アルゴリズムとデータ構造

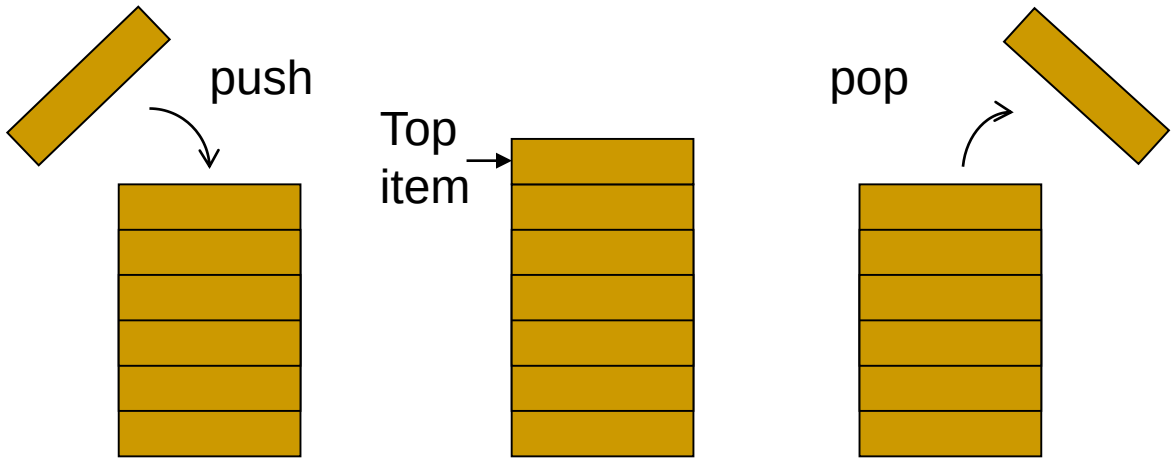
- 第3回講義トピック：基本データ構造（その２）
  - スタック
  - キュー
  - 抽象データ型

Algorithms and Data Structures

- Lecture note 3: basic data structures (part 2)
  - stack
  - queue
  - abstract data structure

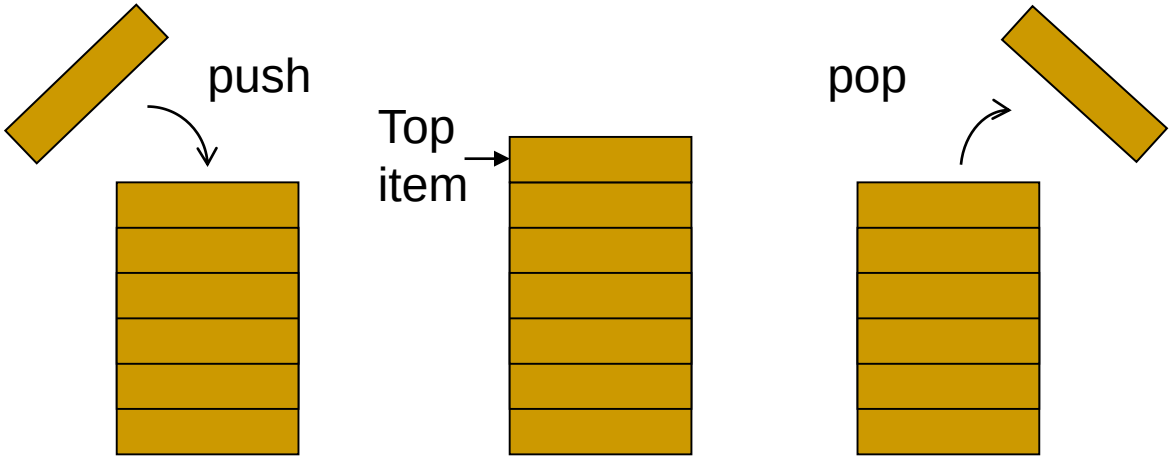
スタック (stack)

