@[toc]

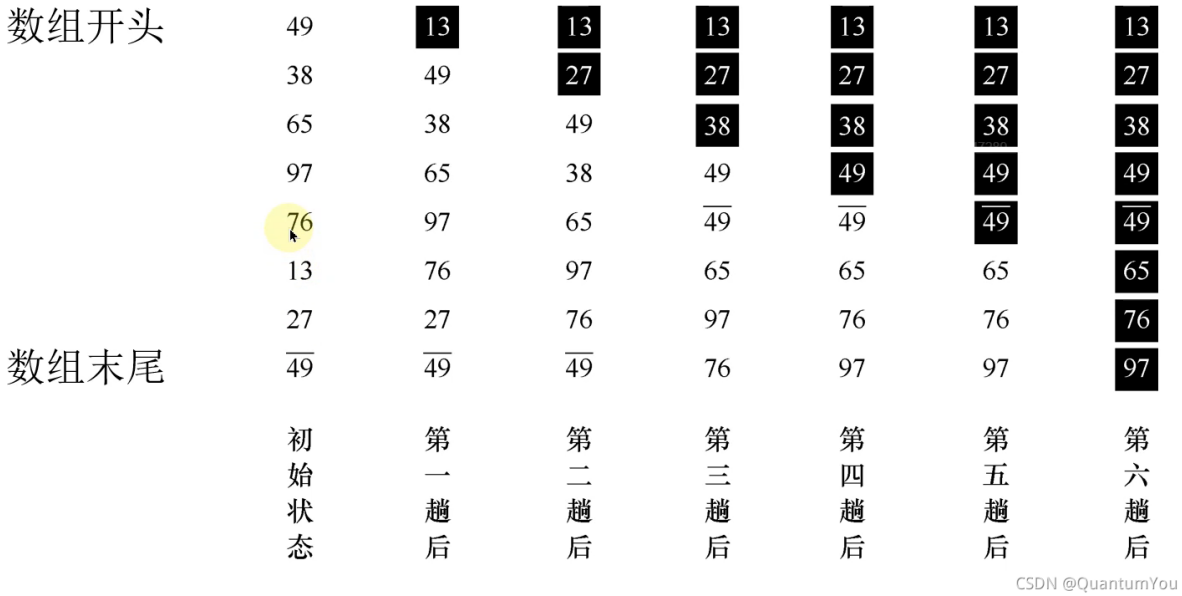
第十一次直播 排序

- 交换排序有：冒泡排序，选择排序

冒泡排序

冒泡排序的基本思想是：从后往前（或从前往后）两两比较相邻元素的值，（若`A[j-1]>A[j]`），则交换它们，直到序列比较完。我们称它为第一趟冒泡，结果是将最小的元素交换到待排序列的第一个位置。关键字最小的元素如气泡一般逐渐往上"漂浮"直至"水面".下一趟冒泡时前一趟确定的最小元素不再参与比较，每趟冒泡的结果是把序列中的最小元素放到了序列的最终位置。∴...这样最多做n-1趟冒泡就能把所有元素排好序

冒泡过程



CSDN @QuantumYou

冒泡排序分析

小知识点

- 内存copy接口，当你copy整型数组，或者浮点型时，要用`memcpy`
- `strcpy` 是遇到0 停止拷贝
- 添加`flag` 的原因在于减少排序次数，发现没有交换代码

总体排序代码

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
typedef int ElemType;
typedef struct{
    ElemType *elem;//存储元素的起始地址
    int TableLen;//元素个数
}SSTable;

