数据结构与算法

Data Structure and Algorithm

极夜酱

目录

Part I

基础篇

Chapter 1 数据结构与算法

1.1 算法

1.1.1 算法（Algorithm）

算法是一个很古老的概念，最早来自数学领域。

有一个关于算法的小故事：在很久很久以前，曾经有一个顽皮又聪明的熊孩子，天天在课堂上调皮捣蛋。终于有一天，老师忍无可忍，对熊孩子说："臭小子，你又调皮啊！今天罚你做加法，算出 $1+2+3+\cdots+9999+10000$ 累加的结果，算不完不许回家！"

老师以为，熊孩子会按部就班地一步一步计算：

$$1+2=3$$

$$3+3=6$$

$$6+4=10$$

$$10+5=15$$

$$\cdots$$

这还不得算到明天天亮？够这小子受的！老师心里幸灾乐祸地想着。谁知仅仅几分钟后……

"老师，我算完了！结果是 50005000，对不对?"

"这，这，这……你小子怎么算得这么快？我读书多，你骗不了我的！"

看着老师惊讶的表情，熊孩子微微一笑，讲出了他的计算方法。

首先把从 1 到 10000 这 10000 个数字两两分组相加：

$$1 + 10000 = 10001$$

$$2 + 9999 = 10001$$

$$3 + 9998 = 10001$$

$$4 + 9997 = 10001$$

$$\cdots$$

一共有 $10000 \div 2 = 5000$ 组，所以 1 到 10000 相加的总和可以这样来计算：

$$(1 + 10000) \times 10000 \div 2 = 50005000$$

这个熊孩子就是后来著名的犹太数学家约翰 · 卡尔 · 弗里德里希 · 高斯，而他所采用的这种等差数列求和的方法，被称为高斯算法。

算法是解决问题的一种方法或一个过程，是一个由若干运算或指令组成的有穷序列。求解问题的算法可以看作是输入实例与输出之间的函数。

算法有 5 个特点：

1. 有穷性（finiteness）：算法必须能在执行有限个步骤之后终止。
2. 确定性（definiteness）：算法的每一步骤必须有确切的定义。
3. 输入项（input）：一个算法有 0 个或多个输入。
4. 输出项（output）：一个算法有一个或多个输出，没有输出的算法是毫无意义的。
5. 可行性（effectiveness）：算法中执行的任何计算步骤都是可以被分解为基本的可执行的操作步。

1.1.2 算法描述

算法是可完成特定任务的一系列步骤，算法的计算过程定义明确，通过一些值作为输入并产生一些值作为输出。

流程图（flow chart）是算法的一种图形化表示方式，使用一组预定义的符号来说明如何执行特定任务。

- 圆角矩形：开始和结束
- 矩形：数据处理
- 平行四边形：输入/输出
- 菱形：分支判断条件
- 流程线：步骤

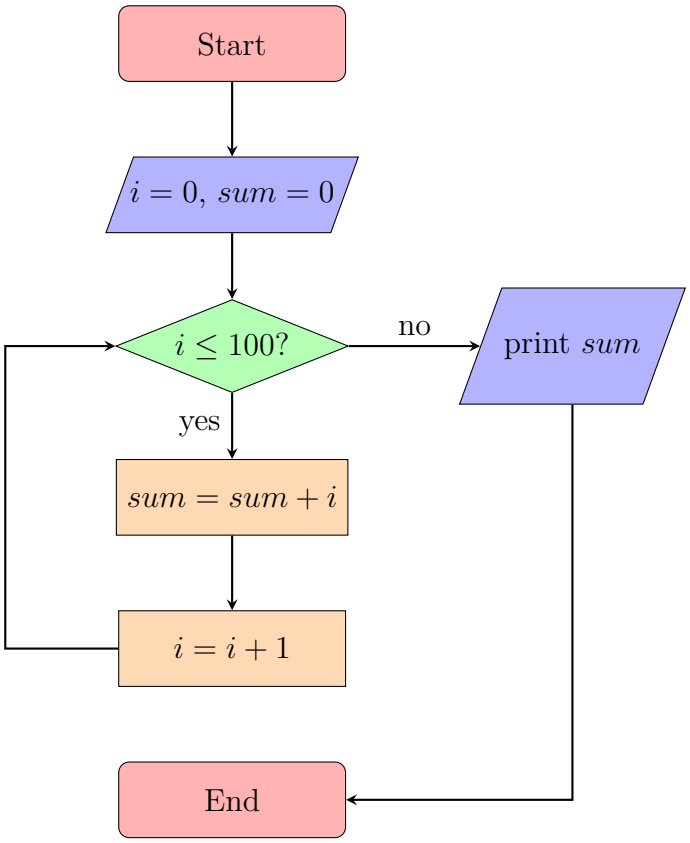


图 1.1: 计算 $\sum_{i=1}^{100} i$ 的流程图

伪代码（pseudocode）是一种非正式的，类似于英语结构的，用于描述模块结构图的语言。使用伪代码的目的是使被描述的算法可以容易地以任何一种编程语言实现。

Algorithm 1 插入排序

```
1: procedure INSERTIONSORT(A[0..n-1])
2:     for j = 2 to n - 1 do
3:         key = A[j]
4:         i = j - 1
5:         while i > 0 and A[i] > key do
6:             A[i+1] = A[i]
7:             i = i - 1
8:         end while
9:         A[i+1] = key
10:    end for
11:    return A
12: end procedure
```

1.2 算法效率

1.2.1 算法效率

算法有高效的，也有拙劣的。在高斯的故事中，高斯所用的算法显然是更加高效的算法，它利用等差数列的规律，四两拨千斤，省时省力地求出了最终结果。而老师心中所想的算法，按部就班地一个数一个数进行累加，则是一种低效、笨拙的算法。虽然这种算法也能得到最终结果，但是其计算过程要低效得多。

在计算机领域，我们同样会遇到各种高效和拙劣的算法。衡量算法好坏的重要标准有两个：时间复杂度、空间复杂度。

让我们来想象一个场景：某一天，小灰和大黄同时加入了同一家公司。老板让他们完成一个需求。一天后，小灰和大黄交付了各自的代码，两人的代码实现的功能差不多。但是，大黄的代码运行一次要花 100 ms，占用内存 5 MB；小灰的代码运行一次要花 100 s，占用内存 500 MB。

“小灰，收拾东西走人，明天不用来上班了!”

小灰虽然也按照老板的要求实现了功能，但他的代码存在两个很严重的问题：运行时间长、占用空间大。

算法效率分析指的是算法求解一个问题所需要的时间资源和空间资源。效率可以通过对算法执行基本运算（步数）的数目进行估算，度量一个算法运算时间的三种方式：

- 最好情形时间复杂度
- 最坏情形时间复杂度
- 平均情形时间复杂度

最坏情形是任何规模为 n 的问题实例运行时间的上界，即任何规模为 n 的实例，其运行时间都不会超过最坏情况的运行时间。

对某些算法，最坏情况经常发生。例如在某个数据库中查询不存在的某条诗句就是查询算法的最坏情形。平均情形有时跟最坏情形差不多。

1.2.2 时间复杂度（Time Complexity）

算法的效率主要取决于算法本身，与计算模型（例如计算机）无关，这样可以通过分析算法的运行时间从而比较出算法之间的快慢。分析一个算法的运行时间应该主要关注与问题规模有关的主要项，其它低阶项，甚至主要项的常数系数都可以忽略。

渐进时间复杂度用大写 O 来表示，所以也被称为大 O 表示法。
时间复杂度有如下原则：

1. 如果运行时间是常数量级，则用 $O(1)$ 表示。
2. 只保留时间函数中的最高阶项。
3. 如果最高阶项存在，则省去最高阶项前面的系数。

在编程的世界中有各种各样的算法，有许多不同形式的时间复杂度，例如：$O(1)$、$O(logn)$、$O(n)$、$O(nlogn)$、$O(n^2)$、$O(2^n)$、$O(n!)$ 等。

1.2.3 空间复杂度（Space Complexity）

内存空间是有限的，在时间复杂度相同的情况下，算法占用的内存空间自然是越小越好。如何描述一个算法占用的内存空间的大小呢？这就用到了算法的另一个重要指标——空间复杂度。

和时间复杂度类似，空间复杂度是对一个算法在运行过程中临时占用存储空间大小的量度，它同样使用了大 O 表示法。

正所谓鱼和熊掌不可兼得，很多时候，我们不得不在时间复杂度和空间复杂度之间进行取舍。在绝大多数时候，时间复杂度更为重要一些，我们宁可多分配一些内存空间，也要提升程序的执行速度。

1.3 基础算法

1.3.1 暴力枚举

暴力破解法也称穷举法，思想就是列举出所有可能情况，然后根据条件判断此答案是否合适，合适就保留，不合适就丢弃。暴力法主要利用计算机运算速度快、精确度高的特点。因此暴力法是通过牺牲时间来换取答案的全面性。

鸡兔同笼

上有三十五头，下有九十四足，问鸡兔各几何？

```
void count(int head, int foot) {
    for(int chicken = 0; chicken <= head; chicken++) {
        int rabbit = head - chicken;
        if(chicken*2 + rabbit*4 == foot) {
            printf("鸡：%2d\t兔：%2d\n", chicken, rabbit);
        }
    }
}
```

百钱买百鸡

公鸡 5 文钱 1 只，母鸡 3 文钱 1 只，小鸡 1 文钱 3 只，如果用 100 文钱买 100 只鸡，那么公鸡、母鸡和小鸡各应该买多少只？

```
void buy(int n, int money) {
    for(int x = 0; x <= n/5; x++) {
        for(int y = 0; y <= n/3; y++) {
            int z = n - x - y;
            if(z > 0 && z % 3 == 0 && 5*x + 3*y + z/3 == money) {
                printf("公鸡：%3d\t母鸡：%3d\t小鸡：%3d\n", x, y, z);
            }
        }
```

```
    }
}
```

1.3.2 字符串逆序

将一个字符串中的字符顺序颠倒过来实现逆序。

字符串逆序

```
void reverse(char *str) {
    int i = 0;
    int j = strlen(str) - 1;
    while(i < j) {
        char temp = str[i];
        str[i] = str[j];
        str[j] = temp;
        i++;
        j--;
    }
}
```

1.3.3 随机算法

随机算法就是在算法中引入随机因素，通过随机数选择算法的下一步操作，它采用了一定程序的随机性作为其逻辑的一部分。

只有随机数生成器的情况下如何计算圆周率的近似值？蒙特卡洛算法就是一种随机算法，用于近似计算圆周率 π 的值。

蒙特卡洛算法是以概率和统计理论方法为基础的一种计算方法，将所求解的问题同一定的概率模型相联系，用电子计算机实现统计模拟或抽样，以获得问题的近似解。为象征性地表明这一方法的概率统计特征，故借用赌城蒙特卡罗命名。

蒙特卡洛算法的基本思想就是当样本数量足够大时，可以用频率去估计概率，这也是求圆周率 π 的常用方法。

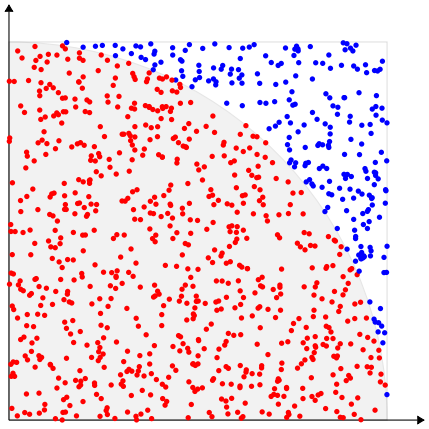


图 1.2: 蒙特卡洛算法

当在 $[0,1]$ 的范围内随机选择一个坐标 (x,y) 时，每个坐标点被选中的概率相等，则坐标落在直径为 1 的正方形中的圆的概率为：

$$P\left(\sqrt{x^2+y^2} \leq 1\right) = \frac{\pi}{4}$$

在生成大量随机点的前提下能得到尽可能接近圆周率的值。

蒙特卡罗算法

```
double montePI(int n) {
    int cnt = 0;        // 圆内点的数量
    for(int i = 0; i < n; i++) {
        double x = rand() / (RAND_MAX + 1.0);   // [0, 1]
        double y = rand() / (RAND_MAX + 1.0);   // [0, 1]
        if(sqrt(x*x + y*y) <= 1) {
            cnt++;
        }
    }
    return 4.0 * cnt / n;
}
```

1.4 数据结构

1.4.1 数据结构（Data Structure）

数据结构是算法基石，是计算机数据的组织、管理和存储的方式，数据结构指的是相互之间存在一种或多种特定关系的数据元素的集合。一个好的数据结构可以带来更高的运行或者存储效率。

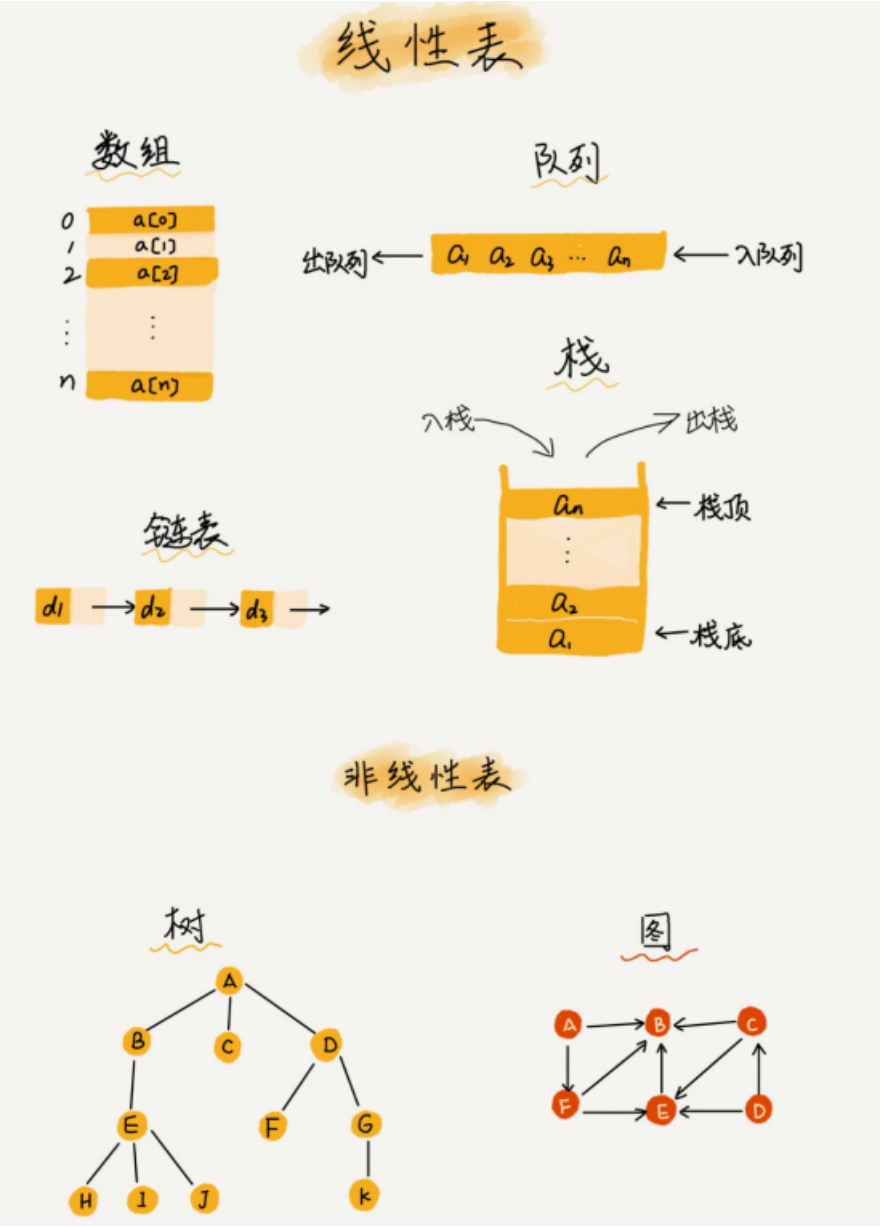


图 1.3: 常用数据结构

Chapter 2 数组

2.1 数组

2.1.1 数组（Array）

数组是数据结构中最简单的结构，很多编程语言都内置数组。数组是有限个相同类型的变量所组成的集合，数组中的每一个变量被称为元素。

创建数组时会在内存中划分出一块连续的内存，将数据按顺序进行存储，数组中的每一个元素都有着自己的下标（index），下标从 0 开始一直到数组长度-1。因为数组在存储数据时是按顺序存储的，存储数据的内存也是连续的。

对于数组来说，读取元素是最简单的操作。由于数组在内存中顺序存储，所以只要给出数组的下标，就可以读取到对应位置的元素。像这种根据下标读取元素的方式叫作随机读取。但是需要注意的是，数组的下标范围必须在 0 到数组长度-1 之内，否则会出现数组越界。数组读取元素的时间复杂度是 $O(1)$。

数组拥有非常高效的随机访问能力，只要给出下标，就可以用常量时间找到对应元素。有一种高效查找元素的算法叫作二分查找，就是利用了数组的这个优势。

数组的劣势体现在插入和删除元素方面。由于数组元素连续紧密地存储在内存中，插入、删除元素都会导致大量元素被迫移动，影响效率。总的来说，数组所适合的是读操作多、写操作少的场景。

更新数组元素

```
1 arr = {3, 1, 2, 5, 4, 9, 7, 2}
2 arr[5] = 10
3 print(arr[5])
```

2.2 查找算法

2.2.1 顺序查找（Sequence Search）

顺序查找也称线性查找，是一种按照序列原有顺序进行遍历比较查询的基本查找算法。

对于任意一个序列以及一个需要查找的元素（关键字），将关键字与序列中元素依次比较，直到找出与给定关键字相同的元素，或者将序列中的元素与其都比较完为止。若某个元素的值与关键字相等，则查找成功；如果直到最后一个元素，元素的值和关键字比较都不等时，则查找不成功。

最好的情况就是在第一个位置就找到，算法时间复杂度为 $O(1)$。

最坏情况是关键字不存在，需要进行 n 次比较，时间复杂度为 $O(n)$。

平均查找次数为 $(n+1)/2$，平均时间复杂度为 $O(n)$。

顺序查找

```
1  int sequenceSearch(int *arr, int n, int key) {
2      for(int i = 0; i < n; i++) {
3          if(arr[i] == key) {
4              return i;
5          }
6      }
7      return -1;
8  }
```

2.2.2 二分查找（Binary Search）

二分查找法也称折半查找，是一种效率较高的查找方法。折半查找要求线性表必须采用顺序存储结构，而且表中元素按关键字有序排列。

算法思想是假设表中元素是按升序排列，将表中间位置的关键字与查找关键字比较，如果两者相等，则查找成功；否则利用中间位置记录将表分成前、后两个子表，如果中间位置的关键字大于查找关键字，则进一步查找前一子表，否则进一步查找后一子表。重复以上过程，直到找到满足条件的记录，使查找成功，或直到子表不存在为止，此时查找不成功。

二分查找法的时间复杂度为 $O(logn)$。

二分查找

```
1  int binarySearch(int *arr, int n, int key) {
2      int start = 0;
3      int end = n - 1;
4      while(start <= end) {
5          int mid = (start + end) / 2;
6          if(arr[mid] == key) {
7              return mid;
8          } else if(arr[mid] < key) {
9              start = mid + 1;
10         } else {
11             end = mid - 1;
12         }
13     }
14     return -1;
15 }
```

2.3 数组元素插入与删除

2.3.1 插入元素

在数组中插入元素存在 3 种情况：

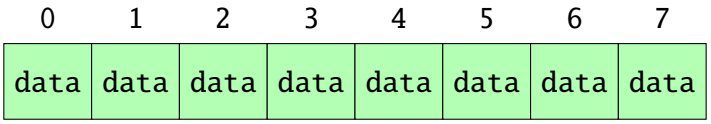


图 2.1: 数组

尾部插入

直接把插入的元素放在数组尾部的空闲位置即可。

中间插入

首先把插入位置及后面的元素向后移动，腾出位置，再把要插入的元素放入该位置上。

插入元素

```
1  int insert(int *arr, int n, int index, int val) {
2      if(index < 0 || index >= n) {
3          return n;
4      }
5      for(int i = n - 1; i >= index; i--) {
6          arr[i+1] = arr[i];
7      }
8      arr[index] = val;
9      n++;
10     return n;
11 }
```

超范围插入

数组的长度在创建时就已经确定了，要实现数组的扩容，只能创建一个新数组，长度是旧数组的 2 倍，再把旧数组中的元素全部复制过去，这样就实现了数组的扩容。

数组插入元素最好情况是尾部插入，无需移动任何元素，时间复杂度为 $O(1)$。最坏情况是在第一个位置插入，这样就需要移动后面所有 $n-1$ 个元素，时间复杂度为 $O(n)$。因此，总体的时间复杂度为 $O(n)$。

2.3.2 删除元素

数组的删除操作与插入操作过程相反，如果被删除的元素位于数组中间，其后的元素都需要向前挪动一位。

删除元素

```
int delete(int *arr, int n, int index) {
    if(index < 0 || index >= n) {
        return n;
    }
    for(int i = index + 1; i < n; i++) {
        arr[i-1] = arr[i];
    }
    n--;
    return n;
}
```

数组的删除操作，由于只涉及元素的移动，时间复杂度为 $O(n)$。

对于删除操作，其实还存在一种取巧的方式，前提是数组元素没有顺序要求。如需要删除数组中某个元素，可直接把最后一个元素复制到被删除元素的位置，然后再删除最后一个元素。这样一来，无须进行大量的元素移动，时间复杂度降低为 $O(1)$。当然，这种方式只作参考，并不是删除元素主流的操作方式。

Chapter 3 链表

3.1 链表

3.1.1 单向链表（Singly Linked List）

为避免元素的移动，采用线性表的另一种存储方式：链式存储结构。链表是一种在物理上非连续、非顺序的数据结构，由若干结点（node）所组成。

单向链表的每一个结点又包含两部分，一部分是存放数据的数据域 data，另一部分是指向下一个结点的指针域 next。结点可以在运行时动态生成。

```
1 typedef struct Node {
2     dataType data;          // 数据域
3     struct Node *next;      // 指针域
4 } Node;
```