- スタックはすでにある要素の上に新しい要素を積み重ねるので、プッシュダウンスタック (pushdown stack)とも呼ぶ。
- スタックの2つ基本的な操作
  - プッシュ (push) : スタックに項目を押し込む。
    - リストの先頭に挿入する。
  - ポップ (pop) : スタックから項目を取り出す。
    - リストの先頭を削除する。
- LIFO (last-in-first-out): 最後に入るものが最初に出る。



Stack

- Stack, also called as pushdown stack, is a collection of data elements with order.
- Stack has two main operations
  - push: to insert elements to its head.
  - pop: to take elements from its head; the elements are to be deleted after pop operation.
- LIFO (last-in-first-out): the last pushed element will be popped in first.



連結リストによるスタックの記述

- リンク構造で表した場合は先頭への追加と削除しかできない特殊なリストとしても考えられる。

```
typedef struct StackNode *StackPointer
struct StackNode {
    ItemType item;
    StackPointer next;
};

StackPointer stack;
```

Stack implemented by linked list

- A stack can be considered as a special case of linked list, where insertion and deletion of elements can only be done from its head.

```
typedef struct StackNode *StackPointer
struct StackNode {
    ItemType item;
    StackPointer next;
};

StackPointer stack;
```

リストによるスタックの実装

```
typedef struct StackNode *StackPointer
struct StackNode{ int key; StackPointer next; };
StackPointer head, z, t;
void StackInit() {
  head = malloc(sizeof *head);
  z = malloc(sizeof *z);
  head->next = z; head->key = 0; z->next = z; }
void push(int v) {
  t = malloc(sizeof *t);
  t->key = v; t->next = head->next;
  head->next = t;}
int pop() {
  int x; t = head->next;
  head->next = t->next; x = t->key;
  free(t); return x; }
int StackEmpty() { return head->next == z; }
```

Stack implemented by linked list

```
typedef struct StackNode *StackPointer
struct StackNode{ int key; StackPointer next; };
StackPointer head, z, t;
void StackInit() {
  head = malloc(sizeof *head);
  z = malloc(sizeof *z);
  head->next = z; head->key = 0; z->next = z; }
void push(int v) {
  t = malloc(sizeof *t);
  t->key = v; t->next = head->next;
  head->next = t;}
int pop() {
  int x; t = head->next;
  head->next = t->next; x = t->key;
  free(t); return x; }
int StackEmpty() { return head->next == z; }
```

配列によるスタックの実装

- スタックの最大の大きさが前もって予測できる場合には、配列を使っても実現できる。
- この場合、リストのリンク部分がない分、配列を使う方が効率があがる場合が多い。

```
#define MAX 100
int stack[MAX+1], p;
void StackInit() { p = 0; }
void push(int v) { stack[p++] = v; }
int pop() { return stack[--p]; }
int StackEmpty() { return !p; }
```

＊スタックの追加と削除は先頭からしか行われないので、配列による実装に適している。

Stack implemented by array

- Stack can also be implemented by using array when the maximum number of elements can be predicted previously.
- Stack using array is more efficient than using linked list.

```
#define MAX 100
int stack[MAX+1], p;
void StackInit() { p = 0; }
void push(int v) { stack[p++] = v; }
int pop() { return stack[--p]; }
int StackEmpty() { return !p; }
```

スタックを利用して算術式を計算する

- スタックを利用して途中の計算結果を記憶する
- 演算の順番は括弧による指定と左から右への約束に従う
- 例：5*(((9+8)*(4*6))+7)

```
push(5);
        push(9); push(8);
        push(pop()+pop());
        push(4); push(6);
        push(pop()*pop());
      push(pop()*pop());
   push(7);
   push(pop()+pop());
push(pop()+pop());
printf(“%d¥n”, pop());
```

