算法讨论班第 30 期

——王海璐

2016 年 5 月 5 日星期四

116. Populating Next Right Pointers in Each Node

Given a binary tree

```
struct TreeLinkNode {
  TreeLinkNode *left;
  TreeLinkNode *right;
  TreeLinkNode *next;
}
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to `NULL`.

Initially, all next pointers are set to `NULL`.

Note:

- You may only use constant extra space.
- You may assume that it is a perfect binary tree (ie, all leaves are at the same level, and every parent has two children).

For example,

Given the following perfect binary tree,

```
     1
   /  \
  2    3
 / \  / \
4  5  6  7
```

After calling your function, the tree should look like:

```
        1 -> NULL
      /  \
     2 -> 3 -> NULL
    / \  / \
   4->5->6->7 -> NULL
```

117. Populating Next Right Pointers in Each Node II

Follow up for problem "*Populating Next Right Pointers in Each Node*".

What if the given tree could be any binary tree? Would your previous solution still work?

Note:

- You may only use constant extra space.

For example,

Given the following binary tree,

```
        1
      /  \
     2    3
    / \    \
   4   5    7
```

After calling your function, the tree should look like:

```
        1 -> NULL
      /  \
     2 -> 3 -> NULL
    / \    \
```

```
4-> 5 -> 7 -> NULL
```

分析：两个指针 p、q，来控制内外两层循环，内循环是每一层去处理下一层的连接操作；外循环用来更新层，q 是每一层第一个节点指针。循环的过程中利用已经连接好的 **next 指针域**。

对于**完全二叉树，**每一层的 q 更新是 q=q->left 即可。

对于**任意的二叉树，**每一层内循环连接下一层，并记录下一层的第一个节点指针。具体：设置三个指针：nextNewHead（下一层的第一个节点）、LastHead（本层的第一个节点）、temp（链接的临时变量）。

内循环的开始，将 nextNewHead 赋值给 LastHead，将 nextNewHead 和 temp 置为 NULL，循环过程中用 temp 去链接，只有 nextNewHead 为 NULL 的时候给其赋值，表示下一层的开始节点。外循环判断当 nextNewHead 为空的时候就结束。

116、117 代码分别如下：

```
 9  class Solution {
10  public:
11        void connect(TreeLinkNode *root) {
12          if (root == NULL || root->left == NULL)
13          {
14              return;
15          }
16          TreeLinkNode* p;
17          TreeLinkNode* q = root;
18          while (q->left != NULL)
19          {
20              p = q;
21              while (p != NULL)
22              {
23                  p->left->next = p->right;
24                  if (p->next != NULL)
25                  {
26                      p->right->next = p->next->left;
27                  }
28                      p = p->next;
29              }
30              q = q->left;
31          }
32      }
33  };
```

```
class Solution {
public:
 void connect(TreeLinkNode *root) {
        if (root == NULL )  {
            return;
        }
        TreeLinkNode* nextNewHead=root;//下一层的第一个节点
        TreeLinkNode* LastHead = root;//本层的第一个节点
        TreeLinkNode *temp = NULL;//链接的临时变量
        while (nextNewHead!= NULL){
            LastHead = nextNewHead;
            nextNewHead = temp = NULL;
            while (LastHead != NULL){
                if (LastHead->left != NULL) {
                    if (temp!=NULL){
                        temp->next = LastHead->left;
                    }
                    temp = LastHead->left;
                    if (nextNewHead==NULL){
                        nextNewHead = temp;
                    }

                }
                if (LastHead->right != NULL {
                    if (temp != NULL){
                        temp->next = LastHead->right;
                    }
                    temp = LastHead->right;
                    if (nextNewHead == NULL){
                        nextNewHead = temp;
                    }
                }
                LastHead = LastHead->next;
            }
        }
    }
};
```

124. Binary Tree Maximum Path Sum

Given a binary tree, find the maximum path sum.

For this problem, a path is defined as any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The path does not need to go through the root.

For example:

Given the below binary tree,

