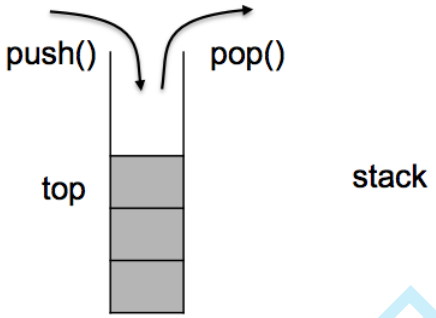
11.stack和queue

1. stack的介绍和使用

1.1 stack的介绍

> stack的文档介绍



1.2 stack的使用

函数说明	接口说明
stack()	构造空的栈
empty()	检测stack是否为空
size()	返回stack中元素的个数
top()	返回栈顶元素的引用
push()	将元素val压入stack中
pop()	将stack中尾部的元素弹出

最小栈

```
class MinStack
{
public:
 void push(int x)
 {
     // 只要是压栈，先将元素保存到_elem中
     _elem.push(x);

     // 如果x小于_min中栈顶的元素，将x再压入_min中
     if(_min.empty() || x <= _min.top())
         _min.push(x);
 }

 void pop()
 {
     // 如果_min栈顶的元素等于出栈的元素，_min顶的元素要移除
     if(_min.top() == _elem.top())
         _min.pop();
```

```
        _elem.pop();
    }

    int top(){return _elem.top();}
    int getMin(){return _min.top();}

private:
    // 保存栈中的元素
    std::stack<int> _elem;

    // 保存栈的最小值
    std::stack<int> _min;
};
```

栈的弹出压入序列

```
class Solution {
public:
    bool IsPopOrder(vector<int> pushV,vector<int> popV) {
        //入栈和出栈的元素个数必须相同
        if(pushV.size() != popV.size())
            return false;

        // 用s来模拟入栈与出栈的过程
        int outIdx = 0;
        int inIdx = 0;
        stack<int> s;

        while(outIdx < popV.size())
        {
            // 如果s是空，或者栈顶元素与出栈的元素不相等，就入栈
            while(s.empty() || s.top() != popV[outIdx])
            {
                if(inIdx < pushV.size())
                    s.push(pushV[inIdx++]);
                else
                    return false;
            }

            // 栈顶元素与出栈的元素相等，出栈
            s.pop();
            outIdx++;
        }

        return true;
    }
};
```

逆波兰表达式求值

```
class Solution {
public:
    int evalRPN(vector<string>& tokens) {
        stack<int> s;

        for (size_t i = 0; i < tokens.size(); ++i)
```

```
        {
            string& str = tokens[i];

            // str为数字
            if (!("+" == str || "-" == str || "*" == str || "/" == str))
            {
                s.push(atoi(str.c_str()));
            }
            else
            {
                // str为操作符
                int right = s.top();
                s.pop();

                int left = s.top();
                s.pop();

                switch (str[0])
                {
                case '+':
                    s.push(left + right);
                    break;
                case '-':
                    s.push(left - right);
                    break;
                case '*':
                    s.push(left * right);
                    break;
                case '/':
                    // 题目说明了不存在除数为0的情况
                    s.push(left / right);
                    break;
                }
            }
        }

        return s.top();
    }
};
```

请课后练习下面的OJ题目：

用两个栈实现队列

1.3 stack的模拟实现

从栈的接口中可以看出，栈实际是一种特殊的vector，因此使用vector完全可以模拟实现stack。

```
#include<vector>

namespace bite
{
    template<class T>
    class stack
    {
    public:
        stack() {}
        void push(const T& x) {_c.push_back(x);}
```

```
        void pop() {_c.pop_back();}
        T& top() {return _c.back();}
        const T& top()const {return _c.back();}
        size_t size()const {return _c.size();}
        bool empty()const {return _c.empty();}
    private:
        std::vector<T> _c;
    };
}
```