void ST_Init(SSTable &ST,int len)
{
```

```
    ST.TableLen=len;
    ST.elem=(ElemType *)malloc(sizeof(ElemType)*ST.TableLen);
    int i;
    srand(time(NULL));//随机数生成，每一次执行代码就会得到随机的10个元素
    for(i=0;i<ST.TableLen;i++)
    {
        ST.elem[i]=rand()%100;//生成的是0-99之间
    }
}
void ST_print(SSTable ST)
{
    for(int i=0;i<ST.TableLen;i++)
    {
        printf("%3d",ST.elem[i]);
    }
    printf("\n");
}
void swap(ElemType &a,ElemType &b)
{
    ElemType tmp;
    tmp=a;
    a=b;
    b=tmp;
}
// 64 94 95 79 69 84 18 22 12 78
// 12 64 94 95 79 69 84 18 22 78
void BubbleSort(ElemType A[],int n)
{
    int i,j;
    bool flag;
    for(i=0;i<n-1;i++)//i最多访问到8
    {
        flag=false;
        for(j=n-1;j>i;j--)//把最小值就放在最前面
        {
            if(A[j-1]>A[j])
            {
                swap(A[j-1],A[j]);
                flag=true;
            }
        }
        if(false==flag)
            return;
    }
}

void BubbleSort1(ElemType A[], int n)
{
    int i, j,flag;
    for (i=0;i<n-1;i++)//i是控制有多少个有序了
    {
        flag = 0;
        for (j = n-1; j>i;j--)//内层控制比较，交换
        {
            if (A[j - 1] > A[j])
            {
                swap(A[j - 1], A[j]);
                flag = 1;
```

```
            }
        }
        if (0 == flag)
        {
            break;
        }
    }
}
// 64 94 95 79 69 84 18 22 12 78
//比64小的放在左边，比64大的放在右边
//int Partition(ElemType A[],int low,int high)
//{
//  ElemType pivot=A[low];
//  while(low<high)
//  {
//      while(low<high&&A[high]>=pivot)
//          --high;
//      A[low]=A[high];
//      while(low<high&&A[low]<=pivot)
//          ++low;
//      A[high]=A[low];
//  }
//  A[low]=pivot;
//  return low;
//}



int Partition(int* arr, int left, int right)
{
    int k, i;
    for (k = i = left;i<right;i++)
    {
        if (arr[i] < arr[right])
        {
            swap(arr[i], arr[k]);
            k++;
        }
    }
    swap(arr[k], arr[right]);
    return k;
}
//递归实现
void QuickSort(ElemType A[],int low,int high)
{
    if(low<high)
    {
        int pivotpos=Partition(A,low,high);//分割点左边的元素都比分割点要小，右边的比分割点大
        QuickSort(A,low,pivotpos-1);
        QuickSort(A,pivotpos+1,high);
    }
}

