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。

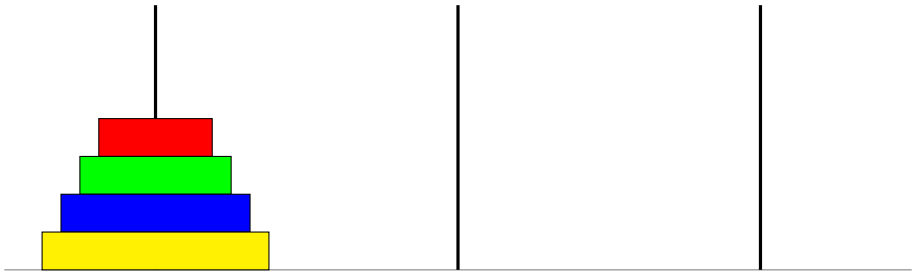

图 1.1: 汉诺塔

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?

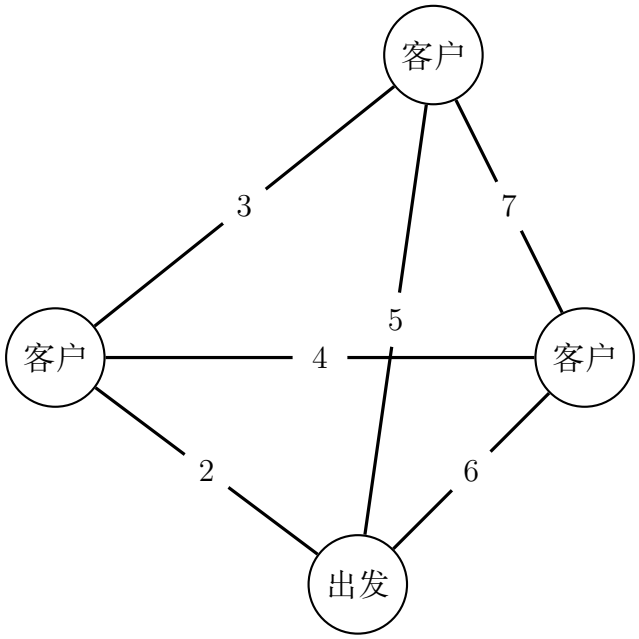

图 1.2: 快递客户路线

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：

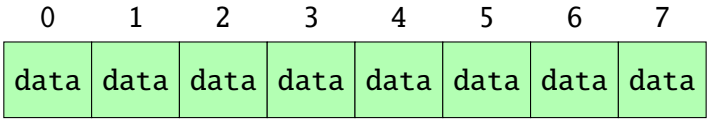

图 2.1: 数组

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。

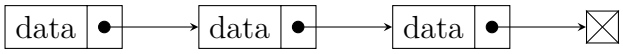

图 3.1: 单向链表

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
```

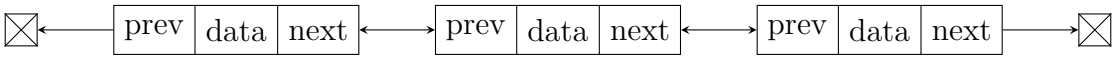

图 3.2: 双向链表

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。

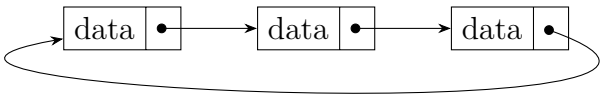

图 3.3: 单向循环链表

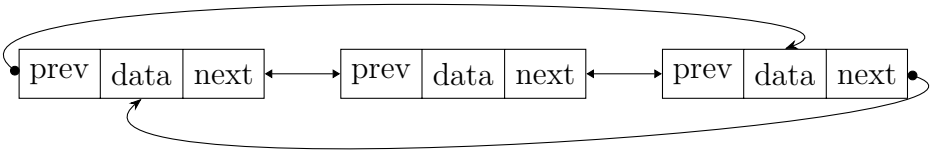

图 3.4: 双向循环链表

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。

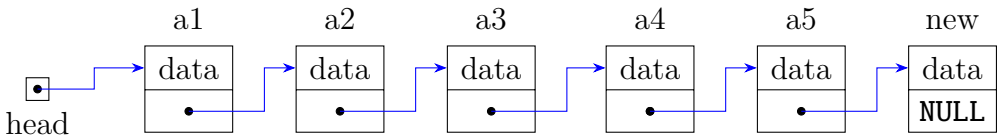

图 3.5: 尾部插入

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。

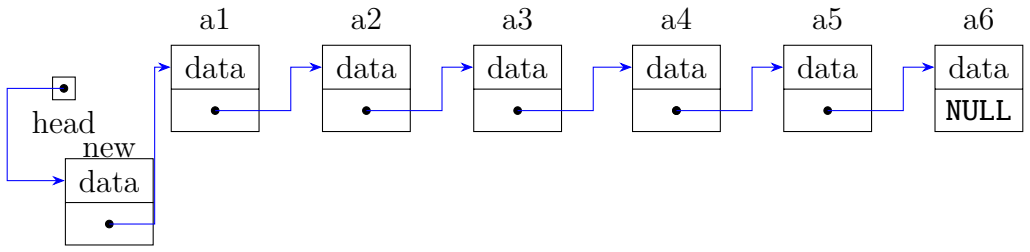

图 3.6: 头部插入

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。

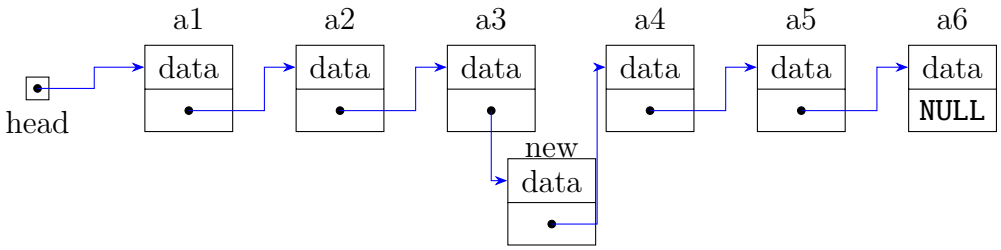

图 3.7: 中间插入

