第二次讨论课

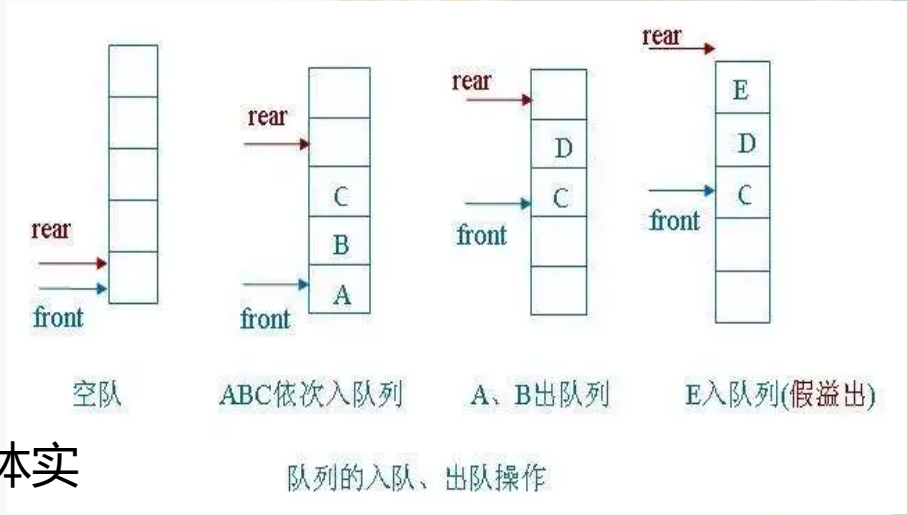
STL序列式容器探讨
队列（queue）

Discussion on STL sequential container

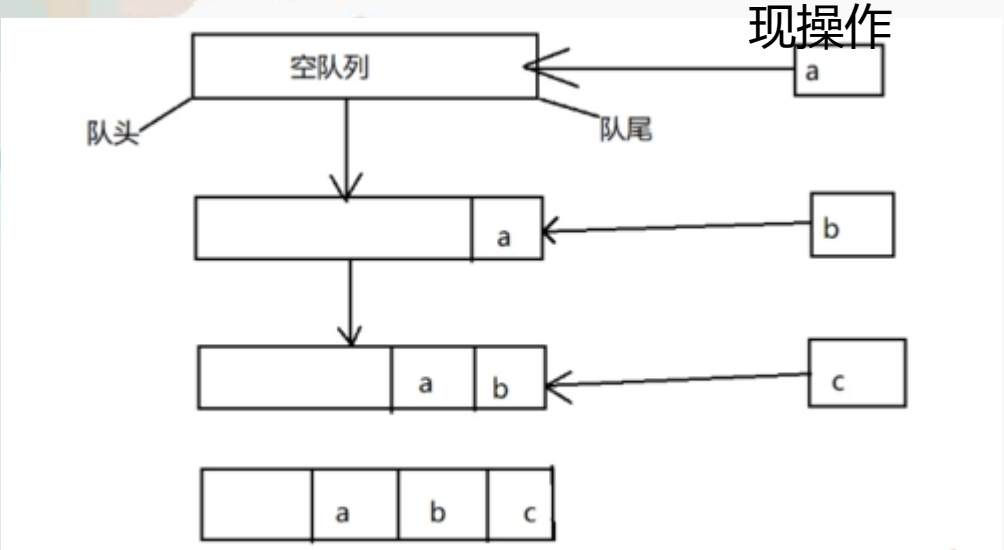
目录

1.1、概述：

队列是一种特殊的线性表，特殊之处在于它只允许在表的前端（front）进行删除操作，而在表的后端（rear）进行插入操作，和栈一样，队列是一种操作受限制的线性表。进行插入操作的端称为队尾，进行删除操作的端称为队头。队列中没有元素时，称为空队列。



关键基本操作的具体实现操作



队列的数据元素又称为队列元素。在队列中插入一个队列元素称为入队，从队列中删除一个队列元素称为出队。因为队列只允许在一端插入，在另一端删除，所以只有最早进入队列的元素才能最先从队列中删除，故队列又称为先进先出（FIFO—first in first out）线性表。

1.2、概述：

顺序队列：

1.建立顺序队列结构必须为其静态分配或动态申请一片连续的存储空间，并设置两个指针进行管理。一个是队头指针front，它指向队头元素；另一个是队尾指针rear，它指向下一个入队元素的存储位置，如图所示：

2.每次在队尾插入一个元素是，rear增1；每次在队头删除一个元素时，front增1。随着插入和删除操作的进行，队列元素的个数不断变化，队列所占的存储空间也在为队列结构所分配的连续空间中移动。当front=rear时，队列中没有任何元素，称为空队列。当rear增加到指向分配的连续空间之外时，队列无法再插入新元素，但这时往往还有大量可用空间未被占用，这些空间是已经出队的队列元素曾经占用过得存储单元。

3.顺序队列中的溢出现象：

（1）"下溢"现象：当队列为空时，做出队运算产生的溢出现象。“下溢”是正常现象，常用作程序控制转移的条件。

（2）"真上溢"现象：当队列满时，做进栈运算产生空间溢出的现象。“真上溢”是一种出错状态，应设法避免。

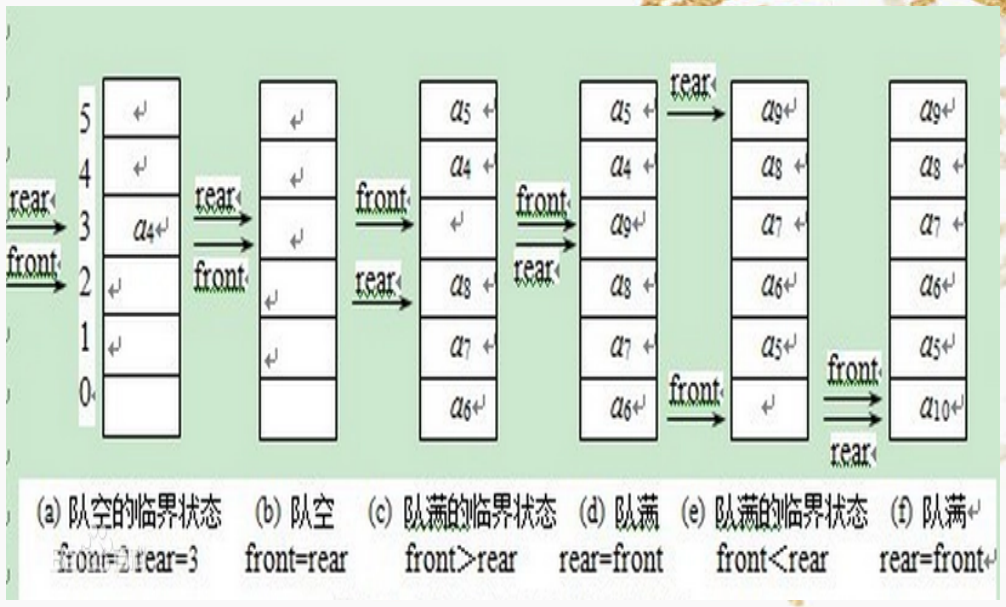
（3）"假上溢"现象：由于入队和出队操作中，头尾指针只增加不减小，致使被删元素的空间永远无法重新利用。当队列中实际的元素个数远远小于向量空间的规模时，也可能由于尾指针已超越向量空间的上界而不能做入队操作。该现象称为"假上溢"现象。

1.3、概述：

循环队列：

1.在实际使用队列时，为了使队列空间能重复使用，往往对队列的使用方法稍加改进：无论插入或删除，一旦rear指针增1或front指针增1 时超出了所分配的队列空间，就让它指向这片连续空间的起始位置。自己真从MaxSize-1增1变到0，可用取余运算rear%MaxSize和front%MaxSize来实现。这实际上是把队列空间想象成一个环形空间，环形空间中的存储单元循环使用，用这种方法管理的队列也就称为循环队列。除了一些简单应用之外，真正实用的队列是循环队列。

