第七章、排序

1、掌握直接插入排序、希尔排序、冒泡排序、简单选择排序的思想及实现方法;

2、掌握快速排序、堆排序、归并排序的思想和及实现方法。

3、掌握算法复杂度及其分析方法;熟悉算法特点及其适用场景。

定义

稳定性：关键字相同的元素排序前后相对位置不变

速查速查

大类	算法名称	空间复杂度	时间复杂度	稳定性	适用于
插入排序	直接插入	$O(1)$	最好：$O(n)$ → 原本有序 最坏：$O(n^2)$ → 原本逆序 平均：$O(n^2)$	√	顺序表、链表
	折半插入	$O(1)$	最好：$O(n)$ → 原本有序 最坏：$O(n^2)$ → 原本逆序 平均：$O(n^2)$	√	顺序表
	希尔	$O(1)$	未知 优于直接插入排序	x	顺序表
交换排序	冒泡	$O(1)$	最好：$O(n)$ → 原本有序 最坏：$O(n^2)$ → 原本逆序 平均：$O(n^2)$	√	顺序表、链表
	快排	最坏：$O(n^2)$ 最好：$O(log_2 n)$	最坏：$O(n)$ → 有序 or 逆序 最好：$O(nlog_2 n)$ → 划分很平均	x	
选择排序	简单选择	$O(1)$	$O(n^2)$	x	顺序表、链表
	堆排序	$O(1)$	建堆：$O(n)$ 排序：$O(nlog_2 n)$ → 总：$O(nlog_2 n)$	x	
归并排序		$O(n)$	$O(nlog_2 n)$	√	

插入排序：

```
//直接插入排序
void InsertSort(int A[],int n){
    int i,j,temp;
    for(i=1;i<n;i++)              //将各元素插入已排好序的序列中
        if(A[i]<A[i-1]){          //若A[i]关键字小于前驱
            temp=A[i];            //用temp暂存A[i]
            for(j=i-1;j>=0 && A[j]>temp;--j)   //检查所有前面已排好序的元素
                A[j+1]=A[j];      //所有大于temp的元素都向后挪位
            A[j+1]=temp;          //复制到插入位置
        }
}
```

算法评价

- 空间：O(1)
- 时间
  - 最好：O(n) → 原本有序，$n-1$ 趟处理，每一趟对比关键字一次，不移动元素
  - 最坏：$O(n^2)$ → 原本逆序，第 $n-1$ 趟，对比关键字 n 次，移动元素 $n+1$ 次
  - 平均：$O(n^2)$

*时间复杂度来源于对比关键字和移动元素，若有 n 个元素，需要 n-1 个 for 循环

链表的插入排序

时间复杂度来自于关键字的对比，依然是$O(n^2)$

折半插入排序算法

```
void InsertSort(int A[],int n){
    int i,j,low,high,mid;
    for(i=2;i<=n;i++){            //依次将A[2]~A[n]插入前面的已排序序列
        A[0]=A[i];                //将A[i]暂存到A[0]
        low=1;high=i-1;           //设置折半查找的范围
        while(low<=high){         //折半查找(默认递增有序)
            mid=(low+high)/2;     //取中间点
            if(A[mid]>A[0]) high=mid-1; //查找左半子表
            else low=mid+1;       //查找右半子表
        }
        for(j=i-1;j>=high+1;--j)
            A[j+1]=A[j];          //统一后移元素，空出插入位置
        A[high+1]=A[0];           //插入操作
    }
}
```

希尔排序

```
void ShellSort(int A[],int n){
    int d, i, j;
    //A[0]只是暂存单元，不是哨兵，当j<=0时，插入位置已到
    for(d= n/2; d>=1; d=d/2)    //步长变化
        for(i=d+1; i<=n; ++i)
            if(A[i]<A[i-d]){    //需将A[i]插入有序增量子表
                A[0]=A[i];              //暂存在A[0]
                for(j= i-d; j>0 && A[0]<A[j]; j-=d)
                    A[j+d]=A[j];        //记录后移，查找插入的位置
                A[j+d]=A[0];            //插入
            }//if
}
```

算法评价 $\begin{cases}\text{空间：}O(1)\\ \text{时间}\begin{cases}\text{一般：未知}\rightarrow n\text{ 如果不大，在某个范围内，}O(n^{1.3})\\ \text{最坏：}O(n^2)\rightarrow d=1\end{cases}\end{cases}$

