講義「アルゴリズムとデータ構造」

第3回　基本的なデータ構造（リスト、スタック、キュー）

大学院情報科学研究科 情報理工学専攻
情報知識ネットワーク研究室
喜田拓也

今日の内容

抽象データ型とは

基本的なデータ構造（抽象データ型）

リスト：　最も基本的なデータ集合の表現

配列／連結リスト／双連結リストによる実装

スタック：　積み上げ式のデータ格納方式

キュー：　入れた順に取り出せるデータ格納方式

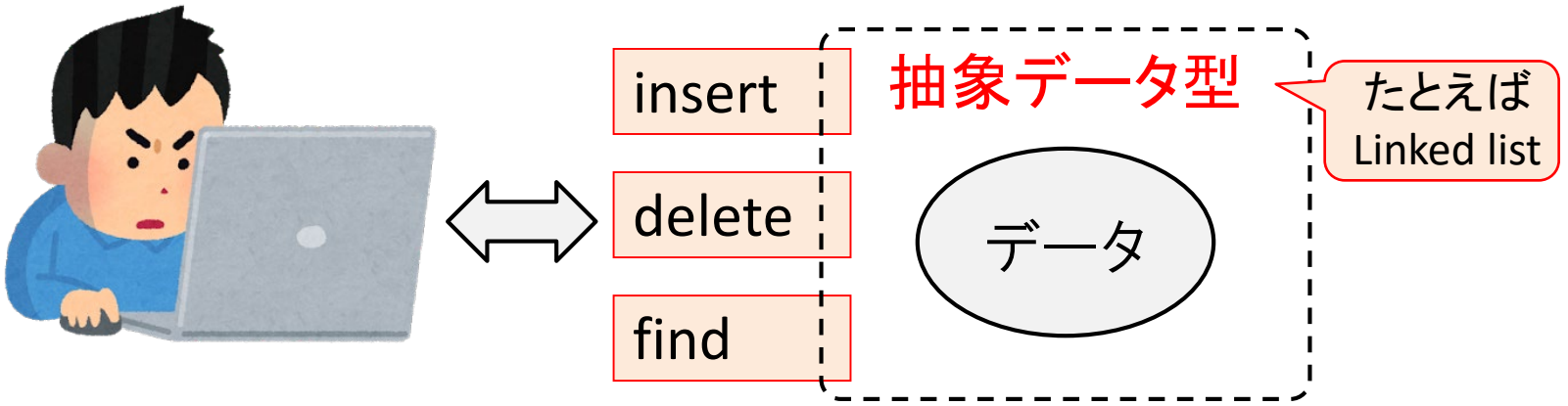
ポイント

抽象データ型とその実装

ポインタを用いたデータ構造

抽象データ型(Abstract Data Type)とは

データ型を，それに適用される一連の操作で抽象的に定めたもの

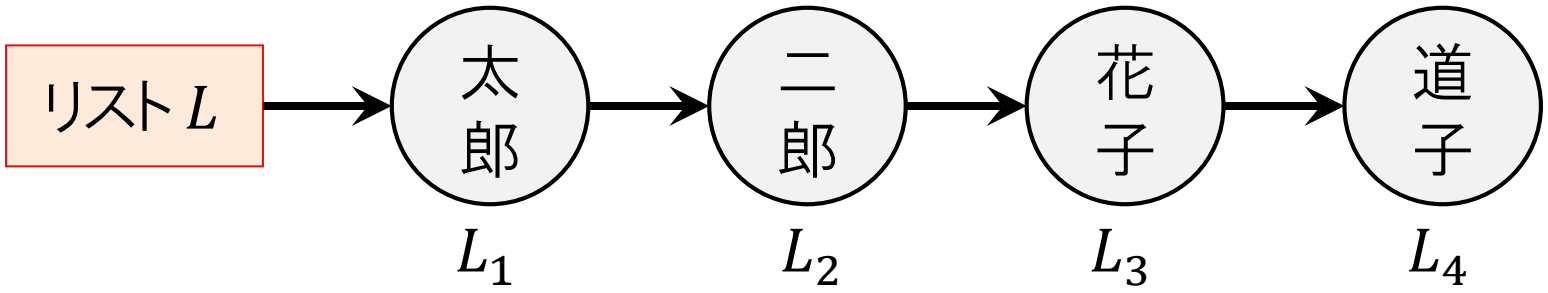


データとその操作を抽象データ型でカプセル化することのメリット

- 定められた正しい方法でのみデータがアクセスされる
- 不正な操作からデータを保護することができる
- ユーザはデータ構造の実現方法について知る必要がない
- プログラマは他の部分とは独立して内部を実現することができる

