情報構造 第八回

スタック／キュー

今日の予定

- 基本データ構造
  - スタック
  - キュー（待ち行列）
- 有向グラフの探索法
  - 深さ優先探索（スタックで実現）
  - 幅優先探索（キューで実現）

スタック・キュー

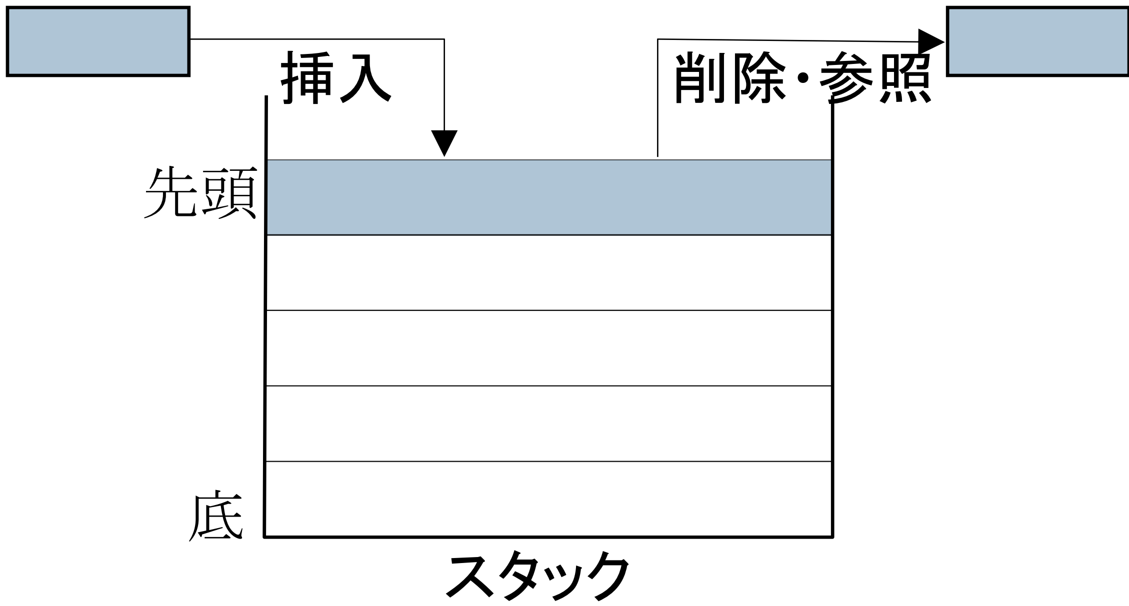
データ構造：抽象データ型

- 抽象データ型
  - 基本データ構造
    - リスト，スタック，キュー（待ち行列），順序木，2分木，集合，辞書
  - 高度なデータ構造
    - 2分探索木，AVL木，平衡木
- 仕様
  - 要素や構造を記述
  - 操作は「事前条件ー事後条件」で提示
- 実現
  - 既定義のデータ構造で定義：C言語などで既に定義されているデータ型
    - 配列，構造体，レコード，ポインタなど

スタック

スタック

- スタック：
  - 後入れ先出し: LIFO (last-in-first-out)
    - stack, pushdown list
- 要素は同じ型
- 要素数は有限
- 要素は一列に並び
- 要素の挿入，削除，参照
  - 先頭と呼ばれる要素に対してのみ



スタックの仕様

- **要素**：要素は同じ型を持つ
- **構造**：要素どうしの関係は線形
- **操作**：先頭とよばれる特定の要素の位置に着目し，先頭の位置に対してだけ，要素の**挿入**・**削除**・**参照**がおこなえる
- 要素数が0のものを空のスタックと呼ぶ

スタックの操作

- スタック型 Stack, 要素型 Element, スタック変数 S, 要素型データ e, 要素型変数 v

- +int PushDown(Stack *S, Element e)
  - **Post**: データ e をスタック *S の先頭に入れる．挿入された要素が先頭．関数値は真 (1) に
- +int PopUp(Stack *S)
  - **Pre**: スタック *S が空でない
  - **Post**: スタック *S の先頭要素を削除．関数値は真 (1) に
- +int Retrieve(Stack *S, Element *v)
  - **Pre**: スタックが空でない
  - **Post**: スタック *S の先頭要素の値が 変数*v に設定．関数値は真 (1) に
- int Empty(Stack *S)
  - **Post**: 関数値は，スタック *Sが空の時真 (1)，さもなければ偽 (0)
- void Create(Stack *S)
  - **Post**: *Sは空スタック

スタックが**空**のとき、
関数値は**偽**(**0**)となり、
実行前と同じ状態のまま

使用例：簡単な行編集プログラム

- 入力文字列を出力：#で直前の入力を削除，＠ですべての入力を削除

```
typedef char Element;
void Edit(){
    Stack S, T;  Element e;  char c;
    Create(&S);
    do{
        scanf("%c", &c);    /* 1文字入力 */
        if(c == '#'){ if(!PopUp(&S, c)){ ERROR("PopUp Error"); return; } }  /* 1文字削除 */
        else if (c == '@'){ Create(&S); } /* すべての文字を消去 */
        else { if(!PushDown(&S, c)){ ERROR("PushDown Error");  return; } } /* 積む */
    } while( c != '¥n' );  /* 入力文字が改行できない限り繰り返す */
    Create( &T );  /* SをTに積み直す */
    while(!Empty(&S)){ PushDown( &T, (Retrieve(&S, &e), e )); PopUp(&S); }
    while(!Empty(&T)){
        printf("%c", (Retrieve(&T, &e), (char)e));
        PopUp(&T);}}    /* Tを印刷 */
```

```
Retrieve(&S, &e);
PushDown(&T, e);
```

対話的に文字列を入力	**xyz@aa#bcstu###cd 改行**
つぎの文字列が出力	**abccd** 改行

使用例：整数算術式の計算

- **演算子**：+, -, *, / (中置2項演算子)
  - [+ と -], [* と /] は同じ優先順位．すべて左結合性をもつ
- **括弧**：( ) 内の式を優先
- **被演算子**：十進数数字列
- **記号**⊥（垂直記号，Up Tack）：算術式の右端，演算子スタックの底
  - ⊥はすべての演算子より低い優先順位を持つとする
- **入力**：整数算術式⊥
  - 算術式は入力バッファ上にある
  - 入力ポインタの指す記号を赤で示す
- **出力**：算術式を評価した結果の整数値（被演算子スタック上）
- **道具**：演算子用スタック，被演算子用スタック

算術式を、演算子用スタックで、逆ポーランド記法に**変換**し、被演算子用スタックで、式を**評価**

	初期状態	最終状態
入力バッファ	2＊14＋5＊3⊥	2＊14＋5＊3⊥
演算子スタック	⊥	⊥
被演算子スタック		43

逆ポーランド記法

• 中値記法：	2+3	(1+2)*(3-4)
• ポーランド記法（前置記法）：	+23	*+12-34
• 逆ポーランド記法（後置記法）：	23+	12+34-*

- ポーランド記法は演算子が関数適用のようである
- 逆ポーランド記法は式の計算が最初の要素を取り出して行える
  - => スタック

アルゴリズムの原理：入力が数字または演算子

入力バッファ　２＊１４＋５＊３⊥

演算子スタック　⊥＊

被演算子スタック　２ ←14

入力が数字列

数字列を整数値に変換して、
被演算子スタックにプッシュダウン
入力ポインタを進める。

２＊１４＋５＊３⊥

優先順位の高さ

⊥＊ ·＞ ＋

2 14 ← 28

⊥ ← ＋

28

入力が順位の低い演算子＋
演算子＊をポップアップ、
被演算子2,14もポップアップ。
計算2*14
結果28をプッシュダウン。
再び、＋と演算子スタックの**先頭との比較**

２＊１４＋５＊３⊥

⊥＋ ＜· ＊

28 5

入力が順位の高い演算子＊
＊の右被演算子が未入力より、
演算子＊をプッシュダウン。
入力ポインタを進める。

アルゴリズムの原理：入力が左括弧

９＋２＊(５−３)⊥

⊥＋＊ ← (

９ ２

入力が左括弧

括弧内の式を先に計算する準備のため、
内側の式の明示の'('をプッシュダウン。
入力ポインタを進める。

９＋２＊(５−3)⊥

⊥＋＊ (← −

９ ２ ５

左括弧と**演算子**

演算子をプッシュダウンし、
括弧内の式を先に計算する。

アルゴリズムの原理：入力が右括弧

９＋２＊（5−3）⊥

⊥＋＊（ −　　）

９ ２ 5 3　2 =5−3

入力が右括弧

括弧内の式の計算

⇒演算子−をポップアップ、

被演算子5,3もポップアップ。

計算5−3

結果2をプッシュダウン。

演算子スタックの先頭が

演算子である限り続ける。

（ 2 ）

９＋２＊（5−3）⊥

⊥＋＊（　　）

９ ２ ２

入力が右括弧(つづき)

スタックの先頭が**左括弧(**

⇒括弧内の計算を終了

左括弧(をポップアップ。

入力ポインタを進める。

アルゴリズムの原理：入力が⊥

９＋２＊（５−３）⊥

⊥＋＊ ・＞ ⊥

９ ２ ２ 4

⊥＋ ・＞ ⊥

９４ 13

入力が⊥

スタックの先頭が演算子
⇒演算子＊をポップアップ、
被演算子2,2もポップアップ。
計算2*2
結果4をプッシュダウン。
演算子スタックの先頭が
演算子である限り続ける。

