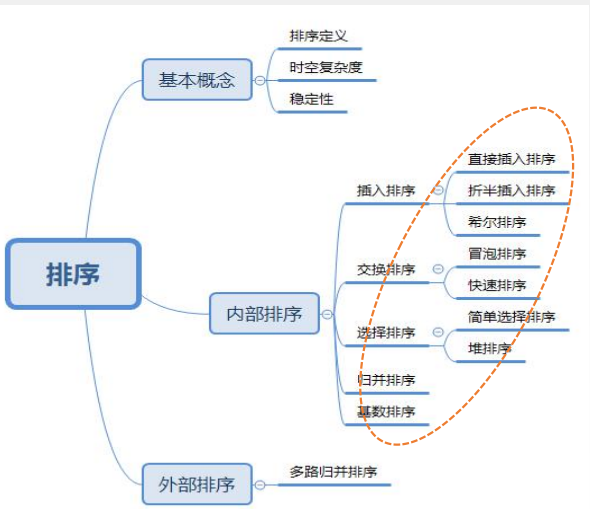
王道考研——数据结构

WWW.CSKAOYAN.COM

第七章 排序

本章总览

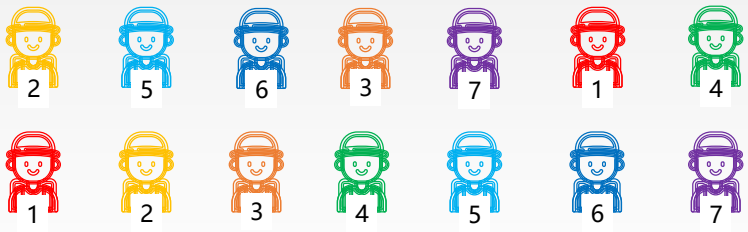


本节内容

排序

基本概念

排序



排序 重新排列表中的元素，使表中的元素满足按关键字递增或递减

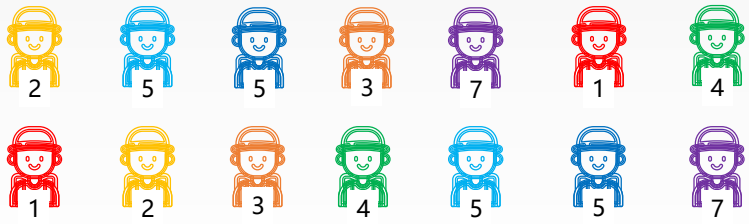
输入 n个记录R1，R2，R3，…，Rn，对应关键字k1，k2，k3，…，kn

输出 输入序列的重新排列R1'，R2'，R3'，…，Rn'，k1'≤k2'≤….≤kn'
（其中≤可以换成其他有比较含义的符号）

排序

排序算法的稳定性

若待排序表中有两个元素Ri和Rj，其对应的关键字ki=kj，且在排序前Ri在Rj前面，若使用某排序算法后，Ri仍然在Rj前面。则称这个排序算法是稳定的，否则称排序算法不稳定。



稳定！

排序

算法的稳定性

若待排序表中有两个元素Ri和Rj，其对应的关键字ki=kj，且在排序前Ri在Rj前面，若使用某排序算法后，Ri仍然在Rj前面。则称这个排序算法是稳定的，否则称排序算法不稳定。