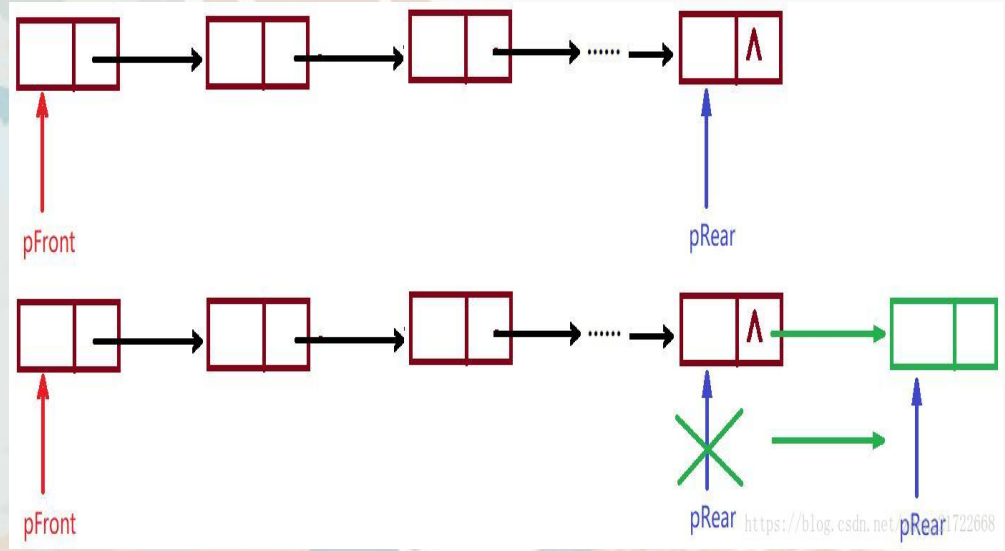
2.在循环队列中，当队列为空时，有front=rear，而当所有队列空间全占满时，也有front=rear。为了区别这两种情况，规定循环队列最多只能有MaxSize-1个队列元素，当循环队列中只剩下一个空存储单元时，队列就已经满了。因此，队列判空的条件时front=rear，而队列判满的条件时front=（rear+1）%MaxSize。队空和队满的情况如图：



2.1、关键基本操作的具体实现概述

1、初始化队列：

```
//让队列的队头，队尾分别指向空

void LQueueInit(LQueue *q)
{
        assert(q);
        q->pFront = q->pRear =
NULL;
}
```



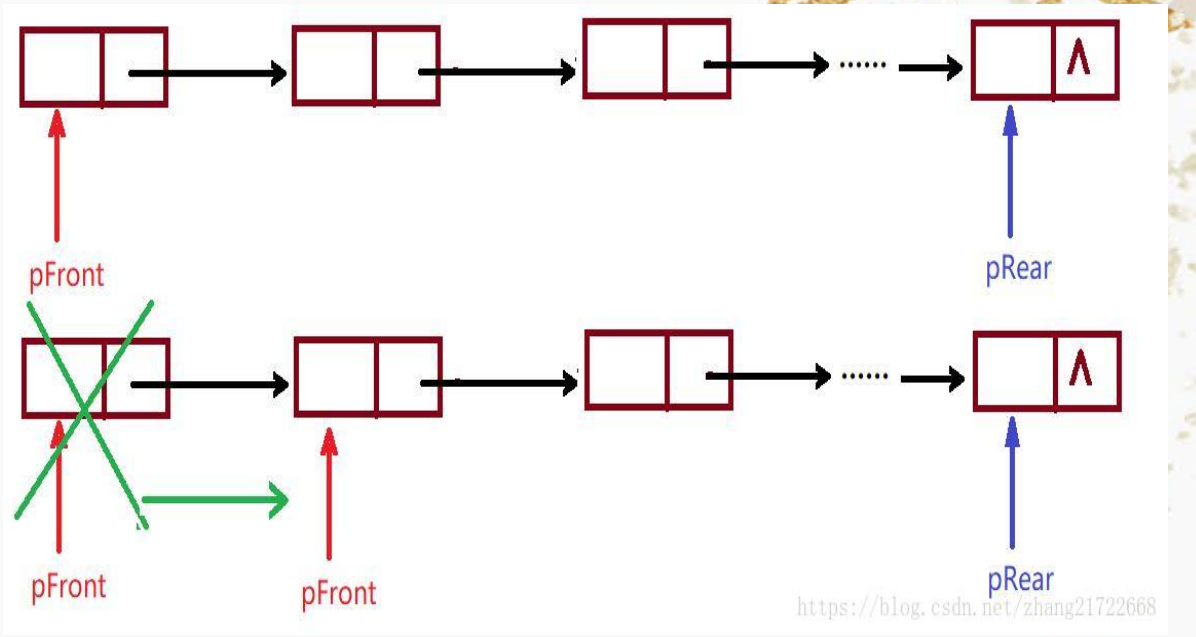
2、元素入队：

```
//判断队中是否有元素
//找到队尾元素
//让新入队的元素链在原先队列的队尾上，更新新的队尾

void LQueuePush(LQueue *q, QElemType data)
{
        assert(q);
        if (NULL == q->pFront)
        {
                q->pFront = q->pRear = BuyLQNode(data);
                return;
        }
        q->pRear->_pNext = BuyLQNode(data);
        q->pRear = q->pRear->_pNext;}
```

2.2、关键基本操作的具体实现概述

3、元素出队：

```
//这里的出队列采用是让队头元素不断后移，刷新队头元素，这样优化时间
效率

void LQueuePop(LQueue *q)
{
		assert(q);
		QNode *pDel;
		if (NULL == q->pFront)
		{
			return;
		}
		if (q->pFront == q->pRear)
		{
			q->pRear = NULL;
		}
		pDel = q->pFront;
		q->pFront = q->pFront->_pNext;
		free(pDel);
}
```



2.3、关键基本操作的具体实现概述

4、返回队头元素：

```
//直接返回队头元素

QElemType LQueueTop(LQueue *q)
{
        assert(q);
        return q->pFront->data;
}
```

5、返回队尾元素：

```
//直接返回队尾元素

QElemType LQueueBack(LQueue *q)
{
        assert(q);
        return q->pRear->data;
}
```

6、计算队列长度：

```
//定义一个变量count
//将队列从头到尾遍历，每访问一个元素count加1一下
//最后count的值就是队列长度
int LQueueSize(LQueue *q)
{       int count = 0;
        QNode *pCur;
        assert(q);
        pCur = q->pFront;
        while (pCur)
        {       pCur = pCur->_pNext;
                count++;}
        return count;}
```

7、队列判空操作：

```
int LQueueEmpty(LQueue *q)
{
        return NULL == q->pFront;
}
```

3.1、应用

字符串中第一个唯一字符：

给定一个字符串 s，找到 它的第一个不重复的字符，并返回它的索引。如果不存在，则返回 -1。

示例 1：
输入: s = "leetcode"
输出: 0

示例 2:
输入: s = "loveleetcode"
输出: 2

示例 3:
输入: s = "aabb"
输出: -1

3.2、应用

我们可以借助队列找到第一个不重复的字符。队列具有「先进先出」的性质，因此很适合用来找出第一个满足某个条件的元素。

具体地，我们使用与方法二相同的哈希映射，并且使用一个额外的队列，按照顺序存储每一个字符以及它们第一次出现的位置。当我们对字符串进行遍历时，设当前遍历到的字符为 cc，如果 cc 不在哈希映射中，我们就将 cc 与它的索引作为一个二元组放入队尾，否则我们就需要检查队列中的元素是否都满足「只出现一次」的要求，即我们不断地根据哈希映射中存储的值（是否为 -1−1）选择弹出队首的元素，直到队首元素「真的」只出现了一次或者队列为空。

在遍历完成后，如果队列为空，说明没有不重复的字符，返回 -1，否则队首的元素即为第一个不重复的字符以及其索引的二元组。

3.2、应用

```
class Solution {
public:
    int firstUniqChar(string s) {
        unordered_map<char, int> position;//建立哈希映射，第一个数据类型为char，第二个为int
        queue<pair<char, int>> q;//把字符及其下标作为一个二元组放入队列
        int n = s.size();
        for (int i = 0; i < n; ++i) {
            if (!position.count(s[i])) {函数count（），判断字符出现次数，字符已经出现过返回1，没出现过返回0
                position[s[i]] = i;//获得字符对应的下标
                q.emplace(s[i], i);//函数emplace（），把字符和下标作为二元组存入队列
            }
            else {
                position[s[i]] = -1;//字符出现两次
                while (!q.empty() && position[q.front().first] == -1) {
                    q.pop();//弹出队首元素
                }
            }
        }
        return q.empty() ? -1 : q.front().second;//队列中字符全部重复，全部弹出，否则返回字符串中第一个唯一字符下标
    }
};
```

