第3章

数据结构

在运行缓慢的程序中更换数据结构, 其效果类似于对患者实施器官移植手术。那些重要的**抽象数据类型**(abstract data types), 如容器、字典和优先级队列等, 都可用功能等价但手段各异的**数据结构**(data structures)实现。由于从原理上来讲, 我们仅仅是从一种正确的实现方案转换为另一种同样正确的实现方案而已, 所以更换数据结构不会影响到程序的正确性。然而, 新的实现方法会重新协调抽象数据类型中各种操作的执行时间, 因此整个程序的性能会有惊人的改善。正如患者所需要的只是移植某一个器官, 解决程序运行缓慢的问题可能也仅需更换一个模块。

不过, 生来就有颗健康心脏的人显然比苦等移植手术的患者更幸福。因此, 从一开始便围绕着优良的数据结构进行程序设计, 才能取得最好效果。我们假设读者此前对基本的数据结构和指针操作已有所了解。不过, 如今的数据结构课程(CS Ⅱ)[1]更关注数据抽象和面向对象, 而较少涉及应如何在内存中表示数据的存储结构这样的细节问题。我们将对这部分内容进行回顾, 以确保你能着实明了。

学习数据结构与其他科目一样, 掌握基本内容远胜于略知高深概念, 所以我们将侧重于三种基本的抽象数据类型(即容器、字典和优先级队列), 并考虑如何用数组和列表实现它们。当然还存在众多更为精巧的实现方法, 而选用时应如何权衡的细节问题将留待卷Ⅱ中相应抽象数据类型的词条中详加讨论。

3.1 紧接数据结构与链接数据结构

数据结构基本上可简单地划分为两类: **紧接**(contiguous)和**链接**(linked), 这取决于该数据结构是基于数组还是指针:

- **紧接分配数据结构**(contiguous allocated data structures)由许多相邻的内存块组合而成一体, 数组、矩阵、堆和散列表皆属此类。
- **链接数据结构**(linked data structures)由大量分散于各处的内存块通过指针连接而成, 列表、树和图的邻接表皆属此类。

在本节中, 我们将评述紧接数据结构与链接数据结构各自的相对优势。二者的权衡相当微妙, 绝非一目了然的事。因此, 即便是对以上两种数据结构都已熟悉的读者, 我依然强烈建议你一起来重温这些内容。

[1] 译者注: 作者仍采用ACM Curriculum'78对计算机科学课程的划分方法, 其中CS Ⅱ水平的课程一般需要介绍基本数据结构知识, 并深入讨论程序设计的方法。最新课程指导方案可见http://www.acm.org/education/curricula-recommendations。

3.1.1 数组

数组(array)是最基本的紧接数据结构。数组是一种由固定大小的数据记录所组成的结构, 它能让我们通过**下标**(index)或(与其等价的)存储地址高效定位其中的元素。

将数组比作一条盖满房屋的街道真是再恰当不过了, 数组元素相当于这里的房屋, 而数组下标则相当于房屋门牌号。假定所有房屋大小相同, 且从1到n依次编号, 那么可通过门牌号迅速算出每间房屋的精确位置。[1]

紧接分配的数组具有下列优点:

- **给定下标可在常数时间内访问元素**(constant-time access given the index) —— 由于数组中每个元素的下标直接对应了内存中的特定地址, 如果能知道下标, 则可迅速访问该数据项。
- **节约空间**(space efficiency) —— 数组完全由数据构成, 不存在因链接或其他格式信息(formatting information)而造成的浪费。此外, 数组由数目固定的若干记录构成, 从而不需要记录终止(end-of-record)信息来标记数组结束。
- **内存局部性**(memory locality) —— 编程常常需要对数据结构中所有元素从头至尾进行迭代, 事实上这是程序设计中的一种惯用法(idiom)。由于数组体现了极好的内存局部性, 因此它非常适宜于开展迭代。在对数据进行相继访问时, 数据地址在物理上的连续性有助于充分利用现代计算机系统结构中的高速*缓存*。

数组的不足之处在于, 我们无法在程序执行过程中调整数组的大小。假定初始仅分配了能容纳n个记录的空间(房间), 如果我们试图加入第$n+1$个记录(房客), 程序便会立刻崩溃。为弥补此缺陷, 我们可在初始时分配非常大的数组, 但这种方案很浪费空间, 而且仍然会面临空间不足的问题。

实际上, 利用神奇的**动态数组**(dynamic arrays), 我们便可根据需要让数组高效地扩张。假定数组的初始大小为1, 每当面临空间耗尽的情况时, 便将其长度加倍, 即从m变成$2m$。这种加倍过程的操作步骤如下: 首先按紧接方式分配一个长为$2m$的新数组, 再将原数组中的内容复制到新数组的前半部分, 最后将原数组所占空间退还给存储分配系统。

很明显, 在每次加倍扩张过程中对原有内容重新复制是一种重复劳动。那么在n次插入后, 我们究竟需要多少次复制工作呢?[2] 假设n为2的幂, 那么当数组刚拥有n个位置时, 最后一次插入操作还会导致一次加倍, 而在此之前我们还执行了$\log n$次加倍操作。事实上, 数组在第$1, 2, 4, 8, \cdots, n$次插入时都会因空间不够而扩张, 而每次都需重新复制数组中所有元素。易知第i次加倍时复制了2^{i-1}个元素, 因此总的搬运(复制)次数M可由下式给出:

$$M = n + \sum_{i=1}^{\log n} 2^{i-1} = n + \left(\frac{n}{2} + \frac{n}{4} + \cdots + 2 + 1\right) = \sum_{i=0}^{\log n} \frac{n}{2^i} \leqslant n \sum_{i=1}^{\infty} \frac{1}{2^i} = 2n$$

实际上, 每个元素平均只需搬运2次。你可以看到, 处理这样一个动态数组总共才花费了$O(n)$时间, 而事先分配足够大的静态数组再逐个放入元素所需时间也是这个量级!

[1] 日本的房屋不以物理位置编号, 它们通常根据建造次序来编号。若无一份详尽的地图, 在日本一般很难找到某地址所在的具体位置。

[2] 译者注: 注意数组初始大小为1且包含一个元素, 随后连续插入n个元素, 最终数组中共有$n+1$个元素。

使用动态数组不能保证在最坏情况下以常数时间插入元素, 这是其主要缺陷。不过, 除了极少数(相对而言)[1]可能引发数组加倍的操作之外, 所有访问操作以及大部分插入操作都能较快地完成。那么我们将得到另一种承诺: 完成对第n个元素的插入操作还算比较快, 而这个"快"的意思是到该项操作为止的所有插入操作总共只需$O(n)$时间。实际上, 类似这样**分摊**(amortized)意义下的保证会经常在数据结构的分析中出现。

3.1.2 指针与链接结构

指针(pointer)是联系链接结构中各个部分的纽带, 因为它代表着内存位置的地址。相比于用变量直接存储某数据项自身值的副本这种方案而言, 指向该数据项的指针变量能提供更多自由度。你可以认为手机号码是指向它主人的指针, 因为手机用户在世界上随意来去。

关于指针的使用方法和能力范围, 不同的程序设计语言中有着明显的差异, 因此我们先简要复习C语言中的指针。假定指针`p`对应着内存中的一处地址, 那么你可通过`p`访问位于此处的一个特定数据块。[2] C语言中的指针一般带有编译时(compile time)事先声明的数据类型, 这意味着它只能指向此类型的变量。我们记指针`p`所指向的变量为`*p`, 且记某变量`x`所在的地址(可用指针存储)为`&x`。此外, `NULL`是一种特殊的指针值, 用于表示数据结构的终止或未赋值的指针。