```
      1
     / \
    2   3
```

Return `6`.

分析：**递归**，从**下往上**计算以每个节点为根的最大和，更新最大和的值。

为了计算一个节点父节点为根的最大和，需要记录经过其父节点的最大部分和，递归函数的返回值， max of (root->val, root->val + lmax, root->val + rmax)

总的最大和是max of {root->val,root->val+lmax,root.val->rmax, root->val + lmax + rmax}。
代码如下：

```
10  class Solution {
11  public:
12      int maxSum(TreeNode*root, int &m){
13          if (root==NULL) {
14              return 0;
15          }
16          int lmax = 0;
17          int rmax = 0;
18          int value = root->val;
19          if (root->left) {
20              lmax = maxSum(root->left, m);
21              if (lmax>0) {
22                  value += lmax;
23              }
24          }
25          if (root->right) {
26              rmax = maxSum(root->right, m);
27              if (rmax>0){
28                  value += rmax;
29              }
30          }
31          //更新最大值
32          //max is the max of {root->val,root->val+lmax,root.val->rmax, root->val + lmax + rmax}
33          if (value>m){
34              m = value;
35          }
36          //返回值
37          //return max of (root->val, root->val + lmax, root->val + rmax)
38          return max(root->val, max(root->val + lmax, root->val + rmax));
39      }
40      int maxPathSum(TreeNode* root) {
41          if (root==NULL){
42              return 0;
43          }
44          int m = INT_MIN;
45          int x = maxSum(root,m);
46          return m;
47      }
48  };
```

23. Merge k Sorted Lists

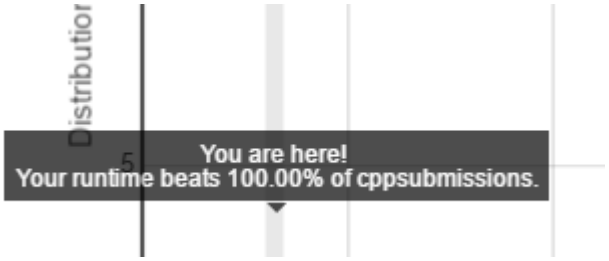
Merge *k* sorted linked lists and return it as one sorted list. Analyze and describe its complexity.

Subscribe to see which companies asked this question

分析：n 表示总共有 n 个链表，m 表示链表的长度。
最简单想到的 O(n*m)方法，超时。
然后想到用归并排序的思想，利用递归，时间复杂度 O(m*log n)。
最后利用希尔排序的思想，每次设置 len/2 的间隔，进行合并，最后得到很好的时间复杂度。
代码如下：

```
class Solution {
public:
 ListNode*merge2Lists(ListNode*list1, ListNode*list2)
    {
        ListNode *head = new ListNode(-1);
        ListNode *cur = head;
        while (list1 != NULL && list2 != NULL){
            if (list1->val<list2->val)  {
                cur->next = list1;
                list1 = list1->next;
            }else{
                cur->next = list2;
                list2 = list2->next;
            }
            cur = cur->next;
        }
        if (list1){
            cur->next = list1;
        }else{
            cur->next = list2;
        }
        return head->next;
    }
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        if (lists.empty() || lists.size() == 0) {
            return NULL;
        }
        if (lists.size() == 1){
            return lists[0];
        }
        int len = lists.size();
        while (len > 1) {
            int k = (len + 1) / 2;
            for (int i = 0; i < len / 2; i++)
                lists[i] = merge2Lists(lists[i], lists[i + k]);
            len = k;
        }
        return lists[0];