//冒泡排序与快速排序
int main()
{
    SSTable ST;
```

```
128     ElemType A[10]={ 64, 94, 95, 79, 69, 84, 18, 22, 12 ,78};
129     ST_Init(ST,10);//初始化
130     //memcpy(ST.elem,A,sizeof(A));//内存copy接口，当你copy整型数组，或者浮点型
    时，要用memcpy
131     ST_print(ST);
132     //BubbleSort1(ST.elem,10);//冒泡排序
133     QuickSort(ST.elem,0,9);
134     ST_print(ST);
135     system("pause");
136 }
```

快速排序

- 分治思想
- 快速排序的核心是分治思想：假设我们的目标依然是按从小到大的顺序排列，我们找到数组中的一个分割值，把比分割值小的数都放在数组的左边，把比分割值大的数都放在数组的右边，这样分割值的位置就被确定。数组一分为二，我们只需排前一半数组和后一半数组，复杂度直接减半采用这种思想，不断地进行递归，最终分割得只剩一个元素时，整个序列自然就是有序的

快速排序分析：

1	最初	k i 3 87 2 93 78 56 61 38 12 **40**
2	第 1 次比较后	k i 3 87 2 93 78 56 61 38 12 **40**
3	第 2 次比较后	k i 3 87 **2** 93 78 56 61 38 12 **40**
4	第 3 次比较发生交换后效果	k i 3 2 87 93 78 56 61 38 12 **40**
5	第 3 次将要发生交换	k i 3 2 87 93 78 56 61 **38** 12 **40**
6	第 3 次发生交换后	k i 3 2 38 93 78 56 61 87 12 **40**
7	第 4 次发生交换后	k i 3 2 38 12 78 56 61 87 93 **40**
8	最后一次交换	k i CSDN @QuantumYou

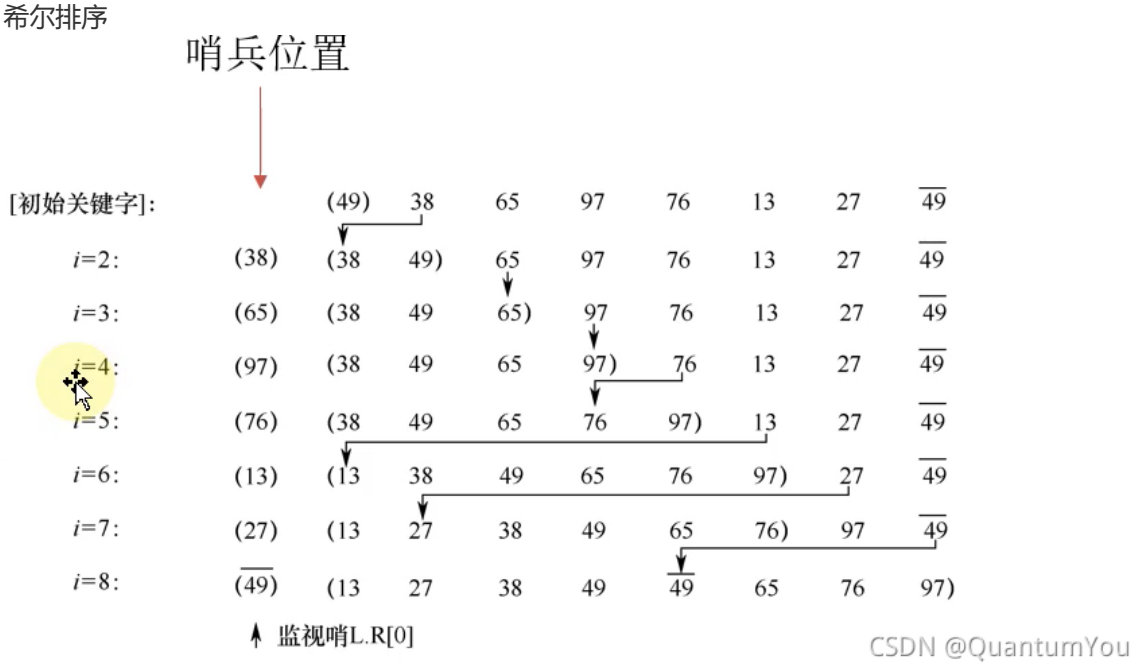
`i` 用来遍历数组的下标

`k` 用来比40小的元素将要存的下标

