データ構造入門及び演習
14回目:まとめ

2014/07/18

担当:見越　大樹
61号館304号室

再帰呼び出し

- 再帰呼び出しとは？
  - 関数が自分自身（同じ形の関数）を呼び出すこと
  - 再帰：リカーシブ(recursive)コール，元に戻る，繰り返し
- 効果：
  - 繰り返し処理の代用
  - プログラムサイズ（ソースコード）を格段に小さくできることもある

階乗・和を実現する再帰関数例

```
int  factorial(int n)
{
    if( n==0 )
        return 1;
    else
        return n*factorial(n-1);
}
```

- 条件
  - n = 0 の場合 n! = 1
  - 上記以外 n! = n * (n-1)!

```
int  sum( int n ) {
    if (n==1) return 1;
    else
        return n+sum(n-1);
}
```

- 条件
  - n = 1 の場合 ∑n = 1
  - 上記以外 ∑n = n + ∑(n-1)

フィボナッチ数

フィボナッチ数　F(n)　は以下の条件を満たす
　（ア）n = 0の場合 : F(n)　= 0
　（イ）n = 1の場合 :　F(n)　= 1
　（ウ）上記以外の場合 : F(n) = F(n - 1) + F(n - 2)

```
int fibonacci(int n){
    if(n==0)
        return 0;
    else if( n == 1)
        return 1;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}
```

ポインタ

- ポインタとは？
  - 変数が格納されている位置(アドレス)を値とする変数

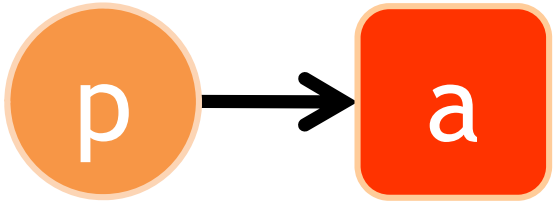
- ポインタの宣言：
  - 型名の後ろ　もしくは 変数名の前にアスタリスクをつける

    char* p;　　または　　char *p;

    (pをポインタ変数と呼ぶ)

- ポインタへのアドレスの代入：
  - ポインタpに変数aのアドレスを代入

```
char* p;
char  a='T';
p = &a;
```



pはaを指す
(pはaのアドレスを持つ)

構造体の宣言

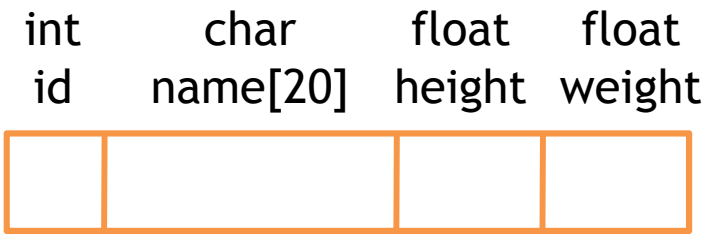
1. 構造体テンプレートの宣言
   - どのような型の変数や配列を1つにまとめるかを決定する

```
struct student {
    int id;
    char name[20];
    float height;
    float weight;
};
```

構造体テンプレート名(タグ)

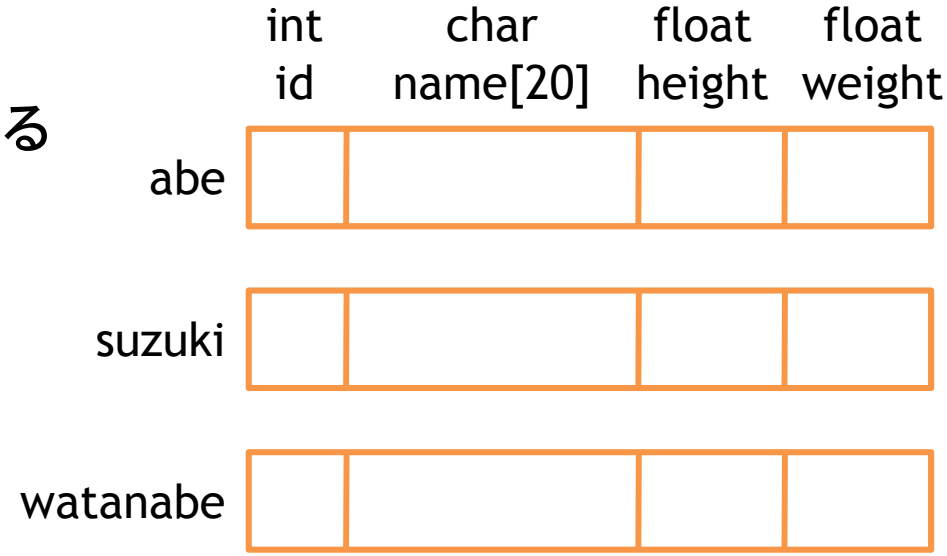
メンバ(構成要素)

← セミコロン



2. 構造体変数の宣言
   - テンプレートを持った変数を決定する

```
struct student abe;
struct student suzuki;
struct student watanabe;
```



構造体の初期化

```
// 宣言
struct student {
    int id;
    char name[20];
    float height;
    float weight;
};
```

	int id	char name[20]	float height	float weight
abe	101	阿部一朗	178.5	63.5
suzuki	114	鈴木健二	167.5	53.0
watanabe	199	渡辺隆史	175.0	82.4

```
// 初期化
struct student abe = { 101, “阿部一朗”, 178.5, 63.5 };
struct student suzuki = { 114, “鈴木健二”, 167.5, 53.0 };
struct student watanabe = { 199, “渡辺隆史”, 175.0, 82.4 };
```

構造体メンバの参照法

- 構造体変数のメンバを参照するには，「. (ピリオド)」を使用する

```
printf( “学生番号：%d　氏名：%s　身長：%f　体重：%f¥n”,
                abe.id,  abe.name,  abe.height,  abe.weight );
```

構造体変数　↑　メンバ

ピリオド

- 構造体変数へ値を代入するときも同様

```
abe.height = 179.2;
abe.weight = 65.4;
strcpy( abe.name, “阿倍一郎”);
```

	int id	char name[20]	float height	float weight
abe	101	阿部一朗	178.5	63.5

構造体を指し示すポインタ

- 基本的な考え方は変数に対するポインタと同じ
- 宣言する場合は，ポインタ名の前に「＊」をつける

```
// 構造体テンプレートの宣言
struct student {
    int id;
    char name[20];
    float height;
    float weight;
};
// ポインタの宣言
struct student *tmp;
```

- ポインタへのアドレス代入の方法：

```
struct student abe;
tmp = &abe;
```

ポインタを使った構造体の参照

- ポインタを使って構造体のメンバを参照するには，
「-> (アロー演算子)」を使う

```
printf( “学生番号：%d　氏名：%s　身長：%f　体重：%f¥n”,
        tmp->id,  tmp->name,  tmp->height,  tmp->weight );
```

- 以下のように書いても同じ意味

```
printf( “学生番号：%d　氏名：%s　身長：%f　体重：%f¥n”,
        (*tmp).id, (*tmp).name, (*tmp).height, (*tmp).weight );
```

「*tmpはポインタtmpが指し示す変数」

- 書き方が煩雑なので，通常はアロー演算子を使う

文字列処理におけるポインタの実行例

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    int  i;
    char name[]="Nihon University";
    char *p;
    p=name;
    // name(ｱﾄﾞﾚｽ)をポインタpに代入して確定する
    // pが指すｱﾄﾞﾚｽが不定だと暴走する

    printf("*** case1 ***¥n");
    printf("address: name=%p, p=%p¥n", name, p);
    printf("value: name=%s, p=%s¥n¥n", name, p);

    printf("*** case2 ***¥n");
    putchar(*p);      //putchar：１文字表示
    putchar(*(p+1));
    putchar(*(p+2));//ポインタ(p+1)が指すアドレス
    putchar(*(p+3));//  の内容を表示
    putchar(*(p+4));
    putchar(*(p+5));
    putchar(*(p+6));
    putchar(*(p+7));
    putchar(*(p+8));
    printf("¥n¥n");
```

実行結果

```
*** case1 ***
address: name=0012FF40, p=0012FF40
（*コンピュータによって異なる）
value: name=Nihon University, p=Nihon
University

*** case2 ***
Nihon Uni
```

文字列のコピーと文字列の連結

```
// 文字列のコピー,
//文字列p2をp1にコピー
void fstrcpy(char *p1, char *p2)
{
   // p1にp2を1文字ずつコピー
   while(*p2  !=  '¥0')
   {
      *(p1++) = *(p2++);
   }
   // p1の末尾にヌル文字を代入
   *p1 = '¥0';
}
```

```
// 文字列の連結,
//文字列p2を文字列p1の後ろに連結
void fstrcat(char *p1, char *p2)
{
   // p1の末尾まで(ヌル文字が表れるまで)
   //ポインタの指す位置を動かす
   while( *p1 != '¥0' )
   {
      p1++;
   }

   // p1の末尾にp2を1文字ずつコピーして,
   //p1の末尾にヌル文字を代入
   while(*p2 != '¥0')
   {
      *(p1++) = *(p2++);
   }
   *p1 = '¥0';
}
```

クイックソート

- 基準値を決めて，それより大きい数と小さい数のグループに分割する
- 分割されたグループに対しても同じ処理を繰り返す

クイックソートプログラム