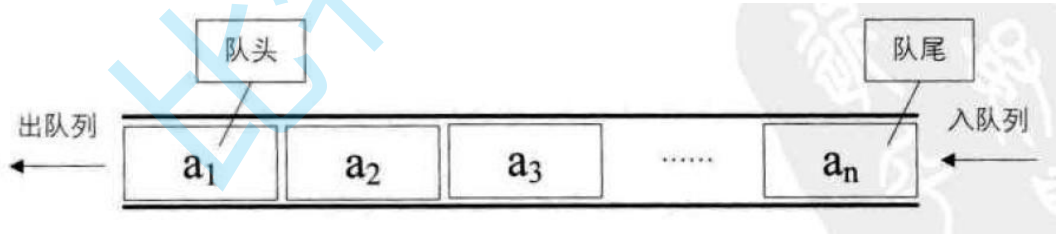
2. queue的介绍和使用

2.1 queue的介绍

queue的文档介绍

翻译：

1. 队列是一种容器适配器，专门用于在FIFO上下文(先进先出)中操作，其中从容器一端插入元素，另一端提取元素。
2. 队列作为容器适配器实现，容器适配器即将特定容器类封装作为其底层容器类，queue提供一组特定的成员函数来访问其元素。元素从队尾入队列，从队头出队列。
3. 底层容器可以是标准容器类模板之一，也可以是其他专门设计的容器类。该底层容器应至少支持以下操作:
    - empty：检测队列是否为空
    - size：返回队列中有效元素的个数
    - front：返回队头元素的引用
    - back：返回队尾元素的引用
    - push_back：在队列尾部入队列
    - pop_front：在队列头部出队列
4. 标准容器类deque和list满足了这些要求。默认情况下，如果没有为queue实例化指定容器类，则使用标准容器deque。



2.2 queue的使用

函数声明	接口说明
queue()	构造空的队列
empty()	检测队列是否为空，是返回true，否则返回false
size()	返回队列中有效元素的个数
front()	返回队头元素的引用
back()	返回队尾元素的引用
push()	在队尾将元素val入队列
pop()	将队头元素出队列

> 请课后练习下面的OJ题目：
>
> 用队列实现栈

2.3 queue的模拟实现

> 因为queue的接口中存在头删和尾插，因此使用vector来封装效率太低，故可以借助list来模拟实现queue，具体如下：

```
#include <list>
namespace bite
{
 template<class T>
    class queue
    {
    public:
        queue() {}
        void push(const T& x) {_c.push_back(x);}
        void pop() {_c.pop_front();}
        T& back() {return _c.back();}
        const T& back()const {return _c.back();}
        T& front() {return _c.front();}
        const T& front()const {return _c.front();}
        size_t size()const {return _c.size();}
        bool empty()const {return _c.empty();}
    private:
        std::list<T> _c;
    };
}
```

3.1 priority_queue的介绍和使用

3.1 priority_queue的介绍

> priority_queue文档介绍
>
> 翻译：
>
> 1. 优先队列是一种容器适配器，根据严格的弱排序标准，它的第一个元素总是它所包含的元素中最大的。
> 2. 此上下文类似于堆，在堆中可以随时插入元素，并且只能检索最大堆元素(优先队列中位于顶部的元素)。
> 3. 优先队列被实现为容器适配器，容器适配器即将特定容器类封装作为其底层容器类，queue提供一组特定的成员函数来访问其元素。元素从特定容器的“尾部”弹出，其称为优先队列的顶部。
> 4. 底层容器可以是任何标准容器类模板，也可以是其他特定设计的容器类。容器应该可以通过随机访问迭代器访问，并支持以下操作：
>   - empty()：检测容器是否为空
>   - size()：返回容器中有效元素个数
>   - front()：返回容器中第一个元素的引用
>   - push_back()：在容器尾部插入元素
>   - pop_back()：删除容器尾部元素

5. 标准容器类vector和deque满足这些需求。默认情况下，如果没有为特定的priority_queue类实例化指定容器类，则使用vector。
6. 需要支持随机访问迭代器，以便始终在内部保持堆结构。容器适配器通过在需要时自动调用算法函数make_heap、push_heap和pop_heap来自动完成此操作。

3.2 priority_queue的使用

优先级队列默认使用vector作为其底层存储数据的容器，在vector上又使用了堆算法将vector中元素构造成堆的结构，因此priority_queue就是堆，所有需要用到堆的位置，都可以考虑使用priority_queue。注意：默认情况下priority_queue是大堆。

函数声明	接口说明
priority_queue()/priority_queue(first, last)	构造一个空的优先级队列
empty( )	检测优先级队列是否为空，是返回true，否则返回false
top( )	返回优先级队列中最大(最小元素)，即堆顶元素
push(x)	在优先级队列中插入元素x
pop()	删除优先级队列中最大(最小)元素，即堆顶元素

