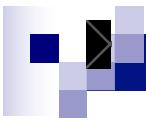


データ構造とアルゴリズム

第一回 ガイダンス、時間計算量、
基本的データ構造 (線形) .

早稲田理工空
WASEDA Scice-tech

早稲田理工 田老师



■ アルゴリズム

- 入力データから正しい出力を計算するためには、一連の手順を記述したもの。
- 正しさと効率が重要。

■ データ構造: 存在指

- 計算のために、記憶領域に効率良くデータを格納するための配置法。
- 時間と記憶領域を効率化。

"

N. Wirth 1976

アルゴリズム + データ構造 = プログラム"



アルゴリズムとデータ構造

よいアルゴリズムとは？

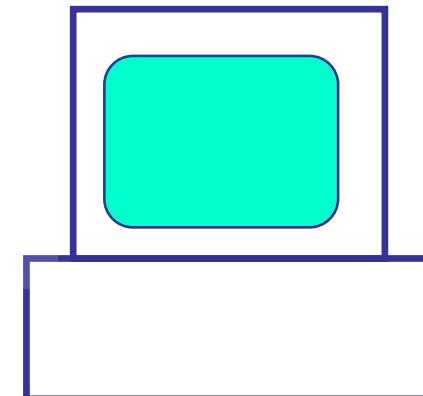
いろいろな性能

- 計算時間
- メモリサイズ
- 外部記憶への入出力数
- ネットワークの通信量

ほかの要素

- モジュラリティ Modularity
- 再利用可能性 Reusability
- 読みやすさ Readability
- 移植しやすさ Portability
- ...

・ここでは、計算時間とメモリサイズで測る

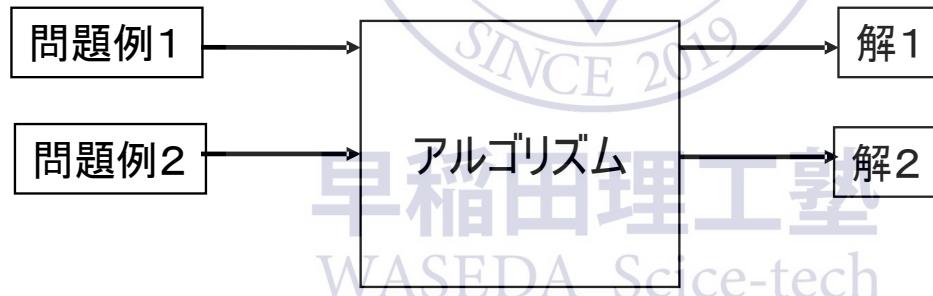


計算量の評価

■ 計算量の種類

- 時間計算量(time complexity)
- 領域計算量(space complexity)

■ ある問題を解くアルゴリズム



1つの問題 = 無限個の問題例の集合

計算量

ステップ数
メモリ量



重要





ここは大事！

「XXアルゴリズムの計算時間は入力の二乗時間オーダー」
「クイックソートアルゴリズムは $O(n \log n)$ 時間」
オーダーとは何か？



1.3 計算量の評価、漸近的計算量

<http://www.shutterstock.com/>

「オーダー」記法の勉強 工塾

WASEDA Scice-tech

ステップね。ここは大事！

漸近的計算量: $O(\text{ビッグオー})$

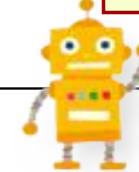
$$f(n) = n, \quad T(n) = O(n)$$

オーダー評価

漸近的上界 = asymptotic upper bound

- 計算量 $T(n)$ は十分大きな n に対して評価する。
- 定数倍の差はないものとみなす。

重要



定義: 漸近的上界(ビッグオー記法) 記号 O

$T(n) = O(f(n)) \Leftrightarrow$ ある実数 $c > 0$ と自然数 n_0 が存在して全ての
 $n \geq n_0$ に対して $T(n) \leq cf(n)$ が成り立つ

「 $T(n)$ は、オーダー $f(n)$ である」または、「 $T(n)$ は、ビッグオー $f(n)$ である」と読む

単にオーダー記法ともいう

意味 「関数 $T(n)$ は $f(n)$ と同じか小さい」

漸近的上界はいくらでも存在するができるだけ単純で精度のよいものがよい

(1) $2n^2 + 5n + 1000 = O(n^2) \leftarrow$ 一番良い best!

(2) $2n^2 + 5n + 1000 = O(n^2+n) \leftarrow$ (1)より複雑

(3) $2n^2 + 5n + 1000 = O(n^3) \leftarrow$ (1)より精度が悪い

$$f(n) = 2n^2, \quad g(n) = 5n, \quad m(n) = 1000$$

アルゴリズムとデータ構造

$$f(n) \cdot g(n) = 10n^3$$

漸近的計算量の性質

性質： $T_1(n)=O(f(n))$, $T_2(n)=O(g(n))$ のとき、次の等式が成立

- $T_1(n) \pm T_2(n) = O(\max\{f(n), g(n)\})$ ← いくつかの処理を順次行う場合は一番遅い処理が全体の処理速度を支配する
- $T_1(n)T_2(n) = O(f(n)g(n))$ ← 処理を繰り返し行うとその回数分時間がかかる

証明してみよう！

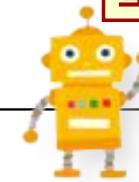
(例) $n^2 + n^3 = O(n^3)$, $n^2 \cdot n^3 = O(n^5)$

オーダー表記の注意点 左辺の精度 \geq 右辺の精度

(例) $\textcircled{O} 2n^2 + 5n + 1000 = O(n^2)$
 $\times O(n^2) = 2n^2 + 5n + 1000$

オーダーの計算法：基本のやりかた

重要



規則1： $T(n)$ が n の多項式ならば、最大次数の項のオーダーになる

例： $2n^2 + 3n + 100 = O(n^2)$

例： $10n + 2\sqrt{n} + 5 = 10n^1 + 2n^{0.5} + 5$ ここに数式を入力します。

規則2： 次のオーダーの式が成立する：

- $\log(n) = O(n)$
- $n = O(2^n)$
- 任意の $c > 0$ に対して、 $\log n = O(n^c)$

規則3： $T(n)$ がいくつかの項の和ならば、最大次数の項のオーダーになる

例： $3n^2 + 2\sqrt{n} + 100 \log n + 5 = O(n^2)$

アルゴリズムとデータ構造

$3n^2 + 2\sqrt{n} + 100 \log n + 5 = O(\log n)$

訂正：前回のスライドで上のように記載していましたがこれは間違います。
正しくは左のとおりです。ご指摘ありがとうございます。（ちなみに、 $n^2 = 2^{2\log n}$
ですので、 n^2 の方が $\log n$ より、指数的に大きいです。）

オーダーの計算法：自分で計算してみる 増加のオーダーによる比較(1/2)

重要



計算時間が $T_1(n)$ と $T_2(n)$ のアルゴリズムではどちらが速いか?
⇒ 増加のオーダーが小さい方が速い

$T_1(n)$ VS $T_2(n)$

規則4: $T_1(n)$ よりも $T_2(n)$ の方が増加のオーダーが大きい

$$\Leftrightarrow \lim_{n \rightarrow \infty} \frac{T_2(n)}{T_1(n)} = 0 \quad \Leftrightarrow \lim_{n \rightarrow \infty} \frac{T_1(n)}{T_2(n)} = \infty$$

規則5: $T_1(n)$ と $T_2(n)$ は増加のオーダーが等しい

$$\Leftrightarrow T_1(n) = \Theta(T_2(n)) \Leftrightarrow T_2(n) = \Theta(T_1(n))$$

$T_1(n)$ と $T_2(n)$ は増加のオーダーが等しい

$$\lim_{n \rightarrow \infty} \frac{T_1(n)}{T_2(n)} = c \text{ for some } 0 < c < \infty$$

$$\lim_{x \rightarrow a} \underbrace{h(x)}_{\downarrow} = \lim_{x \rightarrow a} \frac{f'(x)}{g'(x)}$$

オーダーの計算法: 増加のオーダーによる比較(2/2)

発展

規則6: l'Hospitalの法則

1. $f(x), g(x): R \rightarrow R$ が微分可能で $f(a)=g(a)=0, x=a$ 以外で $g'(x) \neq 0$ であれば

$$\lim_{x \rightarrow a} \frac{f(x)}{g(x)} = \lim_{x \rightarrow a} \frac{f'(x)}{g'(x)}$$

2. $f(x), g(x): R \rightarrow R$ が微分可能で $\lim_{x \rightarrow \infty} f(x) = \lim_{x \rightarrow \infty} g(x) = \infty$ であれば

$$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = \lim_{x \rightarrow \infty} \frac{f'(x)}{g'(x)}$$

(例) $n^{0.01}$ と $\log^{100} n$ ではどちらが漸近的増加率が大きいか?

