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。

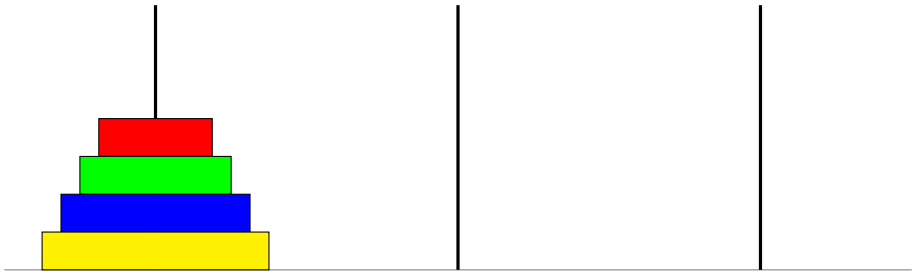

图 1.1: 汉诺塔

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?

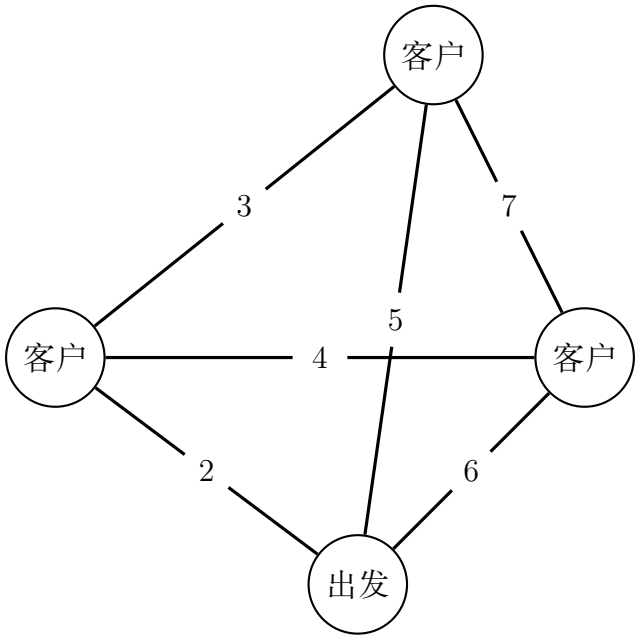

图 1.2: 快递客户路线

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：

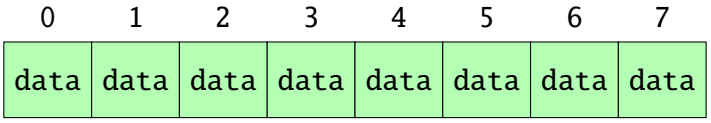

图 2.1: 数组

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 " + index + " out of bounds for length " + size
        );
    }

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

    for (int i = size - 1; i >= index; i--) {
```

```
20          data[i + 1] = data[i];
21      }
22
23      data[index] = elem;
24      size++;
25  }
```

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(
4               "Index " + index + " out of bounds for length " + size
5           );
6       }
7
8       T elem = data[index];
9       for (int i = index; i < size - 1; i++) {
10          data[i] = data[i + 1];
11      }
12      size--;
13      return elem;
14  }
```

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

3.1.1 单向链表（Singly Linked List）

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

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

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

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

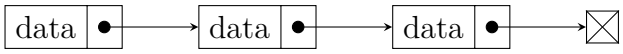

图 3.1: 单向链表

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

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

3.1.2 双向链表（Doubly Linked List）

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

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

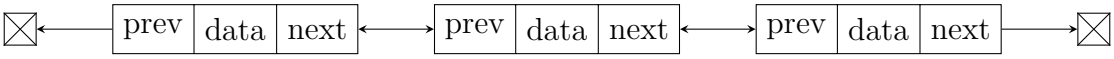

图 3.2: 双向链表

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

3.1.3 循环链表（Circular Linked List）

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

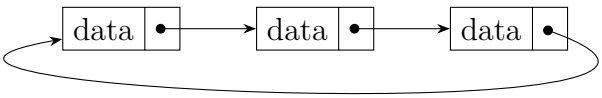

图 3.3: 单向循环链表

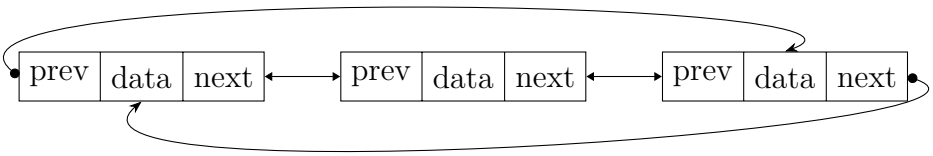

图 3.4: 双向循环链表

3.2 链表的增删改查

3.2.1 查找结点

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

链表中的数据只能按顺序进行访问，最坏的时间复杂度是 $O(n)$。

查找结点

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

3.2.2 更新结点

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

更新结点

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

```
4       for(int i = 0; i < pos; i++) {
5           temp = temp->next;
6       }
7       temp->data = val;
8   }
```