下面的链表(linked list)类型声明展现了所有链接数据结构都具备的一些共性:

```
typedef struct list {
    item_type item;             /* 数据项 */
    struct list *next;          /* 指向后邻的指针 */
} list;
```

图3.1基于`list`类型展示了一个存放人名的链表实例, 我们可以发现, 链接数据结构的下列特性尤为重要:

- 在链接数据结构(此处为`list`)中, 每个结点都包含一个或多个数据域(此处仅有一个`item`), 这些数据域保留着我们需要存储的数据。
- 每个结点至少包含一个指针域, 每个指针域(此处仅有一个`next`)会指向一个其他结点。这意味着, 链接数据结构所用空间中有相当一部分得贡献给指针, 而未能用于存储数据。
- 最后, 我们需要一个指向数据结构头部的指针, 这样才知道从何处开始访问该数据结构。

图 3.1 链表示例: 结点的数据域和指针域

[1] 译者注: 从量级的角度看, $\log n$相比于n绝对可以称得上是极少数了。

[2] C语言允许直接操纵内存地址, 这种方式会让Java程序员心烦意乱, 不过我们将会避免使用这种技巧。

这里的`list`实际上是**单链表**(singly linked list), 它是最简单的链接结构, 可用于实现**列表**(list)这种抽象数据类型, 该类型支持三种最基本的操作: 查找、插入和删除。在**双链表**(doubly linked lists)中, 每个结点同时包含了指向前邻(predecessor)元素和后邻(successor)元素的指针, 这种方式能简化某些操作, 不过要以每个结点中再多存储一个指针为代价。

1. 在列表中查找

在列表中查找数据项x可通过迭代或递归实现。在下面的实现中我们选用递归方式: 如果x在列表中, 要么x就是第一个元素要么x位于余下部分(比原列表小)。我们最终可将原问题简化到在空表中查找的情况, 而空表显然不可能包含x。

```
list *search_list(list *l, item_type x)
{
    if (l == NULL)
        return NULL;

    if (l->item == x)
        return l;
    else
        return search_list(l->next, x);
}
```

2. 在列表中插入

在一个单链表插入元素是非常好的指针操作练习, 其解答如下文所示。[1] 由于我们不需要以任何形式的次序维护列表, 不妨将每个新项插入至最易于到达的位置。插入到列表头部无需进行任何遍历, 但该方法需要我们对指向列表数据结构头部的指针(记为`l`)进行更新。

```
void insert_list(list **l, item_type x)
{
    list *p;            /* 暂用指针 */
    p = malloc(sizeof(list));
    p->item = x;
    p->next = *l;
    *l = p;
}
```

这里需要指出两个C语言的独特用法: 其一, `malloc`函数会为新结点分配一块足以包含x的内存; 其二, 双重星号(`**l`)常令人迷惑, 这种记法表示`l`是指向某个指针(`*l`)的指针, 而`*l`指针才真正指向某个列表结点。因此在程序的最后一行, `*l = p;`语句将`p`的值复制到`l`所指向的位置, 该位置所存储的外部变量即指向列表头部的指针。

3. 在列表中删除

从列表中删除元素略为复杂。我们首先要找到待删除项的前邻指针, 可通过递归方式得到(`x`指向待删除项):

[1] 译者注: 这里考虑的是无序列表, 因此插入位置不太重要。读者不妨思考如何维护有序列表的问题。

```
list *item_ahead(list *l, list *x)
{
    if ((l == NULL) || (l->next == NULL)) {
        printf("Error: predecessor sought on null list.\n");
        return NULL;
    }

    if ((l->next) == x)
        return l;
    else
        return item_ahead(l->next, x);
}
```

这个前邻必须得找出来, 因为删除操作完成后它会指向一个已不存在的结点, 所以必须更改前邻结点的`next`指针。一旦排除了待删除元素不存在的情况, 那么后续的实际删除工作其实非常简单。需要特别注意的是, 当首元素被删除时, 列表的头部(`l`)必须重置(`*z`指向待删除项):

```
void delete_list(list **l, list **z)
{
    list *p;                /* 暂用指针 */
    list *pred;             /* 前邻指针 */

    p = *l;
    pred = item_ahead(*l, *z);

    if (pred == NULL)       /* 删除后所要粘接的前邻完全不存在 */
        *l = p->next;
    else
        pred->next = (*z)->next;
    free *z;                /* 释放结点所占用的内存 */
}
```

C语言要求显式地去释放已分配的内存, 因此当我们不再需要某结点时, 必须使用`free`操作将已删除结点所占用的内存还给系统。不过, 这将让原有指针变成**空悬引用**(dangling reference), 也即指向某个不复存在的位置, 所以在使用该指针时得特别小心。当然, Java中通常没有此类问题, 因为它的内存管理模型更强一些。

3.1.3 对比

相对于静态数组而言, 链表的优点是:

- 使用链接结构不会发生溢出现象, 除非内存确实已满。
- 插入和删除比静态数组中的相应操作要简单。
- 处理大记录时, 比起移动记录项本身来说, 改变指针值既简单又迅速。

而相对于链表而言, 数组的优点是:

- 数组不浪费空间, 链表却需要额外空间以存储指针域。
- 随机访问数组中的元素非常高效。

- 相比于随机的指针跳转而言, 只有使用数组才有可能更好地利用内存局部性和缓存性能。

领悟要义: 对于有限的存储资源应如何使用且在何处更能发挥作用的问题, 动态内存分配为我们提供了灵活的解决方案。

最后, 关于这两种基本数据结构还存在一种统一的处理思想, 也即它们都可视为递归式对象:

- **链表**[1]—— 除去链表的首元素后, 余下部分是一个稍小的链表。这个论断同样也适用于字符串, 因为从字符串中移除部分字符后, 剩下部分仍是字符串。由此可见, 链表是递归式对象。
- **数组** —— 从一个具有n个元素的数组中分离出前k个元素, 将会得到两个稍小的数组, 大小分别为k和$n-k$。因此, 数组也是递归式对象。

这种观点能引出更简洁的表处理过程, 还能启发我们设计高效的分治算法, 比如快速排序和二分查找。

3.2 容器: 栈与队列

容器(container)这个术语代表一种抽象数据类型, 它允许对数据项进行*不依赖于数据内容*的放入和取出。[2] 与之相对的是**字典**(dictionary), 它是一种基于键值或内容来检索和处理元素的抽象数据类型, 我们将在3.3节中讨论。

我们可通过容器支持的特定取出次序来区分容器的类型。在下面两类最重要的容器(栈和队列)中, 其取出次序取决于放入元素的次序。

- **栈**(stack) —— 支持按"后进先出"(LIFO)次序取出。栈不但易于实现, 而且也非常高效。因此, 对取出次序无任何特殊要求时(例如批处理作业[3]的执行), 栈很可能是最适合的容器。对栈的*放入*和*取出*操作通常称为**推入**(Push)和**弹出**(Pop):

