ヒープソートと
マージソート

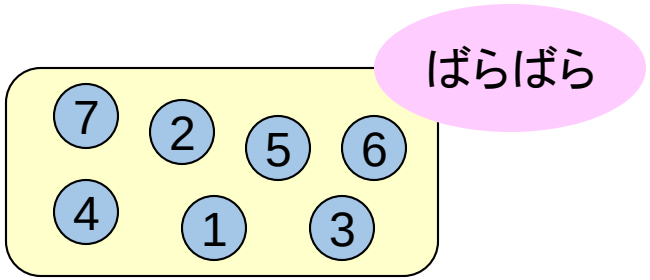
この回の要点

- ヒープソート
  - 二分探索木構造を使って並べ替えを行う
  - 通常の二分探索木では贅沢→ヒープ構造
    - 「親は子よりも小さい」という条件のみ
    - 探索はできないが、挿入・削除が簡単
    - 計算量は$O(n \log n)$である
  - ヒープは配列を使用すると効率よく実装可能
- マージソート
  - 並んでいる2つのリストを合体させて並べるのは簡単
  - 全体を小さく分けていき、その後、合体させる
    - 2分割して、合体させると計算量は$O(n \log n)$である
  - 定数項が小さく高速

ヒープソート

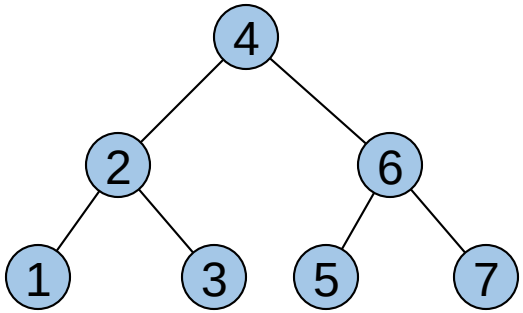
- 単純なソートアルゴリズム
  - データどうしを比較し、必要ならば交換
  - データは線形な構造を持つ
- ヒープソートアルゴリズム
  - 2つのメソッドを持つ探索木を利用
    - データを挿入する（insert）
    - 最小のデータを取り出す（removeMin）
  - ヒープソートの手順
    - すべてのデータをinsertによって探索木に挿入する
    - removeMinにを繰り返すことによって、木から小さい順に取り出す
  - 探索木の計算量
    - 単純な二分探索木 → $O(n \log n)$だが、最悪は $O(n^2)$
    - 平衡木 → 常に$O(n \log n)$
    - 平衡木を用いれば、insertとremoveMinのどちらも$O(n \log n)$で実行可能であり、トータルも$O(n \log n)$で実行できる

