数据结构与算法

Data Structure and Algorithm

极夜酱

目录

Part I

基础篇

Chapter 1 数据结构与算法

1.1 算法

1.1.1 算法（Algorithm）

算法是一个很古老的概念，最早来自数学领域。

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

老师以为，熊孩子会按部就班地一步一步计算：

$$1+2=3$$

$$3+3=6$$

$$6+4=10$$

$$10+5=15$$

$$\cdots$$

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

"老师，我算完了！结果是 50005000，对不对?"

"这，这，这……你小子怎么算得这么快？我读书多，你骗不了我的！"

看着老师惊讶的表情，熊孩子微微一笑，讲出了他的计算方法。

首先把从 1 到 10000 这 10000 个数字两两分组相加：

$$1 + 10000 = 10001$$

$$2 + 9999 = 10001$$

$$3 + 9998 = 10001$$

$$4 + 9997 = 10001$$

$$\cdots$$

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

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

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

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

算法有 5 个特点：

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

1.1.2 算法描述

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

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

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

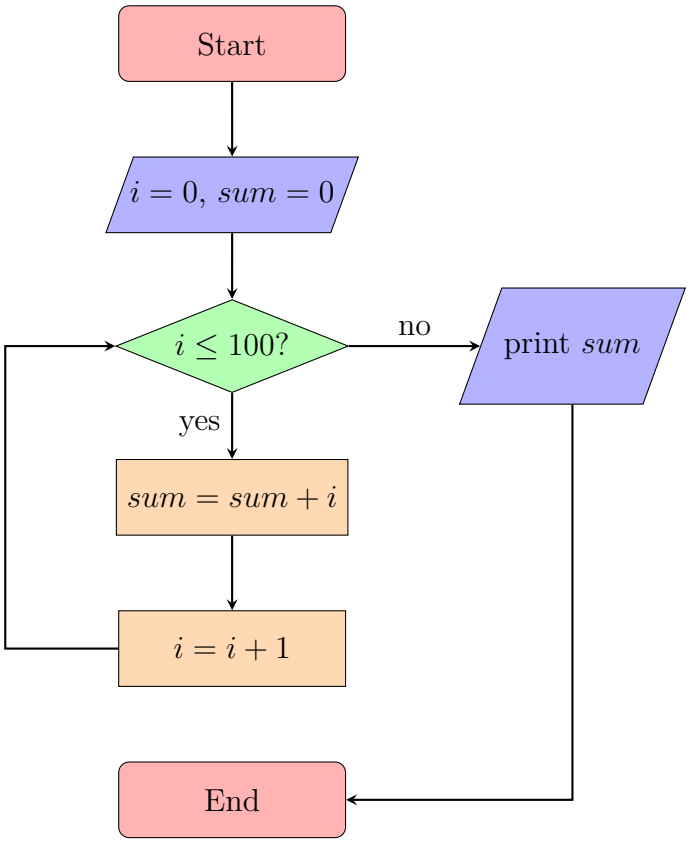


图 1.1: 计算 $\sum_{i=1}^{100} i$ 的流程图

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

Algorithm 1 插入排序

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

1.2 算法效率

1.2.1 算法效率

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

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

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

“小灰，收拾东西走人，明天不用来上班了!”

