算法数理工学
第7回

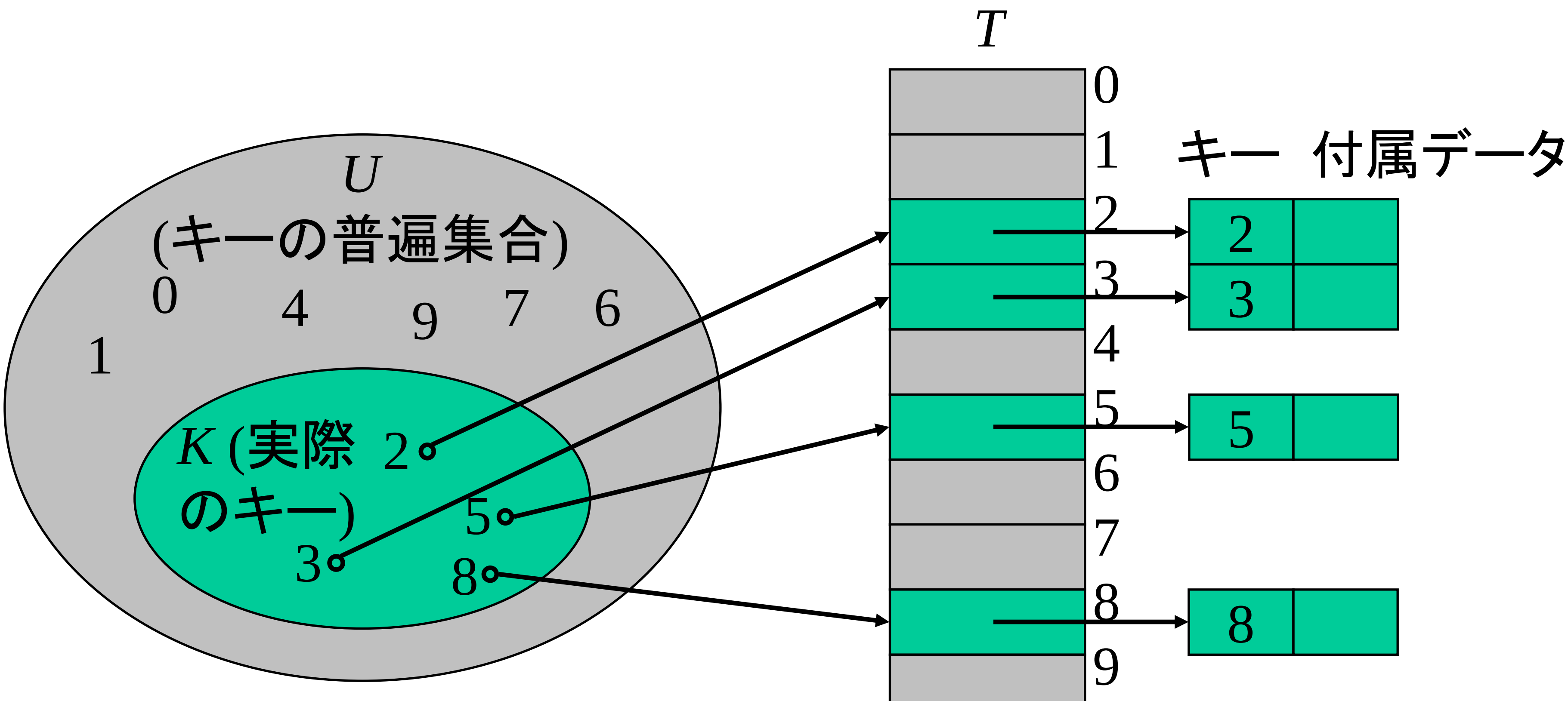
定兼　邦彦

ハッシュ表

- 辞書操作 (INSERT, DELETE, SEARCH) を効率よく実現するデータ構造
- 応用：C言語のコンパイラでの記号表の管理
  - キー：変数名などの文字列
- ハッシュ表は実際的な場面では極めて速い
  - 妥当な仮定の下で，SEARCHの時間の期待値は O(1)
  - 最悪の場合 $\Theta(n)$

