数据结构与算法

Data Structure and Algorithm

极夜酱

目录

Chapter 1 计算复杂性理论

1.1 时间/空间复杂度

1.1.1 算法（Algorithm）

算法是解决问题的一种方法，由一系列的步骤组成。算法有 5 个特点：

1. 有穷性（finiteness）：算法必须在有限个步骤后终止。
2. 确定性（definiteness）：算法的每个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 1 个或多个输出，没有输出的算法是毫无意义的。
5. 可行性（effectiveness）：算法中执行的任何步骤都可以被分解为基本的可执行操作。

1.1.2 时间复杂度（Time Complexity）

算法有高效的，也有拙劣的。衡量算法好坏的标准有时间复杂度和空间复杂度。

想象一个场景：老板让小灰和大黄完成一个需求，两人都完成并交付了各自的代码，代码的功能是一样的。但是，大黄的代码运行一次要花 100ms，占用 5MB 内存；小灰的代码运行一次要花 100s，占用 500MB 内存。

“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，但他的代码存在很严重的问题：运行时间长、占用空间大。

算法的时间复杂度是指算法中基本运算的执行次数，其中基本运算指的是加减法、交换、比较等操作。

算法的效率应该取决于算法本身，与机器无关。分析算法运行效率时应该考虑的是运行时间与输入规模之间的关系。通过计算算法中基本运算的执行次数，可以得到一个关于输入规模 n 的函数。

时间复杂度一般采用大 O 表示法，表示算法的运行时间与输入规模之间的增长关系。常见的时间复杂度包括 $O(1)$、$O(logn)$、$O(n)$、$O(nlogn)$、$O(n^2)$、$O(2^n)$、$O(n!)$ 等。

时间复杂度需要满足以下原则：

1. 如果运行时间是常量级，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，则省去最高阶项前的系数。

大 O 符号

大 O 符号用于表示时间复杂度的上界（最坏情况），即算法的阶不会超过大 O 符号中的函数 $f(n)$。

$$0 \leq f(n) \leq cg(n) \tag{1.1}$$

$$\begin{aligned} f(n) &= n^2 + n \\ &= O(n^2) \\ &= O(n^3) \end{aligned}$$

大 Ω 符号

大 Ω 符号用于表示时间复杂度的下界（最好情况），即算法的阶不会低于大 Ω 符号中的函数 $f(n)$。

$$0 \leq cg(n) \leq f(n) \tag{1.2}$$

$$\begin{aligned} f(n) &= n^2 + n \\ &= \Omega(n^2) \\ &= \Omega(100n) \end{aligned}$$

小 o 符号

小 o 符号用于表示时间复杂度的上界，即算法的阶一定低于小 o 符号中的函数 $f(n)$。

$$0 \leq f(n) < cg(n) \tag{1.3}$$

$$\begin{aligned} f(n) &= n^2 + n \\ &= o(n^3) \end{aligned}$$

小 ω 符号

小 ω 符号用于表示时间复杂度的下界，即算法的阶一定高于小 ω 符号中的函数 $f(n)$。

$$0 \leq cg(n) < f(n) \tag{1.4}$$

$$\begin{aligned} f(n) &= n^2 + n \\ &= \omega(n^2) \end{aligned}$$

Θ 符号

若 $f(n) = O(g(n))$ 且 $f(n) = \Omega(g(n))$，则称 $f(n)$ 的阶与 $g(n)$ 的阶相等：

$$f(n) = \Theta(g(n)) \tag{1.5}$$

$$\begin{aligned} f(n) &= n^2 + n \\ g(n) &= 100n^2 \\ &= \Theta(n^2) \end{aligned}$$

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?

```
1  void buy(int n, int money) {
2      for (int x = 0; x <= n / 5; x++) {
3          for (int y = 0; y <= n / 3; y++) {
4              int z = n - x - y;
5              if (z > 0 && z % 3 == 0 && 5*x + 3*y + z/3 == money) {
6                  printf("x = %d, y = %d, z = %d\n", x, y, z);
7              }
8          }
9      }
10 }
```

1.1.3 空间复杂度（Space Complexity）

算法占用的内存空间自然是越小越好，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，很多时候不得不在时间复杂度和空间复杂度之间进行取舍。绝大多数时候，时间复杂度更为重要，宁可多分配一些内存空间，也要提升程序的执行速度。

1.1.4 均摊时间复杂度（Amortized Time Complexity）

均摊时间复杂度用于分析一组操作中，虽然某个操作的时间复杂度很高，但是经过若干次操作后，这组操作的平均时间复杂度较低的情况。

动态数组

```java
public void add(T element) {
    if (size == capacity) {
        capacity *= 2;
        T[] newArr = (T[]) new Object[capacity];
        System.arraycopy(arr, 0, newArr, 0, size);
        arr = newArr;
    }
    arr[size++] = element;
}
```

这段代码实现了往数组中添加数据的功能。在数组没有满的情况下，直接将数据添加到数组末尾，时间复杂度为 $O(1)$。

但是当数组已满时，需要对数组创建一个更大的数组，将原数组中的数据复制到新数组中，然后再将新数据添加到数组的末尾，这个过程的时间复杂度为 $O(n)$。

假设数组的容量为 n，每执行 n 次 add() 操作，才会进行一次扩容。那么，可以将这一次的 $O(n)$ 的时间均摊到前面的 n 次操作中，这样 add() 操作的均摊时间复杂度就是 $O(1)$。

1.2 递推方程

1.2.1 递推（Recurrence）

递归算法无法直接根据语句的执行次数计算出时间复杂度，但是可以通过递推方程迭代展开进行求解。

求和

```
1 int sum(int n) {
2     if (n <= 0) {
3         return 0;
4     }
5     return sum(n - 1) + n;
6 }
```

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1) + 1 & n \geq 1 \end{cases} \tag{1.6}$$

$$\begin{aligned} T(n) &= T(n-1) + 1 \\ &= [T(n-2) + 1] + 1 \\ &= [[T(n-3) + 1] + 1] + 1 \\ &= \dots \\ &= T(n-k) + k \end{aligned}$$

$$n - k = 0 \Rightarrow n = k$$

$$\begin{aligned} T(n) &= T(n-n) + n \\ &= T(0) + n \\ &= 1 + n \\ &= O(n) \end{aligned}$$

插入排序

```
1  void insert_sort(int arr[], int n) {
2      if (n <= 1) {
3          return;
4      }
5      insert_sort(arr, n-1);
6      int last = arr[n-1];
7      int j = n - 2;
8      while (j >= 0 && arr[j] > last) {
9          arr[j+1] = arr[j];
10         j--;
11     }
12     arr[j+1] = last;
13 }
```

$$T(n) = \begin{cases} 1 & n <= 1 \\ T(n-1) + n & n \geq 1 \end{cases} \tag{1.7}$$

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= [T(n-2) + n] + n \\ &= [[T(n-3) + n] + n] + n \\ &= \dots \\ &= T(n-k) + nk \end{aligned}$$

$$n - k = 0 \Rightarrow n = k$$

$$\begin{aligned} T(n) &= T(n-n) + n^2 \\ &= T(0) + n^2 \\ &= 1 + n^2 \\ &= O(n^2) \end{aligned}$$

汉诺塔

有三根柱子 A、B、C，A 柱子上从下到上套有 n 个圆盘，要求将 A 柱子上的圆盘移动到 C 柱子上。每次只能移动一个圆盘，且大圆盘始终不能叠在小圆盘上面。

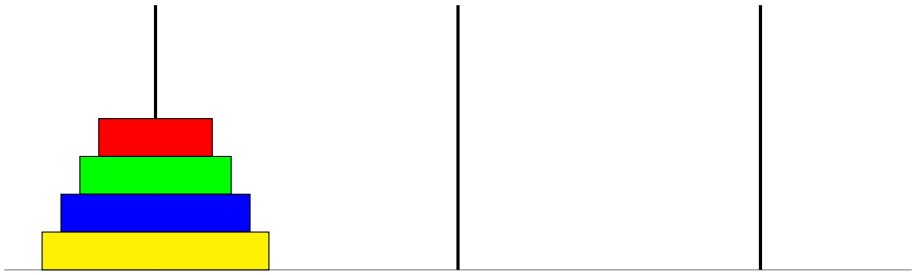


图 1.1: 汉诺塔

递归算法求解汉诺塔问题：

1. 将前 n - 1 个圆盘从 A 柱借助于 C 柱搬到 B 柱。
2. 将最后一个圆盘直接从 A 柱搬到 C 柱。
3. 将 n - 1 个圆盘从 B 柱借助于 A 柱搬到 C 柱。

```
def hanoi(n, A, B, C):
    if n == 1
        move(1, A, C)
    else
        hanoi(n-1, A, C, B)
        move(n, A, C)
        hanoi(n-1, B, A, C)
```

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n-1) + 1 & n > 1 \end{cases} \tag{1.8}$$

$$
\begin{aligned}
T(n) &= 2T(n-1)+1 \\
&= 2[2T(n-2)+1]+1 \\
&= 2[2[2T(n-3)+1]+1]+1 \\
&= \dots \\
&= 2^k * T(n-k) + 2^k - 1
\end{aligned}
$$

$$
n - k = 1 \Rightarrow n = k + 1
$$

$$
\begin{aligned}
T(n) &= 2^{n-1} * T(1) + 2^{n-1} - 1 \\
&= 2^{n-1} + 2^{n-1} - 1 \\
&= 2^n - 1 \\
&= O(2^n)
\end{aligned}
$$

假设每次移动花费 1 秒，解决一个 64 层的汉诺塔问题大约需要 5800 亿年。

1.3 P=NP?

1.3.1 旅行商问题（Traveling Salesman Problem）

小灰最近在工作中遇到了一个棘手的问题。公司正在开发一个物流项目，其中一个需求是为快递员自动规划送快递的路线。

有一个快递员，要分别给三家顾客送快递，他自己到达每个顾客家的路程各不相同，每个顾客之间的路程也各不相同。那么想要把快递依次送达这三家，并最终回到起点，哪一条路线所走的总距离是最短的呢?

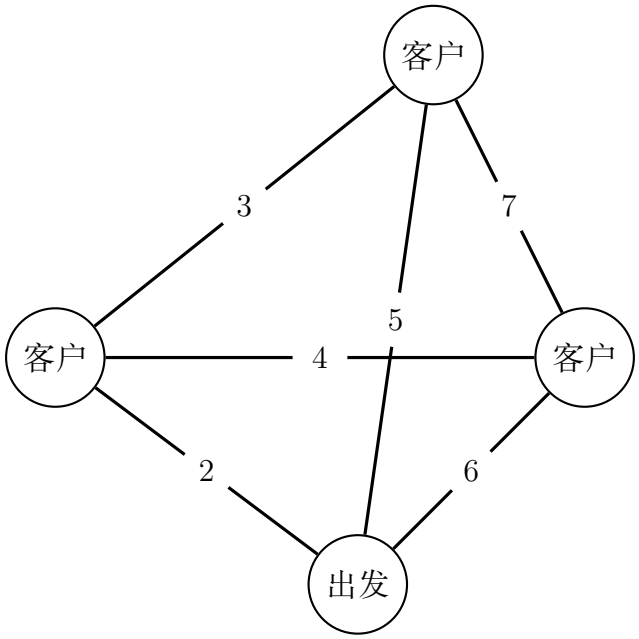


图 1.2: 快递客户路线

为了寻求最优路线，小灰研究了好久，可惜还是没有找到一个高效率的解决方案。不只是小灰，当前的计算机科学家们也没有找到一个行之有效的优化方案，这是典型的旅行商问题。

有一个商品推销员，要去若干个城市推销商品。该推销员从一个城市出发，需要经过所有城市后，回到出发地。每个城市之间都有道路连通，且距离各不相同，推销员应该如何选择路线，使得总行程最短呢?

这个问题看起来很简单，却很难找到一个真正高效的求解算法。其中最容易想到

的，是使用穷举法把所有可能的路线穷举出来，计算出每一条路线的总行程。通过排列组合，从所有路线中找出总行程最短的路线。显然，这个方法的时间复杂度是 $O(n!)$，随着城市数量的增长，花费的运算时间简直不可想象！

后来，人们想出了许多相对优化的解决方案，比如动态规划法和分枝定界法等。但是，这些算法的时间复杂度仍然是指数级的，并没有让性能问题得到根本的解决。

像这样的问题有很多，旅行商问题仅仅是其中的一例。对于这类问题统称为 NP 问题。

1.3.2 P=NP?

算法的设计与分析在计算机科学领域有着重要的应用背景。1966 ~ 2005 年期间，Turing 奖获奖 50 人，其中 10 人以算法设计，7 人以计算理论、自动机和复杂性研究领域的杰出贡献获奖。计算复杂性理论中的 P=NP? 问题是世界七大数学难题之一。

一些常见的算法的时间复杂度，例如二分查找法 $O(logn)$、归并排序 $O(nlogn)$、Floyd 最短路径 $O(n^3)$ 等，都可以用 $O(n^k)$ 表示。这些算法都是多项式时间算法，即能在多项式时间内解决问题。这类问题被称为 P 问题（Polynomial）。

人们常说，能用钱解决的问题都不是问题。在计算机科学家眼中，能用多项式时间解决的问题都不是问题。

然而，世间还存在许多变态的问题，是无法（至少是暂时无法）在多项式时间内解决的，比如一些算法的时间复杂度是 $O(2^n)$，甚至 $O(n!)$。随着问题规模 n 的增长，计算量的增长速度是非常恐怖的。这类问题被称为 NP 问题（Non-deterministic Polynomial），意思是“不确定是否能在多项式时间内解决”。

有些科学家认为，所有的 NP 问题终究都可以在多项式时间内解决，只是我们暂时还没有找到方法；也有些科学家认为，某些 NP 问题永远无法在多项式时间内

解决。这个业界争论用 P=NP? 这个公式来表达。

这还不算完，在所有的 NP 问题当中，还存在着一些大 BOSS，被称为 NPC 问题。

1.3.3 NPC

这里所说的 NPC 问题可不是游戏当中的 NPC。要想理解 NPC 问题，需要先了解归约的概念。

归约（reduction）可以简单理解成问题之间的转化。例如问题是一个一元一次方程的求解问题 Q ：$3x + 6 = 12$，这个问题可以转化成一个一元二次方程 $Q' : 0x^2 + 3x + 6 = 12$。

问题 Q 并不比问题 Q' 难解决，只要有办法解决 Q'，就一定能够解决 Q。对于这种情况，可以说问题 Q 归约于问题 Q'。

同时，这种归约可以逐级传递，比如问题 A 归约于问题 B，问题 B 归约于问题 C，问题 C 归约于问题 D，那么可以说问题 A 归约于问题 D。

在 NP 问题之间，也可以存在归约关系。把众多的 NP 问题层层归约，必定会得到一个或多个终极问题，这些归约的终点就是所谓的 NPC 问题（NP-Complete）。旅行商问题被科学家证明属于 NPC 问题。

俗话说擒贼先擒王，只要有朝一日，我们能够找到 NPC 问题的多项式时间算法，就能够解决掉所有的 NP 问题！但遗憾的是，至今还没有人能够找到可行的方法，很多人认为这个问题是无解的。

回到最初的快递路线规划问题，既然是工程问题，与其钻牛角尖寻求最优解，不如用小得多的代价寻求次优解。最简单的办法是使用贪心算法，先选择距离起点最近的地点 A，再选择距离 A 最近的地点 B，以此类推，每一步都保证局部最优。这样规划出的路线未必是全局最优，但平均情况下也不会比最优方案差多少。

Chapter 2 数组

2.1 查找算法

2.1.1 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的查找算法。

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 $O(n)$。

顺序查找

```
template <typename T>
int sequence_search(T *arr, int n, T key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) {
            return i;
        }
    }
    return -1;
}
```

2.1.2 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

```
template <typename T>
int binary_search(T *arr, int n, T key) {
    int start = 0;
    int end = n - 1;

    while (start <= end) {
        int mid = (start + end) / 2;
        if (arr[mid] == key) {
            return mid;
        } else if (arr[mid] < key) {
            start = mid + 1;
        } else {
            end = mid - 1;
        }
    }
    return -1;
}
```

2.2 数组

2.2.1 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。

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。数组读取元素的时间复杂度是 $O(1)$。

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。

2.2.2 插入元素

在数组中插入元素存在 3 种情况：

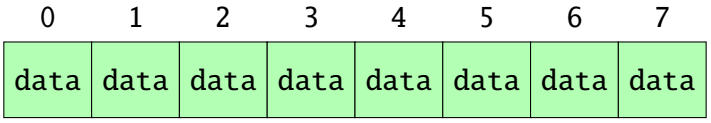


图 2.1: 数组

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。

扩容

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。

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 $O(n)$。

插入元素

```
public void add(T elem) {
    if (size == capacity) {
        resize();
    }
    data[size++] = elem;
}

public void add(int index, T elem) throws IndexOutOfBoundsException {
    if (index < 0 || index > size) {
        throw new IndexOutOfBoundsException("Index out of bounds");
    }

    if (size == capacity) {
        resize();
    }

    for (int i = size - 1; i >= index; i--) {
        data[i + 1] = data[i];
    }
```

```
20
21      data[index] = elem;
22      size++;
23  }
```

2.2.3 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

```
1  public T remove(int index) throws IndexOutOfBoundsException {
2      if (index < 0 || index >= size) {
3          throw new IndexOutOfBoundsException("Index out of bounds");
4      }
5
6      T elem = data[index];
7      for (int i = index; i < size - 1; i++) {
8          data[i] = data[i + 1];
9      }
10     size--;
11     return elem;
12 }
```

数组的删除操作，由于涉及元素的移动，时间复杂度为 $O(n)$。

Chapter 3 链表

3.1 链表种类

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 class Node:
2     def __init__(self, data):
3         self.data = data
4         self.next = None
```

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 NULL。

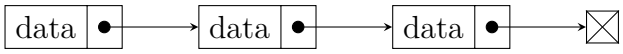


图 3.1: 单向链表

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。

```
1 class Node:
2     def __init__(self, data):
3         self.data = data
4         self.prev = None
5         self.next = None
```

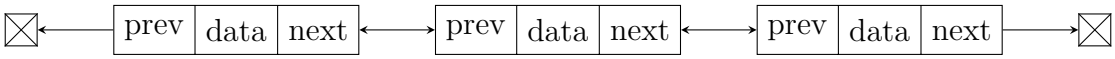


图 3.2: 双向链表

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。

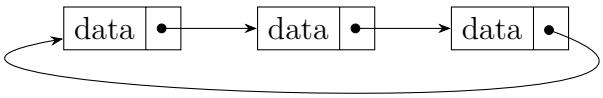


图 3.3: 单向循环链表

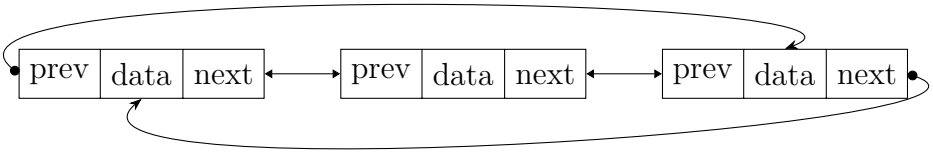


图 3.4: 双向循环链表

3.2 链表

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

```
public T get(int index) throws IndexOutOfBoundsException {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException("Index out of bounds");
    }

    Node<T> current = head;
    for (int i = 0; i < index; i++) {
        current = current.next;
    }
    return current.data;
}
```

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

```
public void set(int index, T data) throws IndexOutOfBoundsException {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException("Index out of bounds");
```

```
4       }
5
6       Node<T> current = head;
7       for (int i = 0; i < index; i++) {
8           current = current.next;
9       }
10      current.data = data;
11  }
```

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。

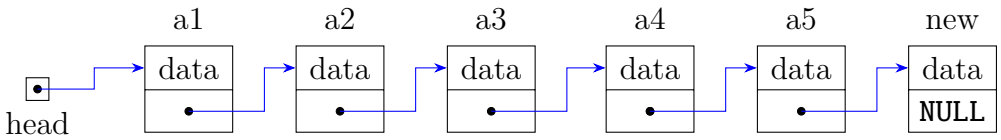


图 3.5: 尾部插入

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。

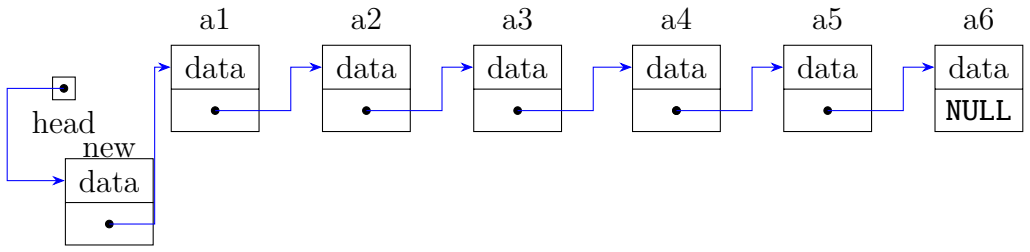


图 3.6: 头部插入

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。

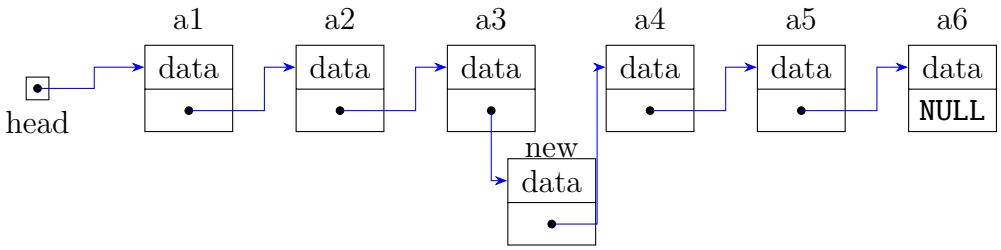


图 3.7: 中间插入

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 $O(1)$。

插入结点

```
1  public void add(T data) {
2      Node<T> newNode = new Node<T>(data, null);
3      if (isEmpty()) {
4          head = newNode;
5          tail = newNode;
6      } else {
7          tail.next = newNode;
8          tail = newNode;
9      }
10     size++;
11 }
12
13 public void add(int index, T data) throws IndexOutOfBoundsException {
14     if (index < 0 || index > size) {
15         throw new IndexOutOfBoundsException("Index out of bounds");
16     }
17
18     Node<T> newNode = new Node<T>(data, null);
```

```
19      if (index == 0) {
20          newNode.next = head;
21          head = newNode;
22      } else {
23          Node<T> prev = head;
24          for (int i = 0; i < index - 1; i++) {
25              prev = prev.next;
26          }
27          newNode.next = prev.next;
28          prev.next = newNode;
29      }
30      size++;
31  }
```

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。

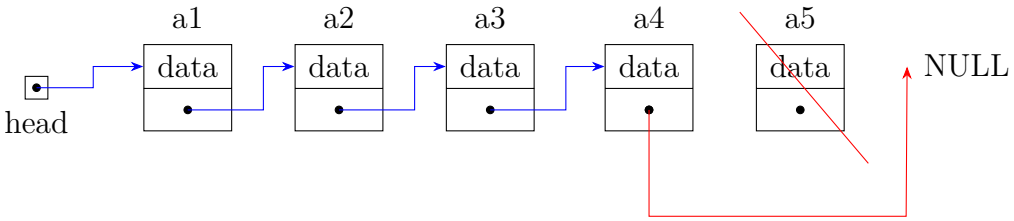


图 3.8: 尾部删除

头部删除

把链表的头结点设置为原先头结点的 next 指针。

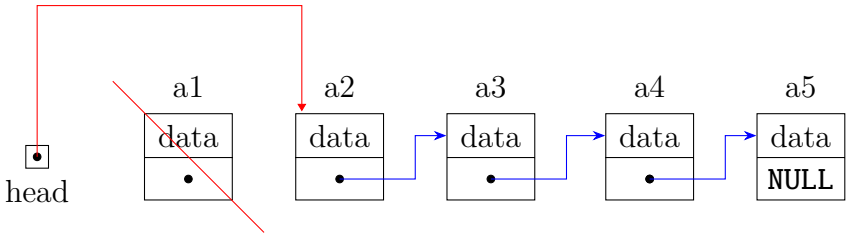


图 3.9: 头部删除

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。

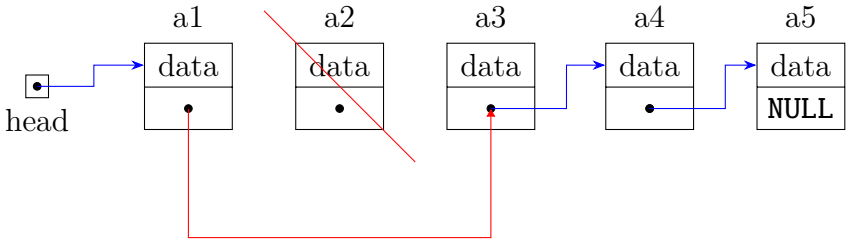


图 3.10: 中间删除

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 $O(1)$。

删除结点

```
1  public T remove(int index) throws IndexOutOfBoundsException {
2      if (index < 0 || index >= size) {
3          throw new IndexOutOfBoundsException("Index out of bounds");
4      }
5
6      T data;
7      if (index == 0) {
8          data = head.data;
9          head = head.next;
```

```
    } else {
        Node<T> prev = head;
        for (int i = 0; i < index - 1; i++) {
            prev = prev.next;
        }
        data = prev.next.data;
        prev.next = prev.next.next;
    }
    size--;
    return data;
}
```

3.2.5 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。

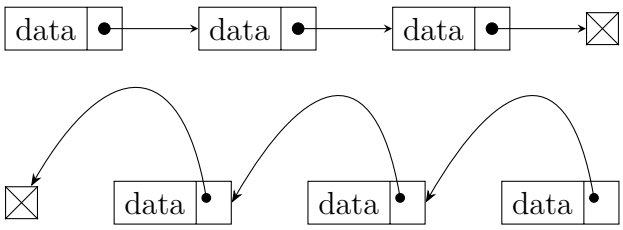


图 3.11: 反转链表

反转链表（递归）

```
private Node<T> reverseRecursive(Node<T> current) {
    if (current == null || current.next == null) {
        return current;
```

```
    }
    Node<T> newHead = reverse(current.next);
    current.next.next = current;
    current.next = null;
    return newHead;
}

public void reverseRecursive() {
    head = reverseRecursive(head);
}
```

反转链表（迭代）

```
public void reverse() {
    Node<T> prev = null;
    Node<T> current = head;
    Node<T> next = null;
    while (current != null) {
        next = current.next;
        current.next = prev;
        prev = current;
        current = next;
    }
    head = prev;
}
```

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）。

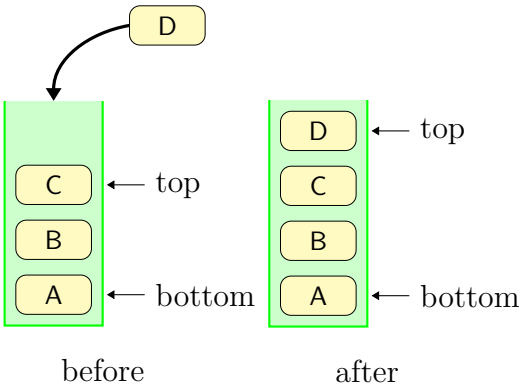


图 4.1: 栈

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。

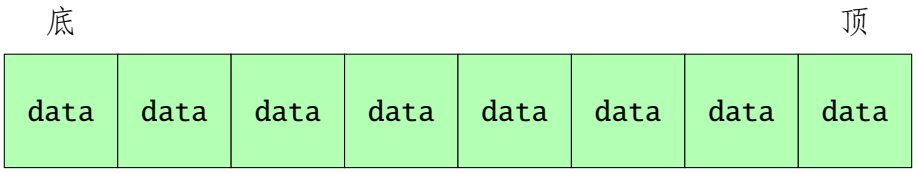


图 4.2: 顺序栈

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。

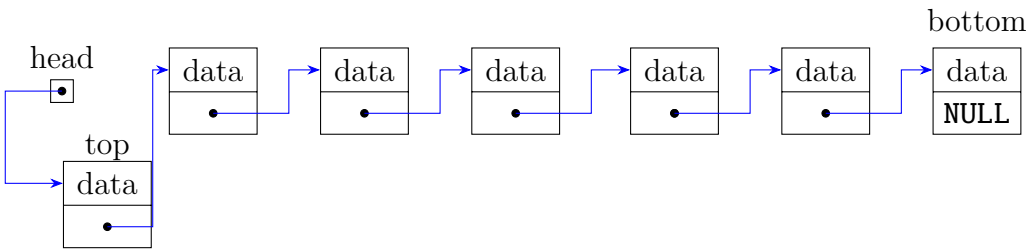


图 4.3: 链式栈

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 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 stack_t *stack_push(stack_t *stack, T elem) {
2     array_append(stack->data, elem);
3     return stack;
4 }
```

