

序 言

本书文本及配套代码的开源网站：<https://github.com/ClazyChen/ds-lab>。

在计算机学科研究生考试中，为了考取办学层次更高的学校，学生们往往需要付出额外的努力。此时，如果只使用市面上常见的计算机考研辅导丛书（简称为辅导书），可能达不到足够深刻的理解程度。通常解决这一问题的方案有三种：

1. 阅读一些更加专业的书籍。比如说，许多同学通过 Patterson 的《软件 / 硬件接口》^[1]一书辅助自己学习《组成原理》。
2. 进行一些较为复杂的实验。比如说，许多同学通过清华大学陈渝老师课题组设计的 rCore 实验^{[2][3]}辅助自己学习《操作系统》。
3. 报名一些高水平的培训班，或者寻找高水平的师姐 / 师兄辅导。显然，这相对来说比较昂贵。

本书试图为准备参加研究生考试的计算机系学生，尤其是打算报考“双一流”学校的学生，提供一个新的选项。本书将立足于现代 C++^[4]来描述《数据结构》，它的内容涵盖了严蔚敏《数据结构》（简称为严书）^[5]和邓俊辉《数据结构》（简称为邓书）^[6]这两本经典教材，可以认为覆盖了研究生考试中《数据结构》的考试范围。下面简要地对严书、邓书、辅导书和本书进行一个对比。

在相当长的时间里，严书是学习《数据结构》的经典教材。目前，严书也是计算机统考考研（408）的主要参考教材。严书假定读者已经熟悉经典标准的 C 语言（C99），主要使用 C 语言风格的伪代码进行讲解。对于只了解 C 语言基础编程的读者来说，严书比较容易上手。

邓书可以认为是严书的新版本，它是清华大学计算机考研（912）的主要参考教材。邓书假定读者熟悉经典标准的 C++ 语言（C++98），但不涉及 C++ 中的标准模板库（STL）。邓书的容量显著大于严书，比如，它包含了伸展树、跳表等较为复杂的数据结构；同时也删除了一些不常使用的内容。邓书中以模板类的形式实现各个数据结构，C++ 语言的面对对象编程（OOP）特性有助于读者对数据结构有更深刻的了解。

面上的辅导书通常以严书作为底本，针对考研的现实需求，对教材中的知识体系进行了提炼。提炼后的知识体系往往非常枯燥，因此，通常，辅导书还有搭配的视频课程，把知识体系进一步细化讲解。同时，辅导书往往加入大量的习题，有助于学生及时检查和巩固自己的学习内容。辅导书往往更加专注于理论知识体系，因此，看完一本辅导书仍然无法实现快速排序、KMP 等基本算法，是比

较常见的事情。

表 1 本书和其他经典数据结构书籍的对比

	本书	邓书	严书	辅导书
知识量	多	中	少	少
知识串联	多	多	中	少
代码量	多	多	中	少
代码范式	面向对象	面向对象	面向过程	
编程语言	现代 C++	传统 C++	C 伪代码	
练习内容	配套实验	拓展阅读	习题	习题
考研参考	无限制	清华 912	统考 408	统考 408

本书和上述书籍的对比如表1所示。本书适用于对数据结构感兴趣的学生。本书利用丰富的代码和实验，为学习数据结构提供了一个不同于现有教材和辅导书的新视角。本书的实验体系相对邓书进行了精简化。对于每个数据结构，笔者会提供一个基类，这个基类中实现了许多学生不需要关心的代码。学生只需要实现那些需要学习的知识点的对应代码，而那些不需要关心的部分则是透明的。您也可以继承笔者提供的示例类，只改写其中的一个或几个方法，然后对自己实现的数据结构进行测试。如果学生愿意对照配套的实验代码学习其中的实验部分，能够对《数据结构》有更深刻的认识，并对现代 C++ 编程有一定了解。

本书允许学生只学习其中的理论部分，它的理论部分相当于一本没有习题的辅导书。不添加习题的主要原因是，编写习题过程中有极大可能出现错误。和市面上已有的辅导书相比，本书的理论部分也是视角独特的。本书将力求使您理解本学科考试范围内每一项技术的设计思想，能够在学会之后独立设计出书中的每一个数据结构和算法，最大程度减少学生备考的记忆量。对于不以“双一流”高校为目标的学生，笔者建议您跳过带 * 的部分。

本书的文本及其配套代码使用 MIT 许可证开源。作为一名清华大学研究生在读的学生，我深知对《数据结构》和现代 C++ 编程的理解比较有限，再次衷心希望读者为我提供宝贵的意见。

Clazy Chen

目 录

序 言.....	I
目 录.....	III
第 1 章 绪论	1
1.1 对象	1
1.2 数据结构	3
1.3 算法	5
1.3.1 算法的定义.....	5
1.3.2 算法的评价.....	6
1.3.3 实验：从 1 加到 N.....	7
1.4 正确性检验	10
1.4.1 经典的归纳和递降 *.....	11
1.4.2 带映射的归纳和递降 *.....	12
1.4.3 良序关系 *.....	13
1.4.4 字典序 *.....	13
1.4.5 超限的归纳和递降 *.....	14
1.4.6 实验：最大公因数.....	15
1.4.7 实验：数组求和.....	16
1.4.8 实验：函数零点.....	19
1.5 复杂度分析	20
1.5.1 复杂度记号的定义.....	21
1.5.2 复杂度记号的常见理解误区.....	22
1.5.3 实验：判断 2 的幂次.....	24
1.5.4 实验：快速幂.....	27
1.5.5 实验：复杂度的积分计算.....	28
1.5.6 多重循环复杂度的简单估算.....	29
1.6 本章小结	31
第 2 章 向量	32
2.1 线性表	32
2.2 向量的结构	34

2.3 循秩访问	35
2.4 向量的容量和规模	36
2.4.1 初始话	36
2.4.2 装填因子	38
2.4.3 改变容量和规模 *	39
2.4.4 扩容策略 *	40
2.4.5 等差扩容和等比扩容 *	41
2.4.6 分摊复杂度分析 *	42
2.5 插入、查找和删除	43
2.5.1 插入一个元素	44
2.5.2 平均复杂度分析	45
2.5.3 查找一个元素	46
2.5.4 删除一个元素	47
2.5.5 实验：插入连续元素	48
2.5.6 实验：向量合并	50
2.5.7 实验：按值删除元素	52
2.6 置乱和排序	56
2.6.1 实验：随机置乱 *	56
2.6.2 偏序和全序 *	58
2.6.3 自上而下的归并排序	60
2.6.4 自下而上的归并排序 *	63
2.6.5 基于比较的排序的时间复杂度	65
2.6.6 信息熵 *	66
2.6.7 有序性和逆序对 *	67
2.6.8 实验：先验条件下的归并排序 *	67
2.7 有序向量上的算法	69
2.7.1 折半查找	69
2.7.2 消除尾递归	71
2.7.3 实验：迭代形式的折半查找	72
2.7.4 实验：向量唯一化	73
2.8 实验：循环位移	76
2.9 本章小结	80

第 3 章 列表	81
3.1 列表的结构	81
3.2 列表的节点	81
3.2.1 单向列表节点	81
3.2.2 双向列表节点	83
3.2.3 哨兵节点	84
3.3 插入、查找和删除	85
3.3.1 后插一个元素	85
3.3.2 前插一个元素	87
3.3.3 查找一个元素	89
3.3.4 删除一个元素	89
3.3.5 实验：在头部和尾部连续插入元素	91
3.3.6 实验：顺序删除和随机删除 *	92
3.3.7 实验：倒置列表	94
3.4 列表的归并排序	98
3.4.1 基于值的归并排序	98
3.4.2 基于指针的归并排序 *	98
3.4.3 实验：归并排序的性能差异 *	100
3.5 循环列表	101
3.6 静态列表	102
3.6.1 静态列表的结构	102
3.6.2 静态列表上的节点	103
3.6.3 在静态列表上删除一个元素 *	104
3.7 本章小结	106
第 4 章 栈	107
4.1 栈的性质	107
4.1.1 栈的定义	107
4.1.2 栈的结构	108
4.1.3 后入先出	109
4.2 出栈序列	110
4.2.1 出栈序列的定义	110
4.2.2 出栈序列的计数	111
4.2.3 出栈序列的计数方法 *	111

4.2.4 随机出栈序列 *	112
4.3 括号序列	113
4.3.1 合法括号序列的计数.....	113
4.3.2 实验：判断括号序列是否合法.....	114
4.4 栈与表达式	115
4.4.1 表达式的定义.....	115
4.4.2 表达式中的元素.....	116
4.4.3 将字符串解析为中缀表达式.....	119
4.4.4 中缀表达式转换为后缀表达式.....	119
4.4.5 后缀表达式转换为中缀表达式.....	122
4.4.6 后缀表达式转换为前缀表达式.....	125
4.4.7 后缀表达式的计算.....	125
4.4.8 实验：中缀表达式的计算.....	126
4.5 栈与递归	127
4.5.1 实验：消除尾递归的扩展形式 *	127
4.5.2 实验：计算组合数 *	129
4.5.3 实验：消除一般的递归 *	132
4.6 栈的扩展	133
4.6.1 共享栈.....	133
4.6.2 最小栈 *.....	134
4.7 本章小结	137
第 5 章 队列	138
5.1 先入先出	138
5.2 队列的结构	139
5.2.1 链式队列.....	139
5.2.2 整体搬移 *.....	140
5.2.3 循环队列.....	143
5.2.4 双栈队列 *.....	145
5.2.5 实验：队列的性能对比 *.....	146
5.3 双端队列	147
5.4 本章小结	148

第 6 章 矩阵	150
6.1 矩阵的结构	150
6.1.1 按行优先和按列优先	150
6.1.2 实验：朴素矩阵乘法 *	152
6.1.3 空间局部性和时间局部性 *	154
6.1.4 实验：并行矩阵乘法 *	155
6.1.5 实验：基于分治的矩阵乘法 *	156
6.1.6 Strassen 算法 *	158
6.2 矩阵的压缩存储	160
6.2.1 欠自由度矩阵的压缩存储	160
6.2.2 稀疏矩阵的压缩存储	161
6.3 本章小结	162
第 7 章 树	163
7.1 树的结构	163
7.1.1 线性结构和全序关系 *	163
7.1.2 半线性结构和偏序关系	165
7.1.3 树上的节点关系	166
7.1.4 树上的层次结构	168
7.1.5 树的递归定义	169
7.1.6 树的形态差异性	170
7.2 插入和删除	171
7.2.1 树的定义 *	171
7.2.2 插入元素和插入子树 *	173
7.2.3 删除子树和分离子树 *	174
7.3 树的遍历	174
7.3.1 层次遍历	175
7.3.2 先序遍历和后序遍历	177
7.3.3 树的节点析构 *	178
7.3.4 势能分析法 *	179
7.3.5 实验：三种遍历的对比 *	180
7.4 树和栈的关系	182
7.4.1 树和森林的计数 *	182
7.4.2 实验：树和森林与栈的映射 *	182

7.4.3 树的逆线性化 *	185
7.5 本章小结	186
第 8 章 二叉树	188
8.1 从树到二叉树	188
8.1.1 有序树和二叉树	188
8.1.2 二叉树的节点	189
8.1.3 长女-次妹表示法	190
8.1.4 二叉树和栈的映射 *	191
8.2 二叉树的遍历	193
8.2.1 利用半线性递归进行遍历	193
8.2.2 利用线性递归进行遍历	195
8.2.3 利用迭代器进行遍历 *	200
8.3 二叉树的逆线性化	205
8.3.1 实验：利用中序序列和另一序列重建二叉树	205
8.3.2 利用后序序列和另一序列重建二叉树	210
8.3.3 利用先序序列和层次序列重建二叉树 *	211
8.3.4 利用增强序列重建二叉树 *	213
第 9 章 优先队列	216
9.1 线性优先队列	217
9.1.1 实验：在无序序列上查找最小值 *	217
9.1.2 实验：用有序序列实现优先队列 *	219
9.2 选择排序	221
9.2.1 简单选择排序	221
9.2.2 双选择排序 *	223
9.2.3 宾果排序 *	224
9.2.4 实验：稳定版本的选择排序 *	224
9.3 插入排序	225
9.3.1 实验：直接插入排序和折半插入排序	225
9.3.2 实验：列表版本的选择排序和插入排序 *	228
9.3.3 实验：插入排序的自适应性 *	229
9.4 锦标赛树	230
9.4.1 胜者树的结构 *	230
9.4.2 完全二叉树	232

9.4.3 胜者树的插入 *	233
9.4.4 实验：胜者树的建立 *	235
9.4.5 胜者树的删除 *	236
9.4.6 锦标赛排序 *	237
9.5 完全二叉堆	237
9.5.1 完全二叉堆的结构	237
9.5.2 完全二叉堆的插入	238
9.5.3 完全二叉堆的删除	239
9.5.4 完全二叉堆的建立	239
9.5.5 实验：堆排序	240
9.6 左式堆	242
9.6.1 空节点路径长度 *	242
9.6.2 旋转下沉 *	243
9.6.3 堆的合并 *	244
9.6.4 左偏树 *	245
9.6.5 随机堆 *	246
9.7 本章小结	247
第 10 章 树的应用	249
10.1 表达式树	249
10.1.1 表达式树的结构	249
10.1.2 表达式树的计算	250
10.1.3 表达式树的建立 *	251
10.1.4 抽象语法树 *	252
10.2 编码树	253
10.2.1 编码效率问题 *	253
10.2.2 前缀无歧义编码	255
10.2.3 二叉编码树	255
10.2.4 最优编码树 *	257
10.2.5 Huffman 树	258
10.2.6 建立最优 PFC 编码树的时间复杂度 *	259
10.3 并查集	259
10.3.1 并查集的结构	259
10.3.2 按规模合并和按秩合并 *	261

目 录

10.3.3 路径压缩、分裂和折半 *	263
10.4 本章小结	265
参考文献	266
附录 A C++ 的安装和配置	268
附录 B 清华大学计算机考研指南	269
致 谢	283

第1章 绪论

1.1 对象

本书采用现代 C++ 的编程范式^[4]。作为一门 OOP 的编程语言，C++ 的设计范式总是从类和对象起步。在本书的第一节，我们将以一个“本书所有数据结构和算法的基类”作为引入，讲解如何利用 C++20 中的模块特性设计类。下面是第一个程序，它是一个模块接口文件，功能上类似于旧标准 C++ 中的头文件。它定义了命名空间 `dslab` 中的一个 `Object` 类，作为本书中所有数据结构和算法的基类。

```

1 // Object.ixx
2 module;
3 #include <string>
4 #include <typeinfo>
5 export module Framework.Object;
6 export namespace dslab {
7 class Object {
8 public:
9     virtual std::string type_name() const {
10         return typeid(*this).name();
11     }
12 }
13 }
```

在旧标准 C++ 中，通常在头文件 (.h) 里写定义，而在源文件 (.cpp) 里写实现。C++20 里引入了模块 (module) 的概念，用模块接口文件 (.ixx) 代替头文件。模块的引入大大提高了大型 C++ 项目的编译速度，因为相同的模块只需要编译一次，而被不同源文件 `#include` 的同一个头文件会引入大量的重复编译工作。本书中的所有代码都使用模块。模块接口文件的后缀名 (.ixx) 是 MSVC 推荐的后缀名；如果您使用的是 GCC，需要开启 `-fmodules` 选项来支持它；如果您使用的是 Clang，需要修改配置文件才能支持 .ixx。您也可以简单地修改后缀名为普通的 C++ 文件名，如 .cxx 或 .cc。

在模块接口文件的开始，我们需要引入标准库。我们通过一行单独的 `module;` 进入全局模块环境，然后和往常那样，将需要使用的标准库 `#include` 到程序中。这种写法适用于版本较低的编译器。目前 MSVC 已经允许用户采用 `import std;` 这样的方法一次性引入整个标准库，这是 C++23 标准中的内容，本书的配套代码也都使用这种方式引入。一次性引入整个标准库看起来很吓人，但在模块的支持下，它比旧标准 C++ 里单独 `#include <vector>` 更快。

接着，我们需要通过 `export module` 说明，当前模块接口文件导出的是什么

模块。和 Java 类似，模块名可以用点分隔来表示子模块。本书采用的模块命名规则和 Java 相同，即采用目录名 + 文件名的格式。

本书定义的所有内容定义在命名空间 `dslab` 下，因此，我们总是采用 `export namespace dslab` 的方式导出 `dslab` 中的所有内容。如果我们需要导出特定的类或函数，则需要将关键字 `export` 移动到对应的类或函数的前面。

在上述代码的命名空间内不再含有模块的特性。

我们使用 `typeid(*this).name()` 方法来在运行时获取一个对象的类型名，这是因为 C++ 仍然没有加入反射（reflection）这个高级语言喜闻乐见的特性。这个方法没有标准的实现，MSVC 中会返回一个比较可读的结果，而 GCC 和 Clang 中则会返回不那么可读的结果。因此，我们后续实现派生于 `Object` 的数据结构和算法的时候，往往需要重写这个方法，将其改为更加可读的版本。最后，`typeid(*this).name()` 返回的是 C 风格的字符串类型 (`const char*`)，在本书中会尽可能避免使用 C 风格指针，因此将其隐式转换为 C++ 风格的 `std::string` 返回。

下面我们编写一个测试的源文件 (.cpp)，来导入刚才实现的 `Object` 类。

```

1 // ObjectTest.cpp
2 import Framework.Object;
3 import std;
4 using namespace std;
5 class Test : public dslab::Object {};
6 int main() {
7     cout << (Test()).type_name() << endl;
8     return 0;
9 }
```

在源文件 (.cpp) 中，可以通过 `import` 关键字来导入定义的模块。在这个测试文件中，我们使用了一个 `Test` 类继承此前定义的 `Object` 类，并查看 `Test` 类型临时变量的类型名。如前所述，这个程序在不同的编译器下会得到不同的输出结果。

类似于其他高级语言，我们可能希望 `import Framework;` 提供一次性引入模块 `Framework` 中所有子模块的功能。此时，我们可以建立一个新的模块接口文件，用来实现这个功能。

```

1 // Framework.ixx
2 export module Framework;
3 export import Framework.Object;
```

其中，混合使用 `export import` 表示，将子模块 `Framework.Object` 导入之后，作为亲模块 `Framework` 的一部分导出。未来我们增加其他的子模块，只需要在 `Framework.ixx` 里增加对应的一行，就可以让它一并作为 `Framework` 的一部分被导出。

1.2 数据结构

数据结构 (data structure) 是什么? 这是一个很多新手都不会思考的问题。当然, 我们可以简单地把“数据结构”这个词拆分为**数据**和**结构**, 即: 数据结构是计算机中的数据元素以某种结构化的形式组成的集合。

您可能会觉得这种定义过于草率, 或者和您在教材上看到的定义不同。这是因为计算机是一门工程学科, 对于不涉及工程实现的问题, 都不存在标准化的定义。比如, “数据结构”和“算法”, 甚至“计算机”这样的基本概念, 都不存在标准化的定义。一些教材会把算盘甚至算筹划归“计算机”的范畴, 并把手工算法(如尺规作图, 甚至按照菜谱烹饪食物)划归“算法”的范畴。这种概念和定义的争议在计算机领域广泛存在, 它主要来自以下几个原因。

1. 为了叙述简便, 有些概念会借用一个已经存在的专有名词, 从而引发歧义。如**树** (tree) 这个词在计算机领域就有常用但迥然不同的两个概念。图论中的“树”出现得比较早, 但没有人愿意将工程界经常出现的“树”称为“有限有根有序有标号的树”^[7]——英文里这些词并不能缩写为“四有树”。
2. 研究人员各执一词, 从而引发歧义。这个现象的典型例子是“计数时从 0 开始还是从 1 开始”。从 0 开始是有一定数学上的优越性的, 可以避免一些公式出现刻意的“+1”余项^[6]; 但从 1 开始计数更符合自然习惯^[5]。这个问题直接导致在有些问题(如“树的高度”)上, 不同教材的说法不同。考生如果参考了错误的教材就会导致无辜的失分。
3. 随着计算机领域的快速发展, 一些概念的含义会发生变化。如众所周知, **字节** (byte) 表示 8 个二进制位; 但在远古时代, 不同计算机采用的“字节”定义互不相同, 有些计算机甚至是十进制的, 那个时候一个字节可能表示 2 个十进制位^[7]。
4. 受计算机科学家的意识形态影响, 同一概念的用词有所不同。如“树上的上层邻接和下层邻接节点”这一概念, 现在普遍使用的词是 parent 和 child; 然而思想老旧的人可能还在用 father 和 son^[8], 进步主义者则可能会用 mother 和 daughter^[9]。
5. 计算机领域的大多数成果出自美国, 而英语翻译为汉语时, 不同译者可能采用不同的译法。如 robustness 有音译的**鲁棒性**^[10]和意译的**健壮性**^[11], hash 有音译的**哈希**^[5]和意译的**散列**^[6]。
6. 从业人员为了销售产品或取得投资, 存在滥用、炒作部分计算机概念的情况。如“人工智能”“大数据”“云计算”“区块链”“元宇宙”等概念是这个现象的重灾区。

本书在描述无标准化的定义时，将尽可能让您把握住概念的要点。正如：数据结构是**计算机**中的数据元素以某种结构化的形式组成的**集合**。

数据结构中的数据是存储在**计算机**中的数据。我们在解题的时候往往会在纸上画出数据结构的图形，这是为了让自己更好地理解数据在计算机中的组织方式。计算机中的数据和纸上的数据会有很多的不同。比如，计算机处理数据时存在高速缓存（Cache，参见《组成原理》），这会使算法的局部性对算法性能造成重大影响，而这是在纸上无法分析出的。

我们研究的计算机都是二进制计算机，这意味着数据结构的数据元素总是一个二进制数码串，程序会将这些二进制数据转换为有意义的数据类型，比如**char**、**int**、**double**或某个自定义的类。因为一台计算机存储的二进制串不能无限长，所以我们讨论整数数据结构或者浮点数数据结构的时候，其元素的真正取值范围并不会是数学上的 Z 或者 R ，而是一个有限集合。这一特性经常被一些数学成绩优秀的学生所忽略，尤其当他们试图用数学方法完成一些证明时。另一个有限性，即数据结构规模（存放的元素个数）的有限性，则不那么容易被忽略。

通常情况下，一个数据结构中的数据元素具有相同的类型，比如，一个数据结构不能既存储**int**又存储**std::string**。一些情况下，我们可能希望元素具有多个可能的类型，此时可以选择 C 语言的**union**或者 C++ 的**std::variant**。更加特殊的情况下，我们可能希望元素是任意类型的，此时可以选择 C 语言的**void***或者 C++ 的**std::any**。这种特殊的情况更多地被视为一种**编程技巧**，而非**数据结构**中研究的理论问题。

数据结构中的数据元素具有实质上的**互异性**。这是由于，计算机中不可能存在两个完全一样的数据：至少它们的地址不同。这是我们把数据结构定义为数据的**集合**的原因。这种互异性在我们设计算法的时候需要尤其注意，比如，如果我们将序列 $[a_1, b, c]$ 修改为了 $[a_2, b, c]$ ，其中 a_1 和 a_2 的值相同、地址不同，看起来好像修改前后这个序列本身没有什么区别，但实际上可能会引起重大的错误。因为我们在修改这个序列的过程中，不能保证没有外部的指针指向 a_1 。如果修改之后， a_1 的内存被释放，外部指向它的指针就会变成悬垂指针，造成严重的后果。很多有经验的程序员也容易忽视这一点，在本书中您还将看到更加实际的例子来说明它。

现在，我们回到数据结构的实现中来。现在我们需要继承上一节中实现的**Object**，设计一个数据结构的基类。作为一个抽象的数据结构，我们需要从数据结构的定义中挖掘共性。

1. 数据结构总是存储相同的类型。对于支持泛型编程的 C++ 来说，我们可以使用模板类型作为数据结构中的元素类型。

2. 数据结构是有限多个元素组成的，因此任何数据结构都具有`size()`方法。在 C++ 中，表示规模（size）的类型是`size_t`，它通常是一个无符号整数类型，可能是`uint32_t`或者`uint64_t`。

于是，我们可以这样设计`DataStructure`类：

```

1 template <typename T>
2 class DataStructure : public Object {
3 public:
4     virtual size_t size() const = 0;
5     virtual bool empty() const {
6         return size() == 0;
7     }
8 };

```

其中，获取规模的方法`size`被设计成了纯虚函数（pure virtual function），这意味着它的子类必须要实现这个方法。和规模相关，我们支持判空方法`empty`。

可能有的读者会认为，数据结构作为数据元素的集合，它理应支持增加元素、删除元素、查找元素这样的方法，然而事实却并非如此。有一些数据结构，它们可能一旦建立之后就无法添加或删除元素，或者只能添加和删除特定的元素，即**写受限**。同样地，另一些数据结构可能内部对用户不透明，用户只被允许访问特定的元素，并不能在数据结构中自由地查找，即**读受限**。在本书的后续部分，您将看到写受限和读受限的具体例子。

1.3 算法

1.3.1 算法的定义

和数据结构经常同时出现的另一个名词是**算法**（algorithm）。算法通常指接受某些**输入**（input），在**有限**（finite）步骤内可以产生**输出**（output）的计算机计算方法^[7]。输出和输入的关系可以理解为算法的功能。对于同一功能，可能存在多种算法，对于相同的输入，它们通过不同的步骤可以得到**等价**的输出。这里并不一定要求得到相同的输出，比如我们要求一个数的倍数，输入 a 的情况下，输出 $2a$ 和 $3a$ 都是正确的输出，在“倍数”的观点下，这两个输出等价。

通常，算法的定义除了上述的三个要素：输入、输出和有限之外，还包括**可行**（effective）和**确切**（definite）。比如，算法中如果包含了“如果哥德巴赫猜想正确，则 …”，则在当前不满足可行性；如果包含了“任取一个 …”，则不满足确切性（对同一算法，同样的输入必须产生同样的输出）。不过，在计算机上用代码写成的算法，通常都具有可行性和确切性，所以我们一般不讨论它们。有限性可以通过白盒测试和黑盒测试评估。在设计“算法”的基类时，我们只考虑输入和输出。

正如我们所熟知的那样,C++有一个概念同样具有输入和输出:函数(function)。但是,直接用函数来表示一个算法并不OOP,因此我们采用仿函数(functor)而不是普通的全局函数。仿函数是一种类,它重载了括号运算符operator()。仿函数对象可以像一个普通的函数一样被调用,并且可以被转换为std::function。和普通的全局函数相比,仿函数具有两方面的优势:

1. 仿函数可以有成员变量。比如,可以定义一个m_count来统计一个仿函数对象被调用的次数。而普通的全局函数则无法做到这一点,非static的变量会在函数结束后被释放,而static的变量又强制被全局共享。
2. 仿函数可以有成员函数(也称为方法)。当仿函数的功能非常复杂时,它可以将功能拆解为大量的成员函数。因为这些成员函数都处在仿函数内部,所以不会污染外部的命名空间,并且可以清楚地看到它们和仿函数之间的关系。必要的时候,还可以通过嵌套类显式地指明其中的关系。而普通的全局函数,则无法在函数内嵌套一个没有实现的子函数。

于是,我们用仿函数设计了算法类Algorithm:

```

1 template <typename OutputType, typename... InputTypes>
2 class Algorithm;
3
4 template <typename OutputType, typename... InputTypes>
5 class Algorithm<OutputType(InputTypes...)> : public Object {
6 public:
7     using Output = OutputType;
8     virtual OutputType operator()(InputTypes... inputs) = 0;
9 };

```

这个类只定义了一个纯虚的括号运算符重载。这里我们使用了可变参数模板(以“...”表示),以处理不同输入的算法。另一方面,我们借用了std::function的表示形式,要求用户使用OutputType(InputTypes...)的方式给出模板参数表。比如,一个输入两个整数、输出一个整数的算法可以继承于Algorithm<int(int, int)>。

1.3.2 算法的评价

作为一种解决问题的方法,算法的评价是多维度的,它们构成了算法题的主流题型。即使是在算法设计题中,也经常有“对设计的算法进行评价”的附加要求。本节将简要介绍算法的几个典型的评价维度。

正确性。正确性检验通常分为两个方面:

1. **有限性检验。**如前所述,有限性是算法定义的组成部分之一。有限性检验,主要用于判断带有无限循环(如while(true))或强制跳转(goto)的算法是

否必定会终止。没有无限循环或强制跳转的情况，有限性是默认的。这一方面对应着算法定义中的**有限性要求**。

2. **结果正确性检验**。即验证输出的结果满足算法的需求。在算法有确定的正确结果时，这一检验是“非黑即白”的；而在算法没有确定的正确结果时，可能需要专用的检验程序甚至人工打分（比如较早的象棋 AI，往往是以高手对一些局面的形势打分作为基础训练数据的）。这一方面对应着算法定义中的**输入和输出**。

算法定义中的另外两点，**可行性和确切性**，正如我们之前所讨论的那样，通常都可以被直接默认，而不需要进行检验。

效率。评价算法效率的标准可以简单地概括为多、快、好、省^[12]。在《数据结构》中，通常只研究“快”和“省”这两个方面，而《网络原理》则需要考虑全部的四个方面。在《网络原理》中，“多”代表网络流量，“好”代表网络质量。

1. **时间效率**（快）。在计算机上运行算法一定会消耗时间，时间效率高的算法消耗的时间比较短。
2. **空间效率**（省）。在计算机上运行算法一定会消耗空间（硬件资源），空间效率高的算法消耗的硬件资源比较少。

在不同的计算机、不同的操作系统、不同的编程语言实现下，同一算法消耗的时间和空间可能大相径庭。为了抵消这些变量对算法效率评价的干扰作用，在《数据结构》这门学科里进行算法评价时，往往不那么注重真实的时间、空间消耗，而倾向于做**复杂度分析**。关于复杂度的讨论参见后文。

稳健性（robustness，又译健壮性、鲁棒性；在看到英文之前，我曾一度认为“鲁棒性”这个词来源于山东大汉身体棒的地域刻板印象）。即算法面对意料之外的输入的能力。

泛用性。即算法是否能很方便地用于设计目的之外的其他场合。

在上述 4 个评价维度中，正确性和效率是《数据结构》学科研究的主要内容，算法评价题也总是围绕正确性检验和复杂度分析命题；这两个问题留到后面的小节里展开讨论。而稳健性和泛用性，则在课程设置上属于《软件工程》讨论的内容，在下一节会用一个实验展示它，后续不再赘述。

1.3.3 实验：从 1 加到 N

算法和实现它的代码（code）或程序（program）有本质区别^[13]。按照通常意义的划分，算法更接近于理科的范畴，而实现它的代码更接近工科的范畴。一些同学可能很擅长设计出精妙绝伦的算法，但需要耗费巨大的精力才能实现它，并遗留不计其数的 Bug；另一些同学可能在设计算法上感到举步维艰，但如果拿到已有

的设计方案，可以轻松完成一份漂亮的代码。算法设计和工程实现对于计算机学科的研究同等重要。参加过语文高考的学生可能会很有感受：自己有一个绝妙的构思，但没有办法在有限的时间下把它说清楚。专精算法设计而忽略工程实现的同学会有类似的感觉。

上一节中介绍的算法评价维度中，**稳健性和泛用性**是高度依赖于算法的实现的（当然，也有少数情况和算法本身的设计相关）。在本节将通过一个实验作为例子，向读者展示：对于相同的算法，代码实现的不同会影响这两个评价维度。本节的实验可以在 *Sum.cpp* 中找到。

我们讨论一个非常简单的例子：从 1 加到 n 的求和。输入一个正整数 n ，输出 $1 + 2 + \dots + n$ 的和。我们从这个例子出发，介绍本书使用的实验框架。首先，我们定义本问题的基类：

```
1 class Sum : public Algorithm<int (int)> { };
```

因为这个算法不需要用到额外的成员变量或成员函数，所以它简单地继承了我们定义的算法类 `Algorithm<int, int>`。我们也可以使用别名（alias）技术来指定 `Sum` 和 `Algorithm<int, int>` 是同一个类：

```
1 using Sum = Algorithm<int (int)>;
```

相对于 C 语言的 `typedef`，使用关键字 `using` 的别名技术更加清晰，并且支持带模板参数的语法。当我们使用 C++ 编程时，应当使用别名替代 `typedef`。

现在我们回到求和问题上来。作为实验的一部分，您可以自己实现一个类，继承 `Sum`，并重写 `operator()`，来和我提供的示例程序做对比。这一过程可以发生在您阅读下面对示例程序的解析之前，也可以发生在之后。在这个实验的示例程序里，我设计了几个 `Sum` 的派生类来完成这个算法功能。

首先，我们可以简单地把每个数字加起来，就像这样：

```
1 // SumBasic
2 int operator()(int n) override {
3     int sum { 0 };
4     for (int i { 1 }; i <= n; ++i) {
5         sum += i;
6     }
7     return sum;
8 }
```

使用一对大括号（而不是等于号）对数据进行初始化，称为统一初始化，这是现代 C++ 建议的初始化方式。它有助于提示用户这里发生了初始化（调用了构造函数），而不是发生了赋值（调用了 `operator=`）。另一方面，统一初始化有更加严格的数据类型检查，这可以防止某些意料之外的隐式类型转换。

此外，如果不需要使用返回值，则建议使用前置运算符`++i`而不是后置运算符`i++`。尽管当`i`是整数的时候编译器往往会自动进行优化，可当`i`是自定义类型时不一定会这样。

另一方面，`override`关键字用于提示编译器该方法是重载方法。永远要记得为您的重载方法添加这个关键字，以避免由于细小的区别（如遗漏`const`）而不小心定义了一个新方法。

上面的这个算法显然称不上好。著名科学家高斯在很小的时候就发现了等差数列求和的一般公式。我们可以使用公式，得到另一个可行的算法。

```
1 // SumAP
2 int operator()(int n) override {
3     return n * (n + 1) / 2;
4 }
```

由于这个算法的正确性十分显然，您或许觉得这个程序毫无问题，直到您发现了另外一个程序：

```
1 // SumAP2
2 int operator()(int n) override {
3     if (n % 2 == 0) {
4         return n / 2 * (n + 1);
5     } else {
6         return (n + 1) / 2 * n;
7     }
8 }
```

通过对比 SumAP 和 SumAP2，您会立刻意识到 SumAP 存在的问题：对于某个区间内的 n ， $\frac{n(n+1)}{2}$ 的值不会超过`int`的最大值，但 $n(n+1)$ 会超过这个值。比如，当 $n = 50,000$ 的时候，SumAP2 和 SumBasic 都能输出正确的结果，而算法 SumAP 则会因为数据溢出返回一个负数。您可以算出使 SumAP 错误而 SumAP2 仍然正确的区间。

但 SumAP2 也很难称之为无可挑剔，比如说，如果 n 更大一些，比如取 100,000，则它也无法输出一个正确的值。这种情况下，甚至最朴素的 SumBasic 也无法输出正确的值。

那么，我们思考一个问题：上述的三个实现，哪些是正确的？

在我们实际进行代码实现的时候，通常会倾向于 SumAP2，因为它既有较高的效率（相对于 SumBasic 而言），又保证了在数据不溢出的情况下能输出正确的结果（相对于 SumAP 而言）。但在我们进行算法评估的时候，通常认为这三个实现都是正确的，并且 SumAP 和 SumAP2 实质上是同一种算法。也就是说，数据溢出这种问题并不在我们的评估模型之内，算法虽然执行在计算机上，但又是独立于计算机的；算法虽然需要代码去实现，但又是独立于实现它的代码的。在《数据结

表 1.1 三种 Sum 算法的正确性检验

n		SumBasic	SumAP	SumAP2
10	普通值	√	√	√
0	边界值	√	√	√
50,000	临界值	√	✗	√
100,000	溢出值	✗	✗	✗
-10	非法值	√	✗	✗

构》这门学科中的研究对象，通常和体系结构、操作系统、编程语言等因素都没有关系。

在本节的末尾，提出一个有趣的问题：SumAP2 在 n 非常大的时候也会出现溢出问题。当然，这超过了 `int` 所能表示的上限。但是，这种情况下，返回什么样的值是合理的？SumAP2 返回的值（有可能是一个负数）真的合理吗？这是纯粹的工程问题，并不在《数据结构》研究的范围内。

一种可能的方案是，在溢出的时候返回 `std::numeric_limits<int>::max()`，这是 C++ 中用来替代 C 语言宏 `INT32_MAX` 的方法。这种方案称为“饱和”。饱和保证了数值不会因为溢出而变为负值，当数值具有实际意义时，一个不知所云的负值可能会引发连锁的负面反应。比如，路由器可能会认为两个节点之间的距离为负（事实上应当是无穷大），从而完全错误地计算路由。因此，如果实现了饱和，在泛用性上可以得到一定的提升。

还有一个问题是，如果 n 为负数，则应该如何输出？当然，我们的题目要求 n 是正整数，负数是非法输入；但我们也希望程序能输出一个有意义的值。按照朴素的想法（也就是 SumBasic），这个时候应该输出 0，然而 SumAP 和 SumAP2 都做不到这一点。在示例程序中给出了一个考虑了饱和和负数输入的实现，您也可以独立尝试实现这个功能。

1.4 正确性检验

在上一节已经说明，正确性检验可以拆解为两个方面：有限性检验和结果正确性检验。在实际解题的过程中，这两项检验往往可以一并完成，即检验算法是否能在有限时间内输出正确结果。

解决正确性检验的一般方法是**递降法**。它的思想基础是在计算机领域至关重要、并且是《数据结构》学科核心的**递归思维方法**；它的理论依据则是作为在整数公理系统中举足轻重的**数学归纳法**。

在本节中，将从数学归纳法的角度出发介绍递降法的原理，这部分有助于让您对递归思维有更加深刻的理解。当然在实际考试中只要会用递降法解题即可，您也可以跳过数学的部分（如果您对数学感到头疼，这是建议的选项）。

1.4.1 经典的归纳和递降 *

在高中理科数学中介绍了数学归纳法的经典形式。由于各省教材不同，您可能接触到过两种表述不太一样的数学归纳法：

定理 1.1 (第一归纳法)：令 $P(n)$ 是一个关于正整数 n 的命题，若满足：

1. $P(1)$ 。
2. $\forall n \geq 1, P(n) \rightarrow P(n + 1)$ 。

则 $\forall n, P(n)$ 。

定理 1.2 (第二归纳法)：令 $P(n)$ 是一个关于正整数 n 的命题，若满足：

1. $P(1)$ 。
2. $\forall n \geq 1, (\forall k \leq n, P(k)) \rightarrow P(n + 1)$ 。

则 $\forall n, P(n)$ 。

其中，第一归纳法是皮亚诺 (Piano) 公理体系的一部分。您可以很轻松地用第一归纳法推导出第二归纳法。另一方面，第二归纳法的**归纳假设** $\forall k \leq n, P(k)$ ，显然比第一归纳法的归纳假设 $P(n)$ 更强，所以实际应用数学归纳法进行证明时，通常都使用第二归纳法，而不使用第一归纳法。

在上面的表述中，归纳的过程是从 1 开始的，这比较符合数学研究者的工作习惯。在计算机领域，归纳法常常从 0 开始（有时甚至从 -1 开始）。显然，这并不影响它的正确性。本节后续对“正整数”相关问题的讨论，一般替换成“自然数”也同样可用。

我们将第二归纳法称为**经典归纳法**。经典归纳法可以用来处理有关正整数的命题。相应地，对于输入是正整数的算法，可以使用与经典归纳法相对应的**经典递降法**。

定理 1.3 (经典递降法)：令 $A(n)$ 是一个输入正整数 n 的算法，若满足：

1. $A(1)$ 可在有限时间输出正确结果。
2. $\forall n > 1, A(n)$ 可在有限时间正确地将问题化归为有限个 $A(k_j)$ ，其中 $k_j < n$ 。

则 $\forall n, A(n)$ 可在有限时间输出正确结果。

经典递降法的正确性由经典归纳法保证，您可以自己证明这一点。显然，经典递降法的应用范围非常狭小：只能用来处理输入是正整数的算法。对于实际的算法，它的输入数据通常是多个数、乃至于数组和各种数据结构，而非孤零零的一个正整数。因此，需要对经典归纳法进行推广，从而使其可以应用到更广的范围中，

并使其相对应的递降法能够处理更加多样的输入数据。

1.4.2 带映射的归纳和递降 *

在《线性代数》的第一章“行列式”中，您一定见过下面这个经典的命题：

$$\begin{vmatrix} 1 & 1 & \cdots & 1 \\ x_1 & x_2 & \cdots & x_n \\ \vdots & \vdots & & \vdots \\ x_1^{n-1} & x_2^{n-2} & \cdots & x_n^{n-1} \end{vmatrix} = \prod_{1 \leq i < j \leq n} (x_i - x_j)$$

通常这个命题都是用数学归纳法证明的，您不会有任何违和感。

但在这个命题中， n 并不是唯一的变量。在 n 之外，还有 x_1, x_2, \dots, x_n 这 n 个变量。换句话说，这个命题所接受的是一个任意有限长的向量 (x_1, x_2, \dots, x_n) 。在使用数学归纳法进行证明时，实际上指定了一个映射 $f(x_1, x_2, \dots, x_n) = n$ ，输入数据是这个映射的原象，而在这个映射的象上做数学归纳法。有些数学书为了指明这一点，会在证明的开头写上“对行列式的阶数 n 做归纳”；而有些书则图省事略去了这句话。

在这个例子中，通过“对行列式的阶数做归纳”，将复杂的输入数据映射到了正整数集，这是一种典型的应用归纳法的技术。类似地，在证明对整数的命题时，可以对它的绝对值做归纳，等等。一般而言，可以将经典归纳法改写成下面的“带映射的”形式：

定理 1.4 (带映射的经典归纳法)：令 $P(x)$ 是一个关于 $x \in X$ 的命题， $f : X \rightarrow \mathbb{N}^+$ 是满射，若满足：

1. $f(x) = 1 \rightarrow P(x)$ 。
2. $[\forall y, ((f(y) < f(x)) \rightarrow P(y))] \rightarrow P(x)$ 。

则 $\forall x, P(x)$ 。

可以对命题 $Q(n) = (\forall x, f(x) = n \rightarrow P(x))$ 使用经典的归纳法来证明上面这个带映射的形式。相应地，也存在带映射的经典递降法。

定理 1.5 (带映射的经典递降法)：令 $A(x)$ 是一个输入 $x \in X$ 的算法， $f : X \rightarrow \mathbb{N}^+$ 是满射，若满足：

1. 当 $f(x) = 1$ 时， $A(x)$ 可在有限时间输出正确结果。
2. $\forall x, f(x) > 1, A(x)$ 可在有限时间正确地将问题化归为有限个 $A(y_j)$ ，其中 $f(y_j) < f(x)$ 。

则 $\forall x, A(x)$ 可在有限时间输出正确结果。

为了让上述定义的每个 $Q(n)$ 都存在， f 需要保证是满射。这又是一项对递降

法应用范围的限制，需要想办法清除掉它，得到更加一般的、更加通用的解题方法。

1.4.3 良序关系 *

在带映射的归纳法中，需要通过满射将定义域 X 映射到正整数集 \mathbb{N}^+ 上。很多时候，这样的满射并不容易构造。与其试图用高超的技巧去构造满射，不如从正整数集 \mathbb{N}^+ 入手：放宽映射的象的条件，不要求它一定是 \mathbb{N}^+ ，只需要满足一些和 \mathbb{N}^+ 相似的性质即可。

对于一般的集合，引入**良序** (well-order) 的概念。如果集合 S 上的一个关系 \leq 满足：

1. **完全性。** $x \leq y$ 和 $y \leq x$ 至少有一个成立。
2. **传递性。** 如果 $x \leq y$ 且 $y \leq z$ ，那么 $x \leq z$ 。
3. **反对称性。** 如果 $x \leq y$ 和 $y \leq x$ 均成立，那么 $x = y$ 。
4. **最小值。** 对于集合 S 的任意非空子集 A ，存在最小值 $\min A = x$ 。即对于 A 中的其他元素 y ，总有 $x \leq y$ 。

那么称 \leq 是 S 上的一个**良序关系**，同时称 S 为**良序集**。类似于数值的小于等于“ \leq ”和小于“ $<$ ”关系，对于关系 \leq ，如果 $x \leq y$ 且 $x \neq y$ ，则可以记为 $x < y$ 。

根据上述定义，熟知的小于等于关系“ \leq ”在正整数集 \mathbb{N}^+ 上是良序的，而在整数集 \mathbb{Z} 上不是良序的。当然，可以通过定义“绝对值小于等于”让整数集 \mathbb{Z} 成为良序集。同时，熟知的小于等于关系“ $<$ ”在非负实数集 $\mathbb{R}^+ \cup \{0\}$ 上不是良序的，因为它不满足最小值条件（任取一个开区间作为它的子集，都没有最小值）。

和熟知的正整数集 \mathbb{N}^+ 相比，一般的良序集具有下面的相似性质：

定理 1.6 (无穷递降): 在良序集 S 中，不存在无穷序列 $\{x_n\}$ ，使得 $x_{j+1} < x_j$ 对一切 j 成立。

您可以用反证法证明它。这一性质使得在一般的良序集上做归纳法成为可能，后面的小节会回到这个问题上来。

1.4.4 字典序 *

在上一小节，您发现熟知的小于等于关系在非负实数集 $\mathbb{R}^+ \cup \{0\}$ 上并不是良序的。如果您感到不服气，想要尝试去构造 \mathbb{R} 上的良序关系，几乎一定会无功而返（目前还没有数学家定义出实数集 \mathbb{R} 上的显式良序关系）。然而我们有下面的定理：

定理 1.7 (良序定理): (ZFC) 任何集合都存在良序关系。

在集合论的 ZFC 公理体系下，上述定理成立。在 ZF 公理体系下，该定理和**选择公理** (AC) 等价。选择公理是有些反直观的（有兴趣的话可以自行搜索），与它

等价的良序定理同样反直观，基本上只能用于理论推导，很难实际应用。

幸运的是，在计算机领域的日常研究中，并不需要使用无所不能的良序定理来“设”出一个良序关系。计算机领域中，输入数据所属的集合总是**可数的**(countable)，而可数集合总是可以很轻松地定义良序关系。其中一个典型的例子是所谓的**字典序**(lexicographical order)。

定理 1.8 (字典序)：设 $\{S_n\}$ 是一个良序集序列， \leq_j 是集合 S_j 上的一个良序关系。则对于无限笛卡尔积 $\prod S_j = S_1 \times S_2 \times S_3 \times \dots$ 中的两个元素 $a = (a_1, a_2, a_3, \dots)$ 和 $b = (b_1, b_2, b_3, \dots)$ ，定义 $a \leq b$ 当且仅当：存在某个 k ，使得 $a_j = b_j$ 对于 $1 \leq j < k$ 成立，但 $a_k \leq_k b_k$ 。那么 \leq 是 $\prod S_j$ 上的一个良序关系。

上面定义的良序关系，是针对无限长向量的。将它稍微修改一下，就可以用来定义任意长向量。下面给出了一种证明方法，您也可以自己解决这个问题。在每个集合 S_j 中增加一个元素 \emptyset_j ，让这个元素作为 $S_j \cup \{\emptyset_j\}$ 的最小值。对于非无限长的向量 (a_1, a_2, \dots, a_n) ，将其延伸为 $(a_1, a_2, \dots, a_n, \emptyset_{n+1}, \emptyset_{n+2}, \dots)$ ，就得到了无限长向量。这个映射是一一对应的，从而可以用无限长向量的字典序去定义任意长向量的字典序。特别地，由 $S_j = \{a, b, \dots, z\}$ 构成的任意长向量的字典序，就是英文字典中排列单词的顺序。

在本节的最后需要说明，实际上计算机领域的输入数据所属集合总是有限的，因为任何硬件设备都存在可承载的数据量上限。但是，在不能用枚举法的情况下，通常都会选择将输入集合从有限集扩大为可数集，从而使用针对可数集的递降法。

1.4.5 超限的归纳和递降 *

在定义良序关系之后，就可以使用**超限归纳法**来证明命题。超限归纳法本质上是经典数学归纳法在集合论上的一般形式，可由良序关系的定义直接导出，而无需用到良序定理。

定理 1.9 (超限归纳法)：令 $P(x)$ 是一个关于 $x \in X$ 的命题， $f : X \rightarrow S$ 是满射， S 是关于 \leq 的良序集，若满足：

1. $(f(x) = \min S) \rightarrow P(x)$ 。
2. $[\forall y, (f(y) < f(x)) \rightarrow P(y)] \rightarrow P(x)$ 。

则 $\forall x, P(x)$ 。

上面的表述和经典的超限归纳法表述有所不同。它融合了之前介绍过的“**映射**”策略。当 S 取为正整数集 \mathbb{N}^+ ，良序关系取为熟知的“ $<$ ”时，超限归纳法的特例就是经典归纳法。这是最自然的良序关系和良序集。在考试解题过程中，通常都只需要用到 \mathbb{N}^+ 或它的子集。尽管解题的时候，通常只会用到这个最自然的良序集，但“良序”的思维，仍然广泛存在于计算机领域的各个学科中。

和超限归纳法对应，可以得到一般形式的递降法。这是我们解决各种算法分析问题的基本方法。

定理 1.10 (递降法): 令 $A(x)$ 是一个输入 $x \in X$ 的算法， $f : X \rightarrow S$ 是满射， S 是关于 \preceq 的良序集，若满足：

1. 当 $f(x) = \min S$ 时， $A(x)$ 可在有限时间输出正确结果。
2. $\forall x, f(x) \neq \min S$ ， $A(x)$ 可在有限时间正确地将问题化归为有限个 $A(y_j)$ ，其中 $f(y_j) < f(x)$ 。

则 $\forall x, A(x)$ 可在有限时间输出正确结果。

递降法和递归 (recursion) 是密不可分的。在上述递降法的表述中，条件 (1) 对应了递归边界，条件 (2) 则对应了递归调用。边界情况通常对应的是最简单的情况，而递归调用则用来将复杂问题拆解成简单问题。只要您有一定的递归编程的经验，那么递降法是非常容易理解的。

递降用于分析算法，而递归用于设计算法，二者思路上相似，只是侧重点不同。当我们设计算法的时候， $A(x)$ 是一个待解决的算法问题，我们尝试将它拆解为有限个规模较小的子问题 $A(y_j)$ ，递归地解决这些子问题，直到到达平凡情况，也就是 $f(x) = \min S$ 的情况。而当我们拿到一个算法对它进行分析的时候，首先证明平凡情况下算法是有限和正确的，然后再证明非平凡情况下，将 $A(x)$ 化归为有限个 $A(y_j)$ 的过程是有限和正确的。

由于递降法用到的计算机思维事实上是递归，我们将不再区分“递降法”和“递归法”。递归（递降）法是《数据结构》学科最为核心的思维方法，在这一章只是用简单的例子介绍它，后面的章节中还会反复出现，并不断增加问题的难度和思维的深度。

下面的几个小节将用几个实际的例子，来说明递降法在正确性检验中的作用。

1.4.6 实验：最大公因数

本节以求最大公因数为例，展示如何证明递归算法的正确性。相关代码可以在 *Gcd.cpp* 中找到。关于这个问题，如果您有一定编程基础，肯定能一眼就知道如何解决，不妨尝试一下。

如果您没有编程基础，可以参考下面的实现，这是大名鼎鼎的最大公因数算法：欧几里得 (Euclid) 辗转相除法的递归形式。

```

1 // GcdEuclid
2 int operator()(int a, int b) override {
3     if (b == 0) {
4         return a;
5     } else {
6         return (*this)(b, a % b);
7     }
8 }
```

```
7     }
8 }
```

在这个算法中，递归边界是 $(a, 0)$ ，因此可以定义 $f(a, b) = \min(a, b)$ ，将输入数据映射到熟知的良序集 \mathbb{N} 。接着就可以用递降法处理这个问题了。

1. 如果 $a < b$ ，那么通过 1 次递归，可变换为等价的 $\gcd(b, a)$ 。
2. 如果 $a \geq b > 0$ ，那么由于 $b > a \% b$ 对一切正整数 a, b 成立，所以通过 1 次递归，可变换为 $f(\cdot)$ 更小的 $(b, a \% b)$ 。接下来只要证 $\gcd(a, b) = \gcd(b, a \% b)$ 。您可以自己完成这一证明。下面提供了一种比较简单的证法。
设 $a = kb + l$ ，其中 $l = a \% b$ 。那么，对于 a 和 b 的公因数 d ，设 $a = Ad$ ， $b = Bd$ ，则 $l = (A - kB)d$ 。因此 d 也是 b 和 l 的公因数。反之，对于 b 和 l 的公因数 d' ，也可推出 d' 也是 a 和 b 的公因数。因此 a 和 b 的公因数集合，与 b 和 l 的公因数集合相同；它们的最大值（即最大公因数）显然也相同。
3. 如果 $b = 0$ ，到达边界， $\gcd(a, 0) = a$ 正确。

如此便完成了 Euclid 辗转相除法的正确性证明。

1.4.7 实验：数组求和

递降法的条件 2 允许在一个递归实例中，调用自身有限次。在上一节中，一个 \gcd 只会调用一次自身；这一小节展示了一个多次调用自身的例子。

作为第一章的实验，我们仍然讨论一个简单的求和问题：给定一个规模为 n 的数组 A ，求 $A[0] + A[1] + \dots + A[n - 1]$ 的和。相关代码可以在 *ArraySum.cpp* 中找到。

```
1 class ArraySum : public Algorithm<int(span<const int>) > {};
```

您可以将它改为您更熟悉的版本，比如 C 风格用指针 `int*` 和长度 `size_t` 定义的数组，或者使用 C++ 的变长数组 `std::vector` 并结合迭代器。这里我们使用 C++20 引入的 `std::span` 来表示数组的视图，它既可以处理 `int*` 定义的数组，又可以处理 `std::vector`。

经典的数组求和方法，是定义一个累加器，把每个数逐一加到累加器上。这个算法在 C++ 中被封装成了 `std::accumulate`：

```
1 // ArraySumBasic
2 int operator()(span<const int> data) override {
3     return accumulate(begin(data), end(data), 0);
4 }
```

在 C++17 中，引入了一个新的累加算法 `std::reduce`，它允许乱序计算，从而给了硬件技术（如 SIMD，参见《组成原理》）更高的优化空间。

```
1 // ArraySumReduce
```

```

1 int operator()(span<const int> data) override {
2     return reduce(begin(data), end(data), 0);
3 }
4 
```

在这个例子中，如果 n 很大，`ArraySumReduce` 和 `ArraySumBasic` 之间可以有好几倍的时间消耗差异（您可以运行参考程序来看到这一点），因此在允许乱序求和的时候，总是应当采用 `std::reduce`。

现在回到我们的递归话题来。一个基于经典递降法的朴素思路是：要求 n 个元素的和，可以先求前 $n - 1$ 个元素的和，然后将它和最后一个元素相加。这是一个递归算法，如下所示。

```

1 // ArraySumReduceAndConquer
2 int operator()(span<const int> data) override {
3     if (auto size { data.size() }; size == 0) {
4         return 0;
5     } else if (size == 1) {
6         return data[0];
7     } else {
8         return (*this)(data.first(size - 1)) + data[size - 1];
9     }
10 } 
```

这种将待解决问题分拆为一个规模减小的问题 + 有限个平凡的问题的思想，被称为**减治**（reduce-and-conquer）^[6]。在《数据结构》中使用减治思想，通常是将数据结构（这里是数组）分拆成几个部分，其中只有一个部分的规模和原数据结构的规模相关（减治项），其他部分的规模都是有界的（平凡项）。在上面的例子中，数组被分拆成了两个部分，前 $n - 1$ 个元素组成的部分是减治项，而最后一个元素是平凡项。虽然这个算法的正确性非常显然，但作为练习，您还是可以借助之前介绍的递降法去证明它的正确性。

循环可以被看成是减治的一种特殊形式。我们可以认为循环的第一次执行是平凡项（起始处减治），此后的执行是减治项；或者，我们也可以认为循环的最后一次执行是平凡项（结尾处减治），此前的执行是减治项。在下面的循环代码的结尾处减治，就对应了刚刚介绍的减治算法。您可以修改减治算法的形式，使其对应下面循环代码的起始处减治。

```

1 // ArraySumLoop
2 int operator()(span<const int> data) override {
3     int sum { 0 };
4     for (int x : data) {
5         sum += x;
6     }
7     return sum;
8 } 
```

下面介绍另一种不同的思路。我们可以设计这样的算法：先算出数组前一半的和，再算出数组后一半的和，最后把这两部分的和相加（这个算法成立的基础是加法结合律）。这也是一个递归算法，如下所示。

```

1 // ArraySumDivideAndConquer
2 int operator()(span<const int> data) override {
3     if (auto size { data.size() }; size == 0) {
4         return 0;
5     } else if (size == 1) {
6         return data[0];
7     } else {
8         auto mid { size / 2 };
9         return (*this)(data.first(mid)) + (*this)(data.last(size
10            - mid));
11    }

```

您可以选取合适的映射，将数据范围从数组映射到通常的良序集上，并完成正确性的证明。一个显然的映射是 $f(A) = A.size()$ 。上述算法设计中，我们将待解决问题拆分为多个规模减小的问题，这种思想称为**分治**（divide-and-conquer）。在《数据结构》中使用分治思想，通常是将数据结构（这里是数组）分拆为几个部分，每个部分的规模都和原数据的规模相关（分治项）。广义的分治也可以包含若干个平凡项。在上面的例子中，数组被分拆为了两个部分，每个部分的规模大约是原规模的一半。减治和分治是设计算法的重要思路，在本书中将广泛使用。这两种思想也能为您解决考试中遇到的算法问题提供强大的助力。

表 1.2 数组求和算法的耗时（单位：ms）

n	Basic	SIMD	减治递归	分治递归
10^2	0	0	0	0
10^4	0	0	栈溢出	0
10^6	1	0	栈溢出	36
10^8	145	20	栈溢出	3418

我们注意到，分治算法和普通算法相比，只是修改了加法的运算次序。我们不期待它能达到 `std::reduce` 的性能，但测试会发现，它竟然连基本算法的性能都远远不如。这是因为递归调用本身存在不小的开销。如果等价的迭代方法（在后续章节中，将讲解递归和迭代的相互转换）并不复杂，那么通常用迭代替换递归，以免除递归调用本身的性能开销。

1.4.8 实验：函数零点

以上两个例子都是建立在递归上的算法。很多算法可能并不包含递归；对这些算法做有限性检验，不是要排除无穷递归，而是要排除无限循环。下面展示了一个循环的例子，代码可以在 *ZeroPoint.cpp* 中找到。

我们考虑一个函数的零点，给定一个函数 $f(x)$ 和区间 (l, r) ，保证 $f(x)$ 在 (l, r) 上连续，并且 $f(l) \cdot f(r) < 0$ 。根据介值定理，我们知道 $f(x)$ 在 (l, r) 上存在至少一个零点。由于计算机中的浮点数计算有精度限制，我们只需要保证绝对误差不超过给定的误差限 ϵ 。为了简单起见，我们现在统一给定 $l = -1, r = 1, \epsilon = 10^{-6}$ 。

```

1 class ZeroPoint : public Algorithm<function<double(
2     double)>> {
3     static constexpr double limit_l { -1.0 };
4     static constexpr double limit_r { 1.0 };
5 protected:
6     static constexpr double eps { 1e-6 };
7     virtual double apply(function<double(double)> f, double l,
8         double r) = 0;
9 public:
10    double operator()(function<double(double)> f) override {
11        return apply(f, limit_l, limit_r);
12    }
13};

```

上面的例子展示了我们将 `Algorithm` 定义为仿函数的优势，我们可以在零点问题的基类中定义私有（private）成员来表示给定的初值 l 和 r ；另一个给定的初值 ϵ 在算法的实现中需要用到，则可以定义为受保护的（protected）成员。

函数的零点可以通过二分的方法取得，这个方法在高中的数学课程中介绍过。
如果您熟悉编程，应该可以自己实现一个版本。

```

1 // ZeroPointIterative
2 double apply(function<double(double)> f, double l, double r)
3     override {
4     while (r - l > eps) {
5         double mid { l + (r - l) / 2 };
6         if (f(l) * f(mid) <= 0) {
7             r = mid;
8         } else {
9             l = mid;
10        }
11    }
12    return l;
13}

```

尽管是经典的算法，但您仍然可以关注其中的一些细节。一个细节是我们定义 `mid` 为 $l + \frac{r-l}{2}$ 而不是 $\frac{l+r}{2}$ 。这两种写法都是推荐的写法。在整数的情况下，前者的优点是不会溢出，并保证结果接近 l （后者存在负数时无法保证，而这一点在写

二分算法时有时很致命)；后者的优点是少一次减法计算。浮点数的情况比整数简单很多，两种写法，甚至 $\frac{l}{2} + \frac{r}{2}$ (在整数的情况下不应该使用)都是可以的。

另一个细节是我们的判断条件为 $f(l) \cdot f(mid) \leq 0$ ，而没有用严格小于号。您可以自己思考，若改为严格小于，会发生什么样的变化？如果一时没有头绪，您可以实现这样的一个类，拿来和示例代码一起测试。作为一门工程学科，计算机学科非常需要您通过实验建立起来的经验。

分析循环问题的手段，和分析递归问题是相似的。递归函数的参数，在循环问题里就变成了循环变量。在处理上面的迭代算法时，首先找到循环的停止条件： $r-l < \epsilon$ 。这个条件里， ϵ 作为输入数据，在循环中是不变量，而 l 和 r 是循环中的变量。因此，使用递降法的时候可以将 ϵ 看成常量，而 l 和 r 作为递归参数。

定义映射 $f(l, r) = \lfloor \frac{r-l}{\epsilon} \rfloor$ 就可以将问题映射到通常良序集，后面的做法和前面几个例子基本相同，您可以自己完成正确性检验。请注意在证明结果正确性时要留意 $f(mid) = 0$ 的情况。从这个映射中，我们可以发现规定 ϵ 是必要的，即使不考虑**double**的位宽限制，计算机也无法保证在有限时间里找到精确解，只能保证找到满足精度条件的近似解。

这种“将循环视为递归”然后用递降法处理的方法，等价于将上面的迭代算法改写为以下与其等价的递归算法。

```

1 // ZeroPointRecursive
2 double apply(function<double(double)> f, double l, double r)
3     override {
4         if (r - l <= eps) {
5             return l;
6         } else {
7             double mid { l + (r - l) / 2 };
8             if (f(l) * f(mid) <= 0) {
9                 return apply(f, l, mid);
10            } else {
11                return apply(f, mid, r);
12            }
13        }
14    }

```

其通用做法是：找到循环的停止条件，然后将条件中出现的、在循环内部被改变的变量视为递归的参数，以此将循环改写为递归。当然，实际解题的时候犯不着费劲改写成递归再分析，用这个思路直接分析循环就可以了。

1.5 复杂度分析

复杂度(complexity)分析的技术被用于评价一个算法的效率。在考试中它出现的频率比正确性检验更高。在上文中提到过，一个算法的真实效率(运行时间、

占用的硬件资源)会受到所用计算机、操作系统以及其他条件的影响,因此无法用来直接进行比较。因此,进行复杂度分析时通常不讨论绝对的时间(空间)规模,而是采用**渐进复杂度**来表示其大致的增长速度。

1.5.1 复杂度记号的定义

以下给出复杂度记号的标准定义。

假设在问题规模为 n 的情况下,算法在某一计算机上执行的绝对时间单元(空间单元)的数量为 $T(n)$ 。

1. 对充分大的 n , 如果 $T(n) \leq C \cdot f(n)$, 其中 $C > 0$ 是和 n 无关的常数,那么记 $T(n) = O(f(n))$ 。
2. 对充分大的 n , 如果 $C_1 \cdot f(n) \leq T(n) \leq C_2 \cdot f(n)$, 其中 $C_2 \geq C_1 > 0$ 是和 n 无关的常数,那么记 $T(n) = \Theta(f(n))$ 。
3. 对充分大的 n , 如果 $C \cdot f(n) \leq T(n)$, 其中 $C > 0$ 是和 n 无关的常数,那么记 $T(n) = \Omega(f(n))$ 。

用 $O(\cdot)$ 、 $\Theta(\cdot)$ 和 $\Omega(\cdot)$ 记号表示的时间(空间)随输入数据规模的增长速度称为**渐进复杂度**,或简称**复杂度**。

从上述定义中可以得到,当 $T(n)$ 关于 n 单调递增并趋于无穷大时, $f(n)$ 是阶不比它低的无穷大量, $h(n)$ 是阶不比它高的无穷大量,而 $g(n)$ 是和它同阶的无穷大量。当然 $T(n)$ 并不一定单调递增趋于无穷大。一般而言,复杂度记号是在问题规模充分大的前提下,从增长速度的角度对算法效率的定性评价。

在“充分大的 n ”和“忽略常数”两个前提下,复杂度记号里的函数往往特别简单。比如,多项式 $\sum_{i=0}^k a_i n^i$ ($a_k > 0$) 可以被记作 $\Theta(n^k)$,因为充分大的 n 下,次数较低的项都可以被省略,忽略常数又使我们可以省略系数 a_k 。

有些教材为简略起见,只介绍了 $O(\cdot)$ 一个符号,这非常容易引起理解错误或混淆^[5]。在下一小节中会展示很多错误理解的例子。因此,即使您准备参加的考试中不要求另外两种复杂度记号,也希望您理解这些记号。在本书中,这三种复杂度记号都会使用。其中, $\Theta(\cdot)$ 记号包含了比 $O(\cdot)$ 更多的信息,如果您准备参加的考试只要求 $O(\cdot)$ 这一个记号,您可以在不引起歧义的前提下,在作答时将笔记中的 $\Theta(\cdot)$ 用 $O(\cdot)$ 代替。在只使用 $O(\cdot)$ 的文献中,通常 $O(\cdot)$ 实际上表达的含义也是 $\Theta(\cdot)$;而在严谨的文献中,除了 $O(1)$ 和 $\Theta(1)$ 没有区别外,其他情况下这两者都是被严格区分的。

除了这三种复杂度记号之外,还有少见的两种复杂度记号: $o(\cdot)$ 和 $\omega(\cdot)$ 。因为在科研生活里也极少使用,所以不进行介绍。需要指出,一些错误的理解甚至可能

成为408考试的标准答案，这种情况造成的失分是无法避免的，只能在考试中揣摩出题者的意图了。

在上面的定义中，引入了时间单元和空间单元的概念。因为渐进复杂度的记号表示中不考虑常数，所以这两个单元的大小是可以任取的。例如，一个时间单元可以取成：

- 一秒（毫秒，微秒，纳秒，分钟，小时等绝对时间单位）。
- 一个CPU周期。
- 一条汇编语句。
- 一次基本运算（如加减乘除）。
- 一次内存读取。
- 一条普通语句（不含循环、函数调用等）。
- 一组普通语句。

又例如，一个空间单元可以取成：

- 一个比特（字节、半字、字、双字等绝对单位）。
- 一个结构体（固定大小）所占空间。
- 一个页（参见《操作系统》）。
- 一个栈帧（参见《操作系统》）。

这些单位并不一定能直接地相互转换。比如，即使是同一台计算机，它的“一个CPU周期”对应的绝对时间也可能会发生变化（CPU过热时降频）。但是这些单位在转换时，转换倍率必然存在常数的上界。比如通常情况下，能正常工作的内存绝不可能需要多于 10^9 个CPU周期才能完成读取。

这个常数级别的差距，在复杂度分析里被纳入到了 C 、 C_1 、 C_2 中，而不会影响到渐进复杂度。因此，复杂度分析成功回避了硬件、软件、环境条件等“算法外因素”对算法效率的影响。

1.5.2 复杂度记号的常见理解误区

这一小节单独开辟出来，讨论和复杂度记号（尤其是 $O(\cdot)$ ）有关的注意点^[7]。由于复杂度记号总是作为一门学科的背景知识出现，您可能会没有意识到这是一个相当容易混淆的概念，从而陷入某些误区而不自知。

第一，**不可交换**。已知 $n^3 = O(n^4)$ ， $n^2 = O(n^4)$ ，那么，是否有 $n^3 = O(n^4) = n^2$ 呢？显然是不可能的。等于号“=”的两边通常都是可以交换的，但在复杂度记号这里并非如此。在进行复杂度的连等式计算时，始终需要记住：

1. 等式左边包含的信息不少于右边。（最核心的性质）
2. 复杂度记号本身损失了常数的信息。因此复杂度记号只能出现在等式的右侧。

如果出现在左侧，那么右侧也必须是复杂度记号。

3. 从 $\Theta(\cdot)$ 转换成 $O(\cdot)$ 或 $\Omega(\cdot)$ ，会损失一侧的信息。因此连等式中 $\Theta(\cdot)$ 只能出现在 $O(\cdot)$ 或 $\Omega(\cdot)$ 的左侧。只有一种情况除外，就是 $O(1) = \Theta(1)$ 。

例如， $2n^2 = \Theta(n^2) = O(n^3) = O(n^4)$ 是正确的。

第二，不是所有算法都可以用 $\Theta(\cdot)$ 评价。这个问题很容易从数学角度看出来，比如说 $T(n) = n \cdot (\sin \frac{n\pi}{2} + 1)$ ，就不存在“与它同阶的无穷大量”。正是因为这个原因，我们更多地使用表示上界的 $O(\cdot)$ ，而不是看起来可以精确描述增长速度的 $\Theta(\cdot)$ 。

第三，只有 $\Theta(\cdot)$ 才可以进行比较。已知算法 A 的复杂度是 $O(n)$ ，算法 B 的复杂度是 $O(n^2)$ ，那么，算法 A 的复杂度是否一定低于算法 B？

这是最容易误解的一处，我们可以说 $O(n)$ 的复杂度低于 $O(n^2)$ ，但切不能想当然认为算法 A 的复杂度低于算法 B。这是因为，尽管已知条件告诉我们“算法 B 的复杂度是 $O(n^2)$ ”，但已知条件并没有排除“算法 B 的复杂度同时也是 $O(n)$ 甚至 $O(1)$ ”的可能。类似地， $\Omega(\cdot)$ 也不可比较，只有表示同阶无穷大量的 $\Theta(\cdot)$ 有比较的意义。考试中也可能遇到需要比较复杂度的情况，这是纯粹的数学问题，因此不做展开。

计算机工程师对数学严谨性并不敏感。在相当大规模的人群中，对 $O(\cdot)$ 形式的复杂度进行比较已经成为了一种习惯。作为一门工程学科，当一种做法被人们普遍接受、成为共识的时候，通常就被认为是合理的，以至于人们忽视了，只有当 $O(\cdot)$ 事实上在表达 $\Theta(n)$ 的语义时，这样的比较才是成立的。如上一点所说，并非所有复杂度分析都能用 $\Theta(\cdot)$ 形式的结果，只有了解了严谨的说法，才能避免在阅读文献时理解上出现谬误。

第四，不满足对减法和除法的分配律。思考一下，是否有 $O(H_n) - O(\ln n) = O(H_n - \ln n) = O(1)$ 呢？这里记号 $H_n = \sum_{i=1}^n \frac{1}{i}$ 用来表示调和级数的部分和，在 $n \rightarrow \infty$ 时， $H_n - \ln n \rightarrow \gamma$ ，其中 γ 是 Euler 常数。这一性质在复杂度分析的领域非常重要，之后还会再次遇到。

答案是否定的。不论是三种复杂度记号中的哪一种，都不服从对减法的分配律。对加法和乘法，分配律是成立的，但是右侧的复杂度符号数量必须比左侧少（直观地理解是，复杂度记号包裹的范围越大，就会损失越多的信息）。比如 $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$ 。您可以自己证明这些公式。

第五，复杂度记号和情况的好坏无关。也就是说，尽管表示上下界，但 $O(\cdot)$ 和 $\Omega(\cdot)$ 不代表“最坏情况（worst case）”和“最好情况（best case）”。

情况（case） 表示和问题规模 n 无关的输入数据特征。比如说，我们想要在规

模为 n 的整数数组 A 中，寻找第一个偶数（找不到时返回 0），可以使用如下算法。

```

1 int operator()(span<const int> data) override {
2     for (auto a : data) {
3         if (a % 2 == 0) {
4             return a;
5         }
6     }
7     return 0;
8 }
```

容易发现，除了问题规模 n 之外，这 n 个整数自身的特征也会影响找到第一个偶数的时间。如果 $A[0]$ 就是偶数，则只进行了一次奇偶判断，可以认为 $T(n) = 1$ ；如果 A 中每一个元素都是奇数，则需要对每个元素都进行奇偶判断，可以认为 $T(n) = n$ 。对于同样的 n ，不同特征的输入数据会得到不同的 $T(n)$ 。

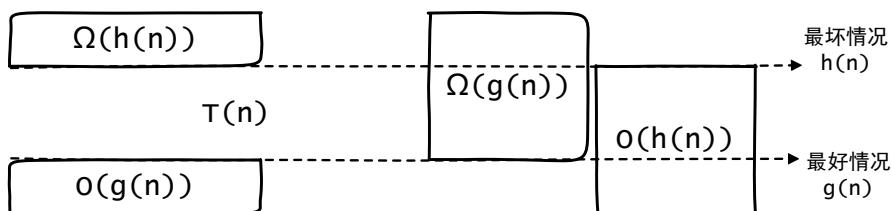


图 1.1 最好情况和最坏情况的评估

因此，在引入了情况的时候， $T(n)$ 不再是一个准确的值而是一个范围，它的下限对应了最好情况，上限对应了最坏情况。如果设 $T(n)$ 在最好情况下为 $g(n)$ ，最坏情况下为 $h(n)$ ，那么对 $g(n)$ 和 $h(n)$ 可以分别做复杂度分析，得到它在最好情况和最坏情况下的复杂度。在这个复杂度分析的过程中，三种符号都是可以使用的。也正是因为情况和复杂度记号无关，所以在只使用 $O(\cdot)$ 的书上，无论是最好、最坏还是平均都可以使用这个记号。

不同的符号表达的侧重点是不同的。如图1.1所示，当我们使用 $O(\cdot)$ 描述最好情况时，可以体现出“最好”究竟有多好，而 $\Omega(\cdot)$ 描述最好情况则做不到这一点。相应地， $\Omega(\cdot)$ 才可以体现出“最坏”有多坏。因此，在无法用准确的 $\Theta(n)$ 表达时，我们通常使用 $O(\cdot)$ 描述最好，用 $\Omega(\cdot)$ 描述最坏。

1.5.3 实验：判断 2 的幂次

下面几个小节，将通过一些简单的算法，介绍复杂度分析的基本方法。更多的复杂度分析将穿插在整本书中。在这一节我们讨论，判断一个数是否为 2 的幂次，代码可以在 *IsPower2.cpp* 中找到。

```

1 class IsPower2 : public Algorithm<bool(int)> {};
```

这个问题并不难，一种常规的实现是：

```

1 // IsPower2Basic
2 bool operator()(int n) override {
3     return n > 0 && (n & (n - 1)) == 0;
4 }
```

这里判断 $n > 0$ 是为了稳健性。您可以从二进制数的特征出发，证明上面这个算法的正确性。对于有经验的程序员来说，写出上面的实现易如反掌，但并不是每一个人都能记住它。C++20 提供了一个专门的函数，判断一个二进制数据是否恰好只有一个非 0 比特位（对于整数来说，这等价于 2 的幂次）。

```

1 // IsPower2SingleBit
2 bool operator()(int n) override {
3     return n > 0 && has_single_bit(static_cast<unsigned>(n));
4 }
```

这里 `std::has_single_bit` 只接受无符号数，所以需要先判断是否大于 0，再进行静态强制转换。`static_cast` 是 C++ 推荐的写法，您也可以采用 C 语言风格的强制类型转换，但 C++ 风格的写法是更加类型安全的，它会在编译期检查转换是否合法。

这种简单的检查方法只需要常数次的计算，时间复杂度和空间复杂度都是 $O(1)$ 。对于既不熟悉二进制位运算，又不了解标准库的程序员，可能会写出下面的算法来解决这个问题：

```

1 // IsPower2Recursive
2 bool operator()(int n) override {
3     if (n % 2 == 1) {
4         return n == 1;
5     } else {
6         return (*this)(n / 2);
7     }
8 }
```

这个算法涉及递归，显然它的时间复杂度不再是 $O(1)$ 。为了证明上述算法的有限性，可以采用前述的递降法，如构造 $f(n) = \max(d \mid n \% 2^d = 0)$ 为满足 2^d 整除 n 的最大的 d ，从而映射到通常良序集。当 n 为奇数的时候， $f(n) = 0$ 到达边界值。但对这种简单的问题，也可以直接显式地计算 $T(n)$ ，即函数体的执行次数。显式地计算出有限的 $T(n)$ ，也就在复杂度分析的同时“顺便”证明了算法的有限性。

设 $n = k \cdot 2^d$ ，其中 k 为正奇数， d 为自然数。则 $T(n) = T(k \cdot 2^d) = 1 + T(k \cdot 2^{d-1}) = 2 + T(k \cdot 2^{d-2}) = \dots = d + T(k) = d + 1 = O(d) = O(\log(n/k)) = O(\log n)$ 。另一边的 $\Omega(1)$ 是显然的。

上面这个 $T(n)$ 的计算过程，本质上仍然是使用了递降法，将 $T(\cdot)$ 的参数不断递降到递归边界（正奇数 k ），思路和正确性检验的递降法是一致的，它利用了 $T(\cdot)$

的递归式去显式地计算这个值。这里注明一点：因为计算机领域广泛使用二进制，所以未指定底数的对数符号“ \log ”，底数默认为 2。而因为换底公式的存在，在复杂度记号下无论使用什么底数都没有区别。您可能会发现，这个算法在给定错误值，如 0 和 -1 的时候会陷入无限递归，这属于稳健性问题，您可以自己改正它。

同样，可以显式地计算下面这个算法的 $T(n)$ 。

```

1 // IsPower2Iterative
2 bool operator()(int n) override {
3     int m { 1 };
4     while (m < n) {
5         m *= 2;
6     }
7     return m == n;
8 }
```

上面的两个实现，并不是同一算法的迭代形式和递归形式。根据上面的分析可以看到，递归算法的时间复杂度为 $O(\log n)$ 和 $\Omega(1)$ ，而迭代算法为 $\Theta(\log n)$ ，您可以自己证明这一点。此外，这个迭代算法也存在稳健性问题，如当输入 $n = 2^{31} - 1$ 时，就会陷入无限循环，您可以想办法改正它。

如果您掌握了对循环和递归相互转换，您可以实现 IsPower2Recursive 的迭代版本，以及 IsPower2Iterative 的递归版本。这两个算法的区别在两个方面。第一，迭代版本的迭代方向是“递推”而不是“递降”；第二，递归边界和循环终止条件不对应。第二点是更加本质的区别，它指示了在最好情况下（ n 为奇数时），时间复杂度不同的原因。

另一方面，二者的空间复杂度也有所不同。在迭代算法中，只引入了 1 个临时变量 m ，因此空间复杂度是 $O(1)$ 。而在递归算法中，看似一个临时变量都没有引入，空间复杂度也应该是 $O(1)$ ，实则不然。递归算法在达到递归边界之前，每一次递归调用的函数都在等待内层递归的返回值。到达递归边界、判断完成后，这一结果被一级一级传上去，途中调用函数占据的空间才会被销毁（参见《操作系统》）。

因此，对于递归算法，递归所占用的空间在复杂度意义上等于最大递归深度。IsPower2Recursive 的空间复杂度同样是 $O(\log n)$ 和 $\Omega(1)$ 的。现代编译器可能会将它优化成迭代形式，但在《数据结构》的解题过程中，永远不要考虑编译器优化问题。

考试时往往更加重视时间复杂度，因为现代计算机的内存通常足够普通的程序使用，而且《数据结构》中涉及的大多数算法，空间复杂度要么显而易见、要么能在计算时间复杂度的时候顺便算出来。但空间效率仍然是衡量数据结构的重要指标。这个空间效率不单指空间复杂度，也包含被复杂度隐藏下去的和数据结构相关的常数。比方说，如果在同一计算机上，数据结构 A 比数据结构 B 的常数低

10倍，那么它就能存放10倍的数据，这个优势是非常大的——即使二者空间复杂度一致。

1.5.4 实验：快速幂

在上一小节的实验中，如果我们认为 n 是“问题规模”，那么就不能说奇数的情况是“最好情形”，因为此时没有“情形”的概念；这使得我们不能说最好 $O(1)$ 和最坏 $\Omega(\log n)$ 。但如果认为 n 的位宽 $\log n$ 是“问题规模”，则可以这么说。这从侧面反映了用位宽，也就是“输入规模”来表示问题规模的好处。

邓书上也建议这样表示问题规模，并使用了快速幂作为举例阐述这个问题。下面，本书也将快速幂作为一个实验，来深入分析“问题规模”这个概念。快速幂是一个解决求幂问题的算法。求幂问题中，我们输入两个正整数 a 和 b ，输出 a^b 。代码可以在Power.cpp中找到。

```
1 class PowerProblem : public Algorithm<int(int, int)> {};
```

基本的求幂算法，和求和类似：

```
1 int operator()(int a, int b) override {
2     int result { 1 };
3     for (int i { 0 }; i < b; ++i) {
4         result *= a;
5     }
6     return result;
7 }
```

您可以毫不费力地看出这个算法的时间复杂度是 $\Theta(b)$ 。当 b 比较大时，这个算法的时间效率很低。这是因为，在计算 a^b 的时候，采用的递推式是 $a^b = (a(a(a(a(a \dots (a \cdot a) \dots)))))$ ，像普通的 b 个数相乘一样简单地循环，没有利用 a^b 的在计算上的自相似性。

事实上，我们可以将这 b 个 a 两两分组。如果 b 是偶数，那么恰好可以分成 $\frac{b}{2}$ 组，于是我们只需要计算 $(a^2)^{\frac{b}{2}}$ 。奇数的情形需要乘上那个没能进组的 a 。于是，我们通过1到2次乘法，将 b 次幂问题化归到了 $\frac{b}{2}$ 次幂问题。

```
1 int operator()(int a, int b) override {
2     if (b == 0) {
3         return 1;
4     } else if (b % 2 == 1) {
5         return a * (*this)(a * a, b / 2);
6     } else {
7         return (*this)(a * a, b / 2);
8     }
9 }
```

容易证明这个算法的时间、空间复杂度均为 $\Theta(\log b)$ 。示例代码还提供了它的迭代版本，您也可以试着自己将它改为迭代。通常， $\Theta(\log b)$ 复杂度的求幂算法都称为快速幂，除了上面介绍的这种实现（借助 $a^b = (a^2)^{\frac{b}{2}}$ ），还有另一种实现（借助 $a^b = (a^{\frac{b}{2}})^2$ ），您也可以试着去实现它。

现在问题来了：这个算法的问题规模是什么？

最自然的想法是，它的问题规模是 b 。上述两种算法的复杂度都可以用这个问题规模表示；至于 a 的值，则被归入“情况”的范畴，并且它也不会影响到这两个算法的复杂度。

另一种学说认为，它的问题规模是 $\log b$ 。这一学说的依据是：问题规模应当是描述这一问题的输入需要的数据量（即输入规模）。在这个问题中，要描述问题中的 b ，在二进制计算机中需要 $\log b$ 个比特的数据，所以问题规模是 $\log b$ 。

这两种方法各有利弊。第一种学说的优点在于形象直观，容易理解；第二种学说的优点则在于有迹可循，定义统一。比如，如果让快速幂中的 b 允许超出 `int` 限制，比如使用 Java 中的 `BigInteger`，那么 b 势必用数组或者类似的数据结构表示。这个时候，因为实质上输入的是一个数组，即使是“直观派”也会倾向于将“数组的大小”也就是 $\log b$ 作为问题规模。于是，“直观派”无法让问题规模的定义在扩展的情况下保持统一，而“输入规模派”可以做到这一点。

在大多数情况下，这两种学说并无分歧。包括邓书在内的大部分教材支持“输入规模派”^[6]，而较早的书上没有讨论这个问题^[5]，通常这个问题对解题也没有任何影响。

1.5.5 实验：复杂度的积分计算

递归算法的复杂度分析在之后还会讨论更多技术。在比较简单的试卷上，命题者通常不会用递归算法命题，而是使用更为简单的迭代算法，命制选择或填空题。这类题目的共同点是迭代的次数非常清晰。比如前面的 `IsPower2Iterative`，可以一眼就看出来迭代的次数是 $\Theta(\log n)$ 。

这种题目通常可以用积分计算，而不需要用递降法。下面是一个没什么实际意义的例子。可以在 `ComplexityIntegral.cpp` 中找到它，并通过测试验证下面计算得到的时间复杂度。

```

1 int f(int n) {
2     int result = 0;
3     for (int i { 1 }; i <= n; ++i) {
4         for (int j { 1 }; j <= i; ++j)
5             for (int k { 1 }; k <= j; ++k)
6                 for (int l { 1 }; l <= j; l *= 2)
7                     result += k * l;

```

```

8     }
9     return result;
10    }

```

很容易看出，要分析该算法的时间复杂度，只需要算循环体被执行了几次，也就是计算：

$$T(n) = \sum_{i=1}^n \sum_{j=1}^i \sum_{k=1}^j (1 + \lfloor \log j \rfloor) = \sum_{i=1}^n \sum_{j=1}^i j \cdot (1 + \lfloor \log j \rfloor)$$

要显式地求出这个和式非常困难。幸运的是，需要求出的是复杂度而不是精确的值，常数和小项都可以在求和过程被省略掉。这给了您解决它的手段：离散的求和问题可以直接转换成连续的积分问题。

如果我们只需要一个渐进复杂度，那么积分也不需要真的去求，每次积的时候直接乘上一个线性量就可以。如果想要估算常数，则求积分的时候可以每一步只保留最高次项。

比如，上面的求和式可以计算如下：

$$\begin{aligned} T(n) &= \sum_{i=1}^n \sum_{j=1}^i j \cdot (1 + \lfloor \log j \rfloor) = O\left(\int_0^n \int_0^x y(1 + \log y) dy dx\right) \\ &= O\left(\int_0^n \int_0^x y \log y dy dx\right) = O\left(\int_0^n x^2 \log x dx\right) = O(n^3 \log n) \end{aligned}$$

其中，第一步是将求和符号转换成积分；消去积分符号的过程是做了两次“乘上一个线性量”的操作。可以看出对于 i 的分析来说，可以直接使用更简单的 $\log j$ 替代实际执行次数 $1 + \lfloor \log j \rfloor$ 。如图1.2所示，该函数的实际执行时间确实和 $n^3 \log n$ 近似成线性关系。

如果想要估算常数，只需要：

$$\begin{aligned} T(n) &= \sum_{i=1}^n \sum_{j=1}^i j \cdot (1 + \lfloor \log j \rfloor) \sim \int_0^n \int_0^x y(1 + \log y) dy dx \\ &\sim \int_0^n \int_0^x y \log y dy dx \sim \int_0^n \frac{1}{2} x^2 \log x dx \sim \frac{1}{6} n^3 \log n \end{aligned}$$

这里最后的 $\frac{1}{6}$ 就是常系数；在一些难度较高的算法分析题中，常系数可能会被用到。

1.5.6 多重循环复杂度的简单估算

上面这种积分的方法是计算此类循环算法的时间复杂度及其常数的一般方法。在熟练之后，可以用一些小技巧来处理。下面介绍一些小技巧；当然，如果时间充

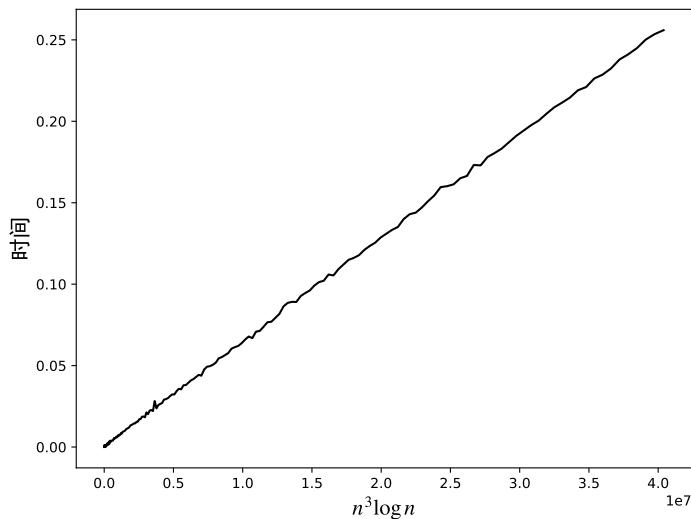


图 1.2 复杂度验证结果

足，还是建议用积分方法去严格地进行计算。

在这类循环算法中，每一层循环主要有下面这几种典型的形式：

```

1 for (int i { 1 }; i <= n; ++i) { /* (1) */ }
2 for (int i { 1 }; i <= n; i *= 2) { /* (2) */ }
3 for (int i { 2 }; i <= n; i *= i) { /* (3) */ }
```

1. 对于线性增长的循环，无论它出现在什么位置，都会为复杂度增加一个线性项 n 。
2. 对于指数增长的循环，如果它出现在最内层（或可以被交换到最内层，下同），那么会为复杂度增加一个对数项 $\log n$ ；如果它不出现在最内层，通常不会影响复杂度。
3. 对于幂塔增长的循环，如果它出现在最内层，那么会为复杂度增加一个迭代对数项 $\log^* n$ （关于迭代对数，您不需要了解更多）；如果它不出现在最内层，通常不会影响复杂度。

当 (2) (3) 出现在非最内层时，通常不会影响复杂度。这一点看起来没有那么显然，甚至很容易被忘记。我们用一个典型的例子来说明上面的 (2)。(3) 的情况是类似的。

```

1 int f(int n) {
2     int result = 0;
3     for (int i { 1 }; i <= n; i *= 2) {
4         for (int j { 1 }; j <= i; ++j)
5             for (int k { 1 }; k <= j; ++k)
6                 for (int l { 1 }; l <= j; l *= 2)
7                     result += k * l;
8     }
```

```

9     return result;
10    }

