

# 最短経路問題

---

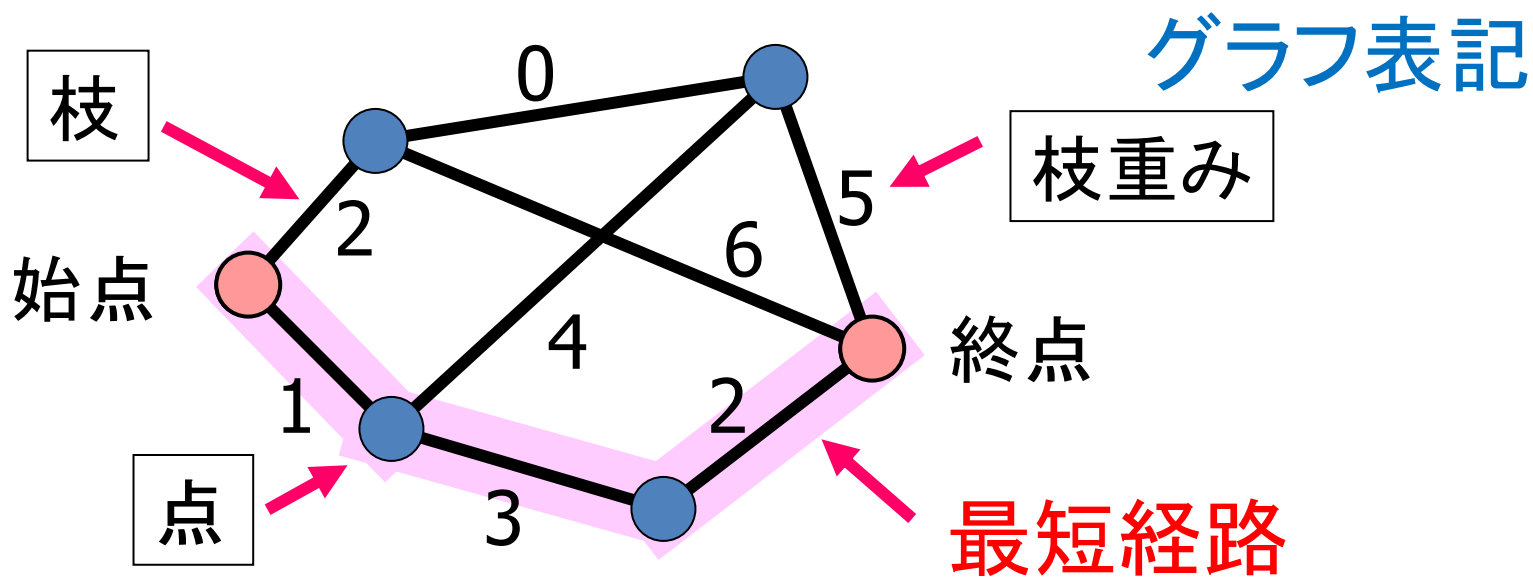
オープンラボ C言語プログラミング 課題

電気電子物理工学科伊藤研究室

2023年11月13日版

# 最短経路問題

- 始点から終点へ行く最も短い経路  
= 枝重み合計が最小の経路を見つける



# Bellman方程式

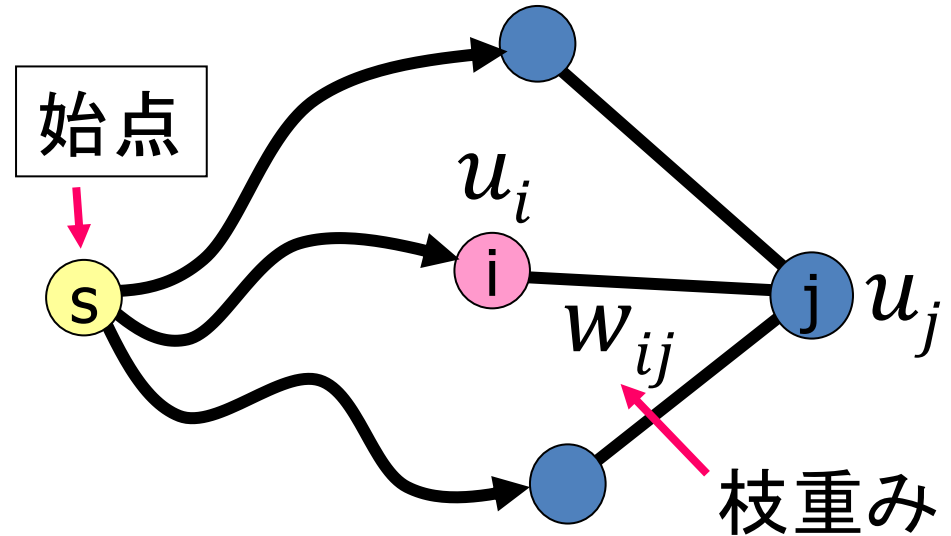
- 点 $j$ の最短経路長 $u_j$ について成り立つ関係式

$$u_s = 0$$

$$u_j = \min_i \{u_i + w_{ij}\}$$

$\forall j \neq s$

Bellman方程式



- Bellman方程式(連立方程式)を満たす $u_j$ を求めれば、それが最短経路長

# Bellman方程式を解く(1)

- 1. 点 $j$ の最短経路長 $u_j$ に適切な初期値を設定
  - 通常は $+\infty$ (実際には十分大きな正の値で代用)
- 2. 全点 $j$ と、その全ての入力枝 $(i, j)$ について $u_i + w_{ij}$ が $u_j$ よりも小さければ、その値で $u_j$ を更新( $u_j > u_i + w_{ij}$ ならば $u_j$ に $u_i + w_{ij}$ を代入)
- 3. 全点 $j$ について $u_j$ の更新がなければ終了  
いずれかの点で更新があれば、2.に戻る

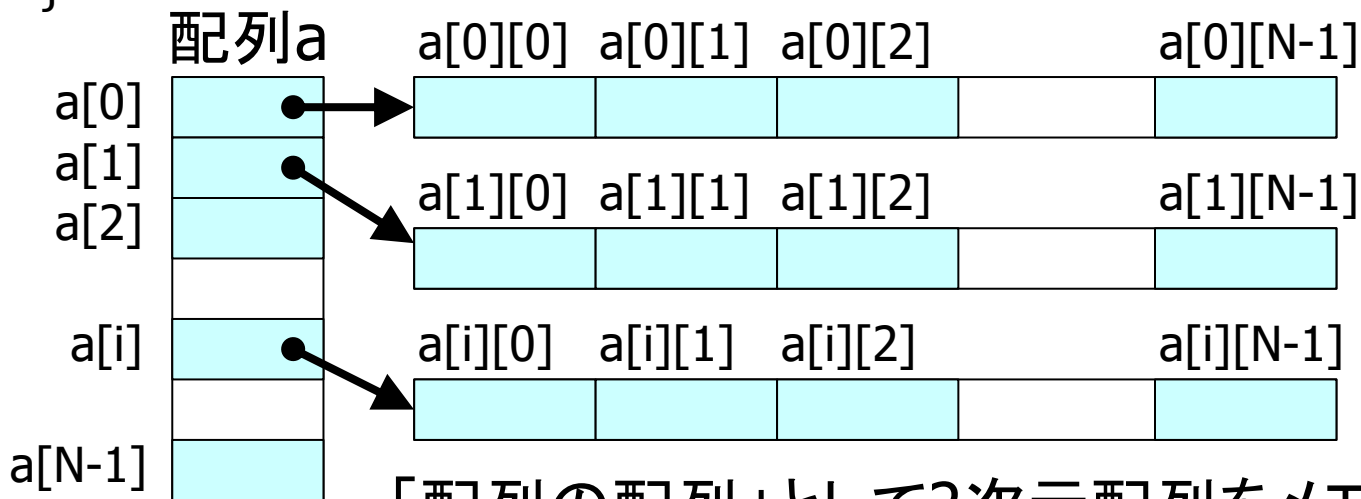
→ Bellman-Ford法

計算複雑度  $O(NE)$  または  $O(N^3)$  (点数 $N$ , 枝数 $E$ )

# 枝を記録する2次元配列

```
#define INTINF 999999 ← 十分大きい正数で正無限大を近似
int N; ← グラフの点の数
int **a; ← 枝を表す2次元配列を指す
int i,j;
Integer infinity
```

```
a = (int **)malloc(N*sizeof(int *)); ← 要素数Nの配列を割り当て
for(i=0 ; i<N ; i++){
    a[i] = (int *)malloc(N*sizeof(int)); ← 要素数Nの配列を割り当て
    for(j=0 ; j<N ; j++) a[i][j] = INTINF; ← 各要素を無限大で初期化
}
```

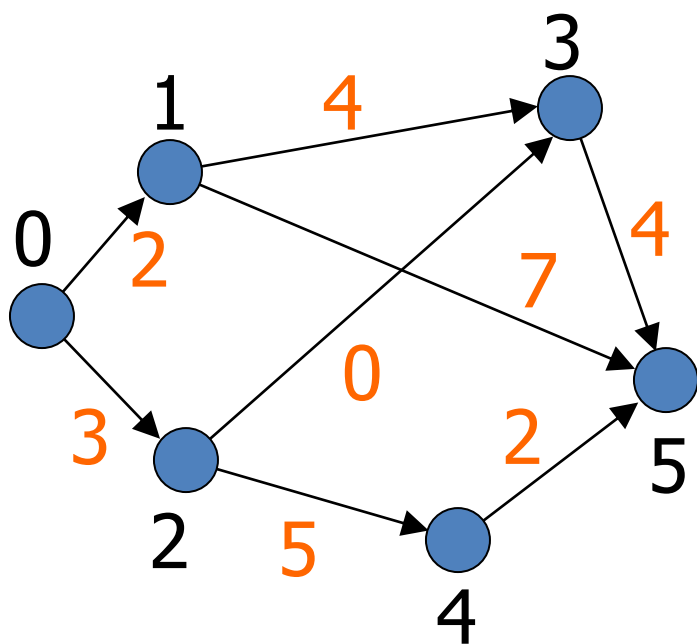


「配列の配列」として2次元配列をメモリ上に構築

# グラフを記録するデータファイルの書式

## ■ グラフは、テキストファイルから読み込む

グラフ → グラフを記述する  
テキストファイル



```
N 6  
E 0 1 2  
E 0 2 3  
E 1 3 4  
E 1 5 7  
E 2 3 0  
E 2 4 5  
E 2 5 2  
E 3 5 4  
E 4 5 2  
L 5 7
```

## ■ テキストファイルの書式

- *N Num*  
グラフの点数が *Num* 個
- *E From To Weight*  
始点 *From*、終点 *To* の  
枝の重みが *Weight*
- *L Node Length*  
点 0 から点 *Node* への  
最短経路長は *Length*

# ファイルからデータ読み取り

- テキストデータでファイルに記録されたグラフの情報を読み取る
- C言語プログラムからファイルを開き、内容を読み出し、解析、データ構造に記録、ファイルを閉じる

```
FILE *fp;
if( _tfopen_s( &fp, strFilename, _T("r") ) ) return;

char buf[256];
while(1){
    fgets( buf, 255, fp );
    if( feof(fp) ) break;
    // 読み込んだ行の解析とデータ記録をここで行う
}
fclose(fp);
```

