C语言习题课

2023.12.19

习题课

2021-Exam

2022-12-08 22:30 - 2023-01-31 23:00

2021级期末机试原题

考试时长：3小时（后延长30分钟，共3小时30分钟）

大家自行体验，不计分，自己把握时间试一下，今年机试题型和难度与去年**几乎相同**！

题目列表

#	名称	解决状态	通过人数
A	升序检测 (sorted)		90 / 100
B	二分查找 (bsearch)		80 / 92
C	五子棋 (gobang)		24 / 51
D	列队 (lineup)		11 / 21

1.升序检测（sorted）

➢ 题目描述

输入一个长度为 n 的整数数列（允许有重复数字），对其进行 m 次操作：每次将数列中第 x 个（注意：这里下标从 1 开始）元素从数列中取出，然后将其加入到数列首部。
判断所有操作完成后，数列是否从小到大排好序。

➢ 输入格式

第一行一个整数 T，表示数据组数，T ≤10。
每组数据包括三行：

- 第一行两个整数 n，m，表示数列长度和操作次数，n ≤ 1000, m≤1000。
- 第二行 n 个整数，表示初始数列，每个整数在（常见环境的）int范围内。
- 第三行 m 个数，第 i 个数表示第 i 次操作的数列下标 x，1 ≤ x ≤ n。

➢ 输出格式

每行一个字符串：

- 若数组由小到大排好序，输出I love C programming language。
- 否则，输出mayi is a good teacher。

- *注解：已放宽时限，使用数组实现即可*

Input

```
3
4 3
1 2 3 4
2 3 4
4 4
2 3 4 1
3 3 3 4
4 2
3 4 2 1
3 4
```

Output

```
mayi is a good teacher
I love C programming language
I love C programming language
```

1.升序检测（sorted）

Input

3 → 三组数据

---

4 3 → 数列长度是4，操作次数是3

1 2 3 4 → 初始数列

2 3 4 → 将第 i 个元素从数列中取出，然后加入到数列首部

---

4 4

2 3 4 1

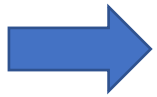
3 3 3 4

---

4 2

3 4 2 1

3 4



初始：1 2 3 4
---> 2 1 3 4
---> 3 2 1 4
---> 4 3 2 1

不是升序，
输出 “mayi is a good teacher”

1.升序检测（sorted）

Input

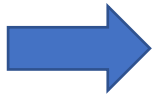
3 → 三组数据

---

4 3 → 数列长度是4，操作次数是3

1 2 3 4 → 初始数列

2 3 4 → 将第 i 个元素从数列中取出，然后加入到数列首部



```
初始：1 2 3 4
--->  2 1 3 4
--->  3 2 1 4
--->  4 3 2 1
```

替换操作：

假设sort是要替换数组下标

```
scanf("%d", &sort);
int temp = number[sort];
for (int j = sort; j > 1; j--){
    number[j] = number[j - 1];
}
number[1] = temp;
```

1.升序检测（sorted）

主函数框架：

```
int main() {
    int T, n, m;
    scanf("%d", &T);
    for (int i = 0; i < T; i++) {
        scanf("%d%d", &n, &m);
        mysorted(n, m);
        judge(number, n);
    }
    return 0;
}
```

判断升序：

```
void judge(int numb[], int len) {
    int issorted = 1;
    for (int i = 0; i < len - 1; i++) {
        if (numb[i] > numb[i + 1]) {
            issorted = 0;
        }
    }
    if (issorted == 0) {
        printf("mayi is a good teacher\n");
    } else {
        printf("I love C programming
language\n");
    }
}
```

2.二分查找（bsearch）

➢ 题目描述

给定一个包含 n 个整数的数组 A，这 n 个整数各不相同且按升序排列（即，A[0]<A[1]<…< A[n-1]）。
现对该数组进行 q 次询问，每次询问输入一个数，需返回该数在数组 A 中的下标。
若该数不存在，则输出 -1。

➢ 输入格式

- 第一行两个 int 型整数 n, q (1⩽n, q⩽1e6)，分别表示数组大小和询问的次数。
- 第二行 n 个 int 型整数。保证升序排序，且互不相同。
- 接下来 q 行，每行一个 int 型整数，代表被查询的数。

➢ 输出格式

每行一个整数，为待查询整数在数组中的下标。
若待查询整数不存在，则输出 -1。

➢ 数据规模与约定

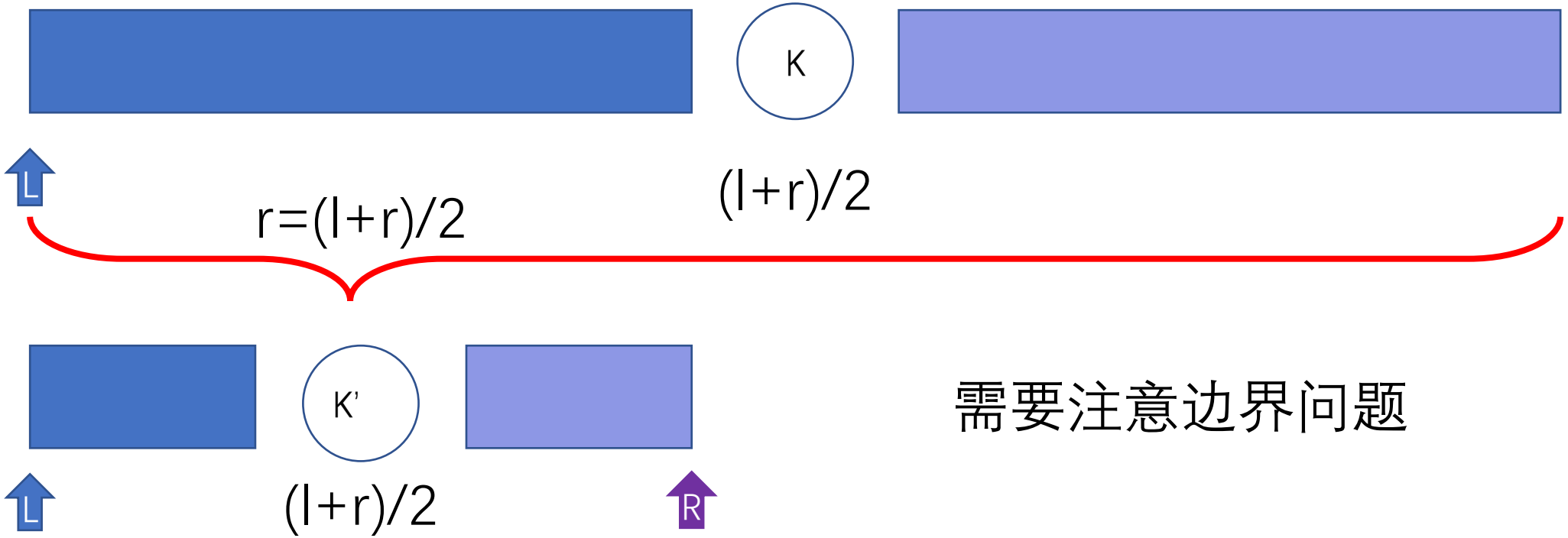
对于 60% 的数据，1⩽n, q⩽10,000
对于剩余 40% 的数据，1⩽n, q⩽1,000,000

Input	Output
4 3 -1 2 3 4 -1 2 3	0 1 2

二分查找&快速排序

二分查找（折半查找）

对一有序的数据集合，先找出数据集合最中间的元素，将数据划分为两个子集，将最中间的元素和关键字进行比较，如果相等则返回，如果大于关键字，则在较大的数据子集中查找，如果想小于反之，直至找到为止。



需要注意边界问题

二分查找&快速排序

二分查找（折半查找）

```
int search(int nums[], int size, int target) // nums是数组，size是数组的大小，target是需要查找的值
{
    int left = 0;
    int right = size - 1;
    while (left <= right)
    {
        int middle = left + ((right - left) / 2); //等同于 (left + right) / 2，防止溢出
        if (nums[middle] > target)
            right = middle - 1; // target在左区间，所以[left, middle - 1]
        else if (nums[middle] < target)
            left = middle + 1; // target在右区间，所以[middle + 1, right]
        else
            return middle; //既不在左边，也不在右边，找到答案了
    }
    return -1; //没有找到目标值
}
```

2.二分查找（bsearch）

二分查找：

```
int find(int list[], int n, int left, int right) {
    int middle = (left + right) / 2;
    if (list[left] == n)   return left;
    if (list[right] == n)   return right;
    if (middle == left || middle == right)
        return -1;
    else {
        if (list[middle] < n)
            return find(list, n, middle, right);
        else
            return find(list, n, left, middle);
    }
}
```

2.二分查找（bsearch）

主函数框架：

```
#include <stdio.h>
int main() {
    int length, operate_times;
    scanf("%d%d", &length, &operate_times);
    int list[1000005];
    for (int i = 0; i < length; i++) {
        scanf("%d", &list[i]);
    }
    for (int j = 1; j <= operate_times; j++) {
        int seek;
        scanf("%d", &seek);
        int answer = find(list, seek, 0, length - 1);
        printf("%d\n", answer);
    }
    return 0;
}
```

3.五子棋（gobang）

➢ 题目描述

首先在横线、竖线或斜对角线上形成 5 子连线者获胜。规定执黑者为先手。假设现在轮到后手（即执白者）落子，并且从此刻开始，对局双方突然变得足够聪明。此时，先手想知道游戏能否在两步以内决出胜负（提示：只考虑接下来的两步，不要考虑得过于复杂）。

➢ 输入格式

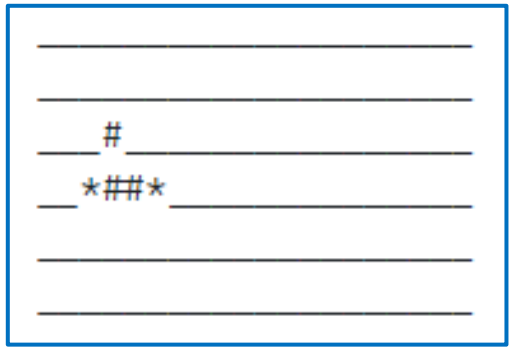
第一行为一个整数 T，表示数据组数。
接下来有 T 组数据，每组数据输入 20 行长度为 20 的字符串：

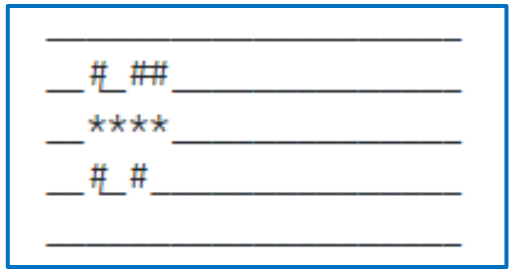
- 以 # 表示黑子
- 以 * 表示白子
- 以 _ 表示空格，即尚未落子。

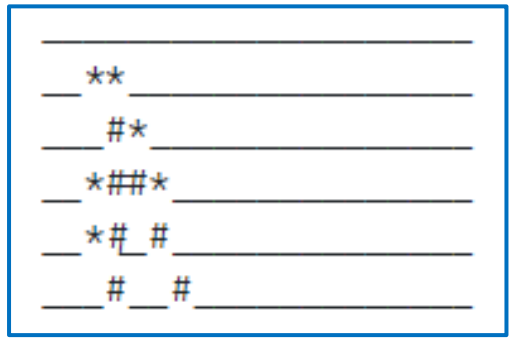
➢ 输出格式

对于每组数据，输出一行一个字符串：

- 如果两步之内无法决出胜负，输出 "Not Sure"
- 如果先手胜，输出 "Win"
- 如果后手胜，输出 "Lose"。

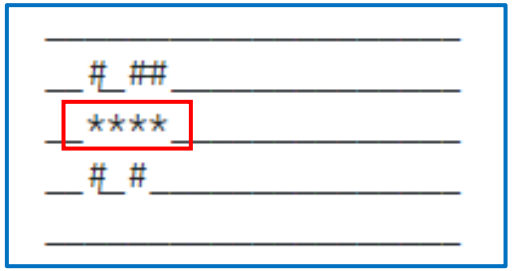




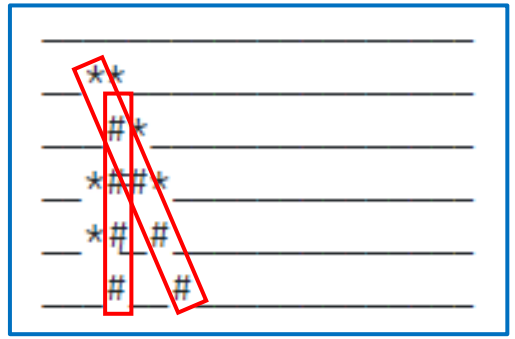


3.五子棋（gobang）

- 以 # 表示黑子
- 以 * 表示白子
- 以 _ 表示空格，即尚未落子
- 假设现在轮到后手（即”*”）落子
- 能否在两步以内决出胜负
- 如果先手（“#”）胜，输出 "Win “
- 如果先手（“#”）胜，输出 "Lose"

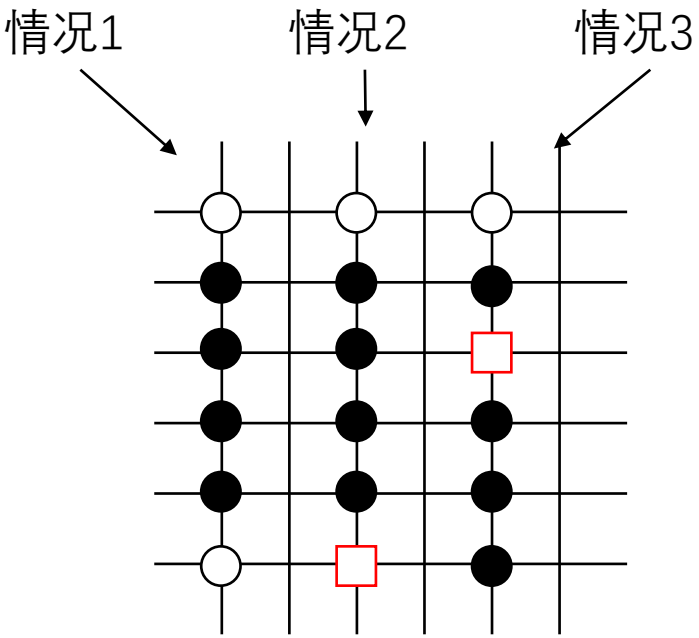


Lose



Win

```
int T = 0;
scanf("%d ", &T);
for (int h = 0; h < T; h++){
    for (int i = 1; i < 21; i++){
        for (int j = 1; j < 21; j++){
            scanf("%c", &a[i][j]);
        }
        scanf(" ");
    }
    // 判断是否获胜，并输出
}
```



3.五子棋（gobang）

水平方向：

```
int horizontal(char x){
    int count=0;
    for (int i=1;i<21;i++){
        for (int j=1;j<21;j++){
            int h=a[i][j]+a[i][j+1]+a[i][j+2]+a[i][j+3]+a[i][j+4];
            if (h==4*x+'_'){
                count++;
            }
        }
    }
    return count;
}
```

3.五子棋（gobang）

斜向右下方向：

```
int left(char x){
    int count=0;
    for (int i=1;i<21;i++){
        for (int j=1;j<21;j++){
            int h=a[i][j]+a[i+1][j+1]+a[i+2][j+2]+a[i+3][j+3]+a[i+4][j+4];
            if (h==4*x+'_'){
                count++;
            }
        }
    }
    return count;
}
```

- 竖直方向、斜向左下方向也是类似
- 有没有可能把这几个判断写到一起，这样就不会反复调用这个双循环了

3.五子棋（gobang）

主函数框架：

```
int p = check('*');
int q = check('#');
if (p >= 1){
    printf("Lose");
}
else{
    if (q >= 2){
        printf("Win");
    } else{
        printf("Not Sure");
    }
}
```

Check函数：

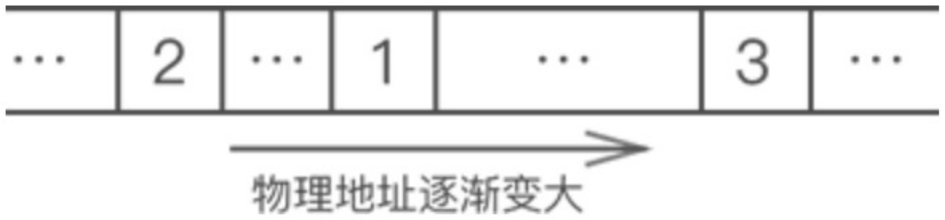
```
int check(char a1){
    return horizontal(a1)+vertical(a1)+left(a1)+right(a1);
}
```

复习: 链表

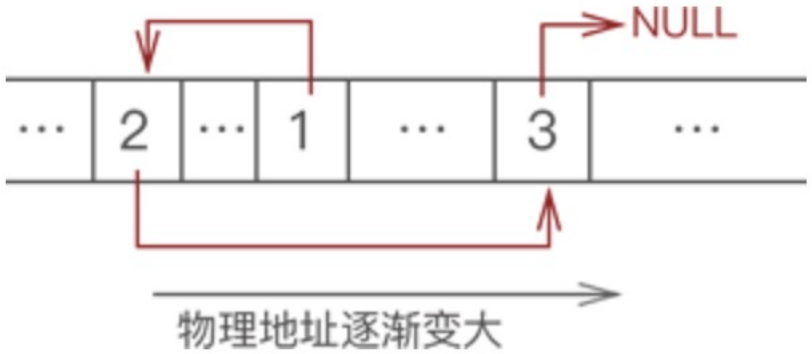
➢ 链表（链式存储结构）及创建

链表，别名链式存储结构或单链表，用于存储逻辑关系为 "一对一" 的数据。与顺序表不同，链表不限制数据的物理存储状态，换句话说，使用链表存储的数据元素，其物理存储位置是随机的。

