
实验课程	数据结构实验	成绩	
实验项目	实验四　查找和排序算法实现	指导老师	

一、 实验目的：

1、 领会折半查找的过程和算法设计；

2、 领会二叉排序树的定义，二叉树排序树的创建、查找和删除过程及其算法设计；

3、 领会快速排序的过程和算法设计；

4、 领会堆排序的过程和算法设计；

5、 掌握二路归并排序算法及其应用。

二、 使用仪器、器材

微机一台

操作系统：WinXP

编程软件：C/C++编程软件

三、 实验内容及原理

1、教材 P362 实验题 2：实现折半查找的算法

编写一个程序 exp9-2.cpp，输出在顺序表（1，2，3，4，5，6，7，8，9，10）中采用折半查找方法查找关键字 9 的过程。

2、教材 P362 实验题 4：实现二叉排序树的基本运算算法

编写一个程序 bst.cpp,包含二叉排序树的创建、查找和删除算法，再次基础上编写 exp9-4.cpp 程序完成以下功能。

（1） 由关键字序列（4，9，0，1，8，6，3，5，2，7）创建一棵二叉排序 bt 并以括号表示法输出。

（2） 判断 bt 是否为一棵二叉排序树。

（3） 采用递归和非递归两种方法查找关键字为 6 的结点，并输出其查找路径。

（4） 分别删除 bt 中关键字为 4 和 5 的结点，并输出删除后的二叉排序。

3、实现快速排序算法

编写一个程序 exp4-3.cpp 实现快速排序算法，用相关数据进行测试并输出各趟的排序结果。

4、教材 P397 实验题 7：实现堆排序算法

编写一个程序 exp10-7.cpp 实现堆排序算法，用相关数据进行测试并输出各趟的序结果。

5、实现二路归并排序算法。

编写一个程序 exp4-5.cpp，采用二路归并排序方法，对有 n 个记录的待排序序列（例如 8 个记录的序列{8, 2, 21, 34, 12, 32, 6, 16, 11, 5}）进行排序，要求（1）写出采用二路归并排序方法进行排序的过程；（2）写出二路归并排序算法程序；（3）给出二路归并排序算法的时间复杂度和稳定性。

6、教材 P397 题 11：实现英文单词按字典序排列的基数排序算法

编写一个程序 exp10-11.cpp，采用基数排序方法将一组英文单词按字典序排列。假设单词均由小写字母或空格构成，最长的单词 MaxLen 个字母，用相关数据进行测试并输出各趟的排序结果。

四、 实验过程原始数据记录

第 1 题：

```
// exp9-2.cpp

#include <iostream>
using namespace std;
#define ElemType int

typedef struct
{
    ElemType* elem;
    int tableLen;
}SSTable;
void initSSTable(SSTable& s, int len)
{
    s.tableLen = len;
    s.elem = new ElemType[s.tableLen];
    for (int i = 0; i < s.tableLen; i++)
        s.elem[i] = i + 1;
}
```

```cpp
int BinarySearch(SSTable L, ElemType key)
{
    // 初始化下标指针
    int low = 0, high = L.tableLen - 1, mid = -1;
    cout << "折半查找算法开始" << endl;
    cout << "目前low为" << low << "，high为" << high << endl;
    while (low <= high)
    {
        mid = (low + high) / 2;
        cout << "目前mid为" << mid << endl;
        if (key == L.elem[mid])
        {
            cout << "在位置" << mid << "找到" << key << "，算法结束" << endl;
            return mid;
        }
        else if (key > L.elem[mid])
        {
            cout << "在中间位置" << mid << "没有找到，且查找元素" << key << ">" <<
L.elem[mid];
            low = mid + 1;
            cout << "，更新low为" << low << endl;
        }
        else
        {
            cout << "在中间位置" << mid << "没有找到，且查找元素" << key << "<" <<
L.elem[mid];
            high = mid - 1;
            cout << "，更新high为" << high << endl;
        }
    }
    // 没找到
    return -1;
}

int main()
{
    // 初始化顺序表
    SSTable s;
    initSSTable(s, 10);
    BinarySearch(s, 9);
    return 0;

}
```