ヒープソートの原理



例えばAVL木を構成

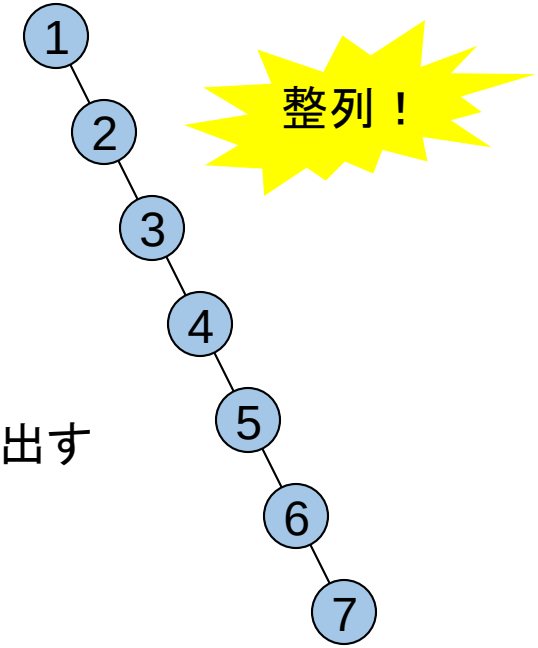
$O(n \log n)$



小さい順に取り出す

（通りがけ順）

$O(n \log n)$



整列！

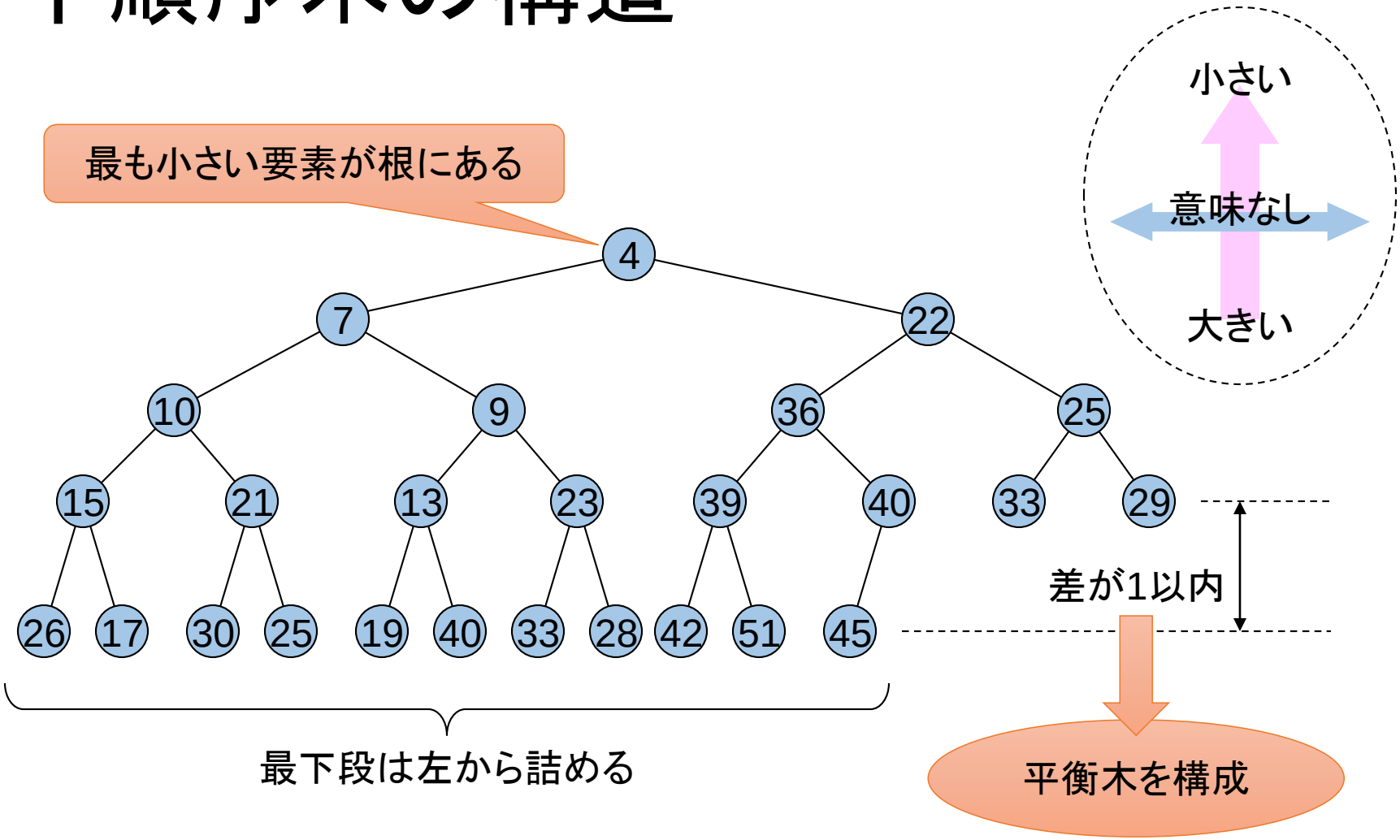
優先順序付待ち行列

- AVL木やB木などを整列に使うことは・・・
  - 間違いではないが、ぜいたくな方法
  - 整列に必要な操作は、insertとremoveMinのみ
  - 探索（search）は必要ない
  - 無駄が多い方法（＝オーダーは同じでも定数部分が大きい）
- insertとremoveMinだけならもっと良い構造がある
  - 優先順位付待ち行列（priority queue）
  - 通常の待ち行列（queue）は、入れた順に取り出される
  - 優先順位付待ち行列は、優先順位の高い要素から取り出される（入れた順番は関係ない）
  - 単なる待ち行列を一般化したもの
    - 通常の待ち行列は、要素を入れた順番を優先順位としたものに相当（番号が小さいほど優先順位が高いとする）
    - 大きいほど優先順位が高いとすれば、スタックになる
  - 探索キーを優先順位として使用する

半順序木

- 半順序とは
  - 順序関係の一種
  - すべての要素間で順序が定義されているとは限らない順序関係
    - （←→全順序）
  - 比較できない要素がある
- 半順序木（partial ordered tree）とは
  - 優先順位付待ち行列の実現のひとつ
  - 「親は子より小さい（か等しい）」という条件の木構造（二分木）
  - 探索木の条件よりもゆるい（子どうしの関係はない）
  - 構造が探索木よりも簡単→処理の手間が小さい
- 平衡木となるようにする
  - 深さの差は1以内
  - 最下段の要素は左に詰める
  - 挿入、削除を$O(\log n)$に抑えるため
- 挿入（insert）処理を持つ
- もっと小さい要素を削除（removeMin)処理を持つ

