ヒープソート

計算量

- アルゴリズムの性能を評価するのに「計算量」という概念を用いる
- 計算量とは:
  - そのアルゴリズムが，おおよそどれだけのステップ数で実行可能かを表す
  - O(n) などのO記号で表す（読み方は，オーダ n）
  - ある関数 g(n)が O(f(n))であるとは，次のような条件を満たすときに成立

$$g(n) = O(f(n)) \Leftrightarrow (\exists c>0)(\exists n_0 \geqq 0)(\forall n \geqq n_0)\ g(n) \leqq c \cdot f(n)$$

ヒープソート

- 計算量
  - O(n log n)の計算量を持つソートアルゴリズム
  - ヒープソートはどんなデータに対してもO(n log n)であることが保証されている
- 基本的な考え方
  - 選択法の改良

n個の値が入った配列 a[n] のソート

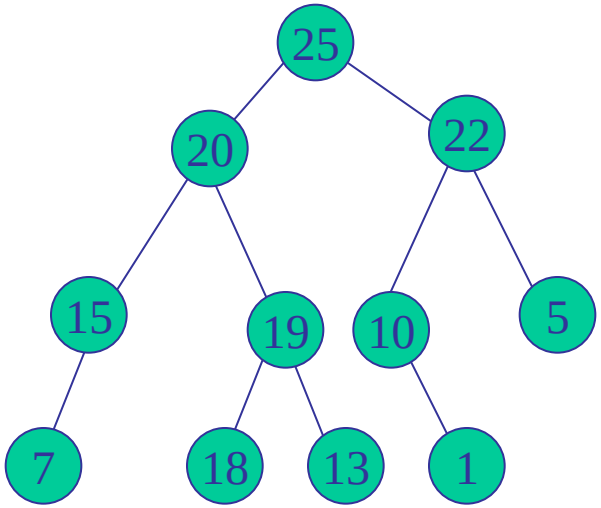
```
for ( i = n - 1; i > 0; i--) {
    a[0],...,a[i]の中で最大値を求め，a[p]とする;
    a[i]と a[p]を入れ替える;
}
```

ヒープソートの特徴

- 選択法
  - 各ステップで，最大値を求める操作を繰り返す
  - 途中で得られる情報を捨てる
    → 残りのデータの大小関係についての情報を捨てる
- ヒープソート
  - 各ステップで，最大値を求める操作を繰り返す
  - 途中で得られる情報をデータ構造に蓄える
    → 残りのデータの大小関係の情報をデータ構造の中に蓄えて，少ない手間で最大値を求める

部分順序付き木

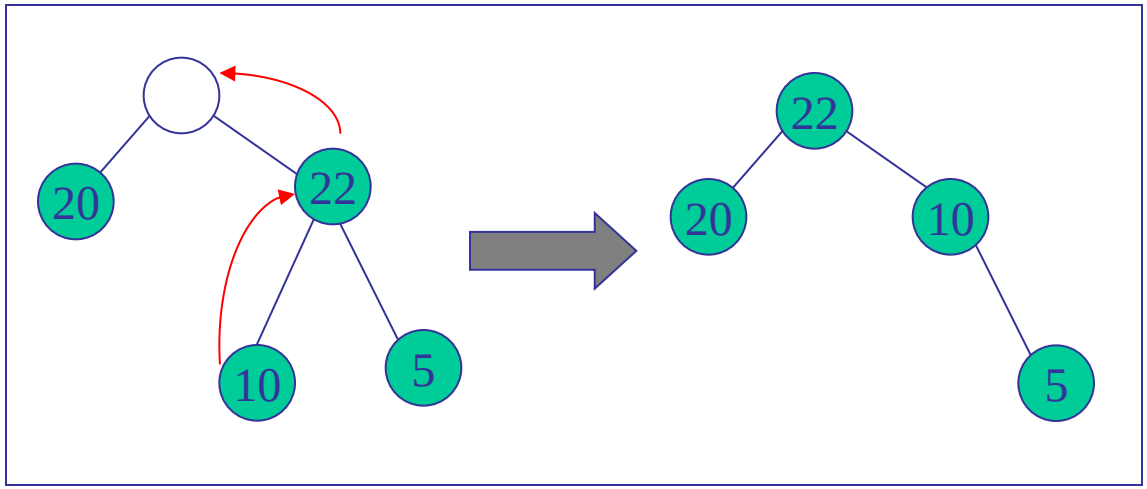
- 2分木の構造
- 最大値を根ノードに置く
- 各ノードの値が，その下の左右2つの子供の値よりも大きくなるようにした木（子供同士どちらが大きいかは問わない）



