries are inclusive).

Table 2: B+ Tree commands to be implemented

Question 1: Fun with Orders ..... [20 points]

Note#1: For this question, you do NOT need to modify the B+ tree code we give you. You only need to report the required counts and to re-arrange the insertion order.

Note#2: Here, we are NOT doing the bulk-loading algorithms in the textbook or the foils - we are inserting one-word-at-a-time.

Let's warm up with a simple question. Suppose you want to insert the words in a big text file into an empty B+ tree, one at a time. How important is the insertion order (in terms of the space-efficiency of the resulting B+ tree)?

Lets find out! Please follow the steps below:

1. *Set page size*: Change the first line in `SRC/parms`, to 1024 (from the default = 128).
2. `python gen_textfile.py`
   This will generate two big text files (`bulkload_random`, `bulkload_sort`). The files contaion all the words in `Datafiles` in random order, and sorted order, respectively.
   `python gen_textfile.py`
3. `cd SRC; make` *# Compile the source code*
4. `./main` *# Run the main program*
5. *# Load text file "bulkload_random":*
   `o`
   `../bulkload_random`
6. *# Count the number of pages in the B+ tree:*
   `n`
7. *# Exit; Clear and recompile the source code:*
   `x`
   `make spotless`
   `make`

8. Repeat step 3-5 for "bulkload_sort".

Then, please answer the following questions:

(a) **[5 points]** How many pages does the B+ tree have when the words are inserted in **random** order?

Solution: 7630 (or around)
Grading:
Full points if the number is within 10%
-5 points if not

(b) **[5 points]** How many pages does the B+ tree have if the words are inserted in **alphabetical** order?

Solution: 10385
Grading:
-5 points if wrong

(c) **[10 points]** Can you do better than random? Try the following "*delayed-third*" heuristic: insert the words in sorted order, but omitting every third word; then insert the left-overs in random order.

More formally, suppose the text file has $3N$ words. Let W be the word set. Let $w[i]$ denote the ith word in W in alphabetical order ($1 \leq i \leq 3N$). Divide W into two subsets: $S = \{w[3i+1], w[3i+2], where\ 0 \leq i \leq N-1\}$ and $R = \{w[3i], where\ 1 \leq i \leq N\}$. First, insert the words in set S in alphabetical order. Then insert the words in set R in random order.

Report the number of pages in the B+ tree, under the above *delayed-third* heuristic.

Solution: 6922
Grading:
-10 points if wrong

(d) **[0 points]** **[optional but encouraged]** Can you do even better than the *delayed-third* heuristic? If yes, provide a script (you may use the provided python script as a starting point) that generates your input file "`bulkload_my`" (of course, your text file must have the same words as in "`bulkload_random`" and "`bulkload_sort`") and

- report the number of pages of the resulting B+ tree,
- and explain your idea and your results in a few sentences.

We will announce in class the names of all people whose solution beats the "*delayed-third*" heuristic!

Question 2: Count My Range ..... [80 points]

IMPORTANT: Before attempting this question, please switch the B+ tree page size back to 128 (*i.e.,* change the first number in `SRC/parms` to 128).

Note: For questions asking for the number of pages read during the query, there is no single correct answer. A range of numbers will be accepted for grading.

Suppose you are asked the following query for your B+ tree: how many keys are included in the range between "christos" and "huanchen" (inclusive)?

Naive / Bad solution: A naïve way to answer this query is to find the boundary leaf nodes that contain "christos" and "huanchen", and then traverse the leaf nodes between them and count the keys. This solution, however, requires a large number of disk accesses, especially when the subtree is wide, and is thus very slow.

Good solution: We modified the B+ tree page header to include in each page a "SubtreeKeyCount" field (see `SRC/def.h`) that will hold the number of keys included in the subtree of the page (*i.e.*, the page is the subtree root). Note that we only modified the page header struct in `def.h` (and a few other files such as `FetchPage.c`, `Flushpage.c`, `fillIn.c`, `PrintTree.c`, to make the code compile). The actual implementation of updating the "`SubtreeKeyCount`" field, is left for you to do.

(a) Your first task is to implement the operations on the "SubtreeKeyCount" field. Your code must update the "SubtreeKeyCount" fields dynamically. For example, after each key insertion, the "SubtreeKeyCount" fields in the related pages must be updated. Your code must be efficient (i.e., you cannot recompute the "SubtreeKeyCount" for every page in a brute-force way upon insertion). Be sure to implement *subtreeKeyCount.c* so that the `c arg` command in the main program is implemented. Also, please add the usual comments in your code.