$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{\log^{100} n}{n^{0.01}} &= \lim_{n \rightarrow \infty} \frac{100 \log^{99} n \times (1/n)}{0.01 n^{-0.99}} = \lim_{n \rightarrow \infty} \frac{100 \log^{99} n}{0.01 n^{0.01}} \\ &= \lim_{n \rightarrow \infty} \frac{100 \cdot 99 \log^{98} n}{0.01^2 n^{0.01}} = \dots = \lim_{n \rightarrow \infty} \frac{100!}{0.01^{100} n^{0.01}} = 0 \end{aligned}$$

演習(発展問題)

(5) つぎのオーダー表記を簡略化せよ。

(a) $O(n \log n + n^2) + O(n^{1.83} \log n)$

明らかに $n \log n < n^2$, $n \log n < n^{1.83} \log n$ である。

また、

$$\lim_{n \rightarrow \infty} \frac{n^{1.83} \log n}{n^2} = \lim_{n \rightarrow \infty} \frac{\log n}{n^{0.17}} = \lim_{n \rightarrow \infty} \frac{1/n}{0.17n^{-0.83}}$$

$$= \lim_{n \rightarrow \infty} \frac{1}{0.17n^{0.17}} = 0$$

よって十分大きな n に対して $n^{1.83} \log n < n^2$ であるから

$$O(n \log n + n^2) + O(n^{1.83} \log n) = O(n^2)$$

(b) $O(n^{\log n} + n^{100} + n^{30} \log n)$

$n^{30} \log n < n^{100}$ は明らか。

また十分大きな n に対して $n^{100} < n^{\log n}$ が成り立つ。

よって

$$O(n^{\log n} + n^{100} + n^{30} \log n) = O(n^{\log n})$$

考えて
みよう

任意の $a, b > 0$ に対し

$$\lim_{n \rightarrow \infty} \frac{\log^a n}{n^b} = 0$$

であるから十分大きな n に対しては、常に $\log^a n < n^b$ である。

慣れてしまったら
この知識から
直接結論付けても
OK

(c) $O(n^3 \sin^2 n) O(2^n / \log n)$

$\sin^2 n \leq 1$ であるから

$$\begin{aligned} O(n^3 \sin^2 n) O(2^n / \log n) \\ = O(n^3 2^n / \log n) \end{aligned}$$

2022 年度 10 月期入学 / 2023 年度 4 月期入学
京都大学 大学院情報学研究科
修士課程 知能情報学専攻 入学者選抜試験問題
(情報学基礎)

2022 年 8 月 5 日 9:00~11:00

設問 1 自然数 n の関数 $f(n)$ に対するビッグオー記法 $f(n) = O(g(n))$ を考える。ここで、 $g(n)$ は自然数 n の関数である。以下に示す各 $f(n)$ について、最も簡潔な形を持つ $g(n)$ を答えよ。

- (1) $f(n) = 5 \log n + 2(\log n)^3 + 3n^3$
- (2) $f(n) = n \log n + 10n^2 + 100n$
- (3) $f(n) = 4n! + 2n^n + 8n \log n$

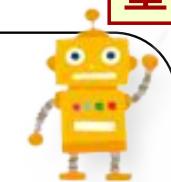
アルゴリズムの計算量

計算量を問題例の入力長Nの関数としてオーダー評価したもの
以下の2つの評価法がある。

重要

最悪計算量(worst case complexity)

入力長がNである問題例の中で最大の計算量



平均計算量(average case complexity)

入力長がNである問題例の計算量の期待値

(例) 入力長がNの問題例に対し確率(N-1)/Nで $O(N)$ 、確率1/Nで $O(N^2)$ であるようなアルゴリズムの計算量

最悪時間計算量 $O(N^2)$

平均時間計算量 $O(N)$

通常は、最悪計算量で評価することが多い。

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組の操作で抽象的に定めたもの。

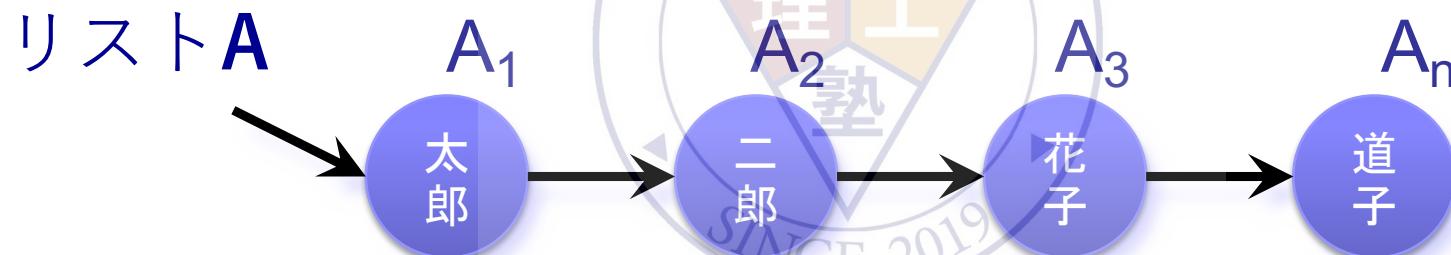


早稲田理工塾

- データ構造(のAPI)を「抽象データ型」ともいう。
- データ構造には、「それは何か(What)」と「それをどのように実現するか？(How)」の二つの面がある。
- 現代的なプログラム言語やライブラリーはこの考え方に基づく。(例:C++, Java, Ruby, Python などなど)

リストとは？（抽象データ型として）

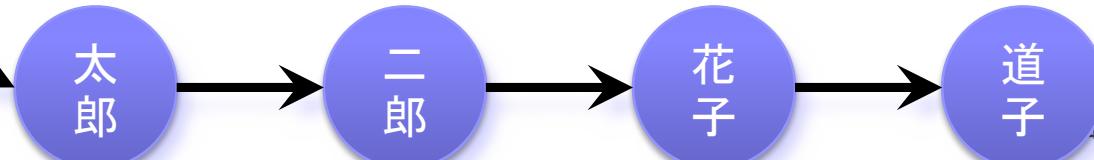
- 0個以上の要素を一列にならべたもの A

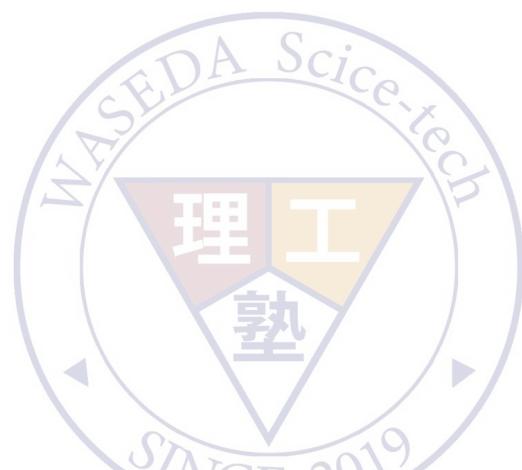
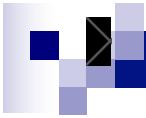


- 空リスト：要素を含まないリストのこと
- リストの長さ：要素数 n
- A_i ：最初から i 番目の要素 ($1 \leq i \leq n$)
- リスト中の場所を指示するための「位置」をもつ

リストに対する操作

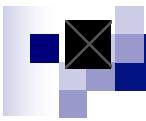
- `List L = create ()` : 空のリストを返す.
- ✓ `insert(L, p, x)` : リストLの位置pの次に要素xを挿入する
- ✓ `delete(L, p)` : リストLの位置pの要素を削除する
- `search (L, x)` : リストLに要素xが含まれていれば1を, なければ0を返す
- `addFirst(L, x)` : リストLの先頭位置に要素xを挿入する
- `find(L, i)` : リストLのi番目のセルの内容を返す
- `last(L)` : リストLの最後のセルの位置を返す
- ✓ `next(L, p)` : 位置pの1つ次のセルの位置を返す
- ✓ `previous(L, p)` : リストLにおいて、位置pの1つ前のセルの位置を返す





さまざまなリストの実現方法(とくに連結リスト)

リストの実現方法



リスト：

■ 配列(array)による実装

1. 配列(array)



