算法笔记

邓岩*

2018 年 11 月 13 日

*837123564@qq.com

目录

1 第一章

1.1 No.1

```
// $x_a$
```

2 动态规划

2.1 切钢条

```
//
// Created by 邓岩 on 2018/11/3.
//

# include <stdio.h>
# include <stdlib.h>
# include <string.h>

//动态规划切钢条

void bottom_to_up(int n) //自底向上法
{
    int rr[11] = {0, 1, 5, 8, 9, 10, 17, 17, 20, 24, 30};
    int * r = (int *)calloc(n + 1, sizeof(int));
    memcpy(r, rr, sizeof(int) * 11);
    int * s = (int *)calloc(n + 1, sizeof(int));

    for (int i = 1; i < n+1; ++i) {
        for (int j = 0; j <= i; ++j) {
            if (r[i] < r[j] + r[i - j]) {
                r[i] = r[j] + r[i - j];
                s[i] = j;
            }
        }
    }
    printf("  长度: ");
    for (int k = 0; k < n + 1; ++k) {
        printf("%3d ", k);
    }
    printf("\n  价格: ");
    for (int k = 0; k < n + 1; ++k) {
        printf("%3d ", r[k]);
    }
    printf("\n  切割: ");
    for (int k = 0; k < n + 1; ++k) {
        printf("%3d ", s[k]);
    }
}

int up_to_bottom(int n, int * rr, int * r) //带备忘的自顶向下法
{
    if (r[n] > 0)
        return r[n];
    for (int i = 1; i < n + 1; ++i) {
        if (r[n] < rr[i] + up_to_bottom(n - i, rr, r)) {
            r[n] = rr[i] + up_to_bottom(n - i, rr, r);
        }
```

```
48      }
49      return r[n];
50  }
51
52  int main(void)
53  {
54      int n ;
55      printf(" 请输入钢条长度: ");
56      scanf("%d", &n);
57      int t[11] = {0, 1, 5, 8, 9, 10, 17, 17, 20, 24, 30};
58      int * rr = (int *)calloc(n + 1, sizeof(int));
59      memcpy(rr, t, sizeof(int) * 11);
60      int * r = (int *)calloc(n + 1, sizeof(int));
61      printf("%d", up_to_bottom(n, rr, r));
62  }
```

2.2 矩阵链乘法

```
1   //
2   // Created by 邓岩 on 2018/11/3.
3   //
4
5   # include <stdio.h>
6   # include <stdlib.h>
7   # include <string.h>
8
9   //动态规划处理矩阵链乘法
10
11
12  void print_optimal_parens(int (*s)[7], int i, int j) //这里的 7 为二维数组 s 的列数
13  {
14      if (i == j)
15          printf("A%d", i);
16      else {
17          printf("(");
18          print_optimal_parens(s, i, s[i][j]);
19          print_optimal_parens(s, s[i][j] + 1, j);
20          printf(")");
21      }
22  }
23
24  void matrix_chain_order(const int * p, int n)
25  {
26      int j;
27      int (*m)[n + 1] = calloc((n + 1) * (n + 1), sizeof(int));
28      int (*s)[n + 1] = calloc((n + 1) * (n + 1), sizeof(int));
29      for (int l = 2; l <= n; ++l) {
30          for (int i = 1; i <= n - l + 1; ++i) {
31              j = i + l - 1;
32              m[i][j] = INT32_MAX;
33              for (int k = i; k < j; ++k) {
34                  int q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
35                  if (q < m[i][j]) {
36                      m[i][j] = q;
37                      s[i][j] = k;
38                  }
```

```
            }
        }
    }
    for (int i1 = 0; i1 < n + 1; ++i1) {
        for (int i = 0; i < n + 1; ++i) {
            printf("%5d ", m[i1][i]);
        }
        for (int i = 0; i < n + 1; ++i) {
            printf("%5d ", s[i1][i]);
        }
        printf("\n");
    }
    print_optimal_parens(s, 1, n);
    free(m);
    free(s);
}

int main(void)
{
    int p[] = {30, 35, 15, 5, 10, 20, 25};
    //有六个矩阵，p[0] 存放着第一个矩阵的行，其余的 p[i] 表示第 i 个矩阵的列数
    matrix_chain_order(p, 6);
}
```