```
#include <stdio.h>
#define  NUM 10     //  データ数
void  quick_sort( int a[], int left, int right );
void  swap( int*, int* );

void main()
{
    //  入力データ
    int  a[NUM] = { 44, 89, 61, … , 93, 76 };

    //  クイックソート
    //quick_sort( a, 0, NUM-1 );
}

void  swap( int *a, int *b )
{
    int  tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
}
```

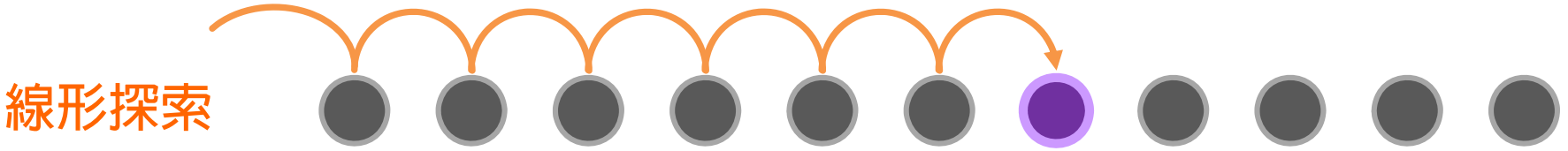
```
void  quick_sort( int a[], int left, int right )
{
    int  pl, pr, pivot;
    pl = left;
    pr = right;
    pivot = a[(pl+pr)/2];    //  基準値

    do{
        while( a[pl] < pivot ){ pl++; }   //  左カーソル
        while( a[pr] > pivot ){ pr--; }   //  右カーソル
        if( pl <= pr ){
            swap( &a[pl], & a[pr] );
            pl++;
            pr--;
        }
    } while( pl <= pr );

    if( left < pr )
        quick_sort( a, left, pr );    //  再帰呼び出し
    if( pl < right )
        quick_sort( a, pl, right );   //  再帰呼び出し
}
```

整列処理(クイックソート)のまとめ

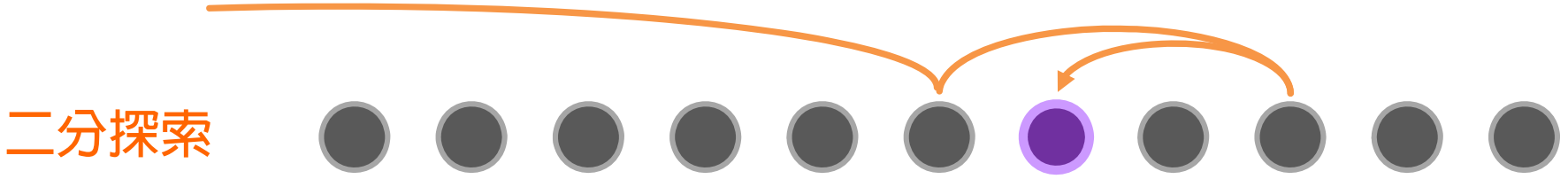
- 高速なソートアルゴリズムの一つ
  - 平均的には最も早いソート法
- 整列処理を行う範囲を次第に小さくする
  - ピボットの設定方法によって効率が異なる
- 再帰処理を使用することによって，効率的で短いソースコードを実現
- データ列の分割方法について理解すること
  - 左カーソル，右カーソルの動き
  - ピボットを挟んだデータの交換方法
  - 分割後の領域範囲

探索 (Search)

- よく知られた探索アルゴリズム:
  - 線形探索 (Linear Search)　1年次に学習済み
    - for文を使って先頭から順番に探査する
    - 探索の基本，遅い，ソート済みでないデータにも適用できる
  - 二分探索 (Binary Search)　本日学習
    - 高速，ソート済みデータにのみ適用できる



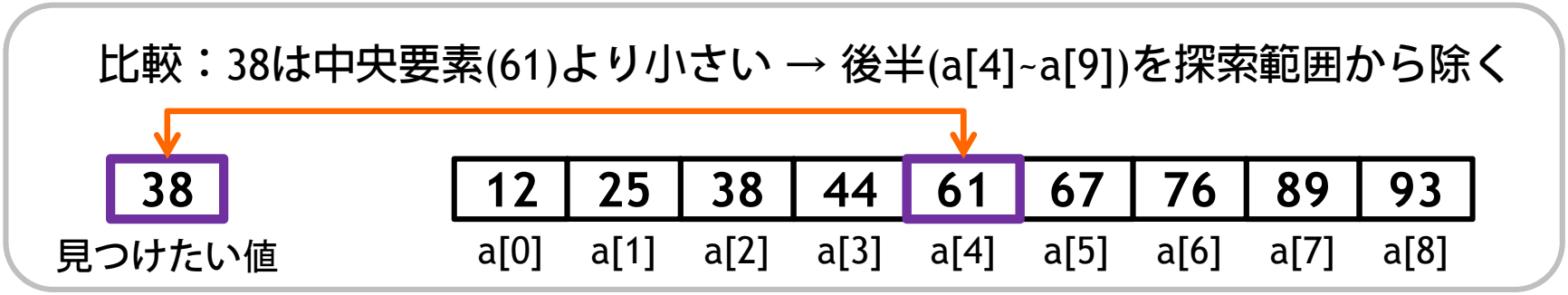
前から順に一つ一つ探していく



対象を半分ずつに絞って探していく

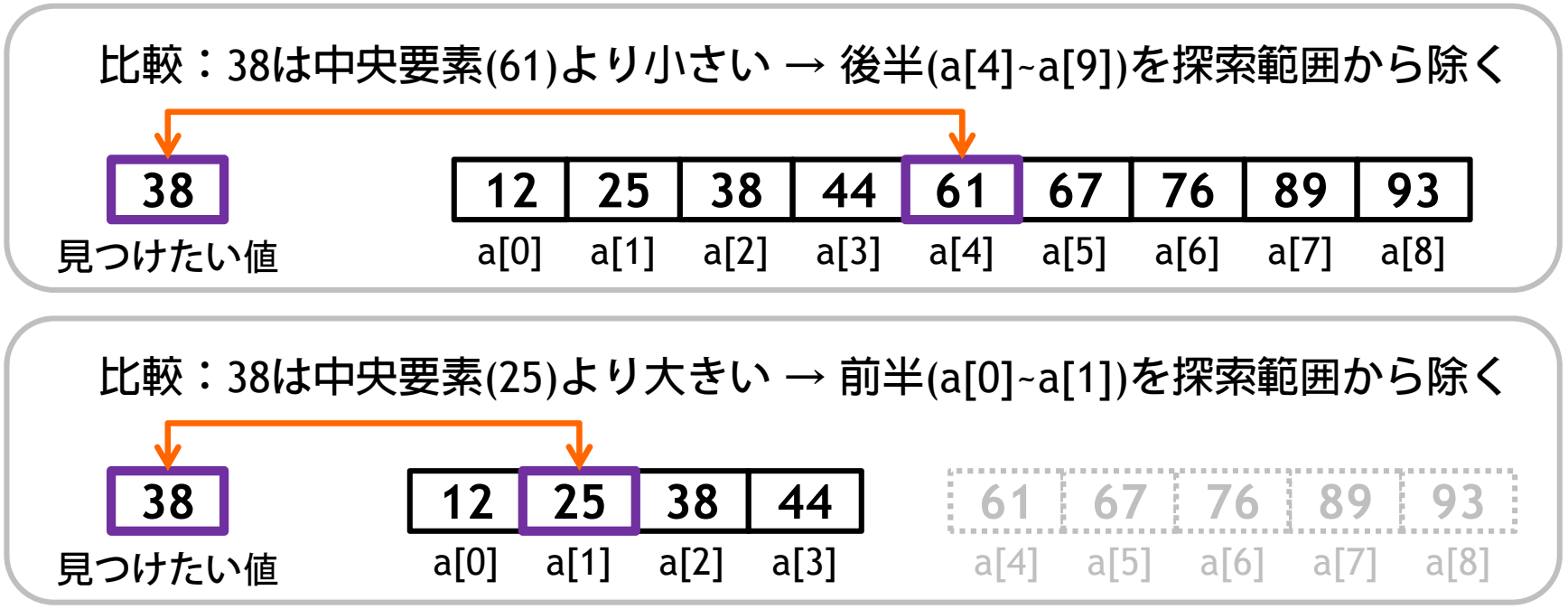
二分探索

- 整列済データを2つに分け，目的の探索範囲を絞り込む方法



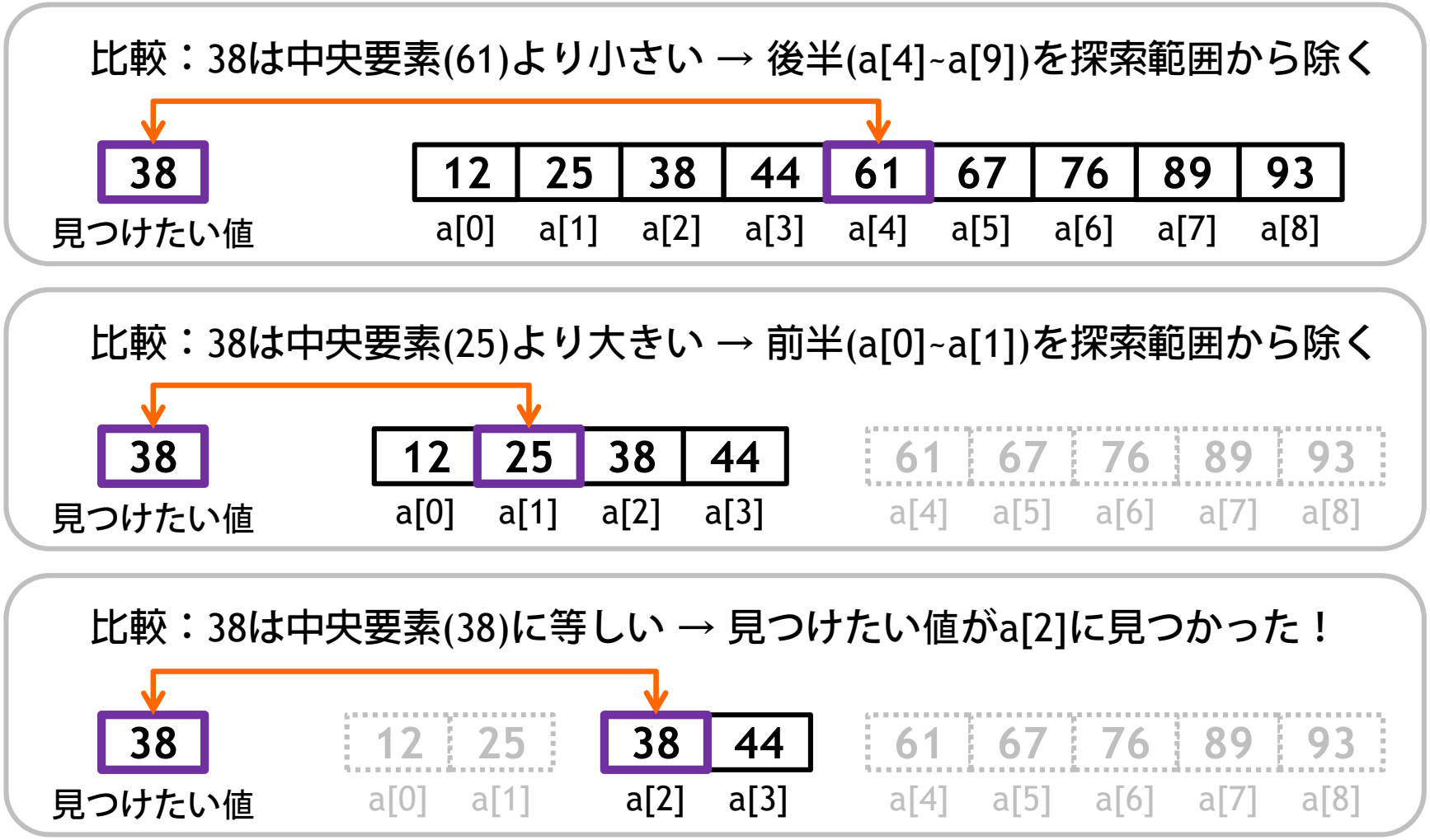
二分探索

- 整列済データを2つに分け，目的の探索範囲を絞り込む方法



二分探索

- 整列済データを2つに分け，目的の探索範囲を絞り込む方法



二分探索プログラム