```

上面的这个程序，和本节一开始展示的例子相比，只有最外层 i 的循环，从线性递增改成了指数递增。以下通过积分方法解决此问题的方法。

$$\begin{aligned}
T(n) &= \sum_{i=1}^{\lfloor \log n \rfloor} \sum_{j=1}^{2^i} j \cdot (1 + \lfloor \log j \rfloor) = O\left(\int_0^{\log n} \int_0^{2^x} y(1 + \log y) dy dx\right) \\
&= O\left(\int_0^{\log n} \int_0^{2^x} y \log y dy dx\right) = O\left(\int_0^{\log n} 4^x \cdot x dx\right) \\
&= O\left(\int_0^n t^2 \cdot \log t \cdot \frac{dt}{t}\right) = O(n^2 \log n)
\end{aligned}$$

可以发现，我们在最后一步进行换元 $t = 2^x$ ，此时由于 $dx = \frac{dt}{t \ln 2}$ 会引入一个分母（一次项），该分母和积分引入的、同样是一次的分子会抵消，抵消的结果就是这一层积分并不会升幂。换句话说，在上面这个程序中，指数递增的外层循环 i 是否存在，对时间复杂度没有影响。

1.6 本章小结

本书的小结部分是不全面的，我只会介绍每一章的核心要点。如果您认为有必要，应当在自己的笔记上进行更加全面、更加适合您本人学习路径的总结。

本章的核心内容是**递归**的思想方法。

1. 您可以从超限归纳法和递降法的角度，理解递归思想的数学背景。
2. 您可以利用递归的思路分析迭代算法，将迭代终止条件和递归边界联系起来，认识到递归算法和迭代算法的等效性。
3. 您了解了利用递归的思想进行正确性检验和复杂度分析的技术。
4. 您了解了减治和分治这两种经典的递归设计方法。

第 2 章 向量

2.1 线性表

线性表是指相同类型的有限个数据组成的序列。本书按照邓书的方法，将线性表分为**向量**（vector）和**列表**（list）两种形式^[6]，分别对应C++中的`std::vector`和`std::list`。在另一些教材中，这两个词被称为**顺序表**和**链表**^[5]，分别对应Java里的`ArrayList`和`LinkedList`。向量（顺序表）和列表（链表）这两对概念通常可以混用。向量和列表代表着两种最基本的数据结构组织形式：**顺序结构**和**链式结构**。在这一节，将首先让读者对这两种组织形式有一个基本的印象。

首先，我们思考这样的问题：抛开顺序结构或链式结构不谈，线性表这个概念本身具有怎样的性质？这将会指导我们设计`AbstractLinearList`这个类。

对比它的基类`DataStructure`。一般的数据结构的定义中，“某种结构化的形式”，在线性表这里被具体化为了“序列”。既然是序列，那么它会具有头和尾，会具有“第*i*个”这样的概念；每个元素有它在序列上的“上一个”和“下一个”元素。这是一个朴素的想法，从数学角度容易理解。但从计算机的角度，请您思考这样的问题：“元素”应该用什么表示？我们应该如何找到序列中的一个元素呢？

把计算机中的内存想像为一座巨大的旅馆，线性表是一个居住在其中的旅游团。现在旅游团预定了一大片连续的房间，比如，从1000到1099，并且让第1个游客住在1000，第2个游客住在1001，以此类推，那么我们就很方便地可以知道第*i*个游客的房间号。这种情况下，我们只需要知道游客的序号，就能知道它们居住的房间号。

但并不是每个人都会按部就班地居住，一些人可能喜欢阳光，一些人可能想和伙伴们做邻居，于是，这些游客开始交换房间。房间被交换之后，我们再也无法直接知道第*i*个游客的房间号。一个可能的想法是，从1000到1099每一个房间都敲敲门。这种方法虽然可行，但无疑是低效的。

更加糟糕的是，旅游团可能没订到连续的房间，游客们散落在旅馆的各处。因为我们不可能像推销员那样每个房间都敲门（这会被赶出去的），所以再也无法找到我们想要的第*i*个游客了。为了应对这种情况，旅游团的导游往往会记录下各个游客所在的房间号，以便能够找到他们。

在上面这个比喻中，我们可以看到，如果数据结构中的元素被连续地储存，那么我们可以通过它们的序号（在邓书中，称为秩，rank）来找到它们；如果数据结构中的元素并没有被连续地储存，则我们只能通过它们的位置（position）来找到

它们。上面的三种情况，分别是地址连续、且地址和秩相关的顺序结构（向量）；地址连续、但地址和秩无关的静态链式结构（静态链表）；地址不连续、也和秩无关的动态链式结构（动态链表）。图2.1示意了三种结构的区别。

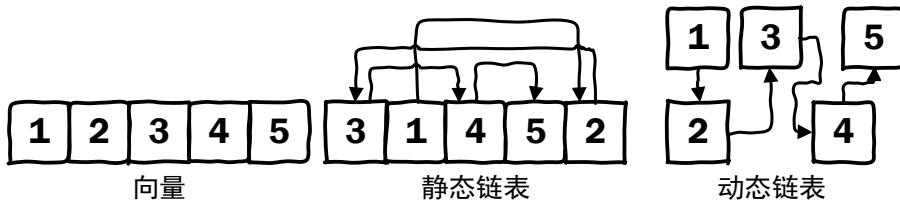


图 2.1 向量、静态链表和动态链表的对比

显然，如果发生了第二种情况，导游通常还是会选择记录房间号而不是逐个敲门。那么既然没有省事，也就没有必要预定一大片房间了。因为旅馆老板，也就是操作系统，可能会乘机宰客。比如，旅游团一次定了 100 个房间，但中途有 50 人提前结束了旅行。由于旅游团定的是整单，老板不允许单独退这 50 个人的房间。于是，旅游团要么承担 50 间空房的代价（空间损失）；要么再定 50 个房间，请剩下的 50 人搬到新房间住（时间损失），然后把原来的 100 个房间一并退掉。

这个例子展示了静态链表是一个不实用的数据结构，因此在邓书中它被删除。在包括静态链表的教材中，它也不是重点的一部分。我们将聚焦在向量和动态链表（列表）上。如前所述，向量和列表里定位元素的方法是不同的。向量是循秩访问，而列表则循位置访问。因此，我们设计的线性表抽象类中需要反映这一点。

```

1 template <typename T, typename Pos>
2 class AbstractLinearList : public DataStructure<T> {
3 public:
4     virtual T& get(Pos p) = 0;
5     virtual void set(Pos p, const T& e) = 0;
6
7     virtual Pos insert(Pos p, const T& e) = 0;
8     virtual Pos find(const T& e) const = 0;
9     virtual T remove(Pos p) = 0;
10    virtual void clear() = 0;
11 }

```

如果您查看 `AbstractLinearList.ixx`，还会看到一些其他的方法。正如序言中所说的那样，那些和《数据结构》研究的内容关系不大的方法将在书里被隐藏。上面的代码中只展示了一些重要的方法。

1. 根据位置 p ，访问或修改在位置 p 上的元素。
2. 将元素插入到位置 p 上。
3. 在线性表中查找一个给定的元素，返回它所在的位置。
4. 删除位置 p 上的元素，以及将整个线性表清空。

您可能会觉得，这个线性表类中包含的内容过多或者过少。如果您感兴趣，我也非常鼓励您实现自己的抽象类模板。本书给出的模板保证您在不参与设计细节的情况下能完成本书设计的有趣实验，您可以随时用自己实现的程序替换其中的一部分。如果您打算这么做，请记得使用版本控制程序或备份，保证您可以回到程序能正常运行的时刻。

2.2 向量的结构

向量 (vector) 是一个基于**数组** (array) 的数据结构，它在内存中占据的是一段连续的空间。**C++** 建议更多地使用标准库中的向量 `std::vector` 代替数组。向量和数组相比，其最重要的区别在于它是运行时可变长的，而在其他使用上，二者基本可以等同。因此，向量上的算法可以很容易地修改为数组上的算法（即使不使用 `std::span`）。在 408 中通常不考虑可变长这一性质，此时讨论的顺序表就是数组，但您同样可以用本章中介绍的向量算法解决相关题目。

作为一种基于数组的线性表，向量的元素次序和数组的元素次序相同。如果一个向量 V 基于长度为 n 的数组 A 构建，那么向量 V 的第 i 个元素就是 $A[i]$ 。我们知道，C 语言的数组可以视为指针，于是向量 V 的第 i 个元素的地址就是 $A + i$ 。

向量里的元素数量 n ，称为向量的**规模** (size)；而向量所占有的连续空间能够容纳的元素数量 m ，称为向量的**容量** (capacity)。这两者通常是不同的，规模必定不大于容量，而在不超过容量的前提下，向量的规模可以灵活变化，从而赋予了它比数组更高的灵活性。前面的那个旅行团的例子也可以帮助理解规模和容量的关系。一个 n 个人的旅行团订了连续的 m 个房间，这里 m 可以大于 n ，这样，如果旅行过程中有新人加入团队，他们就可以直接加入到已经预定的连续房间中来。

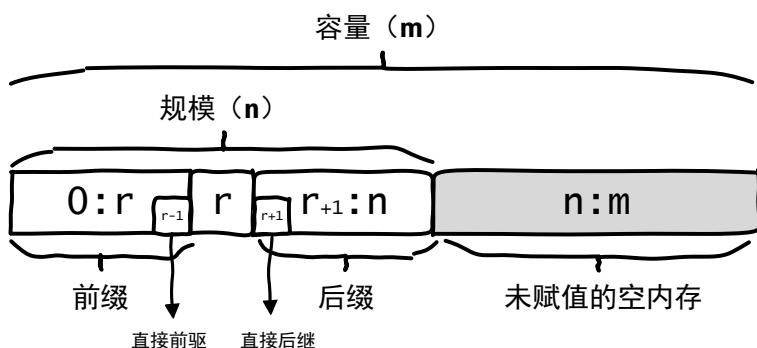


图 2.2 向量的基本概念

如图2.2所示，对于向量中的每一个元素 $V[i]$ 来说，它前面的元素称为它的**前驱** (predecessor)，它后面的元素称为它的**后继** (successor)。特别地，和它位置相

邻的前驱，也就是直接前驱为 $V[i - 1]$ ，相应地，直接后继为 $V[i + 1]$ 。所有的前驱构成了前缀（prefix），也就是 $V[0:i]$ ；所有的后继构成了后缀（suffix），也就是 $V[i + 1:n]$ 。本书中我们用 $V[a:b]$ 来简记 $V[a], V[a + 1], \dots, V[b - 1]$ 这个子序列，这是 Python 的切片（slice）语法，借助它可以简化很多叙述，尤其在记忆一些比较复杂的算法时很有用。

2.3 循秩访问

了解了向量的定义之后，我们将之前实现的线性表抽象类进行细化，构建向量的抽象类 `AbstractVector`。随后本书会讲解一个示例实现，您可以自行继承 `AbstractVector` 进行实现，因为那些不需要您关注的方法都已经在这个抽象类中实现。比如说迭代器，对于读者而言实现迭代器会花费比较多的时间，因此，尽管示例代码中实现了一个迭代器以使我们的向量能够支持 *STL* 里的算法，但迭代器被排除在基本实验体系之外。我们的目标是学习数据结构，而不是复现 *STL*。如果您感兴趣，可以自己查看示例代码进行学习；也欢迎您为本书的示例代码提出建议。

向量的核心特征是循秩访问。称元素 $V[i]$ 在向量 V 中的序号 i 为它的秩（rank）。对于建立在数组 A 上的向量 V ，因为 V 和 A 的元素次序是一致的，所以 $V[i] = A[i] = *(A + i)$ 。因此，只要知道一个元素的秩，就可以在 $O(1)$ 的时间内访问该元素。通常，我们用 `size_t` 类型存储下标。为了强调向量是循秩访问的，我们给它一个别名。

```
1 using Rank = size_t;
```

接下来，我们开始构筑向量抽象类。首先，除了在 `DataStructure` 里定义的规模 `size` 之外，我们还需要定义容量 `capacity`。同时，因为向量是可变长的，所以规模和容量都是可以变化的，还需要两个修改它们的方法。有了修改规模的方法，线性表里的 `clear` 就可以直接用 `resize(0)` 实现。其次，我们需要构筑循秩访问的体系，重写 `AbstractLinearList` 里面的 `get` 和 `set` 方法。

```
1 template <typename T>
2 class AbstractVector : public AbstractLinearList<T, Rank> {
3 protected:
4     virtual T* data() = 0;
5 public:
6     virtual size_t capacity() const = 0;
7     virtual void reserve(size_t n) = 0;
8     virtual void resize(size_t n) = 0;
9
10    T& get(Rank r) override {
11        return data()[r];
12    }
13}
```

```

12 }
13 void set(Rank r, const T& e) override {
14     data()[r] = e;
15 }
16 void clear() override {
17     resize(0);
18 }
19 };

```

操作底层内存的时候，因为和所有权无关，所以不能使用智能指针，只能使用裸指针。为了避免裸指针被外部获取，我们将向量的获取首地址方法`data()`设置为受保护的，而公共方法只会返回引用。

现在，我们已经拥有了一个抽象类`AbstractVector`，它还缺少下面这些组件的实现：获取规模、容量和首地址的方法；修改规模和容量的方法；插入、删除、查找的方法。如果您打算自己实现向量类，只需要在抽象类的基础上补充它们即可；您也可以继承本书提供的示例向量类，重写其中的部分方法。我们的示例类会从下面开始。

```

1 template <typename T>
2 class Vector : public AbstractVector<T> {
3     std::unique_ptr<T[]> m_data { nullptr };
4     size_t m_capacity { 0 };
5     size_t m_size { 0 };
6
7     T* data() override { return m_data.get(); }
8 public:
9     size_t capacity() const override { return m_capacity; }
10    size_t size() const override { return m_size; }
11 }

```

C++14起，允许用户使用智能指针管理数组，因此我们使用`std::unique_ptr`来申请内存。它的主要好处是不需要在析构函数里手动释放，减少了手动管理内存的麻烦。本书中若无特殊情况，将总是使用智能指针来表示所有权，避免在任何地方使用`delete`关键字释放内存。

2.4 向量的容量和规模

2.4.1 初始化

当我们建立一个新的数据结构的时候，有几种情况是比较典型的。在这里，以向量为例展示它们，后面讨论其他的数据结构的时候不再赘述。

零初始化。即，生成一个空的数据结构。对于向量来说，这应该包含一个大小为0的数组，并把容量和规模都赋值为0。然而，C++不支持大小为0的数组，所

以只能把`data`赋值为`nullptr`, 就像在上一节中展示的默认值那样。

指定大小的初始化。即, 给定 n , 生成一个规模为 n 的数据结构, 其中的每个元素都采用默认值 (即元素采用零初始化)。对于向量而言, 可以申请一片大小为 n 的内存, 如下面的代码所示。

```
1 Vector(size_t n) : m_data { std::make_unique<T[]>(n) },
    m_capacity { n }, m_size { n } {}
```

复制初始化。即, 给定相同数据结构的一个对象, 复制该对象里的所有数据及数据之间的结构化关系。对于向量来说, 只需要在申请大小为 n 的内存之后, 将给定的向量的元素逐一复制进来即可, 如下面的代码所示。注意这里在初始化器中显式调用了上面的“指定大小的初始化”的构造函数, 为向量进行了初步的初始化, 然后再把另一个向量的数据复制进来。

```
1 Vector(const Vector& rhs) : Vector(rhs.m_size) { std::copy_n(
    rhs.m_data.get(), rhs.m_size, m_data.get()); }
```

移动初始化。即, 给定相同数据结构的一个对象, 将该对象里的所有数据及数据之间的结构化关系移动到当前对象处。**移动** (move) 语义和**复制** (copy) 有显著的不同, 因为在移动之后, “被移动”的对象失去了对数据的所有权, 我们永远不会从被移动后的对象里访问那些数据。下面展示了向量移动初始化的一个例子。

```
1 Vector(Vector&& rhs) noexcept : m_data { std::move(rhs.m_data) }
    , m_capacity { rhs.m_capacity }, m_size { rhs.m_size } {
2     rhs.m_capacity = 0;
3     rhs.m_size = 0;
4 }
```

通过对智能指针调用`std::move`, 我们可以在常数时间内, 将被移动对象的数据转移到新对象里。可以看出, 被移动之后, `rhs`的数据区被复位为空指针, 规模和容量都被置 0, 就像它被零初始化了一样, 成为了一个空向量。

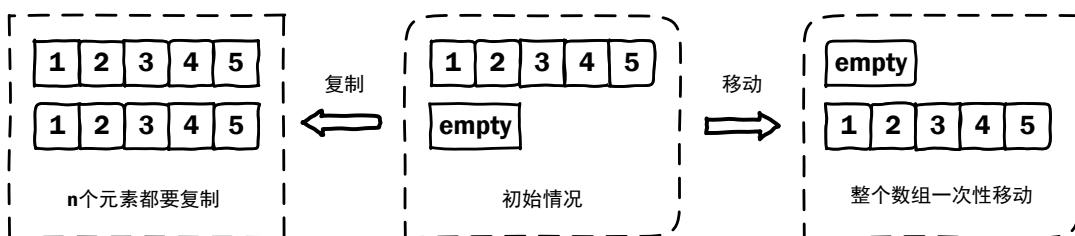


图 2.3 复制和移动的区别

从图2.3中可以直观了解到复制和移动的语义区别。当数据结构里的元素数量很多时, 逐元素地复制是一项复杂、琐碎、漫长的工程, 而移动则是一项简单、整体、快速的工作。在本书的示例代码中, 我们经常会给同一个函数提供一个复制版

本和一个移动版本。比如，对于插入（insert），我们定义一个插入`const T&`类型的方法用于复制（不破坏源），又定义了一个插入`T&&`类型的方法用于移动（破坏源）。这些方法之间往往只有一个或几个`std::move`的区别，因此本书通常省略移动版本的方法，而只展示复制版本的方法。尽管如此，在您的日常编程中需要时刻注意，只要复制和移动的时间成本有可能相差比较远，就应该同时定义并实现复制和移动两个版本的方法，而不能只实现复制。

需要注意的是，析构函数、复制构造函数、移动构造函数、复制赋值运算符、移动赋值运算符这 5 个函数，一旦显式定义其中的一个（比如想要定义复制构造函数），编译器就不会生成其他的函数。处于“一荣俱荣，一损俱损”的关系。因此，我们在日常编程的时候，通常选择不实现它们中间的任意一个函数（0 原则，rule of zero），因为日常编程的时候，通常都使用的是 STL 对象，而 STL 里已经将这些功能实现了。但是，在我们实现一些比较底层的结构时候，没法依靠 STL 里的实现，需要实现这 5 个函数中的一个或几个。此时，就必须要将所有的 5 个函数实现（5 原则，rule of five）。我们可以使用`=default`来显式使用自动生成的函数，但如果我们不显式说明它，这些函数将不会被包含在这个类中。

初始化列表初始化。即，使用初始化列表`std::initializer_list`对数据结构进行初始化。初始化列表也是一个典型的线性容器，可以直接使用 STL 方法复制。

```
1 Vector(std::initializer_list<T> ilist) : Vector(ilist.size())
2 {
3     std::move(ilist.begin(), ilist.end(), m_data.get());
}
```

支持初始化列表初始化之后，我们就可以用下面的形式来初始化一个向量。

```
1 Vector V {1, 2, 3};
```

2.4.2 装填因子

设向量的容量为 m ，规模为 n ，则称比值 $\frac{n}{m}$ 为**装填因子**（load factor）。正常情况下，这是一个 $[0, 1]$ 之间的数。装填因子是衡量向量效率的重要指标。

1. 如果装填因子过小，则会造成内存浪费：申请了巨大的数组，但其中只有少量的单元被向量中的元素用到，其他单元都被闲置了。
2. 如果装填因子过大（超过 1），则会引发数组越界，造成段错误（segmentation fault）。

刚开始的时候，装填因子一定是在 $[0, 1]$ 之间的。但因为数组的容量 m 是固定的，而向量的规模 n 是动态的，所以一开始分配的 m 可能后来会不够用，从而产

生装填因子大于 1 的问题，此时就需要令 m 增大，这一操作称为扩容（expand）。由于 408 不讨论变长顺序表的问题，所以下面的几个章节可以略读。

2.4.3 改变容量和规模 *

您可能会想到，除了扩容之外，我们还可以进行缩容（shrink），降低 m 的值从而避免装填因子过小，造成内存浪费。但是，现实中很少进行缩容。因为扩容和缩容都需要时间，在扩容的场合是实现可变长特性的必需，但在缩容的场合仅仅是节约了空间而已。有多个原因让我们不愿意缩容：

1. 我们可以接受一定程度的空间浪费，因为很少有程序能占满全部的内存。
2. 如果缩容之后，又因为规模扩大而不得不扩容，一来一回浪费了不少时间，而价值甚微。
3. 当不得不考虑空间时，我们有很多其他方法可以节约出这些空间，不一定要使用缩容的技术。比如复制初始化生成一个新向量，然后清空原向量来释放内存；按照之前介绍的方法，这个新向量的装填因子为 1，处在空间最大利用的状态。

因此，在本书中将不再讨论缩容；您可以在自己的向量类中实现这个特性，并观察它的效果。

无论是扩容还是缩容，我们都需要重新申请一片内存。在扩容的场合，这很好理解。我们预定了 1000 到 1099 的房间，但在我们预定之后，1100 号房间可能被其他旅客占用了。这时如果我们想要预定连续的 200 个房间，就需要重新找一段空房间。缩容的场合，则是为了安全性考虑，不允许释放数组的部分内存。重新申请内存之后，我们将数据复制到新内存中。下面是扩容的一个实现。

```

1 void reserve(size_t n) override {
2     if (n > m_capacity) {
3         auto tmp { std::make_unique<T[]>(n) };
4         std::move(m_data.get(), m_data.get() + m_size, tmp.get());
5         m_data = std::move(tmp);
6         m_capacity = n;
7     }
8 }
```

图2.4展示了当插入元素时如果发现容量不足，所进行的扩容过程。其中释放原先的内存这一步，本书中所采用的智能指针会自动完成，而 C 语言和旧标准 C++ 则需要显式地 `delete[]` 释放内存。

可以看出，扩容是一项成本很高的操作，因为它需要开辟一块新的内存。设扩容之后的容量为 m ，则扩容算法的时间复杂度为 $\Theta(m)$ 。由于 m 可能会很大，我们

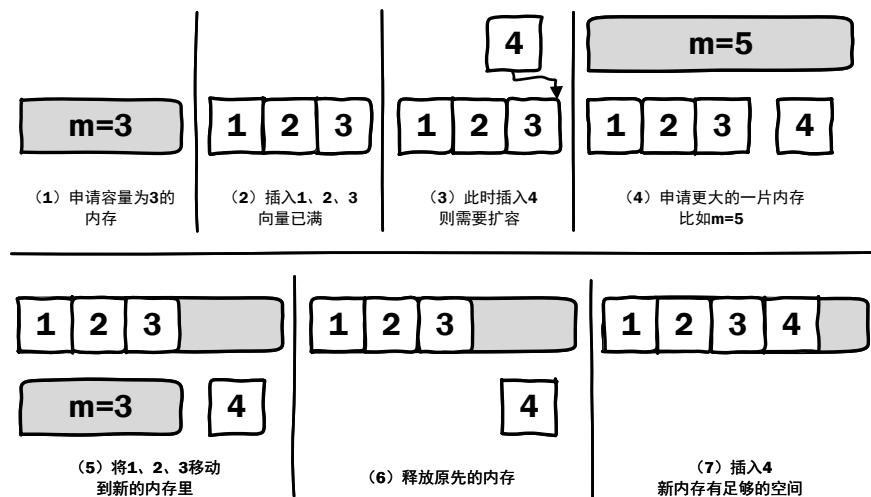


图 2.4 扩容过程

不希望经常扩容。在下一节里将会讨论一些扩容策略。

我们设计了一个方法 `resize` 来用改变规模，当规模超过容量（装填因子超过 1）的时候调用 `reserve` 扩容。需要注意的是，一些其他的方法也会改变规模，比如插入方法 `insert` 会让规模增加 1，而删除方法 `remove` 会让规模减 1。我们需要在实现插入方法的时候也考虑扩容问题；如果您实现了缩容，那么在实现删除方法的时候也需要考虑缩容问题。

```

1 void resize(size_t n) {
2     if (n > m_capacity) {
3         reserve(n);
4     }
5     m_size = n;
6 }
```

2.4.4 扩容策略 *

当我们调用 `resize` 的时候，可以立刻知道，需要扩大到多少容量才能容纳目标的规模。但实际情况下，很多时候元素是被一个一个加入到向量中的，这个时候，按照 `resize` 的策略，每次都扩容到新的规模，是一个糟糕的选择。假设初始化为了一个规模为 n 的向量，然后元素一个一个被加入，那么按照 $n \rightarrow n+1 \rightarrow n+2 \rightarrow \dots$ 的次序扩容，每加入一个元素，都会造成至少 n 个元素的复制，时间效率极差。

因此，在面对持续插入的时候，我们需要设计新的扩容策略，以降低扩容发生的频率。这个策略应该是由向量的设计者提供的，而不是用户：如果用户知道更加合适的策略，他们会主动使用 `reserve` 进行扩容。但是，用户通常没有精力用在这种细节上；这个时候，向量的设计者提供的扩容策略就会成为一个不错的备选项。

现在我们尝试为扩容策略的问题添加一个抽象的描述。当我们讨论扩容的时候，显然不需要知道向量中的数据内容是什么。因此，扩容策略作为一个算法，输入向量的当前规模 n 和当前容量 m ，输出新的容量 m' 。这个描述具有良好的可重用性，它同样可以用于缩容。

```

1 class AbstractVectorAllocator : public Algorithm<size_t(size_t,
2     size_t)> {
3     protected:
4         virtual size_t expand(size_t capacity, size_t size) const =
5             0;
6         virtual size_t shrink(size_t capacity, size_t size) const =
7             0;
8     public:
9         size_t operator()(size_t capacity, size_t size) override {
10             if (capacity <= size) {
11                 return expand(capacity, size);
12             } else {
13                 return shrink(capacity, size);
14             }
15         }
16     };
17 }
```

基于上述的分析，我们可以用这个类表示扩容策略。在后面的章节，将继承这个类并重写 `expand` 方法，实现不同策略的扩容，而缩容方法则直接返回 `capacity`（永不缩容）。当您需要实现缩容的时候，也可以使用这个模板，重写 `shrink` 方法。如果您对如何扩容有一些自己的想法，笔者建议您先实现自己的扩容策略，然后再阅读后面的理论部分。

2.4.5 等差扩容和等比扩容 *

那么，应该如何设计扩容策略呢？一个简单的想法是，既然每次容量 +1 不行，那就加多一点。这种思路可以被概括为等差数列扩容方法。如果选取 d 作为公差，那么在本节开始的那个例子中，将按照 $n \rightarrow n + d \rightarrow n + 2d \rightarrow \dots$ 的次序扩容。

```

1 template <size_t D> requires (D > 0)
2 class VectorAllocatorAP : public AbstractVectorAllocator {
3     protected:
4         size_t expand(size_t capacity, size_t size) const override
5             {
6                 return capacity + D;
7             }
8     };
8 }
```

这里使用了 C++20 引入的 `requires` 语法，要求用户给定的模板参数 $D > 0$ 。因为很明显，如果 $D \leq 0$ ，`expand` 将永远扩不起来。

既然有了等差数列，另一个很容易想到的方法是按照等比数列扩容。如果选

取 q 作为公比，则会按照 $n \rightarrow qn \rightarrow q^2n \rightarrow \dots$ 的次序扩容。

```

1 template <typename Q> requires (Q::num > Q::den)
2 class VectorAllocatorGP : public AbstractVectorAllocator {
3     protected:
4         size_t expand(size_t capacity, size_t size) const override
5             {
6                 size_t newCapacity { capacity * Q::num / Q::den };
7                 return std::max(newCapacity, capacity + 1);
8             }
9 }
```

这里允许了 C++11 提供的编译期有理数 `std::ratio` 作为模板参数，`Q::num` 表示分子，而 `Q::den` 表示分母。请注意，需要保证新的容量比原有容量大，否则扩容就没有意义。上面的做法保证了容量至少会扩大 1。比如，当 $Q = \frac{3}{2}$ ，往容量为 0 的向量里连续插入元素时，容量变化为 $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 9 \rightarrow \dots$ ，如果没有容量至少扩大 1 的设计，等比扩容将永远停留在 0 容量。

2.4.6 分摊复杂度分析 *

很显然，进行单次扩容操作的时候，等差扩容的时间复杂度为 $O(n+d) = O(n)$ ，等比扩容的时间复杂度为 $O(qn) = O(n)$ （因为 q 是常数），两者看起来没有区别；甚至和我们已经知道效率很低的情况（ $d=1$ 的等差扩容）也没有区别。这也意味着，我们评价时间效率的方法可能出现了一些问题。

问题的关键在于，我们设计扩容策略的目的是按照等差或等比的数列扩容，而不是一次扩容。所以，评价这两种扩容规则的标准，不是进行一次扩容的效率或进行一次扩容后的装填因子，而是比较一系列扩容操作的总体效率和在这一系列扩容操作中的平均装填因子。用已有的复杂度分析工具不足以对这两种策略的效率进行准确评价。为了对一系列操作进行分析，需要引入新的复杂度分析标准。

一般地，假设 O_1, O_2, \dots, O_n 是连续进行的 n 次操作，则当 $n \rightarrow \infty$ ，这 n 次连续操作所用时间的平均值的复杂度，称为这一操作的**分摊复杂度**，对分摊复杂度的分析称为**分摊分析**。分摊分析的原则之一是：使用相同效果的操作序列。所以，要比较上述两种算法，不应该把每次操作取为“进行一次扩容”（因为两种方法扩容量不一样），而应该取为“向量 V 的规模增加 1”。连续进行 n 次操作，就可以考虑向量 V 的规模从 0 增长为 n 的过程。

在等差扩容方法中，容量依次被扩充为 $d, 2d, 3d, \dots, n$ ，共进行 $\frac{n}{d}$ 次扩容。因此，分摊复杂度为：

$$T(n) = \frac{d + 2d + 3d + \dots + n}{n} = \frac{\left(\frac{n}{d}\right) \cdot d + \frac{\left(\frac{n}{d}\right)\left(\frac{n}{d}-1\right)}{2} \cdot d}{n} = \frac{\frac{n}{d} + 1}{2} = \Theta\left(\frac{n}{d}\right)$$

另一方面，进行 k 次扩容之后的装填因子至少为 $\frac{kd}{(k+1)d} = \frac{k}{k+1}$ ，当 $k \rightarrow \infty$ 时，装填因子趋于 100%。

在等比扩容方法中，容量被依次扩充为 q, q^2, q^3, \dots, n ，共进行 $\log_q n$ 次扩容。因此，分摊复杂度为：

$$T(n) = \frac{q + q^2 + q^3 + \dots + n}{n} = \frac{q \cdot \frac{1-n}{1-q}}{n} = \Theta\left(\frac{q}{q-1}\right) = O(1)$$

另一方面，装填因子不断在 $\left[\frac{1}{q}, 1\right]$ 之间线性增长，平均装填因子为 $\frac{1+q}{2q}$ 。可以看出，不管怎样选择 q ，对分摊复杂度都没有影响，而更小的 q 能够带来更大的平均装填因子。当选择 $q = 2$ 时，平均装填因子为 75%。

可以看出，等比扩容的装填因子并没有很低，而换来了分摊时间复杂度上巨大的优化。因此，我们倾向于选择等比扩容。至于等比扩容的公比，则是一个值得讨论的话题。从上面的推导中，我们发现分摊时间复杂度的系数为 $\frac{q}{q-1}$ ，它会随 q 的增加而降低；另一方面，平均装填因子也会随 q 的增加而降低。因此，选择更大的 q ，事实上是以时间换空间的做法。

因为分摊 $O(1)$ 已经很快，所以通常选取的 q 比较小。常见的公比选择有 2 和 $\frac{3}{2}$ 。邓书上介绍的版本选择了 2，这也是 GCC 和 Clang 的选择；而 MSVC 则采用更节约空间的 $\frac{3}{2}$ 。

需要指出的是，等比扩容也存在一些劣势：容量越大，装填因子不高带来的空间浪费愈发明显，所以有些对空间要求较高的情况下，也采用二者相结合的方式：在容量比较小时等比扩容、在容量比较大的时候等差扩容。这种思想在《网络原理》里的慢启动中得到了应用。

最后，既然有等比扩容，必然也有等比缩容。当我们讨论缩容的时候，通常需要结合一个**缩容阈值**，当装填因子低于这个阈值时才引发缩容，而不是每当规模低于容量时就缩容，否则就丧失了向量的灵活性。尽管本书不实现缩容（相当于缩容阈值为 0），但当缩容和缩容阈值被纳入讨论，可以命制一些有趣的问题。设等比扩容的公比为 $q_1 > 1$ ，等比缩容的公比为 $q_2 < 1$ ，缩容阈值为 θ ，您可以进行思考和计算，这三个变量满足什么条件时，才能保证对于任意的、由插入和删除组成的操作序列，扩容和缩容总和的分摊时间复杂度为 $O(1)$ 。

2.5 插入、查找和删除

对于任何数据结构，都有三种基本的操作：

1. **插入** (insert)：向数据结构中插入一个元素。
2. **查找** (find)：查找一个元素在数据结构中的位置。

3. **删除** (delete)：从数据结构中移除一个元素。

在这一节中，我们以向量为例介绍这三种基本的操作。

2.5.1 插入一个元素

要将待插入的元素 e 插入到 $V[r]$ ，那么可以将原来的向量 $V[0:n]$ 分成 $V[0:r]$ 和 $V[r:n]$ 两部分。

- 插入之前，向量是 $V[0:r]$, $V[r:n]$ 。
- 插入之后，向量是 $V[0:r]$, e , $V[r:n]$ 。

可以发现，在插入的前后，前一段 $V[0:r]$ 的位置是不变的，而后一段 $V[r:n]$ 需要整体向后移动 1 个单元的位置。据此，可以设计下面的算法，算法的原理如图2.5所示。

```

1 Rank insert(Rank r, const T& e) override {
2     std::move_backward(m_data.get() + r, m_data.get() + m_size,
3                         m_data.get() + m_size + 1);
4     m_data[r] = e;
5     ++m_size;
6 }
```

这里需要使用 `std::move_backward` 而不能用 `std::move`，因为向后移动 1 个单元这个过程，需要先移动最后一个元素、再移动倒数第二个元素、以此类推；如果从第一个元素开始移动，就会覆盖掉还没移动的元素。如果您对 STL 算法不熟练，也可以改写为熟悉的循环形式。

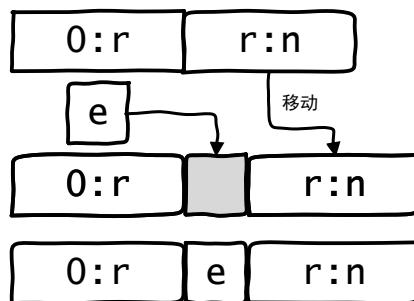


图 2.5 往向量中插入一个元素

在上面的算法中忽略了一点：插入有可能造成装填因子大于 1 的问题。如果准备插入的时候发现向量已满（装填因子等于 1），那么应当先使用上一节讨论的扩容策略，扩大向量的容量，然后再执行插入。您会发现，扩容的过程中包括了一次移动，`std::move_backward` 又包括一次移动，这两次移动中会有一些重复的、不必要的赋值操作。不过，因为扩容被调用的次数很少，我们可以忽略这些额外的性能损失。

为了让我们在上一节设计的算法被嵌入到向量里来，我们可以为向量模板增加一个参数，写成下面这样的形式。

```
1 template <typename T, typename Alloc = VectorAllocatorGP<std::ratio<3, 2>>>
2 requires std::is_base_of_v<AbstractVectorAllocator, Alloc>
```

在上面这个模板参数表声明中，我们规定模板的第二个参数`Alloc`必须是我们上面实现的`AbstractVectorAllocator`的派生类。用户可以不显式地指定扩容策略，而选择我们默认的策略（公比为 $\frac{3}{2}$ 的等比扩容）。加入这个参数之后，我们只需要在`insert`的开头进行一次判断，就可以自动地在插入满向量时进行扩容。

最后，我们上面实现的插入算法，传入的是`const`引用类型，这意味着元素并没有被真正的“插入”，被实际插入的是一个副本。如果我们希望让元素真正被插入，那么应当使用移动语义。移动语义版本的插入算法和复制语义基本相同。如前所述，以后将不再展示移动语义的版本。

```
1 Rank insert(Rank r, T&& e) override {
2     if (m_size == m_capacity) {
3         reserve(Alloc {}(m_capacity, m_size));
4     }
5     std::move_backward(m_data.get() + r, m_data.get() + m_size,
6                         m_data.get() + m_size + 1);
7     m_data[r] = std::move(e);
8     ++m_size;
9     return r;
}
```

2.5.2 平均复杂度分析

为了更定量地分析插入操作的时间效率，引入一个新的复杂度分析策略：**平均复杂度**。

在介绍复杂度时曾经强调，复杂度是依赖于数据规模，不依赖于输入情况的分析手段。在上面的插入算法中，数据规模通常认为是 n ，而 r 是具体情况带来的参数。为了研究不同具体情况对算法时间效率的影响，有三种常见的分析手段：

- 最坏时间复杂度**：研究在情况最坏的情况下复杂的度。很多算法有硬性的时
间限制（如在复试的机试中，通常要求输出结果的时间不能多于 1s 或 2s），
此时常常使用最坏时间复杂度分析。这是最常用的时间复杂度分析。
- 最好时间复杂度**：研究在情况最好的情况下的复杂度。研究最好时间复杂度
的意义远小于最坏时间复杂度。最好时间复杂度有时用于嘲讽某种算法的效
率：在最好的情况下，这种算法的复杂度也只能达到（某个复杂度），而我的
新算法在最坏的情况下也可以达到（某个更好的复杂度）。

3. **平均时间复杂度**: 研究在平均情况下的复杂度。如果没有硬性的时间限制，则平均时间复杂度往往能更好地反映一个算法的总体时间效率。平均时间复杂度需要知道每种情况发生的先验概率，在这个概率的基础上计算 $T(n)$ 的数学期望的复杂度。在针对现实数据的实验研究中，常见的假设包括正态分布、帕累托（Pareto）分布和泊松（Poisson）分布；而在《数据结构》学科中，通常假设成等可能的分布，以方便进行理论计算。

平均复杂度很容易和分摊复杂度发生混淆，需要加以区分。下面是它们的一些典型的差异：

1. 分摊复杂度是一系列连续操作的平均效率，而平均复杂度是单次操作的期望效率。
2. 分摊复杂度的一系列连续操作是有可能（通常都）存在后效的，而平均复杂度只讨论单次操作的可能情况。
3. 分摊复杂度需要指定每次进行何种的基本操作，而平均复杂度需要指定各种情况的先验概率。

最坏、最好、平均时间复杂度对应统计里的最大值、最小值和数学期望。显然，其他统计量，比如方差，在分析的时候也是有价值的，也深得科研人员重视。但在《数据结构》的考试中，是不会涉及到这些统计量的分析的，只需要知道最坏、最好和平均时间复杂度的分析技术即可。

现在回到插入的算法，它的时间复杂度是 $\Theta(n - r)$ 。显然，最好时间复杂度是 $O(1)$ （插入在末尾的情况），最坏时间复杂度是 $\Theta(n)$ （插入在开头的情况）。这里有略微不严谨的地方，因为 r 的最大值可以取到 n ，此时 $n - r = 0$ ，不再符合复杂度记号的定义；不过，因为我们清楚任何算法的时间都不可能为 0，所以一般不在这个细节上做区分。

为了求平均时间复杂度，一个合理的假设是， r 的取值对于 $[0:n + 1]$ 之间的整数是等概率的（注意有 n 个可以插入的位置，而不是 $n - 1$ 个）。在这个假设下，容易算出单次插入的平均时间复杂度为 $\Theta(n)$ 。

2.5.3 查找一个元素

查找需要返回找到的位置。对向量而言，只需要得到被查找元素的秩就可以了。和插入、删除相比，查找具有更加丰富的灵活性，甚至于一些编程语言（如 SQL）的核心就是查找。

最简单的查找是按值查找。即，给定被查找元素的值，在数据结构中找到等于这个值的元素。对于更加复杂的查找类型，比如按区间查找等，人们设计了更加复杂的数据结构来应对。对于按值查找的问题，最简单的方案就是检测向量中的每

个元素是否等于要查找的元素 e ，如果等于，就将它的秩返回。

```

1 Rank find(const T& e) const override {
2     for (size_t i { 0 }; i < m_size; ++i) {
3         if (m_data[i] == e) {
4             return i;
5         }
6     }
7     return m_size;
8 }
```

关于找不到的情况，有多种处理方式。上面采用的方法是返回无效的秩，除了返回序列尾部溢出的 `m_size` 之外，返回序列头部溢出的 `-1` 也是常见的选择。此外，也可以将返回值的类型改为 `std::optional`，当找不到的时候返回一个无效值。

设 e 在向量中的秩为 r ，那么在查找成功的情况下，上述算法的时间复杂度为 $\Theta(r)$ 。在查找失败的情况下，算法的时间复杂度为 $\Theta(n)$ 。这里可以分析，在等可能条件下，查找成功时的平均时间复杂度是 $\Theta(n)$ 。注意，查找成功的概率是一个很难假设的值，所以在分析平均时间复杂度时，通常只分析“查找成功时”和“查找失败时”的平均时间复杂度，而不会将它们混为一谈。

因为对于向量 V 和待查找元素 e 的情况没有更多的先验信息，所以暂时也没有比上面更高效的解决方案。**利用信息思考**是计算机领域重要的思维方式。在设计算法时，应尽可能利用更多的先验信息。反之，如果先验信息不足，则算法的效率受到信息论限制，不可能会特别高。这个思维方式在后文介绍各种算法的设计过程时，还会反复出现。

不过，这个算法还是有一些值得推敲的地方：如果 e 在向量 V 中出现了多次，那么这个算法只会返回最小的秩。您可以思考一下，如何将其修改成返回最大的秩？并分析修改后的算法时间复杂度变化。

2.5.4 删除一个元素

删除元素是插入元素的逆操作。在插入元素时，让被插入元素的后继后移；因此在删除元素的时候，只需要让被删除元素的后继前移即可。需要注意前移和后移在方向上的差别，插入时的 `std::move_backward`，逆操作应该是 `std::move`。

```

1 T remove(Rank r) override {
2     T e { std::move(m_data[r]) };
3     std::move(m_data.get() + r + 1, m_data.get() + m_size,
4               m_data.get() + r);
5     --m_size;
6     return e;
7 }
```

和插入一样，可以分析出删除操作时间复杂度 $\Theta(n - r)$ ，平均时间复杂度 $\Theta(n)$ ，空间复杂度 $O(1)$ 。到此为止，我们实现了完整的 `Vector` 类，可以开始实验了。如果您自己完成了向量的设计，可以使用 `VectorTest.cpp` 进行简单的功能测试，并在实验中将 `dslab::Vector` 替换为自己的向量类。

2.5.5 实验：插入连续元素

在这个实验中，我们将用实验观察，等差扩容和等比扩容的时间效率差异。因为我们评估的是分摊时间，所以需要构造一个插入连续元素的场景来进行观察。从2.5.1节中我们知道，在位置 r 插入一个元素的时间复杂度为 $\Theta(n - r)$ 。为了降低插入连续元素这个操作本身对，更好地观察扩容时间，我们固定每次都在向量的末尾插入元素。代码可以在 `VectorInsert.cpp` 中找到。

我们构造下面的类，作为插入连续元素问题的基类。因为在测试过程中，每次连续插入结束之后，需要将向量重置为空（避免已分配的空间影响），所以这里定义了一个 `reset` 方法。

```
1 class VectorInsertProblem : public Algorithm<size_t(size_t)> {
2 public:
3     virtual void reset() = 0;
4 };
```

返回值定义为连续插入结束后的向量容量，因为笔者希望观察连续插入结束之后扩容到了多大。您也可以将其改为 `void` 或者其他您希望观察的变量。接下来，我们让在2.4.5节中定义的等差扩容和等比扩容策略，作为 `Vector` 类的参数传入。

```
1 template <typename Vec>
2     requires is_base_of_v<AbstractVector<size_t>, Vec>
3 class VectorInsertBasic : public VectorInsertProblem {
4     Vec V;
5 public:
6     size_t operator()(size_t n) override {
7         for (size_t i { 0 }; i < n; ++i) {
8             V.insert(V.size(), i);
9         }
10    return V.capacity();
11 }
12    void reset() override {
13        V.clear();
14    }
15 };
```

上面的模板接受一个 `Vec` 作为向量类型名，这是为了兼容您自己写的向量类。它用到了零初始化、赋值、插入、获取容量和规模的方法，如果您继承了 `AbstractVector`，这些方法都应当已经实现。您也可以创建自己的测试类参与

对比测试。对于本书的示例`Vector`实现，使用下面的模板。

```

1 template <typename Alloc>
2     requires is_base_of_v<AbstractVectorAllocator, Alloc>
3 class VectorInsert : public VectorInsertBasic<Vector<int, Alloc>
4     >> {
5 public:
6     string type_name() const override {
7         return Alloc {}.type_name();
8     }
9 };

```

这里重载了`type_name`，否则由于模板复杂，输出的类型名会非常长。在实验的示例代码中，我们比较了 $D = 64$ 、 $D = 4096$ 、 $Q = \frac{3}{2}$ 、 $Q = 2$ 、 $Q = 4$ 这些情况。您可以发现，当 n 比较小的时候，差异不明显；而当 n 比较大，如达到 10^6 的量级时， $D = 64$ 会极其缓慢，而 $D = 4096$ 也慢慢和三种等比扩容拉开距离（如果您增加一个 $n = 10^7$ 的用例，会更加明显）。相反，三种等比扩容的区别非常微小，我们可以判断出，这已经非常接近插入这些元素本身需要的时间，和扩容的关系不大。

表 2.1 不同扩容方式下连续插入元素的耗时对比（单位：ms）

n	等差 $d = 64$	等差 $d = 4,096$	等比 $q = \frac{3}{2}$	等比 $q = 2$	等比 $q = 4$
	515	20	12	13	11
200,000	2318	57	22	21	21
600,000	3662	89	33	34	33
800,000	5164	121	43	42	42
1,000,000	6649	155	50	54	51

如何证明上面的判断呢？去计算连续插入中原子操作的数量（计算时间复杂度的常数）是困难、繁琐且容易出错的工作。您可以加入一个 $Q = 10^6$ 的向量，它的容量会直接从 1 跳变到 10^6 ，也就是，在 $n = 10^6$ 的情况下，它没有进行任何多余的扩容。您会发现，即使是这样，也没有和上面三种等比扩容有显著差异，从而能够验证上面的判断。这种实验方法在计算机学科的学习中是一个有力武器。这个实验请您自己完成，您也可以加入一些自己实现的其他扩容策略来观察效果。比较有挑战性的工作是，在实现缩容之后，类似本实验，设计一个新的实验评估缩容的效果（请注意，缩容本身是时间换空间的做法，仅仅测试缩容的时间性能是不合适的）。

需要指出的是，这个实验结果只是粗略性的，因为连续插入而不做其他事情是一个非常特殊的用例。当这种情况真正发生的时候，应当直接`reserve`足够大的

空间，而不是让向量按照设计者提供的默认扩容方案慢慢扩容。

2.5.6 实验：向量合并

如果要插入的不是单个元素，而是多个元素，情况会有什么变化呢？现在，假设我们有两个向量 $V[0:n]$ 和 $V_1[0:n_1]$ ，我们希望将整个 V_1 插入到 V 的位置 r 处，实现向量合并。代码可以在 *VectorConcat.cpp* 中找到。这个问题不难，建议您自己实现算法参与对比。

```

1 class VectorConcat : public Algorithm<Vector<int>& (Rank) > {
2 protected:
3     Vector<int> V {}, V1 {};
4 public:
5     void initialize(int n, int n1) {
6         V.reserve(static_cast<size_t>(n) + n1);
7         V1.reserve(n1);
8         V.resize(n);
9         V1.resize(n1);
10    }
11 }
```

和上一节类似，我们定义了一个辅助函数用来对问题进行初始化。我们不希望扩容影响实验结果，所以预先给 V 分配了 $n + n_1$ 的空间。

如果我们采用2.5.1节中介绍的方法，将 V_1 中的元素一个一个插入到该位置，那么时间复杂度会高达 $\Theta((n - r) \cdot n_1)$ ，平均 $\Theta(n \cdot n_1)$ ，略显笨重。

```

1 // VectorConcatBasic
2 Vector<int>& operator()(Rank r) override {
3     for (int i : V1) {
4         V.insert(r, i);
5         ++r;
6     }
7     return V;
8 }
```

您可以敏锐地发现，只要再次使用在讨论单元素插入时的分析方法，就可以得到更加高效的算法。要将待插入的向量 V_1 插入到 $V[r]$ ，那么可以将原来的向量 $V[0:n]$ 分成 $V[0:r]$ 和 $V[r:n]$ 两部分。

1. 插入之前，向量是 $V[0:r]$, $V[r:n]$ 。
2. 插入之后，向量是 $V[0:r]$, V_1 , $V[r:n]$ 。其中， $V[r:n]$ 被转移到了 $V[r+n_1:n+n_1]$ 的位置上。

这样设计出了一个批量插入的算法，两种方法的对比如图2.6所示。

```

1 // VectorConcatFast
2 Vector<int>& operator()(Rank r) override {
3     V.resize(V.size() + V1.size());
```

```

4     move_backward(begin(V) + r, end(V) - V1.size(), end(V));
5     move(begin(V1), end(V1), begin(V) + r);
6     return V;
7 }

```

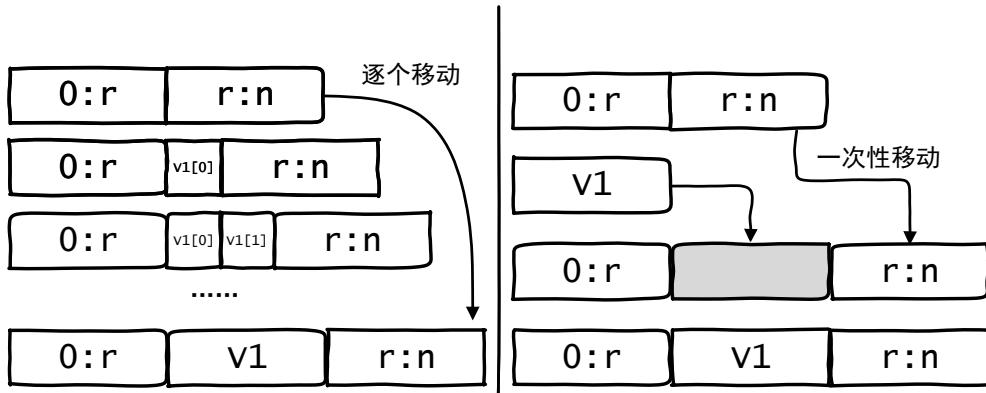


图 2.6 向量合并的两种方法

您可以自行分析上述批量插入算法的时间复杂度和平均时间复杂度。因为已经预先 `reserve`, 这里 `resize` 可以保证在 $O(1)$ 时间里完成。示例程序中进行了四类情况的评估：插入少量元素到开头、插入大量元素到开头、插入少量元素到结尾、插入大量元素到结尾。这里的少量可以认为是常数 $n_1 = O(1)$, 而大量可以认为 $n_1 = \Theta(n)$ 。您可以对比前面得到的时间复杂度，验证您对这两个算法的评估结论。您也可以将上述两个示例程序和您自己的实现进行对比。

表 2.2 向量合并算法的耗时对比（单位：ms）

①	逐个插入	批量插入
插入少量元素到开头	1 ^②	4
插入大量元素到开头	1060	7
插入少量元素到结尾	0	0
插入大量元素到结尾	5	2

① $n = 2 \times 10^5$, 少量元素 = 10^2 , 大量元素 = 10^5 。

② 在这个场景下逐个插入反而性能更高, 看起来是反直觉的, 它来自于编译器优化。

批量插入的算法中体现出的用块操作代替多次单元操作的思想，在以线性表为背景的算法设计题中应用广泛。在我们的实现中，插入操作和扩容操作是解耦的：容量已经被预先分配好。在实际情况下，我们也要同时考虑扩容的成本。按

照解耦的实现，在扩容申请了新的数组空间之后，会将原数组的元素复制过去，然后再执行插入，对这些元素进行移动。很显然， $V[r]$ 之后的元素被移动了两次。事实上，如果进行耦合的实现，可以让这些元素只被移动一次：在扩容申请了新的数组空间之后， $V[r]$ 之后的元素可以直接移动到它们的目的位置，为 V_1 里的元素“空出一块空间”。您可以自己实现这个耦合算法。因为需要在外部访问`Vector`里的成员，因此您应当将您的类或函数声明为友元。

2.5.7 实验：按值删除元素

在2.5.4节，我们讨论的删除是按位置删除，具体到向量就是按秩删除。另一种删除的方式是按值删除，也就是说，我们给定一个元素 e ，想要删除向量中和 e 相等的每一个元素。代码可以在`VectorRemove.cpp`中找到。同样，建议您自己实现一个算法。

在分析这个问题之前，先对按值操作进行一些深层次的理解。我们知道，数据结构是元素的集合，元素之间是互不相同的，但这并不妨碍它们相等，因为我们可以定义“相等”（反映到C++中，就是重载运算符`operator==`）。比如说，我们定义值相同为“相等”，这样就不需要考虑元素的地址；这是按值删除的思路。顺着这个思路，我们可以扩展按值删除到更加一般的按条件删除。比如说，向量里的元素是一个结构体，我们定义结构体的某个属性相同为“相等”，而其他属性可以不被考虑；此时，某个属性等于给定的值就是我们定义的条件。在C++的STL中，按值删除对应的算法是`std::remove`，而按条件删除对应的算法是`std::remove_if`，二者具有高度的相似性。其他的一些算法也有相应的“按条件”版本，您可以根据需要自行了解，这里不再赘述。

下面我们来讨论按值删除元素的问题。为了评估算法的性能，示例程序构造了一个场景：在一个规模为 n 的向量中，第奇数个元素为1，第偶数个元素为0，我们的算法将要删除所有的0，也就是说删除一半的元素。

```

1 class VectorRemove : public Algorithm<size_t, int> {
2 protected:
3     Vector<int> V {};
4     virtual void batchRemove(int e) = 0;
5 public:
6     size_t operator()(int e) override {
7         size_t n = V.size();
8         batchRemove(e);
9         return n - V.size();
10    }
11    void initialize(size_t n) {
12        V.resize(n);
13        for (size_t i { 0 }; i < n; ++i) {
14            V[i] = i % 2;

```

```

15     }
16 }
17 };

```

和上一节一样，我们从最朴素的想法开始：逐个查找、逐个删除。我们在向量 V 中查找要删除的 e ，每发现一个就删除一个，直到没有等于 e 的元素为止。如图2.7所示。

```

1 // VectorRemoveBasic
2 void batchRemove(int e) override {
3     Rank r { 0 };
4     while (r = find(begin(V), end(V), e) - begin(V), r < V.size
5           ()) {
6         V.remove(r);
7     }

```

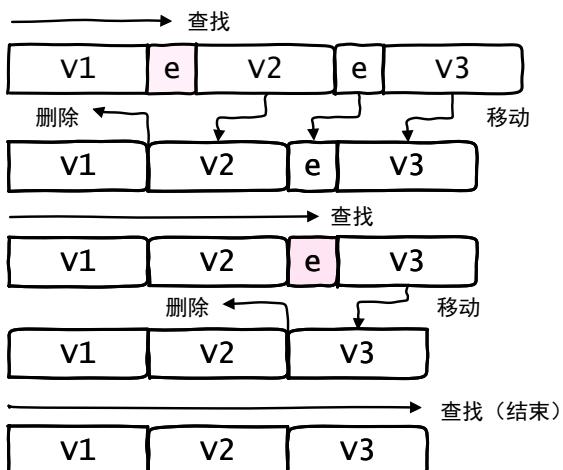


图 2.7 向量按值删除元素：逐个查找、逐个删除

上述朴素算法的空间复杂度是 $O(1)$ ，但是时间效率是很低的。这里使用的 `std::find`，原理和2.5.3节介绍的查找一样，时间复杂度为 $\Theta(r)$ 。`std::find`返回的是一个迭代器，和起始迭代器`std::begin`相减之后就可以得到找到的位置（秩）。而另一方面，在找到之后，`remove`的时间复杂度为 $\Theta(n - r)$ 。所以，每次循环的时间复杂度为 $\Theta(n)$ 。在极端情况下（比如我们设计的实验场景），有 $\Theta(n)$ 个元素要被删除，此时时间复杂度为 $\Theta(n^2)$ 。考虑到 C++ 的标准算法库在考试中手写代码时通常不能使用（阅卷者可能看不懂），所以如果您不熟练，非常建议您在理解这些标准算法之后把它改写成 C 语言的风格，直到您能熟练地在 STL 和 C 语言风格之间做转换为止。学习的时候使用 STL 的好处是精简代码，减少记忆量，容易抓住主要矛盾。

在设计算法的时候，题目不一定会给出要求的复杂度。这个时候，可以对比一

下相似问题的复杂度。比如，之前介绍讨论批量插入（向量合并）的时候可以做到线性的时间复杂度，没有理由批量删除（按值删除）需要平方级的时间复杂度。因此，我们需要寻找提高时间效率的切入口。

为了降低时间复杂度，就要设法降低在算法中进行的不必要工作。在不必要工作中，有几种比较典型。一种是不到位工作，它代表了在算法中，本能够一步到位的计算被拆分成了若干个碎片化的步骤，而产生的时间浪费。比如在向量合并的逐个插入算法中，后缀 $V[r + 1:n]$ 就接连移动了 n_1 次；我们通过合并这些移动实现了算法优化。

另一种是重复工作，它代表了在算法中，重复计算了同一算式造成的时间效率浪费。在上面的朴素的按值删除算法中，就有一项非常明显的重复工作。如图2.7所示，第一次检索了第一段 V_1 ，第二次检索了 $V_1 + V_2$ ，第三次检索了 $V_1 + V_2 + V_3$ ；这里 V_1 被检索了 3 次， V_2 被检索了 2 次，而事实上只需要检索一次即可。在我们查找完毕的时候，可以记录下当前查找到的位置；下一次查找的时候从记录下的位置开始记录，如图2.8所示。借用 C++ 的 STL，我们对朴素算法做很小的改动就可以做到这一点，您可以对比两张图和两份代码。

```

1 // VectorRemoveImproved
2 void batchRemove(int e) override {
3     Rank r { 0 };
4     while (r = find(begin(V) + r, end(V), e) - begin(V), r < V.
5         size()) {
6         V.remove(r);
7     }
}

```

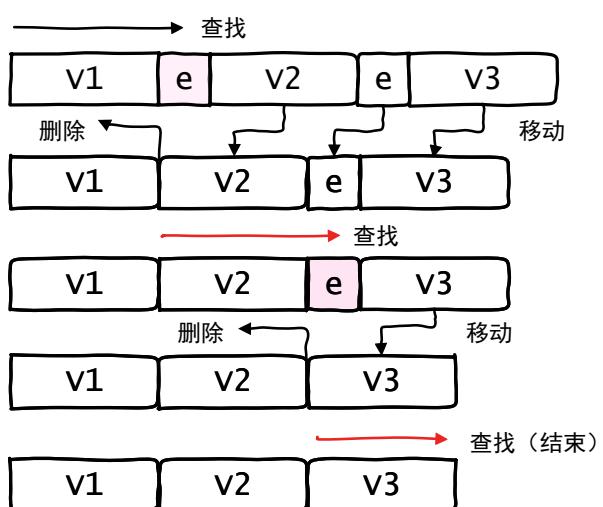


图 2.8 向量按值删除元素：一次查找、逐个删除

然而您会发现，在最坏的情况下（所有元素都要被删除），光是 `remove` 就要

花费 $\Theta(n^2)$ 的时间，上面这个算法的优化程度仍然不够。因此，下一步优化就要从 `remove` 入手，需要将 `remove` 的工作展开来，看看其中哪些是不必要的。

在 `remove` 中，主要消耗时间的是元素移动的操作。您可以发现，如果 $V[0:i]$ 中有 k 个元素要删除，那么最后一个元素 $V[i - 1]$ 就要向前移动 k 次：依次移动到 $V[i - 2] \rightarrow V[i - 3] \rightarrow \dots \rightarrow V[i - k - 1]$ 的位置上。比如，在图2.8中， V_3 就移动了 2 次。您可以敏锐地发现者正是前面所讲述的不到位工作。这一系列的移动被拆成了 k 次，而实际上是可以一步到位，直接从 $V[i - 1]$ 移动到目标位置 $V[i - k - 1]$ 的。

为什么可以直接移动到目标位置呢？注意到，无论是上面的 Basic 还是 Improved 算法，当检索到 $V[i]$ 的时候，前缀 $V[0:i]$ 的所有元素都已经被检索过了，因此 k 的值已经确定了，并且前 $i - 1$ 个元素已经移动到了正确的目标位置。所以您可以用归纳法的思路，证明直接移动的正确性。证明完成之后，剩下的就只有编码的工作了。强烈建议您自己完成这个算法再阅读示例程序。下面展示了一种实现。

```

1 // VectorRemoveFSP
2 void batchRemove(int e) override {
3     Rank k { 0 };
4     for (Rank r { 0 }; r < V.size(); ++r) {
5         if (V[r] == e) {
6             ++k;
7         } else {
8             V[r - k] = move(V[r]);
9         }
10    }
11    V.resize(V.size() - k);
12 }
```

非常显然，现在时间复杂度被缩减到 $\Theta(n)$ 了。上面这个算法的思路可以被概括为快慢指针，这是线性表算法设计中非常典型的技巧。快指针即探测指针，指向 $V[r]$ ；慢指针即更新指针，指向 $V[r - k]$ 。快指针找到需要保留的元素，然后将它们移动到慢指针的位置处。如图2.9所示。

表 2.3 按值删除算法的耗时对比（单位：ms）

n	Basic	Improved	快慢指针
10^3	2	0	0
10^4	213	19	0
10^5	20192	1972	1

您可以通过示例程序的测试明显感知到这三种算法的性能差异。作为结束，上述算法在 C++ 的 STL 中有一个简单的记法：

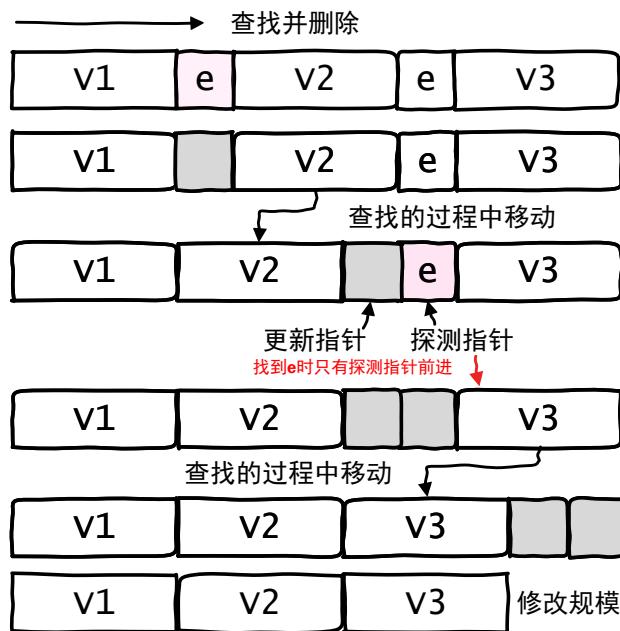


图 2.9 向量按值删除元素：快慢指针

```

1 // VectorRemoveErase
2 void batchRemove(int e) override {
3     V.resize(remove(begin(V), end(V), e) - begin(V));
4 }
```

这里使用了 STL 提供的 `std::remove`, 它并不会真正地将 `e` 删除, 而是将非 `e` 的元素都移动到向量的前半部分, 并返回新的尾部迭代器。用户还需要进行一次 `resize` 才能真正清除掉已经无效的后半部分, 这个真正清除的过程被称为 **擦除** (`erase`)；这个方法也被称为“删除-擦除”法。STL 中的向量容器 `std::vector` 提供了原生的 `erase` 方法, 您可以直接使用它进行按值删除。

2.6 置乱和排序

一般数据结构重点讨论的只有插入、删除和查找三种基本操作, 但向量作为一种非常基础的数据结构, 经常被用来在考试中作为算法设计题的背景。下面这两个小节分别从熵增和熵减的角度出发, 讨论**置乱** (`shuffle`) 和**排序** (`sort`) 的算法。

2.6.1 实验：随机置乱 *

通常说的置乱都是指随机置乱。给定一个向量 `V` 和一个随机数发生器 `rand`, 随机打乱向量中的元素。在理论分析的时候, 可认为随机数发生器是理想的, 即每次调用能够随机生成一个非负整数。当然现实中的随机数发生器做不到理想, 我们将在本小节的末尾讨论它们的区别。

置乱算法接收一个向量将它置乱。代码可以在 *Shuffle.cpp* 中找到。如果您想到了解决方案，可以先自己实现它。

```
1 class Shuffle : public Algorithm<void(Vector<int>&)> { };
```

直接看这个“向量置乱”的问题，很容易没有头绪。不妨将这个问题迁移到比较熟悉的领域：比如洗牌。想必大家都非常熟悉洗牌。随机置乱的目的和洗牌是一样的，但如果用洗牌的方法去做随机置乱，即抽出一沓牌、把这沓牌放到牌堆底部、再抽一沓牌，则会面临三个问题：

1. 您不知道重复多少次抽牌比较合理。
2. 在有限次抽牌之后，牌的 $n!$ 种随机次序并不是等概率的。
3. 每次抽牌都要伴随大量的元素移动，时间效率非常低下。

解决随机置乱问题可以从上面的第二个问题，也就是“随机次序等概率”入手。为了保证随机次序是等概率的，那么就要构造 $n!$ 种等可能的情况。根据乘法原理，可以很自然地想到，如果将每种次序表示为一个 n 元随机变量组 (X_1, X_2, \dots, X_n) ，其中 X_i 两两独立，并且 X_i 恰好有 i 个等可能的取值，那么这 $n!$ 种次序就是等可能的了。接下来，只需要建立在全排列和这样的 n 元组的一一对应的映射关系即可。当然，不能直接把全排列用上。全排列的两个元素不是相互独立的，它自身不是符合条件的 n 元组。

为了构造符合条件的映射，又可以采用递归的思想方法：

1. 如果 $n = 1$ ，全排列和 n 元组可以直接对应。
2. 对于 $n > 1$ ，考虑 $V[n - 1]$ 在打乱后的秩，显然，它可以取 $0, 1, \dots, n - 1$ 这 n 个等可能的值，令这个数为 X_n ，然后将 $V[n - 1]$ 从打乱前后的向量中都删除，就化为了 $n - 1$ 的情况。反复利用这个化归方法，最终可化归到 $n = 1$ 的情况。

以上就成功构造出了满足条件的一一映射关系，您可以在理解它的基础上自己设计相应的随机置乱算法。下面给出了一个示例实现。

```
1 void operator()(Vector<int>& V) override {
2     for (auto i { V.size() }; i > 1; --i) {
3         auto j { rand() % i };
4         swap(V[i - 1], V[j]);
5     }
6 }
```

显然上面这个算法是时间 $\Theta(n)$ 、空间 $O(1)$ 的。并且上面的分析表明，如果 `rand` 真的能随机生成一个非负整数（不是随机生成一个 `unsigned int`！），那么这个算法就能将所有的 $n!$ 个排列等概率地输出。此外，C++11 在 STL 中也提供了置乱算法，需要包含 `<random>` 库使用。

```

1 class ShuffleStd : public Shuffle {
2     default_random_engine m_engine { random_device {}() };
3 public:
4     void operator()(Vector<int>& V) override {
5         shuffle(begin(V), end(V), m_engine);
6     }
7 };

```

这里的默认随机数引擎可以被替换为其他用户定义的引擎，关于随机数引擎的问题和数据结构无关，不再赘述。默认的随机数引擎基于梅森旋转算法，产生随机数的速度较慢，但具有较好的均匀性。图2.10展示了4个元素随机置乱 10^6 次的结果，全部的24种可能情况出现的频率非常接近，平均相对误差只有大约0.4%。

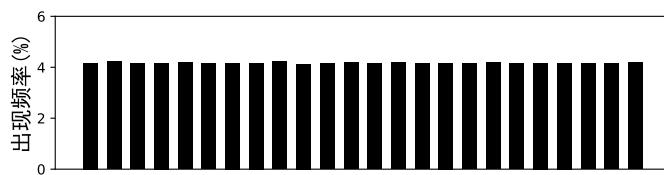


图 2.10 随机置乱的结果

下面回到随机生成器的问题上来，真实的rand会受到位宽的限制。如果每次随机生成一个随机的32位非负整数，那么 n 次随机一共只有 $2^{32n} = o(n!)$ 种可能的取值，所以在 n 充分大的时候，必然会有一些排列不可能被输出。

另一方面，即使忽略位宽的限制，也不可能做到等概率输出。因为当随机数生成器的返回值是在 $[0, 2^k - 1]$ 中随机生成的非负整数时， $n!$ 在 $n \geq 3$ 时不是 2^k 的因子（不论 k 有多大），所以这 $n!$ 个排列不可能是等概率的。不过，当 n 比较小时概率可以认为近似相等，您可以在示例程序的运行结果中看出这一点。

除此之外，现实中的rand是伪随机。对于同一个种子，生成的伪随机序列是相同的，所以并不能真正“随机”地打乱向量中的元素。当然，这是另一个话题了。当我们在后面的章节讨论散列的时候，再对伪随机问题进一步探讨。

2.6.2 偏序和全序 *

在讨论完置乱问题之后，接下来讨论排序问题。在具体介绍排序算法前，首先需要界定清除，序（order）是一个什么东西。在上一章定义过良序的概念，但要对一个向量做排序，并不一定要要求它的元素是某个定义了良序关系的类型。比如说， n 个实数同样可以关于熟知的“ \leq ”排序。因此，需要引入条件更松的序关系的定义。

将良序关系定义中的第4个条件（最小值）去掉，就变成了全序（total order）关系。如果集合 S 上的一个关系 \preceq 满足：

1. **完全性。** $x \leq y$ 和 $y \leq x$ 至少有一个成立。
2. **传递性。** 如果 $x \leq y$ 且 $y \leq z$, 那么 $x \leq z$ 。
3. **反对称性。** 如果 $x \leq y$ 和 $y \leq x$ 均成立, 那么 $x = y$ 。

那么称 \leq 是 S 上的一个**全序关系**。显然良序关系是全序关系的子集。

和良序关系相比, 全序关系更加符合常规的认知。比如, 实数集上熟知的“ $<$ ”就是全序关系。由于完全性的存在, 凡是具有全序关系的数据类型, 都可以进行排序; 反之, 在《数据结构》里的“通常意义的排序”问题中, 都假定数据结构中的元素数据类型具有“先验的全序关系”。

在 C++ 中, 排序函数`std::sort`接受三个参数, 其中第三个参数就表示“自定义的全序关系”。基本数据类型(如`int`和`double`), 以及一些组合类型(如`std::tuple`)定义了内置的全序关系(即熟知的“ $<$ ”), 但也可以使用其他的全序关系进行排序。其他编程语言中的排序函数也有类似的设计。

除了全序关系之外, 还有一种序关系在《数据结构》中也经常会提到: **偏序**(partial order) 关系。

如果集合 S 上的一个关系 \leq 满足:

1. **自反性。** $x \leq x$ 。
2. **传递性。** 如果 $x \leq y$ 且 $y \leq z$, 那么 $x \leq z$ 。
3. **反对称性。** 如果 $x \leq y$ 和 $y \leq x$ 均成立, 那么 $x = y$ 。

那么称 \leq 是 S 上的一个**偏序关系**。偏序关系和全序关系相比, 第 1 个条件(完全性)变成了更简单的自反性; 也就是说, 并不是 S 中的任意两个元素都能进行比较。比如, 令 S 为“考生组成的集合”, \leq 定义为“考生 x 的每一门分数都小于等于考生 y ”。您可以轻易验证, 这个关系是偏序关系但不是全序关系。

在计算机编程中直接定义偏序关系是不方便的, 因为 \leq 的返回值往往是`bool`类型, 不存在`true`和`false`之外的第三个选项(无法比较)。并且, 无法比较的情况不能随意地返回一个`true`或`false`的值, 因为这可能导致传递性被破坏。所以, 当在编程时需要定义一个偏序关系时, 往往会将它扩展成一个全序关系。比如, 给 S 中的所有元素做标号, 当已有的偏序关系无法比较时, 则根据标号的大小进行比较。扩展成全序关系之后, 就可以进行排序了。由扩展成的全序关系的不同, 可能会产生不同的排序结果。

在 C++20 中定义了各种序关系, 用于作为航天飞机运算符`operator<=`的返回值。比如, 偏序关系被定义为`std::partial_ordering`, 这个类型包括大于、小于、等价以及无法比较四种比较结果。除此之外, 还提供了包括大于、小于和等于的强序关系`std::strong_ordering`和包括大于、小于和等价的弱序关系`std::weak_ordering`。

`::weak_ordering`。这两者的区别在于“等于”和“等价”，或者说“相同”和“相等”。强序关系不允许两个不同的元素相等（这是非常强的条件），而弱序关系允许。一般而言，全序关系即对应强序关系，而经过一个映射的全序关系（如二维坐标只比较一个维度）则对应弱序关系。

2.6.3 自上而下的归并排序

现在回到向量排序的问题。对于一个线性表，如果它的数据类型是全序的；且对其中的任意一个元素 x ，和 x 的后缀中的任意一个元素 y ，总是有 $x \leq y$ ，则称它是**有序的** (ordered)。对于无序线性表，通过移动元素位置使其变为有序的过程，称为**排序** (sort)。在计算机领域所说的有序，一般都是指升序。所以在上面的定义中使用的是后缀。如果您想要讨论降序或者其他顺序（比如按最小素因子排序），只需要重新定义全序关系 \leq ，即可以回归为升序的情况进行处理。

```

1 template <typename T, template<typename> typename Linear>
2     requires std::is_base_of_v<AbstractLinearList<T, typename
3         Linear<T>::position_type>, Linear<T>>
4 class AbstractSort : public Algorithm<void(Linear<T>&) > {
5     protected:
6         std::function<bool(const T&, const T&)> cmp { std::less<T>() }
7         };
8         virtual void sort(Linear<T>& L) = 0;
9     public:
10         template <typename Comparator>
11         void operator()(Linear<T>& L, Comparator cmp) {
12             this->cmp = cmp;
13             sort(L);
14         }
15         void operator()(Linear<T>& L) override {
16             sort(L);
17         }
18     };

```

在上面的程序中，我们规定了一个排序的抽象模板类，它接收一个线性表（通过概念限制）对其进行排序。因为排序的结果一定是一个序列，所以我们只考虑线性表。约定俗成地，我们通常使用严格的序关系，比如 $<$ ，而不是不严格的 \leq 。`C++` 提供了模板 `std::less` 来表示使用类型 `T` 定义的小于运算符，用户也可以自定义仿函数传入这个模板中。

排序是计算机领域最重要的算法之一。在计算机出现至今，人们提出了各种各样的排序算法，并且仍然有不少研究者在从事着排序算法的研究。在《数据结构》中，将专门有一章讨论各种排序算法。在本节，先介绍一种最基本、最经典的排序方法：**归并排序** (merge sort) [14]。归并排序的发明人是大名鼎鼎的冯·诺依曼 (von Neumann)，这位“计算机之父”在 1945 年设计并实现了该算法。

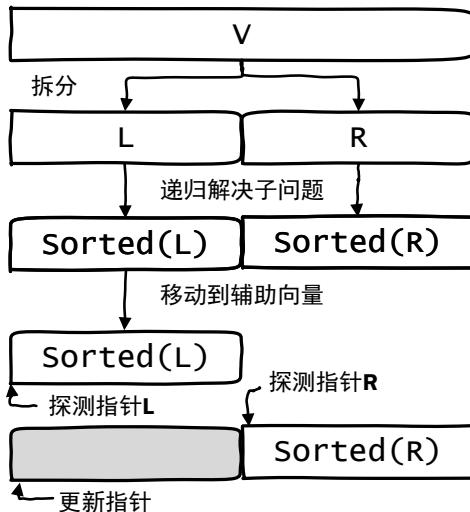


图 2.11 向量的归并排序

归并排序的设计采用的仍然是递归的思想：

1. 规模 $n \leq 1$ 的向量总是天然有序的。
2. 对于规模 $n > 1$ 的向量，可以将其分成前后两部分，长度分别为 $\frac{n}{2}$ 和 $n - \frac{n}{2}$ （在上一章您见过这个分法），从而将规模为 n 的问题化归为两个规模较小的子问题。这些子问题可以继续递归下去直到化为 1。解决子问题之后， V 的前半部分和后半部分分别有序，只需要将这 2 个有序序列合并为 1 个有序序列，就可以解决原问题了。这一合并的过程就称为归并（merge）。

您可以根据上面的思想，自己实现一个归并排序的算法（一个建议是，将归并的过程提取为单独的函数以提高可读性），然后和下面的示例算法进行比较。这个代码比较长。最好自己先写一份代码，因为直接读示例代码很难记住。注意，在《数据结构》部分，代码的记忆既不是重点也没有必要。归并排序这个知识点的核心是上面的这一段文字，即归并的思想。

下面的示例代码既可以用于向量，又可以用于后面的章节介绍的列表。

```

1 template <typename T, template<typename> typename Linear>
2 class MergeSort : public AbstractSort<T, Linear> {
3 protected:
4     Vector<T> W;
5     using Iterator = typename Linear<T>::iterator;
6     void merge(Iterator lo, Iterator mi, Iterator hi, size_t
7     leftSize) {
8         W.resize(leftSize);
9         std::move(lo, mi, std::begin(W));
10        auto i { std::begin(W) };
11        auto j { mi }, k { lo };
12        while (i != std::end(W) && j != hi) {
13            if (*this->cmp(*j, *i)) {
                *k++ = std::move(*j++);
  
```

```

14     } else {
15         *k++ = std::move(*i++);
16     }
17 }
18 std::move(i, std::end(W), k);
19 }
20 void mergeSort(Iterator lo, Iterator hi, size_t size) {
21     if (size < 2) return;
22     auto mi { lo + size / 2 };
23     mergeSort(lo, mi, size / 2);
24     mergeSort(mi, hi, size - size / 2);
25     merge(lo, mi, hi, size / 2);
26 }
27 protected:
28     void sort(Linear<T>& L) override {
29         mergeSort(std::begin(L), std::end(L), L.size());
30     }
31 };

```

C++ 提供了`std::inplace_merge`函数（传入 `lo`、`mi` 和 `hi` 的迭代器）来进行归并，您可以用这个函数代替上述手写的`merge`方法。归并排序的算法中有很多细节可以挖掘，下面列出了其中的一部分。在阅读并理解上述归并排序的算法的基础上，您可以尝试回答以下问题（其中，假定比较函数`cmp`的时间、空间复杂度都是 $O(1)$ 的），然后再查看后面的分析。

在归并的一开始，将前半部分移动到了辅助向量中。为什么前半部分需要移动出去，而后半部分不需要？

在归并的过程中，事实上也采用了快慢指针的思想。快指针（探测指针）是 `j`，慢指针（更新指针）是 `k`。还有一个探测指针是 `i`，不过它工作在辅助向量 `W` 上。如果前半部分不移动的话，`i` 也会工作在原向量上，它和 `k` 不构成快慢关系。于是，一旦后半部分比前半部分的元素小，就会把前半部分的元素覆盖掉；而对后半部分而言，除非前半部分已经全部加入到原向量中，否则 `j` 永远能够在 `k` 前面。而当前半部分全部加入之后，`i < mi - lo` 不再成立，循环结束。

在归并的最后，我们将前半部分（此时在辅助向量里）多余的元素移动回原向量。为什么后半部分多余的元素不需要？

后半部分的数据没有移动出去，如果前半部分的元素已经全加入到原向量了，则后半部分剩余的元素已经在它们应该在的位置上，不需要再移动了。

辅助向量 `W` 的长度至少是多少？并由此确定归并排序的空间复杂度。

辅助向量 `W` 的长度至少为最大的 $mi - lo$ ，也就是 $\frac{n}{2}$ ，因此空间复杂度为 $\Theta(n)$ 。这里需要注意，递归产生的 $\Theta(\log n)$ ，相比于辅助数组的 $\Theta(n)$ 来说可以忽略；但在解答题的场合，应当书写的解题过程中表示考虑到了此种情况。

归并排序的时间复杂度是多少？如果划分的时候，左半部分不取 $\frac{n}{2}$ 而是取 kn （其中 $0 < k < 1$ ），时间复杂度又会变成多少？如果取作 $\max\left(\frac{n}{2}, C\right)$ ，其中 C 是一个给定的常数，那么时间复杂度又会变成多少？

这个问题是归并排序相关的一个经典问题，考试中也可能出现。对于原始的归并排序（折半二分），您可以列出 $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$ 的方程，递降计算出 $T(n) = \Theta(n \log n)$ 。当取 kn （定比二分）时做法类似，时间复杂度不会变，但常数会增加，您可以自行计算。如果左半部分的长度存在上限 C （定长二分），则在 n 充分大时， $T(n) = T(n - C) + T(C) + \Theta(n) = \Theta(n^2)$ 。因此两部分的划分必须按比例取，而不能受到某个固定值 C 的限制。

此类问题通常可以用**主定理** (master theorem)^[15]解决。主定理是用来处理分治算法得到的递归关系式的“神兵利器”。它的证明太过复杂，这里只陈述结论。

定理 2.1 (主定理): 设 $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ ，则：

1. 若 $f(n) = O(n^{\log_b a - \varepsilon})$ ，其中 $\varepsilon > 0$ ，则 $T(n) = \Theta(n^{\log_b a})$ 。
2. 若 $f(n) = \Theta(n^{\log_b a})$ ，则 $T(n) = \Theta(n^{\log_b a} \log n)$ 。
3. 若 $f(n) = \Omega(n^{\log_b a + \varepsilon})$ ，其中 $\varepsilon > 0$ ，且 $\overline{\lim}_{n \rightarrow \infty} \frac{f\left(\frac{n}{b}\right)}{f(n)} < 1$ ，则 $T(n) = \Theta(f(n))$ 。

2.6.4 自下而上的归并排序 *

上一节中展示的归并排序是自上而下的。这里的“自上而下”指的是，我们首先调用整个向量的归并排序 $\text{mergeSort}(V)$ ，在这个函数中递归地调用子向量的归并排序 $\text{mergeSort}(L)$ 和 $\text{mergeSort}(R)$ 。以此类推，直到“最下方”的递归实例，也就是递归边界（只有一个元素的向量）。

自上而下的归并排序是归并排序的经典实现，下面介绍一种自下而上的方法。我们分析归并函数 merge 的调用次序会发现，由于归并发生在递归调用之后，所以归并的次序反而是自下而上的：首先对“最下方”的子向量做归并，得到长度为 2 的有序子向量。一个子向量可以被归并，当且仅当它的左半和右半的子向量已经被归并完成。

如图2.12中左图所示，在自上而下的归并排序中，只要左半和右半的子向量都已经被归并完成，就会引发这两个子向量的归并。对于 7 个元素组成的向量，一共会触发 6 次归并，归并的次序在图中用 1 至 6 表示。由于一个向量的归并，只需要保证它左半和右半归并完成之后进行，而并不要求在左半和右半归并完成之后立即进行，所以我们可以调整这 6 次归并的次序。我们首先将向量视作长度为 1 的段两两合并，然后将它视作长度为 2 的段（每个段内已经有序）两两合并，再视作长度为 4 的有序段两两合并，以此类推，直到归并整个向量为止。上面的每一步

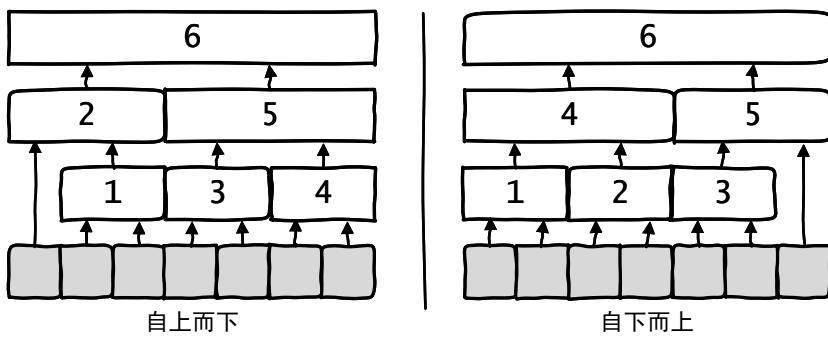


图 2.12 自上而下和自下而上的归并次序

称为一趟 (run)。这种方法同样保证了一个子向量晚于它的左半和右半被归并，因而也能保证正确性。

如图所示，因为向量规模不一定是 2 的幂次，每一趟的末尾处需要特殊处理。从图中还可以看出，自下而上的归并排序对于子向量的分拆方式，是有可能和自上而下不同的。下面展示了自下而上的归并排序的一个实现，它复用了自上而下版本的 merge 函数，只是在调用 merge 的次序上和自上而下的版本有所区别。

```

1 template <typename T, template<typename> typename Linear>
2 class MergeSortUpward : public MergeSort<T, Linear> {
3 protected:
4     void sort(Linear<T>& L) override {
5         auto n { L.size() };
6         for (size_t width { 1 }; width < n; width *= 2) {
7             auto lo { std::begin(L) };
8             Rank i { 2 * width };
9             while (i < n) {
10                 auto mi { lo + width };
11                 auto hi { mi + width };
12                 this->merge(lo, mi, hi, width);
13                 lo = hi;
14                 i += 2 * width;
15             }
16             if (n + width > i) {
17                 auto mi { lo + width };
18                 auto hi { std::end(L) };
19                 this->merge(lo, mi, hi, width);
20             }
21         }
22     }
23 }
```

外层循环进行的次数（也就是趟数）很显然是 $\lceil \log n \rceil$ 。在每一趟中，各个归并段是互不重叠，且可以证明至少有 $\frac{2}{3}$ 的元素参加了归并，所以每一趟的时间复杂度为 $\Theta(n)$ 。因此，自下而上的归并排序时间复杂度同样是 $\Theta(n \log n)$ 。

自下而上的归并排序的时间消耗随 n 的增长是不平滑的。当 n 从 2^k 增长到 $2^k + 1$ 时，自下而上的归并排序需要增加整整一趟；因此，它在 n 略小于或等于一个 2 的幂次的时候表现更好，而在略大于一个 2 的幂次时表现较差。关于这部分性能的具体讨论，参见《排序》章中的 Tim 排序。

2.6.5 基于比较的排序的时间复杂度

归并排序是一种**基于比较** (comparison-based) 的排序。所谓基于比较，就是在算法进行过程的每一步，都依赖于元素的比较（即调用 `cmp`）进行。基于比较的排序是针对全序关系设计的。大多数的排序算法都是基于比较的。还有一些不基于比较的排序，它们不是针对待排序数据类型的全序性设计的，而是针对待排序数据类型的其他性质设计的，因而应用范围会更小。在后文中会介绍一些不基于比较的排序。

下面将说明一个重要结论：

定理 2.2： 基于比较的排序在最坏情况下的时间复杂度为 $\Omega(n \log n)$ 。

这是本书中第一次使用信息论方法，讨论时间复杂度的最优性。信息论方法在考试中通常不会直接考到，但用信息论的思路，有助于在算法设计题中判断自己是否能得到时间复杂度层面上的满分。此外，信息论方法也对记忆知识点有帮助。

1. 因为是最坏情况，不妨假设所有元素互不相等。在排序算法开始之前，这 n 个元素可能的顺序关系有 $n!$ 种，而在排序算法开始之后，这 n 个元素可能的顺序关系只有 1 种（因为已经找到了它们的顺序）。
2. 另一方面，每次比较都有两种结果（`if` 分支和 `else` 分支）。剩下的可能的顺序关系被分为 2 个部分，根据比较结果，只保留其中的 1 个部分。在最坏情况下，每次保留的都是元素较多的部分，从而每次比较至多排除一半的可能。

综合以上两点，至少需要进行 $\log_2(n!) = \Theta(n \log n)$ 次比较，于是就证明了上述的定理。最后一步的结论基于一个重要的公式：

定理 2.3 (斯特林公式)： 在 $n \rightarrow \infty$ 时， $n! \sim \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n$ 。

这一公式的证明是纯数学的话题。感兴趣的话可自行在网上查找，这里不再叙述。在《数据结构》的学习中需要记住的公式并不多，斯特林 (Stirling) 公式是必须记住的公式之一。或者，您也可以只记住 $\log(n!) = \Theta(n \log n)$ ，因为这是它的主要应用。

由此可见，归并排序在基于比较的排序中，已经达到了最优的时间复杂度。当然，空间复杂度不是最优的，它需要 $\Theta(n)$ 的额外空间。关于排序的更多性质，在后面的专门章节中将继续分析。从这个时间复杂度下界也可以看出，排序需要付出的努力总是比置乱要高，这也符合我们对信息的一般认知：排序是熵减的过程，

而置乱是熵增的过程，熵减总是要付出更多的努力。

2.6.6 信息熵 *

基于上一小节的讨论，本小节对香农（Shannon）提出的信息熵（information entropy）概念进行简要的介绍。在使用信息论判断复杂度问题时，并不一定需要使用信息熵去定量计算，因此如果您不感兴趣，也可以跳过本节。

在信息熵提出之前，人们很难定量地衡量一份信息所包含的信息量。香农创造性地引入概率论和热力学中的熵的概念，对信息的多少进行了定量描述。对于一个信息来说，在我们识别之前，会对它有一些先验的了解。如果我们已经先验地确切知道这个信息的内容，那么这个信息就完全是无效信息，所蕴含的信息量是 0；反过来，我们对这个信息的先验了解越少、越模糊，这个信息所蕴含的信息量越丰富。

假设一个信息在我们已知的先验了解下，共有 n 种可能发生的情况。则对于每个情况，设其发生的概率为 p ，我们定义该情况的不确定性为 $-\log p$ 。对于一个信息整体，我们考虑它所有可能发生的情况的平均不确定性（即数学期望），定义其为该信息的信息熵，即

$$H = E(-\log p) = - \sum_{i=1}^n p_i \log p_i$$

现在联系上一小节讨论的场景。在没有其他先验信息的情况下，一个乱序序列出现 $n!$ 种排列的可能性是相等的，所以它的信息熵为：

$$H = -\log\left(\frac{1}{n!}\right) = \Theta(n \log n)$$

在排序结束时，序列只有唯一的可能性，所以信息熵为 0。在最坏情况下，我们进行 1 次比较-判断可以最多消除一半的不确定性，由于取了对数，所以一次判断能造成的熵减是 $O(1)$ 的。综上所述，基于比较的排序的最坏时间复杂度是 $\Omega(n \log n)$ 的。

需要注意的是，即使信息熵的初值和终值都为 0，也不代表可以在 $O(1)$ 的时间内完成算法。比如，对于完全倒序的序列来说，它的信息熵也为 0，但是将其进行排序需要 $\Theta(n)$ 的时间对序列进行倒置。所以，信息熵方法通常只能求出一个理论边界，并不代表这个边界是可以达到的。

2.6.7 有序性和逆序对 *

前面两个小节的讨论显示，对于没有任何先验信息的乱序序列来说，它的信息熵是 $\Theta(n \log n)$ 的，因此基于比较的排序在最坏情况下，永远不可能突破这一时间复杂度限制。但是，如果我们事先了解到一些先验信息，初始状态的信息熵就会下降，从而在理论上可以以更低的时间复杂度进行排序。

一个比较常见的情况是“基本有序”的条件。对于基本有序，通常有两种理解方式。

1. 认为基本有序就是信息熵很低的状态。我们知道，信息熵对应了信息的不确定性，也就是“无序性”，信息熵比较高的序列无序性也比较高。因此，反过来也可以认为，信息熵比较低的序列基本有序。
2. 认为基本有序指的是逆序对很少的状态。对于一个序列 $A[0:n]$ ，如果 $i < j$ 但 $A[i] > A[j]$ ，则称 (i, j) 是一个逆序对。采用逆序对对序列有序性进行刻画，和信息论方法是分离的（因为一次交换可以最多消除 $\Theta(n)$ 个逆序对），但可以对顺序和倒序进行区分，在分析算法时常常可以起到重要的作用。

信息熵和逆序对都是我们分析和解决排序问题的方法。信息熵的视角更加宏观，我们很难计算每一步操作削减了多少信息熵，因此它往往用于复杂度层面上的分析；而逆序对的视角更加微观，很容易计算每一步操作消除了几个逆序对，所以可以用于具体的算法性质分析。下面我们从逆序对的角度回顾归并的过程。每次向更新指针的位置移动一个元素：

1. 如果被移动的元素来自于前半部分的探测指针，那么不会对逆序对的数量产生影响。
2. 如果被移动的元素来自于后半部分的探测指针，那么我们消除了它和前半部分剩余元素之间的逆序对。也就是说，我们消除的逆序对数量等于前半部分的剩余元素数量。

根据这一性质，我们可以在归并排序的过程中统计逆序对的数量。因为在前半和后半之一的元素被用尽之后，后半元素不需要移动，而前半元素需要从辅助空间中移回；所以，上面讨论的情况（2）会比情况（1）有数量更多的移动。因此我们可以看出，归并排序在处理完全倒序的序列时，尽管它的信息熵为 0，但排序算法并不能发现这一先验信息，而是会进行更多的移动。

2.6.8 实验：先验条件下的归并排序 *

下面讨论一个基本有序的场景：如果向量已经基本有序，只有开头的长度为 L （未知）的一小段前缀是乱序的（即前缀外全部有序，且前缀中的元素都比前缀

外的元素小), 如何改进我们的归并排序, 让它可以有更高的时间效率? 改进之后的时间复杂度是多少?

