13 2分木,push-downスタック,待ち行列

動的データ構造を用いると

自由に色々なデータ構造を構成できるが、データへのアクセスに手間がかかることも多く、デバッグ等も難しくなる。例えば、

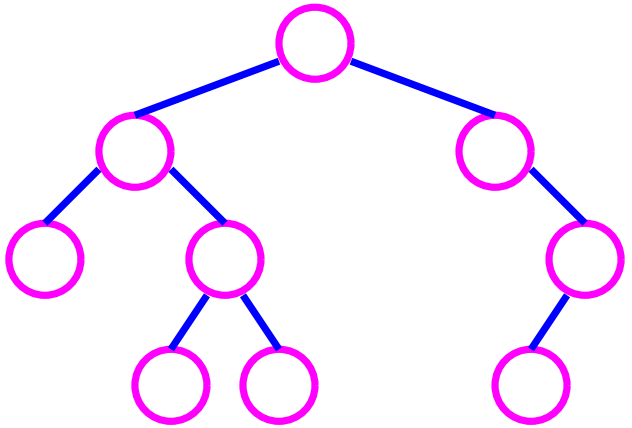
- 使わなくなった領域を解放し忘れると知らず知らずの内に使えるメモリが少くなってゆく。(メモリ洩れという。) また逆に、
- まだ使っている領域を間違って解放してしまうと、同じ領域を2重に使うために訳の分からない結果になる。

⇨ この節では静的に (i.e. 宣言によって) 確保された領域であっても色々な用途に使えることを例示する。

データ量の上限が予め分かっているなら、
静的な領域を用いて2分木構造等を動的に表すことが出来る。

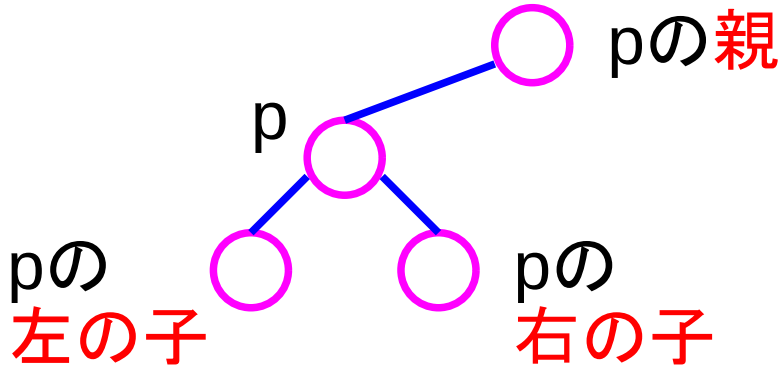
13–1 2分木

2分木： 一般に、(データ構造に限らず) 次のような形の階層構造を2分木と呼ぶ。



この様な構造で、やはり
節点 ··· ○
枝 ··· 節点と節点を結ぶ線

p の親 ··· p の上方にある節点
p の左の子 ··· p の左下にある節点
p の右の子 ··· p の右下にある節点
p の子 ··· p の左右の子



根 ··· 親を持たない節点
葉 ··· 子を持たない節点
節点のレベル ··· 根とその節点を結ぶのに必要な枝の本数
2分木の高さ ··· レベルの最大値

2分木の表現 :　2分木をプログラムの中で表す方法としては、...

- ポインタを用いる方法
  節点のデータを入れる小さな記憶領域を節点毎に動的に確保し、それらをポインタで繋ぐ。　(2分木の各枝をポインタで表す。)

- 配列で表す方法 ···(2分木の大きさの上限が決まっている場合に可)
  次の様に節点に番号付けすれば、2分木の各節点と非負整数を1対1に対応づけることが出来る。

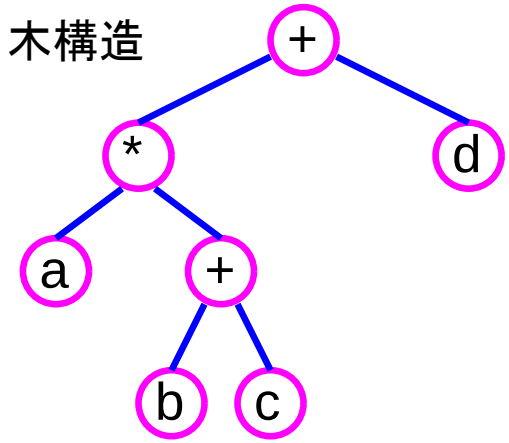
$$\begin{cases} \text{根の番号} = 0, \\ \text{節点 p の番号が i なら} \\ \quad \text{pの左の子の番号} = 2\text{i}+1, \\ \quad \text{pの右の子の番号} = 2\text{i}+2, \\ \quad \text{pの親の番号} \quad = \lfloor (\text{i-1})/2 \rfloor \end{cases}$$

⇨ 節点の番号と配列の添字を対応させて

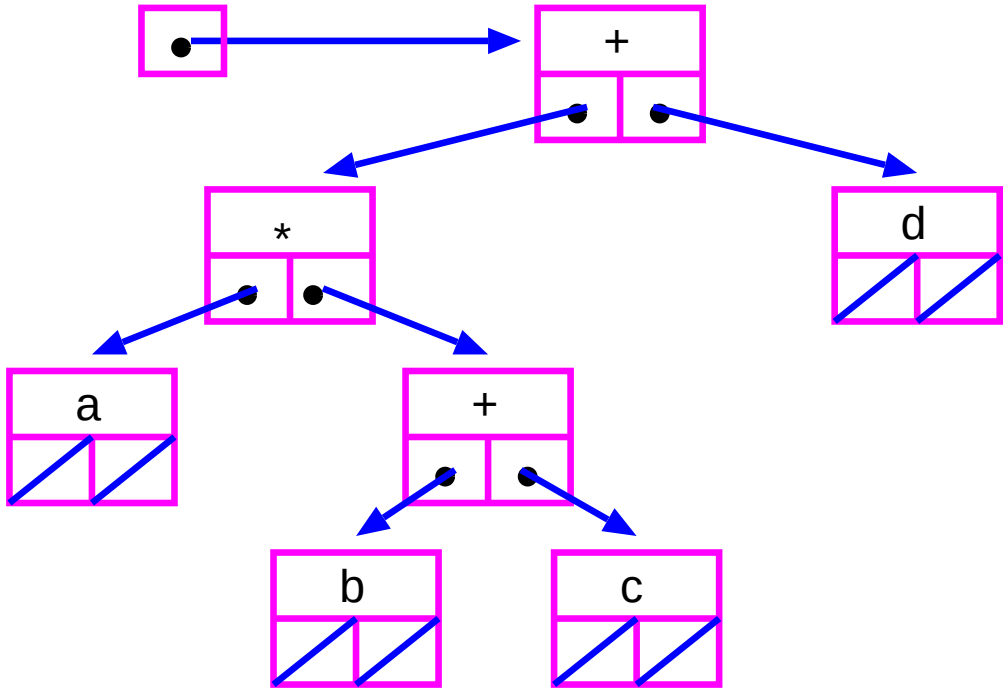
$$\text{a[i]} = \text{番号 i の節点のデータ}$$

とすればよい。

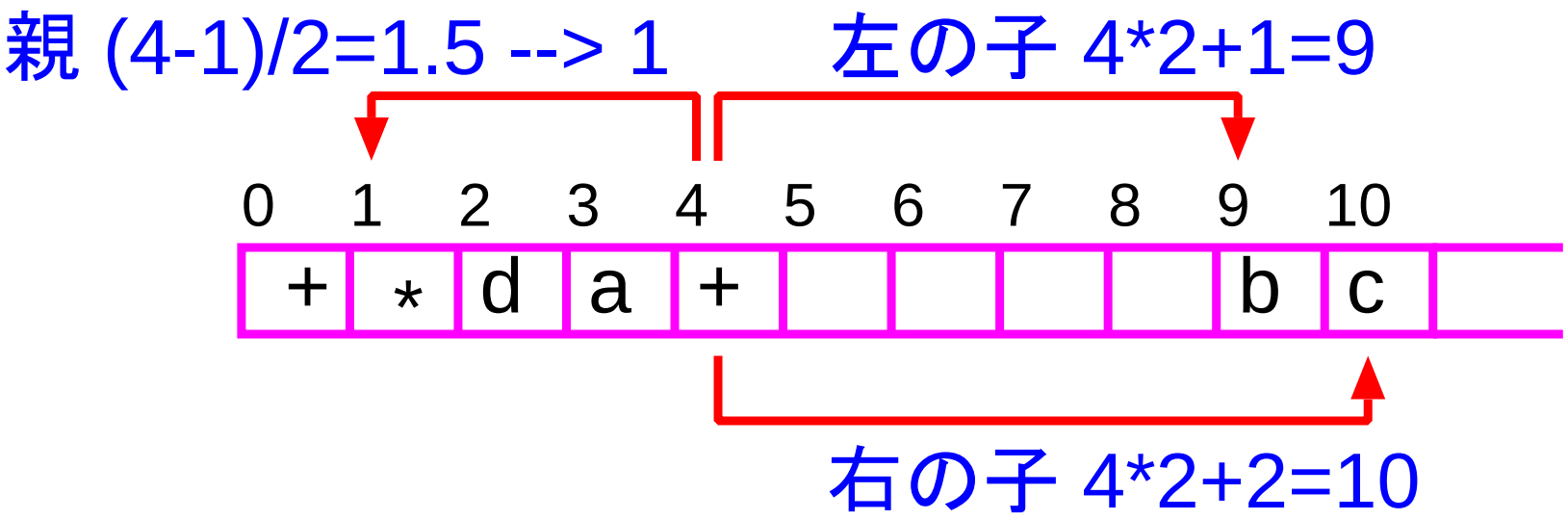
例13. 1 (2分木の表現)　算術式 `a*(b+c)+d` の構造は次の様な2分木で表せる。

