C++数据结构阶段性考核试卷

一. 选择题

1. 将长度为n的单链表连接在长度为m的单链表之后,其算法的时间复杂度为()

A. O(m)

B. O(1)

C. O(n)

D. O(m+n)

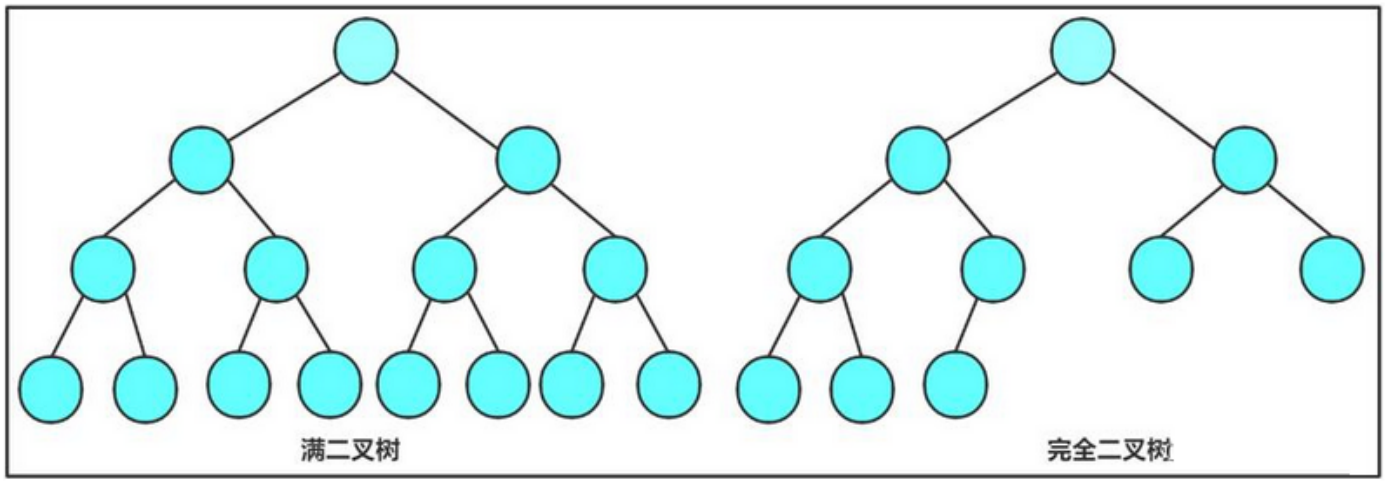
解析思路

> 💡 **答案：A**
>
> 单链表由于需要找到最后一个非空节点，所以需要遍历长度为m的单链表；
>
> 连接的过程只需要修改找到的最后一个非空节点的next指针，指向长度为n的单链表即可，复杂度为O(1)；
>
> 所以总体的时间复杂度就是O(m)。

2. 下面关于二叉树的说法正确的是()

A. 满二叉树是完全二叉树
B. 满二叉树中有可能存在度数为1的节点
C. 完全二叉树是满二叉树
D. 完全二叉树中某个节点可以没有左孩子，只有右孩子

解析思路



答案：A

满二叉树指的是除了最后一层全部是叶子结点，其他层没有叶子结点；

完全二叉树是除了最后一层不满，其他层都是满的，并且最后一层是靠左的（没有左孩子就不会有右孩子）

根据定义可知，满二叉树一定是完全二叉树。

3. 某二叉树共有 399 个结点，其中有 199 个度为 2 的结点，则该二叉树中的叶子结点数为（）

A. 不存在这样的二叉树
B. 200
C. 198
D. 199

解析思路

答案：B

N0 = N2 + 1公式推导：

设Ni表示度为i的节点个数，则节点总数 N = N0 + N1 + N2。

节点个数与节点边的关系： N个节点的树有N-1个边。

边与度的关系：N - 1 = N1 + 2 * N2。

故：N0 + N1 + N2 - 1 = N1 + 2 * N2。

因此，得：N0 = N2 + 1。

根据二叉树的基本性质，对任何一棵二叉树，度为0的结点(即叶子结点)总是比度为2的结点多一个。

题目中度为2的结点为199个，则叶子结点为199+1=200。

4. 在一个长度为n的顺序表中删除第i个元素，要移动_____个元素。如果要在第i个元素前插入一个元素，要后移_____个元素。

A. n-i，n-i+1
B. n-i+1，n-i
C. n-i，n-i
D. n-i+1，n-i+1

解析思路

💡 **答案：A**

温馨提示：题目要求删除的是第i个元素，而不是下标为i的元素。第i个元素的i是从1开始计算的，而下标是从0开始的。

1. 删除第i个元素，则需要将第i+1个元素到第n个元素向前搬移，所以要移动：n - (i+1)+1 = n-i
2. 要在第i个元素前插入一个元素，则需要将第i个元素到第n个元素向后搬移，所以要移动：n - i+1