这个问题的示例代码可以在 *VectorMergeSort.cpp* 中找到, 您可以根据自己的理解改写归并排序, 并和示例代码进行比较。

这个问题也是一个排序经典问题。我们可以用信息熵和逆序对两种方法对这个场景进行分析。我们可以发现在这个场景中, 乱序前缀之外的所有元素既不会提供信息熵也不会提供逆序对, 因此在给定的先验条件下, 序列的信息熵为 $\Theta(L \log L)$, 逆序对数量为 $O(L^2)$ 。而原有的归并排序算法在最好情况下, 时间复杂度也是 $\Theta(n \log n)$ 的。所以必须要改进。改进的时候, 可以针对已知的方程 $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$ 做优化。

最直接的想法是从递归方程的目标入手, 也就是将 $T(n)$ 替换为 $T(L)$ 。从信息熵和逆序对的角度我们都可以发现, 问题中的特殊场景相当于把排序问题的规模从 n 降低到了 L 。我们可以从后向前遍历整个向量, 以确定 L 的值。这种方法看起来非常直观, 但确定 L 并没有那么简单。示例代码中给出了一个样例, 它通过 $\Theta(n)$ 的时间找到 L , 您可以自己寻找解决方法并和它对比。

```

1 void sort(Linear<T>& L) override {
2     Rank mid { L.size() - 1 };
3     while (this->cmp(L[mid - 1], L[mid])) {
4         if (--mid == 0) return;
5     }
6     auto max_left { *max_element(begin(L), begin(L) + mid) };
7     auto left { lower_bound(begin(L) + mid, end(L), max_left,
8         this->cmp) };
9     this->mergeSort(begin(L), left, left - begin(L));
}
```

另一个思路是从递归方程的形式入手。注意到, 在这个方程中, 递归项 $2T\left(\frac{n}{2}\right)$ 只要不改动递归方式, 就是没法做优化的; 而余项 $\Theta(n)$ 是有机会被优化的。需要在“比较好的情况”(即题中给出的“基本有序”的情况)下, 让 $\Theta(n)$ 变得更小。归并排序在“将前半部分移动到辅助空间”的时候, 就已经需要付出 $\Theta(n)$ 的时间。所以, 我们需要在归并的最开始进行一次判断, 判断是否可以不将前半部分移动到辅助空间: 只需要判断前半部分的结尾是否小于后半部分的开头就可以。在我们之前实现的归并算法中, 只需要加入下面的一行代码:

```
1 if (cmp(*mi - 1, *mi)) return;
```

这样, 如果归并前的序列已经有序, 就不需要进行归并。于是, 对于不需要归并的部分, 递归方程就变化为 $T(n) = 2T\left(\frac{n}{2}\right) + O(1)$, 也就是 $\Theta(n)$ 。而需要归并的部分由题意, 长度不超过 L , 在这部分利用前面获得的结论, 就可以得到时间复杂

度为 $\Theta(L \log L)$ 。因此，改进后的时间复杂度为 $\Theta(n + L \log L)$ 。

由于归并排序的实际时间性能还和很多其他的因素相关，所以本书提供的实验代码只能大致地进行定性分析。如果您想要得到更加精准的分析结果，应当多次随机取平均值。我们可以从实验结果中看到，在题目给定的条件（前缀乱序）下，两种改进策略的时间性能差不多；同时在 L 不大的时候，两种改进策略的性能都显著高于未改进的版本。您可以自己用上面介绍的两种方法，对自下而上的归并排序进行修改，使其时间复杂度达到 $\Theta(n + L \log L)$ ，并加入实验框架进行比较。

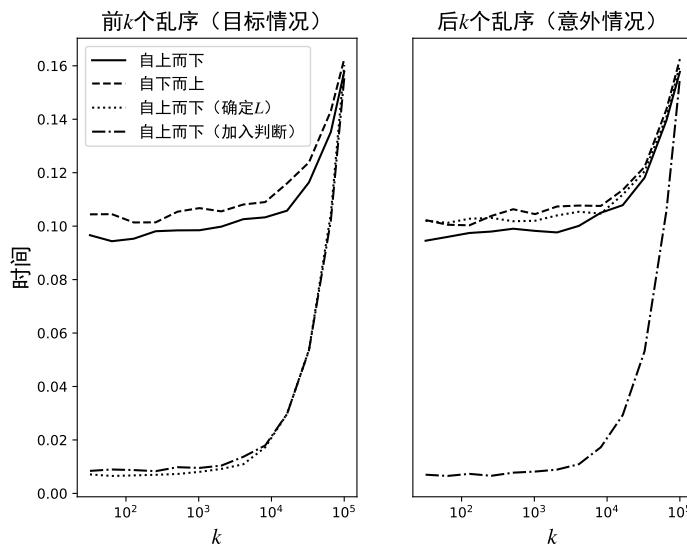


图 2.13 归并排序在局部乱序情况下的性能

如图2.13所示，在本小节介绍的两种方法中，从递归方程入手的方法具有更好的泛用性。我们可以测试一个对偶问题：有且只有后缀是乱序的问题。在这个对偶问题上，从递归方程入手的方法仍然可以做到高效率，而从递归目标入手的方法则因为无法识别先验信息，和未改进的版本性能相若。从图中还可以看出，即使是没有做针对性优化的版本，由于有序段内不需要移动后半段，在“整体有序”的情况下性能也可以得到一定的提升。

2.7 有序向量上的算法

2.7.1 折半查找

排序之后得到的有序向量，在查找时有额外的优越性。排序之后要执行查找操作，就不再需要一个一个元素看是否相等了。类似排序，我们首先建立针对有序线性表的抽象查找类。

```

1 template <typename T, template <typename> typename Linear =
2     DefaultVector>
3     requires std::is_base_of_v<AbstractLinearList<T, typename
4         Linear<T>::position_type>, Linear<T>>
5     class AbstractSearch : public Algorithm<typename Linear<T>::
6         position_type(const Linear<T>&, const T&)> {
7     protected:
8         using output_type = typename Linear<T>::position_type;
9         std::function<bool(const T&, const T&)> cmp { std::less<T>()
10            };
11        virtual output_type search(const Linear<T>& L, const T& e) =
12            0;
13    public:
14        template <typename Comparator>
15        output_type operator()(const Linear<T>& L, const T& e,
16            Comparator cmp) {
17            this->cmp = cmp;
18            return search(L, e);
19        }
20        output_type operator()(const Linear<T>& L, const T& e)
21            override {
22                return search(L, e);
23            }
24    };

```

这里可以使用刚才介绍的，基于比较的算法思路。将被查找的元素 e 和向量中的某个元素 $V[i]$ 比较，比较结果有 2 种：

1. 如果 $V[i] > e$ ，那么只需要保留 $V[0:i]$ 作为新的查找区间。
2. 如果 $V[i] \leq e$ ，那么只需要保留 $V[i:n]$ 作为新的查找区间。

当取 $i = \frac{n}{2}$ （折半）时，可以保证新的查找区间长度大约是原来的一半，从而在 $\Theta(\log n)$ 的时间里完成查找。所以这个思路称为**折半查找**。当然，也存在其他二分的方法（ i 取其他值），参见后面的《查找》一章。

您可以借助上面的设计或者自己的理解，设计折半查找的算法。下面给出了一个使用递归的示例代码，它实现了刚才的设计。

```

1 template <typename T>
2 class BinarySearchRecursive : public AbstractSearch<T,
3     DefaultVector> {
4     Rank search(const Vector<T>& V, const T& e, Rank lo, Rank hi
5     ) const {
6         if (hi - lo <= 1) {
7             return lo < V.size() && this->cmp(V[lo], e) ? hi : lo;
8         }
9         Rank mi { lo + (hi - lo) / 2 };
10        if (this->cmp(e, V[mi])) {
11            return search(V, e, lo, mi);
12        } else {
13            return search(V, e, mi, hi);
14        }
15    };

```

```

13     }
14 protected:
15     Rank search(const Vector<T>& V, const T& e) override {
16         return search(V, e, 0, V.size());
17     }
18 };

```

上面的算法，时间复杂度和空间复杂度均为 $\Theta(\log n)$ ，您可以自己证明。

查找是算法设计的重点。在设计的时候，需要尤其注意多个相等元素的时候是返回秩最大、秩最小还是任意一个，以及查找失败的时候返回何种特殊值。如果是无序向量，正如2.5.3节那样，那么在查找失败的时候很自然地会返回一个无效值，比如说 -1 或者 $V.size()$ 。但是在有序向量的情况下，即使查找失败，我们也可以返回一些有意义的值，向调用者传递一些额外的信息。下面以前面的折半查找算法为例，分析查找成功和查找失败的情况下返回值的设计。

因为如果 $e < V[mi]$ 就只保留前半段 $V[lo:mi]$ ，所以我们在任何时刻都可以保证， $V[hi] > e$ （可认为初始值 $V[V.size()] = +\infty$ ）。另一方面，因为另一边的比较是不严格的，所以我们保证的是 $V[lo] \leq e$ ；注意，这个式子只有 $lo \neq 0$ 也就是 lo 被修改过一次之后才成立。

而当进入递归边界的时候， $V[lo:hi]$ 有且只有一个元素 $V[lo]$ 。此时，可以分成小于、等于、大于三种情况讨论。

1. 如果 $V[lo] < e$ ，那么我们可以定位到 $V[lo] < e < V[hi]$ ，此时返回 hi ，就是大于 e 的第一个元素。
2. 如果 $V[lo] = e$ ，那么我们直接返回 lo ，符合查找算法的期待；并且，由于 $e < V[hi]$ ，所以如果有多个等于 e 的元素，则返回的是最大的秩。
3. 如果 $V[lo] > e$ ，根据前面的分析可知，这种情况只可能发生在 $lo = 0$ 的情况。

于是， e 小于向量中的所有元素，此时返回 lo ，也是大于 e 的第一个元素。

综上所述，我们验证了上述折半查找算法的正确性，并且在查找成功时，返回的是最大的秩，在查找失败时，返回的是大于 e 的第一个元素的秩。二分算法在设计的时候非常容易出错；当您自己设计二分算法的时候，也可以使用上面的思考流程来分析自己的算法。

2.7.2 消除尾递归

查找 $V[0:n]$ 中某个元素 e 的下标，这个问题在计算前有 $n+1$ 种（包括 -1 ）可能的结果，计算后有 1 种确定的答案，因此从信息论的角度讲，最坏时间复杂度一定是 $\Omega(\log n)$ 的。但空间复杂度并不一定要是 $\Omega(\log n)$ 。在这一小节，将介绍一种叫做**消除尾递归**的技术，使用这个技术，可以将上面的折半查找算法的空间复杂

度降为 $O(1)$ 。

如果一个递归函数只在返回 (return) 前调用自身，则称其为**尾递归** (tail recursion)。在实际的编程过程中如果开启了编译器优化选项，则尾递归在通常会被编译器自动优化。

本小节只介绍对于尾递归的消除方法，其他类型的递归消除将在后文中讨论。对于尾递归，只需要将递归函数的参数作为循环变量，就可以将其改写为不含递归的形式，从而降低空间复杂度。下面的模板是典型的尾递归。

```

1 R operator()(Args... args) override {
2     if ((*pred)(args...)) {
3         return (*bound)(args...);
4     }
5     return apply(*this, (*next)(args...));
6 }
```

示例代码可以在 *VectorSearch.cpp* 中找到，这里只提取出关键部分。如果您对模板元编程的技巧不感兴趣，下面的说明略读即可。其中，需要传入三个仿函数指针 `pred`（用于判断是否进入递归边界的谓词）、`bound`（用于在递归边界上生成返回值）和 `next`（用于在非递归边界上生成下一个递归调用的参数）。`std::apply` 用于将 `next` 返回的元组再次传递到仿函数中进行调用。此外，递归参数往往是值而不是引用。引用总是作为整个递归过程中共享的参数，如折半查找中的向量 V 和待查找元素 e ，而不是每次递归调用时传递的参数。因此，我们可以直接用值传递 `args...`，而不需要使用 `std::forward` 进行完美转发。上面的递归算法可以改写成如下的迭代算法，其中 `std::tie` 表示元组的绑定赋值。

```

1 R operator()(Args... args) override {
2     while (!(*pred)(args...)) {
3         tie(args...) = (*next)(args...);
4     }
5     return (*bound)(args...);
6 }
```

2.7.3 实验：迭代形式的折半查找

您可以利用上面的尾递归模板，将折半查找递归形式进行拆解，分解出 `pred`、`bound` 和 `next`，然后代入到上面的迭代模板中，将其改写为迭代形式。示例代码仍然在 *VectorSearch.cpp* 中。比如，`next` 可以实现为：

```

1 tuple<Rank, Rank> operator()(Rank lo, Rank hi) override {
2     Rank mi { lo + (hi - lo) / 2 };
3     if (cmp(e, V[mi])) {
4         return { lo, mi };
5     } else {
6         return { mi, hi };
7 }
```

```

7 }
8 }
```

下面是迭代形式的一个示例程序，它和最初的递归形式是完全等价的。

```

1 Rank search(const Vector<T>& V, const T& e) override {
2     Rank lo { 0 }, hi { V.size() };
3     while (hi - lo > 1) {
4         Rank mi { lo + (hi - lo) / 2 };
5         if (this->cmp(e, V[mi])) {
6             hi = mi;
7         } else {
8             lo = mi;
9         }
10    }
11    return lo < V.size() && this->cmp(V[lo], e) ? hi : lo;
12 }
```

在示例的测试程序中，我们构造了长度为 n 的向量，将其随机赋值为某个区间的数并排序，然后重复 10^4 次查找（这是因为单次查找的效率太高，无法正确反映各个算法的性能差异）。我们会发现，由于进行了很多抽象，使用模板的方法性能会显著低于直接递归或迭代的性能；而递归和迭代之间相差无几（编译器对递归的版本进行了自动优化）。但即使成模板，它作为对数复杂度的算法，效率仍然远远高于在2.5.3节中讨论的顺序查找，您可以自己实现一个基于顺序查找的算法参与对比。

在本节的最后再次强调，现代 C++ 编译器如果开启了优化选项，通常可以在编译的过程中自动消除尾递归，所以在实际上机编程时，不需要刻意将尾递归改写成循环形式。但在《数据结构》中分析算法的时候，不应该考虑编译器做的优化，所以在算法设计题中，如果出现了未被改写的尾递归，就会有被扣空间分的风险。此外，您也不应当总是依赖编译器的尾递归优化，因为一些语言（比如 Python）拒绝提供尾递归优化^[16]，在 C++ 中运行良好的代码移植到这些语言上可能会发生问题。

2.7.4 实验：向量唯一化

下面讨论**唯一化** (unique) 问题。给定一个向量，我们希望删除它中间相等的重复元素，只保留秩最小的那个。这里再次看到了相等和相同的区别，数据结构中是不可能有相同的元素的，而相等的元素我们可以用它的地址（位置、秩）来区分。代码可以在 *VectorUnique.cpp* 中找到。

```

1 template <typename T>
2 class VectorUnique : public Algorithm<void, Vector<T>&> {};
```

一个简单但有效的解法是：从左到右考察 V 中的每个元素，删除这个元素的后缀中，所有和它相等的元素。在看下面的代码前，您可以自己实现这个算法。

```

1 // VectorUniqueBasic
2 void operator()(Vector<T>& V) override {
3     for (Rank r { 0 }; r < V.size(); ++r) {
4         V.resize(remove(begin(V) + r + 1, end(V), V[r]) - begin(
5             V));
6     }
}

```

在上面的算法里，利用了我们在2.5.7节中使用的“删除-擦除”法一次性地删除后缀里的所有重复元素；因此，最好情况下（所有元素都相等），可以达到 $\Theta(n)$ 的时间复杂度。虽然按值删除达到了 $\Theta(n - r)$ 的时间复杂度，但是因为 r 需要遍历整个向量，所以在最坏情况下（即所有元素都互不相同）需要进行 $\Theta(n^2)$ 次比较。

值得一提的是，如果按值删除的时候采用的是最坏情况 $\Theta(n^2)$ 的朴素（逐个删除）算法，那么唯一化在最坏情况下时间复杂度仍然是 $\Theta(n^2)$ ，并不会来到 $\Theta(n^3)$ 。这是初学者容易出现的错误，即直接把循环内外的时间复杂度相乘。这种天真的想法可能是来自于公式 $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$ 。这个公式本身没有问题，但我们联系一下概率论里的 $P(A)P(B) = P(AB)$ 的条件就能发现端倪：如果内层循环和外层循环是有联系的（不独立的），那么就不能直接相乘。比如，在上面的代码中，外层循环的 r 的区间是 0 到 $V.size()$ ；然而， $V.size()$ 并不是一个常量 n ，在内层循环中它会发生变化。在使用朴素算法进行按值删除的时候，删除的元素越多，花费的时间越长，但同时缩小的向量规模也就越多，减少的外层循环的轮数也就越多。您可以在这个基础上，完成对朴素按值删除情况下，上述唯一化算法的时间复杂度分析。

在有序向量的情况下，问题可以得到进一步的简化。相等的元素总是排在连续的位置的。所以，可以通过快慢指针的方法。快指针一次经过一片相等的元素，而慢指针保留这些元素中的第一个（这个算法的前提条件并不是有序，只要求相等的元素排在一起即可，但通常建立这一条件的方法是排序）。您已经见过好几次快慢指针的方法，应该可以自己写出来。下面是一个示例程序。

```

1 // VectorUniqueFSP
2 void operator()(Vector<T>& V) override {
3     Rank r { 0 }, s { 0 };
4     while (++s < V.size()) {
5         if (V[r] != V[s]) {
6             V[++r] = move(V[s]);
7         }
8     }
9     V.resize(++r);
10}

```

这个算法的时间复杂度是 $\Theta(n)$ 的。在 STL 中，提供了 `std::unique` 来进行唯一化操作，它同样要求相等的元素排在一起。`std::unique` 和 `std::remove` 一样不提供擦除功能，需要再进行 `resize`。

如果定义了全序关系（即可排序），那么无序向量的唯一化可以化归到有序向量的情况进行处理：先进行一次 $\Theta(n \log n)$ 的排序，再用有序向量唯一化。但在排序的时候，会损失“元素原先的位置”这一信息，所以需要开辟一个额外的 $\Theta(n)$ 的空间保存这一信息，以在唯一化之后能够顺利还原。您可以尝试实现这样的算法，并和给出的示例算法相比较。它的时间复杂度为 $\Theta(n \log n)$ ，优于前面的 `VectorUniqueBasic`；但需要引入辅助向量，空间复杂度为 $\Theta(n)$ 。

```

1  template <typename T>
2  class VectorUniqueSort : public VectorUnique<T> {
3      struct Item {
4          T value;
5          Rank rank;
6          bool operator==(const Item& rhs) const {
7              return value == rhs.value;
8          }
9          auto operator<=(const Item& rhs) const {
10             return value <= rhs.value;
11         }
12     };
13     Vector<Item> W;
14     void moveToW(Vector<T>& V) {
15         W.resize(V.size());
16         for (Rank r { 0 }; r < V.size(); ++r) {
17             W[r].value = move(V[r]);
18             W[r].rank = r;
19         }
20     }
21     void moveToV(Vector<T>& V) {
22         V.resize(W.size());
23         for (Rank r { 0 }; r < W.size(); ++r) {
24             V[r] = move(W[r].value);
25         }
26     }
27 public:
28     void operator()(Vector<T>& V) override {
29         moveToW(V);
30         stable_sort(begin(W), end(W));
31         W.resize(unique(begin(W), end(W)) - begin(W));
32         sort(begin(W), end(W), [] (const Item& lhs, const Item&
33             rhs) {
34             return lhs.rank < rhs.rank;
35         });
36         moveToV(V);
37     }
38 };

```

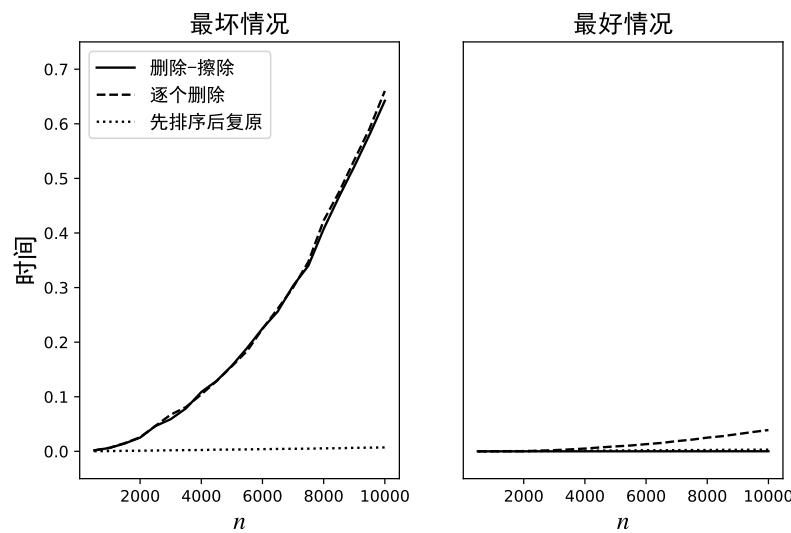


图 2.14 最坏和最好情况下的向量唯一化性能

如图2.14所示，通过实验我们看到，在最坏情况下（所有元素互不相同），直接对无序向量进行处理，无论采用“删除-擦除”法还是逐个删除法都非常慢，而先排序再复原则快得多。而在最好情况下（所有元素都相同），“删除-擦除”法时间复杂度仅为 $\Theta(n)$ 最快，先排序后复原需要 $\Theta(n \log n)$ 较慢，而逐个删除的方法仍然需要 $\Theta(n^2)$ 最慢。

2.8 实验：循环位移

通常我们遍历向量都采用从前向后、一个一个元素遍历的形式。在本章的最后，用**循环位移**（cyclic shift）作为例子，讨论一下非常规的遍历方法。代码可以在 *CyclicShift.cpp* 中找到。

给定向量 $V[0:n]$ 和位移量 k ，则将原有的向量 $V[0], V[1], \dots, V[n - 1]$ ，变换为 $V[k], V[k + 1], \dots, V[n - 1], V[0], V[1], \dots, V[k - 1]$ ，称为**循环左移**。相应地，变换为 $V[n - k], V[n - k + 1], \dots, V[n - 1], V[0], V[1], \dots, V[n - k - 1]$ ，称为**循环右移**。循环左移和循环右移的实现方法大同小异，这一小节只讨论循环左移，右移的情况请您自己完成。建议您先自己设计解决这个问题的算法并实现。

循环左移的过程可以表示为 $(V[0:k], V[k:n]) \rightarrow (V[k:n], V[0:k])$ 。这个形式非常类似于交换。相信您一定知道最经典的交换函数的实现：

```

1 template <typename T>
2 void swap(T& a, T& b) {
3     T tmp { move(a) };
4     a = move(b);
5     b = move(tmp);
6 }
```

一个最朴素的想法，就是用类似的辅助空间，暂存 $V[0:k]$ 中的元素，然后通过 3 次移动来完成循环左移，如图2.15所示。

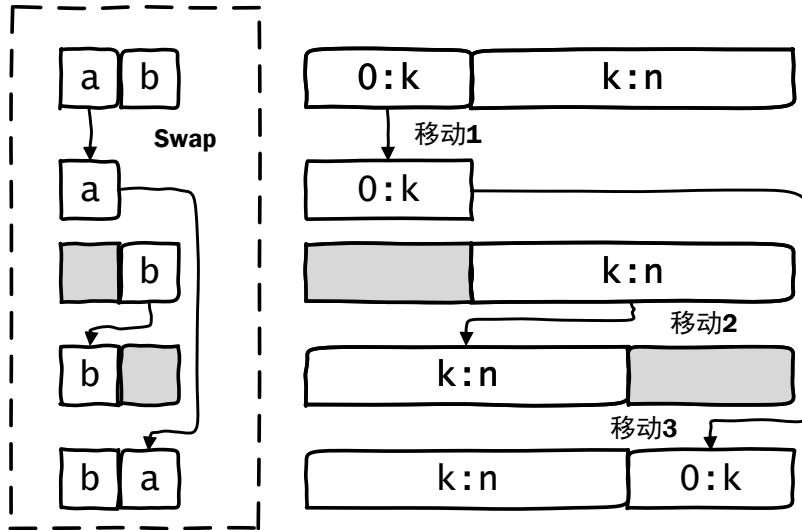


图 2.15 用三次移动实现循环左移，与交换元素的对比

```

1 // CyclicShiftMove
2 void operator()(Vector<T>& V, size_t k) override {
3     Vector<T> W(k);
4     move(begin(V), begin(V) + k, begin(W));
5     move(begin(V) + k, end(V), begin(V));
6     move(begin(W), end(W), end(V) - k);
7 }
```

这一算法的时间复杂度是 $\Theta(k) + \Theta(n - k) + \Theta(k) = \Theta(n + k)$ 。考虑到 $k = O(n)$ ，也可以简化为 $\Theta(n)$ 。空间复杂度则为 $\Theta(k)$ 。

下面的目标则是将空间复杂度降到 $O(1)$ 。为了保持时间复杂度仍然为 $\Theta(n)$ 不变，需要尽可能一步到位地移动元素。当我们让 $V[i + k]$ 移动到 $V[i]$ 的位置上时，需要暂存 $V[i]$ 到辅助空间去。下一步，如果继续将 $V[i + k + 1]$ 移动到 $V[i + 1]$ 的位置，那么需要的辅助空间就会增大。为了防止辅助空间增大，则需要考虑“不用暂存”的元素：也就是已经被移动的 $V[i + k]$ 。下一步将 $V[i + 2k]$ 移动到 $V[i + k]$ 的位置，这就是不需要新的辅助空间的。

因此可以得到一个算法：将 $V[i+k]$ 移动到 $V[i]$ ，再将 $V[i+2k]$ 移动到 $V[i+k]$ ，以此类推。由于向量中的元素是有限的，您可以证明，存在 j ，使得 $(i + jk) \% n = i$ ，也就是经过 j 次移动后回到了 $V[i]$ 。最后一次赋值，将辅助空间里的 $V[i]$ 拿出来赋给 $V[i - k]$ 也就是 $V[i + (j - 1)k]$ 即可。这样实现了一个“轮转交换”的功能。

需要注意的是，这样一轮并不一定能经过 V 中所有的元素。比如在 $n = 6, k = 2, i = 0$ 时，只轮转交换了 $V[0], V[2], V[4]$ 这 3 个元素，而对另外 3 个元素则没有移动。我们可能需要进行多轮“轮转交换”，各轮共享同一个辅助空间。图2.16展示了 $n = 12, k = 3$ 的轮转交换例子。

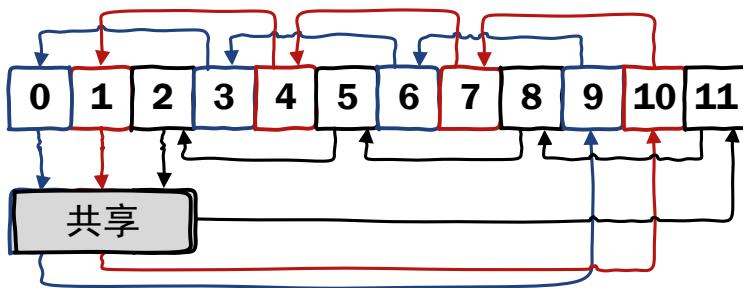


图 2.16 用轮转交换进行循环位移

下面证明：对任意的秩 $0 \leq r < n$ ，都存在唯一的 $0 \leq i < d$ 和 $0 \leq j < \frac{n}{d}$ ，使得 $r = (i + jk) \% n$ 。其中， $d = \gcd(n, k)$ 是 n 和 k 的最大公因数。

证明 因为 r 和数对 (i, j) 的取值范围都是 n 元集，所以只要证明 (i, j) 到 r 是单射，就蕴含了它同时是满射。因此，只需要证明对于不等的 (i_1, j_1) 和 (i_2, j_2) ， $(i_1 + j_1 k) \% n \neq (i_2 + j_2 k) \% n$ 。

假设存在整数 q ，使得 $(i_1 - i_2) + (j_1 - j_2)k + qn = 0$ 。由于 $(j_1 - j_2)k + qn$ 必定是 d 的倍数，而 $|i_1 - i_2| < d$ ，所以只能有 $i_1 = i_2$ 。

设 $k = k_1 d, n = n_1 d$ ，那么 $(j_1 - j_2)k_1 + qn_1 = 0$ 。因为 $(k_1, n_1) = 1$ ，所以 $j_1 - j_2$ 必定是 n_1 的倍数，但 $|j_1 - j_2| < \frac{n}{d} = n_1$ ，所以只能有 $j_1 = j_2$ 。这和 (i_1, j_1) 与 (i_2, j_2) 不等矛盾，故由反证法得到单射成立。■

于是，遍历顺序应当是：依次从 $0, 1, \dots, d - 1$ 出发，以 k 为步长遍历 $\frac{n}{d}$ 次回到起点。这个算法的书写有一定挑战性，您可以效仿前面的 `swap` 去实现轮转交换。

```

1 // CyclicShiftSwap
2 void operator() (Vector<T>& V, size_t k) override {
3     size_t d { static_cast<size_t>(gcd(V.size(), k)) };
4     for (Rank i { 0 }; i < d; ++i) {
5         T tmp { move(V[i]) };
6         Rank cur { i }, next { (cur + k) % V.size() };
7         while (next != i) {
8             V[cur] = move(V[next]);
9             cur = next;
10            next = (cur + k) % V.size();
11        }
12        V[cur] = move(tmp);
13    }
14 }
```

这一算法的时间复杂度是 $\Theta(d) \cdot \Theta\left(\frac{n}{d}\right) = \Theta(n)$, 空间复杂度降低到了 $O(1)$ 。

在很多教材上会介绍另一种解法：三次反转（reverse）。基于反转的原理是，为了实现 $(V[0:k], V[k:n]) \rightarrow (V[k:n], V[0:k])$, 本质上是需要让后半段移动到前半段，前半段移动到后半段。而反转恰好能实现这个功能，且不需要移动算法那样的额外空间。在第一次反转后， $V[0:n]$ 变为了 $\overline{V}[n:0]$ （这里用上划线表示反转），我们可以将它拆分为 $(\overline{V}[n:k], \overline{V}[k:0])$ ，然后再将这两段分别反转即可，如图2.17所示。

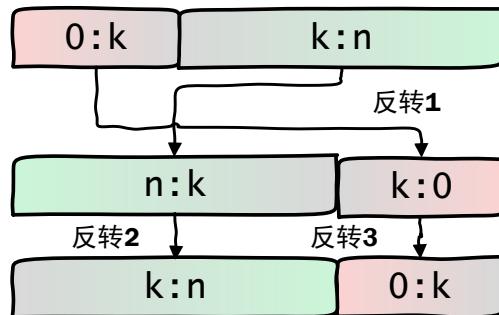


图 2.17 用三次反转进行循环位移

```

1 // CyclicShiftReverse
2 void operator()(Vector<T>& V, size_t k) override {
3     reverse(begin(V), end(V));
4     reverse(begin(V), end(V) - k);
5     reverse(end(V) - k, end(V));
6 }

```

表 2.4 循环位移算法的耗时对比（单位：ms）

位移量 k , $n = 10^8$		三次移动	循环位移	三次反转
1	最小值	2788	1909	3424
10,000	较小值	2769	2250	3418
5×10^7	中间值	4236	1869	3428
9×10^7	较大值	5370	1917	3442
10,007	素数	2751	5783	3439

三次反转的时间复杂度同样是 $\Theta(n)$, 而空间复杂度是 $O(1)$ 。下面从理论层面来比较以上三种做法的时间效率。三次移动的做法，写入内存的次数为 $k + (n - k) + k = n + k$ 。轮转交换的做法，写入次数为 n （每个元素都一步到位地写入了目标位置）。三次反转的做法，写入次数为 $n + (n - k) + k = 2n$ 。显然，轮转交换的写入次数最少，三次移动次之，三次反转最多。但实际上，由于三次移动中开辟 $\Theta(k)$ 的空间（并自动做零初始化）和释放本身需要时间，所以它在 k 比较大的时候反而会慢于

三次反转。您可以从实验结果中看到这一点。此外，您还能从实验结果中看到，虽然理论上轮转交换的写入次数是 n ，但它的时间消耗上下浮动很大（并不像三次移动那样随 k 变大而变大），有些时候时间消耗还会大于三次移动和三次反转。这是算法的局部性导致的，具体机制参见《组成原理》。三次反转虽然很多时候不是速度最快的算法，但对于各种情况的 k 性能相当稳定。

最后，C++ STL 提供了 `std::rotate` 函数进行循环位移。根据编译器的不同，其可能会选用轮转交换或者三次反转的算法（MSVC 使用的是三次反转）。您可以从实验结果中看出您的编译器使用了何种算法实现该函数。

2.9 本章小结

向量（顺序表）是程序设计中最经常使用的数据结构，因此本章具有较大的容量。向量的顺序结构是简单的，向量的循秩访问特性是自然的，无论是学习还是考试，这一章的重点都在算法设计上。本章通过较多的实验展示了一些算法设计的典型技巧。其中，对于排序和查找两个重点内容，我们还将在后续的算法章节再次讨论。

本章的主要学习目标如下：

1. 您有了对算法设计中的不必要工作的优化意识。
2. 您学会了对顺序表的整块进行操作，而不是对单个元素进行操作。
3. 您学会了使用快慢指针进行探测和更新。
4. 您了解到可以从信息的观点分析时间复杂度问题。
5. 您了解到排序预处理可以为后续问题的解决提供帮助。
6. 您了解了分析分摊复杂度和平均复杂度的意义。
7. 您深化了对递归-迭代关系的认识，掌握了消除尾递归的方法。

向量本身是很简单的数据结构，记忆它的插入、删除、查找和各种变形，也是相对简单的；重要的地方在于希望您能理解并掌握书中这些典型算法的设计思路、优化思路。希望本书的内容能为您解决基于向量的算法设计问题提供帮助。

第3章 列表

本章介绍另一种线性表：**列表**（list）。列表和向量的区别在于，列表不要求在内存中占据的空间是连续的。这一特点赋予了列表更大的灵活性，使列表的一些操作比向量效率更高；但同时，这一特点也让列表无法通过秩定位到内存地址，丧失了循秩访问的能力，从而在另一些操作上效率不如向量。本书会介绍列表是如何工作的，至于列表和向量的对比表格，希望读者在阅读后自己完成。

3.1 列表的结构

列表不要求在内存中占据的空间是连续的，因此不能循秩访问，只能循位置访问：通过指向列表中某个元素的指针（也就是该元素的地址）来访问它。那么，如何获得一个元素的地址呢？首先，把所有元素的地址汇总在一张表看起来是可行的，但如果这么做的话，这张表本身如何存储就变成了一个新的问题。如果这个表使用向量存储，那么我们就放弃了列表的不连续的灵活性；如果这个表使用列表存储，那么就成为了一个嵌套的问题。所以，我们不能把所有元素的地址汇总在一张表，也就是说，我们不能采用集中式的地址存储，需要采用分布式的地址存储。

所谓分布式的地址存储，就是我们在每个元素处，同时存储它前一个和后一个元素的地址。这样，我们无论是从前往后还是从后往前，都能遍历整个列表。很明显，这样带来了一个坏处，就是我们只能“逐个”访问元素，而不能像向量那样根据任意的秩访问元素。这就是列表选择更大灵活性的代价。除此之外，列表还有一些其他的代价，比如每个元素处除了它本身的空间，还需要存两个地址，因而有可能需要付出比向量更大的空间（注意这里是有可能，因为向量的装填因子很小时，所造成的大空间浪费更大）。为了降低列表的空间浪费，有时我们会放弃存储前一个元素的地址，只存储后一个元素的地址。这种情况称为**单向列表**（forward list）或单链表，相对应地，同时存储前一个和后一个元素地址的情况称为**双向列表**（bidirectional list）或双链表。

3.2 列表的节点

3.2.1 单向列表节点

为了设计列表的抽象类，我们需要首先将列表中的每个元素抽象出来。列表中每个数据元素及其附加属性（即，前后元素的地址）组成了一个数据单元，或者

称**节点**（node）。《计算机科学技术名词》^[17]认为“节点”是数据结构中数据元素的连接点或端点，而“结点”是计算机网络中的网络拓扑设备，并注明二者互为别称。本书不做区分。本着从简到繁的精神，我们从单向列表的节点开始。

```

1 template <typename T>
2 class ForwardListNode {
3     T m_data;
4     ForwardListNodeInst<T> m_next { };
5 public:
6     ForwardListNode() = default;
7     ForwardListNode(const T& data) : m_data(data) {}
8     T& data() { return m_data; }
9     ForwardListNodeInst<T>& next() { return m_next; }
10 };

```

以下对**ForwardListNodeInst**这个类型进行说明。在邓书中，列表中的“位置”**ListNodePos**直接被定义为**ListNode**的裸指针类型。本书希望引导您学习更加丰富的现代 C++ 编程范式，而智能指针是其中的重要一环，因此在本书的示例代码中，定义了三种代理（proxy）类型^[4]，分别对应裸指针、**const**裸指针和**std::unique_ptr**智能指针。其具体实现详见 PointerProxy.ixx。

1. 以**Pos**结尾的类型是裸指针的代理。非空时，它表示一个实例的位置。
2. 以**ConstPos**结尾的类型是**const**裸指针的代理。非空时，它表示一个实例的位置，且不允许通过此代理修改实例。
3. 以**Inst**结尾的类型是**std::unique_ptr**智能指针的代理。非空时，它持有一个实例的所有权。

上述三种类型与其代理的类型之间支持隐式转换，智能指针代理也可以被隐式转换为裸指针。这使得不了解智能指针的读者在使用本书的框架进行实验时，不需要经常通过**get**方法转换为裸指针，也不需要经常用**std::move**方法传递所有权。尽管如此，智能指针对您掌握现代 C++ 的编程技巧是非常有益的，如果有时间，我仍然非常推荐您学习它。本书的正文及示例代码都将以智能指针为主。

回到单向列表的节点上。单向列表因为只有后向指针，所以我们将它声明为智能指针，让前一个节点具有后一个节点的所有权。这样，我们只要释放一个节点，就可以自动地释放掉它的所有后继。不过，这个想法看起来很美好，实际上却会出现问题。

对于规模为 n 的列表 L 来说，如果我们清空列表，释放它的第一个节点 $L[0]$ （请注意，虽然可以这样表记，但列表是不能循秩访问的），那么在 $L[0]$ 的析构函数中会释放 $L[0]$ 的后向指针，也就是 $L[1]$ 。而在 $L[1]$ 的析构函数中，又会释放 $L[2]$ ，以此类推。如图3.1中的左图所示，黄色节点代表析构中的节点，可以看出在最后一个节点被析构的时候，前面所有的节点都在等待。对于规模为 n 的列表 L ，会触

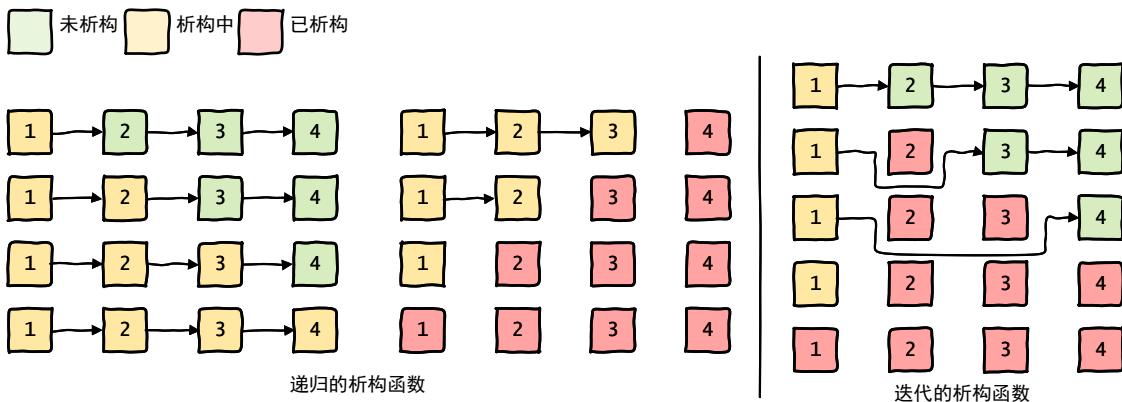


图 3.1 列表节点的递归析构和迭代析构

发 n 层析构函数的递归，当 n 非常大时，会引发栈溢出（stack overflow）错误。

因此，我们需要手动设计列表节点的析构函数，将析构函数的递归改为迭代，而不能依赖于 `std::unique_ptr` 的自动释放空间。下面展示了一个迭代版本的析构函数，如图3.1的右图所示，它不会造成大量节点在析构中状态等待的情况。

```

1 virtual ~ForwardListNode() {
2     auto p { std::move(m_next) };
3     while (p != nullptr) {
4         p = std::move(p->m_next);
5     }
6 }
```

3.2.2 双向列表节点

而当我们试图扩展单向列表到双向列表的时候，我们有两种选择。

1. 让前向指针和后向指针共享节点的所有权，也就是两个指针都声明为 `std::shared_ptr`。这种做法过于浪费空间，因为 `std::shared_ptr` 除了指针本身之外，还包括一个引用计数。
2. 仍然保留只有后向指针拥有节点的所有权，也就是后向指针仍然是 `std::unique_ptr`，而前向指针则使用不涉及所有权的裸指针。这种做法节约了空间，但会破坏前向指针和后向指针的对称性，提高算法设计难度。

在本书中，采用后向指针单独拥有所有权，而前向指针使用裸指针的设计，因为对于基础数据结构而言，增加两个引用计数的成本是难以接受的。另一方面，本书通过将指针隐藏在代理类型下，一定程度解决了对称性的破坏问题。代理类之间的隐式转换可以让代码仍然具有良好的对称性。

```

1 template <typename T>
2 class ListNode {
3     T m_data;
```

```

4     ListNodePos<T> m_prev {};
5     ListNodeInst<T> m_next {};
6 public:
7     ListNode() = default;
8     ListNode(const T& data) : m_data(data) {}
9     virtual ~ListNode() { /* ... */ }
10    T& data() { return m_data; }
11    ListNodePos<T>& prev() { return m_prev; }
12    ListNodeInst<T>& next() { return m_next; }
13 }

```

如果您对智能指针不感兴趣，也可以使用裸指针改写本书中的这些节点类，变成您比较熟悉的形式。不要忘记**delete**每一个创建的节点。

3.2.3 哨兵节点

在单向列表中，我们只能从前向后访问节点，所以，我们只需要知道指向第一个节点的指针，就可以从前向后依次访问所有节点。而在双向列表中，我们可以从两个方向访问节点，因此我们需要知道指向第一个节点和最后一个节点的指针，才可以从两个方向依次访问所有节点。

在邓书中，每个列表存在两个不存储实际数据的虚拟节点，称为**头哨兵节点**和**尾哨兵节点**，或者简称为**头节点**（head）和**尾节点**（tail），用来标志列表的开始和结束。即使列表是空列表，这两个节点也存在。哨兵节点的引入使得许多实现得到了简化（simplify）和统一化（unify），比如前插（在一个节点之前插入新元素）不需要对第一节点进行特殊判定。在一些教材中介绍的列表是没有哨兵节点的，转而使用指向第一个节点和最后一个节点的裸指针来对头尾进行定位，称为**头指针**和**尾指针**。这不会影响到列表的功能，但有些功能的实现会略显复杂。在阅读完本章之后，您可以自己尝试写一个没有哨兵节点的列表，比较一下哪些功能的实现有所区别。

```

1 template <typename T>
2 class AbstractList : public AbstractLinearList<T, ListNodePos<T
3     >> {
4 public:
5     virtual ListNodePos<T> head() = 0;
6     virtual ListNodePos<T> tail() = 0;
7     virtual ListNodePos<T> insertAsNext(ListNodePos<T> p, const
8         T& e) = 0;
9     virtual ListNodePos<T> insertAsPrev(ListNodePos<T> p, const
10        T& e) = 0;
11     T& get(ListNodePos<T> p) override {
12         return p->data();
13     }
14     void set(ListNodePos<T> p, const T& e) override {
15         p->data() = e;
16     }
17 }

```

```
13 }
14 };
```

根据头节点在第一个节点之前、尾节点在最后一个节点之后的特性，您可以自己补出此处没有展示的、继承线性表抽象类的 `first` 和 `last` 等方法。此外，我们将插入操作区分为了前插（插入为 p 的直接前驱）和后插（插入为 p 的直接后继）。因为后向指针和前向指针不是对称的，所以前插和后插也有一些区别。

单向列表的抽象类实现和双向列表基本一致。区别在于，单向列表不支持从后向前的访问；因此，单向列表不支持 `last`、`back` 和 `prev` 等方法，我们需要将线性表抽象类中定义的这些方法设置为 `private` 隐藏。

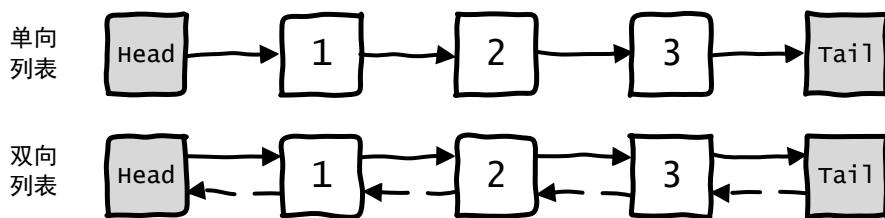


图 3.2 单向列表和双向列表

如图3.2所示，我们在单向列表和双向列表的两头各添加一个哨兵节点，代表所有权的智能指针用实线箭头表示，不涉及所有权的裸指针则用虚线箭头表示。由于哨兵节点的存在，用户总是可以知道列表的头和尾的位置。因此，即使在单向列表中，用户无法直接获得尾部（`back`）元素中的数据（需要从头节点向后依次遍历来寻找尾部），也无法直接删除尾部（`pop back`）的元素，但用户可以直接在尾部之后插入（`push back`）。如果没有尾节点的存在，则我们无法直接进行 `push back` 操作。在 C++ 的标准库中，`std::forward_list` 就是这样设计的：它包含一个最简单、最节约空间的抽象，没有尾节点也不支持 `push back`。

需要特别指出，哨兵节点仅仅用于列表的实现方法，而非列表的组成部分，因此在图3.2我们将它们用灰色表示。当我们具体讨论到某个列表 $L[0] \rightarrow L[1] \rightarrow \dots \rightarrow L[n - 1]$ 的时候，第一个节点 $L[0]$ 和最后一个节点 $L[n - 1]$ 都是实际存储元素的节点，而非头节点或尾节点。在列表中， $L[0]$ 没有前驱；而在列表的具体实现中，我们可以认为头节点是它的直接前驱以方便设计代码。

3.3 插入、查找和删除

3.3.1 后插一个元素

我们首先讨论后插。给定被插节点的位置 p 和待插入元素 e ，将元素 e 插入到列表中，使其对应的节点成为 p 的直接后继，这种类型的插入称为后插。对于双向

列表，假设 p 原有的直接后继是 q ，则二者的指针连接形成 $p \leftrightarrow q$ 的形式。我们希望将 e 插入到二者之间，形成 $p \leftrightarrow e \leftrightarrow q$ 的形式。这里可以看出引入哨兵的好处。在引入哨兵节点之后，对于任何一个被插节点，它总是有直接后继 q ；当被插节点是最后一个节点时， q 就是尾哨兵节点，而不需要进行特判。反过来，题目中如果遇到不含哨兵节点的情况（408 中可能是默认的），则需要特别地考虑在首尾边界的情况，必要时进行特判。

从宏观层面上来说，我们需要对 p 的后向指针、 q 的前向指针以及新生成的 e 的双向指针进行赋值，然而，这四个指针的赋值顺序需要仔细斟酌。这是设计列表算法的易错点，因为“拆散-重组”的过程不是一次性发生的，必定存在先后顺序。允许的顺序有许多种（所以不建议您背诵），但如果顺序错误，就有可能在拆散的过程中丢失了信息，导致无法进行重组。如果不熟练，那么当您设计这样的算法时，建议在纸上进行演算以确保“拆散-重组”过程是可行的。

比如说，如果我们直接断开 p 到 q 的后向指针，将 e 接入，形成 $p \rightarrow e | q$ 的形式，那么 q 会因为失去所有权而被智能指针自动回收（这种情况可以被生动地形容为“断链”）。因此，我们可以通过 `std::swap` 交换智能指针的所有权，然后再将 q 连接到 e 的后向指针处。建立好 $p \rightarrow e \rightarrow q$ 之后，再将前向指针补上。

```

1 ListNodePos<T> insertAsNext(ListNodePos<T> p, const T& e)
2     override {
3         auto node { ListNodeInst<T>::make(e) };
4         std::swap(p->next(), node);
5         p->next()->next() = std::move(node);
6         p->next()->prev() = p;
7         p->next()->next()->prev() = p->next();
8         ++m_size;
9     }

```

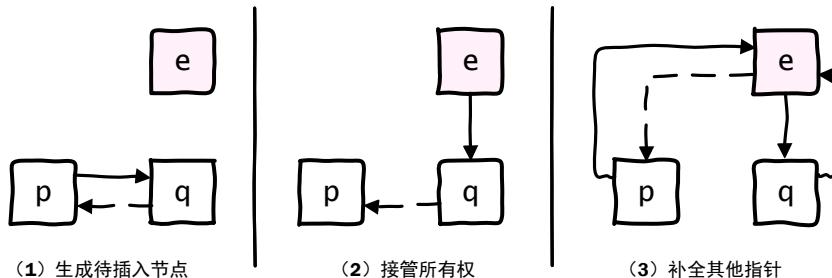


图 3.3 双向列表的后插

因为使用了交换语义，上面的这种做法产生了比预期更多的赋值次数。我们看到在上面的算法中，对于 `node` 的重复赋值是不到位工作，它经历了 e 变化为 q 、

再变化为空的过程，而实际上可以让它一步到位地变化为空。于是，我们不应该从 $p \rightarrow e | q$ 开始，可以从另一个后向指针，也就是 $p | e \rightarrow q$ 开始。这样就能减少一次赋值。

从图3.3中可以看出，在后插的过程中，应当首先让 e 的后向指针接管 q 的所有权；而其他三项的赋值次序是可以随意的。当然，尽管次序可以随意，但在书写的时候仍然需要当心。比如，在 e 的后向指针接管 q 的所有权之后，我们不再能通过 p 的后向指针来找到 q ，只能通过 e 的后向指针来找到它。这些细小的注意点有时熟练的程序员也会忽略，因此在考试的时候借助图形和手工演算进行检查是有意义的。

```

1 ListNodePos<T> insertAsNext(ListNodePos<T> p, const T& e)
2     override {
3         auto node { ListNodeInst<T>::make(e) };
4         node->next() = std::move(p->next());
5         node->next()->prev() = node;
6         node->prev() = p;
7         p->next() = std::move(node);
8         ++m_size;
9         return p->next();
}
```

单向列表的情况大同小异，只是减少了对前向指针赋值的操作。可以看出，在列表上做后插的时间复杂度和空间复杂度均为为 $O(1)$ 。

3.3.2 前插一个元素

对于双向节点，前插的过程和后插大同小异。因为头哨兵节点的存在，我们总是可以找到 p 的直接前驱 q ，于是我们只需要在 $q \leftrightarrow p$ 之间插入 e ，和前面的 $p \leftrightarrow q$ 只有字母上的区别。您可以自己实现它，并检验自己的实现是否正确。请注意前向指针是裸指针，后向指针是智能指针。当然，也可以直接用后插实现前插。

```

1 ListNodePos<T> insertAsPrev(ListNodePos<T> p, const T& e)
2     override {
3         auto q { p->prev() };
4         return insertAsNext(q, e);
}
```

对于单向列表，情况变得有些令人沮丧。因为单向列表中，我们无法使用前向指针，也就无法定位到 p 的直接前驱。那么，我们是否无法进行前插呢？是，也不是。我们可以用两种方法实现前插：

1. 从头节点开始，沿着后向指针访问整个列表，寻找 p 的直接前驱，然后再采用和上面一样的方式进行前插。显而易见，如果 p 在列表中的秩是 r ，则该方法的时间复杂度高达线性的 $\Theta(r)$ 。我们放弃循秩访问而使用列表，为的是

允许操作不连续内存，从而在插入和删除的时候达到 $O(1)$ 的常数复杂度；因此，这种方法虽然万无一失，但和我们的初衷背道而驰。

2. 使用后插代替前插。我们首先找到 p 的直接后继 q 进行一次后插，形成 $p \rightarrow e \rightarrow q$ 的形式，然后我们交换 p 和 e 节点的值，形成 $e \rightarrow p \rightarrow q$ ，如图3.4所示。这样在值上实现了前插，但是在位置上并没有实现前插。比如，如果原先外部有一个指针指向 p ，我们执行前插之后，这个指针并不知道 p 已经被 e “偷梁换柱”了，它可能还以为自己指向的是 p ，从而产生不可估计的错误。因此，这种方法事实上损失了安全性。但是，毕竟它是一个 $O(1)$ 的方法，我们也只能接受它，并期待用户正确地使用这个方法。

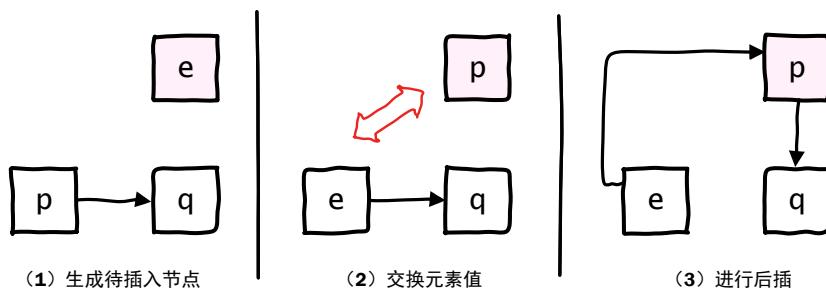


图 3.4 单向列表上使用后插代替前插

很显然，两种前插都存在一些问题，因此 C++ 标准库中的单向列表就不允许进行前插。如果我们采用从头节点开始找前驱的方法进行前插，不了解它复杂度的用户可能会错误使用这个方法，造成用户写出的程序非常缓慢。这个问题曾经让旧标准的 C++ 用户不胜其扰：在 C++11 之前，`std::list` 的获取规模方法 `size()` 是 $\Theta(n)$ 的，而用户很容易忘记这一点，从而造成用户应用程序时间效率的严重损失。如果我们使用后插代替前插，则会出现上述的指针指向混乱风险。在本书中为了让您了解单向列表实现前插的技巧，所以介绍了这两种方法，并在下面实现用后插代替前插的算法。在现实中设计程序时，应当避免这种高风险的设计。

回到使用后插代替前插的技术上。和上一节一样，我们注意到采用交换语义会引入不到位的赋值。在消除交换语义之后，一个可能的实现如下。

```

1 ForwardListNodePos<T> insertAsPrev(ForwardListNodePos<T> p,
2   const T& e) override {
3   auto node { ForwardListNodeInst<T>::make(std::move(p->data
4     ())) };
5   node->next() = std::move(p->next());
6   p->next() = std::move(node);
7   p->data() = e;
8   if (m_tail == p) {
9     m_tail = p->next();
10 }
11 }
```

```

9     ++m_size;
10    return p;
11 }

```

前面讨论过，如果原先外部有一个指针指向 p ，我们执行前插之后，这个指针并不知道 p 已经被替换了。我们无法保证用户使用的“外部指针”的安全性，但我们需要保证数据结构内部的指针的安全性。这个可能丧失安全性的指针就是尾节点指针 m_tail 。当我们要在尾节点处进行前插（也就是 push back）的时候，上述方法事实上进行了后插，并将原先的尾节点赋值为 e 。在这种情况下，我们需要将尾节点指针指向新的尾节点处。

另外，请注意这里必须对 p 中的数据使用移动语义，而不能使用复制语义，否则当 p 中的数据非常大（比如，一个被嵌套在列表里的向量）时，会引入大量的（非常数的）性能损失。

3.3.3 查找一个元素

无论是单向列表还是双向列表，查找元素的过程和无序向量都是一样的，只需要从前向后一个一个查找即可。因为不支持循秩访问的缘故，即使是有序列表，也不支持折半查找。您很容易自己实现顺序查找，下面是一个示例的实现。

```

1 ListNodePos<T> find(const T& e) const override {
2     ListNodePos<T> p { m_head->next() };
3     while (p != m_tail) {
4         if (p->data() == e) {
5             return p;
6         }
7         p = p->next();
8     }
9     return m_tail;
10 }

```

3.3.4 删除一个元素

列表删除是插入的逆操作。对于双向列表，假设被删除的节点为 p ，由于头节点和尾节点的存在，我们可以保证它存在直接前驱 q_1 和直接后继 q_2 。因此，我们需要将 $q_1 \leftrightarrow p \leftrightarrow q_2$ 变形为 $q_1 \leftrightarrow q_2$ 。显然，我们只需要对 q_1 的后向指针和 q_2 的前向指针各进行一次赋值。顺序仍然是重要的，但难度比插入降低了不少（因为 2 次赋值的顺序一共只有 2 种）。下面是一个可行的实现。

```

1 T remove(ListNodePos<T> p) override {
2     auto e { std::move(p->data()) };
3     p->next()->prev() = p->prev();
4     p->prev()->next() = std::move(p->next());
5     --m_size;

```

```

6 |     return e;
7 |
}

```

类似地，上面的做法也不适用于单向列表的删除。对于单向列表，我们需要采用和前插相似的技术，首先找到 p 的直接后继 q_2 ，以及 q_2 的后继 q_3 。我们在 $p \rightarrow q_2 \rightarrow q_3$ 上交换 p 和 q_2 节点上的值，从而在值的角度，变成了 $q_2 \rightarrow p \rightarrow q_3$ ，然后重新复制 q_2 的后向指针，变为 $q_2 \rightarrow q_3$ 即可。双向列表和单向列表在删除节点上的区别如图3.5所示，请注意在双向列表の場合，当移交 q_2 的所有权之后，节点 p 会被智能指针自动释放（用灰色表示）。

图中 q_3 是虚线节点，这是因为即使在存在尾节点的列表中， q_3 仍然可能为空 (`nullptr`)。如果 q_3 为空，那么 q_2 就是尾节点（在有哨兵的情况下），交换元素之后我们会把原先的尾节点删除。所以，我们需要进行一次特判，在上述情况下让新的 q_2 成为新的尾节点。在消除了交换语义带来的不到位赋值之后，得到的一种示例程序如下所示。

```

1 T remove(ForwardListNodePos<T> p) override {
2     auto e { std::move(p->data()) };
3     p->data() = std::move(p->next()->data());
4     p->next() = std::move(p->next()->next());
5     if (p->next() == nullptr) {
6         m_tail = p;
7     }
8     --m_size;
9     return e;
10 }

```

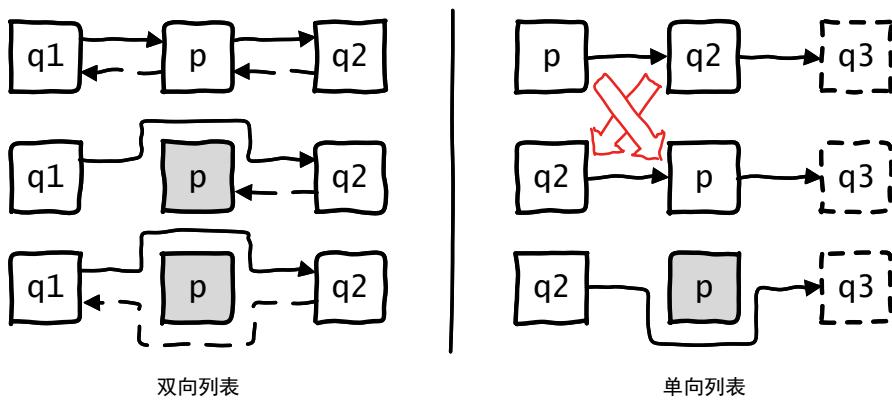


图 3.5 在列表中删除一个元素

和3.3.2节一样，这里的尾节点指针 `m_tail` 也是需要维护的“外部的指向节点的指针”。我们需要对这个指针重新赋值以避免安全性问题。在 C++ 标准库的单向列表中，为了安全性考虑，只提供“删除后继”的方法 `erase_after` 而不提供直

接删除的方法。

从上面对插入、查找和删除的分析中可以看出，列表的三种基本操作完全不涉及规模。所以，如果不需要在 $O(1)$ 时间里获取规模，那么列表是可以不记录这个属性的。不记录规模可以使列表在插入和删除时减少一次赋值，从而略微提高列表的性能。如果您自己实现了列表和单向列表，可以通过 *ListTest.cpp* 和 *ForwardListTest.cpp* 进行测试。

3.3.5 实验：在头部和尾部连续插入元素

随着三种基本操作分析完毕，我们可以开始对列表展开实验。这一节讨论在列表的头部和尾部连续插入元素的情况，代码可以在 *ListInsert.cpp* 中找到。

在示例代码中，采用示例实现的向量、双向列表和单向列表类作为实验对象。您可以加入自己实现的版本，和它们的性能进行比较。我们考虑两种环境。第一种是连续在头部插入元素，也就是每次插入的元素会成为线性表的第一个元素；另一种是连续在尾部插入元素，也就是每次插入的元素会成为线性表的最后一个元素。

这个实验表明，当在尾部连续插入元素时，向量的效率显著高于列表，但是单向列表并没有和双向列表之间有很大的差异。根据您的运行环境不同，向量和列表大致呈现一个数倍的性能差距，总体来说是常数的。需要特别指出的是，尽管在尾部连续插入元素的测试中向量的性能高于列表，但对于单次插入元素的过程则不一定。触发扩容时，向量单次插入元素的时间复杂度可达 $\Theta(n)$ ，而列表是稳定的 $O(1)$ 。这一点是列表的独特优势，在对响应时间稳定性有要求的场合下，向量的扩容变得不能接受，而列表的插入时间是可以保证很低的。

当在头部插入元素时，可以显著地看到，向量 $\Theta(n^2)$ 的时间复杂度远高于列表 $\Theta(n)$ 的时间复杂度，当 $n = 10^6$ 的时候向量的情况已经需要耗费几十秒（在笔者的计算机上）；我们可以直接做一个判断，在这种情况下抛出一个 `error` 拒绝执行，如下所示。

```

1 void operator()(size_t n) override {
2     if constexpr (is_base_of_v<AbstractVector<int>, Linear>) {
3         if (n > 100000) {
4             throw runtime_error { R"(Vector::push_front() is too
5                 slow for n > 100'000)" };
6         }
7         for (size_t i { 0 }; i < n; ++i) {
8             m_container.push_front(i);
9         }
10    }

```

因为向量的耗时增长很快，我们在 $n = 10^5$ 量级已经可以看出它的性能，因此不需要花费大量的时间等待它在更大的规模上完成实验。这里采用了 C++17 引入的 `if constexpr` 语法，从而实现在编译期判断。上面的代码对于以向量为参数的模板实例，会将 `if constexpr` 内的判断编译进去；而对于以列表为参数的模板实例，则不会将 `if constexpr` 内的判断编译进去；这个向量和列表的区别是编译期完成的，运行期不会再做判断。这一语法简洁清晰，可以用来替代 C 语言风格很容易被滥用的 `#ifdef` 等条件编译命令。

表 3.1 不同数据结构连续插入元素的耗时对比（单位：ms）

插入位置	n	向量	列表	单向列表
尾部	10^4	0	2	2
	10^5	1	22	21
	10^6	9	258	234
头部	10^4	2	2	2
	10^5	254	22	20
	10^6	39167	222	198

在这个实验中，您还能发现的一点是，双向列表和单向列表在时间上的性能差异实际上非常小。双向列表虽然节点多了一个字段，但消耗的时间和单向列表相比，只有非常有限的增加，与此同时，它获得了从后向前访问的能力、更强的灵活性以及更好的安全性（关于安全性，在前插和删除的时候已经讨论过）。因此，除非性能要求苛刻或空间不足，否则都建议使用双向列表。

3.3.6 实验：顺序删除和随机删除 *

当我们连续地在列表中插入节点时，这些节点也会存在一定程度上的空间连续性，因为它们的空间是操作系统连续分配的。这会导致我们的实验并不能真实地反映出现实情况下的列表所包含的节点地址弱局部性这一特征。在这个实验中我们将会认识到，如果列表所包含的节点地址局部性很差，会对列表的性能造成很大的负面影响。

我们考虑下面的连续删除场景。代码可以在 *ListRemove.cpp* 中找到。

```

1 template <typename T>
2 class ContinuousPop : public Algorithm<void()> {
3     protected:
4         List<T> L;
5         Vector<ListNodePos<T>> V;
6     public:
```

```

7   virtual void initialize(size_t n) = 0;
8   void operator()() override {
9     for (auto p : V) {
10      L.remove(p);
11    }
12  }
13 };

```

在这个框架中，我们使用向量 V 来存储列表 L 的每一个节点的位置。通过初始化函数，我们让向量 V 成为顺序的或者随机的，然后，按照向量 V 中存储的节点次序依次删除列表中的元素。对于顺序存储的情况，每次向 L 加入元素后储存在 V 中即可，如下所示。对于随机存储的情况，可以在顺序存储之后增加一个 `std::shuffle`（我们在向量置乱的时候介绍过它）。

```

1 // SequentialPop
2 void initialize(size_t n) override {
3   L.clear();
4   V.clear();
5   for (size_t i { 0 }; i < n; ++i) {
6     V.push_back(L.insertAsNext(L.last(), i));
7   }
8 }

```

从这个例子中可以看到，当 n 比较小的时候，顺序删除和随机删除的性能基本一致；但当 n 比较大的时候，二者就会产生一个明显的差异。这个性能差异就是随机删除的局部性丧失引起的。

表 3.2 顺序删除和随机删除的耗时对比（单位：ms）

n	顺序删除	随机删除
10^5	21	25
10^6	190	285
10^7	2154	4649

在现实中，我们生成的列表可能不是以连续插入的形式分配内存的。因此，它可能天然就具有一个很差的局部性。这就意味着，现实中的列表在较坏的情况下有可能会更接近于本实验中随机访问的性能。结合上一个实验，我们意识到，列表和向量之间有着巨大的性能差距，甚至在数据结构规模较小的情况下，列表 $O(1)$ 插入、删除的优势体现不出来，从时间消耗的角度上来说还不如 $\Theta(n)$ 的向量操作。因此对于小型线性表，无论我们是否需要循秩访问、折半查找等向量特性，都应该优先使用向量而不是列表。

3.3.7 实验：倒置列表

当在向量上设计算法的时候，由于向量循秩访问的特性，我们可以对秩进行运算，从而快速、精准地找到某个元素。一个典型的例子就是折半查找。循秩访问使得向量中的元素地位比较“平等”；而在列表的场合，循位置访问使得列表中的元素地位比较“不平等”，比较“任人唯亲”。比如，从头节点出发访问列表中的一个元素时，所需要消耗的时间会和被访问元素的位置相关；越接近头部的元素，访问需要的时间越短。

我们可以站在更高的层次，理解列表只能直接访问首尾两端、以及循位置访问“任人唯亲”的特性：列表是**线性递归定义**的。具体来说，列表可以被这样定义：

1. \emptyset 是列表（空列表，没有元素）。
2. 设 x 是一个节点， L 是一个列表，则 $x \rightarrow L$ 是列表（ x 的后向指针指向 L 的第一个节点）。

这个定义具有和《绪论》章所述递降法相似的形式，它也就意味着列表适合使用减治思想设计算法。首先，我们研究递归边界（空列表）的情况；其次，对于规模为 n 的列表，我们将第一个节点提取出来，递归地研究后 $n - 1$ 个节点组成的子列表，再和第一个节点进行合并。对称地，我们可以想到列表具有另一个定义：

1. \emptyset 是列表（空列表，没有元素）。
2. 设 x 是一个节点， L 是一个列表，则 $L \rightarrow x$ 是列表（ L 的最后一个节点的后向指针指向 x ）。

这一定义同样可以引出一种减治算法的设计。它和前一种定义的区别在于，平凡项 x 在减治项 L 的头部还是尾部。对于双向列表来说，头部减治和尾部减治是等价的，提取出平凡项 x 都只需要 $O(1)$ 的时间。当然，我们需要注意，减治递归算法应用在列表上时，递归深度高达 $\Theta(n)$ ，这不但导致了较高的空间复杂度，同时在 n 较大时很可能引发栈溢出错误；因此在设计完成减治算法之后，应当尽可能将其修改为迭代算法。

单向列表的头部和尾部不具有对称性，因此情况会有所不同。考虑单向列表 $L[0] \rightarrow L[1] \rightarrow L[2] \rightarrow \dots \rightarrow L[n - 1]$ 。当在头部进行减治时，只需要 $O(1)$ 的时间就可以找到 $L[0]$ ；而在尾部进行减治时，需要 $\Theta(n)$ 的时间才能找到 $L[n - 1]$ 。看起来单向列表应当总是在头部进行减治，然而事实却不一定如此。本节将以倒置列表为例子，对两种减治方法进行比较。代码可以在 *ListReverse.cpp* 中找到。为了便于和双向列表上的算法进行对比，我们总是传入双向列表 `List<T>&` 作为参数类型，并在设计单向列表算法的时候不使用涉及前向指针的方法。

对于双向列表的倒置，我们可以直接从头尾两个方向遍历，交换对称的两个

节点的数据域。这种做法和向量上的做法一致，也可以使用`std::reverse`。

```

1 // ReverseBasic
2 void operator()(List<T>& L) override {
3     auto head { L.head() };
4     auto tail { L.tail() };
5     for (auto size { L.size() }; size >= 2; size -= 2) {
6         head = head->next();
7         tail = tail->prev();
8         std::swap(head->data(), tail->data());
9     }
10 }
```

下面讨论单向列表，首先考虑在头部进行减治的情况。按照减治的思想，在将 $L[0]$ 摘出列表之后，把从 $L[1]$ 开始的剩余列表反转，然后再进行合并：把被摘出的 $L[0]$ 接到反转后的剩余列表的尾部，如图3.6中的左图所示。

```

1 // ReverseReduceAtHead
2 void operator()(List<T>& L) override {
3     if (L.size() < 2) {
4         return;
5     }
6     auto first { L.pop_front() };
7     (*this)(L);
8     L.push_back(first);
9 }
```

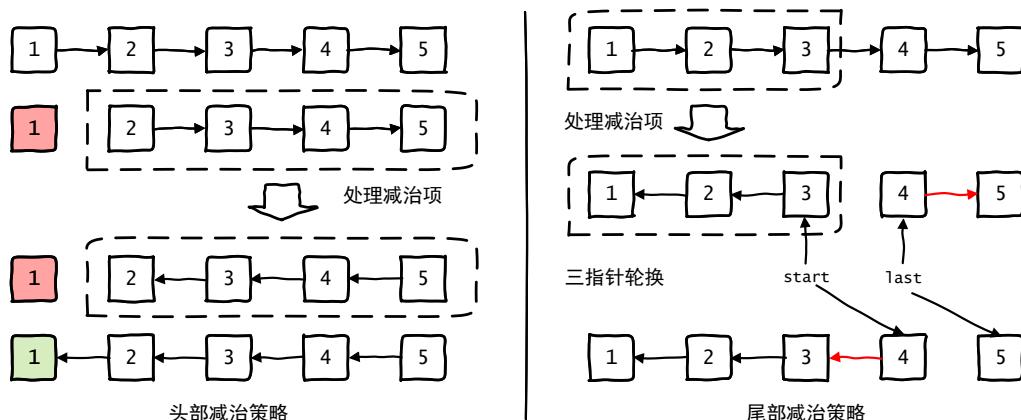


图 3.6 利用头部减治和尾部减治倒置列表

它的时间复杂度和空间复杂度均为 $\Theta(n)$ 。这个算法并不是尾递归，这意味着我们将它改写成迭代的时候会遇到困难。这一困难来自于我们在做递归调用的时候，需要保存当前递归实例摘出的元素`first`。当我们进行到最后一层递归调用的时候，需要保存之前每一次摘出的元素，这就意味着不可避免地需要 $\Theta(n)$ 的空间。这也就意味着，将此方法改写为迭代之后，几乎和下面的“极端”做法等价。

```

1 // ReverseMove
2 void operator()(List<T>& L) override {
```

```

3     Vector V(L.size());
4     move(begin(L), end(L), begin(V));
5     reverse(begin(V), end(V));
6     move(begin(V), end(V), begin(L));
7 }
```

这个做法的时间复杂度和空间复杂度同样都是 $\Theta(n)$ 。它将所有元素移动到一个辅助向量里，在向量里面倒置，再移动回列表里。

无论是命题者还是我们自己，都很难接受这样的算法。从刚才的分析中已经可以发现，影响空间效率的主要原因是，单向列表只有后向指针。当在头部进行减治的时候，被提取出来的 $L[0]$ 彻底和 $L[1:n]$ 断开，无法通过 $L[1:n]$ 上的指针找到 $L[0]$ ，从而必须要额外的空间来存储它。于是就可以想到，是否可以利用单向列表的不对称性，采用在尾部进行减治的策略设计算法？答案是肯定的。当在尾部进行减治的时候，我们可以先处理 $L[0:n - 1]$ ，然后利用 $L[n - 1]$ 的后向指针直接找到 $L[n]$ 。下面展示了一个示例代码。

```

1 template <typename T>
2 class ReverseReduceAtTail : public ReverseForwardList<T> {
3     ListNodeInst<T> reverse(ListNodeInst<T>& start, size_t size
4         ) {
5         if (size == 1) {
6             return move(start->next());
7         }
8         auto last { reverse(start, size - 1) };
9         auto tmp { move(last->next()) };
10        last->next() = move(start);
11        start = move(last);
12        return tmp;
13    }
14 public:
15     void operator() (List<T>& L) override {
16         auto old_first { L.first() };
17         auto start { move(L.head()->next()) };
18         auto last { reverse(start, L.size()) };
19         L.head()->next() = move(start);
20         old_first->next() = move(last);
21     }
22 }
```

如图3.6中右图所示，上述算法中将减治项和平凡项合并的过程，就是在 `start`、`last` 和红色指针之间进行了一个轮换。这份代码很容易被改写为迭代，因为它在进行递归调用处理减治项的时候，并不需要存储合并的时候需要用到的变量。我们可以使用固定的变量 `start` 和 `size`，作为每次递归调用的参数，并使用固定的变量 `last` 存储每次递归调用的返回值。一个示例的迭代写法如下所示。

```
1 // ReverseReduceAtTailIterative
```

```

2 void operator()(List<T>& L) override {
3     auto old_first { L.first() };
4     auto start { move(L.head()->next()) };
5     auto last { move(start->next()) };
6     for (auto size { L.size() }; size > 1; --size) {
7         auto tmp { move(last->next()) };
8         last->next() = move(start);
9         start = move(last);
10        last = move(tmp);
11    }
12    L.head()->next() = move(start);
13    old_first->next() = move(last);
14 }
```

上述算法的时间复杂度为 $\Theta(n)$, 空间复杂度为 $O(1)$ 。此外, 这是一个针对指针域进行操作的算法, 在整个算法过程中, 只有各个后向指针的指向发生了变化, 而每个节点内部的数据域没有变化, 也没有创建或删除节点。上面介绍在头部进行减治的算法时涉及到了创建和删除节点, 您可以自己尝试对示例代码进行修改, 让它也成为一个针对指针域进行操作的算法。

表 3.3 倒置列表算法的耗时对比 (单位: ms)

n	数据域 Basic	数据域 转移到向量	指针域 头部减治 递归	指针域 尾部减治 递归	指针域 尾部减治 迭代
10^3	0	0	0	0	0
10^4	0	0	栈溢出	栈溢出	1
10^5	2	12	栈溢出	栈溢出	16
10^6	25	134	栈溢出	栈溢出	190
10^7	248	1142	栈溢出	栈溢出	1617

另一方面, 之前介绍的 ReverseBasic 算法, 逐个交换双向列表中对称的节点上的数据, 是针对数据域进行操作的算法。在整个算法过程中, 只有各个节点内部的数据域发生了变化, 而没有改变每个节点的指针指向关系, 也没有创建或删除节点。

针对指针域和**针对数据域**, 是设计列表算法时的两种不同路径。针对指针域的算法将列表上的节点当做一个整体, 通过修改指针来改变节点在列表中的顺序。针对数据域的算法则将列表当做一个访问受限 (只能访问已知位置的直接前驱和直接后继) 的向量处理。如果一个向量上的算法没有涉及循秩访问, 那就可以原封不动地迁移到列表上。涉及循秩访问的算法如折半查找则不适用。这种做法显得非常投机取巧, 但效率却不一定低, 在现实程序开发中不失为一种选择。为了避免

考生投机取巧，试卷经常会强制要求只能针对指针域设计算法。请您根据自己对双向列表的理解，将上面的尾部减治算法改为双向列表的版本。

在实际解题时给出的列表，有可能无法直接通过 `size` 方法获取规模，此时有两种方法可以解决。其一是直接遍历一遍整个列表做个统计，因为不会影响时间复杂度和空间复杂度，这种做法通常是可行的。其二是改变迭代的终止条件（递归边界）为不涉及规模的形式，在不同的算法中这一修改会有所不同。您可以尝试对 `ReverseBasic` 以及尾部减治算法进行修改，使其不涉及列表的规模。

3.4 列表的归并排序

3.4.1 基于值的归并排序

本节讨论在列表上做归并排序，这是一个比较复杂的列表操作。一种基本的思路是基于值的归并排序，即之前在 2.6.3 和 2.6.4 节使用的归并排序。因为从迭代器的观点看，向量和列表作为线性表具有高度的相似性，所以无论是双向列表还是单向列表，都可以使用相同的代码进行归并排序。唯一的区别在于向量的迭代器支持随机访问，而列表无法支持这一点。因此，在我们想要找到中点的时候，向量的版本可以直接使用左边界和右边界上的平均值，而列表的版本只能传入一个规模并手动向前依次迭代；幸运的是，迭代的时间是 $\Theta(n)$ 的，和归并的时间相同，因此不会引入更高的时间复杂度。

3.4.2 基于指针的归并排序 *

显然，上面这种归并排序没能发挥列表的特点。在归并的时候，没有必要使用快慢指针，而是可以充分利用列表的灵活性，达到不需要辅助空间的效果。比如，原先的列表 L 可以分成 L_1 和 L_2 两部分，现在为了对列表 $L_1 \rightarrow L_2$ ，我们可以首先将它切斷成两个独立的列表 L_1 和 L_2 。随后，令 $L = \emptyset$ 清空，依次比较 L_1 和 L_2 的首节点，将较小的节点移动到 L 的末尾。全部移动结束后，就完成了对 L 的归并。这是一个相当有挑战性的工作，几乎达到了笔试中手写代码的最大可能难度，需要对列表有深刻的理解才能写出来。非常建议您进行这项尝试。这里的示例程序仍然有进一步优化的空间。

上一小节讨论的基于值的归并排序，在整个排序过程中没有修改任何节点的相对位置，只是改动了每个节点里的数据元素的值。因此这个版本的实现和向量相同，因为向量也不能改变数据单元的相对位置，向量中相对位置和绝对位置（地址）是统一的。而这里的归并排序，则在整个排序过程中，每个节点的值都没有被修改，被改动的只有节点的前向和后向指针，从而改变节点的相对位置。这是在邓

书中介绍的版本，它的原理仍然是归并排序，但归并的实现和向量的版本完全不同。

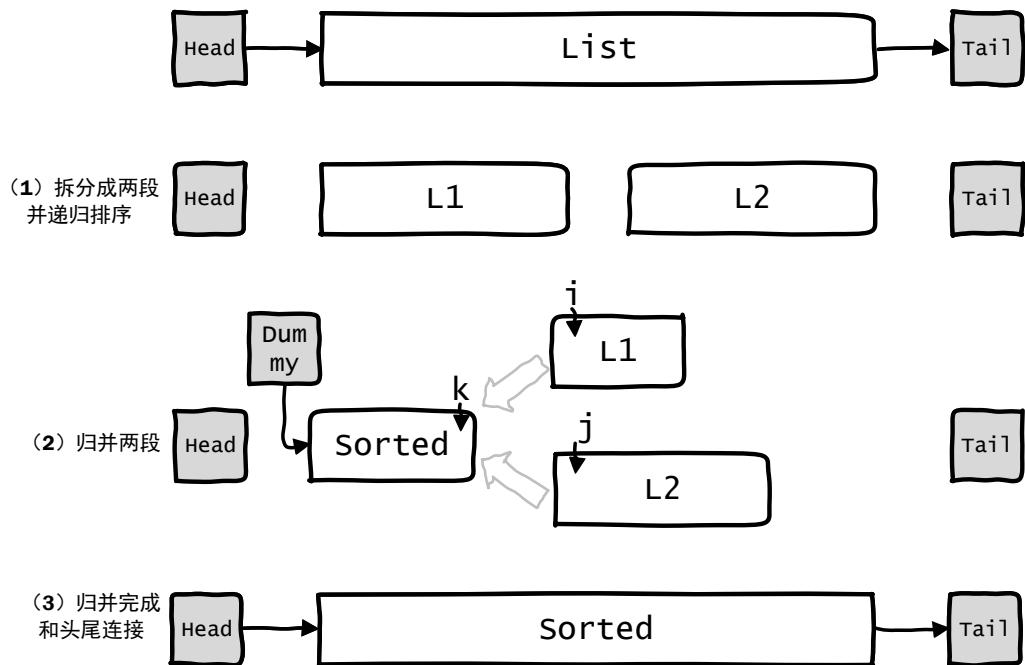


图 3.7 基于指针的归并排序

哨兵节点在这个版本的归并排序里会产生麻烦，因为它们会破坏拆分后的列表结构。如图3.7所示，如果在拆分列表前不分离哨兵节点，则拆分后的前后两段，一个只有头节点，一个只有尾节点，需要各自单独判断；如果先分离哨兵节点再拆分，则两段都既没有头节点也没有尾节点，不需要单独判断。当数据区域被排序完成之后，再将头尾节点接入即可。在归并的时候，示例代码里使用了一个哑节点（dummy）来起到临时头节点的功能，这对于不带头节点的列表算法设计是常用的技术。在归并的过程中，哑节点不是列表整体的头节点，而是正在被归并的这一段的临时头节点，归并结束之后，排好序的这一段会和哑节点分离。您也可以直接使用被分离下来的头节点担任这一职能（示例代码中为了展示哑节点的作用没有这么做）。

```

1 template <typename T, template<typename> typename List>
2     requires std::is_base_of_v<AbstractList<T>, List<T>>
3 class ListMergeSort : public AbstractSort<T, List> {
4     using Pos = ListNodePos<T>;
5     using Inst = ListNodeInst<T>;
6     Inst dummy { Inst::make() };
7     Inst& forward(Pos from, size_t step) {
8         for (size_t i { 1 }; i < step; ++i) {
9             from = from->next();
10        }

```

```

11     return from->next();
12 }
13 void connect(Pos& prev, Inst& next) {
14     next->prev() = prev;
15     prev->next() = std::move(next);
16     prev = prev->next();
17     next = std::move(prev->next());
18 }
19 void merge(Inst& L1, Inst& L2) {
20     Pos L { dummy };
21     while (L1 && L2) {
22         if (this->cmp(L1->data(), L2->data())) {
23             connect(L, L1);
24         } else {
25             connect(L, L2);
26         }
27     }
28     if (L1) {
29         L->next() = std::move(L1);
30     } else {
31         L->next() = std::move(L2);
32     }
33     L1 = std::move(dummy->next());
34 }
35 void mergeSort(Inst& head, size_t size) {
36     if (size < 2) return;
37     auto mi { std::move(forward(head, size / 2)) };
38     mergeSort(head, size / 2);
39     mergeSort(mi, size - size / 2);
40     merge(head, mi);
41 }
42 protected:
43     void sort(List<T>& L) override {
44         if (L.empty()) {
45             return;
46         }
47         auto first { std::move(L.head()->next()) };
48         auto tail { std::move(L.tail()->prev()->next()) };
49         mergeSort(first, L.size());
50         forward(first, L.size() - 1)->next() = std::move(tail);
51         L.head()->next() = std::move(first);
52     }
53 };

```

上面的这个做法因为只需要调用`forward`来从前向后地移动指针，所以也同样支持单向列表。将所有对前向指针的赋值删去即可。

3.4.3 实验：归并排序的性能差异 *

基于指针的归并排序，将空间复杂度从 $\Theta(n)$ 降低到了 $\Theta(\log n)$ （一定要注意递归本身的空间复杂度，不能误以为是 $O(1)$ ），那么显然地，它会在时间上有所损失。为

了评估归并排序的性能差异，我们设计了一个实验。代码可以在 *ListMergeSort.cpp* 中找到。

表 3.4 归并排序的耗时对比（单位：ms）

n	向量 自上而下	向量 自下而上	顺序 列表 自上而下 基于值	顺序 列表 自下而上 基于值	顺序 列表 自上而下 基于指针	随机化 列表 自上而下 基于值	随机化 列表 自下而上 基于值	随机化 列表 自上而下 基于指针
10^4	11	12	14	15	16	16	20	16
10^5	143	148	203	254	242	287	448	289
10^6	1749	1759	2415	2917	3252	4105	5299	3585

我们对比三个归并排序，分别为用基于值的方法实现的归并排序（自上而下和自下而上版本），以及基于指针的归并排序。实验结果表明，和向量的情况不同，在列表的场合，自下而上的归并排序消耗的时间会显著高于自上而下的归并排序。这是因为，自上而下版本的一些归并段的左右端点来自递归调用的参数，只有中点需要自己找；自下而上版本因为没有递归，所以在已知左端点的情况下，需要自己找中点和右端点。在循秩访问的向量中，这只是多了一次加法，时间消耗可以忽略不计；而在循位置访问的列表中，找端点需要循着后向指针一个一个找，时间消耗就明显增加了。自上而下的版本用时明显比向量归并排序多，一定程度上也来自于这个原因。

另一个影响归并排序效率的重要因素来自于局部性。实验中提供了一个 `shuffle` 方法，用于将列表中的节点随机打乱。您可以观察调用了该方法和不调用的区别。在不调用 `shuffle` 的情况下，自下而上的版本就会和自上而下版本有一定的差距，而基于指针的版本在 n 较大的情况下会更慢。这是因为不随机打乱的时候，列表中的节点是连续生成的，因此基于值的排序能利用到局部性，而基于指针的排序在排序过程中会将节点次序打乱从而丧失局部性。

但在调用了 `shuffle` 的情况下，情况会发生变化。所有版本的列表归并排序都会因为损失局部性而性能降低。由于循着后向指针一个一个找的过程没有了局部性，自下而上版本和自上而下版本的差距会进一步拉大。基于指针的版本因为移动指针本来就会破坏局部性，受到的影响比较小。

3.5 循环列表

循环列表 (circular list) 基于循环链表，后者是普通的、线性的链表的一个变体。简单地说，循环列表就是舍弃了头节点和尾节点，让第一个节点 $L[0]$ 的前向

