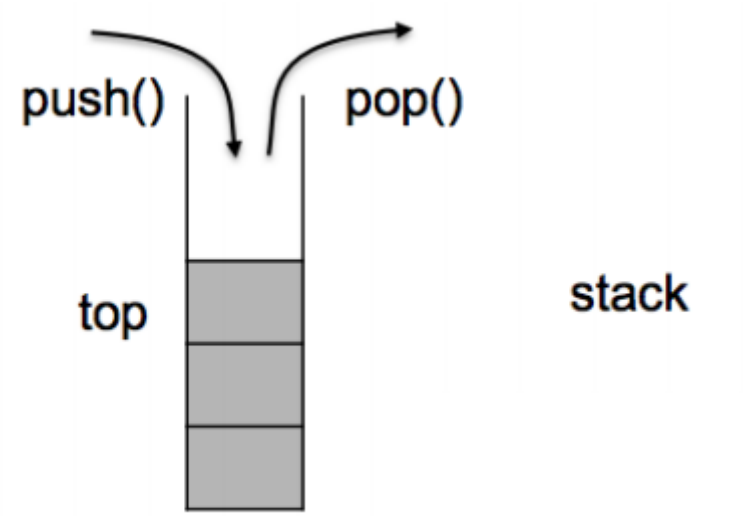
Stack和Queue复习

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

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。

stack模拟实现库中位置

这个模拟实现是基于前面复习的vector模拟实现而实现的，vector位置这里就不给出了。

代码块

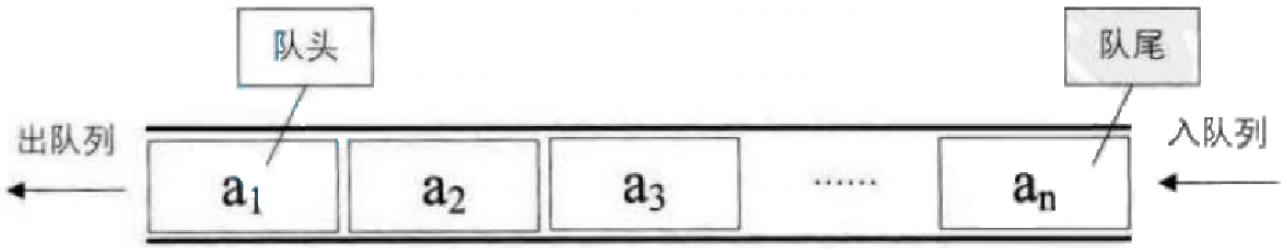
```
#pragma once
#include<iostream>
#include"vector.hpp"

using namespace Vector;

namespace Stack
{
    template<class T, class Container = vector<int>>
    class stack
    {
    public:
        stack() {}

        bool empty() const { return _con.empty(); }

        void pop() { _con.pop_back(); }

        void push(const T& val) { _con.push_back(val); }

        size_t size() const { return _con.size(); }

        void swap(stack& x) noexcept { _con.swap(); }

        T& top() { return _con.back(); }
        const T& top() const { return _con[0]; }

    private:
         Container _con;
    };
}
```

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

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模拟实现在库中位置

这个模拟实现是基于前面复习的list模拟实现而实现的，list位置这里就不给出了。

代码块

```
#pragma once
#include<iostream>
#include"list.hpp"

using namespace List;

namespace Queue
{
    template<class T, class Container = list<T> >
    class queue
    {
    public:
        queue() {}

        T& back(){ return _con.end()._node->_prev->_data; }
        const T& back() const { return _con.end()._node->_prev->_data; }

        bool empty() const { return _con.empty(); }

        T& front() { return _con.begin()._node->_data; }
        const T& front() const { return _con.begin()._node->_data; }

        void pop() { _con.pop_front(); }
        void push(const T& val) { _con.push_back(val); }

        size_t size() const { return _con.size(); }

        void swap(queue& x) noexcept { _con.swap(); }

    private:
            Container _con;
```

```
32        };
33    }
34
```

3. priority_queue的介绍和使用

3.1 priority_queue的介绍

priority_queue文档介绍

> 翻译：
>
> a. 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。
>
> b. 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)。
>
> c. 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。
>
> d. 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
>
> e. 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。
>
> f. 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。
>
> g. 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。
>
> h. 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()	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)的元素,即堆顶元素

