数据结构和算法

1.衡量一个算法的标准

时间复杂度：程序大概要运行的次数

空间复杂度：算法执行过程中大概所占用的最大内存

2. 复习结构体以及结构体指针

```
struct Student
{
    int sid;
    char name[200];
    int age;
}

int main (void)
{
    struct Student st = {1000, "zhangsan", 23};
    stuct Student *pst;
    pst = &st;
    pst -> sid = 99;
}
```

注意：pst -> sid 等价于 (*pst).sid，等价于st.sid

```
*结构体变量作为指针在函数中传递*/

/*一个字节是一个地址，指针根据前面的类型来决定一共指向多少个字节
例如，int *p中，p指向4个字节；而char *p中，p指向一个字节；但p作为指针变量，永远占4个字节 */

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

struct Student
{
    /* data */
    int age;
    char sex;
    char name[100];
};

//此函数无效
void InputStudent(struct Student stu)
```

```
{
    stu.age = 10;
    strcpy(stu.name, "hangyu"); //注意，不能写成stu.name = "hangyu"
    stu.sex = 'M';
    return;
}

void InputStudent1(struct Student * pstu) //注意，不管指向的变量占多少字节，pstu都只占用4个
{
    (*pstu).age = 10;
    strcpy(pstu->name, "hangyu"); //注意，不能写成stu.name = "hangyu"
    pstu->sex = 'M';
    return;
}

int main (void)
{
    struct Student st1; //初始化

    InputStudent(st1); //此函数无效
    printf("The student's name is %s, age is %d, sex is %c\n", st1.name, st1.age, st1.sex);

    InputStudent1(&st1);
    printf("The student's name is %s, age is %d, sex is %c\n", st1.name, st1.age, st1.sex);
    return 0;
}
```

3. 数组复习（自己手动构建一个数组以及实现一些功能）

1. 初始化数组

```
struct Arr
{
    int *pBase;     //指向第一个元素
    int len;        //能容纳的最大元素个数（用于内存分配）
    int cnt;        //当前数组有效元素个数
};
```

```
void init_arr(struct Arr * pArr, int length)
{
    pArr->pBase = (int*)malloc(sizeof(int)*length);
    if (pArr->pBase == NULL)            //如果分配失败
    {
        printf("Fail to allocate\n");
        exit(-1);                       //终止整个程序
```

```
    }
    else
    {
        pArr->len = length;
        pArr->cnt = 0;
    }
    return;
}
```

2. 判断数组是否为空

```
bool is_empty(struct Arr *pArr)
{
    if (pArr->cnt == 0)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

3. 判断数组是否满了

```
bool is_full(struct Arr *pArr)
{
    if(pArr ->cnt == pArr ->len)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

4. 打印整个数组

```
void show_arr(struct Arr * pArr)
{
    if(is_empty(pArr))          //如果为空
    {
        printf("Is empty\n");
    }
    else
    {
        for (int i = 0; i<pArr->cnt; i++)
        {
            printf("%d ",pArr->pBase[i]);    //重点！！！！
        }
```

```
        printf("\n");
    }
    return;
}
```

pArr指向的是结构体，其中的pBase才是指向数组的

5. 在尾部加进元素

```
bool append_arr(struct Arr *pArr, int num)
{
    if(is_full(pArr))       //满了
    {
        return false;
    }
    else
    {
        pArr->pBase[pArr->cnt] = num;
        (pArr->cnt) ++;
        return true;
    }
}
```

6. 在数组中插入元素

```
//pos为插入的位置（第pos个index），其他所有元素向后移一位
bool insert_arr(struct Arr *pArr, int pos, int val)
{
    if(is_full(pArr))
    {
        return false;
    }
    if(pos<1 || pos >pArr->cnt+1)
    {
        return false;
    }

    for (int i = (pArr->cnt) -1; i>=pos-1; i--)
    {
        pArr->pBase[i+1] = pArr->pBase[i];
    }
    pArr->pBase[pos-1] = val;
    (pArr->cnt)++;
    return true;
}
```

