

Lesson11---stack和queue

[本节目标]

- 1. stack的介绍和使用
- 2. queue的介绍和使用
- 3. priority_queue的介绍和使用
- 4. 容器适配器
- 5. 本章总结

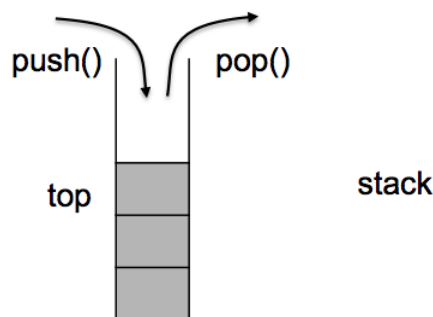
1. stack的介绍和使用

1.1 stack的介绍

[stack的文档介绍](#)

翻译：

1. stack是一种容器适配器，专门用在具有后进先出操作的上下文环境中，其删除只能从容器的一端进行元素的插入与提取操作。
2. stack是作为容器适配器被实现的，容器适配器即是对特定类封装作为其底层的容器，并提供一组特定的成员函数来访问其元素，将特定类作为其底层的，元素特定容器的尾部(即栈顶)被压入和弹出。
3. stack的底层容器可以是任何标准的容器类模板或者一些其他特定的容器类，这些容器类应该支持以下操作：
 - empty：判空操作
 - back：获取尾部元素操作
 - push_back：尾部插入元素操作
 - pop_back：尾部删除元素操作
4. 标准容器vector、deque、list均符合这些需求，默认情况下，如果没有为stack指定特定的底层容器，默认情况下使用deque。



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](#)

```
1 class MinStack
2 {
3 public:
4     void push(int x)
5     {
6         // 只要是压栈，先将元素保存到_elem中
7         _elem.push(x);
8
9         // 如果x小于_min中栈顶的元素，将x再压入_min中
10        if(_min.empty() || x <= _min.top())
11            _min.push(x);
12    }
13
14    void pop()
15    {
16        // 如果_min栈顶的元素等于出栈的元素，_min顶的元素要移除
17        if(_min.top() == _elem.top())
18            _min.pop();
19
20        _elem.pop();
21    }
22
23    int top(){return _elem.top();}
24    int getMin(){return _min.top();}
25
26 private:
27     // 保存栈中的元素
28     std::stack<int> _elem;
29
30     // 保存栈的最小值
31     std::stack<int> _min;
32 };
```

[栈的弹出压入序列](#)

```

1  class Solution {
2  public:
3      bool IsPopOrder(vector<int> pushV, vector<int> popV) {
4          //入栈和出栈的元素个数必须相同
5          if(pushV.size() != popV.size())
6              return false;
7
8          // 用s来模拟入栈与出栈的过程
9          int outIdx = 0;
10         int inIdx = 0;
11         stack<int> s;
12
13         while(outIdx < popV.size())
14         {
15             // 如果s是空, 或者栈顶元素与出栈的元素不相等, 就入栈
16             while(s.empty() || s.top() != popV[outIdx])
17             {
18                 if(inIdx < pushV.size())
19                     s.push(pushV[inIdx++]);
20                 else
21                     return false;
22             }
23
24             // 栈顶元素与出栈的元素相等, 出栈
25             s.pop();
26             outIdx++;
27         }
28
29         return true;
30     }
31 };

```

逆波兰表达式求值

```

1  class Solution {
2  public:
3      int evalRPN(vector<string>& tokens) {
4          stack<int> s;
5
6          for (size_t i = 0; i < tokens.size(); ++i)
7          {
8              string& str = tokens[i];
9
10             // str为数字
11             if (!("+ " == str || "- " == str || "*" == str || "/" == str))
12             {
13                 s.push(atoi(str.c_str()));
14             }
15             else
16             {
17                 // str为操作符
18
19                 int right = s.top();

```

```

19         s.pop();
20
21         int left = s.top();
22         s.pop();
23
24         switch (str[0])
25         {
26             case '+':
27                 s.push(left + right);
28                 break;
29             case '-':
30                 s.push(left - right);
31                 break;
32             case '*':
33                 s.push(left * right);
34                 break;
35             case '/':
36                 // 题目说明了不存在除数为0的情况
37                 s.push(left / right);
38                 break;
39         }
40     }
41 }
42
43 return s.top();
44 }
45 };

