2.3 向量-无序向量

cpp实现链接	http://dsa.cs.tsinghua.edu.cn/~deng/ds/src_link/vector/vector_bracket.h.htm http://dsa.cs.tsinghua.edu.cn/~den
备注	代码理解！
完成日	@2020/03/13
视频起始	https://www.bilibili.com/video/av82410486?p=62
讲义	02.Vector.C.unsorted_Vector.pdf

▼ 02-C-1 无序向量

前两章节，构建出了模板：

```
template <typename T> class Vector {} //Vector模板类
Vector <int/float/...> myVector;
Vector <BinTree>; // 以二叉树构成forest
```

对于难以排序的一些元素，进行构造和访问

▼ 02-C-2 循秩访问

- `V.get(r)`和`V.put(r,e)`进行读写，但是便捷性不如A[r]，考虑沿用下标的访问方式
- 重载下标操作符[]

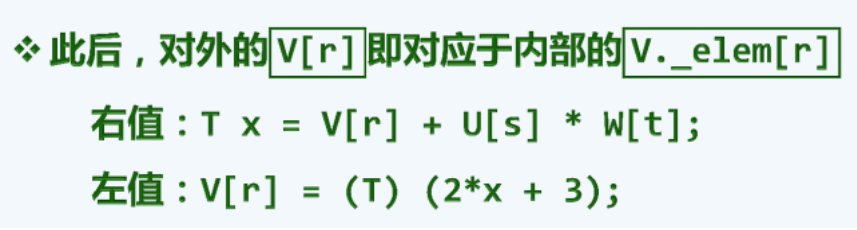
```
/* vector_bracket.h */
template <typename T> T & Vector<T>::operator[] ( Rank r ) //重载下标操作符
{ return _elem[r]; } // assert: 0 <= r < _size

template <typename T> const T & Vector<T>::operator[] ( Rank r ) const //仅限于做右值
{ return _elem[r]; } // assert: 0 <= r < _size
```

当使用`A[r]`，则相当于在访问`A._elem[r]`

  - 可以作为右值，即调用用于计算
  - 也可以作为左值，即可以将结果代入 ⇒实现原理来自采用接口为**引用**的定义 `&`

对于访问较为重要的为**秩 ⇒ 循秩访问**



- 讲解时，则采用更为简易的方式处理意外和错误（如，入口参数越界等），因此当前考虑在输入前已经进行断言，实际应该进行更为实际的处理

```
// 0 <= r<_size
```

▼ 02-C-3 插入

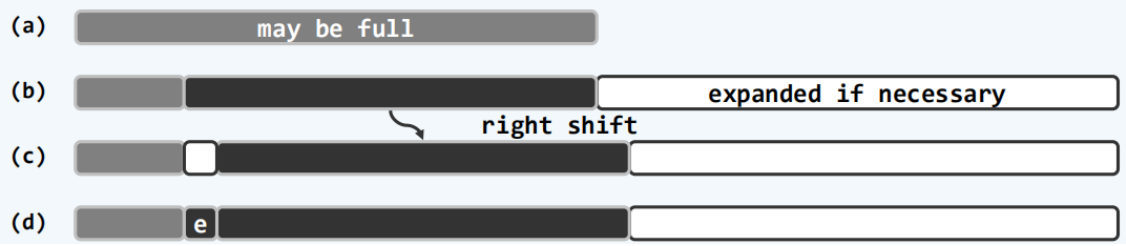
- 代码实现

  Vector_insert.h
  http://dsa.cs.tsinghua.edu.cn/~deng/ds/src_link/vector/vector_insert.h.htm

- 步骤：检查是否需要扩容，然后后续元素右移，然后插入元素，更新容量，返回秩
- 实现代码

```
template <typename T> //将e作为秩为r元素插入
Rank Vector<T>::insert ( Rank r, T const& e ) { //assert: 0 <= r <= size
   expand(); //若有必要，扩容
   for ( int i = _size; i > r; i-- ) _elem[i] = _elem[i-1]; //自后向前，后继元素顺次后移一个单元
   _elem[r] = e; _size++; //置入新元素并更新容量
   return r; //返回秩
}
```

- 分析：第四行for循环是从右开始（即从最后一个元素开始向前）；还有考虑插入之后可能带来的规模容量问题！
- 图示：



▼ 02-C-4 区间删除

- 代码实现

  主要头文件

  vector_removeInterval.h
  https://dsa.cs.tsinghua.edu.cn/~deng/ds/src_link/vector/vector_removeinterval.h.htm

