

3. 列表

(a) 接口与实现

Don't lose the link.

- Robin Milner

邓俊辉

deng@tsinghua.edu.cn

从静态到动态

❖ 根据是否修改数据结构，所有操作大致分为两类方式

- 1) 静态：仅读取，数据结构的内容及组成一般不变：get、search
- 2) 动态：需写入，数据结构的局部或整体将改变：insert、remove

❖ 与操作方式相对应地，数据元素的存储与组织方式也分为两种

- 1) 静态：数据空间整体创建或销毁

数据元素的物理存储次序与其逻辑次序严格一致

可支持高效的静态操作

比如向量，元素的物理地址与其逻辑次序线性对应

- 2) 动态：为各数据元素动态地分配和回收的物理空间

逻辑上相邻的元素记录彼此的物理地址，在逻辑上形成一个整体

可支持高效的动态操作

从向量到列表

❖ 列表 (list) 是采用动态储存策略的典型结构

其中的元素称作节点 (node)

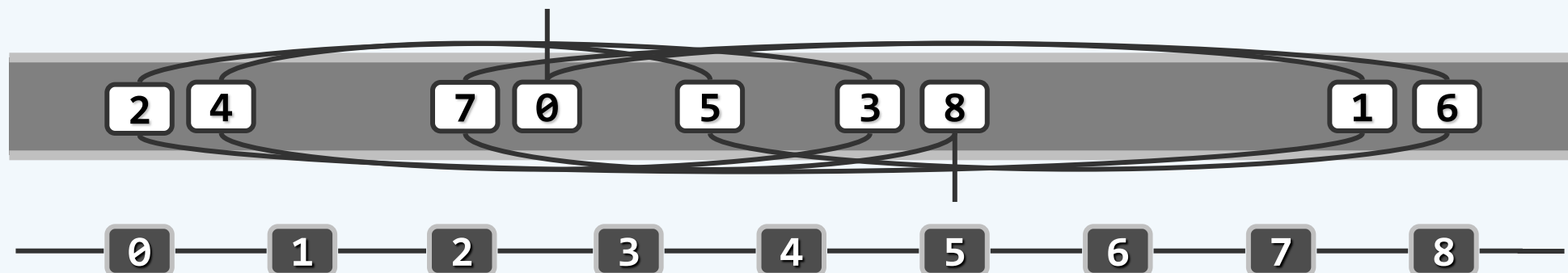
各节点通过指针或引用彼此联接，在逻辑上构成一个线性序列

$$L = \{ a_0, a_1, \dots, a_{n-1} \}$$

❖ 相邻节点彼此互称前驱 (predecessor) 或后继 (successor)

