10-Struct & 11-Linkedlist 题解

CPL 助教团队

2023 年 1 月 25 日

目录

> 教学周历中本（两）周课程的知识点为“结构体、联合体、枚举类型、链表”，对应教材章节为 12.4, 13.7, 17.6, 17.7, 16.1 - 16.5。

如果你想要在期末考试中取得高分，请重视本次作业的 3、4、5 题，期末机考 T4 将会对标或略高于本次作业后三题。

1 聪明的保罗

> 本题知识点：结构体、排序、qsort。

简单题。

可以使用结构体保存球队的名称和各种参数信息，按照不同的比较方式排序即可。一个可能需要注意的细节为“平均评分”为浮点数。

理论上大部分排序方式（如课上讲过的冒泡排序、快速排序、归并排序等）都可以通过本题，但为了避免将相似的代码复制多次，我们推荐使用库函数 qsort() 结合函数指针以达到高效复用代码段的目的。

最后重复一遍，如果你对某一个库函数不熟悉但需要使用它，请首先阅读**手册**了解它的行为。

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

typedef struct {
    char name[21];
    char player_name[21];
    int atk;
    int def;
    int tac;
} Team;

int atk_cmp(const void* a, const void* b) {
    return ((Team*)b)->atk - ((Team*)a)->atk;
}
int def_cmp(const void* a, const void* b) {
    return ((Team*)b)->def - ((Team*)a)->def;
}
int tac_cmp(const void* a, const void* b) {
    return ((Team*)b)->tac - ((Team*)a)->tac;
}
```

```
22
23 int (*team_cmp[3])(const void*, const void*) = {
24     *atk_cmp,
25     *def_cmp,
26     *tac_cmp
27 };
28
29 Team team[1000];
30
31 int main(void) {
32     int N;
33     scanf("%d", &N);
34     for (int i = 0; i < N; i++) {
35         scanf("%s", team[i].name);
36         for (int j = 0; j < 11; j++) {
37             char name[21];
38             int a, d, t;
39             scanf("%s%d%d%d", name, &a, &d, &t);
40             team[i].atk += a;
41             team[i].def += d;
42             team[i].tac += t;
43         }
44     }
45     for (int i = 0; i < 3; i++) {
46         qsort(team, N, sizeof(Team), team_cmp[i]);
47         for (int j = 0; j < N; j++)
48             printf("%s%c", team[j].name, " \n"[j == N - 1]);
49     }
50     return 0;
51 }
```

2 多项式运算

本题知识点：结构体、模拟。

首先需要存储和表示多项式。为了“对齐”系数，一个简单的方法是“倒序”存储和表示多项式，即使用 a[i] 表示 x^i 的系数。

在运算的过程中，可以模仿小学生列竖式的思想寻找规律。

$$设P_1=\sum_{i=0}^{p_1}a_ix^i,P_2=\sum_{i=0}^{p_2}b_ix^i，其中a_i=0(i>p_1),b_i=0(i>p_2)$$

$$P_1\pm P_2=\sum_{i=0}^{\max(p_1,p_2)}(a_i\pm b_i)x^i$$

$$P_1\times P_2=\sum_{k=0}^{p_1+p_2}(\sum_{i+j=k}a_i\cdot b_j)x^k$$

计算完对应的系数之后，剩下就是输出了，这一部分许多同学实现的复杂程度远远超过了我们的想象，比起将输出部分的代码复制三遍，一个更好的办法可能是封装成函数使用。

标程对于多项式的运算和内存管理、输出都有非常优雅的实现，值得大家学习。

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define max(a, b) ((a) > (b) ? (a) : (b))
#define newPoly(pname, p)                                   \
    poly_t pname;                                           \
    pname.maxOrder = (p);                                   \
    pname.coe = (int *)malloc(sizeof(int) * ((p) + 1));
#define maxLen 12
char *name;
struct poly {
    int maxOrder;
    int *coe;  // coefficients
};
typedef struct poly poly_t;
void printItem(int coe, int p)
{
    if (p == 0)
    {
        printf("%d", coe);
        return;
    }
    if (coe != 1 && coe != -1)
        printf("%d", coe);
    else if (coe == -1)
        printf("-");
    printf("%s", name);
    if (p > 1)
```

