

# 03-B

列表

接口与实现

邓俊辉

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

百只骆驼绕山走，九十八只在山后；尾驼露尾不见头，头驼露头出山沟

# ListNode ADT

❖ 作为列表的基本元素

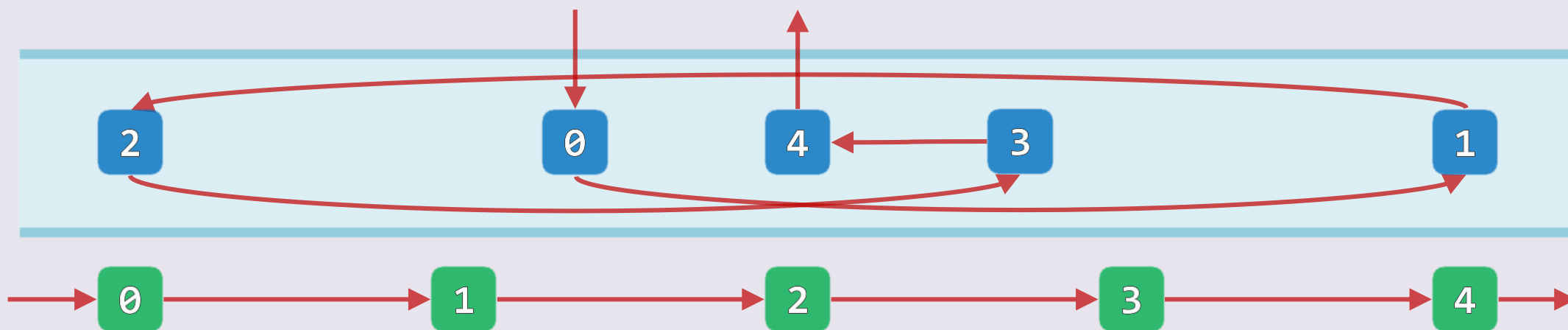
列表节点首先需要

独立地“封装”实现

❖ 列车 ~ 车厢 ~ 货物

list ~ node ~ data

操作接口	功能
pred()	当前节点前驱节点的位置
succ()	当前节点后继节点的位置
data()	当前节点所存数据对象
<u>insertAsPred(e)</u>	插入前驱节点，存入被引用对象e，返回新节点位置
<u>insertAsSucc(e)</u>	插入后继节点，存入被引用对象e，返回新节点位置



# ListNode

```
template <typename T> using ListNodePosi = ListNode<T>*; //列表节点位置 (C++.0x)
```

```
template <typename T> struct ListNode { //简洁起见, 完全开放而不再严格封装
```

```
    T data; //数值
```

```
    ListNodePosi<T> pred; //前驱
```

```
    ListNodePosi<T> succ; //后继
```

```
ListNode() {} //针对header和trailer的构造
```

```
ListNode(T e, ListNodePosi<T> p = NULL, ListNodePosi<T> s = NULL)
```

```
    : data(e), pred(p), succ(s) {} //默认构造器
```

```
ListNodePosi<T> insertAsPred( T const & e ); //前插入
```

```
ListNodePosi<T> insertAsSucc( T const & e ); //后插入
```

```
};
```



# List ADT

操作接口	功能	适用对象
size()	报告列表当前的规模（节点总数）	列表
first(), last()	返回首、末节点的位置	列表
insertAsFirst(e), insertAsLast(e)	将e当作首、末节点插入	列表
insert(p, e), insert(e, p)	将e当作节点p的直接后继、前驱插入	列表
remove(p)	删除位置p处的节点，返回其中数据项	列表
disordered()	判断所有节点是否已按非降序排列	列表
sort()	调整各节点的位置，使之按非降序排列	列表
find(e)	查找目标元素e，失败时返回NULL	列表
search(e)	查找e，返回不大于e且秩最大的节点	有序列表
deduplicate(), uniquify()	剔除重复节点	列表/有序列表
traverse()	遍历列表	列表