★算法的稳定性是算法的性质，并不能衡量一个算法的优劣

不稳定！

排序

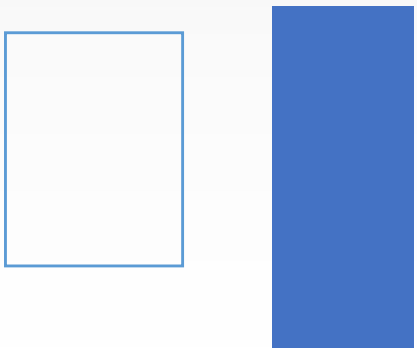
内部排序 指在排序期间元素全部存放在内存中的排序

外部排序 指在排序期间元素无法全部同时存放在内存中，必须在排序的过程中根据要求不断地在内、外存之间进行移动

排序

内部排序 指在排序期间元素全部存放在内存中的排序

外部排序 指在排序期间元素无法全部同时存放在内存中，必须在排序的过程中根据要求不断地在内、外存之间进行移动



王道考研/CSKAOYAN.COM

排序

移动和比较

★ 时空复杂度决定内部排序算法的性能

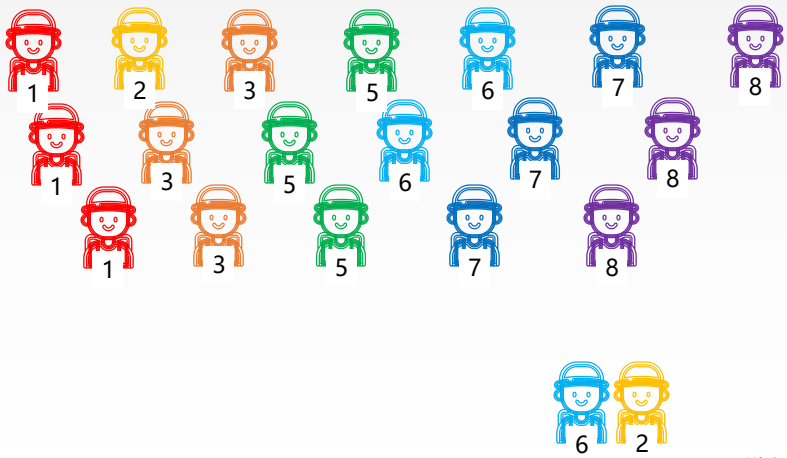
本节内容

排序

插入排序
直接插入排序

直接插入排序

插入排序 每次将一个待排序的序列插入到一个前面已排好序的子序列当中



直接插入排序

直接插入排序

L[1...n] | 有序序列L[1...i-1] | L(i) | 无序序列L[i+1...n]

有序序列L[1...i] | 无序序列L[i+2...n]

· 初始L[1]是一个已经排好序的子序列
· 对于元素L(i)(L(2)~L(n))插入到前面已经排好序的子序列当中：

1）查找出L(i)在L[1...i-1]中的插入位置k
2）将L[k...i-1]中的所有元素全部后移一个位置
3）将L(i)复制到L(k)

空间复杂度 O(1)

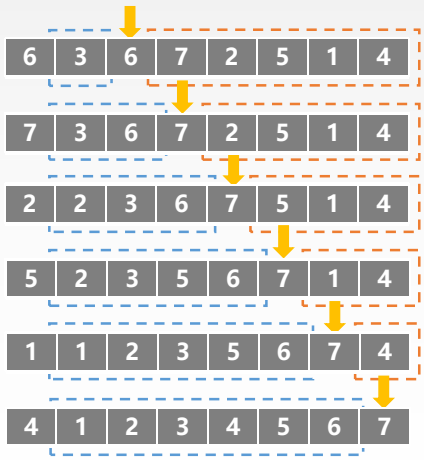
直接插入排序

直接插入排序

```
void InsertSort(ElemType A[],int n){
    int i,j;
    for(i=2;i<=n;i++){
        A[0]=A[i];
        for(j=i-1;A[0].key<A[j].key;j--)
            A[j+1]=A[j];
        A[j+1]=A[0];
    }
}
```

O(n)　O(n^2)　O(n^2)

稳定！　顺序存储和链式存储



排序

O(n^2)　O(1)　稳定　顺序存储和链式存储

本节内容

排序

插入排序
折半插入排序

折半插入排序

折半插入排序

折半插入排序

```
void BInsertSort(ElemType A[],int n){
    int i,j;
    int low, high, mid;
    for(i=2; i<=n; i++){
        A[0]=A[i];
        low=1; high=i-1;
        while(low<=high){
            mid=(low+high)/2;
            if(A[mid].key > A[0].key)
                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];
    }
}
```

折半查找

$O(\log_2 n)$

移动

$O(n)$

$O(n^2)$　　$O(1)$　　稳定!　　顺序存储

排序

$O(n^2)$　$O(1)$　稳定　顺序存储和链式存储

$O(n^2)$　$O(1)$　稳定　顺序存储

本节内容

排序

插入排序
希尔排序

希尔排序

希尔排序 缩小增量排序

$d_3 d d_2 d_1$

$d_1, d_2, d_3, \ldots, d_k$

基本思想：

先将排序表分割成d个形如L[i, i+d, i+2d, ..., i+kd]的“特殊”子表，分别进行直接插入排序，当整个表中的元素已呈“基本有序时”，再对全体记录进行一次直接插入排序。

