プログラミング通論

I2クラス

第13回 2014/01/14

講義内容（予定）

- 第1回(2013/10/01) C言語基本機能の復習
- 第2回(2013/10/08) 基本的データ型(1) ポインタ
- 第3回(2013/10/22) 基本的データ型(2) スタック、キュー、デク
- 第4回(2013/10/29) 再帰呼び出し(1) 関数と引数の復習、再帰の概念
- 第5回(2013/11/05) 再帰呼び出し(2) 分割統治法
- 第6回(2013/11/12) 再帰呼び出し(3) 再帰呼び出しの除去
- 第7回(2013/11/19) 中間試験と解説
- 第8回(2013/11/26) リスト構造(1) リストの定義、基本操作
- 第9回(2013/12/03) リスト構造(2) リストの応用
- 第10回(2013/12/10) リスト構造(3) 抽象データ型としてのリスト
- 第11回(2013/12/17) 整列(1) 基本整列法
- 第12回(2014/01/07) 整列(2) 高速手法
- 第13回(2014/01/14) 基数整列法、マージソート
- 第14回(2014/01/21) 探索
- 第15回(2014/01/28) 進んだ話題

前回の復習

- 整列：基本整列法の復習
- 整列：高速手法
  - shell sort
  - quick sort

本日の内容

- 大きいレコードの整列（復習）
- 整列：高速手法
  - heap sort
  - distribution counting
  - radix sort

卒業見込みの人へ

- 成績報告時期が異なるので、申し出ること

小テスト #12 解説

```
(init) 13  5 11  8  7 12  4  2 10  3  1  6  9  0
(h= 4) 13  0  1  2  7  3  4  6  9  5 11  8 10 12
(h= 1) 13  0  1  2  3  4  5  6  7  8  9 10 11 12
```

- 講義中に紹介するアルゴリズムは、断りのない限り a[0] を対象としないことに注意

大きいレコードの整列(1)

- recordtypeが大きい場合、swapで移動していては時間がかかる
  - 構造体のコピーは、各メンバを全てコピー
- recordtypeへのポインタ（カーソル）の配列を用意して、その要素（ポインタ）をソート

大きいレコードの整列(2)

```
void insertion_sort2(recordtype a[], int n){
  int i, k, v, next; recordtype tmp;
  int p[LIMIT];
  for(i = 0; i <= n; i++) p[i] = i;
  a[0].key = -∞;
  for(i = 2; i <= n; i++){
    v = p[i]; k = i;
    while(a[p[k-1]].key > a[v].key){
      p[k] = p[k-1]; k--;
    }
    p[k] = v;
}
```

cursorの配列

cursorだけを並べ替え

大きいレコードの整列(3)

```
/* カーソルの整列終了後aの中身を入れ替え */

for(i = 1; i <= n; i++){
  if(p[i] != i){
    tmp = a[i]; next = i;
    do{
      k = next; a[k] = a[p[k]];
      next = p[k]; p[k] = k;
    }while(next != i);
    a[k] = tmp;
  }
}
}
```