例如，使用链表存储 {1,2,3}，数据的物理存储状态如下图所示：

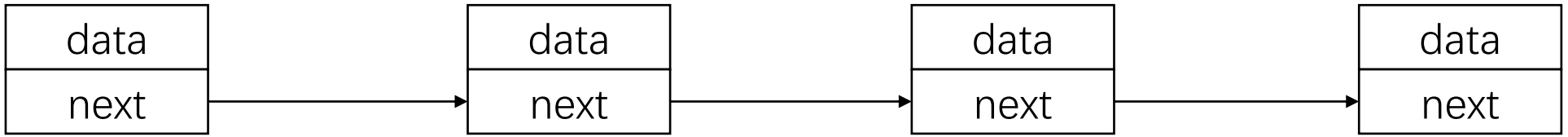


上图根本体现出各数据之间的逻辑关系。对此，链表的解决方案是，每个数据元素在存储时都配备一个指针，用于指向自己的直接后继元素。如下图所示：

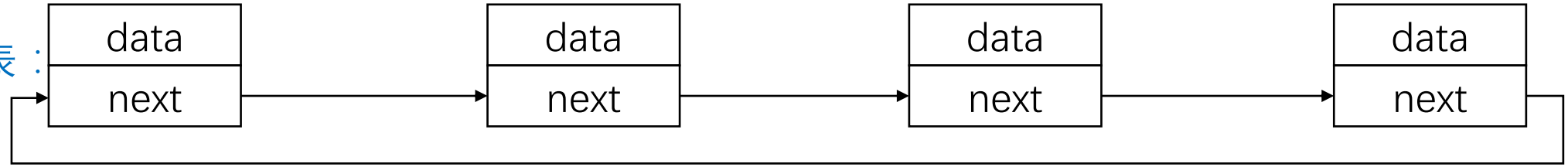


复习: 链表

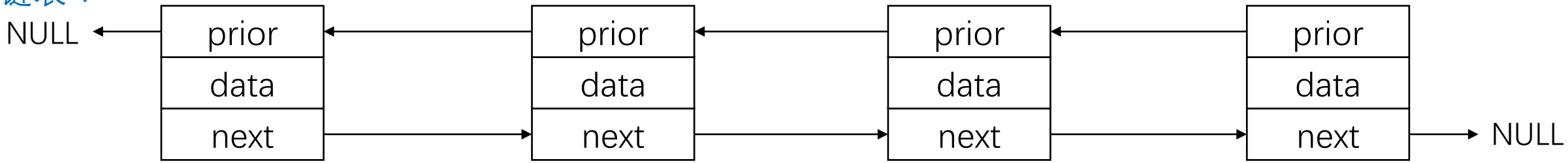
单向链表：



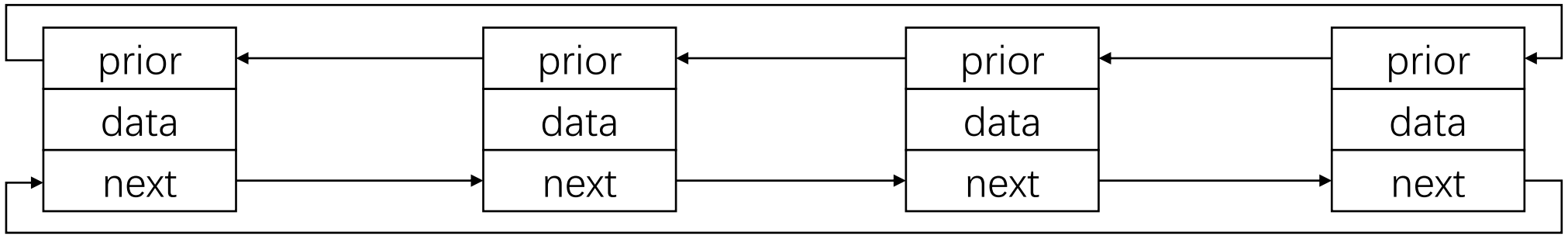
单向循环链表：



双向链表：



双向循环链表：



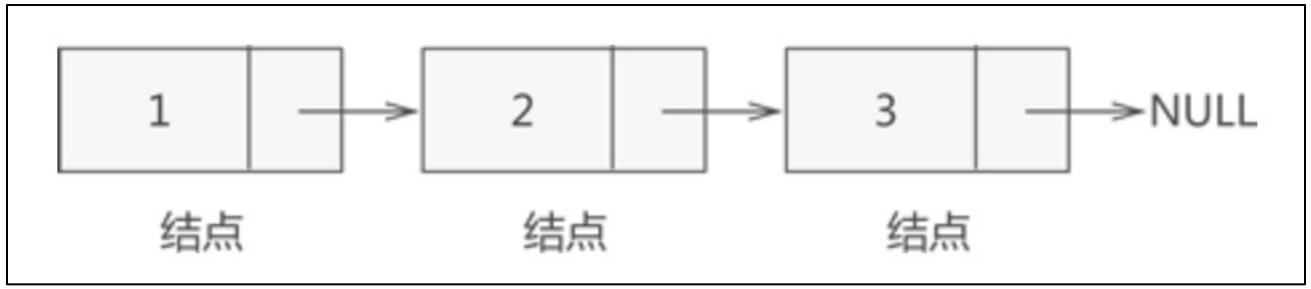
单向链表

➢ 链表的节点

链表中每个数据的存储都由以下两部分组成：

1. 数据元素本身，其所在的区域称为数据域；
2. 指向直接后继元素的指针，所在的区域称为指针域；

上图所示的结构在链表中称为节点。也就是说，链表实际存储的是一个一个的节点，真正的数据元素包含在这些节点中，如下图所示：



链表中每个节点的具体实现，需要使用 C 语言中的结构体，具体实现代码为：

```
typedef struct Linklist{
    int elem;  //代表数据域
    struct Linklist *next;  //代表指针域，指向直接后继元素
}Linklist; //link为节点名，每个节点都是一个 link 结构体
```

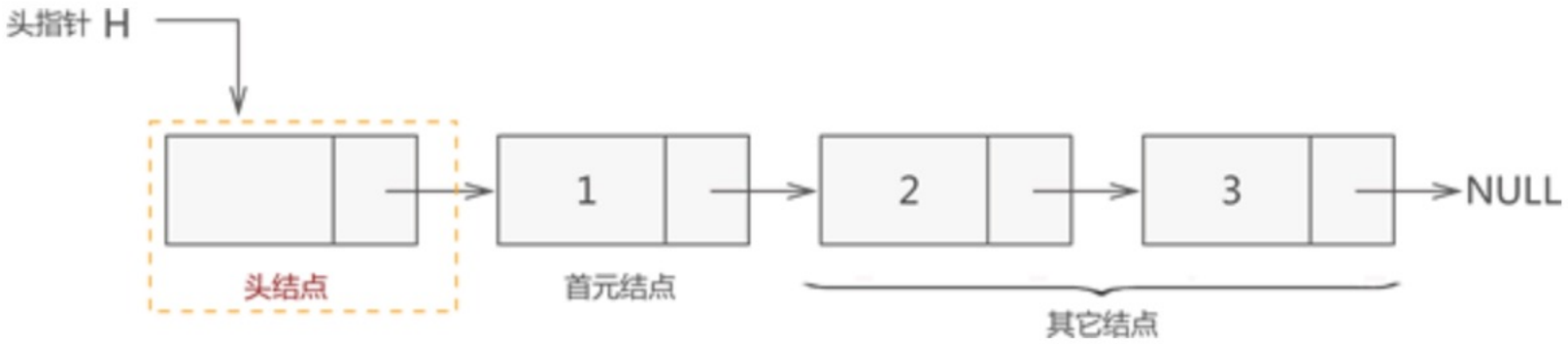
单向链表

➢ 头节点，头指针和首元节点

一个完整的链表需要由以下几部分构成：

1. 头指针：一个普通的指针，它的特点是永远指向链表第一个节点的位置。很明显，头指针用于指明链表的位置，便于后期找到链表并使用表中的数据；
2. 节点：链表中的节点又细分为头节点、首元节点和其他节点。
   - 头节点：其实就是一个不存任何数据的空节点，通常作为链表的第一个节点。对于链表来说，头节点不是必须的，它的作用只是为了方便解决某些实际问题；
   - 首元节点：由于头节点（也就是空节点）的缘故，链表中称第一个存有数据的节点为首元节点。首元节点只是对链表中第一个存有数据节点的一个称谓，没有实际意义；
   - 其他节点：链表中其他的节点；

因此，一个存储{1,2,3}的完整链表结构如下图 所示：



单链表的创建和初始化

- 创建一个存储{1,2,3,4 }且无头节点的链表，C 语言实现代码如下：

```
linklist * initlinklist(){
    linklist * p=NULL;  //创建头指针
    linklist * temp = (linklist*)malloc(sizeof(linklist));//创建首元节点
    //首元节点先初始化
    temp->elem = 1;
    temp->next = NULL;
    p = temp;//头指针指向首元节点
    //从第二个节点开始创建
    for (int i=2; i<5; i++) {
        //创建一个新节点并初始化
        linklist *a=(linklist*)malloc(sizeof(linklist));
        a->elem=i;
        a->next=NULL;
        temp->next=a; //将temp节点与新建立的a节点建立逻辑关系
        temp=temp->next; //指针temp每次都指向新链表的最后一个节点，其实就是a节点，
这里写temp=a也对
    } return p; //返回建立的节点，只返回头指针 p即可，通过头指针即可找到整个链表
}
```

单链表的创建和初始化

- 创建一个存储 {1,2,3,4} 且含头节点的链表，则 C 语言实现代码为：

```
linklist * initlinklist(){
    linklist * p=(linklist*)malloc(sizeof(linklist));  //创建一个
头结点
    linklist * temp=p;  //声明一个指针指向头结点，
    //生成链表
    for (int i=1; i<5; i++) {
        linklist *a=(linklist*)malloc(sizeof(linklist));
        a->elem=i;
        a->next=NULL;
        temp->next=a;
        temp=temp->next;
    }
    return p;
}
```

4.列对（lineup）

➢ 题目描述

有很多小朋友，排成一个 n 行 m 列的方阵。第 i 行第 j 列的小朋友编号为 i*m + j（i 与 j 均从 1开始）。例如，n=4，m=6 的初始方阵编号如下：

7 8 9 10 11 12
13 14 15 16 17 18
19 20 21 22 23 24
25 26 27 28 29 30

我们可以对方阵下达“子方阵交换”指令，指令格式包含9个整数：$x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4, a$。
其中，

- x_1, y_1, x_2, y_2 表示一个子方阵：(x_1, y_1) 是子方阵的左上角坐标，(x_2, y_2) 是子方阵的右下角坐标
- 类似地，x_3, y_3, x_4, y_4 也表示一个子方阵。
- 上述两个子方阵长、宽分别相同，没有公共元素，也没有相邻元素（不允许横、竖相邻，但允许对角相邻）。
- a 表示行号（1≤a≤n）

该指令要求两个子方阵作为整体进行交换，并输出交换后方阵第 a 行的编号之和。

4.列对（lineup）

例如，在上面的初始方阵上，执行指令 1, 1, 2, 3, 3, 4, 4, 6, 2，交换两个子方阵后，将得到如下新方阵：

22 23 24 10 11 12
28 29 30 16 17 18
19 20 21 7 8 9
25 26 27 13 14 15

然后，我们需要输出第二行（a=2）的编号之和，即138

➢ 输入格式

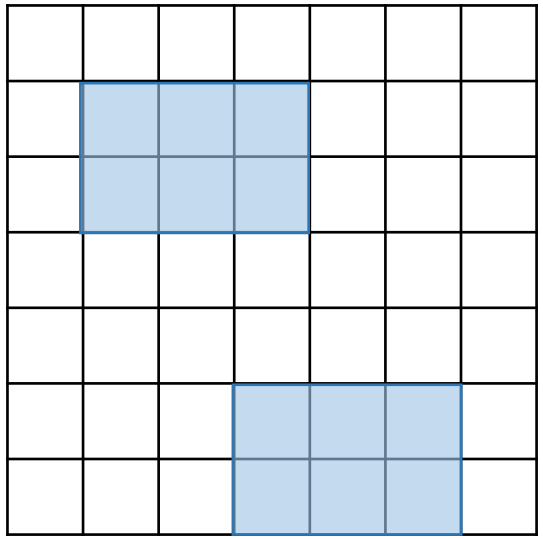
第一行：两个整数 n, m
第二行：一个整数 q，表示共会下达 q 次“子方阵交换”指令。注意，从初始方阵开始，这 q 个指令是连续作用的。
随后 q 行，每行 9 个数，代表一条“子方阵交换”指令：$x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4, a$

➢ 输出格式

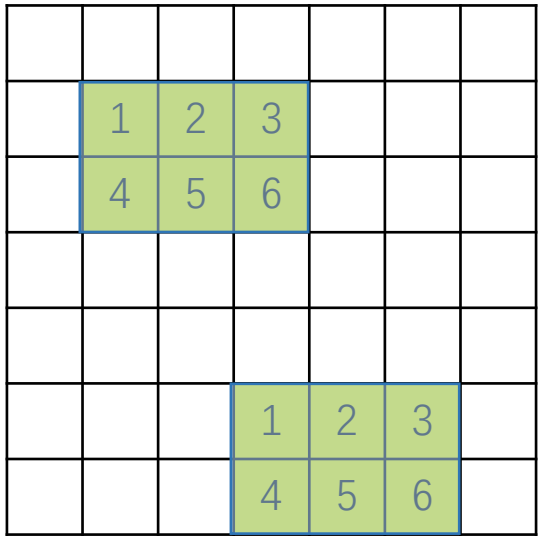
一共 q 行，每行一个数，表示需要计算的相应行中编号之和。
注意：这一加和需要使用 long long 来表示和输出。

4.列对（lineup）

示例

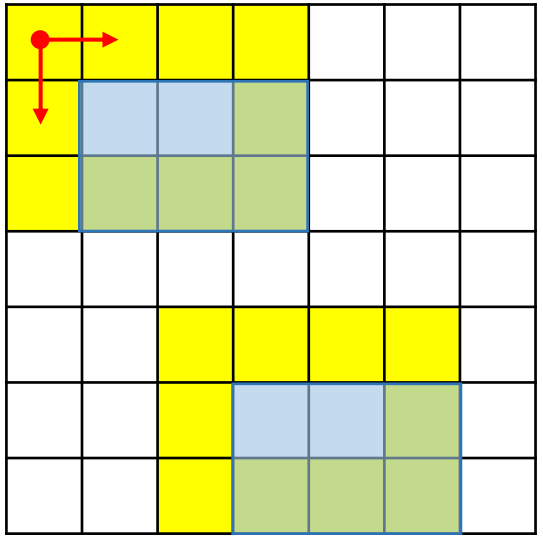


二维数组



$l * w$ 对元素

链表



$2 * (l + w)$ 对指针

```
struct member{
    int ID;
    struct member* right;
    struct member* down;
}members[2502][2502]
```

4.列对（lineup）

链表初始化：

```
struct member{
    int ID;
    struct member* right;
    struct member* down;
}members[2502][2502],*p1,*p2,*p3,*p4,*p5,*p6,*p7
,*p8,*temp1,*temp2;

void node_create(){
    for(int i=0;i<=n;i++){
        for(int j=0;j<=m;j++){
            members[i][j].ID=i*m+j;
            members[i][j].right=&members[i][j+1];
            members[i][j].down=&members[i+1][j];
        }
    }
}
```

给定行号之后求和：

```
void line_sum(){
    int a;long long int val=0;
    scanf("%d",&a);
    struct member* temp=&members[a][0];
    for(int i=0;i<m;i++){
        temp=temp->right;
        val+=temp->ID;
    }
    printf("%lld\n",val);
}
```

4.列对（lineup）

链表更新：

```
int a=x2-x1+1,b=y2-y1+1;
p1=&members[x1][0],p2=&members[0][y1];
p3=&members[x3][0],p4=&members[0][y3];
for(int i=1;i<y1;i++){
    p1=p1->right;}
for(int i=1;i<x1;i++){
    p2=p2->down;}
for(int i=1;i<y3;i++){
    p3=p3->right;}
for(int i=1;i<x3;i++){
    p4=p4->down;}
p5=p1,p6=p2,p7=p3,p8=p4;
for(int i=0;i<b;i++){
    p5=p5->right;
    p7=p7->right;}
for(int i=0;i<a;i++){
    p6=p6->down;
    p8=p8->down;
}
```

```
p5=p1,p6=p2,p7=p3,p8=p4;
for(int i=0;i<b;i++){
    p5=p5->right;
    p7=p7->right;}
for(int i=0;i<a;i++){
    p6=p6->down;
    p8=p8->down;}
for(int i=0;i<a;i++){
    temp1=p1->right;temp2=p5->right;
    p1->right=p3->right;
    p5->right=p7->right;
    p7->right=temp2;
    p3->right=temp1;
    p1=p1->down;
    p3=p3->down;
    p5=p5->down;
    p7=p7->down;
}
for(int i=0;i<b;i++){
//......}
```

4.列对（lineup）

- 二维数组的写法：

```
for (int i = 1; i <=q ; ++i) {
    int x1, y1, x2, y2, x3, y3, x4, y4,a;
    scanf("%d%d%d%d%d%d%d%d",&x1, &y1, &x2, &y2, &x3, &y3, &x4, &y4,&a);
    for (int j = x1,l=0; j <=x2 ; ++j,++l) {
        for (int k = y1,p=0; k <=y2 ; ++k,++p) {
            int mid=num[j][k];
            num[j][k]=num[x3+l][y3+p];
            num[x3+l][y3+p]=mid;
        }
    }
    long long count=0;
    for (int j = 1; j <=m ; ++j) {
            count+=num[a][j];
    }
    printf("%lld\n",count);
}
```

期末机考复习

周数	内容	知识点	备注	阅读材料
3 (0-intro)	Introducing C	课程简介、 演示开发环境、 `hello-world` 、 `Game: Guess the Number`	`Game: Guess the Number` 涉及到第 4 到 8 周的内容，目的是让学生对将要学习的内容有个初步认识	1.1、1.2；2.1、2.2、2.3
4 (1-types-io)	Variables, Types, I/O	Variables, Data Types; Operators, Expressions, Assignment Statements; `int` , `double` , `char` , `C string` ; `printf` , `scanf` ; `math.h` , `ctype.h`	本次课程不讲授 `float` 、 `unsigned` 等 C 语言中初学者容易犯错的知识点, 留到第 10 周	2.4–2.8；3.1–3.2；4.1–4.5；22.3；23.4、23.5
5 (2-if-for-array)	If, For, Array	`if` 语句、初步介绍 `for` 循环语句、一维数组	软件学院 2 个学时内讲不了 `switch/case` （5.3），可以安排学生自学	5.1–5.3；6.3；8.1
6 (3-for-a-while)	For, While, Do-While	更多 `for` 例子、 `while` 与 `do-while` 语句、 `break/continue`	请务必讲解 `selection sort` 与 `binary search` (这周与下周两周内)	6.1、6.2、6.4、6.5
7 (4-loops)	Loops; Multi-dimensional Arrays	More examples on loops; `break/continue`	建议讲解 Conway's Game of Life	8.2、8.3
8 (5-function)	Function; Scopes	函数的概念与使用; 作用域与程序结构	9.7 内容可选	9.1 – 9.5、10.1 – 10.5
9 (6-recursion)	Recursion	递归的概念与举例	建议介绍 `merge-sort` ; 本节内容不作高要求	9.6
10 (7-data-types)	Data Types	基本数据类型	介绍 Undefined Behaviors (最迟在此次课介绍)	7.1 – 7.6
11 (8-pointers-arrays)	Pointers	指针的基本概念，指针与一维数组，动态内存分配		11.1 – 11.5；12.1 – 12.3；17.1 – 17.4
12 (9-pointers-c-strings)	Pointers and C Strings	指针与字符串		12.4；13.1 – 13.6
13 (10-double-pointers)	Double Pointers	指针与字符串数组、命令行参数、指针与二维数组、函数指针		12.4；13.7；17.6、17.7
14 (11-struct)	Struct; Union; Enum	结构体、联合体、枚举类型	软件学院 16.4 节选读; 介绍 18.4 节内容 (如何解读声明语句)	16.1 – 16.5
15 (12-linkedlists)	Linked Lists			