建议使用画图的方式会更直观明了。

5. 排序的方法有很多种，（）法是基于选择排序的一种方法，是完全二叉树结构的一个重要应用。

A. 快速排序
B. 插入排序
C. 归并排序
D. 堆排序

解析思路

💡 **答案：D**

首先思考选择排序的思想是：找到最大或者最小的元素，放在数组的最前或最后；

然后根据完全二叉树用于排序就是堆了，堆排序也符合选择排序的思想，所以就是堆排序。

6. 以下属于链表的优点的是（）

A. 用数组可方便实现
B. 插入操作效率高
C. 不用为节点间的逻辑关系而增加额外的存储开销
D. 可以按元素号随机访问

解析思路

💡 **答案：B**

链表的优点：

1. 插入的效率比较高，时间复杂度为O(1)
2. 按需申请空间，不涉及到扩容

链表的缺点：

1. 不支持随机访问
2. 存储空间不连续，不同的节点通过指针链接起来的，会增加额外的存储开销。

链表是每个节点维护next指针，来实现逻辑上的顺序，由于物理上并不连续，所以无法使用索引访问。
由上可知，A，C，D是错误的，链表插入是O(1)的时间复杂度，所以插入操作效率高。

7. 已知二叉树的前序序列为BCDEFAG，中序序列为DCFAEGB，请问后序序列为()

A. DAFEGCB
B. DAEGFCB
C. DAFGECB
D. DAEFGCB

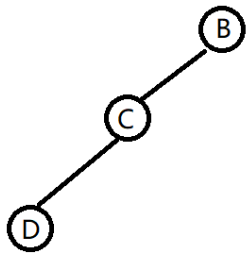
解析思路

①. 通过前序序列确定根节点，并且通过中序序列划分左右子树。由中序序列的划分情况判断，根节点只有左子树，没有右子树。

前序序列：BCDEFAG　　中序序列：DCFAEGB

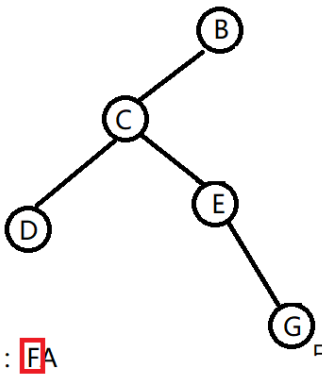
②. 通过前序序列确定B左子树的节点，并且通过中序序列划分左右子树。由中序序列的划分情况判断，C节点的左子树为D，右子树为FAEG。

前序序列：CDEFAG　　中序序列：DCFAEG



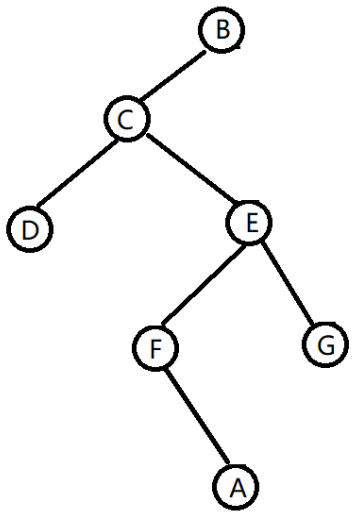
③. 分析C的右子树：通过前序序列确定C右子树的节点，并且通过中序序列划分左右子树。由中序序列的划分情况判断，E节点的左子树为FA，右子树为G。