Shell's idea $d_1 = \lfloor n/2 \rfloor$ $d_{i+1} = \lfloor d_i / 2 \rfloor$ 直到最后一个$d_k = 1$

希尔排序

```
void ShellSort(ElemType A[], int n){
    for(int dk=n/2; dk>=1; dk=dk/2)
        for(int i=dk+1; i<=n; ++i)
            if(A[i].key<A[i-dk].key){
                A[0]=A[i];
                for(int j=i-dk; j>0&&A[0].key<A[j].key; j-=dk)
                    A[j+dk]=A[j];
                A[j+dk]=A[0];
            }
}
```

	3	6	5	2	5	1	4

	2	5	1	3	6	5	4

	1	2	3	4	5	5	6

$O(n^2)$ $O(1)$

不稳定！ 顺序存储

排序

$O(n^2)$ $O(1)$ 稳定 顺序存储和链式存储
$O(n^2)$ $O(1)$ 稳定 顺序存储
$O(n^2)$ $O(1)$ 不稳定 顺序存储

本节内容

排序

交换排序
冒泡排序

冒泡排序

冒泡排序

基本思想：

假设待排序表长为n，从后往前（从前往后）两两比较相邻元素的值，若为逆序（即A[i-1]>A[i]），则交换他们直到序列比较结束。

3 7 2 8 6 9 1 4 5

3 7 2 8 6 9 1 4 5

3 2 7 8 6 9 1 4 5

3 2 7 8 6 9 1 4 5

3 2 7 6 8 9 1 4 5

3 2 7 6 8 9 1 4 5

3 2 7 6 8 1 9 4 5

3 2 7 6 8 1 4 9 5

3 2 7 6 8 1 4 5 9

★一次冒泡会将一个元素放置到它最终的位置上

冒泡排序

```
void BubbleSort(ElemType A[], int n){
    for(int i=0; i<n-1; i++){
        bool flag=false;
        for(int j=n-1; j>i; j--)
            if(A[j-1].key > A[j].key){
                swap(A[j-1], A[j]);
                flag = true;
            }
        if(flag == false)
            return;
    }
}
```

O(n)　O(n^2)　O(n^2)

O(1)　稳定!　顺序存储和链式存储

3 7 2 8 6 9 1 4 5

1 3 7 2 8 6 9 4 5

1 2 3 7 4 8 6 9 5

1 2 3 4 7 5 8 6 9

1 2 3 4 5 7 6 8 9

1 2 3 4 5 6 7 8 9

排序

O(n^2)　O(1)　稳定　顺序存储和链式存储

O(n^2)　O(1)　稳定　顺序存储

O(n^2)　O(1)　不稳定　顺序存储

O(n^2)　O(1)　稳定　顺序存储和链式存储

本节内容

排序

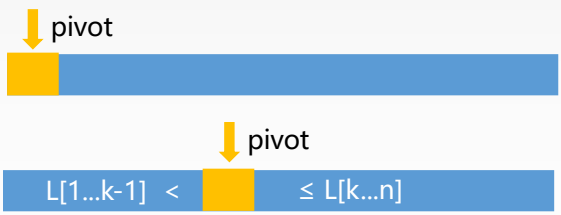
交换排序
快速排序

快速排序

快速排序

基本思想：

在待排序表L[1...n]中任取一个元素pivot作为基准，通过一趟排序将待排序表划分为具有如下特点的两部分：



一趟Partition

★一次划分会将一个元素pivot放置到它最终的位置上

快速排序

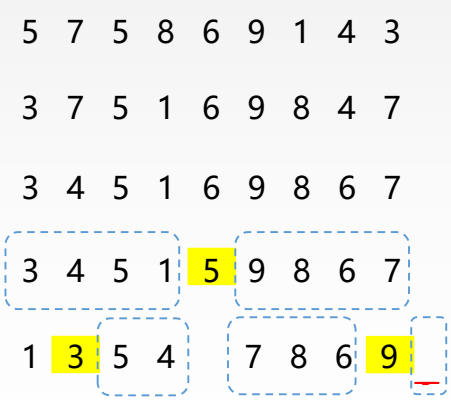
Partition

基本思路：
初始化标记low为划分部分第一个元素的位置， high为最后一个元素的位置，然后不断地移动两标记并交换元素：
1）high向前移动找到第一个比pivot小的元素
2）low向后移动找到第一个比pivot大的元素
3）交换当前两个位置的元素
4）继续移动标记，执行1），2），3）的过程，直到low大于等于high为止。

