内部排序

插入排序

快速排序

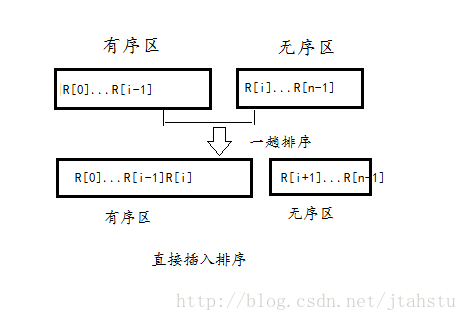
堆及选择排序

归并排序

基数排序

10.1 概述  
1、排序：将一个数据元素（或记录）的任意序列，重新排列成一个按关键字有序的序列。  
2、排序：无序->有序  
性质：稳定排序/不稳定排序  
分类：内部排序，外部排序  
常用内部排序：插入排序、希尔排序、快速排序、选择排序、堆排序、归并排序、基数排序  
  
10.2 插入排序  
1.1、插入排序的思想：将一个元素插入到一个有序表中  
插入位置方法不同：直接插入、折半插入、2路插入、表插入  
1.2、直接插入排序：最简单的排序方法

思想：将一个元素向一个有序序列插入



1.3、直接插入排序算法：

**[html]** [view plaincopy](http://blog.csdn.net/jtahstu/article/details/42390449)

1. void InsertSort(RecType R[],int n) //对R[0..n-1]按递增有序进行直接插入排序
2. {
3. int i,j,k;
4. RecType tmp;
5. for (i=1; i<n; i++)
6. {
7. tmp=R[i];
8. j=i-1;            //从右向左在有序区R[0..i-1]中找R[i]的插入位置
9. while (j>=0 && tmp.key<R[j].key)
10. {
11. R[j+1]=R[j]; //将关键字大于R[i].key的记录后移
12. j--;
13. }
14. R[j+1]=tmp;      //在j+1处插入R[i]
15. }
16. }

1.4、算法分析  
时间复杂度O(n2)，空间复杂度O(1)

1.5、稳定的排序方法

2、折半插入排序

2.1、思想：查找过程用折半查找方法

2.2、折半插入算法：

**[html]** [view plaincopy](http://blog.csdn.net/jtahstu/article/details/42390449)

1. void InsertSort1(RecType R[],int n) //对R[0..n-1]按递增有序进行直接插入排序
2. {
3. int i,j,low,high,mid;
4. RecType tmp;
5. for (i=1; i<n; i++)
6. {
7. tmp=R[i];
8. low=0;
9. high=i-1;
10. while (low<=high)
11. {
12. mid=(low+high)/2;
13. if (tmp.key<R[mid].key)
14. high=mid-1;
15. else
16. low=mid+1;
17. }
18. for (j=i-1; j>=high+1; j--)
19. R[j+1]=R[j];
20. R[high+1]=tmp;
21. }
22. }

2.4、算法分析  
时间复杂度O(n2)，空间复杂度O(1)

2.5、稳定的排序方法

3、2-路插入排序  
减少直接插入法的移动元素的个数，分成2路子有序序列

需要n个记录的辅助空间

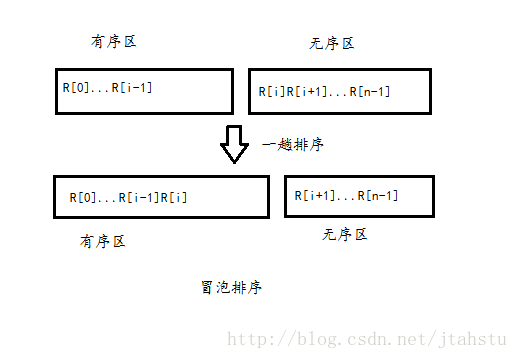
4、表插入排序  
对静态链表作插入排序  
将一个记录插入到已排序好的的有序表中  
以修改2n次指针值代替移动记录  
  
  
5.1、希尔排序  
缩小增量排序，属于插入排序  
5.2、算法思想：先将整个待排序记录分割成若干个子序列分别进行直接插入排序，待整个序列中的记录基本有序时，再进行一次全体记录的插入排序  
5.3、希尔排序算法

**[html]** [view plaincopy](http://blog.csdn.net/jtahstu/article/details/42390449)

1. void ShellSort(RecType R[],int n) //希尔排序算法
2. {
3. int i,j,gap,k;
4. RecType tmp;
5. gap=n/2; //增量置初值
6. while (gap>0)
7. {
8. for (i=gap; i<n; i++) //对所有相隔gap位置的所有元素组进行排序
9. {
10. tmp=R[i];
11. j=i-gap;
12. while (j>=0 && tmp.key<R[j].key)//对相隔gap位置的元素组进行排序
13. {
14. R[j+gap]=R[j];
15. j=j-gap;
16. }
17. R[j+gap]=tmp;
18. j=j-gap;
19. }
20. gap=gap/2; //减小增量
21. }
22. }