  - Push(x, s): 在栈s的顶端插入x。
  - Pop(s): 返回(并删除)[4]栈s的顶端数据项。

  我们在许多现实生活的场景中都会遇到LIFO次序。例如挤在地铁车厢的乘客遵循LIFO次序下车。又如塞进我家冰箱中的食品, 不管它们如何提醒"快到保质期了!", 通常还是按LIFO次序取出。在算法领域中, LIFO往往会出现于递归算法的运行过程中。

[1] 译者注: 原文在此处使用的是"列表", 这与前文提及的"数据结构"不符, 因此我们改为"链表"。不过, 作为抽象数据类型的列表确实也是递归的。

[2] 译者注: 这里将retrieval译为"取出", 而不是依赖于内容的"检索"。

[3] 译者注: 参见https://en.wikipedia.org/wiki/Batch_processing。

[4] 译者注: 在栈和队列中, 返回和删除这两个操作也有可能是分开的, 例如STL的功能设计。

- **队列**(queue) —— 支持按“先进先出”(FIFO)次序取出。在服务场景下按FIFO次序控制等待时间无疑是最公正的方案。若要最小化最长等待时间, 你就需要队列这种容器来保存作业(job), 从而让它们按FIFO次序处理。注意, 无论采用FIFO还是LIFO次序, 作业的平均等待时间都是相同的。不过, 许多计算领域中的应用问题中都存在着一些能够无限等待的作业, 在这些情况下关于最长等待时间的讨论没有太大意义。
  相比栈而言, 实现队列需要几分技巧, 如果在实际应用(如某些仿真)中维护时序的问题确实比较重要, 那么此时选用队列是最合适的。对队列的放入和取出操作通常称为**入队**(Enqueue)和**出队**(Dequeue)。
  - Enqueue(x, q): 在队列q的尾部插入x。
  - Dequeue(q): 返回(并删除)队列q位于头部的数据项。

  稍后在图的广度优先搜索相关章节中, 我们将看到队列将成为控制该过程的基本抽象数据类型。

利用数组或链表都可以实现栈和队列并完成各项功能。至于到底选哪个, 关键问题在于我们事先是否了解待装入容器元素个数的上限, 如果知道这个限值的话则可使用静态分配的数组。

3.3 字典

字典这种抽象数据类型允许按内容访问数据项。你将某个数据项放入(stick)字典, 而需要时则能在字典中查到它。字典所支持主要的操作有:

- Search(D, k) —— 任给某个待查键k, 如果字典D中存在键值为k的元素, 则返回指向该元素的指针。
- Insert(D, x) —— 将数据项x加入字典D中。[1]
- Delete(D, p) —— 利用指向字典D中某元素x的指针p, 将x从D中删除。[2]

有些字典还能高效地支持若干其他操作,[3] 它们都很有用(例如利用这些功能即可按序关系迭代访问此类抽象数据类型中的全部元素):

- Maximum(D)或Minimum(D)[4]—— 在字典D中检索具有最大(或最小)键值的项。具备此功能的字典可充当优先级队列, 关于此内容将在3.5节中讨论。
- Predecessor(D, p)或Successor(D, p) —— 设指针p指向字典D中某个键为k的元素,[5] 本操作可在字典D中检索键值在序关系下刚刚小于(或刚刚大于)k的元素。

[1] 译者注: 应特别注意字典脱胎于集合, 插入一个已存在的元素意味着无需任何变动。

[2] 译者注: 此处略有修改, 指针变量以p表示更为清晰。此外, 本节中字典中操作的返回值一般均为指针。

[3] 译者注: 很多教材中将这种功能更齐全的抽象数据类型称为“集合”(其实“全序集”一词更准确)。请注意, 全序集在任意元素之间定义了序关系。

[4] 译者注: 原文采用Max(D)和Min(D), 为与后文中出现的操作名一致, 此处作相应更正。

[5] 译者注: 此处略有修改, 需假设p所指为字典中的元素, 以便和后面的字典实现一致。当然, 直接使用键值k获得Predecessor(D, k)和Successor(D, k)更复杂一些, 因为k是否存于字典中尚未知。

利用这些字典操作可以完成许多常见的数据处理任务。举例来说, 假定我们想去掉邮寄列表(mailing list)中所有重复的名字, 然后再按某种次序打印剩下的名字。我们可以这样解决: 首先初始化一个空字典D, 并以名字作为记录查找的键。随后依次读入邮寄列表中各项, 对读入的任意一个记录, 通过Search操作查找该记录所包含的名字是否已存在于D中: 若不存在则利用Insert操作将此记录插入D中。一旦处理完邮寄列表中的所有项, 我们就可以将字典中的现有元素全部导出: 从第一项Minimum(D)开始, 再不断调用Successor操作找到下一个元素, 直到我们遇到最后一项Maximum(D)为止, 如此我们便按次序对所有元素遍历完毕。

通过抽象的字典操作来描述问题, 可避免纠缠于采用数据结构表达而带来的烦琐细节, 从而将注意力集中到亟待处理的问题上。

我们将在本节的余下部分详细考察一些简单的字典实现方案, 其数据结构基于数组或链表。在实际问题中, 一些功能更强的字典实现方式同样是很有吸引力的可选方案, 比如二叉查找树(见3.4节)和散列表(见3.7节)。关于实现字典的各式数据结构的全面讨论将在15.1节中给出。我们强烈建议读者去检索卷Ⅱ中的各种数据结构, 这样会对你的所有待选方案有一个更好的认识和了解。

停下来想想**:** 比较字典的各种实现**(I)**

问题: 考虑7种字典的基本操作: 查找(Search)、插入(Insert)、删除(Delete)、定序后邻(Successor)、定序前邻(Predecessor)、最小元(Minimum)、最大元(Maximum)。[1] 当字典借助下列数据结构实现时, 这些操作在最坏情况下的渐近运行时间分别是什么量级?

- **无序数组**(unsorted array)。
- **有序数组**(sorted array)。

解: 此问题(与下一问题)揭示了数据结构设计中某些固有的平衡。假设数据以某种形式组织, 我们虽然可在该结构上更高效地实现抽象数据类型的某些操作, 但这却会以降低另一些操作的效率为代价。

除了手里的这个数组之外, 假设我们还能够使用一些额外变量, 如数组中目前实有元素的个数n。注意, 为此我们必须在那些会改变数组的操作(如插入和删除)中维护这些额外变量的值, 并将维护费用计入相应的操作其开销之中。

事实上, 无序数组和有序数组完全能够实现字典, 而各个基本操作[2]的开销分别为(标记星号的项需要略作思考):

字典操作	无序数组	有序数组
Search(A, k)	$O(n)$	$O(\log n)$
Insert(A, x)	$O(1)$	$O(n)$
Delete(A, i)	$O(1)^*$	$O(n)$
Successor(A, i)	$O(n)$	$O(1)$

[1] 译者注: 所谓“定序后邻”和“定序前邻”指的是在序关系下的前邻和后邻, 这是为了区分链表中的(物理)前邻和(物理)后邻, 其具体定义在前文已给出。

[2] 译者注: 我们对各操作略作修改以保持一致, 字典名为A, i代表数组下标, 返回值也为下标。后续实现时分别以U和S表示不同的数组, 且下标从0开始。

字典操作	无序数组	有序数组
Predecessor(A, i)	$O(n)$	$O(1)$
Minimum(A)	$O(n)$	$O(1)$
Maximum(A)	$O(n)$	$O(1)$