第二次讨论课

STL序列式容器探讨
向量（vector）

Discussion on STL sequential container

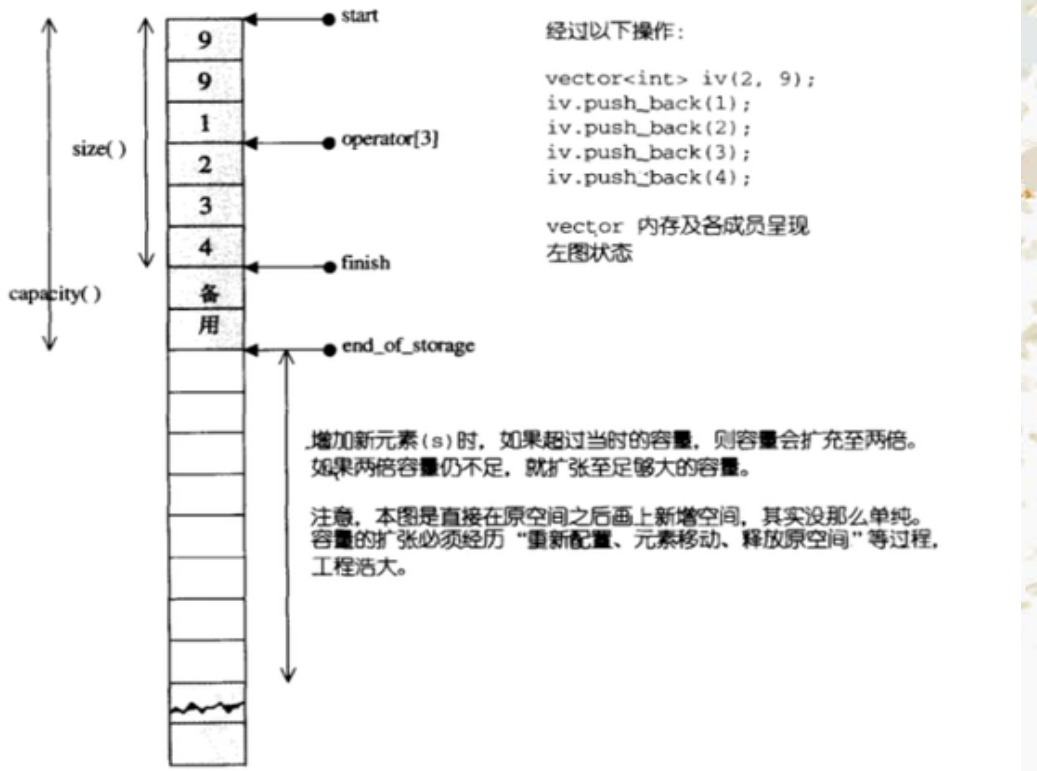
目录

1.1、概述：

向量（vector）是STL提供的最简单，也是最常用的容器类模板之一。它与数组之间的相似性在于提供了对序列中的元素进行随机访问。但与传统的数组不同之处在于，vector对象在运行时可以动态改变自身的大小以便容纳任何数目的元素。

它提供了对元素的快速，随机访问，以及在序列尾部快速的插入和删除操作。当然，它也支持在序列中的其他地方插入和删除元素，但这时效率会有所降低，因为vector对象必须移动对象位置以容纳新的元素或收回被删除元素的空间。

简单来说Vector是一个称为向量的顺序容器。



1.2、概述：

vector特点：

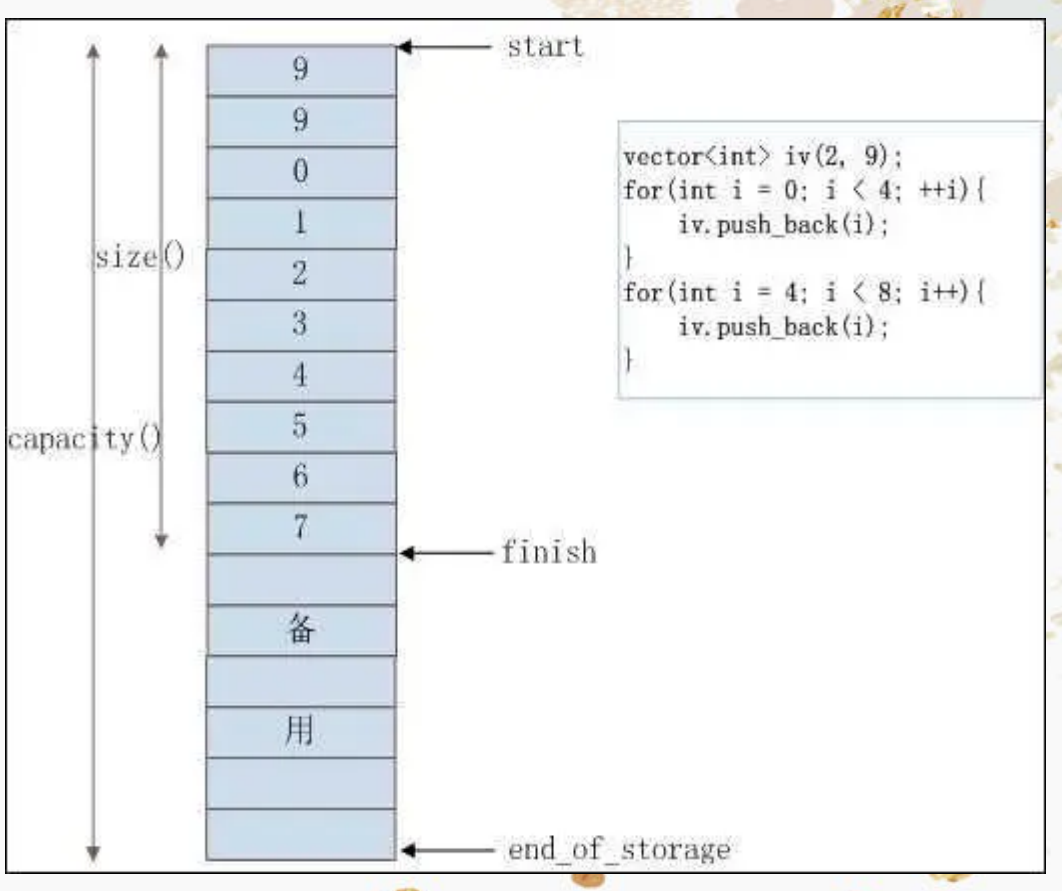
1. 动态（相当于一个动态数组）；

2. 可用于存放各种数据结构（内置的，自定义的，当然也可以是vector）（当是vector时注意申明格式eg：vector<vector<Information>>与vector<vector<Information> >，前者错误，后者正确，两个‘>’要有空格）；

3. 支持对vector元素的随即访问；

4. 以拷贝的方式用一个vector初始化另一个vector时，两个vector类型要一致；

5. 每一个vector的容量要比其大小相对较大或相等（eg：vec.capacity()>=vec.size()）。



2.1、关键基本操作的具体实现概述

1、定义和初始化vector对象
vector<T> v1;//v1是一个空vector，它潜在的元素是T类型的，执行默认初始化
vector<T> v2(v1);//v2中包含v1所有元素的副本
vector<T> v2=v1;//等价于v2(v1),v2中包含有v1所有元素的副本
vector<T> v3(n,val);//v3中包含着n个重复的元素
vector<T> v4(n);//v4包含了n个重复执行值初始化的对象
vector<T> v5{a,b,c...};//v5包含了初始值个数的元素，每个元素被赋予相应的初始值
vector<T> v5={a,b,c};//等价于上一个

2、尾部插入数字：vec.push_back(a);

