# 

## 《算法分析理论及应用》课程实验报告

**班级：软工182 姓名：邓棋 学号：2018081062**

**一、实验题目**

1. 实现堆排序的算法。
2. 实现折半查找的算法。
3. 实现二叉查找树的算法。

**二、实验内容**

**1. 堆排序。（完成实验代码、伪代码）**

堆排序（请完成堆排序的设计策略描述、时间复杂度的分析、伪代码以及代码实现）。

**2. 折半查找。（完成实验代码、伪代码）**

折半查找（请完成折半查找的设计策略描述、时间复杂度的分析、伪代码以及代码实现）。

**3.二叉查找树。（完成实验代码、伪代码）**

二叉查找树查找（请完成二叉查找树查找算法的设计策略描述、时间复杂度的分析、伪代码以及代码实现）。

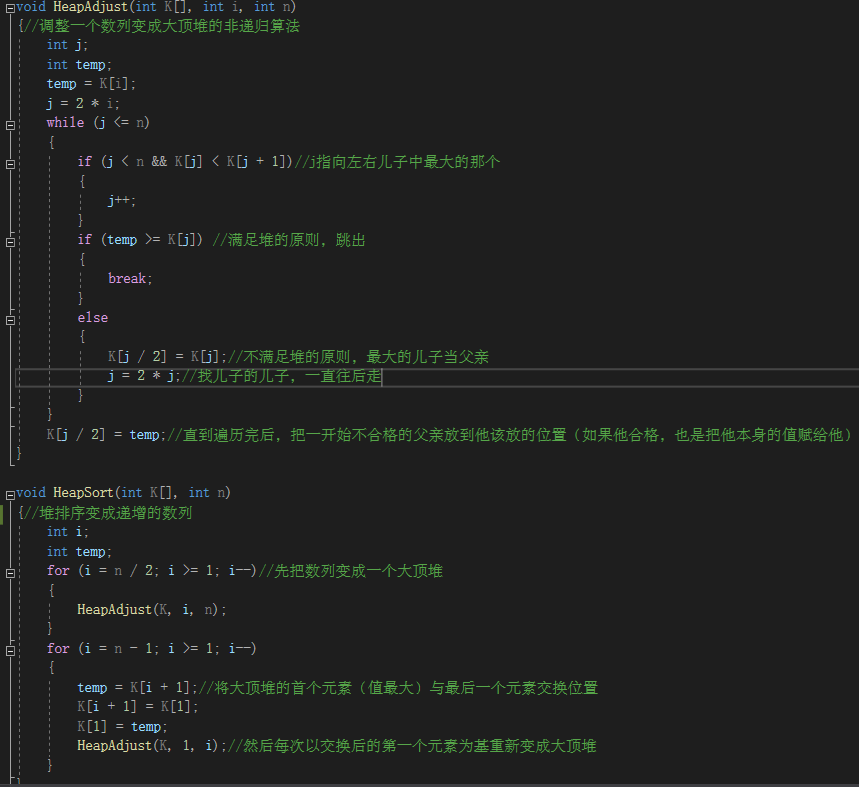
**三、实验目的**

1. 理解减治法的思想，算法策略。

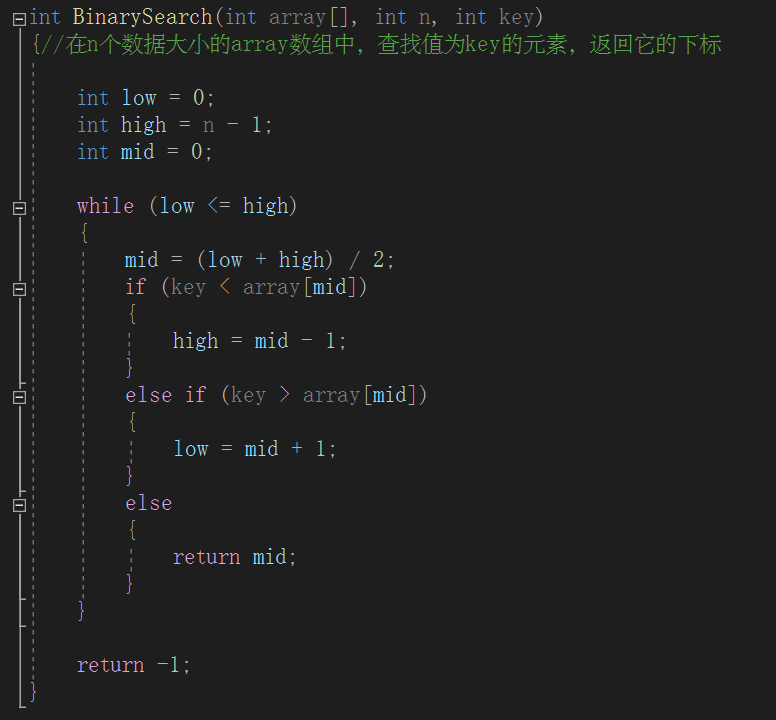
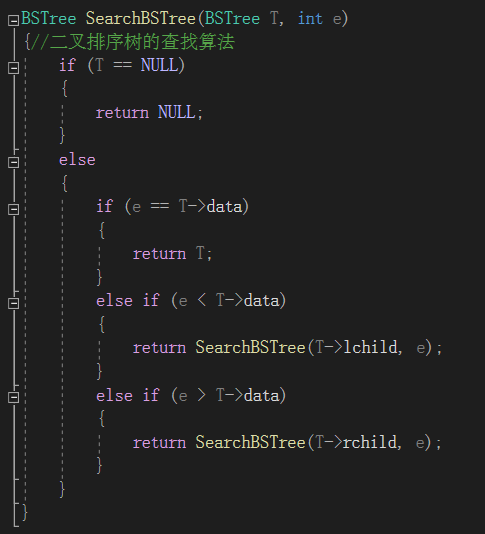
2. 掌握利用减治法解决问题的基本思想，会用高级语言对算法进行描述，并对算法复杂度（时间和空间）进行分析。

