Carnegie Mellon University
Department of Computer Science
15-415/615 - Database Applications
C. Faloutsos & A. Pavlo, Spring 2014

Homework 3

IMPORTANT

- **Hand in** your answers in **class at 1:30pm on Thursday, 2/20/2014** and **submit online** the appropriate `tar` file of your solution, that is:
  - **Hard copy**: Your answers to all questions, as well as any *new or modified* code in the directory
  - **Online**: a `tar` of your code such that running `make load` compiles it and runs the code as if it were a first run.
- **Platform:** We shall run your program on the **andrew machines**.
- **Test** your code for as many corner cases as you can imagine - we will grade it by running `diff` on its output against our answers, on a set of 'secret' test cases we have created (like an empty tree, a tree with a single entry, etc etc).

REMINDERS: As we said earlier:

- **Plagiarism**: Homework may be discussed with other students, but all homework is to be completed **individually**.
- **Typeset** all of your answers whenever possible. Illegible handwriting may get no points, at the discretion of the graders.
- **Late homeworks**: If you are turning your homework in late, 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.

Revision : 2014/03/06 10:50

1 Preliminaries - Our B+ Tree Implementation

The goals of this assignment are two-fold: (a) to make you more familiar with the B+ Tree data structure, and (b) to illustrate that it can lead to significantly faster responses, for appropriate queries.

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.

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.S14/hws/HW3/btree.tar` or through AFS (`/afs/cs.cmu.edu/user/christos/www/courses/dbms.S14/hws/HW3/btree.tar`).

- Untar the file.
- Type `make load` to compile everything and load the data files
- Type `./main` to start the program and feel free to search with commands like `S alex` and `S vagelis`

`make load` inserts the entire dictionary (split into thousands of files) and then you can search for the keys "alex" and "vagelis" 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.

Homework 3 continues…

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

Table 1: Existing interface

Question 1: Prefix and Substring Search in a B+ Tree[100 points]

1. For each command record the number of pages (from disk) the B+ Tree reads and output it whenever a command is completed.
2. For the command `P <key>` search for any word in the B+ Tree where the string `<key>` is a prefix to that word. The B+ Tree should return a list of all words for which the given key is a prefix as well as the total number of such words. Of course, implement this intelligently so that the number of pages read from disk is minimized.
3. For the command `M <key>` search for any word in the B+ Tree where the string `<key>` is found anywhere in that word, including in the middle. The B+ Tree should return a list of all words for which the given key is a substring as well as the total number of such words. Of course, implement this intelligently so that the number of pages read from disk is minimized.

An example of the correct response for such commands after the appropriate modifications can be seen in Table 2. In the source code you have downloaded, there are a number of shell files which will be useful for you to implement this functionality. We suggest you work off of those.

Once implemented please answer the following questions so we can appropriately grade your implementation:

(a) For the key *data* 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 need are read to see if the word is used in the document?
  iii. [**3 points**] How many words have the key as a prefix?
  iv. [**3 points**] What are the words that have the key as a prefix?
  v. [**3 points**] How many pages are read in finding the words that have the key as a prefix?
  vi. [**3 points**] How many words have the key as a substring?
  vii. [**3 points**] What are the words that have the key as a substring?
  viii. [**3 points**] How many pages are read in finding the words that have the key as a substring?

(b) For the key *andy* 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 need are read to see if the word is used in the document?
  iii. [**3 points**] How many words have the key as a prefix?
  iv. [**3 points**] What are the words that have the key as a prefix?

Question 1 continues. . .

   v. [**3 points**] How many pages are read in finding the words that have the key as a prefix?
   vi. [**3 points**] How many words have the key as a substring?
   vii. [**3 points**] What are the words that have the key as a substring?
   viii. [**3 points**] How many pages are read in finding the words that have the key as a substring?

(c) For the key *christos* 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 need are read to see if the word is used in the document?
   iii. [**3 points**] How many words have the key as a prefix?
   iv. [**3 points**] What are the words that have the key as a prefix?
   v. [**3 points**] How many pages are read in finding the words that have the key as a prefix?
   vi. [**3 points**] How many words have the key as a substring?
   vii. [**3 points**] What are the words that have the key as a substring?
   viii. [**3 points**] How many pages are read in finding the words that have the key as a substring?

(d) [**40 points**] We will run your implementation on the three keys above as well as a variety of other test cases and assign points for the remaining points proportionally. Make sure you test for corner cases.

Question 1 continues. . .

S alex	P alex	M alex
*** Searching for word alex	*** Searching for prefix alex	*** Searching for substring alex
found in alex	Prefix found in alex	Substring found in alex
-------document #1-----	Prefix found in alexander	Substring found in alexander
rembrandtism	Prefix found in alexanders	Substring found in alexanders
sinomenine	Prefix found in alexandra	Substring found in alexandra
inductorium	Prefix found in alexandreid	Substring found in alexandreid
alex	Prefix found in alexandrian	Substring found in alexandrian
resoothe	Prefix found in alexandrianism	Substring found in alexandrianism
usuary	Prefix found in alexandrina	Substring found in alexandrina
sulphatase	Prefix found in alexandrine	Substring found in alexandrine
eurythmical	Prefix found in alexandrite	Substring found in alexandrite
buffont	Prefix found in alexas	Substring found in alexas
ridgepoled	Prefix found in alexia	Substring found in alexia
salvadoraceous	Prefix found in alexian	Substring found in alexian
cytopathologic	Prefix found in alexic	Substring found in alexic
nonbursting	Prefix found in alexin	Substring found in alexin
clapping	Prefix found in alexinic	Substring found in alexinic
batholith	Prefix found in alexipharmacon	Substring found in alexipharmacon
octachord	Prefix found in alexipharmacum	Substring found in alexipharmacum
tautometric	Prefix found in alexipharmic	Substring found in alexipharmic
clockroom	Prefix found in alexipharmical	Substring found in alexipharmical
eta	Prefix found in alexipyretic	Substring found in alexipyretic
ensate	Prefix found in alexis	Substring found in alexis
spiropentane	Prefix found in alexiteric	Substring found in alexiteric
renomination	Prefix found in alexiterical	Substring found in alexiterical
unsentimentalist	Prefix found in alexius	Substring found in alexius
schemeful	"alex" is the prefix of 25 words	Substring found in antialexin
remissly	18 pages read	Substring found in catalexis
		Substring found in dicatalexis
10 pages read		Substring found in hypercatalexis
		Substring found in malexecution
		Substring found in paralexia
		Substring found in paralexic
		"alex" is in 32 words
		74552 pages read

