データ構造と アルゴリズム I

第6回

前半まとめテスト内容


ソーティングと順序統計量

入力: n 個の数の列 $\langle a_1, a_2, ..., a_n \rangle$

出力: $a'_1 \le a'_2 \le \ldots \le a'_n$ であるような入力列の置換 $\langle a'_1, a'_2, ..., a'_n \rangle$

- 実際には，入力は数だけでなくデータの集合 (レコード) の場合が多い
- レコード = (キー, 付属データ)
- キーの順序にレコードをソートする
- レコード自身でなく，そのポインタを並び替える

順序統計量

- n 個の数の集合に対する i 番目の順序統計量 = その集合で i 番目に小さい数
- 入力をソートしてからi 番目を出力⇒$\Theta(n \lg n)$ 時間
- 実は，ソートなしで $O(n)$ 時間で求めることができる
  - ソートの後に説明

今までのソーティングアルゴリズム

1. 挿入ソート: $O(n^2)$ 時間だが，入力サイズが小さいときには高速．in-place (入力の配列以外のメモリが不要)
2. マージソート: $\Theta(n \lg n)$ 時間だが，実行には一時的な配列が必要

新しいソーティングアルゴリズム

1. ヒープソート: $O(n \lg n)$ 時間, in-place.
2. クイックソート: 最悪 $O(n^2)$ 時間だが平均実行時間は$\Theta(n \lg n)$．実用上は高速．in-place.

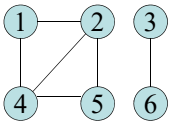
ヒープソート

- O(n lg n) 時間アルゴリズム, in-place
- ヒープ (heap) と呼ばれるデータ構造を用いる
- ヒープはプライオリティキュー (priority queue) を効率よく実現する

210

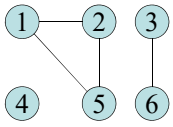
B.4 グラフ

- 無向グラフ (undirected graph)
- $G = (V, E)$
  - V: G の頂点集合 (vertex set)
  - E: G の辺集合 (edge set)



211

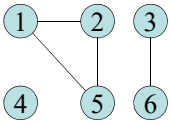
- 辺集合 E は頂点の非順序 (unordered) 集合
- 1つの辺は (u,v) で表現される
  - (u,v) と (v,u) は同一の辺を意味する



212

用語の定義

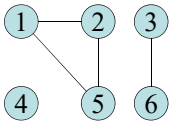
- (u,v) が $G = (V, E)$ の辺であるとき，頂点 v は頂点 u に隣接 (adjacent) しているという.
  - 無向グラフでは隣接関係は対称的
- 頂点 v の次数 (degree)
  = v に接続している辺の数



213

経路 (path)

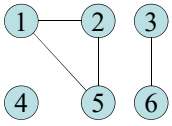
- $G = (V, E)$ の頂点列 $< v_0, v_1, v_2,..., v_k >$ が頂点 v_0 から v_k までの長さ k の経路とは
  $(v_{i-1}, v_i) \in E \quad (i=1,2,...,k)$
- 経路の長さ＝経路上の辺の数
- u から v への経路 p が存在するとき，v は経路 p を経由して u から到達可能 (reachable) という



214

閉路 (cycle)

- 経路 $< v_0, v_1, v_2,..., v_k >$ が閉路であるとは
  - $v_0 = v_k$
  - 少なくとも1つの辺を含む



215

グラフに関する例題 (ラムゼーの定理)

- 人間同士の関係を，知り合いかそうでないかに分類する．
- AがBの知り合いなら，BはAの知り合いであることを仮定する．
- 任意に選ばれた6人の人に関して，以下のいずれかが必ず成立する
  - 全員が知り合い同士の3人がいる（6人からうまく3人を選ぶと，お互いに知り合い）
  - 全員が知り合いでない3人がいる（6人からうまく3人を選ぶと，だれも知り合いでない）

216

ラムゼーの定理

- グラフの用語で言い換えると，
  - 6個の頂点を持つ任意の無向グラフに関して，次のどちらかが必ず成立する
    - 互いに辺で結ばれた3つの頂点が存在 (完全グラフもしくはクリークと呼ばれる)
    - お互いの間に辺が存在しない3つの頂点が存在 (独立集合と呼ばれる)