リストとは？

0個以上の要素を一列にならべたもの

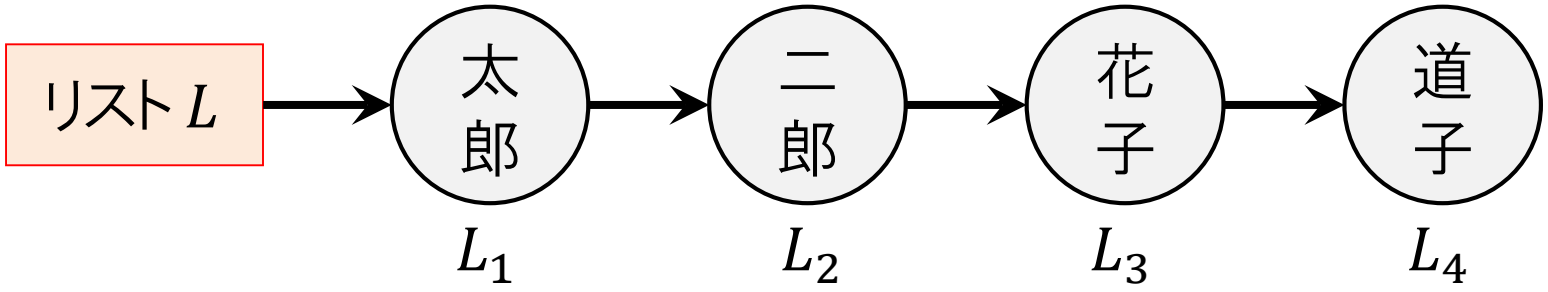


要素 L_i: 最初から i 番目の要素（$1 \leq i \leq n$）
リスト中の場所を指示するための「位置」をもつ

リストの長さ: 要素数n

空リスト: 要素を含まないリストのこと

リストに対する操作



INSERT(x, p, L)： リストL（要素数n）の位置pの次の位置に要素xを挿入

DELETE(p, L)： リストLの位置pの次の要素を削除

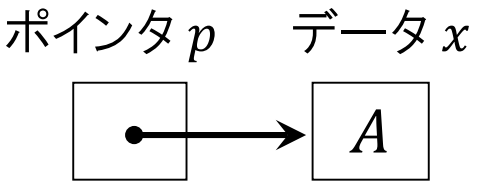
FIND(i, L)： リストLのi番目のセルの内容を返す

LAST(L) ： リストLの最後のセルの位置を返す

PREVIOUS(p, L)： リストLにおいて、位置pの1つ前のセルの位置を返す

ここで「位置p」とはメモリ上の位置を指し示すポインタのこと

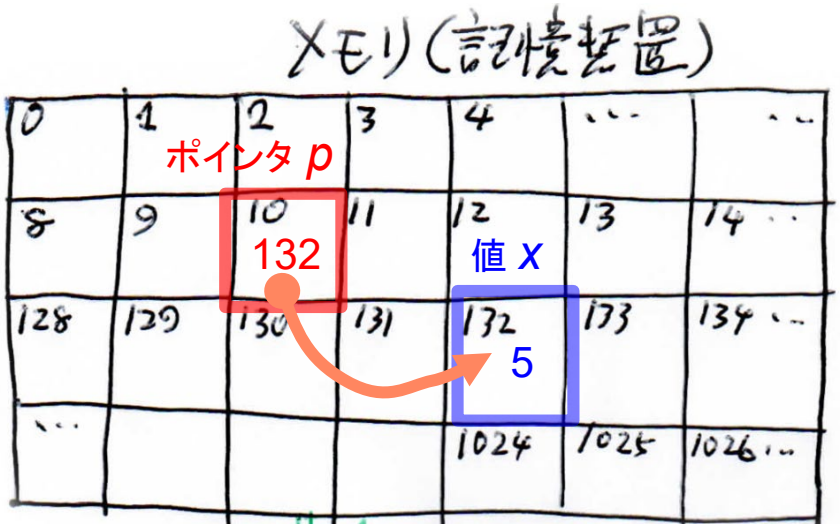
ポインタ(pointer)ってなに？



セルの位置を示すデータ

機械語レベルではセルの番地とセルの型情報の組

プログラミングにおいては，その値を具体的に知る必要はない



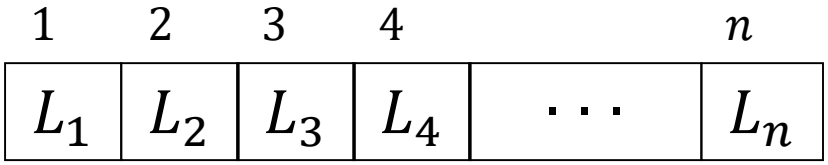
C言語の場合

ポインタpが指す変数の値を，*pで取り出せる

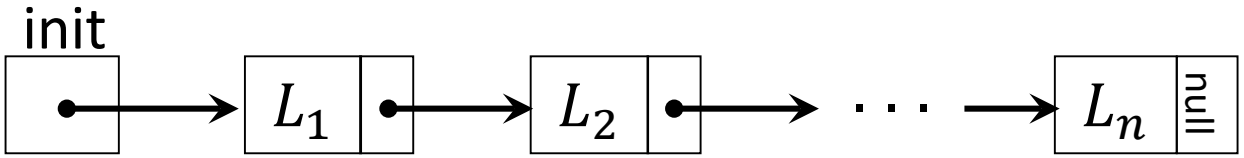
変数xのアドレス(格納場所)は&xで参照できる

リストの実装の方策

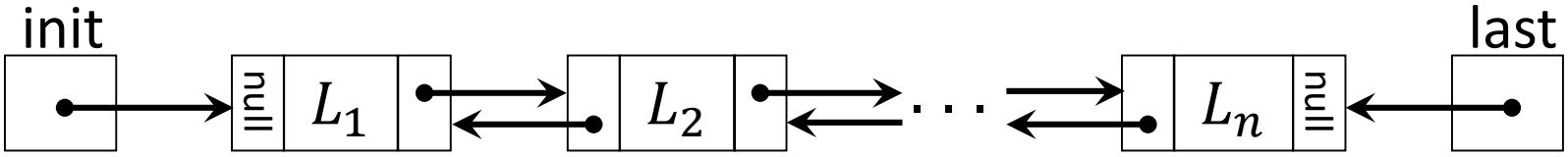
1. 配列(array)：メモリ上のn個の連続領域に格納



2. 連結リスト(linked list)：ポインタで次の要素の格納領域を指す



3. 双方向連結リスト(doubly linked list)：２個のポインタで前後を指す



連結リストによるリストの実装（C言語版）

```
typedef struct cell {
  int element;
  struct cell *next;
} cell;

cell *init=NULL;  //空のリスト

void list_add(int x)
{
  //整数要素xを先頭へ追加
  cell *new=
   (cell *)malloc(sizeof(cell));
  new->element=x;
  new->next=init;
  init=new;
}
```

struct cell {･･･}: 構造体cellを･･･と定義

typedef ･･･ cell: ･･･をデータタイプcell型として定義

initはcell型データを指すポインタ型

sizeof(cell): cell型のデータサイズ（バイト）

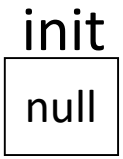
malloc(n):nバイトのメモリーを確保

(cell *)malloc(n): 確保したnバイトの領域をcell型データ格納領域とみなす

連結リストによるリストの実装（C言語版）つづき

```
typedef struct cell {
  int element;
  struct cell *next;
} cell;

cell *init=NULL;  //空のリスト

void list_add(int x)
{
  //整数要素xを先頭へ追加
  cell *new=
   (cell *)malloc(sizeof(cell));
  new->element=x;
  new->next=init;
  init=new;
}
```