3.2.3 插入结点

链表插入结点，分为 3 种情况：

尾部插入

把最后一个结点的 next 指针指向新插入的结点。

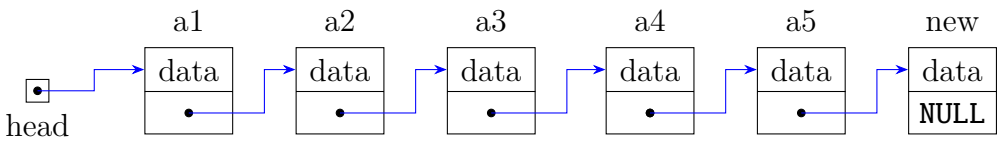

图 3.5: 尾部插入

头部插入

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

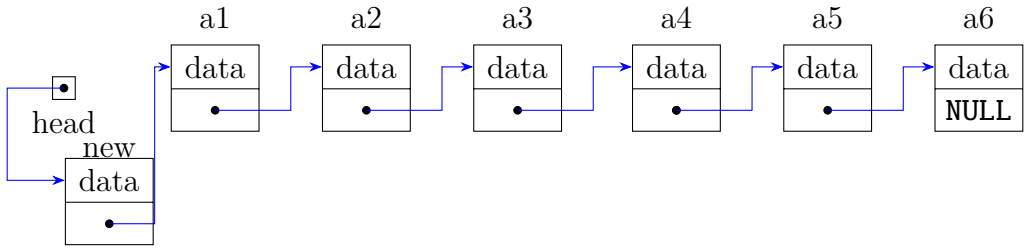

图 3.6: 头部插入

中间插入

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

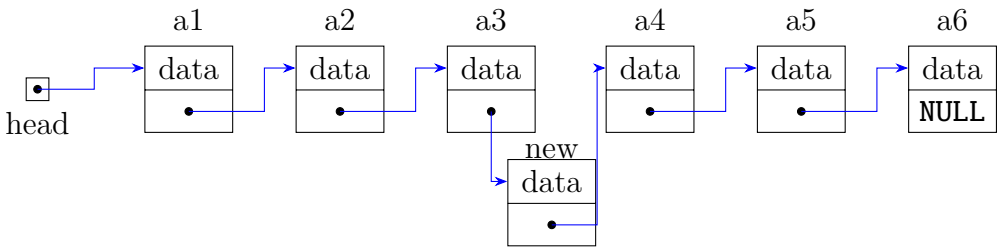

图 3.7: 中间插入

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

3.2.4 删除结点

链表的删除操作也分 3 种情况：

尾部删除

把倒数第二个结点的 next 指针指向空。

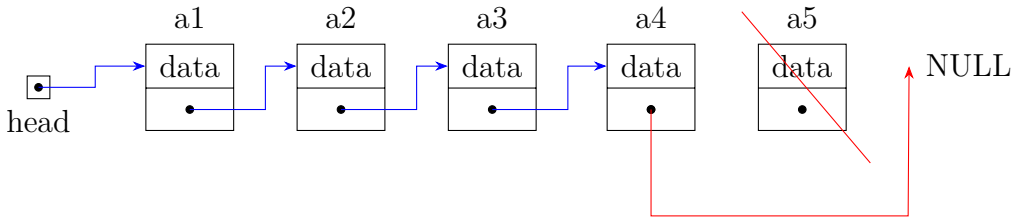

图 3.8: 尾部删除

头部删除

把链表的头结点设置为原先头结点的 next 指针。

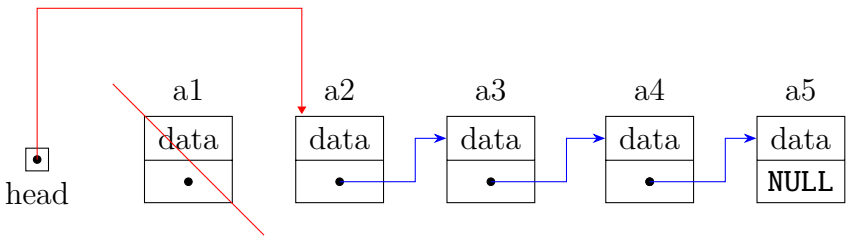

图 3.9: 头部删除

中间删除

把要删除的结点的前置结点的 next 指针，指向要删除结点的下一个结点。

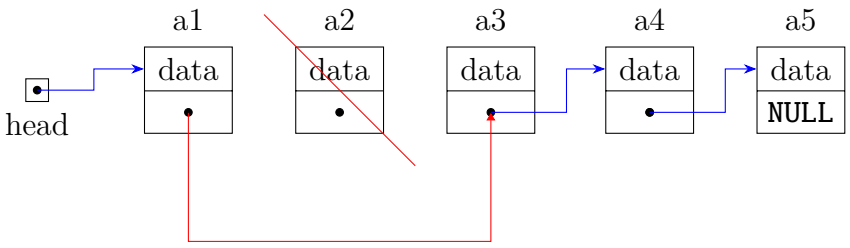

图 3.10: 中间删除

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

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

3.3 带头结点的链表

3.3.1 带头结点的链表

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

插入结点

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

删除结点

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

```
10 }
```

3.3.2 数组 VS 链表

数据结构没有绝对的好与坏，数组和链表各有千秋。

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

表 3.1: 数组 VS 链表

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

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

3.4 倒数第 k 个结点

3.4.1 倒数第 k 个结点

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

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

倒数第 k 个结点

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

3.5 环形链表

3.5.1 环形链表

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

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

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

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

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

环形链表

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

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

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

3.6 反转链表

3.6.1 逆序输出链表

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

逆序输出链表

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

3.6.2 反转链表

反转一个链表需要调整链表中指针的方向。

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

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

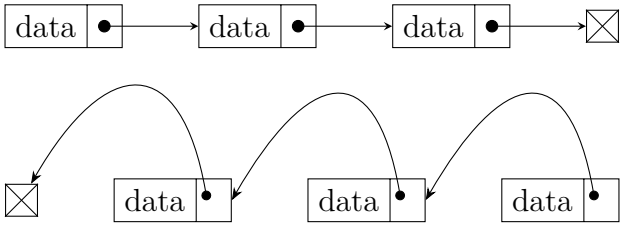

图 3.11: 反转链表

反转链表（递归）

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

反转链表（迭代）

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