5.数据结构与算法（17道）

5.1数组与链表的区别?

（1）数组的元素个数在定义时就必须确定，且元素的类型必须一致；而链表的元素个数自由，且元素内可以有不同类型的数据。

（2）数组的元素在内存中是按顺序存储的，而链表的元素是随机存储的。

（3）要访问数组的元素可以按下标索引来访问，速度比较快；如果对它进行插入/删除操作的话，就得移动很多元素，所以对数组进行插入/删除操作效率很低。由于链表是随机存储的，如果要访问链表中的某个元素的话，那就得从链表的头逐个遍历，直到找到所需要的元素为止，所以链表的随机访问的效率就比数组要低；链表在插入/删除操作上有很高的效率（相对数组）。一句话总结就是：数组的访问效率高，而链表的插入/删除效率高。

5.2 a = b * 2; a = b / 4; a = b % 8; a = b / 8 * 8 + b % 4 ; a = b * 15;

效率最高的算法?

答案:

a = b * 2	a = b << 1;
a = b / 4	a = b >> 2;
a = b % 8	a = b & 7; // 7 = (0b111)
a = b / 8 * 8 + b % 4	a = ((b >> 3) << 3) + (b & 3); // 3 = 0b11
a = b * 15	a = (b << 4) - b

解读: *、/、%分别可以用<<、>>、&来实现，效率更高。

5.3 C语言程序代码优化方法

（1）选择合适的数据结构与算法；

（2）使用尽量小的数据类型；

（3）使用自加、自减指令；
（4）用移位实现乘除法运算；
（5）求余运算用&（如a=a%8改为a=a&7）；
（6）平方运算用*（如a=pow(a,2.0)改为a=a*a）；
（7）延时函数的自加改为自减；
（8）switch语句中根据发生频率来进行case排序；
（9）减少运算的强度。

5.4时间换空间、空间换时间的例子?

（1）时间换空间：冒泡排序。
（2）空间换时间：快速排序。

5.5什么是满二叉树、完全二叉树、平衡二叉树?

（1）当一个树每一层的结点个数都达到最大时，这个树是满二叉树。
（2）当一个树除了最后一层外其他每一层的结点数都达到最大，且最后一层的叶子结点都靠左排列时，这个树是完全二叉树。满二叉树是一种特殊的完全二叉树。
（3）当且仅当一个树两个子树的高度差不超过1时，这个树是平衡二叉树。

5.6堆和栈的的区别?

数据结构的堆和栈	栈是一种先进后出的数据结构。堆是一种经过排序的树形数据结构（通常是二叉堆），每个结点都有一个值，根结点的值最小或最大，常用来实现优先队列，堆的存储是随意的。
C语言内存分配的堆和栈	栈是向下生长的，栈中分配函数参数和局部变量，其分配方式类似于数据结构中的栈。堆是向上生长的，堆中分配程序员申请的内存空间（一旦忘记释放会造成内存泄漏），其分配方式类似于数据结构中的链表。

5.7快慢指针有哪些应用?

判断链表是否有环	两个指针同时从链表的第一个节点出发，一个指针一次走一步，另一个指针一次走两步，如果走得快的指针追上走得慢的指针，则链表存在环；如果走得快的指针走到链表的末尾（NULL）都没有追上走得慢的指针，则链表不存在环。
找出链表的中间节点	两个指针同时从链表的第一个节点出发，一个指针一次走一步，另一个指针一次走两步，当走得快的指针走到最后一个节点时，走得慢的指针就刚好走

	到链表的中间节点。
删除链表倒数第n的节点	两个指针同时从链表的第一个节点出发，慢指针不动，快指针先走到第n个节点，然后两个指针开始一起走动，每次走一步，当快指针走到最后一个节点时，慢指针就处于链表的倒数第n个节点。
删除排序链表中的重复项	慢指针从第一个节点出发，快指针从第二个节点出发，两个指针一起走动，每次走一步，如果两个指针指向的节点数据相同，则释放快指针指向的节点，然后快指针指向下一个节点……如此循环直到快指针指向末尾（NULL）。

5.8对线性表（7,34,55,25,64,46,20,10）进行散列存储时,若选用H（K）=K %9作为散列函数,则散列地址为1的元素有几个?