4.1.5 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 T stack_pop(stack_t *stack) {
2     return array_remove(stack->data, stack_size(stack) - 1);
3 }
```

4.2 括号匹配

4.2.1 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以正确的顺序用相同类型的右括号闭合。

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，或者栈中没有左括号，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:
            stack.append(paran)

    return not stack
```

4.3 表达式求值

4.3.1 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。

表达式求值

```
def calculate(expression):
    stack = []
    tokens = expression.split()

    for token in tokens:
        if token in "+-*/":
            right = stack.pop()
            left = stack.pop()
            if token == "+":
                stack.append(left + right)
            elif token == "-":
                stack.append(left - right)
            elif token == "*":
                stack.append(left * right)
            elif token == "/":
                stack.append(left / right)
        else:
            stack.append(int(token))

    return stack.pop()

```

```
expressions = [
    "1 2 +",            # 1 + 2 = 3
    "2 3 4 + *",        # 2 * (3 + 4) = 14
    "1 2 + 3 4 + *",    # (1 + 2) * (3 + 4) = 21
    "3 4 2 + * 5 *",    # 3 * (4 + 2) * 5 = 90
    "50 20 - 2 /",      # (50 - 20) / 2 = 15
]

for expression in expressions:
    print(expression, "=", calculate(expression))
```

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 $O(1)$。

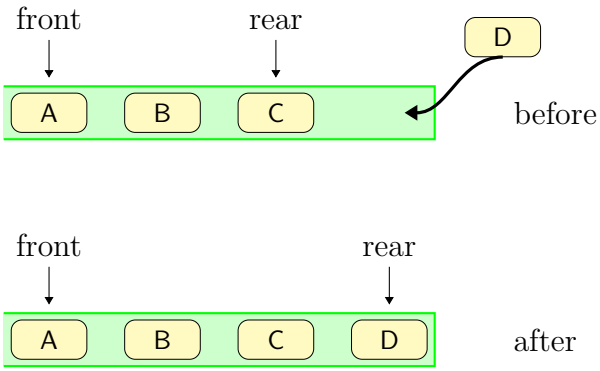


图 5.1: 入队

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 $O(1)$。

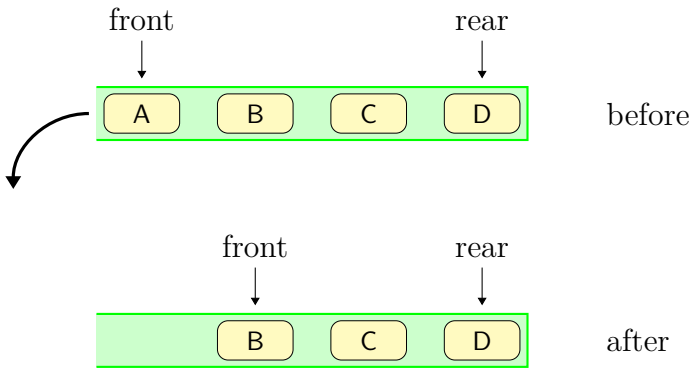


图 5.2: 出队

5.1.4 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。

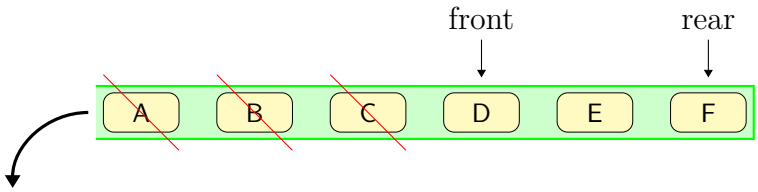


图 5.3: 队列存在的问题

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。

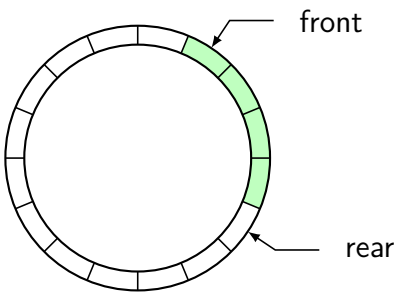


图 5.4: 循环队列

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 queue_t *queue_enqueue(queue_t *quque, T elem) {
2     queue->data[queue->rear] = elem;
3     queue->rear = (queue->rear + 1) % queue->max;
4     return queue;
5 }
```

出队

```
T queue_enqueue(queue_t *queue) {
    T elem = queue->data[queue->front];
    queue->front = (queue->front + 1) % queue->max;
    return elem;
}
```

5.2 双端队列

5.2.1 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。

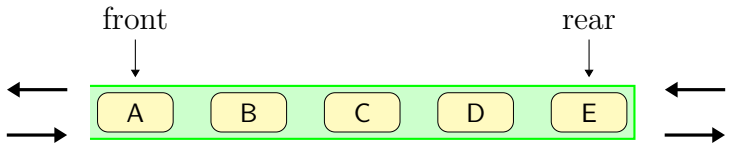


图 5.5: 双端队列

双端队列

```
1  class Deque:
2      def __init__(self):
3          self.__data = []
4
5      def is_empty(self):
6          return self.__data == []
7
8      def __len__(self):
9          return len(self.__data)
10
11     def push_front(self, elem):
12         self.__data.insert(0, elem)
13
14     def push_back(self, elem):
15         self.__data.append(elem)
16
17     def pop_front(self):
18         return self.__data.pop(0)
19
20     def pop_back(self):
21         return self.__data.pop()
22
```

```
23      def front(self):
24          return self.__data[0]
25
26      def back(self):
27          return 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	Alice
002123	Bob
002931	Charlie
003278	Daniel

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。

6.1.2 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。

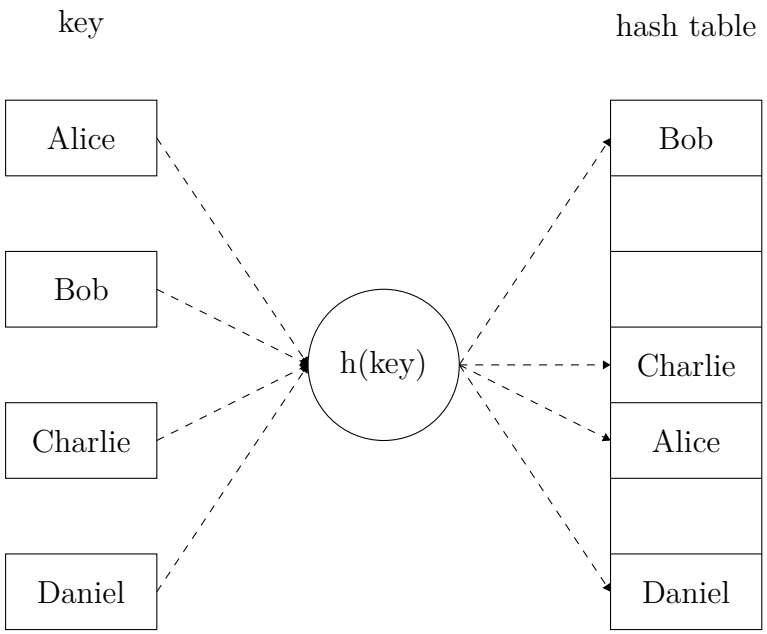


图 6.1: 哈希函数

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。

6.1.3 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：

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

$$h(56793542) = 641$$

6.1.4 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：

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

6.2 哈希冲突

6.2.1 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。

6.2.2 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。

6.2.3 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。

6.2.4 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：

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

6.2.5 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。

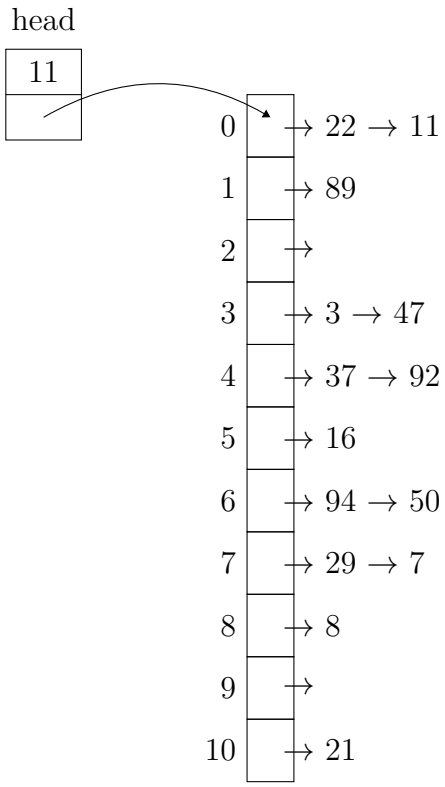


图 6.2: 分离链接法

6.2.6 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。

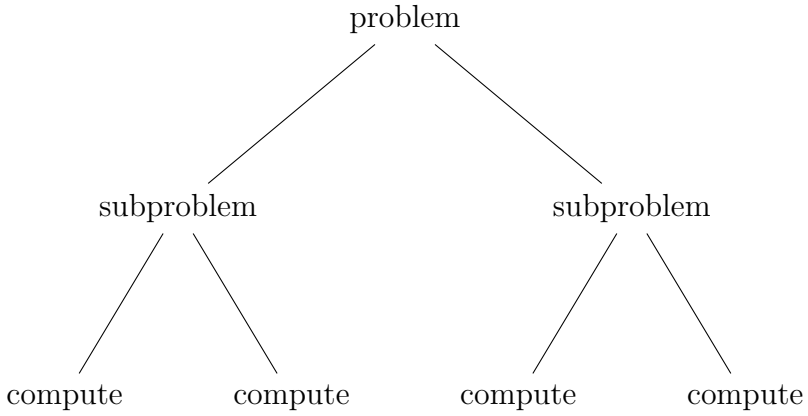


图 7.1: 分治法

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 大整数运算

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):
```

```
    # pad the shorter number with leading zeros
    if len(num1) > len(num2):
        num2 = num2.zfill(len(num1))
    else:
        num1 = num1.zfill(len(num2))

    result = ""
    carry = 0
    for i in range(len(num1) - 1, -1, -1):
        digit_sum = int(num1[i]) + int(num2[i]) + carry
        carry = digit_sum // 10
        digit = digit_sum % 10
        result = str(digit) + result

    if carry > 0:
        result = str(carry) + result
    return result
```

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。

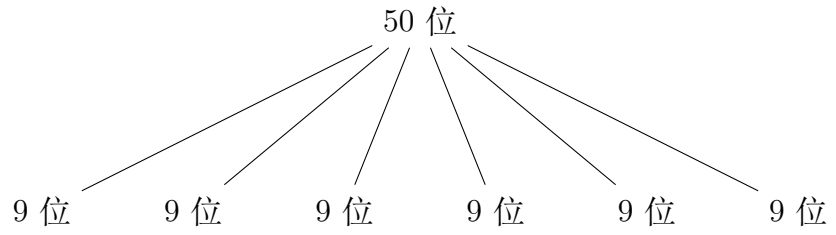


图 7.2: 大整数加法优化

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。

7.2.2 大整数乘法

起初对于大整数乘法，认为只要按照大整数相加的思路稍微做一下变形，就可以轻松实现。但是随着深入的学习，才发现事情并没有那么简单。如果沿用大整数加法的思路，通过列竖式求解……

$$
\begin{array}{r}
93281 \\
\times \quad 2034 \\
\hline
373124 \\
279843 \\
000000 \\
186562 \\
\hline
189733554
\end{array}
$$

乘法竖式的计算过程可以大体分为两步：

1. 整数 B 的每一个数位和整数 A 所有数位依次相乘，得到中间结果。
2. 所有中间结果相加，得到最终结果。

这样的做法确实可以实现大整数乘法，由于两个大整数的所有数位都需要一一彼此相乘。如果整数 A 的长度为 m，整数 B 的长度为 n，那么时间复杂度就是 $O(m*n)$。如果两个大整数的长度接近，那么时间复杂度也可以写为 $O(n^2)$。

那么有没有优化方法，可以让时间复杂度低于 $O(n^2)$ 呢?

利用分治法可以简化问题的规模，可以把大整数按照数位拆分成两部分。

$$
\text{整数 } 1 = \underbrace{81325}_{A}\underbrace{79076}_{B}
$$

$$
\text{整数 } 2 = \underbrace{9213}_{C}\underbrace{52184}_{D}
$$

即：

$$\text{整数 } 1 = A \times 10^5 + B$$
$$\text{整数 } 2 = C \times 10^5 + D$$

如果把两个大整数的长度抽象为 m 和 n，那么：

$$\text{整数 } 1 = A \times 10^{m/2} + B$$
$$\text{整数 } 2 = C \times 10^{n/2} + D$$

因此：

$$\begin{aligned}
&\text{整数 } 1 \times \text{整数 } 2 \\
&= (A \times 10^{m/2} + B) \times (C \times 10^{n/2} + D) \\
&= AC \times 10^{\frac{m+n}{2}} + AD \times 10^{\frac{n}{2}} + BC \times 10^{\frac{m}{2}} + BD
\end{aligned}$$

如此一来，原本长度为 n 的大整数的 1 次乘积，被转化成了长度为 $n/2$ 的大整数的 4 次乘积。

通过递归把大整数不断地对半拆分，一直拆分到可以直接计算为止。

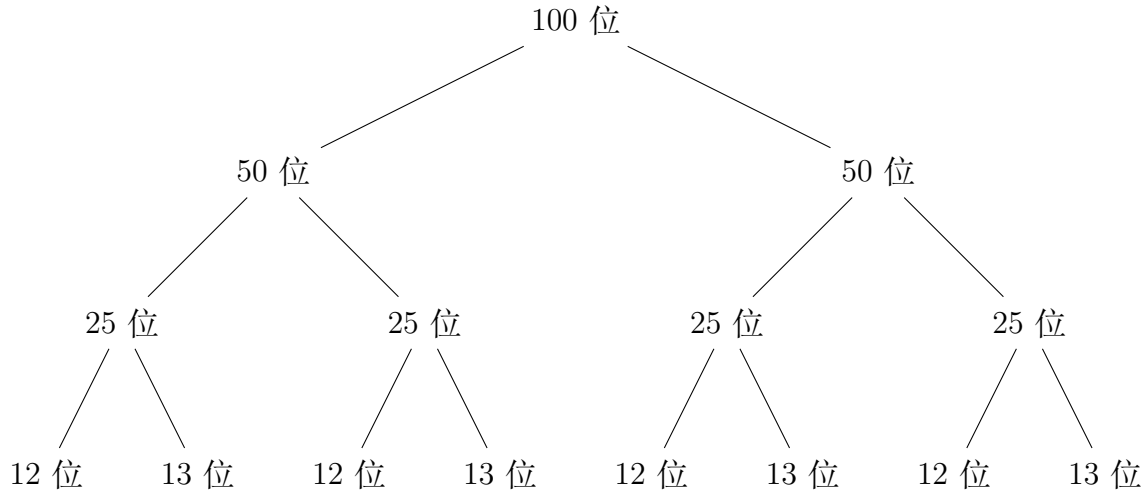


图 7.3: 大整数拆分

但是先别高兴地太早，这个方法真的提高了效率吗?

$$T(n) = \begin{cases} O(1) & n = 1 \\ 4T(n/2) + O(n) & n > 1 \end{cases}$$

求解得到：

$$T(n) = O(n^2)$$

闹了半天，时间复杂度还是 $O(n^2)$ 啊，白高兴一场。但是努力的方向并没有白费，这里还是有优化的方法的。

在大整数乘法运算中，如果只简单地利用分治法将大整数的位置减半，并不能降低时间复杂度的阶。分治法把两个大整数相乘到的问题转化为四个较小整数相乘，性能的瓶颈仍旧在乘法上。

分治算法的时间复杂度方程为 $W(n) = aW(n/b) + f(n)$，其中 a 为子问题数，n/b 为子问题规模，$f(n)$ 为划分与合并工作量。当 a 较大、b 较小、$f(n)$ 不大时，$W(n) = \Theta(n^{log_b a})$。减少 a 是降低 $W(n)$ 的阶的一种途径。利用子问题的依赖问题，可以使某些子问题的解通过组合其它子问题的解而得到。

那么，怎样才能减少乘法运算的次数呢？哪怕由四次乘法变成三次乘法也好呀。通过对之前的乘法等式做一些调整，可以减少乘法的次数。

$$\begin{aligned}
&\text{整数 } 1 \times \text{整数 } 2 \\
&= (A \times 10^{n/2} + B) \times (C \times 10^{n/2} + D) \\
&= AC \times 10^n + AD \times 10^{n/2} + BC \times 10^{n/2} + BD \\
&= AC \times 10^n + (AD + BC) \times 10^{n/2} + BD \\
&= AC \times 10^n + (AD - AC - BD + BC + AC + BD) \times 10^{n/2} + BD \\
&= AC \times 10^n + ((A - B)(D - C) + AC + BD) \times 10^{n/2} + BD
\end{aligned}$$

这样一来，原本的 4 次乘法和 3 次加法，转变成了 3 次乘法和 6 次加法。

骗人！最后式子里明明包含五次乘法啊！

AC 出现了两次，BD 也出现了两次，这两个乘积分别只需计算一次就行了，所以总共只需要三次乘法。

$$T(n) = \begin{cases} O(1) & n = 1 \\ 3T(n/2) + O(n) & n > 1 \end{cases}$$

求解得到：

$$T(n) = O(n^{log_2 3}) \approx O(n^{1.59})$$

7.3 快速幂

7.3.1 快速幂（Fast Exponentiation）

使用传统算法计算 x^n 的时间复杂度为 $\Theta(n)$，然而利用快速幂的算法时间复杂度为 $\Theta(logn)$。

$$x^n = \begin{cases} x^{n/2} * x^{n/2} & n\text{为偶数} \\ x^{(n-1)/2} * x^{(n-1)/2} * x & n\text{为奇数} \end{cases}$$

例如计算 2^{18} 只需要 4 步即可：

$$2^{18} = 2^9 * 2^9$$
$$2^9 = 2^4 * 2^4 * 2$$
$$2^4 = 2^2 * 2^2$$
$$2^2 = 2^1 * 2$$

快速幂

```
def fast_exponentiation(x, n):
    if n == 0:
        return 1

    if n % 2 == 0:
        half = fast_exponentiation(x, n // 2)
        return half * half
    else:
        half = fast_exponentiation(x, (n - 1) // 2)
        return half * half * x
```

7.4 矩阵乘法

7.4.1 矩阵乘法

假设 A 和 B 为 n 阶矩阵（$n = 2^k$），计算时，对于 C 中 n^2 个元素，每个元素都需要做 n 次乘法，因此 $W(n) = O(n^3)$。

利用简单的分治策略，可以将矩阵分块计算计算。

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

其中，

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{12}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{21}B_{22} \end{aligned}$$

这样就把原问题转换为了 8 个子问题，递推方程为：

$$W(n) = \begin{cases} 1 & n = 1 \\ 8W(n/2) + cn^2 & n > 1 \end{cases}$$

求解得到：

$$W(n) - O(n^3)$$

简单的分治算法并不能降低时间复杂度的阶，但是矩阵乘法可以通过减少子问题的个数进行优化。

7.4.2 Strassen 矩阵乘法

让 M_1，M_2，…，M_7 分别对应矩阵乘法的 7 个子问题：

$$
\begin{aligned}
M_1 &= A_{11}(B_{12} - B_{22}) \\
M_2 &= (A_{11} + A_{12})B_{22} \\
M_3 &= (A_{21} + A_{22})B_{11} \\
M_4 &= A_{22}(B_{21} - B_{11}) \\
M_5 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
M_6 &= (A_{12} - A_{22})(B_{21} + B_{22}) \\
M_7 &= (A_{11} - A_{21})(B_{11} + B_{12})
\end{aligned}
$$

利用这些中间矩阵，可以得到结果矩阵：

$$
\begin{aligned}
C_{11} &= M_5 + M_4 - M_2 + M_6 \\
C_{12} &= M_1 + M_2 \\
C_{21} &= M_3 + M_4 \\
C_{22} &= M_5 + M_1 - M_3 - M_7
\end{aligned}
$$

在这些运算中，一共有 7 个子问题和 18 次矩阵加减法，时间复杂度为：

$$
W(n) = \begin{cases} 1 & n = 1 \\ 7W(n/2) + 18(n/2)^2 & n > 1 \end{cases}
$$

求解得到：

$$
W(n) = O(n^{log7}) \approx O(n^{2.8075})
$$

Coppersmith-Winograd 算法是目前已知最好的矩阵乘法算法，时间复杂度为 $O(n^{2.376})$。矩阵乘法可以应用在科学计算、图形处理、数据挖掘等方面，在回归、聚类、主成分分析、决策树等挖掘算法中常常涉及大规模矩阵运算。

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。

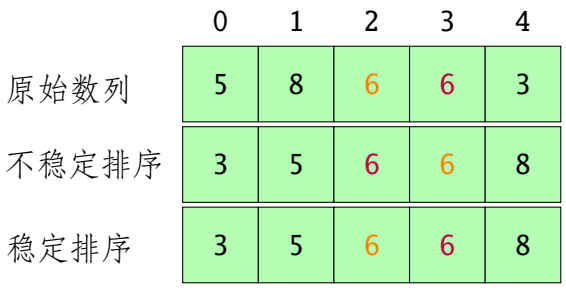


图 8.1: 排序稳定性

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。

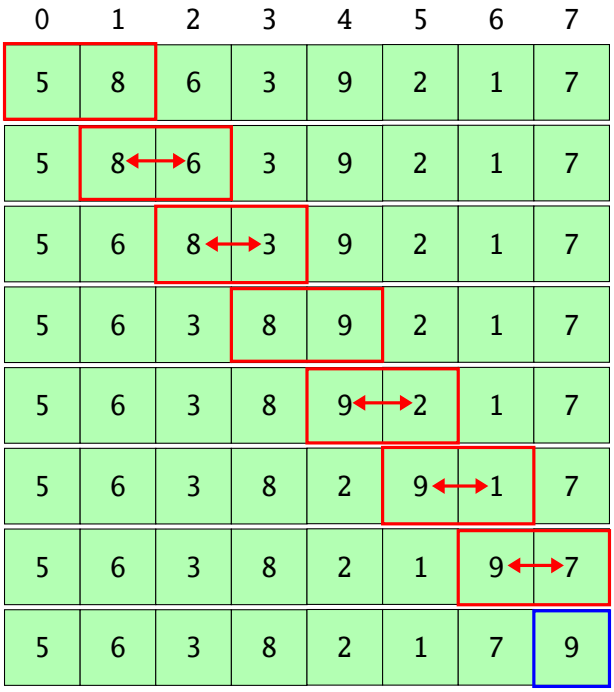


图 8.2: 冒泡排序第 1 轮

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：

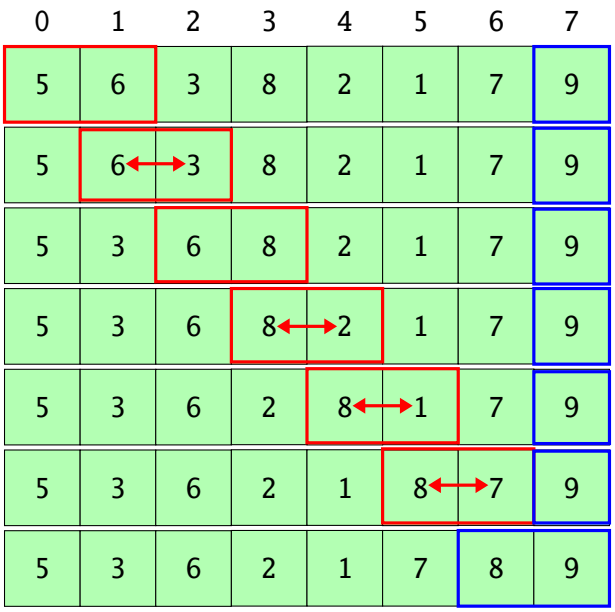


图 8.3: 冒泡排序第 2 轮

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：

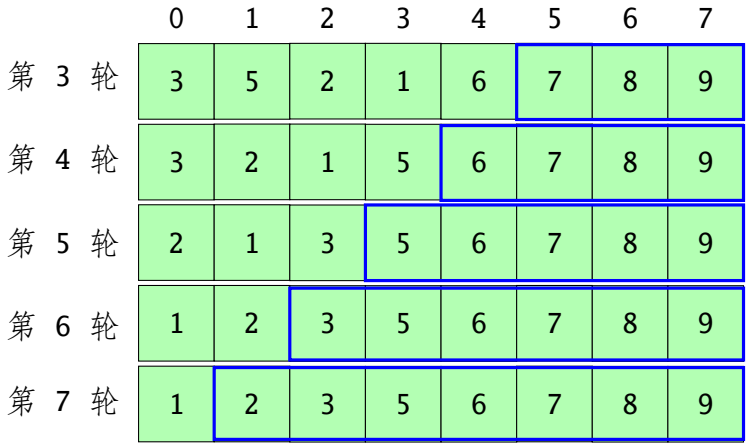


图 8.4: 冒泡排序第 3 ～ 7 轮

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

```c
int *bubble_sort(int *arr, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
    return arr;
}
```

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)。

```c
int count_inverses(int *arr, int n) {
    int count = 0;
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (arr[i] > arr[j]) {
                count++;
            }
        }
    }
    return count;
}
```

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

```
int *bubble_sort(int *arr, int n) {
    for (int i = 0; i < n; i++) {
        bool swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (!swapped) {
            break;
        }
    }

    return arr;
}
```

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

```
int *bubble_sort(int *arr, int n) {
    int right = n - 1;
    for (int i = 0; i < n; i++) {
        bool swapped = false;
        int last = 0;
        for (int j = 0; j < right; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
                last = j;
            }
        }
        if (!swapped) {
            break;
        }
        right = last;
    }

    return arr;
}
```

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

```c
int* selection_sort(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;
            }
        }
        swap(arr[i], arr[minIndex]);
    }

    return arr;
}
```

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

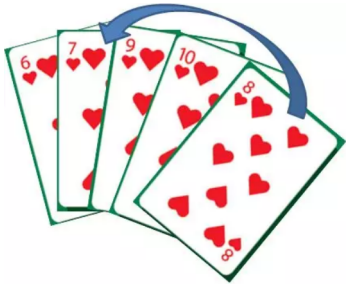
```
int* selection_sort(int* arr, int n) {
    for (int i = 0; i < n / 2; i++) {
        int minIndex = i;
        int maxIndex = n - i - 1;

        for (int j = i; j < n - i; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
            if (arr[j] > arr[maxIndex]) {
                maxIndex = j;
            }
        }

        swap(arr[i], arr[minIndex]);

        // In case of i == maxIndex before swap(arr[i], arr[minIndex]), it's now at m
        if (i == maxIndex) {
            maxIndex = minIndex;
        }

        swap(arr[n - i - 1], arr[maxIndex]);
    }

    return arr;
}
```

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。



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 8.6: 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  public static int[] insertionSort(int[] arr) {
2      for (int i = 1; i < arr.length; 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
12     return arr;
13 }
```

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

```java
public static int[] insertionSort(int[] arr) {
    for (int i = 1; i < arr.length; i++) {
        int temp = arr[i];
        int left = 0;
        int right = i - 1;

        while (left <= right) {
            int mid = (left + right) / 2;
            if (temp < arr[mid]) {
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }

        for (int j = i - 1; j >= left; j--) {
            arr[j + 1] = arr[j];
        }
        arr[left] = temp;
    }

    return arr;
}
```