```

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。

```

1  #include<vector>
2
3  namespace bite
4  {
5      template<class T>
6      class stack
7      {
8      public:
9          stack() {}
10         void push(const T& x) {_c.push_back(x);}
11         void pop() {_c.pop_back();}
12         T& top() {return _c.back();}
13         const T& top()const {return _c.back();}
14         size_t size()const {return _c.size();}
15         bool empty()const {return _c.empty();}
16     private:
17         std::vector<T> _c;

```

```
18     };  
19 }
```

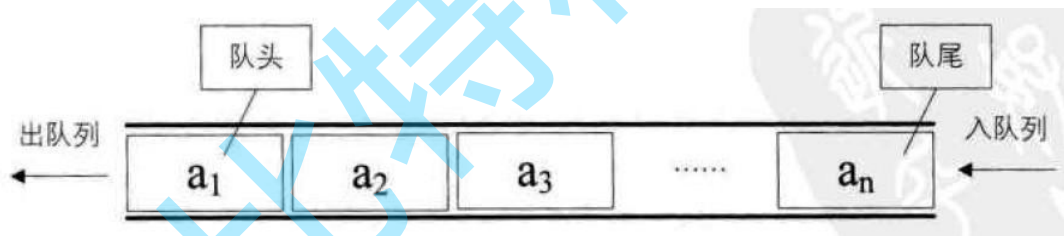
2. queue的介绍和使用

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：

```
1  #include <list>
2  namespace bite
3  {
4      template<class T>
5      class queue
6      {
7      public:
8          queue() {}
9          void push(const T& x) {_c.push_back(x);}
10         void pop() {_c.pop_front();}
11         T& back() {return _c.back();}
12         const T& back()const {return _c.back();}
13         T& front() {return _c.front();}
14         const T& front()const {return _c.front();}
15         size_t size()const {return _c.size();}
16         bool empty()const {return _c.empty();}
17     private:
18         std::list<T> _c;
19     };
20 }
```

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：
 - o empty(): 检测容器是否为空
 - o size(): 返回容器中有效元素个数
 - o front(): 返回容器中第一个元素的引用
 - o 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
<code>priority_queue()/priority_queue(first, last)</code>	构造一个空的优先级队列
<code>empty()</code>	检测优先级队列是否为空，是返回true，否则返回false
<code>top()</code>	返回优先级队列中最大(最小元素)，即堆顶元素
<code>push(x)</code>	在优先级队列中插入元素x
<code>pop()</code>	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。

```
1  #include <vector>
2  #include <queue>
3  #include <functional> // greater算法的头文件
4
5  void TestPriorityQueue()
6  {
7      // 默认情况下，创建的是大堆，其底层按照小于号比较
8      vector<int> v{3,2,7,6,0,4,1,9,8,5};
9      priority_queue<int> q1;
10     for (auto& e : v)
11         q1.push(e);
12     cout << q1.top() << endl;
13
14     // 如果要创建小堆，将第三个模板参数换成greater比较方式
15     priority_queue<int, vector<int>, greater<int>> q2(v.begin(), v.end());
16     cout << q2.top() << endl;
17 }
```

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。

```

1  class Date
2  {
3  public:
4      Date(int year = 1900, int month = 1, int day = 1)
5          : _year(year)
6            , _month(month)
7            , _day(day)
8      {}
9
10     bool operator<(const Date& d) const
11     {
12         return (_year < d._year) ||
13                (_year == d._year && _month < d._month) ||
14                (_year == d._year && _month == d._month && _day < d._day);
15     }
16
17     bool operator>(const Date& d) const
18     {
19         return (_year > d._year) ||
20                (_year == d._year && _month > d._month) ||
21                (_year == d._year && _month == d._month && _day > d._day);
22     }
23
24     friend ostream& operator<<(ostream& _cout, const Date& d)
25     {
26         _cout << d._year << "-" << d._month << "-" << d._day;
27         return _cout;
28     }
29
30 private:
31     int _year;
32     int _month;
33     int _day;
34 };
35
36 void TestPriorityQueue()
37 {
38     // 大堆, 需要用户在自定义类型中提供<的重载
39     priority_queue<Date> q1;
40     q1.push(Date(2018, 10, 29));
41     q1.push(Date(2018, 10, 28));
42     q1.push(Date(2018, 10, 30));
43     cout << q1.top() << endl;
44
45     // 如果要创建小堆, 需要用户提供>的重载
46     priority_queue<Date, vector<Date>, greater<Date>> q2;
47     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 }