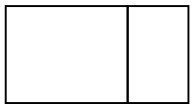
最初に空リストができる



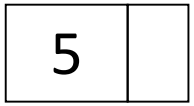
最初は，どこも指していない
ことを示す特別な値「null」
が入っている

list_add(5)を実行すると

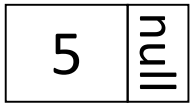
① メモリ上に空のセルが作られる



② そのセルのelement部分に値xが入る

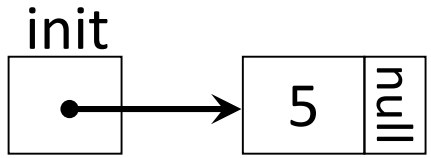


③ ポインタnextにはinitが指す位置が入る



最初にaddされる
要素の場合はnull

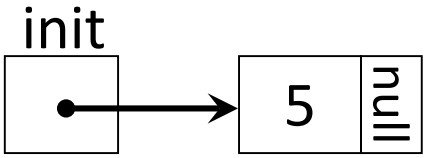
④ initはその新しくできたセルを指す



連結リストによるリストの実装（C言語版）つづき

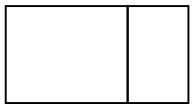
```
typedef struct cell {
  int element;
  struct cell *next;
} cell;

cell *init=NULL;  //空のリスト

void list_add(int x)
{
  //整数要素xを先頭へ追加
  cell *new=
   (cell *)malloc(sizeof(cell));
  new->element=x;
  new->next=init;
  init=new;
}
```

現状のリストの様子

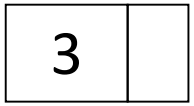


さらにlist_add(3)を実行すると

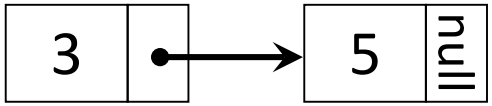
① メモリ上に空のセルが作られる



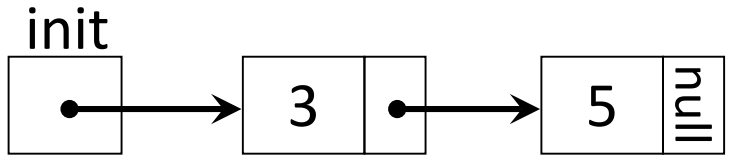
② そのセルのelement部分に値xが入る



③ ポインタnextにはinitが指す位置が入る



④ initはその新しくできたセルを指す



双方向連結リストによるリストの実装(C言語版)

双方向連結リストによる実装

```
typedef struct cell {
  int element;
  struct cell *prev;
  struct cell *next;
} cell;

cell *init=NULL; //空のリスト
cell *final=NULL;

void list_add(int x)
{ //整数要素xを先頭へ追加
  cell *new=(cell *)malloc(sizeof(cell));
  new->element=x;
  new->next=init;
  new->prev=NULL;
  if(init==NULL) final=new;
  else init->prev=new;
  init=new;
}
```

cell型は1つ前のデータを指すポインタprevももつ

最後尾のデータを指すポインタfinalも必要

先頭に追加する場合は1つ前のデータはなし

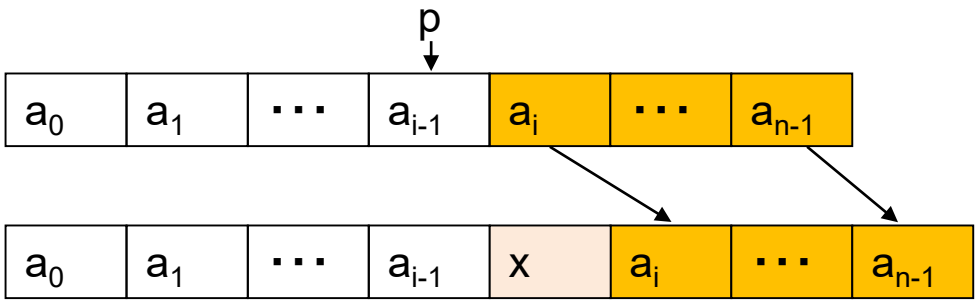
最初に格納されたデータが最後尾のデータ(finalが指すデータ)となる

2つ目以降に格納されたデータはprevポインタを新しい先頭データを指すように更新する

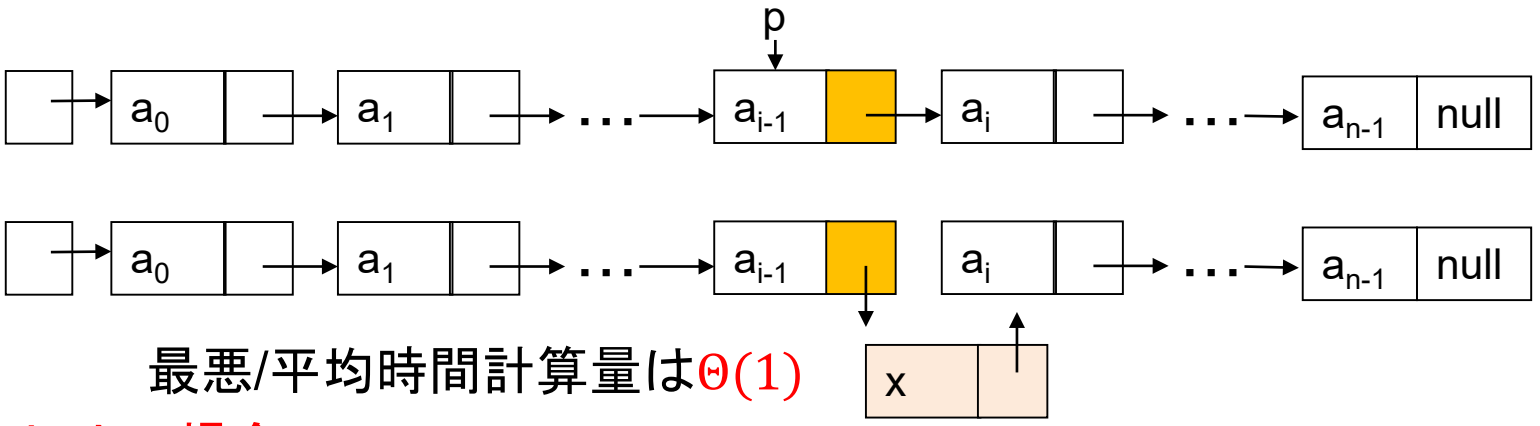
連結リストが得意な処理(挿入の場合)

