最短経路問題

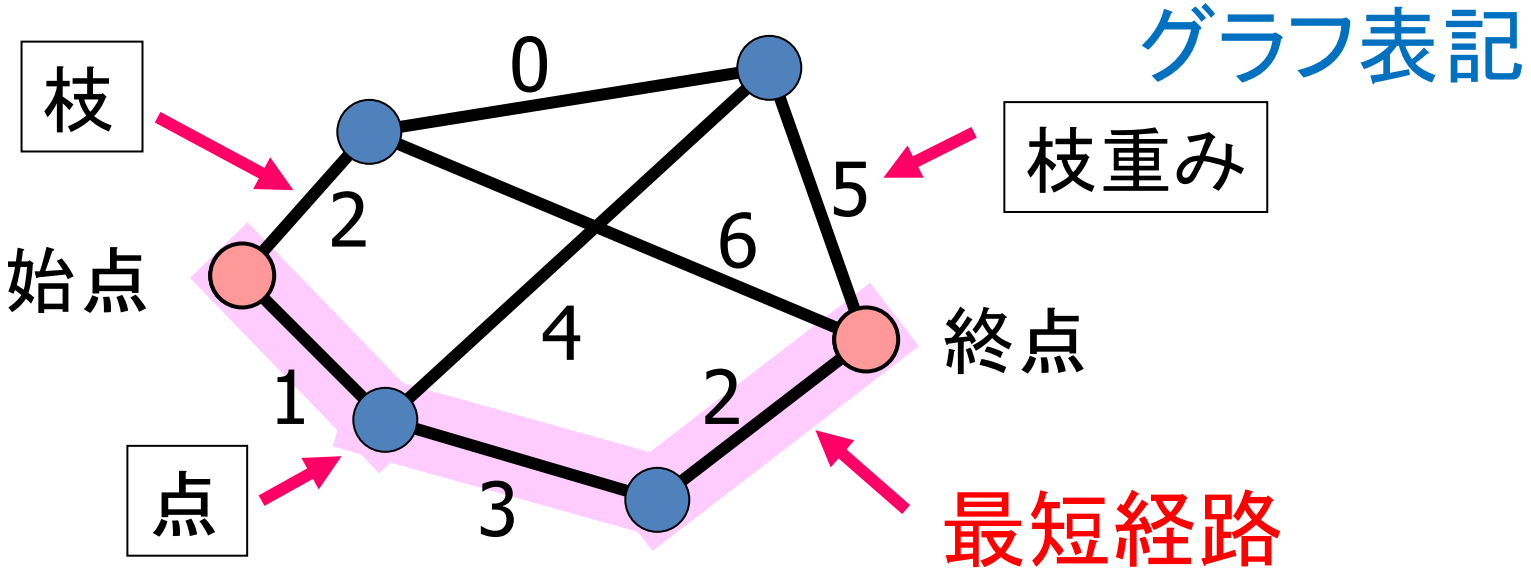
オープンラボ　C言語プログラミング　課題

電気電子物理工学科伊藤研究室

2024年11月5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



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;
        }
    }
    候補集合中の候補がなくなればbreak(終了);   ← 候補がなければ最短経路長が全て求まった
    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;
