大阪電気通信大学 情報通信工学部 光システム工学科 2年次配当科目

コンピュータアルゴリズム

探索アルゴリズム(2)

第8講: 平成20年11月28日 (金) 4限 E252教室

中村 嘉隆(なかむら よしたか)
奈良先端科学技術大学院大学 助教
y-nakamr@is.naist.jp
http://narayama.naist.jp/~y-nakamr/

第7講の復習

- 探索アルゴリズム
  - 探索するデータ構造
    - レコードの列 → 表
  - 線形探索(linear search)
    - 前から順に探索 (探索 $O(n)$)
  - 2 分探索(binary search)
    - 整列された領域の中央の値を調べ, 領域を半減させながら探索 (探索 $O(\log n)$)
  - 2 分探索木(binary search tree)
    - 大小関係を木構造で表して探索(探索 $O(\log n)$)

今日の講義の内容

- 探索アルゴリズム
  - 線形探索・2 分探索・2 分探索木の復習
  - 平衡木
    - できるだけ完全 2 分探索木になるように, 要素の追加・削除時に木の形を再構成
    - 平衡木の例として AVL 木を紹介
  - ハッシュ法
    - ハッシュ関数を使って, 探索の計算量を $O(1)$ に近づける

復習: 探索(サーチング)問題とは

- サーチング: Searching, 探索
  - n 個のレコード列から, キーの値を指定して, それと等しいキーを持つレコードを選ぶ処理
- レコード(record)とキー(key)
  - **レコード**とは, ひとかたまりのデータ
  - **キー**とは, レコードの中にある 1 つのフィールド(要素)
  - 例:成績{学籍番号, 名前, 出席点, 試験点}
    - レコードは 1 人分のデータ(例:{5433, 中村, 30, 55})
    - キーは, 要素のどれか(例えば, 学籍番号)
  - ここでは簡単のため同じキーを持つレコードは複数存在しないとする

復習: 探索するレコードの表とサイズ

- 探索はある列 (表) に対して行う
  - その表を作るのに必要な計算量も考慮が必要
  - 問題のサイズ
    =レコード数
- 表の分類
  - 静的な表
    - 一度表を作ると二度と作り替えない
      ➡ 探索さえ早くすればよい
  - 動的な表
    - 表を作ったあとでも, レコードの追加, 削除がある
      ➡ レコードの追加, 削除の手間も考慮

番号	名前	点数
1	たろう	76
2	はな	82
3	こん	74

問題のサイズ n

レコード

キー

復習: 線形探索

- 線形探索: linear search, sequential search, 逐次探索, 順探索
  - アルゴリズム
    - 配列, またはリストに並べられたデータを一つ一つ順に端から調べる

5 回優勝した横綱は?(キー: 優勝回数)

143kg の横綱は?(キー: 体重)

	朝青龍	武蔵丸	若乃花	貴乃花	曙	旭富士	大乃国
	139kg	235kg	134kg	159kg	232kg	143kg	203kg
	15回	12回	5回	22回	11回	4回	2回
	[1]	[2]	[3]	[4]	[5]	[6]	[7]

復習：線形探索のまとめ

- 入力
  - レコードの列（並び方は自由）
- アルゴリズム
  - 前から順番にキーを調べていく
- 計算量
  - 探索 $O(n)$，表への追加 $O(1)$，削除 $O(n)$
- その他
  - 番兵による高速化
  - 応用例：自己再構成リスト

復習：2 分探索

- 2 分探索：binary search
  - 入力はキーであらかじめ整列された列（表）とする
    - 整列は前に勉強した
  - キーの大小判定することで，目的のキーが列（表）の前にあるか後ろにあるか判断できる
  - 列の中央の要素のキーと探索したいキーを比較し，探索する領域を半減させる

復習：2 分探索の概念図

- キー 21 を持つ動物を探したい
  - lo = 1, hi = 16, mid = 8

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]
キー	5	8	13	19	21	26	33	34	36	40	45	55	58	69	74	81
	虎	牛	馬	猫	鶏	犬	鷹	鼠	狸	兎	羊	豚	猿	狐	人	魚

  - lo = 1, hi = 7, mid = 4

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]
5	8	13	19	21	26	33	34	36	40	45	55	58	69	74	81
虎	牛	馬	猫	鶏	犬	鷹	鼠	狸	兎	羊	豚	猿	狐	人	魚