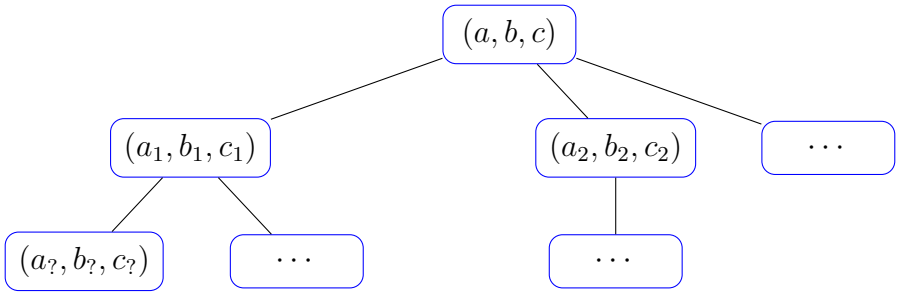
```
29          printf("^%d", p);
30  }
31  void printPoly(poly_t P)
32  {
33      for (int i = P.maxOrder; i >= 0; i--)
34      {
35          int coe = P.coe[i];
36          if (coe > 0 && i != P.maxOrder)
37              printf("+");
38          else if (coe == 0)
39              continue;
40          printItem(coe, i);
41      }
42      printf("\n");
43  }
44  void polyAddMinus(poly_t P1, poly_t P2, int add)
45  {
46      newPoly(P, max(P1.maxOrder, P2.maxOrder));
47      for (int i = 0; i <= P.maxOrder; i++)
48          P.coe[i] = (i <= P1.maxOrder ? P1.coe[i] : 0) +
49              (i <= P2.maxOrder ? P2.coe[i] : 0) * (add ? 1 : -1);
50      printPoly(P);
51      free(P.coe);
52  }
53  void polyMul(poly_t P1, poly_t P2)
54  {
55      newPoly(P, P1.maxOrder + P2.maxOrder);
56      memset(P.coe, 0, sizeof(int) * (P.maxOrder + 1));
57      for (int i = P1.maxOrder; i >= 0; i--)
58          for (int j = P2.maxOrder; j >= 0; j--)
59              P.coe[i + j] += P1.coe[i] * P2.coe[j];
60      printPoly(P);
61      free(P.coe);
62  }
63  int main(void)
64  {
65      int p1, p2;
66      name = (char *)malloc(sizeof(char) * maxLen);
67      scanf("%d%d%s", &p1, &p2, name);
```

```
68      newPoly(P1, p1);
69      newPoly(P2, p2);
70      for (int i = p1; i >= 0; i--)
71          scanf("%d", P1.coe + i);
72      for (int i = p2; i >= 0; i--)
73          scanf("%d", P2.coe + i);
74      polyAddMinus(P1, P2, 1);
75      polyAddMinus(P1, P2, 0);
76      polyMul(P1, P2);
77      free(P1.coe);
78      free(P2.coe);
79      return 0;
80  }
```

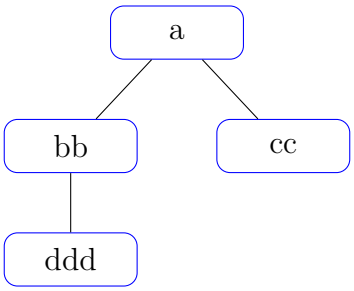
3 文件管理系统

本题知识点：结构体、模拟、递归。

在本题中，我们需要用指针维护一个（最多二叉）的树形结构，支持删除和查询（子树）siz 。什么叫树形结构？[1]其实我们已经见过它很多次了，大家可以回忆一下 7-data-types 的那次题解。



上图是当时我们为了表示递归遍历的顺序画出来的，其实这个顺序画出来就是一个典型的树形结构，它长得比较整齐。如果我们把本题中的第一个样例画出来，则会是下面这样。



考虑到每一个“文件”结构的相似性，我们可以定义一个结构体。

[1]具体的定义会在下学期离散数学课上给出

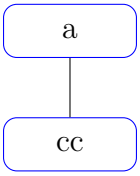
```
struct file_ {
    char name[11]; // 请注意，这里的长度至少为 10 + 1 = 11
    struct file_ *file1, *file2;
};
typedef file_ file;
```

其中，$file1$ 和 $file2$ 两个指针分别表示 cur 下方的（可能存在的）直接相连的子文件。

在尝试查询的过程中，我们需要查询的是每一个文件下方直接或间接有多少个子文件，我们可以将这个问题分解到两个直接与当前的文件相连的子文件上，那么就可以自然地写出一个递归函数。

```
int query(file* cur) {
    if (cur == NULL) return 0;
    return 1 + query(cur->file1) + query(cur->file2);
}
```

