

## 3. 列表

### (b) 无序列表

邓俊辉

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

## 秩到位置

❖ 可否模仿向量的循秩访问方式？

❖ 可以，比如，通过重载下标操作符

❖ `template <typename T> //assert:  $0 \leq r < \text{size}$`

`T List<T>::operator[] (Rank r) const { //O(r), 效率低下, 可偶尔为之, 却不宜常用`

`Posi(T) p = first(); //从首节点出发`

`while ( $0 < r--$ ) p = p->succ; //顺数第r个节点即是`

`return p->data; //目标节点`

`} //任一节点的秩, 亦即其前驱的总数`

❖ 时间复杂度为 $O(r)$ ，线性正比于待访问节点的秩

以均匀分布为例，单次访问的期望复杂度为

$$(1 + 2 + 3 + \dots + n) / n = (n + 1) / 2 = O(n)$$

## 查找

❖ 在节点p ( 可能是trailer ) 的n个 ( 真 ) 前驱中 , 找到等于e的最后者

❖ `template <typename T> //从外部调用时 , 0 <= n <= rank(p) < _size`

`Posi(T) List<T>::find(T const & e, int n, Posi(T) p) const { //顺序查找 , 0(n)`

`while (0 < n--) //从右向左 , 逐个将p的前驱与e比对`

`if ( e == ( p = p->pred )->data ) return p; //直至命中或范围越界`

`return NULL; //若越出左边界 , 意味着查找失败`

`} //header的存在使得处理更为简洁`

❖ 典型的调用模式 : 通过返回值判定

`x = L.find(e, n, p) ? cout << x->data : cout << "not found";`

❖ `Posi(T) find(T const & e) const //重载全局查找接口`

`{ return find(e, _size, trailer); } //从内部调用时 , rank(trailer) == _size`

## 插入

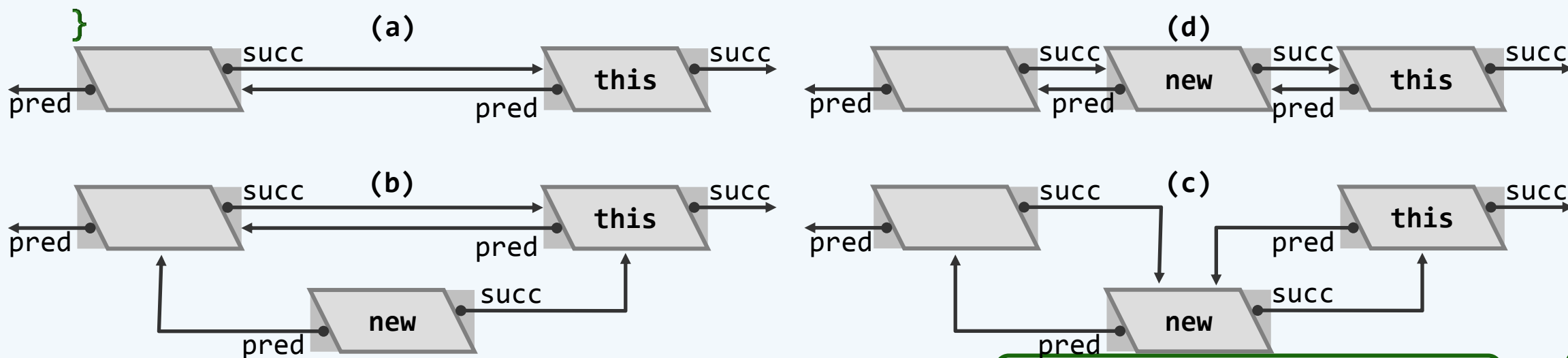
❖ `template <typename T> Posi(T) List<T>::insertBefore(Posi(T) p, T const& e)`  
`{ _size++; return p->insertAsPred(e); }` //e当作p的前驱插入

❖ `template <typename T> //前插入算法 (后插入算法完全对称)`

`Posi(T) ListNode<T>::insertAsPred(T const& e) { //O(1)`

`Posi(T) x = new ListNode(e, pred, this); //创建 (耗时100倍)`

`pred->succ = x; pred = x; return x; //建立链接, 返回新节点的位置`



## 基于复制的构造

❖ `template <typename T> //基本接口`

```
void List<T>::copyNodes(Posi(T) p, int n) { //O(n)
```

```
    init(); //创建头、尾哨兵节点并做初始化
```

```
    while (n--) //将起自p的n项依次作为末节点插入
```

```
        { insertAsLast(p->data); p = p->succ; }
```

```
}
```

❖ 重载的接口

```
List<T>::List(List<T> const& L) //O(_size)
```

```
{ copyNodes(L.first(), L._size); }
```

```
List<T>::List(List<T> const& L, int r, int n) //O(r + n)
```

```
{ copyNodes(L[r], n); }
```

## 删除

❖ template <typename T> //删除合法位置p处节点，返回其数值

```
T List<T>::remove(Posi(T) p) { //O(1)
```