  - lo = 5, hi = 7, mid = 6

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]
5	8	13	19	21	26	33	34	36	40	45	55	58	69	74	81
虎	牛	馬	猫	鶏	犬	鷹	鼠	狸	兎	羊	豚	猿	狐	人	魚

  - lo = 5, hi = 5, mid = 5　見つかった！！

復習：2 分探索のデータ構造

- データ構造は配列型
  - 配列型はランダムアクセスが可能
    - 添え字でちょうど真ん中の位置のレコードにアクセスできる
  - リストはランダムアクセス不可能（前から辿るのみ）
- レコードの追加，削除は整列された状態を保持する必要がある
  - 追加は，探索して入る位置を決めた後，その後ろの要素を後ろにずらして挿入
  - 削除は，位置を探索した後，その後ろの要素を前にずらす

復習：2 分探索のデータ構造：追加と削除

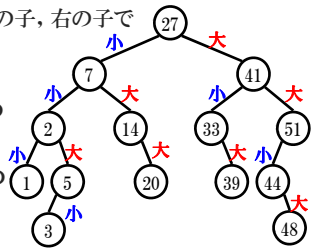
- レコードの追加
  - 追加する位置の探索
    - これは 2 分探索すれば $O(\log n)$ で求まる
    - プログラムで見つからなかった場合に −1 を返すのではなく，直前の位置を返すようにすればよい
  - 配列への要素の挿入
    - 追加位置から後ろのレコードは 1 つずつ後ろにずらす必要がある $O(n)$
  - $O(\log n) + O(n) = O(n)$
- レコードの削除
  - 削除する位置の探索 $O(\log n)$
  - 配列の要素の削除 $O(n)$
  - $O(\log n) + O(n) = O(n)$

復習：2 分探索のまとめ

- 入力
  - 探索するキーで整列されたレコードの列
- アイデア
  - 探索するキーと，列の中央の要素のキーの大小関係で探索範囲を半減させる
- 計算量
  - 探索 $O(\log n)$，表への追加 $O(n)$，削除 $O(n)$
- その他
  - 線形探索に比べて，探索の計算量は小さいが，追加の計算量が多い
  - 表への追加が多い（動的な）場合はおすすめできない
  - 静的な表への探索に向いている

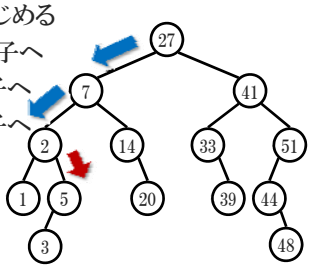
復習：2 分探索木とは

- 以下の特徴を持つ木構造
  - 各節点は最大で 2 個の子を持つ
    - その 2 個の子は，左の子，右の子である
  - 左の子（子孫）は，親より小さな値を持つ
  - 右の子（子孫）は，親より大きな値を持つ



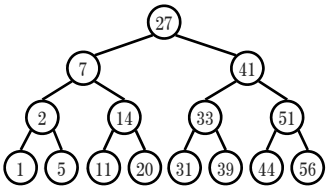
復習：2 分探索木の概念図

- キー 5 を持つノードを探したい
  - 根（キー：27）からはじめる
  - 5 < 27 なので，左の子へ
  - 5 < 7 なので，左の子へ
  - 2 < 5 なので，右の子へ
  - 5 = 5 なので，終了



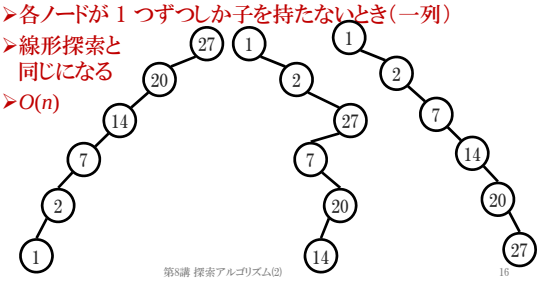
復習：2 分探索木の計算量

- 探索の計算量
  - 最良の場合
    - 完全 2 分木のとき
      - ノード数 n (= $2m$) に対して木の高さは $\log n$ (= m)
      - 最大でも $\log n$ 回木を辿れば，目的のノードに辿り着く
    - $O(\log n)$
  - 平均的な場合
    - このときも最良の場合の 1.39 倍しか悪化しない（証明略）
    - $O(1.39 \log n)$ $=O(\log n)$



