

簡潔データ構造 第2回

marimo

2023 年 4 月 25 日

方針

実は現実のマシンと Word-RAM は全然違う

- メモリは word 単位でしか読めない
 - もっと言うと読むのはライン単位
- 表引きは遅い
 - RAM アクセスは 10ns ぐらいかかるが 1 クロックは 0.5ns ぐらい
- もっと多くのことが 1 命令でできる
 - word 内の rank,select はできる

あと説明した select の索引は実際に使う程度のサイズでは大きすぎるので工夫しましょう

64bit のマシンを使う限り最大ケースは 2^{64} bits のときなので、このときに大丈夫なら大丈夫です

メモリの読み方

- Word-RAM はメモリを任意の bit から w bits だけ読み出すことができた。
- 実際のプロセッサはメモリはワードサイズ単位とかバイト単位とかでしか読めないで、小ブロックがワードをまたがると実行速度上の問題がある。
 - 一方でバイト単位にすると索引サイズが大きくなるので困る。

紹介した rank の索引では $n = 2^{32}$ だと大ブロックは $l = 2^{10}$ の長さになり、小ブロックの配列に入る整数は 10 bits で困る。そこで大ブロックの長さを 2^{16} ぐらいに固定してやると小ブロックで持つ整数が 16bit でちょうど持ててよい。

索引のブロックサイズを大きくする分には索引が小さくなるだけなので困らない

表引きは遅い

使う表は $\Omega(\sqrt{n} \log \log n)$ ぐらいでかなり大きくなりがちなのでキャッシュに乗らない

RAM にアクセスすることになるがこれは 10ns ぐらいかかる

一方で bit 演算とかは平均すれば 1 命令 0.5 クロックぐらいでできる (およそ 0.3ns) なので, bit 演算で頑張った方が速くなることが多い

表を使わない場合, 小ブロックを $\frac{1}{2} \lg n$ bits にする必要がなくなるので小ブロックをもう少し大きくても大丈夫になる。具体的にはワードサイズぐらいか, キャッシュのラインサイズを考慮して 256 ビットぐらい?

CPU の拡張命令

現代の CPU は実は word-size の四則と bit 演算と少しの制御よりももっと沢山のことができる

最近の x64 プロセッサについている拡張命令として, POPCNT と PDEP と TZCNT というのがある

POPCNT x x (1 ワード) の中の 1 の数を数える

PDEP $x\ y$ x の下位 POPCNT y ビットを y の立っているビットのところに分配する

TZCNT x x の下位の連続する 0 の個数を数える

これを使えば表を使わない方針を高速化できる。

$$\text{rank}(x, i) = \text{POPCNT}(x \ll w - i)$$

$$\text{select}(x, i) = \text{TZCNT}(\text{PDEP}(1 \ll i, x))$$

CPU の拡張命令

最適化で使ってくれるときもあるが、明示的に使いたい場合は次の関数を使う

	POPCNT	PDEP	TZCNT
gcc	<code>__builtin_popcount</code>	<code>_pdep_u64</code>	<code>_tzcnt_u64</code>
clang	<code>_mm_popcnt_u64</code>	<code>_pdep_u64</code>	<code>_tzcnt_u64</code>
rustc	<code>_popcnt64</code>	<code>_pdep_u64</code>	<code>_tzcnt_u64</code>

ただし, gcc, clang はコンパイラオプションで `-msse4.2, -mbmi2` を必要とし, さらに `<x86intrin.h>` を include する必要がある。

また, rustc の関数群は全て `std::arch::x86_64` 内に定義されている。オプションに `-C target-feature=+bmi1,+bmi2,+sse4.2` を付ける必要がある。

selectについて

こちらには rank ほどのめばしい高速化，簡単化はない

先程述べた，葉で select をするときの高速化ぐらい

実装するのがとても大変なので，とりあえず rank を使って二分探索するようにして
よいです

少し賢くする方法として，密なブロックでのみ二分探索するようにすれば $O(\lg \lg n)$ 時間でできます。これでも実用上十分高速です。

まとめ

1. とりあえず access を作しましょう。
2. 次に rank を作しましょう。
3. select は rank による二分探索 $O(\lg n)$ を作る
 - (a) 余裕がある人は大ブロックに分割して Sparse なら全部持って Dense なら二分探索のやつ $O(\lg \lg n)$ を実装する
 - (b) めちゃくちゃ余裕がある人は定数時間 select を作しましょう。^{*1}

これ以降紹介するデータ構造は全て簡潔ビットベクトルを使うのでとりあえず動くものを作しましょう。

1 これは嘘かもしれません。普通に遅くて実用性がありません。

おまけ

良い感じっぽそうな実装として cs-poppy^{*2}というやつがありました。

かなり簡単な構造なので論文を見てもらえばすぐ分かると思いますが、基本的なアイデアは

- キャッシュミスを減らすのが本質
 - 紹介した rank の構造の大ブロックの各ワードの中に一緒に内部の小ブロックの情報を持つ bit 列を並べちゃう
 - キャッシュミスが一回分減る
 - 紹介した select(Clerk) はでかすぎるので、二分探索とかでごまかす
- なんか SIMD とか使ってもっと良い感じにしてるやつもありました。^{*3}

2 Zhou et al, Space-Efficient, High-Performance Rank and Select Structures on Uncompressed Bit Sequences, 2013

3 Kurpicz, Engineering Compact Data Structures for Rank and Select Queries on Bit Vectors