データ構造とアルゴリズム
第4週

掛下　哲郎
kake@is.saga-u.ac.jp

前回のまとめ

- 逐次探索
  - 平均、最悪ともに O(n)
  - 未登録データを探索した場合，全要素の確認が必要
- 順序関係を利用した逐次探索
  - 平均、最悪ともに O(n)
- m-ブロック法
  - 平均、最悪ともにO($\sqrt{n}$)
- 二分探索
  - 平均、最悪ともにO(log n)
- ハッシュ法
  - 平均O(1)。最悪時はO(n)

未登録データを探索した場合でも，全要素の確認は不要

講義スケジュール

週	講義計画
1ー2	導入
3	探索問題
4ー5	基本的なデータ構造
6	動的探索問題とデータ構造
7	アルゴリズム演習（第1回）
8ー9	データの整列
10ー11	グラフアルゴリズム
12	文字列照合のアルゴリズム
13	アルゴリズム演習（第2回）
14	アルゴリズムの設計手法
15	計算困難な問題への対応

今日学ぶこと

データ構造（Data Structure）とは

- 計算機へのデータの蓄え方
- データ構造とアルゴリズムの依存性

基本的なデータ構造

- 配列
- 連結リスト
- キュー
- スタック

データの「挿入」と「取り出し」に特徴のあるデータ構造

- ヒープ
- 2分探索木

要書き取り

配列（Array）

- 配列 s [n]
  - 連続したメモリ番地にデータを保持
  - 保持しているデータ数nを変数に保持
  - 添字を指定したデータアクセス　→ O(1)
  - データ探索
    - 逐次探索を使用　→ O(n)
    - 二分探索を使用　→ O(log n)
  - データの「挿入」と「削除」の手間は？

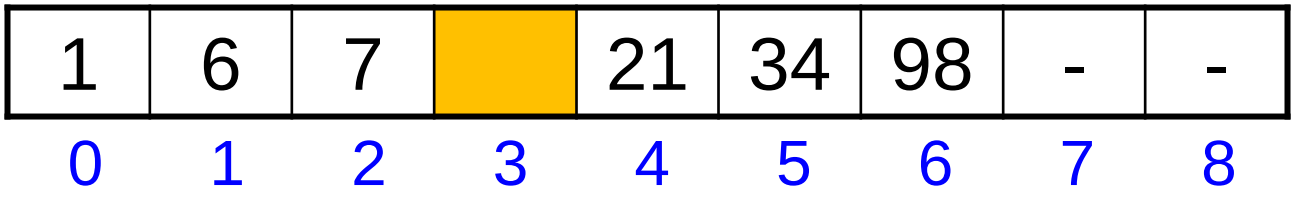
配列要素を並べ替えなくてもOK

配列要素の並べ替えが必要

21	1	6	98	13	34	7	-	-
0	1	2	3	4	5	6	7	8

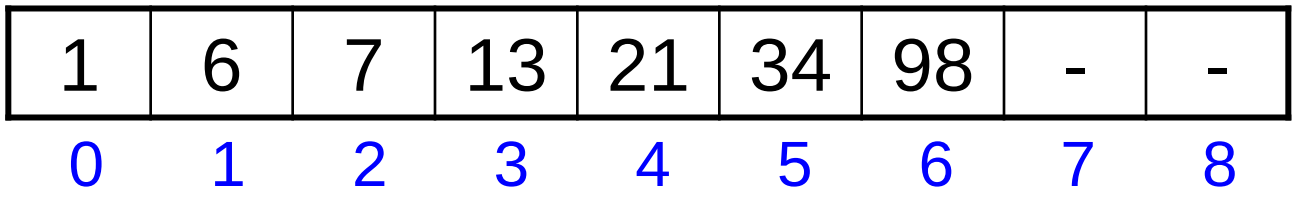
配列へのデータ挿入と削除

- データ削除に伴い空きマスが発生すると面倒.
  - 検索時や挿入時に空きマスを区別する必要がある
  - 二分探索が使えない

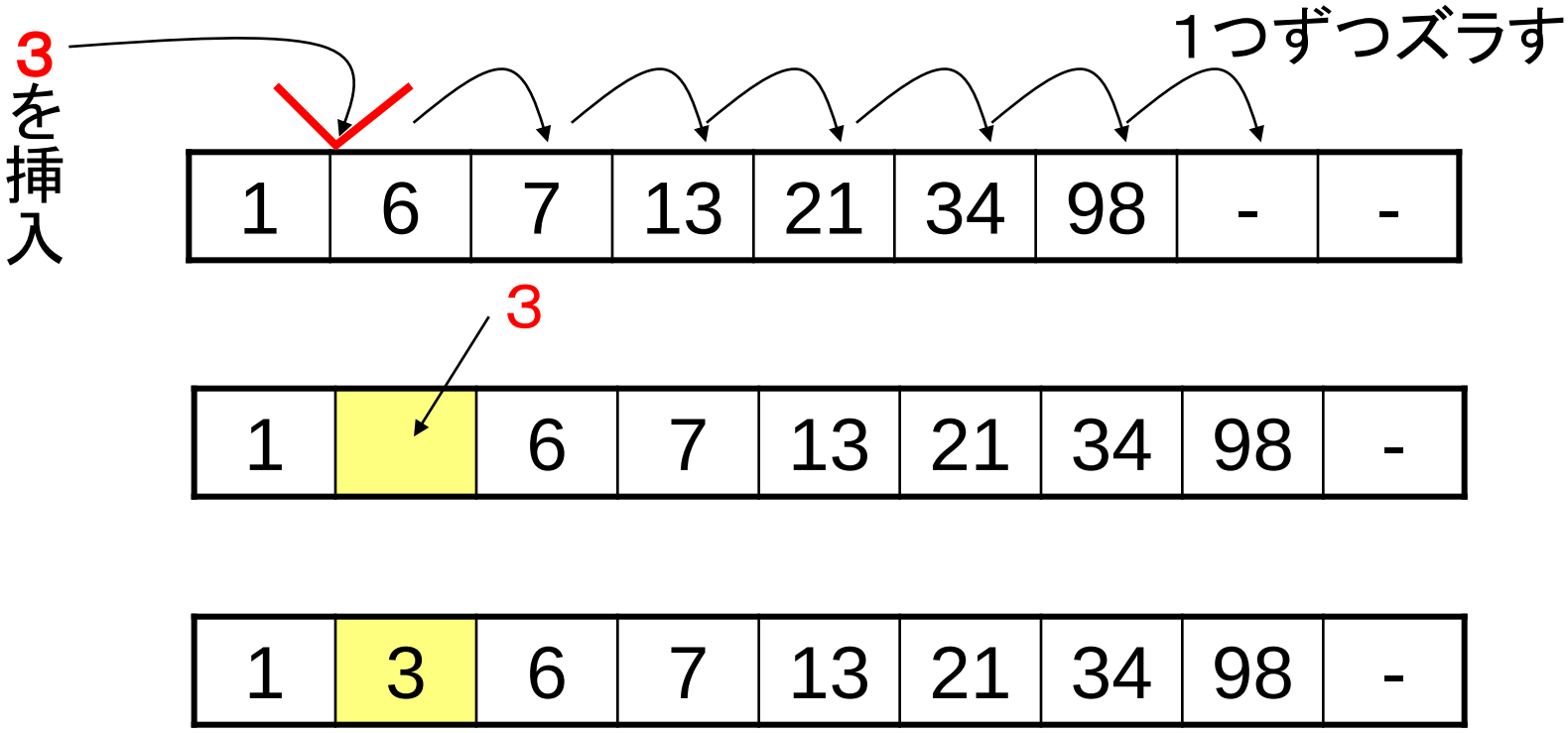


- データの大小順を維持しようとすると面倒.