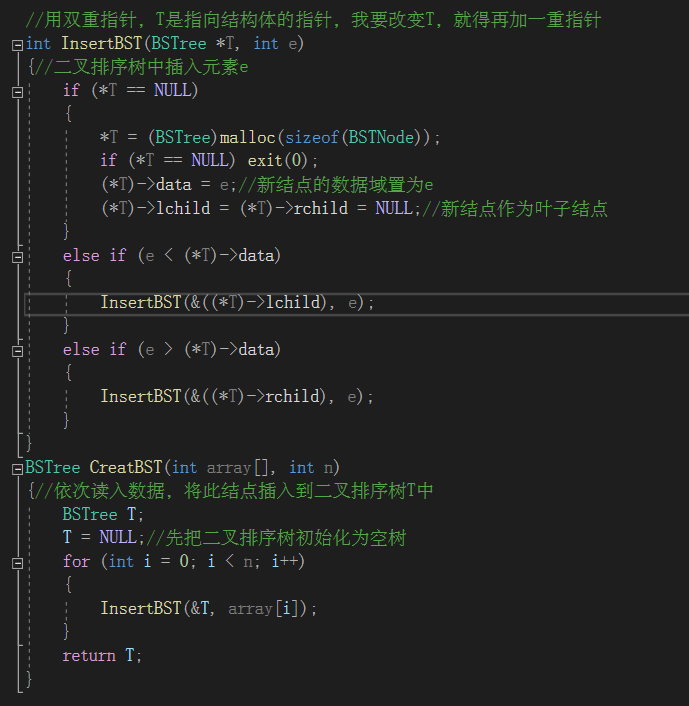
**四、实验代码**

**1、堆排序**

****

**2、折半查找**

**3、二叉查找树**



**五、实验总结**

书写以下实验总结：1、算法伪代码编写；2、算法设计策略描述；3、算法时空复杂度分析。4、遇到的问题及解决方法。

1. **算法伪代码编写**

**堆排序：**

void HeapAdjust(数组K, 调整的起始序号i, 数组元素个数n)

{//调整一个数列变成大顶堆的非递归算法

用一个变量j来存放2i

用临时变量临时储存起始序号对应的元素

while (当j依然在数组范围内)

{

j对应的编号每次指向左右儿子中最大的那个

如果满足大顶堆的原则，跳出break

else

{

K[j / 2] = K[j];//不满足堆的原则，最大的儿子当父亲

j = 2 \* j;//找儿子的儿子，一直往后走

}

}

K[j / 2] = temp;//直到遍历完后，把一开始不合格的父亲放到他该放的位置（如果他合格，也是把他本身的值赋给他）

}

void HeapSort(数组K, 数组元素个数n)

{//堆排序变成递增的数列

从n/2开始循环递减，将数列变成一个大顶堆

HeapAdjust(K, i, n);

for (i = n - 1; i >= 1; i--)

