

第一节 接口与实现

2016年8月19日 21:57

3.1.1 两个概念

向量与列表结构同属线性结构

要解决的两个问题：

1. 如何根据统一的接口规范来定制并且实现一个数据结构，这种定制的方法和实现的形式将会被我们后续的数据结构所延用
2. 这种最基本的数据结构，如何通过更加有效的算法使得我们对外的接口能够更加高效率地工作

两个概念：

Abstract Data Type（抽象数据类型）

在一组数据的模型上定义的一组操作（类似于int等普通数据类型，ADT可以定义一个结构，而该结构可以继承ADT的操作）

Data Structure（数据结构）

基于某种特定的语言真正实现的一套完整的算法

3.1.2 向量ADT

向量，可以认为是高级编程语言中数组的推广，是数组的抽象与泛化。

它是由一组元素按线性次序封装而成。

数组下标——秩（rank），循秩访问。

元素类型不限于基本类型。

作为抽象数据类型，向量也提供了很多接口。

3.1.3 接口操作实例

Insert()、put()、get()……

3.1.4 构造与析构

```
❖ Vector(int c = DEFAULT_CAPACITY)
    { _elem = new T[_capacity = c]; _size = 0; } //默认

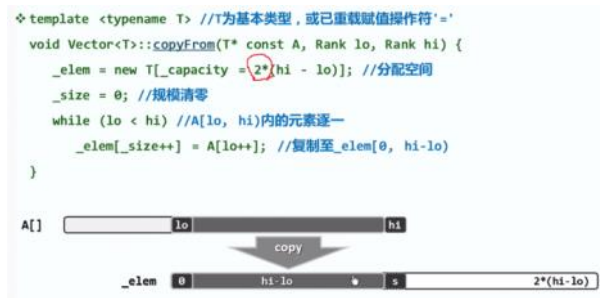
❖ Vector(T const * A, Rank lo, Rank hi) //数组区间复制
    { copyFrom(A, lo, hi); }

❖ Vector(Vector<T> const& V, Rank lo, Rank hi)
    { copyFrom(V._elem, lo, hi); } //向量区间复制

Vector(Vector<T> const& V)
    { copyFrom(V._elem, 0, V._size); } //向量整体复制
```

屏幕剪辑的捕获时间: 2016/8/21 21:48

3.1.5 复制



屏幕剪辑的捕获时间: 2016/8/21 21:52

Lo 不包含在向量元素内

第二节 可扩容向量

2016年8月21日 21:52

3.2.1 可扩容向量

上溢：开辟的空间不足以存放所有数据。

下溢：装填因子（利用率） $\lambda = \text{数据} / \text{可用空间} \ll 50\%$

3.2.2 动态空间管理

思路：在即将发生上溢时，扩大容量空间。



屏幕剪辑的捕获时间: 2016/8/22 11:17

是否可以将视频里向量扩容代码中的：

```
for (int i = 0; i < _size; i++) _elem[i] = oldElem[i];
```

替代为：

```
memcpy(_elem, oldElem, _size * sizeof(T));
```

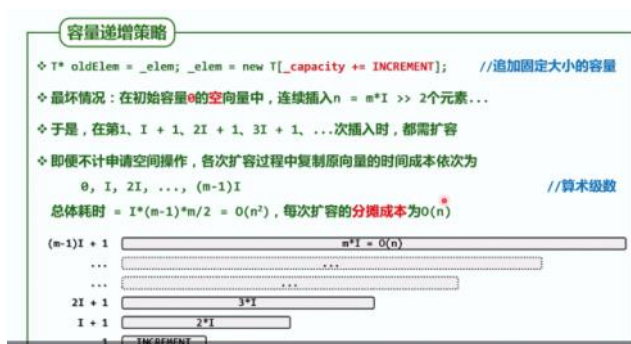
P.S.本题涉及C++的相关知识

否，因为后者能否达到目的与元素类型T有关。正确

EXPLANATION

当T为非基本类型且有对应的赋值运算符以执行深复制时，前一段代码会调用赋值运算符，而后一段只能进行浅复制。

3.2.3 递增式扩容（成本非常之高）



3.2.4 加倍式扩容

容量加倍策略

◇ `T* oldElem = _elem; _elem = new T[_capacity <<= 1];` //容量加倍

◇ 最坏情况：在初始容量1的满向量中，连续插入 $n = 2^m \gg 2$ 个元素...