ファイルを開く

ファイル名は変数strFilename (CString型)に代入されているとする

読み出し(read)モード

無限ループ

1行読み込み

最後の行まで読み終わっていればループ終了

ファイルが存在しないなどopenに失敗したら処理を中止して戻る

データを読み終わったらファイルを閉じる

# 読み込んだ行の解析

```
char DLM[] = " "; ← 区切り文字(文字列を空白文字で区切る)
```

```
char buf[256];
```

```
char *ptr, *ptr1, *ptr2;
```

```
char *pTokenContext;
```

データ例

N 10000

```
while(1){
```

```
  fgets( buf, 255, fp );
```

```
  if( feof(fp) ) break;
```

```
  if( strlen(buf)>0 && buf[strlen(buf)-1] == '\n' ) buf[strlen(buf)-1] = '\0';
```

```
  switch( buf[0] ){
```

```
  case 'N': ← 最初の1文字が 'N' のとき(例えば"N 10000")
```

```
    ptr = strtok_s( buf, DLM, &pTokenContext ); ← "N"を読み取る
```

```
    ptr1 = strtok_s( NULL, DLM, &pTokenContext ); ← ptr1は"10000"を指す
```

```
    N = atoi( ptr1 ); ← 点数を表す数字文字列を  
    // ここで枝を記録する2次元配列を構築 数値(整数)に変換
```

```
    break;
```

```
  } ← 最初の1文字が 'E' や 'L' のときの処理をここに書き加える
```

```
}
```

**注意:** ファイルでは、枝の情報(E)よりも先に点の数の情報(N)が記述されている必要あり



# 枝の読み込みと記録

```
int nFrom, nTo;
while(1){
    // ここでファイルから1行読み込み(詳細は省略)
    switch( buf[0] ){
    case 'E':
        ptr = strtok_s( buf, DLM, &pTokenContext );
        ptr1 = strtok_s( NULL, DLM, &pTokenContext );
        ptr2 = strtok_s( NULL, DLM, &pTokenContext );
        ptr = strtok_s( NULL, DLM, &pTokenContext );
        nFrom = atoi( ptr1 );
        nTo = atoi( ptr2 );
        if(nFrom < 0 || nFrom >= N || nTo < 0 || nTo >= N ) continue;
        a[nFrom][nTo] = atoi( ptr );
        break;
    }
}
```

データ例

E	0	1	2
E	0	2	1
E	1	3	0

← 枝の始点  
← 枝の終点  
← 枝の重み

始点、終点が範囲外\*なら  
枝の情報を無視(スキップ)  
(\*データファイルが正しければ生じない)

↑ 枝の情報を記録

# 最短経路長(解)の読み込みと記録

```
int nNodeL, nLengthShortest;
```

```
while(1){
```

```
// ここでファイルから1行読み込み(詳細は省略)
```

```
switch( buf[0] ){
```

```
case 'L':
```

```
ptr = strtok_s( buf, DLM, &pTokenContext );
```

```
ptr1 = strtok_s( NULL, DLM, &pTokenContext );
```

```
ptr2 = strtok_s( NULL, DLM, &pTokenContext );
```

```
nNodeL = atoi( ptr1 );
```

```
nLengthShortest = atoi( ptr2 );
```

```
break;
```

```
}
```

データ例

L 9999 509

← 点  
← 最短経路長

このグラフにおいて  
点nNodeLの最短経路長がnLengthShortest



- 自分がプログラムで求めた結果のu[nNodeL]がnLengthShortest(正解)と一致するか確認

# Bellman-Ford法のプログラム

```
#define INTINF 999999 ← 正無限大を表す十分大きな値  
int i,j;                (すでに定義(define)済みならば改めて定義しなくてよい)  
int bUpdate;  
int *u = (int *)malloc(N*sizeof(int)); ← 経路長を記録する配列を用意  
  
for(i=1 ; i<N ; i++) u[i] = INTINF; ← 各点の経路長を無限大に初期化  
u[0] = 0; ← 点0を始点とし、経路長は0とする  
  
while(1){ ← 無限ループ  
    bUpdate = 0;                (最短経路長が求まるまで繰り返し)  
    for(j=0 ; j<N ; j++){  
        for(i=0 ; i<N ; i++){ ← iとjに関する2重ループ  
            if( u[j] > u[i]+a[i][j] ){  
                u[j] = u[i]+a[i][j];  
                bUpdate = 1; ← 経路長を更新したことを記録  
            }  
        }  
    }  
}   
if( bUpdate == 0 ) break; ← 経路長の更新がなかった、  
                            すなわち最短経路長が求まったので終了  
}
```

# 改善1: ループ多重化の順序検討

## ■ Bellman方程式とBellman-Ford法プログラム

$$\text{Bellman方程式 } u_j = \min_i \{u_i + w_{ij}\}$$

↓ Bellman方程式の定義通りにプログラム化

```
for(j=0 ; j<N ; j++){  
  for(i=0 ; i<N ; i++){  
    if( u[j] > u[i]+a[i][j] ){  
      u[j] = u[i]+a[i][j];  
    }  
  }  
}
```

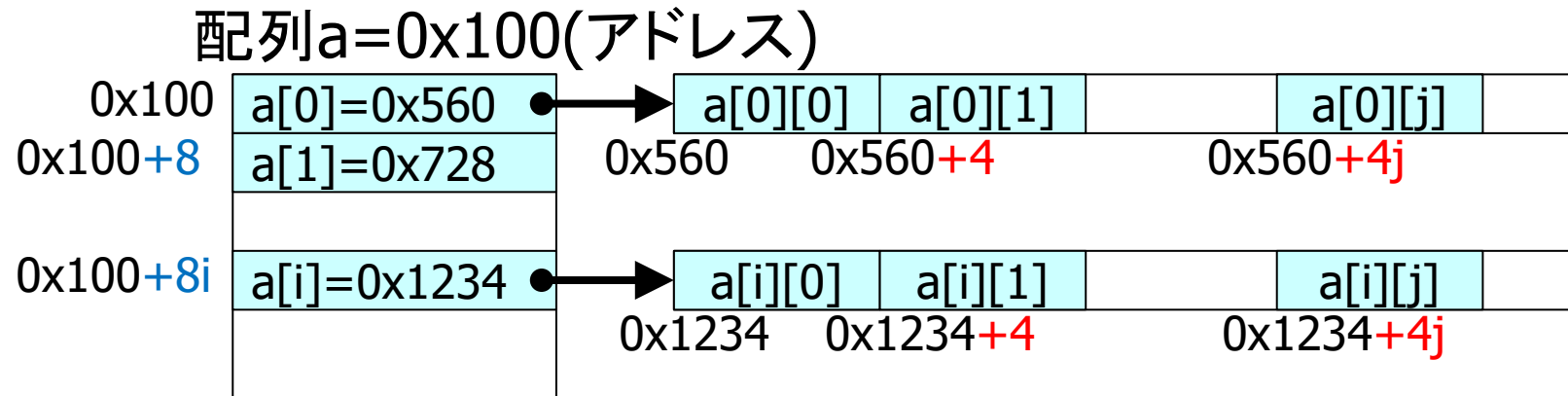


jに関するループの中に  
iに関するループがある  
二重ループになっている

・・・配列aがa[0][j],a[1][j],a[2][j],...の様に参照される

# 改善1: ループ多重化の順序検討

- 配列要素の参照(読み書き)は手間がかかる  
整数型配列  $a[i][j]$  (要素1つ当たり4バイト)

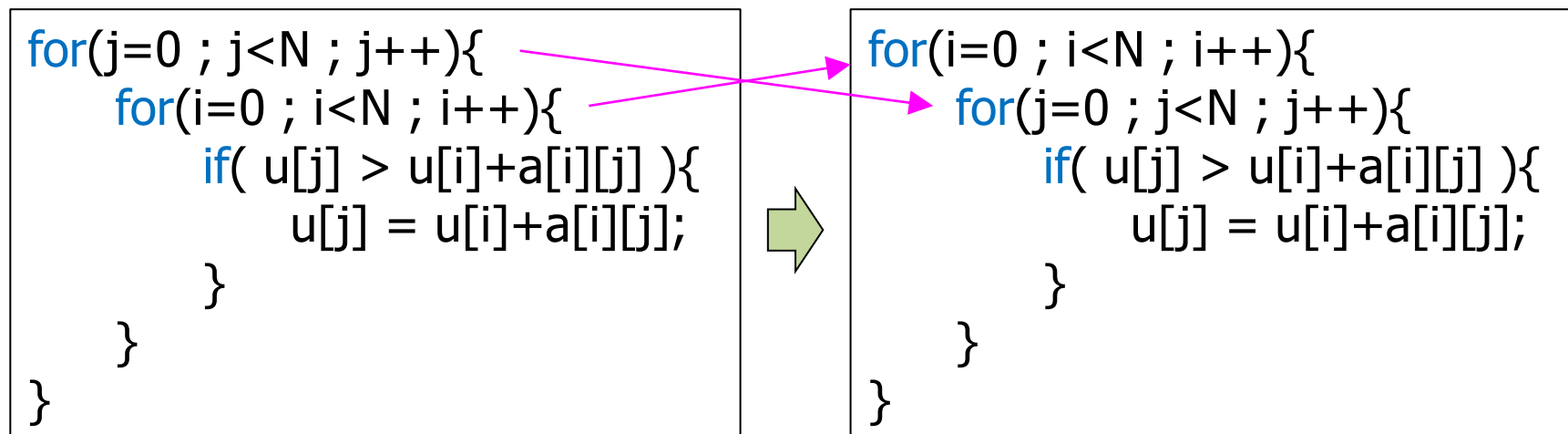


- ①  $i$ の値を8倍\*する 計算機システムが行う処理
- ② 配列  $a$ のアドレスに加算して  $a[i]$ のアドレスを求める
- ③  $a[i]$ の値(配列  $a[i]$ の先頭アドレス)を読み出す
- ④  $j$ の値を4倍する
- ⑤ 配列  $a[i]$ のアドレスに加算して  $a[i][j]$ のアドレスを求める
- ⑥  $a[i][j]$ を読み出し、または書き込み

\*64ビットモードの場合。32ビットモードの場合は4倍する。

# 改善1: ループ多重化の順序検討

## ■ iに関するループとjに関するループを入れ替え



i値とj値のすべての組み合わせを試すことは変化なし

→ 得られる結果に違いはない

・・・配列aがa[i][0],a[i][1],a[i][2],...の様に参照される

C言語コンパイラの最適化により、前ページ①②③の処理がjに関するループの外で1回だけ実行されるようになる

※使用するコンパイラや最適化レベルによって効果は異なる

# 改善2: 配列→リスト

---

- 枝( $i, j$ )の重みを二次元配列で記録  $a[i][j]$
- 点 $j$ を $0, 1, 2, \dots$ と順に増やして枝( $i, j$ )の有無を調査



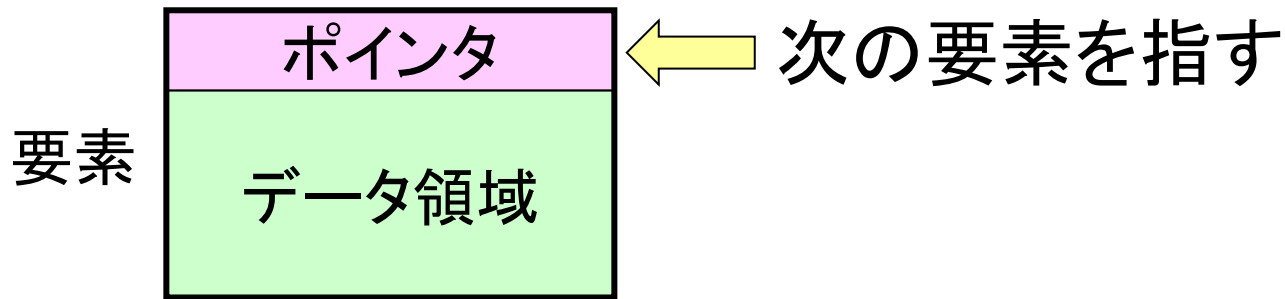
枝数が少ない場合は無駄が多く、効率が悪い  
(多くの場合に点 $i, j$ の間に枝は存在しない)



