アルゴリズム B 第 5 回　　2021 年 11 月 8 日

5　データの探索

データの列から指定したデータを探索する方法を考える．一般には，データはいくつかの項目の組からなり，どの項目で探索するか指定する．探索の対象となる項目を**キー** (key) と呼ぶ．以下では簡単のため，キーの値は重複しないものとするが，キーの値が重複する場合にも以下のアルゴリズム・データ構造は適切な修正により適用可能である．

実数 x に対して，x 以下の整数で最大のものを $\lfloor x \rfloor$ と表し，x 以上の整数で最小のものを $\lceil x \rceil$ と表す．$\lfloor \cdot \rfloor$ を**床関数** (floor function)，$\lceil \cdot \rceil$ を**天井関数** (ceiling function) という．

逐次探索

逐次探索 (sequential search)（**線形探索** (linear search)）は，先頭から順にキーを調べるアルゴリズムである．長さ n の配列の場合，最悪計算量は $O(n)$ である．配列だけでなく，連結リストなどにも適用可能である．

2 分探索

2 分探索 (binary search) は，ソートされた配列からデータを探すアルゴリズムである．キーは比較可能であるとして，配列 $a[1], a[2], \ldots, a[n]$ に対して，$a[1] < a[2] < \cdots < a[n]$ が成り立つとする．次の関数 BinarySearch$(a[\,], x, i, j)$ は，$a[k] = x$ となる添字 k を $i \le k \le j$ の範囲で探す．

BinarySearch$(a[\,], x, i, j)$
  if $i > j$ **then return**(“探索失敗”)
  $m = \lceil (i + j)/2 \rceil$
  if $(a[m] = x)$ **then return**(m)
  if $(a[m] < x)$ **then** BinarySearch$(a[\,], x, m + 1, j)$
  else BinarySearch$(a[\,], x, i, m - 1)$

最悪計算量は $O(\log n)$ である．

2 分探索木

2 分木 T において，各頂点にデータが割り当てられているとする．2 分探索の場合と同様に，キーは比較可能であるとする．T が **2 分探索木** (binary search tree) であるとは，各頂点の値が，左部分木のどの頂点の値よりも大きく，右部分木のどの頂点の値よりも小さいことである．次の操作を行うことができる．

- Minimum (Maximum): T に含まれる値の最小値（最大値）を求める．
- Search(x): x が T に含まれるかどうか判定する．
- Insert(x): x を T に挿入する．
- Delete(x): x を T から削除する．

これらの操作は以下のように実現できる．（簡略化のため頂点とキーを同一視して記述している．）

■MINIMUM (MAXIMUM)　根から順に左（右）の子をたどっていき，葉に到達したらその値が最小値（最大値）である．

■SEARCH(x)　x を根 r と比較する．$x = r$ ならば x は T に含まれる．$x < r$ のときは左部分木に対して，$x > r$ のときは右部分木に対して同様に繰り返す．葉に到達しても一致しないならば，x は T に含まれない．

■INSERT(x)　SEARCH と同様にして x を探索する．キーが重複しないと仮定したので，x は見つからず葉 l に到達する．$x < l$ ならば x を l の左の子として追加し，$x > l$ ならば x を l の右の子として追加する．

■DELETE(x)　SEARCH と同様にして x を探索する．x が葉である場合は，それを取り除く．x が 1 個の子を持つ場合，x を取り除き，x の子を子孫とともに x の位置に移動する．x が 2 個の子を持つ場合，x の右部分木で最小の頂点を y とする（y は左の子を持たない）．x を取り除き，y のみを x の位置に移動する．もとの y に右の子がいる場合には，その子孫とともに，もとの y の位置に移動する．

2 分探索木の高さを h としたとき，各操作の最悪計算量は $O(h)$ である．高さ h はデータの挿入・削除の順序に依存する．例えば，n 個のデータを小さい順に挿入したとき，木の高さは $n-1$ である．このような場合，探索の効率は非常に悪くなる．一方，n 個のデータを一様ランダムに挿入したとき，各操作の平均計算量は $O(\log n)$ である．木の高さの平均が $O(\log n)$ であることも知られている．

データの挿入・削除の際に追加の操作を行って木のバランスをとり，木の高さを常に $O(\log n)$ に保つことで，最悪計算量を $O(\log n)$ にした**平衡木** (balanced tree) と呼ばれるデータ構造が数多く知られている．代表的なものとして，**AVL 木** (AVL tree)，**2 色木（赤黒木）**(red-black tree) などがある．

ハッシュ表

ハッシュ表 (hash table) は，順序構造を用いずに，データの挿入・探索・削除がほぼ定数時間で行えるデータ構造である．長さ m の配列 a を用意する．キー全体の集合を S として，**ハッシュ関数** (hash function) $h\colon S \to \{1, 2, \ldots, m\}$ を固定する．キー x を持つデータを $a[h(x)]$ に格納する．異なるキーに対するハッシュ値が等しくなる可能性があり，これを**衝突** (collision) という．衝突への対処法として以下のものがある．

■**チェイン法** (chaining)　データを配列 a に格納する代わりに，ハッシュ値ごとに用意した連結リストに格納する．配列 a には連結リストへのポインタを格納する．

データの個数を n として，各データに対するハッシュ値は独立かつ一様とする．このとき，各操作の平均計算量は $O(1+\alpha)$ である．ただし，$\alpha = n/m$ であり，α を**負荷率（占有率）**(load factor) という．

■**開番地法** (open addressing)　配列 a の別の場所にデータを格納する．操作するデータのキーを x とする．

データを挿入するとき，$a[h(x)]$ が空または削除済みならばその場所に格納する．$a[h(x)]$ にデータが格納されているとき，最も単純な**線形探査法** (linear probing) では，配列 $a[h(x)+1], a[h(x)+2], \ldots$ の順に調べ，空または削除済みの場所にデータを格納する．他に **2 重ハッシュ法** (double hashing) などが用いられる．

データを探索するとき，挿入の際と同様に調べるが，削除済みの場所は飛ばして次を調べる．

データを削除するとき，探索の際と同様に格納場所を探し，データを削除した場所を「削除済み」とする．

データが独立かつ一様に配列 a に格納される**一様ハッシング** (uniform hashing) を仮定する（線形探査法では成立しない）．このとき，不成功探索の平均探査回数は $1/(1-\alpha)$ 以下であり，成功探索の平均探査回数は $(1/\alpha)\log(1/(1-\alpha))$ 以下である（log は自然対数とする）．