我们必须了解实现每个操作的具体过程, 才能明白其开销何以如是。我们先讨论维护*无序数组*(不妨记为U)情况下的各种操作。

- 实现查找操作是以键k与无序数组中每个元素的键进行对比, 检验到值为k的元素则终止(有可能与所有元素都比对过)。因此, 查找操作在最坏情况(在U中找不到键k)下需要耗费线性时间。
- 实现插入操作只需将x复制到数组中的`U`$[n]$位置, 再将实有元素个数n增加1即可。由于插入操作只改变少数几个元素而已, 因此该操作只需常数时间。
- 删除操作需要一点技巧, 因此在上表中已用*加以标注。根据删除操作的定义, 我们已知待删除元素的下标i, 因此不再需要花时间寻找该元素。而在数组U中删除下标为i的元素会出现一个空位, 我们必须填补。一种思路是让从$U[i+1]$到$U[n-1]$的所有元素依次向前挪动一位, 不过当删除的是数组首元素时, 该方案需要用$\Theta(n)$时间。而另一种思路则更好: 只需将$U[n-1]$的值写到`U`$[i]$中, 再将实有元素个数n减少1即可, 这种方案只需常数时间。
- 从遍历操作的定义可知, 定序前邻/定序后邻操作的返回值是指在序关系下`U`$[i]$的前一元素/后一元素的下标, 而不能想当然地误以为是i `-` `1`和i `+` `1`, 因为无序数组中物理意义(即存储位置)下的前邻/后邻并不是逻辑意义(即序关系)下的前邻/后邻。事实上, $U[i]$的定序前邻对应着小于`U`$[i]$那些元素中的最大元。与此类似, $U[i]$的定序后邻对应着大于$U[i]$那些元素中的最小元。这两种操作都需要搜遍U中所有元素才能让此类元素最终胜出。
- 最小元(最大元也类似)在序关系下定义, 因此需要以线性时间扫描整个无序数组才能找到。如果我们单独设置额外变量指示最小元和最大元, 便能在$O(1)$时间予以报表。这个想法似乎很诱人, 但却再也无法保证常数时间的删除操作, 因为如果所删除的数据项是最小元, 这套约定会迫使你花费线性时间去找到新的最小元。

以*有序数组*(不妨记为S)实现字典完全颠倒了我们在无序数组中关于各种操作难易程度的看法。现在利用二分查找算法可实现$O(\log n)$时间的查找操作, 原因在于有序数组的中位数必然位于$S[n/2]$。由于有序数组的前半部分和后半部分仍为有序数组, 只要根据大小关系正确选出待查元素所处的那一部分, 查找操作即可不断递归执行下去。由于每次元素个数会减半, 从n个元素到仅有1个元素最多只需$\lceil \log n \rceil$次即可。

这种元素有序排列的方式也能让我们在处理字典的其他检索操作时获益。最小元和最大元位于`S[0]`@和$S[n-1]$, 而$S[i]$的定序前邻和定序后邻则对应着$S[i-1]$和$S[i+1]$。

然而, 插入和删除变得更为费时, 因为插入时要给新元素腾地方, 而删除时要填补空位, 这些都可能需要搬迁许多元素。因此, 插入和删除都将变为线性时间操作。 ∎

领悟要义: 数据结构的设计必须平衡它所支持的所有各种操作。同时支持甲操作和乙操作的最快数据结构, 往往并不是仅支持甲操作/乙操作的那些数据结构中最快的。

停下来想想: 比较字典的各种实现(II)

问题: 当字典借助下列数据结构实现时, 7种字典的基本操作在最坏情况下的渐近运行时间分别是什么量级?

- **无序单链表**(singly-linked unsorted list)。
- **无序双链表**(doubly-linked unsorted list)。
- **有序单链表**(singly-linked unsorted list)。
- **有序双链表**(doubly-linked sorted list)。

解: 评估这些实现方案需要考虑两类对比问题: 单链与双链; 有序与无序。下表列出用不同数据结构实现的操作开销, 某些较难理解的以*标记。[1]

字典操作	无序单链表	无序双链表	有序单链表	有序双链表
Search(L, k)	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Insert(L, x)	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Delete(L, p)	$O(n)^*$	$O(1)$	$O(n)^*$	$O(1)$
Successor(L, p)	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Predecessor(L, p)	$O(n)$	$O(n)$	$O(n)^*$	$O(1)$
Minimum(L)	$O(1)^*$	$O(n)$	$O(1)$	$O(1)$
Maximum(L)	$O(1)^*$	$O(n)$	$O(1)^*$	$O(1)$

同无序数组一样, 无序链表中的查找操作运行注定会很慢, 而维护性的操作却较快。

- **插入/删除** —— 从单链表中删除元素是这些操作中最为复杂的。根据删除操作的定义, 我们已有指针p指向待删除元素所对应的结构体(记为d)。但是我们*真正需要*的指针(不妨称为p的"链表前邻")得指向链表中位于d这块数据之前的那个结构体(记为d'), 因为d'才是那个需要改变的结点。如果不知道p的链表前邻那就什么也做不了, 因此我们需要用线性时间在单链表中寻找d'。[2] 双链表则避免了此问题, 因为我们可通过结点指针域直接找到p的链表前邻。
  有序双链表中的删除比有序数组中的删除更快, 原因在于将待删除元素移出并黏合断开的部分比移动元素以填补空位的效率更高。由于前邻指针的问题, 有序单链表中的删除同样会更麻烦一些。

[1] 译者注: 我们对以下各操作和后文相应部分都略作修改以保持一致, 字典名为L, 指针变量为p。

[2] 实际上, *确实存在*一种方法可在常数时间内从单链表中删除某个元素。图3.2给出了示意: 设`x`为指针, 用`x->next`对应结点的全部内容覆盖`x`对应结点(包括数据域和指针域), 然后释放`x->next`之前所指向的结点(请提前保存指针值)。当然得特别注意边界情况: 如果`x`指向单链表中的第一个结点, 此时不必更改链首指针; 如果`x`指向单链表中的最后一个结点, 可在链表最后再放置一个持续守护的哨兵结点, 它总对应着单链表在物理意义下的最后一个结点。不过, 这将让我们无法在常数时间内完成最小元/最大元操作, 因为删除后不像以前那样还有线性时间去寻找新的最小元/最大元。

- **查找** —— 相比有序数组而言, 链表中的元素有序排列所能带来的好处不多。二分查找不再可行, 这是因为在有序链表中如果不去遍历中位数之前的所有元素, 则无法找到它的位置。不过, 有序链表*真正*带来的好处是能够迅速终止不成功的查找, 可举例说明其原因: 假定我们所寻找的"Abbott"一直未能找到, 而一旦发现"Costello", 我们便可推断出此有序链表中必然不存在"Abbott"。不过, 最坏情况下的查找操作依然要花费线性时间。[1]
- **遍历型操作** —— 链表前邻指针问题再一次让定序前邻操作的实现复杂化。不过在以上这两类有序列表中, 结点在逻辑上的后邻都等价于结点指针域中的后邻, 因此定序后邻操作可在常数时间内实现。
- **最大元** —— 最小元位于有序链表头部, 其访问非常容易。而最大元位于有序链表尾部, 因此无论是有序单链表还是有序双链表, 通常都要$\Theta(n)$时间方可到达此处。然而, 我们可以设置一个单独指向有序链表尾部的指针`last`, 不过必须在插入和删除操作中额外花时间维护它。双链表可在常数时间内更新这个尾指针: 在插入时, 检查`last->next`是否仍为`NULL`; 在删除时, 若链表尾部元素被删, 则令`last`指向删除前`last`的链表前邻。