図：部分順序付き木の例

部分順序付き木での最大値の取り出し

- 常に，木の根にある値が最大値となる
- 最大値を求めるだけなら，O(1)の計算量
- 最大値を取り除いた後に，再び部分順序付き木の条件を満たすように木を作り直すことが必要



図：部分順序付き木の組み替え

1. 取り除いた頂点の場所に左右の子供のうち値の大きい方を移す．
2. これを木の一番下の段に達するまで繰り返す

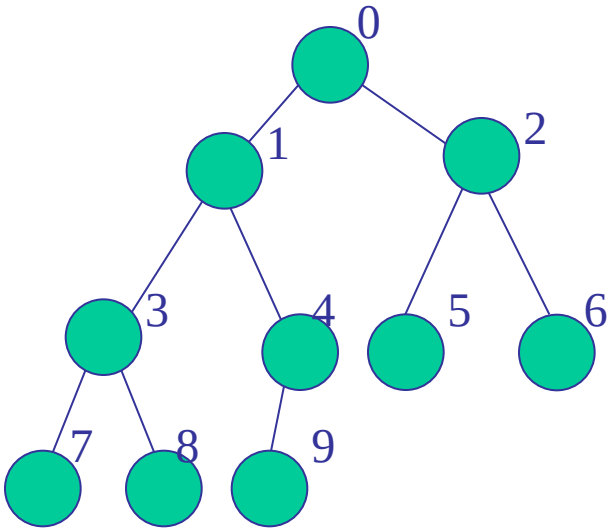
ヒープソートで用いるデータ構造

- 部分順序付き木を配列上に表現

  具体的なデータ構造の例

  – 配列の先頭 a[0]に木の根を置く

  – それ以外は，頂点 a[i] の左の子供を a[2*i+1] に 右の子供を a[2*i+2] に置く



0	1	2	3	4	5	6	7	8	9

親子子
親　子子
親　子子
親　子子
親　子

ヒープの各頂点の添字と配列の対応（n = 10 の場合）

ヒープ

- 2*i+1 や 2*i+2 が配列の要素数を越えている場合は子がないものと見なす
- 配列を隙間なく使う
- ある頂点からその親に行くのも，子に行くのも添字の計算だけでできる
- 配列上に表現した部分順序付き木をヒープと呼ぶ

ヒープの特徴

- 完全二分木の一部のみを表現できる
  - 一般の「木」を表すことはできない
  - 葉までの深さが k または k+1のいずれかで，しかも深さ k+1の葉がすべて左側によせてあるもののみ表現できる

ヒープでの最大値の取り出し

1. 最初に，a[n - 1] をa[0] に移動
   - 取り除いた頂点の代わりに子供を持ってくるだけでは，完全2分木の形がくずれる
   - 最初に，配列の最後の要素 a[n - 1] をa[0] に移動して，n を1減らす

   → これで要素数が1減った完全二分木ができる
2. 次に，部分順序木の木の条件を満たすように木の作り替えを行う
   - a[n - 1] を a[0] に移動したときに，部分順序付き木の条件が満たされていないのは，a[0] とその子供の間だけ

   → 1点 a[k] とその子供の間でだけ条件がくずれているヒープを正しく作り直すアルゴリズム： downheap アルゴリズム(入力として k=1, r=n を与える)

DownHeapアルゴリズム

- 入力
  - k　ヒープの条件がくずれている位置
  - r　現在のヒープの要素数 - 1
  - a　データを格納している配列へのポインタ
- 計算量
  - 木の高さが O(*log* n), 交換一回当たりの手間が O(1)なので, 全体の計算量は O(*log* n)

DownHeapアルゴリズム手順

1. j = 2k + 1とする.
2. j ≦r ならば

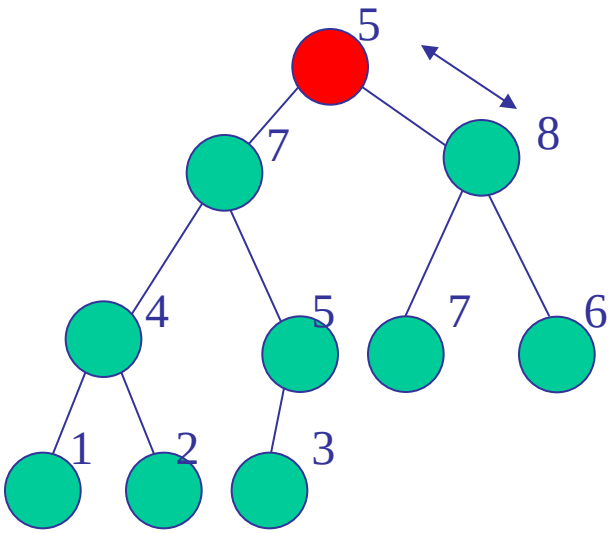
(a)j≠r ならば

a[j] と a[j + 1]を比較して大きい方を j とする.

