LeetCode Summary

本文件是在解决LeetCode算法问题过程中的总结与反思汇总。

Little Tips

- 如果要使用线性复杂度在一维数组中解决问题，优先考虑滑动窗口和双指针，以及动态规划。
- Top K的问题求解时，优先考虑堆(优先队列)来解决，因为它会自动维护前K个元素的有序性。这里有一些例外，比如215.数组中的第K个最大元素，就使用了快速选择算法(QuickSelect)来快速地定位第K大的元素。

贪心

排序

值得注意的是排序类问题并不是简单的排序算法的实现，而是基于基础排序算法而诞生的一系列问题，其中以快速排序算法的划分法诞生的变式问题最多。

215.数组中的第K个最大元素

这是典型的一道Top K求解问题，一般来说这类问题使用堆(优先队列)来解决即可。但是这道题要求实现O(n)的算法，所以堆排序在时间复杂度上不符合要求。

事实上，这道题使用的是基于快速排序的快速选择算法，使用快速排序中的划分法，一点点逼近数组中第K大的数字。在这个过程中还要引入随机选择划分元的方法来进一步降低极端情况出现的可能性，从而使得算法的整体期望复杂度降至O(n)，这种算法叫做快速选择(quick select)算法。

如果遇到要求第K个最大或者最小元素，而**不期望给出前K个最大最小元素**时，使用快速选择算法是一个比较好的选择。

含注释的代码如下：

```
class Solution {
    /*这里假设区间是左闭右闭的[Start, End]*/
    int partition(vector<int>& nums, int Start, int End)
    {
        /*随机选择划分元，可以使得算法复杂度降至O(n)*/
        int RandomIndex = (rand() % (End - Start + 1)) + Start;

        /*Attention : 选好划分元之后，和最左元素交换，方便后续操作*/
        swap(nums[Start], nums[RandomIndex]);
        int Tmp = nums[Start];
        while(Start < End)  // 划分过程，不再详述
        {
            while(Start < End and nums[End] > Tmp) --End;
```

```
            nums[Start] = nums[End];
            while(Start < End and nums[Start] <= Tmp) ++Start;
            nums[End] = nums[Start];
        }
        nums[Start] = Tmp;
        return Start;
    }

    /*快速划分算法: quick select algorithm*/
    int quickSelect(vector<int>& nums, int k, int Start, int End)
    {
        int Pivot = partition(nums, Start, End);
        /* 如果随机选择的划分元正好是第k小的元素，直接返回 */
        if(Pivot == k - 1)
            return nums[Pivot];
        else if(Pivot < k - 1)
            /* 否则向一侧区间进行递归 */
            return quickSelect(nums, k, Pivot + 1, End);
        else
            return quickSelect(nums, k, Start, Pivot - 1);

    }
public:
    int findKthLargest(vector<int>& nums, int k)
    {
        /* 初始化随机数种子 */
        srand((unsigned)time(NULL));
        int n = nums.size();
        /* 第k大数字也是第n+1-k小的数字 */
        return quickSelect(nums, n + 1 - k, 0, n - 1);
    }
};
```

912.堆排序

堆排序是解决Top K排序的重要方法，堆排序在面试时也往往需要直接手撕。

值得注意的是，**堆一棵完全二叉树**，所以在静态存储的树结构中，一个节点编号和它的左右孩子编号之间存在定量关系，这是整个堆排序算法运行的重要原理。一般来说有两种换算方法：

- 如果数组下标从1开始，那么左孩子 = 2 * index，右孩子 = 2 * index + 1
- 如果数组下标从0开始，那么左孩子 = 2 * index + 1，右孩子 = 2 * index + 2

一般来说数组下标都是从0开始的，所以一般选择第二种换算法。

堆排序涉及的核心步骤就是堆的调整(adjustHeap)，要重点掌握。

```
/* 向下调整堆的函数，[Low, High]划定了调整范围*/
void adjustHeap(vector<int>& nums, int Low, int High)
{
    // 取出Low节点和其左孩子，注意左孩子的下标是如何计算的
```

```
    int i = Low, j = i * 2 + 1;
    while(j <= High)
    {
        // 调整j为左右孩子中的较大值，准备和i进行交换
        if(j + 1 <= High and nums[j + 1] > nums[j])
            j = j + 1;

        // 如果左右孩子的较大值大于父亲节点，那么交换之并继续向下调整
        if(nums[i] < nums[j])
        {
            swap(nums[i], nums[j]);
            i = j;
            j = i * 2 + 1;
        }
        // 否则调整到此结束
        else
            break;
    }
}
```