3、使用下标访问元素，cout<<vec[0]<<endl;记住下标是从0开始的。

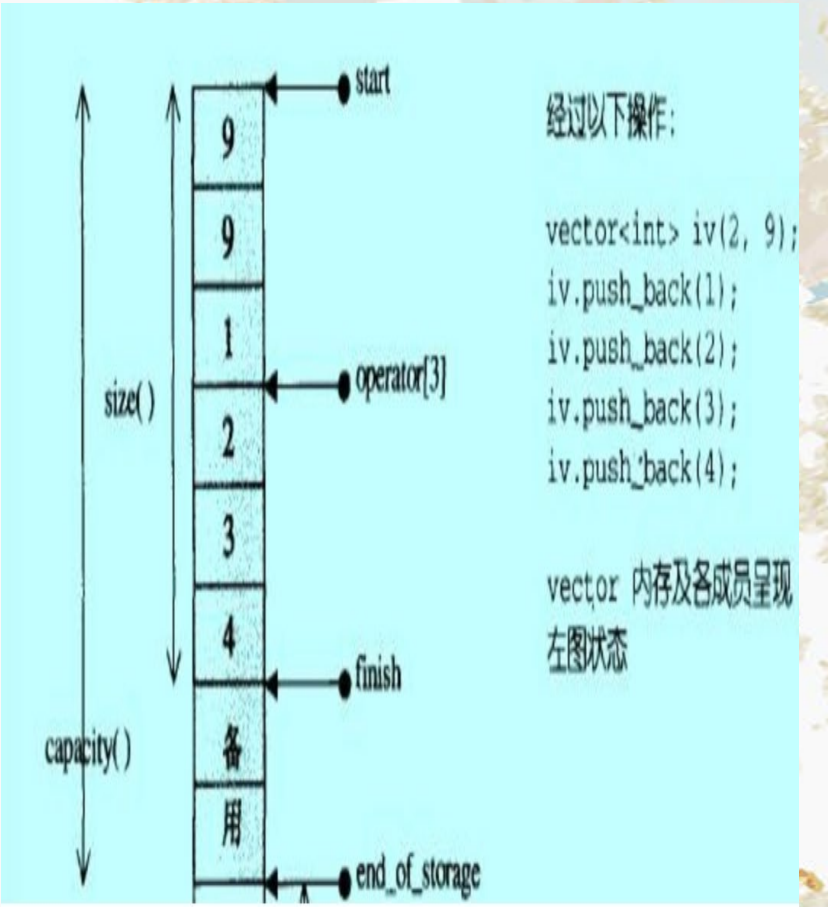
4、使用迭代器访问元素.

5、插入元素：　vec.insert(vec.begin()+i,a);在第i+1个元素前面插入a;

6、删除元素：　vec.erase(vec.begin()+2);删除第3个元素
vec.erase(vec.begin()+i,vec.end()+j);删除区间[i,j-1];区间从0开始

7、向量大小:vec.size();

8、清空:vec.clear();



2.2、关键基本操作的具体实现概述

vector中的常用函数接口：

Vec.begin();//向量中的首个元素的地址，加n后表示第n个元素的地址（eg：vec.begin()+3）；
Vec.end();//向量中最后一个元素的地址；
Vec.rbegin();//指向反序后的向量首元素；
Vec.rend();//指向反序后的尾元素；
vec.front();//返回vector中的第一元素的值
vec.back();//返回vector中的最后一个元素的值
Vec.insert();//向向量中任意位置插入一个或多个元素；
Vec.clear();//向量清空；
Vec.push_back();//将元素压入向量的尾部；
//Vec.push_front();//vector中无此用法（用了之后会造成元素的迁移，与vector设计的初衷相违背）；
Vec.erase();//删除向量中任意一个或一段元素；
Vec.pop_back();//删除向量中最后一个元素；
Vec.size();//返回向量的大小
Vec.capacity();//返回向量容量的大小
Vec.empty();//向量为空返回false，不为空返回true；
Vec1.swap(vec2);//交换向量vec1和向量vec2；

代码实现

```
template <class T, class Alloc = alloc>
void vector<T, Alloc>::insert_aux(iterator position, const T& x){
    //还有备用空间
    if (finish != end_of_storage){
        //在备用空间起始处构造一个元素，并以vector最后一个值为其初值
        construct(finish, *(finish - 1));
        //调整水位
        ++finish;
        //拷贝一个元素
        T copy_x = x;
        //把vector插入位置position之后的元素往后移一位
        copy_backward(position, finish - 2, finish - 1);
        //给position指向的地方赋值，赋值内容为前面拷贝的元素
        *position = copy_x;
    }
    //已无备用空间
    else
    {
        const size_type old_size = size();
        //如果原来的vector为空，则申请一个元素的空间，否则申请可以容纳原来2倍元素的空间
        const size_type len = old_size == 0 ? 1 : 2 * old_size;
        //全局函数，申请空间
        iterator new_start = data_allocator::allocate(len);
        iterator new_finish = new_start;
try
        {
            //将原来vector的position之前的内容拷贝到新的vector前面
            new_finish = uninitialized_copy(start, position, new_start);
            //调用构造函数为新插入的元素赋值
            construct(new_finish, x);
            //调整水位
            ++new_finish;
            //将原来vector的postion之后的内容拷贝到新的vector后面
            new_finish = uninitialized_copy(position, finish, new_finish);
        }
        catch (...){
            //析构
            destroy(new_start, new_finish);
            //释放刚刚申请的内存
            data_allocator::deallocate(new_start, len);
            throw;
        }
        //析构原vecotr
        destroy(begin(), end());
        //释放原vector的空间
        deallocate();

        //调整迭代器指向新的vector
        start = new_start;
        finish = new_finish;
        end_of_storage = new_start +len;
    }
}
```

第二次讨论课

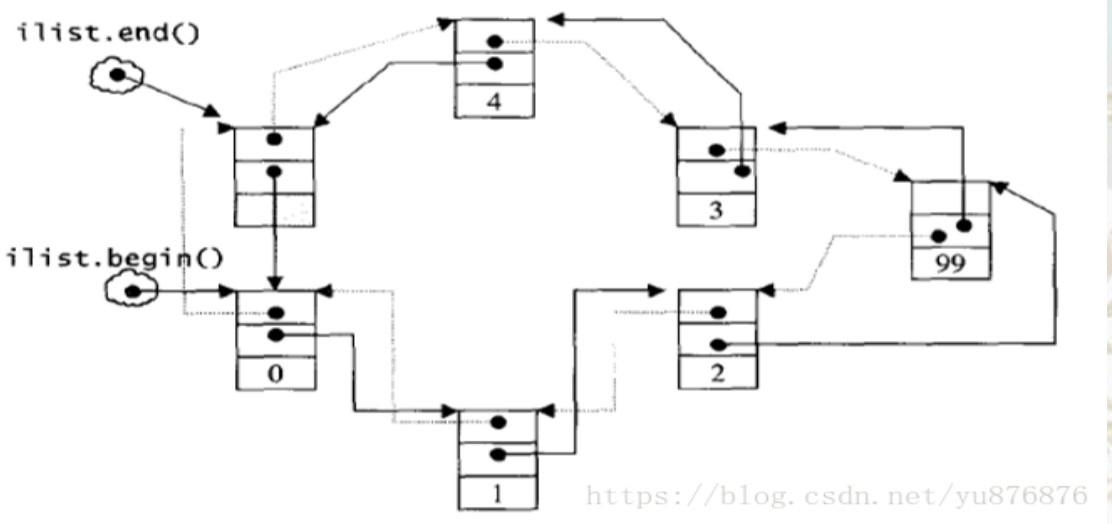
STL序列式容器探讨
列表（list）

Discussion on STL sequential container

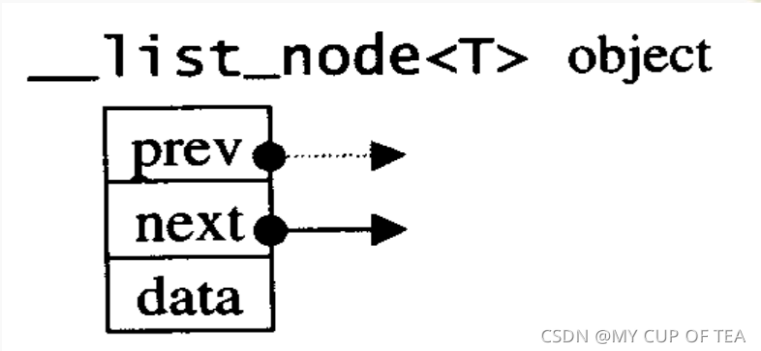
目录

1.1、概述：

列表（list）容器，又称双向链表容器，即该容器的底层是以双向链表的形式实现的。这意味着，list 容器中的元素可以分散存储在内存空间里，而不是必须存储在一整块连续的内存空间中。跟任意其它类型容器一样，它能够存放各种类型的对象。



list相较于vector连续的线性空间就显得很复杂，他的存储空间是不连续的，好处是每次插入和删除一个元素的时候，只需要配置或者释放一个元素的空间，使得插入和删除十分的方便，时间复杂度为O（1）。



list 本身和list的节点是不同的结构，需要分开设计list节点结构是一个双向的链表。

1.2、概述：

list迭代器不能像vector一样使用普通的指针，因为考虑到内存分配的不连续性，list的迭代器必须有能力指向list的的节点，并正确的进行递增、递减、取值（获取成员节点的数据值）、成员的存取（取用的是节点的成员）等操作。