机考的考点回顾

题目列表

#	名称	解决状态	通过人数
A	升序检测 (sorted)	数组和循环	90 / 100
B	二分查找 (bsearch)	数组、循环、二分查找算法（递归）、函数	80 / 92
C	五子棋 (gobang)	字符串、二维数组、双循环、函数	24 / 51
D	列队 (lineup)	结构体、链表、复杂情况	11 / 21

今年的两次机考

题目列表

#	名称	解决状态	通过人数
A	说的道理 (reverse.c)	✔ 已解决	178 / 189
B	三只小猪 (tictactoe.c)	✔ 已解决	161 / 215
C	□Jo娜贝尔 (Bye-JonaBell.c)	✔ 已解决	171 / 188

题目列表

#	名称	解决状态	通过人数
A	假新闻 (fake_news.c)	✔ 已解决	74 / 75
B	密码锁 (coded-lock.c)	✔ 已解决	89 / 119
C	漫游小狐狸 (WanderBell.c)	✔ 已解决	90 / 108

下次还填非常简单

说的道理(reverse.c)

➢ 题目描述

请将给定的字符串倒序输出，并将字符串中的小写字母与大写字母互转，如 5AKiYARy 👉 YrayIka5

➢ 输入格式

共两行，
第一行为一个数字 *len*，表示字符串的长度，保证所有数据的 len≤105 ；
第二行为一个长度为 *len* 的字符串，仅含有 26 个英文字母的大小写与 10 个数字，**无任何空白符**。

➢ 输出格式

共一行，为倒序的，互转了大小写的字符串。

测试样例

Input	Output
13 A0Miyu1SdDL24	42ldDs1UYIm0a

说的道理(reverse.c)

- 代码

```c
void reverseAndToggleCase(char *str)
{
    int length = strlen(str);
    char reversedString[length + 1];

    for (int i = 0; i < length; i++)
    {
        if (islower(str[i]))
        {
            reversedString[i] = toupper(str[i]);
        }
        else if (isupper(str[i]))
        {
            reversedString[i] = tolower(str[i]);
        }
        else
        {
            reversedString[i] = str[i];
        }
    }

    reversedString[length] = '\0';

    for (int i = length - 1; i >= 0; i--)
    {
        printf("%c", reversedString[i]);
    }
}
```

```c
char c;
/*...*/
if (c >= 'a' && c <= 'z') {
    // c是小写字母
}
```

判断是否大小写

- ctype.h
- ascii判断

大小写字母互转

- ctype.h
- ascii计算

三只小猪 (tictactoe.c)

➢ 题目描述

三只小猪在下 x 子棋，分别用数字1、2、4表示三只小猪的棋子，用数字0表示该位置上还没有棋子。
在$n \times n$ 的棋盘上，如果在横、竖、对角线三个方向上，有连续 x 个同种棋子连成一线，则这种棋子的“主猪”就获胜了。
1、2、4 在棋盘上的个数之间并没有关联，也不需要在意这个关联
他们的棋盘上可能出现多只猪都获胜或者没有猪获胜的情况，对于这种情况，你只需要输出draw表示平局，否则输出获胜的棋子编号即可

➢ 输入格式

第一行有两个整数 t 和 x，分别表示输入的棋盘个数和获胜需要的连续棋子个数，即题面中的x。
接下来有 t 组测试数据，每组测试数据的第一行是一个整数 n，表示棋盘大小。接下来有 n 行，每行有 n 个整数，用空格隔开。
我们对数据做出以下保证：
对于 100% 的数据，保证 t≤10；
有 50% 的数据，保证 x=3，n=3，最多只有一只猪获胜；
有 20% 的数据，保证 x=4，n=4，最多只有一只猪获胜；
对于剩余 30% 的数据，保证 $1 \leqslant x \leqslant n \leqslant 10$，可能有多只猪都获胜。

测试样例

Input	Output
2 3 3 0 1 0 2 1 4 2 1 2 3 2 0 4 1 2 4 0 1 2	1 2

➢ 输出格式

输出共 t 行，每行为获胜的棋子编号（1、2、4）或者draw。

三只小猪 (tictactoe.c)

- **0、1、2、4的特点**

都是2的n次方，0=0000 1=0001 2=0010 4=0100，所以可以用位运算

位运算-原理

- 计算机程序的整数都是以二进制形式存储在内存中

计算机中的负整数有三种表示方法，即原码、反码和补码。三种表示方法均有符号位和数值位两部分，符号位都是用0表示“正”，用1表示“负”，对负整数数值位来说三种表示方法各不相同。
原码：直接将数值位写为整数的绝对值对应的二进制。
反码：将原码的符号位不变，其他位依次按位取反
补码：反码+1得到补码。（严格的定义较为复杂，不在此详述）

如果A=60，那么在计算机中的存储方式(short类型)是：
A = 0 000 0000 0011 1100 2^2+2^3+2^4+2^5=4+8+16+32=60

位运算-原理

- 计算机程序的整数都是以二进制形式存储在内存中

如果A=-60，那么在计算机中的存储方式是：
原码：A = 1 000 0000 0011 1100
反码：A = 1 111 1111 1100 0011
补码：A = 1 111 1111 1100 0100
在计算机系统中，数值一律用补码来表示和存储。原因在于，使用补码，可以将符号位和数值域统一处理。加法和减法也可以统一处理（CPU只有加法器），当减去一个数的时候，计算机可以转化为加上这个数的补码。此外，补码与原码相互转换，其运算过程是相同的（按位取反，末位加一），不需要额外的硬件电路。
60+(-60)= 0 000 0000 0011 1100
+ 1 111 1111 1100 0100 = 0

位运算-案例

在知道二进制存储机制后，我们可以看一下C语言的位运算

运算符	描述
&	按位与运算符，按二进制位进行"与"运算。
|	按位或运算符，按二进制位进行"或"运算。
^	异或运算符，按二进制位进行"异或"运算。
~	取反运算符，按二进制位进行"取反"运算。
<<	二进制左移运算符。将一个运算对象的各二进制位全部左移若干位（左边的二进制位丢弃，右边补0）。
>>	二进制右移运算符。将一个数的各二进制位全部右移若干位，正数左补0，负数左补1，右边丢弃。

浮点数可以位运算吗？原理上可以，但没有意义，且会报错。

```
error: invalid operands to binary << (have 'float' and 'int')
```

位运算-案例

按位与操作(a=60, b=30)

```
#include <stdio.h>
void printBinary(short a) //输出short的二进制表达
{
    for (int i = 15; i >= 0; i--)
    {
        printf("%1d", a & 1 << i ? 1 : 0);
        if (i == 15 || i == 12 || i == 8 || i == 4)
            printf(" ");
    }
};

int main()
{
    short a = 60, b = 13;
    printBinary(a & b);
    return 0;
}
```

```
0 000 0000 0000 1100
```

基本原则：

0&0=0;
0&1=0;
1&0=0;
1&1=1;

a= 0 000 0000 0011 1100
b= 0 000 0000 0000 1101
a&b= 0 000 0000 0000 1100

位运算-案例

按位或操作

```
#include <stdio.h>
void printBinary(short a) //输出short的二进制表达
{
    for (int i = 15; i >= 0; i--)
    {
        printf("%1d", a & 1 << i ? 1 : 0);
        if (i == 15 || i == 12 || i == 8 || i == 4)
            printf(" ");
    }
    printf("\n");
};
int main()
{
    short a = 60, b = 13;
    printBinary(a | b);
    return 0;
}
```

```
0 000 0000 0011 1101
```

基本原则：

0|0=0;
0|1=1;
1|0=1;
1|1=1;

a=　　0 000 0000 0011 1100
b=　　0 000 0000 0000 1101
a|b=　0 000 0000 0011 1101

位运算-案例

异或操作 $a \oplus b = (\neg a \wedge b) \vee (a \wedge \neg b)$

```
#include <stdio.h>
void printBinary(short a) //输出short的二进制表达
{
    for (int i = 15; i >= 0; i--)
    {
        printf("%1d", a & 1 << i ? 1 : 0);
        if (i == 15 || i == 12 || i == 8 || i == 4)
            printf(" ");
    }
    printf("\n");
};
int main()
{
    short a = 60, b = 13;
    printBinary(a ^ b);
    return 0;
}
```

```
0 000 0000 0011 0001
```

基本原则：

0^0=0;
0^1=1;
1^0=1;
1^1=0;

a= 0 000 0000 0011 1100
b= 0 000 0000 0000 1101
a^b= 0 000 0000 0011 0001

位运算-案例

取反操作，按二进制位进行"取反"运算

```
#include <stdio.h>
void printBinary(short a) //输出short的二进制表达
{
    for (int i = 15; i >= 0; i--)
    {
        printf("%1d", a & 1 << i ? 1 : 0);
        if (i == 15 || i == 12 || i == 8 || i == 4)
            printf(" ");
    }
    printf("\n");
};
int main()
{
    short a = 60, b = 13;
    printBinary(~a);
    return 0;
}
```

```
1 111 1111 1100 0011
```

基本原则：

1->0

0->1

a=	0 000 0000 0011 1100
~a=	1 111 1111 1100 0011

位运算-案例

二进制左移运算符

```
#include <stdio.h>
void printBinary(short a) //输出short的二进制表达
{
    for (int i = 15; i >= 0; i--)
    {
        printf("%1d", a & 1 << i ? 1 : 0);
        if (i == 15 || i == 12 || i == 8 || i == 4)
            printf(" ");
    }
    printf("\n");
};
int main()
{
    short a = 60, b = 13;
    printBinary(a << 2);
    printBinary(-a << 2);
    return 0;
}
```

```
0 000 0000 1111 0000
1 111 1111 0001 0000
```

将一个运算对象的各二进制位全部左移若干位（左边的二进制位丢弃，右边补0）。

```
a=      0 000 0000 0011 1100=60
a<<2=   0 000 0000 1111 0000=240
-a=     1 111 1111 1100 0100=-60
-a<<2=1 111 1111 0001 0000=-240
```

位运算-案例

二进制右移运算符

```
#include <stdio.h>
void printBinary(short a) //输出short的二进制表达
{
    for (int i = 15; i >= 0; i--)
    {
        printf("%1d", a & 1 << i ? 1 : 0);
        if (i == 15 || i == 12 || i == 8 || i == 4)
            printf(" ");
    }
    printf("\n");
};
int main()
{
    short a = 60, b = 13;
    printBinary(a << 2);
    printBinary(-a << 2);
    return 0;
}
```

```
0 000 0000 0000 1111
1 111 1111 1111 0001
```

将一个数的各二进制位全部右移若干位，正数左补0，负数左补1，右边丢弃。

a=	0 000 0000 0011 1100=60
a>>2=	0 000 0000 0000 1111=15
-a=	1 111 1111 1100 0100=-60
-a>>2=	1 111 1111 1111 0001=-15

左移1位=乘2？右移1位=除2？

位运算-实际应用

交换变量：

```
int main(void)
{
    int x, y;
    // some codes .
    x ^= y ^= x ^= y;
    // 异或真神奇！不过这究竟是为什么呢？
    // some codes .
}
```

异或进行交换

```
int main(void)
{
    int x, y;
    // some codes .
    x = x ^ y;
    y = y ^ x; //可以看成在原来条件下的y= y^x^y
    x = x ^ y; //可以看成在原来条件下的下x=x^y^y^x^y
    // some codes .
}
```

拆开来看是这样的。
但这里有个陷阱
如果x和y指向同一个地址，那么三次异或会置0

位运算-实际应用

快速计算

```
#include <stdio.h>
void printBinary(short a) //输出short的二进制表达
{
    for (int i = 15; i >= 0; i--)
    {
        printf("%1d", a & 1 << i ? 1 : 0);
        if (i == 15 || i == 12 || i == 8 || i == 4)
            printf(" ");
    }
    printf("\n");
};
int main()
{
    short a = 4, b = 13;
    printBinary(a << 15);
    return 0;
}
```

```
0 000 0000 0000 0000
```

```
int main()
{
    short a = 4, b = 13;
    for (int i = 0; i < 15; i++)
        a *= 2;
    printf("%d", a); //输出 0
    return 0;
}
```

如果直接使用二进制的位运算进行编程，可以节约程序运行的时间，提升速度。

只有在移动后没有舍弃1时才在数值意义上成立（溢出问题）。

但这个操作和实际对a乘15次2的结果是一样的。

三只小猪 (tictactoe.c)

- 0、1、2、4的特点
- 检查行、列、对角线

```c
int checkWin(int board[][10], int n, int x)
{
    int result[4] = {0, 0, 0, 0};
    int flag = 0; // 不止赢一次
    // 检查行
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j <= n - x; j++)
        {
            int count = 1;
            int current = board[i][j];
            if (current == 0)
                continue;
            for (int k = 1; k < x; k++)
            {
                if (board[i][j + k] == current)
                {
                    count++;
                }
                else
                {
                    break;
                }
            }
            if (count == x)
            {
                result[current - 1] = 1;
            }
        }
    }
    // 检查列
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j <= n - x; j++)
        {
            int count = 1;
            int current = board[j][i];
            if (current == 0)
                continue;
            for (int k = 1; k < x; k++)
            {
                if (board[j + k][i] == current)
                {
                    count++;
                }
                else
                {
                    break;
                }
            }
            if (count == x)
            {
                result[current - 1] = 1;
            }
        }
    }
```

三只小猪 (tictactoe.c)

- 0、1、2、4的特点
- 检查行、列、对角线

```c
    // 检查主对角线
    for (int i = 0; i <= n - x; i++)
    {
        for (int j = 0; j <= n - x; j++)
        {
            int count = 1;
            int current = board[i][j];
            if (current == 0)
                continue;
            for (int k = 1; k < x; k++)
            {
                if (board[i + k][j + k] == current)
                {
                    count++;
                }
                else
                {
                    break;
                }
            }
            if (count == x)
            {
                result[current - 1] = 1;
            }
        }
    }

```

```c
    // 检查副对角线
    for (int i = 0; i <= n - x; i++)
    {
        for (int j = x - 1; j < n; j++)
        {
            int count = 1;
            int current = board[i][j];
            if (current == 0)
                continue;
            for (int k = 1; k < x; k++)
            {
                if (board[i + k][j - k] == current)
                {
                    count++;
                }
                else
                {
                    break;
                }
            }
            if (count == x)
            {
                result[current - 1] = 1;
                flag++;
            }
        }
    }
```

三只小猪 (tictactoe.c)

- 0、1、2、4的特点
- 检查行、列、对角线
- 没有那么聪明的办法

```
5       int result[4] = {0, 0, 0, 0};
6       int flag = 0; // 不止赢一次
7       // 检查行
```

位运算变成数组运算。消耗会更大

一头小猪可能会有多次胜利，所以原本flag自增的办法不行

```
115     flag = result[0] + result[1] + result[3];
116     if (flag == 0 || flag > 1)
117         return 0;
118     else
119     {
120         if (result[0] == 1)
121             return 1;
122         else if (result[1] == 1)
123             return 2;
124         else
125             return 4;
126     }
127     return 0;
128 }
129
```

三只小猪 (tictactoe.c)

- 0、1、2、4的特点
- 检查行、列、对角线
- 没有那么聪明的办法-数组
- 写一下输入输出

这里的输入输出基本就是模板了

```
131
132 int main()
133 {
134     int t, x;
135     scanf("%d %d", &t, &x);
136
137     while (t--)
138     {
139         int n;
140         scanf("%d", &n);
141
142         int board[10][10];
143         for (int i = 0; i < n; i++)
144         {
145             for (int j = 0; j < n; j++)
146             {
147                 scanf("%d", &board[i][j]);
148             }
149         }
150
151         int result = checkWin(board, n, x);
152         if (result == 0)
153         {
154             printf("draw\n");
155         }
156         else
157         {
158             printf("%d\n", result);
159         }
160     }
161
162     return 0;
163 }
```

Jo娜贝尔 (Bye-JonaBell.c)

➢ 题目描述

约瑟夫问题变形，每个JonaBell可以被射击l次，共n只，报数到k进行一次射击，水枪初始满弹，子弹打空后下一次射击改为装填，最大容量为b

➢ 输入格式

一行，四个整数，依次为：

1. JonaBell 的数目 n ；
2. 会被水枪射击的 JonaBell 的报数 k ；
3. 水枪的最大装填数 b ；
4. 每只 JonaBell 的尾巴数目 l。

➢ 输出格式

共一行，一个整数，为不会和大家 say goodbye 的 JonaBell 的编号。

测试样例

Input	Output
6 5 6 1	1

搞不好期末考试玲娜贝儿
还得被拉出来一次
大家多看看

Jo娜贝尔 (Bye-JonaBell.c)

- 链表刚刚已经讲了
- 用了一些看起来很像循环队列的东西

构造数组，每次遍历对n取模，实现循环效果。
生命为0的元素挪到队尾（偷懒直接删了，如果输出顺序的话不能删）

```c
int findLastJonaBell(int n, int k, int b, int l)
{
    int queue[MAX_SIZE][2];
    int bullet = b;
    int query = 0;
    int count = 1;
    // 初始化队列
    for (int i = 0; i < n; i++)
    {
        queue[i][0] = i + 1;
        queue[i][1] = l;
    }

    while (n > 1)
    {
        if (count == k)
        {
            count = 1;
            if (bullet > 0)
            {
                bullet--;
                queue[query][1]--;
                // 将queue[query][1]=0的元素挪到队尾
                if (queue[query][1] == 0)
                {
                    for (int i = query; i < n - 1; i++)
                    {
                        queue[i][0] = queue[i + 1][0];
                        queue[i][1] = queue[i + 1][1];
                    }
                    n--;
                }
                else
                {
                    query = (query + 1) % n;
                }
            }
            else
            {
                bullet = b;
                query = (query + 1) % n;
            }
        }
        else
        {
            query = (query + 1) % n;
            count++;
        }
    }
    return queue[0][0];
```