直接アドレス表

- 出現する可能性のあるキーの全集合 (普遍集合, universal set) が大きくない場合にうまく働く
- キーが普遍集合 $U = \{0,1,\ldots, m-1\}$ から選択され，どの2つの要素も同じキーをもたないと仮定する
- 直接アドレス表 (direct-access table) T で動的集合を表現する
- 配列 $T[0..m-1]$ の各要素が U のキーに対応
- $T[k]$ は，キー k を持つ要素をさす．そのような要素がなければ $T[k]$ = NIL
- $T[k]$ をスロット k と呼ぶ


T
0
1
2
3
4
5
6
7
8
9
キー 付属データ
U
(キーの普遍集合)
0 4 9 7 6
1
K (実際のキー)
2
5
3
8
2
3
5
8

辞書操作の実現

- DIRECT_ADDRESS_SEARCH(T,k)
  - return $T[k]$
- DIRECT_ADDRESS_INSERT(T,x)
  - $T[\mathrm{key}(x)] = x$
- DIRECT_ADDRESS_DELETE(T,x)
  - $T[\mathrm{key}(x)] = \mathrm{NIL}$
- いずれも O(1) 時間
- T にオブジェクトそのものを格納してもいい

直接アドレス表の欠点

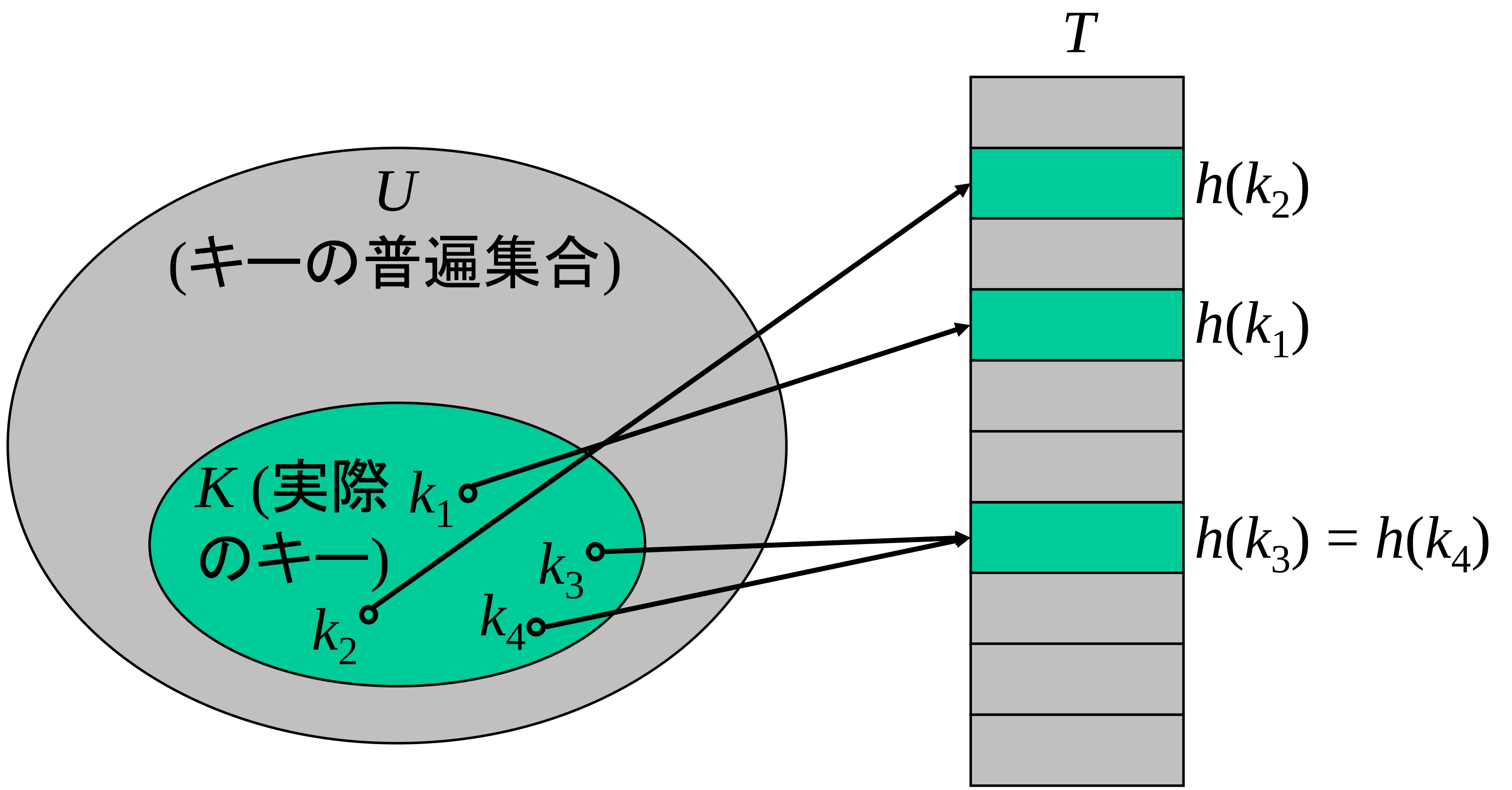
- キーの普遍集合 U が大きい場合は非現実的
  - 表 T をメモリに格納できない
  - T に割り付けた領域のほとんどが無駄になる
- 辞書に格納されているキーの集合 K が U に比べて十分に小さい場合はハッシュ表が有効

