

データ構造入門及び演習

12回目: スタック・キュー

2014/07/04

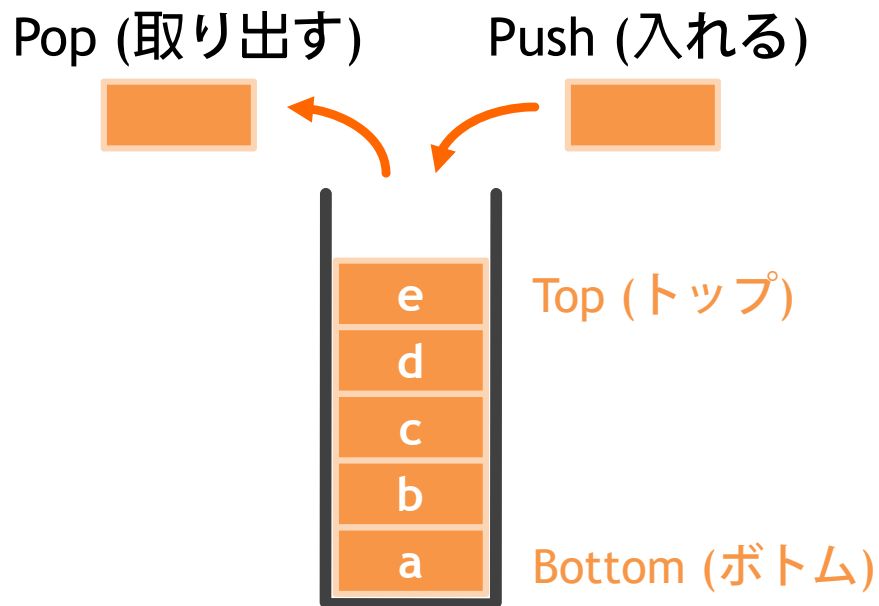
担当: 見越 大樹

61号館304号室

スタック・キュー

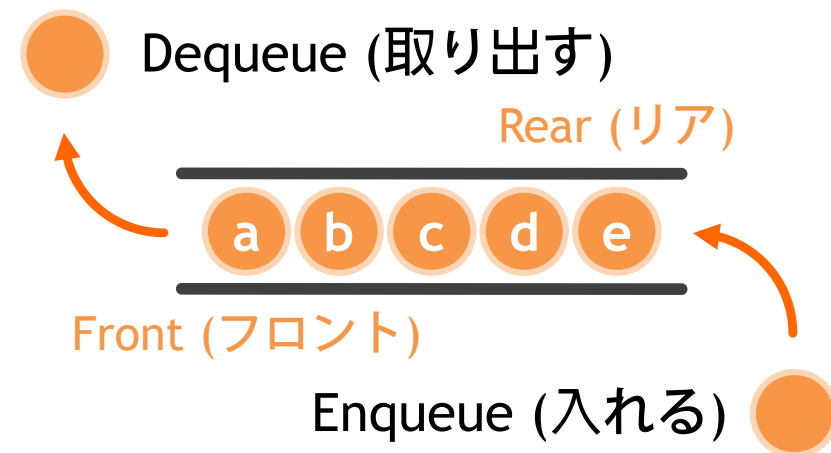
- データを一時的に蓄積する際のデータ構造

スタック (Stack)



LIFO: Last In First Out
(最後に入ったものが最初に出る)

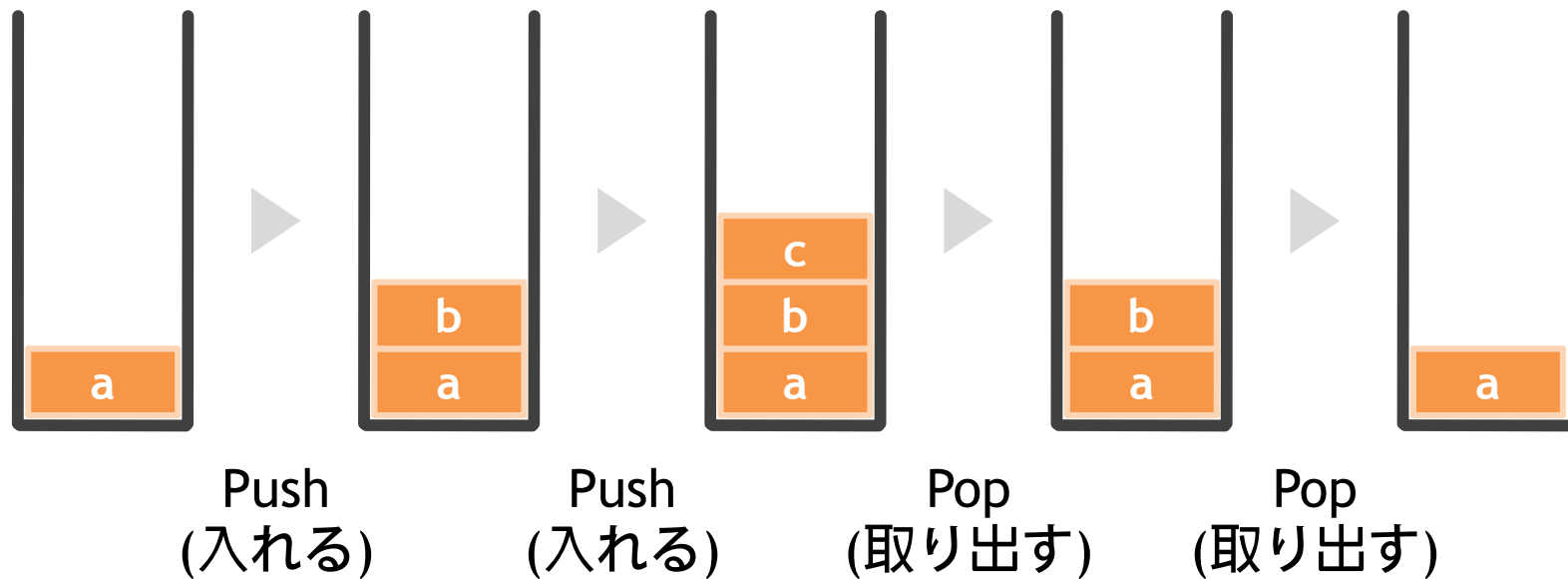
キュー (Queue)