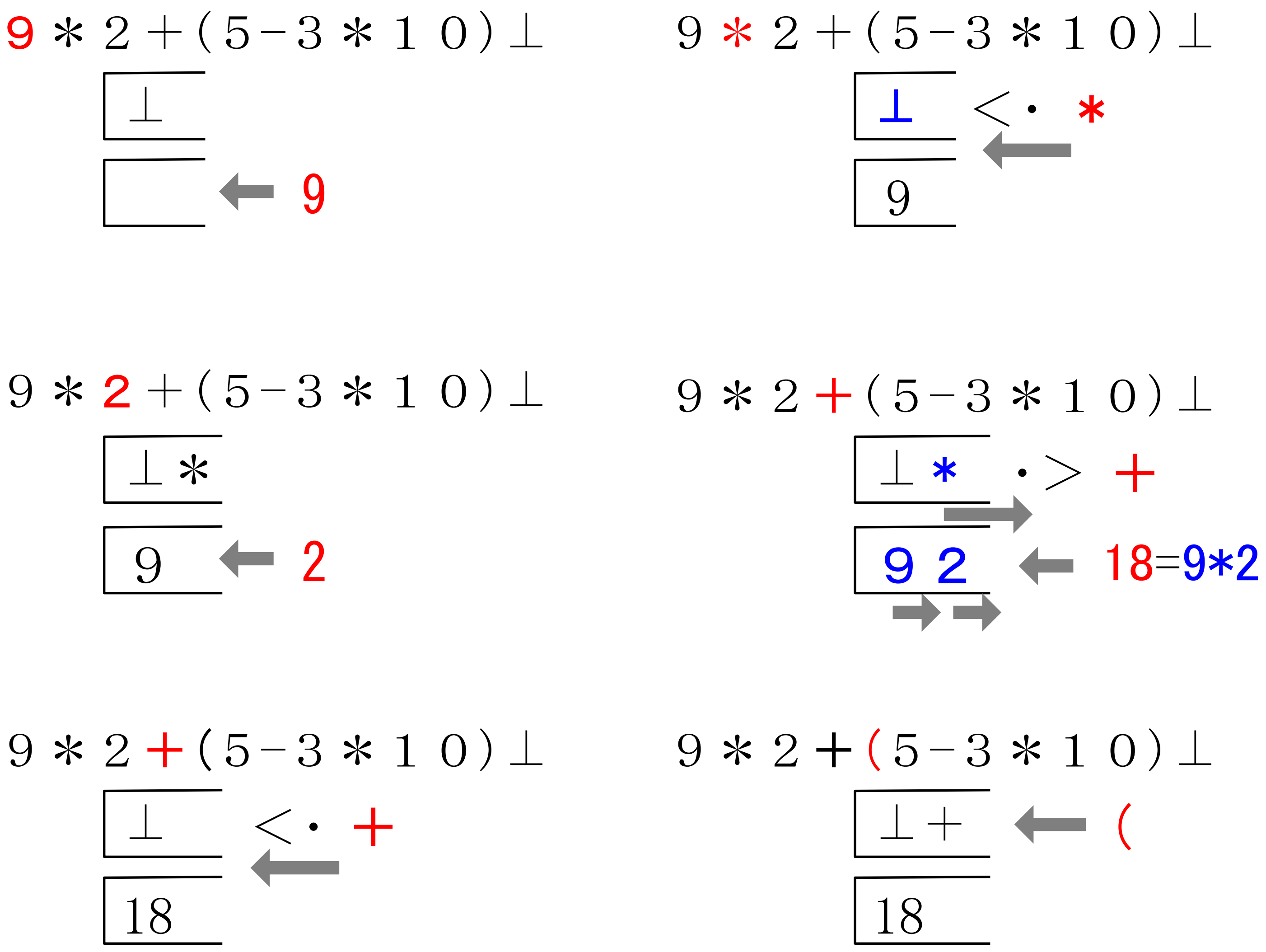
９＋２＊（５−３）⊥

⊥ ⊥

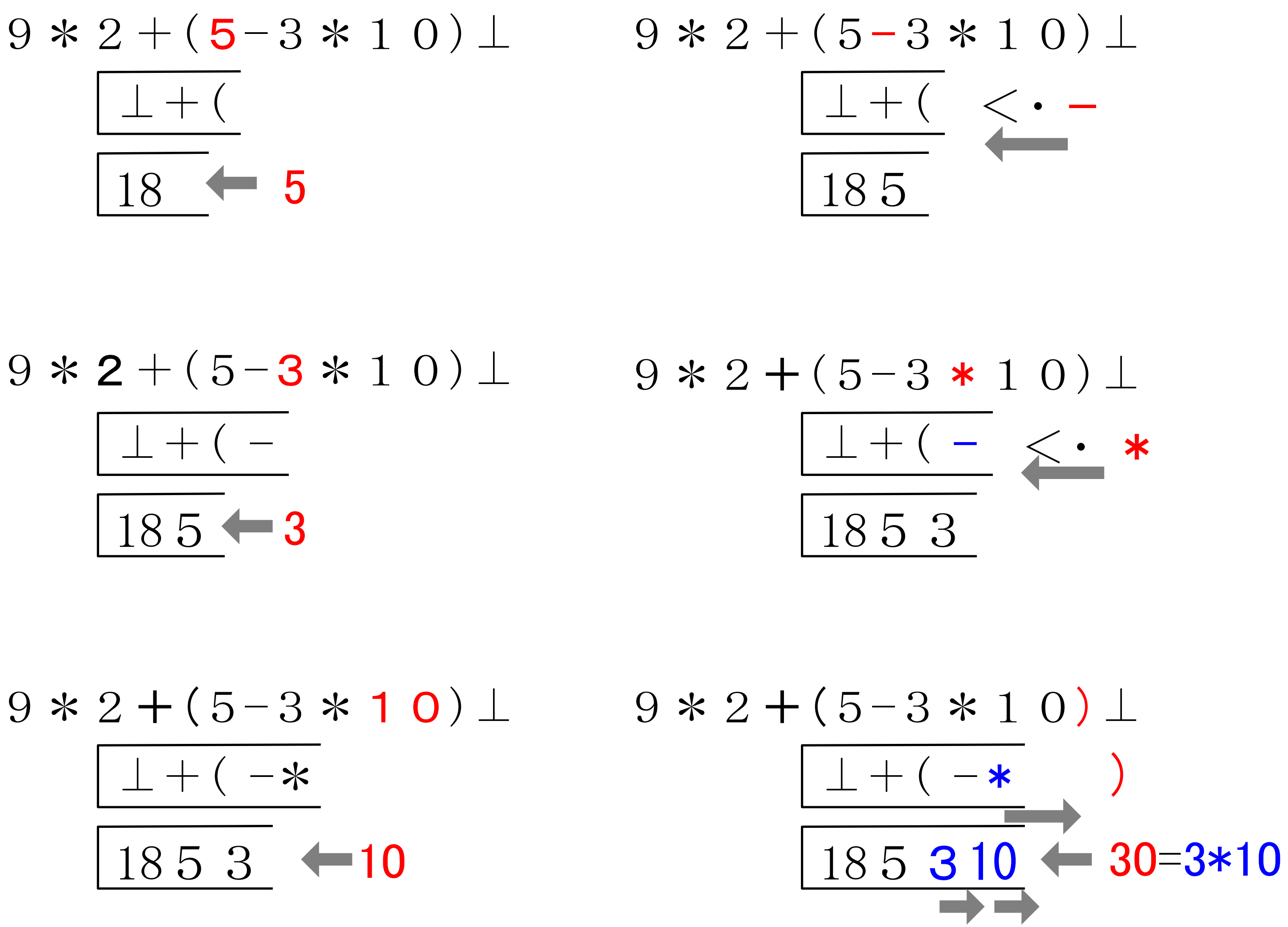
13 答え

入力が⊥
スタックの先頭も⊥
⇒計算が終了

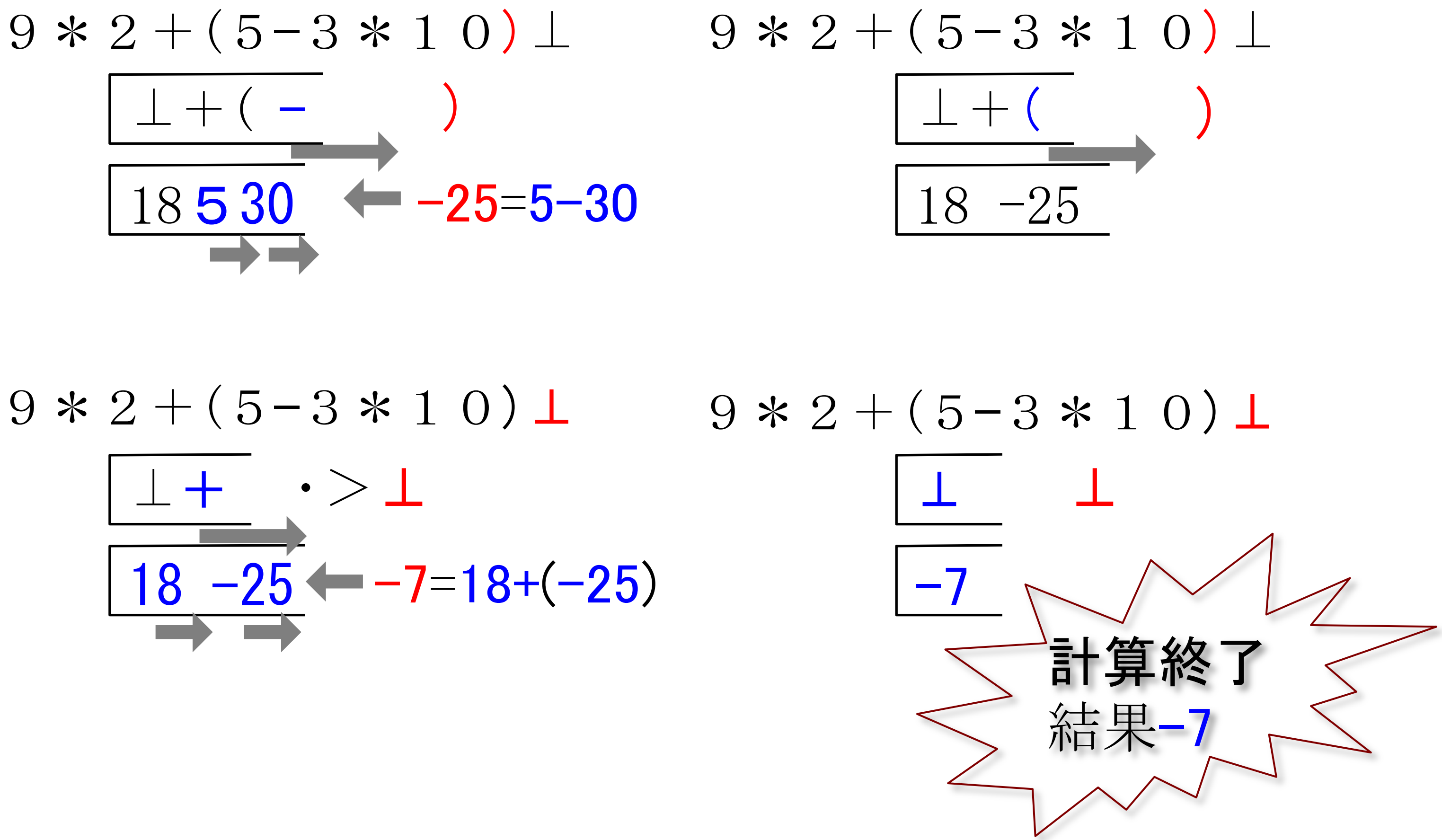
9＊2＋（5－3＊10）⊥ の計算 すべての処理



9＊2＋（5－3＊10）⊥ の計算 すべての処理



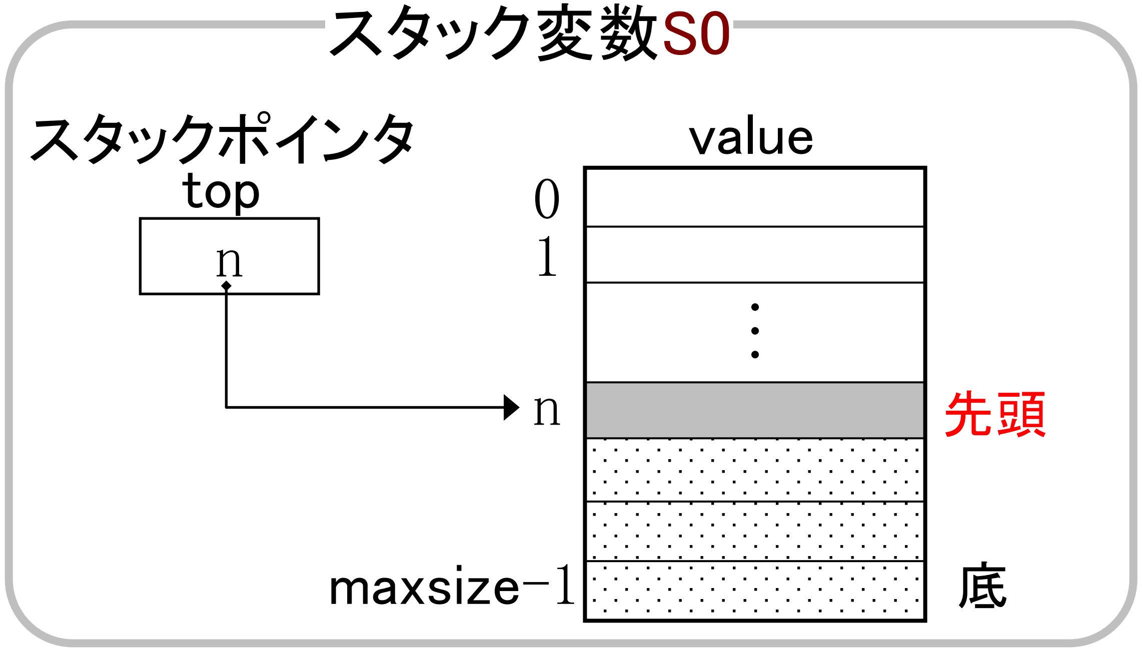
９＊２＋（５－３＊10）⊥ の計算 すべての処理



スタックの配列による実現

- 表現

```
#difine maxsize 100
typedef struct{
    int top;
    Element value[maxsize];
}Stack;
Stack S0;
```



実現アルゴリズム

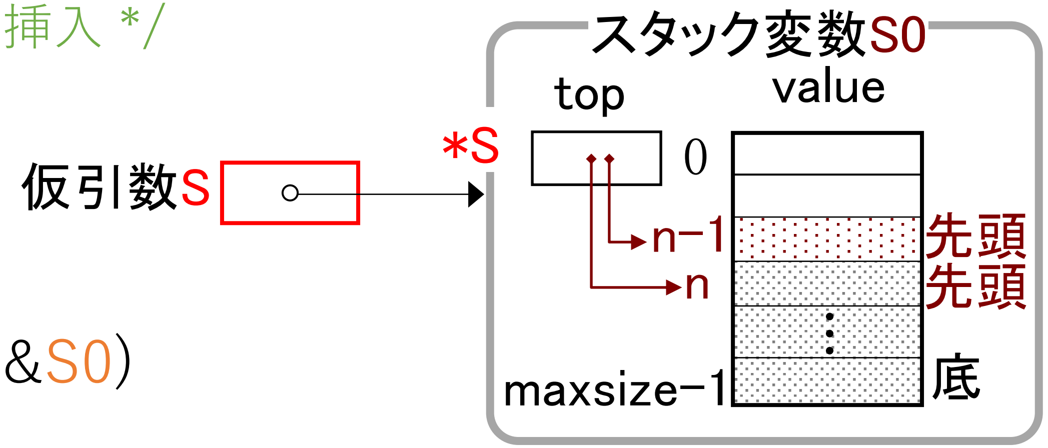
- 実現の制約：スタックの大きさが配列長で制限される
- maxsize超えのデータ数の挿入で，実現に基づくエラーになる

- データ e0 をスタックS0 の先頭にプッシュダウン PushDown(&S0, e0)
- スタック S0 の先頭ををポップアップ PopUp(&S0)
- スタックS0の先頭要素を変数v0に読み出すRetrieve(&S0, &v0)
- スタックS0を空にするCreate(&S0)
- スタックS0が空か？ Empty(&S0)