接下来考虑删除，如果我们删除了 bb 文件，那么就变成了这样。



只需要找到 bb 的直接的上一级文件，然后把 bb 所在的指针设为 NULL 就可以了。我们也可以类似地写出一个递归函数。

```
void remove(file* cur, char* target) {
    if (cur == NULL) return;
    if (cur->file1 != NULL &&
        (!strcmp(cur->file1->name, target))) {
        cur->file1 = NULL;
        return;
    }
    if (cur->file2 != NULL &&
        (!strcmp(cur->file2->name, target))) {
        cur->file2 = NULL;
        return;
    }
    remove(cur->file1, target);
    remove(cur->file2, target);
}
```

最后就是这个结构如何构建的问题了，相似地，我们也可以递归地找到直接的上一级文件，然后把上一次文件的 $file1$ 或 $file2$ 指针设为它就可以了。

```
void insert(file* cur, char* target, char* newFileName) {
    if (cur == NULL) return;
    if (!strcmp(cur->name, target)) {
        file* newfile = (file*) malloc(sizeof(file));
        strcpy(newfile->name, newFileName);
        if (cur->file1 == NULL) cur->file1 = newfile;
        else cur->file2 = newfile;
        return;
    }
    insert(cur->file1, target, newFileName);
    insert(cur->file2, target, newFileName);
}
```

那么本题就做完了，上面已经给出了核心代码，所以就不再给出完整的标程了。

另外一点需要说明的是，调换输入的先后顺序，在文件系统构建的过程中，本题是允许出现“找不到直接的上一级文件”的情况的。~~但是由于我造数据的时候偷懒了，所以没有这种情况。~~ 事实上，如果要兼容这种情况，实现也不会变得特别复杂，因为本题是以名称作为文件的唯一标识的，我们可以直接遍历现有的所有的文件，对比当前的 *target* 名称是否被找到，如果没找到，那就直接创建一个新的文件。

还有一点想要说明的是，我们在答疑过程中，发现很多同学将递归写成了这种样子，然后获得了时间超时的结果。

```
file* find(file* cur, char* target) {
    if (find(cur->file1) != NULL)
        return find(cur->file1);
    ...
}
```

需要注意，这种情况会反复调用 find(cur->file1) 这个函数，造成大量冗余计算。我们可以用老师课堂上讲过的记忆化的方法来处理这种情况。

```
file* find(file* cur, char* target) {
    file* tmp = find(cur->file1);
    if (tmp != NULL)
        return tmp;
    ...
}
```

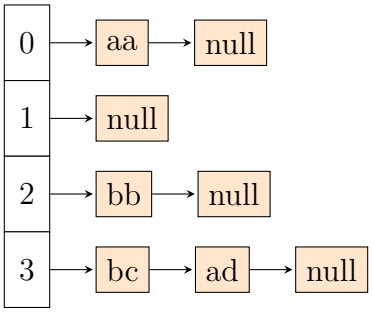
4 字符串哈希

本题知识点：结构体，指针数组，链表，哈希，字符串。

哈希是一种常见的用来判断一个字符串集合中某个输入字符串是否存在的算法。当字符串数量太过庞大的时候，我们难以将输入的字符串和每个已存在的字符串都进行比较，因此我们可以提取它们的一些数字特征（比如将字符串中的字母视为一个很大的 26 进制数并对一个大质数取模），仅仅对拥有相同数字特征的字符串进行比较，就能大大提高检索的效率。上面这个提取数字特征的过程就称作字符串哈希。

假设我们已经有了一个哈希函数 $hash(s)$ 可以返回给我们一个 int 类型的数字，它表示了这个字符串的某种特征，为了将所有满足这种特征（hash 值相同）的字符串放到一起，我们可以利用桶的思想，实现多个链表，将每一个字符串 s 放入 $hash(s)$ 所指示的链表中。这样，当我们遇到一个新的字符串 t 时，我们只需要去 $hash(t)$ 所指示的链表中寻找是否存在一个字符串为 t 即可。

举个例子，设 $hash(s) = \sum(s_i - 97) \mod 4$ （把字符串中的字符看成 ascii 码），假设原有的集合为 $\{"ad","bc","aa","bb"\}$，那么我们构造出的链表应该是这样的。