时间复杂度和增量序列有关

最坏 d=1，$O（n^2）$

仅针对顺序表

冒泡排序

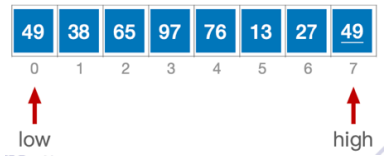
```
void swap(int &a, int &b)
    {
        int temp = a;
        a = b;
        b = temp;
    }
```

```
void BubbleSort(int A[],int n){
    for(int i=0;i<n-1;i++){
        bool flag=false;            //表示本趟冒泡是否发生交换的标志
        for(int j=n-1;j>i;j--)      //一趟冒泡过程
            if(A[j-1]>A[j]){        //若为逆序
                swap(A[j-1],A[j]);  //交换
                flag=true;
            }
        if(flag==false)
            return;          //本趟遍历后没有发生交换，说明表已经有序
    }
}
```

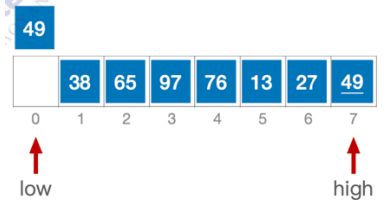
算法评价

$\begin{cases}\text{空间：}O（1）\\ \text{时间}\begin{cases}\text{最好：}O(n)\rightarrow\text{有序，比较次数 }n-1\text{，交换次数 }0\\ \text{最坏：}O(n^2)\rightarrow\text{逆序，比较次数}=(n-1)+(n-2)+\ldots+1=\dfrac{n(n-1)}{2}=\text{交换次数}\\ \text{平均：}O(n^2)\end{cases}\end{cases}$

快速排序

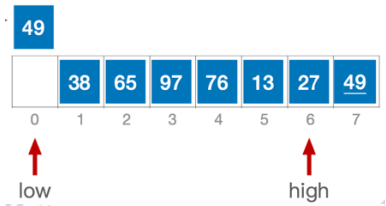


初始顺序 belike

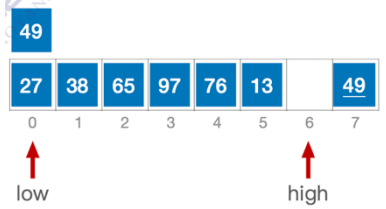


通常取首元素为枢轴（pivot）

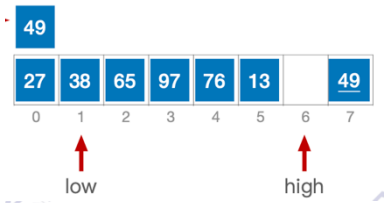
high 指向的 49 和枢轴一样，不管，向左移



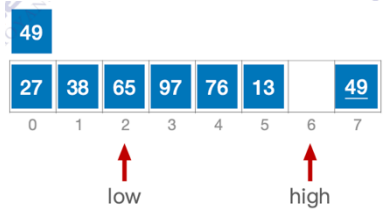
high 移到 27，27 比 49 小，所以把 27 移到 low 指针那里去