0 1 ... n-1

a_0	a_1	...	a_{n-1}
-------	-------	-----	-----------

n=4

n個の連続領域に格納

add	data
0	a_0
1	a_1
2	a_2
3	a_3
.	.
.	.
.	.
.	.
n-1	a_{n-1}

配列

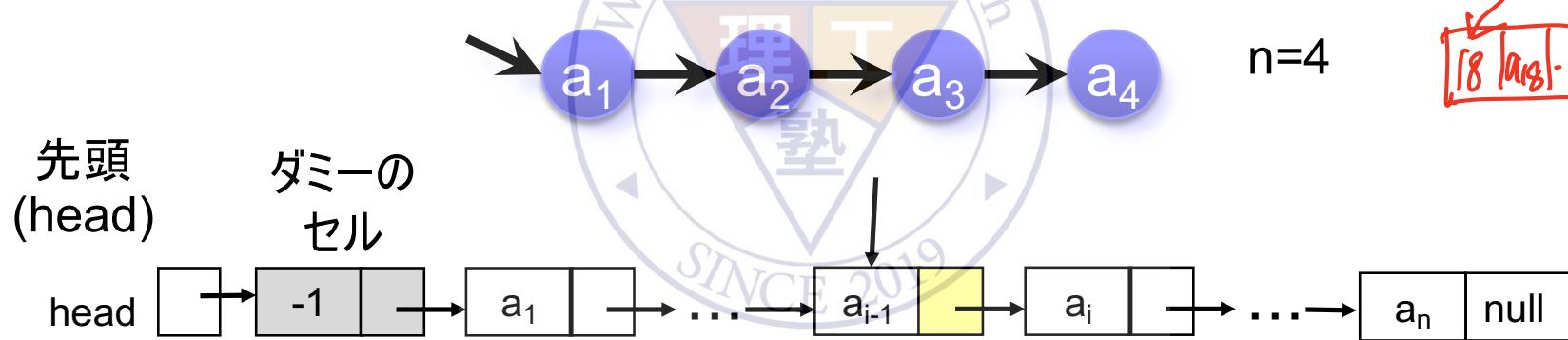
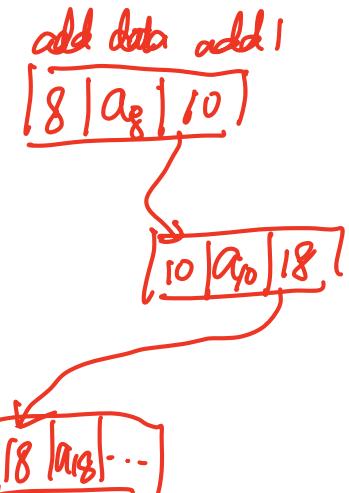
int a[100]; % 100個の整数 $a[0], a[1], \dots, a[99]$ からなる配列

链表

基本

リスト：連結リスト(linked list)による実装

- 要素を保持する「セル」をポインタでつないで、リストを表す。
- 途中への挿入削除を効率よく行える



早稲田理工塾
WASEDA Scice-tech

セル*の構造体(C言語のコード)

セル*(箱図)

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

element	next
cell	
要素	次のポインタ

*セル = 区切られた箱や部屋のこと

復習: ポインタとは？

ポインタ (pointer)

- ◆セルの位置を示すデータ
- ◆機械語レベルでは、セルの番地そのもの
- ◆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。

早稲田理工塾
WASEDA Scice-tech

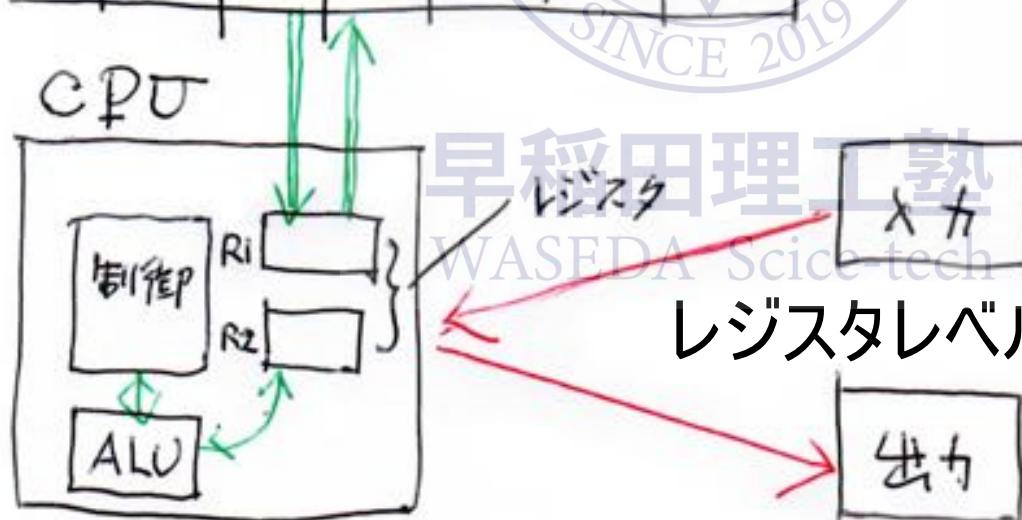


復習: ポインタは番地



C言語の場合

- ◆ ポインタ p が指す変数の値 x を、 $x = *p$ で表す。
- ◆ 変数 x を指すポインタ p を $p = &x$ で取り出せる。



リストの実装方法



リストとは

要素を0個以上1列に並べたもの

(注意)リストは連結リストを指すことが多い

[リスト a_0, a_1, \dots, a_{n-1} の実現法]

[用語]「実装」とは、アルゴリズムや抽象データ型を、プログラムとして実際に作成すること、または、そのくわしい方法。

1. 配列(array)

n個の連続領域に格納



2. 連結リスト(linked list)

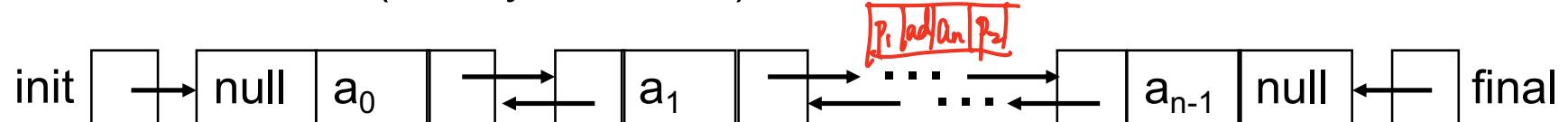
上図

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)

ポインタで前後の要素の格納領域を指す



注) initとfinalポインタを、head(先頭)とtail(末尾)と呼ぶことも多い。

アルゴリズムとデータ構造

連結リストが得意な処理(挿入)

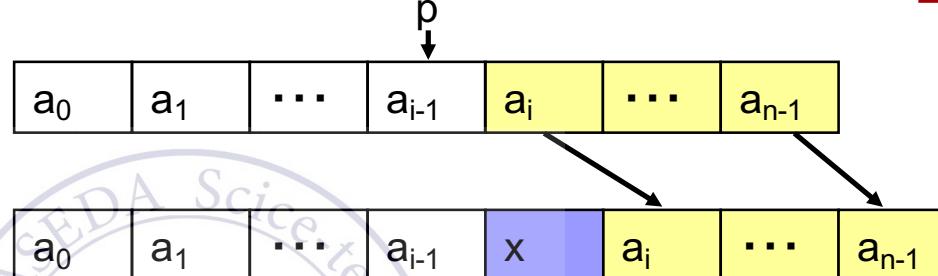
INSERT(x, p, L) : リスト L (要素数 n) の位置 p の次の位置に要素 x を挿入

重要

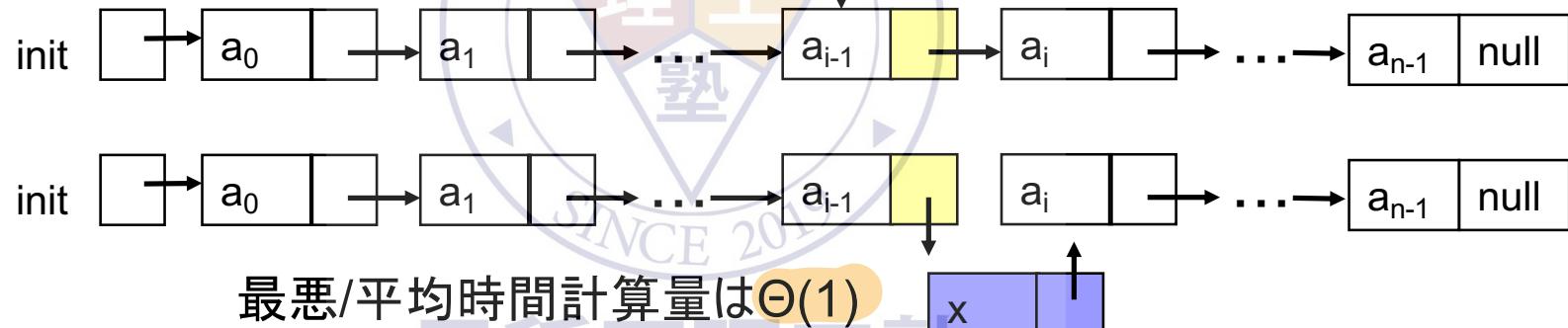
配列の場合

最悪/平均

時間計算量は $\Theta(n)$

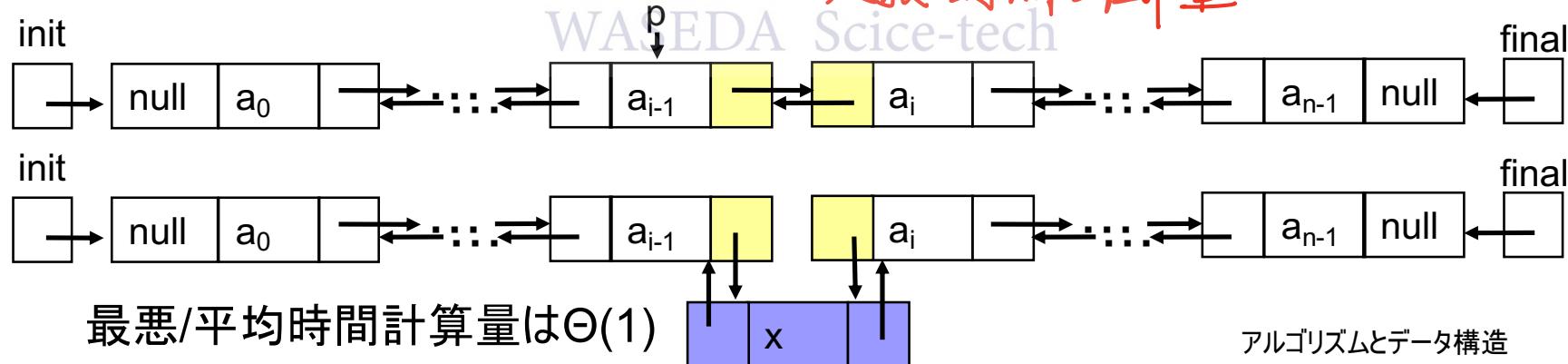


連結リストの場合



双向連接リストの場合

定数時間計算量



擬似コード(伪代码)