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

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

- 最好情形时间复杂度
- 最坏情形时间复杂度
- 平均情形时间复杂度

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

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

1.2.2 时间复杂度（Time Complexity）

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

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

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

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

1.2.3 空间复杂度（Space Complexity）

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

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

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

1.3 基础算法

1.3.1 暴力枚举

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

鸡兔同笼

上有三十五头，下有九十四足，问鸡兔各几何？

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

百钱买百鸡

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

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

```
    }
}
```

1.3.2 字符串逆序

将一个字符串中的字符顺序颠倒过来实现逆序。

字符串逆序

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

1.3.3 随机算法

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

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

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

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

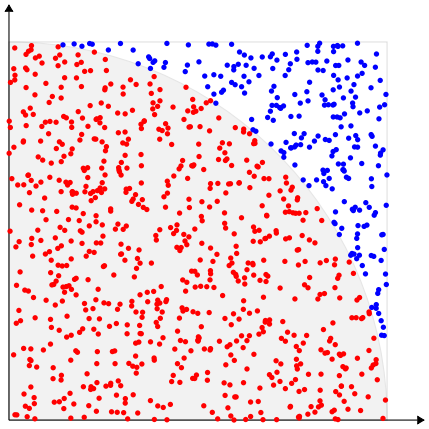


图 1.2: 蒙特卡洛算法

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

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

在生成大量随机点的前提下能得到尽可能接近圆周率的值。

蒙特卡罗算法

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

1.4 数据结构

1.4.1 数据结构（Data Structure）

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

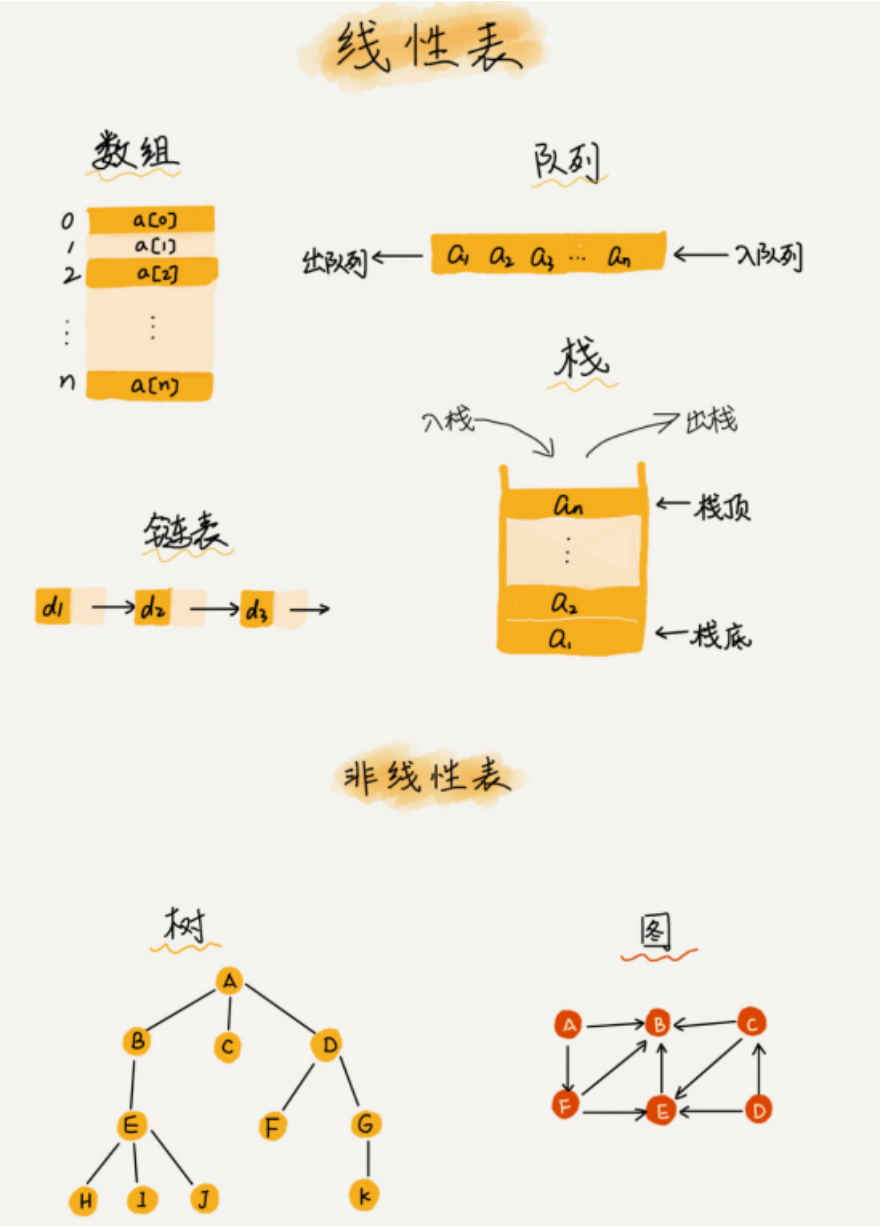


图 1.3: 常用数据结构

Chapter 2 数组

2.1 数组

2.1.1 数组（Array）

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

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

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

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

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

更新数组元素

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

2.2 查找算法

2.2.1 顺序查找（Sequence Search）

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

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

最好的情况就是在第一个位置就找到，算法时间复杂度为 $O(1)$。

最坏情况是关键字不存在，需要进行 n 次比较，时间复杂度为 $O(n)$。