Using a stack to evaluate arithmetic expressions

- To evaluate an arithmetic expression, we need to temporally store results of partial expressions.
- A stack can be used for such purpose.
- Example: 5*(((9+8)*(4*6))+7)

```
push(5);
        push(9); push(8);
        push(pop()+pop());
        push(4); push(6);
        push(pop()*pop());
      push(pop()*pop());
   push(7);
   push(pop()+pop());
push(pop()*pop());
printf("%d¥n", pop());
```

逆ポーランド記法

- 算術式の記述方法は最もよく使う中間記法のほか、前置記法と後置記法がある。
- 前置記法は演算子が被演算子の前に、後置記法では演算子が被演算子の後にそれぞれ置かれる。
- 後置記法は逆ポーランド記法とも言われる
  - 前置記法:　　＊５＋＊＋９８＊４６７
  - 中間記法:　　5*(((9+8)*(4*6))+7)
  - 後置記法:　　５９８＋４６＊＊７＋＊
- 特に後置記法はスタックを使って計算処理に適している
  - 被演算子が入力された場合は単にスタックにプッシュする
  - 演算子が入力された場合はスタックから被演算子をポップし、演算を行う

逆ポーランドの特徴

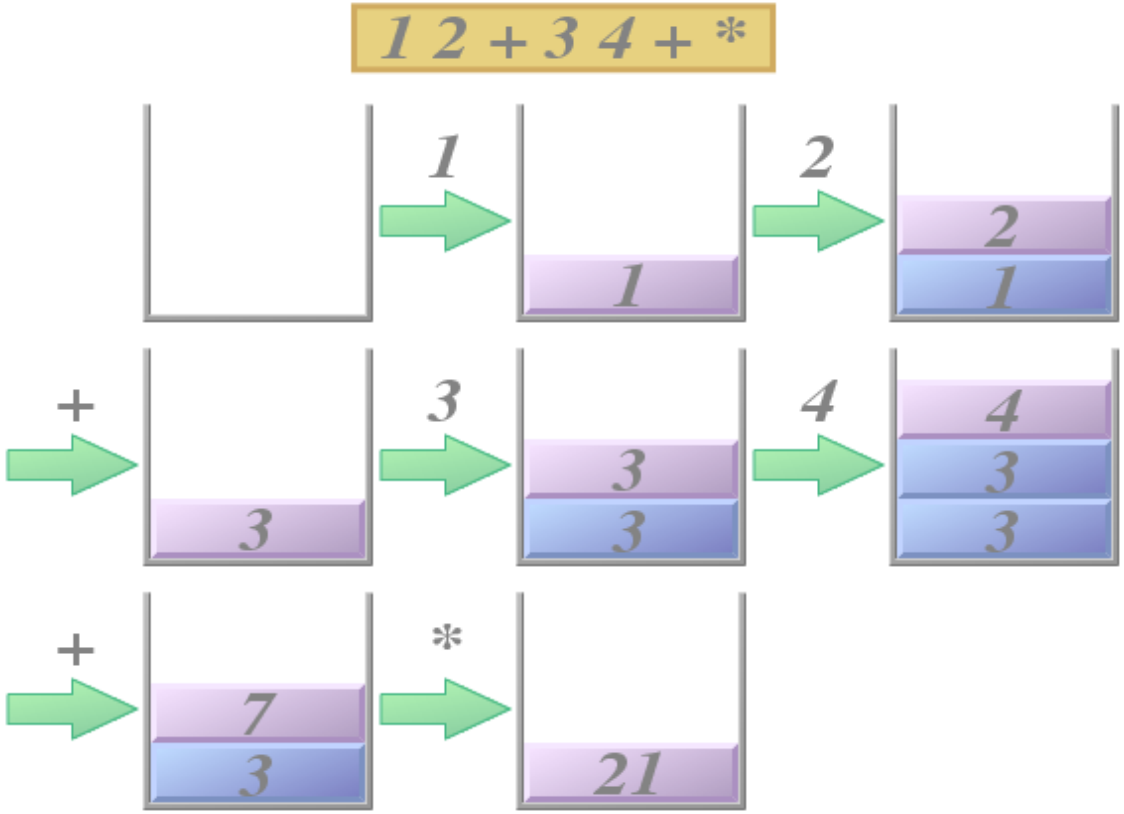
- 逆ポーランド記法は日本語表記と語順が似ている。
  - 例えば、1 + 2 を逆ポーランド記法で表すと 、1 2 + となる。
  - これは日本語「1 と 2 を足す(+)」と語順が同じである。
- 逆ポーランド記法にすると括弧が不要となる。
- 逆ポーランド記法の数式は人間には読みにくいが、コンピュータが処理をする場合にはとても都合が良い。