【注意】

- 默认情况下，priority_queue是大堆。

代码块

```
#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;
}

```

- **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));
```

```
48      q2.push(Date(2018, 10, 28));
49      q2.push(Date(2018, 10, 30));
50      cout << q2.top() << endl;
51  }
52
```

3.3 priority_queue模拟实现

priority_queue在代码库中位置

这个模拟实现是基于前面复习的vector模拟实现而实现的，vector位置这里就不给出了。

push_heap和pop_heap是基于DeepSeek实现的，但是不对，在之后把二叉树部分复习完再来尝试写一下，目前就先用库中的接口。

代码块

```
1   #pragma once
2   #include "vector.hpp"
3   #include <functional>
4
5   using namespace Vector;
6
7   namespace Priority_queue
8   {
9       template<class T>
10      struct less {
11          bool operator() (const T& x, const T& y) const { return x < y; }
12      };
13
14      template<class T>
15      struct greater {
16          bool operator() (const T& x, const T& y) const { return x > y; }
17      };
18
19      template<typename RandomIt, typename Compare>
20      void push_heap(RandomIt first, RandomIt last, Compare comp)
21      {
22          if (last - first < 2) return;
23
24          auto len = last - first;
25          auto child = len - 1;
26          auto parent = (child - 1) / 2;
27
28          while (child > 0 && comp(first[child], first[parent]))
29          {
30              std::swap(first[parent], first[child]);
```

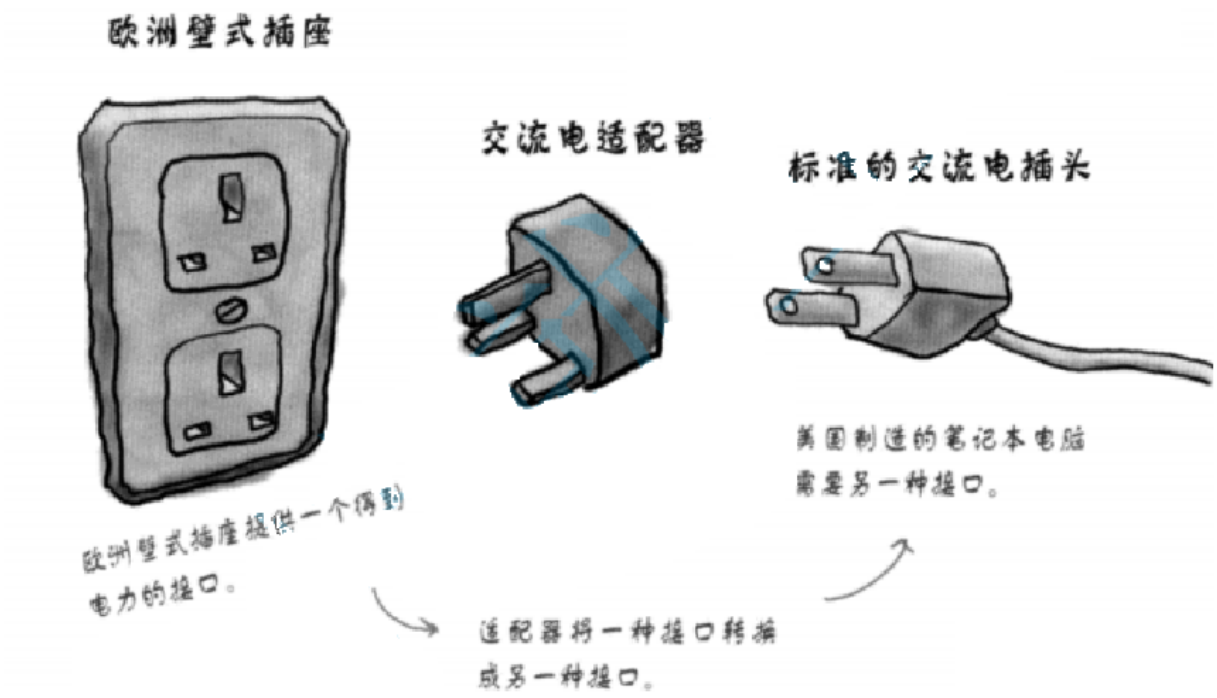
```
31              child = parent;
32              parent = (child - 1) / 2;
33          }
34      }
35
36      template<typename RandomIt, typename Compare>
37      void pop_heap(RandomIt first, RandomIt last, Compare comp)
38      {
39          if (last - first < 2) return;
40
41          auto len = last - first - 1;
42          auto parent = 0;
43          auto child = 2 * parent + 1;
44
45          while (child < len)
46          {
47              if (child + 1 < len && comp(first[child], first[child + 1]))
    child++;
48
49              if (comp(first[child], first[parent]))
50              {
51                  std::swap(first[parent], first[child]);
52                  parent = child;
53                  child = 2 * parent + 1;
54              }
55
56              else break;
57          }
58      }
59
60      template <class T, class Container = vector<T>, class Compare =
    less<T> >
61      class priority_queue
62      {
63      public:
64          explicit priority_queue(const Compare& comp = Compare())
65                  : _con(), _comp(comp) { }
66
67          void pop()
68          {
69              std::pop_heap(_con.begin(), _con.end(), _comp);
70              _con.pop_back();
71          }
72
73          void push(const T& val)
74          {
75              _con.push_back(val);
```