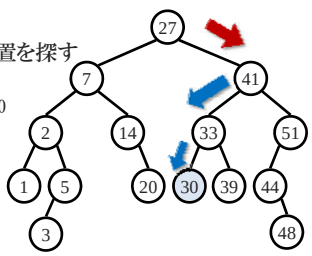
復習：2 分探索木の計算量

- 探索の計算量
  - 最悪の場合
    - 各ノードが 1 つずつしか子を持たないとき（一列）
    - 線形探索と同じになる
    - $O(n)$



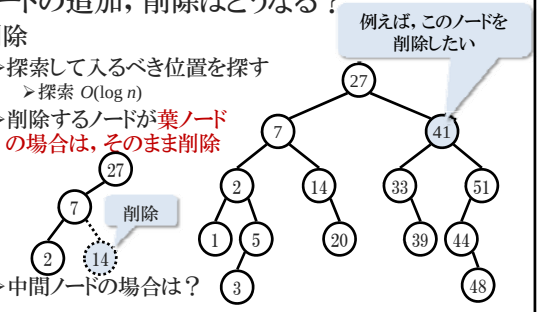
復習：2 分探索木のデータ構造

- リスト型で木構造を作る
- レコードの追加，削除はどうなる？
  - 追加
    - 探索して入るべき位置を探す
      例：キー 30 のデータ
      - 27 → 41 → 33 → 30
    - 探索 $O(\log n)$
    - 挿入は $O(1)$
    - 全体で $O(\log n) + O(n)$ $= \mathbf{O(\log n)}$



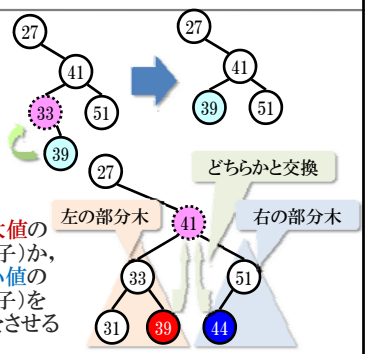
復習：2 分探索木のデータ構造

- レコードの追加，削除はどうなる？
  - 削除
    - 探索して入るべき位置を探す
      - 探索 $O(\log n)$
    - 削除するノードが葉ノードの場合は，そのまま削除
    - 中間ノードの場合は？



復習：2 分探索木からのノードの削除

- 中間ノードの削除
  - 子が 1 つの場合
    - 子を親とつなげる
  - 子が 2 つの場合
    - 左の部分木の最大値のノード（最も右奥の子）か，右の部分木の最小値のノード（最も左奥の子）を持ってきて代わりをさせる



復習：2 分探索木の削除の計算量

- 削除ノードの探索
  - $O(\log n)$
- 削除するノードが葉ノードの場合
  - $O(1)$ で削除可能
- 中間ノードの場合
  - 交換候補を左右どちらかの部分木を辿って見つける → $O(\log n)$
  - 見つかったら交換は $O(1)$ で可能
- 削除全体では，
  $O(\log n)+\{O(\log n)+O(1)\} = O(\log n)$

復習：2 分探索木の計算量のまとめ

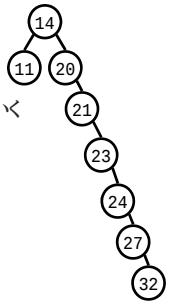
- 探索の計算量
  - 平均 $O(\log n)$，最悪 $O(n)$
  - 最悪 $O(n)$ なので保証が必要なら使わない方がよい
- 表へのレコードの追加，削除の計算量
  - 追加 $O(\log n)$
  - 削除 $O(\log n)$

追加削除も小さい計算量で可能

- データ構造はリストを使って木構造にする

復習：2 分探索木の落とし穴

- 木の形が最悪になりやすいことがある
  - 途中でどんどんレコードが追加されるとする（動的）
  - このとき，ある程度整列された順で追加されると，木の形が一直線になっていく
  - 例：{14,11,20} の木に，21，23，24，27，32 のキーの要素が入ってくるとする
  - このような入力は与えやすいので注意
  - そのような入力が予想されるときには 2 分探索木は使わない方がよい



