数据结构与算法

Data Structure and Algorithm

极夜酱

目录

Part I

基础篇

Chapter 1 栈

1.1 栈

1.1.1 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）。

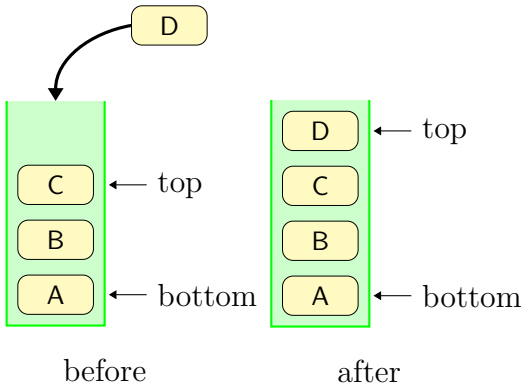

图 1.1: 栈

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。

1.1.2 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。

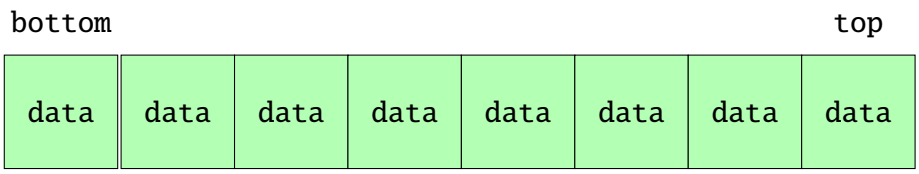

图 1.2: 顺序栈

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。

1.1.3 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。

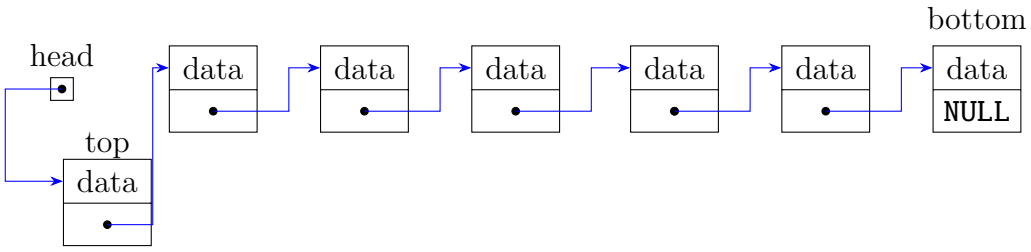

图 1.3: 链式栈

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。

1.1.4 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。

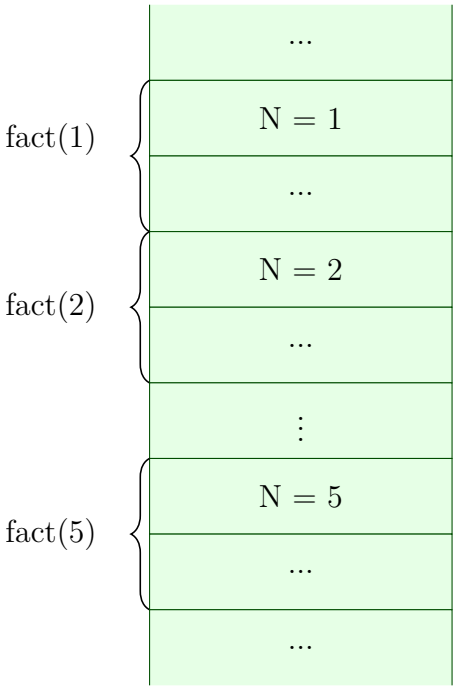

图 1.4: 函数调用栈

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 1.5: 面包屑导航

1.2 入栈与出栈

1.2.1 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 }
```

1.2.2 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 }
```

1.3 最小栈

1.3.1 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]
```

1.4 括号匹配

1.4.1 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
```

1.5 表达式求值

1.5.1 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	)