◇ 于是，在第1、2、4、8、16、...次插入时都需扩容

◇ 各次扩容过程中复制原向量的时间成本依次为

1, 2, 4, 8, ..., $2^m = n$

//几何级数

总体耗时 = $O(n)$ ，每次扩容的分摊成本为 $O(1)$

2^m(m-1)

...

4

2

1

0

8

4

2

1

0

2^m = O(n)

...

8

4

2

1

0

屏幕剪辑的捕获时间: 2016/8/22 11:32

	递增策略	倍增策略
累计 扩容时间	$O(n^2)$	$O(n)$
分摊 扩容时间	$O(n)$	$O(1)$
装填因子	$\approx 10\%$	$> 50\%$

屏幕剪辑的捕获时间: 2016/8/22 11:33

3.2.5 分摊复杂度（分摊分析）

平均分析 vs. 分摊分析

平均分析 (average-case complexity)
假设数据的所有可能排列是均匀分布，并计算其成本平均值
将各种可能的排列，作为独立事件分别考虑
忽略了操作之间的相关性或依赖性
往往不能准确评价数据结构和方法的实际性能

分摊复杂度 (amortized complexity)
对数据结构或算法实施多次操作，并求总成本分摊到每次操作
从实际可行的角度，对一系列操作整体性的考量
更关注实际数据了可能出现的操作序列
可以更准确地评价数据结构和算法的实际性能

屏幕剪辑的捕获时间: 2016/8/22 11:37

分区 数据结构 的第 2 页

第三节 无序向量

2016年8月22日 11:38

3.3.1 概述

无序：没有顺序，不能排列顺序

3.3.2 循秩访问

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i

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);`

屏幕剪辑的捕获时间: 2016/8/22 17:19

3.3.3 插入

插入

```
template <typename T> //e作为秩为r元素插入, 0 <= r <= 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; //置入新元素, 更新容量
    return r; //返回秩
```

屏幕剪辑的捕获时间: 2016/8/22 18:30

3.3.4 区间删除

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
}
```

(a)

(b)

(c)

(d)

屏幕剪辑的捕获时间: 2016/8/22 18:34

由后往前复制，否则会覆盖部分数据（区间重叠部分）。

3.3.5 单元素删除

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, $0 \leq r < \text{size}$`

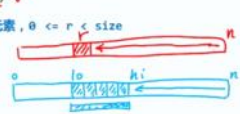
```
T Vector<T>::remove(Rank r) { //  $O(n - r)$ 
    T e = _elem[r]; // 备份被删除元素
    remove(r, r + 1); // 调用区间删除算法
    return e; // 返回被删除元素
}
```

✧ 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



屏幕剪辑的捕获时间: 2016/8/22 18:40

3.3.6 查找

查找

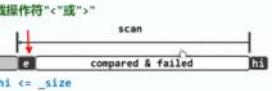
✧ 无序向量: `T` 为可判等的基本类型, 或已重载操作符 `"=="` 或 `"!="`

有序向量: `T` 为可比较的基本类型, 或已重载操作符 `"<"` 或 `">"`

例如: 稍后介绍的词典类 `Entry`

✧ `template <typename T> // $0 \leq lo < hi < \text{size}$`

```
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)
```



屏幕剪辑的捕获时间: 2016/8/22 18:43

输入敏感的算法: 在最好和最坏情况下, 复杂度相差极其悬殊的算法。

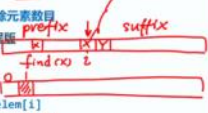
3.3.7 唯一化

唯一化: 算法

✧ 应用实例: 网络搜索的局部结果经过过去重操作, 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() { // 置顶版 + 猫读版
    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
}
```