5.4、算法分析  
时间复杂度O(n1.3)，空间复杂度O(1)  
5.5、不稳定的排序方法  
  
10.3 交换排序  
1、冒泡排序

1.1、思想：依次比较第i个关键字和第i+1个关键字，大者排后，一趟得到最大值



1.2、冒泡排序算法：

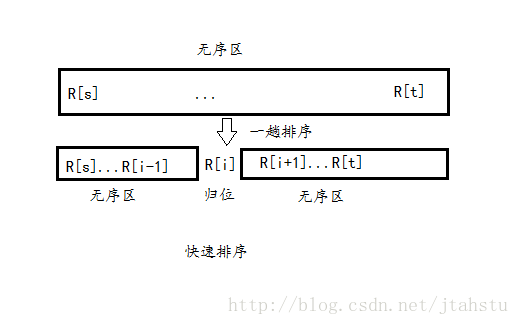
**[html]** [view plaincopy](http://blog.csdn.net/jtahstu/article/details/42390449)

1. void BubbleSort(RecType R[],int n)
2. {
3. int i,j,k;
4. RecType tmp;
5. for (i=0; i<n-1; i++)
6. {
7. for (j=n-1; j>i; j--) //比较,找出本趟最小关键字的记录
8. if (R[j].key<R[j-1].key)
9. {
10. tmp=R[j];  //R[j]与R[j-1]进行交换,将最小关键字记录前移
11. R[j]=R[j-1];
12. R[j-1]=tmp;
13. }
14. }
15. }

1.3、算法分析  
时间复杂度O(n2)，空间复杂度O(1)  
1.4、稳定的排序方法

2、快速排序

2.1、思想：一趟排序将序列分成两个部分，前者小于某个值（第一个值），后者大于某个值（第一个值）。之后再次分别快速排序。



2.2、快速排序算法：

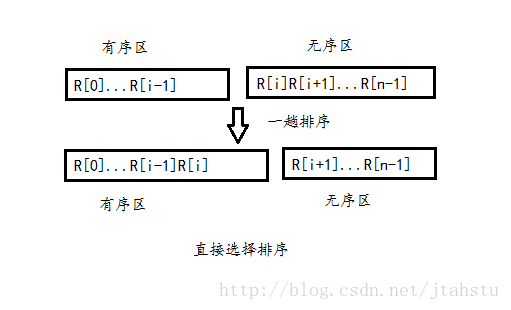
**[html]** [view plaincopy](http://blog.csdn.net/jtahstu/article/details/42390449)

1. void QuickSort(RecType R[],int s,int t) //对R[s]至R[t]的元素进行快速排序
2. {
3. int i=s,j=t;
4. RecType tmp;
5. if (s<t) //区间内至少存在两个元素的情况
6. {
7. tmp=R[s];     //用区间的第1个记录作为基准
8. while (i!=j)  //从区间两端交替向中间扫描,直至i=j为止
9. {
10. while (j>i && R[j].key>=tmp.key)
11. j--;   //从右向左扫描,找第1个小于tmp.key的R[j]
12. R[i]=R[j]; //找到这样的R[j],R[i]"R[j]交换
13. while (i<j && R[i].key<=tmp.key)
14. i++; //从左向右扫描,找第1个大于tmp.key的记录R[i]
15. R[j]=R[i]; //找到这样的R[i],R[i]"R[j]交换
16. }
17. R[i]=tmp;
18. QuickSort(R,s,i-1);    //对左区间递归排序
19. QuickSort(R,i+1,t);    //对右区间递归排序
20. }
21. }

2.3、算法分析  
时间复杂度O(nlog2(n))，空间复杂度O(log2(n))  
2.4、不稳定的排序方法  
  
10.4 选择排序

选择排序基本思想：每一趟在n-i+1个记录中选取关键字最小的记录作为有序序列中第i个记录

1、直接选择排序  
1.1、思想：从当前无序区中选出关键字最小的元素R[k],将它与无序区的第一个元素R[i]交换，使R[0..i]和R[i+1..n-1]分别变为新的有序区和新的无序区

  
1.2、直接选择排序算法：

**[html]** [view plaincopy](http://blog.csdn.net/jtahstu/article/details/42390449)

1. void SelectSort(RecType R[],int n)
2. {
3. int i,j,k,l;
4. RecType temp;
5. for (i=0; i<n-1; i++)  //做第i趟排序
6. {
7. k=i;
8. for (j=i+1; j<n; j++) //在当前无序区R[i..n-1]中选key最小的R[k]
9. if (R[j].key<R[k].key)
10. k=j;           //k记下目前找到的最小关键字所在的位置
11. if (k!=i)          //交换R[i]和R[k]
12. {
13. temp=R[i];
14. R[i]=R[k];
15. R[k]=temp;
16. }
17. }
18. }

1.3、算法分析  
时间复杂度：O(n2) , 空间复杂度：O(1)  
1.4、不稳定的排序方法  
  
2、堆排序  
堆的定义：n个元素的序列{k1,k2,...,kn}当且仅当满足下列关系时，称为堆：Ki<=K2\*i且Ki<=K2\*i+1或Ki>=K2\*i且Ki>=K2\*i+1