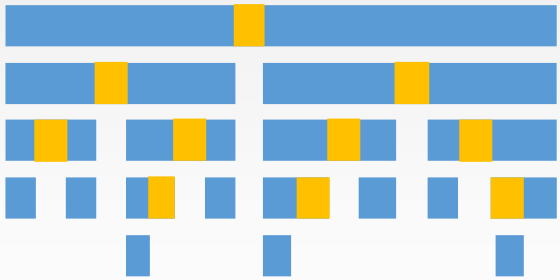
快速排序

```
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;
}
void QuickSort(ElemType A[], int low, int high){
    if(low < high){
        int pivotpos = Partition(A, low, high);
        QuickSort(A, low, pivotpos-1);
        QuickSort(A, pivotpos+1, high);
    }
}
```

不稳定！



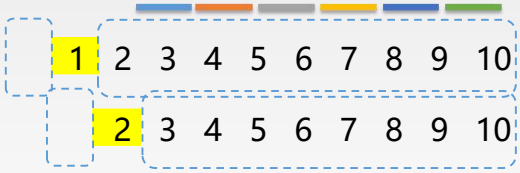
快速排序



最好、平均时间复杂度O($n\log_2 n$)

最好、平均空间复杂度O($\log_2 n$)

快速排序



.....

初始基本有序或逆序

最坏时间复杂度O(n^2)

最坏空间复杂度O(n)

排序

$O(n^2)$ $O(1)$ 稳定 顺序存储和链式存储

$O(n^2)$ $O(1)$ 稳定 顺序存储

$O(n^2)$ $O(1)$ 不稳定 顺序存储

$O(n^2)$ $O(1)$ 稳定 顺序存储和链式存储

$O(n\log_2 n)$ $O(\log_2 n)$ 不稳定 顺序存储（链式存储）

王道考研/CSKAOYAN.COM

本节内容

排序

选择排序
直接选择排序

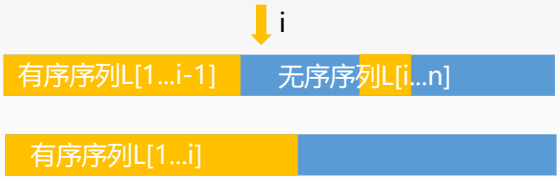
王道考研/CSKAOYAN.COM

选择排序

选择排序

基本思想：

每一趟在后面n-i+1（i=1,2,,n-1)个待排序元素中选取关键字最小的元素，作为有序子序列的第i个元素，直到n-1趟做完，待排序元素只剩下1个。



★一趟排序会将一个元素放置在最终的位置上

选择排序

直接选择排序

```
void SelectSort(ElemType A[], int n){
    for(int i=0; i<n-1; i++){
        int min=i;
        for(int j=i+1; j<n; j++)
            if(A[j]<A[min])
                min=j;
        if(min!=i)
            swap(A[i], A[min]);
    }
}
```

不稳定！

$O(n^2)$　$O(1)$　顺序存储和链式存储

★时间复杂度与初始序列无关

3	1	5	3	2	4
1	3	5	3	2	4
1	2	5	3	3	4
1	2	3	5	3	4
1	2	3	3	5	4
1	2	3	3	4	5

排序

$O(n^2)$ $O(1)$ 稳定 顺序存储和链式存储

$O(n^2)$ $O(1)$ 稳定 顺序存储

$O(n^2)$ $O(1)$ 不稳定 顺序存储

$O(n^2)$ $O(1)$ 稳定 顺序存储和链式存储

$O(n\log_2 n)$ $O(\log_2 n)$ 不稳定 顺序存储（链式存储）

$O(n^2)$ $O(1)$ 不稳定 顺序存储和链式存储

本节内容

排序

选择排序
堆排序

王道考研/CSKAOYAN.COM

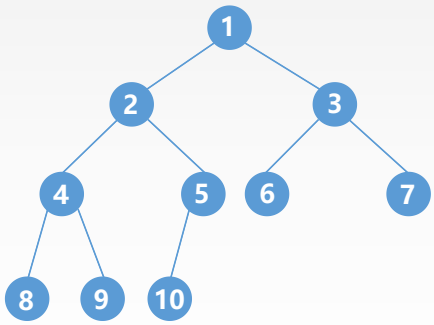
堆排序

堆

n个关键字序列L[1...n]称为堆，当且仅当该序列满足：
① 若L(i)≤L(2i)且L(i)≤L(2i+1)，则称该堆为 **小根堆**
② 若L(i)≥L(2i)且L(i)≥L(2i+1)，则称该堆为 **大根堆**
(1≤i≤⌊n/2⌋)