復習：2 分探索木のまとめ

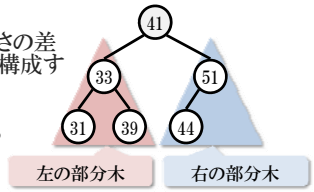
- 入力
  - 左の子孫は小さなキー，右の子孫は大きなキーを持つ 2 分木
- アイデア
  - 各ノードのキーと探索したいキーを大小比較することで，探索範囲を片方の部分木に限定していく
- 計算量
  - 探索 平均 $O(\log n)$，最悪 $O(n)$
  - 表への追加 平均 $O(\log n)$，削除 平均 $O(\log n)$
- その他
  - 最悪で $O(n)$ になるため注意が必要（平均は $O(\log n)$）
  - 整列されたデータを追加していくと木の形が直線的になり，計算量が最悪に近づく

平衡木

- 平衡木（balanced tree）
- 2 分探索木の欠点
  - 偏った木の形（子が1つしかない節点が多い木）だと探索が $O(n)$ になる
  - 完全 2 分木の形が理想
    - できるだけ左右の部分木の大きさを揃えたい
- AVL 木
  - Adel'son-Vel'skii と Landis が考案
  - 各節点の左右の部分木の深さの差を 1 以内にした木
  - 探索の計算量が最悪でも $O(\log n)$ を保証

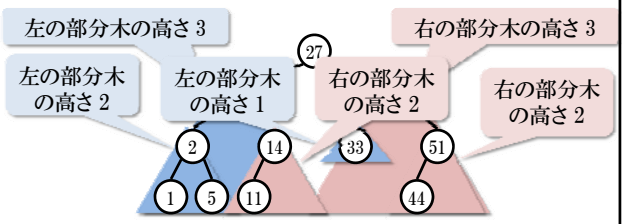
AVL 木のアイディア

- 要素の追加，削除が起こったときに木の形が偏るなら再構成する
  - ただし再構成の計算量が $O(\log n)$ を超えてはいけない
- 完全にバランスさせる必要はない
  - 最悪でも $O(\log n)$ にさえなれば良い
- 各節点の左右の部分木の高さの差が 2 以上になったら，木を再構成する
  - 部分木
    - ある節点より子孫で構成される部分的な木
  - 高さの差 −1, ±0, +1 は許す



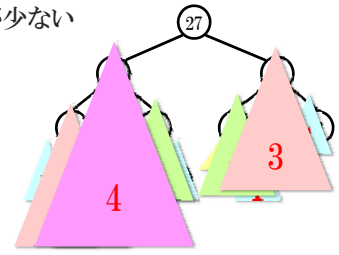
AVL 木

- 左右の部分木の高さの差が高々 1 の 2 分探索木



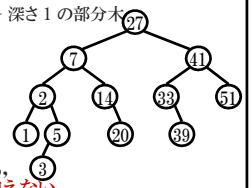
AVL 木での探索の最悪計算量

- 最も偏った形の AVL 木
  - 全ての頂点で木の高さが 1 だけ違い，最も頂点数が少ない



AVL 木での探索の最悪計算量

- 最も頂点数が少ない最も偏った AVL 木の頂点数
  - 各高さの部分木で最も頂点数の少ない場合
    - 深さ 1 の頂点数 $N(1) = 1$
    - $N(2)$ = 部分木の根 + 深さ 1 の部分木 + 深さ 0 の部分木 $= 1 + N(1) + N(0) = 1 + 1 + 0 = 2$
    - $N(3)$ = 部分木の根 + 深さ 2 の部分木 + 深さ 1 の部分木 $= 1 + N(2) + N(1) = 1 + 2 + 1 = 4$
    - つまり深さ h の場合 $N(h) = 1 + N(h\text{-}1) + N(h\text{-}2)$
  - 漸化式を解くと $N(h) = O(\frac{(1+\sqrt{5})^h}{2})$
  - 高さに対して頂点数は指数的に増える，頂点数に対して高さは対数的にしか増えない
    - ➡ 最悪時でも $O(\log n)$