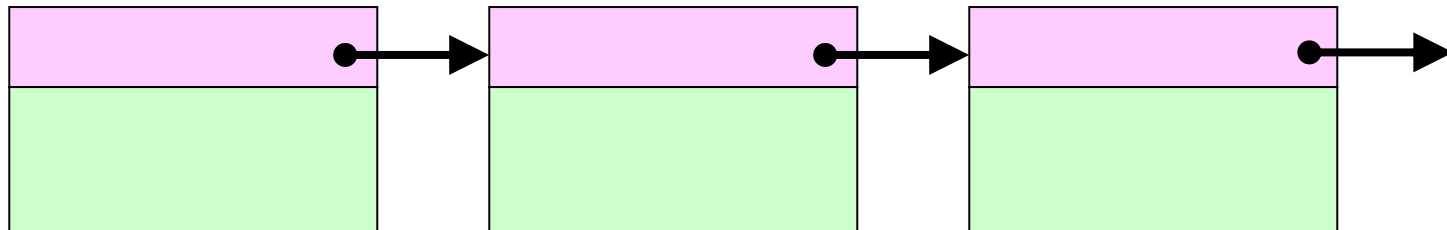
- 二次元配列の代わりに  
枝が存在する終点 $j$ だけを記録する方式が必要
- ➡ リスト構造(Linked list)を利用

# リスト構造とは

- 要素が順につながったデータ構造
- データ領域の他に、次の要素を指すポインタを備える



- ポインタを用いて要素をつなげる=リスト





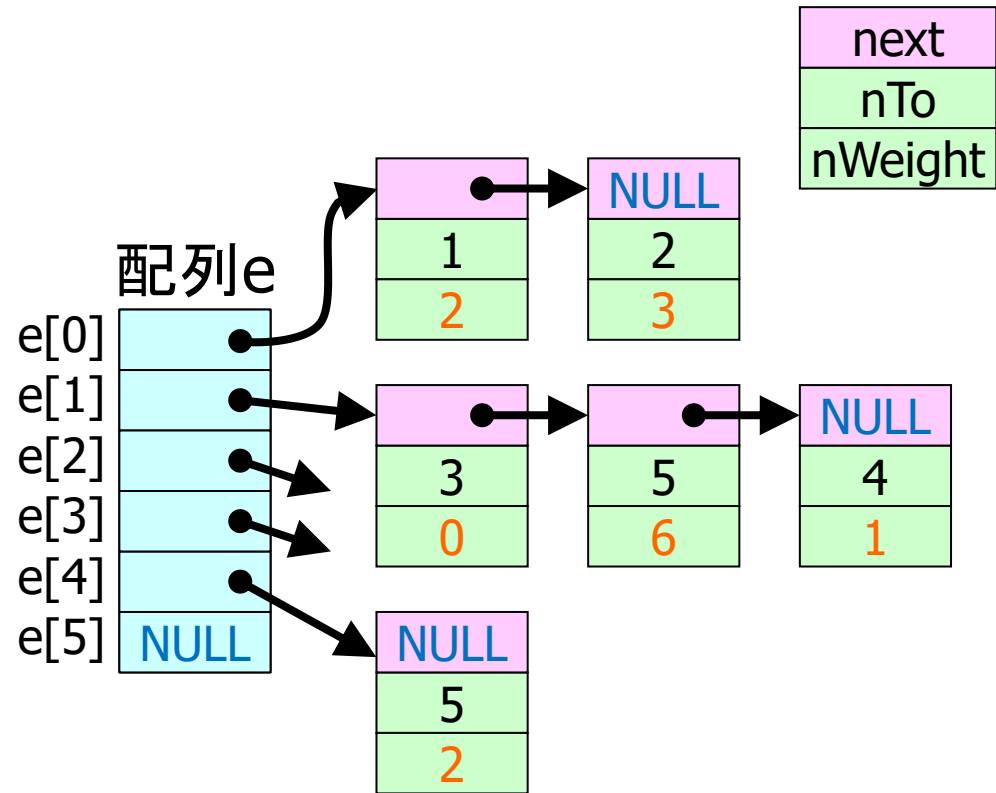
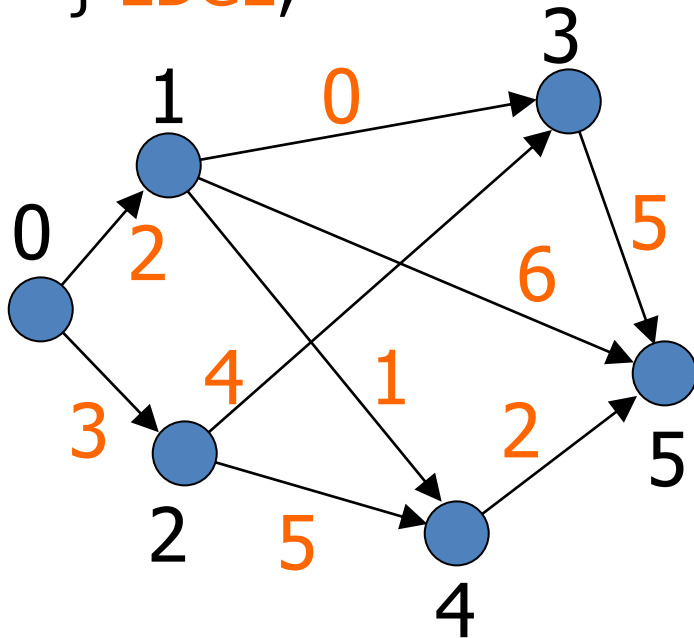
# グラフの枝をリスト構造で表す

## ■ リストの要素(枝)

```
typedef struct EdgeRec {  
    struct EdgeRec *next; // 次の要素を指すポインタ  
    int nTo; // 終点番号  
    int nWeight; // 枝重み  
} EDGE;
```

← 構造体

次の要素を指すポインタ



# グラフの枝をリスト構造で表す

## ■ 始点の配列を作成

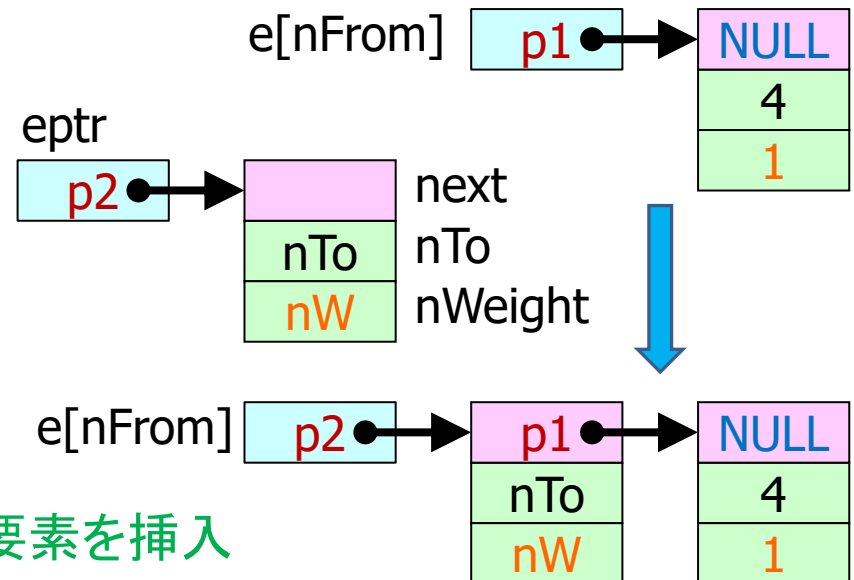
```
int N;      ← グラフの点の数
EDGE **e;  ← 始点ごとの枝リストへの先頭を記録する1次元配列
int i;
```

```
e = (EDGE **)malloc(N*sizeof(EDGE *)); ← 要素数Nの配列を割り当て
for( i=0 ; i<N ; i++ ) e[i] = NULL; ← 各リストを空に初期化
```

## ■ 枝(nFrom,nTo)を追加

```
int nTo,nFrom,nW;
EDGE *eptr;      ← 要素を1つ確保
```

```
eptr = (EDGE *)malloc(sizeof(EDGE));
eptr->nTo = nTo;
eptr->nWeight = nW;
eptr->next = e[nFrom];
e[nFrom] = eptr; } リストの先頭に要素を挿入
```



# グラフの枝をリスト構造で表す

## ■ 枝を順に調べる

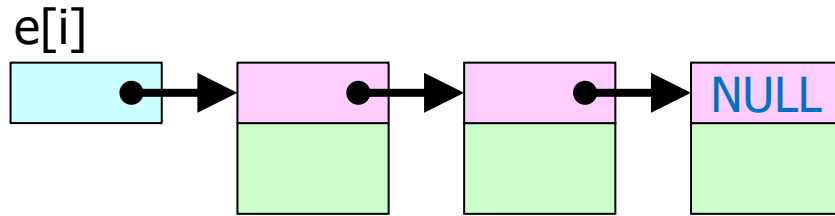
```
int i,j,nW;
```

```
EDGE *eptr;
```

```
for( eptr=e[i] ; eptr!=NULL ; eptr=eptr->next ){ ← 始点iの枝を順に調べる
    j = eptr->nTo; ← 終点j、枝重みnW
    nW = eptr->nWeight;
    // ここで枝(i,j)の重みがnW
    // 点jの最短経路長の比較・更新などを処理
}
```

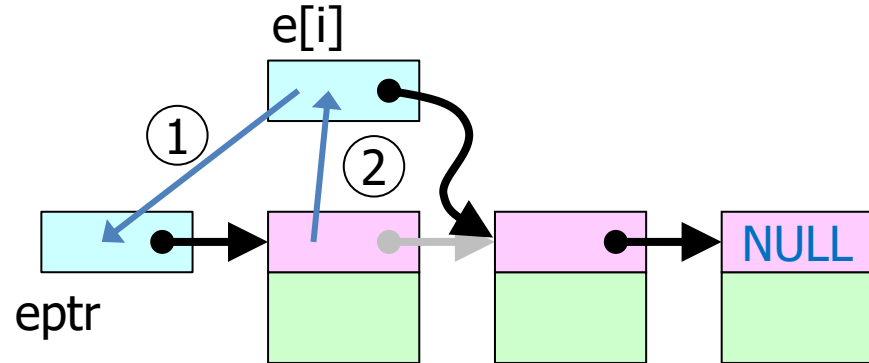
# リストの削除

- 要素のために確保したメモリを解放する



リスト上の要素を先頭から順に削除

```
while( e[i] ){  
    eptr = e[i]; ①  
    e[i] = e[i]->next; ②  
    free( eptr );  
}
```



