プログラミング通論'19 ＃11 – 高速な整列アルゴリズム

久野 靖 (電気通信大学)

2020.2.21

今回は次のことが目標となります。

- 様々な整列アルゴリズムについて理解する
- 整列における安定性の概念を理解する

1　計算量の検討とマージソート/クイックソート

1.1　より高速な整列のために必要なこと

前回扱った基本的な整列アルゴリズム (選択ソート、挿入ソート、バブルソート) ではいずれも、n 個の数それぞれに対して、n に比例する回数の操作を行うため、計算量が $O(n^2)$ となっていました。$O(n^2)$ では、数万程度くらいまでしか短い時間 (数秒程度) では扱えません。

ただ 1 つだけ、コムソートでは「間隔 d を n から始めて一定比率で狭めていき、1 になったときには整列が終わっている」ようにすることで、「n 個の数それぞれに対する操作を $\log n$ 回」で済ませることができ、$O(n \log n)$ の時間計算量を達成していました。

整列では扱う数 n は変えようがないので、それらに全体対して操作を行う回数を減らす (絶対値ではなく、n に比例から $\log n$ に比例、さらにできれば定数 C に比例というふうに) ことがポイントになります。今回はそれを実現する整列アルゴリズムを複数見て行きます。

1.2　マージ (併合) 操作

マージ (併合、merge) 操作とは、2 つの整列ずみの列を 1 本の整列ずみの列に合成する操作です。図 1 は、以前扱った「先頭に要素数を格納した整数配列」を用いた列に対するマージ操作を例示しています。

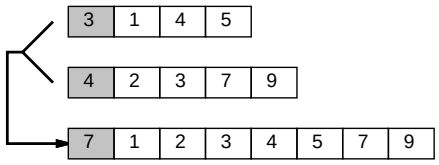

図 1: 列に対する併合操作

2 つの列を受け取り、併合した列を返すメソッド ivec_merge を見てみましょう。

```
int *ivec_new(int size) {
  int *a = (int*)malloc((size+1) * sizeof(int));
  a[0] = size; return a;
}
int *ivec_merge(int *b, int *c) {
  int ib = 1, ic = 1, ia = 1, *a = ivec_new(b[0]+c[0]);
  while(ia <= a[0]) {
```

```
    if(ib > b[0])          { a[ia++] = c[ic++]; }
    else if(ic > c[0])     { a[ia++] = b[ib++]; }
    else if(c[ic] < b[ib]) { a[ia++] = c[ic++]; }
    else                   { a[ia++] = b[ib++]; }
  }
  return a;
}
```

返す列の長さは 2 つの列の長さの和として分かるので、まず返す列 a の領域を確保します。それから、変数 ia を a の格納位置、ib、ic を b、c の取り出し位置 (いずれも最初は 1) としてから、列 a が一杯になるまでのループに入ります。

ループの中では、列 b の取り出し位置が最後まで来ていたら c から取り出し (取り出し位置は進める)、列 c の取り出し位置が最後まで来ていたら b から取り出し (〃)、いずれも最後でなければ 2 つの列の取り出し位置の要素の大小で小さい側から取り出し (〃)、a に格納します (格納位置は進める)。ループが終わったら a に結果ができているので返します。

1.3 キューを使ったマージソート

ではこれを整列に活かすにはどうしたらよいでしょう。1 つの分かりやすい方法は、最初はすべての数を「長さ 1 の列」と考え (長さ 1 ならそのままで整列ずみと言えます)、それらを順次マージしていくことです。マージ中の多くの列を蓄えるのにはキューが使えます。すなわち図 2 のように、キューから 2 つ deq で取り出して 1 つにマージしたものを enq で投入することを繰り返すわけです。