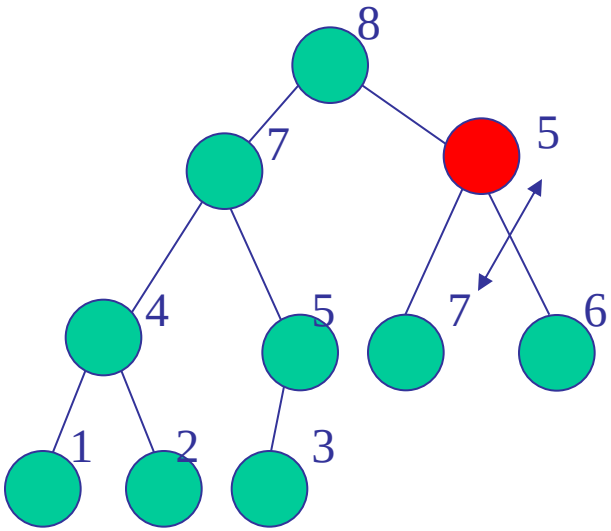
(b) i. もし a[k] が a[j] 以上ならば終了.

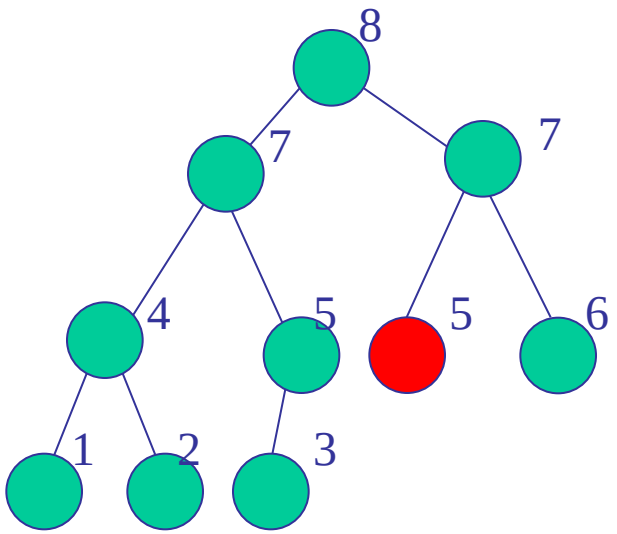
ii.そうでなければ, a[k]と a[j]の値を入れ替え,

その後 k = j, j = 2k +1 として2.へ



左右の子の大きいほうと交換する。このいずれも自分より小さいか、ヒープの底までたどり着いたら終了。





HeapMainアルゴリズムト

- DownHeap を用いて，ヒープソートを行うアルゴリズム

  1. i = n - 1とする.

  2. i > 0ならば,

     (a) a[0]とa[i]の値を入れ替える.

     (b) i = i - 1とする.

     (c) k = 0, r = i としてDownHeapを呼び出す.

     (d) 2. へ

- ヒープから最大値を取り出して，ヒープの大きさを1減らして，DownHeapを用いてヒープを再構築する，という操作を繰り返す

InitializeHeapアルゴリズム(1/3)

- 与えられたデータからヒープのデータ構造を作るアルゴリズム

  - i = [n/2] -1 とする.
  - i ≧0ならば
    - k = i, r = n-1 として DownHeapを呼び出す
    - i = i - 1とする
    - 2. へ

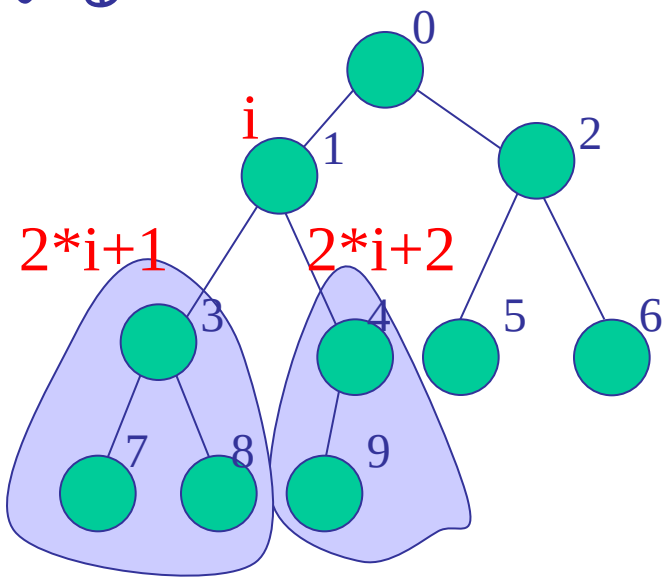
InitializeHeapアルゴリズム(2/3)