链表的第一个结点被称为头结点，最后一个节点被称为尾结点，尾结点的 next 指针指向空 NULL。

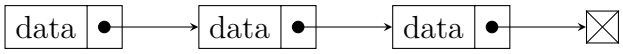


图 3.1: 单向链表

与数组按照下标来随机寻找元素不同，对于链表的其中一个结点 A，只能根据结点 A 的 next 指针来找到该结点的下一个结点 B，再根据结点 B 的 next 指针找到下一个节点 C……

数组在内存中的存储方式是顺序存储，链表在内存中的存储方式则是随机存储。链表采用了见缝插针的方式，每一个结点分布在内存的不同位置，依靠 next 指针关联起来。这样可以灵活有效地利用零散的碎片空间。

3.1.2 双向链表（Doubly Linked List）

那么，通过链表的一个结点，如何能快速找到它的前一个结点呢？要想让每个结点都能回溯到它的前置结点，可以使用双向链表。

双向链表比单向链表稍微复杂一点，它的每一个结点除了拥有 data 和 next 指针，还拥有指向前置结点的 prev 指针。

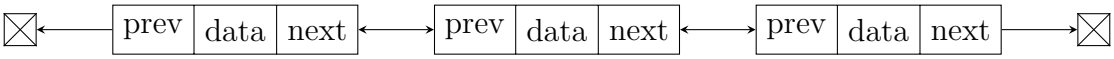


图 3.2: 双向链表

单向链表只能从头到尾遍历，只能找到后继，无法找到前驱，因此遍历的时候不会死循环。而双向链表需要多分配一个指针的存储空间，每个结点有两个指针，分别指向直接前驱和直接后继。

3.1.3 循环链表（Circular Linked List）

除了单向链表和双向链表以外，还有循环链表。对于单向循环链表，尾结点的 next 指针指向头结点。对于双向循环链表，尾结点的 next 指针指向头结点，并且头结点的 prev 指针指向尾结点。

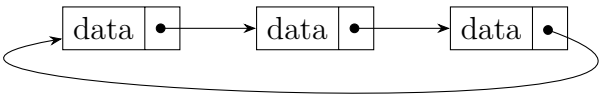


图 3.3: 单向循环链表

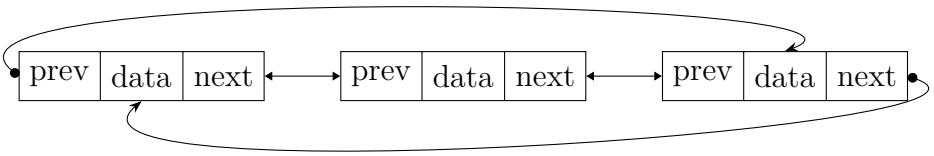


图 3.4: 双向循环链表

3.2 链表的增删改查

3.2.1 查找结点

在查找元素时，链表不像数组那样可以通过下标快速进行定位，只能从头结点开始向后一个一个结点逐一查找。

链表中的数据只能按顺序进行访问，最坏的时间复杂度是 $O(n)$。

查找结点

```
Node* search(List *head, dataType val) {
    // 查找元素位置
    Node *temp = head;
    while(temp) {
        if(temp->data == val) {
            return temp;
        }
        temp = temp->next;
    }
    return NULL;        // 未找到
}
```

3.2.2 更新结点

如果不考虑查找结点的过程，链表的更新过程会像数组那样简单，直接把旧数据替换成新数据即可。

更新结点

```
void replace(List *head, int pos, dataType val) {
    // 找到元素位置
    Node *temp = head;
    for(int i = 0; i < pos; i++) {
```

```
5           temp = temp->next;
6       }
7       temp->data = val;
8   }
```

3.2.3 插入结点

链表插入结点，分为 3 种情况：

尾部插入

把最后一个结点的 next 指针指向新插入的结点。

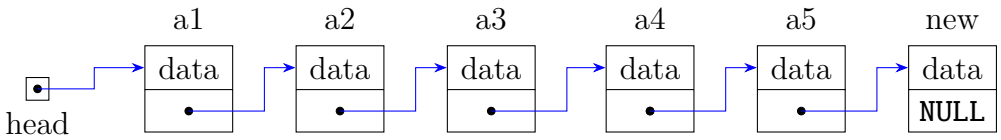


图 3.5: 尾部插入

头部插入

先把新结点的 next 指针指向原先的头结点，再把新结点设置为链表的头结点。

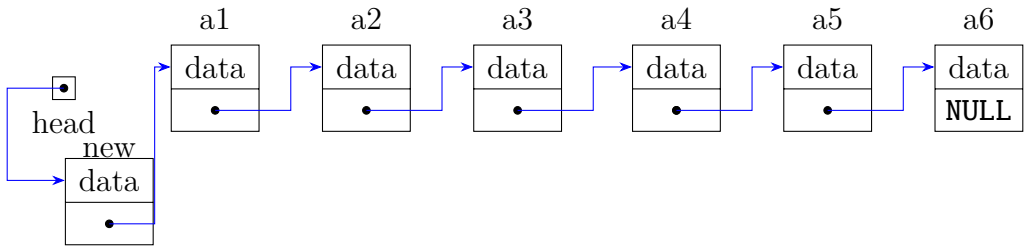


图 3.6: 头部插入

中间插入

先把新结点的 next 指针指向插入位置的结点，再将插入位置的前置结点的 next 指针指向新结点。

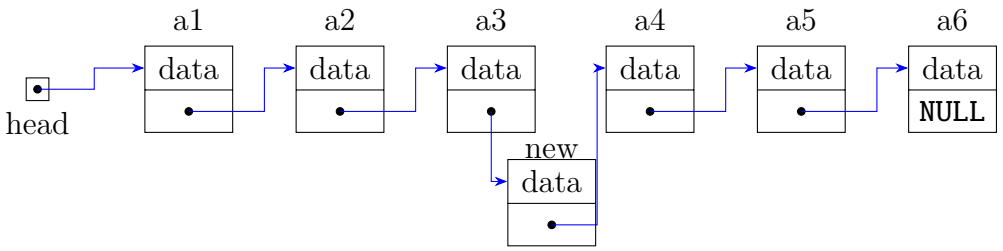


图 3.7: 中间插入

只要内存空间允许，能够插入链表的元素是无穷无尽的，不需要像数组考虑扩容的问题。如果不考虑插入之前的查找元素的过程，只考虑纯粹的插入操作，时间复杂度是 $O(1)$。

3.2.4 删除结点

链表的删除操作也分 3 种情况：

尾部删除

把倒数第二个结点的 next 指针指向空。

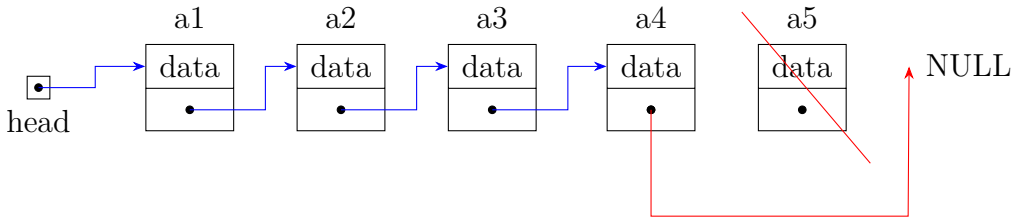


图 3.8: 尾部删除

头部删除

把链表的头结点设置为原先头结点的 next 指针。

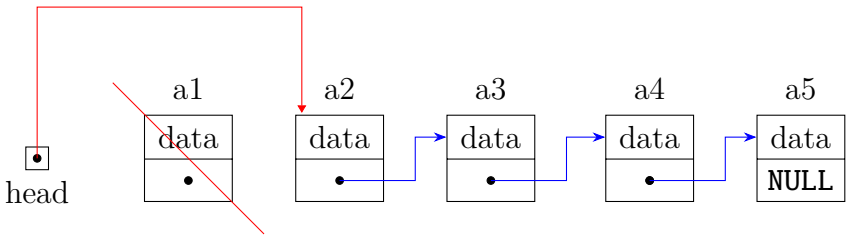


图 3.9: 头部删除

中间删除

把要删除的结点的前置结点的 next 指针，指向要删除结点的下一个结点。

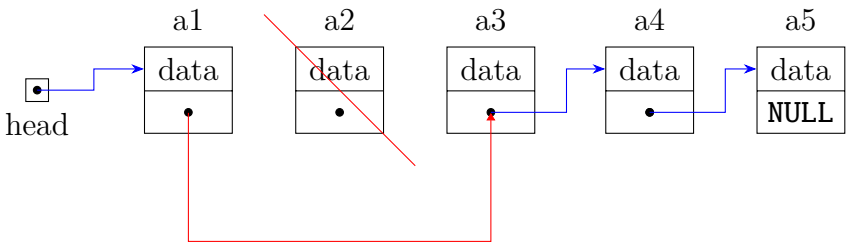


图 3.10: 中间删除

许多高级语言，如 Java，拥有自动化的垃圾回收机制，所以不用刻意去释放被删除的结点，只要没有外部引用指向它们，被删除的结点会被自动回收。

如果不考虑删除操作之前的查找的过程，只考虑纯粹的删除操作，时间复杂度是 $O(1)$。

3.3 带头结点的链表

3.3.1 带头结点的链表

为了方便链表的插入、删除操作，在链表加上头结点之后，无论链表是否为空，头指针始终指向头结尾。因此对于空表和非空表的处理也统一了，方便了链表的操作，也减少了程序的复杂性和出现 bug 的机会。

插入结点

```
1  void insert(List *head, int pos, dataType val) {
2      Node *newNode = (Node *)malloc(sizeof(Node));
3      newNode->data = val;
4      newNode->next =  NULL;
5
6      // 找到插入位置
7      Node *temp = head;
8      for(int i = 0; i < pos; i++) {
9          temp = temp->next;
10     }
11     newNode->next = temp->next;
12     temp->next = newNode;
13 }
```

删除结点

```
1  void delete(List *head, int pos) {
2      Node *temp = head;
3      for(int i = 0; i < pos; i++) {
4          temp = temp->next;
5      }
6      Node *del = temp->next;
7      temp->next = del->next;
8      free(del);
9      del = NULL;
```

```
10 }
```

3.3.2 数组 VS 链表

数据结构没有绝对的好与坏，数组和链表各有千秋。

比较内容	数组	链表
基本	一组固定数量的数据项	可变数量的数据项
大小	声明期间指定	无需指定，执行期间增长或收缩
存储分配	元素位置在编译期间分配	元素位置在运行时分配
元素顺序	连续存储	随机存储
访问元素	直接访问：索引、下标	顺序访问：指针遍历
插入/删除	速度慢	快速、高效
查找	线性查找、二分查找	线性查找
内存利用率	低效	高效

表 3.1: 数组 VS 链表

数组的优势在于能够快速定位元素，对于读操作多、写操作少的场景来说，用数组更合适一些。

相反，链表的优势在于能够灵活地进行插入和删除操作，如果需要频繁地插入、删除元素，用链表更合适一些。

3.4 倒数第 k 个结点

3.4.1 倒数第 k 个结点

输入一个链表，输出该链表中倒数第 k 个结点。例如一个链表有 6 个结点 [0, 1, 2, 3, 4, 5]，这个链表的倒数第 3 个结点是值为 3 的结点。

算法的思路是设置两个指针 p1 和 p2，它们都从头开始出发，p2 指针先出发 k 个结点，然后 p1 指针再进行出发，当 p2 指针到达链表的尾结点时，则 p1 指针的位置就是链表的倒数第 k 个结点。

倒数第 k 个结点

```
1  public static Node findLastKth(LinkedList list, int k) {
2      Node p1 = list.getHead();
3      Node p2 = list.getHead();
4
5      int i = 0;
6      while(p2 != null && i < k) {
7          p2 = p2.next;
8          i++;
9      }
10     while(p2 != null) {
11         p1 = p1.next;
12         p2 = p2.next;
13     }
14     return p1;
15 }
```

3.5 环形链表

3.5.1 环形链表

一个单向链表中有可能出现环，不允许修改链表结构，如何在时间复杂度 $O(n)$、空间复杂度 $O(1)$ 内判断出这个链表是有环链表？如果带环，环的长度是多少？环的入口结点是哪个？

暴力算法首先从头结点开始，依次遍历单链表的每一个结点。对于每个结点都从头重新遍历之前的所有结点。如果发现当前结点与之前结点存在相同 ID，则说明该结点被遍历过两次，链表有环。但是这种方法的时间复杂度太高。

另一种方法就是利用快慢指针，首先创建两个指针 p1 和 p2，同时指向头结点，然后让 p1 每次向后移动一个位置，让 p2 每次向后移动两个位置。在环中，快指针一定会反复遇到慢指针。比如在一个环形跑道上，两个运动员在同一地点起跑，一个运动员速度快，一个运动员速度慢。当两人跑了一段时间，速度快的运动员必然会从速度慢的运动员身后再次追上并超过。

环的长度可以通过从快慢指针相遇的结点开始再走一圈，下一次回到该点的时的移动步数，即环的长度 n。

环的入口可以利用类似获取链表倒数第 k 个结点的方法，准备两个指针 p1 和 p2，让 p2 先走 n 步，然后 p1 和 p2 一块走。当两者相遇时，即为环的入口处。

环形链表

```
1  public static Node cycleNode(LinkedList list) {
2      Node p1 = list.getHead();
3      Node p2 = list.getHead();
4
5      while(p1 != null && p2 != null) {
6          if(p2.next == null || p2.next.next == null) {
7              return null;
```

```
8           }
9           p1 = p1.next;
10          p2 = p2.next.next;
11          if(p1 == p2) {
12              return p1;
13          }
14      }
15      return null;
16  }
17
18  public static int cycleLength(LinkedList list) {
19      Node node = cycleNode(list);
20      if(node == null) {
21          return 0;
22      }
23      int len = 1;
24      Node cur = node.next;
25      while(cur != node) {
26          cur = cur.next;
27          len++;
28      }
29      return len;
30  }
31
32  public static Node cycleEntrance(LinkedList list) {
33      int n = cycleLength(list);
34      if(n == 0) {
35          return null;
36      }
37
38      Node p1 = list.getHead();
39      Node p2 = list.getHead();
40      for(int i = 0; i < n; i++) {
41          p2 = p2.next;
42      }
43
44      while(p1 != p2) {
```

```
45          p1 = p1.next;
46          p2 = p2.next;
47      }
48      return p1;
49  }
```

3.6 反转链表

3.6.1 逆序输出链表

输入一个单链表，从尾到头打印链表每个结点的值。由于单链表的遍历只能从头到尾，所以可以通过递归达到链表尾部，然后在回溯时输出每个结点的值。

逆序输出链表

```
1 public static void inverse(Node head) {
2     if(head != null) {
3         inverse(head.next);
4         System.out.print(head.data + " ");
5     }
6 }
```

3.6.2 反转链表

反转一个链表需要调整链表中指针的方向。

递归反转法的实现思想是从链表的尾结点开始，依次向前遍历，遍历过程依次改变各结点的指向，即另其指向前一个结点。

而迭代反转法的实现思想非常直接，就是从当前链表的首结点开始，一直遍历至链表尾部，期间会逐个改变所遍历到的结点的指针域，另其指向前一个结点。

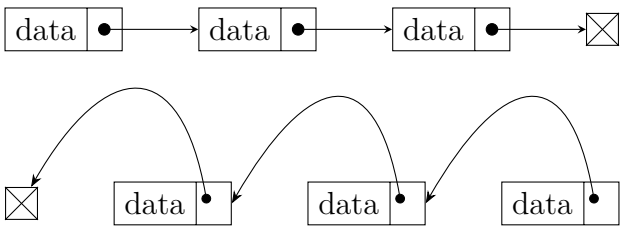


图 3.11: 反转链表

反转链表（递归）

```java
public static Node reverseList(Node head) {
    if(head == null || head.next == null) {
        return head;
    }
    Node next = head.next;
    head.next = null;
    Node newHead = reverseList(next);
    next.next = head;
    return newHead;
}
```

反转链表（迭代）

```java
public static Node reverseListIterative(LinkedList list) {
    Node newHead = new Node(-1);
    Node head = list.getHead();
    while(head != null) {
        Node next = head.next;
        head.next = newHead.next;
        newHead.next = head;
        head = next;
    }
    return newHead.next;
}
```

3.7 跳表

3.7.1 跳表

给定一个有序链表，如何根据给定元素的值进行高效率查找?

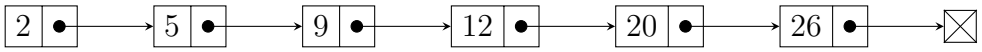


图 3.12: 有序链表

是不是可以用二分查找，先找到中间元素?

链表和数组可不一样！数组能直接用下标快速定位，而链表只能从头结点一个一个元素向后找。对于链表，确实没办法像数组那样进行二分查找，但是可以在链表的基础上做一个小小的升级。

如果给你一本书，要求你快速翻到书中的第 5 章，当然是首先查阅书的目录，根据目录提示，翻到第 5 章所对应的页码。

其实一个链表也可以拥有自己的目录，或者说索引。

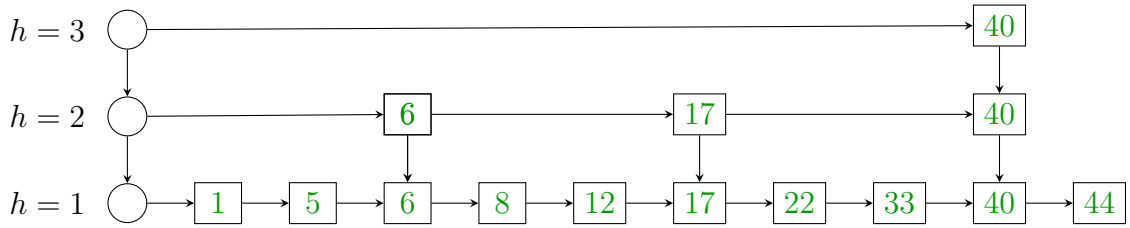


图 3.13: 跳表

在原始链表的基础上，增加了一个索引链表。原始链表的每三个结点，有一个结点在索引链表当中。当需要查找结点 17 的时候，不需要在原始链表中一个一个结点访问，而是首先访问索引链表。在索引链表找到结点 17 之后，顺着索引链表的结点向下，找到原始链表的结点 17。

这个过程，就像是先查阅了图书的目录，再翻到章节所对应的页码。由于索引链表的结点个数是原始链表的一半，查找结点所需的访问次数也相应减少了一半。

这个过程，就像是先查阅了图书的目录，再翻到章节所对应的页码。由于索引链表的结点个数是原始链表的一半，查找结点所需的访问次数也相应减少了一半。基于原始链表的第 1 层索引，抽出了第 2 层更为稀疏的索引，结点数量是第 1 层索引的一半。这样的多层索引可以进一步提升查询效率。

假设原始链表有 n 个结点，那么索引的层级就是 $logn-1$，在每一层的访问次数是常量，因此查找结点的平均时间复杂度是 $O(logn)$，这比起常规的线性查找效率要高得多。

但相应的，这种基于链表的优化增加了额外的空间开销，各层索引的结点总数是 $\frac{n}{2}+\frac{n}{4}+\frac{n}{8}+\frac{n}{16}+\cdots\approx n$。也就是说，优化之后的数据结构所占空间，是原来的 2 倍，这是典型的以空间换时间的做法。

像这样基于链表改造的数据结构，有一个全新的名字，叫做跳表。

3.7.2 跳表插入结点

假设要插入的结点是 10，首先按照跳表查找结点的方法，找到待插入结点的前置结点（仅小于待插入结点）。按照一般链表的插入方式，把结点 10 插入到结点 8 的下一个位置。

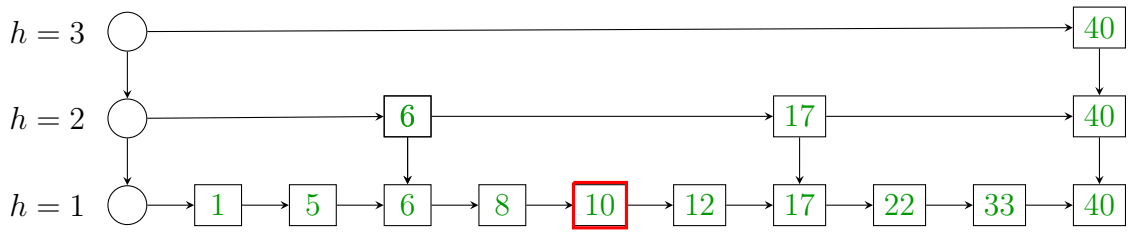


图 3.14: 插入结点 10

这样是不是插入工作就完成了呢？并不是。随着原始链表的新结点越来越多，索引会渐渐变得不够用了，因此索引结点也需要相应作出调整。

调整索引的方法就是让新插入的结点随机晋升成为索引结点，晋升成功的几率是50%。新结点在成功晋升之后，仍然有机会继续向上一层索引晋升。

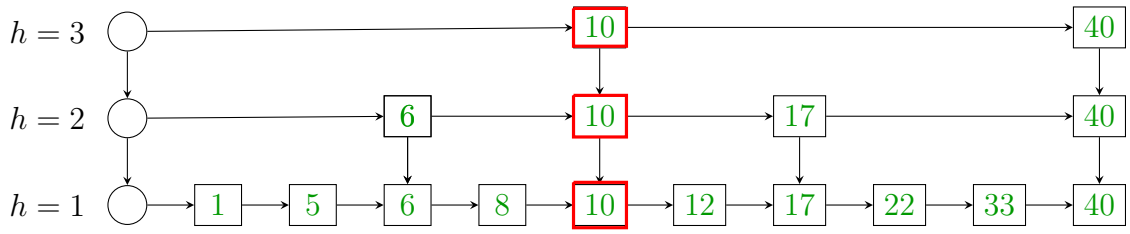


图 3.15: 晋升结点

3.7.3 跳表删除结点

至于跳表删除结点的过程，则是相反的思路。

假设要从跳表中删除结点 10，首先按照跳表查找结点的方法，找到待删除的结点。按照一般链表的删除方式，把结点 10 从原始链表当中删除。这样是不是删除工作就完成了呢？并不是。还需要顺藤摸瓜，把索引当中的对应结点也一一删除。