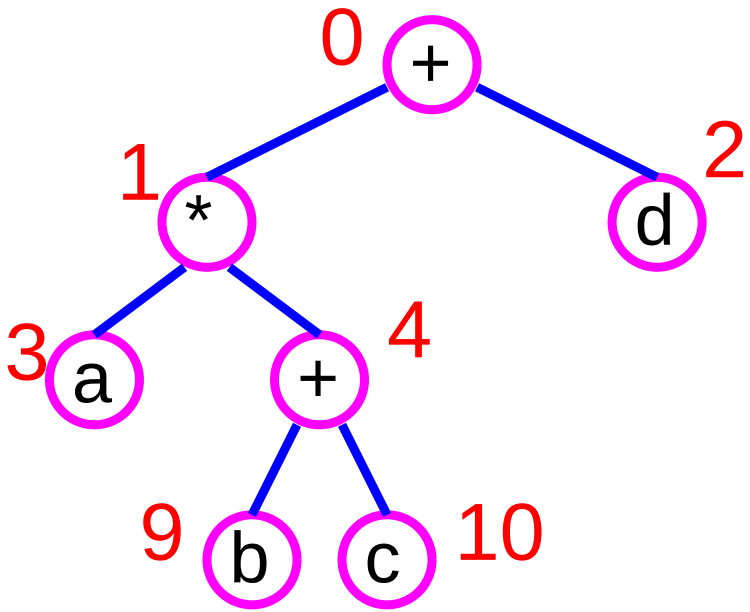


この2分木は、 ポインタを用いて表すと 次の様になる。



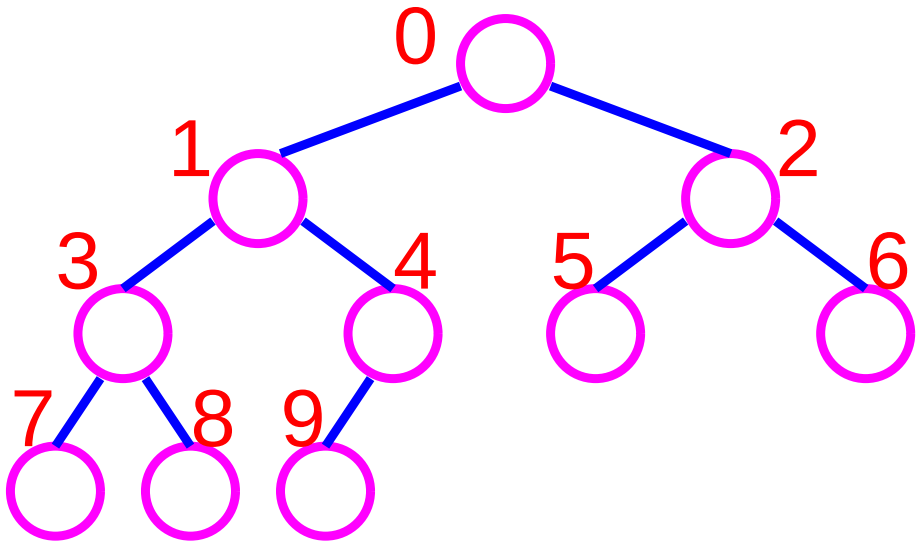
また、 配列で表すと 次の様になる。





完全2分木

高さ h の2分木において、レベル i (<h) の節点が 2^i 個(満杯)存在しレベル h の節点が左詰めに並んでいる時、この2分木を特に完全2分木という。例えば、10個の節点からなる完全2分木は次の様な構造を持つ。



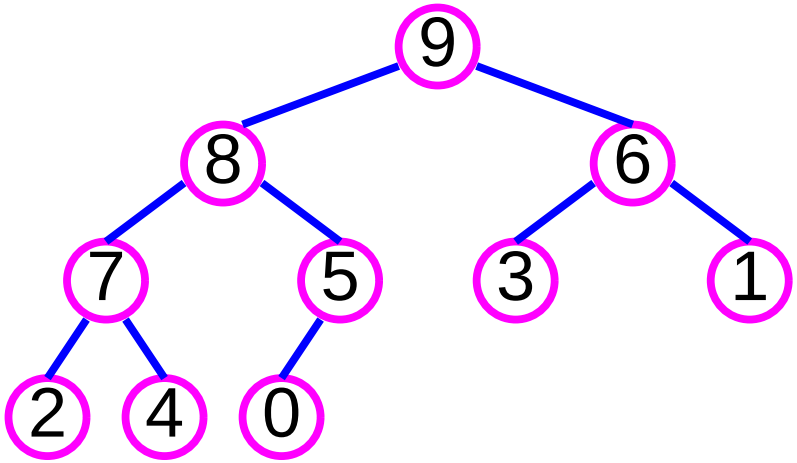
⇨ 節点数 n の完全2分木を配列 a で表す場合、
各節点の情報は a[0] ~ a[n-1] に格納される。
(途中に穴は出来ない。)

ヒープ : 完全2分木において、各節点に数値データがラベル付けされていて、どの枝についても

(親の数値データ) $\geq$ (子の数値データ)

が成り立つ時、この完全2分木、あるいは、この完全2分木を表す配列を特にヒープ(または 整列2分木)という。

例えば、各節点に非負整数をラベル付けした完全2分木



あるいは、この完全2分木を表す配列

0	1	2	3	4	5	6	7	8	9
9	8	6	7	5	3	1	2	4	0

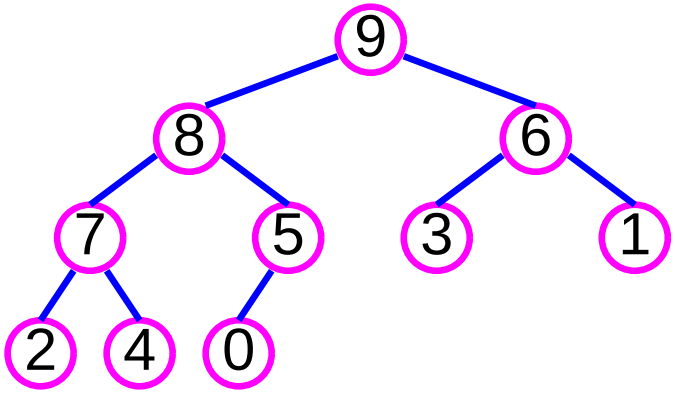
はヒープである。

例題13. 2 (heapsort)　int型配列に入ったデータをヒープソート手法で小さい順に並べ替えるための汎用のモジュールを作成せよ。

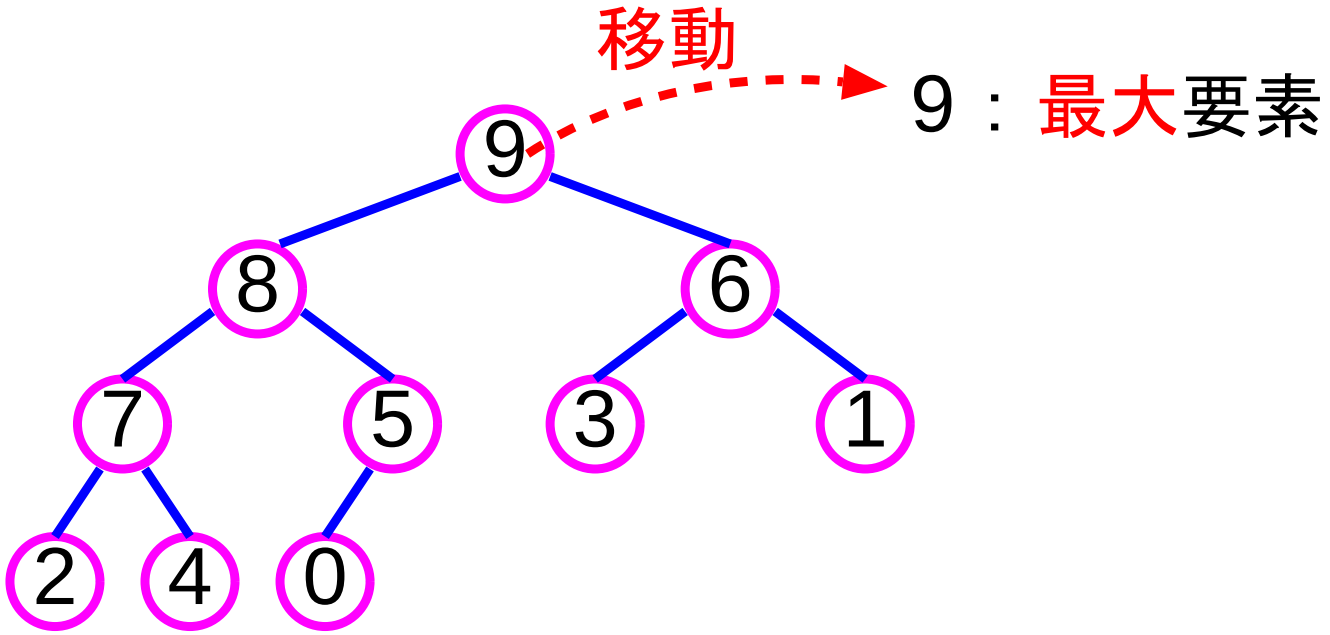
(考え方)　ヒープソート(整列2分木法)は、最悪の場合でもそれなりに効率良く整列化を行うアルゴリズムとして有名なものである。

