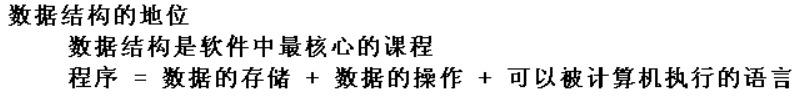
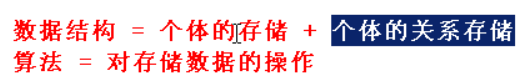
把数据结构算法学好了之后不意味着立马就可以做出一些什么东西来，虽然做不了东西出来，但是它很重要，是软件设计里的核心，学懂了它，在看其它的很多知识时就容易理解了，如内存里的栈和堆是什么？其实就是数据结构。

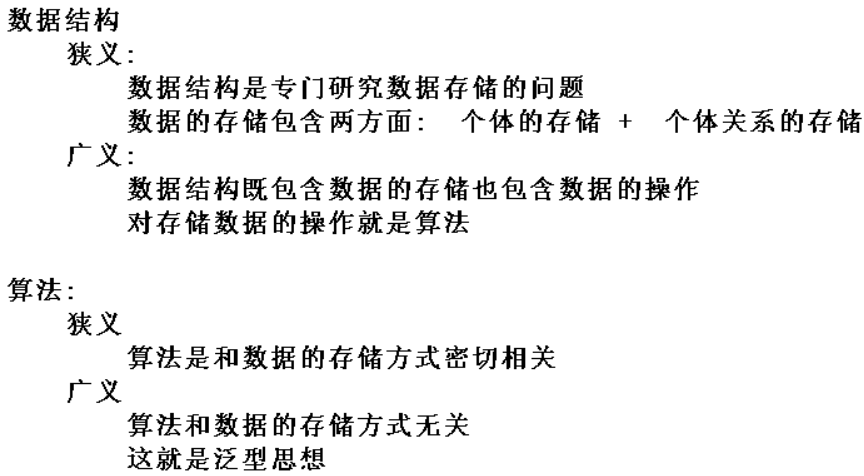
数据结构是用于研究数据存储的问题。

算法是用于研究操作这些数据的问题。



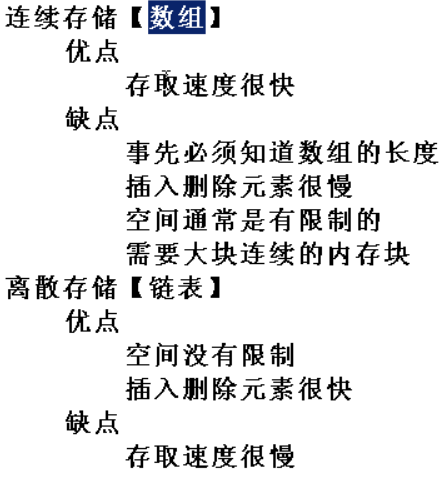


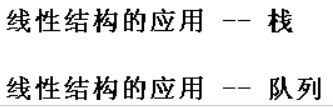




数据的存储结构有几种：

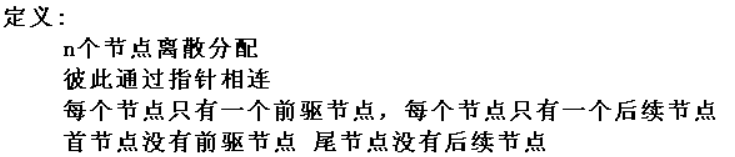
1、线性

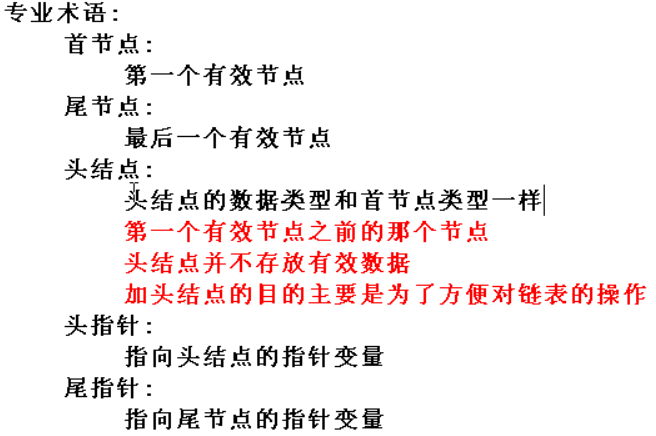


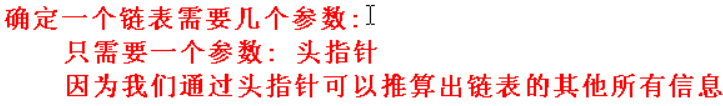


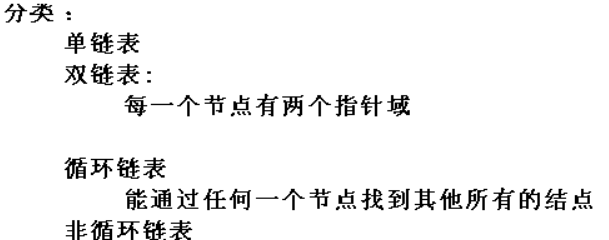
2、非线性：树、图

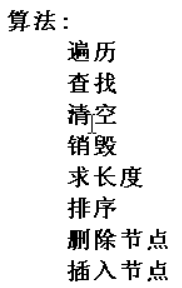
# 链表

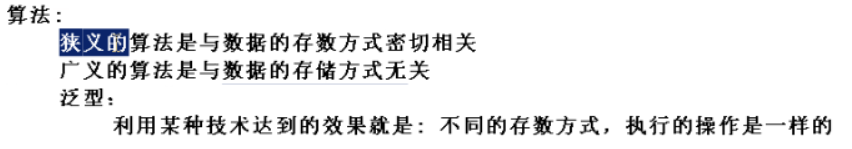












## 链表的实现

完整代码见：“实现一个链表.c”，这是给链表多加了一个头节点的，加入头节点的好处是拿到头节点就拿到了整个链表，头结点这个位置是永远不会变的，如果没有头节点，当我们把头节点传入方法中，比如排序、增、删等方法中时，如果首节点位置发生改变，则需要返回首节点，没有头节点的见“实现一个链表2.c”，可查看代码看具体不同之处在哪里。

#include<stdio.h>

#include<malloc.h>

typedef struct Node {

int data;

struct Node \* nextP;

}NODE, \*NODEP;

NODEP createList() {

int len;

int data;

int i;

NODEP headP = malloc(sizeof(NODE)); // 头结点

headP->nextP = NULL;

NODEP endP = headP; // 尾结点

printf("请输入您需要生成的链表节点的个数：len = ");

scanf("%d", &len);

for(i = 0; i < len; i++) {

printf("请输入第%d个节点的值：", i + 1);

scanf("%d", &data);

NODEP nodeP = malloc(sizeof(NODE));

nodeP->data = data;

nodeP->nextP = NULL;

endP->nextP = nodeP;

endP = nodeP;

}

return headP;

}

traverseList(NODEP headP) {

NODEP nodeP = headP->nextP;

while (nodeP != NULL) {

printf("%d\n", nodeP->data);

nodeP = nodeP->nextP;

}

}

main() {

NODEP headP = createList();

traverseList(headP);

}

链表的其它操作：

int isEmpty (NODEP headP) {

if (headP->nextP == NULL) {

return 1;

} else {

return 0;

}

}

int listLength(NODEP headP) {

NODEP nodeP = headP->nextP;

int lenght = 0;

while (nodeP != NULL) {

nodeP = nodeP->nextP;

lenght++;

}

return lenght;

}

sort(NODEP headP) {

int length = listLength(headP);

if (length < 2) {

return;

}

int swapCount;

int i;

for (swapCount = length; swapCount > 0; swapCount--) {

NODEP currentNodeP = headP; // 当前正在比较是否要交换位置的节点

NODEP preNodeP = NULL; // 当前节点的前一个节点

NODEP secondNodeP; // 当前节点的下一个节点

for (i = 0; i < swapCount; i++) {

if(preNodeP != NULL && currentNodeP->data > currentNodeP->nextP->data){

secondNodeP = currentNodeP->nextP; // 保存第二个节点

preNodeP->nextP = secondNodeP; // 前节点指向第二个节点

currentNodeP->nextP = secondNodeP->nextP; // 当前节点指向第三个节点

secondNodeP->nextP = currentNodeP; // 第二个节点指向当前节点

preNodeP = secondNodeP; // secondNodeP变为preNodeP

//currentNodeP = currentNodeP->nextP; // 交换位置后，currentNodeP在secondNodeP的后面，所以currentNodeP直接进入下一轮比较

} else {

// 如果没有交换过，则当前节点变为前节点，当前节点的下一个节点变为当前节点

preNodeP = currentNodeP;

currentNodeP = currentNodeP->nextP;