実現アルゴリズム PushDown/PopUp

- データ e0 をスタックS0 の先頭にプッシュダウン PushDown(&S0, e0)

```
int PushDown(Stack *S, Element e){
    if( S->top == 0) return 0;            /* スタックが一杯：実現の制約*/
    else{
        S->top = S->top -1;               /* eを先頭に挿入 */
        S->value[S->top] = e; return 1;  }
}
```



- スタック S0 の先頭ををポップアップ PopUp(&S0)

```
int PopUp(Stack *S){
    if(Empty(S)) return 0;                /* スタックが空：事前条件に反する */
    else{
        S->top = S->top + 1; return 1} }          /* 先頭要素を削除 */
}
```

実現アルゴリズム Retrive/Create/Empty

- スタックS0の先頭要素を変数v0に読み出すRetrieve(&S0, &v0)

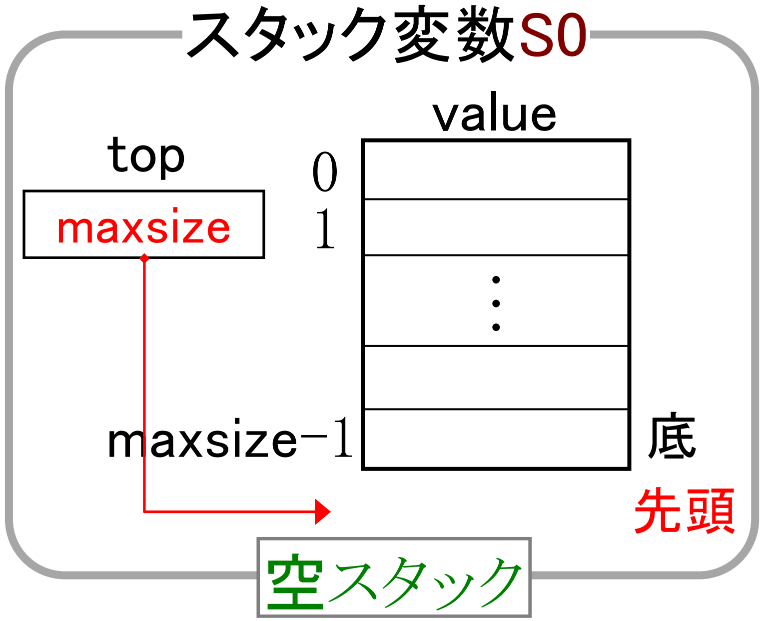
```
int Retrieve(Stack *S, Element *v){
    if(Empty(S)) return 0;             /* スタックが空 */
    else{ *v = S->value[S->top];  return 1; } }  /* 先頭の要素の読出 */
```

- スタックS0を空にするCreate(&S0)

```
void Create(Stack *S){
    S->top = maxsize;      /*空のスタック*/
```

- スタックS0が空か？ Empty(&S0)

```
int Empty(Stack *S){
    return (S->top >= maxsize);  }
```



実現アルゴリズムの効率

- リストを配列にべた詰めで実現：×
  - 途中の要素の挿入・削除に手間がかかる
  - 末尾要素の挿入・削除は一定時間

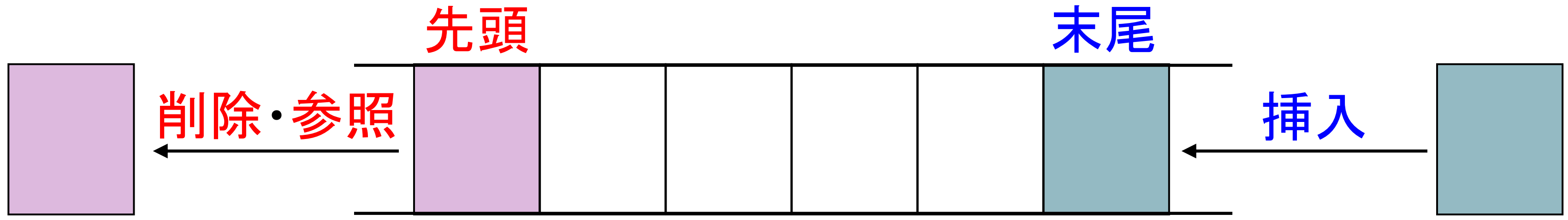
- スタックを配列にべた詰めで実現：〇
  - 挿入削除が先頭だけ
  - 途中の要素の挿入・削除は行わない
  - => 配列へのべた詰め実現が最適！

- スタックを連結リストによる実現：△
  - ポインタの領域が無駄になる

キュー（待ち行列）

キュー（待ち行列）

- キュー（待ち行列）
  - 先入れ先出し，FIFO（first-in-firast-out）
- 要素は同じ型
- 要素は有限
- 要素は一列に並び
- 要素の挿入は末尾
- 要素の削除・参照は先頭



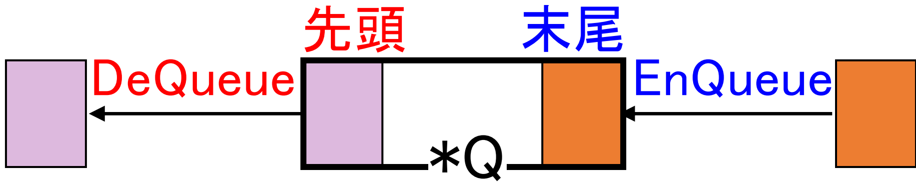
待ち行列

キューの仕様

- **要素**：要素は同じ型を持つ
- **構造**：要素どうしの関係は線形
- **操作**：先頭と末尾と呼ばれる要素の位置に着目
  - 削除と参照は先頭の要素に対して行う
  - 挿入は末尾の要素に対して行う
- 要素数が0のものを空のキューと呼ぶ

キューの操作

- キュー型 Queue, 要素型 Element,　キュー変数 Q, 要素型データe, 要素型変数 v
- +int EnQueue(Queue *Q, Element e)
  - **Post**: データ e をキュー *Q の末尾に入れる．挿入された要素が末尾．関数値は真 (1)に
- +int DeQueue(Queue *Q)
  - **Pre**: キュー *Q が空でない
  - **Post**: キュー *Q の先頭要素を削除．関数値は真 (1) に

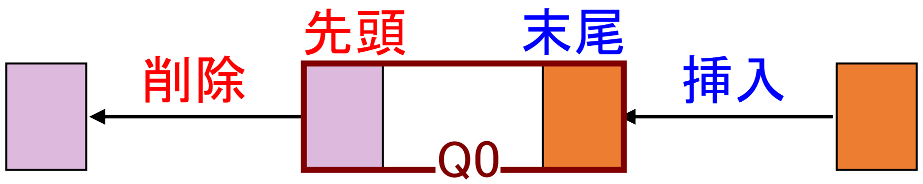


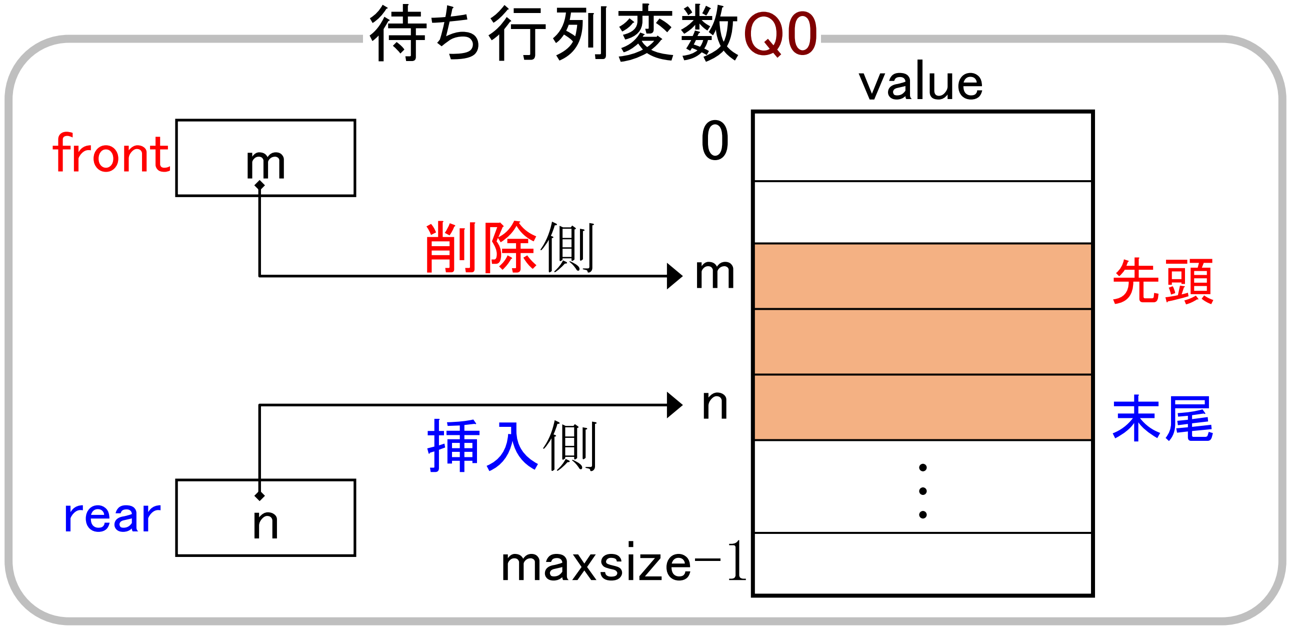
- +int Retrieve(Queue *Q, Element *v)
  - **Pre**: キュー*Qが空でない
  - **Post**: キュー *Q の先頭要素の値が 変数*v に設定．関数値は真 (1)に
- int Empty(Stack *S)
  - **Post**: 関数値は，キュー *Qが空の時真 (1)，さもなければ偽 (0)
- void Create(Stack *S)
  - **Post**: *Qは空キュー

キューの配列による実現

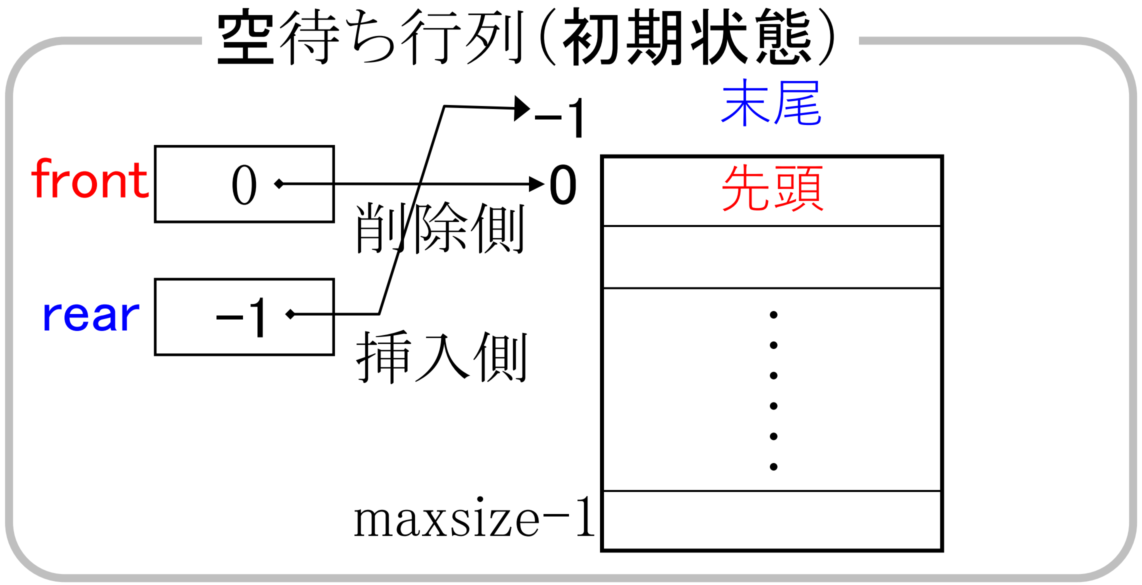
- 表現

```
#define maxsize 100
type struct{
    int front, rear;
    Element value[maxsize];
} Queue;
Queue Q0;
```