アルゴリズムは次の通り。

① 整列すべきデータが節点のラベルとして振り分けられたヒープを構築する。

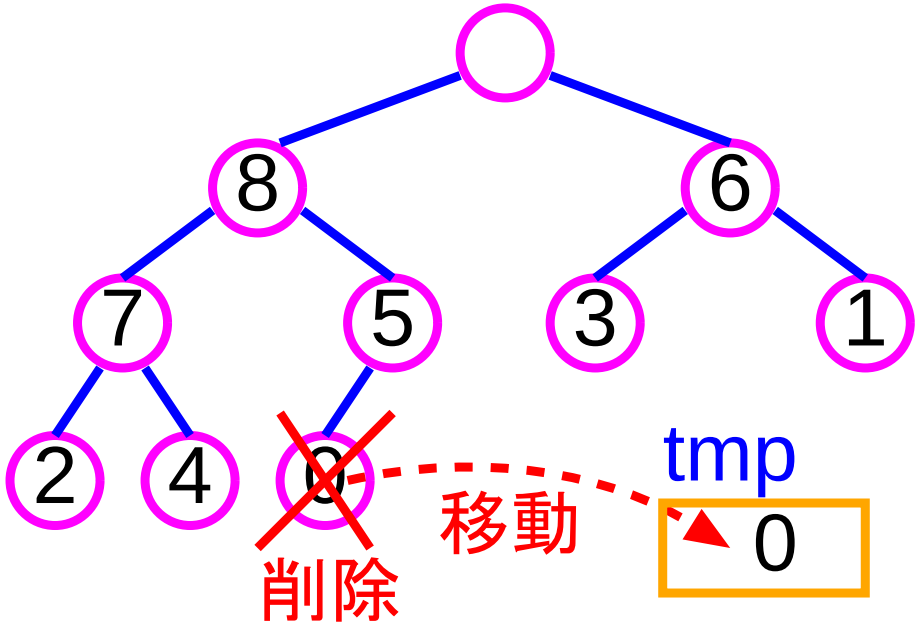


② 根にラベル付けされたデータは最大であることが分かるので、このデータを最大要素として出力用配列に移す。

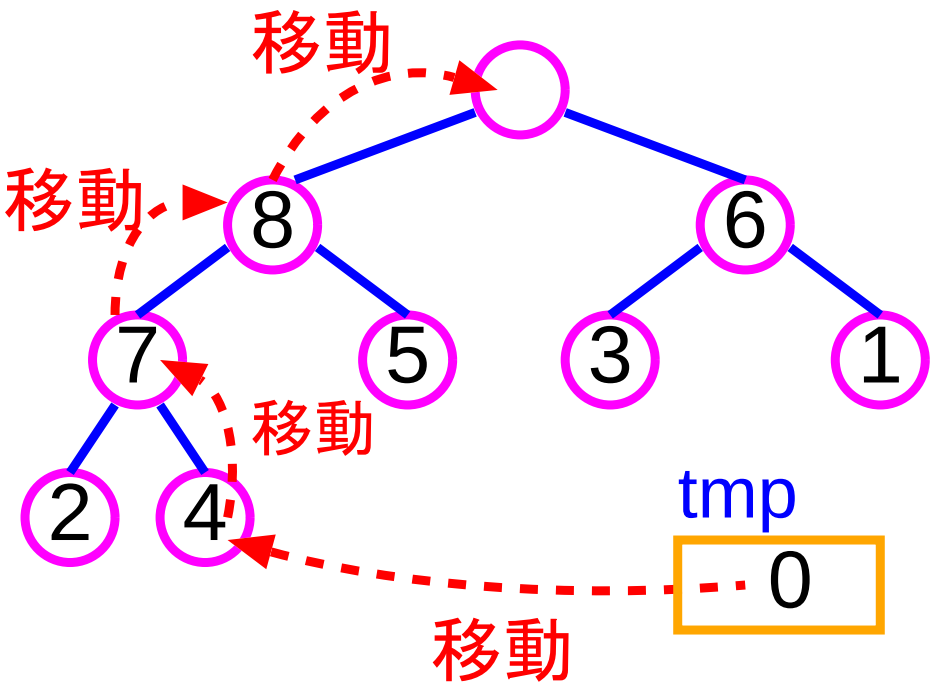


③ レベルが最大の葉節点の内、最も右側の節点をヒープから除去し、その中のデータを一時記憶領域 tmp に移す。

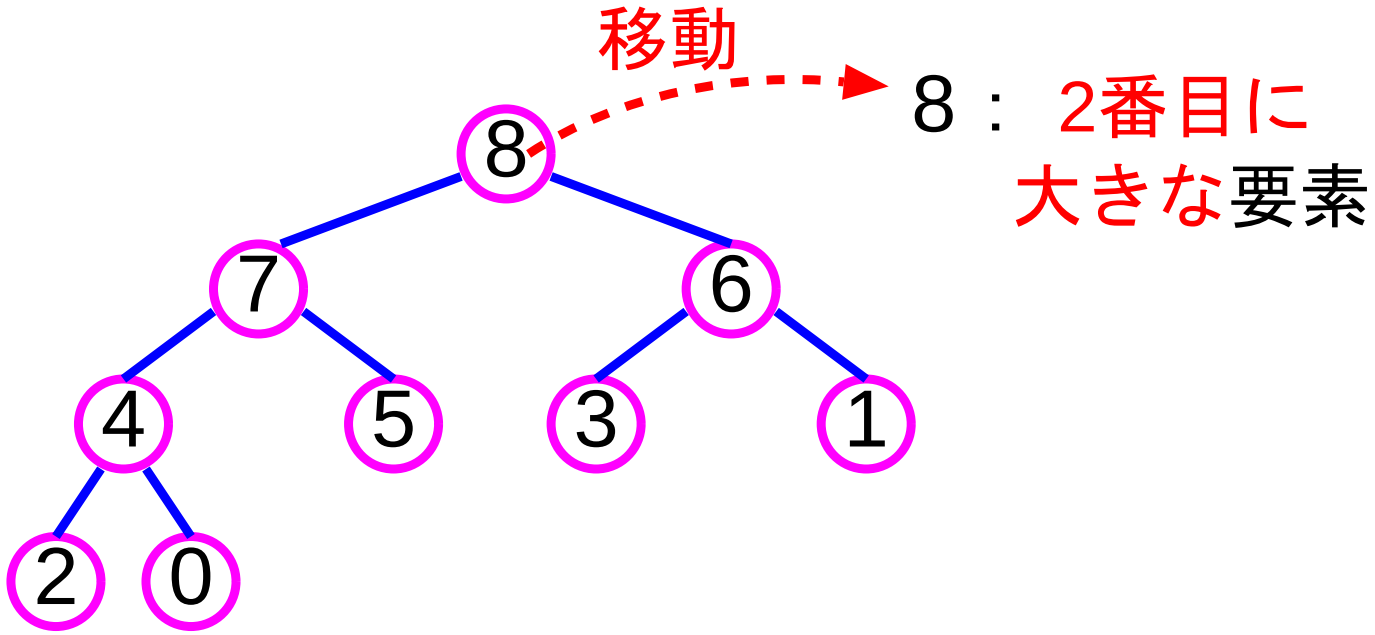
(tmpの中のデータを根節点に入れてもヒープにならない。)



④ ラベルの無い節点の中に大きい方の子節点データを移動する操作を繰り返し、適当なところでtmpの中のデータを「ラベルの無い節点」に移すことによって、再びヒープにする。



⑤ 根にラベル付けされたデータはヒープに残ったデータの中で最大であることが分かるので、このデータを出力用配列の然るべき場所に移す。



⑥ 節点数が1になるまで③~ ⑤を繰り返す。

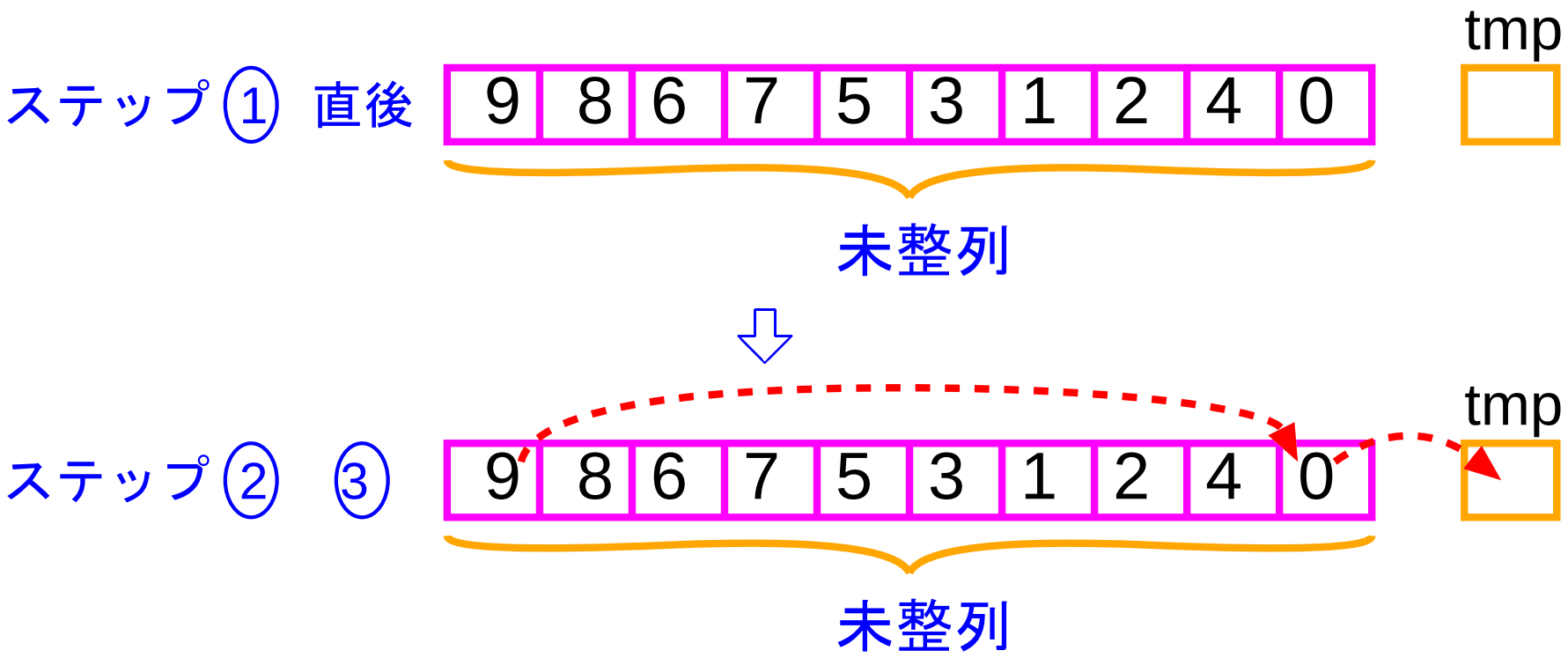
ポインタを用いて上記ヒープを表そうとすると 、
上記ステップ③でレベルが最大の節点のうちで最も右側の節点(およびこの節点の親節点)へのアクセスが必要になる。しかし、ポインタを用いてヒープを表す場合は、(可能ではあるが) 未整列データの個数からこれらの節点を割り出す簡単な方法はない。

⇨ 配列を用いて上記ヒープを表すことにする。

では、上記ヒープをプログラム内でどう表現すれば良いのか？

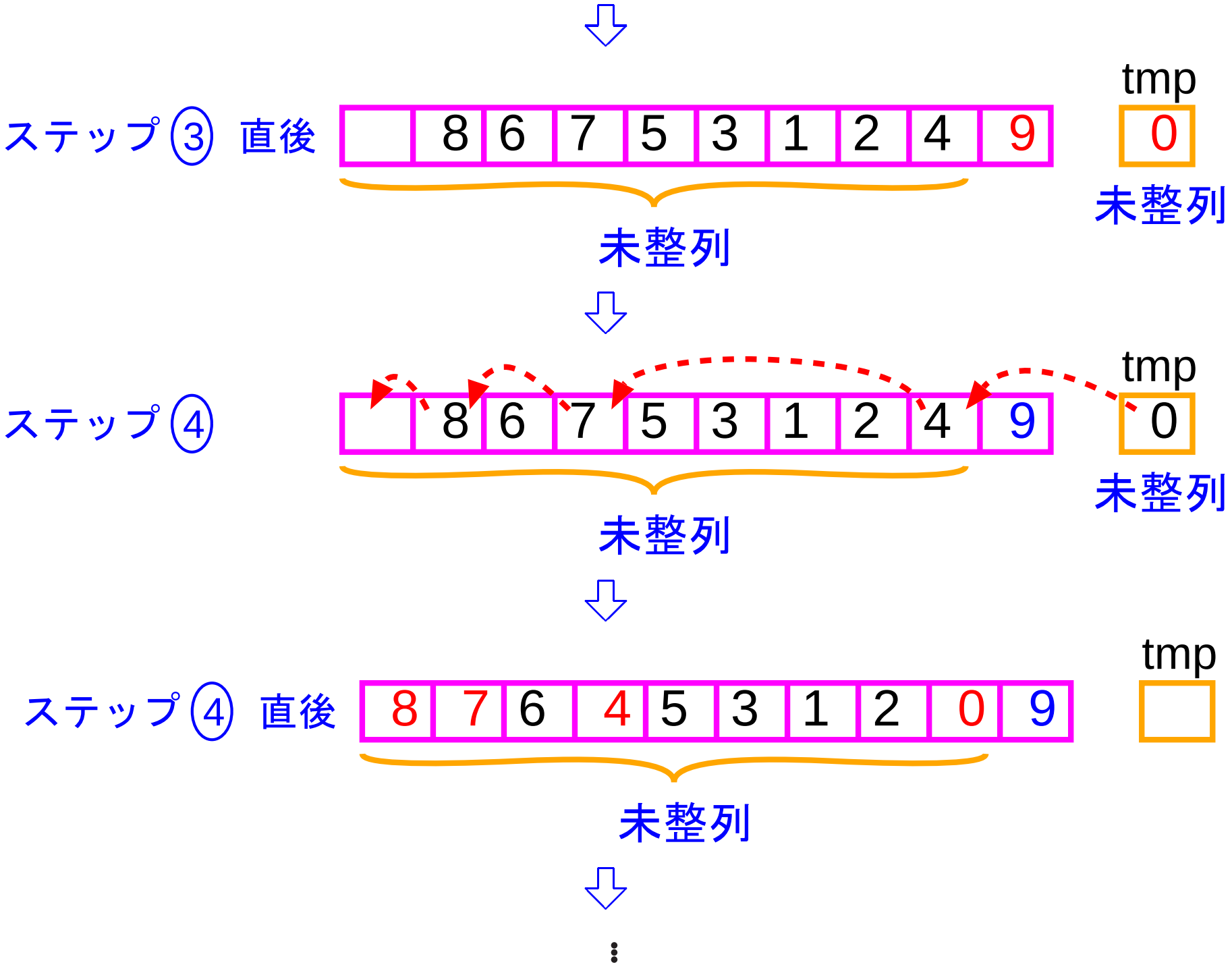
配列を用いて上記ヒープを表すことにする。

上記の例の場合、ヒープを表す配列の様子は次の様に変わって行く。



補足：

ステップ②で2分木の根に配置されたデータの整列後の配置場所は、残った未整列データの構成するヒープの中でレベルが最大の節点のうちで最も右側の節点になるので、ステップ②と③(および⑤と③)は入れ換えた方が処理が効率良く進む。