第 2 题：

// bst.h

```
#pragma once
#define ElemType int

struct BiNode
{
    ElemType elem;
    BiNode* lchild, * rchild;
};

void insertBST(BiNode*& bt, ElemType key);     // 插入节点
void bracketPrint(BiNode* T);                  // 括号表示法输出
bool isBiSort(BiNode* T);                      // 判断是否是二叉排序树
void BiSortTree(BiNode*& bt, ElemType elem[], int len);   // 建立二叉排序树
BiNode* searchBST(BiNode* bt, ElemType key);          // 搜索（递归）
BiNode* searchBST_NoR(BiNode* bt, ElemType key);      // 搜索（非递归）
void deleteNode(BiNode*& bt);                  // 删除节点
bool deleteBST(BiNode*& bt, ElemType key);     // 删除值为 key 的节点
```

// bst.cpp

```
#include <iostream>
#include "bst.h"
using namespace std;

// 插入节点
void insertBST(BiNode*& bt, ElemType key)
{
    // 当树根为空时
   // 直接建立(子)树
    if (!bt)
    {
        bt = new BiNode;
        bt->elem = key;
        bt->lchild = NULL;
        bt->rchild = NULL;
    }
    // 如果树根不空，则递归往下，直到空
    else
```

```
    {
        if (key < bt->elem)
            insertBST(bt->lchild, key);
        else
            insertBST(bt->rchild, key);
    }
}
// 括号表示法输出
void bracketPrint(BiNode* T)
{
    if (!T) // 递归终止条件
        return;

    cout << T->elem;

    // 只有在左右孩子至少有一个存在时才访问
    if (T->lchild || T->rchild)
    {
        cout << "(";        // 要访问孩子时添加（
        bracketPrint(T->lchild);   // 递归调用
        cout << ",";        // 访问完毕左孩子要访问右孩子时添加,
        bracketPrint(T->rchild);   // 递归调用
        cout << ")";        // 访问完毕左右孩子添加）
    }
}

int MIN = -111;
bool flag = true;
// 判断是否是二叉排序树
bool isBiSort(BiNode* T)
{
    if (!T->lchild && flag)
        isBiSort(T->lchild);
    if (T->elem < MIN)
        flag = false;
    MIN = T->elem;
    if (!T->rchild && flag)
        isBiSort(T->rchild);
    return flag;
}
// 建立二叉排序树
void BiSortTree(BiNode*& bt, ElemType elem[], int len)
{
    bt = NULL;
```

```
    for (int i = 0; i < len; i++)
        insertBST(bt, elem[i]);
}
// 搜索（递归）
BiNode* searchBST(BiNode* bt, ElemType key)
{
    if (!bt)
        return NULL;
    else
    {
        cout << bt->elem << " ";
        if (key == bt->elem)
        {
            cout << "找到了" << endl;
            return bt;
        }
        else if (key < bt->elem)
            return searchBST(bt->lchild, key);
        else
            return searchBST(bt->rchild, key);
    }
}
// 搜索（非递归）
BiNode* searchBST_NoR(BiNode* bt, ElemType key)
{
    BiNode* p = bt;
    while (p && p->elem != key)
    {
        cout << p->elem << " ";
        // key大走右
        if (key > p->elem)
            p = p->rchild;
        // key小走左
        else
            p = p->lchild;
    }
    if (!p)
        return NULL;
    else
    {
        cout << p->elem << " ";
        cout << "找到了" << endl;
        return p;
    }
```

```
}
// 删除节点
void deleteNode(BiNode*& bt)
{
    BiNode* p = NULL;
    BiNode* parent, * pre;

    // bt为叶子节点
    if (!bt->lchild && !bt->rchild)
    {
        p = bt;
        bt = NULL;
        delete p;
    }
    // 左子树
    else if (!bt->lchild)
    {
        p = bt;
        bt = bt->rchild;
        delete p;
    }
    // 右子树
    else if (!bt->rchild)
    {
        p = bt;
        bt = bt->lchild;
        delete p;
    }
    // 左右都不空
    else
    {
        // 初始化parent与pre
        parent = bt;
        pre = parent->lchild;
        // 当删除一个左右都不空的节点时
        // 为了可以仍然构成一棵排序树
        // 需要找到左子树中最大的节点替换被删除节点
        // 下面的while循环即为找到左子树中最大节点
        while (pre->rchild)
        {
            parent = pre;
            pre = pre->rchild;
        }
        // 找到后进行替换
```

```
        bt->elem = pre->elem;
        // 接上右子树
        if (parent != bt)
            parent->rchild = pre->lchild;
        // 接上左子树
        else
            parent->lchild = pre->lchild;
    }
}
// 删除值为key的节点
bool deleteBST(BiNode*& bt, ElemType key)
{
    if (!bt)
        return false;
    else
    {
        if (key == bt->elem)
            deleteNode(bt);
        else if (key < bt->elem)
            return deleteBST(bt->lchild, key);
        else
            return deleteBST(bt->rchild, key);
        return true;
    }
}
```