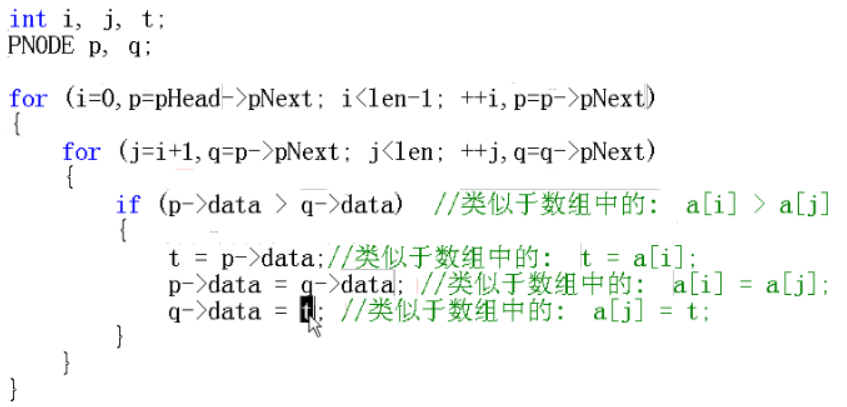
}

}

}

}

链表排序的另一种写法是只交换节点中的数据，而不是交换节点的位置，如下：



// 在链表headP的index位置插入一个节点，并存入data数据

int insert(NODEP headP, int index, int data) {

NODEP nodeP = headP->nextP;

NODEP preNodep = headP;

int i = 0;

int result = 0;

while (nodeP != NULL && i <= index) {

if (i == index) { // 位置找到，进行插入操作

NODEP newNodeP = malloc(sizeof(NODE));

newNodeP->data = data;

preNodep->nextP = newNodeP;

newNodeP->nextP = nodeP;

result = 1;

} else {

preNodep = nodeP;

nodeP = nodeP->nextP;

}

i++;

}

return result;

}

// 删除链表headP第index位置的元素

int delete(NODEP headP, int index, int \* data) {

NODEP nodeP = headP->nextP;

NODEP preNodep = headP;

int i = 0;

int result = 0;

while (nodeP != NULL && i <= index) {

if (i == index) { // 位置找到，进行删除操作

\*data = nodeP->data;

preNodep->nextP = nodeP->nextP;

free(nodeP);

result = 1;

} else {

preNodep = nodeP;

nodeP = nodeP->nextP;

}

i++;

}

return result;

}

main() {

NODEP headP = createList();

printf("创建的链表为：");

traverseList(headP);

/\*int result = isEmpty(headP);

if (result == 1) {

printf("链表为空\n");

} else {

printf("链表不为空\n");

}

printf("链表长度为%d\n", listLength(headP));

sort(headP);

printf("排序后的链表为：");

traverseList(headP);

int result = insert(headP, 6, 6);

if (result > 0) {

printf("插入节点成功，新链表为：");

traverseList(headP);

} else {

printf("插入节点失败\n");

} \*/

int data;

int result = delete(headP, 2, &data);

if (result > 0) {

printf("删除节点成功，新链表为：");

traverseList(headP);

printf("删除的节点为：%d\n", data);

} else {

printf("删除节点失败\n");

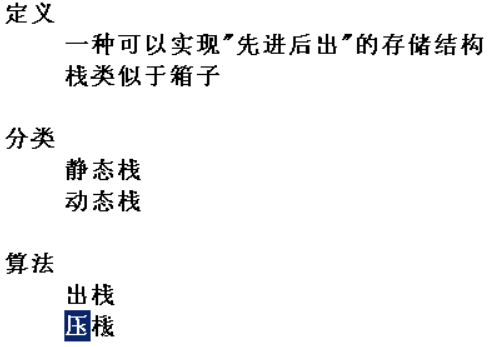
}

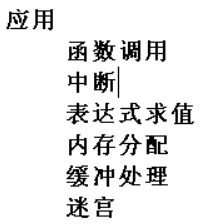
}

# 线性结构的两种常见应用

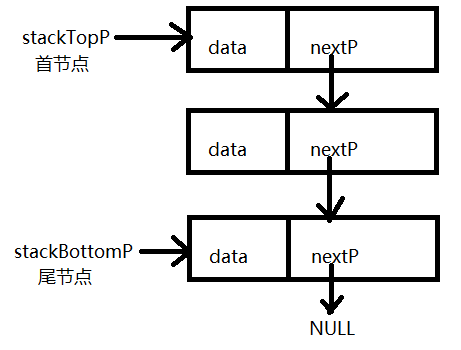
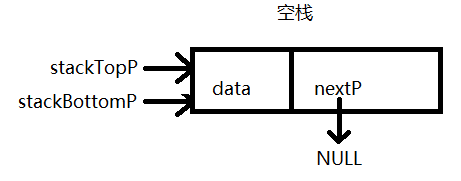
## 栈

队列是先进先出，栈是后进先出





栈的实现：

思路：  
  

使用链表实现，特殊的就是没有头节点，链表的首节点就是栈顶节点，插入的新节点只能放到首节点的前面，即插入的新节点必须要成为首节点。链表的尾结点为栈的栈底节点，且这个栈底节点是个空节点，不代表真实的数据，如果栈中只有一个栈底节点时，则为空栈。

具体代码见：“实现一个链表栈.c”。 我不明白为什么需要一个不代表真实数据的栈底，有什么好处么？我实现的不需要栈底的代码见：“实现一个链表栈2.c”。

代码：

#include<stdio.h>

#include<malloc.h>

typedef struct Node {

int data;

struct Node \* nextP;

}NODE, \*NODEP;

typedef struct Stack {

NODEP stackTopP;

NODEP stackBottomP;

}STACK, \*STACKP;

// 初始化栈

void init(STACKP stackP) {

NODEP nodeP = malloc(sizeof(NODE));

nodeP->nextP = NULL;

stackP->stackBottomP = nodeP;

stackP->stackTopP = nodeP;

}

// 入栈

void push(STACKP stackP, int e) {

NODEP newNodeP = malloc(sizeof(NODE));

newNodeP->data = e;

newNodeP->nextP = stackP->stackTopP;

stackP->stackTopP = newNodeP;

}

// 遍历栈

void traverseStack(STACKP stackP) {

NODEP nodeP = stackP->stackTopP;

while (nodeP != stackP->stackBottomP) {

printf("%d ", nodeP->data);

nodeP = nodeP->nextP;

}

printf("\n");

}

// 出栈，返回值代表是否出栈成功，eP用于保存出栈的元素

int pop(STACKP stackP, int \* eP) {

if (!isEmpty(stackP)) {

NODEP stackTopP= stackP->stackTopP;

stackP->stackTopP = stackTopP->nextP;

\*eP = stackTopP->data;

free(stackTopP);

return 1;

} else {

return 0;

}

}

int isEmpty(STACKP stackP) {

if (stackP->stackBottomP == stackP->stackTopP) {

return 1;

} else {

return 0;

}

}

void clear(STACKP stackP) {

if (!isEmpty(stackP)) {

NODEP stackTopP = stackP->stackTopP;

while (stackTopP != stackP->stackBottomP) {

stackP->stackTopP = stackTopP->nextP;

free(stackTopP);

stackTopP = stackP->stackTopP;

}

}

}

main() {

STACK stack;

init(&stack);

push(&stack, 1);

push(&stack, 2);

push(&stack, 3);

push(&stack, 4);

push(&stack, 5);

traverseStack(&stack);

/\*int val;

int result = pop(&stack, &val);

if (result == 0) {

printf("弹栈失败\n");

} else {

printf("弹栈成功，弹出来的元素是：%d\n", val);

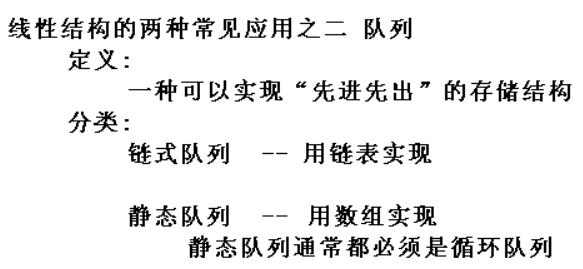
}\*/

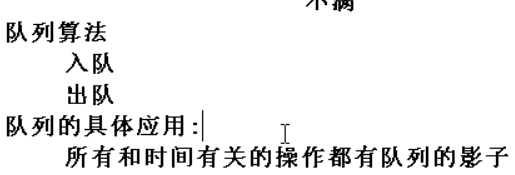
clear(&stack);