ステップ③ 直後
8 6 7 5 3 1 2 4 9
tmp
0
未整列
未整列
ステップ④
8 6 7 5 3 1 2 4 9
tmp
0
未整列
未整列
ステップ④ 直後
8 7 6 4 5 3 1 2 0 9
tmp
未整列

モジュールの構成はどうすれば良いのか？

int型配列に入ったデータを小さい順に並べ替える作業を外部からこのモジュールに依頼できる様にしなければならない。そのためには、指定された配列内のデータをヒープソート手法で小さい順に並べ替える関数

```
void sort(int a[], int size)
```

ソーティング手法を外部に説明する必要があるかも知れない。そのためには、例えば引数として指定されたchar型配列に "Heapsort" という文字列を入れる関数

```
void sort_method(char *method_name)
```

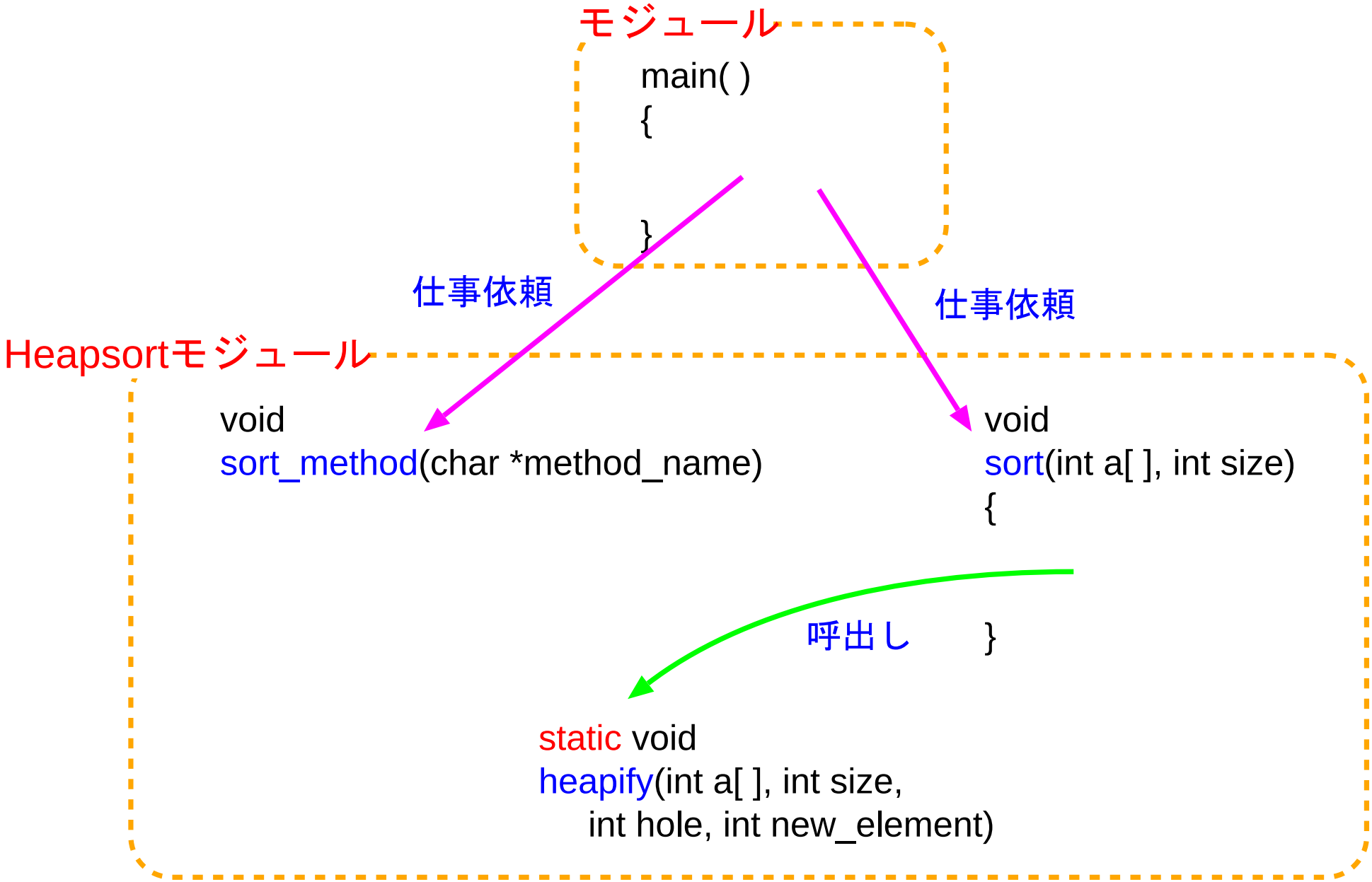
最後に、ヒープソートの手順の見通しを良くするために、上記手順のステップ④を行う関数

```
heapify( )
```

補足：

この関数は実際には2分木の再帰的な構造に沿って再帰的に構成することが出来る。

この関数 heapify( ) は外部から直接使うことはないので、もちろんstaticな外部関数にして外部からは見えなくすべきである。


モジュール
main( )
{
}
仕事依頼
仕事依頼
Heapsortモジュール
void
sort_method(char *method_name)
void
sort(int a[ ], int size)
{
呼出し
}
static void
heapify(int a[ ], int size,
int hole, int new_element)

(プログラミング)　出来たモジュールを次に示す。

```
[motoki@x205a]$  nl btree-heapsort.c
     1  /*****************************************************
     2  /* Heapsortモジュール : 2分木の利用例
     3  /*---------------------------------------------------
     4  /*     外部へのサービスを行うために、次の２つの関数がこの
     5  /*     モジュールの中に用意されている。
     6  /*     (1)整列化アルゴリズムの名前を答える関数 sort_metho
     7  /*     (2)配列要素をHeapsortアルゴリズムで
     8  /*                          で小さい順に並べ替える関数 sort
     9  /*****************************************************

    10  #include <stdio.h>

    11  static void heapify(int a[], int size,
                            int hole, int new_element);
```

```
12  /*-----------------------------------------------
13  /* 整列化アルゴリズムの名前を答える
14  /*-----------------------------------------
15  /* (引数) method_name : 出力用。このchar型配列に方式の..
16  /*                      を(文字列として)入れて返す。配..
17  /*                      大きさは 9 文字分必要。
18  /*-----------------------------------------------
19  void sort_method(char *method_name)
20  {
21    sprintf(method_name, "Heapsort");
22  }

23  /*-----------------------------------------------
24  /* 配列要素を小さい順に並べ替える (heapsort)
25  /*-----------------------------------------
26  /* (仮引数) a     : int型配列
```

```
27  /*            size : int型配列 a の大きさ
28  /* (関数値)  :  なし
29  /* (機能) :  heapsortアルゴリズムを使って、配列要素
30  /*                  a[0],a[1],a[2], ..., a[size-1]
31  /*              を値の小さい順に並べ替える。
32  /*-----------------------------------------------
33  void sort(int a[], int size)
34  {
35    int  k, tmp;

36    /* 下からheapを構築してゆく */
37    for (k=size/2-1; k>=0; --k)    /* 葉以外の個数=size/2
38      heapify(a, size, k, a[k]);

39    /* 大きい順にheapから取り出してゆく */
40    for (k=size-1; k>=1; --k) {
41      tmp  = a[k];
```

```
42        a[k] = a[0];
43        heapify(a, k, 0, tmp);
44      }
45    }

46    /*-----------------------------------------
47    /* 番号 hole の節点より下の部分がheapの条件を満たす時...
48    /* 新要素を加えてhole以下の部分がheapの条件を満たす様...
49    /*-----------------------------------------
50    /* (仮引数) a     : int型配列
51    /*      tree_size : 2分木と見做す部分配列 ... の大きさ  *
52    /*           hole : a[0]~ a[tree_size]の表す2分木の節..
53    /*   new_element : 2分木の節点に振り分けてい... */
54    /* (関数値)  :  なし
55    /* (機能) :  番号 hole の節点のデータ記憶域は空で、その..
56    /*           new_element という値がどの節点にも記録され.
57    /*           ない、また、hole より下の部分がheapの条件...
```

```
58  /*              たしている、という状況を想定する。この様な...
59  /*              の時に、hole より下にあるデータを上に shift
60  /*              する操作を繰り返し行い、適当な時点で空の節..
61 /*              新しい要素 new_element を割り当てることに...
62 /*              hole 以下の部分が全面的に heapの条件を満た..
63  /*-----------------------------------------------------
64  static void heapify(int a[], int tree_size, int hole,
                                              int new_element)
65  {
66    int  siftup_cand;      /* siftup candidate */

67    while ((siftup_cand = hole*2+1) < tree_size) {
68      if (siftup_cand+1<tree_size        /*右の子も居て*/
69          && a[siftup_cand]<a[siftup_cand+1]) /*右の子*/
70        ++siftup_cand;                  /*の方が大きい場合は*/
71                                      /*右の子がsiftupの候補*/
72      if ( new_element >= a[siftup_cand])
```

```
73          break;  /* new_elementをholeの場所に入れれば良い

74        a[hole] = a[siftup_cand];      /* sift up */
75        hole    = siftup_cand;
76      }
77      a[hole] = new_element;
78  }
```

(実行)　このheapsortモジュールの動作確認は例題4.18(quicksort)と同じように行うことが出来る。次の通り。

```
[motoki@x205a]$  nl check-sort-program.c
     1  /*************************************************
     2  /* Sortプログラムの動作確認
     3  /*-------------------------------------
     4  /*     大きさ100の配列にランダムに整数を生成し、
```

```
 5  /*      その配列要素を別途用意された整列化プログラムを使っ..
 6  /*      昇順に並べ替えて出力する。
 7  /*************************************************************

 8  #include <stdio.h>
 9  #include <stdlib.h>    /* 乱数発生のライブラリ関数を使う.

10  #define  SIZE   100
11  #define  WIDTH  10

12  void set_an_array_random(int a[], int size);
13  void pretty_print(int a[], int size);
14  void sort(int a[], int size);
15  void sort_method(char *method_name);

16  int main(void)
17  {
```

```
18      int   a[SIZE], seed;
19      char  sort_name[40];

20      printf("Input a random seed (0 - %d):  ", RAND_MAX);
21      scanf("%d", &seed);
22      srand(seed);

23      set_an_array_random(a, SIZE);
24      printf("\nbefore sorting:\n");
25      pretty_print(a, SIZE);

26      sort(a, SIZE);
27      sort_method(sort_name);         /* Sortモジュールに
28                                      /* 何のアルゴリズムか..
29      printf("\nafter sorting (%s):\n", sort_name);
30      pretty_print(a, SIZE);
31      return 0;
```

```
32  }

33  /*-----------------------------------------------
34  /* 引数で与えられた配列の各要素をランダムに設定 */
35  /*-----------------------------------------------
36  /* (仮引数) a    : int型配列
37  /*           size : int型配列 a の大きさ
38  /* (関数値)  :  なし
39  /* (機能) :  配列要素 a[0]~ a[size-1] に 0~ 999 の間の.
40  /*            を設定する。
41  /*-----------------------------------------------
42  void set_an_array_random(int a[], int size)
43  {
44    int i;

45    for (i=0; i<size; ++i)
46      a[i] = rand() % 1000;
```

```
47  }

48  /*-------------------------------------------------
49  /* 引数で与えられた配列の要素を順番に全て出力... */
50  /*-------------------------------------------------
51  /* (仮引数) a     : int型配列
52  /*            size : int型配列 a の大きさ
53  /* (関数値)  :  なし
54  /* (機能) :  配列要素 a[0]~ a[size-1] の値を順番に全て..
55  /*            する。但し、各々の値は横幅7カラムのフィー...
56  /*            に出力することにし、また、1行にWIDTH個の...
57  /*            を出力する。
58  /*-------------------------------------------------
59  void pretty_print(int a[], int size)
60  {
61    int i, count=1;
```

```
    62    for (i=0; i<size; ++i, ++count) {
    63      printf("%7d", a[i]);
    64      if (count >= WIDTH) {
    65        printf("\n");
    66        count = 0;
    67      }
    68    }
    69    if (count > 1)
    70      printf("\n");
    71  }
[motoki@x205a]$  gcc check-sort-program.c btree-heapsort.c
[motoki@x205a]$  ./a.out
Input a random seed (0 - 2147483647):   333

before sorting:
    556     289     435     368     666     319     214     273      13
     64     943     869     956      50     298     112     218
```

```
   603   936   515   385   671   776   137   886
   718   913   204   153   281   870   473   495   14
   432   208   548   653   517   950   951   629   52
   630   476   893   498   861   917   626   998   80
   913   521   544   470    27   825   340   500   67
   105   104   397     6   110   914   308    61   89
    18   878   305   264   376   518   181   354   51
   985   782   857   881   252   236   706   945   73

after sorting (Heapsort):
     4     5     6    18    27    50    61    64   10
   110   112   132   137   144   153   181   204   20
   218   236   252   264   273   281   289   298   30
   319   336   340   354   368   376   385   397   43
   470   473   476   495   498   500   515   517   51
   520   521   544   548   556   563   585   603   60
   629   630   631   649   653   666   671   672   70
```