因为list的迭代器具备前移和后移的能力，所以使用Bidirectional iterators。

list的插入和接合操作(splice)都不会造成原有的list迭代器失效，vector不可以，因为vector插入操作可能造成记忆体的重新配置，会导致原有的迭代器失效。

list删除元素(erase)也只有指向被删除元素的那个迭代器会失效，其余的迭代器不会受到任何的影响。

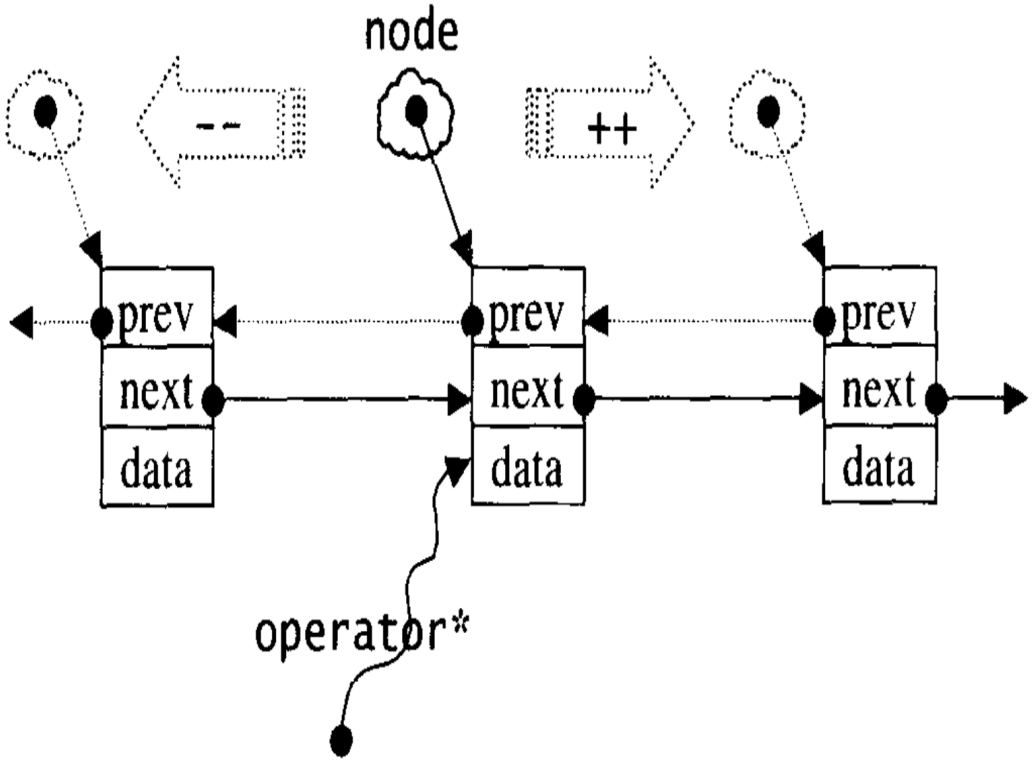


图 4-4 list 的节点与 list 的迭代器

2.1、关键基本操作的具体实现概述

1、list的构造函数

list<int>a{1,2,3}
list<int>a(n) //声明一个n个元素的列表，每个元素都是0
list<int>a(n, m) //声明一个n个元素的列表，每个元素都是m
list<int>a(first, last) //声明一个列表，其元素的初始值来源于由区间所指定的序列中的元素，first和last是迭代器

2、begin()和end()
通过调用list容器的成员函数begin()得到一个指向容器起始位置的iterator，可以调用list容器的end()函数来得到list末端下一位置。

3、push_back()和push_front()
使用list的成员函数push_back和push_front插入一个元素到list中。其中push_back()是从list的末端插入，而push_front()是从list的头部插入。

4、empty()
判断list是否为空。

5、clear()
清空list中的所有元素

6、front()和back()
通过front()可以获得list容器中的头部元素，通过back()可以获得list容器的最后一个元素。注意：当list元素为空时，这时候调用front()和back()不会报错。因此在编写程序时，最好先调用empty()函数判断list是否为空，再调用front()和back()函数。

2.2、关键基本操作的具体实现概述

7、resize()

调用resize(n)将list的长度改为只容纳n个元素，超出的元素将被删除。如果n比list原来的长度长，那么默认超出的部分元素置为0。也可以用resize(n, m)的方式将超出的部分赋值为m。

```
list<int>b{1, 2, 3, 4};
b.resize(2);
list中输出元素：1,2
list<int>b{1, 2, 3, 4};
b.resize(6);
list中输出元素：1,2,3,4,0,0
list<int>b{1, 2, 3, 4};
b.resize(6,9);
list中输出元素：1,2,3,4,9,9
```

8、pop_back()和pop_front()

使用pop_back()可以删掉尾部第一个元素，pop_front()可以删掉头部第一个元素。注意：list必须不为空，如果当list为空的时候调用pop_back()和pop_front()会使程序崩掉。

9、assign()

有两种情况

（1）a.assign(n, val):将a中的所有元素替换成n个val元素

例如：

```
list<int>b{1,2,3,4,5};
b.assign(5,10);
```

b中的元素变为10, 10, 10, 10, 10

（2）a.assign(b.begin(), b.end())

```
list<int>a{6,7,8,9};
list<int>b{1,2,3,4,5};
b.assign(a.begin(),a.end());
```

b中的元素变为6,7,8,9

第二次讨论课

STL序列式容器探讨
双端队列（deque）

Discussion on STL sequential container

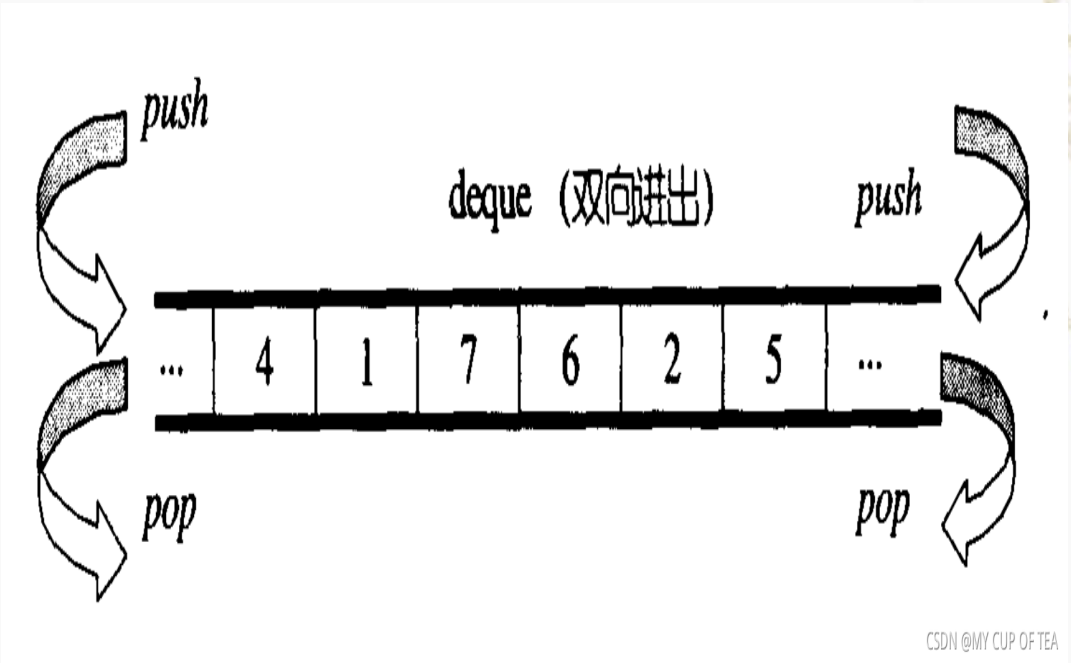
目录

1.1、概述：

deque容器为一个给定类型的元素进行线性处理，像向量一样，它能够快速地随机访问任一个元素，并且能够高效地插入和删除容器的尾部元素。但它又与vector不同，deque支持高效插入和删除容器的头部元素，因此也叫做双端队列。

