2.向量

（b） 可扩充向量

邓俊辉

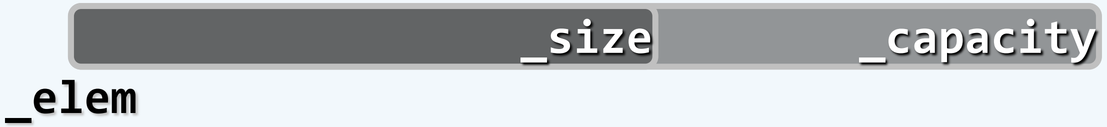
deng@tsinghua.edu.cn

静态空间管理

❖ 开辟内部数组_elem[]并使用一段地址连续的物理空间

_capacity：总容量

_size：当前的实际规模n



❖ 若采用静态空间管理策略，容量_capacity固定，则有明显的不足

1. **上溢**（overflow）：_elem[]不足以存放所有元素

   尽管此时系统仍有足够的空间

2. **下溢**（underflow）：_elem[]中的元素寥寥无几

   装填因子（load factor）λ = _size/_capacity << 50%

❖ 更糟糕的是，一般的应用环境中难以准确**预测**空间的需求量

❖ 可否使得向量可随实际需求动态**调整**容量，并同时保证高效率？

动态空间管理

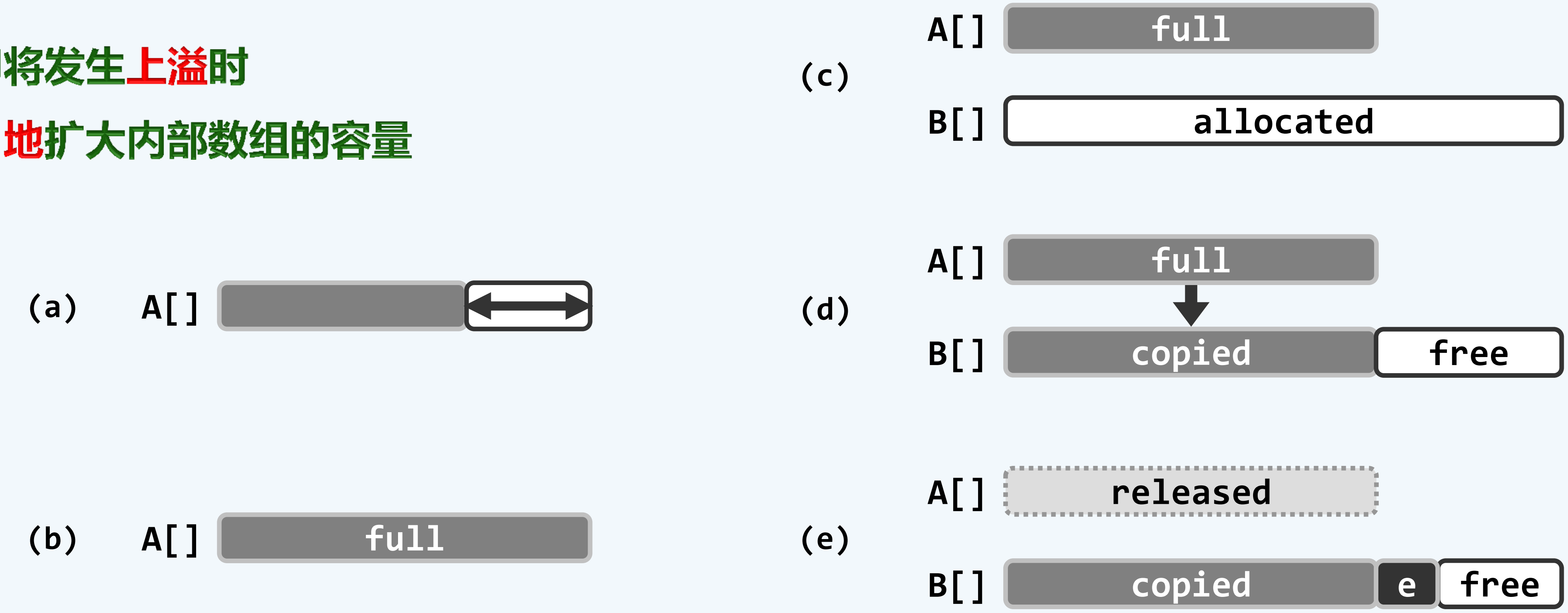
❖ 蝉的哲学：

身体每经过一段时间的生长，以致无法为外壳容纳

即蜕去原先的外壳，代之以...