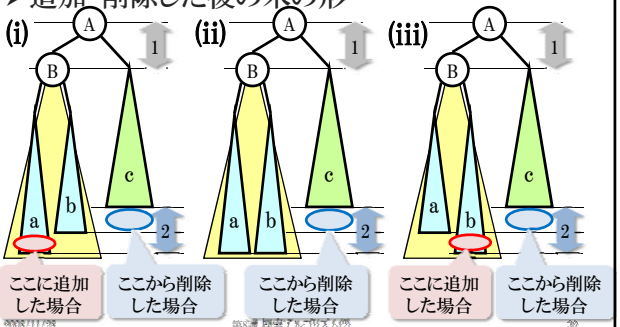


AVL 木への要素の追加，削除

- 手順は次の 2 ステップ
  - 2 分探索木と同様に場所を探し，挿入・削除
  - その結果，木の形が AVL 木の条件を満たさなくなったら再構成
- 挿入後の木の形の可能性
  - 各節点の左右の部分木の高さの差が高々 1 以内
    - ➡ AVL 木の条件を満たすので再構成なし
  - 高さの差が 2 以上になる節点が出てくる
    - ➡ 再構成

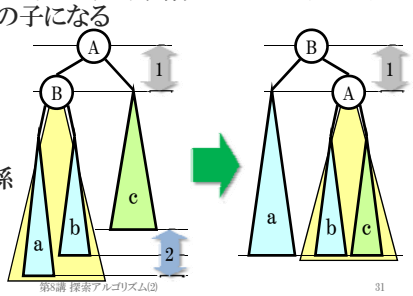
AVL 木の再構成を必要とする形

- 追加・削除した後の木の形



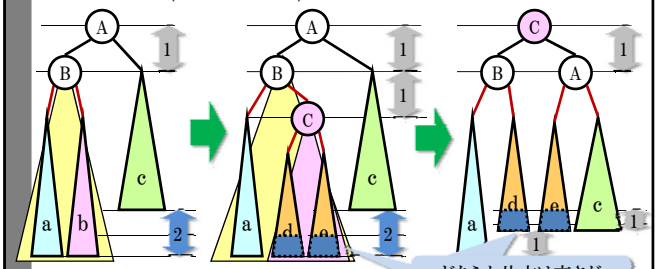
AVL 木の再構成 (i)

- A と B を付け替え，B を親とする
  - 節点 A と B のキーの大小関係は B < A なので，A は B の右の子になる
- 部分木 b は A の左の部分木にする
  - 部分木 b は A の左の子孫
  - つまり全て A より小さい



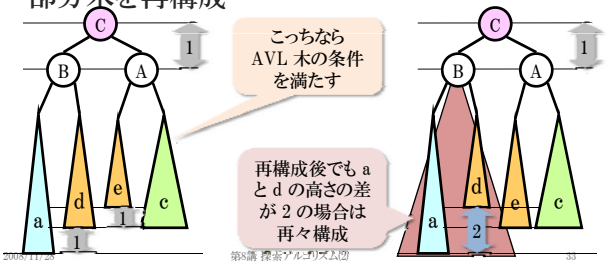
AVL 木の再構成 (ii)（削除のみ）

- a < B < b < A < c
- a < B < (d < C < e) < A < c



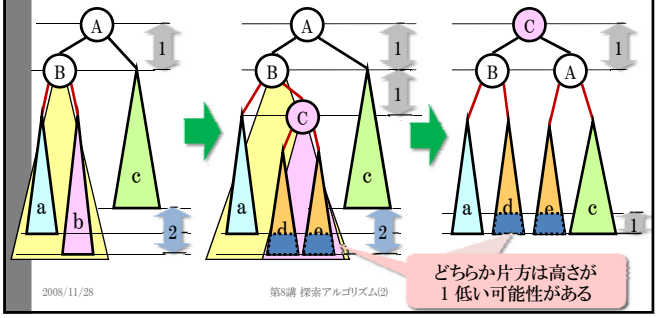
AVL 木の再構成 (ii)（削除のみ）

- (ii) の再構成をした結果，以下の a と d のように，まだ高さの差が 2 ある場合は，B 以下の部分木を再構成



AVL 木の再構成 (iii)