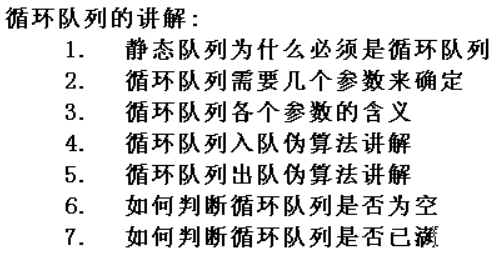
traverseStack(&stack);

}

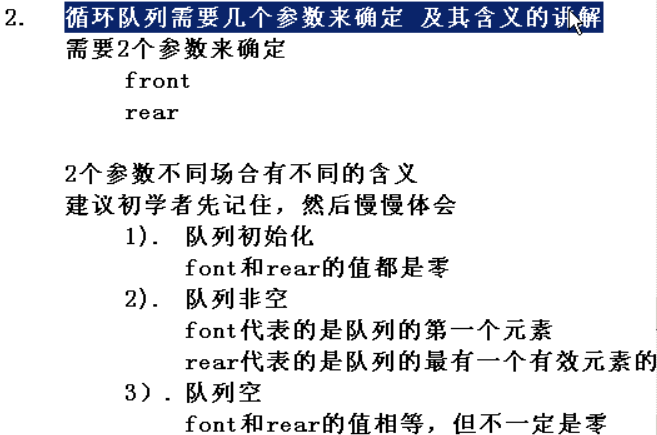
## 队列



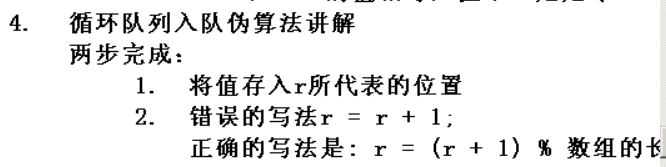


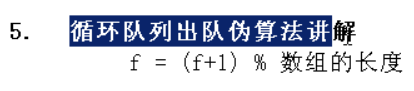


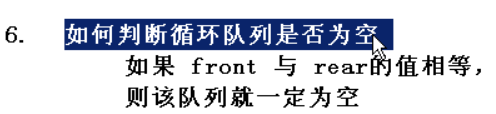
1、答案：循环就不需要移动元素的位置

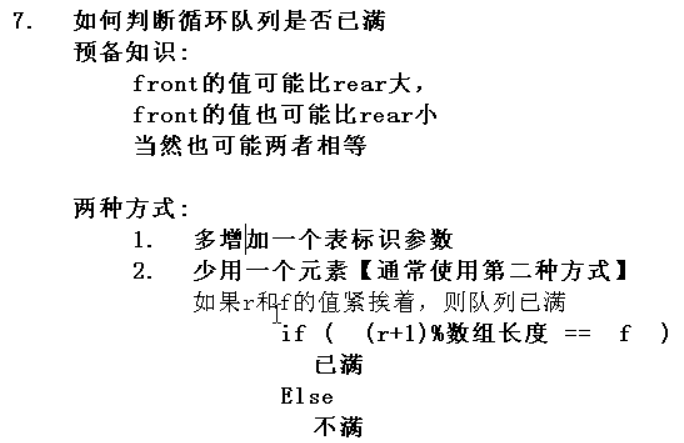


上面截图不完整，rear代表的是队列的最后一个有效元素的下一个位置。 （为什么要表示下一个位置，因为如果不这样的话，初始化时，也就是队列为空时，font和rear的值都为0，判断队列为空就是看font和rear是否都为0，如果此时插入1个数据，rear还是0的话，则无法判断队列是否为空了，所以要让rear指向最近插入数据位置的下一个位置）









第2种方式，如果不少用一个元素的话，当队列满时，则front和rear会相等，此时无法判断是否是满还是空。如果数组要用完的话，只能增加一个参数去记录所用元素的数量来判断是满还是空。

具体代码见：“实现一个循环队列.c”，我自己实现了一个第3种方案，不需要多增加参数，而且用完所有元素，也实现了相同的功能，见“实现一个循环队列2.c”；

实现代码：

#include<stdio.h>

#include<malloc.h>

typedef struct Queue {

int \* arrayP;

int front;

int rear;

} QUEUE, \* QUEUEP;

void init(QUEUEP queueP) {

queueP->arrayP = malloc(sizeof(int) \* 6);

queueP->front = 0;

queueP->rear = 0;

}

int enqueue(QUEUEP queueP, int value) {

if (isFull(queueP)) {

return 0;

} else {

queueP->arrayP[queueP->rear] = value;

queueP->rear = (++queueP->rear) % 6;

return 1;

}

}

int outQueue(QUEUEP queueP, int \* valP) {

if (!isEmpty(queueP)) {

\*valP = queueP->arrayP[queueP->front];

++queueP->front;

return 1;

} else {

return 0;

}

}

void traverseQueue(QUEUEP queueP) {

if (!isEmpty(queueP)) {

int i;

for (i = queueP->front; i != queueP->rear; i = (i + 1) % 6) {

printf("%d ", queueP->arrayP[i]);

}

printf("\n");

}

}

int isFull(QUEUEP queueP) {

if ((queueP->rear + 1) % 6 == queueP->front) {

return 1;

} else {

return 0;

}

}

int isEmpty(QUEUEP queueP) {

if (queueP->front == queueP->rear) {

return 1;

} else {

return 0;

}

}

main() {

QUEUE queue;

init(&queue);

enqueue(&queue, 1);

enqueue(&queue, 2);

enqueue(&queue, 3);

enqueue(&queue, 4);

enqueue(&queue, 5);

enqueue(&queue, 6);

traverseQueue(&queue);

int value;

int result = outQueue(&queue, &value);

if (result == 0) {

printf("出队失败\n");

} else {

printf("出队成功，出队的元素为：%d\n", value);

}

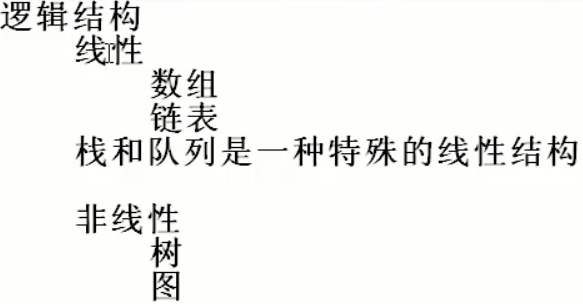
traverseQueue(&queue);

}

# 线性结构总结

数据结构的分类：

按逻辑结构分：

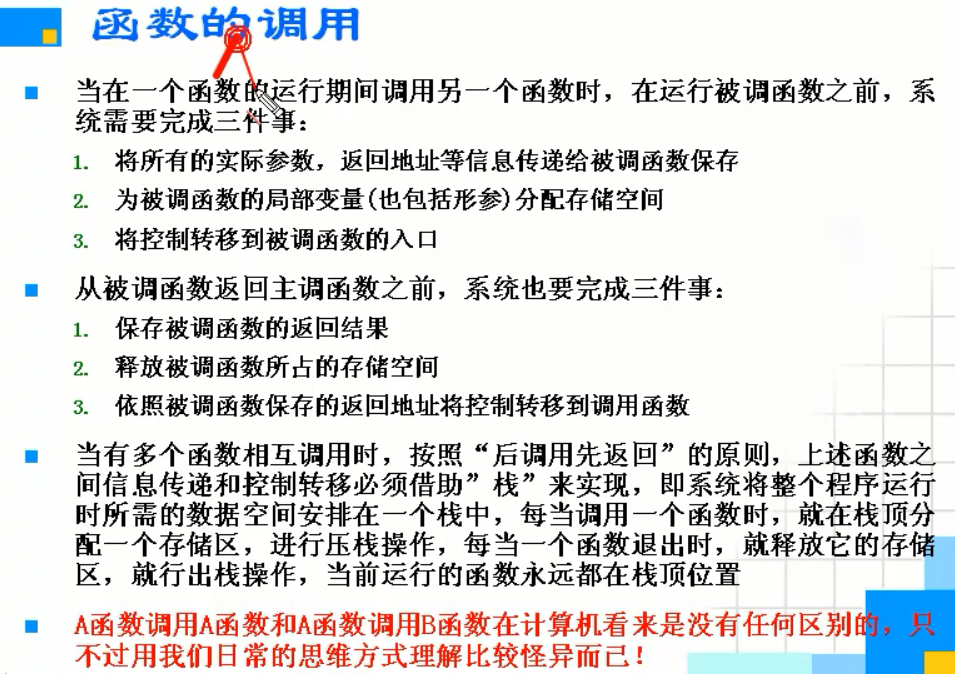


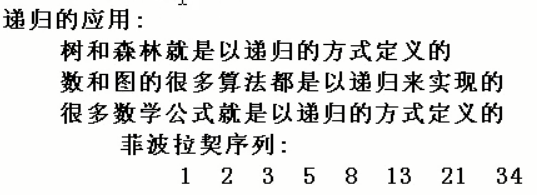
栈和队列可以用数组实现也可用链表实现。

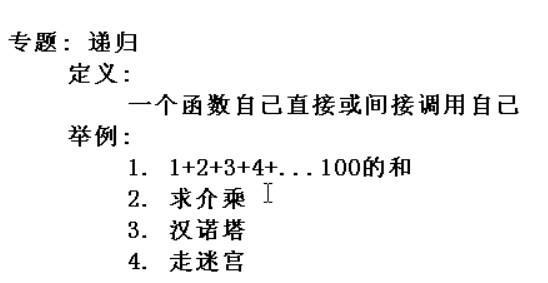
线性结构的算法已经非常的成熟了，我们不用自己去写关于这个结构的算法了，别人都写好了，比如Java中的List，表示链表，ArrayList是用数组实现的链表，LinkedList则是使用链表实现的链表。 树和图的算法就非常的多，现在还源源不断的有人写出新的算法。树的算法非常的复杂和不成熟，所以sun公司目前都没有写出关于树的工具，而关于线性结构的工具就有很多，如ArrayList、LinkedList。TreeSet不属于树，它是一个无序集合，底层的存储并不是使用树，只不过是它的排序使用到了树的一些知识。