屏幕剪辑的捕获时间: 2016/8/22 18:56

3.3.8 遍历

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

屏幕剪辑的捕获时间: 2016/8/22 20:32

第四节 有序向量：唯一化

2016年8月22日 20:34

3.4.1 有序性

无序向量：元素是否相等——比对

有序向量：元素大小——比较

有序/无序序列中，任意/总有一对相邻元素顺序/逆序

✧ 因此，相邻逆序对的数目，可用以度量向量的逆序程度

✧ template <typename T> //返回逆序相邻元素对的总数

```
int Vector<T>::disordered() const {
    int n = 0; //计数器
    → for (int i = 1; i < _size; i++) //逐一检查各对相邻元素
        n += (_elem[i - 1] > _elem[i]); //逆序则计数
    return n; //向量有序当且仅当 n = 0
} //若只需判断是否有序，则首次遇到逆序对之后，即可立即终止
```

屏幕剪辑的捕获时间: 2016/8/23 10:02

3.4.2 唯一化（低效版）

低效算法

✧ 观察：在有序向量中，重复的元素必然相互紧邻构成一个区间
因此，每一区间只需保留单个元素即可

✧ template <typename T> int Vector<T>::uniquify() {

```
    int oldSize = _size; int i = 0; //从首元素开始
    → while (i < _size - 1) //从前向后，逐一比对各对相邻元素
        //若雷同，则删除后者；否则，转至后一元素
        (_elem[i] == _elem[i + 1]) ? remove(i + 1) : i++;
    return oldSize - _size; //向量规模变化量，即删除元素总数
} //注意：其中 _size 的减小，由 remove() 隐式地完成
```

屏幕剪辑的捕获时间: 2016/8/23 10:06

3.4.3 复杂度（低效版）

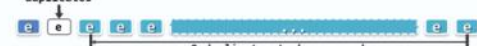
多余的复杂度主要来自于：单个删除所引起的，频繁的remove操作的调用

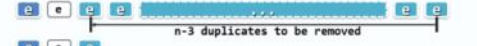
低效算法：复杂度


✧ 运行时间主要取决于while循环，次数共计： $_{size} - 1 = n - 1$


✧ 最坏情况下：每次都需调用remove() 耗时 $O(n-1) \sim O(1)$ ；累计 $O(n^2)$
——尽管省去find()，总体竟与无序向量的deduplicate()相同


duplicates

(a) 

(b) 

(c) 

(d) 

(e) 

屏幕剪辑的捕获时间: 2016/8/23 10:22

3.4.4 唯一化（高效版）

高效算法

✧ 反思：低效的根源在于，同一元素可作为被删除元素的后继多次前移

启示：若能以重复区间为单位，成批删除雷同元素，性能必将改进

```

template <typename T> int Vector<T>::uniquify() {
    Rank i = 0, j = 0; // 各对互异“相邻”元素的秩
    while (++j < _size) // 逐一扫描，直至末元素
        // 跳过雷同者；发现不同元素时，向前移至紧邻于前者右侧
        if (_elem[j] != _elem[j-1]) _elem[++i] = _elem[j]; else ...;
    _size = ++i; shrink(); // 直接截除尾部多余元素
    return j - i; // 向量规模变化量，即被删除元素总数
} // 注意：通过remove(lo, hi)批量删除，依然不能达到高效率

```

屏幕剪辑的捕获时间: 2016/8/23 10:28

3.4.5 实例与分析（高效版）

高效算法：实例与复杂度

✧ 共计 $n - 1$ 次迭代，每次常数时间，累计 $O(n)$ 时间

(a) Initial vector: 3, 3, 3, 3, 5, 5, 5, 5, 5, 8, 8, 8, 13, 13, 13, 13

(b) After removing the first duplicate (3): 3, 3, 3, 5, 5, 5, 5, 5, 8, 8, 8, 13, 13, 13, 13