【注意】

1. 默认情况下，priority_queue是大堆。

```
#include <vector>
#include <queue>
#include <functional>   // greater算法的头文件

void TestPriorityQueue()
{
  // 默认情况下，创建的是大堆，其底层按照小于号比较
  vector<int> v{3,2,7,6,0,4,1,9,8,5};
  priority_queue<int> q1;
  for (auto& e : v)
      q1.push(e);
  cout << q1.top() << endl;

  // 如果要创建小堆，将第三个模板参数换成greater比较方式
  priority_queue<int, vector<int>, greater<int>> q2(v.begin(), v.end());
  cout << q2.top() << endl;
}
```

2. 如果在priority_queue中放自定义类型的数据，用户需要在自定义类型中提供> 或者< 的重载。

```
class Date
{
public:
  Date(int year = 1900, int month = 1, int day = 1)
```

```
        : _year(year)
        , _month(month)
        , _day(day)
    {}

    bool operator<(const Date& d)const
    {
        return (_year < d._year) ||
               (_year == d._year && _month < d._month) ||
               (_year == d._year && _month == d._month && _day < d._day);
    }

    bool operator>(const Date& d)const
    {
        return (_year > d._year) ||
            (_year == d._year && _month > d._month) ||
            (_year == d._year && _month == d._month && _day > d._day);
    }

    friend ostream& operator<<(ostream& _cout, const Date& d)
    {
        _cout << d._year << "-" << d._month << "-" << d._day;
        return _cout;
    }

  private:
    int _year;
    int _month;
    int _day;
  };

  void TestPriorityQueue()
  {
    // 大堆，需要用户在自定义类型中提供<的重载
    priority_queue<Date> q1;
    q1.push(Date(2018, 10, 29));
    q1.push(Date(2018, 10, 28));
    q1.push(Date(2018, 10, 30));
    cout << q1.top() << endl;

    // 如果要创建小堆，需要用户提供>的重载
    priority_queue<Date, vector<Date>, greater<Date>> q2;
    q2.push(Date(2018, 10, 29));
    q2.push(Date(2018, 10, 28));
    q2.push(Date(2018, 10, 30));
    cout << q2.top() << endl;
  }
```

3.3 在OJ中的使用

[数组中第K个大的元素]

```
class Solution {
public:
int findKthLargest(vector<int>& nums, int k) {
   // 将数组中的元素先放入优先级队列中
   priority_queue<int> p(nums.begin(), nums.end());
```

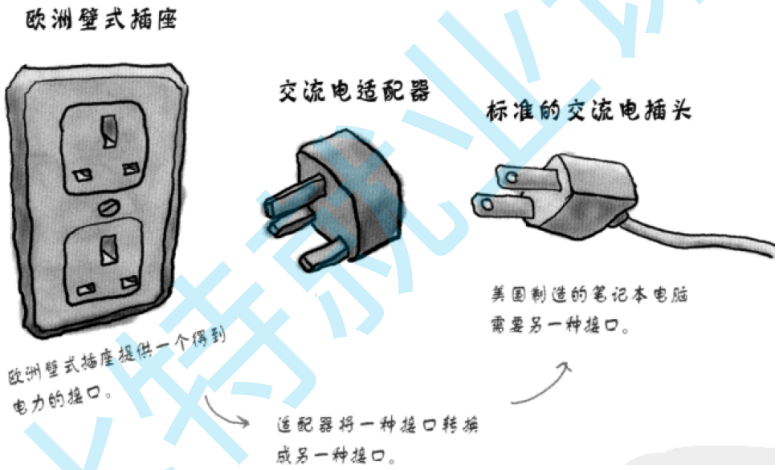
```
    // 将优先级队列中前k-1个元素删除掉
    for(int i= 0; i < k-1; ++i)
    {
        p.pop();
    }

    return p.top();
}
};
```

3.4 priority_queue的模拟实现

> 通过对priority_queue的底层结构就是堆，因此此处只需对对进行通用的封装即可。
>
> 优先级队列的模拟实现

4. 容器适配器

4.1 什么是适配器

> **适配器是一种设计模式**(设计模式是一套被反复使用的、多数人知晓的、经过分类编目的、代码设计经验的总结)，**该种模式是将一个类的接口转换成客户希望的另外一个接口。**
>
> 
>