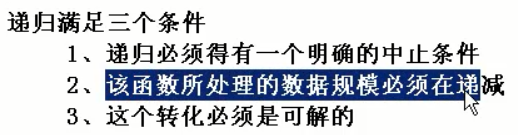
按物理结构分：

# 递归

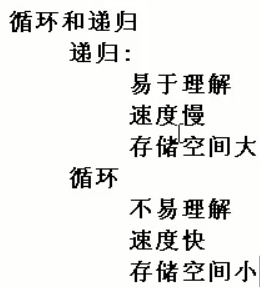








几乎所有的循环都可以使用递归来实现



## 求n的阶乘，如n为5，则求1 X 2 X 3 X 4 X 5 = ?

#include<stdio.h>

int factorial(int n) {

if (n == 1 || n == 2) {

return n;

} else {

return n \* factorial(n -1);

}

}

main() {

int result = factorial(4);

printf("%d\n", result);

}

## 求1 ~ n的和，如n为5，则求1 + 2 + 3 + 4 + 5 = ？

int sum(int n) {

if (n == 1) {

return n;

} else {

return n + sum(n - 1);

}

}

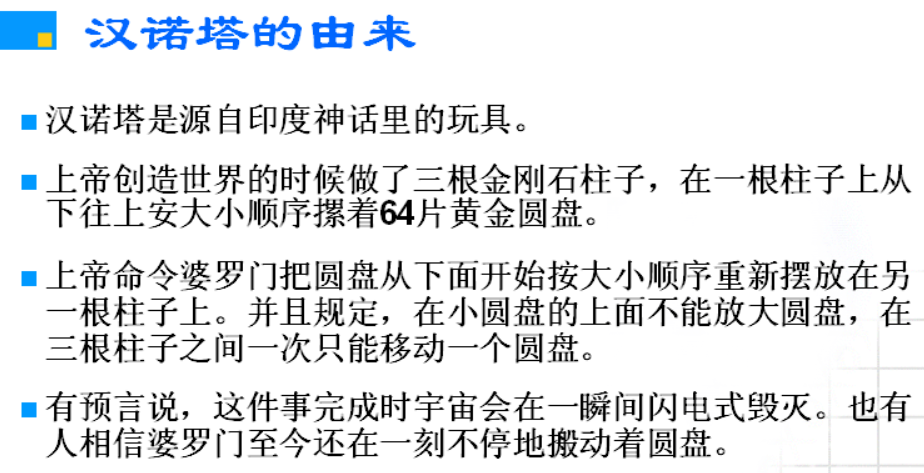
main() {

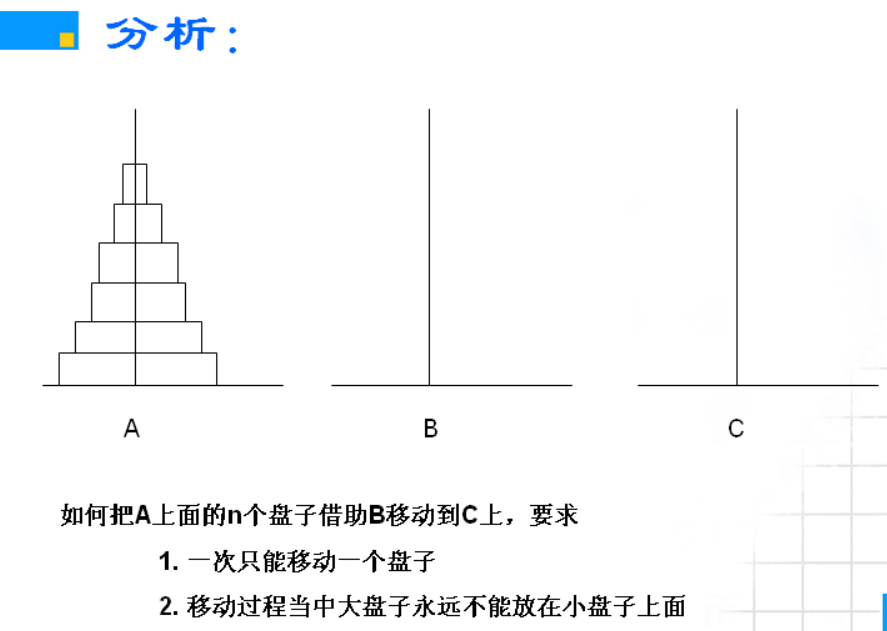
int result = sum(100);

printf("%d\n", result);

}

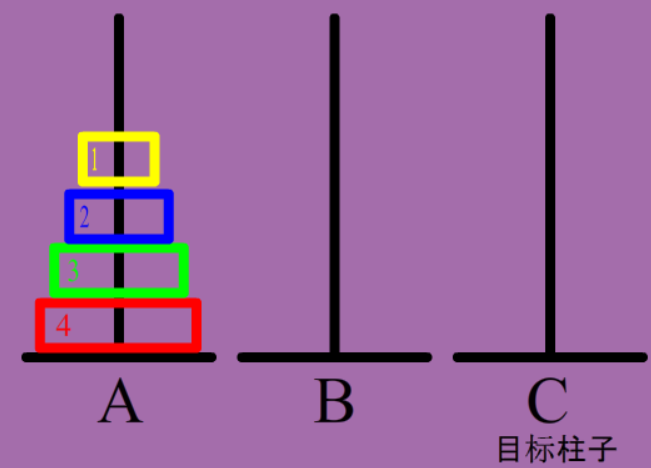
## 汉诺塔

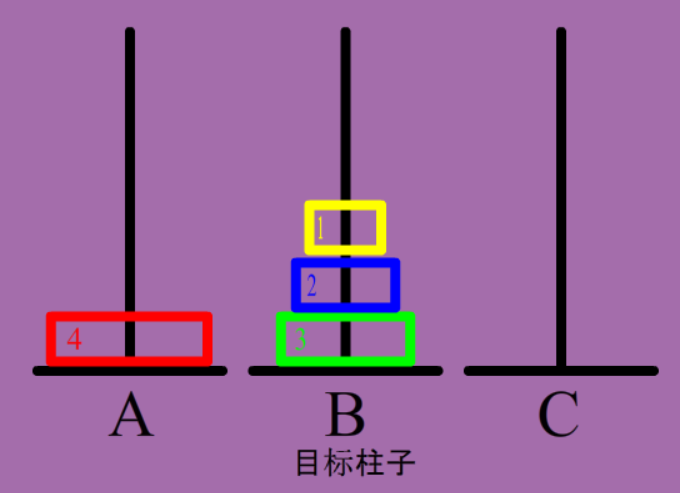


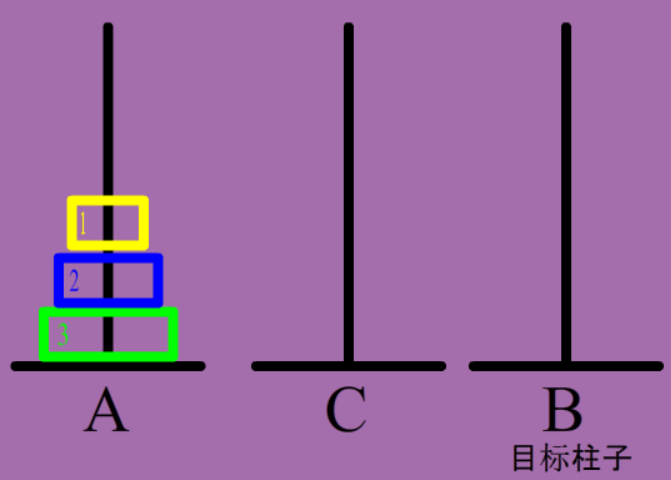
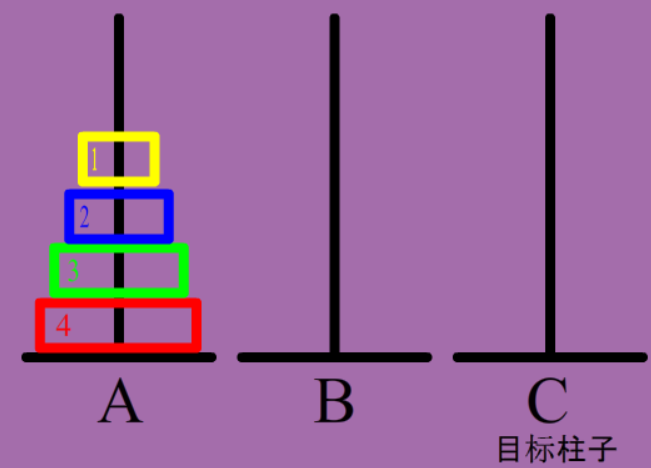


