データ構造とアルゴリズム
（第11回）

モバイルコンピューティング研究室
柴田史久



1

本日の講義内容

- 整列（４）
  - ヒープソートの原理
  - 配列によるヒープ
  - ヒープソートの実装

2

教科書 第16章（pp.356～378）

ヒープソート

3

ヒープソートの原理(1) 探索を利用した整列

- 探索を利用した整列
- 手順
  - 探索木を生成
  - 配列のすべての要素を探索木に挿入:insert
  - 小さいものから順に探索木から取り出す:deleteMin

```
static void sort(int[] a)
{
  int n = a.length ;  // 配列の要素数
  SomeSearchTree tree = new SomeSearchTree() ;  // 探索木を準備
  for (int i = 0 ; i < n ; i++) {// 配列の要素を探索木に挿入
    tree.insert(a[i]) ;
  }
  for (int i = 0 ; i < n ; i++) {// 小さいものから取り出して配列へ戻す
    a[i] = tree.deleteMin() ;
  }
}
```

4

ヒープソートの原理(2) 必要な操作

- 探索木に対して2つの操作
  - 要素を挿入（insert）
  - 最小のキーを持つデータを取り出す（deleteMin）
- insertで n 個の要素を登録
- deleteMinでキーの小さいものから取り出す
  - deleteMinは最小の要素を取り出して削除
  - 取り出す要素数は n 個
- insert，deleteMinが $O(\log n)$ 以下なら
  全体の計算量は $O(n \log n)$

5

ヒープソートの原理(3) 探索アルゴリズム

- ハッシュ法
  - insertは $O(1)$ だが，deleteMin を効率よく実行不可
- 二分探索木
  - 基本はinsert，deleteMinともに $O(\log n)$
  - 最悪ケースは $O(n)$
- 平衡木
  - insert，deleteMinともに $O(\log n)$
  - 計算量のオーダーは問題ないが定数係数が大きい
  - search は不要．無駄な部分がある

6

ヒープソートの原理(4) 優先順位付き待ち行列

- insertとdeleteMinを効率よく実行できないか？
  - 優先順位付き待ち行列なら可能
- 優先順位付き待ち行列（priority queue）
  - キューなので挿入と取り出しという基本操作を持つ
  - 各要素は「優先順位」を持つ
  - 取り出す際は最大の優先順位を持つ要素から
- 優先順位付き待ち行列は待ち行列を一般化したもの
  - 登録順に番号を振り，小さいものを優先すれば待ち行列
  - 登録順に番号を振り，大きいものを優先すればスタック

7

ヒープソートの原理(5) 半順序木

- 半順序木（partial ordered tree）
  - 優先順位付き待ち行列を効率よく実現できるデータ構造
  - 二分木
  - 親の値は子の値より大きくない
    （＝親は子より小さいか等しい）
  - 木はバランスが取れた形とする
    - （広い意味での）完全二分木（p.240参照）
    - 最下段の葉は左詰め
    - 根から最も近い葉と最も遠い葉への経路長の差は1以内

8

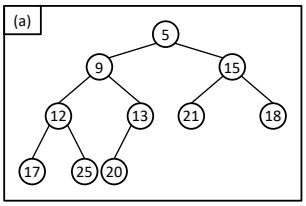
deleteMinの操作

- 最小の要素を見つける
  - 必ず木の根が最小の要素（半順序木の条件から）
- 根の要素を取り除いた後が問題
  - どうやって半順序木の条件を保つか
- 取り除いた要素の代わりを根において，
  半順序木の条件が成り立つように木を再構成

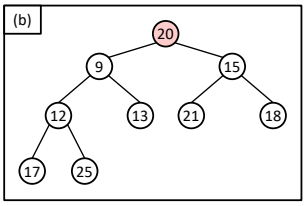
9

deleteMinの過程(1)

- 初期状態
  - 半順序木の条件を満たす



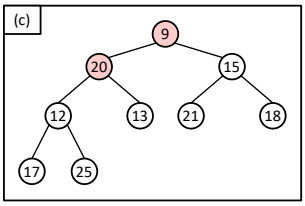
- 根の要素5を削除
- 最下段右の要素20を
  根の位置に移動



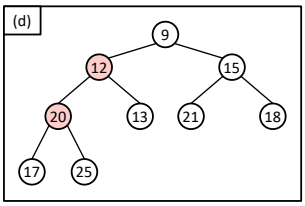
10

deleteMinの過程(2)

