CS 341 – Fall 2024
Assignment #3 – Just A Lil' Bit

Due: 10/16/2024

This assignment is going to introduce you to a new data structure, a Bit Vector, and explore how we can leverage existing data types to store/encode additional information. On most typical machines the architecture is constructed such that your standard primitive integer is stored in either two (2) or four (4) bytes. That means in some instances to represent the number zero (0) we need a total of thirty-two (32) zeroes (0's) to achieve this….THAT seems a ***bit*** wasteful (← see what I did there! ☺). We are going to see in this assignment that we accomplish the same end goal with one (1) single bit!

This assignment will be broken into distinct phases (five (5) to be specific) in order to help you progress and track your progress. Please make sure that you complete each phase in-turn before moving on to the next (these phases are for YOUR benefit so that way you are not biting off more than you can chew). This project MAY be worked on in teams of two (2) – we will discuss "good" pair programming practices in lecture.

Additionally, for this project we are going to focus on good Software Engineering techniques and methods to accomplish our task. I have provided you the basis for the assignment on Canvas (`Files` → `Assignments` → `A3`). You are responsible for downloading the associated files and modifying them as necessary. **NOTE:** You may NOT change any of the existing code (e.g., you cannot change a method to return a void when it was originally written to return a char). Make sure to review any feedback from Assignment #2, with respect to proper programming practices, before moving on to Assignment #3.

Development Process:

Phase I – 10/4/2024

Your task is to create `bitarray.cpp` to finish the implementation of the class `BitArray`. The `BitArray` class is an implementation of a fixed-length bit vector data structure. This means that once we have created a `BitArray`, it will remain that size for the lifetime of the object. For Phase I you will be implementing the following methods:

```
BitArray(int size)
BitArray(const BitArray & array)
~BitArray()
get(int position)
```

Don't forget to implement the Destructor – you don't want any memory leaks in your code! In this phase you will be introduced to the concept of a Copy Constructor. C++ mandates that any Class that contains dynamically allocated memory MUST contain a Copy Constructor – we will talk more about WHY in lecture. Once you have completed this you can then create a test driver (`driver.cpp`) to test your `BitArray` class. Sample output of this is shown below:

```
Number of Bits: 32
|01110100|01100101|01110011|01110100|
```

If you have successfully completed this – you are ready to move on to Phase II.

Phase II – 10/7/2024

For the second phase of this assignment, you are asked to complete the implementation for our `BitArray` class. You will need to move the code that you used to initialize the `BitArray` in Phase I into your initialize method here. There is one additional method in this class that you will not be implementing fully in this phase – however, you will need to stub it out in order to get your program to compile. You are NOT allowed to use the `BitSet` library in any of your code (I have provided the print method implementation that contains the ONLY valid use of the `BitSet` library).

You will need to leverage your understanding of bitwise operators to complete this phase. For a refresher:

- | ← OR
  - 0011 | 1000 = 1011
- & ← AND
  - 0011 & 0101 = 0001
- ^ ← EXCLUSIVE OR
  - 0101 ^ 0110 = 0010
- << ← LEFT SHIFT
  - 0001 << 1 = 0010
- >> ← RIGHT SHIFT
  - 0100 >> 2 = 0001

You will to complete the following methods:

```
initialize(char * word, int size)
flip(int position)
set(int position, int bit)
get8(int position) const
complement()
clear()
```

Sample test output of this phase should look as follows:

```
|01110100|01100101|01110011|01110100|

0th Bit Set: False
10th Bit Set: True

Setting 0th position to 1...
|11110100|01100101|01110011|01110100|

Setting 10th position to 0...
|11110100|01000101|01110011|01110100|

Complement:
|00001011|10111010|10001100|10001011|

Clear BitArray:
|00000000|00000000|00000000|00000000|
```

If you have successfully completed this – you are ready to move on to Phase III.