Reverse polish notation (RPN)

- Also known as postfix notation, in which every operator follows its operands.
- In contract:
  - Infix notation: operators are in between of operands.
  - Prefix notation: operators precede operands.
- For example:
  - Prefix notation: ＊ 5 + ＊ + 9 8 ＊ 4 6 7
  - Infix notation: 5*(((9+8)*(4*6))+7)
  - Postfix (reverse polish) notation: 5 9 8 + 4 6 ＊ ＊ 7 + ＊
- Since there is no ambiguity for operators with respect to their operands, postfix notation needs no parentheses.
- Using stack with reverse polish notation
  - with an operand, push it in the stack
  - with an operator, pop two operands in the stack and than push the evaluation result.

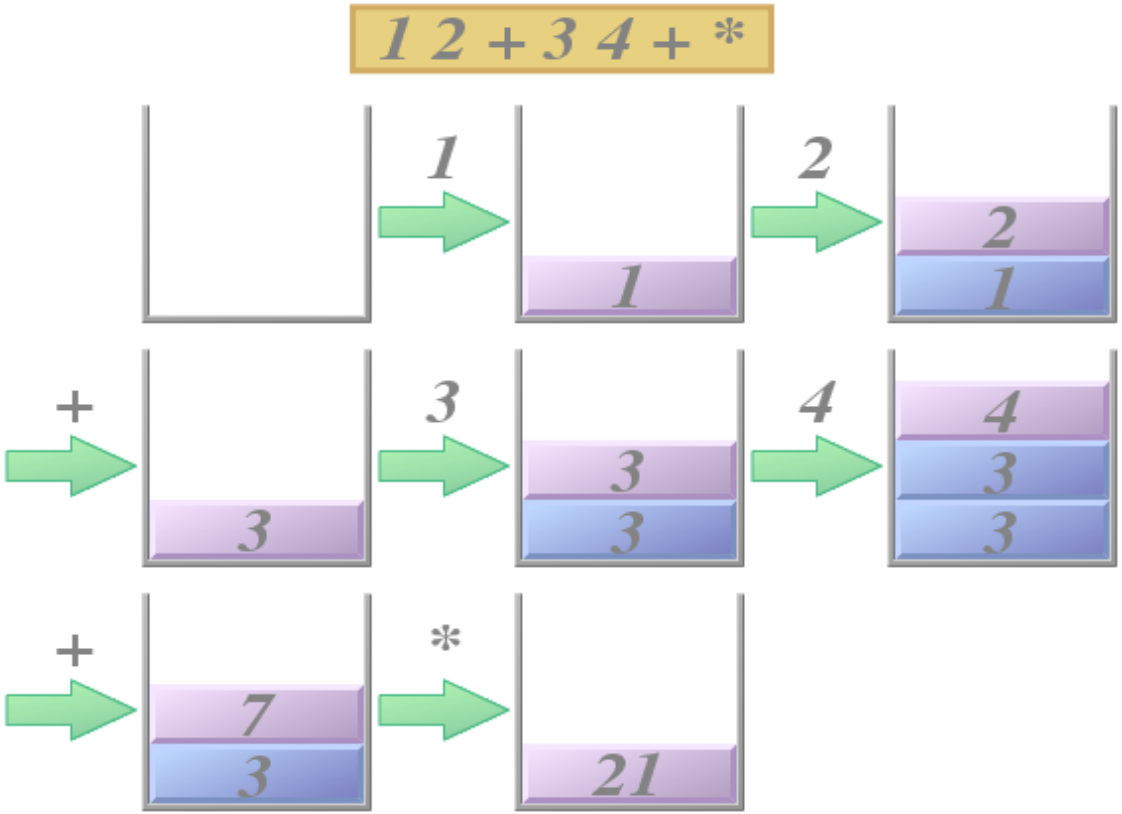
スタックを使った計算過程

中間記法の式：(1+2)*(3+4)