- 根の要素20を2つの子の
  うちの小さいほうと交換
  - 要素9と交換



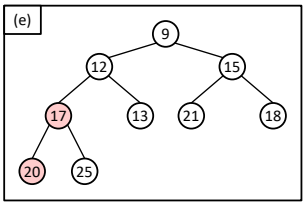
- 要素20を2つの子のうち
  の小さいほうと交換
  - 要素12と交換



11

deleteMinの過程(3)

- 要素20を2つの子のうち
  の小さいほうと交換
  - 要素17と交換



12

deleteMinの計算量

- 木の高さは高々 $\log_2 n$
- 平均すると交換は $\log_2 n/2$ 回
- 親子の大小を比較して交換するのは $O(1)$
- よって deleteMin の1回の計算量は $O(\log n)$

13

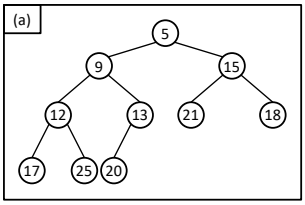
insertの操作

- 要素を最下段の一番左寄りの空いている場所へ置く
  - 最下段がいっぱいなら木の高さを1つ増やして左端へ
- その後，半順序木の条件が満足するように再構成

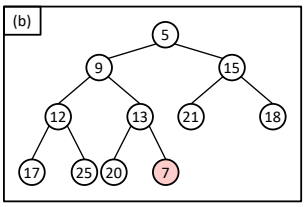
14

insertの過程(1)

- 初期状態
  - 半順序木の条件を満たす
  - 要素7を新たに追加する



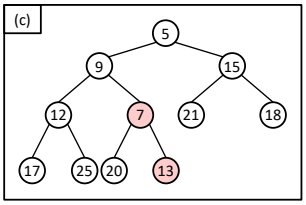
- 最下段の最も左寄りの空いている場所に要素7を追加



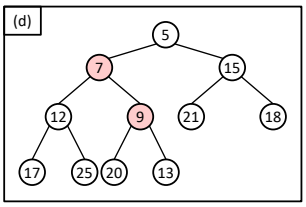
15

insertの過程(2)

- 要素7を親要素13と交換
  - 半順序木の条件を満たすようにするため



- 要素7を親要素9と交換
  - 半順序木の条件を満たすようにするため
  - これで半順序木の条件をクリアできた



16

insertの計算量

- 木の高さは高々 $\log_2 n$
- 平均すると交換は $\log_2 n/2$ 回
- 親子の大小を比較して交換するのは $O(1)$
- よって insert の1回の計算量は $O(\log n)$

17

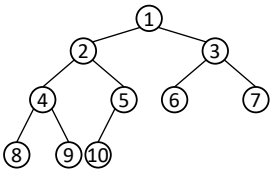
半順序木の実現方法

- insertの過程で子から親をたどる必要
- 通常の木構造は親から子のリンクはあるが逆はない
- リンクを使うと面倒
- 配列を使って実現
  - ヒープ：配列を使って半順序木を表現するデータ構造

18

ヒープ(1)

- 二分木の接点に，左から右，上から下の順で1から番号を振っていく
- この番号を各要素の配列内での添字とする
  - 根は a[1]，根の子は a[2], a[3]
  - a[i] の子は a[2*i] と a[2*i+1]
  - a[i] の親は a[i/2]
- ここではJavaの添字と違って 1 からスタート

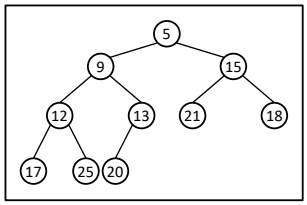


19

ヒープ(2)

- 最下段を左詰めにすれば n 個の要素を持つヒープはa[1] ～ a[n] になる

a[1]	5
a[2]	9
a[3]	15
a[4]	12
a[5]	13
a[6]	21
a[7]	18
a[8]	17
a[9]	25
a[10]	20



20

ヒープの実装(1)