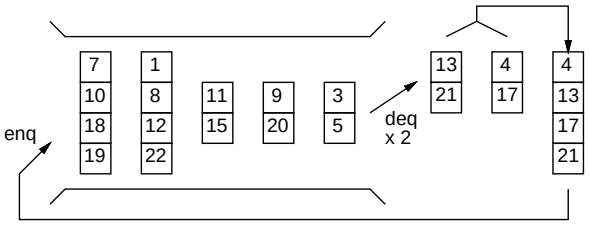

図 2: キューを用いたマージソート

1 回マージすることにキューの内容は差し引き 1 ずつ減って行きますから、最後に 1 つの列になり、その列がもとの列の整列ずみの内容となっています。これをコードにしたものを示します。

```
// mergesort1 --- merge sort unsing queue
#include <stdlib.h>
#include "pqueue.h"
(ivec_new, ivec_merge をここに)
void mergesort1(int *a, int n) {
  pqueuep q = pqueue_new(n+1); int *v, *w;
  for(int i = 0; i < n; ++i) {
    v = ivec_new(1); v[1] = a[i]; pqueue_enq(q, v);
  }
  while(true) {
    v = (int*)pqueue_deq(q); if(pqueue_isempty(q)) { break; }
    w = (int*)pqueue_deq(q); pqueue_enq(q, ivec_merge(v, w));
    free(v); free(w);
  }
  for(int i = 0; i < n; ++i) { a[i] = v[i+1]; }
}
```

キューの各要素はポインタ (整数へのポインタ) になるので、ポインタ用のキューを使うものとします (コードは整数用と同様、授業サイトに掲載)。まず元の配列 (これは個数つきではなく他の例列コードと合わせて個数を別に渡しています) から、n 個の長さ 1 の個数つき配列を作り、その要素としてデータを入れてキューに enq していきます。そのあと無限ループで 2 つ deq してマージしたものを enq しますが (取り出した 2 つの領域は以後不要なので `free`)、ただし 1 つ取り出した時点でキューが空なら終わりなのでループを抜け、元の領域にデータをコピーし戻します。

演習 1 キューを使ったマージソートのコードを動かし、整列できることや整列時間を確認しなさい (単体テストすること)。加えて、キューの代わりにスタック (ポインタなので `pstack`) を使うとどうなるか理由とともに予想し、こちらも単体テストして確認しなさい。今回の課題全般に、前回の `expect_sort_iarray` を (必要なら改造の上) 利用するとよいでしょう。

演習 2 「長さ 1 から始めてマージしていく」やり方は、キューを使わないでも実現可能である。そのようなマージソートのコードを作成し、整列できることや整列時間を確認しなさい (単体テストすること)。

1.4 分割統治による再帰マージソート

先の例では「長さ 1 から始めてマージしていく」考え方でしたが、逆に分割統治、つまり「長さ n の問題を分割して扱う」ことを再帰的に行う考え方でもできます。具体的には「$\frac{n}{2}$ ずつの列に分けて、自分を再帰呼び出しして整列させ、終わったらそれをマージする」形になります。

```
// mergesort2 --- merge sort with recuresive division
#include <stdlib.h>
(merge はここでは省略)
static void ms(int *a, int i, int j, int *b) {
  if(i >= j) { return; }
  int k = (i + j) / 2;
  ms(a, i, k, b); ms(a, k+1, j, b);
  merge(b, a+i, k-i+1, a+k+1, j-k);
  for(k = 0; k < j-i+1; ++k) { a[i+k] = b[k]; }
}
void mergesort2(int *a, int n) {
  int *b = (int*)malloc(n * sizeof(int)); ms(a, 0, n-1, b); free(b);
}
```

この場合、「配列の何番から何番を整列する」というふうにパラメタで明示する方が扱いやすいので、`mergesort2` は範囲を明示して再帰手続き `ms` を呼ぶ仲介だけの役割です。また、マージする時は作業用の配列が欲しいので、それを割り当てて `ms` に渡し、終わったら解放します。

`ms` ですが、配列と整列範囲 (`i` から `j` まで) と作業用配列 `b` を受け取ります。範囲の長さが 1 以下なら整列は済んでいるのですぐ戻ります。それ以外は、中間点 `k` を決めて、`i`〜`k` と `k+1`〜`j` をそれぞれ自分を再帰呼び出しして整列します。終わったらマージしますが、そのパラメタは結果を受け取る配列と、1 番目の配列および長さ、2 番目の配列および長さです。1 番目と 2 番目の配列はポインタ計算 `a+i`、`a+k+1` で求まることに注意。マージ後は結果を `b` から `a` の当該位置にコピーし戻します。