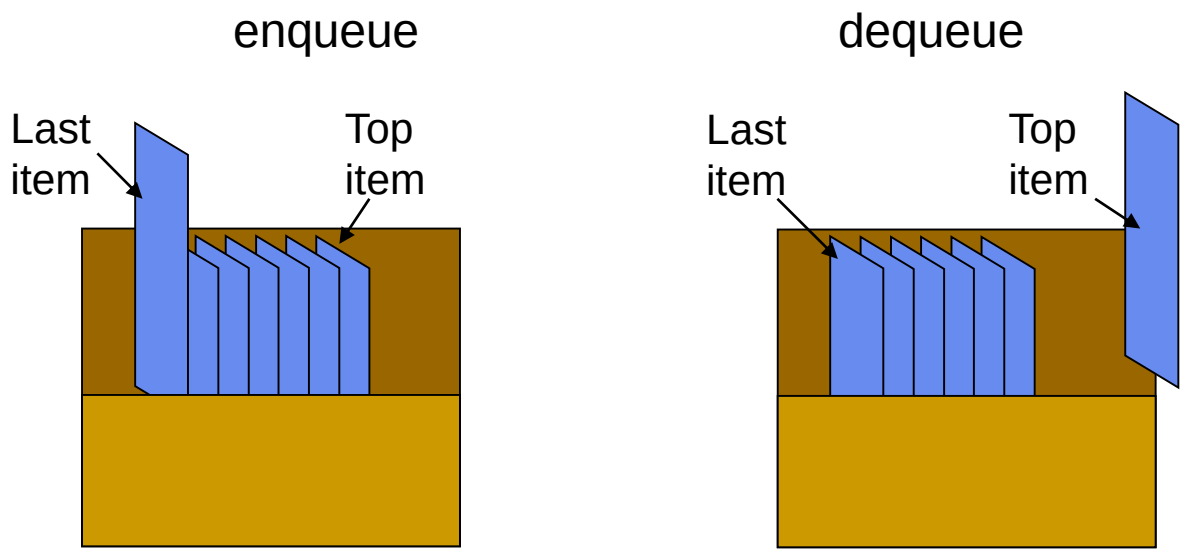
RPN evaluation using stack

Infix notation : (1+2)*(3+4)
Postfix (RPN) notation: 12+34+*



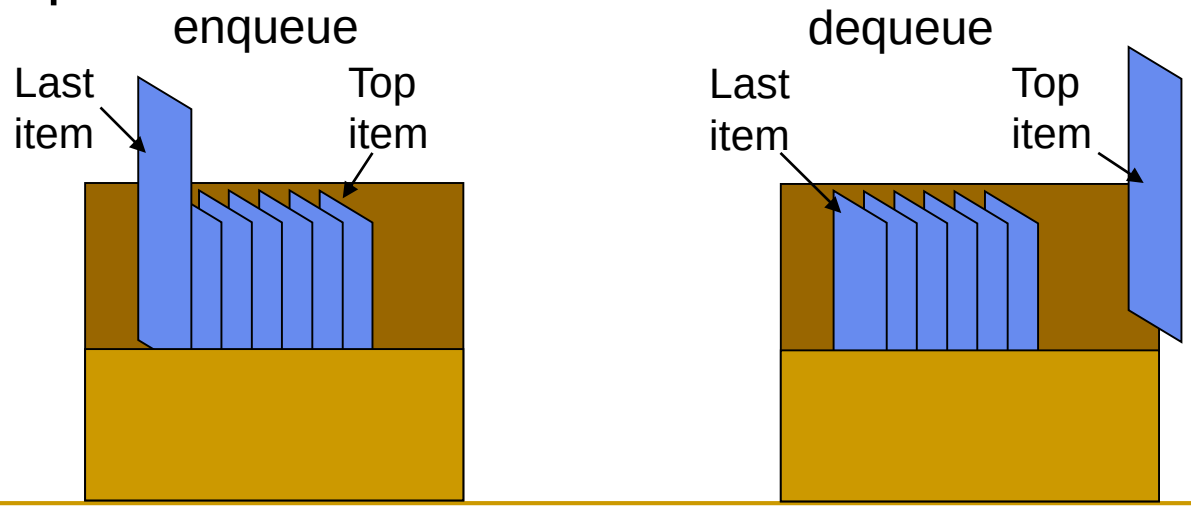
キュー (queue) : 順番を待つ列

- FIFO (first-in-first-out):
  - 最初に入れたものが最初に出てくる
- 2つ基本的な操作
  - enqueue: キューの末尾に要素を挿入する
  - dequeue: キューの先頭の要素を削除（取り出す）する



Queue

- A queue is a collection of data elements with order.
- A queue has two main operations:
  - enqueue: to insert an element to its tail
  - dequeue: to take (and delete after output) an element in the head
- FIFO (first-in-first-out): the first inserted element will be output in first.



連結リストによるキューの表現

- キューも特殊なリストの1つ
- 2つのポインタが必要headとtail あるいは frontとrear

```
typedef struct Node *QueuePointer