配列の場合

最悪/平均
時間計算量は$\Theta(n)$

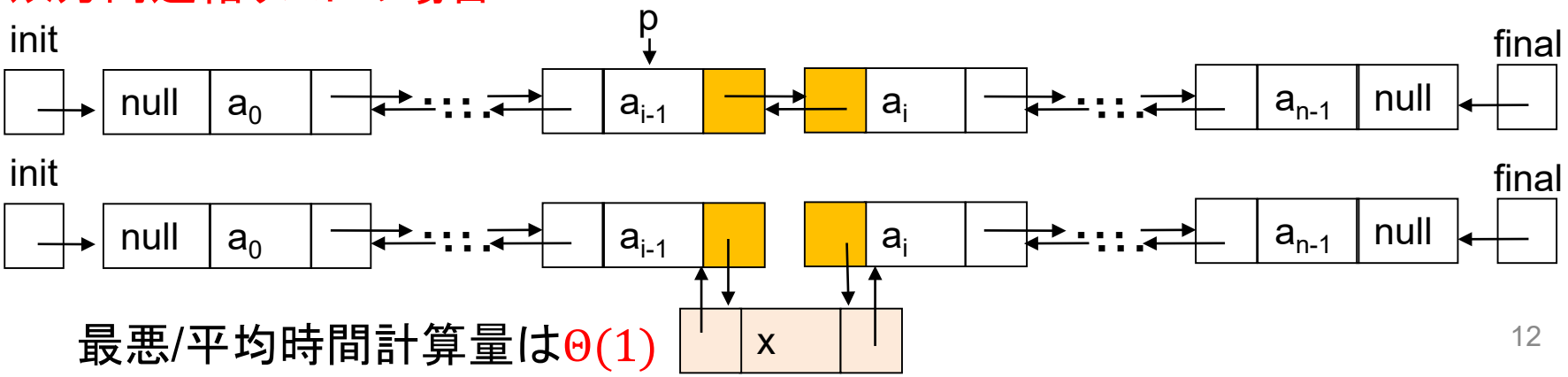


連結リスト
の場合



最悪/平均時間計算量は$\Theta(1)$

双方向連結リストの場合

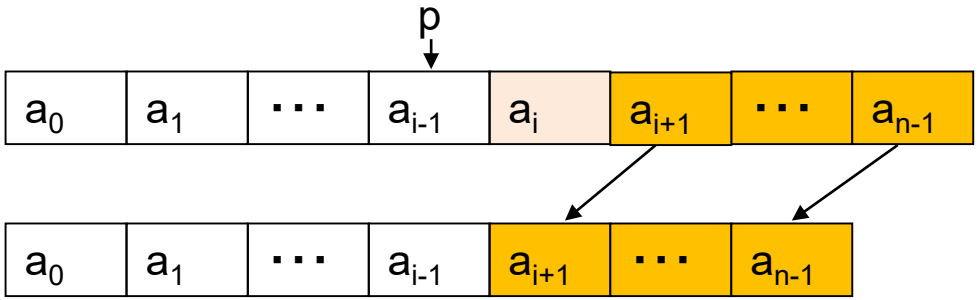


最悪/平均時間計算量は$\Theta(1)$

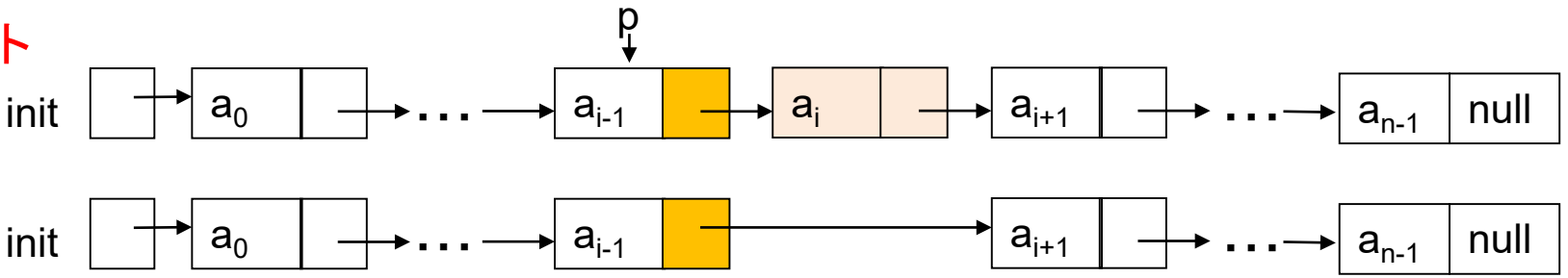
連結リストが得意な処理(削除の場合)