FIFO: First In First Out
(最初に入ったものが最初に出る)

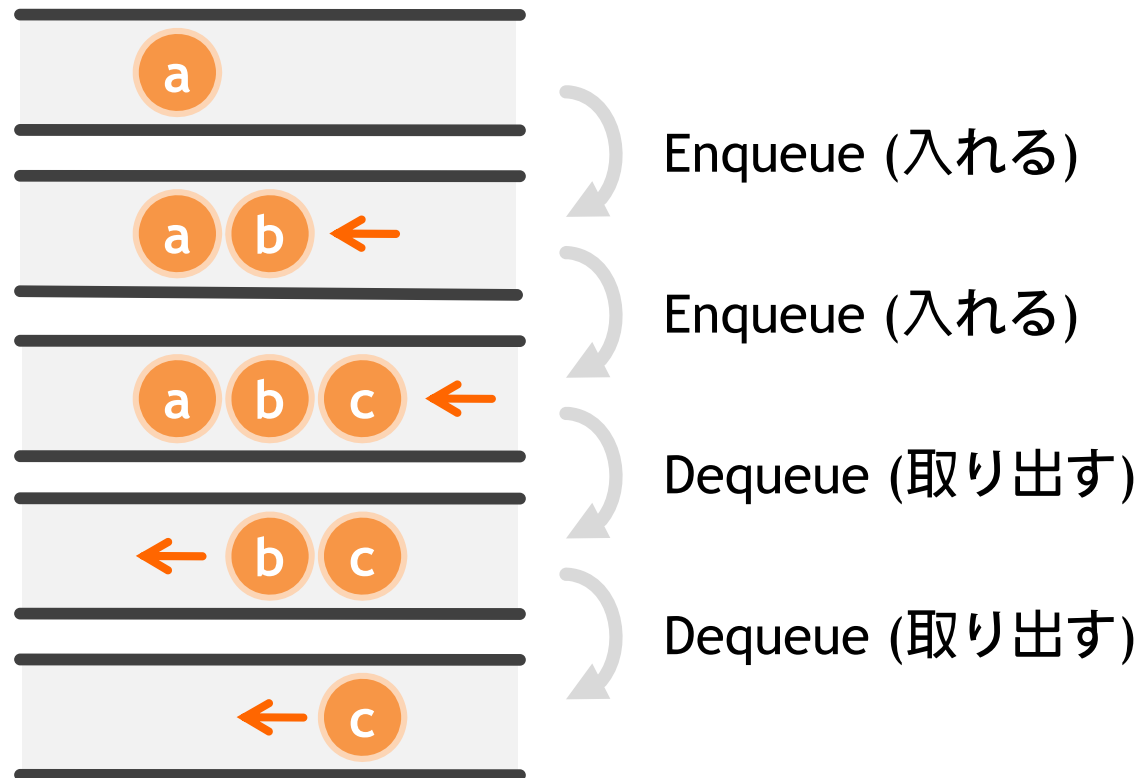
スタック

- 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)



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: **F**irst **I**n **F**irst **O**ut (最初に入ったものが最初に出る)

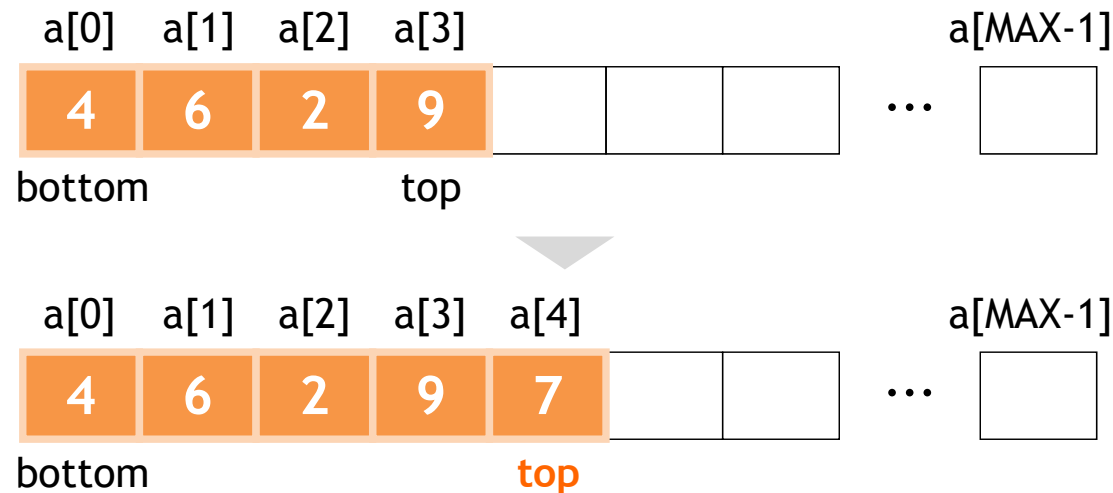


配列によるスタックの実現

配列サイズ：MAX

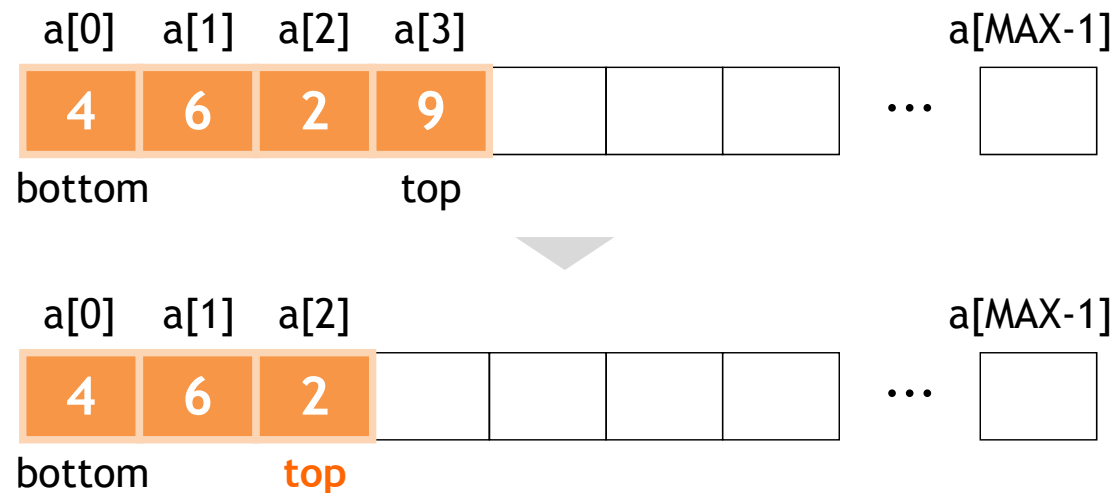
Push
(入れる)