8.5 希尔排序

8.5.1 希尔排序（Shell Sort）

希尔排序本质上是直接插入排序的升级版。对于插入排序而言，在大多数元素已经有序的情况下，工作量会比较小。这个结论很明显，如果一个数组大部分元素都有序，那么数组中的元素自然不需要频繁地进行比较和交换。

如何能够让待排序的数组中大部分元素有序呢？需要对原始数组进行预处理，使得原始数组的大部分元素变得有序。采用分组的方法，可以将数组进行一定程度地粗略调整。

例如一个有 8 个数字组成的无序序列 {5, 8, 6, 3, 9, 2, 1, 7}，进行升序排序。让元素两两一组，同组两个元素之间的跨度为数组总长度的一半。

接着让每组元素进行独立排序，排序方式使用直接插入排序即可。由于每一组的元素数量很少，所以插入排序的工作量很少。这样一来，仅仅经过几次简单的交换，数组整体的有序程序得到了显著提高，使得后续再进行直接插入排序的工作量大大减少。

但是这样还不算完，还可以进一步缩小分组跨度，重复上述工作。

例如一个有 8 个数字组成的无序序列 {5, 8, 6, 3, 9, 2, 1, 7}，进行升序排序。

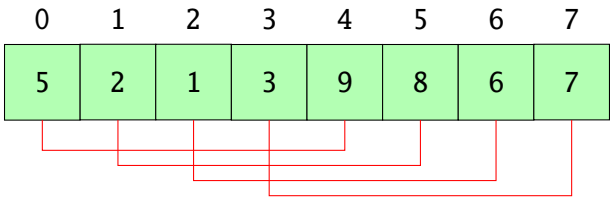


图 8.7: 跨度为 4 分组交换

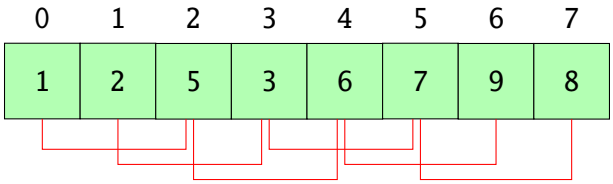


图 8.8: 跨度为 2 分组交换

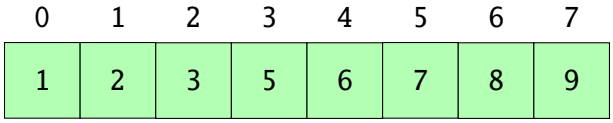


图 8.9: 跨度为 1 分组交换

希尔排序的发明者是计算机科学家 Donald Shell。希尔排序中说使用的分组跨度被称为希尔排序的增量。增量的选择可以有很多种，最朴素的就是 Donald Shell 在发明希尔排序时所提出的逐步折半的方法。

8.5.2 算法分析

希尔排序利用分组粗略调整的方式减少了直接插入排序的工作量，使得算法的平均时间复杂度低于 $O(n^2)$。但是在某些极端情况下，希尔排序的最坏时间复杂度仍然是 $O(n^2)$，甚至比插入排序更慢。

例如 {2, 1, 5, 3, 7, 6, 9, 8}，无论是以 4 为增量，还是以 2 为增量，每组内部的元素都没有任何交换。直到增量缩减为 1，数组才会按照直接插入排序的方式进行调整。

对于这样的数组，希尔排序不但没有减少直接插入排序的工作量，反而白白增加了分组操作的成本。

这是因为每一轮希尔增量之间都是等比的，这就导致了希尔增量存在盲区。为了避免这样的极端情况，科学家发明了许多更为严谨的增量方式。其中最具有代表性的是 Hibbard 增量和 Sedgewick 增量。

Hibbard 增量序列

Hibbard 增量序列为 $1, 3, 7, 15, \ldots$，通项公式为 $2^i - 1$。

利用这种增量方式的希尔排序，最坏时间复杂度是 $O(n^{3/2})$。

Sedgewick 增量序列

Sedgewick 增量序列为 $1, 5, 19, 41, 109, \ldots$，通项公式为 $9 \times 4^i - 9 \times 2^i + 1$ 和 $4^{i+2} - 3 \times 2^{i+2} + 1$。

利用这种增量方式的希尔排序，最坏时间复杂度是 $O(n^{4/3})$。

这两种增量方式的时间复杂度需要很复杂的数学证明，有些是人们的大致猜想。

时间复杂度	**空间复杂度**	**稳定性**
$O(n^{1.3\sim2})$	$O(1)$	不稳定

表 8.4: 希尔排序算法分析

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。

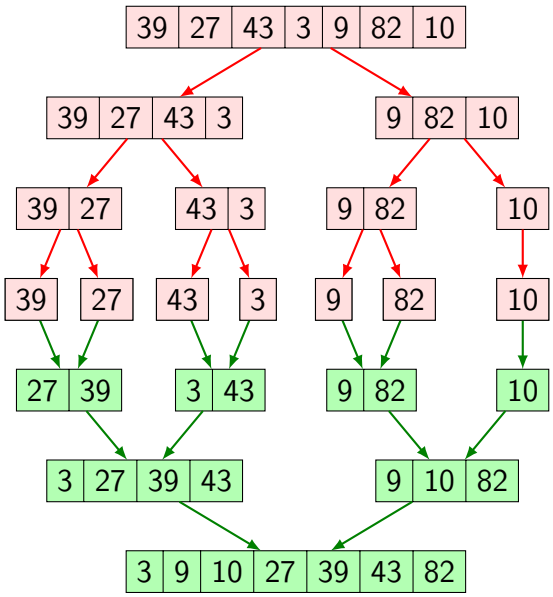


图 8.10: 归并排序

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 8.5: 归并排序算法分析

归并排序

```
1 def merge_sort(lst):
```

```
    """Merge Sort (original v1.0)"""
    if len(lst) <= 1:
        return lst

    mid = len(lst) // 2
    left_half = lst[:mid]
    right_half = lst[mid:]

    merge_sort(left_half)
    merge_sort(right_half)

    i = 0
    j = 0
    k = 0

    while i < len(left_half) and j < len(right_half):
        if left_half[i] < right_half[j]:
            lst[k] = left_half[i]
            i += 1
        else:
            lst[k] = right_half[j]
            j += 1
        k += 1

    while i < len(left_half):
        lst[k] = left_half[i]
        i += 1
        k += 1

    while j < len(right_half):
        lst[k] = right_half[j]
        j += 1
        k += 1

    return lst
```

8.6.2 归并排序优化

简单的归并排序利用分治法，递归地将对小规模子数组进行处理。但是递归会使小规模问题中方法调用太过频繁，因此对于规模较小的子数组可以采用插入排序。一般来说插入排序在小数组中比归并更快，这种优化可以使归并排序的运行时间缩短 $10\% \sim 15\%$。

另一个可以优化的地方是对于单次合并的过程，例如将子数组 $arr[start..mid]$ 和 $arr[mid+1..end]$ 进行合并，如果 $arr[mid] \leq arr[mid+1]$ 的话，说明 $arr[start..end]$ 已经为有序状态，无序再进行不必要的合并。

归并排序优化

```
def merge_sort(lst):
    """Merge Sort (optimized v2.0)"""
    if len(lst) <= 10:
        return insertion_sort(lst)

    mid = len(lst) // 2
    left_half = lst[:mid]
    right_half = lst[mid:]

    left_half = merge_sort(left_half)
    right_half = merge_sort(right_half)

    i = 0
    j = 0
    k = 0

    while i < len(left_half) and j < len(right_half):
        if left_half[i] < right_half[j]:
            lst[k] = left_half[i]
            i += 1
        else:
            lst[k] = right_half[j]
```

```
            j += 1
        k += 1

    while i < len(left_half):
        lst[k] = left_half[i]
        i += 1
        k += 1

    while j < len(right_half):
        lst[k] = right_half[j]
        j += 1
        k += 1

    return lst
```

8.7 快速排序

8.7.1 快速排序（Quick Sort）

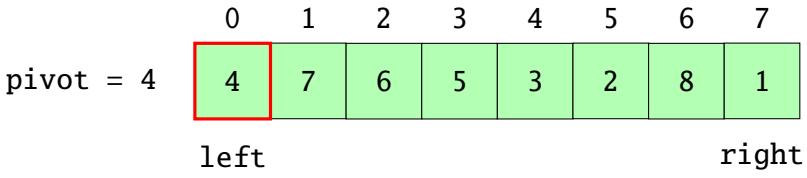
快速排序是很重要的算法，与傅里叶变换等算法并称二十世纪十大算法。

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。

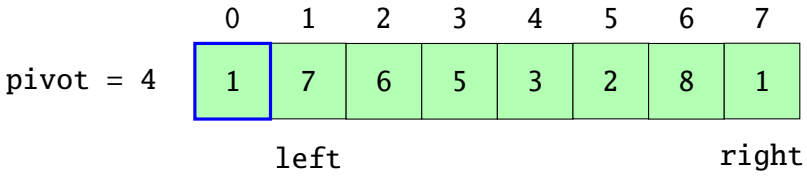
选择基准元素最简单的方式是选择数列的第一个元素。这种选择在绝大多数情况下是没有问题的，但是如果对一个原本逆序的数列进行升序排序，整个数列并没有被分成一半，每一轮仅仅确定了基准元素的位置。这种情况下数列第一个元素要么是最小值，要么是最大值，根本无法发挥分治法的优势。在这种极端情况下，快速排序需要进行 n 轮，时间复杂度退化成了 $O(n^2)$。
如何避免这种极端情况呢？可以不选择数列的第一个元素，而是随机选择一个元素作为基准元素。这样一来，即使是在数列完全逆序的情况下，也可以有效地将数列分成两部分。当然，即使是随机选择，每一次也有极小的几率选到数列的最大值或最小值，同样会对分治造成一定影响。

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？

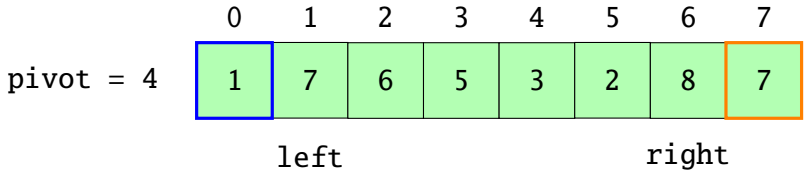
例如一个有 8 个数字组成的无序序列，进行升序排序。选定基准元素 pivot，设置两个指针 left 和 right，指向数列的最左和最右两个元素。



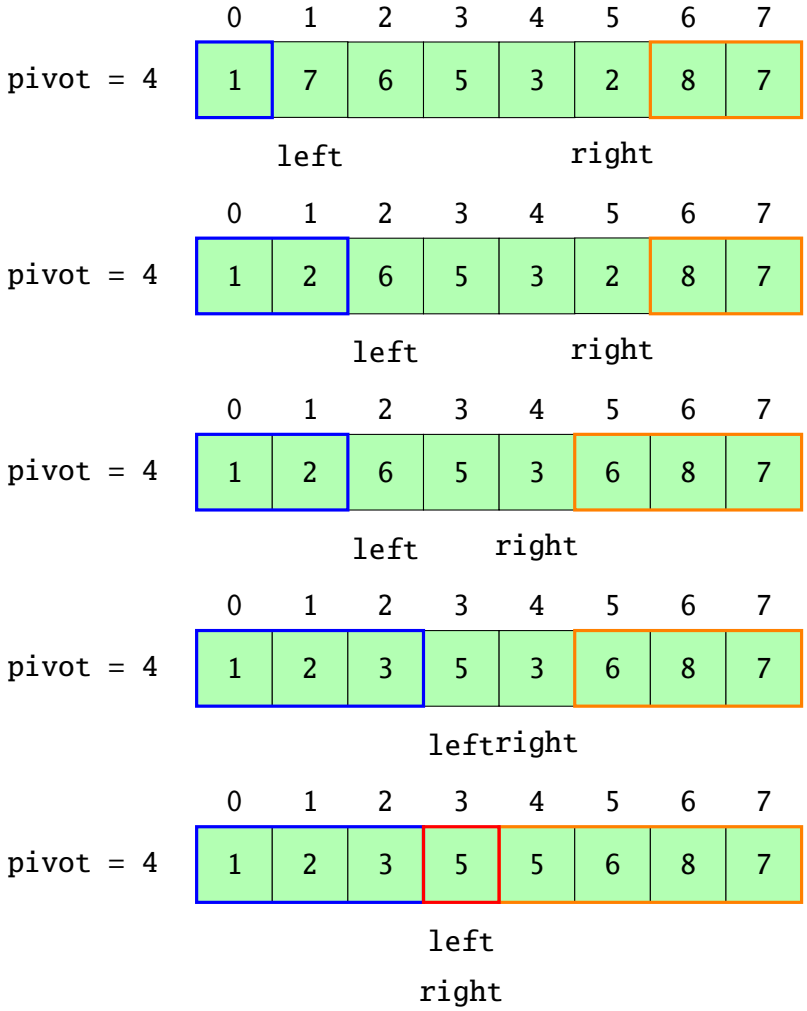
从 right 指针开始，把指针所指向的元素和基准元素做比较。如果比 pivot 大，则 right 指针向左移动；如果比 pivot 小，则把 right 所指向的元素填入 left 指针所指向的位置，同时 left 向右移动一位。



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 8.6: 快速排序算法分析

快速排序

```
    def partition(lst, start, end, pivot):
    i = start + 1
    j = end
    while i <= j:
        while i <= j and lst[i] <= pivot:
            i += 1
        while i <= j and lst[j] >= pivot:
            j -= 1
        if i <= j:
            lst[i], lst[j] = lst[j], lst[i]

    lst[start], lst[j] = lst[j], lst[start]
    return j

def quick_sort(lst):
    """Quick Sort (original v1.0)"""
    def sort(start, end):
        if start < end:
            pivot = lst[start]
```

```
20
21              pivot_index = partition(lst, start, end, pivot)
22              sort(start, pivot_index - 1)
23              sort(pivot_index + 1, end)
24
25      sort(0, len(lst) - 1)
26      return lst
```

8.7.3 随机选择基准值

快速排序利用分治法，通过一趟排序将数组分为两部分，其中一部分小于等于基准值，另一部分大于等于基准值，然后再递归对两个子问题排序。

基本的快速排序采用序列的第一个元素作为基准值，但是这不是一种好方法。当数组已经有序时，这样的分割效率非常糟糕。为了缓解这种极端情况，可以在待排序数组中随机选择一个元素作为基准值。

随机选取基准值

```
1  def quick_sort(lst):
2      """Quick Sort (optimized v2.0)"""
3      def sort(start, end):
4          if start < end:
5              pivot_index = random.randint(start, end)
6              lst[start], lst[pivot_index] = lst[pivot_index], lst[start]
7              pivot = lst[start]
8
9              pivot_index = partition(lst, start, end, pivot)
10             sort(start, pivot_index - 1)
11             sort(pivot_index + 1, end)
12
13     sort(0, len(lst) - 1)
14     return lst
```

8.7.4 三数取中

虽然随机选取基准值可以减少出现分割不好的几率，但是最坏情况下还是 $O(n^2)$。另一种选取基准值的方法就是三数取中，也就是取序列中 start、mid、end 三个元素的中间值作为基准值。

三数取中

```python
def median_of_three(lst, start, end):
    mid = start + (end - start) // 2

    if lst[start] > lst[mid]:
        lst[start], lst[mid] = lst[mid], lst[start]
    if lst[start] > lst[end]:
        lst[start], lst[end] = lst[end], lst[start]
    if lst[mid] > lst[end]:
        lst[mid], lst[end] = lst[end], lst[mid]

    return mid

def quick_sort(lst):
    """Quick Sort (optimized v2.1)"""
    def sort(start, end):
        if start < end:
            pivot_index = median_of_three(lst, start, end)
            lst[start], lst[pivot_index] = lst[pivot_index], lst[start]
            pivot = lst[start]

            pivot_index = partition(lst, start, end, pivot)
            sort(start, pivot_index - 1)
            sort(pivot_index + 1, end)

    sort(0, len(lst) - 1)
    return lst
```

8.7.5 三数取中 + 插入排序

对于很小和部分有序的数组，快速排序的效率不如插入排序。因此当待排序数组被分割到一定大小后，可直接采用插入排序。

三数取中 + 插入排序

```
def quick_sort(lst):
    """Quick Sort (optimized v2.2)"""
    def sort(start, end):
        if end - start <= INSERTION_SORT_THRESHOLD:
            lst[start:end + 1] = insertion_sort(lst[start:end + 1])
        else:
            pivot_index = median_of_three(lst, start, end)
            lst[start], lst[pivot_index] = lst[pivot_index], lst[start]
            pivot = lst[start]

            pivot_index = partition(lst, start, end, pivot)
            if pivot_index is not None:
                sort(start, pivot_index - 1)
                sort(pivot_index + 1, end)

    sort(0, len(lst) - 1)
    return lst
```

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  def counting_sort(lst):
2      if len(lst) == 0:
3          return lst
4
5      max_val = max(lst)
6      min_val = min(lst)
7
8      counts = [0] * (max_val - min_val + 1)
```

```
    for elem in lst:
        counts[elem - min_val] += 1

    lst.clear()
    for i in range(len(counts)):
        lst.extend([i + min_val] * counts[i])

    return lst
```

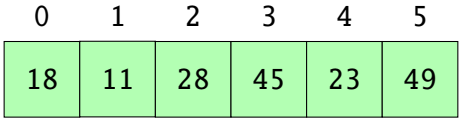
8.9 桶排序

8.9.1 桶排序（Bucket Sort）

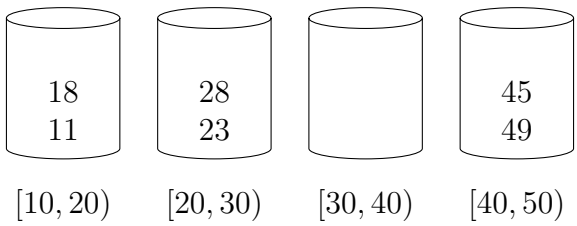
桶排序是计数排序的扩展版本。计数排序可以看成每个桶只存储相同元素，而桶排序每个桶存储一定范围的元素。

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。

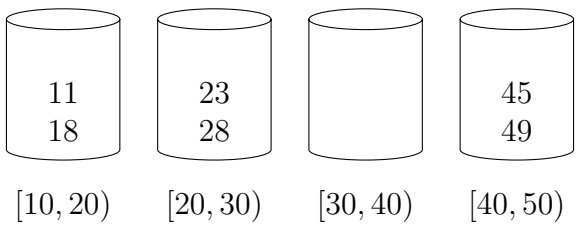
例如一个待排序的序列：



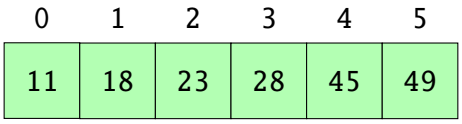
确定桶的个数与每个桶的取值范围，遍历待排序序列，将元素放入对应的桶中：



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

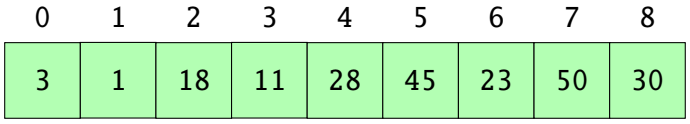
```python
def bucket_sort(lst):
    if len(lst) == 0:
        return lst

    max_val = max(lst)
    min_val = min(lst)
    bucket_size = (max_val - min_val) // len(lst) + 1

    buckets = [[] for _ in range(len(lst))]

    for elem in lst:
        index = (elem - min_val) // bucket_size
        buckets[index].append(elem)

    lst.clear()
    for bucket in buckets:
        bucket.sort()
        lst.extend(bucket)

    return lst
```

8.10 基数排序

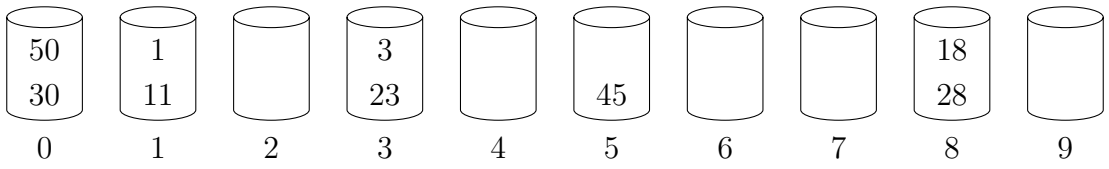
8.10.1 基数排序（Radix Sort）

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。

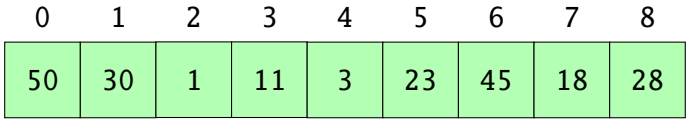
例如一个待排序的序列：



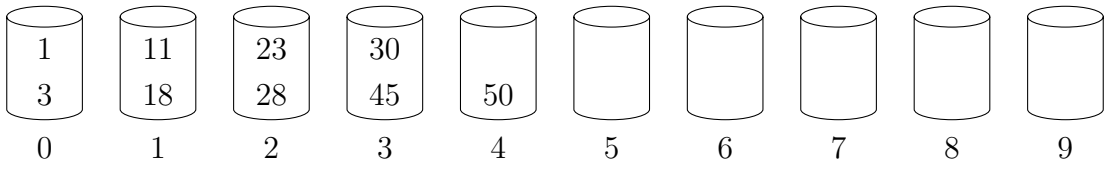
根据个位数值放入对应的桶中：



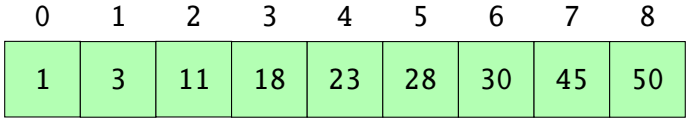
将桶中的元素按顺序赋值到原始数组中：



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：



8.11 猴子排序

8.11.1 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 8.11: 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。

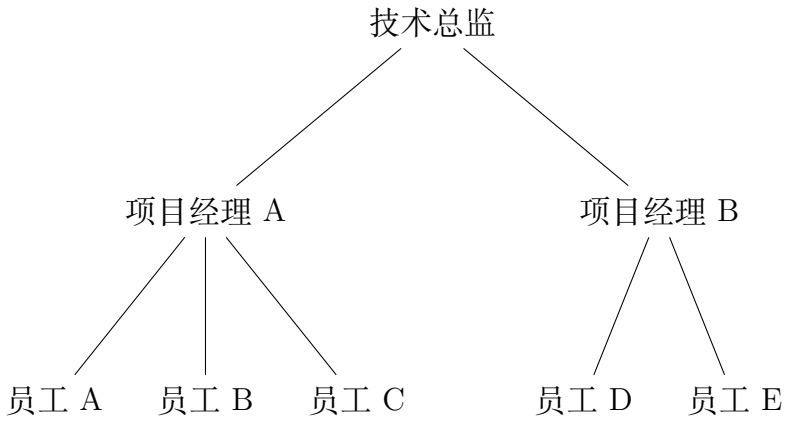


图 9.1: 职级关系

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

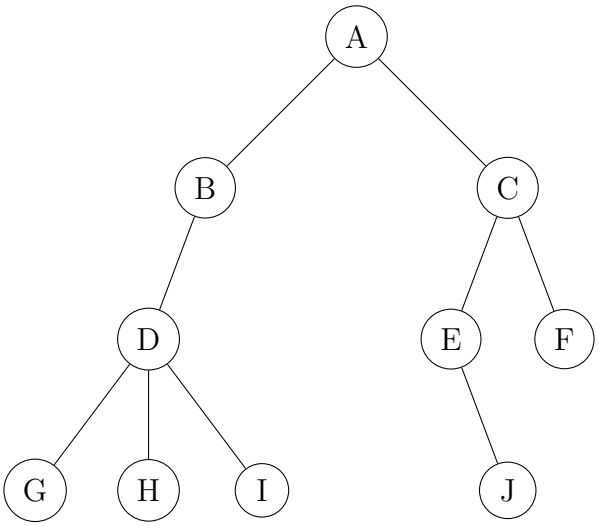


图 9.2: 树

- 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。

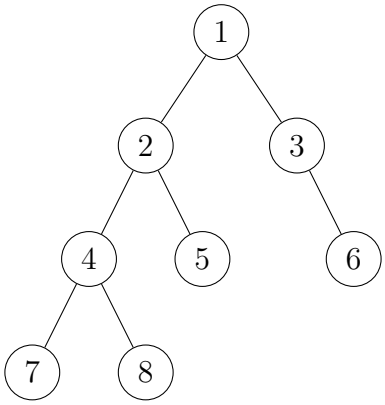


图 9.3: 二叉树

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。

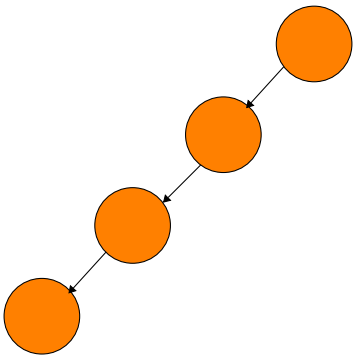


图 9.4: 左斜树

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。

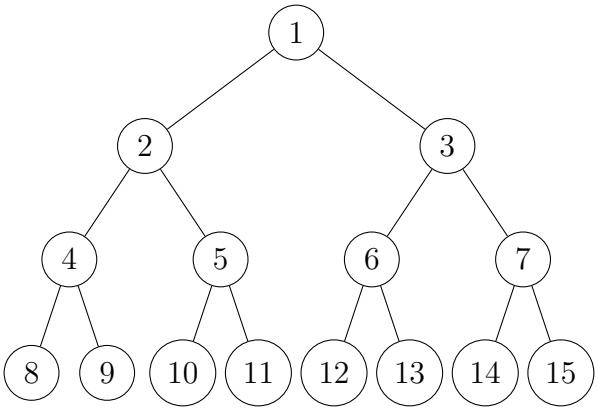


图 9.5: 满二叉树

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。

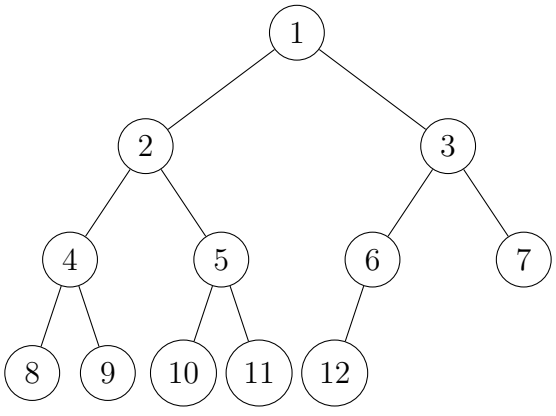


图 9.6: 完全二叉树

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

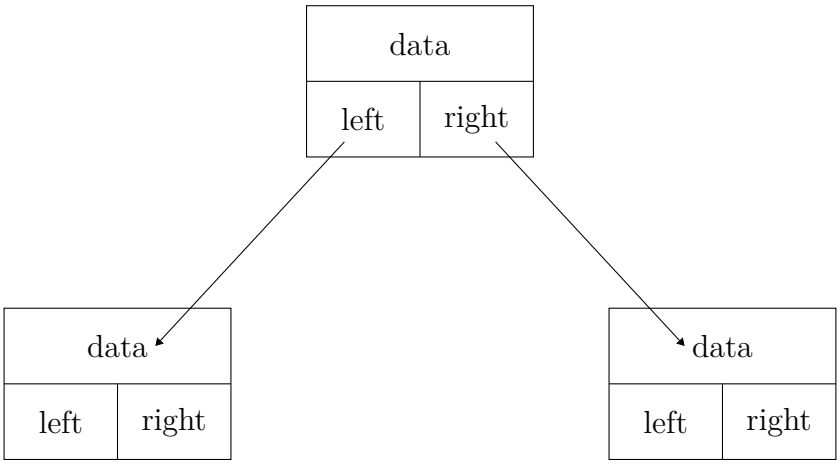


图 9.7: 链式存储结构

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。

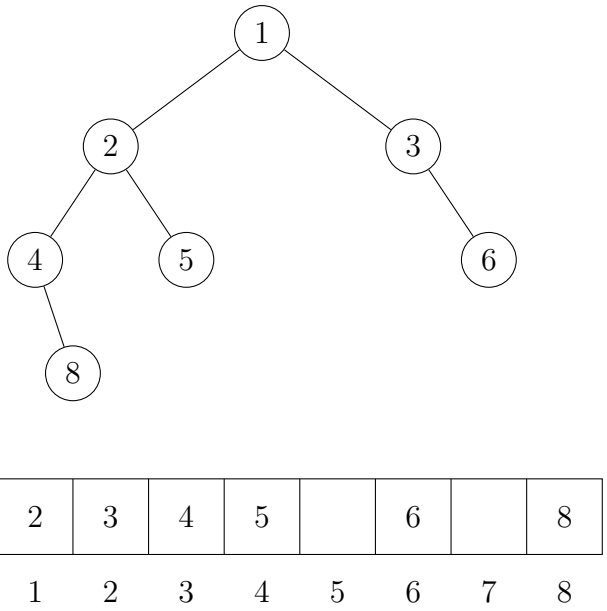


图 9.8: 数组存储

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。

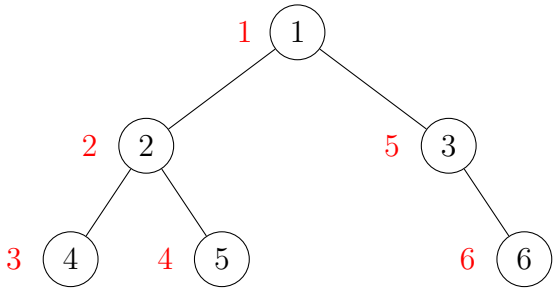


图 9.9: 前序遍历

前序遍历