平均查找次数为 $(n+1)/2$，平均时间复杂度为 $O(n)$。

顺序查找

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

2.2.2 二分查找（Binary Search）

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

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

二分查找法的时间复杂度为 $O(logn)$。

二分查找

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

2.3 数组元素插入与删除

2.3.1 插入元素

在数组中插入元素存在 3 种情况：

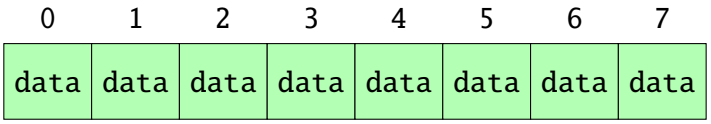


图 2.1: 数组

尾部插入

直接把插入的元素放在数组尾部的空闲位置即可。

中间插入

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

插入元素

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

超范围插入

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

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

2.3.2 删除元素

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

删除元素

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

数组的删除操作，由于只涉及元素的移动，时间复杂度为 $O(n)$。

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

Chapter 3 链表

3.1 链表

3.1.1 单向链表（Singly Linked List）

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

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

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

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

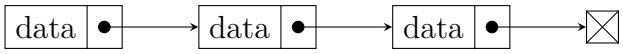


图 3.1: 单向链表

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

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

3.1.2 双向链表（Doubly Linked List）

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

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

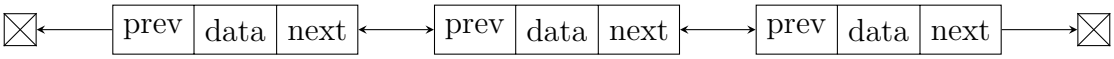


图 3.2: 双向链表

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

3.1.3 循环链表（Circular Linked List）

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

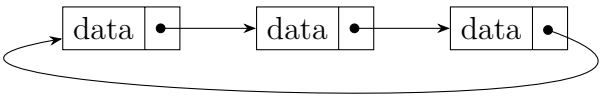


图 3.3: 单向循环链表

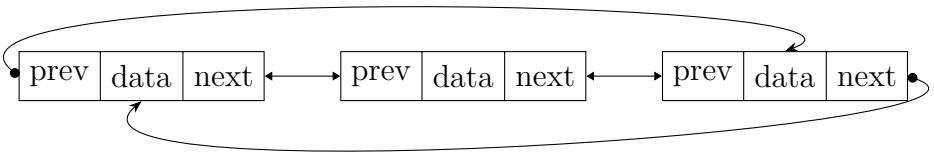


图 3.4: 双向循环链表

3.2 链表的增删改查

3.2.1 查找结点

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

链表中的数据只能按顺序进行访问，最坏的时间复杂度是 $O(n)$。

查找结点

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

3.2.2 更新结点

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

更新结点

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

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

3.2.3 插入结点

链表插入结点，分为 3 种情况：

尾部插入

把最后一个结点的 next 指针指向新插入的结点。

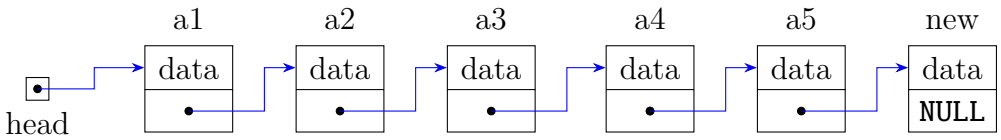


图 3.5: 尾部插入

头部插入

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