这时候询问“e”这个字符串是否在集合中，我们先取它的哈希值 $hash(e) = 3$，然后找到 3 这个值所指示的链表，去分别比对链表中的元素是否有 e 。容易发现，比对 $hash("e")$ 所指示的链表中出现的字符串所得到的结果和把全部字符串都比对一遍的结果是一样的。

用陈振宇老师的话来讲[2]，数据处理的核心是降维，字符串哈希又何尝不是一种降维呢？（手动菜狗）

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

char *strdup(char *);

typedef struct node {
        struct node *next;
        char *str;
} node;

```

[2]如果你们下学期选修了他的《数据科学基础》的话

```
node *nodes[500000];

int hash(char *s, int mod) {
        int len = strlen(s);
        int val = 0;
        int res = 0;
        for (int i = 0; i < len; i++) {
                if ('a' <= s[i] && s[i] <= 'z')
                    val = s[i] - 'a' + 26;
                else val = s[i] - 'A';
                res *= 52;
                res += val;
        }
        int ret = res % mod;
        return ret < 0 ? ret + mod : ret;
}

void add_str(char *s) {
        int id = hash(s, 500000);
        if (!nodes[id]) {
                nodes[id] = malloc(sizeof(node));
                nodes[id]->next = NULL;
                nodes[id]->str = s;
                return;
        }
        node *h = nodes[id];
        while (h->next != NULL) h = h->next;
        h->next = malloc(sizeof(node));
        h->next->next = NULL;
        h->next->str = s;
}

bool find_str(char *s) {
        int id = hash(s, 500000);
        node *h = nodes[id];
        while (h != NULL) {
                if (!strcmp(h->str, s)) return true;
                h = h->next;
        }
```

```
        return false;
}

char buf[1024];
int main() {
        int m, q;
        scanf("%d%d", &m, &q);
        while (m--) {
                scanf("%s", buf);
                add_str(strdup(buf));
        }
        while (q--) {
                scanf("%s", buf);
                if (find_str(buf)) puts("Yes");
                else puts("No");
        }
        return 0;
}
```

需要注意的是，哈希函数的选择会对运行效率产生很大的影响，我们希望这个函数可以将所有字符串尽可能映射到不同的桶中，否则运行效率提升不大，像我上面那个用来举例子的函数（求和）就是一个很差的哈希函数，一个较为常用的哈希函数就如标程实现的那样，将字符串看成一个 26 进制数对大数取模。哈希本身是一件比较有意思的事情，大家以后可以自己探索一下。

5 内存分配器

如需获取解析请参考 7-data-types 的题解，此处直接给出一个指针和链表的参考实现。

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

struct node {
  int siz, id; // id = 0: free
  struct node *next;
};
typedef struct node Node;

Node *newNode(int siz) {
  Node *node = (Node *)malloc(sizeof(Node));
```

