2020年度前期・高度データ構造

本日は cuckoo hash と呼ばれるデータ構造を扱います。

ハッシュ

- $U = \{1, 2, 3, ..., u\}$ とする.
- $S \subseteq U$ とし, $|S| = n$ とする.
- 任意の $x \in U$ について,
  member(x, S), insert(x, S), delete(x, S)を
  効率よく処理するためのデータ構造をハッシュという.
  - 他のクエリ（min, max, predecessor, successor）は
    必ずしもサポートしなくてよい.

まず、ハッシュについて概説します。

いつものように、全体集合を U = {1, 2, 3, …, u} とし、その部分集合 S を考えます。

U の任意の要素 x について、member(x, S), insert(x, S), delete(x, S) を効率よく処理するためのデータ構造をハッシュと呼びます。

他のクエリは必ずしもサポートする必要はありません。

本日の流れ

- まず，基本的なハッシュ法の一つである チェイン法について概説する.
- チェイン法を用いた場合の member と insert/delete の平均的な時間計算量を解析する.
- この解析と同様の考え方で，cuckoo hash の性能を見積もる.

本日の講義の流れです。

まず、基本的なハッシュ法の一つであるチェイン法について概説します。

続いて、チェイン法を用いた場合の member と insert/delete の平均的な時間計算量を解析します。

この解析と同様の考え方で，cuckoo hash の性能を見積もっていきます。

チェイン法

- 集合 $S \subseteq U$ のサイズを n とする.
- 集合 S の要素を，サイズ r の配列 T（ハッシュ表）に格納する. ただし，$n \leq r < u$ とする.
- このとき，ハッシュ関数 $\mathrm{h} : U \to \{1, 2, ..., r\}$ を用いて，T の $\mathrm{h}(x)$ 番目の位置に x を格納する.
すなわち，$\mathrm{T}[\mathrm{h}(x)] = x$ とする.

チェイン法を説明します。

n を集合 S の要素数とします。
集合 S の要素を、サイズ r の配列 T に格納していきます。この配列をハッシュ表と呼びます。
ここで、n ≤ r < u とします。

このとき、U から {1, 2, ..., r} への関数 h を用いて、T の h(x) 番目の位置に x を格納します。
つまり、T[h(x)] = x とします。この関数 h をハッシュ関数と呼びます。

（次ページに続く）

チェイン法

- 集合 $S \subseteq U$ のサイズを n とする.
- 集合 S の要素を，サイズ r の配列 T（ハッシュ表）に格納する. ただし，$n \leq r < u$ とする.
- このとき，ハッシュ関数 $\mathrm{h} : U \to \{1, 2, ..., r\}$ を用いて，T の $\mathrm{h}(x)$ 番目の位置に x を格納する.
  すなわち，$\mathrm{T}[\mathrm{h}(x)] = x$ とする.
- 異なる $x, y \in U$ について，$\mathrm{h}(x) = \mathrm{h}(y)$ となるとき，x と y は衝突するという.
  $r < u$ より，衝突する要素が必ずある（鳩ノ巣原理）.
- そこで，衝突した x と y を同じ連結リストに格納し，$\mathrm{T}[\mathrm{h}(x)]$ にはその連結リストへのポインタを格納する.

異なる U の要素 x と y に対して、h(x) = h(y) となるとき，x と y は衝突するといいます。

r < u なので，鳩ノ巣原理より、衝突する要素が必ず存在します。

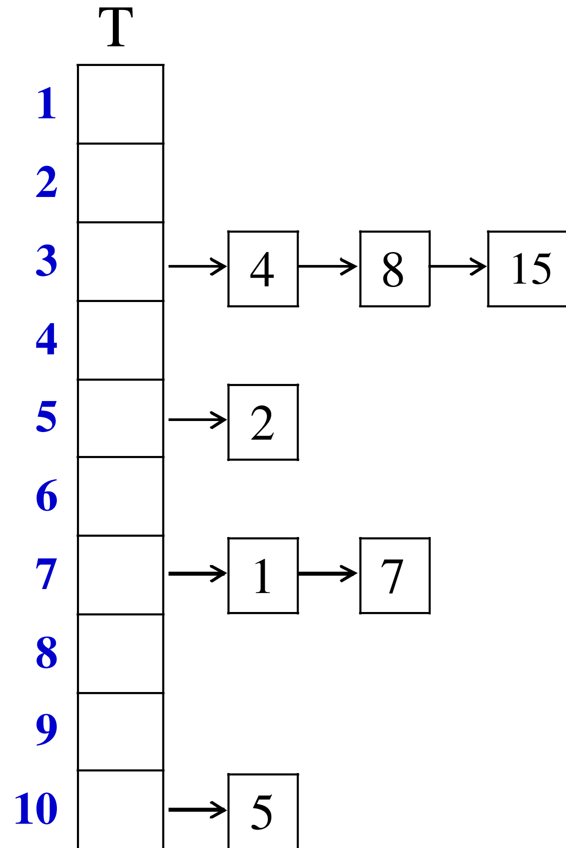
そこで、衝突した x と y を同じ連結リストに格納し、T[h(x)] にはその連結リストへのポインタを格納します。

これがチェイン法によるハッシュです。

次ページで具体例を示します。

チェイン法

- $S = \{1, 2, 4, 5, 7, 8, 15\}$
- $|T| = 10$
- h(1) = **7**,
  h(2) = **5**,
  h(4) = **3**,
  h(5) = **10**,
  h(7) = **7**,
  h(8) = **3**,
  h(15) = **3**



