LeetCode Summary

贪心

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
*/
    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
    }
};
```

二叉树

图

滑动窗口

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小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;
    }
};
```

动态规划

模拟

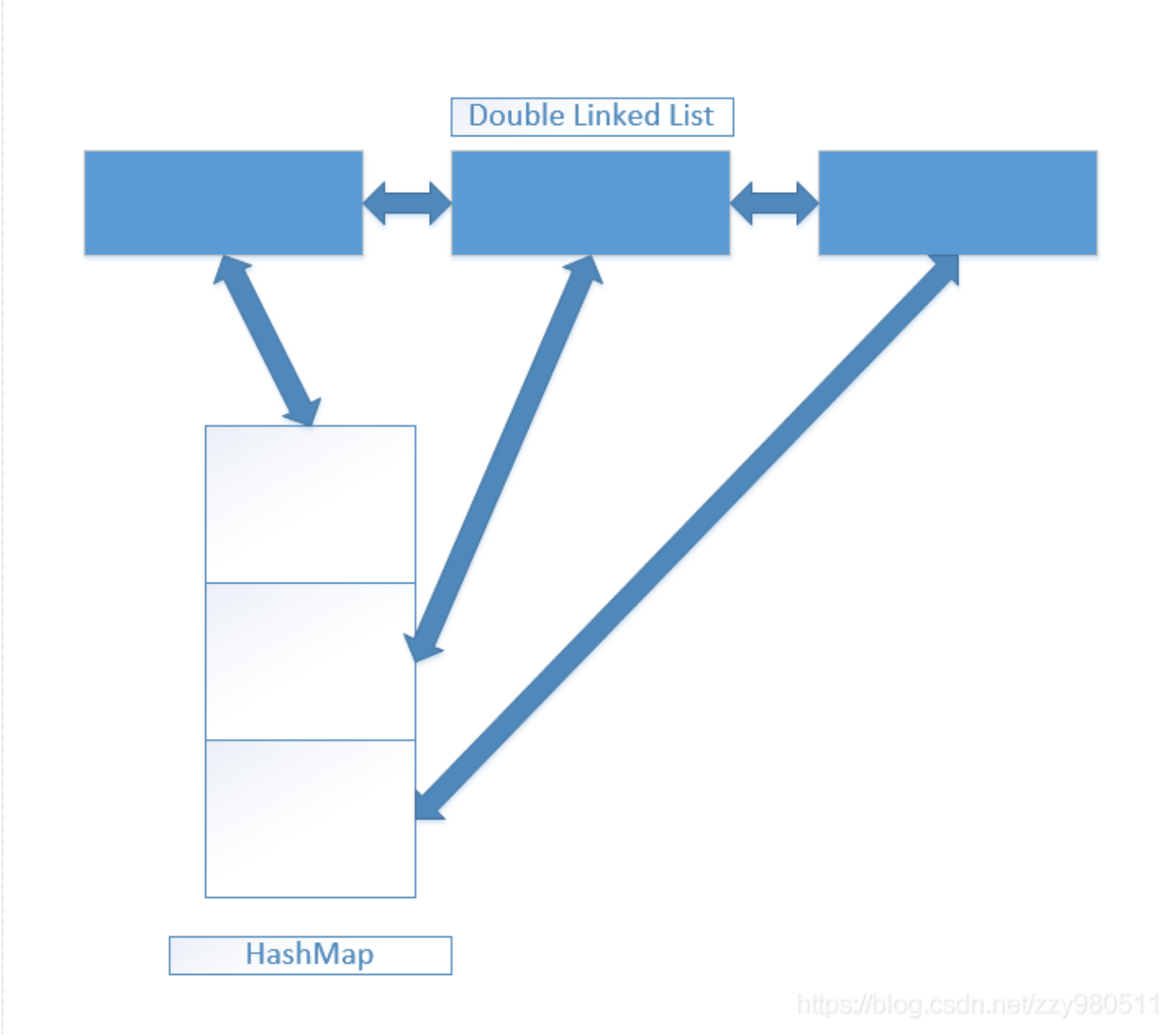
单调栈

设计类问题

146.LRU 缓存

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;
    }

    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);
        }
```

```
        }
    };
```