```


3.3 在OJ中的使用

数组中第K个大的元素

```
1 class Solution {
2 public:
3     int findKthLargest(vector<int>& nums, int k) {
4         // 将数组中的元素先放入优先级队列中
5         priority_queue<int> p(nums.begin(), nums.end());
6
7         // 将优先级队列中前k-1个元素删除掉
8         for(int i= 0; i < k-1; ++i)
9         {
10             p.pop();
11         }
12
13         return p.top();
14     }
15 };
```

3.4 priority_queue的模拟实现

通过对priority_queue的底层结构就是堆，因此此处只需对堆进行通用的封装即可。

```
1 #include <vector>
2 // priority_queue--->堆
3 namespace bite
4 {
5     template<class T>
6     struct less
7     {
8         bool operator()(const T& left, const T& right)
9         { return left < right; }
10    };
11
12    template<class T>
13    struct greater
14    {
15        bool operator()(const T& left, const T& right)
16        { return left > right; }
17    };
18
19    template<class T, class Container=std::vector<T>, class Compare=less<T>>
20    class priority_queue
21    {
22    public:
23        // 创造空的优先级队列
24        priority_queue(): c(){}
25
26        template<class Iterator>
27        priority_queue(Iterator first, Iterator last)
28            : c(first, last)
```

```

29     {
30         // 将c中的元素调整成堆的结构
31         int count = c.size();
32         int root = ((count - 2) >> 1);
33         for (; root >= 0; root--)
34             AdjustDown(root);
35     }
36
37     void push(const T& data)
38     {
39         c.push_back(data);
40         AdjustUP(c.size() - 1);
41     }
42
43     void pop()
44     {
45         if (empty())
46             return;
47
48         swap(c.front(), c.back());
49         c.pop_back();
50         AdjustDown(0);
51     }
52
53     size_t size()const
54     { return c.size(); }
55
56     bool empty()const
57     { return c.empty(); }
58
59     // 堆顶元素不允许修改, 因为: 堆顶元素修改可以会破坏堆的特性
60     const T& top()const
61     { return c.front(); }
62 private:
63     // 向上调整
64     void AdjustUP(int child)
65     {
66         int parent = ((child - 1) >> 1);
67         while (child)
68         {
69             if (Com()(c[parent], c[child]))
70             {
71                 swap(c[child], c[parent]);
72                 child = parent;
73                 parent = ((child - 1) >> 1);
74             }
75             else
76             {
77                 return;
78             }
79         }
80     }

```

```

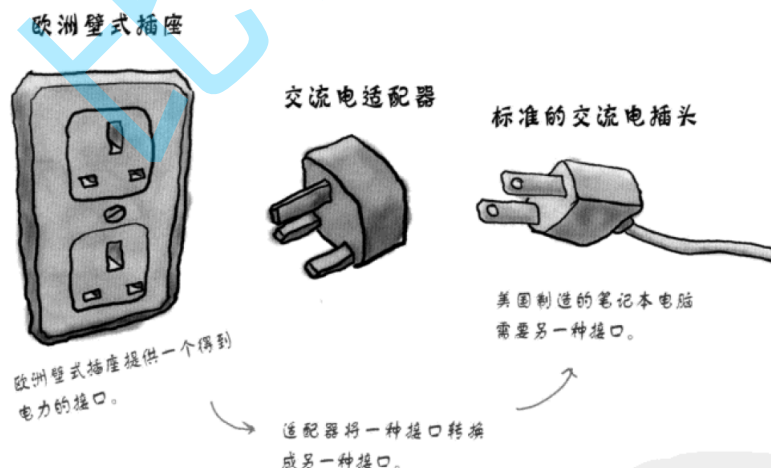
82 // 向下调整
83 void AdjustDown(int parent)
84 {
85     int child = parent * 2 + 1;
86     while (child < c.size())
87     {
88         // 找以parent为根的较大的孩子
89         if (child + 1 < c.size() && Com()(c[child], c[child+1]))
90             child += 1;
91
92         // 检测双亲是否满足情况
93         if (Com()(c[parent], c[child]))
94         {
95             swap(c[child], c[parent]);
96             parent = child;
97             child = parent * 2 + 1;
98         }
99         else
100             return;
101     }
102 }
103 private:
104     Container c;
105 };
106 }