在实现了向下调整堆的函数之后，接下来就是重建堆的函数，即从数组n/2的位置(n是是数组的总长度)开始倒序进行调整直到第一个节点，代码如下所示：

```
void buildHeap(vector<int>& nums)
{
    int n = nums.size();
    // 从中间节点进行倒序调整直到第一个节点
    for(int i = n / 2 ; i >= 0 ; --i)
        adjustHeap(nums, i, n - 1);
}
```

在实现了上述两个函数之后，堆排序就很简单了：

- **首先重建堆，这时位于nums[0]这个位置的一定是最大元素，将其与最后一个元素进行交换，那么此时它就到了它应该在的位置上**
- **对[0, n-2]这个范围进行向下调整，从而再一次得到了一个堆**
- **...**
- **重复这个过程直至所有元素都已经归位**

```
//堆排序
void heapSort(vector<int>& nums)
{
    // 重建堆
    buildHeap(nums);
    int n = nums.size();
    for(int i = n - 1 ; i >= 0 ; --i)
    {
        // 将当前堆的最大元素交换到它的应有位置上，并向下调整堆
        swap(nums[0], nums[i]);
```

```
            adjustHeap(nums, 0, i - 1);
        }
    }
```

链表

206.反转链表

经典题，有递归和迭代两种做法，都必须掌握。

迭代解法比较符合人的逻辑思维习惯，从前向后依次将节点反转。 这里需要额外注意的一点是，原先头节点的next指针必须置为*nullptr*。为此，我们将Left指针的初始值置为nullptr，这样可以直接完成操作。

```
class Solution {
public:
    ListNode* reverseList(ListNode* head)
    {
        if(not head)
            return head;
    /*将Left初始值置为nullptr*/
    /*可以直接保证翻转完之后最后一个节点的next指针是nullptr*/
        ListNode* Left = nullptr;
        ListNode* Right = head;
        while(Right)
        {
            ListNode *Tmp = Right->next;
            Right->next = Left;
            Left = Right;
            Right = Tmp;
        }
        return Left;
    }
};
```

递归法就比较**逆天**了，递归法的本质是递推+回归。所以这个过程首先假设后面的链表全部翻转完毕了，在此基础上再去考虑要对返回的指针进行怎样的操作。

```
class Solution {
public:
/*
    注意：递归函数的返回值是翻转后的链表的头节点
    这点非常重要
*/
    ListNode* reverseList(ListNode* head)
    {
        if(not head or not head->next)
            return head;
```

```
        ListNode* Tmp = reverseList(head->next);
        head->next->next = head;  // 将新的链表节点接入链表
        head->next = nullptr;     // 最后一个节点的next指针是nullptr
        return Tmp;  // 返回值应该是反转后链表的头节点
    }
};
```

25.K个一组翻转链表

这是一道困难题，但是并没有什么新颖的算法，只是有很多细碎的边界条件需要处理，核心还是反转链表。这里多了两个部分的逻辑：

- **1.当剩余节点个数不足k个时，及时返回链表**，这部分的逻辑如下：

```
// ListNode *Head = head, *Tail = head;
int Counter = 0;
// 从Head开始向后数k - 1个节点，找到Tail节点
// 之所以是k-1是因为我们是从链表的第一个节点开始数的
while(Counter < k - 1)
{
    Tail = Tail->next;
    Counter++;
    // 如果当前剩余节点数量不足K个，直接返回首节点
    if(not Tail)
        return Dummy->next;
}
```

- 2.**完成局部的K个链表反转之后，将子链表接回原先的链表**，这部分的逻辑如下：

```
auto ReversePair = reverseList(Head, Tail);

// 将反转之后的子链表再次连接到原先的链表上
Previous->next = ReversePair.first;
ReversePair.second->next = Next;
```