2.3 最长公共子序列

```
//
// Created by 邓岩 on 2018/11/3.
//

# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# include <unistd.h>
# include <sys/stat.h>
# include <fcntl.h>

void LCS_PRINT(char ** t, char * y, int i, int j)
{
    int n = strlen(y);
    char (*b)[n + 1] = t;
    if (i == 0 || j == 0)
        return;
    if (b[i][j] == '\\') {
        LCS_PRINT(t, y, i - 1, j - 1);
        printf("%c", y[j - 1]);
    } else if (b[i][j] == '|') {
        LCS_PRINT(t, y, i - 1, j);
    } else {
        LCS_PRINT(t, y, i, j - 1);
    }
}

void LCS_LENGTH(char * x, char * y)
{
    int len1 = strlen(x);
```

```
    int len2 = strlen(y);
    char (*b)[len2 + 1] = calloc((len1 + 1) * (len2 + 1), sizeof(char));
    int (*c)[len2 + 1] = calloc((len1 + 1) * (len2 + 1), sizeof(int));
    for (int i = 1; i <= len1; ++i) {
        for (int j = 1; j <= len2; ++j) {
            if (*(x + i - 1) == *(y + j - 1)) {
                c[i][j] = c[i - 1][j - 1] + 1;
                b[i][j] = '\\';
            } else if (c[i - 1][j] >= c[i][j - 1]) {
            //c[i][j], c[i - 1][j], c[i][j - 1] 是相邻的，也就是说箭头指向有较大最大 LCS 的方向
                c[i][j] = c[i - 1][j];
                b[i][j] = '|';
            } else {
                c[i][j] = c[i][j - 1];
                b[i][j] = '-';
            }
        }
    /*for (int i = 0; i <= len1; ++i) {
        for (int j = 0; j <= len2; ++j) {
            printf("%d ", c[i][j]);
        }
        for (int j = 0; j <= len2; ++j) {
            printf("%c ", b[i][j]);
        }
        printf("\n");
    }*/
    LCS_PRINT(b, y, len1, len2);
}

int main(void)
{
    char * file1 = "/Users/dengyan/exam.c";
    char * file2 = "/Users/dengyan/exam";
    struct stat stat1;
    stat(file1, &stat1);
    char * x = (char *)malloc(stat1.st_size + 1);
    int fdx = open(file1, O_RDONLY);
    read(fdx, x, stat1.st_size);
    x[stat1.st_size] = 0;
    stat(file2, &stat1);
    char * y = (char *)malloc(stat1.st_size + 1);
    int fdy = open(file2, O_RDONLY);
    read(fdy, y, stat1.st_size);
    y[stat1.st_size] = 0;

    //char x[100] = "ABCBDAB";
    //char y[100] = "BDCABA";
    LCS_LENGTH(x, y);
}
```