```
int Partition1(int* arr, int left, int right)
{
    int k, i;//k记录要放入比分割值小的数据的位置
    for (i = left, k = left; i < right; i++)
    {
        if (arr[i] < arr[right])
        {
            swap(arr[k], arr[i]);
            k++;
        }
    }
    swap(arr[k], arr[right]);
    return k;
}
```

直接插入排序

插入排序分为
1、直接插入排序
2、折半插入排序
3、希尔排序



CSDN @QuantumYou

插入排序的总体代码如下

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
typedef int ElemType;
typedef struct{
    ElemType *elem;//整型指针
    int TableLen;
}SSTable;

void ST_Init(SSTable &ST,int len)
{
    ST.TableLen=len+1;//实际申请11个元素的空间
    ST.elem=(ElemType *)malloc(sizeof(ElemType)*ST.TableLen);
    int i;
    srand(time(NULL));
    for(i=0;i<ST.TableLen;i++)
    {
        ST.elem[i]=rand()%100;//随机了11个数，但是第一个元素是没有用到的
    }
}
void ST_print(SSTable ST)
{
    for(int i=0;i<ST.TableLen;i++)
    {
        printf("%3d",ST.elem[i]);
    }
    printf("\n");
}
//插入排序，从小到大排序，升序
void InsertSort(ElemType A[],int n)
{
```

```
    int i,j;
    //24 66 94  2 15 74 28 51 22 18  2
    for(i=2;i<=n;i++)//第零个元素是哨兵，从第二个元素开始拿，往前面插入
    {
        if(A[i]<A[i-1])
        {
            A[0]=A[i];//放到暂存位置，A[0]即是暂存，也是哨兵
            for(j=i-1;A[0]<A[j];--j)//移动元素，内层循环控制有序序列中的每一个元素和要插入的元素比较
                A[j+1]=A[j];
            A[j+1]=A[0];//把暂存元素插入到对应位置
        }
    }
}
//折半查找 插入排序
void MidInsertSort(ElemType A[],int n)
{
    int i,j,low,high,mid;
    for(i=2;i<=n;i++)
    {
        A[0]=A[i];
        low=1;high=i-1;
        while(low<=high)//先通过二分查找找到待插入位置
        {
            mid=(low+high)/2;
            if(A[mid]>A[0])
                high=mid-1;
            else
                low=mid+1;
        }
        for(j=i-1;j>=high+1;--j)
            A[j+1]=A[j];
        A[high+1]=A[0];
    }
}
//希尔排序
//多轮插入排序，考的概率很低，因为编写起来复杂，同时效率并不如快排，堆排
void ShellSort(ElemType A[],int n)
{
    int dk,i,j;
    // 73 29 74 51 29 90 37 48 72 54 83
    for(dk=n/2;dk>=1;dk=dk/2)//步长变化
    {
        for(i=dk+1;i<=n;++i)//以dk为步长进行插入排序
        {
            if(A[i]<A[i-dk])
            {
                A[0]=A[i];
                for(j=i-dk;j>0&&A[0]<A[j];j=j-dk)
                    A[j+dk]=A[j];
                A[j+dk]=A[0];
            }
        }
    }
}

int main()
{
```

```
    SSTable ST;
    ST_Init(ST,10);//实际申请了11个元素空间
    ST_print(ST);
    InsertSort(ST.elem,10);
    //MidInsertSort(ST.elem,10);
    //ShellSort(ST.elem,10);
    ST_print(ST);
    system("pause");
}
```

第十二次直播 排序

快速排序

- 挖坑法, 第一次学习的方法

```
// 挖坑法，王道书上使用的方法，最左边作为分割值
int Partition(ElemType A[],int low,int high)
{
    ElemType pivot=A[low];//把最左边的值暂存起来
    while(low<high)
    {
        while(low<high&&A[high]>=pivot)//让high从最右边找，找到比分割值小，循环结束
            --high;
        A[low]=A[high];
        while(low<high&&A[low]<=pivot)//让low从最左边开始找，找到比分割值大，就结束
            ++low;
        A[high]=A[low];
    }
    A[low]=pivot;
    return low;
}
```

折半排序

- 不难看出折半插入排序仅减少了比较元素的次数，约为`O(mlog2n)`,该比较次数与待排序表的初始状态无关，仅取决于表中的元素个数n:而元素的移动次数并未改变，它依赖于待排序表的初始状态。因此，折半插入排序的时间复杂度仍为`O(n2)`,但对于数据量不很大的排序表，折半插入排序往往能表现出很好的性能