```
#include  <stdio.h>
#define  NUM 9
int  binary_search( int  a[], int  key );

void  main()
{
    int  i, key, id;

    //  入力データ
    int  a[] = { 12, 25, 38, 44, 61, 67, 76, 89, 93 };

    //  見つけたい数値を「key」に入力
    printf("見つけたい数値を入力してください:");
    scanf( "%d", &key );

    //  二分探索
    id = binary_search( a, key );

    //  結果の表示
    if( id == -1 )
        printf( "探索に失敗しました. ¥n¥n" );
    else
        printf( "%dは%dに見つかりました. ¥n¥n", key, id );
}
```

```
int  binary_search( int  a[], int  key ) {
    int  pl, pr, pc;

    //  初期化：配列の最初と最後の番号を格納
    pl = 0;    pr = NUM-1;

    //  繰り返し処理
    while(1){

        //  終了条件1(左右のカーソルが逆転，探索失敗)
        if( pl > pr )  return -1;

        //  1. 配列中央の番号を格納
        pc = (pl+pr)/2;

        //  2. 中央が見つけたい値よりも小さければ
        if( a[pc] < key )
            pl = pc+1;    //  探索範囲縮小(小さい方を削除)

        //  3. 中央が見つけたい値よりも大きければ
        else if( a[pc] > key )
            pr = pc-1;    //  探索範囲縮小(大きい方を削除)