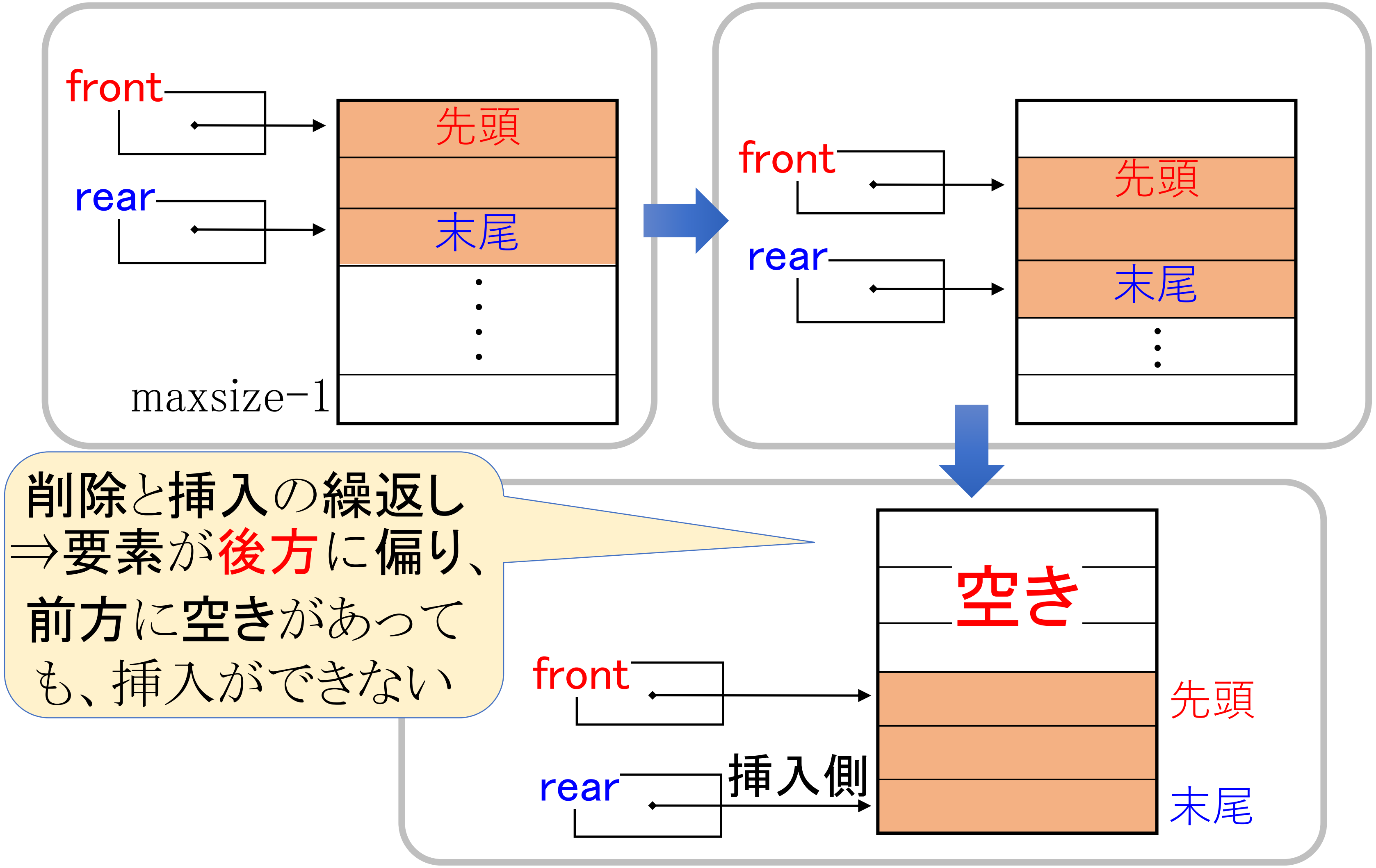




キューの配列による実現：初期状態



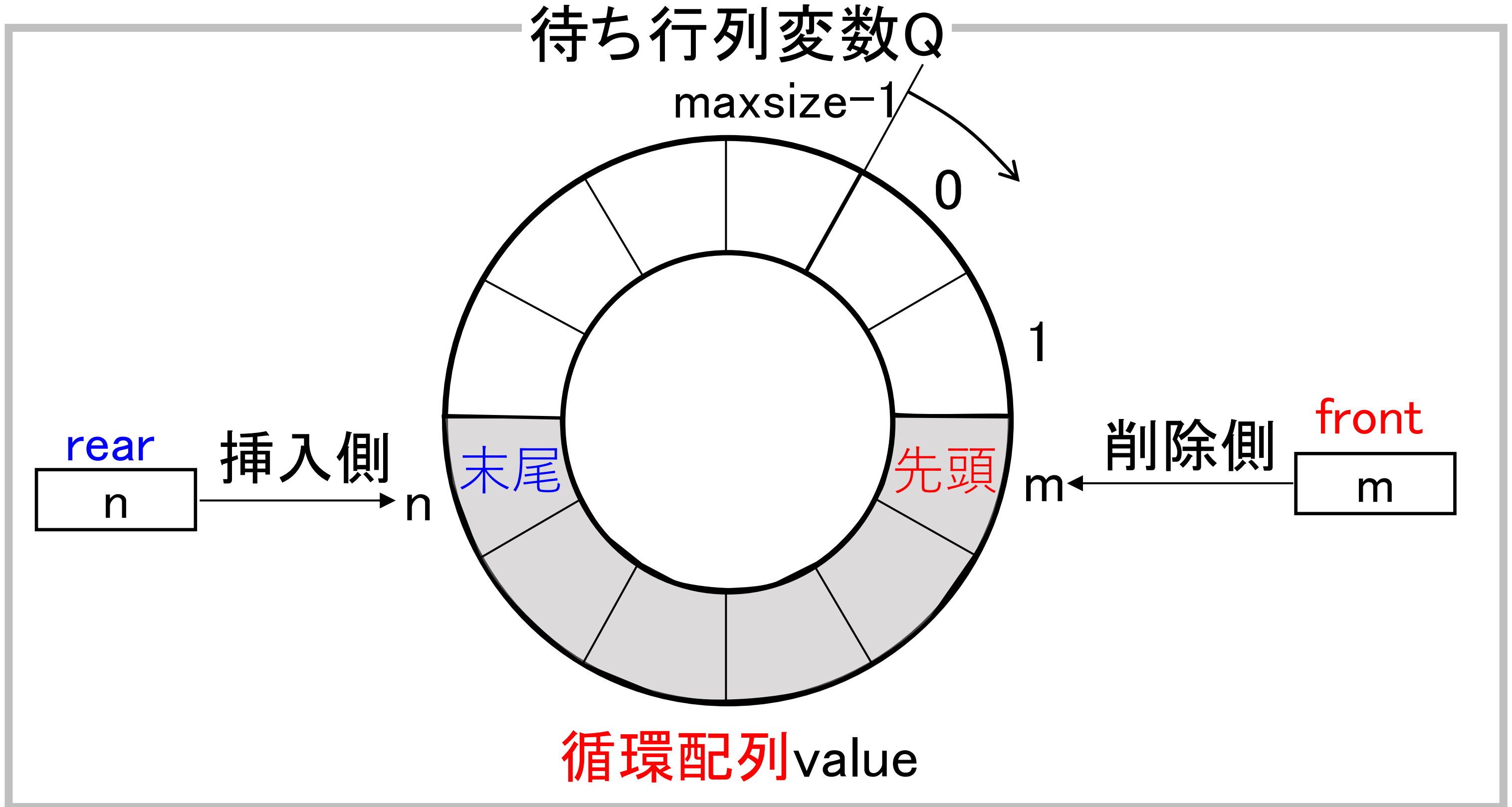
配列による実現の欠点



front
rear
先頭
末尾
maxsize−1
front
rear
先頭
末尾
削除と挿入の繰返し
⇒要素が後方に偏り、
前方に空きがあって
も、挿入ができない
空き
front
rear
挿入側
先頭
末尾

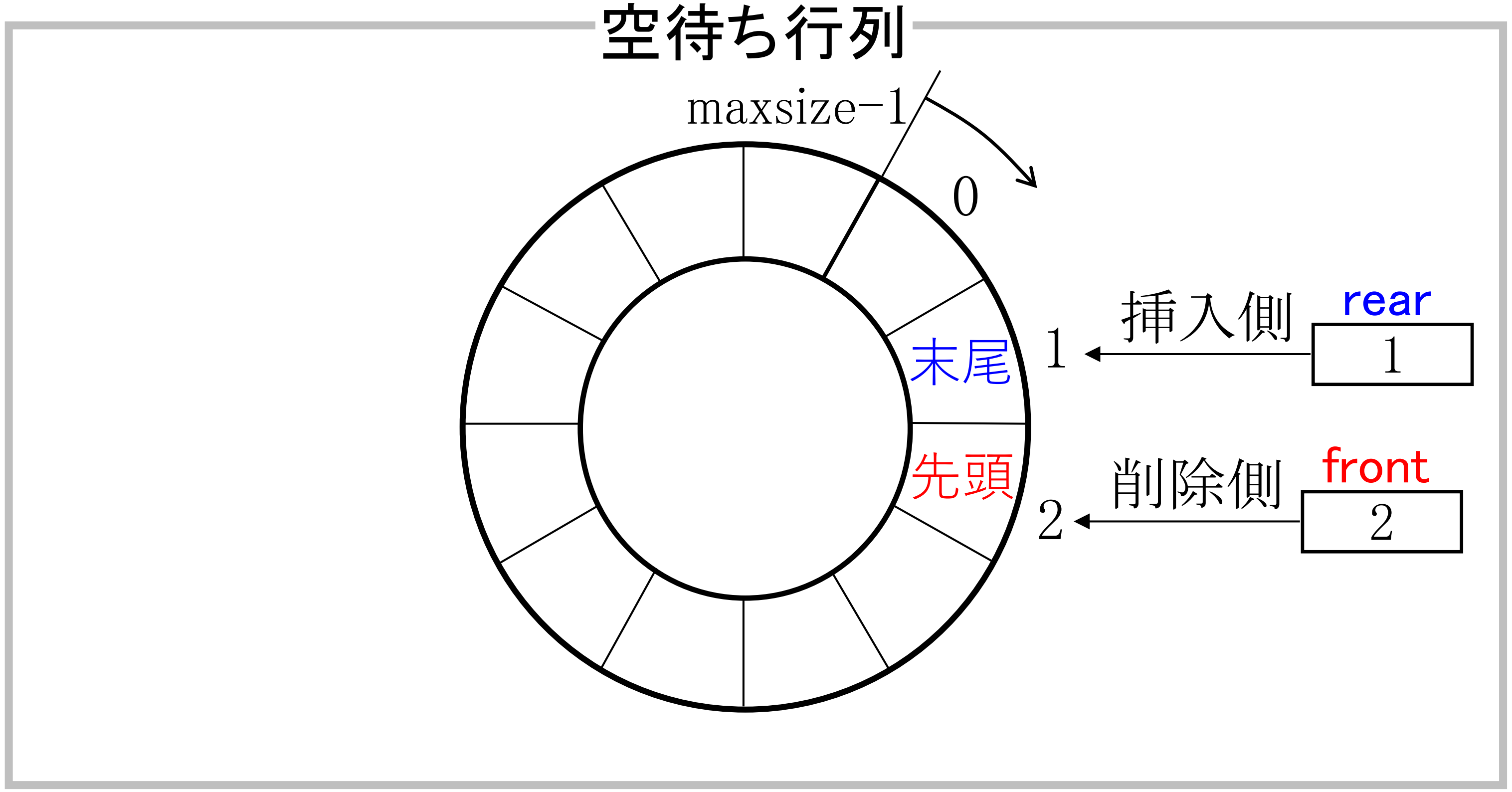
循環配列による実現

- 配列の最後と最初の位置をつなげ環状にする
  - 配列の最後の位置（maxsize-1）のとき最初の位置0から要素を挿入できる
    => 挿入し続けることができる
- 表現：前の型Queueと同じ