(c) After removing the second duplicate (3): 3, 5, 3, 3, 5, 5, 5, 5, 5, 8, 8, 8, 13, 13, 13, 13

(d) After removing the third duplicate (3): 3, 5, 8, 3, 5, 5, 5, 5, 5, 8, 8, 8, 13, 13, 13, 13

(e) After removing the fourth duplicate (3): 3, 5, 8, 13, 5, 5, 5, 5, 5, 8, 8, 8, 13, 13, 13, 13

(f) Final vector: 3, 5, 8, 13, 5, 5, 5, 5, 5, 8, 8, 8, 13, 13, 13, 13

屏幕剪辑的捕获时间: 2016/8/23 10:33

第五节 有序向量：二分查找

2016年8月23日 10:35

3.5.1 概述

有序向量：二分查找

无序向量：逐一查找

3.5.2 接口

```
template <typename T> //查找算法统一接口, 0 <= lo < hi <= _size
Rank Vector<T>::search(T const & e, Rank lo, Rank hi) const {
    return (rand() % 2) ? //按各50%的概率随机选用
        binSearch(_elem, e, lo, hi) //二分查找算法, 或者
        : fibSearch(_elem, e, lo, hi); //Fibonacci查找算法
}
```

存在的特殊情况：目标元素不存在；或者存在多个

3.5.3 语义

语义约定

至少，应该便于有序向量自身的维护： $V.insert(1 + V.search(e), e)$
即便失败，也应给出新元素适当的插入位置
若允许重复元素，则每一组也需按其插入的次序排列

约定：在有序向量区间 $V[lo, hi)$ 中，确定不大于 e 的最后一个元素

若 $-\infty < e < V[lo]$ ，则返回 $lo - 1$ (左侧哨兵)

若 $V[hi - 1] < e < +\infty$ ，则返回 $hi - 1$ (末元素 = 右侧哨兵左邻)

保持有序向量的可维护性，例如：插入某个元素，有序向量任然保持有序性。

3.5.4 原理

版本A：原理

减而治之：以任一元素 $x = S[mi]$ 为界，都可将待查找区间分为三部分

$S[lo, mi) <= S[mi] <= S(mi, hi)$ // $S[mi]$ 称作轴点

只需将目标元素 e 与 x 做一比较，即可分三种情况进一步处理：

- $e < x$ ：则 e 若存在必属于左侧子区间 $S[lo, mi)$ ，故可递归深入
- $x < e$ ：则 e 若存在必属于右侧子区间 $S(mi, hi)$ ，亦可递归深入
- $e = x$ ：已在此处命中，可随即返回 // 若有多个，返回何者？

二分（折半）策略：轴点 mi 总是取作中点——于是
每经过至多两次比较，或者能够命中，或者将问题规模缩减一半

3.5.5 实现

版本A：实现

```

template <typename T> //在有序向量区间[lo, hi)内查找元素e
static Rank binSearch(T* A, T const& e, Rank lo, Rank hi) {
    while (lo < hi) { //每步迭代可能要做两次比较判断，有三个分支
        Rank mi = (lo + hi) >> 1; //以中点为轴点
        if (e < A[mi]) hi = mi; //深入前半段[lo, mi)继续查找
        else if (A[mi] < e) lo = mi + 1; //深入后半段(mi, hi)
        else return mi; //在mi处命中
    }
    return -1; //查找失败
}

```

多使用小于号。（当有序向量从小到大排列时，小于号两侧排列与其在向量中的位置一致。）

3.5.6 实例

二分查找的复杂度为 $O(\log n)$ 常系数约为1.5

3.5.7 查找长度



屏幕剪辑的捕获时间: 2016/8/23 11:38

第一节 有序向量：fibonacci查找

2016年8月25日 9:47

4.1.1 构思

由于我们之前的二分查找中，向左总是比向右比较次数少，所以我们考虑能否让向右的递归深度小一些（也就是让二分点不取中点，而是左侧取的数更多一些）这样算法能够进一步化简。