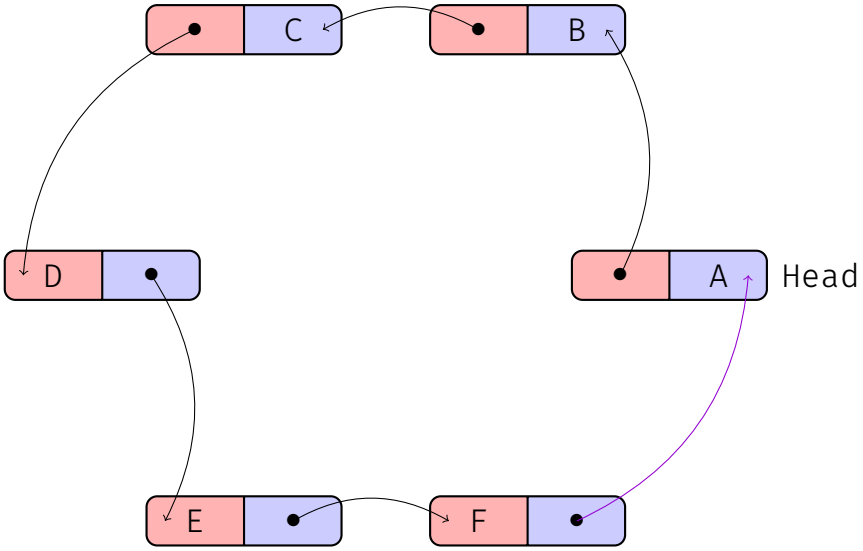
```
13    node->next = NULL;
14    node->siz = siz;
15    node->id = 0;
16    return node;
17  }
18
19  int fitSize(int m) {
20    int ret = 0;
21    for (; (1 << ret) < m; ++ret);
22    return ret;
23  }
24
25  Node *getBest(Node *head, int siz) {
26    for (Node *p = head; p; p = p->next) {
27      if (p->id != 0) continue;
28      if (p->siz == siz) return p;
29    }
30    for (Node *p = head; p; p = p->next) {
31      if (p->id != 0) continue;
32      if (p->siz > siz) return p;
33    }
34    return NULL; // never reaches here
35  }
36
37  void alloc(Node *head, int id, int siz) {
38    while (true) {
39      Node *tmp = getBest(head, siz);
40      if (tmp->siz == siz) {
41        tmp->id = id;
42        return;
43      } else {
44        // split
45        // before : ... -> tmp -> ...
46        // after  : ... -> tmp' -> tmp2 -> ...
47        Node *tmp2 = newNode(tmp->siz - 1);
48        tmp2->next = tmp->next;
49        tmp->next = tmp2;
50        tmp->siz--;
51      }
```

```
52    }
53  }
54
55  void query(Node *head) {
56    int cnt = 0;
57    for (Node *p = head; p; p = p->next) ++cnt;
58    printf("%d\n", cnt);
59    for (Node *p = head; p; p = p->next) {
60      printf("%d", p->id);
61      if (p->next) printf(" ");
62    }
63    printf("\n");
64  }
65
66  int main() {
67    int n, q;
68    scanf("%d%d", &n, &q);
69    Node *head = newNode(n);
70    while (q--) {
71      char op[3];
72      scanf("%s", op);
73      if (op[0] == 'A') {
74        int id, m;
75        scanf("%d%d", &id, &m);
76        alloc(head, id, fitSize(m));
77      } else {
78        // op[0] == 'Q'
79        query(head);
80      }
81    }
82    return 0;
83  }
```

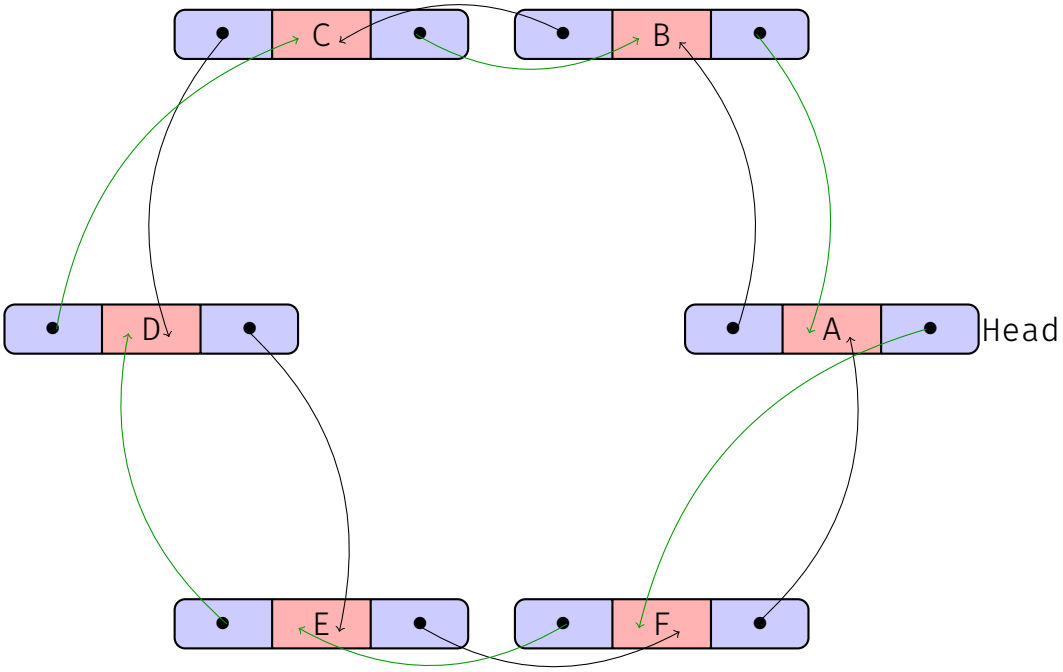
6 链表 *

题意已经说明，这是一个循环链表，在原有的链表的基础上，我们需要把尾结点的 *next* 指针指向头结点（紫色箭头）。[3]