7. 删除数组中的某个元素
8. 倒置整个数组
9. 数组从小到大排序

4. Typedef

目的：对一些类型可以自己取名字

```
#include <stdio.h>

typedef int ZHANGSAN;

int main(void)
{
    ZHANGSAN j = 20;
    printf("%d\n", j);
}
```

也可以

```
#include <stdio.h>

typedef struct Student
{
    int sid;
    char name[100];
    char sex;
}ST;

int main(void)
{
    struct Student st; //等价于ST st

    ST st2;
    st2.sid = 100;
    printf("%d\n", st2.sid);
}
```

或者

```
#include <stdio.h>

typedef struct Student
{
    int sid;
    char name[100];
    char sex;
} * PST;                //struct Student* == PST

int main(void)
{
    struct Student st;
    PST ps = &st;
    ps->sid = 100;
```

```
    printf("%d\n",st.sid);
}
```

再或者

```
#include <stdio.h>

typedef struct Student
{
    int sid;
    char name[100];
    char sex;
} * PST, ST;                //struct Student* == PST, struct Student == ST

int main(void)
{
    ST st;
    PST ps = &st;
    ps->sid = 100;
    printf("%d\n",st.sid);
}
```

5. 链表linkedlist

定义：n个节点，离散分配，彼此通过指针相互连接，每个节点只有一个前驱/后驱，首节点没有前驱节点，尾节点没有后驱节点

头节点目的：并不存放有效数据，只是方便对链表的操作（增删改查），类型和首节点一样

头指针：指向头节点的指针变量

首节点

尾节点

尾指针：指向尾节点的指针变量

确定一个链表需要一个参数：头指针，通过头指针可以推算出链表的全部信息

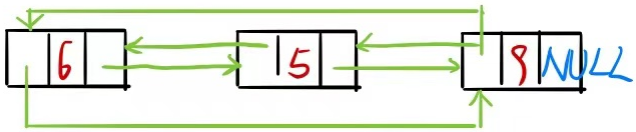
节点的定义：（一个节点的指针域，指向下一个节点整体）

```
typedef struct Node
{
    int num;
    struct Node * pNext;
}NODE, *PNODE;              //NODE == struct Node, PNODE == struct Node *
```

链表的分类：

单链表

双链表：每一个节点有两个指针域



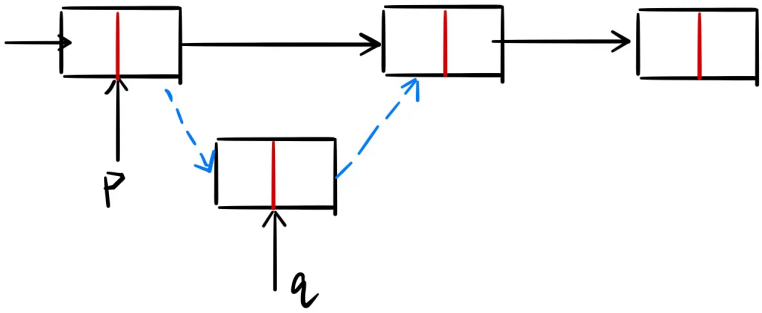
循环链表：能通过一个节点找到所有其他节点

非循环链表

链表操作

每一个节点有：num和pNext

1. 插入数值：q和p分别是指向两个节点的指针，q指向的节点要插入到原链表之间



方法1：定义临时变量r

```
r = p->pNext; p->pNext = q; q->pNext = r;
```

方法2：更好的：

```
q->pNext = p->pNext;
p->pNext = q;
```

2. 删除节点：要把p指向的节点的后面的节点删除

相当于直接把p指向后面的节点

```
p->pNext = p->pNext->pNext;      //这样写不对，因为没有free空间，会造成溢出
```