  事实上, 我们无法找到一种能在单链表中高效寻找结点链表前邻的方法。那么问题来了, 究竟怎样才能在$\Theta(1)$时间内实现有序单链表的最大元操作呢?[2] 其诀窍在于让删除操作承担此部分的开销。在*本身就要*用线性时间的删除操作中额外再加一次搜遍整个链表以更新尾指针的过程, 这不会影响删除操作的渐近复杂度。只要你想明白了这一点就能拿到奖励——常数时间的最大元操作。[3] ∎

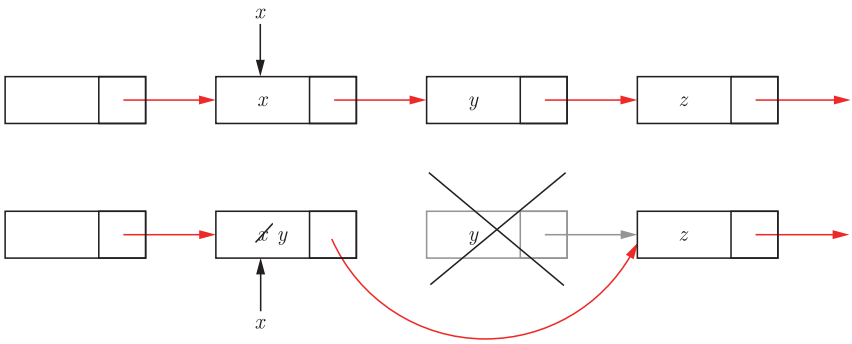

图 3.2 删除单链表某结点而无需获取其链表前邻(覆盖待删除结点内容并释放它之前的链表后邻结点)

3.4 二叉查找树

我们已见过若干种数据结构, 其中一些允许快速查找, 另一些允许灵活更新, 但没有一个能*二者兼备*。无序双链表支持$O(1)$时间的插入和删除, 但其查找在最坏情况下需要线性

[1] 译者注: Abbott和Costello是美国著名的谐星搭档, 不妨移步http://www.abbottandcostello.net/。

[2] 译者注: 实际上, 无论是无序单链表和有序单链表都可以实现。不过, 有序单链表如果删除尾部元素, 在删除之前利用尾部元素的前邻直接更新尾指针即可。

[3] 译者注: 这是作者的幽默, 其实想不明白也会有的。

时间。有序数组支持二分查找, 从而获得了对数时间的查询, 但这必须以线性时间的更新为代价。

二分查找要求我们能迅速访问*两个元素*——分别是小于/大于指定结点键值对应键集的中位数。为将上述思想予以融合, 我们需要拓展之前的链表结构, 让它的每个结点都包含两个指针。这就是藏于二叉查找树背后的基本思想。

一棵**有根二叉树**(rooted binary tree)可作如下递归定义: 要么它为空; 要么它含有一个称为*根*的结点, 除此之外还包含两棵有根二叉树, 分别称为左子树和右子树。有根树中"兄弟"结点之间的次序不可忽视, 因此左和右是有区别的。图3.3给出了由3个结点所形成的有根二叉树的5种不同形状。

二叉查找树(binary search tree)会给二叉树中每个结点都加上一个唯一的键标签, 使得对于任意结点(不妨设键值为k), 其左子树中所有结点刚好包含小于k的那些键值, 而其右子树中所有结点刚好包含大于k的那些键值。[1] 查找树的这种标签机制非常特别。对于任意n个键形成的集合, 若以这些键对一棵具有n个结点且形状确定的二叉树添加标签, 那么恰好仅存在一种标签方案能使该树成为二叉查找树。图3.3对3个结点所能形成的树分别给出了可行标签方案。

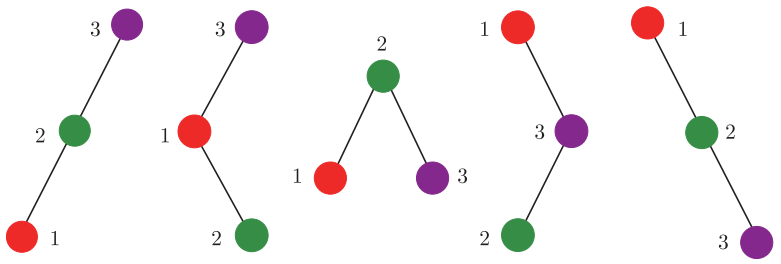

图 3.3 5棵具有3个结点的不同二叉查找树(任意结点t其左/右子树的结点键值小于/大于t的键值)

3.4.1 实现二叉查找树

二叉树的结点具有左孩子指针域、右孩子指针域和一个数据域, 有时还可能设置父亲指针域, 它们的关系如图3.4所示。我们先给出二叉树结构的类型声明:

```
typedef struct tree {
    item_type item;          /* 数据项 */
    struct tree *parent;     /* 指向父亲的指针 */
    struct tree *left;       /* 指向左孩子的指针 */
    struct tree *right;      /* 指向右孩子的指针 */
} tree;
```

查找、遍历、插入和删除是二叉树所支持的基本操作, 下面逐一讨论。

[1] 如果允许在二叉查找树(或其他字典结构)中直接存放重复的键值其实很不好, 常常会导致一些不易察觉的错误。要想支持重复数据项的处理, 更好的方案是给每个结点再额外增加一个指针, 并显式地维护一个列表去存放与当前结点键值相同的所有数据项。

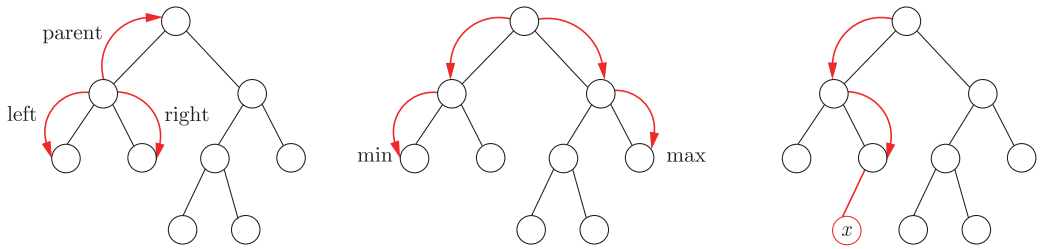

图 3.4 [左] 二叉查找树中的结点关系(父子和兄弟)示意 [中] 在二叉查找树中寻找最小元/最大元 [右] 在二叉查找树中的合理位置插入新结点(存放数据项x)

1. 树的查找

二叉查找树标签方案会确定每个键的所在位置, 而这些位置都是唯一的。从根开始查找, 若当前结点包含待查键x则停止, 否则根据x与当前结点的键值对比情况(小于或大于)而转到其左孩子或右孩子结点。[1] 由于二叉查找树的左子树和右子树自身也都是二叉查找树, 因此上述算法是正确的。这种递归结构引出了下面的递归查找算法:

```
tree *search_tree(tree *l, item_type x)
{
    if (l == NULL)
        return NULL;

    if (l->item == x)
        return l;

    if (x < l->item)
        return search_tree(l->left, x);
    else
        return search_tree(l->right, x);
}
```