3 树

3.1 红黑树

```
1  //
2  // Created by 邓岩 on 2018/10/29.
3  //
4
5  # include <stdio.h>
6  # include <stdlib.h>
7
8  //红黑树
9  //每个节点包含 5 个属性，颜色，权值，左子树指针，右子树指针
10 //一颗红黑树是满足下列红黑形式的二叉搜索树
11 //1. 每个节点或者是红色的，或者是黑色的
12 //2. 根节点是黑色的
13 //3. 每个叶节点是黑色的，这里说的叶节点是代表每个真正的叶节点还有两个子节点，那两个子节点也是黑色的，
14     //比如一个只有一个根节点的树，它为黑色，但是视作它还有两个子节点，也均为黑色
15 //4. 如果一个节点是红色的，则它的两个叶节点都是黑色的
16 //5. 对于每个节点，从该节点到其所有后代叶节点的简单路径上，均包含个数相同的黑色节点
17
18 //做左旋转时，x 的父节点域和其右节点域会改变，右孩子的父节点域和右孩子的左孩子的父节点域会改变
19 //旋转操作在任何情况都能正常进行
20 //由于每次增加的都是红节点，且旋转操作绕两红点进行，
21     //性质上就像将一个红节点移动到了另一边，所以旋转操作不会破坏性质 5
22 //此代码对于被删除的节点没有释放空间，而且管理节点的 T 的左孩子域没有指向树的最小节点，
23     //树中的最大节点的右孩子域没有指向管理节点，后面进行更改
24
25 #define RED 1
26 #define BLACK 0
27
28 typedef struct node {
29     struct node * left;
30     struct node * right;
31     struct node * parent;
32     int value;
33     int color;
34 }Node, *pNode;
35
36 struct node * nil;
37
38 pNode tree_minimum(pNode x) //找到根节点为 x 的最小节点
39 {
40     if (x == NULL)
41         return NULL;
42
43     while (x->left != nil)
44         x = x->left;
45     return x;
46 }
47
48 pNode rb_find(pNode T, int value) //找到根节点为 x 且权值为 value 的节点
49 {
50     T = T->parent; //根节点
51     while (T != nil && T->value != value)
52         if (T->value < value)
53             T = T->right;
```

```
54          else
55              T = T->left;
56
57      return T;
58  }
59
60  pNode tree_successor(pNode x)
61  {
62      if (x == NULL)
63          return NULL;
64
65      if (x->right) //如果有右孩子
66          return tree_minimum(x->right);
67
68      //以下会有两种情况，既然该节点没有右孩子
69      //那么如果该节点是其父节点的左孩子，这样的话，其父节点就是后继
70      //或者它是父节点的右孩子，那么存在一个它最近祖先 x，该 x 是另一个节点 y 的左孩子，那 y 就是它的后继
71
72      while (x->parent && x==x->parent->right) //当结束时，x 必然是某节点的左子树，那么只需要找它的父亲即可
73          x = x->parent;
74
75      return x->parent;
76  }
77
78  void left_rotate(pNode x)
79  {
80      pNode y = x->right;
81
82      x->right = y->left;
83      if (y->left != nil)
84          y->left->parent = x;
85
86      y->parent = x->parent;
87      if (x->parent->parent == x) //如果 x 是根节点
88          x->parent->parent = y;
89      else if (x->parent->left == x)
90          x->parent->left = y;
91      else
92          x->parent->right = y;
93
94      y->left = x;
95      x->parent = y;
96  }
97
98  void right_rotate(pNode x)
99  {
100     pNode y = x->left;
101
102     x->left = y->right;
103     if (y->right != nil)
104         y->right->parent = x;
105
106     y->parent = x->parent;
107     if (x->parent->parent == x)
108         x->parent->parent = y;
109     else if (x->parent->left == x)
```

```
110         x->parent->left = y;
111     else
112         x->parent->right = y;
113 
114     y->right = x;
115     x->parent = y;
116 }
117 
118 void rb_insert_fixup(pNode T, pNode x) // T 是一个管理节点，T->parent 指向根节点
119 {
120     pNode y;
121 
122     while (x->parent->color == RED) //如果 x 的父亲是红节点，此时 x 的祖节点必为黑色 (不然在插入前就不满足条件)
123     {
124         if (x->parent->parent->left == x->parent) { //如果 x 的父亲是左孩子
125             y = x->parent->parent->right; //y 是其叔节点
126             if (y != nil && y->color == RED) { //此状态下可能会破坏性质 4
127                 x->parent->color = BLACK; //父节点变为黑
128                 y->color = BLACK; //叔节点变为黑
129                 x->parent->parent->color = RED; //祖节点变为红，如果其曾祖节点为红色，那么性质 4 被破坏
130                 x = x->parent->parent; //进行下一轮循环，查看其祖节点和曾祖节点是够满足要求
