

## 2. 向量

### (c) 无序向量

邓俊辉

[deng@tsinghua.edu.cn](mailto:deng@tsinghua.edu.cn)

## 元素访问

- ❖ 似乎不是问题：通过 `V.get(r)` 和 `V.put(r, e)` 接口，已然可以读、写向量元素
- ❖ 但就便捷性而言，远不如数组元素的访问方式：`A[r]`      //可否沿用借助下标的访问方式？
- ❖ 可以！为此，需重载下标操作符“`[]`”  

```
template <typename T> //0 <= r < _size  
T & Vector<T>::operator[](Rank r) const { return _elem[r]; }
```
- ❖ 此后，对外的 `V[r]` 即对应于内部的 `V._elem[r]`  
右值： $T \ x = V[r] + U[s] * W[t];$   
左值： $V[r] = (T) \ (2*x + 3);$
- ❖ 为便于讲解，这里采用了简易的方式处理意外和错误（比如，入口参数越界等）  
实际应用中，应采用更为严格的方式

## 插入

❖ template <typename T> //e作为秩为r元素插入,  $0 \leq r \leq \text{size}$

```
Rank Vector<T>::insert(Rank r, T const & e) { //O(n-r)
```

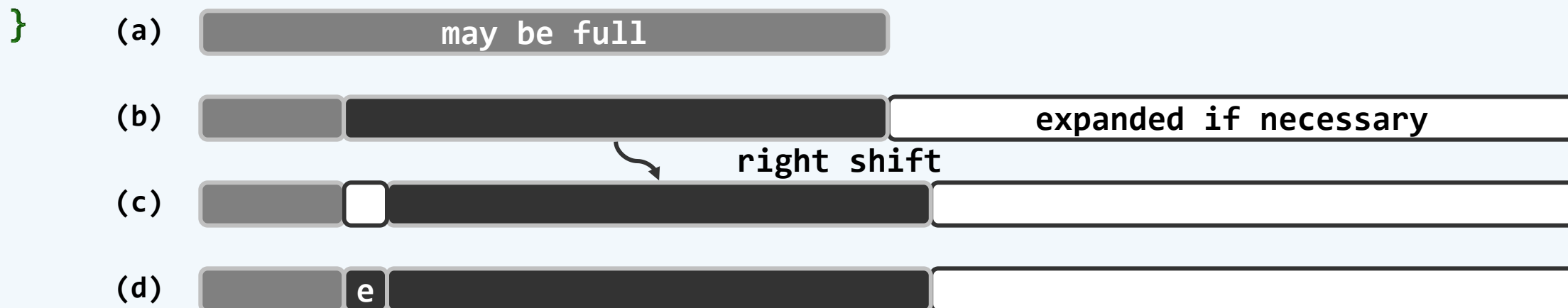
```
    expand(); //若有必要, 扩容
```

```
    for (int i = _size; i > r; i--) //自后向前
```

```
        _elem[i] = _elem[i-1]; //后继元素顺次后移一个单元
```

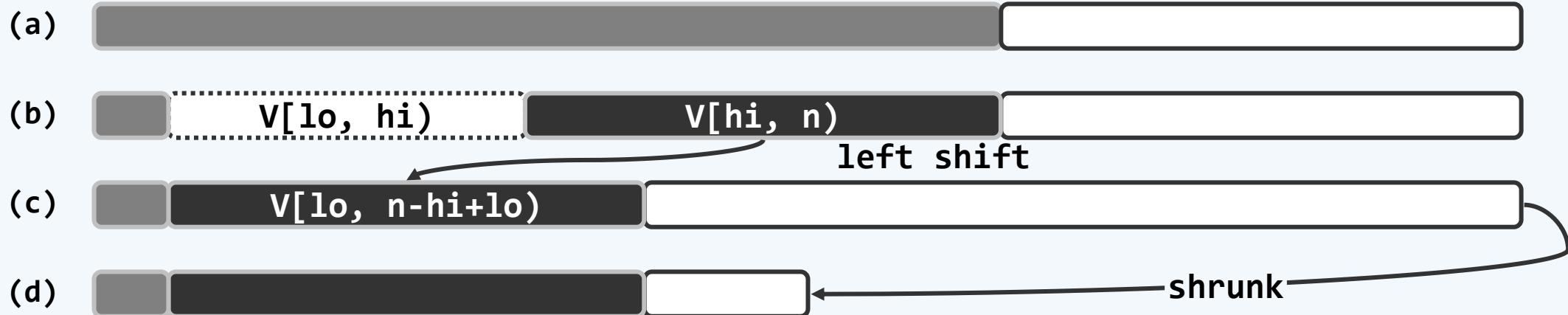
```
    _elem[r] = e; _size++; //置入新元素, 更新容量
```

```
    return r; //返回秩
```