集合 S = {1, 2, 4, 5, 7, 8, 15} を考え、S 中の要素をサイズ 10 のハッシュ表 T に格納していきます。

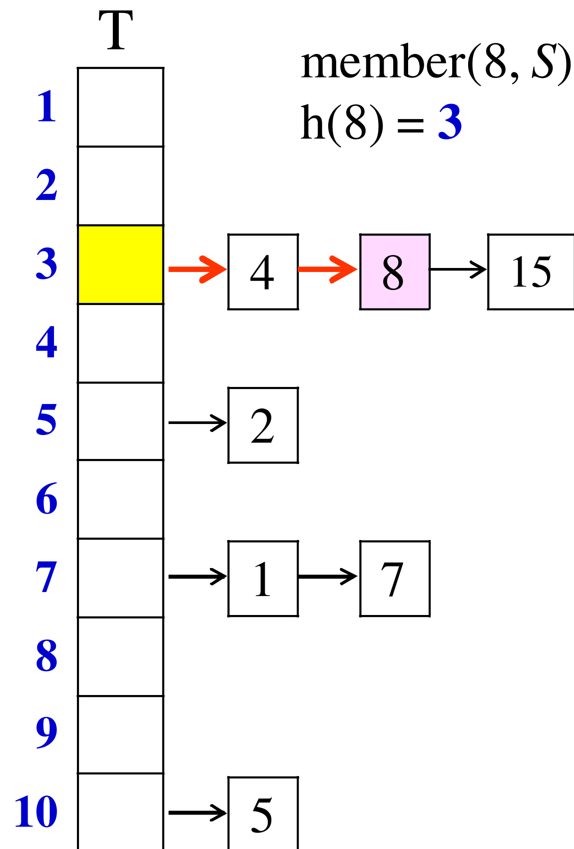
ハッシュ関数 h がこのように定義されているとしましょう。

このとき、例えば h(4) = h(8) = h(15) = 3 なので、S の要素 4, 8, 15 は衝突しています。これらの要素は、連結リストに格納され、T[3] はこの連結リストの先頭へのポインタが格納されています。

チェイン法による member

member(x, S)

1. h(x) を計算し, T[h(x)] にアクセスする.
2. T[h(x)] の連結リストを先頭から走査し, x を探す. x があれば **yes**, なければ **no** を返す.



チェイン法による member クエリは以下のようになります。

1. まず h(x) を計算し、T[h(x)] にアクセスします。
2. 次に、T[h(x)] の連結リストを先頭から走査し, x があれば yes, なければ no を返します。

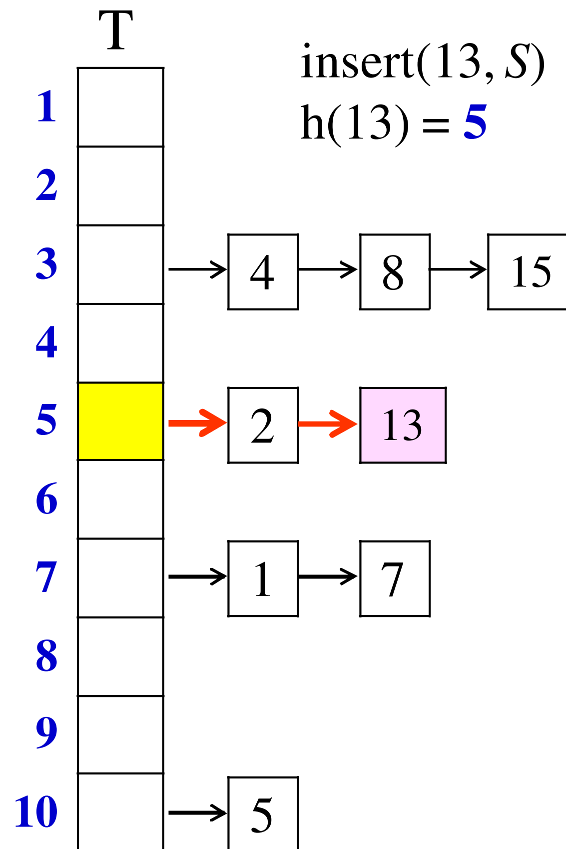
右の図の例では、member(8, S) を実行しています。

h(8) = 3 なので、T[3] にアクセスし、先頭からリストを探索して、8 が見つかりました。

チェイン法による insert

insert(x, S)

1. member(x, S) = **yes** のとき，終了．
2. member(x, S) = **no** のとき，T[h(x)] の連結リストの最後尾に x を追加する．



チェイン法による insert 操作は以下のようになります。

1. member クエリを実行し、答えが yes であればすでのその要素は T 中にあるので、終了します。
2. member クエリの答えが no であれば、T[h(x)] の連結リストの最後尾に x を追加します。

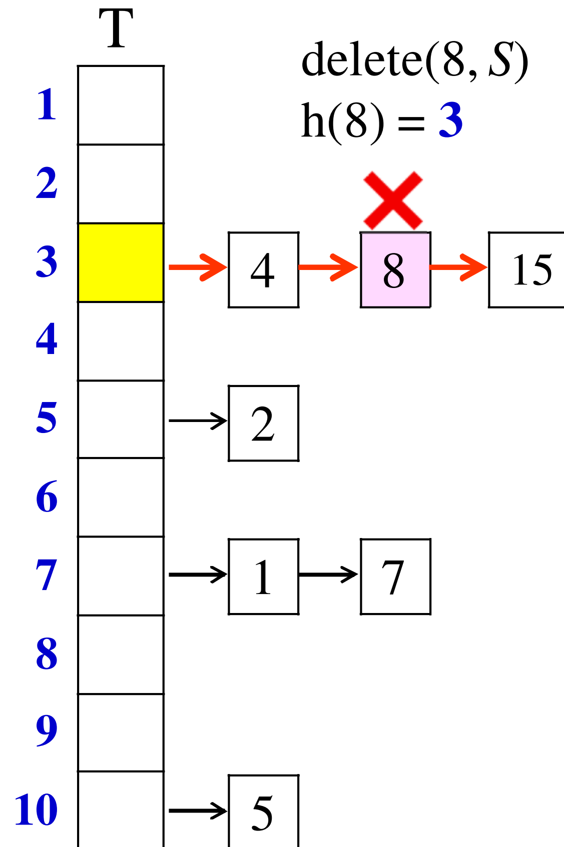
右の図の例では、13 を insert しようとしています。

h(13) = 5 で、member クエリの答えは no　です。
よって、T[5] の連結リストの末尾に 13 を追加します。

チェイン法による delete

delete(x, S)

1. member(x, S) = **yes** のとき，T[h(x)] の連結リストから x を削除．
2. member(x, S) = **no** のとき，終了．



delete(8, S)
h(8) = **3**

チェイン法による delete 操作は以下のようになります。

1. member クエリを実行し、答えが yes であれば、T[h(x)] の連結リストから x を削除します。
2. member クエリの答えが no であれば、x はそもそも T 中にないので、終了します。