全ての*i*についてリスト削除  
配列*e*自体も解放(*free*)する

# 改善3: アルゴリズムを変更

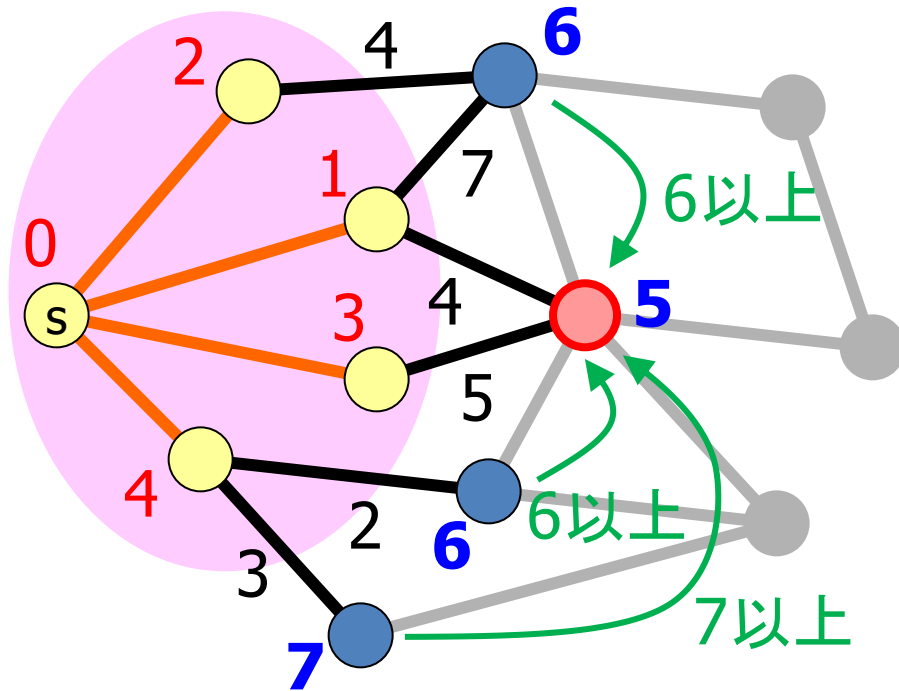
---

- Bellman-Ford法は、枝重みが正、ゼロ、負の場合に最短経路を求める  
(ただし、枝重み和が負のループはない場合)
- 枝重みが正、ゼロ(つまり負でない)に限定される場合は、もっと効率よく最短経路を求めることができる
- 今回のプログラミング演習では、枝重みは正、ゼロのみ



# 非負枝重みの最短経路問題(2)

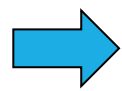
- 枝重みが非負(0または正)なので、他の点を経由したときの経路長は、他の点の経路長以上



現在の最小値5が  
その点の最短経路長



全ての点の最短経路長  
が決まるまで繰り返す



Dijkstra法

計算複雑度  $O(N^2)$  (点数  $N$ )

# Dijkstra法の実装1

```
#define INTINF 999999  ← 正無限大を表す十分大きな値
for(i=0 ; i<N ; i++) determined[i] = 0;  ← 最短経路長決定済みか否か
for(i=1 ; i<N ; i++) u[i] = INTINF;  ← 最短経路長uを無限大に初期化
s = 0;
u[s] = 0;  ← 始点を点0とし、その最短経路長は0とする
determined[s] = 1;

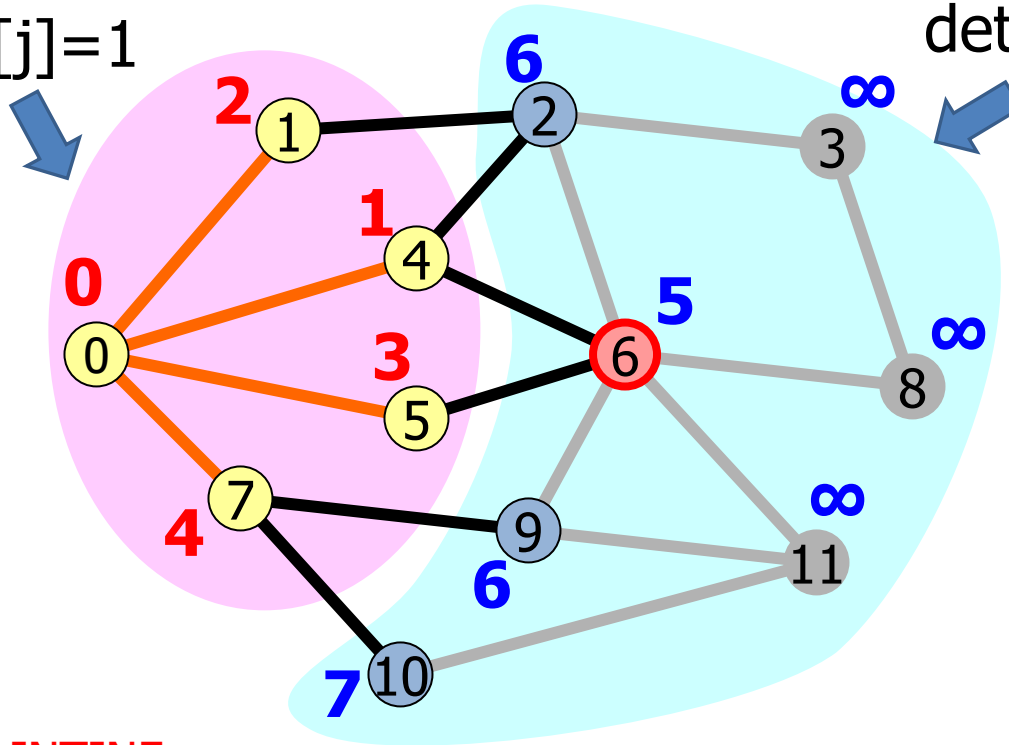
for(i=1 ; i<N ; i++){  ← 残りのN-1個の点について最短経路長を求める
    点sを始点とする枝(s,j)のそれぞれについて{ ①
        L = u[s]+枝(s,j)の重み;  ← Lは枝(s,j)を通り点jへ到達する経路の経路長
        if( u[j] > L ) u[j] = L;
    }
    最短経路長が未確定(determined[j]=0)な点のうち、u[s]が最小の点sを求める; ②
    determined[s] = 1; //点sの最短経路長を確定
}
```



# Dijkstra法の実装1の考え方

最短経路長決定済み  
determined[j]=1

最短経路長未決定  
determined[j]=0



## ② の実装

```
int nMin = INTINF;  
for( j=1 ; j<N ; j++ ){  
    if( determined[j]==0 && nMin>u[j] ){  
        nMin = u[j];  
        s = j;  
    }  
}
```

← 最短経路長が未決定の点の中で、経路長が最短の点を探る

# 改善4: Dijkstra法の実装1の改善

## ■ 2つのループ

```
① for( eptr=e[s] ; eptr!=NULL ; eptr=eptr->next ){  
    j = eptr->nTo;  
    L = u[s]+eptr->nWeight;  
    if( u[j] > L ) u[j] = L;  
}
```

← 点sを始点とする全ての枝について  
繰り返し

```
int nMin = INTINF;  
② for( j=1 ; j<N ; j++ ){  
    if( determined[j]==0 && nMin>u[j] ){  
        nMin = u[j];  
        s = j;  
    }  
}
```

← (0を除く)全ての点について繰り返し

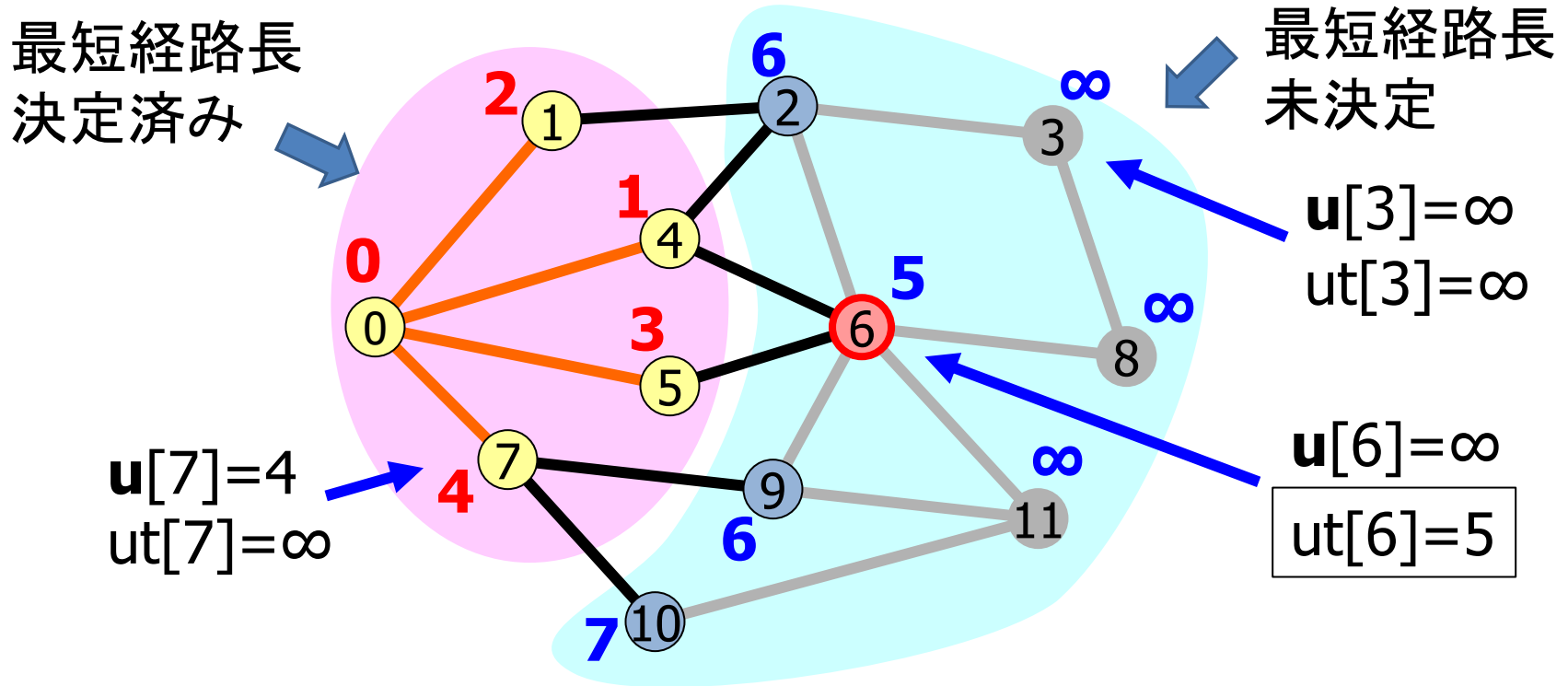
← "determined[j]==0"と"nMin>u[j]"の  
2つの条件判定を実行

一般に、①の繰り返し回数よりも②の繰り返し回数が多い

➡ ②のループ内の処理を簡素化すると高速化

# Dijkstra法の実装2の考え方

- 2つの最短経路長配列を利用  $\begin{cases} u[n]: \text{最短経路長} \\ ut[n]: \text{暫定最短経路長} \end{cases}$



最短経路長決定済みの点については  $ut[n]=\infty$  とする

- ➡  $ut[j]$  が最小の点を探索すれば、経路長未決定の点の中から暫定経路長最短の点を探索することになる

# Dijkstra法の実装2

```
for(i=0 ; i<N ; i++) ut[i] = INTINF; ← 暫定最短経路長ut、および  
for(i=1 ; i<N ; i++) u[i] = INTINF; ← 最短経路長uを無限大に初期化  
s = 0;  
u[s] = 0; ← 始点を点0とし、その最短経路長は0とする
```

```
for(i=1 ; i<N ; i++){ ← 残りのN-1個の点について最短経路長を求める  
  点sを始点とする枝(s,j)のそれぞれについて{ ①  
    L = u[s]+枝(s,j)の重み; ← Lは枝(s,j)を通り点jへ到達する  
    if( u[j]==INTINF && ut[j] > L ) ut[j] = L; ← 経路の経路長  
  } ← 点jは最短経路長が未決定  
  int nMin = INTINF;  
  for( j=1 ; j<N ; j++ ){  
    if( nMin > ut[j] ){ ② ← ループ②の中の処理が実装1より減少  
      nMin = ut[j];  
      s = j;  
    }  
  }  
  u[s] = ut[s]; //点sの最短経路長を確定  
  ut[s] = INTINF  
}
```