}
```

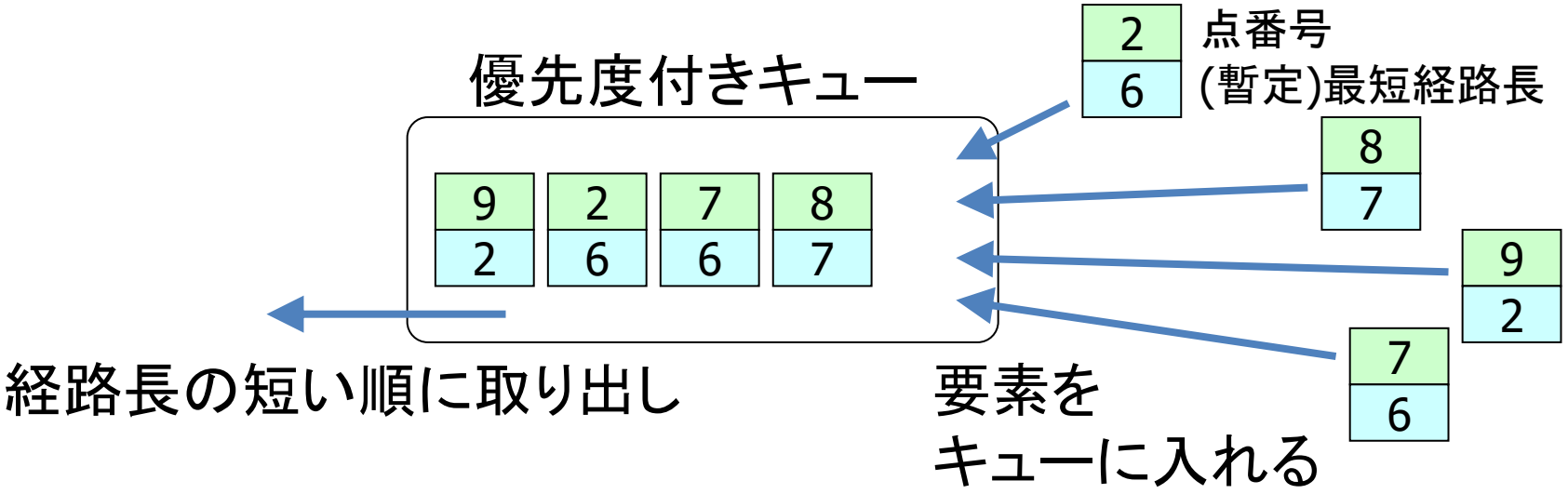
改善6: 最短経路長の候補選択を高効率化

- 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

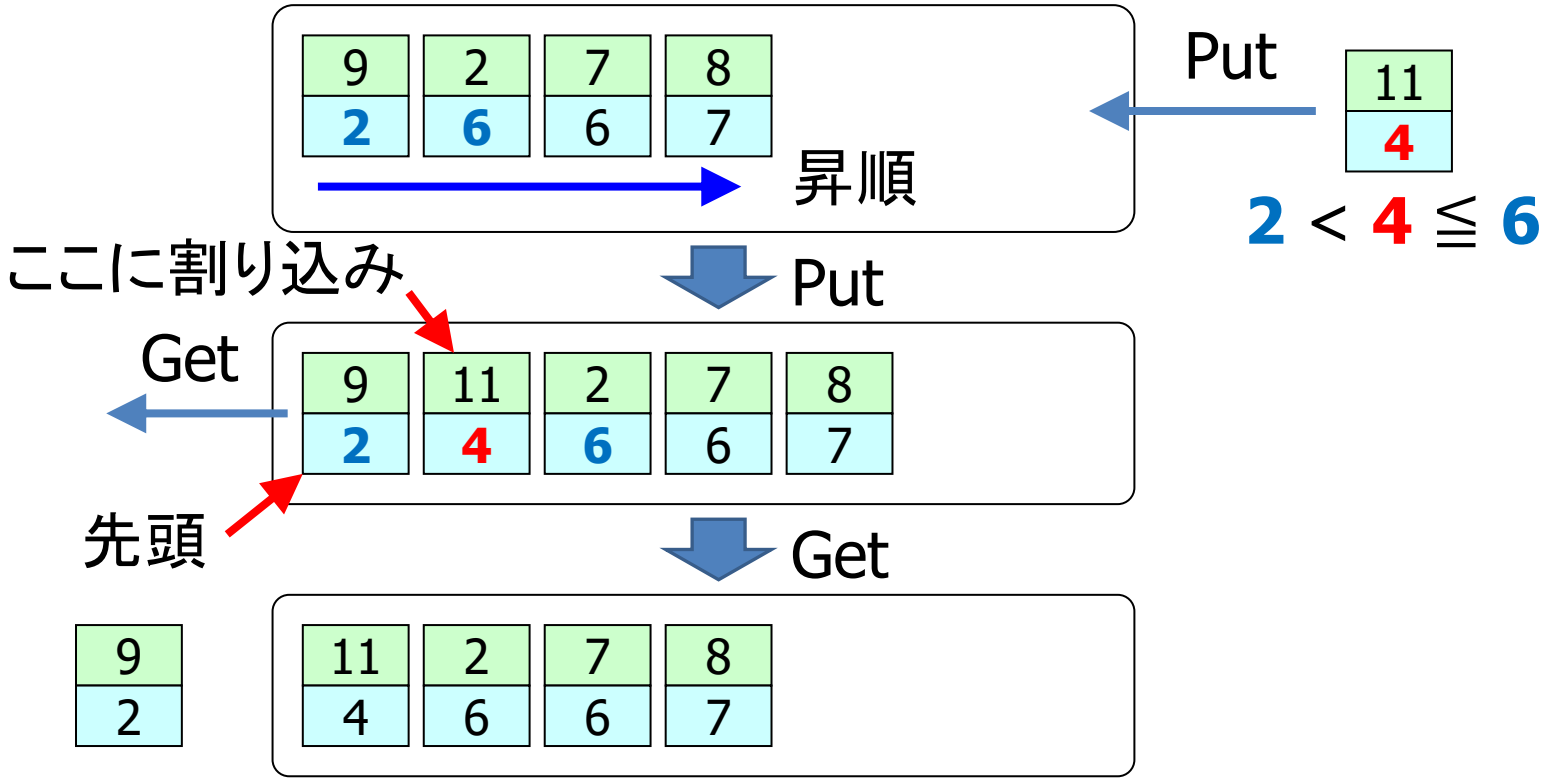
優先度付きキュー



- 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

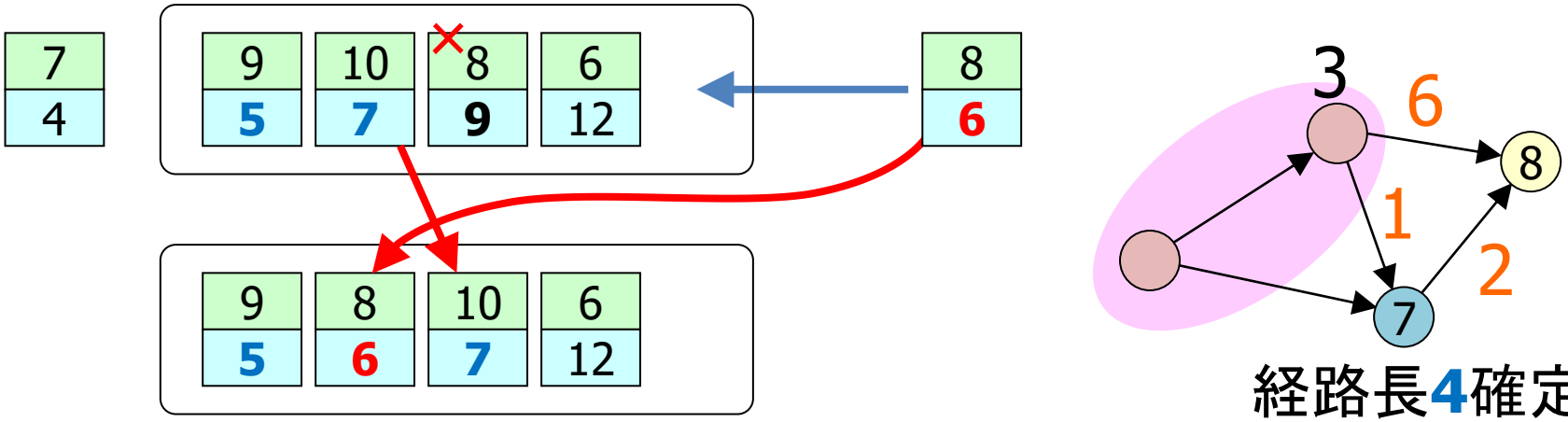


優先度付きキュー