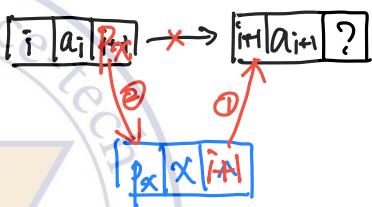
insert(x, p, L)

- 配列 a_1, a_2, \dots, a_n \xleftarrow{x} $a_1, a_2, \dots, a_{j-1}, \underset{j}{\textcircled{X}}, a_{j+1}, \dots, a_n$

```
for (i=n, i--, i≥j) {  
    ai+1 = ai; // 把元素往后移一位  
}
```

- 連結リスト

- ① $x.\text{next} = i+1$
- ② $a_i.\text{next} = p_x$



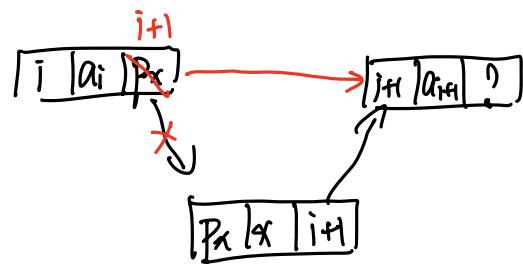
delete(p, L)

- 配列

```
for(i=j, i++, i<n) {  
    ai = ai+1; // 移动  
}
```

- 連結リスト

$a_i.\text{next} = i+1$



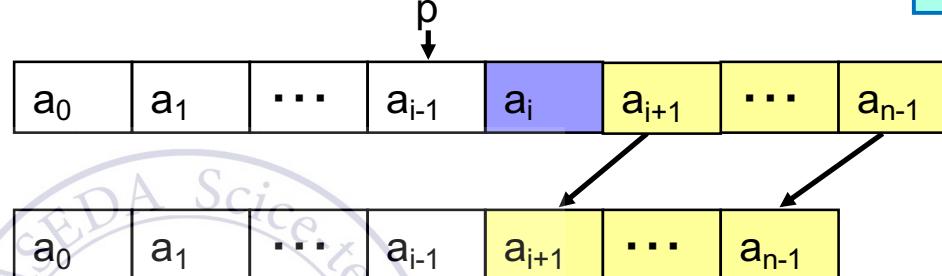
連結リストが得意な処理(削除)

DELETE(p, L) : リスト L (要素数 n) の位置 p の次の位置の次の要素を削除

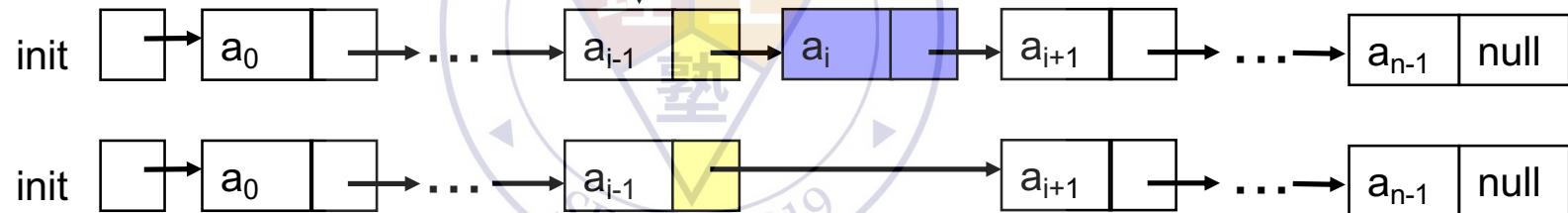
発展

配列の場合

最悪/平均
時間計算量は $\Theta(n)$

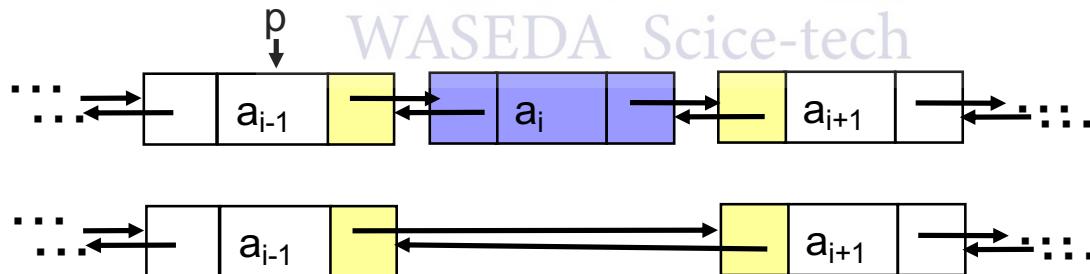


連結リストの場合



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)

FIND(i, L) : リスト L (要素数 n)の i 番目のセルの内容を返す

発展

LAST(L) : リスト L (要素数 n)の最後のセルの位置を返す

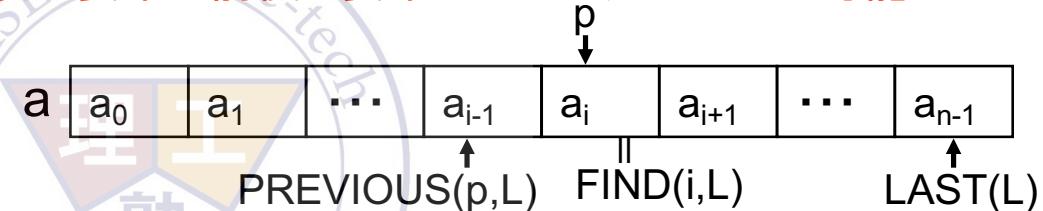
PREVIOUS(p, L) : リスト L (要素数 n)において、位置 p の1つ前のセルの位置を返す

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

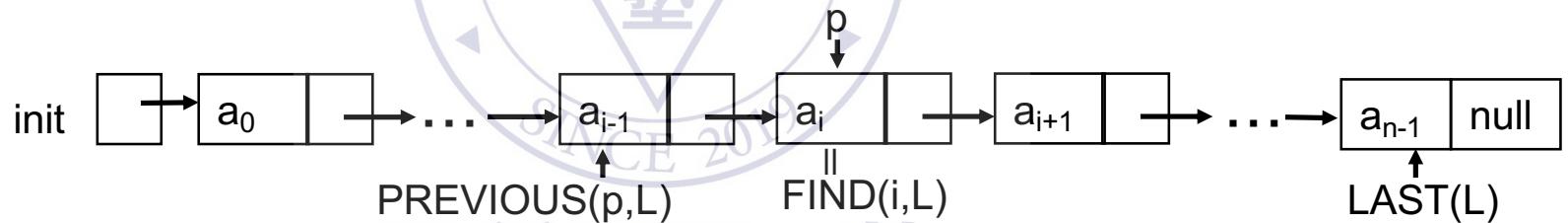
FIND(i, L) : $\Theta(1)$

LAST(L) : $\Theta(1)$

PREVIOUS(p, L) : $\Theta(1)$

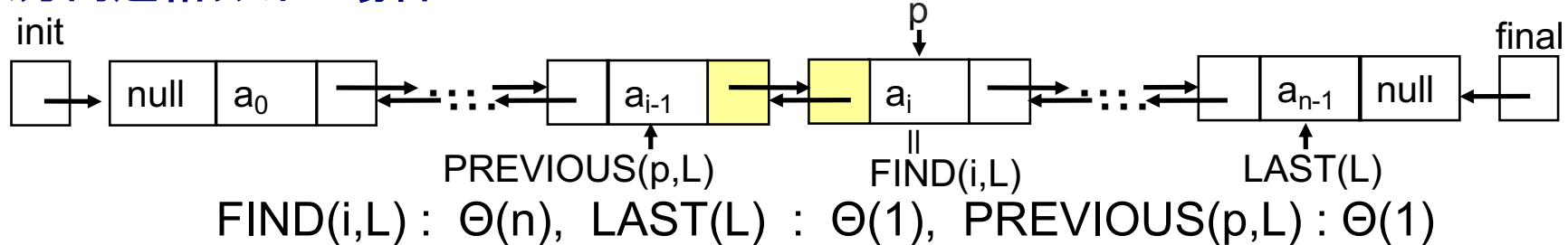


連結リスト
の場合



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) : $\Theta(1)$

C言語によるリストの定義(1/3)