整列済みのcursorを用いてデータを整列

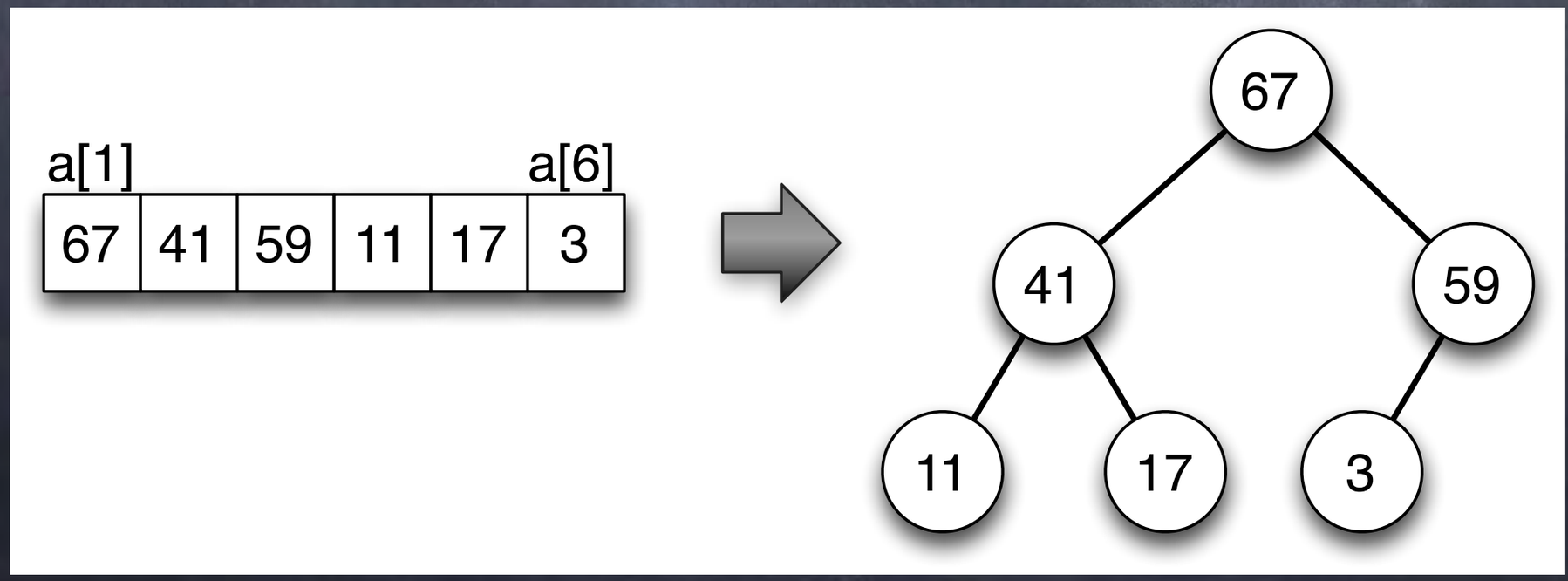
p[k] が示す場所のデータと入れ替える操作を繰り返す

heap sort (1)

- 配列 a[1], ..., a[n] において、a[i].key >= a[2*i].key && a[i].key >= a[2*i+1].key を満たすものをヒープと呼ぶ（大小関係は逆も可）
- a[i] を親、a[2*i], a[2*i+1] を子と考えて図示すると2分木（2又に分かれる木構造）になる
  - 半順序木

heap sort (2)

- 以下は半順序木の例
  - a[1] -> a[2], a[3]
  - a[2] -> a[4], a[5]
  - a[3] -> a[6]
- (補足) 木の根は上にあり、下に向かって育つ



heap sort (3)

- a[first] を開始点（root = 木の根）とするヒープを考える
- a[first+1], ..., a[last] はヒープ関係を満たしているが、a[first] はヒープ関係を満たしていないかもしれない、とする
- a[first] を下の方のノードと交換して、ヒープ関係を満足させる操作 pushdown() を定義する

heap sort (4)

```
void pushdown(recordtype a[], int first, int last){
  int r = first; int k = 2*r;
  while(k <= last){
    if(k < last && a[k].key < a[k+1].key){ k++; }
    if(a[r].key >= a[k].key){ break; }
    swap(&a[r], &a[k]);
    r = k; k = 2*r;
  }
}
```

より大きい方と比較

heap関係なら終了

heap関係でないので入れ替え

さらに下のノードと比較

heap sort (5)

```
void heapsort(recordtype a[], int n){
  int i;
  for(i = n/2; i >= 1; i--){ pushdown(a, i, n); }
  for(i = n; i >= 2; i--){
    swap(&a[1], &a[i]);
    pushdown(a, 1, i-1);
  }
}
```

全データをheapに構成

最大のものを最後に

残りデータをheapに

heap sort (6)

- pushdown によるヒープ構成 ... O(log n)
- ヒープソートの比較回数 ... O(n log n)

distribution counting (1)