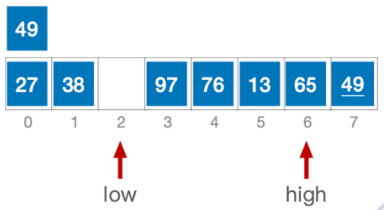
现在 high 指针空了，low 指针向右探测



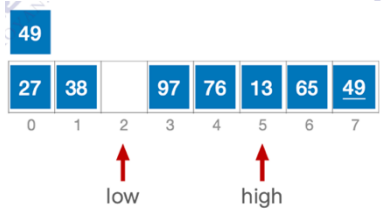
38 比 49 小，不管，继续向右



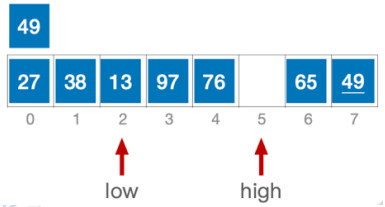
65 比 49 大，弄到 high 指针那里去



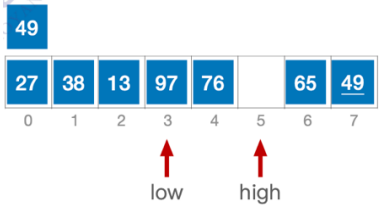
low 指针空咯，high 指针向左探测



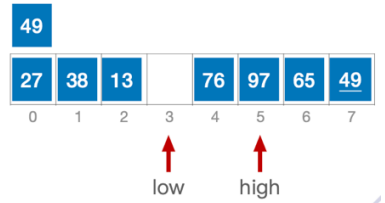
13 比 49 小，往 low 指针放



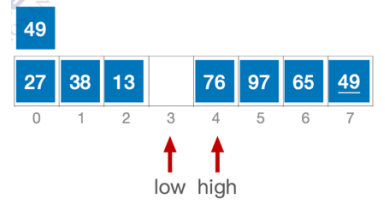
high 指针空出来，low 指针继续向右探测



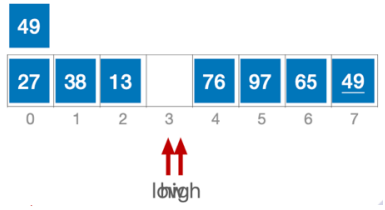
97 比 49 大，放到 high 指针去



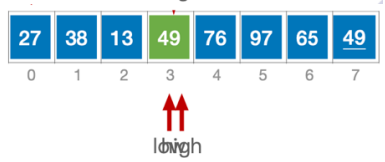
空出 low，high 向左探测



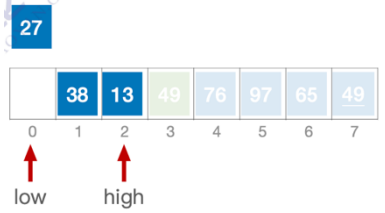
76 比 49 大，不管，继续向左



low 和 high 重合，第一次排序结束。把枢轴 49 放进来

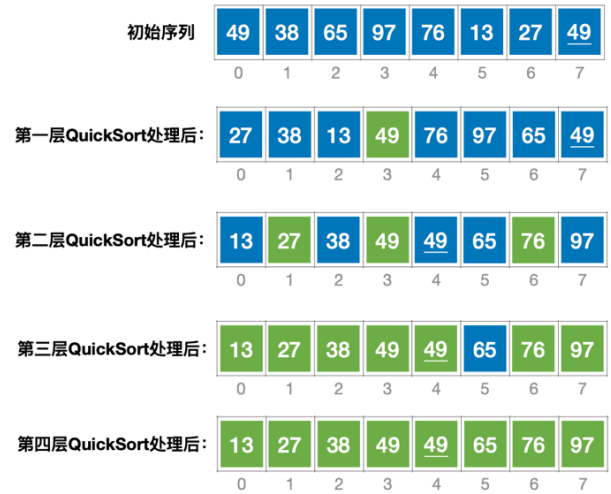


此时所有元素以 49 为基准分为两部分，49 左比 49 小，右边是比 49 大。



取 27 为新的枢轴元素，以此类推。排好了以后再去排后半部分

每次排序后的序列情况：




```c
//用第一个元素将待排序序列划分成左右两个部分
int Partition(int A[],int low,int high){
    int pivot=A[low];          //第一个元素作为枢轴
    while(low<high){           //用low、high搜索枢轴的最终位置
        while(low<high&&A[high]>=pivot) --high;
        A[low]=A[high];        //比枢轴小的元素移动到左端
        while(low<high&&A[low]<=pivot) ++low;
        A[high]=A[low];        //比枢轴大的元素移动到右端
    }
    A[low]=pivot;              //枢轴元素存放到最终位置
    return  low;               //返回存放枢轴的最终位置
}

//快速排序
void QuickSort(int A[],int low,int high){
    if(low<high){   //递归跳出的条件
        int pivotpos=Partition(A,low,high); //划分
        QuickSort(A,low,pivotpos-1);   //划分左子表
        QuickSort(A,pivotpos+1,high);  //划分右子表
    }
}
```

算法评价
- 空间
  - 最坏：$O(n^2)$
  - 最好：$O(log_2 n)$
- 时间
  - 最坏：$O(n)$ → 有序 or 逆序
  - 最好：$O(nlog_2 n)$ → 划分很平均

堆排序

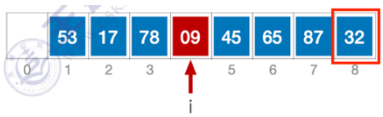
大根堆：根>左右孩子

小根堆：根<左右孩子

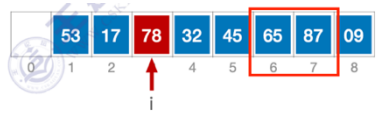
*step*1.建立大根堆

检查所有非终端结点（顺序存储的完全二叉树中：$i \le \lfloor n/2 \rfloor$）是否满足大根堆堆要求，比如这个，检查 i=0～4。

$$i\begin{cases}\text{左孩子：}2i\\ \text{右孩子：}2i+1\\ \text{父结点：}i/2\end{cases}$$



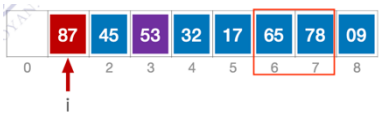
从 4 开始依次向前检查。
将 i=4 与 i=8 和 i=9（此处没有）比较，不满足则与结点较大的孩子进行交换。
这里需要交换