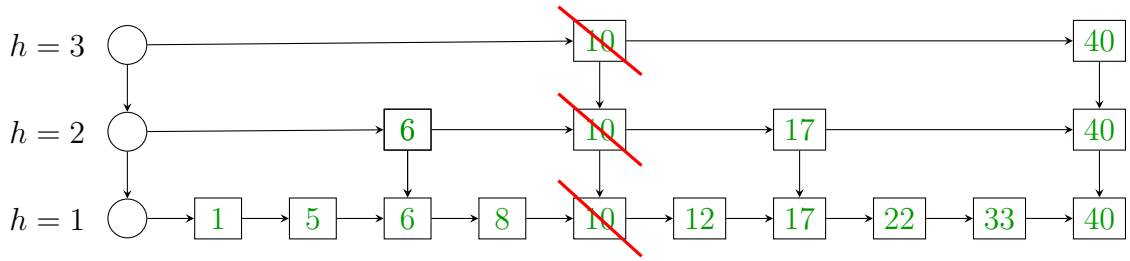


图 3.16: 删除结点

在实际的程序中，跳表采用的是双向链表，无论前后结点还是上下结点，都各有两个指针相互指向彼此。并且跳表的每一层首尾各有一个空结点，左侧的空节点是负无穷大，右侧的空节点是正无穷大。

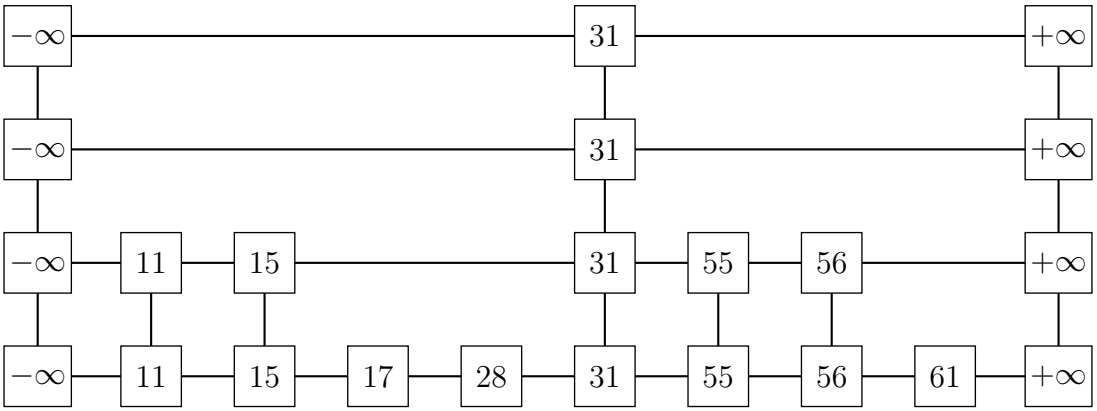


图 3.17: 跳表

Chapter 4 栈

4.1 栈

4.1.1 栈（Stack）

栈，又名堆栈，是一种运算受限的线性数据结构，栈只能在表尾进行插入和删除操作。

栈中的元素只能先进后出（FILO, First In Last Out）。最早进入栈的元素所存放的位置叫作栈底（bottom），最后进入栈的元素存放的位置叫作栈顶（top）。

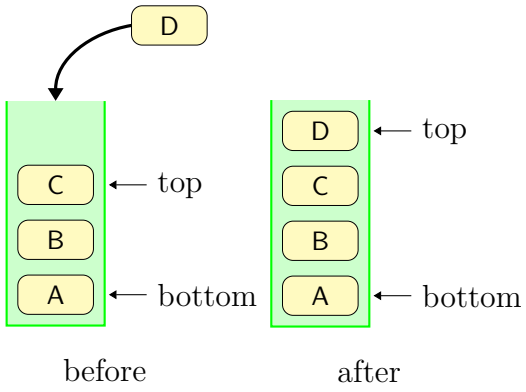


图 4.1: 栈

栈这种数据结构既可以用数组来实现，也可以用链表来实现。

4.1.2 顺序栈

使用数组方式实现的栈称为静态栈。可以根据下标来表示栈顶在数组中的位置，对于空栈，栈顶为-1。

进行入栈操作时，栈顶指针 +1；出栈时，栈顶指针-1。

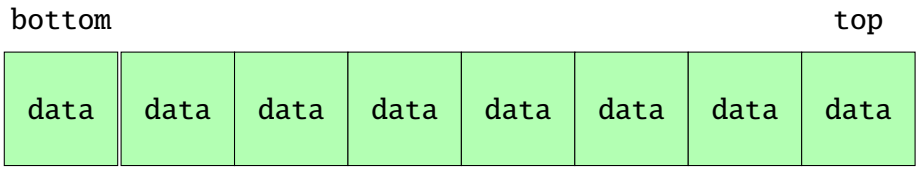


图 4.2: 顺序栈

对满栈进行入栈和对空栈进行出栈操作操作都会产生数组的越界并引起程序崩溃，称为上溢和下溢。因此使用顺序栈前需要提前声明一个数组的大小，如果数组大小不够则可能发生数组越界，如果数组太大则会浪费一定的空间。

使用数组实现的栈的执行效率会比用链表来实现的高，入栈和出栈不需要移动大量元素，只需要移动栈顶指针即可。

4.1.3 链式栈

使用链表方式实现的栈称为动态栈。通过在表头插入一个元素来实现入栈，通过删除表尾元素来实现出栈。

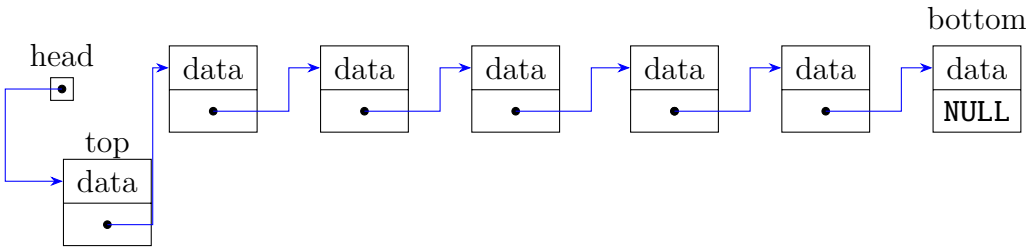


图 4.3: 链式栈

动态栈有链表的部分特性，元素与元素之间在物理存储上可以不连续，但是功能有些受限制，动态栈只能在栈顶处进行插入和删除操作，不能在栈尾或栈中间进行插入和删除操作。

动态栈的元素内存是动态分配的，避免了静态栈可能会浪费空间的问题，但是对申请和释放空间的调用开销会比较大。

4.1.4 栈的应用

栈的输出顺序和输入顺序相反，所以栈同行用于对历史的回溯。例如实现递归的逻辑，就可以用栈回溯调用链。

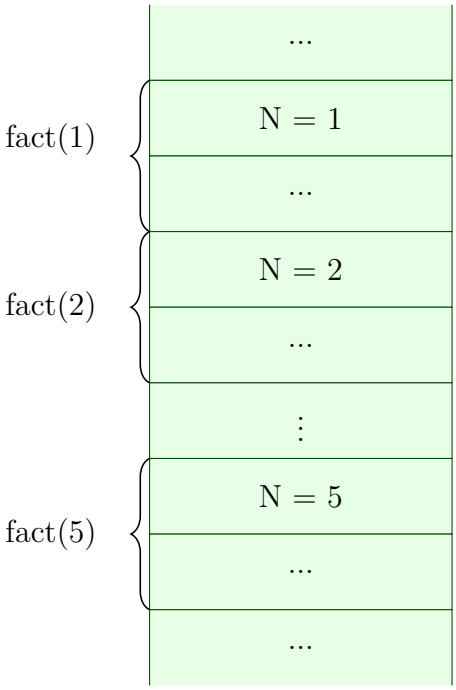


图 4.4: 函数调用栈

栈还有一个著名的应用场景就是面包屑导航，使用户在流浪页面时可以轻松地回溯到上一级更更上一级页面。

图 4.5: 面包屑导航

4.2 入栈与出栈

4.2.1 入栈（Push）

入栈操作就是把新元素放入栈中，只允许从栈顶一侧放入元素，新元素的位置将会成为新的栈顶。最初，栈为空，栈顶的初始值为-1。每当向栈中添加元素时，栈顶指针 +1。

入栈只影响最后一个元素，不涉及元素的整体移动，所以无论是以数组还是链表实现，时间复杂度都是 $O(1)$。

入栈

```
1 void push(Stack *stack, dataType val) {
2     stack->data[++stack->top] = val;
3 }
```

4.2.2 出栈（Pop）

出栈操作就是把新元素从栈中弹出，只有栈顶元素才允许出栈，出栈元素的前一个元素将会成为新的栈顶。从栈中移出元素，栈顶指针-1。数组中元素的删除并非真正意义上把元素从内存中清除，出栈只需对栈顶-1 即可，后期向栈中添加元素时，新元素会将旧元素覆盖。

出栈只影响最后一个元素，不涉及元素的整体移动，所以无论是以数组还是链表实现，时间复杂度都是 $O(1)$。

出栈

```
1 dataType pop(Stack *stack) {
2     return stack->data[stack->top--];
3 }
```

4.3 最小栈

4.3.1 最小栈

设计一个支持 push()、pop()、peek() 和 getMin() 操作的栈，并能在常数时间内检索到最小元素。

对于栈来说，如果一个元素 a 在入栈时，栈里有其它的元素 b、c、d，那么无论这个栈在之后经历了什么操作，只要 a 在栈中，b、c、d 就一定在栈中。因此，在操作过程中的任意一个时刻，只要栈顶的元素是 a，那么就可以确定栈里面现在的元素一定是 a、b、c、d。

那么可以在每个元素 a 入栈时把当前栈的最小值 m 存储起来。在这之后无论何时，如果栈顶元素是 a，就可以直接返回存储的最小值 m。

当一个元素要入栈时，取辅助栈的栈顶存储的最小值，与当前元素比较得出最小值，将这个最小值插入辅助栈中。当一个元素要出栈时，把辅助栈的栈顶元素也一并弹出。这样在任意一个时刻，栈内元素的最小值就存储在辅助栈的栈顶元素中。

最小栈

```
class MinStack:
    def __init__(self):
        self.stack = []
        self.min_stack = [math.inf]

    def push(self, data):
        self.stack.append(data)
        self.min_stack.append(min(data, self.min_stack[-1]))

    def pop(self):
        self.stack.pop()
        self.min_stack.pop()
```

```
13
14      def peek(self):
15          return self.stack[-1]
16
17      def get_min(self):
18          return self.min_stack[-1]
```

4.4 括号匹配

4.4.1 括号匹配

给定一个只包括"("、")"、"["、"]"、"{" 和"}" 的字符串，判断字符串是否有效。有效字符串需满足左括号必须用相同类型的右括号闭合，并且左括号必须以正确的顺序闭合。

判断括号的有效性可以使用栈来解决。通过遍历字符串，当遇到左括号时，会期望在后续的遍历中，有一个相同类型的右括号将其闭合。由于后遇到的左括号要先闭合，因此将这个左括号放入栈顶。

当遇到右括号时，需要将一个相同类型的左括号闭合。此时可以取出栈顶的左括号并判断它们是否是相同类型的括号。如果不是相同的类型，或者栈中并没有左括号，那么字符串无效。在遍历结束后，如果为空栈，说明字符串中的所有左括号闭合。

注意有效字符串的长度一定为偶数，因此如果字符串的长度为奇数，可以直接返回判断出字符串无效，省去后续的遍历判断过程。

括号匹配

```
def valid_paratheses(s):
    if len(s) % 2 == 1:
        return False

    pairs = {")": "(", "]": "[", "}": "{"}
    stack = list()
    for paran in s:
        if paran in pairs:
            if not stack or stack[-1] != pairs[paran]:
                return False
            stack.pop()
        else:
```

```
13              stack.append(paran)
14
15      return not stack
```

4.5 表达式求值

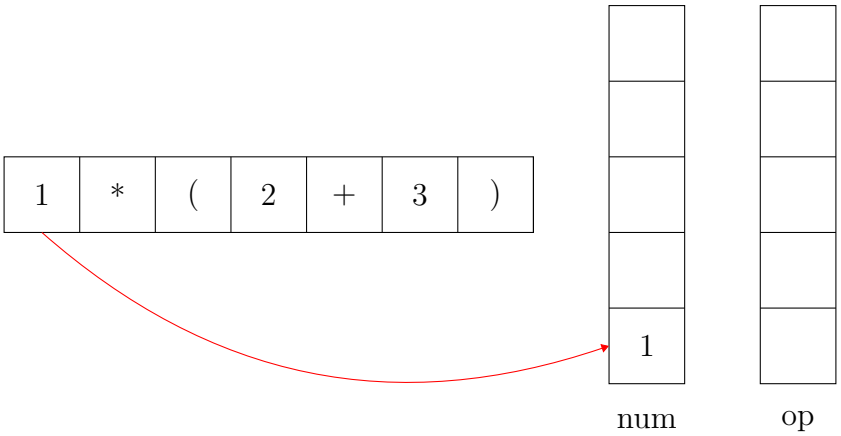
4.5.1 表达式求值

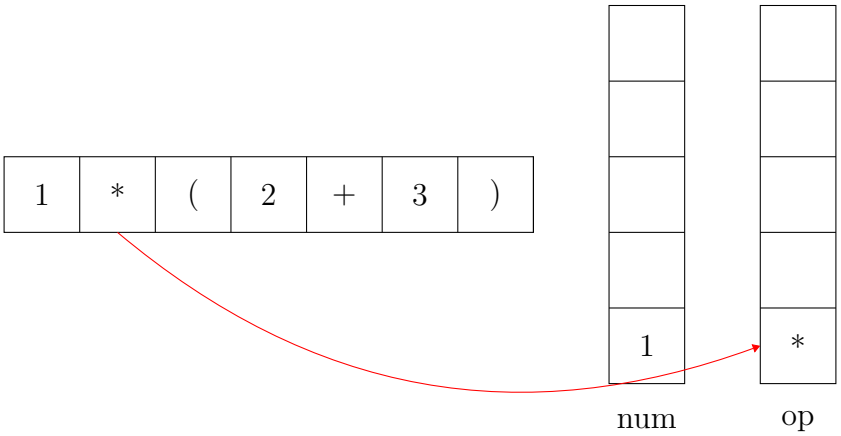
逆波兰表达式是一种后缀表达式，所谓后缀就是指运算符写在运算数的后面。平常使用的算式则是一种中缀表达式，如 $(1+2)*(3+4)$，该算式的逆波兰表达式写法为 $1\ 2+3\ 4+*$。

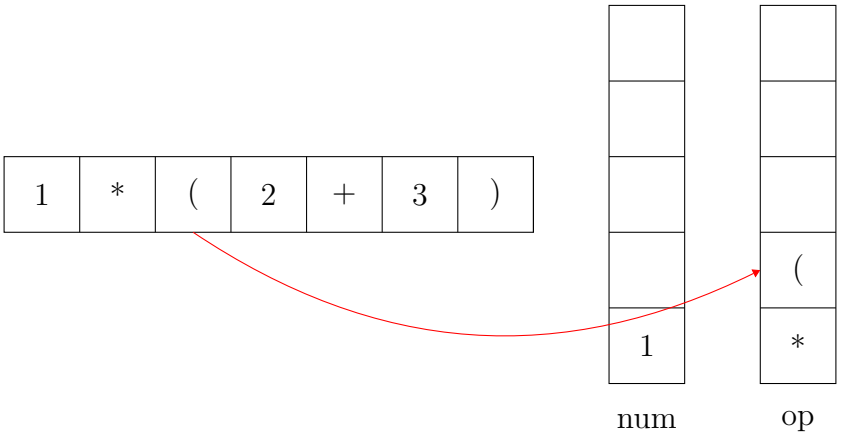
逆波兰表达式的优点在于去掉了中缀表达式中的括号后表达式无歧义，因此适合用栈操作运算。遇到数字则入栈，遇到算符则取出栈顶两个数字进行计算，并将结果压入栈中。

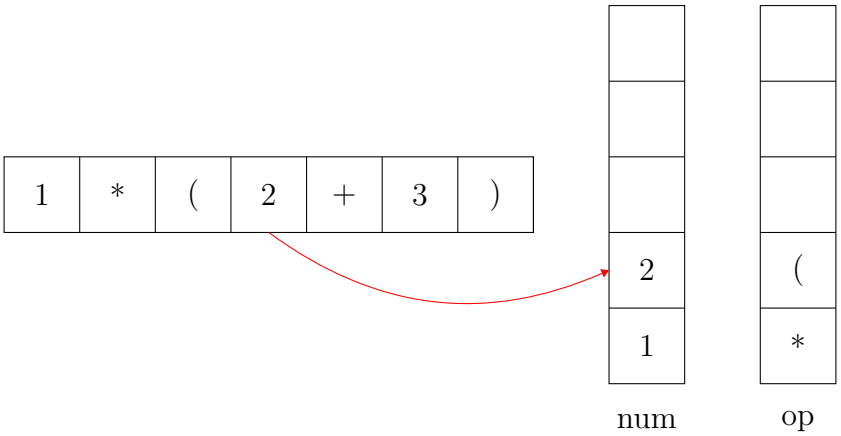
在对中缀表达式求值时，一般都会将其转换为后缀表达式的形式，转换过程同样需要用到栈，规则如下：

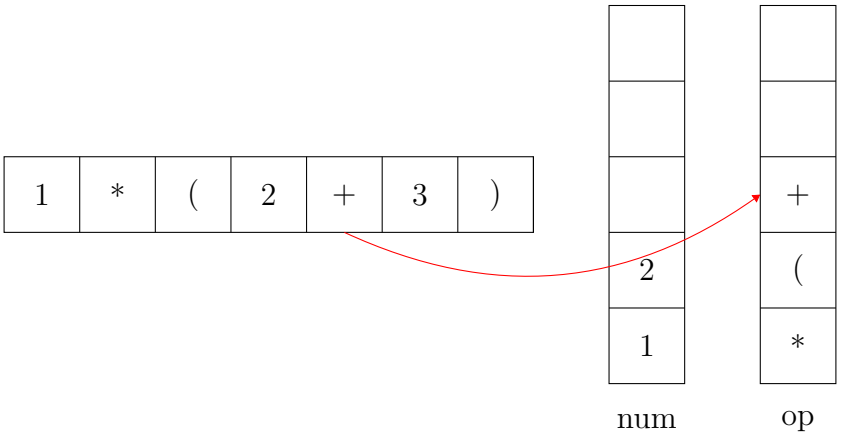
1. 如果遇到操作数，就直接将其输出。
2. 如果遇到左括号，将其放入栈中。
3. 如果遇到右括号，则一直出栈并输出，直到遇到左括号为止。注意，左括号只出栈并不输出。
4. 如果遇到任何其它的运算符，如果为栈为空，则直接入栈。否则从栈中出栈元素并输出，直到遇到优先级更低的元素（或者栈为空）位置。在出栈完这些元素后，再将当前遇到的运算符入栈。有一点需要注意，只有在遇到右括号的情况下才将左括号出栈，其它情况都不会出栈左括号。
5. 如果读取到了表达式的末尾，则将栈中所有元素依次出栈输出。