217

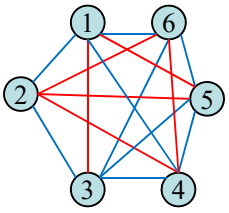
課題

- ラムゼーの定理を証明せよ
- 小学生でも分かる問題の記述：
  - 赤と青の色鉛筆を使って，六角形を書いて，さらにすべての頂点が結ばれるように線を引きましょう．線を引くときには，赤，青の鉛筆のどちらを使っても構いません．この場合，赤の線だけの三角形，および青の線だけの三角形が全く作られないようにすることはできません．この理由はなぜでしょうか?

218

ラムゼーの定理の証明

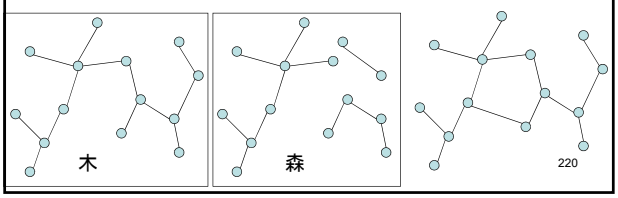
- 知り合い同士を青いエッジ，知らない同士を赤いエッジで結ぶ．
- あるノードからは，ちょうど5本のエッジが出ている．
- かつ，青か赤のどちらかは少なくとも3本存在．
- 青が3本以上の場合，その先のノードn1, n2, n3間に，少なくとも1つ青があれば，青のクリークが存在
- そうでなければ，n1, n2, n3が独立集合 (互いに知り合い同士でない)
- 赤が3本の場合も同様．



219

B.5 木

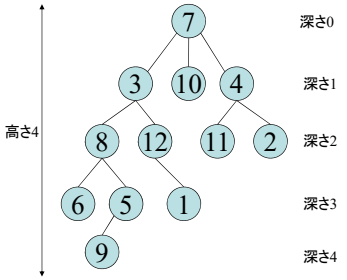
- 木 (tree): 閉路を持たない連結無向グラフ
- 森 (forest): 閉路を持たない無向グラフ (連結でなくてもいい)



220

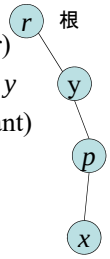
根付き木

- 根付き木 (rooted tree)：唯一の他と区別される頂点 (根, root) を持つ木



221

先祖，子孫



- 根付き木 T 上の節点 x の先祖 (ancestor)
⇔根 r から x に至る経路上の任意の節点 y
- y が x の先祖⇔ x が y の子孫 (descendant)
- x を根とする部分木 (subtree rooted at x)
⇔x を根とし，x の子孫からなる部分木
- p は x の親 (parent)，x は p の子 (child)
- 同一の親を持つ2節点：兄弟 (sibling)
- 子を持たない節点：外部節点(external node)
または葉 (leaf)
- 葉でない節点：内部節点 (internal node)

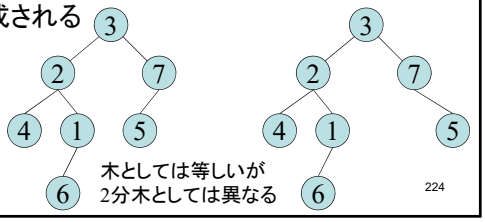
クイズ：この人は誰?

- 僕には兄弟姉妹はいないけど，この人の父親は，僕のお父さんの子供だ．
- この人は誰?
- 僕をx, xの父親をpx, この人をy, この人の父親をpyとすると，py=child(px)=x.
- よってparent(y)=x. この人は僕の子供．

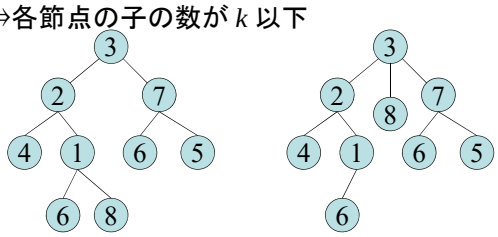
2分木 (binary tree)