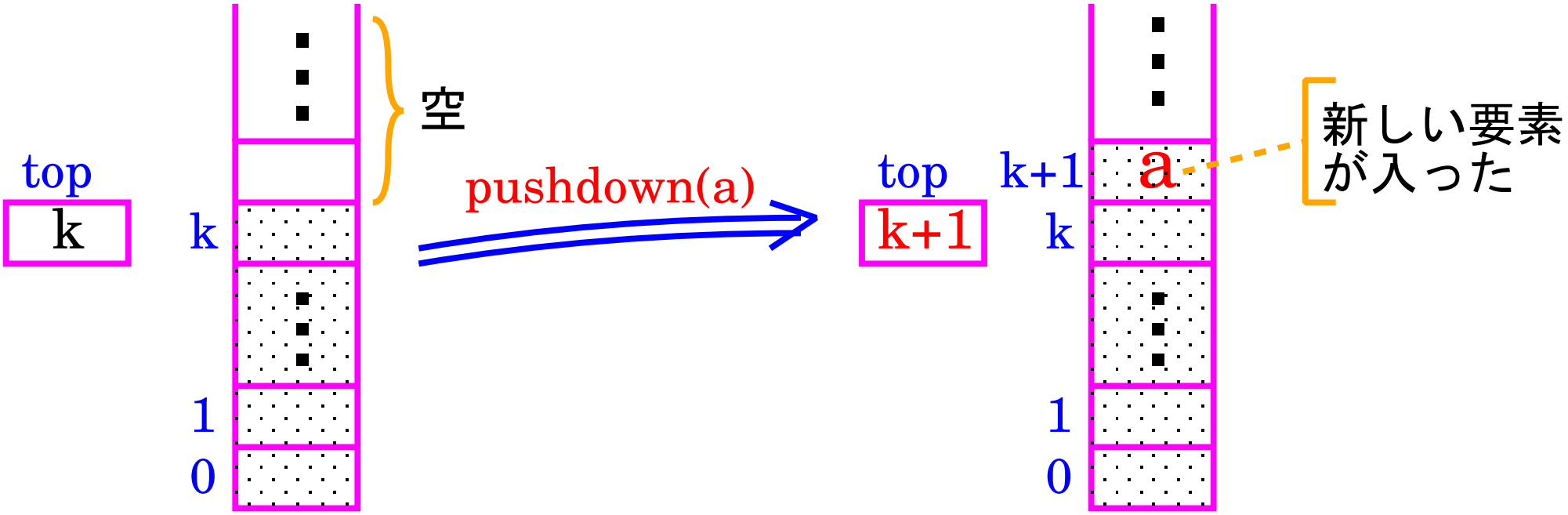
```
    730     736     776     782     803     825     829     836     85
    869     870     878     881     886     893     895     913     91
    917     936     943     945     950     951     956     957     98
[motoki@x205a]$
```

13–2 push-downスタック

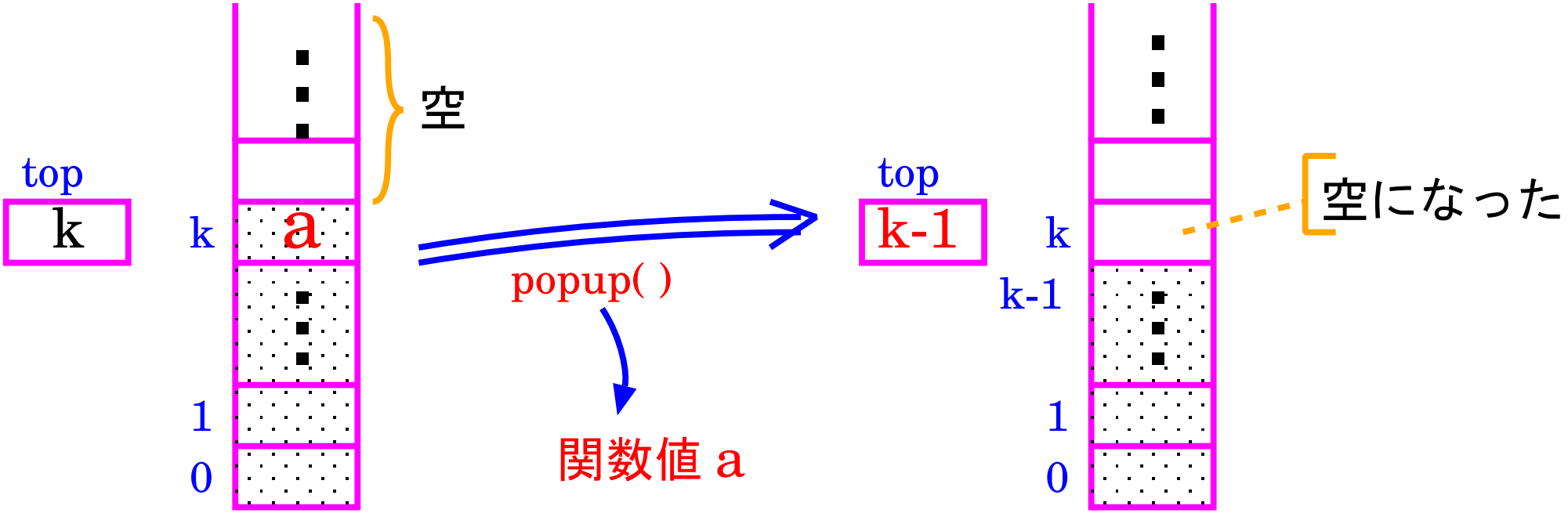
push-downスタック：

… pushdownとpopupの操作を備え、データの出し入れが
後入れ先出し(last-in-first-out, LIFO)になるデータ記憶領域。

(pushdown操作)

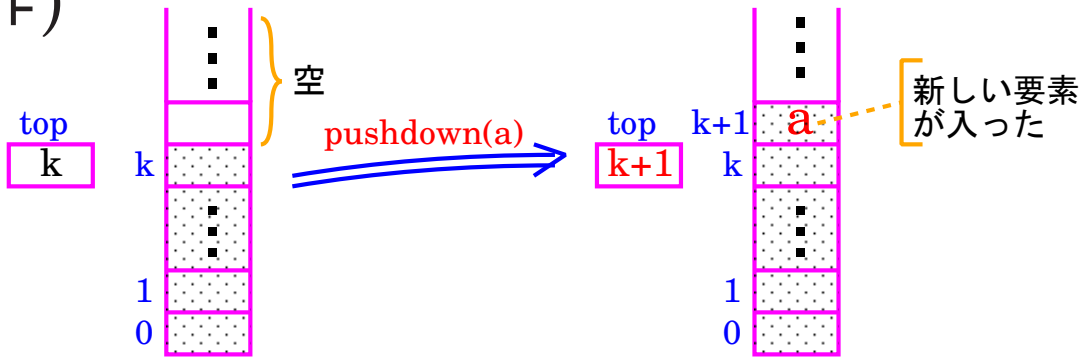


(popup操作)

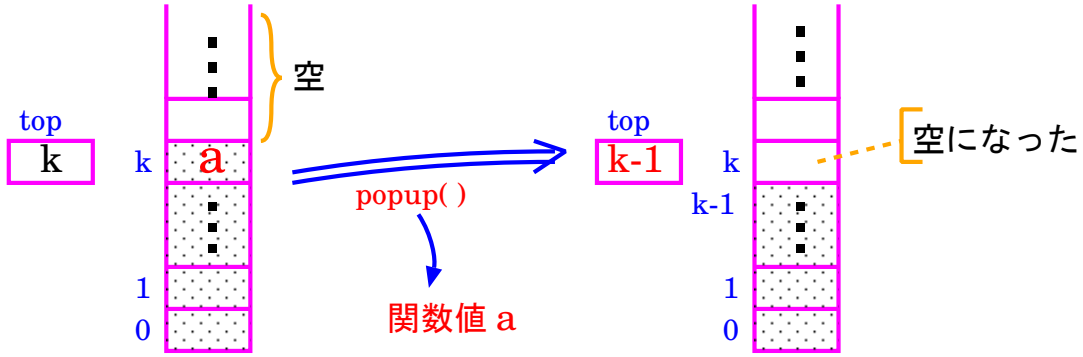


- スタックに入れる要素の個数がいくらでも大きくなり得る場合は、線形リストを用いてスタックを実装するしかない。
- スタックの場合は蓄えられたデータの途中に要素を 挿入することも 途中から要素を 抜き出すこともない ので、スタックに入れる要素の個数の上限が分かっているなら、配列を用いてスタックを実装するのが良い。

(pushdown操作)



(popup操作)



例題13. 3 (スタックを使って文字列を反転する)　int 型データが最大100個入るスタックを表す汎用のモジュールを作成せよ。 そして、このスタックモジュールを利用して、①文字列を1個読み込み ②それを反転した後 ③出力する プログラムを作成せよ。

補足：

char 型データを int のスタックに入れることになるが、char 型は整数型の一種であるので問題は起きない。char型データ用のスタックを用いてもよいが、int 型用スタックにしておくと別の用途にも使えるので、ここではint型データ用のスタックモジュールを作成することにした。

(考え方)　スタック領域をどこに確保するか

スタック領域をスタックモジュールを呼び出す側で確保してpushdownやpopupの操作を行う関数に引数としてスタック領域のアドレスを引き渡すというのでは、元々スタックモジュールを呼び出す側で任意の時点でスタック領域に直接手を加えられることになり、「後入れ先出し」の原則が保証されなくなる。

従って、スタック領域はスタックモジュールの中に確保し、これらの領域へは外部から直接アクセスできなくするべきである。pushdownやpopupの操作を行う関数は、これらの領域を外部から間接的に操作するためのインターフェースとして働くことになる。