4.2 STL标准库中stack和queue的底层结构

> 虽然stack和queue中也可以存放元素，但在STL中并没有将其划分在容器的行列，而是将其称为**容器适配器**，这是因为stack和队列只是对其他容器的接口进行了包装，STL中stack和queue默认使用deque，比如：
>
> class template
> **std::stack**
> `template <class T, class Container = deque<T> > class stack;`
>
> class template
> **std::queue**
> `template <class T, class Container = deque<T> > class queue;`

class template

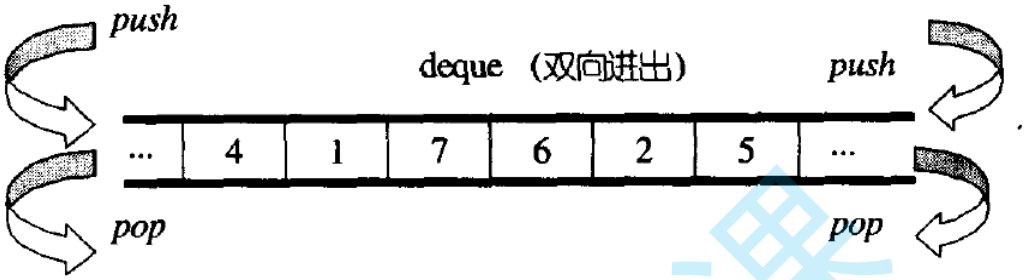
std::**priority_queue**

```
template <class T, class Container = vector<T>,
  class Compare = less<typename Container::value_type> > class priority_queue;
```

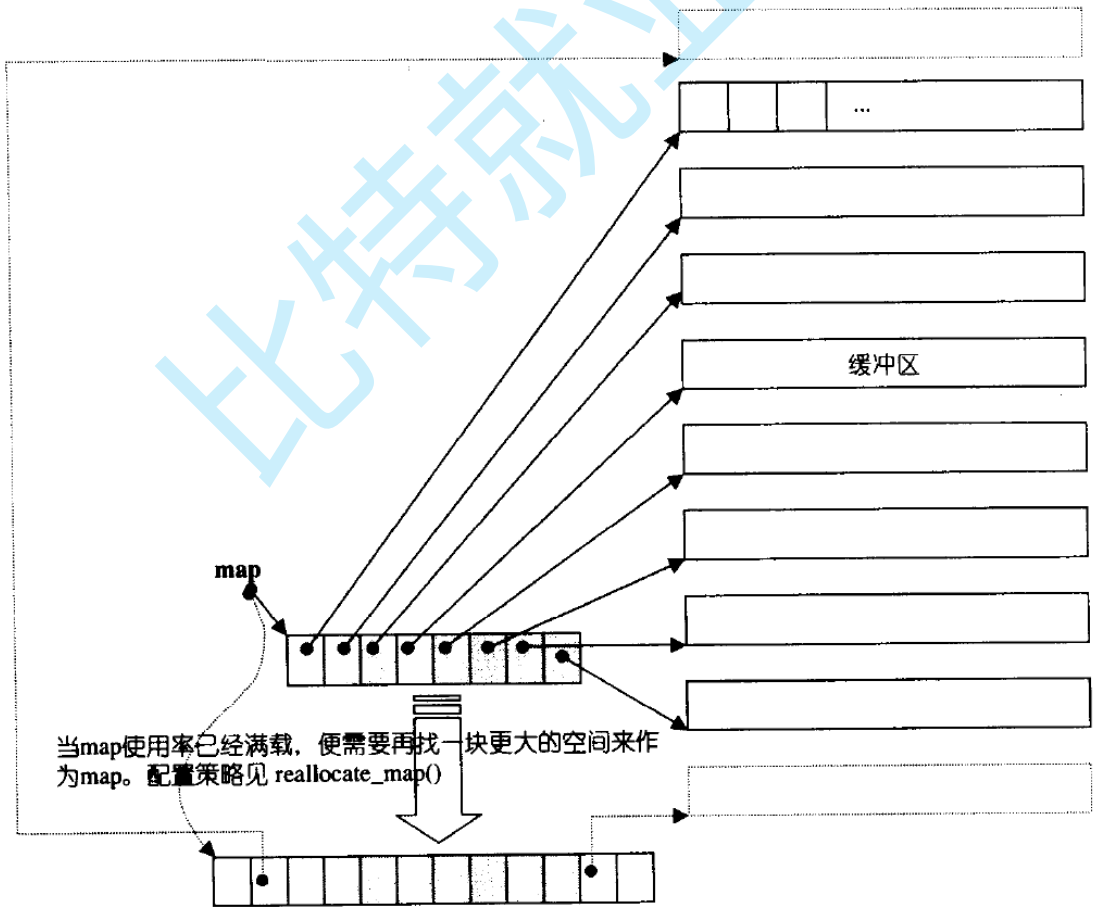
4.3 deque的简单介绍(了解)