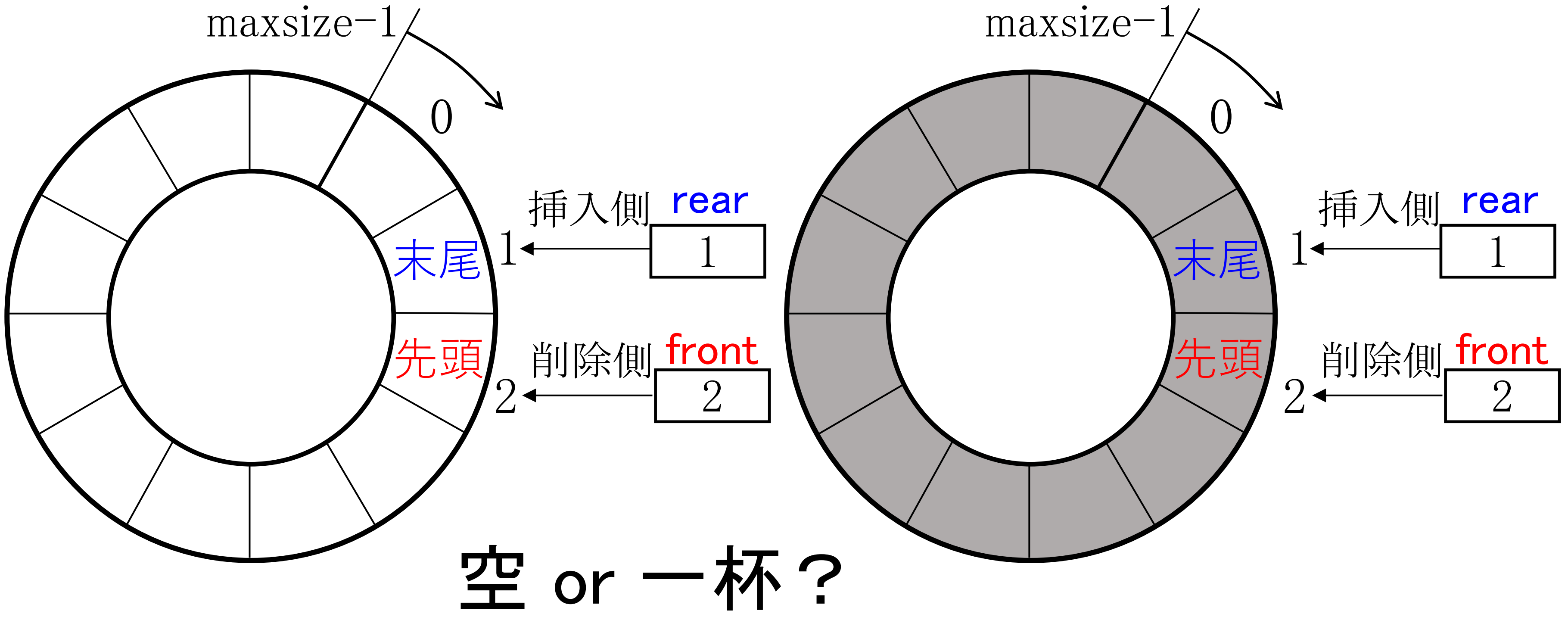
循環配列：空キュー

- Q.rearがQ.frontの１つ反対回りにある状態を空キューとする



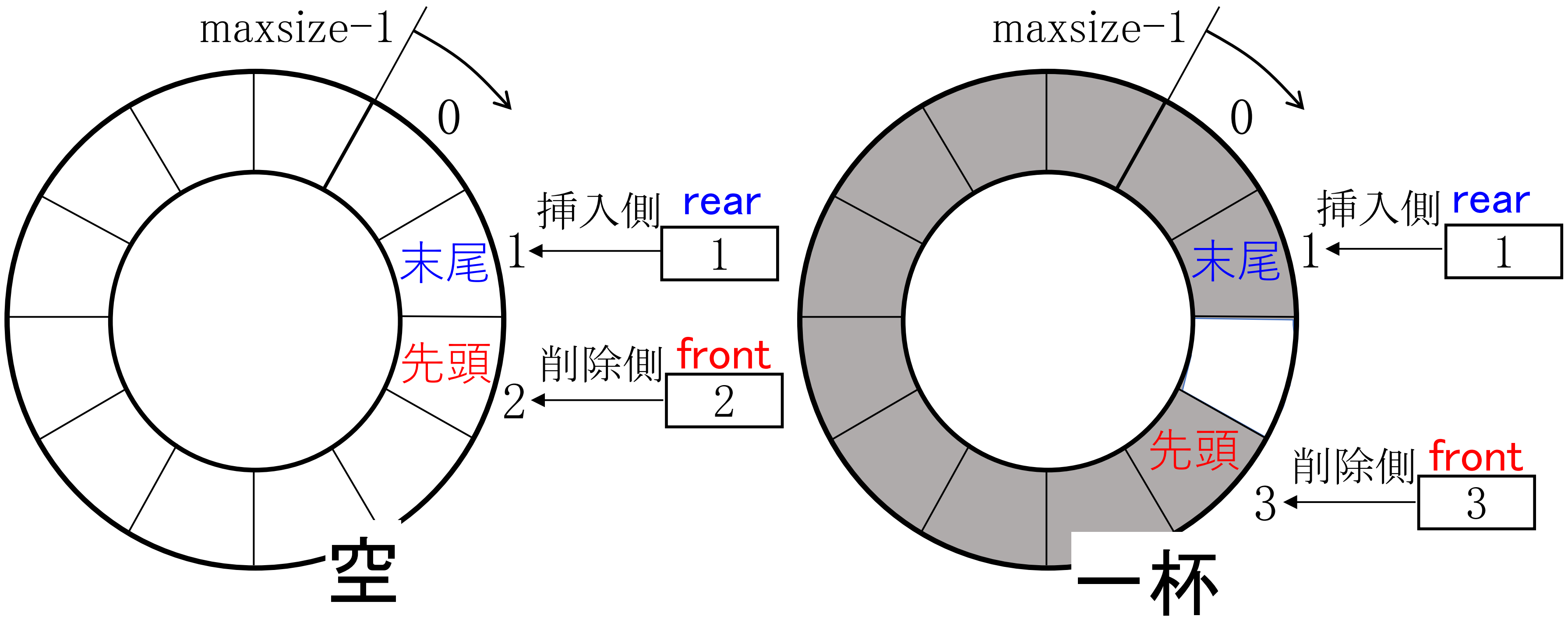
循環配列：空か一杯か？

- 空の状態とも一杯の状態ともとれる



循環配列：空と一杯を分けるために

- キューの最大長をmaxsize-1とする



実現アルゴリズム

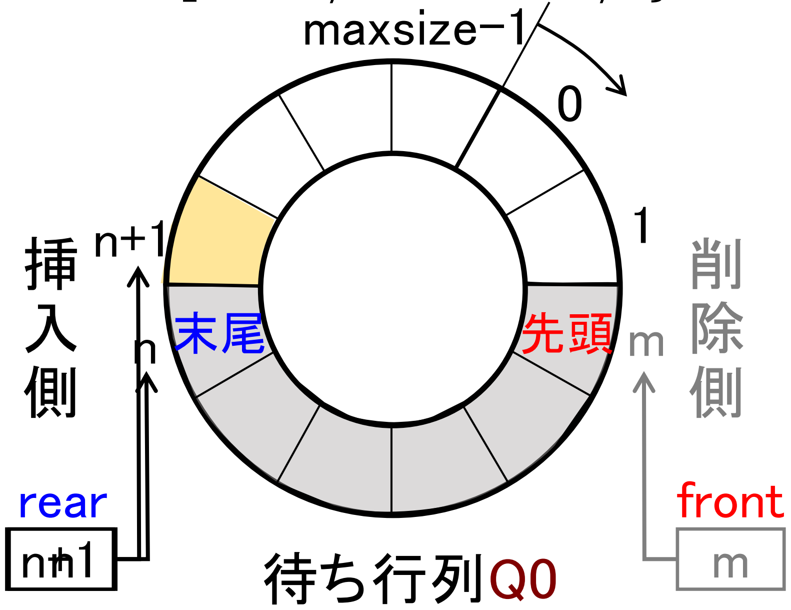
- 実現の制約：キューの大きさが配列長で制限される
- maxsize個以上のデータ挿入で，実現に基づくエラー

- データeをキューQ0に挿入 EnQueue(&Q0, e)
- キューQ0の要素を削除 DeQueue(&Q0)
- キューQ0の先頭要素の読み出し Retrieve(&Q0, &v0)
- キューQ0が空か？ Empty(&Q0)
- キューQ0を空にする Create(&Q0)
- 補助関数：位置を表す仮引数 i に循環的に1を加えた値を返す AddOne(i)

実現アルゴリズム EnQueue

- データeをキューQ0に挿入 EnQueue(&Q0, e)

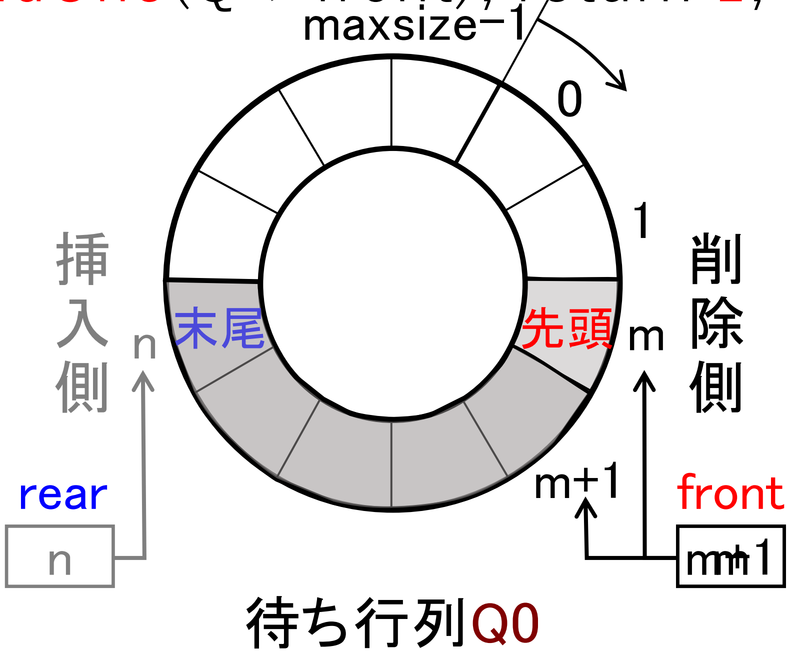
```
+int EnQueu(Queue *Q, Element e){
   if( AddOne( AddOne(Q->rear) ) == Q->front) return 0; /* 一杯*/
   else{
      Q->rear = AddOne(Q->rear);          /* eを末尾に挿入 */
      Q->value[Q->rear] = e; return 1; }
}
```



実現アルゴリズム DeQueue

- キューQ0の要素を削除 DeQueue(&Q0)

```
+int DeQueue(Queue *Q){
    if(Empty(Q)) return 0;        /* キューが空 */
    else{
        Q->front = AddOne(Q->front); return 1;    /* frontを進める */
    }
}
```



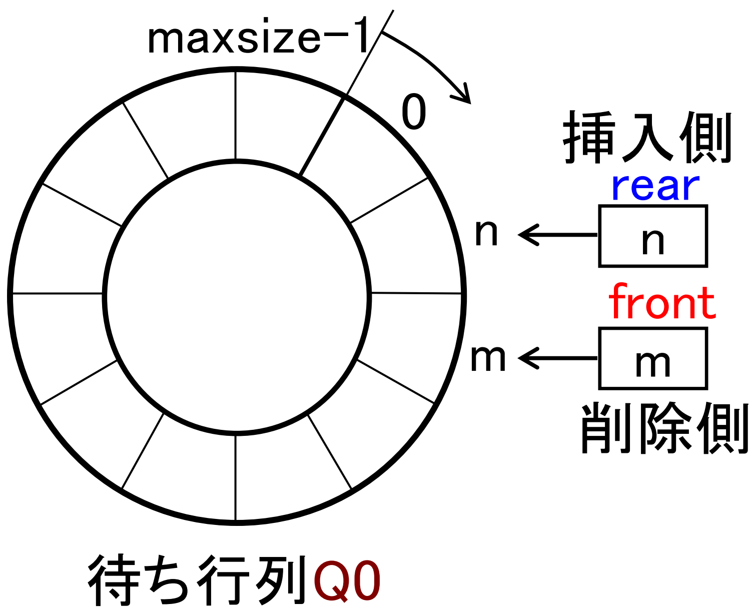
実現アルゴリズム Retrieve/Empty

- キューQ0の先頭要素の読み出し Retrieve(&Q0, &v0)

```
+int Retrieve(Queue *Q, Element *v){
    if (Empty(Q))  return 0;      /* キューが空 */
    else{  *v = Q->value[Q->front];  return 1; } }   /* *vに読出 */
```

- キューQ0が空か？ Empty(&Q0)

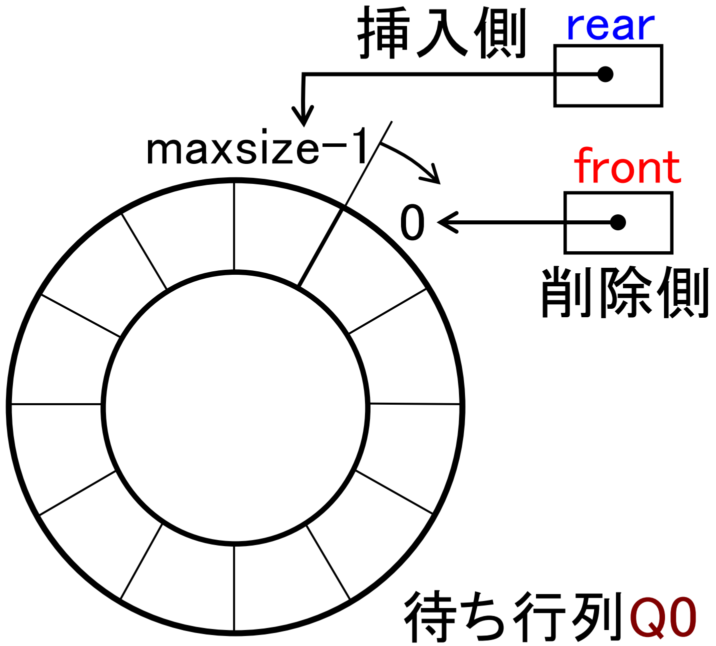
```
int Empty(Queue *Q){
    return(AddOne(Q->rear) == Q->front); }
```



実現アルゴリズム Create/補助関数AddOne

- キューQ0を空にする Create(&Q0)

```
void Create(Queue *Q){
    Q->front = 0;
    Q->rear = maxsize -1; }
```