配列の場合

最悪/平均
時間計算量は$\Theta(n)$

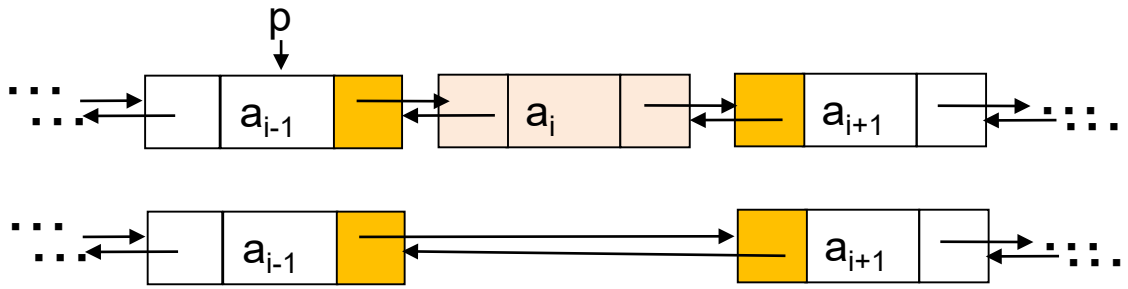


連結リスト
の場合



最悪/平均時間計算量は$\Theta(1)$

双方向連結リストの場合

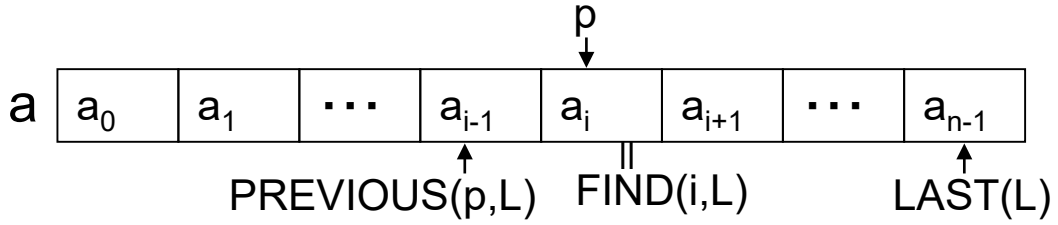


最悪/平均時間計算量は$\Theta(1)$

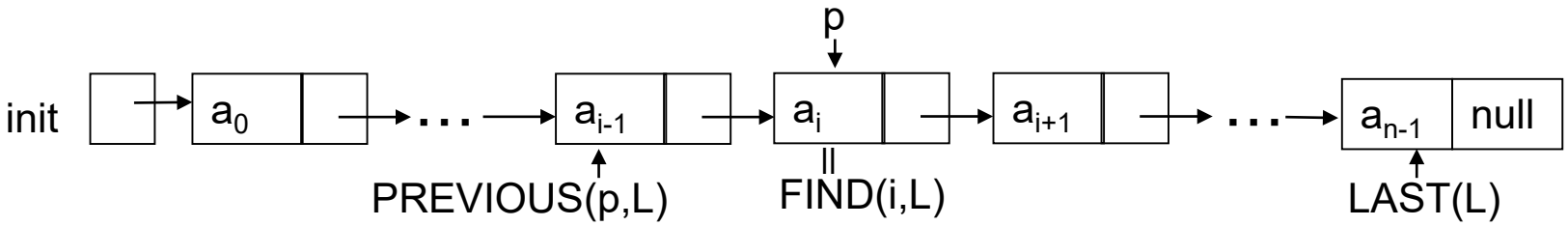
配列が得意な処理(i番目の要素へのアクセス)

