In-place Chained Hash Table

One of the fastest algorithm for successful search and unsuccessful search
using in-place doubly linked list.

@ADMIS_Walker

Compiled on 2019/10/22.

要旨

ハッシュ法は，入力されたキーと対応する値に定数時間でアクセスする探査アルゴリズムである[1)]．これは，コンピュータにおける配列アクセスが定数時間であることを利用している[2)]．もし，キーが連続した整数値であるなら，キーが配列インデックスと等しいときに，キーと対応する値を配列に格納しておく．すると，キーに紐付いた値は，配列インデックスにキーを代入することで取得できる[3)]．これを拡張して，ハッシュ法では，キーがどのような値でも，配列アクセスを利用した高速な応答性能を実現する．まず，キーをインデックスとして配列にアクセスするには，キーを配列サイズ未満の一意な正の整数に変換する必要がある．そこで，ハッシュ関数によりキーを正の整数に変換する．ハッシュ関数は，一定の法則で値をシャッフルする関数で，同じ入力には同じ出力を得る．出力のハッシュ値は，一見乱数のような値であり，剰余演算などにより，ハッシュ値を配列サイズ未満に丸める．以上により，任意のキーに対応する値を，ハッシュ関数の定める配列インデックスに登録しておくことで，定数時間によるアクセスができる．このとき，配列はキー (key) と値 (value) の対応を表す表 (table) であるため，ハッシュテーブルと表現する．また，key と value の対応を key-value ペアと表現する．ハッシュ法が定数時間による探査を実現する戦略は，以上である．

ところが，この手法をそのまま実装すると，配列サイズ未満に丸めたハッシュ値が，高い確率で衝突するため[4)]，実用に耐えない．1 つの配列要素に 2 つの値は格納できないため，衝突を解決するアルゴリズムが必要となる．本投稿で示すアルゴリズムも，数多ある衝突解決アルゴリズムの 1 つである．一般に，衝突を解決する処理はハッシュ法の効率を落とすため，衝突が発生しないよう，丸めたハッシュ値は配列全体に均一に分布することが理想的である．しかし現実には，丸めたハッシュ値が均一に分布することはなく，配列の利用率が高くなる程，簡単に衝突する．配列の利用率は load factor と呼ばれ[5)]，load factor が高い場合には，配列に格納した key-value ペアをより広い配列に移すことで衝突を解決する．この操作をリハッシュと呼ぶ[6)]．しかし，メモリ効率と実行効率を考えると，衝突の度にリハッシュする訳には行かない．そのため，load factor が低い場合には，リハッシュ以外の方法で衝突を解決する．

リハッシュ以外の衝突解決策は，主に 2 種類に大別される．1 つは chaining に代表される open hashing[7)] である．衝突が発生した際，新しく 1 要素分のメモリ領域を確保し，片方向リスト[8)] により現在の要素に追加する．この手法は，要素の削除と追加を繰り返す場合でも安定して動作する一方，要素をポインタにより接続するため，アドレスが不連続

---

1) 特定のキーと対応する値を返却することから，この機構を「ある単語」と「その説明」に見立てて「辞書」という名称で実装するプログラミング言語も多い．

2) 単純には，アドレス線に読み込み先のメモリ番地を指定すると，対応するメモリ番地からデータ線に値が出力されるように，コンピュータが構成されている．それぞれのメモリ番地はアドレス線が自身の番地を示していないか常に電圧を確認しており，電圧が自分の番地を示すと，スイッチがオンとなり，データ線に値を出力する．このため，定数時間で値が出力される．

3) 整数値が 0 始まりでない場合は，当然，配列インデックスからオフセットを差し引く．

4) 同じ配列要素に 2 つ以上の値が割り当てられることを，secondary clustering と呼ぶ．ハッシュ関数が低品質な場合は，secondary clustering が特に深刻となる．

5) 日本語では座席利用率と呼ぶ．

6) リハッシュする際は，リハッシュ時間を定数時間に収めるため，通常は倍サイズの配列に移す．ただし，ここでの定数時間とは，遷移先が倍サイズ以上の配列サイズであれば，繰り返しリハッシュした場合の計算量を 1 要素ごとに分割したコストが O(n) となり，要素数に比例した計算量と見なせることを意味する．

7) 別名：closed addressing.

8) 英名：singly linked list.

となり CPU キャッシュが効き難い．もう 1 つは linear probing や quadratic probing に代表される closed hashing[9] である．衝突が発生した際，それぞれの探査規則に従い，隣接する空き要素に値を格納する．Linear probing[10] は，隣接する $1, 2, 3, ...k$ 番目の要素を線形探査し，空き要素に値を格納する．配列が隙間なく埋まった場合は，境界が分からず探査時間は大きく増加する．これを primary clustering と呼ぶ．Quadratic probing は，隣接する $1^2, 2^2, 3^2, ...k^2$ 番目の要素を探査する．飛び値を取るため，primary clustering し難い．いずれの probing も，空の要素が探査終了条件の 1 つ[11] であるため，要素を削除すると「要素が削除された」のか「要素が存在しない」のか判断できない．このため，要素削除時は，削除フラグを付与する．これらの probing 手法は，要素の削除と追加を繰り返す場合において，性能低下とそれに伴うリハッシュを必要とする一方，隣接する配列や，飛び値の場合にも比較的近いアドレスに要素を格納するため，CPU キャッシュが効き易い利点がある．

Closed hashing には，これにまでに挿入したキーのハッシュ先とは衝突していないにも関わらず，primary clustering により，ハッシュ先が別の隣接要素の退避先として使用されていることがある．この場合，従来手法では，単に衝突として扱い probing により空き領域へ挿入する．これは挿入時間を低下させる反面，探査には追加の時間を必要とする．

本投稿では，ハッシュ先自体に衝突のない場合は，挿入済みの要素を入れ替え，ハッシュ先が一意に定まるキーを優先的に挿入するアルゴリズムを提案する．これには，挿入済みの要素が，何処に挿入されるべきであったかを把握する必要がある．このため，ハッシュテーブルには双方向リスト[12] 構造を用いる．通常のリスト構造は，ポインタを用いるためアドレスが不連続となり CPU キャッシュが効き難くなる．そのため，双方向リスト構造は，要素の接続をポインタではなく uint8 または uint16 を用いた相対位置による in-place 実装により実現した．

Key と value に uint64 を用いた結果として，uint8 を双方向リスト構造に用いた場合は，テーブルサイズ 10^2〜10^7 において，従来手法よりも高速な successful search または unsuccessful search を実現した．uint16 を双方向リスト構造に用いた場合は，closed hashing として従来手法より 25 %[13] 程度高いメモリ効率を実現した．また，サイズが 10^7〜$10^{8.3}$ の巨大なテーブルにおいて，従来手法より高速な unsuccessful search を実現した．

---

[9] 別名：open addressing.
[10] 和名：線形走査法
[11] 最悪計算量を保証するため，探査回数を制限することもある．
[12] 英名：doubly linked list.
[13] 従来の 75 % のメモリ使用率.

目次

第 1 章　序論 1
1.1　先行研究 1
1.2　目的 3

第 2 章　アルゴリズム 4
2.1　挿入 5
2.2　探査 7
2.3　削除 7
2.4　配列の末尾処理 7
2.5　ハッシュ値の計算 9

第 3 章　実装 15
3.1　ベンチマーク用コード 15

第 4 章　ベンチマーク 18
4.1　環境 18
4.2　各ハッシュテーブルの概要と測定条件 18
4.3　結果 19

第 5 章　考察 27

第 6 章　結論 31

付録 33
A　CPU が各演算に消費する clock 数 33

第 1 章

序論

ハッシュテーブルの挙動を理解する上で，数理モデルの示す理論的な計算量は大きな助けとなる．Fig. 1.1 に Knuth [1998] よる比較を示す．L は linear probing, U は quadratic probing [1)]，S は separate chaining である．Fig. 1.1 が示すように，load factor が高くなるにつれて，平均探査回数は上昇する傾向が見られる．この特性は，linear probing では特に顕著であるが，quadratic probing では幾らか改善されている．しかし，いずれのアルゴリズムも unsuccessful search では $\alpha = 0.65$ 前後，successful search でも $\alpha = 0.85$ 前後において，急激に平均探査回数が増加している[2)]．一方で，chaining を利用する C, S, SO については，性能悪化は限られることがわかる．このとき，C は coalesced chaining[3)]，S は separate chaining，SO は separate chaining with ordered lists である．

1.1 先行研究

現実的なハッシュテーブルを検討するには，単にアルゴリズムのみならず，対応する実装との比較が望ましい．ここでは，C++ 言語におけるハッシュテーブルを実装を示す．

std::unorderd_map

Chaining 系の実装の 1 つ．STL に収録された C++ 標準のハッシュテーブルである．アルゴリズムの選択からわかる通り，安定性とメモリ効率を重視している．

google::dense_hash_map

Quadratic probing の実装の 1 つ．google-sparsehash@googlegroups.com [2005] に収録されている．キーの 1 つを空符号として登録する必要があり，キーとして使用できなくなる．要素の削除が必要な場合は，削除符号も登録する必要があり，同様にキーとして使用できなくなる．この実装は，メモリ使用量の削減と，それに伴うキャッシュ効率の向上，また，要素探査時のコードの単純化と分岐予測精度の向上が期待される．Load factor は，探査速度向上のため 50 % に制限されている．

---

1) $M \to \infty$ において，quadratic probing と double hashing は uniform hashing と等価.

2) このため，性能が悪化する手前でリハッシュし，性能悪化を防ぐことが一般的である．なお，リハッシュする load factor は，実装依存である．

3) Closed hashing の一種で片方向リストにより，次に辿る要素を示す.

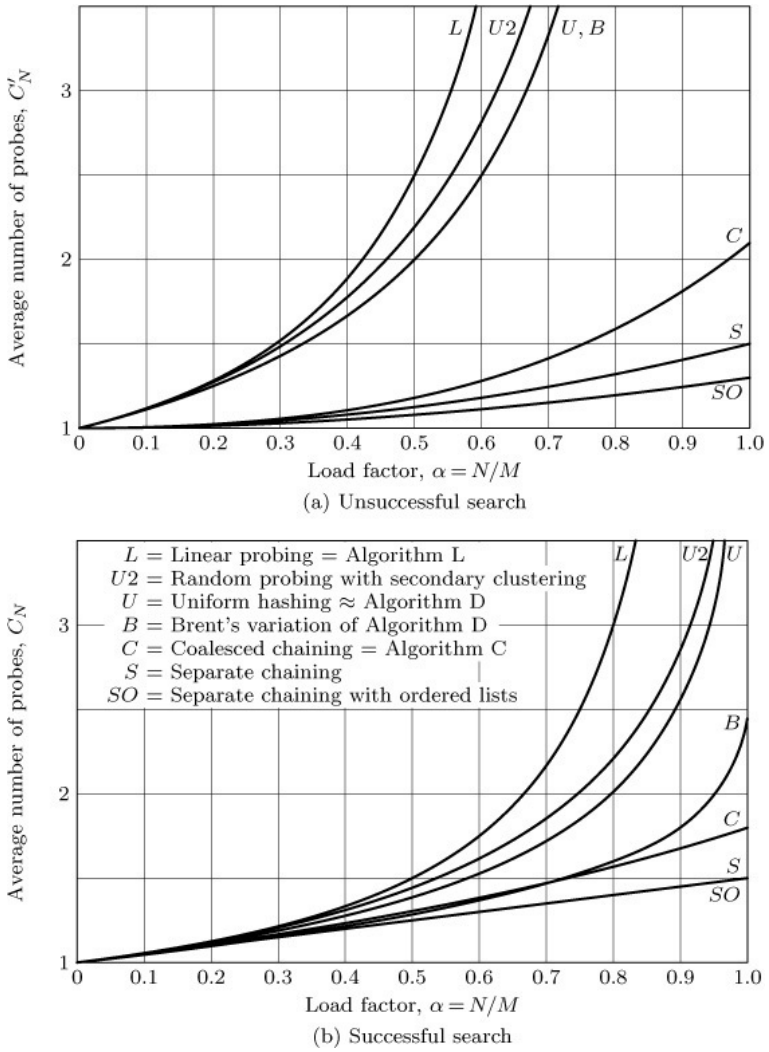

Figure 1.1. Comparison of collision resolution methods: limiting values of the average number of probes as $M \to \infty$ [Knuth 1998]. N is a number of elements on the table. M is the table size.

いくつかのハッシュテーブルでは，キーのハッシュ値をテーブルサイズに丸めるために剰余演算を用いている．一般的に，整数除算に必要な CPU cycle は 14~46 clocks 程度[4] である．このとき dense_hash_map の性能は，ハッシュテーブルが L2 キャッシュに収まるとき，探査速度は 250 query/μs 程度[5] であり，探査 1 回の実行時間は 4 ns [6] となる．このため，探査 1 回あたりの CPU cycle は 15 clock 程度[7] とわかる．探査には整数除算以外にもハッシュ値の計算や配列の走査が必要であることを考慮すると，整数除算に必要な CPU cycle は dense_hash_map 探査時間に対して CPU cycle が大きい．このため実装を確認すると，dense_hash_map では整数除算の代わりとして，テーブルサイズが $2^k - 1$ $(k = 1, 2, ...)$ となるよう制御し，ビットマスクによりハッシュ値の最下位 k ビットだけを取り出して配列インデックスとしていることがわかる．

---

[4] Fog [2018] より AMD Ryzen7 1700 の場合．

[5] AMD Ryzen7 1700 (8C/16T) 3.7 GHz の場合．詳細は，第 4 章を参照．

[6] $250[\text{query}/\mu\text{s}] = \frac{1}{250}[\mu\text{s}/\text{query}] = \frac{10^3}{250}[\text{ns}/\text{query}] = 4[\text{ns}/\text{query}]$

[7] 3.7 GHz の CPU では単位クロックあたりの実行時間が $\frac{1}{3.7[\text{GHz}]}[\text{sec}/\text{clock}] = \frac{1}{3.7\times10^9}[\text{sec}/\text{clock}] = 2.7 \times 10^{-10}[\text{sec}/\text{clock}] = 2.7 \times 10^{-1}[\text{ns}/\text{clock}]$ であるから，探査 1 回あたりの CPU cycle は $\frac{4[\text{ns}/\text{query}]}{2.7\times10^{-1}[\text{ns}/\text{clock}]} = \frac{4}{2.7\times10^{-1}}[\text{clock}/\text{query}] \simeq 15[\text{clock}/\text{query}]$ 程度である．

ska::flat_hash_map

Robin Hood hashing の実装の 1 つ．Robin Hood hashing は衝突解決法の 1 つで，片方向リストによりハッシュ先のテーブルアドレスを示しており，次の要素位置を示す coalesced chaining とは片方向リストの使い方が逆である．本来の挿入位置がわかるため，要素挿入時に，より近い位置へ要素が移動するよう調整できる．探査時には，ハッシュ先になるべく近い位置へ移動させられた要素を線形探査する．flat_hash_map では線形探査のコストに配慮し，サイズ n のテーブルに対して，探査を隣接する $log_2(n)$ 個の要素に制限している [Skarupke 2017]．Load factor は，探査速度向上のため 50 % に制限されている．

1.2　目的

理想的なハッシュテーブルは，衝突がなく，ハッシュ計算の必要もない，単なる配列である．現実のハッシュテーブルは，ハッシュを計算し，衝突を解決する必要がある．単に高い探査性能を求めるのであれば，Fig. 1.1 が示すように load factor を下げてしまえば，衝突解決の頻度が下がり，平均探査回数は，どの手法でも同じような回数に落ち着く．しかし，メモリ資源は有限であり，高いメモリ効率を達成すれば，それだけ CPU キャッシュにも乗り易くなる．加えて，高い load factor まで稼働するほどリハッシュしづらく，実利用時の安全マージンも広く取れる．

Open hashing 系のアルゴリズムは chain 構造を持っており，primary clustering の影響を一切受けない[8)]．このため，Fig. 1.1 に示すように，closed hashing 系のアルゴリズムよりも平均探査回数が少ない．一方で，chain 構造はポインタにより実装されるため，メモリアドレスが不連続となり，キャッシュ効率が悪く，理論上の性能を発揮しない．

Closed hashing 系のアルゴリズムは，primary clustering により平均探査回数の悪化が発生し易い．Primary clustering は特に linear probing で顕著であり，このため，同様に線形探査を行う Robin Hood hashing の flat_hash_map による実装では探査回数を制限している．Quadratic probing では primary clustering の影響は小さいものの，要素を隣接する $1^2, 2^2, 3^2, ...k^2$ 番目の配列に格納するため，距離が遠い程キャッシュミスし易い．

本投稿では，primary clustering の影響を完全に避けるために chain 構造を採用し，高いキャッシュ効率を得るために closed hashing を用いる．このとき，closed hashing として chain 構造を実現するために，双方向リストを in-place 実装する．これにより，従来より高い探査性能を持つハッシュテーブルが実現できることを，サンプル実装とベンチマークにより実証することが目的である．

[8)] Load factor 増加による性能悪化は secondary clustering によるもの．

第 2 章

アルゴリズム

Linear probing や quadratic probing など従来の closed hashing では，ハッシュ先が重複していない場合でも，別のキーの退避先として配列要素が使用中の場合，本来 1 回目の探査でアクセスできるキーであっても，probing により衝突を解決しなければならない．Robin Hood hashing は，1 回でアクセスできる位置に要素を移動できるものの，ハッシュ先の重複[1] に対しては線形探査が必要となる．

本投稿では，ハッシュ先が別のキーの退避先として利用されてしまう primary clustering に付随した問題に対して，逆方向にリストを辿ることで，要素が使用済みの場合でも適切に移動し，ハッシュ先が必ず 1 つ目の要素となるように調整する．また，要素探査時は，順方向にリストを辿ることで平均探査回数を削減する．メモリ使用量を削減するため，リストは int 型により相対位置を記録し，これら順方向と逆方向を合わせて双方向リスト構造とする．以上のアルゴリズムを実現するハッシュテーブルを in-place chained hash table と呼ぶこととし，以降，IpCHashT と略記する．

IpCHashT のデータ構造における提案は，本質的に ADMIS [2017] の提案と同様である．以下，Fig. 2.1，Fig. 2.7〜2.17，Fig. 2.18〜2.23 は，ADMIS [2017] に由来する．ただし，本投稿では，挿入アルゴリズムに新しく soft insertion を採用している．探査においても，successful search を優先するオプションと，unsuccessful search を優先するオプションについて新たに実装と評価を行う．また，パディングサイズは従来固定であったが，テーブルサイズによってパディングサイズを動的に変更することで，テーブルサイズが小さい場合に最大 load factor が低下する問題に対処する．要素の挿入においては，操作手順を再検討し，より簡潔な実装とする．

IpCHashT では，Fig. 2.1 に示すデータ構造を備える．各要素には key-value ペアの他に，双方向リストのための prev 要素と next 要素を持つ．T_shift には uint8 または uint16 型を用い，相対位置による双方向リストを構成する．これは，ポインタ接続におけるメモリ消費量が無視できないためである[2]．

本章で示す Fig. 2.2〜2.23 は，青色を初期状態とし，赤色が挿入時の変更を，緑色は要素の移動に伴う変更を，それぞれ表す．Fig. 2.2 では，Fig. 2.3〜2.23 に用いる記号を説明している．Fig. 2.3〜2.23 では，丸印ひとつ 1 つが配列要素を，双方向に張られた 2 つの矢印が双方向リストを表す．Open hashing を用いたハッシュテーブルの説明では，連続したアドレス空間上の配列要素が，２辺を共有した四角形の連なりにより表現されることが多い．同様に，本投稿では IpCHashT の配列要素を抽象化して表すが，一見すると Fig. 2.3 のように隣接した連続構造であっても，実際には，Fig. 2.4 のように各要素が断続的なアドレス空間上に存在することがある．

---

[1] secondary clustering.