struct Node{ int key; QueuePointer next; };
typedef struct {
  QueuePointer head;
  QueuePointer tail;
} Queue;
Queue queue;
void QueueInit() { queue.head=NULL; queue.tail=NULL; }
…
```

*enqueue(), dequeue()については連結リストの機能を参照

Queue by linked list

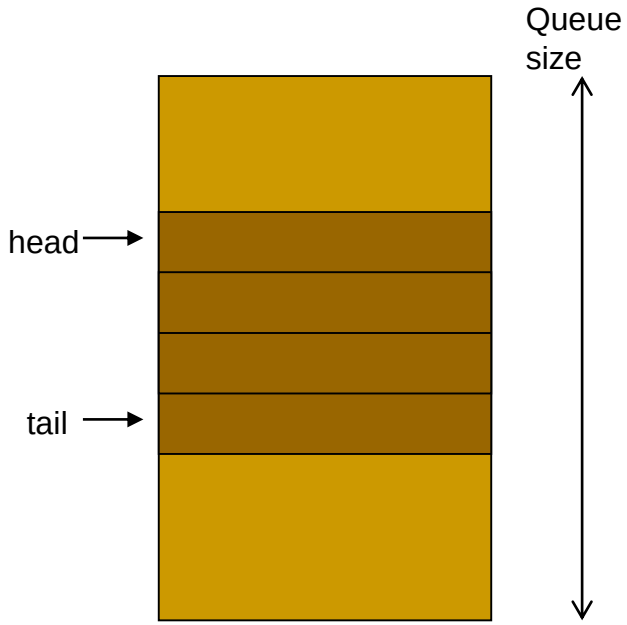
- A queue can also be considered as a special case of linked list
- Two pointers head and tail (or front and rear)

```
typedef struct Node *QueuePointer
struct Node{ int key; QueuePointer next; };
typedef struct {
  QueuePointer head;
  QueuePointer tail;
} Queue;
Queue queue;
void QueueInit() { queue.head=NULL; queue.tail=NULL; }
…
```

*for implementation of enqueue and dequeuer to refer to linked list.