- 分配計数法
- キーの取り得る範囲が限られる場合に適用
- 0からm-1までの範囲の整数をキーとするレコードn個からなるデータを整列
  - まずキーを値ごとに数える
  - 次にこの個数によって並べ替える

distribution counting (2)

```
void distribution_counting(recordtype a[], int n){
  int i, j, count[m];
  recordtype b[LIMIT];
  for(j = 0; j < m; j++){ count[j] = 0; }
  for(i = 1; i <= n; i++){ count[a[i].key]++; }
  for(j = 1; j < m; j++){ count[j] += count[j-1]; }
  for(i = n; i >= 1; i--){
    b[count[a[i].key]] = a[i];
    count[a[i].key]--;
  }
  for(i = 1; i <= n; i++){ a[i] = b[i]; }
}
```

mはキーの範囲で、どこかで定義済みとする

出現頻度

累積数に変換

元データをソート後の場所に移動

ソート済みデータで上書き

radix sort (1)

- 基数整列法
- キーの値の、桁ごとに整列する

radix sort (2)

- 基本概念（Wikipediaより）

170, 45, 75, 90, 2, 24, 802, 66
↓ 1の桁でソート

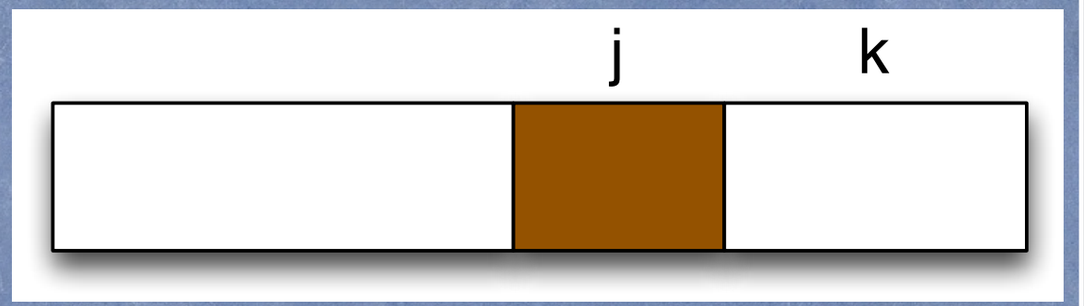
170, 90, 2, 802, 24, 45, 75, 66
↓ 10の桁でソート

2, 802, 24, 45, 66, 170, 75, 90
↓ 100の桁でソート

2, 24, 45, 66, 75, 90, 170, 802

radix sort (3)

```
/* xをkビット右へシフトし、その左jビットを取り出す */
void bits(int x, int k, int j){
  return (x>>k) & ~(~0 << j);
}
```



下jビットが1

- 上記のようなビット演算を用意しておく
- キーが 2^31 より小さい正整数からなるときは、次頁の関数を radixsort(a, 1, n, 30) として呼び出す

radix sort (4)

```
void radixsort(recordtype a[], int l, int r, int b){
  int i, j;
  if(r > l && b >= 0){
    i = l; j = r;
    do{
      while(bits(a[i].key, b, 1) == 0 && i < j){ i++; }
      while(bits(a[j].key, b, 1) == 1 && i < j){ j--; }
      if(i != j){ swap(&a[i], &a[j]); }
    }while( j != i);
    if(bits(a[r].key, b, 1) == 0){ j++; }
    radixsort(a, l, j-1, b-1);
    radixsort(a, j, r, b-1);
}}
```

a[l], …, a[r] を下からbビット目の値でソート

quick sortのpartition()に似たやり方

全部0ならずらす

0の部分をソート

1の部分をソート

radix sort (5)

- radixsort() 関数はキーの1桁が1ビットの前提
- キーの1桁がsビットで、全体でwビットあるとすると、w/s回やればソートできる（次項）

radix sort (6)

```
void straightradix(recordtype a[], int n){
  int count[m]; /* distribution counting */
  int i, j, pass; recordtype b[LIMIT];
  for(pass = 0; pass < w/s; pass++){
    for(j = 0; j < m; j++){ count[j] = 0; }
    for(i = 1; i <= n; i++){
      count[bits(a[i].key, pass*s, s)]++;
    }
    for(j = 1; j < m; j++){ count[j] += count[j-1]; }
    for(i = n; i >= 1; i--){
      b[count[bits(a[i].key, pass*s, s)]] = a[i];
      count[bits(a[i].key, pass*s, s)]--;
    }
    for(i = 1; i <=n; i++){ a[i] = b[i]; }
}}
```