（大小順）

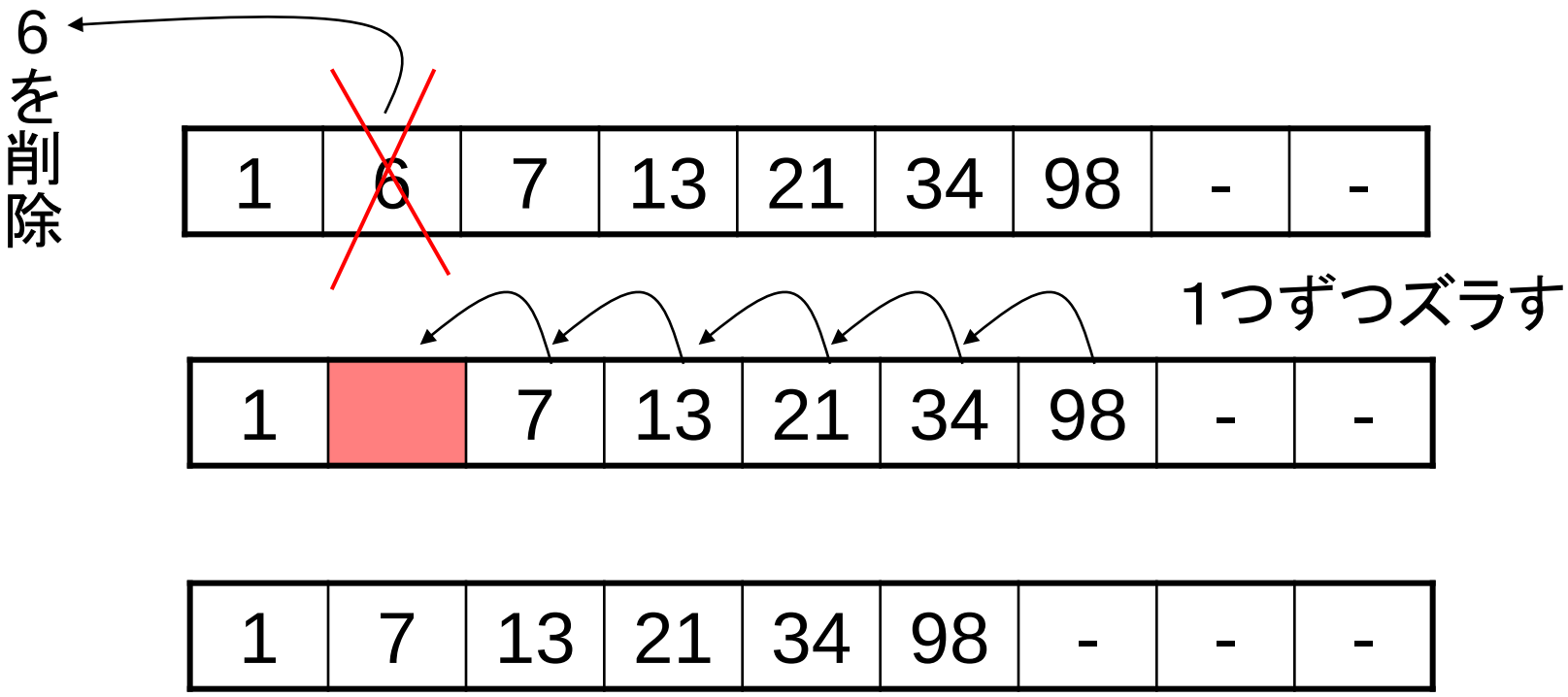


配列への挿入



先頭に挿入する場合は全てズラす → 最悪 O(n)

配列からの削除



先頭から削除する場合も全てズラす → 最悪 O(n)

要書き取り

配列に関する整理

- 長所:
  - 実装が容易
  - データアクセスは O(1).
  - 二分探索を用いたデータ検索はO(log n) .
- 短所
  - 空きマスをなくすために，挿入・削除とも最悪O(n).
  - データの順序関係を維持しつつ，詰め合わせを行うために，アルゴリズム上の工夫が必要
  - 最初に定義した大きさ以上にデータを格納できない

遅い

何か工夫できないか？

- 配列のイメージ
  - ノートに、先頭から順番にきっちり詰めて記録

1, 3, 6, 7, 13,
14, 16, 30, 45,
57 . . . . . .
. .

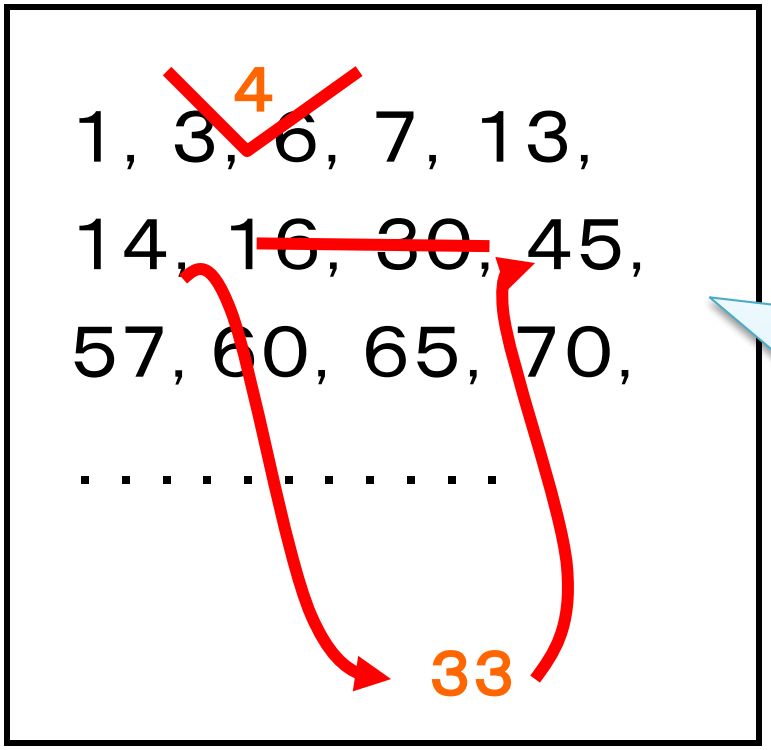
4を挿入

1, 3, 4, 6, 7, 13,
14, 16, 30, 45,
57, ...
.