演習 3 省略されている `merge` を書いて補って再帰マージソートを動かし、整列できることや整列時間を確認しなさい (単体テストすること)。

1.5 クイックソート

再帰版マージソートは言うなれば「とりあえず列を 2 つに分けて (自分を再帰呼び出しして) 整列してしまう」ため、その整列できた 2 つの列を 1 つにするために面倒なマージという操作が必要なのでした (図 3 左)。そこで逆に、まず列を「ある値 p より大きい部分と小さい部分に分割し (partition)、それからそれぞれを (自分を再帰呼び出しして) 整列する」ようにすれば、再帰が終わった時はもう「p より小、p、p より大」の順になっているので整列が完了します (図 3 右)。これが**クイックソート** (quick sort) で、p の値のことをピボットと呼びます。

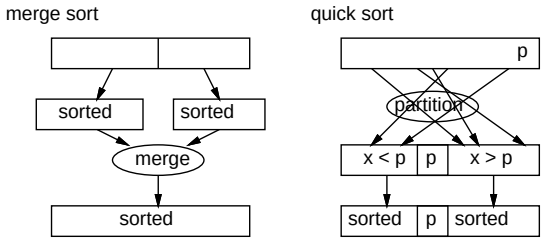

図 3: 再帰マージソートとクイックソート

コードを示します。quicksort は範囲を指定して下請け qs を呼びます。qs はピボット p には左端の値を使います。ランダムな列であればどこを取っても同じなのでこれでよいですが、整列済みの列だとまずいですね。

```
// quicksort --- quick sort with recuresive division
static void iswap(int *a, int i, int j) {
  int x = a[i]; a[i] = a[j]; a[j] = x;
}
void qs(int *a, int i, int j) {
  if(j <= i) { return; }
  int s = i, pivot = a[j];
  for(int k = i; k < j; ++k) {
    if(a[k] < pivot) { iswap(a, s++, k); }
  }
  iswap(a, j, s); qs(a, i, s-1); qs(a, s+1, j);
}
void quicksort(int *a, int n) { qs(a, 0, n-1); }
```

内側のループが分かりづらいですが、s は「ここより手前は p より小さい」という範囲を表す変数であり、変数 k を使って整列範囲全体を調べながら、p より小さい値があればそれを s の位置と交換することで s の場所に置き、s は 1 つ増やすようにすることで、ループの最後には全部の p より小さい値が s の手前に集まります。そこで最後に右端にあるピボットを s の位置と交換することで「p より小、p、p より大 (厳密には以上)」と並ぶわけです。

演習 4 上のクイックソートを動かし、整列できることや整列時間を確認しなさい (単体テストすること)。また、既に整列されている列を渡したときの挙動についても調べ、そのときの問題を解消する方法を実装しなさい (これも単体テストすること)。(ヒント: ピボットを整列範囲内からランダムに選べばよいです。)

1.6 ボゴソート

高速な整列アルゴリズムの話題の中ですが、逆に「できるだけ遅いアルゴリズム」ということで考案されたのが**ボゴソート** (bogo sort) です。その原理は簡単で「列をランダムにシャッフルし、並んでい

るかチェックする。並んでいなければまたシャッフルし…」と繰り返します。

演習 5 ボゴソートを実装し、動作と時間を確認しなさい (10 個くらいでやるのが無難かも)。また、時間計算量を見積もり、実測と比較検討しなさい (単体テストすること)。

演習 6 ボゴソートと同程度に遅い整列アルゴリズムは他にもある。考案し、実装してみなさい (単体テストすること)。

2　完全 2 分木の配列表現とヒープソート

2.1　2 分木とその表現

木 (tree) とは地面に生えている…ではなく、数学の場合「閉路を含まない**グラフ** (頂点と辺の集合)」ですが、コンピュータ科学の場合は「**根** (root)」はら始まり複数の「**節** (node)」がたどれるような (そしてやはり閉路は含まない) データ構造です。ある節にとって根に近い側の辺につながる節は「**親** (parent)」、それ以外の辺についながる節は「**子** (child)」となります。各節において、根からその節までの経路の長さ (辺の数) を深さ、子の数を**次数**と呼び、次数が 0 の節を「**葉** (leaf)」と呼びます。

データ構造として実現するときは、子の節へのポインタを親の節が持つので、その個数が問題になります。次数の最大が 2 である木は **2 分木** (binary tree)、3 以上である場合は**多分木** (N-ary tree) と呼びます。2 分木では子が最大 2 つであり、これを「左の子」「右の子」のように呼ぶことがあります。