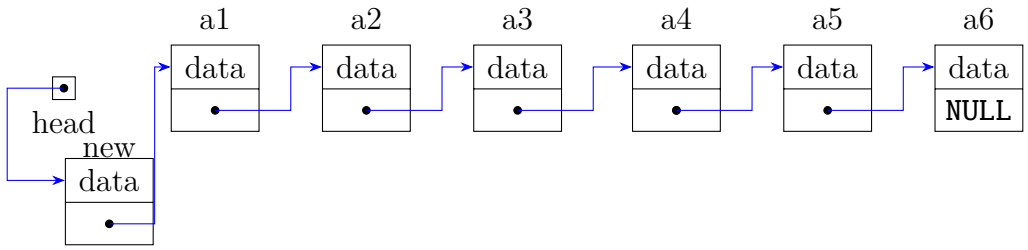


图 3.6: 头部插入

中间插入

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

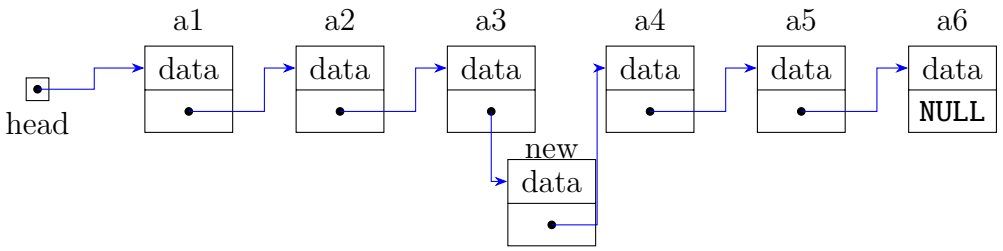


图 3.7: 中间插入

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

3.2.4 删除结点

链表的删除操作也分 3 种情况：

尾部删除

把倒数第二个结点的 next 指针指向空。

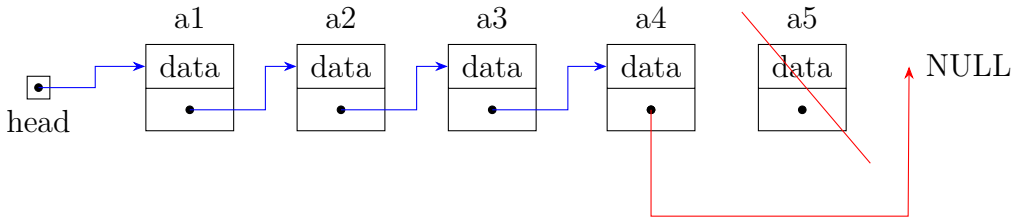


图 3.8: 尾部删除

头部删除

把链表的头结点设置为原先头结点的 next 指针。

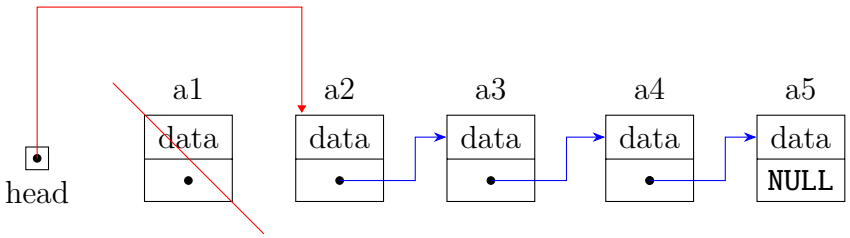


图 3.9: 头部删除

中间删除

把要删除的结点的前置结点的 next 指针，指向要删除结点的下一个结点。

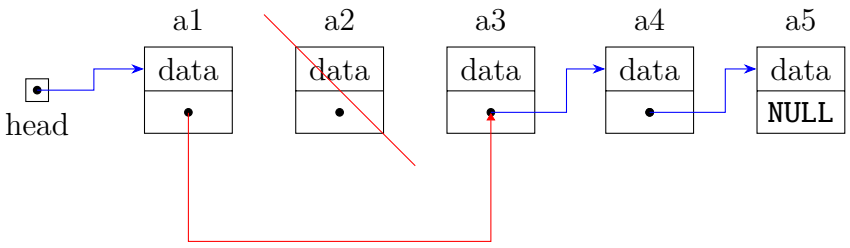


图 3.10: 中间删除

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

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

3.3 带头结点的链表

3.3.1 带头结点的链表

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

插入结点

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

删除结点

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

```
10 }
```

3.3.2 数组 VS 链表

数据结构没有绝对的好与坏，数组和链表各有千秋。

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

表 3.1: 数组 VS 链表

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

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

3.4 倒数第 k 个结点

3.4.1 倒数第 k 个结点

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

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

倒数第 k 个结点

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

3.5 环形链表

3.5.1 环形链表

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

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

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

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

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

环形链表

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

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

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

3.6 反转链表

3.6.1 逆序输出链表

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

逆序输出链表

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

3.6.2 反转链表

反转一个链表需要调整链表中指针的方向。

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

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

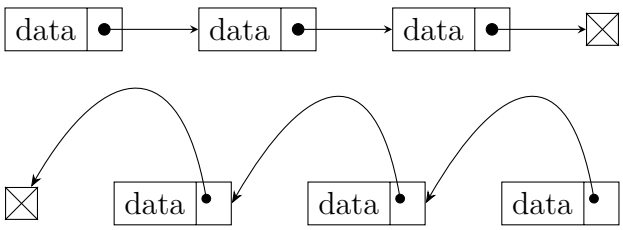


图 3.11: 反转链表

反转链表（递归）

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

反转链表（迭代）

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

3.7 跳表

3.7.1 跳表

给定一个有序链表，如何根据给定元素的值进行高效率查找?

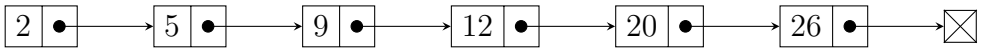


图 3.12: 有序链表

是不是可以用二分查找，先找到中间元素?