4.1.2 实现

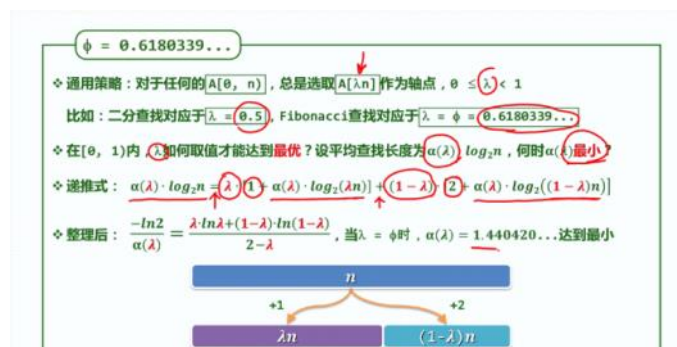
实现

```
template <typename T> // 0 <= lo <= hi <= _size
static Rank fibSearch(T* A, T const & e, Rank lo, Rank hi) {
    Fib fib(hi - lo); // 用 O(log_phi n) = O(log_phi(hi - lo)) 时间创建 Fib 数列
    while (lo < hi) {
        while (hi - lo < fib.get()) fib.prev(); // 至多迭代几次?
        // 通过向前顺序查找，确定形如 Fib(k) - 1 的轴点 (分摊 O(1))
        Rank mi = lo + fib.get() - 1; // 按黄金比例切分
        if (e < A[mi]) hi = mi; // 深入前半段 [lo, mi] 继续查找
        else if (A[mi] < e) lo = mi + 1; // 深入后半段 (mi, hi)
        else return mi; // 在 mi 处命中
    }
    return -1; // 查找失败
}
```

4.1.3 实例

优于 binsearch() 二分查找

4.1.4 最优性



$$\alpha(\lambda) \cdot \log_2 n = \lambda \cdot [1 + \alpha(\lambda) \cdot \log_2(\lambda n)] + (1 - \lambda) \cdot [2 + \alpha(\lambda) \cdot \log_2((1 - \lambda)n)]$$

第二节 有序向量：二分查找（改进）

2016年8月25日 10:50

4.2.1 构思


在二分查找A版本中，问题来源是：分而治之时左右判断不平衡—>而导致这种不平衡在于：循环有三个出口>、<或=。

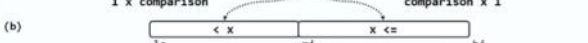
改进思路为：牺牲=这个出口，那么只需判断一次。当我们的区间长度一直缩为1时，我们再判断是否相等。

4.2.2 版本B

版本B：实现

```
template <typename T> static Rank binSearch(T* A, T const& e, Rank lo, Rank hi) {
    while (1 < hi - lo) { //有效查找区间的宽度缩短至1时，算法才会终止
        Rank mi = (lo + hi) >> 1; //以中点为轴点，经比较后确定深入
        (e < A[mi]) ? hi = mi : lo = mi; // [lo, mi) 或 (mi, hi)
    } //出口时 hi = lo + 1, 查找区间仅含一个元素 A[lo]
    return (e == A[lo]) ? lo : -1; //返回命中元素的秩或者-1
} //相对于版本A，最好（坏）情况下更坏（好）；各种情况下的SL更加接近，整体性能更趋稳定
```

(a) 

(b) 


4.2.3 语义

版本B：语义约定

◇ 以上二分查找及Fibonacci查找算法
均未严格地兑现search()接口的语义约定：返回不大于e的最后一个元素

◇ 只有兑现这一约定，才可有效支持相关算法，比如：V.insert(1 + V.search(e), e)