        //  終了条件2(中央とkeyが一致，探索成功)
        else if( a[pc] == key )    //
            return  pc;    //  配列中央の番号を返す
    }
}
```

二分探索のまとめ

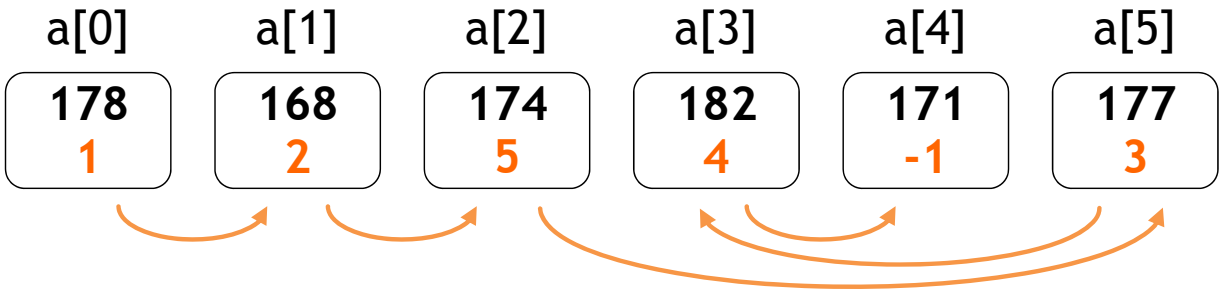
- 線形探索に比較すると非常に高速な探索が可能
- ソート済みデータのみ適用可能であり，事前にデータの整列処理が必要
- 探索範囲の絞り込み手順を理解
  - 右カーソル「pr」，左カーソル「pl」の動き
  - 中央カーソル「pc」の設定

リスト構造 (List)

- リストの例：
  1. 配列を使用したリスト：
     - 配列番号(インデックス)が次のノードへのつながり情報となる
     - つながり情報の変更に時間がかかる
  2. 構造体を使用したリスト：
     - 構造体のメンバ変数に次のノードへのつながり情報（ポインタ）を持たせる
     - つながり情報の変更が簡単かつ高速

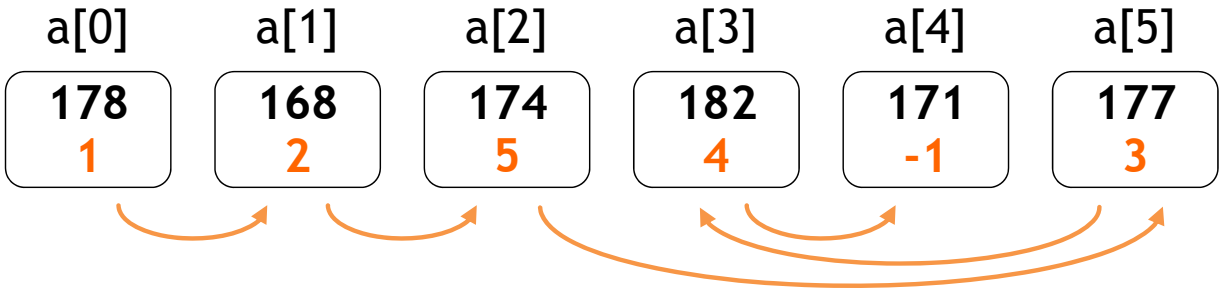
構造体を使用したリスト

配列番号順

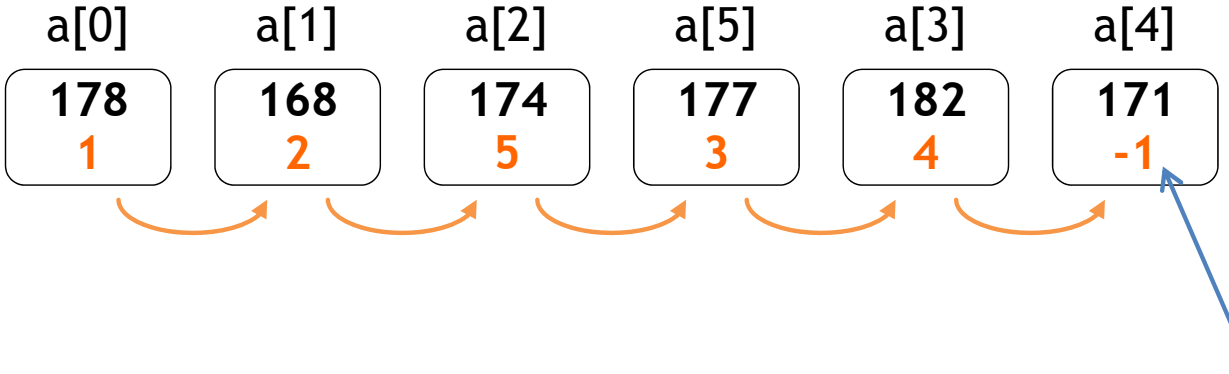


構造体を使用したリスト

配列番号順



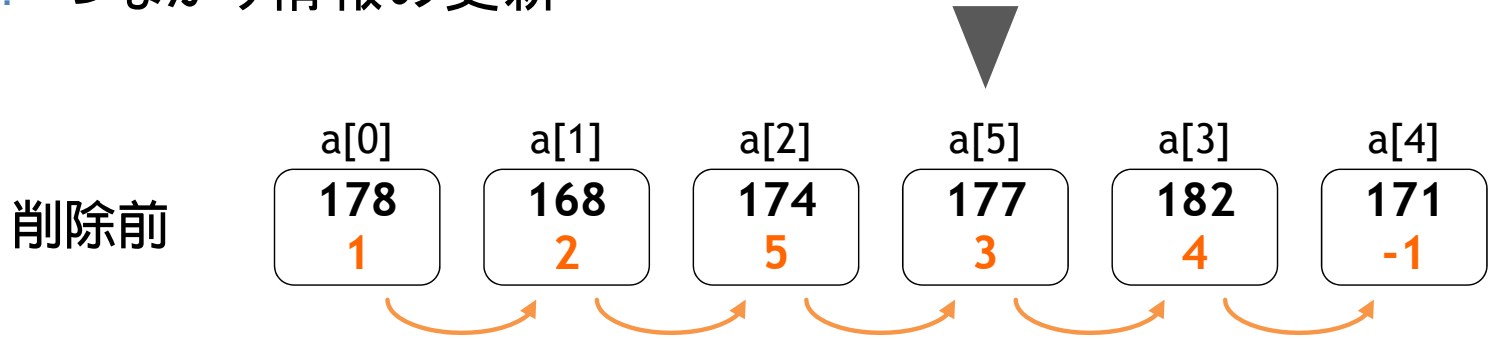
つながり順



末尾には-1が入る

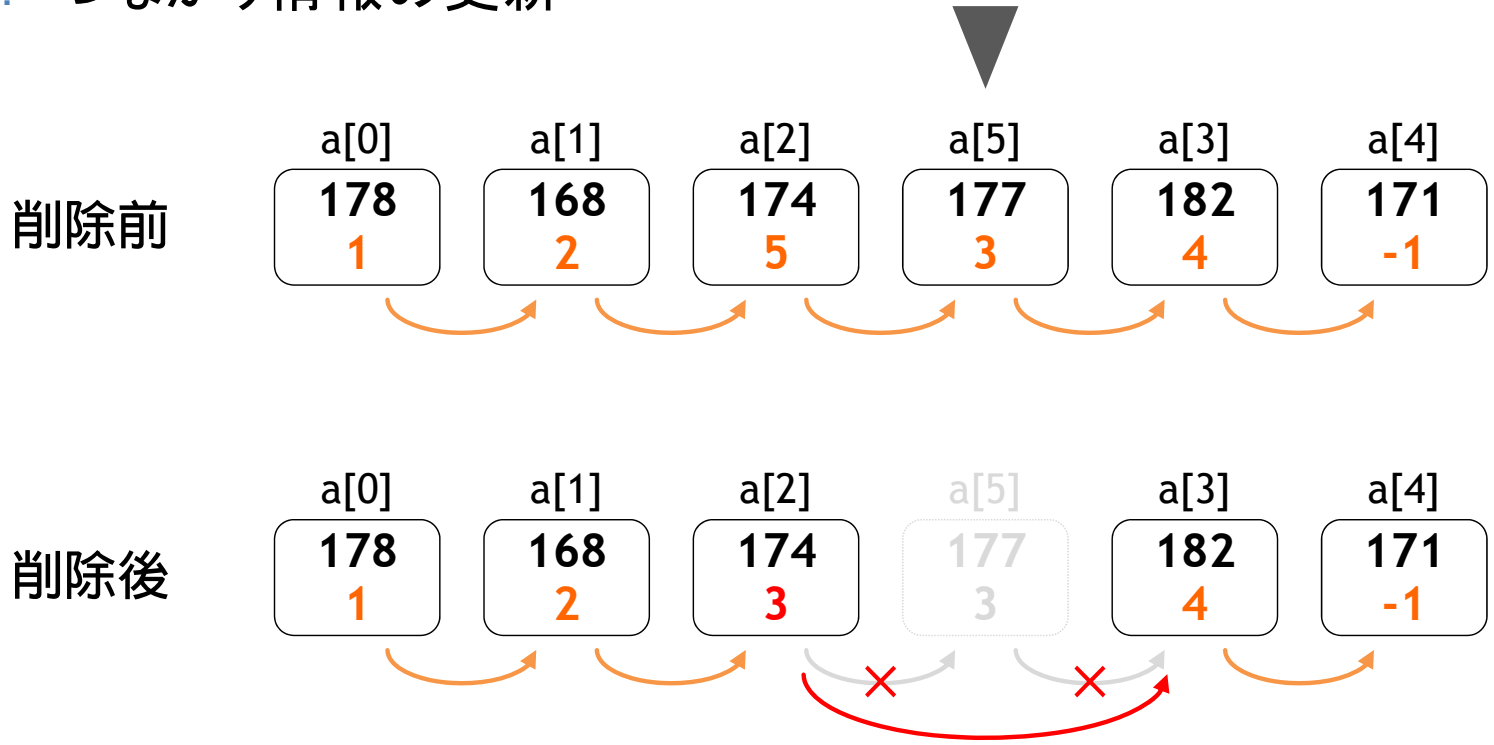
データの削除方法

- 手順:177 (4番目)を削除する場合
  1. つながり情報の更新



データの削除方法

- 手順:177 (4番目)を削除する場合
  1. つながり情報の更新



注意：リストからは削除されるが，物理的には削除されない！

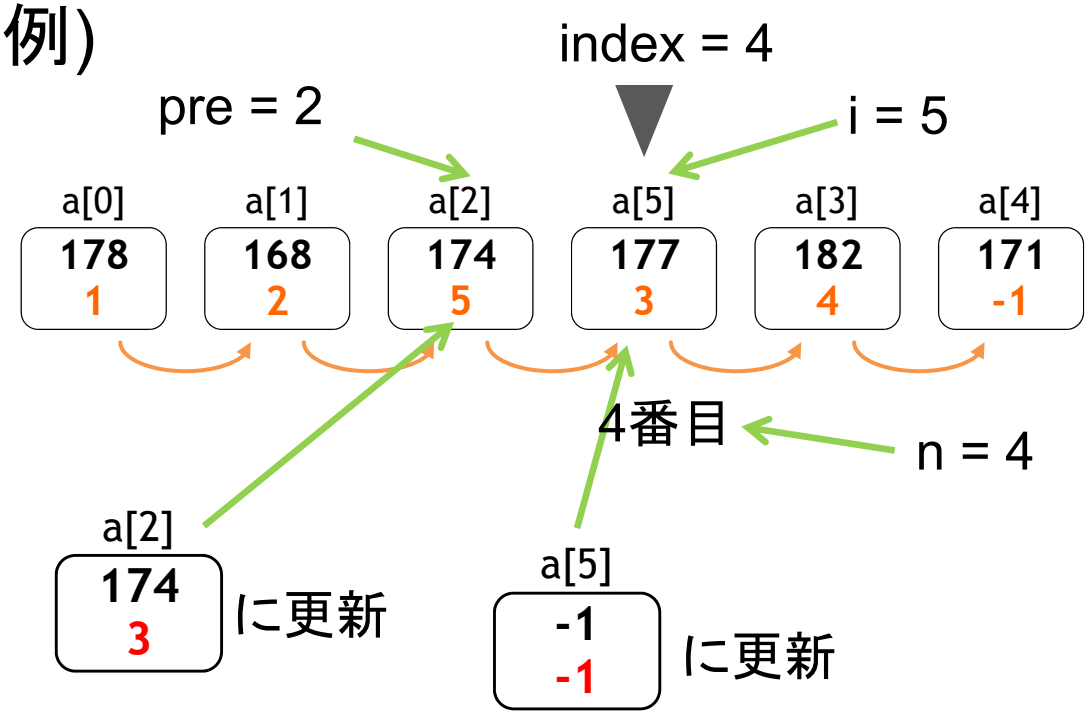
リストからの削除 (Delete)


```
int Delete (int index, MyList data[]){
  int n = 0;
  int i = 0;
  int pre = -1;

  // 削除対象範囲外
  if(index <= 1) return -1;

  while(1){
    if(data[i].IntData != -1){
      n++; // 先頭から何番目か
      // 削除場所に到達(データを削除)
      if(n == index){
        data[pre].NextIndex =
                        data[i].NextIndex;
        data[i].IntData = -1;
        data[i].NextIndex = -1;
        return 0;
      }
    }
      pre = i;
      i = data[i].NextIndex;