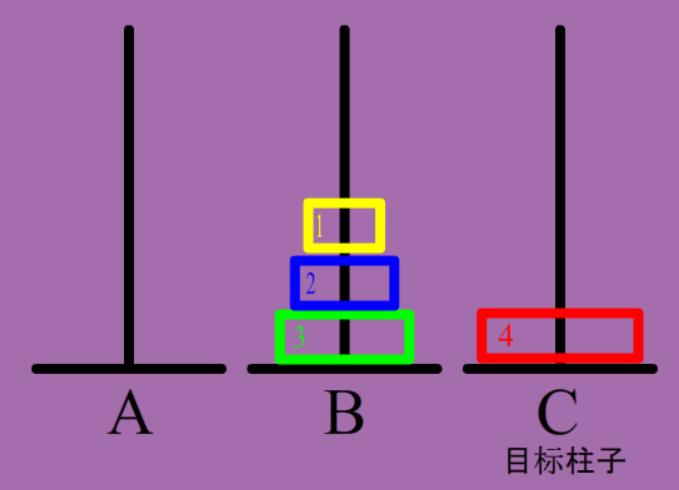
分析:

1、把柱子A中的所有磁盘移到C柱子上，如下图：（此时B是临时柱子，C是目标柱子，这一步是函数入口），

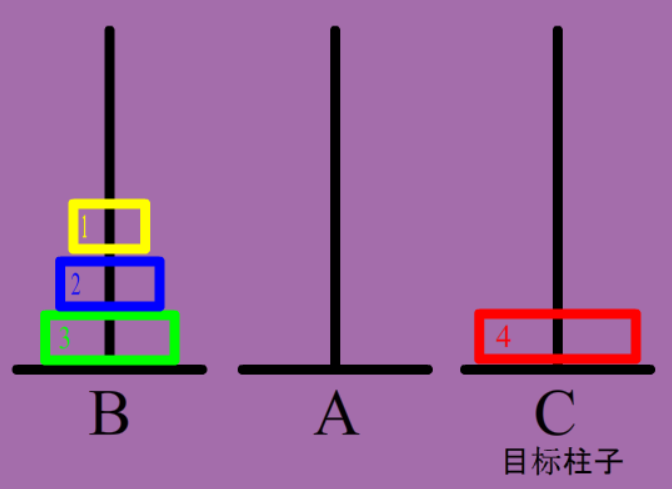
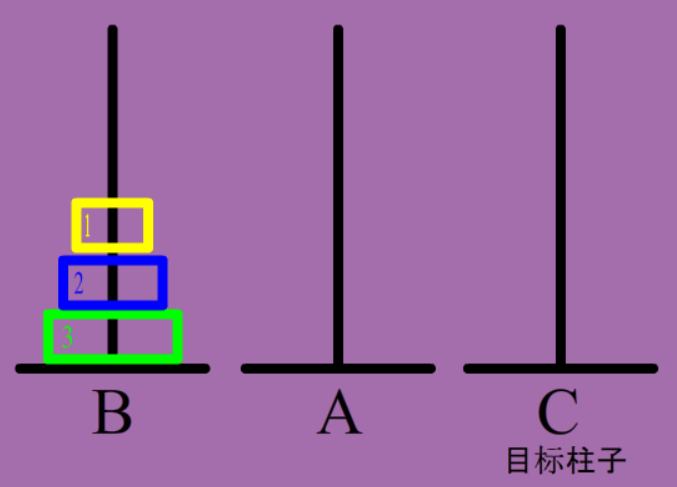
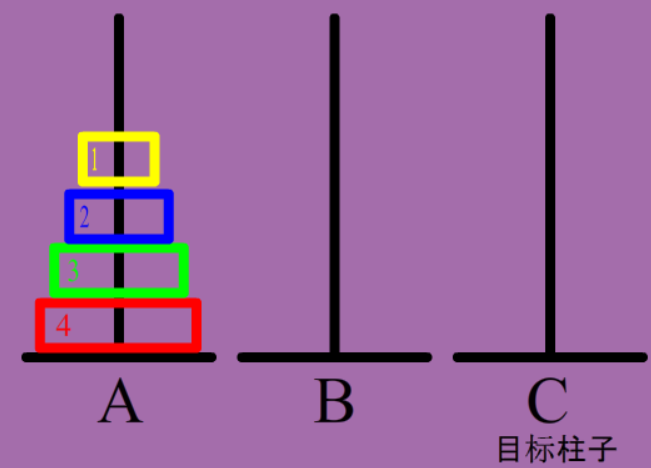


2、要想完成上面的问题，可以先把A中的1、2、3磁盘移到B，如下图：（这一步其实B就变成了目标柱子，C变成了临时柱子，这一步是函数的解决步骤）  
   
 注：上图是把A中的1、2、3移到了B，我们可以把A中的4当成是透明的或者不存在的，再把B和C的位置换一下，这样的效果如下图：

--->与原来问题相同---->   
 A中的所有磁盘要移到B，这又跟步骤1要解决的问题是一样的了，所以这一步可以递归调用

3、A中剩下的一个磁盘肯定是最大的，直接移到C即可，如下图：（这一步C又变成了目标柱子，其实磁盘移到哪个柱子上哪个柱子就是目标柱子，这一步是函数的解决步骤）  
 

4、现在需要把B中的1、2、3移到C柱子上，则C是目标柱子，A就变成了临时柱子，可以把A、B柱子换一下顺序，会发现现在的问题又跟之前一模一样了，所以就可以用递归完成了，如下图：

--->--->与原来问题相同---->

（因为C上的磁盘是最大的，所以可以把C上的磁盘当透明的或者不存在的）

具体代码见：“汉诺塔.c”。

#include<stdio.h>

/\*\*

@n 磁盘数量

@discFromLocation 磁盘所在位置

@discToLocation 磁盘目标位置

@discTempLocation 磁盘临时位置

\*/

void towerOfHanoi(int n, char discFromLocation, char discToLocation, char discTempLocation) {

if (n >= 1) { // 最少要有一个磁盘才需要移到

// 1、把n- 1 个磁盘从 discFromLocation 移到 discTempLocation（递归调用）

towerOfHanoi(n - 1, discFromLocation, discTempLocation, discToLocation);

// 2、把discFromLocation中剩下的最大的磁盘移到discToLocation（具体移动，无需递归，直接输出）

printf("%c -> %c\n", discFromLocation, discToLocation);

// 3、步骤1已经把n- 1个磁盘移到了discTempLocation，所以现在需要把discTempLocation中的磁盘移到discToLocation

towerOfHanoi(n - 1, discTempLocation, discToLocation, discFromLocation);

}

}

main() {

int n;

printf("请输入磁盘的个数：");

scanf("%d", &n);

char discFromLocation = 'A';

char discToLocation ='C';