前序序列：EFAG　　中序序列：FAEG



④. 分析E的左子树：通过前序序列确定E左子树的节点，并且通过中序序列划分左右子树。由中序序列的划分情况判断，F的右子树为A，左子树为空。

前序序列：FA　　中序序列：FA



答案：C

这道题的关键在于如何根据树的前序和中序序列画出二叉树，只要画出二叉树就好办了。

概念：

前序遍历：根节点，左子树，右子树。
中序遍历：左子树，根节点，右子树。
后序遍历：左子树，右子树，根节点。

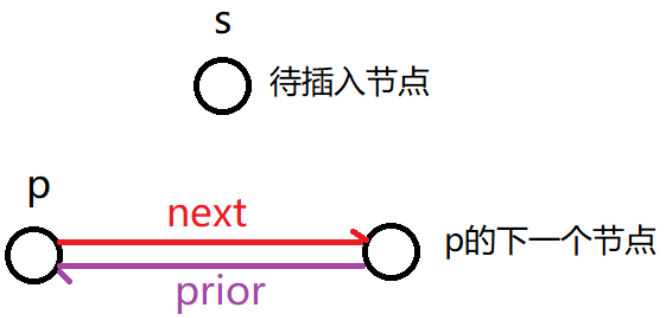
步骤如下：

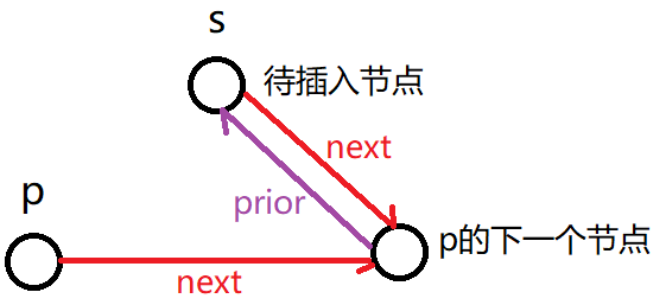
1.已知前序序列为BCDEFAG，所以可知根节点为B且B的左子节点为C。
2.已知中序序列为DCFAEGB，所以D,C,F,A,E,G都属于左子树，根节点B无右子树。
3.C为左子树的根节点，由中序序列知D为C的左子节点，F,A,E,G属于C的右子树
4.由此规律可递归推出二叉树的结构

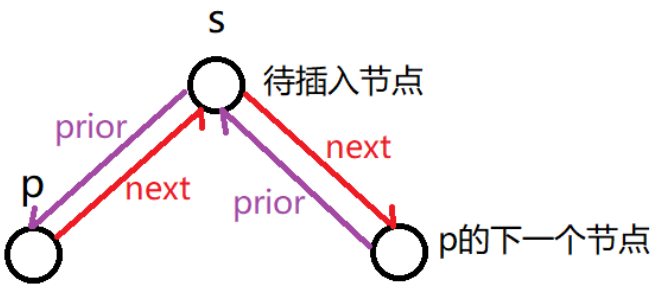
8. 对于双向循环链表，每个结点有两个指针域next和prior，分别指向前驱和后继。在p指针所指向的结点之后插入s指针所指结点的操作应为()

A. p->next = s; p->next ->prior = s; s ->prior = p; s->next = P->next;
B. s->prior = p; s->next = p ->next; p ->next = s; p->next ->prior = s;
C. p ->next = s; s ->prior = p; p->next ->prior =s; s ->next = p ->next;
D. s->prior = p; s->next =p->next; p->next ->prior = s; p ->next = s;