[2] 例えば 64 bits CPU の場合，ポインタサイズは 64 bits であるから，T_key と T_val が uint64 型の場合，テープルサイズの 50% が双方向リストに由来する．

```
template <class T_key, class T_val, typename T_shift>
struct element{
    element(){
        T_shift maxShift = (T_shift) 0; maxShift =~ maxShift;
        prev = maxShift;
        next = (T_shift) 0;
    }
    ~element(){}

    T_key key;
    T_val val;
    T_shift prev;
    T_shift next;
};
```

Figure 2.1. Pseudo C++ data structure for IpCHashT element. “T_key” is a key type and “T_val” is val a value type. “T_shift” is used for doubly linked lists and specifies uint8 or uint16. Specifying a type larger than uint16 has little merit. “prev” indicates the distance from the previous element and “next” indicates the distance from next element. Unlike the address in the pointer, the link list is expressed in the interval $[0, \max(\mathrm{T\_shift}) - 1]$, then max(T_shift) is the maximum possible value of T_shift type. “0” indicates itself and “$\max(\mathrm{T\_shift}) - 1$” is the maximum link distance. There is no method to link outside of a section. “max(T_shift)” is reserved for prev, and “prev==max(T_shift)” indicates that the element is empty. “prev==0” indicates that the element is the head of the linked list, and “next==0” indicates that the element is the tail of the linked list.
図 2.1 IpCHashT 要素の C++ 擬似構造体．T_key はキーの型，T_val は val の型である．T_shift は双方向リストに用いる型で，uint8 または uint16 を指定する．uint16 より大きな型を指定するメリットは殆どない．prev は 前の要素までの相対距離を，next は次の要素までの相対距離を表す．ポインタにおけるアドレスとは異なり，区間 [0, max(T_shift) - 1] の範囲でリンクを表現する．ただし，max(T_shift) は T_shift 型の取り得る最大値である．0 のときに自分自身を示し，max(T_shift) - 1 がリンクできる最大距離である．区間外へのリンクはできない．max(T_shift) は，予約されており，’prev==max(T_shift)’ のとき，要素が空であることを示す．また，’prev==0’ であればリストの先頭であること，’next==0’ であればリストの末尾であることがわかる．

2.1　挿入

高い探査性能を達成するためには，命令数を削減するだけでなく，キャッシュミスを抑える必要がある．一般に，CPU は配列アクセスに対して，参照の局所性を利用してキャッシングする．そのため，必要な要素を配列に隙間なく詰め込むことで空間的局所性を，可能な限り連続した位置に配置することで逐次的局所性を，それぞれ高め，キャッシュミスを削減する．

key-value ペアを 1 つ挿入するには，次の 1) 〜 4) の場合を考える．1) ハッシュ先の配列が空の場合は，Fig. 2.7 のように単に要素を詰める．2) 既に要素が挿入されており chain の間に空きがある場合は，Fig. 2.8 のように間に挿入する．3) chain の間に空きがない場合は，Fig. 2.9 のように末尾に空きを探し挿入する．4) 挿入先の要素が異なるハッシュ値を持つ要素の退避先に使用されている場合は，Fig. 2.10〜2.17 のように挿入する．要素の退避先と locator の再接続先との兼ね合いのため，4) には多くの場合分けが必要となる．

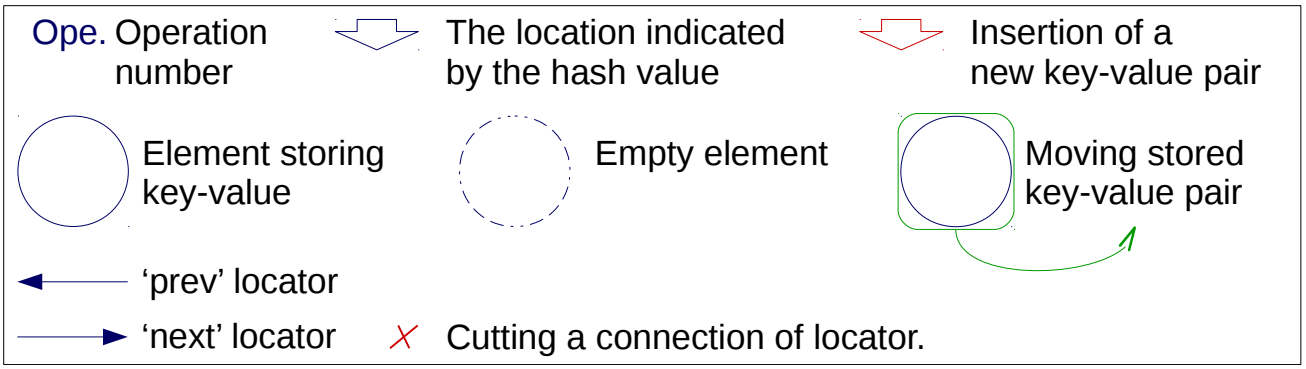

Figure 2.2. Symbols used in Fig. 2.3∼2.23. “ Ope. ” indecates the execution order, and “ Ope. 0 ” means the initial state. The down arrow is placed on the array element indicated by the hash destination. Each element is composed by a circle, a prev locator, and a next locator. However, locators are omitted when there are no connection. Circles written with dotted lines represent empty elements. Green boxes and arrows represent movement of elements. Crosses indicate that the linke is deleted. The color scheme is blue for the initial state, red for element insertion and deletion, and green for element movement.

図 2.2 Fig. 2.3∼2.23 に用いる記号．“Ope.” は，実行順序を表し，“Ope. 0” の場合は初期状態を意味する．下向き矢印は，ハッシュ先が示す配列要素の上に置かれる．各要素は，丸 1 つと prev locator 1 つ，next locator 1 つで構成される．ただし，接続が無い場合 locator は省略される．点線で書かれた丸は空の要素を表す．また，緑色の枠線と矢印は要素の移動を表す．バツ印はリストの削除表す．配色は，青色を初期状態，赤色を要素の挿入と削除，緑色を要素の移動，としている．

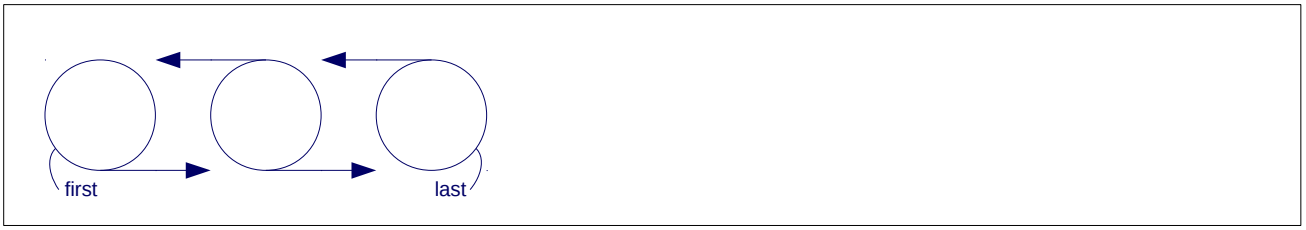

Figure 2.3. An abstract representation of the element chain inserted into IpCHashT. In this case, the hash destinations of the three elements indicate the address of the first element, and the conflict is resolved by doubly linked list. Each element is connected by prev and next locators that indicate relative position, and there may be another element between each element.

図 2.3 IpCHashT に挿入された要素の抽象表現．この場合，3 つの要素のハッシュ先は，いずれも first 要素のアドレスを示すため，双方向リストにより衝突を解決している．各要素は prev locator と next locator の示す相対位置により接続されており，各要素間に別の要素がある可能性がある．

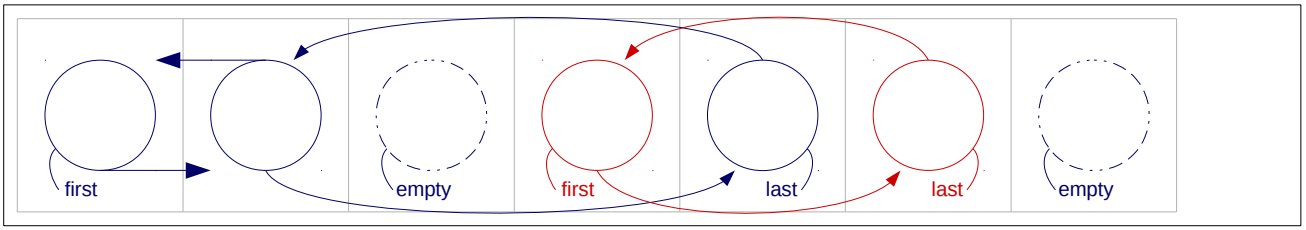

Figure 2.4. An example of mapping the abstract representation shown in Fig. 2.3 to a continuous address space. A gray frame represents the array. Even with the same abstract representation, the state of the continuous address space depends on the array state at the time of insertion. Also, deleting elements can cause memory fragmentation. The blue list is a map of the abstract representation shown in Fig. 2.3, and the red list is another chain inserted between them. In this example, the blue list is fragmented and the tail element is inserted in a distant place even though there is an empty element in the foreground. This fragmentation occurs when three or more elements are inserted into the blue list and then the value stored in the left empty element is deleted.

図 2.3 Fig. 2.3 に示す抽象表現を連続アドレス空間上に写像した一例．灰色の枠線は配列を表す．同じ抽象表現でも，連続アドレス空間の状態は，要素挿入時の配列状態により異なる．また，要素の削除はメモリを断片化させることがある．青色のリストは Fig. 2.3 に示す抽象表現の写像であり，赤色のリストは間に挿入された別の chain である．この例では，青色のリストが断片化されており，手前に空の要素があるにも関わらず，末尾要素が遠い場所に格納されている．この断片化は，3 つ以上の要素が青色のリストに挿入された後に，左側の空要素に格納されていた値が削除されたことにより発生する．

chain 間に空きは，線形探査により調べる．これには，対象の要素を全て調べ上げる必要があり，処理に時間が掛かる．また，11 通りの場合分けを全て実装するには非常に手間も掛かる．Fig. 2.7〜2.17 に示す場合分けは，任意の箇所に空きがあることを考慮している．しかし，要素削除なく挿入する場合には，メモリの断片化は発生しないため，必要な場合分けは Fig. 2.7，2.9，2.12，2.17 の 5 通りである．また，要素削除を伴う場合においても，挿入コストが高いため推奨しない．これは，要素の削除と再挿入によるメモリ断片化への耐性を捨て，実装コストの低減と，挿入の高速化を優先している．

2.2　探査

挿入済みの要素を探査するには，Fig. 2.5，2.6 に示すアルゴリズムが考えられる．Fig. 2.5 は successful search を優先した設定であり，以降，successful search major option と呼ぶ．Fig. 2.6 は unsuccessful search を優先した設定であり，以降，unsuccessful search major option と呼ぶ．

Fig. 2.6 の unsuccessful search major option は，キーの比較コストによって successful search major option の結果より悪化することが十分に考えられるが，比較コストの低いキーの場合には，unsuccessful search 時の分岐が少なく，高い性能が期待される．実際に，分岐予測の失敗におけるペナルティは，10~20 clock [3] 程度であり，無視できない．

2.3　削除

通常，リストの要素を削除するには，要素を削除した上で，ポインタを繋ぎ変えればよい．しかし，IpCHashT では，幾つかの場合を考慮する必要がある．

key-value ペアを 1 つ削除するには，次の 1) 〜 4) と，メモリの断片化を防ぐ処理として 5) を考える．1) 単一要素の場合は，Fig. 2.18 のように単に削除する．2) 末尾要素を削除する場合は，Fig. 2.19 のように要素と locator を削除する．3) 先頭の要素が削除された場合，探査不能としないため，先頭に別の要素をつなぎ替える必要がある．Fig. 2.20 では，断片化を防ぐために，末尾のデータを先頭へ移動させている．4) chain の中間要素を削除する場合は，Fig. 2.21 のように末尾要素を移動させる．5) 別の削除処理によって，末尾の要素が移動すると，Fig. 2.22，2.23 のように，chain の要素間に空きができ，断片化する場合がある．これを Fig. 2.22，2.23 に示す操作を繰り返すことにより修正する．ただし，空き要素の探査は線形探査する必要がある．また，要素を移動させる度に別の chain に空きができる可能性があるため，再帰的に行う必要がある．この処理は，実行コストが高いため推奨しない．

以上を勘案して，1) 〜 4) の操作を実装する．5) は実装しない．

2.4　配列の末尾処理

要素の衝突が発生した際，closed hashing では，現在より後のアドレスに key-value ペアを格納することで衝突を解決する．しかし，ハッシュ値が配列の末尾を示した場合は，退避先の配列要素が存在しない．この場合，配列の末尾に達した場合は，1) 引き続いて先頭から辿るように処理する，2) 予め末尾に余分な配列を確保する，の二択である．まず，1) は，配列の読み込みが不連続となりキャッシュミスを誘発する．このため採用できない．次に，2) は，キャッシュミスを誘発する危険はないものの，余分な配列をどの程度確保するか問題となる．

2) で最も簡単な実装は，パディングを固定長とすることである．ただし，これには欠点があり，テーブルサイズが小さいとき，パディングが不足すると衝突を解決できずにリハッシュが発生し，load factor の上限は上がらない，逆に，パディング過剰の場合，不用意にリハッシュが抑制され，探査効率が落ちる．

[3] Fig. A.1 の “Wrong” branch of “if” (branch misprediction) を参照.

```
template <class T_key, class T_val>
(bool, T_val) find(T_key key_in){
    uint64 hashVal = hashFunc( key_in );
    uint64 idx = hashVal & tableSize_minus1;

    if(! isHead( table[ idx ] ) ){ return ( false, none ); }
    for(;;){
        if( table[ idx ].key == key_in ){ return ( true, table[ idx ].val ); }
        if( table[ idx ].next == 0 ){ return ( false, none ); }
        idx += table[ idx ].next;
    }
}
```

Figure 2.5. Pseudo C++ code for the “find()” function executed while **successful search major option** is specified. This function is tuned to speed up “successful searches” more than “unsuccessful seatches”. The “isHead()” function checks the hash destination is a head element or not, and if it is the head, it follows the list. At this time, the “isHead()” function returns true when the “prev” element which is a member element of the structure shown in Fig. 2.1 is 0. Then the “find()” function returns true and a corresponding value to the key when a “key_in” is equal to the key on the array. In all other cases, the “find()” function returns false. Compared with the existing closed hashing method, the number of key comparisons is limited to the number of elements on the doubly linked list, therefore high-speed successful search is possible. As with the dense_hash_map, the check with the “isHead()” function becomes unnecessary when one key is registered as an empty mark and is guaranteed not to be used. This is because the initial value of an element outside the list is guaranteed not to match any key. For this reason, removing the “isHead()” function will speed up unsuccessful search too while the key comparison cost is small. However, this post does not deal with such special conditions.