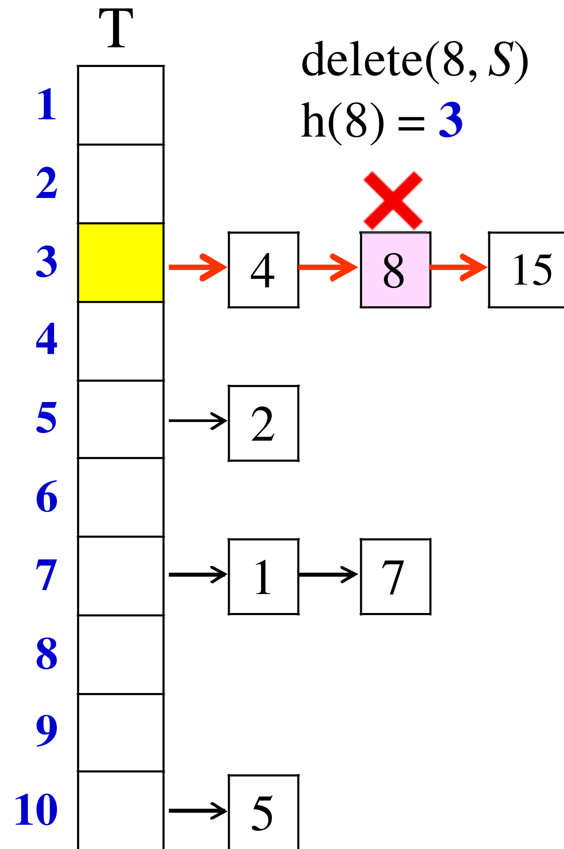
右の図の例では、8 を削除しようとしています。
h(8) = 3 で、member クエリの答えは yes です。

よって、T[h(x)] の連結リストから 8 を削除します。

チェイン法による delete

delete(x, S)

1. member(x, S) = **yes** のとき，T[h(x)] の連結リストから x を削除.
2. member(x, S) = **no** のとき，終了.



insert /delete の時間計算量は，member の時間計算量＋定数時間．よって，以降は member の時間計算量を考える.

これまでの例を見てもわかるように、insert および delete の時間計算量は、member の時間計算量＋定数時間です。

よって、以降は、member の時間計算量のみを考えていきます。

万能ハッシュ関数

- ハッシュ関数 h のハッシュ値の分布が偏っていると, member に時間がかかってしまう.
  - 例）すべての $x \in U$ について, $\mathrm{h}(x) = 1$ の場合, member の計算に $O(n)$ 時間かかる.
  - ハッシュ値はなるべく「バラけて」ほしい.

これまでの例を見てわかるように、ハッシュ関数 h のハッシュ値の分布に偏りがあると、member の計算に時間がかかってしまいます。

例えば、U のすべての要素 x について h(x) = 1 であるような極端な場合には、member に要する時間は O(n) になってしまいます。

よって、ハッシュ値はなるべくバラけて欲しいわけです。

万能ハッシュ関数

- ハッシュ関数 h のハッシュ値の分布が偏っていると, member に時間がかかってしまう.
  - 例）すべての $x \in U$ について, $\mathrm{h}(x) = 1$ の場合, member の計算に $O(n)$ 時間かかる.
  - ハッシュ値はなるべく「バラけて」ほしい.
- $\mathrm{h} : U \to \{1, 2, \ldots, r\}$ が以下の条件を満たすとき, h を**万能ハッシュ関数**（universal hash function）という.
  1. 任意の異なる $x, y \in U$ について, $\mathrm{h}(x) = \mathrm{h}(y)$ となる確率が $O(1/r)$.
  2. 任意の $x \in U$ について, ハッシュ値 $\mathrm{h}(x)$ を $O(1)$ 時間で計算できる.

そこで、万能ハッシュ関数というものを導入します。

関数 h が以下の条件を満たすとき、h を万能ハッシュ関数といいます。

1. U の任意の異なる要素 x, y について、h(x) = h(y) となる確率は O(1/r) である。
2. U の任意の要素 x について、そのハッシュ値 h(x) を定数時間で計算できる。

以降、万能ハッシュ関数が存在すると仮定（詳細は口述）して、議論を進めていきます。

万能ハッシュ関数を用いたチェイン法

- member(x, S) の時間計算量の期待値を求める.
- h を万能ハッシュ関数と仮定する.
- 各 $y \in S$-{x} について，h(x) = h(y) となる確率は $O(1/r)$.
- よって，member の時間計算量の期待値は

$$\sum_{y \in S - \{x\}} O(1/r) = (n-1) \cdot O(1/r) = O(n/r) = O(1)$$

$$(\because n \leq r)$$

万能ハッシュ関数を用いた場合のチェイン法について、member の時間計算量の期待値を求めます。

万能ハッシュ関数の仮定から、x 以外の S の任意の要素 y について、h(x) = h(y) となる確率、すなわち x と y が衝突する確率は O(1/r) です。

よって、member の時間計算量の期待値は x 以外のすべての S の要素について 確率 O(1/r) を足した値となり、結果として O(1) になります。

ここで、n ≤ r であることを思い出しましょう。

member の最悪時間計算量の改善

- 万能ハッシュ関数に基づくチェイン法では，member, insert/delete の時間計算量の期待値は $O(1)$.
- しかしながら，最悪時間計算量はどれも $O(n)$ ...
- member と delete の最悪時間計算量が $O(1)$ で, insert の時間計算量の期待値が $O(1)$ となるように改良したのが，これから紹介する **Cuckoo hash**.

というわけで、万能ハッシュ関数に基づくチェイン法では、member, および insert / delete の時間計算量の期待値は O(1) となります。

これでもすでに良い性能なのですが、しかしながら、最悪時間計算量はいずれも O(n) です。

そこで、member と delete の最悪時間計算量が O(1) で, insert の時間計算量の期待値が O(1) となるように改良したのが，これから紹介する Cuckoo hash です.

Cuckoo hash

- 2つのハッシュ表 T_1, T_2 と，2つの万能ハッシュ関数 h_1, h_2 を使用する.
- ハッシュ表の各位置には高々1つの要素しか格納されない.
- 任意の $x \in S$ は，$T_1[h_1(x)]$ または $T_2[h_2(x)]$ のいずれかに格納される.

cuckoo hash では、2つのハッシュ表 T_1, T_2 と，2つの万能ハッシュ関数 h_1, h_2 を使用します.

