データ構造とアルゴリズム

08 バケット，ハッシュ，ヒープ

宮本 裕一郎
miyamoto あっと sophia.ac.jp

上智大学 理工学部 情報理工学科

目次

- ここまでに，いくつかのアルゴリズムを紹介してきた．
- また，グラフ理論の用語と記号も紹介した．
- 今回は，これまでの知識を前提として，少し複雑なデータ構造であるヒープを紹介する．
- ついでにバケット，ハッシュといったデータ構造も紹介する．

バケット，ハッシュ，ヒープ

バケットとバケットソート
ハッシュ
優先キューとヒープ
ヒープソート
演習問題

- まず，簡単なデータ構造であるバケットを紹介する．
- とってが付いた運搬用の容器をバケットという．
- 日本語では大抵の場合「バケツ」とよばれる．
- データ構造としてのバケットは，直感的には，バケツのようなものを複数用意しその中にデータを入れる感じであると思えば良い．
- 何か具体例がないとイメージが湧きにくいと思うので，バケットを用いた整数の並べ替えを紹介する．

整数の並べ替え

- n 個の整数が与えられたとき，それらを昇順（小さい順）に並べ替える問題を考える．

問題 (整数の並べ替え)

入力 整数列 $X = (x_1, x_2, \ldots, x_n) \in \mathbb{Z}^n$

出力 整数列 $Y = (y_1, y_2, \ldots, y_n) \in \mathbb{Z}^n$，ただし $y_1 \leq y_2 \leq \cdots \leq y_n$ かつ X から Y への全単射が存在

例 (8 つの整数の並べ替え)

入力 (8, 4, 2, 7, 9, 9, 5, 6)

出力 (2, 4, 5, 6, 7, 8, 9, 9)

- 整数の並べ替え（問題）のアルゴリズムとしてバケットソート（bucket sorting）がある．

整数の並べ替えに対するバケットソートの直感的理解

- まず，入力整数列を全部見て，最小値 x_{min} と最大値 x_{max} を知る．
    - 入力が (8, 4, 2, 7, 9, 9, 5, 6) ならば，最小値 x_{min} は 2，最大値 x_{max} は 9 である．
- $x_{\mathrm{max}} - x_{\mathrm{min}} + 1$ 個のバケツを用意し，それぞれに x_{min}，$x_{\mathrm{min}} + 1$，$x_{\mathrm{min}} + 2, \ldots, x_{\mathrm{max}}$ の番号を付ける．
    - 入力が (8, 4, 2, 7, 9, 9, 5, 6) ならば，8 個のバケツを用意し，それぞれに $2, 3, \ldots, 9$ の番号を付ける．
- 入力整数列のそれぞれの値を，対応する番号のバケツに入れる．
- 入力整数列を全て入れ終わったら，番号順にバケツを見て，中身を取り出しつつ一列に並べる．
- 非常に単純である．

バケットソート

- バケットソートを `BacketSort` として関数っぽく以下に記す．

`BacketSort`$(x_1, x_2, \ldots, x_n)$:

Step 1 $x_{\mathrm{min}} = \min\{x_1, \ldots, x_n\}$, $x_{\mathrm{max}} = \max\{x_1, \ldots, x_n\}$ とする．

Step 2 空集合の列 $(S_{x_{\mathrm{min}}}, S_{x_{\mathrm{min}}+1}, \ldots, S_{x_{\mathrm{max}}})$ を用意する．

Step 3 それぞれの $x \in (x_1, \ldots, x_n)$ に関して，S_x に x を加える．

Step 4 $Y = ()$ とする．

Step 5 それぞれの $S \in (S_{x_{\mathrm{min}}}, \ldots, S_{x_{\mathrm{max}}})$ に関して，S の要素を Y の末尾にすべて加える，すなわち $Y = Y \circ S$ とする．

Step 6 Y を出力する．

- $L = x_{\mathrm{max}} - x_{\mathrm{min}}$ とすると，バケットソートの時間複雑度は $\mathrm{O}(n + L)$ である．（配列へのランダムアクセスを仮定している．）
- それぞれのバケット S はリストで保存すると良い．
- L が小さい場合には，バケットソートは非常に速い．
- ここでは整数の並べ替えを例に挙げているが，一般に，バケットソートは出現する要素の種類が限られている場合に有効である．