ハッシュ表

- 直接アドレス表では，キー k はスロット k に格納
- ハッシュ表 $T[0..m-1]$ では，スロット $h(k)$ に格納
- h: ハッシュ関数 (hash function)
  - $h: U \to \{0,1,\ldots,m-1\}$
- 必要な領域: $\Theta(|K|)$
- 要素の探索: O(1)　(平均時間)

ハッシュ関数の衝突

- 衝突 (collision): 2つのキーが同じスロットにハッシュされること



衝突の回避方法

- 別のハッシュ関数を用いる
  - $|U| > m$ なので完全に回避することは不可能
- チェイン法
- オープンアドレス法

チェイン法による衝突解決

- 同じスロットにハッシュされたすべての要素を連結リストに格納
- CHAINED_HASH_INSERT(T,x)
  - リスト $T[h(\mathrm{key}(x))]$ の先頭に x を挿入, O(1) 時間
- CHAINED_HASH_SEARCH(T,k)
  - リスト $T[h(k)]$ の中からキー k を持つ要素を探索
- CHAINED_HASH_DELETE(T,x)
  - リスト $T[h(\mathrm{key}(x))]$ から x を削除, 双方向リストを用いれば O(1) 時間

ハッシュ表に格納される構造体(英語と日本語のペア)

```
typedef struct dlobj_ {
  struct dlobj_*next;      // 後の要素へのポインタ
  struct dlobj_*prev;      // 前の要素へのポインタ
  char *eng;               // 英語文字列
  char *jpn;               // 日本語文字列
} dlobj;
```

ハッシュ関数の選び方

- h: ハッシュ関数 (hash function)
  - $h: U \to \{0,1,\ldots,m-1\}$
- $x \neq y \Rightarrow h(x) \neq h(x)$ になるのが理想だが，$|U| > m$ なら無理
- キーが文字列のとき，例えば

```
int hash_func(char *key) {
  int h = 0, i = 0;
  while (key[i] != 0) {
    h = h * 101 + key[i]; // ハッシュ値を計算
    i++;
  }
  return abs(h); // 非負の値にする
```

- この値を m (ハッシュ表のサイズ) で割った余りをハッシュ値とする

```
dlobj *hash_search(hash *H, char *key)
{
  int h;
  h = hash_func(key) % H->m;
  return dlist_search(H->T[h], key);
}

void hash_insert(hash *H, char *eng, char *jpn)
{
  dlobj *obj;
  int h;

  obj = dlobj_new(eng, jpn);

  h = hash_func(eng) % H->m;
  dlist_insert(H->T[h], obj);
  H->n++;
}
```

```
void hash_delete(hash *H, dlobj *obj)
{
  int h;
  h = hash_func(obj->eng) % H->m;
  dlist_delete(H->T[h], obj);
  H->n--;
}
```

チェイン法を用いるハッシュ法の解析

- SEARCHは最悪の場合 $\Theta(n)$ 時間
- 平均時間を解析する
- スロット m 個, n 要素を格納するハッシュ表 T の負荷率 (load factor) $\alpha = n/m$ と定義
- α は1つのチェインに格納される要素数の平均
- 解析は α を変数として行う (n, m が共に無限大に近づくとき，α はある定数に留まるとする)