```
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。

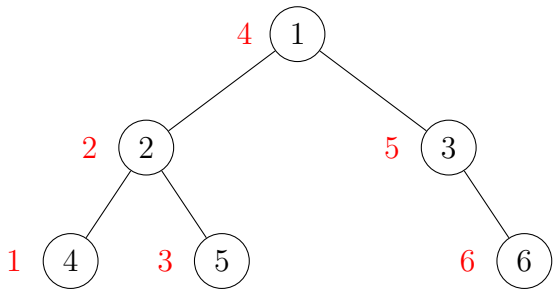


图 9.10: 中序遍历

中序遍历

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

```
8 }
```

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。

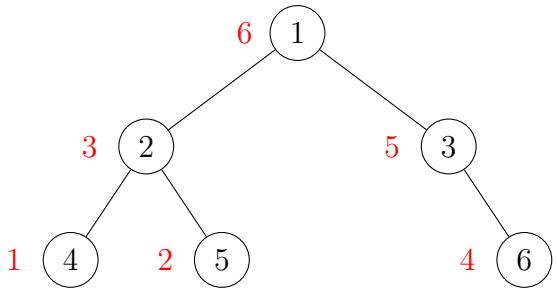


图 9.11: 后序遍历

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;
    }

```

```
6       Queue q = new Queue();
7       q.enqueue(node);
8       while(!q.empty()) {
9           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。

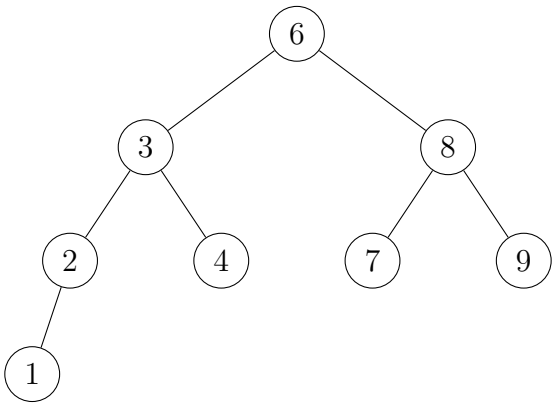


图 9.12: 二叉搜索树

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。

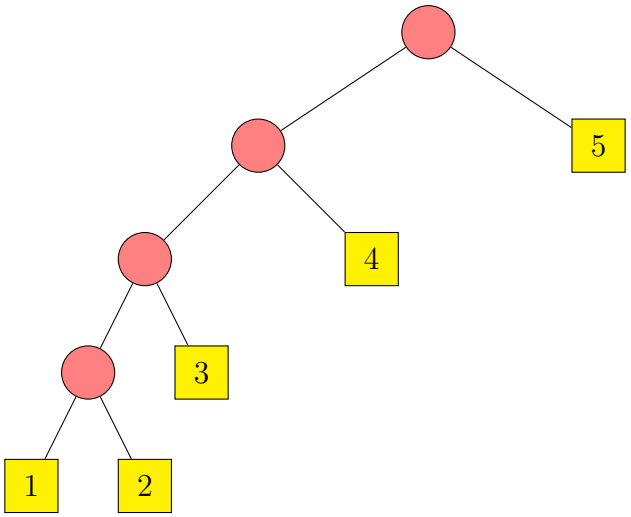


图 9.13: WPL = 5 * 1 + 4 * 2 + 3 * 3 + 2 * 4 + 1 * 4 = 34

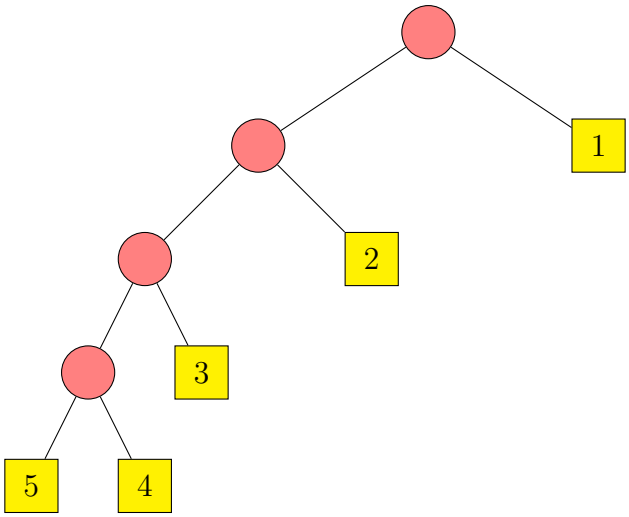


图 9.14: WPL = 1 * 1 + 2 * 2 + 3 * 3 + 4 * 4 + 5 * 4 = 50

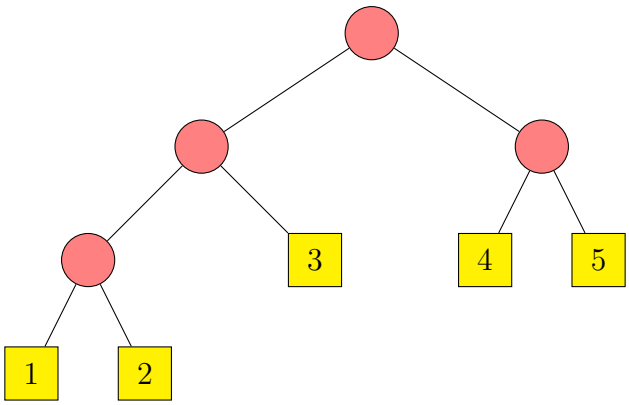


图 9.15: WPL = 3 * 2 + 4 * 2 + 5 * 2 + 1 * 3 + 2 * 3 = 33

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。

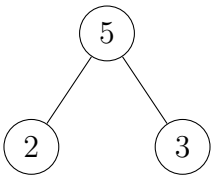


图 9.16: 合并 2 和 3

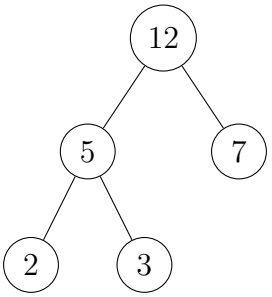


图 9.17: 合并 5 和 7

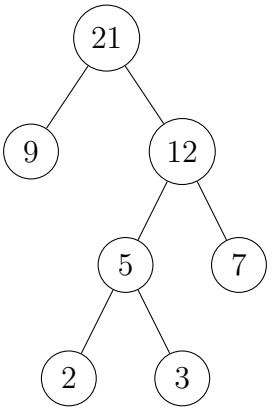


图 9.18: 合并 9 和 12

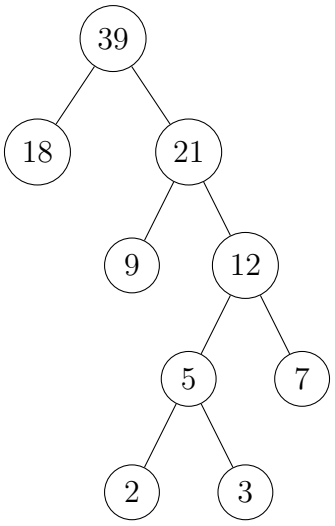


图 9.19: 合并 18 和 21

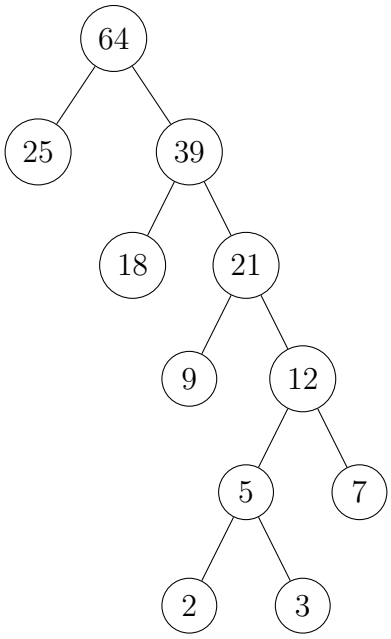


图 9.20: 合并 25 和 39

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。

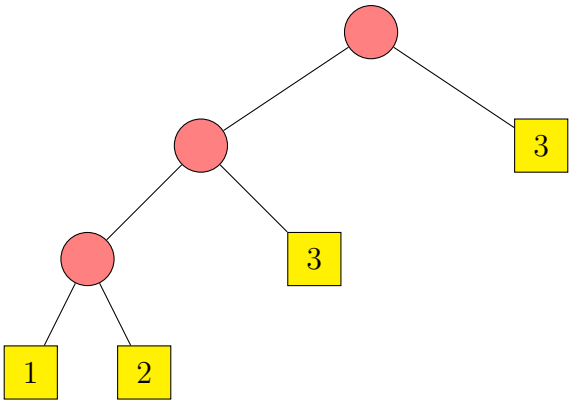


图 9.21: WPL = 3 * 1 + 3 * 2 + 1 * 3 + 2 * 3 = 18

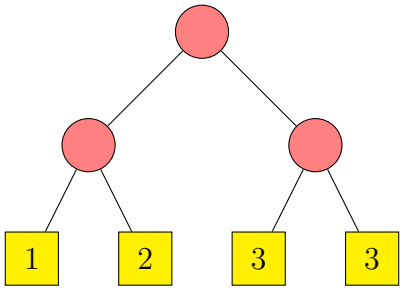


图 9.22: WPL = 1 * 2 + 2 * 2 + 3 * 2 + 3 * 2 = 18

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。

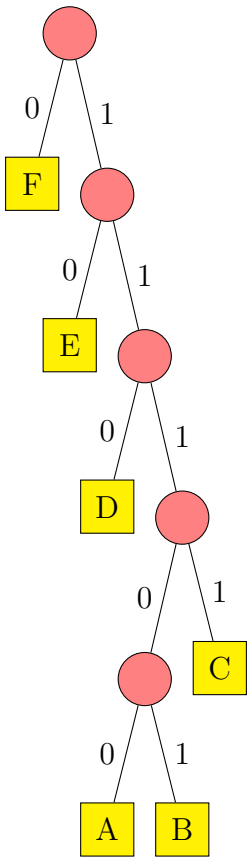
使用不等长编码，让出现频率高的字符用的编码短一些，出现频率低的字符编码长一些，可以使编码的总长度减小。但是不等长编码是不能随意设计的，如果一个字符的编码恰好是另一个字符编码的前缀，就会产生歧义的问题。

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。

9.7 堆排序

9.7.1 堆（Heap）

二叉堆本质上是一种完全二叉树，分为最大堆和最小堆两个类型。在最大堆中，任何一个父结点的值都大于等于它左右孩子结点的值。在最小堆中，任何一个父结点的值都小于等于它左右孩子结点的值。

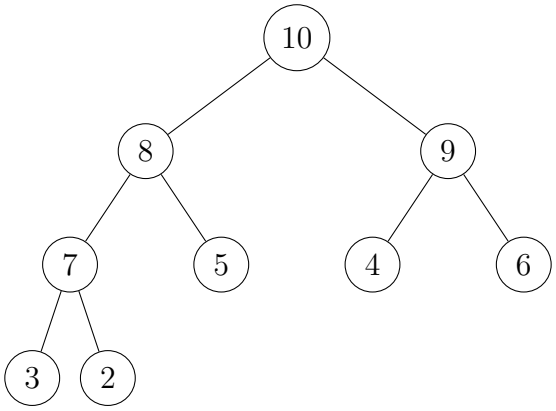


图 9.23: 大顶堆

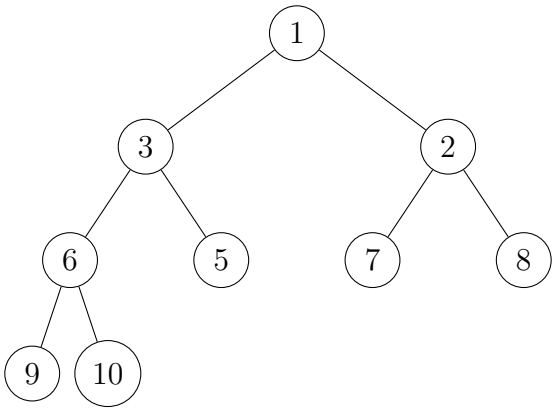


图 9.24: 小顶堆

二叉堆的根结点称为堆顶，在最大堆中堆顶是整个堆中的最大元素，在最小堆中堆顶是整个堆中的最小元素。

在二叉堆中插入结点、删除结点、构造二叉堆的操作都基于堆的自我调整。

二叉堆虽然是一棵完全二叉树，但它的存储方式并不是链式存储，而是顺序存储。

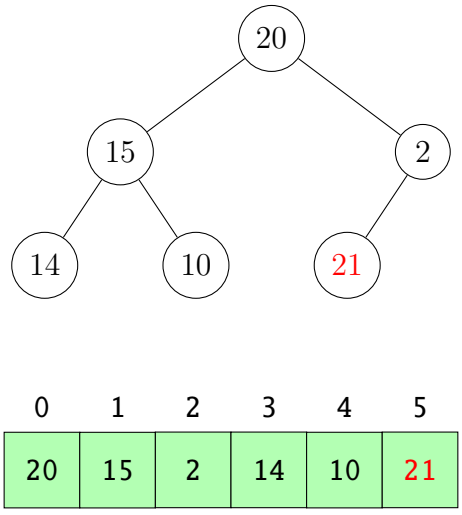
数组中，通过下标可以定位到结点的左右孩子，假设父结点的下标是 parent，那么它的左孩子下标为 2 * parent + 1、右孩子下标为 2 * parent + 2。

9.7.2 插入结点

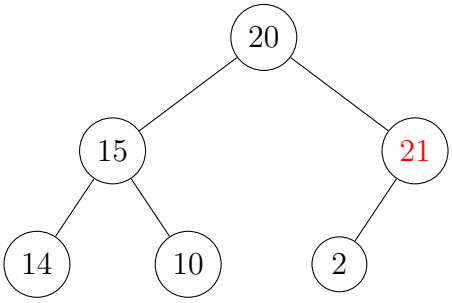
二叉堆的插入操作可以看成是结点上浮，当在堆中插入一个结点时，必须满足完全二叉树的标准，那么被插入结点的位置是完全二叉树的最后一个位置。在最大堆中，如果新结点的值大于它的父结点的值，则让新结点上浮，即和父结点交换位置。

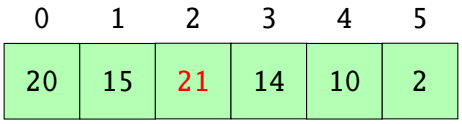
堆的插入时间复杂度取决于树高为 $O(logn)$。

例如在大顶堆 {20, 15, 2, 14, 10} 中插入 21：

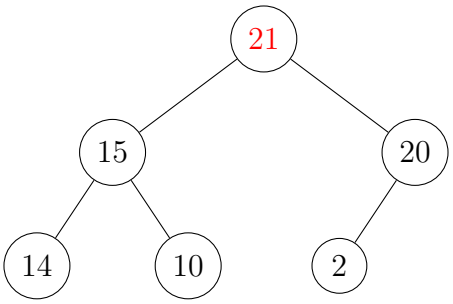


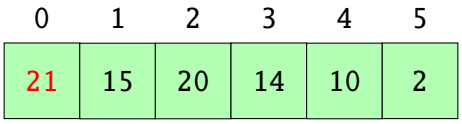
将新元素 21 与其父结点 2 比较，因为 21 > 2，将 21 和 2 的位置交换：





因为 21 > 20，将 21 与 20 的位置交换：



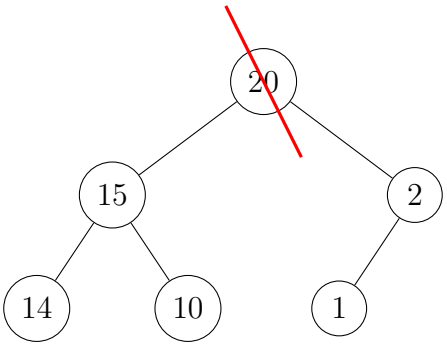


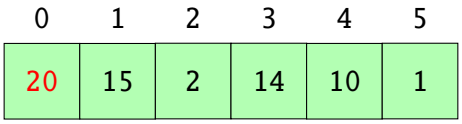
9.7.3 删除结点

二叉堆的删除操作总是从堆的根结点删除元素。根结点被删除之后为了能够保证该树还是一棵完全二叉树，需要将完全二叉树的最后一个结点补到根结点的位置，让其继续符合完全二叉树的定义。二叉堆的删除结点操作可以看作是结点下沉。在最大堆中，如果新堆顶元素小于它的左右孩子中较大的那个结点，则与它的较大的子结点交换位置。

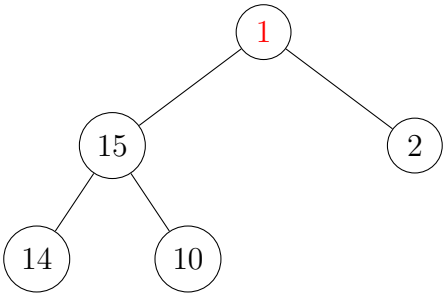
堆的删除时间复杂度取决于树高为 $O(logn)$。

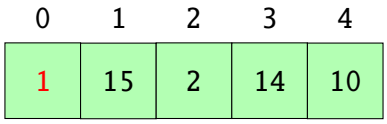
例如删除大顶堆的堆顶元素 20：



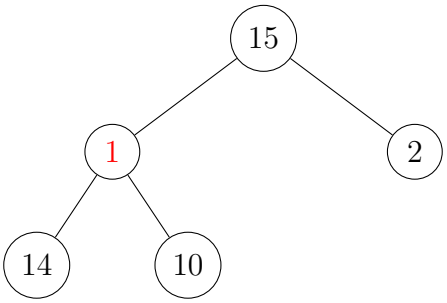


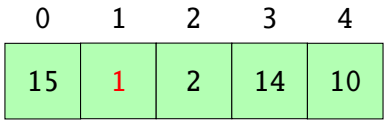
移动最后一个结点到堆顶，使其满足二叉树的性质：



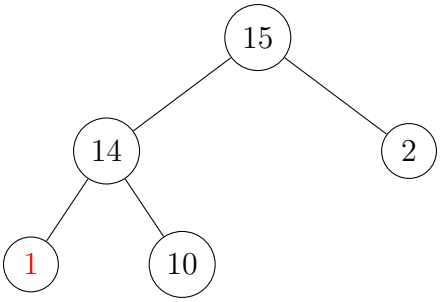


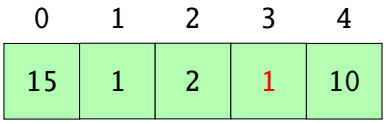
将堆顶元素 1 与其子结点比较，因为 $15 > 2$，交换较大子结点 15 与 1 的位置：





继续将元素 1 与其子结点比较，因为 $14 > 10$，交换较大子结点 14 与 1 的位置：

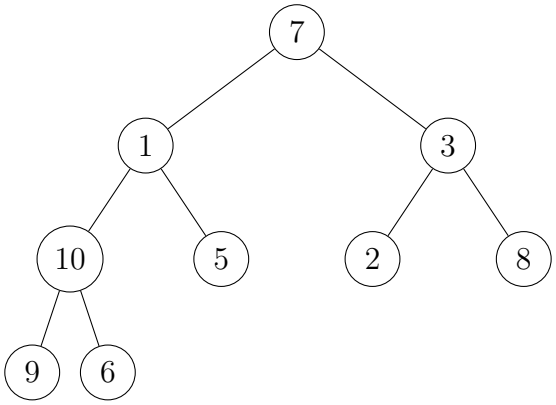




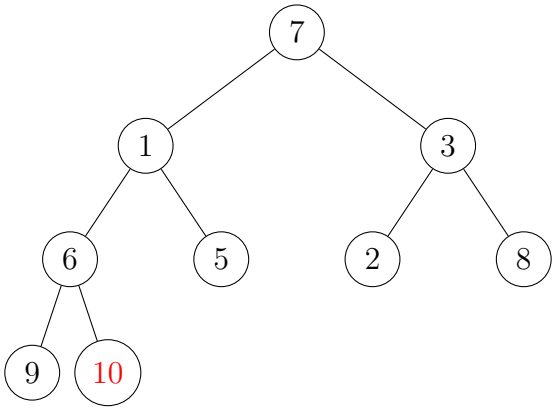
9.7.4 构建二叉堆

构建二叉堆，就是把一个无序的完全二叉树调整为二叉堆，本质上就是让所有非叶子结点依次下沉。

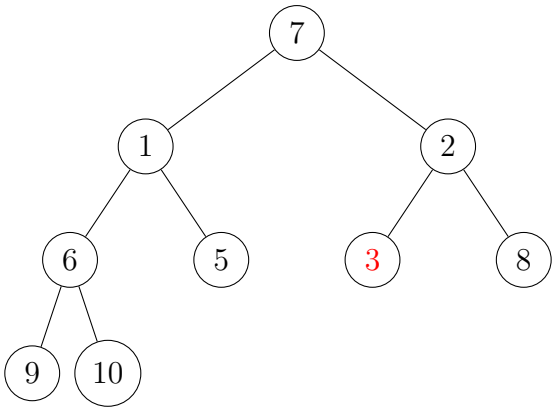
例如将一个无序的完全二叉树构建成最小堆：



首先从最后一个非叶子结点开始，结点 10 下沉：

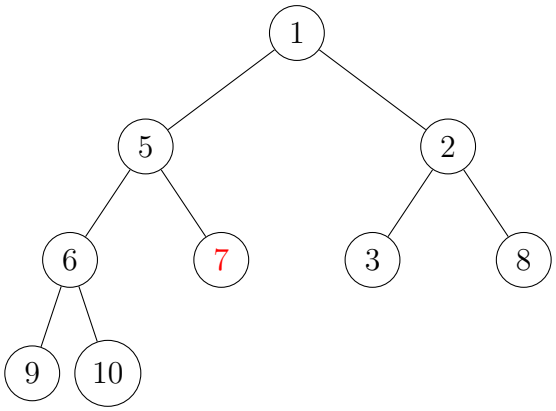


接着处理倒数第二个非叶子结点，结点 3 下沉：



倒数第三个非叶子结点 1 无需移动。

最后处理倒数第四个非叶子结点 7 下沉：

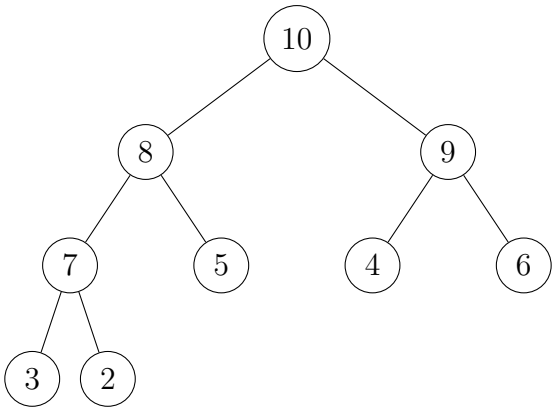


最终一棵无序完全二叉树就调整成了一个最小堆。

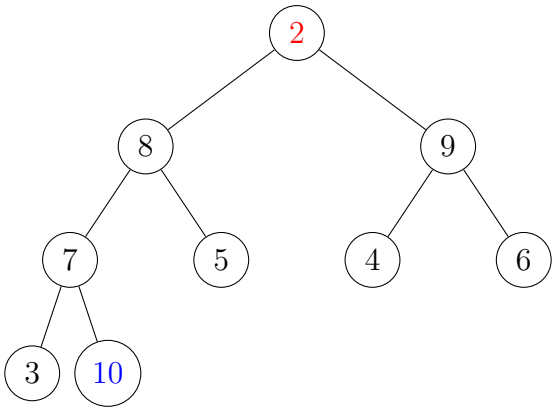
9.7.5 堆排序（Heap Sort）

有了二叉堆的构建、删除和自我调节，实现堆排序就是水到聚成了。当删除一个最大堆的堆顶后（并不是完全删除，而是替换到堆的最后面），经过自我调节，第二大的元素就会被交换上来，成为最大堆的新堆顶。

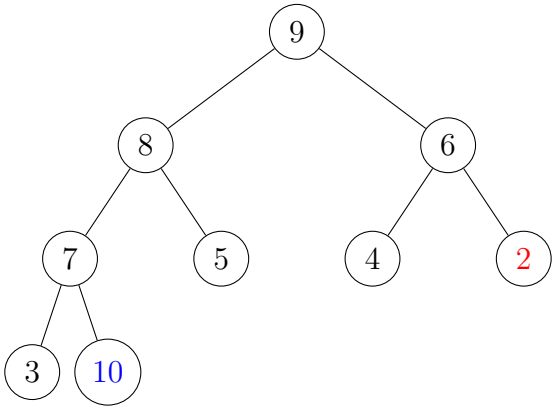
首先将待排序数组构建成大顶堆：



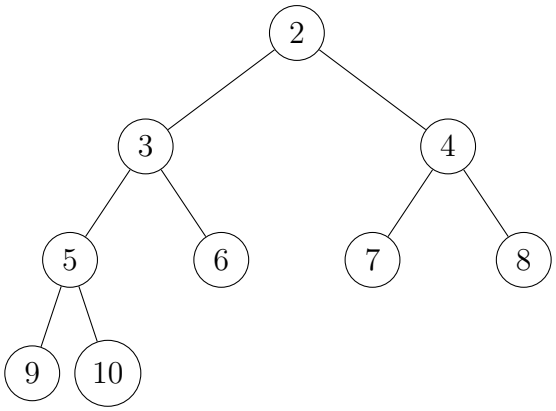
移除堆顶元素（与最后元素交换）：



将新的堆顶元素进行下沉，重新调整为大顶堆：



只要反复删除堆顶，反复调节二叉堆，所得到的的集合就成为了一个有序集合。



堆排序

```java
public static void downAdjust(int[] arr, int parentIndex, int len) {
    // 保存父结点的值，用于最后的赋值
    int temp = arr[parentIndex];
    int childIndex = 2 * parentIndex + 1;

    while(childIndex < len) {
        // 如果有右孩子，且右孩子大于左孩子的值，则定位到右孩子
        if(childIndex + 1 < len
            && arr[childIndex + 1] > arr[childIndex]) {
            childIndex++;
        }
        // 如果父结点小于任何一个孩子的值，直接跳出
        if(temp >= arr[childIndex]) {
            break;
        }
        // 无需真正交换，单向赋值即可
        arr[parentIndex] = arr[childIndex];
        parentIndex = childIndex;
        childIndex = 2 * childIndex + 1;
    }
    arr[parentIndex] = temp;
}

public static void heapSort(int[] arr) {
    // 把无序数组构建成二叉堆
```

```
26      for(int i = (arr.length-2) / 2; i >= 0; i--) {
27          downAdjust(arr, i, arr.length);
28      }
29
30      // 循环删除堆顶元素，移到数组尾部，调节堆产生新的堆顶
31      for(int i = arr.length - 1; i > 0; i--) {
32          // 最后一个元素和第一个元素交换
33          int temp = arr[i];
34          arr[i] = arr[0];
35          arr[0] = temp;
36          // 下沉调整最大堆
37          downAdjust(arr, 0, i);
38      }
39  }
```