- a < B < b < A < c
- a < B < (d < C < e) < A < c



再構成の計算量

- 追加，削除する位置の探索 $O(\log n)$
- 部分木の高さの調査 $O(\log n)$
- 節点の付け替え $O(1)$

- つまり，再構成に必要な計算量は $O(\log n)$
  - ちなみに，ランダムに要素の追加・削除を行った場合に再構成が発生する確率は，追加 約 47%，削除 約 21% という実験結果がある

AVL 木のまとめ

- 2 分探索木の拡張
- 各節点において，左右の部分木の高さの差が高々 1 になるように常に保つ
- 要素の追加・削除時に必要に応じて木の再構成を行う
- 計算量
  - 探索の計算量 最悪でも $O(\log n)$
  - 探索 $O(\log n)$，追加 $O(\log n)$，削除 $O(\log n)$，再構成 $O(\log n)$
- 木の再構成の操作の分，アルゴリズムが複雑

ハッシュ法

- ハッシュ(hash)
- いままでとはまったく違うアイデア
- うまく設計すれば，探索・追加・削除の計算量を平均して全て $O(1)$ にできる
- 事実上最速の探索アルゴリズム
- 実用上非常に有益

- しかし，やはり欠点もある

ハッシュ法のアイディア

- いままでの探索アルゴリズム
  - キーの値の比較が基本
  - 最も効率が良くても探索領域の半減 → $O(\log n)$
- ハッシュ法のアイデア
  - キーの値の範囲が分かっているとする
    例：1 から 100
  - その場合，添え字 1 から 100 までの配列を用意
  - キー x のデータがほしい場合は，配列 $[x]$ にダイレクトアクセス
    → $O(1)$!!!!!

キー 3

キー 9

	キー	レコード
[1]	1	ada
[2]		未使用
[3]	3	few
[4]		未使用
[5]		未使用
[6]	6	def
[7]		未使用
[8]	8	beg
[9]	9	cek
[10]	10	rok
[11]		未使用
[12]	12	ff

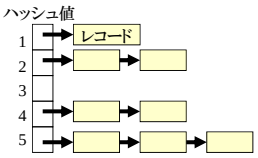
ハッシュ法のアイディア

- 先ほどの配列を使う方法の欠点
  - なかなかキーの範囲が分かることは少ない
  - それにキーが正整数のみとも限らない
  - 範囲が広すぎるとメモリがたくさん必要
- ある関数を定義して，キーを変換
  例：キーが整数のとき，下 2 桁の添え字を持つ配列の位置に格納する(この場合，関数 $h(x) = x \bmod 100$ となる)
  - このような下 2 桁の値を**そのキーのハッシュ値**という
    - キー 1345 のレコードはハッシュ値 45 なので配列 [45] へ
    - メモリ領域も 100 で済む
    - じゃ，キー 945 のレコード(これもハッシュ値 45)もあった場合どうする？？

mod とは剰余(余り)を求める演算子

チェイン法と開番地法

- チェイン法
  - レコードを追加するとき，既に同じハッシュ値を持つレコードがあるときは**リストでつなげる**
  - 探索するとき，同じハッシュ値を持つレコードが 2 つ以上ある場合はリストを辿る
- 開番地法
  - レコード x を追加するとき，ハッシュ値 $h(x)$ の場所にレコードがある場合は，$h(x)+1$ に**そのレコードを格納**する
  - 探索するとき，$h(x)$ の位置から順に調べる必要がある



身近なハッシュ法の例

- 辞書
  - **目次のある**辞書
    - 目次で「ア」「カ」「サ」「タ」…の場所を調べる
    - タ行の項目なら，目次の「タ」のページから調べればよい
    - 辞書は開番地法になっている
    - 人間は目次の項目がたくさんあると目次を読むのに時間がかかるが，計算機は機械的な計算で値が求まるので目次の項目が多くても問題ない
  - 2 分探索で例に出したのは目次のない辞書