{

将大顶堆的首个元素（值最大）与最后一个元素交换位置

HeapAdjust(K, 1, i);//然后每次以交换后的第一个元素为基重新变成大顶堆

}

}

**折半查找：**

1. low=1；high=n； //设置初始查找区间

2. 测试查找区间［low，high］是否存在，若不存在，则查找失败；否则

3. 取中间点mid=(low+high)/2; 比较k与r[mid]，有以下三种情况：

3.1 若k<r[mid]，则high=mid-1；查找在左半区进行，转2；

3.2 若k>r[mid]，则low=mid+1；查找在右半区进行，转2；

3.3 若k=r[mid]，则查找成功，返回记录在表中位置mid；

**二叉查找树：**

由二叉查找树的定义，在二叉查找树root中查找给定值k的过程是：

⑴若root是空树，则查找失败；

⑵若k＝根结点的值，则查找成功；

⑶若k<根结点的值，则在root的左子树上查找；

⑷若k>根结点的值，则在root的右子树上查找；

上述过程一直持续到查找成功或者待查找的子树为空，如果待查找的子树为空，则查找失败**。**

1. **算法设计策略描述**

**堆排序：**

大顶堆需要的条件是一个序列中每个元素大于等于其二倍下标所对应的元素。堆排序就只需要从一个序列的中间位置开始，序列中元素与其二倍下标的元素关系类似于二叉树中的双亲和子结点，因此只需要将其对应的以此结点为根的二叉树转换为一个大顶堆，每转换一棵子树，序列往前递减1，重复上述过程，直到将序列中的第一个元素为根的二叉树转换为大顶堆，这样就将序列转换成了一个大顶堆。之后只需要将大顶堆的首个元素（值最大）与最后一个元素交换位置，再以交换后的第一个元素为基重新构造大顶堆，循环n-1次后序列就变成了了有序递增的。

**折半查找：**

在n个数据大小的数组中，将查找的左边界设为0，右边界为n-1，取左右边界的中值；如果目标值比左右边界中值所对应的元素大，那么左边界变为中值加1，重新计算中值；如果目标值比左右边界中值所对应的元素小，那么右边界变为中值减1，重新计算中值；如果目标值和左右边界中值所对应的元素相等，那么就返回中值（即下标）；直到左右边界重合时，循环结束。

**二叉查找树：**

二叉查找树中序遍历的结果是一个单调递增的数列，这样的结构限制了左孩子比根结点小，右孩子比根结点大，这样来看，目标值只需要每次与根结点的值进行比较，如果相等，就已经查找到了，如果比根结点所对应的值要大，就进入它的右子树，再次比较，如果比根结点所对应的值要小，就进入它的左子树，再次比较，一个递归循环的过程，即可将整棵二叉查找树遍历完。

1. **算法时空复杂度分析**

**堆排序：**

初始化建堆只需要对二叉树的非叶子节点调用HeapAdjust ()函数，由下至上，由右至左选取非叶子节点来调用HeapAdjust ()函数。那么倒数第二层的最右边的非叶子节点就是最后一个非叶子结点。

　　假设高度为k，则从倒数第二层右边的节点开始，这一层的节点都要执行子节点比较然后交换（如果顺序是对的就不用交换）；倒数第三层呢，则会选择其子节点进行比较和交换，如果没交换就可以不用再执行下去了。如果交换了，那么又要选择一支子树进行比较和交换；高层也是这样逐渐递归。

　　那么总的时间计算为：s = 2^( i - 1 ) \* ( k - i )；其中 i 表示第几层，2^( i - 1) 表示该层上有多少个元素，( k - i) 表示子树上要下调比较的次数。

　　S = 2^(k-2) \* 1 + 2^(k-3)2…..+2(k-2)+2^(0)\*(k-1) ===> 因为叶子层不用交换，所以i从 k-1 开始到 1；

　　S = 2^k -k -1；又因为k为完全二叉树的深度，而log(n) =k，把此式带入；

得到：S = n - log(n) -1，所以时间复杂度为：O(n)

在取出堆顶点放到对应位置并把原堆的最后一个节点填充到堆顶点之后，需要对堆进行重建，只需要对堆的顶点调用HeapAdjust ()函数。

每次重建意味着有一个节点出堆，所以需要将堆的容量减一。HeapAdjust ()函数的时间复杂度k=log(n)，k为堆的层数。所以在每次重建时，随着堆的容量的减小，层数会下降，函数时间复杂度会变化。重建堆一共需要n-1次循环，每次循环的比较次数为log2(i)，则相加为：log22+ log23+…+ log2 (n-1)+ log2 (n)≈log2 (n!)。可以证明log2 (n!)和n log2 (n)是同阶函数：

∵(n/2)n/2≤n!≤nn,

∴n/4log2(n)=n/2log2(n1/2)≤n/2log2 (n/2)≤log2(n!)≤nlog2(n)

所以时间复杂度为O(nlog2n)

初始化建堆的时间复杂度为O(n)，排序重建堆的时间复杂度为nlog2n，所以总的时间复杂度为O(n+nlog2n)=O(nlog2n)。另外堆排序的比较次数和序列的初始状态有关，但只是在序列初始状态为堆的情况下比较次数显著减少，在序列有序或逆序的情况下比较次数不会发生明显变化。

因为堆排序没有使用与n有关的变量，空间复杂度为常数：O(1)。

**折半查找：**

折半查找的基本思想就是将n个元素分成大致相等的两部分，然后中值与目标值做比较，然后不断再分再分，时间复杂度取决于while循环的次数，n个元素的渐进：n, n/2, n/4, ……, n/2k  k即为循环的次数,故时间复杂度为O(log2n)