堆排序的空间复杂度为 $O(1)$，因为算法并没有开辟额外的集合空间。

至于空间复杂度，假设二叉堆总共有 n 个元素，那么下沉调整的最坏时间复杂度就等同于二叉堆的高度 $O(logn)$。

堆排序的算法步骤分为两部分：

1. 把无序数组构建成二叉堆：进行 $n/2$ 次循环，每次循环进行一次下沉调节，因为此步骤的计算规模为 $n/2 * logn$，时间复杂度为 $O(nlogn)$。
2. 循环删除堆顶元素，移到数组尾部，调节堆产生新堆顶：进行 n - 1 次循环，每次循环进行一次下沉调节，因此次步骤的计算规模为 $(n-1) * logn$，时间复杂度为 $O(nlogn)$。

综合堆排序的两个步骤，整体时间复杂度为 $O(nlogn)$。

9.8 AVL 树

9.8.1 AVL 树

AVL 树的命名是取自两位发明者的首字母 G. M. Adelson-Velsky 和 E. M. Landis，AVL 树也称为平衡二叉树。AVL 树能够调整自身的平衡性，AVL 树遵循高度平衡，任何结点的两个子树高度差不会超过 1。

对于 AVL 树的每一个结点，平衡因子（balance factor）是它左子树高度和右子树高度的差值。只有当二叉树所有结点的平衡因子都是-1、0、1 这三个值时，这棵树才是一个合格的 AVL 树。

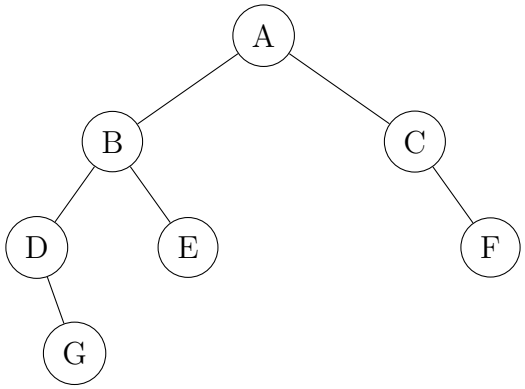


图 9.25: AVL 树

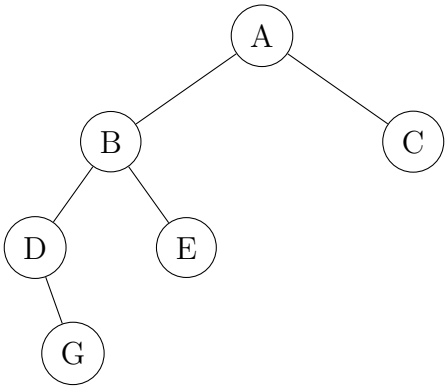


图 9.26: 非 AVL 树

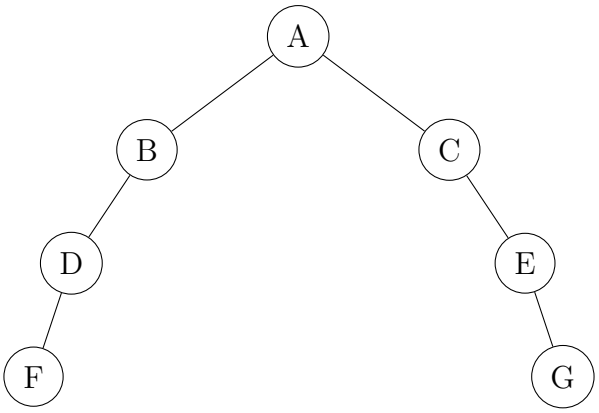


图 9.27: 非 AVL 树

9.8.2 失衡调整

当 AVL 树插入或删除结点时，平衡有可能被打破。

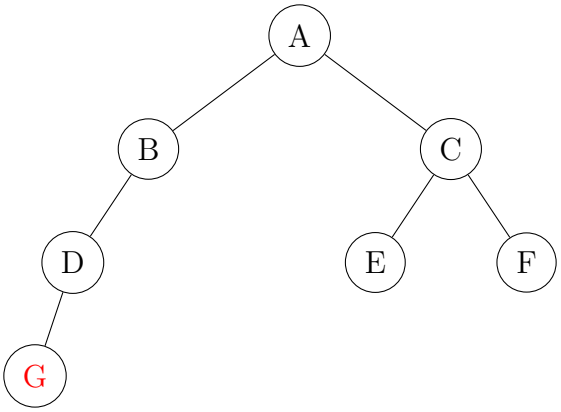


图 9.28: 失衡

通过对 AVL 树进行左旋转、右旋转的操作，就能使其重新恢复平衡。

左旋转

逆时针旋转 AVL 树的两个结点 X 和 Y，使得父结点被自己的右孩子取代，而自己成为左孩子。

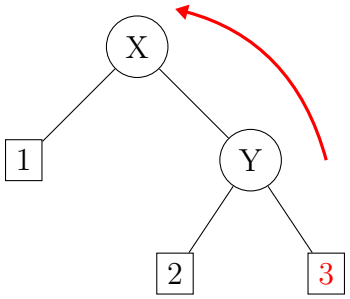


图 9.29: 左旋转（前）

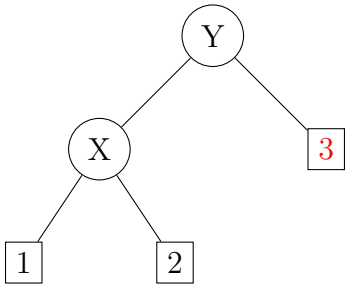


图 9.30: 左旋转（后）

右旋转

顺时针旋转 AVL 树的两个结点 X 和 Y，使得父结点被自己的左孩子取代，而自己成为右孩子。

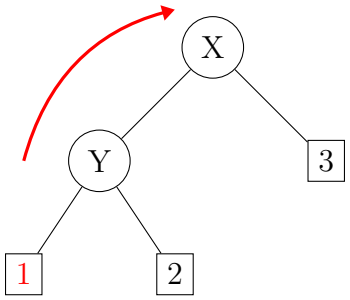


图 9.31: 右旋转（前）

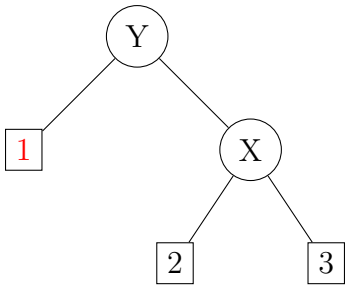


图 9.32: 右旋转（后）

AVL 树的失衡调整可以分为四种情况：

左左局面（LL）：右旋转

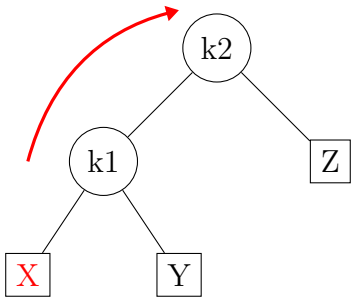


图 9.33: 左左局面

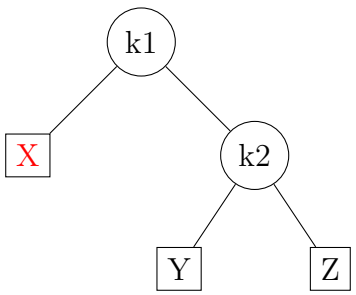


图 9.34: 右旋转

LL 旋转

```
1 static AVLNode* LLRotation(AVLTree *k2) {
2     AVLTree *k1 = k2->left;
3     k2->left = k1->right;
4     k1->right = k2;
```

```
5
6       k2->height = max(height(k2->left), height(k2->right)) + 1;
7       k1->height = max(height(k1->left), k2->height) + 1;
8       return k1;
9   }
```

右右局面（RR）：左旋转

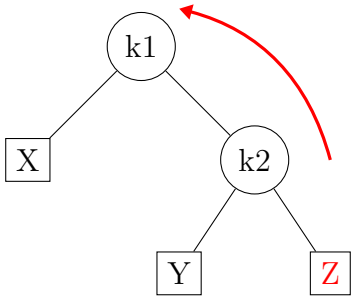


图 9.35: 右右局面

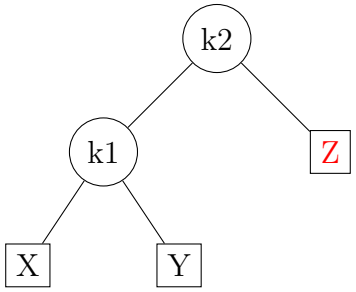


图 9.36: 左旋转

RR 旋转

```
1   static AVLNode* RRRotation(AVLTree *k1) {
2       AVLTree *k2 = k1->right;
3       k1->right = k2->left;
4       k2->left = k1;
5
6       k1->height = max(height(k1->left), height(k1->right)) + 1;
7       k2->height = max(k1->height, height(k2->right)) + 1;
8       return k2;
```

```
9 }
```

左右局面（LR）：先左旋转、再右旋转

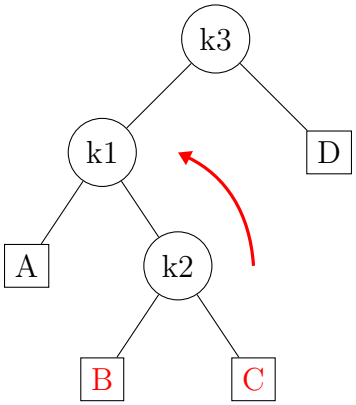


图 9.37: 左右局面

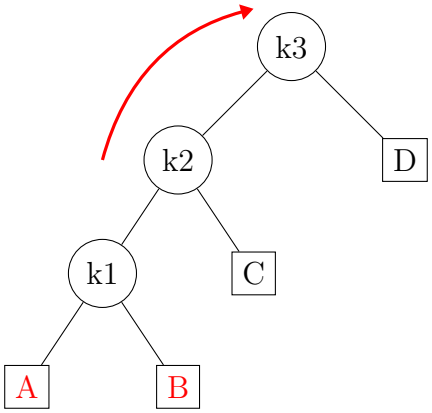


图 9.38: 左旋转

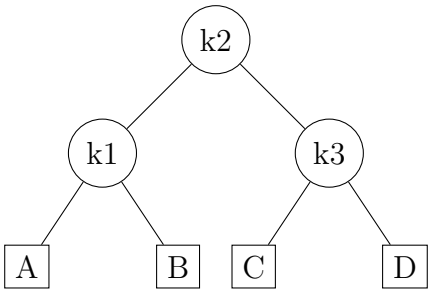


图 9.39: 右旋转

LR 旋转

```
1 static AVLNode* LRRotation(AVLTree *k3) {
2     k3->left = RRRotation(k3->left);
3     return LLRotation(k3);
4 }
```

右左局面（RL）：先右旋转、再左旋转

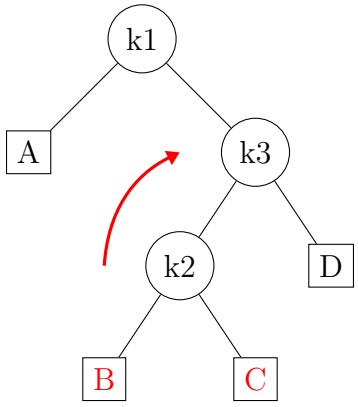


图 9.40: 右左局面

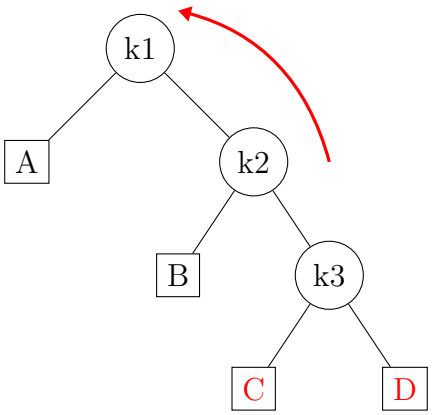


图 9.41: 右旋转

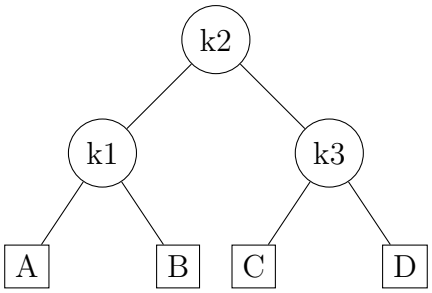


图 9.42: 左旋转

RL 旋转

```
1 static AVLNode* RLRotation(AVLTree *k1) {
2     k1->right = LLRotation(k1->right);
3     return RRRotation(k1);
4 }
```

9.8.3 插入/删除结点

例如依次向 AVL 树添加结点 3, 2, 1, 4, 5, 6, 7, 16, 15, 14。

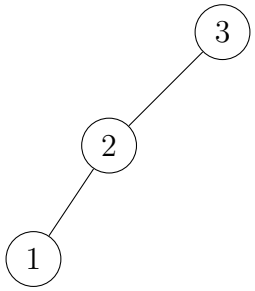


图 9.43: 左左局面：插入 3、2、1

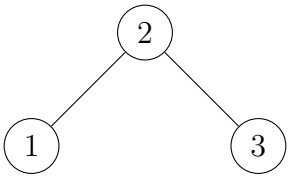


图 9.44: 右旋转

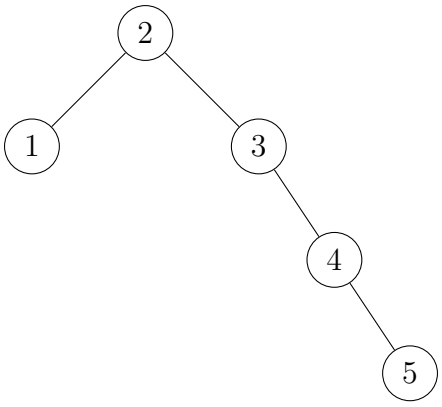


图 9.45: 右右局面：插入 4、5

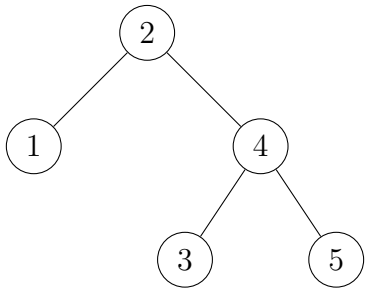


图 9.46: 左旋转

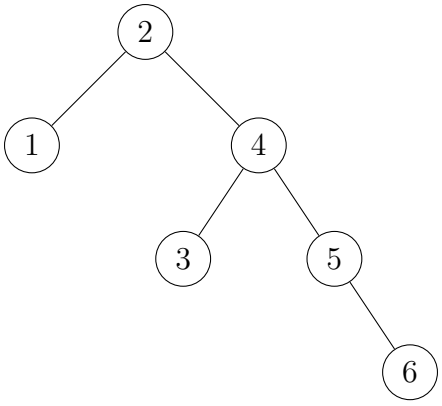


图 9.47: 右右局面：插入 6

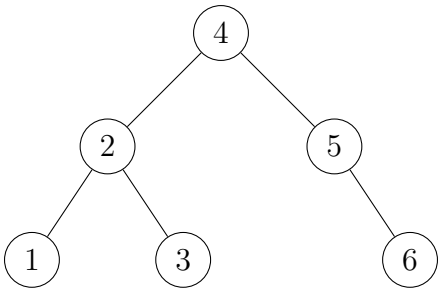


图 9.48: 左旋转

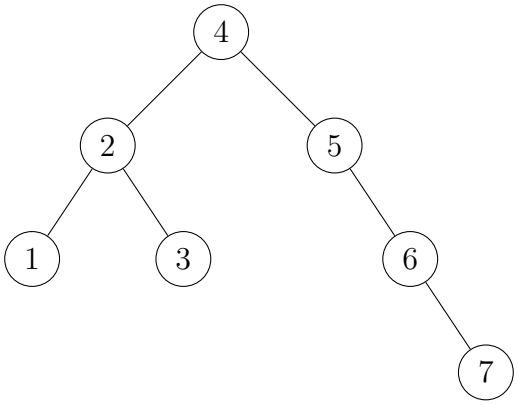


图 9.49: 右右局面：插入 7

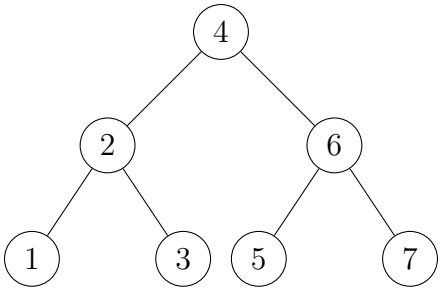


图 9.50: 左旋转

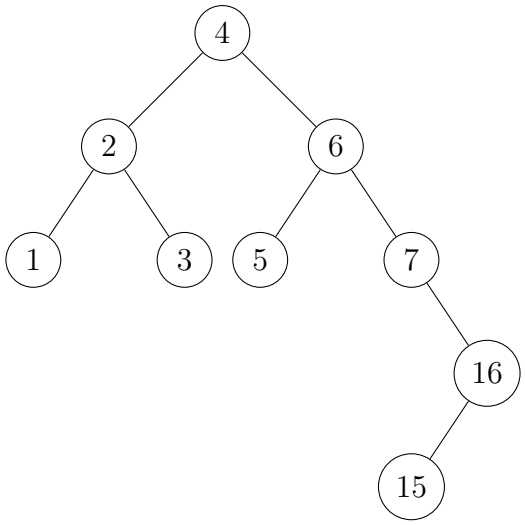


图 9.51: 右左局面：插入 16、15

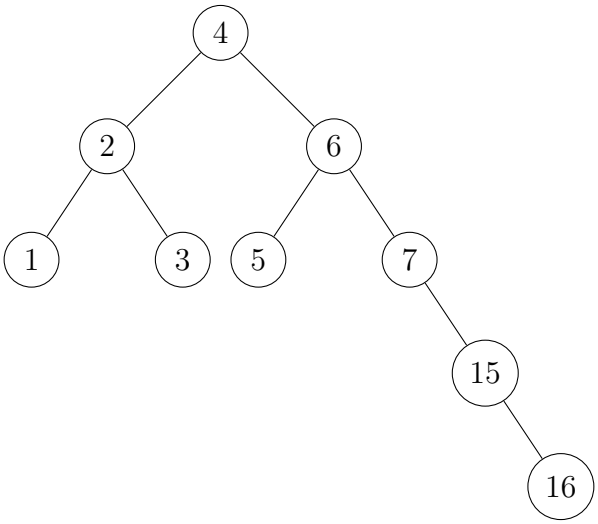


图 9.52: 右旋转

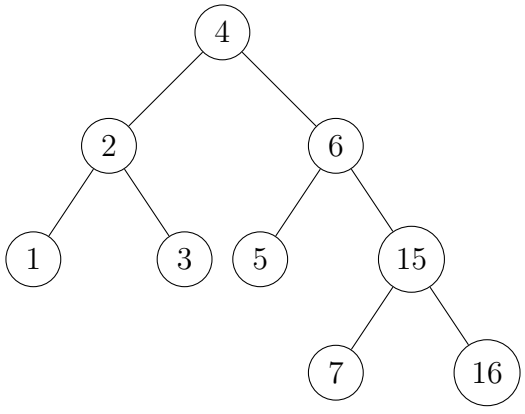


图 9.53: 左旋转

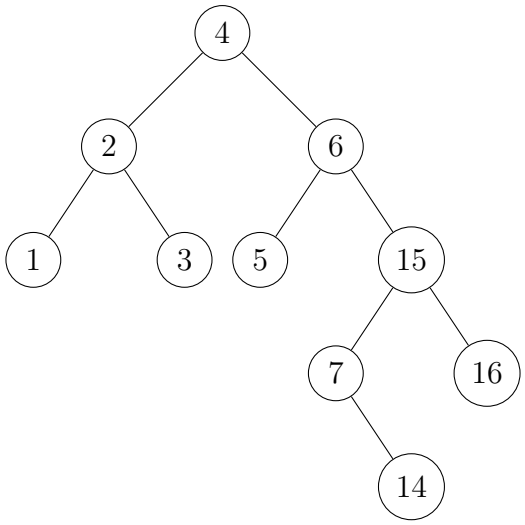


图 9.54: 右左局面：插入 14

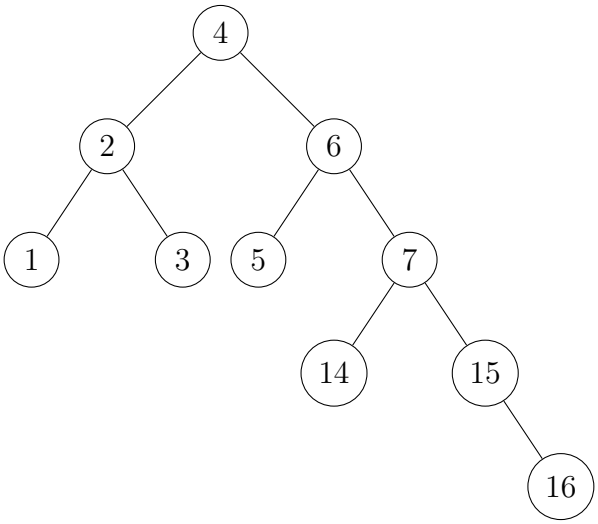


图 9.55: 右旋转

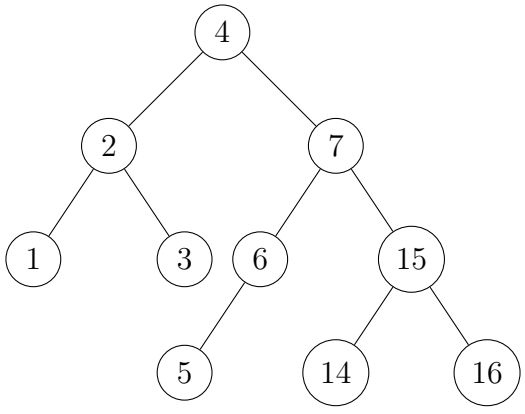


图 9.56: 左旋转

插入结点

```
AVLNode* insert(AVLTree *tree, dataType val) {
    if(!tree) {
        tree = createNode(val, NULL, NULL);
        tree->height = 1;
        return tree;
    }

    if(val < tree->data) {
        tree->left = insert(tree->left, val);
        if(height(tree->left) - height(tree->right) == 2) {
            if(val < tree->left->data) {
                tree = LLRotation(tree);
            } else {
                tree = LRRotation(tree);
            }
        }
    } else {
        tree->right = insert(tree->right, val);
        if(height(tree->right) - height(tree->left) == 2) {
            if(val > tree->right->data) {
                tree = RRRotation(tree);
            } else {
                tree = RLRotation(tree);
            }
        }
    }

    tree->height = max(height(tree->left), height(tree->right)) + 1;
    return tree;
}
```

删除结点

```
static AVLNode* deleteNode(AVLTree *tree, AVLNode *del) {
    if(!tree || !del) {
        return NULL;
    }

    if(del->data < tree->data) {
        tree->left = deleteNode(tree->left, del);
        if(height(tree->right) - height(tree->left) == 2) {
            AVLNode *rightNode = tree->right;
            if(height(rightNode->left) > height(rightNode->right)) {
                tree = RLRotation(tree);
            } else {
                tree = RRRotation(tree);
            }
        }
    } else if(del->data > tree->data) {
        tree->right = deleteNode(tree->right, del);
        if(height(tree->left) - height(tree->right) == 2) {
            AVLNode *leftNode = tree->left;
            if(height(leftNode->right) > height(leftNode->left)) {
                tree = LRRotation(tree);
            } else {
                tree = LLRotation(tree);
            }
        }
    } else {
        if(tree->left && tree->right) {
            if(height(tree->left) > height(tree->right)) {
                // 如果左子树比右子树高:
                // 1. 找出左子树的最大结点
                // 2. 将最大结点的值赋给tree
                // 3. 删除最大结点
                AVLNode *max = getMax(tree->left);
                tree->data = max->data;
                tree->left = deleteNode(tree->left, max);
            } else {
                // 如果右子树比左子树高（或相等）:
```

```
38                  // 1. 找出右子树的最小结点
39                  // 2. 将最小结点的值赋给tree
40                  // 3. 删除最小结点
41                  AVLNode *min = getMin(tree->right);
42                  tree->data = min->data;
43                  tree->right = deleteNode(tree->right, min);
44              }
45          } else {
46              AVLNode *temp = tree;
47              tree = tree->left ? tree->left : tree->right;
48              free(temp);
49          }
50      }
51      return tree;
52  }
```

9.9 红黑树

9.9.1 红黑树（Red Black Tree）

红黑树是一种自平衡的二叉查找树，除了符合二叉查找树的基本特性外，它还具有如下附加特性：

1. 结点是红色或黑色的。
2. 根结点是黑色的。
3. 叶子结点都是黑色的空结点 NIL。
4. 红色结点的两个子结点都是黑色的，即从叶子到根的所有路径上不能有连续的两个红色结点。
5. 从任一结点到其每个叶子的所有路径都包含相同数目的黑色结点。