131             } else if (x == x->parent->right) { //此时其叔节点要么为空 (空可以看作黑) 要么为黑，如果 x 为内侧插入
132                 x = x->parent;
133                 left_rotate(x);
134             } else {//外侧插入
135                 x->parent->color = BLACK;
136                 x->parent->parent->color = RED;
137                 right_rotate(x->parent->parent);
138             }
139         } else { //如果 x 的父亲是右孩子
140             y = x->parent->parent->left; //y 是其叔节点
141             if (y != nil && y->color == RED) {
142                 x->parent->color = BLACK; //父节点变为黑
143                 y->color = BLACK; //叔节点变为黑
144                 x->parent->parent->color = RED; //祖节点变为红
145                 x = x->parent->parent; //查看其祖节点和曾祖节点是够满足要求
146             } else if (x == x->parent->left) { //此时其叔节点要么为空 (空可以看作黑),
147                 //要么为黑，如果 x 为内侧插入
148                 x = x->parent;
149                 right_rotate(x);
150             } else {
151                 x->parent->color = BLACK;
152                 x->parent->parent->color = RED;
153                 left_rotate(x->parent->parent);
154             }
155         }
156         T->parent->color = BLACK; //这部不能掉，不然当根节点变成红色后会引发不必要的操作,
157             //根节点由红变黑不影响任何性质
158     }
159 }
160 
161 void rb_insert(pNode T, int val)
162 {
163     pNode x, y, z;
164 
165     y = nil;
```

```
166     x = T->parent; //根节点
167     z = (pNode)malloc(sizeof(Node));
168     z->color = RED;
169     z->value = val;
170     z->left = nil;
171     z->right = nil;
172
173     while (x != nil) {
174         y = x;
175         if (x->value < val)
176             x = x->right;
177         else
178             x = x->left;
179     }
180
181     z->parent = y;
182
183     if (y == nil)
184     {
185         z->color = BLACK;
186         T->parent = z;
187         z->parent = T;
188         return;
189     }
190     else if (y->value < z->value)
191         y->right = z;
192     else
193         y->left = z;
194     rb_insert_fixup(T, z);
195 }
196
197 void rb_transplant(pNode u, pNode v)
198 {
199     if (u->parent->parent == u) { //u 为根节点
200         u->parent->parent = v;
201     } else if (u->parent->left == u) {
202         u->parent->left = v;
203     } else {
204         u->parent->right = v;
205     }
206     v->parent = u->parent;
207 }
208
209 /*
210  * 运行此函数时，就说明了 x 的前继 y 是一个黑节点
211  * x 如果为黑，那么 x 点必为 nil 点，
212      (因为 y 为黑，并且 y 没有左孩子，那么其不可能有一个非 nil 的黑右孩子)，否则一开始便破坏了性质 5
213  * 如果 x 为红，将其直接变为黑即可
214  */
215
216 void rb_delete_fixup(pNode T, pNode x)
217 {
218     /*
219      * 第一次进入循环时 x 必为 nil，此时需要根据 x 的兄弟节点来判别情况
220      * 如果 x 的兄弟是红色，那么其兄弟必然有两个非 nil 的左右黑孩子，否则一开始便不符合情况 5
221      * 如果其兄弟为黑，那么其兄弟要么没有孩子，要么有两个红孩子，要么有一个红孩子
```

```
222         */
223
224         pNode w;
225         while (x != T->parent && x->color == BLACK) {
226             if (x == x->parent->left) { //如果 x 是左孩子
227                 w = x->parent->right; //w 是 x 的兄弟节点
228     //case1:
229                 if (w->color == RED) {
230
231                     /*
232                      * 如果 w 是红色，此时 w 和 x 的父亲就是黑色，而且 w 必有两个非 nil 黑孩子，否则破坏性质 5
233                      * 这两个黑孩子可能有红孩子，如果左孩子没有红孩子，会进行下面的 case2 操作
234                      * 将 w 和其父亲反色，然后左旋转
235                      * 此时的状态可以看作 w 是根节点为黑色，w 有一个左孩子，是 w 和 x 此前的父亲 (红色)，
236                      w 有一个右孩子，就是旋转前的右孩子 (黑色，可能有数量不定的红孩子)