指针指向最后一个节点 $L[-1]$ ，让最后一个节点 $L[-1]$ 的后向指针指向第一个节点 $L[0]$ ，首尾相连，形成的环状结构。循环列表同样可以分为单向循环列表和双向循环列表。循环列表的三种基本操作和本章中介绍的普通线性列表几乎完全一致，所以不再赘述。邓书中也删除了这部分内容。

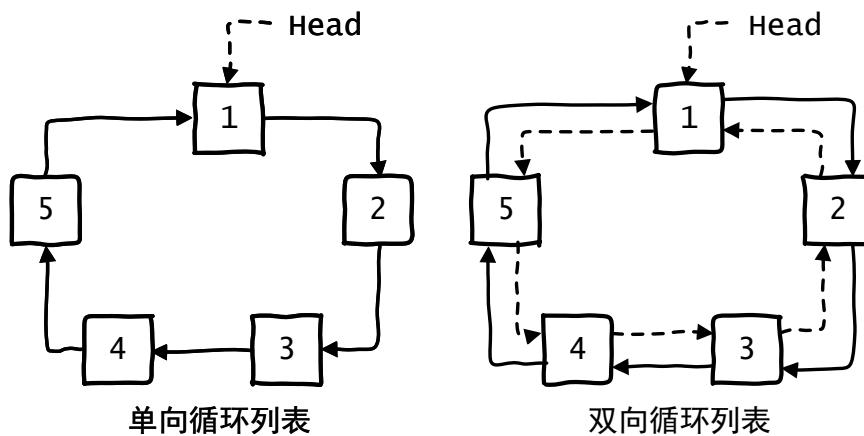


图 3.8 循环列表

图3.8展示了单向循环列表和双向循环列表的结构。由于需要保证列表的节点首尾相连，所以无法在循环列表的环上嵌入哨兵节点。图中的`head`称为**头指针**，它是一个指向列表中第一个元素的裸指针；类似地，也可以（可选地）用一个**尾指针**指向列表中的最后一个元素。

注意到，如果循环列表中只有一个元素，那么它的后向指针（双向循环列表中，也包括前向指针）将指向自己，因此它持有自身的所有权。您可以通过画图演算发现，使用本章中介绍的普通线性列表的删除方法，删除循环列表中唯一的一个元素时，会无法释放这个唯一元素的空间。因此，对最后一个元素进行删除的时候需要进行特判。如果您实现了循环列表，可以通过 *CircularListTest.cpp* 进行测试。

循环列表的主要特征是：循环列表上的指针沿着前向指针或后向指针走下去，就可以在列表上无限轮询每个节点。这种无限轮询的特点使它在《网络原理》和《操作系统》这两门学科“持续性维护”的情景中有用武之地，但在《数据结构》中不常用。在 *CircularListIterate.cpp* 中您可以看到无限轮询的例子。

3.6 静态列表

3.6.1 静态列表的结构

静态列表 (static list) 指的是基于数组实现的列表（或者也可以基于向量实现，从而允许扩容）。列表在建立的时候，申请一块连续的内存，所有的节点都存在这

部分内存中。与向量有所区别的是，节点的相对位置可以和它们在数组里的位置不同。在静态列表中，前向指针和后向指针往往用所基于的数组（向量）中的下标代替，因此，静态列表虽然本质上是一种循位置访问的结构（列表），但它的位置是用秩表示的，因为列表中的节点存在数组或向量里。

和动态分配内存的动态列表相比，由于装填因子的问题，静态列表需要消耗更多的空间，所以它非常不常用（仅出现在不提供指针/引用语法的古代编程语言中），邓书删除了这部分内容，也几乎不会出现在考试中。但在机试里，由于动态分配内存可能会引起时间消耗上的不确定性，并且机试题目给定的空间几乎总是绰绰有余的，所以静态列表仍然有发挥的空间。

单链/双链、线性/循环、动态/静态，这三对关系是相互独立的，可以组成 8 种不同结构的列表实现。在本节将以双链、线性为例，介绍静态列表的实现方法。为了支持扩容，这里基于向量实现。由于静态列表和动态列表非常相似，这里只介绍静态列表的特有特点。

3.6.2 静态列表上的节点

和动态列表不同，静态列表用来存储前向指针和后向指针的类型变成了秩。

```

1 template <typename T>
2 class StaticListNode {
3     T m_data;
4     Rank m_prev { 0 };
5     Rank m_next { 0 };
6 };

```

对于秩而言，我们需要指定特殊值用来对应指针情形中的空指针。在这里，我们认为秩为 0 表示指向空；因此在实现静态列表的时候，我们不能使用秩为 0 的数据单元来存放数据，否则会产生混淆。当然，也可以规定一个不可能出现的值（比如-1）。除了秩为 0 的单元外，我们还需要规定固定的单元用来存放头节点和尾节点。

```

1 template <typename T>
2 class StaticList : public AbstractStaticList<T> {
3     Vector<StaticListNode<T>> m_data {};
4     const static Rank m_head { 1 };
5     const static Rank m_tail { 2 };
6     StaticListNode<T>& getNode(Rank r) override {
7         return m_data[r];
8     }
9 };

```

这里，我们采用了一个`getNode`方法在秩和位置之间建立联系。静态列表的操作和动态列表相比大同小异，因为不需要考虑智能指针的所有权问题，静态列表

可以更加随意一些，比如，下面是静态列表的后插实现。

```

1 Rank insertAsNext(Rank p, const T& e) override {
2     Rank r { m_data.size() };
3     m_data.push_back(StaticListNode<T>(e, p, getNode(p).next()));
4     getNode(p).next() = r;
5     getNode(getNode(r).next()).prev() = r;
6     return r;
7 }
```

3.6.3 在静态列表上删除一个元素 *

如果我们按照动态列表的方法在静态列表上删除，可以得到下面的算法。

```

1 T remove(Rank r) override {
2     T e { std::move(getNode(r).data()) };
3     getNode(getNode(r).prev()).next() = getNode(r).next();
4     getNode(getNode(r).next()).prev() = getNode(r).prev();
5     return e;
6 }
```

上述算法存在一个致命的问题：被删除的节点的内存无法被释放。因为被删除的节点在向量 V 中，所以不能直接释放内存。所以，需要将被删除的节点从向量 V 中删除，以释放 V 中的空间，使其可以被分配给其他节点。然而，直接调用向量的删除元素方法也是不行的。因为删除 $V[0:n]$ 中的一个节点 $V[r]$ ，会导致它的后缀 $V[r+1:n]$ 整体前移。 $V[r+1:n]$ 的前移会导致这些节点的秩发生变化，如果其他节点的前向或后向指针指向了它们，则需要更新这些指针。所以，如果用从 V 中删除节点的方式来清理内存，则删除一个节点的时间复杂度高达 $\Theta(n - r) = O(n)$ ，这是列表所不能允许的。

为了将删除节点的影响降到最低，则希望 r 越大越好。当 $r = n - 1$ （被删除的节点在向量末尾）时， $\Theta(n - r) = O(1)$ ，这是列表所理想的结果。因为列表的元素次序和 V 中的元素次序无关，所以您可以很自然地想到，只需要将 $V[r]$ 交换到 $V[n - 1]$ ，就可以顺利在 $O(1)$ 的时间内删除了。一个可能的实现原理如图3.9所示。

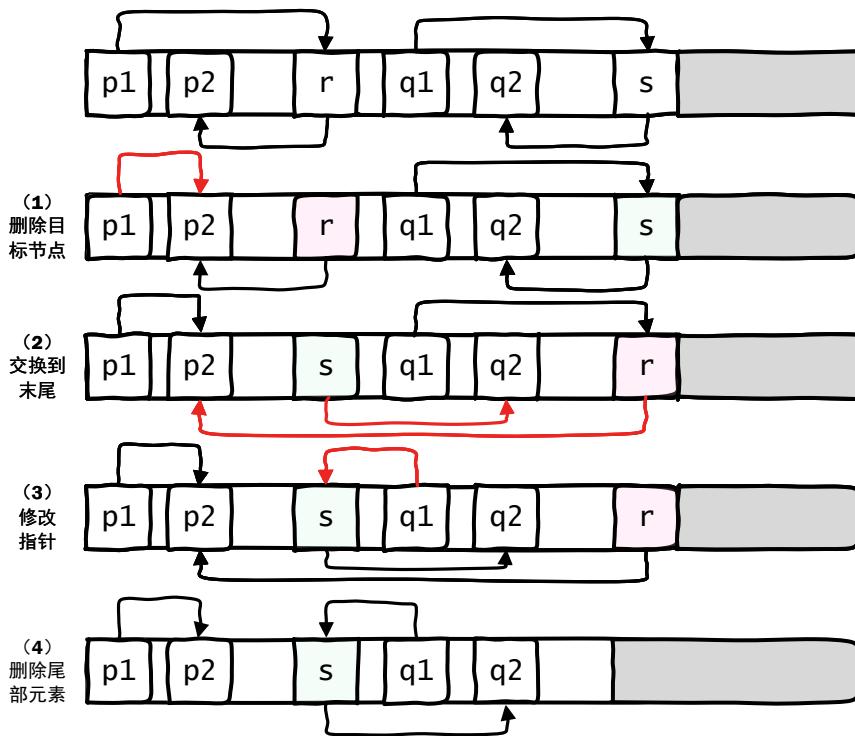


图 3.9 在静态列表上删除一个元素

```

1 T remove(Rank r) override {
2     T e { std::move(getNode(r).data()) };
3     getNode(getNode(r).prev()).next() = getNode(r).next();
4     getNode(getNode(r).next()).prev() = getNode(r).prev();
5     Rank s { m_data.size() - 1 };
6     if (r != s) {
7         getNode(r) = std::move(getNode(s));
8         getNode(getNode(r).prev()).next() = r;
9         getNode(getNode(r).next()).prev() = r;
10    }
11    m_data.pop_back();
12    return e;
13 }
```

对于元素次序不重要的向量结构，在执行删除时，可以将被删除的元素移动到向量末尾再删除，可以将删除操作的时间复杂度从 $\Theta(n - r)$ 降低为 $O(1)$ ，这是一个常用的技巧。在后面的章节中还会再次出现。当然，根据我们在3.3.2节中的经验，这种对值的直接操作是不安全的。所以，如果确保空间足够，也可以选择静态列表在删除元素的时候不释放空间：在机试的时候通常都是这么做的。

如果您实现了静态列表，可以使用 *StaticListTest.cpp* 进行测试。

3.7 本章小结

和向量相比，列表的链式结构显得不那么直观，在列表上进行插入、删除等操作时对指针的多次赋值，对初学者而言颇有难度。以列表为背景的算法设计题也常常从链式结构本身的性质入手，重点考察学生对于通过指针赋值进行链表操作的能力。在具体的题目中，有无头节点、有无尾节点、单向还是双向、顺序还是循环等因素，都会影响链表算法的具体实现，因此记忆代码是不可行的，应当理解设计列表操作的一般方法。

本书采用智能指针实现列表，为读者提供了一个不同于教材的视角：所有权视角。每个节点被且仅被一个智能指针持有所有权，我希望这一特性使得读者能够在设计链表操作的时候能够有迹可循（尽管遇到的题目使用的是裸指针）。下面列出了本章的一些学习目标。

1. 您学会了绘制列表的指针关系图，通过图形设计算法。
2. 您学会了采用后向智能指针控制列表节点所有权，并能从所有权的视角触发设计指针赋值操作序列，注意避免“断链”和更新“外部指针”。
3. 您了解到列表作为线性递归定义的数据结构，适合使用减治策略设计算法。
4. 您了解到设计列表上的算法时，可以有针对数据域和针对指针域两种路径。
5. 您了解到对没有头节点的列表设计算法时，可以使用哑节点作为临时头节点。

表 3.5 向量、列表和单向列表的对比

	向量	双向列表	单向列表
存储连续	√	✗	✗
性能稳定	✗	√	√
增删灵活	✗	√	○
位置安全	○	√	✗
节约空间	√	✗	○

向量和列表的区别也是一个重要的问题。本书从插入、删除和归并排序等几种典型场景出发，对向量、双向列表和单向列表的区别作出了分析。在表中提供了几个视角，您应当根据自己的理解，列出更加全面的对比表格。

第4章 栈

线性表是一个非常“开放的”数据结构，您可以读、写、插入和删除线性表上的任何一个数据单元。但在实际的计算模型中，并不是所有的数据结构都允许您这样做。通过对允许访问的数据进行限制，我们可以过滤掉多余的信息，简化模型的复杂程度，从而更快更好地解决问题。

栈 (stack)、**队列** (queue) 以及**双端队列** (deque) 就是典型的“访问受限制”的数据结构，它们的实现都是以线性表作为基础的；如果说线性表是“容器”，那么栈和队列这些结构更接近于“接口”，在 C++ STL 中称它们为容器适配器 (adapter)。另一种经典的限制访问的数据结构是**优先队列** (priority queue)，它需要以树作为基础，因此放到较后的章节讨论。栈和队列限制了用户访问元素的范围，这使得它们能够防止用户做出对进程、系统、计算机或网络有害的“非法”行为，在《操作系统》和《网络原理》中都能看到它们的应用。本章将首先讨论栈及其应用。

4.1 栈的性质

4.1.1 栈的定义

栈 (stack) 是一种特殊的线性表。对于栈 $S[0:n]$ ，它的插入、访问（因为只能读一个元素，不存在“查找”）、删除操作均只能对**栈顶**元素 (top 或 peek，即栈的最后一个元素) $S[n - 1]$ 进行。如图4.1所示，当栈顶为 x 时，三种操作都只能在 x 处进行。红色标出了操作之后的新栈顶，蓝色标出了操作之后的不可访问区域。

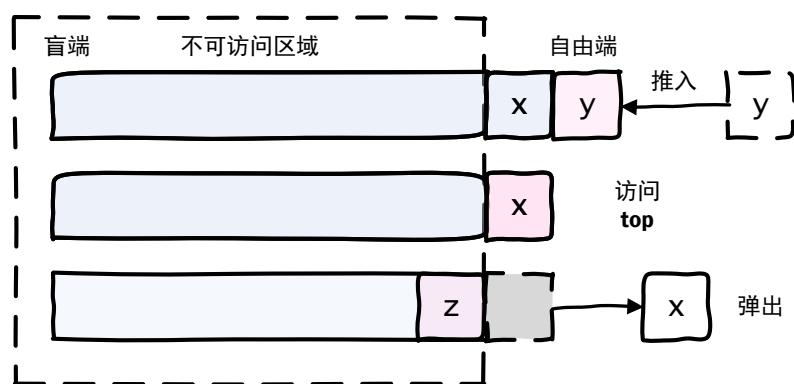


图 4.1 栈的推入、访问、弹出

1. 栈的插入就是在 $S[n - 1]$ 后面插入新的元素（称为入栈或推入，push）。
2. 栈的访问就是取 $S[n - 1]$ 。
3. 栈的删除就是将 $S[n - 1]$ 从栈中删除（称为出栈或弹出，pop）。

由于弹出后的元素往往另有他用，出栈操作会将被删除的栈顶元素返回。

```

1 template <typename T>
2 class AbstractStack : public DataStructure<T> {
3 public:
4     virtual void push(const T& e) = 0;
5     virtual T pop() = 0;
6     virtual T& top() = 0;
7 };

```

4.1.2 栈的结构

由于栈实质上是对线性表的访问权限作出限定，所以很容易在线性表的基础上建立栈。以向量（顺序表）实现的栈称为顺序栈，以列表（链表）实现的栈称为链栈。您很容易自己实现一个栈，并使用 *StackTest.cpp* 进行测试。

```

1 template <typename T, typename Linear = Vector<T>>
2     requires std::is_base_of_v<AbstractLinearList<T>, typename
3         Linear::position_type>, Linear>
4 class Stack : public AbstractStack<T> {
5     Linear L;
6 public:
7     constexpr static bool is_vector = std::is_base_of_v<
8         AbstractVector<T>, Linear>;
9     void push(const T& e) override {
10         if constexpr (is_vector) {
11             L.push_back(e);
12         } else {
13             L.push_front(e);
14         }
15     }
16     T pop() override {
17         if constexpr (is_vector) {
18             return L.pop_back();
19         } else {
20             return L.pop_front();
21         }
22     }
23     T& top() override {
24         if constexpr (is_vector) {
25             return L.back();
26         } else {
27             return L.front();
28         }
29     }
30     size_t size() const override { return L.size(); }
31 };

```

这里需要注意的是，如果以向量（常规方式）实现栈，则我们总是使用末尾元素代表栈顶，就像在上一节所定义的那样。向量在头部做插入和删除操作的效率

非常低，所以向量只能以尾部作为栈顶。但如果以单向列表实现栈，末尾元素变得不可访问，所以应当反过来以起始元素代表栈顶。双向列表因为是对称的，所以在末尾或起始都可以作为栈顶。

显然无论是顺序栈还是链栈，取顶的时间复杂度是 $O(1)$ ，入栈和出栈的时间复杂度在分摊意义下也是 $O(1)$ 。根据上一章的介绍您可以知道，用列表实现的栈，时间效率和空间效率都不如向量，但稳定性稍好。在绝大多数应用场景下，使用向量实现栈都比使用列表实现栈更优，很少会出现使用链栈的场景。

4.1.3 后入先出

因为栈只能从尾部进行操作的特性，栈可以被写作 $[S_0, S_1, S_2, \dots, S_{n-1}]$ 的形式，左侧的中括号表示不可操作的一端（盲端），右侧的尖括号表示可操作的一端（自由端）。更常见的一种表示方法是把它竖过来：

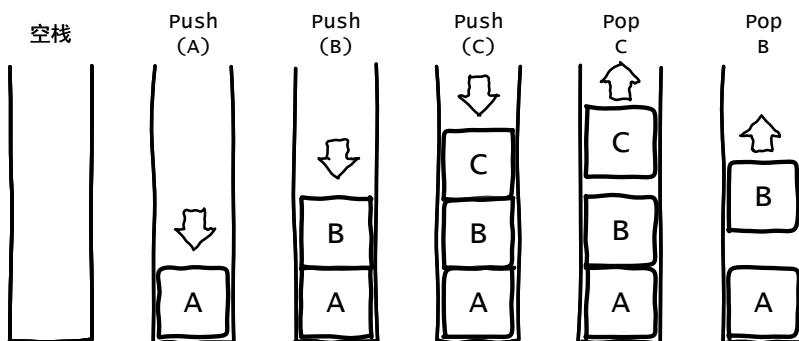


图 4.2 栈的桶式表示

如图4.2所示，可以把栈想成一个桶，盲端是桶的底部，自由端（栈顶）是桶中最上面的物品。于是，`push`操作就是往桶里放东西，后放的东西总是放在先放的东西的上面，像上图中，B 放在 A 上面，C 放在 B 上面。而要取东西（`pop`）的时候，每次只能取桶里最上面的东西，也就是栈顶元素。在上图里，为了把 B 取出来，必须先把 B 上面的 C 取出来。

就像列表相关问题经常用链式图帮助思考一样，处理和分析栈的相关问题经常要用到上面这种桶式图。从上图中不难发现，栈的最重要的性质是：先入栈的元素后出栈，后入栈的元素先出栈，简称为“**后入先出**”（LIFO, Last In First Out）。生活中有不少 LIFO 的例子，比如，桌子上一大摞书需要先搬开上面的才能拿下面的。这些现象被抽象化到计算机中处理就对应栈这种数据结构。严格来说，栈也应该采用 LIFO 进行定义，只要能够实现 LIFO 功能的数据结构都可以认为是栈，而不需要限制在4.1.1节中这种带盲端和自由端的线性表。不过由于这种单端自由的线性表实现非常简单，通常默认栈是用这种方法实现的。在《队列》和《优先队

列》章中您将看到其他适配器结构会有多种不同的实现方法。

和栈相关的题目，通常有两个方向。

1. 考察对栈性质（也就是 LIFO）的理解。
2. 在访问受限的情况下设计算法。

在下面的几节，将介绍栈的几个应用以加深您对栈性质的理解。这些应用本身也是重要的考点，栈的大多数题目都出自这些应用。

4.2 出栈序列

4.2.1 出栈序列的定义

出栈序列（又称**栈混洗序列**）是关于栈的一个重点问题。下面首先给出出栈序列的定义。

给定一个序列 $A = (a_1, a_2, \dots, a_n)$ ，如果对于序列 $B = (b_1, b_2, \dots, b_n)$ ，存在对空栈 S 的入、出栈操作各 n 次的操作序列 $O = (o_1, o_2, \dots, o_{2n})$ ，使得当序列 O 中的入栈操作为依次将 a_1, a_2, \dots, a_n 入栈时，序列 O 中的出栈操作出栈的元素恰好依次为 b_1, b_2, \dots, b_n ，则称序列 B 为序列 A 的一个**出栈序列**。

操作序列中的每个元素是 **push** 或者 **pop**，为简化叙述，本书将使用 \vee 和 \wedge 分别表示它们。如果您用桶式图想象栈的形象，那么这两个符号是直观的。

一个显而易见的结论是：出栈序列是全排列的子集。以入栈序列为 $(1, 2, 3)$ 为例，则 $(3, 2, 1)$ 是它的一个出栈序列，对应的操作序列为 $\vee\vee\vee\wedge\wedge\wedge$ 。您可以自己验证， $(2, 3, 1)$ 、 $(2, 1, 3)$ 、 $(1, 3, 2)$ 以及 $(1, 2, 3)$ 自身，都是它的一个出栈序列；而 $(3, 1, 2)$ 则无法成为一个出栈序列。

事实上，如果入栈序列是 $(1, 2, \dots, n)$ ，则 $B = (b_1, b_2, \dots, b_n)$ 是出栈序列，当且仅当不存在“312”模式。即：不存在 $i < j < k$ ，使得 $b_j < b_k < b_i$ 。这个命题您可以在阅读了出栈序列的相关知识后自己证明。

根据出栈序列的定义，可以立刻得到，在给定 A 的情况下，从 O 到 B 是一个满射。那就自然地会引发猜想，从 O 到 B 也应该是一个单射。这个问题可以用递归法分析，具体证明过程您可以自己补全。设 $O = (o_1, o_2, \dots, o_{2n})$ 中的最后一个 \vee 是 o_j ，则下一个操作 o_{j+1} 一定是 \wedge ，且 o_j 入栈的和 o_{j+1} 出栈的元素都是 a_n 。那么，将 o_j 、 o_{j+1} 和 a_n 删除就递归到了 $n - 1$ 的情形。递归边界 $n = 1$ 结论显然。这样，每个出栈序列就唯一对应了一个操作序列。

4.2.2 出栈序列的计数

从上一小节的分析可知，给定入栈序列的情况下，出栈序列的数量等于操作序列的数量，而操作序列 $O = (o_1, o_2, \dots, o_{2n})$ 的数量和入栈序列的内容无关，只和入栈序列的长度 n 相关。

在这一小节中，先设长度为 $2n$ 的操作序列 O 的数量是 $f(n)$ 。为了求解这个函数，需要确定操作序列 O 需要满足的条件。

1. 包括 n 个 \vee 和 n 个 \wedge 。
2. 对于操作序列的任意一个前缀，前缀中 \vee 的数量一定不小于 \wedge 的数量。否则，在这个前缀的操作结束之后，栈的规模会变成负数，这是不可能的。

容易验证满足上述两个条件的序列，也一定是合法的、可以生成对应出栈序列的操作序列。下面利用这两个条件，去推导 $f(n)$ 满足的递归式。

由于条件 (2)， o_1 必定是 \vee 。设 o_1 入栈的 a_1 在 o_k 时出栈，其中 $1 < k \leq 2n$ 。又由于 a_1 在 o_1 的时候被压入了栈的底部，所以 o_k 之后，栈变成了空栈。因此， (o_1, o_2, \dots, o_k) 和 $(o_{k+1}, o_{k+2}, \dots, o_{2n})$ 各自都是一个比较短的、符合条件的操作序列。因而 k 必须是偶数，设 $k = 2i$ ，其中 $1 \leq i \leq n$ 。

在这两个操作序列中， o_1 和 o_k 已经被确定了，而 $(o_2, o_3, \dots, o_{k-1})$ 有 $f(i-1)$ 种可能性， $(o_{k+1}, o_{k+2}, \dots, o_{2n})$ 有 $f(n-i)$ 种可能性。于是：

$$f(n) = \sum_{i=1}^n f(i-1)f(n-i)$$

上述递归方程可以被改写为：

$$f(n+1) = \sum_{k=0}^n f(k)f(n-k)$$

使用生成函数法可以得到，这个递归方程可以解出显式的通项公式：

$$f(n) = \frac{C_{2n}^n}{n+1} = \frac{(2n)!}{(n+1)! \cdot n!}$$

这个数被称为卡特兰 (Catalan) 数，记为 $\text{Catalan}(n)$ 。

4.2.3 出栈序列的计数方法 *

本节介绍如何推导出栈序列的计数。如果您对数学问题不感兴趣，也没有必要特意去了解它，可以选择跳过本节的内容。

使用数学归纳法可以证明，这个递归方程的解确实是 $\text{Catalan}(n)$ ，不过数学归纳法要先猜出答案才能证明。这里介绍解决此类递归方程问题的生成函数法^[18]。生成函数法是《组合数学》学科的内容，并不在考研数学的要求范围内。

证明 设 $H(x) = \sum_{n=0}^{\infty} h_n x^n$, 其中 h_n 为待求解的 Catalan(n)。于是,

$$\begin{aligned} H^2(x) &= \sum_{n=0}^{\infty} h_n x^n \sum_{k=0}^{\infty} h_k x^k = \sum_{n=0}^{\infty} \sum_{k=0}^{\infty} h_n h_k x^{n+k} \\ &= \sum_{n=0}^{\infty} x^n \sum_{k=0}^n h_k h_{n-k} = \sum_{n=0}^{\infty} h_{n+1} x^n = \frac{H(x) - h_0}{x} \end{aligned}$$

◦

由 $H(0) = h_0 = 1$, 解得 $H(x) = \frac{1-\sqrt{1-4x}}{2x}$ 。将分子上的二次根式泰勒展开, 即可得到

$$H(x) = \sum_{n=0}^{\infty} \frac{C_{2n}^n}{n+1} x^n$$

■

此外有一种基于一一映射的计数方法, 这种方法相对生成函数法更加巧妙^[14]。

证明 我们分析出栈序列需要满足的两个条件。

1. 在长度为 $2n$ 的序列中安排 n 个 \vee 和 n 个 \wedge , 可能的情况有 C_{2n}^n 种。
2. 如果满足条件 (1) 而不满足条件 (2), 则必定存在一个最小的 k , 使得序列的前 k 项中, \vee 比 \wedge 少 1 个; 后 $n-k$ 项中, \vee 比 \wedge 多 1 个。那么, 保持后 $n-k$ 项不变, 令前 k 项的 \vee 变为 \wedge 、 \wedge 变为 \vee , 则得到了一个新的序列, 这个序列中有 $n+1$ 个 \vee 和 $n-1$ 个 \wedge 。可以证明这是一个一一映射。因此, 不满足条件的情况有 C_{2n}^{n+1} 种。

因此, 出栈序列的数量为

$$C_{2n}^n - C_{2n}^{n+1} = \frac{C_{2n}^n}{n+1}$$

■

递归方程法和一一映射法, 是计数问题的两种基本方法。相对来说, 递归方程法比较常规, 很容易列出递归方程, 但计算比较复杂; 一一映射法则需要较高的构造技巧, 而计算比较简单。

4.2.4 随机出栈序列 *

这一小节从扩展卡特兰数的角度出发, 讨论如何生成一个随机的栈操作序列^[14]。

考虑包含 p 个 \vee 和 q 个 \wedge 的序列, 其中 $0 \leq p \leq q$ 。如果在这个序列前添加 $q-p$ 个 \vee 可以得到合法的栈操作序列, 则下文称其为 p, q 的后缀操作序列 (显然, 一个合法操作序列的任一后缀都是后缀操作序列)。设 p, q 的后缀操作序列有 $C_{p,q}$

个，则可以得到 $C_{n,n} = \text{Catalan}(n)$ 。

直接对栈的操作序列做递归比较困难。而定义了后缀栈操作序列之后，就可以研究生过程中的中间结果。我们的总体目标是生成 $n + n$ 的后缀操作序列。如果从前到后生成，在某一步处已经生成了 $n - p$ 个 \vee 和 $n - q$ 个 \wedge ，那么剩余部分是一个 p, q 的后缀操作序列，有 $C_{p,q}$ 种等概率的可能性。从而，这一步生成 \vee 的概率是 $\frac{C_{p-1,q}}{C_{p,q}}$ ，生成 \wedge 的概率是 $\frac{C_{p,q-1}}{C_{p,q}}$ 。

上述结论表明，只要计算出 $C_{p,q}$ 的通项公式，就可以从前向后随机地逐个生成操作序列上的操作。使用上一小节介绍的方法构造一一映射，容易证明：

$$C_{p,q} = C_{p+q}^q - C_{p+q}^{q+1} = \frac{q-p+1}{q+1} C_{p+q}^q$$

因此，生成 \vee 的概率为：

$$\frac{C_{p-1,q}}{C_{p,q}} = \frac{q-p+2}{q-p+1} \cdot \frac{p}{p+q}$$

这意味着不需要实际计算组合数，就可以得到每次生成 \vee 和 \wedge 的概率。从而可以得到时间复杂度为 $\Theta(n)$ 的算法（假设随机数生成器的时间复杂度为 $O(1)$ ）生成一个长度为 $2n$ 的合法操作序列。

4.3 括号序列

4.3.1 合法括号序列的计数

卡特兰数不仅仅是出栈序列（合法操作序列）的数量，同时也是其他很多问题的答案。这些问题的解决，也普遍具有两条路径：

1. 从模型角度入手，将其变换为等价的出栈序列或合法操作序列问题，然后套用卡特兰数的通项公式。（一一映射法）
2. 从计算角度入手，列出递归方程，发现和卡特兰数的递归方程的相似性后，利用已知的卡特兰数通项公式求解。（递归方程法）

一个典型的例子是合法括号序列问题。考虑左括号和右括号组成的合法括号序列，可以用以下的递推定义：

1. 空串是合法括号序列。
2. 如果 S 和 T 是合法括号序列，那么 $(S)T$ 也是合法括号序列。

条件（2）的一个等价形式拆分成两个条件：如果 S 是合法括号序列，则 (S) 是合法括号序列；以及，如果 S 和 T 是合法括号序列，那么 ST 也是合法括号序列。这种两个条件的版本更符合直观认知。

您会发现，从 $(S)T$ 递归到长度更短的 S 和 T ，这个递归方法和前面推导合法

操作序列数量时使用的递归方法如出一辙，因此，用 \vee 代替左括号，用 \wedge 代替右括号，就可以建立合法括号序列到合法操作序列的一个映射。容易证明这是一个双射，从而化归到出栈序列的问题上来。另一个方向也是类似的，容易通过条件(2) 得到和之前完全一样的递归方程，从而解出卡特兰数的通项公式。

在这个问题上，从模型角度入手建立一一对应是显然的。但有一些问题，模型上可能一时看不出来，但通过计算角度发现结果是卡特兰数之后，就可以自然地联想到从模型上可以建立一一对应。比如，将凸 n 边形通过若干条互不相交的对角线划分为 $n - 2$ 个三角形，划分方法数量。这个问题的答案也是卡特兰数，由于和《数据结构》关系不大，这里不再展开。您可以自己分析这个问题。后面还会遇到的一个非常重要的卡特兰数应用是树和二叉树的计数问题，我们将在对应章节进行分析讨论。

4.3.2 实验：判断括号序列是否合法

回到合法括号序列的问题上来。发现合法括号序列的数量和出栈序列相同之后，在讨论合法括号序列的问题时，就可以自然地联想到从栈的角度入手。数量上相同只是初步的结论，更重要的是建立了——映射。比如，借助这个一一映射，就可以用栈来判断一个括号序列是否合法。

在这个实验中，我们将同时讨论小括号、中括号和大括号，因此我们首先建立括号直接的对应关系。在存在多种括号的场合，只需要修改合法括号序列中的条件(2)，将递推定义的合法括号序列包括 $[S]T$ 和 $\{S\}T$ 。代码可以在 *ParenMatch.cpp* 中找到。括号匹配算法接受一个字符串。这不是一个有难度的问题，建议您自己实现它。

```

1 class ParenMatch : public Algorithm<bool, const string&> {
2 protected:
3     char left(char c) {
4         switch (c) {
5             case '(': case ')': return '(';
6             case '[': case ']': return '[';
7             case '{': case '}': return '{';
8             default: return 0;
9         }
10    }
11 }

```

从上面的分析中我们可以看到，左括号和右括号在建立一一映射的时候分别被映射为了 \vee 和 \wedge 。因此，我们从左到右扫描括号序列的时候，每次扫描到左括号，就进行一次入栈操作；每次扫描到右括号，就进行一次出栈操作。由于我们考虑了三种括号，所以我们需要保证左括号和右括号是同一种。因此，每次扫描到左

括号，就将它入栈（因此栈里只有左括号）；每次扫描到右括号，就从栈里弹出一个左括号，判断是否和右括号匹配。示例程序如下所示。

```

1 bool operator() (const string& expr) {
2     Stack<char> S;
3     for (char c : expr) {
4         if (char l { left(c) } ; l == c) {
5             S.push(c);
6         } else if (l) {
7             if (S.empty() || S.top() != l) return false;
8             S.pop();
9         }
10    }
11    return S.empty();
12 }
```

如果只有一种括号，那么栈中的所有元素都是相等的。在这种情况下，栈也变得不必要，因为我们事实上只需要知道栈的规模，用一个整数来表示即可；从而将空间复杂度降低到了 $O(1)$ 。请您自己完成这种情况的算法设计，并使用在4.2.4节中介绍的随机序列生成方法进行测试。

4.4 栈与表达式

4.4.1 表达式的定义

括号主要的用处，就是给表达式规定计算顺序。所以，自然地就能想到，栈也可以被用来计算表达式。

回顾上一节，括号序列和栈的操作序列相对应，而一个出栈序列，是由入栈序列和操作序列共同决定的。从信息的角度看，如果要在出栈序列和表达式之间建立联系，且已知操作序列和括号序列之间有联系，那么顺理成章地，就会想到，入栈序列应当和表达式中的其他部分——也就是**操作数** (operand) 和**运算符** (operator) 建立联系。

那么接下来，就需要推导这个联系了。为了更清晰地展示推导过程，以下采用表达式 $1 + 2 * (3 - 4)$ 作为例子。

1. 建立完整的括号序列。

这样做的目的是，让括号序列能够完全地和运算次序形成一一对应。原有的表达式中，有一些括号被省略了，这一步的目的是将它补全。在例子中， $1 + 2 * (3 - 4)$ 被变换为 $((1) + ((2) * ((3) - (4))))$ 。

这里给每一个操作数也加上了一堆括号；这是因为，我们的目标是将“入栈序列”和“操作数与运算符”之间建立联系。在表达式中，操作数和运算符一共有 7 个 ($1, 2, 3, 4, +, *, -$)，因此入栈序列的长度为 7，相应地，操作序

列的长度应该为 14，也就是说括号序列的长度应该为 14（7 对括号）。在这 7 对括号中，有 3 对是用来描述运算符的计算次序的，而另外 4 对则用来描述操作数的位置。这样才能达成一一对应。

2. 构造入栈序列和出栈序列。

利用第 1 步中建立的“括号对”和“操作数与运算符”的一一对应关系，决定每一对括号对应的入栈或出栈的元素。如果某一对括号对应的运算符或者操作数为 c ，那么将左括号变换为 $\vee(c)$ ，右括号变换为 \wedge （出栈元素恰好为 c ），就得到了一个带参数的操作序列。

在例子中，第 1 步加入了 7 个括号，其中 3 对用来描述 3 个运算符的运算次序，4 对用来描述 4 个操作数的位置，也就是说，每一对括号都对应了一个运算符或者操作数。它对应的带参数操作序列是： $\vee(+)\vee(1)\wedge\vee(*)\vee(2)\wedge\vee(-)\vee(3)\wedge\vee(4)\wedge\wedge\wedge\wedge$ 。对应的入栈序列是： $+ \ 1 \ * \ 2 \ - \ 3 \ 4$ ；对应的出栈序列是： $1 \ 2 \ 3 \ 4 \ - \ * \ +$ 。

定义这个入栈序列为**前缀表达式**（又称波兰式），出栈序列为**后缀表达式**（又称逆波兰式）。在前缀表达式和后缀表达式中，由于两个数字可能连在一起，所以为了区分边界，通常使用空格或其他分隔符隔开相邻的两个元素。您所熟知的数学的表达式形式，即 $1 + 2 * (3 - 4)$ ，被称为**中缀表达式**。

这三个概念来自于运算符相对于操作数的位置。在前缀表达式中，运算符出现在它所对应的操作数之前；后缀表达式放在之后，而中缀表达式的运算符出现在中间。

后缀表达式和前缀表达式的性质大多是对偶的，而后缀表达式更适合使用计算机计算，因此，在考试中基本上不会出现前缀表达式，而后缀表达式则是非常重要的考察内容。

4.4.2 表达式中的元素

为简单考虑，本书中只讨论操作数为整数的表达式，讨论的运算符则包括 7 种：加、减、乘、除、取余、乘方、阶乘。通过对负号和阶乘的支持，我们要求表达式处理中允许单目运算符的存在，且单目运算符可以放在操作数的左侧或者右侧。表达式解析作为《编译原理》的组成部分，如果从自动机的角度观察它会更加本质；但这已经超出了《数据结构》的讨论范畴。

需要指出的是，表达式的实现在各个教材中会有很大的、基础性的差异。比如，针对表达式中的元素的抽象，有三种解决方案。

1. 分离方案^[6]。操作数和运算符被分离，使用不同的类型的数据单元进行存储。大多数教材采用了这样的方案。这种方案的优点在于不需要自己写类对数据

单元进行封装（可以使用基本数据类型，如**char**和**int**），缺点是在处理表达式时往往需要两个栈分别处理操作数和运算符。而在表达式中，操作数和运算符本质上都是表达式的元素，分离方案没有显示出这一点。

2. 合取方案。在每个数据单元中，存储一个运算符和一个操作数。对于输入的中缀表达式，将每个运算符和紧随其后的一个操作数合并在一个数据单元存储。通过在整个表达式的开头和结尾增加一对括号，我们可以保证在中缀表达式中每个操作数都紧跟在某个运算符之后。这种方案的优点在于处理中缀表达式时非常方便，缺点是将操作数和它前面的运算符合并，这一操作并不自然。
3. 析取方案。在每个数据单元中，存储一个运算符或一个操作数。这种方法可以很好地表达操作数和运算符都是表达式元素的本质。

本书将采用析取方案，我们将从表达式元素的基本功能开始，逐步往其中添加功能，使其可以支持表达式转换和计算。由于表达式是一个充满实现多样性的话题，本书在此实现了充分的解耦。您可以直接使用本书提供的表达式元素类，只关注表达式本身的算法。或者，您可以使用本书提供的抽象表达式元素类，自己实现析取方案的表达式元素类。或者，您还可以使用本书所用的抽象表达式元素类，实现合取方案的表达式元素类（您需要修改表达式类来支持合取方案）。或者，您也可以不使用表达式元素类，改用基本数据类型，通过分离方案实现表达式的各种算法。

表达式的每个元素可以定义如下。

```

1 class AbstractExpressionElement {
2 public:
3     virtual char getOperator() const = 0;
4     virtual const int& getOperand() const = 0;
5     virtual bool isOperator() const = 0;
6     virtual bool isOperand() const = 0;
7     virtual bool operator==(char op) const = 0;
8     virtual bool operator!=(char op) const = 0;
9     virtual void setOperator(char op) = 0;
10    virtual void setOperand(const int& num) = 0;
11    virtual std::pair<bool, bool> operandPosition() const = 0;
12    virtual int apply(int lhs, int rhs) const = 0;
13 }

```

我们使用**apply**方法表示运算，当元素是运算符时，它接收至多两个操作数，返回运算的结果；当元素是操作数时，它直接返回操作数本身。**operandPosition**方法则表示操作数的位置，当元素是运算符时，两个返回值各表示左侧和右侧是否可以接受操作数（比如阶乘，就只允许左侧有操作数）；当元素是操作数时，当然两边都不能有操作数。上面的接口我们可以通过C++17引入的**std::variant**很

方便地实现。

```

1  class ExpressionElement : public AbstractExpressionElement {
2      std::variant<char, int> m_element;
3  public:
4      char getOperator() const override {
5          return std::get<char>(m_element);
6      }
7      const int& getOperand() const override {
8          return std::get<int>(m_element);
9      }
10     bool isOperator() const override {
11         return std::holds_alternative<char>(m_element);
12     }
13     bool operator==(char op) const override {
14         return isOperator() && getOperator() == op;
15     }
16     bool isOperand() const override {
17         return std::holds_alternative<int>(m_element);
18     }
19     void setOperator(char op) override {
20         m_element = op;
21     }
22     void setOperand(const int& num) override {
23         m_element = num;
24     }
25     std::pair<bool, bool> operandPosition() const override {
26         if (isOperator()) {
27             switch (getOperator()) {
28                 case '(': return { false, true };
29                 case ')': case '!': return { true, false };
30                 case '+': case '-':
31                 case '*': case '/':
32                 case '%': case '^': return { true, true };
33             }
34         }
35         return { false, false };
36     }
37     int apply(int lhs, int rhs) const override {
38         if (isOperator()) {
39             switch (getOperator()) {
40                 case '+': return lhs + rhs;
41                 case '-': return lhs - rhs;
42                 case '*': return lhs * rhs;
43                 case '/': return lhs / rhs;
44                 case '%': return lhs % rhs;
45                 case '^': return Power {}(lhs, rhs);
46                 case '!': return Factorial {}(lhs);
47             }
48         }
49         return getOperand();
50     }
51 };

```

4.4.3 将字符串解析为中缀表达式

本节讨论将字符串解析为中缀表达式。我们将表达式看成是析取元素组成的线性表，每一个元素存储一个操作数或一个运算符。在将字符串解析为中缀表达式的这个过程中，我们需要进行两件事：

1. 将操作数据提取出来。每个运算符都是一个字符，而操作数可能不止有一个字符。因此，当扫描到数字时，不能立刻将它加入到表达式中，需要继续向下扫描，直到扫描到运算符或字符串尾，才能确认这个数字结束。
2. 将表达式中的负号和减号区分开来。减号需要两个操作数进行运算，而负号只是将它后面的操作数取反，二者具有不同的语义。通过分析可以发现，运算符`-`表示负号，当且仅当它在字符串起始位置，或前一个元素是左括号。

一种可能的实现如下所示。在上面的分析中我们发现，总是需要对字符串尾和字符串头进行特判。为了消除这些特判，我们可以选择在字符串的两边增加一对括号。

```

1 Expression(const std::string& expr) {
2     std::string expr2 { '(' + expr + ')' };
3     std::string num;
4     for (auto& c : expr2) {
5         if (isdigit(c)) {
6             num += c;
7         } else {
8             if (!num.empty()) {
9                 this->push_back(ExpressionElement(std::stoi(num)));
10                ;
11                num.clear();
12            }
13            if (c == '-' && this->back() == '(') {
14                num += c;
15            } else {
16                this->push_back(ExpressionElement(c));
17            }
18        }
19    }
}

```

这里采用的`std::stoi`将字符串转换为整数，语义清晰但相对低效，您可以自己改写处理操作数的部分以提高效率。

4.4.4 中缀表达式转换为后缀表达式

在4.4.1节中，我们已经介绍了将中缀表达式转换为后缀表达式的方法。在考试中为简化起见，通常采用以下的快速手工方法，如图4.3所示。

1. 补全运算符对应的括号，而不补操作数对应的括号。

2. 对于每一对括号，设它对应的运算符为 c ，则删掉左括号，将右括号替换为 c ；如此就得到了后缀表达式。

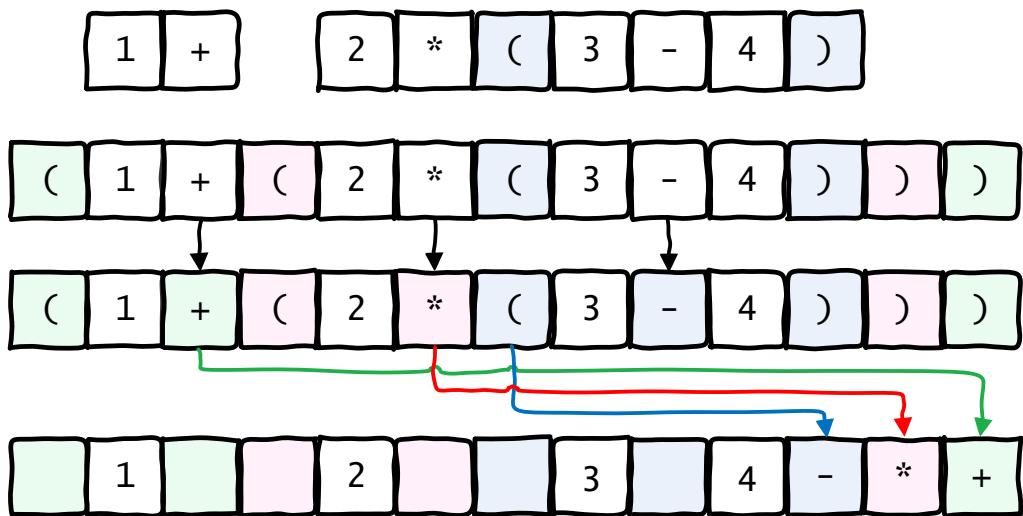


图 4.3 中缀表达式转换为后缀表达式

如果要得到前缀表达式，类似地，只需要删掉右括号，将左括号替换为 c 即可。前缀表达式的运算符位于入栈的位置上，而后缀表达式的运算符位于出栈的位置上，这都是非常自然的；反而，我们熟悉的中缀表达式，运算符的位置是不自然的。所以，为了计算中缀表达式，可以先将它转换为前缀表达式或后缀表达式。此处以后缀表达式为例，您可以自行完成前缀表达式的情况。

在手工方法中，操作数仍然在它原本的位置上，而运算符则被移动到了它出栈的位置（必定后于运算符出现的位置）。因此，我们可以对中缀表达式从前到后扫描，当扫描到操作数的时候将其直接加入后缀表达式，而当扫描到运算符的时候将其入栈，在运算符出栈的时候，再将它加入到后缀表达式中。但是，当在计算机中处理中缀表达式的时候，并不是每一个运算符都有对应的一对括号，因此，我们需要配合运算符的优先级进行处理。

我们的目标是寻找每个运算符应当在哪个位置出栈。我们考虑不为左右括号的运算符 c ，因为左右括号不会被加入到后缀表达式中。考虑表达式 $(A c B)$ ，其中，左右的一对括号是 c 对应的括号， A 和 B 都是表达式。

1. 如果这对括号实际存在，那么我们分析表达式 B 的情况。如果 B 是操作数或者被一对括号包裹，则它确实参与了 c 的运算。如果 B 是其他情况，则可以设 B 为 $B_1 c_1 B_2$ ，其中 c_1 是 B 在最后一步进行的运算， B_1 和 B_2 都是表达式。那么，只有 c 的优先级小于 c_1 时， B 作为整体才会参与 c 的运算。否则，在 $A c B_1 c_1 B_2$ 中，会先计算 $A c B_1$ ， c 对应的括号应当不包括 $c_1 B_2$ 这一段。
2. 反过来，如果这对括号实际不存在，那么我们需要定位到它的位置。根据上

面的分析可以知道，我们从 c 向后可以继续扫描，直到扫描到优先级不大于 c 的运算符 c_1 为止（不包括括号里的运算符）。 c 对应的括号应当出现在 c_1 之前；所以在扫描到 c_1 的时候将 c 出栈。由于我们在上一节中，在表达式两端添加了一堆括号，所以不需要特别判断扫描到末尾的情况（因为末尾一定有实际存在的括号）。

如果存在右结合运算符，则不等号的严格性会发生变化，(1) 中的小于应当改为不大于；(2) 中的不大于应当改小于。本书实现的运算符中不包含右结合运算符，因此不考虑这个问题；本书的示例代码允许扩展到右结合运算符中，您可以自己尝试将运算符实现为右结合的形式。

综上所述，我们可以得到这样的算法框架：

1. 当扫描到操作数时，直接加入后缀表达式。
2. 当扫描到非括号的运算符 c 时，首先将栈内优先级不小于 c 的运算符依次弹出并加入后缀表达式，然后将 c 入栈。

现在考虑括号的情况。我们会发现，左括号作为运算符，很难确定它的优先级。一方面，当扫描到左括号的时候，它应当具有最高的优先级，左括号内部永远应当先于当前的栈顶运算符进行计算。另一方面，当左括号在栈内时，它应当具有（除了右括号外）最低的优先级，因为必须计算完括号才能脱括号，所以在找到其对应的右括号之前，任何运算符都不应该让左括号出栈。

这就意味着我们不能用单一的优先级来描述运算符，而是需要将其拆分为栈内优先级和栈外优先级。当我们扫描到运算符 c 时，将栈顶的栈内优先级，和 c 的栈外优先级进行比较。如果栈顶的栈内优先级不小于 c 栈外优先级，则将栈顶弹出。左括号出栈后不进入后缀表达式，右括号则不必入栈，因为右括号对应的栈操作是 \wedge ，扫描到右括号的时候恰好会弹出它对应的左括号。

除了左括号外，其他运算符的栈内和栈外优先级相同；右括号的栈内优先级和栈外优先级都是最低。左括号具有最高的栈外优先级，以及除右括号之外最低的栈内优先级。因此，左括号入栈必定不会引起其他运算符出栈（括号内总是先于括号外计算），且仅当扫描到右括号的时候左括号可以出栈。

如果您想要同时实现左结合和右结合运算，可以对栈内和栈外的优先级做进一步细化的区分。在只考虑左结合运算的情况下，两个运算符的比较函数如下。

```

1 inline static const std::unordered_map<char, int> priority_left
2 {
3     {'(', 0}, {')', 0}, {'^', 3}, {'!', 4},
4     {'+', 1}, {'-', 1}, {'*', 2}, {'/', 2}, {'%', 2}
5 };
inline static const std::unordered_map<char, int> priority_right {

```

```

6   {'(', 6}, {')', 0}, {'^', 3}, {'!', 4},
7   {'+', 1}, {'-', 1}, {'*', 2}, {'/', 2}, {'%', 2}
8 };
9 bool prior(char next) const override {
10    if (isOperator()) {
11        auto prior_left { priority_left.find(getOperator()) };
12        auto prior_right { priority_right.find(next) };
13        if (prior_left != priority_left.end() && prior_right != priority_right.end()) {
14            return prior_left->second > prior_right->second;
15        }
16    }
17    return false;
18 }
```

上面虽然进行了很多分析，但目的只是为了让您更好地理解我们为什么要这样设计算法。算法的实际实现非常简单，如下所示。如果您想要实现前缀表达式，需要改为从后向前扫描，感兴趣的话可以自己完成。

```

1 void infix2suffix() {
2     Stack<ExpressionElement> S;
3     Expression suffix;
4     for (auto& e : *this) {
5         if (e.isOperand()) {
6             suffix.push_back(std::move(e));
7         } else {
8             while (!S.empty() && S.top().prior(e.getOperator()))
9                 {
10                     if (auto op { S.pop() }; op != '(') {
11                         suffix.push_back(std::move(op));
12                     } else {
13                         break;
14                     }
15                 }
16             if (e != ')') {
17                 S.push(std::move(e));
18             }
19         }
20     }
21     *this = std::move(suffix);
22 }
```

4.4.5 后缀表达式转换为中缀表达式

现在考虑上述问题的反问题：如何从后缀表达式转换为中缀表达式？

下面从信息的角度分析，“中缀转后缀”和“后缀转中缀”有什么不同。注意到，中缀表达式是由“语义元素”（操作数和非括号的运算符）以及“括号序列”共同组成的。“语义元素”是参与运算的元素，对应了入、出栈过程中操作的数据；而“括号序列”是进行运算的次序，对应了入、出栈过程中的操作序列。所以知道

中缀表达式，也就知道了操作序列的每一步入、出栈了什么元素，从而得到入栈序列（前缀表达式）和出栈序列（后缀表达式）。从信息的角度看，中缀表达式中包含的信息，包括了前缀表达式和后缀表达式所需的信息。

但是，后缀表达式仅仅是由“语义元素”组成的，它本身只是一个出栈序列。从前面的小节中您已经知道，给定出栈序列而不给定操作序列，则有 $Catalan(n)$ 种可能的入栈序列。所以，如果仅仅依赖后缀表达式的文本，而不借助其他先验信息，那么从后缀表达式反推中缀（前缀）表达式是不可能的、是缺少信息的。要能够实现后缀表达式到中缀表达式的转换，必须要知道文本以外的额外信息。

这个先验的额外信息就是：每个运算符需要的操作数数量。在常见的运算符中，加、减、乘、除、乘方，操作数数量都是 2；阶乘的操作数数量是 1。需要特别注意的是，当“减号”和“负号”都被作为运算符处理，且采用同一符号 $-$ 时，它的操作数数量是不确定的，这一不确定性会导致后缀表达式出现歧义。中缀表达式可以通过判断 $-$ 之前是否为操作数来进行区分，而后缀表达式无法进行区分。比如：对于后缀表达式 $1\ 2\ -\ 3\ -\ -$ ，它的三个 $-$ 中有一个代表负号，另外两个代表减号。因此，这个后缀表达式可能对应 3 种不同的中缀表达式： $1 - ((-2) - 3)$ 、 $(1 - 2) - (-3)$ 和 $-((1 - 2) - 3)$ 。前面几节我们处理中缀表达式的时候，在解析阶段就已经将减号和负号进行了分离。负号被认为是操作数的一部分，而所有出现在表达式中的运算符 $-$ 都是减号；在后缀表达式的处理中我们也将延续这一约定以避免上述歧义。

下面解释，这个先验的额外信息是如何帮助后缀表达式转换成中缀表达式的。这一部分的内容如果不感兴趣可以略过。

假设某个运算符 c 的操作数数量是 2。在补全了所有括号的中缀表达式中，这个操作数出现在 $((A) c (B))$ 这个片段里，其中 A 和 B 都是中缀表达式。按照上一节的方法，这个片段化为后缀表达式是 $A' B' c$ ，其中 A' 和 B' 是 A 和 B 对应的后缀表达式。于是得到：

定理 4.1：对于后缀表达式 E 中的每个运算符 c ，都存在 E 中的一个连续的子序列 E_0 ， E_0 是以 c 结尾的后缀表达式。

在此之前讨论一般的出栈序列的时候， A' 和 B' 的划分是任意的，所以会得到卡特兰数的递归方程。但在讨论后缀表达式的时候，需要受到操作数数量的限制，因此 A' 和 B' 的划分不是任意的。事实上，只有一种划分是合法的。

我们知道，后缀表达式的开头必定是操作数，末尾必定是运算符。对于语义元

素组成的序列 $C = (c_1, c_2, \dots, c_n)$, 定义:

$$f(c_i) = \begin{cases} 1, & c_i \text{ is operand,} \\ 1 - \text{Cnt}(c_i), & c_i \text{ is operator with Cnt operands} \end{cases}$$

定理 4.2: 令 $f(C) = \sum f(c_i)$, 则对于合法的后缀表达式 E , $f(E) = 1$ 。

证明 使用数学归纳法, 对 E 中含有的运算符数量归纳。

1. 对于单个操作数, $f(E) = 1$ 。
2. 对于以 k 元运算符结尾的表达式, 设 E 为 $A_1 A_2 \dots A_k c$, 其中 A_i 是后缀表达式。从而 $f(E) = \sum f(A_i) + f(c) = k + (1 - k) = 1$ 。

■

定理 4.3: 对于合法后缀表达式 E 的任意前缀 $E_j = (e_1, e_2, \dots, e_j)$, $j \leq n$, 始终有 $f(E_j) \geq 1$ 。

证明 使用数学归纳法, 对 j 归纳。

1. 由于 e_1 是操作数, 所以 $f(E_1) = 1$ 。
2. 如果 e_j 是操作数, 则 $f(E_j) = f(E_{j-1}) + 1 \geq 1 + 1 > 1$ 。
3. 如果 e_j 是运算符, 根据定理4.1, 必定存在某个 $(e_t, e_{t+1}, \dots, e_j)$, $1 \leq t < j$ 是以 e_j 结尾的后缀表达式。那么根据定理4.2, $f(e_t, e_{t+1}, \dots, e_j) = 1$, 从而 $f(E_j) = f(E_{t-1}) + f(e_t, e_{t+1}, \dots, e_j) \geq f(e_t, e_{t+1}, \dots, e_j) = 1$ 。

■

定理 4.4: 对于以 k 元运算符 c 结尾的后缀表达式 E , 只有唯一的合法划分方式使 E 被划分为 $A_1 A_2 \dots A_k c$ 。

证明 使用反证法证明。

如果有不止一种合法的划分方式, 设两种划分方式中 (A_1, A_2, \dots, A_k) 序号最大的一个不同的项是 A_j 和 A'_j 。不妨设 A_j 的长度大于 A'_j , 那么 A_j 则可以被拆分为两段 $P A'_j$ 。因为 A_j 和 A'_j 都是合法的后缀表达式, 所以由定理4.2, $f(P) = f(A_j) - f(A'_j) = 1 - 1 = 0$ 。但是 P 是 A_j 的前缀, 根据定理4.3应有 $f(P) \geq 1$, 矛盾。

■

因此, 递归地进行唯一合法的划分, 就可以得到最终的中缀表达式。类似地, 您可以得到前缀表达式转换为中缀表达式的手段。

需要说明的是, 上面的转换方式并不实用, 笔者在此处也没有给出相应的示例代码, 仅仅从理论的角度进行介绍。实际转换的时候, 通常利用**表达式树**作为中介进行转换, 表达式树的内容将会在后面的章节介绍。在本章介绍表达式, 是希望您在思维中巩固“栈”和“表达式”之间的联系。

4.4.6 后缀表达式转换为前缀表达式

在上一小节中使用了一个 $f(\cdot)$ 函数用来辅助证明，这个函数并不是凭空产生的，而是另一个栈的产物。我们采用下面的方法构造一个操作序列。

1. 将后缀表达式中的操作数 i , 用 $\vee(i)$ 替代。
2. 将后缀表达式中的运算符 c , 用 $\wedge^k \vee(c)$ 替代, 其中 \wedge^k 表示 k 个连续的 \wedge , k 是这个运算符的运算所需操作数数量。
3. 在整个序列的结尾处增加一个 \wedge 。

根据上一小节的定理4.2和定理4.3, 我们可以确定, 这样得到的操作序列是合法的。

我们发现, 在这个操作序列中, 对应的入栈序列恰好是后缀表达式。自然地, 我们想到要分析它的出栈序列。仍然以前面的 $1\ 2\ 3\ 4\ -\ *\ +$ 作为例子, 它对应 $\vee(1)\ \vee(2)\ \vee(3)\ \vee(4)\ \wedge\ \wedge\ \vee(-)\ \wedge\ \wedge\ \vee(*)\ \wedge\ \wedge\ \vee(+)\ \wedge$, 从而可以得到出栈序列是 $4\ 3\ -\ 2\ *\ 1\ +$ 。这恰好是倒序的前缀表达式。对于一般情形, 您可以用递归(归纳)的方法证明这个结论。对偶地, 您也可以得到前缀表达式转换成后缀表达式的手段。

4.4.7 后缀表达式的计算

后缀表达式的计算是考试中的传统题型之一。要计算后缀表达式, 只需要对上一小节中的操作序列作出一点点修改。

1. 将后缀表达式中的操作数 i , 用 $\vee(i)$ 替代。
2. 将后缀表达式中的运算符 c , 用 $\wedge^k \vee(r)$ 替代, 其中 \wedge^k 表示 k 个连续的 \wedge , k 是这个运算符的运算所需操作数数量。 r 是本次运算的结果。设第 i 个 \wedge 出栈的数为 A_i , 则 $r = c(A_1, A_2, \dots, A_k)$ 。
3. 在整个序列的结尾处增加一个 \wedge 。这次出栈的数就是后缀表达式的计算结果。

这个算法的正确性也可以递归证明。您可以自己实现这一算法的代码, 下面展示了一个示例。

```

1 int calSuffix() const {
2     Stack<int> S;
3     for (auto& e : *this) {
4         if (e.isOperand()) {
5             S.push(e.getOperand());
6         } else {
7             auto [l, r] { e.operandPosition() };
8             int rhs { r ? S.pop() : 0 };
9             int lhs { l ? S.pop() : 0 };
10            S.push(e.apply(lhs, rhs));
11        }
12    }

```

```

13 |     return S.pop();
14 |

```

考试中实际出现的后缀表达式计算题目，可以使用上述算法手工计算。不过和之前一样，笔者推荐使用表达式树而不是栈进行计算。诚然，表达式树的做法比栈要复杂一些；但表达式树的方法更加清晰，更加适合答题结束后的检查，更加不容易出错。

知道后缀表达式如何计算之后，由于之前已经介绍过各种表达式之间互相转换的方法，所以您也就能够写出计算前缀表达式和中缀表达式的算法了。

4.4.8 实验：中缀表达式的计算

在这个实验中将实现中缀表达式的计算，从而实现一个类似计算器的功能。代码可以在 *Expression.cpp* 中找到。

```

1 class CalExpr : public Algorithm<int, const string&> {};

```

根据前面的结论，我们可以先将中缀表达式转换为后缀表达式，再使用后缀表达式计算。

```

1 int operator()(const string& expr) override {
2     Expression e { expr };
3     e.infix2suffix();
4     return e.calSuffix();
5 }

```

因为无论是中缀转后缀，还是后缀求值，都是从左到右依次进行，所以我们也能够将中缀转后缀和后缀求值的过程合并在一起。此时，因为运算符的优先级只能和运算符相比，所以需要使用两个栈，将运算符和表达式拆开。如下所示。

```

1 int calInfix() const {
2     Stack<int> Sr;
3     Stack<ExpressionElement> So;
4     for (auto& e : *this) {
5         if (e.isOperand()) {
6             Sr.push(e.getOperand());
7         } else {
8             while (!So.empty() && So.top().prior(e.getOperator()))
9                 {
10                     if (auto op { So.pop() }; op != '(') {
11                         auto [l, r] { op.operandPosition() };
12                         int rhs { r ? Sr.pop() : 0 };
13                         int lhs { l ? Sr.pop() : 0 };
14                         Sr.push(op.apply(lhs, rhs));
15                     } else {
16                         break;
17                     }
18                 }
19             }
20         }
21     }

```

```

18     if (e != ')') {
19         So.push(e);
20     }
21 }
22
23 return Sr.pop();
24 }
```

我们的实验中提供了一些简单的例子来测试它的正确性，并使用大量的 1 相加来对算法的性能进行简单的测定。直接计算的常数会比通过后缀表达式间接计算低一些，但是差距非常小，几乎可以忽略不计。

如果您使用了合取方法来定义表达式中的元素，则上面的算法可以得到大幅度简化：因为表达式里的每个元素都是运算符，而操作数是附带在前方最近的运算符上的。推荐您进行这样的尝试。此外，引入 k 元运算符也是一个有趣的修改方向。您可能会发现，`operandPosition` 中包含的位置信息并没有实际的作用，事实上我们只需要知道一个运算符具有多少个参数。我们修改这个方法，并将 `apply` 实现为接受向量（而不是数对）的方法，即可兼容任意元运算符。这里需要使用向量而不是可变参数包的原因是，我们无法在编译期知道参数列表的长度。

4.5 栈与递归

栈和递归的关系非常紧密，因为调用递归函数本质上相当于使用了系统栈。具体的原理参见《操作系统》。系统栈的空间是有限的，如果递归层次过多，就会发生栈溢出（stack overflow）错误。为了避免这种情况发生，我们可以通过手写栈将递归改写为迭代形式。本节将介绍使用栈消除递归的方法。

4.5.1 实验：消除尾递归的扩展形式 *

在2.7.2节，我们介绍了消除尾递归的方法。尾递归因为每个递归实例只会在尾部调用一次自身，所以不需要使用栈。在这一小节，我们将介绍一种尾递归的扩展形式：每个递归实例会在尾部调用不止一次自身。这样的递归函数通常没有返回值。

在这一节，我们制造一个没有返回值的场景来对这种情况进行分析。给定 w ，求 w 个数位均为 1、2、3 或 4 的 w 位十进制数的数量。当然，我们知道答案是 4^w ，不过我们希望将所有符合条件的数枚举一遍，放入一个向量中，最后再返回向量的规模。代码可以在 `Generate4.cpp` 中找到。

```

1 class Generate4 : public Algorithm<size_t (size_t)> {
2 protected:
3     Vector<size_t> v;
```

```
4 };
```

我们可以写出这样的递归算法，对所有符合条件的数进行枚举。

```
1 class Generate4Solver : public Algorithm<void(Vector<size_t>&, size_t)> {
2     protected:
3         size_t minn, maxn;
4     public:
5         Generate4Solver() = default;
6         Generate4Solver(size_t w) : minn(Power {}(10, w - 1)), maxn(Power {}(10, w) - 1) {}
7     };
8
9 class Generate4RecursiveSolver : public Generate4Solver {
10 public:
11     void operator()(Vector<size_t>& V, size_t n) override {
12         if (minn <= n && n <= maxn) {
13             V.push_back(n);
14         } else {
15             for (size_t i : {1, 2, 3, 4}) {
16                 (*this)(V, n * 10 + i);
17             }
18         }
19     }
20 }
```

从上述算法中，我们可以很快提取到一些关键信息。

1. 递归边界： $n \in [10^{w-1}, 10^w)$ 。
2. 递归边界上的返回：将 n 加入向量。
3. 非递归边界时的调用： $10n + j$ ，其中 $j = 1, 2, 3, 4$ 。

类似于只调用一次自身的尾递归，我们可以拟造出递归形式的模板。和尾递归的区别主要有两个：(1) 没有返回值；(2) 非递归边界时的调用返回包含多组参数的向量。

```
1 void operator()(Args... args) override {
2     if ((*pred)(args...)) {
3         (*bound)(args...);
4     } else {
5         for (auto&& nextArgs : (*next)(args...)) {
6             apply(*this, nextArgs);
7         }
8     }
9 }
```

在我们的例子中，每个非边界的递归实例会调用 4 次自身（尾递归）。记递归函数为 f ，那么首先调用 $f(10n+1)$ ，完成它的所有递归实例之后再进入 $f(10n+2)$ 。这个过程类似于我们定义了一个 4 元运算符 c ，在计算后缀表达式 $(A_1)(A_2)(A_3)(A_4)c$ ，

其中 A_j 表示 $f(10n + j)$ 。我们可以仿照后缀表达式的计算，在 A_1 全部计算完成之后再让 A_2 开始入栈；但因为我们是从上到下递归计算的，所以我们可以一并将 $A_4 A_3 A_2 A_1$ 倒序入栈，这样， A_1 会成为栈顶， A_1 全部计算完成之后 A_2 成为栈顶，以此类推。下面是一个利用栈的迭代实现的模板，其中特别需要注意采用了倒序遍历。

```

1 void operator()(Args... args) override {
2     Stack<tuple<Args...>> S { { args... } };
3     while (!S.empty()) {
4         tie(args...) = S.pop();
5         if ((*pred)(args...)) {
6             (*bound)(args...);
7         } else {
8             for (auto&& nextArgs : (*next)(args...) | views:::
9                  reverse) {
10                 S.push(move(nextArgs));
11             }
12         }
13     }
}

```

套用模板，可以得到 Generate4 问题的迭代版本。它和前面的递归版本等价。

```

1 void operator()(Vector<size_t>& V, size_t n) override {
2     Stack<size_t> S { n };
3     while (!S.empty()) {
4         n = S.pop();
5         if (minn <= n && n <= maxn) {
6             V.push_back(n);
7         } else {
8             for (size_t i : {4, 3, 2, 1}) {
9                 S.push(n * 10 + i);
10            }
11        }
12    }
}

```

类似于尾递归的情况，使用模板总是比不使用慢。但和尾递归的情况不同，采用栈进行迭代反而会慢于直接递归。同时，采用栈进行迭代也不会降低算法的空间复杂度，因此它通常只作为一种避免栈溢出系统错误的手段，在平时书写代码的时候并不实用。递归的高速得益于编译器的优化；这告诉我们挑战编译器的能力边界，对绝大多数编程人员来说都是极具浪漫主义色彩的冒险行为。

4.5.2 实验：计算组合数 *

对上一节介绍的尾递归扩展形式稍作改动，可以用来计算一些有返回值的递归函数。这些函数在计算完每个递归调用的结果之后，对这些结果进行一个简单

的处理，然后返回。本节以计算组合数的问题作为例子，代码可以在 *Combine.cpp* 中找到。

众所周知，组合数满足递归公式：

$$C_n^m = \begin{cases} 1, & m = 0 \text{ or } m = n, \\ C_{n-1}^{m-1} + C_{n-1}^m, & \text{otherwise} \end{cases}$$

基于这个公式，可以设计下面的算法。

```

1 // CombineRecursive1
2 int operator()(int n, int m) override {
3     if (m == 0 || m == n)
4         return 1;
5     return (*this)(n - 1, m - 1) + (*this)(n - 1, m);
6 }
```

它和上一节介绍的尾递归扩展形式的不同在于，在递归调用两个实例之后，还进行了一次加法。看起来由于加法的存在，这个算法似乎不是尾递归；但是，我们可以通过将加法“吸收”到递归函数内部，将它改写成上一节的形式。如下所示。

```

1 class CombineRecursive2 : public CombineProblem {
2     int sum { 0 };
3     void combine(int n, int m) {
4         if (m == 0 || m == n) {
5             ++sum;
6         } else {
7             combine(n - 1, m - 1);
8             combine(n - 1, m);
9         }
10    }
11 public:
12     int operator()(int n, int m) override {
13         sum = 0;
14         combine(n, m);
15         return sum;
16     }
17 };
```

接着，我们可以使用上一节的模板，将其改写为使用栈进行迭代的形式。

```

1 int operator()(int n, int m) override {
2     int sum { 0 };
3     Stack<pair<int, int>> S { {n, m} };
4     while (!S.empty()) {
5         auto [n, m] { S.pop() };
6         if (m == 0 || m == n) {
7             ++sum;
8         } else {
9             S.push({ n - 1, m - 1 });
10            S.push({ n - 1, m });
11        }
12    }
13 }
```

```

12     }
13     return sum;
14 }
```

需要指出的是，这个方法是极其低效的，因为 sum 只能不断加一，所以该方法的时间复杂度高达 $\Theta(C_n^m)$ 。其中，使用栈的方法将比递归更加低效。为了提高时间效率，我们可以利用之前介绍过的不必要工作对算法的过程进行检查。我们可以发现，在这个计算过程中包含了大量的重复工作，比如， $C_n^m = C_{n-1}^{m-1} + C_{n-1}^m = C_{n-2}^{m-2} + 2C_{n-2}^{m-1} + C_{n-2}^m$ ，因而 C_{n-2}^{m-1} 被计算了 2 次。为了解决这个问题，我们可以利用 **记忆化搜索** (memory search) 的技术，将我们已经计算过的结果储存起来，如下所示。

```

1 class CombineMemorySearch : public CombineProblem {
2     vector<vector<int>> C;
3     void initialize(size_t n) {
4         C.resize(n + 1);
5         for (size_t i = 0; i <= n; ++i) {
6             C[i].resize(i + 1, 0);
7             C[i][0] = C[i][i] = 1;
8         }
9     }
10 public:
11     int operator()(int n, int m) override {
12         if (n >= C.size())
13             initialize(n);
14         if (C[n][m] == 0) {
15             C[n][m] = (*this)(n - 1, m - 1) + (*this)(n - 1, m);
16         }
17         return C[n][m];
18     }
19 }
```

上述算法的空间复杂度为 $\Theta(n^2)$ ；在不考虑初始化的情况下，最坏时间复杂度为 $\Theta(m(n-m))$ 。它的好处是在计算 C_n^m 的时候可以得到很多中间结果，当我们反复调用它的时候，这些储存下来的中间结果可以得到复用，从而降低了分摊的时间消耗。如果我们只需要计算 C_n^m 这一个数，有更快的计算方法，那就是直接利用 $C_n^m = \frac{(n+m)!}{n!m!}$ 这个公式：这个公式一共只需要进行 $\Theta(m)$ 次计算（上下的 $n!$ 可以被约去），从而我们可以做到 $O(1)$ 的空间复杂度和 $\Theta(m)$ 的时间复杂度。您可以自己完成这个算法。

测试表明，Recursive2 的效率会明显高于 Recursive1，这说明了尾递归在编译器优化上的优势。当我们有机会使用尾递归的时候，应当尽可能使用尾递归。

4.5.3 实验：消除一般的递归 *

在前面两节中，我们从尾递归过渡到了它的扩展形式，允许函数进行多次递归调用。对于一般的递归而言，这多次递归调用不一定出现在尾部，递归调用之间可以穿插其他代码。我们可以将一般的递归概括为下面的形式。

```

1 R operator()(Args... args) override {
2     if ((*pred)(args...)) {
3         return (*bound)(args...);
4     } else {
5         vector<R> V {};
6         while ((*hasnext)(V, args...)) {
7             V.push_back(apply(*this, (*next)(V, args...)));
8         }
9         return (*finalize)(V, args...);
10    }
11 }
```

由于递归调用依赖于此前的递归调用结果，所以在这种情况下，我们没法一次性地将所有递归调用的参数入栈，需要将用来存储已有递归调用结果的向量 V 也缓存在栈内。

```

1 R operator()(Args... args) override {
2     vector<R> V {};
3     Stack<tuple<vector<R>, Args...>> S { { {} }, { {}, args... }, { {} },
4         args... };
5     while (S.size() > 1) {
6         tie(V, args...) = S.top();
7         optional<R> r {};
8         if ((*pred)(args...)) {
9             r = (*bound)(args...);
10        } else if ((*hasnext)(V, args...)) {
11            tie(args...) = (*next)(V, args...);
12            S.push({ {}, args... });
13        } else {
14            r = (*finalize)(V, args...);
15        }
16        if (r.has_value()) {
17            S.pop();
18            get<0>(S.top()).push_back(r.value());
19        }
20    }
21    return get<0>(S.top())[0];
22 }
```

这里在栈中预先存放了一个空向量，用来存储最终返回的值。上述形式肉眼可见地低效，仅仅作为考试中需要改写为迭代形式而没有找到合适方法时，保底的通用解法。您可以在 *Generate4Sum.cpp* 中看到一个使用例子。

4.6 栈的扩展

4.6.1 共享栈

当我们使用顺序栈的时候，我们为栈申请了一片连续内存作为存储空间。注意到，我们的栈只使用了前半部分的空间，而没有使用后半部分的空间。有一种高效利用空间的方法称为**共享栈**（shared stack），它由两个共享同一片连续内存的栈组成。其中一个栈使用前半部分，以秩为 0 的元素为栈底，以秩最大的元素为栈顶；另一个栈使用后半部分，以秩为 $n - 1$ 的元素为栈底，以秩最小的元素为栈顶。两个栈的栈顶“相向而行”。由于现在往往不需要如此精打细算空间消耗，现在已经很少看到这个数据结构，邓书也不介绍它。您可以将此作为栈的一个练习自己实现，并使用 *SharedStackTest.cpp* 测试。

我们使用向量 V （而不是数组）作为实现共享栈的基础，以提供变长特性。使用两个变量 topf 和 topb 分别表示前向栈（栈顶向秩大的方向移动）和后向栈（栈顶向秩小的方向移动）的栈顶位置。在我们的设计中，这两个变量均代表“下一个”的值，比如前向栈的栈顶元素是 $V[2]$ 时， topf 的值是 3，后向栈的栈顶元素是 $V[2]$ 时， topb 的值是 1。您也可以记录栈顶元素本身的秩。

接下来，我们利用类的嵌套，具体构造两个栈。这里展示了后向栈的设计，前向栈与其大体相同。

```

1 class BackwardStack : public AbstractStack<T> {
2     SharedStack& S;
3 public:
4     BackwardStack(SharedStack& s) : S(s) {}
5     void push(const T& e) override {
6         S.V[S.m_topb--] = e;
7     }
8     T& top() override {
9         return S.V[S.m_topb + 1];
10    }
11    T pop() override {
12        return std::move(S.V[++S.m_topb]);
13    }
14    size_t size() const override {
15        return S.V.size() - 1 - S.m_topb;
16    }
17 }
```

唯一需要注意的是，在进行入栈操作的时候，除了上面的基本操作，还需要进行一次共享栈是否已满（是否已经接触到另一个栈的栈顶）的判定。如果共享栈已满，则需要进行扩容。我们可以使用向量本身的扩容，然后将后向栈移动到向量尾部的位置。

```

1 void expand() {
```

```

2     size_t oldsize { V.size() };
3     V.resize(std::max(V.capacity() + 1, V.capacity() * 2));
4     std::move_backward(V.begin() + m_topb + 1, V.begin() +
5         oldsize, V.end());
6     m_topb += V.size() - oldsize;
}

```

为了和您自己实现的向量类兼容，此处没有对向量自身的扩容策略进行萃取，而是采用了固定的2倍扩容策略，您可以将其改为自己的扩容策略，或者改为萃取的形式。

经典的向量中，空闲的元素位于内存的尾部，而共享栈中，空闲的元素位于内存的中部，因此不适用向量本身的规模机制。您可以认为这里的两个指针`topf`和`topb`实现了和向量中的`size`同样的内存管理功能。

4.6.2 最小栈 *

栈对于用户可以访问的位置做出了很强的限制，有时我们希望略微放开这些限制，使得用户可以访问到一些更多的信息。**最小栈** (minimum stack) 是其中的一个典型的例子。在最小栈中除了栈本身所支持的三种基本操作之外，还支持`min`操作，返回当前栈内元素的最小值。要求在栈的三种基本操作的分摊复杂度保持 $O(1)$ 的前提下，新增的`min`操作的分摊复杂度也为 $O(1)$ 。这里假定比较两个元素的操作时间复杂度为 $O(1)$ 。您自己实现的最小栈可以使用 *MinStackTest.cpp* 测试。

一个基本的思想是维护一个变量 m 来存储栈中的最小值。当插入的元素 $e < m$ 的时候更新 m 没有问题，但当删除的元素 $e = m$ 的时候，我们既不知道这个 e 是否是当前栈中唯一的最小值 m ，又不知道当唯一最小值被删除之后，新的最小值是多少。这就导致如果我们进行连续的删除操作，则需要反复地遍历整个栈来寻找新的最小值。这在两个方面不能达到我们的要求。

1. 时间复杂度上会存在问题。弹出元素的时候，时间复杂度高达 $\Theta(n)$ ，而无法满足我们需求的 $O(1)$ 。
2. 栈的性质上会存在问题。我们不应当访问栈顶以外的其他元素，即使在栈的实现内部也是如此。

为了在弹出元素的时候，能够立刻找到新的最小值，我们想到用空间换时间的方法。我们定义一个辅助线性表 S_m ，规定 $S_m[k] = \min(S[0], S[1], \dots, S[k])$ 。因为栈 S 只能在栈顶进行操作，而 $S_m[k] = \min(S_m[k-1], S[k])$ ，所以 S_m 也只会在尾部进行操作，它同样是一个栈。

```

1 template <typename T>
2 class MinStack : public Stack<T> {
3     Stack<T> minStack;

```

```

4 public:
5     void push(const T& e) override {
6         Stack<T>::push(e);
7         if (minStack.empty()) {
8             minStack.push(this->top());
9         } else if (auto t { minStack.top() } ; t < this->top()) {
10            minStack.push(t);
11        } else {
12            minStack.push(this->top());
13        }
14    }
15    T pop() override {
16        minStack.pop();
17        return Stack<T>::pop();
18    }
19    const T& min() const {
20        return minStack.top();
21    }
22 };

```

上面的实现方式需要 $\Theta(n)$ 的额外空间。我们可以发现，在辅助栈中存在大量重复的元素。比如，在栈 $S = [3, 4, 5, 1, 2]$ 时，辅助栈 $S_m = [3, 3, 3, 1, 1]$ ，其中 3 被重复了 3 次，1 被重复了 2 次。因此，直觉上看存在空间复杂度并非最优的可能性。为了判断是否已经取得了最优的空间复杂度，我们需要对辅助栈进行分析。考虑一个略简单的模型，即栈内的元素互不相等的情况：设栈中的元素为 $1, 2, \dots, n$ 。我们对辅助栈和栈操作序列建立一个一一对应。

1. 补充定义 $S_m[-1] = n + 1$ 。对于 $S[j]$ ，如果 $S[j] \neq S[j - 1]$ ，则将 $S[j]$ 替换为 $\vee^k \wedge$ ，其中 $k = S[j - 1] - S[j]$ 。
2. 如果 $S[j] = S[j - 1]$ ，则将 $S[j]$ 替换为 \wedge 。

请您自己证明上面的这个映射得到的操作序列是合法的。因此，辅助栈的所有可能情况数量为 $\text{Catalan}(n)$ 。从信息论的角度看，我们至少需要 $\log(\text{Catalan}(n))$ 的空间才能表示这些情况。又因为

$$\log(\text{Catalan}(n)) \sim 2n \log\left(\frac{2n}{e}\right) - 2n \log\left(\frac{n}{e}\right) \sim 2n$$

当允许元素相等的时候，情况只会更多，需要的空间也只会更多。因此，无论采用如何的方式，只要存储了辅助栈，那么在最坏情况下需要的额外空间复杂度总是 $\Theta(n)$ 。

下面将介绍另一种最小栈的实现方法。该方法极具迷惑性，它看起来空间复杂度为 $O(1)$ ，但实际上仍然为 $\Theta(n)$ 。尚不清楚此类方法是否在考试中被认为是 $O(1)$ 方法。

根据之前的分析，如果采用辅助栈 S_m 来存储每个前缀的最小值，则需要至少

$\Theta(n)$ 的空间。因此，如果我们想要降低空间复杂度，就必须抛弃辅助栈的设计。我们观察辅助栈和原栈之间的联系：如果辅助栈在第 k 个位置发生了变化 $x \rightarrow y$ ，也就意味着原栈的第 k 个位置插入了 y ，并且 $y < x$ 。现在我们不想要辅助栈，也就是说我们需要借助原栈的信息从 y 还原到 x 。

一个直接的想法是，在原栈的每个位置上存数据对 $(x, S[k])$ ，其中 $S[k]$ 是栈在该位置上的实际元素， x 则是该位置被弹出之后的新的最小值。采用一个额外变量 y 来维护当前的最小值，当 $(x, S[k])$ 被弹出时，令 $y = x$ 。当然，我们知道这个方法和最小栈实质上完全一样，同样需要 $\Theta(n)$ 的辅助空间。

于是我们想到对这个数据对进行变形，改为存一个映射的象 $z = f(x, S[k])$ 。那么，我们需要在已知 y 和 z 的情况下，能够还原出 x 和 $S[k]$ 。如果仅凭 z 还原，就是上面的数据对思路， z 的信息量是 x 和 $S[k]$ 的和，需要 $\Theta(n)$ 的辅助空间。所以我们会想到， y 也能提供一部分信息， z 中需要存储的信息量可以下降。于是，我们可以把这个映射表述为 $(y, z) = \mathbf{F}(x, S[k])$ ，其中 y 的语义是确定的，即 $y = \min(x, S[k])$ 。

直观上看， \mathbf{F} 输入两个值，输出两个值，那么可以让 $x, S[k]$ 的信息完全被包含在 y, z 中；但事实并非如此。由于 $y = \min(x, S[k])$ 打破了对称性，导致 y 包含的信息量降低了，从而使得 z 需要包含更多的信息量。假设 $x, S[k]$ 都是某个 m 元集中，等概率随机选取的元素，那么 $(x, S[k])$ 的信息熵为 $2 \log m$ ，而 $y = \min(x, S[k])$ 的信息熵为

$$\begin{aligned} H(y) &= - \sum_{k=1}^m \frac{2k-1}{m^2} \log \frac{2k-1}{m^2} = \log n - \sum_{k=1}^m \frac{2k-1}{m^2} \log \frac{2k-1}{m} \\ &= \log m - \frac{1}{2} \sum_{k=1}^m \frac{2}{m} \cdot \frac{2k-1}{m} \log \frac{2k-1}{m} = \log m - \frac{1}{2} \int_0^2 x \log x dx + o(1) \\ &= \log m - \left(1 - \frac{1}{2 \ln 2}\right) + o(1) \end{aligned}$$

上述计算结果表明， $H(y)$ 和 $\log m$ 之间存在 $\theta = 1 - \frac{1}{2 \ln 2} \approx 0.28$ 的差距，这使得每个 z 需要额外的至少 0.28 比特来存储这些损失的信息，使用和 $S[k]$ 一样的空间是做不到的。

我们知道 $y = \min(x, S[k])$ ，一个非常简单的想法就是令 $z = \max(x, S[k])$ 。这样，我们通过 (y, z) 就能确定 $\{x, S[k]\}$ ，只是分不清楚哪个是 x 、哪个是 $S[k]$ 。只要增加 1 个比特（通常称为**标志位**）来指示这一点，就可以实现逆运算 $(x, S[k]) = \mathbf{F}^{-1}(y, z)$ 。

作为举例，如果我们保证栈内的所有元素都非负，就可以让符号位作为标志

位，即：

$$z = \begin{cases} S[k], & S[k] \geq x, \\ -x, & S[k] < x \end{cases}$$

此时还原的方法为：

$$\begin{cases} x &= |\min(y, z)|, \\ S[k] &= \max(y, z) \end{cases}$$

在实际软件开发中，带符号整数的比特数通常只能取 16、32、64 等标准位宽以做到字节对齐；所以通常都允许我们划出一些比特来存储额外的信息。但从根本上来说，这种方法仍然具有 $\Theta(n)$ 的空间开销（每个元素包含了 1 个标志位）。

您可能会注意到，在前面使用信息论方法说明辅助栈需要的空间时，空间复杂度的常数至少为 2；但上述的标志位方法常数为 1。这并不是突破了信息论的限制，而是因为辅助栈中的一些信息是包含在原栈里的。比如，在知道原栈的情况下， $S_m = [3, 3, 3, 1, 1]$ 和 $S_m = [2, 2, 2, 1, 1]$ 可以以相同的方式表示，因为二者都是 3 个 $S[0]$ 和 2 个 $S[3]$ ，这降低了描述辅助栈需要的信息量。上述表示方式的数量，相当于将 n 个元素划分成任意多的段的划分数量，也就是 2^{n-1} ，因此至少需要 $\log(2^{n-1}) = n - 1$ 个比特进行刻画；这正好对应了 z_1, z_2, \dots, z_{n-1} 的标志位。 z_0 的标志位没有包含任何信息，因为只有一个元素的情况下，最小值必然是它本身。

4.7 本章小结

作为适配器的栈本身非常简单，无论采用顺序栈还是链栈进行实现都没有难度。初学者可能会好奇，栈能做到的事情线性表（向量或列表）都能做到，为什么要自己给自己设限制呢？这个问题的答案，和 OOP 中将一些方法设置为私有的原因是一样的。通过加以限制，我们的目光被聚焦了，更容易抓住解决问题的要点。

栈的操作很简单，这也就意味着，将其他复杂的问题，比如括号匹配、表达式计算等转化为栈的问题，可以让问题得到大幅的简化。本章的主要学习目标如下。

1. 您理解了栈操作序列的性质，在遇到相似性质的场景时可以联想到用栈解决。
2. 您了解到出栈序列的计数是卡特兰数，在遇到相似的递归方程时可以联想到用栈解决。
3. 您掌握了利用栈将任意递归改写为迭代的技术。

第5章 队列

5.1 先入先出

和栈相比，队列就要简单的多。栈是一个后入先出（LIFO）的数据结构，而队列（queue）是一个先入先出（FIFO，First In First Out）的数据结构。栈有一个不可操作的盲端，而插入、删除、访问三种操作都在自由端；而队列的两端都可以操作，但只能从其中一端插入，而只能从另一端删除或访问。允许删除（出队，`dequeue`）或访问的一端称为队头（front），允许插入（入队，`enqueue`）的一端称为队尾（rear），如图5.1所示。类似于栈，您很容易自己实现一个队列，并使用 `QueueTest.cpp` 进行测试。

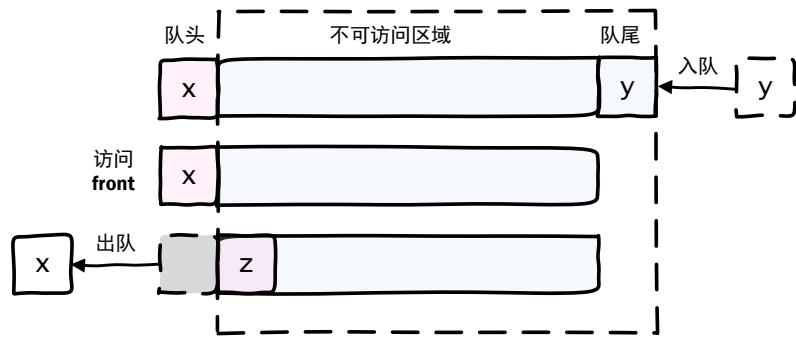


图 5.1 队列的入队、访问、出队

从“队列”这个名字就可以看出来，它非常适合用来模拟奶茶店窗口前排队这样的场景。新过来买奶茶的人，总是会排到队伍的末尾；而只有队伍最前面的人可以买到奶茶，并在买到以后心满意足地离开队伍。

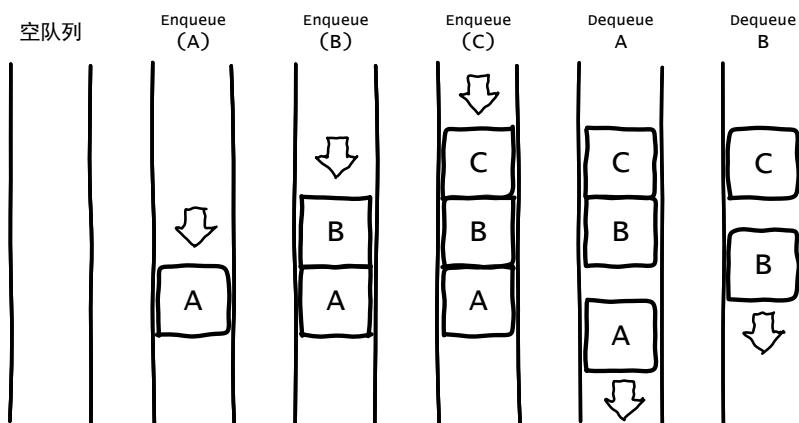


图 5.2 队列的桶式表示

和栈一样，队列也可以用桶式图来分析，只不过队列用的是漏桶，如图5.2所示。从图中的例子中可以看出，如果将 A 、 B 、 C 三个元素依次加入队列，那么为了访问 B ，必须要将 B 前面的元素 A 先出队，才能让 B 也出队。因为没有盲端，所以队列没法简单地写成 $Q[0:n]$ ，而是应该写成 $Q[d:n]$ ，其中 d 是已经出队的元素数量，而队列的规模为 $n - d$ 。