因为折半查找没有使用与n有关的变量，空间复杂度为常数：O(1)。

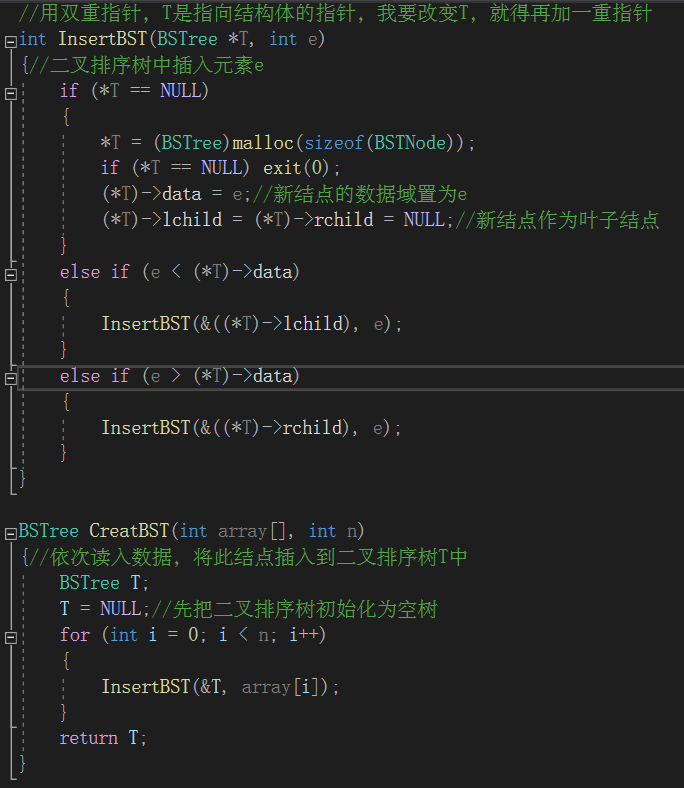
**二叉查找树：**

如果二叉排序树是平衡的，则n个节点的二叉排序树的高度为log2n+1,其时间复杂度为O(log2n)，近似于折半查找。如果二叉排序树完全不平衡，则其深度可达到n，时间复杂度为O(n)，退化为顺序查找。一般的，二叉排序树的查找性能在O(log2n)到O(n)之间。因此，为了获得较好的查找性能，就要构造一棵平衡的二叉排序树。

因为需要建立一棵二叉排序树，需要有n个结点，所以空间复杂度为O(n)。

1. **遇到的问题及解决方法**

构建二叉排序树的时候，总是提示无法读取内存，存在野指针，因为我一直用的C语言写，没有C++的引用参数，就显得麻烦一点，只需要把形参改为双重指针即可。因为形参T是指向结构体的指针，我要改变T，就得再加一重指针。



**六、算法策略的英文描述（字数>200）**

**Heap sorting:**

The condition of the large top heap is that each element in a sequence is greater than or equal to the element corresponding to its double subscript. Heap sorting only needs to start from the middle position of a sequence. The relationship between the elements in the sequence and their double-subscript elements is similar to that of the parent and children nodes in the binary tree. Therefore, it is only necessary to convert the corresponding binary tree rooted at this node to a large top heap. The sequence is decremented by one every time a subtree is transformed, and the above process is repeated until the binary tree whose root is the first element in the sequence is converted to a large top heap, so that the sequence is converted into a large top heap. After that, you only need to swap the first element (the largest value) of the large top heap with the last element, and then rebuild the large top heap based on the first element after the exchange. After n-1 cycles the sequence becomes an ordered increment.

**Binary Search:**

In an array of n data sizes, the left boundary of the search is set to 0, the right boundary is set to n-1, and the median value of the left and right boundaries is taken. If the target value is larger than the element corresponding to the median value of the left and right boundaries, then the left boundary becomes the median plus 1 and the median is recalculated; if the target value is smaller than the element corresponding to the median of the left and right boundaries, then the right boundary becomes the median minus 1, recalculating the median; if the target and the median of the left and right boundaries correspond The elements of are equal, then the median value (that is the subscript) is returned; until the left and right boundaries coincide, the loop ends.

**Binary Search Tree:**

The result of the order traversal in the binary search tree is a monotonically increasing sequence. This structure restricts the left child is smaller than the root node and the right child is larger than the root node. In this way, the target value only needs to be connected with the root node each time to compare the values ​​of the points. If they are equal, you have already found them. If the target value is larger than the value corresponding to the root node, just enter its right subtree and compare again. If the value is smaller than the value corresponding to the root node, then enter its left subtree and compare again. A recursive loop process which can traverse the entire binary search tree.