在完成上述两部分的逻辑之后，剩下的就是简单的链表反转问题，全部代码如下：

```
class Solution {
    /*反转[Head, Tail]这个范围内的链表*/
    pair<ListNode*, ListNode*> reverseList(ListNode *Head, ListNode* Tail)
    {
        ListNode *Previous = Head, *Present = Head->next;
        while(Previous != Tail)
        {
            ListNode *Tmp = Present->next;
            Present->next = Previous;
```

```
            Previous = Present;
            Present = Tmp;
        }
        return {Tail, Head};    // 反转之后，头尾节点正好反过来
    }
public:
    ListNode* reverseKGroup(ListNode* head, int k) {
        ListNode *Dummy = new ListNode(0, head);

        // Head和Tail记录的分别是k个节点小组中的头尾节点
        ListNode *Head = head, *Tail = head;
        ListNode *Previous = Dummy, *Next = nullptr;
        while(Head)
        {
            int Counter = 0;
            // 从Head开始向后数K - 1个节点，找到Tail节点
            while(Counter < k - 1)
            {
                Tail = Tail->next;
                Counter++;
                // 如果当前剩余节点数量不足K个，直接返回首节点
                if(not Tail)
                    return Dummy->next;
            }
            // 记录Tail之后的下一个节点
            Next = Tail->next;
            auto ReversePair = reverseList(Head, Tail);

            // 将反转之后的子链表再次连接到原先的链表上
            Previous->next = ReversePair.first;
            ReversePair.second->next = Next;

            // 调整指针准备下一次反转
            Previous = ReversePair.second;
            Head = Tail = Next;
        }
        return Dummy->next;
    }
};
```

二叉树

图

滑动窗口 & 双指针

滑动窗口和(双指针)是一种高效解决线性序列问题的算法思想，它可以将线性序列中的一些问题时间复杂度降低到O(n)。

3.无重复字符的最长子串

这道题是一道非常经典的滑动窗口问题，它要求我们在一个字符串中找到最长的不含重复字符的子串(注意子字符串必须是连续的)。

结合本题的数据规模可以知道要求的应该是O(n)复杂度的算法。

在字符串问题中涉及到不重复或者计数问题时往往要用到滑动窗口+哈希表。这道题不同的是，因为涉及到重复问题，只需要集合记录元素即可。

官方题解如下，使用的是unordered_set来记录窗口中出现的字符，左边界每排出一个字符，右边界就不断向前推进探索当前可以到达的最大位置并更新答案：

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        // 哈希集合，记录每个字符是否出现过
        unordered_set<char> occ;
        int n = s.size();
        // 右指针，初始值为 -1，相当于我们在字符串的左边界的左侧，还没有开始移动
        int rk = -1, ans = 0;
        // 枚举左指针的位置，初始值隐性地表示为 -1
        for (int i = 0; i < n; ++i) {
            if (i != 0) {
                // 左指针向右移动一格，移除一个字符
                occ.erase(s[i - 1]);
            }
            while (rk + 1 < n && !occ.count(s[rk + 1])) {
                // 不断地移动右指针探测更长的长度
                occ.insert(s[rk + 1]);
                ++rk;
            }
            // 第 i 到 rk 个字符是一个极长的无重复字符子串
            ans = max(ans, rk - i + 1);
        }
        return ans;
    }
};
```

上面是官方题解的做法，这个做法中存在一个问题，那就是在检测到重复字符时它只是一次将窗口左边界前移一个长度，事实上还可以**有更快的方法**。那就是在哈希表中记录下来每一个字符出现的位置，当有重复字符出现时直接跳转到它的下一个字符，这样可能会有一个问题，那就是中间跳过去的这些字符是否需要删掉？

答案是不需要的，直接通过max运算就可以保证正确性，代码如下：

```
class Solution {
public:
    int lengthOfLongestSubstring(string s)
    {
        unordered_map<char, int> HashTable;
```

```
        int n = s.length();
        int Left = 0, Right = 0;     // [Left, Right]是滑动窗口范围
        int Ans = 0;
        while(Right < n)
        {
    // 发现重复字符，直接跳过中间的所有字符位置，不用一个个向窗口外排出
            if(HashTable.count(s[Right]))
                /*使用max运算符保证跳转位置大于当前左边界*/
                Left = max(Left, HashTable[s[Right]]);
            HashTable[s[Right]] = Right + 1;     // 更新字符出现位置
            Ans = max(Ans, Right - Left + 1);    // 更新最大长度
            ++Right;
        }
        return Ans;
    }
};
```

15.三数之和

这是最经典的三指针问题，其实三指针问题也只是双指针问题的变式。本题的要求是在一个序列中找出所有不重复的相加之和等于0的三元组。

首先简化一下这个问题，如果本题让求的是二元组，这就是最经典的双指针问题。我们只需要先将整个序列进行排序，然后分别用两个指针指向头尾，使它们逐渐向中间靠拢即可不重不漏地搜索到所有符合要求得二元组，代码逻辑如下：