1. 队列的插入就是在 $Q[n - 1]$ 后面插入新的元素。
2. 队列的访问就是取 $Q[d]$ 。
3. 队列的删除就是将 $Q[d]$ 从队列中删除。

```

1 template <typename T>
2 class AbstractQueue : public DataStructure<T> {
3 public:
4     virtual void enqueue(const T& e) = 0;
5     virtual T dequeue() = 0;
6     virtual T& front() = 0;
7 };

```

5.2 队列的结构

5.2.1 链式队列

和栈一样，因为列表的时间和空间效率都很低，所以我们倾向于使用向量来实现队列（顺序队列）。不过，基于列表的队列相对来说，实现起来比较简单，因此邓书中选择以基于列表的队列（链式队列）为例进行讲解，本书也首先讨论这种队列。链式队列相对顺序队列的优点主要是稳定性，它保证了入队、出队操作都是 $O(1)$ 的。顺序队列由于向量扩容的存在，入队只能保证分摊复杂度是 $O(1)$ 的。

```

1 template <typename T, template<typename> typename Linear =
2     ForwardList>
3     requires std::is_base_of_v<AbstractLinearList<T, typename
4         Linear<T>::position_type>, Linear<T>>
5 class LinkedQueue : public AbstractQueue<T> {
6     Linear<T> L;
7 public:
8     void enqueue(const T& e) {
9         L.push_back(e);
10    }
11    T dequeue() {
12        return L.pop_front();
13    }
14    T& front() {
15        return L.front();
16    }
17 };

```

5.2.2 整体搬移 *

由于队列在两端都可以操作，所以上一小节中的代码不能被简单复用到顺序队列中。对于向量 $V[0:n]$ ，如果我们选择让 0 成为队头、 n 成为队尾，那么每次出队的时候都需要进行一次 $\Theta(n)$ 的移动。当连续出队的时候，就会和2.5.7节的情况一样，产生大量的不到位工作。反之，如果我们选择让 0 成为队尾、 n 成为队头，那么每次入队的时候都需要进行一次 $\Theta(n)$ 的移动。当连续入队的时候，同样会产生大量的不到位工作。这两种方案是具有对称性的，本书采用 0 成为队头、 n 成为队尾的方法，和5.1节所定义的 $Q[d:n]$ 保持一致。

为了减少不到位工作，我们需要降低移动的频率。在出队的时候，我们不立刻进行移动，而是设置一个额外的成员变量来存储队头 d 的位置，让队头 d 指向下一个元素，进行逻辑上的删除。在合适的时机，再进行一次整体搬移，对已出队的元素进行物理上的删除。上述设计的代码框架如下所示，在这个示例代码中暂时没有引入整体搬移。

```

1 template <typename T, template<typename> typename Vec =
2     DefaultVector>
3     requires std::is_base_of_v<AbstractVector<T>, Vec<T>>
4 class Queue : public AbstractQueue<T> {
5     Vec<T> V;
6     Rank m_front;
7 public:
8     size_t size() const override {
9         return V.size() - m_front;
10    }
11    void enqueue(const T& e) override {
12        V.push_back(e);
13    }
14    T dequeue() override {
15        return std::move(V[m_front++]);
16    }
17    T& front() override {
18        return V[m_front];
19    }
}

```

下面讨论在何种时候进行整体搬移。一个比较自然的想法是**贪心** (greedy)。在不触发整体搬移的情况下，上述代码中的出队方法`dequeue`的时间开销非常低。因此，我们希望除非必要，否则不进行整体搬移。

所谓“必要”，就是队列必须要处理内存问题的时候。在上面这种实现中， $Q[d:n]$ 实际是建立在 $V[0:n]$ 的基础上的。当向量的规模 n 达到容量 m 的时候，向量达到满的状态。当向量满的时候，如果要继续往队列里插入元素，则必须要处理内存问题：要么对向量进行扩容，要么对队列进行整体搬移，把已出队元素占用的 d 个

存储单元空出来，如图5.3所示。



图 5.3 顺序队列假满时的搬移和扩容

- 如果队列的元素数量 $n - d$ 没有达到 m (即 $d > 0$)，则可以称为“假满”。在这种情况下，可以选择对向量进行扩容，也可以不对向量进行扩容，改为将 $Q[d:n]$ 中的元素整体搬到新的向量 $V[0:n-d]$ 中，从而空出 d 个元素的空间。

```

1 void moveElements() {
2     std::move(std::begin(V) + m_front, std::end(V), std::
3         begin(V));
4     V.resize(V.size() - m_front);
5     m_front = 0;
}

```

- 如果队列的元素数量 $n - d$ 也达到了 m (即 $d = 0$)，则可以称为“真满”。在这种情况下，只能对向量进行扩容。

当队列发生“假满”时，是选择对向量进行扩容，还是将元素进行搬移，这是一个值得思考的问题。如果每次都选择扩容，那么就没有办法重新利用起 $V[0:d]$ 这些已经出队的元素所占用的空间。那么当出队的元素非常多时，空间利用率就会非常低。对于队列 $Q[d:n]$ 来说，如果 $d \approx n$ ，则几乎所有的空间都被已出队元素浪费了。只要不断地入队、出队，就会最终发生内存不够用的情况。

但是，如果每次都选择搬移，则会在时间上出现问题。考虑 $d = 1$ 的情况，进行搬移需要移动 $n - 1 = \Theta(n)$ 个元素。那么如图5.4所示，当入队、出队交错进行时，队列不断在“假满”和“真满”之间“反复横跳”，每次入队-出队的连续操作。因此，对于入队和出队的连续操作序列，每次操作的分摊时间复杂度高达 $\Theta(n)$ ，这是我们不能容忍的。图5.4所示的情形，通过不断变换状态（“假满”和“真满”），从而连续触发开销较大的操作（整体搬移），导致连续操作下的分摊时间开销较大。这种情形被称为抖动（thrashing）。抖动现象影响的是分摊时间，而不一定会影响分摊时间复杂度，因为常数的巨大上升对于系统性能来说同样致命。这一现象

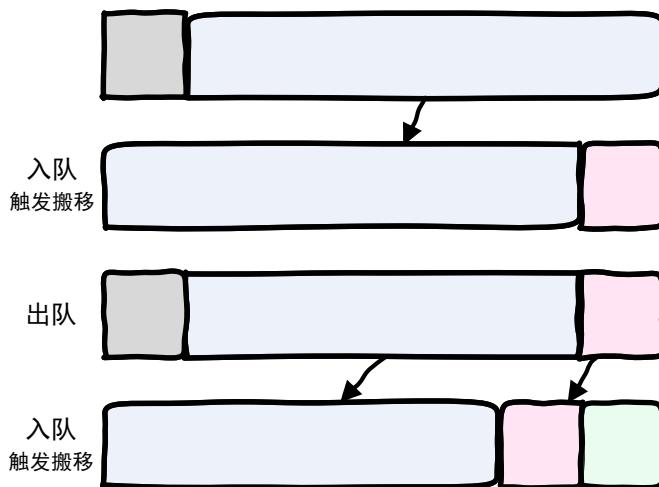


图 5.4 顺序队列连续搬移产生的抖动

在《组成原理》和《操作系统》中还会遇到，如 Cache 抖动、页面抖动等。

空间和时间的权衡，是我们在设计含有内存管理的数据结构时，所必须要处理的问题。在2.4.5节介绍向量的扩容方式时，我们也曾经遇到过这样的抉择。对于连续插入的操作，等差扩容可以达到更高的装填因子（接近 100%），但是分摊时间复杂度高达 $\Theta(n)$ 。等比扩容的装填因子较低（以 2 倍扩容为例，最低为 50%），但是分摊时间复杂度仅为 $O(1)$ 。由于二者的时间差距非常巨大，空间差距则是有限的，所以我们总是使用等比扩容，而不是使用等差扩容。同样地，我们不允许队列操作的分摊时间复杂度因为抖动现象而变为 $\Theta(n)$ ，所以每次都搬移的做法肯定不予考虑。但在队列这个场景下，如果完全不进行搬移，则浪费的空间会无休止地越来越多，这也是我们无法接受的。因此，我们需要选择在合适的时机进行搬移。

在本书的示例代码中，选择规定一个阈值 $\theta \in (0, 1]$ 。在入队时，如果 $\frac{d}{m} \geq \theta$ ，则进行整体搬移；否则进行扩容。特别地， $\theta = 1$ 代表禁止搬移，当且仅当 $d = n = m$ 即空队列假满的情况下，才将 d 和 n 复位（reset）到 0。您可以自己证明，对于入队和出队操作组成的操作序列，任意大于 0 的阈值 θ ，都可以使元素搬移的分摊复杂度只有 $O(1)$ 。

示例代码并没有选择在假满时才进行搬移，这样可以在入队的时候减少一次是否 $n = m$ 的判断，代价则是搬移的次数有可能更多。需要指出，队列这个数据结构只规定了 FIFO 这一个特性，如何在空间和时间中间做权衡，属于队列的具体实现。我们可以采用示例代码中的方法，在特定的条件下进行搬移，也可以在总是搬移、从不搬移、或者在其他的条件下进行搬移。根据具体实现的不同，队列的时间效率和空间效率可能会有所不同。若无特殊说明，通常要求队列的入队、出队分摊时间复杂度为 $O(1)$ 。您可以在这个要求的基础上，设计自己的搬移策略（比如，为搬移增加假满的条件），并在后面的实验中和示例代码进行性能比较。

5.2.3 循环队列

如上一节所述，队列的具体实现有许多种，而利用整体搬移的技巧提高空间利用率的方法只是其中一类。除此之外，我们还可以利用其它方法对空出来的 $V[0:d]$ 这部分空间进行重复利用。如图所示，当发生“假满”现象时，我们不进行搬移或扩容，而是从逻辑上将空出来的 $V[0:d]$ 接在 $V[d:m]$ 后面。这种逻辑上的移动，可以通过将“逻辑秩”对向量的规模 m 取模实现。通过将下标 $\%m$ ，逻辑上的 $V[m:m+d]$ 可以被映射到实际上的 $V[0:d]$ ，从而循环利用已出队元素留下的 $V[0:d]$ 空间。当队列首尾相接，即队列中恰好有 m 个元素时，变为“真满”状态，此时再入队需要对向量进行扩容。这一循环利用的思想指导下建立的队列就称为循环队列（circular queue）。

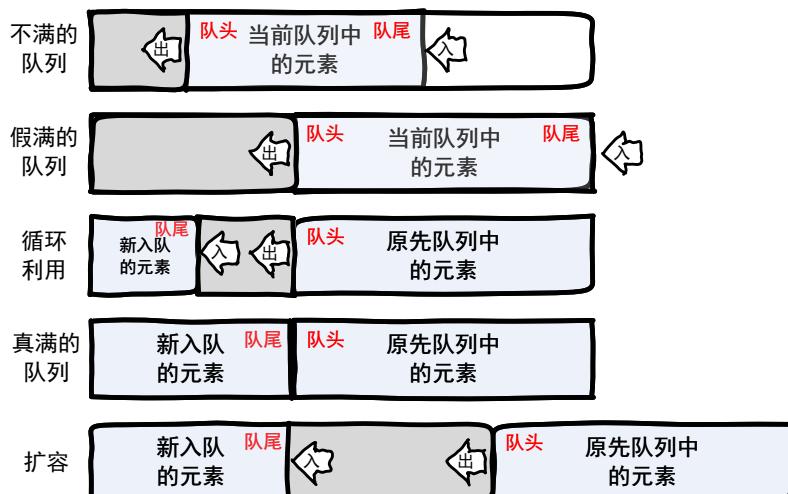


图 5.5 循环队列在假满和真满时的处理方法

在很多实际应用中，循环队列是不允许扩容或缩容的。这是因为在很多应用场景下，循环队列的容量 m 具有一定的实际意义，比如对应一定的硬件资源数量，或者对应一定的时间片数量；这通常是系统设计时确定的参数，不是运行时能轻易改变的。本书主要介绍的是软件实现的循环队列，所以仍然让它基于向量实现，当“真满”时会引发扩容。

循环队列有多种实现，我们首先展示示例代码使用的实现方法，然后说明它相对于其他方法的优点和缺点。在示例代码中，我们将存储队列的队头 d 和队长 $n - d$ （在模 m 意义下）。

```

1 template <typename T, template<typename> typename Vec =
2     DefaultVector>
3     requires std::is_base_of_v<AbstractVector<T>, Vec<T>>
4     class CircularQueue : public AbstractQueue<T> {
5         Vec<T> V;
6         size_t m_front = 0;

```

```

6     size_t m_size = 0;
7     bool full() {
8         return m_size == V.capacity();
9     }
10    void expandQueue() {
11        V.resize(std::max(V.capacity() + 1, V.capacity() * 2));
12        std::move_backward(std::begin(V) + m_front, std::begin(V)
13                           ) + m_front + m_size, std::end(V));
14    }
15    public:
16        void enqueue(const T& e) override {
17            if (full()) {
18                expandQueue();
19                V[(m_front + m_size) % V.size()] = e;
20                ++m_size;
21            }
22            T e { std::move(V[m_front]) };
23            m_front = (m_front + 1) % V.size();
24            --m_size;
25            return e;
26        }
27        T& front() override {
28            return V[m_front];
29        }
30    };
31

```

您可以发现，这里使用的判满和扩容方法，和前面实现共享栈时完全相同，在向量扩容的时候需要进行额外的一次搬移。除了像图5.5那样，将后半段 $V[d:n]$ 搬移到 $V[m-n+d:m]$ 的位置上外，如果扩容的倍数至少为 2 倍，还可以选择将 $V[0:d]$ 搬移到 $V[n:n+d]$ 的位置上，请您自己实现这种方法。

下面介绍几种其他的存储方式，您可以自行用这些方法实现循环队列。

1. 存储队头 d 和队尾 n 。它的优点在于在入队的时候，可以少算一次加法；缺点在于无法分辨空队和满队（在这两种情况下， d 和 n 都相等）。为了解决这个问题，可以改为在队列“差一个元素满”的时候就扩容，从而保证永远不会出现满队的情况。
2. 存储队头 d 和队尾 n ，以及一个空队标志。通过引入一个额外的空队标志来分辨空队和满队。这种方法更适用于硬件设计等底层应用场景。在硬件循环队列中，每个数据单元会非常大，而队列容量比较小（比如，每个队列元素是一个 I/O 缓冲区）且不能扩容，那么像上面一样“浪费一个位置”是非常奢侈的行为，而一个空队标志引入的额外组合逻辑电路是可以被接受的，它的延迟少于示例代码中引入的加法器延迟。

5.2.4 双栈队列 *

从逻辑结构上思考，队列的 FIFO 特性决定了，它的输出和输入次序是相同的，而栈的 LIFO 特性决定了，它的输出和输入次序是相反的（对于连续的输入和输出而言）。负负得正，我们自然地会想到，可以用两个栈来实现一个队列。

从存储结构上思考，在上一节介绍的循环队列中，我们发现队列可以从逻辑上拆分为两段。如图5.5所示，一段是 $V[0:n \% m]$ (队尾段)，一段是 $V[d \% m:m]$ (队头段)，待利用的内存空间（灰色部分）在两段之间。这种形式和共享栈高度相似。

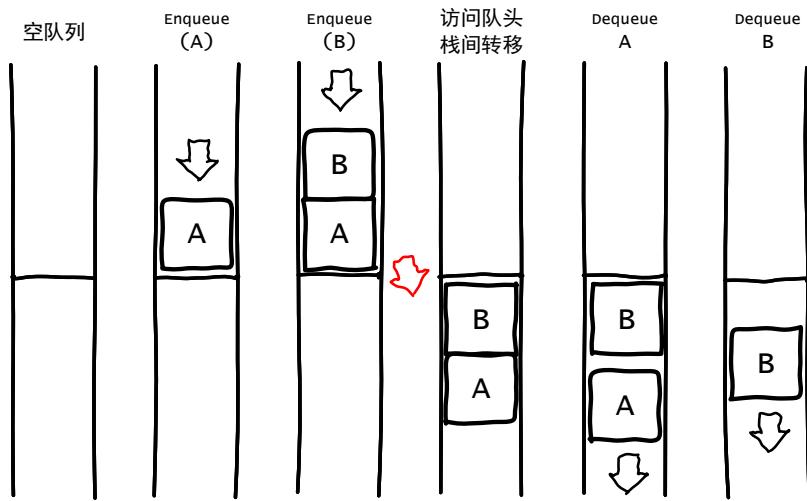


图 5.6 双栈队列的桶式表示

上面两个思路都提示我们可以基于两个栈来实现队列（**双栈队列**）。特别地，如果这两个栈以共享栈的形式储存，则双栈队列可以和前面两节一样只基于一个向量，从而节约了空间。在双栈队列中，两个栈分别作为队尾栈和队头栈，队尾栈的自由端是队尾，队头栈的自由端则是队头。如图5.6所示，我们可以将这两个栈表示为两个盲端相接的桶，此时就如同在5.2中的漏桶里加上了一块“隔板”。

当我们需要入队的时候，直接在队尾栈推入；当我们需要出队的时候，直接在队头栈弹出。需要进行特别处理的情况，是队尾栈非空而队头栈空的情况。此时，逻辑上的队头在队尾栈的盲端，我们需要把隔板抽掉，让队尾栈里的元素“掉下来”。容易发现，这一操作只需要将队尾栈的元素依次弹出，并推入到队头栈即可。弹出和访问队头的时候，都可能触发上述栈间转移。下面是一个实现双栈队列的示例。

```

1 template <typename T, template<typename> typename Vec =
2     DefaultVector>
3     requires std::is_base_of_v<AbstractVector<T>, Vec<T>>
4 class StackQueue : public AbstractQueue<T> {

```

```

4     SharedStack<T, Vec> SS;
5     auto& backStack() {
6         return SS.getStacks().first;
7     }
8     auto& frontStack() {
9         return SS.getStacks().second;
10    }
11 public:
12     void enqueue(const T& e) override {
13         backStack().push(e);
14     }
15     T dequeue() override {
16         if (frontStack().empty()) {
17             while (!backStack().empty()) {
18                 frontStack().push(backStack().pop());
19             }
20         }
21         return frontStack().pop();
22     }
23     T& front() override {
24         if (frontStack().empty()) {
25             while (!backStack().empty()) {
26                 frontStack().push(backStack().pop());
27             }
28         }
29         return frontStack().top();
30     }
31     size_t size() const override {
32         return SS.size();
33     }
34 };

```

和整体搬移、循环利用相对应，双栈队列处理空间利用率问题的方案是栈间转移。因为栈间转移是单向的，只可能从队尾栈转移到队头栈，所以任何一个元素在入队和出队之间恰好经历过一次栈间转移。这也意味着栈间转移的分摊时间复杂度是 $O(1)$ 的。

需要特别指出，本节介绍的整体搬移、循环利用、栈间转移和链式队列（节点直接释放），是解决队列空间效率问题的不同解决方案。循环队列和循环列表毫不相干，也不存在“链式循环队列”这一说。

5.2.5 实验：队列的性能对比 *

这一节通过一个实验讨论各种队列实现方式的性能，代码可以在 *QueueEvaluation.cpp* 中找到。我们构造了两个场景，一个是连续 n 次入队，一个是随机 n 次入队和 n 次出队的组合（采用4.2.4节中介绍的随机出栈序列实现）。在这两个场景下，我们对四种队列实现进行测试，其中整体搬移策略的顺序队列使用三个阈值：

$\frac{1}{2}$ 、 $\frac{3}{4}$ 和 1（禁止搬移）。双栈队列用同样采用了两个实现，分别是空间效率更高的共享栈实现，以及判断、移动次数更少的非共享栈实现。

表 5.1 不同队列的耗时对比（单位：ms）

n		搬移 閾值 $\frac{1}{2}$	搬移 閾值 $\frac{3}{4}$	搬移 閾值 1	链式	循环	双栈 共享栈	双栈 独立栈
仅插入	10^5	1	1	1	19	5	2	1
仅插入	10^6	11	14	17	198	51	27	9
仅插入	10^7	116	118	114	1953	575	245	102
插入/删除	10^5	3	2	2	37	3	10	6
插入/删除	10^6	32	27	26	382	31	105	62
插入/删除	10^7	364	273	256	3847	317	1100	627

通过实验可以发现，对于入队为主的情况，使用整体搬移的顺序队列的效率最高，因为它仅仅相当于连续执行了向量的尾插，没有任何多余的计算。双栈队列也可以达到这个效率水平，而循环队列由于取模运算的原因，会花费更多的时间。

但是对于入队和出队都比较多的情况，双栈队列策略因为每次出队都要进行一次判断，必要时还会进行栈间转移，所以会造成比较多的效率损失；而循环队列的出队则几乎没有损失。所以，在这种情况下，禁止搬移的效率最高，双栈队列的效率通常不如循环队列，整体搬移策略的效率则视具体场景和搬移的閾值而定。当閾值较高时，通常效率还是高于循环队列的。

所以总体来说，如果队列以入队为主，很少触发搬移，则建议使用整体搬移的策略；如果入队和出队都很多，则可以考虑循环队列，或者调高整体搬移的閾值。双栈队列的时间效率相对来说较低。无论哪种使用场景下，链式队列的时间效率都是最低的（即使我们使用了单向列表）；除非必要，否则总是不采用链式队列。

5.3 双端队列

本节简要地介绍一下**双端队列**（deque）。

双端队列是栈和队列的简单推广，在大多数《数据结构》教材中都省略了它。和栈和队列相比，双端队列基本没有什么新的知识点。笔者在这里介绍双端队列，是希望您加深对队列这一节的理解。您可以自己实现双端队列作为练习。

顾名思义，双端队列就是在队首和队尾，都可以进行插入、删除和访问的数据结构。对于双端队列 $Q_D[d:n]$ 来说，从队首 $Q_D[d]$ 和队尾 $Q_D[n]$ 两端都可能插入新的元素，也都有可能删除旧的元素。

所以，如果使用向量去实现双端队列，那么就要让 $Q_D[d:n]$ 尽可能保持位于向量 $V[0:m]$ 的中段。这是因为，双端队列的队首和队尾是对称的。因此，如果双端队列的初始位置偏前，那么在连续向队首 $Q_D[d]$ 插入新元素的过程中，会频繁地让 d 下降到 0 以下，从而需要对双端队列的元素做整体搬移。双端队列的初始位置偏后，也可以类似讨论。

设一次搬移之后，双端队列被放入了向量 $V[0:m]$ 中，作为 $Q_D[d:n]$ 。为了保证装填因子不太低，应有 $n - d = \Theta(m)$ 。那么，就必须保证， $d = \Theta(m)$ 。否则，连续进行 $d = o(m)$ 次对队首的插入，就要触发一次 $\Theta(n - d) = \Theta(m)$ 赋值操作的搬移，从而搬移的分摊复杂度会高于 $O(1)$ 。类似地，也需要保证 $m - n = \Theta(m)$ 。那么一个自然的想法，就是让 $m - n = d$ ，对称放置。

通过上面的分析可以知道，在没有先验信息的情况下，每次搬移应当让双端队列的前后，留出对称的空间以备插入。当然，实际应用中，双端队列的两端操作的频率并不一定是相等的，这个时候，就应当在保证 $d = \Theta(m), m - n = \Theta(m)$ 的前提下，适当调整这两段空间的比例。

如果采用循环队列的思路去实现双端队列，则没有以上的这些麻烦。因为循环队列是一个环状的结构，所以从队首插入与从队尾插入共享同一块空间。因此，无论两段操作的频率相等还是不等，对循环队列而言都是没有区别的。只需要在普通的循环队列的基础上简单地增加几个接口，就可以实现循环的双端队列。采用列表去实现双端队列，同样不会遇到这些麻烦，增加几个接口即可，不再赘述。

C++ STL 中的 `std::deque` 容器不是简单的顺序结构或链式结构，而是块 (chunk) 状结构。关于块状结构的简单讨论参见《查找》章，它可以认为是顺序结构和链式结构之间的折中。STL 中的队列适配器 `std::queue` 默认是基于 `std::deque` 的，因此它的性能也介于链式队列和几种顺序队列之间。它具有和链式队列一样稳定 $O(1)$ 的优点，并在时间上略快于链式队列。

最后，尽管出栈序列的计数很简单，但与之类似的“出双端队列序列”的计数却相当困难，它没有卡特兰数这样简明的解。除了双端队列之外，并行栈、串行栈（请注意串行栈不等于双栈队列）等情况也至今仍有一些科学家在投入研究^{[19] [20] [21]}。如果您感兴趣且学习过《组合数学》，可自行了解相关研究。

5.4 本章小结

由于队列的性质是 FIFO，所以队列的“出队序列”是唯一的。所以相对于“出栈序列”有 $\text{Catalan}(n)$ 种可能性的栈来说，队列要简单很多。本章注重介绍队列的多种实现，希望让读者理解在设计数据结构的内存管理策略时，可以采用的不同

思路。

链式结构是最简单的方法，但是时间复杂度的常数较高。而在采用顺序结构进行设计的时候，则需要使用一些技巧。可以在适当的时候进行整体搬移，显式释放已出队元素的空间；也可以采用和向量不同的方法去管理顺序结构。

对于向量 $V[0:n]$ ，只有一个端 n 是变量，因此向量天然和栈一样是单端的。为了让它支持队列在头、尾两端的操作，则我们需要两个端。无论是直接基于共享栈的双栈队列，还是使用循环利用技术的循环队列，都是基于这个思路。这两种方法都将 0 和 m 这两个端点位置首尾相接形成一个环，不同的是，循环队列让这个环被打通了，而双栈队列（基于共享栈的实现）在 0 和 m 这个端点处放了一个隔板，通过栈间转移的方式让元素从隔板的一侧转移到另一侧。

第6章 矩阵

在上面的几章中，我们对线性的数据结构进行了讨论。向量和列表作为线性表，存储的是一维的数据。当一维来到二维甚至更高维时，维度的增加并不会改变数据结构的线性性质，正如数学中的矩阵同样被认为是《线性代数》的组成部分。

本书将介绍多种二维线性表，三维或更高维的情况是相似的。根据每一维度上的元素个数是否为定值，二维线性表可以分为下列四种情况：

1. 第一维为定值、第二维也为定值，也就是**矩阵** (matrix)。在数学计算中大量使用矩阵，是最为常见的二维线性表。本章将对矩阵展开讨论。
2. 第一维不为定值、第二维为定值。这种结构通常被称为**块状表** (chunk list)。在双端队列的实现、分块查找等场景中会使用到。本章将不对这种情况进行介绍，参见《查找》的对应章节。
3. 第一维为定值、第二维不为定值。这种结构通常被称为**桶** (bucket)。在拉链法的散列表、桶排序、基数排序等场景中会使用到。本章将不对这种情况进行介绍，参见《散列》的对应章节。
4. 第一维不为定值、第二维也不为定值。这种结构比较自由、形式多样，在图的表示中会使用到。本章将不对这种情况进行介绍，参见《图》的对应章节。

在数学的矩阵 $A_{m \times n} = (a_{ij})_{m \times n}$ 中， a_{ij} 表示第 i 行、第 j 列的元素。在本章中，我们沿用这种叫法，将称第一维为**行** (row)，第二维为**列** (column)。

6.1 矩阵的结构

6.1.1 按行优先和按列优先

对于一个矩阵 $A_{m \times n}$ ，它的行数 m 和列数 n 都是固定的，用户只能获取或修改某个位置上的值，而不能插入或删除元素。清空一个矩阵，可以认为是将一个矩阵上所有的元素都赋值为 0。

```

1 template <typename T, size_t R, size_t C>
2     requires (R >= 0 && C >= 0)
3 class AbstractMatrix : public DataStructure<T> {
4 public:
5     virtual T& get(size_t r, size_t c) = 0;
6     virtual void set(size_t r, size_t c, const T& e) {
7         get(r, c) = e;
8     }
9     size_t size() const override {
10        return R * C;

```

```

11 }
12     virtual void clear() = 0;
13     T& operator[](std::pair<size_t, size_t> p) {
14         return get(p.first, p.second);
15     }
16 };

```

对于一般的矩阵 $A[0:m, 0:n]$, 它的存储需要 mn 个数据单元。这是一个定值, 所以我们可以将数据存入一片连续的 mn 个内存单元中, 以便于循秩访问。如果一行一行地存储 (同一行的数据, 内存地址是连续的), 就称为**按行优先**; 如果一列一列地存储, 就称为**按列优先**。

具体来说, 按行优先的意思是, $A[0:m][0:n]$ 会被展开成: $A[0][0], A[0][1], \dots, A[0][n-1], A[1][0], A[1][1], \dots, A[1][n-1], \dots$ 以此类推。同一行 (第一维相等) 的数据会被存放在连续的内存里, 图6.1展示了两个 3×3 的矩阵在按行优先和按列优先的情况下, 各个元素在内存中的实际次序。

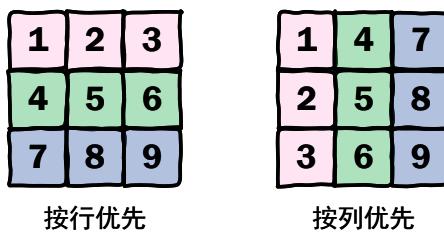


图 6.1 按行优先存储和按列优先存储

编程语言标准通常会对二维数组的按行优先或者按列优先作出规定。在 C/C++ 中的二维数组 $A[0:m][0:n]$ 是按行优先存储的, 而 MATLAB 等语言中二维数组是按列优先存储的。因此在 C/C++ 中, 直接采用二维数组就可以得到按行优先存储的矩阵。下列两种方式都可以定义一个二维数组:

```

1 T A[R][C]; // C style
2 std::array<std::array<T, C>, R> A; // C++ style

```

除了直接使用二维数组之外, 我们也可以用一个一维数组作为基础, 显式实现按行优先或按列优先存储的矩阵。按行优先的一个例子如下所示, 请您自己思考按列优先的情况。

```

1 template <typename T, size_t R, size_t C = R>
2     requires (R >= 0 && C >= 0)
3 class RowMajorMatrix : public AbstractMatrix<T, R, C> {
4     std::array<T, R * C> m_data;
5 public:
6     T& get(size_t r, size_t c) override {
7         return m_data[r * C + c];
8     }
9     void clear() override {

```

```

10     m_data.fill(T{});
11 }
12 };

```

6.1.2 实验：朴素矩阵乘法 *

在《线性代数》的学习过程中，我们了解了很多矩阵运算。矩阵乘法是矩阵运算中最基本的内容之一，无论是在计算机领域还是在其他工程学科中，都会广泛地使用到矩阵乘法。因此，如何高效地计算矩阵乘法，在学术界一直是一个热点话题。本书以矩阵乘法为例，介绍各种矩阵的特殊性质。为了简化问题，本书只讨论 $n \times n$ 方阵的乘法。代码可以在 *NaiveMatrixMultiply.cpp* 里找到。

```

1 template <typename T, size_t N, template<typename, size_t,
2   size_t> typename Matrix>
3   requires std::is_base_of_v<AbstractMatrix<T, N, N>, Matrix<
4     T, N, N>>
5 class NaiveMatrixMultiply : public Algorithm<void(const Matrix<
6   T, N, N>&, const Matrix<T, N, N>&, Matrix<T, N, N>&)> {};

```

这里的 **Matrix** 可以选用上一小节的按行优先或按列优先的矩阵实现。考虑矩阵乘法 $AB = C$ ，则按照矩阵乘法的定义，可以得到 c_{ij} 的表达式：

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}$$

如果直接用这个表达式，在 $AB = C$ 中依次计算矩阵 C 里的每个元素 c_{ij} ，那么可以得到下面的算法。

```

1 // MatrixMultiply_IJK
2 void operator() (const Matrix<N>& A, const Matrix<N>& B, Matrix<
3   N>& C) override {
4   C.clear();
5   for (size_t i { 0 }; i < N; ++i) {
6     for (size_t j { 0 }; j < N; ++j) {
7       for (size_t k { 0 }; k < N; ++k) {
8         C[{i, j}] += A[{i, k}] * B[{k, j}];
9       }
10    }
11 }

```

该算法的时间复杂度显然是 $\Theta(n^3)$ 。我们注意到，这个算法的三重循环之间互相没有依赖关系，因此，这三重循环的次序可以被任意地交换。除了上面的 IJK 版本之外，还有 IKJ、JIK 等版本。在示例代码中，我们对于按行优先和按列优先的矩阵，分别使用全部的 6 种版本进行测试。

虽然所有的这些版本使用的都是 $\Theta(n^3)$ 的算法，但是这些版本运行的时间仍

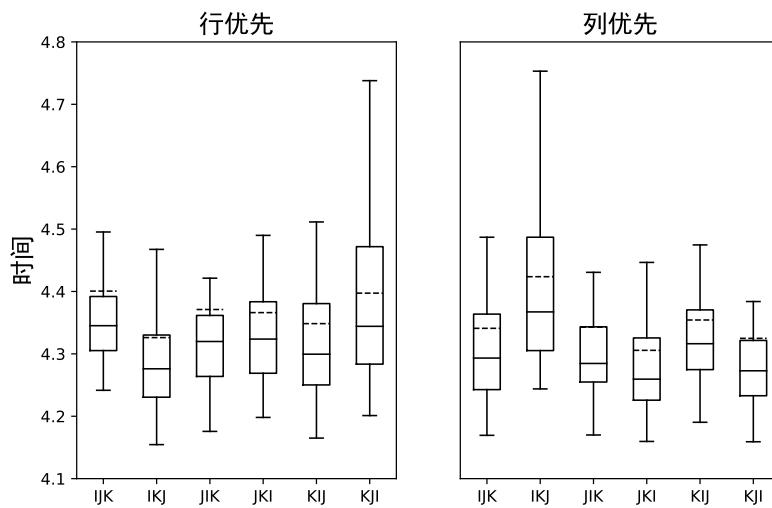


图 6.2 行优先和列优先在不同版本朴素乘法下的时间消耗

然存在一定的差异^[22]。图6.2展示了 $n = 400$, 运行 50 次, 去除异常值 (outlier) 之后的统计结果。可以看出, 虽然只是 50 次实验的统计, IKJ 版本在按行优先和按列优先的情况下已经展现出了显著的区别。

我们在3.3.6节里曾经介绍过, 数据的局部性会影响到算法的时间效率。在按行优先的矩阵里, 每一行的元素存储在连续的空间中, 所以如果连续访问同一行的元素, 则具有更好的局部性。反之, 在按列优先的矩阵里, 如果连续访问同一列的元素, 则具有更好的局部性。比如, 矩阵 A 的下标是 a_{ik} , 那么如果 I 在 K 的外层, 则访问次序为 $a_{00}, a_{01}, \dots, a_{0,n-1}, a_{10}, \dots$, 即连续访问同一行的元素; 另一方面, 如果 K 在 I 的外层, 即为连续访问同一列的元素。对于矩阵 B 和 C 也可以用同样的方法分析。我们看到, I 出现 2 次都是在行号上, J 出现 2 次都是在列号上, K 在行号和列号上各出现一次。因此, 对于 IKJ 来说, 外层是两个矩阵的行变化, 中层是一个行变化和一个列变化, 内层是两个列变化 (在同一行), 从而行优先的矩阵在 IKJ 下有最好的性能, 反之列优先的矩阵在 IKJ 下有最差的性能。其他的版本由于不像 IKJ 那么“极端”, 在图6.2中的实验规模下并不显著。

在考试中, 也有可能会综合考虑处理器、存储器、操作系统等软硬件功能的具体情况, 将《数据结构》、《组成原理》和《操作系统》的知识融合起来, 从而制造难度非常高的综合题。当给定存储器结构、访存和计算延迟、Cache 配置、虚存管理规则等信息之后, 就可以对各个版本的时间性能做更加细致的定量分析。

6.1.3 空间局部性和时间局部性 *

无论是按行优先还是按列优先，它都只能体现出矩阵作为二维线性表，在其中一个维度（行或列）上的存储连续性，而忽略了在另一个维度上的存储连续性。因此，按行优先的矩阵，只有在连续访问同一行的元素时，才具有较好的局部性；按列优先的矩阵亦然，它的局部性只体现在连续访问同一列的元素时。因此在朴素矩阵乘法的例子中，根据 I、J、K 三重循环的次序，有的版本更适用于按行优先的矩阵，有的版本更适用于按列优先的矩阵。

那么，有没有办法可以实现两个维度上的存储连续性呢？很遗憾，这是不可能的。如果在两个维度上都具有局部性，就要求横向或纵向相邻的两个元素地址相差不超过某个常数 C 。因为一个 $n \times n$ 的矩阵里，两个元素的位置相距不超过 $2(n - 1) = \Theta(n)$ 个单位（横向或纵向相邻为 1 个单位），所以任意两个元素的地址相差应当不超过 $C\Theta(n) = \Theta(n)$ 。然而，一个 $n \times n$ 的矩阵需要 $\Theta(n^2)$ 的空间，因此它的第一个元素和最后一个元素的地址，至少相差了 $\Theta(n^2)$ ，从而产生了矛盾。

因此，仅靠存储连续性（空间局部性）是无法提供两个维度上的局部性的。为了提供两个维度上的局部性，我们需要引入存储连续性之外的另一个提供局部性的通道：时间局部性。

无论是按行优先还是按列优先，它都只能体现出矩阵作为二维线性表，在其中一个维度（行或列）上的存储连续性，而忽略了在另一个维度上的存储连续性。因此，按行优先的矩阵，只有在连续访问同一行的元素时，才具有较好的局部性；按列优先的矩阵亦然，它的局部性只体现在连续访问同一列的元素时。因此在朴素矩阵乘法的例子中，根据 I、J、K 三重循环的次序，有的版本更适用于按行优先的矩阵，有的版本更适用于按列优先的矩阵。

1. **时间局部性** (temporal locality) 指：一个数据被访问之后，它可能在短时间内被多次访问。
2. **空间局部性** (spatial locality) 指：一个数据在被访问之后，它附近存储的数据也将被访问。

乍看上去，时间局部性似乎是空间局部性的加强：空间局部性只要求在一个数据访问之后，接着访问它附近存储的数据；而时间局部性则要求在一个数据访问之后，多次访问它本身。如果只从 Cache 的角度分析确实如此，但如果从访问权限的角度看，则时间局部性具有更加丰富的含义。如果一个算法仅仅在一小段时间里集中访问某个存储单元，那么它只需要在这一小段时间里获得这个存储单元的访问权限。以 IJK 版本的朴素乘法为例，该算法对结果矩阵 C 的每个元素 c_{ij} 具有非常强的时间局部性，在整个算法运行的 $\Theta(n^3)$ 时间里，只有 $\Theta(n)$ 的时间（一

轮最内层循环) 会访问它。这一时间局部性是对每个元素 c_{ij} 的, 和 C 是按行优先还是按列优先没有关系, 和算法中 I、J 分别在三重循环的哪一层也没有关系。因此, 从时间局部性的角度出发, 在两个维度上都有局部性。

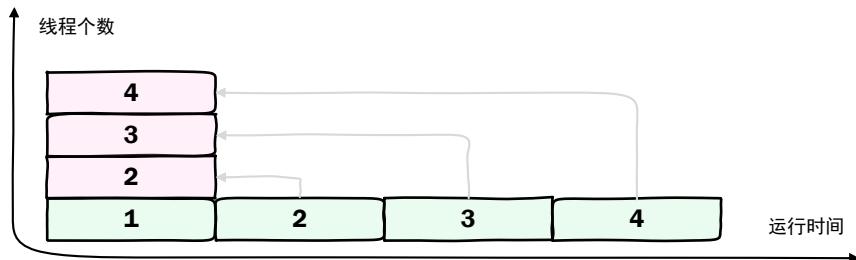


图 6.3 利用时间局部性实现并行算法

如图6.3所示, 假设原先的程序为绿色, 时间局部性使得程序可以被分为 4 段, 输出中的每个存储单元只会在其中的一段里被写入。那么, 这 4 段之间不会发生写入冲突, 它可以被放在 4 个线程中, 在同一时间段里并行计算。通过把绿色的程序段 2、3、4 移动到红色的位置, 整个程序的时间消耗降低到了原先的 $\frac{1}{4}$ 。

6.1.4 实验：并行矩阵乘法 *

回到 IJK 版本的朴素乘法上来。依据时间局部性, 我们就可以把整个算法拆分成 n^2 个线程, 每个线程负责计算一个元素 c_{ij} 。这样, 每个线程都只需要运算 $\Theta(n)$ 的时间, 并且互相之间不会产生冲突。如果处理器支持无限多个线程并行计算, 那么该并行算法的时间复杂度可以直接被降到 $\Theta(\log n) + \Theta(n) = \Theta(n)$ (其中 $\Theta(\log n)$ 是分派线程的时间)。现实中由于处理器支持的线程数有限, 对时间效率只会有常数级的提升, 但提升幅度仍然明显。

在 Visual Studio 中提供了对并行计算库 OpenMP 的支持, 在 *MatrixMultiply.cpp* 中提供了上述并行算法的示例代码, 如下所示。由于并行计算技术超出了《数据结构》的范围, 本书不会深入讨论如何对并行计算进行优化, 更多的是定性分析。

```

1 // MatrixMultiplyParallel
2 void operator()(const Matrix<T, N, N>& A, const Matrix<T, N, N>& B, Matrix<T, N, N>& C) override {
3     C.clear();
4     #pragma omp parallel for
5     for (int i { 0 }; i < N; ++i) {
6         #pragma omp parallel for
7         for (int j { 0 }; j < N; ++j) {
8             #pragma omp parallel for
9             for (size_t k { 0 }; k < N; ++k) {
10                 C[{i, j}] += A[{i, k}] * B[{k, j}];
11             }
12         }
13     }
14 }
```

```

11 }
12 }
13 }

```

6.1.5 实验：基于分治的矩阵乘法 *

在并行的朴素矩阵乘法（假设支持无限多个线程）中，每个线程写入的是一个 c_{ij} ，而它读取的是矩阵 A 中的一行，以及矩阵 B 中的一列。换言之，我们可以将矩阵 A 划分为 n 个行向量，即 $(\alpha_0^T, \alpha_1^T, \dots, \alpha_{n-1}^T)^T$ ，将矩阵 B 划分为 n 个列向量，即 $(\beta_0, \beta_1, \dots, \beta_{n-1})$ ，如图6.4中上图所示。使用下面的公式计算每个 c_{ij} 。

$$c_{ij} = \alpha_i^T \cdot \beta_j$$

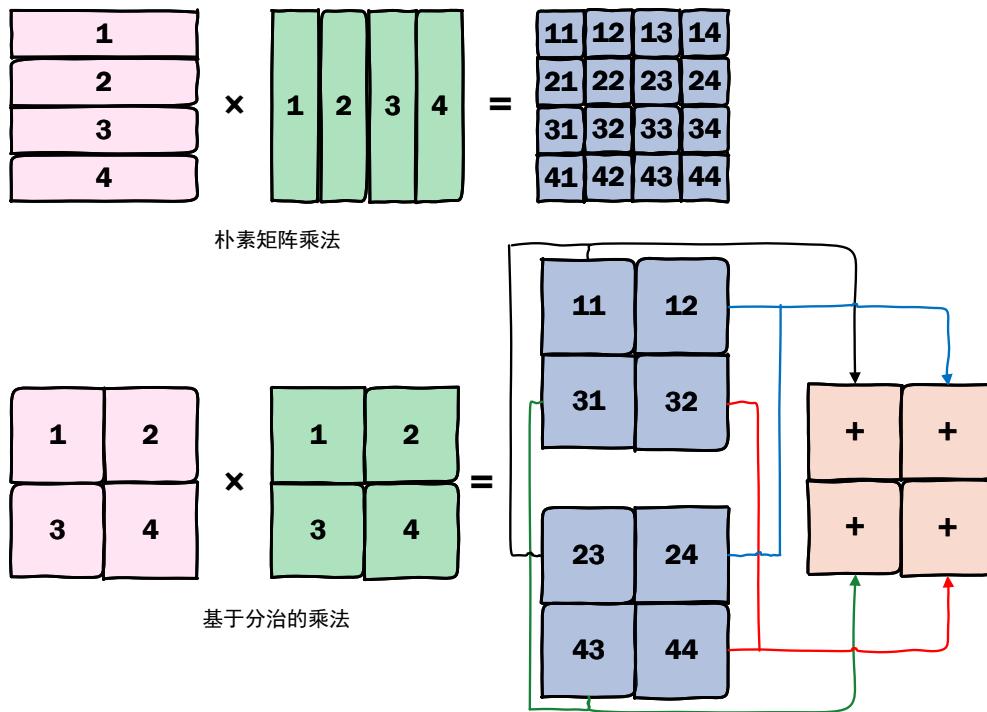


图 6.4 朴素矩阵乘法和基于分治的乘法

这并不是唯一一种矩阵划分方法。我们在归并排序中将线性表划分为长度相等的前后两段，这一策略在矩阵乘法中仍然适用。如图6.4中下图所示，我们可以将矩阵 A 、 B 、 C 的每个维度都分为长度相等的前后两段，即将每个矩阵等分为四部分，使用下面的分块矩阵乘法公式进行计算。

$$\begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \times \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix} = \begin{pmatrix} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{01} + A_{11}B_{11} \end{pmatrix}$$

可以看出，在基于分治的矩阵乘法中，两个 $n \times n$ 的矩阵相乘，可以转化为 8

次 $\frac{n}{2} \times \frac{n}{2}$ 的子矩阵相乘，以及 4 次 $\frac{n}{2} \times \frac{n}{2}$ 的子矩阵相加。由于 $\frac{n}{2} \times \frac{n}{2}$ 的矩阵相加时间复杂度为 $\Theta\left(\frac{n}{2} \times \frac{n}{2}\right) = \Theta(n^2)$ ，所以可以得到下面的时间复杂度递归式：

$$T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$$

利用主定理可以轻易地得到 $T(n) = \Theta(n^3)$ ，和朴素矩阵乘法相同。代码可以在 *MatrixMultiply.cpp* 中找到。为了降低问题的复杂性，我们只研究 n 是 2 的整次幂的情况。在示例代码中为了直观性，使用了一个类似于 `std::span` 的视图作为中间层，核心代码如下。

```

1 template <typename T, size_t N, template <typename, size_t,
2   size_t> typename Matrix>
3 class MatrixSpan {
4   Matrix<T, N, N>& m;
5 public:
6   pair<size_t, size_t> p;
7   size_t sz;
8   T& at(pair<size_t, size_t> q) {
9     return m[{ p.first + q.first, p.second + q.second }];
10 }
11 MatrixSpan operator[](pair<size_t, size_t> q) {
12   return MatrixSpan<T, N, Matrix>(m, { p.first + q.first *
13     sz / 2, p.second + q.second * sz / 2 }, sz / 2);
14 }
```

借助上述视图，可以将分治的矩阵算法以下面的形式实现。这里并没有像图6.4中那样，将递归调用计算出的结果放到两个矩阵中再加和，而是直接加到了结果矩阵 C 上。在这种情况下只能进行 4 路并行（矩阵 C 的每一块对应一个计算线程），如下面的代码所示。当处理器支持无限多的并行线程时，时间复杂度递归方程为 $T(n) = 2T(\frac{n}{2})$ 。因此和朴素矩阵乘法一样，该算法在无限多线程的情况下时间复杂度也为 $\Theta(n)$ 。感兴趣的同学可以自行实现 8 路并行的算法（此实验和《数据结构》无关），并推导出它在无限多线程情况下的时间复杂度为 $\Theta(\log^2 n)$ 。

```

1 template <typename T, size_t N, template <typename, size_t,
2   size_t> typename Matrix>
3 class MatrixMultiplyDivideAndConquer : public MatrixMultiply<T,
4   N, Matrix> {
5   using Span = MatrixSpan<T, N, Matrix>;
6   void multiply(Span A, Span B, Span C) {
7     if (A.sz == 1) {
8       C.at({0, 0}) += A.at({0, 0}) * B.at({0, 0});
9     } else {
10      #pragma omp parallel sections
11      {
12        #pragma omp section
13        {
14          multiply(A[{0, 0}], B[{0, 0}], C[{0, 0}]);
15        }
16      }
17    }
18 }
```

```

13         multiply(A[{0, 1}], B[{1, 0}], C[{0, 0}]);
14     }
15 #pragma omp section
16 {
17     multiply(A[{0, 0}], B[{0, 1}], C[{0, 1}]);
18     multiply(A[{0, 1}], B[{1, 1}], C[{0, 1}]);
19 }
20 #pragma omp section
21 {
22     multiply(A[{1, 0}], B[{0, 0}], C[{1, 0}]);
23     multiply(A[{1, 1}], B[{1, 0}], C[{1, 0}]);
24 }
25 #pragma omp section
26 {
27     multiply(A[{1, 0}], B[{0, 1}], C[{1, 1}]);
28     multiply(A[{1, 1}], B[{1, 1}], C[{1, 1}]);
29 }
30 }
31 }
32 }
33 public:
34     void operator()(const Matrix<T, N, N>& A, const Matrix<T, N,
35                     N>& B, Matrix<T, N, N>& C) override {
36         C.clear();
37         multiply(Span { A }, Span { B }, Span { C });
38     }
39 };

```

您在实验中会发现，上述算法和朴素矩阵乘法有数倍的时间性能差距，和并行的朴素矩阵乘法差距更远。这主要来自于递归时需要进行的额外赋值操作，和1.4.7节中数组求和的分治算法性能低下类似。另一方面，我们可以从实验的结果中看出，由于分块策略对两个维度是对称的，因此基于分治的矩阵乘法在处理按行优先和按列优先的矩阵时性能相似，并不会受到矩阵自身存储方式带来的局部性不对称的影响。

6.1.6 Strassen 算法 *

基于分治的矩阵乘法与朴素矩阵乘法相比，元素之间做加法和乘法运算的次数没有任何变化。对于加法而言，结合 $T(n) = 8T(\frac{n}{2}) + 4 \times (\frac{n}{2} \times \frac{n}{2})$ 与 $T(1) = 0$ ，可以解得 $T(n) = n^2(n - 1)$ 。而对于乘法而言，结合 $T(n) = 8T(\frac{n}{2})$ 与 $T(1) = 1$ ，可以解得 $T(n) = n^3$ 。这都和朴素矩阵乘法相同。从这个角度也能分析出基于分治的矩阵乘法的时间复杂度为 $\Theta(n^3)$ 。

我们希望进行更少的加法和乘法运算。从主定理的形式中我们可以看出，只有将系数“8”降下去才可以降低时间复杂度。这要求我们在计算分块矩阵的时候，利用每个 $C_{ij} = A_{i0}B_{0j} + A_{i1}B_{1j}$ 的相似性，使用少于 8 次的子矩阵乘法，就能计算

出全部的 C_{ij} 。

Strassen 算法可以做到这一点，它只需要 7 次子矩阵乘法就能计算出全部的 C_{ij} ，从而将算法时间复杂度从 $\Theta(n^3)$ 降低到了 $\Theta(n^{\log 7})$ 。不过，该算法的常数比较高，在较小的 n 下实际表现可能尚不如前面介绍的普通分治算法。在 *MatrixMultiply.cpp* 的示例代码中展示了 Strassen 算法的一种非并行的实现，如何更好地实现该算法仍然是一个受关注的问题^[23]^[24]。由于此算法已经完全超出《数据结构》的研究范围，在此不再展开其具体原理，只对其进行简要叙述，无需记忆。

- 首先，进行 10 次子矩阵加减法运算，将加减法的结果写入辅助矩阵。这 10 次矩阵加减法的内容是，对矩阵 A 和 B 分别进行同行相加、同列相减和主对角线相加。

$$\begin{cases} S_1 = A_{00} + A_{01}, & S_9 = B_{00} + B_{01} \\ S_2 = A_{10} + A_{11}, & S_7 = B_{10} + B_{11} \\ S_8 = A_{00} - A_{10}, & S_3 = B_{10} - B_{00} \\ S_6 = A_{01} - A_{11}, & S_0 = B_{01} - B_{11} \\ S_4 = A_{00} + A_{11}, & S_5 = B_{00} + B_{11} \end{cases}$$

- 其次，递归地进行 7 次子矩阵乘法，乘法的结果仍然写入辅助矩阵。这 7 个乘法可以并行。

$$\begin{cases} P_0 = A_{00} \times S_0 \\ P_1 = S_1 \times B_{11} \\ P_2 = S_2 \times B_{00} \\ P_3 = A_{11} \times S_3 \\ P_4 = S_4 \times S_5 \\ P_5 = S_6 \times S_7 \\ P_6 = S_8 \times S_9 \end{cases}$$

- 最后，进行 8 次子矩阵加减法运算，将运算结果写入结果矩阵 C 中。

$$C = \begin{pmatrix} P_4 + P_3 - P_1 + P_5 & P_0 + P_1 \\ P_2 + P_3 & P_4 + P_0 - P_2 - P_6 \end{pmatrix}$$

请您自己列出递归关系式，并使用主定理分析它的时间复杂度。

6.2 矩阵的压缩存储

6.2.1 欠自由度矩阵的压缩存储

对于矩阵 $A[0:m, 0:n]$ 来说，如果先验地知道矩阵所满足的一些性质，那么矩阵并没有 mn 个自由度，也就不需要用 mn 个数据单元去存储它了，这种技术称为矩阵的压缩存储。比如三角矩阵和对称矩阵，都只有大约一半的元素需要存储。我们采用如下的抽象压缩矩阵模板类，定义一个模板参数 D 用来存储压缩矩阵的实际自由度。

```

1 template <typename T, size_t R, size_t C, size_t D>
2     requires (R >= 0 && C >= 0 && D >= 0)
3 class AbstractCompressedMatrix : public AbstractMatrix<T, R, C>
4 {
5     protected:
6         std::array<T, D> m_data;
7         virtual size_t index(size_t r, size_t c) const = 0;
8     public:
9         size_t size() const override {
10             return D;
11         }
12         T& get(size_t r, size_t c) override {
13             return m_data[index(r, c)];
14         }
15     };

```

我们研究矩阵的压缩存储，核心问题是：矩阵中的一个元素 $A[r][c]$ 实际对应 `m_data` 中的哪一个元素。这是一个 $[0:R] \times [0:C] \rightarrow [0:D]$ 的映射 $f(r, c) = d$ ，也就是函数 `index`。对于不同类型的压缩矩阵，其区别也在于对函数 `index` 的设计。对于对称矩阵、上下三角矩阵等类型的压缩矩阵，您可以自己进行设计，并使用 `MatrixTest.cpp` 观察效果。比如，对称矩阵的一种实现如下：

```

1 template <typename T, size_t N> requires (N >= 0)
2 class SymmetricMatrix : public AbstractCompressedMatrix<T, N, N
3 , N*(N+1)/2> {
4     protected:
5         size_t index(size_t r, size_t c) const {
6             return (r < c) ? c * (c + 1) / 2 + r : r * (r + 1) / 2 +
7                 c;
8         }
9     };

```

上面这个版本有两种理解方法，如图6.5所示。第一种理解是，我们只存储了下三角矩阵 ($r \geq c$)，而上三角的部分则由下三角对称得到；同时我们采用按行优先的方法存储。此时， $f(r, c) = \frac{r(r+1)}{2} + c$ 。从对称矩阵 $A = A^T$ 的角度观察，我们可以意识到另一种理解也是可行的：即，我们只存储了下三角矩阵，并且是按列优先存储。

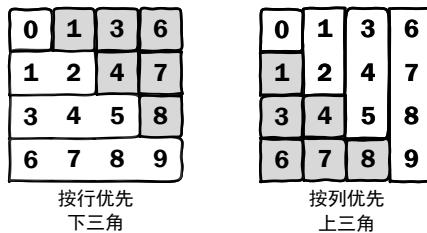


图 6.5 对称矩阵存储的两种理解方式

对称地，按行存储上三角矩阵和按列存储下三角矩阵，也共用同一个映射关系，请您自己写出相应的 `index` 函数。出于数学上使用矩阵的习惯，可能会要求您的矩阵下标从 1 开始，在这种情况下 `index` 函数又会有所不同，您同样可以很快推导出来。因此，这个知识点的考察方式灵活，难度却比较低，没有必要采用记忆的方法，建议您在考试过程中自己进行推导，可以通过找规律归纳总结。

还有一种常见的特殊矩阵是三对角矩阵，即除了 $|r - c| \leq 1$ 的元素之外，其他元素都是 0。它的一种简单的映射方式如下所示，请您分析它采用的存储方法，并计算另一种存储方法对应的映射。

```

1 size_t index(size_t r, size_t c) const override {
2     return 2 * r + c;
3 }
```

6.2.2 稀疏矩阵的压缩存储

除了满足某种特殊条件的矩阵可以压缩存储外，另一种情况就是所谓的**稀疏矩阵**。稀疏矩阵的意思是其中的绝大多数元素是 0（虽然不知道哪些元素是 0），这种矩阵在人工智能领域比较常见。矩阵的稀疏与否取决于具体的应用场景，没有绝对定义。对于 n 阶方阵而言，有的时候认为非 0 元素数量为 $O(n \log n)$ 时矩阵是稀疏的，而有的时候要求非 0 元素数量为 $O(n)$ 甚至 $O(1)$ 。

本节简要介绍存储稀疏矩阵的**三元组法**，其他方法将会在后面的章节中讨论。三元组，就是用“行号 + 列号 + 值”的三元组线性表存储所有的非 0 元素。三元组的压缩效率非常高，但是要“循秩”访问 $A[r][c]$ 的时候就必须遍历整个线性表查找，访问效率非常低。因此，三元组是一种以时间换空间的技术。下面展示了用向量存储三元组的例子。

```

1 template <typename T, size_t R, size_t C = R>
2     requires (R >= 0 && C >= 0)
3 class TripleMatrix : public AbstractMatrix<T, R, C> {
4     Vector<Triple<T>> V;
5     T zero {};
6 public:
7     T& get(size_t r, size_t c) override {
```

```

8   for (auto& t : V) {
9     if (t.row() == r && t.col() == c) {
10       return t.value();
11     }
12   }
13   V.push_back(Triple<T> { r, c, T {} });
14   return V.back().value();
15 }
16 const T& get(size_t r, size_t c) const override {
17   for (const auto& t : V) {
18     if (t.row() == r && t.col() == c) {
19       return t.value();
20     }
21   }
22   return zero;
23 }
24 };

```

在上面的示例代码中，我们为`get`函数重载了一个非`const`版本和一个`const`版本。当用户调用非`const`版本进行写入时，如果访问的位置在三元组向量里查不到，那么就会在向量末尾插入一个新的三元组存储这个位置上的元素。而当用户调用`const`版本进行写入时，如果访问的位置在向量里差不多，会直接返回 0。

如果保证每一个三元组中存储的值都非 0，那么三元组的空间复杂度及访问元素时最坏时间复杂度均为 $\Theta(k)$ ，其中 k 为矩阵中非 0 元素的数量。但因为我们可以给矩阵的元素赋值为 0，因此三元组向量的规模可能会大于 k 。为了解决这个问题，我们可以在非`const`访问元素的时候增加一步，使用2.5.7节介绍的方法删除所有值为 0 的元素。这一做法的时间复杂度是 $\Theta(k)$ ，并不影响访问元素的最坏时间复杂度。

6.3 本章小结

在计算机领域中通常不需要使用三维或更高维度的线性数据结构，而二维线性表是仍然比较常见的。本章介绍的是二维线性表中最为基本的一种：矩阵。需要指出，矩阵尽管是二维的，但它仍然是线性结构（两个方向上的线性）。因此在矩阵上做算法设计时，方法和一维连续存储的线性结构（向量）相似：

1. 取出矩阵的一行或一列作为平凡项，进行减治递归。
2. 将矩阵按行或按列二分为两半，或同时按行列二分为四块，进行分治递归。

这将与后面介绍的半线性结构与非线性结构有显著的不同。您可以在学习了树和图相关知识后，加深对此问题的理解。

第7章 树

在之前的章节中，本书对线性表展开了介绍。从存储结构的角度，本书介绍了顺序存储的向量和链式存储的列表。从访问限制的角度，本书介绍了 LIFO 的栈和 FIFO 的队列。接着，本书以矩阵为例引入了二维的线性结构。

从本章开始，本书将展开介绍半线性 (semi-linear) 的数据结构。本章将介绍最基本的半线性结构：树 (tree)。本章将围绕树的半线性性质展开，使读者了解半线性结构相对于前面几章所介绍的线性结构，有哪些区别和联系、有哪些独特的性质，以及应该如何在半线性结构上设计算法。在实际程序设计中往往用到的是更加复杂的半线性数据结构，我们将在后续的章节介绍它们。

特别需要强调的是，在半线性数据结构的部分，清华大学的本科教学所用的邓书，无论是知识点的数量，还是知识点的深度，都超出了统考大纲很多。本书的内容和邓书的定义没有冲突，但不确认和其他的教材没有冲突。在树的部分，存在很多定义有争议的内容，请您根据您使用的教材进行甄别。

7.1 树的结构

7.1.1 线性结构和全序关系 *

在此前的几章中介绍的数据结构都是线性结构，它们的元素之间存在一个线性的次序。线性次序是线性结构的核心要素，任意两个元素之间，可以比较它们在线性结构中的先后。换言之，线性次序中的先后比较，对线性结构中的元素构成了一个全序关系（参见2.6.2节）。

这一全序关系可以产生以下推论：

1. 线性结构中存在第一个和最后一个元素。
2. 线性结构是线性递归定义的。除去第一个元素，或除去最后一个元素，都是规模减一的同种线性结构。
3. 一般地，线性结构可以在任意位置处进行划分。该位置的前缀是一个同种的线性结构，该位置及其后缀构成另一个同种的线性结构。

依据上述的第 3 点推论，依据划分时前段和后段的长度不同，可以得到三种在线性结构上设计算法的方式，如图7.1所示。设线性表的规模为 n ，则：

1. 如果前段为 $O(1)$ 长度，而后段为 $\Theta(n)$ 长度，则前段为平凡项，后段为减治项，可以进行头部减治算法的设计。

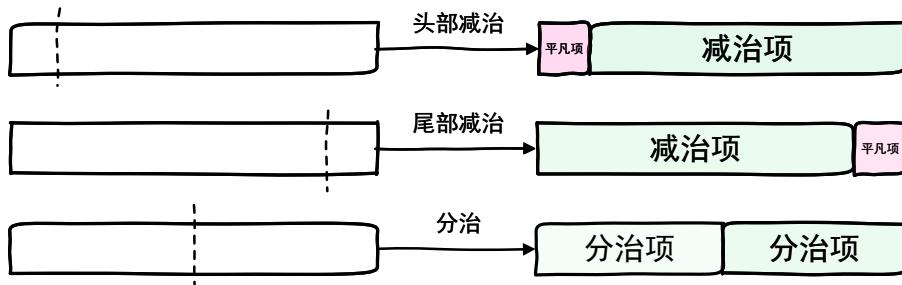


图 7.1 线性结构上设计算法的三种方式

2. 如果前段为 $\Theta(n)$ 长度，而后段为 $O(1)$ 长度，则前段为减治项，后段为平凡项，可以进行**尾部减治算法**的设计。
3. 如果前段和后段都为 $\Theta(n)$ 长度（通常为等长），则前段和后段均为分治项，可以进行**分治算法**的设计。由于非连续存储的链式结构中，访问中间的位置需要较多的时间，因此此算法通常只用于连续存储的顺序结构。

由于有限全序集必是良序集，所以事实上线性结构中的位置关系也构成良序关系。线性结构的另一个性质是良序关系的推论：除第一个元素外，每个元素都有唯一的直接前驱；除最后一个元素外，每个元素都有唯一的直接后继。矩阵及其他二维或高维的线性表不满足这一性质，对一个 d 维的线性表来说，它的每个元素具有至多 d 个“直接”前驱和至多 d 个“直接”后继。所以，从这个角度体现不了矩阵的线性特征。但如果我们从此前的三点推论出发，就能够发现矩阵和一维线性表之前的显著相似性。

1. 矩阵存在第一行、最后一行，以及第一列、最后一列。
2. 矩阵是线性递归定义的。除去第一行、最后一行、第一列、或最后一列，都是规模减少一行（或一列）的矩阵。
3. 一般地，矩阵可以在任意位置处，按任意一个维度进行划分，或同时按两个维度进行划分。矩阵的前若干行和后若干行都是矩阵，前若干列和后若干列也都是矩阵。矩阵任意位置的左上、左下、右上和右下也都是矩阵。

由于矩阵是连续存储的，因此我们可以仿照向量，在矩阵上设计减治算法和分治算法。减治算法的平凡项在矩阵中通常是一个行向量或一个列向量。对于减治算法和分治算法，典型的例子就是上一章介绍的朴素矩阵乘法和基于分治的矩阵乘法。正是由于这种相似性，我们认为矩阵也是一种广义的（二维的）线性结构。事实上，矩阵中的元素位置关系构成弱序关系。对于任意两个元素，我们可以比较它们的行（或列）在前、在后还是相同。

7.1.2 半线性结构和偏序关系

这一章将介绍的树 (tree) 则是一种半线性结构，它和矩阵这种二维线性结构有明显的区别。最重要的区别就在于：半线性结构的两个元素不一定可以比较次序。也就是说，树上的两个元素的位置，除了“前驱”、“后继”、“相等”之外，还有“无法比较”这种可能，它构成偏序关系。

本章将全面介绍半线性结构的特点，使您对它与线性结构的区别有深刻的理解。在本节中，我们将首先引入半线性结构相关的各种概念。

我们在列表的基础上进行扩展。在列表中，每个节点都有一个前向指针（表示位置的裸指针）和一个后向指针（表示所有权的智能指针）。前向指针指向直接前驱，后向指针指向直接后继。如果一个元素 x 沿着前向指针可以到达 y ，那么 y 是 x 的一个前驱。反之，如果一个元素 x 沿着后向指针可以到达 y ，那么 y 是 x 的一个后继。非空列表有且只有一个元素没有前驱（第一个元素），也有且只有一个元素没有后继（最后一个元素）。由于所有的元素都通过前向指针和后向指针互相连接，所以列表中两个元素的位置总是可以比较的，构成全序关系。

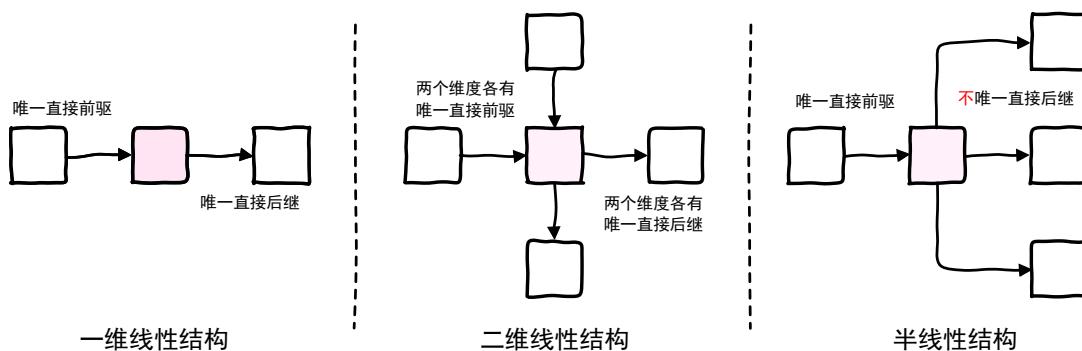


图 7.2 线性结构和半线性结构的对比

现在，我们将后向指针的数量限制放开。如图7.2中右图所示（省略了前向指针），在半线性结构中，每个节点都有一个前向指针（表示位置的裸指针）和任意多的有限个后向指针（表示所有权的智能指针，通常被实现为一个线性表）。这种形式的链式图很类似于一棵树开枝散叶的姿态。我们可以使用和列表相同的方法定义树上的前驱和后继。没有前驱的元素被称为根 (root)，而没有后继的元素被称为叶子 (leaf)。叶子以外的节点（包括根）被统称为内部节点 (internal node)。

如果在一个半线性结构中只有一个根，则称它为一棵树 (tree)；如果不要求只有一个根，则称它为一个森林 (forest)。因为人们的书写习惯是自上而下、自左而右的，所以在绘制表示树和森林的链式图的时候，通常把树根画在最上方，让树自上而下地“生长”；偶尔也会像图7.2那样画在最左方，自左而右地“生长”；而

不是像现实中的树那样，根在最下方，自下而上地生长。

特别地，规模为0的树称为空树。空树和规模为1的树（只有一个根，没有其他节点）合称为平凡树，它们经常作为递归算法的递归边界。

显然，树是特殊的森林，通常森林也会被实现为树的线性表。我们对于线性表已经非常熟悉，因此本章将更多地关注树的问题。

如前所述，树是在列表的基础上，通过取消后向指针只能有一个的限制，扩展得到的数据结构。因此，我们可以借用线性表中的概念，定义树上的各个概念，比如前驱、后继等。但习惯上，树上使用另一套名词，下面的各个小节将一一对这些名词展开介绍。

7.1.3 树上的节点关系

树上有许多来自于家族血缘关系。这是因为树开枝散叶的姿态，非常类似于家族的繁衍生息。

我们称一个节点的前向指针为亲代指针，指向它的亲代或亲节点（parent）；一个节点的后向指针为子代指针，指向它的子代或子节点（child）。亲代和子代的概念，分别对应线性表中的直接前驱和直接后继。

如果一个节点 x 沿着亲代指针能到达节点 y ，则称 y 是 x 的祖先（ancestor）。反之，如果一个节点 x 沿着子代指针能到达节点 y ，则称 y 是 x 的后代（descendant）。一个节点总是它自身的祖先和后代。自身以外的祖先（后代）称为真祖先或真后代。祖先和后代的概念，分别对应线性表中的前驱和后继。

拥有共同亲节点的节点之间，互为姊妹（sibling）。这个概念在线性表中没有对应。图7.3展示了从红色节点的角度，亲代、子代和姊妹在树上的所在位置。

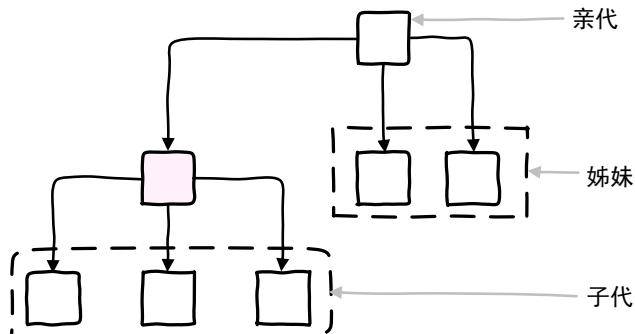


图 7.3 亲代、子代和姊妹

这里稍微说明一下本书中几个名词的选取。由于人类社会自古以来的男尊女卑问题，传统上的血缘关系树通常是父系亲族树。因此，早年间的学者将血缘关系上的称谓沿用到数据结构上时，也常常借用父系亲族树上的名词，使用 father、son、

brother 等单词。相应地，中文文献上译为“父亲”、“儿子”和“兄弟”。

随着性别平等意识的兴起，近年的英文文献已基本规范为 parent、child 和 sibling，相应地，中文文献也开始使用中性化的名词。但是，parent 常见的中性化翻译“双亲”^[5]有严重的误导性，容易让人误认为一个节点有 2 个 parent。为了避免在这一问题上产生迷惑，在本书中使用了一个冷门翻译“亲代”。对于 child，常见的中性化翻译“孩子”没有这个问题，不过为了对应，这里采用了“子代”的译法。对于 sibling，直接沿用了常见的中性化翻译“姊妹”。

我相信虽然用的词汇不同，但您都是可以理解并轻松能够转换的，这一转换的难度和数据结构知识本身的难度相比不值一提。考试时应以您使用的教材为准。

明确了树上的亲代、子代和姊妹等概念后，类似列表的节点，树上的节点可以如下定义：

```

1 template <typename T>
2 class TreeNode {
3     T m_data;
4     TreeNodePos<T> m_parent { nullptr };
5     Vector<TreeNodeInst<T>> m_children {};
6 public:
7     TreeNode() = default;
8     TreeNode(const T& data) : m_data(data) {}
9     virtual ~TreeNode() { /* ... */ }
10    T& data() { return m_data; }
11    TreeNodePos<T>& parent() { return m_parent; }
12    Vector<TreeNodeInst<T>>& children() { return m_children; }
13    bool isRoot() const { return m_parent == nullptr; }
14    bool isLeaf() const { return m_children.empty(); }
15    bool isParent(TreeNodePos<T> node) const {
16        return node != nullptr && node->m_parent == this;
17    }
18    bool isChild(TreeNodePos<T> node) const {
19        return m_parent == node;
20    }
21    bool isSibling(TreeNodePos<T> node) const {
22        return m_parent == node->m_parent;
23    }
24};

```

树的子代指针也可以用列表或者单向列表存储，请您自己分析这两种方案和示例代码中使用向量的方案比有何优劣。析构函数的设计将在后文中讨论，此处您只需要意识到，不能让它保留默认析构函数，否则会遇到3.2.1节所述列表析构函数相同的问题。

类似单向列表，我们也可以删去亲代指针，只保留子代指针。只保留子代指针的方法称为树的“子代表示法”^[5]，而本书中所采用的方法称为“亲代 + 子代表示法”^[6]。由于在半线性结构中，亲代指针和子代指针是不对称的，所以舍弃亲代

指针的子代表示法会遇到比单向列表更大的挑战：不但无法访问亲代，而且也无法访问姊妹。出于简单起见，本书不再给出子代表示法的示例代码，仅在理论部分对其进行分析。

7.1.4 树上的层次结构

类似于线性表中的秩，我们可以定义树中每个元素的深度（depth）。线性表中的第一个元素的秩为 0，对应地，根的深度也为 0。线性表中其他元素的秩等于其直接前驱的秩 +1，对应地，在树中，非根元素的深度为其亲代的深度 +1。

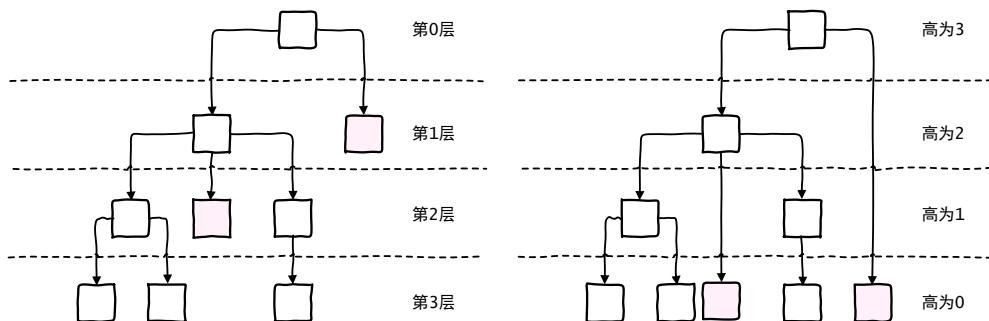


图 7.4 树的层次结构

当我们定义了深度之后，就可以把节点按照深度分成若干层（level）。习惯上我们从第 0 层开始，自上而下地画出每一层，就像图 7.4 的左图那样。深度为 d 的节点，就称其位于第 d 层。有些教材会定义树根在第 1 层而非第 0 层^[5]，这违反了计算机学科从 0 开始计数的思维方式，从而会导致后面的一些等式和不等式多出一个额外的 +1 项，既不美观又不方便记忆。这种情况下，笔者建议记忆时仍然按照第 0 层进行记忆，而在具体考试中通过检查来纠正。

对于深度的计算，可以根据深度的定义递归得到。由于一个节点只有一个亲代指针，所以该方法可以很自然地写成尾递归，并改写成下面的迭代形式。

```

1 size_t depth() const {
2     size_t d { 0 };
3     for (auto p { m_parent }; p != nullptr; p = p->m_parent) {
4         ++d;
5     }
6     return d;
7 }
```

显而易见，上述算法的时间复杂度为 $\Theta(d)$ ，其中 d 为所计算的节点的深度。

既然沿着亲代指针递归到根可以定义深度，那么反过来，也可以沿着子代指针递归到叶子，从而定义高度（height）。定义叶子的高度为 0，非叶子元素的高度为其子代高度的最大值 +1。注意由于子代可以不止一个，所以定义和深度有些许

不同。如图7.4中的右图所示，当按照高度对节点进行划分的时候，一些节点所在的层次会发生变化，如图中的2个红色节点。为避免歧义，通常说层次的时候指的都是按照深度划分的。

因为一个节点的子代指针数量是不确定的，所以我们计算高度的时候，无法借助尾递归改写为迭代。计算高度的示例代码如下所示。

```

1 size_t height() const {
2     if (isLeaf()) {
3         return 0;
4     }
5     size_t h { 0 };
6     for (const auto& child : m_children) {
7         h = std::max(h, child->height());
8     }
9     return h + 1;
10 }
```

我们将在后面的小节里具体分析这个算法。

7.1.5 树的递归定义

借助节点的深度（高度的情况是类似的，请您自己分析），我们可以在树上的元素位置之间定义一个弱序关系：深度小的元素排在深度大的元素前面。在这个弱序关系的基础上，我们可以进一步规定，同一层的元素中，左边的元素排在右边的元素前面，从而将其加强为全序关系。这种在半线性结构上构造全序关系，从而将半线性结构中的元素整理为一个线性表的做法，称为**半线性结构的线性化**。

在研究矩阵的时候，同样存在弱序关系到全序关系的转化。比如，在按行优先的矩阵中，我们首先根据元素所在行的序号，规定元素位置之间的弱序关系；其次，我们进一步规定，同一行的元素按照元素所在列的序号排列，从而将这个弱序关系加强为全序关系。尽管二者的线性化从形式上如此相似，作为二维线性结构的矩阵和作为半线性结构的树仍然有本质的区别。在按行优先的矩阵中，同一行的元素是实实在在存储在一起的，我们可以很方便地从一个元素出发，去访问同一行的其他元素。事实上，得益于矩阵的两个维度都是固定长度的，即使在按列优先的矩阵中我们也可以做到这一点。因此，矩阵能够很容易地在两个维度上进行减治和分治。

但是，树并不是按照层次递归定义的，当我们把树按照层次分成“上半部分”和“下半部分”时，我们只能保证“上半部分”是一个规模较小的树，而“下半部分”通常不是树。这会给我们的算法设计带来阻碍。比如，我们把图7.4的树的0、1层分为上半部分，2、3层分为下半部分，那么上半部分是一个规模为3的树，而下半部分则不是树，而是一个由3棵树组成的森林，3棵树的规模从左到右分别为

3、1、2。

于是我们发现，单独进行“上与下”的层次划分是不够的，还需要结合“左与右”，才能将一棵树分成多个树。又因为链式结构难以定位中间位置，所以“上半部分”通常只取第 0 层（根节点）。也就是说，我们可以将一棵树分成下面几个部分，以实现分治：

1. 第 0 层（树根），只有一个节点，作为平凡项。
2. 第 1 层及以下的部分，为包含若干棵树的森林，每棵树都作为一个分治项。

这一递归的思路基于树的递归定义，您可以回忆列表的递归定义，和下面的定义进行比照。

1. 平凡树是树。
2. 如果树根 r 有至少 1 个子节点 v_1, v_2, \dots, v_k ，且每个 v_j 的后代组成的结构都是树，则以 r 为根的也是树。

根据树的递归定义，在一棵树中，每个节点 v 的后代都组成了一棵树，称其为以 v 为根的子树。因为一个节点就对应了一棵子树，所以在不引起歧义的情况下，节点和子树这两个词可以互相指代。一个节点的每个子节点为根的子树，组成了该节点的子树森林。一个节点的子节点数量，也就是子树森林中树的个数，称为该节点的度（degree）。将一棵树拆分为根节点及其子树森林，并对子树森林中的每棵子树进行分治的做法，是在树上进行分治的最自然、最普遍的做法。

7.1.6 树的形态差异性

定义树的高度为其根节点的高度，则节点的高度也就等于以其为根的子树的高度。特别地，规模为 0 的空树的高度为 -1 。可以看出，高度的概念和子树是密切相关的，在7.1.4节中计算高度的代码也利用了树的递归定义。它的时间复杂度为 $\Theta(n_s)$ ，其中 n_s 为以所计算的节点为根的子树的规模。子树的规模可以用下面的方法计算，它同样利用了树上的自然分治。

```

1 size_t size() const {
2     size_t s { 1 };
3     for (const auto& child : m_children) {
4         s += child->size();
5     }
6     return s;
7 }
```

根据定义，您可以自行证明下面的定理：

定理 7.1：对于树上的任意节点，其深度和高度之和不大于树的高度。

高度是树的一个重要属性，通常用 h 表示。对于非平凡树而言， $2 \leq h \leq n$ 。两侧的等号对应了树的两种极端的形态。

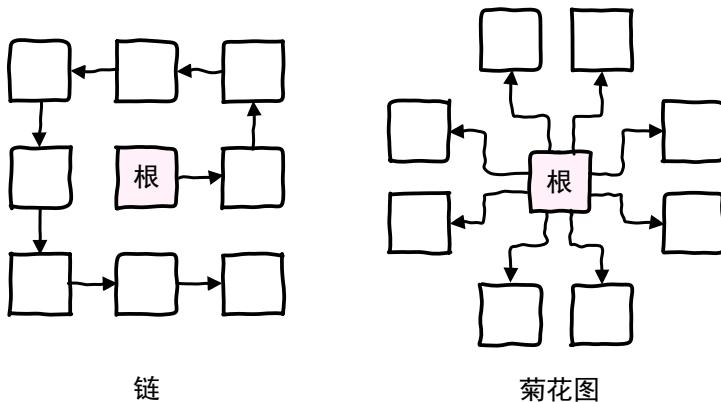


图 7.5 链和菊花图

当 $h = n$ 时，树上只有 1 个叶子，且每个内部节点都只有 1 个子代。这种情况下，树就退化成了一条链。退化成链的树，每一层都只有 1 个节点，它常常作为“最坏情况”的例子被用来分析一些算法在最坏情况下的复杂度。比如，当一个算法的最坏时间复杂度为 $\Theta(h)$ 时，如果没有被额外条件限制，那么在最坏情况下该复杂度是 $\Theta(n)$ 的。

而在 $h = 2$ 的情况下，除了根之外，所有的节点都是叶子，整棵树只有 2 层。由于在拓扑上，这种树可以被画成一个中心节点辐射出 $n - 1$ 个“花瓣”节点的形状，所以被称为菊花图。菊花图对应着另外一些算法的最坏情况。比如，当一个算法的最坏时间复杂度为 $\Theta(d)$ ，其中 d 为节点度数时，如果没有被额外条件限制，那么在最坏情况下该复杂度为是 $\Theta(n)$ 的。

从链到菊花图，树可以是“高瘦型”，也可以是“矮胖型”，具有一个很大的浮动范围。如图7.5所示，同样是 9 个节点的树，左图中的链树高为 8，根节点度数为 1，而右图中的菊花图树高为 1，根节点度数为 8，性质差异非常大。这为我们的算法设计造成了麻烦。一些算法会从树根出发查找叶子（从上到下访问），它们在“矮胖型”树表现更好，而在“高瘦型”树中需要承担 $\Theta(n)$ 的复杂度。而另一些算法可能需要经常遍历节点的每个子代（从左到右访问），在“矮胖型”树上就会面临高昂的成本。这一矛盾使得我们希望为树增加一些限制，从而引出后面几章将介绍的更加复杂的半线性结构。

7.2 插入和删除

7.2.1 树的定义 *

在列表中，因为每个节点都可以沿着前向指针最终到达第一个节点，或者沿着后向指针最终到达最后一个节点，所以只需要存储开头或结尾的位置即可。为

了便于操作，我们定义了哨兵节点放置在头尾两侧。树的亲代指针和子代指针是不对称的。沿着亲代指针总是可以到达树根，但沿着子代指针则无法确定会到达哪一片叶子。因此，在存储树的时候通常只存储树根的位置。

树根在树上是一个特殊的节点，因为除空树外，每棵树在树根所在的第0层，都只能有树根一个节点。即使我们在树根的亲代位置添加了一个“根哨兵节点”，也不能改变树根的这一特殊性，我们仍然不能插入新元素作为根哨兵的新子代，或作为树根的姊妹。所以，在树上我们通常不采用哨兵节点的设计。

下面是树的抽象类中主要方法的定义。您可以在学习本节内容的基础上，基于此前对列表实现方法的掌握，自己实现一个树，并使用 *TreeTest.cpp* 进行测试。

```

1 template <typename T>
2 class AbstractTree : public DataStructure<T> {
3 public:
4     virtual TreeNodePos<T> root() = 0;
5     virtual TreeNodePos<T> insertAsChild(TreeNodePos<T> p, const
6         T& e, size_t index) = 0;
7     virtual TreeNodePos<T> insertAsRoot(const T& e) = 0;
8     virtual TreeNodePos<T> find(const T& e) const = 0;
9     virtual T remove(TreeNodePos<T> p) = 0;
10    virtual void clear() = 0;
11 };

```

列表中，在一个位置处插入一个元素有两种可能：插入为直接前驱或直接后继。在树中由于亲代指针和子代指针不具有对称性，我们总是从树根出发，向子代的方向建树，所以很少需要“插入为亲代”的情况，大多数情况都是“插入为子代”。因此，在上述示例代码中只定义了插入为子代的方法，其中 `index` 表示“插入为从左到右的第几个节点”。因为树根不是任何节点的子代，所以需要单独定义一个插入为树根的方法。

和列表相似，树的一个基本实现如下。在该示例代码中没有设计一个维护树规模的单独字段，而是在调用 `size` 的时候现场计算。这个原因将在后面的小节里阐述；您也可以在自己的实验代码中实现为用单独字段存储规模的形式。

```

1 template <typename T>
2 class Tree : public AbstractTree<T> {
3     TreeNodeInst<T> m_root { nullptr };
4 public:
5     TreeNodePos<T> root() override { return m_root; }
6     size_t size() const override { return m_root->size(); }
7     Tree() = default;
8     Tree(TreeNodeInst<T>&& root) : m_root(std::move(root)) {}
9 };

```

7.2.2 插入元素和插入子树 *

有了列表的基础，在树上设计插入、删除这样简单的算法，您应当已经得心应手。在树上插入一个元素，您只需要生成被插入的节点，并加入到被插入位置的子代线性表中。该节点在子代线性表中的秩即为`index`。下面的示例代码中使`index`的默认值为-1，此时新节点将会被插入到子代线性表的末尾，这是“插入一个新子代”的一种比较自然的理解。

```

1 TreeNodePos<T> insertAsChild(TreeNodePos<T> p, const T& e,
2     size_t index = -1) override {
3     auto node { TreeNodeInst<T>::make(e) };
4     node->m_parent = p;
5     if (index == -1) {
6         index = p->children().size();
7     }
8     p->children().insert(index, std::move(node));
9 }
```

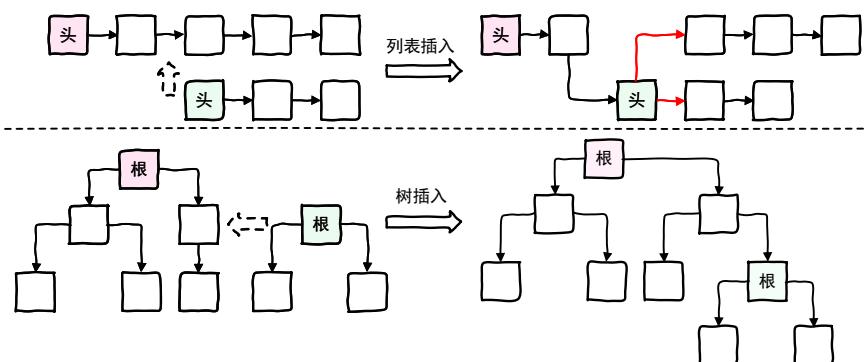


图 7.6 列表的插入和树的插入

插入为树根的情况更加简单。在这里我们进行一个扩展，即插入为子代的不一定是一个元素，而有可能是另一棵树。在列表的场合这是无法进行的，因为这会导致列表“分叉”，如图7.6中的上图所示，红色箭头表示的两个后向指针会破坏列表的性质。所以如果要将一个列表插入到另一个列表，那么就只能插入到目标列表的尾部。而下图所示的树因为允许多个子代指针，所以不需要担心分叉的问题，从而我们可以很自然地将一棵树插入到另一棵树的任何一个位置。插入一棵子树的做法和插入一个元素非常相似，请您自己设计，并和下面的示例代码进行比较。

```

1 TreeNodePos<T> attachAsChild(TreeNodePos<T> p, TreeNodeInst<T>
2     >&& node, size_t index = -1) override {
3     node->parent() = p;
4     if (index == -1) {
5         index = p->children().size();
6     }
7     p->children().insert(index, std::move(node));
```

```

7 |     return p->children()[index];
8 |

```

插入元素或子树的时间复杂度均为 $\Theta(d - r)$, 其中 d 为被插入位置的度, r 为插入的节点在子代向量中的秩。但是, 如果我们用单独的字段来存储树的规模, 那么插入子树的时候就需要计算子树的规模, 所以时间复杂度变成了 $\Theta(d - r + n_s)$, 其中 n_s 为插入的子树的规模。出于这一考虑, 我们没有引入规模字段。

7.2.3 删 除子树和分离子树 *

在列表上删除一个元素的时候, 会让它的直接后继代替它的位置, 成为它直接前驱的直接后继。而在树上删除一个元素的时候, 因为它的直接后继(子代)可能不止一个, 无法判断将哪一个代替它的位置, 所以通常不会直接删除孤立的一个元素, 而是删除以它为根的整棵子树。因为被删除的子树可能仍有用处(比如, 插入到别的位置上去), 所以这里除了常规的 `remove` 方法之外, 还设计了一个 `detach` 方法用来分离子树。因为使用了智能指针, 二者只有返回值上有差别; 在不使用智能指针的项目中, `remove` 方法还需要负责将被删除的子树所占空间释放掉。下面展示了分离子树的示例代码。

```

1 TreeNodeInst<T> detach(TreeNodePos<T> p) override {
2     if (p == m_root) {
3         return std::move(m_root);
4     } else {
5         auto& children { p->parent() ->children() };
6         size_t index { 0 };
7         while (children[index] != p) {
8             ++index;
9         }
10        return children.remove(index);
11    }
12 }