チェイン法とは異なり、cuckoo hash では、ハッシュ表の各位置には高々1つの要素しか格納されません。

また、S の任意の要素 x は、T_1[h_1(x)] または T_2[h_2(x)] のいずれかに格納されます。

次ページに、具体例を示します。

member with Cuckoo hash

	T_1		T_2	
1	4		1	1
2				2
3	8		7	3
4			15	4
5				5
6	2		5	6
7				7

member(x, S)
  if $T_1[h_1(x)] = x$ or $T_2[h_2(x)] = x$ then return **yes**
  else return **no**

この図は、S = {1, 2, 4, 5, 7, 8, 15} を2つのハッシュ表 T_1 と T_2 からなる cuckoo hash に格納した状況を示しています。

member クエリは、このシンプルな方法で実現可能です。

つまり、x を探すときには、T_1 の h_1(x) 番目と T_2 の h_2(x) 番目をチェックして、どちらかにあれば yes, どちらにもなければ no を返せばよい、ということになります。

member with Cuckoo hash

	T_1		T_2	
1	4		1	1
2				2
3	8		7	3
4			15	4
5				5
6	2		5	6
7				7

member(8, S)

$h_1(8) = 3$
$h_2(8) = 6$

member(x, S)
  if $T_1[h_1(x)] = x$ or $T_2[h_2(x)] = x$ then return **yes**
  else return **no**

例として、member(8, S) を実行してみましょう。

まず、8 のハッシュ値を計算します。その値が
h_1(8) = 3
h_2(8) = 6
だったとしましょう。

T_1 の3番目に8 が格納されているので、member(8, S) の答えは yes となります。

member with Cuckoo hash

定理 1

Cuckoo hash による member(x, S) の最悪時間計算量は $O(1)$ である.

- 高々2つの配列の要素を参照するだけだから.

member(x, S)
  if $T_1[h_1(x)] = x$ or $T_2[h_2(x)] = x$ then return **yes**
  else return **no**

次の定理が自明に成り立ちます。

cuckoo hash による member(x, S) の最悪時間計算量は O(1) です。

member のアルゴリズムでは、T_1 の h_1(x) 番目と、T_2 の h_2(x) 番目をチェックするだけです。
h_1 と h_2 はそれぞれ万能ハッシュ関数なので、h_1(x) と h_2(x) はそれぞれ O(1) 時間で計算できます。

したがって、member クエリに要する時間は O(1) です。

delete from Cuckoo hash

	T_1	T_2	
1	4	1	1
2			2
3	8	7	3
4		15	4
5			5
6	2	5	6
7			7

delete(x, S)

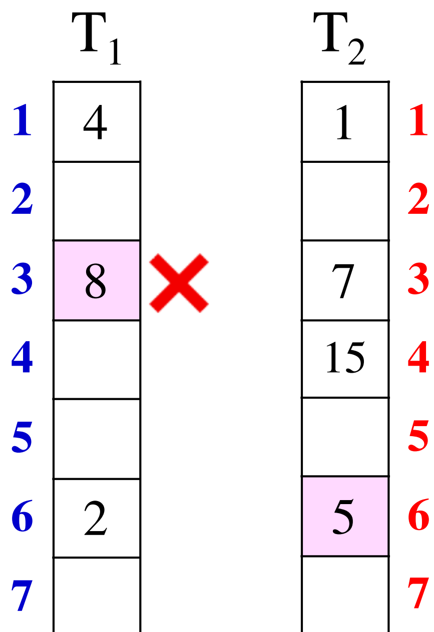
  if $T_1[h_1(x)] = x$ then $T_1[h_1(x)] \leftarrow$ nil

  if $T_2[h_2(x)] = x$ then $T_2[h_2(x)] \leftarrow$ nil

次に、cuckoo hash を用いた delete について解説します。

delete のアルゴリズムもシンプルです。
x が T_1 と T_2 のそれぞれ h_1(x) 番目、h_2(x) 番目にあるかどうかを調べ、あれば削除するだけです。

delete from Cuckoo hash



delete(8, S)

$h_1(8) = 3$
$h_2(8) = 6$

delete(x, S)
  if $T_1[h_1(x)] = x$ then $T_1[h_1(x)] \leftarrow$ nil
  if $T_2[h_2(x)] = x$ then $T_2[h_2(x)] \leftarrow$ nil

例えば、この cuckoo hash から 8 を削除する場合には、このように2か所（ピンクの要素）をチェックし、どちらかに 8 があればそれを削除します。

delete with Cuckoo hash

定理 2

Cuckoo hash による delete(x, S) の最悪時間計算量は $O(1)$ 時間である.

- 高々2つの配列の要素を参照するだけだから.

delete(x, S)
  if $T_1[h_1(x)] = x$ then $T_1[h_1(x)] \leftarrow$ nil
  if $T_2[h_2(x)] = x$ then $T_2[h_2(x)] \leftarrow$ nil

cuckoo hash による delete も、明らかに O(1) 時間で実行可能です。

ここまでは、ほぼ自明な結果です。
次ページ以降で、cuckoo hash の鍵となる insert 操作について解説していきます。

Cuckoo hash の insert の直感的な説明

- insert(x, S) は以下のように行う.

1. $T_1[h_1(x)]$ が空だったら, x を $T_1[h_1(x)]$ に格納する.
2. $T_1[h_1(x)]$ に別の要素 y が入っていたら,
   y を“蹴っ飛ばして”, x を $T_1[h_1(x)]$ に格納する.
   a. $T_2[h_2(y)]$ が空だったら, y を $T_2[h_2(y)]$ に格納する.
   b. $T_2[h_2(y)]$ に別の要素 z が入っていたら,
      z を“蹴っ飛ばして”, y を $T_2[h_2(y)]$ に格納し,
      再帰的に insert(z, S) を行う.

cuckoo hash における insert(x, S) アルゴリズムは以下の通りです。

1. もし T_1[h_1(x)] が空だったら, x を T_1[h_1(x)] に格納します。

2. もし T_1[h_1(x)] に別の要素 y が入っていたら, y を“蹴っ飛ばして”, x を T_1[h_1(x)] に格納します。

   a. もし T_2[h_2(y)] が空だったら, y を T_2[h_2(y)] に格納します。

   b. もし T_2[h_2(y)] に別の要素 z が入っていたら, z を“蹴っ飛ばして”, y を T_2[h_2(y)] に格納し, 再帰的に insert(z, S) を実行する.

