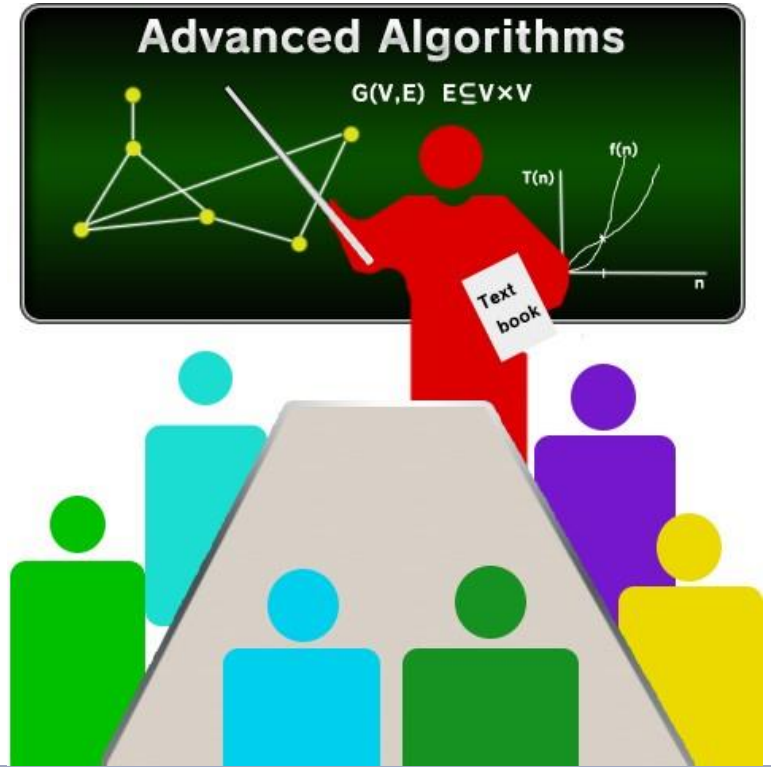


アルゴリズム特論 [AA201X]
Advanced Algorithms

Lecture02. Priority Queue and Heap

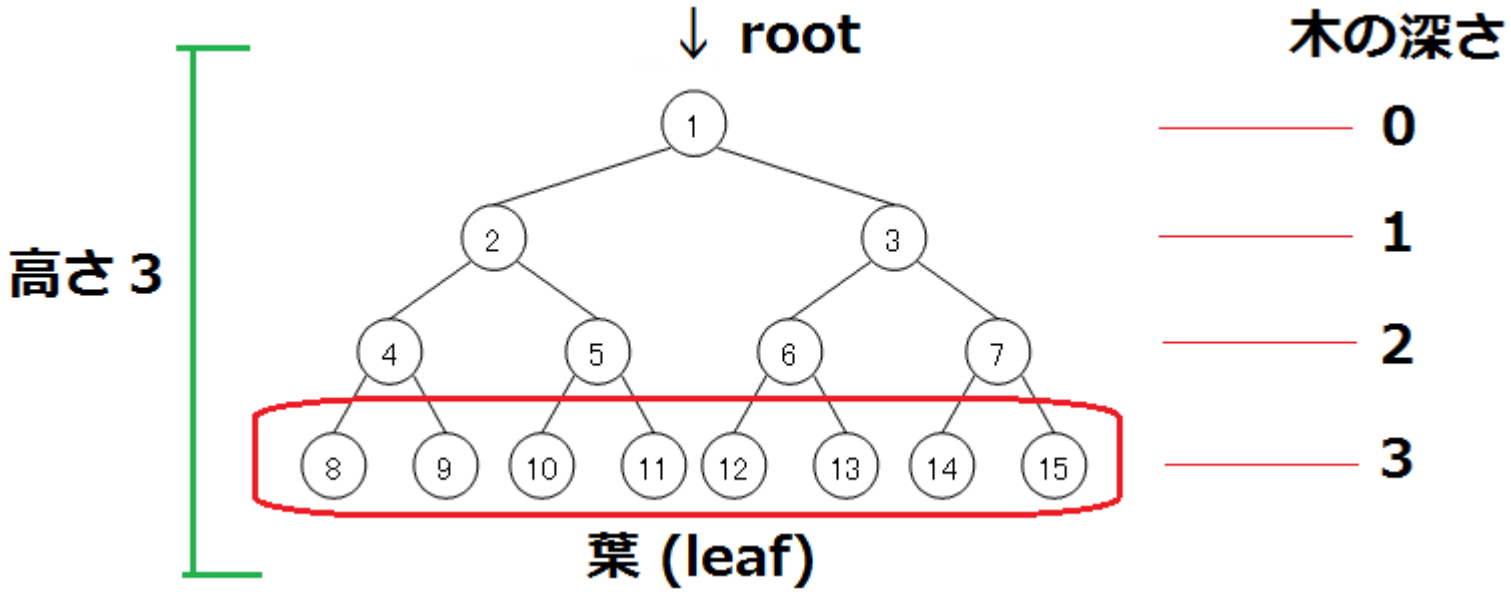
Exercise 02 のために

- 二分木
- 順序関係
- ヒープ構造、構築・維持

Binary Tree (二分木)