```

注意, 尽管删除子树和分离子树只有返回值有区别, 但二者的时间复杂度是不同的。删除子树的时间复杂度为 $\Theta(r + n_s)$, 而分离子树只需要 $\Theta(r)$ 。这是因为在删除子树的代码中, 智能指针释放空间的过程被隐藏了, 而释放整棵子树的空间需要 $\Theta(n_s)$ 的时间。

7.3 树的遍历

为了在树上查找一个元素, 我们需要首先制定一个方案遍历树上的每一个节点: 只有这样才能将树上的每个节点上的值和待查找的元素对比。也就是说, 我们需要以某种方式将树上的节点排列为一个线性序列, 按照这个线性序列的次序

一一访问每个节点上的值。这一过程称为**遍历** (traverse)。于是，树上的查找问题，就被转换为了树的遍历问题，也就是半线性结构的线性化问题。

我们定义下面的抽象类型作为树的遍历算法的基类。其中，第一个参数表示被遍历子树的树根位置，第二个参数表示访问每个节点的函数。

```
1 template <typename T>
2 class AbstractTreeTraverse : public Algorithm<void(
    TreeNodeConstPos<T>, std::function<void(const T&)>) > {};
```

7.3.1 层次遍历

借助节点的深度（即所在层次）可以得到一个自然的线性化策略：深度小的元素排在深度大的元素前面；在同一层，左边的元素排在右边的元素前面。这一方法线性化得到的序列称为层次序列，以层次序列遍历树的过程即称为**层次遍历** (level-order traverse)，一些书上也称为**层序遍历**^[5]。

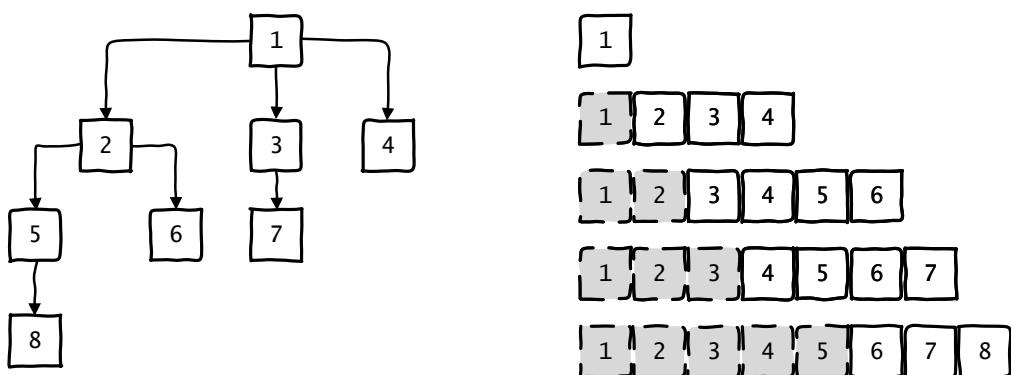


图 7.7 树的层次遍历

要设计层次遍历的算法，关键在于如何让同一层的元素按照从左到右的次序排列。如图7.7中的左图所示，由于层次的划分会“切断”树上的指针相连关系，访问完节点 5 需要访问节点 6 时，需要向上（沿亲代指针）移动 1 步再向下（沿子代指针）移动 1 步；访问完节点 6 需要访问节点 7 时，需要向上移动 2 步再向下移动 2 步；从节点 7 到节点 8，则是向上移动 2 步、向下移动 3 步。这种向上向下移动的不确定性为我们设计算法带来了困难。

为了避免这个不确定性，我们希望在访问节点 2 的时候，就提前“预订”好访问节点 5 和 6，这样就不需要再从 5 去找到 6。同样地，访问节点 3 的时候“预订”好节点 7，也就不需要再从 6 去找到 7。您可以发现这是一个“先入先出”的过程，如图7.7中的右图所示，在访问节点 1 后，预订节点 2、3、4（入队）。接着，队头的节点 2 出队被访问，预订的 5 和 6 入队，以此类推。除树根在算法开始时入队外，每个节点会在其亲代被访问时入队，自身被访问时出队。

定理 7.2： 在这个过程中，每个节点都会且只会入队一次，也会且只会出队一次。证明 使用数学归纳法，对树的规模 n 归纳。

1. 对于平凡树显然。
2. 对于非平凡树，其根节点在入队之后会立刻出队，随后将第 1 层的节点依次入队。后续过程相当于同时对每个规模更小的子树做层次遍历。

■

从上面的证明可以看出，尽管层次遍历没有遵循树的自然分治结构，但我们仍然可以利用子树的思想去研究它。该定理同时意味着在访问单个节点的时间复杂度为 $O(1)$ 的情况下，这个算法的时间复杂度为 $\Theta(n)$ 。下面我们说明，这个算法确实能够产生满足本小节开头定义的层次遍历序列。

1. 由于队列先进先出的特点，所以如果 u 在 v 的前面，那么 u 的子代也会在 v 的子代前面。
2. 记节点 v 的深度为 $d(v)$ 。对于节点 u 和 v ，如果 $d(u) < d(v)$ 。记 u_1 是 u 的亲代， u_2 是 u_1 的亲代，以此类推，对 v 也类似定义。最终， $u_{d(u)}$ 为树根，必定在不为树根的 $v_{d(u)}$ 前面。根据 (1)，可以逐步推出每一个 u_j 都在 v_j 前面。因此，满足“深度小的在深度大的节点前”的要求。
3. 对于同一层的节点 u 和 v ，如果 u 在 v 左侧。设它们的最低公共祖先 (lowest common ancestor, LCA) 为 a ，那么 a 作为 u 祖先的子代 u_j 会在作为 v 祖先的子代 v_j 之前。根据 (1)，可以逐步推出 u 在 v 之前。因此，也满足“同一层的节点，左边在右边之前”的要求。

LCA 是在处理“树上两个节点”相关的证明问题时，一个比较常用的技巧。当我们需要对两个节点的情况进行分析的时候，通常会选取它们的 LCA。在 LCA 为根的子树之外的节点，通常与所研究的问题无关，可以忽略；而在子树内，我们可以保证被研究的两个节点位于 LCA 的不同子树上，这是一个新的条件。因此，我们选取 LCA，相当于为所研究的问题人为地增加了一个条件，从而降低了问题的解决难度。

上述算法的示例代码如下所示。

```

1 // TreeLevelOrderTraverse
2 void operator()(TreeNodeConstPos<T> p, std::function<void(const
3   T&)> visit) const override {
4   if (!p) {
5     return;
6   }
7   Queue<TreeNodeConstPos<T>> Q;
8   Q.enqueue(p);
9   while (!Q.empty()) {
10     auto node { Q.dequeue() };
11     visit(node);
12     for (auto child : node.get().children) {
13       Q.enqueue(child);
14     }
15   }
16 }
```

```

10     visit(node->data());
11     for (auto& child : node->children()) {
12         Q.enqueue(child);
13     }
14 }
15 }
```

7.3.2 先序遍历和后序遍历

在7.1.5节中，我们已经意识到层次序列不遵循树的递归定义。当我们研究的问题和树的结构相关，比如说计算每个节点的高度或计算每个子树的规模时，层次遍历就变得无能为力。回顾树的递归定义：

1. 平凡树是树。
2. 如果树根 r 有至少 1 个子节点 v_1, v_2, \dots, v_k ，且每个 v_j 的后代组成的结构都是树，则以 r 为根的也是树。

可以自然地引出一个符合自然分治思想的遍历方法：

1. 递归边界：平凡树。
2. 对于非平凡树，首先访问树根，然后依次递归地遍历每棵子树。

因为访问树根在递归遍历每棵子树之前，因此上面这种方法称为**先序遍历**或**先根遍历** (pre-order traverse) [5]，一些书上也称为**前序遍历**。相应地，如果访问树根在递归遍历每棵子树之后，就称为**后序遍历**或**后根遍历** (post-order traverse)。与之相对应的就是先序序列和后序序列。先序遍历的一个示例实现如下所示，您可以思考将visit移动到哪个位置可以变为后序遍历。

```

1 // TreePreOrderTraverse
2 void operator()(TreeNodeConstPos<T> p, std::function<void(const
3     T&)> visit) const override {
4     if (!p) {
5         return;
6     }
7     visit(p->data());
8     for (auto& child : p->children()) {
9         operator()(child, visit);
10    }
11 }
```

如果说层次遍历的次序是按上下优先，上下在同一层才考虑左右，那么先序和后序遍历的次序就是按左右优先，只不过这里的“左右”是在同一棵子树内的。在层次序列中，只有在同一“行”（同一层）的节点，左右次序才能决定位置关系。而在先序序列和后序序列中，相应地，也只有在同一“列”（有祖先-后代关系）的节点，上下次序才能决定位置关系。您可以近似地理解为，类似于矩阵线性化的按

行优先和按列优先，层次序列是“按行优先”，而先序序列和后序序列是“按列优先”的。具体来说，先序序列和后序序列满足这样的性质：

1. 如果节点 u 是节点 v 的祖先，那么 u 在先序序列上的位置在 v 之前， u 在后序序列上的位置在 v 之后。
2. 如果节点 u 和节点 v 无祖先-后代关系，设 u 和 v 的 LCA 为 a ，则先序、后序序列中， u 在 v 前等价于在 a 的子代中， u 的祖先在 v 的祖先左侧。

您应当可以发现，先序遍历符合4.5.1节中介绍的尾递归扩展形式，因而可以借助栈改写成非递归的形式。后序遍历则不服从尾递归的扩展形式，如果要去除递归会比较复杂。下面是先序遍历消除递归之后的形式，可以看出，它和层次遍历的形式具有高度的相似性，区别有二：首先，先序遍历是基于栈的，层次遍历是基于队列的；其次，栈的 LIFO 特性使得先序遍历需要将各个子代指针逆序地加入到栈中，而队列的 FIFO 特性使得层次遍历只需要将各个子代指针顺序地加入到栈中。

```

1 // TreePreOrderTraverseIterative
2 void operator()(TreeNodeConstPos<T> p, std::function<void(const
3   T&)> visit) override {
4   if (!p) {
5     return;
6   }
7   Stack<TreeNodeConstPos<T>> S;
8   S.push(p);
9   while (!S.empty()) {
10     auto node { S.pop() };
11     visit(node->data());
12     for (size_t i { node->children().size() }; i > 0; --i) {
13       S.push(node->children()[i - 1]);
14     }
15   }
}

```

先序遍历和层次遍历的代码上的相似性，也体现了它们在性质上的相似性。比如，如果节点 u 是节点 v 的祖先，那么 u 在先序序列和层次序列上的位置都在 v 之前。因而关于先序遍历的一些研究结果，可以迁移到层次遍历中，反之亦然。

7.3.3 树的节点析构 *

下面我们来分析树的节点 `TreeNode` 的析构函数设计。析构一个节点，也就是析构以该节点为根的子树。如前所述，我们不能让析构函数保持默认，否则当树的形态为一条链时，析构函数的 $\Theta(n)$ 级递归调用就会引发栈溢出问题。

默认的析构函数事实上进行了一次后序遍历：先递归地析构掉每个子树，再析构当前节点本身。后序遍历是很难去除递归的，但在析构这个场合，问题可以得到简化：被析构后的节点是不复存在的。所以，当我们需要析构一棵树时，我们可

以先找到一片叶子，将其析构并从树上删除，然后再找一片叶子，将其析构并从树上删除，以此类推，直到待析构的树变为平凡树为止。由于我们采用了向量存储子代指针，而向量在尾部删除时效率最高，所以我们每次寻找叶子的时候，总是向最右侧的分支寻找。析构函数的一个示例实现如下。

```

1 TreeNodePos<T> rightMost() {
2     TreeNodePos<T> p { this };
3     while (!p->isLeaf()) {
4         p = p->children().back();
5     }
6     return p;
7 }
8 virtual ~TreeNode() {
9     auto p { rightMost() };
10    while (p != this) {
11        auto q { p->parent() };
12        q->children().pop_back();
13        p = q->rightMost();
14    }
15 }
```

7.3.4 势能分析法 *

上一小节所述的析构函数因为有两层循环，其时间复杂度并不直观。这里介绍一种**势能分析法**^[18]来解决复杂的时间复杂度分析问题。**势能** (potential) 是一个物理概念，在复杂度分析领域借用了这个概念。首先，我们来看一个物理模型：

一架飞机具有动能 E_v 和重力势能 E_p ，并且能量转换效率为 100%。那么，消耗的燃料释放的内能，会完全转换为动能和重力势能，也就是

$$U = \Delta E_v + \Delta E_p$$

现在飞机消耗燃料不断加速 ($\Delta E_v > 0$)。在这个过程中，飞机可能不断升高或降低，而升高降低加减的重力势能会互相抵消。于是，考虑整个加速过程，我们可以得到下面的公式：

$$\sum_{i=1}^n U^{(i)} = \sum_{i=1}^n E_v^{(i)} + E_p^{(n)} - E_p^{(0)}$$

那么我们要分析燃料释放的内能 $\sum_{i=1}^n U^{(i)}$ ，只需要将每个时间段增加的动能相加得到 $\sum_{i=1}^n E_v^{(i)}$ ，然后关心加速过程前后的重力势能即可。

现在我们回到复杂度分析来。我们定义一个势能函数 ϕ 。将每次操作消耗的时间（对应燃料释放的内能） a_i 分成两部分，一部分为“基本时间”（对应增加速度

产生的动能变化量) b_i , 另一部分为累积的势能 $\Delta\phi$, 即:

$$a_i = b_i + \Delta\phi$$

考虑整个算法的执行过程, 可以得到下面的公式:

$$\sum_{i=1}^n a_i = \sum_{i=1}^n b_i + \phi(n) - \phi(0)$$

于是, 我们想要求整个操作消耗的时间 $\sum_{i=1}^n a_i$, 只需要研究总的“基本时间”
 $\sum_{i=1}^n b_i$, 以及算法开头结尾的势能变化即可。接下来用本小节讨论的析构函数为例
子, 介绍势能分析法的使用。下面的过程是分步骤书写的, 这是为了展示思考过
程。待您熟悉势能分析法后可以合起来写。

1. 定义每轮外层循环 (析构函数里的循环) 为“一次操作”。因为每次操作都会
删除 (析构) 一个节点, 所以总操作次数为 $\Theta(n)$ 的。
2. 在第 i 次操作的过程中, 找亲代和尾部删除都是 $O(1)$ 的操作, 假设 `rightMost`
进行了 k_i 轮内层循环, 则一次操作的时间 $a_i = 2 + k_i$ 。
3. 定义势能 ϕ 为变量 p 的深度。那么, 从 p 到 q 势能减小 1, `rightMost` 势能
增加 k_i , 于是 $\Delta\phi = k_i - 1$ 。
4. 算出基本时间 $b_i = 3 = O(1)$ 。累加得总的基本时间为 $\Theta(n)$ 。
5. 在循环结束时 p 到达树根, $\phi(n) = 0$ 。循环开始时 $\phi(0) = k_0$ 。因此 $\sum a_i =$
 $\sum b_i + \phi(n) - \phi(0) = \Theta(n) - k_0$ 。
6. 初始的 k_0 深度是循环外的 `rightMost` 函数产生的。因此整个算法的时间复
杂度即为 $\Theta(n)$ 。

势能分析法是解决复杂的多重循环问题时的重要方法。考试中遇到的势能分
析通常比较简单, 势能函数通常是一个简单直观的变量 (比如本题)。本书后续还
会遇到一些复杂的势能分析法的例子。

7.3.5 实验: 三种遍历的对比 *

本节中介绍了层次遍历、先序遍历和后序遍历三种遍历方式。如果认为 `visit`
函数的时间复杂度是 $O(1)$ 的, 则三者的时间复杂度均为 $\Theta(n)$, 而空间复杂度有所
不同。先序遍历和后序遍历的空间复杂度和递归深度相关, 因此为 $\Theta(h)$ 。层次遍
历的空间复杂度则为 $\Theta(\max_j(n_j + n_{j+1}))$, 其中 n_j 和 n_{j+1} 是第 j 层和第 $j+1$ 层的节
点数量, 请您利用复杂度的定义自行证明上述结论。

在 *ChainFlower.cpp* 中, 我们使用链和菊花图分别对这三种遍历的性能进行测

表 7.1 不同遍历算法的耗时对比（单位：ms）

树的形态	n	先序遍历 递归	先序遍历 迭代	后序遍历 递归	层次遍历 搬移队列	层次遍历 循环队列
链	10^3	0	0	0	0	0
链	10^4	栈溢出	1	栈溢出	2	1
链	10^5	栈溢出	16	栈溢出	27	13
链	10^6	栈溢出	117	栈溢出	212	126
菊花图	10^3	0	0	0	0	0
菊花图	10^4	2	1	2	1	1
菊花图	10^5	19	11	16	17	16
菊花图	10^6	149	110	153	126	147

试。首先容易测试得到，递归形式的先序遍历和后序遍历对于比较长的链会发生栈溢出错误。和先序、后序遍历形式相似的递归算法也会发生栈溢出错误。比如，如果您没有采用上一小节介绍的方法重写节点的析构函数，而是采用了默认析构函数（和后序遍历相似），那么比较长的链在析构的时候也会发生栈溢出错误。此外，7.1.5节中介绍的子树规模计算算法相当于先序遍历，因此也有可能发生栈溢出错误。为了避免规模计算中的栈溢出错误，我们可以将其改为非递归的先序遍历。

其次，我们可以看到同为非递归的形式，基于栈的先序遍历的性能会高于基于队列的层次遍历。这是因为和首、尾两个自由端的队列相比，只有一个自由端的栈行为更加简单，它需要实现队列的内存管理机制，从而可以获得更高的性能。

注意到，先序遍历和层次遍历的性能差异在菊花图上不明显，而在链上比较明显。您可能会感到疑惑，因为层次遍历在菊花图上的空间复杂度为 $\Theta(n)$ ，而在链上为 $O(1)$ ，为什么空间复杂度较高的情况反而时间性能更好？这是队列自身的实现机制决定的。示例代码中的队列默认采用整体搬移策略进行实现，在链上进行层次遍历的时候，将不断进行入队和出队操作，每次都会触发整体搬移以释放刚刚出队的元素的空间；而在菊花图上进行层次遍历的时候，批量性地入队再批量性地出队，不会进行整体搬移。您可以结合示例代码中对队列的实现进行分析。如果不采用整体搬移策略，而是采用循环队列，链的性能就会高于菊花图，因为循环队列没有整体搬移的开销，只有扩容的开销。您还可以将它改为链式队列和双栈队列观察效果，复习《队列》章中介绍的各种队列的性能对比。

7.4 树和栈的关系

7.4.1 树和森林的计数 *

树不但是递归定义的，而且是可以和森林相互定义的。树的根节点的子树构成了森林，而森林是树的线性表。因此，我们研究树和森林的计数问题时，也可以将二者统一起来。

设规模为 n 的树的数量为 $T(n)$ ，规模为 n 的森林的数量为 $F(n)$ 。在我们研究计数问题的时候，通常讨论的是无标号（unlabelled）树，也就是对树的形态进行计数。我们交换树上的节点而不改变树的形态时，认为是同一棵无标号树。而带标号（labelled）树的每个节点有唯一的标号，交换两个节点认为是不同的树。显然规模为 n 的带标号树数量是无标号树数量的 $n!$ 倍。

1. 递归边界（平凡树和空森林）为 $T(1) = T(0) = F(0) = 1$ 。
2. 因为树的根节点的子树构成了一个森林，所以添加与减少一个根节点，可以在规模为 n 的树和规模为 $n - 1$ 的森林之间建立一个一一映射。因此 $T(n) = F(n - 1)$ 。
3. 因为森林是树的线性表，所以森林中除第一棵树外，其余的部分也构成一棵森林（可能是空森林）。这个做法是我们非常熟悉的线性表头部减治。设第一棵树的规模为 k ，则可以得到

$$F(n) = \sum_{k=1}^n T(k)F(n-k)$$

我们结合（2）（3）两点，消去 $T(n)$ ，就发现了一个在《栈》章里多次见过的递归方程：

$$F(n) = \sum_{k=1}^n F(k-1)F(n-k)$$

结合 $F(0) = 1$ ，这也就意味着 $F(n) = \text{Catalan}(n)$ 。进而可以得到， $T(n) = \text{Catalan}(n-1)$ 。

7.4.2 实验：树和森林与栈的映射 *

在《栈》章中我们讨论过，计数问题有两种基本方法：递归方程法和一一映射法。既然在上一小节中，通过递归方程法可以解出树和森林的计数是卡特兰数，那么也就自然地能够想到，我们可以在树（森林）的形态和栈操作序列之间建立一一映射。

我们可以回忆典型的出栈序列问题：合法括号序列的问题。在合法括号序列问题中， $(S)T$ 被拆分为了 S 和 T 两个部分，这一做法和上一小节里对森林的拆分

如出一辙。森林被拆分为了“第一棵树的子树森林”和“除第一棵树之外的树组成的森林”，而与 $(S)T$ 中的括号相对应的，也就是上述两部分之外的部分，也就是第一棵树的根节点。

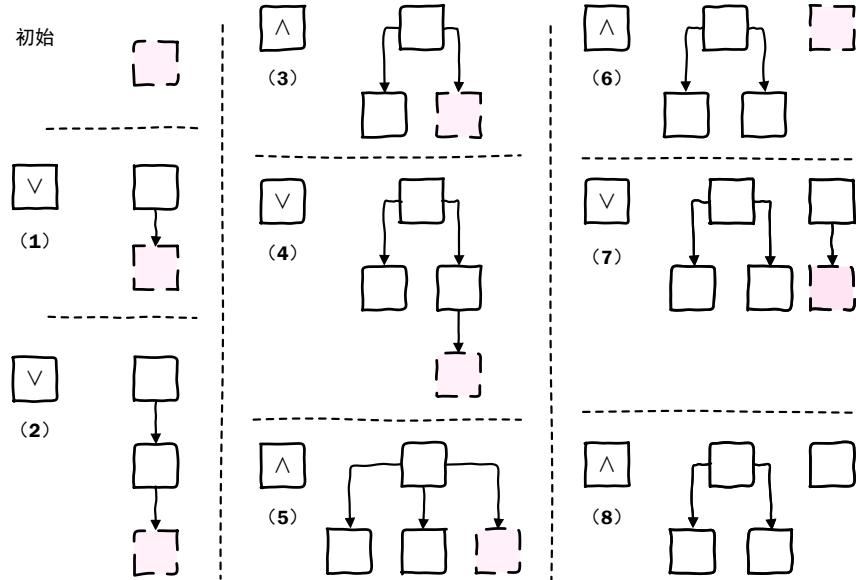


图 7.8 栈操作序列转换为森林的过程

合法括号序列也就一一对应了操作序列，所以操作序列和无标号森林之间建立了一一对应的关系。图7.8展示了操作序列 $v \vee \wedge v \vee \wedge \wedge v \wedge$ 转换为森林的过程，它遵循以下的步骤：

1. 初始时，指针（红色虚线节点）位于第 0 层（树根层）。
2. 遇到 \vee ，在指针处生成一个节点，指针转移到新生成节点的子代位置处（准备生成“第一棵树的子树森林”）。
3. 遇到 \wedge ，指针移动到当前位置亲代的下一个姊妹位置处（准备生成“除第一棵树之外的树组成的森林”）。

在操作序列的基础上引入入栈序列，我们就可以让每次入栈的元素等于树的对应节点上存储的元素，从而在栈和带标号森林之间，建立了如下的一一映射：

1. 空操作序列对应空森林。
2. 非空序列 $v(a) O_1 \wedge(a) O_2$ 对应的森林中， a 是第一棵树的树根， O_1 对应第一棵树的子树森林， O_2 对应除第一棵树之外的树组成的森林。

在建立一一映射的过程中，我们可以发现，尽管森林是树的线性表，但是森林中的各个树的地位不是平等的。那么，由于树中每个节点的子树都组成了森林，

所以树中每个节点的子树地位也不是平等的。这一不平等性会带来对称性的损失。我们可以自然地想到，如果在森林中，让第一棵树的树根放在中间，第一棵树的子树森林放在一边，除第一棵树之外的树组成的森林放在另一边，那么就可以形成很好的对称性。这一思路能够导出下一章所介绍的二叉树，本章暂时不进行讨论。

基于上述一一映射，我们也可以定义森林的遍历方式。假设非空森林的第一棵树树根为 a ，第一棵树的子树森林为 F_1 ，第一棵树之外的树组成的森林为 F_2 ，那么，按照 $a \rightarrow F_1 \rightarrow F_2$ 的次序递归地进行遍历，称为森林的**先序遍历**；按照 $F_1 \rightarrow a \rightarrow F_2$ 的次序递归地进行遍历，称为森林的**中序遍历**。这里的先序和中序同样是依照 a 在 F_1 和 F_2 间的位置得名。根据上面的分析可以知道，森林的先序序列等于其对应栈的入栈序列，森林的中序序列等于其对应栈的出栈序列。

树和森林之间只有一个根节点之差。您可以很容易地证明下面的命题：

定理 7.3： 树的先序遍历，等于访问根节点，再对子树森林做先序遍历。树的后序遍历，等于对子树森林做中序遍历，再访问根节点。

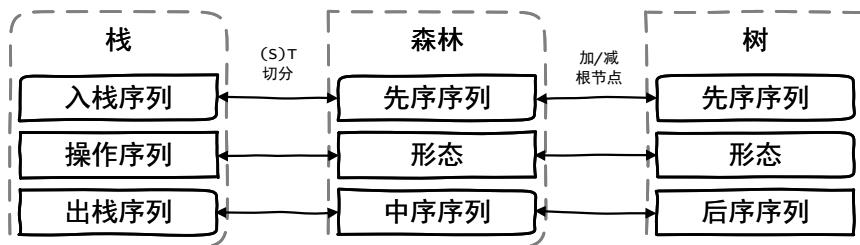


图 7.9 树、森林和栈的对应关系

于是，树的先序序列和后序序列，也和栈的入栈序列、出栈序列之间建立了联系。树、森林和栈的对应关系如图7.9所示。示例代码中提供了一个栈操作序列向树的转换，并验证了入栈序列、出栈序列和树的相应遍历序列的对应关系。代码可以在 *Stack2Tree.cpp* 中找到，转换的核心代码如下所示。

```

1 Tree<T> operator()(size_t n) override {
2     if (n == 0) {
3         return Tree<T> {};
4     }
5     Tree<T> tree { T {} };
6     auto pos { T.root() };
7     auto vec { randop(n-1, '(', ')') };
8     for (auto c : vec) {
9         if (c == '(') {
10             pos = tree.insertAsChild(pos, T {});
11         } else {
12             pos = pos->parent();
13         }
14     }
15     return tree;
}

```

需要特别说明的是，由于历史原因，在树和森林的遍历定义上存在多种观点。本书所采用的树和森林的先序、中序、后序遍历定义，是我认为较为直观的定义方式，和您所采用的教材上的定义方式可能会有所不同（一些教材会从二叉树的角度出发进行定义，见《二叉树》章）。当您在考试中解答相关问题时，请您以教材上的定义为准。

7.4.3 树的逆线性化*

我们知道在栈中，入栈序列、操作序列和出栈序列中的任意两者都可以确定第三者；所以在树和森林中也是如此。知道形态的情况下，任一遍历序列都可以为每个节点标上号，从而可以导出另一个遍历序列。更加重要的结论是，如果知道树的先序序列和后序序列，就能确定树的形态（森林的情况相似，不做赘述）。

将半线性的树转换为遍历序列的做法称为线性化，反过来，根据遍历序列确定树的半线性结构的过程称为逆线性化。用上一小节的技术，借助栈作为跳板去逆线性化是可行的，只是略显复杂。尤其是考试中在纸上做逆线性化时，我们需要一种更简单、更快速的方法。本节介绍一种直观的逆线性化方式，便于在纸上演算进行。该方法并非为我创造，但我看到它的时间较为久远，现在找不到出处，请知道出处的读者联系我注明引用。

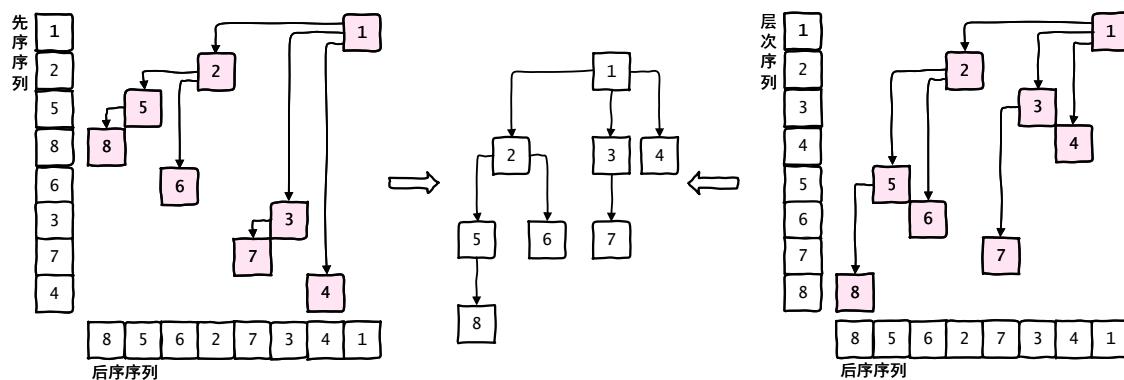


图 7.10 树的逆线性化方法

假设先序序列为 12586374，后序序列为 85627341，图展示了逆线性化为一棵树的过程。具体的步骤如下：

1. 画出一个表格，纵轴上从上到下是先序序列，横轴上从左到右是后序序列。
2. 用横纵轴定位每个节点。
3. 每个节点的右上方区域为其祖先，左下方区域为其后代。其中，右上方最近的节点为其亲代，相连接即可。

下面对该方法的正确性进行说明，如果您不感兴趣，也可以跳过此部分。这一方法利用了先序和后序序列的性质：如果节点 u 是节点 v 的祖先，那么 u 在先序序列上的位置在 v 之前，在后序序列上的位置在 v 之后。这一性质可以保证一个节点的祖先都在其右上方区域，后代都在其左下方区域。

另一方面，如果 u 和 v 之间没有祖先-后代关系，设 u 和 v 的 LCA 为 a ，则先序、后序序列中， u 在 v 前等价于在 a 的子代中， u 的祖先在 v 的祖先左侧。也就是说，当 u 和 v 没有祖先-后代关系的情况下， u 要么在先序、后序序列上都在 v 前，要么在先序、后序序列上都在 v 后，也就是说只会出现在左上方或右下方区域。综合上面两点，就保证了每个节点的右上方区域都是祖先，左下方区域都是后代。由于一个节点的祖先成一条链，所以它右上方最近的节点在这条链上也离它最近。因此，右上方最近的节点就是它的亲代。

您可能会好奇层次序列的情况。由于层次序列和队列相关，而先序、后序序列和栈相关，所以二者之间有比较大的差异。层次序列和先序序列是无法进行树的逆线性化的，您会发现，使用这两个序列甚至无法分辨链和菊花图。层次序列和后序序列则同样可以通过表格法进行逆线性化，如图中右图所示。

当纵坐标从先序序列变为层次序列时，由于层次序列中祖先仍然一定在后代之前，所以祖先一定在右上方。但是，因为层次序列和后序序列不具有那样良好的对称关系，所以右上方的不一定是祖先。在层次序列中，只有 u 和 v 同层的条件下，可以通过 LCA 处的左右关系判断 u 和 v 的前后关系。所以，我们只能保证同层的节点在左上方或者右下方。

考虑一个节点 v 右上方的节点，对于在它上层的某个节点 p ，设 v 的祖先中与其同层的节点为 u 。如果 p 在 u 左侧，那么由于 p 和 u 的 LCA 同时也是 p 和 v 的 LCA，我们可以通过 LCA 确定 p 在 v 的左上方。而如果 p 在 u 的右侧，那么 p 在 u 的右下方，但仍位于 v 的右上方。所以我们在确定亲代的时候，应当选取“右侧第一个在上方的节点”，才能排除掉亲代在右侧的姊妹。

7.5 本章小结

半线性结构是《数据结构》中的重点和难点章节。相对于线性结构来说，半线性结构将元素位置的全序关系弱化为偏序关系，将列表的单个后向指针扩展为多个子代指针，从而取得了比线性结构更好的灵活性。半线性结构的性质和此前学习的线性结构有显著的区别，本章从最基本的半线性结构树和森林出发，介绍了半线性结构的特点，主要包括三个方面：

1. 半线性结构的递归定义方法。借助半线性结构的递归定义，可以衍生出设计

分治算法的思路。

2. 半线性结构的形态多样性。在设计算法时，需要考虑兼顾不同形态。
3. 半线性结构的线性化（遍历）和逆线性化（重建）技术。

本章的主要学习目标如下：

1. 您了解到树可以划分为层次和子树。树是按子树递归定义的，而不是按层次递归定义的。
2. 您学会了将一棵树拆分为根节点及其子树森林，并对子树森林中的每棵子树进行分治的做法。以及将一棵森林拆分为第一棵树的根节点、第一棵树的子树森林以及其余树组成的森林的分治做法。
3. 您学会了利用 LCA 分析树上两个节点关系的问题。
4. 您了解到链和菊花图是特殊的树，会对应树上算法的极端情况。
5. 您了解到树和森林与栈之间存在一一映射关系，掌握了逆线性化的方法。
6. 您了解到可以利用势能分析法对复杂的多重循环问题进行时间复杂度分析。

在实际程序设计中，往往会在树和森林的基础上，为半线性结构附加更强的条件，以适应不同的应用场景。这将在后续的几章中介绍。本章的内容则偏向于理论，本章所介绍的许多知识，比如深度、高度、子树等概念、三种遍历方法、树和栈的对应关系以及逆线性化方法等，都是后续研究其他半线性结构（如二叉树）的基础。

第8章 二叉树

8.1 从树到二叉树

8.1.1 有序树和二叉树

在二进制计算机中，“2”具有特殊而重要的地位，比如，每次比较都有2种结果，分别可以对应**if**分支和**else**分支。如果我们有多于2个的分支，则通常需要多于1次的比较，比如，下面的4分支代码需要3次比较。

```

1 if /* compare 1 */ {
2     /* branch 1 */
3 } else if /* compare 2 */ {
4     /* branch 2 */
5 } else if /* compare 3 */ {
6     /* branch 3 */
7 } else {
8     /* branch 4 */
9 }
```

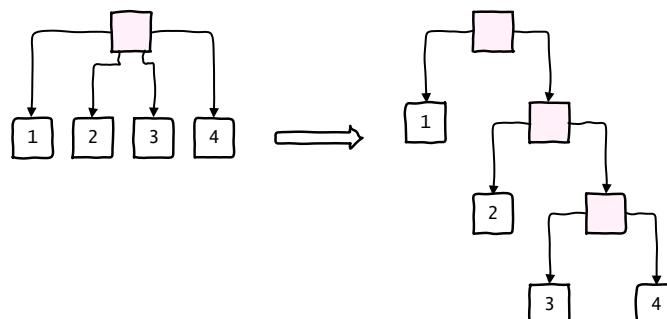


图8.1 从过程上分析的树和从操作上分析的二叉树

从过程上分析上面这段代码时，可以认为它将程序分为四个可能的分支进行处理，也就是图8.1的左图所示的树。红色节点表示判断过程，1、2、3、4四个节点表示不同的分支。而从操作上分析上面这段代码时，则可以得到图8.1的右图，红色节点被分成了3次不同的判断操作。这种树被称为二叉树（binary tree），它具有如下特征：

1. 每个节点至多有2个子节点。
2. 每个节点的左子代和右子代是不同的分支。

其中，性质（2）尤其重要。在普通的树中，虽然子树之间也有先后次序（即**有序树**），但这仅限于子树存在的情况下，空树是在先后次序里不占位置的。在树的实现中，子代指针的线性表**children**里也是不允许存在空指针的。但在二叉树

中，一个节点如果只有一棵子树，那么该子树是左子树或右子树，这是两种不同的情况。这个问题正如在**if-else**的结构中，下面两段代码的功能是完全不同的：

```

1 // if-only (left child only)
2 if /* compare */ {
3     /* branch */
4 } else {}
5
6 // else-only (right child only)
7 if /* compare */ {} else {
8     /* branch */
9 }
```

从**if-else**判断结构的例子中我们可以看出，二叉树天然地适合对二进制计算机的许多行为进行建模，因此相比于普通的树，二叉树在计算机领域具有非常广泛的应用。

由于您应当在《树》章里已经对半线性结构有了一定的了解，本章将侧重于介绍二叉树的“二叉性”带来的特殊性质。本章的内容仍然以理论知识为主。在树与二叉树的基础上进一步增加条件，可以衍生出更多的数据结构，从而适用于不同的应用场景；这些衍生的数据结构将在后面的章节中介绍。

8.1.2 二叉树的节点

二叉树的节点设计，和树的节点相似。树的子代指针线性表，在二叉树的节点中变为了两个子代指针，分别用于指示左子代和右子代，如下所示。

```

1 template <typename T>
2 class BinaryTreeNode {
3     T m_data;
4     BinaryTreeNodePos<T> m_parent { nullptr };
5     BinaryTreeNodeInst<T> m_left { nullptr };
6     BinaryTreeNodeInst<T> m_right { nullptr };
7 public:
8     BinaryTreeNode() = default;
9     BinaryTreeNode(const T& data) : m_data(data) {}
10    virtual ~BinaryTreeNode() { /* ... */ }
11    T& data() { return m_data; }
12    BinaryTreeNodeInst<T>& left() { return m_left; }
13    BinaryTreeNodeInst<T>& right() { return m_right; }
14    BinaryTreeNodePos<T>& parent() { return m_parent; }
15    bool isRoot() const { return m_parent == nullptr; }
16    bool isLeaf() const { return m_left == nullptr && m_right == nullptr; }
17    bool isLeftChild() const { return m_parent != nullptr &&
18        m_parent->left() == this; }
19    bool isRightChild() const { return m_parent != nullptr &&
        m_parent->right() == this; }
```

在上面的示例代码中没有提到或没有实现的方法，请您自己参考在《树》章中已经学到的设计技巧进行设计。您应当能够实现下列的方法：

1. 通过向亲代查找，实现深度 `depth` 方法。
2. 通过分治技术，实现高度 `height` 和规模 `size` 方法。
3. 通过设计合适的析构次序，实现无递归的析构函数。

和《树》章类似，可以实现二叉树的插入、查找和删除等方法，因为和普通的树几乎完全相同，在本章中不再赘述。

8.1.3 长女-次妹表示法

我们首先考虑二叉树的计数问题。设规模为 n 的二叉树的形态数为 $B(n)$ ；对于一个规模为 n 的二叉树，假设它树根的左子树规模为 k ，则右子树的规模为 $n-1-k$ ，因此，可以得到下面的递归方程：

$$B(n) = \sum_{k=0}^{n-1} B(k)B(n-1-k)$$

这又是一个典型的 Catalan 数递归方程。结合递归边界 $B(0) = 1$ ，可以直接得到 $B(n) = \text{Catalan}(n)$ 。因此，规模为 n 的二叉树的数量，等于规模为 n 的森林的数量、规模为 $n+1$ 的树的数量、长度为 $2n$ 的栈操作序列的数量。

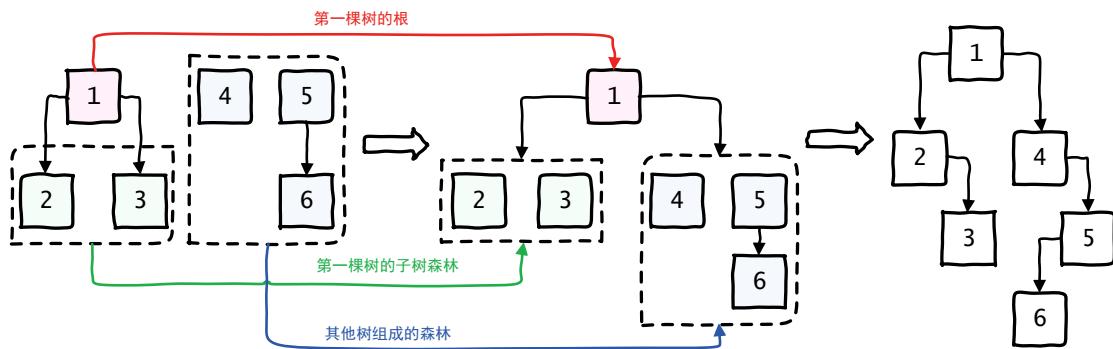


图 8.2 森林转换为二叉树

我们会自然想到在二叉树和树、森林以及栈之间建立一一映射关系。回想森林的递归定义，森林可以被拆分为“第一棵树的子树森林”和“除第一棵树之外的树组成的森林”，这两者是地位不平等的，正如二叉树的左子代和右子代是不同的。因此，我们可以建立如下的映射关系，如图8.2所示。

1. 空森林对应空二叉树。对于非空的森林，第一棵树的树根对应二叉树的树根。
2. 第一棵树的子树森林，对应二叉树的左子代（递归对应）。
3. 除第一棵树之外的树组成的森林，对应二叉树的右子代（递归对应）。

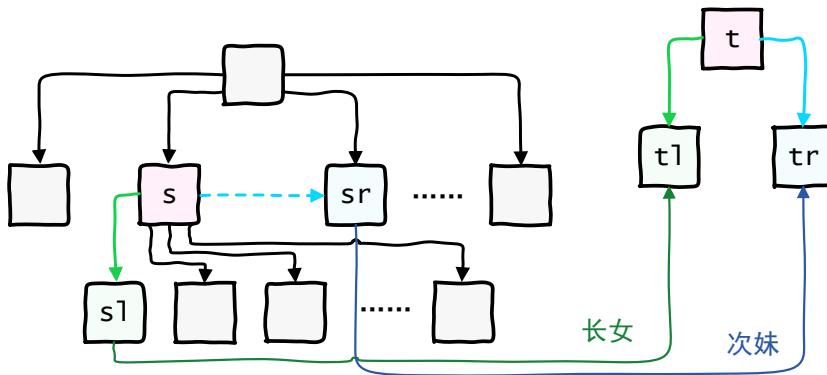


图 8.3 长女-次妹表示法

如图8.3所示,对于森林中的一个节点*s*,如果它在二叉树中的对应节点为*t*,那么:

1. *t* 的左子代, 对应 *s* 的第一个子代。
2. *t* 的右子代, 对应 *s* 的下一个姊妹。如果 *s* 是森林中某一棵树的树根, 则它的下一个姊妹定义为下一棵树的树根。

因此,这种对应方法可以被称为“第一子代 + 下一姊妹”表示法,在本书中将其简称为**长女-次妹表示法**。因为树可以认为是“只有一棵树的森林”,所以长女-次妹表示法也可以用于构造树到二叉树的映射。因为树的根节点没有姊妹,所以树通过长女-次妹表示法映射到的二叉树,其树根必然没有右子代。

8.1.4 二叉树和栈的映射 *

借助长女-次妹表示法,就可以将树和森林转换为二叉树进行研究。进一步地,我们可以将长女-次妹表示法和栈到森林的转换方法相结合,从而得到栈和二叉树的对应关系,如图8.4所示。

1. 初始时,指针(红色虚线节点)位于第0层(树根层)。
2. 遇到 \vee ,在指针处生成一个节点,指针转移到左子代的位置处。
3. 遇到 \wedge ,指针向左上方移动到底(移动到某个左子代为止),然后转移到右姊妹的位置处。

图8.4中展示的操作序列转换为二叉树的过程,您可以先将其通过《树》章中介绍的方法转换为森林,再通过长女-次妹表示法转换为二叉树,感受这些数据结构之间的对应关系。请您仿照7.4.2节中的方法,自行写出栈操作序列转换为二叉树的代码,并和下面的示例进行比较。

```

1 BinaryTree<int> buildTree(string op) {
2     BinaryTree<int> T {};
3     auto pos { T.root() };

```

```

4     int cur { 0 };
5     bool left { true };
6     for (char c : op) {
7         if (c == '(') {
8             if (T.empty()) {
9                 pos = T.insertAsRoot(cur++);
10            } else {
11                if (left) {
12                    pos = T.insertAsLeftChild(pos, cur++);
13                } else {
14                    pos = T.insertAsRightChild(pos, cur++);
15                }
16            }
17            left = true;
18        } else {
19            if (left) {
20                left = false;
21            } else {
22                while (pos->isRightChild()) {
23                    pos = pos->parent();
24                }
25                pos = pos->parent();
26            }
27        }
28    }
29    return T;
30 }

```

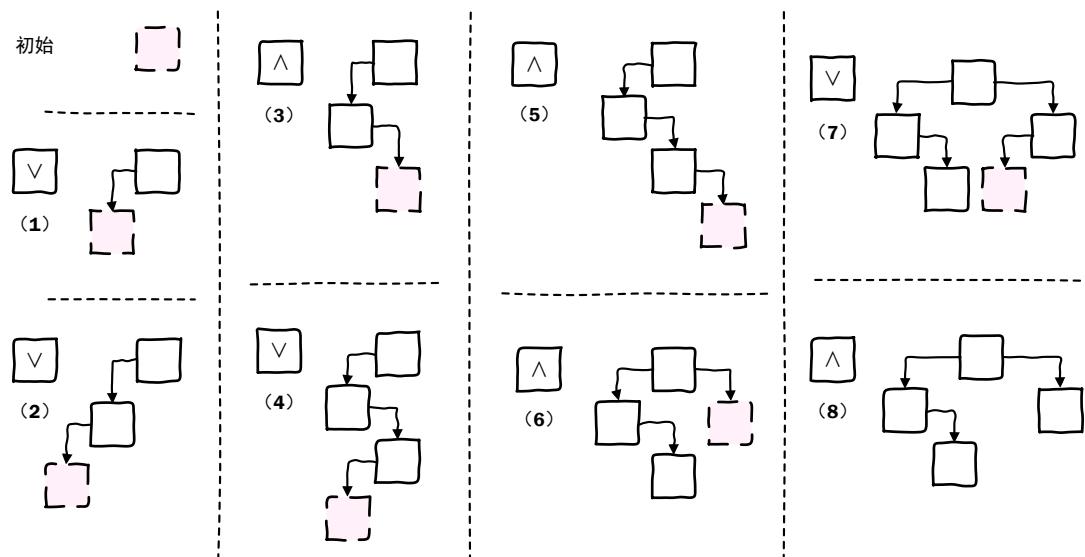


图 8.4 栈操作序列转换为二叉树的过程

8.2 二叉树的遍历

8.2.1 利用半线性递归进行遍历

在图7.9中展示了树、森林和栈的对应关系。除了栈的操作序列可以和树、森林的形态形成一一对应之外，入栈序列、森林的先序序列和树的先序序列也可以形成一一对应，出栈序列、森林的中序序列和树的后序序列同样能形成一一对应。那么，当二叉树加入到这个对应关系之后，这两个序列又分别对应二叉树的什么遍历方法呢？

我们可以承袭《树》章中的方法，定义二叉树的四种遍历方法：先序遍历、中序遍历、后序遍历和层次遍历。其中，前三种遍历方法可以以下列方式统一定义：

1. 将非空二叉树拆分为根节点 N ，左子树 L 和右子树 R 三部分。
2. 定义 NLR 次序的遍历为**先序遍历**， LNR 次序的遍历为**中序遍历**， LRN 次序的遍历为**后序遍历**。
3. N 、 L 、 R 有 $3! = 6$ 种排列方式。所以反过来，可以定义 NRL 次序的遍历为**逆后序遍历**， RNL 次序的遍历为**逆中序遍历**， RLN 次序的遍历为**逆先序遍历**^[7]。这三种遍历和（2）中定义的三种常见的遍历具有良好的对称性。

先序遍历的一个递归实现如下所示，只需要移动 `visit` 语句的位置，就可以得到中序遍历和后序遍历的示例代码。

```

1 // BinaryTreePreOrderTraverse
2 void traverse(BinaryTreeNodeConstPos<T> p, std::function<void(
3     BinaryTreeNodeConstPos<T>)> visit) override {
4     if (!p) {
5         return;
6     }
7     visit(p);
8     traverse(p->left(), visit);
9     traverse(p->right(), visit);
}

```

结合长女-次妹表示法和森林遍历的定义，可以直接得到，森林的先序序列与所对应二叉树的先序序列相同，而森林的中序序列也对应着二叉树的中序序列。这样就补全了栈、树、森林、二叉树之间的对应关系图，如图所示。您可以看出，在二叉树和树的对应关系里，二叉树的先序序列同样对应了树的先序序列，但二叉树的中序序列对应着树的后序序列。在一些教材中站在二叉树的视角出发进行定义^[5]，因此会将二叉树的中序遍历所对应的树的遍历，也称为“中序遍历”，尽管在这种遍历方式中树根总是在所有子树之“后”被遍历。本书中为了理解直观没有采用这种定义方式，仍然定义这种遍历为树的后序遍历。同时，本书鼓励您避免在二叉树和树之间做直接映射，而是以森林作为跳板进行。

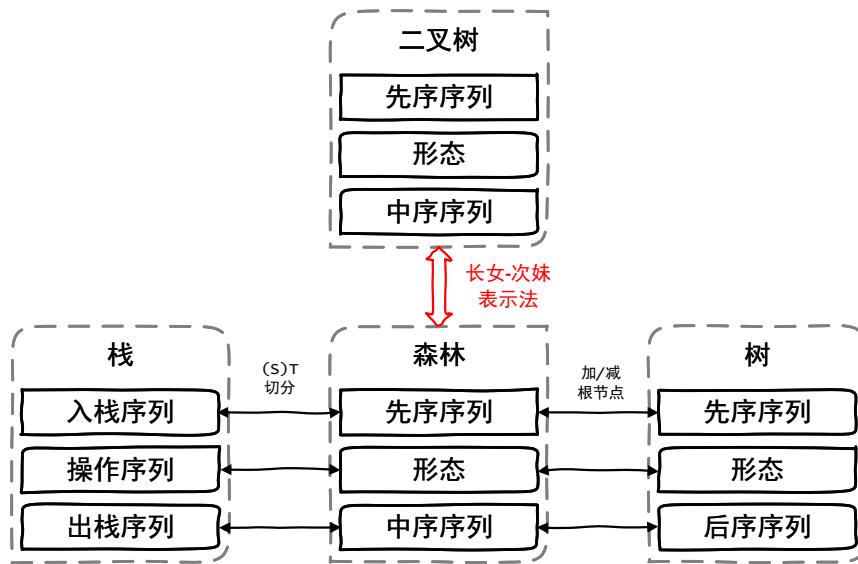


图 8.5 栈、树、森林和二叉树的对应关系

层次遍历的定义和树的层次遍历相同。因为长女-次妹表示法会将树（森林）中的姊妹节点变成一条右链（由右子代指针连接的链），层次关系受到了破坏，所以二叉树的层次序列和树（森林）的层次序列没有直接的关联关系。和树一样，二叉树的层次遍历也需要借助队列实现。

```

1 // BinaryTreeLevelOrderTraverse
2 void traverse(BinaryTreeNodeConstPos<T> p, std::function<void(
3     BinaryTreeNodeConstPos<T>)> visit) override {
4     Queue<BinaryTreeNodeConstPos<T>> Q { p };
5     while (!Q.empty()) {
6         if (auto node { Q.dequeue() }; node) {
7             visit(node);
8             Q.enqueue(node->left());
9             Q.enqueue(node->right());
10        }
11    }

```

和树一样，如果将层次遍历代码中的队列变为栈，并倒转右子代和左子代入栈的次序，就可以得到一种基于栈的先序遍历实现。

```

1 // BinaryTreePreOrderTraverseSemilinear
2 void traverse(BinaryTreeNodeConstPos<T> p, std::function<void(
3     BinaryTreeNodeConstPos<T>)> visit) override {
4     Stack<BinaryTreeNodeConstPos<T>> S { p };
5     while (!S.empty()) {
6         if (auto node { S.pop() }; node) {
7             visit(node);
8             S.push(node->right());
9             S.push(node->left());
10        }

```

```
10 }
11 }
```

逆后序遍历的遍历次序和后序遍历相反（NRL 和 LRN），同时它在形式上和先序遍历有相似性（NRL 和 NLR），这两方面的对称性使得它可以作为先序遍历和后序遍历之间的“桥梁”（逆先序遍历类似，不再赘述）。因为逆后序遍历和先序遍历具有形式相似性，只是 L 和 R 位置的不同，所以将上面的代码中 L 和 R 的入栈次序交换，就可以得到逆后序遍历。而为了得到后序遍历，只需要将逆后序遍历得到的序列颠倒。栈的 LIFO 特性恰好可以用于颠倒序列，因此，我们可以增加一个辅助栈，借用逆后序遍历作为跳板，得到一种基于栈的后序遍历实现，示例代码如下所示。

```
1 // BinaryTreePostorderTraverseSemilinear
2 void traverse(BinaryTreeNodeConstPos<T> p, std::function<void(
    BinaryTreeNodeConstPos<T>)> visit) override {
3     Stack<BinaryTreeNodeConstPos<T>> S { p };
4     Stack<BinaryTreeNodeConstPos<T>> St;
5     while (!S.empty()) {
6         if (auto node { S.pop() }; node) {
7             St.push(node);
8             S.push(node->left());
9             S.push(node->right());
10        }
11    }
12    while (!St.empty()) {
13        visit(St.pop());
14    }
15 }
```

基于栈的版本并不会降低递归版本的空间复杂度 $\Theta(h)$ 。全部的 $Catalan(n)$ 棵二叉树，其平均树高大约为 $2\sqrt{\pi n}$ ^[25]（值得一提的是，这个结果两倍于树的平均高度^[26]）。这一结论的证明需要较强的数学功底，因此不进行展开，一般只需要记忆其为 $\Theta(\sqrt{n})$ 的即可。

8.2.2 利用线性递归进行遍历

由于森林可以被拆分为第一棵树的根节点，以及“第一棵树的子树森林”、“其余树组成的森林”两个规模较小的森林，所以森林通过长女-次妹表示法转换为二叉树的过程是非常自然的，我们也可以直接地得到森林的先序（中序）遍历和二叉树的先序（中序）遍历之间的对应关系。

而树转换为二叉树时，遍历序列的对应关系则不那么直接。树的先序遍历形式和上一小节展示的二叉树先序遍历既有相同，又有不同。一方面，树的先序遍历和二叉树的先序遍历，都是先访问当前节点，再从左到右依次访问每个子树；另一

方面，树的先序遍历会产生 d 个递归调用，其中 d 是当前节点的度，而二叉树的先序遍历只会产生 2 个递归调用。

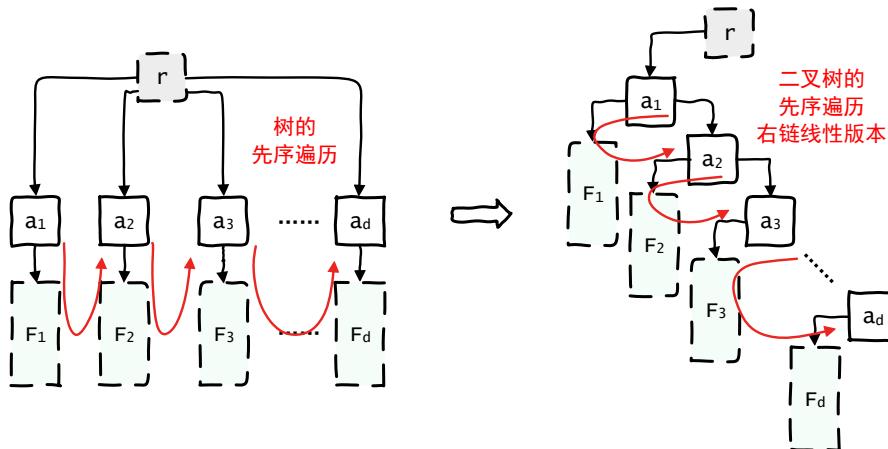


图 8.6 从树的先序遍历得到二叉树先序遍历的线性形式

您会自然地想到，如果将树的先序遍历通过长女-次妹表示法转换到二叉树上，可以得到二叉树先序遍历的另一种形式。如图8.6所示，树根 r 的子树森林中，各个树的根节点 a_1, a_2, \dots, a_d ，会被映射为二叉树的右链；而 a_j 的子树森林，则被映射为二叉树上相应节点的左子树。基于这一思想实现的二叉树先序遍历如下所示。

```

1 // BinaryTreePreOrderTraverseLinearRecursive
2 void traverse(BinaryTreeNodeConstPos<T> p, std::function<void(
    BinaryTreeNodeConstPos<T>)> visit) override {
3     while (p != nullptr) {
4         visit(p);
5         traverse(p->left(), visit);
6         p = p->right();
7     }
8 }
```

这种思路称为**线性递归**，因为事实上将二叉树当做了线性表，线性表中的每个元素是右链上的一个节点及其左子树。如果在长女-次妹表示法下，这就是将森林当做树的线性表，所以事实上这也是一种非常自然的思路。相应地，上一小节那种基于二叉树自身的“二叉”性质，将二叉树拆分为根节点、左子树和右子树从而递归分析的做法可以称为**半线性递归**。

下面将上面的递归版本先序遍历改写为基于栈的非递归版本。回顾《树》中基于栈的树的先序遍历，我们需要将子代逆序入栈；因此，在子代被映射为右链之后，我们需要自下而上地将右链入栈。自下而上的过程可以用一个辅助栈实现，如下所示。这种使用辅助栈的做法，对于没有亲代指针、采用“子代表示法”的二叉树同样可用。在“亲代 + 子代表示法”的二叉树里，也可以将其改为利用亲代指针向上的做法，请您自己完成该过程。

```

1 template <typename T>
2 class BinaryTreePreOrderTraverseLinearRight : public
3   AbstractBinaryTreeTraverse<T> {
4     void pushRightChain(Stack<BinaryTreeNodeConstPos<T>>& S,
5       BinaryTreeNodeConstPos<T> p) {
6       Stack<BinaryTreeNodeConstPos<T>> St;
7       while (p) {
8         St.push(p);
9         p = p->right();
10      }
11    }
12  }
13 public:
14   void traverse(BinaryTreeNodeConstPos<T> p, std::function<
15     void(BinaryTreeNodeConstPos<T>)> visit) override {
16     Stack<BinaryTreeNodeConstPos<T>> S;
17     pushRightChain(S, p);
18     while (!S.empty()) {
19       auto node { S.pop() };
20       visit(node);
21       pushRightChain(S, node->left());
22     }
23   };

```

可以看出，因为需要自下而上进栈，这种做法显得比较复杂，但可以为我们后续的设计提供启发。以下将从这个代码出发构造中序遍历和后序遍历，并导出一个更简单的先序遍历实现。

在上面的代码中，我们自下而上地让右链入栈，同时每次出栈时，先访问当前节点，再访问左子树。我们想要避免“自下而上”这个别扭的过程。如果将自下而上改为自上而下，其他不变，那么就相当于在树的先序遍历中，逆转了各个子树的遍历次序。您一定还记得，二叉树的先序遍历 NLR，逆转子树遍历次序可以得到逆后序遍历 RLN。类似地，树的先序遍历逆转次序之后，也可以得到树的逆后序遍历。我们已经知道，树的后序遍历对应二叉树的中序遍历，因此树的逆后序遍历也就对应了二叉树的逆中序遍历。于是可以得到下面的结论：如果将上面的代码改为自上而下地入栈，其他不变，则得到的代码是逆中序遍历。

逆中序遍历和中序遍历的关系是特殊的。LNR 和 RNL 之间既是**顺序颠倒**的关系，又是**左右交换**的关系。在这里，如果要从逆中序遍历得到中序遍历，既可以用上一小节的方法使用一个辅助栈解决，也可以简单直接地交换左右解决：即把挂载左子树的右链，改为挂载右子树的**左链**。请您自己实现两种方案。下面的示例代码展示了交换左右的方案。

```

1 template <typename T>
2 class BinaryTreeInOrderTraverseLinear : public
3   AbstractBinaryTreeTraverse<T> {
4     void pushLeftChain(Stack<BinaryTreeNodeConstPos<T>>& S,
5       BinaryTreeNodeConstPos<T> p) {
6       while (p) {
7         S.push(p);
8         p = p->left();
9       }
10    }
11  public:
12    void traverse(BinaryTreeNodeConstPos<T> p, std::function<
13      void(BinaryTreeNodeConstPos<T>) > visit) override {
14      Stack<BinaryTreeNodeConstPos<T>> S;
15      pushLeftChain(S, p);
16      while (!S.empty()) {
17        auto node { S.pop() };
18        visit(node);
19        pushLeftChain(S, node->right());
20      }
21    }
22  };

```

接下来考虑后序遍历的问题。从先序遍历出发，我们可以用逆后序遍历或者逆先序遍历作为跳板，得到后序遍历的实现。上一小节讨论了逆后序遍历，这里从逆先序遍历出发。首先，我们颠倒先序遍历的结果得到逆先序遍历。用一个辅助栈实现颠倒的方法您应当已经很熟悉，另一种方法是将每个步骤的次序都逆转。在先序遍历中，我们自下而上地让右链进栈，每次出栈时先访问当前节点，再访问左子树；因此在后序遍历中，我们可以自上而下地让左链进栈，每次出站时先访问右子树，再访问当前节点，这样就得到了逆先序遍历。接着，我们将逆先序遍历的 L 和 R 交换，变为挂载右子树的左链，这样就得到了后序遍历。

请您从上面基于右链的先序遍历出发，沿着上面的思路实现基于左链的后序遍历，并和下面的示例代码进行对比。请注意这里需要区分“上一个被访问的节点”来自左子树还是右子树。如果来自左子树，则需要先访问右子树再访问当前节点；如果来自右子树，则直接访问当前节点即可。

```

1 template <typename T>
2 class BinaryTreePostorderTraverseLinear : public
3   AbstractBinaryTreeTraverse<T> {
4     void pushLeftChain(Stack<BinaryTreeNodeConstPos<T>>& S,
5       BinaryTreeNodeConstPos<T> p) {
6       while (p) {
7         S.push(p);
8         p = p->left();
9       }
10    }
11  };

```

```

9 public:
10    void traverse(BinaryTreeNodeConstPos<T> p, std::function<
11        void(BinaryTreeNodeConstPos<T>)> visit) override {
12        Stack<BinaryTreeNodeConstPos<T>> S;
13        BinaryTreeNodeConstPos<T> last { nullptr };
14        pushLeftChain(S, p);
15        while (!S.empty()) {
16            if (auto node { S.top() }; node->left() == last &&
17                node->right()) {
18                pushLeftChain(S, node->right());
19            } else {
20                last = S.pop();
21                visit(last);
22            }
23        }
24    };

```

最后我们回到先序遍历上。其实从上面对于中序和后序的分析已经很明显地可以发现，右链会导出逆中序（逆先序）遍历，要得到中序（后序）遍历，需要交换 L 和 R 讨论左链。右链会导出逆序的遍历，这个现象是有依据的。

1. 如果先访问右链上的节点，再递归地处理左子树，这样对于同一个节点来说，访问 R 在访问 L 之前，因而必然只会得到逆序的遍历。
2. 如果想要递归地处理左子树，再访问右链上的节点，那么在递归的进行过程中，就需要将右链上的节点暂存，最后再取出。这种做法相当于用一个辅助栈存储序列，最后再从辅助栈中取出。因此，得到的算法相当于先进行逆序的遍历，再利用辅助栈颠倒遍历次序。

于是，我们可以自然地想到，先序遍历也应该基于左链去设计。事实上，您会发现基于左链的先序遍历甚至服从尾递归的扩展形式，进而可以使用《栈》章里介绍的方法实现代码。请您分析先序遍历在左链上的访问次序，实现基于栈的遍历算法，并和下面的示例代码进行比较。

```

1 // BinaryTreePreOrderTraverseLinear
2 void traverse(BinaryTreeNodeConstPos<T> p, std::function<void(
3     BinaryTreeNodeConstPos<T>)> visit) override {
4     Stack<BinaryTreeNodeConstPos<T>> S { p };
5     while (!S.empty()) {
6         for (auto node { S.pop() }; node; node = node->left()) {
7             visit(node);
8             S.push(node->right());
9         }
10    }

```

本小节介绍的遍历算法都是利用线性递归实现的，它来自于树的遍历，或者

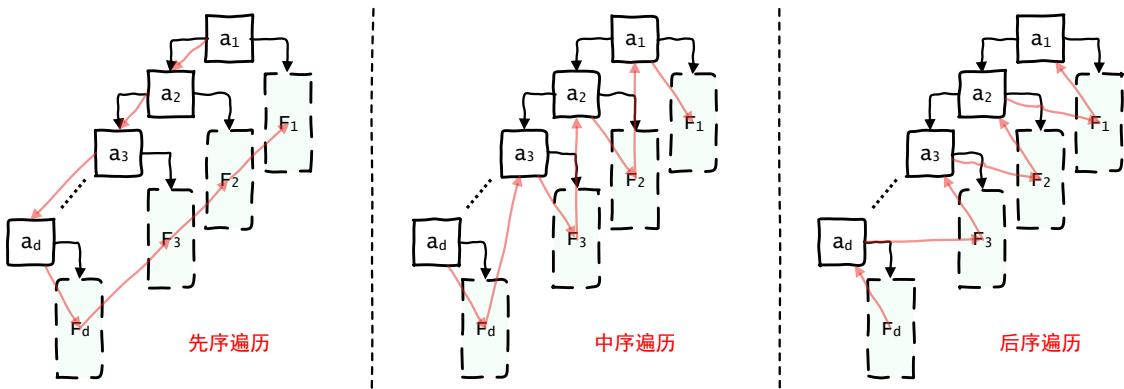


图 8.7 基于二叉树左链的三种遍历方法

将森林当做树的线性表的思想。通过分析我们发现，因为先序、中序和后序遍历中 L 在 R 之前，所以适合基于左链设计线性递归算法。图8.7展示了将二叉树作为左链线性表的情况下，先序、中序和后序遍历的遍历次序。阅读本小节之后，您应当学会如何基于栈设计二叉树的先序、中序和后序遍历，并能够对树、森林和二叉树的关系以及二叉树的遍历算法有更加深刻的认识。

8.2.3 利用迭代器进行遍历 *

无论是半线性递归还是线性递归，都是在从整体的角度出发进行分治，去建立二叉树的遍历算法：将“整棵”二叉树的遍历问题，转化为几个规模较小的二叉树的遍历问题。这一小节将从局部的角度出发去构建二叉树的遍历算法，这种思想称为 **迭代器** (iterator)。

我们在研究线性表的时候，将线性表的线性性质抽象了出来：线性表以某种“位置”定位元素，可以通过一个位置找到它后一个位置（单向列表找不到前一个位置），并且可以获取线性表的起始位置和终止位置。遍历序列是半线性数据结构的线性化，对于遍历序列，也可以刻画起始位置、终止位置、直接后继（直接前驱）位置这几个属性。以先序遍历为例，先序遍历的起始位置总是树根，而终止位置可以认为是 `nullptr`；那么，如果我们可以根据某个位置，定位到它在先序序列下的直接后继的位置，那么我们就可以利用这个先序遍历迭代器，和单向列表一样，不断查找“下一个节点”，从而实现先序遍历。一般地，我们可以定义树上的迭代算法如下：

```

1 template <typename T>
2 class AbstractBinaryTreeIterate {
3 public:
4     virtual BinaryTreeNodeConstPos<T> begin(

```

```

5     BinaryTreeNodeConstPos<T> p) = 0;
6     virtual BinaryTreeNodeConstPos<T> next(
7         BinaryTreeNodeConstPos<T> p) = 0;
8 }
```

下面依次讨论先序遍历、中序遍历和后序遍历中，应当如何实现`begin`和`next`。这些算法都可以在考试时画个图现场推导，而不需要进行记忆。首先，对于先序序列而言：

1. 如果节点有子代，则直接后继为其子代（左子优先）。
2. 如果节点无子代，则沿亲代指针向上回溯，途中如果有右姊妹，则直接后继为这一右姊妹。
3. 如果回溯到树根仍没有右姊妹，先序遍历结束，没有直接后继。

该算法的时间复杂度为 $O(d)$ ，其中 d 为当前节点的深度。示例代码如下所示。

```

1 // BinaryTreePreOrderIterate
2 BinaryTreeNodeConstPos<T> begin(BinaryTreeNodeConstPos<T> p)
3     override {
4         return p;
5     }
6     BinaryTreeNodeConstPos<T> next(BinaryTreeNodeConstPos<T> p)
7         override {
8             if (p == nullptr) {
9                 return nullptr;
10            }
11            if (auto& left { p->left() }; left) {
12                return left;
13            }
14            if (auto& right { p->right() }; right) {
15                return right;
16            }
17            while (p) {
18                if (auto sib { p->rightSibling() }; sib) {
19                    return sib;
20                }
21                p = p->parent();
22            }
23        return nullptr;
24    }
```

中序的情况：

1. 如果有右子，则向右下一步，然后沿左下到底（直到没有左子为止），即右子的最左侧通路终点就是直接后继。
2. 如果无右子，则向左上到底（直到遍历到左子或者根节点为止）。如果遍历到一个左子节点，则它的亲节点就是直接后继（右上一步）；如果遍历到根节点，则中序遍历结束，没有后继。

这个算法可以概括为：有右子，右下一步左下到底；无右子，左上到底右上一步。因为有可能向下到底或向上到底，所以这个算法的时间复杂度为 $O(d + h)$ ，其中 d 为当前节点的深度， h 为当前节点的高度。另外需要注意中序遍历和先序遍历不同，它的起始位置是**最左侧通路**（即从根节点出发的左链）的末尾节点，而不是整棵树的根节点。

```

1 template <typename T>
2 class BinaryTreeInOrderIterate : public
3     AbstractBinaryTreeIterate<T> {
4         BinaryTreeNodeConstPos<T> leftMost(BinaryTreeNodeConstPos<T
5             > p) {
6             while (p->left() != nullptr) {
7                 p = p->left();
8             }
9             return p;
10        }
11 public:
12     BinaryTreeNodeConstPos<T> begin(BinaryTreeNodeConstPos<T> p
13         ) override {
14         return leftMost(p);
15     }
16     BinaryTreeNodeConstPos<T> next(BinaryTreeNodeConstPos<T> p)
17         override {
18         if (p == nullptr) {
19             return nullptr;
20         }
21         if (auto& right { p->right() }; right) {
22             return leftMost(right);
23         } else {
24             while (p->isRightChild()) {
25                 p = p->parent();
26             }
27             return p->parent();
28         }
29     }
30 };

```

最后讨论后序的情况。后序遍历比较复杂，因为我们首先就会发现，后序遍历的起始位置就不好确定。中序遍历的起始位置是最左侧通路的末端，但后序遍历在最左侧到达末端之后，如果节点有右子代，它的右子代仍然在它前面被访问。因此，后序遍历的起始位置应当是**最长偏左侧通路**（沿着子代指针向下到底，左右子都有时，优先左子）的末端。以最长偏左侧通路为基础，后序遍历的直接后继可以用下面的方式确定。

1. 如果是根节点，遍历结束，没有直接后继。
2. 如果是右子，或者无右姊妹的左子，则直接后继是亲节点。
3. 如果是有右姊妹的左子，则直接后继是右姊妹出发最长偏左侧通路的末端节

点。

```

1 template <typename T>
2 class BinaryTreePostOrderIterate : public
3     AbstractBinaryTreeIterate<T> {
4         BinaryTreeNodeConstPos<T> childMost(BinaryTreeNodeConstPos<
5             T> p) {
6             while (!p->isLeaf()) {
7                 if (p->left()) {
8                     p = p->left();
9                 } else {
10                     p = p->right();
11                 }
12             }
13             return p;
14         }
15     public:
16         BinaryTreeNodeConstPos<T> begin(BinaryTreeNodeConstPos<T> p
17             ) override {
18             return childMost(p);
19         }
20         BinaryTreeNodeConstPos<T> next(BinaryTreeNodeConstPos<T> p)
21             override {
22             if (p == nullptr) {
23                 return nullptr;
24             }
25             if (auto sib { p->rightSibling() }; sib) {
26                 return childMost(sib);
27             } else {
28                 return p->parent();
29             }
30         }
31     };
32 }
```

后序遍历的时间复杂度为 $O(h)$, 其中 h 为当前节点的高度。

直接前驱的情况是相似的。比如, 先序遍历 (NLR) 的直接前驱也就是逆先序遍历 (RLN) 的直接后继, 而逆先序遍历和后序遍历 (LRN) 具有形式一致性, 只需要交换 L 和 R 即可。所以, 先序遍历的直接前驱 (逆先序遍历的直接后继) 算法, 和后序遍历的直接后继算法是对称的。类似可以得到另外两种遍历的直接前驱和相应直接后继算法的对称性。您可以自行实现三种逆序遍历的直接后继算法, 它们对应了三种正序遍历的直接前驱。

利用迭代器, 可以写出下面的遍历算法模板。

```

1 template <typename T, template <typename> typename It>
2     requires std::is_base_of_v<AbstractBinaryTreeIterate<T>, It
3         <T>>
4     class BinaryTreeIterativeTraverse : public
5         AbstractBinaryTreeTraverse<T> {
6             It<T> it;
7     public:
```

```

6   void traverse(BinaryTreeNodeConstPos<T> p, std::function<
7     void(BinaryTreeNodeConstPos<T>)> visit) override {
8     for (auto node { it.begin(p) }; node; node = it.next(
9       node)) {
10       visit(node);
11     }
12   };

```

上面的模板参数 `It` 可以传入先序遍历、中序遍历、后序遍历或者三种相应的逆遍历。虽然每一步求直接后继的时间复杂度和当前节点的深度和高度有关，但可以证明，上述迭代器形式的遍历和之前介绍的递归及迭代的遍历方法一样，时间复杂度都是 $\Theta(n)$ 。这一证明可以利用之前介绍的势能分析法完成，下面给出了先序遍历的证明过程，请您自己完成中序遍历和后序遍历的时间复杂度证明。

1. 定义每次调用 `begin` 或 `next` 为一次操作，总操作次数为 $\Theta(n)$ 。
2. 在第 i 次操作中，如果不是叶子，那么选取子代即可， $a_i = 1$ ；如果是叶子，设向上 k_i 层之后找到右姊妹（或到达根节点），则 $a_i = 1 + k_i$ 。
3. 定义势能 $\phi = h_T - d$ ，其中 h_T 为树高， d 为当前节点的深度。那么，如果不是叶子选取了子代，则 $\Delta\phi = -1$ ；如果是叶子向上查找，则 $\Delta\phi = k_i$ 。
4. 结合 (2) (3) 得到 $a_i \leq 1 + \Delta\phi$ ，累加得 $\sum a_i \leq \Theta(n) + \phi(n) - \phi(0)$ 。因为从根节点出发、到根节点结束，所以 $\phi(n) = \phi(0) = h_T$ ，因而 $\sum a_i \leq \Theta(n)$ 。
5. 另一方面，因为每个节点都访问过一遍，所以 $\sum a_i \geq \Theta(n)$ ，故先序遍历的时间复杂度为 $\Theta(n)$ 。

空间复杂度的问题上，使用迭代器的版本的空间复杂度是 $O(1)$ ，看上去比递归或用栈的版本要小。但实际上，由于使用迭代器的版本需要用到亲代指针（此前两小节介绍的遍历方法都不需要使用亲代指针），而保存亲代指针本身需要 $\Theta(n)$ 的空间；从这个角度看，空间并没有被节约。

本节介绍了大量的二叉树遍历方法，其核心记忆点可以整理成下表（逆序遍历的性质请自行补全）。

表 8.1 二叉树的遍历算法

算法思想	层次遍历	先序遍历	中序遍历	后序遍历
半线性递归（递归）	-	NLR	LNR	LRN
半线性递归（迭代）	基于队列	基于栈	-	基于栈（反序）
线性递归	-	基于左链	基于左链	基于左链
迭代器	-	子代	最左侧通路	最长偏左侧通路

8.3 二叉树的逆线性化

8.3.1 实验：利用中序序列和另一序列重建二叉树

在《树》章中我们知道，利用树的先序序列和后序序列，或者层次序列和后序序列，都可以实现树的逆线性化，并且给出了适合在纸上演算的“表格法”。因为在长女-次妹表示法下，树的先序序列对应二叉树的先序序列，后序序列对应二叉树的中序序列，所以利用二叉树的先序序列和中序序列，也可以实现二叉树的逆线性化。请您在不依赖长女-次妹表示法的情况下，从二叉树的结构本身出发，利用递归方法证明二叉树可以被先序序列和中序序列重建。

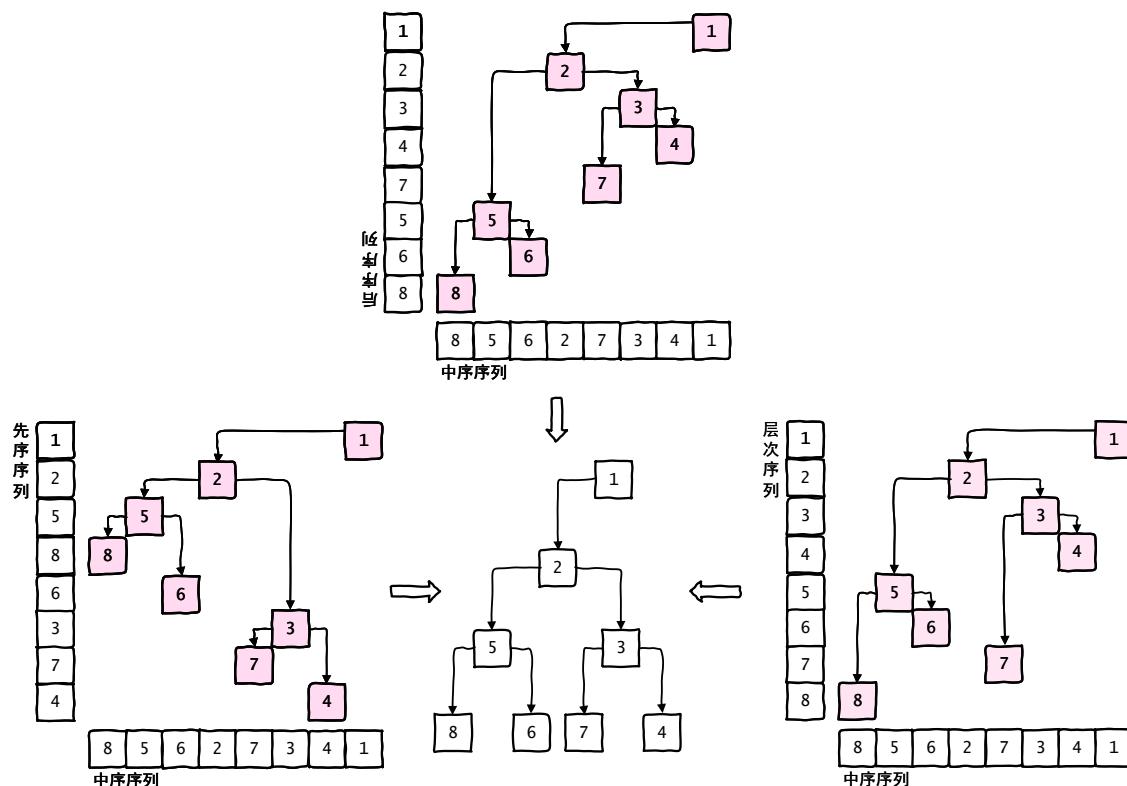


图 8.8 利用中序序列和另一序列重建二叉树

在二叉树的逆线性化时，同样可以使用表格法。其中，纵轴上从上到下是先序序列，横轴上从左到右是中序序列。类似地，纵轴上也可以是从上到下的层次序列，或者从下到上的后序序列，如图8.8所示。

在《树》章中已经证明了表格法的正确性，这里我们讨论如何实现这个算法。表格法的执行过程中，一个关键的步骤是利用横轴的位置和纵轴的位置，确定在表格里的节点位置，也就是对于每一个节点，我们需要去定位到它在横轴和纵轴上的位置。有两种对称的做法可以做到这一点：

1. 遍历先序序列上的元素，查找每个元素在中序序列上的位置。

2. 遍历中序序列上的元素，查找每个元素在先序序列上的位置。

通常来说，方法（1）更加自然，因为遍历先序序列上的元素的时候，可以从上而下地建立二叉树。反之，如果遍历中序序列上的元素，则需要从左到右地建立二叉树，技术上会更加困难，如果您感兴趣可以自行思考和实现。

为了查找元素在中序序列上的位置，一个直接的思路是直接遍历另一序列，线性查找和所查找元素相等的项。这个思路比较简单，您应当能够在理解表格法原理的情况下自己写出相应的代码。示例代码如下所示，可以在 *Rebuild.cpp* 中找到。

```

1 template <typename T>
2 class RebuildSeqLookup : public Rebuild<T> {
3     BinaryTreeNodeInst<T> rebuild(const Vector<T>& pre, const
4                                     Vector<T>& in, Rank preL, Rank preR, Rank inL, Rank inR)
5     {
6         if (preL == preR) {
7             return nullptr;
8         }
9         auto root { BinaryTreeNodeInst<T>::make(pre[preL]) };
10        auto inRoot { find(begin(in) + inL, begin(in) + inR, pre
11                            [preL] - begin(in)) };
12        auto leftSize { inRoot - inL };
13        if (auto left { rebuild(pre, in, preL + 1, preL + 1 +
14                            leftSize, inL, inRoot) }) {
15            left->parent() = root;
16            root->left() = move(left);
17        }
18        if (auto right { rebuild(pre, in, preL + 1 + leftSize,
19                               preR, inRoot + 1, inR) }) {
20            right->parent() = root;
21            root->right() = move(right);
22        }
23        return root;
24    }
25 public:
26     BinaryTree<T> operator()(const Vector<T>& pre, const Vector<
27                                 T>& in) override {
28         return BinaryTree<T>(rebuild(pre, in, 0, pre.size(), 0,
29                             in.size()));
30     }
31 };

```

上述算法每次调用非边界的递归函数，只会调用一次 `make` 生成新的节点。因此，上述算法会调用 n 次非边界的递归，而每次非边界的递归都会调用一次 `std::find` 的线性查找。因此，上述算法的最坏时间复杂度高达 $\Theta(n^2)$ ，当树的形态为左链的时候会取到这个最坏时间复杂度。

不过，在大多数情况下上述算法的时间复杂度并没有那么坏。根据对称性，在查找规模为 n 的子树的树根在中序序列上的位置时，平均需要查找的次数约为 $\frac{n}{2}$ 。

因此，重建整个树的平均时间为：

$$\bar{T}(n) = \frac{1}{2} \sum_{v \in T} n_v$$

其中， n_v 是以节点 v 为根的子树规模。在上述求和式中，每个节点 v 被计算了 $d_v + 1$ 次（在其所有祖先处各计算一次），其中 d_v 是节点 v 的深度。所以：

$$\bar{T}(n) = \frac{1}{2} \left(n + \sum_{v \in T} d_v \right)$$

因此，该算法的平均时间和树的内部路径长度 (internal path length) 相关。内部路径长度是每个节点到根节点的路径上的边数之和，也就是每个节点的深度之和。利用生成函数方法可以证明，随机二叉树的平均内部路径长度大约为 $n\sqrt{\pi n}$ ^[7]。这一结论的推导较为复杂，但可以关联随机二叉树的平均树高为 $2\sqrt{\pi n}$ ，以降低记忆难度。因此该算法的平均时间复杂度为 $\Theta(n^{3/2})$ 。

逆线性化作为线性化的逆过程，非线性的时间复杂度显然让我们不满意。我们希望能够迅速地找到元素在中序序列中的位置。另一个思路是，预先建立一个表 T ，从元素 e 映射到其在另一序列中的秩 $T[e]$ 。然后直接查表 T 。要实现这个思路，本书目前介绍的数据结构是不够的。目前介绍的数据结构，都是根据位置查找元素，而上述思路需要一个“反向”的查找数据结构，根据元素去查找位置。在 C++ 中，提供了用于建立映射的模板 `std::map` 或 `std::unordered_map`（将在《高级搜索树》和《散列》章中介绍它们的实现原理），它可以帮助我们实现这个做法。下面展示了一个使用 `std::unordered_map` 构建表的例子，您可以从实验中看到，在树的规模较大的情况下，该算法的性能显著高于前面线性查找位置的算法。

```

1 template <typename T>
2 class RebuildTable : public Rebuild<T> {
3     Vector<Rank> pre2in;
4     void buildTable(const Vector<T>& pre, const Vector<T>& in) {
5         unordered_map<T, Rank> table;
6         pre2in.resize(pre.size());
7         for (Rank r { 0 }; r < in.size(); ++r) {
8             table.emplace(in[r], r);
9         }
10        for (Rank r { 0 }; r < pre.size(); ++r) {
11            pre2in[r] = table[pre[r]];
12        }
13    }
14    BinaryTreeNodeInst<T> rebuild(const Vector<T>& pre, const
15        Vector<T>& in, Rank preL, Rank preR, Rank inL, Rank inR)
16        {
17            // ...
18            auto inRoot { pre2in[preL] };
19            // ...

