动态数组

概述

动态数组可以实现容量的自动扩大和缩小，进而可以提高内存的利用

接口总览

```
#define ELEMENT int  //元素类型
#define DEFAULT_CAPACITY 10  //默认容量
#define ELEMENT_NOT_FOUND -1   //返回查找状态

template<class T>
class ArrayList
{
public:

//函数名：ArrayList
//形参：容量大小
//说明：默认设置容量10
//作用：初始化size，capacity，element指针
//from golden_cat
    ArrayList(int capaticy = DEFAULT_CAPACITY);


//函数名：ClearHeap
//作用：清除数组中的数据（堆区）
//自己new的数据空间
    void clearHeap() ;

//函数名：size
//返回值：int
```

```
25 //作用：返回数组的有效元素个数
26       int size();
27
28 //函数名：Is_Empty
29 //返回值：bool
30 //作用：判断数组是否为空
31 //isempty-1
32      bool Is_Empty();
33
34 //函数名：IndexOf
35 //返回值：int
36 //参数：数组元素数据
37 //作用：传入数据，得到数据第一次出现位置索引
38 //说明：传入null，返回第一个null数据的索引
39 //       未搜做成功：返回 ELEMENT_NOT_FOUND
40    int IndexOf(T Ele);
41
42 //函数名：get
43 //返回值：元素数据类型
44 //参数：下标
45 //作用：传入下标得到元素值
46    T get(int index) ;
47
48    /**
49     * 是否包含某个元素
50     * @param element
51     * @return
52     */
53    bool contains(T Ele);
54
55
56     /**
57     * 设置index位置的元素
58     * @param index
59     * @param element
60     * @return 原来的元素
61     */
62    T set(int index,T element) ;
63
```

```
64
65 //函数名：add
66 //作用：添加元素
67    void add();  //添加任意数量的任意元素到尾部
68    void add(T Ele);  //添加一个Ele大小的元素到数组尾部
69
70    void add(int index ,T Ele ,int num = 1); //在任意位置添加一个元素
71
72 //函数名：ensureCapacity
73 //作用：需要扩容时扩容，并使元素指针指向扩容后的空间
74     void ensureCapacity(int n);
75
76
77   //函数名：trim
78   //作用：需要缩小容量时缩容
79     void trim(); //修剪
80
81 //函数名：Dele_ArrayForIndex
82 //作用：根据下标删除元素，并返回被删除的元素值
83    T Dele_ArrayForIndex(int index);
84 //函数名：Dele_ArrayForElement
85 //作用：删除第一次出现这个元素值的位置的元素
86 //未查找成功则报错
87    void Dele_ArrayForElement(int Ele);
88
89 //删除new出来的堆空间数据
90    T Dele_Array_Heap(int index);
91
92 public:
93    int m_size; //有效位个数
94    T * m_elements;  //元素指针，指向数组第一个元素地址
95    int m_capacity;  //容量
96
97 };
```

实现函数

```
1 template<class T>
2            ArrayList<T>::ArrayList(int capaticy = DEFAULT_CAPACITY) {
```

```
            m_capacity = (capaticy < DEFAULT_CAPACITY) ? DEFAULT_CAPACITY : capaticy;
            m_elements = new T[m_capacity];
            m_size = 0;
    }
```

说明：

核心语句：

```
m_capacity = (capaticy < DEFAULT_CAPACITY) ? DEFAULT_CAPACITY : capaticy;
```

输入容量小于预设值或未输入容量，则按预设值大小

```
template<class T>
     void ArrayList<T>::clear() {
             for(int i = 0; i < m_size; i++)
             {
                     delete this->m_elements[i];
                     this->m_elements[i] = nullptr;
             }
       m_size = 0;
    }
```

```
template<class T>
      int ArrayList<T>::size() {
            return m_size;
    }
```

```
template<class T>
   bool ArrayList<T>::Is_Empty()
   {
            return m_size == 0;
            //return (m_size == 0) ? true : false;
    }
```

```
核心：return m_size == 0;
```

```
template<class T>
   int ArrayList<T>::IndexOf(T Ele)
   {
     //兼容元素为null的情况
           if(Ele == NULL)
           {
                   for(int i = 0; i < m_size; i++)
                   {
                           if(this->m_elements[i] == NULL)
                                   return i;
                   }
           }

           for(int i = 0; i < m_size; i++)
           {
                   if(Ele == m_elements[i])
                           return i;
           }
           return ELEMENT_NOT_FOUND;
   }
```

```
template<class T>
   T ArrayList<T>::get(int index) {
           if(index < 0 || index >= m_size)  //下标判断
                   throw "false,overflow!";
           else
                   return m_elements[index];
   }

```

```
template<class T>
```

```
    bool ArrayList<T>::contains(T Ele)
    {
            //if(IndexOf(Ele) != -1)
            //      return true;
            //else
            //      return false;
            /*return (IndexOf(Ele) != -1) ? true : false; */
            return IndexOf(Ele) != ELEMENT_NOT_FOUND;
    }

 核心：return IndexOf(Ele) != ELEMENT_NOT_FOUND;找到元素则返回true
```

```
template<class T>
    T ArrayList<T>::set(int index,T element) {
            if(index < 0 || index !< m_size)
                    throw "false,overflow!";  //边界判断
            T temp = this->m_elements[index];   //recreat =
            this->m_elements[index] = element;
            return temp ;
    }
```

```
template<class T>
    void ArrayList<T>::add()
    {
            int num;
            T Ele;
            cin >> num;   //recreat >>
            int NewSpace = m_size+num; //静态扩充容量

//容量够用，直接在结尾添加
            if(NewSpace <= m_capacity)
            {
                    for(int i = m_size ; i< NewSpace ; i++)
                    {
                    cin >> Ele;
                    this->m_elements[i] = Ele;
```

```
                m_size ++;   //@important
                }
        }
  //容量超过，重新分配空间
        else
        {
                T * NewArray = new T[NewSpace];
   //拷贝原来的元素数据
            for(int i = 0;i < m_size ;i++)
                {
                        NewArray[i] = this->m_elements[i];
                }
   //在原来数据的后边加上需要添加的数据
                for(int i = m_size ; i < NewSpace ; i++)
                {
                        cin >> Ele;
                        NewArray[i] = Ele;
                }
   //释放原来的空间，更新element指针
                m_size = NewSpace;//@important
                m_capacity = NewSpace;//@important
                delete[] this->m_elements;
                this->m_elements = NewArray;
                cout << "NewSpace has been created!" <<endl;
        }

   }

```

不需要：

```
   //当不需要扩容：o（1）
   //需要扩容：o（n）
   //均摊复杂度：o（1）一般与最好情况相同 （11111111119复杂度突变时）
template<class T>
   void ArrayList<T>::add(T Ele)
   {
        add(m_size,Ele);   //give an initial number
```

```
    }
```