假新闻 (fake_news.c)

➢ 题目描述

输入为一句话，单词首字母大写且没有空格

➢ 输入格式

输入仅一行，为一个由多个单词组成的句子，不包含任何标点符号、空格和换行符

- 满足条件：每个单词的首字母大写，其余全部小写，长度小于 1024 字节

➢ 输出格式

输出仅一行，为一个句首字母大写，其余全小写，并用单个空格分隔各个单词的句子。

测试样例

Input	Output
TrumpIsMakingAmericaGreatAgain	Trump is making america great again

假新闻 (fake_news.c)

- 先加空格

  用memmove后移，添加空格

```
 5 void formatSentence(char *sentence)
 6 {
 7     int length = strlen(sentence);
 8     int start = 0;
 9     int end = 0;
10 
11     // 添加空格
12     for (int i = 1; i < length; i++)
13     {
14         if (isupper(sentence[i]) && !isspace(sentence[i - 1]))
15         {
16             memmove(sentence + i + 1, sentence + i, length - i);
17             sentence[i] = ' ';
18             length++;
19             i++;
20         }
21     }
```

假新闻 (fake_news.c)

- 先加空格
- 再把除了第一个大写字母之外的字母转换为小写字母

最后一位处理了一下

```
23      // 将除了第一个大写字母之外的字母转换为小写字母
24      for (int i = 1; i < length; i++)
25      {
26          sentence[i] = tolower(sentence[i]);
27      }
28      sentence[length] = '\0';
29  }
```

```
31  int main()
32  {
33      char sentence[2048];
34      fgets(sentence, sizeof(sentence), stdin);
35
36      formatSentence(sentence);
37
38      printf("%s\n", sentence);
39
40      return 0;
41  }
```

密码锁 (coded-lock.c)

➢ 题目描述

滚轮数字密码锁，每位数字都是一个独立的数字轮盘，范围是连续的 1~*N*，每位数字均可向前/向后拨动。朝向正前方的数字串即为当前密码。

对于给定的密码锁和密码串，该密码串能否完成“对称打乱”？
如果可以，**最少**需要多少次**拨动操作**？

Fradow 已设置好密码，热情的 Sakiyary 却提议按如下规则进行“对称打乱”（见样例解释）：

(1) 指定一个正整数 D。

(2) 定义**拨动操作**：对某位数字，一次性向前拨 D 个数字或者向后拨 D 个数字。

(3) 对于每位数字，可不限次数执行上述拨动操作。

(4) 目的是，使密码锁上的数字串呈左右对称（即“回文串”）。此时便称完成了“对称打乱”。

(5) 如果密码本身是对称的，也视作完成了“对称打乱”。

测试样例

Input	Output
5 8 3 1 2 8 8 4	3
2 4 2 1 4	Impossible

➢ 输入格式

输入共两行：
第一行为三个整数 *W*, *N*, *D*，含义如上。
第二行为 *W* 个整数，表示 设置的密码。
　　注意：对于 *W* 个整数，均在 [1,*N*] 范围中。

➢ 输出格式

若可以“对称打乱”，则输出一个整数，表示所需的最小拨动操作次数。
若无法“对称打乱”，则输出 Impossible。

密码锁 (coded-lock.c)

- 卡题不要紧，助教提交了108次，仅有2次通过（一己之力拉低AC率）。

正确实现前述的数学运算。你将获得 80% 的分数。	答案错误	95	171ms	732KiB	C	12-18 16:44
正确实现将获得 100% 的分数。	答案错误	85	496ms	720KiB	C	12-18 20:53

```c
int minOperations(int *password, int length, int N, int D)
{
    int operations = 0;

    for (int i = 0; i < length / 2; i++)
    {
        int diff = abs(password[i] - password[length - i - 1]);
        if (diff == 0)
        {
            continue;
        }
        else
        {
```

```c
#define INTMAX 2147483646
        {
            int clockwise = diff;
            clockwise = findMultiple(clockwise, N, D);
            int counterclockwise = abs(N - diff);
            counterclockwise = findMultiple(counterclockwise, N, D);
            if (diff != 0 && clockwise == INTMAX && counterclockwise ==
                INTMAX)
                return 0;
            operations += clockwise < counterclockwise ? clockwise :
                counterclockwise;
        }
    }
    return operations;
}
```

密码锁 (coded-lock.c)

- 卡题不要紧，助教提交了108次，仅有2次通过（一己之力拉低AC率）。

正确实现前述的数学运算。你将获得 80% 的分数。					
答案错误	95	171ms	732KiB	C	12-18 16:44
正确实现将获得 100% 的分数。					
答案错误	85	496ms	720KiB	C	12-18 20:53

- 一些错误代码

```
16 int findMultiple(int diff, int N, int D)
17 {
18     int multiple = 0;
19     long long result = 0;
20
21     while (true)
22     {
23         result = diff + N * multiple;
24         if (result % D == 0)
25         {
26             return result / D;
27         }
28         else if (multiple > 51451444)
29         {
30             break;
31         }
32         multiple++;
33     }
34
35     return INTMAX;
36 }
```

diff：对位元素间的差值（正转或者反转）
N：数据范围1~N
D：拨动的步长

为什么95分至今仍是一个不解之谜。

密码锁 (coded-lock.c)

- 卡题不要紧，助教提交了108次，仅有2次通过（一己之力拉低AC率）。

正确实现前述的数学运算。你将获得 80% 的分数。	答案错误	95	171ms	732KiB	C	12-18 16:44
正确实现将获得 100% 的分数。	答案错误	85	496ms	720KiB	C	12-18 20:53

- 正确代码

```
16 int findMultiple(int diff, int N, int D)
17 {
18     int multiple = 0;
19     long long result = diff;
20
21     while (true)
22     {
23         result = result + D;
24         multiple++;
25         if (result % N == 0)
26         {
27             return multiple;
28         }
29         else if (multiple > 1145140)
30         {
31             break;
32         }
33         result %= N;
34     }
35
36     return INTMAX;
37 }
```

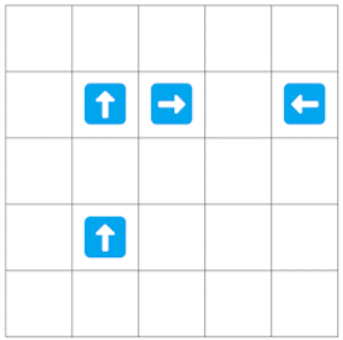
diff：对位元素间的差值（正转或者反转）
N：数据范围1~N
D：拨动的步长

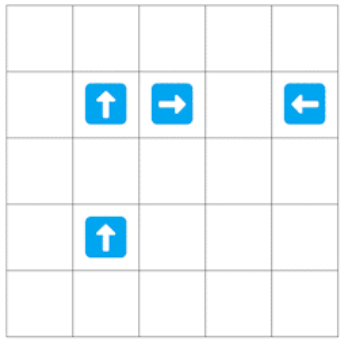
用模运算实现循环。
这里的break判断原因：
N最大为100000，进行N次计算后，result必然出现了重复（鸽巢原理）

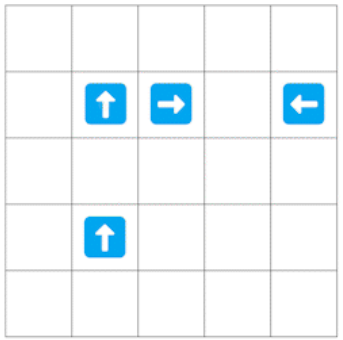
（判断方法不唯一）

漫游小狐狸 (WanderBell.c)

➢ 题目描述







•对于 20% 的数据，*WanderBell* 无法被救出。
请不要拿0分
•对于 40% 的数据，只需检查每一个箭头相邻位置的箭头情况
实在没办法可以这么干

➢ 输入格式

输入 R+1 行：
第一行包含 2 个整数 R,C 表示网格的行数和列数；接下来的 R 行，每行 C 个字符，表示网格；其中，.
表示网格空白，L, R, U, D 分别表示网格中的向左、向右、向上和向下的箭头。

➢ 输出格式

输出一行：
包含一个整数，表示需要改变方向的箭头的最小数目；若 ant-hengxin 无法救出 WanderBell，输出 −1。

漫游小狐狸 (WanderBell.c)

➢ 复杂到给了简化思路，就很简单了

> 考虑某个带箭头的单元格 U，假设它指向右👉。那就**查看它右边的所有单元格，如果这些单元格中都没有箭头**【我们称满足这种条件的箭头单元格为**危险的单元格**】，小狐狸就危险了。*ant-hengxin* 必须改变 U 中的箭头方向。否则，如果小狐狸被放在了 U 上，它就会走出网格边缘。
>
> 问题是，如何改变 U 中的箭头方向？
>
> 答案也很简单：看看 U 的上、下、左、右某个方向上是否有带箭头的单元格。如果有，则将 U 指向它（某一个即可）【我们称这一步为**化险为夷大法**】。当然，如果都没有，*ant-hengxin* 也就**回天乏术**了。
>
> 为什么这样就可以了呢？
>
> 因为，每个箭头单元格都这样处理之后，就不存在**危险单元格**了。

好了，现在你知道该怎么做了吗？

- 去找到那些**危险单元格**吧。
- 对每个**危险单元格**，执行**化险为夷大法**或者发出**回天乏术**之叹。

或者，等价地，按下面的描述来做：

考虑每一个箭头单元格：

- 如果该箭头所在格子的上、下、左、右**四个**方向上都**没有箭头**，则无论 *ant-hengxin* 如何调整箭头方向都无济于事（小狐狸哭泣~）
- 如果该箭头所在格子的上、下、左、右**某个**方向上**有箭头**，记为 A，则调整当前箭头的方向，使其指向 A。

漫游小狐狸 (WanderBell.c)

- 复杂到给了简化思路，就很简单了

上面的文字最多看一遍就够了

考虑每一个箭头单元格：

- 如果该箭头所在格子的上、下、左、右**四个**方向上都**没有箭头**，则无论 *ant-hengxin* 如何调整箭头方向都无济于事（小狐狸哭泣~）
- 如果该箭头所在格子的上、下、左、右**某个**方向上**有箭头**，记为 A，则调整当前箭头的方向，使其指向 A。

```
48 int check(int i, int j)
49 {
50
51     if (grid[i][j] == 'U')
52     {
53         for (int k = 1; i - k >= 0; k++)
54         {
55             if (grid[i - k][j] != '.')
56                 return 0;
57         }
58     }

       ......

84     return modify(i, j);
```

```
 9 char dir[4] = {'U', 'R', 'D', 'L'};
11 int modify(int i, int j)
12 {
13     for (int k = 1; i - k >= 0; k++)
14     {
15         if (grid[i - k][j] != '.')
16         {
17             grid[i][j] = dir[0];
18             return 1;
19         }
20     }

   ......

45     flag = 1;
46     return 0;
47 }
```

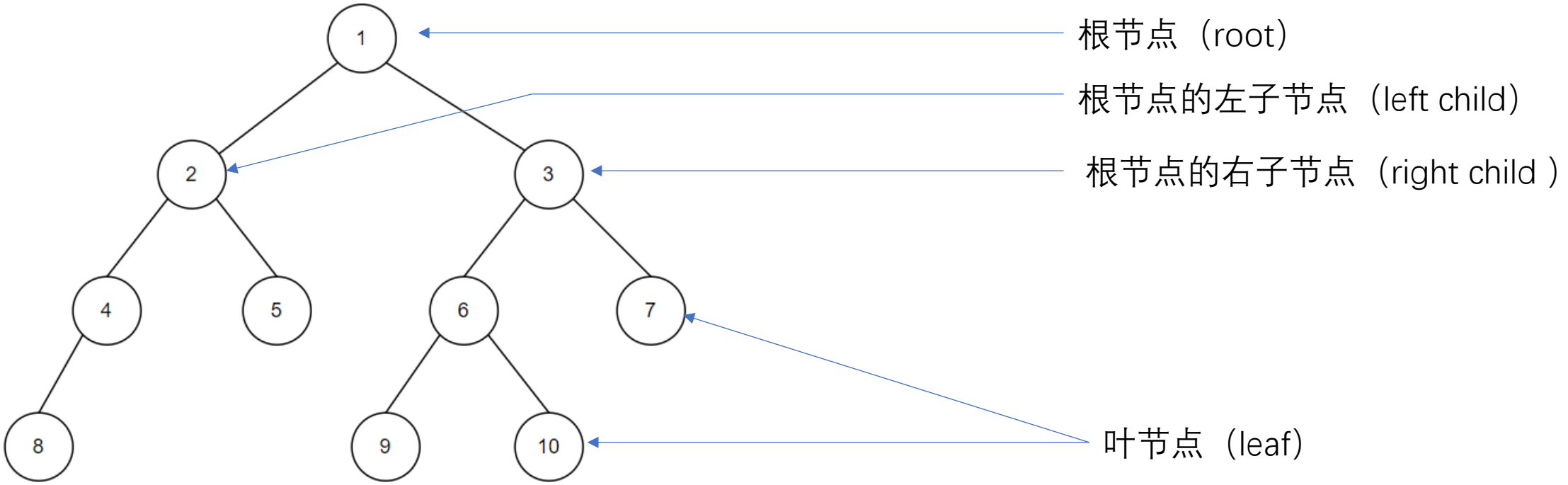
二叉树-定义

- 定义
- 一种数据结构
- 由节点（Node）组成，这些节点通过边（Edge）连接在一起
- 每个节点最多有两个子节点，分别称为左子节点和右子节点
- 结构是树状的，因为它呈现出分层次的层级结构，最上面的节点被称为根节点

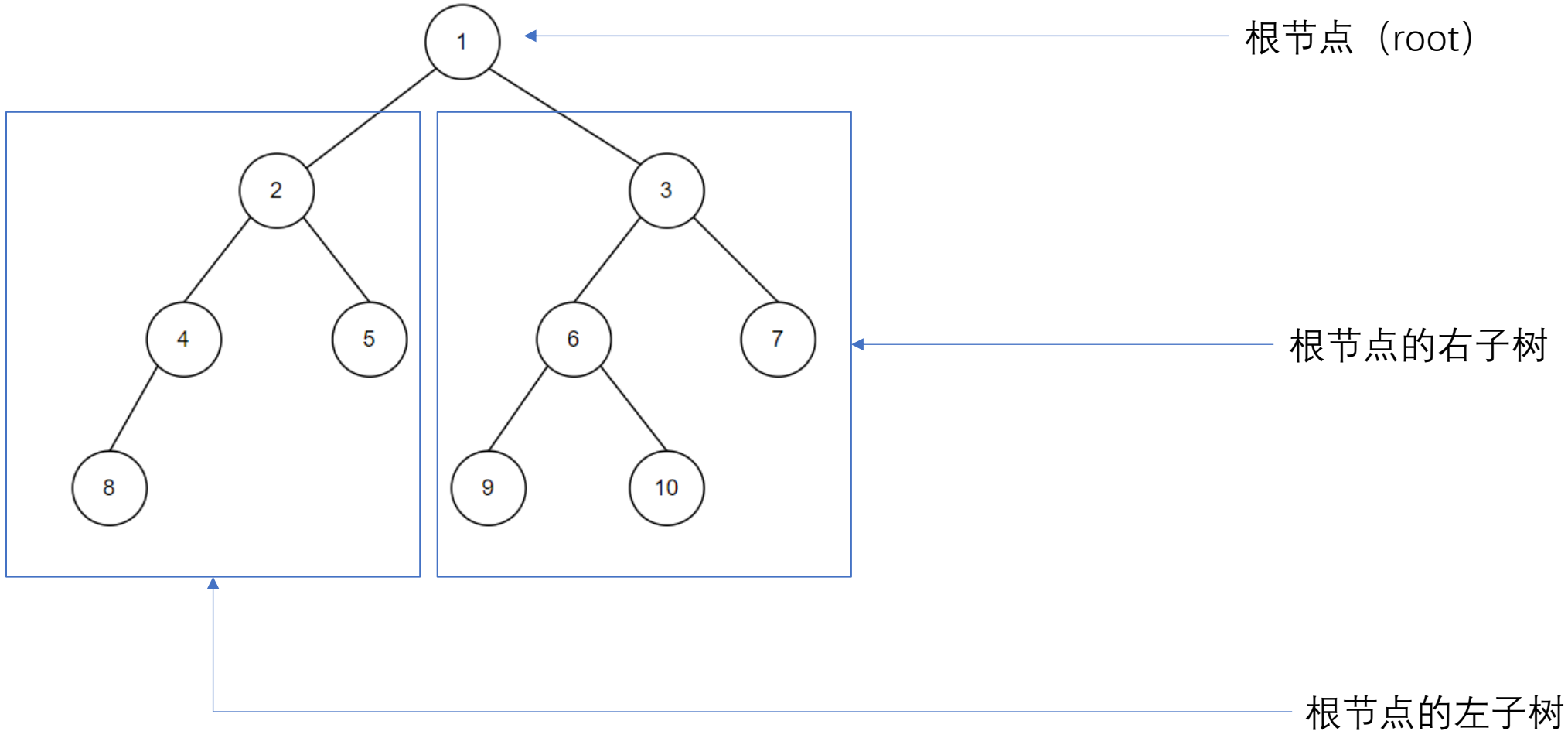
- 主要特征

1.根节点（root）：树的顶部节点，是树的起点
2. 父节点（parent）和子节点（child）：每个节点可以有零个、一个或两个子节点，分别称为左子节点和右子节点
3. 叶节点（leaf）：没有子节点的节点称为叶节点，它们位于树的底部
4. 深度（depth）：从根节点到某个节点的唯一路径的长度称为该节点的深度
5. 高度（height）：树中任意节点的深度的最大值称为树的高度
6. 子树（subtree）：节点及其所有后代节点（包括该节点本身）形成的树称为子树