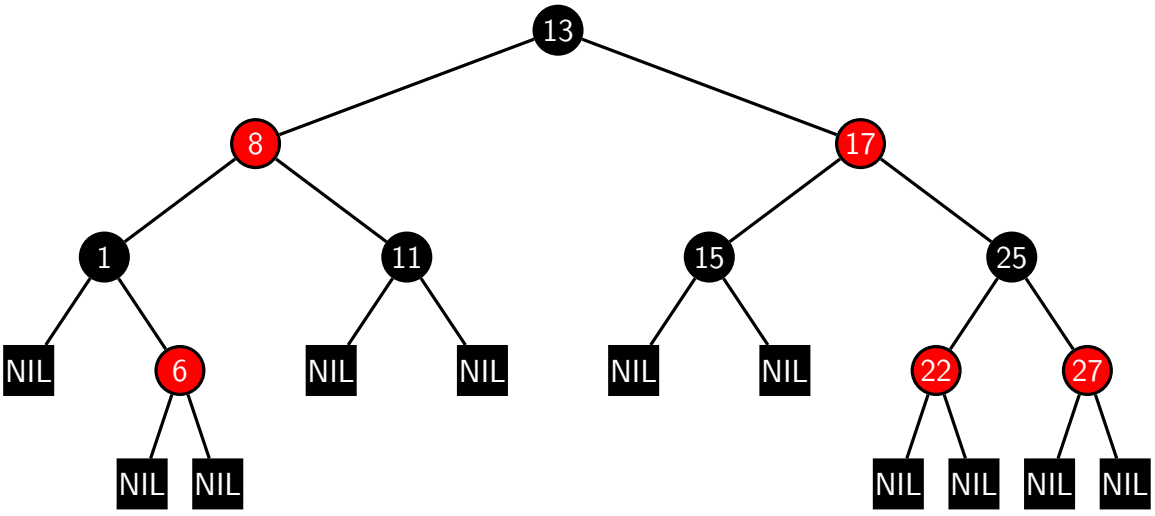


图 9.57: 红黑树

天呐，这条条框框的太多了吧！

正是因为这些规则限制，才保证了红黑树的自平衡，红黑树从根到叶子的最长路径不会超过最短路径的 2 倍。

红黑树的应用有很多，其中 JDK 的集合类 TreeMap 和 TreeSet 底层就是红黑树实现的。在 Java8 中，连 HashMap 也用到了红黑树。

9.9.2 失衡调整

当插入或删除结点时，红黑树的规则可能被破坏，需要调整使其重新符合规则。

例如向红黑树中插入新结点 14，由于父结点 15 是黑色结点，这种情况不会破坏红黑树的规则，无需做任何调整。

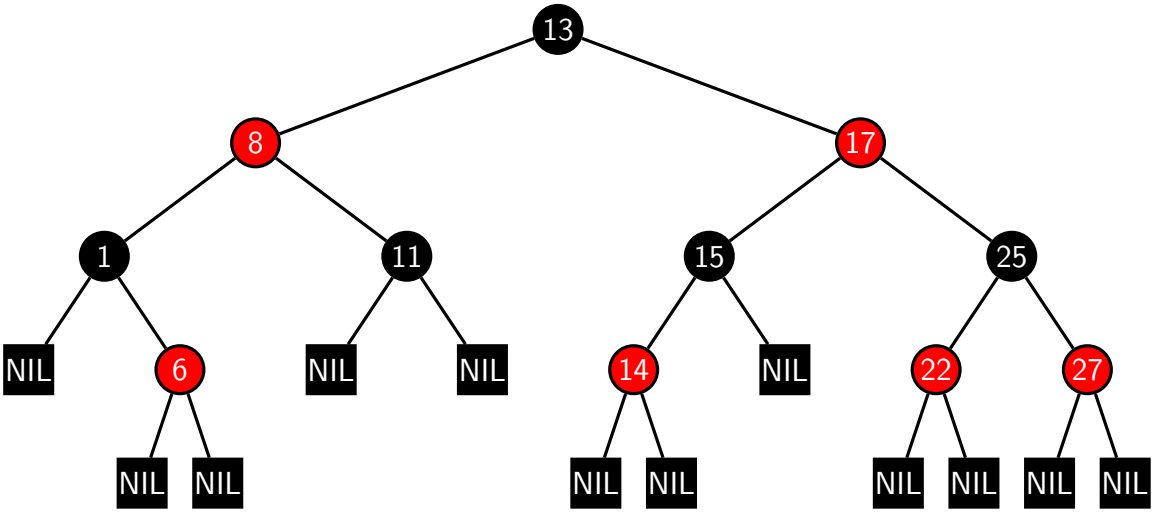


图 9.58: 插入 14

向红黑树中插入新结点 21，由于父结点 22 是红色结点，违反了红黑树的规则 4（红色结点的两个子结点都是黑色的）。

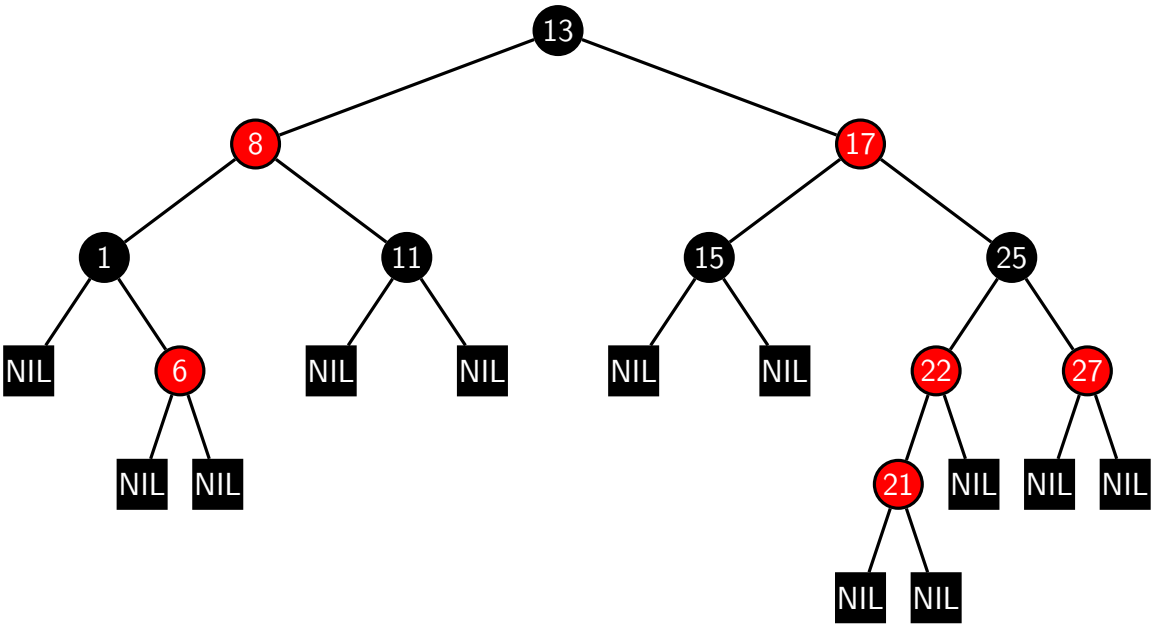


图 9.59: 插入 21

调整的方法有变色和旋转两种，而旋转又包含左旋转和右旋转两种方式。

为了重新符合红黑树的规则，有时需要把红色结点变为黑色，或是把黑色结点变为红色。

例如对于红黑树的一部分（子树），新插入的结点 Y 是红色结点，它的父结点 X 也是红色结点，不符合规则 4（红色结点的两个子结点都是黑色的），因此可以把结点 X 变为黑色。

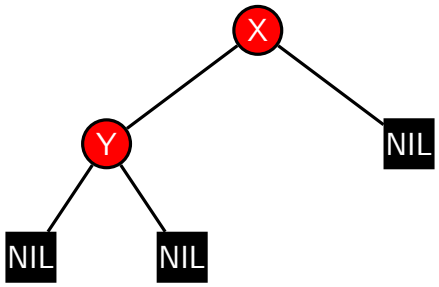


图 9.60: 违反规则 4

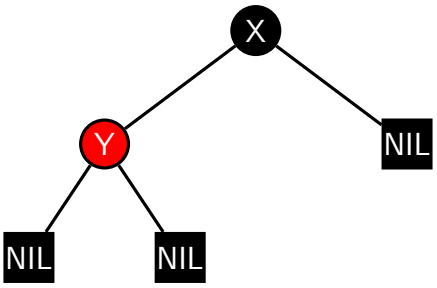


图 9.61: 变色

但是，如果这是简单的把一个结点变色，会导致相关路径凭空多出一个黑色结点，这样就会打破规则 5（从任一结点到其每个叶子的所有路径都包含相同数目的黑色结点），因此还需要其它的调整策略。

9.9.3 红黑树插入结点

红黑树插入新结点时，可以分为五种不同的局面。每一种局面有不同的调整方法。

局面 1

新结点（A）位于树根，没有父结点。

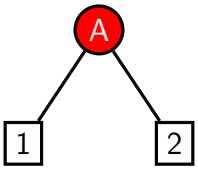
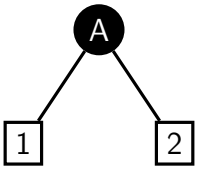


图 9.62: 局面 1

这种局面，直接让新结点变色为黑色，规则 2（根结点是黑色的）满足。同时黑色的根结点使每条路径上的黑色结点数目都增加了 1，因此并没有打破规则 5（从任一结点到其每个叶子的所有路径都包含相同数目的黑色结点）。



局面 2

新结点（B）的父结点是黑色的。新插入的红色结点 B 并没有打破规则，无需调整。

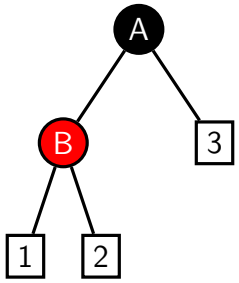


图 9.63: 局面 2

局面 3

新结点（D）的父结点和叔叔结点都是红色。

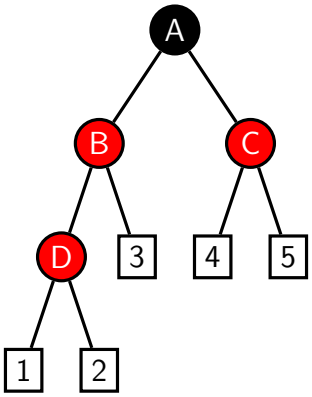
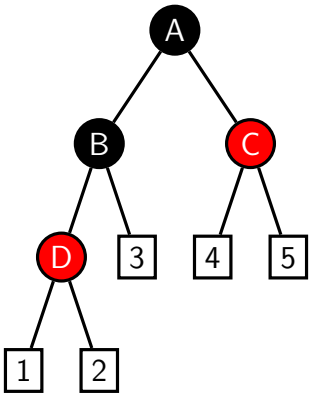
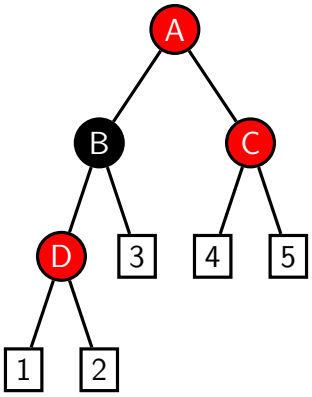


图 9.64: 局面 3

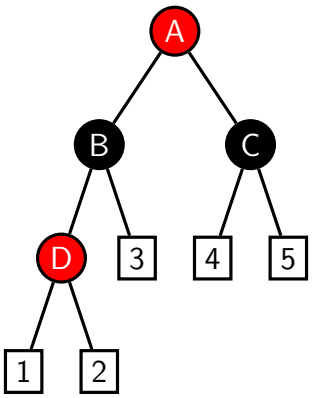
这种局面，两个红色结点 B 和 D 连续，违反了规则 4（红色结点的两个子结点都是黑色的），因此需要先让结点 B 变为黑色。



但是这样一来，结点 B 所在路径凭空多出了一个黑色结点，打破了规则 5（从任一结点到其每个叶子的所有路径都包含相同数目的黑色结点），因此再让结点 A 变为红色。



这时结点 A 和 C 又成为了连续的红色结点，再将结点 C 变为黑色。



局面 4

新结点（D）的父结点是红色，叔叔结点是黑色或者没有叔叔，且新结点是父结点的右孩子，父结点是祖父结点的左孩子。

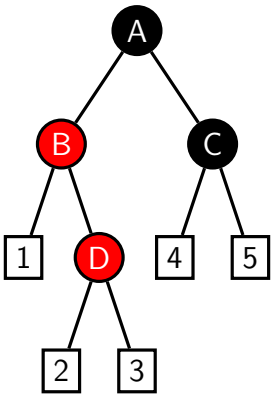
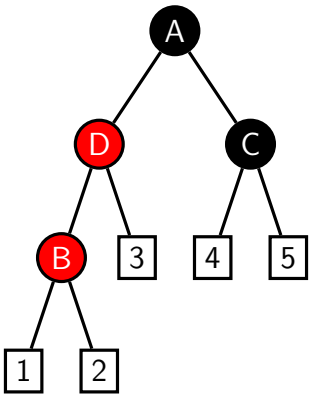


图 9.65: 局面 4

这个局面可以以结点 B 为轴，做一次左旋转，使得新结点 D 成为父结点，结点 B 成为 D 的左孩子。这样一来进入了局面 5。



局面 5

新结点（D）的父结点是红色，叔叔结点是黑色或者没有叔叔，且新结点是父结点的左孩子，父结点是祖父结点的左孩子。

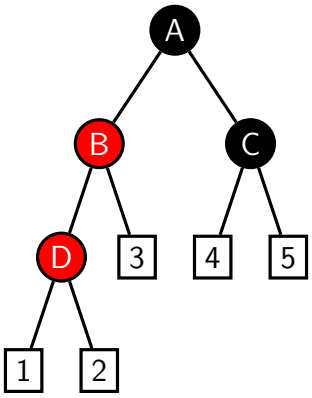
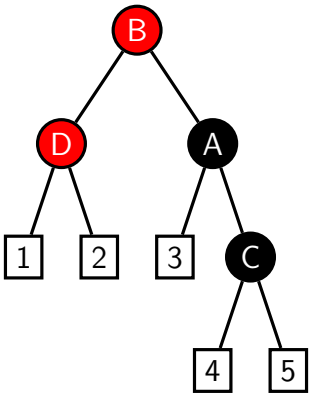
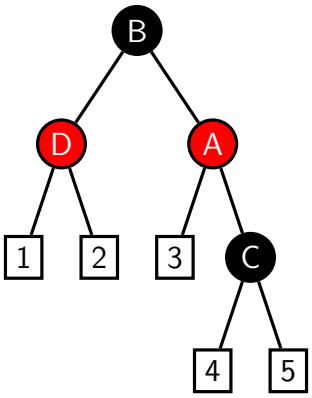


图 9.66: 局面 5

这一局面可以以结点 A 为轴，做一次右旋转，使得结点 B 成为祖父结点，结点 A 成为 B 的右孩子。

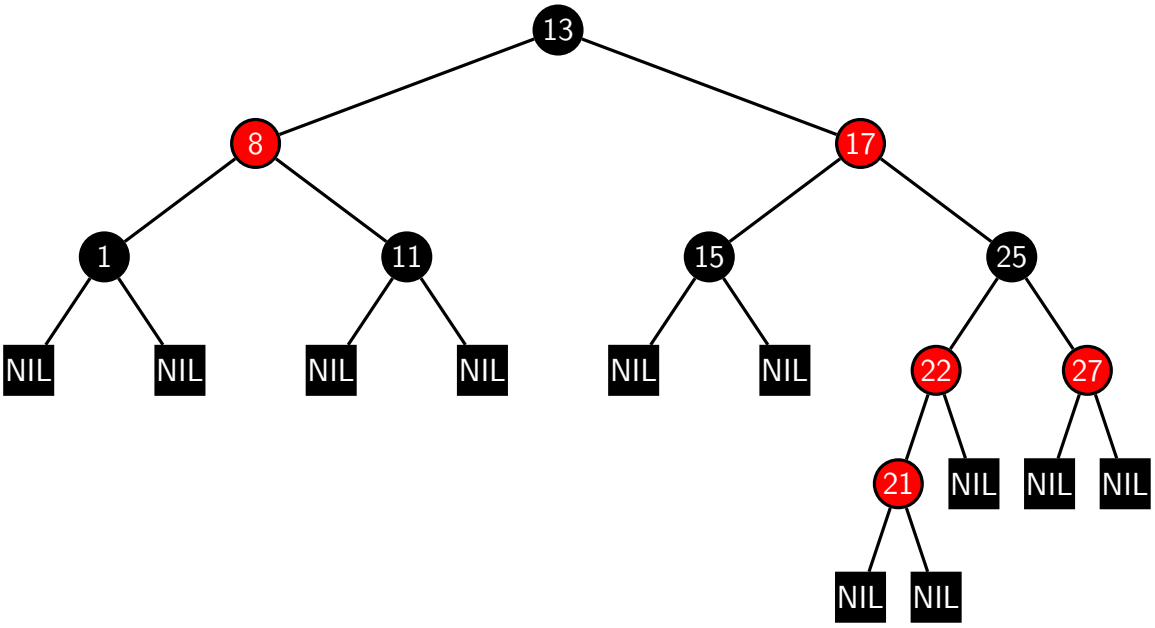


再将结点 B 变为黑色，结点 A 变为红色。

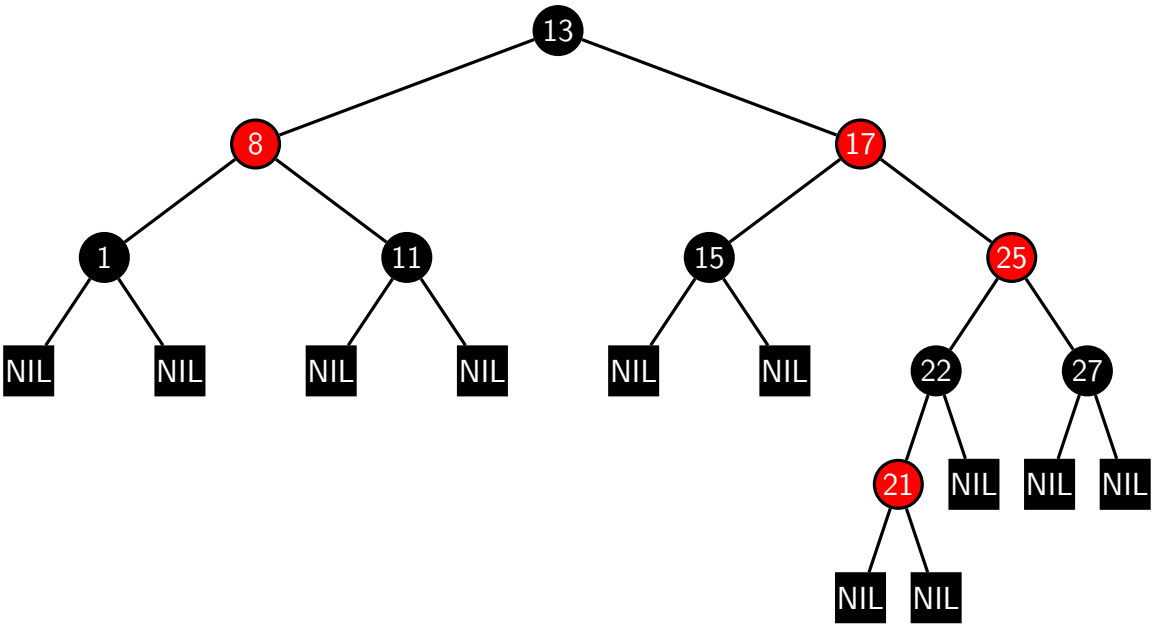


红黑树的插入操作设计到这 5 种局面。如果局面 4 中父结点 B 是右孩子，则成为了局面 5 的镜像，原本的右旋转改为左旋转；如果局面 5 中父结点 B 是右孩子，则成为了局面 4 的镜像，原本的左旋转改为右旋转。

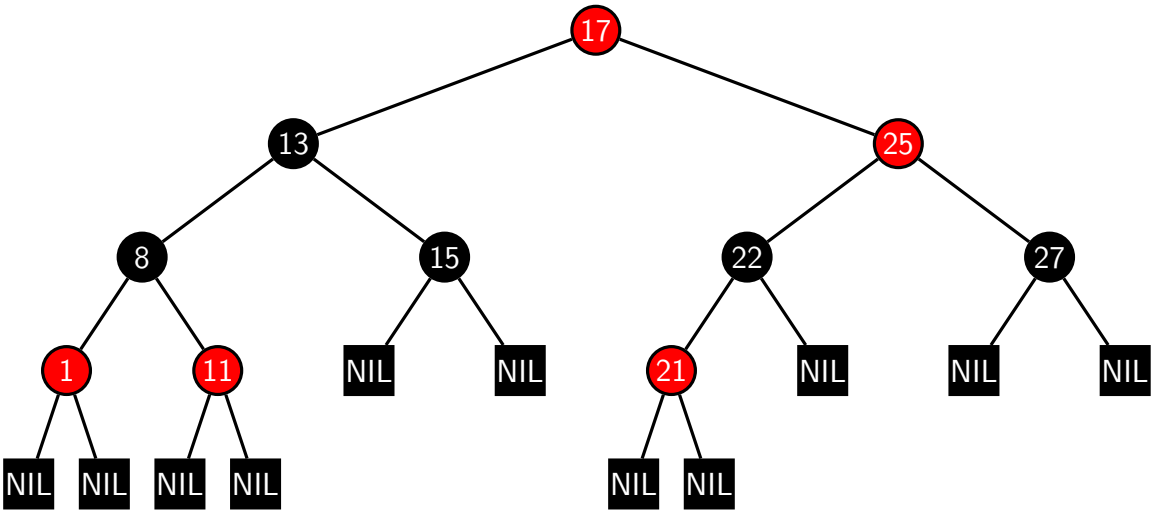
例如在一个红黑树中插入新结点 21，需要根据不同局面进行调整。



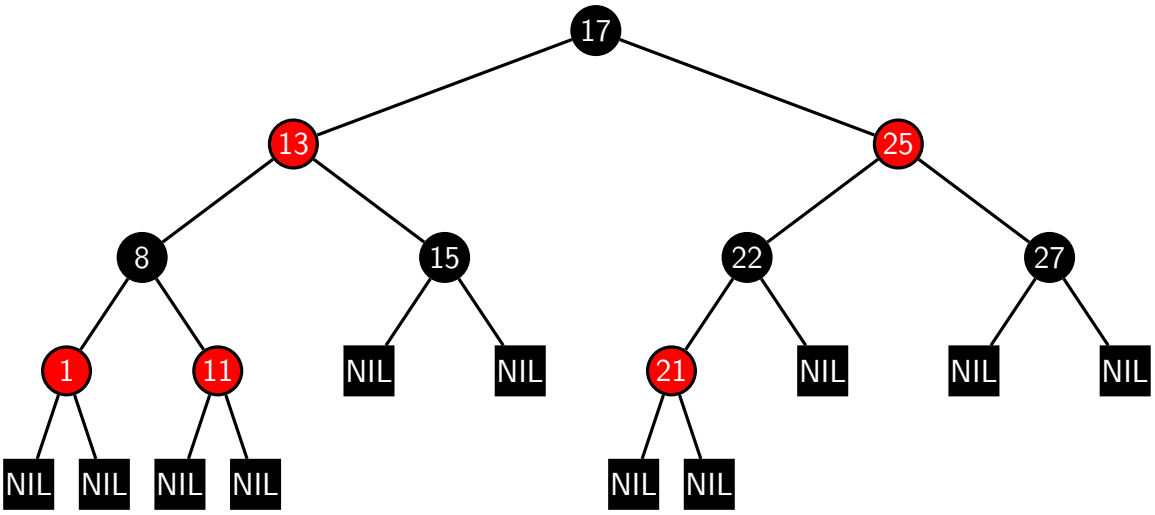
新结点 21 和它的父结点 22 是连续的红色结点，违背了规则 4。当前情况符合局面 3（新结点的父结点和叔叔结点都是红色）。于是经过三次变色（22 变为黑色，25 变为红色，27 变为黑色），将以结点 25 为根的子树符合了红黑树的规则。



但结点 25 和结点 17 成为了连续的红色结点，违背了规则 4。于是可以将结点 25 看做一个新结点，当前正好符合局面 5 的镜像（新结点的父结点是红色，叔叔是黑色或者没有叔叔，且新结点是父结点的右孩子，父结点是祖父结点的右孩子）。因此可以以根结点 13 为轴进行左旋转，使得结点 17 成为新的根结点。



再让结点 17 变为黑色，13 变为红色，使红黑树重新符合规则。



9.9.4 二叉查找树删除结点

在介绍红黑树的删除操作之前，需要先理解二叉查找树的删除操作。

二叉查找树的删除可分为三种情况：

待删除结点无子结点

如果待删除结点没有子结点，直接删除即可。

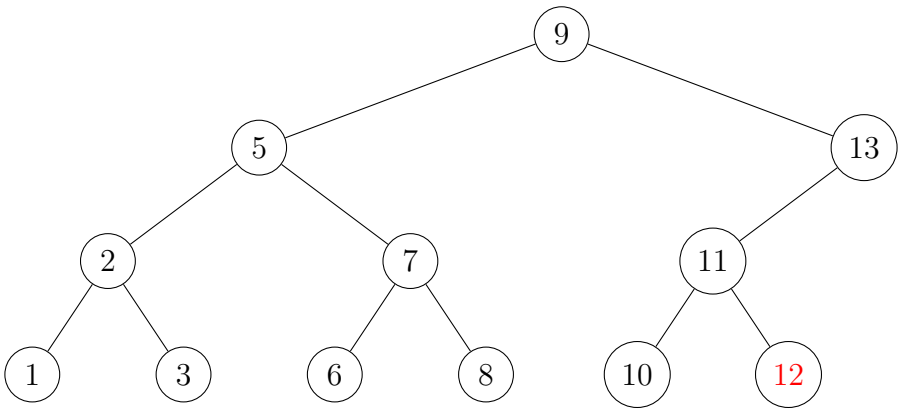


图 9.67: 删除结点 12

待删除结点只有一个孩子

如果待删除结点只有一个孩子，让孩子取代待被删除结点。

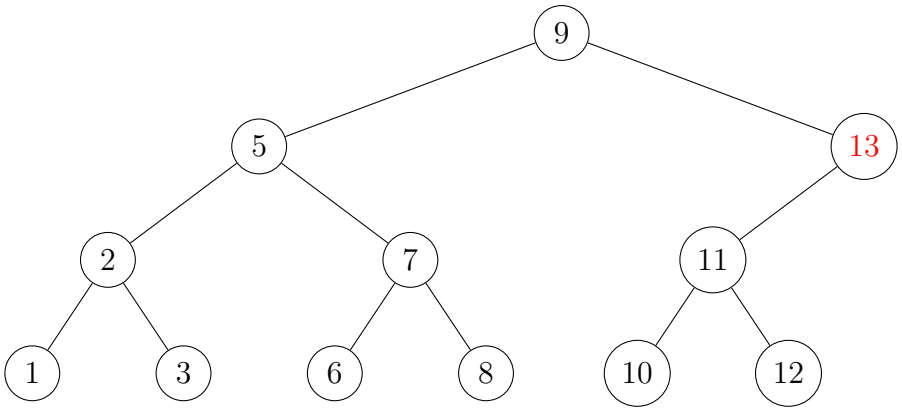
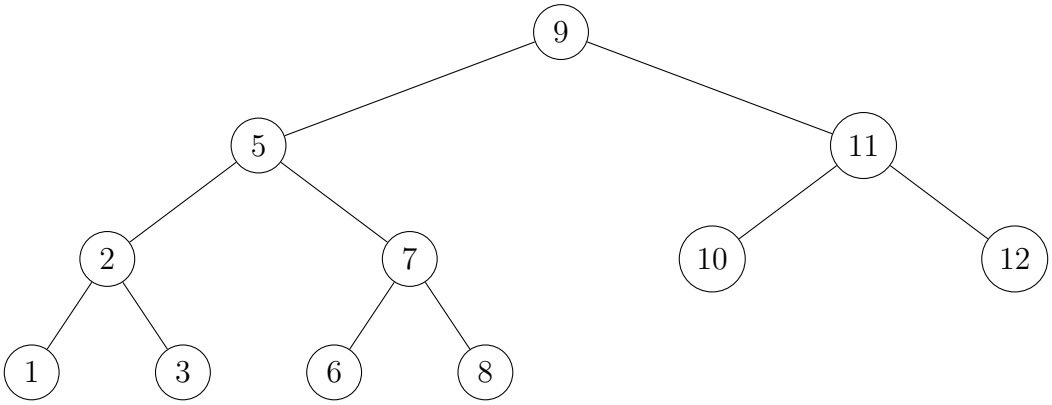


图 9.68: 删除结点 13



待删除结点有两个孩子

选择仅小于或仅大于待删除结点的结点取代，习惯上更多地会选择仅大于待删除结点的结点。

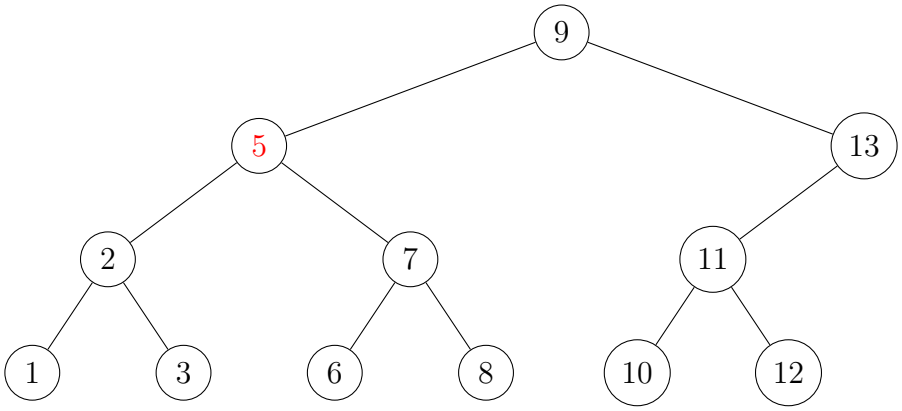
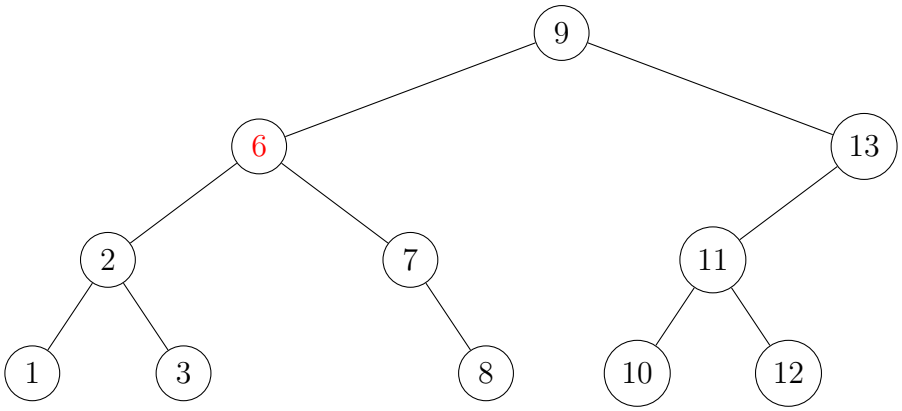