(1<=i<=n/2向上取整)

2.1、堆排序过程：

  输出堆顶元素  
  将剩余的n-1个元素序列重新建成一个堆，再输出堆顶元素  
实现堆排序需要解决2个问题：  
  如何从一个无序的序列建立一个堆

  如何在输出堆顶元素后，调整剩余元素为一个新的堆

2.2、堆排序的算法：

**[html]** [view plaincopy](http://blog.csdn.net/jtahstu/article/details/42390449)

1. void sift(RecType R[],int low,int high)
2. {
3. int i=low,j=2\*i;     //R[j]是R[i]的左孩子
4. RecType temp=R[i];
5. while (j<=high)
6. {
7. if (j<high && R[j].key<R[j+1].key) //若右孩子较大,把j指向右孩子
8. j++;     //变为2i+1
9. if (temp.key<R[j].key)
10. {
11. R[i]=R[j];              //将R[j]调整到双亲结点位置上
12. i=j;                    //修改i和j值,以便继续向下筛选
13. j=2\*i;
14. }
15. else break;                 //筛选结束
16. }
17. R[i]=temp;                      //被筛选结点的值放入最终位置
18. }
19. void HeapSort(RecType R[],int n)
20. {
21. int i;
22. RecType tmp;
23. for(i=n/2; i>=1; i--) //循环建立初始堆
24. sift(R,i,n);
25. for(i=n; i>=2; i--) //进行n-1趟堆排序，每一趟堆排序的袁旭个数减一
26. {
27. tmp=R[1]; //将最后一个元素同当前区间内R[1]对换
28. R[1]=R[i];
29. R[i]=tmp;
30. sift(R,1,n-1); //筛选R[1]节点，得到i-1个节点的堆
31. }
32. }

2.3、算法分析  
时间复杂度：O(nlog2(n)) , 空间复杂度：O(1)  
2.4、不稳定的排序方法

10.5 归并排序.

1.1、思想：多次将两个或两个以上的有序表组合成一个新的有序表  
1.2、归并排序算法：

**[html]** [view plaincopy](http://blog.csdn.net/jtahstu/article/details/42390449)

1. void Merge(RecType R[],int low,int mid,int high)
2. {
3. RecType \*R1;
4. int i=low,j=mid+1,k=0; //k是R1的下标,i、j分别为第1、2段的下标
5. R1=(RecType \*)malloc((high-low+1)\*sizeof(RecType));  //动态分配空间
6. while (i<=mid && j<=high)     //在第1段和第2段均未扫描完时循环
7. if (R[i].key<=R[j].key)    //将第1段中的记录放入R1中
8. {
9. R1[k]=R[i];
10. i++;
11. k++;
12. }
13. else                       //将第2段中的记录放入R1中
14. {
15. R1[k]=R[j];
16. j++;
17. k++;
18. }
19. while (i<=mid)                //将第1段余下部分复制到R1
20. {
21. R1[k]=R[i];
22. i++;
23. k++;
24. }
25. while (j<=high)                 //将第2段余下部分复制到R1
26. {
27. R1[k]=R[j];
28. j++;
29. k++;
30. }
31. for (k=0,i=low; i<=high; k++,i++) //将R1复制回R中
32. R[i]=R1[k];
33. }
34. void MergePass(RecType R[],int length,int n) //对整个数序进行一趟归并
35. {
36. int i;
37. for (i=0; i+2\*length-1<n; i=i+2\*length) //归并length长的两相邻子表
38. Merge(R,i,i+length-1,i+2\*length-1);
39. if (i+length-1<n)                //余下两个子表,后者长度小于length
40. Merge(R,i,i+length-1,n-1);  //归并这两个子表
41. }
42. void MergeSort(RecType R[],int n) //自底向上的二路归并算法
43. {
44. int length;
45. for (length=1; length<n; length=2\*length) //进行log2n趟归并
46. MergePass(R,length,n);
47. }

1.3、算法分析  
时间复杂度：O(nlog2(n)) , 空间复杂度：O(n)  
1.4、稳定的排序方法

10.6 基数排序（了解）

思想：基数排序是通过“分配”和“收集”过程来实现排序的，不需进行关键字间的比较，是一种借助多关键字排序的思想对单关键字排序的方法。  
基数排序有两种：最低位优先（LSD）和最高位优先（MSD）

最低位优先的过程是：先按最低位的值对元素进行排序，在此基础上，再按次低位进行排序，依此类推，由低位向高位，每趟都是根据关键字的一位并在前一趟的基础上对所有元素进行排序，直至最高位。

10.7 各种内排序方法的比较和选择

如图：



影响因素：  
1、待排序的元素数目n(问题规模)  
2、元素的大小（每个元素的规模）  
3、关键字的结构及其初始状态  
4、对稳定性的要求  
5、语言工具的条件  
6、存储结构  
7、时间和空间复杂度等