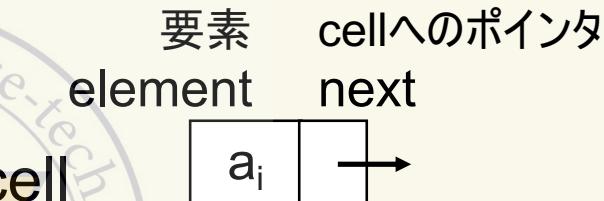
演習を受ける人へ: これから先の連結リストのC言語による実装は、演習でやります。演習の問題と合わせて、しっかり理解してください

連結リスト

```
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。

C言語によるリストの定義(2/3)

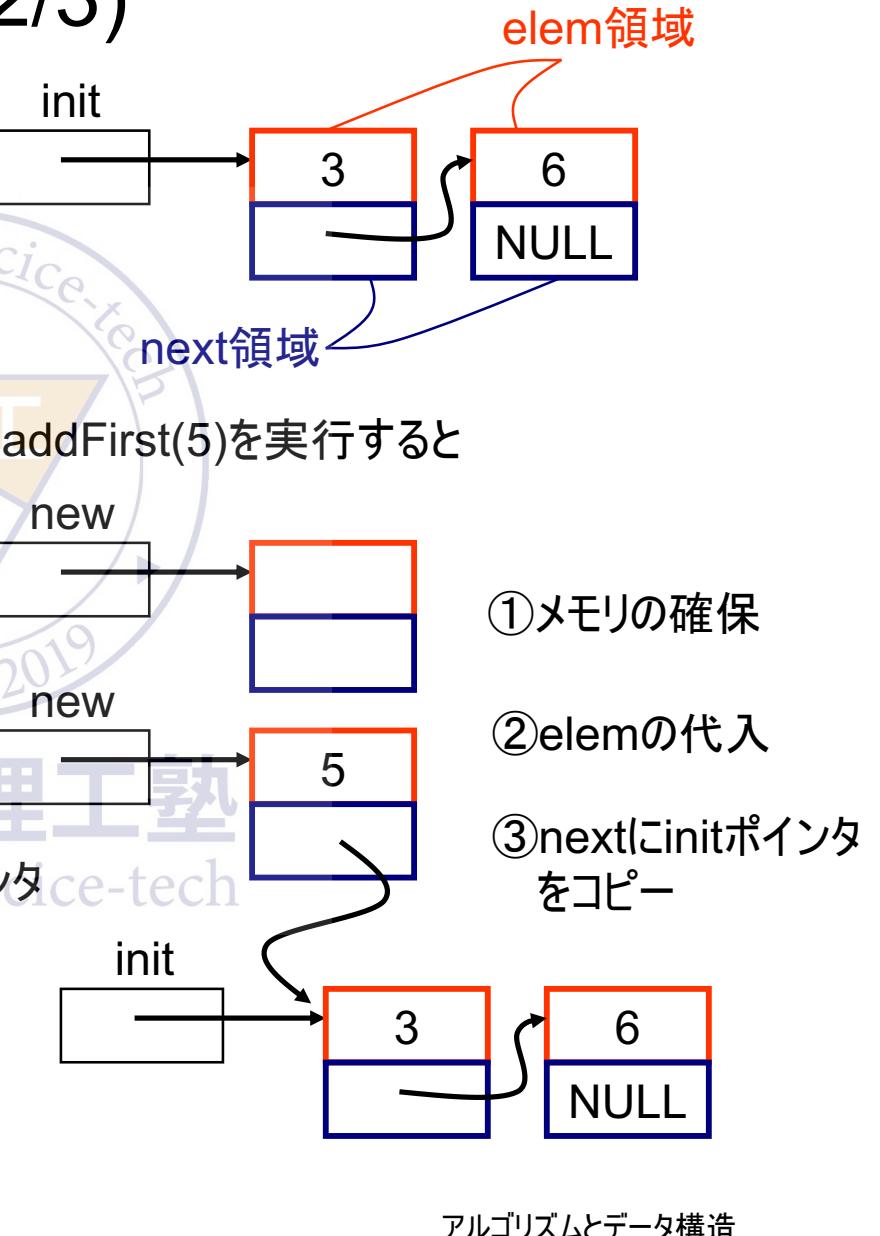
```

typedef struct cell {
    int elem;
    struct cell *next;
} cell;

cell *init=NULL;      //空のリスト

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

```



C言語によるリストの定義(3/3)：双向リスト

```
typedef struct cell {  
    int elem;  
    struct cell *prev;  
    struct cell *next;  
} cell;  
  