Hint: You may implement the "naïve" way (describe above) first and then verify your "SubtreeKeyCount" implementation against the "naïve" count.

Based on your "SubtreeKeyCount" implementation, please answer the following questions:

i. **[0 points]** Reset the B+ tree page size to 128 (i.e., change the first number in SRC/parms to 128).

ii. **[3 points]** Load all the files in `Datafiles` directory EXCEPT `dict000000` to your B+ tree. What is the "SubtreeKeyCount" for page **#1** (*i.e.,* `c 1`)? How many pages are read during the query?

Solution: 234196, 1
Grading:
-2 if the subtreeKeyCount is wrong
-1 if the # pages read is wrong

iii. **[3 points]** What is the "SubtreeKeyCount" for page **#35643** (*i.e.,* `c 35643`)? How many pages are read during the query?

Solution: 36394, 1
Grading:
-2 if the subtreeKeyCount is wrong
-1 if the # pages read is wrong

iv. **[3 points]** What is the "SubtreeKeyCount" for page **#2016** (*i.e.,* `c 2016`)? How many pages are read during the query?

Solution: 4, 1
Grading:
-2 if the subtreeKeyCount is wrong
-1 if the # pages read is wrong

v. **[5 points]** Now insert `Datafiles/dict000000` into your B+ tree (*i.e.,* `i Datafiles/dict000000`). How many pages are read during the query?

Solution: 262 (or around)
Grading:
-1 if the # is between 1.5x and 2x
-2 if the # is between 2x and 4x
-3 if the # is between 4x and 8x
-5 if the # is > 8x

vi. **[2 points]** What is the "SubtreeKeyCount" for page **#1** now?

Solution: 234221
Grading:
-2 if wrong

vii. **[2 points]** What is the "SubtreeKeyCount" for page **#35643** now?

Solution: 36395
Grading:
-2 if wrong

viii. **[20 points]** We will test your code using our "private" test cases, which we will publish *after* the due date. Make sure your code compiles and is well-tested.

Solution: Grading: There are 5 test cases. Each test case is worth 2 points. So a total of 10 points are for CORRECTNESS.
The other 10 points are for EFFICIENCY.
If the # pages read **DURING BULK INSERTION** in my solution is x:
-3 if 2x - 4x
-6 if 4x - 8x
-10 if > 8x

(b) You are now ready to implement an efficient "range key count" query (the `r leftkey rightkey` command in Table 2)! Note that:

- both the left and the right boundaries are inclusive

- the boundary keys may not exist in the B+ tree. For example, for query "r huanchen huanchen", if "huanchen" exists in the B+ tree, the query returns 1; otherwise, the query returns 0.
- the range boundaries may be flipped (e.g., `r zoo aaron`) - then the query returns 0.

You could/should use your "SubtreeKeyCount" implementation in part (a) for an efficient algorithm.

Based on your "range key count" implementation, load all the files in `Datafiles` directory to your B+ tree (*i.e., make load*) and answer the following questions:

i. **[4 points]** Run command `r huanchen zhang`. How many keys fall in the range [huanchen, zhang]? How many pages are read during the query?

Solution: 146269, 47 (or around)
Grading:
-2 if the range is wrong
-1 if the # pages read is between 2x and 4x
-2 if the # pages read is > 4x

ii. **[4 points]** Run command `r andy pavlo`. How many keys fall in the range [andy, pavlo]? How many pages are read during the query?

Solution: 131521, 42 (or around)
Grading:
-2 if the range is wrong
-1 if the # pages read is between 2x and 4x
-2 if the # pages read is > 4x

iii. **[4 points]** Run command `r tequila vodka`. How many keys fall in the range [tequila, vodka]? How many pages are read during the query?

Solution: 29213, 39 (or around)
Grading:
-2 if the range is wrong
-1 if the # pages read is between 2x and 4x
-2 if the # pages read is > 4x

iv. **[30 points]** Again, we will test your code using our "private" test cases, which we will publish *after* the due date. Make sure your code compiles and is well-tested.

Solution: Grading: There are 10 test cases. Each test case is worth 1.5 points. So a total of 15 points are for CORRECTNESS.
The other 15 points are for EFFICIENCY.
If the # pages read in my solution is x:
-5 if 2x - 4x
-10 if 4x - 8x
-15 if > 8x

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, etc.). As mentioned, we will use several, *additional*, "private" 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.

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.