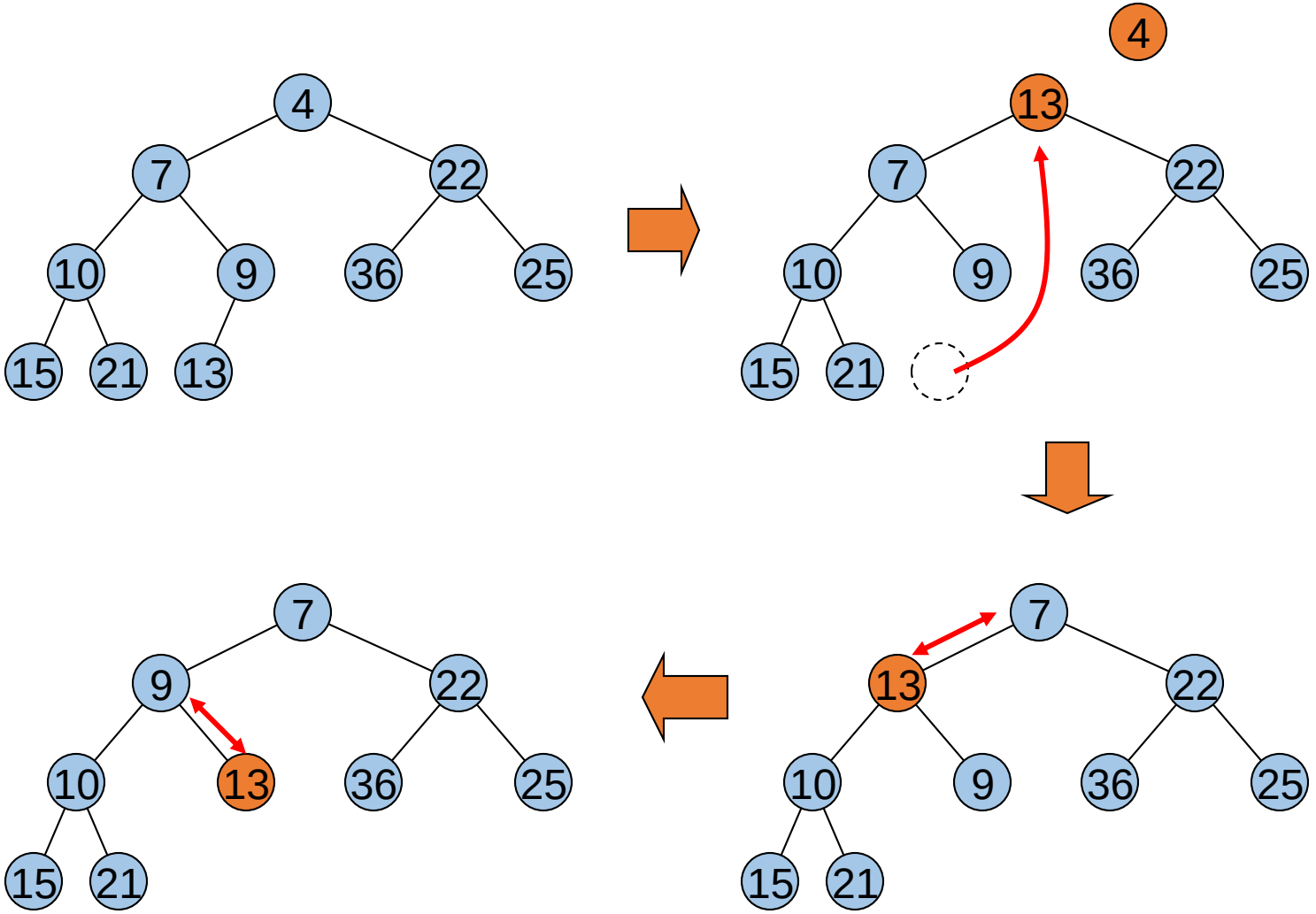
半順序木の構造


小さい
意味なし
大きい
最も小さい要素が根にある
4
7
22
10
9
36
25
15
21
13
23
39
40
33
29
26
17
30
25
19
40
33
28
42
51
45
差が1以内
最下段は左から詰める
平衡木を構成

removeMin処理1

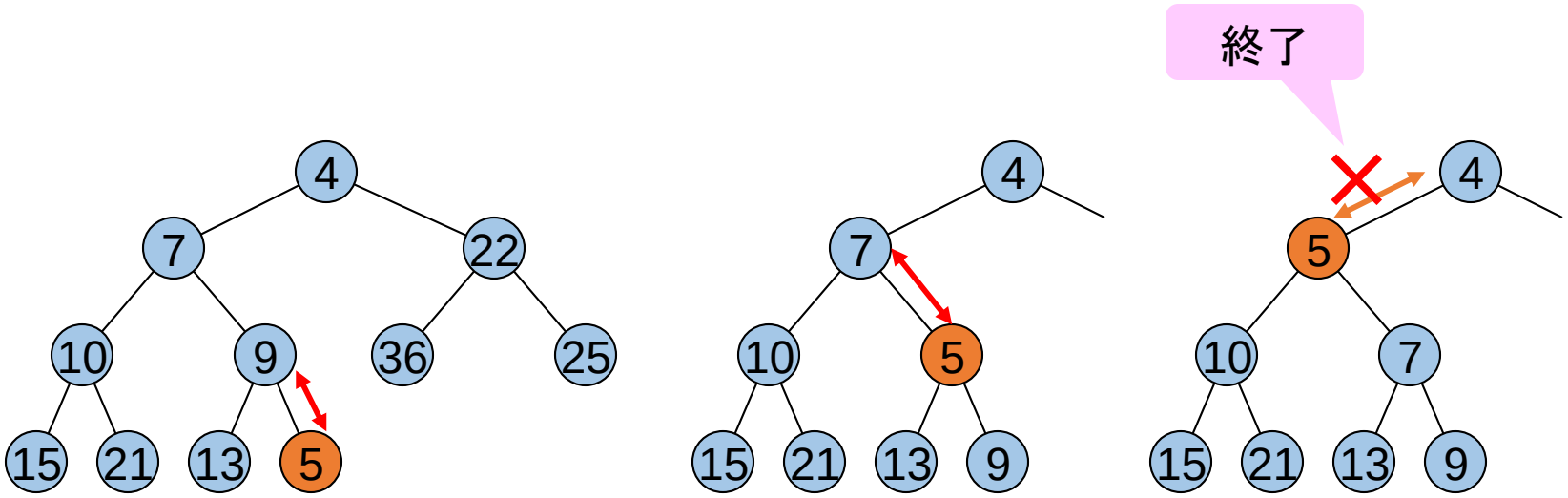
- 半順序木の最小要素は根ノード
- 単に根ノードを取り出すと、木構造が崩れる
- 根ノードを削除した後、半順序木を再構成
- 手順
  1. 根ノードを取り外す
  2. 最下段の最も右のノードを根に移動する
  3. 親が子よりも大きければ、子の小さいほうと親とを交換
  4. これを、根から葉に向かって繰り返す
  5. 親が小さくなれば終了