# 改善5: 監視員法(sentinel法)

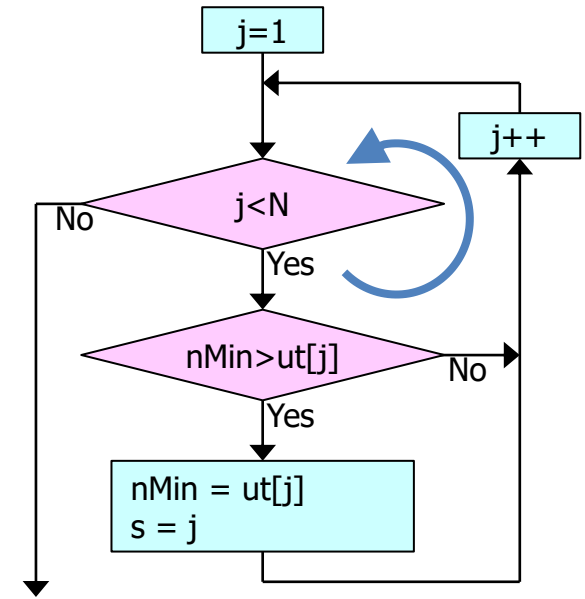
## ■ ループ②に着目

```
int nMin = INTINF;
for( j=1 ; j<N ; j++ ){
    if( nMin > ut[j] ){
        nMin = ut[j]; ← 最小値をnMinに記録
        s = j;
    }
}
```

ほとんどの場合に $j < N$ が成り立ち、次に $nMin > ut[j]$ を調べるので必ず2回比較を実行している

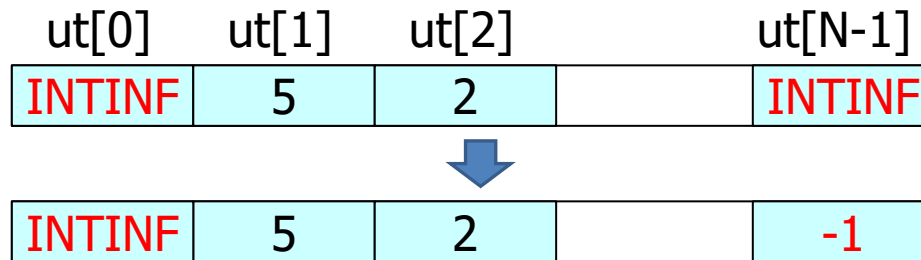


比較の実行回数を減らす工夫として、 $nMin > ut[j]$ が成り立つときにのみ $j < N$ を調べる



# 監視員(sentinel)とは

- ループ最後の  $j == N-1$  のときに必ず  $nMin > ut[j]$  が成り立つようにする



← 十分小さな値  
 $j == N-1$  のとき  
 $nMin > ut[j]$  が  
必ず成立

・・・ループの最後を監視

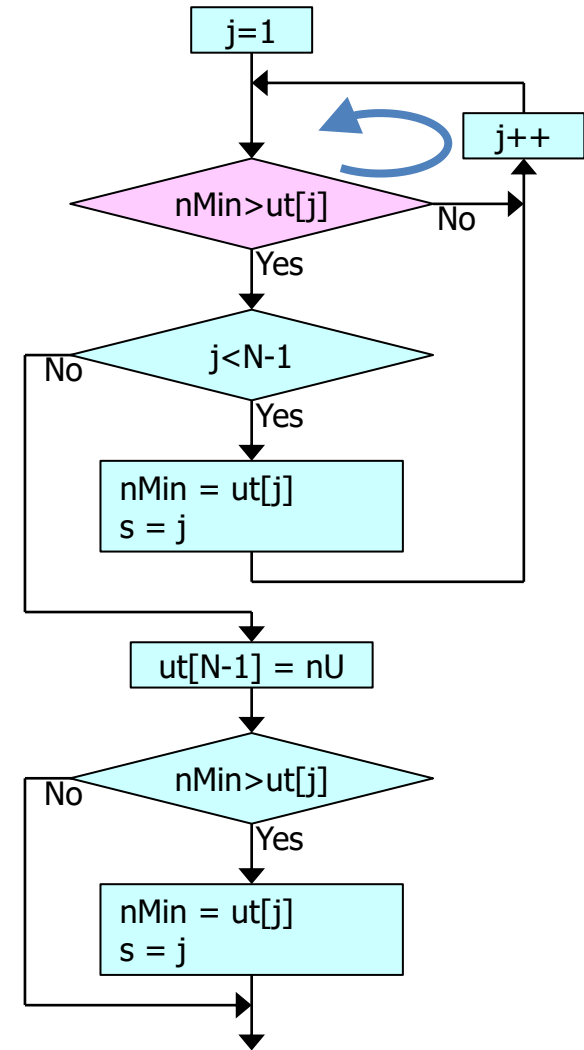
- $nMin > ut[j]$  が成り立つときに、 $j < N-1$  か  $j == N-1$  かに  
よって必要な処理を行う

# 監視員法(sentinel法)の実装

```
int nMin = INTINF;
int nU = ut[N-1];
ut[N-1] = -1;
for( j=1 ; ; j++ ){
    if( nMin > ut[j] ){
        if( j < N-1 ){
            nMin = ut[j];
            s = j;
        }else{
            ut[N-1] = nU;
            if( nMin > ut[j] ){
                nMin = ut[j];
                s = j;
            }
        }
    }
}
```

←  $ut[N-1]$ の値を保存  
← 十分小さな値を設定  
←  $j$ を1ずつ増やしながら繰り返し  
←  $j < N-1$ の場合  
←  $j == N-1$ の場合  
←  $ut[N-1]$ の値を戻す  
← 改めて比較実行

ここに比較が無いのがポイント



→  $nMin > ut[j]$ が成り立つ確率は低いので、平均的に比較実行回数が低減

# 改善6: 最短経路長の候補選択を高効率化

---

- 最短経路長を決定するのを、全点を調べずに済ませたい



- 次の最短経路長を決定する点の候補だけを「集合」に入れ、集合内だけを調べる



# Dijkstra法の実装3

```
for(i=1 ; i<N ; i++) u[i] = INTINF; ← 最短経路長を無限大で初期化
```

```
s = 0;
```

```
u[s] = 0; ← 始点を点0とし、その最短経路長は0とする
```

```
while( 1 ){ ← 無限ループ
```

```
  点sを始点とする枝(s,j)のそれぞれについて{
```

```
    L = u[s]+枝(s,j)の重み; ← Lは枝(s,j)を通り点jへ到達する経路の経路長
```

```
    if( u[j] > L ){
```

```
      u[j] = L; ← 点jの経路長(暫定最短経路長)Lをu[j]に記録
```

```
      点jの経路長Lを、候補集合に追加;
```

```
    }
```

```
  }
```

← 候補がなければ最短経路長が全て求まった

```
  候補集合中の候補がなくなればbreak(終了);
```

```
  候補集合から経路長最短の要素の点をsとする;
```

```
// ここでu[s]は点sについて確定した最短経路長;
```

```
  候補集合から点sの要素を取り除く;
```

```
}
```

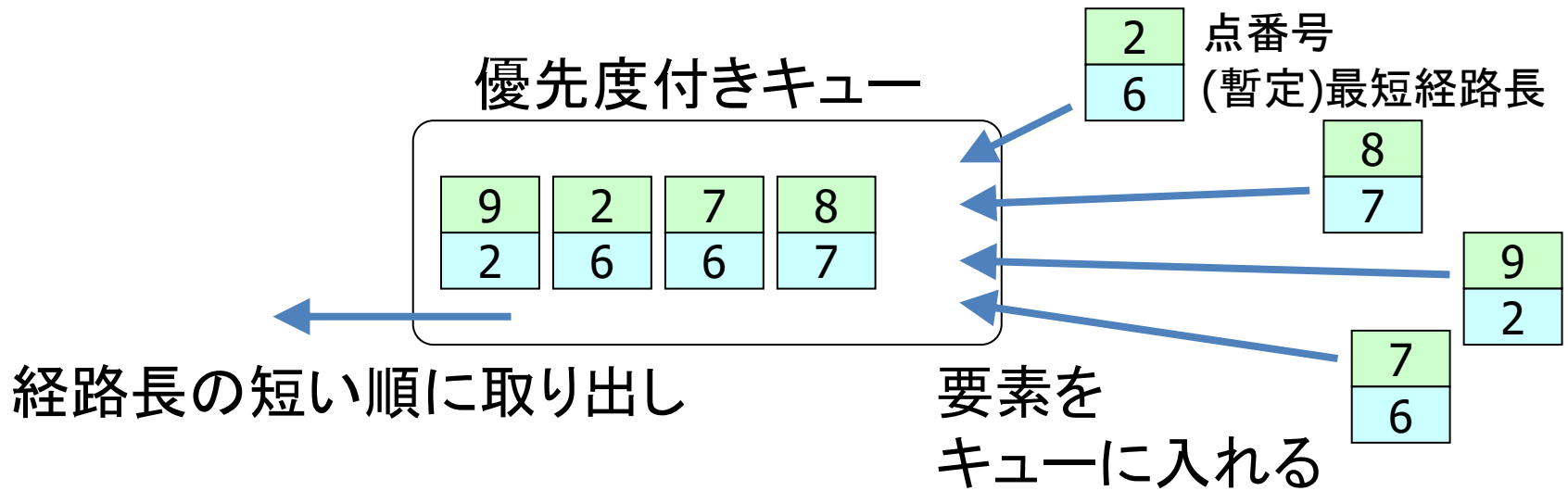
# 改善6: 最短経路長の候補選択を高効率化

- 最短経路長を決定するのを、全点を調べずに済ませたい



- 次の最短経路長を決定する点の候補だけを「集合」に入れ、集合内だけを調べる
- 優先度付きキュー(Priority Queue)を利用
  - 「待ち行列」・・・複数の要素を入れ、1つずつ取り出し
  - キューの中から経路長が小さい順に要素を取り出す