6以降を、全て
一旦消して書き直し

適当に手を抜く

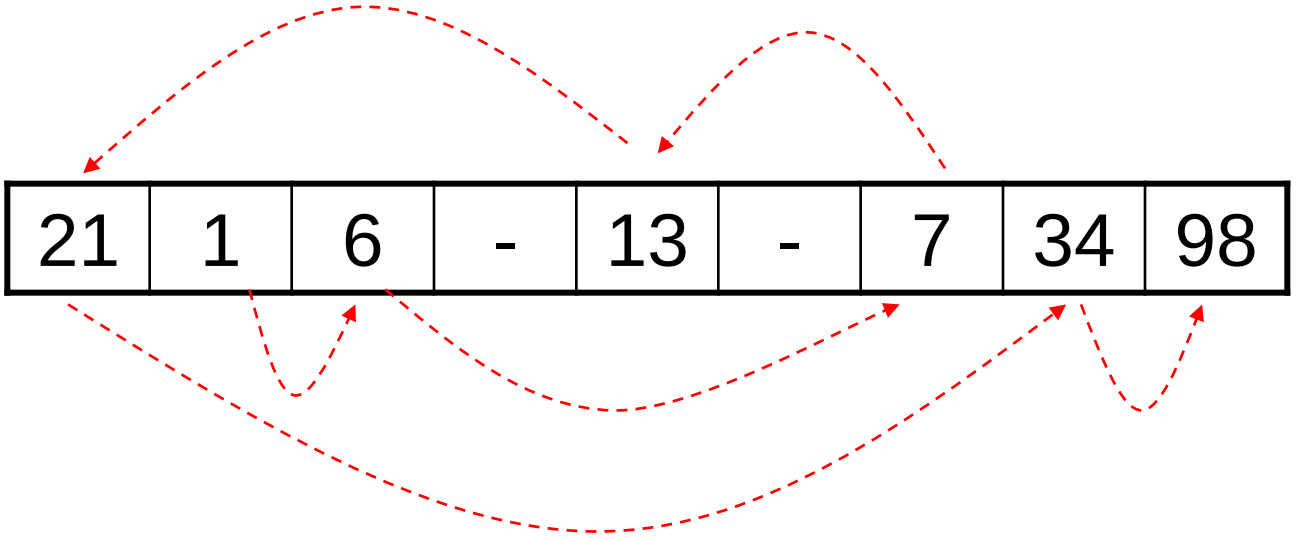
- 普通は、融通をきかせて書き足す。



順番さえ分かれば良い
→
データそのものを律儀に
詰めて並べなくて良い？

順番を明確に定義できれば良い

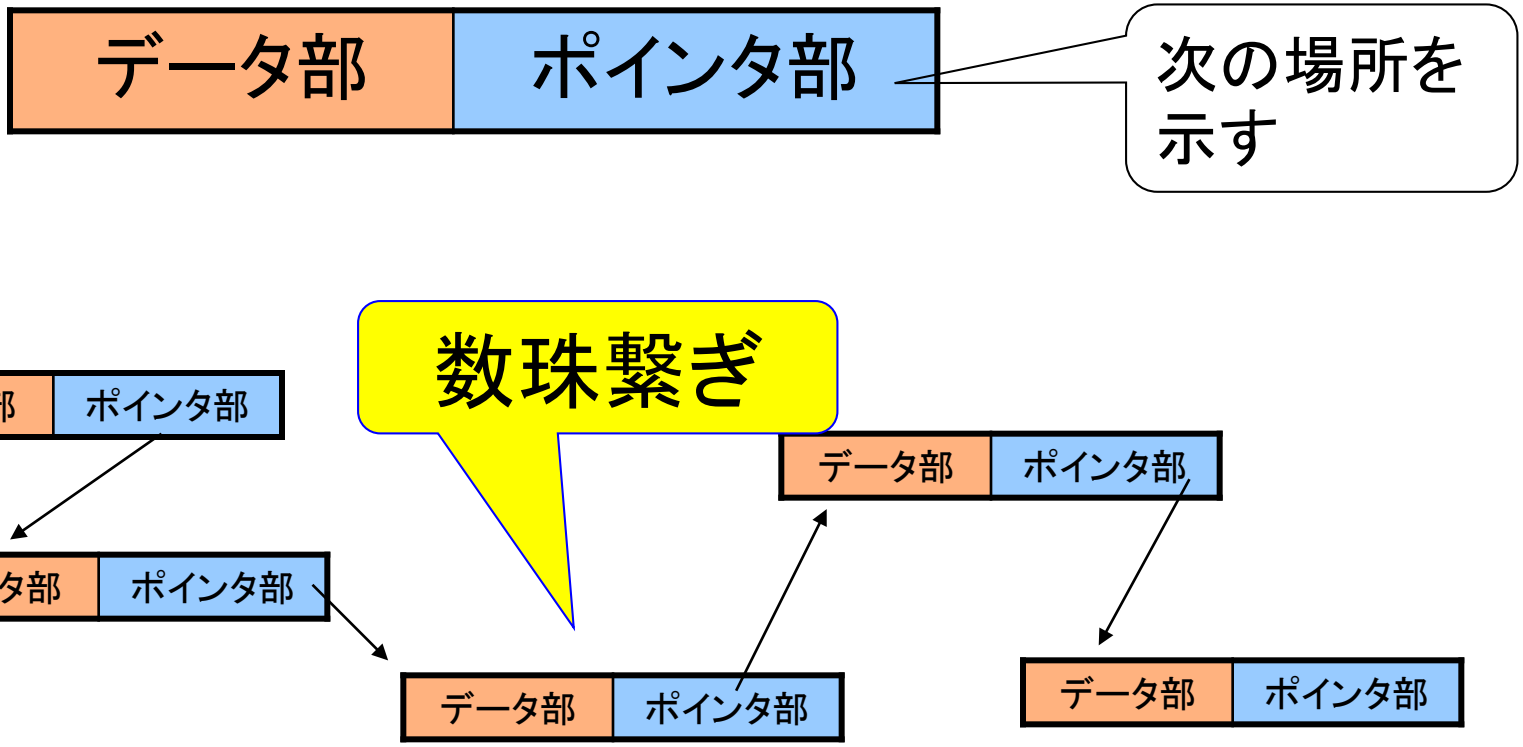
1	6	7	13	21	34	98		-



データ自体はバラバラに格納されていても，順番だけ「矢印」で示しておく

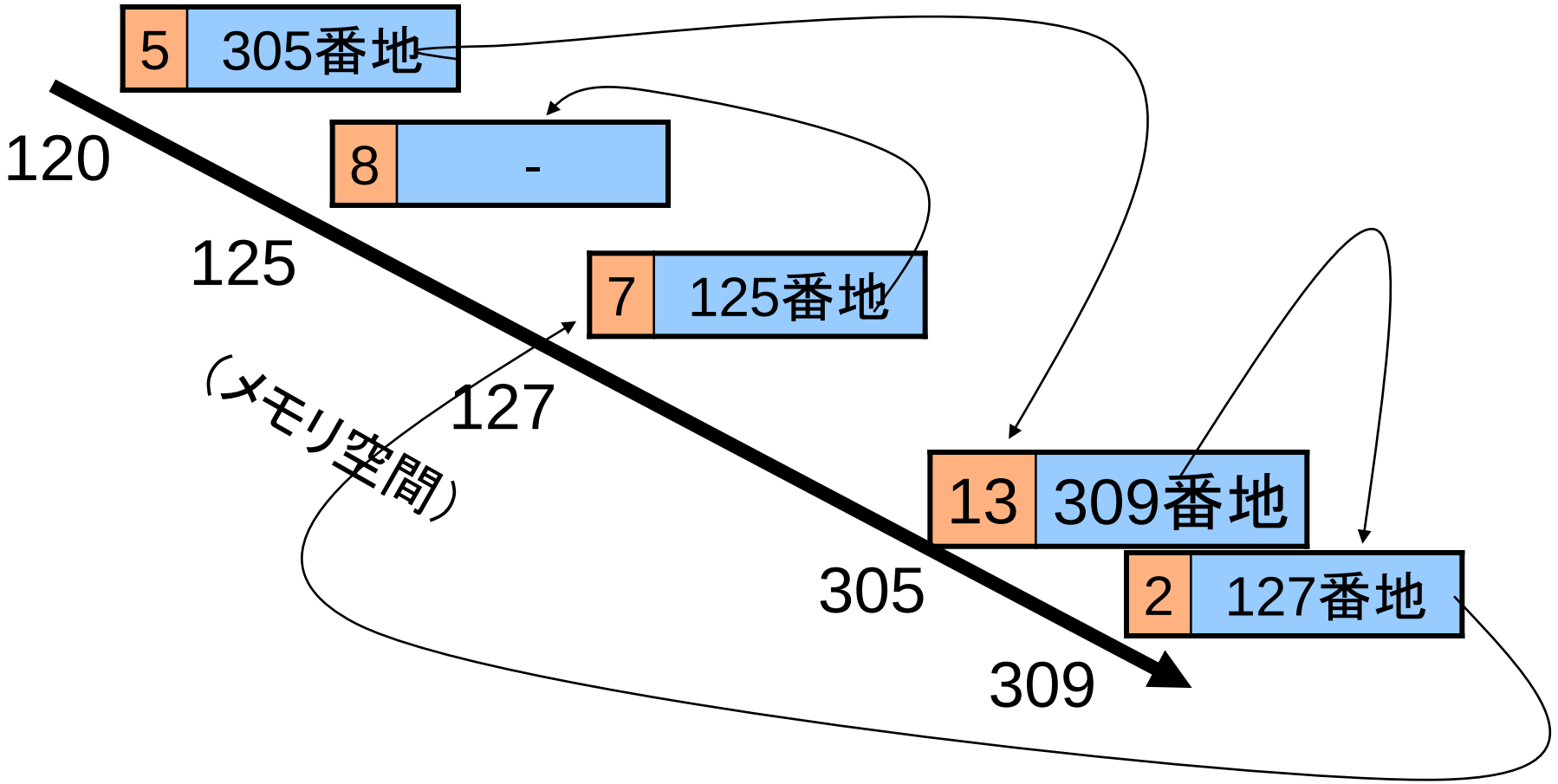
連結リスト (Linked List)

- 「データ」と「次の場所」を，ペアにして記録.