removeMin処理2



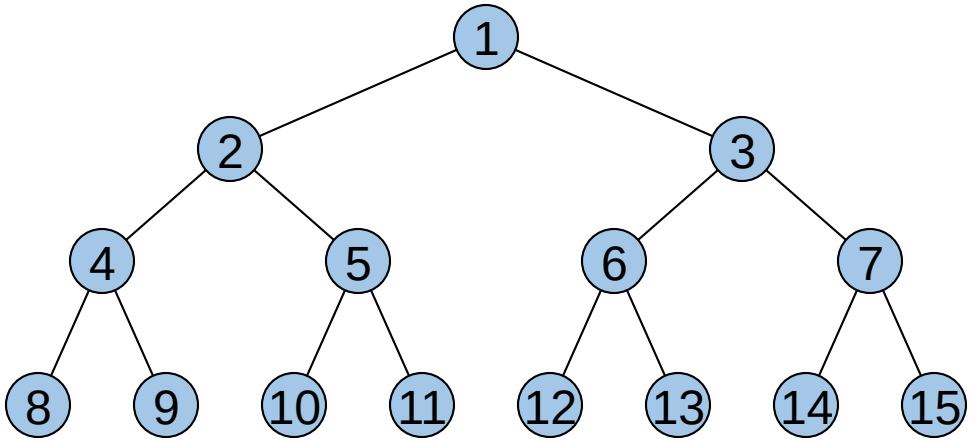
insert処理

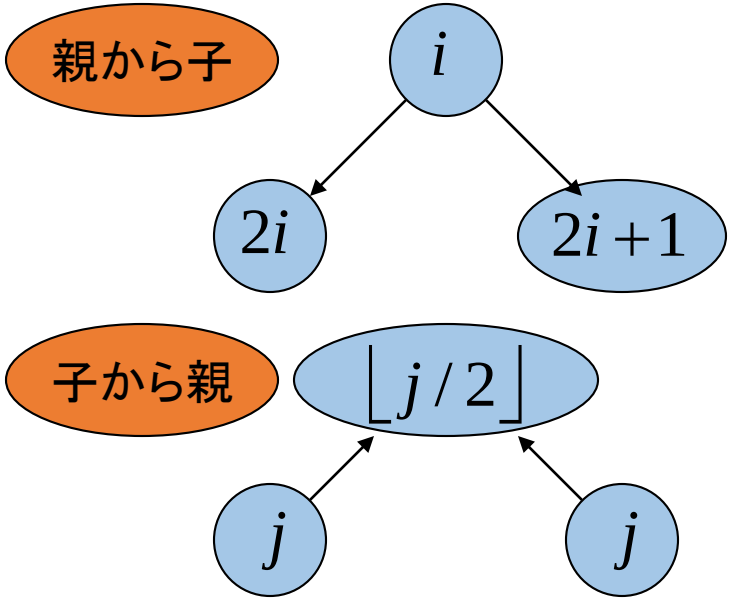
1. 挿入する要素を、最下段の空いている右端に挿入
   - もし、最下段がいっぱいなら、次の段の左端
2. 子が親より小さければ交換
3. これを葉から根に向かって繰り返す
4. 親が小さくなれば終了



ヒープ構造1

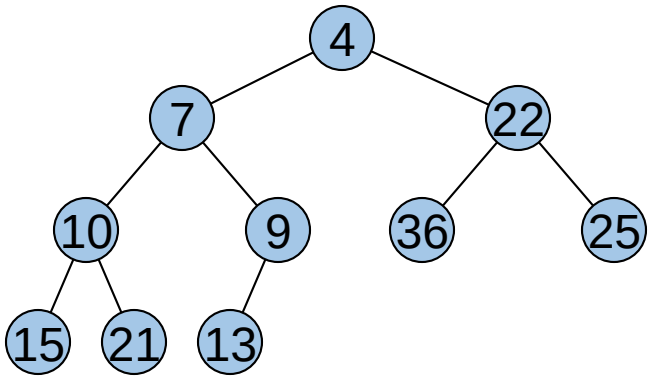
- ヒープには2つの意味
  - 任意のタイミングで割り当て・解放可能なメモリ（ヒープメモリ）
  - 半順序木を配列を使って実現したもの（ヒープ構造）
- 半順序木は木構造（リンクを使用した）でも実現可能だが、ヒープを使うと効率よく実現できる
- 完全二分木のノードに1から番号を振ると規則性がある

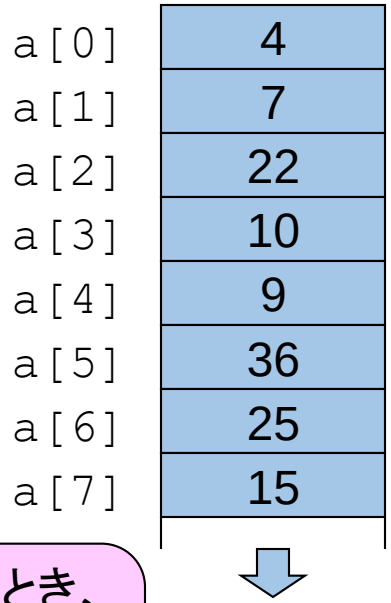


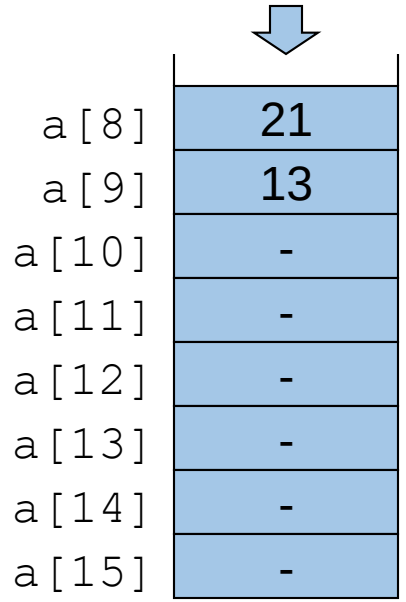


ヒープ構造2

- 半順序木は最下段は左に詰まっている
  - 配列aによって表現した場合、a[0]～a[n-1]







C言語の配列を使ってヒープを実現するとき、
ノード a[i] の子は何か？
ノード a[j] の親は何か？
が、問題

ヒープ構造の実装

- 汎用のヒープ構造Heap型
  - Heap.hとHeap.ccで実装する
  - C言語の配列は0～である
    - i を i+1 に変えて計算し、最後に1を引く
    - 左の子2i → 2(i+1)-1=2i+1
    - 右の子2i+1 → 2(i+1)+1-1=2i+2
    - 親 (int)(i/2) → (int)((i+1)/2)-1
  - 操作
    - void insert(Heap *h,void *d,int(*)(void*,void*))
    - void *removeMin(Heap *h,int(*)(void*,void*))
  - 挿入、取り出しには比較が必要
    - 比較関数int(*comp)(void*,void*)を与える

Heap.h

```
#ifndef __Heap__h
#define __Heap__h
/***
 *** ヒープ構造
 ***/
#include <stdio.h>
#include <stdlib.h>

// ヒープ型
typedef struct {
  void **heap;
  int size;
  int num;
} Heap;

// プロトタイプ宣言
Heap *makeHeap(int);
void free(Heap*);
void insert(Heap*,void*,int(*)(void*,void*));
void *removeMin(Heap*,int(*)(void*,void*));

#endif // __Heap__h
```

Heap.cc

```
/***
 *** ヒープの実装
 ***/
#include "Heap.h"

// 生成
Heap *makeHeap(int n){
  Heap *h=(Heap*)malloc(sizeof(Heap));
  h->heap=(void**)malloc(sizeof(void*)*n);
  h->size=n;
  h->num=0;
  return h;
}

// 破棄
void free(Heap *h){
  free(h->heap);
  free((void*)h);
}
```

続く

Heap.cc

```
// 挿入
void insert(Heap *h,void *d,int(*comp)(void*,void*)){
  if(h->num>=h->size) return;          // ヒープが満杯
  h->heap[h->num]=d;                   // ヒープの末尾に入れる
  for(int i=h->num;i>0;){              // i:自分の番号
    int j=(int)((i+1)/2)-1;            // j:親の番号
    if(comp(h->heap[i],h->heap[j])>=0)
      break;                           // 自分iが親j以上なら終わり
    void* t=h->heap[i];                // 自分と親を入れ替える
    h->heap[i]=h->heap[j];
    h->heap[j]=t;
    i=j;                               // 親を自分にしてもう一度
  }
  h->num++;                            // データ数を増やす
}
```