- 1) 当有多个命中元素时，必须返回最靠后（秩最大）者
- 2) 失败时，应返回小于e的最大者（含哨兵[lo - 1]）



我们需要改进我们的版本符合约定语义。

4.2.4 版本C（符合约定语义的版本）

版本C：实现

```
template <typename T> static Rank binSearch(T* A, T const& e, Rank lo, Rank hi) {
    while (lo < hi) { //不变性: A[lo, lo) <= e < A[hi, n)
        Rank mi = (lo + hi) >> 1; //以中点为轴点，经比较后确定深入
        (e < A[mi]) ? hi = mi : lo = mi + 1; // [lo, mi) 或 (mi, hi)
    } //出口时，A[lo = hi]为大于e的最小元素
    return --lo; //故 lo - 1 即不大于e的元素的最大秩
}
```

◇ 与版本B的差异

- 1) 待查找区间宽度缩短至0而非1时，算法才结束
- 2) 转入右侧子向量时，左边界取作mi + 1而非mi — A[mi]会被遗漏？
- 3) 无论成功与否，返回的秩严格符合接口的语义约定...

4.2.5 正确性

版本C: 正确性

- 不变性: $A[lo, lo] \leq e < A[hi, n]$ // $A[hi]$ 总是大于 e 的最小者
- 初始时, $lo = 0$ 且 $hi = n$, $A[0, lo] = A[hi, n] = \emptyset$, 自然成立
- 数学归纳: 假设不变性一直保持至(a), 以下无非两种情况...
- 单调性: 显而易见

Handwritten notes:

- $lo-1$ (with an arrow pointing to the lo index in case (a))
- $e < A[mi]$
- $A[mi] \leq e$
- A red box containing $\leq e$ and $< e$ with a vertical line between them.

Diagram illustrating the partitioning process:

(b) Initial state: $lo = 0$, $hi = n$. The array is divided into two parts: $lo \dots mi$ and $mi+1 \dots n$. The invariant $A[lo, lo] \leq e < A[hi, n]$ is shown.

(a) After the first comparison (comparison x 1), the pivot element $A[mi]$ is compared with e . The invariant is maintained.

(c) After the second comparison (comparison x 1), the pivot element $A[mi]$ is compared with e . The invariant is maintained.

第三节 有序向量：插值查找

2016年8月25日 20:02

4.3.1 原理 Interpolation Search (插值查找)

原理与算法

- 假设：已知有序向量中各元素随机分布的规律
- 比如：均匀且独立的随机分布 $O(\log n)$
- 于是： $[lo, hi]$ 内各元素应大致按照线性趋势增长
- 因此：通过猜测轴点 mi ，可以极大地提高收敛速度
- 例如：在英文词典中

插值公式：

$$mi = lo + (hi - lo) \cdot \frac{e - A[lo]}{A[hi] - A[lo]}$$

在英文词典中

- binary 大致位于 226 处
- search 大致位于 1926 处

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
5	10	12	14	20	31	38	59	42	46	49	51	54	59	72	79	82	86	92

4.3.2 实例

实例

查找元素 $e = 50$

初始范围： $lo = 0, hi = 18$

插值： $mi = 0 + (18 - 0) * (50 - 5) / (92 - 5) = 9.3$

取： $mi = 9$

比较： $A[9] = 46 < e$

更新范围： $lo = 10, hi = 18$

插值： $mi = 10 + (18 - 10) * (50 - 49) / (92 - 49) = 10.2$

取： $mi = 10$

比较： $A[10] = 49 < e$

更新范围： $lo = 11, hi = 18$

插值： $mi = 11 + (18 - 11) * (50 - 51) / (92 - 51) = 10.8$

取： $mi = 10 < lo$

查找完成 (NOT_FOUND)