      // 末尾まで到達したが，削除データが無い
      if(i == -1)  return -1;
    }
  }
```

削除対象リスト

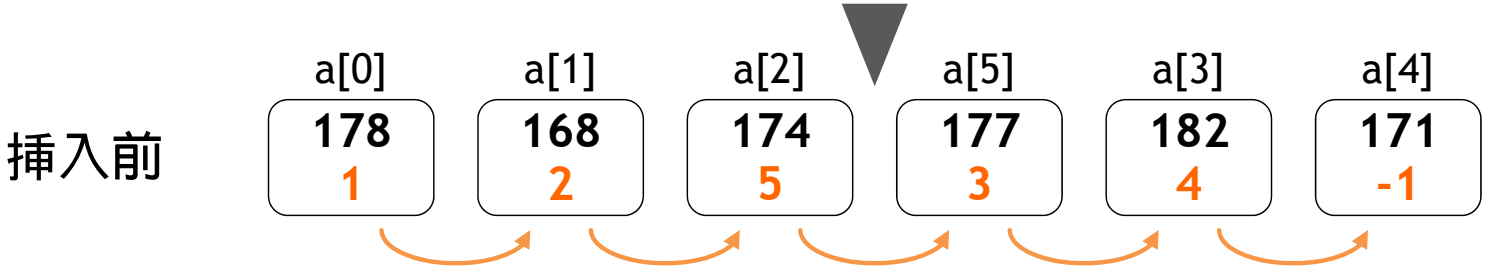
削除するデータの場所

例)



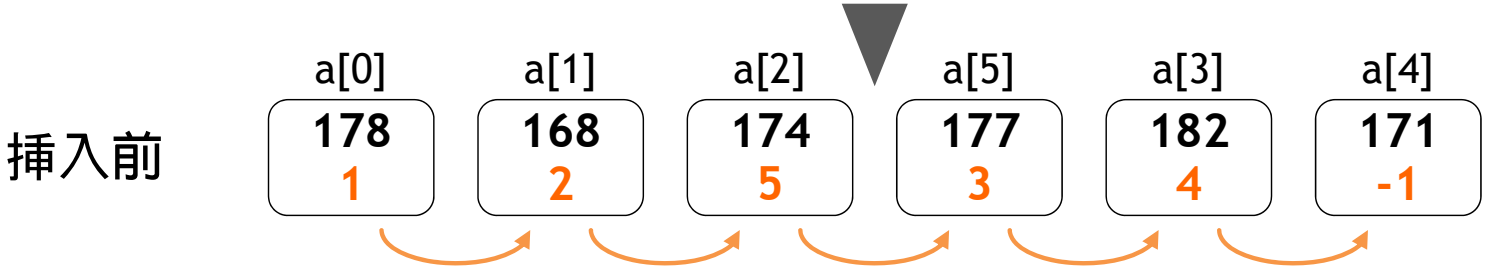
データの挿入方法

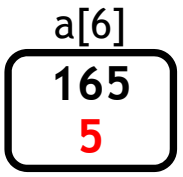
- 手順：174と177の間(4番目)に165を挿入する場合



データの挿入方法

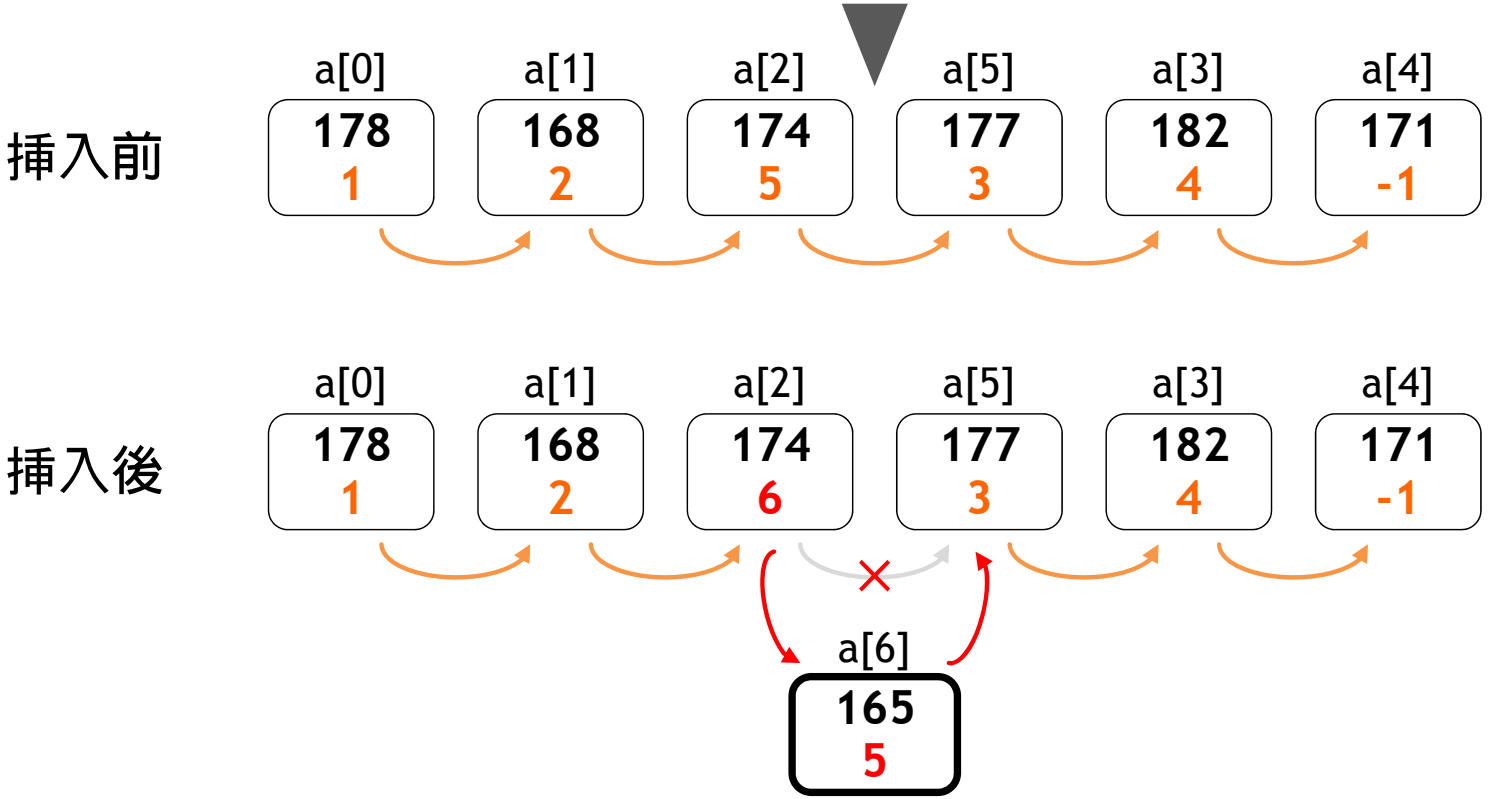
- 手順:174と177の間(4番目)に165を挿入する場合
  1. 未使用の配列にデータを格納





データの挿入方法

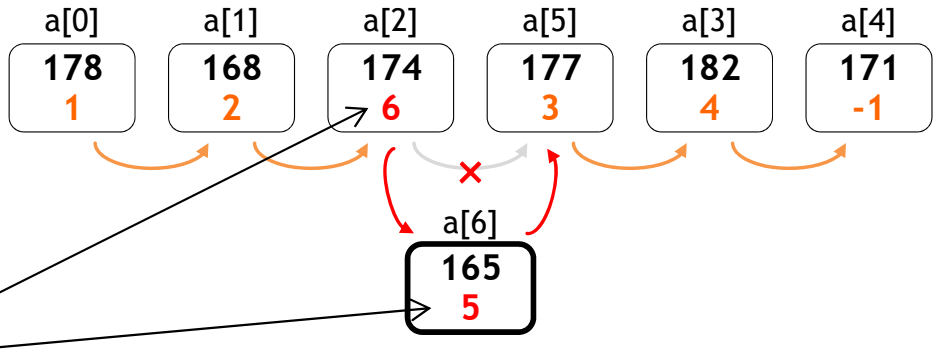
- 手順：174と177の間(4番目)に165を挿入する場合
  1. 未使用の配列にデータを格納
  2. つながり情報の更新



リストへの挿入 Insert

555 ← IntData：データ
3 ← NextIndex :つながり情報

```
int Insert(int index, int ins_data, MyList data[]){
// index : 挿入場所，ins_data : 挿入データ, data :  リスト
  int blank = 0; // 配列の空いている場所
  int n = 0;
  int i = 0, pre = -1;

  //空いている箇所を探す
  for(blank = 0; blank < MAX; blank++)
    if(data[blank].IntData == -1)  break;

  // 削除対象範囲外 or 配列が満杯で挿入不可能
  if(index <= 1 || blank == MAX) return -1;

  while(1){
    if(data[i].IntData != -1){
      n++; // データの挿入場所まで移動
      if(n == index){// データを挿入
        data[blank].IntData = ins_data;
        data[blank].NextIndex =
                          data[pre].NextIndex;
        data[pre].NextIndex = blank;
        return 0;
      }
      pre = i;
      i = data[i].NextIndex;