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

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

其实一个链表也可以拥有自己的目录，或者说索引。

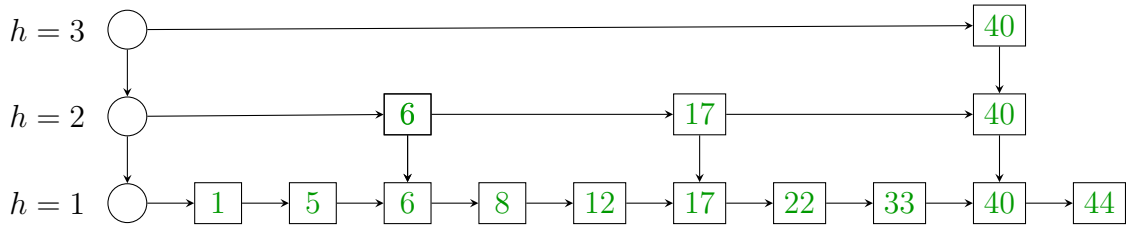


图 3.13: 跳表

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

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

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

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

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

像这样基于链表改造的数据结构，有一个全新的名字，叫做跳表。

3.7.2 跳表插入结点

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

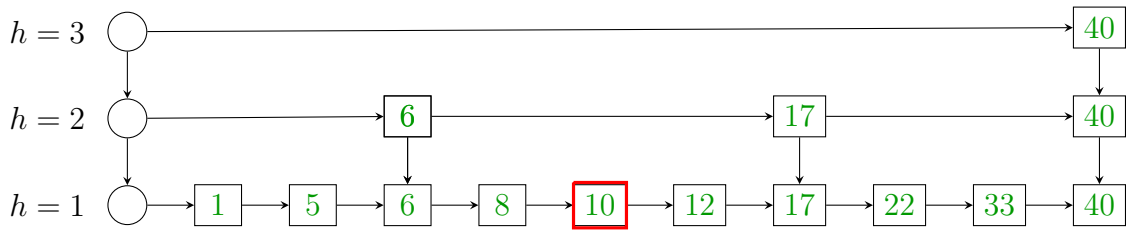


图 3.14: 插入结点 10

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

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

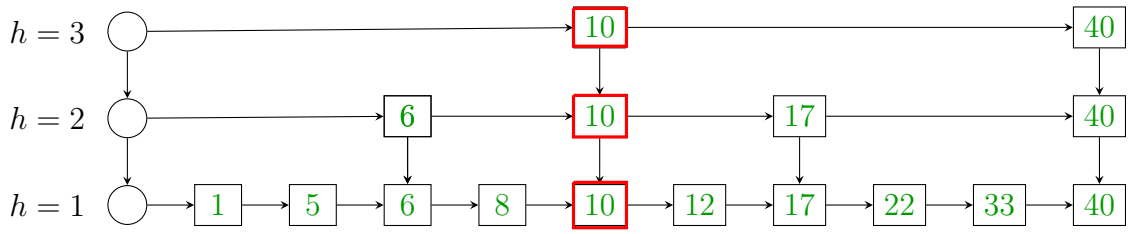


图 3.15: 晋升结点

3.7.3 跳表删除结点

至于跳表删除结点的过程，则是相反的思路。

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

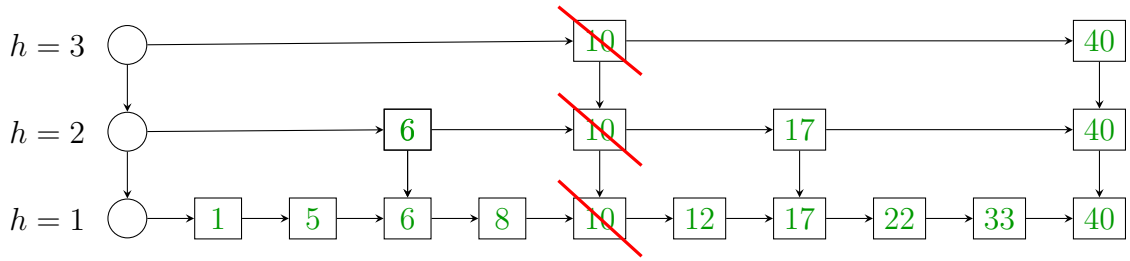


图 3.16: 删除结点

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

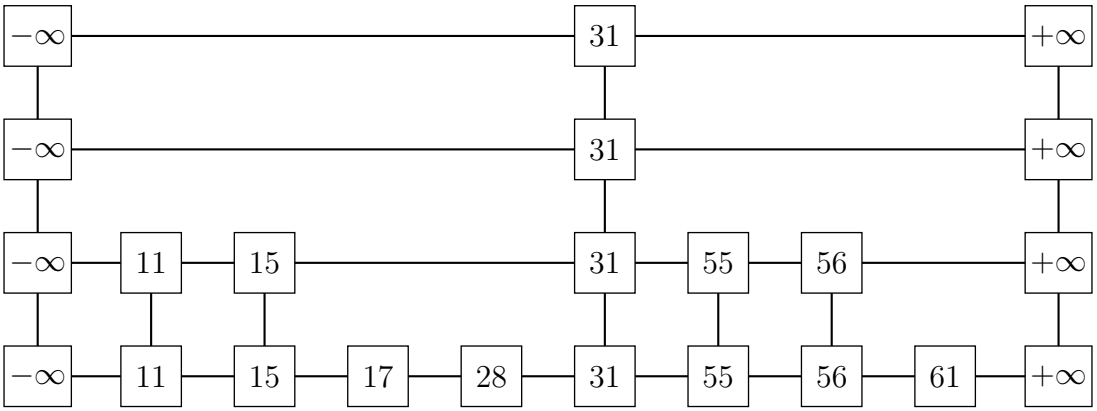


图 3.17: 跳表

Chapter 4 栈

4.1 栈

4.1.1 栈（Stack）

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

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

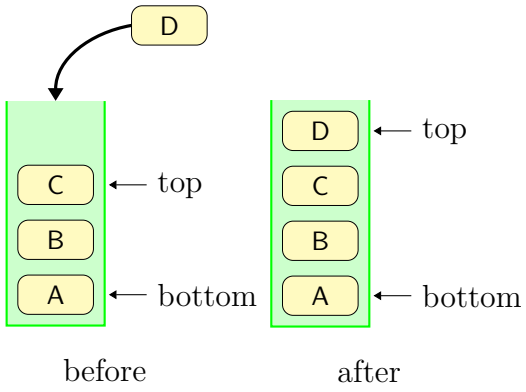


图 4.1: 栈

栈这种数据结构既可以用数组来实现，也可以用链表来实现。

4.1.2 顺序栈

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

进行入栈操作时，栈顶指针 +1；出栈时，栈顶指针-1。

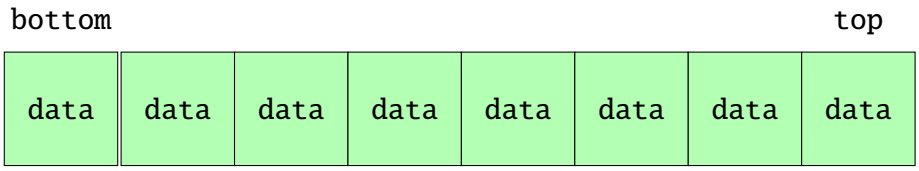


图 4.2: 顺序栈

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

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

4.1.3 链式栈

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

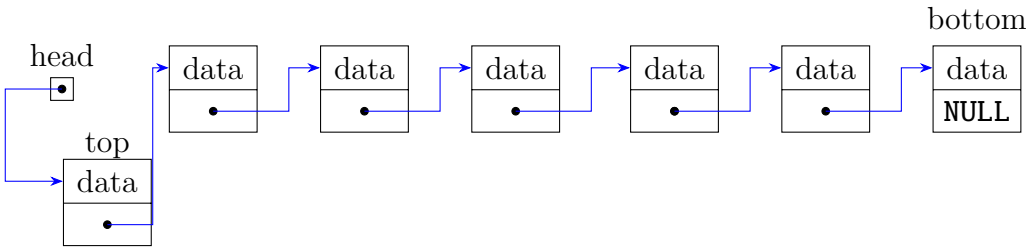


图 4.3: 链式栈

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