4.3.3 性能分析

事实：

在插值查找算法中

每经过一次迭代

或者说每经过一次比较

我们都可以将查找的范围

由原先的规模 n 缩减为根号 n

插值查找的复杂度为： $\log \log n$

4.3.4 字宽折半

复杂度的估算：

一个向量的宽度为 n ，我们知道若按照二进制打印以后的位宽为 $\log_2 n$

每一次迭代后，查找范围缩减为 \sqrt{n} 位宽则变为 $1/2 \log n$ 。

联系之前二分查找复杂度 $\log n$ ，这里就变为 $\log \log n$ 。

4.3.5 综合对比

综合

- ❖ 从 $O(\log n)$ 到 $O(\log \log n)$, 是否值得?
- ❖ 通常优势不明显
 - 除非查找区间宽度极大, 或者比较操作成本极高
- 比如, $n = 2^{(2^5)} = 2^{32} = 4G$ 时
 $\log_2(n) = 32, \log_2(\log_2(n)) = 5$
- ❖ 易受小扰动的干扰和“蒙骗”
- ❖ 须引入乘法、除法运算
- ❖ 实际可行的方法
 - 首先通过插值查找, 将查找范围缩小到一定的范围
 - 然后再进行二分查找

大规模: 插值查找
中规模: 折半查找
小规模: 顺序查找

第四节 起泡排序

2016年8月25日 21:03

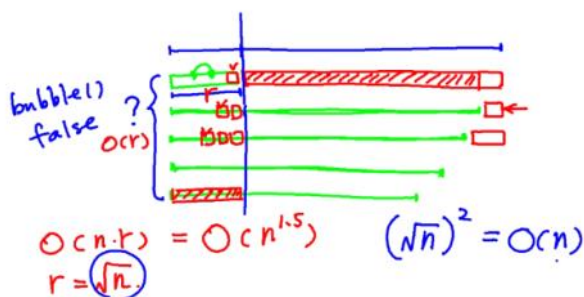
4.4.1 构思 (bubble sort)

我们对之前的起泡排序进行改进。未排序部分如果在上一次循环中没有进行交换（即为不存在逆序对，也就是均为顺序）那么我们就可以提前跳出循环。

4.4.2 改进



4.4.3 反例



如图这样的排序，红色部分已经有序，绿色部分为无序，而绿色部分与红色部分相差很大，不影响。

问题是我们如果能有一种技巧

及时地检测出这样一种情况

也就是说 实质需要排序的元素

集中在一个宽度仅为根号 n 的区间中

而不是整个向量

那么即使套用最原始的起泡排序算法

所需要的时间也无非是

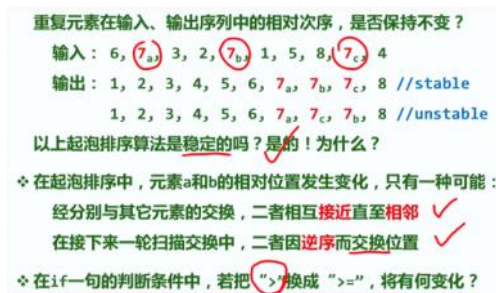
根号 n 的平方等于 $O(n)$

4.4.4 再改进



我们采用变量last，记录下每次扫描中最后一次交换的逆序对。这样在下一趟扫描中便不必对已经有序的部分进行扫描交换。

4.4.5 综合评价



起泡排序具有稳定性，但是最好情况复杂度为 $O(n)$ ，而最坏情况下复杂度为 $O(n^2)$ 。

试用以下算法对 $V=\{19, 17, 23\}$ 排序：

1. 先按个位排序
2. 在上一步基础上，再按十位排序

这个算法的是否正确？

若第2步用的排序算法是稳定的，则正确

若第2步不稳定，可能的情况是： $\{19, 17, 23\} \rightarrow \{23, 17, 19\} \rightarrow \{19, 17, 23\}$

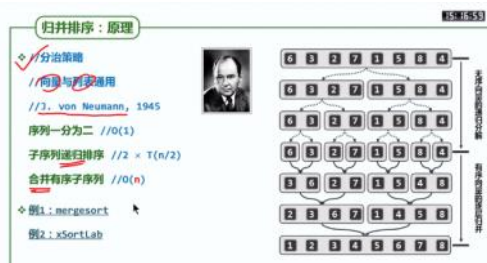
以上算法称为“**基数排序** (radix sort)”，适用于被排序元素可分为若干个域的情况，它的正确性要依赖于对每个域分别排序时的稳定性