2 分木のうち、すべての葉でない節の次数が 2 であるものを「**全 2 分木** (full binary tree)」と呼びます。ここで「**完全 2 分木** (complete binary tree)」という用語もあるのですが、これには (1) 全ての葉の深さが等しい全 2 分木 (節数は $2^n - 1$ に限られる)、(2) 前期に加えてそれに左から N 個ぶん葉を追加したもの、の 2 通りの意味があります。図 4 にこれらの構造を例示しました (最後の完全 2 分木で次数が 1 の節にくっついているのは「左の子」であることに注意)。

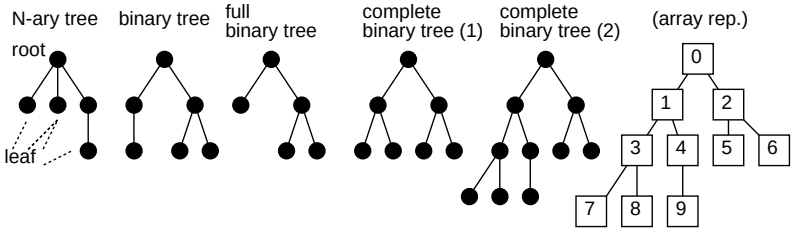

図 4: 木とさまざまな 2 分木

ここまで、いかにもポインタを使った動的データ構造のような流れで来ましたが、完全 2 分木 (2) については、配列を使った次のような表現方法があります。

- 根を添字 0 の位置に割り当てる。
- 任意の節 i(添字 i) について、左の子は添字 $2i+1$、右の子は添字 $2i+2$ に置く (そうすると、逆に子 i に対して親の添字は $(i-1)/2$ で表せる。除算は整数除算)。

図 4 の右端の図が、隣の完全 2 分木を配列表現したときの番号を表しています。[1]

2.2　完全 2 分木による最大ヒープと押し下げ

ヒープ (heap) とは「積み上げた束」という意味の英語ですがコンピュータ科学ではメモリの空き領域を集めて保持したものを通常言います。`malloc` はヒープからメモリを取って来る関数です。そしてもう 1 つの用法として、「**最大ヒープ** (maximum heap)」といった場合、次のような性質を持つ木構造を言います (大小を逆にした「最小ヒープ」もあります)。

[1] ここでは C 言語に合わせて根を 0 番としましたがが、根が 1 番の方が計算式は分かりやすく、左の子が $2i$、右の子が $2i+1$ になり、親が $i/2$ になります。

- 親の節の「値」が、(子があるとき) いずれの子の節の「値」よりも大きい。

上記が成り立てば当然、根には最大値があるので、その最大値を取り出し続けて並べれば整列が行えます。これをヒープソート (heap sort) と呼びます。ただし、効率がよい整列アルゴリズムであるためには、1 つ最大値を取り出したあと、そこに適当な代わりの値 (配列表現では配列に残っている最後の値を通常使います) を入れて上記の最大ヒープの条件が崩れた後、その条件を再度成り立たせる必要があり、しかもそれが $\log n$ の手間である必要があります。どうでしょうか。

実際にその方法を例示しましょう。図 5 左の最大ヒープにおいて、先頭の要素を取り除き、配列末尾にあった「8」をその位置に置いて埋めたとします (図 5 中)。この状態では最大ヒープではありません。そこで、この「8」を左または右の子と交換します。

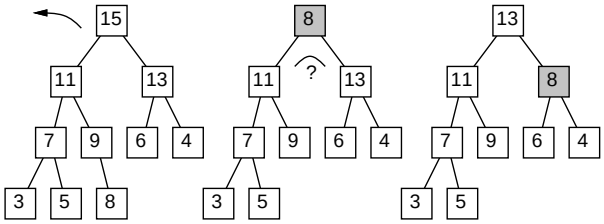

図 5: 最大ヒープの押し下げ操作

どちらと交換すべきでしょうか。もちろん、「大きい方と」交換すべきです。そうすれば、交換しなかった側の枝は確かに「親が大きい」が維持されていて、それ以上修正は不要です。さて、次に交換で新たに「8」を置いたところから下を検討します。2 分木は再帰的データ構造 (部分が全体と同じ構造) ですから、先と同様に進めます。まず最大ヒープになっているか調べますが、今度は左右の子とも「8」より小さいので、これで OK で終わりです (図 5 右)。