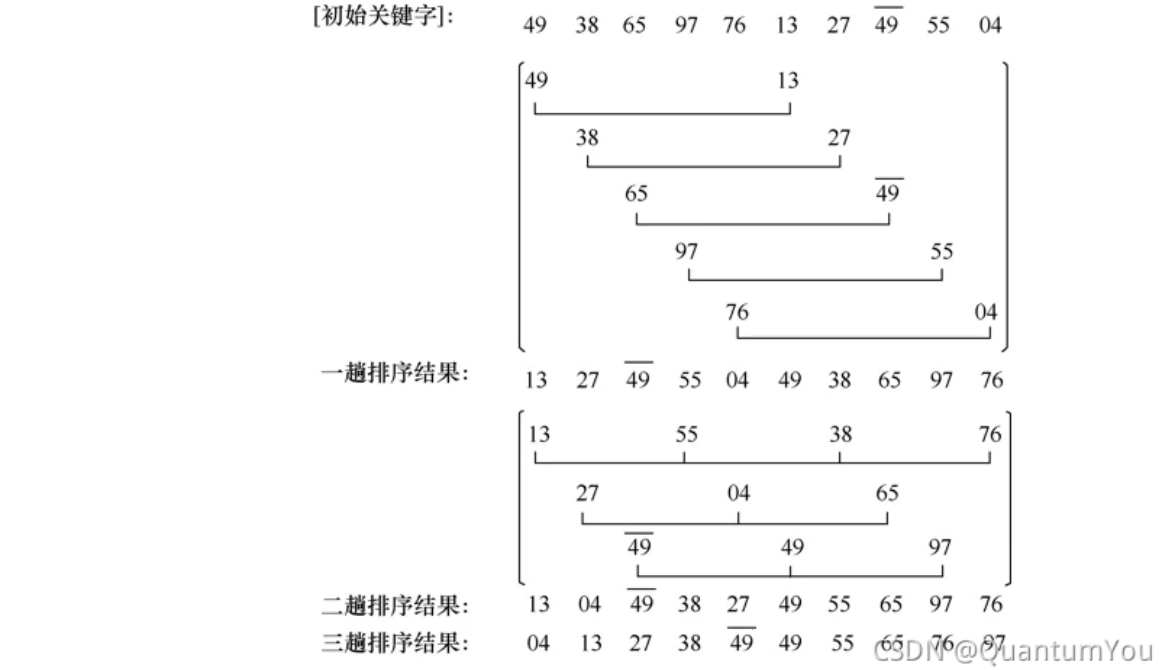
折半查找-->二分查找

```
//折半查找 插入排序，考的很少
void MidInsertSort(ElemType A[],int n)
{
    int i,j,low,high,mid;
    for(i=2;i<=n;i++)
    {
        A[0]=A[i];
        low=1;high=i-1;//low有序序列的开始，high有序序列的最后
        while(low<=high)//先通过二分查找找到待插入位置
        {
            mid=(low+high)/2;
            if(A[mid]>A[0])
```

```
                high=mid-1;
            else
                low=mid+1;
        }
        for(j=i-1;j>=high+1;--j)
            A[j+1]=A[j];
        A[high+1]=A[0];
    }
}
```

希尔排序

步长不同，通过多趟来实现，相当于减少了元素移动



CSDN@QuantumYou

- **注意**上述的第二个的步长为2，图的步长为3，绘画错误。

选择排序

简单选择排序

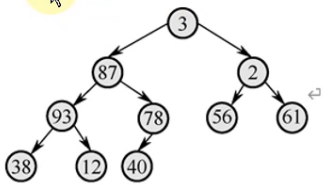
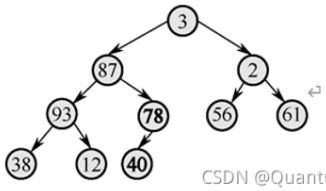
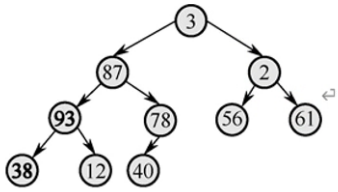
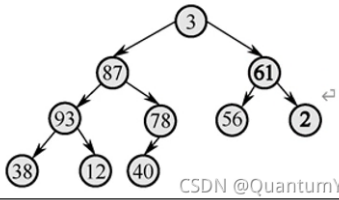
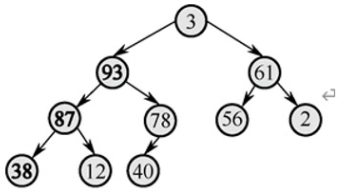
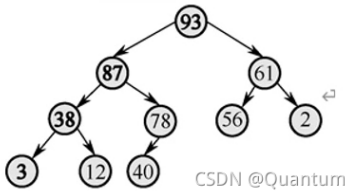
- 假设排序表为`L[1…n]`，第i趟排序即从`L[i…n]`中选择关键字最小的元素与L(i)交换，每一趟排序可以确定一个元素的最终位置，这样经过n-1趟排序就可使得整个排序表有序
- 选择排序每一次选择最小的值

```
void SelectSort(ElemType A[],int n)
{
    int i,j,min;//min记录最小的元素的下标
    for(i=0;i<n-1;i++)//最多可以为8
    {
        min=i;
        for(j=i+1;j<n;j++)//j最多可以为9
        {
            if(A[j]<A[min])
                min=j;
        }
        if(min!=i)
```

```
        {
            swap(A[i],A[min]);
        }
    }
}
```