	1	2	3	4	5	6	7	8	9	10
小根堆	7	9	16	13	11	19	21	17	15	12

	1	2	3	4	5	6	7	8	9	10
大根堆	36	24	11	21	17	10	8	13	15	16



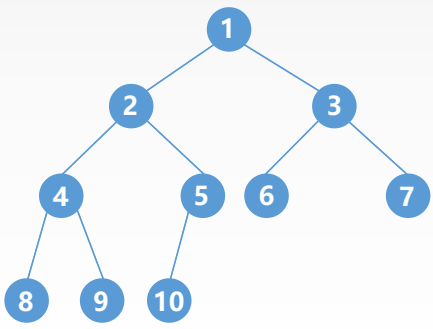
在排序过程中将L[1...n]视为一棵**完全二叉树**的顺序存储结构

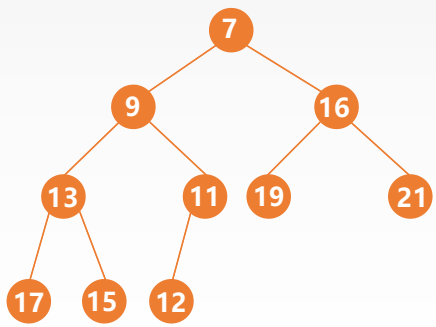
堆排序

小根堆

	1	2	3	4	5	6	7	8	9	10
	7	9	16	13	11	19	21	17	15	12

① 若L(i)≤L(2i)且L(i)≤L(2i+1)，则称该堆为 **小根堆**

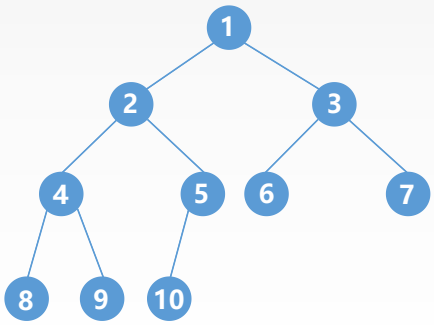


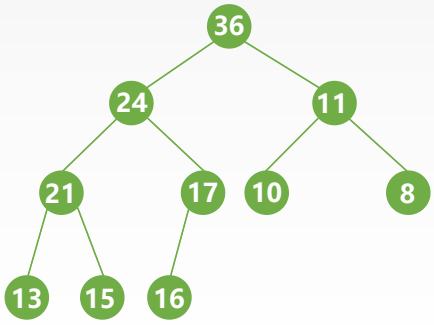


堆排序

大根堆

1	2	3	4	5	6	7	8	9	10
36	24	11	21	17	10	8	13	15	16





堆排序

堆的初始化 大根堆

对所有具有双亲结点含义编号从大到小（$\lfloor n/2 \rfloor$~1）做出如下调整：

1）若孩子结点皆小于双亲结点，则该结点的调整结束
2）若存在孩子结点大于双亲结点，则将最大的孩子结点与双亲结点交换，并对该孩子结点进行1）、2），直到出现1）或到叶节点为止

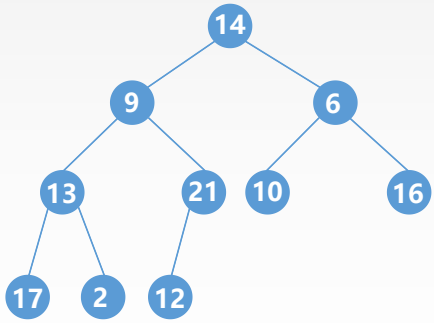
堆排序

堆的初始化 大根堆

对所有具有双亲结点含义编号从大到小（⌊n/2⌋~1）做出如下调整：

1）若孩子结点皆小于双亲结点，则该结点的调整结束
2）若存在孩子结点大于双亲结点，则将最大的孩子结点与双亲结点交换，并对该孩子结点进行1）、2），直到出现1）或到叶节点为止