insert の例

	T_1
1	
2	
3	
4	
5	
6	15
7	

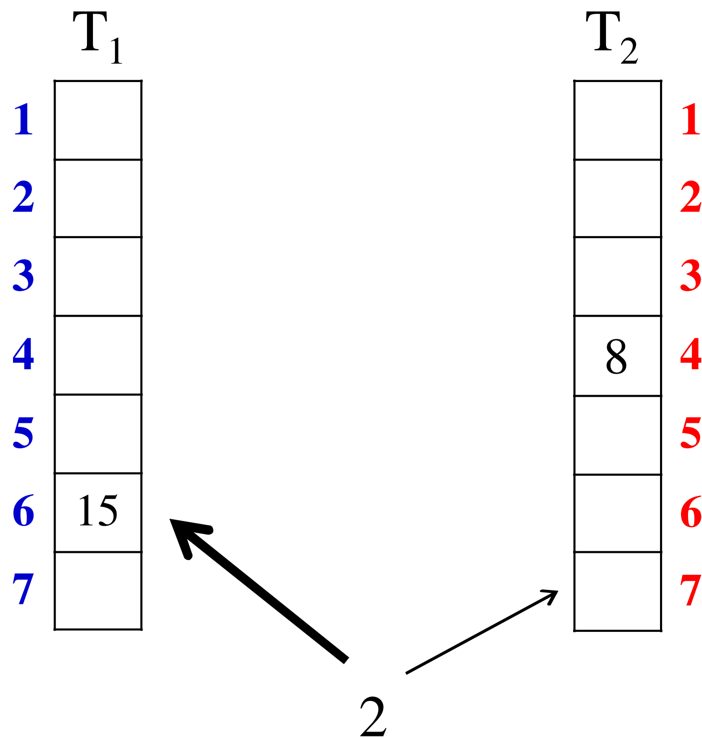
T_2	
	1
	2
	3
8	4
	5
	6
	7

insert(2, S)

insert の例を示します。

この状況で、insert(2, S) を実行することを考えます。

insert の例



insert(2, S)

$h_1(2) = 6$
$h_2(2) = 7$

2 のハッシュ値を計算します。

h_1(2) = 6 なので、T_1[6] をチェックすると、15 があります。

insert の例

	T_1			T_2	
1					1
2					2
3					3
4				8	4
5					5
6	2				6
7					7

insert(2, S)

15

2 が T_1[6] に入り、15 が「蹴っ飛ばされて」出てきました。

そこで、15 の insert を行います。

insert の例

T_1

1	
2	
3	
4	
5	
6	2
7	

T_2

	1
	2
	3
8	4
	5
	6
	7

15

insert(2, S)

$h_1(15) = 6$
$h_2(15) = 4$

15 のハッシュ値を計算します。

h_2(15) = 4 なので、T_2[4] をチェックすると、8 が入っています。

insert の例

	T_1
1	
2	
3	
4	
5	
6	2
7	

T_2	
	1
	2
	3
15	4
	5
	6
	7

insert(2, S)

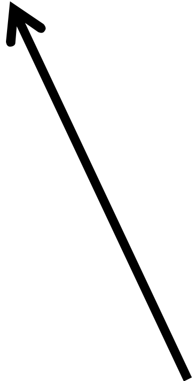
8

15 が T_2[4] に入り、8 が「蹴っ飛ばされて」出てきました。

そこで、8 の insert を行います。

insert の例

T_1 T_2 insert(2, S)



	T_2	
		1
		2
		3
	15	4
		5
		6
		7

8

$h_1(8) = 3$
$h_2(8) = 4$

8 の ハッシュ値を計算します。

h_1(8) = 3 なので、T_1[3] を
チェックすると、空です。

insert の例

T_1

1	
2	
3	8
4	
5	
6	2
7	

T_2

	1
	2
	3
15	4
	5
	6
	7

insert(2, S)

$h_1(8) = 3$
$h_2(8) = 4$

T_1[3] に 8 を格納し、insert 操作が完了しました。

insert の例

	T_1
1	
2	
3	8
4	
5	
6	2
7	

T_2	
	1
	2
	3
15	4
	5
	6
	7

insert(2, S)

$h_1(8) = 3$
$h_2(8) = 4$

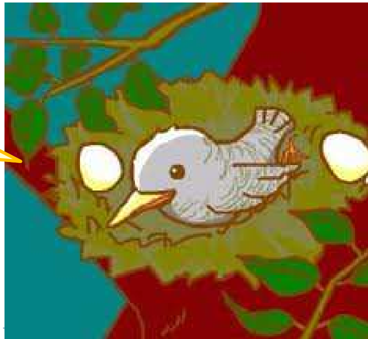
ハッシュ表にすでに入っている要素を
“蹴っ飛ばす”アイディアは
カッコウの習性からヒントを得たもの

cuckoo hash が発表された論文によると、ハッシュ表にすでに入っている要素を“蹴っ飛ばす”というこのアイディアは、カッコウの習性からヒントを得たものだそうです。

余談：Cuckoo hash という名の由来

- カッコウは，他種の鳥（オオヨシキリなど）の巣に卵を産む（托卵）.
- カッコウのヒナは，オオヨシキリのヒナよりも早く羽化し，卵を“蹴っ飛ばして”巣から落としてしまう.
- そうとは知らないオオヨシリキの親鳥は，カッコウを自分の子供と勘違いして，育ててしまう.

エイッ！



(このスライドはカッコウの修正に関する余談です)

カッコウは，オオヨシキリなどの多種の鳥の巣に卵を産みます。この習性を托卵といいます。

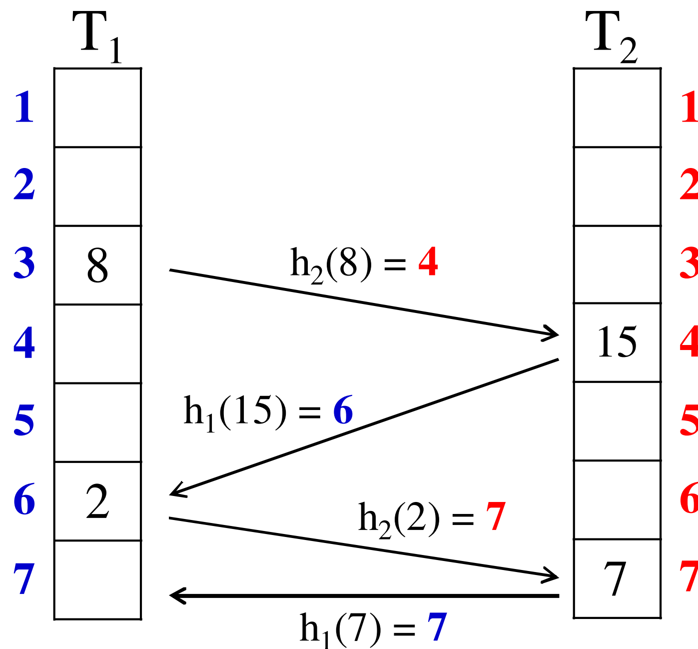
カッコウのヒナは、オオヨシキリのヒナよりも早く羽化し，卵を“蹴っ飛ばして”巣から落としてしまいます。