図 2.5 Successful search major option を指定した場合に実行される find 関数の擬似 C++ コード．Successful search が高速に実行されるようにチューニングされている．isHead 関数によりハッシュ先の要素が双方向リストの先頭か否かを確認し，先頭の場合に双方向リストを辿る．このとき，isHead 関数は，Fig. 2.1 に示す element 構造体の prev 要素が 0 のときに true を返す．リストの先頭の場合，key_in が配列上のキーと一致すれば，対応する値と true を返す．それ以外の場合は全て false を返す．既存の closed hashing 法と比較して，キーの比較回数が双方向リスト上の要素数に制限されるため，高速な successful search が可能となる．なお，dense_hash_map と同様に，あるキーを空符号として登録し，使用されないことが保証される場合は，isHead 関数による検査が不要となる．これは，リスト外の要素の初期値が，いずれのキーとも一致しないことが保証されるためである．このため，isHead 関数を削除すると，キーの比較コストが小さい場合に，unsuccessful search も高速になる．ただし，条件が特殊なため，この投稿では扱わない．

テーブルサイズが小さい場合，パディングサイズが大きいいと，テーブルサイズよりもパディングサイズが支配的となる．逆に，テーブルサイズが大きい場合，パディングサイズが小さいと，退避先の配列が不足し，最大 load factor が悪化する．したがって，パディングサイズは，テーブルサイズに応じて調整が必要となる．

必要なパディングサイズは，線形に増加すると推測されるため，

$$\mathrm{pSize} = \begin{cases} (1/a)\cdot \mathrm{tSize} & (\mathrm{tSize} < \mathrm{limit}) \\ \mathrm{limit} & (\mathrm{tSize} \geq \mathrm{limit}) \end{cases}$$

a : threshold,　pSize : padding size,　tSize : table size,　limit : limit of T_shift size

とすればよい．しかし，テーブルサイズが小さい場合ハッシュ先は十分に分散しないため，単純に原点を通した調整では load factor の上限値が安定しない．したがって，先ほどの数式に，バイアス項を付与した

$$\mathrm{pSize} = \begin{cases} (1/a)\cdot \mathrm{tSize} + b & (\mathrm{tSize} < \mathrm{limit}) \\ \mathrm{limit} & (\mathrm{tSize} \geq \mathrm{limit}) \end{cases}$$

a : threshold,　b : bias,　pSize : padding size,　tSize : table sizelimit : limit of T_shift size

をパディングの調整に用いる．

```
template <class T_key, class T_val>
(bool, T_val) find(T_key key_in){
    uint64 hashVal = hashFunc( key_in );
    uint64 idx = hashVal & tableSize_minus1;

    for(;;){
        if( table[ idx ].key == key_in ){
            if( isEmpty( table[ idx ] ) ){ return ( false, none ); }
            return ( true, table[ idx ].val );
        }
        if( table[ idx ].next == 0 ){ return ( false, none ); }
        idx += table[ idx ].next;
    }
}
```

Figure 2.6. Pseudo C++ code for the "find()" function executed while **unsuccessful search major option** is specified. This function is tuned to speed up "unsuccessful searches" more than "successful seatches". However, since the number of key comparisons is greater than that of successful search major options, data structures with high key comparison costs are considered to perform worse than successful search major options. In a case of unsuccessful search, it is necessary to find out all the keys associated with hash destination do not match "key_in". Prioritizing key comparison makes unsuccessful search faster than ensuring the element is head. However, comparing the keys even if the hash destination is not a head of the list, comparing keys will cause a malfunction when a initial value of key element is equal to key_in. For this reason, "isEmpty()" needs to check that the element is not empty.
図 2.6 Unsuccessful search major option を指定した場合に実行される find 関数の擬似 C++ コード．Unsuccessful search が高速に実行されるようにチューニングされている．ただし，キーの比較回数は successful search major option よりも多くなるため，キーの比較コストが高いデータ構造では，successful search major option より性能が悪化すると考えられる．Unsuccessful search では，キーの一致の有無に関わらず，ハッシュ先に関係するキーを全て探査し，key_in と一致しないことを確認する必要がある．そのため，isHead より先頭要素であるかを確かめるよりも，キーの比較を優先する方が，高速に動作する．ところが，リストの先頭でない場合もキーを比較するため，キーが要素の初期値と等しい場合に誤動作する．このため，isEmpty により要素が空でないことを確認する必要がある．

このとき，定数 a, b は，load factor を最大化する，最小のパディングサイズとなる値が望ましい．この定数が不用意に大きいと，IpCHashT は T_shift の限界まで要素を挿入しようとする．これは，特に，テーブルサイズが小さい場合に顕著となる．また，uint16 を用いる場合は，その最大 T_shift サイズ - 1 の 65534 要素先まで接続できてしまうため，影響が広い区間に渡って続くことになる．なお，この場合 65535 は空フラグとして予約されている．

定数 a, b の最適値はともかくとして，実用に耐えうる定数を探すだけであれば，実験的に求められる．これには，Fig. 4.1 に示す IpCHashT<uint64,uint64> (uint8, maxLF100) の load factor が最大となる，最小のパディングサイズを探せばよい．例えば，$a = 18$, $b = 35$, $\mathrm{limit} = 254$ である．なお，254 以上のパディングは，少なくともテーブルサイズ 10^0〜$10^{8.3}$ において，大きな違いがないため，uint16 の場合においても $\mathrm{limit} = 254$ としており，a, b にも同じ定数を用いる．

2.5　ハッシュ値の計算

ハッシュテーブルでは，キーからハッシュ値を生成し，テーブルサイズに収まるように丸める．このとき，剰余演算を用いて丸めることが多い．また，剰余演算を用いる場合には，ハッシュ値とテーブルサイズが互いに素となるよう素数にするのが望ましい [石畑 1989]．しかしながら，第 1 章で示したように，整数除算は探査速度が至上命題となる場合には，あまりにも遅い．したがって，dense_hash_map と同様に，テーブルサイズを $2^k - 1$ $(k = 1, 2, ...)$ とし，ハッシュ値の最下位 k ビットだけをビットマスクにより取り出して，配列インデックスとする．

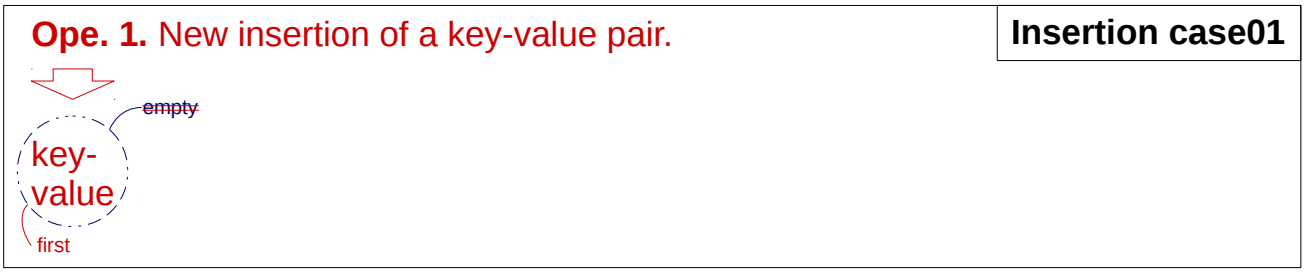

Figure 2.7. Insertion case01.

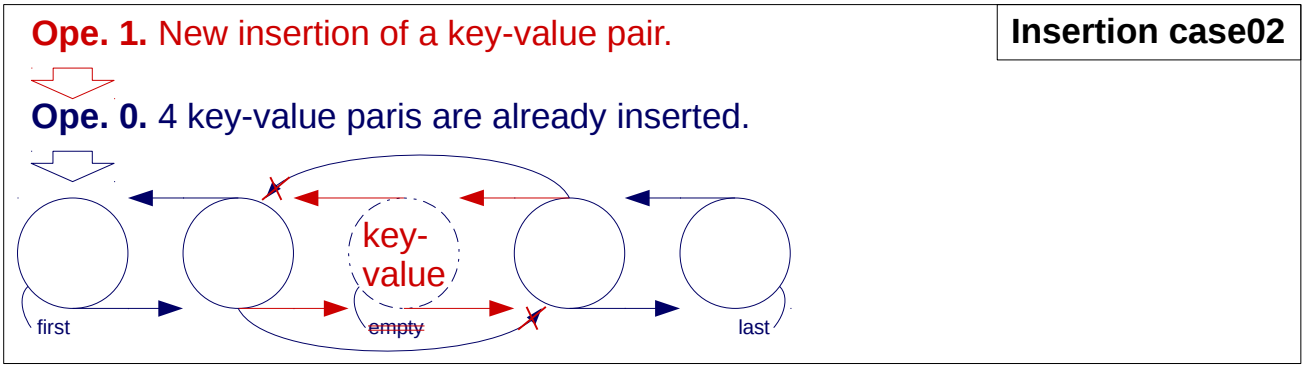

Figure 2.8. Insertion case02.

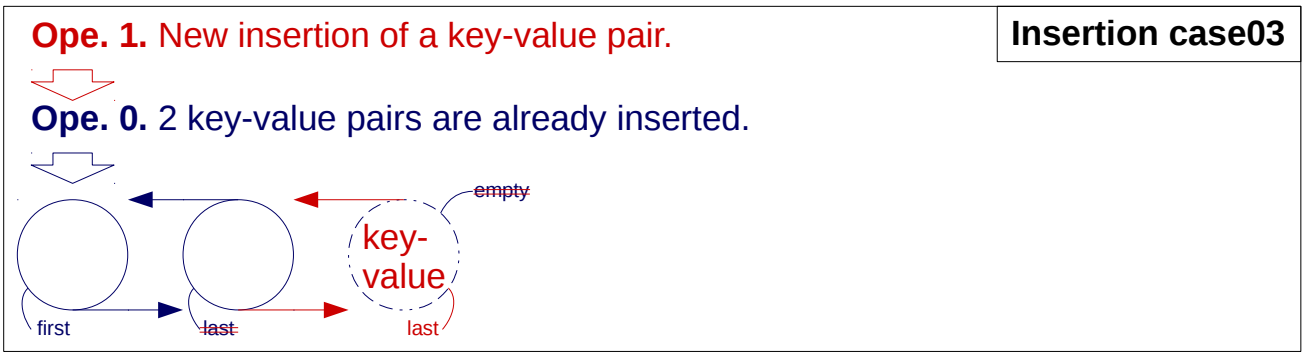

Figure 2.9. Insertion case03.

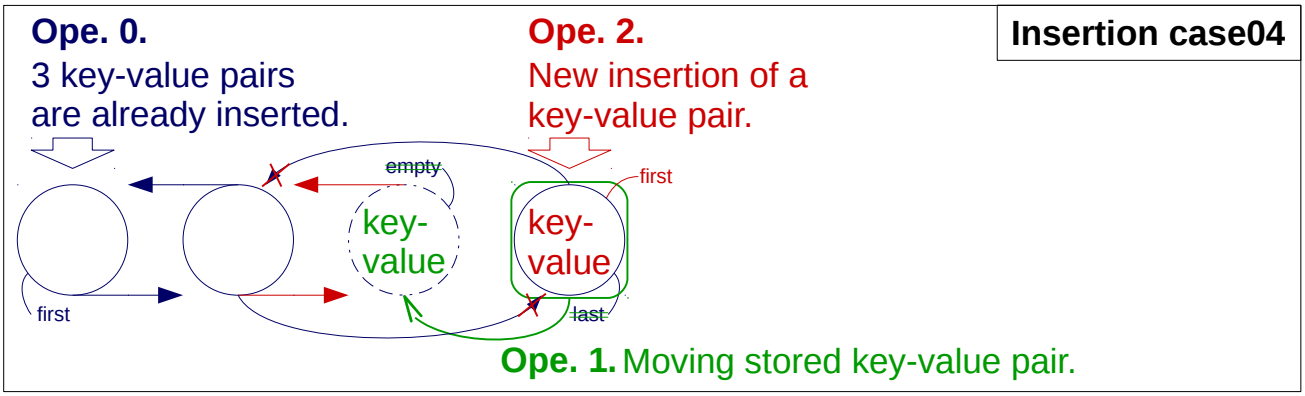

Figure 2.10. Insertion case04.

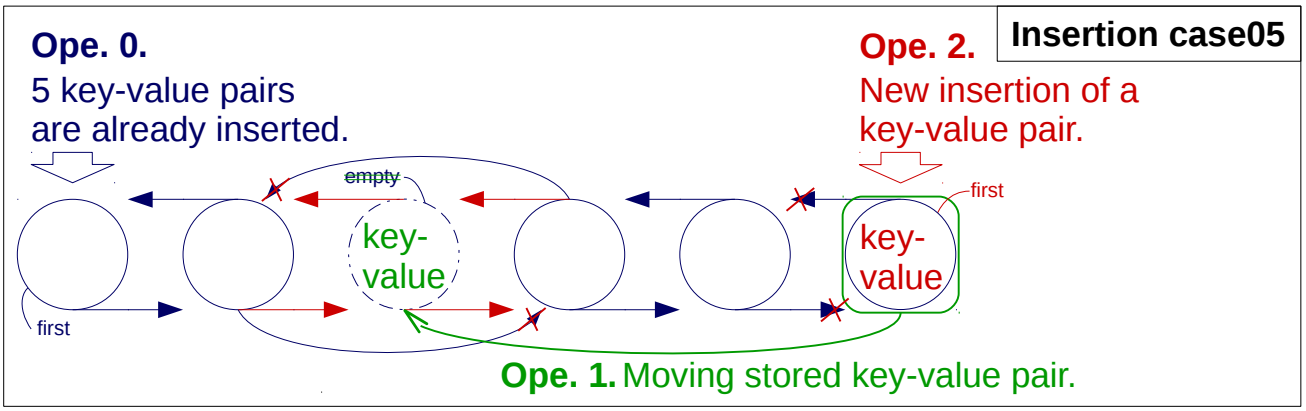

Figure 2.11. Insertion case05.

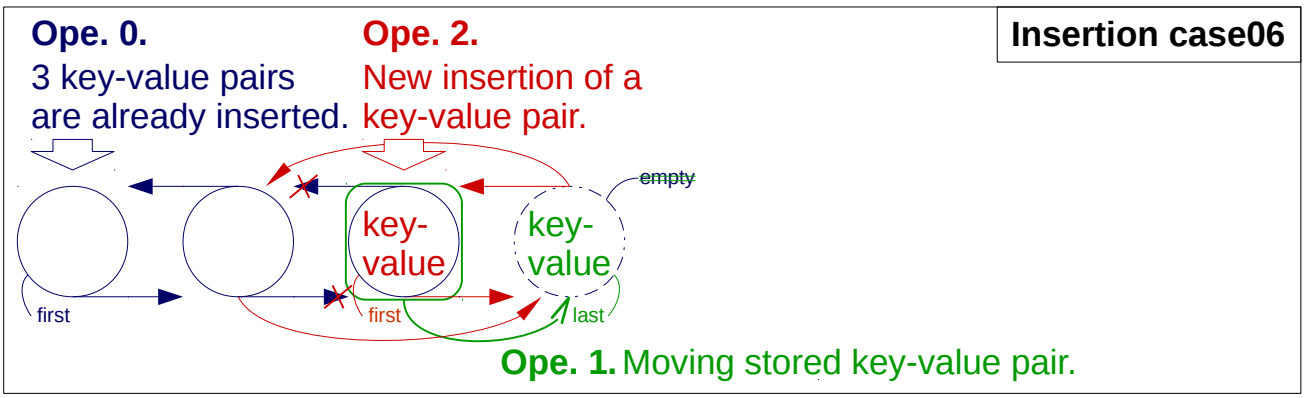

Figure 2.12. Insertion case06.

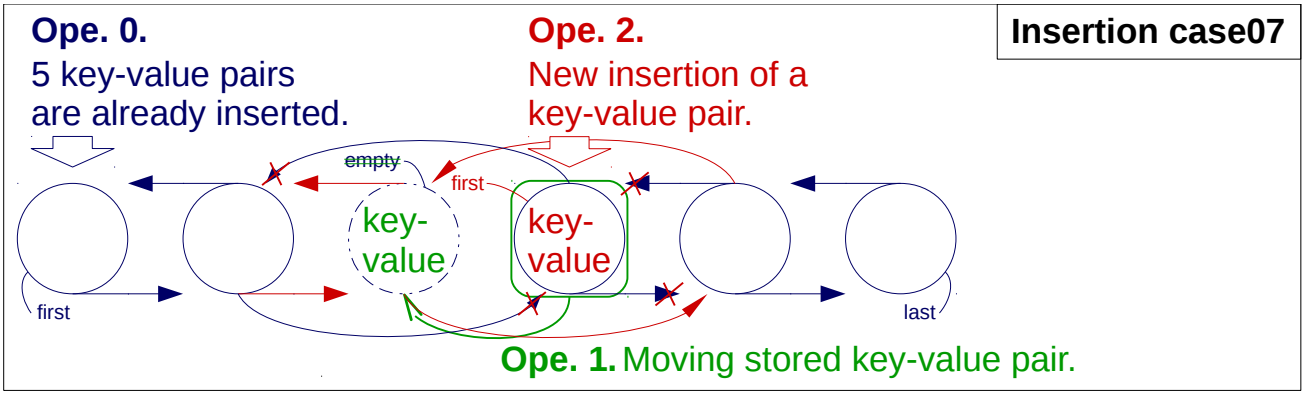

Figure 2.13. Insertion case07.

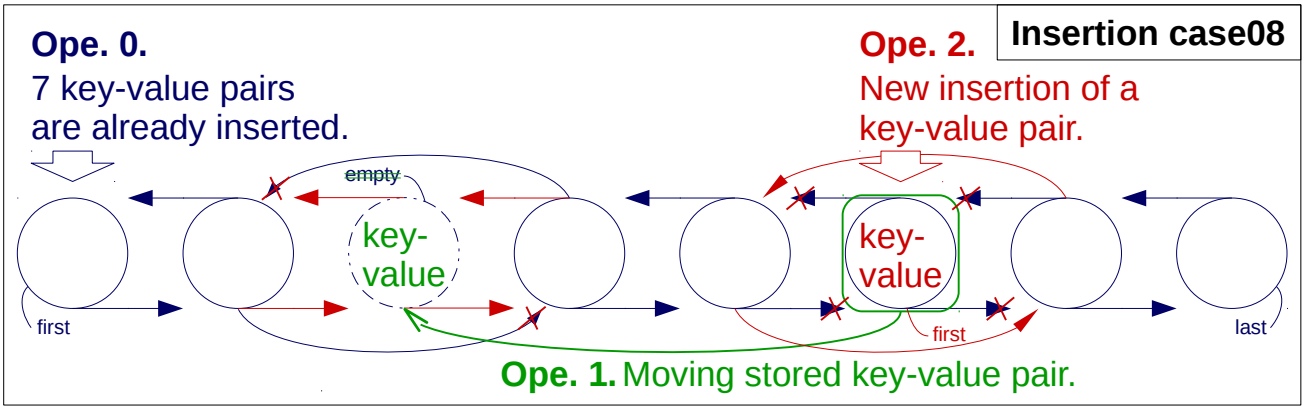

Figure 2.14. Insertion case08.

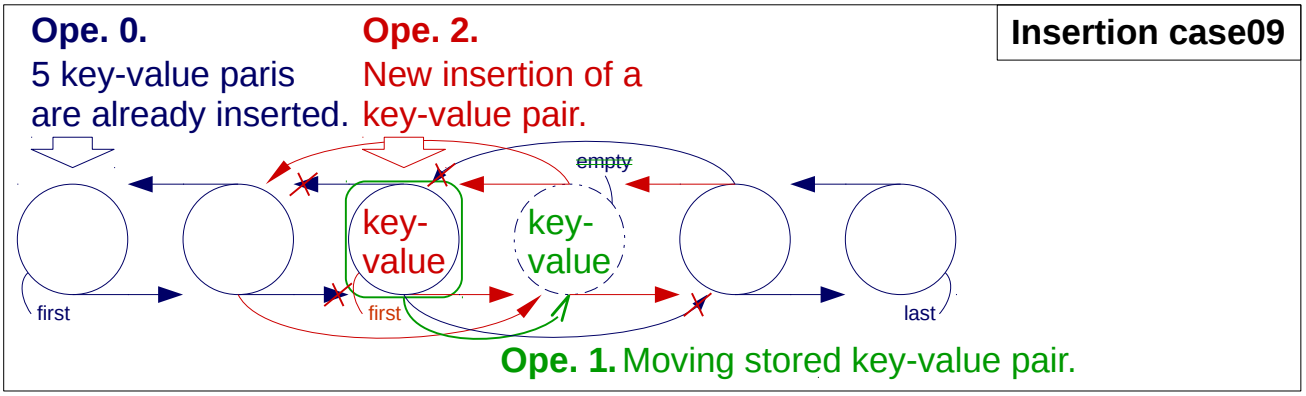

Figure 2.15. Insertion case09.