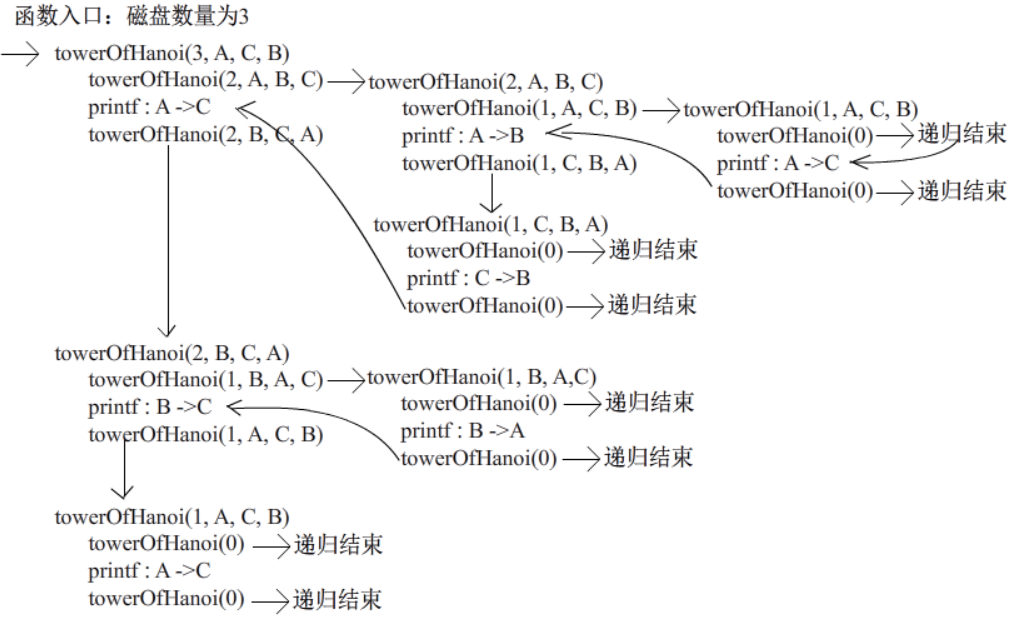
char discTempLocation = 'B';

towerOfHanoi(n, discFromLocation, discToLocation, discTempLocation);

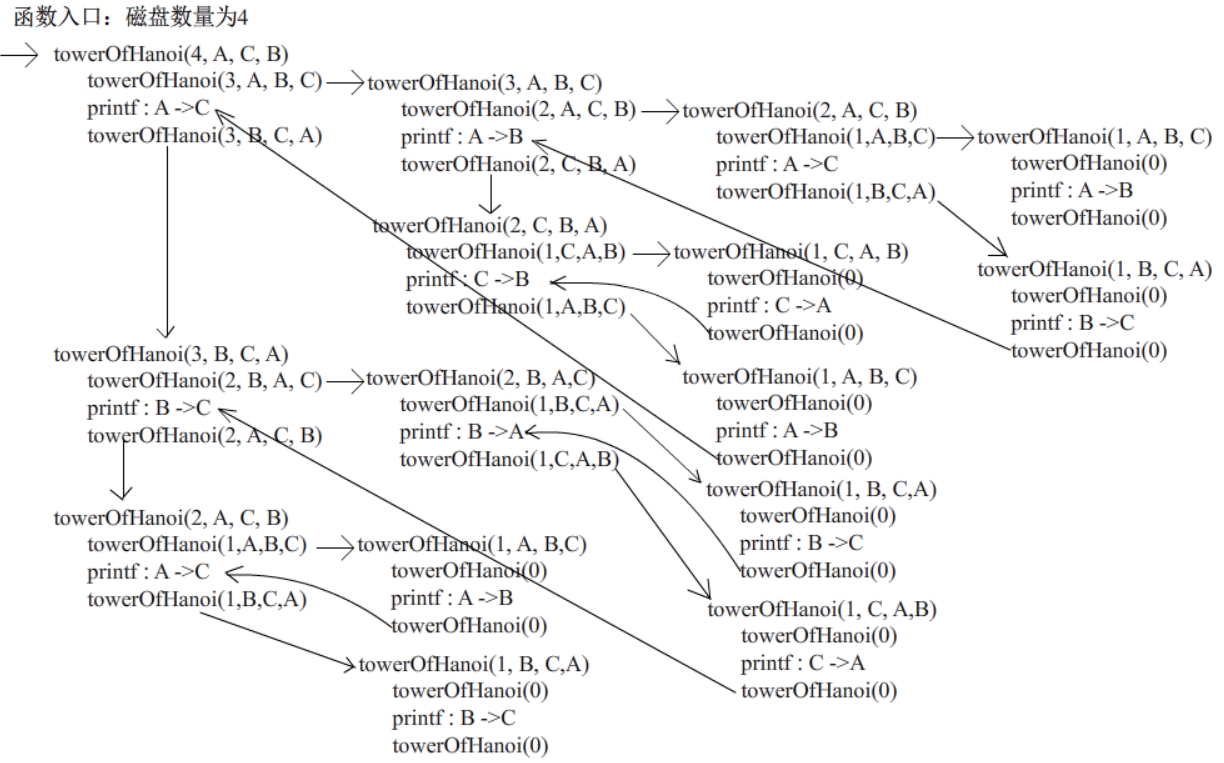
system("pause");

}

假设磁盘数量为3，递归调用的分析图如下：

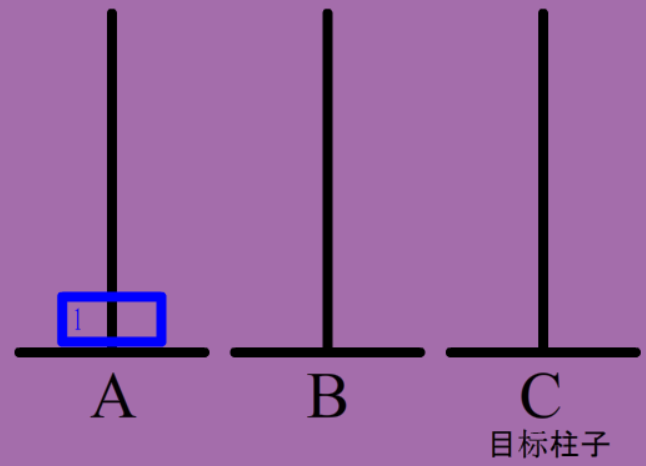


假设磁盘数量为4，递归调用的分析图如下：



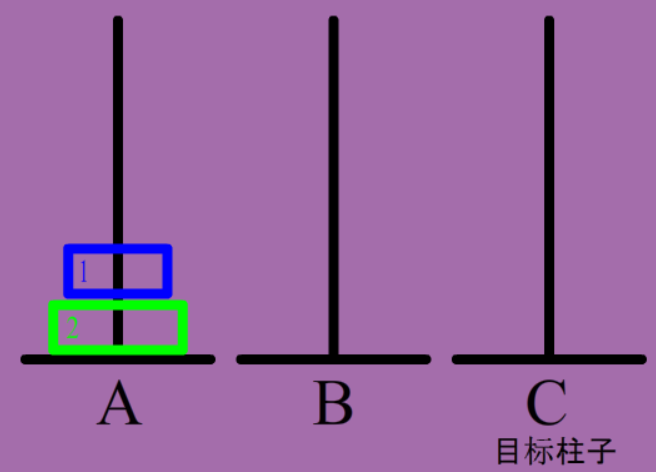
这些递归分析能看得懂，但是看懂了不一定有什么用，所以尽可能的把问题简单化，而不是看这复杂的分析，看完这些复杂的分析你可能还是不能理解为什么递归它就解决了这个问题了，具体原理是怎样的？我们来把问题简单化

案例1：我们假设只有1个磁盘，如下：



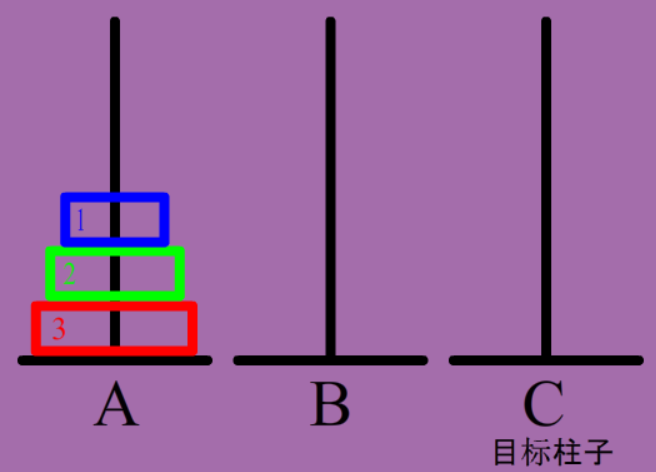
要把磁盘1从A柱子移到C柱子上是很容易的，对应的，把1移到B柱子上跟移到C上是一样的，都没有难度。