分析复杂度：最小index=size O（1），最大O（n），平均1+2+3+...+n/n=O(n)

```
template<class T>
   void ArrayList<T>::add(T Ele = 0, int num = 1, int index = m_size - 1)
   {
           if(index < 0 || index > m_size)
                   throw "false,overflow!";

           ensureCapacity(m_size+num);  //expand 1.5动态扩容
//
//扩容后直接在结尾添加元素即可

           for(int i = m_size - 1; i >= index ;i--)
           {
                   this->m_elements[i+num] = this->m_elements[i];
           }
           //每次循环都计算index+1，可以改进
           //@important .. 减少计算次数，优化
  int choice = 1;
  int ele = Ele;
  cout << "请输入选择插入的模式：" << endl
       << " 1. 插入任意位数的任意数据(默认)" << endl
       << " 2. 插入任意位数的相同数据(调用有第一个参数)" << endl;
       switch(choice)
       {
           case 1:
              for(int i = index; i < index + num; i++)
              {
                  cin >> ele;
                  this->m_elements[i] = ele;
                  m_size++;
              }
              break;
           case 2:
               cout << "输入连续添加的数值，默认是调用函数的第一个参数哦" << endl;
              cin >> ele;
```

```
            for(int i = index; i < index + num; i++)
            {
                this->m_elements[i] = ele;
                m_size++;
            }
            break;
     default:
       break;
       }
   }
```

```
template<class T>
    void ArrayList<T>::ensureCapacity(int n){
      //不需要扩容
            if(n <= m_capacity)
                    return;
    //扩容
           // int new_capacity = m_capacity + (m_capacity >> 1);
   int new_capacity = n*(1+0.5);

            T * NewArray = new T[new_capacity];
            for(int i = 0;i < m_size;i++)
            {
                    NewArray[i] = this->m_elements[i];
            }   //o(n)
   //释放旧空间，重新指向
            delete[] this->m_elements;
        cout <<"old_capacity:"<< m_capacity <<"--->new_capacity:"<< new_capacity << endl
            m_capacity = new_capacity;
            this->m_elements = NewArray;
    }
```

```
template<class T>
void ArrayList<T>::trim()
{
```

```
    //每次缩容一半
   int new_capacity = m_capacity >> 1;
   // int new_capacity = m_capacity*0.6;
 //有效元素位数大于容量的一半，或者容量的一半小于预设值，则不进行缩容
   if(m_size >= new_capacity || new_capacity < DEFAULT_CAPACITY)  return;

            T * new_array = new T[new_capacity];//此时size小于newcapacity

           for(int i = 0; i < m_size; i++)
           {
                   new_array[i] = this->m_elements[i];
           }

       cout <<"old_capacity:"<< m_capacity <<"--->new_capacity:"<< new_capacity << endl;
           m_capacity = new_capacity;
           delete[] this->m_elements;
           this->m_elements = new_array;

}
```

```
template<class T>
   T ArrayList<T>::Dele_ArrayForIndex(int index)
   {
     if(index < 0 || index >= m_size -1 ) {
         cout << "删除失败！" << endl;
         return;
         }

           T temp;
           temp = this->m_elements[index];

           for(int i = index+1;i < m_size; i++)
           {
                   this->m_elements[i-1] = this->m_elements[i];
           }
           //@important 减少了减法次数

           m_size--; //@important
```

```
        trim();
        return temp;
}
```

```
template<class T>
void ArrayList<T>::Dele_ArrayForElement(int Ele)
{
        this->Dele_ArrayForIndex(this->IndexOf(Ele));
}
```

```
        ostream& operator<<(ostream& cout,ArrayList<ELEMENT>& array)
{
        cout << "[ ";
        for(int i = 0;i< array.m_size ;i++)
        {
                if(i != 0) cout << ",";
                cout << array.m_elements[i];
        }
        cout << " ]" <<endl;
        return cout ;
}
```

复杂度分析

add（T Ele）最好O（1）最坏O（n），均摊O（1）

add（int index，T Ele）最好O（1），最坏O（n），平均O（n）

Dele_ArrayForIndex(int index)　最小index=size O（1），最大O（n），平均

1+2+3+...+n/n=O(n)

补充分析

1.构造函数开辟的是堆空间，清除时需要释放，删除时可以不释放（如果元素是指针也可以释放）

2.trim函数调用应该在删除操作后（size--后）调用

3.优化循环加减法

4.缩小容量倍数 * 扩大容量倍数 ！ = 1即可，否则复杂度震荡，例如初始容量2，初始有两个元素，此时添加一个元素扩容两倍容量为4，删除此元素后满足一半缩容条件，缩容后容量又变为2，这就 导致对一个元素重复添加删除时始终需要扩容和缩容，时间复杂度都是O（n）
５． //数组插入寻找位置复杂度1，插入操作复杂度n
//链表插入寻找位置复杂度n，插入操作复杂度1
//综合来看，动态数组和链表插入和删除数据时间复杂度都是n。且动态数组的set和get时间复杂度为1（不用查找），而链表的为n（需要查找）
/但是，链表对空间的利用率更高

链表

单链表

我们更多还是使用双向链表，但是哈希表其实是单链表实现的。
链表可以节省空间，没有空间浪费
我们学习的是一种离散形式的储存方式，以及如何用指针动态管理这个储存单元

节点实现

```
struct Node
 {
   int element;
   //shared_ptr<Node> next;
   Node* next;
   Node(int ele,Node* n):element(ele),next(n){}
 };
```

说明：单链表的节点中需要声明元素指（数据），指向节点的指针，以及构造函数。
在c++中，我们都是使用指针代表一个节点，初始时元素和指针都是空的（未定义的），构造函数帮助我们初始化，否则编译器会报错（一般由于指针未初始化）。

接口展示

```
class LinkedList
{
public:
```

```
//初始化头节点
    LinkedList();
        /**
        * 清除所有元素
        */
     void clear();
     //根据下标返回对应节点
    Node* node(int index);
     //根据元素找到第一次出现的位置
    int IndexOf(int Ele);
    /**
     * 获取index位置的元素
     * @param index
     * @return
     */
    int get(int index);
    /**
     * 是否包含某个元素
     * @param element
     * @return
     */
    bool contains(int Ele);
     /**
     * 设置index位置的元素
     * @param index
     * @param element
     * @return 原来的元素
     */
    int set(int index,int element);
  //结尾添加
    void add(int Ele);
  //任意位置添加
    void add(int index ,int Ele);
  //删除任意位置
    int Dele_ForIndex(int index);

public:
    int m_size;  //容量
    Node* first;  //头指针
};
```