バケット，ハッシュ，ヒープ

- バケットとバケットソート
- **ハッシュ**
- 優先キューとヒープ
- ヒープソート
- 演習問題

ハッシュ

- データ構造としてのバケットの本質は「そのバケットに入っているべきものがあるかないかを即座に判定できること」にある．
- 一方で，分類するべき種類の可能性が多い場合には膨大なバケットが必要になり効率が落ちる．
  - 例えば，最大 10 文字の英単語（ただし小文字のみ使用）をバケットで分類しようとすると 26^{10} 個のバケットが必要になる．
- ハッシュ（hash）というデータ構造を用いると，
  - ほぼ要素数の入れ物だけを用意して，
  - 指定した要素があるかないかを（期待値の意味で）時間複雑度 $O(1)$ で判定できる．

ハッシュの例

- 例えば，英単語の集合 $W = \{w_1, w_2, \dots, w_n\}$ が与えられているとする．単語 $w_i \in W$ を構成する文字を先頭から順に $w_{i,1}w_{i,2}\dots$ とする．この英単語の集合 W をハッシュで保存したい場合には以下の方法が考えられる．
- 与えられた単語に対して関数 $f\colon W \to \mathbb{Z}_{\geq 0}$ を

$$f(w_i) = (\tau(w_{i,1}) + \tau(w_{i,2}) + \dots) \bmod n$$

  と定義する．ただし，$\tau(\mathtt{a}) = 1,\ \tau(\mathtt{b}) = 2,\ \dots,\ \tau(\mathtt{z}) = 26$ とする．
  - 例えば，$W = \{\mathtt{acm}, \mathtt{atm}, \mathtt{amm}\}$ ならば，
    $f(\mathtt{acm}) = (1 + 3 + 13) \mod 3 = 2$
    $f(\mathtt{atm}) = (1 + 20 + 13) \mod 3 = 1$
    $f(\mathtt{amm}) = (1 + 13 + 13) \mod 3 = 0$
    である．
- 箱を n 個用意し，$f(w) = x$ となる単語は x 番目の箱に入れることにする．
- このようにすると，よほど技巧的に集合 W を作らないかぎり，それぞれの箱にはほぼ 1 つの単語が入ることになる．

ハッシュの例の続き

- また，新たな単語 w が与えられて，それが W にあるかないかを判定したければ $f(w)$ 番目の箱だけを調べれば良い．
  - 例えば，aom が W に含まれているか否か判定したければ，$f(\mathtt{aom}) = (1 + 15 + 13) \bmod 3 = 2$ なので 2 番目の箱の中だけを調べれば良い．
- この例で定義した関数 f はハッシュ関数などとよばれる．
- ハッシュは並べ替えなどでは使えないが「その要素をすぐに見つけられるように集合を保存したい」場合には便利である．

ハッシュのまとめ

- データ構造としてのハッシュは，多くのプログラミング言語においてクラスなどとして実装されている．
- 先程の英単語の保存の例でわかる通り，「英単語のようなもの」を重複を少なくしつつ「バケット」に格納できる．
- これを利用して，添字として整数しか使えないバケットだけでなく，添字として文字列を使ったバケットのようなものも省メモリで実現できる．
- 添字が文字列のバケットは，文字列が「用語」，中身が「用語の内容」と見なすならば「辞書」のようなものである．
- よってデータ構造としてのハッシュは，プログラミング言語によっては辞書とよばれることもある．

バケット，ハッシュ，ヒープ

優先キュー

- 要素の追加，優先度が高い要素の取り出しを頻繁に行いたいという場面は多い．そこでは，**優先キュー（priority queue）**という概念が有用である．
- 優先キューに要求される仕様は
  - 要素の追加を（いつでも）高速にできる．
  - 優先度が最も高い要素の取り出しを（いつでも）高速にできる．

  である．
- 優先キューの実現として代表的なものに，**d-ヒープ（d-heap）**がある．
- d-ヒープの紹介のために，まず幾つかグラフ理論の用語を定義する．

無向グラフの閉路と森と木

定義 (閉路)

無向グラフ C の頂点集合 $V(C)$ を $\{v_1, \ldots, v_n\}$，枝集合を $\{e_1, \ldots, e_n\}$ とする．$(v_1, e_1, v_2, \ldots, v_n, e_n, v_1)$ が（無向）歩道となっているならば，グラフ C を無向閉路（undirected cycle or undirected circuit）という．

定義 (森)

無向グラフのうち，部分グラフとして閉路を含まないものを森（forest）という．

定義 (木)

森のうち，連結なものを木（tree）という．

無向グラフの path

定義 (次数)

無向グラフ G の頂点 $v \in V(G)$ を端点とする枝の集合を $\delta(v)$ で表し，その本数 $|\delta(v)|$ を次数（degree）という．

定義 (path)

木のうち，どの頂点の次数も 2 以下であるものを path という．Path の頂点のうち次数が 1 のものを，path の端点という．

定義 (部分グラフとしての path)