cell *init=NULL; //空のリスト  
cell *final=NULL;
```

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

双向連結リスト

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

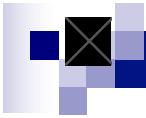
先頭に追加する場合は1つ前の
データはなし

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

注) initとfinalポインタを、head(先頭)とtail(末尾)と呼ぶことが多い。

アルゴリズムとデータ構造



特別な(制限された)リストのいろいろ
スタックとキュー 早稲田理工塾
WASEDA Scice-tech

アルゴリズムとデータ構造

スタック(stack)

スタックとは

要素の挿入、削除がいつも先頭からなされるリスト

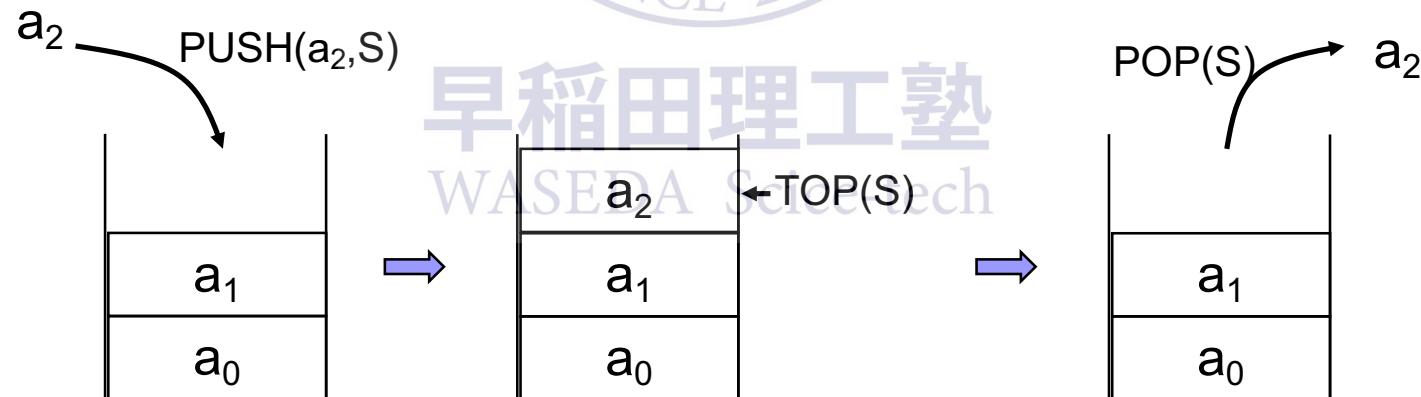
LIFO(last-in-first-out)

[基本操作]

$\text{TOP}(S)$ スタックSの先頭の位置を返す

$\text{POP}(S)$ スタックSの先頭の要素を削除

$\text{PUSH}(x, S)$ スタックSの先頭に要素xを挿入

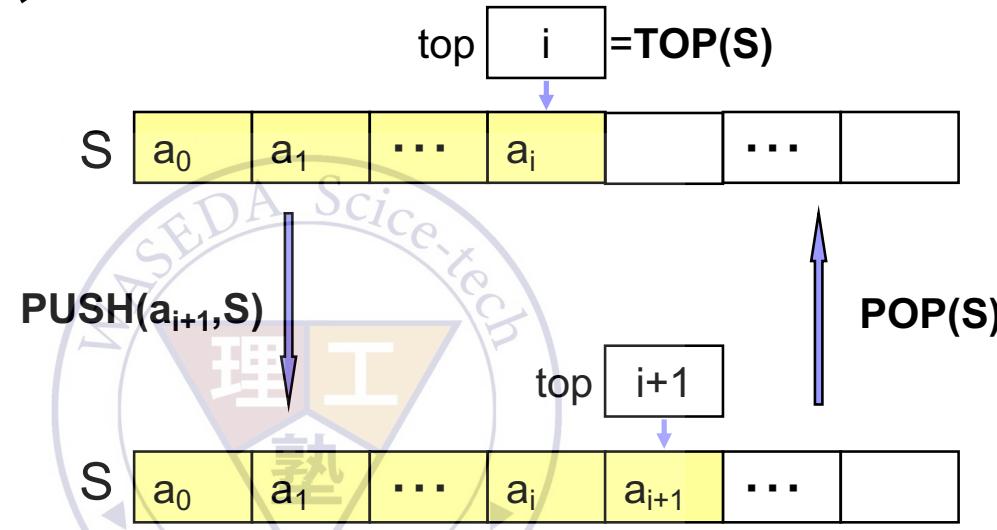


基本

スタックの実現法

配列による実現

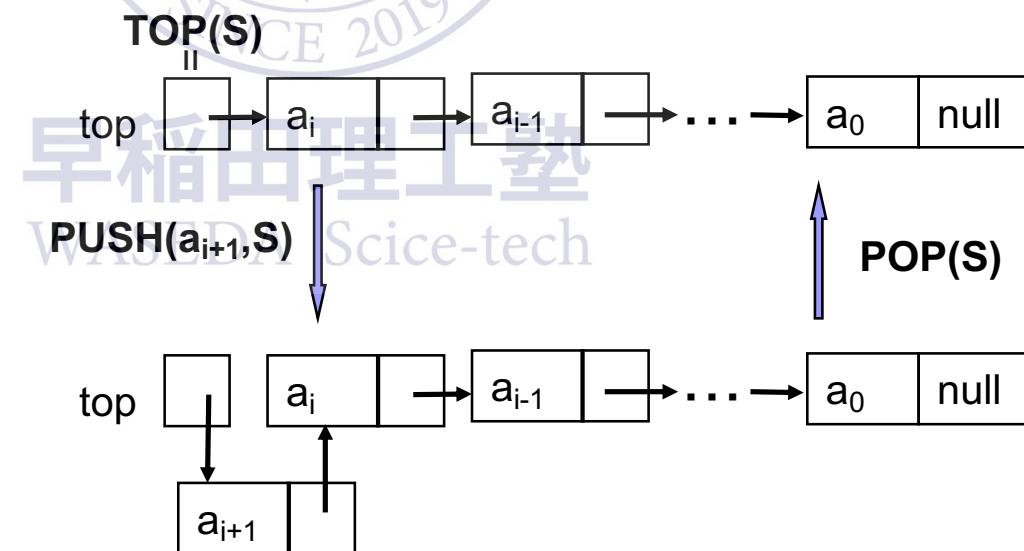
すべての操作の
時間計算量は $\Theta(1)$



連結リストによる実現

すべての操作の
時間計算量は $\Theta(1)$

発展



待ち行列(キュー)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(fast-in-fast-out)ともいう



[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

アルゴリズムとデータ構造

待ち行列(キュー, queue)

待ち行列(キュー)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

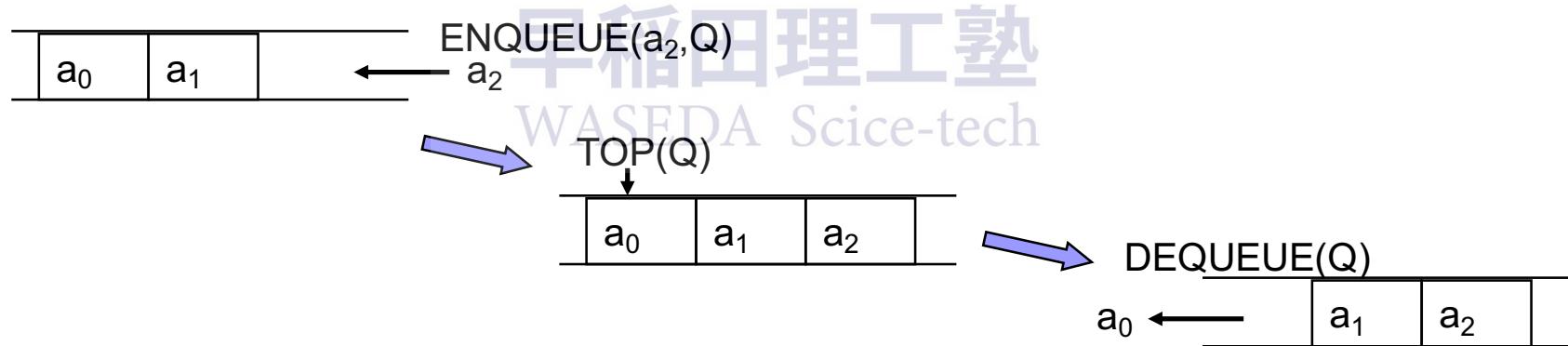
FIFO(fast-in-fast-out)

[基本操作]

$\text{TOP}(Q)$ キューQの先頭の位置を返す

$\text{ENQUEUE}(x, Q)$ 要素xをキューQの最後尾に入れる

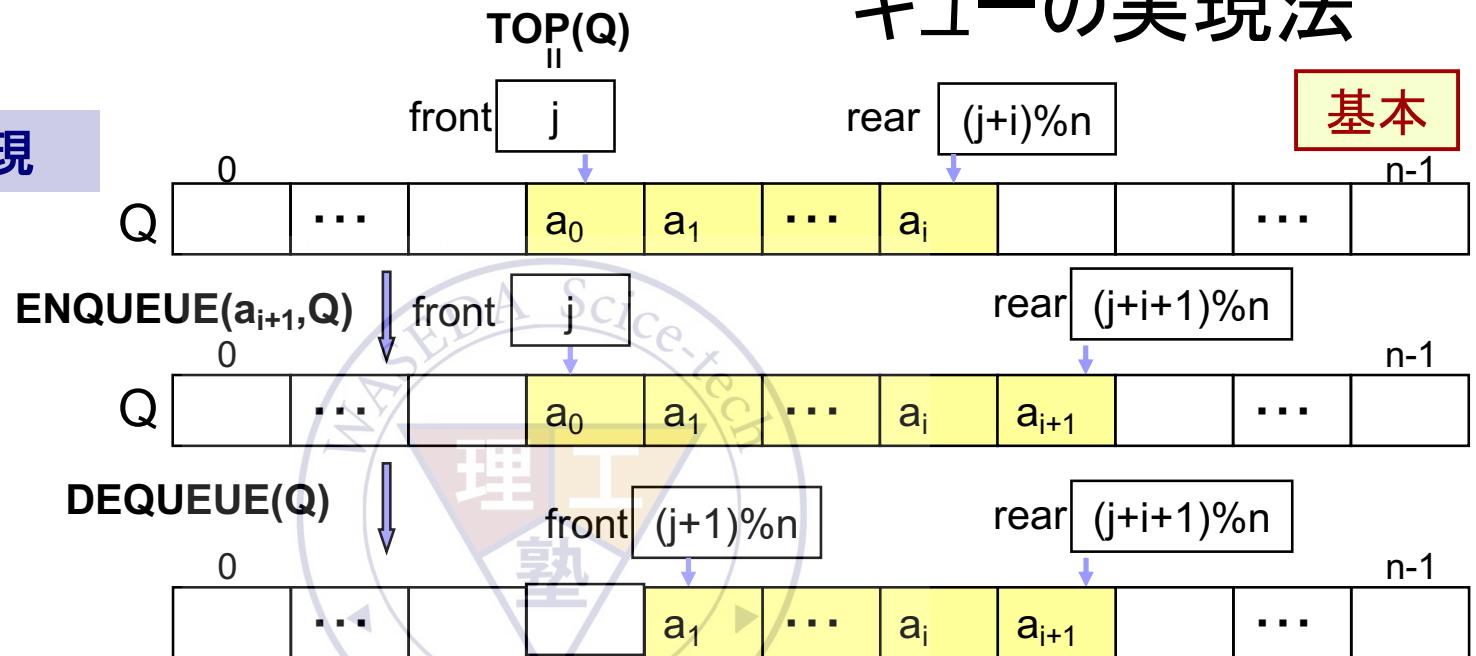
$\text{DEQUEUE}(Q)$ 先頭の要素をキューQから除く



キューの実現法

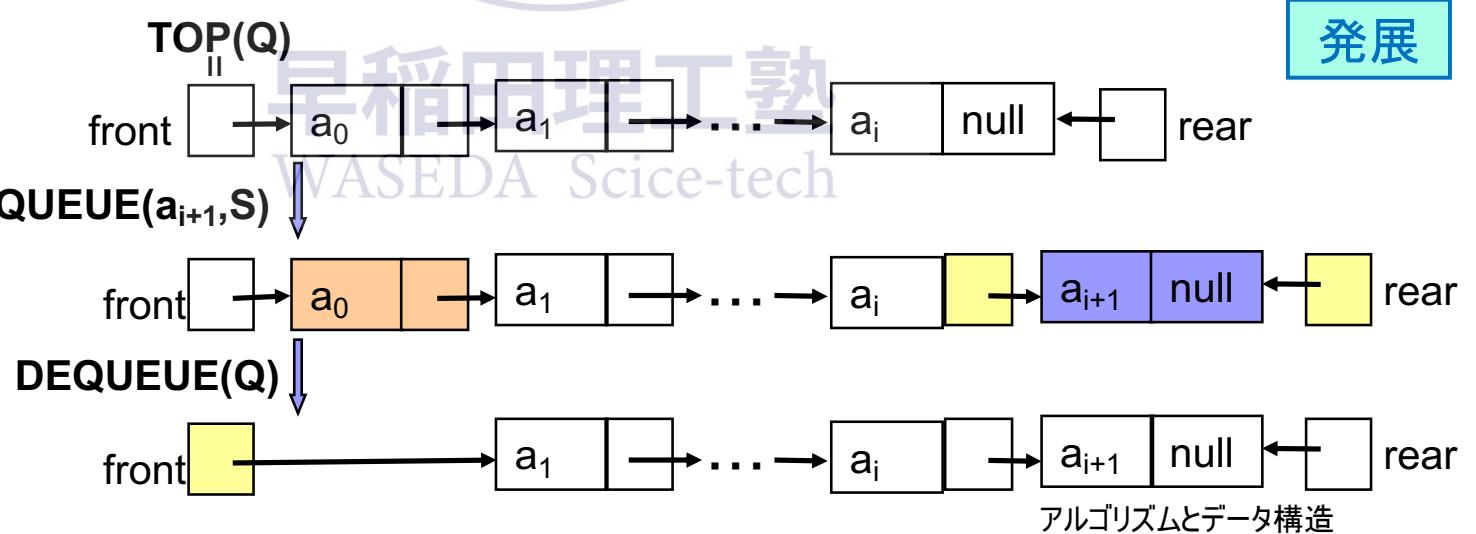
配列による実現

すべての操作の
時間計算量は
 $\Theta(1)$



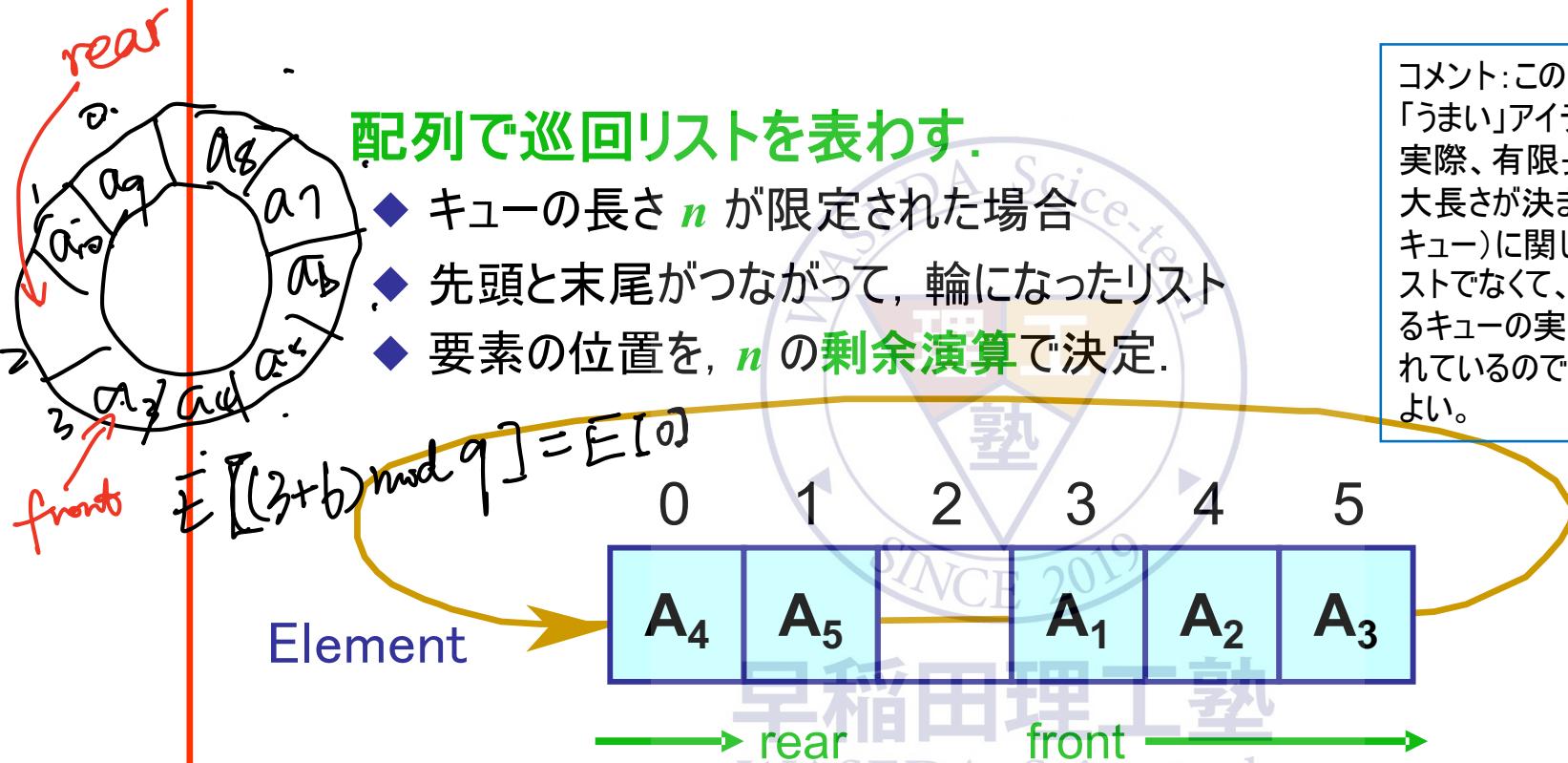
連結リスト による実現