图 9.69: 删除结点 5



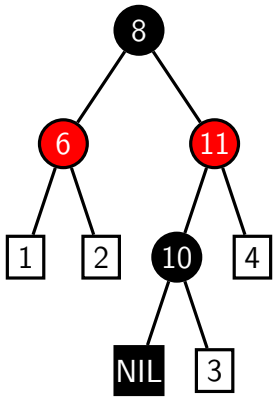
9.9.5 红黑树删除结点

红黑树的删除操作要比插入操作复杂得多。

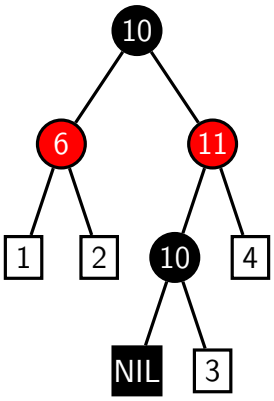
第一步

如果待删除结点有两个非空的孩子结点，转化成待删除结点只有一个孩子（或没有孩子）的情况。

例如删除结点 8:



因为结点 8 有两个孩子，可以选择仅大于 8 的结点 10 复制到 8 的位置，结点颜色变成待删除结点的颜色。

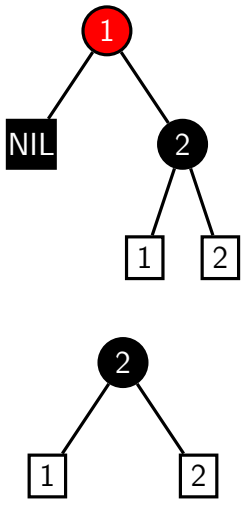


结点 10 能成为仅大于 8 的结点，必定没有左孩子结点，所以问题转换成了待删除结点只有一个右孩子（或者没有孩子）的情况。

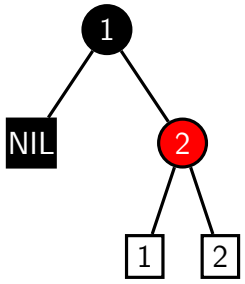
第二步

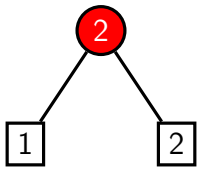
根据待删除结点和其唯一子结点的颜色，分情况处理。

情况 1：自身是红色，子结点是黑色。直接按照二叉查找树的删除操作，删除结点 1 即可。

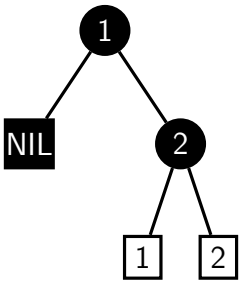


情况 2：自身是黑色，子结点是红色。按照二叉查找树的删除操作，删除结点 1。此时这条路径凭空少了一个黑色结点，因此需要将结点 2 变成黑色即可。





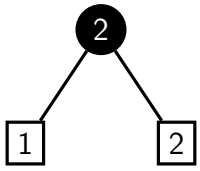
情况 3：自身是黑色，子结点也是黑色，或者子结点是空叶子结点。这种情况最为复杂，涉及到很多变化。首先还是按照二叉查找树的删除操作，删除结点 1。此时这条路径凭空少了一个黑色结点，而且并不能改变结点 2 的颜色来解决问题。这时需要进入第三步，专门解决父子双黑的情况。



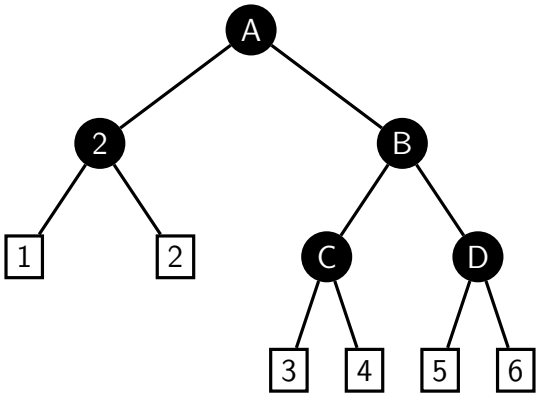
第三步

遇到双黑结点，在子结点顶替父结点后，可分为 6 种情况处理。

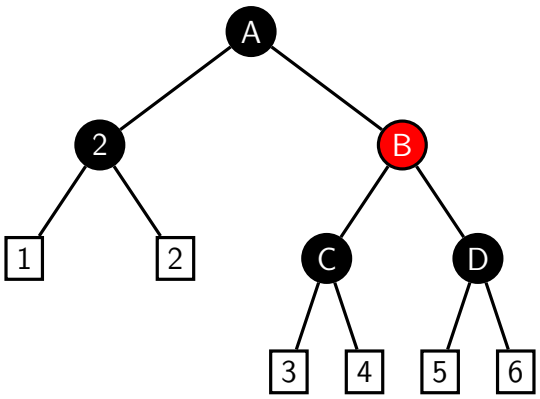
情况 1：结点 2 是红黑树的根。此时所有路径都减少了一个黑色结点，并未打破规则，无需调整。



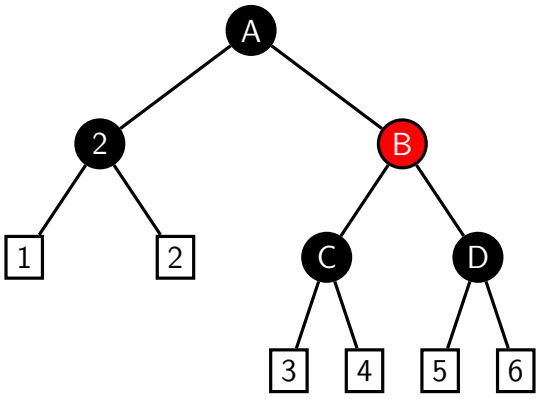
情况 2：结点 2 的父亲、兄弟和侄子结点都是黑色。



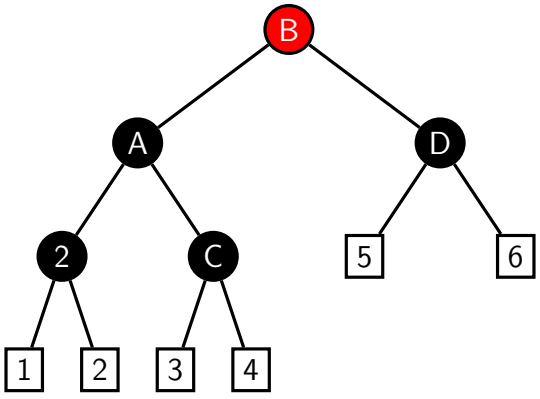
直接把结点 2 的兄弟结点变为红色。这样结点 B 所在路径少了一个黑色结点，两边扯平了。



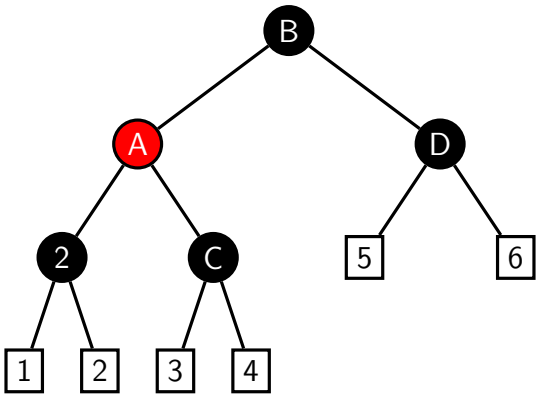
情况 3： 结点 2 的兄弟结点是红色。



以结点 2 的父结点 A 为轴进行左旋转。

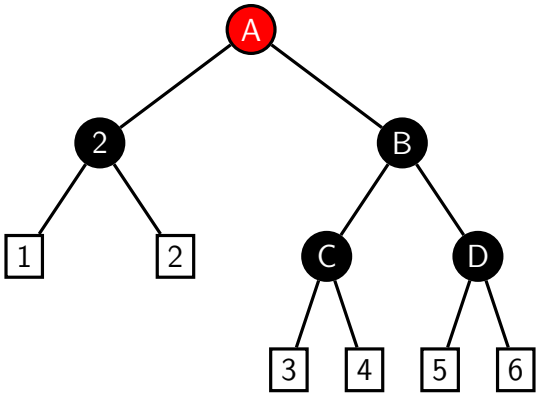


然后结点 A 变为红色，结点 B 变为黑色。

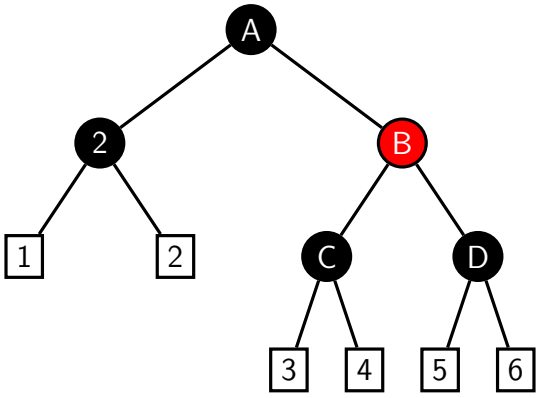


这样的变化就有可能转换成情况 4、5、6 中的任意一种。

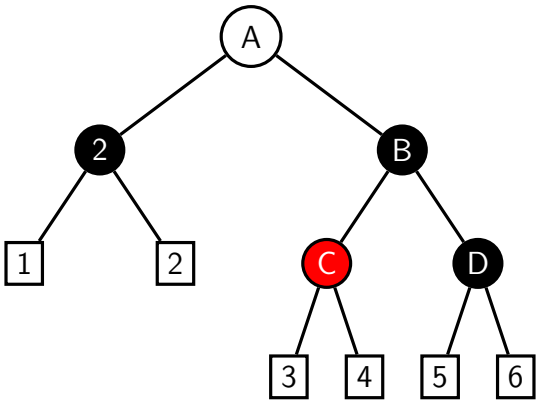
情况 4：结点 2 的父结点是红色，兄弟和侄子结点是黑色。



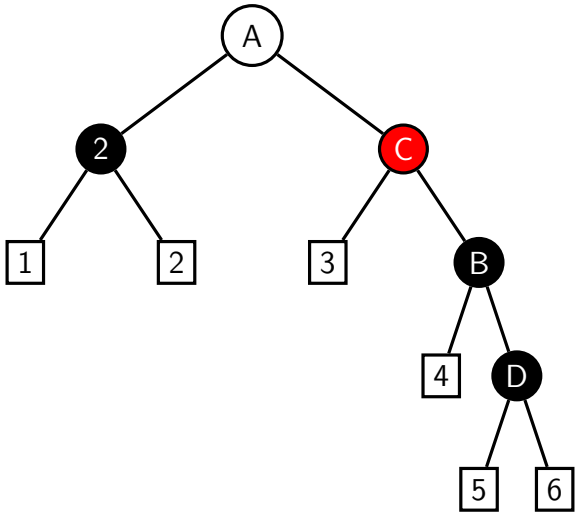
将结点 2 的父结点 A 变为黑色，兄弟结点 B 变为红色。这样结点 2 的路径补充了黑色结点，而结点 B 的路径并没有减少黑色结点。



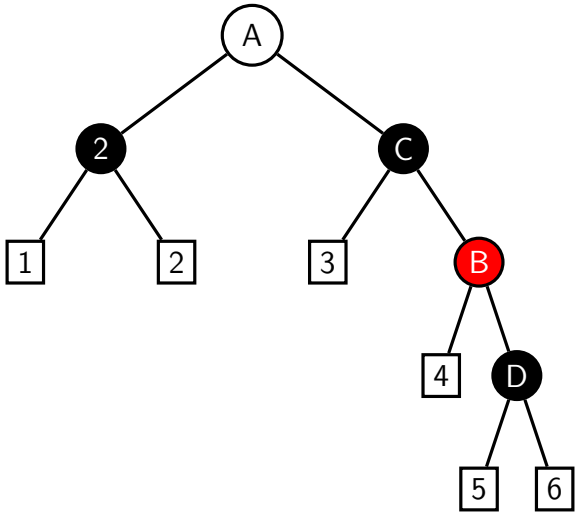
情况 5：结点 2 的父结点随意，兄弟结点 B 是黑色右孩子，结点 2 的左侄子是红色，右侄子是黑色。



以结点 2 的兄弟结点 B 为轴进行右旋转。

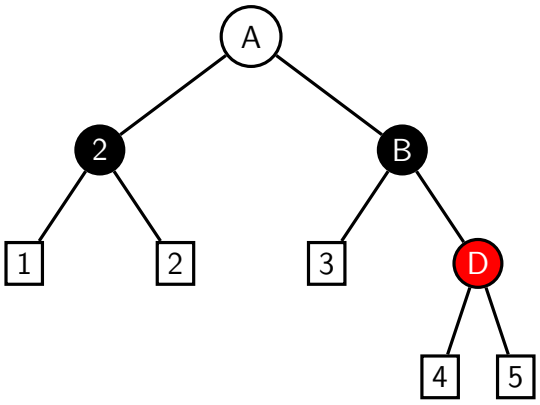


将结点 B 变为红色，结点 C 变为黑色。

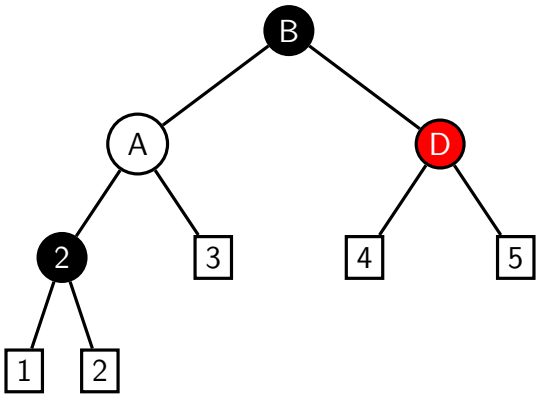


这样的变化就转换成了情况 6。

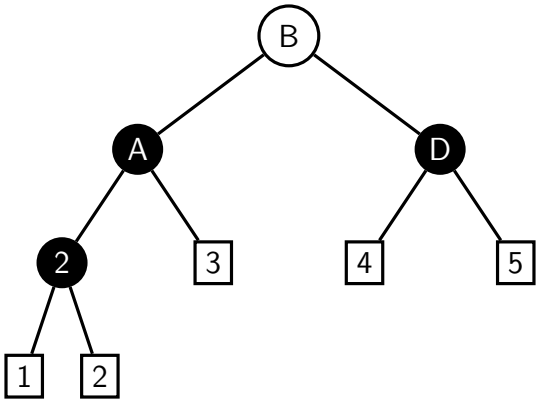
情况 6：结点 2 的父结点随意，兄弟结点 B 是黑色右孩子，结点 2 的右侄子的红色。



以结点 2 的父结点 A 为轴进行左旋转。

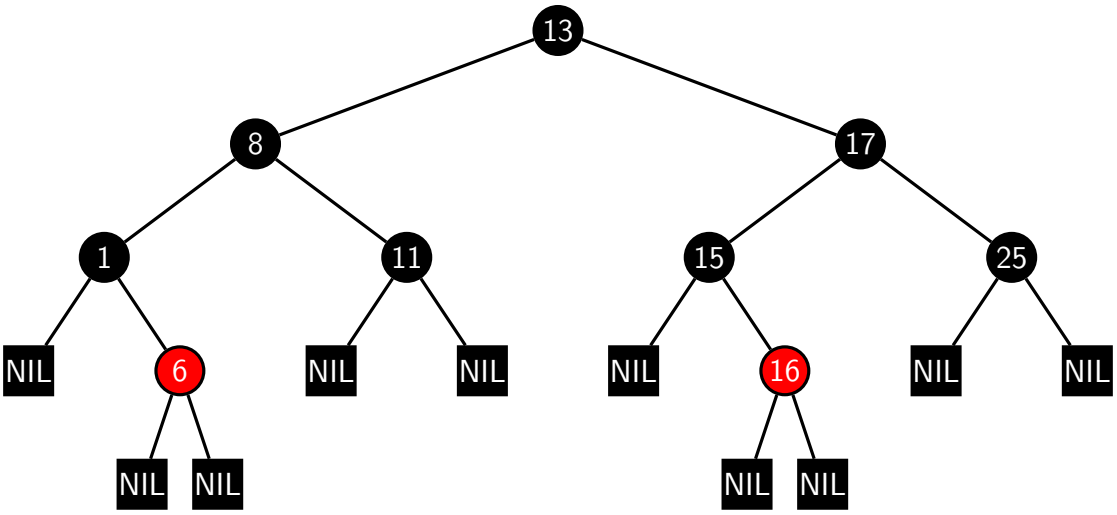


将结点 A 和 B 的颜色交换，让结点 D 变为黑色。

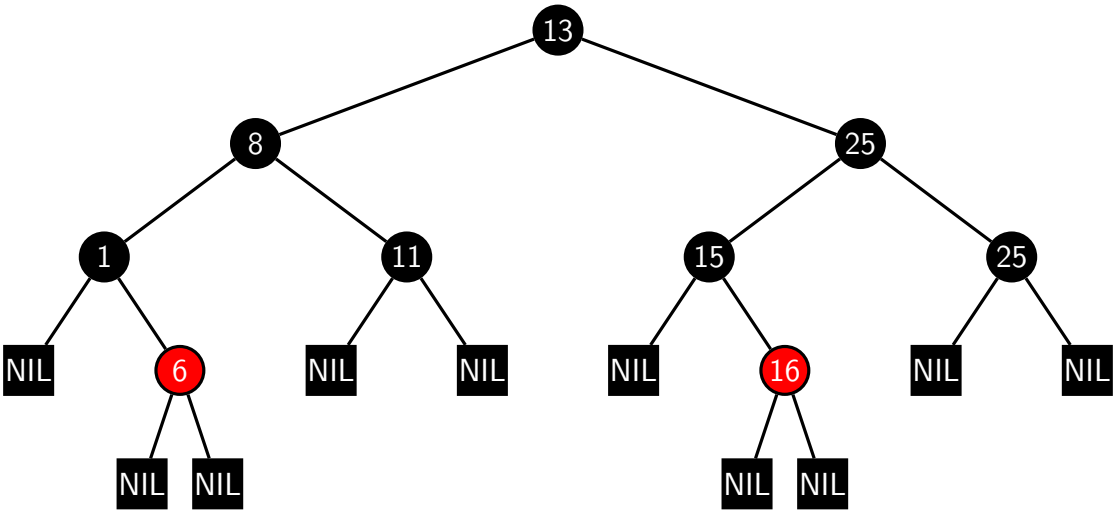


这样经过结点 2 的路径由之前的随机 + 黑变成了随机 + 黑 + 黑，补充了一个黑色结点。经过结点 D 的路径由之前的随机 + 黑 + 红变成了随机 + 黑，黑色结点并没有减少。

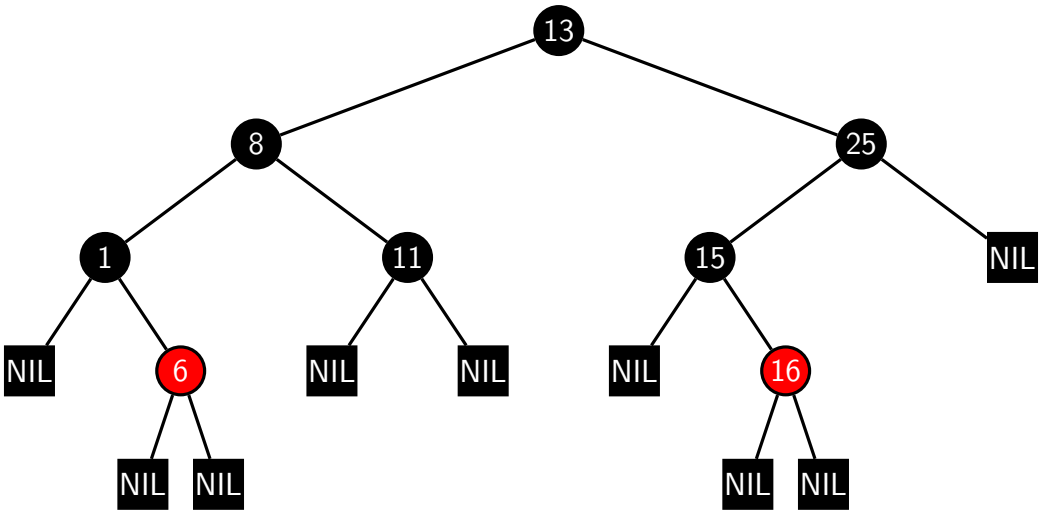
例如在一个红黑树中删除结点 17，需要根据不同情况进行调整。



由于待删除结点 17 有两个孩子，子树当中仅大于 17 的结点是 25，所以把结点 25 复制到 17 的位置，保持黑色。

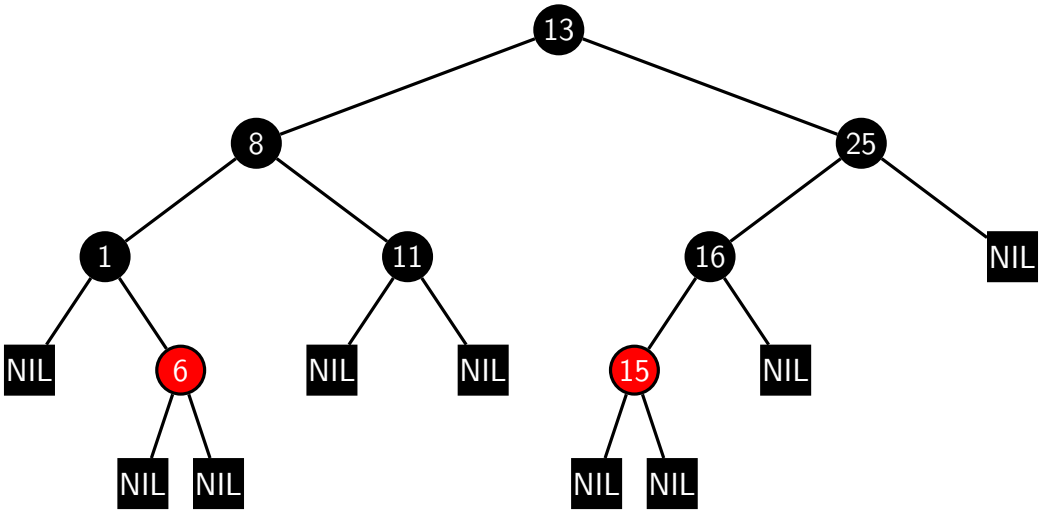


接着需要删除原本的结点 25，这个情况对应的是删除结点操作中的第二步情况 3（待删除结点是黑色，子结点是空叶子结点）。

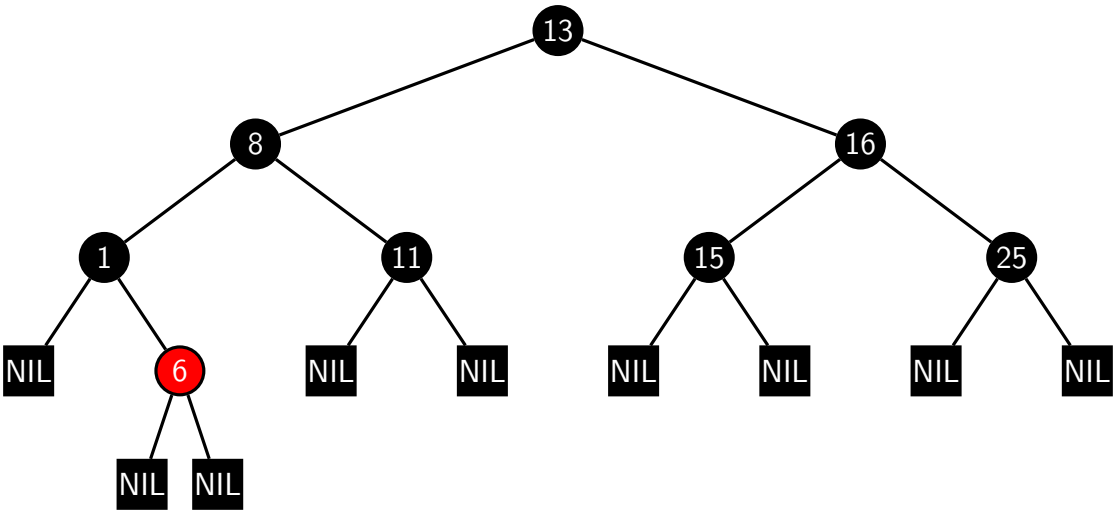


此时，以结点 25 为根的子树符合第三步情况 5 的镜像（结点 NIL 的父结点随意，兄弟结点 15 是黑色左孩子，结点 NIL 的右侄子是红色，左侄子是黑色）。

通过左旋转和变色，将子树转化成第三步情况 6 的镜像（结点 NIL 的父结点随意，兄弟结点 16 是黑色左孩子，结点 NIL 的左侄子是红色）。



通过右旋转和变色，使整棵二叉树重新符合红黑树的规则。



9.9.6 红黑树与 AVL 树的区别

AVL 树是严格平衡的二叉树，要求每个结点的左右子树高度差不超过 1，而红黑树则要宽松一些，要求任何一条路径的长度不超过其它路径长度的 2 倍。

正因为这个差别，AVL 树的查找效率更高，但平衡调整的成本也更高。在需要频繁查找时，选用 AVL 树更合适，在需要频繁插入删除时，选用红黑树更合适。

9.10 B 树

9.10.1 B 树

B-树就是 B 树，中间的是连字符而不是减号，谁要是把 B-树读成 “B 减树”，那可就丢人现眼了。

B 树主要应用于文件系统以及部分数据库索引，比如著名的非关系型数据库 MongoDB。数据库索引使用树型结构的原因在于树的查询效率高，而且可以保持有序。既然这样，为什么索引没有采用二叉查找树来实现呢？二叉查找树查询的时间复杂度是 $O(logn)$，性能已经足够高了，难道 B 树可以比它更快？

其实从算法逻辑上来讲，二叉查找树的查找速度和比较次数都是最小的。但是，我们不得不考虑一个现实的问题——磁盘 I/O。数据库索引是存储在磁盘上的，当数据量比较大的时候，索引的大小可能有好几个 G 甚至更多。当利用索引查询的时候，显然不可能把整个索引全部加载到内存，能做的只有逐一加载每一页的磁盘页，这里的磁盘页就对应着索引树的结点。

如果利用二叉查找树作为索引结构，假设需要查询数据 10，一共需要进行 4 次磁盘 I/O。

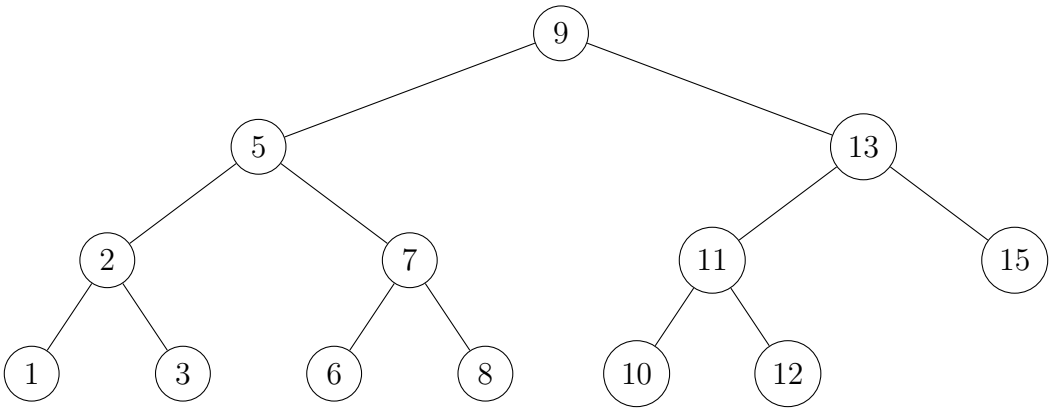


图 9.70: 索引树

最坏情况下，磁盘 I/O 的次数等于索引树的高度。既然如此，为了减少磁盘 I/O 次数，就需要把原本瘦高的树结构变得矮胖，这就是 B 树的特征之一。

B 树是一种多路平衡查找树，它的每一个结点最多包含 k 个孩子，k 被称为 B 树的阶，k 的大小取决于磁盘页的大小。

一个 m 阶的 B 树具有以下特征：

1. 根结点至少有 2 个孩子。
2. 每个中间结点都包含 k - 1 个元素和 k 个孩子（$\frac{m}{2} \leq k \leq m$）。
3. 每个叶子结点都包含 k - 1 个元素（$\frac{m}{2} \leq k \leq m$）。
4. 所有叶子结点都位于同一层。
5. 每个结点中的元素从小到大排列，结点中 k - 1 个元素正好是 k 个孩子包含的元素的值域分划。

例如对于一个 3 阶 B 树：

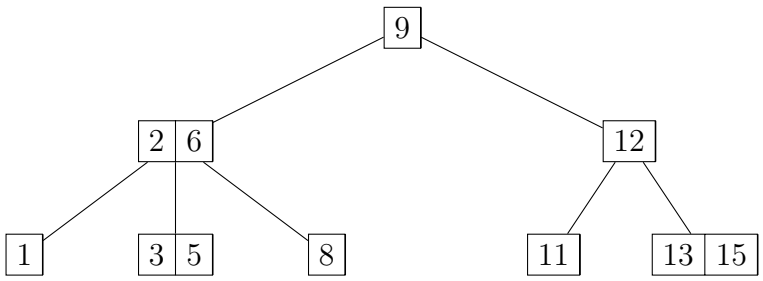


图 9.71: 3 阶 B 树

其中结点 (2, 6) 包含两个元素，并且有三个孩子 1、(3, 5) 和 8，其中 1 小于元素 2，(3, 5) 在元素 2 和 6 之间，8 大于 6。

9.10.2 查询结点

这树长得真奇怪，它真的能实现高效查询吗?