```
// 下面的代码是最原始的双指针算法
// 假设数组为nums，其大小为n
int First = 0, Second = n - 1;
// 和等于0，说明已经找到了满足要求的二元组
if(nums[First] + nums[Second] == 0)
    Ans.push_back({nums[First], nums[Second]});
// 如果当前二元组的和偏小，说明First指针指向的值太小
else if(nums[First] + nums[Second] < 0)
    ++First;
// 如果当前二元组的和偏大，说明Second指针指向的值太大
else
    --Second;
```

这样就将这个问题的复杂度整体降至了O(nlogn)的级别，即排序本身的复杂度，而上面的这段代码复杂度为O(n)。

事实上，双指针算法可以写成下面这样，如下所示：

```
int Second = n - 1;
for(int First = 0 ; First < Second ; ++First)
{
    // 跳过重复的二元组
    if(First > 0 and nums[First] == nums[First - 1])
```

```
            continue;
        while(First < Second and nums[First] + nums[Second] > 0)
            --Second;

        // 检查循环跳出原因
        // 1.如果是因为指针相遇，那么说明进行到了尽头
        if(First == Second)
            break;

        // 2.如果当前找到了一组解，则加入答案
        if(nums[First] + nums[Second] == 0)
            Ans.push_back({nums[First], nums[Second]});
        // 这里还隐藏了一个逻辑，但不用写
        // if(nums[First] + nums[Second] < 0)
            // continue;
    }
```

这和最原始版本相比，对于右指针Second的移动变得更加紧凑，代码效率可能也会更高，"可能"是指从理论上分析代码的整体复杂度是一致的，但实际运行时发现上述写法比原始写法要快一些，可能是代码更加紧凑的原因，也有可能是测试用例的原因。

回到本题，也是同样的思路，只是我们先固定一个指针(First)，移动剩下的两个指针(Second, Third)即可，目标值也从0变成了-nums[First]。这个道理明白了之后，就知道：

所谓三指针问题，甚至四指针问题，也只是先固定其中n-2个指针之后的双指针问题，它们的逻辑和解法本质上都是一样的。

但是本题还有很值得学习的点，那就是**如何写出简洁优雅的三指针代码**，首先给出我的原始版本代码，献一下丑:)。

```
//我的原始版本代码，思路正确，但代码效率还是偏低
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums)
    {
        sort(nums.begin(), nums.end());
        int n = nums.size();
        vector<vector<int>> Ans;
        // 最外层循环，枚举固定指针的位置
        for(int Fixed = 0 ; Fixed < n - 2 ; ++Fixed)
        {
            // 跳过重复的元素值，防止搜到重复的组
            if(Fixed > 0 and nums[Fixed] == nums[Fixed - 1])
                continue;
            // 初始化双指针的位置
            int Left = Fixed + 1, Right = n - 1;
            // 此时的目标值是nums[Fixed]的负值
            int Target = -nums[Fixed];
            while(Left < Right)
            {
```

```
                // 跳过重复的nums[Left]值
                if(Left > Fixed + 1 and nums[Left] == nums[Left - 1])
                {
                    ++Left;
                    continue;
                }
                // 传统的双指针代码，这没什么好说的...
                if(nums[Left] + nums[Right] == Target)
                {
                    Ans.push_back({nums[Fixed], nums[Left], nums[Right]});
                    ++Left;
                    --Right;
                }
                else if(nums[Left] + nums[Right] < Target)
                    ++Left;
                else
                    --Right;
            }
        }
        return Ans;

    }
};
```

官解的代码就使用了双指针第二种写法，效率却明显提升了不少，猜测可能和测试用例有关，但无论如何推荐第二种写法:

通过	108 ms	23.4 MB	C++	2023/02/21 16:40	添加备注
通过	160 ms	23.3 MB	C++	2023/02/21 16:40	添加备注
通过	164 ms	23.4 MB	C++	2023/02/21 16:40	添加备注