7



Pop
(取り出す)

9

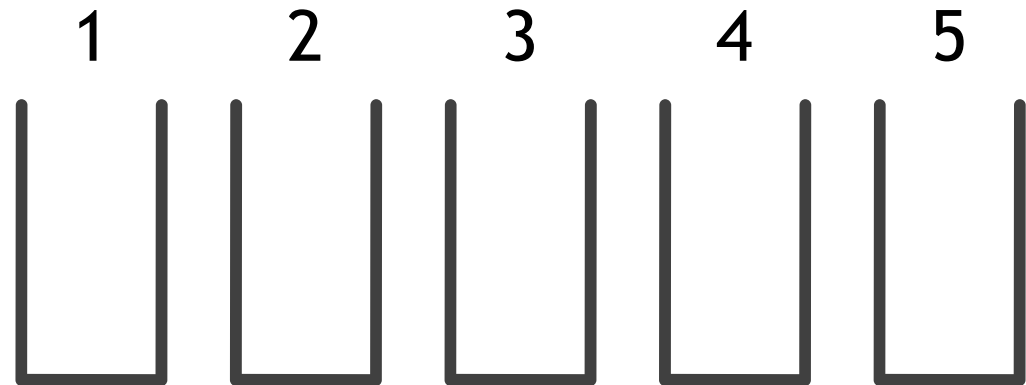


問題1 スタック

1. Push(A)
2. Push(B)
3. Push(C)
4. $x = \text{Pop}()$
5. $y = \text{Pop}()$

$x =$

$y =$



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
```

```
#define NO_DATA -1
```

```
int stack[STACK_SIZE];
```

```
int sp= -1;
```

```
// 関数のプロトタイプ宣言
```

```
void ShowStack();
```

```
int Push( int data );
```

```
int Pop();
```

```
// スタックの最大データ数
```

```
// 無効ノード
```

```
// スタックの実体配列
```

```
// 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
```

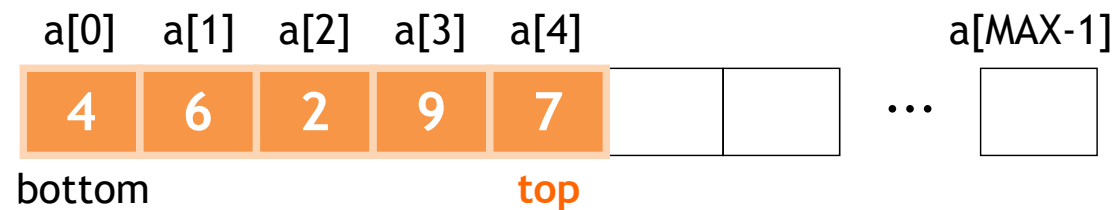
```
// スタック表示
```

```
// プッシュ
```

```
// 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");  
}
```

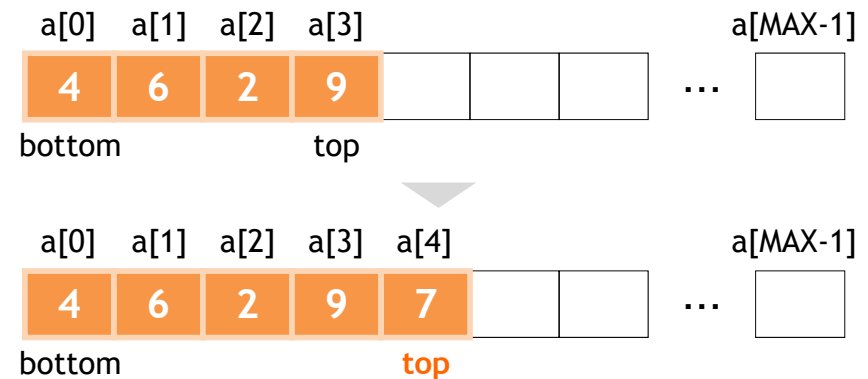


スタックへのプッシュ関数

```
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
    }
}
```

Push
(入れる)

7

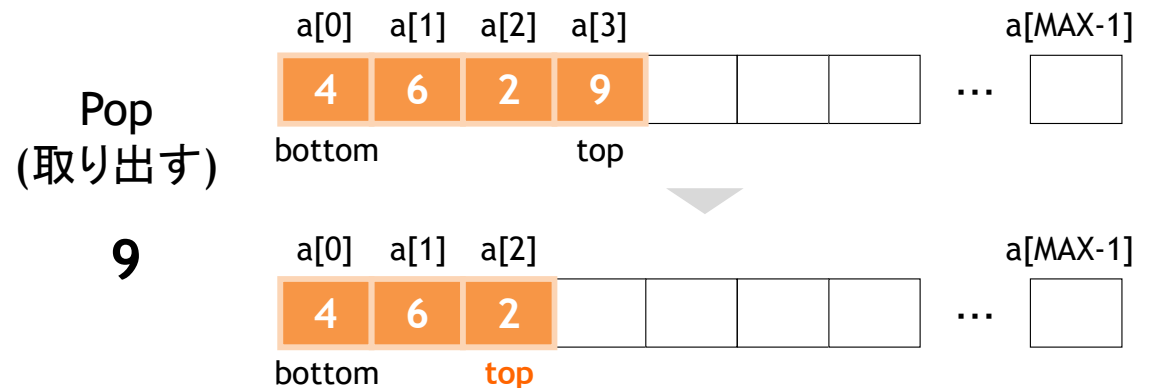


スタックからのポップ関数

```
int Pop()
{
    if ( sp < 0 ){
        return (-1);
    }else{
        int data;
        data = stack[sp];
        sp--;
        return (data);
    }
}
```

// スタックポインタは負か？
// 異常終了

// スタックの値をdataに保存
// スタックポインタを-1する
// ポップしたデータを返却する



スタックの実現 (main関数)