- 頂点(節点,vertex,node) と辺(edge)で構成（グラフの一種）
- 1つの頂点（親）に最大2個の頂点（子）が接続する

- 頂点のうち、特に木の一番深いところにあるものを葉という
- 葉以外の頂点は内部節点(internal node) と呼ばれる
- 葉を除いた全ての頂点が2個の子を持つ木のことを完全二分木(Complete BinaryTree) という
- 深さ d (根は0として数える) のところにある頂点の数は、高々 2^d 個
- 高さが h のとき、木に含まれる頂点の総数 N は　$2^h+1 \leqq N \leqq 2^{h+1}-1$



高さ h の木と頂点の総数 N の関係

- 深さdあたりに、2^d 個あるから 初項1 公比2の等比数列の和
- ただし、木の深さ、高さは 0から数えていることに注意

$$N_{max} = \sum_{d=0}^{h} 2^d = \frac{a(1-r^{h+1})}{1-r} = \frac{1-2^{h+1}}{1-2} = 2^{h+1} - 1$$

$$N_{min} = \sum_{d=0}^{h-1} 2^d + 1 = \frac{a(1-r^{h})}{1-r} + 1 = \frac{1-2^{h}}{1-2} + 1 = 2^h - 1 + 1 = 2^h$$

- 両辺のlog (底 2)をとると $h = \log N$
- 木の高さは整数であるから $h = \lfloor \log N \rfloor$（小数部分切り捨て）

- 今日の講義で O(log n)をたくさん見かけた？
  ⇒ 時間計算量が木の高さに依存する処理（アルゴリズム）

Heap Structure （ヒープ構造）

- 二分木にて、各頂点の親や子との間で、
  定義された順序関係（ルール）が必ず成立しているもの

  ・値が大きい（それ以上）
  ・値が小さい（それ以下）
  ・アルファベット順

- ヒープ構造を配列[1 : n]を用いて実装する
- 配列で表現した完全二分木上の親子
- A[n] の親ノード （木の上方を見る）⇒ A[n/2]
- A[n] の子ノード （木の下方を見る）⇒ A[2*n] と A[2*n+1]

順序関係

- $\geqq$, $\leqq$ といった、
  以下の３つを全て満たす二項関係

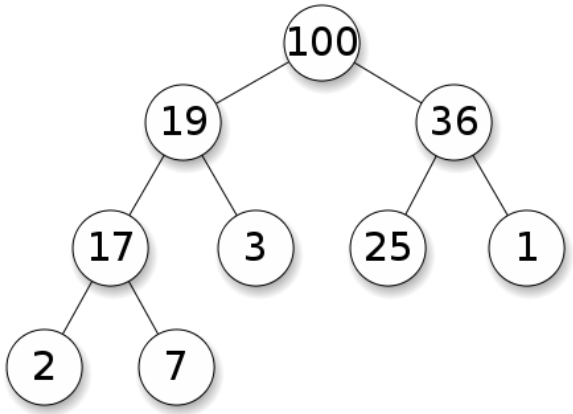
- 反射律　　$a \geqq a$
- 反対称律　　$a \geqq b,\ b \geqq a \Rightarrow a = b$

  （ただの > や < を含む不等式では、両辺を入れ替えると
  絶対に成立しない。$\geqq$や$\leqq$の時は、＝のときだけは成り立つ）

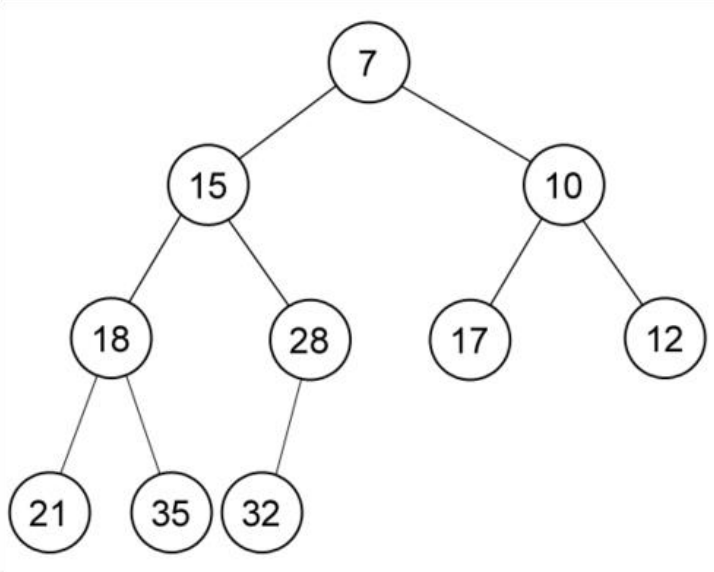
- 推移律　　$a \geqq b,\ b \geqq c \Rightarrow a \geqq c$

