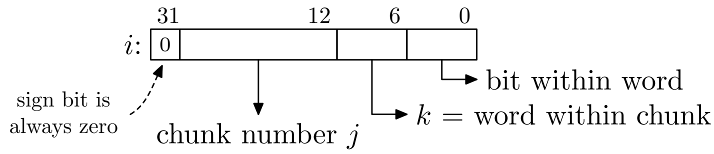
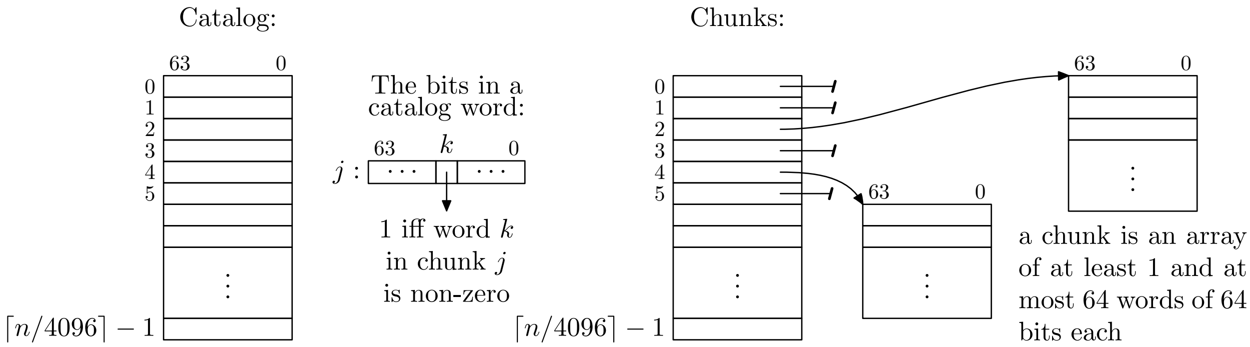
Sparse Fixed Bit Set

Here is a useful data structure for representing sets of at most n bits identified by non-negative integer numbers i. It saves on memory by dividing bit space into chunks of 64 words à 64 bits and storing only non-zero words, assuming all other words to be zero. Given a bit number i, the least significant 6 bits are the number of the bit within the word, the next 6 bits are the number k of the word within the chunk, and the remaining (most significant) bits are the chunk number j.



Both the size n and bit numbers i are represented by 32 bit integers (`int` in C#). Since only non-negative integers are used (sign bit is always zero), the size is at most $2^{31} = 2\,147\,483\,648$ bits. The structure uses two parallel arrays of size $\lceil n/4096 \rceil$, the "catalog" (holding 64 bit words) and the "chunk list" (holding pointers to the chunks). Chunks are arrays of up to 64 words of 64 bits, that is, type `ulong[]` in C#.



To determine if bit i is set (`true`) of clear (`false`), proceed as follows:

1. assert $0 \le i < n$
2. let $j = i \gg 12$, the index into `catalog` and `chunks`
3. let $k = i \gg 6$, the word number within the chunk
4. let $f = \texttt{catalog}[j]$, the chunk's 'non-empty' flags
5. if $(f \& ((1 \ll k)) = 0$: return `false` (word k in chunk j is all zero and `chunks`$[j]$ is `null`)
6. let $o = \mathrm{pop}(f \& ((1 \ll k) - 1))$, the number of 1 bits right of bit k in f
7. let $w = \texttt{chunks}[j][o]$, the word of bits
8. return $(w \& (1 \ll i)) \neq 0$, extract bit $i \& \texttt{3F}_{16}$

This procedure assumes that only the low 5 (for `int`) or 6 (for `long`) bits of the shift count are used, such that, for example, $x \ll 32 \equiv x \ll 0$ and $x \ll 33 \equiv x \ll 1$ etc. This is true of both C# (CLR) and Java (JVM). If this is not true, high-order bits must be masked away explicitly: let $k = (i \gg 6) \& \texttt{3F}_{16}$ in step 3 and return $(w \& (1 \ll (i \& \texttt{3F}_{16}))) \neq 0$ in step 8.

Pop is the "population count" function that returns the number of 1 bits in its argument. For example, $\mathrm{pop}(5) = 2$ because $5_{10} = 101_2$. This can be efficiently implemented (or is provided as a CPU instruction). The idiom "$(1 \ll k) - 1$" creates a bit mask where all bits right of bit k are 1 and all other bits are zero; for example, $(1 \ll 4) - 1 = 1111_2$.

Setting and clearing bits is similar, but involves allocating/releasing chunk arrays and adding words to / removing words from the chunk arrays.