そうとは知らないオオヨシリキの親鳥は、カッコウを自分の子供と勘違いして、育ててしまいます。鳥のこの習性を刷り込みといいます。

演習問題

提出〆切：７月30日（木）23:59

- S = {2, 7, 8, 15} を格納する下記の Cuckoo hash を用いて，以下の操作を行った際の動作を示せ.
  1. insert(10, S)　ただし，$h_1(10) = 3$, $h_2(10) = 4$
  2. insert(9, $S \cup \{10\}$)　ただし，$h_1(9) = 6$, $h_2(9) = 1$



それでは、今週の演習問題です。

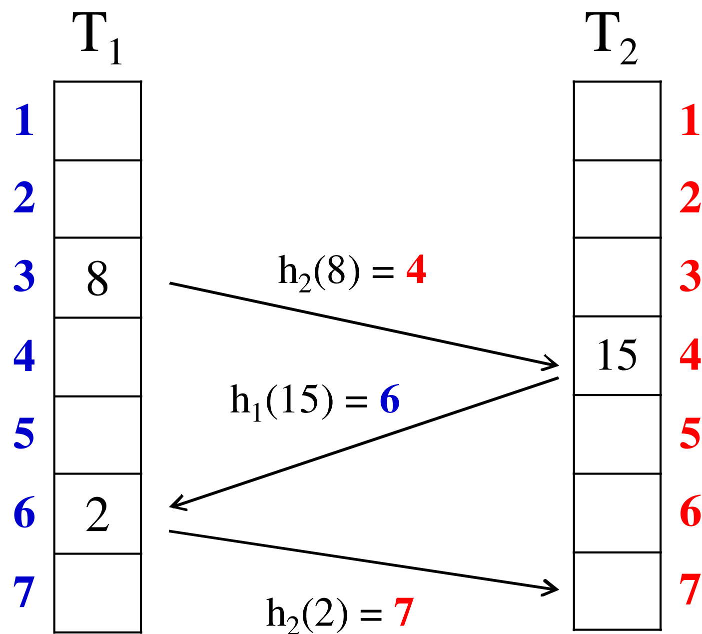
S = {2, 7, 8, 15} を格納する下記の cuckoo hash を用いて，以下の操作を行った際の動作を示してください。

1. insert(10, S)。ただし，h_1(10) = 3, h_2(10) = 4 とします。
2. insert(9, S∪{10})。つまり、1. で 10 を追加したのちに、さらに 9 を追加した状態を示してください。ただし，h_1(9) = 6, h_2(9) = 1 とします。

なお、図の矢印は、それぞれ T_1、T_2 にすでに格納されている要素の、反対側のハッシュ表への行先を示しています。

Cuckoo グラフ

- Cuckoo グラフ : T_1 と T_2 のセルを節点とし, 他方のハッシュ表への行き先を辺とするグラフ.