deque的特点：
支持随机访问，即支持[]以及at(),但是性能没有vector好
可以在内部进行插入和删除操作，但性能不及list。

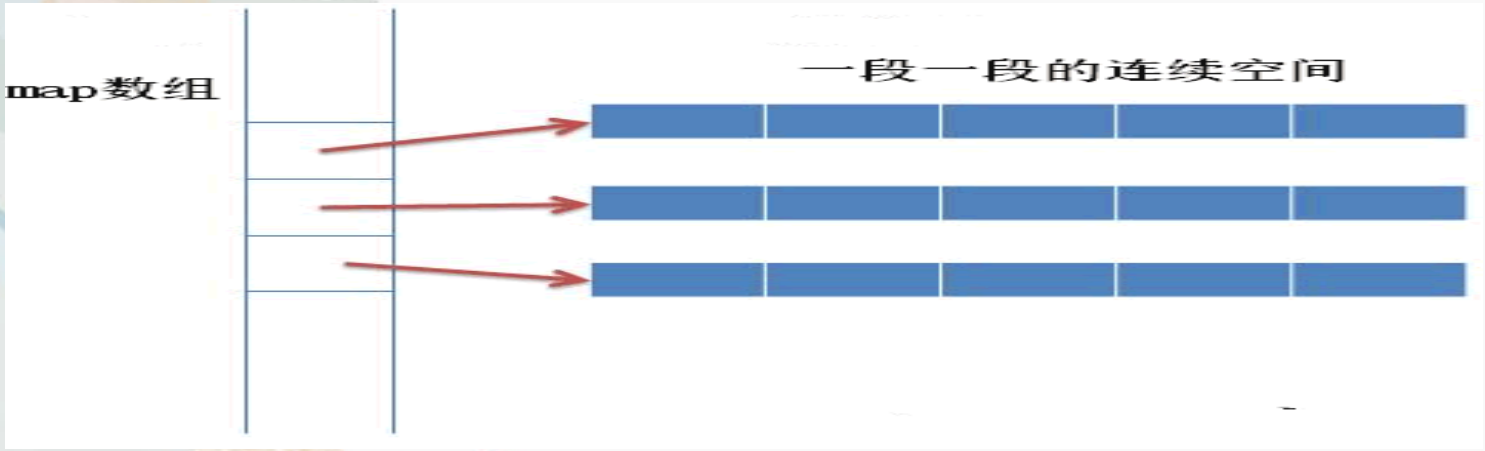
deque和vector的不同之处：
deque两端都能够快速的插入和删除元素。vector只能在尾端进行。
deque的元素存取和迭代器操作会稍微慢一些。因为deque的内部结构会多一个简介过程。
迭代器是特殊的智能指针，而不是一般指针。它需要在不同的区块之间跳转。
deque可以包含更多的元素，其max_size可能更大。因为不止使用一块内存。
不支持对容量和内存分配时机的控制。



2、deque的定义

和 vector 容器采用连续的线性空间不同，deque 容器存储数据的空间是由一段一段等长的连续空间构成，各段空间之间并不一定是连续的，可以位于在内存的不同区域。

为了管理这些连续空间，deque 容器用数组（数组名假设为 map）存储着各个连续空间的首地址。也就是说，map 数组中存储的都是指针，指向那些真正用来存储数据的各个连续空间（如下图所示）。



3.1、关键基本操作的具体实现概述

1插入操作：

deque<string> dec;

void push_front(const T& x): 双端队列头部增加一个元素X

dec.push_front("world"); //头部插入

void push_back(const T& x):双端队列尾部增加一个元素x

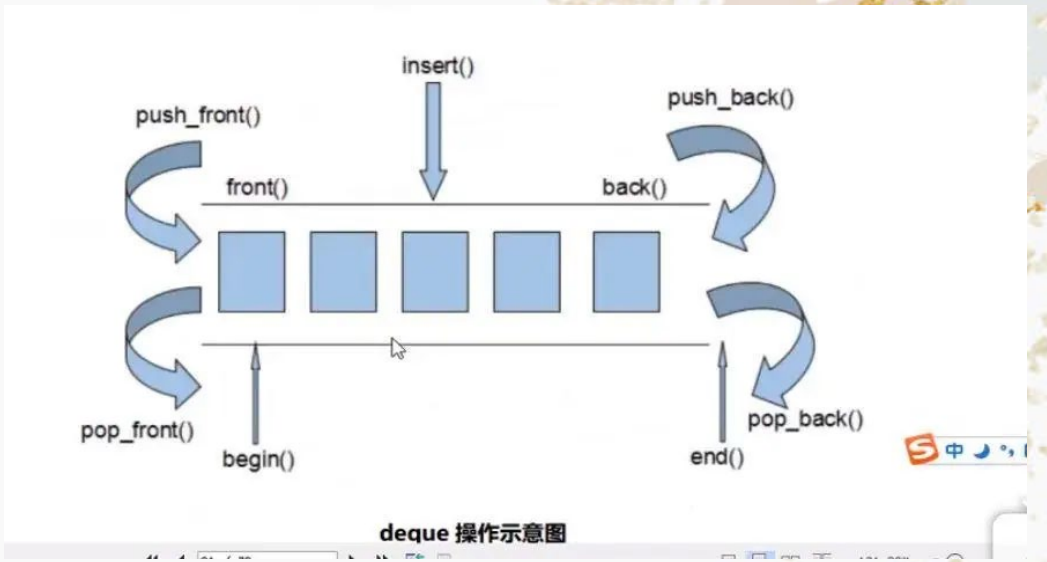
dec.push_back("hello"); //尾部插入

iterator insert(iterator it,const T& x):双端队列中某一元素前增加一个元素x

dec.insert(dec.end(), "aaaaaa");

void insert(iterator it,int n,const T& x):双端队列中某一元素前增加n个相同的元素x

dec.insert(dec.end(), 5, "bbb");



deque 操作示意图

void insert(iterator it,const_iterator first,const_iteratorlast):双端队列中某一元素前插入另一个相同类型向量的[forst,last)间的数据

deque<string> t_dec(2,"ccc");

dec.insert(dec.end(), t_dec.begin(), t_dec.end());

3.2、关键基本操作的具体实现概述

2、删除操作：

iterator erase(iterator it):删除双端队列中的某一个元素
dec.erase(dec.begin());

iterator erase(iterator first,iterator last):删除双端队列中[first,last)中的元素
dec.erase(dec.end()-3, dec.end());

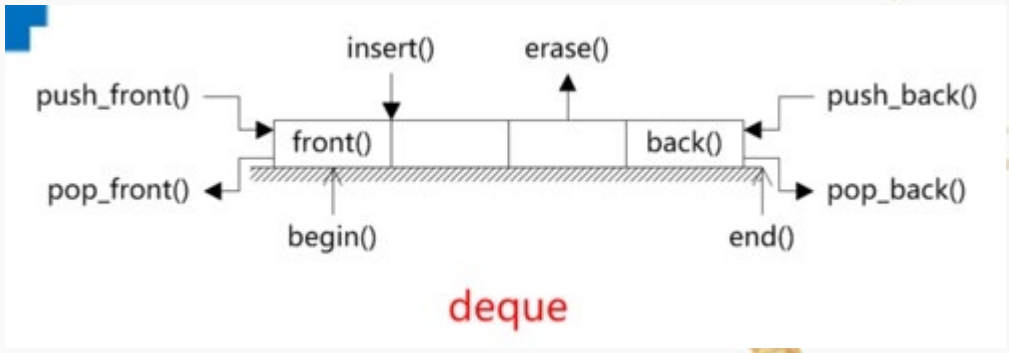
void pop_front():删除双端队列中最前一个元素
dec.pop_front(); //头部删除

void pop_back():删除双端队列中最后一个元素
dec.pop_back(); //尾部删除

void clear():清空双端队列
dec.clear(); //清空

删除指定元素：

```
//删除指定元素
deque<string>::iterator iter;
for(iter = dec.begin(); iter != dec.end(); iter++){
   if(*iter == "aaa") {
      iter = dec.erase(iter);
   }
}
```



4、Deque的应用

我们充分发挥deque两端进出的功能来解决这个问题

//描述

//给定一个长度为 n 的数组 num 和滑动窗口的大小 size ，找出所有滑动窗口里数值的最大值。

//

//例如，如果输入数组{2,3,4,2,6,2,5,1}及滑动窗口的大小3，那么一共存在6个滑动窗口，他们的最大值分别为{4,4,6,6,6,5}； 针对数组{2,3,4,2,6,2,5,1}的滑动窗口有以下6个： {[2,3,4],2,6,2,5,1}， {2,[3,4,2],6,2,5,1}， {2,3,[4,2,6],2,5,1}， {2,3,4,[2,6,2],5,1}， {2,3,4,2,[6,2,5],1}， {2,3,4,2,6,[2,5,1]}。