```
void main()
{
    int i, input, ret, data;

    // スタックの初期化
    for( i=0;i<STACK_SIZE;i++ ){ stack[i] = NO_DATA; }

    while(1){
        // 操作の指定
        printf("(1) Push:1   (2) Pop:2   (3) Exit:3¥n");
        scanf("%d",&input);

        // 1:プッシュを行う
        if( input == 1 ){
            printf("data -> ");
            scanf("%d",&data);
            ret = Push( data );
            if( ret != -1 )
                ShowStack();
            else
                printf("Stack Overflow!!¥n¥n");
        }

        // 2:ポップを行う
        else if( input == 2 ){
            ret = Pop();
            if( ret != -1 ){
                printf("Pop data is %d¥n", ret );
                ShowStack();
            }else{
                printf("Stack Empty!!¥n" );
            }
        }

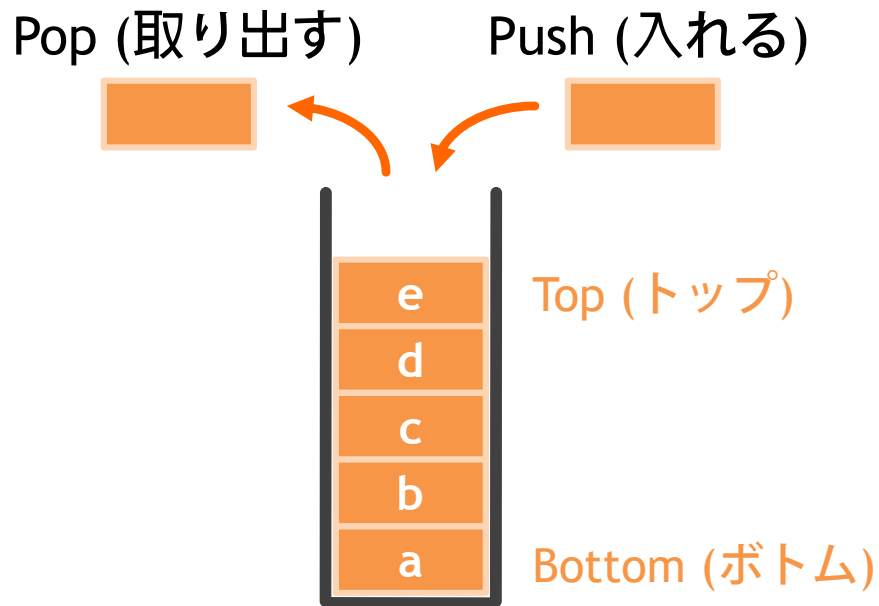
        // 3:終了
        else if( input == 3 )
            break;