1	2	3	4	5	6	7	8	9	10
14	9	6	13	21	10	16	17	2	12



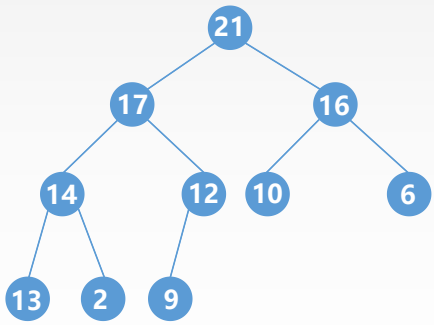
堆排序

堆的初始化 大根堆

对所有具有双亲结点含义编号从大到小（⌊n/2⌋~1）做出如下调整：

1）若孩子结点皆小于双亲结点，则该结点的调整结束
2）若存在孩子结点大于双亲结点，则将最大的孩子结点与双亲结点交换，并对该孩子结点进行1）、2），直到出现1）或到叶节点为止

1	2	3	4	5	6	7	8	9	10
21	17	16	14	12	10	6	13	2	9



堆排序

堆的初始化 大根堆

```
void BuildMaxHeap(ElemType A[], int len){
    for(int i=len/2; i>0; i--)
        AdjustDown(A, i, len);
}
```

```
void AdjustDown(ElemType A[], int k, int len){
    A[0]=A[k];
    for(int i=2*k; i<=len; i*=2){
        if(i<len && A[i]<A[i+1])
            i++;
        if(A[0]>=A[i])
            break;
        else{
            A[k]=A[i];
            k=i;
        }
    }
    A[k]=A[0];
}
```

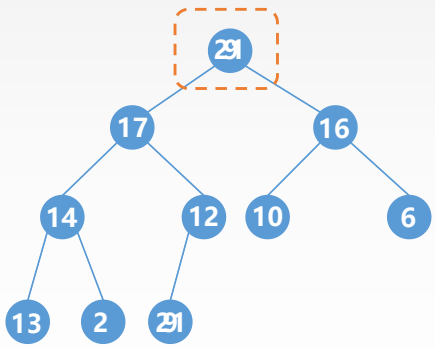
O（h）

初始建堆O（n）

堆排序

堆排序 不断地输出堆顶元素，并向下调整

```
void HeapSort(ElemType A[], int len){
    BuildMaxHeap(A,  len);
    for(int i=len; i>1; i--){
        Swap(A[i], A[1]);
        AdjustDown(A, 1, i-1);
    }
}
```

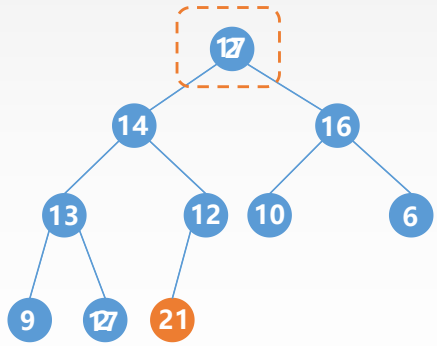


输出序列：21

堆排序

堆排序 不断地输出堆顶元素，并向下调整

```
void HeapSort(ElemType A[], int len){
    BuildMaxHeap(A,  len);
    for(int i=len; i>1; i--){
        Swap(A[i], A[1]);
        AdjustDown(A, 1, i-1);
    }
}
```



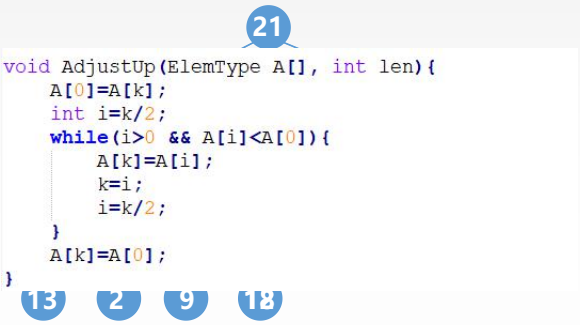
$O(nlog_2n)$　　$O(1)$

不稳定　　顺序存储（链式存储）

输出序列：21　17

堆排序

堆的插入 将新结点放置在末端然后进行向上调整




```
void AdjustUp(ElemType A[], int len){
    A[0]=A[k];
    int i=k/2;
    while(i>0 && A[i]<A[0]){
        A[k]=A[i];
        k=i;
        i=k/2;
    }
    A[k]=A[0];
}
```