正确写法：

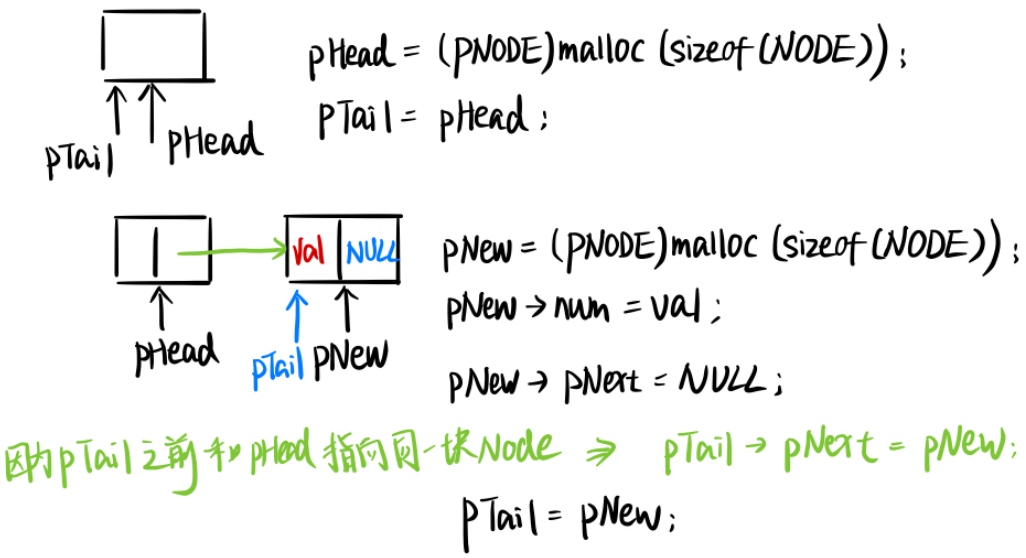
```
r = p->pNext;
p->pNext = p->pNext->pNext;
free(r);
```

3.

创建一个链表并且实现遍历方法

```
/*创建一个链表*/
#include <stdio.h>
#include <stdlib.h>

typedef struct Node
{
    int num;
    struct Node *pNext;
}NODE, *PNODE;
```

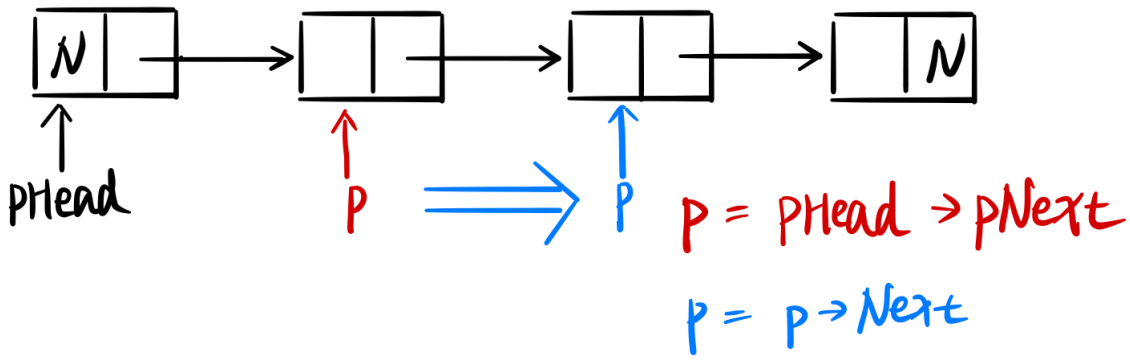
创造链表：




```
PNODE create_list()             //创建非循环单链表，并将头节点赋给PNODE
{
    int len;
    int i;
    int val;

    //创建一个不存放有效数据的头节点，pHead是头指针
    PNODE pHead = (PNODE)malloc(sizeof(NODE));

    //pTail永远指向最后一个节点
    PNODE pTail = pHead;
    pTail->pNext = NULL;
    if (pHead == NULL)
    {
        printf("Fail\n");
        exit(-1);
    }
```

```
    printf("Please enter the length of the array: \n");
    scanf("%d", &len);

    for (i = 0; i< len; i++)
    {
        printf("Please enter the num: ");
        scanf("%d", &val);
        PNODE pNew = (PNODE)malloc(sizeof(NODE));
        if (pHead == NULL)
        {
            printf("Fail\n");
            exit(-1);
        }
        //尾插法创造新节点
        pNew->num = val;
        pTail->pNext = pNew;
        pNew->pNext = NULL;
        pTail = pNew;
    }
    return pHead;
}
```