- 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



➡ 点8の要素をいったん削除し、改めて挿入

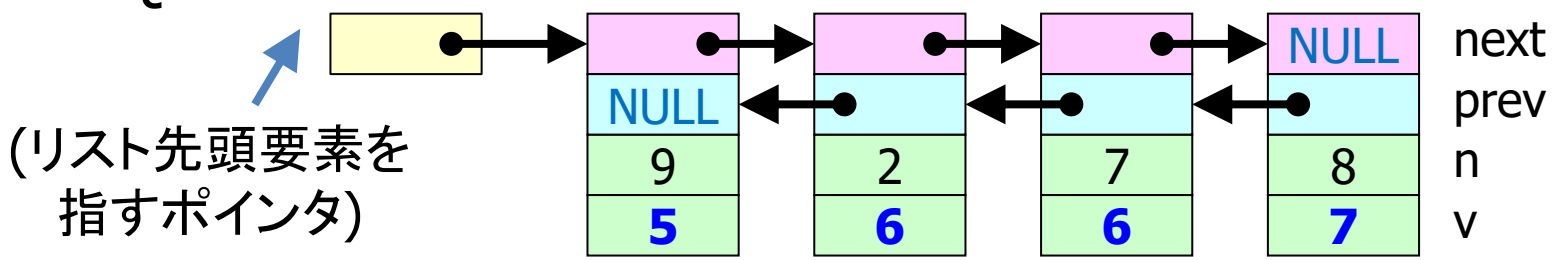
キューを実装するためのデータ構造

- 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

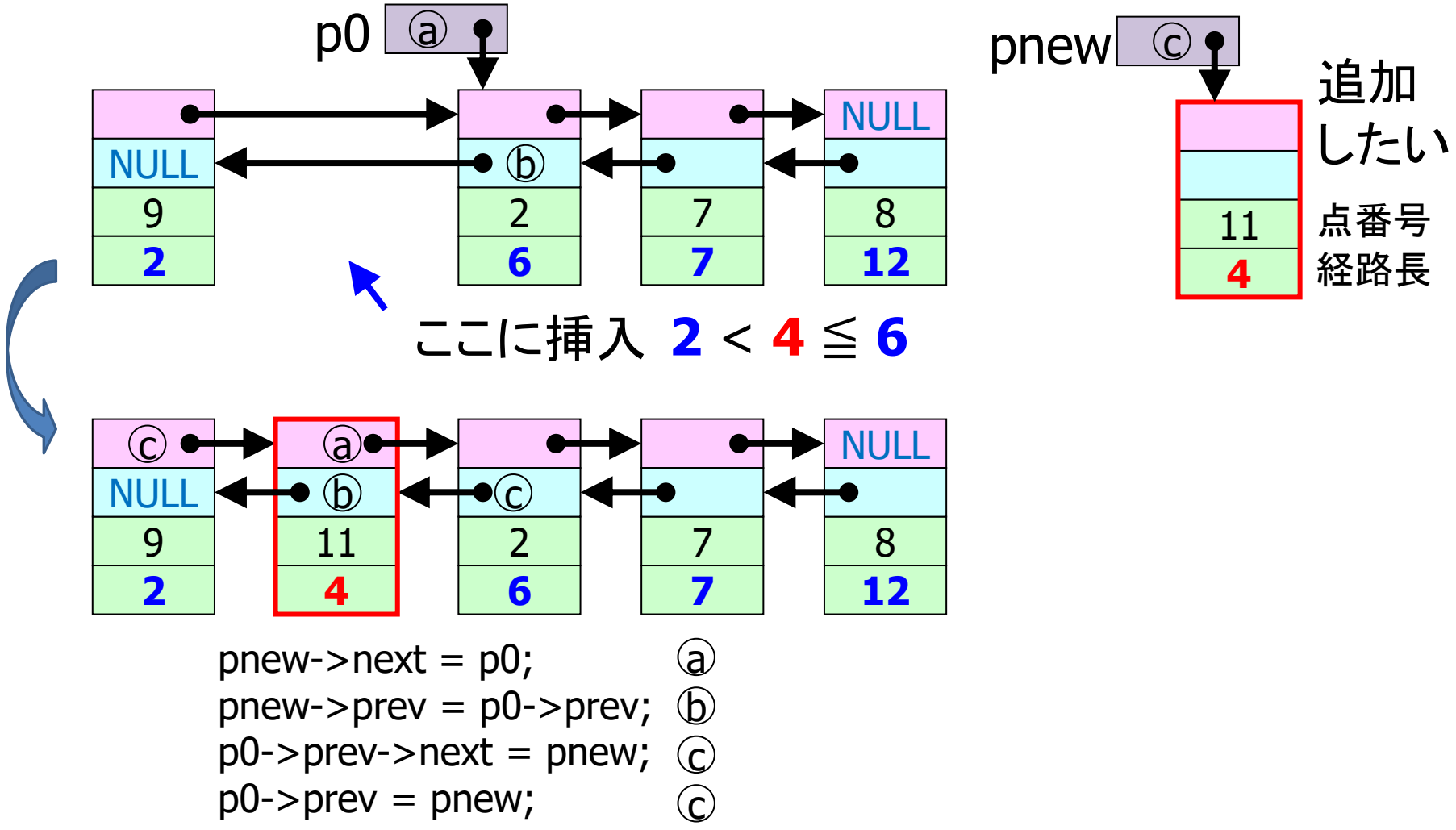
双方向リストを利用

リスト要素の定義

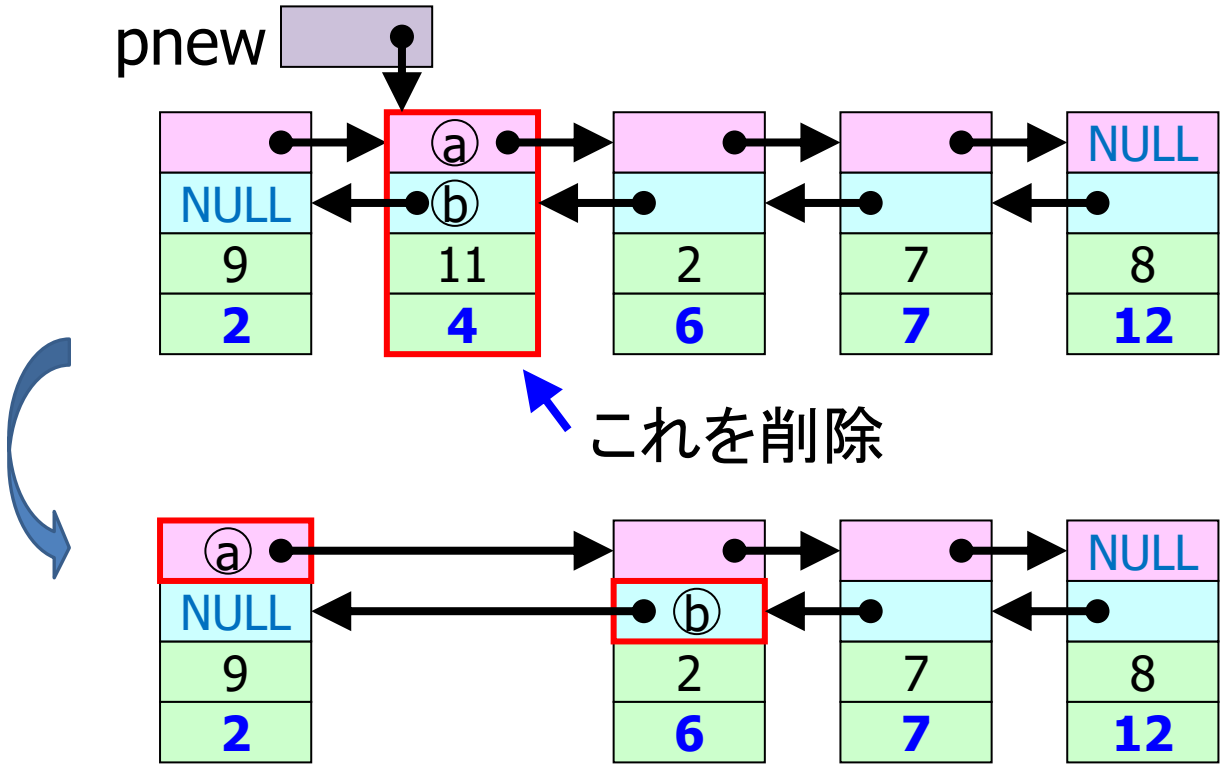
```