      // 末尾に到達したが，
      // 挿入場所に辿り着かなかった
      if(i == -1){
        return -1;
      }
    }
  }
}
```



つながり情報の更新

pre= 2, blank = 6

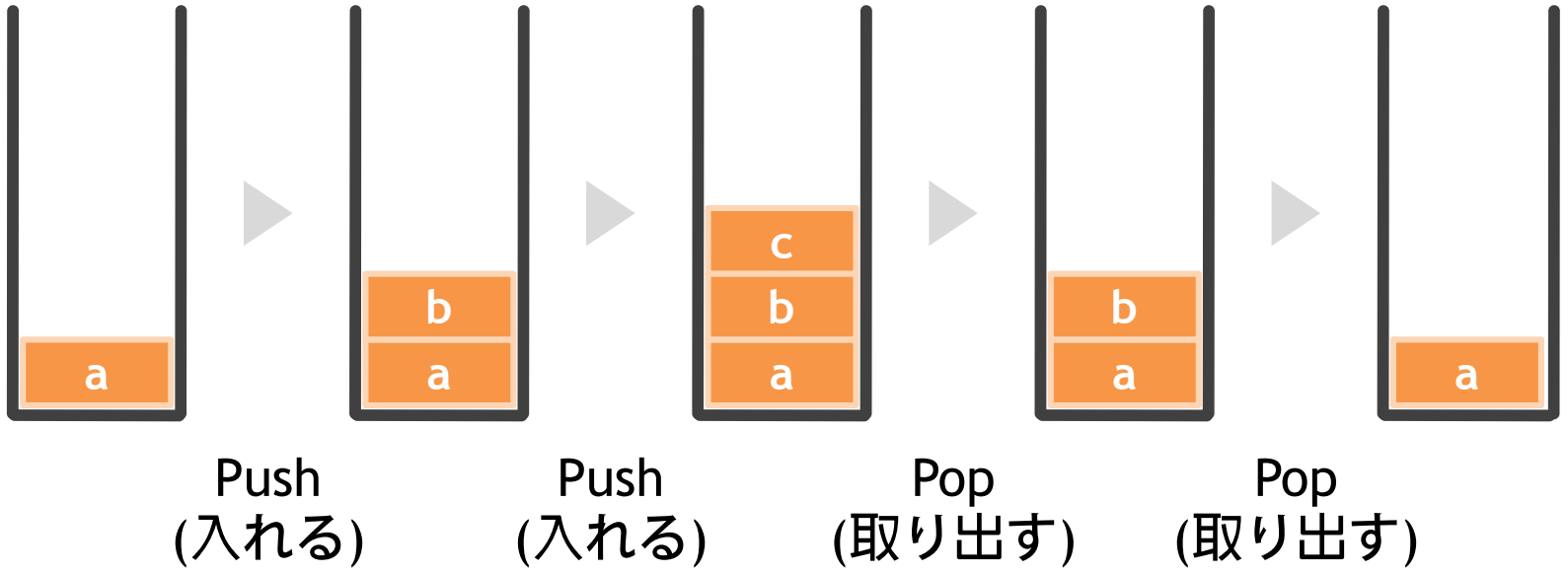
構造体を使用したリスト　まとめ

- 構造体を使用してつながり情報を含むリストを実現

- リスト(構造体)のデータ表示・データ挿入・データ削除
  - つながり情報を利用
  - データシフトを行う必要がない！
  - 処理が高速

スタック

- データを積み上げる
- LIFO: Last In First Out (最後に入ったものが最初に出る)

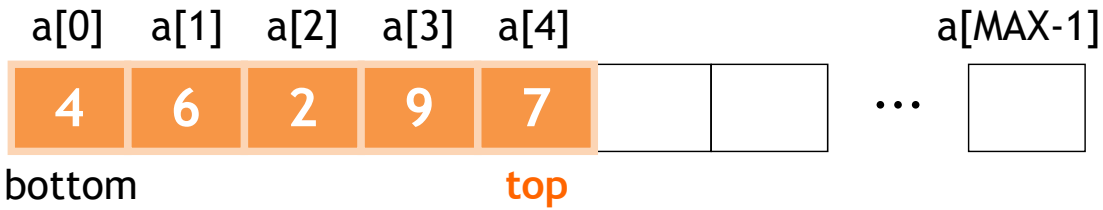


スタックの実現(変数宣言)

```
#include <stdio.h>

// スタックの配列による実現
#define  STACK_SIZE  5          // スタックの最大データ数
#define  NO_DATA      -1        // 無効ノード
int  stack[STACK_SIZE];         // スタックの実体配列
int  sp= -1;                    // スタックポインタ初期化 (トップ)
                                // 配列末尾の添え字を示す

// 関数のプロトタイプ宣言
void  ShowStack();              // スタック表示
int  Push( int data );          // プッシュ
int  Pop();                     // ポップ
```

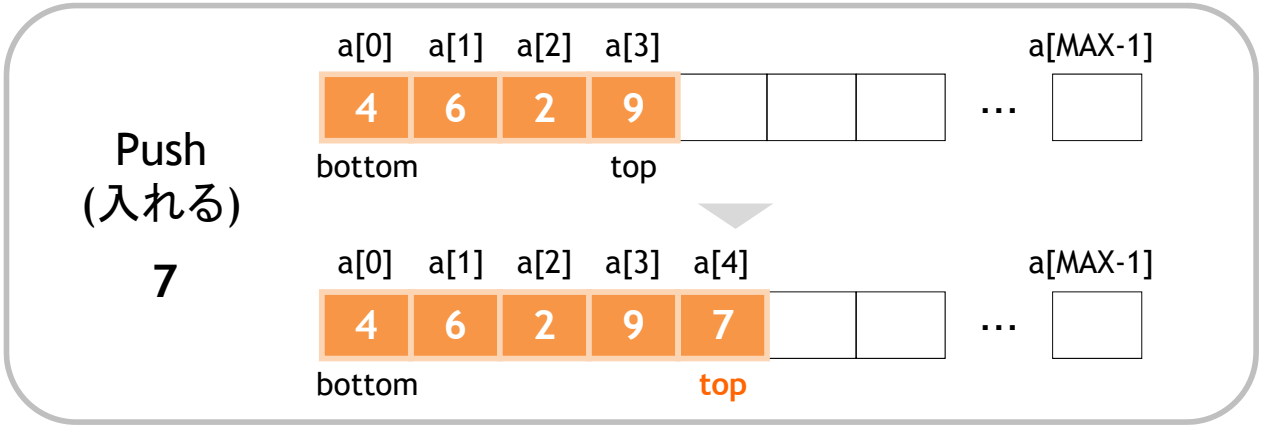
スタック内容の表示関数

```
void ShowStack()
{
        int  i;
        printf("Stack : ");
        for ( i=0; i<=sp; i++ ){          //  sp:スタックポインタ
                printf("[%d]", stack[i]);
        }
        printf(" ¥n");
}
```



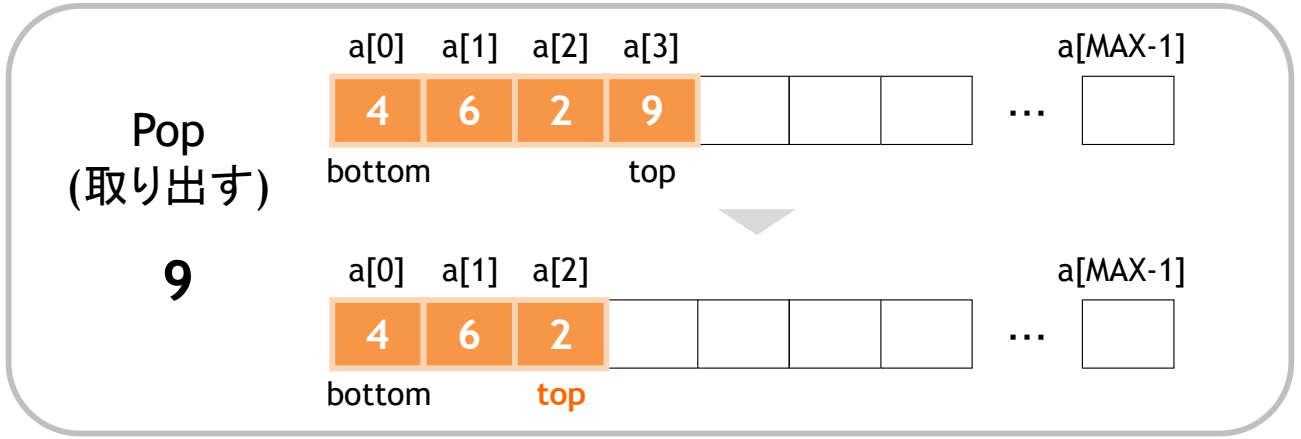
スタックへの プッシュ関数