配列の場合 連続領域であるためi番目の要素や前後の要素に1ステップでアクセス可能

FIND(i,L): $\Theta(1)$
LAST(L): $\Theta(1)$
PREVIOUS(p,L): $\Theta(1)$

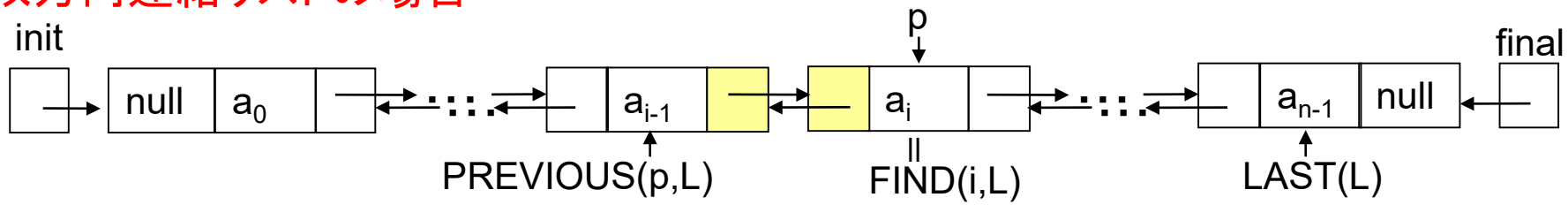


連結リストの場合



FIND(i,L): $\Theta(n)$, LAST(L): $\Theta(n)$, PREVIOUS(p,L): $\Theta(n)$

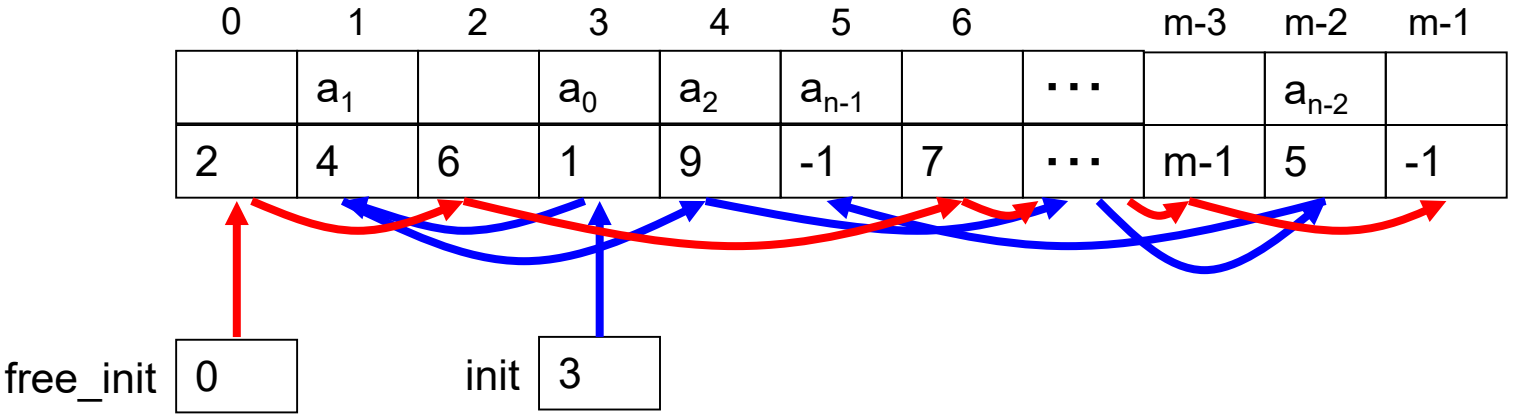
双方向連結リストの場合



FIND(i,L): $\Theta(n)$, LAST(L): $\Theta(1)$, PREVIOUS(p,L): $\Theta(1)$

配列によるコンパクトな連結リストの実現方法

連結リストを配列内に配置し，要素が入っていないセルを管理するリストと混在させることで実現する



メリット

- メモリー確保、解放の時間を節約できる
- メモリー使用量を制御できる
- ガベージコレクションの必要がない
- INSERT操作が$\Theta(1)$時間でできる

DELETEは$\Theta(n)$時間

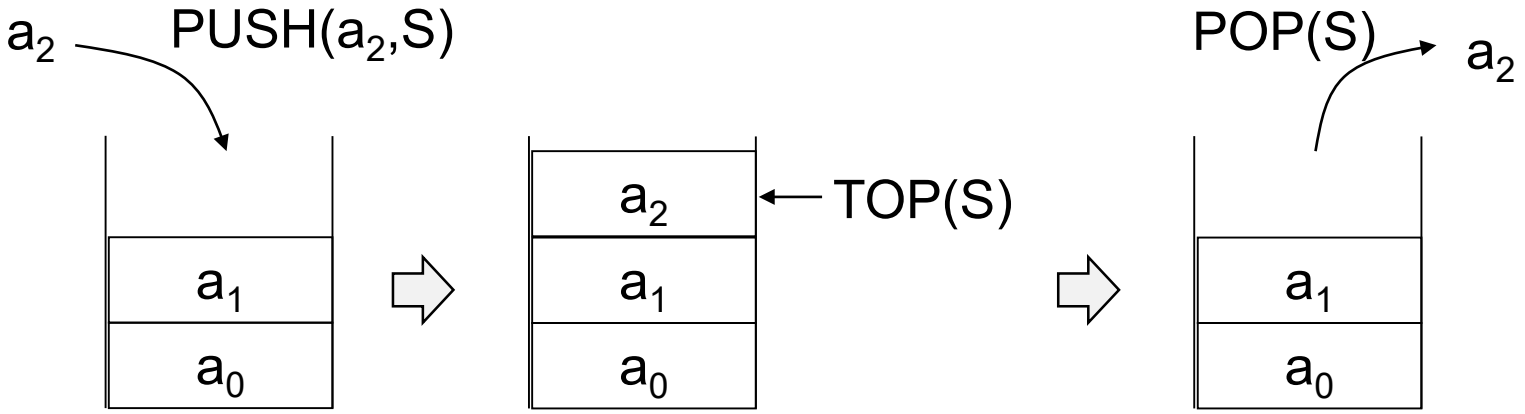
スタック(stack)とは？

要素の挿入、削除がいつも先頭からなされるリスト

LIFO (last-in-first-out)

基本操作

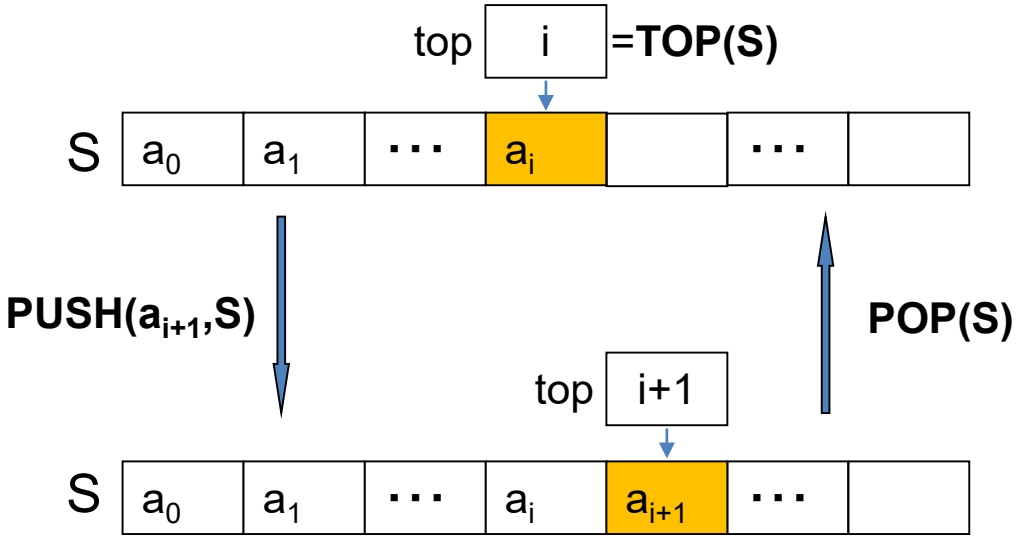
TOP(S)	スタックSの先頭の位置を返す
POP(S)	スタックSの先頭の要素を削除
PUSH(x,S)	スタックSの先頭に要素xを挿入



