# CS6360.004 Fall 2019

# Assignment 6 – B+ Tree Index

Your assignment is to write a program that implements B+ tree indexing.  It will read a text file containing data and build the index, treating the first n columns as the key, **where n will be specified by the user**.  Writing this program will require knowledge of random-access I/O.  The only allowable language is C/C++.   
  
The structure of the index is as follows: First 256 bytes: Name of the text file you have indexed.  This must be blank-filled on the right. You may need other metadata in this first block, so I suggest you allocate 1K so you can read it in as a block. (The size of the key is one such piece of metadata; see **1** below.)  The rest of the file is 1K blocks of index data, according to the way a B+ tree is structured.  This implies that your program should read in a block of data, manipulate it, and possibly (if you are inserting key/pointer pairs) write it back out **as a block**.  Use a long (8-byte) record address for your pointers.  These "pointers" are the byte offset in the text file of the data record. Thus the first record is at offset 0, the second is at offset of 0 plus the length of the first record, and so on.  Note that the structure mixes text data (the key) and binary data (the pointer.)

**Some conditions:** Converting the pointer to text digits is not allowed.  Reading everything into memory, doing the manipulation, and writing it out is also not allowed because this program could potentially handle millions of records. You should buffer at most 3 disk blocks of the index file. This is because, if you’re inserting, you will need to read a node, and if you must split it, you’ll need another 1K block for half of the keys, and possibly a third one if you must create a new root.  
  
This program should run entirely from the command line.  Each of the functions and their command-line parameters are given below, assuming the name of your program is INDEX:  
  
**1. Create an index.** This takes four parameters:

1. -create
2. The name of a text file to be indexed. Use the file I have provided.
3. The name of an output index that is created. Use the extension .indx for this.
4. How many bytes of the record, starting from the first position, are considered to be the key. Note: **DO NOT** hard-code this number in your program. The code must use whatever number is provided on the command line as long as it is between 1 and 40, inclusive. If the record contains fewer characters than the key length, pad the key on the right with blanks.

If the output file exists, overwrite it.  Do not ask.  Build the index by starting with a blank structure and inserting new keys.  There may be duplicate keys in the input file.  Your program should list them and the line number of the input file on which they occur.  Duplicates should not be inserted into the B+ tree.  The command line is:

 INDEX -create <input file> <output file> <key size>

For example: INDEX -create CS6360Asg5Data.txt CS6360Asg5.indx 15

--Creates an index with a key length of 15 bytes.

INDEX –create CS6360Asg5Data.txt CS6360Asg5.indx 21  
--Creates an index with a key length of 21 bytes.

**2. Find a record by key.** This displays the entire record, including the key, and gives its position, in bytes, within the file.  If the key is not in your index, the program must give a message to that effect.  As you can guess, the reason for having the name of the text file in your B+ tree index is so you can find it for this and the subsequent command.  The command line is:

 INDEX -find <index filename> <key>  
  
for example, if you ran this command line:  
 INDEX -find MyIndex.indx 11111111111111A  
  
your program would display a message that the key was not found.  The command line   
 INDEX -find MyIndex.indx 64541668700164B  
  
would display:  At 2127, record:  64541668700164B ANESTH, BIOPSY OF NOSE  
  (This presumes that the byte offset of this record from the start of the file is 2127).

If the key you supply is longer than the key length specified, your program should truncate it. If it is shorter, pad it on the right with blanks.  
  
**3. Insert a new text record.**  As with creating a new index, the first <n> bytes must contain a unique key. You will get the number of bytes by reading the header record in the index file.  If the key is not in the index, write the record at the end of the data file (remembering the position, since you will need it), then inserting the key into your index structure with the pointer to the start of the new record.  Display a message indicating success and the position of the new record in the text file.  If the key is already in the index, display a message to that effect and do not insert the record.  The program must insert the key into the index and the record into the text file.  The record needs to be in quotes because it could contain spaces and other punctuation.  This has the following command line:  
  INDEX -insert <index filename> "new text line to be inserted."   
  
For example, the following command line would attempt to insert a record into the file.  From the previous example, you know its key is already in your index, so you would get an error message:  
  INDEX -insert MyIndex.indx “64541668700164B Some new Record"  
  
This command line would return success:  
    INDEX -insert MyIndex.indx "11111111111111D Some new Record"  
  
  
**4. List sequential records.**  The program must show the record containing the given key and the next n records following it, if there are that many.  If no record exists with the given key, show records that contain the next-larger key and give a message that the actual key was not found.  As with the -find function, show the records in order by key and their position within the text file.  Obviously this must traverse the tree in key order, which is trivial if your B+ tree is correct.  Here is the command line:

  INDEX -list <index filename> <starting key> <count>

**Hint:** It really helps to have a hex file dump utility when you're working on something like this, since the file you create is not all text.

Also on eLearning is a script you can use for testing your program. If it runs correctly with the script and does not crash, and your program follows the other requirements, you’re assured of at least an 80. (My apologies for the nature of the data, but those are actual codes and medical procedures from a publicly-available source.)

**This is an individual project. Do not collaborate, and do not use code you find on the Web. None of it meets requirements.**

**To hand in through eLearning:** Your C/C++ source files, including .cpp and .h files, as a Zip file.  Also **include the index** created by your program. Name the Zip file using your NetID and the name of the assignment. If I were to hand this in, it would be jxc064000Asg6.zip. Also hand in the script modified to run your program so that we can run your code using the name you gave the executable or Java file.

|  |  |
| --- | --- |
| **Grading Criteria:** | |
| The program works correctly and follows the criteria set forth above.  Your program must not crash for bad data. | 80 |
| Good, clean structured design. Write small functions to perform basic operations. Don’t code everything in *main*. | 10 |
| Program comments.   An undocumented program will lose all 10 points here.  If you're not sure what good comments are, read my description linked on my Notes for Students Web page.  Part of this is using good variable names. | 10 |

**Some grading guidelines:**

1. Hard-coding the key length: -30
2. Using decimal digits instead of binary numbers for record pointers: -20
3. Building the entire structure in memory, then writing it to disk (This generally occurs together with the next problem): -50
4. Not writing the index using blocks of memory (That is, if your code does not put key/pointer pairs into an array of 1024 bytes): -50
5. Using sorting functions and/or collection classes. There is no need to sort anything if you use the algorithms defined in the textbook and discussed in class. -30