        // 他:入力ミス！
        else
            printf("再入力してください！ ¥n¥n");
    }
}
```

スタック・キュー

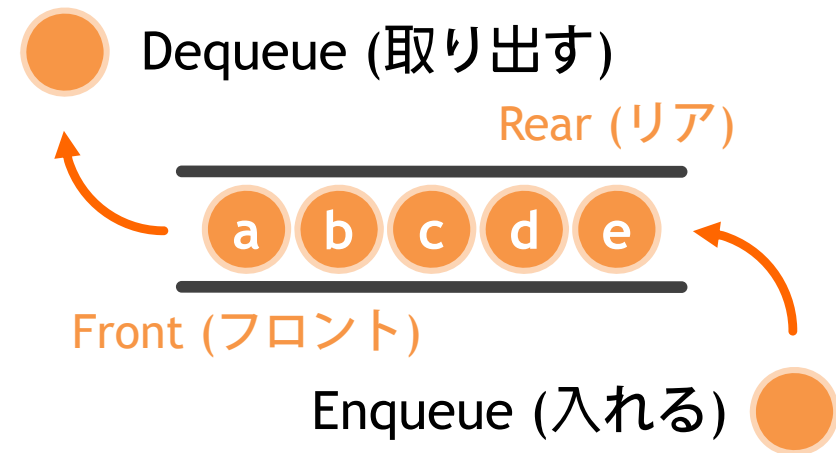
- データを一時的に蓄積する際のデータ構造

スタック (Stack)



LIFO: Last In First Out
(最後に入ったものが最初に出る)

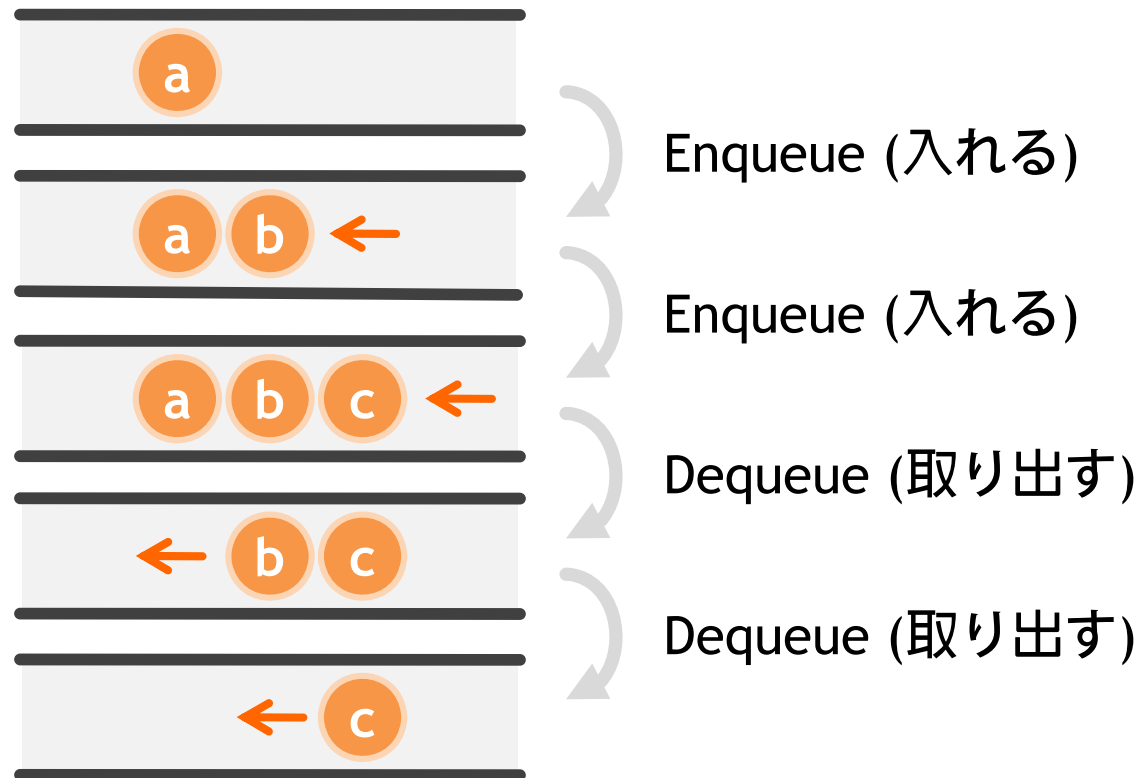
キュー (Queue)



FIFO: First In First Out
(最初に入ったものが最初に出る)

キュー

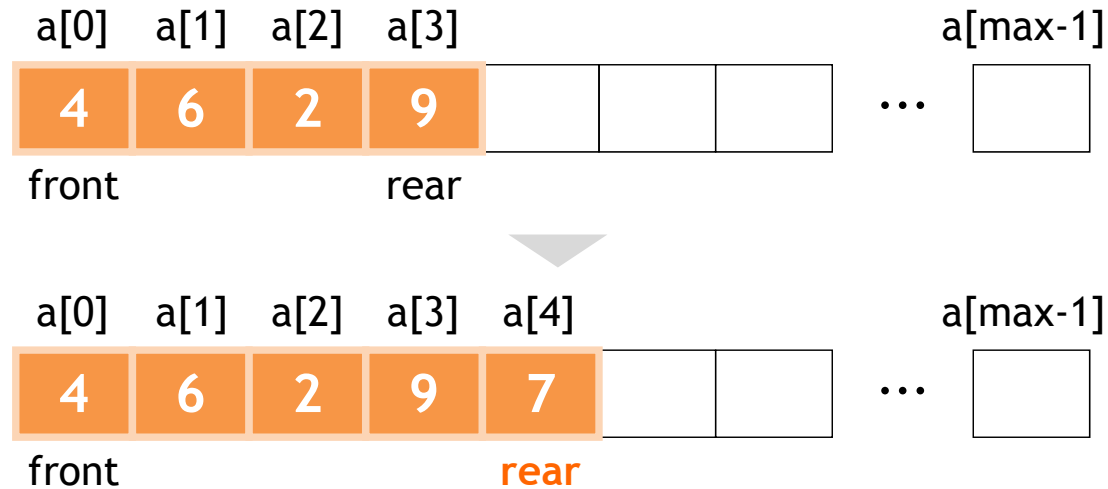
- データを入ってきた順に並べる, 待ち行列 (スーパーのレジと同じ)
- FIFO: **F**irst **I**n **F**irst **O**ut (最初に入ったものが最初に出る)



配列によるキューの実現

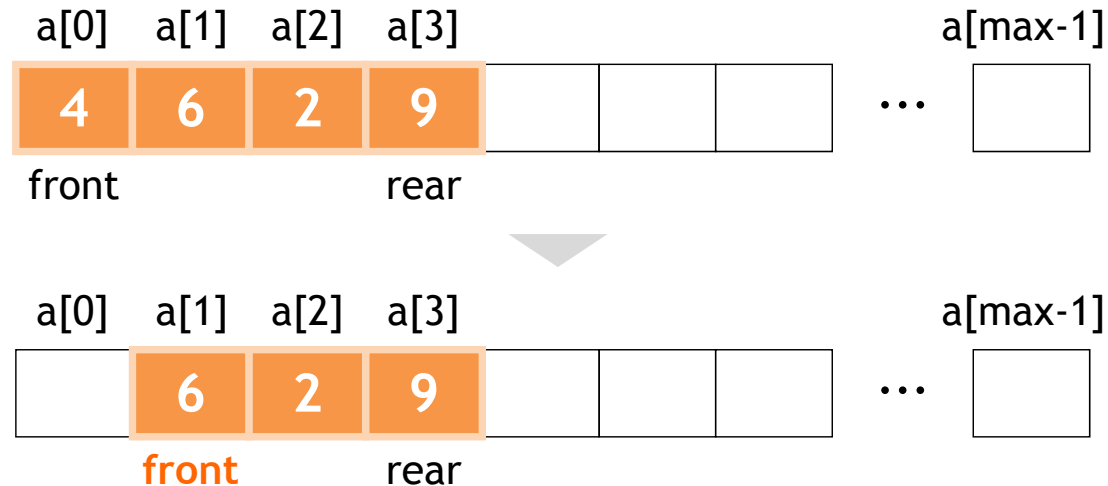
Enqueue
(入れる)

7 →



Dequeue
(取り出す)

← 4

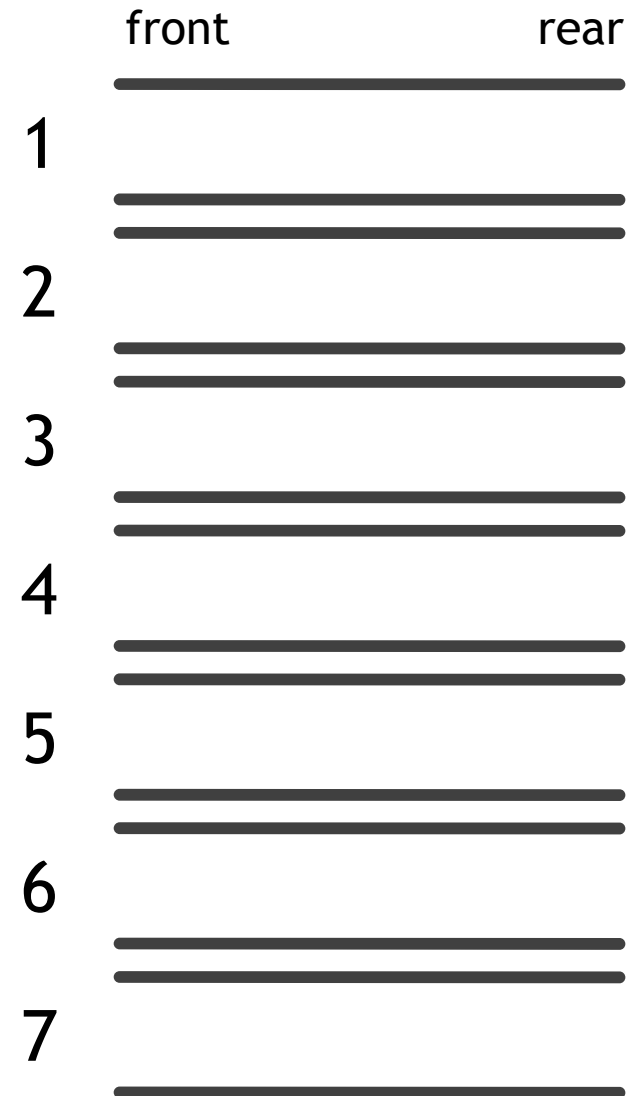


問題2

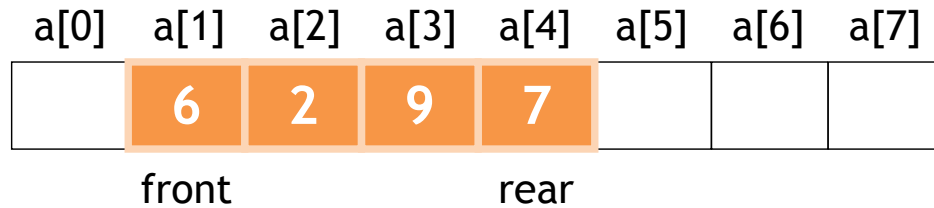
1. Enq(A)