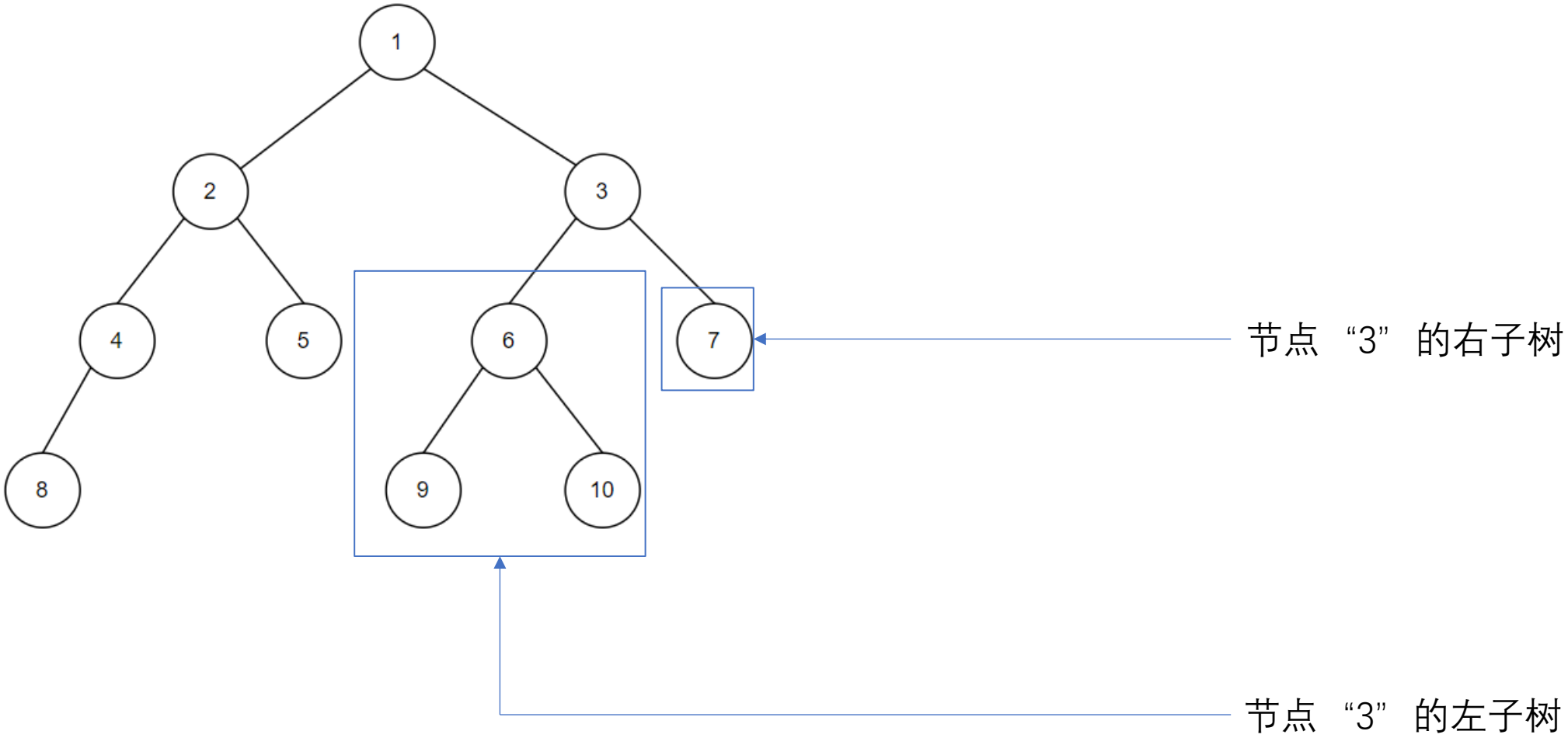
二叉树-定义


1
2
3
4
5
6
7
8
9
10
根节点（root）
根节点的左子节点（left child）
根节点的右子节点（right child ）
叶节点（leaf）

二叉树-定义



二叉树-定义



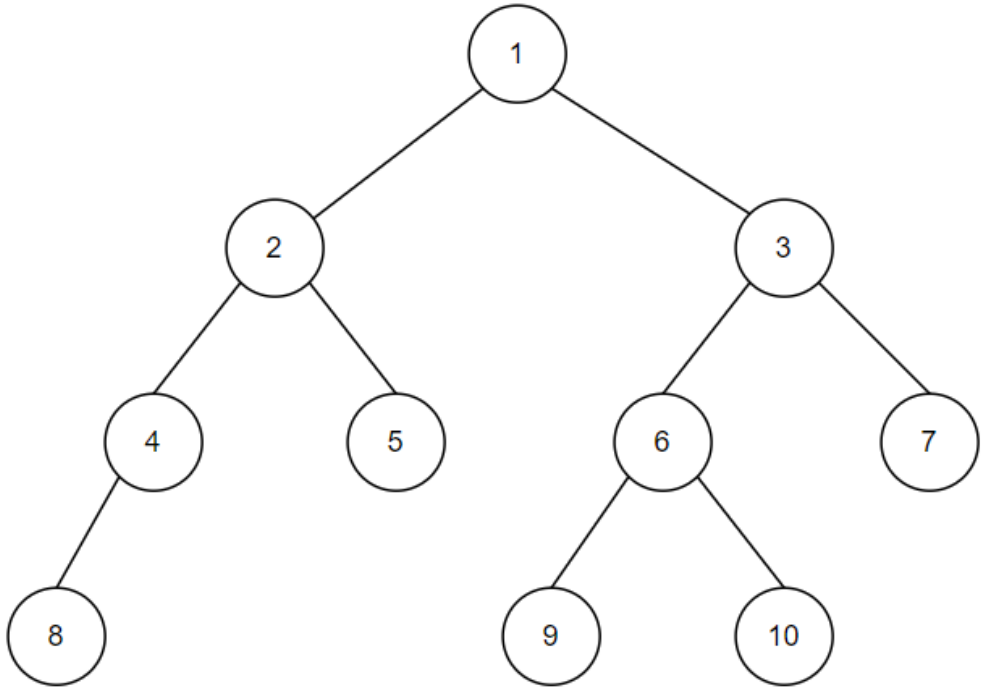
二叉树-定义

使用C语言定义二叉树

```c
struct TreeNode {
    int data;
    struct TreeNode* left;
    struct TreeNode* right;
};
```

```c
struct TreeNode* createNode(int value) {
    struct TreeNode* newNode = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    newNode->data = value;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
```

```c
int main() {
    struct TreeNode* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);
    root->right->left = createNode(6);
    root->right->right = createNode(7);
    root->left->left->left = createNode(8);
    root->right->left->left = createNode(9);
    root->right->left->right = createNode(10);
    return 0;
}
```

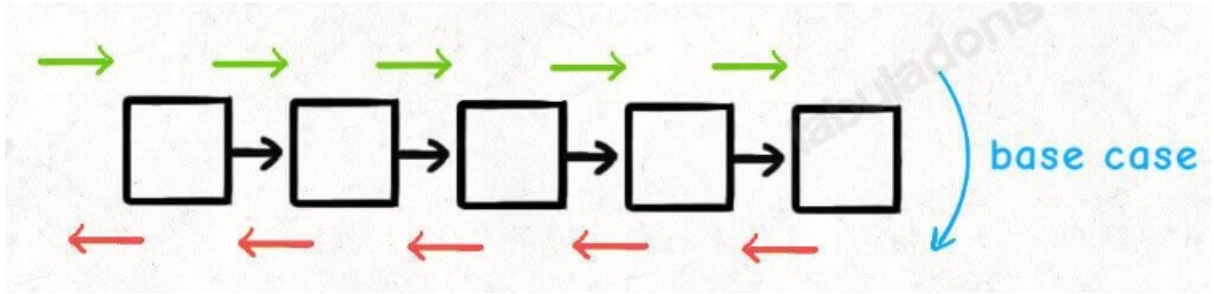


二叉树-如何遍历二叉树

➢ 如何遍历二叉树

- 单链表和数组的遍历可以是迭代的，也可以是递归的
- 如何递归地遍历链表？

```
struct Node {
    int data;          // 存储节点数据
    struct Node* next;  // 指向下一个节点的指针
};
```




```
// 递归前序打印链表
void printListPreorder(struct Node* cur) {
    if (cur == NULL) { return;} // base case
        printf("%d ", cur->data);
        printListPreorder(cur ->next);
}
```

```
// 递归后序打印链表
void printListPostorder(struct Node* cur) {
    if (cur == NULL) { return;} } // base case
        printListPostorder(cur ->next);
        printf("%d ", cur ->data);
}
```

二叉树-如何遍历二叉树

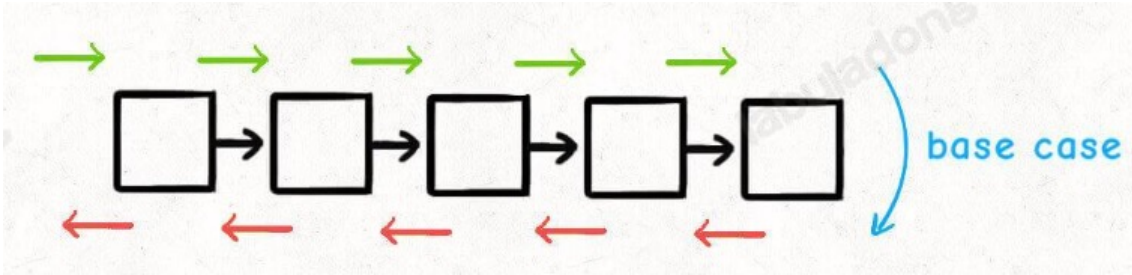
- 如何遍历二叉树

- 单链表和数组的遍历可以是迭代的，也可以是递归的
- 如何递归地遍历链表？

- 只要是递归形式的遍历，都可以有前序位置和后序位置，分别在递归之前和递归之后。

- **前序位置，就是刚进入一个节点（元素）的时候，后序位置就是即将离开一个节点（元素）的时候**，那么进一步，你把代码写在不同位置，代码执行的时机也不同，比如通过后序遍历来倒序打印链表

- 二叉树无非就是一个“二叉”链表
  - 遍历左子树
  - 遍历右子树



二叉树-如何遍历二叉树

- 如何遍历二叉树

深度优先搜索（DFS）
深度优先搜索是一种递归的遍历方式，它沿着树的深度尽可能远地访问节点，然后再回溯到上一层，继续探索其他分支。DFS有三种常见的方式：前序遍历、中序遍历和后序遍历。这三种方式各有其应用场景，可以用于解决不同类型的问题。

- 前序遍历：先访问根节点，然后递归地访问左子树和右子树。
- 中序遍历：先递归地访问左子树，然后访问根节点，最后递归地访问右子树。
- 后序遍历：先递归地访问左子树和右子树，最后访问根节点。

广度优先搜索（BFS）
广度优先搜索是一种按层次遍历的方式，从根节点开始，逐层访问节点，直到遍历完整个树。BFS通常使用队列来实现，确保每一层的节点按顺序被访问。

二叉树-DFS

- 深度优先搜索（DFS）
  - 前序遍历：先访问根节点，然后递归地访问左子树和右子树。

```
void preOrderTraversal(struct TreeNode* root) {
    if (root == NULL) { // base case
        return;
    }
    // 1. 访问当前节点
    printf("%d ", root->data);
    // 2. 递归遍历左子树
    preOrderTraversal(root->left);
    // 3. 递归遍历右子树
    preOrderTraversal(root->right);
}
```

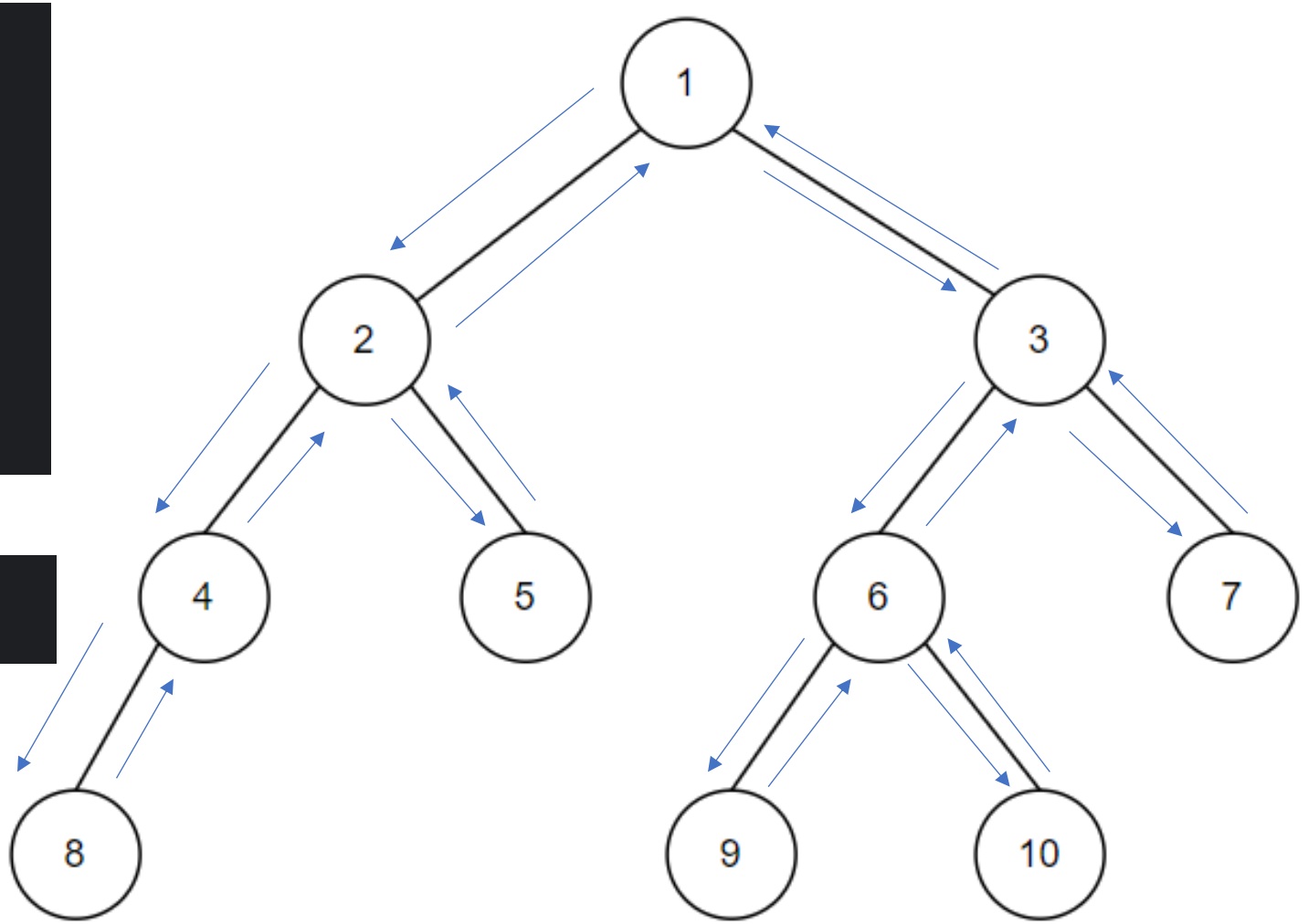
```
int main() {
    struct TreeNode* root = createNode(1);
    // create other nodes
    preOrderTraversal(root);
    return 0;
}
```

二叉树-前序遍历

- 前序遍历

```
void preOrderTraversal(struct TreeNode* root) {
    if (root == NULL) { // base case
        return;
    }
    // 1. 访问当前节点
    printf("%d ", root->data);
    // 2. 递归遍历左子树
    preOrderTraversal(root->left);
    // 3. 递归遍历右子树
    preOrderTraversal(root->right);
}
```

```
1 2 4 8 5 3 6 9 10 7

Process finished with exit code 0
```



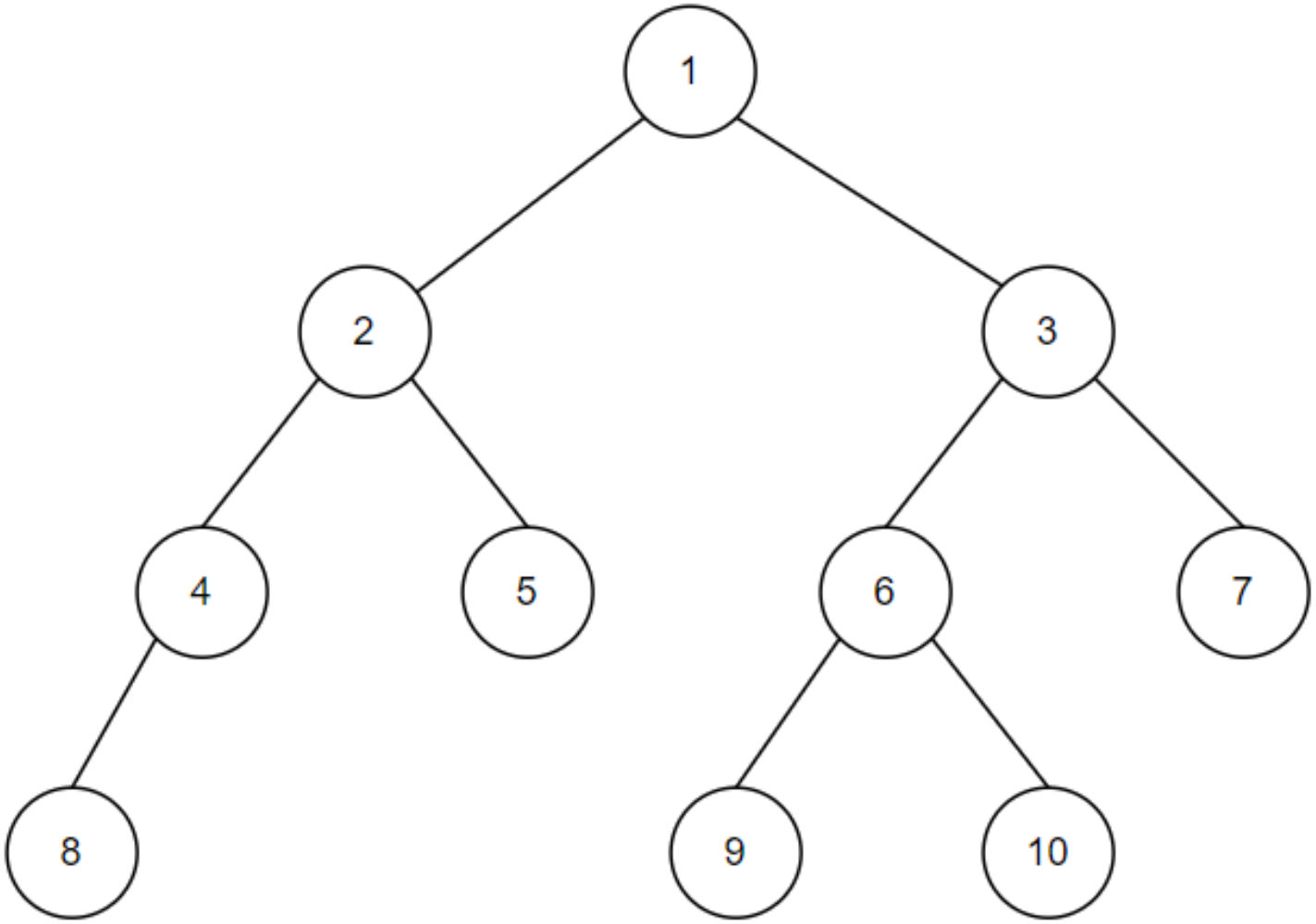
二叉树-中序遍历和后序遍历

```
void inOrderTraversal(struct TreeNode*
root) {
    if (root == NULL) {return;}
        inOrderTraversal(root->left);
        printf("%d ", root->data);
        inOrderTraversal(root->right);
}
```