Table 2: Example response: 'S' for 'search for word'; 'P' for 'prefix search', and 'M' for 'middle-search' = substring search.

Question 1 continues. . .

Clarifications/Hints

- We have provided the following empty files:
  - `prefix_search.c`
  - `prefix_searchLeaf.c`
  - `prefix_treesearch.c`
  - `ss_search.c`
  - `ss_searchLeaf.c`
  - `ss_treesearch.c`
  - `FindPrefixPosition.c`
  - `CheckSubstring.c`
  - `CheckPrefix.c`

  We suggest you use them for filling in your implementation.
- Your implementation should not be case sensitive. All keys are inserted after converting them to lower case.
- Make sure all searches are only for alphanumeric strings.
- 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_prefix`
  - `make test_substring`

  `test_prefix` and `test_substring` test your implementation for the `P` and `M` commands, respectively. If `diff` is empty for both `test_prefix` and `test_substring`, 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**.
- Note, there are two valid ways to perform prefix and substring search. We will allow either but the number of pages read must match one of the two methods.
- See def.h for important data structure information.

2 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.). **Note:** We will use extra (unpublished) test cases to grade.

Question 1 continues. . .

- **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).

3 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. **Online:** 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); submit your `tar` file via blackboard, under `Assignments / Homework 3`. Use the name `[your-andrew-id]-HW3.tar`.
2. **Hard copy**: Submit in class your answers to the questions listed, and all the changes that you made to the source code for each operation that you are asked to implement. To save trees, please include in the hard copy **only** the functions that you changed.

Solution:

Test cases:

1. "data" - This is very normal string to search for (normal search, prefix, and substring).
   (a) "data" found with search after reading 10 pages.
   (b) "data" is the prefix of 6 words, requiring 11 page reads.
   (c) "data" is the substring of 22 words, requiring 74552 page reads.
2. "andy" - This is very normal string to search for (normal search, prefix, and substring).
   (a) "andy" found with search after reading 10 pages.
   (b) "andy" is the prefix of 1 word, requiring 10 page reads.
   (c) "andy" is the substring of 46 words, requiring 74552 page reads.
3. "christos" - This is very normal string to search for (normal search, prefix, and substring).
   (a) "christos" found with search after reading 10 pages.
   (b) "christos" is the prefix of 1 word, requiring 11 page reads.
   (c) "christos" is the substring of 1 words, requiring 74552 page reads.

Question 1 continues…

4. "aa" - This string is at the very beginning of the alphabet so it tests that side of the tree
   (a) "aa" found with search after reading 10 pages.
   (b) "aa" is the prefix of 13 words, requiring 13 page reads.
   (c) "aa" is the substring of 130 words, requiring 74552 page reads.
5. "zyz" - This string is very rare and occurs at the very end of the dictionary such that we make sure it goes all the way to the end
   (a) "zyz" is not found with search after reading 10 pages.
   (b) "zyz" is the prefix of 2 words, requiring 10 page reads.
   (c) "zyz" is the substring of 2 words, requiring 74552 page reads.
6. "alskj" - In the original dictionary this string does not exist (even substring). This is to make sure they can appropriately return that nothing was found.
   (a) "alskj" is not found with search after reading 10 pages.
   (b) "alskj" is the prefix of 0 words, requiring 10 page reads.
   (c) "alskj" is the substring of 0 words, requiring 74552 page reads.
7. "alskj" - We then insert a new document that contains this word and this string as a substring of many different words. We then re-run the tests to make sure that the BTree uses these newly inserted words.
   (a) "alskj" is found with search after reading 10 pages.
   (b) "alskj" is the prefix of 3 words, requiring 10 page reads.
   (c) "alskj" is the substring of 4 words, requiring 74554 page reads.

You can see the full solution at `http://www.cs.cmu.edu/~christos/courses/dbms.S14/hws/HW3/sols.tar` and the code at `http://www.cs.cmu.edu/~christos/courses/dbms.S14/hws/HW3/btree_sol.tar`

A summary of how we graded the assignments:

- There are 4 additional test cases provided. Each is graded with the same distribution as for the three given but divide all point values in half for these additional queries.
- -3 if the tar file was submitted incorrectly (without makefiles, data files, etc.)
- -2 for small compilation errors
- -3 if corrupted directory to the point that we can't compile and had to email the student.

Question 1 continues. . .

- -2 if they dont print the number of pages read for an unsuccessful 'S' command
- -0.5 for every error such as FetchPage: Pagenum -3 out of range (1,100704) FetchPage: corrupted Page -3 OR segmentation fault
- -5 if they dont count the number of pages for each of the two search methods.
- -4 for not printing out the number of pages for S (but having the right number in the printed document for the 3 given cases)