❖ 在即将发生**上溢**时

适当地扩大内部数组的容量



扩容算法实现

```
❖ template <typename T> void Vector<T>::expand() { //向量空间不足时扩容
     if (_size < _capacity) return; //尚未满员时，不必扩容
     _capacity = max(_capacity, DEFAULT_CAPACITY); //不低于最小容量
     T* oldElem = _elem; _elem = new T[_capacity <<= 1];//容量加倍
     for (int i = 0; i < _size; i++) //复制原向量内容
        _elem[i] = oldElem[i]; //T为基本类型，或已重载赋值操作符'='
     delete [] oldElem; //释放原空间
  } //得益于向量的封装，尽管扩容之后数据区的物理地址有所改变，却不致出现野指针
```

❖ 为何必须采用 容量加倍 策略呢？其它策略是否可行？

容量递增策略

❖ `T* oldElem = _elem; _elem = new T[` $\boxed{\texttt{\_capacity += INCREMENT}}$ `];` //追加固定增量

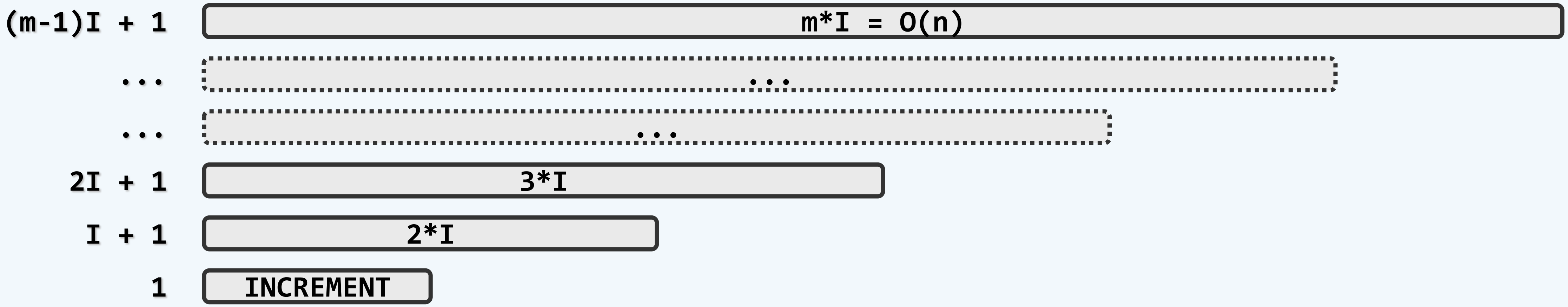
❖ 最坏情况：在初始容量0的空向量中，连续插入n = m*I >> 2个元素...

❖ 于是，在第1、I + 1、2I + 1、3I + 1、...次插入时，都需扩容

❖ 即便不计申请空间操作，各次扩容过程中复制原向量的时间成本依次为

0, I, 2I, ..., (m-1)I //算术级数

总体耗时 = I * (m-1) * m/2 = $O(n^2)$，每次扩容的**分摊成本**为O(n)



容量加倍策略

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

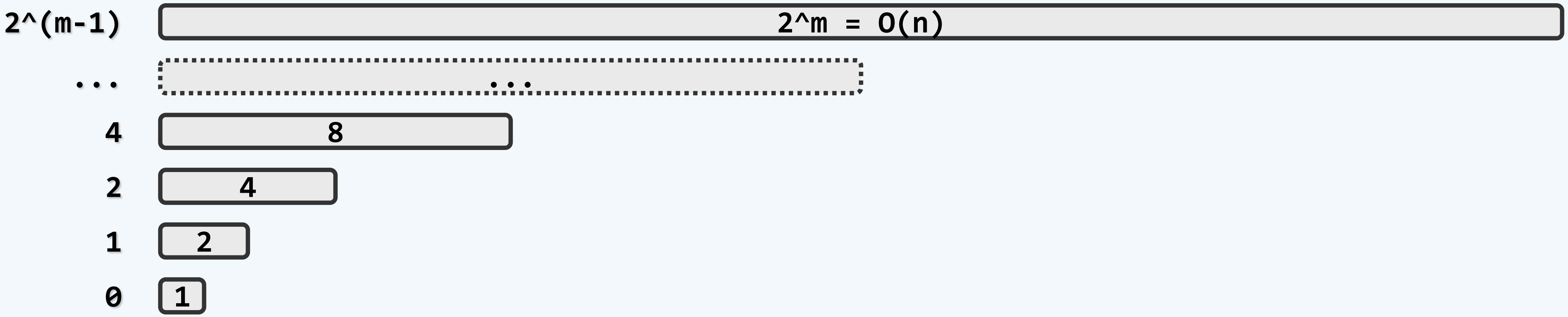
❖ 最坏情况：在初始容量1的满向量中，连续插入n = 2^m >> 2个元素...