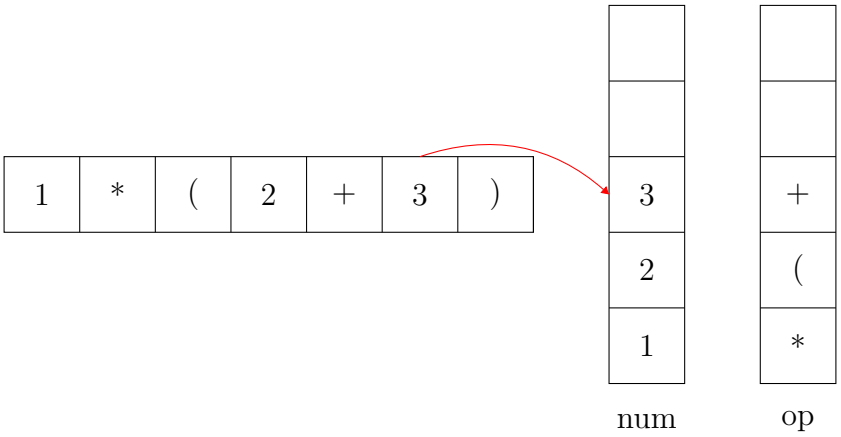

1 * (2 + 3)
1
num
op

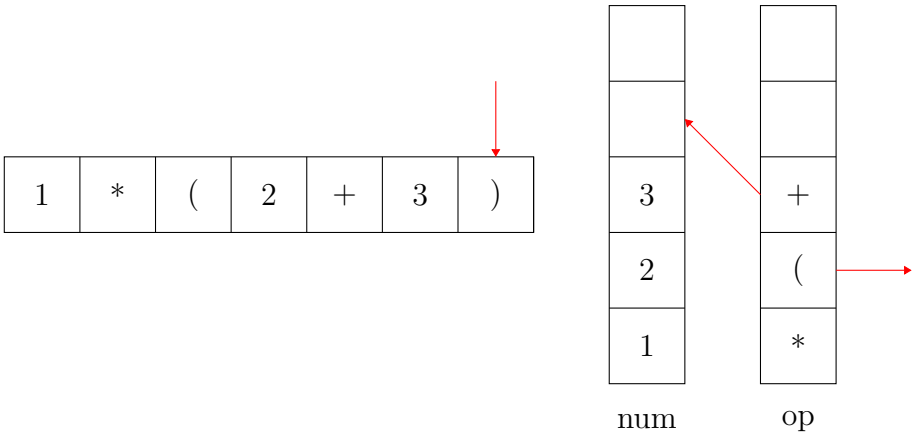

1 * (2 + 3)
1
*
num
op

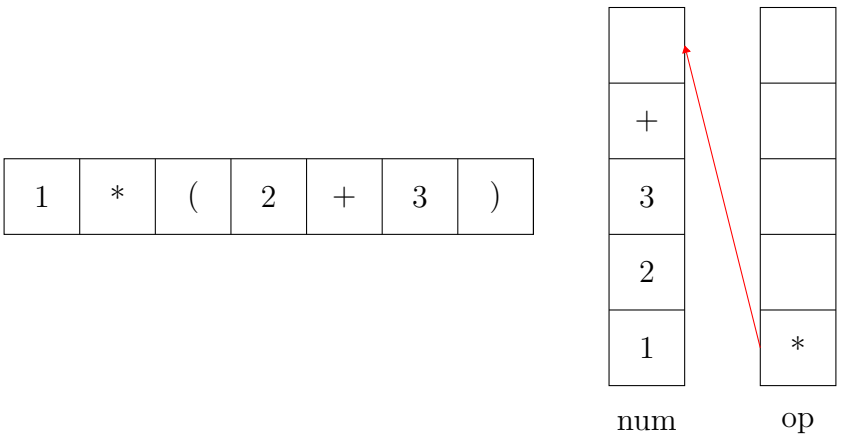

1 * (2 + 3)
(
1
*
num
op

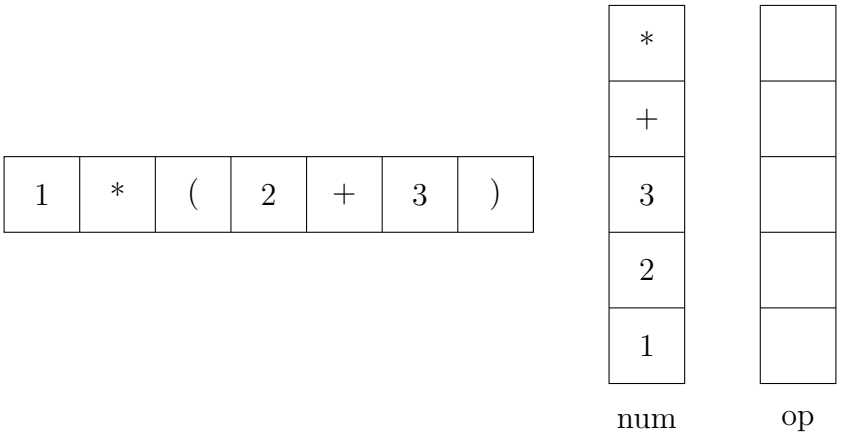

1 * (2 + 3)
2
1
num
(
*
op


1 * (2 + 3)
2
1
num
+
(
*
op


1 * (2 + 3)
3
2
1
num
+
(
*
op







表达式求值

```
def priority(op):
    """
        运算符的优先级
        乘除法优先级高于加减法
```

```
5           Args:
6               op (str): 运算符
7           Returns:
8               (int): 优先级
9       """
10      if op == "*" or op == "/":
11          return 2
12      elif op == "+" or op == "-":
13          return 1
14      else:
15          return 0
16
17  def infix_to_postfix(exp):
18      """
19          中缀表达式转换后缀表达式
20          转换后的后缀表达式操作数之前带空格
21          Args:
22              exp (str): 中缀表达式
23          Returns:
24              (str): 后缀表达式
25      """
26      postfix = ""    # 保存生成的后缀表达式
27      s = stack.Stack()
28
29      number = ""
30      for ch in exp:
31          # 如果是数字，保存每一位数字
32          if ch.isdigit():
33              number += ch
34              continue
35
36          # 如果读取一个完整数字，直接输出
37          if len(number) > 0:
38              postfix += number + " "
39              number = ""
40
41          # 空格忽略
```

```
42          if ch == " ":
43              continue
44
45          # 如果是运算符，并且空栈，则直接入栈
46          if s.is_empty():
47              s.push(ch)
48          # 如果遇到左括号，将其放入栈中
49          elif ch == "(":
50              s.push(ch)
51          # 如果遇到右括号，则一直出栈并输出，直到遇到左括号为止
52          # 注意，左括号只出栈并不输出
53          elif ch == ")":
54              while s.peek() != "(":
55                  postfix += s.pop() + " "
56              s.pop()
57          # 如果遇到任何其它的运算符，如果为栈为空，则直接入栈
58          # 否则从栈中出栈元素并输出，直到遇到优先级更低的元素（或为空）
59          # 在出栈完这些元素后，再将当前遇到的运算符入栈
60          # 只有遇到右括号的情况下才将左括号出栈
61          else:
62              while not s.is_empty()
63                      and priority(ch) <= priority(s.peek()):
64                  postfix += s.pop() + " "
65              s.push(ch)
66
67      # 如果读取一个完整数字，直接输出
68      if len(number) > 0:
69          postfix += number + " "
70          number = ""
71
72      while not s.is_empty():
73          postfix += s.pop() + " "
74
75      return postfix.rstrip()
76
77  def calculate(postfix):
78      """
```

```
    表达式求值
    Args:
        postfix (str): 后缀表达式
    Returns:
        (int): 表达式结果
    """
    s = stack.Stack()

    tokens = postfix.split()
    for token in tokens:
        # 数字则入栈
        try:
            s.push(int(token))
        # 运算符则出栈2次，将计算结果入栈
        except ValueError:
            num2 = s.pop()
            num1 = s.pop()
            if token == '+':
                s.push(num1 + num2)
            elif token == '-':
                s.push(num1 - num2)
            elif token == '*':
                s.push(num1 * num2)
            elif token == '/':
                s.push(int(num1 / num2))
    return s.pop()
```

Chapter 5 队列

5.1 队列

5.1.1 队列（Queue）

队列是一种运算受限的线性数据结构，不同于栈的先进后出（FILO），队列中的元素只能先进先出（FIFO, First In First Out）。

队列的出口端叫作队头（front），队列的入口端叫作队尾（rear）。队列只允许在队尾进行入队（enqueue），在队头进行出队（dequeue）。

与栈类似，队列既可以用数组来实现，也可以用链表来实现。其中用数组实现时，为了入队操作的方便，把队尾位置规定为最后入队元素的下一个位置。

5.1.2 入列（enqueue）

入队就是把新元素放入队列中，只允许在队尾的位置放入元素，新元素的下一个位置将会成为新的队尾。入队操作的时间复杂度是 $O(1)$。

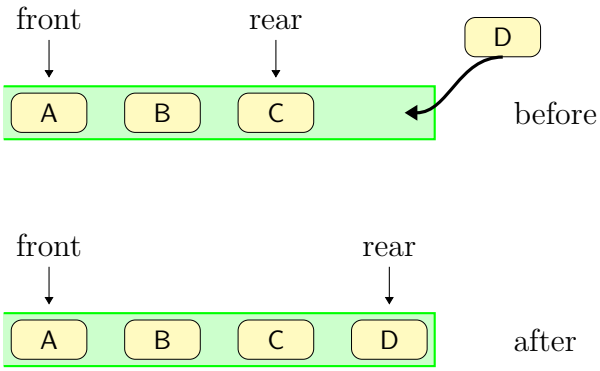


图 5.1: 入队

5.1.3 出队（dequeue）

出队就是把元素移出队列，只允许在队头一侧移出元素，出队元素的后一个元素将成为新的队头。出队操作的时间复杂度是 $O(1)$。

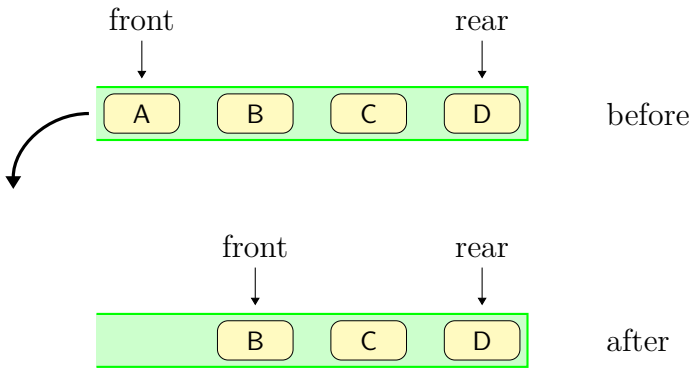


图 5.2: 出队

5.2 循环队列

5.2.1 循环队列（Circular Queue）

如果不断出队，队头左边的空间就失去了作用，那队列的容量就会变得越来越小。

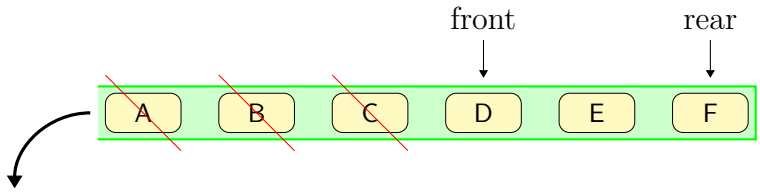


图 5.3: 队列存在的问题

用数组实现的队列可以采用循环队列的方式来维持队列容量的恒定。为充分利用空间，克服假溢出的现象，在数组不做扩容的情况下，将队列想象为一个首尾相接的圆环，可以利用已出队元素留下的空间，让队尾指针重新指回数组的首位。这样一来整个队列的元素就循环起来了。

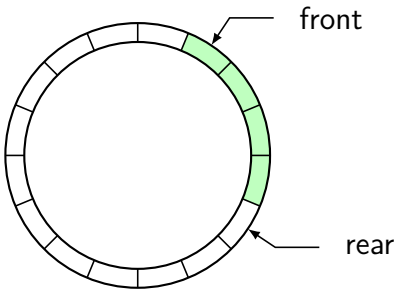


图 5.4: 循环队列

在物理存储上，队尾的位置也可以在队头之前。当再有元素入队时，将其放入数组的首位，队尾指针继续后移即可。队头和队尾互相追赶，这个追赶的过程就是入队的出队的过程。

如果队尾追上队头说明队列满了，如果队头追上队尾说明队列为空。循环队列并非真正地把数组弯曲，利用求余操作就能使队头和队尾指针不会跑出数组的范围，逻辑上实现了弯曲的效果。

假设数组的最大容量为 MAX：

- 入队时队尾指针后移：(rear + 1) % MAX
- 出队时队头指针后移：(front + 1) % MAX
- 判断队满：(rear + 1) % MAX == front
- 判断队空：front == rear

需要注意的是，队尾指针指向的位置永远空出一位，所以队列最大容量比数组长度小 1。

入队

```
1 void enqueue(Queue *queue, dataType val) {
2     queue->data[queue->rear] = val;
3     queue->rear = (queue->rear + 1) % queue->max;
4 }
```

出队

```
1 dataType dequeue(Queue *queue) {
2     dataType ret = queue->data[queue->front];
3     queue->front = (queue->front + 1) % queue->max;
4     return ret;
5 }
```

5.3 栈实现队列

5.3.1 栈实现队列

栈是特性是 FILO，而队列是 FIFO，因此可以使用两个栈来实现队列的效果。

可以将一个栈当作输入栈，用于 push() 数据，另一个栈当作输出栈，用于 pop() 和 peek() 数据。每次 pop() 或 peek() 时，若输出栈为空则将输入栈的全部数据依次弹出并压入输出栈，这样输出栈从栈顶往栈底的顺序就是队列从队首往队尾的顺序。

用两个栈来实现队列的情况在生活中也经常出现。去医院挂号等待，等待的时候把病历给护士，护士面前放了两堆病历。等着无聊就看护士是怎么管理病历的。发现一个堆是倒着放的，一个堆是正着放的。新过来的病人把病历给她时，她就把病历倒着放到第一堆，有病人看病结束后，从第二堆的开头翻一个新的病历，然后叫病人。如果第二堆没了话，就直接把第一堆翻过来放到第二推上面。

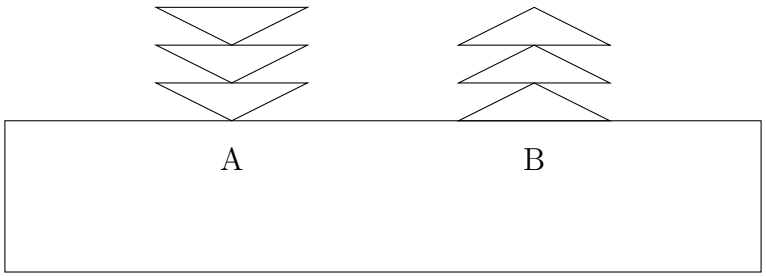


图 5.5: 双栈实现队列

双栈实现队列

```
1  class Queue:
2      def __init__(self):
3          self.in_stack = list()
4          self.out_stack = list()
5
6      def is_empty(self):
7          return not self.in_stack and not self.out_stack
8
```

```
9       def enqueue(self, data):
10          self.in_stack.append(data)
11
12      def dequeue(self):
13          if not self.out_stack:
14              while self.in_stack:
15                  self.out_stack.append(self.in_stack.pop())
16          return self.out_stack.pop()
```

用双栈实现的队列 push() 的时间复杂度为 $O(1)$，pop() 和 peek() 为均摊 $O(1)$，因为对于每个元素，至多入栈和出栈各两次。

5.4 队列实现栈

5.4.1 双队列实现栈

为了满足栈的特性，在使用队列实现栈时，应满足队列前端的元素是最后入栈的元素。可以使用两个队列实现栈的操作，其中 queue1 用于存储栈内的元素，queue2 作为入栈操作的辅助队列。

入栈操作时，首先将元素入队到 queue2，然后将 queue1 的全部元素依次出队并入队到 queue2，此时 queue2 的前端的元素即为新入栈的元素，再将 queue1 和 queue2 互换，则 queue1 的元素即为栈内的元素，queue1 的前端和后端分别对应栈顶和栈底。

由于 queue1 用于存储栈内的元素，判断栈是否为空时，只需要判断 queue1 是否为空即可。

5.4.2 单队列实现栈

在两个队列实现栈的方法中，其中一个队列的作用相当于临时变量。因此只使用一个队列就能实现栈了。

入栈操作时，首先获得入栈前的元素个数 n，然后将元素入队到队列，再将队列中的前 n 个元素（即除了新入栈的元素之外的全部元素）依次出队并入队到队列，此时队列的前端的元素即为新入栈的元素，且队列的前端和后端分别对应栈顶和栈底。

单队列实现栈

```
1 import collections
2
3 class Stack:
4     def __init__(self):
5         self.queue = collections.deque()
6
```

```
7       def is_empty(self):
8           return not self.queue
9
10      def push(self, data):
11          n = len(self.queue)
12          self.queue.append(data)
13          for _ in range(n):
14              self.queue.append(self.queue.popleft())
15
16      def pop(self):
17          return self.queue.popleft()
18
19      def peek(self):
20          return self.queue[0]
```

5.5 双端队列

5.5.1 双端队列（Deque, Double Ended Queue）

双端队列是一种同时具有队列和栈的性质的数据结构，双端队列可以从其两端插入和删除元素。

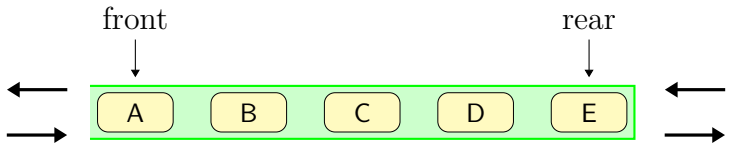


图 5.6: 双端队列

双端队列

```
class Deque:
    def __init__(self):
        self.data = []

    def is_empty(self):
        return not self.data

    def add_front(self, val):
        self.data.insert(0, val)

    def add_rear(self, val):
        self.data.append(val)

    def remove_front(self):
        if not self.is_empty():
            return self.data.pop(0)

    def remove_rear(self):
        if not self.is_empty():
            return self.data.pop()

    def get_front(self):
```

```
23          if not self.is_empty():
24              return self.data[0]
25
26      def get_rear(self):
27          if not self.is_empty():
28              return self.data[len(self.data)-1]
```

Chapter 6 哈希表

6.1 哈希表

6.1.1 哈希表（Hash Table）

例如开发一个学生管理系统，需要有通过输入学号快速查出对应学生的姓名的功能。这里不必每次都去查询数据库，而可以在内存中建立一个缓存表，这样做可以提高查询效率。

学号	姓名
001121	张三
002123	李四
002931	王五
003278	赵六

表 6.1: 学生名单

再例如需要统计一本英文书里某些单词出现的频率，就需要遍历整本书的内容，把这些单词出现的次数记录在内存中。

单词	出现次数
this	108
and	56
are	79
by	46

表 6.2: 词频统计

因为这些需要，一个重要的数据结构诞生了，这个数据结构就是哈希表。哈希表也称散列表，哈希表提供了键（key）和值（value）的映射关系，只要给出一个 key，就可以高效地查找到它所匹配的 value。

哈希表的时间复杂度几乎是常量 $O(1)$，即查找时间与问题规模无关。

哈希表的两项基本工作：

1. 计算位置：构造哈希函数确定关键字的存储位置。
2. 解决冲突：应用某种策略解决多个关键字位置相同的问题。

6.2 哈希函数

6.2.1 哈希函数（Hash Function）

哈希的基本思想是将键 key 通过一个确定的函数，计算出对应的函数值 value 作为数据对象的存储地址，这个函数就是哈希函数。

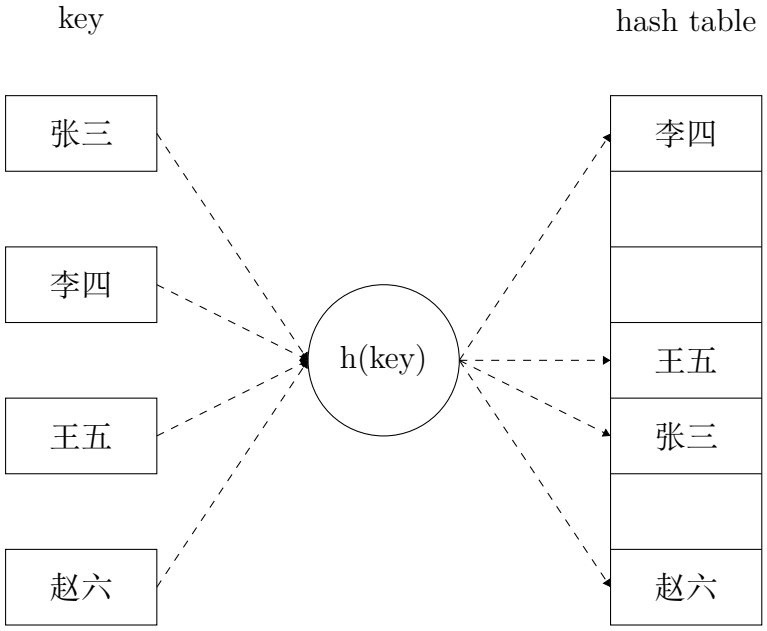


图 6.1: 哈希函数

哈希表本质上也是一个数组，可是数组只能根据下标来访问，而哈希表的 key 则是以字符串类型为主的。

在不同的语言中，哈希函数的实现方式是不一样的。假设需要存储整型变量，转化为数组的下标就不难实现了。最简单的转化方式就是按照数组长度进行取模运算。

一个好的哈希函数应该考虑两个因素：

1. 计算简单，以便提高转换速度。
2. 关键字对应的地址空间分布均匀，以尽量减少冲突。

6.2.2 数字关键字的哈希函数构造方法

对于数字类型的关键字，哈希函数有以下几种常用的构造方法：

直接定址法

取关键字的某个线性函数值为散列地址。

$$h(key) = a * key + b$$

例如根据出生年份计算人口数量 h(key) = key - 1990：

地址	出生年份	人数
0	1990	1285 万
1	1991	1281 万
2	1992	1280 万
...	...	...
10	2000	1250 万
...	...	...
21	2011	1180 万

表 6.3: 直接定址法

除留余数法

哈希函数为 h(key) = key % p，p 一般取素数。

例如 h(key) = key % 17：

地址	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
关键字	34	18	2	20			23	7	42		27	11		30		15	

表 6.4: 除留余数法

数字分析法

分析数字关键字在各位上的变化情况，取比较随机的位作为散列地址。

例如取 11 位手机号码的后 4 位作为地址 h(key) = int(key + 7)。

再例如取 18 位身份证号码中变化较为随机的位数：

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**
3	3	0	1	0	6	1	9	9	0	1	0	0	8	0	4	1	9
省		市		区		年				月		日		辖			校验

表 6.5: 数字分析法

折叠法

把关键字分割成位数相同的几个部分，然后叠加。

例如将整数 56793542 每三位进行分割：

$$\begin{array}{rr} & 542 \\ & 793 \\ + & 056 \\ \hline = & 1319 \end{array}$$

$$h(56793542) = 319$$

平方取中法

计算关键字的平方，取中间几位。

例如整数 56793542：

$$\begin{array}{rr} & 56793542 \\ * & 56793542 \\ \hline = & 3225506412905764 \end{array}$$

$$h(56793542) = 641$$

6.2.3 字符串关键字的哈希函数构造方法

对于字符串类型的关键字，哈希函数有以下几种常用的构造方法：

ASCII 码加和法

$$h(key) = \left(\sum key[i]\right) \ mod \ N$$

但是对于某些字符串会导致严重冲突，例如：a3、b2、c1 或 eat、tea 等。

移位法

取前 3 个字符移位。

$$h(key) = \left(key[0] \times 27^2 + key[1] \times 27 | key[2]\right) \ mod \ N$$

对于一些字符串仍然会冲突，例如 string、strong、street、structure 等。

一个有效的改进是涉及关键字中所有 n 个字符：

$$h(key) = \left(\sum_{i=0}^{n-1} key[n-i-1] \times 32^i\right) \ mod \ N$$

哈希函数

快速计算 $h(abcde) = a * 32^4 + b * 32^3 + c * 32^2 + d * 32 + e$

```
int hash(char *key, int tableSize) {
    int h = 0;          // hash value
    int i = 0;
    while(key[i] != '\0') {
        h = (h << 5) + key[i];
        i++;
    }
    return h % tableSize;
}
```