```

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;
```

1

class template

std::queue

```
template <class T, class Container = deque<T> > class queue;
```

class template

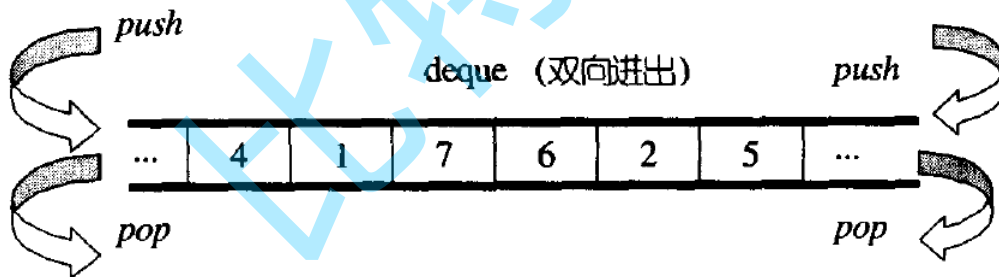
std::priority_queue

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

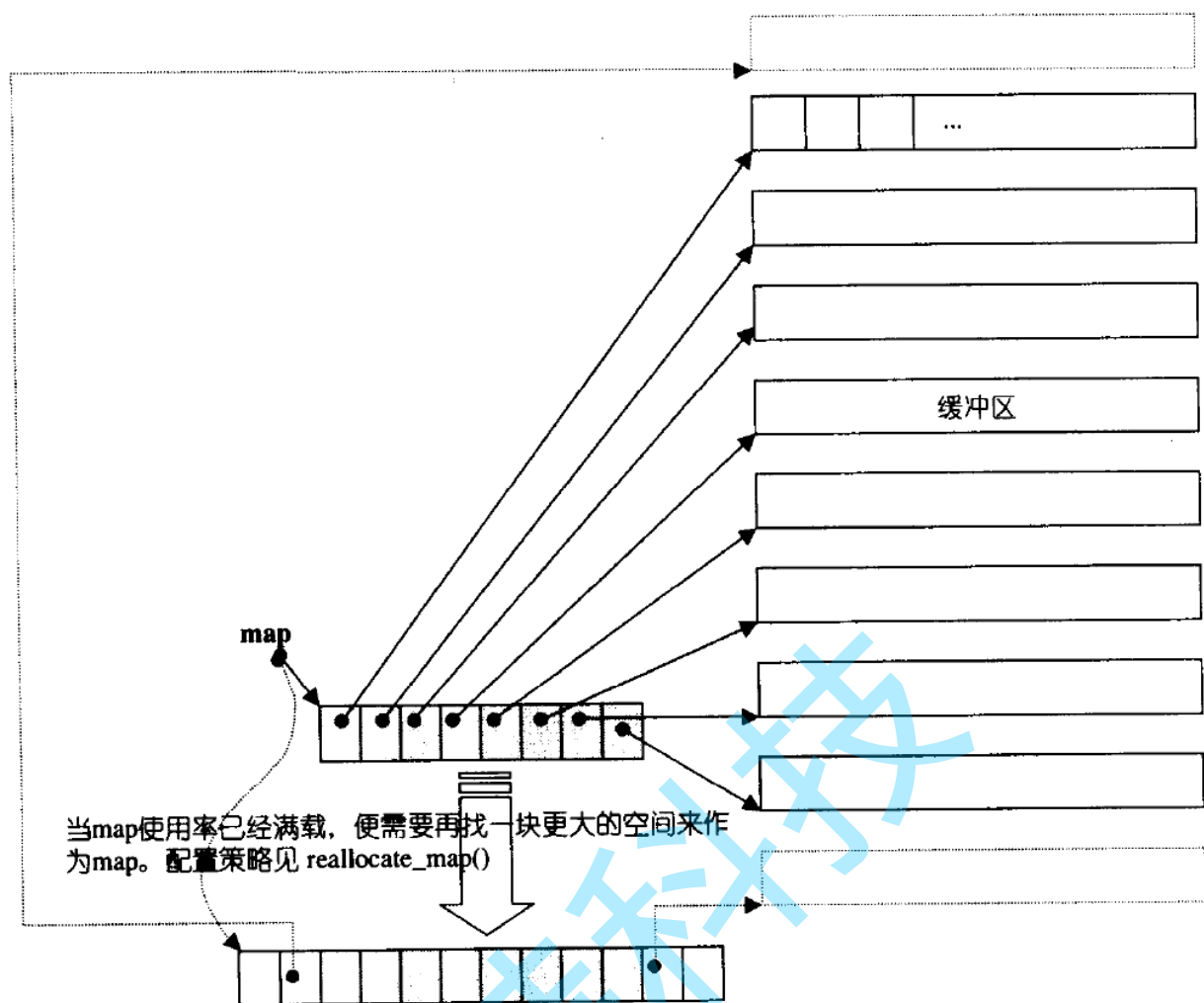
4.3 deque的简单介绍(了解)

4.3.1 deque的原理介绍

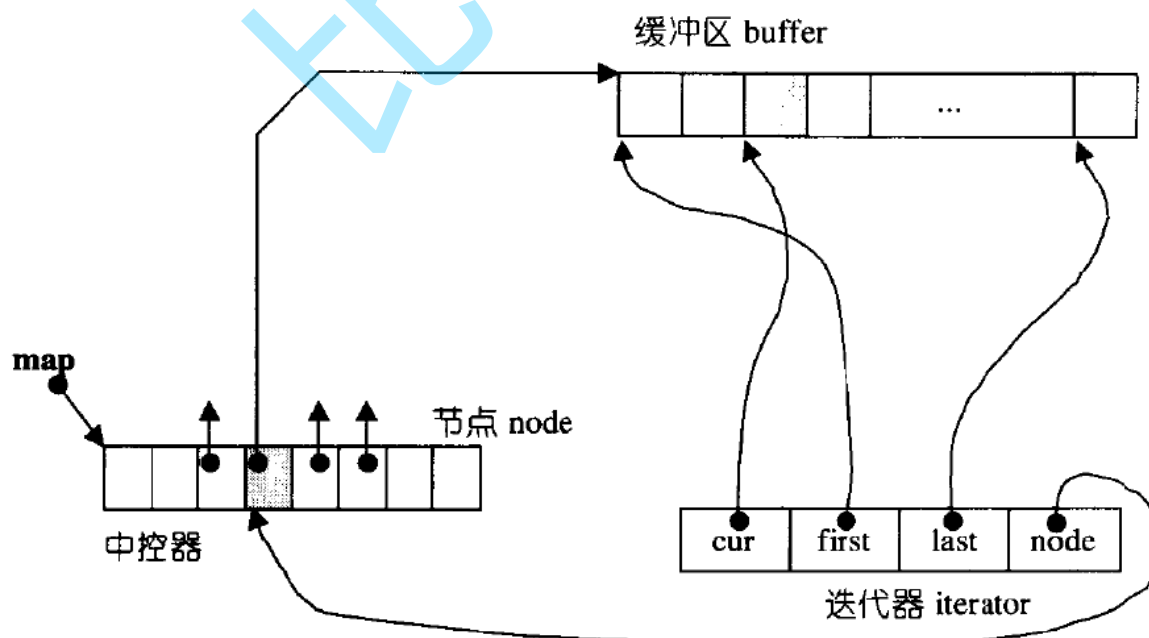
deque(双端队列): 是一种双开口的"连续"空间的数据结构, 双开口的含义是: 可以在头尾两端进行插入和删除操作, 且时间复杂度为 $O(1)$, 与vector比较, 头插效率高, 不需要搬移元素; 与list比较, 空间利用率比较高。



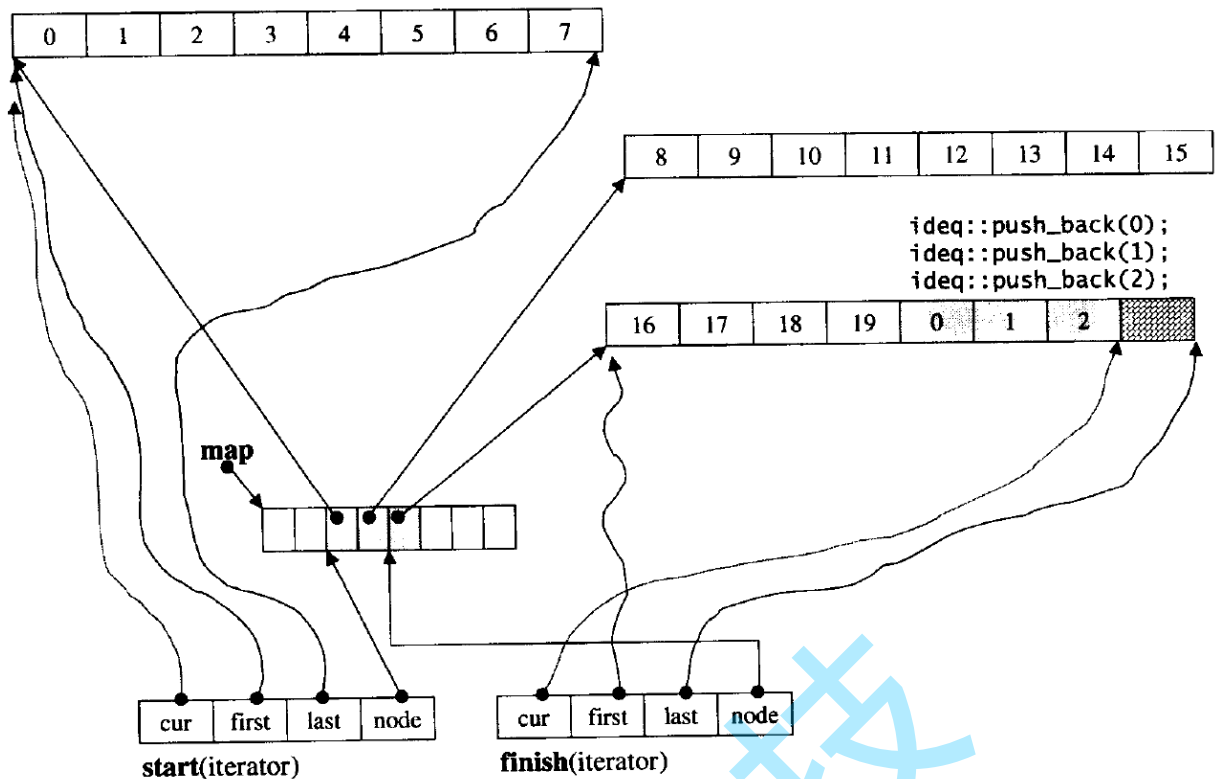
deque并不是真正连续的空间, 而是由一段段连续的小空间拼接而成的, 实际deque类似于一个动态的二维数组, 其底层结构如下图所示:



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(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

```
1 #include<deque>
2 namespace bite
3 {
```

```

4     template<class T, class Con = deque<T>>
5     //template<class T, class Con = vector<T>>
6     //template<class T, class Con = list<T>>
7     class stack
8     {
9     public:
10        stack() {}
11        void push(const T& x) {_c.push_back(x);}
12        void pop() {_c.pop_back();}
13        T& top() {return _c.back();}
14        const T& top()const {return _c.back();}
15        size_t size()const {return _c.size();}
16        bool empty()const {return _c.empty();}
17    private:
18        Con _c;
19    };
20 }

```

4.5.2 queue的模拟实现

```

1  #include<deque>
2
3  #include <list>
4  namespace bite
5  {
6      template<class T, class Con = deque<T>>
7      //template<class T, class Con = list<T>>
8      class queue
9      {
10     public:
11        queue() {}
12        void push(const T& x) {_c.push_back(x);}
13        void pop() {_c.pop_front();}
14        T& back() {return _c.back();}
15        const T& back()const {return _c.back();}
16        T& front() {return _c.front();}
17        const T& front()const {return _c.front();}
18        size_t size()const {return _c.size();}
19        bool empty()const {return _c.empty();}
20    private:
21        Con _c;
22    };
23 }

```