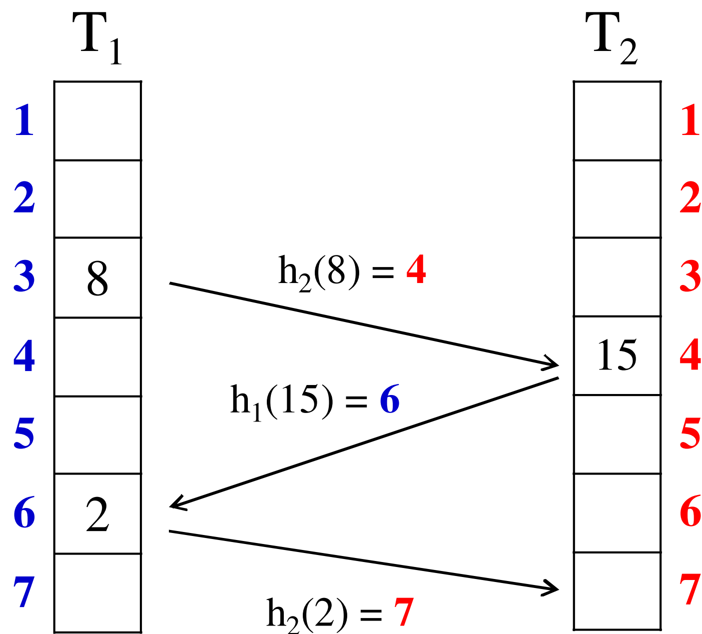
続いて、insert の実行時間の解析を行っていきます。

その準備として、cuckoo グラフを定義します。

cuckoo グラフは、T_1 と T_2 のセルを節点とし、他方のハッシュ表への行き先を辺とするグラフです。
辺は、この図では矢印で表示されています。

Cuckoo グラフ

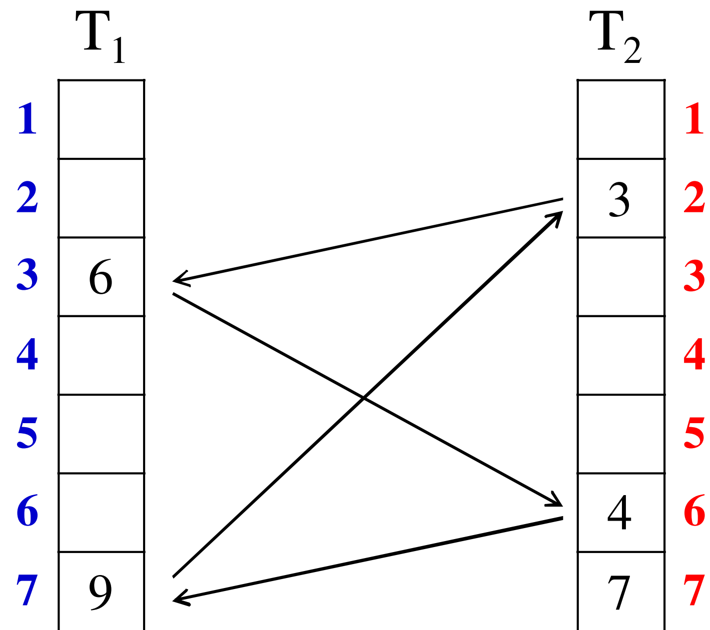
- Cuckoo グラフのパスは，チェイン法の連結リストの拡張と見なせる．



この cuckoo グラフのパスは、チェイン法の連結リストの拡張と見なすことができます。

Cuckoo グラフのサイクル

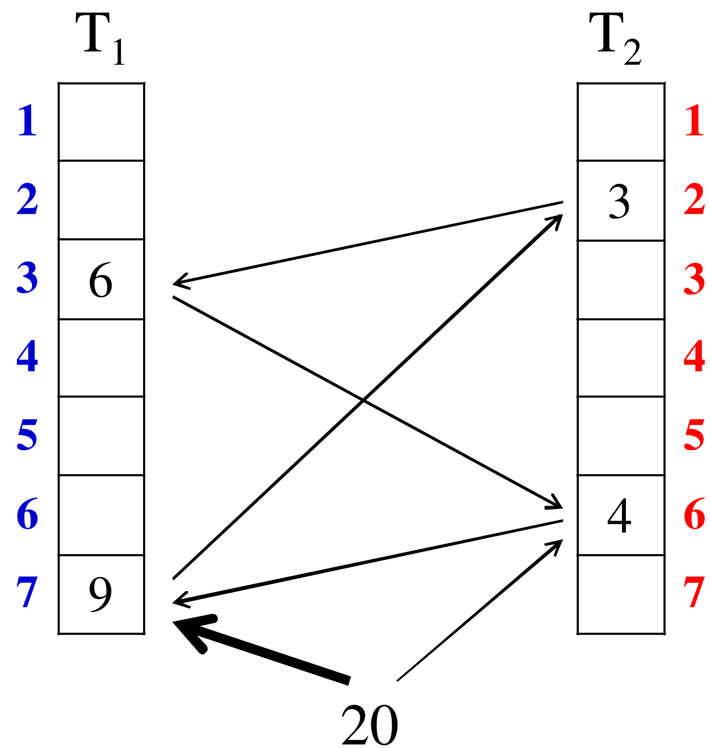
- チェイン法の連結リストとは異なり，Cuckoo グラフにはサイクルがありうる．



ただし、チェイン法の連結リストとは異なり、cuckoo グラフにはこの図の場合のようにサイクルがありえます。

Cuckoo グラフのサイクル

- サイクルがあると，insert が無限ループに陥る．



サイクルがあると、insert の操作中に無限ループに陥ってしまう可能性があります。

insert の “正しい” 疑似コード

insert(x, S)

1. **if** member(x, S) **then return** ;
2. **repeat** n times
3. $x \leftrightarrow T_1[h_1(x)]$;
4. **if** x = nil **then return** ;
5. $x \leftrightarrow T_2[h_2(x)]$;
6. **if** x = nil **then return** ;
7. rehash(); insert(x) ;

ループ回数の上限を決めておく

Cuckoo グラフのサイクルに入ってしまったら，新しいハッシュ関数を使って，S の全要素を insert しなおす．

無限ループを回避するために、insert アルゴリズムでは、ループ回数の上限をあらかじめ決めておきます。

集合 S の要素数が n なので、n 回ループしてしまったら、その後は無限に続いてしまうので、上限を n としておきます。

cuckoo グラフのサイクルに入ってしまったら、同じハッシュ関数を使い続けることができません。

そこで、このアルゴリズムでは、新しいハッシュ関数を使って S の全要素を insert しなおします（7行目）

Cuckoo hash の衝突確率

補題 1

任意の $x, y \in S$ について，x と y が衝突する確率，すなわち x と y が Cuckoo グラフの同一パス上に存在する確率は $O(1/r)$.

前ページのアルゴリズムの性能を評価するために、cuckoo hash の衝突確率を見積もります。

次の補題が成立します。

S の任意の要素 x, y について、x と y が衝突する確率、すなわち x と y が Cuckoo グラフの同一パス上に存在する確率は O(1/r)　です。

補題 1 を示すための準備

補題 2

ある定数 $c > 1$ について，$r \geq cn$ を満たすとする．このとき，無向 Cuckoo グラフの任意の節点 i と j の間に長さ k のパスが存在する確率は高々 $1/rc^k$ である．

- つまり，パス長 k が大きくなればなるほど，2節点間に長さの k パスが存在する確率が指数的に小さくなっていく．

補題1を示すために、まずこの補題2を示します。

ある定数 c > 1 について，r ≥ cn を満たすとします．
このとき，無向 Cuckoo グラフの任意の節点 i と j の間に長さ k のパスが存在する確率は高々 1/rck となります．

つまり、この補題は、パス長 k が大きくなればなるほど，2節点間に長さの k パスが存在する確率が指数的に小さくなっていく、ということを意味しています。

元々の cuckoo グラフは有向グラフですが、辺の向きを無視した無向グラフでパスの存在確率が $1/rc^k$ で抑えられるなら、当然有向グラフでも同じ確率以下で抑えられます。

補題 2 の証明

- $k = 1$ のとき
  - i と j の組み合わせは r^2 通り.
    よって，各 i と j の間に辺がある確率は $1/r^2$.
  - Cuckoo グラフには合計 n 個の辺があるので，
    任意の位置 i と j に辺がある確率は，

$$\sum_{x \in S} (1/r^2) = n/r^2 \leq 1/rc$$

$r \geq cn$ → $n \leq r/c$ より

補題2を証明していきます。数学的帰納法を用います。

k = 1 のときを考えます。

i と j の組み合わせは r^2 通りです。

よって、各 i と j の間に辺がある確率は 1/r^2 となります。

cuckoo グラフには合計 n 個の辺があるので、任意の位置 i と j に辺がある確率は n / r^2 です。

ここで、r ≧ cn より、n ≦ r/c を得ます。

したがって、n / r^2 ≦ 1/rc が成り立ちます。

k = 1 の場合の証明は以上です。

補題 2 の証明

- $k > 1$ のとき
  - i と j の間に長さ $k > 1$ のパスが存在する確率を求めたい.
  - 無向 Cuckoo グラフを考えているので, i と j の間に長さ k 未満のパスは存在しないと仮定.
  - つまり, 以下の場合を考える.

1. 位置 i と, 任意の固定された位置 l の間に, 位置 j を含まない長さk-1の最短パスが存在する.
2. l と j の間に辺が存在する.

k > 1 の場合を考えます。

いま、i と j の間に長さ k > 1 のパスが存在する確率を求めようとしています。

無向 cuckoo グラフを考えているので、i と j の間に長さ k 未満のパスは存在しないと仮定します。
(もし、そのようなパスがあると、無向グラフの同じ辺を何回も辿って、パスを任意に長くするこがとができてしまうため)

つまり、以下の場合を考えます。

1. 位置 i と, 任意の固定された位置 l の間に, 位置 j を含まない長さ k-1 の最短パスが存在する.
2. l と j の間に辺が存在する.

