CS 341 – Fall 2020
Assignment #3 – Duplicate Dawgs
Due: 10/13/2020

Have you ever been in a position where you have a large data set and you wish to know whether a specific number is a member of the set? Obviously, you could scan the list – but this is very expensive in terms of memory. Instead, you want to optimize this process and use your space efficiently. The Linux kernel actually makes use of this approach to provide optimization. This third assignment will explore a new data structure – the Vector, and see how we can leverage it to explore bit manipulation in C++ and provide such optimization.

For this assignment you will be analyzing a large data set consisting of Integers. Our faithful CEO (Blue IV) has again entrusted us with a task to improve his existing data. This time around Blue IV has given us a very LARGE data set comprised of Integers. These Integers are not ordered and are machine generated log entries for memory access. Blue IV wants to see who the biggest memory hogs are in the machine!

His task for you, is to find all the duplicate entries in the set of numbers. Sounds easy enough…but we know Blue IV, he may be cute and cuddly on the outside but he always is lurking with a catch just around the next corner and here it is: he has been reading up on this great new class at Butler, called CS 341 – Advanced Data Structures, and has learned about this concept of a Bit Vector. A Bit Vector can be defined as follows:

"A bit vector is a mapping from a domain to values in the set {0, 1}"

*Why a Bit Vector – by capitalizing on this mapping we can efficiently store the same information using the minimum number of bits necessary – this optimizes space consumption. The '0' represents the condition where the bit is NOT set and the '1' represents the condition where the bit IS set. The integers we are using required 32 bits of memory for storage.

Now Blue IV has thrown down the gauntlet – he wants us to accomplish all of this in 4 KB of memory (Why? "4 is my lucky number!"). If that wasn't enough (really Blue IV!?), he wants to not only have the list of duplicate values but also ORDER the new list by frequency – meaning that he would want to first element in the list to be the most frequently occurring number and so on and so forth. Are you up to the challenge!?

A few notes about the specific requirements of the program:

- The program should load SPACE delimited numerical data from a text file (`data.txt`).
- The program should then insert the data into a Bit Vector.
    - The duplicates should be identified and placed into a new vector.
- The program should then output to a file (`duplicates.txt`) a list of duplicates listed in descending order in terms of frequency.
- The program should handle invalid cases (e.g., invalid text entry, file I/O, etc.).
- The Bit Vector should be placed on the Heap (memory management).
    - The program should contain no memory leaks – make sure to use Valgrind!
        - `valgrind --log-file=valgrind.txt A3.exe`

Development Process:

For the development of your code: you should create `BitVector` and `Dictionary` Classes. The `BitVector` Class should have the following attributes and operations:

- Private:
  - `std::vector<int> * data_`
    - `// This will hold the data from the text file`
- Public:
  - Constructor(s)
  - Destructor
  - Accessor Method(s)
  - `bool getBit(int position);`
    - `// Gets the bit at the given position`
  - `void setBit(int position);`
    - `// Sets the bit at the given position`
  - `void findDuplicates(std::vector<int> data, std::vector<int> & duplicates)`
    - `// This will identify the duplicates via our bit vector and store the list of them in another vector`

For the `Dictionary` Class you should include the following attributes and operations (Note: We can reuse our `Node` Class here through Aggregation):

- Private:
  - `Node id_`
    - `// This will hold number entry`
  - `Node data_`
    - `// This will hold the frequency value`
- Public:
  - Constructor(s)
  - Destructor
  - Accessor Method(s)

Tying this all together you will need to write a `driver` that will test this hierarchy and provide the necessary functionality as described earlier as outlined by Blue IV. Your driver program should allow for the user to input a space delimited text file (`data.txt`) to be loaded and built into our Bit Vector and then with the final resulting duplicate sorted list by frequency being output to another text file (`duplicates.txt`). Finally, you will need to create a `makefile` that properly links all of this code together and creates an executable named **A3.exe**

An example of sample output is as follows, given the following sample input file:

```
29 7 37 47 16 34 22 47 7 3
```

Your `duplicates.txt` file would contain the following:

```
47 2
7 2
```

I will be grading what is located in the **master branch** of your GitHub repository. It is strongly recommended that you commit and push often! Please make use of the Git feature of leaving descriptive messages along with your commits. Be sure to add me to any repository as a collaborator so I can view and grade your submissions. Failure to do so will result in a 0 on the assignment as I will have nothing to grade!

Submission:

All assignments must be submitted on Butler GitHub (github.butler.edu). I will allow you to work on this assignment with up to one (1) other person. If you choose to work on this project with a partner you need to email me letting me know of your "group." Be sure to make note of specific contributions of each team member so I can assign grades accordingly. **Note:** You are NOT required to work with a partner. The name of your Butler GitHub repository must be as follows: **cs341_fall2020_bits**

The directory structure of the repository must contain the following files:

- **driver.cpp**
- **BitVector.h**
- **BitVector.cpp**
- **Dictionary.h**
- **Dictionary.cpp**
- **Node.h**
- **Node.cpp**
- **data.txt**
- **duplicates.txt**
- **makefile**

Each source file (`.cpp/.h`) **must** include the Honor Pledge and digital signature – if working in pairs, please ensure both students' digital signatures are present on the files.

Helpful Hints:

Bitwise Operations

& (Bitwise AND)	Takes two numbers as operands and performs an AND on every bit.
| (Bitwise OR)	Takes two numbers as operands and performs an OR on every bit.
^ (Bitwise XOR)	Takes two numbers as operands and performs a XOR on every bit.
<< (Left Shift)	Takes two numbers and left shifts the bits of the first operand by the number of places specified by the second operand.
>> (Right Shift)	Takes two numbers and right shifts the bits of the first operand by the number of places specified by the second operand.
~ (Bitwise NOT)	Inverts all the bit of one operand.