Phase III – 10/10/2024

For the third phase of this assignment you are tasked with providing an implementation (`set.cpp`) for the `Set` class. This class is designed to model the mathematical concept of a set. The core functionality of `Set` requires you to implement functions that affect the entire `BitArray`, rather than single bits as performed in the previous phase. Additionally, you will need to revisit your `BitArray` class and implement the `set8` method. This will allow you to update a character within your bit array.

Sample test output for this phase is shown below:

```
|01110100|01100101|01110011|01110100|
|01110010|01111001|01100001|01101110|

Cardinality: 17

Set Union (A U B):
|01110110|01111101|01110011|01111110|

Set Intersection (A X B):
|01110000|01100001|01100001|01100100|
```

If you have successfully completed this – you are ready to move on to Phase IV.

Phase IV – 10/13/2024

For the forth phase of this assignment, you are tasked with providing the implementation (`dictionary.cpp`) for the `Dictionary` class. In this phase you will be implementing the following four (4) methods in addition to the Constructor(s), Destructor, and `initialize` method:

```
rank_range(int start, int end, int bit)
select_range(int start, int end, int k, int bit)
rank(int end, int bit)
select(int k, int bit)
```

Details on the behavior of each of these methods can be found in the comments of the Header file. For this phase we will implement the non-optimized version of both rank and select.

In Phase V we will return to these two methods and see how we can optimize the rank and select process.

Sample test output for this phase is shown below:

```
|01110100|01100101|01110011|01110100|
Rank: 17
Rank Range (15-31): 9
Select(5,1): 8
Select Range(0,15,2,1): 3
```

Phase V – 10/16/2024

For the fifth phase of this assignment, you are tasked with the responsibility of improving the overall efficiency of your design. You may have noticed that when writing the rank and select methods in Phase IV you likely used a loop and did some basic computation to arrive at the answer. This style of code runs in linear time `O(n)` in the worst case, since someone could conceivably set `i = 0` and `end = n`.

We want our code to run faster in practice, so we're going to modify our data structure to get a speedup of **8x**. Let's explore how this is possible: let's assume that we have a 32 bit array – if we are searching the entire bit array this requires 32 calls to our `getBit` method. If we create a Lookup Table storing the number of 1's for all possible byte combinations we can turn those 32 calls into just 4!!!! Now…what is this Lookup Table? It will store each value, in terms of the byte notation, and the number of 1's in said byte. For example:

`lookupTable_[00000000]`	`lookupTable_[0]`	0
`lookupTable_[00000001]`	`lookupTable_[1]`	1
…	…	…
`lookupTable_[00000100]`	`lookupTable_[4]`	1
`lookupTable_[00000101]`	`lookupTable_[5]`	2
…	…	…
`lookupTable_[10101011]`	`lookupTable_[171]`	5
`lookupTable_[10101100]`	`lookupTable_[172]`	4
…	…	…

Now – we may still have to do some additional checks, to our `getBit` method, if we aren't dealing with a FULL byte, but this still provides optimization!

Sample test output for the following phase is shown below:

```
|01110100|01100101|01110011|01110100|

Rank: 17
Select(5,1): 8

Printing Lookup Table...
```

Once you have completed this task successfully you have successfully finished the assignment!

Submission:

All assignments must be submitted on Butler GitHub (`github.butler.edu`). **Note:** If working in teams of two (2) – both students must submit the code to their individual repository.

The directory structure of the repository must contain the following files – please note the naming:

- **driver.cpp**
- **bitarray.h**
- **bitarray.cpp**
- **set.h**
- **set.cpp**
- **dictionary.h**
- **dictionary.cpp**
- **valgrind.txt**
- **makefile**

It is **STRONGLY** recommended that you commit your changes after the completion of each phase – this will not only serve as a "checkpoint" for your progress but will also allow me to award partial credit in the case that a specific phase is not completed successfully.

** Don't forget: Each source file (`.h`/`.cpp`) **must** include the Honor Pledge and digital signature. **
– **NOTE:** For teams of two (2), both signatures must be present.