排序

$O(n^2)$ $O(1)$ 稳定 顺序存储和链式存储

$O(n^2)$ $O(1)$ 稳定 顺序存储

$O(n^2)$ $O(1)$ 不稳定 顺序存储

$O(n^2)$ $O(1)$ 稳定 顺序存储和链式存储

$O(n\log_2 n)$ $O(\log_2 n)$ 不稳定 顺序存储（链式存储）

$O(n^2)$ $O(1)$ 不稳定 顺序存储和链式存储

$O(n\log_2 n)$ $O(1)$ 不稳定 顺序存储（链式存储）

本节内容

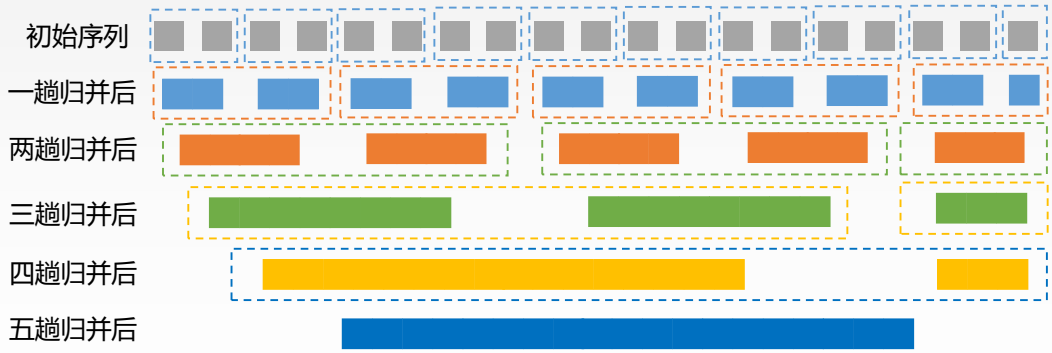
排序

归并排序

王道考研/CSKAOYAN.COM

归并排序

归并排序



2路归并排序

归并排序

合并两个有序线性表

```
ElemType *B=(ElemType *)malloc((n+1)*sizeof(ElemType));
void Merge(ElemType A[], int low, int mid, int high){
    for(int k=low; k<=high; k++)
        B[k]=A[k];
    for(int i=low,int j=mid+1,int k=i; i<=mid && j<=high; k++){
        if(B[i]<=B[j])
            A[k]=B[i++];
        else
            A[k]=B[j++];
    }
    while(i<=mid)
        A[k++]=B[i++];
    while(j<=high)
        A[k++]=B[j++];
}
```

O(high-low+1)

归并排序

归并排序

```
void MergeSort(ElemType A[], int low, int high){
    if(low<high){
        int mid = (low+high)/2;
        MergeSort(A, low, mid);
        MergeSort(A, mid+1, high);
        Merge(A, low, mid, high);
    }
}
```

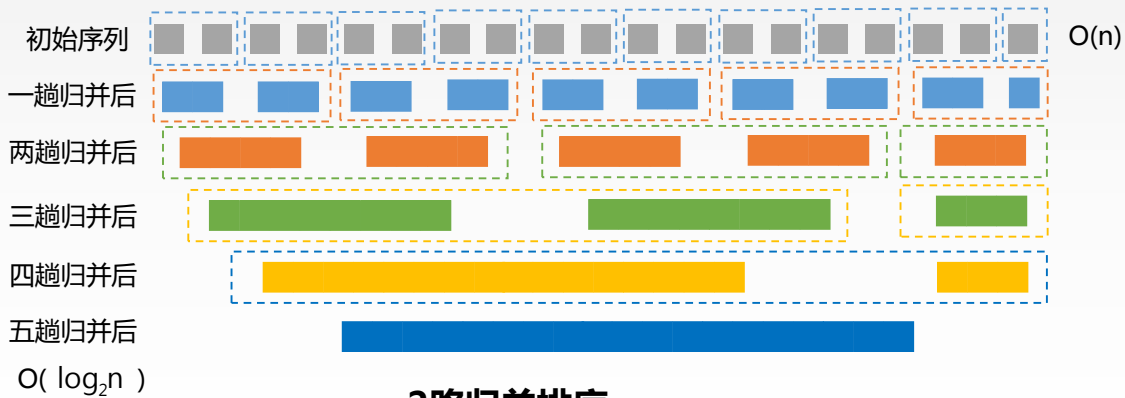
29 15 36 7 18 2 64 40 27

29 15 36 7 18 2 64 40 27

$O(n\log_2 n)$

归并排序

归并排序



2路归并排序

归并排序

归并排序

```
void MergeSort(ElemType A[], int low, int high){
    if(low<high){
        int mid = (low+high)/2;
        MergeSort(A, low, mid);
        MergeSort(A, mid+1, high);
        Merge(A, low, mid, high);
    }
}
```

2 7 15 18 27 29 36 40 64

$O(n\log_2 n)$ O(n)

稳定!