❖ 于是，在第1、2、4、8、16、...次插入时都需扩容

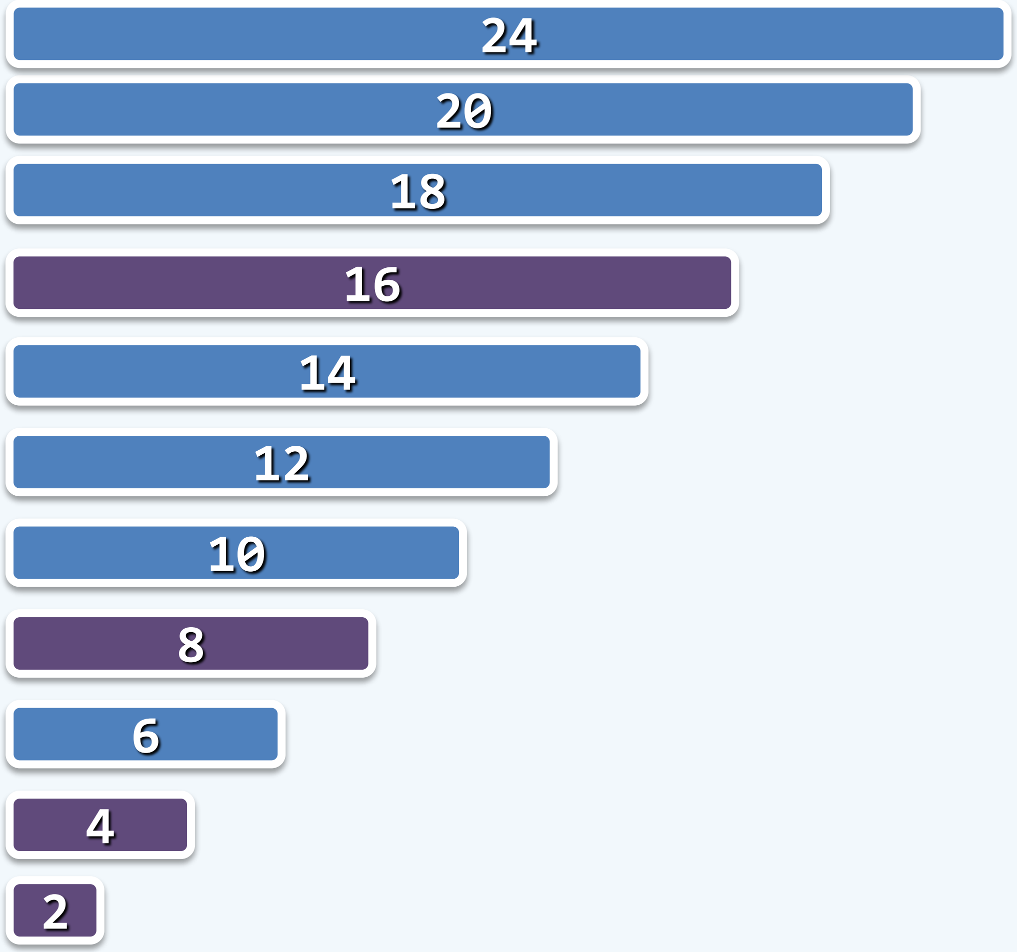
❖ 各次扩容过程中复制原向量的时间成本依次为

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)



对比



	递增策略	倍增策略
累计增容时间	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
分摊增容时间	$\mathcal{O}(n)$	$\mathcal{O}(1)$
装填因子	≈ 100%	> 50%

平均分析 vs. 分摊分析

- 平均复杂度或期望复杂度(average/expected complexity)
  - 根据数据结构各种操作出现概率的分布，将对应的成本加权平均
    - 各种可能的操作，作为**独立**事件分别考查
    - 割裂了操作之间的**相关**性和**连贯**性
    - 往往不能准确地评判数据结构和算法的真实性能
- 分摊复杂度(amortized complexity)
  - 对数据结构**连续**地实施**足够多**次操作，所需总体成本分摊至单次操作
    - 从实际可行的角度，对一系列操作做整体的考量
    - 更加忠实地刻画了可能出现的操作序列
    - 可以更为精准地评判数据结构和算法的真实性能
- 后面将看到更多、更复杂的例子