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

4.1.4 栈的应用

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

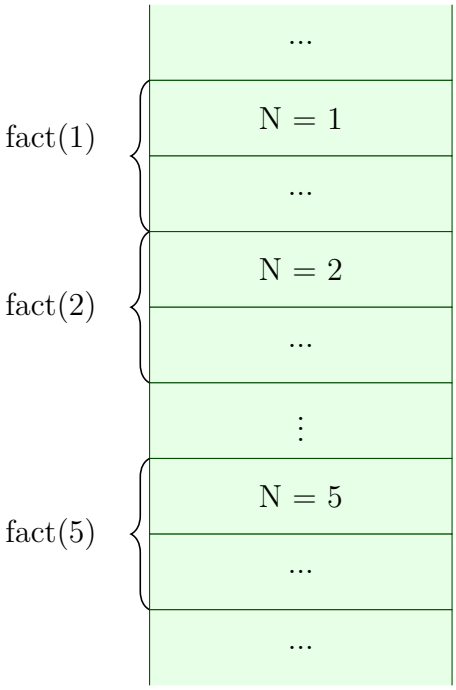


图 4.4: 函数调用栈

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

图 4.5: 面包屑导航

4.2 入栈与出栈

4.2.1 入栈（Push）

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

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

入栈

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

4.2.2 出栈（Pop）

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

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

出栈

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

4.3 最小栈

4.3.1 最小栈

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

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

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

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

最小栈

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

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

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

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

4.4 括号匹配

4.4.1 括号匹配

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

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

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

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

括号匹配

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

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

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

4.5 表达式求值

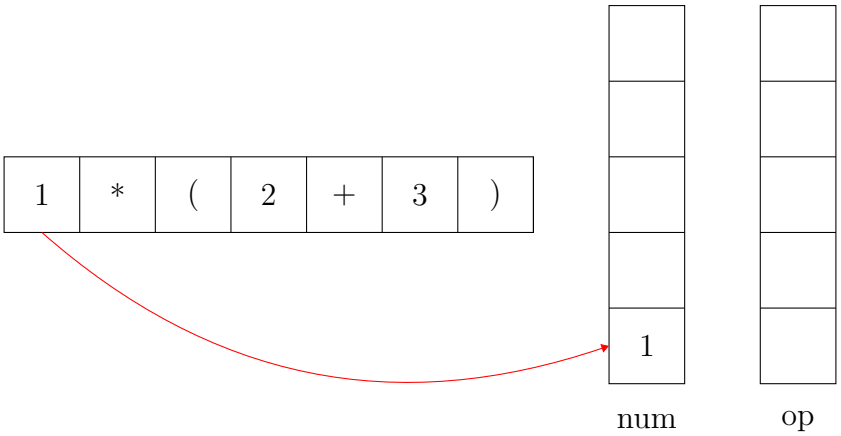
4.5.1 表达式求值

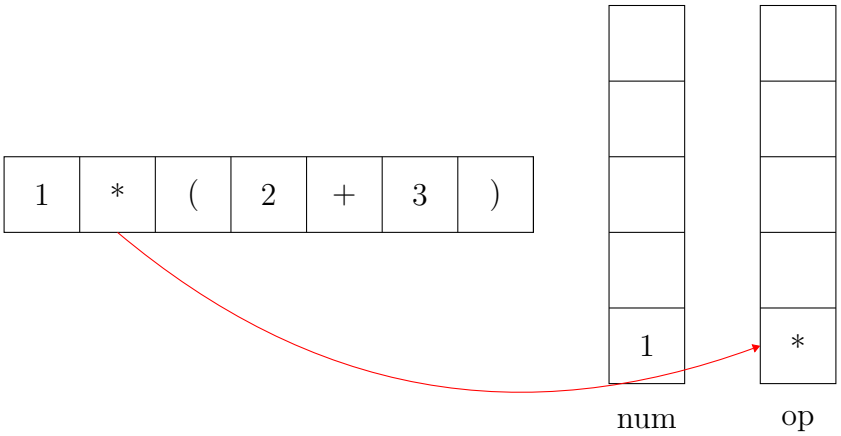
逆波兰表达式是一种后缀表达式，所谓后缀就是指运算符写在运算数的后面。平常使用的算式则是一种中缀表达式，如 $(1+2)*(3+4)$，该算式的逆波兰表达式写法为 $1\ 2+3\ 4+*$。