詳細は教科書 pp.366～369 List 16.2

- Heapクラス
  - ヒープを実現
  - 添字の扱いに注意（スタートは1から）

```
class Heap {
  int[] a ;  // ヒープの実体が入る配列
  int   n ;  // ヒープに入っている要素の個数

  // ヒープを生成
  public Heap(int size) {
    a = new int[size + 1] ;  // a[0] は使わないため1つ余分に確保
    n = 0 ;
  }
}
```

21

ヒープの実装(2)

詳細は教科書 pp.366～369 List 16.2

- downHeapメソッド
  - ヒープの先頭の要素 a[1] を必要な場所まで沈める

```
class Heap {
  private void downHeap() {
    int value = a[1] ;  // 沈められる要素の値をvalueに退避
    int i = 1 ;
    while (i <= n/2) {  // 節iが子を持つ限り繰り返す
      int j = i * 2 ;   // 節iの子のうち，小さいほうを節jとする
      if (j + 1 <= n && a[j] > a[j + 1]) {
        j++ ;
      }
      if (value <= a[j]) break ;  // 親が子より大きければ終了
      a[i] = a[j] ;               // 節iに節jの値を入れる
      i = j ;                     // 節jに注目
    }
    a[i] = value ;      // 節iにvalueの値を入れる
  }
}
```

22

ヒープの実装(3)

詳細は教科書 pp.366～369 List 16.2

- deleteMinメソッド
  - ヒープから最小の要素を削除
  - ヒープの再構成はdownHeapメソッドに任せる

```
class Heap {
  int deleteMin() {
    if (n < 1) {  // ヒープが空なら例外を投げる
      throw new IllegalStateException() ;
    }
    int value = a[1] ;  // 根の要素a[1]を返り値にする
    a[1] = a[n--] ;     // ヒープの最後の要素を先頭に移動
    downHeap() ;        // 移動した値を適切な場所まで沈める

    return value ;      // 準備した返り値を返す
  }
}
```

23

ヒープの実装(4)

詳細は教科書 pp.366～369 List 16.2

- upHeapメソッド
  - ヒープ中のx番目の要素を必要な場所まで浮かび上がらせる

```
class Heap {
  private void upHeap(int x) {
    int value = a[x] ;  // 浮かび上がらせる要素の値を退避

    while (x > 1 && a[x/2] > value) {
      a[x] = a[x/2] ; // 親の値を子に移す
      x /= 2 ;        // 注目点を親に移す
    }
    a[x] = value ;    // 最終的な落ち着き先に代入
  }
}
```

24

ヒープの実装(5)

詳細は教科書 pp.366～369 List 16.2

- insertメソッド
  - ヒープに要素を登録
  - ヒープの再構成はupHeapメソッドに任せる

```
class Heap {
  public void insert(int elem) {
    if (n >= a.length) {// 登録できない場合は例外を投げる
      throw new IllegalStateException() ;
    }
    a[++n] = elem ;   // とりあえず要素をヒープの最後に入れる
    upHeap(n) ;       // 入れた要素を正しい位置まで浮かび上がらせる
  }
}
```

25

ヒープソートの実装(1)

詳細は教科書 p.371 List 16.3

- Heapクラスを利用した実装

作業用ヒープが欠点
O(n)の領域が必要

```
class HeapSort0 {
  public static void sort(int[] a) {
    int n = a.length ;              // 配列の要素数
    Heap heap = new Heap(n) ;       // 作業用のヒープを作成
    for (int i = 0 ; i < n ; i++) {// 配列の全要素をヒープに挿入
      heap.insert(a[i]) ;
    }
    for (int i = 0 ; i < n ; i++) {// 小さい順に取り出して配列へ戻す
      a[i] = heap.deleteMin() ;
    }
  }
}
```

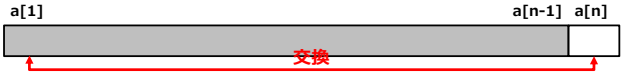
```
static void sort(int[] a)
{
  int n = a.length ;  // 配列の要素数
  SomeSearchTree tree = new SomeSearchTree() ;  // 探索木を準備
  for (int i = 0 ; i < n ; i++) {// 配列の要素を探索木に挿入
    tree.insert(a[i]) ;
  }
  for (int i = 0 ; i < n ; i++) {// 小さいものから取り出して配列へ戻す
    a[i] = tree.deleteMin() ;
  }
}
```