# List

```
#include "listNode.h" //引入列表节点类
```

```
template <typename T> class List { //列表模板类
```

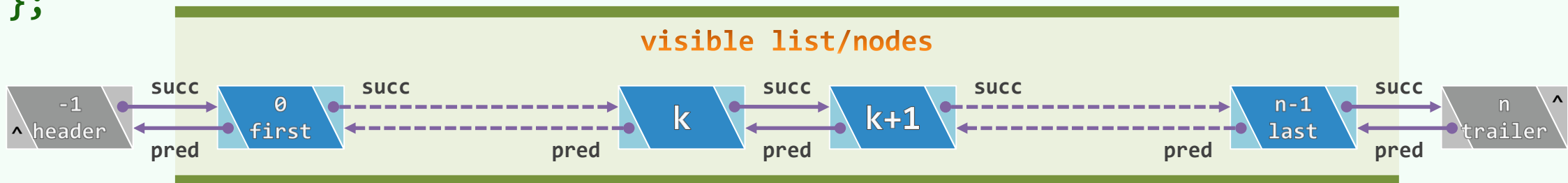
```
private:    Rank _size; ListNodePosi<T> header, trailer; //哨兵
```

```
           //头、首、末、尾节点的秩，可分别理解为-1、0、n-1、n
```

```
protected: /* ... 内部函数 */
```

```
public:     /* ... 构造函数、析构函数、只读接口、可写接口、遍历接口 */
```

```
};
```



# 构造

```
template <typename T> void List<T>::init() { //初始化, 创建列表对象时统一调用
```

```
    header = new ListNode<T>;
```

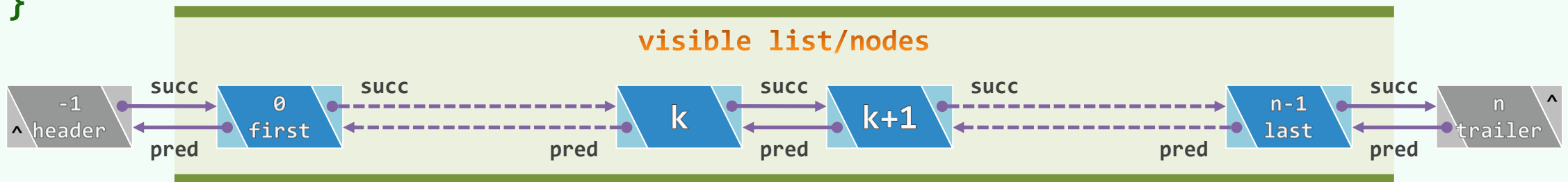
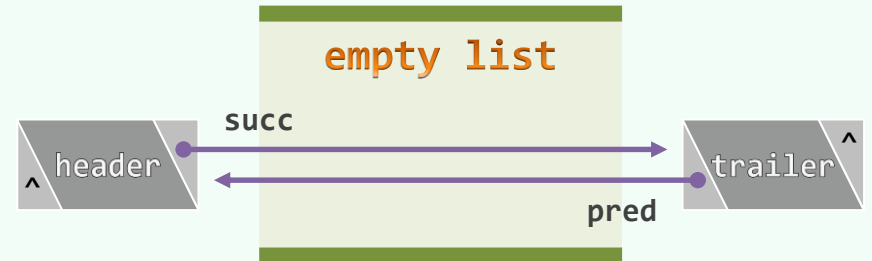
```
    trailer = new ListNode<T>;
```

```
    header->succ = trailer; header->pred = NULL;
```

```
    trailer->pred = header; trailer->succ = NULL;
```

```
    _size = 0;
```

```
}
```



## 重载下标操作符，可模仿向量的循秩访问方式

```
template <typename T> //  $\mathcal{O}(r)$ 效率，虽方便，勿多用
```

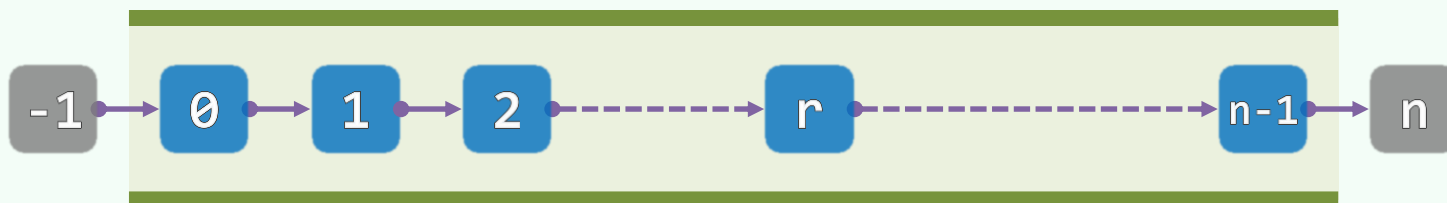
```
ListNodePosi<T> List<T>::operator[] ( Rank r ) const { //  $0 \leq r < \text{size}$ 
```

```
    ListNodePosi<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; //目标节点
```

```
} //秩 == 前驱的总数
```



❖ 时间复杂度为  $\mathcal{O}(r)$

均匀分布时，期望复杂度为  $(1 + 2 + 3 + \cdots + n)/n = \mathcal{O}(n)$