堆排序 （重点）

- 堆就是用数组实现的二叉树，所以它没有使用父指针或者子指针。堆根据“堆属性”来排序，“堆属性”决定了树中节点的位置。

1	数组元素情况	3 87 2 93 78 56 61 38 12 40
	数组元素对应的二叉树	
2	数组元素情况	3 87 2 93 **78** 56 61 38 12 **40**
	数组元素对应的二叉树，arr[4]与 arr[9]进行比较	
3	数组元素情况	3 87 2 **93** 78 56 61 **38** 12 40
	数组元素对应的二叉树，arr[3]与 arr[7]进行比较	
4	数组元素情况	3 87 **61** 93 78 56 **2** 38 12 40
	数组元素对应的二叉树，arr[2]与 arr[6]进行比较，发生交换	
5	数组元素情况	3 **93** 61 **87** 78 56 2 38 12 40
	数组元素对应的二叉树，arr[1]与 arr[3]进行比较，发生交换，发生交换后 arr[3]重新作为父结点，与孩子结点比较	
6	数组元素情况	**93 87** 61 **38** 78 56 2 **3** 12 40
	数组元素对应的二叉树，arr[0]与 arr[1]进行比较，发生交换，发生交换后 arr[1]重新作为父结点，与孩子结点比较	

CSDN @QuantumYou

```
/调整某个父亲节点
void AdjustDown(ElemType A[],int k,int len)
{
    int i;
```

```
    A[0]=A[k];
    for(i=2*k;i<=len;i*=2)
    {
        if(i<len&&A[i]<A[i+1])//左子节点与右子节点比较大小
            i++;
        if(A[0]>=A[i])
            break;
        else{
            A[k]=A[i];
            k=i;
        }
    }
    A[k]=A[0];
}
//用数组去表示树    层次建树
void BuildMaxHeap(ElemType A[],int len)
{
    for(int i=len/2;i>0;i--)
    {
        AdjustDown(A,i,len);
    }
}
void HeapSort(ElemType A[],int len)
{
    int i;
    BuildMaxHeap(A,len);//建立大顶堆
    for(i=len;i>1;i--)
    {
        swap(A[i],A[1]);
        AdjustDown(A,1,i-1);
    }
}
//调整子树
void AdjustDown1(ElemType A[], int k, int len)
{
    int dad = k;
    int son = 2 * dad + 1; //左孩子下标
    while (son<=len)
    {
        if (son + 1 <= len && A[son] < A[son + 1])//看下有没有右孩子，比较左右孩子选大的
        {
            son++;
        }
        if (A[son] > A[dad])//比较孩子和父亲
        {
            swap(A[son], A[dad]);
            dad = son;
            son = 2 * dad + 1;
        }
        else {
            break;
        }
    }
}
void HeapSort1(ElemType A[], int len)
{
    int i;
```

```
    //建立大顶堆
    for (i = len / 2; i >= 0; i--)
    {
        AdjustDown1(A, i, len);
    }
    swap(A[0], A[len]);//交换顶部和数组最后一个元素
    for (i = len - 1; i > 0; i--)
    {
        AdjustDown1(A, 0, i);//剩下元素调整为大根堆
        swap(A[0], A[i]);
    }
}
```

各大排序时间复杂度

排序方式	时间复杂度			空间复杂度	稳 定 性	复 杂 性
	平均情况	最坏情况	最好情况			
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定	简单
希尔排序	$O(n^{1.3})$			$O(1)$	不稳定	较复杂
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定	简单
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	$O(\log_2 n)$	不稳定	较复杂
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定	简单
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定	较复杂
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定	较复杂
基数排序	$O(d(n+r))$	$O(d(n+r))$	$O(d(n+r))$	$O(r)$	稳定	较复杂

CSDN @QuantumYou