答案：4个，分别是：55，64，46，10。

解读：将线性表元素代入散列函数，即可得到散列地址。

5.9写一个程序, 要求功能：求出用1，2，5这三个数不同个数组合的和为100的组合个数。 如：100个1是一个组合，5个1加19个5是一个组合。

（1）最容易想到的算法是暴力解法

思路：设x是1的个数，y是2的个数，z是5的个数，number是组合数，注意到0 <= x <= 100，0 <= y <= 50，0 <= z <= 20。

代码：

```
void count(void)
{
    int x, y, z, number;
    number = 0;
    for (x = 0; x <= 100 / 1; x++)
    {
        for (y = 0; y <= 100 / 2; y++)
        {
            for (z = 0; z <= 100 / 5; z++)
            {
                if (x + 2 * y + 5 * z == 100)
                number++;
            }
        }
    }
```

```
    printf("%d\t", number);
}
```

（2）上述暴力解法的时间复杂度为O(n^3)，程序有些冗余，对其进行优化如下

思路：注意到上面代码的第三个for循环其实不是必要的，因为当1和2的数目确定了之后，加上一定数目的5能不能组成100也就确定了，没必要用for循环一个个去尝试，直接计算即可，优化后时间复杂度变为O(n^2)。

代码：

```
void count(void)
{
    int x, y, z, number;
    number = 0;
    for (x = 0; x <= 100 / 1; x++)
    {
        for (y = 0; y <= 100 / 2; y++)
        {
            if (100 - x - y * 2 >= 0 && (100 - x - y * 2) % 5 == 0) // 判断能否5整除
            number++;
        }
    }
    printf("%d\t", number);
}
```

5.10数组a[N]，存放了数字1 ~ N-1，其中某个数重复一次。写一个函数，找出被重复的数字，时间复杂度必须为O(N)。

思路：数组有N个元素，刚好存放了数字1 ~ N – 1，那么是不是可以将数字与数组的下标对应起来存储，而重复的数字则刚好放在下标为0的位置。一开始假设a[0]是重复的数字，若不是则放到它该放的位置上，换另一个数字作为a[0]，若是则程序结束。如此循环，定能在N次循环内找到重复的数字。（哈希表思维，将元素的值与其存储位置关联起来）

代码：

```
int do_dup(int a[], int N)
{
    int temp;
    // a[0]为监视哨
    while (a[0] != a[a[0]])  // 若两者相等，则说明该数字是重复数字
    {
        temp = a[0];     // 若不相等，则将该数放到以该数为下标的位置上
        a[0] = a[temp];   // 该位置原来的数则放到0为下标的位置上
        a[temp] = temp;
    }
    return a[0];  // 返回重复数字
}
```

5.11写出快速排序的代码。

思路：数组一开始是乱序的，以第一个元素为基准，先将其保存；指针1从后往前寻<=基准的元素，插在基准元素的位置，接着指针2从前往后寻找>基准的元素，插在指针1的位置上；指针1继续往前寻找<=基准的元素，插在指针2的位置上，接着指针2继续往后寻找>基准的元素，插在指针1的位置上；如此循环直到两指针相遇，将基准元素插在相遇的位置，至此，<=基准的元素都在基准元素左边，>基准的元素都在基准元素右边，再递归地对左边的元素进行相同的操作，然后递归地对右边的元素进行相同的操作，即可完成排序，时间复杂度为$O(n\log_2 n)$。（二分思维、递归思维）

代码：

```
void quick_sort(int *num, int start_num, int end_num)
{
    if(start_num < end_num)
    {
        int i = start_num;
        int j = end_num;
        int temp = num[start_num];  // 以第一个元素为基准
        while(i < j)
        {
            while(i < j && num[j] > temp)  // 从后往前寻找比基准小的元素
                j--;
            if(i < j)
                num[i++] = num[j];      // 插在基准元素前面的空位上
            while(i < j && num[i] <= temp) // 从前往后找比基准大的元素
                i++;
            if(i < j)
                num[j--] = num[i];       // 插在基准元素后面的空位上
        }
        num[i] = temp;                  // 基准元素归位
        quick_sort(num, start_num, i - 1);  //递归地对基准元素左边的数据排序
        quick_sort(num, i + 1, end_num);   //递归地对基准元素右边的数据排序
    }
}
```

5.12写出二分查找的代码。

思路：二分查找是对有序数组而言的，那么我们只需要拿中间的元素来跟目标数值比较，如果相等则查找完成；如果中间的元素小于目标数值，那么说明目标元素在右边；如果中间的元素大于目标数值，那么说明目标元素在左边。接着再拿右边或左边的中间元素来比较……如此循环直到找到目标元素/找不到退出循环。时间复杂度为$O(n\log_2 n)$（二分思维）

代码：

```
int binary_search(int array[], int value, int size)
{
    int low = 0;
    int high = size -1;
    int mid;
    while(low <= high)
    {
        mid = (low + high) / 2;     // 二分
        if(array[mid] == value)     // 中间数据是目标数据
            return mid;
        else if(array[mid] < value) // 中间数据比目标数据小
            low = mid + 1;
        else   // 中间数据比目标数据大
            high = mid - 1;
    }
    return -1;
}
```