- 補助関数：位置を表す仮引数 i に循環的に1を加えた値を返す AddOne(i)

```
int AddOne(int i){
    return (i + 1) % maxsize; }
```

$0 \leqq i < \text{maxsize} - 1 \Rightarrow i + 1$

$i = \text{maxsize} - 1 \Rightarrow 0$

実現アルゴリズムの効率

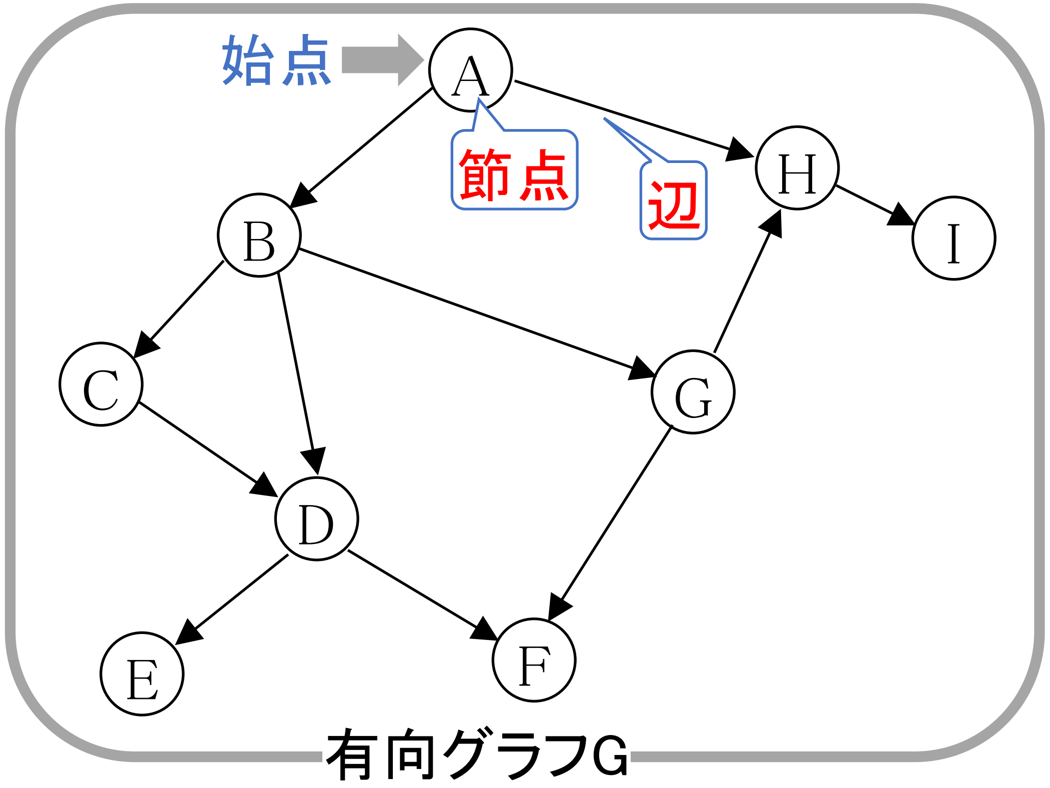
- リストを配列へのベタ詰めの実現：×
  - 途中の要素の挿入・削除に手間がかかる
- キューを配列にベタ詰めで実現：△
  - 要素の挿入・削除が両端の要素のみ
  - …配列ベタ詰めは最良？
  - => 末尾がmaxsize-1になると，配列の最初の空き要素が使えなくなる
- キューを循環配列（環状の配列）にベタ詰めで実現：〇
  - 空き領域を無駄なく使える

有向グラフの探索法

有向グラフの探索法

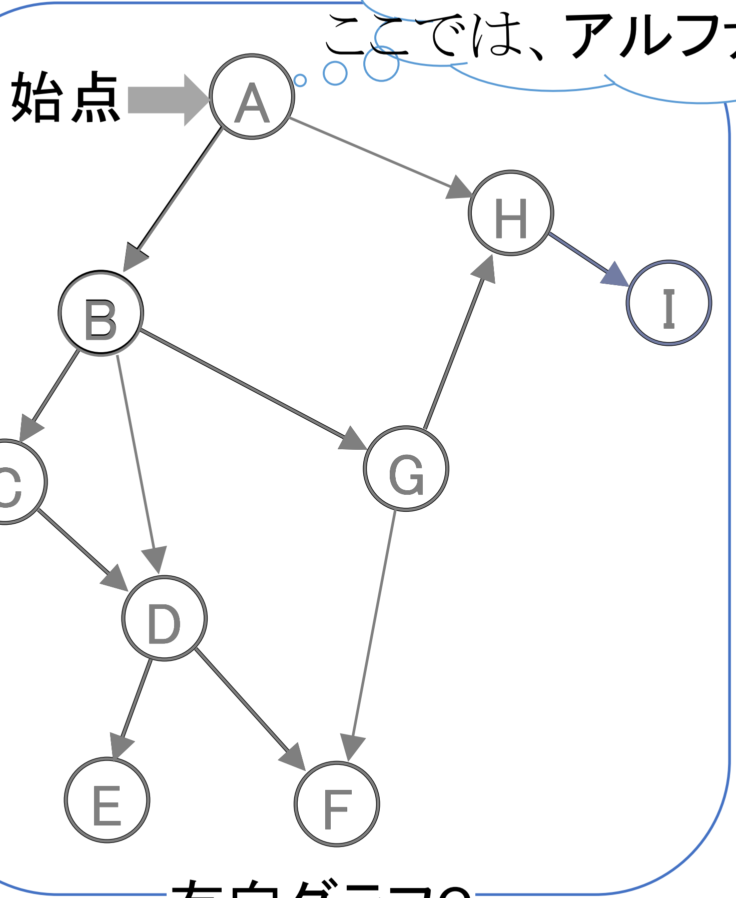
- グラフを系統的にたどって，すべての節点を訪問する

- 2つの系統的な探索法
  - 深さ優先探索法（depth-first search）
  - 幅優先探索法（breadth-first search）

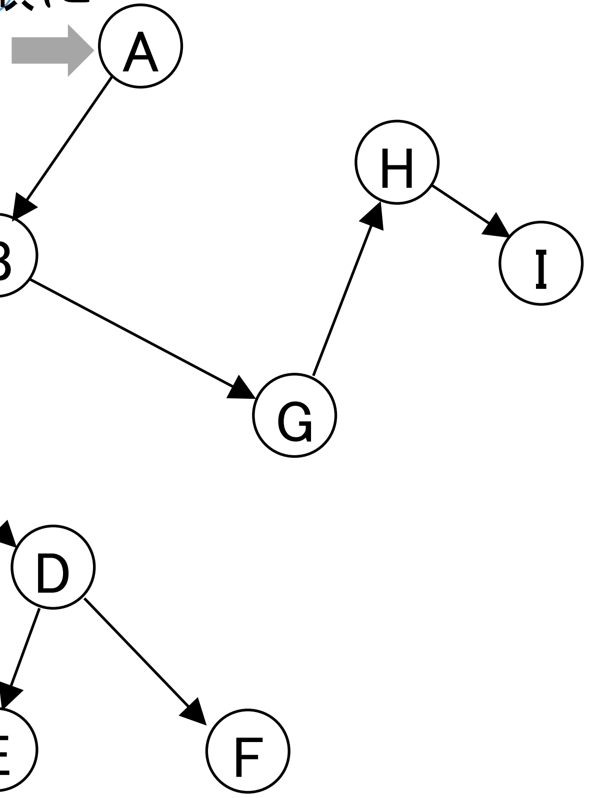


深さ優先探索法

節点は、順位をもつ。
ここでは、アルファベット順に



有向グラフG



Gを深さ優先探索した結果の木

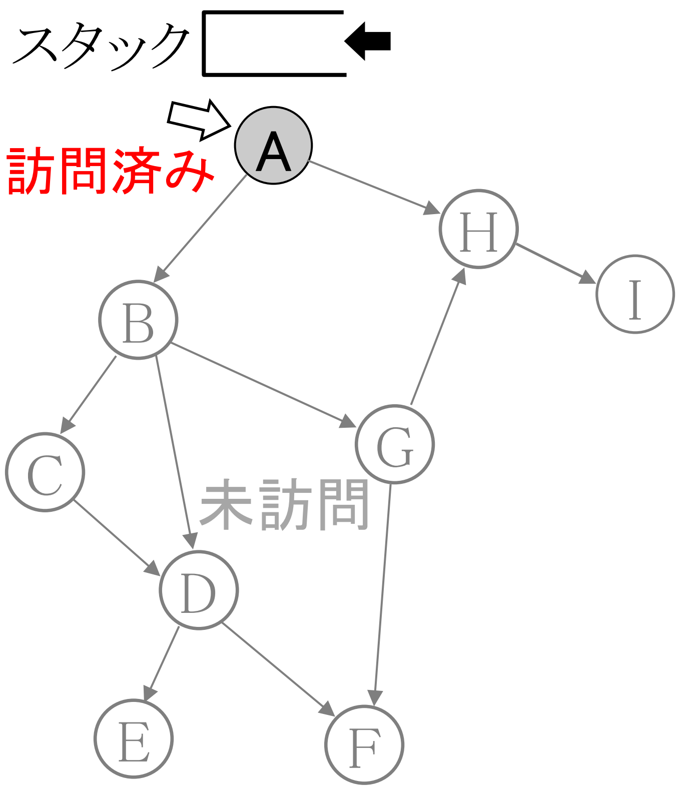
深さ優先探索法

- 0. **始点s**から**深さ優先**で訪問，sから訪問が終了したら探索終了
- 1. **節点v**から**深さ優先**の訪問
  - 1-1. vを訪問済みにする
  - 1-2.
    - vの**隣接未訪問**節点が**ある**とき,
      - 順位の高い節点wを選び辺(v, w)を出力する
      - **wから**の深さ優先訪問を行いそれが終了したら1-2を繰り返す
    - vの**隣接未訪問**節点が**ない**とき
      - vからの深さ優先での訪問を終了