スタックの実現法

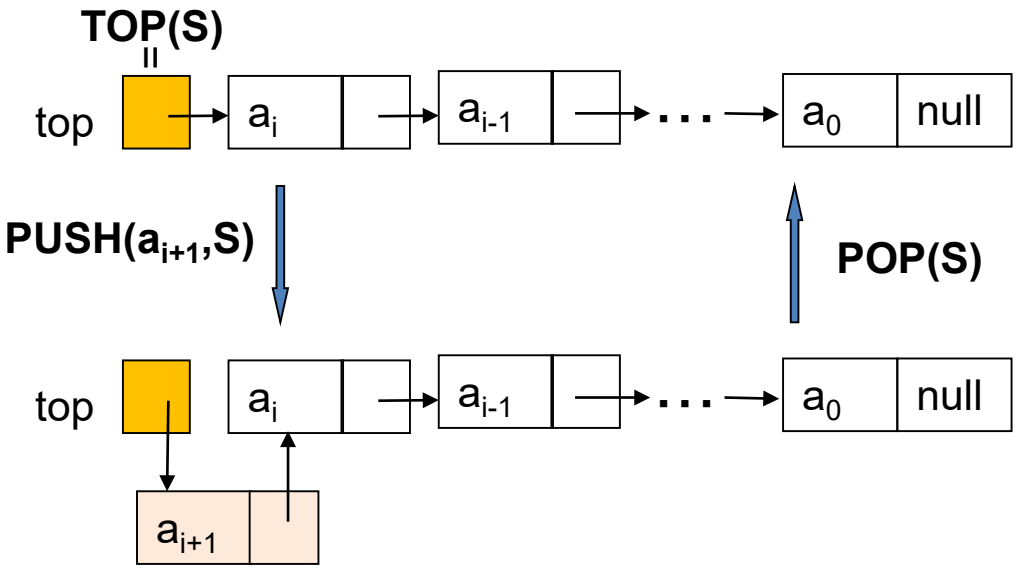
配列による実現

すべての操作の
時間計算量は$\Theta(1)$



連結リストによる実現

すべての操作の
時間計算量は$\Theta(1)$



キュー(queue; 待ち行列)とは？

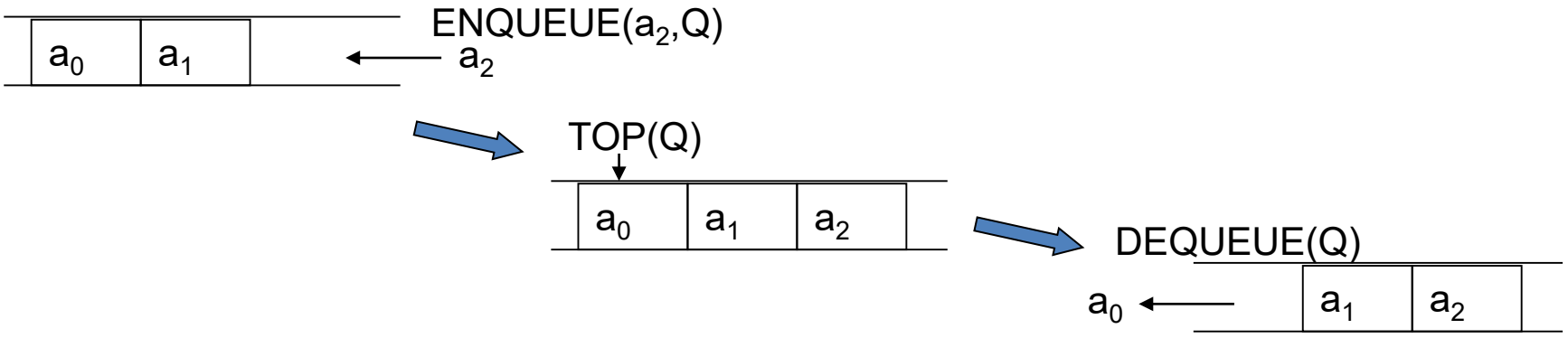
要素の挿入は最後尾、削除は先頭からなされるリスト

FIFO (first-in-first-out)