```
76              std::push_heap(_con.begin(), _con.end(), _comp);
77          }
78
79          bool empty() const { return _con.empty(); }
80          size_t size() const { return _con.size(); }
81          void swap(priority_queue& x) noexcept { _con.swap(x); }
82          const T& top() const { return _con.front(); }
83
84      private:
85              Container _con;
86              Compare  comp;
```

4. 容器适配器

4.1 什么是容器适配器

> 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：

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

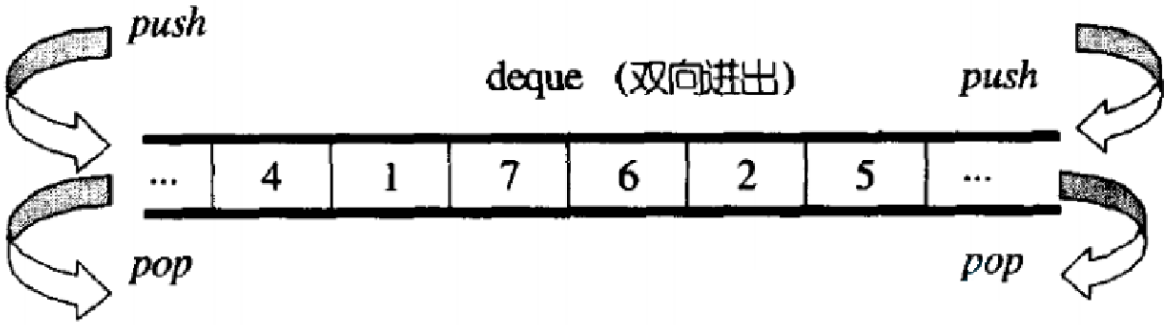
std::priority_queue