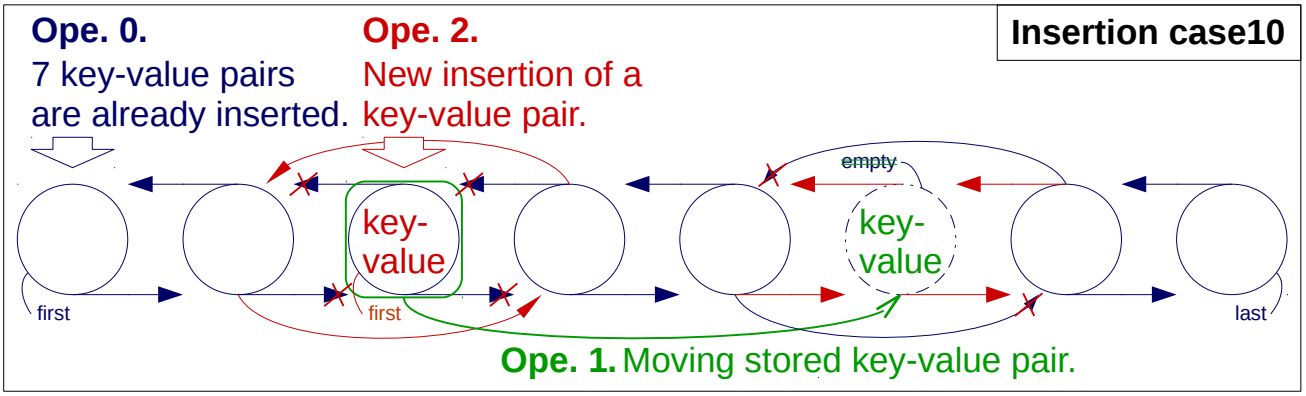

Figure 2.16. Insertion case10.

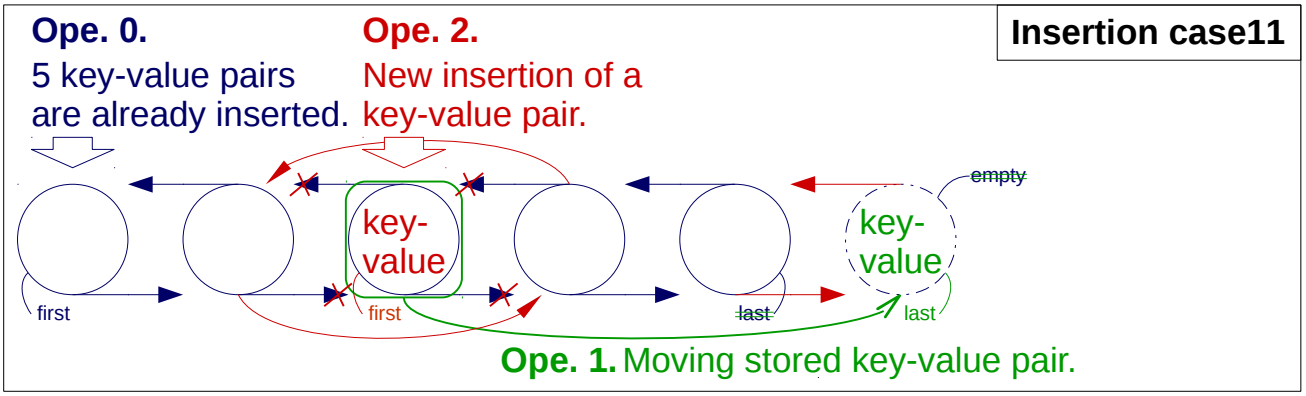

Figure 2.17. Insertion case11.

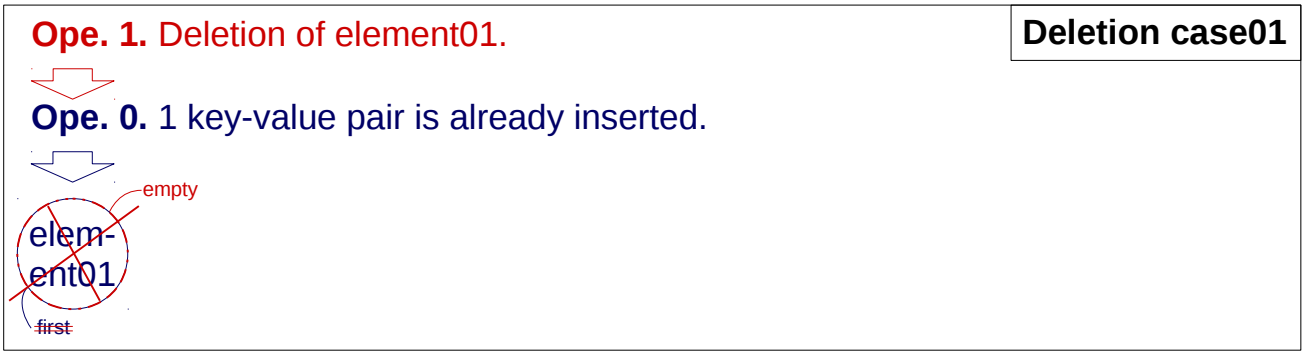

Figure 2.18. Deletion case01.

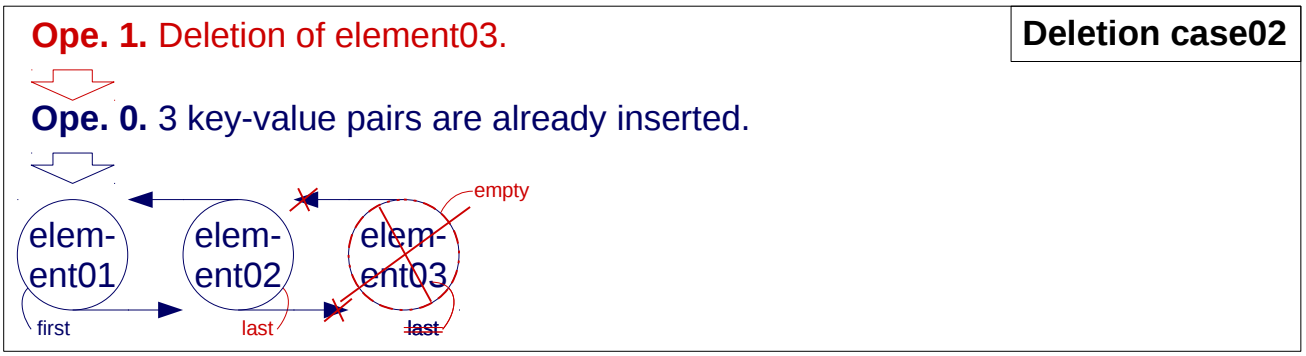

Figure 2.19. Deletion case02.

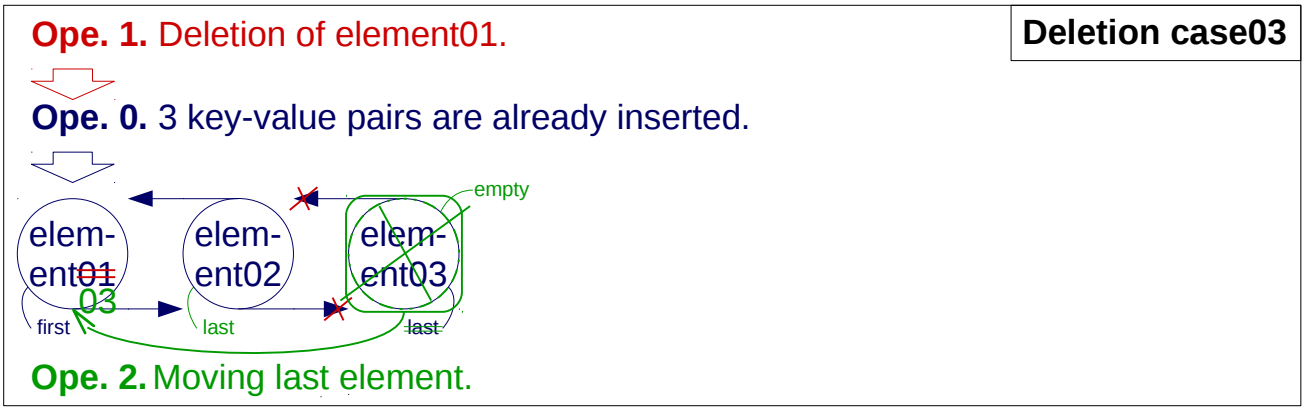

Figure 2.20. Deletion case03.

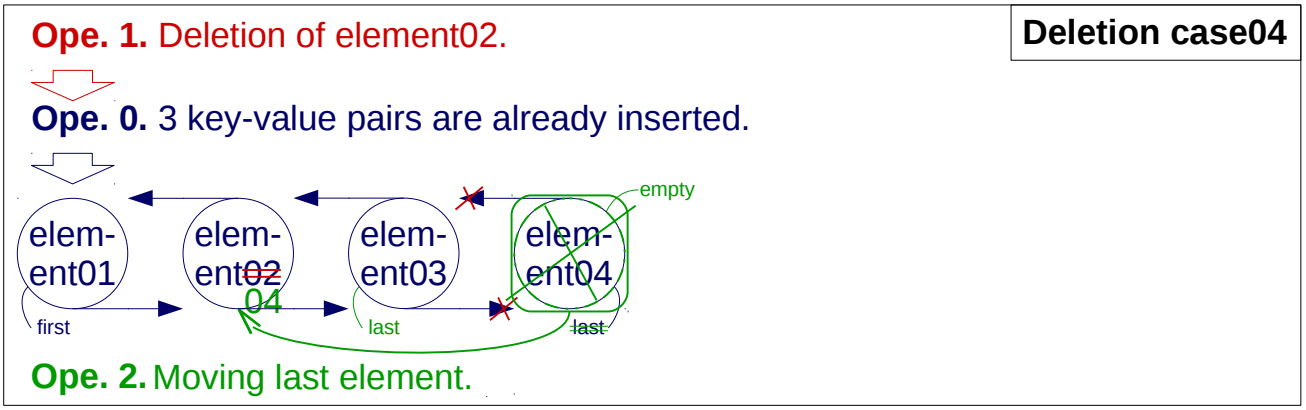

Figure 2.21. Deletion case04.

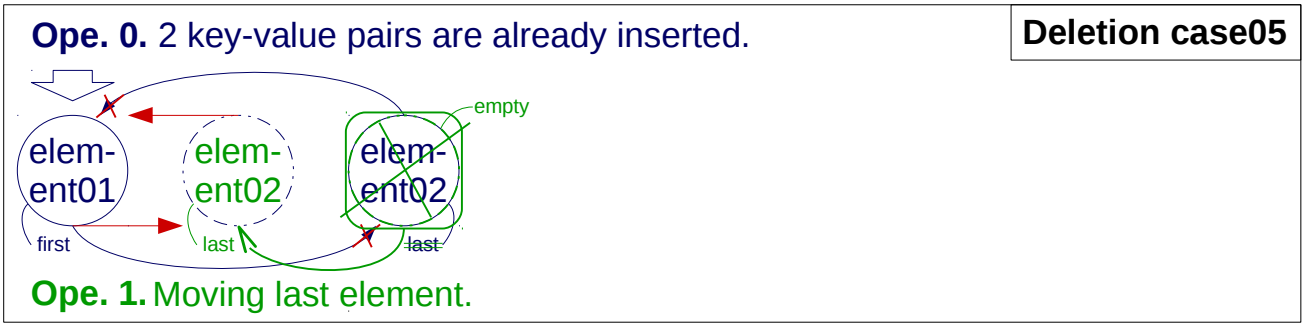

Figure 2.22. Deletion case05. The process of reducing doubly linked list fragmentation. Link distance is reduced by moving a distant element to nearby empty element found by sequential search. As the element moves, a new empty element is created. To completely eliminate fragmentation, it must be performed recursively at least until the linked distance reaches a local minimum. Therefore, the execution cost is very high.

図 2.22 Deletion case05. 双方向リストの断片化を低減する処理．遠くにある要素を，線形探査により探した近くの空き要素へ移動させることでリンク距離を短くする．要素が移動すると新しい空き要素ができるため，断片化を完全に解消するには，少なくともリンク距離が局所的最小値となるまで再帰的に実行する必要がある．このため，実行コストは非常に高い．

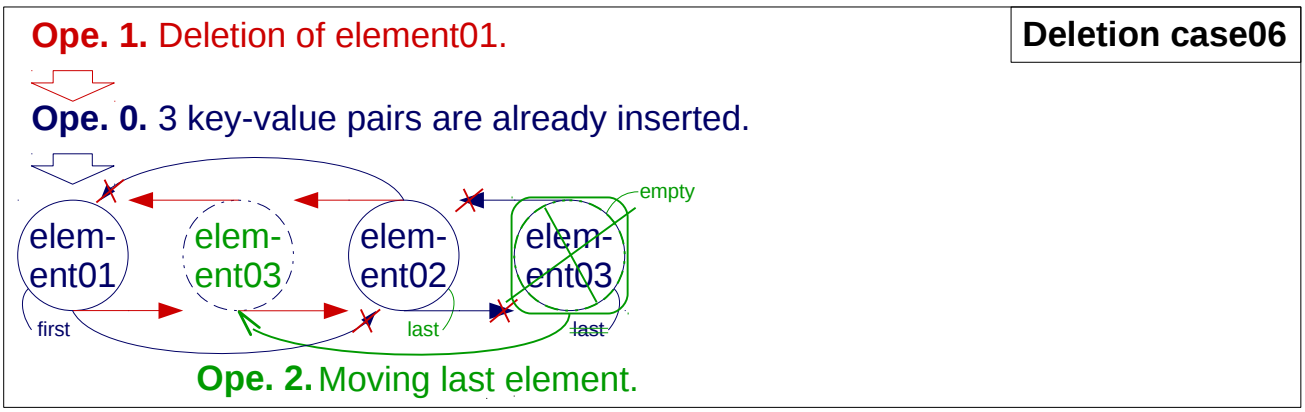

Figure 2.23. Deletion case06. The process of reducing doubly linked list fragmentation. Link distance is reduced by moving a distant element to nearby empty element found by sequential search. As the element moves, a new empty element is created. To completely eliminate fragmentation, it must be performed recursively at least until the linked distance reaches a local minimum. Therefore, the execution cost is very high.

図 2.23 Deletion case06. 双方向リストの断片化を低減する処理．遠くにある要素を，線形探査により探した近くの空き要素へ移動させることでリンク距離を短くする．要素が移動すると新しい空き要素ができるため，断片化を完全に解消するには，少なくともリンク距離が局所的最小値となるまで再帰的に実行する必要がある．このため，実行コストは非常に高い．

第 3 章

実装

第 4 章で使用するベンチマーク用コードについて説明する.

3.1 ベンチマーク用コード

ソースコード

本投稿で使用したソースコードを下記に示す.

https://github.com/admiswalker/InPlaceChainedHashTable-IpCHashT-

ファイル構成

上記に示すソースコードのファイル構成を Table 3.1 に示す.

Table 3.1. File organization.

File or directory name	Description
bench	Results of the benchmarks on this post
tmpBenck	Output directory of the benchmarks
tmpDir	Temporary directory for graph plotting
CHashT.hpp	Inplimentation of "sstd::CHashT"
FNV_Hash.cpp	Hash function for only implimentation test [a)]
FNV_Hash.hpp	Hash function for only implimentation test [a)]
IpCHashT.hpp	Inplimentation of "sstd::IpCHashT" (Proposing method)
Makefile	Makefile
README.md	Read me file
SubStandardLibrary-SSTD–master.zip	Convenient functions set [b)]
bench.hpp	Benchmark
exe_bench	Binary file for benchmark
exe_bench_uM	Binary file for benchmarking allocated memory size called from exe_bench
exe_sProc	Binary file for statistical process of "main_sProc.cpp"
exe_test	Binary file for test of "test_CHashT.hpp" and "test_IpCHashT.hpp"
flat_hash_map-master.zip	Inplimentation of "ska::flat_hash_map" [c)]
googletest-master.zip	Google's C++ test framework [d)]
main_bench.cpp	Entry point for benchmark of "bench.hpp"
main_bench_usedMemory.cpp	Entry point for benchmark of allocated memory size of "bench.hpp"
main_sProc.cpp	Entry point for the benchmarks statistical process.
main_test.cpp	Entry point for test of "test_CHashT.hpp" and "test_IpCHashT.hpp"
plots.py	Plotting funcrions for benchmark
sparsehash-master.zip	Inplimentation of "google::dense_hash_map" [e)]
test_CHashT.hpp	Test code for "CHashT.hpp"
test_IpCHashT.hpp	Test code for "IpCHashT.hpp"
typeDef.h	Type definitions for integer

Origins: [a)] https://qiita.com/Ushio/items/a19083514d087a57fc72, [b)] https://github.com/admiswalker/SubStandardLibrary, [c)] https://github.com/skarupke/flat_hash_map, [d)] https://github.com/google/googletest, [e)] https://github.com/sparsehash/sparsehash

環境／ Software Environment

For benchmarking, g++, git, make, cmake, and python with matplotlib is required.

環境は Ubuntu を想定しており，コンパイルには G++ が必要である．本体のビルドには Make が，Google Test のビルドに CMake が必要となる．また，グラフのプロットには Python インタプリタと matplotlib が必要となる．

実行手順

実行手順は，Fig. 3.1 に示す通りである．

```
$ git clone git@github.com:admiswalker/InPlaceChainedHashTable-IpCHashT- -b v1.0.0
$ cd ./InPlaceChainedHashTable-IpCHashT-
$ make
$ ./exe_test
$ ./exe_bench
$ ./exe_sProc
```

Figure 3.1. The benchmarking process. 1. Cloning benchmark files from Git. 2. Chane directory. 3. Compile. 4. Run test code. 5. Run benchmark. 6. Run statistical processes.