26

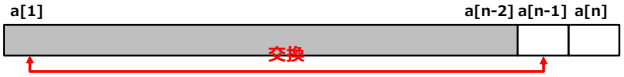
ヒープソート後半の原理

- 最初は配列全体がヒープ

- 先頭の要素a[1]と最後の要素a[n]を交換
- a[1]～a[n-1]に対してヒープを再構成



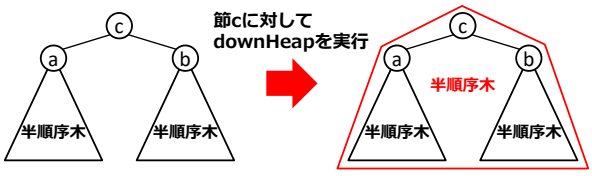
- 先頭の要素a[1]と最後の要素a[n-1]を交換
- a[1]～a[n-2]に対してヒープを再構成



27

ヒープソート前半の原理

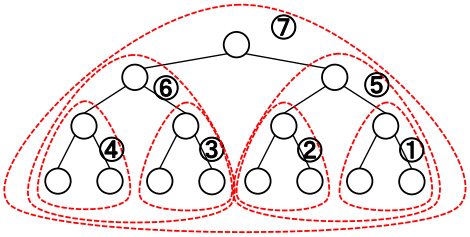
- 節aとbを根とする部分木は半順序木
- 節cに対してdownHeapを実行すると全体が半順序木に



28

downHeapの適用順

- ボトムアップにdownHeapを適用する順序



29

ヒープソートの実装(2)

詳細は教科書 pp.375～376 List 16.4

- downHeapメソッド
  - 範囲をa[from]からa[to]に限定したもの

```
class HeapSort {
  private static void downHeap(int[] a, int from, int to) {
    int value = a[from] ;  // 沈められる要素の値をvalueに退避
    int i = from ;
    while (i <= to/2) {    // 節iが子を持つ限り繰り返す
      int j = i * 2 ;      // 節iの子のうち，小さいほうを節jとする
      if (j + 1 <= to && a[j] > a[j + 1]) {
        j++ ;
      }
      if (value <= a[j]) break ; // 親が子より大きければ終了
      a[i] = a[j] ;              // 節iに節jの値を入れる
      i = j ;                    // 節jに注目
    }
    a[i] = value ;         // 節iにvalueの値を入れる
  }
}
```

30

ヒープソートの実装(3)

詳細は教科書 pp.375～376 List 16.4

- sortメソッド
  - 整列対象はa[1]から最後の要素まで

```
class HeapSort {
  public static void sort(int[] a) {
    int n = a.length - 1 ;  // 渡された配列の最後の添字

    for (int i = n / 2 ; i >= 1 ; i--) {  // 下から上に向かって
      downHeap(a, i, n) ;   // i～nの範囲でdownHeapを実行
    }

    for (int i = n ; i >= 2 ; i--) {
      int tmp = a[1] ;      // i番目と先頭をswap
      a[1] = a[i] ;
      a[i] = tmp ;
      downHeap(a, 1, i - 1) ;  // 先頭からi-1までのヒープを再構成
    }
  }
}
```


ヒープソートの性質

- ヒープを構成する際の大小関係を反転すれば昇順の整列も可能
  - 「親のほうが子より小さくない」
- 計算量：$O(n \log n)$
  - 前半のループ
    - n/2回実行
    - downHeapは平均して $O(\log n)$
    - よって $O(n \log n)$ に見えるが，実は解析すると $O(n)$
  - 後半のループ
    - n-1回実行
    - downHeapは $O(\log n)$
    - よって $O(n \log n)$

まとめ

- ヒープソートの原理
- 配列によるヒープ
- ヒープソートの実装

参考文献

- 定本 Javaプログラマのための アルゴリズムとデータ構造（近藤嘉雪）
- 新・明解 Javaで学ぶ アルゴリズムとデータ構造（柴田望洋）
- 岩波講座ソフトウェア科学 3 アルゴリズムとデータ構造（石畑清）
- Javaで学ぶアルゴリズムとデータ構造 Robert Lafore（著）・岩谷 宏（翻訳）
- Java アルゴリズム+データ構造完全制覇 オングス (著)・杉山 貴章・後藤 大地 (監修)