前驱或后继若存在，则必然唯一

没有前驱/后继的唯一节点称作首 (first/front) /末 (last/rear) 节点



从秩到位置

❖ 向量支持循秩访问 (call-by-rank) 的方式

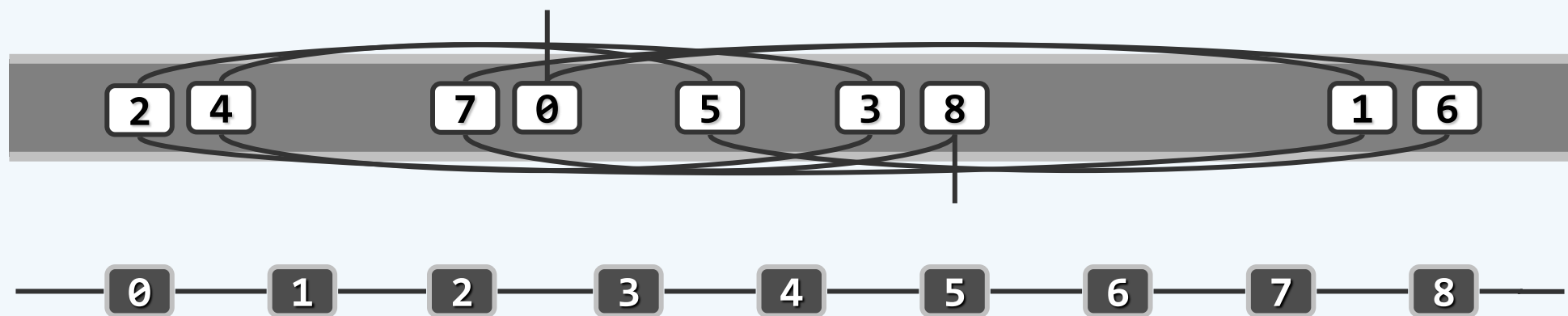
根据数据元素的秩，可在 $O(1)$ 时间内直接确定其物理地址

$V[i]$ 的物理地址 = $V + i \times s$ ， s 为单个单元占用的空间量

❖ 比喻：假设沿北京市海淀区的街道 V ，各住户的地理间距均为 s

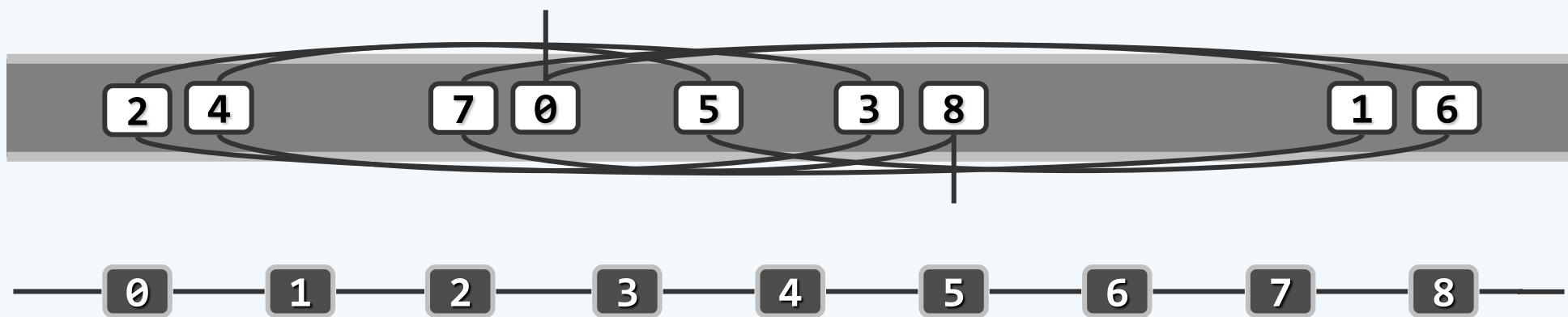
则对于门牌号为 i 的住户，地理位置 = $V + i \times s$

❖ 这种高效的方式，可否被列表沿用？



从秩到位置

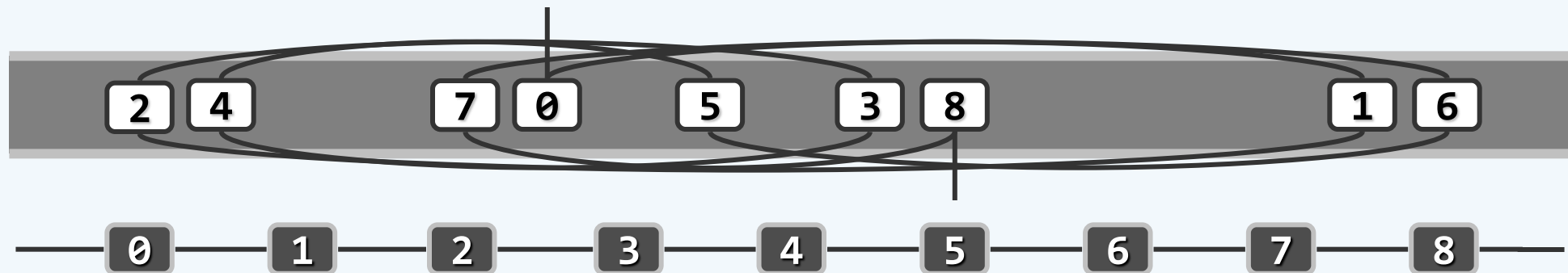
- ❖ 既然同属线性序列，列表固然也可通过秩来定位节点：从头/尾端出发，沿后继/前驱引用...
- ❖ 然而，此时的循秩访问成本过高，已不合时宜 `//List::operator[](Rank r)`，下节详解
//兼顾两种访问方式的skiplist，第九章
- ❖ 因此，应改用**循位置访问**（call-by-position）的方式
亦即，应转而利用节点之间的相互引用，找到特定的节点
- ❖ 比喻：找到 我的朋友A 的亲戚B 的同事C 的战友D ... 的同学Z



列表节点：ADT接口

❖ 作为列表的基本元素，列表节点首先需要独立地“封装”实现
为此，可设置并约定若干基本的操作接口

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



列表节点：ListNode模板类

```
❖ #define Posi(T) ListNode<T>* //列表节点位置 ( ISO C++.0x, template alias )

❖ template <typename T> //简洁起见，完全开放而不再过度封装

    struct ListNode { //列表节点模板类（以双向链表形式实现）

        T data; //数值

        Posi(T) pred; //前驱

        Posi(T) succ; //后继

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

        ListNode(T e, Posi(T) p = NULL, Posi(T) s = NULL)

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

        Posi(T) insertAsPred(T const& e); //前插入

        Posi(T) insertAsSucc(T const& e); //后插入

    };
```



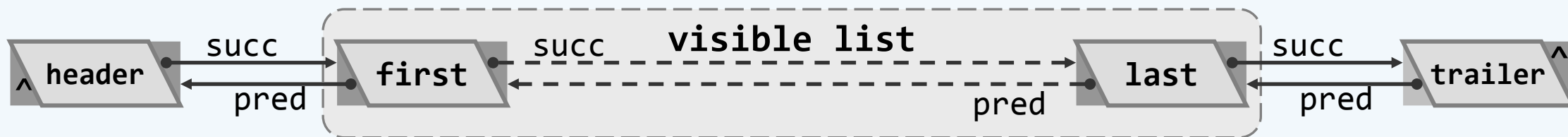
列表：ADT接口

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

列表：List模板类

```
❖ #include "listNode.h" //引入列表节点类

❖ template <typename T> class List { //列表模板类
private:    int _size; //规模
           Posi(T) header; Posi(T) trailer; //头、尾哨兵
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 */
};
```



❖ 等效地，头、首、末、尾节点的秩可分别理解为-1、0、n-1、n

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>; //创建头哨兵节点  
    trailer = new ListNode<T>; //创建尾哨兵节点  
    header->succ = trailer; header->pred = NULL; //互联  
    trailer->pred = header; trailer->succ = NULL; //互联  
    _size = 0; //记录规模  
}
```