続く

Heap.cc

```
// 最小のデータを外して返す
void *removeMin(Heap *h,int(*comp)(void*,void*)){
  if(h->num==0) return NULL;              // ヒープが空ならNULLを返す
  void *min=h->heap[0];                   // 最小要素(＝根ノード)と取り出す
  h->heap[0]=h->heap[h->num-1];           // 最後のノードを根ノードに移動する
  for(int i=0;i<h->num;){                 // i: 自分の番号
    int l=2*i+1;                          // l: 左子ノードの番号
    int r=2*i+2;                          // r: 右子ノードの番号
    if(l>=h->num) break;                  // 左子ノードがないなら終了
    int j=r;                              // j: 比較対象を右子ノードにしておき、
    if(r>=h->num ||                       // もし右がないか、または
       comp(h->heap[r],h->heap[l])>0)     //    右の方が大きければ、
      j=l;                                // 比較対象jを左子ノードにする
    if(comp(h->heap[i],h->heap[j])>0){    // iとjを比較してiの方が大きければ、
      void *t=h->heap[i];                 // iとjを入れ替える
      h->heap[i]=h->heap[j];
      h->heap[j]=t;
      i=j;                                // 子を自分にしてもう一度
    }
    else                                  // そうでないなら、
      break;                              // 終了
  }
  h->num--;                               // データ数を減らす
  return min;                             // 最小データを返す
}
```

ヒープソートの実装

- Sort.h と Sort.cc に実装する
- Heap構造を利用したソート
  - Heapを構築し、そこから取り出すだけ
  - 関数
    - `sortHeap(void*,int,int(*)(void*,void*))`

Sort.h

```
#ifndef __Sort__h
#define __Sort__h
/***
 *** 並べ替え用
 ***/
#include <stdio.h>
#include <stdlib.h>

// プロトタイプ宣言
void sortBubble(void*,int,int(*)(void*,void*));
void sortSelection(void*,int,int(*)(void*,void*));
void sortInsertion(void*,int,int(*)(void*,void*));
void sortHeap(void*,int,int(*)(void*,void*));

int getCompCount();
int getExchgCount();

#endif // __Sort__h
```

Sort.cc

```
#include "Heap.h"

…(略)…

// ヒープソート
void sortHeap(void *d0,int s,int (*comp)(void*,void*)){
  void **d=(void**)d0;
  Heap *heap=makeHeap(s);

  // すべてのデータをヒープに入れて、
  for(int i=0;i<s;i++)
    insert(heap,d[i],comp);

  // 小さい順に取り出す
  for(int i=0;i<s;i++)
    d[i]=removeMin(heap,comp);
}

…(略)…
```

ヒープソートのテスト

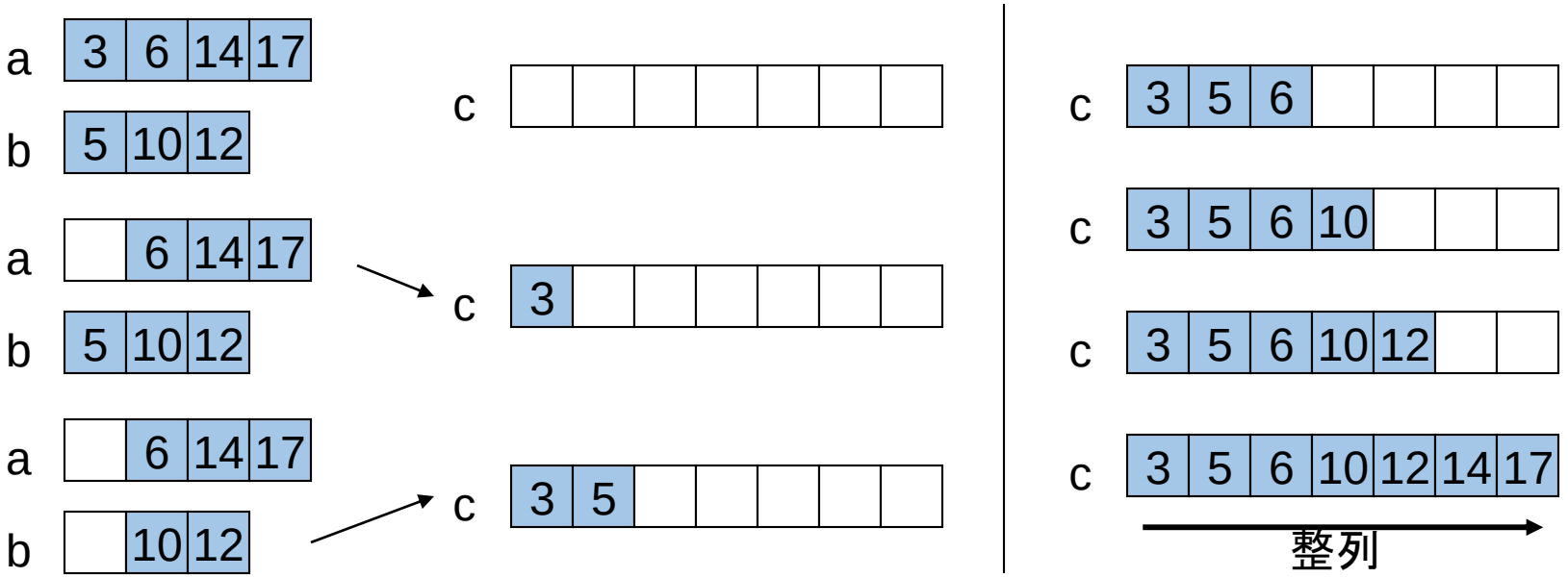
- TestSort1.ccを変更する
  - sortHeap()を使うだけ
- 結果

```
$ ./TestSort
山田(18)-森(55)-中村(33)-石田(27)-東村(31)-牧(12)-
牧(12)-山田(18)-石田(27)-東村(31)-中村(33)-森(55)-
```