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

基本



front から i 番目の要素 = Element[$(\text{front} + i) \bmod n$]

ex. $\text{Elem}[(\text{head} + 5) \bmod n]$ ($\text{head} = 1$ and $n = 6$)

$$= \text{Elem}[7 \bmod 6] = E[1]$$



上智大学

2023年度大学院入試問題（2023年 2月 15日実施）

理学専攻（博士前期・博士後期・前後期共通）

試験科目：専門科目（理工基礎（情報学基礎））

(a) データ構造にデータ x を入れる操作を $\text{push}(x)$ で表し、データ構造からデータを 1 つ取り出す操作を pop で表すことにする。空のスタックに対して 8 個の操作

$\text{push}(4), \text{push}(2), \text{pop}, \text{push}(3), \text{push}(1), \text{pop}, \text{push}(5), \text{pop}$

をこの順番で行ったあとで スタックに入っているデータを記せ。また、同様の操作を空のキューに対して行ったあとで キューに入っているデータを記せ。

スタック: 4 3

キュー: 1 5

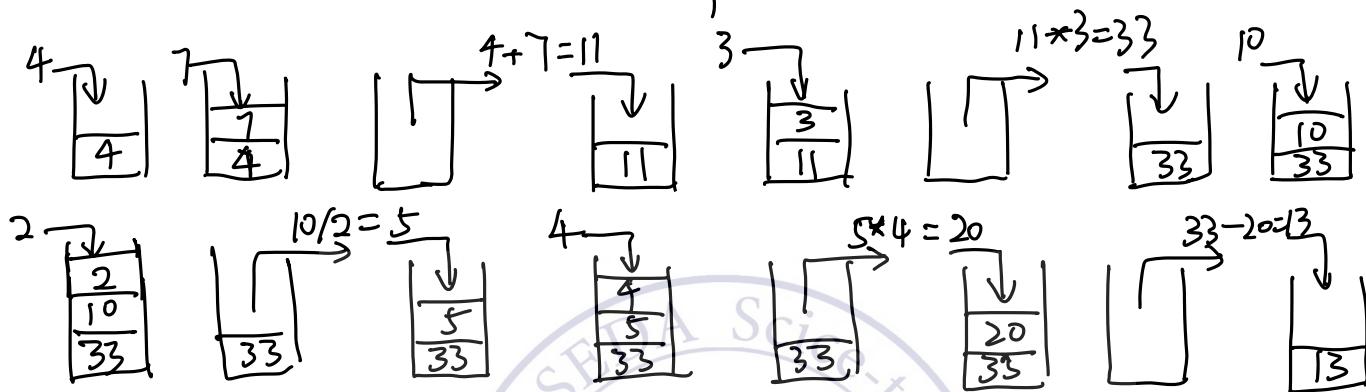
スタック → 四則運算

① 逆ポーランド記法 (後置記法)

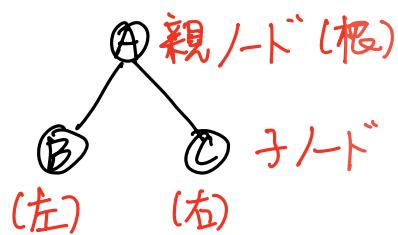
$4 \ 7 + 3 * 10 \ 2 / 4 * -$

$(4+7)*3-(10\div2)4$

数値 → push, 演算子 → pop 2 → 运算 → push(結果)



② ポーランド記法 → 逆ポーランド記法



#. 木の巡回 (traversal)

- DFS (深さ優先探索)

a. 行きがけ順 (前順) 根 → 左 → 右

A-B-C

- , +, 4, 7, *, /, 10, 2, 4

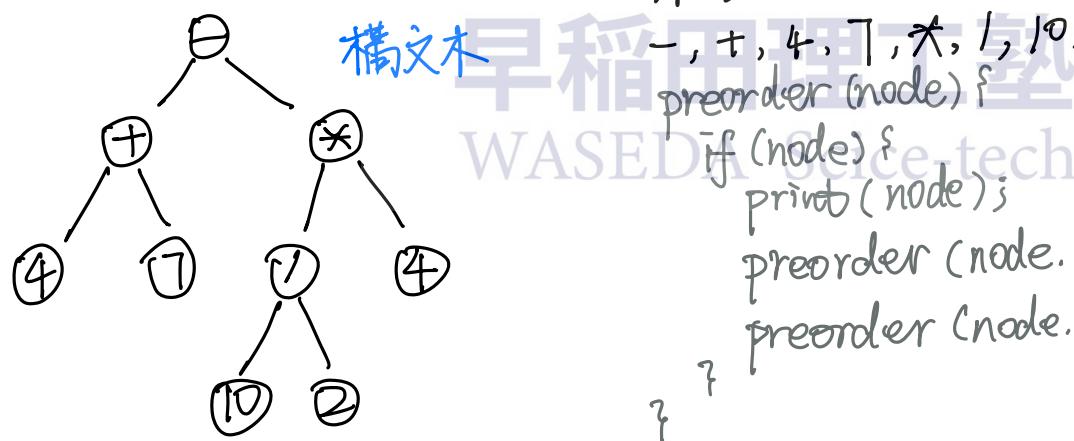
preorder (node){

if (node){

print (node);

preorder (node.left);

preorder (node.right);



b. 通りがけ順 (中順) 左 → 根 → 右

B, A, C

4, +, 7, -, 10, /, 2, *, 4

⇒ ポーランド記法: $(4+7)-(10/2)*4$

inorder (node){

if (node){

inorder (node.left);

print (node);

inorder (node.right);

c. 帰りがけ順 (後順) 左→右→根

B, C, A

4, 7, +, 10, 2, /, 1, 4, *, -

⇒逆ポーランド記法: 4 7 + 10 2 / 1 4 * - ?

- BFS (広さ優先探索) 上→下, 左→右

A, B, C

- , +, * 4, 7, 1, 4, 10, 2