接口函数实现

```
1
2    LinkedList()
3    {
4            m_size = 0;
5            first = new Node(m_size,nullptr);
6    }
7  /*使用头节点，进而使后续添加删除操作统一化，否则还要分情况实现*/
8
9       /**
10       * 清除除了头节点的所有元素
11       */
12     void clear()
13      {
14              m_size = 0;
15              Node* node = first->next;
16
17              while(node != nullptr)
18              {
19                    Node* dele = node;
20                    delete dele;
21                    node = node->next;
22              }
23      }
24
25    //清除所有元素
26            void clear_all()
27      {
28              m_size = 0;
29              Node* node = first;
30
31              while(node != nullptr)
32              {
33                    Node* dele = node;
34                    node = node->next;
35                    delete dele;
```

```
36              }
37      }
38
39    //非头插法时用这个
40    Node* LinkedList::node(int index)
41    {
42      if(index < 0 || index > m_size-1)
43            {
44                cerr << "下标溢出！" << endl;
45                return;
46            }
47
48            Node* node = first->next;
49            for(int i = 0; i < index; i++)
50            {
51                    node = node->next;
52            }
53            return (index != -1)? node : first;
54    }
55
56 //头插元素多时优先用这个
57            Node* LinkedList::node(int index)
58    {
59            if(index == -1)  return first;
60            Node* node = first->next;
61            for(int i = 0; i < index; i++)
62            {
63                    node = node->next;
64            }
65            return node;
66    }
67  //头节点体现优异性，Node可以返回0号索引的节点
68  //很重要：后面添加删除都需要用到，根据下标找元素也用到。
69  //从first-〉next开始，往后移动index次，就可以找到我们需要的节点
70
71
72    int IndexOf(int Ele)
73    {
74            Node* node = first->next;
75            for(int i = 0; i< m_size; i++){
```

```
76                      if(node->element == Ele)
77                              return i;
78                      node = node->next;
79              }
80               return NOT_CONTAINS;
81      }
82  //判断元素是否存在时用到
83
84
85      /**
86       * 获取index位置的元素
87       * @param index
88       * @return
89       */
90      int get(int index)
91      {
92              //Node* node = first->next;
93              //for(int i = 0;i < index; i++)
94              //{
95              //      node = node->next;
96              //}
97              /*return (node->element);*/
98              return node(index)->element;
99      }
100
101
102     /**
103      * 是否包含某个元素
104      * @param element
105      * @return
106      */
107     bool contains(int Ele)
108     {
109             //Node* node = first->next;
110             //while(node != nullptr)
111             //{
112             //      if(node->element == Ele)
113             //              return true;
114   //          node = node->next;
115             //}
```

```
116             //return false;
117             return IndexOf(Ele) != NOT_CONTAINS;
118     }
119
120
121      /**
122      * 设置index位置的元素
123      * @param index
124      * @param element
125      * @return 原来的元素
126      */
127     int set(int index,int element)
128     {
129             Node* old = node(index);
130             node(index)->element = element;
131             return old->element;
132     }
133
134     //O（n）
135     void add(int Ele, index = m_size - 1)
136     {
137             //if(index == 0)  //@important
138             //{
139             //      first->next =new Node(Ele,first->next);
140             //}
141
142             //else
143             //{
144             //Node* p = node(index-1);  //不能直接用node（）赋值
145             //p->next = new Node(Ele,p->next);
146             //}
147
148
149             Node* p = node(index-1);  //不能直接用node（）赋值,因为有头节点，所以统一操作
150             p->next = new Node(Ele,p->next);
151             m_size++;
152     }
153
154     //o(n)
```

```
155    int Dele_ForIndex(int index)
156    {
157            //if(index == 0)
158            //{
159            //      Node* node = first;
160            //      first->next = first->next->next;
161            //      m_size--;
162            //    return node->element;
163            //}
164            //else
165            //{
166            //Node* prv = node(index-1);
167            //Node* node = prv->next;
168            //prv->next = prv->next->next;
169            //m_size--;
170     //   return node->element;
171            //}
172
173            Node* prv = node(index-1);
174            Node* node = prv->next;  //提前保留数值
175            prv->next = prv->next->next;
176            m_size--;
177   int val = node->element;
178   delete node;
179        return val;
180    }
```

判断链表是否有环

```
1   struct ListNode {
2       int val;
3       ListNode *next;
```

```
    ListNode(int x) : val(x), next(NULL) {}
};

class Solution {
public:
    bool hasCycle(ListNode *head) {
        if(head == NULL|| head->next == NULL) return false;

        ListNode * slow = head;
        ListNode * fast = head->next;
        //两种可能，fast=slow时跳出循环，有环
        //fast为空或者fast指向节点是最后一个节点，代表无环且循环到链表结尾了
        while(fast != slow && fast && fast->next)
        {
            fast = fast->next->next;
            slow = slow->next;
        }
        return fast == slow;
    }
};
```

反转链表

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
            //ListNode* NewHead = reverseList(head->next);
            //head->next->next = head;
            //head->next = nullptr;
            //return NewHead;


        ListNode* NewHead = NULL;
        ListNode* temp = NULL;
           while(head)
           {
                   temp = head->next;
                   head->next = NewHead;
```

```
                    NewHead = head;
                    head = temp;
                }
            return NewHead;


    }
};
```

删除链表节点

```
class Solution {
public:
    void deleteNode(ListNode* node) {
        node->val=node->next->val;
        Node* temp = node->next;
        node->next=node->next->next;
        delete temp;
    }
};
```

双向链表

双向链表查找元素比单链表快几乎一倍，因为其根据传入的索引值选择从前往后找还是从后往前找，相当于二分法了。

双向链表比单链表就多了一个prev指针。

```
#pragma once
#include<iostream>
using namespace std;


struct Node
 {
   int element;
   //shared_ptr<Node> next;
   Node* prev;
```

```
    Node* next;
    Node(){prev = nullptr; next = nullptr;}
    Node(Node* p, int ele, Node* n) : prev(p), element(ele), next(n){}
 };

class BiLinkedList
{
public:

    BiLinkedList()
    {
         m_size = 0;
         first = NULL;  //只用头指针，不用头节点
         last = NULL;
    }

    Node* node(int index)
    {
            if(index <= (m_size >> 1))
            {
            Node* node = first;
            for(int i = 0; i < index; i++)
            {
                    node = node->next;
            }
                  return node;
            }
            else
            {
                    Node* node = last;
                  for(int i = m_size-1; i > index; i--)
                    {
                            node = node->prev;
                    }
                       return node;
            }
    }


```

```
50      void add(int Ele)
51      {
52              add(m_size,Ele);
53      }
54
55      void add(int index,int Ele)
56      {
57              if(index == m_size)
58              {
59                      Node* old_last = last;
60                      last = new Node(old_last,Ele,nullptr);
61
62                      if(old_last == nullptr)
63                              first = last;
64                      else
65                              old_last->next = last;
66
67
68              }else{
69
70              Node* next = node(index);
71              Node* prev = next->prev;
72
73              Node* node = new Node(prev,Ele,next);
74              next->prev = node;
75
76              if(index != 0){
77                      prev->next = node;
78                 }else{
79                       first = node; //在第一个节点插入
80                 }
81
82          }
83              m_size++;
84
85      }
86
87  int remove(int index)
88  {
```

```
    Node* dele_node = node(index);
    Node* prev = dele_node->prev;
    Node* next = dele_node->next;

    if(next != nullptr)  //index = m_size-1
            prev->next = next;
    else
            last = prev;  //last points the front node of this dele_node

    if(prev != nullptr)  //index = 0
            next->prev = prev;
    else
            first = next;  //first points the next note of dele_note

    m_size--;
    int val = dele_node->element;
    delete dele_node;//要删除的节点是new出来的
    return val;
}


public:
int m_size;
Node* first;
Node* last;
};

```

循环链表

```
#pragma once
```

```
#include<iostream>
using namespace std;


struct Node
 {
   int element;
   //shared_ptr<Node> next;
   Node* prev;
   Node* next;
   Node(Node* p, int ele, Node* n) : prev(p), element(ele), next(n){}
 };

class CircleLinkedList
{
public:

   CircleLinkedList()
   {
           m_size = 0;
        first = last = NULL; //注意：要手动初始化
   }

   Node* node(int index)
   {
           if(index <= (m_size >> 1))
           {
           Node* node = first;
           for(int i = 0; i < index; i++)
           {
                   node = node->next;
           }
                 return node;
           }
           else
           {
                   Node* node = last;
   //int i = 0; i < m_size - index; i++，从最后一个元素开始往前指size-index-1次
                   for(int i = m_size-1; i > index; i--)
```

```
41                      {
42                              node = node->prev;
43                      }
44                         return node;
45              }
46      }
47
48
49      void add(int Ele)
50      {
51              add(m_size,Ele);
52      }
53
54  //add核心操作：找到添加位置的前一个和后一个节点，利用node函数直接返回
55  //如果在最后添加节点，
56  //初始没有节点时插入第一个节点，也要特殊处理
57  //在头部插入时也要手动修正first的指向
58                      void add(int节点 index,int Ele)
59      {
60              if(index == m_size)
61              {
62                      Node* old_last = last;
63                      last = new Node(old_last,Ele,first);
64                      if(old_last == NULL) //index = 0
65                      {
66                              first = last;
67                              old_last = last;
68                      }
69                      else{
70                      old_last->next = last;
71                      first->prev = last;
72                      }
73
74              }else{
75
76              Node* next = node(index);
77              Node* prev = next->prev;
78
79              Node* node = new Node(prev,index,next);
```

```
80              next->prev = node;
81              prev->next = node;
82              if(next == first)
83                 first = node; //在第一个节点插入
84          }
85              m_size++;
86
87      }
88
89  int remove_index(int index)
90  {
91      if(index < 0 || index >= m_size)
92              throw "delete wrong!";
93      else
94              return remove_node(node(index));
95  }
96
97
98  int remove_element(int Ele)
99  {
100     return remove_index(indexOf(Ele));
101 }
102
103
104 int remove_node(Node* dele_node)
105 {
106     if(m_size == 1)
107     {
108             first = NULL;
109             last = NULL;
110     }
111     Node* prev = dele_node->prev;
112     Node* next = dele_node->next;
113     prev->next = next;
114     next->prev = prev;
115
116
117     if(dele_node == last)
118             last = prev;  //last points the front node of this dele_node
```

```
119
120     if(dele_node == first)
121             first = next;  //first points the next note of dele_note
122
123     m_size--;
124     int val = dele_node->element;
125     delete dele_node;
126     return val;
127 }
128
129 #define ELEMENT_NOT_FOUND -1
130     int indexOf(int element) {
131             if (element == NULL) {
132                     Node* node = first;
133                     for (int i = 0; i < m_size; i++) {
134                             if (node->element == NULL) return i;
135                             node = node->next;
136                     }
137             } else {
138                     Node* node = first;
139                     for(int i = 0; i < m_size; i++)
140                     {
141                             if(node->element == element)
142                                     return i;
143                             node = node->next;
144                     }
145             }
146             return ELEMENT_NOT_FOUND;
147     }
148
149     void reset() {
150             current = first;
151     }
152
153     int next() {
154             if (current == NULL) return NULL;
155
156             current = current->next;
157             return current->element;
158     }
```

```
159
160     int remove_current() {
161             if (current == NULL) return NULL;
162
163             Node* next_node = current->next;
164             int element = remove_node(current);
165             if (m_size == 0) {
166                     current = NULL;
167             } else {
168                     current = next_node;
169             }
170             return element;
171     }
172 public:
173 int m_size;
174 Node* first;
175 Node* last;
176 Node* current;
177 };
```

修改格式并加入注释

```
1  #pragma once
2  #include<iostream>
3  using namespace std;
4
5
6  struct Node
7   {
8     int element; // 节点内的数据
9     Node* prev; // 指向此节点上一个节点的指针
10    Node* next; // 指向下一个节点的指针
11
12    /*构造函数Node
13     *形参：prev指针，元素数值，next指针
14     *作用：创建node节点时用于初始化
15     */
16    Node(Node* p = NULL, int ele = 0, Node* n = NULL) : prev(p), element(ele), next(n){}
17  };
```

```
18
```

```
19
20
21  class CircleLinkedList
22  {
23  public:
24
25      /*循环链表的构造函数：
26       *作用：初始化链表类内的指针和大小
27       */
28
29      CircleLinkedList() {
30              m_size = 0;
31           first = last = NULL;
32      }
33
34
35
36      /*作用：根据索引找到指向此节点的指针
37       *形参：int 类型索引
38       *返回值：Node* 节点指针
39       */
40
41      Node* node(int index)
42      {
43              //从前向后查找
44              if (index <= (m_size >> 1)) {
45
46                Node* node = first;
47
48                for (int i = 0; i < index; i++) {
49                       node = node->next;
50                }
51
52                return node;
53
54              }
55              //从后往前查找
56              else {
57
```

```
58                  Node* node = last;
59
60                  for (int i = 0; i < m_size - index; i++) {
61                          node = node->prev;
62                   }
63              return node;
64
65          }//end else
66
67  }//end if
68
69
70  void add(int Ele,int index = m_size)
71  {
72          //向结尾添加数据时
73          if (index == m_size)
74          {
75
76                  Node* old_last = last; // last原来指向的节点
77                  last = new Node(old_last, Ele, first); // last指向新的节点，更新last指向
78
79                  //添加的是第一个节点时
80                  //使得新节点的prev和next都指向自己
81                  if (old_last == NULL) {
82                      first    = last;
83                     // old_last = last;
84
85                  }//end inner if
86                  else {
87
88                      old_last->next = last;
89                      first->prev    = last;
90                  }//end inner else
91          }//end if
92
93          else
94          {
95
96              Node* next = node(index);
```

```
97              Node* prev = next->prev;
98              Node* node = new Node(prev,Ele, next); // 新节点
99
100             next->prev = node;
101             prev->next = node;
102
103             //在第一个节点插入
104             //需要更新last的指向
105             if (next == first)
106             {
107                     first = node;
108                 last->next = first;
109               }
110
111         }
112
113           m_size++;
114
115   }//end function add
116
117
118
119 int remove_index(int index)
120 {
121
122   if (index < 0 || index >= m_size)
123           throw "wrong!overrange";
124
125   else
126           return remove_node(node(index));
127
128 }
129
130
131
132
133 int remove_node(Node* dele_node)
134 {
135   //当只有一个节点时
```