- 連続実験 TestSort2.cc
  - マージソートと一緒に行う
  - 「// 並べ替え」の部分に、case文を追加する
    - case 3: sortHeap(…); break;
    - case 4: sortMerge(…); break;

マージ（併合）

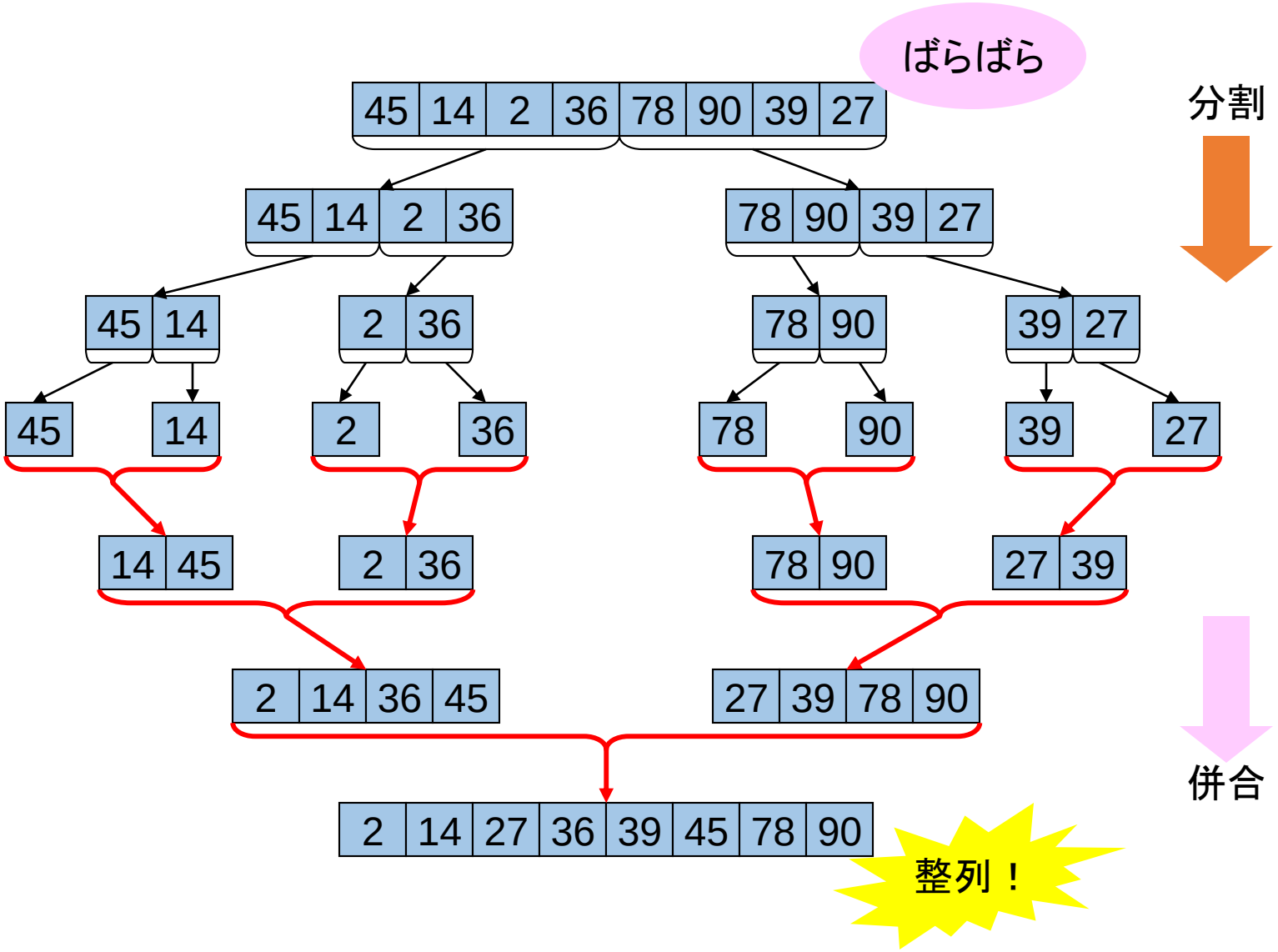
- マージ（merge）とは？
  - 日本語では併合（マージソート＝併合ソート）
  - 整列済みの2つのデータ列a,bを、整列済みの1つのデータ列cにまとめること
- マージの手順
  - 列a,bの先頭の小さい方を取り、列cの末尾に追加する



マージソート

- 原理
  - データの半分ずつを整列しておき、それらをマージすると、全体が整列する
  - この作業を再帰的に繰り返す
- 手順
  - データ列を、真ん中で2つの部分列a,bに分割する
  - 部分列a,bをそれぞれ整列する
  - a,bをマージする
- 利点
  - シーケンシャルにデータにアクセスするため、外部整列に向く
  - リンクリストの整列にも向く
- 欠点
  - 配列を整列する場合は、同じ大きさの作業配列が必要
  - 配列間のコピーに時間がかかる（クイックソートより不利）

マージソートの流れ


ばらばら
分割
45 14 2 36 78 90 39 27
45 14 2 36
78 90 39 27
45 14
2 36
78 90
39 27
45
14
2
36
78
90
39
27
14 45
2 36
78 90
27 39
2 14 36 45
27 39 78 90
2 14 27 36 39 45 78 90
併合
整列！

マージソートの実装

- 3つの関数を使う
  - merge(d0,n1,n2,w0,comp)
    - 配列d0のn1からn2までの範囲をマージする
  - mergeSort(d0,n1,n2,w0,comp)
    - 配列d0のn1からn2までの範囲を再帰的にマージソートする
    - m=(n1+n2)/2とし、
    - n1からmまでをマージソートし、
    - m+1からn2までをマージソートし、
    - n1からn2の範囲をマージする
  - sortMerge(d0,s,comp)
    - 配列d0全体を並べ替える
- 外から見えるのはsortMerge()だけ