凯撒加密

```
/**
```

```
2  * @brief  凯撒加密
3  * @note   加密算法: ciphertext[i] = (plaintext[i] + Key) % 128
4  * @param  plaintext: 明文
5  * @retval 密文
6  */
7  char* encrypt(char *plaintext) {
8      int n = strlen(plaintext);
9      char *ciphertext = (char *)malloc((n + 1) * sizeof(char));
10     for(int i = 0; i < n; i++) {
11         ciphertext[i] = (plaintext[i] + KEY) % 128;
12     }
13     ciphertext[n] = '\0';
14     return ciphertext;
15 }
16
17 /**
18 * @brief  凯撒解密
19 * @note   解密算法: plaintext[i] = (ciphertext[i] - key + 128) % 128
20 * @param  ciphertext: 密文
21 * @retval 明文
22 */
23 char* decrypt(char *ciphertext) {
24     int n = strlen(ciphertext);
25     char *plaintext = (char *)malloc((n + 1) * sizeof(char));
26     for(int i = 0; i < n; i++) {
27         plaintext[i] = (ciphertext[i] - KEY + 128) % 128;
28     }
29     plaintext[n] = '\0';
30     return plaintext;
31 }
```

6.3 冲突处理

6.3.1 装填因子（Load Factor）

假设哈希表空间大小为 m，填入表中元素个数是 n，则称 $\alpha = n/m$ 为哈希表的装填因子。

当哈希表元素太多，即装填因子 α 太大时，查找效率会下降。实用最大装填因子一般取 $0.5 \leq \alpha \leq 0.85$。当装填因子过大时，解决的方法是加倍扩大哈希表，这个过程叫作再散列（rehashing）。

再散列的过程需要遍历原哈希表，把所有的关键字重新散列到新数组中。为什么需要重新散列呢？因为长度扩大以后，散列的规则也随之改变。经过扩容，原本拥挤的哈希表重新变得稀疏，原有的关键字也重新得到了尽可能均匀的分配。

装填因子也是影响产生哈希冲突的因素之一。当不同的关键字可能会映射到同一个散列地址上，就导致了哈希冲突（collision），即 $h(key_i) = h(key_j)$，$key_i \neq key_j$，因此需要某种冲突解决策略。

例如有 11 个数据对象的集合 {18, 23, 11, 20, 2, 7, 27, 30, 42, 15, 34, 35}，哈希表的大小为 17，哈希函数选择 h(key) = key % size。

地址	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
关键字	34	18	2	20			23	7	42		27	11		30		15	

在插入最后一个关键字 35 之前，都没有产生任何冲突。但是 h(35) = 1，位置已有对象，就导致了冲突。

常用的处理冲突的思路有两种：

1. 开放地址法（open addressing）：一旦产生了冲突，就按某种规则去寻找另一空地址。开放地址法主要有线性探测法、平方探测法（二次探测法）和双散列法。

2. 分离链接法：将相应位置上有冲突的所有关键字存储在同一个单链表中。

6.3.2 线性探测法（Linear Probing）

当产生冲突时，以增量序列 1, 2, 3, ..., n - 1 循环试探下一个存储地址。

例如序列 {47, 7, 29, 11, 9, 84, 54, 20, 30}，哈希表表长为 13，哈希函数 h(key) = key % 11，用线性探测法处理冲突。

key	**47**	**7**	**29**	**11**	**9**	**84**	**54**	**20**	**30**
h(key)	3	7	7	0	9	7	10	9	8
冲突次数	0	0	1	0	0	3	1	3	6

地址	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**Δ**
插入 47				47										0
插入 7				47				7						0
插入 29				47				7	29					1
插入 11	11			47				7	29					0
插入 9	11			47				7	29	9				0
插入 84	11			47				7	29	9	84			3
插入 54	11			47				7	29	9	84	54		1
插入 20	11			47				7	29	9	84	54	20	3
插入 30	11	30		47				7	29	9	84	54	20	6

表 6.6: 线性探测法

线性探测法的缺陷在于容易出现聚集现象。

6.3.3 平方探测法（Quadratic Probing）

平方探测法也称为二次探测法，以增量序列 $1^2, -1^2, 2^2, -2^2, \ldots, q^2, -q^2$ $(q \leq \lfloor N/2 \rfloor)$ 循环试探下一个存储地址。

例如序列 {47, 7, 29, 11, 9, 84, 54, 20, 30}，哈希表表长为 11，哈希函数 h(key) = key % 11，用平方探测法处理冲突。

key	47	7	29	11	9	84	54	20	30
h(key)	3	7	7	0	9	7	10	9	8
冲突次数	0	0	1	0	0	2	0	3	3

地址	0	1	2	3	4	5	6	7	8	9	10	Δ
插入 47				47								0
插入 7				47				7				0
插入 29				47				7	29			1
插入 11	11			47				7	29			0
插入 9	11			47				7	29	9		0
插入 84	11			47			84	7	29	9		-1
插入 54	11			47			84	7	29	9	54	0
插入 20	11		20	47			84	7	29	9	54	4
插入 30	11	30	20	47			84	7	29	9	54	4

表 6.7: 平方探测法

但是只要还有空间，平方探测法就一定能找到空闲位置吗?

例如对于以下哈希表，插入关键字 11，哈希函数 h(key) = key % 5，用平方探测法处理冲突。

下标	0	1	2	3	4
key	5	6	7		

表 6.8: 平方探测法存在的问题

对关键字 11 进行平方探测的结果一直在下标 0 和 2 之间波动，永远无法达到其它空的位置。

但是有定理证明，如果哈希表长度是满足 $4k+3$ $(k \in Z^+)$ 形式的素数时，平方探测法就可以探查到整个哈希表空间。

6.3.4 双散列探测法（Double Hashing）

设定另一个哈希函数 $h_2(key)$，探测序列为 $h_2(key)$，$2h_2(key)$，$3h_2(key)$，...。

探测序列应该保证所有的散列存储单元都应该能够被探测到，选择以下形式有良好的效果：

$$h_2(key) = p - (key \ \% \ p) \ \ (p < N \wedge p, N \in \text{素数})$$

6.3.5 分离链接法

分离链接法也称拉链法、链地址法，将相应位置上有冲突的所有关键字存储在同一个单链表中。

例如关键字序列为 {47, 7, 29, 11, 16, 92, 22, 8, 3, 50, 37, 89, 94, 21}，哈希函数 h(key) = key % 11，用分离链接法处理冲突。

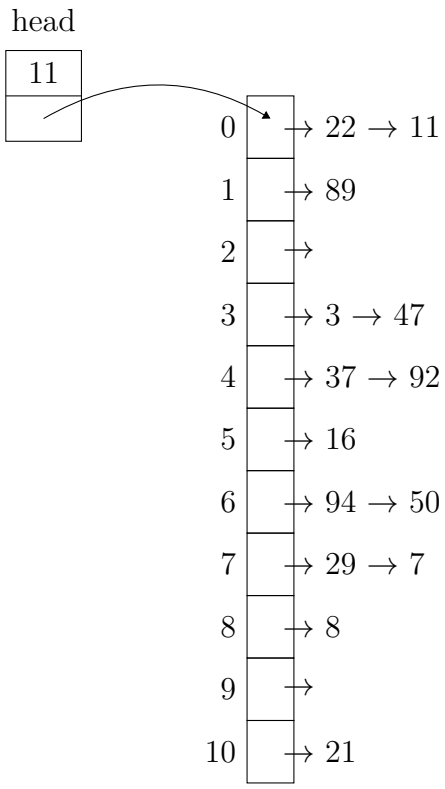


图 6.2: 分离链接法

6.4 性能分析

6.4.1 性能分析

哈希表的平均查找长度（ASL, Average Search Length）用来度量哈希表查找效率。关键字的比较次数，取决于产生冲突的多少。影响产生冲突多少有三个因素：

1. 哈希函数是否均匀
2. 处理冲突的方法
3. 哈希表的装填因子 α

合理的最大装填因子 α 应该不超过 0.85，选择合适的哈希函数可以使哈希表的查找效率期望是常数 $O(1)$，它几乎与关键字的空间大小 n 无关。这是以较小的 α 为前提，因此哈希表是一个以空间换时间的结构。

哈希表的存储对关键字是随机的，因此哈希表不便于顺序查找、范围查找、最大值/最小值查找等操作。

Chapter 7 分治法

7.1 分治法

7.1.1 分治法（Divide and Conquer）

分治策略是将原问题分解为 k 个子问题，并对 k 个子问题分别求解。如果子问题的规模仍然不够小，则再划分为 k 个子问题，如此递归的进行下去，直到问题规模足够小，很容易求出其解为止。

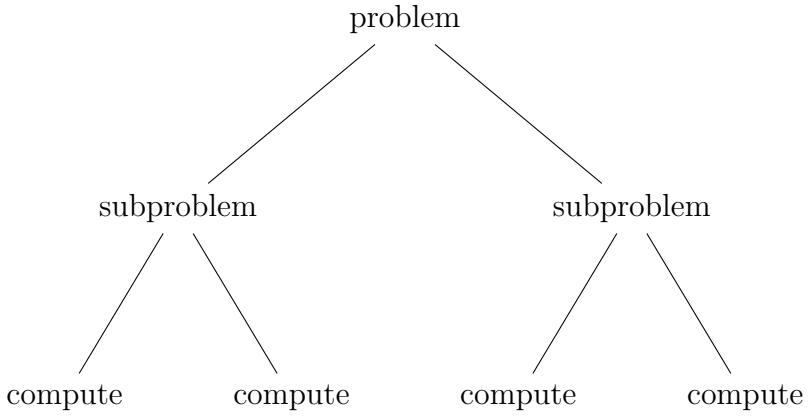


图 7.1: 分治法

将求出的小规模的问题的解合并为一个更大规模的问题的解，自底向上逐步求出原来问题的解。

分治法的适用条件有以下四点：

1. 该问题的规模缩小到一定的程度就可以容易地解决。
2. 该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质。
3. 利用该问题分解出的子问题的解可以合并为该问题的解。

4. 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子问题（如果各子问题是不独立的，则分治法要做许多不必要的工作，重复地解公共的子问题，此时虽然也可用分治法，但一般用动态规划较好）。

人们在大量事件中发现，在用分治法设计算法时，最好使子问题的规模大致相同。即将一个问题分成大小相等的 k 个子问题的处理方法是行之有效的。这种使子问题规模大致相等的做法是出自一种平衡子问题的思想，它几乎总是比子问题规模不等的做法要好。

7.1.2 二分搜索

一个装有 16 个硬币的袋子，16 个硬币中有一个是伪造的，并且那个伪造的硬币比真的硬币要轻一些，要求找出这个伪造的硬币。只提供一台可用来比较两组硬币重量的仪器，利用这台仪器，可以知道两组硬币的重量是否相同。

算法思想是将 16 个硬币等分成 A、B 两份，将 A 放仪器的一边，B 放另一边，如果 A 袋轻，则表明伪币在 A，解子问题 A 即可，否则解子问题 B。

二分搜索每执行一次循环，待搜索数组的大小减少一半，在最坏情况下，循环被执行了 $O(logn)$ 次，循环体内运算需要 $O(1)$ 时间。因此，整个算法在最坏情况下的计算时间复杂性为 $O(logn)$。

7.2 大整数加法

7.2.1 大整数加法

如果有两个很大的整数，如何求出它们的和?

这还不简单? 直接用 long 类型存储，在程序里相加不就行了?

C/C++ 中的 int 类型能表示的范围是 $-2^{31} \sim 2^{31}-1$，unsigned 类型能表示的范围是 $0 \sim 2^{32}-1$，所以 int 和 unsigned 类型变量都不能保存超过 10 位的整数。

有时需要参与运算的数可能会远远不止 10 位，例如计算 100! 的精确值。即便使用能表示很大数值范围的 double 变量，但是由于 double 变量只有 64 位，精度也不足以表示一个超过 100 位的整数。我们称这种基本数据类型无法表示的整数为大整数。

在小学的时候，老师教我们用列竖式的方式计算两个整数的和。

$$\begin{array}{rr} & 426709752318 \\ + & 95481253129 \\ \hline = & 522191005447 \end{array}$$

不仅仅是人脑，对于计算机来说同样如此。对于大整数，我们无法一步到位直接算出结果，所以不得不把计算拆解成一个一个子步骤。

可是，既然大整数已经超出了 long 类型的范围，我们如何来存储这样的整数呢?

存放大整数最简单的方法就是使用数组，可以用数组的每一个元素存储整数的每一个数位。如果给定大整数的最长位数是 n，那么按位计算的时间复杂度是 $O(n)$。

大整数加法

```
1 def big_int_add(num1, num2):
```

```
    # 其中一个数为0，直接返回另一个数
    if num1 == "0":
        return num2
    elif num2 == "0":
        return num1

    # 计算两个数中较长的整数位数
    max_len = max(len(num1), len(num2))
    # 让位数较短的整数前面补0对齐
    num1 = '0' * (max_len - len(num1)) + num1
    num2 = '0' * (max_len - len(num2)) + num2

    result = ""          # 结果
    carry = 0            # 保存进位
    # 从右往左逐位相加
    for i in range(max_len - 1, -1, -1):
        s = int(num1[i]) + int(num2[i]) + carry
        result = str(s % 10) + result
        carry = s // 10

    # 判断最高位是否有进位
    if carry > 0:
        result = str(carry) + result

    # 去除结果前面多余的0
    i = 0
    while result[i] == '0':
        i += 1
    return result[i:]
```

这种思路其实还存在一个可优化的地方。我们之前是把大整数按照每一个十进制数位来拆分，比如较大整数的长度有 50 位，那么需要创建一个 51 位的数组，数组的每个元素存储其中一位。

真的有必要把原整数拆分得那么细吗？显然不需要，只需要拆分到可以被直接计算的程度就够了。int 类型的取值范围是 $-2147483648 \sim 2147483647$，最多有 10

位整数。为了防止溢出，可以把大整数的每 9 位作为数组的一个元素，进行加法运算。如此一来，占用空间和运算次数，都被压缩了 9 倍。

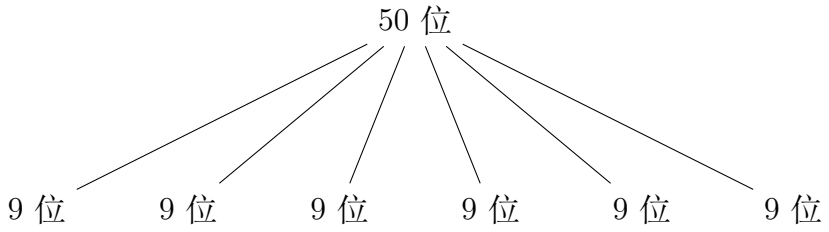


图 7.2: 大整数加法优化

在 Java 中，工具类 BigInteger 和 BidDecimal 的底层实现也是把大整数拆分成数组进行运算，与此处的思路大体类似。

Chapter 8 排序算法

8.1 排序算法

8.1.1 排序算法

应用到排序的常见比比皆是，例如当开发一个学生管理系统时需要按照学号从小到大进行排序，当开发一个电商平台时需要把同类商品按价格从低到高进行排序，当开发一款游戏时需要按照游戏得分从多到少进行排序。

根据时间复杂度的不同，主流的排序算法可以分为三类：

1. $O(n^2)$：冒泡排序、选择排序、插入排序
2. $O(nlogn)$：归并排序、快速排序、堆排序
3. $O(n)$：计数排序、桶排序、基数排序

在算法界还存在着更多五花八门的排序，它们有些基于传统排序变形而来，有些则是脑洞大开，如鸡尾酒排序、猴子排序、睡眠排序等。

例如睡眠排序，对于待排序数组中的每一个元素，都开启一个线程，元素值是多少，就让线程睡多少毫秒。当这些线程陆续醒来的时候，睡得少的线程线性来，睡得多的线程后醒来。睡眠排序虽然挺有意思，但是没有任何实际价值。启动大量线程的资源消耗姑且不说，数值接近的元素也未必能按顺序输出，而且一旦遇到很大的元素，线程睡眠时间可能超过一个月。

8.1.2 稳定性

排序算法还可以根据其稳定性，划分为稳定排序和不稳定排序：

- 稳定排序：值相同的元素在排序后仍然保持着排序前的顺序。
- 不稳定排序：值相同的元素在排序后打乱了排序前的顺序。

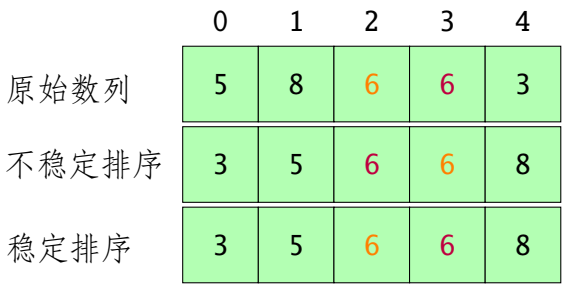


图 8.1: 排序稳定性

8.2 冒泡排序

8.2.1 冒泡排序（Bubble Sort）

冒泡排序是最基础的交换排序。冒泡排序之所以叫冒泡排序，正是因为这种排序算法的每一个元素都可以像小气泡一样，根据自身大小，一点一点向着数组的一侧移动。

按照冒泡排序的思想，要把相邻的元素两两比较，当一个元素大于右侧相邻元素时，交换它们的位置；当一个元素小于或等于右侧相邻元素时，位置不变。

例如一个有 8 个数字组成的无序序列，进行升序排序。

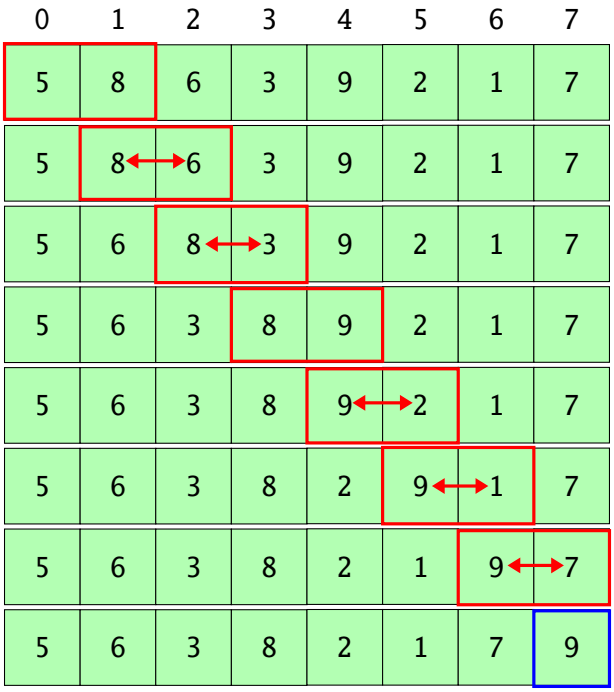


图 8.2: 冒泡排序第 1 轮

这样一来，元素 9 作为数列中最大的元素，就像是汽水里的小气泡一样，浮到了最右侧。这时，冒泡排序的第 1 轮就结束了。数列最右侧元素 9 的位置可以认为是一个有序区域，有序区域目前只有 1 个元素。

接着进行第 2 轮排序：

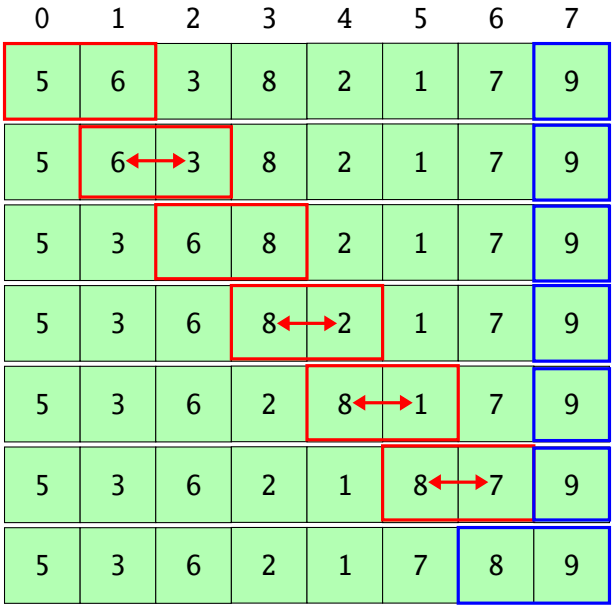


图 8.3: 冒泡排序第 2 轮

第 2 轮排序结束后，数列右侧的有序区有了 2 个元素。

根据相同的方法，完成剩下的排序：

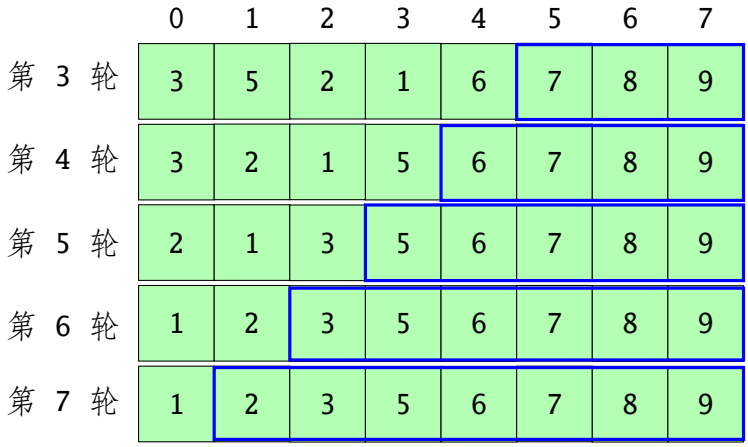


图 8.4: 冒泡排序第 3 ～ 7 轮

8.2.2 算法分析

冒泡排序是一种稳定排序，值相等的元素并不会打乱原本的顺序。由于该排序算法的每一轮都要遍历所有元素，总共遍历 n - 1 轮。

时间复杂度	空间复杂度	稳定性	设计思想
$O(n^2)$	$O(1)$	稳定	贪心法

表 8.1: 冒泡排序算法分析

冒泡排序

```
1  void bubbleSort(int *arr, int n) {
2      for(int i = 0; i < n; i++) {
3          for(int j = 0; j < n-i-1; j++) {
4              if(arr[j] > arr[j+1]) {
5                  swap(&arr[j], &arr[j+1]);
6              }
7          }
8      }
9  }
```

逆序对

假设数组有 n 个元素，如果 A[i] > A[j], i < j，那么 A[i] 和 A[j] 就被称为逆序对 (inversion)。

```
1  int countInversions(int *arr, int n) {
2      int cnt = 0;        // 逆序对数
3      for(int i = 0; i < n-1; i++) {
4          for(int j = i+1; j < n; j++) {
5              if(arr[i] > arr[j]) {
6                  cnt++;
7              }
8          }
9      }
10     return cnt;
11 }
```

8.2.3 冒泡排序第一次优化

常规的冒泡排序需要进行 n - 1 轮循环，即使在中途数组已经有序，但是还是会继续剩下的循环。例如当数组是 {2, 1, 3, 4, 5} 时，在经过一轮排序后已经变为有序状态，再进行多余的循环就会浪费时间。

为了解决这个问题，可以在每一轮循环中设置一个标志。如果该轮循环中有元素发生过交换，那么就有必要进行下一轮循环。如果没有发生过交换，说明当前数组已经完成排序。

冒泡排序第一次优化

```
void bubbleSortOptimize1(int *arr, int n) {
    for(int i = 0; i < n - 1; i++) {
        bool isSorted = false;  // 标记是否发生交换
        for(int j = 0; j < n - i - 1; j++) {
            if(arr[j] > arr[j+1]) {
                swap(&arr[j], &arr[j+1]);
                isSorted = true;    // 发生交换
            }
        }
        // 该轮未发生交换，已经有序
        if(!isSorted) {
            return;
        }
    }
}
```

8.2.4 冒泡排序第二次优化

在经过一次优化后，算法还存在一个问题，例如数组 {2, 3, 1, 4, 5, 6} 在经过一轮交换后变为 {2, 1, 3, 4, 5, 6}，但是在下一轮时后面有很多次比较都是多余的，因为并没有产生交换操作。

为了解决这个问题，可以再设置一个标志位，用于记录当前轮所交换的最后一个元素的下标。在下一轮排序中，只需比较到该标志位即可，因此之后的元素在上一轮中没有交换过，在这一轮中也不可能交换了。

冒泡排序第二次优化

```
void bubbleSortOptimize2(int *arr, int n) {
    int len = n - 1;          // 内层循环执行次数
    for(int i = 0; i < n - 1; i++) {
        bool flag = false;  // 标记是否发生交换
        int last = 0;         // 标记最后一次发生交换的位置
        for(int j = 0; j < len; j++) {
            if(arr[j] > arr[j+1]) {
                swap(&arr[j], &arr[j+1]);
                flag = true;    // 发生交换
                last = j;
            }
        }
        // 该轮未发生交换，已经有序
        if(!flag) {
            return;
        }
        len = last;           // 最后一次发生交换的位置
    }
}
```

8.3 选择排序

8.3.1 选择排序（Selection Sort）

有了冒泡排序为什么还要发明选择排序？冒泡排序有个很大的弊端，就是元素交换次数太多了。

想象一个场景，假设你是一名体育老师，正在指挥一群小学生按照个头从矮到高的顺序排队。采用冒泡排序的方法需要频繁交换相邻学生的位置，同学们心里恐怕会想：“这体育老师是不是有毛病啊?”