```
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        int n = nums.size();
        sort(nums.begin(), nums.end());
        vector<vector<int>> ans;
        // 枚举 a
        for (int first = 0; first < n; ++first) {
            // 需要和上一次枚举的数不相同
            if (first > 0 && nums[first] == nums[first - 1]) {
                continue;
            }
            // c 对应的指针初始指向数组的最右端
            int third = n - 1;
            int target = -nums[first];
            // 枚举 b
```

```
            for (int second = first + 1; second < n; ++second) {
                // 需要和上一次枚举的数不相同
                if (second > first + 1 && nums[second] == nums[second - 1]) {
                    continue;
                }
                // 需要保证 b 的指针在 c 的指针的左侧
                while (second < third && nums[second] + nums[third] > target) {
                    --third;
                }
                // 如果指针重合，随着 b 后续的增加
                // 就不会有满足 a+b+c=0 并且 b<c 的 c 了，可以退出循环
                if (second == third) {
                    break;
                }
                if (nums[second] + nums[third] == target) {
                    ans.push_back({nums[first], nums[second], nums[third]});
                }
            }
        }
        return ans;
    }
};
```

动态规划

53.最大子数组和

这道题的题目是求出一个数组中连续的子数组和，首先看一眼这道题的数据规模是10^5，那么一定是使用O(n)级别的算法来解决。

我的最原始解法使用的是动态规划。动态规划法的递推关系非常简单，设DP[i]是以nums[i]为结尾的最大子数组和，那么：

DP[i] = max(DP[i - 1] + nums[i], nums[i]);

而要求接的答案Ans其实也就是max(DP[i])，在求解DP数组的同时求出即可，非常简单。

```
class Solution {
public:
    int maxSubArray(vector<int>& nums)
    {
        int Ans = nums[0];
        int n = nums.size();
        int DP[n];
        fill(DP, DP + n, 0);
        DP[0] = nums[0];
        for(int i = 1 ; i < n ; ++i)
        {
             DP[i] = max(nums[i], DP[i - 1] + nums[i]); // 递推方程
```

```
            Ans = max(Ans, DP[i]);                      // 更新可能的答案
        }
        return Ans;
    }
};
```

但除此之外，这道题还有一种分治的解法在官方题解中给出了，这本质上也是线段树这种算法的重要功能，使用分治法的代码如下，它对于任意一个区间(l,r)保留了如下的四个信息：

- lsum:区间[l,r]中以l为左端点的最大子段和
- rsum:区间[l,r]中以r为右端点的最大子段和
- msum:区间[l,r]中的最大子段和
- isun:区间[l,r]的区间和

对每一个区间都维持着上述几个信息，那么本题要求的值其实就是msum[0, n - 1]，具体可见官方题解中对上述几个信息的更新和维护方法，这里不再重复，这种方法代码还是非常非常巧妙的。

```
class Solution {
public:
    struct Status {
        int lSum, rSum, mSum, iSum;
    };

    Status pushUp(Status l, Status r) {
        // 注意这里对四项特征的更新策略
        int iSum = l.iSum + r.iSum;
        int lSum = max(l.lSum, l.iSum + r.lSum);
        int rSum = max(r.rSum, r.iSum + l.rSum);
        int mSum = max(max(l.mSum, r.mSum), l.rSum + r.lSum);
        return (Status) {lSum, rSum, mSum, iSum};
    };

    Status get(vector<int> &a, int l, int r) {
        // 递归到区间长度为1时，返回唯一的数值
        if (l == r) {
            return (Status) {a[l], a[l], a[l], a[l]};
        }
        // 从区间中点进行分治
        int m = (l + r) >> 1;

        // 左右分别递归下去，得到子区间的状态信息
        Status lSub = get(a, l, m);
        Status rSub = get(a, m + 1, r);

        // 汇总并向上返回
        return pushUp(lSub, rSub);
    }

    int maxSubArray(vector<int>& nums) {
        return get(nums, 0, nums.size() - 1).mSum;
```