```
136     if (m_size == 1)
137     {
138             first = NULL;
139             last = NULL;
140     }
141
142     //有多个节点时
143     Node* prev = dele_node->prev;
144     Node* next = dele_node->next;
145     prev->next = next;
146     next->prev = prev;
147
148     //删除最后一个：更新last指向,此时prev->next就是first
149     if (dele_node == last)
150             last = prev;  //last points the front node of this dele_node
151
152     // 删除第一个：更新first指向，next->prev就是last
153     if (dele_node == first)
154             first = next;  //first points the next note of dele_note
155
156     m_size--;
157
158     //释放节点前保存返回的节点值
159     int val = dele_node->element;
160
161     delete dele_node;
162     return val;
163 }
164
165
166
167
168 int remove_element(int Ele)
169 {
170     return remove_index(indexOf(Ele));
171 }
172
173
174
```

```
175
176 #define ELEMENT_NOT_FOUND -1
177
178 int indexOf(int element)
179 {
180    //兼容元素为NULL时
181             if (element == NULL)
182             {
183
184                     Node* node = first;
185
186                     for (int i = 0; i < m_size;  i++)
187                     {
188                             if (node->element == NULL) return i;
189
190                             node = node->next;
191                     } // end for
192             } // end if
193
194             //一般情况
195             else {
196
197                     Node* node = first;
198
199                     for (int i = 0; i < m_size; i++)
200                     {
201                             if (node->element == element)
202                                     return i;
203
204                             node = node->next;
205                     } // end for
206             } // end else
207
208             return ELEMENT_NOT_FOUND;
209    } // end function indexOf
210
211
212
213
```

```
214     void reset() {
215             current = first;
216     }
217
218
219
220     //使得current指针向下指，并返回其指向的值
221     int next() {
222             if (current == NULL) return NULL;
223
224             current = current->next;
225             return current->element;
226     }
227
228
229
230
231     int remove_current()
232     {
233
234             if (current == NULL) return NULL;
235
236             //提前保存
237             Node* next_node = current->next;
238             int element = remove_node(current);
239
240             if (m_size == 0) {
241                     current = NULL;
242             }
243
244             else {
245                     current = next_node;
246             }
247
248             return element;
249     }
250
251
252
```

```
public:

int m_size; // 链表大小
Node* first; // 指向链表第一个节点的指针
Node* last; // 指向最后一个节点的指针
Node* current; // 指向当前节点的指针

};
```

解决约瑟夫环的问题

```
#include<iostream>
//#include"Bidriclinkedlist.h"
#include"CircleLinkedList.h"
using namespace std;


ostream& operator<<(ostream& cout,CircleLinkedList & list)
{
   Node* p = list.first;
   for(int i = 0;i<list.m_size;i++)
   {
           cout << p->prev->element <<'-';
           cout << p->element << '-';
           p = p->next;
           cout << p->element <<' ';
   }
   return cout;
}

int main()
{
  CircleLinkedList list;
  list.add(0,10);
  list.add(1,20);
  list.add(2,30);
  list.add(11);
  list.add(22);
  list.add(33);
```

```
29
30    list.remove_element(30);
31    cout << list <<endl;
32    list.reset();
33    while(list.m_size != 0){
34    list.next();
35    list.next();
36    cout << list.remove_current() << endl;
37    }
38
39
40    //cout << list << endl;
41    //list.remove(list.first);
42    //cout << list << endl;
43    //list.remove(list.last);
44    //cout << list << endl;
45    //list.remove(0);
46    //cout << list << endl;
47
48
49    getchar();
50    return 0;
51  }
```

修改-双链表无头结点的删除操作

```
1     Node* node(int index)
2     {
3             if(index <= (m_size >> 1))
4             {
5             Node* node = first;
6             for(int i = 0; i < index; i++)
7             {
8                     node = node->next;
9             }
```

```
                return node;
            }
            else
            {
                    Node* node = last;
                    //for(int i = m_size-1; i > index; i--)
                    for(int i = 0; i < m_size - index - 1; i++)
                    {
                            node = node->prev;
                    }
                       return node;
            }

    }

 int i = 0; i < m_size - index - 1; i++
```

```
int delete_list(int index)
{
    Node* dele_node = node(index);

    Node* prev = dele_node->prev;
    Node* next = dele_node->next;

//三总情况分别讨论
    if(index == m_size -1) {
            prev->next = next;
            last = prev;
    }else if(index == 0) {
            next->prev = prev;
            first = next;
    } else
    {
      prev->next = next;
      next->prev = prev;
    }

    m_size--;
```

```
    int val = dele_node->element;
    delete dele_node;
    return val;
}

void clear()
{
    Node* temp = first;
    while(temp != NULL)
    {
            Node *temp2 = temp->next; //*注意先保留next不然delete后找不到了
            delete temp;
            temp = temp2;
    }
    m_size = 0;
}

```

修改分析2-add是重载还是直接默认形参?

添加元素，如果m_size是非静态成员变量，其在类内并没有初始化，而默认参数必须要求参数初始化。此时可以使用函数重载实现多接口，如下：

```
    void add(int ele)
    {
            add(ele,m_size);
    }


     void add(int Ele,int index = m_size)
    {
            if(index == m_size)
            {
                    Node* old_last = last;
                    last = new Node(old_last,Ele,nullptr);

                    if(old_last == nullptr)
```

```
                        first = last;
                else
                        old_last->next = last;


        }else{

        Node* next = node(index);
        Node* prev = next->prev;

        Node* node = new Node(prev,index,next);
        next->prev = node;

        if(index != 0){
                prev->next = node;
           }else{
                 first = node; //在第一个节点插入
           }

    }
        m_size++;

}
```

如果把m_size改为静态变量，则可以直接赋值默认形参，但是注意需要在类外初始化，不要再构造函数内初始化。

```
static int m_size;


int Queue::m_size  = 0;
```

这里注意一下，static静态变量默认初始化为0。
还有就是，如果接口和实现分离，有默认参数的函数中，一般我们都把默认参数放在声明处而不是定义处。如果声明和定义都有默认参数，编译器将会报错。

链式队列

链式队列可以直接在双向链表的基础上改进出来。这里使用外部节点，因为node()函数返回值需要是Node*类型。

```
#pragma once
#include<iostream>


template<class ValType>
class Node
 {
 public:
   ValType element;
   Node* prev;
   Node* next;
   Node(Node* p = NULL, ValType ele = NULL, Node* n = NULL) : prev(p), eleme
 };

//如果用类外声明Node，注意
//last = new Node<T>(old_last,Ele,nullptr);
//Node<T>* p = NULL;
//好处：类外实现template<class T>
//Node<T>* Queue<T>::node(int index)

template<class T>
class Queue : private Node<T>  //继承的父类属性都变为private
{

public:

   template<class T>
   friend std::ostream& operator << (std::ostream& out ,Queue<T> que);

   Queue();