もしどちらかが「8」より大きいなら、再度大きい方の枝を選び交換して続けますが、最後は葉まで来ますからそこで必ず終わります。この、先頭に任意の値を置いて最大ヒープの性質を満たすように下に移して行く操作を「**押し下げ** (pushdown)」と呼んでいます。ノード数 n の完全 2 分木の高さ (葉までの深さの最大) は $\log_2 n$ ですから、押し下げ操作は最大 $\log n$ ステップで終わります。ということで、「次々に最大を取り出していく」部分の計算量は $O(n \log n)$ となります。

まだ 1 つ忘れていますね。最初に最大ヒープを作るのはどうしたらいいでしょうか。それは、今度は配列の最後から (つまり木の低い側から) 順にすべての節に対して押し下げを実施すればよいのです。葉のところはどのみち入れ換えはないですが、その上の段からは必要に応じて入れ換えが起きていきます。どの段階でも、ある節の押し下げを行うときには、その下はすべて最大ヒープになっているので、同じ押し下げの手順で OK です。ということは、n 個の値に対して最大 $\log n$ ステップの押し下げを行うので、最初にヒープを作る部分の計算量も $O(n \log n)$ です。

2.3 ヒープソートのコード

ここまでで全部説明はしてしまったので、あとはコードを見るだけです。まず見慣れないものとして、`P`、`L`、`R` という define がありますが、この define にはパラメタ `i` が付いています。これはちょうど関数と同じように右辺の中に埋め込まれて展開されます。関数でもよかったのですが、単なる計算式なので関数呼び出しが起きるよりは効率のよいマクロにしたかったということがあります。`iswap` はいつも通りです。

押し下げ操作ですが、配列、押し下げる節番号、そして最大ヒープの末尾の節番号を渡します。まず左の子の節番号を `k` に求めて、それが末尾を超えていない間繰り返しになります。次にループの中ですが、最初の if 文では、終わりまでに 1 以上余裕があれば右の子の節もあるので、その値の方が大きいならそちらを `k` にします (`k` は入れ換える場所なわけです)。次に、左右の子の大きいものより、親 (つまり入れ換えようかと思っていた節) が大きければ、もう入れ換える必要はないのでループを

抜けます。そうでなければ下へ行き、親と k 番を入れ換えたあと、k はその左の子にしてからループを周回します。

```
// heapsort1 --- heap sort using balanced bin-tree
#define P(i) ((i-1)/2)
#define L(i) (2*i+1)
#define R(i) (2*i+2)

static void iswap(int *a, int i, int j) {
  int x = a[i]; a[i] = a[j]; a[j] = x;
}
void pd(int *a, int i, int j) {
  for(int k = L(i); k <= j; ) {
    if(k < j && a[k] < a[k+1]) { ++k; }
    if(a[P(k)] >= a[k]) { break; }
    iswap(a, P(k), k); k = L(k);
  }
}
void heapsort1(int *a, int n) {
  for(int i = n-1; i >= 0; --i) { pd(a, i, n-1); }
  for(int i = n-1; i > 0; --i) { iswap(a, 0, i); pd(a, 0, i-1); }
}
```

ヒープソート本体は前述の通り簡単です。まず最初のループで、配列の後ろから前に無かって全ノードの押し下げをしてヒープを作ります。そのあと、最大ヒープの先頭から 1 つ取って後ろに起き、末尾を 1 減らす (実際には交換で両方一辺にやります)、そして押し下げを繰り返して行きます。

演習 7 ヒープソートの動作と実行時間を確認しなさい。またヒープソートの特徴として、このコードのように一気にヒープを作るのでなく、値を随時追加したり取り出したりして、取り出すと常に「現時点で入っているもののうち最大」が取れるようにもできることが挙げられる。ヒープソートのコードを流用してそのような機能を持つ情報隠蔽されたデータ構造 maxbuf を作ってみよ (中身はほぼ最大ヒープ)。それで n 個値を追加して n 個取り出した時もヒープソートができることになるので、その動作と時間も確認すること (単体テストすること)。