例如查询数据 5，只需进行 3 次磁盘 I/O。B 树在查询中的比较次数其实不比二叉查找树少，尤其当单一结点中的元素数量很多时。可是相比磁盘 I/O 的速度，内存中的比较耗时是几乎可以忽略的。所以只要树的高度足够低，I/O 次数足够少，就可以提升查询效率。相比之下结点内部元素多一些也没有关系，仅仅

是多了几次内存交互，只要不超过磁盘页的大小即可。这些就是 B 树的优势之一。

9.10.3 插入结点

例如在 3 阶 B 树中插入元素 4，由于结点 (3, 5) 已经是两元素结点，无法再增加。父结点 (2, 6) 也是两元素结点，也无法再增加。根结点 9 是单元素结点，可以升级为两元素结点。于是拆分结点 (3, 5) 和结点 (2, 6)，让根结点 9 升级为 (4, 9)，结点 6 独立为根结点的第二个孩子。

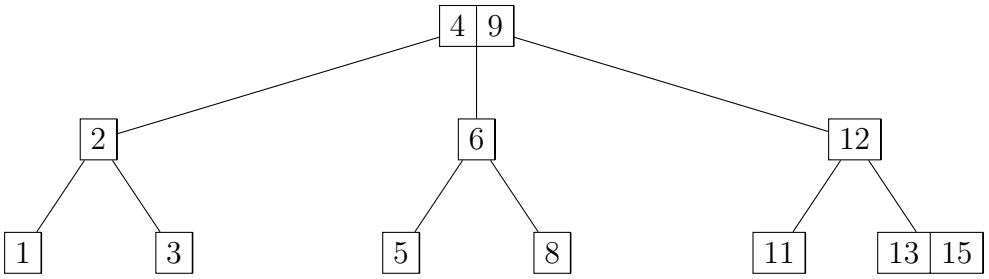


图 9.72: 插入结点 4

为了插入一个元素，让整个 B 树的那么多结点都发生了连锁改变，确实有点麻烦。但也正因为如此，让 B 树能够始终维持多路平衡。因此自平衡是 B 树的另一大优势。

9.10.4 删除结点

例如在 3 阶 B 树中删除元素 11，删除结点 11 后，结点 12 只有一个孩子，不符合 B 树特征。因此找出 12、13 和 15 中的中位数 13，取代结点 12，而结点 12 下移成为第一个孩子。

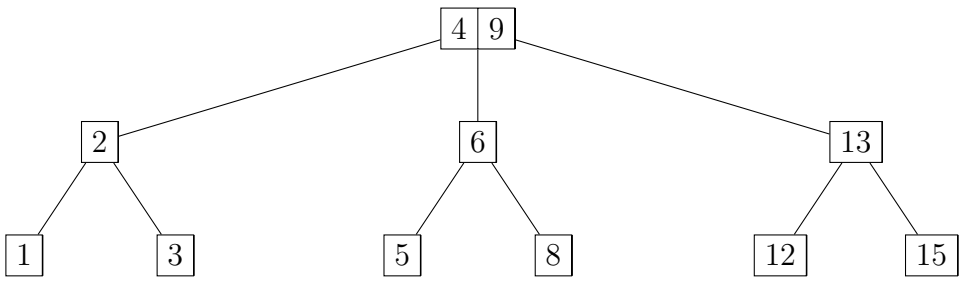


图 9.73: 删除结点 11

9.11 B+ 树

9.11.1 B+ 树

B+ 树是 B 树的一种变体，有着比 B 树更高的查询性能。B+ 树和 B 树有一些共同点，但是 B+ 树也具备一些新的特征。

一个 m 阶的 B+ 树具有以下特征：

1. 有 k 个子树的中间结点包含 k 个元素（B 树中是 k - 1 个元素）。
2. 所有的叶子结点中包含了全部元素的信息，及指向含这些元素记录的指针，且叶子结点本身按照关键字大小连接。
3. 所有中间结点元素都同时存在于子结点，在子结点元素中是最大（或最小）元素。

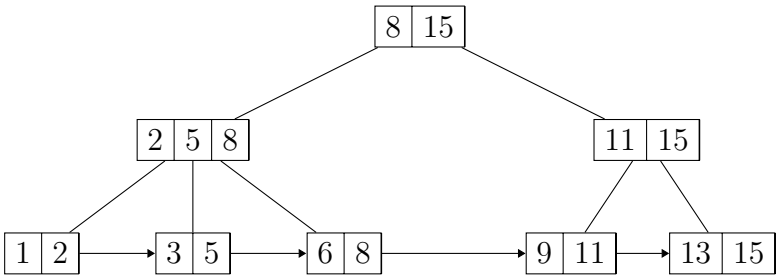


图 9.74: B+ 树

这又是什么怪树？不但结点之间含有重复元素，而且叶子结点之间还用指针连在一起。

这些正是 B+ 树的几个特性。首先，每一个父结点的元素都出现在子结点中，是子结点的最大（或最小）元素。因此根结点的最大元素也就是整个 B+ 树的最大元素，之后无论插入或删除多少元素，始终要保持最大元素在根结点中。至于叶子结点，由于父结点的元素都出现在子结点，因此所有叶子结点包含了全部元素信息。并且每一个叶子结点都带有指向下一个结点的指针，形成了一个有序链表。

9.11.2 单行查询

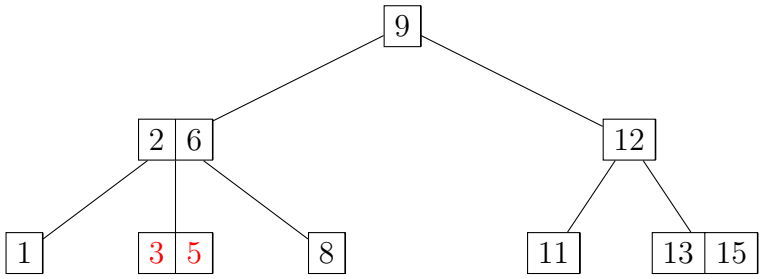
B+ 树的好处主要体现在查询性能上。在单元素查询的时候，B+ 树会自顶向下逐层查找结点，最终找到匹配的叶子结点。例如在 B+ 树中查询元素 3，需要进行 3 次磁盘 I/O。

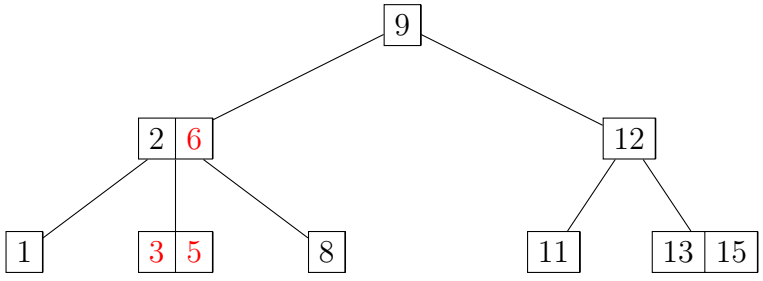
虽然流程看起来和 B 树差不多，但是有两点不同。首先，B+ 树的中间结点并不关联数据，仅仅是索引，所以同样大小的磁盘页可以容纳更多的结点元素。这就意味着，数据量相同的情况下，B+ 树的结构比 B 树更加矮胖，因此查询时 I/O 次数也更少。

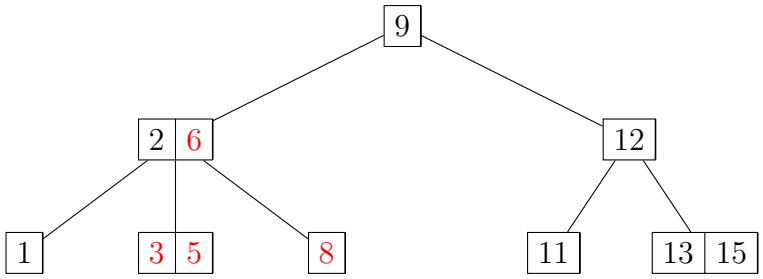
其次，B+ 树的查询性能必须最终查找到叶子结点，而 B 树只要找到匹配元素即可。因此，B 树的查询性能并不稳定，最好情况下只查询根结点，最坏情况下是查到叶子结点，而 B+ 树的每一次查询都是稳定的。

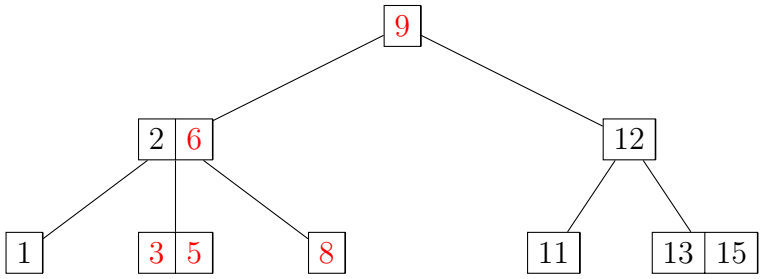
9.11.3 范围查询

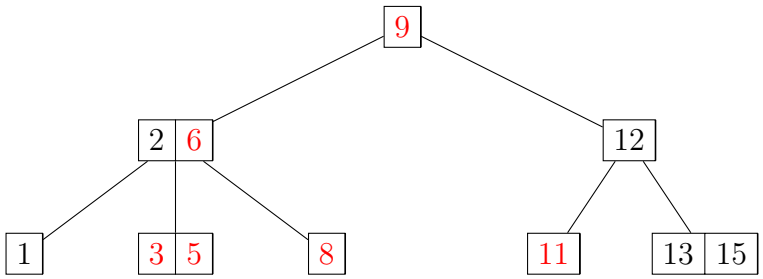
B 树中进行范围查询只能通过繁琐的中序遍历。例如在 B 树中查询范围为 3 到 11 的元素：



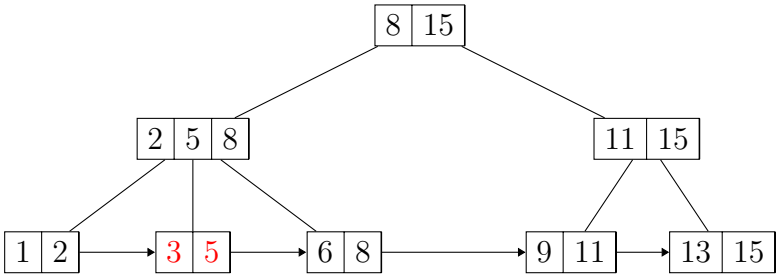


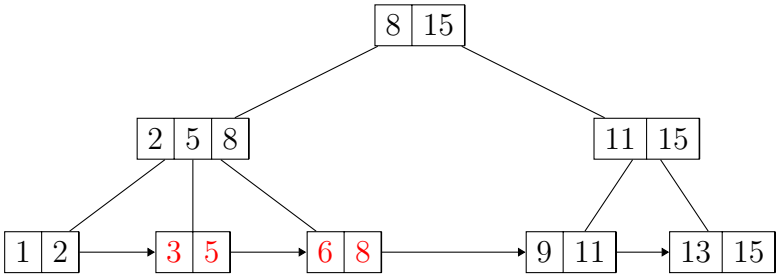


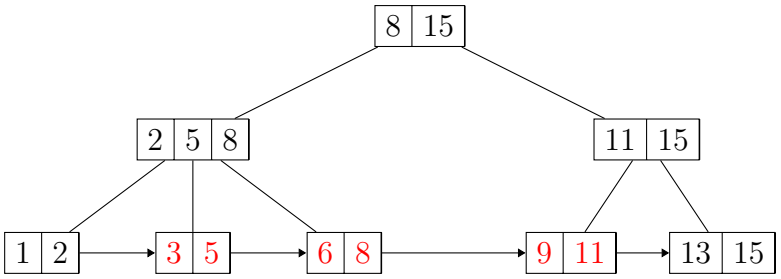




反观 B+ 树的范围查询，则要简单的多，只需要在链表上做遍历即可：






8 15
2 5 8
11 15
1 2
3 5
6 8
9 11
13 15

9.12 并查集

9.12.1 并查集（Disjoint Set）

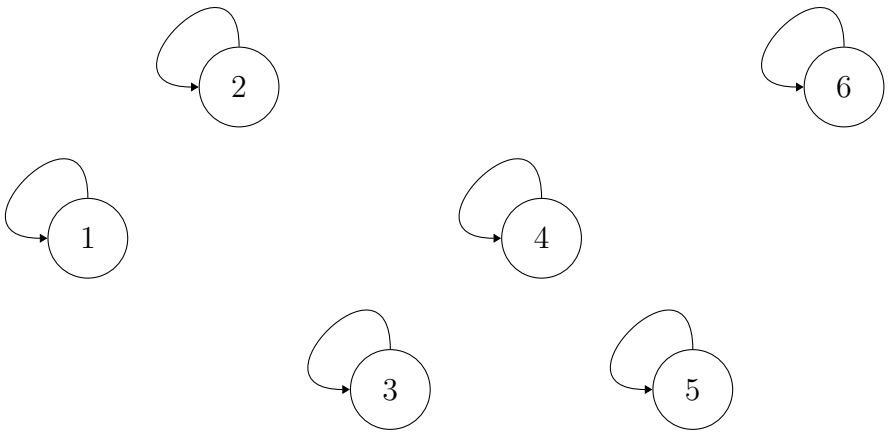
并查集是一种简洁优雅的数据结构，主要用于解决一些元素分组的问题。

它管理一系列不相交的集合，并支持两种操作：

1. 合并（union）：把两个不相交的集合合并为一个集合。
2. 查询（find）：查询两个元素是否在同一个集合中。

例如在某个家族中，如果 x 和 y 是亲戚，y 和 z 是亲戚，那么 x 和 z 也是亲戚。如果 x 和 y 是亲戚，那么 x 的亲戚都是 y 的亲戚，y 的亲戚也都是 x 的亲戚。为了判断两个人是否为亲戚，只需判断他们是否属于同一个集合即可。

并查集的思想在于用集合中的一个元素代表集合，类似于把集合看做帮派，代表元素则是帮主，最开始所有元素各自为一个集合（各自的帮主就是自己）。



例如将元素 1 和 3 合并，就是将 1 和 3 比武，假设 1 赢了，3 就认 1 作帮主。

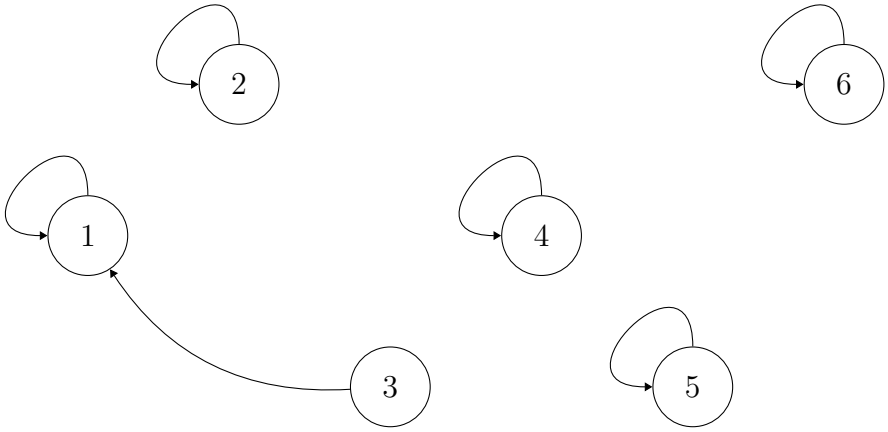


图 9.75: 合并 1 和 3

再合并元素 2 和 3，但 3 表示“别跟我打，让我的帮主来收拾你”。假设还是 1 赢了，2 也认 1 作帮主。

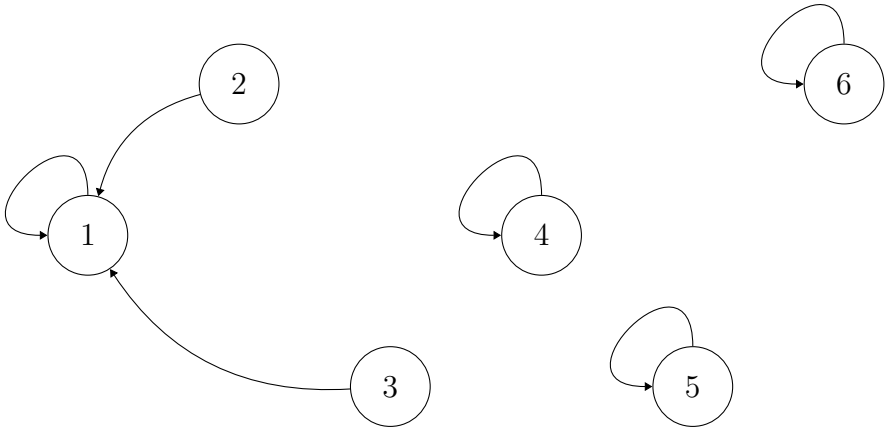


图 9.76: 合并 2 和 3

假设元素 4、5、6 也进行了相关合并：

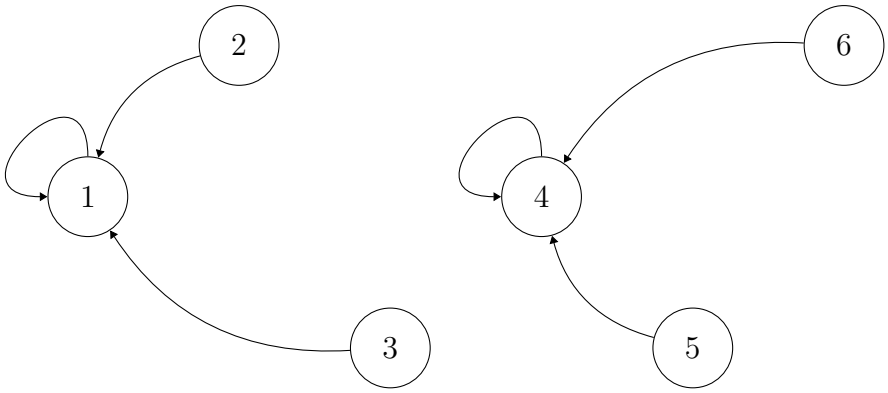


图 9.77: 合并 4、5、6

现在再合并元素 2 和 6，将它们的帮主 1 和 4 比武，假设 1 胜利后，4 认 1 为帮主，当然它的手下也都跟着投降了。

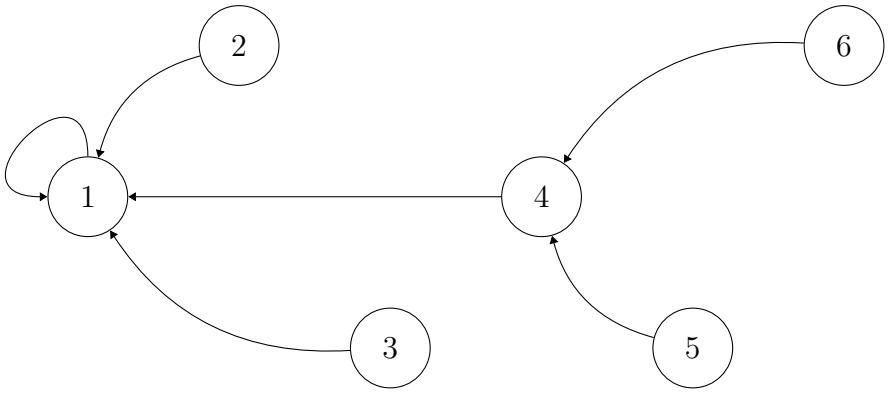


图 9.78: 合并 2 和 6

并查集是一种树型结构，要寻找集合的代表元素，只需要一层一层向上访问父结点，直达树的根结点即可。根结点的父结点就是它自己。

假设有 n 个元素，利用数组 parent 存放每个元素的父结点，一开始每个元素的父结点为自己。

初始化

```
1 void init(int n) {
2     parent = (int *)malloc(sizeof(int) * n);
3     for(int i = 0; i < n; i++) {
4         parent[i] = i;
```

```
    }
}
```

要判断两个元素是否属于同一个集合，只需要看它们的根结点是否相同。利用递归的方法可以实现对代表元素的查询，一层一层访问父结点，直至根结点。

查询

```
int find(int val) {
    if(parent[val] == val) {
        return val;
    }
    return find(parent[val]);
}
```

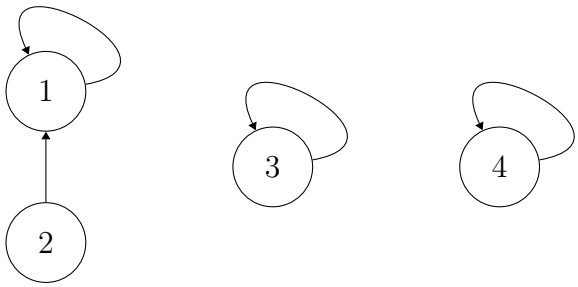
合并操作需要先找到两个集合的根结点，将前者的父结点设置为后者即可（也可将后者的父结点设置为前者）。

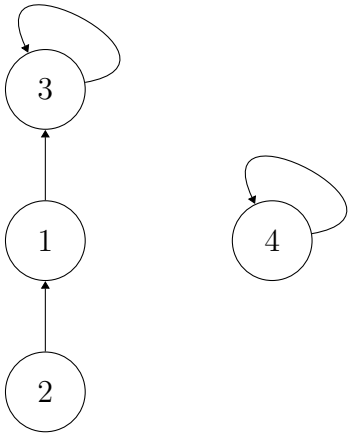
合并

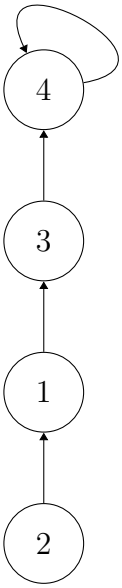
```
void merge(int i, int j) {
    parent[find(i)] = find(j);
}
```

9.12.2 路径压缩（Path Compression）

最简单的并查集效率是比较低的，分别进行 merge(2, 3) 和 merge(2, 4)：

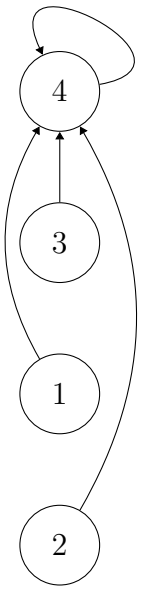






这样可能会形成一条长链，随着链越来越长，想要从底部找到根结点会变得越来越难。利用路径压缩的方法可以解决这个问题，因为我们只关心一个元素所对应的根结点，因此每个元素到根结点的路径最好尽可能短。

实现的时候只需在查询过程中，把沿途的每个结点的父结点都设置为根结点即可。在下一次查询的时候，就可以节省很多时间。

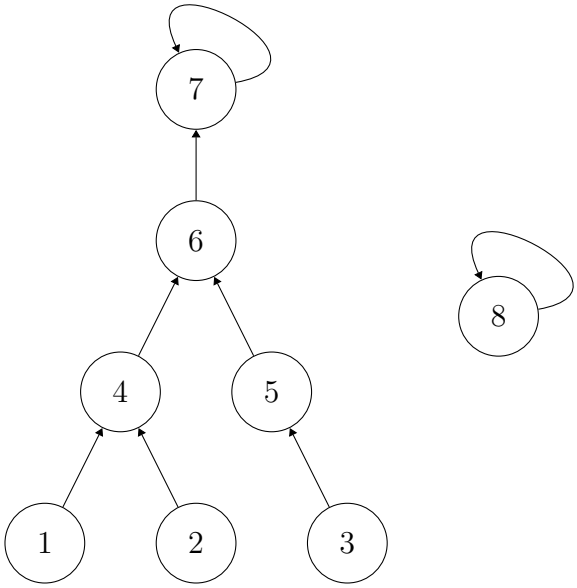


查询（路径压缩）

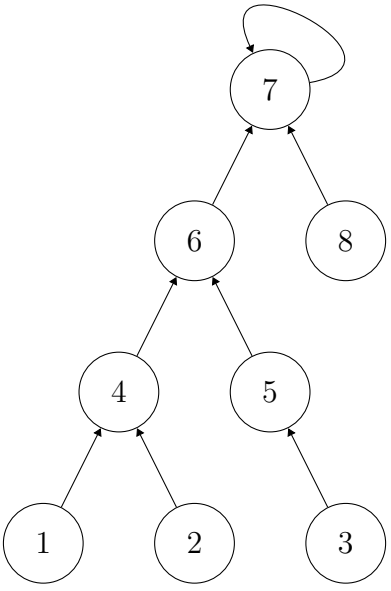
```
int find(int val) {
    if(parent[val] == val) {
        return val;
    } else {
        parent[val] = find(parent[val]);
        return parent[val];
    }
}
```

9.12.3 按秩合并

如果需要将一棵比较复杂的树与一个单元素进行合并，例如 merge(7, 8) 时，是把元素 7 的父结点设为 8 好，还是把 8 的父结点设为 7 呢?



如果把 7 的父结点设为 8，会使树的深度加深，原来树中的每个元素到根结点的距离都变长了，之后寻找根结点的路径也会变长。虽然有路径压缩，但路径压缩也是要消耗时间的。而把 8 的父结点设为 7，并不会影响到不相关的结点。



因此在合并两个集合时，应该把简单的树往复杂的树上合并。利用数组 rank 记录每个结点对应的树的深度，一开始所有元素的 rank 设为 1。合并时比较两个根结点，把 rank 较小者往较大者上合并。深度相同的情况下，无论如何合并，都会使树的深度增加 1。

按秩合并

```c
void init(int n) {
    parent = (int *)malloc(sizeof(int) * n);
    rank = (int *)malloc(sizeof(int) * n);
    for(int i = 0; i < n; i++) {
        parent[i] = i;
        rank[i] = 1;
    }
}

void merge(int i, int j) {
    // 找到对应根结点
    int x = find(i);
    int y = find(j);
    if(rank[x] <= rank[y]) {
        parent[x] = y;
    } else {
        parent[y] = x;
    }
    // 如果深度相同且根结点不同，则新的根结点深度+1
    if(rank[x] == rank[y] && x != y) {
        rank[y]++;
    }
}
```