案例2：现在我们假设有两个磁盘，如下：

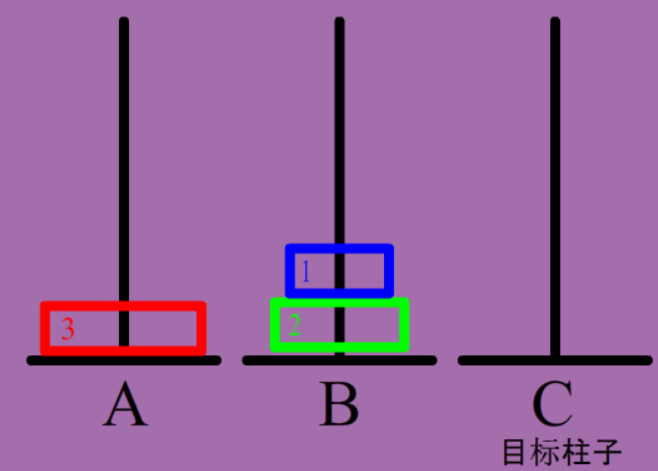
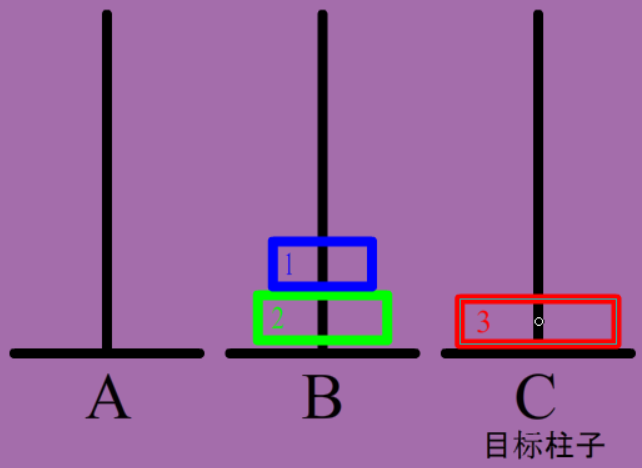


通过刚刚的案例1，我们知道移动一个磁盘到另一个磁盘是非常简单的，A1-->B，A2-->C，B2-->C，完成！！两个磁盘依然是相当的简单！！注：把A上的磁盘移到C或者移到B上都是一样的简单。

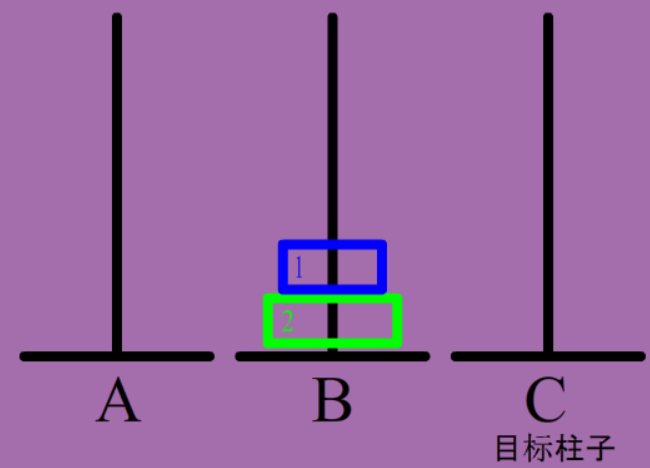
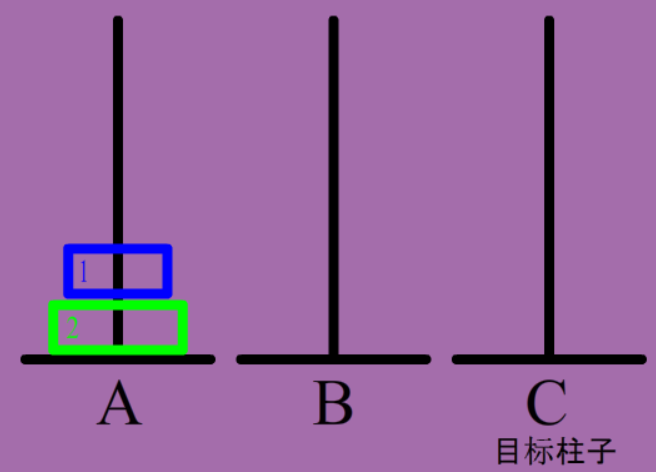
案例3：来难一点，3个磁盘：



通过案例2我们知道把一根柱子上的两个磁盘移到另一根柱子上是很容易的，所以我们可以把A柱子上的1、2移到B，再把3放到C，如下：

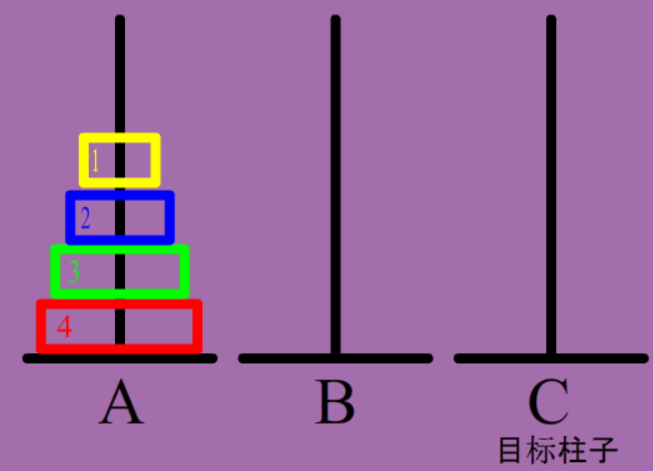
--> 

到这一步的话,相信大家都知道怎么去移动了,这时要把B上的1、２移到Ｃ上，跟之前总共只有两个磁盘时的移动技巧是一样的，或者你可以把Ｃ上的３想象成不存在的，因为３是最大的，其它的任何磁盘都可以放在它的上面，所以可以把３号磁盘想象成不存在，如下图：

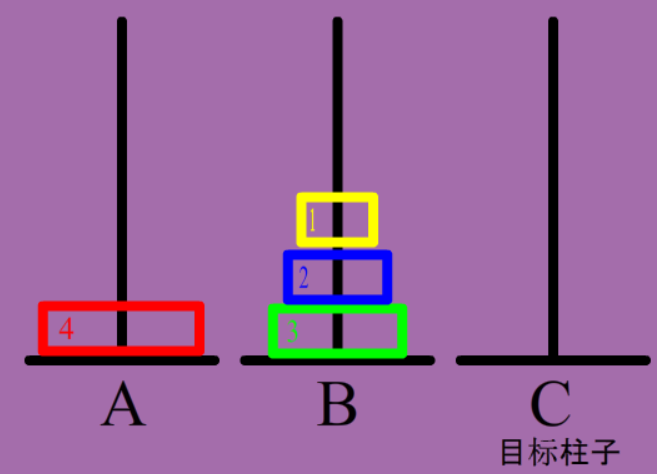
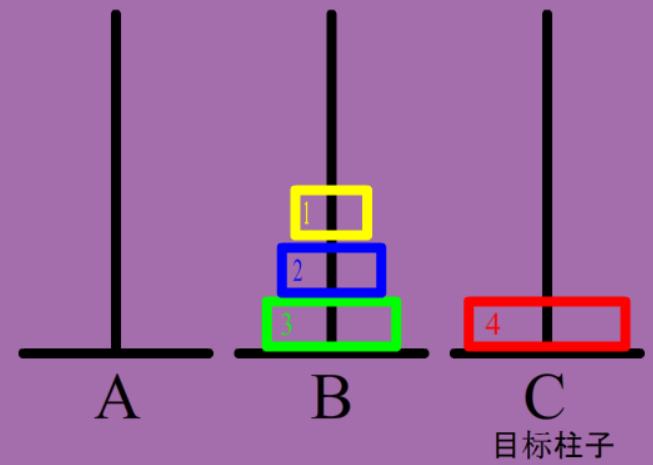
-->对比-->

把Ｂ上的１、２移到Ｃ，跟把Ａ上的１、２移到Ｃ是没有什么区别的。

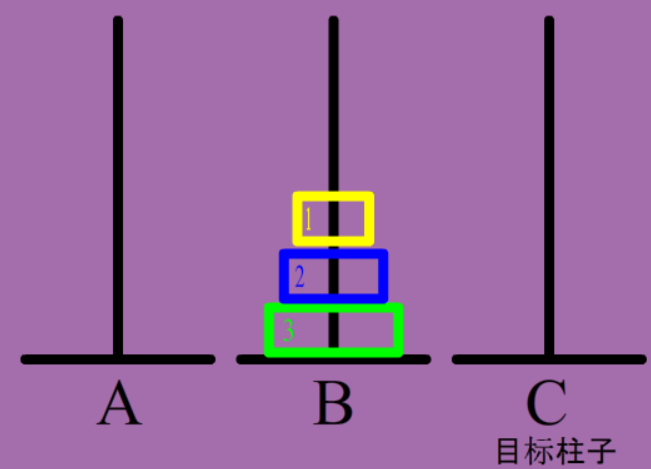
这么去想问题的话，好像把3个磁盘移到另一个柱子也不是难事，把3个磁盘从A移到C或者B也是一样的，没有什么区别。  
案例4：现在再来加点难度，假设有4个磁盘，如下：