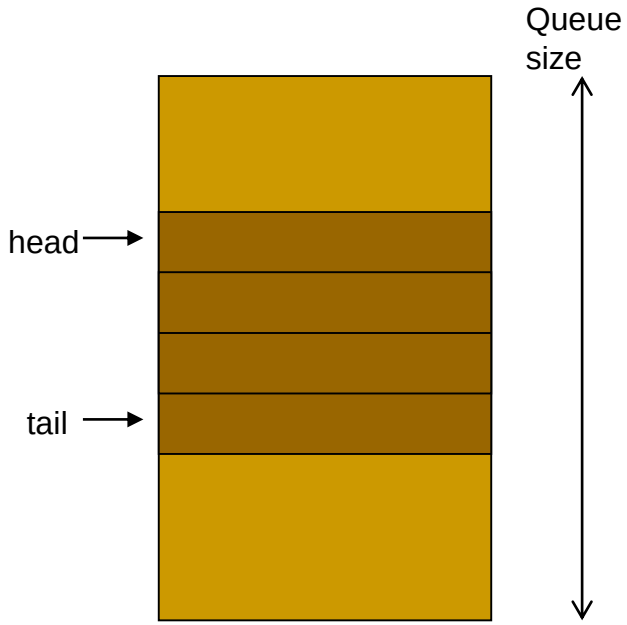
配列による表現

```
#define MAX 100
int queue[MAX+1], head, tail;
void QueueInit() { head =0; tail = 0; }
void enqueue(int v) { queue[tail++] = v;
  /*配列の端末にきた時には0に戻る*/
  if (tail > MAX) tail =0; }
int dequeue() {
  int t = queue[head++];
  /*配列の端末にきた時には0に戻る*/
  if (head > MAX) head = 0;  return t; }
int QueueEmpty() { return head == tail; }
```



Queue by an circular array

```
#define MAX 100
int queue[MAX+1], head, tail;
void QueueInit() { head =0; tail = 0; }
void enqueue(int v) { queue[tail++] = v;
   /* when the index reach the end of array,
      it is circulated to the beginning again*/
   if (tail > MAX) tail =0; }
int dequeue() {
   int t = queue[head++];
 /* when the index reach the end of array,
      it is circulated to the beginning again*/
   if (head > MAX) head = 0;  return t; }
int QueueEmpty() { return head == tail; }
```



抽象データ型

- データの構造とそのプログラムによる実現法を分離する考え方。
- ユーザーはデータの構造と使い方（インターフェース部分）だけを知り、その実現方法についてはブラックボックスでもかまわない。
- もちろん、実現法が異なれば、性能も大きく異なるが、開発者以外のユーザーはその部分には関与しない。
- 例えば:
  - 配列や連結リストはともに「線形リスト」という抽象データ型を詳細化したもので、その定義に従って中のデータを操作することができる。
  - スタックとキューのデータ構造を実現したライブラリがあるとする。プログラムでは、定義された基本操作だけを利用し、具体的な実現法については関与しない。

抽象データ型の特徴と利点

- 目的：ライブラリや汎用ツールなどプログラムの開発用の道具にする。
- 特徴：データ構造とそのアルゴリズムの内部定義が分離され、内部定義の部分に外部からは直接参照できない。決められたやり方で、関数などのインターフェースを通してのみ、参照することができる。
- 利点：
  - 複雑なデータ構造とアルゴリズムをインターフェースから切り離すことで、利用するユーザーの複雑さを限定することができる。
  - プログラムが理解しやすくなり、基本データ構造やアルゴリズムの変更や改良が全体に対する影響が少なくなる。

Abstract data structures

- We have studies about data structures of stack and queue. The two data structures are abstract because they are defined by their abstract structures and abstract operations.
- A stack or queue can be implemented by linked list or array. Through they are different in implementation and therefore performance, but the abstract structures and operations are the same.
- To use an abstract stack or queue, the user do not need to touch with the detail implementation (to leave them black box) but only the abstract structures and operations (called interface).