第五节 归并排序

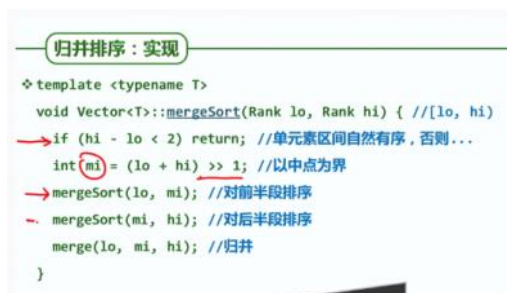
2016年8月26日 11:11

4.5.1 构思 (merge sort)

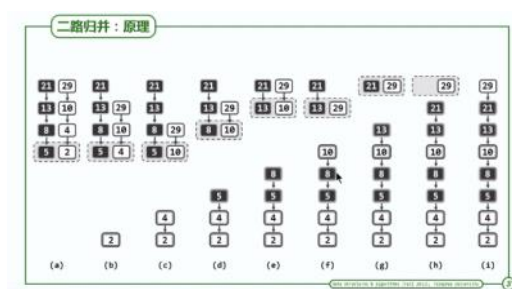
分治策略



4.5.2 归并排序：主算法



4.5.3 二路归并：实例

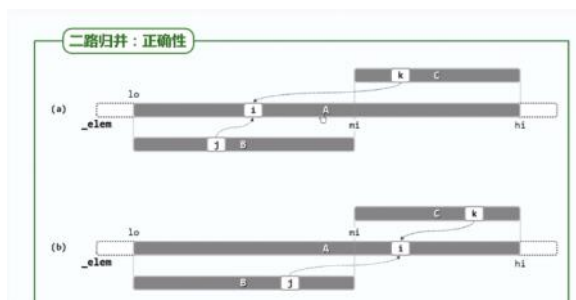


4.5.4 二路归并：实现

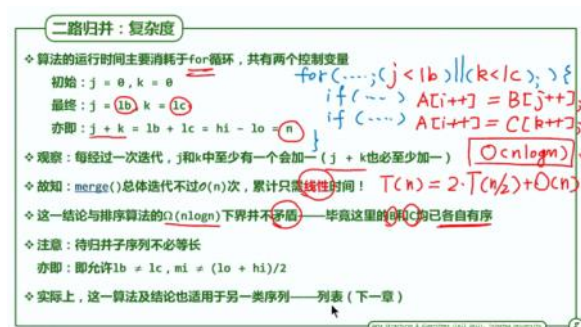


4.5.5 二路归并：正确性

在设计时，由于B向量为复制得来。C向量是直接存放在原向量尾部，所以正确性的判断时，我们要考虑是否会在C中数据会被无意覆盖掉的情况。分类存在以下四种图示情况。



4.5.6 归并排序：性能分析



以下函数是二分查找的递归版，

```

1 template <typename T>
2 static Rank binSearch(T* A, const T &e, Rank lo, Rank hi)
3 { //在A[lo, hi)中查找元素e
4     if (lo == hi) { //递归基
5         return lo - 1;
6     }
7     int mi = (lo + hi) / 2; //中点
8     return e < A[mi] ? binSearch(A, e, lo, mi)
9         : binSearch(A, e, mi + 1, hi); //在子向量中深入查找
10 }

```

对于规模为n的向量，该递归版的时间、空间复杂度和课堂上所学的迭代版的时间、空间复杂度分别是，

- ① $O(n), O(n \log_2(n)), O(n), O(1)$
- ② $O(n \log_2(n)), O(n \log_2(n)), O(n \log_2(n)), O(n \log_2(n))$
- ③ $O(\log_2(n)), O(1), O(\log_2(n)), O(1)$
- * $O(\log_2(n)), O(\log_2(n)), O(\log_2(n)), O(1)$ ✓