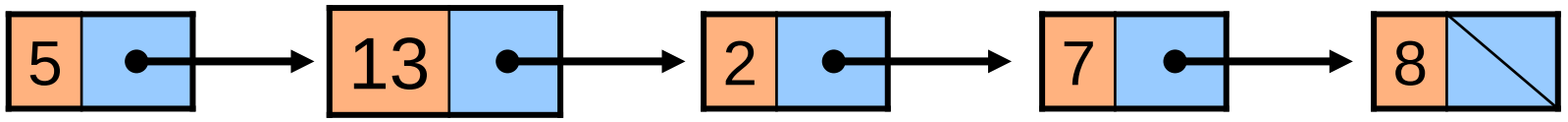
連結リスト

格納データ：5，13，2，7，8



連結リスト

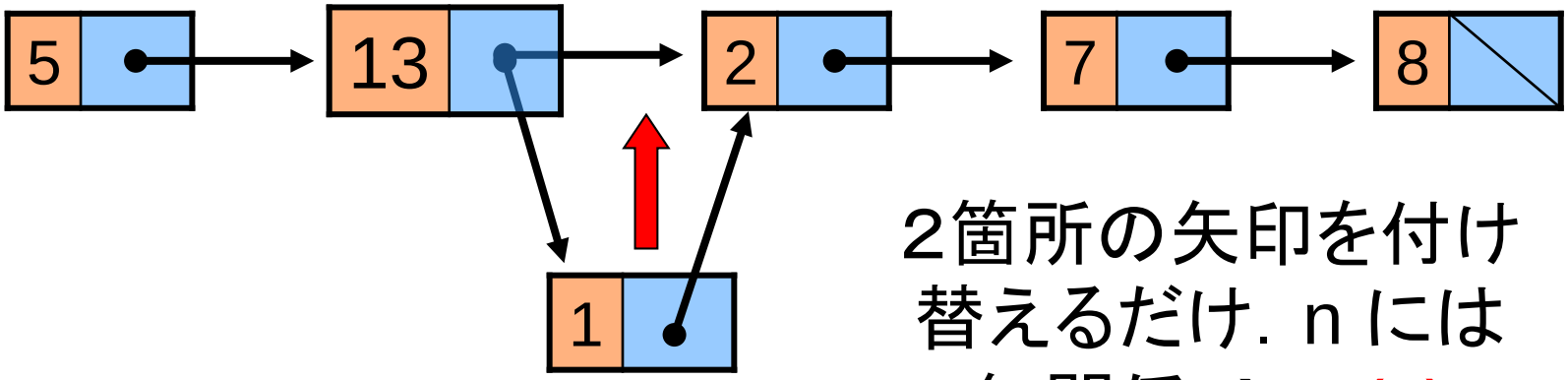
- 簡略化して図示



- 連結リストに対する操作の手間は？
  - データの挿入
  - データの削除
  - データの検索

連結リストへの「挿入」

- 挿入場所が分かっていれば，O(1)で良い



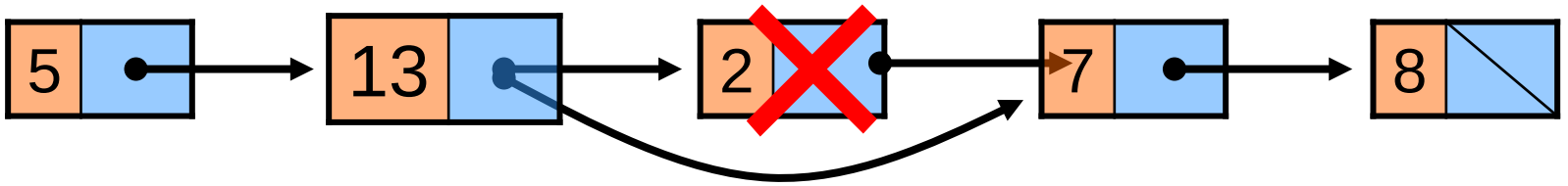
2箇所の矢印を付け替えるだけ．n には無関係 → O(1)

アルゴリズム

1. 新たな要素を挿入する直前の要素を検索する．
2. 新たな要素を生成し，値と次の要素への参照を設定する．
3. 直前の要素が新たな要素を参照するように設定する．

連結リストからの「削除」

- 削除場所が分かっていれば，O(1)で良い



1箇所の矢印を付け替えるだけ．

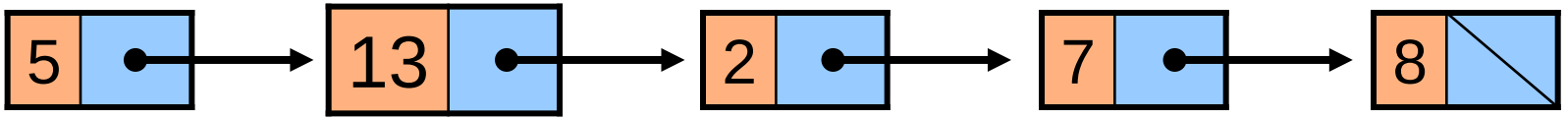
n には無関係 → O(1)

アルゴリズム

1. 削除する要素の直前の要素を検索する．
2. 直前の要素から，削除後の次の要素への参照を設定する．
3. 要素を削除する．

連結リスト内でのデータの「検索」

- k番目の要素を検索する.
- 指定した値xを保持している要素を検索する.
- 計算量は最悪 O(n)



アルゴリズム

1. リスト冒頭の要素から順に，次の要素がある限り以下の処理を繰り返す.
    1-1. 当該要素が指定した値を保持しているならば，その要素への参照を返す.
    1-2. 次の要素を求める.
2. 検索失敗を返す.

整理

配列

長所

- データアクセスは O(1) で高速
- 二分探索が適用可能

短所

- データが大小順に並ぶ場合は，挿入・削除ともO(n)で遅い．
- 最初に定義した大きさ以上にデータを格納できない

連結リスト

長所

- 場所が分かっていれば，挿入・削除がO(1)で高速
- 最初に定義したサイズを超えて，自由にデータを追加できる．

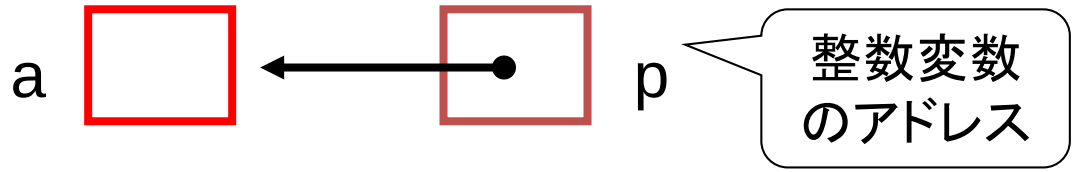
短所

- k番目のデータのアクセスやデータ検索が低速 → O(n)
- 二分探索の適用不可

連結リストのC/C++による表現

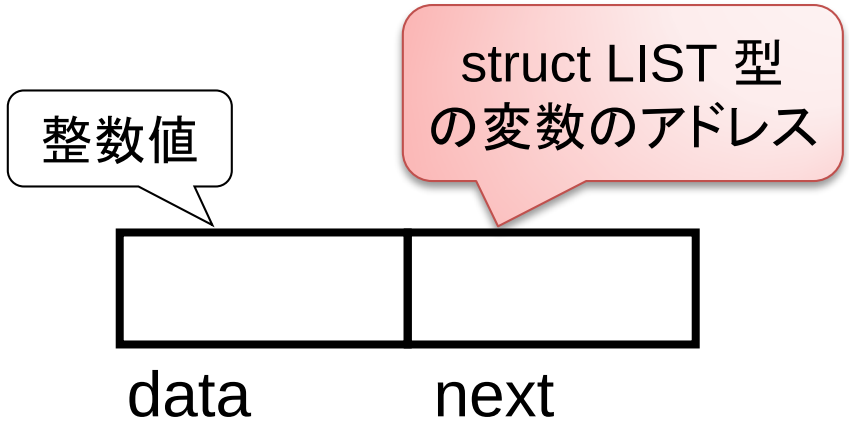
ポインタ

```
int a;      // 整数値を保持
int *p;     // 整数変数のアドレスを保持
```



連結リスト要素の宣言

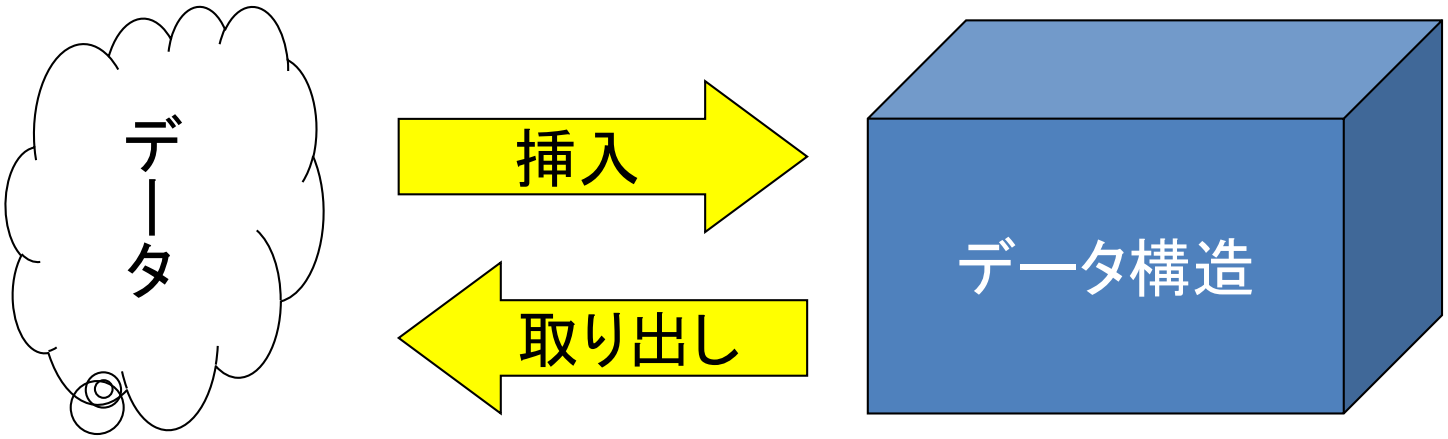
```
struct LIST {
    int data;
    struct LIST *next;
}
```