在程序运行的世界里，虽然计算机并不会产生什么负面情绪，但是频繁的数组元素交换意味着更多的内存读写操作，严重影响了代码运行效率。

有一个简单的办法，就是每一次找到个子最矮的学生，直接交换到队伍的前面。

例如一个有 8 个数字组成的无序序列，进行升序排序。

	0	1	2	3	4	5	6	7
原数组	5	8	6	3	9	2	1	7
第 1 轮	1	8	6	3	9	2	5	7
第 2 轮	1	2	6	3	9	8	5	7
第 3 轮	1	2	3	6	9	8	5	7
第 4 轮	1	2	3	5	9	8	6	7
第 5 轮	1	2	3	5	6	8	9	7
第 6 轮	1	2	3	5	6	7	9	8
第 7 轮	1	2	3	5	6	7	8	9

图 8.5: 选择排序

8.3.2 算法分析

算法每一轮选出最小值，再交换到左侧的时间复杂度是 $O(n)$，一共迭代 n - 1 轮，总的时间复杂度是 $O(n^2)$。

由于算法所做的是原地排序，并没有利用额外的数据结构，所以空间复杂度是 $O(1)$。

时间复杂度	空间复杂度	稳定性	设计思想
$O(n^2)$	$O(1)$	不稳定	减治法

表 8.2: 选择排序算法分析

选择排序

```
void selectionSort(int *arr, int n) {
    for(int i = 0; i < n-1; i++) {
        int minIndex = i;
        for(int j = i+1; j < n; j++) {
            if(arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        if(i != minIndex) {
            swap(&arr[i], &arr[minIndex]);
        }
    }
}
```

8.3.3 选择排序优化

选择排序的整体思想是在一个序列当中选出一个最小的元素，和第一个元素交换，然后在剩下的找最小的，和第二个元素交换。这样最终就可以得到一个有序序列。但是为了更加高效，可以每次选择出一个最小值和一个最大值，分别放在序列的最左和最右边。

选择排序优化

```
void selectionSortOptimize(int *arr, int n) {
    int left = 0;
    int right = n - 1;
    while(left < right) {
        int min = left;
        int max = right;
        for(int i = left; i <= right; i++) {
            if(arr[i] < arr[min]) {
                min = i;
            }
            if(arr[i] > arr[max]) {
                max = i;
            }
        }
        swap(&arr[max], &arr[right]);
        // 考虑特殊情况，最小值在最右位置
        if(min == right) {
            min = max;
        }
        swap(&arr[min], &arr[left]);
        left++;
        right--;
    }
}
```

8.4 插入排序

8.4.1 插入排序（Insertion Sort）

如何对扑克牌进行排序呢？例如现在手上有红桃 6, 7, 9, 10 这四张牌，已经处于升序排序状态。这时候抓到了一张红桃 8，如何让手上的五张牌重新变成升序呢？

使用冒泡排序？选择排序？恐怕正常人打牌的时候都不会那么做。最自然最简单的方式，是在已经有序的四张牌中找到红桃 8 应该插入的位置，也就是 7 和 9 之间，把红桃 8 插入进去。

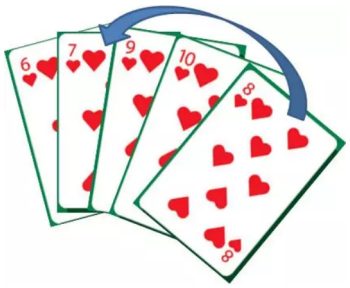


图 8.6: 理牌

例如一个有 8 个数字组成的无序序列，进行升序排序。

	0	1	2	3	4	5	6	7
原数组	5	8	6	3	9	2	1	7
第 1 轮	5	8	6	3	9	2	1	7
第 2 轮	5	6	8	3	9	2	1	7
第 3 轮	3	5	6	8	9	2	1	7
第 4 轮	3	5	6	8	9	2	1	7
第 5 轮	2	3	5	6	8	9	1	7
第 6 轮	1	2	3	5	6	8	9	7
第 7 轮	1	2	3	5	6	7	8	9

图 8.7: 插入排序

8.4.2 算法分析

插入排序要进行 n - 1 轮，每一轮在最坏情况下的比较复制次数分别是 1 次、2 次、3 次、4 次... 一直到 n - 1 次，所以最坏时间复杂度是 $O(n^2)$。

至于空间复杂度，由于插入排序是在原地进行排序，并没有引入额外的数据结构，所以空间复杂度是 $O(1)$。

时间复杂度	空间复杂度	稳定性	设计思想
$O(n^2)$	$O(1)$	稳定	减治法

表 8.3: 插入排序算法分析

插入排序

```
1  void insertionSort(int *arr, int n) {
2      for(int i = 1; i < n; i++) {
3          int temp = arr[i];
4          int j = i - 1;
5          while(j >= 0 && temp < arr[j]) {
6              arr[j+1] = arr[j];
7              j--;
8          }
9          arr[j+1] = temp;
10     }
11 }
```

8.4.3 折半插入排序（Binary Insertion Sort）

折半插入排序是对插入排序的改进，其过程就是不断依次将元素插入前面已经排好序的序列中，在寻找插入点时采用了折半查找。

折半插入排序

```
1  void binaryInsertionSort(int *arr, int n) {
```

```
    for(int i = 1; i < n; i++) {
        int temp = arr[i];
        int start = 0;
        int end = i - 1;
        while(start <= end) {
            int mid = start + (end - start) / 2;
            if(arr[mid] > temp) {
                end = mid - 1;
            } else {
                start = mid + 1;
            }
        }
        int j;
        for(j = i - 1; j > end; j--) {
            arr[j+1] = arr[j];
        }
        arr[j+1] = temp;
    }
}
```

8.5 鸡尾酒排序

8.5.1 鸡尾酒排序（Cocktail Sort）

对一个无序数组 {2, 3, 4, 5, 6, 7, 8, 1} 进行升序排序，如果按照冒泡排序，步骤如下：

图 8.8: 冒泡排序

从 2 到 8 已经是有序了，只有元素 1 的位置不对，却还要进行 7 轮排序，这也太憋屈了吧！

鸡尾酒排序正是用于解决这种问题的。鸡尾酒排序又叫快乐小时排序，它基于冒泡排序做了一点小小的优化。

鸡尾酒排序的第一轮与冒泡排序相同，从左向右比较和交换。

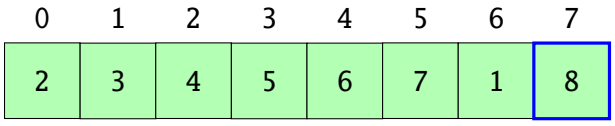


图 8.9: 鸡尾酒排序第 1 轮

第二轮开始不一样了，反过来向右向左比较和交换。

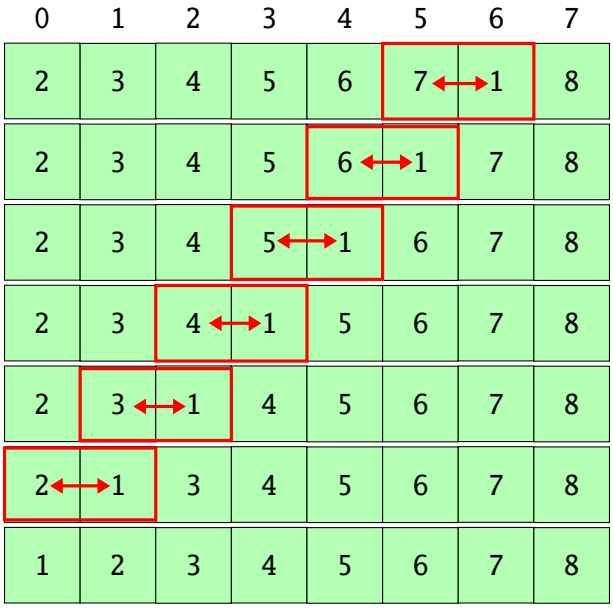


图 8.10: 鸡尾酒排序第 2 轮

第二轮结束后，此时虽然已经有序，但是算法并没有结束。

第三轮再次从左向右比较和交换，在此过程中，没有元素发生交换，证明已经有序，排序结束。

鸡尾酒排序的过程就像钟摆一样，左右来回比较和交换。本来冒泡要用 8 轮排序的场景，鸡尾酒用 3 轮就解决了。

鸡尾酒排序的优点是能够在大部分元素已经有序的情况下，减少排序的回合数；而缺点也很明显，就是代码量几乎扩大了一倍。

鸡尾酒排序

```
1  void cocktailSort1(int *arr, int n) {
2      for(int i = 0; i < n / 2; i++) {
3          // 从左向右
4          bool isSorted = true;    // 标记当前轮是否有序
5          for(int j = i; j < n - i - 1; j++) {
```

```
            if(arr[j] > arr[j+1]) {
                swap(&arr[j], &arr[j+1]);
                isSorted = false;   // 发生交换
            }
        }
        if(isSorted) {
            break;
        }

        // 从右向左
        isSorted = true;
        for(int j = n - i - 1; j > i; j--) {
            if(arr[j] < arr[j-1]) {
                swap(&arr[j], &arr[j-1]);
                isSorted = false;
            }
        }
        if(isSorted) {
            break;
        }
    }
}
```

8.5.2 鸡尾酒排序优化

类似冒泡排序的优化方法，鸡尾酒排序也可以记录每一轮排序后最后一次元素交换的位置。但是对于双向的鸡尾酒排序而言，需要设置两个边界值。

鸡尾酒排序优化

```
void cocktailSort2(int *arr, int n) {
    int lastLeft = 0;           // 左侧最后一次交换位置
    int lastRight = 0;          // 右侧最后一次交换位置
    int leftBorder = 0;         // 无序区左边界
    int rightBorder = n - 1;    // 无序区右边界

```

```
    for(int i = 0; i < n / 2; i++) {
        // 从左向右
        bool isSorted = true;   // 标记当前轮是否有序
        for(int j = leftBorder; j < rightBorder; j++) {
            if(arr[j] > arr[j+1]) {
                swap(&arr[j], &arr[j+1]);
                isSorted = false;   // 发生交换
                lastRight = j;
            }
        }
        if(isSorted) {
            break;
        }
        rightBorder = lastRight;

        // 从右向左
        isSorted = true;
        for(int j = n - i - 1; j > i; j--) {
            if(arr[j] < arr[j-1]) {
                swap(&arr[j], &arr[j-1]);
                isSorted = false;
                lastLeft = j;
            }
        }
        if(isSorted) {
            break;
        }
        leftBorder = lastLeft;
    }
}
```

8.6 归并排序

8.6.1 归并排序（Merge Sort）

归并排序算法采用分治法：

1. 分解：将序列每次折半划分。
2. 合并：将划分后的序列两两按序合并。

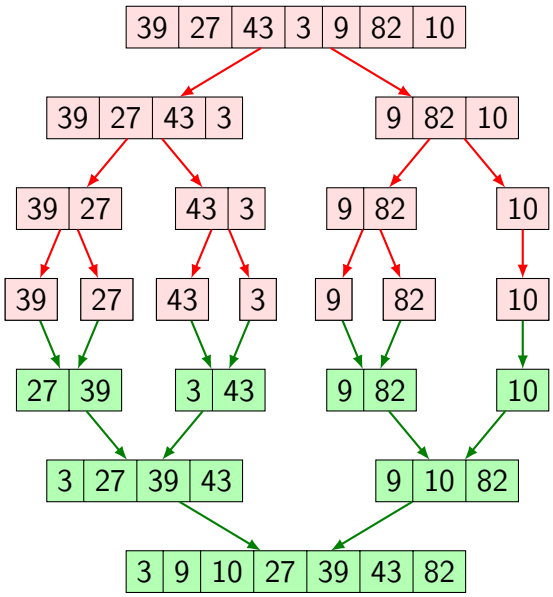


图 8.11: 归并排序

8.6.2 算法分析

归并排序每次将数组折半对分，一共分了 $logn$ 次，每一层进行合并操作的运算量是 n，所以时间复杂度为 $O(nlogn)$。归并排序的速度仅次于快速排序。

时间复杂度	空间复杂度	稳定性	设计思想
$O(nlogn)$	$O(n)$	稳定	分治法

表 8.4: 归并排序算法分析

归并排序

```
1  void merge(int *arr, int start, int mid, int end, int *temp) {
2      int i = start;
3      int j = mid + 1;
4      int k = 0;
5
6      while(i <= mid && j <= end) {
7          if(arr[i] <= arr[j]) {
8              temp[k++] = arr[i++];
9          } else {
10             temp[k++] = arr[j++];
11         }
12     }
13
14     while(i <= mid) {
15         temp[k++] = arr[i++];
16     }
17     while(j <= end) {
18         temp[k++] = arr[j++];
19     }
20
21     for(int i = 0; i < k; i++) {
22         arr[start+i] = temp[i];
23     }
24 }
25
26 void mergeSort(int *arr, int start, int end, int *temp) {
27     if(start < end) {
28         int mid = start + (end - start) / 2;
29         mergeSort(arr, start, mid, temp);
30         mergeSort(arr, mid+1, end, temp);
31         merge(arr, start, mid, end, temp);
32     }
33 }
```

8.7 快速排序

8.7.1 快速排序（Quick Sort）

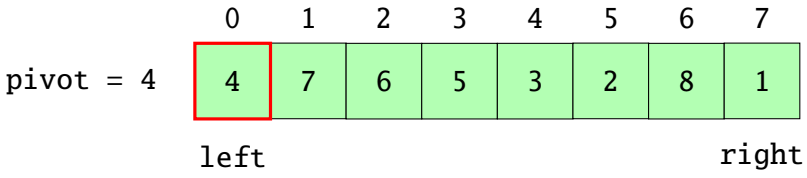
快速排序是很重要的算法，与傅里叶变换等算法并称二十世纪十大算法。

快速排序之所以快，是因为它使用了分治法。快速排序在每一轮挑选一个基准（pivot）元素，并让其它比它小的元素移动到数列一边，比它大的元素移动到数列的另一边，从而把数列拆解成了两个部分。

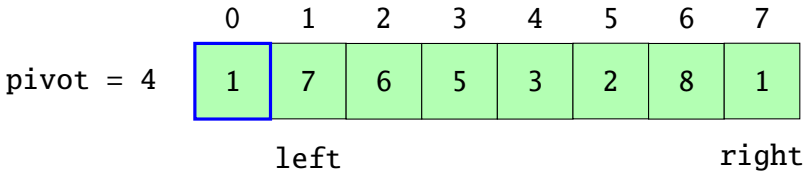
选择基准元素最简单的方式是选择数列的第一个元素。这种选择在绝大多数情况下是没有问题的，但是如果对一个原本逆序的数列进行升序排序，整个数列并没有被分成一半，每一轮仅仅确定了基准元素的位置。这种情况下数列第一个元素要么是最小值，要么是最大值，根本无法发挥分治法的优势。在这种极端情况下，快速排序需要进行 n 轮，时间复杂度退化成了 $O(n^2)$。
如何避免这种极端情况呢？可以不选择数列的第一个元素，而是随机选择一个元素作为基准元素。这样一来，即使是在数列完全逆序的情况下，也可以有效地将数列分成两部分。当然，即使是随机选择，每一次也有极小的几率选到数列的最大值或最小值，同样会对分治造成一定影响。

确定了基准值后，如何实现将小于基准的元素都移动到基准值一边，大于基准值的都移动到另一边呢？

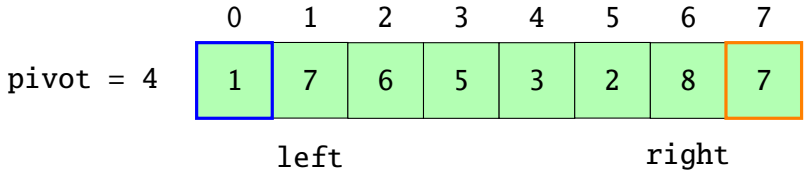
例如一个有 8 个数字组成的无序序列，进行升序排序。选定基准元素 pivot，设置两个指针 left 和 right，指向数列的最左和最右两个元素。



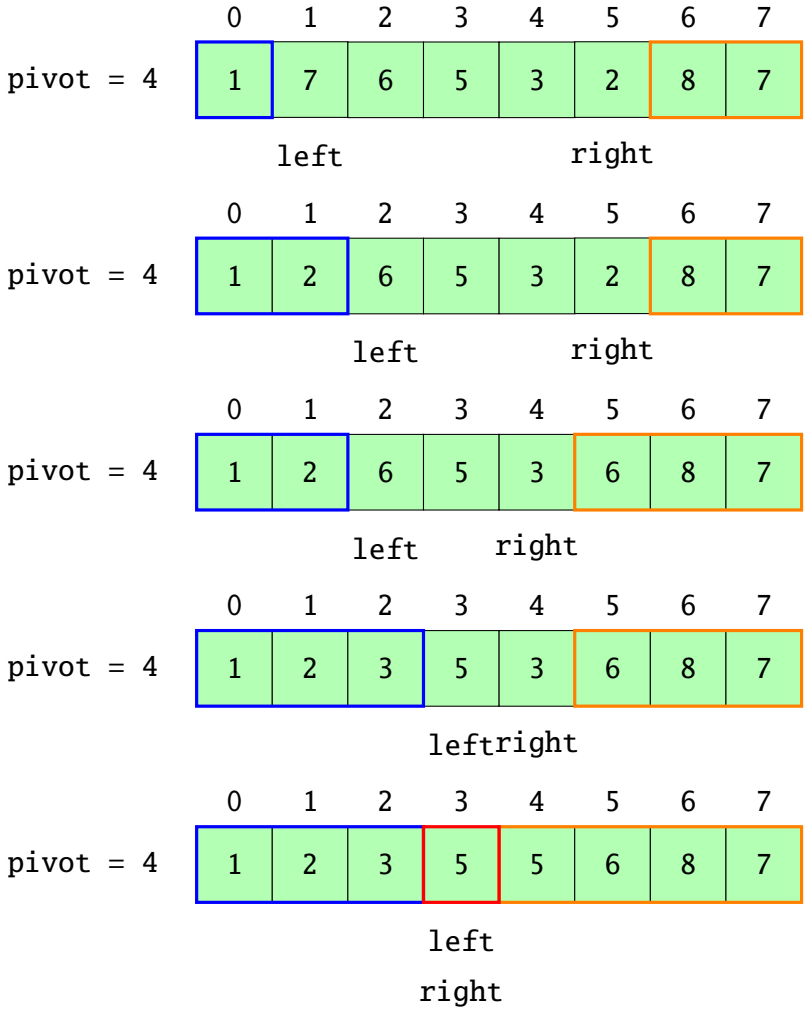
从 right 指针开始，把指针所指向的元素和基准元素做比较。如果比 pivot 大，则 right 指针向左移动；如果比 pivot 小，则把 right 所指向的元素填入 left 指针所指向的位置，同时 left 向右移动一位。



接着，切换到 left 指针进行比较，把指针所指向的元素和基准元素做比较。如果小于 pivot，则 left 指针向右移动；如果大于 pivot，则把 left 所指向的元素填入 right 指针所指向的位置，同时 right 向左移动一位。



重复之前的步骤继续排序：



当 left 和 right 指针重合在同一位置的时候，把之前的 pivot 元素的值填入该重合的位置。此时数列左边的元素都小于基准元素，数列右边的元素都大于基准元素。