2. Enq(B)
3. Enq(C)
4. Deq()
5. Deq()
6. Enq(D)
7. $x = \text{Deq}()$

$x =$

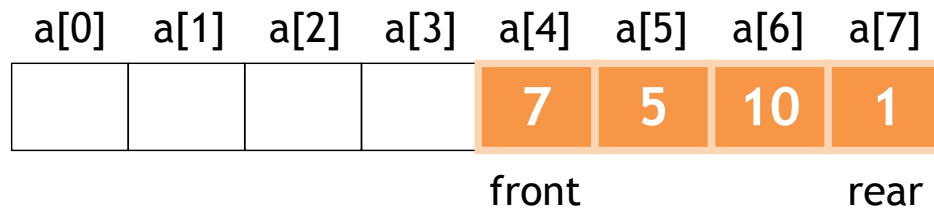
配列サイズは5とする



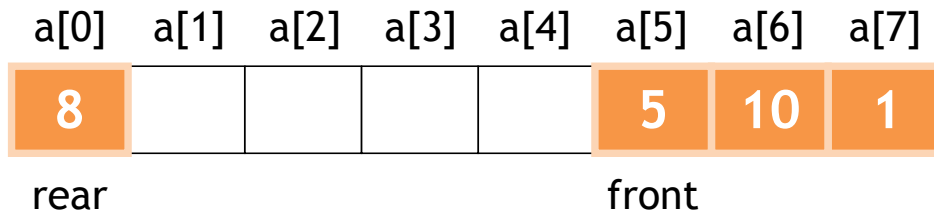
リングバッファの概念(循環配列)



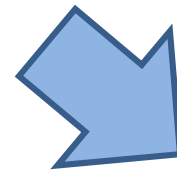
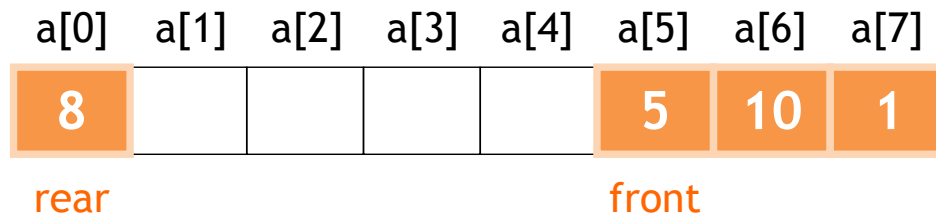
EnqueueとDequeueを繰り返すと、
データが配列の末尾に到達し、
それ以上データが格納できない



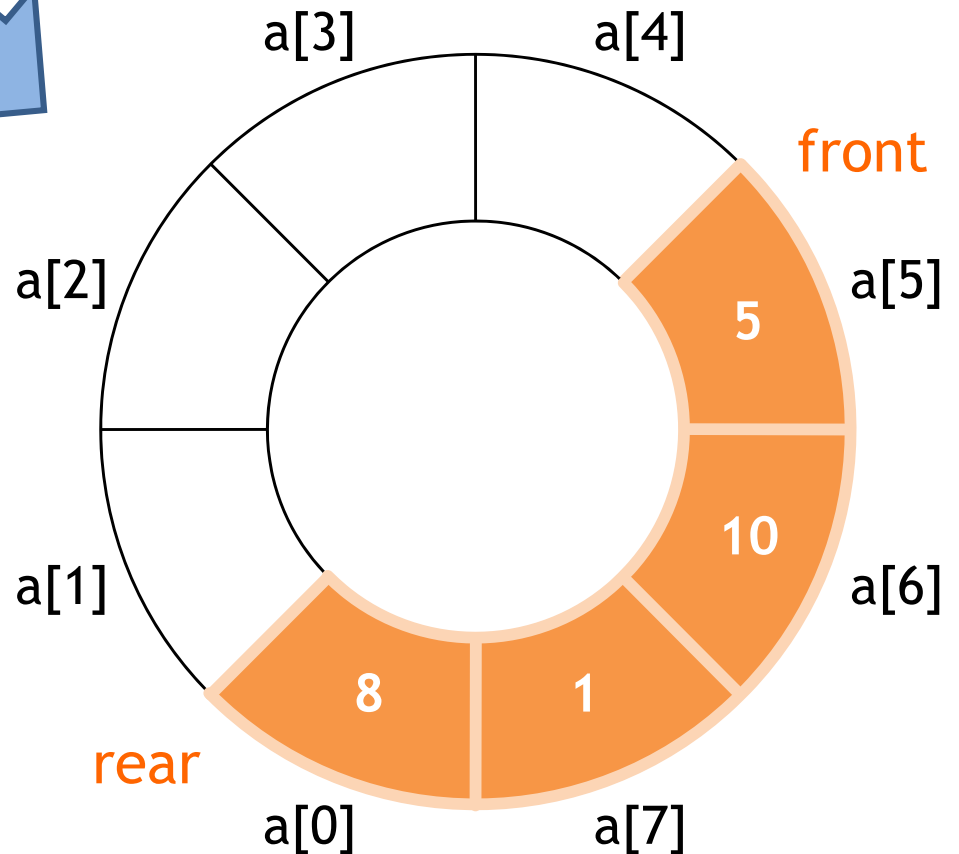
使用していない前半の配列を使用
配列の最初と最後が接続されている
ものとみなす



リングバッファの概念(循環配列)



使用法：
配列の先頭と末尾の
番号を保持する！




問題3

配列サイズは5，
リングバッファとする

1. Enq(A)
2. Enq(B)
3. Enq(C)
4. Enq(D)
5. Enq(E)
6. Deq()
7. Deq()
8. Enq(F)
9. x = Deq()



x = 

	front	rear
1		
2		
3		
4		
5		
6		
7		
8		
9		

キューの配列による実現(変数宣言)