基本操作

TOP(Q)	キューQの先頭の位置を返す
ENQUEUE(x, Q)	要素xをキューQの最後尾に入れる
DEQUEUE(Q)	先頭の要素をキューQから除く

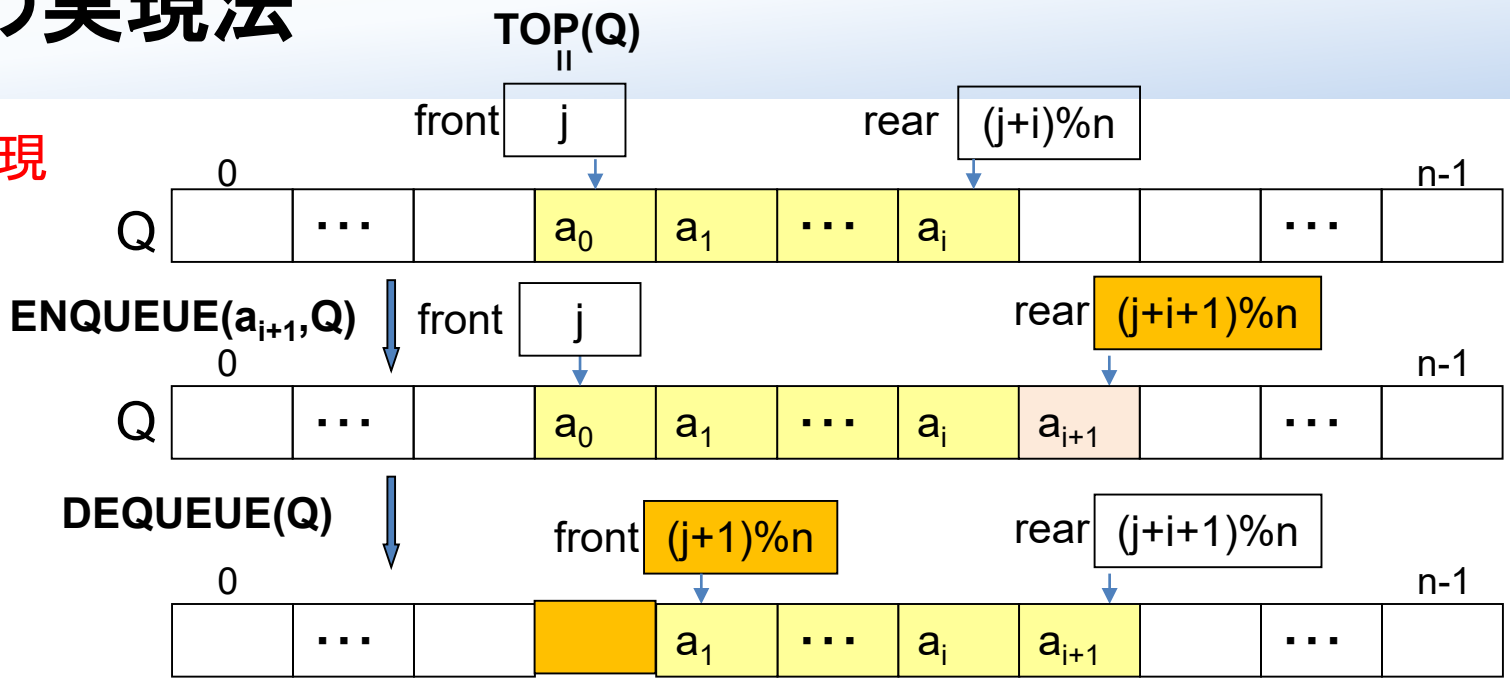
※「エンキュー」「デキュー」と読む



キューの実現法

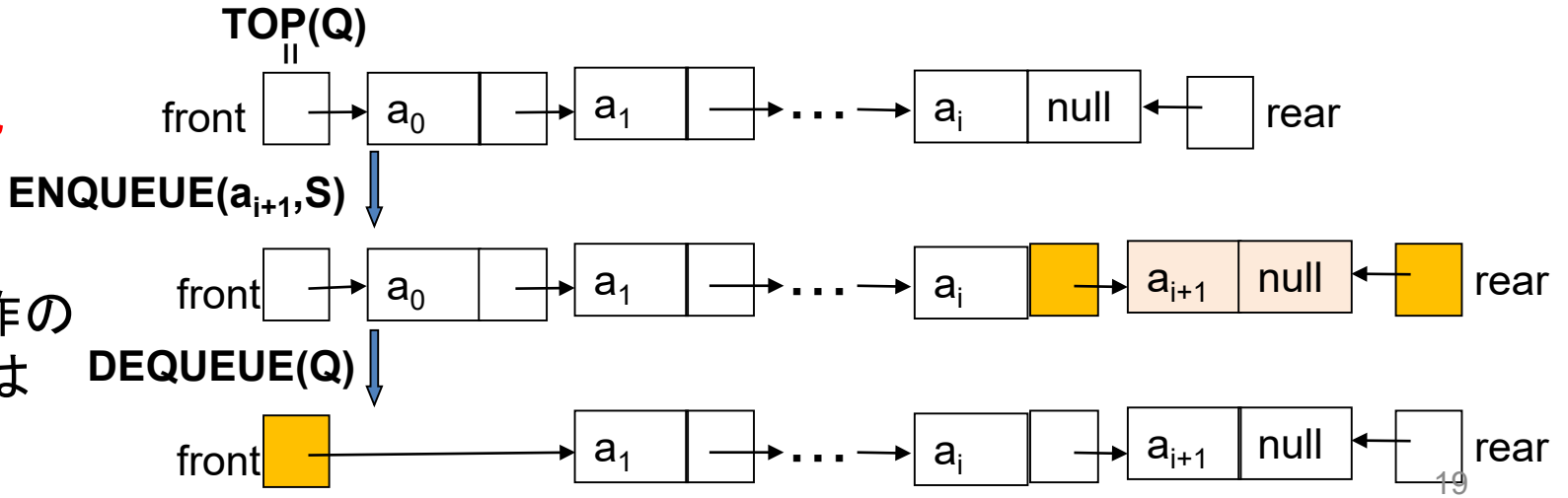
配列による実現

すべての操作の時間計算量は $\Theta(1)$



連結リストによる実現

すべての操作の時間計算量は $\Theta(1)$



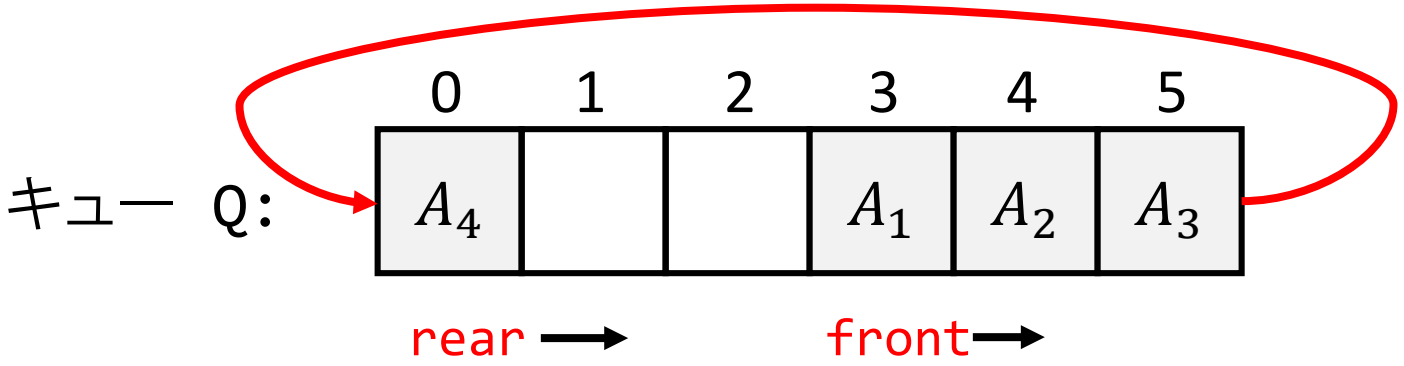
配列によるキューの実現

配列で巡回リストを表す

キューの長さ n が限定された場合

先頭と末尾がつながって，輪になったリスト

要素の位置を n の剰余演算で決定



先頭から i 番目の要素 = Q[(front+ i) mod n]

（例）front=3, n=6のとき，（0から数えて）3番目の要素は？

Q[(front + 3) mod n] = Q[6 mod 6] = Q[0]

今日のまとめ

基本的なデータ構造

リスト：　最も基本的なデータ集合の表現

配列／連結リスト／双連結リストによる実装

スタック：　積み上げ式のデータ格納方式

キュー：　入れた順に取り出せるデータ格納方式

ポイント

ポインタを用いたデータ構造

抽象データ型とその実装