8 4 2 5 1 9 6 10 3 7

```
void postOrderTraversal(struct
TreeNode* root) {
    if (root == NULL) {return;}
        postOrderTraversal(root->left);
        postOrderTraversal(root->right);
        printf("%d ", root->data);
}
```

8 4 5 2 9 10 6 7 3 1



二叉树-求最大深度

➢ 思路1：遍历思路

- 遍历一遍二叉树，用一个外部变量记录每个节点所在的深度，取最大值就可以得到最大深度
- 前序遍历

```
// 记录最大深度
int res = 0;
// 记录遍历到的节点的深度
int depth = 0;
// 主函数
int maxDepth(TreeNode root) {
    traverse(root);
    return res;
}
```

```
void traverse(TreeNode root) {
  if (root == null) {
  return;
  }
  // 前序位置
  depth++;
  if (root.left == null && root.right == null) {
        // 到达叶子节点，更新最大深度
  res = Math.max(res, depth);
  }
  traverse(root.left);
  traverse(root.right);
  // 后序位置
  depth--;

}
```

二叉树-求最大深度

➢ 思路2：分解思路

- 容易发现，一棵二叉树的最大深度可以通过子树的最大深度推导出来
- 整棵树的最大深度等于左右子树的最大深度取最大值，然后加上根节点自己的深度
- 后序遍历
- 一棵二叉树的前序遍历结果 = 根节点 + 左子树的前序遍历结果 + 右子树的前序遍历结果
- 前序位置的代码只能从函数参数中获取父节点传递来的数据，而后序位置的代码不仅可以获取参数数据，还可以获取到子树通过函数返回值传递回来的数据

```java
// 定义：输入根节点，返回这棵二叉树的最大深度
int maxDepth(TreeNode root) {
    if (root == null) {
        return 0;
    }
    // 利用定义，计算左右子树的最大深度
    int leftMax = maxDepth(root.left);
    int rightMax = maxDepth(root.right);
    // 整棵树的最大深度等于左右子树的最大深度取最大值，
    // 然后再加上根节点自己
    int res = Math.max(leftMax, rightMax) + 1;
    return res;
}
```

二叉树-求最大深度

➢ 思路2：分解思路

- 一棵二叉树的前序遍历结果 = 根节点 + 左子树的前序遍历结果 + 右子树的前序遍历结果

- 前序位置的代码只能从函数参数中获取父节点传递来的数据，而后序位置的代码不仅可以获取参数数据，还可以获取到子树通过函数返回值传递回来的数据

- 现在给你一棵二叉树，两个简单的问题：
  1、如果把根节点看做第 1 层，如何打印出每一个节点所在的层数？
  2、如何打印出每个节点的左右子树各有多少节点？

二叉树-前序遍历和后序遍历

- 现在给你一棵二叉树，两个简单的问题：
  1、如果把根节点看做第 1 层，如何打印出每一个节点所在的层数？
  2、如何打印出每个节点的左右子树各有多少节点？

```
void traverse(TreeNode root, int level) {
    if (root == null) {
        return;
    }
    printf("node %s, level %d 层", root, level);
    traverse(root.left, level + 1);
    traverse(root.right, level + 1);
}
// main
traverse(root, 1);
```

```
// 输入一棵二叉树，返回这棵二叉树的节点总数
int count(TreeNode root) {
    if (root == null) {
        return 0;
    }
    int leftCount = count(root.left);
    int rightCount = count(root.right);
    // 后序位置
    printf("节点 %s 的左子树有 %d 个节点，右子树有 %d 个节点",
            root, leftCount, rightCount);
    return leftCount + rightCount + 1;
}
```

二叉树-前序遍历和后序遍历

- 现在给你一棵二叉树，两个简单的问题：
  1、如果把根节点看做第 1 层，如何打印出每一个节点所在的层数？
  2、如何打印出每个节点的左右子树各有多少节点？

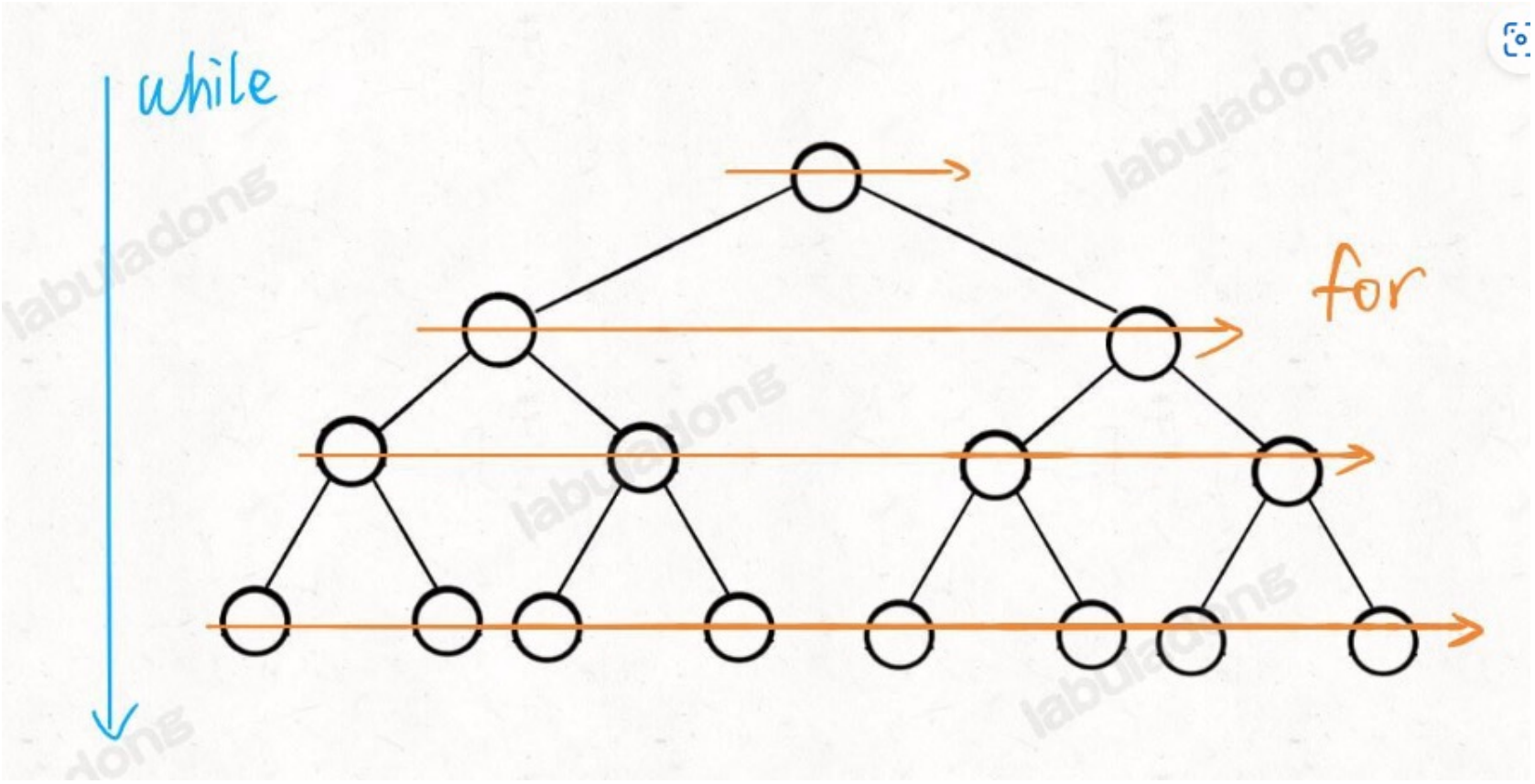
- 这两个问题的根本区别在于：一个节点在第几层，你从根节点遍历过来的过程就能顺带记录，用递归函数的参数就能传递下去；而以一个节点为根的整棵子树有多少个节点，你需要遍历完子树之后才能数清楚，然后通过递归函数的返回值拿到答案。

- 换句话说，一旦你发现题目和子树有关，那大概率要给函数设置合理的定义和返回值，在后序位置写代码了

二叉树-遍历二叉树

广度优先搜索（BFS）

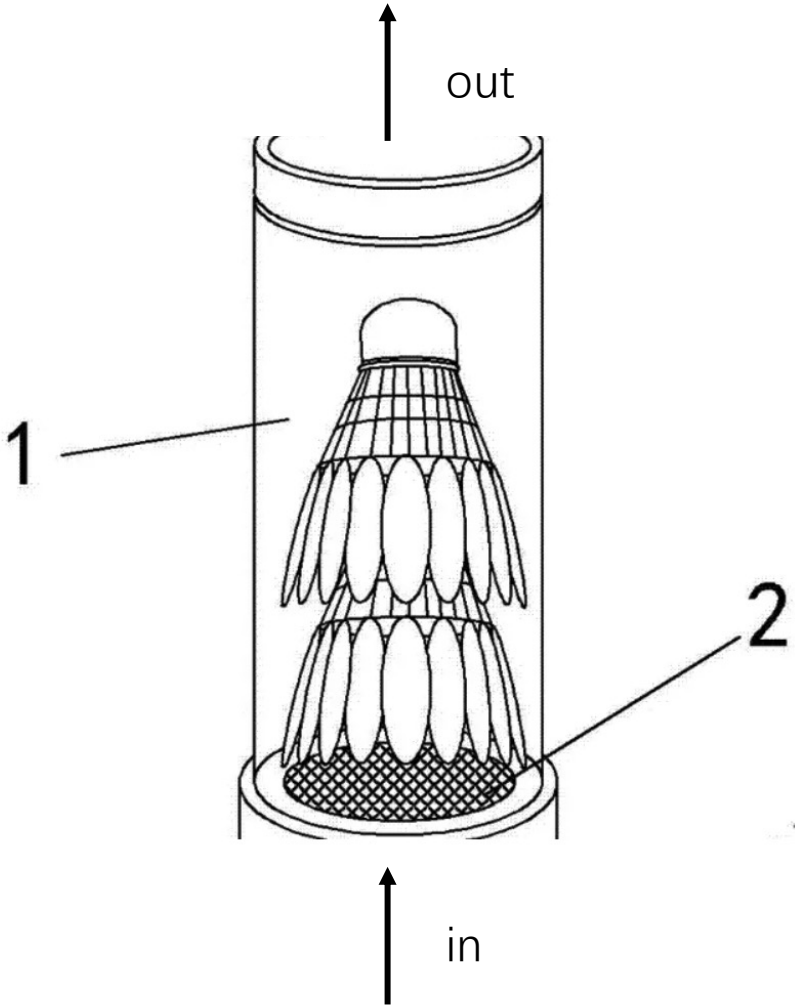
广度优先搜索是一种按层次遍历的方式，从根节点开始，逐层访问节点，直到遍历完整个树。BFS通常使用队列来实现，确保每一层的节点按顺序被访问。



二叉树-BFS-队列

➢ 队列

- 一种常见的数据结构
- 它按照先进先出（First In First Out，FIFO）的原则管理元素
- 最先被添加到队列的元素将首先被移除
- 常常被用于需要按照顺序处理元素的情境
- 常和栈（后进先出）做类比



二叉树-BFS-队列

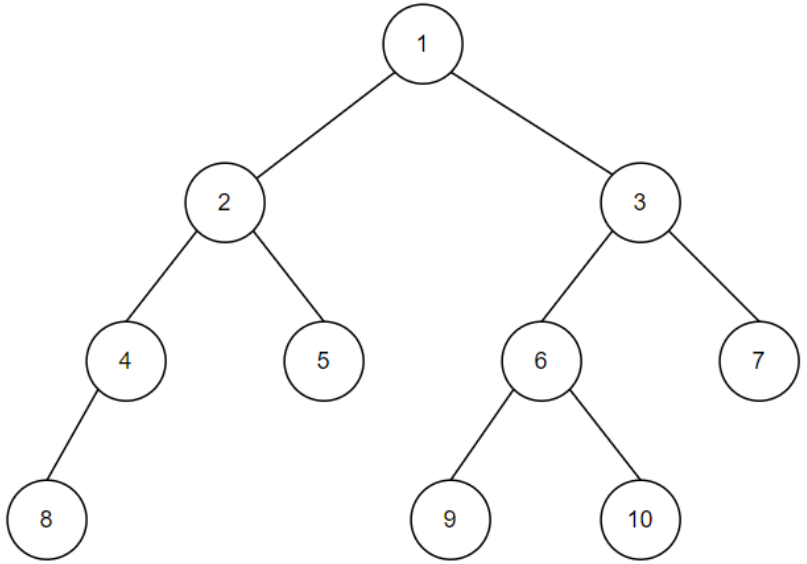
```c
#define MAX_SIZE 100

typedef struct {
    int array[MAX_SIZE];
    int front;
    int rear;
} Queue;
```

```c
// 入队操作
void enqueue(Queue *queue, int value) {
    if (isFull(queue)) {
        printf("Queue is full. Cannot enqueue.\n");
        return;
    }
    if (isEmpty(queue)) {
        queue->front = 0;
    }
    queue->rear = (queue->rear + 1) % MAX_SIZE;
    queue->array[queue->rear] = value;
}
```

```c
// 出队操作
void dequeue(Queue *queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty. Cannot dequeue.\n");
        return;
    }

    if (queue->front == queue->rear) {
        // 队列中只有一个元素
            initializeQueue(queue);
    } else {
        queue->front = (queue->front + 1) % MAX_SIZE;
    }
}
```

二叉树-BFS

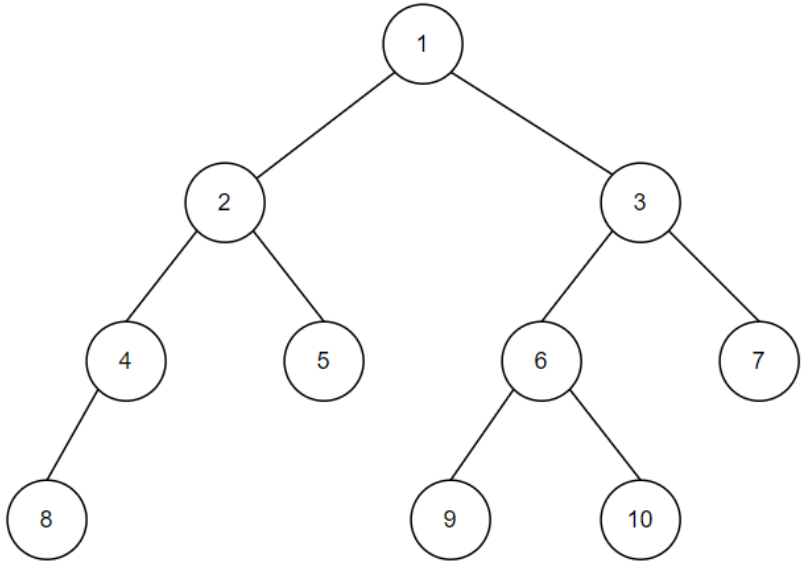
```
BFS(root):
    if root is null:
        return

    create an empty queue
    enqueue(root)

    while queue is not empty:
        current = dequeue(queue)
        print current.data

        if current.left is not null:
            enqueue(current.left)

        if current.right is not null:
            enqueue(current.right)
```



Console:
Queue：[ ]

二叉树-BFS

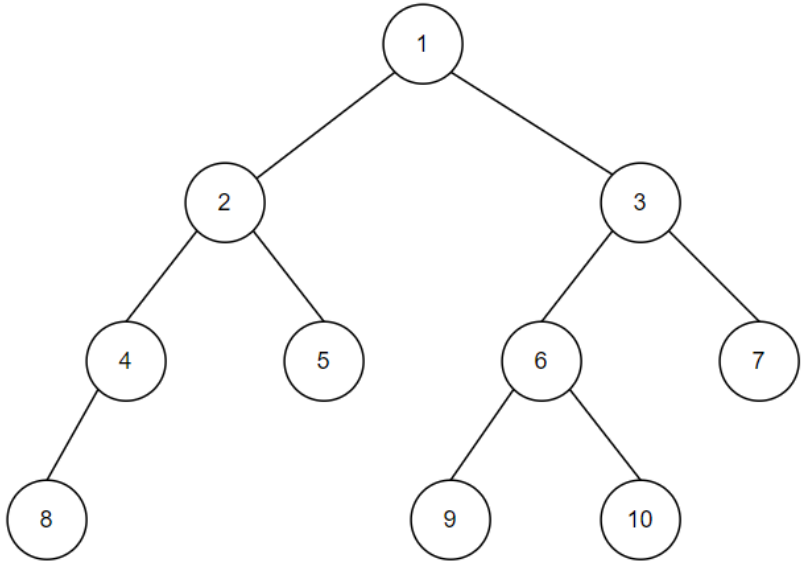
```
BFS(root):
    if root is null:
        return

    create an empty queue
    enqueue(root)

    while queue is not empty:
        current = dequeue(queue)
        print current.data

        if current.left is not null:
            enqueue(current.left)

        if current.right is not null:
            enqueue(current.right)
```



Console:
Queue：[1]

二叉树-BFS

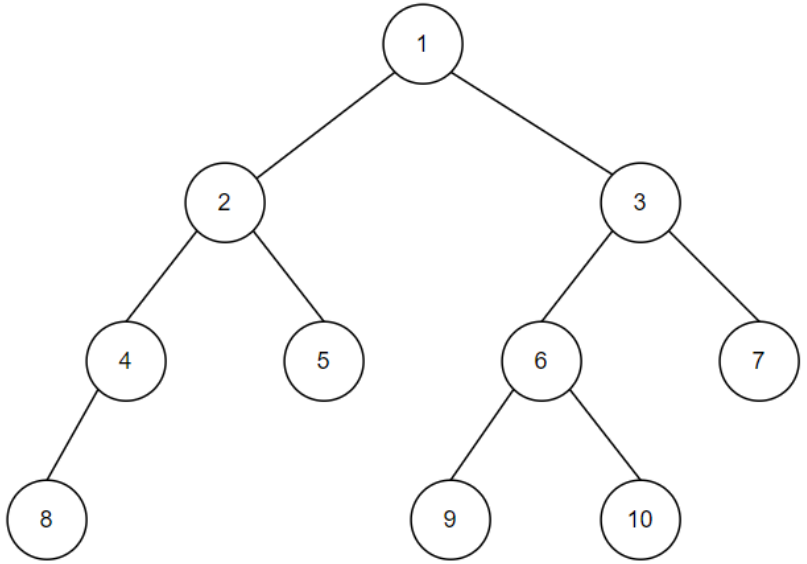
```
BFS(root):
    if root is null:
        return

    create an empty queue
    enqueue(root)

    while queue is not empty:
        current = dequeue(queue)
        print current.data

        if current.left is not null:
            enqueue(current.left)

        if current.right is not null:
            enqueue(current.right)
```