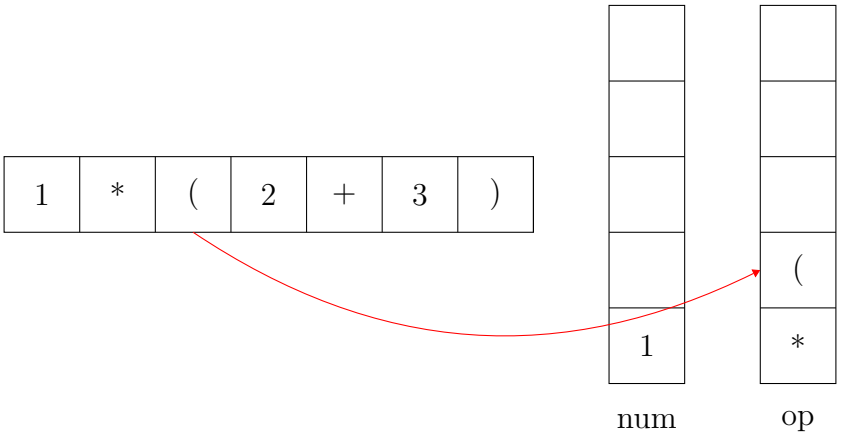
逆波兰表达式的优点在于去掉了中缀表达式中的括号后表达式无歧义，因此适合用栈操作运算。遇到数字则入栈，遇到算符则取出栈顶两个数字进行计算，并将结果压入栈中。

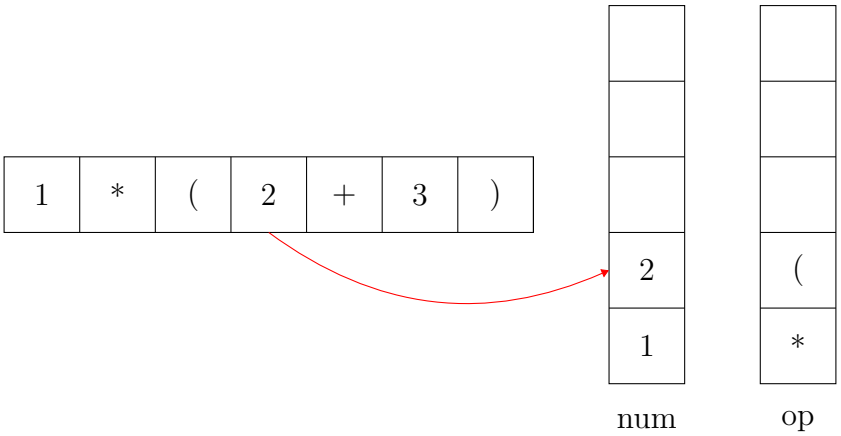
在对中缀表达式求值时，一般都会将其转换为后缀表达式的形式，转换过程同样需要用到栈，规则如下：

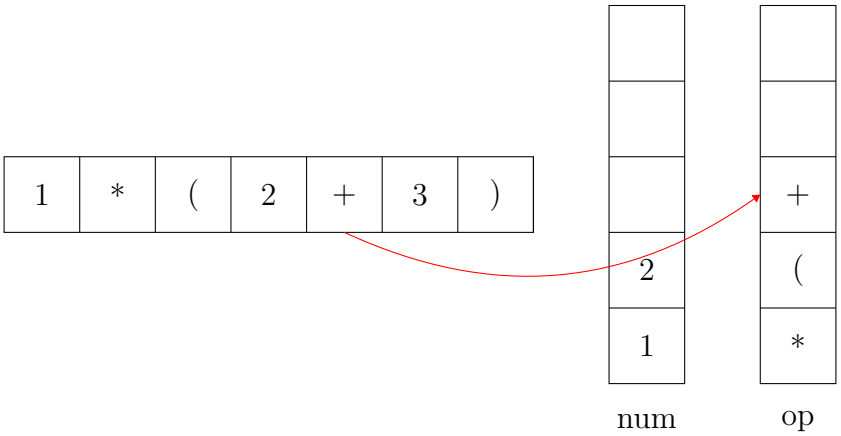
1. 如果遇到操作数，就直接将其输出。
2. 如果遇到左括号，将其放入栈中。
3. 如果遇到右括号，则一直出栈并输出，直到遇到左括号为止。注意，左括号只出栈并不输出。
4. 如果遇到任何其它的运算符，如果为栈为空，则直接入栈。否则从栈中出栈元素并输出，直到遇到优先级更低的元素（或者栈为空）位置。在出栈完这些元素后，再将当前遇到的运算符入栈。有一点需要注意，只有在遇到右括号的情况下才将左括号出栈，其它情况都不会出栈左括号。
5. 如果读取到了表达式的末尾，则将栈中所有元素依次出栈输出。