[3]链表和 Stack Pour 是 2021 年作业题，也对标期末考试 T4，已经放入 Problem Set 中



注意到 Backward 的操作仍然不好实现，我们可以添加一个 *pre* 指针（绿色箭头），指向当前结点的前一个结点。



为了实现更快，我们可以对 Forward 和 Backward 的次数 m 对于**当前链表大小** n 进行取模，请注意 **“当前”**，因为我们可以进行 Insert 和 Remove 的操作，所以 n 的大小会动态变化，在相应的操作中分别维护一下即可。

```
#include <stdio.h>
#include <stdlib.h>
typedef struct linklist list_t;
struct linklist
{
        list_t *prev, *next;
        int val;
};
```

```
9  int N;
10 list_t *create(int n)
11 {
12         list_t *head = (list_t *)malloc(sizeof(list_t));
13         list_t *tail = head;
14         head->val = 1;
15         for (int i = 2; i <= n; i++)
16         {
17                 list_t *newNode = (list_t *)malloc(sizeof(list_t));
18                 newNode->val = i;
19                 newNode->prev = tail;
20                 tail->next = newNode;
21                 tail = newNode;
22         }
23         tail->next = head;
24         head->prev = tail;
25         return head;
26 }
27 list_t *Forward(list_t *ptr, int m)
28 {
29         m %= N;
30         while (m--)
31                 ptr = ptr->next;
32         return ptr;
33 }
34 list_t *Backward(list_t *ptr, int m)
35 {
36         m %= N;
37         while (m--)
38                 ptr = ptr->prev;
39         return ptr;
40 }
41 void Insert(list_t *ptr, int m)
42 {
43         list_t *newNode = (list_t *)malloc(sizeof(list_t));
44         newNode->val = m;
45         newNode->next = ptr->next;
46         newNode->prev = ptr;
47         ptr->next->prev = newNode;
```

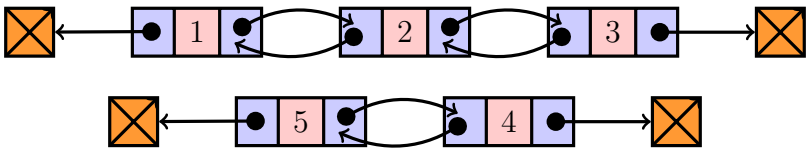
```
48          ptr->next = newNode;
49          N++;
50  }
51  list_t *Remove(list_t *ptr)
52  {
53          list_t *dummy = ptr;
54          ptr = ptr->next;
55          dummy->prev->next = ptr;
56          ptr->prev = dummy->prev;
57          N--;
58          free(dummy);
59          return ptr;
60  }
61  void Print(list_t *ptr)
62  {
63          printf("%d\n", ptr->val);
64  }
65  int main(void)
66  {
67          int T, m;
68          scanf("%d%d", &N, &T);
69          list_t *node = create(N);
70          char op[10];
71          while (T--)
72          {
73                  scanf("%s", op);
74                  switch (op[0])
75                  {
76                  case 'F':
77                          scanf("%d", &m);
78                          node = Forward(node, m);
79                          break;
80                  case 'B':
81                          scanf("%d", &m);
82                          node = Backward(node, m);
83                          break;
84                  case 'I':
85                          scanf("%d", &m);
86                          Insert(node, m);
```

```
87                          break;
88                  case 'R':
89                          node = Remove(node);
90                          break;
91                  case 'P':
92                          Print(node);
93                          break;
94                  default:
95                          break;
96                  }
97          }
98          return 0;
99  }
```

7 Stack Pour*

这道题考察的是同学们对链表这种结构的理解。

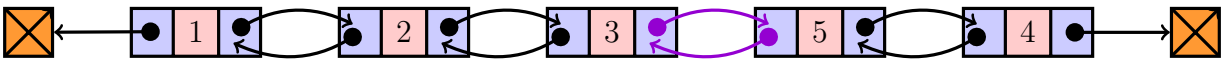
观察模拟 Stack 们倒来倒去的过程，我们可以发现一个栈中的所有数的相对次序不会再变化（具体地说，当一个数的前驱和后继确定的时候，就不会再发生变化），因此我们可以借助指针把这些关系保存下来。下图为一个例子。



在上面这个例子中，$(1,2,3)$ 为一个当前的栈，$(4,5)$ 则是另一个，我们没有确定两个栈的栈顶分别在哪里，但是当我们发现除了指向 NULL （打叉叉的正方形）的指针，其他指针的值不再会发生变化。