定義：木 T が2分木とは

1. T は節点を全く持っていない (空)，または
2. T は根，左部分木 (left subtree) と呼ばれる 2 分木，右部分木 (right subtree) と呼ばれる 2分木の3つの節点集合 (共通要素を持たない) から構成される



- 全2分木 (full binary tree)
⇔各節点が葉または次数（子供の数）が2である木
- k 分木 (k-ary tree)
⇔各節点の子の数が k 以下

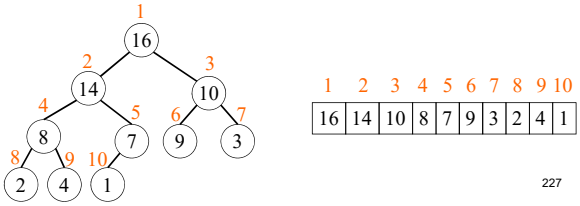


木の重要性

- 大量のデータの整理には ほぼ必ず木構造が用いられる (図書の分類，住所，yahoo!カテゴリ, etc.)
- 木の深さをdとすると，木の葉節点の数は $O(2^d)$
- うまく木を使えば，大量のデータを対数オーダで処理できる!

6.1 ヒープ

- ヒープ：完全2分木とみなせる配列
- 木の各節点は配列の要素に対応
- 木は最下位レベル以外の全てのレベルの点が完全に詰まっている
- 最下位のレベルは左から順に詰まっている

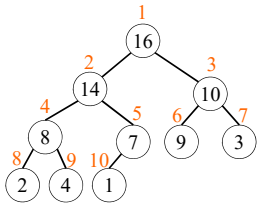


ヒープを表すクラス

```
class HEAP {
public:
   int length;          // 配列 A に格納できる最大要素数
   int heap_size;       // ヒープに格納されている要素の数
   data *A;             // 要素を格納する配列へのポインタ

};
```


- length: 配列 A に格納できる最大要素数
- heap_size: 格納されているヒープの要素数
- heap_size ≤ length
- 木の根: A[1]
- 節点の添え字が i のとき
  - 親 PARENT(i) = $\lfloor i / 2 \rfloor$
  - 左の子 LEFT(i) = $2i$
  - 右の子 RIGHT(i) = $2i + 1$
- 木の高さは $\Theta(\lg n)$



ヒープ条件 (Heap Property)

- 根以外の任意の節点 i に対して
  $A[\text{PARENT}(i)] \geq A[i]$
- つまり，節点の値はその親の値以下
- ヒープの最大要素は根に格納される

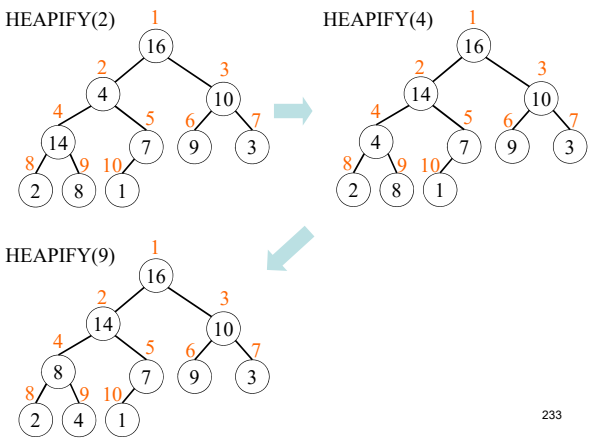
ヒープの操作

- HEAPIFY: ヒープ条件を保持する (根節点が子供より大きいとは限らないが，両側の部分木ではヒープ条件が満たされていることを仮定). $O(\lg n)$
- BUILD_HEAP: 入力の配列からヒープを構成する. 線形時間.
- EXTRACT_MAX: ヒープの最大値を取り除き，残りがヒープ条件を満たすようにする. $O(\lg n)$ 時間.
- HEAPSORT: 配列をソートする. $O(n \lg n)$ 時間.
  - BUILD_HEAPとEXTRACT_MAXから構成される.
- INSERT: ヒープに値を追加する. $O(\lg n)$ 時間.