   /*
   -根据下标找到节点
   -两种匹配模式，下标小从前往后找，下标大从后往前找
   -返回下标
```

```
36      */
37      T front();
38      T back();
39      Node<T>* node(int index);
40      int size();
41       void enqueue(T Ele,int index = m_size);
42       T dequeue(int index = m_size - 1);
43       void clear();
44       void print(std::ostream& out);
45
46  private:
47  static int m_size;
48  Node<T>* first;
49  Node<T>* last;
50
51  };
52
53  template<class T>
54  int Queue<T>::m_size = 0;
55
56  // 静态成员初始化默认为0
57
58
59  template<class T>
60  std::ostream& operator << (std::ostream& out ,Queue<T> que)
61  {
62      que.print(out);
63      return out;
64  }
65
66
67  template<class T>
68  Queue<T>::Queue()
69      {
70          first = nullptr;
71          last = nullptr;
72      }
73
74
```

```
75
76  template<class T>
77  T Queue<T>::front()
78     {
79              return first->element;
80     }
81
82
83  template<class T>
84  T Queue<T>::back()
85     {
86              return last->element;
87     }
88
89
90  template<class T>
91  Node<T>* Queue<T>::node(int index)
92     {
93              if(index <= (m_size >> 1))
94              {
95              Node<T>* node = first;
96              for(int i = 0; i < index; i++)
97              {
98                      node = node->next;
99              }
100                   return node;
101             }
102             else
103             {
104                     Node<T>* node = last;
105                     //for(int i = m_size-1; i > index; i--)
106                     for(int i = 0; i < m_size - index - 1; i++)
107                     {
108                             node = node->prev;
109                     }
110                        return node;
111             }
112
113     }
```

```
114
115
116 template<class T>
117 int Queue<T>::size()
118    {
119            return this->m_size;
120    }
121
122
123 template<class T>
124 void Queue<T>::enqueue(T Ele,int index = m_size)
125    {
126            if(index == m_size)  //末尾添加
127            {
128                    Node<T>* old_last = last;
129                    last = new Node<T>(old_last,Ele,nullptr);
130
131                    if(old_last == nullptr)
132                            first = last; // 添加第一个
133                    else
134                            old_last->next = last;
135
136
137            }else{
138
139            Node<T>* next = node(index);
140            Node<T>* prev = next->prev;
141
142            Node<T>* node = new Node<T>(prev,index,next);
143            next->prev = node;
144
145
146            if(index != 0){
147                    prev->next = node;
148               }else{
149                     first = node; //在第一个节点插入
150               }
151
152        }
```

```
153             m_size++;
154 
155     }
156 
157 
158 template<class T>
159 T Queue<T>::dequeue(int index = m_size - 1)
160 {
161     if(m_size == 0) throw "没有节点可以删除了！";
162 
163     Node<T>* dele_node = node(index);
164     Node<T>* prev = dele_node->prev;
165     Node<T>* next = dele_node->next;
166 
167 
168     if(m_size == 1)       //next == NULL; prev == NULL;
169     {
170     int val = dele_node->element;
171     delete dele_node;
172     first = last = NULL;  //**注意！！！！注意清空指针，以便于下次添加
173     m_size--;
174     return val;
175     }
176 
177 
178     if(index == m_size -1) {
179             prev->next = next; // prev != NULL; next == NULL;
180             last = prev;
181     }else if(index == 0) {
182             next->prev = prev; // next !=NULL; prev == NULL;
183             first = next;
184     } else                  // next != NULL; prev != NULL;
185     {
186       prev->next = next;
187       next->prev = prev;
188     }
189 
190     m_size--;
191     int val = dele_node->element;
```

```
192     delete dele_node;
193     return val;
194 }
195 
196 template<class T>
197 void Queue<T>::clear()
198 {
199     Node<T>* temp = first;
200     while(temp != NULL)
201     {
202             Node<T> *temp2 = temp->next;
203             delete temp;
204             temp = temp2;
205     }
206     m_size = 0;
207 }
208 
209 
210 template<class T>
211 void Queue<T>::print(std::ostream& out)
212 {
213     Node<T>* node = first;
214     while(node != NULL) {
215             out << node->element << ' ' ;
216             node = node->next;
217     }
218 }
219 
220 
```

接口实现

T front();

T back();

Node<T>* node(int index);

int size();

void enqueue(T Ele,int index = m_size);

T dequeue(int index = m_size - 1);

void clear();
void print(std::ostream& out);

Node<T>* node(int index);

```
template<class T>
Node<T>* Queue<T>::node(int index)
   {
//下标判断
     if(index < 0 || index > m_size - 1) throw "Index outof Range!";

          //从前往后找，找index次
           if(index <= (m_size >> 1))
           {
           Node<T>* node = first;
           for(int i = 0; i < index; i++)
           {
                   node = node->next;
           }
                 return node;
           }
           else  //从后往前找，找n - 1 - index 次
           {
                   Node<T>* node = last;
                   for(int i = 0; i < m_size - index - 1; i++)
                   {
                           node = node->prev;
                   }
                      return node;
           }

   }
```

1.从前往后找的次数+从后往前找的次数+1 = size
2.注意是指针移动的次数
3.index = 0 return first
index = m_size - 1 return last

void enqueue(T Ele,int index = m_size);

```
template<class T>
void Queue<T>::enqueue(T Ele,int index = m_size)
   {
           if(index == m_size)  //末尾添加
           {
                   Node<T>* old_last = last;
                   last = new Node<T>(old_last,Ele,nullptr);

                   if(old_last == nullptr)
                           first = last; // 添加第一个
                   else
                           old_last->next = last;


           }else{

           Node<T>* next = node(index); //next一定不是NULL
           Node<T>* prev = next->prev;

           Node<T>* node = new Node<T>(prev,Ele,next);
           next->prev = node;


           if(prev != NULL){
                   prev->next = node;
              }else{
                    first = node; //在第一个节点插入
              }

       }
           m_size++;

   }
```

分情况：因为有first和last两个指针，所以在头部和尾部插入节点时需要更新指针指向；如果从无到有插入第一个节点，需要让first和last都指向一个节点。
正常情况下，只需要得到需要插入位置的节点作为next，找到其前面哪个节点作为prev，创建一个新节点使其前后分别指向prev和next，之后只要让next->prev和prev->next都指向这个新创建的节点即可。
可next->prev和prev->next的前提是prev和next都不为NULL，也就是说，如果在最前插入，prev是NULL，此时再prev->next会出错，所以不能这样，最前面插入我们只需next->prev指向新节点，然后更新first指向即可。
前面说prev是NULL，此时再prev->next会出错，那next->prev会不会出错呢？答案是会的。一般我们插入都有一步是“获得index位置的节点”，而此时的index是从0取值到 m_size - 1 的，也就是说，我们使用这种方法插入节点时永远不能插入在最后方，也就是 m_size 这个位置，此时就需要单独拿出来这种情况进行分析了。if(index == m_size) ..... 这一部分就是对结尾进行分析。结尾插入很简单，只需要创建一个指向last的新节点，然后让原来last的那个节点指向这个新节点即可，最后更新last指向。我用下面两句代码就实现了。