// exp9-4.cpp

```
#include <iostream>
#include "bst.h"
using namespace std;

#define LEN 10

int main()
{
    BiNode* root = NULL;
    int A[LEN] = { 4,9,0,1,8,6,3,5,2,7 };
    cout << "（1）由关键字序列（4，9，0，1，8，6，3，5，2，7）创建一棵二叉排序" << endl;
    BiSortTree(root, A, 10);
    cout << "      以括号表示法输出：";
    bracketPrint(root);
    cout << endl << endl;
```

```
    cout << "（2）判断bt是否为一棵二叉排序树:";
    cout << isBiSort(root) << endl;
    cout << endl;
    cout << "（3）采用递归和非递归两种方法查找关键字为6的结点，并输出其查找路径。" << endl;
    cout << "     递归: ";
    searchBST(root, 6);
    cout << "     非递归: ";
    searchBST_NoR(root, 6);
    cout << endl;
    cout << "（4）删除bt中关键字为4和5的结点" << endl;
    cout << "     删除bt中关键字4: ";
    deleteBST(root, 4);
    bracketPrint(root);
    cout << endl;
    cout << "     删除bt中关键字5: ";
    deleteBST(root, 5);
    bracketPrint(root);
    cout << endl;

    return 0;
}
```

第 3 题：

```
// exp4-3.cpp

#include <iostream>
using namespace std;

#define LEN 8

void PrintArray(int A[], int len, int pivot = -1)
{
    bool flag = 1;
    for (int i = 0; i < len; i++)
    {
        if (A[i] == pivot && flag)
        {
            cout << "[" << A[i] << "] ";
            flag = 0;
        }
        else
```

```
            cout << A[i] << " ";
    }
    cout << endl;
}

// 划分
int Partition(int A[], int low, int high)
{
    static int time = 1;
    cout << "第 " << time << " 趟排序结果: ";
    time++;

    // 选取下标为low的元素作为划分枢纽
    int pivot = A[low];
    // 寻找枢纽的位置下标
    while (low < high)
    {
        // 在high部分找到一个不属于high的元素
        while (low < high && A[high] >= pivot)
            high--;
        // 然后将其替换到low部分去
        A[low] = A[high];
        // 同理，在low部分找到不属于的元素
        while (low < high && A[low] <= pivot)
            low++;
        // 替换到high部分去
        A[high] = A[low];
    }
    // 枢纽放入分界位置
    A[low] = pivot;
    // 打印结果
    PrintArray(A, LEN, pivot);
    // 返回
    return low;
}

// 快排函数（不断递归分区过程）
void QuickSort(int A[], int low, int high)
{
    // 递归终止条件
    if (low < high)
    {
        // 找到一个分界位置
        int pivot = Partition(A, low, high);
```

```
        // 递归分界左边
        QuickSort(A, low, pivot - 1);
        // 递归分界右边
        QuickSort(A, pivot + 1, high);
    }
}

int main()
{
    int A[LEN] = { 49,38,65,97,76,13,27,69 };
    cout << "原数组为: ";
    PrintArray(A, LEN);
    QuickSort(A, 0, LEN - 1);
    return 0;

}
```