遍历链表：




```
//遍历链表，用一个指针p，不断p=p->Next遍历到下一个节点
void traverse_list(PNODE pHead)
{
    PNODE p = pHead->pNext;

    //如果不为空
    while(p != NULL)
    {
        printf("%d ", p->num);
        p = p->pNext;
    }
    printf("\n");

    return;
}

int main(void)
{
```

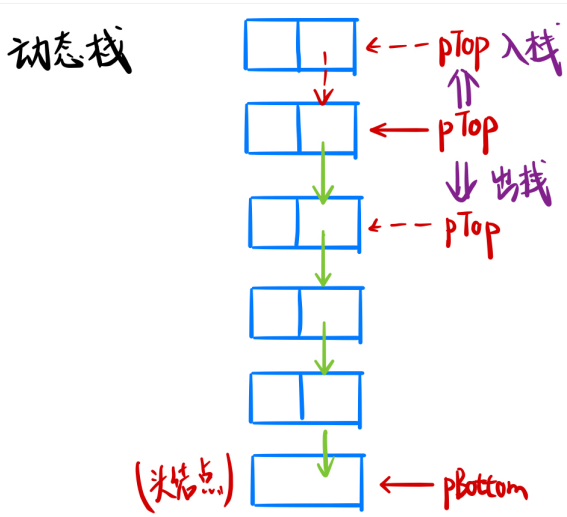
```
    PNODE pHead = NULL;
    pHead = create_list();
    traverse_list(pHead);        //遍历数组

    return 0;
}
```

