ハッシュテーブルの実装と計測

1　はじめに

プログラミング言語 C++ には、標準ライブラリに `std::unordered_map` としてハッシュテーブルが実装されている。しかし、`std::unordered_map` は要素へのポインタや参照を変化させてはならないという規格に基づいて実装されているため [1]、オーバーヘッドが生じ、実行時間が遅いという問題がある。本稿では、2 種類のハッシュテーブルを実装し、それらと `std::unordered_map` の性能を計測し比較する。

2　導入

2.1　計算量

計算量は、アルゴリズムの効率を評価するための尺度であり、**ランダウ記法**を用いて「入力サイズ n に対し、このアルゴリズムは時間計算量 $O(n)$ で動作する」のように記述される。

2.2　連想配列

プログラミングにおいて、複数の要素の列を扱うためには**配列**が用いられる。配列は非負整数を添字とするが、この添字として非負整数以外のデータ型も用いることのできるデータ構造を**連想配列**、あるいは**辞書**という。添字として用いるデータ型を `key`、要素とするデータ型を `value` と呼ぶ。

連想配列は、以下の 3 つの機能を実現する。

- `insert`
  指定した `key` に対し、`value` を指定した値で紐づける。
- `erase`
  指定した `key` と、それに紐づいた `value` の組を削除する。
- `find`
  指定した `key` に紐づいた `value` の値を得る。指定した `key` が含まれていない場合は、そのことを報告する。

配列に `key` と `value` の組をそのまま格納することで、最悪計算量 $\Theta(n)$ で 3 つの操作を実現することができる。しかし、n 個の要素を挿入するのに最悪 $\Theta(n^2)$ の計算量を要し、大きなサイズのデータを扱うためには現実的でない。そのため、扱うデータ型に応じて、値を高速に格納する手法が考えられてきた。

2.3　ハッシュテーブル

ハッシュテーブルは連想配列の一種である。`key` を**ハッシュ関数** h によってある区間 $[0, n)$ に属する整数に変換し、`value` を大きさ n の配列に格納することで機能を実現することができる。

しかし、用いるハッシュ関数の種類によっては、異なる 2 つの `key` が同じ整数に変換されることがある（これを**衝突**という）。衝突に対処する方法は様々であり、本稿では 2 つの手法を実装する。

3　ハッシュテーブルを実現する手法

3.1　連鎖法

連鎖法（Separate Chaining）[2] は、同じハッシュ値を持つ `key` と対応する `value` の組を全て同じリストに格納する。要素数が m のとき、リストの個数 n は $n \geq m$ を満たすようにすることで、リストの平均要素数が 1 以下となる。`insert`、`erase`、`find` の操作は、`key` x のハッシュ値 $h(x)$ に対応するリスト内を線形探索することで実現することができる。

`insert` 操作によって $n < m$ となってしまった場合、リストの個数を 2 倍に増やし、全ての要素のハッシュ値を計算し直し、もう一度リストに入れ直す。m 要素を `insert` するときに要する再計算の計算量の合計は $O(m)$ となる。

ハッシュ関数の性能が悪い場合、全ての要素が同じリストに格納されることになり、操作に $\Theta(m)$ の計算量を要する。高速に動作するためには、以下の条件を満たすハッシュ関数を用いるのが理想的である。

> x, y が 2 つの異なる `key` であるとき、$h(x) = h(y)$ となる確率は十分小さい定数 c を用いて $\frac{c}{n}$ と表される。

このハッシュ関数を用いたとき、任意の `key` x に対し、$h(x)$ に対応するリストの要素数の期待値は $\frac{cm}{n} \leq c$ で抑えられるため、c が定数であることから操作を $O(1)$ で実現することができる。

3.2　開番地法

開番地法（Open Addressing）[2] は、要素を直接長さ n の配列に格納する。新たな `key` x を `insert` する際に $h(x)$ に対応する場所にすでに値が格納されていた場合は、その近傍を探索し、空いている場所を探す。近傍を探索するにあたっては、$h(x)$ に対応する場所から 1 ずつ増やす方法をとり、これを**線形探索法（Linear Probing）**という。配列には、次の 3 つの種類の値を格納する。