237                      * w 的左孩子此时也有两个孩子，左边的是 x(nil)，
238                          右边的是 w 旋转前的左孩子 (黑色，可能有数量不定的红孩子)
239                      */
240
241                     w->parent->color = RED; //旋转点及其右孩子变色
242                     w->color = BLACK;
243                     left_rotate(w->parent); //然后左转，此时 w 的左孩子 (黑色) 成为了 x 父亲的右孩子，
244                         //w 成为了 x 的爷爷
245                     w = x->parent->right; //重新将 w 设置为 x 的兄弟
246                 }
247     //case2:
248                 if (w->left->color == BLACK && w->right->color == BLACK) {
249
250                     /*
251                      * 如果 x 的兄弟左右孩子的颜色都为黑，那么这种情况只可能是由 case1 的旋转操作得来的
252                      * 即 case1 的 w 的左孩子 (非 nil 黑色) 没有红色孩子，
253                          因为不存在互为兄弟的黑色节点都没有红孩子的情况
254                      */
255
256                     w->color = RED; //如果 w 的左孩子有红孩子的话这步会直接导致性质 4 被破坏
257                     x = w->parent; //此时 x 是红色，循环退出后将其变为黑色即可
258                 } else if (w->right->color == BLACK) {
259     //case3:
260                     /*
261                      * 如果 w 的右子树是黑色，那么 w 的右孩子就是 nil。因为 w 本身是黑色，
262                          其不可能有黑色孩子 (否则破坏性质 5)
263                      * 由于 w 为黑色时，其必有一个红孩子，因为不存在互为兄弟的黑色节点都没有红孩子的情况
264                      * w 此时是右子树，它的红孩子是它的左节点，处于内侧，需要通过旋转将红节点移动到外侧
265                      * 将 w 和 w 的左孩子反色后右旋转
266                      * 此时的状态可以看作根节点是 x 的父亲，
267                          颜色未知 (因为旋转前 w 和 x 都为黑色，其父亲什么颜色没有影响)
268                      * 根节点的左孩子是 x(nil)，右孩子是旋转前 w 的左孩子 (黑色，本来是红色，反色后旋转)，
269                          根节点的右孩子的右孩子 (红色，本来是黑色，就是旋转前的 w)
270                      */
271
272                     w->left->color = BLACK; //w 的左孩子原本为红色
273                     w->color = RED; //将 w 由黑色变为红色
274                     right_rotate(w); //将 w 点和其左孩子反色然后右旋转，形状上就像将左孩子移动到了右孩子的位置
275                     w = x->parent->right; //将 w 重新设置为 x 的兄弟，也可以用 w = w->parent
276                 }
277     //case4:
```

```
278 /*
279  * 此时状态有两种，w 有两个红孩子，或者有一个右红孩子
280  * 此时有 w 的父节点 (颜色未知)，w(颜色为黑)，w 的右子节点 (颜色为红)，
281      w 的左子节点 (可能存在，存在就为红，存在与否对下面的操作无影响)
282  * 将 w->parent，w，w->right 变为黑红黑，或者黑黑黑，取决于 w 的父节点的颜色，
283      父节点为黑就变为黑黑黑，否则为黑红黑，也就是 w 变为其父亲的颜色
284  * 因为下面需要绕 w 的父亲进行左旋转，w 就会代替其父亲的位置，所以需要 w 和其父亲同色
285  * 看上去就像是 w 失去了一个孩子，该孩子移动到了 x 处
286  */
287             w->color = w->parent->color; //w 变为父亲的颜色
288             w->parent->color = BLACK; //w 的父亲变为黑色
289             w->right->color = BLACK; //w 的右孩子变为黑色
290             left_rotate(w->parent); //左旋转
291             x = T->parent; //退出循环
292         } else { //x 为右孩子，与上面情况对称
293             w = x->parent->left;
294 //case1:
295             if (w->color == RED) {
296                 w->parent->color = RED;
297                 w->color = BLACK;
298                 right_rotate(w->parent);
299                 w = x->parent->right;
300             }
301 //case2:
302             if (w->left->color == BLACK && w->right->color == BLACK) {
303                 w->color = RED;
304                 x = x->parent;
305             } else if (w->left->color == BLACK) {
306 //case3:
307                 w->right->color = BLACK;
308                 w->color = RED;
309                 left_rotate(w);
310                 w = x->parent->right;
311             }
312 //case4:
313             w->color = x->parent->color;
314             w->parent->color = BLACK;