図 3.1 ベンチマークの実行手順. ソースコードを Git リポジトリからクローンし，ディレクトリを移動，コンパイル，テストコードの実行，ベンチマークの実行，計算結果の統計処理を行っている．

sstd::IpCHashT のオプション

sstd::IpCHashT は，多数のオプションを備えており，ベンチマークで使用するオプションについては，Table 3.2 に示す alias を IpCHashT.hpp に定義する．なお，IpCHashT_u16 については，最大 load factor を 50 % にする意味がないため，alias は設けない．他のオプションは，IpCHashT.hpp に定義されており，#define マクロにより，soft insertion か hard insertion と，素数サイズのテーブルを用いて剰余演算するか，サイズ $2^k - 1$ $(k = 1, 2, ...)$ のテーブルを用いてビットマスクするかを選択できる．

Table 3.2. sstd::IpCHashT aliases.

Alias	T_shift	Maximum load factor [%]	Search option
IpCHashT_u8hS	uint8	50 (half)	Successful search major option
IpCHashT_u8fS	uint8	100 (full)	Successful search major option
IpCHashT_u16hS	uint16	50 (half)	Successful search major option
IpCHashT_u16fS	uint16	100 (full)	Successful search major option
IpCHashT_u8hU	uint8	50 (half)	Unsuccessful search major option
IpCHashT_u8fU	uint8	100 (full)	Unsuccessful search major option
IpCHashT_u16hU	uint16	50 (half)	Unsuccessful search major option
IpCHashT_u16fU	uint16	100 (full)	Unsuccessful search major option

ベンチマークのオプション

ベンチマークのオプションは，bench.hpp に定義されている．Successful search major option か unsuccessful search major option かの選択は，コメントアウトにより手動で行う．

軽くテストする場合は，limitSize を 5×10^6 として $ make, $ ./exe_bench を実行すると，./tmpBench ディレクトリが生成され，10 分程で ベンチマーク結果が保存される．グラフのスケールが合っていない場合は，plots.py を調整する必要がある．

本投稿に示すベンチマークは，limitSize を 2×10^8，loopNum を 100 として，$ make, $ ./exe_bench を実行する．各ベンチマークを 100 回ずつ，tmpBench へ CSV ファイルとして出力するには，数日掛かる．次に $ ./exe_sProc を実行すると，CSV ファイルから中央値が計算され，グラフとして出力される．中央値を用いるのは，分布の形状が不明なこと，平均処理ではグラフのエッジが潰れること，ベンチマーク中の外れ値を除去することが目的である．loopNum は 25 程度でも綺麗な結果が得られるため，まずは 25 で試すとよい．

第 4 章

ベンチマーク

4.1 環境

ベンチマークの実行環境を，Table 4.1 に示す．

Table 4.1. Benchmark execution environment

Component	Type
CPU	AMD Ryzen7 1700 (8Cores/16Threads)
	Base Clock 3GHz / Max Boost Clock 3.7GHz
	Total L1 Cache: 768KB / Total L2 Cache: 4MB / Total L3 Cache: 16MB
Memory	DDR4-2666 32GB
OS	Ubuntu 16.04 LTS
Compiler	gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1 16.04.11)

4.2 各ハッシュテーブルの概要と測定条件

各ハッシュテーブルの key と value に uint64 型を指定して測定する．テーブルごとの設定を次に示す．

std::unordered_map<uint64,uint64>
C++ の標準ライブラリに収録されているハッシュテーブル．

sstd::CHashT<uint64,uint64>
“Separate chaining with list head cells” [1] の実装の 1 つ．アルゴリズム比較用のサンプル実装として，最もシンプルなハッシュテーブルアルゴリズムの 1 つを選択した．本実装では，テーブルサイズを $2^k - 1$ $(k = 1, 2, ...)$ とし，ハッシュ値の最下位 k ビットを配列インデックスとする．全要素数がテーブルサイズを超える場合はリハッシュする．

[1] “Separate chaining with linked lists” がテーブルにポインタしか持たず，ポインタで接続した先から初めて key-value ペアを持つのに対して，“Separate chaining with list head cells” ではテーブルにも key-value ペアを格納する．

sstd::IpCHashT<uint64,uint64> (as uint8 and maxLF50)

提案アルゴリズムの実装の 1 つ．uint8 により双方向リストを構成する．Load factor の最大値を 50 % に制限している．テーブルサイズを $2^k - 1$ $(k = 1, 2, ...)$ とし，ハッシュ値の最下位 k ビットを配列インデックスとする．bench.hpp 内では別名として iHashT_u8h を定義する．

sstd::IpCHashT<uint64,uint64> (as uint8 and maxLF100)

提案アルゴリズムの実装の 1 つ．uint8 により双方向リストを構成する．Load factor の制限はなく，link 距離が “uint8 の最大値 $-1 = 254$” を超えるか，テーブルが全て埋まるまで要素を挿入する．テーブルサイズを $2^k - 1$ $(k = 1, 2, ...)$ とし，ハッシュ値の最下位 k ビットを配列インデックスとする．bench.hpp 内では別名として iHashT_u8f を定義する．

sstd::IpCHashT<uint64,uint64> (as uint16 and maxLF100)

提案アルゴリズムの実装の 1 つ．uint16 により双方向リストを構成する．Load factor の制限はなく，link 距離が “$uint16$ の最大値 $-1 = 65534$” を超えるか，テーブルが全て埋まるまで要素を挿入する．テーブルサイズを $2^k - 1$ $(k = 1, 2, ...)$ とし，ハッシュ値の最下位 k ビットを配列インデックスとする．bench.hpp 内では別名として iHashT_u16 を定義する．

google::dense_hash_map<uint64,uint64>

Quadratic probing の実装の 1 つ．メモリ効率が高く，探査速度が速い一方，キーの値の 1 つを空符号に，削除を行う場合は更にもう 1 つのキー値を削除符号として登録する必要がある．登録された値は，キーとして使用できない．登録には set_empty_key() 関数と set_deleted_key() 関数を用いる．テーブルサイズは $2^k - 1$ $(k = 1, 2, ...)$ であり，ハッシュ値の最下位 k ビットを配列インデックスとしている．

ska::flat_hash_map<uint64,uint64,ska::power_of_two_std_hash<uint64»

Robin Hood Hashing の実装の 1 つ．ska::power_of_two_std_hash<uint64> オプションにより，テーブルサイズを $2^k - 1$ $(k = 1, 2, ...)$ としており，ハッシュ値の最下位 k ビットを配列インデックスとしている．

4.3　結果

ここでは，IpCHashT の双方向リストを uint8 と uint16 により構成した設定と，load factor の最大値を 50 % に制限した設定に加えて，要素探査時に successful search を優先する設定と，unsuccessful search を優先する設定についてベンチマークする．

最大 Loadfactor

最大 load factor とテーブルサイズの関係を Fig. 4.1 に示す．

メモリ使用量

メモリ使用量とテーブルサイズの関係を Fig. 4.2，4.3 に示す．

挿入

挿入速度とテーブルサイズの関係を Fig. 4.4，4.5，4.6 に示す．挿入処理において，hard insertion では find() 関数を用いるが，soft insertion では，find() 関数を用いない．本ベンチマークでは，soft insertion のみを用いている．したがって，successful search major option と unuccessful search major option による違いはない.

探査

探査速度とテーブルサイズの関係を Fig. 4.7，4.9，4.8，4.10 に示す．Fig. 4.7，4.8 は，コンパイル時 IpCHashT に successful search major option を指定した結果，Fig. 4.9，4.10 は，コンパイル時 IpCHashT に unsuccessful search major option を指定した結果である．

削除

削除速度とテーブルサイズの関係を Fig. 4.11，4.12 に示す．Fig. 4.11 は successful search major option を指定した結果，Fig. 4.12 は unsuccessful search major option を指定した結果である．

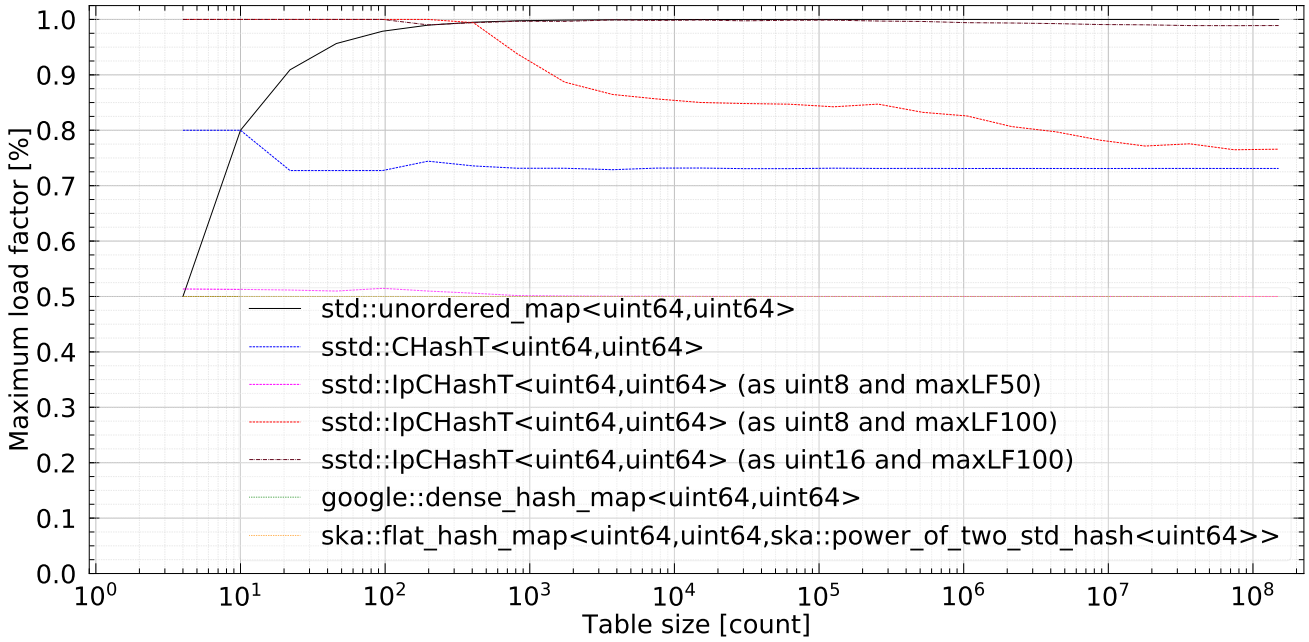

Figure 4.1. Maximum load factor which is median value of 100 samples. Maximum load factor of IpHashT (uint16, maxLF100) is limitted by its maximum length of shift_T. Maximum load factor of IpHashT (uint8, maxLF50), dense_hash_map, and flat_hash_map is artificially limited by 50 %.

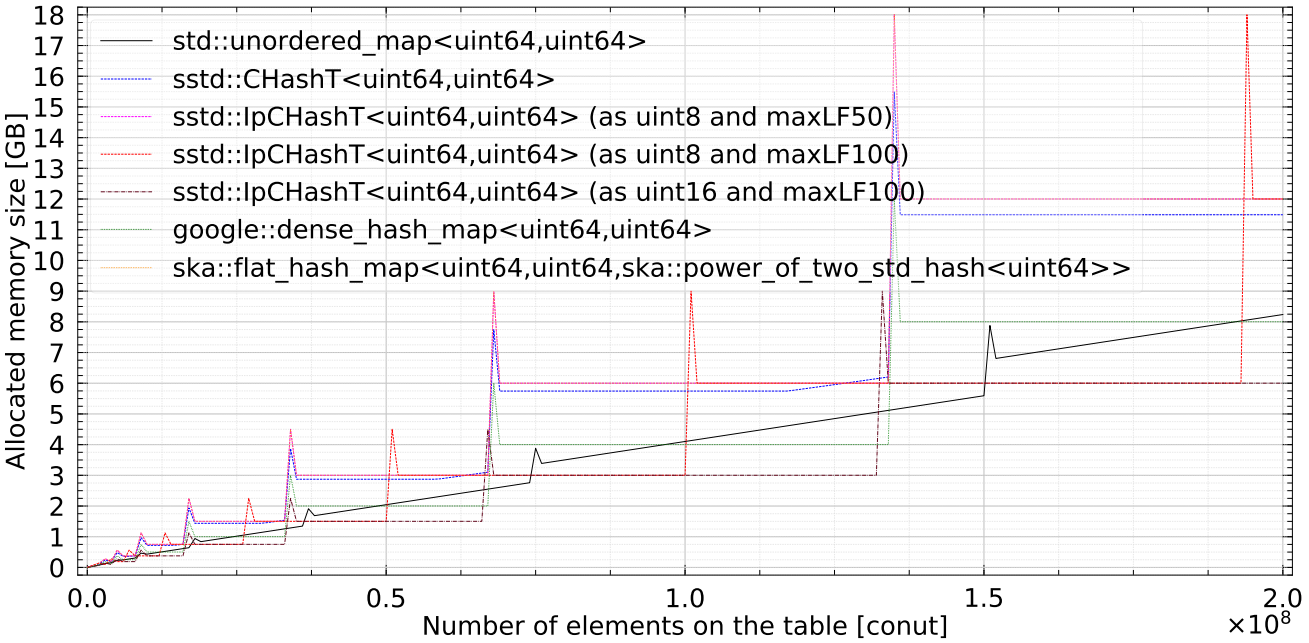

Figure 4.2. Allocated memory size, which is one sample raw data. Peaks of memory size indicate rehash timmings. And the width of peaks is measurement interval since the ture peak is as narrow as one step of x axis. unordered_map seems to allocating memory each by insertion. IpCHashTs are moved the timming of rehashing back and forth by the influence of maximum load factor. flat_hash_map while it is hard to see, is plotted on the same place of IpCHashT (uint8, maxLF50).

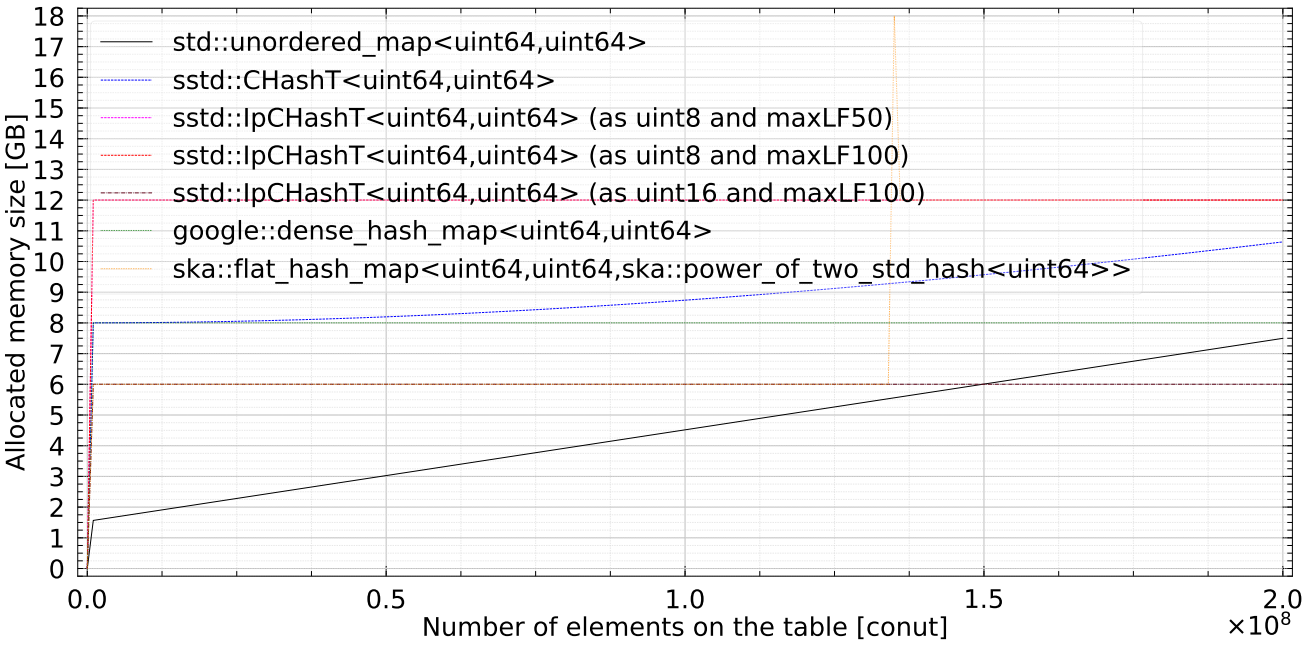

Figure 4.3. Allocated memory size, which is one sample raw data. In this measurement, each hash table is pre-allocated with 2.0×10^8. IpCHashT (uint8, maxLF100), while it is hard to see, is plotted on the same place of IpCHashT (uint8, maxLF50). flat_hash_map is initializing its table size with 2.0×10^8 and the others are initializing their table size with 2.0×10^8 as a containable size. This means that the one with a rehashing peak is flat_hash_map.