なお、「最大ヒープに 1 個値を追加する」場合は押し下げではなく、最後の位置に値を追加してから、その値を 2 分木の上に向かって (それより大きい値にぶつかるまでまたは根に着くまで) 親と交換していく押し上げ (push up) 操作が必要になります。押し上げの方が「左か右か」選択しないのでいくらか簡単です。

2.4 最大ヒープゲーム? option

配列を使った完全 2 分木による最大ヒープは直観的に分かりにくいので、これをゲームにしてみました。大きさを指定するとその大きさの配列に数値が埋められ、2 分木の配置で表示されます。

```
_0_
 1  2
 3  4  5  6
```

ここで「現在位置」は最初 0 番 (根) にありますが、コマンドで移動できます。現在位置と親との間で値を交換するコマンドもあります。タスクは、最大ヒープを構成した上で「移動」コマンドを使うと先頭要素が除去できるので、それを繰り返してなるべく速く全部除去することです。コマンドは次のものです。

- q — 終わる。いつの時点でも終わってよい
- l、r、p — 現在位置を左右の子/親に移す
- x — 現在位置と親の数値を交換する
- s — 配列をシャッフルする。ゲームなので
- m — 最大要素を除去し、末尾の要素をそこに置く。最大ヒープが構成できているときだけ動作する (メッセージで表示)。

コンパイル方法と動かし方は次の通り。何回か「s」を使ってよさげな配置になってから最大ヒープ化をするのがよいです。

```
% gcc8 heapgame.c -lncurses
% ./a.out 7  ← ヒープ中の数値の数
```

この部分はオプションで、詳しく説明すると大変なのでまあコードだけ掲載します。読めば読めると思います。

```
// heapgame.c --- construct heap and remove max screen game.
#include <stdio.h>
#include <stdbool.h>
#include <ncurses.h>
#include <stdlib.h>
#include <time.h>
#define P(i) ((i-1)/2)
#define L(i) (2*i+1)
#define R(i) (2*i+2)

#define MAXARR 128
static int a[MAXARR];
static int max, cur = 0;

static void pos(int n, int *x, int *y) {
  int y1 = 0, x1 = 0;
  for(int i = 1; i <= n; ++i) {
    if(i==1||i==3||i==7||i==15||i==31||i==63) { ++y1; x1 = 0; }
    else { ++x1; }
  }
  *x = x1; *y = y1;
}
static void upd() {
  char buf[10];
  int x, y;
  for(int i = 0; i < max; ++i) {
    pos(i, &x, &y); move(y*2+1, x*3+1);
    sprintf(buf, "%3d", a[i]); addstr(buf);
  }
  pos(cur, &x, &y); move(y*2+1, x*3+3);
}
void iswap(int *a, int i, int j) { int x = a[i]; a[i] = a[j]; a[j] = x; }
```

```
void shuffle(int *a, int size) {
  for(int i = 0; i < size; ++i) { iswap(a, i, rand()%(size-i)); }
}
bool checkheap(int *a, int size) {
  for(int i = 1; i < size; ++i) { if(a[P(i)] < a[i]) { return false; } }
  return true;
}
int main(int argc, char *argv[]) {
  max = atoi(argv[1]); if(max > MAXARR) { max = MAXARR; }
  for(int i = 0; i < max; ++i) { a[i] = i; }
  srand(time(NULL));
  initscr(); noecho(); cbreak(); system("stty raw"); clear();
  while(true) {
    upd(); refresh(); int ch = getch();
    switch(ch) {
case 'q': endwin(); return 0;                              // quit
case 'p': if(cur > 0) { cur = P(cur); } break;             // parent
case 'l': if(L(cur) < max) { cur = L(cur); } break;        // left-child
case 'r': if(R(cur) < max) { cur = R(cur); } break;        // right-chlid
case 'x': if(cur > 0) { iswap(a, cur, P(cur)); } break;    // excange
case 's': shuffle(a, max); break;                          // shuffle
case 'm': char *msg = "NG";                                // move top elt
          if(max <= 1) {
            msg = "GOAL!";
          } else if(checkheap(a, max)) {
            iswap(a, 0, max-1); --max; msg = "MOVED"; cur = 0;
          }
          clear(); move(15, 5); addstr(msg); clrtoeol(); break;
default: break;
    }
  }
}
```