6. 栈Stack

先入后出

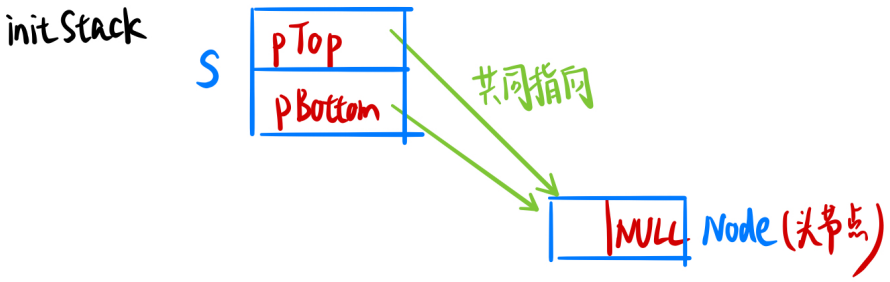
分类：1.静态栈 2.动态栈

静态栈是连续的，类似数组；动态栈类似链表

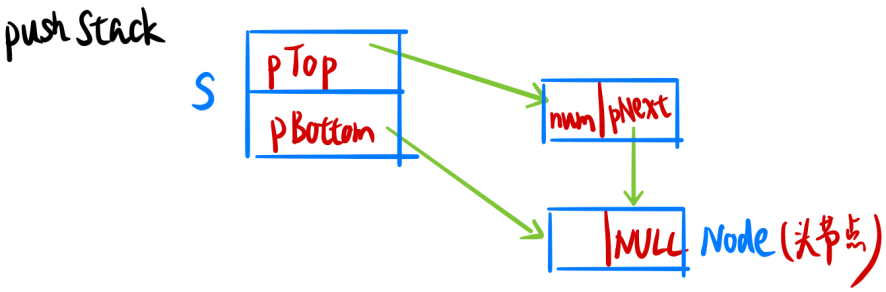


算法：出栈pop/压栈push

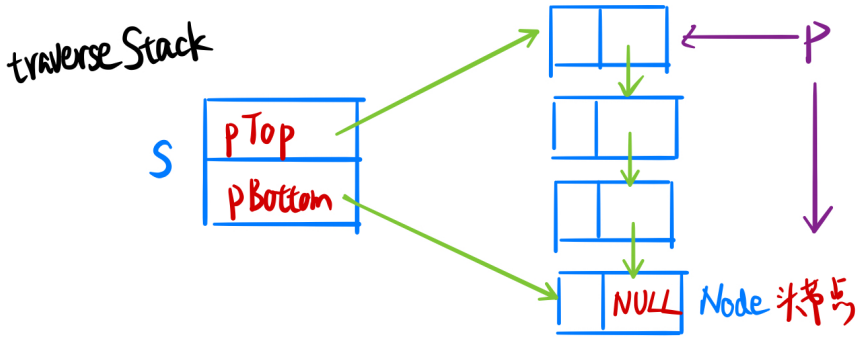
1. 初始化



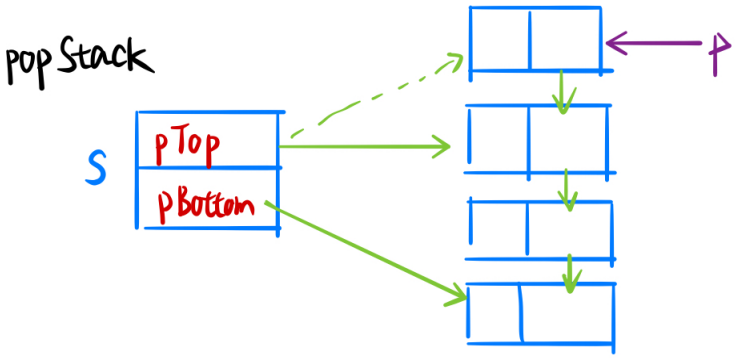
2. 入栈



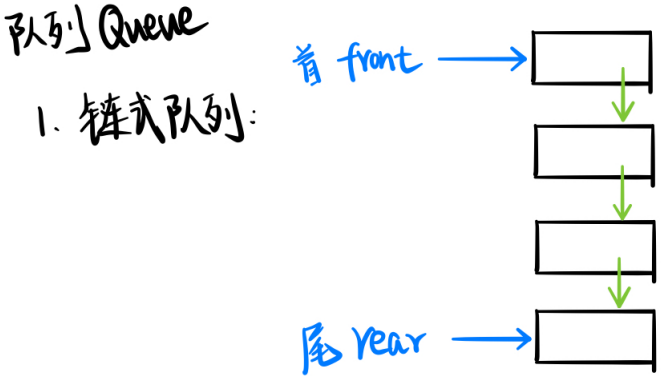
3. 遍历