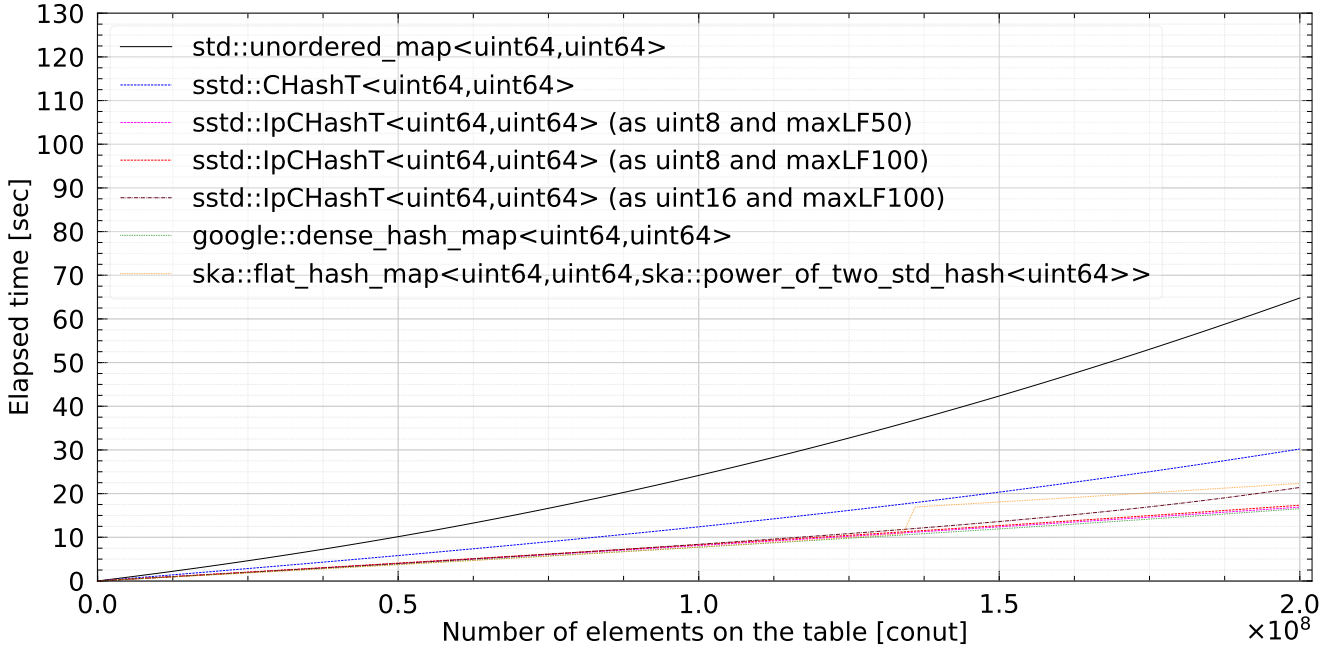

Figure 4.4. Total time of insertion, which is a median value of 100 samples, using pre-allocated table. In this measurement, each hash table is pre-allocated with 2.0×10^8. A discontinuous value around 1.25~$.375 \times 10^8$ represents rehashing of flat_hash_map. This is due to differences in the algorithms that determine the initial table size.

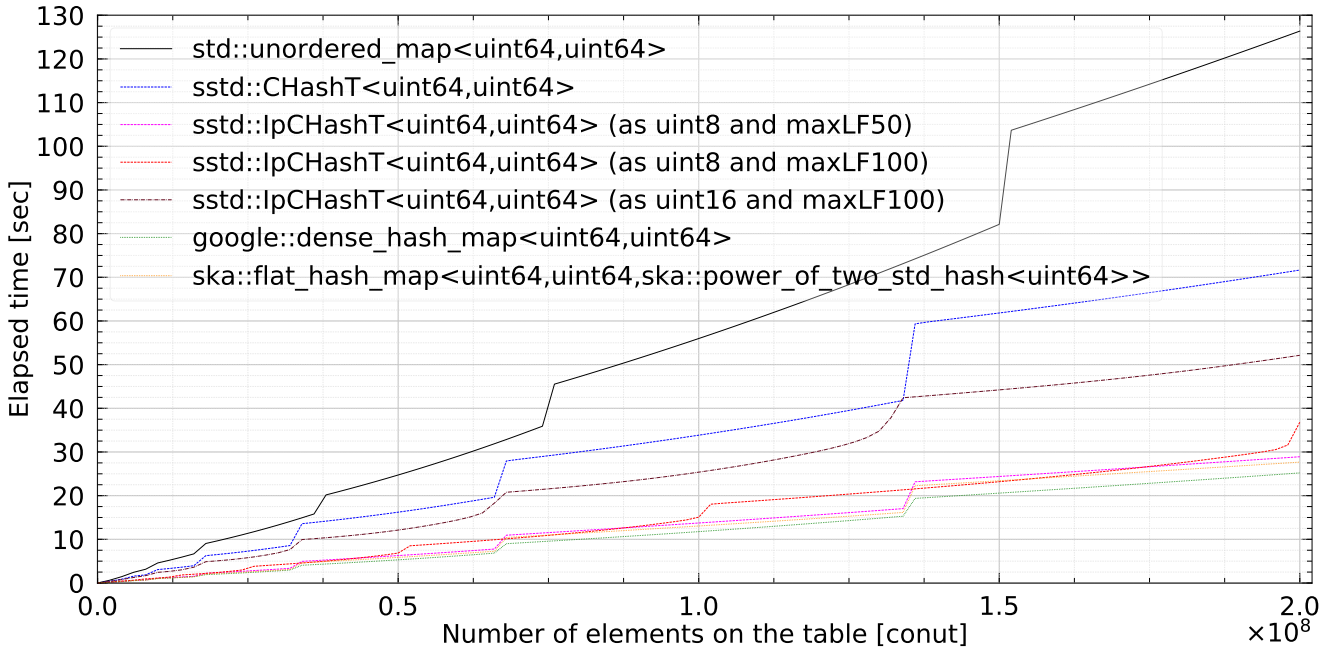

Figure 4.5. Total time of insertion, which is a median value of 100 samples, with rehashing table. Chaining type of hash tables like, CHashT and IpCHashTs (especially uint16-maxLF100 option), are slow down its insertion speed as the load factor increase.

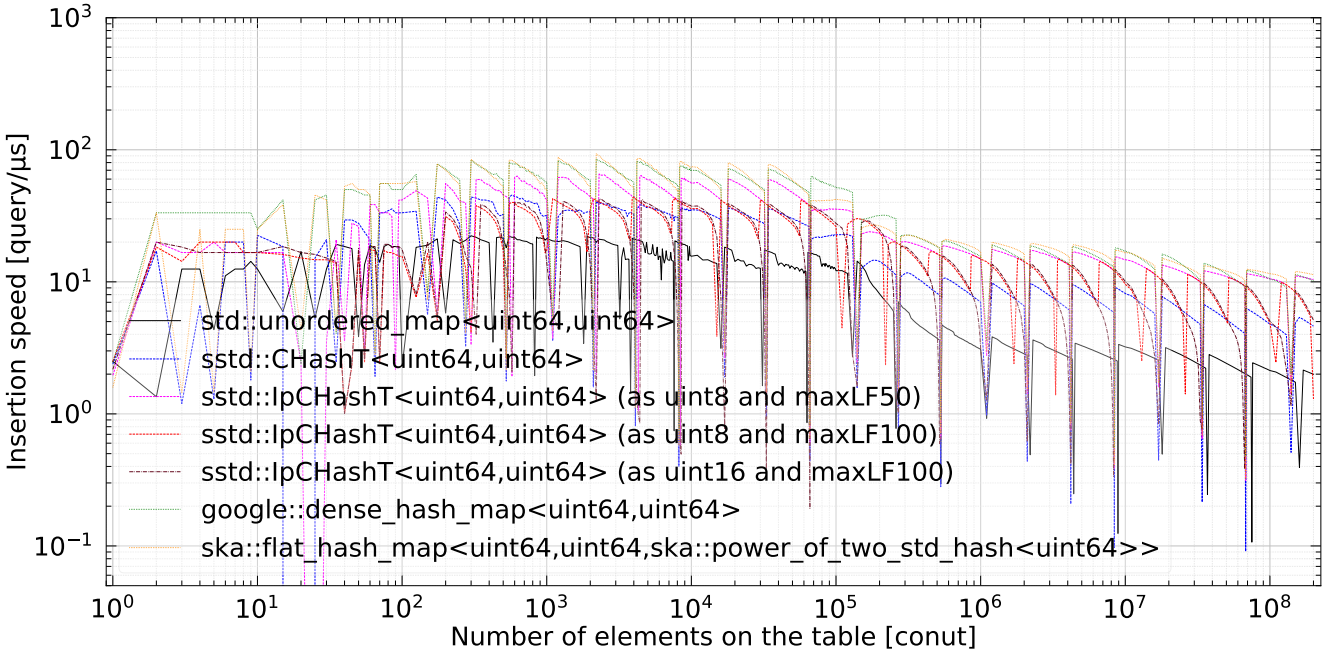

Figure 4.6. Insertion speed, which is a median value of 100 samples. The spikes formed by the valleys represent rehashings. The CPU cache line is around 1.0×10^5 elements for 4 MB L2 cache and around 1.0×10^6 elements for 16 MB L3 cache on AMD Ryzen7 1700.

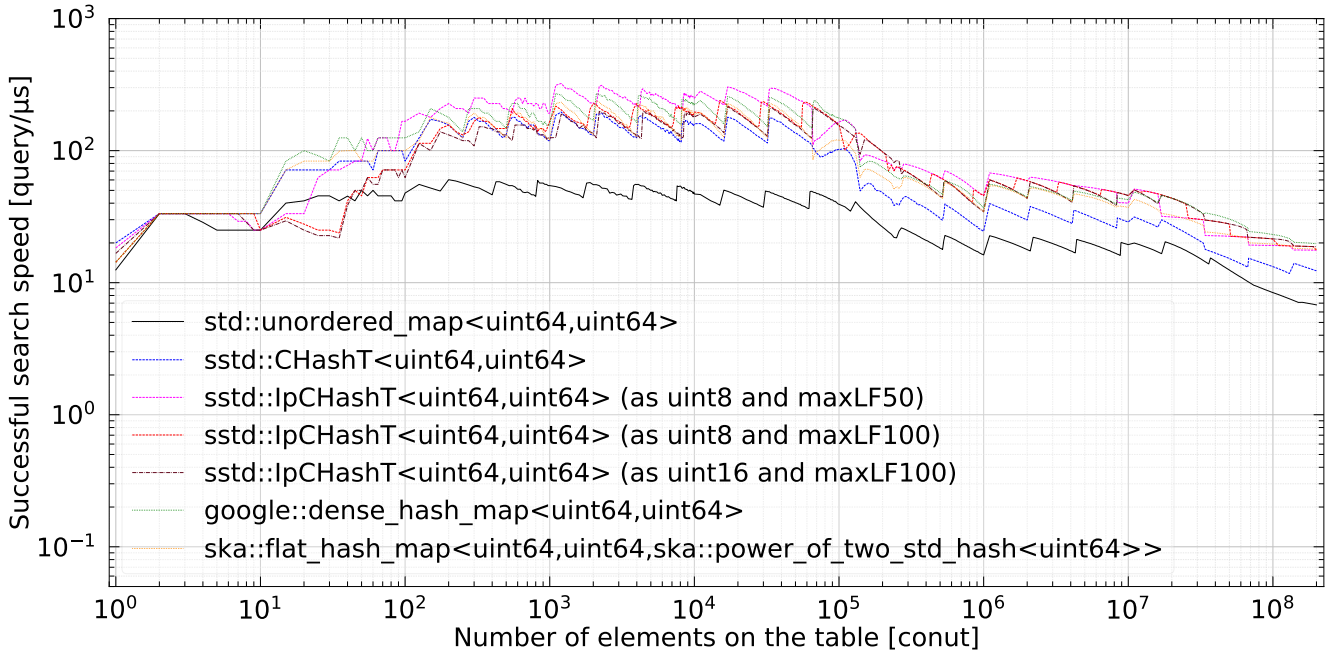

Figure 4.7. (Successful search major option). Successful search speed, which is a median value of 100 samples. About 1.0×10^5 elements will consume totally 4 MB of L2 cache, and about 1.0×10^6 elements will consume totally 16 MB of L3 cache on AMD Ryzen7 1700.

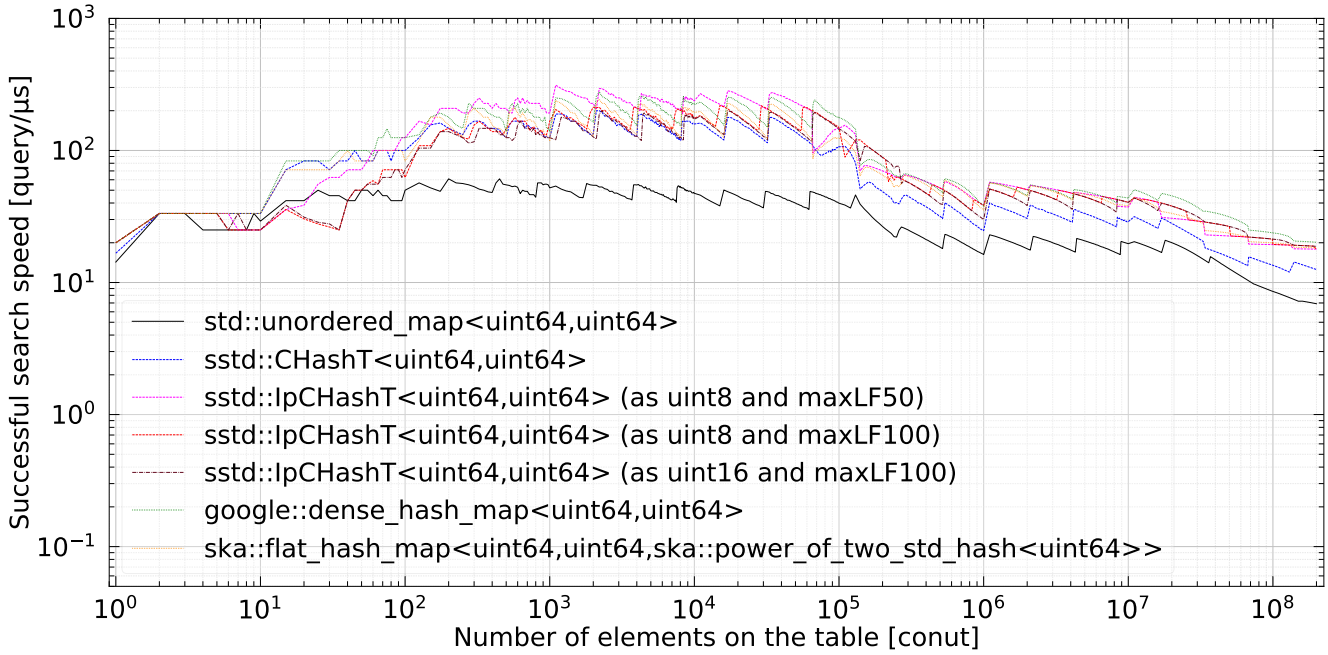

Figure 4.8. (Unsuccessful search major option). Successful search speed, which is a median value of 100 samples. About 1.0×10^5 elements will consume totally 4 MB of L2 cache, and about 1.0×10^6 elements will consume totally 16 MB of L3 cache on AMD Ryzen7 1700.

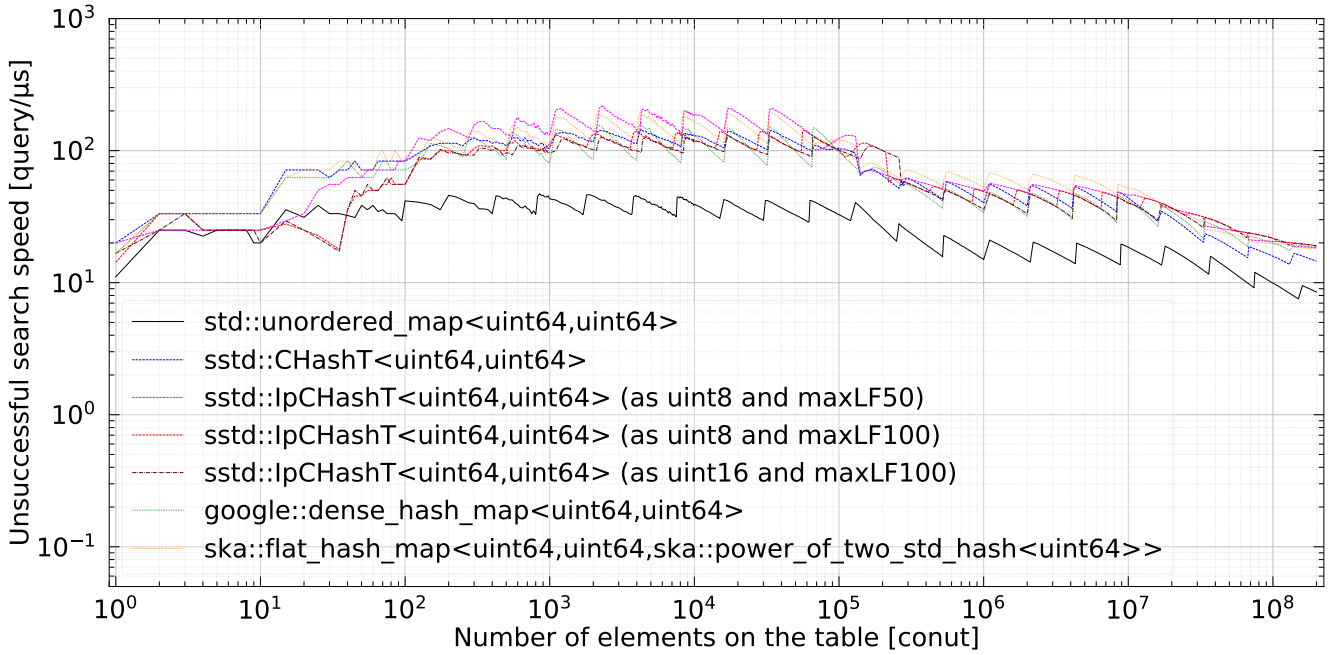

Figure 4.9. (Successful search major option). Unsuccessful search speed, which is a median value of 100 samples. About 1.0×10^5 elements will consume totally 4 MB of L2 cache, and about 1.0×10^6 elements will consume totally 16 MB of L3 cache on AMD Ryzen7 1700.