Sort.h

```
#ifndef __Sort__h
#define __Sort__h
/***
 *** 並べ替え用
 ***/
#include <stdio.h>
#include <stdlib.h>

// プロトタイプ宣言
void sortBubble(void*,int,int(*)(void*,void*));
void sortSelection(void*,int,int(*)(void*,void*));
void sortInsertion(void*,int,int(*)(void*,void*));
void sortHeap(void*,int,int(*)(void*,void*));
void sortMerge(void*,int,int(*)(void*,void*));

int getCompCount();
int getExchgCount();

#endif // __Sort__h
```

Sort.cc

```
// ヒープソート
void sortHeap( … ){ …(略)… }

// マージする
void merge(void *d0,int n1,int n2,void *w0,
                                  int (*comp)(void*,void*)){
  void **d=(void**)d0;
  void **w=(void**)w0;
  for(int i=n1;i<=n2;i++) w[i]=d[i];  // 作業用配列にコピー
  int m=(n1+n2)/2;
  int j=n1;
  int k=m+1;
  for(int i=n1;i<=n2;i++){
    if(j>m) d[i]=w[k++];
    else if(k>n2) d[i]=w[j++];
    else if(comp(w[j],w[k])<0) d[i]=w[j++];
    else d[i]=w[k++];
  }
}
```

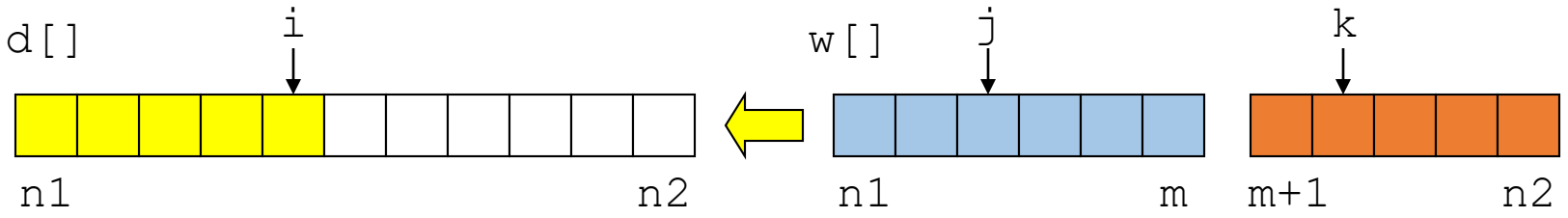
続く

Sort.cc

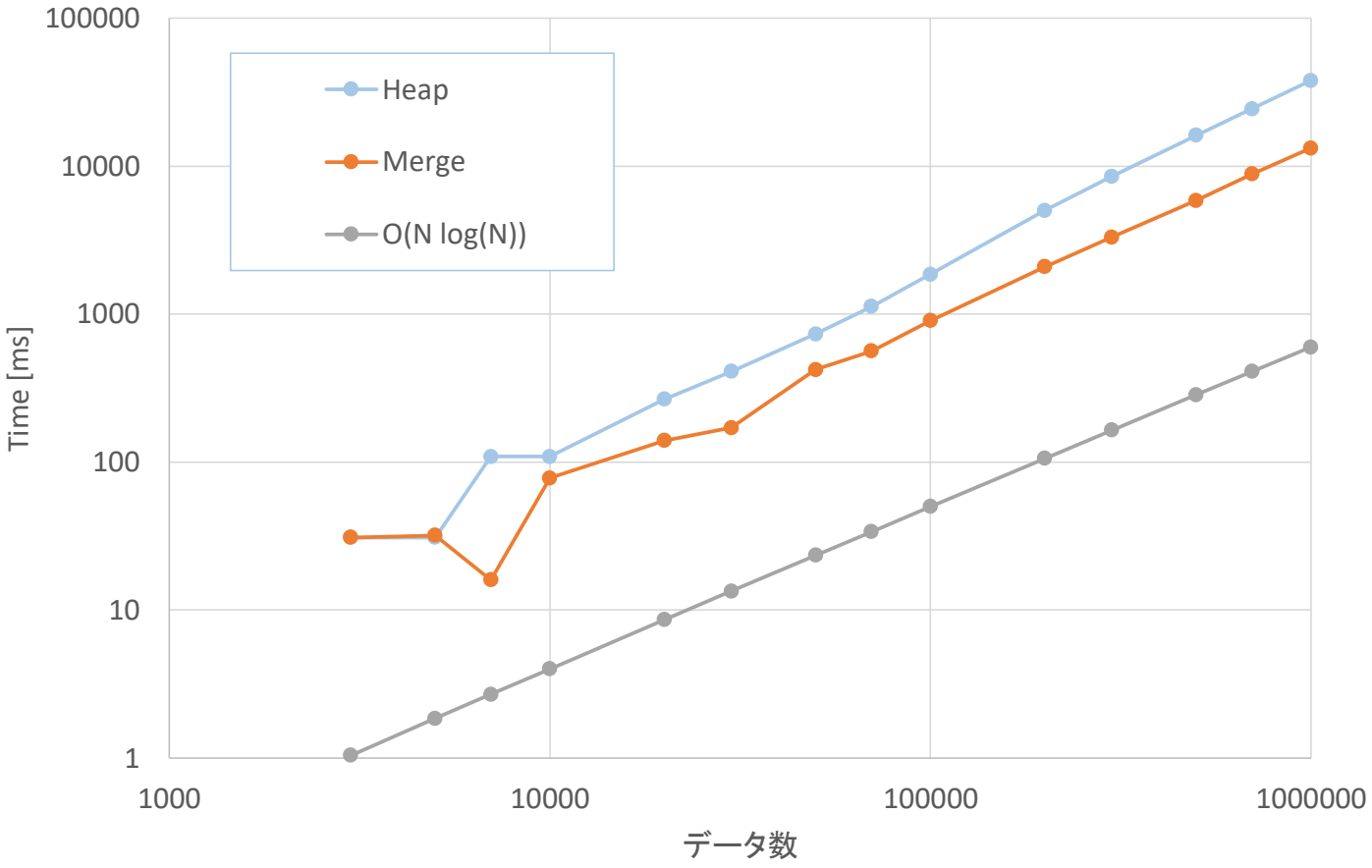
```
// マージソート
// 配列dのn1からn2までの範囲を並べ替える
// 配列wは作業用
void mergeSort(void *d0,int n1,int n2,void *w0,
                          int (*comp)(void*,void*)){
  if(n1==n2) return;
  int m=(n1+n2)/2;
  mergeSort(d0,n1,m,w0,comp);
  mergeSort(d0,m+1,n2,w0,comp);
  merge(d0,n1,n2,w0,comp);
}

// マージソート
void sortMerge(void *d0,int s,int (*comp)(void*,void*)){
  void **w=(void**)malloc(sizeof(void*)*s);   // 作業用領域
  mergeSort(d0,0,s-1,w,comp);
  free((void*)w);
}
```