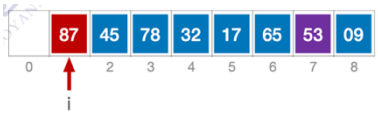
87 大，换一下

45 大，换一下

87 大换一下

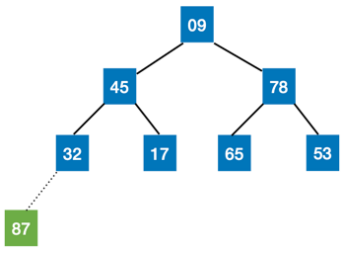


检查当前结点是否满足根≥左、右
不满足，将当前结点与更大的一个孩子互换

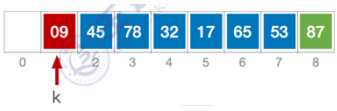


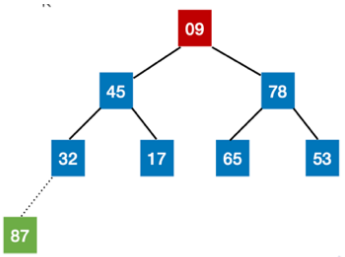
再检查一次，此时已经是叶子结点了，调整完毕
这叫小元素下坠

*step*2.基于大根堆的排序

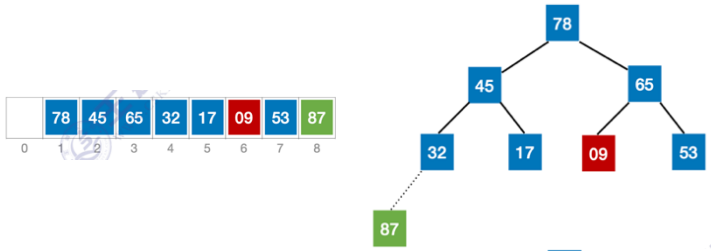


每一趟将堆顶元素加入有序子序列（与待排序序列中的最后一个元素交换

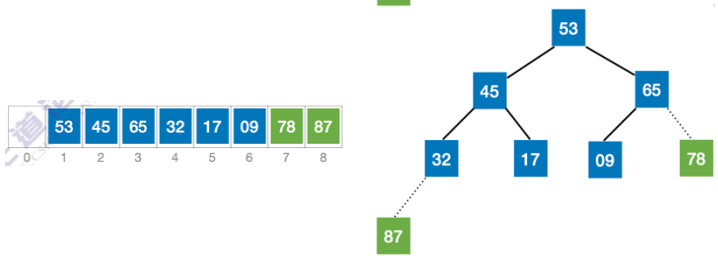




调整大根堆
调整前 belike



调整后



将新的堆顶元素加入有序序列

调整大根堆・・・

```
//建立大根堆
void BuildMaxHeap(int A[],int len){
    for(int i=len/2;i>0;i--)     //从后往前调整所有非终端结点
        HeadAdjust(A,i,len);
}

//将以 k 为根的子树调整为大根堆
void HeadAdjust(int A[],int k,int len){
    A[0]=A[k];                   //A[0]暂存子树的根结点
    for(int i=2*k;i<=len;i*=2){ //沿key较大的子结点向下筛选
        if(i<len&&A[i]<A[i+1])
            i++;                 //取key较大的子结点的下标
        if(A[0]>=A[i])  break;  //筛选结束
        else{
            A[k]=A[i];           //将A[i]调整到双亲结点上
            k=i;                 //修改k值，以便继续向下筛选
        }
    }
    A[k]=A[0];                   //被筛选结点的值放入最终位置
}

//堆排序的完整逻辑
void HeapSort(int A[],int len){
    BuildMaxHeap(A,len);         //初始建堆
    for(int i=len;i>1;i--){      //n-1趟的交换和建堆过程
        swap(A[i],A[1]);         //堆顶元素和堆底元素交换
        HeadAdjust(A,1,i-1);     //把剩余的待排序元素整理成堆
    }
}
```

插入：

删除：被删除的元素用堆底元素互换，让堆底元素下坠直到无法下坠为止

归并排序

```
int *B=(int *)malloc(n*sizeof(int)); //辅助数组B
```

```
//A[low…mid]和A[mid+1…high]各自有序，将两个部分归并
void Merge(int A[],int low,int mid,int high){
    int i,j,k;
    for(k=low;k<=high;k++)
        B[k]=A[k];           //将A中所有元素复制到B中
    for(i=low,j=mid+1,k=i;i<=mid&&j<=high;k++){
        if(B[i]<=B[j])
            A[k]=B[i++];    //将较小值复制到A中
        else
            A[k]=B[j++];
    }//for
    while(i<=mid)   A[k++]=B[i++];
    while(j<=high)  A[k++]=B[j++];
}
```