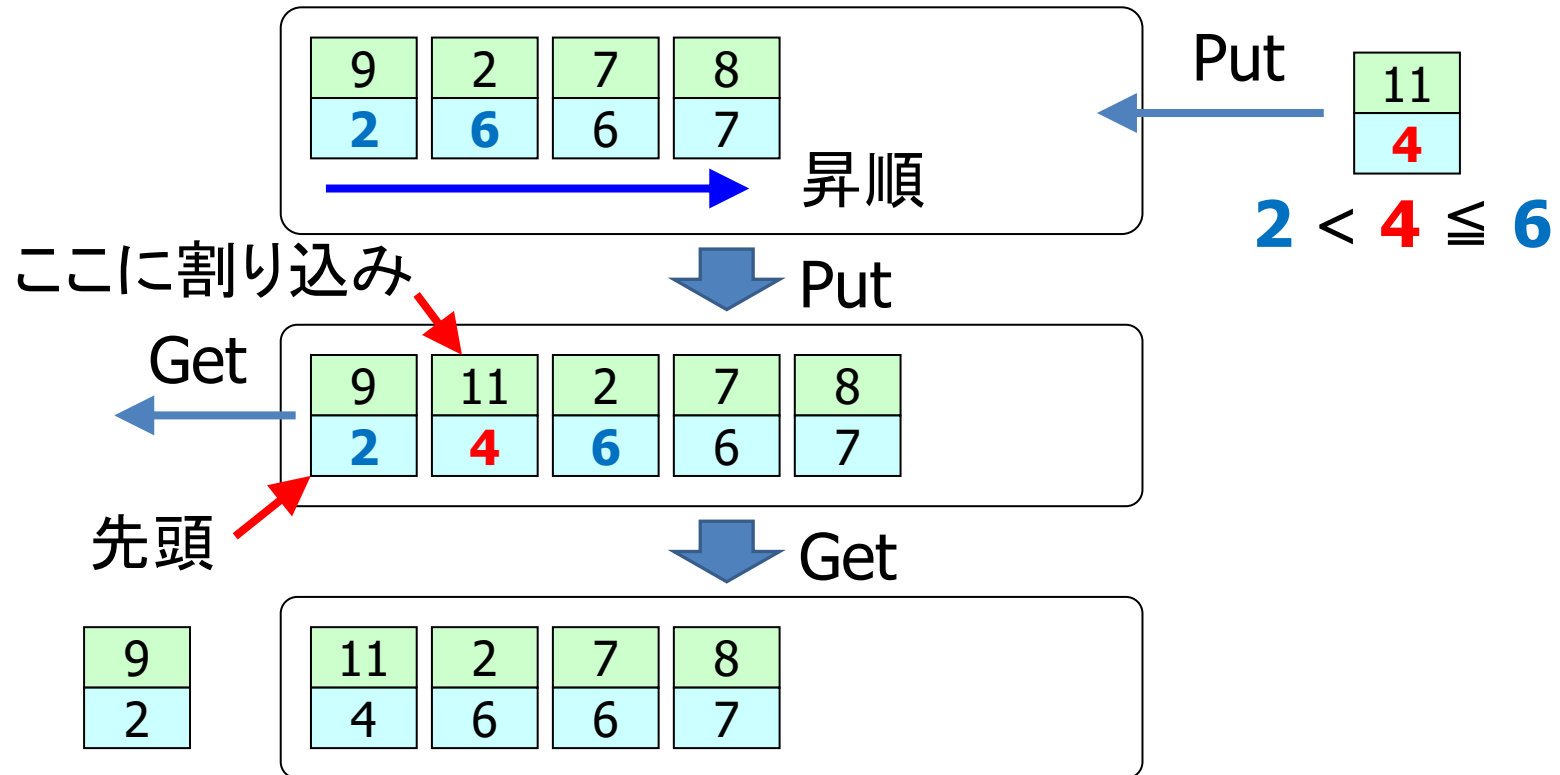
# 優先度付きキュー



- 操作Put: 要素を1つ入れる
- 操作Get: 優先度順に要素1つを取り出し
- 状態取得IsEmpty: キューが空(要素数0)か調べる

# 優先度付きキュー

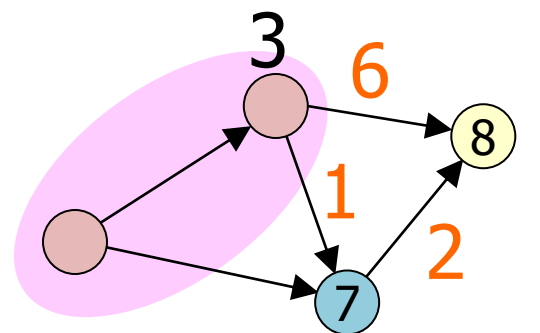
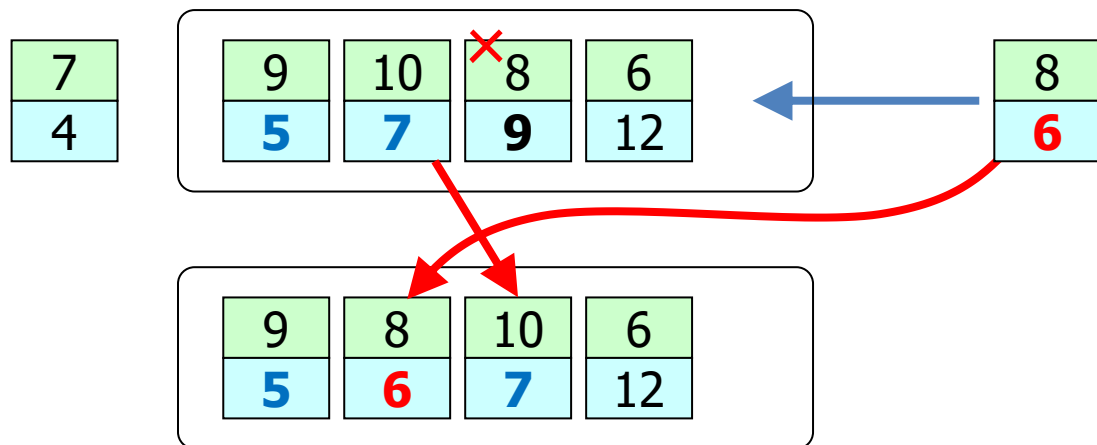
- 優先度順(経路長の短い順)に取り出すため、キュー内では要素を経路長の昇順に記録
  - 要素をPutする際に昇順になるように要素を割り込み
  - Getする際は先頭の要素を取り出し



# 優先度付きキュー

- Dijkstra法での利用では、Putは要素入れ替えを考慮する必要あり
  - すでにキューに入っている点の最短経路長が更新される場合がある

**例** 点8は経路長決定候補であり、暫定経路長=9  
点7の最短経路長4が決定し、点8の経路長が6に減少



経路長4確定

➡ 点8の要素をいったん削除し、改めて挿入

# キューを実装するためのデータ構造

- 要素を順序付けて管理したい(暫定最短経路長の昇順)
- 要素の追加と削除を簡単に行いたい

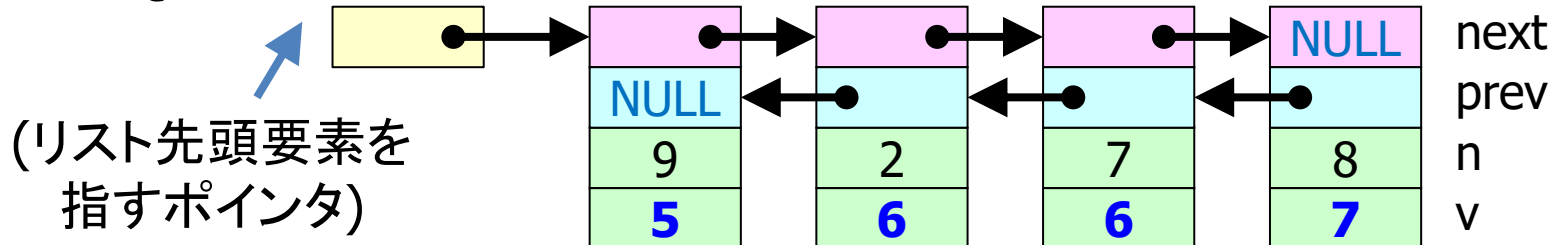


双方向リスト  
を利用

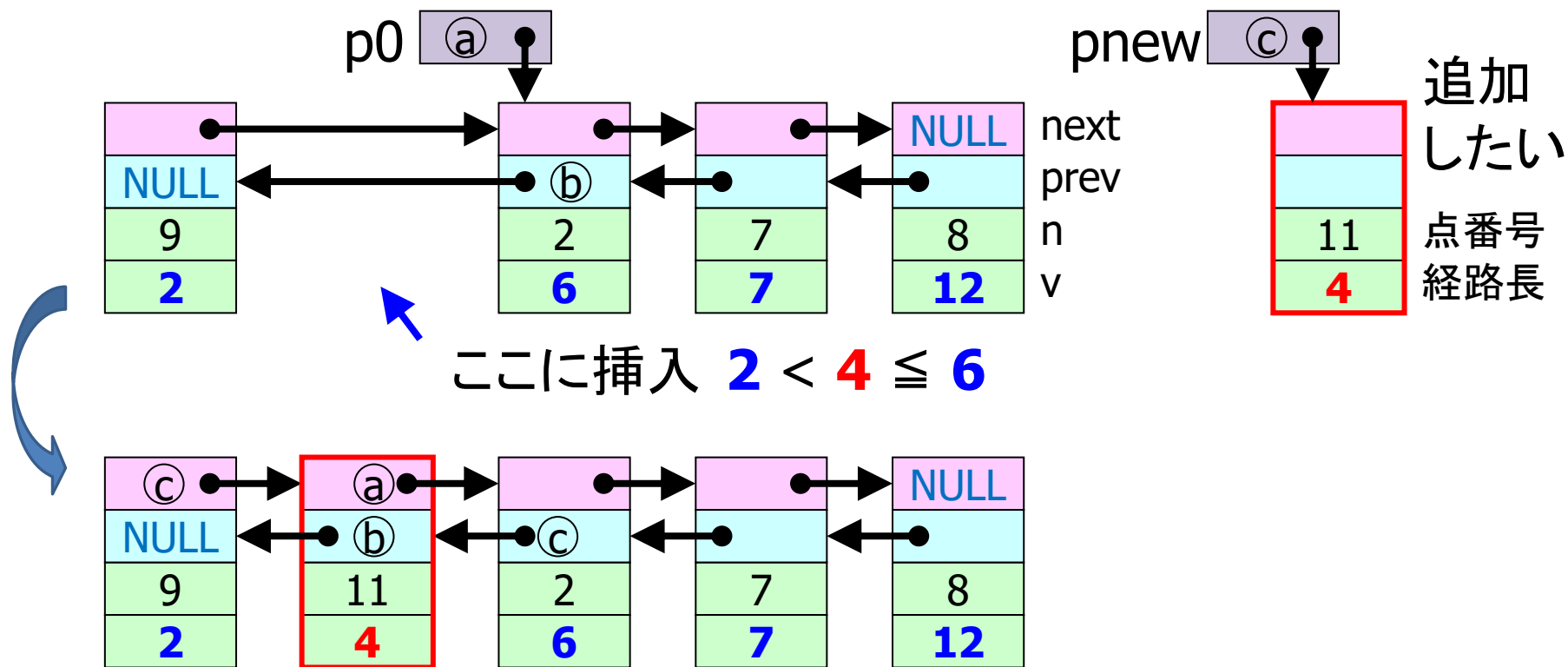
```
typedef struct QElementRec {  
    struct QElementRec *next; // 次要素  
    struct QElementRec *prev; // 前要素  
    int n; // 点番号  
    int v; // 最短経路長  
} QELEMENT;
```