```
int  Push( int  data )
{
  sp++;                          // スタックポインタを +1する
  if ( sp >=STACK_SIZE ){        // 配列サイズ超過？
    sp--;
    return (-1);                 // 異常終了
  }else{
    stack[sp] = data;            // スタックにデータを積む
    return  0;                   // 正常終了
  }
}
```



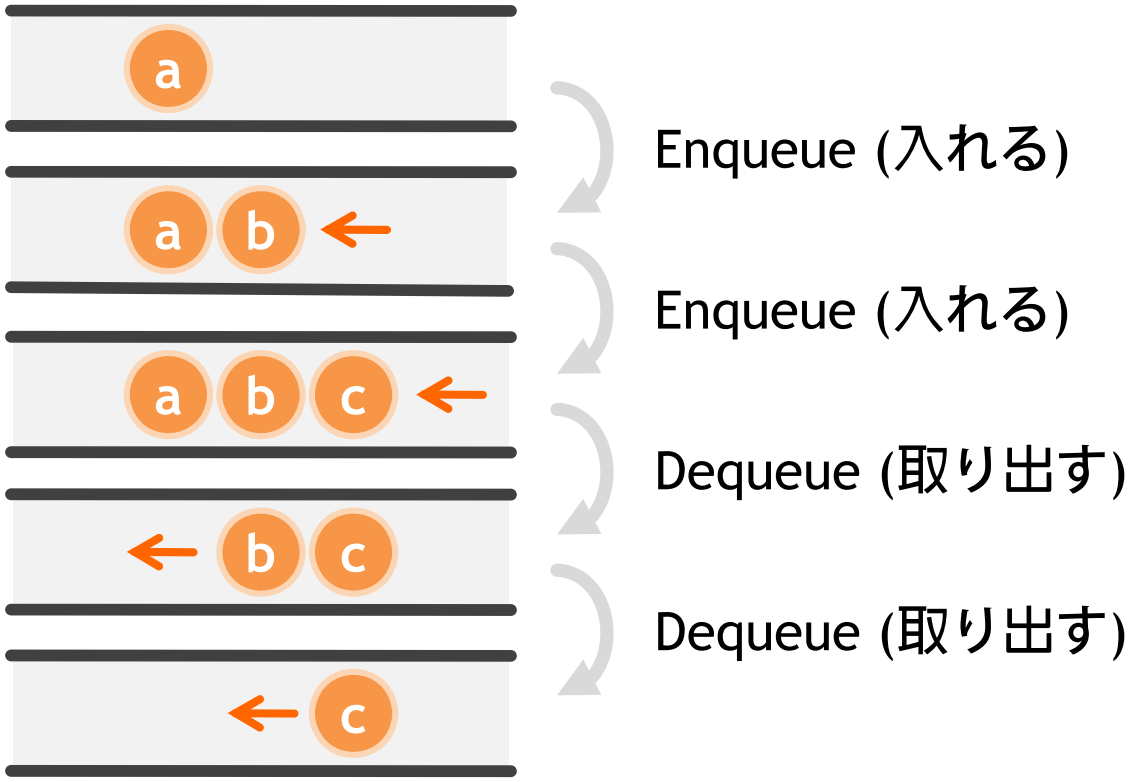
スタックからのポップ関数

```
int  Pop()
{
  if ( sp<0 ){                  //  スタックポインタは負か？
    return (-1);                //  異常終了
  }else{
    int  data;
    data = stack[sp];           //  スタックの値をdataに保存
    sp--;                       //  スタックポインタを-1する
    return (data);              //  ポップしたデータを返却する
  }
}
```

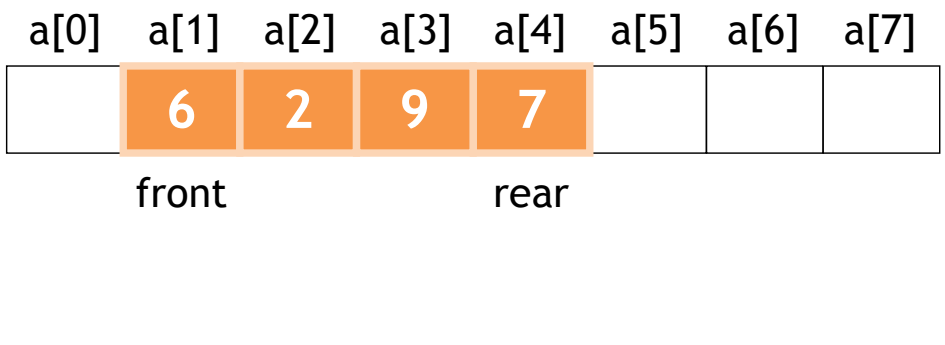


キュー

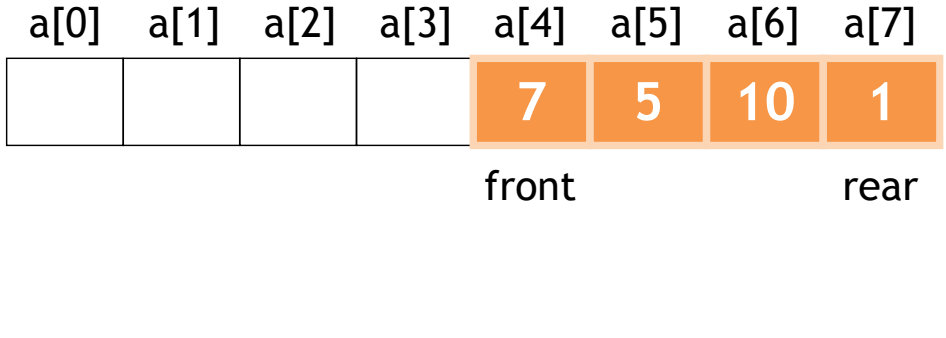
- データを入ってきた順に並べる，待ち行列 (スーパーのレジと同じ)
- FIFO: First In First Out (最初に入ったものが最初に出る)



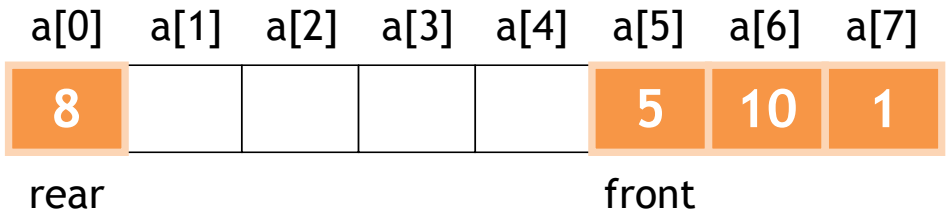
リングバッファの概念（循環配列）



EnqueueとDequeueを繰り返すと，
データが配列の末尾に到達し，
それ以上データが格納できない．



使用していない前半の配列を使用・
配列の最初と最後が接続されている
ものとみなす・（リングバッファ）



キューの配列による実現（変数宣言）

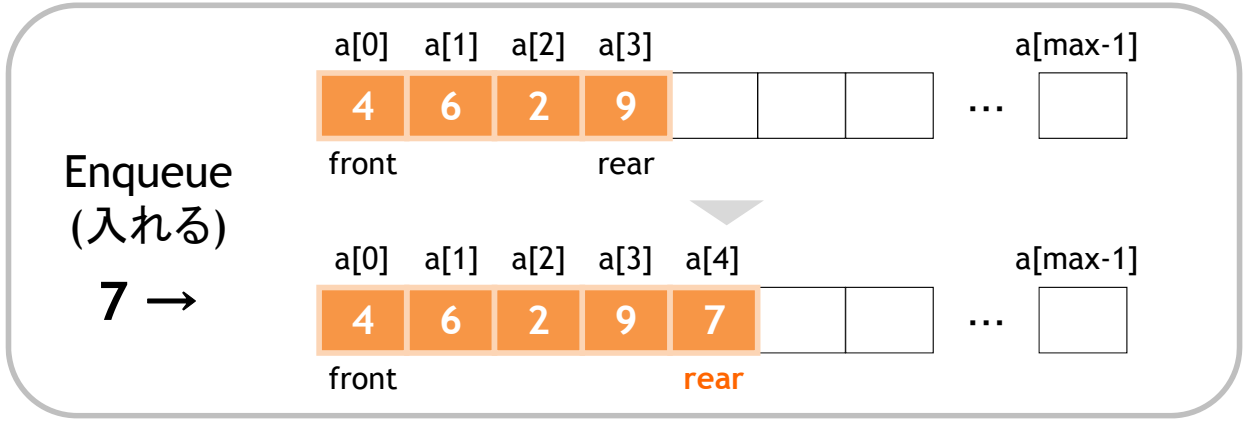
```
#include <stdio.h>

                                    //  キューの配列による実現
#define QUEUE_SIZE   5              //  キューの最大データ数
#define NO_DATA  -1                 //   無効データ定数の定義

int  queue[QUEUE_SIZE];             //  キューの実体の配列
int  front = 0;                     //  キューのフロント
int  rear = 0;                      //  キューのリア
int  num = 0;                       //  キューのデータ数