- その位置に値が格納されている場合、`key` と `value` の組
- その位置に格納された値が削除された場合、そのことを表す特別な値 `del`
- その位置には何も格納されず、削除もされていない場合、そのことを表す特別な値 `null`

$h(x)$ からはじめ、`null` を見つけるまで探索したとき、途中で `key` が x であるような組が見つかった場合、ハッシュテーブルには `key` が x であるような値が存在する。そうでない、すなわち `null` が見つかった場合は、ハッシュテーブルには `key` が x であるような値は存在しない。このことを用いると、`insert`、`erase`、`find` の 3 つの操作を実現することができる。

配列の長さ n は、`null` でない値の個数を q、そのうち `del` でない値の個数を m として、$\frac{1}{8}n \leq m$ かつ $q \leq \frac{1}{2}n$ となるようにとる。条件を満たさなくなった場合は、$3m \leq 2^d$ を満たす最小の非負整数を d とおいたとき、$n = 2^d$ として要素を格納し直す。

開番地法には、衝突確率が低いハッシュ関数を用いたとしても、配列の連続した部分に要素が格納された場合 `null` にたどり着くまでの探索に長い時間がかかるという問題がある。開番地法では、以下の条件を満たすハッシュ関数を用いるのが理想的である。

> ハッシュ値は、$[0, n)$ に属する整数値を一様かつ独立にとる。

この強い仮定のもとで、3 つの操作を計算量 $O(1)$ で実現することができる。

3.3 ハッシュ関数の選択

ハッシュ関数は、衝突確率が十分低くなるように `key` を $[0, n)$ に属する整数に変換する必要がある。このような条件を満たす関数を簡単に定義できるようにするため、本稿では 2 つの簡略化を行う。まず、3.1 と 3.2 で紹介した手法では、$n = 2^d$ となる非負整数 d が存在すると仮定することができる。さらに、ハッシュ関数を次の 2 つのステップに分ける。

- $f(x)$
  `key` x を $[0, 2^{64})$ に属する整数 k に変換する。
- $g(k)$
  $[0, 2^{64})$ に属する整数 k を $[0, 2^d)$ に属する整数 l に変換する。

1 つ目のステップでは、衝突が起こってしまった場合、異なる `key` が同じものであるとみなされてしまい、ハッシュテーブルが正常に動作しない。これは、処理時間が遅くなる以上の問題であるため、衝突を防ぐことは重要な課題となる。2 つ目のステップは、衝突確率がハッシュテーブルの処理時間に影響する。用いる手法に適した関数を選択することが重要となる。

1 つ目のステップは、`key` とするデータ型によって使用する関数が大きく異なり、これはハッシュテーブルを使用するユーザに依存する。そのため本稿では、2 つ目のステップのみを実装する。$g(k)$ として使用するハッシュ関数は、以下のように定める。

- **Multiplicative Hashing**
  Multiplicative Hashing は、$[0, 2^{64})$ に属する奇数をランダムに選び p とおいたとき、$g(k) = \left\lfloor \dfrac{pk \bmod 2^{64}}{2^{64-d}} \right\rfloor$ とする方法である。異なる 2 つの整数 k_1, k_2 に対し $g(k_1) = g(k_2)$ となる確率は $\dfrac{2}{2^d}$ である [2]。
- **Fibonacci Hashing**
  Multiplicative Hashing において、ϕ を黄金比として $\dfrac{2^{64}}{\phi}$ に近い値を p として選ぶ ($p = 11400714819323198485$) と、得られる値が一様に分布するという性質がある [3]。これは Multiplicative Hashing の特別な場合であり、Fibonacci Hashing と呼ばれる。