Console: 1
Queue：[2, 3]

二叉树-BFS

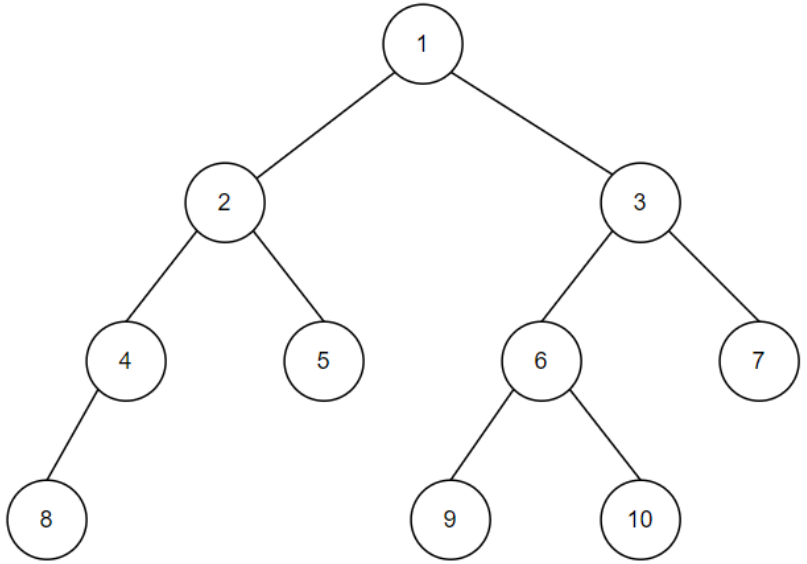
```
BFS(root):
    if root is null:
        return

    create an empty queue
    enqueue(root)

    while queue is not empty:
        current = dequeue(queue)
        print current.data

        if current.left is not null:
            enqueue(current.left)

        if current.right is not null:
            enqueue(current.right)
```



Console: 1 2
Queue：[3, 4, 5]

二叉树-BFS

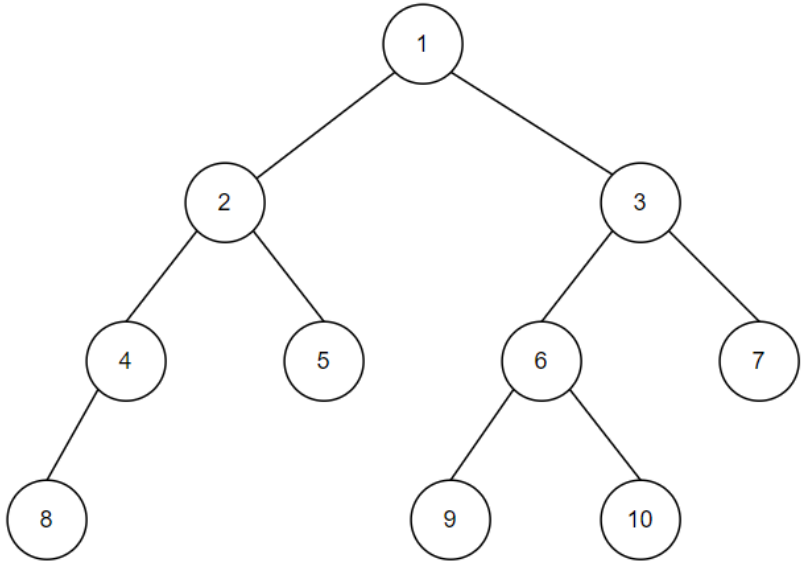
```
BFS(root):
    if root is null:
        return

    create an empty queue
    enqueue(root)

    while queue is not empty:
        current = dequeue(queue)
        print current.data

        if current.left is not null:
            enqueue(current.left)

        if current.right is not null:
            enqueue(current.right)
```



Console: 1 2 3
Queue：[4, 5, 6, 7]

二叉树-BFS

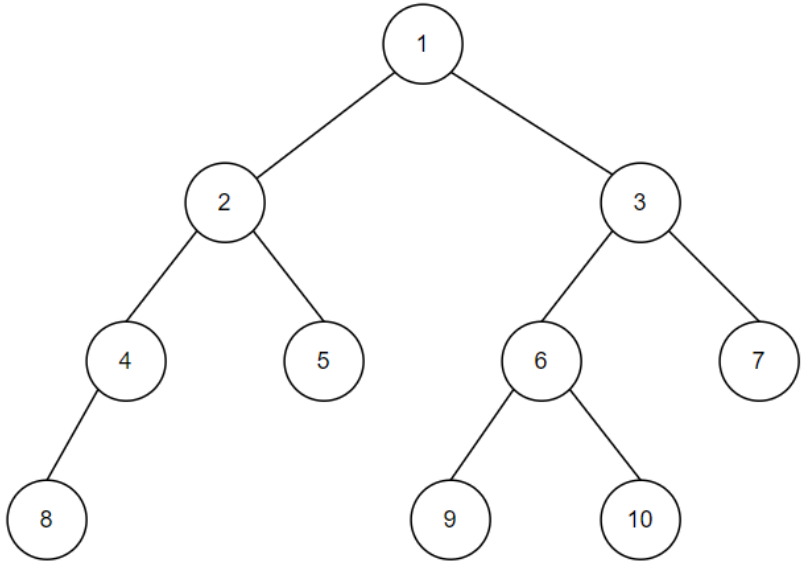
```
BFS(root):
    if root is null:
        return

    create an empty queue
    enqueue(root)

    while queue is not empty:
        current = dequeue(queue)
        print current.data

        if current.left is not null:
            enqueue(current.left)

        if current.right is not null:
            enqueue(current.right)
```



Console: 1 2 3 4
Queue：[5, 6, 7, 8]

二叉树-BFS

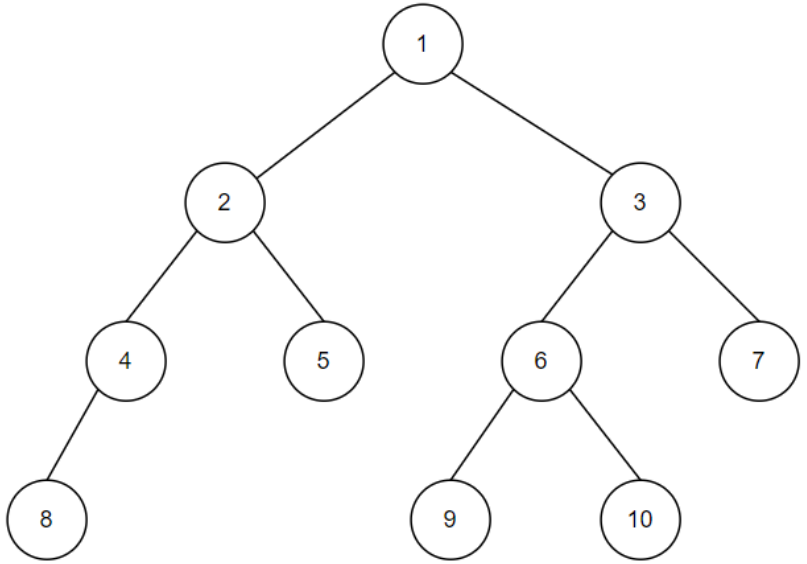
```
BFS(root):
    if root is null:
        return

    create an empty queue
    enqueue(root)

    while queue is not empty:
        current = dequeue(queue)
        print current.data

        if current.left is not null:
            enqueue(current.left)

        if current.right is not null:
            enqueue(current.right)
```



Console: 1 2 3 4 5 6
Queue：[7, 8, 9, 10]

二叉树-BFS

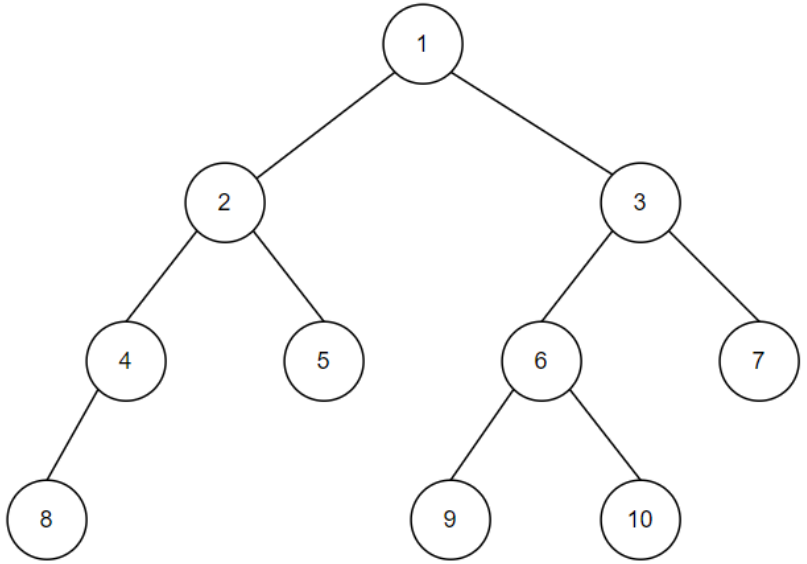
```
BFS(root):
    if root is null:
        return

    create an empty queue
    enqueue(root)

    while queue is not empty:
        current = dequeue(queue)
        print current.data

        if current.left is not null:
            enqueue(current.left)

        if current.right is not null:
            enqueue(current.right)
```



Console: 1 2 3 4 5 6 7
Queue：[8, 9, 10]

二叉树-BFS

```
BFS(root):
    if root is null:
        return

    create an empty queue
    enqueue(root)

    while queue is not empty:
        current = dequeue(queue)
        print current.data

        if current.left is not null:
            enqueue(current.left)

        if current.right is not null:
            enqueue(current.right)
```



Console: 1 2 3 4 5 6 7 8 9 10
Queue：[ ]

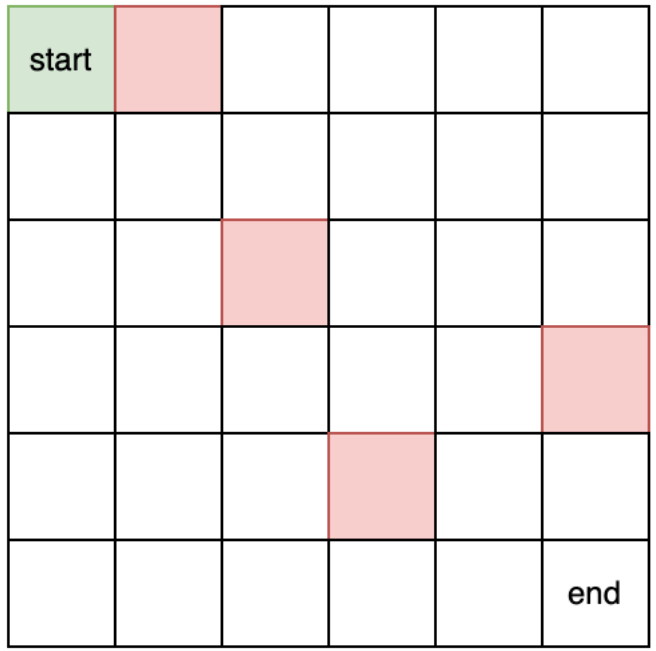
BFS-迷宫寻路

在一个二维迷宫中，你需要找到从起点到终点的最短路径。迷宫由空格和墙壁组成，其中：

- 空格表示可通行的路径；
- 墙壁表示不可通行的障碍物。

你可以从起点出发，沿着空格移动，但不能穿过墙壁。移动可以在上、下、左、右四个方向上进行。

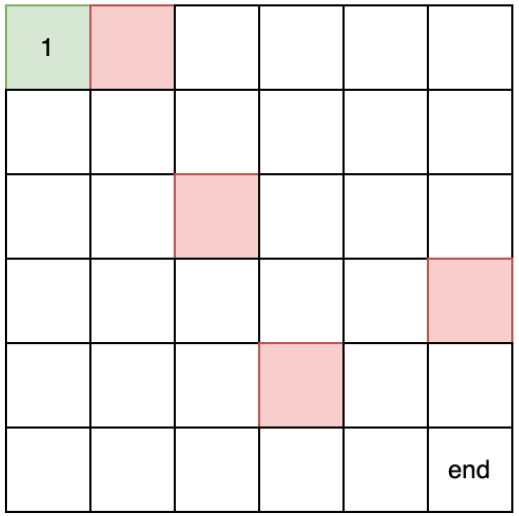
编写一个算法，找到从迷宫的起点到终点的最短路径，并输出路径的坐标或长度。如果没有可行的路径，则输出相应的提示信息。

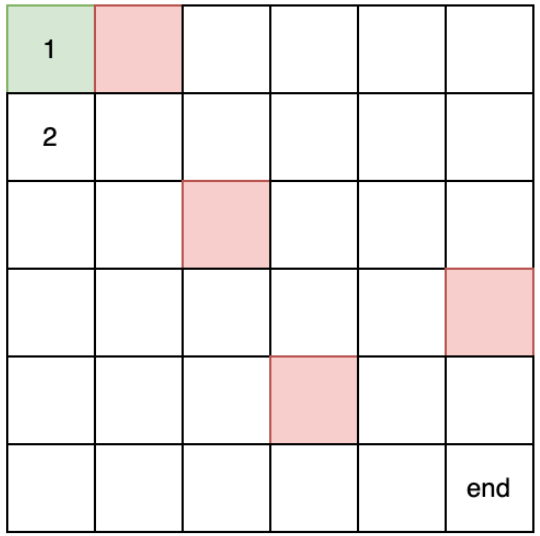


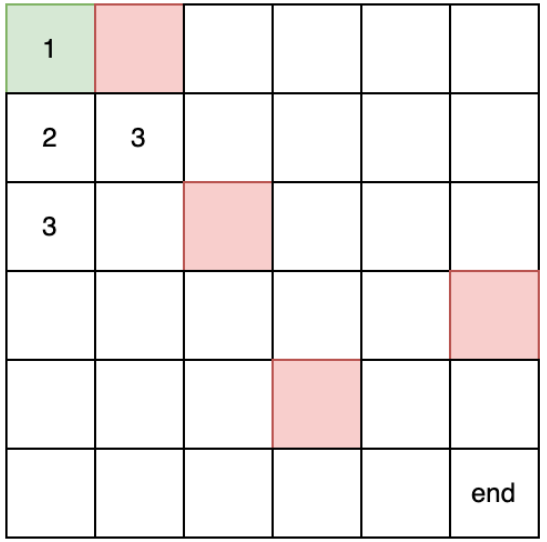
BFS-迷宫寻路

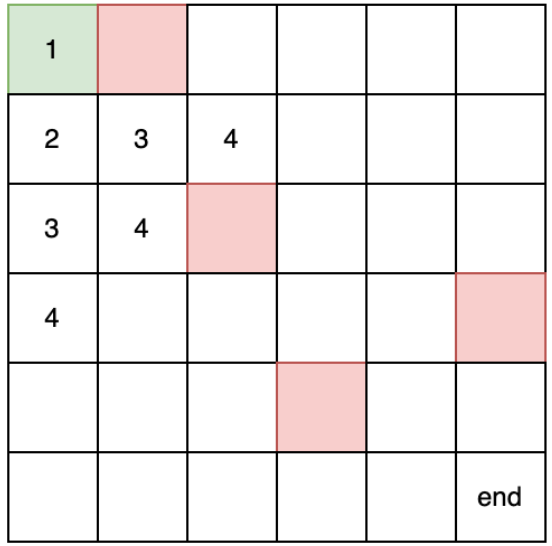
```python
def bfs(maze, start, end):
    # 使用队列来进行广度优先搜索，初始时将起始点入队
    queue = [(start[0], start[1], [])]  # (row, col, path)
    # 记录已经访问过的点
    visited = set()

    # 当队列不为空时，继续搜索
    while queue:
        # 从队列中取出当前位置和路径
        current_row, current_col, path = queue.pop(0)
        # 如果当前位置是终点，返回找到的路径
        if (current_row, current_col) == end:
            return path + [(current_row, current_col)]
        # 如果当前位置已经访问过，跳过
        if (current_row, current_col) in visited:
            continue
        # 将当前位置标记为已访问
        visited.add((current_row, current_col))
        # 遍历四个方向
        for dr, dc in directions:
            new_row, new_col = current_row + dr, current_col + dc
            # 检查新位置是否在迷宫范围内且可通行
            if 0 <= new_row < len(maze) and 0 <= new_col < len(maze[0]) and maze[new_row][new_col] == 0:
                # 将新位置和更新后的路径加入队列
                queue.append((new_row, new_col, path + [(current_row, current_col)]))
    # 如果搜索完整个迷宫仍未找到路径，返回None
    return None
```