第 4 题：

// exp10-7.cpp

```
#include <iostream>
using namespace std;

#define LEN 9

// 打印数组
void PrintArray(int A[], int len)
{
    for (int i = 1; i < len; i++)
        cout << A[i] << " ";
    cout << endl;
}

// 讲以k为根的子树调整为大根堆
void HeadAdjust(int A[], int k, int len)
{
    // 利用数组[0]的位置暂存根节点A[k]
    A[0] = A[k];
    // 沿值更加大的子节点向下筛选
    // i->当前节点k的左孩子
    for (int i = 2 * k; i <= len; i *= 2)
    {
```

```
        // A[i] < A[i + 1]看左右孩子谁更大
        // i<len保证当前节点i有右兄弟
        if (i < len && A[i] < A[i + 1])
            i++;            // i指向更大的孩子
        if (A[0] >= A[i])
            break;
        else
        {
            // 将A[i]调整到双亲节点上
            A[k] = A[i];
            // 修改k值，使其可以继续向下筛选
            k = i;
        }
    }
    // 放入最终位置
    A[k] = A[0];
}

// 建立大根堆
void BuildMaxHeap(int A[], int len)
{
    // len/2是为了找到二叉树中最底层的分支
    // 从底层往上层调整
    for (int i = len / 2; i > 0; i--)
        HeadAdjust(A, i, len);
}

// 堆排序
void HeapSort(int A[], int len)
{
    // 建立大根堆
    BuildMaxHeap(A, len);
    cout << "建立大根堆后数组：";
    PrintArray(A, len + 1);
    for (int i = len; i > 1; i--)
    {
        // 把大根放入数组尾
        swap(A[i], A[1]);
        // 整理剩下待排序元素
        // 保持大根堆样式
        HeadAdjust(A, 1, i - 1);
        cout << "第 " << len - i + 1 << " 趟排序后：";
        PrintArray(A, len + 1);
    }
```

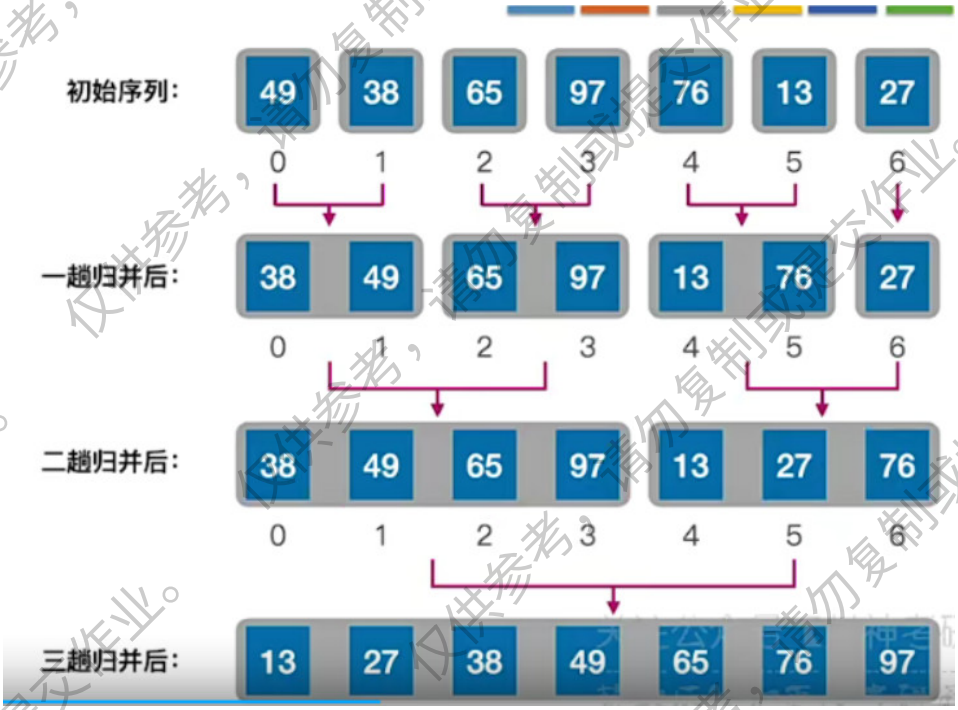
```
}

int main()
{
    int A[LEN] = { 0,53,17,78,9,45,65,87,32 };
    cout << "原始数组: ";
    PrintArray(A, LEN);
    HeapSort(A, LEN - 1);
    return 0;

}
```