typedef struct QElementRec {
    struct QElementRec *next; // 次要素
    struct QElementRec *prev; // 前要素
    int n;          // 点番号
    int v;          // 最短経路長
} QELEMENT;
```



要素を追加: 途中に挿入


p0
ⓐ
NULL
9
2
ⓑ
2
6
7
7
NULL
8
12
pnew
ⓒ
追加
したい
11
4
点番号
経路長
ここに挿入 2 < 4 ≦ 6
ⓒ
NULL
9
2
ⓐ
ⓑ
11
4
ⓒ
2
6
7
7
NULL
8
12
pnew->next = p0; ⓐ
pnew->prev = p0->prev; ⓑ
p0->prev->next = pnew; ⓒ
p0->prev = pnew; ⓒ

双方向リストの操作: 要素を削除



pnew->prev->next = pnew->next; ⓐ
pnew->next->prev = pnew->prev; ⓑ

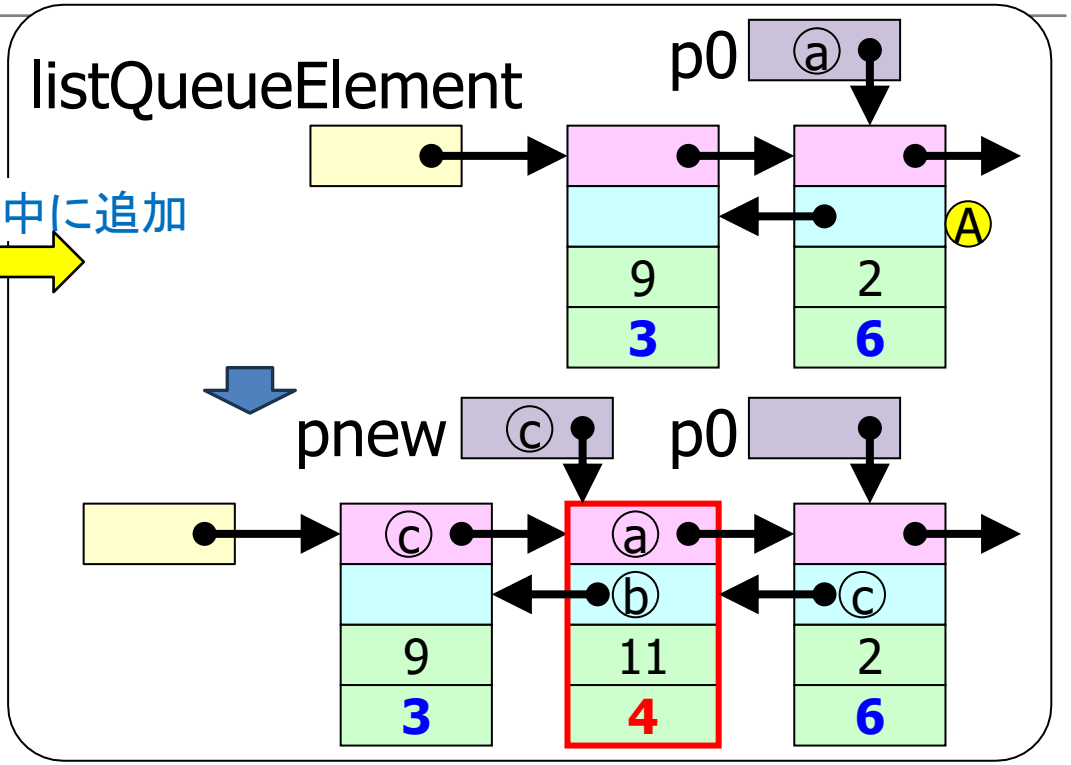
リストの要素追加/削除の位置の考慮

■リストに要素を追加する場合

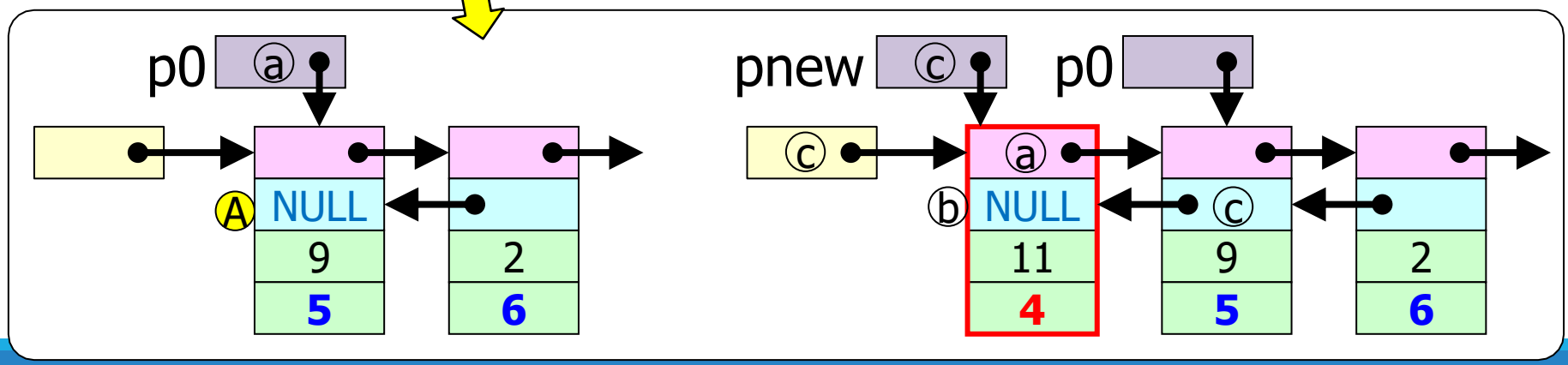
```
if( p0->prev != NULL ){
    pnew->next = p0;              ⓐ
    pnew->prev = p0->prev;        ⓑ
    p0->prev->next = pnew;        ⓒ
    p0->prev = pnew;              ⓒ
}else{
    pnew->next = p0;              ⓐ
    pnew->prev = NULL;            ⓑ
    listQueueElement = pnew;      ⓒ
    p0->prev =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