ハッシュ法の平均的性能

- 各要素は m 個のスロットに同じ程度にハッシュされると仮定する (単純一様ハッシュの仮定 simple uniform hashing)
- ハッシュ値 $h(k)$ は O(1) 時間で計算できると仮定
- キー k をもつ要素の探索は，リスト $T[h(k)]$ の長さに比例した時間が必要

定理1 チェイン法を用いるハッシュ表で，単純一様ハッシュを仮定すると，失敗に終わる探索にかかる時間の平均は $\Theta(1+\alpha)$

定理2　チェイン法を用いるハッシュ表で，単純一様ハッシュを仮定すると，成功する探索にかかる時間の平均は $\Theta(1+\alpha)$

ハッシュ表中の要素数が $n = O(m)$ のとき
$\alpha = n/m = O(m)/m = O(1)$
つまり，すべての辞書操作が平均 $O(1)$ 時間

定理1 チェイン法を用いるハッシュ表で，単純一様ハッシュを仮定すると，失敗に終わる探索にかかる時間の平均は $\Theta(1+\alpha)$

証明　単純一様ハッシュを仮定すると，任意のキーは m 個の各スロットに同程度にハッシュされる．
あるキーの探索が失敗するとき，その時間の平均は，1つのリストを最後まで探索する時間の平均．
リストの長さの平均は負荷率 $\alpha = n/m$.
よって検査される要素数の期待値は α,
時間は $\Theta(1+\alpha)$

定理2　チェイン法を用いるハッシュ表で，単純一様ハッシュを仮定すると，成功する探索にかかる時間の平均は
$\Theta(1+\alpha)$

証明　INSERTにおいて，新しい要素はリストの末尾に挿入されると仮定する.

成功する探索で検査される要素数の期待値は，見つかった要素が挿入されたときに検査された要素数+1.

表に格納されている n 個の要素について平均を取る.

i 番目の要素が付け加えられるときのリストの長さの期待値は $(i-1)/m$

$$\frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{i-1}{m}\right)=1+\frac{1}{nm}\sum_{i=1}^{n}(i-1)$$
$$=1+\left(\frac{1}{nm}\right)\left(\frac{(n-1)n}{2}\right)$$
$$=1+\frac{\alpha}{2}-\frac{1}{2m}$$

成功する探索に必要な時間は $\Theta(1+\alpha)$

ハッシュ関数

- 良いハッシュの条件 = 単純一様ハッシュ
- 各キーは U から確率分布 P に従って独立に取り出されると仮定すると，条件は

$$\sum_{k:h(k)=j} P(k) = \frac{1}{m} \qquad (j = 0,1,\dots,m-1)$$

と書ける

- ただし，一般に P は未知
- キーがランダムな実数k $(0 \le k < 1)$で一様独立のとき，$h(k) = \lfloor km \rfloor$ は上の条件を満たす

除算法

- キー k を m で割った剰余をハッシュ値とする
  - $h(k) = k \bmod m$
- 利点: ハッシュ関数の計算が高速
- m は 2 のべき乗に近くない素数がいい
  - $m = 2^p$ のとき，$h(k)$ は k の最下位 p ビット
  - $m = 2^p - 1$ で k が基数 2^p の文字列のとき，文字を並び替えてもハッシュ値は同じ

例

- $n = 2000$ 個の文字列を格納する場合
- 負荷率 α を 3 に近くするには, $m = 701$ にすればいい
- 701 は素数で, 2のべき乗には近くない
- $h(k) = k \bmod 701$ とすればいい
- このハッシュ関数が実際のデータでうまく働くことを確かめるべき

乗算法

- まず，キー k にある定数 A $(0 < A < 1)$ を掛け，その小数部分 $kA - \lfloor kA \rfloor$ を取り出す
- 次に，その値に m を掛け，小数部分を切り捨てる

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

- m の値はあまり重要ではない
  – 2のべき乗にすると計算が簡単
- $A \approx \left(\sqrt{5} - 1\right)/2 = 0.6180339887 \ldots$ が良いと言われる