ヒープの例　（ソートされているわけではない。親と子の関係にのみ注目せよ！）

- Maximum heap　順序関係：≧
- n = 9, 木の高さ 3
- root が最大値 100 を持つ



- Minimum heap　順序関係：≦
- n = 10, 木の高さ 3
- root が最小値 7 を持つ



どうやってヒープを構成・維持するか？ upheap()関数 で実施

- ある１要素とその親（または子）がヒープ条件を満たすかを調べる
  ⇒ 満たしていないならば、満たすように修正する

- 自分(k)と親(k/2) の大小関係を調べる
- 厳密には、自分(k)の値を一時保存用の変数にしまっておき、
  root(根) にたどりつくまで木の上部へ登っていきながら
  自分が挿入されるべき本当の場所を探す

- ヒープ条件を満たさない場合、その
  合わない親ノードを下へ降ろして進む

逆に、何もしなくてもヒープが成立していたのなら、元の場所 k に一時保存した値をそっと戻すのみとなる。

```
upheap(int A[], int k){
    int v;
    v = A[k];
    while (k > 1 && A[k/2] <= v){
        A[k] = A[k/2];
        k = k/2;
    }
    A[k] = v;
}
```

Maximum heap の場合

どうやってヒープを構成・維持するか？ downheap()関数 で実施

- 自分(k)と子(2k または 2k+1) の大小関係を調べる
- 厳密には、自分(k)の値を一時保存用の変数にしまっておき、leaf (葉) にたどりつくまで木の下部へ降りていきながら自分が挿入されるべき本当の場所を探す

- ヒープ条件を満たさない場合、その合わない子ノードを上に上げて進む

※子ノードを見る場合、基本的には自分の子は２つある。どちらと比較するかは、定義された順序関係による。
Maximum: より大きい値の子を選ぶ
Minimum: より小さい値の子を選ぶ

```
downheap(int A[ ], int k, int n){
  int j, v;
  v = A[k];
  while (k <= n/2) {
    j = k+k;
    if (j<n && A[j]<A[j+1])  j++;
    if (v >= A[j])  break;
    A[k] = A[j];
    k = j;
  }
  A[k] = v;
}
```

Maximum heap の場合

ヒープ構造上の様々な操作

- 基本的には、upheap()とdownheap()が理解できれば問題ない
- ただし、 upheap() や downheap() だけでは何もできない
- それらを使ってヒープを操作することを理解する必要がある

- insert()　　既にあるヒープの末尾に１個の値を追加
- delete_root()　既にあるヒープの先頭(root)の値を削除（取り出す）
- replace()　　既にあるヒープの先頭(root)の値を別の値で置換

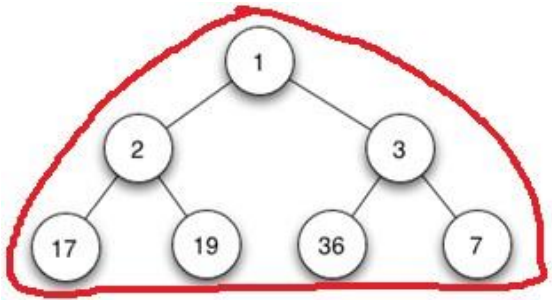
- construct1()　ヒープかどうか不明な配列全体をupheapでヒープ化
- construct2()　ヒープかどうか不明な配列全体をdownheapでヒープ化

- heapsort()　　ヒープソート （ヒープの性質を利用したソート）

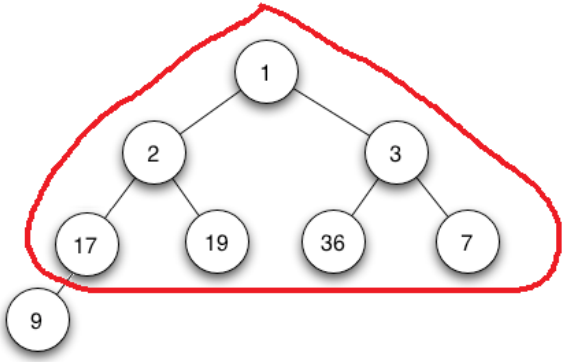
ヒープを壊したら修復せよ

- ヒープ構造になっている配列に対して、適当に値を挿入したり、削除したりすると、全体として配列を見たときにヒープが壊れてしまっている

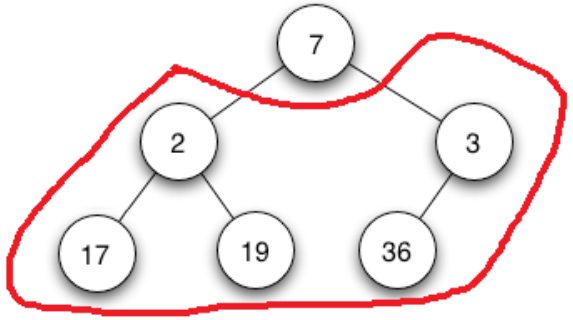
元のminimum heap



- 末尾に９を挿入



根の１を削除（して末尾を根に移す）



- どこまでがヒープのまま残っているかを意識して、最短でヒープに戻せる手段（upheapか？downheapか？）を選択して、ヒープを修復せよ。