315             w->left->color = BLACK;
316             right_rotate(w->parent);
317         }
318     }
319     x->color = BLACK;
320 }
321
322 void rb_delete(pNode T, int val)
323 {
324     pNode x, y, z;
325     int color;
326
327     if ((z = rb_find(T, val)) == nil) //未找到
328         return;
329     y = z;
330     color = y->color;
331
332     if (z->left == nil) {
333
```

```
334         /*
335          * 如果被删除节点最多一个孩子，该节点可能为黑色，也可能为红色
336          * 如果为红色，那么它必定没有孩子，删除一个无孩子的红节点没有任何影响
337          * 如果为黑色，那么它可能有一个红孩子，也可能没孩子
338          * 如果为黑色且有一个红孩子，则直接使用将该红孩子替换它后变为黑色
339          * 如果为黑色且没有孩子，x 则为 nil 节点，那么则需 rb_delete_pickup 修复该红黑树
340          */
341
342         x = z->right;
343         rb_transplant(z, z->right);
344     } else if(z->right == nil) {
345         x = z->left;
346         rb_transplant(z, z->left);
347     } else {
348
349         /*
350          * 此时被删除的 z 节点有两个孩子，先用后继点 y 移动到被删除的 z 处并变为 z 的颜色，
351              那么可以看做实际上是删除了 y 节点
352          * y 必没有左孩子 (如果其有左孩子，那么其左孩子就是 y 的前继)，而 y 本身必定在 z 的右子树上
353          * 如果 y 是红节点，那么其必定也没有右孩子
354          * 如果 y 是黑节点，那么可能有右孩子，
355              如果有，该右孩子是唯一节点且必为红，只需要用该红孩子移动到 y 处变黑即可
356          */
357
358         y = tree_minimum(z->right);
359         color = y->color; //保存后继点 y 的颜色，因为实际上是 y 被删除了
360         x = y->right; //由于 y 可能有唯一的右红孩子，如果确实有红孩子，该红孩子就能顶替 y，如果没有，x 就是 nil
361         if (y->parent == z) {
362
363             /*
364              * 如果 z 的后继正好是其右孩子
365              * 如果 x 为 nil，则 y 没有右孩子，y 有可能为红，也可能为黑
366              * 由于 x 可能为 nil，需要将 nil 的父节点设置为 y，不然
367                  后面的 rb_delete_fixup 传递的 x 是 nil 却无法找到其父节点
368              */
369
370             x->parent = y;
371         } else {
372
373             /*
374              * 如果 y 不是 z 的右孩子，则先用 y 的右孩子 x(可能为红或者 nil) 替换 y
375              * 由于 y 后面要移动到 z 处，先将 z 的右子树和 y 绑定
376              */
377
378             rb_transplant(y, x);
379             y->right = z->right;
380             y->right->parent = y;
381         }
382
383         /*
384          * 将 y 替换 z，并将 z 的左子树和 y 绑定
385          */
386
387         rb_transplant(z, y);
388         y->left = z->left;
389         y->left->parent = y;
```

```
390     }
391
392     /*
393      * 被删除的节点已经由其后继补上，问题是其后继失去后，其后继的右孩子顶替后继所产生的影响
394      * 如果 x 是红色，说明 y 有一个红色右孩子，y 必然是黑色，
395          前面的操作已经将 x 放置到 y 的位置了，后面只需要调用 rb_delete_fixup 将红变黑即可
396      * 如果 y 为红色且没有右孩子，删除一个红色节点没有影响，下面的 if 语句会失败
397      * 如果 y 为黑色且没有右孩子，删除一个黑色节点后会破坏性质 5，由于 x 是 y 的右孩子，那么 x 就是 nil，
398          如果 x 为一个非 nil 的黑节点，在删除进行前，由于 x 的前继 y 并没有左节点，此时已经破坏了性质 5
399      */
400
401     if (color == BLACK)
402         rb_delete_fixup(T, x);
403 }
404
405 void inordered_walk(pNode T)
406 {
407     if (T != nil) {
408         inordered_walk(T->left);
409         printf("%d ", T->value);
410         inordered_walk(T->right);
411     }
412 }
413
414 int main(void)
415 {
416     setbuf(stdout, NULL);
417     int i, n;
418     nil = (pNode)calloc(1, sizeof(Node));
419     nil->color = BLACK;
420     pNode T = (pNode)calloc(1, sizeof(Node));
421     //设置一个空节点，其 parent 指向根节点，左节点指向最小节点，右节点指向 NULL
422     T->parent = nil;
423     printf(" 请输入想要插入的数目: ");
424     scanf("%d", &i);
425     for (int j = 0; j < i; ++j) {
426         scanf("%d", &n);
427         rb_insert(T, n);
428     }
429     rb_delete(T, 3);
430     inordered_walk(T->parent);
431     return 0;
432 }
```

参考文献