仔细看一下，就能发现这个东西足以被称为链表。

如果此时我们指定了 $(1,2,3)$ 的栈顶为 3 这一端，而 $(4,5)$ 的栈顶为 5 这一端，我们现在把第一个栈倒入第二个栈，则倒完之后的结果为 $(4,5,3,2,1)$ ，显然 1 这一端是栈顶。如果我们只考虑数的大小关系的话，我们会发现只需要把 3 右端的指针指向 5 ，并且把 5 左端的指针指向 3 即可。



这时可以发现，合并的过程只和链表两端的空指针有关，并且合并之后还是满足同一种形态（即“左右两边都有空指针”这种形态）。因此我们只需要在最开始的时候维护 n 个只有一个结点（而且有一左一右两个指向 NULL 的指针）的链表，并写一个支持合并的函数就可以了。

还剩下一个问题，那就是怎么确定栈顶和栈底。容易发现，在所有合并都没有开始的时候，这个问题非常的易于回答（你想把哪边叫栈底都行），而且在合并过程中，我们可以推出栈底和栈顶的变化（具体可以参考下面的标程），那么这个问题也就迎刃而解了。

```
#include <stdio.h>
#define MAXN 1000005

int top[MAXN], bottom[MAXN];
int nxt1[MAXN], nxt2[MAXN];
int vis[MAXN];
void link(int x, int y) {
        int* px = nxt1[x] ? &nxt2[x] : &nxt1[x];
        int* py = nxt1[y] ? &nxt2[y] : &nxt1[y];
        *px = y;
        *py = x;
}

int main() {
        int n, m;
        scanf("%d %d", &n, &m);
        for (int i = 1; i <= n; ++i)
                top[i] = bottom[i] = i;

        while (m--) {
                int x, y;
                scanf("%d %d", &x, &y);
                if (top[x] == 0)
                        continue;
                if (top[y] == 0) {
                        top[y] = bottom[x];
                        bottom[y] = top[x];
                        bottom[x] = top[x] = 0;
            continue;
                }
                link(top[x], top[y]);
                top[y] = bottom[x];
                bottom[x] = top[x] = 0;
        }

        vis[0] = 1;
        for (int i = 1; i <= n; ++i) {
        if (top[i] == 0) {
            puts("0");
```

```
40              continue;
41          }
42                  for (int j = bottom[i]; !vis[j];
43              j = vis[nxt1[j]] ? nxt2[j] : nxt1[j]) {
44              vis[j] = 1;
45                          printf("%d ", j);
46          }
47                  puts("");
48          }
49
50          return 0;
51  }
```

需要说明的是，这个标程并不是指针实现的，而是采用了**数组模拟链表**的形式。考虑到指针运行的速度较慢，这种方法一来可以减少写指针犯错误的可能，而且还比较好写，二来可以提高一些运行效率，可以算是规避了指针的一些（小小的）不足之处。大家在期末考试的时候可以采取这种方法，但是在真正的工程或者是项目上（比如我们 project 作业），我们**极不推荐**这种方式，还请大家优雅地使用指针。

8 写在最后

> （2021 年）
>
> 终于写完了本学期最后一份题解，除了解放的快乐，我还感觉到有一点舍不得。
>
> 转眼间已经过去了一个学期，在我们的课程中，大家一共完成了 8 次 OJ 作业和一次小项目，总计 54 题。我知道，我们的 OJ 题量有点大，大家每周可能需要花费 7 到 8 个小时在我们的作业上。现在回过头来看看，我也觉得这样的任务量还是有点大了。
>
> 不管怎么样，任务再难，大家也都坚持下来了。从结果来看，大家编程的水平的提升应该是比较显著的。能看到这样的结果，我就已经很开心了（我知道你们可能有点不开心，q ω q）。感谢大家的理解、支持与陪伴，希望大家能够真正在这门课上有所收获。
>
> 最后还是祝大家期末考试顺利，考出自己想要的成绩。

本学期的最后一份题解也完工了。2022 年的课程中，大家一共完成了 10 次正式的 OJ 作业和两个 project ，总计 53 题。

任务量依然很大，而在今年苏州校区的参与中，我们又一次思考这样大的练习量究竟有没有很好地帮到大家。

依然是感谢大家的理解、支持与陪伴，希望大家能够真正在这门课上有所收获。新年快乐！