リスト要素の定義 →

listQueueElement



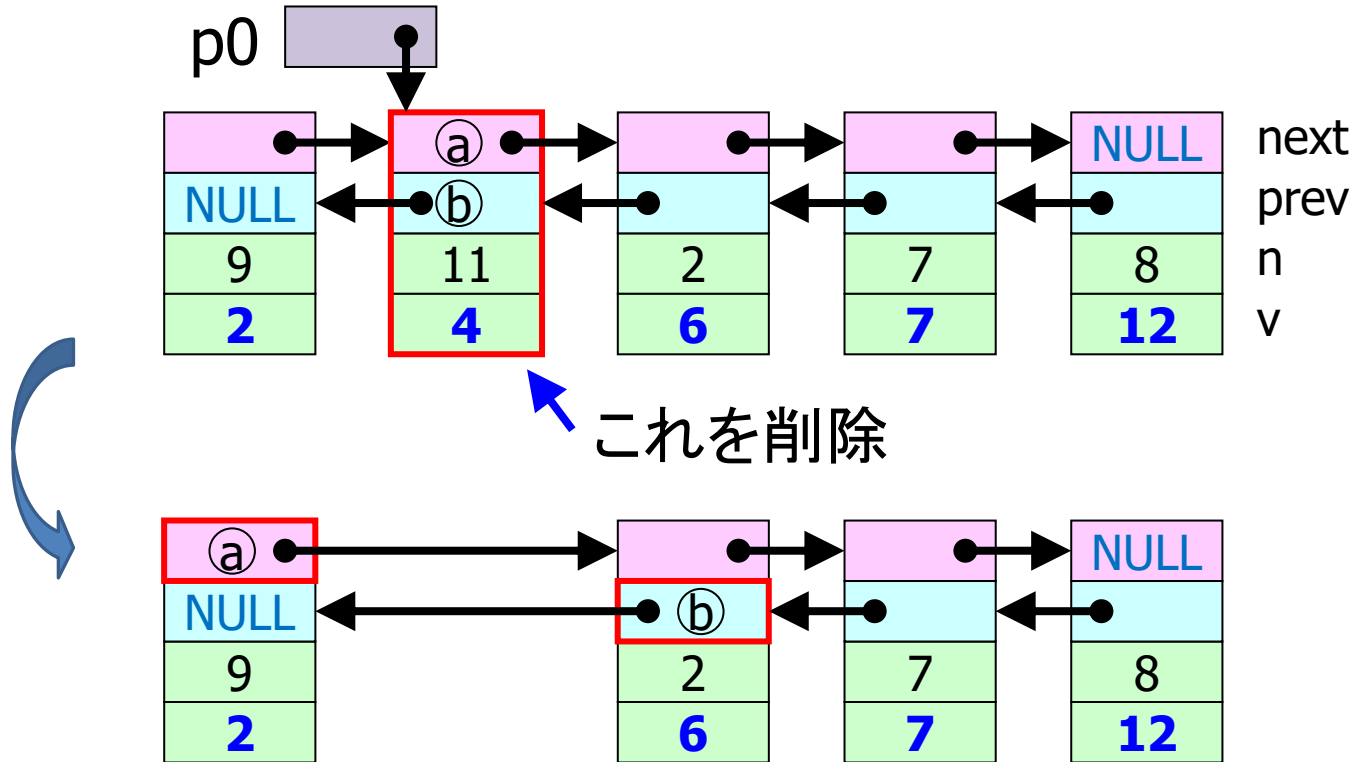
# 双方向リストの操作: 要素を追加



```

pnew->next = p0;           (a)
pnew->prev = p0->prev;    (b)
p0->prev->next = pnew;   (c)
p0->prev = pnew;         (c)
    
```

# 双方向リストの操作: 要素を削除



$p0 \rightarrow prev \rightarrow next = p0 \rightarrow next;$  ①  
 $p0 \rightarrow next \rightarrow prev = p0 \rightarrow prev;$  ②



# リストの要素追加/削除の位置の考慮

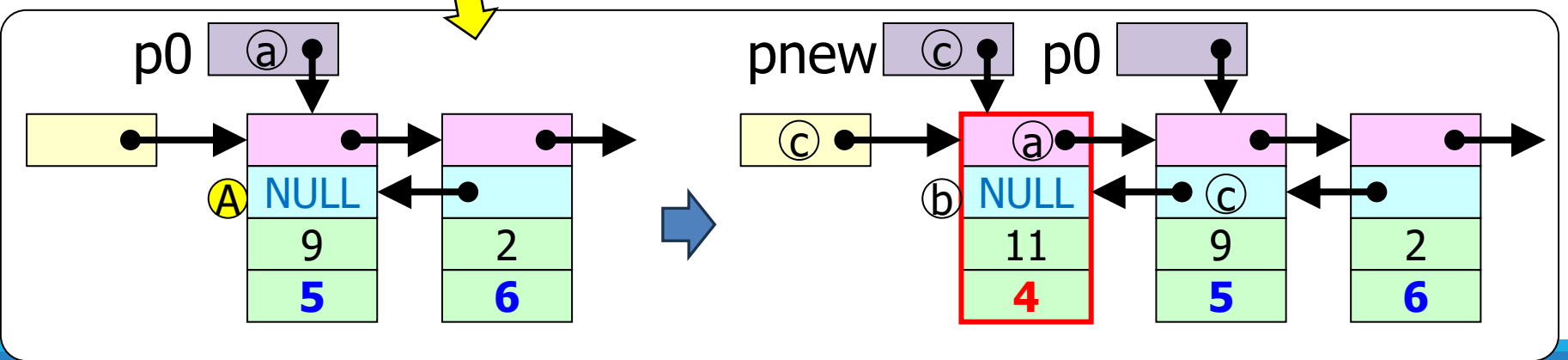
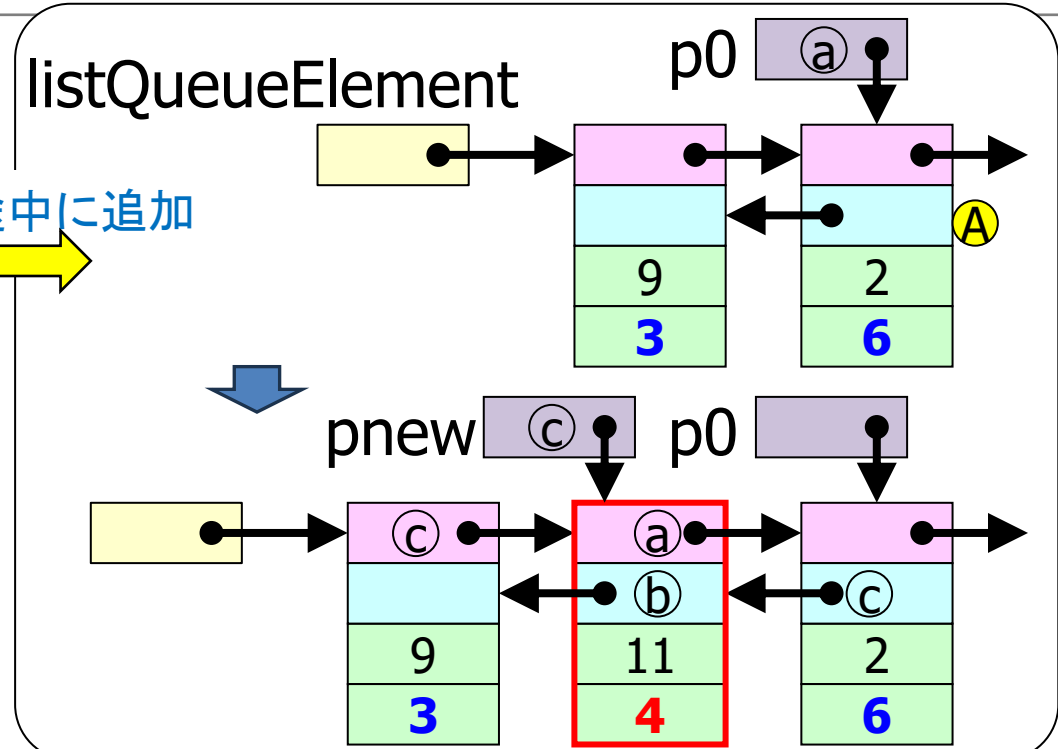
## ■ リストに要素を追加する場合

```

if( p0->prev != NULL ){
    pnew->next = p0;           (a)
    pnew->prev = p0->prev;    (b)
    p0->prev->next = pnew;    (c)
    p0->prev = pnew;          (c)
} else {
    pnew->next = p0;           (a)
    pnew->prev = NULL;        (b)
    listQueueElement = pnew; (c)
    p0->prev = pnew;          (c)
}
    
```

途中に追加

先頭に追加



➡ 場合分けが必要

# 双方向リストの前後にダミー要素追加

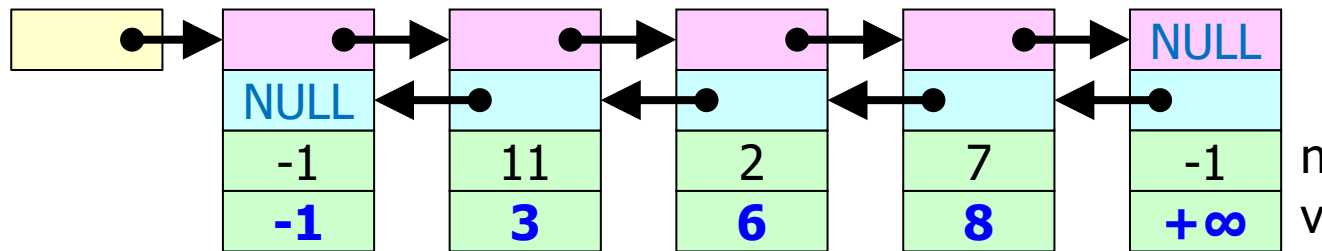
- リストの先頭・末尾の場合に要素の追加・削除はリスト途中の場合とは異なった操作になる



- 場合分けを避けるため、リストの前後にダミー要素を置く → 常にリスト途中の要素追加・削除となる

(リスト先頭のダミー要素を指すポインタ)

listQueueElement



先頭ダミー要素  
QueueHead

末尾ダミー要素  
QueueTail

# リスト上の削除対象要素を検索

- ある点番号  $n$  の要素をリスト上で検索し、見つければ削除

```
p0 = listQueueElement->next; // listQueueElementはHeadダミー要素を指す  
// 本来の先頭要素はlistQueueElement->next
```

```
while( p0 != NULL ){  
    if( p0->n == n ){ ← 点番号nの要素がリストに存在  
        p0->prev->next = p0->next; ← 点番号nの要素をリストから削除  
        p0->next->prev = p0->prev;  
        free( p0 ); ← 要素のメモリ領域を解放  
        break;  
    }  
    p0 = p0->next;  
}
```

★発展: ループの中に比較が2つある

⇒ 監視員法(sentinel法)を利用して高速化が可能

# 優先度付きキューのクラスを定義

## ■ クラス `CPQueue` (Priority Queue)

- コンストラクタ `CPQueue(N)`  
最大要素数Nのキューを作成
- 操作 `Put(int n, int L)`  
点n、その暫定最短経路長Lをキューに入れる
- 操作 `Get(void)`  
キューの最大優先度(経路長最短)の点番号を返すとともに、その要素をキューから削除する
- 属性取得 `IsEmpty(void)`  
キューが(先頭、末尾のダミー要素は除いて)空か否かを関数値として返す

(オブジェクト指向(C++))

# クラスCPQueue

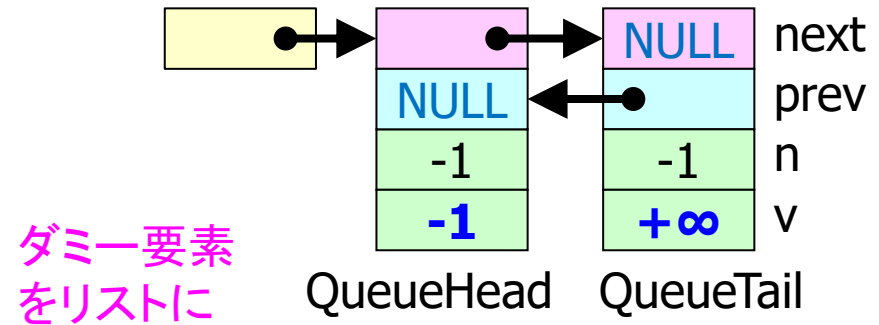
■ メンバ変数 `QELEMENT *listQueueElement;`  
`QELEMENT QueueHead, QueueTail;` ← ダミー要素(先頭、末尾)  
`int nCountElement;`