オープンアドレス指定法

- オープンアドレス指定法 (open addressing)では，要素は連結リストではなくハッシュ表の中に格納される．
- ハッシュ表が埋まるとそれ以上挿入できない
  - 負荷率は1以下
- 連結リストを用いないため，スペースが小さい
- ハッシュ関数を拡張して衝突を回避する
  - 引数: キーと探査番号
  - $h: U \times \{0,1,\ldots,m-1\} \to \{0,1,\ldots,m-1\}$
- 探査列 $\langle h(k,0), h(k,1),\ldots, h(k,m-1)\rangle$ は $\langle 0,1,\ldots,m-1\rangle$ の順列でなければならない

要素の挿入

```
int HASH_INSERT(data *T, data k)
{
    int i, j;
    i = 0;                              //  i: 探査番号
    do {
        j = h(k,i);
        if (T[j] == NIL) {         //  スロットが空なら
            T[j] = k;              //   新しいデータを挿入
            return j;
        } else {
            i++;                   //   探査番号を1増やす
        }
    } while (i != m);          //   m 回試す
    printf(“ハッシュ表オーバーフロー¥n”);
}
```

要素の検索

```
int HASH_SEARCH(data *T, data k)
{
    int i, j;
    i = 0;                                  // i: 探査番号
    do {
        j = h(k,i);
        if (T[j] = k) return j;             // k を発見
        i++;
    } while (T[j] != NIL && i != m);  //  スロットが空か，m 回探索した
    return NIL;                             // k は見つからなかった
}
```

要素の削除

- 削除したい要素のあるスロットを NIL にすると，他の要素が検索できなくなる
  - NIL スロットが見つかると検索は終了するため
- 削除するときは NIL でなく特別な値 DELETED を格納する
- SEARCHではDELETEDが現れても探索を続ける
- INSERTではNILまたはDELETEDの場所に挿入
- 問題点: 探索時間が負荷率 α で表せない
- 要素を削除する必要がある場合はチェイン法が好まれる

衝突回避の方法

- 一様ハッシュ (uniform hashing) を仮定: 各キーに対する探査列として，$\{0,1,...,m-1\}$ の $m!$ 通りの順列のどれもが同程度に現れる
- 近似的な一様ハッシュを用いる
  - 線形探査
  - 2次関数探査
  - ダブルハッシュ法

線形探査

- 通常のハッシュ関数 $h': U \to \{0,1,...,m-1\}$ に対し $h(k,i) = (h'(k)+i) \bmod m$ $(i=0,1,...,m-1)$ を用いる
- 探査されるスロット: $T[h'(k)], T[h'(k)+1],..., T[m-1], T[0], T[1],..., T[h'(k)-1]$
- 異なる探査列は m 通りしかない (開始位置で決定)
- 問題点: 主クラスタ化 (primary clustering) が起きる
- 直前の i 個のスロットが使用中である空きスロットが選択される確率は $(i+1)/m$ であるため，連続する使用中のスロットは常に大きくなる

2次関数探査

- 2次関数探査 (quadratic probing) では $h(k,i) = (h'(k)+c_1 i+c_2 i^2) \bmod m$ (i=0,1,...,m–1) を用いる
- c_1, c_2, m は適切に選ぶ必要がある
- $h(k_1,0) = h(k_2,0)$ の場合は $h(k_1,i) = h(k_2,i)$ となってしまう (副クラスタ化, secondary clustering)
- 異なる探査列は m 通りしかない (開始位置で決定)

ダブルハッシュ法

- $h(k,i) = (h_1(k)+i\ h_2(k)) \bmod m$
- 探査列は，初期位置と次に探査する位置までの距離の両方が k に依存している
- $h_2(k)$ の値は m と互いに素である必要がある
  - $h_2(k)$ と m の最大公約数が d のとき，ハッシュ表の $1/d$ しか検査しない
- 例1: m を2のべき乗，h_2 は常に奇数
- 例2: m を素数， h_2 は m 未満の非負整数 $h_2(k)=1+(k \bmod m')$
- $\Theta(m^2)$ 個の探査列が利用できる

オープンアドレスハッシュ法の解析

- ハッシュ表の負荷率 α をパラメータとして解析
- 一様ハッシュを用いると仮定する

定理 負荷率 $\alpha = n/m < 1$ のオープンアドレスハッシュ表において，失敗に終わる探索に必要な探査数の期待値は $1/(1-\alpha)$ 以下