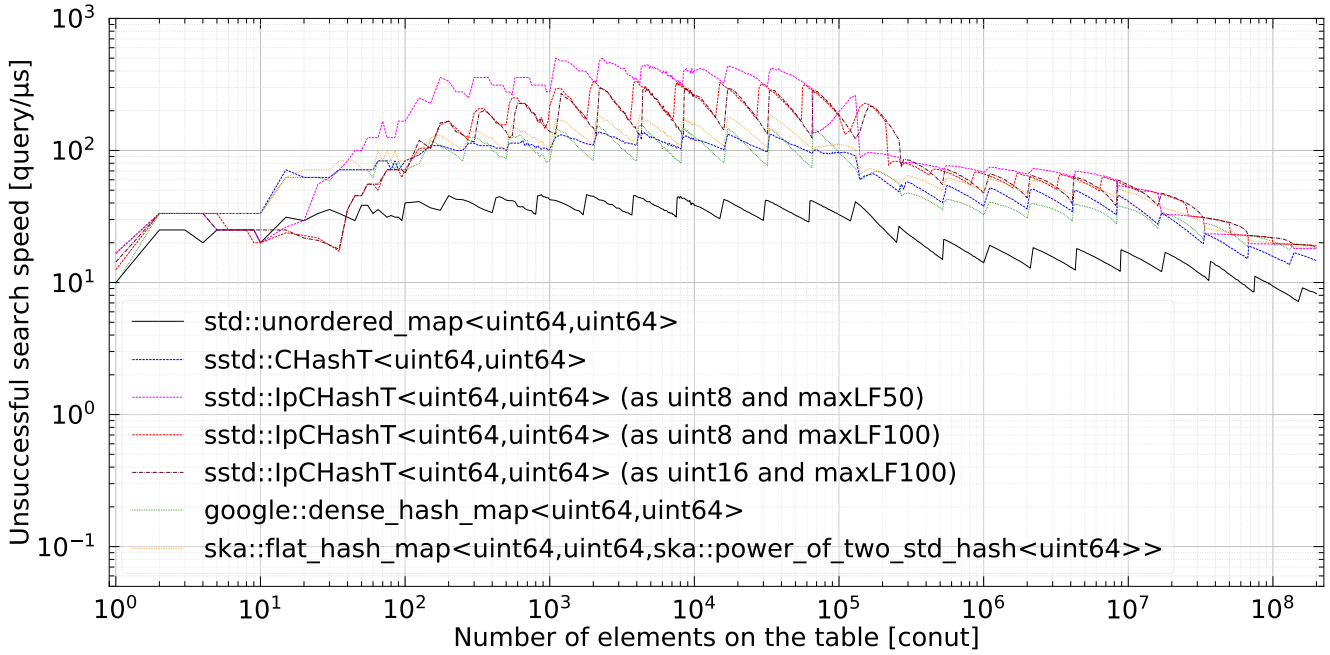

Figure 4.10. (Unsuccessful search major option). Unsuccessful search speed, which is a median value of 100 samples. About 1.0×10^5 elements will consume totally 4 MB of L2 cache, and about 1.0×10^6 elements will consume totally 16 MB of L3 cache on AMD Ryzen7 1700.

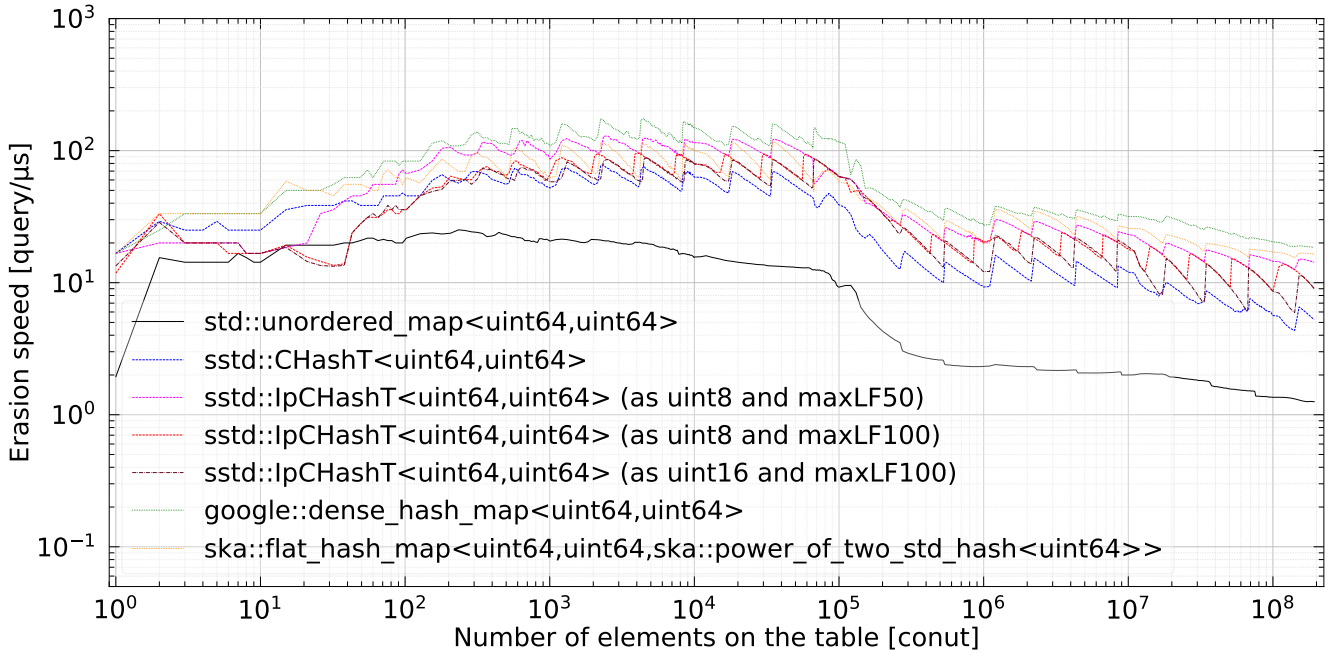

Figure 4.11. (Successful search major option). Erasion speed, which is a median value of 100 samples. About 1.0×10^5 elements will consume totally 4 MB of L2 cache, and about 1.0×10^6 elements will consume totally 16 MB of L3 cache on AMD Ryzen7 1700.

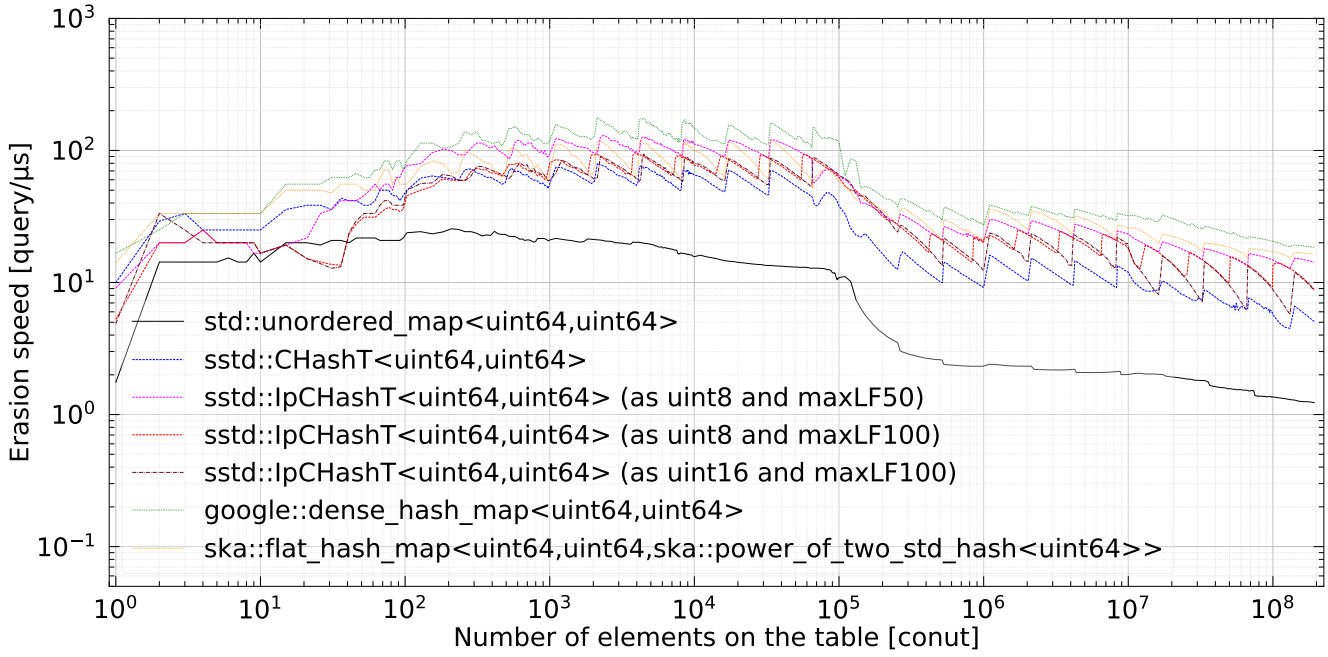

Figure 4.12. (Unsuccessful search major option). Erasion speed, which is a median value of 100 samples. About 1.0×10^5 elements will consume totally 4 MB of L2 cache, and about 1.0×10^6 elements will consume totally 16 MB of L3 cache on AMD Ryzen7 1700.

第 5 章

考察

最大 load factor，メモリ使用量，挿入速度，探査速度，削除速度について考察する．

最大 load factor

Fig. 4.1 に示す，最大 load factor とテーブルサイズの関係について考察する．**unordered_map** はテーブルサイズ 10^2 未満で load factor が低いものの，区間 10^2〜10^8 において，高い load factor を示している．**CHashT** は区間 10^1〜10^8 に渡り 72.5 % 以上を維持している．**IpCHashT (uint8, maxLF50)**，**dense_hash_map**，**flat_hash_map** は，load factor が制限されており，50 % に留まっている．**IpCHashT (uint8, maxLF100)** は，テーブルサイズ 10^3 未満では 90 % 以上の load factor を示す．テーブルサイズが 4×10^2 を超える辺りから，双方向リストの長さが “uint8 の最大値 $- 1 = 254$” に制限されている影響により，load factor が単調に減少していく．**IpCHashT (uint16, maxLF100)** は区間 10^1〜10^8 に渡り 97.5 % 以上を維持している．なお，実際には 100 サンプルの中央値のため，特に CHashT と IpCHashT (uint8, maxLF100) では，特にテーブルサイズが 10^3 未満の場合において，ある程度揺らぎがある．なお，IpCHashT の実装はいずれもパディングされる配列長を load factor の計算に加算している．このため，**IpCHashT (uint8, maxLF50)** では端数が発生し，丁度に 50 % とはならない．

メモリ使用量

Fig. 4.2 に示す，メモリ使用量とテーブルサイズの関係について考察する．複数のピークはリハッシュを示す．ピークは幅を持っているが，これは測定間隔に等しく，実際には要素 1 つ分の幅しか持たない．**unordered_map** は要素数にしたがって，ほぼ線形にメモリ使用量を増加させている．これは，unordered_map が要素ごとにメモリを確保することを示す．また，リハッシュ時のピークも小さいことから，アルゴリズムは “Separate chaining with linked lists” であると推察される．**IpCHashT (uint16, maxLF100)** は殆どの区間で最も高いメモリ効率を示しており，Fig. 4.1 で示した最大 load factor の高さを反映する結果となった．一部，他のハッシュテーブルとは異なるタイミングでリハッシュが発生しており，load factor が 100 % まで達していないことを示している．**dense_hash_map** は load factor が 50 % に制限されているにも関わらず，unordered_map 前後のメモリ使用量を示しており，キーの一部を空符号や削除符号として使用することで，メモリ効率を高める実装の特性が現れている．**CHashT**，**IpCHashT (uint8, maxLF50)**，**flat_hash_map** は，いずれもほぼ同じメモリ使用量を示している．Fig. 4.1 より，**CHashT** は区間 10^1〜10^8 に渡り 72.5 % 以上の最大 load factor を維持しているものの，ポインタによる片方向リストの構築には，1 要素あたり 8 Byte[1)] 必要としており，多くのメモリを消費する結果となった．Load factor の高い区間において，メモリ使用量が増加しており，ハッシュ先が衝突した分のメモリを動的に確保している．

[1)] 64 bits CPU のため．

IpCHashT (uint8, maxLF50)，**flat_hash_map** は探査速度を得るため，単位要素あたり最も多くのメモリを消費しており，unordered_map と比較して 1.5 倍程度となる．実利用に際しては，このメモリ使用量を許容できるかどうかが，1 つの課題となる．**IpCHashT (uint8, maxLF100)** は IpCHashT (uint8, maxLF50) と IpCHashT (uint16, maxLF100) のおよそ中間でリハッシュする挙動を示している．

dense_hash_map と **IpCHashT (uint8, maxLF50)**，**IpCHashT (uint16)** の理論値なメモリ使用量と Fig. 4.2 に示す実測値を比較する．まず，**dense_hash_map** は，

$$\begin{aligned}\text{memory size}|_{\text{type=dense\_hash\_map}} &= (\text{key size} + \text{value size}) \times (\text{elements}/\text{max load factor})\ [\text{Byte}] \\ &= (\text{uint64} + \text{uint64}) \times (\text{elements}/0.5)\ [\text{Byte}] \\ &= (8 + 8) \times (\text{elements}/0.5)\ [\text{Byte}] \\ &= 32 \times \text{elements}\ [\text{Byte}]\end{aligned}$$

となる．また，コンパイラは配列アラインメントを大きい方に一致させるため，IpCHashT の uint8 や uint16 は，next probe と prev probe を合わせて uint64 として確保される．このため，**IpCHashT (uint8, maxLF50)** は，

$$\begin{aligned}\text{memory size}|_{\text{type=IpCHashT(uint8,maxLF50)}} &= (\text{key size} + \text{value size} + \text{next probe size} + \text{prev probe size}) \\ &\quad \times (\text{elements}/\text{max load factor})\ [\text{Byte}] \\ &= (\text{uint64} + \text{uint64} + \text{uint8} + \text{uint8}) \times (\text{elements}/0.5)\ [\text{Byte}] \\ &= (\text{uint64} + \text{uint64} + \text{uint64}) \times (\text{elements}/0.5)\ [\text{Byte}] \\ &= (8 + 8 + 8) \times (\text{elements}/0.5)\ [\text{Byte}] \\ &= 24 \times (\text{elements}/0.5)\ [\text{Byte}] \\ &= 48 \times \text{elements}\ [\text{Byte}]\end{aligned}$$

となる．同様に **IpCHashT (uint16, maxLF100)** は，

$$\begin{aligned}\text{memory size}|_{\text{type=IpCHashT(uint16,maxLF100)}} &= (\text{key size} + \text{value size} + \text{next probe size} + \text{prev probe size}) \\ &\quad \times (\text{elements}/\text{max load factor})\ [\text{Byte}] \\ &= (\text{uint64} + \text{uint64} + \text{uint16} + \text{uint16}) \times (\text{elements}/1.0)\ [\text{Byte}] \\ &= (\text{uint64} + \text{uint64} + \text{uint64}) \times (\text{elements}/1.0)\ [\text{Byte}] \\ &= (8 + 8 + 8) \times \text{elements}\ [\text{Byte}] \\ &= 24 \times \text{elements}\ [\text{Byte}]\end{aligned}$$

となる．

dense_hash_map と **IpCHashT (uint8, maxLF50)** の理論比を考えると，

$$\frac{32 \times \text{elements}\ [\text{Byte}]}{48 \times \text{elements}\ [\text{Byte}]} = \frac{2}{3} \approx 0.67$$

となる．実測値は，Fig. 4.2 より，要素数 1.5×10^8 のとき，8 GB と 12 GB であるから，

$$\frac{8\ [\text{GByte}]}{12\ [\text{GByte}]} = \frac{2}{3} \approx 0.67$$

となる．したがって，理論値と実測値は一致する．

dense_hash_map と **IpCHashT (uint8, maxLF50)** の理論比を考えると，

$$\frac{24 \times \text{elements}\ [\text{Byte}]}{32 \times \text{elements}\ [\text{Byte}]} = \frac{3}{4} = 0.75$$

となる．実測値は，Fig. 4.2 より，要素数 1.5×10^8 のとき，6 GB と 8 GB であるから，

$$\frac{6\ [\text{GByte}]}{8\ [\text{GByte}]} = \frac{3}{4} = 0.75$$

となる．したがって，理論値と実測値は一致する．

Fig. 4.3 に示す，ハッシュテーブルのサイズを 2×10^8 で初期化した場合の，メモリ使用量とテーブルサイズの関係について考察する．**flat_hash_map** は，テーブルサイズを 2×10^8 より大きな最小の $2^k - 1(1, 2, ...)$ として初期化しており，$2^k - 1|_{k=28} = 268435455$ としていることがわかる．実際に，リハッシュ区間は 1.250×10^8〜1.375×10^8 であり，load factor が 50 % となる要素数 $268435455/2 = 134217727.5$ を含んでいる．一方で，**unordered_map** や **CHashT**，**IpCHashTs**，**dense_hash_map** は，リハッシュせずに挿入可能な要素数が 2×10^8 となるように制御されている．このとき，Fig. 4.2 と Fig. 4.3 の比較から **CHashT**，**IpCHashTs**，**dense_hash_map** はほぼ倍の $2^k - 1|_{k=29}$ に，**unordered_map** は $2^k - 1|_{k=29}$ よりは小さいサイズに，それぞれ初期化されていると考えられる．加えて，**CHashT** は，テーブルサイズの増加に伴い，緩やかにメモリ使用量を増加させている．このため，Fig. 4.2 において，メモリ使用量の変化が折れ線に見えるのは，メモリ使用量測定時の解像度不足と考えられる．

挿入速度

Fig. 4.4，4.5，4.6 に示す，挿入速度とテーブルサイズの関係について考察する．

Fig. 4.4 は，テーブルのメモリを事前に確保した場合の挿入速度の累積時間である．ただし，Fig. 4.6 のように，挿入速度はテーブルサイズに依存しており，ここでは，2.0×10^8 に初期化されたハッシュテーブルの速度を示す．先の**メモリ使用量**の項目で示した通り，**flat_hash_map** は実テーブルサイズが 2.0×10^8 より大きな最小の $2^k - 1$ $(k = 1, 2, ...)$ となるように，それ以外のハッシュテーブルはリハッシュせずに挿入可能な要素数が 2.0×10^8 となるように，それぞれ制御されている．まず，**flat_hash_map** は load factor が 50 % を超えた時点でリハッシュしているとわかる．2.0×10^8 要素挿入時の経過時間は，概ね，**IpCHashT (uint8, maxLF50)**・**IpCHashT (uint8, maxLF100)**・**dense_hash_map** と，**IpCHashT (uint16, maxLF100)**・**flat_hash_map** と，**CHashT** と，**unordered_map** に別れる．挿入速度は，Fig. 4.6 のようなテーブルサイズに依存するため，一概には言えないが，**IpCHashT (uint16, maxLF100)** では load factor の増加に従い，挿入に時間が掛かっている．また，**CHashT** や，特に **unordered_map** は，堅調に時間が掛かっている．