ハッシュ法の欠点

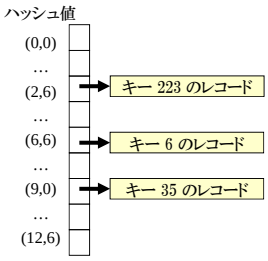
- 同じハッシュ値を持つレコードが多いと効率が悪くなる
  - できるだけレコードがもつハッシュ値が均等にバラけるようにしないといけない
- キーの数に比べて，ハッシュ値の数が少ないとき効率が悪くなる
  例：目次の項目が少ない，「ア」と「ハ」しかない
  - 同じハッシュ値を持つレコード数が増える
  - リストを辿る場合は，線形探索になる
  - レコード数 n，ハッシュ値数 h とすると，各ハッシュ値の平均リスト長は n/h，線形探索で $O(n/h)$

ハッシュ関数

- 元のレコードのキーからハッシュ値を求める関数
  - 異なる入力に対して，できるだけバラけたハッシュ値を返すようにする
- よく使われる手法
  - 剰余（割り算の余り）を使う $h(x) = x \bmod 567$
- 偏りをなくす工夫
  - 複数のハッシュ関数を組み合わせる
  - $h_0(x)$, $h_1(x)$, $h_2(x)$, $h_3(x)$, …… を用意すると同じハッシュ値を持つ可能性が減る
  - と言っても，たくさん用意するのは面倒なので 2 つ $h(x)$と $g(x)$ を用意し，$h_0(x) = h(x)$, $h_1(x) = h(x) + g(x)$, $h_2(x) = h(x) + 2 \times g(x)$, $h_3(x) = h(x) + 3 \times g(x)$, …… とする
    - ⇒ 2 重ハッシュ法（double hashing）

ハッシュ法の概念図

- ハッシュ関数 $h_0(x) = x \bmod 13$, $h_1(x) = x \bmod 7$
- ハッシュ値 $h(x)$ は $(h_0(x), h_1(x))$ とする
  - キー 6 → (6,6)
  - キー 35 → (9,0)
  - キー 223 → (2,6)
- 表のサイズ
  - 13 × 7 = 91 エントリ



ハッシュ法での追加と削除

- 同じハッシュ値を持つレコード数 $O(k)$ とする
- 追加すべき位置は $O(1)$，削除すべき位置は $O(1) + O(k)$ の探索で求まる
- チェイン法の場合は，リストの追加と削除
  - 追加・削除とも $O(1)$
- 開番地法の場合
  - 追加は開いている場所までさらに移動 $O(k)$，削除はその場所の使用済みフラグを解除 $O(1)$
- 両方とも，追加・削除 $O(k)$ でできる
- ここで $k=n/h$
  - n：レコード数，h：ハッシュ値数

ハッシュ値のまとめ

- レコード数 n，ハッシュ値数 h のとき，探索 $O(n/h)$，追加・削除 $O(n/h)$ の計算量
- ハッシュ値数が十分あれば，全て平均 $O(1)$
- ハッシュ値が重なったレコードの処理
  - チェイン法：リストでつなぐ
  - 開番地法：その番地以降で開いているところに入れていく
- ハッシュ関数
  - ハッシュ値を導く関数
  - できるだけバラけた値を導出することが望ましい
    - 剰余関数（mod）が良く使われる
    - 複数のハッシュ関数を組み合わせる 2 重ハッシュ法がある

探索アルゴリズムのまとめ

名前	探索	追加	削除	備考
線形探索	$O(n)$	$O(1)$	$O(n)$	配列，リストどっちも可
2 分探索	$O(\log n)$	$O(n)$	$O(n)$	配列で実現，リスト不可
2 分探索木	平均 $O(\log n)$ 最悪 $O(n)$	平均 $O(\log n)$ 最悪 $O(n)$	平均 $O(\log n)$ 最悪 $O(n)$	整列されたデータの追加に弱い
平衡木（AVL 木）	$O(\log n)$	$O(\log n)$	$O(\log n)$	追加・削除時に再構成が必要
ハッシュ法	平均 $O(1)$ 最悪 $O(n)$	平均 $O(1)$ 最悪 $O(n)$	平均 $O(1)$ 最悪 $O(n)$	レコード数とハッシュ値数の比，ハッシュ関数の精度に依存

第 8 講のまとめ

- 探索アルゴリズム
  - 2 分探索木
  - AVL 木
    - 2 分探索木の拡張
    - できるだけ完全 2 分探索木に近づくように木の構成を保つ
      - 要素の追加，削除時に必要なら木の形を再構成
  - ハッシュ法
    - 場合によっては $O(1)$ で探索可能