4.3.1 deque的原理介绍

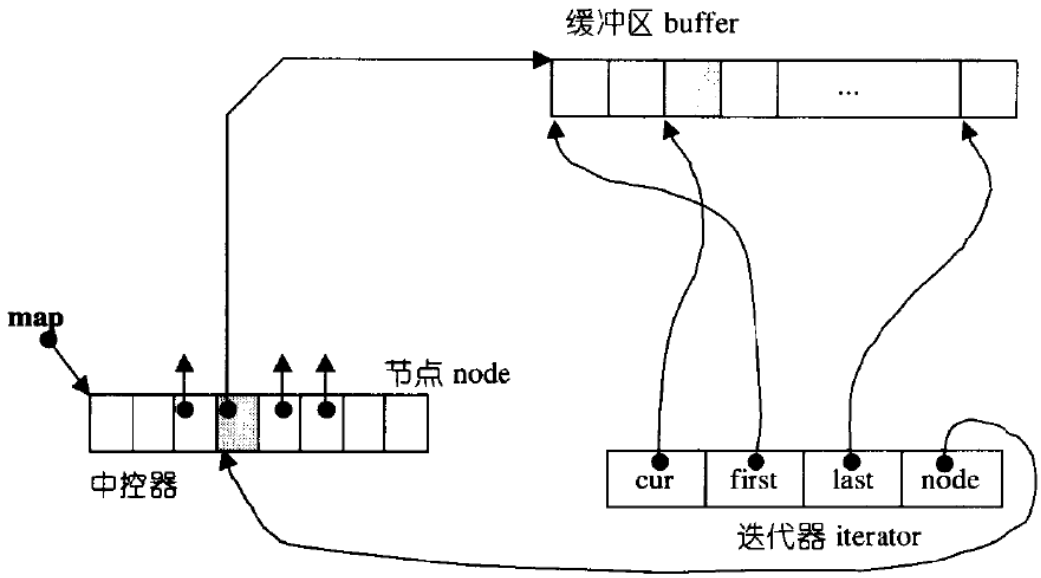
deque(双端队列)：是一种双开口的"连续"空间的数据结构，双开口的含义是：可以在头尾两端进行插入和删除操作，且时间复杂度为O(1)，与vector比较，头插效率高，不需要搬移元素；与list比较，空间利用率比较高。



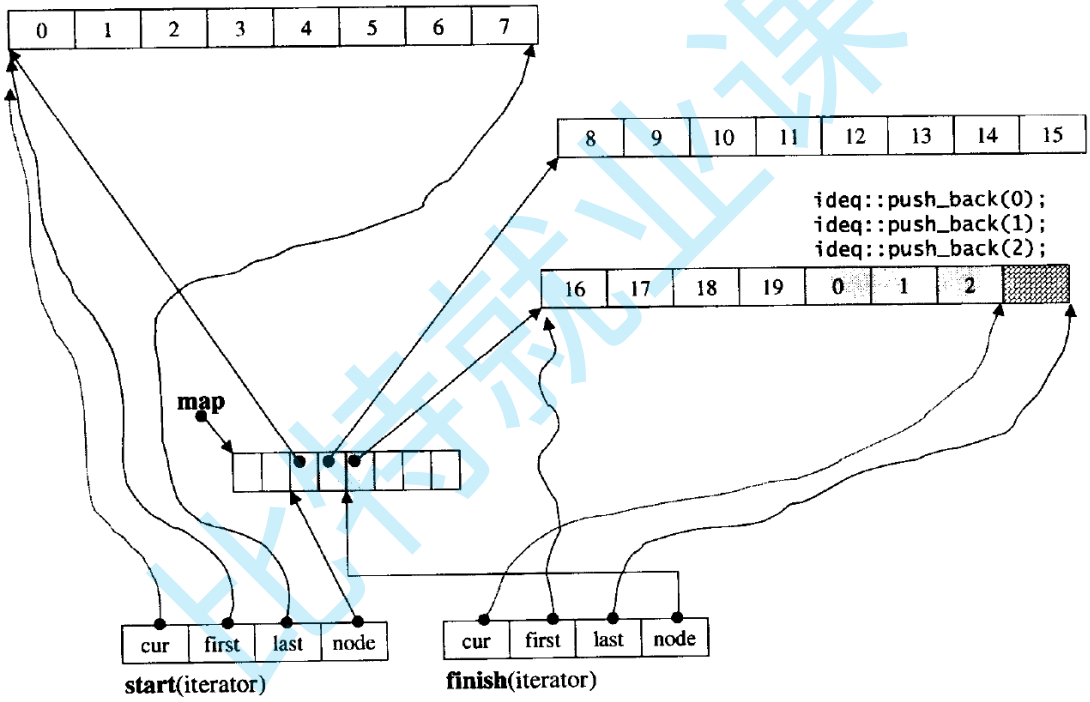
deque并不是真正连续的空间，而是由一段段连续的小空间拼接而成的，实际deque类似于一个动态的二维数组，其底层结构如下图所示：