```
        }
};
```

正如题解中所说的那样，这就是线段树方法的雏形：

题外话

「方法二」相较于「方法一」来说，时间复杂度相同，但是因为使用了递归，并且维护了四个信息的结构体，运行的时间略长，空间复杂度也不如方法一优秀，而且难以理解。那么这种方法存在的意义是什么呢？

对于这道题而言，确实是如此的。但是仔细观察「方法二」，它不仅可以解决区间 $[0, n-1]$，还可以用于解决任意的子区间 $[l, r]$ 的问题。如果我们把 $[0, n-1]$ 分治下去出现的所有子区间的信息都用堆式存储的方式记忆化下来，即建成一棵真正的树之后，我们就可以在 $O(\log n)$ 的时间内求到任意区间内的答案，我们甚至可以修改序列中的值，做一些简单的维护，之后仍然可以在 $O(\log n)$ 的时间内求到任意区间内的答案，对于大规模查询的情况下，这种方法的优势便体现了出来。这棵树就是上文提及的一种神奇的数据结构——线段树。

模拟

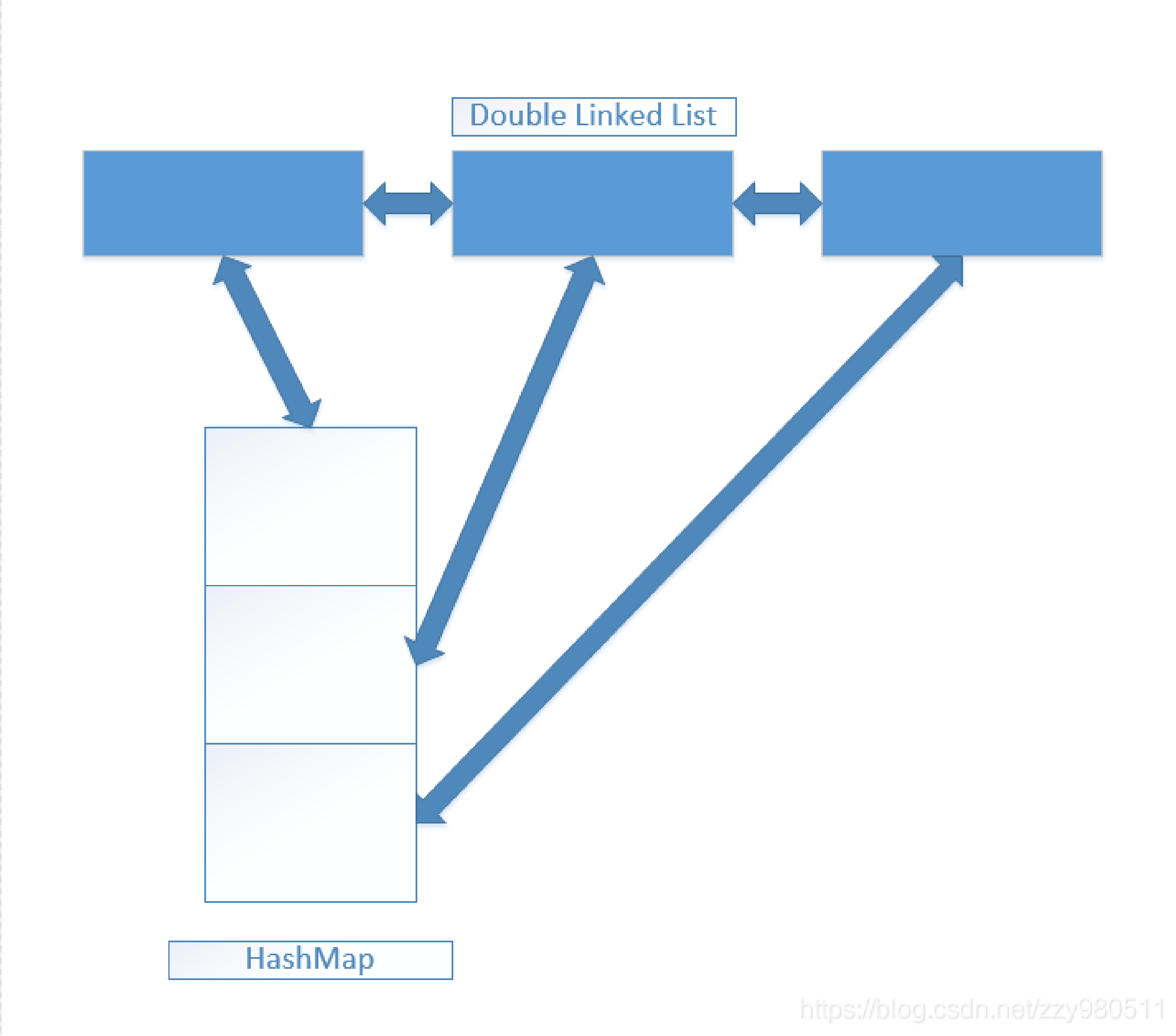
单调栈

设计类问题

146.LRU 缓存

本题是设计类问题的代表，题目中让实现一个含LRU功能的cache，且插入和删除元素的复杂度都是O(1)。这道设计题的核心思想在于链表和哈希表相互索引。

链表节点中存放着完整的key-value对，哈希表存放着<key, ListNode*>，哈希表可以通过ListNode*快速索引到链表，而链表也可以通过key快速索引到哈希表。



具体实现代码如下:

```
// 定义链表节点数据结构，包含完整的键值对，前后向的指针
struct Node
{
    int Key, Value;
    Node* Prev;
    Node* Next;

    // ctor defined here
    Node() : Key(0), Value(0), Prev(nullptr), Next(nullptr){}
    Node(int K, int V) : Key(K), Value(V), Prev(nullptr), Next(nullptr){}
};

class LRUCache {
    // cache容量
    int Capacity;

    // 当前的cache size，即存放的键值对数量
    int CurrentSize;
```

```
    // 双向链表的头尾指针，这其实是两个Dummy节点
    Node* Head;
    Node* Tail;

    // 哈希表
    map<int, Node*> HashMap;

public:
    LRUCache(int capacity) : Capacity(capacity), CurrentSize(0)
    {
        // 初始化头尾链表节点
        Head = new Node();
        Tail = new Node();

        // 修改指针
        Head->Next = Tail;
        Tail->Prev = Head;
    }

    // 向链表头部插入一个节点
    void insertNode(Node* NewNode)
    {
        // 修改指针以在头部插入一个节点
        NewNode->Next = Head->Next;
        NewNode->Prev = Head->Next->Prev;
        Head->Next->Prev = NewNode;
        Head->Next = NewNode;
        ++CurrentSize;      // 增加节点计数
    }

    // 移除指定的链表节点
    void deleteNode(Node* DelNode)
    {
        DelNode->Next->Prev = DelNode->Prev;
        DelNode->Prev->Next = DelNode->Next;
        --CurrentSize;
    }

    // 将节点移至链表的头部
    // 即删除某个节点并插入到链表头部的组合
    void moveToHead(Node* Temp)
    {
        // delete the element and insert it to head of list
        deleteNode(Temp);
        insertNode(Temp);
    }

    // 删除链表尾部的节点，即满足LRU的条件删去最近最久未使用
    Node* removeTail()
    {
        Node* Res = Tail->Prev;
        deleteNode(Tail->Prev);
        return Res;
```

```cpp
    }

    int get(int key)
    {
        // 根据key去索引对应的hash表
        // 如果找到了对应的键值，则直接索引到链表节点得到值
        if(HashMap.find(key) != HashMap.end())
        {
            int Res = HashMap[key]->Value;

            // 将刚刚访问过的节点转移到链表头部
            moveToHead(HashMap[key]);
            return Res;
        }
        // 未找到则返回-1
        return -1;
    }

    void put(int key, int value)
    {
        // 如果当前键值已经存在，那么更新对应的值并将节点移动至头部
        if(HashMap.find(key) != HashMap.end())
        {
            HashMap[key]->Value = value;
            moveToHead(HashMap[key]);
        }

        // 如果键值不存在且Cache未满，则插入对应的键值对和节点
        else if(HashMap.find(key) == HashMap.end() && CurrentSize < Capacity)
        {
            Node* NewNode = new Node(key, value);
            HashMap.insert(make_pair(key, NewNode));
            insertNode(NewNode);
        }

        // 如果键值不存在且Cache已满，则使用LRU策略换出最后的节点
        // 并插入新的节点
        else if(HashMap.find(key) == HashMap.end() && CurrentSize == Capacity)
        {
            // remove the LRU element and insert the new element
            Node* Temp = removeTail();
            HashMap.erase(Temp->Key);
            delete Temp;                                // prevent memory leak

            Node* NewNode = new Node(key, value);
            HashMap.insert(make_pair(key, NewNode));
            insertNode(NewNode);
        }
    }
};
```

数学类问题

模拟 & 找规律