解析思路







答案：D

s->prior=p; //把p赋值给s的前驱

s->next=p->next; //把p->next赋值给s的后继

p->next->prior=s; //把s赋值给p->next的前驱

p->next=s; //把s赋值给p 的后继

顺序：先搞定插入结点的前驱和后继，再搞定后结点的前驱，最后解决结点的后继。

9. 借助于栈输入A、B、C、D四个元素（进栈和出栈可以穿插进行），则不可能出现的输出是（ ）

A. DCBA

B. ABCD

C. CBAD

D. CABD

解析思路

答案：D

A：A B C D依次入栈，再依次出栈，则是A选项的输出。

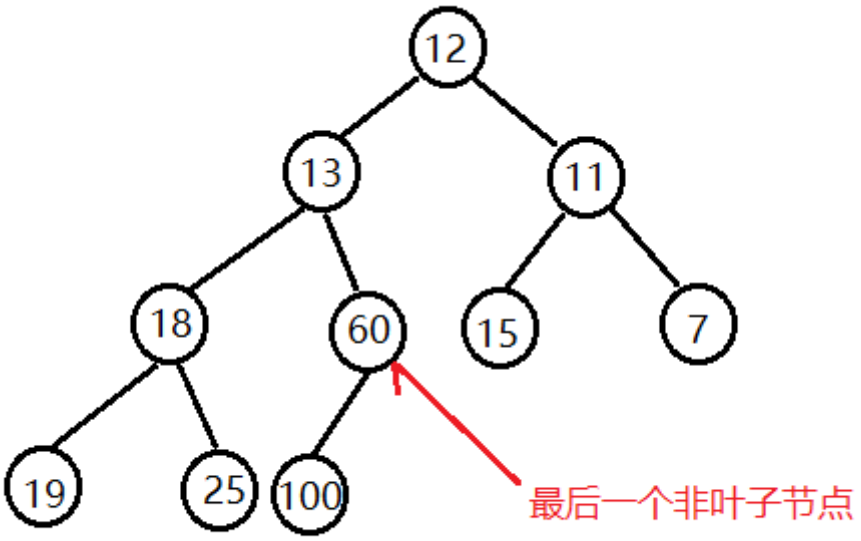
B：A入栈，再出栈；B入栈，再出栈；C入栈，再出栈；D入栈，再出栈；则是B选项的输出。

C：A B C依次入栈，再依次出栈。D再入栈，再出栈。则是C选项的输出。
D选项，ABC入栈后，C先出栈，下一个出栈的元素只能是B，A还在B后面，所以CAB这个出栈顺序有误，D是错的

10. 对于序列{ 12,13,11,18,60,15,7,19,25,100 }，用筛选法建堆，应该从值为（）的数据开始建初始堆。

A. 100
B. 12
C. 60
D. 15

解析思路



答案：C

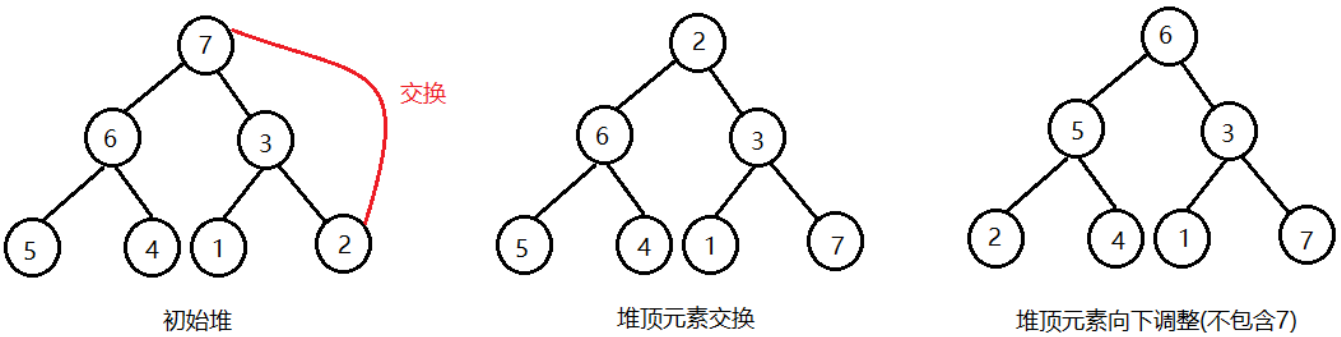
筛选法建堆：就是从最后一个非叶子节点开始向下调整建堆。这里其实就是要计算最后一个非叶子节点的数据。

即：从最后一个结点的父亲结点开始，所以是n/2。

11. 将整数数组（7-6-3-5-4-1-2）按照堆排序的方式进行升序排列，请问在第一轮排序结束之后，数组的顺序是()

