栈和队

Mitake Moca

栈

我们先讲栈这个数据结构都有哪些操作，再来讲为什么会需要栈这个数据结构。

栈可以理解为是一种退化版的链表，因为它能做到的，链表也全都能做到。链表可以在任意的位置进行常数时间复杂度的插入与删除（想做到常数时间复杂度，一般需要你同时维护前驱和后继两个指针域），但是栈不一样，栈只能在一端进行插入与删除。可以把栈视为**只允许在尾结点处进行插入（且插入在尾结点后）和删除的链表**，只不过对于这个位置我们有了一个更专业的名词，叫做**栈顶**，其余的位置都不能进行插入和删除。在其余的位置中，有一个位置很特殊，即对应于链表头结点的地方，我们称之为**栈顶**，如下图所示：

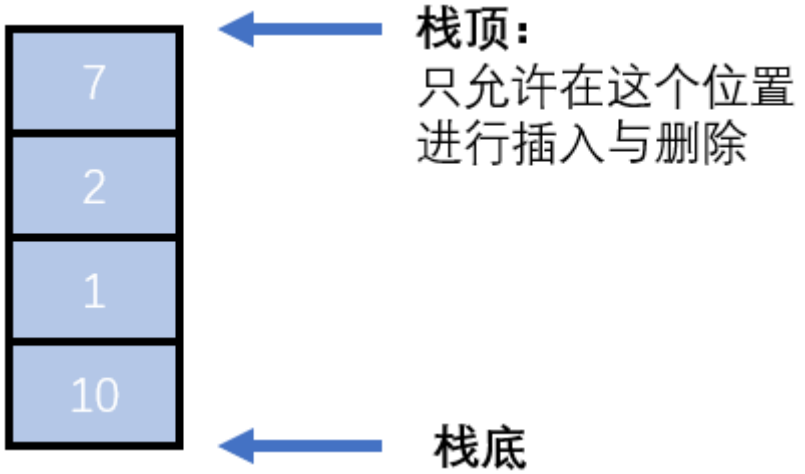

图 1 - 栈顶和栈底

栈所支持的操作都是在栈顶发生的，包括：插入一个元素（push，术语叫**入栈**）、删除一个元素（pop，术语叫**出栈**）、看看栈顶的元素（peek）。这些操作都非常直观，同学们可以形象地用一堆盘子来理解：这堆盘子的最上面就相当于栈顶，最下面就相当于栈底；入栈就是在上面再放一个盘子；出栈就是把最上面的盘子拿走。

比如，我们现在让元素 8 加入上面的图 1 所表示的栈：

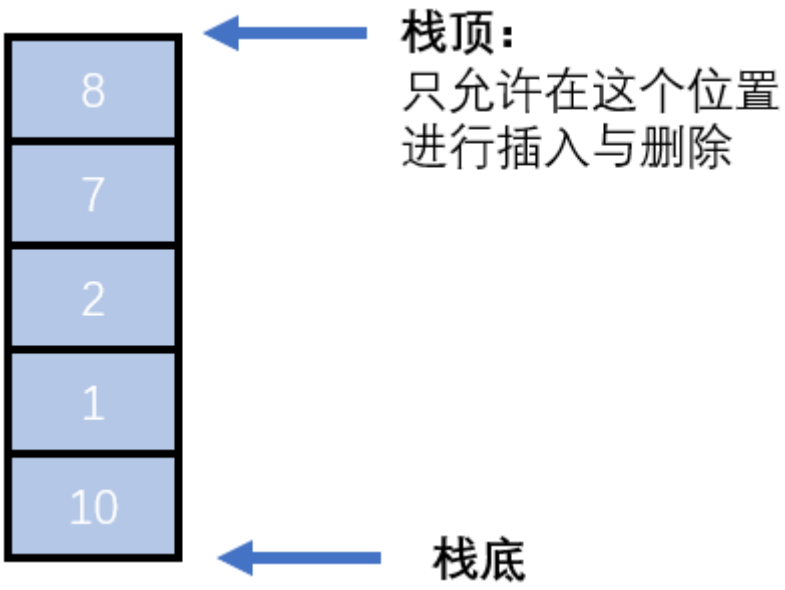

图 2 - 元素 8 入栈

出栈操作也很简单了，比如我们现在让图 2 的栈执行出栈操作，它就会重新变成图 1。

说完了栈支持的操作，那为什么要设计栈这个数据结构呢?

其实是因为栈满足**后进先出**的性质，许多问题需要**优先处理最近发生的事件**，栈完美契合这种场景。比如在程序中，如果我们用函数 A 调用了函数 B，也就是有一段类似于下面的代码：

```
int A() {
    ...
    B();
    ...
}
```

那么只有当对该次函数 B 的调用结束后，才能够回到 A 中继续执行下面的代码。

又比如我们使用 $\mathrm{ctrl}+\mathrm{z}$ 对文档进行撤销操作时，实际上对于编辑操作而言它们也是满足后进先出性质的，即最后一步操作最先被撤销。

栈的实现

有了上面的解释后，你可能会想：既然栈能做的，链表都能做；栈不能做的，链表也可以做，那我还用栈干什么呢，我以后为什么不直接全部都用链表呢？实际上，相较于链表而言，栈不仅更加安全（向外部隐藏了除栈顶外的一切细节），而且实现也更加简单。

我们在写栈通常使用数组来实现，因为栈的连续性更加密切，即使入栈出栈若干次，也并不会改变其余元素的连续性（相比于链表，可以在任意的位置进行插入删除，如果用数组实现的话会破坏连续性，除非再维护一个 `next` 数组）。下面请看我们的代码：

```
// top 维护栈顶，栈底一直是 0
int stk[1005], top = -1;

void push(int a) {
    stack[++top] = a;
```

```
}

int pop() {
    return stack[top--];
}

int peek() {
    return top == -1 ? -1 : return stack[top];
}
```

相比于链表的代码而言，实在是太简洁了。

队

相比于栈，队其实更好理解，因为它更贴近我们的生活。数据结构中的队就像我们生活中的队，想象一下你在食堂排队打饭时：新来的人必须站在队伍后面（enqueue，入队）；队伍最前面的人打完饭离开（dequeue，出队）；想知道谁是下一个打饭的人，就看看队首（front）。

与栈进行对比：

- 栈在一端进行插入和删除（栈顶）；队在一端进行插入（队尾），另一端进行删除（队首）
- 栈是后进先出（后进来的先处理）；队是先进先出（先来的先处理）
- 二者的功能都被链表所支持\

在计算机世界中，有很多场景可能用到队，比如任务排队（打印机打印）、广度有限搜索等。随着同学们的学习，同学们在以后的生活中还会了解到双端队列、优先队列等。

队的实现

我们在这只给大家提供一个用数组实现的普通队了，其实为了节省内存，通常的实现会采用循环队列。

```
// que 代表队数组，front 代表队首，rear 代表队尾
int que[1005];
int front = 0, rear = -1;

void enqueue(int a) {
    queue[++rear] = a;
}

int dequeue() {
    return queue[front++];
}

int front() {
    return front >= rear ? que[front] : -1;
}
```

```
int getLen() {
    return rear - front + 1;
}
```