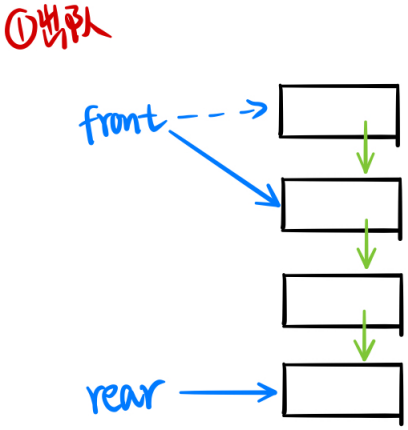


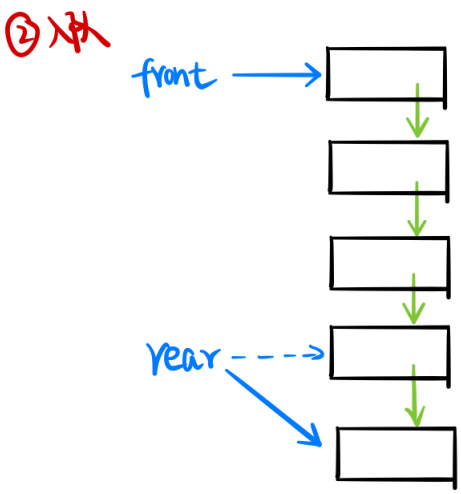
4. 出栈



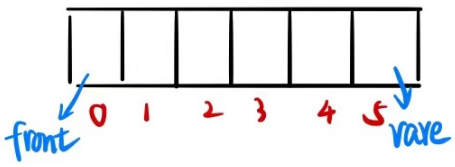
7. 队列Queue





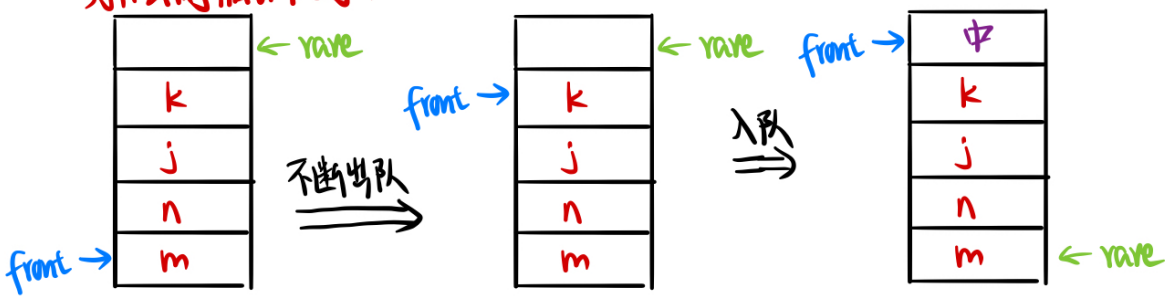


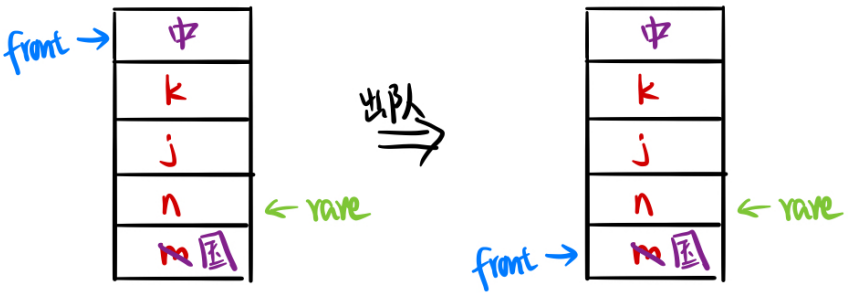
2. 静态队列（用数组实现）：通常是循环队列



※ r指向尾元素的下一个位置

为什么用循环队列？

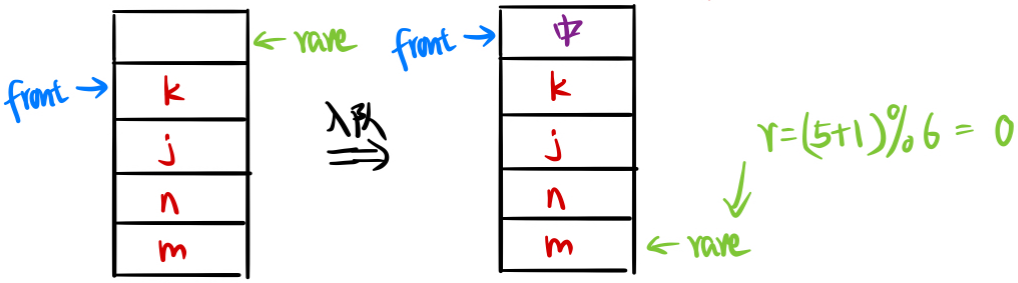