```

```

18     }
19 public:
20     BinaryTree<T> operator() (const Vector<T>& pre, const Vector<
21         T>& in) override {
22         buildTable(pre, in);
23         return BinaryTree<T>(rebuild(pre, in, 0, pre.size(), 0,
24             in.size()));
25     }
26 }
```

在上面的算法中，每次查找位置都是 $O(1)$ 的，因此 `rebuild` 函数的时间复杂度为 $\Theta(n)$ 。而建表的时间复杂度则和具体的情形有关。如果二叉树中的元素恰好是 0 到 $n - 1$ 的数字，则上面代码中的 `std::unordered_map` 可以简单地使用一个向量替代，如下所示。

```

1 void buildTable(const Vector<T>& pre, const Vector<T>& in) {
2     Vector<Rank> table;
3     pre2in.resize(pre.size());
4     table.resize(in.size());
5     for (Rank r { 0 }; r < in.size(); ++r) {
6         table[in[r]] = r;
7     }
8     // ...
9 }
```

在这种情况下，建表的时间为 $\Theta(n)$ ，从而总体可以做到 $\Theta(n)$ 的时间复杂度。但如果沒有这么好的性质，二叉树的元素是其他类型，则 `std::unordered_map` 会使建表成为时间瓶颈，尤其当元素数量相对比较多的时候，具体原理参见《散列》章。而 `std::map` 则会使建表的时间复杂度变为 $\Theta(n \log n)$ ，具体原理参见《高级搜索树》章。

在考试中，建立反向映射表的做法通常不被认为是满分答案。即使给定了元素的范围，使得只需要上面那样的一个向量作为查找表，这种方案也需要额外的 $\Theta(n)$ 空间，存在被扣空间分的可能。另外，如果考试中出现此类题目，严格来说递归版本不一定能够取得满分，因为在树退化成链的情况下有可能会发生栈溢出错误。请您利用上一节的经验或利用《栈》章中介绍的一般方法，自行实现利用栈替代递归的版本。

如果不用反向映射表，那么就只能接受最坏 $\Theta(n^2)$ 的线性查找版本了吗？我们可以通过分析在线性查找中的不必要工作，来寻找这一版本的优化机会。我们知道，线性查找版本的高时间复杂度来自于 `std::find`，虽然我们通过将查找范围限定在 `inL` 到 `inR` 的区间之内，但仍然存在大量的重复。就像之前的分析的那样，在一个元素被加入到树中之前，它在其所有祖先的查找路径上都出现过一次。

考虑最左侧通路 $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k$ ，这 $k + 1$ 个节点就是先序序列上的

前 $k+1$ 个节点。设 v_j 在中序序列上的秩为 r_j 。因为 v_1 的秩为 r_1 ，所以在查找 v_0 在中序序列上的位置时，实际上可以不对 $[0:r_1]$ 这个区间进行查找。在上面的线性查找版本中，我们总是先查找 v_0 再查找 v_1 ，导致查找 v_0 的时候经过的区间，在查找 v_1 的时候又要经历一遍。反过来，如果先查找 v_1 再查找 v_0 ，就只需要从 r_1 开始查找就可以了，也就是说将 v_0 的查找区间限定在了 $(r_1:r_0]$ ，和 v_1 的查找区间加起来才是 $[0:r_0]$ ，也就是原先线性查找版本中单独查 v_0 需要查找的区间。换言之，通过先查 v_1 再查找 v_0 ，我们将第 0 层和第 1 层的查找压缩到了一层中。

那么，逆线性化的线性算法就已经呼之欲出了。反复利用上面的技巧，就可以把所有层的查找压缩到同一层中。中序序列的第一个元素就是最左侧通路的终点 v_k ，我们可以不做任何查找就建立最左侧通路。随后，我们利用图8.7所示的线性递归思路，依次重建最左侧通路右边挂载的各个子树。这个算法尽管思想很直观，但要实现它颇具挑战，下面展示了一个示例。

```

1  template <typename T>
2  class RebuildLinear : public Rebuild<T> {
3      Rank preL, inL;
4      BinaryTreeNodePos<T> buildLeftMost(const Vector<T>& pre,
5                                              const Vector<T>& in, BinaryTreeNodePos<T> start) {
6          auto pos { start };
7          while (pre[preL] != in[inL]) {
8              auto node { BinaryTreeNodeInst<T>::make(pre[++preL]) };
9              node->parent() = pos;
10             pos = pos->left() = move(node);
11         }
12         return pos;
13     }
14     BinaryTreeNodeInst<T> rebuild(const Vector<T>& pre, const
15                                     Vector<T>& in, optional<T> end) {
16         if (inL == in.size() || (end.has_value() && in[inL] ==
17                                  end)) {
18             return nullptr;
19         }
20         auto root { BinaryTreeNodeInst<T>::make(pre[preL]) };
21         auto pos { buildLeftMost(pre, in, root) };
22         ++preL;
23         while (pos) {
24             ++inL;
25             if (auto right { rebuild(pre, in, pos == root ? end :
26                                      pos->parent()->data()) }; right) {
27                 right->parent() = pos;
28                 pos->right() = move(right);
29             }
30             pos = pos->parent();
31         }
32         return root;
33     }
34 }
```

```

30 public:
31     BinaryTree<T> operator() (const Vector<T>& pre, const Vector<
32         T>& in) override {
33         preL = inL = 0;
34         return BinaryTree<T>(rebuild(pre, in, optional<T>{}));
35     }
36 };

```

从实验中可以看出，同样作为 $\Theta(n)$ 的算法，这一方法的性能甚至能高于直接用向量做查找表的做法，更不用说向量查找表还有很强的使用范围限制以及额外的 $\Theta(n)$ 空间。图8.9展示了本节所介绍的各种重建二叉树的算法平均时间（为 50 次随机序列的平均结果得到）。

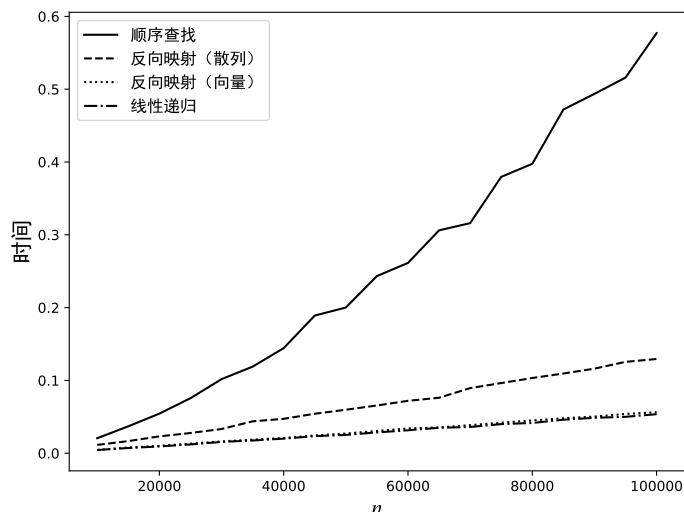


图 8.9 重建二叉树的平均时间消耗

在本节的最后，讨论一下中序序列和其他遍历序列重建二叉树的问题。对于后序序列来说，只要将二者反转，变为逆中序序列（RNL）和逆后序序列（NRL）重建二叉树的问题，就和“中序 + 先序”的情况一致了。如果使用表格法，只需要让后序序列在纵轴上从下到上列出即可，如图8.8所示。表格法同样可以解决“中序 + 层次”重建二叉树的问题。因为层次遍历基于队列，层次序列上相邻的元素在二叉树上可能差了很远，所以“中序 + 层次”的逆线性化无法使用上面介绍的线性算法。

8.3.2 利用后序序列和另一序列重建二叉树

在《树》章中讨论过，树的先序序列和后序序列可以实现树的逆线性化。经过长女-次妹表示法，树的先序序列对应二叉树的先序序列，而树的后序序列对应二叉树的中序序列，从而得到：二叉树的先序序列和中序序列可以实现二叉树的逆

线性化。但如果不经过长女-次妹表示法，仅从先序、后序遍历在树和二叉树两种场合的形式相似性进行考量，则情况又该如何呢？利用先序序列和后序序列，可以实现二叉树的逆线性化吗？

答案是否定的。反例很容易构造。考虑先序序列为 AB，后序序列为 BA 的二叉树，则该二叉树的根节点为 A，A 有一个子代 B，但我们无法确定 B 是 A 的左子代还是右子代，因为这两种情况都满足给定的先序序列和后序序列。然而二叉树是区分左右的，B 是左子代和 B 是右子代，这是两棵不同的二叉树。

利用反例得到否定的结论之后，如果从树的逆线性化角度出发，可以得到更深刻的结论。因为树的先序序列和后序序列可以实现树的逆线性化，所以使用相同的方法，把二叉树当成树处理，也一定可以用二叉树的先序序列和后序序列，重建出不区分左右的二叉“树”。也就是说，可以用先序序列和后序序列，利用树的重建算法进行重建，但其中每存在一个度为 1 的节点，都代表着 2 种不同的二叉树（其唯一子代可能在左侧或右侧）。如果重建出的树一共有 n_1 个度为 1 的节点，则一共可能有 2^{n_1} 种不同的二叉树。这里 n_1 并不需要实际进行重建的过程就可以算出，因为如果 v 是 u 的唯一子代，则必定有：

1. v 在先序序列上是 u 的直接后继。
2. v 在后序序列上是 u 的直接前驱。

反过来，如果 u 和 v 满足上面的两个条件，那么 v 也必然是 u 的唯一子代。请您自己证明这个结论，本书在 7.4.3 节中已经证明了树的版本。

而如果重建出的树没有度为 1 的节点，则通过先序序列和后序序列可以重建出唯一的二叉树。这种没有度为 1 的节点（即所有内部节点的度都为 2）的二叉树，称为**真二叉树** (proper binary tree)。真二叉树是二叉树和树的交集。我们也可以利用递归直接定义真二叉树。之前您已经看到过很多次递归定义，请您自己写出真二叉树递归定义中的两条规则。

和“先序 + 后序”的情况相似，层次序列加上后序序列，也可以和《树》章一样重建出树，当且仅当该树是真二叉树，重建得到的二叉树是唯一的。

8.3.3 利用先序序列和层次序列重建二叉树 *

利用先序序列和层次序列重建二叉树的过程，不能使用上一小节介绍的方法，迁移到树上去分析，因为树的先序序列和层次序列也没法做逆线性化。先序序列和层次序列所包含的信息，都是“自上而下”的次序，因而含有大量的重合。所以，利用先序序列和层次序列，我们很难有足够的信息来进行重建。比如说，假设先序序列和层次序列完全相同，可以重建出的二叉树有 T_n 种，那么：

1. 递归边界： $T_0 = T_1 = 1$ 。

2. 对于 $n \geq 2$ 的非平凡树，如果二叉树的树根度为 1，那么两个序列首元素之外的 $n - 1$ 个元素构成了它的唯一子树，可能性为 T_{n-1} 种。如果二叉树的树根度为 2，那么由于先序序列和层次序列完全相同，所以树根左子代必须是叶子（否则其子代在先序序列上会在其姊妹之前，而在层次序列上会在其姊妹之后）。因此，两个序列前两个元素之外的 $n - 2$ 个元素构成了它的右子树，可能性为 T_{n-2} 种。综上可得 $T_n = T_{n-1} + T_{n-2}$ 。
3. 这是经典的斐波那契 (Fibonacci) 数列的递推形式。因此，

$$T_n = F_{n+1} = \frac{\varphi^{n+1} - \psi^{n+1}}{\sqrt{5}}$$

其中， $\varphi = \frac{1+\sqrt{5}}{2}$, $\psi = \frac{1-\sqrt{5}}{2}$ ，该通项公式的推导应当是《线性代数》的习题。在此基础上，我们可以对一般的情况进行分析。

1. 先序序列 P 和层次序列 L 中，必然有 $P[0] = L[0]$ ，该元素是二叉树的根。对于非平凡树，设 $k = \max\{0 < k \leq n \mid P[0:k] = L[0:k]\}$ 。
2. 如果 $k = n$ ，则可能性为 T_n 种。
3. 如果 $k < n$ ，设 $L[k] = P[d](d \neq k)$ ，则 $P[k-1:d]$ 和 $P[d:n]$ 对应两棵姊妹子树，其中 $P[k-1]$ 和 $P[d]$ 分别是 $P[k-2]$ 的左右子代。因此，前 $k-1$ 项对应 T_{n-1} 种可能。对于这两棵姊妹子树重建时，二者的层次可以互相参考，从而确定这两棵子树的每一层分别有哪些元素，直到有一方的元素被用尽为止，然后又回到一般情况。
4. 在两棵姊妹子树相互参考的阶段，两棵子树各自的每一层结束时，层次序列会进入到另一棵姊妹子树的元素中，所以可以获知每一层次都有哪些元素。我们将每一层次的元素从右向左排列，就可以得到“反方向的层次序列”，也就是交换左右子树之后的层次序列。另一方面，先序序列可以导出逆先序序列，也就是交换左右子树之后的后序序列。于是可以由上一小节的结论得知，在此阶段可以进行树的逆线性化，并在真二叉树的情况下，可以重建出唯一的二叉树。

以上给出了已知先序序列和层次序列的情况下，计算满足条件的二叉树数量的一般方法。综合而言，我们对于四种遍历序列两两组合是否可以重建二叉树这个问题，可以得到下图。

从图8.10中我们可以发现，中序遍历和其他三种遍历之一可以重建二叉树，而其他三种遍历两两组合则不能重建二叉树。这是因为，中序序列中的树根位于左右子树之间。由于二叉树区分左右子树，所以中序序列实际上起到了“区分左右”的作用。而先序遍历和后序遍历中，左右子树都是连续处理的；在层次遍历中，左

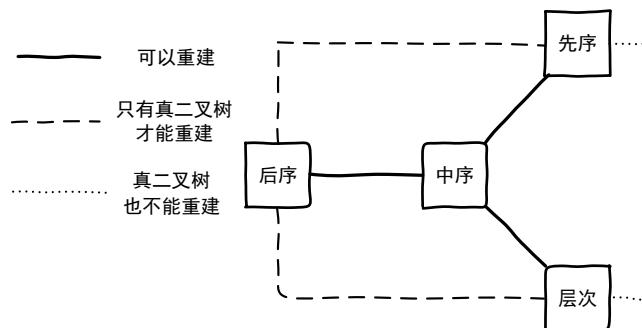


图 8.10 四种遍历序列两两组合是否可以重建二叉树

右子节点也是连续进入队列的；所以先序、后序和层次序列都起不到“区分左右”的作用，只能起到“区分上下”的作用。换言之，在“表格法”里，只有中序序列可以作为横轴，先序、后序和层次序列都只能作为纵轴。这就是在基于两个遍历序列做二叉树的逆线性化时，必须要有中序序列的原因。

另一方面，作为“区分上下”的遍历序列，后序序列是“自下而上”的，而先序序列和层次序列都是“自上而下”的。因此，后序序列和先序（层次）序列也能较为精准地重建出二叉树的形态，并在真二叉树的场合能够唯一确定二叉树；而给定先序和层次序列的时候，两个序列的信息过于重复，有效信息不足，重建结果最为宽泛，可能性最多。

您在阅读本节的内容之后，任给两个遍历序列，都应该能对二叉树进行重建，或计算出满足条件的二叉树的数量。

1. “先序 + 中序”、“后序 + 中序” 或 “层次 + 中序”。结果唯一，以直接用表格法快速求解。
2. “先序 + 后序” 或 “层次 + 后序”。结果通常不唯一，可以将二叉树当做一般的树处理，通过表格法重建出树，或是根据度为 1 的节点数量判断满足条件的二叉树数量。
3. “先序 + 层次”。结果通常不唯一。可以像上面一样，根据是否可以相互参考确定层次，分成多个阶段求解。对于可以确定层次的阶段，采用(2)的方法通过度为 1 的节点数来判断数量；对于不能确定层次的阶段，其数量服从斐波那契数列。最后将各个阶段的可能数量相乘。由于斐波那契数列的增长速度很快，在考试中如果给定的序列较长，将很难通过枚举方法求解，题目难度相对较大。

8.3.4 利用增强序列重建二叉树 *

从前面的讨论中可以发现，真二叉树是一类非常特殊的二叉树，在真二叉树的条件下，“先序 + 后序” 和 “层次 + 后序” 才能重建出唯一的二叉树。对于一般

的二叉树，可以通过“补充节点”的方式将其变形为真二叉树：

1. 对于度为 0 的节点，补充其左右子代。
2. 对于度为 1 的节点，如果它只有左子代，则补充其右子代；如果它只有右子代，则补充其左子代。
3. 特别地，对于空树，补充一个树根。

以下称补充的这些节点为**外部节点** (external node)。外部节点不包含数据，仅仅作为标记使用，在本小节将其记作 \wedge 。由于历史原因，外部节点这个名词在不同语境下可能会表示不同的含义，甚至一些教材中外部节点和叶子节点是同一概念，请注意辨析。在本书中，外部节点总是表示原先不存在于数据结构中的节点；对于不同的数据结构，补充外部节点的方法会有所不同。

您可以很容易证明，规模为 n 的二叉树进行补充之后，会得到规模为 $2n+1$ 的真二叉树，该真二叉树的内部节点都是原二叉树上的节点，而叶子节点都是外部节点。只要将这棵真二叉树的叶子节点删去，就可以得到原二叉树，因此该映射是一一对应的。

对这个真二叉树进行遍历得到的序列称为**增强序列** (enhanced sequence)，增强序列中包含 n 个数据和 $n+1$ 个用于指示位置的 \wedge 标记。对于度为 1 的节点，增强之后，左和右就被区分开了。因此，增强序列也可以起到区分左右的效果，和中序序列的作用相似，可以预见中序序列和增强序列之间有着大量的重复信息。事实上，对中序序列做增强无法引入任何新信息。我们有下面的定理：

定理 8.1： 增强的中序序列是将原中序序列依次插入 $n+1$ 个 \wedge 两两之间得到。

证明 在真二叉树上，每一个内部节点都必然有右子，所以每一个内部节点的直接后继，都是它右子的最左侧通路终点。由于真二叉树的性质，所以最左侧通路的终点既然没有左子，也就必然没有右子，从而它必然是真二叉树上的叶子节点，也就是一个外部节点。

因此，在增强的中序序列上，每一个内部节点的直接后继都是外部节点。同理可证直接前驱也是。结合内部节点的数量比外部节点少 1 个，可以得到增强的中序序列必定是原中序序列和 \wedge 交替的形式。 ■

于是我们可以知道，增强序列引入的信息和中序序列引入的信息是相似的。对于先序、后序或层次序列， \wedge 的位置，同时也指示了中序序列中的一个元素在被增强的序列中处在什么位置，从而计算出中序序列。因为中序序列和另一序列可以重建二叉树，所以我们可以期望增强一个非中序的序列，也可以重建二叉树。

下面证明增强的先序序列可以重建二叉树，对增强的后序序列，采用逆先序序列或逆后序序列作为跳板，即可化归到先序序列的情况，请您自己完成证明。

证明 对于增强的先序序列，其包含 n 个数据和 $n+1$ 个 \wedge 。注意到，序列中的最后一个元素必定是 \wedge ，其余元素包含 n 个数据和 n 个 \wedge ，将数据 d 替换为入栈 $v(d)$ ，则得到了一个长度为 $2n$ 的栈操作序列及其入栈序列。利用栈和二叉树的关系即可重建二叉树。 ■

通过该证明我们可以发现，对于先序遍历来说，增强序列引入的 $n+1$ 个 \wedge ，除最后一个没有实际含义之外，其余 \wedge 各自对应了一个出栈操作（这也是我们将其记作 \wedge 的原因）。因为出栈序列和二叉树的中序序列相对应，所以增强序列引入的这 n 个 \wedge 的位置能够指示中序序列。

对于增强的层次序列，由于其基于队列而非栈，我们不能用栈的方法进行分析。先序、中序、后序的遍历，都可以认为是基于二叉树的 RNL 结构进行的半线性递归，而层次遍历，则可以认为是基于层次进行的递归。因此，在分析层次序列相关的问题的时候，通常也是从层次的角度（也就是树的高度或深度）进行分析。

1. 因为二叉树去掉根节点（第 0 层）之后有可能变为 2 棵树，所以为了能够进行减治，需要将命题扩展为：对于 k 棵二叉树（ k 已知）组成的二叉树森林，已知增强的层次序列，可以进行逆线性化。
2. 递归边界：森林中的最大树高 $h \leq 0$ 的情况，每棵树都是平凡树，显然成立。
3. 对于 $h > 0$ 的情况。因为 k 已知，所以增强层次序列的前 k 个元素就是二叉树森林中每棵树的根。设这 k 个元素中，数据节点有 k_1 个，其余为不含数据的外部节点。那么，根节点（第 0 层）以下的部分，是 $2k_1$ 棵二叉树组成的森林，这些二叉树从左到右分别是 k_1 个数据节点的左右子树。于是，问题化归为高度 $h - 1$ 的情况。

上面的分析采用了头部减治的策略，将根节点（第 0 层）作为平凡项，后续层作为减治项处理。您也可以反其道而行之，将最低层作为平凡项，前面的层作为减治项，请您自己完成这种尾部减治策略下的证明。

第9章 优先队列

在《栈》和《队列》中讨论了两种访问受限的数据结构，分别服从 LIFO 和 FIFO。优先队列（priority queue）是另一种访问受限的数据结构。和 LIFO、FIFO 对应，它遵从的访问模式我称为“推入选出”（PIPO^[27]，Push In Pick Out）。在优先队列中，每个元素 e 有一个优先级 $p(e)$ ，传统上优先级如果用数表示，则越小的数字代表优先级越高。优先级的设置使我们可以在优先队列存储的数据类型上定义良序关系 \leq ：

$$x \leq y \Leftrightarrow p(x) \leq p(y)$$

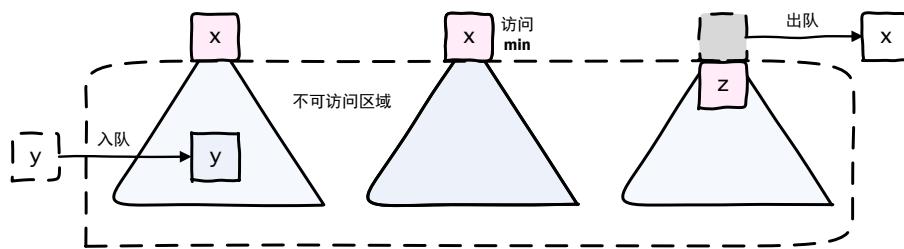


图 9.1 优先队列的入队、访问和出队

如图9.1所示，可以将优先队列认为是一个黑盒。当向优先队列中推入元素时，外部不需要知道推入到了什么位置。而当从优先队列中访问或弹出元素时，黑盒只会给外部返回当前优先队列中优先级最高的元素，也就是良序关系 \leq 下优先队列中的最小值。

尽管优先队列的名字中包含队列（queue），但优先队列和队列的关系并不大。在 C++ 的 STL 中，优先队列的接口名和栈一致，也采用推入（push）、弹出（pop）和顶部（top）的写法。您可以使用 *PriorityQueueTest.cpp* 测试自己的优先队列。

```

1 template <typename T>
2 class AbstractPriorityQueue : public DataStructure<T> {
3 protected:
4     std::function<bool(const T&, const T&)> cmp { std::less<T>() };
5 public:
6     virtual void push(const T& e) = 0;
7     virtual T pop() = 0;
8     virtual const T& top() const = 0;
9 }
```

9.1 线性优先队列

从本书的章节安排，您大约可以猜到优先队列的实现需要依托半线性结构。不过，在介绍半线性结构实现的优先队列之前，本节将首先讨论一下线性结构实现的优先队列是什么样子的。为简单起见，下面将默认基于向量实现优先队列，您可以自己思考列表的情况会有哪些变化。

9.1.1 实验：在无序序列上查找最小值 *

优先队列没有规定推入元素的位置，而向量在尾部插入元素是最快的，所以我们的一个自然的想法就是，每次推入元素的时候都在尾部插入。这样，当我们向优先队列中插入元素的时候是最快的，不需要任何的额外操作。而这样做的副作用则是，优先队列中的元素在向量中无序排列，每次寻找最小值的时候都需要遍历整个序列进行寻找。

```

1 template <typename T>
2 class NaiveVectorPriorityQueue : public AbstractPriorityQueue<T
3   > {
4   Vector<T> V;
5   Rank min() {
6     return min_element(begin(V), end(V), this->cmp) - begin(V
7       );
8   }
9   public:
10  void push(const T& e) override {
11    V.push_back(e);
12  }
13  T pop() override {
14    return V.remove(min());
15  }
16  const T& top() const override {
17    return V[min()];
18  }
19 }
```

可以看出，尽管插入元素的时间复杂度仅有分摊 $O(1)$ ，但是访问和弹出最小值的时间复杂度均高达 $\Theta(n)$ 。我们会很快地发现在访问和弹出中发生的重复工作：如果先访问再弹出，那么我们会执行两次 `min`，重复两次遍历整个序列的过程，找到位置相同的最小值。为了解决这个问题，只需要使用 4.5.2 节中介绍的记忆化搜索技术即可。当我们执行一次 `min` 之后，可以将找到的最小值储存起来，在这个最小值被弹出之前。请您自己实现这个思路，并和下面的示例代码进行比较。

```

1 template <typename T>
2 class MemoryVectorPriorityQueue : public AbstractPriorityQueue<
3   T> {
4   Vector<T> V;
```

```

4     optional<Rank> m;
5     Rank min() {
6         if (!m.has_value()) {
7             m = min_element(begin(V), end(V), this->cmp) - begin(
8                 V);
9         }
10    }
11 public:
12     void push(const T& e) override {
13         V.push_back(e);
14         m.reset();
15     }
16     T pop() override {
17         T e { V.remove(min()) };
18         m.reset();
19         return e;
20     }
21     const T& top() const override {
22         return V[min()];
23     }
24 };

```

我们可以进一步地分析上面这个策略中的重复工作。设当前优先队列的最小值 $m = e_1$, 新加入到优先队列里的元素为 e_2 。上述算法会在 e_2 加入到优先队列里时重置 m , 即使 $e_2 > e_1$, e_1 仍然是当前优先队列的最小值, 下次调用 `min` 仍然需要 $\Theta(n)$ 的时间去寻找它。这之间有大量的重复比较。事实上, 只要我们比较 e_1 和 e_2 , 就可以确定如何更新最小值 m : 只要令 $m = \min(e_1, e_2)$ 即可。这一策略使得在插入元素的时候, 只需要 $O(1)$ 的额外时间就可以更新最小值, 从而在连续插入、访问的时候, 最小值的记录始终不会失效, 可以做到 $O(1)$ 的分摊复杂度。但是, 一旦弹出了最小值, 就只能重新遍历向量去查找。下面展示了这一策略的一种实现方法。

```

1 template <typename T>
2 class UpdMemVectorPriorityQueue : public AbstractPriorityQueue<
3     T> {
4     // ...
5     void update() {
6         if (m.has_value() && this->cmp(V.back(), top())) {
7             m = V.size() - 1;
8         }
9     }
10 public:
11     void push(const T& e) override {
12         V.push_back(e);
13         update();
14     }

```

总结本小节介绍的三种优先队列实现。对于基于无序序列的优先队列, 插入

元素的时间复杂度总是分摊 $O(1)$ 。在朴素的方法中，访问和弹出均为 $\Theta(n)$ ；通过记录最小值，可以在连续访问的情形下，使访问复杂度变为分摊 $O(1)$ ；通过增量更新地记录最小值，可以在连续访问或插入的情形下，使访问复杂度变为分摊 $O(1)$ 。一旦涉及弹出，就会面临不可避免的 $\Theta(n)$ 的时间复杂度瓶颈。

表 9.1 基于无序序列的优先队列在不同场景下的时间消耗对比（单位：ms）

实验 ($n = 10^4$)	Naive	记录最小值	记录最小值 增量更新
仅连续插入	0	0	0
连续插入随后访问	4616	1	1
插入和访问交替	2362	2451	1

在 *VectorPriorityQueue.cpp* 中设计了多个实验来展示这三种优先队列实现的区别。实验 1 只讨论连续插入元素的情形；实验 2 在连续插入元素之后连续访问最小值；实验 3 在连续插入元素的过程间穿插访问最小值。从实验结果中可以看出，当操作次数为 n 时，实验 1 三种实现时间复杂度均为 $\Theta(n)$ ，所用时间相差无几；实验 2 中， $\Theta(n^2)$ 的朴素方法，所用时间远高于 $\Theta(n)$ 的记录最小值的方法；实验 3 中，不带增量更新的方法无法起到作用，时间复杂度也变为 $\Theta(n^2)$ ，只有增量更新地记录最小值仍然能保持在 $\Theta(n)$ 。

最后，如果是插入 n 个元素和删除 n 个元素的随机操作序列（不包含访问最小值），则基于无序队列实现的优先队列的平均分摊时间复杂度为 $\Theta(\sqrt{n})$ 。这里平均是针对随机操作的平均，其分母是随机操作序列的数量即 $\text{Catalan}(n)$ ；分摊是针对按序列连续操作的分摊，其分母是随机操作序列的长度即 $2n$ 。这个结果基于前两章中介绍的树和栈的关系及平均树高的结论。

9.1.2 实验：用有序序列实现优先队列 *

尽管通过设计、维护一个储存最小值位置的变量，我们降低了很多重复工作，但即使增量更新地记录最小值，也仅仅能在连续插入和访问组成的操作序列中做到 $O(1)$ 的分摊复杂度。在操作序列中一旦混入了弹出，大量的重复工作就扑面而来。比如，元素 e_1 的最小值地位被 e_2 取代后， e_2 旋即又被弹出优先队列，这时就又需要 $\Theta(n)$ 的时间重新寻找 e_1 ，并不能从“记忆”中恢复上一个最小值的位置。

在整个优先队列中，之前存在过最多 $\Theta(n)$ 个最小值；而上一小节的记录最小值方案，只能存储最新的一个最小值。这就是重复工作的根源。加强“记忆”的能力、存储历史上的最小值，这一技术在 4.6.2 节中介绍的最小栈里出现过。最小栈通过引入了一个辅助栈，为栈增加了一个输出最小元素的功能。但是，如果我们希

望用相同的方法建立优先队列，会发现由于优先队列的出队次序就是元素的大小次序，所以这个辅助“栈”实质上就是优先队列中的元素组成的有序序列。既然如此，那么也就不需要辅助的数据结构，直接让优先队列中的元素保持有序就可以了。这就引申出了另一种基于线性结构实现优先队列的方案。

1. 任何时候，都保持优先队列中的元素有序。
2. 当访问或删除最小值的时候，因为优先队列中的元素已经有序，所以可以直接输出。注意由于向量尾部操作的效率远高于在头部操作，所以最小值的位置应当被放在向量尾部。也就是说，优先队列中的元素应当是逆序的。
3. 另一方面，当插入一个元素的时候，首先在已有的有序序列中查找该元素应当被插入的位置，然后在该位置处插入。这里可以采用顺序查找，也可以利用原有序列有序的性质进行折半查找。由于插入本身需要平均 $\Theta(n)$ 的时间，所以折半查找并不能降低时间复杂度。

以下是一个采用折半查找维护有序序列实现优先队列的例子。

```

1 template <typename T>
2 class SortedVectorPriorityQueue : public AbstractPriorityQueue<
3   T> {
4   // ...
5   void push(const T& e) override {
6     auto pos = lower_bound(begin(V), end(V), e, this->icmp)
7     - begin(V) };
8     V.insert(pos, e);
9   }
10  T pop() override {
11    return V.pop_back();
12  }
13  const T& top() const override {
14    return V.back();
15  };

```

您可以发现，为了维护序列的有序性，这一方案在插入元素的时候需要高达平均 $\Theta(n)$ 的时间复杂度；另一方面，得益于序列的有序性，在访问或删除元素的时候，时间复杂度是 $O(1)$ 的。和上一小节介绍的无序序列方案相比，该方案相当于将访问或删除时需要付出的时间成本转移到了插入上。

我们将基于有序序列的优先队列加入到上一小节的实验中。可以发现，尽管有序序列插入元素的时候比较慢，但其在访问最小值时高效，使其在实验 2、3 中都取得了不错的成绩。在新设计的实验 4 中，我们首先进行 n 次插入，然后进行 n 次弹出。上一小节的三种在实验 4 中的结果几乎没有区别，而得益于有序序列弹出的 $O(1)$ 性能，其在实验 4 中的耗时显著小于上一小节的三种优先队列。

表 9.2 基于有序序列和无序序列的优先队列在不同场景下的时间消耗对比（单位：ms）

实验 ($n = 10^4$)	无序序列 记录最小值 增量更新		有序序列
	0	9	
仅连续插入	0	9	
连续插入随后访问	1	9	
插入和访问交替	1	10	
连续插入随后删除	2432	9	

9.2 选择排序

9.2.1 简单选择排序

优先队列和排序都建立在定义了良序关系的元素数据类型上。我们可以用下面的模板，借助优先队列实现排序：

1. 建立一个空优先队列 P 。
2. 将待排序的 n 个元素依次插入到 P 中。
3. 从 P 中依次弹出 n 个元素。

这个过程正是上一小节设计的实验 4。由于优先队列的 PIPO 性质，它会首先弹出 n 个元素中的最小值。弹出最小值之后，剩余 $n - 1$ 个元素中的最小值，就是全部 n 个元素中的次小值。因此，优先队列会依次弹出最小值，次小值，第三小值，以此类推，直到最后一个被弹出的最大值为止。于是，我们借助优先队列完成了排序。本章将介绍多种衍生自该模板的排序方式。

将本节中介绍的基于无序序列的优先队列应用于上述模板中，得到的排序算法称为**选择排序** (Selection Sort)。由于在这个模板中只涉及连续的插入和删除，没有访问，所以只需要最朴素的版本，不用引入对最小值位置的维护。选择排序的原理如图9.2所示，它可以看成是一个减治算法。

1. 对于 $n \leq 1$ ，不需要进行排序。
2. 对于 $n > 1$ ，遍历序列找到其中的最小值 m ，将 m 和 $A[0]$ 交换，然后递归地对 $A[1:n]$ 进行排序。

```

1 // Selection Sort
2 void sort(Linear<T>& V) override {
3     for (Rank i { 0 }; i < V.size(); ++i) {
4         auto j { std::min_element(begin(V) + i, end(V), this-
5             cmp) - begin(V) };
6         std::swap(V[i], V[j]);
7     }
}

```

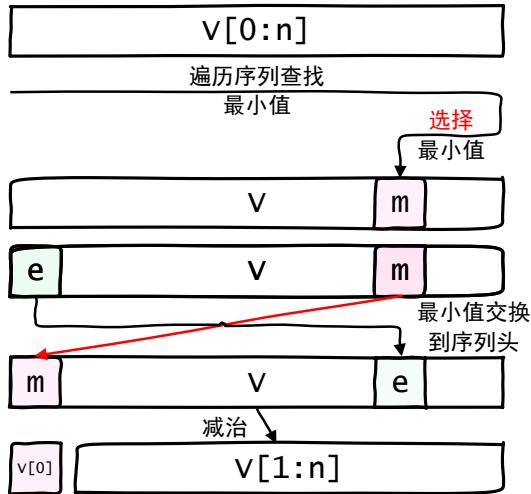


图 9.2 选择排序

选择排序的时间复杂度是 $\Theta(n^2)$ ，空间复杂度是 $O(1)$ 。它是最简单直观的排序方法之一，在数据规模 n 很小的时候性能较高。本书此前已经讨论了归并排序，未来还会介绍很多排序算法。下面以选择排序为例，介绍讨论排序算法时的一些主要的关注点。

1. 时间复杂度。更进一步地，我们通常会从比较次数和移动次数两个方面去研究时间复杂度。在没有优化的情况下，归并排序的比较次数和移动次数均为 $\Theta(n \log n)$ 。选择排序的比较次数为 $\Theta(n^2)$ ，而移动次数为平均 $\Theta(n)$ 。从比较次数和移动次数两个方面对时间效率进行刻画，可以更加精确地反映不同场景下排序算法的性能。比如，比较只需要“读”，而移动需要“写”，在一些存储介质（比如 E²PROM^[1]）上“写”的效率远远低于“读”，此时选择排序移动次数低的优势就能够得到体现。

进一步地，为了定量分析一个排序算法的时间性能，还会讨论时间复杂度前面的常数。对于选择排序而言，其比较次数是固定的，为：

$$\sum_{k=0}^{n-1} k = \frac{n(n-1)}{2} \sim \frac{1}{2}n^2 = \Theta(n^2)$$

其时间复杂度的常数为 $\frac{1}{2}$ 。

2. 空间复杂度。排序算法的空间复杂度只计入排序过程中需要使用的额外辅助空间，而不会计入被排序序列本身所占的空间。通常排序算法的空间复杂度分成三种：常数辅助空间（或称就地，in-place） $O(1)$ 、线性辅助空间 $\Theta(n)$ 、和数据范围相关的辅助空间 $\Theta(R)$ （其中 R 为被排序元素的数据范围，如对月、日组成的生日日期进行排序时， $R = 366$ ）。此前介绍的归并排序的空间复杂度为 $\Theta(n)$ ，而选择排序的空间复杂度为 $O(1)$ 。

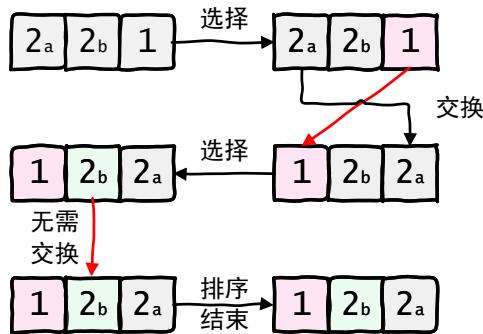


图 9.3 选择排序的不稳定性

3. **稳定性 (stability)**。待排序序列中，有可能出现相等但不相同的元素。假设 a_1 和 a_2 是两个相等但不相同的元素，且在原序列中， a_1 位于 a_2 之前。如果排序算法能够保证在排序之后得到的新序列中， a_1 仍然位于 a_2 之前，则称该排序算法为稳定的。归并排序在待归并元素相等的时候优先选择前半段的元素，因此归并排序是稳定的。而选择排序则是不稳定的，如图9.3所示，在选择排序的过程中元素的相对位置关系可能会变化，原先在 2_b 前面的元素 2_a 因为和 1 交换，移动到了 2_b 后面。

9.2.2 双选择排序 *

选择排序有一个直接的改进：**双选择排序（Double Selection Sort）**。在简单的选择排序中，我们每次遍历待排序部分，只选出其中的最小值；双选择排序则不但选出其中的最小值，还选出其中的最大值。这样，我们就只需要大约 $\frac{n}{2}$ 趟遍历就可以完成整个排序过程。双选择排序的思想看起来简单，但实现起来却并不显然，建议您自己实现，并和下面的示例程序进行对比。

```

1 // Double Selection Sort
2 void sort(Linear<T>& V) override {
3     Rank lo { 0 }, hi { V.size() - 1 };
4     while (lo < hi) {
5         Rank j { lo }, k { lo };
6         for (Rank l { lo + 1 }; l <= hi; ++l) {
7             if (this->cmp(V[l], V[j])) {
8                 j = l;
9             }
10            if (this->cmp(V[k], V[l])) {
11                k = l;
12            }
13        }
14        if (k == lo) {
15            k = j;
16        }
17        swap(V[lo++], V[j]);
18        swap(V[hi--], V[k]);
}

```

```
19 }
20 }
```

双选择排序并不能降低比较次数的常数，它只是将遍历趟数降低到原来的 $\frac{1}{2}$ ，从而提高了算法的时间局部性，能够更好地利用 Cache。回忆6.1.3节中对时间局部性的介绍可以知道，提高时间局部性还有助于提高算法并行度。如果处理器支持无限多个线程并行计算，那么由于选最小和选最大这两个过程可以并行进行，双选择排序的性能可以达到简单选择排序的大约 2 倍。

9.2.3 宾果排序 *

除了同时选出最小值和最大值之外，还可以同时选出所有的最小值，这种改进的选择排序称为**宾果排序**（Bingo Sort）。在宾果排序中，每次找到最小值之后，会将所有等于该最小值的元素依次交换到序列开头。当被排序的数据重复率较高的时候，宾果排序能够在 $\Theta(n)$ 的时间里排好所有等于最小值的元素，而不需要一轮一轮找到重复的最小值，从而有效减少重复工作。如果待排序序列共有 R 个不等的值，则宾果排序的时间复杂度降低为 $\Theta(nR)$ 。但需要注意的是，宾果排序每一轮需要先找最小值、再判断每个元素是否等于这个最小值。于是，每个元素要参与比较 2 次（简单选择排序只需要比较 1 次），因此在元素重复率低的时候反而会起到负面影响。

```
1 // Bingo Sort
2 void sort(Linear<T>& V) override {
3     for (Rank i { 0 }; i < n; ) {
4         auto min { *min_element(begin(V) + i, end(V), this->cmp)
5                 };
6         for (Rank j { i }; j < n; ++j) {
7             if (V[j] == min) {
8                 swap(V[i++], V[j]);
9             }
10        }
11    }
}
```

9.2.4 实验：稳定版本的选择排序 *

另一个改进方向是从稳定性入手。选择排序是不稳定的，因为我们直接将最小值交换到了序列的最左侧。和最小值进行交换的元素，在这一交换的过程中，与其他元素的相对位置关系被打破了。为了让其他元素保持原有的相对位置关系，这里不能直接将最小值交换到序列的最左侧，而是应当执行一次循环右移，如图所示。这种做法看似提高了稳定性，但同时也让选择排序丧失了它最大的优势（移动

次数少)。如果刻意要追求稳定性, 不如使用下一小节介绍的插入排序。因此, 通常不会采用这种“改进”, 一般还是认为选择排序是不稳定的。一种稳定的选择排序实现如下所示, 您也可以用`std::rotate`来进行循环位移。

```

1 // Selection Sort (Stable Version)
2 void sort(Linear<T>& V) override {
3     for (auto i { begin(V) }; i != end(V); ++i) {
4         if (auto j { min_element(i, end(V), this->cmp) }; j != i)
5             {
6                 auto tmp { move(*j) };
7                 move_backward(i, j, j + 1);
8                 *i = move(tmp);
9             }
10    }

```

表 9.3 不同版本选择排序的时间消耗对比 (单位: ms)

n	简单选择排序	双选择排序	宾果排序 互不相同	宾果排序 只取 0-9	稳定版选择排序
2000	81	66	108	0	119
4000	370	261	418	1	441
6000	772	565	937	1	997
8000	1364	1020	1675	2	1792
10000	2140	1608	2605	3	2804

在 *SelectionSort.cpp* 中进行了两个方面的实验。实验 1 对 n 个不同元素进行排序, 可以看出, 双选择排序的性能明显高于简单选择排序, 而宾果排序性能略低, 稳定版本的性能最低。实验 2 对 n 个元素进行排序, 但所有元素被限定在了 $0, 1, \dots, 9$ 之间。在这种情况下, 宾果排序的时间复杂度为 $\Theta(nR) = \Theta(10n)$, 因而性能远远高于其他版本。

9.3 插入排序

9.3.1 实验: 直接插入排序和折半插入排序

按照上一节的思路, 考虑有序序列实现的优先队列, 可以得到另一种排序算法: 插入排序 (Insertion Sort)。插入排序在数据规模 n 较小的时候是时间效率最高的排序算法之一, 它同样可以认为是一个减治算法, 其原理如图9.4所示。

1. 对于 $n \leq 1$, 不需要进行排序。
2. 对于 $n > 1$, 先递归地对 $A[0:n - 1]$ 进行排序, 然后在已排序的 $A[0:n - 1]$

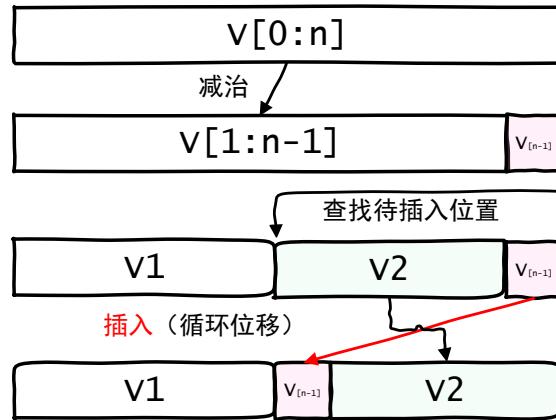


图 9.4 插入排序

中查找 $A[n - 1]$ 的位置，将其插入在对应位置上。在这一步默认采用顺序查找（称为直接插入排序）；如果采用折半查找，则称为**折半插入排序**（Binary Insertion Sort）。

选择排序和插入排序，可以认为是借助优先队列对向量进行排序的方法，分别基于无序序列和有序队列的优先队列；也可以认为是减治思想在排序中的应用（之前介绍的归并排序则是分治），分别基于头部减治和尾部减治。

我们对比直接插入排序和折半插入排序。在单次查找的过程中，直接插入排序的性能最好为 $O(1)$ ，最坏为 $\Theta(n)$ ，平均也为 $\Theta(n)$ ；另一方面，折半插入排序的时间复杂度总是 $\Theta(\log n)$ 。看起来二折半插入排序的时间性能总是高于直接插入排序，但事实上并不一定是这样。

比如说，折半插入排序必须要通过比较找到要插入的位置，再通过移动进行插入，如下所示。

```

1 // Binary Insertion Sort
2 void sort(Linear<T>& V) override {
3     for (Rank i { 1 }; i < V.size(); ++i) {
4         if (this->cmp(V[i], V[i - 1])) {
5             auto pos { upper_bound(begin(V), begin(V) + i, V[i],
6                                   this->cmp) };
7             auto tmp { move(V[i]) };
8             move_backward(pos, begin(V) + i, begin(V) + i + 1);
9             *pos = move(tmp);
10        }
11    }

```

但是对于直接插入排序而言不需要这样，比较和移动可以同时进行。

```

1 // Insertion Sort
2 void sort(Linear<T>& V) override {
3     for (Rank i { 1 }; i < V.size(); ++i) {
4         if (Rank j { i }; this->cmp(V[i], V[i - 1])) {

```

```

5     T tmp { std::move(V[i]) } ;
6     do {
7         V[j] = std::move(V[j - 1]);
8     } while (--j > 0 && this->cmp(tmp, V[j - 1]));
9     V[j] = std::move(tmp);
10    }
11 }
12 }
```

从双选择排序的例子中可以看出，这种做法具有更好的时间局部性。在直接插入排序里，比较和移动一同进行，由于移动本身需要读取并写入元素，所以比较不会引入额外的访存。

这个观点同时也可以引入直接插入排序和选择排序的比较中。直接插入排序的平均比较次数为

$$\sum_{i=1}^n \left(1 + \frac{1}{i+1} \sum_{j=0}^i j \right) = \frac{n(n+5)}{4} \sim \frac{1}{4}n^2 = \Theta(n^2)$$

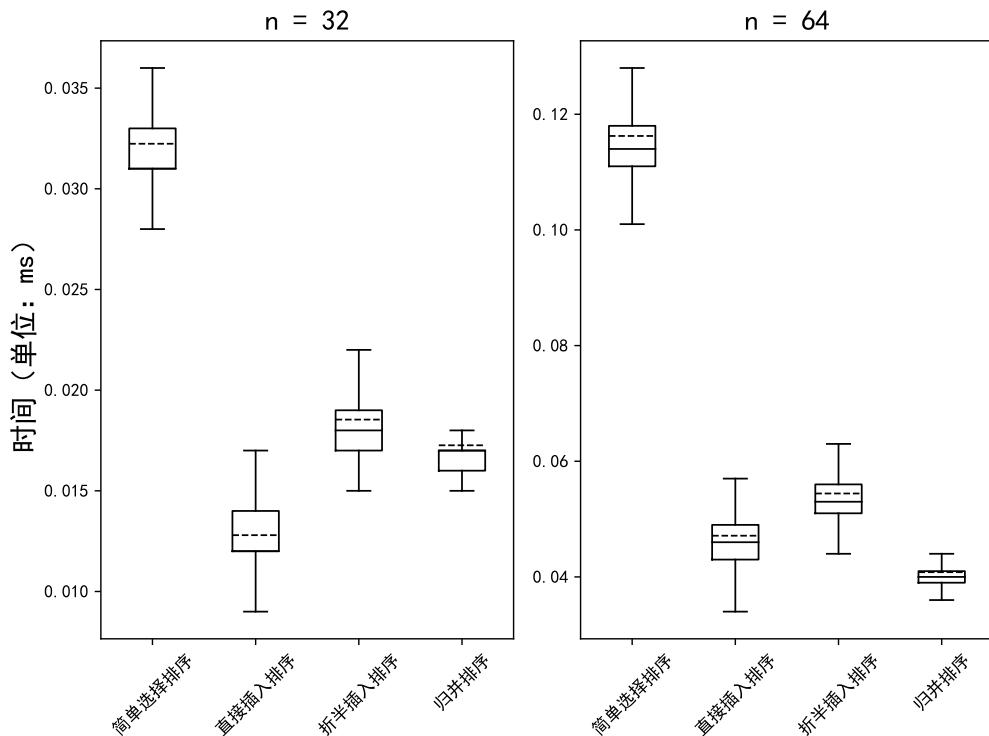


图 9.5 选择排序、插入排序和归并排序小规模排序时间消耗对比

可以看出，直接插入排序的平均比较次数仅为选择排序的一半。与此同时，直接插入排序的移动和比较是同时进行的，故移动次数也为 $\Theta(n^2)$ ，远远高于简单选择排序的 $\Theta(n)$ 。但在 *InsertionSort.cpp* 的实验中我们看到，直接插入排序的用时显著低于简单选择排序，比也双选择排序低不少，这就是移动和比较同时进行，不

引入额外访存成本带来的好处。回到直接插入排序和折半插入排序的比较上，如图9.5所示，当 n 比较小的时候，折半插入排序虽然减少了比较的时间，但其在比较时引入的额外 $\Theta(\log n)$ 次访存的时间反而更高，从而性能反而不如直接插入排序。当然，当 n 比较大的时候，折半插入排序相比于直接插入还是有明显的优势，只是这个时候我们又会倾向于采用 $\Theta(n \log n)$ 的排序方法了。

表 9.4 选择排序、插入排序和归并排序中等规模排序时间消耗对比（单位：ms）

n	简单选择排序	直接插入排序	折半插入排序	归并排序 自上而下
1000	23	10	7	1
3000	217	90	67	3
10000	2263	1019	670	11

除了性能优势之外，插入排序和选择排序同属空间复杂度 $O(1)$ 的就地排序。同时，插入排序和归并排序一样是稳定的，胜过不稳定的选择排序。因此，在规模较小的情况下，通常都采用插入排序，而不使用选择排序。

9.3.2 实验：列表版本的选择排序和插入排序 *

本小节讨论迁移到链式结构上的情况。通常我们讨论排序算法都基于向量（或数组），当算法迁移到链式结构时，其性质可能会发生变化。

在列表上，折半查找变得不再可用，所以我们只讨论直接插入排序和选择排序。对于这两种排序而言，比较的时候都进行的是顺序访问，可以被列表所支持。因此在比较这一部分，列表和向量并无二致。但在移动这一部分则有所不同。在《列表》章介绍过，列表是更加灵活的数据结构，将列表上的一个节点从一个位置移动到另一个位置（已知源位置和目的位置），只需要 $O(1)$ 的时间。借助这个性质设计了列表版本基于指针的归并排序，它同样可以改变列表版本的选择排序、直接插入排序的性质。

对于选择排序而言，移动的次数是很少的，只有 $\Theta(n)$ ，分摊到每一趟就是 $O(1)$ ，即进行 1 次交换。因此， $O(1)$ 的节点移动对选择排序而言没有性能上的收益。但是，通过移动节点代替交换，可以让选择排序具有稳定性。回忆在前面的小节里介绍的稳定版本的选择排序，当时我们因为它循环位移的过程太慢，认为这是没有意义的优化。而在列表的场合，只需要移动 1 个节点的位置就可以实现上述循环位移的过程，等于是在几乎不影响性能的情况下，为选择排序赋予了稳定性。

不过，选择排序在列表上这一点点小优势，和直接插入排序相比可谓九牛一毛。原本直接插入排序和选择排序相比就有 2 倍的比较次数优势，只是在移动次

数上吃亏，但在列表上，“插入”这个过程只需要移动1次节点，变成了单次 $O(1)$ 、总共 $\Theta(n)$ 的操作，和选择排序的移动一样了。于是，直接插入排序在列表上实现的性能甚至和向量相比有所提升。

表 9.5 向量和列表版本的选择排序和插入排序时间消耗对比（单位：ms）

n	简单选择排序 向量	直接插入排序 向量	折半插入排序 向量	简单选择排序 列表	直接插入排序 列表
1000	21	9	6	24	6
3000	203	89	60	238	62
10000	2134	980	625	2605	704

在 *ListSelectInsertSort.cpp* 中进行了实验，实验里采用的列表是节点随机置乱后的。因为 Cache 利用率上的差异，选择排序在列表上的性能会显著下降，但仍然高于之前所述向量上“稳定版本”的选择排序。而即使在随机置乱之后，列表版本的直接插入排序仍然效率明显高于向量版本，甚至十分接近折半插入排序的性能。

9.3.3 实验：插入排序的自适应性 *

回忆2.6.8节中介绍的内容。对于归并排序而言，如果向量已经基本有序，只有一小部分是乱序的，那么可以通过对归并排序进行小幅度修改，使其达到更高的时间效率。而对于插入排序而言，即使不进行任何修改，其也可以利用到向量基本有序的条件，达到更高的时间效率，这一性质称为**自适应性** (adaptability)。

根据上文中的分析，选择排序及其任何变种的比较次数是一定的，即使向量已经基本有序甚至完全有序，选择排序也需要付出大约 $\frac{1}{2}n^2$ 次的比较，所以选择排序是不具备自适应性的。这又是插入排序相对选择排序的一个优势。

在2.6.8节中讨论过，可以使用信息熵和逆序对两种方法对向量的基本有序性进行刻画。信息熵是 $\Theta(n \log n)$ ，而逆序对的数量是 $O(n^2)$ ，对于插入排序这种最坏 $\Theta(n^2)$ 的算法来说，显然用逆序对进行分析会比较合适。

设逆序对数量为 r 。对于直接插入排序的第 k 趟，此时 $V[0:k]$ 已经有序，将 $V[k]$ 插入到 $V[d_k]$ 的位置上。这个过程中，只有 $V[d_k:k]$ 和 $V[k]$ 的相对位置发生变化，通过 $k - d_k + 1$ 次比较和几乎同等数量的移动，消除了 $(j, k), j = d_k, d_k + 1, \dots, k - 1$ 这 $k - d_k$ 个逆序对，并没有增加任何逆序对。在整个直接插入排序中，一共 n 趟排序，将逆序对数量从 r 降低为 0，因此，直接插入排序的时间复杂度为 $\Theta(n + r)$ 。类似地，可以得到折半插入排序的时间复杂度为 $\Theta(n \log n + r)$ 。当 r 很小时，直接插入排序的性能会反而高于折半插入排序；而当 r 较大时，如 $r = \Theta(n^2)$ 的情况下，那么折半插入排序的性能仍然更高。

表 9.6 选择排序、插入排序和归并排序在整体有序下时间消耗对比（单位：ms）

乱序长度	n	简单选择排序	直接插入排序	折半插入排序	归并排序 自上而下
$\frac{1}{10}n$	1000	24	0	0	0
$\frac{1}{10}n$	3000	211	1	0	2
$\frac{1}{10}n$	10000	2380	10	8	8
$\log n$	1000	24	0	0	0
$\log n$	3000	215	0	0	2
$\log n$	10000	2305	0	0	8

您可以从 *InsertionSort.cpp* 的实验 2 和实验 3 中看到这一点。实验 2 中的 n 个元素只有前 $\frac{1}{10}n$ 个是乱序的，而实验 3 中的元素只有前 $\log n$ 个是乱序的。选择排序在实验 2 和实验 3 中的性能，并未呈现出任何提高，因为它不具有自适应性。插入排序的性能有显著的提高。在实验 2 中，由于 $r = \Theta(n^2)$ ，所以在 n 较大时折半插入排序的性能会高于直接插入排序，同时尽管常数相当小，但在 n 足够大时其性能仍然会低于 $\Theta(n \log n)$ 的归并排序。而在实验 3 中，由于 $r = \Theta(\log^2 n) = o(n)$ ，所以直接插入排序的性能相比折半插入排序会更高，并且其性能会高于归并排序。

从上面的分析和实验中可以看出，向量任意位置的“基本有序”性质都有助于提高直接插入排序的时间效率，因而其具有良好的自适应性。与之相比，在 2.6.8 节中改进的归并排序，只有在比较严苛的先验条件下，改进才能产生效果，自适应性较差。下面用一个例子说明这一点。考虑 3、1、4、2 这个序列，对于改进的归并排序而言，无论是第 1 趟还是第 2 趟，都无法根据“前半部分全部小于后半部分”来快速归并，从而无法产生改进效果；而直接插入排序可以利用到跨归并段的 (3, 4)、(1, 4) 和 (1, 2) 这三个非逆序对，降低自己的时间消耗。

9.4 锦标赛树

9.4.1 胜者树的结构 *

在上一节中我们看到，无论采用无序线性表还是有序线性表来实现优先队列，都会面临“偏重一侧”的问题。无序线性表的插入只需要 $O(1)$ 的时间，但需要 $\Theta(n)$ 的时间用于访问和弹出；另一方面，有序线性表的访问和弹出只需要 $O(1)$ 的时间，但需要 $\Theta(n)$ 的时间用于插入。这种严重的不均衡是我们不愿意看到的。一个稳定工作的数据结构，插入和弹出的数量应该是比较接近的，所以无论采用上一节的哪一种实现，都必须承担操作序列下 $\Theta(n)$ 的分摊复杂度。

我们希望在这两个方案之间寻求平衡。无序和有序（全序）之间的平衡点，就是偏序关系。在《树》章中介绍过，半线性结构将元素位置的全序关系弱化为了偏序关系，从而取得了比线性结构更好的灵活性。因此我们可以想到，采用半线性结构来存储优先队列是一个可行的方案。

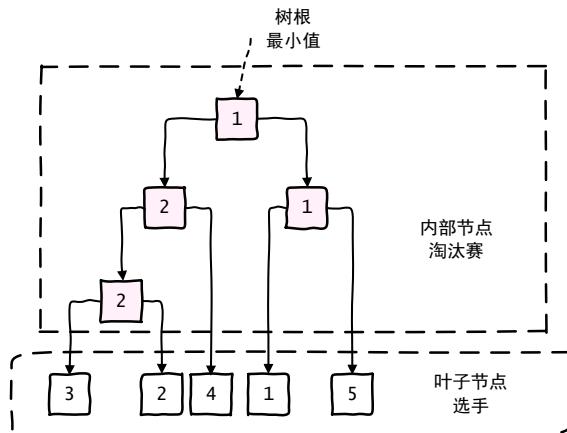


图 9.6 锦标赛树（胜者树）

锦标赛树 (tournament tree) 是一种经典的半线性结构的优先队列，它的存储方式类似于现实生活中的锦标赛。如图9.6所示， l 和 r 是节点 v 的左子代和右子代，根据 l 和 r 上的元素大小 “决出胜败”，来决定 v 上的元素。因为优先队列的目标是取最小值，所以较小的元素应当 “取胜”，也就是说 $v = \min(l, r)$ 。因此，锦标赛树也被称为**胜者树** (winner tree)。

锦标赛最终只会有 1 个冠军，其他选手都会被淘汰，并且 1 场淘汰赛会淘汰 1 个选手。所以在胜者树中，一共有 n 个叶子节点（对应优先队列里的 n 个元素），以及 $n - 1$ 个内部节点（对应 $n - 1$ 场淘汰赛）。锦标赛的“轮次”就是锦标赛树的高度 h 。

1. 当插入元素的时候，新增 1 名“选手”，并让它执行锦标赛直到被淘汰为止。这个时间复杂度是最坏 $\Theta(h)$ 的，并且我们可以预期新选手有更大的可能性“一轮游”“二轮游”，只有很少的机会能够“挺进决赛”，所以平均情况下的时间复杂度可能会更低。
2. 当访问元素的时候，直接访问树根（冠军）即是最小值。
3. 当删除元素的时候，弹出树根（冠军），然后让冠军的“晋级之路”上的比赛重赛。这个时间复杂度是 $\Theta(h)$ 的。

通过上面的分析我们看到，胜者树的插入、删除最坏时间复杂度均为 $\Theta(h)$ ，而访问最小值的时间复杂度为 $O(1)$ 。如果我们能够让胜者树呈现“矮胖”的形状，高度尽可能小一些，那么插入和删除的时间复杂度就能够得到良好的控制。

9.4.2 完全二叉树

我们知道，二叉树中一个节点最多2个子代。因为第0层的根节点只有1个，所以可以递推得，第 k 层的节点至多 2^k 个，第 $0, 1, \dots, k$ 层的节点总和至多 $2^{k+1} - 1$ 个。给定二叉树的规模 n ，如果二叉树的树高 h 满足

$$h = \min\{k \mid 2^{k+1} - 1 \geq n\}$$

则二叉树的树高最低，二叉树达成最为“矮胖”的形状。显然，在给定规模 n 的情况下，最为“矮胖”的二叉树形态通常不止一个。如图9.7所示，当 $n = 8$ 时，图中的三棵二叉树都是高度最低的二叉树。

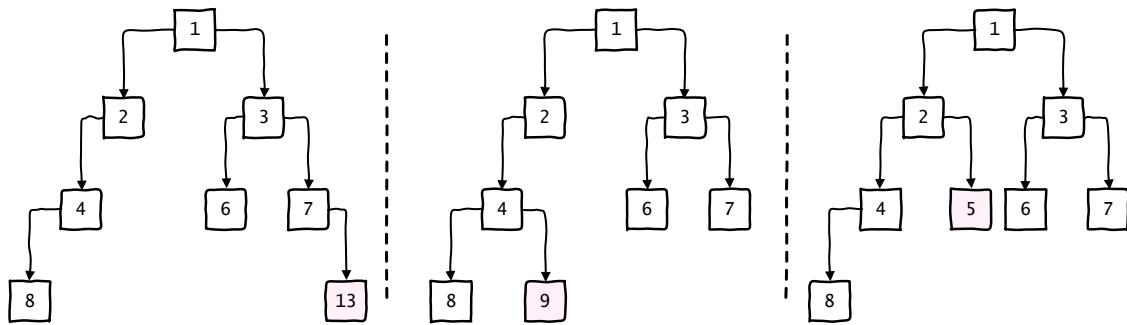


图 9.7 树高最低的二叉树

我们从这些“矮胖”的二叉树中选择其中最方便维护的一个用来实现胜者树。通常，我们使用完全二叉树来实现胜者树：如果增强的层次序列中，外部节点都位于序列的末尾，则称其为完全二叉树（complete binary tree）。在图9.7中，节点上的数字表示其在增强的层次序列中的次序。容易发现，图中所示的3棵二叉树中，只有最右边的是完全二叉树。

由8.3.4节的结论可知，规模为 n 的完全二叉树的形态是唯一的。除此之外，完全二叉树的形态满足下面的性质，您可以根据增强的层次序列，很容易自己证明。

1. 完全二叉树的叶子只出现在最低两层。
2. 完全二叉树次低层的叶子只出现在右侧，最低层的叶子只出现在左侧。
3. 完全二叉树至多有1个度为1的节点，且该节点只能出现在次低层。

最后我们推出：

定理 9.1： 在规模为 n 的二叉树中，完全二叉树的高度是最低的。

证明 设完全二叉树的树高为 h 。根据(3)，第 $0, 1, \dots, h-2$ 层上所有的节点度均为2，因此第 $0, 1, \dots, h-1$ 层上的节点数量为 $2^h - 1$ 。而第 h 层至少1个节点，所以 $n \geq 2^h$ ，即规模为 n 的二叉树至少需要 h 层。 ■

完全二叉树最重要的性质是它可以用一个向量表示，而不需要使用《二叉树》

章中的指针形式。

像图9.7中那样，我们可以将完全二叉树的根节点标号为1（一些教材^[6]可能会将其标号为0，没有实际影响），按照层次序列的次序，将节点依次标号为1, 2, 3, …，以此类推。那么，一个规模为 n 的完全二叉树可以被放在一个规模为 $n+1$ 的向量中，按照层次序列的次序，每个元素的秩依次为1, 2, 3, …, n 。从图9.7上容易观察到下面的性质：

定理 9.2：在向量表示的完全二叉树中，如果一个元素的秩为 r ，则其左右子代的秩分别为 $2r$ 和 $2r+1$ （若大于 n 则不存在）。

您可以轻易利用前面的结论证明上述结论。得益于层次序列是基于层次的，我们可以对层次进行尾部递归，即删去最下面的一层。当我们用深度定义层次时，删去最下面的一层，可以得到一个高为 $h-1$ ，规模为 $2^h - 1$ 的二叉树，该二叉树是完全二叉树、真二叉树且所有叶子都在最低层，称为**满二叉树**（full binary tree）。

另一方面，我们也可以用高度定义层次，得到下面的结论：

定理 9.3：规模为 n 的完全二叉树，去掉所有的叶子节点，得到的是规模为 $\lfloor \frac{n}{2} \rfloor$ 的完全二叉树。

这是因为，叶子节点的入队出现在层次序列的末尾，去掉叶子节点之后，仍然满足增强层次序列的外部节点全部在末尾的条件。

9.4.3 胜者树的插入 *

基于完全二叉树建立胜者树，则规模为 n 的数据需要建立规模为 $2n-1$ 的完全二叉树，其中表示“选手”的叶子节点秩为 $n, n+1, \dots, 2n-1$ ，表示“淘汰赛”的内部节点秩为 $1, 2, \dots, n-1$ ，分别存储每次比赛的胜者。

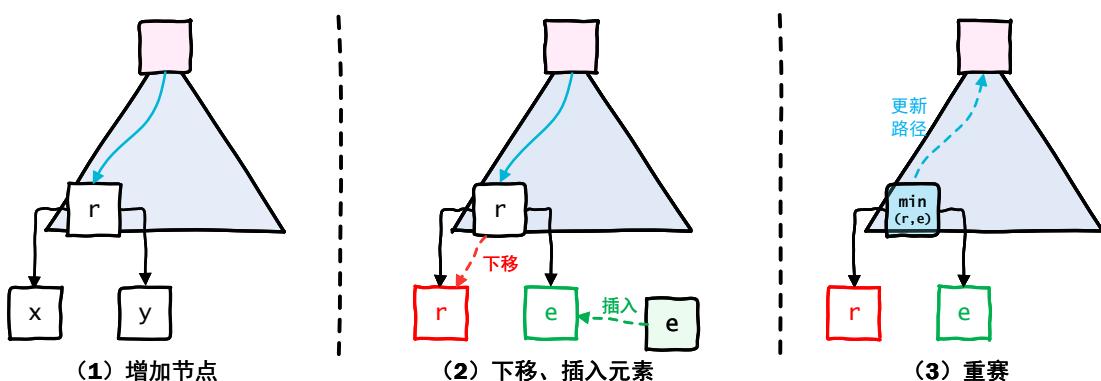


图 9.8 向胜者树中插入元素

因为每场“淘汰赛”必然有2个“选手”对阵，所以胜者树必然是真二叉树。

如图9.8所示，当向非空胜者树中插入一个新元素的时候：

1. 向量尾部增加 2 个元素 x 和 y 。在胜者树上，这两个元素都是某个原叶子节点 r 的子代。
2. 将原叶子节点 r 上的元素移动到 x 上，新插入的元素移动到 y 上。
3. 对 r 的祖先进行重新“比赛”，直到比赛结果和之前没有区别为止。

```

1 bool update(Rank r) {
2     auto m { this->cmp(V[left(r)], V[right(r)]) ? left(r) :
3         right(r) };
4     auto ret { V[m] != V[r] };
5     V[r] = V[m];
6     return ret;
7 }
8
9
10
11
12
13
14
15
16
17
18 }
```

容易看出，该算法的时间复杂度为 $O(h)$ 。如果插入的元素比较大，基本不改变比赛结果，那么算法会迅速结束；反之，如果插入的元素非常小，以至于它能成为整个胜者树的最终胜者，则算法会一直进行到树根为止，最坏时间复杂度为 $\Theta(h) = \Theta(\log n)$ 。在平均情况下，插入的元素“战胜”每个其他元素的概率为 $\frac{1}{2}$ ，所以平均时间复杂度为

$$O(1) + \sum_{k=1}^{\log n} \frac{k}{2^k} = O(1)$$

在上面的算法中，我们从 r 开始向上一层一层地判断是否需要更新，这是一个顺序查找的过程。事实上，由于在完全二叉树中，比一个节点高 k 层的祖先的秩等于该节点的秩除以 2^k ，所以在任意节点 r 到树根的路径上，可以通过折半查找来得到最后一个需要更新的位置。在很多情况下折半查找的性能都高于顺序查找（如上节介绍的插入排序），但在这个场景下并非如此。折半查找的情况下，比较次数为 $\Theta(\log \log n)$ ，高于上面计算出的顺序查找平均时间复杂度。

这一反直觉的事实所基于的原因是，如果折半查找中进入两个分支的概率严重不平衡，则折半查找的效率会显著下降。上述算法里，最后一个需要更新的位置命中高层的概率极低，而命中低层尤其是最低层的概率极高，折半查找的任何一

步里进入两个分支的概率都显著不平衡，因此折半查找的平均性能反而不如顺序查找。

9.4.4 实验：胜者树的建立 *

如果已知一整个无先验信息的序列，将其转换为胜者树，这个过程称为胜者树的建立。根据上一小节的结论，如果采用一个一个插入的方法建立胜者树，则最好和平均时间复杂度为 $\Theta(n)$ ，最坏时间复杂度为 $\Theta(n \log n)$ 。

显然，这一最坏时间复杂度令人不满意。在最坏的情况下，越晚出场的“选手”越“重量级”，最初的比较仅仅是“小打小闹”，产生的胜者仅仅是“山中无老虎，猴子称大王”，做了太多不到位工作。为了解决这个问题，应该一次到位。

1. 直接开辟规模为 $2n$ 的完全二叉树，将已知的序列直接移动到 $n, n+1, \dots, 2n-1$ 的位置上。

2. 逐一进行秩为 $n-1, n-2, \dots, 1$ 的“淘汰赛”。

```

1 void build(Vector<T>&& V0) {
2     auto n { V0.size() };
3     this->V.resize(n * 2);
4     std::move(std::begin(V0), std::end(V0), std::begin(V) + n);
5     for (Rank r { n - 1 }; r > 0; --r) {
6         update(r);
7     }
8 }
```

这个算法的时间复杂度为 $\Theta(n)$ 。在这个例子中，可以再一次看到2.5.6节介绍的“整块操作优于多次局部操作”思想。

表 9.7 胜者树建立的耗时对比（单位：ms）

n	逐个插入	一步到位
初始元素随机分布 ^① 10^4	2	1
初始元素随机分布 10^5	21	10
初始元素随机分布 10^6	212	103
初始元素单调递减 ^② 10^4	11	1
初始元素单调递减 10^5	126	9
初始元素单调递减 10^6	1462	94

^① 对于逐个插入来说，平均性能相当接近于最好情况，所以即使只进行一次随机测试也能反映性能。

^② 对于逐个插入来说是最坏情况，对于一步到位则无影响。

9.4.5 胜者树的删除 *

和插入相比，胜者树的删除更具挑战性。一方面，胜者树的最终胜者，也就是秩为 1 的元素，它可能来自于秩为 $n, n+1, \dots, 2n-1$ 的任意一个位置；另一方面，胜者树减少规模，也就意味着完全二叉树所基于的向量减少规模，而减少规模所删除的元素只能是 $2n-2$ 和 $2n-1$ 这两个位置。

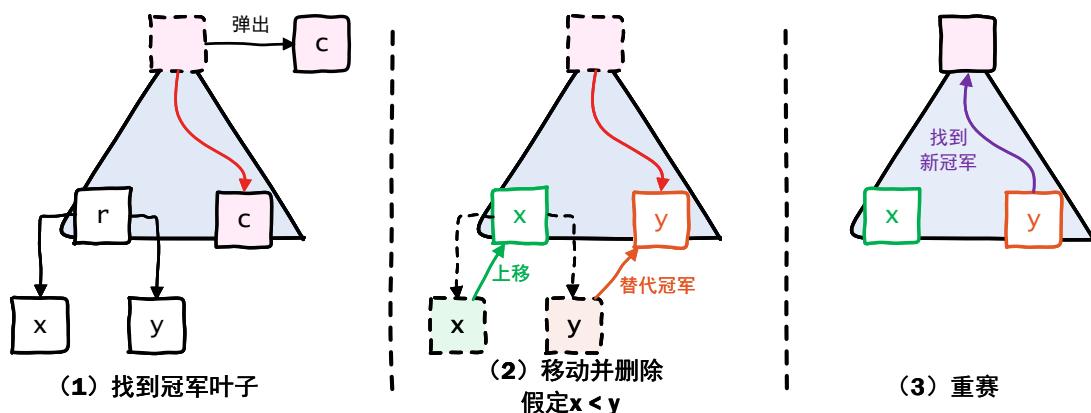


图 9.9 从胜者树中弹出最小值

这是底层向量的删除位置固定，和上层数据结构的删除位置不固定，这二者之间的矛盾。在3.6.3节中，这样的矛盾已经出现过一次，这里可以用相同的方法解决。如图9.9所示，在胜者树上删除“冠军”时：

1. 找到“冠军”所在的叶子节点 c 。
2. 设 $2n-2$ 和 $2n-1$ 的元素分别为 x 和 y ，二者的亲节点（秩为 $n-1$ ）为 r ，那么 $r = \min(x, y)$ 。
3. 将 $\max(x, y)$ 移动到 c 的位置上，然后将节点 x 和 y 删除。因为 $\max(x, y)$ 在比赛 r 中就落败，所以不会影响其祖先的值。
4. 重新进行 c 的祖先的比赛（类似插入的情况）。

您可以基于3.6.3节的经验，自己设计胜者树上删除最小值的算法，并和下面的示例程序进行对比。

```

1 T pop() override {
2     auto n { V.size() / 2 };
3     auto c { 1 };
4     while (c < n - 1) {
5         if (V[c] == V[left(c)]) {
6             c = left(c);
7         } else {
8             c = right(c);
9         }
10    }

```

```

11 auto ret { V[c] };
12 if (V[n - 1] == V[left(n - 1)]) {
13     V[c] = std::move(V[right(n - 1)]);
14 } else {
15     V[c] = std::move(V[left(n - 1)]);
16 }
17 V.pop_back();
18 V.pop_back();
19 for (c = parent(c); c > 0; c = parent(c)) {
20     update(c);
21 }
22 return ret;
23 }
```

该算法的时间复杂度为 $\Theta(\log n)$ 。

9.4.6 锦标赛排序 *

类似于此前介绍的选择排序，将同一套模板应用在胜者树上，即可得到锦标赛排序。请您自己完成锦标赛排序的算法，然后和下面的示例程序进行比较。为了提高时间效率，应采用一次到位的建立胜者树方法，而不是逐个插入。

```

1 void sort(Linear<T>& V) override {
2     if (auto n { V.size() }; n > 0) {
3         WinnerTree<T> W { std::move(V) };
4         for (Rank i { 0 }; i < n; ++i) {
5             V[i] = W.pop();
6         }
7     }
8 }
```

锦标赛排序的时间复杂度为 $\Theta(n \log n)$ ，空间复杂度为 $\Theta(n)$ 。如上一小节所述，在删除胜者树的“冠军”时会引起其他元素的位置变化，因此锦标赛排序是不稳定的。同时，因为树形结构事实上会破坏元素的局部性，所以锦标赛排序不能很好地利用整体有序的性质，自适应性不强。

9.5 完全二叉堆

9.5.1 完全二叉堆的结构

锦标赛树存储 n 个元素需要 $2n$ 个单元，存储效率非常低下。为了在这一点上进行优化，需要发现锦标赛树存储结构上的重复工作。显而易见，锦标赛树存储效率的低下，来自于一个元素会在其祖先处多次存储。越小的元素，在锦标赛树上存储的副本数量越多，最多的树根存储了 $\Theta(\log n)$ 个副本。

要优化存储效率，我们希望同时做到：

1. 保留锦标赛树作为半线性优先队列的性质，即，一个节点的值比其子代的值更小。
2. 删去锦标赛树上的多余副本，每个元素在树上只存在一个副本。

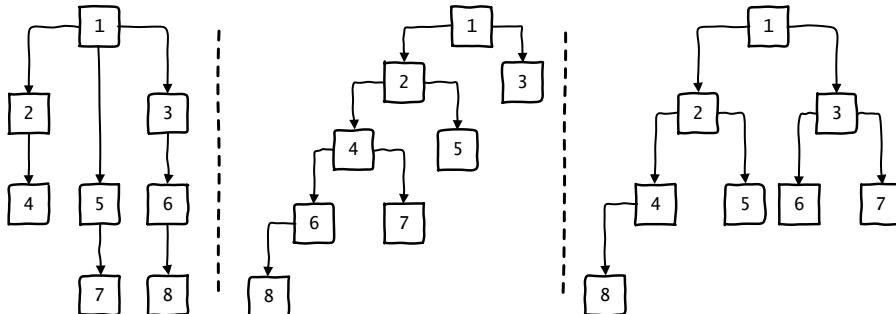


图 9.10 堆

满足上述两点的数据结构称为**堆**(heap)。像锦标赛树一样基于完全二叉树建立的堆，称为**完全二叉堆**(complete binary heap)。图9.10中的3个树都是堆，其中只有最右边的是完全二叉堆。在本书下文和常见用法(如C++中的`std::make_heap`)中，若无特别说明，堆即是指完全二叉堆。另外，请注意堆这个名词在《组成原理》等其他学科的含义会有所不同。相应地，堆的树根称为**堆顶**，像本书这样，堆顶存放最小值的称为**最小堆或小顶堆**；反之，堆顶存放最大值的则为**最大堆或大顶堆**。C++中的`std::priority_queue`默认是大顶堆。

堆中每个元素只会出现1次，因此规模为n的堆，就基于规模为n的完全二叉树，并基于规模为n+1的向量。

9.5.2 完全二叉堆的插入

向堆中插入元素的过程和胜者树是相似的。胜者树需要增加2个元素，其中一个是被插入的元素，另一个则为亲代下移得到；而堆只需要增加1个元素，然后将这个被插入的元素依次和其亲代进行比较，如果比亲代小，就将其和亲代位置交换，直到比亲代大或到达堆顶为止。这个向上移动的过程，称为**上浮**(sift up)或**上滤**。

```

1 void siftUp(Rank r) {
2     while (!isRoot(r) && this->cmp(V[r], V[parent(r)])) {
3         std::swap(V[r], V[parent(r)]);
4         r = parent(r);
5     }
6 }
7
8 void push(const T& e) override {
9     V.push_back(e);
10    siftUp(V.size() - 1);

```

11 }

为了兼容教材^[6]上堆顶秩为 0 的完全二叉堆，在示例代码中允许用户传入模板参数 `NoShift`，选择堆顶的秩是 0 还是 1。根据该模板参数的不同，本节示例代码中的 `root`、`left`、`right` 等函数的实现会有所不同。

9.5.3 完全二叉堆的删除

反过来，从堆中删除最小值，和胜者树相似而略有不同。将堆顶和最后一个元素 x 交换，然后让向量规模减 1，此时原堆顶被删除，而原最后一个元素 x 被错误地放在了堆顶的位置上。因此，接下来的过程是将 x 移动到正确的位置，这个过程称为下沉（sift down）或下滤。

```

1 void siftDown(Rank r) {
2     while (true) {
3         auto m { r };
4         if (left(r) < V.size() && this->cmp(V[left(r)], V[m])) {
5             m = left(r);
6         }
7         if (right(r) < V.size() && this->cmp(V[right(r)], V[m]))
8             {
9                 m = right(r);
10            }
11            if (m == r) {
12                break;
13            }
14            std::swap(V[r], V[m]);
15            r = m;
16        }
17    }
18 T pop() override {
19     auto ret { std::move(V[root()]) };
20     V[root()] = std::move(V[V.size() - 1]);
21     V.pop_back();
22     siftDown(root());
23     return ret;
24 }
```

9.5.4 完全二叉堆的建立

由于堆不需要额外的辅助空间，所以对于堆顶秩为 0 的情况，任何一个向量在不进行任何修改的情况下就可以视为“未堆化（heapify）”的完全二叉树，不需要像胜者树那样移动到新向量的指定位置，再移动回来。将任意一个给定的完全二叉树堆化，有两种路径：

1. 从叶子开始，按照层次序列的逆序（秩从大到小），依次将各个元素下沉。

2. 从堆顶开始，按照层次序列的顺序（秩从小到大），依次将各个元素上浮。

算法（2）等价于向一个空堆依次插入元素，其正确性是显然的。下面证明算法（1）的正确性。

证明 对于堆 H_1 和堆 H_2 以及两个堆顶的公共亲代 r ，在 r 的下沉过程中，每次都和较小的一边子代交换，所以下沉路径上的节点及其姊妹在下沉之后均满足堆的要求，从而在 r 下沉之后，得到的是一个合并的整体的堆。以此法进行递归，其边界为空树（自然是堆）。 ■

类似于上一节中对胜者树建立的讨论，完全二叉堆建立过程中，算法（1）的性能也会好于算法（2）。算法（1）被称为**弗洛伊德（Floyd）算法**，也是 C++ 的 `std::make_heap` 中使用的算法。

```

1 void build(Vector<T>&& V0) {
2     auto n { V0.size() };
3     this->V.resize(n + root());
4     if constexpr (NoShift) {
5         this->V = std::move(V0);
6     } else {
7         std::move(std::begin(V0), std::end(V0), std::begin(V) +
8             1);
9     }
10    for (auto r { n / 2 }; ; --r) {
11        siftDown(r);
12        if (r == root()) {
13            break;
14        }
15    }
}

```

和胜者树的情形一样，依次插入元素的算法（2）虽然平均时间复杂度为 $\Theta(n)$ ，但最坏时间复杂度高达 $\Theta(n \log n)$ 。这是因为算法（2）在最坏情况下，每个元素上浮的距离为其深度。反过来，在算法（1）的最坏情况下，每个元素下沉的距离仅为其高度。在完全二叉树上，对于每个节点 v 都有 $d(v) + h(v) \approx h \approx \log n$ ，深度和高度形成一个“此消彼长”的互补关系，而深度大、高度小的节点（如低层的叶子）数量远远多于深度小、高度大的节点。所以，算法（1）的最坏时间复杂度仅为 $\sum h(v) = \Theta(n)$ 。

9.5.5 实验：堆排序

类似锦标赛排序，也可以建立堆排序。因为堆不需要额外的存储空间，所以堆排序是一个就地排序，空间复杂度为 $O(1)$ 。除此之外，它和锦标赛排序一样，是不稳定的排序，也不具有自适应性。

在堆排序的实现中，有一个小技巧：每个元素的出队也就意味着向量长度减

1, 刚好腾出了一个位置可以存放出队的元素, 这样就不需要用另一个向量去存了 (由于扩容的原因, 即使两个向量的规模之和为 n , 其容量之和最大有可能到 $2n$, 也就是说, 如果用另一个向量去存, 还是有可能达到 $\Theta(n)$ 的空间复杂度)。那么, 对于小顶堆来说, 所有元素出队之后, 最小的元素会被存放在向量末尾, 整个序列成逆序状态。而如果采用大顶堆, 就可以让算法结束时整个序列自然形成顺序状态。请您基于大顶堆自行实现这一改进算法。

```

1 void siftDown(Linear<T>& V, Rank n, Rank r) {
2     while (true) {
3         auto m { r }, lc { left(r) }, rc { right(r) };
4         if (lc < n && this->cmp(V[m], V[lc])) {
5             m = lc;
6         }
7         if (rc < n && this->cmp(V[m], V[rc])) {
8             m = rc;
9         }
10        if (m == r) {
11            break;
12        } else {
13            std::swap(V[r], V[m]);
14            r = m;
15        }
16    }
17 }
18
19 void sort(Linear<T>& V) override {
20     if (int n = V.size(); n > 0) {
21         for (auto r { n / 2 }; r >= 0; --r) {
22             siftDown(V, n, r);
23         }
24         for (auto r { n - 1 }; r >= 0; --r) {
25             std::swap(V[0], V[r]);
26             siftDown(V, r, 0);
27         }
28     }
29 }
```

表 9.8 归并排序、锦标赛排序和堆排序较大规模排序时间消耗对比 (单位: ms)

n	归并排序 自上而下	锦标赛排序	堆排序
10^4	12	13	10
10^5	157	172	132
10^6	1763	2148	1584

在锦标赛排序中, 每次弹出一个元素, 需要从冠军(树根)开始向下经过 $\Theta(\log n)$ 的时间去找到其对应的叶子, 然后又需要至多 $\Theta(\log n)$ 的时间重赛; 而在堆排序

中，每次弹出一个元素，只需要至多 $\Theta(\log n)$ 的时间上浮，显著降低了时间复杂度的常数。

简单选择排序、锦标赛排序（有时也称**树形选择排序**）和堆排序通常被统称为广义的选择排序。除了同样基于优先队列的模板外，它们还都是不稳定、不自适应的排序方法。插入排序尽管也可以认为基于优先队列，但它是稳定、自适应的排序，和上述三种广义的选择排序性质不同。关于各种排序算法的具体讨论，参见《排序》章。

9.6 左式堆

9.6.1 空节点路径长度 *

完全二叉堆是最常见的优先队列，它实现简单，并且对于优先队列的各种基本操作，复杂度都比较低（平时遇到的数据规模下， $\log n$ 不会特别大，所以 $\Theta(\log n)$ 的算法可能比一些常数很大的 $O(1)$ 性能更好）。但是，完全二叉堆在处理两个堆的合并时相当棘手。现在两个规模分别为 n_1 和 n_2 的完全二叉堆要进行合并，如果您直接将它们当成 2 个无序向量做 Floyd 算法，则需要 $\Theta(n_1 + n_2)$ 的时间才能建立新的完全二叉堆，这个线性的时间复杂度显然不能让人满意。

把这两个堆当普通的无序向量去进行 Floyd 算法，损失了很多先验信息，您可能会认为利用这些先验信息能有效降低不必要工作。但是，完全二叉堆的向量表示法是利用这些先验信息首当其冲的障碍：将两个堆的向量连在一起，那么放在后面的堆的拓扑关系必定被完全打散。假设堆 H_1 放在 H_2 前，那么 H_2 因为拓扑关系被打散，故 H_2 上的先验信息无法被利用。此时如果为了利用 H_1 的先验信息，将 H_2 中的元素一一上浮，则其时间复杂度变为 $\Theta(n_2 \log(n_1 + n_2))$ ，当 $n_1 \approx n_2$ 时这个时间复杂度还不如线性的 Floyd 算法。

因此，完全二叉堆实现高效操作的法宝——完全二叉树结构和向量表示法，在处理合并问题时却因限制过于严格成为了累赘。为了高效地处理合并，有必要放松一下限制，采用其他形式的优先队列。

首先，向量表示法是一定会被抛弃的，因为离开了完全二叉树，向量表示法就不再能表示一棵二叉树上的亲子关系。其次，我们在不使用向量表示法的基础上，分析完全二叉堆的各个操作的时间复杂度来源。

1. 查找。查找最小值只需要访问堆顶即可，一定是一个 $O(1)$ 的操作。
2. 插入。插入一个元素首先需要将该元素插入到完全二叉树的末尾位置，即其增强的层次序列中的第一个外部节点的位置上，然后将这个元素上浮。因为规模为 n 的完全二叉树的形态是确定的，所以只要我们知道堆的规模，那么

即使没有向量表示法，也可以从树根出发通过 $\Theta(\log n)$ 个子代指针访问到目标位置。

3. 删除。删除堆顶的元素首先需要将完全二叉树的末尾位置上的元素（即增强的层次序列中最后一个数据节点）移动到堆顶，然后将这个新的堆顶下沉。和 (2) 相似，找到这个元素只需要访问 $\Theta(\log n)$ 个子代指针。同时，下沉的时间也被控制在 $O(h)$ 之内。

从上面的分析中可以看出，完全二叉堆即使不使用向量表示法，也可以保证插入和删除是对数复杂度的。我们还可以发现，尽管完全二叉堆的平衡性非常强，树根到每个叶子（或外部节点）的通路长度都是 $\log n$ 左右，但插入和删除中都关注了一条特定的通路，即树根到完全二叉树末尾位置的通路。我们只需要保证这条通路的长度是 $O(\log n)$ 的，就可以让插入的时间复杂度控制在 $O(\log n)$ 。

因而可以猜想，只要堆中存在一条从树根到叶子（或外部节点）的通路，其长度是 $O(\log n)$ ，就可以控制时间复杂度在对数上。事实上，任意堆都满足这个性质。定义一个节点的空节点路径长度（null path length，下简称为 npl）为其到最近的外部节点的距离，或者用下面的方法递归定义：

1. 外部节点的 npl 为 0。
2. 数据节点的 npl 是其子代 npl 中的较小值 +1。

定理 9.4： 任意二叉树的树根的 $npl = O(\log n)$ 。

证明 设二叉树树根的 npl 为 d ，则前 $d - 1$ 层内没有外部节点。从而该二叉树包含了一棵高度为 $d - 1$ 的满二叉树，即 $n \geq 2^d - 1$ ，故 $d = O(\log n)$ 。 ■

9.6.2 旋转下沉 *

因为 $npl = O(\log n)$ ，所以只要我们可以定位到最短空节点路径，就能将节点插入到该路径的末尾并进行上浮，从而将插入的时间复杂度控制在 $O(\log n)$ 。接下来只需要证明删除的时间复杂度也能控制在 $O(\log n)$ 即可。

对于完全二叉堆来说，插入的上浮路径是确定的，而删除的下沉路径是不确定的，所以需要保证每条通往叶子的路径长度都是 $\Theta(\log n)$ ，才能保证下沉的最坏时间复杂度为 $\Theta(\log n)$ 。但是反过来说，也正是因为每条通往叶子的路径长度都是 $\Theta(\log n)$ ，所以我们才可以任意地选取下沉的路径。如果堆并不是完全二叉堆，每个节点只有最短空节点路径被保证在 $O(\log n)$ ，我们可以把下沉的路径尽可能固定在最短空节点路径上。

如图9.11所示，设正在被下沉的元素为 x ， x 的子代分别为 y 和 z ，并且 x 的最短空节点路径在 y 的一侧。 z 的子代分别为 z_1 和 z_2 ，并且 z 的最短空节点路径在 z_1 的一侧。节点 y, z_1, z_2 下方的矩形代表了该节点的空节点路径相对长度， y 和

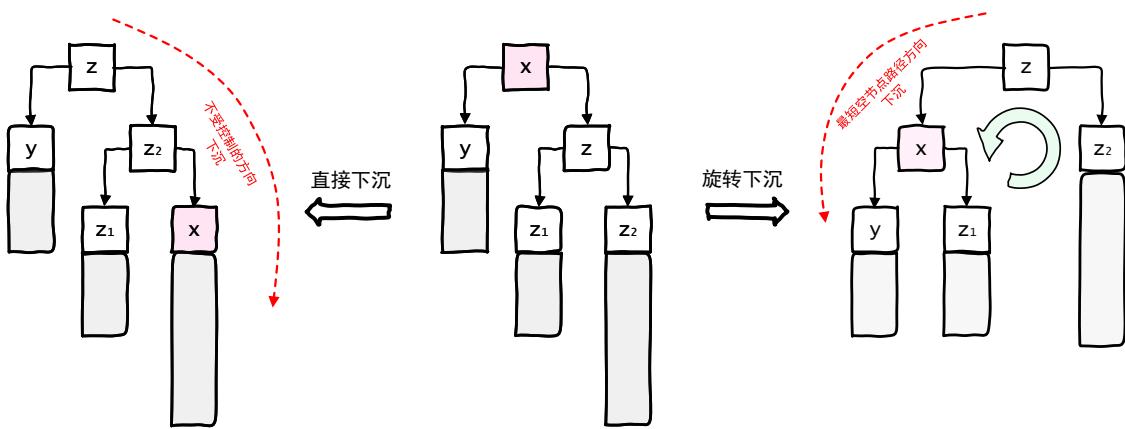


图 9.11 直接下沉和旋转下沉

z_1 是相对短的一侧, z_2 是相对长的一侧。那么 x, y, z 中:

1. 如果 x 最小, 则不需要继续下沉。
2. 如果 y 最小, 则 x 向 y 的方向下沉, 在被控制的路径上。
3. 如果 z 最小, 则根据 (1) (2), 在 x, z_1, z_2 中, 如果 z_2 最小, 按照完全二叉堆的下沉方法, 则需要 x 向 z_2 的方向下沉, 以此类推, 踏上一条不受控制的“不归路”, 如图9.11中的左图所示。

但是, 我们可以避免 x 向 z_2 这条“不归路”继续下沉。和向量表示法的完全二叉树不同, 如果是用指针来表示的二叉树, 则可以在 $O(1)$ 的时间内插入和分离子树。所以, 我们可以对二叉树进行**旋转** (在《二叉搜索树》章将详细讨论这一操作), 如图9.11中的右图所示, 将 x 转到 y 和 z_1 这两条最短空节点路径的一侧, 即可保证 x 的下沉路径长度是受控制的。

设正在下沉的元素为 x , x 的左右子代分别为 l_x 和 r_x (会随下沉变化)。考虑 $\phi_1 = \text{npl}(x)$, 则每次情况 (2) 的直接下沉都会让 ϕ_1 降低 1, 因此直接下沉的次数不超过 $O(\log n)$ 。另一方面, 考虑 $\phi_2 = \text{npl}(l_x) + \text{npl}(r_x)$, 则每次情况 (3) 的旋转下沉都会让 ϕ_2 降低 1。因此旋转下沉的次数也不超过 $O(\log n)$ 。综上所述, 下沉的过程可以保证时间复杂度为 $O(\log n)$ 。

9.6.3 堆的合并 *

因为删除堆顶也就等价于将左右子堆合并, 所以有 $O(\log n)$ 的删除算法之后, 也就得到了 $O(\log n_1 + \log n_2)$ 的合并算法。

1. 沿着 H_1 的最短空节点路径, 找到 H_1 的一个叶子 x 。
2. 将 x 从 H_1 中摘除, 作为新的堆顶, H_1 和 H_2 分别作为 x 的左右子代。
3. 将 x 下沉。

这个算法可以在对数时间里解决了合并问题，但是它的第（1）步显得很臃肿。在完全二叉堆中，我们将最后一个元素移动到堆顶再下沉，是因为向量表示法下只能删除最后一个元素。但是在不使用向量表示法表示的堆中，第（1）步的作用只是构造了一个堆顶而已。为了构造一个堆顶，花费 $O(\log n_1)$ 的时间从 H_1 中找出一片叶子是没有必要的。