以上查找算法能在$O(h)$时间内执行完毕, 其中h代表树的高度。

2. 在树中寻找最小元和最大元

由定义可知, 最小键必然居于根的左子树, 原因是左子树中所有结点囊括了所有小于根结点键值的键。因此, 最小元必然是根的最左子孙, 与此类似, 最大元必然是根的最右子孙。图3.4[中]给出了最小元和最大元的位置示意。

```
tree *find_minimum(tree *t)
{
    tree *min;                      /* 指向最小元的指针 */

    if (t == NULL)
        return NULL;

    min = t;
```

[1] 译者注: 为了简化二叉查找树的实现, 作者在后续代码中直接将数据项x本身作为其键值, 也即不区分数据项和对应键值。

```
    while (min->left != NULL)
        min = min->left;
    return min;
}
```

3. 树的遍历

事实证明, 访问有根二叉树中所有结点是许多算法中的一个重要组成部分。树的遍历是遍历图中所有结点与边问题的一个特例, 而图的遍历问题则是第7章的基石。

树的遍历其主要应用就是列出树中所有结点的标签。事实上, 用二叉查找树能够很容易地将所有标签以*有序形式*报表输出。由定义可知, 所有小于根结点键值的那些键必位于根的左子树中, 而所有大于根结点键值的那些键必位于根的右子树中。因此, 只需遵循此原则递归访问结点, 即可得到查找树的**中序遍历**(in-order traversal):

```
void traverse_tree(tree *l)
{
    if (l != NULL)
    {
        traverse_tree(l->left);
        process_item(l->item);
        traverse_tree(l->right);
    }
}
```

在遍历过程中每个结点仅被处理一次, 因此遍历操作需要$O(n)$时间, 其中n代表树中结点的个数。

只需改变`process_item`相对于访问左子树过程和访问右子树过程的调用位置, 便能得到一些其他遍历次序。最先执行`process_item`会得到**前序遍历**(pre-order traversal), 而最后执行`process_item`则会得到**后序遍历**(post-order traversal)。尽管相对于中序遍历而言, 前序遍历和后序遍历对于查找树来说几乎没有意义, 但在处理以有根树表示的算术或逻辑表达式时, 这两种遍历方案才会体现自己的不可或缺性。

4. 树的插入

为了保证在二叉查找树中插入数据x后还能再利用查找操作找到它, 那么x的插入位置只能有一个, 即我们查询x失败时的所处位置(肯定对应`NULL`), 因此要插入x必须替换这个取值为`NULL`的指针。

以下实现方案(`insert_tree`函数)通过递归将查找键值与插入结点融为一体, 从而可完成键的插入。该函数的3个变元是: (1) 指针`l`, 它所指向的指针对应待查找子树和原二叉树余下部分的交汇点; (2) 待插入数据`x`(为简单起见, 假设其键值也是`x`); (3) 指针`parent`, 它指向`*l`的父亲结点。若遇到`NULL`, 则为`x`分配结点并设定其链接。注意我们所传递的变元类型是指向指针的指针, 也即`l`能确定在查找过程中究竟转到了当前结点的左孩子还是右孩子, 从而保证赋值语句`*l = p;`可将新结点链接到树中恰当的位置。

```
void insert_tree(tree **l, item_type x, tree *parent)
{
    tree *p;                            /* 暂用指针 */
```

```
    if (*l == NULL) {
        p = malloc(sizeof(tree));
        p->item = x;
        p->left = p->right = NULL;
        p->parent = parent;
        *l = p;
        return;
    }

    if (x < (*l)->item)
        insert_tree(&((*l)->left), x, *l);
    else
        insert_tree(&((*l)->right), x, *l);
}
```

前期查找位置的工作可在$O(h)$时间内完成, 而随后的新结点分配以及将其链接到树中都是常数时间操作。因此, 插入需$O(h)$时间, 其中h代表树的高度。

5. 树的删除

删除比插入更富于技巧性, 原因在于删除某一结点意味着需将该结点子孙所对应的两棵子树恰当地链回树中其他位置。一般存在三种情况, 图3.5已分别举例说明, 请仔细观察该图。下面给出处理方案:

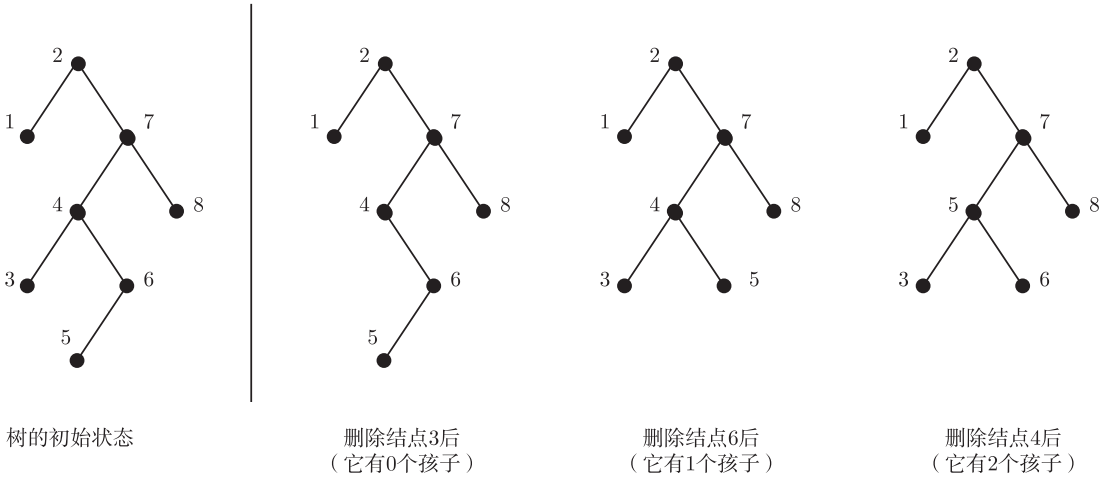

图 3.5 对具有0个、1个和2个孩子的树结点进行删除

- 叶子结点没有孩子, 因此删除叶子结点只需简单地清理指向它的指针即可。
- 待删除结点只有一个孩子的情况也很简单明了。删除后只剩下该结点的父亲和孩子, 而这位父亲只需链接上它的孙子结点即可, 而且还不会破坏树的中序标签特性。

- 若待删除结点有两个孩子, 又该如何处理呢? 我们的解决方案是用待删除结点的定序后邻[1]的键值对其重设标签。这个定序后邻结点的键值必然是待删除结点右子树中最小的, 确切地说就是其右子树的最左子孙。只需将该结点的键值写到删除点, 即可获得具有正确标签的二叉查找树, 这样即可将此类结点删除问题简化成在物理上至多只有一个孩子的删除问题, 而该问题在上文中已解决。

此处略去删除结点的完整实现, 其原因是它有点令人望而生畏, 不过读者可按前文所描述的步骤自行给出具体代码。

最坏情况下的复杂度可分析如下: 设树高为h, 每次删除至多要进行两遍查找操作, 每遍需$O(h)$时间, 再辅以若干次(常数量级)指针操作即可, 因此删除需$O(h)$时间。

3.4.2 二叉查找树究竟能有多好

使用二叉查找树实现字典时, 字典的3种操作均只需$O(h)$时间, 其中h为树高。当树的形状相当平衡时, 树会达到最低高度, 即$h = \lceil \log n \rceil$, 其中n为树中结点数, 而这也是理论上能获得的最好结果。树能够具备最低高度固然很好, 但此时的树必须是极致的平衡形态。