总结：入队时r向后移；出队时f向后移

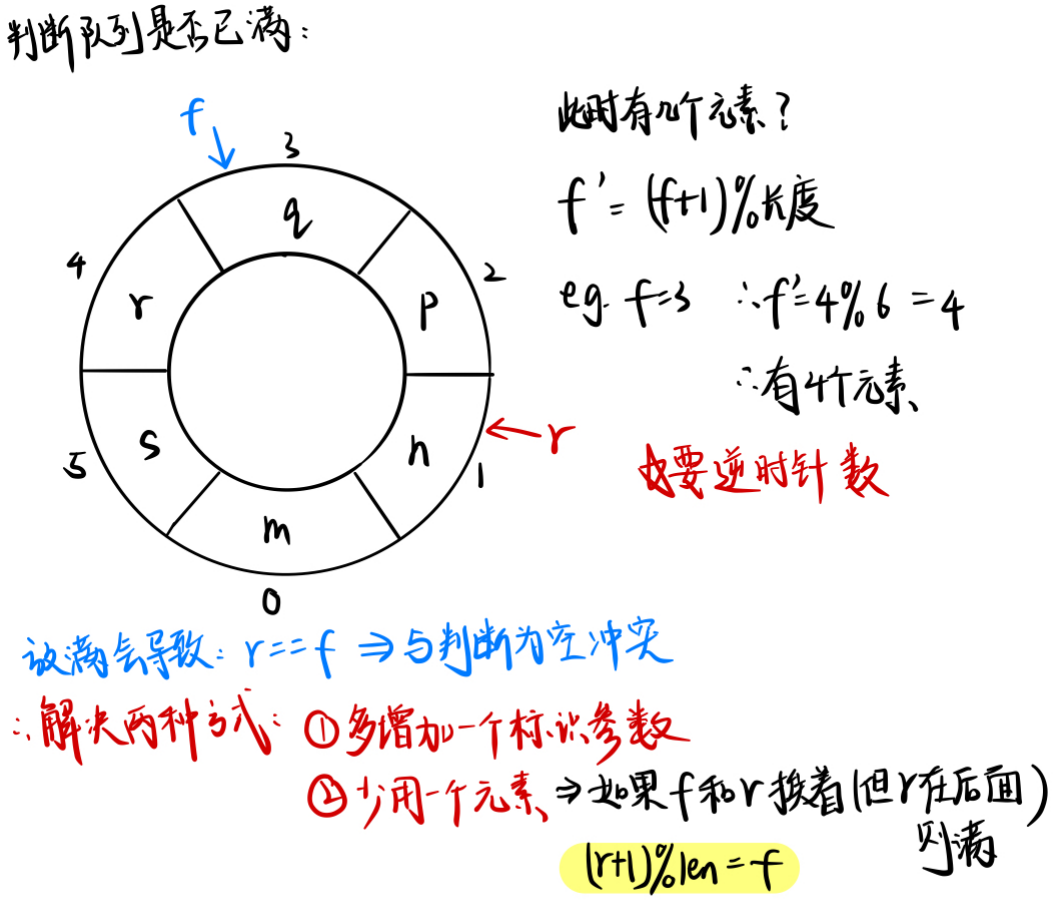
入队：
将值存入rear所代表的位置，r=(r+1)%数组长度　（不能写成 r=r+1）

(n-1)%n = n-1



出队：f向后移：f=(f+1)%数组长度

判断队列为空：r==f



判断一个队列为空：f == r

判断一个队列是否满了：

最好的方法是数组中永远空一个元素，如果(r+1)%len == f，说明满了

8. 哈希表

哈希表是一个数组，数组的下标是对应的类别

```
int harsh_list[100];

hash_list[1];        //年龄为一岁的人口数
```