```
Node<T>* old_last = last;   //保留旧last节点
last = new Node<T>(old_last,Ele,nullptr); //创造，更新last
old_last->next = last; //建立链接
```

但是添加最后一个节点时也要考虑一个特殊情况，那就是从无到有插入第一个节点，此时ols_last ,last ,first都是NULL，old_last->next自然就是错误的了。所以此时我们不用这样，我们只需first = last = 新建立的节点 即可。
总结：
1.一般而言：用node函数获取节点，添加--此时考虑prev为NULL即可
2.特殊1：最后添加节点--此时考虑添加第一个节点（index = size = 0）
3.分情况的本质：
a.指针为NULL
b.函数自身形参范围限制
c.指针的刷新方式不同

T dequeue(int index = m_size - 1);

```cpp
template<class T>
T Queue<T>::dequeue(int index = m_size - 1)
{
    if(m_size == 0) throw "没有节点可以删除了！";

    Node<T>* dele_node = node(index);
    Node<T>* prev = dele_node->prev;
    Node<T>* next = dele_node->next;


    if(m_size == 1)       //next == NULL; prev == NULL;
    {
    int val = dele_node->element;
    delete dele_node;
    first = last = NULL;  //**注意！！！！注意清空指针，以便于下次添加
    m_size--;
    return val;
    }


    if(index == m_size -1) {
            prev->next = next; // prev != NULL; next == NULL;
            last = prev;
    }else if(index == 0) {
            next->prev = prev; // next !=NULL; prev == NULL;
            first = next;
    } else                  // next != NULL; prev != NULL;
    {
      prev->next = next;
      next->prev = prev;
    }

    m_size--;
    int val = dele_node->element;
    delete dele_node;
    return val;
}
```

理解和添加大相径庭，根据指针的NULL情况分情况讨论：

```
Node<T>* dele_node = node(index);
Node<T>* prev = dele_node->prev;
Node<T>* next = dele_node->next;
```

1.next == NULL; prev == NULL;
此时是删除最后一个节点，注意把first和last都清空为NULL即可

2.prev != NULL; next == NULL;
删除最后一个点，注意更新last指向

3.next !=NULL; prev == NULL;
删除第一个节点，更新first指向

4.next != NULL; prev != NULL;
通常情况，正常链接即可

注意事项

节点类作为基类私有继承给Queue

```
template<class T>
class Queue : private Node<T> //继承的父类属性都变为private
```

重载<< 友元函数声明前加模板标识

```
template<class T>
friend std::ostream& operator << (std::ostream& out ,Queue<T> que);
```

实现也要加模板标识

```
template<class T>
std::ostream& operator << (std::ostream& out ,Queue<T> que)
```

节点类最好有一个默认构造函数

```
Node(Node* p = NULL, ValType ele = NULL, Node* n = NULL) : prev(p), element(ele), next(n)
{}
```

如果用类外声明Node，注意

//last = new Node<T>(old_last,Ele,nullptr);
//Node<T>* p = NULL;
//继承时也要跟<T>

//好处：类外实现template<class T>
//Node<T>* Queue<T>::node(int index)

默认形参时类成员对象

enqueue(T Ele,int index = m_size);一般在声明的时候用，实现不用

static int m_size; 类内

template<class T> 类外
int Queue<T>::m_size = 0;
// 静态成员初始化默认为0

注意删除最后一个，添加第一个，不能删除这三种特殊情况

循环队列（数组实现）

循环队列的核心在于索引的映射。

TrueIndex(int index)接受虚拟索引并返回真实索引。front为0，这个时候一切的操作都和动态数组相同，比如在末尾添加元素add(Ele,m_size)，删除某个位置的元素delete(index)。但是在循环队列中，front不一定是0，此时的index也不是相对于0而言的了，而是相对front而言的。size 应该 变为front+ size ，但是注意“循环”二字的意思，循环就是取模的意思。所以实际上真正的索引不是相对于0的size，也不是仅仅相对于front的size+front，而是（size+front）% length。

TrueIndex(相对于front的位置) = 实际位置

```
template<class T>
int CircleQueue<T>::TrueIndex(int index)
{
        index += m_front;
        return (index < m_capacity) ? index : index - m_capacity;
        //模运算优化，2mcapacity > index + front的
```

```
    }
```

注意循环队列的模运算是二维以下的，可以化简为加减运算。

先看下面的添加

```
template<class T>
   void CircleQueue<T>::enqueue(T Ele)
   {
           this->DanaCapacity(m_size+1);
           this->m_element[TrueIndex(m_size)] = Ele;
           m_size++;
   }

```

动态数组中我们是 this->m_element[(m_size)] = Ele;
但是因为是循环队列，所以是this->m_element[TrueIndex(m_size)] = Ele;

再看头部出队

```
   //头部出队
   template<class T>
   T CircleQueue<T>::dequeue()
   {
      T temp = m_element[m_front];
      this->m_element[m_front] = NULL;
      m_front = TrueIndex(1);
      m_size--;
      return temp;
   }
```

动态数组是 this->m_element[0] = NULL;
循环队列是 this->m_element[m_front] = NULL; 本质就是TrueIndex(0)，也就是front本身。

动态数组： m_front ++;
循环队列： m_front = TrueIndex(1);

最后看动态扩容：

```
//动态扩容
template<class T>
void CircleQueue<T>::DanaCapacity(int capa)
{
        int oldCapa = this->m_capacity;
        if(capa <= oldCapa) return;

        int NewCapacity = m_capacity + (m_capacity >> 1);
        T* NewEle = new T[NewCapacity]();

        for(int i = 0; i < m_size ; i++)
        {
                NewEle[i] = this->m_element[TrueIndex(i)];
        }

        delete[] this->m_element;
        this->m_element = NewEle;

        //注意更新容量和头指针
        m_capacity = NewCapacity;
        m_front = 0;

}
```

动态数组： NewEle[i] = this->m_element[i];
循环队列： NewEle[i] = this->m_element[TrueIndex(i)]; 从front往后开始赋值

注意：扩容后更新front

```
#pragma once
#define DEFAULT_CAPACITY 10
```

```
3
4
5
6  template<class T>
7  class CircleQueue
8  {
9  public:
10
11     template<class T>
12     friend std::ostream& operator<<(std::ostream& out ,CircleQueue<T> &que);
13
14
15     //初始化
16     CircleQueue();
17
18     //尾部入队
19     void enqueue(T Ele);
20     //索引映射
21     int TrueIndex(int index);
22     //头部出队
23     T dequeue();
24     //获取头元素
25     T front();
26
27     //动态扩容
28     void DanaCapacity(int capa);
29
30     //size
31     int size()
32     {
33             return m_size;
34     }
35
36     //empty
37     bool IsEmpty()
38     {
39             return m_size == 0;
40     }
41
```

```
42
43  private:
44      int m_front;
45      int m_size;
46      int* m_element;
47      int m_capacity;
48  };
49
50
51
52      //初始化
53  template<class T>
54  CircleQueue<T>::CircleQueue()
55  {
56      m_size = 0;
57      m_front = 0;
58      m_capacity = DEFAULT_CAPACITY;
59      m_element = new T[m_capacity]();  //注意不要再声明新变量了!
60  }
61
62      //尾部入队
63  template<class T>
64      void CircleQueue<T>::enqueue(T Ele)
65      {
66              this->DanaCapacity(m_size+1);
67              this->m_element[TrueIndex(m_size)] = Ele;
68              m_size++;
69      }
70
71
72
73      //索引映射
74      template<class T>
75      int CircleQueue<T>::TrueIndex(int index)
76      {
77              index += m_front;
78              return (index < m_capacity) ? index : index - m_capacity;
79              //模运算优化，2mcapacity > index + front的
80      }
```