結局、スタックモジュールの構成要素としては、少なくとも

- スタック領域 `static int stack[100]`,
- スタックのtop要素を示す変数 `static int top`,
- pushdown操作を行う関数 `void pushdown(int)`,
- popup操作を行う関数 `int popup()`

の4つは必要である。

その他にも、スタックを正しく使うために

- スタックの初期化を行う関数 `initialize_stack(void)`,
- スタックが空かどうかを判定する関数 `int is_empty(void)`

という関数を用意すれば良い。

一方、 スタックモジュールを利用する側に関しては 、

①文字列を１個読み込み

②それを反転した後

③出力する

という作業の中で、スタックが有用となるのは文字列反転の部分である。

文字列を反転するには

文字列の先頭から１文字ずつスタックにpushdownしていき、

それが終わればpopupを繰り返す

だけである。popupは元々の文字列の最後尾から逆順に行われることになるので、これを入力文字列の入っていた配列に先頭から順に格納していけば良い。

(プログラミング)

```
[motoki@x205a]$  nl stack-int100.c
     1  /*************************************************************
     2  /* int型データが最大100個入るスタックを実装したモジュー...
     3  /*-------------------------------------
     4  /*     外部へのサービスを行うために、次の４つの関数がこの
     5  /*     モジュールの中に用意されている。
     6  /*     (1)スタックを空に初期化する関数 initialize_stack,
     7  /*     (2)スタックが空かどうかを調べる関数 is_empty,
     8  /*     (3)スタックに要素を1つ push-down する関数 pushdown
     9  /*     (4)スタックから要素を1つ pop-up する関数 popup
    10  /*************************************************************

    11  #include <stdlib.h>
    12  #define  TRUE   1
    13  #define  FALSE  0
```

```
14  typedef  int  Boolean;

15  static int  stack[100];  /* モジュール外からは見えない *
16  static int  top;         /* モジュール外からは見えない *

17  void initialize_stack(void)
18  {
19    top = -1;
20  }

21  Boolean is_empty(void)
22  {
23    if (top < 0)
24      return TRUE;
25    else
26      return FALSE;
27  }
```

```
28  void pushdown(int k)
29  {
30    if (++top >= 100) {
31      printf("stack overflow\n");
32      exit(EXIT_FAILURE);
33    }
34    stack[top] = k;
35  }

36  int popup(void)
37  {
38    if (top < 0) {
39      printf("popup from empty stack\n");
40      exit(EXIT_FAILURE);
41    }
42    return stack[top--];
```

```
    43  }
[motoki@x205a]$  nl stack-reverse-word.c
     1  /*************************************************
     2  /* スタックを利用する例
     3  /*--------------------------------
     4  /*     文字列を1個読み込み、
     5  /*     それをスタックモジュールを用いて反転した後出力する
     6  /*************************************************

     7  #include <stdio.h>

     8  typedef  int  Boolean;

     9  void initialize_stack(void);
    10  Boolean is_empty(void);
    11  void pushdown(char c);
    12  char popup(void);
```

```
13  int main(void)
14  {
15    char s[100];
16    int  i;

17    printf("Input a string:  ");
18    scanf("%s", s);
19    initialize_stack();
20    for (i=0; s[i]!='\0'; ++i)
21      pushdown(s[i]);
22    for (i=0; !is_empty(); ++i)
23      s[i] = popup();
24    printf("Reversed string: %s\n", s);
25    return 0;
26  }
```

```
[motoki@x205a]$  gcc stack-reverse-word.c stack-int100.c
```

```
[motoki@x205a]$  ./a.out
Input a string:   abc123
Reversed string: 321cba
[motoki@x205a]$
```

逆ポーランド記法：

算術式は普通、(2項) 演算子を被演算数の間に置く中置記法を用いて書き表すが、他に、前置記法 (またはポーランド記法)、後置記法 (または逆ポーランド記法) という書き方も可能である。

例えば、

```
(17 * 5) + 2
```

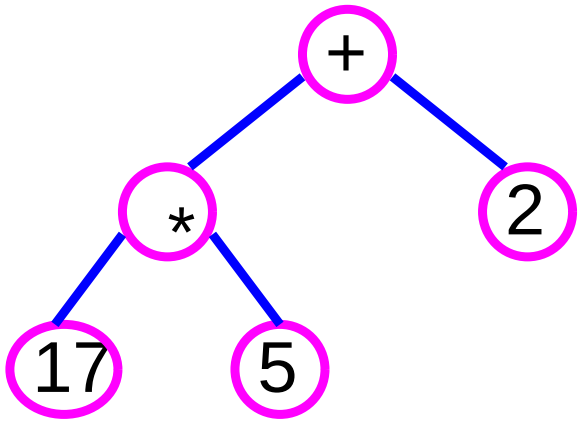
という算術式は、前置記法では

```
+ * 17 5 2
```

後置記法では

```
17 5 * 2 +
```

と書ける。



前置記法と後置記法では計算の順序を明示するのに括弧は不要であるが、被演算数同士を区切るもの (例えば空白) が必要になる。

逆ポーランド記法で書かれた算術式の評価：

⇨ スタックを用いると容易。
すなわち、算術式を左から読みながら、次のことを行えば良い。

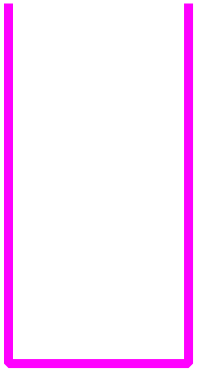
- 被演算数が読まれると、それをスタックに pushdown する。
- 符号反転の単項演算子 ~ が読まれると、スタックから要素を1つ popup し、その符号を反転したものをスタックにpushdown する。
- 2項演算子 +, −, ∗ または / が読まれると、スタックから要素を2つ popup し、
(後でpopupされた値) 読まれた演算 (先にpopupされた値)
という計算の結果をスタックに pushdown する。
- 読み込むものが無くなっていれば、スタック内に算術式の評価値が1つだけ残っているはずである。

この方法では、例えば

```
17  5  *  ~  2  +
```

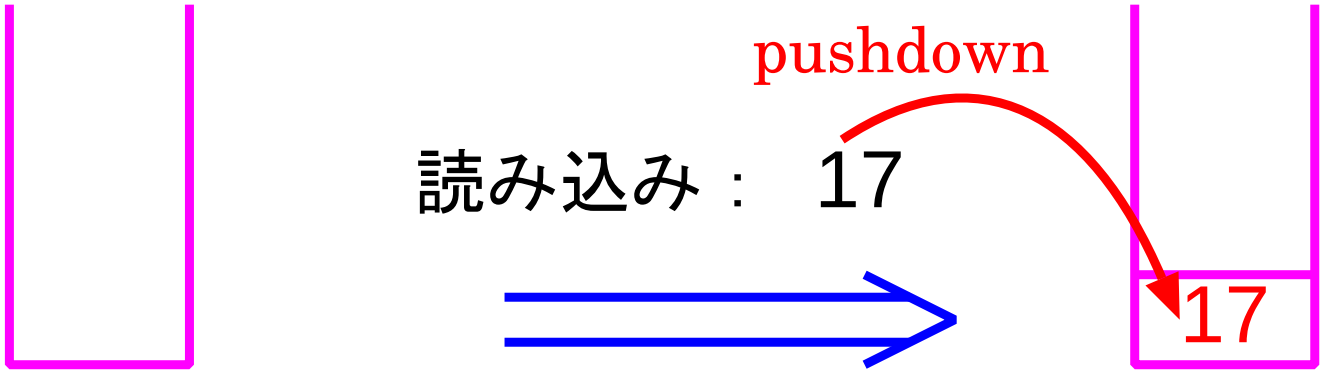
という算術式に対しては、次のように計算が進むことになる。

⓪ (初期状態)