9. 排序

1）快排

```
#include <stdio.h>

int find_pos(int *p, int low, int high)
{
    int num = p[low];
    while(low < high)
```

```
    {
        while(low<high && p[high]>=num)
        {
            high--;
        }
        p[low] = p[high];
        while(low<high && p[low]<=num)
        {
            low++;
        }
        p[high] = p[low];
    }

    p[low] = num;

    return low;     //或者return high也一样
}


void quick_sort(int *p, int low, int high)
{
    int pos;
    if(low < high)
    {
        pos = find_pos(p, low, high);
        quick_sort(p, pos+1, high);     //先排右侧
        quick_sort(p, low, pos-1);      //再排左侧
    }
}

void print_info(int *p, int len)
{
    for(int i = 0; i<len; i++)
    {
        printf("%d ",p[i]);
    }
    printf("\n");
}

int main()
{
    int arr[] = {3,1,5,2,7};
    quick_sort(arr, 0, 5);
    print_info(arr,5);
}
```

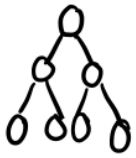
10. 树

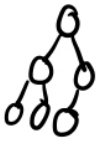
深度为k的二叉树，最多有`2^(k)-1`个节点

二叉树第k层最多有`2^(K-1)`个节点

深度位k的完全二叉树最少有`2^(K-1)`个节点

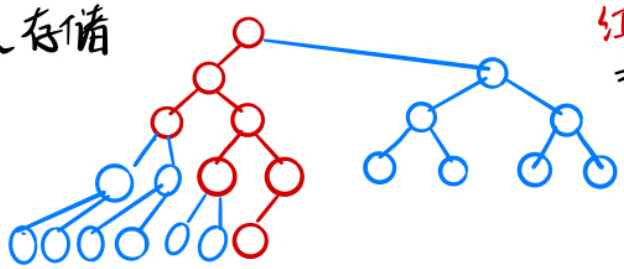
二叉树：
- 一般二叉树：子节点个数小于等于2
- 满二叉树：所有子节点个数都为2
- 完全二叉树：只能由满二叉树最右下的节点向左连续删掉n个节点的二叉树





二叉树存储：
- 连续存储：完全二叉树
- 链式存储

连续存储



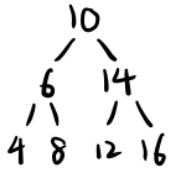
红色是要存的(有效节点)二叉树

⇒需要补全成完全二叉树(蓝色)

用完全二叉树的目的：可以推导出树原来的样子(能点存放有效节点)

二叉树特例：
- 二叉搜索树
- 堆
- 红黑树

二叉搜索树：
- 左子节点总是小于等于根节点
- 右子节点总是大于等于根节点



找一个节点需O(logn)

已知中序、后序画树：后序后出现为根

中序：BDCEAFHG
后序：DECBHGFA

步骤：①找根：后序最后一个：A
②分左右：根据A将中序分成左右
③找左子树根：中序中左子树为BDCE，在后序中找BDCE最后出现的为B
④找B左子树根：因为中序中B第一个出现，所以B无左子树
⑤找B右子树根：因为DCE在后序中，C最后出现，所以C为右子树根
⇒A左子树只剩DE，因为E中序后出现，所以E为右叶
⑥找右子树根：中序中右子树为FHG，因为后序中F最后出现，所以根为F
⑦找F左子叶：中序中F先出现⇒无左子叶
⑧找F右子树根：HG在后序中G后出现⇒G为根
⑨剩下H在中序中在G前出现⇒为左子叶

已知先序、中序画树：先序先出现为根

先序：A B C D E F G H
中序：B D C E A F H G

①找根：A
②找左子树：BDCE中，B在先序先出现：B为左子树根
③找B左子根：CDE中，C在先序中先出现：C为B的左子根
④DE中，D为左，E为右
⑤找右子树：FHG中，F在先序中先出现：F为右子树根
⑥HG在中序中都在F后出现⇒F无左子叶 ⑦F右子叶为G ⑧G左子叶为H