A. 1-2-3-4-5-6-7
B. 2-6-3-5-4-1-7
C. 6-5-3-2-4-1-7
D. 5-4-3-2-1-6-7

解析思路



答案：C

堆排序的原理：

每次将堆顶元素和当前堆结构的最后一个元素进行交换，从而将当前堆中的最值交换到最后，从而将数据进行排序。

不同的顺序排序应该建立什么堆：

排升序需要建立大堆，排降序需要建立小堆。

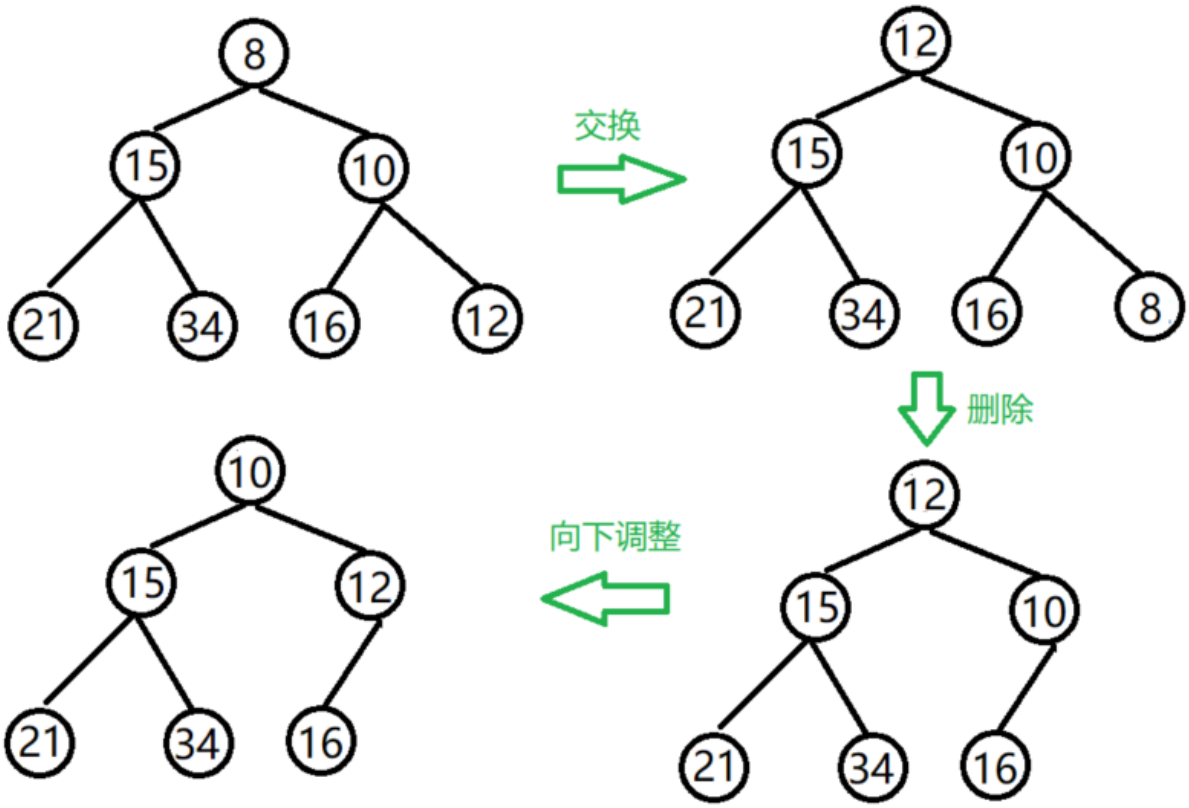
12. 已知小根堆为8,15,10,21,34,16,12，删除关键字8之后需重建堆，最后的叶子节点为()

A. 34
B. 21
C. 16
D. 12

解题思路

答案：C

删除堆顶元素的思路：先将堆顶元素和当前堆结构的最后一个元素交换，堆有效元素个数减一，从而实现堆顶元素的删除。但是交换后的堆顶元素还需要进行向下调整的操作，从而保持现有堆的性质。



13. 若需在O(nlog n)的时间内完成对数组的排序，且要求排序是稳定的，则可选择的排序方法是（）

A. 快速排序
B. 堆排序
C. 归并排序
D. 直接插入排序

解题思路

答案：C

排序稳定指的是相同大小的元素排序后相对位置和排序前是相同的。

比如初始排序序列： 2 1 3 3 5 7 6，排序完成后两个相同元素3的相对位置没有发生改变，则认为该排序算法是稳定的。

快速排序不稳定，时间复杂度也不符合；

堆排序时间复杂度符合，但是不稳定；

归并排序时间复杂度O(nlogn)，稳定，正确；
直接插入排序时间复杂度O(n^2)，稳定。

14. 下列算法中不属于稳定排序的是()

A. 插入排序
B. 冒泡排序
C. 快速排序
D. 归并排序

解题思路

💡 **答案：C**
概念题，快速排序不稳定，其余都是稳定的

15. 以30为基准，设一组初始记录关键字序列为 (30,15,40,28,50,10,70)，则第一趟快速排序结果为()

