

# The University of New South Wales

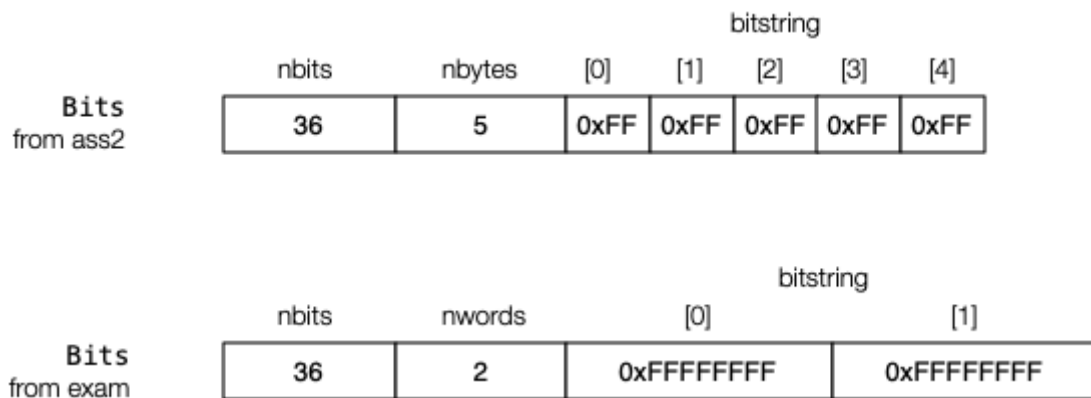
## COMP9315 DBMS Implementation

### 21T1 Final Exam

[\[Instructions\]](#) [\[PostgreSQL\]](#) [\[C\]](#)  
[\[Q1\]](#) [\[Q2\]](#) **[\[Q3\]](#)** [\[Q4\]](#) [\[Q5\]](#) [\[Q6\]](#) [\[Q7\]](#) [\[Q8\]](#)

### Question 3 (20 marks)

In Assignment 2, you wrote signature-based indexing schemes. A critical component of these was a `Bits` ADT for arbitrary-length bit-strings. In the Assignment, the implementation of bit-strings was based on an array of 8-bit bytes. In this question, we plan to re-implement the `Bits` ADT so that it uses an array of 32-bit words (`unsigned int`) to hold the bit-string. The following diagram shows the difference between the two implementations:



Note that all of the bits (40 in the byte-based version, 64 in the word-based implementation) are set to 1, even though only 36 bits are required. The value of the "extra" bits is irrelevant, since these bits should be ignored, and only bits  $0..nbits-1$  should be accessed.

We have supplied a partly complete implementation of the `Bits` data type in the file `bits.c`. This contains seven functions, four of which are complete:

- `newBits(int n)` ... creates a new `Bits` object containing `nbits` bits
- `freeBits(Bits b)` ... releases memory associated with a `Bits` object
- `setBit(Bits b, int i)` ... set the  $i^{\text{th}}$  bit in bit-string `b` to 1 ( $0 \leq i < nbits$ )
- `setAllBits(Bits b)` ... set all bits in the bit-string to 1
- `unsetBit(Bits b, int i)` ... set the  $i^{\text{th}}$  bit in bit-string `b` to 0 ( $0 \leq i < nbits$ )
- `unsetAllBits(Bits b)` ... set all bits in the bit-string to 0
- `showBits(Bits b)` ... display a `Bits` object as a sequence of 1's and 0's

The functions `newBits()`, `freeBits()`, `setAllBits()`, and `unsetAllBits()` are complete and you should *not* modify them. Note that `setAllBits()` sets all bits in the bitstring array, even though not all of them are used. You should ignore the "extra" unused bits.

**Your task:** you must complete the `setBit()`, `unsetBit()` and `showBits()` functions. Note that `showBits()` should show *only* the `nbits` bits in the bit-string, in the order `nbits-1 .. 0`.

Like the assignment, the words in the array contain the low-order bits in `bitstring[0]`, up to the high-order bits in `bitstring[nwords-1]`. Within a word, the low-order bits are at the right-hand end of the word and the higher-order bits are at the left-hand end.

To test the `Bits` ADT, we have supplied a driver command called `./bs` (in the file `bs.c`). The `./bs` command is invoked as follows:

```
$ ./bs Nbits +|- mods
```

where `Nbits` is (suprise!) the number of bits in the bit-string. The second argument can be "+", in which case all bits are initially set to 1, or "-", in which case all bits are initially set to 0. This can be followed by an arbitrary number of *modifiers*. Each modifier contains a bit position, preceded by "+" or "-". If "+", the specified bit is set to 1; if "-", the specified bit is set to 0.

An example:

[illegible]

This creates a 40-bit bit-string and initialises all 40 bits to 1. It then displays this as the initial (`init:`) value, using `showBits()`. It then unsets bits 10, 20 and 30 before setting bit 20 back to 1. After each modification, it displays the mod and then displays the modified bit-string (using `showBits()`).

The argument processing and display of each mod is handled from within `bs.c`, whose code you should *not* change.

There are other examples of using `./bs` in the `tests` directory. You should also be able to devise your own test cases easily enough.

To help you check whether your program is working correctly, there is a script called `run_tests.sh` which will run the program against all of the tests and report the results. It will also add the output from your program into the `tests` directory; comparing your output against the expected output might help you to debug your code. You can run the testing script as:

```
$ sh run_tests.sh
```

Once your function is working (passes all tests), follow the submission instructions below. Even if it fails some (or even all) tests, you should submit because you can get *some* marks. If your program does not compile, or if you simply submit the supplied code, then your "answer" is worth zero marks.

### Submission Instructions:

- Type your answer to this question into the file called `bits.c`
- Submit via: **give cs9315 exam\_q3 bits.c**  
or via: Webcms3 > exams > Final Exam > Q3 submission > Make Submission

*End of Question*