双端队列底层是一段假象的连续空间，实际是分段连续的，为了维护其“整体连续”以及随机访问的假象，落在了deque的迭代器身上，因此deque的迭代器设计就比较复杂，如下图所示：



那deque是如何借助其迭代器维护其假想连续的结构呢?



4.3.2 deque的缺陷

> **与vector比较**，deque的优势是：头部插入和删除时，**不需要搬移元素，效率特别高**，而且在**扩容时，也不需要搬移大量的元素**，因此其效率是必vector高的。
>
> **与list比较**，其底层是连续空间，**空间利用率比较高**，不需要存储额外字段。
>
> 但是，**deque有一个致命缺陷：不适合遍历，因为在遍历时，deque的迭代器要频繁的去检测其是否移动到某段小空间的边界，导致效率低下**，而序列式场景中，可能需要经常遍历，因此**在实际中，需要线性结构时，大多数情况下优先考虑vector和list**，deque的应用并不多，而**目前能看到的一个应用就是，STL用其作为stack和queue的底层数据结构。**

4.4 为什么选择deque作为stack和queue的底层默认容器

> stack是一种后进先出的特殊线性数据结构，因此只要具有push_back()和pop_back()操作的线性结构，都可以作为stack的底层容器，比如vector和list都可以；queue是先进先出的特殊线性数据结构，只要具有push_back和pop_front操作的线性结构，都可以作为queue的底层容器，比如list。但是STL中对stack和queue默认选择deque作为其底层容器，主要是因为：
>
> 1. stack和queue不需要遍历(因此stack和queue没有迭代器)，只需要在固定的一端或者两端进行操作。
> 2. 在stack中元素增长时，deque比vector的效率高(扩容时不需要搬移大量数据)；queue中的元素增长时，deque不仅效率高，而且内存使用率高。
>
> 结合了deque的优点，而完美的避开了其缺陷。

4.5 STL标准库中对于stack和queue的模拟实现

4.5.1 stack的模拟实现

```
#include<deque>
namespace bite
{
    template<class T, class Con = deque<T>>
    //template<class T, class Con = vector<T>>
    //template<class T, class Con = list<T>>
    class stack
    {
    public:
        stack() {}
        void push(const T& x) {_c.push_back(x);}
        void pop() {_c.pop_back();}
        T& top() {return _c.back();}
        const T& top()const {return _c.back();}
        size_t size()const {return _c.size();}
        bool empty()const {return _c.empty();}
    private:
        Con _c;
    };
}
```

4.5.2 queue的模拟实现

```
#include<deque>

#include <list>
namespace bite
{
 template<class T, class Con = deque<T>>
 //template<class T, class Con = list<T>>
    class queue
    {
    public:
        queue() {}
        void push(const T& x) {_c.push_back(x);}
        void pop() {_c.pop_front();}
        T& back() {return _c.back();}
        const T& back()const {return _c.back();}
        T& front() {return _c.front();}
        const T& front()const {return _c.front();}
        size_t size()const {return _c.size();}
```

```
        bool empty()const {return _c.empty();}
    private:
        Con _c;
    };
}
```