```
#include <stdio.h>
```

```
#define QUEUE_SIZE 5
```

```
#define NO_DATA -1
```

```
int queue[QUEUE_SIZE];
```

```
int front = 0;
```

```
int rear = 0;
```

```
int num = 0;
```

```
void ShowQueue();
```

```
int EnQueue(int data);
```

```
int DeQueue();
```

```
// キューの配列による実現
```

```
// キューの最大データ数
```

```
// 無効データ定数の定義
```

```
// キューの実体の配列
```

```
// キューのフロント
```

```
// キューのリア
```

```
// キューのデータ数
```

```
// 関数のプロトタイプ宣言
```

```
// キュー表示
```

```
// エンキュー
```

```
// デキュー
```

キュー内容の表示関数

```
void ShowQueue()
{
    int i;

    printf("Queue : ");
    for ( i=0; i<QUEUE_SIZE; i++ ){
        if ( queue[i] !=NO_DATA ) {
            printf("[%d]",queue[i]);
        }
        else{
            printf("[  ]");
        }
    }
    printf(" ¥n");
}
```

// キュー内容の表示

// 有効データか？

// 有効データ表示

// 無効データ(空)表示

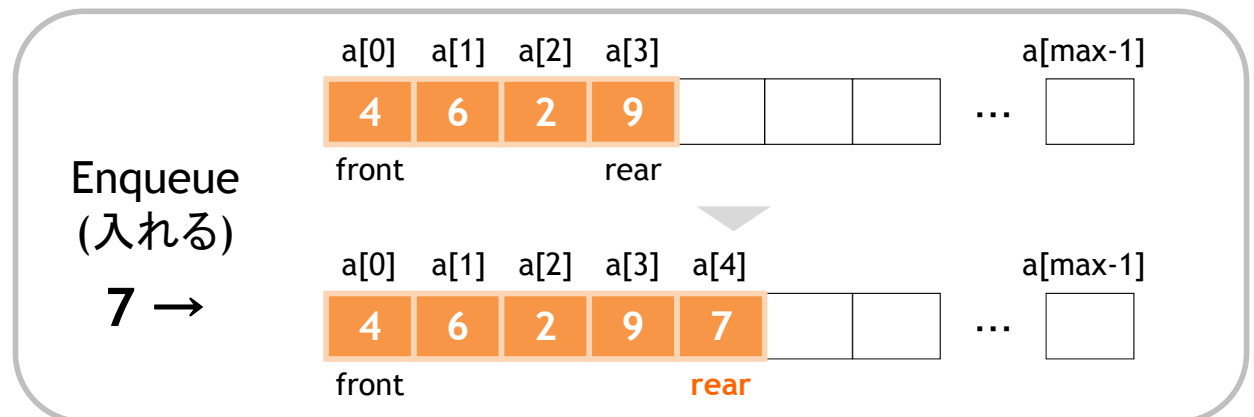
エンキュー関数

```

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
    }
}

```



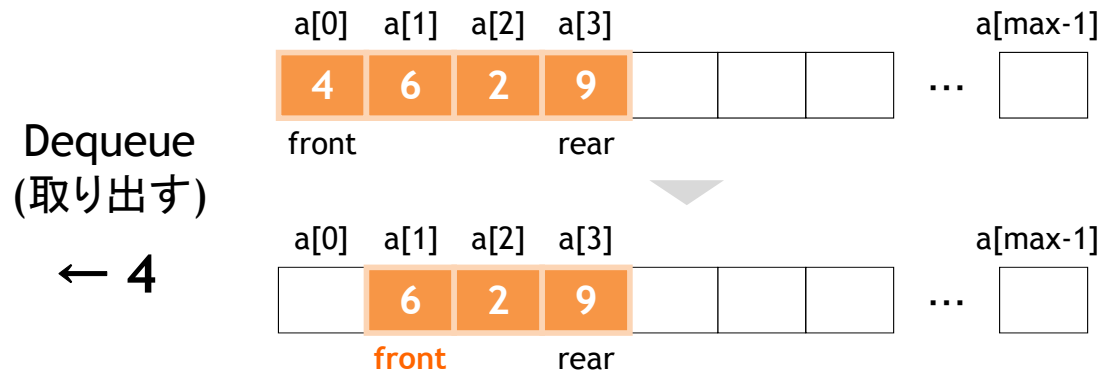
デキュー関数

```

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
    }
}

```



キューの実現(main 関数)

```

void main(){
    int i, mode, data, ret;

    // 初期化
    for( i=0;i<QUEUE_SIZE;i++ )
        queue[i] = NO_DATA;

    // エンキュー&デキュー
    while(1){
        printf("(1)Enqueue (2) Dequeue (0) Exit¥n");
        printf("--> ");
        scanf("%d",&mode);

        if( mode == 0 ){
            break;
        } else if( mode == 1 ){
            printf( "Data: " );
            scanf( "%d",&data );
            ret = EnQueue( data );

```