構造体：複数の変数を組み合わせて新しい型をつくる

データの「挿入」と「取り出し」に着目

- 2つの操作に着目
  - 挿入：データを新しく記録
  - 取り出し：データを読み出し，削除

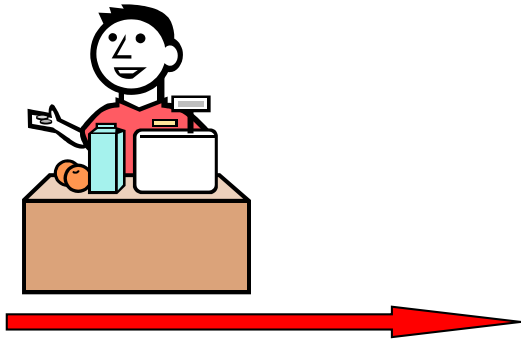


「取り出し」の順番

- 挿入された順に取り出す
  - 早い者順に，取り出される

キュー
（列）

レジでの行列



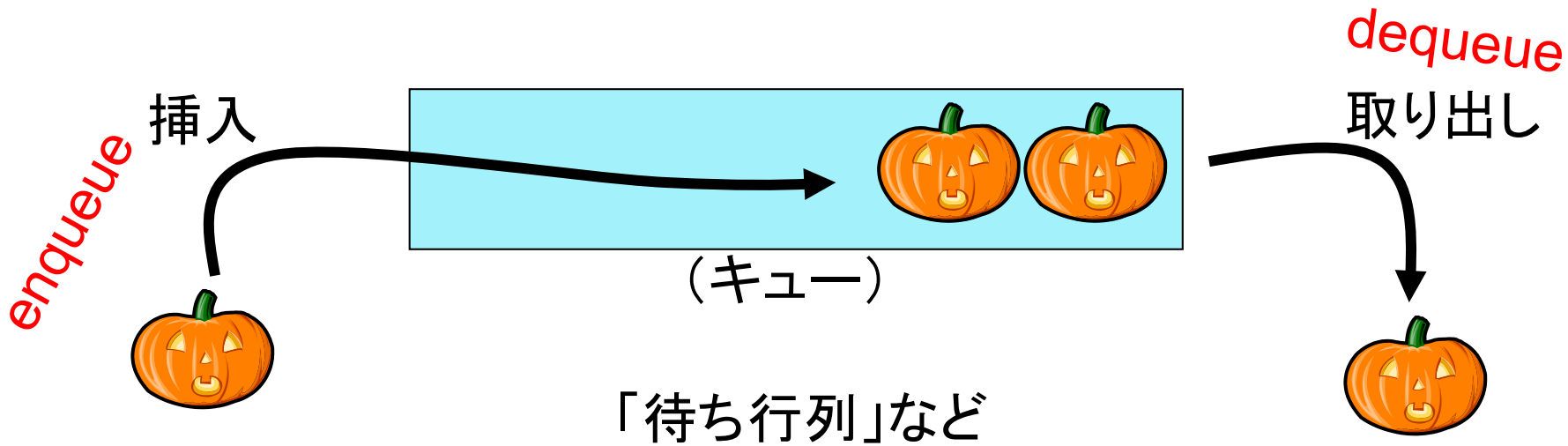
- 最後に挿入されたものから取り出す
  - 遅い者順に，取り出される

スタック
（積む）

山積みの書類

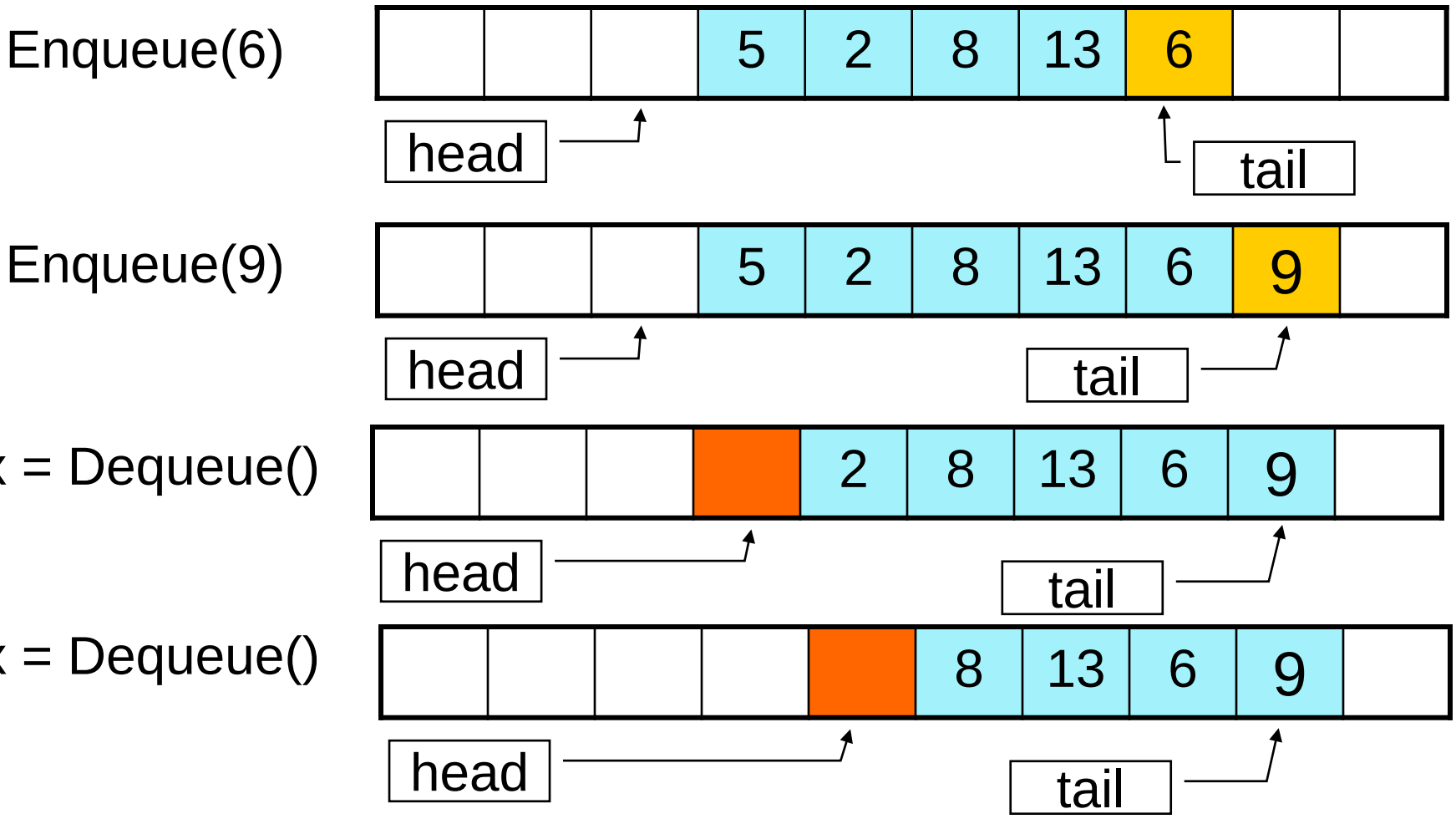
キュー(queue)

- 挿入された順に取り出す

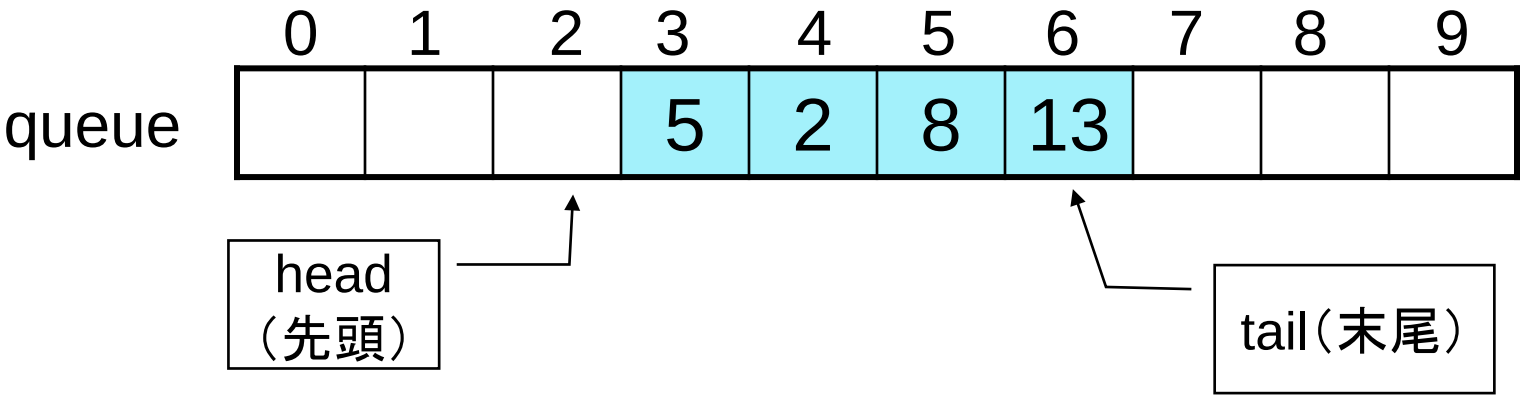


- FIFO (First In First Out)（読み：ファイフォ/フィフォ）
  - 先入れ先出し

Enqueue, Dequeueの例



配列による表現(キュー)



キューへの挿入:Enqueue

1. キューが一杯ならばエラーを返す.
2. 末尾を1つ進め, 新たな要素を保存する.
3. 成功を返す.