8.7.2 算法分析

分治法的思想下，原数列在每一轮被拆分成两部分，每一部分在下一轮又被拆分成两部分，直到不可再分为止。这样平均情况下需要 $logn$ 轮，因此快速排序算法的平均时间复杂度是 $O(nlogn)$。

时间复杂度	空间复杂度	稳定性	设计思想
$O(nlogn) \sim O(n^2)$	$O(logn) \sim O(n)$	不稳定	分治法

表 8.5: 快速排序算法分析

快速排序

```
void quickSort(int *arr, int start, int end) {
    if(start < end) {
        int i = start;
        int j = end;
        int pivot = arr[start];

        while(i < j) {
            while(i < j && arr[j] > pivot) {
                j--;
            }
            if(i < j) {
                arr[i] = arr[j];
                i++;
            }
            while(i < j && arr[i] < pivot) {
                i++;
            }
            if(i < j) {
                arr[j] = arr[i];
                j--;
```

```
            }
        }
        arr[i] = pivot;
        quickSort(arr, start, i-1);
        quickSort(arr, i+1, end);
    }
}
```

8.8 计数排序

8.8.1 计数排序（Counting Sort）

基于比较的排序算法的最优下界为 $\Omega(nlogn)$。计数排序是一种不基于比较的排序算法，而是利用数组下标来确定元素的正确位置。

遍历数列，将每一个整数按照其值对号入座，对应数组下标的元素加 1。数组的每一个下标位置的值，代表了数列中对应整数出现的次数。有了这个统计结果，直接遍历数组，输出数组元素的下标值，元素的值是多少就输出多少次。

从功能角度，这个算法可以实现整数的排序，但是也存在一些问题。如果只以最大值来决定统计数组的长度并不严谨，例如数列 {95, 94, 91, 98, 99, 90, 99, 93, 91, 92}，这个数列的最大值是 99，但最小值是 90。如果创建长度为 100 的数组，前面的从 0 到 89 的空间位置都浪费了。

因此，不应再以数列的 max + 1 作为统计数组的长度，而是以数列 max - min + 1 作为统计数组的长度。同时，数列的最小值作为一个偏移量，用于统计数组的对号入座。

计数排序适用于一定范围的整数排序，在取值范围不是很大的情况下，它的性能甚至快过那些 $O(nlogn)$ 的排序算法。

计数排序

```
1  void countingSort(int *arr, int n) {
2      int max = arr[0];
3      int min = arr[0];
4      for(int i = 1; i < n; i++) {
5          if(arr[i] > max) {
6              max = arr[i];
7          }
8          if(arr[i] < min) {
```

```
            min = arr[i];
        }
    }

    int range = max - min + 1;
    int table[range];
    memset(table, 0, sizeof(table));

    for(int i = 0; i < n; i++) {
        table[arr[i] - min]++;
    }

    int cnt = 0;
    for(int i = 0; i < range; i++) {
        while(table[i]--) {
            arr[cnt++] = i + min;
        }
    }
}
```

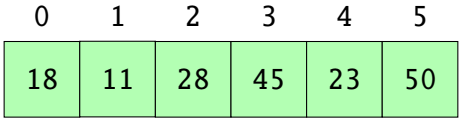
8.9 桶排序

8.9.1 桶排序（Bucket Sort）

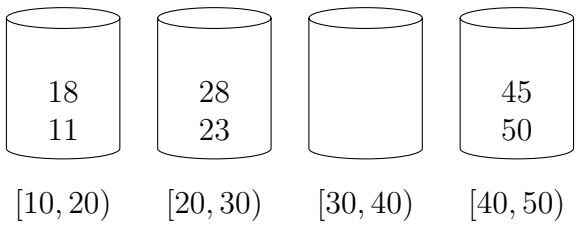
桶排序是计数排序的扩展版本。计数排序可以看成每个桶只存储相同元素，而桶排序每个桶存储一定范围的元素。

每一个桶代表一个区间范围，里面可以承载一个或多个元素。通过划分多个范围相同的区间，将每个子区间自排序，最后合并。桶排序需要尽量保证元素分散均匀，否则当所有数据集中在同一个桶中时，桶排序失效。

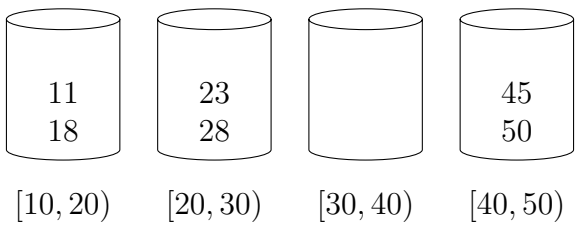
例如一个待排序的序列：



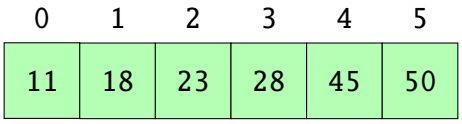
确定桶的个数与每个桶的取值范围，遍历待排序序列，将元素放入对应的桶中：



分别对每个桶中的元素进行排序：



将桶中的元素按顺序赋值到原始数组中：



创建桶的数量取决于数据的区间范围，一般创建桶的数量等于待排序的元素数量，每个桶的区间跨度为：

$$\frac{max - min}{buckets - 1}$$

桶排序

```java
public static void bucketSort(int[] arr) {
        int max = arr[0];
        int min = arr[0];
        for(int i = 1; i < arr.length; i++) {
            if(arr[i] > max) {
                max = arr[i];
            }
            if(arr[i] < min) {
                min = arr[i];
            }
        }
        int diff = max - min;

        // 创建桶
        int bucketNum = arr.length;
        List<LinkedList<Integer>> buckets
            = new ArrayList<>(bucketNum);
        for(int i = 0; i < bucketNum; i++) {
            buckets.add(new LinkedList<>());
        }

        // 遍历数组，把元素放入对应桶中
        for(int i = 0; i < arr.length; i++) {
            // 计算当前元素所放置的桶号
            int num = (arr[i]-min) / (diff / (bucketNum-1));
            buckets.get(num).add(arr[i]);
        }

        // 桶内排序
```

```
30          for(int i = 0; i < bucketNum; i++) {
31              Collections.sort(buckets.get(i));
32          }
33
34          // 取出元素
35          int cnt = 0;
36          for(LinkedList<Integer> list : buckets) {
37              for(int data : list) {
38                  arr[cnt++] = data;
39              }
40          }
41      }
```

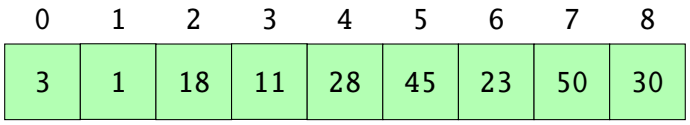
8.10 基数排序

8.10.1 基数排序（Radix Sort）

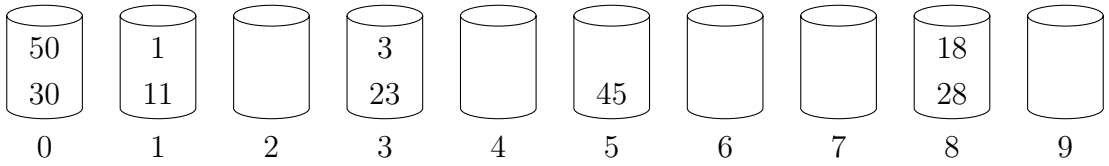
基数排序可以看作是桶排序的扩展，主要思想是将整数按位划分。

基数排序需要准备 10 个桶，分别代表 0 ～ 9，根据整数个位数字的数值将元素放入对应的桶中，之后按照输入赋值到原序列中，再依次对十位、百位等进行同样的操作。

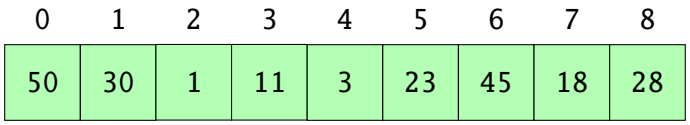
例如一个待排序的序列：



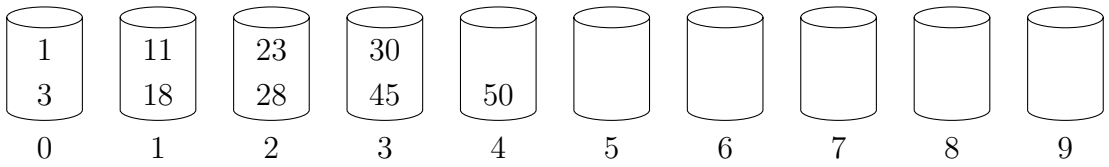
根据个位数值放入对应的桶中：



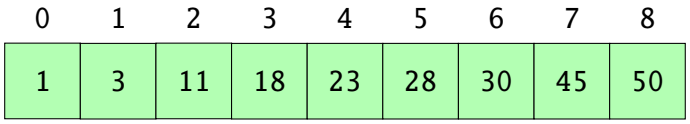
将桶中的元素按顺序赋值到原始数组中：



根据十位数值放入对应的桶中：



将桶中的元素按顺序赋值到原始数组中：



基数排序

```java
public static void radixSort(int[] arr) {
        int max = arr[0];
        for(int i = 1; i < arr.length; i++) {
            if(arr[i] > max) {
                max = arr[i];
            }
        }

        int bucketNum = 10;
        // 从个位开始
        for(int exp = 1; max / exp > 0; exp *= 10) {
            // 创建桶
            List<LinkedList<Integer>> buckets
                = new ArrayList<>(bucketNum);
            for(int i = 0; i < bucketNum; i++) {
                buckets.add(new LinkedList<>());
            }

            // 把元素放到对应桶中
            for(int data : arr) {
                int num = (data / exp) % 10;
                buckets.get(num).add(data);
            }

            // 按顺序取出元素
            int cnt = 0;
            for(LinkedList<Integer> bucket : buckets) {
                for(int data : bucket) {
                    arr[cnt++] = data;
                }
            }
        }
    }
```

8.11 珠排序

8.11.1 珠排序（Bead Sort）

珠排序的算法和算盘非常类似。算盘上有许多圆圆的珠子被串在细杆上，如果把算盘竖起来，算盘上的小珠子会在重力的作用下滑到算盘底部。其中有一个很神奇的细节，如果统计每一横排珠子的数量，下落之后每一排珠子数量正好是下落前珠子数量的升序排序。

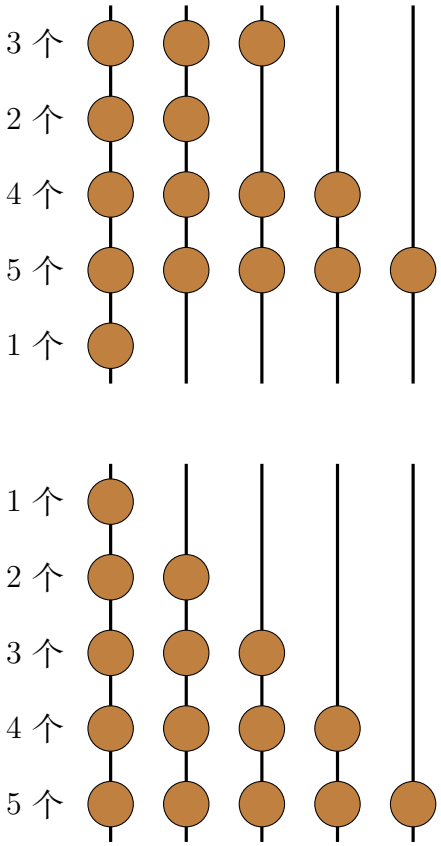


图 8.12: 珠子下落

通过模拟珠子下落的元素可以对一组正整数进行排序。使用二维数组来模拟算盘，有珠子的位置设为 1，没有珠子的位置设为 0。例如一个有 5 个数字组成的无序整型数组 {3, 2, 4, 5, 1} 就可以转化为以下的二维数组：

1	1	1	0	0
1	1	0	0	0
1	1	1	1	0
1	1	1	1	1
1	0	0	0	0

表 8.6: 模拟算盘

接下来模拟算盘珠子掉落的过程，让所有的元素 1 都落到二维数组的最底部。

1	0	0	0	0
1	1	0	0	0
1	1	1	0	0
1	1	1	1	0
1	1	1	1	1

最后把掉落后的算盘转换为一维有序数组 {1, 2, 3, 4, 5}。

8.12 猴子排序

8.12.1 猴子排序（Bogo Sort）

听说过“猴子和打字机”的理论吗?

无限猴子定理（Infinite Monkey Theorem）与薛定谔的猫、电车实验等并居十大思想实验，所谓思想实验即用想象力去进行，而在现实中基本无法去实现的实验。

无限猴子定理讲的是如果让一只猴子在打字机上胡乱打字，只要有无限的时间，总有一天可以恰好打出莎士比亚的著作。如果让无限只猴子在无限的空间、无限的时间里不停地敲打打字机，总有一天可以完整打出一本《哈姆雷特》，甚至是可以打出所有可能的文章。

图 8.13: 无限猴子定理

这个看似不可能的事情，却可以用现有数学原理被推导出来。但在现实中往往被认为是无法实现的，因为人们认为“无限”这个条件通常无法被满足。根据概率论证，即使可观测宇宙中充满了猴子一直不停地打字，能够打出一部《哈姆雷特》的概率仍然小于 $\frac{1}{10^{183900}}$。

无限猴子定理同样可以用在排序中。如果给数组随机排列顺序，每一次排列之后验证数组是否有序，只要次数足够多，总有一次数组刚好被随机成有序数组。可是要想真的随机出有序数列，恐怕要等到猴年马月了。

Chapter 9 树

9.1 树

9.1.1 树（Tree）

许多逻辑关系并不是简单的线性关系，在实际场景中，常常存在着一对多，甚至多对多的情况。树和图就是典型的非线性数据结构。

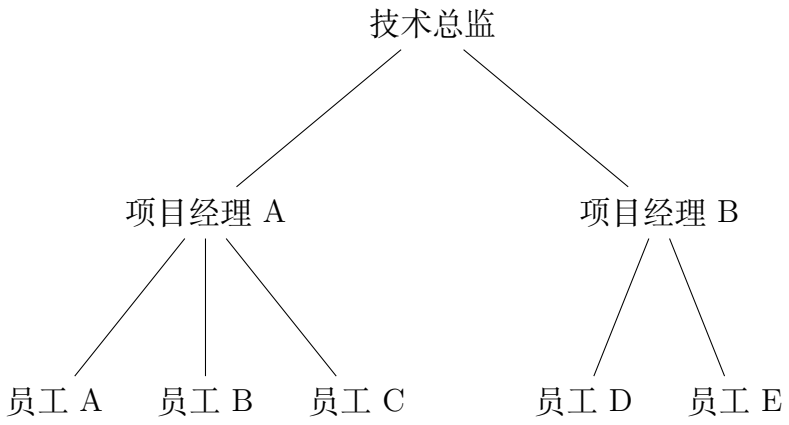


图 9.1: 职级关系

这些结构都像自然界中的树一样，从同一个根衍生出许多枝干，再从每一个枝干衍生出许多更小的枝干，最后衍生出更多的叶子。

树是由 n $(n \geq 0)$ 个有限节点组成的一个具有层次关系的集合，当 $n = 0$ 时称为空树。

在任意一个非空树中，有以下特点：

- 有且有且仅有一个特定的结点称为根（root）。
- 当 $n > 1$ 时，其余结点可分为 m $(m > 0)$ 个互不相交的有限集 $T_1, T_2, \ldots, T_m$，其中每一个集合本身又是一棵树，并且称为根的子树（subtree）。

9.1.2 树的术语

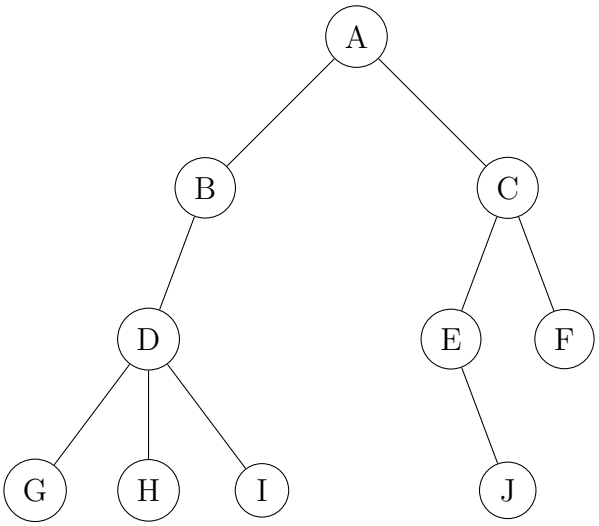


图 9.2: 树

- 根：没有父结点（parent）的结点。
- 内部结点（internal node）：至少有一个子结点（child）的结点。
- 外部结点（external node）/ 叶子结点（leaf node）：没有子结点的结点。
- 度（degree）：结点分支的个数。
- 路径（path）：从根结点到树中某结点经过的分支构成了路径。
- 祖先结点（ancestors）：包含父结点、父结点的父结点等。
- 子孙结点（descendants）：包含子结点、子结点的子结点等。
- 深度（depth）/ 高度（height）：最大层级数。

9.2 二叉树

9.2.1 二叉树（Binary Tree）

二叉树是树的一种特殊形式。二叉树的每个结点最多有两个孩子结点，即最多有 2 个，也可能只有 1 个，或者没有孩子结点。

二叉树结点的两个孩子结点，分别被称为左孩子（left child）和右孩子（right child）。这两个孩子结点的顺序是固定的，不能颠倒或混淆。

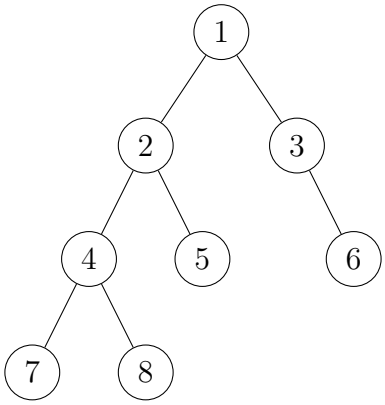


图 9.3: 二叉树

二叉树还有几种特殊的形式：

左斜树（left skew tree）/ 右斜树（right skew tree）

只有左子树或只有右子树的二叉树。

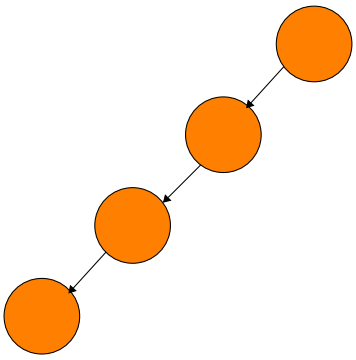


图 9.4: 左斜树

满二叉树（full binary tree）

所有非叶子结点都存在左右孩子，并且所有叶子结点都在同一层。

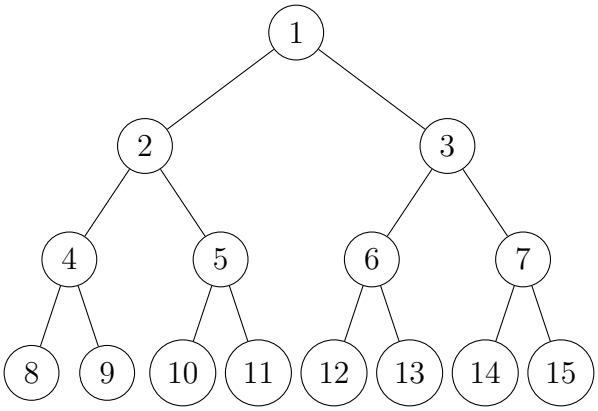


图 9.5: 满二叉树

完全二叉树

对于一个有 n 个结点的二叉树，按层级顺序编号，则所有结点的编号从 1 到 n，完全二叉树所有结点和同样深度的满二叉树的编号从 1 到 n 的结点位置相同。简单来说，就是除最后一层外，其它各层的结点数都达到最大，并且最后一层从右向左连续缺少若干个结点。

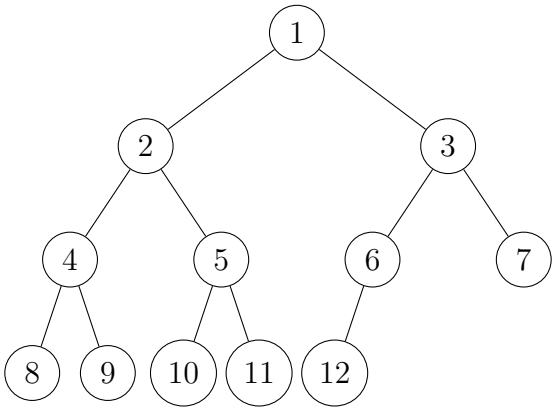


图 9.6: 完全二叉树

9.2.2 二叉树的存储结构

二叉树既可以通过链式存储，也可以使用数组存储：

链式存储结构

一个结点最多可以指向左右两个孩子结点，所以二叉树的每一个结点包含三个部分：

- 存储数据的数据域 data
- 指向左孩子的指针 left
- 指向右孩子的指针 right

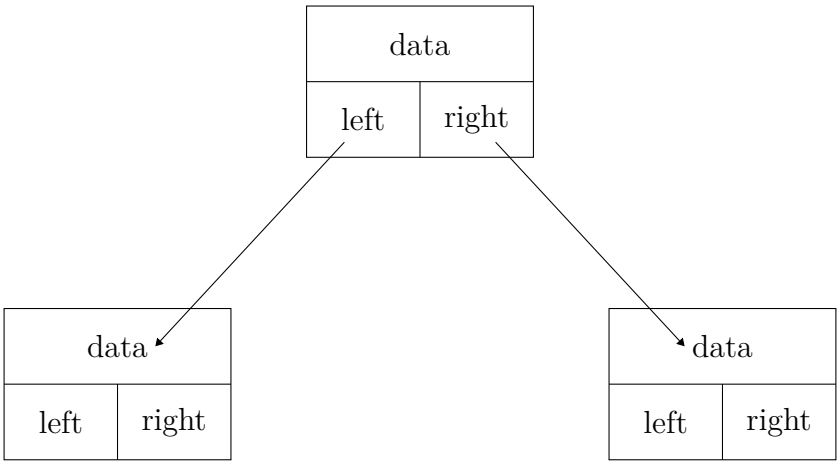
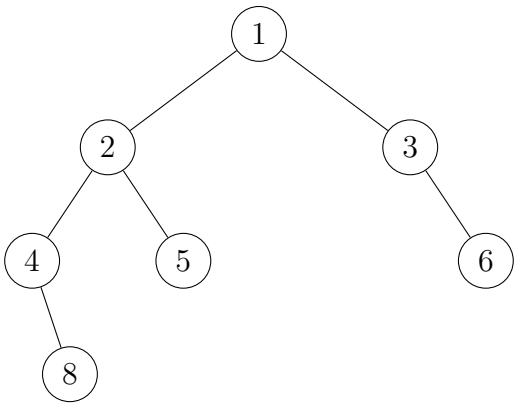