我们的插入算法将每个新项放到某个新生成的叶子结点中, 而它原本就该在这个位置被查找操作发现。[2] 随之而来的是, 树的形态将会随着插入次序的变化而不同, 而更重要的则是对树高的影响。

不幸的是, 通过插入来建立一棵树可能会出现一些很糟糕的情况, 而以上数据结构自身却无法控制插入的次序。可考虑某用户插入有序键值序列的实例: 先用`insert`(a)在树中插入a, 再通过`insert`继续插入b, c, $d \cdots\cdots$ 这样将产生一棵极瘦的树, 其中每个结点只有右孩子, 且树的高度等于结点数。[3]

综上可知, 包含n个结点的二叉树其高度可能在$\log n$到n之间。二叉树的平均高度又将是什么样的呢? 该算法的平均情况分析很难开展, 其原因在于我们必须仔细说明究竟是哪种“平均”。如果假定所有$n!$种插入次序发生的可能性相等, 则此问题将非常明确。果真如此的话, 那么我们真是太走运了, 因为这样所生成的树以高概率具备$O(\log n)$的高度, 4.6节将对此结论给出一个直观的证明。

事实上, 这是体现*随机化方法*功效的一个有力论据。我们经常可以开发一些能以高概率提供优异性能但形式却很简单的算法, 我们还将看到与此类似的思想支撑了最快的排序算法, 也即著名的快速排序。

3.4.3 平衡查找树

随机查找树通常都还不错。但如果我们不幸地碰到上文提到的那种插入次序, 那么最终仍会在最坏情况下得到一棵线性高度[4]的树而告终。这种最坏情况超出了我们的掌控范

[1] 译者注: 原文使用的是immediate successor, 即排序后位于待删除结点x之后的那个结点, 可以想象成序关系下位于x之后且离x最近的结点。不过, 这个概念和前文讨论集合时所用的successor完全一致。

[2] 译者注: 其言外之意是按照查找算法所停止的位置进行插入。

[3] 译者注: 请注意作者所采用树的高度定义。例如图3.5中从左到右的树高分别为5、5、4、4。

[4] 译者注: 即树高为$h = O(n)$, 其中n为结点个数。在下文中若不特别说明, 衡量树高h均以关于n的函数表示。

围, 因为树得按用户要求而构建, 而实际中可能有一些讨厌的用户会给出糟糕的插入次序。

我们若在每棵树的插入/删除操作后略作调整, 则会让情况好转, 即树会尽可能接近平衡进而保证最大高度为对数量级。事实上, 已经有了这样一种精巧的数据结构能保证树高始终为$O(\log n)$, 它就是**平衡二叉查找树**(balanced binary search tree)。因此字典的所有操作(插入、删除和查询)都只需$O(\log n)$时间。平衡二叉查找树的实现方法, 如红黑树(red-black tree)和伸展树(splay tree), 将在15.1节中讨论。

从算法设计的视角来看, 我们非常有必要了解这些树的存在, 更要知道它们能以黑盒的形式高效地实现字典。在分析算法时, 如果要考虑字典操作的开销, 我们一般以平衡二叉查找树在最坏情况下的复杂度为准, 而这样做也是比较合理的。

领悟要义: 就性能而言, 选择了错误的数据结构去完成任务可能会相当糟糕。不过, 在若干较好的数据结构中确定一个最佳者却往往不必太过费心, 因为通常会有好几个选项, 而其性能基本上也差不多。

停下来想想: 利用平衡查找树

问题: 完成读入n个数再按序依次打印的任务。假定我们能使用基于平衡查找树实现的字典, 它支持查找、插入、删除、最小元、最大元、定序后邻和定序前邻操作, 且每个操作都只需对数时间。

(1) 能否仅用插入和中序遍历操作实现$O(n \log n)$时间的排序?

(2) 能否仅用最小元、定序后邻和插入操作实现$O(n \log n)$时间的排序?

(3) 能否仅用最小元、插入和删除操作实现$O(n \log n)$时间的排序?

解: 每个用二叉查找树来排序的算法都必须从构建一棵实实在在的树开始。假设所用字典为T,[1] 我们需要先对树进行初始化(其实也就是设置`NULL`指针), 随后读取并插入数据到T中(共n项)。算法耗时为$O(n \log n)$, 因为每次插入顶多花费$O(\log n)$时间。非常有意思的是, 光是建立数据结构这一步, 就能定出此类排序算法的运行时间量级! 算法思路如下:

- 问题(1)允许使用插入和中序遍历操作。可通过插入所有n个元素以建立查找树, 再使用遍历操作即可按序依次访问全部元素, 详见算法16。
- 问题(2)允许在树建成后使用最小元和定序后邻操作。可先找到最小元, 再不断寻找定序后邻元素即可按序依次遍历所有元素, 详见算法17。
- 问题(3)没有给出定序后邻操作, 但允许使用删除操作。可不断寻找当前情况下的最小元, 再将其从树中删除, 这样还是可以按序依次遍历所有元素, 详见算法18。

[1] 译者注: 字典的相关操作可参阅前文, 此外我们调整了原文的算法描述以保持前后文的一致性。

算法 16 Sort1()

```
1  初始化T
2  while (未读完) do
3  |   读取x
4  |   Insert(T, x)
5  end
6  遍历T
```

算法 17 Sort2()

```
1  初始化T
2  while (未读完) do
3  |   读取x
4  |   Insert(T, x)
5  end
6  y ← Minimum(T)
7  while (y ≠ NULL) do
8  |   打印y指向的数据
9  |   y ← Successor(T, y)
10 end
```

算法 18 Sort3()

```
1  初始化T
2  while (未读完) do
3  |   读取x
4  |   Insert(T, x)
5  end
6  y ← Minimum(T)
7  while (y ≠ NULL) do
8  |   打印y指向的数据
9  |   Delete(T, y)
10 |   y ← Minimum(T)
11 end
```

上述每一种算法都执行了线性次数的对数时间操作, 因此它们都可在$O(n \log n)$时间内执行完。可以看出, 利用平衡二叉查找树设计算法的关键在于将其视为黑盒。[1] ∎

3.5 优先级队列

许多算法需要按特定次序处理项目。例如, 假定你要根据作业的相对重要程度进行作业调度。容易想到, 调度这些作业需要按重要程度对其排序, 随后按序逐个处理。

相对于简单地将元素排序而言, 使用**优先级队列**(priority queue)这种抽象数据类型能提供更多的灵活性, 也即它允许新元素在任意时段进入系统。实际上, 优先级队列处理此情况的开销相比于对元素重新排序要少得多。

优先级队列主要支持下列3种基本操作:

- Insert(Q, x) —— 给定x, 根据其键值情况将x插入至优先级队列Q中。
- Find-Minimum(Q)或Find-Maximum(Q) —— 返回优先级队列中具有最小(最大)键值的元素指针。
- Delete-Minimum(Q)或Delete-Maximum(Q) —— 删除优先级队列中键值最小(最大)元素。

许多现实中的事件产生过程其本质上是通过优先级队列来建模的。单身人士(姑且称为S)心中有一个约会候选人的优先级队列, 尽管他/她内心深处并不那么清楚地了解这一点。如果S在聚会上认识了一位新朋友, 其实都会打出印象分, 它衡量了这位朋友的吸引力或与S的合意程度。尔后S将新朋友加入“秘密笔记”(little black book)优先级队列中, 而其印象分则会作为该项的键值。那么约会过程就是: 从该数据结构中选出最中意的朋友, 再花上一个夜晚互相深入了解, 便可估计出与这位朋友再次约会的可能性, 随后依照新的印象分再将其放回优先级队列中。

[1] 译者注: 严格来说, 此处应该使用多重集合, 因为可能会有重复的键值出现。

领悟要义: 围绕字典和优先级队列这两种抽象数据类型来构建算法, 不但会让算法结构清晰, 还能取得优异的性能。

停下来想想: 优先级队列的几种简单实现方案

问题: 当优先级队列以下列数据结构实现时, 3种基本的优先级队列操作(插入、寻找最小元和删除最小元)在最坏情况下的渐近运行时间分别是什么量级?