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。

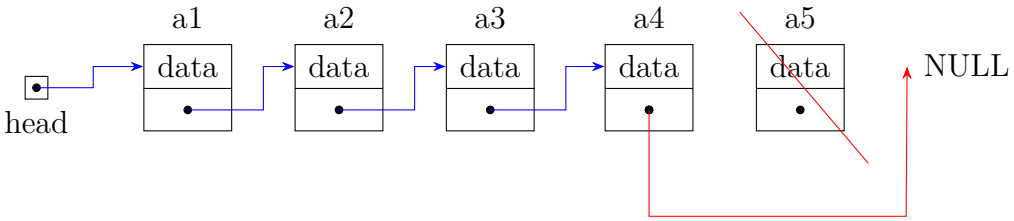

图 3.8: 尾部删除

头部删除

把链表的头结点设置为原先头结点的 next 指针。

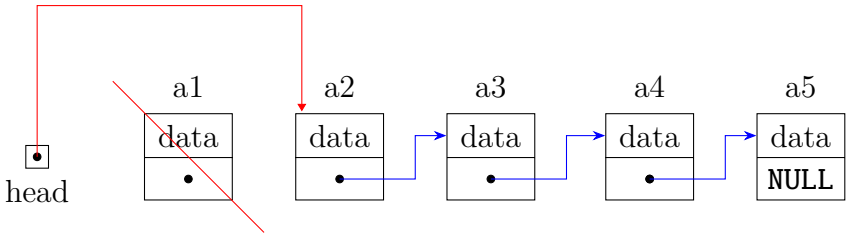

图 3.9: 头部删除

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。

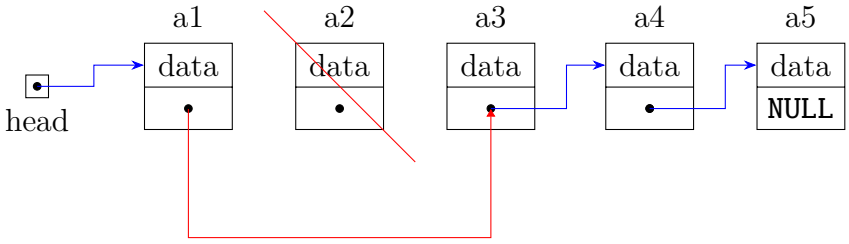

图 3.10: 中间删除

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。

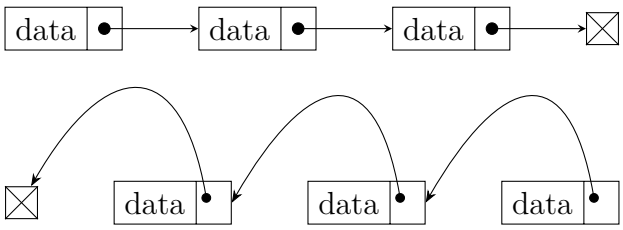

图 3.11: 反转链表

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）。

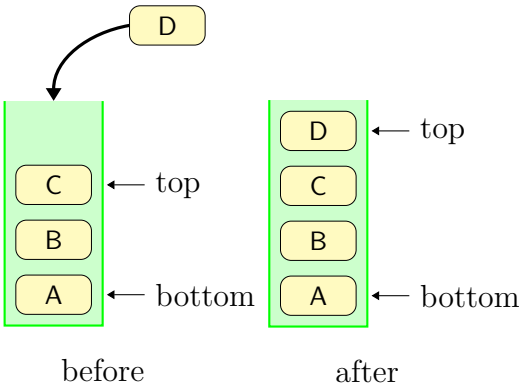

图 4.1: 栈

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。

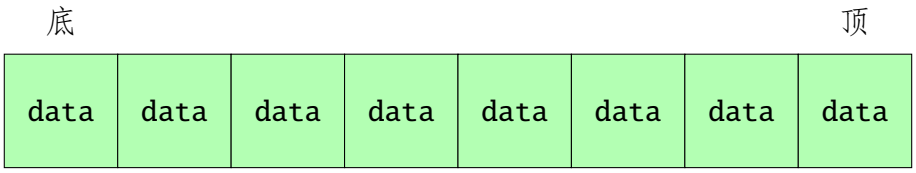

图 4.2: 顺序栈

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。

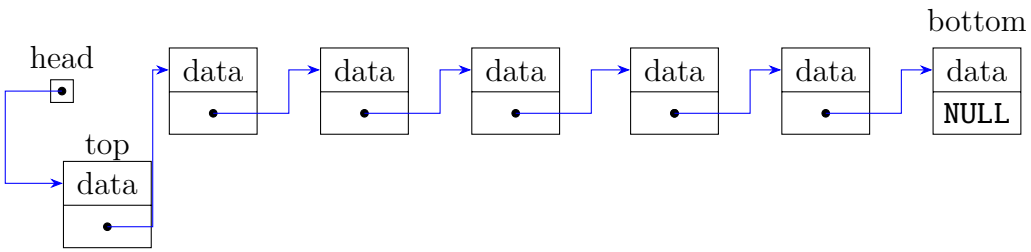

图 4.3: 链式栈

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 $O(1)$。

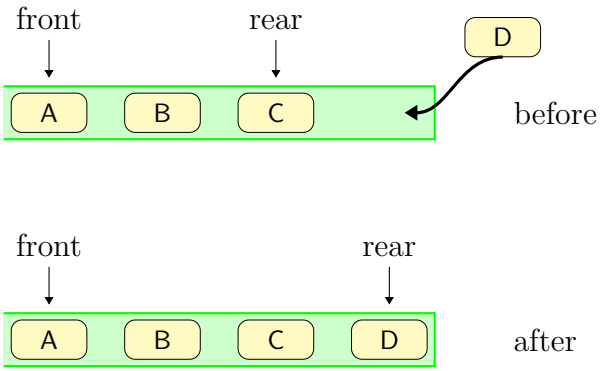

图 5.1: 入队

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 $O(1)$。

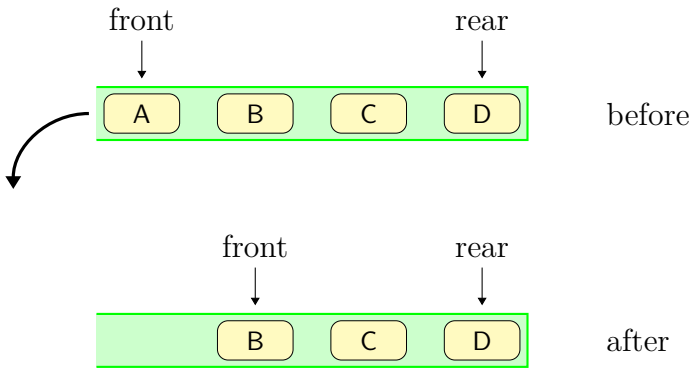

图 5.2: 出队

5.2 循环队列

5.2.1 循环队列（Circular Queue）

如果不断出队，队头左边的空间就失去了作用，那队列的容量就会变得越来越小。