与えられたグラフ G の部分グラフ P が path であるとき，P を G の path であるなどという．特に path P の端点 $v, w \in V(P)$ を明示したいときには P を「v-w path」などと表記する．

定理

与えられた木 T における 2 頂点 $v, w \in V(T)$ を結ぶ v-w *path* は，どのような v, w に関しても唯一である．

木とデータ構造

- グラフとしての木は，先程の定理の特徴もあり，とてもわかりやすい．
- よって人類は古くから（意識するしないに関わらず）データの保存に木を使ってきた．
- 家系図や進化系統樹はその好例である．
    - 厳密には木ではないかもしれないが．
- コンピュータ内部のフォルダ構造（ディレクトリ構造）も同様である．
- 木を利用したデータ構造はたくさん知られているが，今回はヒープに着目する．

根付き木，親，子，先祖，子孫

定義 (根付き木，親，子，先祖，子孫)

- 木のうち，唯一の頂点が**根**（root）として特別視されたものを**根付き木**（rooted tree）という．
- 根付き木においては，隣接する頂点同士に**親子関係**があるといい，（グラフ上の距離の意味で）根からの距離が短い方を**親**（parent），そうでない方を**子**（child）という．さらに，根付き木の頂点 v と根 r を結ぶ v-r path 上の頂点を v の**先祖**（ancestor）という．そして頂点 v を先祖とする頂点 w を v の**子孫**（descendant）という[1]．
- 根付き木のうち，どの頂点も高々 d 個の子しか持たないものを d-**木**（d-tree）という．

Corollary

根付き木においては，根以外の頂点はちょうど 1 つの親を持つ．

[1] この定義では，それぞれの頂点は自分自身の先祖であり子孫でもある．

バイナリヒープ

- 優先キューの実現として代表的なものに，d-ヒープがある．
- ここでは $d = 2$ の場合であるバイナリヒープ（binary heap）を紹介する．

定義 (バイナリヒープ)

以下の条件を満たす根付き 2-木をバイナリヒープという．

- どの頂点にも優先度（あるいはキー値）が設定されている．
- 親子関係にある頂点同士の優先度を見ると，

$$\text{親の優先度} \geq \text{子の優先度}$$

となっている．

- バイナリヒープの要素数が高々 n 個のとき，最優先要素の参照は $O(1)$，最優先要素の取り出しは $O(\log n)$，要素の追加は $O(\log n)$ でできる．

配列によるバイナリヒープの実現

バイナリヒープの簡便な実現方法として配列を用いたものがある．

頂点が n 個のバイナリヒープの場合，以下の 2 条件に従って，頂点列 $(v_1, v_2, \ldots, v_n)$（配列）を生成する．

- v_1 が根である．
- v_i の親は $v_{\lfloor i/2 \rfloor}$ である．すなわち

$$v_{\lfloor i/2 \rfloor}\text{の優先度} \geq v_i\text{の優先度}$$

である．

なお，頂点数 0（$n = 0$）のものもヒープとする．

配列版バイナリヒープの操作

ヒープを $(v_1, \ldots, v_n)$ で表すこととする．まず，ヒープの大きさを返すアルゴリズムを `HeapSize` として記す．

`HeapSize`$(v_1, \ldots, v_n)$:

Step 1 n を出力して終了する．

ヒープの最優先要素を出力するアルゴリズムを `Heap1st` として記す．なお，ヒープの大きさが 0 のときには例外処理が必要となるが省略する．

`Heap1st`$(v_1, \ldots, v_n)$:

Step 1 v_1 を出力して終了する．

ヒープ $(v_1, \ldots, v_n)$ に新たに要素（頂点）v を加えるアルゴリズム Push を以下に記す．これは要素の追加そのものとヒープの整形（親の優先度 ≧子の優先度となるように）の二段階からなる．

Push$((v_1, \ldots, v_n),\ v)$:

Step 1 v を v_{n+1} とし，ヒープを $(v_1, \ldots, v_n, v_{n+1})$ とする．

Step 2 ShiftUpRecursive$(v_1, \ldots, v_n, v_{n+1}),\ n+1)$ を行い終了する．

ShiftUpRecursive$((v_1, \ldots, v_n),\ i)$:

Step 1 $i = 1$（すなわち v_i が根）である，あるいは

$$v_i \text{の優先度} \leq v_{\lfloor i/2 \rfloor} \text{の優先度}$$

ならば終了する．

Step 2 v_i と $v_{\lfloor i/2 \rfloor}$ を入れ替える．

Step 3 ShiftUpRecursive$((v_1, \ldots, v_n),\ \lfloor i/2 \rfloor)$ を行い終了する．

最優先要素の取り出す，すなわち最優先要素を出力してそれをヒープから削除するアルゴリズムを `Pop` として以下に記す．

`Pop`$((v_1, \ldots, v_n))$:

- Step 1 v_1 **を出力する．**
- Step 2 v_1 **に** v_n **を代入し，ヒープを** $(v_1, \ldots, v_{n-1})$ **とする．**
- Step 3 `ShiftDownRecursive`$((v_1, \ldots, v_n),\ 1)$ **を行い終了する．**

`ShiftDownRecursive`$((v_1, \ldots, v_n),\ i)$:

- Step 1 v_{2i} **が存在しないならば終了する．**
- Step 2 v_{2i} **が存在し，かつ，** v_{2i+1} **が存在しないならば以下を行い終了する．**
  - Step 2-1 **「v_iの優先度 $<$ v_{2i}の優先度」ならば** v_i **と** v_{2i} **を入れ替える．**
- Step 3 v_{2i} **と** v_{2i+1} **がともに存在するならば以下を行い終了する．**
  - Step 3-1 v_{2i} **と** v_{2i+1} **のうち優先度の高い方を** v_j **とする．**
  - Step 3-2 **「v_iの優先度 $<$ v_jの優先度」ならば以下を行う．**
    - Step 3-2-1 v_i **と** v_j **を入れ替える．**
    - Step 3-2-2 `ShiftDownRecursive`$((v_1, \ldots, v_n),\ j)$ **を行う．**

バイナリヒープの操作の時間複雑度

以下では，ヒープの要素数は高々 n であるとする．

- ヒープサイズの確認 `HeapSize` の時間複雑度は $\mathrm{O}(1)$ である．
- ヒープの最優先要素の参照 `Heap1st` の時間複雑度は $\mathrm{O}(1)$ である．
- ヒープに要素を 1 つ追加する操作 `Push` の時間複雑度は $\mathrm{O}(\log n)$ である．
- ヒープの最優先要素の取り出し操作 `Pop` の時間複雑度は $\mathrm{O}(\log n)$ である．

最優先要素の取り出し，要素の追加が頻繁に起こるような場面ではヒープは有用である．

バケット，ハッシュ，ヒープ

- バケットとバケットソート
- ハッシュ
- 優先キューとヒープ
- **ヒープソート**
- 演習問題

ヒープソート

ヒープは，ほとんどそのままで，並べ替え問題を解くために利用できる．以下にヒープを用いた並べ替えを HeapSort として記す．

HeapSort$(x_1, x_2, \ldots, x_n)$:

Step 1 **頂点に付加された値が小さいほど優先度が高いという空の（要素数 0 の）ヒープ H を用意する．**

Step 2 $i \in \{1, \ldots, n\}$ **に関して，値 x_i を付加した頂点 v_i をヒープ H に追加する．**

Step 3 $Y = ()$ **とする．**

Step 4 **ヒープ H の要素がなくなるまで，以下を行う．**

  Step 4-1 **ヒープ H から最優先要素を取り出し，それに付加された値を Y の末尾に加える．**

Step 5 Y **を出力して終了する．**

- **ここでは，昇順の並べ替えをするアルゴリズムを示したが降順の場合も同様にアルゴリズムを構築できる．**
- **ヒープソートの時間複雑度は $\mathrm{O}(n \log n)$ である．**

バケット，ハッシュ，ヒープ

- バケットとバケットソート
- ハッシュ
- 優先キューとヒープ
- ヒープソート
- **演習問題**

ヒープソートの練習

問題 整数列 $X = (7,4,7,2,3,5)$ を入力としてヒープソート HeapSort を行う際のヒープの変化を配列および対応する根付き木で描いてみよう．

解答例 ヒープの変化を配列で記すと以下の通りである．

要素追加過程

- $()$
- (7)
  - $(7,4)$
- $(4,7)$
- $(4,7,7)$
  - $(4,7,7,2)$
  - $(4,2,7,7)$
- $(2,4,7,7)$
  - $(2,4,7,7,3)$
- $(2,3,7,7,4)$
  - $(2,3,7,7,4,5)$
- $(2,3,5,7,4,7)$

要素削除過程

- $(2,3,5,7,4,7)$
  - $(7,3,5,7,4)$
  - $(3,7,5,7,4)$
- $(3,4,5,7,7)$
  - $(7,4,5,7)$
- $(4,7,5,7)$
  - $(7,7,5)$
- $(5,7,7)$
- $(7,7)$
- (7)
- $()$

レベルの低い項目はヒープの整形途中に対応している．対応する根付き 2 分木による表現は省略する．