- 木を下の方から積み上げていってヒープを完成させるという方法
- 配列の初期状態
  - a[n/2], a[n/2 + 1], ... ,a[n-1]は，子供を持たない単独の頂点
  - それぞれ単独では，部分付き順序木の条件を満足

InitializeHeapアルゴリズム(3/3)

- 初期状態を出発点として，全体として1つの木を作る
  - 木を2つずつ組み合わせる操作を続けて，全体として1つの木にを作る(ヒープが完成)
  - 2. で a[i]を根とする部分木をヒープの条件を満たすように作り替える
  - この時，a[2*i + 1]とa[2*i + 2]をそれぞれ根とする木はすでに部分ヒープになっている

図:ヒープの初期化操作



ヒープソートの実現

- ソートすべきデータの，配列への取り込み
- InitializeHeapにより，ヒープを構成
- 構成したヒープに，HeapMainを適用

ヒープを作る操作の計算量

- DownHeapで調べる段数:
  - iが [n/2]～n の範囲では0段
  - [n/4] ～ [n/4]の範囲では1段
  - [n/8] ～ [n/4]の範囲では2段
  - 全体では

$$0*\frac{n}{2}+1*\frac{n}{4}+2*\frac{n}{8}+\cdots+(\log n-1)*1$$

  - k = [$\log$ n]とするとこの式は次の式の値を越えない

$$1*2^{k-2}+2*2^{k-3}+\cdots+(k-2)*2+(k-1)*1 = 2^k-(k+1)$$

  - k =O($\log$ n)なので，この式のオーダはO(n)

ヒープソートの計算量

- ヒープを作る操作(InitializeHeap)の計算量
  - O(n*log* n)
- 最大値を取り出す操作(HeapMain)の計算量
  - O(n*log* n)
- ヒープソート全体の計算量
  - これらの和： O(n*log* n)

サンプルプログラム

```
/*              0
        1               2
     3    4          5      6
   7 8  9 10  11 12  13 14
*/
#include<stdio.h>
FILE *infile, *outfile;
int tree[10000];
main()
{
    int i, n, in[20], out[20];
    printf("Input InFilename :");       /*入力データのファイル名を入力*/
    scanf("%s", in);
    if ((infile=fopen(in, "r")) == NULL) {
        printf("can't open file %s¥n", in);
        exit();
    }
    printf("Input OutFilename :");      /*出力データのファイル名を入力*/
    scanf("%s", out);
    if ((outfile=fopen(out,"w")) == NULL) {
        printf("can't open file %s¥n", out);
        exit();
    }
```

```
    n=0;
    while(fscanf(infile,"%d", &(tree[n])) != EOF) {
        n++;
    }
    InitializeHeap(n);
    HeapMain(n);
    for (i=0; i<n; i++) {
        fprintf(outfile,"%d¥n", tree[i]);
    }
    fclose(outfile);
    fclose(infile);
}

HeapMain(int n)
{           /*ヒープソートを行う*/
    int i, k, r, tmp;
    i=n-1;
    while (i>=0) {
        tmp = tree[0]; tree[0] = tree[i]; tree[i] = tmp;
        i--;
        k = 0; r = i;
        DownHeap(k, r);
    }
}
```

```
InitializeHeap(int n)
{           /*ヒープを構成する*/
    int i, k, r;
    for (i = n/2-1; i >= 0; i--){
        k=i; r=n-1;
        DownHeap(k,r);
    }
}

DownHeap(int k, int r)
{           /*ヒープを正しく作り直す*/
    int j, tmp;
    j = 2*k+1;
    while (j <= r){
        if ((j!=r) & (tree[j+1]>tree[j]))j=j+1;
        if (tree[k] >= tree[j]) {
            break;
        }
        else{
            tmp=tree[k]; tree[k]=tree[j]; tree[j]=tmp;
            tmp=k; k=j; j=2*tmp+1;
        }
    }
}
```