A. 10，28，15，30，50，40，70
B. 10，15，28，30，50，40，70
C. 10，28，15，30，40，50，70
D. 10，15，28，30，40，50，70

解题思路

💡 **答案：B**

咱们上课老师讲的分割方法有3种。一个序列按照不同分割方式都能达到分割的效果。但大部分情况下都是挖坑法，因为挖坑法是对hore方法的改进，如果挖坑法不行，在尝试hore方法，以及双指针的方法思路。

下面的思路是使用挖坑法：

初始化左指针为1，右指针为6

从右指针开始比较，右指针自减，到10的时候发现比30小，两者交换得到：
10,15,40,28,50,30,70

从左指针开始比较，左指针自增，到40的时候发现比30大，两者交换得到：
10,15,30,28,50,40,70

从右指针开始比较，右指针自减，到28的时候发现比30小，两者交换得到：
10,15,28,30,50,40,70

此时完成第一趟排序，已经满足30的左边都比30小，右边都比30大

二. 编程题

1. 左叶子之和：左叶子之和_牛客题霸_牛客网

解题思路：

1. **左叶子节点的定义：**如果当前节点的左子树根节点不为空，且它的左右子树均为空。
2. 计算左叶子节点之和，可以通过向下递归的方式找到左子树根节点不为空，且左子树根节点的左右子树均为空的节点，并且将其val值进行累加。

代码+注释：

```
int sumOfLeftLeaves(struct TreeNode* root ) {
    //如果root为NULL，则直接返回0
    if(root == NULL)
        return 0;

    int sum = 0;
    //如果左子树根节点不为空，且左子树根节点的左右子树均为空。则进行结果累加
    if(root->left && root->left->left == NULL && root->left->right == NULL)
        sum += root->left->val;

    //递归遍历查找左右子树中，满足左叶子节点性质的节点，并进行累加
    sum += sumOfLeftLeaves(root->left) + sumOfLeftLeaves(root->right);
    //返回sum结果，递归过程中是向上层交付计算结果
    return sum;
}
```

2. 另一棵树的子树：572. 另一棵树的子树 - 力扣（LeetCode）

解题思路：

1. 判断subroot是否为root的子树，需要判断subroot是否和root中的某一棵子树相同。
2. 所以该题目划分为2部分：
   a. 判断两棵树是否相同，递归遍历两棵树的节点进行比较判断。
   b. 递归遍历root中的每一棵树和subroot进行比较，判断两棵树是否相同。

代码+注释：

```
//判断两棵树是否相同：
bool isSameTree(struct TreeNode* p, struct TreeNode* q){
    if(p == NULL && q == NULL)
        return true;
     if(p == NULL || q == NULL)
        return false;
    //如果p和q两个节点值相等，则继续递归判断左右子树中两棵树的节点值是否相等
    if(p->val == q->val)
        return  isSameTree(p->left, q->left) && isSameTree(p->right, q->right);
    return false;
}


//判断是否为另一棵树的子树：
bool isSubtree(struct TreeNode* root, struct TreeNode* subroot){
    if(root == NULL)
        return false;
    if(isSameTree(s, t))
        return true;
    //递归判断左右子树和root是否相等，其中一棵树与root相等，整个结果都为true，所以这里是或的关系
    return isSubtree(root->left,subroot) || isSubtree(root->right, subroot);
}

```

3. 链表的回文结构：链表的回文结构_牛客题霸_牛客网

解题思路：

💡 此题可以先找到中间节点，然后把后半部分逆置，最近前后两部分一一比对，如果节点的值全部相同，则即为回文。

1. 先通过快慢指针的方式，找到链表中间节点。
2. 再将后半部分链表进行反转。
3. 最后将链表前半部分和反转后的后半部分链表逐节点进行比较判断。

代码+注释：

```
bool chkPalindrome(ListNode* A) {
```

```
    //如果A为空，或只有一个节点，则直接返回true
    if (A == NULL || A->next == NULL)
        return true;
    ListNode* slow, *fast,;
    slow = fast = A;
    //1. 通过快慢指针的方法，找到链表中间节点
    while (fast && fast->next)
    {
        //慢指针走一步，快指针走两步
        slow = slow->next;
        fast = fast->next->next;
    }

    //2. 链表后半部分逆置，三指针的方式实现。
    ListNode* prev, *cur, *nxt;
    prev = NULL;
    cur = slow;
    while (cur)
    {
        //记录cur的下一个节点
        nxt = cur->next;
        //实现反转，将cur的next指向prev前一个节点
        cur->next = prev;
        //更新prev和cur，以此向后进行遍历反转
        prev = cur;
        cur = nxt;
    }

    //3. 逐节点进行比对
    while (A && prev)
    {
        //如果两个节点值不相等，则直接返回false
        if (A->val != prev->val)
            return false;
        //更新指针，继续向后遍历
        A = A->next;
        prev = prev->next;
    }
    return true;
}
```