```
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

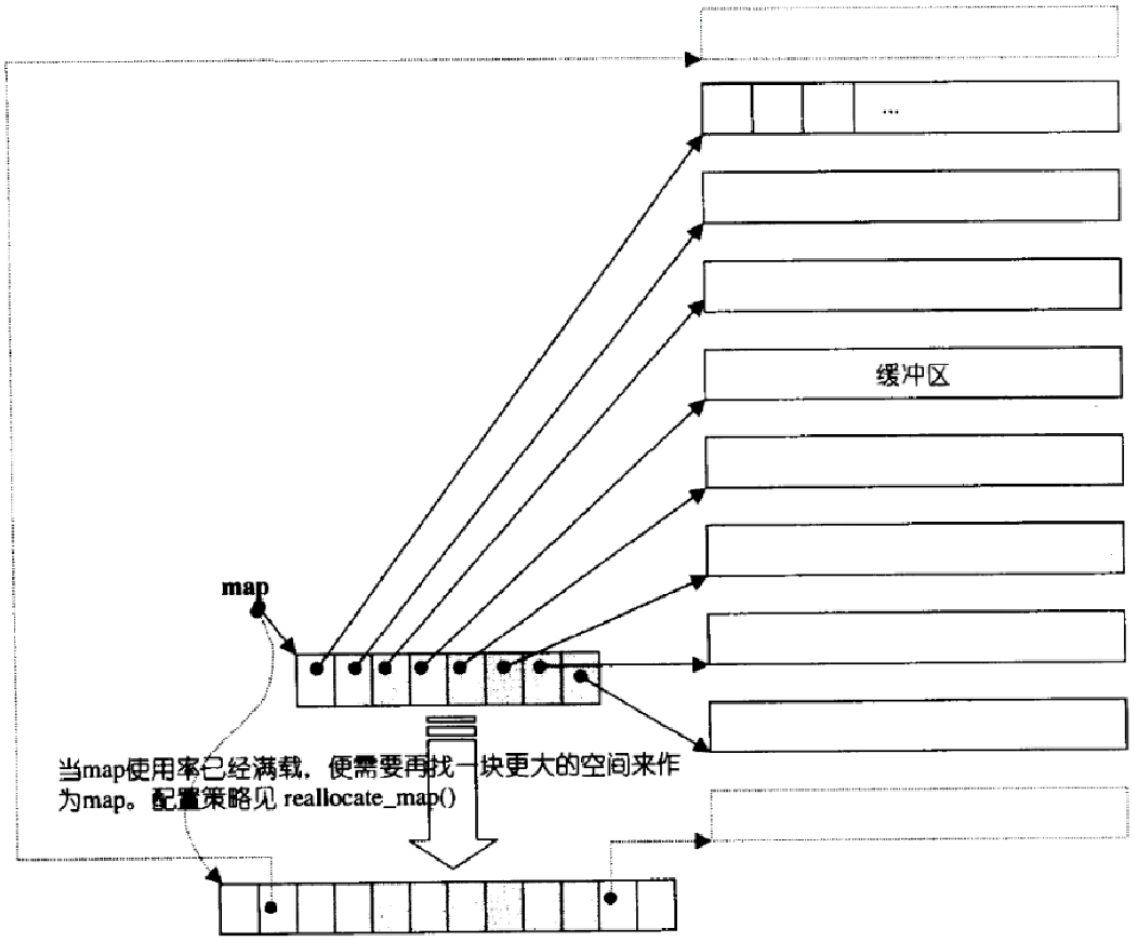
4.3.1 deque的原理介绍

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。



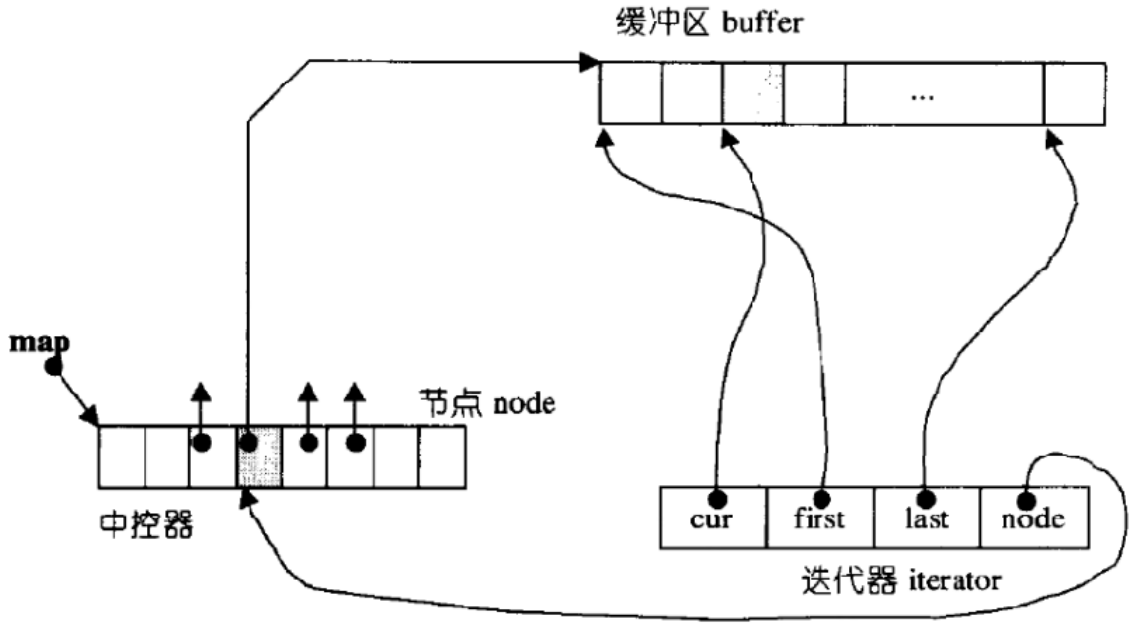
deque并不是真正连续的空间，而是由一段段连续的小空间拼接而成的，实际deque类似于一个

