Carnegie Mellon University in Qatar

Database Applications

15-415 - Spring 2020

Project 3

Out: March 30, 2020

Due: April 18, 2020

1 Project Objectives

DBMSs heavily use and rely on B^+ trees to speed-up operations like equality searches, range searches, grouping, and ordering, among others. This project is designed to make you familiar with the implementation of a B^+ tree data structure. You will extend a basic B^+ tree implementation by incorporating additional functionalities described later in the document.

2 The B^+ Tree Package

You are provided with a B^+ Tree Package which contains an implementation of a B^+ tree with *Alternative 3*. This package will be the basis of this project. In the forthcoming paragraphs, we describe the package content and how to compile it.

2.1 A Brief Primer

The package encompasses seven folders alongside a `README` and a `Makefile`. The directories and their contents are as follows:

1. `bin`: main driver program for creating and using the index.
2. `conf`: configuration parameters for tuning the index.
3. `datasets`: two demo datasets, namely `dictionary` and `movies`.
4. `db`: stores the persisted index data files (described below).
5. `man`: contains user and programming manuals.
6. `src`: the project's source code.
7. `tests`: demos and sample tests with their solutions.

The `README` contains information about using the provided `Makefile` for compiling, cleaning, and testing the code, in addition to a description of the commands supported by the basic B^+ tree.

2.2 Compilation

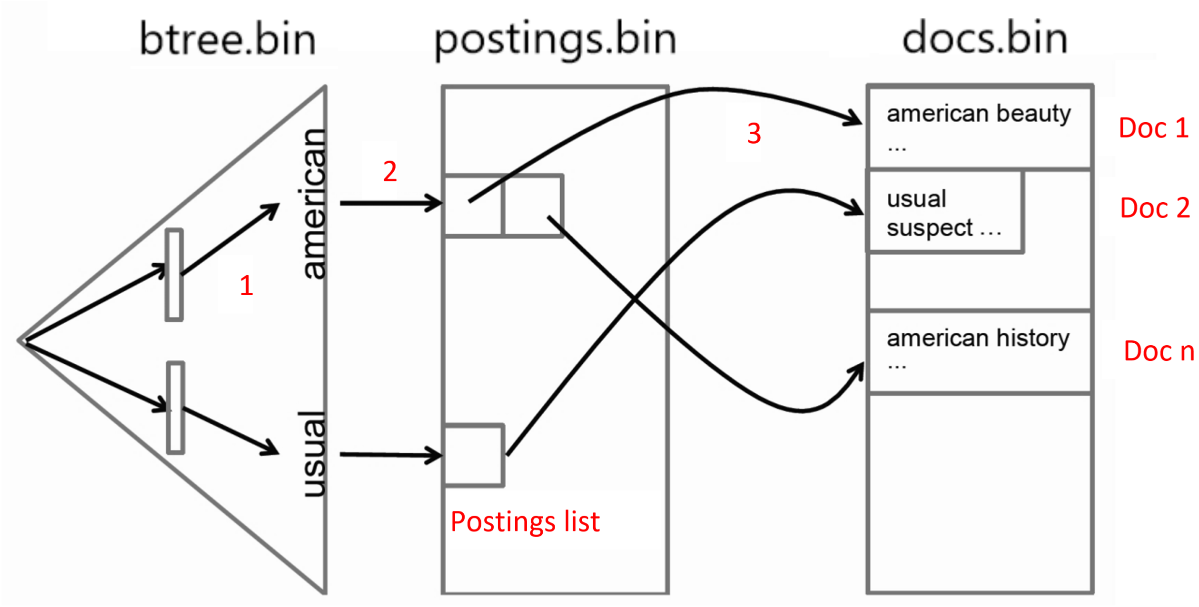
To compile the source code, unzip the package and type `make`. This compiles the code and creates an executable called `main` in addition to three files in the `db` folder: `btree.bin`, `postings.bin`, and `docs.bin`. All information in the B^+ tree is stored in these files as follows:

1. `btree.bin`: stores the tree nodes including keys and pointers.
2. `postings.bin`: stores the list of pointers to data records.
3. `docs.bin`: stores the actual data records.

To allow the program to access a tree and its data records across multiple executions, do not delete these files and ensure that they reside in the `db` folder. Conversely, delete those files via `make clean-db` when you wish to create a new tree.

2.3 High-Level View

Figure 1 below presents a high-level view of the structure of our B^+ tree with *Alternative 3*. Each non-leaf node stores a set of keys and pointers to other nodes (as illustrated by arrow 1). In a leaf node, however, a pointer associated with a particular key refers to a list of pointers called the postings list (as illustrated by arrow 2). Each element or *posting* in the postings list is a pointer referring to a text document that contains the key (as illustrated by arrow 3).