演習 8 ゲームを何回かやって特性を体験しなさい。そのあと、次のような改訂をしてみなさい。

a. 現状では、押し下げの操作が複数キーを必要とするなど操作性がよくない。コマンドを追加して操作性を改善してみなさい。

b. 人間がやるのでなく、最大ヒープ化や押し下げをコンピュータに実行させなさい (スローモーションでようすが見られるとなおよいでしょう)。

c. 一定時間ごとに数値がヒープに追加されていき、消す速度を上回って上限を超えたらアウトみたいゲーム性を盛り込んでみなさい (一定時間ごと、は ncurses getch timeout などでぐぐって情報を調べる必要あり)。

d. 整列と関係あってもなくてもよいので、ncurses を使って何か面白いと思うものを作りなさい。

3　安定な整列と非安定な整列

だいぶ今更な話題なのですが、整列には「安定な整列」と「非安定な整列」があります。安定な整列とは、「キーとして同じ値の要素が複数含まれていた場合、整列結果における要素の並び順が元の並び順と同じままである」ものを言います。例えば図 6 を見てください。上の列を数値 (キー) の昇順に並べるとします。「3」の要素は 3 つあり、元の列では○、△、□の順に並んでいます。安定な整列では整列後もこの順番が維持されますが、非安定な整列では維持されません。

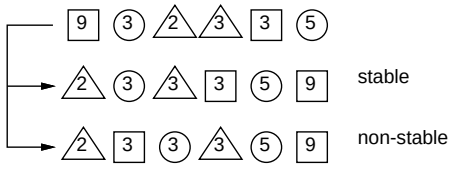

図 6: 安定な整列の概念

どのような整列でも、「元の項目番号」を振っておいて、それを追加のキー (もともとのキーが同じ値だったとき使う) にすれば、安定な整列にできます。ただ、それは結構面倒なので、安定性が必要なら最初から安定なアルゴリズムを利用した方が楽でしょう。

これまでのアルゴリズムで安定性について調べたいとします。それにはたとえば、これまでの例で整数を整列していたものを、すべて次のようなレコード型の値にします (unsigned short では最大は 65536 になりますが、通常の実験には十分でしょう)。

```
struct sortval { unsigned short key, seq; };
```

そして実験時に、key には乱数を入れますが、範囲を狭くして同じキーが複数現れるようにします。そして seq には一連番号を入れます。整列が終わってから、「隣接値でキーが同じで連番が逆になるもの」がないか調べればよいでしょう。

演習 9　ここまでに学んだ (またはこの後出て来るものでもよい) 整列アルゴリズムからなるべく多く選び、安定性について検討しなさい。また、実際に上記の方法で試してみて、自分の検討した結論が合っているかどうか確認しなさい。実験に用いたプログラムをきちんと説明・掲載すること。

本日の課題 11a

「演習 1」〜「演習 9」で動かしたプログラム 1 つを含むレポートを本日中 (授業日の 23:59 まで) に久野までに提出してください。

1. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
2. プログラムどれか 1 つのソースと「簡単な」説明。
3. レビュー課題。提出プログラムに対する他人 (ペア以外) からの簡単な (ただしプログラムの内容に関する) コメント。
4. 以下のアンケートの回答。

   Q1. さまざまな整列手法からいくつくらい理解しましたか。

   Q2. 最大ヒープとは何か分かりましたか。

   Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 11b

注意: B 課題は今回の 11b で最後で、次回からは A 課題のみとなります。期末も近くて皆様も大変でしょうから。それで、11B の期間は通常より 1 週間長く取っていますのでそのつもりで力作をどうぞ。

「演習 1」〜「演習 9」(ただし 11a で提出したものは除外、以後も同様) の (小) 課題から選択して 2 つ以上プログラムを作り、レポートを提出しなさい。できるだけ複数の演習から選ぶこと。レポートは「# 13 授業前日」23:69 を期限とします。

1. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
2. 1 つ目の課題の再掲 (どの課題をやったか分かればよい)、プログラムのソースと「丁寧な」説明、および考察 (課題をやってみて分かったこと、分析、疑問点など)。
3. 2 つ目の課題についても同様。
4. 以下のアンケートの回答。

   Q1. 自力で書ける整列アルゴリズムは何と何でしょう。
   Q2. 整列が安定かどうか判断できるようになりましたか。
   Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。