//

//窗口大于数组长度或窗口长度为0的时候，返回空。

//

//数据范围： 0≤n≤10000，0 ≤size≤10000，数组中每个元素的值满足 0≤ value ≤ 10000

思路：

把deque当作这个滑动窗口。我们要做的就是维护每个滑动窗口的最大值

现在有这几种情况

1，未形成窗口开始加入deque的数小于窗口长度，这时一直添加即可

2，位于deque前面的数比后面的数先进入队列，根据这个数现在的位置判断它是否出窗口，若不再窗口范围内，弹出。

3，添加元素，每次移动窗口必有元素淘汰，也必有新元素进入，2我们讨论了离开元素，这是要维护窗口最大值，我们要用新元素不断更新deque，如deque中为3，4，5，新元素为6，新元素比他们都大，应该将他们全部更新，毕竟新元素待在窗口中的时间也比他们长

一些处理

Deque中的元素可能会被比他大的元素淘汰，也可能会因为窗口离开而淘汰，这时我们向deque中添加的应该是元素下标

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

vector<int> maxInWindows(const vector<int>& num, unsigned int size)
    {
        vector<int> ret;
        int n = num.size();
           deque<int> dq;
           for (int i = 0; i < n; ++i) {
               while (!dq.empty() && num[dq.back()] < num[i]) {
                   dq.pop_back();
               }
               dq.push_back(i);
               // 判断队列的头部的下标是否过期
               if (dq.front() + size <= i) {
                   dq.pop_front();
               }
               // 判断是否形成了窗口
               if (i + 1 >= size) {
                   ret.push_back(num[dq.front()]);
               }
           }
           return ret;
    }
int main(){
    unsigned int size;int n;
    cin>>n>>size;
    vector<int>v;
    for(int i=0;i<n;i++){
        int m;
        cin>>m;
        v.push_back(m);
    }
    vector<int>res(n);
    res=maxInWindows(v,size);
    for(int i=0;i<res.size();i++){
        cout<<res[i]<<" ";
    }
    return 0;
```

AC代码

第二次讨论课

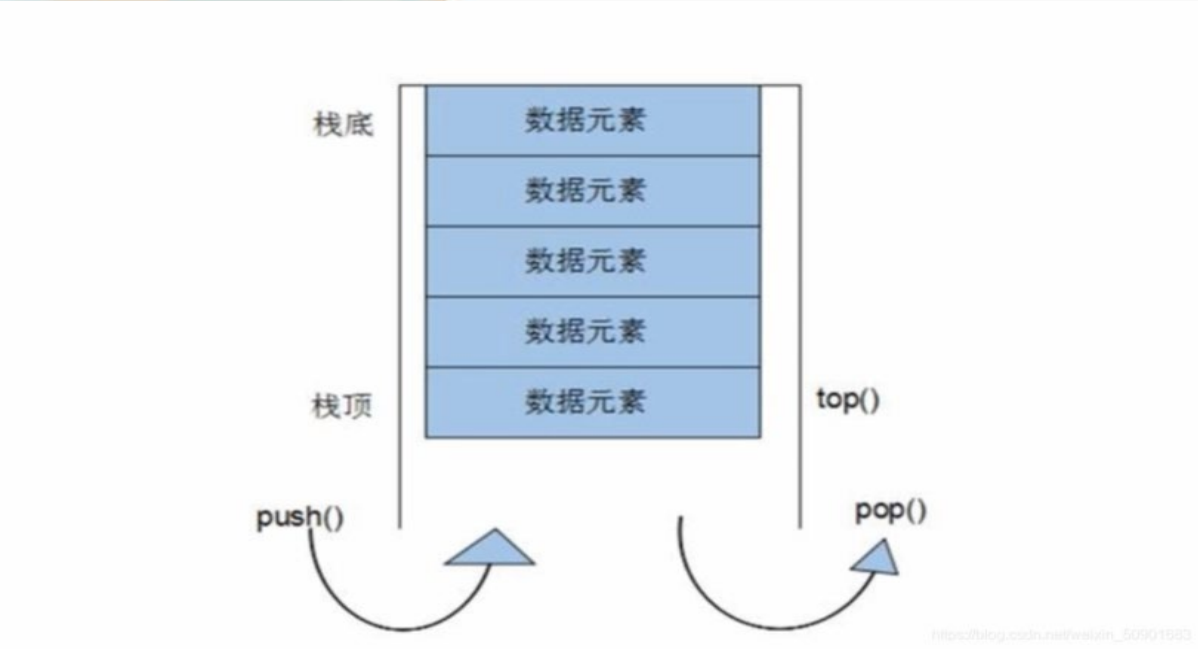
STL序列式容器探讨
栈（stack）

Discussion on STL sequential container

目录

1、概述：

栈（stack）是一种后进先出的数据结构，即LIFO（last in first out），最后加入栈的元素将最先被取出来，在栈的顶端进行数据的插入与取出，这一端称为栈顶，相对的，把另一端称为栈底。
向一个栈插入新元素称为进栈，入栈或压栈。从一个栈删除元素又称为出栈或退栈。



栈是一种逻辑数据结构，其具有后进先出的特性。同时，我们也可以把它想象成一个容器，一个真实容器，添加与删除只能在容器顶部完成。栈的应用非常广，我们知道任何程序从内存进入CPU执行。

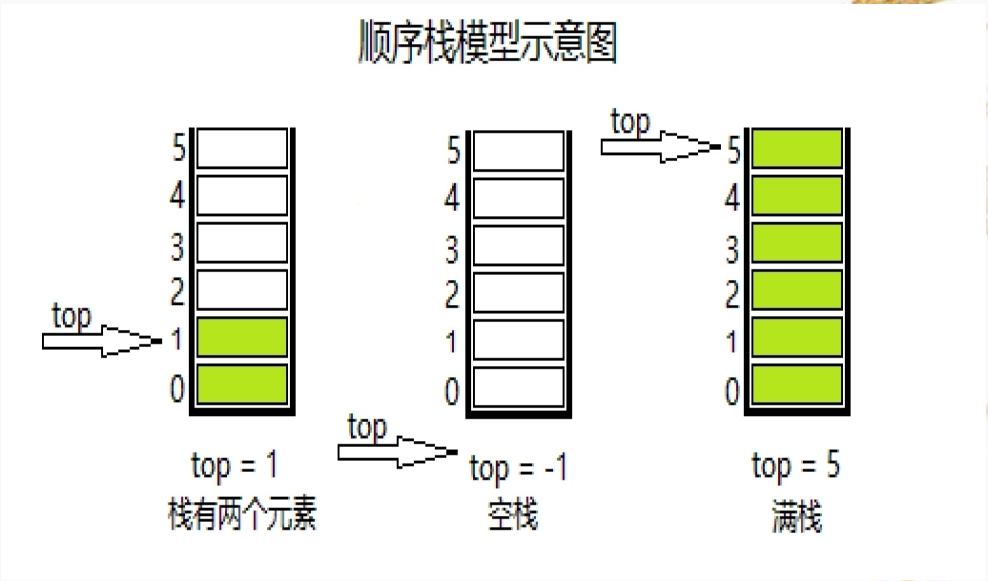
系统为了保证程序正确的执行，将程序二进制代码存放在一个系统运行栈上面。调用函数A，则A的代码入栈，函数A中调用了函数B，则B入栈，B执行结束后，先出栈，这时再继续执行函数A。因此这种后进先出的工作方式在很多地方都非常有用。

2、定义：

1、栈(stack)是仅限定在表尾进行插入和删除操作的线性表。

栈就是一个线性表，只不过，栈的Insert 和 delete只能在表尾。普通的线性表，在表中的任意位置都可以进行insert和delete操作。
LIFO: Last In First Out 后进先出，先进后出。栈顶(Top): 进行插入和删除操作的一端。栈(Bottom)

栈其实我们计算机科学中，更多的一种思想，“先进后出的思想”。在很多算法或应用中，需要用到“先进后出的思想”，我们可以考虑用栈来实现。



3、存储结构：

顺序结构： 用一组地址连续的空间来存储数据元素。

链式结构：用地址不连续的空间来存储数据元素，可能需要额外开辟一些空间，来存储“数据元素之间的逻辑关系"。

3.1、关键基本操作的具体实现概述

1、初始化一个栈：InitStack

```
//初始化一个栈
// @maxl: 代表制定这个栈的最大可以存储多少个元素
//返回值： 返回初始化好的顺序栈指针
SeqStack * InitStack(int maxl)
{
        SeqStack *s = malloc(sizeof(*s));
        s->data = malloc(sizeof(SElemType)* maxl);
        s->top = -1;
        s->maxlen = maxl;
        return s;
}
```

2、销毁一个栈：DestroyStack