Each node or posting is a `C struct` whose binary representations are stored in `btree.bin` and `postings.bin`, respectively. `docs.bin` is a file of concatenated text documents. When `main` is executed and the aforementioned files are present, the program loads all the data in those files into the tree. Otherwise, the program simply creates a new empty tree.

Handout continues on the next page(s)

2.4 Functionalities

As mentioned above, our B^+ tree implementation stores words as keys. Since the tree enables us to retrieve documents given words (as opposed to finding words in given documents), the tree is referred to as an *inverted index*.

When executed, `main` loops indefinitely, accepting and processing commands from the user (see `src/main.c`). Table 1 below summarizes all the supported commands and their corresponding outputs:

Command	**Output**
`C`	Prints all the keys that are present in the tree, in asCending lexicographical order.
`i <doc>`	Parses the text in `<doc>` 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 `<doc>` make the "keys" of the B^+ tree, and the value for all the keys is set to the text of `<doc>`.
`p <num>`	Prints the keys in a particular `page` of the B^+ tree where `<num>` is the page number. It also prints some statistics about the page such as the number of bytes occupied, the number of keys in the page, etc.
`s <key>`	This searches the tree for `<key>` where `<key>` is a single word. If found, the program prints "Found the key!" and if not, it prints "Key not found!"
`S <key>`	This Searches the tree for `<key>`. If found, the program prints the text of all documents in which the key is present, also known as the posting list of `<key>`. If not, it prints "Key not found!"
`T`	Prints the tree in a neaT in-order format! If the tree is empty, it prints "Tree empty!" instead.
`q`	quits the program.

To see a demo of how the tree works, you can type `make demo-movies`. The demo inserts all the text documents in the `movies` dataset into the tree (using the command `i`), then prints the resulting state of the tree in an in-order format (using the command `T`), and, finally, searches for the word `"american"` (using the command `S`). As described in Table 1, the command `S` will print the text of each document containing the word `"american"`.

3 Additional Functionalities

In this project, you will implement two additional commands as described in Table 2 below:

Command	**Output**
`f <key1> <keys2>`	Print in *alphabetical order* (`f`orward) the distinct keys that are in the range defined by `<key1>` and `<key2>` (including the bounds). If `<key1>` and `<key2>` are not in alphabetical order, print "Invalid key order!" If no documents have keys within the given range, print "Keys in the given range not found!"
`b <key1> <keys2>`	Print in reverse alphabetical order (backward) the distinct keys that are in the range defined by `<key1>` and `<key2>` (including the bounds). If `<key1>` and `<key2>` are not in alphabetical order, print "Invalid key order!" If no documents have keys within the given range, print "Keys in the given range not found!"

Implement the two aforementioned commands in `src/api/range_search.c`. Note that for the command `b`, you are not allowed to store the output of the command `f` in an array and subsequently print it.

For this purpose, you should read and understand, in particular, the following header and source files:

- `common/common.h`: page (node) and key structure definitions.
- `api/search.c`: API for looking up a key in the index.
- `btree/search_tree.c`: recursively search the tree for a key.
- `btree/search_leaf.c`: search for a key in a leaf node.
- `btree/find_child_page.c`: to traverse down the tree for a key lookup.

Your solution code may call or adapt any of the existing functions or structures in the original source code, but you may not modify them in any way.

4 Testing

For your convenience, we have provided you with sample tests and their corresponding outputs in the folder `tests`. To see if your implementation of the commands `f` and `b` run correctly on the test files, type:

- `make test-range`
- `make test-range-rev`

Each test matches your solution output with our reference output and if there is no difference, your implementation will pass the test. In addition to the provided test cases, you must devise tests (of your own) that run on different datasets (i.e., documents) of your choice and different argument values. Also, consider corner cases like invalid inputs, non-existent words, etc.

5 Getting Started

To jump-start your implementation, you should:

- Run the demo and make sure you understand the tree structure.
- Study the important data structures defined in `common.h`.
- Understand how the basic search (in `btree/search_tree.c`) works.

6 Final Deliverable

In an archive (named `P3_<AndrewID>.zip`) add your modified `range_search.c` (in addition to any other supplementary `.c` files that are part of your solution). If you will alter the `Makefile`, please make sure to include your updated version as well.

7 Getting Help

You can get help by visiting the professor and the TA during their office hours or by appointment. You can also post your questions on Piazza [[1]].

[1] `piazza.com/qatar.cmu/spring2020/15415`