Ⓐ

途中に追加



先頭に追加



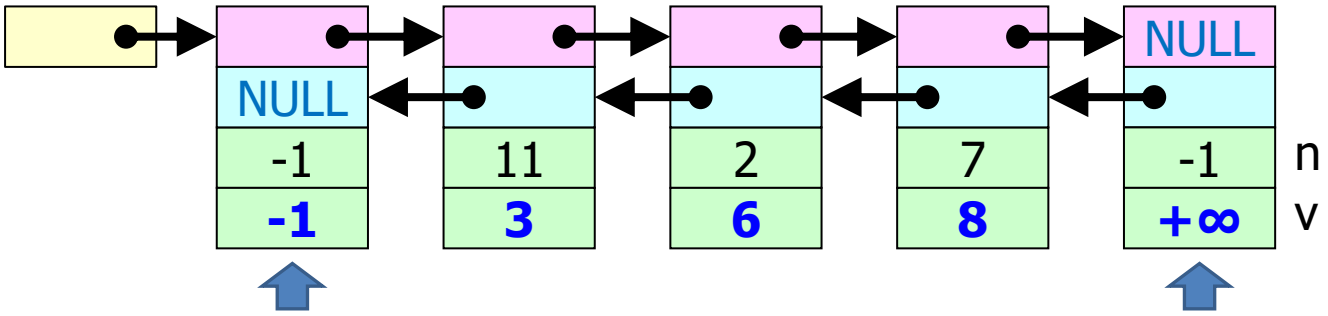
➡ 場合分けが必要

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



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、見つかれば削除

```
p0 = listQueueElement->next;  // listQeueuElementはHeadダミー要素を指す
                              // 本来の先頭要素はlistQeueuElement->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;
        p0->next->prev = p0->prev;   ← 点番号nの要素をリストから削除
        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(void)
    空(ダミー要素のみ)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;
int nCountElement;
```

← ダミー要素(先頭、末尾)

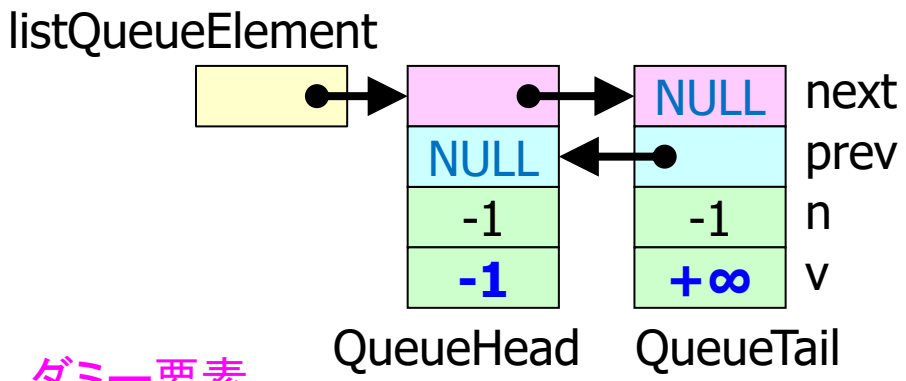
■ コンストラクタ

```
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;
    nCountElement = 0;
}
```

ダミー要素をリストに

← リスト先頭を指す

← (ダミー要素を除く)要素数を0に初期化



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

要素数を1増加

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;
for(i=1 ; i<N ; i++) u[i] = INTINF;
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 );
        }
    }
    if( pq.IsEmpty() ) break;
    s = pq.Get();
}
```

優先度付きキューのオブジェクトを作成

オブジェクト名(変数名): pq

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

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?