## 区间删除

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



## 单元素删除

❖ 可以视作区间删除操作的特例： $[r] = [r, r + 1)$

❖ `template <typename T> //删除向量中秩为r的元素,  $0 \leq r < \text{size}$`

```
T Vector<T>::remove(Rank r) { //O(n - r)
```

```
    T e = _elem[r]; //备份被删除元素
```

```
    remove( r, r + 1 ); //调用区间删除算法
```

```
    return e; //返回被删除元素
```

```
}
```

❖ 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  $= hi - lo = O(n)$

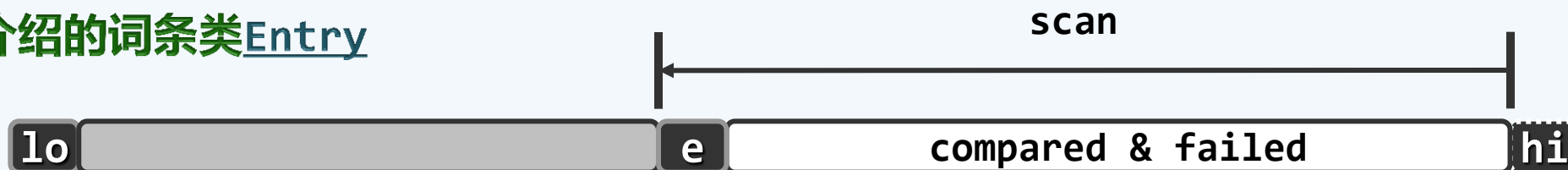
如此，将导致总体 $O(n^2)$ 的复杂度

## 查找

❖ 无序向量：T为可**判等**的基本类型，或已重载操作符"=="或"!="

有序向量：T为可**比较**的基本类型，或已重载操作符"<"或">"

例如：稍后介绍的词条类Entry



❖ `template <typename T> //0 <= lo < hi <= _size`

`Rank Vector<T>::find(T const & e, Rank lo, Rank hi) const`

`{ //0(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 (input-sensitive) : 最好 $O(1)$ ，最差 $O(n)$

## 唯一化：算法

❖ 应用实例：网络搜索的局部结果经过去重操作，汇总为最终报告

❖ `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
```

```
}
```

## 唯一化：正确性

❖ 不变性：在当前元素 $v[i]$ 的前缀 $v[0, i)$ 中，各元素彼此互异

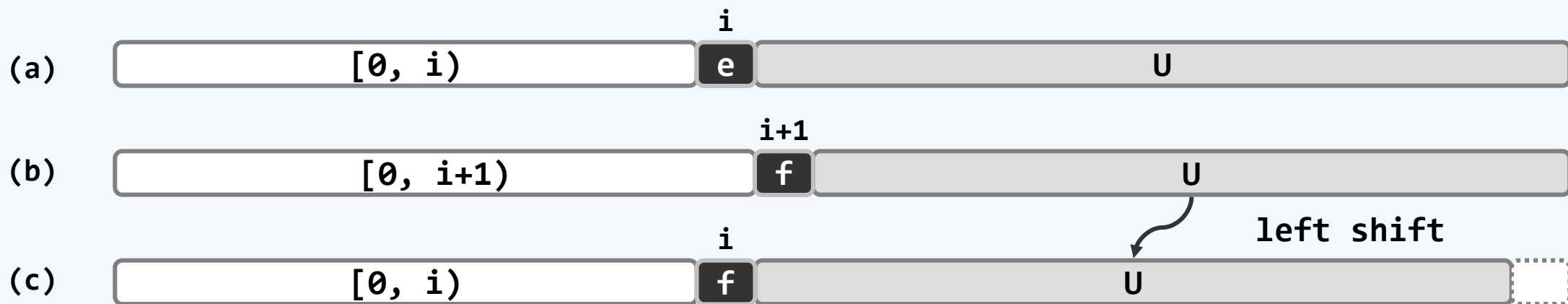
初始 $i = 1$ 时自然成立；其余的一般情况，...

❖ 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





## 唯一化：复杂度

❖ 每轮迭代中`find()`和`remove()`累计耗费线性时间，故总体为 $O(n^2)$

❖ 可进一步优化，比如...

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 $O(n^2)$ ；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 $O(n \log n)$

//下节



## 遍历

❖ 遍历向量，统一对各元素分别实施visit操作

如何指定visit操作？如何将其传递到向量内部？

❖ 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]); }
```

❖ 体会两种方法的优劣

## 遍历：实例

❖ 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  
}
```

❖ 作为练习，可模仿此例，实现减一、加倍，甚至求和等遍历功能