キューからの取り出し:Dequeue

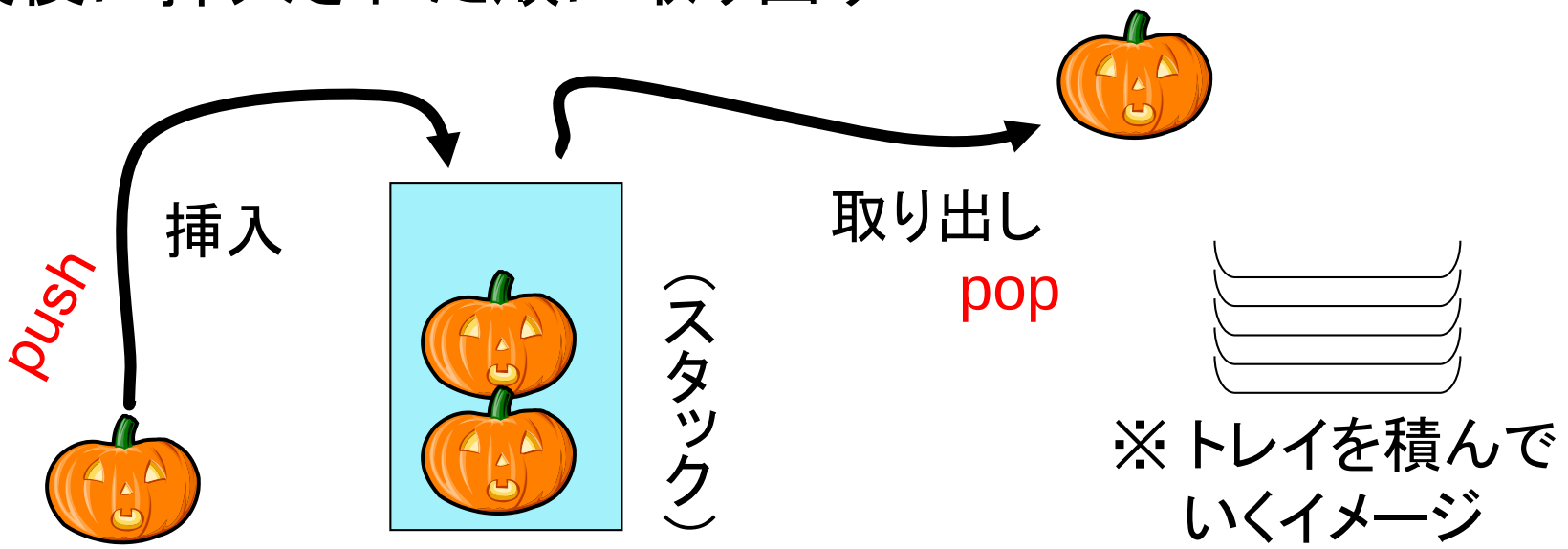
1. キューが空ならばエラーを返す.
2. 先頭を1つ進め, その場所の要素を返す.

先頭を1つ進める.
→ (先頭+1)% 配列サイズ
※ 末尾を進めるのも同様

キューが一杯か?
キューが空か?
→ 先頭と末尾が一致するか?

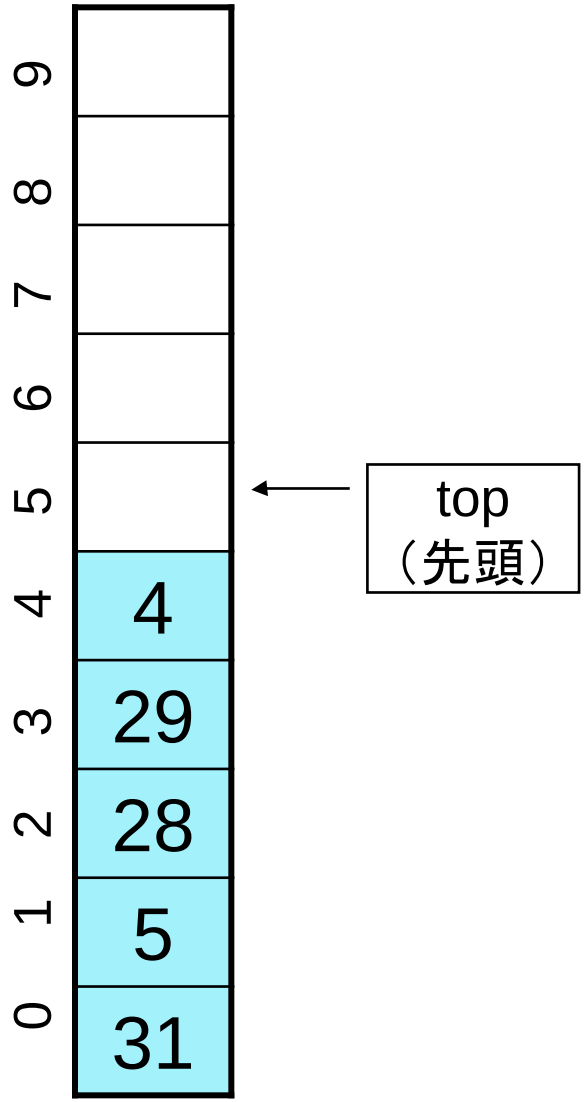
スタック(stack)

- 最後に挿入された順に取り出す



- LIFO (Last In First Out)（読み：ライフォ）
  - 後入れ先出し

配列による表現(スタック)