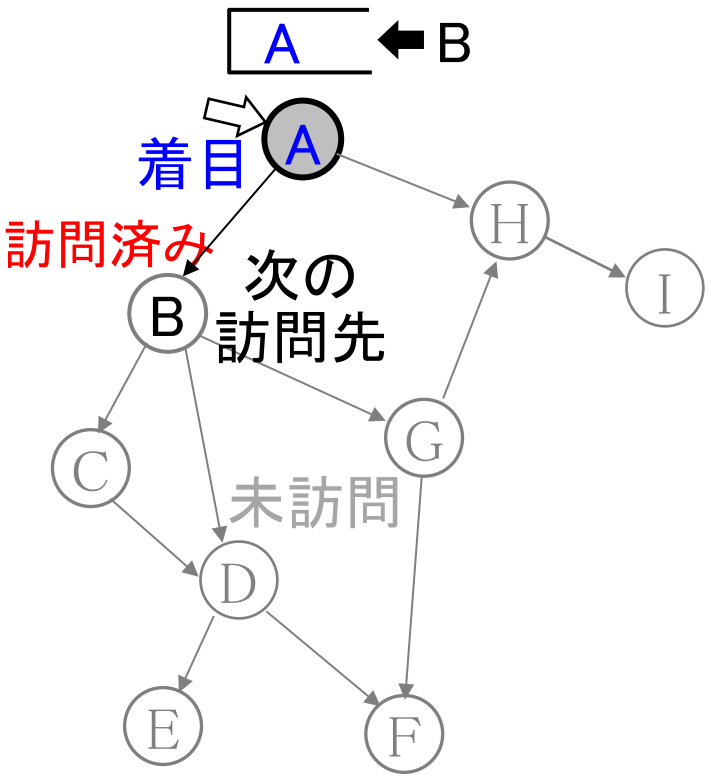
⇒再帰的定義で実現

⇒スタックをつかっても実現可能

深さ優先探索をスタックで実現

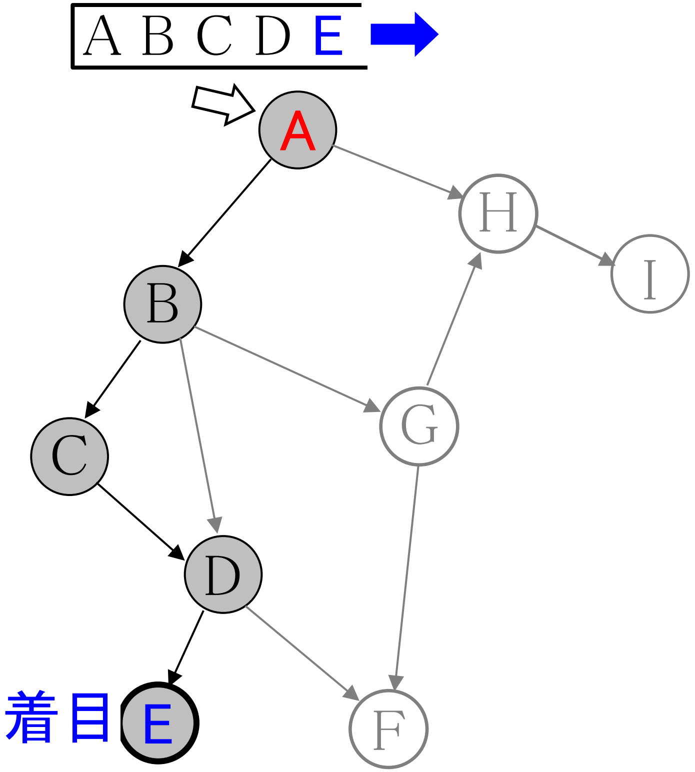


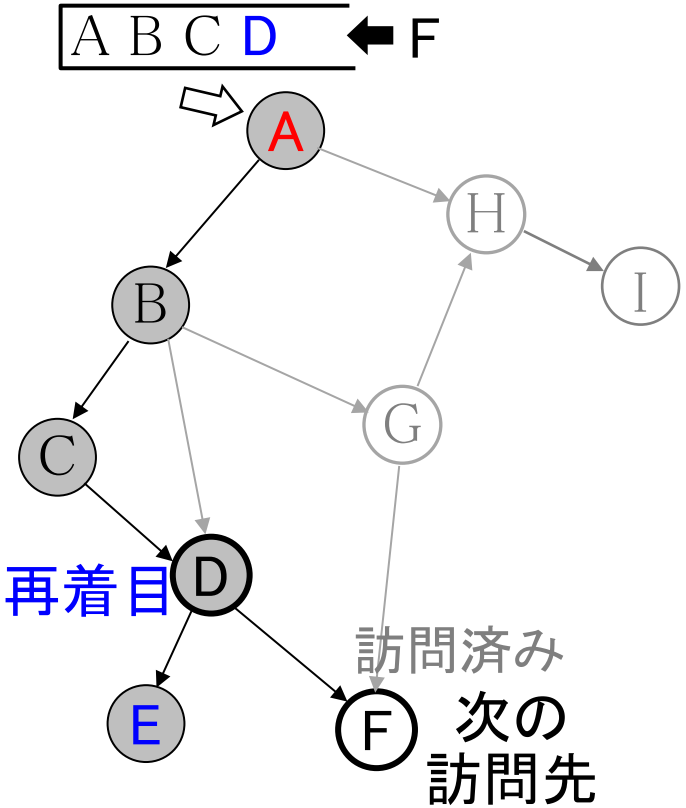
始点Aを訪問済みにして、
スタックにプッシュダウン。



スタック先頭のAに着目。
Aの次の未訪問隣接節点Bを
訪問済みとし、プッシュダウン。
辺(A,B)を出力。

深さ優先探索をスタックで実現

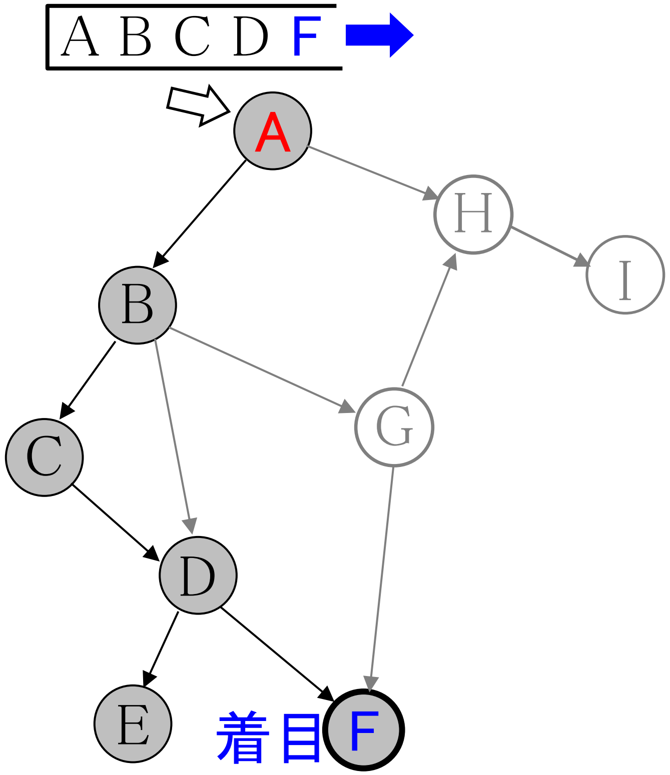




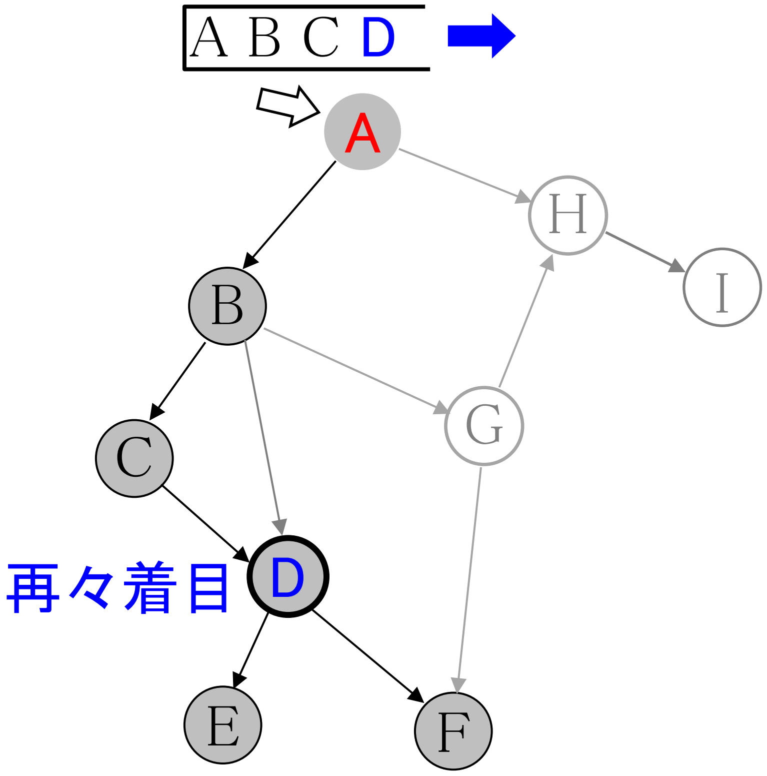
C,D,Eと**プッシュダウン**。
辺(B,C),(C,D),(D,E)を出力.
Eに**着目**、**隣接節点がない**ため、
Eを**ポップアップ**する。

スタックの先頭Dに再び着目。
Dの未訪問隣接節点Fを
訪問済みとし、**プッシュダウン**。
辺(D,F)を出力。

深さ優先探索をスタックで実現

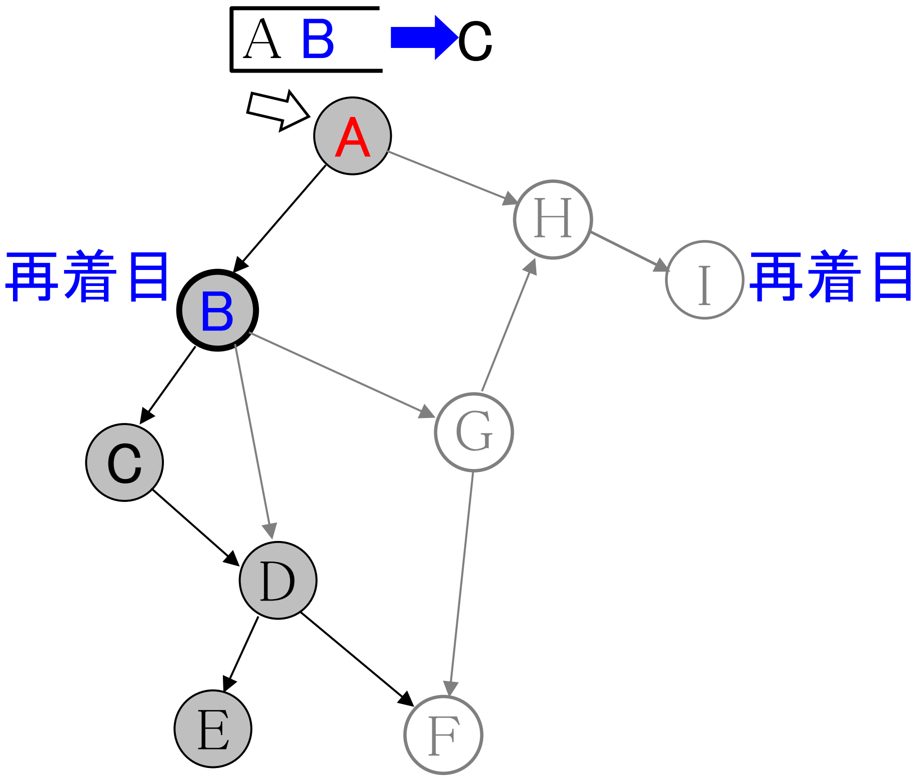


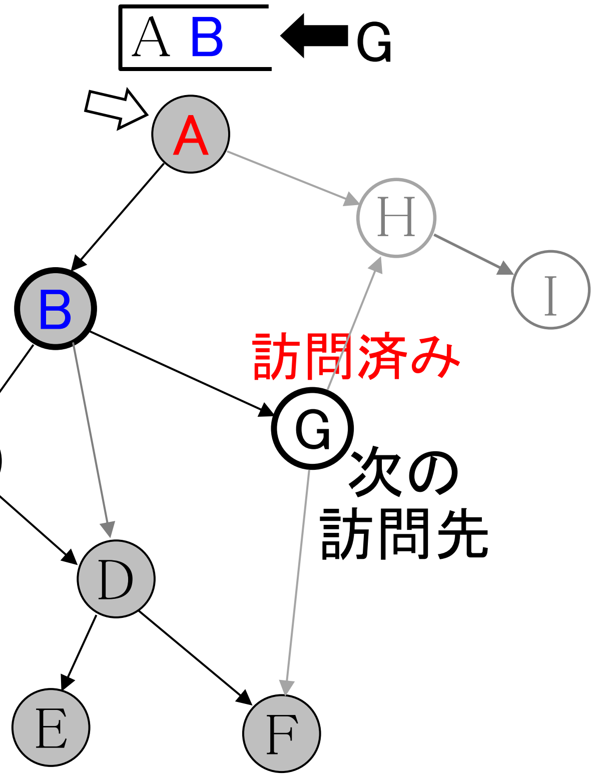
Fに着目。
Fに未訪問隣接節点がないので、
スタックからFをポップアップ。



スタックの先頭Dに再び着目。
Dの未訪問隣接節点がないので、
スタックからDをポップアップ。

深さ優先探索をスタックで実現

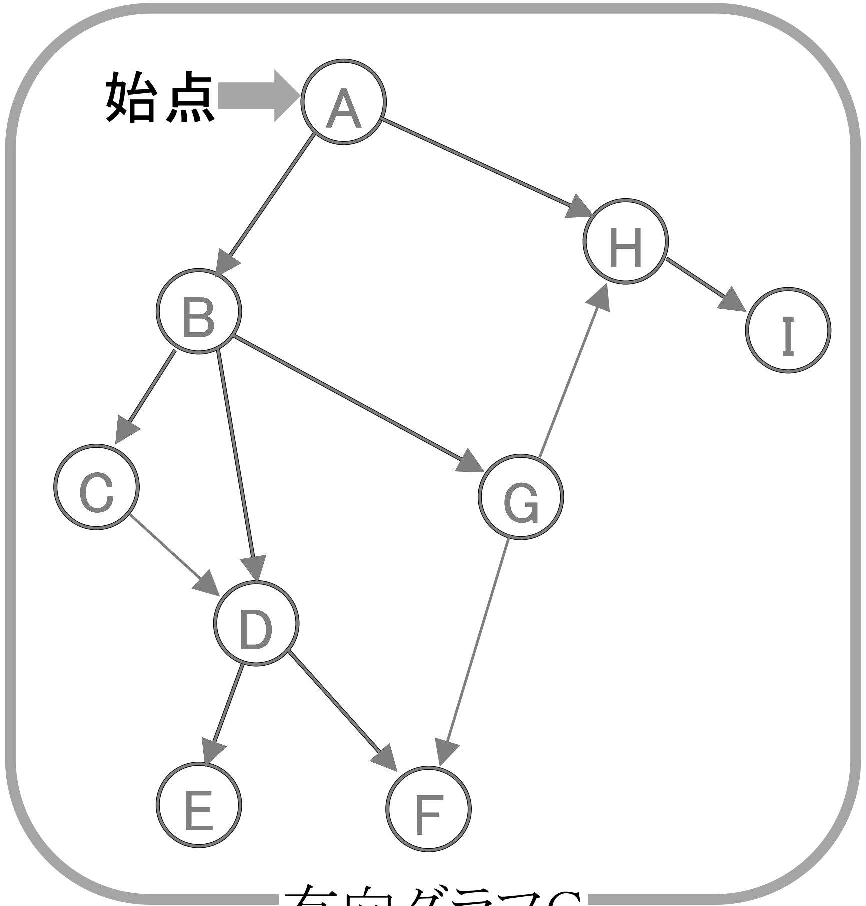




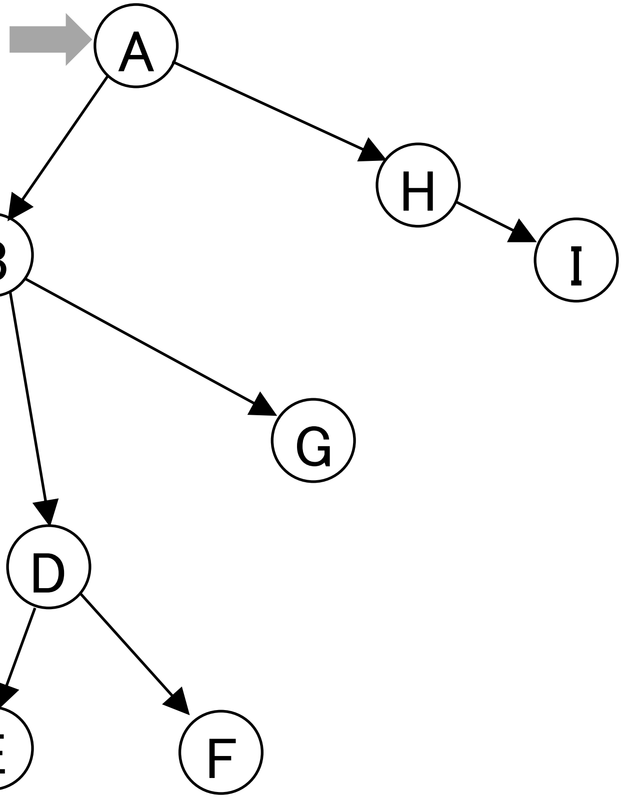
Cの未訪問隣接節点もないため、
Cも**ポップアップ**。
スタックの先頭Bに再着目。

