# Chapter-4 C++標準テンプレートライブラリの利用(上級)～

## 4.4 双方向リスト listの特徴

### 4.4.1 処理速度

listの代表的な関数の処理速度は次のようになります。

1. 要素の追加。O(1)
2. 要素へのアクセス時間がO(M)
3. 要素の検索時間がO(N)
4. 要素の削除の時間がO(1)

#### 1. 要素の末尾への追加。O(1)

末尾への要素の追加は、要素の数にかかわらず一定。追加するノードの連結を設定するだけです。
次のプグラムを見てください。

```
void PushBack(T value)
{
    //新しいノードを確保。
    Node<T>* newNode = new Node<T>;
    newNode->value = value;
    if (m_begin == nullptr) {
        //先頭ノードが空
        m_begin = newNode;
        m_end = m_begin;
    }
    else {
        //終端ノードの末尾に新しいノードを接続する。
        //まず新しいノードの前のノードを、古い終端ノードにする。
        newNode->prev = m_end;
        //古い終端ノードの次のノードを新しいノードにする。
        m_end->next = newNode;
        //終端ノードを更新。
        m_end = newNode;
    }
}
```

#### 2.要素へのアクセス時間がO(N)

先頭ノードから順を追ってアクセスするしか方法がないため、アクセス時間はO(N)。 本家のstd::listには存在しないが、あえてoperator[]を実装するなら次のようになる。

```
T& operator[](int i)
{

    int count = 0;

    Node<T>* node = m_begin;
    Node<T>* targetNode = nullptr;
    while (node != nullptr) {
        if (count == i) {
            //目的に要素に到達した。
            targetNode = node;
            break;
        }
        count++;
    }
    targetNode->value;
}
```

### 3.要素の検索時間がO(N)

これは可変長配列と同じで先頭から調べていくしかないので、O(N)となる。

```
Node<T>* Find(T value)
{
    for (
        Node<T>* node = m_begin;
        node != nullptr;
        node = node->next
        )
    {
        if (node->value == value) {
            //発見。
            return node;
        }
    }
    return nullptr;
}
```

### 4.要素の削除時間がO(1)

要素の削除は、ノードのリンクを変更するのみ。要素数によって処理時間が変わらないんためO(1)となる。

```
void Erase(Node<T>* node)
{
    //ノードを削除する。
    if (node == m_end) {
        //削除しようとしているノードが終端ノード、
        //削除するノードの前のノードが終端ノードになる。
```

```
            m_end = node->prev;
            if (m_end) {
                //終端ノードの次のノードにnullを代入。
                m_end->next = nullptr;
            }
            if (m_end == nullptr) {
                //リストが空になっている。
                m_begin = nullptr;
            }
        }
        else if (node == m_begin) {
            //削除しようとしているノードが先頭ノード。
            //先頭ノードを次のノードにする。
            m_begin = node->next;
            if (m_begin != nullptr) {
                //次のノードがnull出なければ。
                //先頭ノードの手前のノードは存在しないのでnullptrを代入
                m_begin->prev = nullptr;
            }
            if (m_begin == nullptr) {
                //リストが空になっている。
                m_end = nullptr;
            }
        }
        else {
            //削除しようとしているノードが中間ノード。
            node->prev->next = node->next;
        }
        //ノードを削除
        delete node;
    }
```

## 4.4.2 メモリの断片化

listはメモリの断片化が起きる。
メモリが断片化すると、見かけ上は十分なメモリが残っているのに、メモリのアロケートに失敗することがある。

## 4.4.3 どのようなケースでvectorを使うべき？？？

頻繁に要素の削除、追加がある場合はlistの使用を検討すべき。ただし、メモリの断片化には最新の注意が必要。特にPS5などの家庭用ゲーム機。
リストを利用する場合もメモリプールなどの仕組みを使って、断片化が起きないような工夫をすることがホトンである。

## 4.4.4 listはvectorと違い、要素を追加してもイテレーターが無効になることはない。

vectorは要素を追加した際に、新しい領域を確保して、そこに古いデータをコピーして、古い領域を削除するため、イテレーターが無効になることがある。 このような動作は、listでは起きないため、要素を追加してもイテレーターが無効になることはない。

### 4.4.5 要素を削除すると、イテレーターが無効になる。

これはvectorと同様に起きる。

## 評価テスト

次の評価テストを行いなさい。
評価テストへジャンプ