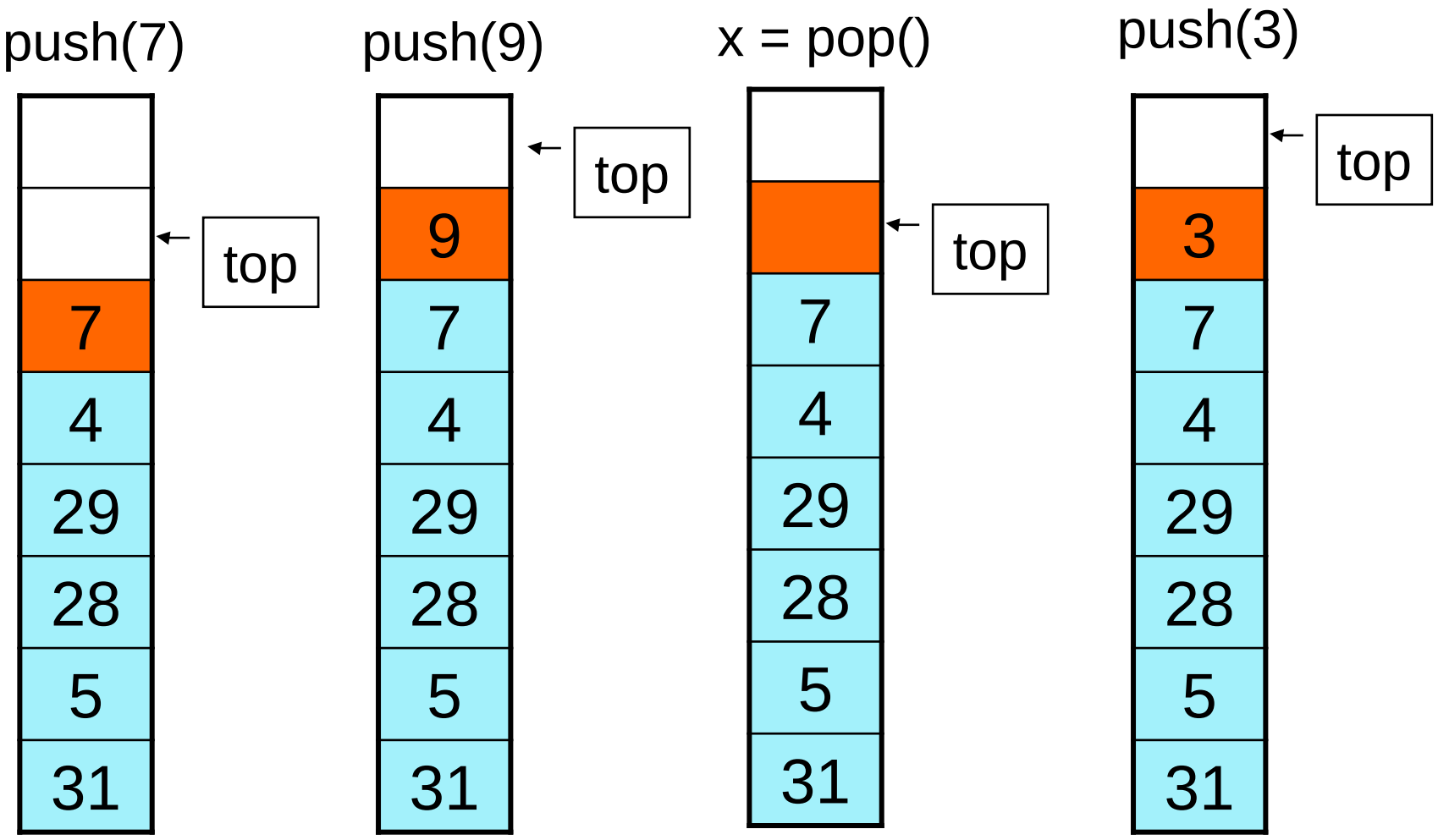
スタックへの挿入push

1. スタックが一杯ならばエラーを返す.
2. 先頭に要素を追加する.
3. 先頭を上に1つ移動する.
4. 成功を返す.

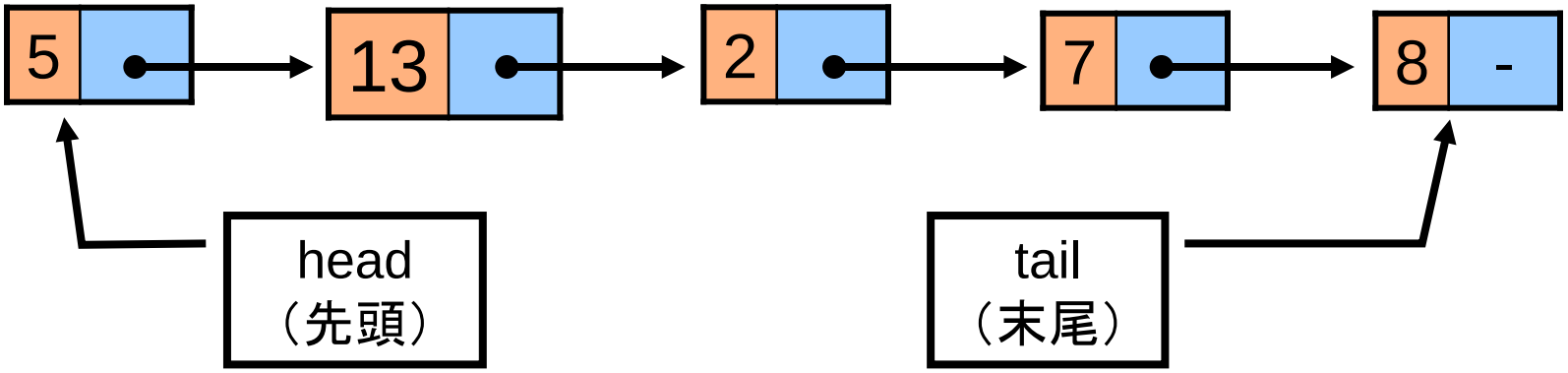
スタックからの取り出しpop

1. スタックが空ならばエラーを返す.
2. 先頭を下に1つ移動する.
3. 先頭の要素を返す.

push/pop の例


push(7)
top
7
4
29
28
5
31
push(9)
top
9
7
4
29
28
5
31
x = pop()
top
7
4
29
28
5
31
push(3)
top
3
7
4
29
28
5
31

連結リストによる表現（キュー）

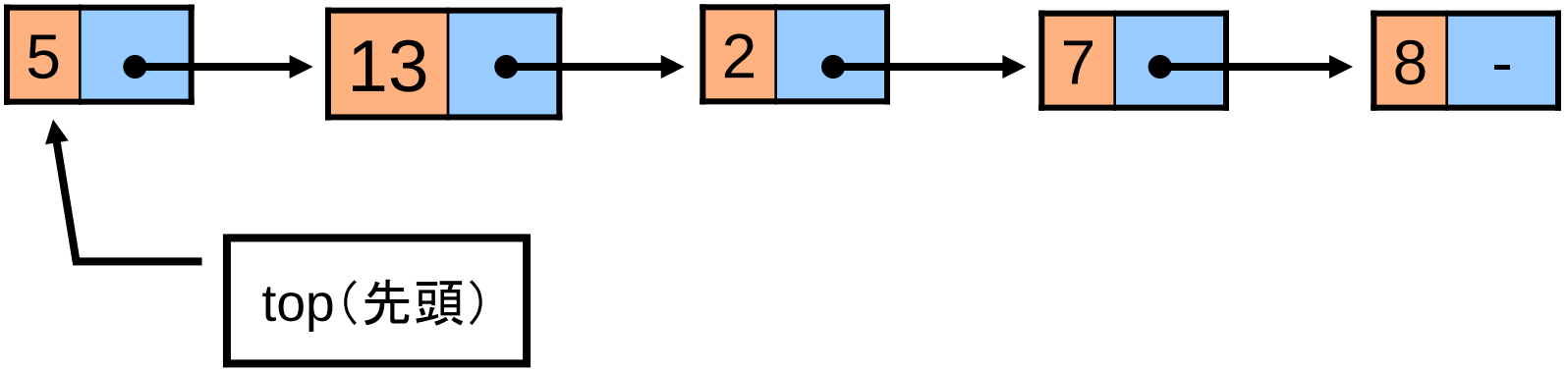


Enqueue：連結リスト末尾へのデータ挿入
→ O(1)

ただし，末尾の位置を覚えておく必要がある

Dequeue：連結リスト先頭からのデータ取り出し
→O(1)

連結リストによる表現（スタック）



push：連結リスト先頭へのデータ挿入
→ O(1)

pop：連結リスト先頭からのデータ取り出し
→ O(1)

要書き取り

整理

- キュー (queue)
  - FIFO (先入れ先出し)
  - 挿入・取り出しともにO(1)
- スタック (stack)
  - LIFO (後入れ先出し)
  - 挿入・取り出しともにO(1)

スタックの活用事例

- 括弧の対応検査
- ルーチン呼び出し
- 後置記法の式の計算（逆ポーランド電卓）
- 後置記法の式の生成

括弧の対応検査

対応する括弧の例

開き括弧	閉じ括弧
(	)
{	}
[	]
“	”
‘	’

※括弧は入れ子になることがある。

○
A+B*(C-D)
while((ch = getchar()) != ‘a’)

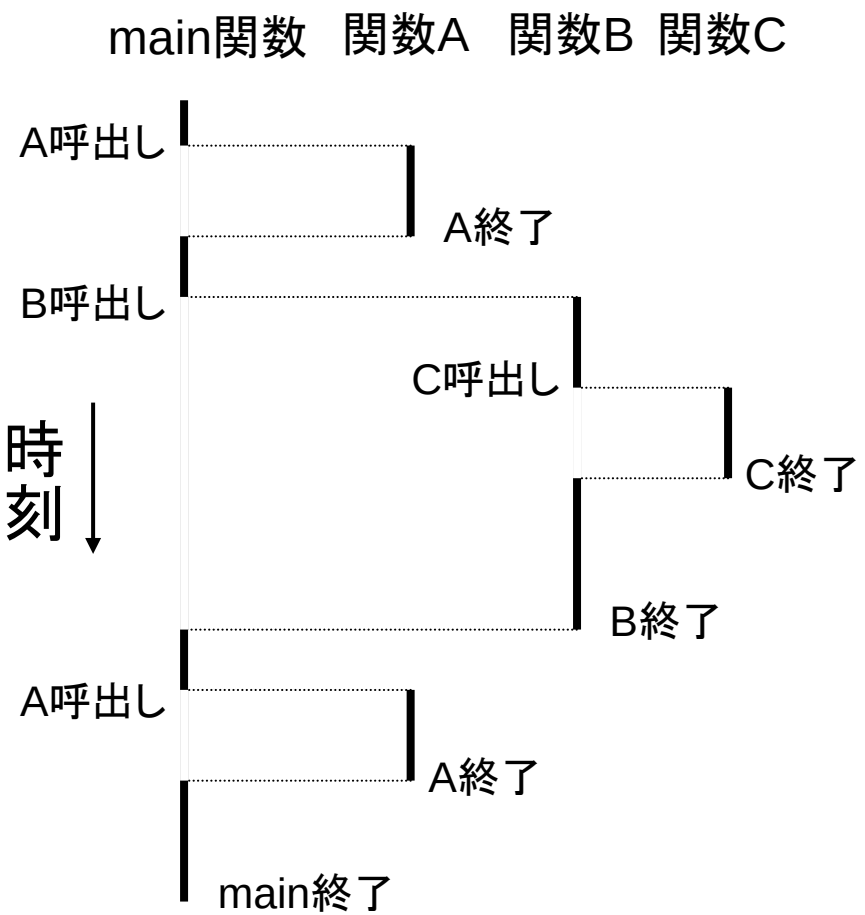
×
A+B*(C-D){
while((ch = getchar()) != ‘a”)