6.2 ヒープ条件の保持

- HEAPIFY(i): クラスヒープのメンバ関数, $A[i]$ を根とする部分木がヒープになるようにする. ただし LEFT(i) とRIGHT(i) を根とする2分木はヒープと仮定.

```
void HEAPIFY(int i) {
  int l, r, largest;
  data tmp;
  l = LEFT(i);   r = RIGHT(i);
  if (l <= heap_size && A[l] > A[i]) largest = l; // A[i] と左の子で
  else  largest = i;                              //大きい方をlargestに
  if (r <= heap_size && A[r] > A[largest])        // 右の子の方が大きい
       largest = r;
  if (largest != i) {
    tmp = A[i];  A[i] = A[largest];  A[largest] = tmp;
                                     // A[i]を子供と入れ替える
    HEAPIFY(largest);}}
```

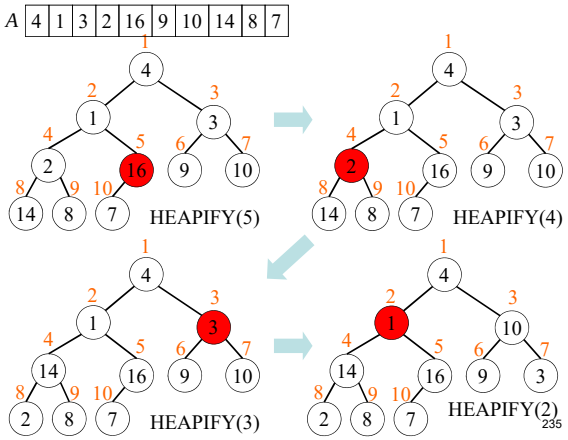



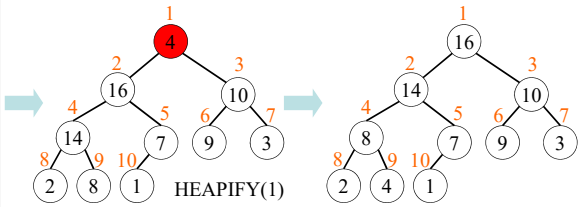
6.3 ヒープの構成

- HEAPIFYでは左右の部分木がヒープである必要がある
- 全体をヒープにするには，2分木の葉の方からヒープにしていけばいい

```
void BUILD_HEAP(int n, data D[])
{
 int i;
 heap_size = n;
 A=D;

 for (i = n/2; i >= 1; i--) {
  HEAPIFY(i);
 }
}
```

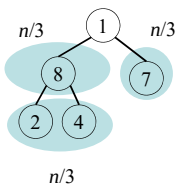


課題

- 配列 A=<1,7,3,6,5,2,4>に対するBUILD_HEAPの動作を示せ

HEAPIFYの実行時間

- 節点 i を根とする，サイズ n の部分木に対するHEAPIFYの実行時間 $T(n)$
  - 部分木のサイズは $2n/3$ 以下
  - $T(n) \leq T(2n/3) + \Theta(1)$
  - $T(n) = O(\lg n)$
- 高さ h の節点におけるHEAPIFYの実行時間は $O(h)$



BUILD_HEAPの計算量の解析

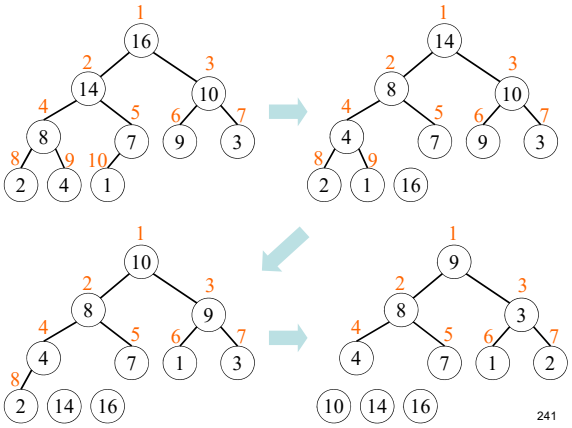
- $O(\lg n)$ 時間のHEAPIFYが $O(n)$ 回
  ⇒$O(n \lg n)$時間　(注: これはタイトではない)
- $O(n)$ が示せる.

