Data Structures **Name:** ____________
Fall 2016, Prof. Bolton **Net ID:** ____________
Project 2

1 B-Tree

1.1 Summary

You will design and implement a B-Tree Structure. The application of the structure will include spell checking and word counting. Words in an inputFile will be sequentially investigated and compared to a comprehensive list of correctly spelled words – a lexicon. The lexicon will stored in a b-tree structure to provide for a reasonably efficient search time. (You will also store the lexicon in a linked list for comparison purposes). The details of the design of the B-tree structure must conform to the standard constraints of a B-Tree as discussed in class. Major factors to consider for the design should be theoretical efficiency (both time and space), practical efficiency, and appropriate memory management. Although some tips are provided below (and have been provided in class), the determination of *how* to make the structure efficient is largely your charge.

1.2 Programming Languages

I encourage you to submit your projects using C++. You may choose a different programming language with prior approval from me. However there are caveats as not all programming languages have the same characteristics. Also note, if choosing a language other than C++, you may by chance choose a language which the TAs are not familiar, thus limiting the amount of assistance they might provide. Note one of the main goals of our class projects is for you to learn how to construct various data structures from the most *elemental programming constructs*. Thus you will not receive credit when using any pre-existing structures from programming libraries or code that has been created or designed by others. For example in C++ you **cannot use** pre-existing such as vectors, stacks, lists, trees etc. To help facilitate io and timing, you may use ctime, string, ifstream, iostream, and string stream.

Please note that complex data structures (non-elemental constructs) are "built-into" some programming languages. If this is the case, you cannot use the built-in structure. For example, in Python, both list and "array" structures are fairly complex and not elemental programming constructs, e.g., they can change size dynamically. If you are using Python, you will not receive credit when using these structures.

If you have any questions as to what structures are permitted (and which are not permitted), given your language of choice, please ask me.

1.3 Input Requirements

The program will take three command line arguments **(Do not prompt the user)**.

```
./project2 [inputFile] [dictionaryFile] [removalFile]
```

Argument 1 is inputFile: the name of a text file to be spell-checked. Argument 2 is dictionaryFile: the name of a text file that contains all correctly spelled words (a lexicon). Argument 3 is removalFile: the name of a textfile that contains words to be removed from the lexicon – see Output Section for details.

1.4 Structural and Operational Requirements

1. Linked List (for relative comparison)
   - Creates a linked list class that will reorder the elements in the list, based on frequency of access. Specifically, after each retrieval from the list, the item retrieved will be relocated to the front of the list.
2. B-Tree Implementation
   - Appropriately designs a b-tree structure to allow for the creation of a b-tree of order m (for any non-negative int m).
   - Must provide for basic b-tree operations: insert, remove, ...
   - Appropriately allocate / deallocate memory
   - Valid state of a b-tree is always maintained
3. Application: Spell Checking
   - The program sequentially investigate each word in inputFile and confirm that this word exists in the lexicon. If so, it is assumed the word is spelled correctly; if not, the word is spelled incorrectly. The lexicon will be stored in a b-tree.
   - The program will keep a count of the number of misspelled words in inputFile.
4. Application: Word Counting
   - The program will count the frequency of occurrence of all unique words in inputFile. (**Hint**: it may be convenient to keep this count in each node.) For example, if the input file contained only sentence "The cat in the hat is in the flat."; the word frequency count would be as follows:
     - the: 3
     - cat: 1
     - in: 2
     - ...

Notes: To earn full marks it is expected that you will make the b-tree structure very efficient (in space and time).

1.5 Output Requirements

The main method should read in the inputFile and dictionary

The program should then execute the following operations, provide the following output, and then exit normally:

1. Store lexicon in a linked list. Reorder the linked list based on frequency of access (when an item is accessed, move that item to the front of the list). Keep track of the number of misspelled words.

   - **Print to the screen, the total number of misspelled words in inputFile.**
   - **Print to the screen, the runtime during the spell checking. (How – see notes below.)**
   - **Print to screen the word with the highest frequency of occurrence. Also print its corresponding word count.**

2. Store lexicon in a b-tree of order 2. Perform spell check of inputFile. Keep track of the number of misspelled words.

   - **Print to the screen, the total number of misspelled words in inputFile.**
   - **Print to the screen, the runtime during the spell checking. (How – see notes below.)**
   - **Print to screen the word with the highest frequency of occurrence. Also print its corresponding word count.**

3. As noted in class, the choice of order m of a b-tree will directly affect the height of the b-tree, thus directly affecting search time. Experiment with some range of values m ($m \neq 2$) to determine which value of m leads to the quickest runtime (approximately) for the spell checker (given the sample inputs provided). In your cover letter discuss why you believe this particular value of m was a good choice (both practically and theoretically) given the sample inputs provided.

   - **Print to the screen, the total number of misspelled words in inputFile, when using a b-tree of order m, where m was chosen based on your experimental findings.**
   - **Print to the screen, the runtime during the spell checking.**
   - **Print to screen the word with the highest frequency of occurrence. Also print its corresponding word count.**

4. Remove all words contained in the removalFile from the lexicons (the b-trees and linked list) used above in ouput steps 1 - 3. Repeat output steps 1 - 3 (and print corresponding outputs 1 - 3).

Notes: To compute the runtime of your spell checker, use the tools provided in ctime header. For usage, see example posted. "Begin" the timer when the spell checking begins – when the program begins to traverse inputFile, (after tree creation). "End" the timer when the spell checking is complete.

No other outputs should be observed.

1.6 Submission and Compilation Requirements

You must fill out a cover letter for all project submissions – See BB. Submission Deadline – See BB. Budget your time well. Include significant time for design / planning and testing / debugging. Please submit early and often! Your last submission (before the end of the grace period) will be graded.

Your code must compile and run on the class server.

As each of your designs may be drastically different and you may organize your code into multiple files of various name, you must include clear and precise compilation instructions in the cover letter (e.g. list out the exact g++ command that should be entered including all pertinent file names). (No Makefiles necessary.)

If the program does not compile (following your instructions) or the program does not run to completion or exits abnormally, the submission will not be accepted, and the resulting grade will be 0.

1.7 Testing and Debugging (Optional)

You may wish to construct an interactive interface to test the functionality of your structure at intermediate stages of development. This would likely be most efficient with an interactive interface that allowed you to interactively test various functionalities of your structure given different inputs. *If you do implement a testing interface, please be sure to comment it (so that it does NOT execute) before submission.*

1.8 Rubric

Efficiency should be considered.

List of Requirements	**Percentage**
b-tree implementation	0.50
linked list (based on frequency of access) implementation	0.10
spell checking (traversal of input and lexicon)	0.10
word frequency (traversal of input and lexicon)	0.10
choice of m for part 4	0.10
main method conforms to specs	0.10
TOTAL	**1.0**