```
//销毁一个栈
void DestoryStack(SeqStack*s)
{
        if(s==NULL)
                return ;
        free(s->data);
        free(s);
}
```

3、清空一个栈：ClearStack

```
//清空一个栈
void ClearStack(SeqStack*s)
{
        if(s==NULL)
                return ;
        s->top=-1;
}
```

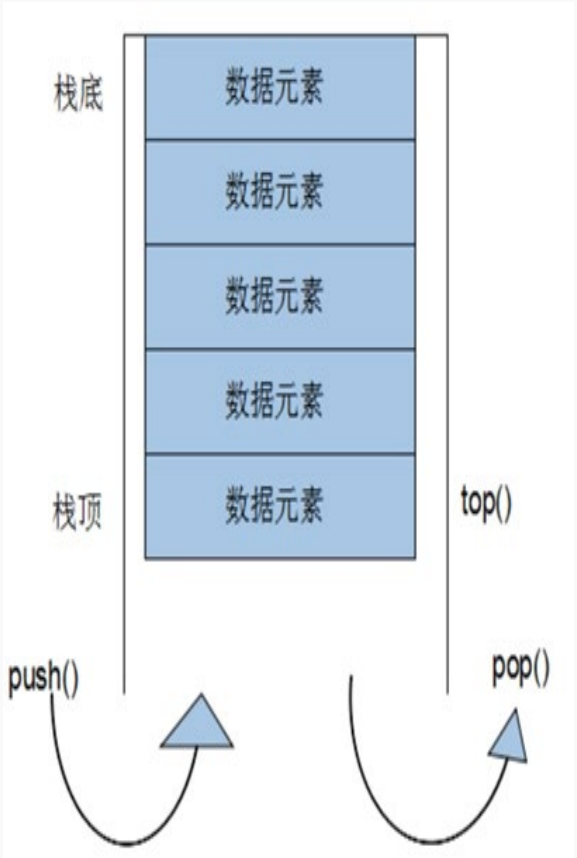
4、判断一个栈是否为空

```
//1 表示为空栈或不存在
//0 表示非空栈
int StackisEmpty(SeqStack*s)
{
        if(s==NULL ||s->top ==-1 )
        {
                return 1;
        }
        return 0;
}
```

3.2、关键基本操作的具体实现概述

5、入栈，把一个元素加入到栈中Push：

```
//入栈操作
//s：栈指针，表示那个栈
//x: 要入栈的元素
//返回值 表示入栈是否成功，1 表成功 0 表失败
int Push(SeqStack *s ,SElemType x)
{
        //最大元素个数为1 s->top=0
        if(s==NULL ||s->top == s->maxlen -1)
        {
                return 0;
        }
        s->data[++s->top]=x;

        if(s->top==s->maxlen -1)
                return 0;
        else
                return 1;

}
```



6、出栈，把栈顶元素给弹出：Pop

```
int Push(SeqStack *s ,SElemType x)
{
        //最大元素个数为1 s->top=0
        if(s==NULL ||s->top == s->maxlen -
1)
        {
                return 0;
        }
        s->data[++s->top]=x;

        if(s->top==s->maxlen -1)
                return 0;
        else
                return 1;
}
```

3.3、关键基本操作的具体实现概述

7、返回栈中元素个数，即栈的长度StackLength：

```
//返回栈中的元素个数咯
int Stacklength(SeqStack *s)
{
        if(s==NULL)
        {
                return 0;
        }
        return s->top +1;
}
```

8、返回栈顶元素，但不出栈：GetTop

```
//获取栈顶元素的但是不出栈
int GetTop(SeqStack *s,SElemType *e)
{
        if(s==NULL || s->top == -1)
        {
                return 0;
        }
        *e = s->data[s->top];
        return 1;
}
```

4、应用

问题描述：
给定一个二维数组，0表示可以通行，1表示该处有墙不能通行，试问给定两点坐标，判断这两个点能不能相通。

问题很短，那我们用栈来解决吧

思路

我们已知一个点的坐标，将它相邻的某一个点压入栈中，表明我们已经行走到这个点了，这是将上一个点坐标标记起来，表明我们已经走过。

情况1：直接走到目标点(不用多说了吧)

情况2：遇到这样一个点which它周围的点都是1，也就是说走到死路呐，咋办？当我们遇到情况二时我们运用栈的特性，将栈中元素弹出(也就是上一个点)这是我们知到上一个点的信息，再次寻找上一个点周围是否还有道路，如果有，则顺着这条路前进。

情况3：如果一直遇到情况2，栈中元素弹完啦还是没有找到目标点咋办，这就说明我们已经尝试过所有道路了，这个点根本不能到达目标点 return false;

```
    int** map;//迷宫
    int row;//迷宫行列(含边界)
    int col;
    pos now;//当坐标
    int opt;//对位移偏量的选择
    stack<pos> path;
    int endrow;
    int endcol;
};
```

```
void findPath()
{
    int xbegin ,ybegin;
    cout<<"输入起点的行和列："<<endl;
    cin>>xbegin>>ybegin;
    cout<<"输入终点的行和列："<<endl;

    cin>>endrow>>endcol;
    endrow--;
    endcol--;
    //位移偏量
    pos setMove[4];
    setMove[0].posRow=0;  setMove[0].posCol=1; //右移
    setMove[1].posRow=1;  setMove[1].posCol=0; //下移
    setMove[2].posRow=0;  setMove[2].posCol=-1; //左移
    setMove[3].posRow=-1; setMove[3].posCol=0; //上移


    //边界-1
    now.posRow=xbegin-1;
    now.posCol=ybegin-1;


    map[xbegin-1][ybegin-1]=1;
```

```
while(now.posRow!=endrow||now.posCol!=endcol)
{
    //先找每一步的怎么走
    int Row;
    int Col;
    while(opt<=3)
    {
        //移动
        Row=now.posRow+setMove[opt].posRow;
        Col=now.posCol+setMove[opt].posCol;

        //判断是否撞墙
        if(map[Row][Col]==0)
            break;
        opt++;//撞墙，换下一种方式
    }
    //如果能移动
    if(opt<=3)
    {
        cout<<"move:";
        cout<<Row+1<<","<<Col+1<<"  ";
        //填入栈中
        path.push(now);
        //移动到下一个位置
        now.posRow=Row;
        now.posCol=Col;

        opt=0;
        //并堵住来的路
        map[Row][Col]=1;

    }
```

```
    //如果不能移动
    else
    {
        //向后退
        if(path.empty())
        {
            cout<<"没有出路！"<<endl;
            return ;
        }
        //退回至上一处
        pos next;//下一步的坐标
        next=path.top();
        cout<<"move:";
        cout<<next.posRow+1<<","<<next.posCol+1<<"  ";
        path.pop();
        if(next.posRow==now.posRow)//行坐标相同
        {
            int lastmove=next.posCol-now.posCol;
            //1,上次左移
            //-1,上次右移
            opt=2+lastmove ;

        }
        else
        {
            int lastmove=next.posRow-now.posRow;
            opt=3+lastmove;

        }
        now=next;
    }
}
cout<<"找到出路"<<endl;
```

升级：

刚刚我们用栈模拟了一种像无头苍蝇乱撞的过程，我们发现，最后达到的点如果不行就会最先退出，这好像和递归有点像欸，比如递归找斐波拉契数时，最后递归的是最先得出结果的，那现在用递归进行升级。

```
int dict[4][2]={{1,0},{-1,0},{0,1},{0,-1}};
bool isinarea(int x,int y,vector<vector<int> >& grid){
    return x>=0&&y>=0&&x<grid.size()&&y<grid[0].size();
}
void dfs(int x,int y,vector<vector<int> >& grid){
    if(!isinarea(x, y ,grid))return ;
    if(grid[x][y]==1)return ;
    grid[x][y]=1;
    for(int i=0;i<4;i++){
        int nx=x+dict[i][0];
        int ny=y+dict[i][1];
        dfs(nx,ny,grid);
    }
}
```

嗯？为啥就这么一点
我们在上个代码中是进行手动压栈和弹栈，但是递归帮我们自动实现了这个功能，大概是因为这个原因吧

其实，这种不撞南墙不回头的方法还有一个大名dfs
Depth First Search,在搜索领域有举足轻重的位置，但是看起来稍稍有点暴力

汇报完毕谢谢观看

The user can demonstrate on a projector or computer, or print the presentation and make it into a film to be used in a wider field.

汇报人：稻壳儿