第 5 题：

（1）写出采用二路归并排序方法进行排序的过程



（2）写出二路归并排序算法程序

```
// exp4-5.cpp

#include <iostream>
using namespace std;

#define LEN 7
```

```
// 打印数组
void PrintArray(int A[], int len)
{
    for (int i = 0; i < len; i++)
        cout << A[i] << " ";
    cout << endl;
}

// 创建辅助数组B
int B[LEN] = {  };

// 将以mid分界的两有序序列进行归并
void Merge(int A[], int low, int mid, int high)
{
    // for循环变量
    int i, j, k;
    // 将A中所有元素复制到B中
    for (k = low; k <= high; k++)
        B[k] = A[k];
    // 下面开始归并
    // i辅助数组B左边指针，j辅助数组B右边指针
    // k原数组指针
    for (i = low, j = mid + 1, k = i; i <= mid && j <= high; k++)
    {
        if (B[i] <= B[j])
        {
            A[k] = B[i];
            i++;
        }
        else
        {
            A[k] = B[j];
            j++;
        }
    }
    // 左边指针没有移动到尾部
    while (i <= mid)
    {
        A[k] = B[i];
        k++;
        i++;
    }
    // 右边指针没有移动到尾部
    while (j <= high)
```

```
    {
        A[k] = B[j];
        k++;
        j++;
    }
}

// 归并排序
void MergeSort(int A[], int low, int high)
{
    // 递归终止条件
    if (low<high)
    {
        // 从中间划分
        int mid = (low + high) / 2;
        // 归并左边
        MergeSort(A, low, mid);
        // 归并右边
        MergeSort(A, mid + 1, high);
        // 将划分到最细的low、mid、high进行归并
        Merge(A, low, mid, high);
    }
}

int main()
{
    int A[] = { 49,38,65,97,76,13,27 };
    cout << "原数组：";
    PrintArray(A, LEN);
    MergeSort(A, 0, LEN - 1);
    cout << "排序后：";
    PrintArray(A, LEN);
    return 0;
}
```

（3）写出二路归并排序算法的时间复杂度和稳定性

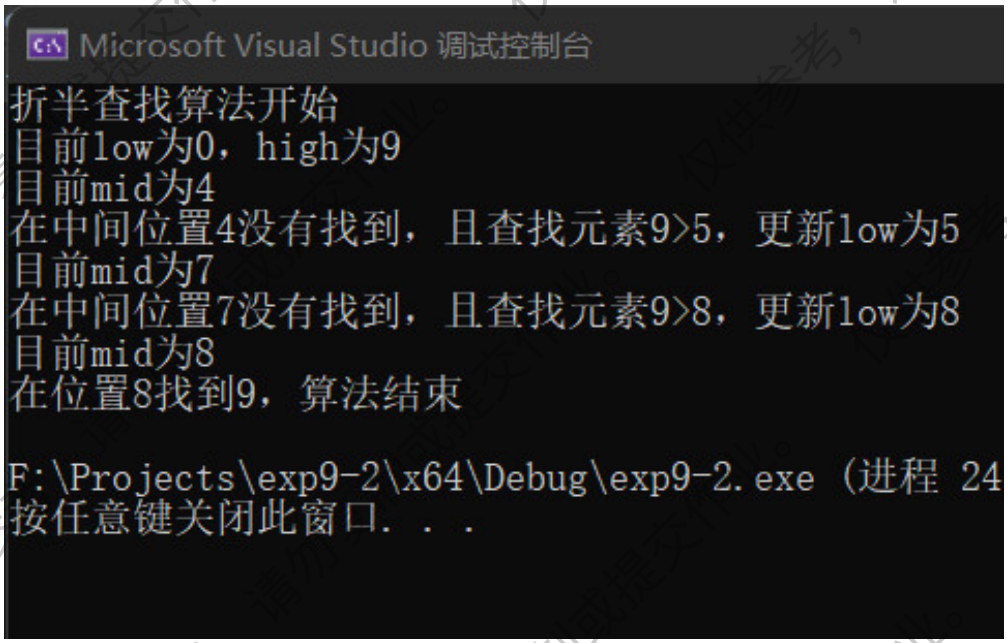
时间复杂度：O(n^2);