- 无序数组。
- 有序数组。
- 平衡二叉查找树。

解: 实现这3种基本操作时, 其实你会发现有一些微妙的细节完全出乎意料, 即便使用无序数组这样的简单数据结构也是如此。从前文可知, 通过无序数组而构造的字典可实现常数时间的插入和删除, 而查找和最小元则需线性时间。线性时间的删除最小元操作由以下两步组合而成: 先调用Find-Minimum寻找最小元, 再用无序数组的Delete操作删除该元素。

对有序数组而言, 我们可实现线性时间的插入和删除, 且寻找最小元仅需常数时间。然而, 所有的优先级队列其删除操作仅涉及最小元。如果按逆序存储数组(最大元在首位), 则最小元将永远位于数组末尾。只要简单地对数组内元素的个数n进行减1操作, 即可实现删除, 并且还不用移动任何元素, 因此删除操作可在常数时间内执行完毕。

所有问题皆已清楚, 然而下表却宣称其中所有数据结构能实现常数时间的寻找最小元操作:

操作	无序数组	有序数组	平衡二叉查找树
Insert(Q, x)	$O(1)$	$O(n)$	$O(\log n)$
Find-Minimum(Q)	$O(1)$	$O(1)$	$O(1)$
Delete-Minimum(Q)	$O(n)$	$O(1)$	$O(\log n)$

实现常数时间的寻找最小元操作的技巧在于额外再用一个变量存储其指针/下标, 因此, 无论何时我们需要寻找最小元, 只需返回该指针值即可。在插入中更新此指针不难: 当且仅当插入值小于它时才更新。而在删除最小元操作中又该怎么办呢? 我们先清除现有的最小元指针, 再去重新找一次最小元并让该指针指向它, 待其他操作需要时即可直接使用这个预先设置好的(canned)[1]指针。在无序数组和平衡二叉查找树中寻找新的最小元分别需要线性时间和对数时间, 均与对应删除操作运行时间的量级完全一致, 因此可将其融入删除操作而不改变其渐近时间。 ∎

优先级队列是非常有用的抽象数据类型。事实上, 这些形式各异的优先级队列将成为本章中两部*算法征战逸事*的主角。4.3节中会从排序问题切入, 并讨论一种特别精妙的优先级队列实现——堆。此外, 卷Ⅱ中的15.2节将展示一整套各式各样的优先级队列实现方案。

[1] 译者注: 从情景喜剧中的"canned laughter"(罐头笑声或背景笑声)引申而来, 这是预先录制好的笑声。

3.6 算法征战逸事: 剥离三角剖分

计算机图形学中所用的几何模型通常表现为一种由三角形剖出的曲面[1](triangulated surface), 如图3.6[左]所示。高性能引擎拥有专门的硬件设备用于对三角形进行渲染和描影。这种硬件设备计算速度非常之快, 以至于将三角形结构投入硬件引擎这部分开销[2]变成了渲染操作的瓶颈。

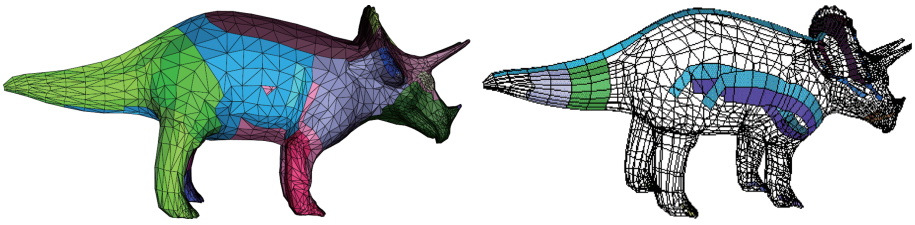

图 3.6 [左] 恐龙的三角剖分模型(triangulated model) [右] 模型中的若干三角形带

尽管每个三角形可通过详细指明它的3个端点以描述之, 但还有更高效的方法可供选择。假定我们将这些三角形划分成若干条由相邻三角形构成的**带**(strip)并沿带走动来处理三角形, 而不是去孤立地考虑每个三角形。由于每个三角形与其相邻三角形都有两个公共顶点, 我们节约了重传两个冗余顶点以及与这两个顶点相关的其他信息所带来的开销。为使这些三角形的表述清楚明确, OpenGL三角形网格渲染器假定走动均按朝左和朝右这样交替转向。图3.7[左]的三角形带其顶点序列为(1, 2, 3, 4, 5, 7, 6), 它可以非常清晰地描述网格中的5个三角形, 还展示了左/右交替转向模式。图3.7[右]的三角形带给出了网格中的全部7个三角形, 而转向方式则是(1, 2, 3, 左-4, 右-5, 左-7, 右-6, 右-1, 右-3)。[3]

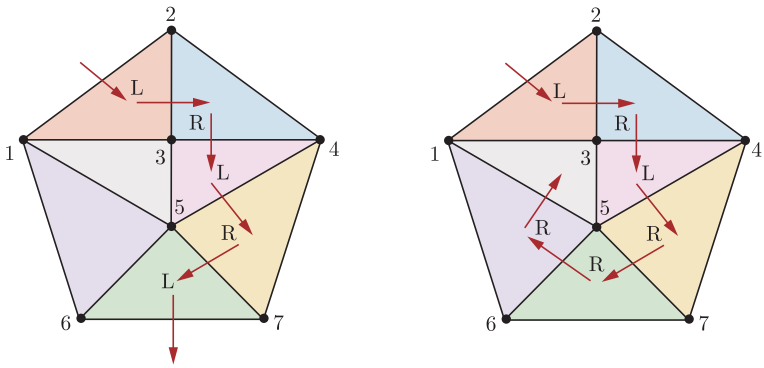

图 3.7 将三角形网格划分为带: 左右交替转向方式(本例中只能部分覆盖)[左]和随意转向模式(充分发挥灵活性从而完全覆盖)[右]

[1] 译者注: 可译为三角剖分曲面或三角曲面, 此类曲面由三角形组成, 即用非常小的三角形剖分原有的真实曲面而得到。曲面中的三角形一般称为三角面片, 为了不引入过多的专有名词, 下文一律仍译为三角形。

[2] 译者注: 作者用很形象的语言描述了硬件引擎读入三角形结构的行为, 可想象在投币机中投入硬币要费去一定的时间。

[3] 译者注: 以顶点子序列(2, 3, 4)为例, 当前三角形由其组成, 注意它与上一个三角形的邻边为(2, 3)。以(2, 3)为底边右转则会选到(3, 4)作为新的邻边, 并得到新顶点5, 因此会用"右-5"表述, 而下一次处理的三角形(或顶点子序列)则为(3, 4, 5)。