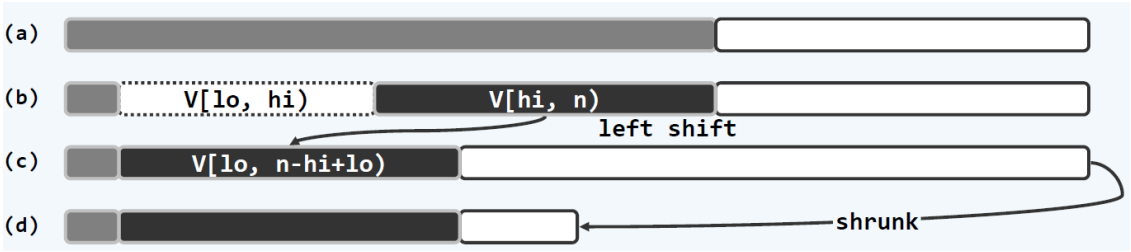
  涉及规模缩减的头文件

  vector_shrink.h
  https://dsa.cs.tsinghua.edu.cn/~deng/ds/src_link/vector/vector_shrink.h.htm

- 自前向后移位！删除区间[lo, hi)

```
template <typename T> int Vector<T>::remove ( Rank lo, Rank hi ) { //删除区间[lo, hi)
   if ( lo == hi ) return 0; //出于效率考虑，单独处理退化情况，比如remove(0, 0)
   while ( hi < _size ) _elem[lo++] = _elem[hi++]; //[hi, _size)顺次前移hi - lo个单元
   _size = lo; //更新规模，直接丢弃尾部[lo, _size = hi)区间
   shrink(); //若有必要，则缩容
   return hi - lo; //返回被删除元素的数目
```

- 图示



▼ 02-C-5 单元素删除

- 代码实现

  vector_remove.h
  https://dsa.cs.tsinghua.edu.cn/~deng/ds/src_link/vector/vector_remove.h.htm

- 直接是为区间删除的特例，`[r] = [r, r+1)`，代码实现：

```
template <typename T> T Vector<T>::remove ( Rank r ) { //删除向量中秩为r的元素，0 <= r < size
   T e = _elem[r]; //备份被删除元素
   remove ( r, r + 1 ); //调用区间删除算法，等效于对区间[r, r + 1)的删除
   return e; //返回被删除元素
}
```

- 若是反推，单元素推广到区间删除
  - 区间删除算法复杂度O(n)
  - 每次循环前移算法复杂度O(n^2)

  反向操作，不切合实际！

❖ 反过来，基于remove(r)接口，通过反复的调用，实现remove(lo, hi)呢？

❖ 每次循环耗时正比于删除区间的后缀长度 = n - hi = O(n)
而循环次数等于区间宽度 = hi - lo = O(n)
如此，将导致总体$O(n^2)$的复杂度

▼ 02-C-6 查找

- 代码实现

  Entry.h
  http://dsa.cs.tsinghua.edu.cn/~deng/ds/src_link/entry/entry.h.htm

  词条类

  vector_find.h
  http://dsa.cs.tsinghua.edu.cn/~deng/ds/src_link/vector/vector_find.h.htm

  查找函数实现头文件

- 实现原理分析

  首先是通过逆序查找，从hi的位置开始 ⇒ 直接命中所查找元素中秩最大者！

  退出条件：低于最低点lo（如果是0，则最终输出-1），或者找到相等的值

```
❖template <typename T> //0 <= lo < hi <= _size
 Rank Vector<T>::find(T const & e, Rank lo, Rank hi) const
 { //O(hi - lo) = O(n)，在命中多个元素时可返回秩最大者
    while ((lo < hi--) && (e != _elem[hi])); //逆向查找
    return hi; //hi < lo意味着失败；否则hi即命中元素的秩
 } //Excel::match(e, range, type)
```

- 输入敏感！有可能直接在高位取到，则复杂度O(1)；最差则可能遍历整个向量O(n)

▼ 02-C-7 唯一化

- 实现代码：

  Vector_deduplicate.h
  http://dsa.cs.tsinghua.edu.cn/~deng/ds/src_link/vector/vector_deduplicate.h.htm