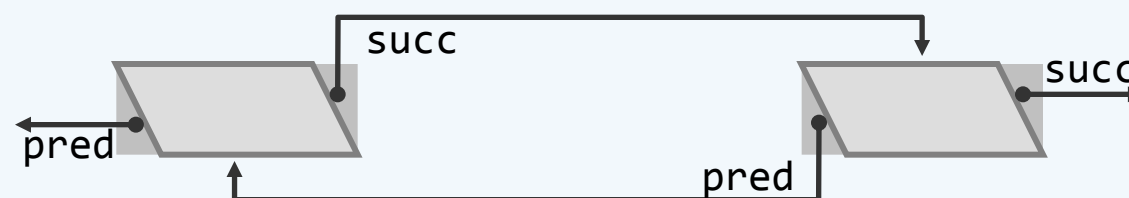
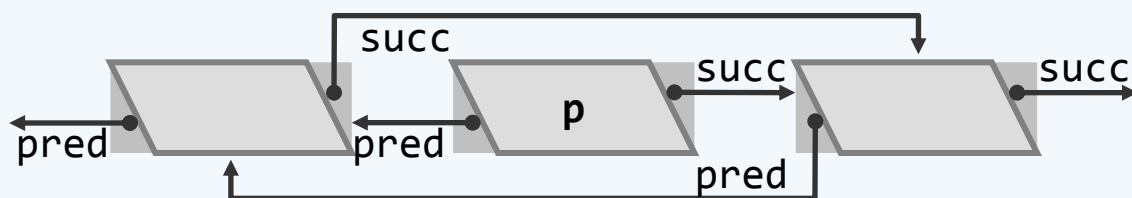
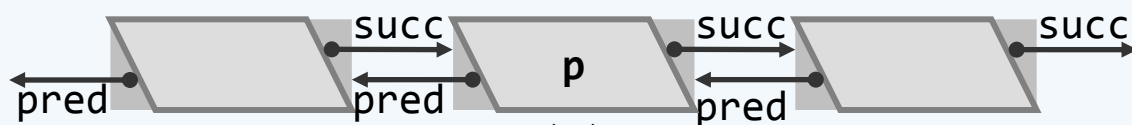
```
    T e = p->data; //备份待删除节点数值（设类型T可直接赋值）
```

```
    p->pred->succ = p->succ;
```

```
    p->succ->pred = p->pred;
```

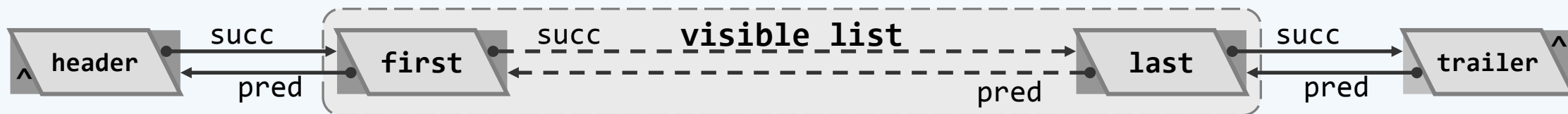
```
    delete p; _size--; return e; //返回备份数值
```

```
}
```



## 析构

- ❖ `template <typename T> List<T>::~~List() //列表析构`  
`{ clear(); delete header; delete trailer; }` //清空列表，释放头、尾哨兵节点
- ❖ `template <typename T> int List<T>::clear() { //清空列表`  
`int oldSize = _size;`  
`while (0 < _size) //反复删除首节点，直至列表变空`  
`remove(header->succ);`  
`return oldSize;`  
`} //O(n)，线性正比于列表规模`



- ❖ 若`remove(header->succ)`改作`remove(trailer->pred)`呢？

## 唯一化

```
❖ template <typename T> int List<T>::deduplicate() { //剔除无序列表中的重复节点

    if ( _size < 2) return 0; //平凡列表自然无重复

    int oldSize = _size; //记录原规模

    Posi(T) p = first(); Rank r = 1; //p从首节点起

    while ( trailer != ( p = p->succ ) ) { //依次直到末节点

        Posi(T) q = find(p->data, r, p); //在p的r个（真）前驱中，查找与之雷同者

        q ? remove(q) : r++; //若的确存在，则删除之；否则秩递增——可否remove(p) ?

    } //assert: 循环过程中的任意时刻，p的所有前驱互不相同

    return oldSize - _size; //列表规模变化量，即被删除元素总数

} //正确性及效率分析的方法与结论，与Vector::deduplicate()相同
```



## 遍历

❖ template <typename T>

```
void List<T>::traverse(void (*visit)(T&)) { //函数指针
```

```
    Posi(T) p = header;
```

```
    while ((p = p->succ) != trailer) visit(p->data);
```

```
}
```

❖ template <typename T>

```
template <typename VST>
```

```
void List<T>::traverse(VST& visit) { //函数对象
```

```
    Posi(T) p = header;
```

```
    while ((p = p->succ) != trailer) visit(p->data);
```

```
}
```