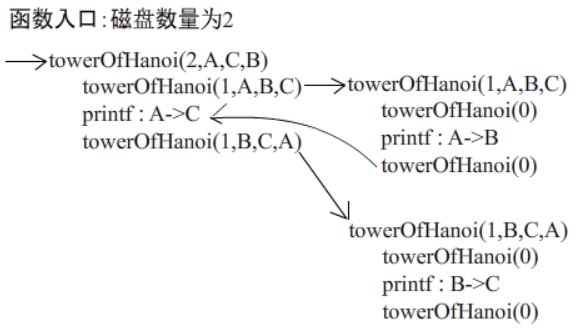
刚刚我们已经分析过了，把3个磁盘移到另一个柱子也是很简单的，所以我们可以把A柱子上的1、2、3移到B上，再把A上的4移到C，如下：

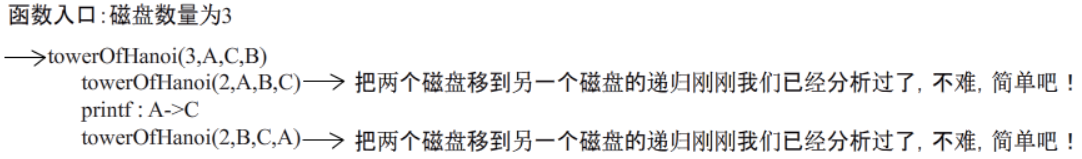
这时C上的4是最大的，任何的磁盘都可以放到它上面，所以我们又可以把磁盘4想象成不存在的，变成下图：

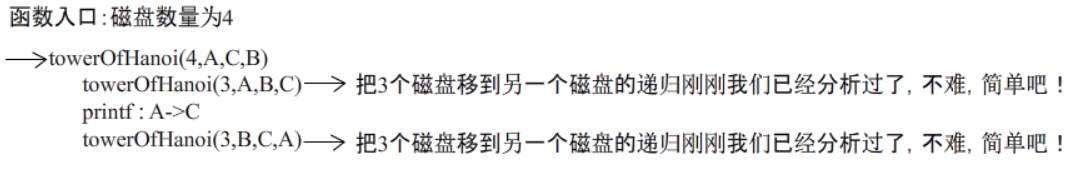


现在把B上的1、2、3移到C上也没有难度了。反过来说就是把4个盘子移到另一根柱子上也不难，现在再假设有5个磁盘，怎么移？我相信大家也不会觉得难了吧！如果我们再假设有6个、7个、8个。。。。不论有多少个磁盘，都是很简单的，但是磁盘一多，我们就没有办法记住前面的所有移动步骤，而计算机可以，这时再来分析递归，还是从简单的说起，假设只有两个磁盘：



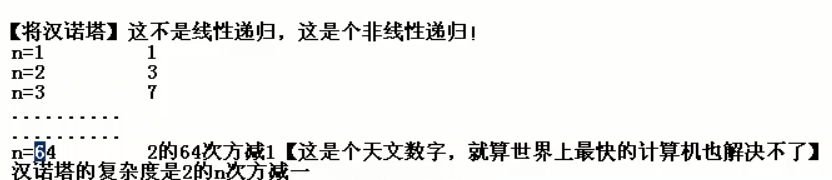
这张递归估计大家就很容易看得懂了。



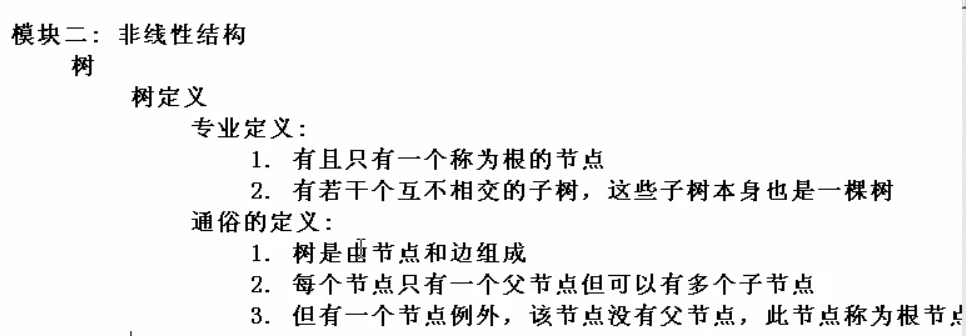


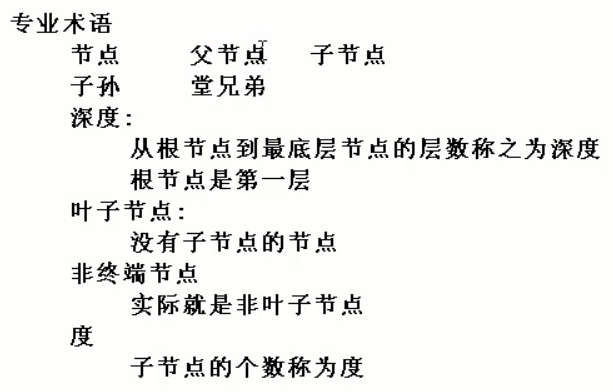
说是这么说简单，其实数量多了，把完整的递归画出来看的话还是很难理解的，这跟我们最开始不使用计算机直接肉眼看，有了技术我们只道移到两个跟移到1个一样简单，移到3个又跟移到两2个差不多，移到4个又跟移到3个差不多，但是要移到64个就难了，所以对于递归的分析图也一样，只要能理解一个数量少的就行了，数量多的画出来肯定也是难以理解的，也没有必要。

注: 磁盘数量不要输入64,因为:



# 树



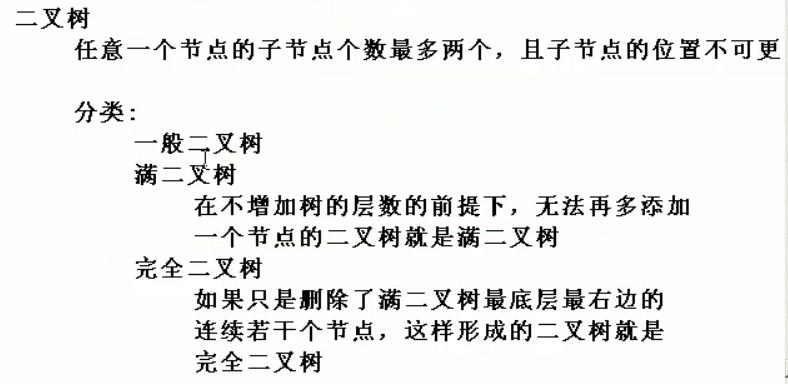


一棵树如果只有根节点，则这个根节点也算是叶子节点

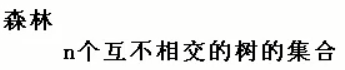
一棵树的度为拥有最多子节点的那个点的度，如有个节点它的子节点最多，有8个子节点，则这整个树的度就是8。

## 树的分类



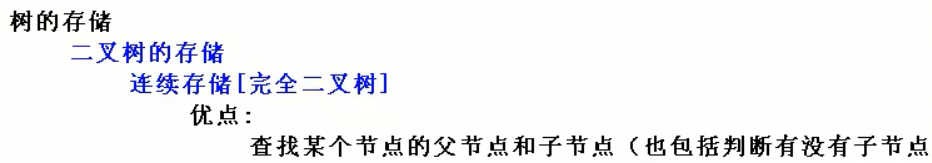


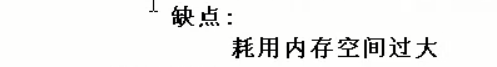
一般树是可以无序的，而二叉树必须是有序的



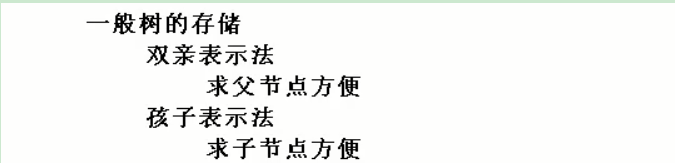
内存是线性的，而树不是，所以不能直接把树存到内存，需要先把树进行转换

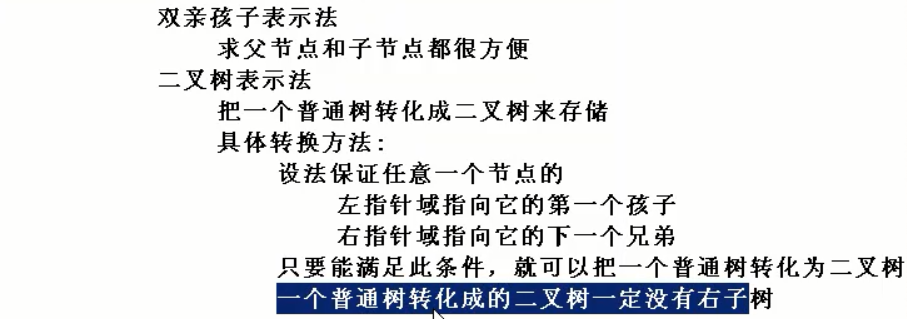
## 树的存储















## 二叉树的操作