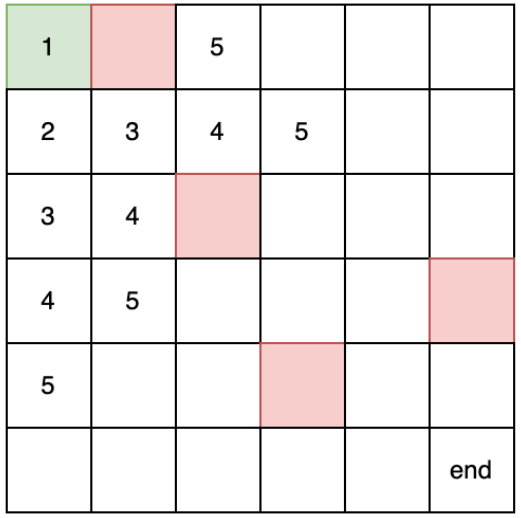
BFS-迷宫寻路

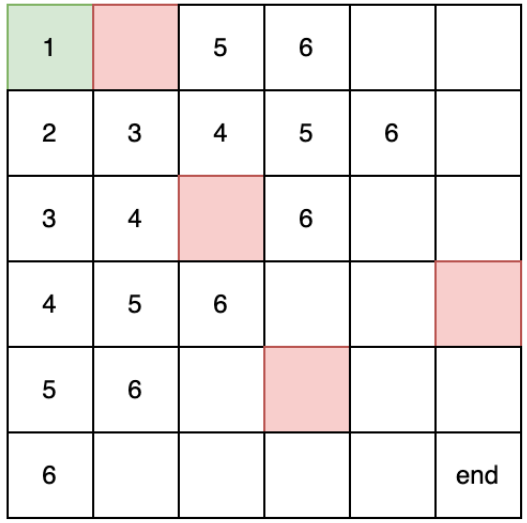

1
end

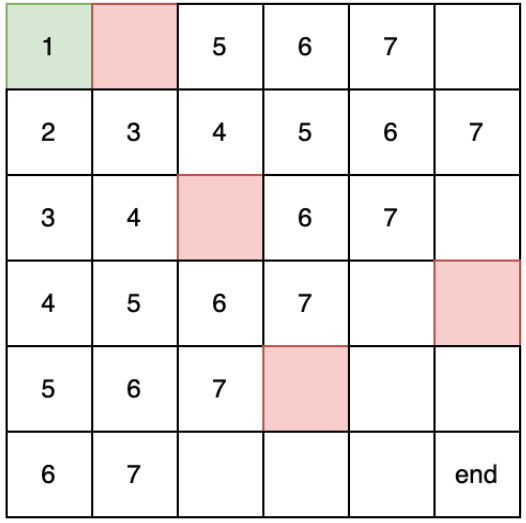

1
2
end

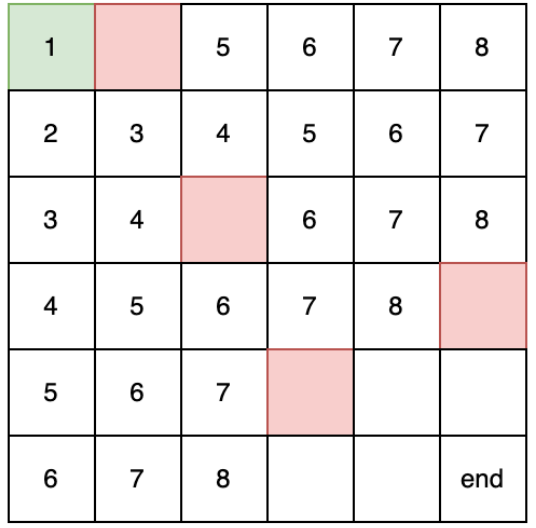

1
2
3
3
end


1
2
3
3
4
4
end

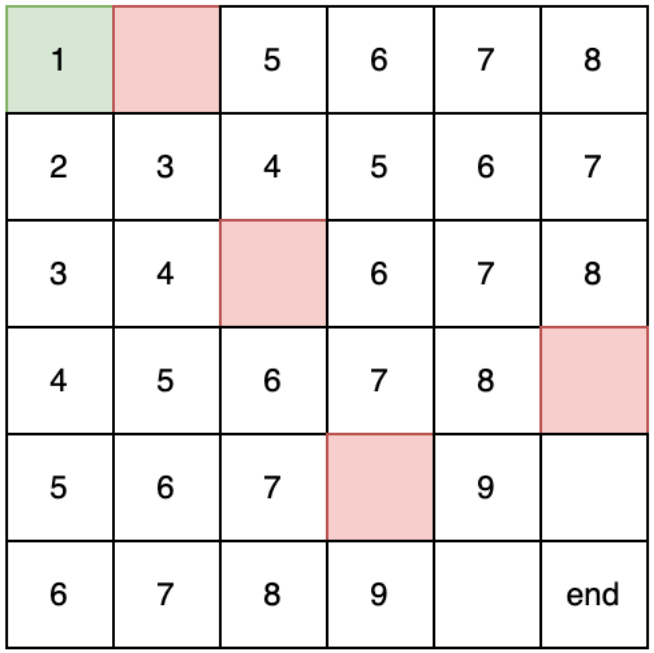

1
5
2
3
4
5
3
4
4
5
5
end

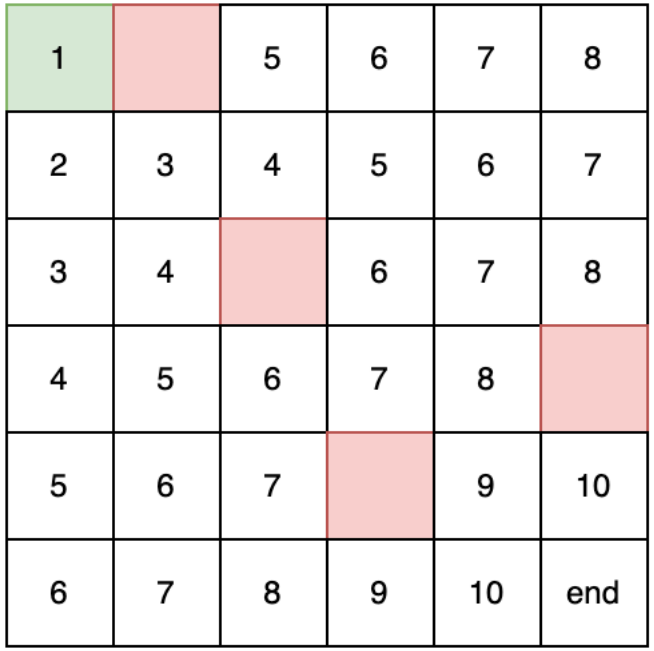

1
5
6
2
3
4
5
6
3
4
6
4
5
6
5
6
6
end

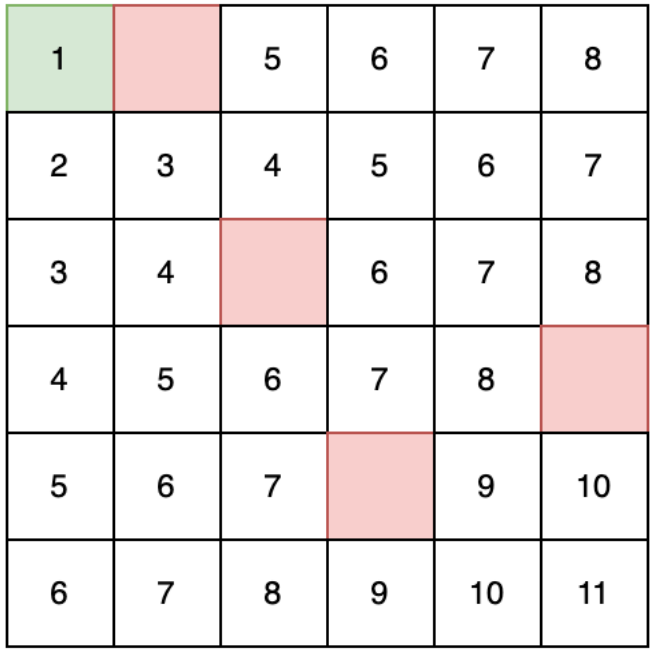

1
5
6
7
2
3
4
5
6
7
3
4
6
7
4
5
6
7
5
6
7
6
7
end


1
5
6
7
8
2
3
4
5
6
7
3
4
6
7
8
4
5
6
7
8
5
6
7
6
7
8
end

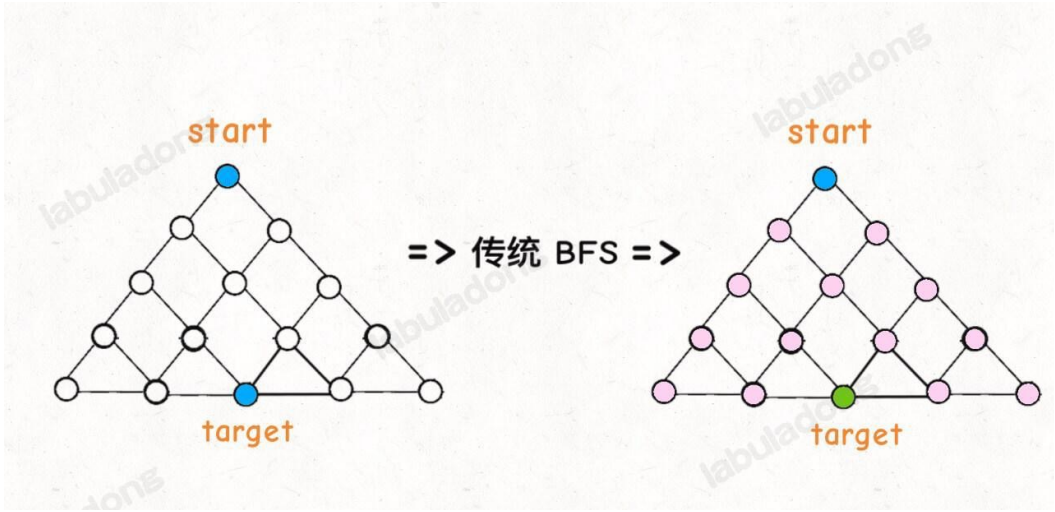
BFS-迷宫寻路

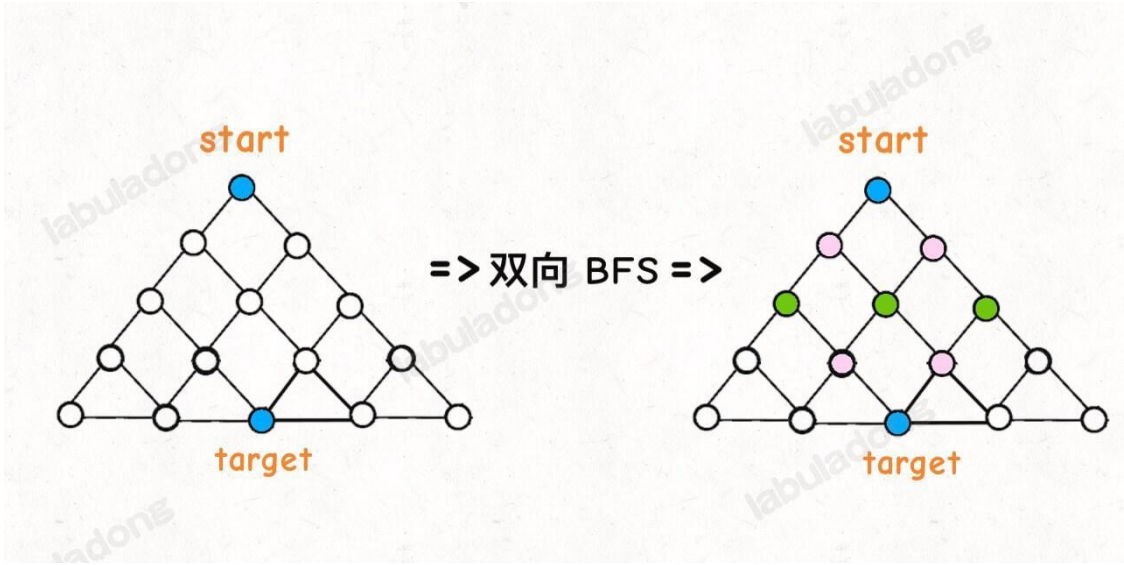

1
5
6
7
8
2
3
4
5
6
7
3
4
6
7
8
4
5
6
7
8
5
6
7
9
6
7
8
9
end


1
5
6
7
8
2
3
4
5
6
7
3
4
6
7
8
4
5
6
7
8
5
6
7
9
10
6
7
8
9
10
end

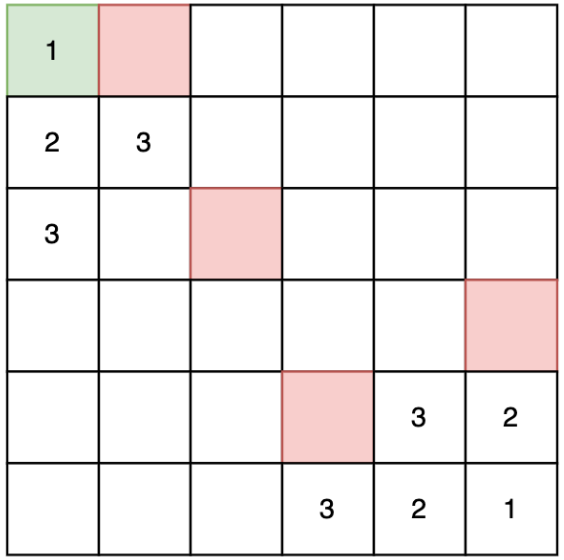

1
5
6
7
8
2
3
4
5
6
7
3
4
6
7
8
4
5
6
7
8
5
6
7
9
10
6
7
8
9
10
11

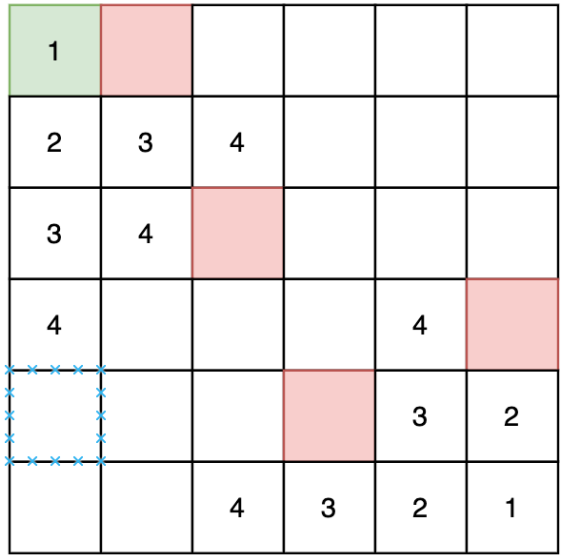
迷宫寻路-双向BFS

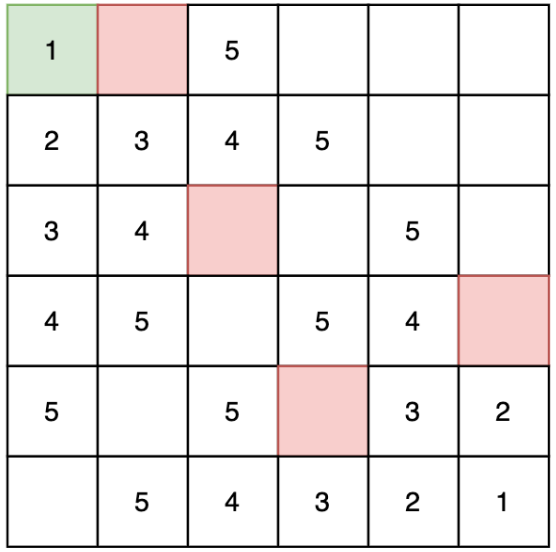

start
target
=> 传统 BFS =>
start
target

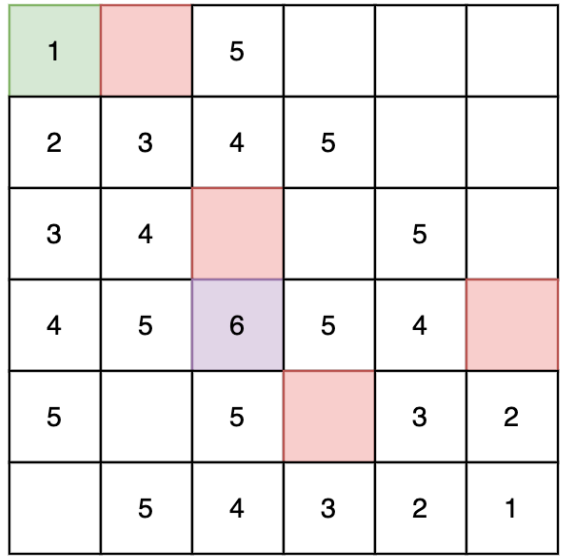

start
target
=> 双向 BFS =>
start
target

迷宫寻路-双向BFS


1
2 3
3
3 2
3 2 1


1
2 3 4
3 4
4 4
3 2
4 3 2 1


1 5
2 3 4 5
3 4 5
4 5 5 4
5 5 3 2
5 4 3 2 1


1 5
2 3 4 5
3 4 5
4 5 6 5 4
5 5 3 2
5 4 3 2 1

延伸

关于二叉树

- 二叉搜索树（BST）
- 平衡二叉树（AVL）
- 霍夫曼树（Huffman Tree）
- 堆和堆排序（Heap）
- 多叉树

关于DFS

- 全排列问题
- N皇后问题
- 图的连通性检测

关于BFS

- 树的最小深度
- 解开密码锁的最少次数

期末机考复习

➢ Runtime Error

1. 数组开的太小，访问了不该访问的内存区域，数组范围可以是上限+5
2. 很大的数组在main函数中定义，应该定义为全局变量。main函数中的数组和临时变量一样会被放在栈区，如果数组过大，就会导致栈发生上溢
3. 发生除0错误
4. 指针指向错误
5. 程序抛出了未接受的异常

➢ Time Limit Exceeded

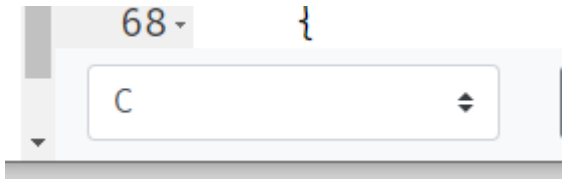
1. 程序时间复杂度较高，优化算法
2. 存在死循环

➢ Memory Limit Exceeded：内存超限

1. 程序运行所用的内存太多了，超过了对应题目的限制
2. 内存空间申请过大

Compile Error 编译错误

1. 编译器选取错误，使用高版本的特性，却使用低版本编译器
2. 语法错误
3. 头文件问题，使用window系统的头文件，但OJ却是linux系统，少添加了头文件



提交不了看看这里

期末机考复习

➢ 机考的时候

- 前面的题比较简单，搞懂题目的意思再开始敲代码，争取能够做的又快又准确
- 如果题目有提示或者注释，一定要仔细看.
- 关注题面的核心而不是表情包
- 后面题稍微难一些，可以优先拿一部分分数，然后反复提交，慢慢完善
- 注意数据范围，浮点数float/double，整型int/long long int，数组开合适的大小
- 注意看清楚输出的要求：字母大小写、是否有空格或者换行等
- 做不完很正常，所以一定要多次提交，确保代码能够正常运行起来

➢ 复习建议

- 逐个回顾一下每次课程的知识点，比如各种数据类型、数组、条件、循环、排序、查找、结构体、指针、链表等等
- 动手敲代码，刷OJ，包括但不限于以前的作业题目、LeetCode 等公开的OJ网站
- 没有思路的时候，在网上查看别的人的代码或者讲解，学习别人的代码不可耻
- 学会使用搜索工具，比如百度、谷歌等等，有的问题自己去搜索比问助教更靠谱

谢谢！

祝同学们取得理想的成绩！