これらの方法に共通する問題点は、入力の上位ビットの情報が失われやすいことである [3]。そこで、pk の代わりに k と $\left\lfloor \dfrac{k}{2^{64-d}} \right\rfloor$ の排他的論理和と p の積を採用することで、上位ビットの情報を反映することができる。本稿では、この工夫を施した Fibonacci Hashing を使用する。

4 設計と実装

3.3 で述べたように、データ型を整数に変換する処理はユーザに依存するため、`key` としては 64 ビット整数を使用する（ただし、`value` としては任意の型を使用することができる）。ハッシュテーブルのインターフェイスとして、`insert`、`erase`、`find` 関数を実装した。コードは GitHub に公開している。

開番地法 `https://github.com/KodamaD/Nada2021Report/blob/main/code/open_addressing.hpp`

連鎖法 `https://github.com/KodamaD/Nada2021Report/blob/main/code/separate_chaining.hpp`

開番地法の実装において、値を格納するために、必要最低限のメモリ量の 3 倍程度のメモリを確保している。`value` のデータ量が大きい場合これはオーバーヘッドとなるが、ユーザはこれに対処するために `value` のポインタを代わりに格納するという選択をすることができる。

5　計測と比較

`std::unordered_map`、連鎖法、開番地法のそれぞれについて、以下の 4 つの項目に関して実行時間を計測した。

- n 要素の `insert`
- n 要素の `erase`
- ハッシュテーブルに格納されている n 要素の `find`
- ハッシュテーブルに格納されていない n 要素の `find`

計測においては、5 回の実行における実行時間の平均をとった。$n = 10^5, 10^6, 10^7$ とした場合の実行時間は以下のようである（単位はミリ秒）。

表 1: $n = 10^5$

ハッシュテーブルの手法	`insert`	`erase`	存在する要素の `find`	存在しない要素の `find`
`std::unordered_map`	28.8	34.2	5.2	8.8
連鎖法	79.2	48.2	5.0	6.8
開番地法	15.4	8.6	4.0	4.2

表 2: $n = 10^6$

ハッシュテーブルの手法	`insert`	`erase`	存在する要素の `find`	存在しない要素の `find`
`std::unordered_map`	514.6	406.8	101.6	145.6
連鎖法	1023.4	531.0	100.2	116.2
開番地法	219.4	142.2	95.8	97.2

表 3: $n = 10^7$

ハッシュテーブルの手法	`insert`	`erase`	存在する要素の `find`	存在しない要素の `find`
`std::unordered_map`	6856.2	5263.4	1817.8	2391.0
連鎖法	27753.0	10898.0	1812.8	1802.8
開番地法	5155.4	2610.2	1502.4	1504.4

連鎖法は `insert`、`erase` 操作が非常に遅い。原因として、各種操作をポインタを介して行っていることが挙げられる。対照的に、開番地法はどの操作においても最も高速であり、`std::unordered_map` より高速なハッシュテーブルの実現に成功している。

6　まとめ

連鎖法、開番地法を用いたハッシュテーブルを実装し、実行時間の計測を行った。連鎖法は `find` 操作を除けば非常に低速であるが、開番地法はどの操作でも高速である。別のハッシュ関数を採用したり、メモリ操作の最適化を施したりすることで、さらに高速なハッシュテーブルが実現できる可能性があると感じた。Robin Hood Hashing や Swiss Tables といった、より高速な手法も提案されているため、今後はそれらについても学び、実装したい。

参考文献

[1] Richard Smith “Working Draft, Standard for Programming Language C++”
(http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/n4861.pdf)

[2] Pat Morin（著）、堀江慧・陣内佑・田中康隆（訳）『みんなのデータ構造』ラムダノート株式会社 2018 年

[3] Malte Skarupke “Fibonacci Hashing: The Optimization that the World Forgot (or: a Better Alternative to Integer Modulo)”
(https://probablydance.com/2018/06/16/fibonacci-hashing-the-optimization-that-the-world-forgot-or-a-better-alternative-to-integer-modulo/)