αが定数なら O(1) 時間で実行できる

$\alpha = 0.5$ なら 2回 以下

$\alpha = 0.9$ なら 10回 以下

証明 失敗に終わる探索では，
毎回の探査では異なるキーを格納しているスロットにアクセスし，最後に未使用のスロットにアクセスする．

$i = 0,1,\ldots$ に対して，

$p_i = \Pr\{$未使用のスロットを見つける前にちょうど i 回の探査を行った$\}$

$q_i = \Pr\{$未使用のスロットを見つける前に少なくとも i 回の探査を行った$\}$

と定義する．探査回数の期待値は

$$1 + \sum_{i=0}^{\infty} i p_i = 1 + \sum_{i=1}^{\infty} q_i$$

最初の探査が使用中のスロットにアクセスする確率は n/m であるから

$$q_1 = \frac{n}{m}$$

一様ハッシュ法では，2回目の探査は残りの m–1 個のスロットの1つに対して行われ，その中には n–1 個の使用中のスロットがあるため

$$q_2 = \frac{n}{m} \cdot \frac{n-1}{m-1}$$

一般に $$q_i = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \cdots \cdot \frac{n-i+1}{m-i+1} \le \left(\frac{n}{m}\right)^i = \alpha^i$$

失敗に終わる探索に必要な探査回数の期待値は

$$
\begin{aligned}
1+\sum_{i=0}^{\infty} i p_i &= 1+\sum_{i=1}^{\infty} q_i \\
&\leq 1+\alpha+\alpha^2+\alpha^3+\cdots \\
&= \frac{1}{1-\alpha}
\end{aligned}
$$

系　一様ハッシュを仮定すると，負荷率αのオープンアドレスハッシュ表に，ある要素を挿入するために必要な探査回数の平均は$1/(1-\alpha)$ 以下

証明　キーを挿入するには未使用スロットを発見する必要がある．その探査回数の期待値は失敗に終わる探索での探査回数の期待値に等しい．

定理 一様ハッシュを仮定し，表内の各キーは等確率で探索の対象になるとする．負荷率 α のオープンアドレスハッシュ表において，成功に至る探索に必要な探査回数の期待値は

$$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}+\frac{1}{\alpha}$$

α=0.5 のとき 3.387回 以下

α=0.9 のとき 3.670回 以下

証明　キー k の探索は，それを挿入したときと同じ探査列を探査する．系より，k がハッシュ表に $i+1$ 番目に挿入されたキーならば，探索に必要な探査回数の期待値は $1/(1-i/m) = m/(m-i)$以下

ハッシュ表に存在する n 個のキーについて平均を取ると，成功に至る探索に必要な探査回数の平均が得られる．

$$
\begin{aligned}
\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} &= \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \\
&= \frac{1}{\alpha}(H_m - H_{m-n}) \\
&\leq \frac{1}{\alpha}(\ln m + 1 - \ln(m-n)) \\
&= \frac{1}{\alpha}\ln\frac{1}{1-\alpha} + \frac{1}{\alpha}
\end{aligned}
$$

万能ハッシュ法

- 運が悪いと，n 個のキーが同じスロットにハッシュされ，平均検索時間が $\Theta(n)$ になってしまう
- 万能ハッシュ法 (universal hashing) では，ハッシュ関数をランダムに選択する
- どのように意地悪くキーを選択しても，平均として良い性能を示す．

- H: キーの普遍集合 U から値域 $\{0,1,\ldots,m-1\}$へのハッシュ関数の有限集合
- H が万能 (universal) ⇔ 全ての異なるキーの組 $x, y \in U$ に対し, $h(x) = h(y)$ となるハッシュ関数 $h \in H$ の個数が $|H|/m$ 以下
- ハッシュ関数を万能な H の中からランダムに選んだときに, x と y が衝突する確率が $1/m$ 以下
- これは $h(x)$ と $h(y)$ が値域 $\{0,1,\ldots,m-1\}$ からランダムに選択されたときの衝突確率に等しい

定理 h を万能な集合から選択されたハッシュ関数とする. h を用いて n 個のキーをサイズが m のハッシュ表にハッシュする. 衝突はチェイン法で解消する. このとき, キー k のハッシュ先のリストの長さの期待値 $\mathrm{E}[n_{h(k)}]$ は

高々α(キーが表に存在しないとき)

高々$1+\alpha$(キーが表に存在するとき)

期待値の計算は全てハッシュ関数に関して行い,

キーの分布については何も仮定しないことに注意

証明:異なるキーのペア k, l に対して，指標確率変数 $X_{kl} = \begin{cases} 1 & (h(k) = h(l)) \\ 0 & (h(k) \neq h(l)) \end{cases}$ を定義する.

