プログラミング通論

I2クラス

第12回 2014/01/07

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

- 中間試験の解説
- 整列（ソート）
  - selection sort
  - insertion sort

本日の内容

- 整列：基本整列法の復習
  - bubble sort
  - 大きいレコードの整列
- 整列：高速手法

小テスト #11 解説

- 設問1
  - element: 0, 0, 1, 2, 0
  - nexta: 1, 2, 3, -1, 5
- 設問2
  - element: 0, 0, 1, 2, 0
  - nexta: 1, 3, 3, -1, 5
- （デバッガを用いた表示を紹介）

整列

- 整列（ソート）
  - データのリストを線形順序で並べ替える
    - 並べ替えの対象は「キーを含むレコード」
- キー
  - レコードの中で順序比較の対象となる欄

selection sort (1)

- 選択整列法
- 最小値を見つけて、a[1] に移動する
- 2番目に小さい値を見つけて、a[2] へ
- 以下同様
- なお、a[0] は使わない実装であることに注意

selection sort (2)

```
void selection_sort(recordtype a[], int n){
  int i, j, minindex;
  for(i = 1; i < n; i++){
    minindex = i;
    for(j = i+1; j <=n; j++){
      if(a[j].key < a[minindex].key){ minindex = j; }
    }
    swap(&a[minindex], &a[i]);
  }
}
```

データ全体を処理する

残りから最小を探す

見つけた最小を前方に移動

insertion sort (1)

- 挿入整列法
- 先頭からi番目までが整列済みのとき
- a[i+1] を、正しい位置に挿入する

insertion sort (2)

```
void insertion_sort(recordtype a[], int n){
  int i, j; recordtype v;
  a[0].key = -∞; /* keyとしてあり得る最小値 */

  for(i = 2; i <= n; i++){
    v = a[i]; j = i;
    while(a[j-1].key > v.key){ a[j] = a[j-1]; j--; }
    a[j] = v;
  }
}
```

a[i] が移動対象

iにあったものをjに挿入

bubble sort (1)

- バブル整列法
- 隣と比べて、逆順なら入れ替え
- 可視化すると、泡が浮かび上がるように見えることから命名

bubble sort (2)

```
void bubble_sort(recordtype a[], int n){
  int i, j;
  for(i = 1; i < n; i++){
    for(j = n; j >= i+1; j--){
      if(a[j].key < a[j-1].key){
        swap(&a[j], &a[j-1]);
      }
    }
  }
}
```

データ全体を処理する

より小さいものは前に

bubble sort (3)

- 最悪の場合
- 比較回数 … 約 n^2 / 2 ← 平均に同じ
- 交換回数 … 約 n^2 / 2 ← 平均は半分

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; recordype 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

shell sort (1)

- 間隔hだけ離れた部分データを挿入ソート
  - a[i], a[i + h], a[i + 2*h], a[i + 3*h], ...
    - 間隔hで挿入した部分データを、h-整列していると呼ぶ
- 終わったら、より小さいhで同じことをする
- 挿入場所を先頭から探す、よりは早くなる

shell sort (2)

初期状態

5	8	7	4	2	3	1	6	9

h=4整列済み

2	3	1	4	5	8	7	6	9

h=1整列済み

1	2	3	4	5	6	7	8	9

shell sort (3)

```
void shellsort(recordtype a[], int n){
  int i, j, h; recordtype v;
  for(h = 1; h <= n/9; h = 3*h + 1);
  for( ; h > 0; h /= 3){
    for(i = h+1; i <= n; i++){
      v = a[i]; j = i;
      while(j > h && a[j-h].key > v.key){
        a[j] = a[j-h]; j -= h;
      }
      a[j] = v;
}}}
```

間隔hを決める

hを徐々に縮める

insertion sortに同じ

shell sort (4)

- insertion sortの場合、挿入すべき場所を探すのに、最悪O(n)かかる
- shell sortの場合、h個ずつ飛ばして探すので比較回数が少ない + 一度の移動量が大きい
- n^(3/2) 以上に比較されることはないことが知られている

quick sort (1)

- 分割統治法による整列アルゴリズム
- 列を2つに分割して、それぞれを整列していく

1. pivotを1つ選び、pivotより小さい部分列と
   大きい部分列に分ける
2. 小さい側をquicksort
3. 大きい側をquicksort

分割統治法

- Divide and conquer
- 解くべき問題を小規模な部分問題に分割し、部分問題の解を結合して全体の解を得ようとする方法
- 以下の3ステップからなる
  1. Divide -- 問題を「小問題」に分割
  2. Conquer -- 小問題を解決
  3. Combine -- 結果を統合

quick sort (2)

```
void quicksort(recordtype a[], int l, int r){
  int i;
  if(l < r){
    i = partition(a, l, r);
    quicksort(a, l, i-1);
    quicksort(a, i+1, r);
  }
}
```

pivotはpartition()が決めて

全データを並び替え + 添字を返す

pivotを境目にして小問題に分割

それぞれを統治すれば完了

quick sort (3)

```
int partition(recordtype a[], int l, int r){
  int i, j; recordtype v;
  v = a[l]; i = l; j = r+1;
  do{
    do{ i++; }while(a[i].key < v.key);
    do{ j--; }while(a[j].key > v.key);
    if(i < j){ swap(&a[i], &a[j]); }
  }while(j > i);
  a[l] = a[j]; a[j] = v;
  return j;
}
```

先頭要素をpivotとする

注意(後述)

pivotより小さいものを左へ

pivotより大きいものを右へ

quick sort (4)

- partition() の i++ で、添字のレンジチェックを省略した → レンジチェック or 番兵が必須
- quick sortのポイントは、なるべく半分ずつに分割すること → pivotの選び方が重要
  - 乱数で選ぶ
  - いくつかのデータを取って中央値を使う
- 平均比較回数 … O(n log(n))

quick sort (5)

- quick sortはいろいろと研究されていて、様々なバリエーションがある
- 番兵を使わず、partition() も呼び出さない別バージョンを紹介しておく
  - a[0] からデータが入っていることに注意

quick sort (6)

```
void quicksort2(recordtype a[], int n){
  int i, last;
  if(n <= 1) return;
  swap(&a[0], &a[ rand() % n ]);
  last = 0;
  for(i = 1; i < n; i++){
    if(a[i].key < a[0].key){ swap(&a[++last], &a[i]); }
  }
  swap(&a[0], &a[last]);
  quicksort2(a, last); quicksort2(a+last+1, n-last-1);
}
```

pivotは乱数で決める

pivotより小さいものを左へ

小問題に分割して統治