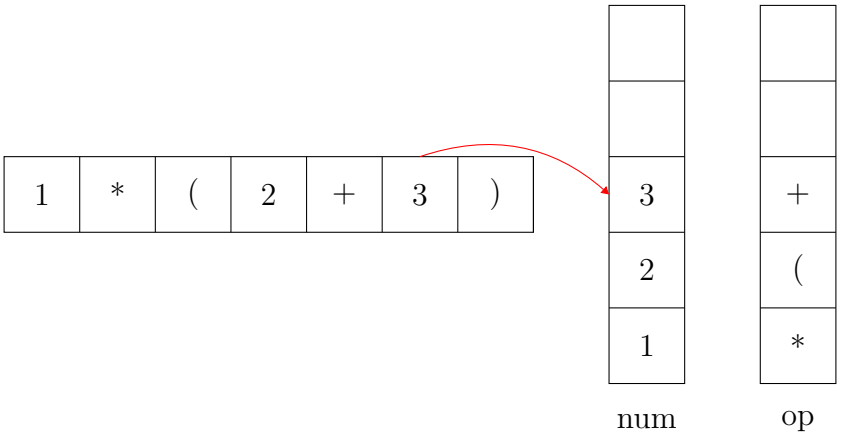

1 * (2 + 3)
1
num
op

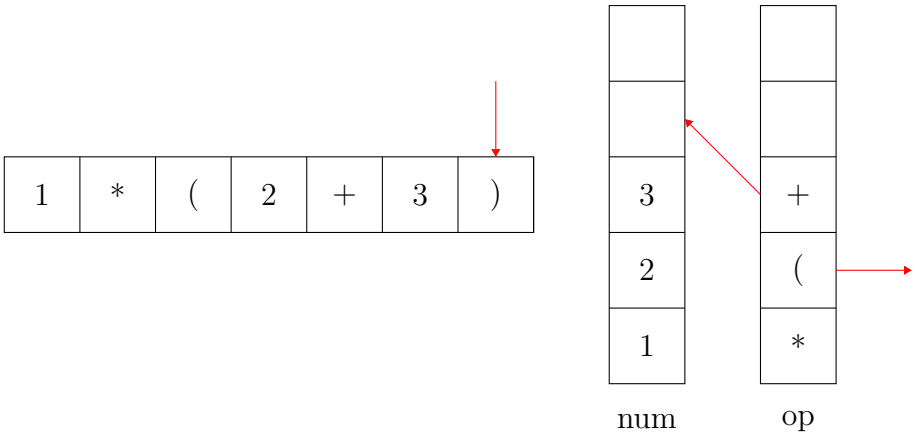

1 * (2 + 3)
1
*
num
op

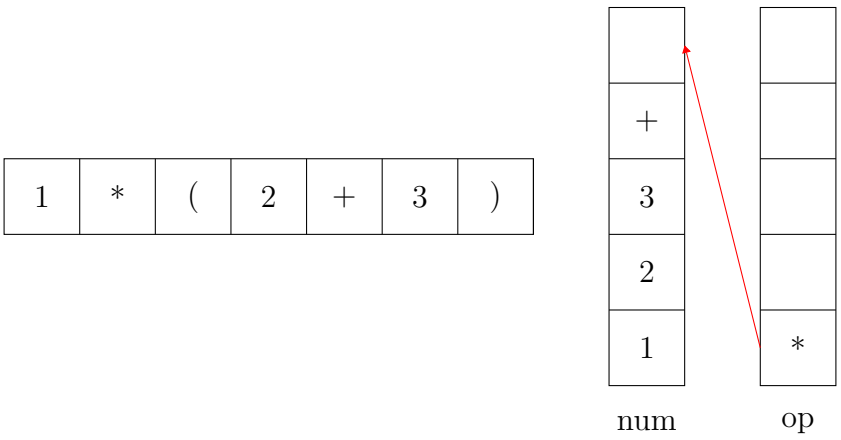

1 * (2 + 3)
(
1
*
num
op

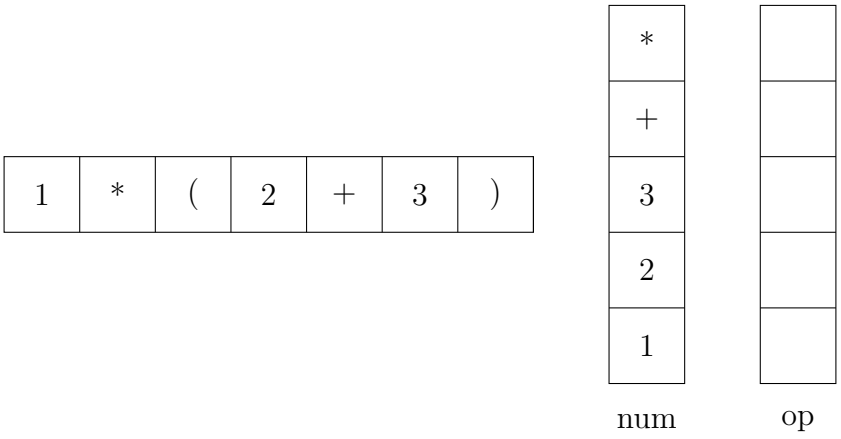

1 * (2 + 3)
2
1
num
(
*
op


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


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







表达式求值

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

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

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

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

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

Chapter 5 队列

5.1 队列

5.1.1 队列（Queue）

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

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

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