    }
};
```



72. Edit Distance

Given two words *word1* and *word2*, find the minimum number of steps required to convert *word1* to *word2*. (each operation is counted as 1 step.)

You have the following 3 operations permitted on a word:

a) Insert a character

b) Delete a character

c) Replace a character

分析：动态规划的思路。
dp[i][j]表示word1的从0到i-1位(前i位)的字符和word2的从0到j-1位（前j位）的字符最短编辑距离。
当word1[i]==word2[j]时，dp[i + 1][j + 1]= dp[i][j]；
否则递推式：dp[i + 1][j + 1] = min(dp[i][j + 1] + 1, min(dp[i + 1][j] + 1, dp[i][j] + 1));
dp[i][j + 1] + 1表示的情况是word1删除字符word1[i]，或者word2插入word1[i]；
dp[i + 1][j] + 1表示的情况是word2删除字符word2[j]，或者word1插入word2[j]；
dp[i][j] + 1表示的是替换。
初始化的条件：
代码如下：**dp[0][i]=i，dp[i][0]=i .表示一个字符串为空串的时候，需要的距离**

```
class Solution {
public:
        int minDistance(string word1, string word2) {
        if (word1.size()==0)
        {
            return word2.size();
        }
        if (word2.size()==0)
        {
            return word1.size();
        }
        int len1 = word1.size();
        int len2 = word2.size();
        //dp[i][j]表示word1的从0到i-1位(前i位)的字符和word2的从0到j-1位（前j位）的字符最短编辑距离
        vector<vector<int>>dp(len1+1, vector<int>(len2+1, 0));
        for (int i = 0; i < len1+1; i++){
            dp[i][0] = i;
        }
        for (int j = 0; j < len2+1; j++)    {
            dp[0][j] = j;
        }
        for (int i = 0; i < len1; i++)  {
            for (int j = 0; j < len2; j++){
                if (word1[i] == word2[j]){
                    dp[i + 1][j + 1] = dp[i][j];
                }
                else    {
                    dp[i + 1][j + 1] = min(dp[i][j + 1] + 1, min(dp[i + 1][j] + 1, dp[i][j] + 1));
                }

            }
        }
        return dp[len1][len2];
    }
};

```

97. Interleaving String

Given *s1*, *s2*, *s3*, find whether *s3* is formed by the interleaving of *s1* and *s2*.

For example,、Given:

s1 = `"aabcc"`,

s2 = `"dbbca"`,

When *s3* = `"aadbbcbcac"`, return true.

When *s3* = `"aadbbbaccc"`, return false.

分析：二维动态规划的思想，dp[][]是一个bool二维数组。
dp[i+1][j+1]:表示s1[0...i]与s2[0...j]能否交替形成s3[0...i+j+1]部分.
状态转移方程:
dp[i+1][j+1] = (dp[i][j+1] && s1[i] == s3[i+j+1]) || (dp[i+1][j] && s2[j] == s3[i+j+1]);
初始化条件是：dp[i + 1][0] = （dp[i][0] && s1[i] == s3[i]）;
dp[0][i + 1] = （dp[0][i] && s2[i] == s3[i]）;
例子可以推断的矩阵为：

			d	b	b	c	a
		0	1	2	3	4	5
	0	1	0	0	0	0	0
a	1	1	0	0	0	0	0
a	2	1	1	1	1	1	0
b	3	0	1	1	0	1	0
c	4	0	0	1	1	1	1
c	5	0	0	0	1	0	1

代码如下：

```cpp
class Solution {
public:
    bool isInterleave(string s1, string s2, string s3)
    {
        int m = s1.size(), n = s2.size();
        if (m + n != s3.size())
            return false;
        bool dp[m + 1][n + 1];
        dp[0][0] = true;
        //初始化边界
        for (int i = 0; i < m; i++)
            dp[i + 1][0] = dp[i][0] && s1[i] == s3[i];
        for (int i = 0; i < n; i++)
            dp[0][i + 1] = dp[0][i] && s2[i] == s3[i];
        for (int i = 0; i < m; i++){
            for (int j = 0; j < n; j++)
            {
                dp[i + 1][j + 1] = (dp[i][j + 1] && s1[i] == s3[i + j + 1]) || (dp[i + 1][j] && s2[j] == s3[i + j + 1]);
            }
        }
        return dp[m][n];
    }
};
```

下期主讲人：郭清沛，题目：稍后