```

        if( ret == -1 )
            printf("Queue is Full!¥n");
        else
            ShowQueue();
    }
    else if( mode == 2 ){
        ret = DeQueue();
        if( ret == -1 ){
            printf( "Queue is Empty!!¥n");
        }
        else{
            printf( "削除したデータは%d です¥n", ret );
            ShowQueue();
        }
    }
    else{
        printf( "再入力してください¥n" );
    }
}
}

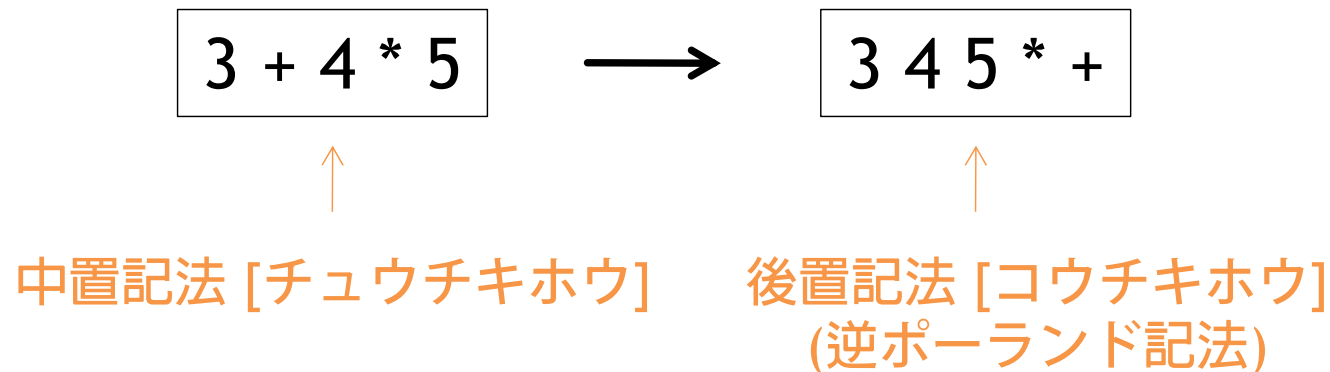
```

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！

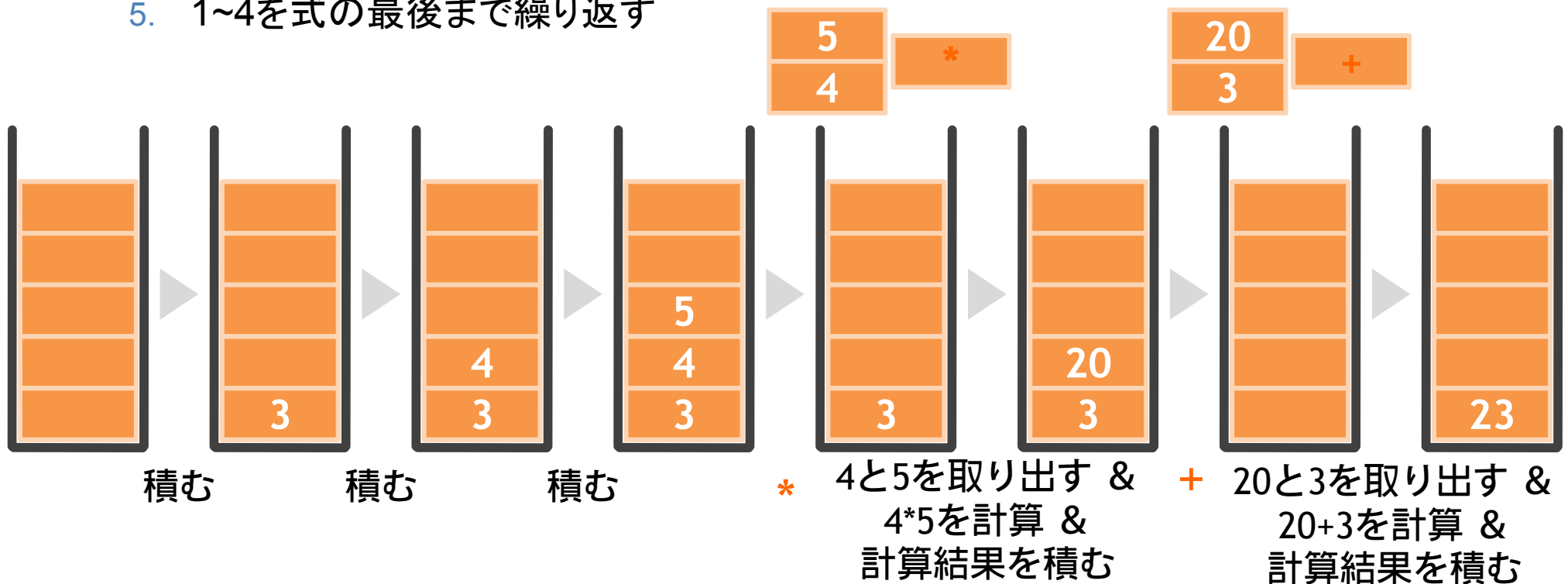


後置記法で書かれた式を計算する

- 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

「3 4 5 * +」の例



問題1

- 次の後置記法式を中置記法で表せ

$$4\ 3\ 2\ -\ + \quad \rightarrow$$

$$6\ 7\ +\ 4\ 2\ -\ * \ 8\ + \quad \rightarrow$$

$$8\ 4\ /\ 3\ * \ 6\ + \quad \rightarrow$$

逆ポーランド記法への変換

$$A + B \rightarrow A B +$$

$$A + B * C \rightarrow A + (B C *) \rightarrow A B C * +$$

$$A * B + C \rightarrow (A B *) + C \rightarrow A B * C +$$

$$\begin{aligned} A * B - C + D &\rightarrow (A B *) - C + D \rightarrow (A B * C -) + D \\ &\rightarrow A B * C - D + \end{aligned}$$

$$(A + B) * C \rightarrow (A B +) * C \rightarrow A B + C *$$

問題2

- 次の中置記法式を後置記法(逆ポーランド記法)で表せ

$$5+3*4 \quad \rightarrow$$

$$(2+3)*(4-1)+5 \quad \rightarrow$$

$$(3+4)*(5-6)+7-1*2 \quad \rightarrow$$