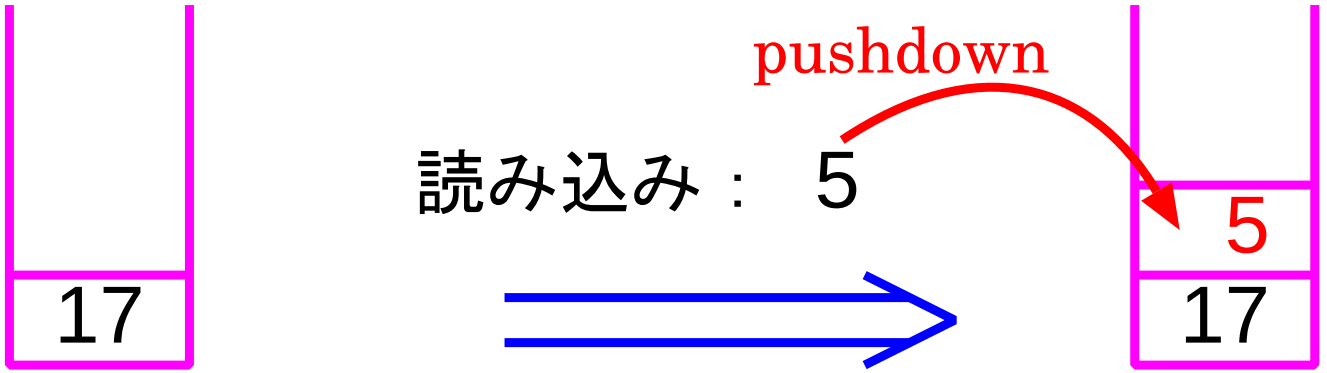


スタックは空

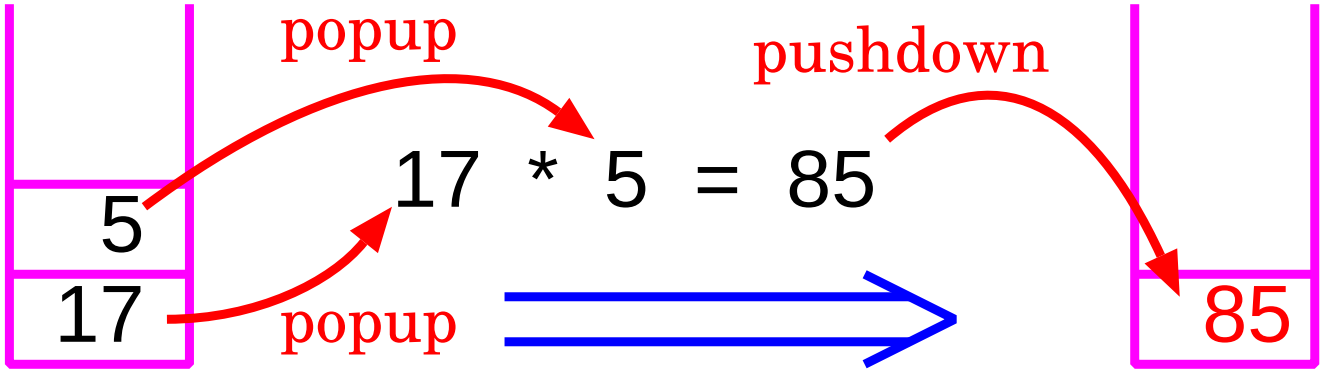
① 被演算数 17 が読まれる。



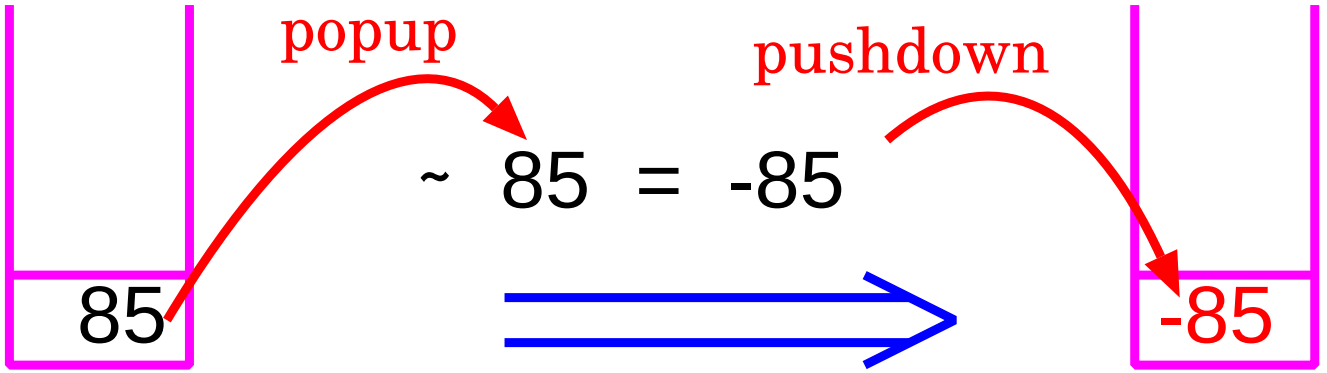
② 被演算数 5 が読まれる。



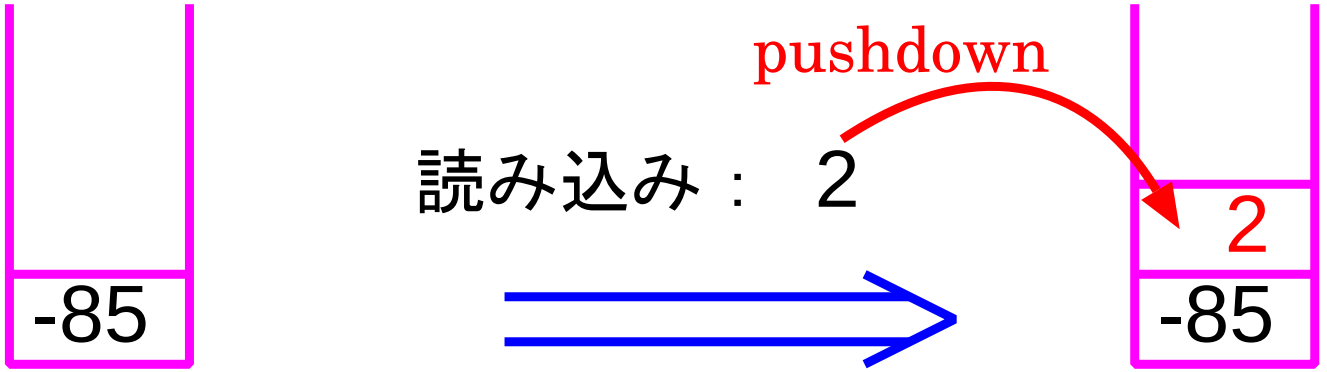
③ 演算子 * が読まれる。



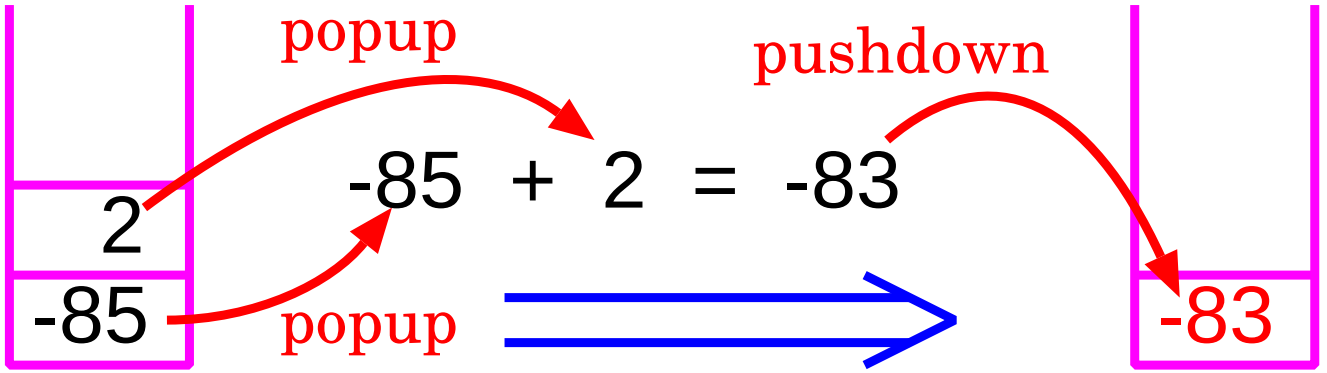
④ 演算子 ~ が読まれる。



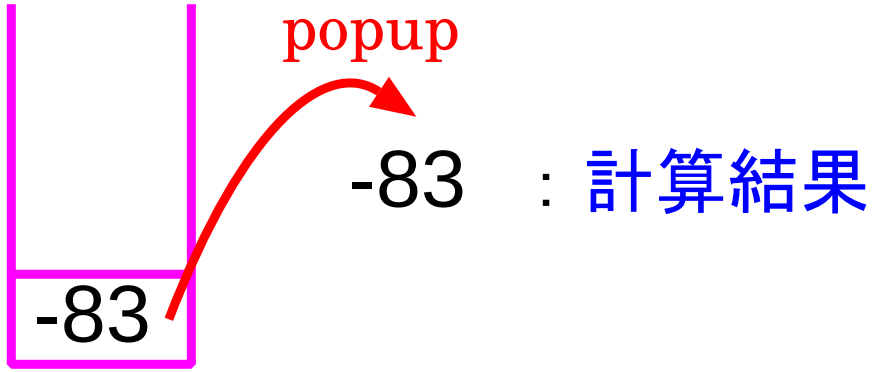
⑤ 被演算数 2 が読まれる。



⑥ 演算子 + が読まれる。



⑦ 読むものが無くなった。



例題13. 4 (逆ポーランド記法で書かれた算術式をスタックを用いて評価
10進非負整数と四則演算(2項の加減乗除 +, −, ∗, / と符号反転 ~)だけから成る、逆ポーランド記法で書かれた算術式を文字列として読み込んで、その値を求めるプログラムを作成せよ。

(考え方)　例題13.3で作成したスタックモジュールを用いて、p.286 ~ 288で説明された手順をそのままプログラムで表すだけである。

(プログラミング)　ここでは簡単のため、

- 簡単のため、読み込む算術式では演算子や整数は空白で区切られているものと仮定し、数式の終わりは Ctrl-d で表す。
- 算術式の誤りのチェックはしない。

また、

- コンパイルに用いたスタックのモジュール
  stack-int100.c は例題13.3で使ったものと同じ。

```
[motoki@x205a]$  nl stack-eval-expression.c
     1  /*************************************************
     2  /* スタックを利用する例
     3  /*-----------------------------------
     4  /*     10進非負整数と四則演算(2項の加減乗除 +, -, *, / と
     5  /*     号反転 ~)だけから成る、逆ポーランド記法で書かれた...
     6  /*     式を文字列として読み込んで、その値を求める
     7  /*************************************************

     8  #include <stdio.h>

     9  typedef  int  Boolean;

    10  void    initialize_stack(void);
    11  Boolean is_empty(void);
    12  void    pushdown(int k);
    13  int     popup(void);
```

```
14  int main(void)
15  {
16    int  a, b, num, i;
17    char buf[13];

18    initialize_stack();
19    while (scanf("%12s", buf) > 0) {
20      switch (buf[0]) {
21      case '~':
22        a = popup();      /* buf[1]=='\0'のチェックは省略 >
23        pushdown(-a);
24        break;
25      case '+':
26        b = popup();      /* buf[1]=='\0'のチェックは省略 >
27        a = popup();
28        pushdown(a+b);
```

```
29          break;
30        case '-':
31          b = popup();        /* buf[1]=='\0'のチェックは省略 *
32          a = popup();
33          pushdown(a-b);
34          break;
35        case '*':
36          b = popup();        /* buf[1]=='\0'のチェックは省略 *
37          a = popup();
38          pushdown(a*b);
39          break;
40        case '/':
41          b = popup();        /* buf[1]=='\0'のチェックは省略 *
42          a = popup();
43          pushdown(a/b);
44          break;
45        default:              /* 数字の列のはず */
```

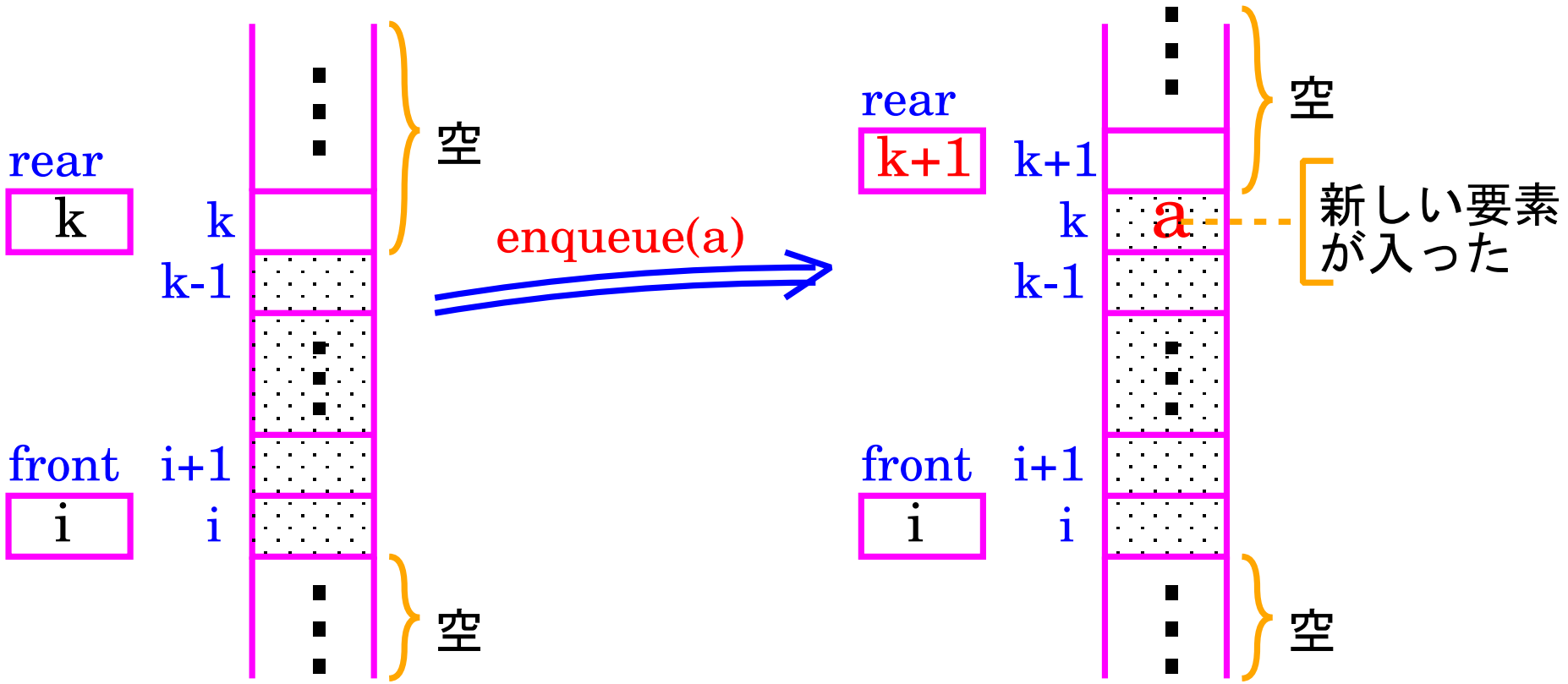
```
46          num = 0;
47          for (i=0; buf[i]!='\0'; ++i)
48            num = num*10 + (buf[i]-'0');   /* buf[i]が数字
49          pushdown(num);                    /* であることの *
50          break;                          /* チェックは省略 *
51        }
52      }
53      printf("\nresult of evaluation: %d\n", popup());
54          /* スタックが空になっていることのチェックは省略 */
55      return 0;
56    }
[motoki@x205a]$  gcc stack-eval-expression.c stack-int100.c
[motoki@x205a]$  ./a.out
17 5 * ~ 2 +
 Ctrl-d
result of evaluation: -83
[motoki@x205a]$
```