动态的二维数组，其底层结构如下图所示：

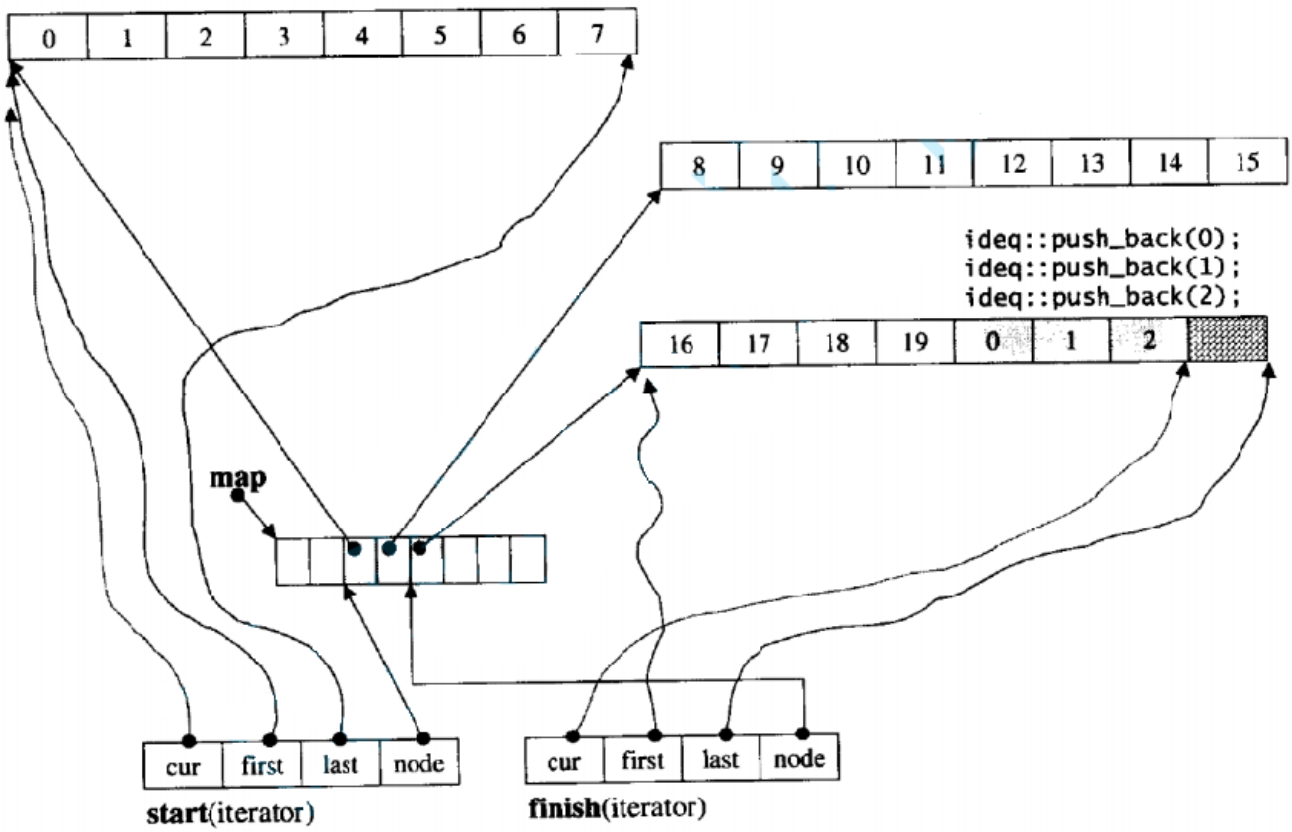


双端队列底层是一段假象的连续空间，实际是分段连续的，为了维护其“整体连续”以及随机访问

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**，因此其效率是比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：

i. 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。

ii. 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。