トークン（token）
- プログラム中の個別の単語
- 例：A, +, while, (,), ch, =, getchar, !=, ‘a’

括弧の対応検査アルゴリズム

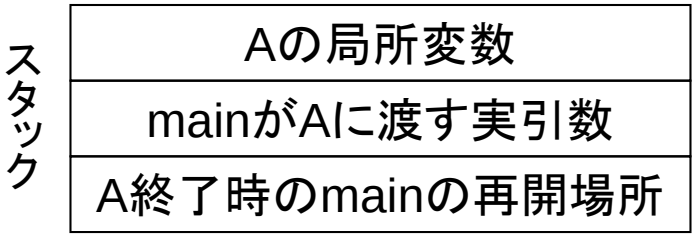
- 最初はスタックを空にする。
- トークン列を左から順に読む。
- 開き括弧を読んだらスタックにpushする。
- 閉じ括弧を読んだらスタックからpopし、対応する開き括弧か確認する。
  - 対応しない開き括弧ならば，対応は取れていない．
- トークン列を読み終わった後でスタックが空ならば、括弧の対応は取れている。
  - そうでなければ，対応は取れていない．

ルーチン呼び出し例（その1）



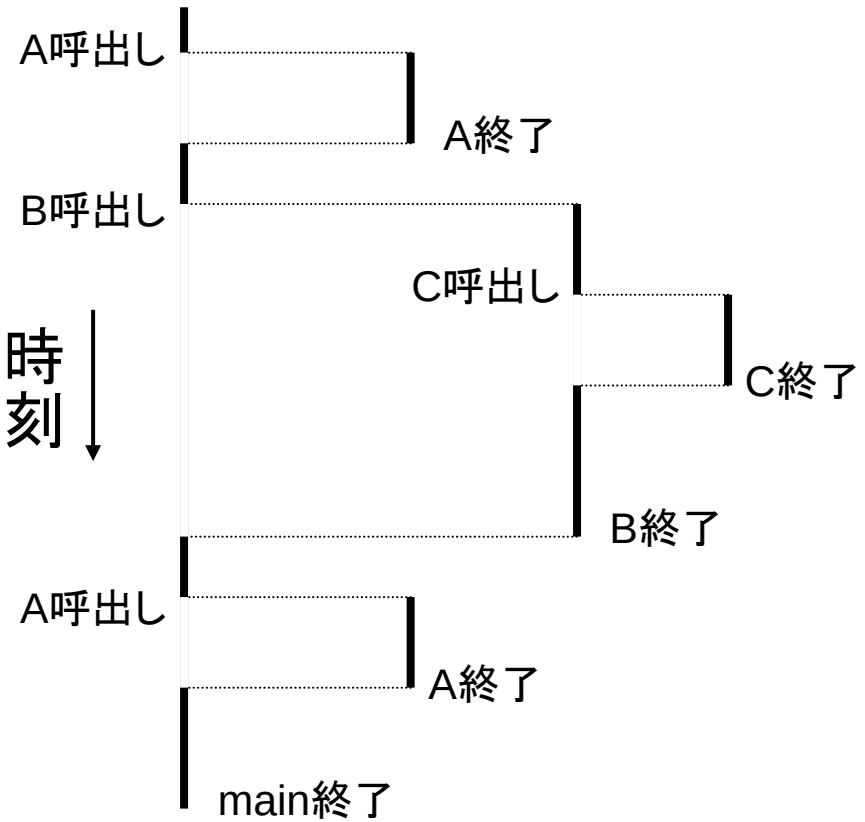
A呼出し

A実行開始



A終了
空スタック

ルーチン呼び出し例（その2）



B呼出し

スタック

mainがBに渡す実引数
B終了時のmainの再開場所

C呼出し

スタック

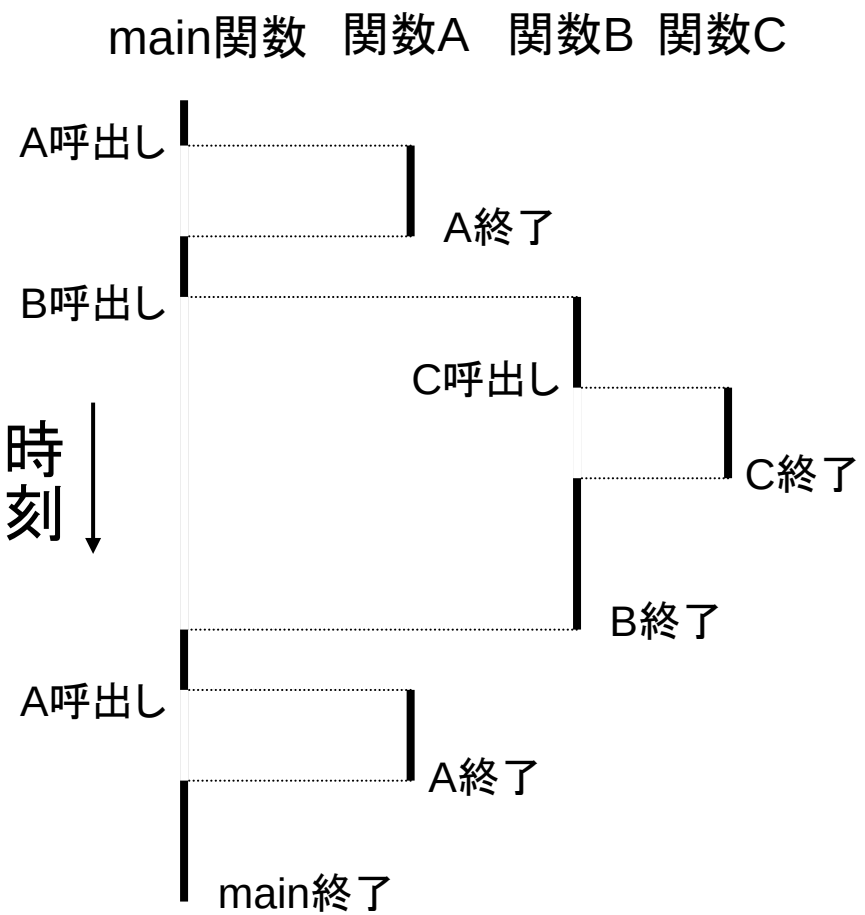
BがCに渡す実引数
C終了時のBの再開場所
Bの局所変数
mainがBに渡す実引数
B終了時のmainの再開場所

C終了

スタック

Bの局所変数
mainがBに渡す実引数
B終了時のmainの再開場所

ルーチン呼び出し例（その3）



B終了
空スタック

A呼出し

スタック

mainがAに渡す実引数
A終了時のmainの再開場所

A終了
空スタック

ルーチン呼出し処理のアルゴリズム

- ルーチン呼出し処理
  - スタックに戻り番地、実引数をpushする。
- ルーチン開始処理
  - スタックにルーチンの局所変数をpushする。
- ルーチン終了処理
  - スタックから局所変数、実引数、戻り番地をpopし、戻り番地から実行を再開する。
- 任意のルーチン呼出しに対応できる。
  - 例：関数AがA自身を呼び出す（再帰呼び出し）

確認テスト(第4回目)

1. 配列でのデータ挿入・削除アルゴリズム
   - 要素をソートしない場合
   - 要素をソートする場合
2. 連結リストでのデータ挿入・削除・検索アルゴリズム
   - 要素をソートしない場合
3. 括弧の対応検査アルゴリズム