4. 归并排序的实现

解题思路：

💡 **归并排序是采用分治法的一个非常典型的应用。其基本思想是：将已有序的子序合并，从而得到完全有序的序列，即先使每个子序有序，再使子序列段间有序。**

递归方式实现：

归并排序，从其思想上看就很适合使用递归来实现，并且用递归实现也比较简单。期间我们需要申请一个与待排序列大小相同的数组用于合并过程两个有序的子序列，合并完毕后再将数据拷贝回原数组。

非递归方式实现：

归并排序的非递归算法并不需要借助栈来完成，我们只需要控制每次参与合并的元素个数即可，最终便能使序列变为有序。

代码+注释：

```
//递归方式：
void _MergeSort(int* a, int begin, int end, int* tmp)
{
    if (begin >= end)
        return;
    //计算中间位置的下标
    int mid = (begin + end) / 2;

     //1.通过向下递归的方式，划分子数组。
    _MergeSort(a, begin, mid, tmp);
    _MergeSort(a, mid+1, end, tmp);


    //2.将划分后的子数组进行合并
    int begin1 = begin, end1 = mid;
    int begin2 = mid + 1, end2 = end;
    int i = begin;
    while (begin1 <= end1 && begin2 <= end2)
    {
        if (a[begin1] < a[begin2])
        {
            tmp[i++] = a[begin1++];
        }
        else
        {
            tmp[i++] = a[begin2++];
        }
    }
    while(begin1 <= end1)
    {
```

```
        tmp[i++] = a[begin1++];
    }

    while (begin2 <= end2)
    {
        tmp[i++] = a[begin2++];
    }

    //3.将合并后的tmp数组部分内容，拷贝到原始数组a中
    memcpy(a + begin, tmp + begin, sizeof(int) * (end - begin + 1));
}

void MergeSort(int* a, int n)
{
    int* tmp = (int*)malloc(sizeof(int) * n);
    if (tmp == NULL)
    {
        perror("malloc fail");
        return;
    }
     _MergeSort(a, 0, n - 1, tmp);
    free(tmp);
}

//非递归方式：
void MergeSortNonR(int* a, int n)
{
    int* tmp = (int*)malloc(sizeof(int) * n);
    if (tmp == NULL)
    {
        perror("malloc fail");
        return;
    }
    int gap = 1;
    while (gap < n)
    {
        for (size_t i = 0; i < n; i += 2 * gap)
        {
            int begin1 = i, end1 = i + gap - 1;
            int begin2 = i + gap, end2 = i + 2 * gap - 1;
            if (end1 >= n || begin2 >= n)
            {
                //区间1右边界超出范围或者区间2左边界超出范围，当前一轮循环结束
                break;
            }
            if (end2 >= n)
            {
```

```
78                  //区间1范围符合要求，区间2右边界超过范围，修正区别2范围为[begin2,n-
    1]
79                  end2 = n - 1;
80              }
81
82              //合并有序数组
83              int j = begin1;
84              while (begin1 <= end1 && begin2 <= end2)
85              {
86                  if (a[begin1] < a[begin2])
87                  {
88                      tmp[j++] = a[begin1++];
89                  }
90                  else
91                  {
92                       tmp[j++] = a[begin2++];
93                  }
94              }
95
96              while (begin1 <= end1)
97              {
98                  tmp[j++] = a[begin1++];
99              }
100
101             while (begin2 <= end2)
102             {
103                 tmp[j++] = a[begin2++];
104             }
105
106             //临时数组中有效区间范围导入原数组有效区间
107             memcpy(a + i, tmp + i, sizeof(int) * (end2-i+1));
108         }
109         gap *= 2;
110     }
111     free(tmp);
112 }
```