稳定性：不稳定

五、实验结果及分析

1、

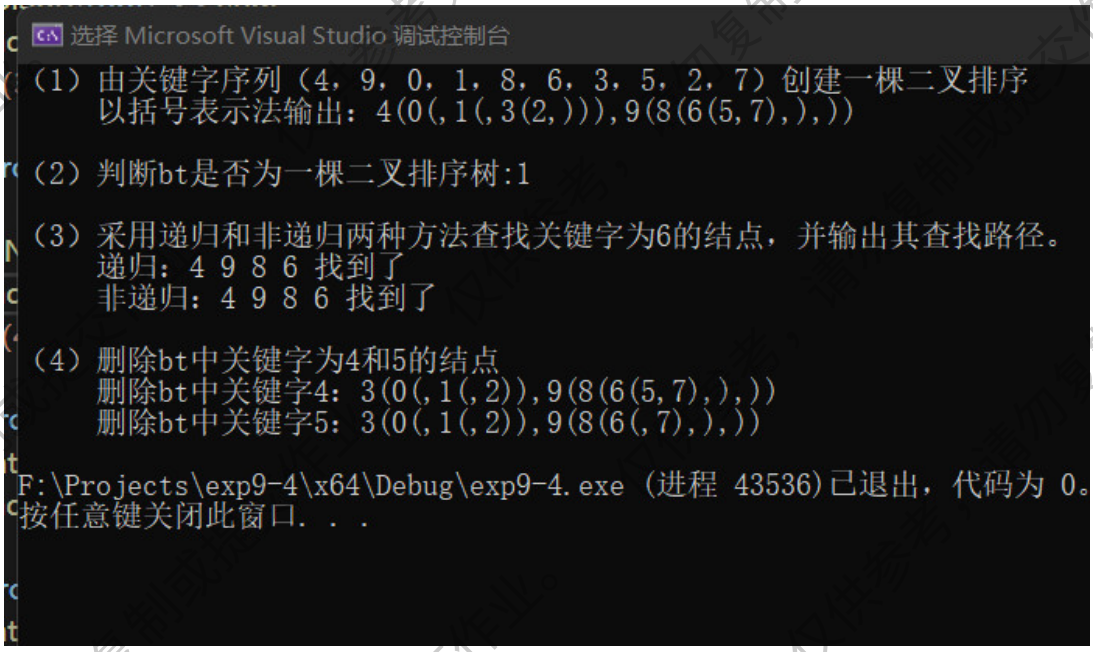
运行结果：



运行结果正确

2、

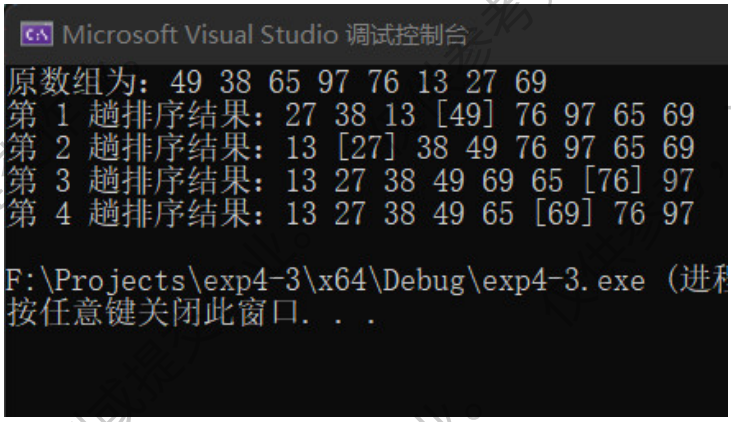
运行结果：



运行结果正确

3、

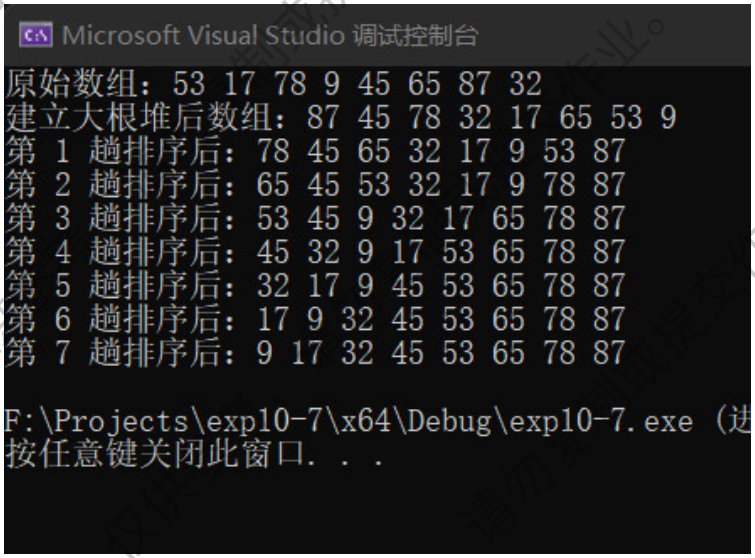
运行结果：



运行结果正确

4、

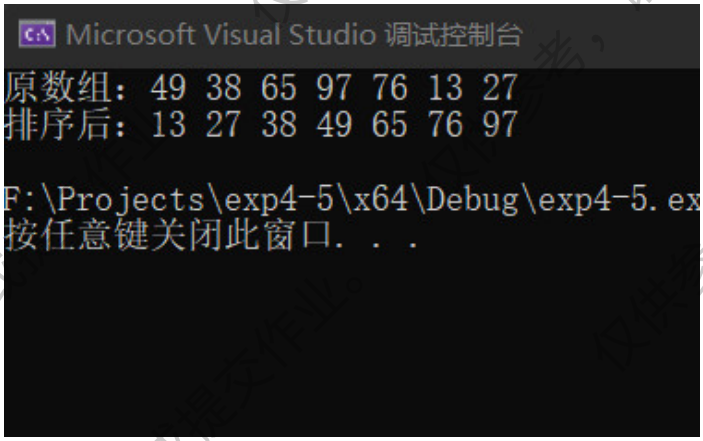
运行结果：



运行结果正确

5、

运行结果：



运行结果正确

调试日志：

bug： 在快速排序题中，划分时应该是 low-mid-1；mid+1-high 两个分区，但是在写代码时粗心大意写成了 low-mid；mid-high，导致死循环。

同时下图中的 while 应该为 if

```
// 快排函数（不断递归分区过程）
void QuickSort(int A[], int low, int high, int depth = 0)
{
    depth++;
    //cout << "第 " << depth << " 趟排序结果：";
    // 递归终止条件
    while (low < high)
    {
        // 找到一个分界位置
        int pivot = Partition(A, low, high);
        //PrintArray(A, LEN);
        cout << A[0] << A[1];
        // 递归分界左边
        QuickSort(A, low, pivot, depth);
        // 递归分界右边
        QuickSort(A, pivot, high, depth);
    }
}
```

实验体会与心得：

本次的数据结构实验主要是排序相关的实验，其中快速排序重要的思想是选取枢纽元素进行划分，如果枢纽可以均匀划分数组，则快排递归深度越浅，速度更快。