```
postorder (node) {  
    if (node) {  
        postorder (node.left);  
        postorder (node.right);  
        print (node);  
    }  
}
```

例題:

$$(4+7)*3 - (10/2)*4 - \dots$$

1. 構文木: 通りがけ順によって作成

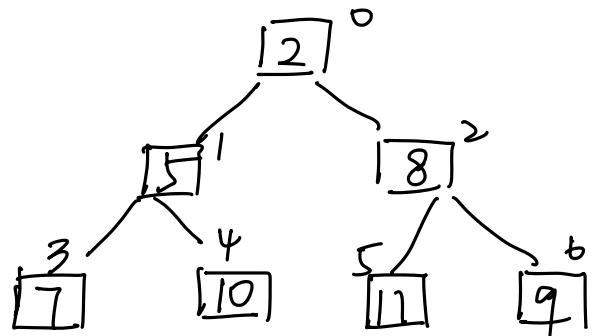
2. 後置記法: 帰りがけ順によつて得る

3. スタックにより、計算の過程を

4. 2の結果を得た抜き出しコード:

#. ヒープ heap 現在

0	1	2	3	4	5	6
2	5	8	7	10	11	9



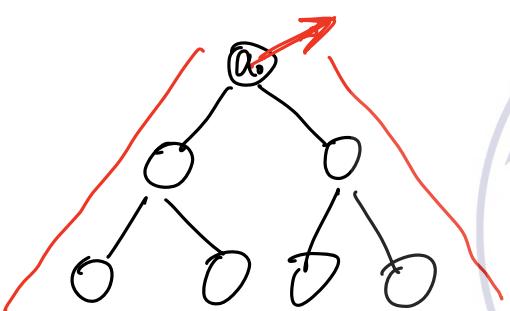
Max Heap: 親ノード \geq 子ノード

Min Heap: 親ノード \leq 子ノード

図1 Min Heap

layer: 3, height = 2

・要素の削除



step: ① 根を取り出す $x = a_0$

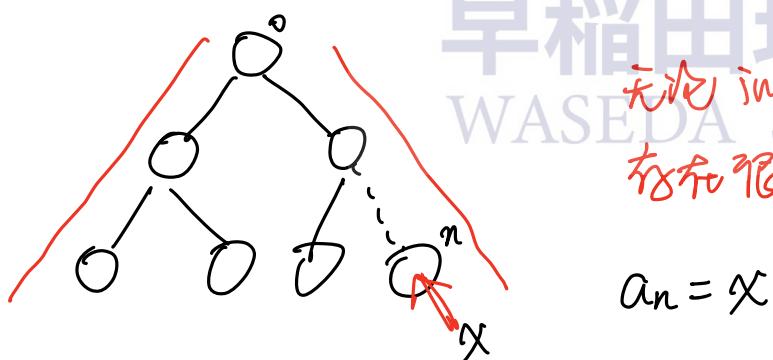
② 最後の葉ノードを根に移動

③ Heap構造に整列:

$$\min = \min\{right, left\}$$

swap(v , \min)

・要素の挿入



早稲田理工塾

WASEDA Scice-tech

无论 insert, 还是 delete, heap 和 queue
都有很大的共性.

$$a_n = x$$

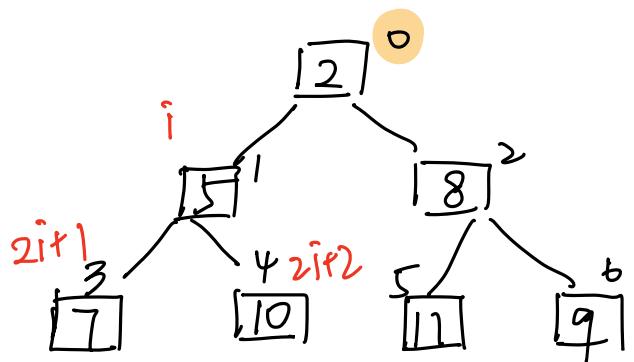
補充: swap(a, b) {

$$\text{temp} = a;$$

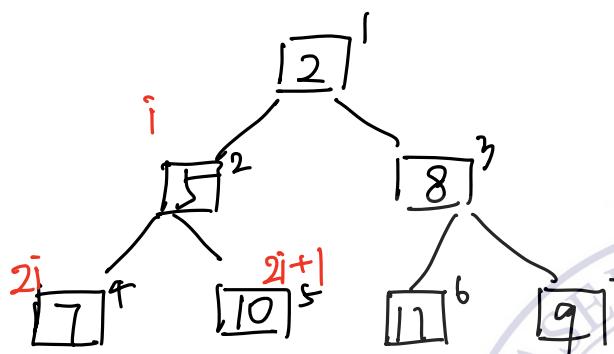
$$a = b;$$

$$b = \text{temp};$$

}



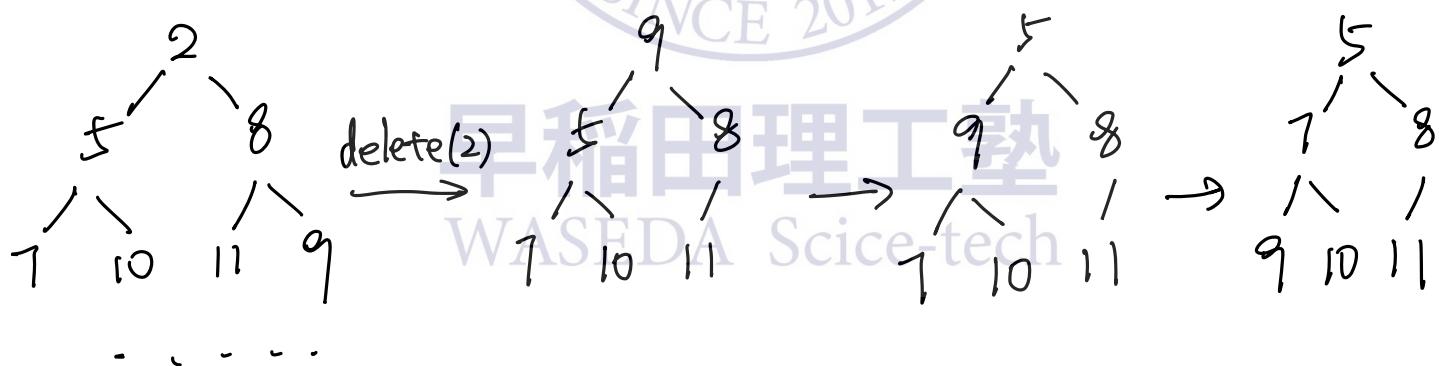
親ノード: i
子ノード: $2i+1$ (左) $2i+2$ (右)



親ノード: i
子ノード: $2i$ (左), $2i+1$ (右)

作業:

1. 図1. ノード依次削除過程



2. 葦木数 N , 木の高さ H , 葦木ノード数 N_{leaf}

① 用 N 表示 H

② 用 N 表示 N_{leaf}

③ 用 H 表示 N (范围)