5.13单向链表逆序。

思路：尾插法，即先将第一个节点的next指针指向NULL，然后依次将第二个节点的next指针指向第一个节点，第三个节点的next指针指向第二个节点……直至最后一个节点，并将指向最后一个节点的指针以头指针返回。

代码：

```
LNODE *link_reverse(LNODE *head)
{
    LNODE *pb, *pt;
    if(head == NULL)
        return head;
    pb = head -> next;
    head -> next = NULL;
    while(pb != NULL)  // 尾插法
    {
        pt = pb -> next;
        pb -> next = head;  // 插入新链表
        head = pb;  // 新链表头指针
        pb = pt;
    }
    return head;  // 返回头指针
}
```

5.14手写：字符串排序问题，时间复杂度O(n)，任意字符串的排序并统计重复的个数，例如输入：$-%-#aeartvDEtGD%!%

输出：!1#1$1%3-2D2E1G1a2e1r1t2v1

思路：字符的ASCII码总共只有128个（0~127），那么我们是不是可以建立一个长度为128的数组并初始化为0，然后遍历每个字符，并以字符的ASCII为下标将该元素+1。这样一来，只需要遍历一遍字符串，就可以将字符串排好序并统计好个数。接下来在遍历一遍这个数组，将对应的字符和个数打印出来即可。（哈希表思维，将元素的值与其存储位置关联起来）

代码：

```
#include <stdio.h>
#include <string.h>
void sort(char a[], int len)
{
    int i;
    char b[128] = {0}, key;
    for(i = 0; i < len; i++)
    {
        key = a[i];  // 将字符对应的ASCII码作为下标
        b[key]++;  // 对应元素+1
    }
    for(i = 0; i < 128; i++)
    {
        if(b[i] != 0)
            printf("%c%d", i, b[i]);  // 按顺序打印出字符跟个数
    }
    printf("\n");
}
```

5.15判断链表是否有环。

思路：双指针法，定义两个指针，同时从链表的头节点出发，一个指针一次走一步，另一个指针一次走两步。如果走得快的指针追上了走得慢的指针，那么链表就是环形链表；如果走得快的指针走到了链表的末尾（fast == NULL）都没有追上第一个指针，那么链表就不是环形链表。

代码：

```
bool IsLoop(NODE *head)
{
    if (head == NULL)
        return false;

    NODE *slow = head -> next;  // 初始时，慢指针从第一个节点开始走1步
    if (slow == NULL)
        return false;

    NODE *fast = slow -> next;  // 初始时，快指针从第一个节点开始走2步
    while (fast != NULL && slow != NULL)  // 当单链表没有环时，循环到链表末尾结束
    {
        if (fast == slow)  // 快指针追上慢指针
```

```
            return true;
        slow = slow -> next;  // 慢指针走一步
        fast = fast -> next;  // 快指针走两步
        if (fast != NULL)
            fast = fast -> next;
    }
    return false;
}
```

5.16已知两个已经从小到大排列的数组，将它们中所有的数组合成一个新的数组，要求新的数组也是按照从小到大排列的。

思路：原来的两个数组已经是顺序排列好的，那么为了减小时间复杂度，我们只需要新建一个数组，接着从后往前比较两个数组的元素的大小，并从后往前填充到新数组即可。时间复杂度为O(n)。

代码：

```
int merge(int *array1, int len1, int *array2, int len2, int *array3)
{
    int len3 = len1 + len2;
    while(len1 > 0 && len2 > 0)
    {
        array3[len3-- - 1] = array1[len1 -1] > array2[len2 - 1] ? array1[len1-
- - 1] : array2[len2-- -1];  // 比较数组元素大小并填充到新数组
    }
    while(len1 > 0)  // 若数组1还有元素则依次填充
    {
        array3[len3-- - 1] = array1[len1-- - 1];
    }
    while(len2 > 0)  // 若数组2还有元素则依次填充
    {
        array3[len3-- - 1] = array2[len2-- - 1];
    }
    return 0;
}
```

5.17写出二叉树的中序遍历代码。

思路：中序遍历，即先遍历左子树，然后访问根节点，最后遍历右子树，在遍历左子树跟右子树时，也是同样按照左、中、右的顺序。前序遍历和后续遍历则分别是中、左、右和左、右、中的顺序。由于每次都是同样的过程，因此可用递归实现。

代码：

```
void in_order_traversal(Node *node)
{
    if(NULL != node -> left)
        in_order_traversal(node -> left);
```

```
    printf("%d\t", node -> data);
    if(NULL != node -> right)
        in_order_traversal(node -> right);
}
```

讨论 评论

帖子还没人回复，快来抢沙发！