13–3 自習 待ち行列

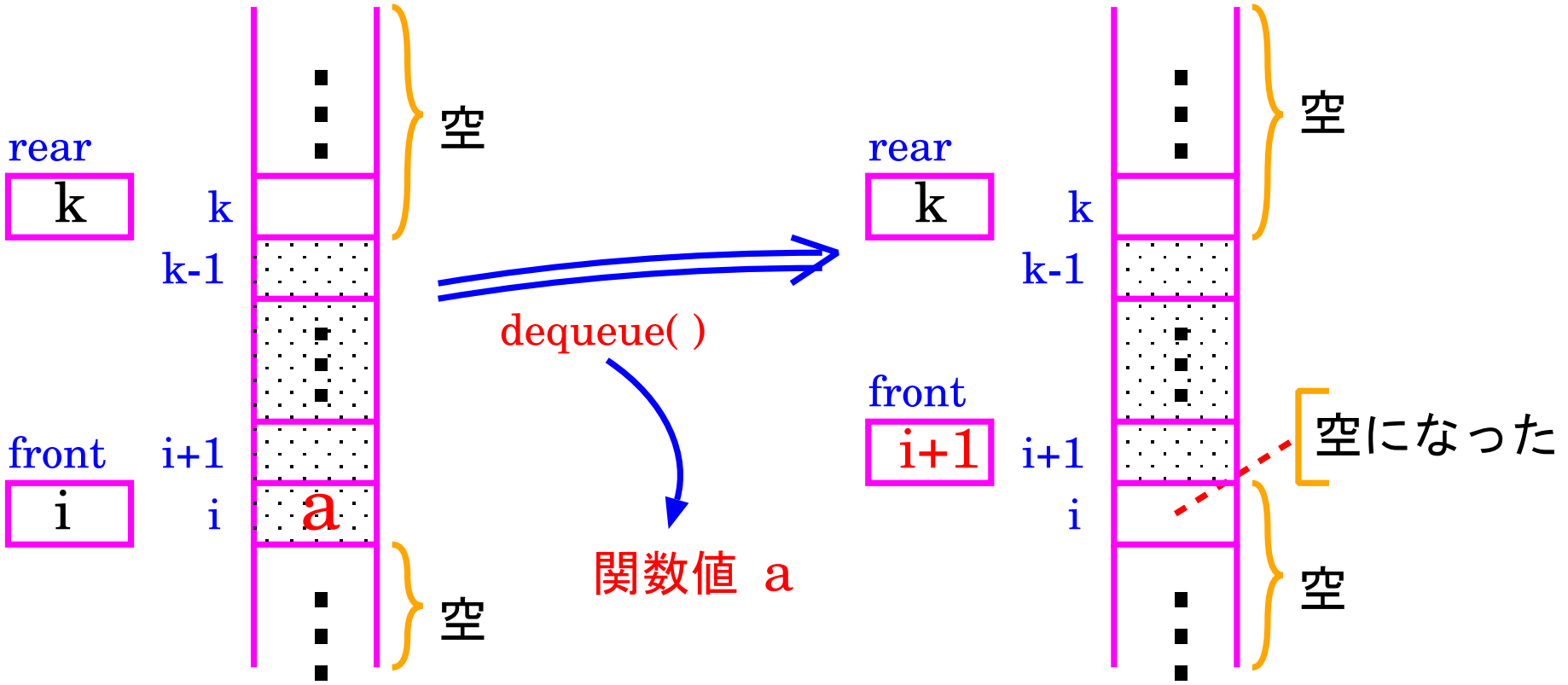
待ち行列：

… 「enqueue」と「dequeue」の操作を備え、データの出し入れが先入れ先出し(first-in-first-out,FIFO)に従うデータ記憶領域。

(enqueue操作)

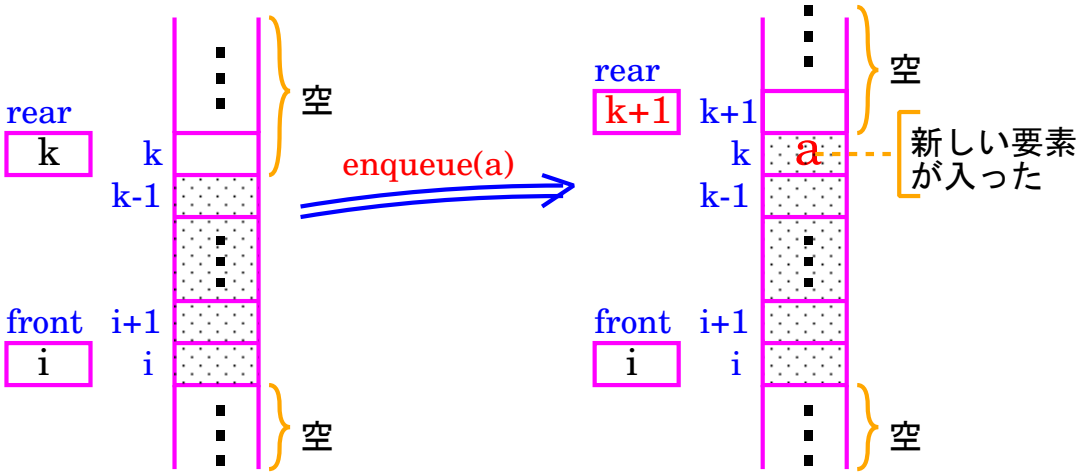


(dequeue操作)

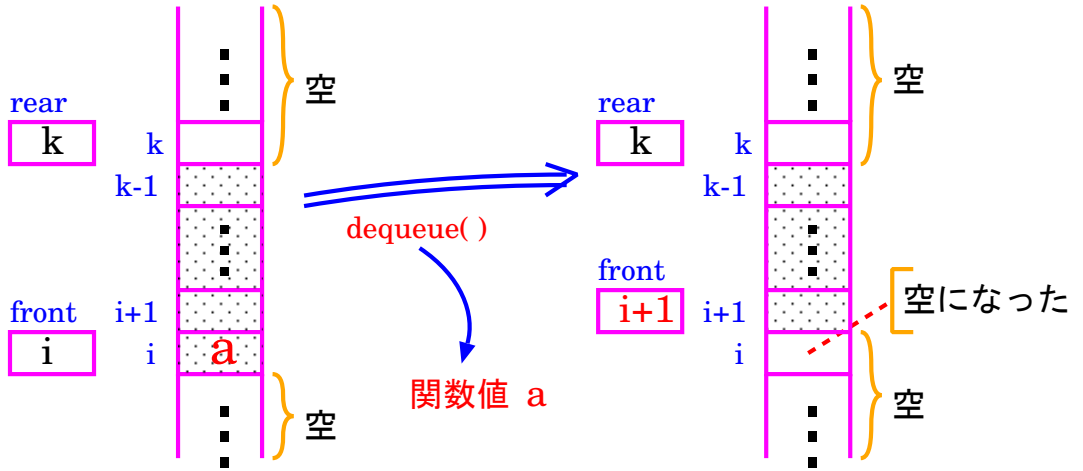

rear
k
k
k-1
front
i
i+1
i
a
空
空
dequeue( )
関数値 a
rear
k
front
i+1
k
k-1
i+1
i
空
空になった
空

- 待ち行列に入る要素の個数に上限がないなら、線形リストを用いて待ち行列を実装するしかない。[スタックの場合と同様。]
- 待ち行列に入る要素の個数の上限が分かっているなら、配列を用いて待ち行列を実装するのが良い。[スタックの場合と同様。]

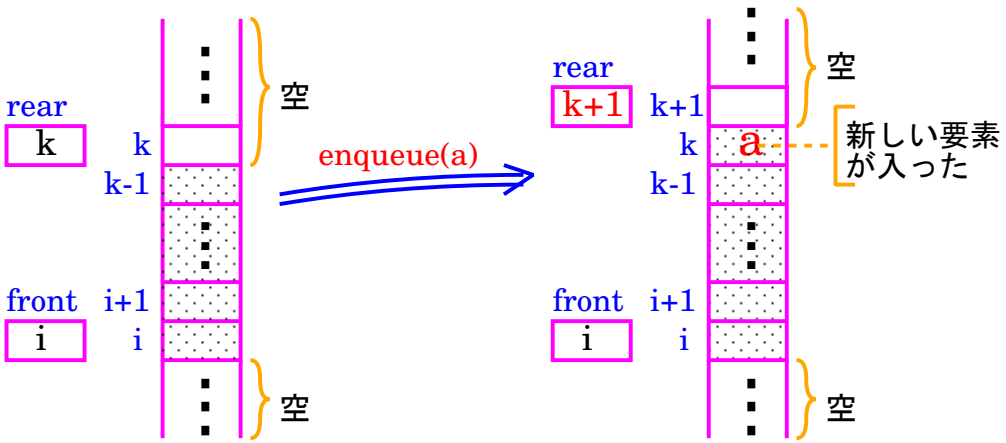
(enqueue操作)



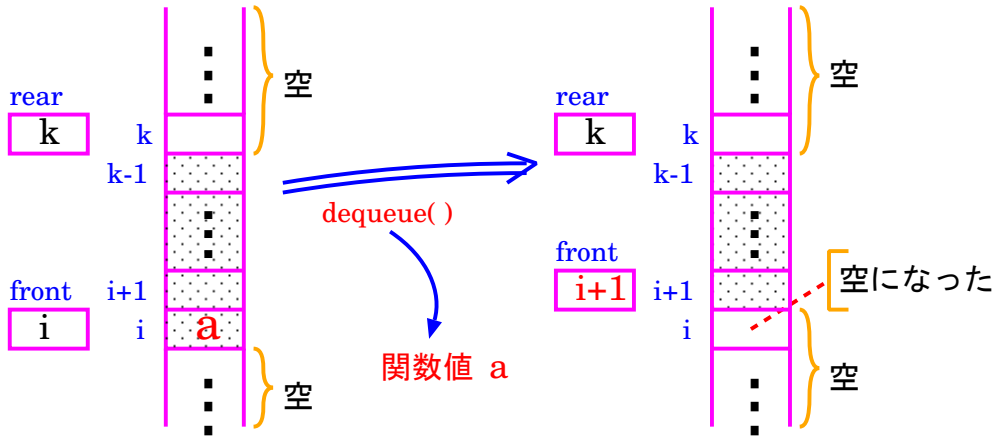
(dequeue操作)



(enqueue操作)



(dequeue操作)



但し、この場合、何も工夫しないと時間と伴に rear の値も front の値も大きくなる一方で、確保した配列の領域を超えてしまう。

$\Longrightarrow$ 大きさ N の配列 queue で待ち行列を表す場合は、
queue[N−1] の次に queue[0] が繋がっていると見なす。

授業では詳細は省略します。
必要なら、ケリー&ポール10.7節等を読んで下さい。