Bに再着目。
Bの未訪問隣接節点Gを
訪問済みとし、**プッシュダウン**。
辺(B,G)を出力。

幅優先探索法



有向グラフG



Gを**幅優先**探索した結果の木

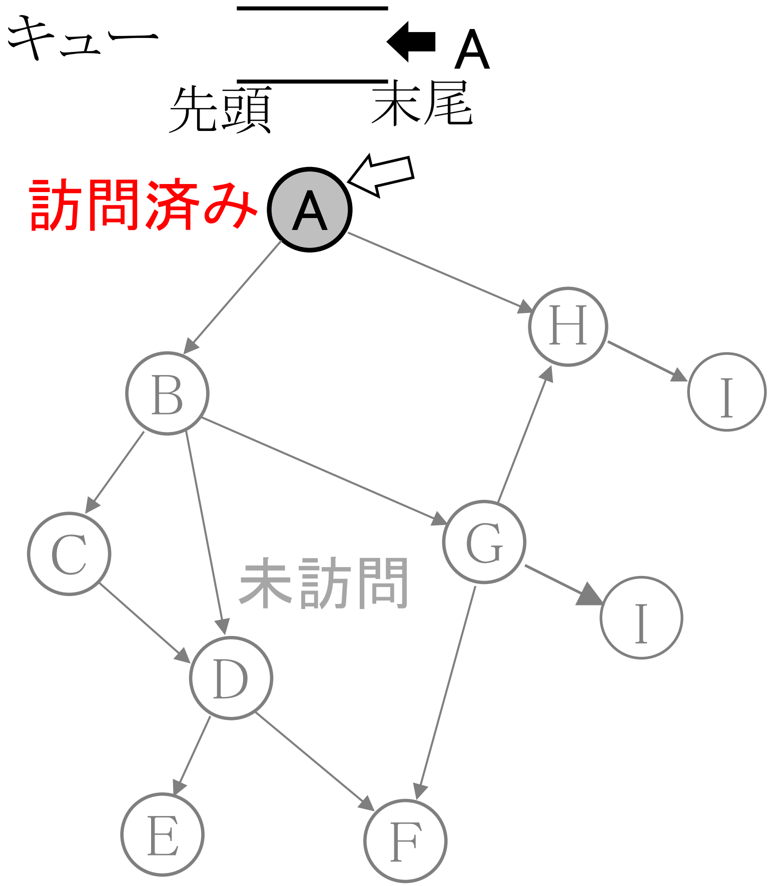
幅優先探索法

- 0. 始点sを訪問済みにする．初期節点列としてsを与える
- 1. 訪問済みの節点列$v_1, v_2, \cdots, v_n$が与えられたとき
  - 1-1. $v_1, v_2, \cdots, v_n$に隣接するすべての未訪問節点列$w_1, w_2, \cdots, w_m$を求め，訪問済みにする．そのような節点がなければ探索終了とする
  - 1-2. このとき，節点v_iの隣接節点としてw_jを**訪問済み**にしたならば辺（v_i, w_j）を出力
  - 1-3. 節点列 $w_1, w_2, \cdots, w_m$を$v_1, v_2, \cdots, v_n$として，ステップ1を繰り返す

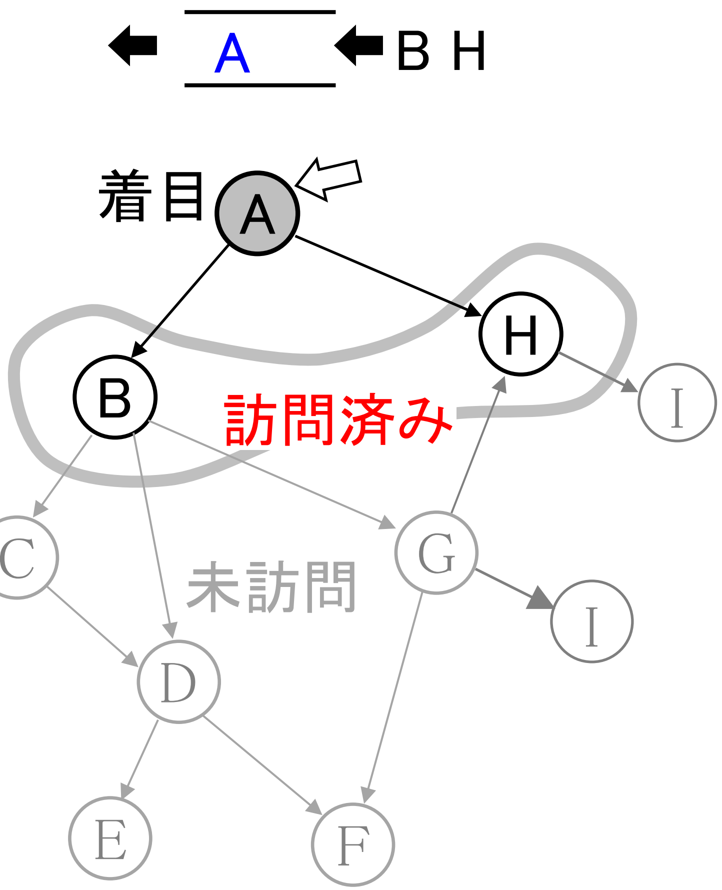
⇒再帰的定義で実現

⇒キュー（待ち行列）でも実現可能

幅優先探索をキューで実現



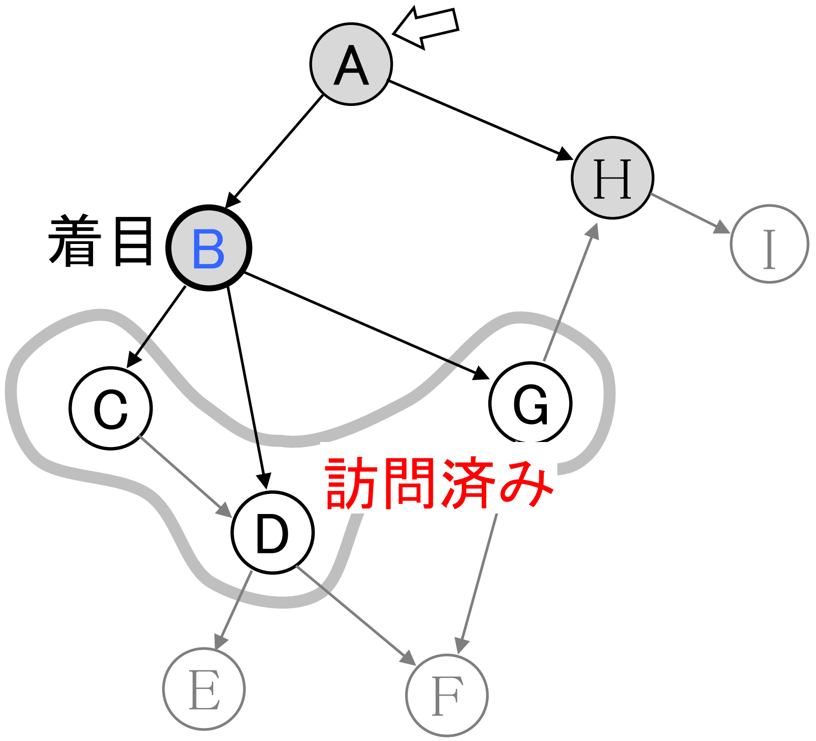
始点Aを**訪問済み**にして、
待ち行列に**挿入**。

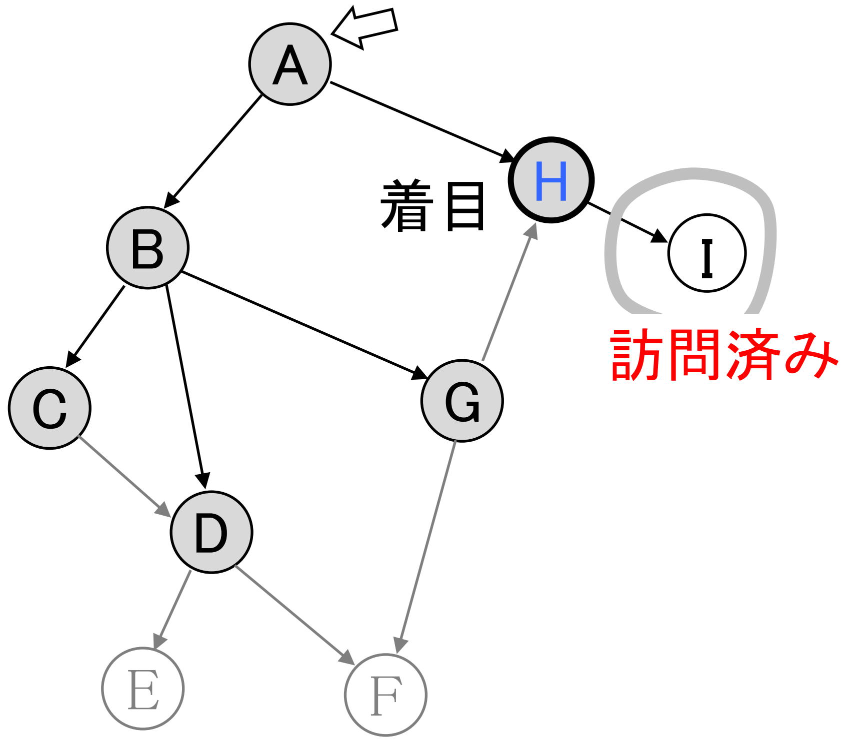


待ち行列の先頭のAに着目。
Aの未訪問隣接節点B,Hを**訪問済み**とし、**挿入**。Aは**削除**。
辺(A,B),(A,H)を出力。

幅優先探索をキューで実現

待ち行列 ← B H ← C D G　　　　← H C D G ← I





待ち行列の先頭のBに着目。
Bの未訪問隣接節点C,D,Gを
訪問済みとし挿入、Bは**削除**。
辺(B,C),(B,D),(B,G)を出力。

待ち行列の先頭のHに着目。
Hの未訪問隣接節点I,を
訪問済みとし挿入、Hは**削除**。
辺(H,I)を出力。

探索法のまとめ

- 2つの系統的な探索法
  - 深さ優先探索法（depth-first search）
    - スタックで実現可能
  - 幅優先探索法（breadth-first search）
    - キュー（待ち行列）で実現可能

まとめ

- 基本データ構造：
  - スタック
  - キュー（待ち行列）
- 有向グラフの探索法
  - 深さ優先探索（スタックで実現）
  - 幅優先探索（キューで実現）

演習

- スタックをC言語で実現する
  - 配列べた詰めでも，連結リストでもOKです
  - （スタックはリストの操作が制限された物と考えられるので簡単？）
  - 最低ひとつは作って下さい
- 作成したスタックを使って「使用例：簡単な行編集プログラム」を作って下さい
- キューをC言語で実現する
  - 配列ベタ詰めでも，連結リストでもOKです
  - ただの配列でも，循環配列でも，連結リストで循環配列を実現してもOK
  - 最低ひとつは作ってください

提出について

- LETUSにて
- 提出物：ソースコードのファイル（pdfの説明をつけてもOK）
- 2023/6/12 10:30まで