ハッシュ関数の定義より，1つのキーのペアが衝突を起こす確率は高々 $1/m$. つまり $\Pr\{h(k)=h(l)\} \leq 1/m$

よって $\mathrm{E}[X_{kl}] \leq 1/m$.

キー k に対し，k 以外で k と同じスロットにハッシュされるキーの個数を確率変数 Y_k で表す.

$$Y_k = \sum_{\substack{l \in T \\ l \neq k}} X_{kl}$$

$$\mathrm{E}[Y_k] = \mathrm{E}\left[\sum_{\substack{l \in T \\ l \neq k}} X_{kl}\right] = \sum_{\substack{l \in T \\ l \neq k}} \mathrm{E}[X_{kl}] \leq \sum_{\substack{l \in T \\ l \neq k}} \frac{1}{m}$$

- $k \notin T$ のとき $n_{h(k)} = Y_k$ かつ $|\{l : l \in T \text{ and } l \neq k\}| = n$
  従って $$\mathrm{E}[n_{h(k)}] = \mathrm{E}[Y_k] \leq \frac{n}{m} = \alpha$$
- $k \in T$ のとき, キー k はリスト $T[h(k)]$ に存在し, カウント Y_k には k は含まれていないので
  $$n_{h(k)} = Y_k + 1 \quad \text{かつ} \quad |\{l : l \in T \text{ and } l \neq k\}| = n - 1$$
  従って $$\mathrm{E}[n_{h(k)}] = \mathrm{E}[Y_k + 1] \leq \frac{n-1}{m} + 1 < 1 + \alpha$$

万能ハッシュ関数族の設計

- どんなキー k も 0 から p-1 までの範囲に入るような十分大きな素数 p を選ぶ. $p > m$ を仮定.
- $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$ と定義
  ただし $a \in \{1,2,\ldots,p\text{-}1\}$, $b \in \{0,1,,\ldots,p\text{-}1\}$.
  m は素数でなくてもいい.
- 定理: ハッシュ関数のクラス
  $H_{p,m} = \{h_{a,b}: a \in Z_p^*, b \in Z_p\}$ は万能である.
- 証明: 相異なるキー $k, l \in Z_p$ を考える. ハッシュ関数 $h_{a,b}$ に対し
  $r = (ak+b) \bmod p$
  $s = (al+b) \bmod p$ と置く.

- 命題：$r \neq s$

  $r-s \equiv a(k-l) \pmod p$ である．

  a も $k-l$ も法 p の下で 0 ではない．

  p は素数だから右辺の積も 0 ではない．

- (k,l) を固定する． $p(p-1)$ 個存在するハッシュ関数のパラメタのペア (a,b) は，(k,l) を異なるペア (r,s) に写像する．

- $r \neq s$ であるペアは $p(p-1)$ 個存在するので，(a,b) と (r,s) には1対1対応がある．

$$a = \left((r-s)((k-l)^{-1} \bmod p)\right) \bmod p$$

$$b = (r-ak) \bmod p$$

- (a,b) を一様ランダムに選べば，(r,s) も一様ランダム．

- 従って，相異なるキー k と l が衝突する確率は，法 p の下で相異なる r と s をランダムに選択したときに $r \equiv s \pmod{m}$ となる確率に等しい．
- r を固定すると，r 以外の p–1 個の値の中で $r \equiv s \pmod{m}$ となる s の個数は高々

$$\left\lceil \frac{p}{m} \right\rceil - 1 \leq \frac{p+m-1}{m} - 1 = \frac{p-1}{m}$$

- よって，s と r が衝突する確率は高々 $1/m$
- 従って，任意の異なる値 $k,l \in Z_p$ のペアに対し

$$\Pr\{h_{a,b}(k) = h_{a,b}(l)\} \leq \frac{1}{m}$$

つまり $H_{p,m} = \{h_{a,b}: a \in Z_p^*, b \in Z_p\}$ は万能