                                    //  関数のプロトタイプ宣言
void  ShowQueue();                  //  キュー表示
int  EnQueue(int data);             //  エンキュー
int  DeQueue();                     //  デキュー
```

キュー内容の表示関数

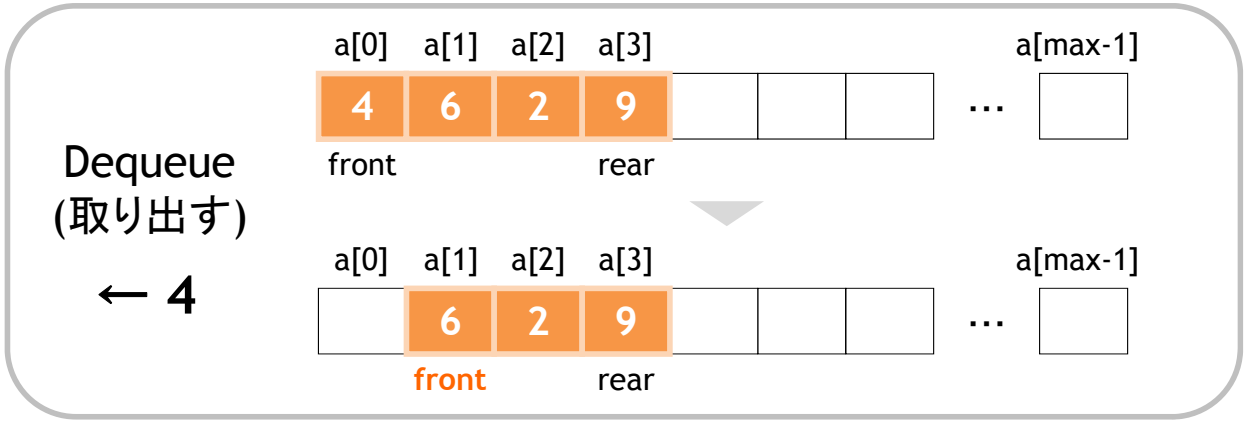
```
void ShowQueue()
{
  int i;

  printf(“Queue : ”);                          // キュー内容の表示
  for ( i=0; i<QUEUE_SIZE; i++ ){
    if ( queue[i] !=NO_DATA ) {               // 有効データか？
      printf(“[%d]”,queue[i]);                // 有効データ表示
    }
    else{
      printf(“[   ]”);                        // 無効データ(空)表示
    }
  }
  printf(“ ¥n");
}
```

エンキュー関数

```
int  EnQueue(int data)    //  エンキューするデータ：data
{
  if ( rear >= QUEUE_SIZE ){
    rear = 0;    //  rearが配列サイズを超えたらリングバッファを設定する
  }

  if ( num >= QUEUE_SIZE ){  //  データ数が配列サイズ超過？
    return (-1);   //  異常終了
  }
  else{              //  データ数が配列サイズを超えていなければ
    queue[rear] = data;  // 最後尾にデータを入れる(図示の場合は添え字4に入れる)
    rear++;          //  リア+1
    num++;           //  データ数+1
    return (0);      //  正常終了
  }
}
```



デキュー関数




```
int  DeQueue()
{
  if ( num == 0 ){
    return (-1);      // データなし：異常終了
  }
  else{
    int  data;
    data = queue[front];            //  デキュー：frontデータをdataに保存
    queue[front] = NO_DATA;  //  frontデータに無効データを入れる
    num--;                                //  データ数-1
    front++;                             //  front+1

    if ( front == QUEUE_SIZE )
      front = 0;        //  frontが配列サイズを超えたらリングバッファを設定
    return (data);    //  デキューしたデータを返却
  }
}
```

スタック・キューのまとめ

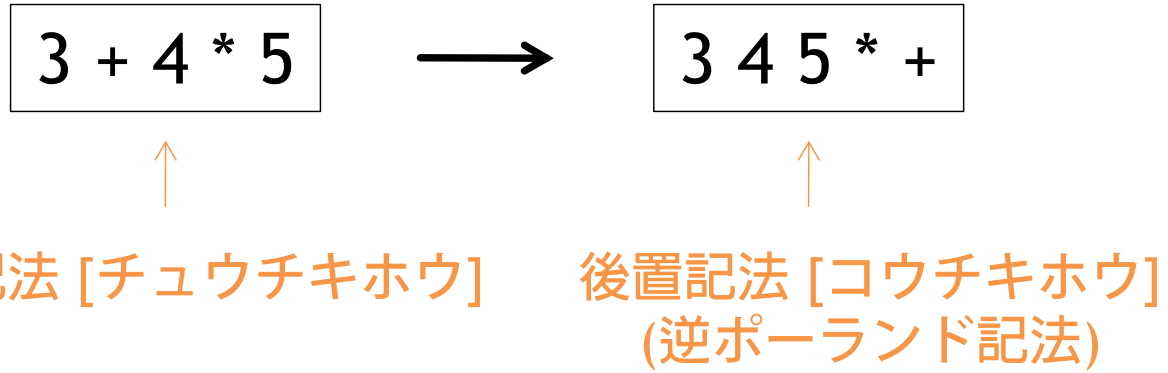
- スタック・キューのデータ構造
  - 用語定義：
    - LIFO，プッシュ，ポップ，トップ，ボトム
    - FIFO，エンキュー，デキュー，フロント，リア，リングバッファ

- 配列を使用してスタック・キューを実現
  - LIFO，FIFOの実現
  - リングバッファの実現

逆ポーランド記法 (後置記法)

- 「3 + 4 * 5」をコンピュータはどうやって計算するか？

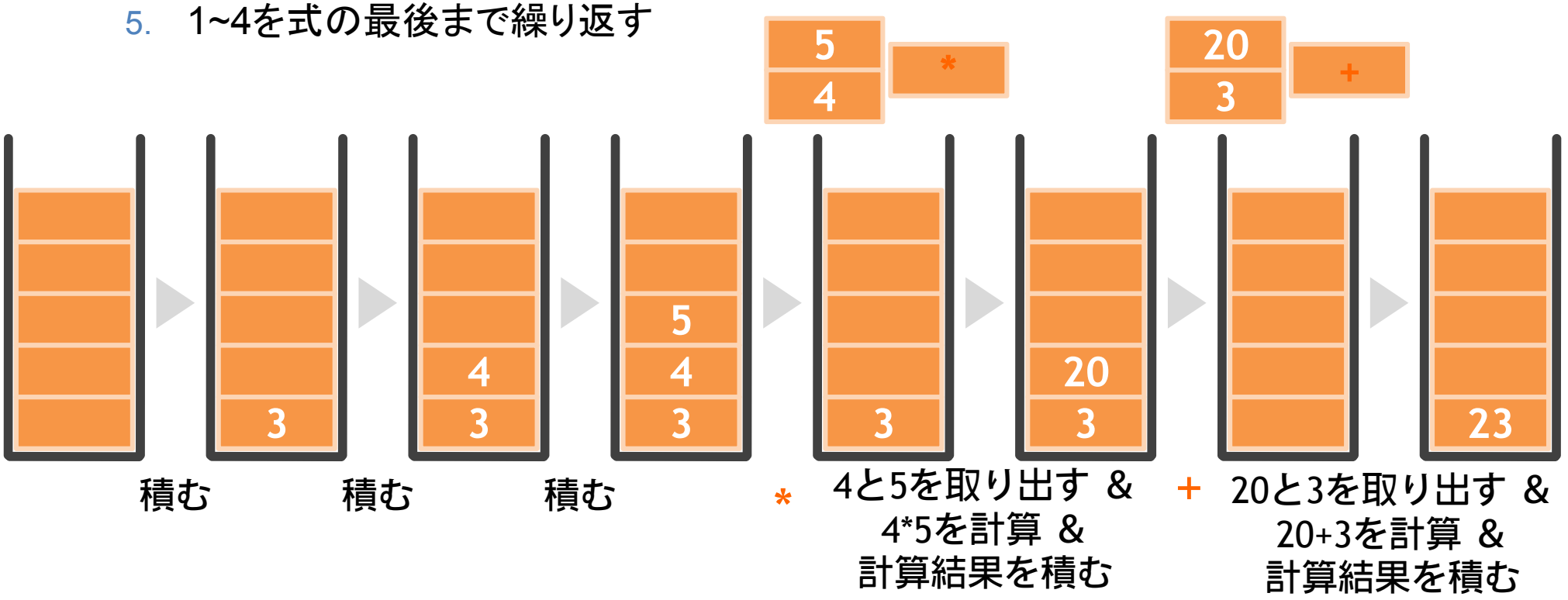
- コンピュータはそのまま計算できないので，特別な記法に変換する必要がある！



後置記法で書かれた式を計算する

- ルール (手順):
  1. 左から1文字ずつ読む
  2. 数字が来たらスタックに積む
  3. 演算子(+ - * /)が来たら，スタックの上から2つの数字を取り出して演算する
  4. 演算したら結果をスタックの上に積む
  5. 1~4を式の最後まで繰り返す

「３４５＊＋」の例



期末試験のお知らせ

- 7/25(金)に，期末試験を実施します
  - 1講時（5511教室）：筆記試験
    - 筆記用具以外持ち込み不可
  - 2講時（5511教室）：実技試験
    - 教科書，参考書，ノート，講義資料は持ち込み可
    - 過去に作成したプログラムの閲覧やコピーは禁止，単位取得不可

- 講義内容全て
  - 演習問題，講義資料中の練習問題の復習をしてください