```
81
82  //头部出队
83  template<class T>
84  T CircleQueue<T>::dequeue()
85  {
86     T temp = m_element[m_front];
87     this->m_element[m_front] = NULL;
88     m_front = TrueIndex(1);
89     m_size--;
90     return temp;
91  }
92
93  //获取头元素
94  template<class T>
95  T CircleQueue<T>::front()
96  {
97          return this->m_element[m_front];
98  }
99
100 //动态扩容
101 template<class T>
102 void CircleQueue<T>::DanaCapacity(int capa)
103 {
104         int oldCapa = this->m_capacity;
105         if(capa <= oldCapa) return;
106
107         int NewCapacity = m_capacity + (m_capacity >> 1);
108         T* NewEle = new T[NewCapacity]();
109
110         for(int i = 0; i < m_size ; i++)
111         {
112                 NewEle[i] = this->m_element[TrueIndex(i)];
113         }
114
115         delete[] this->m_element;
116         this->m_element = NewEle;
117
118         //注意更新容量和头指针
119         m_capacity = NewCapacity;
```

```
120             m_front = 0;
121 
122     }
123 
124 template<class T>
125 std::ostream& operator<<(std::ostream& out ,CircleQueue<T> &que)
126 {
127     for(int i = 0; i< que.m_capacity; i++)
128     {
129             out << que.m_element[i] << ' ';
130     }
131     return out;
132 }
```

双向循环队列

相较于单向队列，多了下面三个功能：

rear可以根据TrueIndex(m_size -1)直接推出，无需再声明。

```
1     //获取尾部元素
2     T rear();
3 
4     //尾部删除
5     T dequeueRear();
6 
7     //头部入队
8     void enqueueFront(T Ele);
```

双向循环队列的索引映射多了一个-1的映射，因为头部入队且front=0时，就会在相对位置为-1的地方进行操作。

```
1     //索引映射
2     template<class T>
3     int CircleDeque<T>::TrueIndex(int index)
4     {
5         index += m_front;
6             //index < 0 :加一个capacity
```

```
            //index > 0 : 分两种情况，一种是小于capacity，保持不变
            //一种是大于capacity，要减一个capacity
            //目的：把不在0-capacity这个区间内的数都映射回这一段区间

            if(index < 0)
                    return index + m_capacity;
            else {
               return (index < m_capacity) ? index : index - m_capacity;
            }
            //模运算优化，2mcapacity > index + front的
    }

```

```
#pragma once
#define DEFAULT_CAPACITY 10
#include<iostream>
using namespace std;


template<class T>
class CircleDeque
{

public:

    template<class T>
    friend std::ostream& operator<<(std::ostream& out ,CircleDeque<T> &que);


    //初始化
    CircleDeque();

    //尾部入队
    void enqueue(T Ele);
    //索引映射
    int TrueIndex(int index);
```

```
    //头部出队
    T dequeue();
    //获取头元素
    T front();

    //获取尾部元素
    T rear();
    //尾部删除
    T dequeueRear();

    //头部入队
    void enqueueFront(T Ele);

    //动态扩容
    void DanaCapacity(int capa);

    //size
    int size()
    {
            return m_size;
    }

    //empty
    bool IsEmpty()
    {
            return m_size == 0;
    }


private:
    int m_front;
    int m_size;
    int* m_element;
    int m_capacity;
};


```

```
    //初始化
```

```cpp
template<class T>
 CircleDeque<T>::CircleDeque()
{
    m_size = 0;
    m_front = 0;
    m_capacity = DEFAULT_CAPACITY;
    m_element = new T[m_capacity]();
    //注意不要再声明新变量了
}

template<class T>
std::ostream& operator<<(std::ostream& out ,CircleDeque<T> &que)
{
    cout << "frontIndex: " << que.m_front << ' ' << " front: " << que.front()
    cout << "  content: ";
    for(int i = 0; i< que.m_capacity; i++)
    {
            out << que.m_element[i] << ' ';
    }
    return out;
}


    //尾部入队
template<class T>
    void CircleDeque<T>::enqueue(T Ele)
    {
            this->DanaCapacity(m_size+1);
            this->m_element[TrueIndex(m_size)] = Ele;
            m_size++;

    }



    //索引映射
    template<class T>
    int CircleDeque<T>::TrueIndex(int index)
    {
```

```
102         index += m_front;
103             //index < 0 :加一个capacity
104             //index > 0 : 分两种情况，一种是小于capacity，保持不变
105             //一种是大于capacity，要减一个capacity
106             //目的：把不在0-capacity这个区间内的数都映射回这一段区间
107 
108             if(index < 0)
109                     return index + m_capacity;
110             else {
111                return (index < m_capacity) ? index : index - m_capacity;
112             }
113             //模运算优化，2mcapacity > index + front的
114     }
115 
116     //头部出队
117     template<class T>
118     T CircleDeque<T>::dequeue()
119     {
120        T temp = m_element[m_front];
121        this->m_element[m_front] = NULL;
122        m_front = TrueIndex(1);
123        m_size--;
124        return temp;
125     }
126 
127     //获取头元素
128     template<class T>
129     T CircleDeque<T>::front()
130     {
131             return this->m_element[m_front];
132     }
133 
134     //动态扩容
135     template<class T>
136     void CircleDeque<T>::DanaCapacity(int capa)
137     {
138             int oldCapa = this->m_capacity;
139             if(capa <= oldCapa) return;
```

```
140 
```

```
141             int NewCapacity = m_capacity + (m_capacity >> 1);
142             T* NewEle = new T[NewCapacity]();
143 
144             for(int i = 0; i < m_size ; i++)
145             {
146                     NewEle[i] = this->m_element[TrueIndex(i)];
147             }
148 
149             delete[] this->m_element;
150             this->m_element = NewEle;
151 
152             //注意更新容量和头指针
153             m_capacity = NewCapacity;
154             m_front = 0;
155 
156     }
157 
158 
159     //获取尾部元素
160     template<class T>
161     T CircleDeque<T>::rear()
162     {
163             return m_element[TrueIndex(m_size - 1)];
164     }
165 
166 
167     //尾部删除
168     template<class T>
169     T CircleDeque<T>::dequeueRear()
170     {
171        int rear = TrueIndex(m_size - 1);
172        T temp = m_element[rear];
173        this->m_element[rear] = NULL;
174        m_size--;
175        return temp;
176     }
177 
178     //头部入队
179     template<class T>
```

```
180    void CircleDeque<T>::enqueueFront(T Ele)
181    {
182            this->DanaCapacity(m_size + 1);
183            m_front = TrueIndex(-1);
184            this->m_element[m_front] = Ele;
185            m_size++;
186    }
```