補題２の証明

- $k > 1$ のとき（つづき）
  - 数学的帰納法の仮定より，i と l の間に長さ k-1 のパスが存在する確率は高々 $1/rc^{k-1}$.
  - l と j の間に辺がある確率は，高々 $1/rc$.（$k = 1$の場合の議論より）
  - よって，すべての l について合計すると，

$$\sum_{l=1}^{r}(1/rc^{k-1})(1/rc) = r(1/rc^{k-1})(1/rc) = \frac{1}{rc^{k}}$$

（つづき）

数学的帰納法の仮定より、i と l の間に 長さ k-1 のパスが存在する確率は高々 1/rc^{k-1} です。

k = 1の場合の議論より、l と j の間に辺がある確率は、高々 1/rc です。

よって，すべての l について合計すると、このような式になり、これを整理すると 1/rc^k が得られます。

k > 1 の場合も証明できたので、補題2を示すことができました。

補題 1 の証明

補題 1

任意の $x, y \in S$ について，x と y が衝突する確率，すなわち x と y が Cuckoo グラフの同一パス上に存在する確率は $O(1/r)$.

- x と y が同一パス上に存在するのは以下の4通り：
  $h_1(x)$ と $h_1(y)$，$h_1(x)$ と $h_2(y)$, $h_2(x)$ と $h_1(y)$, $h_2(x)$ と $h_2(y)$.
- 補題2より，すべてのパス長 k について和をとると

$$4\sum_{k=1}^{\infty} \frac{1}{rc^k} = \frac{4}{r}\sum_{k=1}^{\infty} \frac{1}{c^k} = \frac{4}{r} \cdot \frac{1}{c-1} = O\left(\frac{1}{r}\right)$$

では、補題1の証明を行います。

x と y が cuckoo グラフの同一パス上に存在するのは以下の4通りです。

$h_1(x)$ と $h_1(y)$,
$h_1(x)$ と $h_2(y)$,
$h_2(x)$ と $h_1(y)$,
$h_2(x)$ と $h_2(y)$.

補題2から、すべてのパス長 k について和をとるとこのような式となり、これを整理すると O(1/r) が得られます。

rehash の時間計算量

- Cuckoo hash の insert において，サイクルに入らなかった場合，補題 1 を使ってチェイン法と同様の議論ができ，この場合の時間計算量の期待値は $O(1)$ となる．
- では，サイクルに入ったときに，rehash に要する時間はどれほどだろうか？

補題 3

任意の定数 $c > 2$ について，n 個の要素を insert する間に起きる rehash の回数の期待値は $O(1)$ である．また，rehash のならし時間計算量の期待値は $O(1)$ である．

cuckoo hash の insert において、サイクルに入らなかった場合は、先ほどの補題 1 を使ってチェイン法と同様の議論ができます。
したがって、サイクルに入らなかった場合の insert の時間計算量の期待値は O(1) となります.

では、サイクルに入ったときに, rehash に要する時間はどれくらいになるのでしょう？

これについて、以下の補題3が成立します。

任意の定数 c > 2 について、n 個の要素を insert する間に起きる rehash の回数の期待値は O(1) です。
また、rehash のならし時間計算量の期待値は O(1) です。

補題 3 の証明

- サイクルとは，位置 i から位置 i へ戻ってくるパスのことである．したがって，補題 2 の $j = i$ の場合を考える．
- 一般性を失うことなく，i を T_1 の位置とする．
位置 i から位置 i に戻ってくる任意のサイクルの長さは偶数である．
- T_1 のすべての位置について補題 2 の確率を足すと

$$r\sum_{h=1}^{\infty}\frac{1}{rc^{2h}}=\sum_{h=1}^{\infty}\frac{1}{c^{2h}}<\sum_{k=1}^{\infty}\frac{1}{c^{k}}=\frac{1}{c-1}$$

- 上記のサイクルは T_2 の位置を必ず含んでいる．
よって，Cuckoo グラフにサイクルが存在する確率は高々 $1 / (c\text{-}1)$ である．

補題3を証明します。

サイクルとは、位置 i から位置 i へ戻ってくるパスのことです. したがって、補題 2 の j = i の場合を考えればよいことになります。

一般性を失うことなく, i を T_1 の位置とします。位置 i から位置 i に戻ってくる任意のサイクルの長さは必ず偶数です。

T_1 のすべての位置について補題 2 の確率を足すと、このような式が得られ、これを整理すると 1/c-1 未満であることがわかります。

上記で考えたサイクルは、T_2 の位置を必ず含んでいます。よって、T_2 から始まるサイクルを追加して考える必要はありません。

よって、Cuckoo グラフにサイクルが存在する確率は高々 1 / (c-1) となります。

補題 3 の証明

- 簡単のため $c = 3$ とすると, n 回の insert の間に1つのサイクルが存在する確率は 1 / 2, 2つのサイクルが存在する確率は 1 / 4, 3つなら 1 / 8, ・・・
- よって, n 回の insert 中に起きる rehash の回数の期待値は

$$\sum_{h=1}^{\infty} (1/2^h) = 1$$

- サイクルに入らない場合の insert の時間計算量の期待値は $O(1)$ なので, n 個の要素を insert しなおす時間計算量の期待値は $O(n)$.
- よって, rehash のならし時間計算量の期待値は $O(1)$.

簡単のため c = 3 とします。

このとき、n 回の insert の間に１つのサイクルが存在する確率は 1 / 2,
2つのサイクルが存在する確率は 1 / 4,
3つなら 1 / 8, ・・・

というようにサイクルの個数に対して、確率は指数的に小さくなっていきます。

よって、n 回の insert 中に起きる rehash の解すの期待値は、1/2 + 1/4 + 1/8 + … = 1 となります。

サイクルに入らない場合の insert の時間計算量の期待値は O(1) なので、n 個の要素を insert しなおす 時間計算量の期待値は O(n) です.

よって, rehash のならし時間計算量の期待値は O(1) となります（証明終）

Cuckoo hash の insert の時間計算量

定理 3

Cuckoo hash による insert(x, S) の
ならし時間計算量の期待値は $O(1)$ である.

結論として、この定理を得ます。

cuckoo hash による insert(x, S)
のならし時間計算量の期待値
は O(1) です。

万能ハッシュ関数

- そもそも，万能ハッシュ関数なんて都合のいいものがあるのだろうか？
- $p = u+1$ を素数とする．ハッシュ関数

$$h(x) = (ax \bmod p) \bmod r$$

の a を U からランダムに選んだとき，
$h(x)$ は万能ハッシュ関数の要件を満たす．

万能ハッシュ関数について少しだけ触れておきます。

本日の議論はすべて、万能ハッシュ関数の存在を仮定して行ってきました。

p = n+1 を素数とします。このとき、ハッシュ関数 h(x) = (ax mod p) mod r の a を全体集合 U からランダムに選んだとき、h(x) は万能ハッシュ関数の要件を満たすことが知られています。

ここで、mod は余りを計算する演算（剰余）です。