6.4 ヒープソート

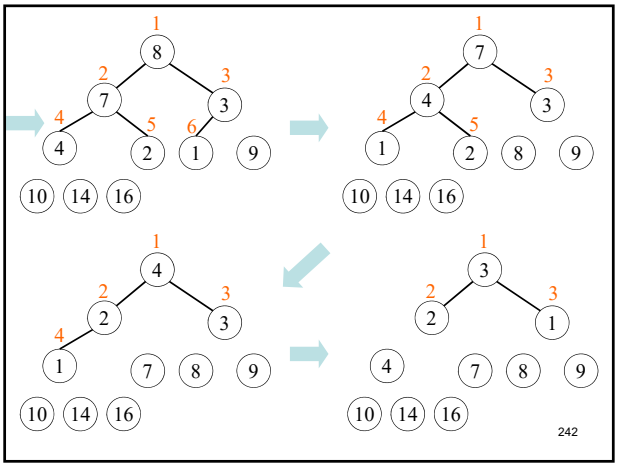
- まずヒープを作る
- すると最大要素が $A[1]$ に入る
- $A[1]$と$A[n]$を交換すると，最大要素が$A[n]$に入る
- ヒープのサイズを1つ減らしてヒープを維持する

```
void HEAPSORT(int n, data D[])
{int i;
 data tmp;
 BUILD_HEAP(n,D);
 for (i = n; i >= 2; i--) {
  tmp = A[1];  A[1] = A[i];  A[i] = tmp;
   // 根と最後の要素を交換
  heap_size = heap_size - 1;
  HEAPIFY(1);
 }}
```

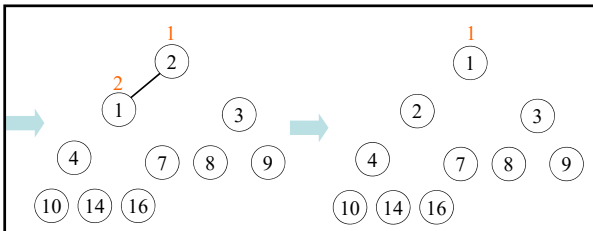
240



241



242



243

課題

- BUILD_HEAPで配列<7,6,4,1,5,2,3>が得られた後のHEAP_SORTの動作を示せ

244

計算量

- BUILD_HEAP: O(n) 時間
- HEAPIFY: 合計O(n lg n)

全体で O(n lg n) 時間

245

6.5 プライオリティキュー

- 要素の集合 *S* を保持するためのデータ構造
- 各要素はキーと呼ばれる値を持つ
- 次の操作をサポートする
  - INSERT(*S*,*x*): *S* に要素 *x* を追加する
  - MAXIMUM(*S*): 最大のキーを持つ *S* の要素を返す
  - EXTRACT_MAX(*S*): 最大のキーを持つ *S* の要素を削除し，その値を返す

246

応用：計算機のジョブ割り当て

- 実行中のジョブと優先順位をプライオリティキューに保持
- ジョブが終了または割り込み発生時に，一時中断しているジョブの中から最大の優先順位のジョブを選び実行 (EXTRACT-MAX)
- 新しいジョブはプライオリティキューに挿入される (INSERT)

247

ヒープによるプライオリティキューの実現

- EXTRACT_MAX(): *A*[1] を返してHEAPIFY

```
data EXTRACT_MAX()  // O(lg n) 時間
{
 data MAX;
 if (heap_size < 1) {
  cout << "ERROR ヒープのアンダーフロー" << endl;
  exit(1);
 }
 MAX = A[1];
 A[1] = A[heap_size];
 heap_size = heap_size - 1;
 HEAPIFY(1);
 return MAX;
}
```

248

```
void INSERT(data key)  // O(lg n) 時間
{
 int i;

 heap_size = heap_size + 1;
 if (heap_size > length) {
  cout << "ERROR ヒープのオーバーフロー" << endl;
  exit(1);
 }

 i = heap_size;
 while (i > 1 && A[PARENT(i)] < key) {
  A[i] = A[PARENT(i)];
  i = PARENT(i);
 }
 A[i] = key;
}
```

249

課題

- 4, 1, 2, 13, 9, 10の順に，これらの優先度を持つジョブが到達した後のヒープの状態を示せ
- 優先度の高いジョブが二つ処理された後の状態を示せ

250