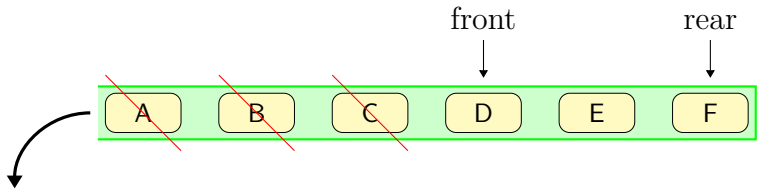

图 5.3: 队列存在的问题

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

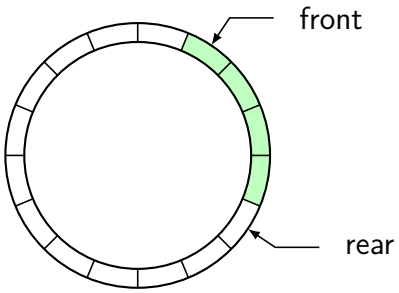

图 5.4: 循环队列

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

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

假设数组的最大容量为 MAX：

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

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

入队

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

出队

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

5.3 双端队列

5.3.1 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。

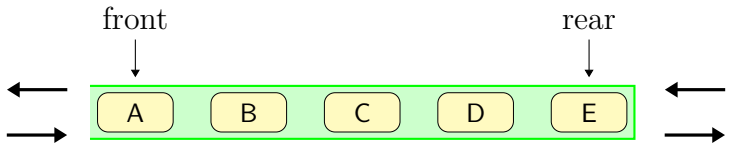

图 5.5: 双端队列

双端队列

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

    def is_empty(self):
        return not self.data

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

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

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

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

    def get_front(self):
```

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