Fig. 4.5 は，テーブルサイズを 0 で初期化した場合の挿入速度の累積時間である．Fig. 4.4 と比較して，**CHashT** と **IpCHashT (uint16, maxLF100)** の累積時間の増加が顕著である．CHashT では，そもそもの挿入が遅い分リハッシュにも時間が掛かっている．また，IpCHashT (uint16, maxLF100) では，load factor の増加に伴い，空き要素の線形探査に時間が掛かるためである．なお，他の条件の IpCHashT は，比較的 load factor の低い領域を使用するため，線形探査の影響は軽微である．**unordered_map** は，挿入とリハッシュの両方に時間が掛かっていることが伺える．これは，**CHashT** と同じ傾向である．この中で，**dense_hash_map** は最も高い性能を示した．計算量の観点からは，**IpCHashT (uint8, maxLF50)**，**IpCHashT (uint8, maxLF100)**，**flat_hash_map**，**dense_hash_map** は同程度のオーダーであると考えられる．

Fig. 4.6 は，テーブルサイズと挿入速度の関係である．キャッシュの変わり目を除き，**dense_hash_map** 及び **flat_hash_map** が広い区間で高い性能を示している．**IpCHashT (uint8, maxLF50)** は，これに次ぐ性能を示しており，複雑な挿入アルゴリズムの割に高速である．これは，結局のところ，巨視的には挿入操作が単なる線形探査のためである．しかしながら，**IpCHashT (uint8, maxLF100)** と **IpCHashT (uint16, maxLF100)** の示す通り，線形探査による空き領域の探査は load factor の高い領域において大きく効率を落としている．**CHashT** は L2，L3 キャッシュの外へ格納が増えるにしたがい，大きく速度を落としている．

探査速度

Fig. 4.7，4.8，4.9，4.10 に示す，探査速度とテーブルサイズの関係について考察する．

まず，successful search の速度，Fig. 4.7，4.8 について考察する．

Successful major option をコンパイル時に IpCHashT へ指定した速度は，Fig. 4.7 である．**IpCHashT (uint8, maxLF50)** は，L2 キャッシュから溢れるテーブルサイズ 10^5 前後を除き，区間 10^2〜10^7 の殆どに渡り最速の探査性能を示す．**IpCHashT (uint16, maxLF100)** は Fig. 4.2 に示すように dense_hash_map の 75 % のメモリ使用量であるにも関わらず，L3 キャッシュ内から CPU キャッシュ外の区間 1.5×10^5〜3.5×10^7 において，同程度の速度を保持している．ただし，L2 キャッシュ内の区間 1.0×10^1〜1.5×10^5 と非常にテーブルサイズの大きな区間 3.5×10^7〜2.0×10^8 においては dense_hash_map の方が高速に動作している．

Unsuccessful major option をコンパイル時に IpCHashT へ指定した速度は，Fig. 4.8 である．**IpCHashT (uint8, maxLF50)** は，L2 キャッシュから溢れるテーブルサイズ 10^5 前後を除き，区間 10^2〜10^7 の殆どに渡り 1~2 番目の探査速度を保持している．一方で，その他のオプションの IpCHashT は，優位性のある速度に達していない．

次に，unuccessful search の速度，Fig. 4.9，4.10 について考察する．

Successful major option をコンパイル時に IpCHashT へ指定した速度は，Fig. 4.9 である．**IpCHashT (uint8, maxLF50)** は，区間 1.0×10^2〜1.5×10^5 において高速に動作するが，区間 1.5×10^5〜1.5×10^7 においては **flat_hash_map** が最速である．区間 1.5×10^7〜2.0×10^8 においては，よりメモリ効率の高い **IpCHashT (uint8, maxLF100)** や **IpCHashT (uint16, maxLF100)** が最速である．なお，Fig. 1.1 (a) に示すように，Separate Chaining はアルゴリズム上 unsuccessful search に強く，一部区間では **CHashT** であっても dense_hash_map を上回る速度を示している．

Unsuccessful major option をコンパイル時に IpCHashT へ指定した速度は，Fig. 4.10 である．**IpCHashT (uint8, maxLF50)** は，特に L1, L2 キャッシュに収まる区間 1.0×10^2〜1.5×10^5 において，dense_hash_map の ~4 倍程度高速に動作している．また，区間 1.5×10^5〜1.5×10^7 でも ~3 倍程度高速に動作している．一方で，区間 1.5×10^7〜2.0×10^8 においては，よりメモリ効率の高い **IpCHashT (uint16, maxLF100)** が速度を伸ばした．Fig. 4.10 の結果は特に，Chaining 系のアルゴリズムが高速に動作する結果となった．

削除速度

Fig. 4.11，4.12 に示す，削除速度とテーブルサイズの関係について考察する．

まず，速度の違いについて，successful major option を指定した場合と，unsuccessful major option を指定した場合の違いについて考察する．このベンチマークでは，存在する key-value ペアを削除しているため，Fig. 4.11 に示す successful search を優先する設定が，Fig. 4.12 に示す unsuccessful search を優先する設定より高速に処理すると期待されたが，明白な違いは認められなかった．これは，単に計算量がオーダーで異なるため，違いが埋もれていると考えられる．

dense_hash_map は，区間 1.0×10^1〜2.0×10^8 のほぼ全てにおいて最速を保持している．**IpCHashT (uint8, maxLF50)** は，L2 キャッシュ内の区間 2.0×10^2〜1.5×10^5 において，flat_hash_map より高速に動作するものの，区間 1.5×10^2〜2.0×10^8 においては，**flat_hash_map** の方が高速に動作している．

第 6 章

結論

本投稿では，双方向リスト構造を内包する closed hashing アルゴリズムとして，In-place Chained Hash Table を提案した．Fig. 1.1 に示すように，chaining 系のアルゴリズムは，successful search に加え，特に unsuccessful search に対して高い性能を示すことが期待された．

まず，**IpCHashT (uint8, maxLF50)** の unsuccessful search major option について結論を述べる．

Successful search speed について，Fig. 4.8 より，テーブルサイズ 1.0×10^2〜1.0×10^7 において，L2 キャッシュを跨ぐ 1.0×10^5 前後を除き，殆どの区間で，1 番目ないし，2 番目の実行速度を示した．ただし，1.0×10^7〜2.0×10^8 の巨大なハッシュテーブについては，dense_hash_map がよい性能を示した．

Unsuccessful search speed については，Fig. 4.10 より，テーブルサイズ 1.0×10^2〜1.0×10^7 において，L2 キャッシュを跨ぐ 1.0×10^5 前後を除き，殆どの区間で，1 番目ないし，2 番目の実行速度を示した．ただし，1.0×10^7〜2.0×10^8 の巨大なハッシュテーブについては，最もメモリ効率の高い IpCHashT (as uint16 and maxLF100) が，最もよい性能を示した．

挿入速度に関しては，Fig. 4.6 に示す通り，必ずしも最速ではないものの，Fig. 4.5 より，通常の使用において，累積の挿入速度や，リハッシュ時間は極端に遅い訳ではないことを確認した．

削除速度に関しては，Fig. 4.12 に示す通り，2~3 番目の速度を示した．

IpCHashT (uint8, maxLF50) の successful search major option について結論を述べる．

Successful search speed について，Fig. 4.7 より，Fig. 4.8 に示す unsuccessful search major option の場合と比較して，テーブルサイズ 3.0×10^5〜1.0×10^7 において，性能の改善が見られる．それ以外の区間においては，大きな性能改善は見られず，1.0×10^7〜2.0×10^8 の巨大なハッシュテーブについては，同様に性能が悪化した．

Unsuccessful search speed については，Fig. 4.9 より，unsuccessful search major option の場合と比較して，L2 キャッシュサイズ内の 1.0×10^2〜1.0×10^5 において，大きく性能を落とした．また，1.0×10^5〜1.0×10^7 においては，flat_hash_map が最も高い性能を示した．

IpCHashT (uint16, maxLF100) の successful search major option について結論を述べる．

Successful search speed について，Fig. 4.7 より，区間 1.0×10^5〜3.5×10^7 において，dense_hash_map と同程度の性能を示した．区間 3.5×10^7〜2.0×10^8 においては，dense_hash_map には劣るものの，2 番目の性能に収まった．

Unsuccessful search speed については，Fig. 4.9 より，区間 1.0×10^5〜3.5×10^7 において，dense_hash_map と同程度の性能を示した．区間 3.5×10^7〜2.0×10^8 においては，最も速い性能を示した．

これらは，dense_hash_map の 75 % のメモリ使用量であることを鑑みれば，よい結果であるといえる．

Table 6.1. Comparison of insertion speed, search speed (successful search / unsuccessful search), deletion speed, and memory efficiency of each implementation. In particular, search speed is evaluated separately for the number of elements on the hash table $10^{2\sim5}$, $10^{5\sim7}$, and $10^{7\sim8.3}$. Measurement conditions are key and value: uint64, CPU: AMD Ryzen7 1700, and memory: DDR4-2666 32GB. “best” is a hash table that shows the highest performance of each item. In addition, “good”, “medium”, and “bad” are qualitative evaluations.

表 6.1 各実装の挿入速度と探査速度（成功する場合／失敗する場合），削除速度，メモリ効率の比較．特に探査速度はハッシュテーブル上の要素数が $10^{2\sim5}$，$10^{5\sim7}$，$10^{7\sim8.3}$ の場合について個別に評価する．測定条件は，key, value が uint64 で，CPU が AMD Ryzen7 1700，メモリが DDR4-2666 32GB である．best は各項目で最も高い性能を示したハッシュテーブルを示す．その他，good, medium, bad は定性的な評価である．

Implementation of hash table	Insert	Successful search			Unsuccessful search			Erase	Memory efficiency
		$10^{2\sim5}$	$10^{5\sim7}$	$10^{7\sim8.3}$	$10^{2\sim5}$	$10^{5\sim7}$	$10^{7\sim8.3}$		
IpCHashT (u8-LF50-S)	medium	best	best	bad	medium	medium	medium	medium	bad
IpCHashT (u8-LF50-U)	medium	best	medium	bad	best	best	medium	medium	bad
IpCHashT (u16-LF100-S)	bad	bad	good	good	bad	bad	good	bad	best
IpCHashT (u16-LF100-U)	bad	bad	bad	medium	good	good	best	bad	best
dense_hash_map	best	good	medium	best	bad	bad	medium	best	good
flat_hash_map	good	medium	medium	bad	bad	good	medium	medium	bad

Note: u8: uint8, u16: uint16, S: Successful search major option, U: Unsuccessful search major option.

ただし，挿入速度に関しては，Fig. 4.6 が示す通り load factor が高い場合に極端に速度が低下するため，Fig. 4.5 のようなリハッシュを伴う要素の挿入には，dense_hash_map の 2.1 倍程度[1] の時間が掛かる．もちろん，Fig. 4.4 のように，事前にハッシュテーブルが初期化されている場合，この差は小さくなり，load factor の比較的小さな領域を使用する場合には，差異は軽微である．

削除速度に関しては，Fig. 4.11 に示す通り，CHashT よりは速い，程度の速度を保っている．

以上の結果は，Table 6.1 のようにまとめられる．これらの結果から，少なくとも，AMD Ryzen7 1700 上では，key type と value type が uint64 のハッシュテーブルについて，次のことが言える．**IpCHashT (uint8, maxLF50, successful search major option)** は，successful search の区間 10^2〜10^7 で最も優れた結果を出す．**IpCHashT (uint8, maxLF50, unsuccessful search major option)** は，successful search の区間 10^2〜10^7 で優れた結果を，successful search の区間 10^2〜10^7 で最も優れた結果を出す．全体によい結果を出す区間が長い．一方で，メモリ効率は高くなく，10^7〜2.0×10^8 の巨大なハッシュテーブでは大きく性能を落とす．また，Fig. 2.5，Fig. 2.6 で示されるように，successful search major option よりもキーの比較回数が多くなるため，キーの比較コストが大きな型では，探査速度の低下が予測される．**IpCHashT (uint16, maxLF100, successful search major option)** は，最も高いメモリ効率を示し，successful search の区間 10^5〜$10^{8.3}$ において，安定した性能を示す．**IpCHashT (uint16, maxLF100, unsuccessful search major option)** は，最も高いメモリ効率を示し，unsuccessful search の区間 10^7〜$10^{8.3}$ で最も優れた結果を出す．

以上より，双方向リスト構造の in-place 実装による chaining のアルゴリズムの closed hashing 化について，Fig. 1.1 から期待されたような，高い探査性能を持つことを，ベンチマークにより実証した．特に unsuccessful search において見込まれていた，理論上は他のアルゴリズムより突出して高い探査性上も実証された．

[1] 2.0×10^8 要素を挿入するとき，$52.5[\mathrm{sec}]/25.0[\mathrm{sec}] = 2.1$ 倍の速度差が発生している．

付録

ここでは，Fig. A.1 に，CPU が各演算に対して消費する clock 数を示す.

A　CPU が各演算に消費する clock 数

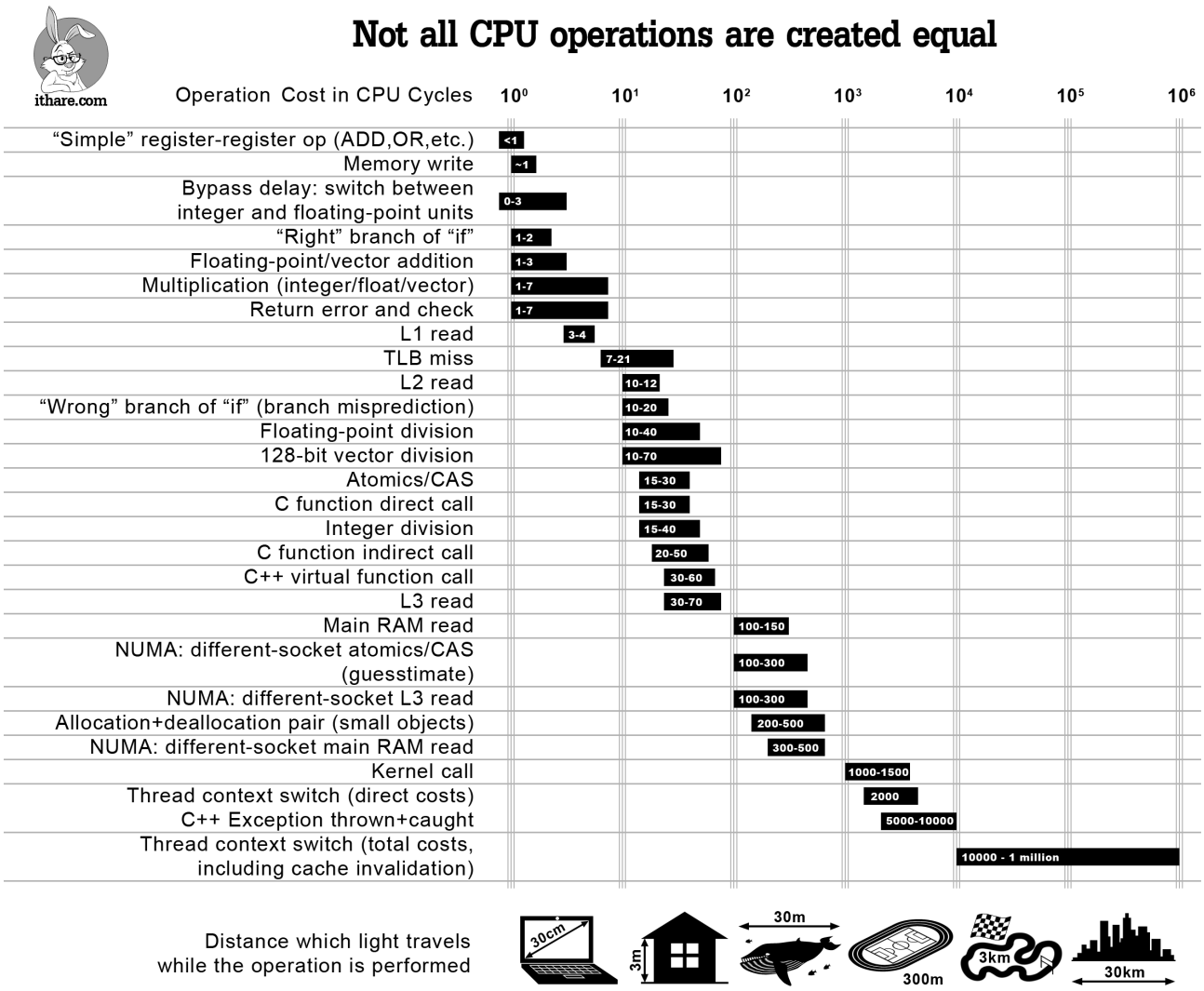

Figure A.1. Infographics: Operation Costs in CPU Clock Cycles ["No Bugs" Hare 2016].

参考文献

[1] ADMIS, W. 2017 「要素の挿入と削除による性能低下を抑えた高速ハッシュテーブルアルゴリズムの提案」, URL：https://admiswalker.blogspot.com/2017/01/ichasht.html.

[2] Fog, A. 2018 “Software optimization resources - 4. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs,” URL: https://www.agner.org/optimize/instruction_tables.pdf.

[3] google-sparsehash@googlegroups.com 2005 “sparsehash,” URL: http://goog-sparsehash.sourceforge.net/doc/dense_hash_map.html,https://github.com/sparsehash/sparsehash.

[4] Knuth, D. E. 1998 *The Art of Computer Programming Volume 3 - Sorting and Searching*, Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

[5] "No Bugs" Hare 2016 “Infographics: Operation Costs in CPU Clock Cycles,” URL: http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/.

[6] Skarupke, M. 2017 “I Wrote The Fastest Hashtable - Probably Dance,” URL: https://probablydance.com/2017/02/26/i-wrote-the-fastest-hashtable/,https://postd.cc/i-wrote-the-fastest-hashtable-1/.

[7] 石畑清 1989 『アルゴリズムとデータ構造』, 岩波講座ソフトウェア科学 / 長尾真 [ほか] 編, 第 3 号, 岩波書店, URL：http://ci.nii.ac.jp/ncid/BN0325221X.