関数merge()の動作

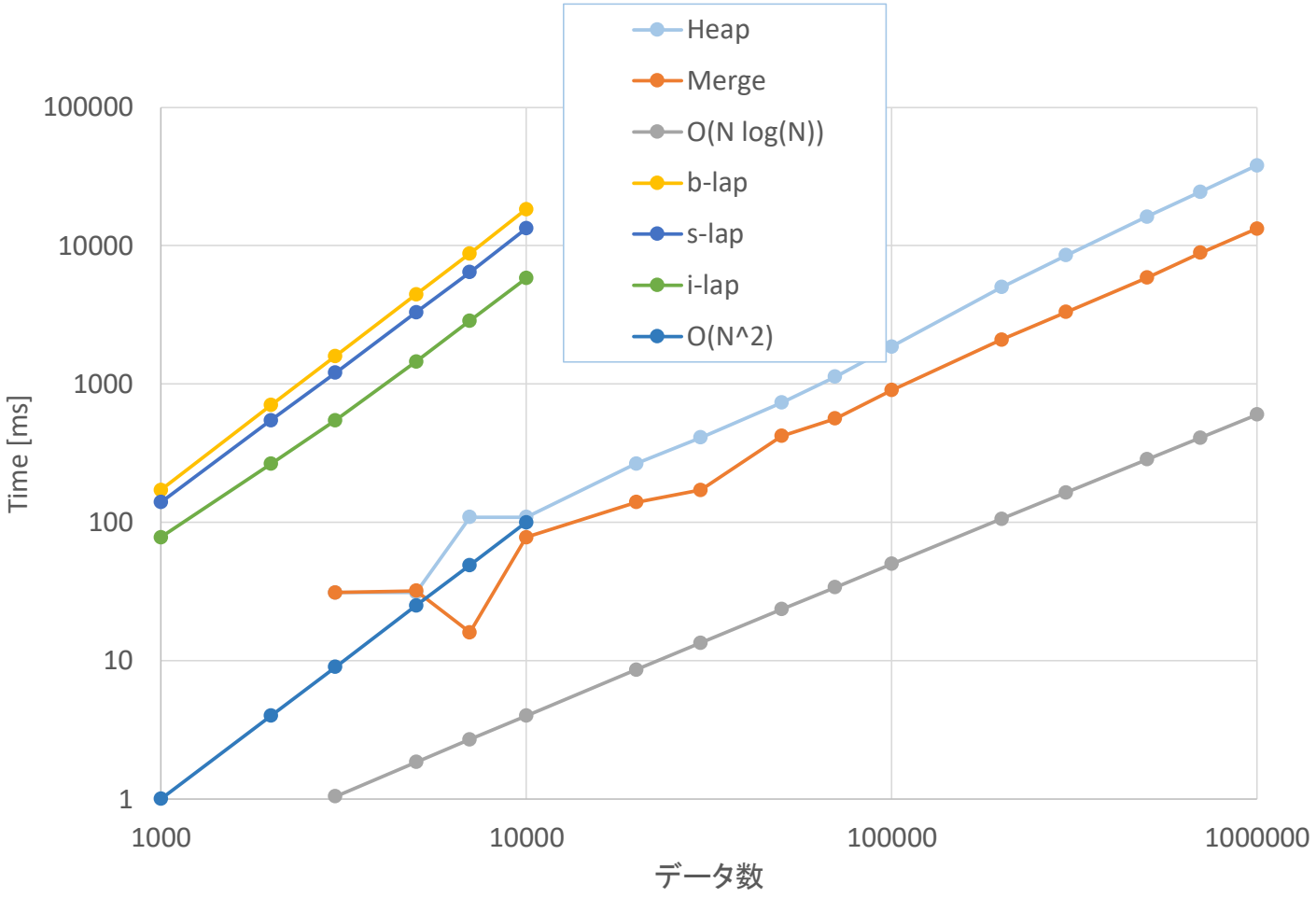
```
/*** 配列dのn1からn2までの半分ずつをマージする ***/
  for(int i=n1;i<=n2;i++) w[i]=d[i];  // 作業用配列にコピー
  int m=(n1+n2)/2;
  int j=n1;
  int k=m+1;
  for(int i=n1;i<=n2;i++){
    if(j>m) d[i]=w[k++];
    else if(k>n2) d[i]=w[j++];
    else if(comp(w[j],w[k])<0) d[i]=w[j++];
    else d[i]=w[k++];
  }
```



ヒープソートとマージソートの性能



遅い並べ替えとの比較


Heap
Merge
O(N log(N))
b-lap
s-lap
i-lap
O(N^2)
100000
10000
1000
100
10
1
Time [ms]
1000
10000
100000
1000000
データ数

課題190107

1. 以下のデータをこの順にヒープに追加したとき、構築される半順序木を示せ。（最終結果のみ）
   - 12,7,2,16,10,8,3,5,11,6,1,4,9,13,14,15
2. 以下のデータをマージソートで並べ替えるとき、マージが行われる様子を示せ。
   - 12,7,2,16,10,8,3,5,11,6,1,4,9,13,14,15

- 提出方法：
  - レポートはワードで作成し、メールに添付すること。
  - ファイル名は scXXXXXX-al190107.docx とすること。レポートには学籍番号と氏名を必ず書くこと。
  - メールで渕田まで送付すること。
    - 提出先：fuchida@ibe.kagoshima-u.ac.jp
  - メールタイトル：”アルゴリズム課題190107”
  - メール本文にも学籍番号と氏名を必ず書くこと。
  - 期限：2019年1月13日（日）24:00

ヒープソートとマージソート

終了