## ■ コンストラクタ

`CPQueue::CPQueue(void)`

```
{  
    QueueHead.n = -1;  
    QueueHead.v = -1;  
    QueueHead.next = &QueueTail;  
    QueueHead.prev = NULL;  
    QueueTail.n = -1;  
    QueueTail.v = INTINF;  
    QueueTail.next = NULL;  
    QueueTail.prev = &QueueHead;  
    listQueueElement = &QueueHead; ← リスト先頭を指す  
    nCountElement = 0; ← (ダミー要素を除く)要素数を0に初期化  
}
```

listQueueElement



ダミー要素  
をリストに

# クラスCPQueue

## ■ 操作 Put

```
void CPQueue::Put(int n, int L)
```

```
{
```

```
    QELEMENT *pnew=NULL, *p0 = listQueueElement->next;
```

```
    while( p0 != NULL ){
```

```
        if( p0->n == n ){ // キュー中にnの要素があれば削除する
```

```
            p0->prev->next = p0->next;    p0->next->prev = p0->prev;
```

```
            nCountElement--; pnew = p0;
```

```
            break;
```

```
        }
```

```
        p0 = p0->next;
```

```
    }
```

```
    if( pnew == NULL ) pnew = (QELEMENT *)malloc( sizeof(QELEMENT) );
```

```
    pnew->n = n; pnew->v = L; // n, Lを昇順の位置に挿入する
```

```
    p0 = listQueueElement->next;
```

```
    while( L > p0->v ) p0 = p0->next;
```

```
    pnew->next = p0;    pnew->prev = p0->prev;
```

```
    p0->prev->next = pnew;    p0->prev = pnew;
```

```
    nCountElement++; ← 要素数を1増加
```

```
}
```

要素p0(点番号n)をリストから削除

要素数を1減少

要素のメモリ領域を再利用

要素のメモリ領域を割り当て

} pnewをp0の前に挿入

# クラスCPQueue

## ■ 操作 Get

```
int CPQueue::Get(void)
```

```
{
```

```
    QELEMENT *p0 = listQueueElement->next;    // 先頭要素
```

```
    int n = p0->n;
```

```
    // 先頭要素を削除
```

```
    p0->prev->next = p0->next;
```

```
    p0->next->prev = p0->prev;
```

```
    if( nCountElement > 0 ) free( p0 );
```

```
    nCountElement--;
```

```
    return n;
```

```
}
```

← 削除する要素のメモリ領域を解放

nCountElement(要素数)が0ならば  
nCountElement==0は非ゼロ(真)、  
nCountElementが0でなければ  
nCountElement==0はゼロ(偽)

## ■ 属性取得 IsEmpty

```
int CPQueue::IsEmpty(void)
```

```
{
```

```
    return nCountElement == 0;
```

```
}
```

## ■ デストラクタ

```
int CPQueue::~~CPQueue(void)
```

```
{
```

```
    // 何もする必要なし
```

```
}
```

# Dijkstra法の実装3

```
#include "CPQueue.h"
```

```
CPQueue pq();
```

 ← 優先度付きキューのオブジェクトを作成

```
for(i=1 ; i<N ; i++) u[i] = INTINF;
```

オブジェクト名(変数名): pq

```
s = 0;
```

```
u[s] = 0;
```

```
while( 1 ){
```

点sを始点とする枝(s,j)のそれぞれについて{

```
    L = u[s]+枝(s,j)の重み;
```

```
    if( u[j] > L ){
```

```
        u[j] = L;
```

```
        pq.Put( j, L );
```

点jについて、より小さい暫定最短経路長が得られたら、キューに入れる(入れ替える)

```
    }
```

キューが空、すなわち候補がなければ最短経路長が全て求まった

```
}
```

```
if( pq.IsEmpty() ) break;
```

```
s = pq.Get();
```

← キューの先頭要素を取り出し、点番号をsに読み取り

```
}
```



# 終わりに

---

- 候補挿入位置を配列先頭から探索しているが、既存候補は経路長が昇順になっているので二部探索など効率よい探索手法が使えるはず
  - それに適したデータ構造は?

# 発展

---

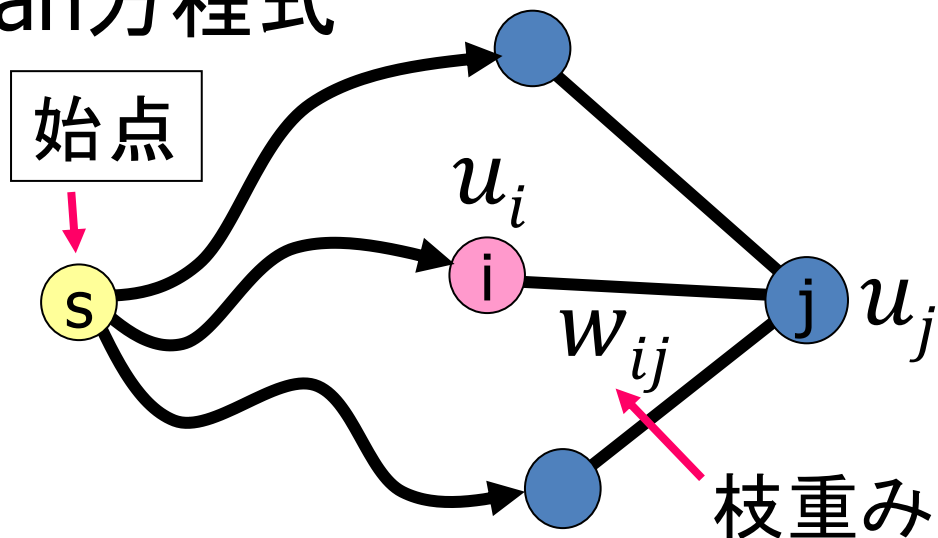
- 最短経路長を与える経路を知るには？
- 2番目の最短経路、3番目の最短経路は？

# 経路長最短の経路を表示する(発展)

最短経路に関するBellman方程式

$$u_s = 0$$

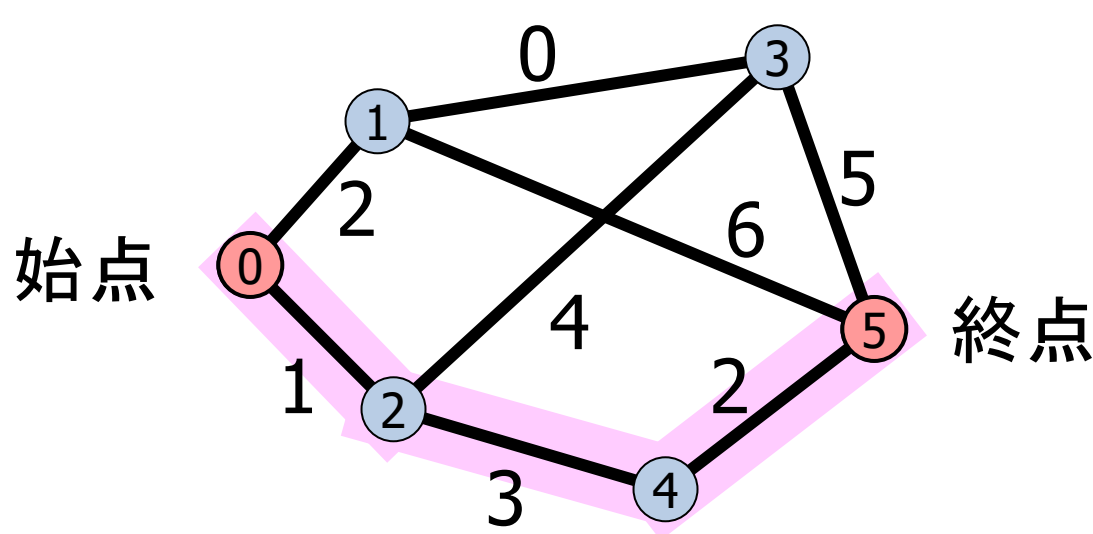
$$u_j = \min_i \{u_i + w_{ij}\}$$
$$\forall j \neq s$$



最短経路上で直前の点がどれかが分かる

➡ 配列 `prev[j]` を用意し、`prev[j] = i` を記録

# 経路長最短の経路を表示する(発展)



prev[0] = -1  
prev[1] = 0  
prev[2] = 0  
prev[3] = 1  
prev[4] = 2  
prev[5] = 4

終点から順に直前点をたどる

終点番号 5

→ 4 = prev[5] → 2 = prev[4] → 0 = prev[2]



経路は 5 ← 4 ← 2 ← 0 ……逆順になっている

# 経路長最短の経路を表示する(発展)

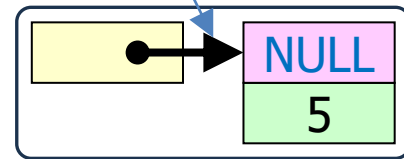
点番号  $j = 5$ ; // 経路の終点

```
while( j >= 0 ){  
    点jを経路として記録;  
    j = prev[j];  
}
```

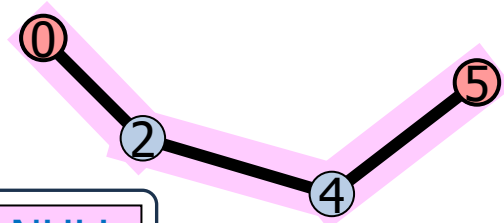
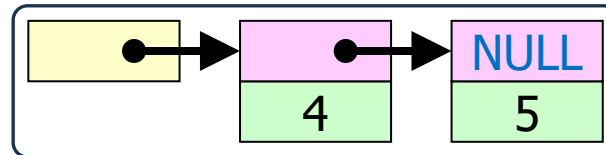
prev[0] = -1 ... 始点まで  
たどれば終了

リスト要素追加

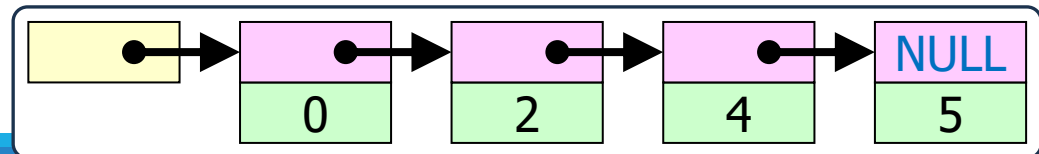
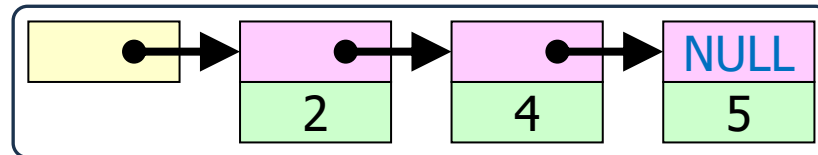
listSPath  
NULL



リストへの追加  
により逆順化



リストの先頭から  
順に表示



# 経路長最短の経路を表示する(発展)

```
int *prev = (int *)malloc( N*sizeof(int) );
```

最短経路を求める  
配列prevに直前点を記録



配列prevを基に最短経路上の  
点をリストに逆順に記録



リスト上の点を順に表示



リスト要素と配列prevのメモリ解放