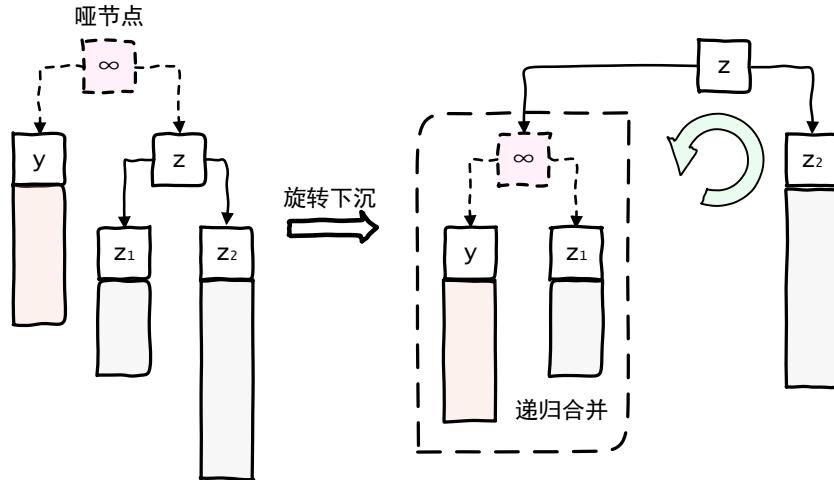


图 9.12 合并两个堆

如图9.12所示，意识到第（1）步只是构造了一个堆顶之后，我们就可以人为地制造一个值为 ∞ 的哑节点作为堆顶。由于堆的性质，这个堆顶在第（3）步一定会被下沉到叶子，最后再将这片 ∞ 的叶子删去即可。进一步地，这个哑节点也不需要被显式地构造出来。如图9.12所示，可以直接将图中的 y 和 z 合并递归到 y 和 z_1 合并，而不需要实际地创建节点 ∞ 并进行下沉。

1. 递归边界：任何堆和空堆合并得到其本身。
2. 对于非空堆 H_1 和 H_2 ，不妨假设 H_1 的堆顶更小（否则交换）。设 H_1 在最短空节点路径的一侧子堆为 H_{11} ，另一侧的子堆为 H_{12} 。保留 H_1 的堆顶和 H_{12} ，递归地将 H_{11} 和 H_2 合并，代替原 H_{11} 的位置。

尽管表述为了递归的形式，但上述算法的本质仍然和上一小节展示的旋转下沉是一致的。

9.6.4 左偏树 *

在上一小节我们知道，npl 受控制的堆可以保证插入和删除的时间复杂度在 $O(\log n)$ 之内。不过，在上一小节的分析中缺失了一环：需要有一种手段来找到受控制的 npl。通常，我们会在每个节点处记录该节点的 npl，这样只需要比较左右子代的 npl 哪个更小，就可以确定哪一条 npl 是受控制的。

为了防止这样的比较操作反复进行，我们可以在让两棵子树建立姊妹关系的

时候，就对它们树根的 npl 进行比较，让 npl 较小的子树在右边（出于习惯），npl 较大的子树在左边。这样，任意子树受控制的 npl，总是在它的最右侧通路上。这样的堆称为**左式堆** (leftist heap，也称为**左偏树**)。

您可以根据上一节介绍的堆的合并算法，自行设计左式堆的合并算法，并和下面的示例代码进行对比。

```

1 struct Element {
2     T value;
3     size_t npl { 1 };
4 };
5 size_t npl(const BinaryTreeNodeInst<Element>& node) const {
6     return node ? node->data().npl : 0;
7 }
8 BinaryTreeNodeInst<Element> merge(BinaryTreeNodeInst<Element>
9     >&& H1, BinaryTreeNodeInst<Element>&& H2) {
10    if (!H1) return std::move(H2);
11    if (!H2) return std::move(H1);
12    if (this->cmp(H2->data(), H1->data())) {
13        std::swap(H1, H2);
14    }
15    H1->right() = merge(std::move(H1->right()), std::move(H2));
16    H1->right()->parent() = H1;
17    if (npl(H1->left()) < npl(H1->right())) {
18        std::swap(H1->left(), H1->right());
19    }
20    H1->data().npl = npl(H1->right()) + 1;
21    return H1;
}

```

如前所述，左式堆的删除并不需要像完全二叉堆那样提出一个叶子作为堆顶，再将其下沉，而是可以直接通过合并两个子堆实现。同样地，左式堆的插入也需要将原堆和待插入元素组成的平凡堆合并。

```

1 void push(const T& e) override {
2     H = merge(std::move(H), BinaryTreeNodeInst<Element>::make(e
3         ));
4 }
5 T pop() override {
6     T e { std::move(H.root()->data().value) };
7     H = merge(std::move(H.root()->left()), std::move(H.root()->
8         right()));
9     return e;
}

```

9.6.5 随机堆 *

左式堆的对数复杂度依赖于总是选取最短空节点路径（最右侧通路）进行下沉，这使得我们需要存储 npl 的值，消耗额外的 $\Theta(n)$ 空间。但从9.6.1节中我们知

道，任何树的根节点 npl 都是 $O(\log n)$ 的。可以猜想，从根节点向下，即使每一步都等概率地随机向左或向右，也有很大的概率能在 $O(\log n)$ 的时间里探索到外部节点。极端情况下，如果堆是链形的，则它有 $\frac{1}{2}$ 的概率仅通过 1 步就到达外部节点，路径长度的数学期望仅为 2。另一个极端情况则是完全二叉树，它的路径长度期望也仅仅是大约 $\log n$ 而已。

从极端情况的分析可以猜想，二叉树越“对称”、左右子树的规模越接近，路径长度的数学期望越高。如果左右子树的规模相差较大，则规模小的一边会有很靠近树根的外部节点大概率被选中，从而拉低路径长度的整体数学期望。二叉树最“对称”的情况是满二叉树，此时路径长度为 $\log(n + 1)$ 。于是，可以得到下面的猜想并证明。

定理 9.5：对于任意规模为 n 的二叉树 T ，如果从根节点向下，每一步都等概率随机向左或向右，则到达外部节点的期望路径长度 $p(T) \leq \log(n + 1)$ 。

证明 递归边界（平凡树）成立并且能取等号。

设结论对规模小于 n 的树成立。设规模为 n 的树 T 的左右子树为 T_1, T_2 ，规模为 n_1, n_2 ，则

$$\begin{aligned} p(T) &= 1 + \frac{p(T_1) + p(T_2)}{2} \leq 1 + \frac{\log(n_1 + 1) + \log(n_2 + 1)}{2} \\ &= 1 + \log \sqrt{(n_1 + 1)(n_2 + 1)} \leq 1 + \log \frac{(n_1 + 1) + (n_2 + 1)}{2} = \log(n + 1) \end{aligned}$$

■

由此，衍生出了左式堆的一个变形：**随机堆** (randomized heap)。随机堆和左式堆的实现几乎相同，但它不存储 npl，也不会根据 npl 交换左右两子树。相应地，哑节点 ∞ 并不会总是向右侧旋转下沉，而是随机地向左侧或右侧旋转下沉。从平均时间的角度，随机堆的插入和删除也能达到对数复杂度。

9.7 本章小结

优先队列是算法设计中经常用到的数据结构。优先队列可以用线性或半线性结构实现，出于效率考虑通常采用半线性结构。本章介绍了基于无序序列和有序序列的线性优先队列，并介绍锦标赛树、完全二叉堆和左式堆三种基于半线性结构的优先队列。请您自己根据自己的理解，列出这些数据结构的对比表格。同时，本章也介绍了选择排序、插入排序、锦标赛排序和堆排序，以及许多评价排序算法的角度，您也可以列出这些排序算法的对比表格。

本章的主要学习目标如下：

1. 您理解了优先队列和排序的关系，一种优先队列可以导出一种排序算法。
2. 您理解了选择排序和插入排序及其变形的原理，了解从比较次数、移动次数、就地性、稳定性、迁移到列表的能力、自适应性等方面评价排序算法，知道插入排序是小规模数据的最优排序算法。
3. 您理解了完全二叉树是基于层次序列定义的，可以被向量形式存储。完全二叉树极强的性质有助于保证对数复杂度，但也会成为算法设计的桎梏。
4. 您理解了完全二叉堆和锦标赛树的区别在于是否保存副本。完全二叉堆具有更高的存储效率。
5. 您了解到折半查找中如果进入两个分支的概率严重不平衡，效率会显著下降，甚至低于顺序查找。
6. 您了解到在二叉树中，大部分的节点深度远大于高度，从而造成了纵向的不平衡性。这一不平衡性使得完全二叉堆上浮的成本远大于下沉。
7. 您了解到可以沿着最短空节点路径，通过旋转下沉保证堆操作的对数复杂度。

第 10 章 树的应用

在《优先队列》章中，无论是锦标赛树还是完全二叉堆，都是基于完全二叉树这一十分特殊的二叉树实现的，而完全二叉树的定义依赖于层次序列。层次的划分破坏了树固有的半线性递归特点，采用半线性结构来存储优先队列更多地是看中了其的**偏序性**。另一方面，《二叉搜索树》章介绍的二叉搜索树依赖于中序序列，这是二叉树特有的特征，一般的树也是没有的。

本章将会回到一般的树的**半线性**特点上来，介绍树的一些衍生数据结构。

10.1 表达式树

10.1.1 表达式树的结构

在《栈》章中介绍了表达式和栈的关系，而在《树》章中介绍了栈和树的转换，因此，在表达式和树之间建立联系是十分自然的。现在，我们补上“表达式-栈-森林-树/二叉树”联系的最后一块拼图。

回顾一下在《栈》中的讨论，一个表达式可以用栈表示，其中入栈序列为前缀表达式，出栈序列为后缀表达式，操作序列代表着表达式的计算次序。因为每个运算符需要的操作数数量是已知的先验信息，所以前缀表达式或者后缀表达式中之一，就能完整蕴含一个表达式包含的所有信息。

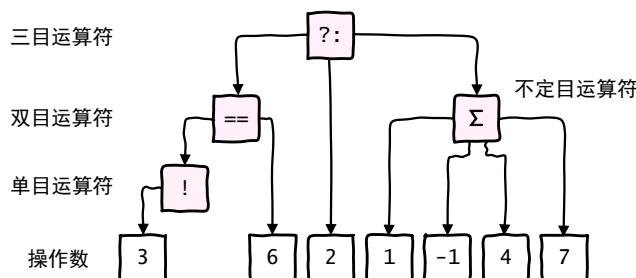


图 10.1 表达式树的结构

反映到树上，前缀表达式对应树的先序序列，后缀表达式对应树的后序序列，而每个运算符需要的操作数数量，就是该运算符所对应节点的子代数量。如图10.1所示，单目运算符（如阶乘）只有 1 个子代，双目运算符有 2 个子代，而所有的操作数都是叶子。在本书所采用的示例程序中，操作数只能出现在运算符的左侧或右侧，所以产生的表达式树形态上像一棵二叉树。但是，必须明确，表达式树不是二叉树。

1. 从图10.1中可以看出，如果表达式中有三目（比如 C++ 中的条件运算符?:）或更高目的运算符（比如 Σ 可以视为任意多的操作数求和），表达式树也可以支持。
2. 表达式树是表达式经栈作为跳板转换成森林得到的直接产物，并没有经长女-次妹表示法转换为二叉树。这里特别需要注意是转换为森林，因为表达式只会返回 1 个值，对应的栈的第一个 \vee 和最后一个 \wedge 必定对应，所以这个森林里只有一棵树，从而形态上显示为表达式树而不是表达式森林。

10.1.2 表达式树的计算

在表达式-栈-森林的关系中，栈的操作序列既对应了表达式的计算次序，又对应了森林的形态。因此，表达式树相当于显式地将表达式如何计算表达了出来，相对于前缀表达式和后缀表达式，表达式树更适合人阅读。因此，在处理表达式相关的题目时，经常首先在纸上将表达式转换为表达式树的形式，再根据需要进行后续的转换。

表达式树的计算是简单的树后序遍历。因为后序序列也就是后缀表达式，所以该过程和计算后缀表达式的过程高度相似。下面的示例程序采用了递归版本，比起显式使用栈代替递归更加简洁，但在表达式递归深度较长的时候会发生栈溢出错误，参见 *Expression.cpp* 中的实验。

```

1 int cal(TreeNodeConstPos<ExpressionElement> node) {
2     int left { node->children().size() > 0 ? cal(node->child(0)
3         ) : 0 };
4     int right { node->children().size() > 1 ? cal(node->child
5         (1)) : 0 };
6     return node->data().apply(left, right);
7 }
```

可以看到，表达式树很好地利用了树的半线性性质：树上的任何一个节点为根都是一棵子树。一棵子树也就对应了一个子表达式，在表达式中，一个子表达式首先在内部进行计算，并以计算出的结果参与到表达式其他部分的后续计算。正如树上不包含的两棵子树没有互不相交一样，在表达式中，两个子表达式的运算先后也和表达式的计算结果无关。上述示例代码借助半线性递归，可以计算任意节点的对应的子表达式的值，而整个表达式树的值，就是树根对应的子表达式的值。

值得一提的是，因为在本书示例代码中 `apply` 被设计为了二元运算且区分左右，因此上面的代码需要区分 `left` 和 `right`，如果 `apply` 原生支持对 `const vector&` 的求值，可以进一步简化表达式树的代码设计。

10.1.3 表达式树的建立 *

类似地，后缀表达式转换为表达式树，就是利用后序序列，以及先验的节点度数信息，去重建树的过程。其过程同样和后缀表达式的计算高度相似。这里需要注意栈内的操作数（或子表达式）是先右后左的，因此需要做子代向量的头插`push_front`以进行逆序，或者尾插之后使用`std::reverse`等方法逆序。

```

1 ExpressionTree suffix2tree() {
2     Stack<TreeNodeInst<ExpressionElement>> S;
3     for (auto& e : *this) {
4         auto node { TreeNodeInst<ExpressionElement>::make(e) };
5         auto [l, r] { e.operandPosition() };
6         for (auto pos : { r, l }) {
7             if (pos) {
8                 auto child { S.pop() };
9                 child->parent() = node;
10                node->children().push_front(std::move(child));
11            }
12        }
13        S.push(std::move(node));
14    }
15    return ExpressionTree { std::move(S.pop()) };
16 }
```

当所有的运算符均为单目或双目运算符时，表达式树呈现出二叉树的形态，此时，去掉所有括号的中缀表达式对应二叉树的**中序序列**。这一点可以由上一小节中，子表达式对应子树得到解释。所以中缀表达式转换为表达式树，则相当于利用中序序列，以及先验的节点度数信息去重建二叉树。

这个过程用程序实现是复杂的。在8.3.1节中，我们用中序序列和先序序列重建二叉树，总是顺序遍历先序序列，而将中序序列作为一个查找表使用，就是因为如果反过来依据中序序列从左到右建立二叉树，则需要不断上上下下。因此，在程序实现的时候，通常考虑仿照计算中缀表达式的做法。在4.4.8节中，中缀表达式的计算事实上相当于将中缀表达式转换成后缀表达式的过程，与后缀表达式计算的过程整合起来。相同的技术也可以用在中缀表达式转换为表达式树上，您可以很容易完成这一程序的设计。

另一方面，正如8.3.1节所述，如果用表格法在纸上重建二叉树，则中序序列也可以发挥很好的作用。同样地，如果用在纸上将中缀表达式转换为表达式树，这一过程也非常容易。由于我们对中缀表达式的熟悉性，我们可以迅速看出中缀表达式的计算次序，从而可以区分各个运算符所在的层次，如图10.2所示。

1. 将中缀表达式上的各个操作数写在二叉树的最低层，作为表达式树的叶子。
2. 按照中缀表达式的计算次序，将各个运算符依次抬升。越晚计算的运算符，在

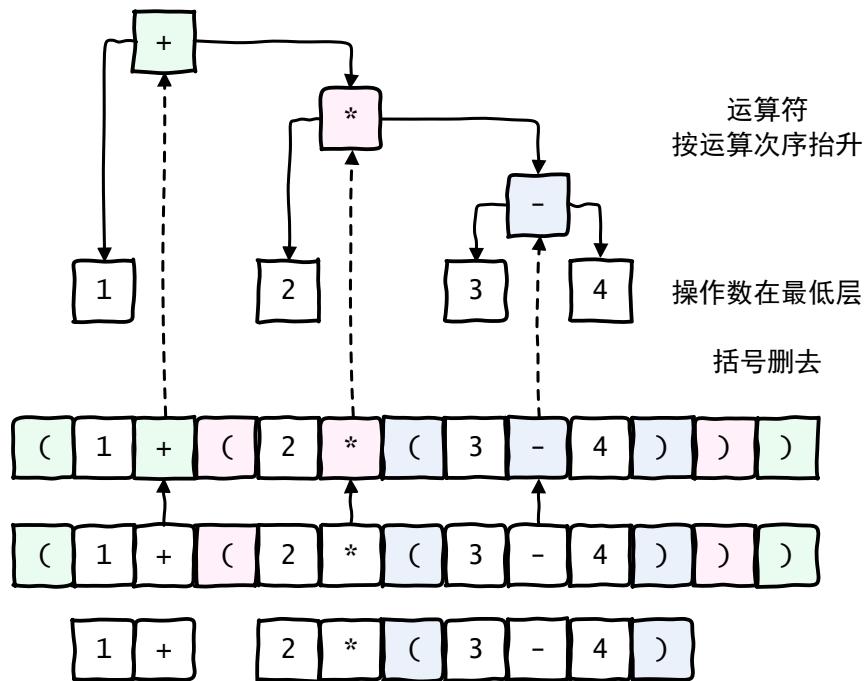


图 10.2 中缀表达式转换为表达式树

二叉树上的位置越高。

到此为止，我们建立了表达式、栈、森林、树、二叉树之间完整的关系网，如图10.3所示。

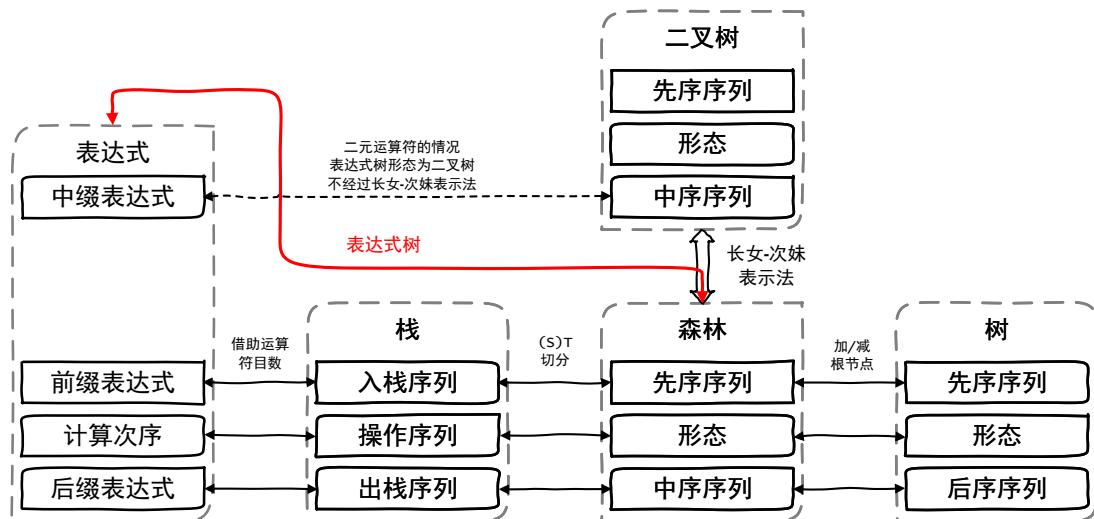


图 10.3 表达式、栈、森林、树、二叉树的关系

10.1.4 抽象语法树 *

表达式树实际上是**抽象语法树** (abstract syntax tree, AST) 的一个特殊形态。上文提到，表达式树上可以有多目运算符，比如 Σ 可以认为是接受不定数量的操作

数的运算符。而在编程语言中，一种语法也可以认为是一个运算符，下表举了一些例子。

表 10.1 C++ 语法视为运算符的例子

语法	操作数数量	格式
语句块	不定	{ a; b; c; ... }
if	2	if (a) b
if/else	3	if (a) b else c
for	4	for (a; b; c) d
基于范围的 for	3	for (a : b) c

和表达式树不同，当将程序组织成树的时候，“运算符”对应的每个“操作数”可能有不同的要求。比如，基于范围的**for**的第一个参数 a 必须是变量声明。在建立抽象语法树的时候，我们首先会建立 a 所对应的子树，该子树的“计算”可以得到一些信息，其中就包含了计算结果的“类型”。越复杂的编程语言，包含的类型信息也就越丰富。这些信息会指导编译器判断程序是否有语法错误。比如，对于下面的程序

```

1 for (1 : "abc" + 9) {
2     // ...
3 }
```

编译器会检测到，基于范围的**for**的第一个参数为 1，类型为整数字面量，不符合要求，从而报出一个错误。接下来，其第二个参数为 "abc" + 9，对这个子“表达式”进一步进行解析，会发现运算符 **operator+** 没有针对字符串字面量 "abc" 和整数字面量 9 的重载，从而报出第二个错误。

关于抽象语法树，更为详细的讨论参见《编译原理》。

10.2 编码树

10.2.1 编码效率问题 *

编码是计算机研究中的一个基础问题，在《组成原理》和《网络原理》中将进一步展开，介绍具体的编码方法。本节讨论比较抽象的编码问题。

在现实世界中的信息是多种多样的，但计算机使用的是二进制，所有的信息都必须转化成二进制数据才能在计算机中存储和计算。信息转换成二进制数据的过程称为 **编码** (encoding)，将二进制数据还原成原始信息的过程称为 **解码** (decoding)。

在编码问题中，信息的基本单位称为**字符**（character）。一种编码方法能够编码的字符必然是有限的，这些字符组成的有限集合称为**字母表**（alphabet）。

尽管字符编码之后会变成二进制数据，但其所蕴含的信息量与其对应的二进制比特位数并不吻合。比如，我们知道在经典的 ASCII 编码系统中，英文字母、数字和标点符号都被编码为 8 位的**char**。那么，如果将一个单词视为一个字符，词典视为字母表，则相同长度的单词，蕴含的信息量可能会大相径庭。比如，同样是 3 个字母组成的单词，虚词 the 包含的信息少得可怜，在绝大多数情况下即使省略它也不影响意思的传递，而缩写 BBC（代表不列颠广播公司）代表了一个具体实体，包含的意思显然要多得多。

一般而言，如果一个字符出现的概率较高，就意味着其包含的信息较少；反之，如果一个字符出现的概率较低，就意味着其包含的信息较多。如果不用单词作为字符，而是用单个的字母，这个结论也是成立的。常见的字母，如 a、e，出现的概率很高，看到它也不知道在什么单词里，故包含的信息较少；而不常见的字母，如 x、z，看到它基本就能联想到一些单词（如 X-ray），故包含的信息较多。

根据2.6.6节中的讨论，我们可以得到，一个未知字符所包含的平均不确定性，即其信息熵为：

$$H = - \sum_{i=1}^n p_i \log p_i$$

其中 p_i 为每个字符出现的概率。显然，当每个字符出现的概率均等时，信息熵最大，为

$$H_{\max} = \log |\Sigma|$$

其中 $|\Sigma|$ 为字母表的大小。对于英文字母而言，字母表的大小是 26，理论最大信息熵为 $\log 26 \approx 4.7$ ，但实际上由于不同字母出现的频率差异很大，其信息熵大约只有 4.14；以单词为单位，英文的信息熵也只有大约 11.82^[28]。但是，显然不可能平均每个单词只有 $\frac{11.82}{4.14} \approx 2.85$ 个字母。为什么会造成这种情况呢？因为用上面的公式研究字母，只会考虑字母本身，而不会考虑字母之间的相互关系（即交叉熵）。同样地，也可以研究单词之间的相互关系，这就会使问题变得更加复杂。在本书中只研究简化的模型，即假定各个字符独立，没有相互关系的情况。

假设一个字符需要 b_i 个比特进行编码，则编码长度的数学期望为

$$E(b) = \sum_{i=1}^n p_i b_i$$

给定字母表和各字符的出现概率的情况下，我们希望编码长度的数学期望尽

可能小。比较上述数学期望和信息熵的公式就可以发现，如果满足 $b_i = -\log p_i$ ，则该编码的效率最高，每 1 个比特能削减 1 的信息熵。当然，由于 $-\log p_i$ 通常不是整数，事实上很难达到这样的编码效率。

10.2.2 前缀无歧义编码

我们希望设计一个编码系统，达到尽可能高的编码效率。设计编码系统，首先需要明确编码系统需要满足的条件。

单射的条件是显然的。编码规则必定不能让两个不同的字符映射为同样的二进制数据，否则解码方将无从下手。

其次，在一长串的二进制数据中，并没有“分隔符”（如空格或换行）将不同字符的编码分割开。比如，考虑这样的一个对英文字母表的编码规则：A 编码成 1，B 编码成 11，C 编码成 111，以此类推。那么无论是什么内容，编码之后都会得到一大串 1，也同样无从编码。

您可能会说，那么人为地增加“分隔符”不就可以了？比如，在上面的编码规则中，每两个字符之间插入一个 0，就能够分辨不同字符的边界了。但是，这一做法事实上等同于所有的字符编码长度都增加了 1，即将 A 编码成 10，B 编码成 110，C 编码成 1110，以此类推。这并不是真的制造了分隔符，只是修改了编码规则使其看起来有分隔符而已。

从这个例子中，您应该能够发现，造成解码困难的原因是：一些字符的编码是另一些字符编码的前缀。如果字母表中任意两个字符的编码互相不为前缀，则这种编码方案称为**前缀无歧义编码**（prefix-free code, PFC）。PFC 一定能在解码时避免歧义，但并不是只有 PFC 才能避免歧义。比如，A 编码成 1，B 编码成 10，C 编码成 100，以此类推。这个编码规则显然不是 PFC，但解码方可以通过数每个 1 后面有几个 0 来分辨不同的字符。具体何种编码可以避免解码歧义，参见《形式语言与自动机》。本书只讨论 PFC 编码。

10.2.3 二叉编码树

对于任意一种编码方式，可以将其转换为一棵**二叉编码树**。对于一个字符 c ，设其编码为 $b_i (0 \leq i < n)$ ，其中每个位 $b_i = 0$ 或 1。从二叉树的树根出发，依次参考每个 b_i ，如果 $b_i = 0$ ，则转向左子代，如果 $b_i = 1$ ，则转向右子代（0 左 1 右是习惯规定）；这样参考完整个编码之后到达的节点，就是字符 c 对应的节点。

图10.4展示了两个编码规则所对应的二叉编码树，其中左边是 PFC 编码树，右边是上一小节所举的非 PFC 的无歧义编码例子。二叉编码树具有下面的性质：

1. 由于单射的要求，任何节点只能对应至多一个字符。

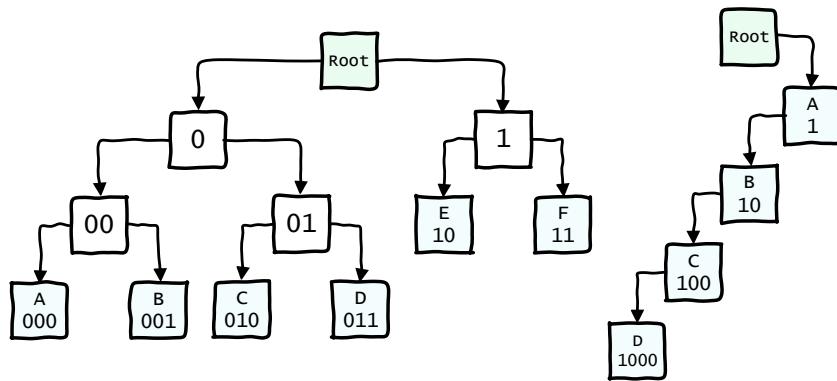


图 10.4 二叉编码树

2. 所有的叶子节点都必须对应某个字符，否则它们没有理由被添加到二叉编码树中。

3. 在 PFC 编码树中，只有叶子节点才会对应某个字符。

对 PFC 编码的解码非常简单。从树根出发，考虑编码的每一位，按照“0 左 1 右”的规则向下试探，每次试探到叶子节点时，就输出相应的字符，然后返回树根。比如，图10.4左边的 PFC 编码树上如果解码序列“00110011”，则先顺着“001”到达 B，然后返回树根，再顺着“10”到达 E，再返回树根，最后顺着“011”到达 D，解码结束，解码结果是 BED。

```

1 std::string decode(std::vector<bool> bits) const {
2     std::string str {};
3     auto node { this->root() };
4     for (auto bit : bits) {
5         if (bit) {
6             node = node->right();
7         } else {
8             node = node->left();
9         }
10        if (node->isLeaf()) {
11            str += node->data();
12            node = this->root();
13        }
14    }
15    return str;
16 }
```

在实际通信技术中通常采用模拟信号传输数据，信号的电平不会只有 2 级。比如，100G 以太网标准在物理层采用 PAM4 技术，具有 4 级电平，此时物理层的编码树将会变为四叉树（但这和 ASCII 等表示层编码规则无关，参见《网络原理》）。从理论分析而非技术实现的角度，多叉编码树的情况和本书中讨论的二叉编码树基本一致，故不再赘述。

10.2.4 最优编码树 *

在二叉编码树中，一个节点的深度就是其对应的编码长度。因此，PFC 编码的平均编码长度，也就是 PFC 编码树的**平均叶深度**（average leaf depth, ald）。在给定字母表和字符出现概率的情况下，平均编码长度最短（即 ald 最小）的规则称为**最优编码规则**，对应的编码树称为**最优编码树**。

定理 10.1： 最优 PFC 编码树必然是真二叉树。

证明 用反证法。如果一个节点只有一个子代，那么可以让这个子代代替它的位置，其所有后代叶子节点的编码长度都会减 1，从而使 ald 更小。 ■

下面，本小节讨论一个简单的情况。如果每个字符出现的概率都相等，应该如何设计最优 PFC 编码树呢？根据之前的讨论，每个字符的编码长度 $b_i = -\log p_i$ 也应该相等，即编码树中的每个叶子大致处在同一深度下。此前介绍的完全二叉树，恰好就满足这一形态特征。当然，我们知道，通常情况下完全二叉树并不是唯一高度最低的二叉树，那么它也同样通常不是唯一的最优 PFC 编码树形态。事实上，最优 PFC 编码树只需要满足完全二叉树形态特征中的一条。

定理 10.2： 当每个字符出现的概率相等时，最优 PFC 编码树的叶子都分布在最低层和次低层。

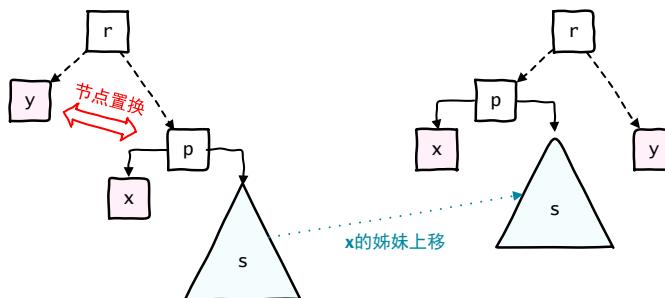


图 10.5 节点置换法

证明 用反证法，如图10.5所示，假设一棵最优 PFC 编码树的两个叶子节点 x 和 y 的深度差至少为 2。

不妨设 $d(x) - d(y) \geq 2$ ，设 p 是 x 的亲节点，则 $d(p) = d(x) - 1$ 。将 p 为根的子树和 y 的位置交换。则交换之后 $d(x') + d(y') = (d(y) + 1) + (d(x) - 1) = d(x) + d(y)$ ，即交换前后 x 和 y 这两片叶子的深度和不变。但由于最优 PFC 编码树必定是真二叉树，所以 x 存在姊妹节点，而 x 的姊妹节点为根的子树上，所有叶子深度都降低了。因而交换之后的 ald 更小。 ■

上面的证明中采用了**节点置换**的方法。和《二叉树》章介绍的利用 LCA 一样，节点置换也是证明二叉树相关题目时的典型方法之一（当然更多的题目只要用递归法就能解决）。它主要用来说明某种特殊的拓扑结构所具有的性质。

上述结论表明，当字符出现概率相等时，所有的最优 PFC 编码树都可以通过改变完全二叉树中最低层叶子节点的位置得到。

10.2.5 Huffman 树

本节讨论一般的情况，如果给定字母表和不等的字符出现概率，如何构造一棵最优 PFC 编码树呢？

首先，考虑递归边界的情况。对于规模为 1 的字母表，编码树只有 1 个根节点。规模为 1 的字母表不包含任何信息（收到字符之前就能知道是什么字符），所以也不需要传递任何数据，唯一的字符编码长度为 0。

其次，对于一个规模为 n 的字母表，考虑字母表中概率最低和次低的两个字符 x 和 y ，因为 x 概率最低，所以 x 必定是最深的叶子（否则和最深的叶子交换， ald 降低）。上一小节已经证明最优 PFC 编码树是一个真二叉树，所以 x 必有姊妹节点，若其不为 y ，将 y 和其交换 ald 不增，因此不妨令其为 y 。综上可知，存在最优 PFC 编码树，使 x 和 y 为姊妹节点。

考虑另一张字母表，这个字母表里没有 x 和 y 这两个字符，而是增加了一个字符 z 。使用这个字母表传递信息时，凡是出现 x 或 y 的地方都用 z 代替，从而有 $p_z = p_x + p_y$ 。在原来的 PFC 编码树中，只要把 x 和 y 这两片叶子删去，用它们的公共亲代代表 z ，就化归成了这个规模 $|\Sigma| - 1$ 的字母表的情况，如此递归下去，最终能达到递归边界。

上述算法是一个尾递归（减治），下面直接给出它的迭代形式。这里因为我们需要关注字母表中的概率最低和次低的字符，所以需要使用优先队列。容易证明其时间复杂度为 $\Theta(n \log n)$ ，这里 $n = |\Sigma|$ 。

```

1 struct SubTree {
2     BinaryTreeNodeInst<char> node;
3     double frequency;
4 };
5 EncodingTree operator() (const std::map<char, double>&
6     frequencies) override {
7     Heap<SubTree> H;
8     for (const auto& [c, f] : frequencies) {
9         H.push(SubTree { BinaryTreeNodeInst<char>::make(c), f });
10    }
11    while (H.size() > 1) {
12        auto t0 { H.pop() }, t1 { H.pop() };
13        auto node { BinaryTreeNodeInst<char>::make(0) };
14        (node->left() = std::move(t0.node))->parent() = node;
15        (node->right() = std::move(t1.node))->parent() = node;
16        H.push(SubTree { std::move(node), t0.frequency + t1.
17            frequency });

```

```

16     }
17     return EncodingTree { std::move(H.pop().node) };
18 }
```

10.2.6 建立最优 PFC 编码树的时间复杂度 *

本小节证明建立最优 PFC 编码树的最坏时间复杂度为 $\Omega(n \log n)$ 。这一问题可以参考 2.6.5 节的方法。对于有 n 个字符的字母表来说，其对应的 PFC 编码树为有 n 个叶子节点的真二叉树。删除所有叶子节点之后，为有 $n - 1$ 个节点的任意形态的二叉树，其形态数量总共有 $\text{Catalan}(n - 1)$ 个，同时， n 个字符在 n 个叶子节点上的排列次序一共有 $n!$ 种，因此，在算法开始时，可能的树数量为：

$$\text{Catalan}(n - 1) \cdot n! = \frac{(2n - 2)!}{(n - 1)!}$$

有一定难度的问题是，算法结束时可能的树有多少棵？注意到在上一小节的算法中是让 x 在左侧、 y 在右侧的，而如果让 y 在左侧、 x 在右侧，就可以得到另一棵最优 PFC 编码树。类似地，对于最优 PFC 编码树的每个内部节点，将其左右子树互换，都可以得到另一棵最优 PFC 编码树。

定理 10.3： 在最坏情况下，所有的最优 PFC 编码树都可以通过若干次交换左右子树的方法得到。

证明 最坏情况下，在 Huffman 树建立过程中所有的字符（包括对应叶子的初始字符，和对应子树的后续被加入到字母表里的字符）出现的概率都互不相同。那么在上一小节的推导过程中，如果 y 不是 x 的姊妹，那么 y 和 x 的姊妹交换， ald 就从不增变为了减少，不满足最优 PFC 编码树的条件，从而 x 和 y 必定是姊妹，只有左右的区别。递归可证其他内部节点的左右子树也都必定是姊妹，故所有最优 PFC 编码树都可以通过交换左右子树得到。 ■

于是，在算法结束时，由于每个内部节点都可以左右子树互换，所以一共有 2^{n-1} 棵可能的树。那么根据信息熵理论和 Stirling 公式，比较的次数至少为

$$\log \left(\frac{(2n - 2)!}{(n - 1)!} \right) - \log(2^{n-1}) = \Theta(n \log n)$$

因此，上一小节所示的 Huffman 树构造算法时间复杂度已达到最优。

10.3 并查集

10.3.1 并查集的结构

并查集 (union-find set) 是一种特殊的集合结构。并查集事实上是一个集合的集合，可以表示为 $\mathcal{U} = \{S_1, S_2, \dots, S_m\}$ 。它满足：

1. 互不相交：对任意的 $i \neq j$, $S_i \cap S_j = \emptyset$ 。
2. 支持任意两个集合的合并（union）：将 S_i 和 S_j 合并加入到 \mathcal{U} 中，而将原先的这两个集合删去以保持互不相交。
3. 支持查找（find）任意一个元素所在的集合。

因为性质（2）（3）只支持合并和查找，所以被称为并查集。另一方面，因为性质（1）互不相交，所以也被称为**不交集**（disjoint set）。

并查集是一类实际问题的抽象。比如网络管理员想要判断两个主机是否可以建立连接，则可以让每一个网络作为一个 S_j 。如果发现两个网络 S_i 和 S_j 之间有交换设备连通，那么就将它们合并。这样，只要查找两个主机所在的集合是否相同，就可以判断它们是否可以建立连接了。在《图》章中也将看到并查集在《数据结构》中的一些应用。

并查集看起来是一种非线性的数据结构（集合），但它可以用半线性结构很好地表示。我们可以让并查集 \mathcal{U} 以森林的形式存储，其中每个 S_j 是一棵树，用这棵树的树根标识它。

1. 合并两个集合 S_i 和 S_j 就是合并两棵树，只需要将 S_j 作为 S_i 树根的一棵新子树插入即可。
2. 查找一个元素所在的集合，只需要沿着亲代指针到底，找到树根即可。

从上面可以发现的一点是，并查集的操作从来都不需要“自上而下”（访问子代），只有“自下而上”（访问亲代）。这也就意味着，并不需要采用《树》章中的“亲代 + 子代表示法”节点来存储并查集，而只需要采用“亲代表示法”，存储每个节点的亲代即可。

具体而言，元素数量 $\sum |S_j| = n$ 的并查集，可以用一个规模为 n 的向量 V 表示，向量中保存每个元素在森林中的亲代的秩。树根因为没有亲代，所以规定树根 r 在向量中的对应元素 $V[r] = r$ 。发现 $V[r] = r$ 之后就可以确定是树根，从而停止继续向亲代查找。严格来说，这个向量里只存储了元素和集合的所属关系，而没有存储元素本身；但讨论并查集的时候，通常并不包括存储元素本身的数据结构（通常是一个向量）。

上述思想实现的并查集如下所示。

```

1 // union
2 void operator()(Rank x, Rank y, Vector<Rank>& parent) override
3 {
4     Rank xRoot { this->find(x, parent) };
5     Rank yRoot { this->find(y, parent) };
6     if (xRoot == yRoot) return;
7     parent[xRoot] = yRoot;
8 }
// find

```

```

9 Rank operator()(Rank x, Vector<Rank>& parent) override {
10    while (x != parent[x]) {
11        x = parent[x];
12    }
13    return x;
14 }
```

该算法中，查找的时间复杂度为 $\Theta(d)$ ，其中 d 是被查找元素的深度。合并的时间复杂度则为 $\Theta(d_x + d_y)$ 。

10.3.2 按规模合并和按秩合并 *

上一小节实现的并查集，其最坏时间复杂度非常糟糕。如果执行 `union(0,1)`、`union(1,2)`、`union(2,3)`，以此类推，那么最后所有的元素会退化成一条链，在这条链上进行查找会产生高达平均 $\Theta(n)$ 的复杂度。

您可能会认为，退化成一条链的原因是合并时每次都将 x 挂载在 y 下面，破坏了对称性。如果改为随机地将 x 挂载在 y 下面，或者将 y 挂载在 x 下面，是否就能避免不平衡了呢？显然仍然是不成立的。在上面的例子中如果采用随机策略，只是从每次合并都让链长加 1，变成平均每两次合并让链长加 1 而已，将常数减少到了一半，但并不能降低复杂度。

要降低复杂度，就需要按照一定的规则进行合并。类似于上一节讨论的编码树的情形，我们可以从节点置换的角度进行思考。如图所示， x 是一个很大的集合， y 是一个很小的集合，那么将 x 挂载在 y 下，会增加更多的元素的深度，从而增加查找的平均时间。此时，如果将 x 和 y 置换，就有助于降低时间消耗。从这个角度可以得到一种优化的合并策略：**按规模合并**（union by size）。

```

1 template <typename Find>
2 class DisjointSetUnionBySize : public AbstractDisjointSetUnion<
3     Find> {
4     Vector<size_t> m_size;
5     void update(const Vector<Rank>& parent) {
6         if (auto cur { m_size.size() }; cur < parent.size()) {
7             m_size.resize(parent.size());
8             std::fill(std::begin(m_size) + cur, std::end(m_size),
9                     1);
10        }
11    }
12 public:
13     void operator()(Rank x, Rank y, Vector<Rank>& parent)
14         override {
15         update(parent);
16         Rank xRoot { this->find(x, parent) };
17         Rank yRoot { this->find(y, parent) };
18         if (xRoot == yRoot) return;
19         if (m_size[xRoot] < m_size[yRoot]) {
```

```

17     parent[xRoot] = yRoot;
18     m_size[yRoot] += m_size[xRoot];
19 } else {
20     parent[yRoot] = xRoot;
21     m_size[xRoot] += m_size[yRoot];
22 }
23 }
24 };

```

定理 10.4：在按规模合并的并查集中，记节点 x 的子树规模为 s_x ，高度为 h_x ，则对任意节点 x ，都有 $h_x \leq \log s_x$ 。

证明 数学归纳法。 $n = 0$ 时显然。对于 $n > 0$ ，设最后一次合并将 x 挂载在 y 下，则这次合并中有可能改变高度的只有 y 一个节点。

1. 如果 y 不改变高度，则 $h'_y = h_y \leq \log s_y < \log(s_y + s_x) = \log s'_y$
2. 如果 y 改变了高度，则 $h'_y = h_x + 1$ 。根据按规模合并， $s_x \leq s_y$ 。所以 $h'_y = h_x + 1 \leq \log s_x + 1 = \log(2s_x) \leq \log(s_y + s_x) = \log s'_y$ 。

综上由数学归纳法得命题成立。 ■

上述定理说明，按规模合并中查找的最坏时间复杂度为 $\Theta(\log n)$ 。

按规模合并的一个变体是**按秩合并** (union by rank)。这里的秩和《向量》章使用的秩意义不同，而是定义为 $r_x = h_x$ 。当 $r_x < r_y$ 时，将 x 挂载在 y 下不会增加任何元素的高度，反之亦然。只有当 $r_x = r_y$ 时才会升高。您可以用相同的方法证明，之前按规模合并的定理对按秩合并仍然有效，从而按秩合并也可以做到 $\Theta(\log n)$ 的最坏时间复杂度。

```

1 // union by rank
2 void operator()(Rank x, Rank y, Vector<Rank>& parent) override
3 {
4     update(parent);
5     Rank xRoot { this->find(x, parent) };
6     Rank yRoot { this->find(y, parent) };
7     if (xRoot == yRoot) return;
8     if (m_rank[xRoot] < m_rank[yRoot]) {
9         parent[xRoot] = yRoot;
10    } else if (m_rank[xRoot] > m_rank[yRoot]) {
11        parent[yRoot] = xRoot;
12    } else {
13        parent[yRoot] = xRoot;
14        m_rank[xRoot]++;
15    }

```

在实际的并查集实现中，往往采用按秩合并而不是按规模合并。这出于两个方面的原因：

1. 秩的数值比规模小。所以空间效率更高。

2. 秩的更新频率比规模低，只有在 $r_x = r_y$ 时才会更新，而规模任意合并都会更新。所以时间效率更高。

您可能会好奇，为什么 $r_x = h_x$ ，却要多此一举做成按秩合并，而非按高度合并？这是因为，按秩合并有可能和其他并查集优化策略联合起来，这时 r_x 就有可能偏离 h_x 。下一小节将介绍一些其他的优化策略。

10.3.3 路径压缩、分裂和折半 *

在前面实现的朴素并查集中，如果需要反复查找 x ，那么就需要反复地遍历 x 到树根的路径，产生大量的重复工作。因为并查集不需要支持分离，所以可以利用记忆化的思想，直接将 x 挂载到树根下。之后这棵树如果没有被合并挂载到其他树下，查找 x 就只需要 $O(1)$ 的时间了。同样地，从 x 到树根的路径上经过的所有节点，都可以直接挂载到树根下。这一优化并查集的有效技术称为**路径压缩**（path compression）。

```

1 // path compression
2 Rank operator()(Rank x, Vector<Rank>& parent) override {
3     auto root { DisjointSetFindBasic::operator()(x, parent) };
4     while (x != root) {
5         auto parentX { parent[x] };
6         parent[x] = root;
7         x = parentX;
8     }
9     return root;
10 }
```

路径压缩虽然可以得到最高的压缩效率，但是需要就本次查找而言，需要先查找树根，再将沿途的节点挂载在树根下，自下而上地走两遍，大致需要 $2d$ 的时间。为了避免第二次遍历的过程，如图10.6所示，有两种路径压缩的变体^[29]。

路径分裂 (path splitting)。路径分裂将原先从 x 到树根 r 的、长度为 d 的路径，等分为 2 条挂载在树根 r 上的、长度为 $\frac{d}{2}$ 的路径。

```

1 // path splitting
2 Rank operator()(Rank x, Vector<Rank>& parent) override {
3     while (parent[x] != parent[parent[x]]) {
4         auto parentX { parent[x] };
5         parent[x] = parent[parent[x]];
6         x = parentX;
7     }
8     return parent[x];
9 }
```

路径折半 (path halving)。路径折半将 x 和 x 的亲代，都挂载在 x 亲代的亲代下，从而让 x 到 r 这条路径从长度为 d 的单链，变为长度为 $\frac{d}{2}$ 的链挂载着另外 $\frac{d}{2}$

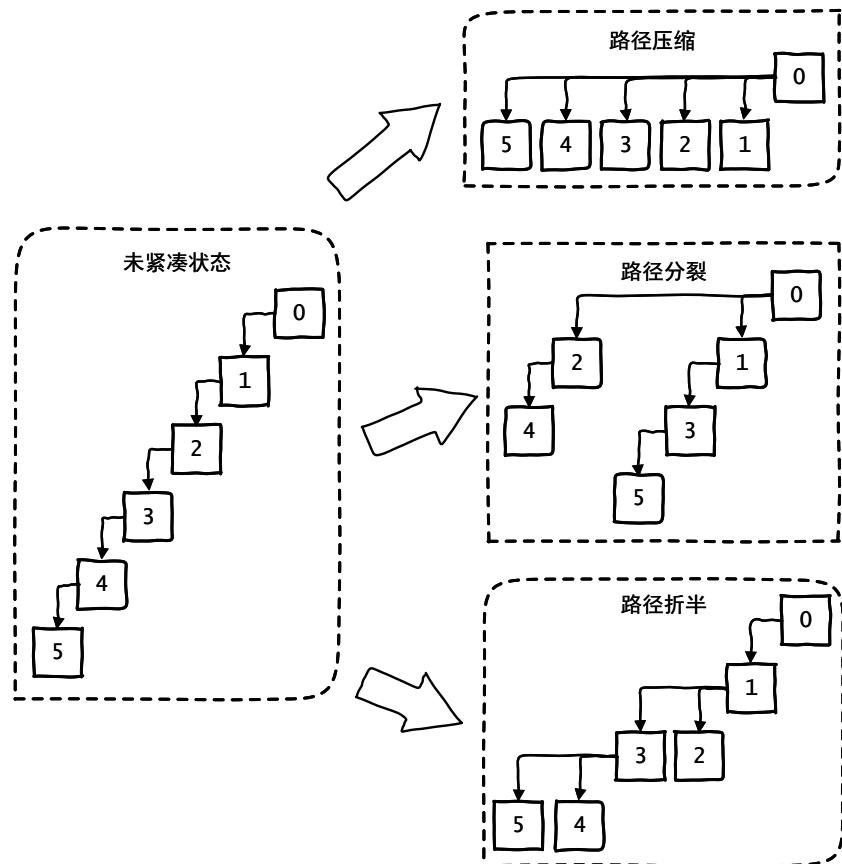


图 10.6 路径压缩、分裂和折半

个节点。该方法十分类似于伸展树的双层压缩。不幸的是，从图10.6中您能够看出来，路径折半对于平均深度降低的效果是最差的。事实上，其分摊时间常数大约是路径压缩和路径分裂的 2 倍^[30]。双层压缩的做法在伸展树上比较有效，得益于伸展树是二叉树。而并查集中的树没有度最大为 2 的限制，可以采用更加强有力的方法。

```

1 // path halving
2 Rank operator()(Rank x, Vector<Rank>& parent) override {
3     while (parent[x] != parent[parent[x]]) {
4         x = parent[x] = parent[parent[x]];
5     }
6     return parent[x];
7 }
```

路径压缩、分裂或折半不会影响树的规模，但它作用在一棵树中最长的路径上，就会使得树的高度降低。因此，当结合上述技术时，树的秩不再能等于树的高度，而是树的高度的一个上界。

类似伸展树，借助势能分析法可以证明路径压缩、分裂或折半单独使用时的最坏分摊时间复杂度为 $\Theta(\log_{1+\frac{m}{n}} n)$ （其中 m 为操作序列长度， $m \geq n$ ），结合按规模

合并或按秩合并之后的最坏分摊时间复杂度为 $\Theta(\alpha(m, n))$ (反 Ackermann 函数, 增长速率极慢, 可以认为是常数)^[30]。该证明较为复杂, 故不做展开。

10.4 本章小结

本章介绍了表达式树、编码树和并查集三种树的衍生数据结构。其中, 表达式树和编码树尽管通常以二叉树的形式出现, 但其同样可以被扩展到多叉树上。本章的内容以复习为主。

1. 理解置换节点的思想, 可以用于证明树上特定拓扑结构所具有的性质。
2. 复习栈、表达式、森林的关系, 理解表达式和树的转换方法, 明白表达式树的本质是森林。
3. 复习三种表达式之间的相互转换及重建二叉树的方法, 掌握用表达式树作为表达式之间转换媒介的方法。
4. 复习优先队列, 掌握在纸上构建二叉或多叉的 Huffman 树的方法。
5. 复习信息熵的观点, 理解编码的意义, 以及分析最优 PFC 编码树时间复杂度的方法。
6. 复习伸展树双层伸展的思想, 理解并查集路径压缩、分裂和折半在降低分摊时间复杂度上的意义。

参考文献

- [1] Patterson D A, Hennessy J L. Computer organization and design arm edition: the hardware software interface[M]. Morgan kaufmann, 2016.
- [2] 陈渝, 等. rCore-Tutorial v3.5.0[EB/OL]. 2021. <https://github.com/rcore-os/rCore-Tutorial-v3/releases/tag/v3.5.0>.
- [3] 陈渝, 等. rCore-Tutorial-Book 第三版[EB/OL]. 2022. <http://rcore-os.cn/rCore-Tutorial-Book-v3/index.html>.
- [4] Gregoire M. Professional c++[M]. John Wiley & Sons, 2021.
- [5] 严蔚敏, 吴伟民. 数据结构: C 语言版[M]. 清华大学出版社, 1997.
- [6] 邓俊辉. 数据结构: C++ 语言版[M]. 清华大学出版社, 2013.
- [7] Knuth D E. The art of computer programming, vol 1: Fundamental[J]. Algorithms. Reading, MA: Addison-Wesley, 1968.
- [8] Reingold E M, Tilford J S. Tidier drawings of trees[J]. IEEE Transactions on software Engineering, 1981(2): 223-228.
- [9] Blackburn P, Meyer-Viol W. Linguistics, logic and finite trees[J]. Logic Journal of the IGPL, 1994, 2(1): 3-29.
- [10] 王红梅, 胡明, 王涛. 数据结构 (C++ 版)[M]. 清华大学出版社, 2005.
- [11] 彭波. 数据结构[M]. 清华大学出版社, 2002.
- [12] 鞠文飞. 多快好省, 改造 SOHO 路由器[J]. 网管员世界, 2007(10): 108-110.
- [13] Kowalski R. Algorithm= logic+ control[J]. Communications of the ACM, 1979, 22(7): 424-436.
- [14] Knuth D E. The art of computer programming: volume 3[M]. Pearson Education, 1997.
- [15] Bentley J L, Haken D, Saxe J B. A general method for solving divide-and-conquer recurrences [J]. ACM SIGACT News, 1980, 12(3): 36-44.
- [16] van Rossum G. Tail recursion elimination[J]. Neopythonic. blogspot. be. Retrieved, 2012, 3.
- [17] 全国科学技术名词审定委员会审定. 计算机科学技术名词 (第三版) [M]. 科学出版社, 2018.
- [18] Cormen T H, Leiserson C E, Rivest R L, et al. Introduction to algorithms[M]. MIT press, 2022.
- [19] Albert M, Bousquet-Mélou M. Permutations sortable by two stacks in parallel and quarter plane walks[J]. European Journal of Combinatorics, 2015, 43(C): 131–164.
- [20] Elvey Price A, Guttmann A J. Permutations sortable by deques and by two stacks in parallel[J]. European Journal of Combinatorics, 2017, 59: 71-95.
- [21] Elvey Price A, Guttmann A J. Permutations sortable by two stacks in series[J]. Advances in Applied Mathematics, 2017, 83: 81-96.
- [22] Bryant R E. Computer systems: A programmer's perspective, 3rd edition[M]. Pearson, 2015.

- [23] Huss-Lederman S, Jacobson E M, Tsao A, et al. Implementation of strassen's algorithm for matrix multiplication[C]//Proceedings of the 1996 ACM/IEEE Conference on Supercomputing. 1996: 32-es.
- [24] Huang J, Smith T M, Henry G M, et al. Strassen's algorithm reloaded[C]//SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2016: 690-701.
- [25] Flajolet P, Odlyzko A. The average height of binary trees and other simple trees[J]. Journal of Computer and System Sciences, 1982, 25(2): 171-213.
- [26] de Bruijn N G, Knuth D E, Rice S. The average height of planted plane trees[M]//Graph theory and computing. Elsevier, 1972: 15-22.
- [27] Zhang C, Chen Z, Song H, et al. Pipo: Efficient programmable scheduling for time sensitive networking[C]//IEEE International Conference on Network Protocols (ICNP). 2021.
- [28] Shannon C E. Prediction and entropy of printed english[J]. Bell system technical journal, 1951, 30(1): 50-64.
- [29] van Leeuwen J, van der Weide R, et al. Alternative path compression techniques[M]. Unknown Publisher, 1977.
- [30] Tarjan R E, Van Leeuwen J. Worst-case analysis of set union algorithms[J]. Journal of the ACM (JACM), 1984, 31(2): 245-281.

附录 A C++ 的安装和配置

在这篇附录里介绍 Windows 操作系统下 C++ 的安装和配置方法。采用 MacOS 或 Linux 操作系统的读者应当有能力自己完成配置。

A.1 使用 MSVC 编译 C++

Microsoft Visual C++ (MSVC) 是笔者在编写此书时使用的编译器，也是本书推荐的编译器；本书采用的模块接口文件后缀名 (.ixx) 就是 MSVC 的标准。得益于强大的 IDE Microsoft Visual Studio，您可以很方便地运行 C++20 编写的程序。

1. 在搜索引擎上输入 Visual Studio，请注意忽略可能在最前方展示位的广告。或者直接访问 Microsoft Visual Studio 中文官方网站。
2. 选择下载 Visual Studio，选择 Community 2022（免费的社区版）。如果您是在校学生并且学校购买了专业版，也可以从学校的下载站中获取。请注意不要下载旧版本的 Visual Studio，因为可能不支持 C++20。
3. 安装 Visual Studio，点选“使用 C++ 的桌面开发”，其他可以不选。
4. 创建项目之后，选择项目 → 属性。在“常规”中将“C++ 语言标准”改为 ISO C++20 标准。另外，在 C/C++ → 优化中，将“优化”改为“最大优化（优选速度）(/O2)”。

A.2 使用 GCC 编译 C++

在 Windows 操作系统中使用 GCC，通常需要使用 MinGW。目前 MinGW 上似乎还不支持 format 库，需要对相应的部分进行修改。目前 VS Code 的 C/C++ 插件似乎在 C++20 项目构建会有一些问题，建议采用命令行构建。

1. MinGW 的下载方式有很多，您可以在搜索引擎中找到。笔者使用的 MinGW 来自 WinLibs 下载站。
2. 选择最新版本的 MinGW，请注意需要选择 64 位的版本。下载之后解压，需要自行配置环境路径，将 g++.exe 的所在路径配置到 includePath 中。
3. 在控制台（终端）使用 `g++ -fmodules-ts -std=c++20 -x c++ -c` 文件名.ixx 来编译模块接口文件 (.ixx)，它会生成一个模块缓存文件 (.gcm) 和一个输出文件 (.o)；使用同样的命令编译源文件 (.cpp) 生成输出文件 (.o)。
4. 最后使用 `g++ a.o b.o main.o -o main.exe` 这样的命令进行连接。

附录 B 清华大学计算机考研指南

本文更新于 2023.3.23。由于 2023 年录取并未完全结束，本文结论还存在一些不确定性，有待录取名单正式公布后做进一步修订。

考研是一项信息博弈，如果在选择考研目标的时候了解了更多的信息，则可以在这项信息博弈中取得先发优势。笔者在清华大学计算机系就读本科及硕士研究生，在自己的考研经历中总结了一些经验，并通过其他考研的同学了解到一些更多的信息。我将这些信息整理总结为本书的一个附录。若希望提供其他学校或者补充的信息，或发现我的理解有错误，或这个文件存在误导性内容，非常希望您通过 GitHub issue 或联系笔者进行告知，提高本指南的质量。非常感谢您的帮助。

B.1 基本信息和免责声明

由于笔者的本科和硕士研究生均就读于清华大学计算机科学与技术系（下简称本部），所以本文主要讨论本部的情况。除此之外，清华大学在同一考试科目（11912）下还招收网络空间与网络安全研究院（网研）和深圳国际研究生院计算机技术专业（深研），以及考数据结构（962）的清深大数据、考模式识别（807）的清深人工智能等相关专业。

1. 我从未上传或公开过任何清华本科的练习及考试题、上课录像、PPT、教材、初试真题、机试真题、面试中被问到的题目。
2. 我本人从未阅读过 912 和机试的往年真题，所以凡是涉及往年真题的内容均不是第一手信息。
3. 我从未报过任何考研机构的辅导班、从未在任何考研机构里工作或投资，所以凡是涉及考研机构的内容亦不是第一手信息。
4. 考研有风险，报考需谨慎。感谢您阅读本文，也感谢您愿意关注笔者设计的 DS Lab，但我不对读者的报考选择和复习安排负责。

B.2 初试介绍

B.2.1 初试考试情况

B.2.1.1 初试概述

考研初试总分为 500 分，包括政治（100）、英语一（100）、数学一（150）、计算机自命题（150）。清华计算机系的自命题科目为 912，包括数据结构（70）、计

算机组原理 (30)、操作系统 (30) 和计算机网络 (20) 四个学科。

在选择报考清华计算机系之后，可以选取三个方向：系统结构、软件、应用。和北京大学不同，这三个方向共享排名和复试分数线，并且允许学生在复试时重新选择自己的研究方向。因此，在初试报名阶段，可以随意地选择三个方向之一。尽管如此，并不建议您在参加考试之前对自己未来的研究方向毫无打算。

具体的历史分数和过线情况可参照网上的统计数据。2023 年本部的分数线为 353，网研为 330，深研为 325。整体而言比同一水平的北京大学分数线要低一些，但也并不是一个很低的分数。跨考的学生如果大三才开始准备，必须慎之又慎，如果不付出充分的努力，是很难获得复试资格的。

以历史情况看，本部稳妥过线的分数大约为 380 分，但单科分数线有时会划得比较随意。其中政治和英语的单科分数线可能会划到 60 分甚至 65 分，数学和专业课则有可能划到 100 分。前车之覆，后车之鉴，在复习时需要尤其注意避免偏科。

B.2.1.2 初试阅卷情况

传统上北京地区被认为是旱区。北京地区的压分现象主要体现在政治主观题、英语主观题上，此外数学的步骤分也会扣得比较狠。这一压分现象造成的（和其他省市相比的）分数下降幅度，可能会由于试卷难度有所不同。考虑到压分现象，上述的政治、英语 60 乃至 65 分可能会比预想中更加困难，请务必正视这两门公共课。

考研的阅卷相对来说比较随意。和高考不同，考研的阅卷标准相对灵活，尤其是英语作文等项目有巨大的不确定性；同一篇作文在不同阅卷者手中可能会有十几分的来去。因此在估计初试总分时，应当尽量为自己留出一些余裕，比如以 390 分为目标。请在填报目标高校之前对自己的分数进行合理估计，评估报考清华计算机系的风险。**我不建议您对自己的初试成绩有过高的预期。**从历史情况看，以初试第一名或极具挑战性的分数（如 420）作为自己的目标，将对您的备考计划和备考心态造成巨大的压力。

B.2.1.3 清华自命题科目情况

清华自命题科目（912）难度和计算机统考（408）**相仿**，但命题风格有很大差异，比较接近清华计算机系本科的考试风格。另一方面，912 没有显式的考纲，通常也认为考试范围和本科生期末考试的范围相仿。不过，从近几年的情况看，一些要求较高的题目，比如大段的操作系统源代码分析等，已经很少出现在 912 的命题中，所以考试时间往往会有极大的富余。

总体而言，912 的命题风格以**理解**为主，不要求学生有较大的知识记忆量；但

如果理解的深度不够，很容易感到无从下手。不过这一点可以通过熟悉往年真题进行弥补，并且由于 912 的命题规律性比 408 更强，所以做往年题可以自己总结出一些真题的命题角度。

B.2.1.4 初试被录取的影响

清华计算机系近几年的录取人数约为 13 人，复录比大约为 2: 1。在总成绩中，初试和复试各占 50%。网研和清深的初试占比也为 50%。

目前为止，初试第一名一定会被录取，但考虑到初试分数高的同学通常各方面能力都比较强，所以不确定这一经验规律是否在未来被打破。同时，因为初试第一名具有极大不确定性，所以任何情况下都不应该完全忽视复试而专注于追求取得初试第一名的成绩。

在 2022 年及之前，初试成绩和总分的相关性比较弱。通常可以认为初试过线之后，初试成绩对是否录取几乎不再有影响，所有同学被拉回同一起跑线。即初试是一张门票。（初试第一名可能除外）

2023 年，除一名 394 的同学未能被录取之外，其他同学的录取与否均按照初试成绩高低确定。目前尚不清楚是否意味着在未来初复试实际占比发生了转向，还是由于本年度初试高分同学在复试阶段做足了准备，引起的偶然现象。

不过，初试成绩较高的同学，即使没有被计算机系录取，通常也可以被校内其他院系录取，如网研院、深研院等。即使没有到达初试分数线，仍然有可能被调剂录取。

B.2.2 初试竞争情况

B.2.2.1 报考规模

总体而言，清华计算机系的报考规模相对较小，初试竞争烈度相对较小，有竞争力的报考人数甚至可能不超过 100。如下是一些可能的原因。

1. 912 本身有难度，科目比较多，对跨考不友好。
2. 目前，自命题学校无法调剂到统考 408 的学校。由于现在大多数“双一流”学校的计算机系都改考 408，选择 912 意味着基本只能在校内调剂，风险较高。
3. 同样考 912 的清华网研院、深研院等院系，现在呈现出越来越多的保护第一志愿的倾向。而总体看来网研院、深研院的分数线比计算机系本部要低，所以会吸引一些同学第一志愿报考网研院、深研院。

B.2.2.2 复习成本

因为难度和 408 相近，所以在完全自学的条件下，11912 的整体复习成本会和 11408 相近。然而，由于辅导机构会倾向于深耕统考科目，所以在报班的条件下，11912 复习成本会显著高于 11408。具体需要复习时间和个人情况有关，我通常建议至少提前 6 个月开始初试复习，以防最后因时间不足而焦虑，引发心态危机。

考虑到基础越扎实的同学对辅导机构的依赖度越低，所以对于名校科班的同学来说，清华计算机系的初试竞争压力是低于北大对应院系的。从分数线上看，相对于北大叉院来说，清华计算机系这边的分数线会低很多（大约 20 分）。

B.2.2.3 本系保护问题

由于多种外部环境因素，现在有越来越多的清华本校学生，以及名校科班学生报考清华计算机系。有一种广为流传的怀疑认为：如果本系的同学总分较低，会通过单科线卡掉几个高分让他们有机会进复试。这种“本系保护”无法证实但也无法证伪，所以请您重视不要让单科成绩掉队。

B.2.3 初试复习用书

B.2.3.1 数据结构之前

数据结构前，可以看徐明星《程序设计基础》入门。《程序设计基础》在清华是《数据结构》的先修课，这门课可以看做是 C++ 语言学习加上一小部分的编程能力训练。如果零基础的同学打算考清华计算机，可以先从这本书开始打基础。

零基础的同学在学习《数据结构》前，应当能使用 C 语言进行编程。清华大学的《数据结构》教材是用 C++98 写的，作为旧标准的 C++，它并没有在 C 语言的基础上增加太多东西，可以认为是 C with class。当然如果学习了 C++ 会更好，比如，如果您学习了现代 C++，就可以很轻松地使用笔者的 DS Lab 进行实验。不过对于跨考学生来说，C++ 作为一门语言特性非常多的语言，学习成本较高，不建议在编程语言上消耗太多的时间和精力，尤其是不应该在初试复习的时候。

由于《数据结构》中会涉及一些相对底层的内容（内存管理），所以使用 C/C++ 进行学习是建议的。使用 Rust 固然也是可以的，但是用 Rust 实现数据结构并不是一个简单的事情。

B.2.3.2 数据结构

数据结构必须看邓俊辉《数据结构》。邓俊辉《数据结构》和其他经典的数据结构教材相比，无论是授课风格还是知识内容上都有显著差异，且 912 中《数据结

构》占比远高于 408，所以邓俊辉《数据结构》为这门课复习的必修课。请注意邓俊辉《数据结构》的教材出版时间较早，目前的 PPT 和网课已经和教材有一些不一致。尽管，目前为止 912 的考试范围仍然限定在教材上，但仍然建议大家使用新版本的课件进行学习。不建议看其他版本的《数据结构》。对于备考来说，邓俊辉《数据结构》可以覆盖所有的知识点，且现版本的 PPT、网课非常新。其他版本，比如严蔚敏的《数据结构》也是优秀的教材，但对备考清华计算机系的帮助很小。

不建议看《算法导论》。《算法导论》的叙述风格太过数学了，对于计算机系的学生来说不那么必要。整个清华计算机系作为工科院系，都是以工程理解为主的。类似于《计算机程序设计艺术》(TAOCP) 之类的更加艰深的书则更加不推荐。在学生刚开始复习考研的时候，总会以为时间还有很多，如果使用高难度书籍进行学习，很容易不小心地就陷入到过于复杂的问题里，最后发现时间不够用。

《面向对象程序设计》和《离散数学》在清华的课程安排上是《数据结构》的先修课，但从考试来看并不需要学这些，性价比不够高。其中和《数据结构》关系最密切的是《离散数学》中的图论部分，如果时间多想了解的话看一下那一章就可以。

在学习《数据结构》的过程中，对于不懂的地方，非常建议自己写程序验证。这一方面锻炼了自己的编程能力，一方面深化了自己对数据结构知识的认识。但我不建议完整复现《数据结构》上的所有代码：它是 C++98 的代码现在已经落后于时代了；编程风格不够美观（节约排版原因）；工作量太大。笔者的 DS Lab 可以为您提供一个新选项。

B.2.3.3 数据结构外的三门

除数据结构以外的三门，都可以不看教材，依托清华 PPT 做复习。这三门课的教材用的都是比较厚的英文版（当然也可以买到翻译版）教材，清华本科的学生也不会全部精读。实际上期中期末考试、以及考研的命题点基本都在 PPT 上。教材分别是特南鲍姆《计算机网络》（龙书）、《软件硬件接口》、《操作系统精髓与设计》，可以买翻译版当工具书用，时间多的话可精读。

一些市面上的其他经典教材，比如袁春风《计算机组成与系统结构》《计算机系统基础》、《深入了解计算机系统》、《自顶向下方法》等，也可以作为参考，但由于不同教材之间内容相互重叠，性价比会边际效应递减。如果从备考初试出发，基于 PPT 复习已经足够，购买教材已经绰绰有余，不需要再买其他经典书了。

对于操作系统，ucore 实验指导书建议阅读，并建议完成 ucore 的实验。ucore 的实验设计水平在不断提高，并且没有学习 Rust 的门槛，只需要有 C 语言基础即可。尽管近三年都没有在 912 考 ucore 的代码题，但不排除这种可能性。另一方面，

ucore 的实验也能让同学们对操作系统的理解加深一个层次。由于不一定会考，所以时间不够可以放弃 ucore。

B.2.3.4 练习册

除非您足够自信，否则做题是有必要的。

1. 邓俊辉《数据结构》自带一本习题册，这本习题册类似于拓展阅读读本。题目价值比较高，但题目风格和考试不太一致。如果有时间，可以作为思考题练习。
2. 912 的真题是第一手复习资料，但通常缺少答案。
3. 清华本科的期中期末考题也是不错的复习资料，但同样经常缺少答案。
4. 可以使用 408 的辅导用书作为练习册，比如王道、竟成或其他机构的书。当然 408 和 912 的知识点可能会冲突。当知识点解释冲突时，请按照**标准 >PPT > 教材 > 练习册**的规则处理。比如在《计算机网络》中，教材更新可能慢于 PPT，PPT 更新又可能慢于 RFC，而考研机构编的练习册不但更新可能非常慢、编纂过程中也可能收入错题。
5. 不建议去做难度特别高的题，尤其是 TAOCP 之类的书上的题。虽然有些题目可能很有趣（甚至结果可以发表），但总体上性价比非常低，远远超出了考研的难度。

B.2.4 初试考研辅导

B.2.4.1 学堂在线

1. 邓俊辉《数据结构》课程建议看完。邓俊辉的《数据结构》在清华计算机系课程中算是首屈一指的教学标杆，我的四年本科生活没有几门课的教学水平能达到这个层次，非常推荐。
2. 向勇、陈渝《操作系统》课程也建议看完。这门课质量也很高，并且清华的操作系统教学比较有特色。
3. 据我所知，目前另外两门清华没有公开的网课。没有基础的跨考同学可以选择其他学校的网课进行学习。科班的同学不太建议再看网课，视频相对文字而言，信息输入效率比较低。

B.2.4.2 自学材料

1. 考研资料的分享网站：**912-project**。您可以从好心师兄建立的清华大学计算机系考研攻略中找到很多信息。遗憾的是，近几年的资料基本都被机构私藏

了，没有公开在这个网站上。

2. 最好的自学网站：**CS-DIY**。**CS** 自学指南这个网站对跨考和本科没有学扎实的科班同学都有非常巨大的帮助，归纳总结了很多经典的公开课程。
3. 语雀笔记。很多师兄师姐留下了出色的笔记，也可以进行参考。比如2021 清华大学计算机 912 考研笔记本，和2022 清华 912 备考笔记。
4. Bilibili 上的视频和知乎上的经验贴等。比如，T 神的数据结构讲解视频备受好评。

自学的时候不要贪多，重点复习初试会涉及到的部分，以及打算在复试材料中用到的部分就可以。如果时间比较多，可以把更多的时间放在自学上，提高自己的自学能力对未来的科研工作很重要。现在供给大家自学的信息来源很丰富，足够让跨考的同学通过自学达到清华计算机本科的平均水平（也就是我这个水平）。如果因为本科（科班）或工作（计算机相关）的压力大，导致考研复习时间不足的情况，可以报班而不是自学。因为这个时候考研复习节约的时间用来做了对考研上岸来说性价比更高的事情。

B.2.4.3 定向班

在报班的时候，如果对方没什么名气，请首先确认自己**不是面对电信诈骗**。近年来，经常有学生在自己上岸以后靠“考研辅导”创收，而实际上并未真正提供合格的辅导服务的情况，一定要三思而后行。

在选择报班的时候，老师的能力非常重要。考研机构基本上请的老师都是往届的师兄师姐，很多都不是全职从事考研辅导工作，他们的能力和责任感都是参差不齐的。请务必确认老师本人对知识的理解足够深刻、并且足够负责。很多时候，宣传里的老师和实际和学生对接的老师并不是同一个，需要小心。

相对于自学而言，培训班的主要优势是答疑。这是因为 912 的报名人数太少，课程制作成本和收益不成比例，所以 912 定向的课程和非定向的 408 课程相比，一般会粗糙很多。因为考研机构的老师基本也都是师兄师姐，且主要服务是答疑，所以您务必要大胆提问、大胆质疑、大胆以超越他们为目标。尊师重道、为尊者讳的做法在考研里用通常不会有多少正面效果。

王道目前是最大的计算机考研机构，王道的定向班相对比较可靠；集中在王道这边的同届、往届同学也比较多。不过再次声明，必须要确认老师的情况。

B.2.4.4 非定向班

408 有非常成熟的辅导培训体系，而且可以达到的上限，也远高于目前已有的 912 培训；所以可以考虑报 408 培训班，再通过自学差额知识点转成 912。从理论

上讲，给知识体系打一个扎实基础，比初试 912 拿下高分更有意义，未来在研究生阶段更容易触类旁通。不过，单独报名 408 培训班来考 912 的风险很大。之前很多同学按照 408 复习 912 结果成绩很低。

所以，如果选择非定向班，请务必通过多自学、多交流降低信息不对称。

B.2.4.5 专门面向 912 的机构

新威考研有一个专门面向 912 的培训。主讲是张威老师（威神），已经办了好多年，有不少往届的上岸同学推荐，质量相对有保证。目前是名气最大的 912 培训机构。威神本人当时以 316 分上岸，并非名校科班出身（哈工大软件），对四门学科的知识体系理解可能比较有局限。但威神深耕 912 多年，就应试而言应该有平均以上的水平。

据我所知，新威考研没有第二个可以担纲的主讲，威神本人的时间、精力和身体状况是一个问题。并且，新威考研价格极低，在 912 整体报考规模非常小的情况下，每年的流水肉眼可见的捉襟见肘。在不转型的情况下，未来资金链可能会成为问题。所以，报名新威考研的风险会略高于王道等大型机构。

在 2023 年本部录取的学生名单中，报名新威考研的学生占一半以上（信息来源：新威考研 B 站官方号）；而未报名的被录取的学生以科班出身、尤其是清华科班出身的学生为主，他们本身对 912 培训机构的需求是比较弱的。从我了解到的情况看，新威考研最重要的作用是帮助对 912 课程体系不了解的跨考学生掌握这些课程的重点，在这方面张威老师大概还是颇有心得的。

新威考研有个押题卷，评价也不错。押题卷主要是评估自己的知识掌握程度，不是真的为了押中题，需要正确看待押题卷的作用。912 命题比较随意，往年题有很多能和新题相似。如果自己做了 912 的往年题，稍微思考一下，肯花点时间设计，每个人都能出押题卷，不过考研的时候，一般也没有这个时间。

由于了解不多，笔者这里先不讨论其他的机构情况。欢迎有了解的同学联系笔者添加。

B.2.4.6 考研交流群

考研交流群，尤其是人数较多的大群，经常会讨论无关的问题。经常水群不如不加群，请确保您有良好的自制力。在无关问题上参与考研群的讨论，这些时间不如用来和自己的家人、朋友、恋人加深感情。此外，大群里还有被其他同学的进度干扰到心态的影响。一部分相互熟悉的研友组成的小群，互相都能信得过的话，通常氛围会好一些。辅导机构的班级经常也会组织小群，不过并不一定会负责任地进行管理。

群里问问题的时候，请提供足够多的信息，讲清楚自己有困惑的地方在哪里。群里的同学们互相回答问题，是出于兴趣做的，没有包讲包会的责任，和培训班的老师不一样。

B.3 复试介绍

B.3.1 复试考试情况

B.3.1.1 复试概述

清华计算机系复试的满分为 500 分，其中机试 100 分、综合面试 80 分、专业面试 320 分。20 至 22 年由于疫情影响没有笔试，复试在线上进行。23 年延续了线上机试，同时本部采用线上面试，但深研院为线下面试。

机试通常包括 3 道题，每道题 100 分，满分 300 分；在计算总分时折算为 100 分。根据公开的文件，综合面试除了英语之外，还可能会关心政治倾向、兴趣爱好、人际关系等各种问题。专业面试则主要关注专业问题。

B.3.1.2 机试命题情况

清华的机试由清华本科算协的同学负责命题，具有命题随意、难度随机、风格多变的特征。机试题没有稳定的命题人，难度非常随机，有可能大家都做对，也有可能几乎全都 0 分；命题没有范围，各种领域的问题都可能会出，具体可以参考往年。往年的题目可以在网上搜到。根据 OIer 的反馈，目前的机试按照提高省选的难度去准备是比较合适的。此外，存在多套机试备用卷。后续调剂可能需要考第二场机试。

B.3.1.3 面试流程情况

面试需要提前准备一些材料，包括简历、个人陈述和面试现场的 PPT。这些是同学们可以自己控制的内容，通过灵活地组织展示材料，可以更好地展现自己的真实实力。综合面试的英语环节会考察口语（形式比较多样，不一定是自我介绍），如果毫无基础则需要预先准备。可以和初试的英语一同准备。

专业面试的老师，专业素养很强，除非自己已经做到了本领域的边界，否则很难糊弄过关。不过清华的面试氛围通常比较轻松，不需要过度紧张。

B.3.1.4 复试对录取的影响

在 2023 年以前，在清华计算机系（不包括调剂）的复试，初试过线基本上回到同一起跑线。根据同学的反馈，实际招生的老师可能并不了解或不在意初试的

成绩，甚至可能不知道初试的满分是多少。复试的成绩可能会相差很远（100 分甚至更多），从而抵消掉初试的差异。尽管 2023 年的录取结果和初试成绩高度相关，但对于考研的学生而言，一年只有一次机会，仍然应当为复试作出充分的准备。

除了客观的机试成绩以外，复试成绩可能会受到各种因素的影响。对比不同届甚至同一届的复试成绩，都是一件意义不大的事情。初试到复试之间大约有 3 个月的时间，对于跨考的同学来说可能不足以展示一个漂亮的材料，可能需要**提前准备**：但一切要以保证初试能过线为前提，切忌好高骛远。

B.3.2 复试竞争情况

进入复试之后，只有二十多个同学参与竞争，此时竞争难度的重点变成了“对手的实力”。因此，这一节站在“对手”层面上进行讨论。

B.3.2.1 本系学生

目前为止，本科在本系且初试过线的同学一定会被录取。同一届满足上述两个条件的学生，目前看来在不断增加，2023 年达到了 4 个，但相对考研同学总数来说还是比较少的。想要挤掉他们，通常只能卷起来让这些同学初试无法过线。本系的大四课程压力不小，应届过初试难度较大。而如果是非应届，通常也不愿意花一年时间在考研，通常就选择就业、出国、保研较低层次的院校、保研到其他系。所以目前为止，本系同学还非常少。

因为本系的学生一般都是当地高考状元或者集训队选手，所以复试名单公布后，可以通过搜索引擎确定数量。

B.3.2.2 其他学校的科班

目前为止，第一作者的高水平论文（CCF B 或以上，尤其是非 AI 方向的）、ACM 区域赛金牌都是极强的竞争力。这两点也可以在复试名单公布后，通过搜索引擎很容易确定。

名校科班在录取的同学中占主体部分。对于名校科班的学生来说，往往可以拥有一份非常漂亮的简历，并且因为专业知识丰富，在面试中回答专业问题占据着很大的优势。那些参与过 ACM 等算法竞赛的同学在机试中也具有优势。因此，经常可以看到名校科班的学生在复试中逆袭。

非名校的科班同学可能本科可以获得的资源有限，难度也相对较高。从近几年的录取情况看，几乎没有本科学校在 211 以下的同学被录取。科班身份仍然可以提供相对丰富的专业经验，但因为本科学校生源问题，本科的课程要求通常相对偏低，最好在本科阶段就努力向名校的标准看齐，不要局限于自己的学校课程

要求。

需要特别注意的是，不要轻信其他人对复试材料的肯定，一般除非太过朴素的，师兄师姐都会给以鼓励，更不用说考研辅导机构了。比如我就会给几乎所有同学鼓励。因为名额少，赛道和 408 隔离，所以每年的竞争情况都可能大不相同，应当在保证初试能过线的情况下尽可能丰富自己的复试武器储备。

B.3.2.3 跨考

目前来看，清华本校在计算机外的其他院系，不会在跨考中占据显著的额外优势。例如，在 2022 年有数学系和叉院的同学没有被录取。

经常有相近专业（软件、电子、通信等）跨考上岸的成功例子，即使稍远一些的其他工科也不乏其人。理科、商科等基础薄弱的跨考同学，也可以凭借自己的努力上岸。但无一例外，必须对计算机专业具有足够的了解。跨考一战上岸通常需要从大二（甚至大一）开始进行准备。初试只是一部分，更重要的是提升自己在科研阵线上的**即战力**。硕士只有三年，老师等不起跨考的同学进来之后继续学习再兑现天赋，跨考和科班是站在比较统一的起跑线上竞争的。这是一种相对公平的做法，但也要求跨考的同学付出更多的努力，在即战力上达到科班的水准。

B.3.3 复试复习资料

B.3.3.1 算法题训练

机试存在每年会变化的保底分数（第一题通常是送分的签到题，以及后面两题可能有的暴力分），达到保底分数就说明能力过关。达到保底分数通常阅读胡凡《算法笔记》这类入门书，适当地针对基本算法进行复习，加上考试的时候心态平稳就可以。达不到保底分数则会比较危险，这个时候如果有较大的项目或竞赛获奖佐证自己的编程能力（并在面试中应答自如），可以挽回一些印象分。

算法题网站（比如 ACwing）上的题可以提供很好的练习。由于算法题网站上往往涵盖了大量的、难度不一的题目，请在练习的过程中务必注意远离自己的舒适区，不要贪图题目数量，在自己能力范围内的题里花费过多时间，限制了自己的进步速度。

可以提前参加 CCF CSP 或者 PAT 等考试感受考试氛围，缓解考场上的紧张。如果考试中感到困难，一定要优先采用暴力方法拿到保底分数，再尝试进行优化。历史上有不少算法能力优秀的学生和第 2 题死磕导致错失第 3 题保底分。

B.3.3.2 充实简历的工作

如果没有机会在本科直接参与科研，或者是跨考的同学，则建议在复习时准备一些可以充实简历的工作，并为面试做准备。

首先，操作系统训练营或者一生一芯这种大型活动，每年都有，可以系统性地提升操作系统（组成原理）的能力，提升的上限很高，对于初试复习也比较有用。不过，这类型活动有一定门槛，对于跨考的同学来说难度可能略高。

翁家翌牵头做的**清华课设指南**：清华大学计算机系课程攻略，是由清华计算机系本科生整理的清华课程体系，包含了大量的清华课设。可以选择自己喜欢的方向去做，也可以在初试复习的时候，顺便做掉那几门课的课设。

传统的三件套是 CPU、OS、编译器，名校科班基本都做过，是课设中相对分量较重的项目。计网的软件路由器是可选项目。三件套做出来能极大提高同学对对应学科的理解，兼顾 912 的初试，三件套的价值是 OS > CPU > 编译器，最好能三件套都做，作为跨考同学，做个青春版也能展示出一定实力。青春版大概相当于，照着经验贴做 ucore lab，华科平台上画的、不带任何附加功能的五级流水 MIPS32，用 lex/yacc 写的没有任何优化的 C 语言编译器。跨考的同学大约需要 2-3 个月完成这些项目，想要展现出更多实力可以在 OS 或者 CPU 上加内容，做深入、有东西可讲、不容易被问倒，比做了很多“手写数字识别”级别的简单项目有用的多。虽然编译原理也很有用，但是和初试没关系，所以不那么建议在编译器上深入做。

随后是论文阅读和复现。这是最简单、直接、见效快的方法，根据自己的目标方向和导师，去选择相应的文章精读。如果有机会，复现其中的一些项目；如果有机会，找到一些可以进一步改进的点。

笔者写的 DS Lab 并不是为复试材料设计的，它的难度和在复试材料中的价值不匹配。对于跨考的同学来说，准备复试材料应当更加重点突出自己的即战力，而不是基本功。

总体来说，准备复试材料时，在精不在多。这一点跨考的同学需要做准备，很多科班的同学也要引起重视。研究生最终还是要研究一个而非很多个方向的，所以有一项精通（至少是精读）的方向很重要。

B.3.3.3 复试辅导机构

据我所知，几乎所有收费的复试辅导性价比都很低。复试群里有免费的模拟复试和信息分享，可以部分解决信息不对称的问题。复试材料的展示和组织，也是可以找师兄师姐、老师、研友等帮忙修改完善的。这个现象的主要原因还是清华计算机的特殊性：层次高、样本少、初试占比低，导致很多一般的复试技巧在应对清

华计算机复试时不够通用。收费的复试辅导可以让自己看起来比较富婆，兴许能得到同学们的另眼相看。

B.3.4 复试简历设计

特别强调，本节的复试简历设计为笔者的一家之言，不一定适用于每一个同学。简历设计对于每个同学而言，具有高度的矛盾特殊性，本文仅做参考。

B.3.4.1 简历设计纲领

在进行复试的复习之前，一定要明确复试的考察目的是什么。复试是用来筛选科研能力的，科研能力主要可以分为四个方面：**知识习得、知识产出、知识验证、知识表达**。

展现这些科研能力的媒介主要包括机试、简历、面试时的 PPT 以及临场问答；这其中，简历和 PPT 是完全由我们自己控制的，所以尤其需要打磨好。如果在这方面无法提起面试老师的兴趣，很有可能无法

复试是用来筛选科研能力的，并非考察学生的整体科研素质。通常情况下，科研能力的展示只需要围绕自己选择的一个方向上的能力。比如说，自己想要研究系统结构的话，一般就不会问到人工智能方向的问题，因为通常情况下老师也不愿意浪费时间，去问无效的问题。所以，简历就像一扇窗户。我们除了把窗外的布景做好，还需要准备好老师在考场上用望远镜细细观察下去的准备。因此简历设计，不但是简历本身的设计，还包括围绕简历的准备工作。面试 PPT 同理。

很多人看简历的时候只看简历中的标签，比如清本、ACM 金牌、X 篇论文等。但简历并不是斗兽棋，除了它的内容以外，形式也是非常重要的。仅凭内容就能上岸的，目前来看还是少数。对大多数同学来说，简历不是有一条加一条分的那种设计，它主要决定的是第一印象。简历上相同的内容，会根据准备的细致程度以及展示时的表现好坏，得到不同的最终印象，反映为不同的最终分数。

B.3.4.2 知识习得

1. 本科的课程成绩和位次可以列出。
2. 本科学得好的课程成绩可以列出。
3. 本科如果有英文学习经验（英文课程、英文教材的课程、国外联培或交换）可以列出。
4. 阅读一定数量的英文文献，可以选择自己打算研究的方向、或者精确到自己的目标导师去精读一部分文献。

B.3.4.3 知识产出

1. 本科产出的学术论文或专利，在投（在申请）的也可列出。如果有这部分内容，很可能会成为老师最核心的关注点，请务必对其背景、技术、创新点、应用等内容做细致的准备。如果不是第一作者，需要说清楚自己负责的内容。
2. 本科的毕业设计可以列出，需要强调自己的创新点。没有创新点也可以列出，作为知识验证能力的一部分。
3. 即使没有付诸实践，可以列一些自己对当前已有技术（尤其是目标导师的工作）的改进思路。

B.3.4.4 知识验证

1. 本科完成的课程设计可以列出，尤其是自己在基本要求之上完成的工作。
2. 本科参加算法类比赛的成绩可以列出。数学建模一类的比赛也在此列。
3. 本科参加项目类比赛的成绩可以列出。由于很多比赛可能老师并不清楚含金量（清华的本科生很少参加比赛，尤其是国内比赛），所以应主要强调项目的创新点和自己的贡献。如果对项目的内容、背景、技术等不够清楚，应提前做好准备。
4. 自己做的论文复现或者自己开发的软硬件作品可以列出。
5. 自己掌握的编程语言、框架、技术等可以列出。在写上去的时候回忆一下自己用这个做过什么，尤其是一些比较冷门的可能引起老师兴趣的内容，比如智能合约这种。
6. 如果是跨考的同学，任何能证明计算机能力的内容都可以列出，一张白纸的简历是最危险的。
7. 可以估算一下自己做工程的能力，如工程开发的效率以及最大的个人工程规模等。代码行数是一个非常直观的体现。声称自己做过一个巨大的项目有可能会得到特别关注，而老师会担心招进来的学生只能解决算法比赛级的小规模代码。简历提到的内容最好都回忆一下。

B.3.4.5 知识表达

1. 本科写作的学术论文（尤其英文学术论文）最佳，不再重复。
2. 英文能力的证明可以列出。包括四六级、托福、雅思、GRE 等。
3. 本科参加英文类比赛的成绩可以列出。
4. 常规的 PPT 都是中文展示、中文讲解的，可以考虑改为使用英文展示或使用英文讲解。

致 谢

感谢 LaTeX 和 ThuThesis 模板对本书的写作提供的帮助。

感谢 Knuth 教授、邓俊辉老师、严蔚敏老师、王道考研编写组、竟成考研编写组的著作为本书的写作提供参考。特别感谢邓俊辉老师讲授的《数据结构》本科课程。

感谢张威师兄在本书写作过程中解答我的疑问。

感谢周洽屹、匡胤鑫、遇见、数据结构罗素、竭泽、redmoon、呆唯、afireswallow、乾之三爻、kingsman、Amour、逃、空空空白等在本书写作过程中参与讨论。

Clazy Chen