图 9.7: 链式存储结构

数组存储

按照层级顺序把二叉树的结点放到数组中对应的位置上。如果某一结点的左孩子或右孩子空缺，则数组的相应位置也空出来。



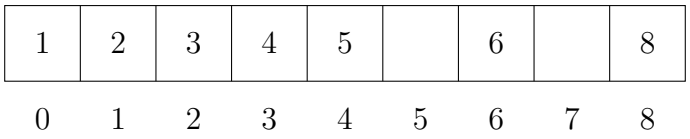


图 9.8: 数组存储

采用数组存储可以更方便地定位二叉树的孩子结点和父结点。假设一个父结点的下标是 parent，那么它的左孩子结点的下标就是 2 * parent + 1，右孩子结点的下标就是 2 * parent + 2。反过来，假设一个左孩子结点的下标是 leftChild，那么它的父结点的下标就是 (leftChild - 1) / 2。

但是，对于一个稀疏的二叉树来说，用数组表示法是非常浪费空间的。对于一种特殊的完全二叉树——二叉堆而言，就是使用数组进行存储的。

9.3 二叉树的遍历

9.3.1 二叉树的遍历

在计算机程序中，遍历（traversal）本身是一个线性操作，所以遍历同样具有线性结构的数组或链表是一件轻而易举的事情。

反观二叉树，是典型的非线性数据结构，遍历时需要把非线性关联的结点转化成一个线性的序列，以不同的方式来遍历，遍历出的序列顺序也不同。

二叉树的遍历方式分为 4 种：

1. 前序遍历（pre-order）：访问根结点，遍历左子树，遍历右子树。
2. 中序遍历（in-order）：遍历左子树，访问根结点，遍历右子树。
3. 后序遍历（post-order）：遍历左子树，遍历右子树，访问根结点。
4. 层次遍历（level-order）：按照从根结点到叶子结点的层次关系，一层一层横向遍历。

9.3.2 前序遍历

二叉树的前序遍历，首先访问根结点然后遍历左子树，最后遍历右子树。在遍历左、右子树时，仍然先访问根结点，然后遍历左子树，最后遍历右子树，如果结点为空则返回。

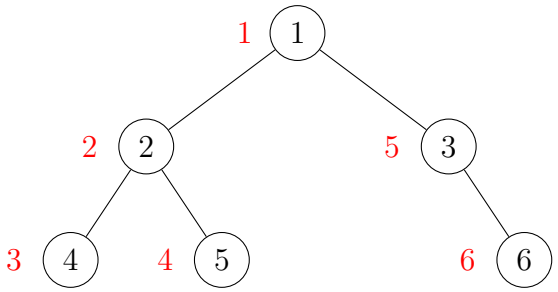


图 9.9: 前序遍历

前序遍历

```c
void preOrder(BST *root) {
    if(!root) {
        return;
    }
    printf("%d ", root->data);
    preOrder(root->left);
    preOrder(root->right);
}
```

9.3.3 中序遍历

二叉树的中序遍历，首先遍历左子树，然后访问根结点，最后遍历右子树，如果结点为空则返回。

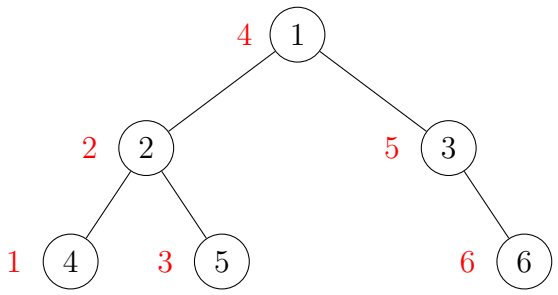


图 9.10: 中序遍历

中序遍历

```c
void inOrder(BST *root) {
    if(!root) {
        return;
    }
    inOrder(root->left);
    printf("%d ", root->data);
    inOrder(root->right);
}
```

9.3.4 后序遍历

二叉树的后序遍历，首先遍历左子树，然后遍历右子树，最后访问根结点，如果结点为空则返回。

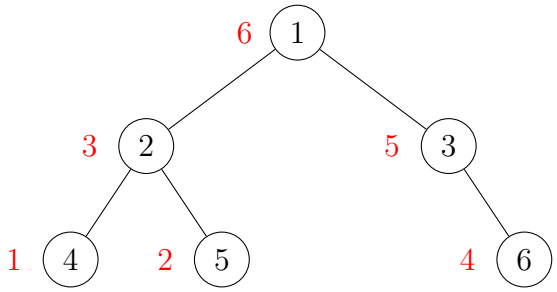


图 9.11: 后序遍历

后序遍历

```
1  void postOrder(BST *root) {
2      if(!root) {
3          return;
4      }
5      postOrder(root->left);
6      postOrder(root->right);
7      printf("%d ", root->data);
8  }
```

9.3.5 二叉树遍历非递归实现

绝大多数可以用递归解决的问题，其实都可以用栈来解决，因为递归和栈都有回溯的特性。

以二叉树的中序遍历为例。当遇到一个结点时，就把它入栈，并去遍历它的左子树。当左子树遍历结束后，从栈顶弹出这个结点并访问它，然后按其右指针再去中序遍历该结点的右子树。

中序遍历（非递归）

```
public void inOrderNonRecursive(BSTNode node) {
    Stack s = new Stack();
    while(node != null || !s.empty()) {
        // 一直向左并将沿途结点压入堆栈
        while(node != null) {
            s.push(node);
            node = node.left;
        }
        if(!s.empty()) {
            node = s.pop();                    //结点弹出堆栈
            System.out.println(node.data);      // 访问结点
            node = node.right;                 // 转向右子树
        }
    }
}
```

9.3.6 层次遍历

二叉树同一层次的结点之间是没有直接关联的，需要队列来辅助完成层序遍历。

层次遍历从根结点开始首先将根结点入队，然后开始循环执行以下操作直到队列为空：结点出队、访问该结点、其左右儿子入队。

层次遍历

```
public void levelOrder(BSTNode node) {
    if(node == null) {
        return;
    }

    Queue q = new Queue();
    q.enqueue(node);
    while(!q.empty()) {
        node = q.dequeue();
```

```
10          System.out.println(node.data);      // 访问结点
11          if(node.left != null) {
12              q.enqueue(node.left);
13          }
14          if(node.right != null) {
15              q.enqueue(node.right);
16          }
17      }
18  }
```

9.4 二叉搜索树

9.4.1 二叉搜索树（Binary Search Tree）

二叉搜索树，也称二叉查找树或二叉排序树，可以是一棵空树。

如果不为空树，那么二叉搜索树满足以下性质：

1. 非空左子树的所有结点的值小于其根结点的值。
2. 非空右子树的所有结点的值大于其根结点的值。
3. 左、右子树均是二叉搜索树。

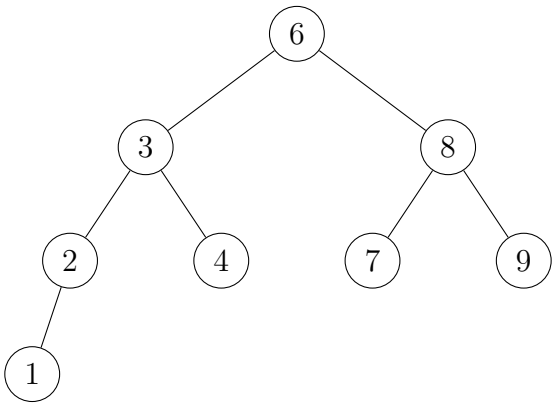


图 9.12: 二叉搜索树

9.4.2 查找结点

在二叉搜索树中查找一个元素从根结点开始，如果树为空，返回 NULL。

如果树不为空，则将根结点的值和被查找的 key 值进行比较：

1. 如果 key 值小于根结点的值，只需在左子树中继续查找。
2. 如果 key 值大于根结点的值，只需在右子树中继续查找。
3. 如果 key 值与根结点的值相等，查找成功。

查找结点（递归）

```
Node* search(Node *root, dataType val) {
    if(!root) {
        return NULL;
    }
    if(val == root->data) {
        return root;
    } else if(val < root->data) {
        return search(root->left, val);
    } else {
        return search(root->right, val);
    }
}
```

由于非递归函数的执行效率高，可将尾递归（在函数最后才使用递归返回）的函数改为迭代函数。

查找结点（迭代）

```
Node* search(Node *root, dataType val) {
    if(!root) {
        return NULL;
    }
    while(root) {
        if(root->data == val) {
            return root;
        } else if(val < root->data) {
            root = root->left;
        } else {
            root = root->right;
        }
    }
}
```

9.4.3 查找最小值和最大值

二叉搜索树中，最小值一定在树的最左分枝的叶子结点上，最大值一定在树的最右分枝的叶子结点上。

查找最小值（递归）

```
Node* findMin(Node *root) {
    if(!root) {
        return NULL;
    } else if(!root->left) {
        return root;
    } else {
        return findMin(root->left);      //沿左分枝继续查找
    }
}
```

查找最大值（迭代）

```
Node* findMax(Node *root) {
    if(!root) {
        return NULL;
    }
    while(root->right) {
        root = root->right;
    }
    return root;
}
```

9.4.4 插入结点

在二叉搜索树中插入结点与查找的算法相似，需要找到插入的位置并将新结点插入。

插入结点

```
BST* insert(BST *root, dataType val) {
    // 空树，插入结点设为树根
    if(!root) {
        return init(val);
    }
    if(val < root->data) {
        root->left = insert(root->left, val);
    } else {
        root->right = insert(root->right, val);
    }
    return root;
}
```

9.5 哈夫曼树

9.5.1 哈夫曼树（Huffman Tree）

树的每一个结点都可以拥有自己的权值（weight），假设二叉树有 n 个叶子结点，每个叶子结点都带有权值 w_k，从根结点到每个叶子结点的长度为 l_k，则树的带权路径长度（WPL, Weighted Path Length）为：

$$WPL = \sum_{k=1}^{n} w_k l_k$$

哈夫曼树是由麻省理工学院的哈夫曼博士于 1952 年发明的，哈夫曼树是在叶子结点和权重确定的情况下，带权路径长度最小的二叉树，也被称为最优二叉树。

例如，有五个叶子结点，它们的权值为 {1, 2, 3, 4, 5}，用此权值序列可以构造出形状不同的多个二叉树。

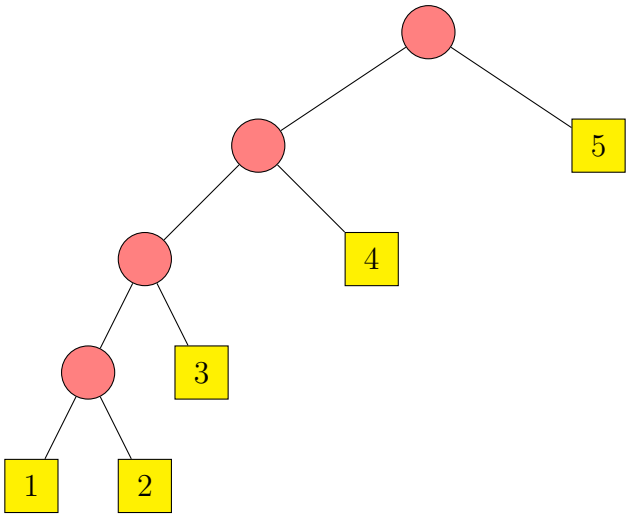


图 9.13: WPL = 5 * 1 + 4 * 2 + 3 * 3 + 2 * 4 + 1 * 4 = 34

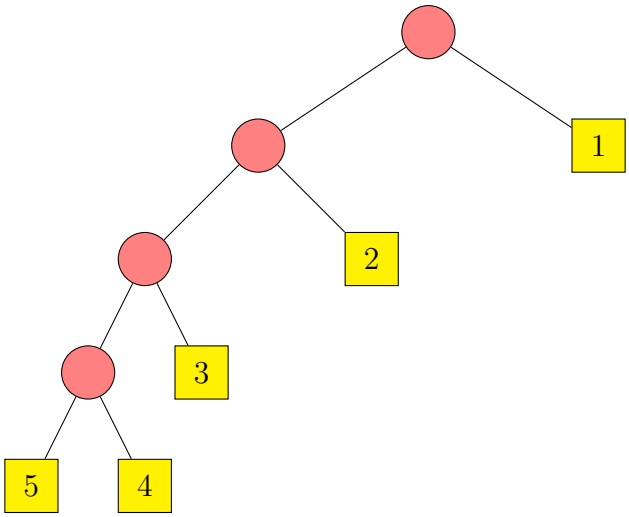


图 9.14: WPL = 1 * 1 + 2 * 2 + 3 * 3 + 4 * 4 + 5 * 4 = 50

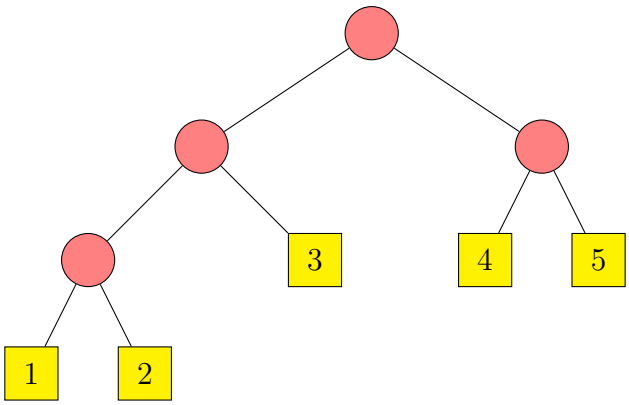


图 9.15: WPL = 3 * 2 + 4 * 2 + 5 * 2 + 1 * 3 + 2 * 3 = 33

怎样才能保证构建出的二叉树带权路径长度最小呢？原则上，应该让权重小的叶子结点远离树根，权重大的叶子结点靠近树根。需要注意的是，同样叶子结点所构成的哈夫曼树可能不止一棵。

9.5.2 哈夫曼树的构造

哈夫曼树的构造方法就是每次把权值最小的两棵二叉树合并。

例如有 6 个叶子结点，权重依次是 2、3、7、9、18、25。

第一步：把每一个叶子结点都当成一棵独立的树（只有根结点的树），这样就形成了一个森林。

第二步：从森林中移除权值最小的两个结点，生成父结点，父结点的权值是这两个结点权值之和，把父结点加入森林。重复该步骤，直到森林中只有一棵树为止。

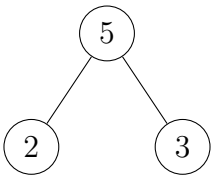


图 9.16: 合并 2 和 3

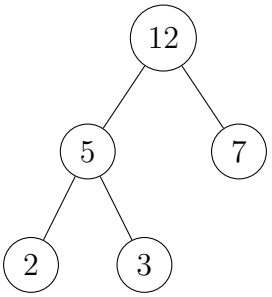


图 9.17: 合并 5 和 7

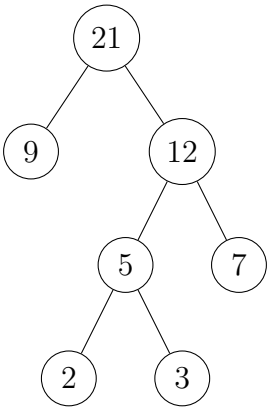


图 9.18: 合并 9 和 12

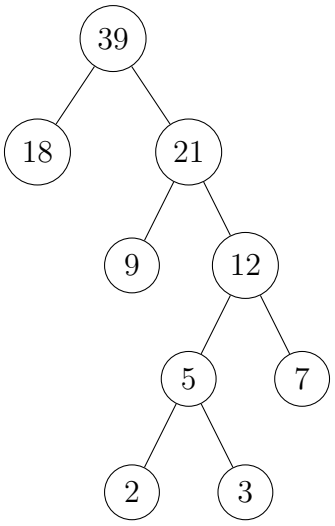


图 9.19: 合并 18 和 21

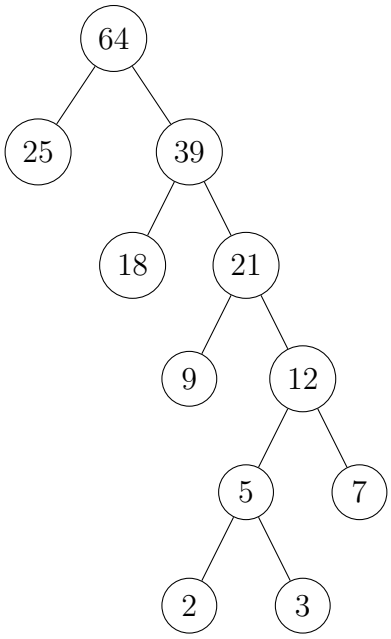


图 9.20: 合并 25 和 39

哈夫曼树有以下几个特点：

1. 没有度为 1 的结点。

2. 哈夫曼树的任意非叶结点的左右子树交换后仍是哈夫曼树。

3. 对同一组权值，可能存在不同构的两棵哈夫曼树。

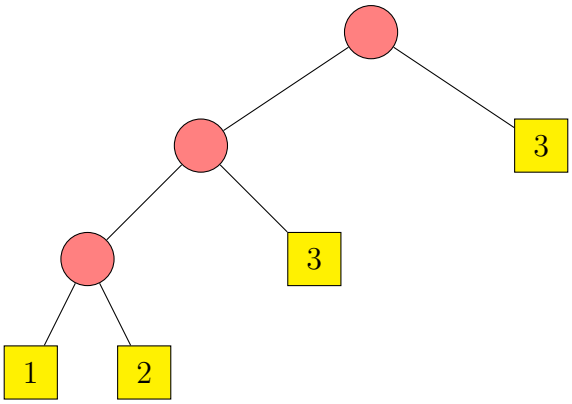


图 9.21: WPL = 3 * 1 + 3 * 2 + 1 * 3 + 2 * 3 = 18

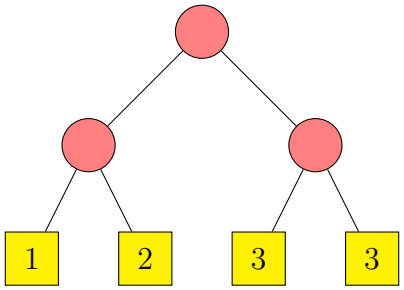


图 9.22: WPL = 1 * 2 + 2 * 2 + 3 * 2 + 3 * 2 = 18

9.6 哈夫曼编码

9.6.1 哈夫曼编码（Huffman Code）

哈夫曼编码是一种高效的编码方式，在信息存储和传输过程中用于对信息进行压缩。要理解哈夫曼编码，需要从信息存储的底层逻辑讲起。

计算机不是人，它不认识中文和英文，更不认识图片和视频，它唯一认识的就是 0（低电平）和 1（高电平）。因此，计算机上一切文字、图象、音频、视频，底层都是用二进制来存储和传输的。

将信息转换成计算机能够识别的二进制形式的过程被称为编码。在 ASCII 码中，每一个字符表示成特定的 8 位二进制数。例如字符串 APPLE 表示成 8 位二进制编码为 01000001 01010000 01010000 01001100 01000101。

显然，ASCII 码是一种等长编码，也就是任何字符的编码长度都相等。等长编码的有点明显，因为每个字符对应的二进制编码长度相等，所以很容易设计，也很方便读写。但是计算机的存储空间以及网络传输的带宽是有限的，等长编码最大的缺点就是编码结果太长，会占用过多资源。

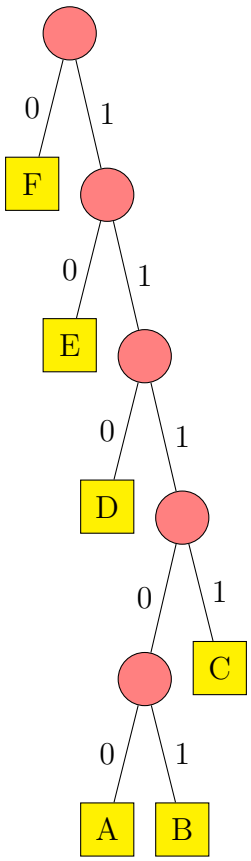
使用不等长编码，让出现频率高的字符用的编码短一些，出现频率低的字符编码长一些，可以使编码的总长度减小。但是不等长编码是不能随意设计的，如果一个字符的编码恰好是另一个字符编码的前缀，就会产生歧义的问题。

哈夫曼编码就是一种不等长的编码，并且任何一个字符的编码都不是另一个字符编码的前缀，因此可以无二义地进行解码，并且信息编码的总长度最小。

哈夫曼编码并非一套固定的编码，而是根据给定信息中各个字符出现的频次，动态生成最优的编码。哈夫曼编码的生成过程就用到了哈夫曼树。

例如一段信息里只有 A、B、C、D、E、F 这 6 个字符，出现的次数分别是 2 次、3 次、7 次、9 次、18 次、25 次。通过把这 6 个字符当成 6 个叶子结点，将出

现次数作为结点的权重，生成一颗哈夫曼树。将哈夫曼树中结点的左分支当做 0、结点的右分支当做 1，从哈夫曼树的根结点到每一个叶子结点的路径，都可以等价为一段二进制编码。



字符	编码
A	11100
B	11101
C	1111
D	110
E	10
F	0

表 9.1: 哈夫曼编码

因为每一个字符对应的都是哈夫曼树的叶子结点，从根结点到这些叶子结点的路径并没有包含关系，最终得到的二进制编码自然也不会是彼此的前缀。