  Vector_uniquify.h
  http://dsa.cs.tsinghua.edu.cn/~deng/ds/src_link/vector/vector_uniquify.h.htm

  vector_sort.h
  http://dsa.cs.tsinghua.edu.cn/~deng/ds/src_link/vector/vector_sort.h.htm

- 应用实例：局部搜索汇总之后的去重
- 代码分析

  从秩为1的元素开始考察（默认第一个元素肯定是不重复的），进行while循环，然后逐一使用find()判断：

  无雷同输出-1 ⇒ 增加一个元素；雷同则调用remove()进行删除！

```
❖template <typename T> //删除重复元素，返回被删除元素数目
 int Vector<T>::deduplicate() { //繁琐版 + 错误版
    int oldSize = _size; //记录原规模
    Rank i = 1; //从_elem[1]开始
    while (i < _size) //自前向后逐一考查各元素_elem[i]
       (find(_elem[i], 0, i) < 0) ? //在前缀中寻找雷同者
          i++ //若无雷同则继续考查其后继
        : remove(i); //否则删除雷同者（至多一个？！）
    return oldSize - _size; //向量规模变化量，即删除元素总数
 }
```

- 正确性分析：具有单调性和不变性

❖ 不变性：在当前元素V[i]的前缀V[0, i)中，各元素彼此互异

初始i = 1时自然成立；其余的一般情况，...

❖ 单调性：随着反复的while迭代

1）当前元素前缀的长度单调非降，且迟早增至_size //1)和2)对应

2）当前元素后缀的长度单调下降，且迟早减至0 //2)更易把握

故算法必然终止，且至多迭代O(n)轮

- 复杂度分析：while循环O(n): find()最坏情况O(n)，则总体最坏情况复杂度O(n^2)
- 进一步优化！

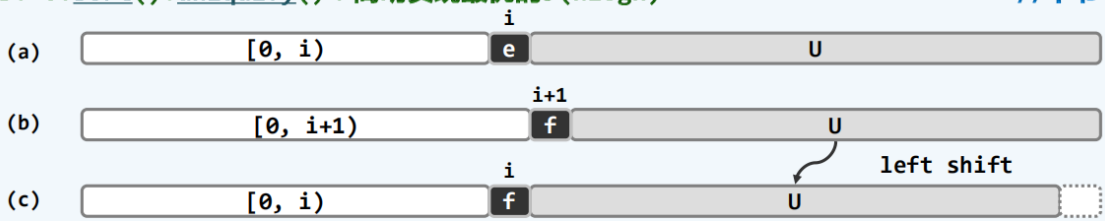
1. 仿照uniquify()高效版的思路，元素移动的次数可降至O(n)

   但比较次数依然是O(n²)；而且，稳定性将被破坏

2. 先对需删除的重复元素做标记，然后再统一删除

   稳定性保持，但因查找长度更长，从而导致更多的比对操作

3. V.sort().uniquify()：简明实现最优的O(nlogn) //下节



▼ 02-C-8 遍历（不理解）

- 实现代码

vector_traverse.h
http://dsa.cs.tsinghua.edu.cn/~deng/ds/src_link/vector/vector_traverse.h.htm

increase_Elem.h
http://dsa.cs.tsinghua.edu.cn/~deng/ds/src_link/_share/increase_elem.h.htm

- 原理：统一对每个元素实现visit操作 ⇒ 问题：如何指定visit？如何将其传递到内部？
- 代码分析
  - 函数指针机制
  - 函数对象机制

❖ 利用函数指针机制，只读或局部性修改

```
template <typename T>
void Vector<T>::traverse(void (*visit)(T&)) //函数指针
   { for (int i = 0; i < _size; i++) visit(_elem[i]); }
```

❖ 利用函数对象机制，可全局性修改

```
template <typename T> template <typename VST>
void Vector<T>::traverse(VST& visit) //函数对象
   { for (int i = 0; i < _size; i++) visit(_elem[i]); }
```

比如，为统一将向量中所有元素分别加一，只需...

首先，实现一个可使单个T类型元素加一的类

```
template <typename T> //假设T可直接递增或已重载操作符“++”
struct Increase { //函数对象：通过重载操作符“()”实现
   virtual void operator()(T & e) { e++; } //加一
};
```

此后...

```
template <typename T> void increase(Vector<T> & V) {
   V.traverse(Increase<T>()); //即可以之为基本操作遍历向量
}
```

没有理解！！！