归并排序

合并两个有序线性表

```
ElemType *B=(ElemType *)malloc((n+1)*sizeof(ElemType));
void Merge(ElemType A[], int low, int mid, int high){
    for(int k=low; k<=high; k++)
        B[k]=A[k];
    for(int i=low,int j=mid+1,int k=i; i<=mid && j<=high; k++){
        if(B[i]<=B[j])
            A[k]=B[i++];
        else
            A[k]=B[j++];
    }
    while(i<=mid)
        A[k++]=B[i++];
    while(j<=high)
        A[k++]=B[j++];
}
```

O(high-low+1)

归并排序

归并排序

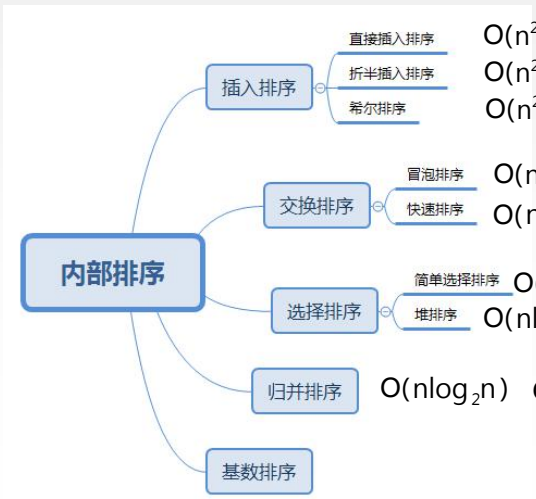
```
void MergeSort(ElemType A[], int low, int high){
    if(low<high){
        int mid = (low+high)/2;
        MergeSort(A, low, mid);
        MergeSort(A, mid+1, high);
        Merge(A, low, mid, high);
    }
}
```

2 7 15 18 27 29 36 40 64

$O(n\log_2 n)$ $O(n)$

稳定! 顺序存储和链式存储

排序



直接插入排序 $O(n^2)$ $O(1)$ 稳定 顺序存储和链式存储

折半插入排序 $O(n^2)$ $O(1)$ 稳定 顺序存储

希尔排序 $O(n^2)$ $O(1)$ 不稳定 顺序存储

冒泡排序 $O(n^2)$ $O(1)$ 稳定 顺序存储和链式存储

快速排序 $O(n\log_2 n)$ $O(\log_2 n)$ 不稳定 顺序存储（链式存储）

简单选择排序 $O(n^2)$ $O(1)$ 不稳定 顺序存储和链式存储

堆排序 $O(n\log_2 n)$ $O(1)$ 不稳定 顺序存储（链式存储）

归并排序 $O(n\log_2 n)$ $O(n)$ 稳定 顺序存储和链式存储

本节内容

排序

基数排序

基数排序

基数排序

借助“分配”和“收集”两种操作对单逻辑关键字进行排序，分为最高位优先 (MSD) 和最低位优先（LSD）。

不基于比较

以r为基数的最低位优先基数排序的过程：

假设线性表由结点序列a_0，a_1,...，a_{n-1}构成，

每个结点a_j的关键字由d元组(k_j^{d-1}，k_j^{d-2},...，k_j^1，k_j^0)组成

$0 \le k_j^i \le r-1 (0 \le j < n, 0 \le i \le d-1)$

324　768　270　121　962　666　857　503　768

n = 9,　d = 3,　r = 10

基数排序

分配和收集：

在排序时使用r个队列Q_0，Q_1，…，Q_{r-1}

分配：开始时，把Q_0，Q_1，…，Q_{r-1}各个队列置空，然后依次考察每一个结点的关键字，若a_j的关键字中$k_j^i = k$，就把a_j放入队列Q_k当中

收集：把Q_0，Q_1，…，Q_{r-1}各个队列中的结点依次收尾相接，得到一个新的结点序列，组成线性表

d次分配收集后，序列会排成有序的序列

基数排序

	324	768	270	121	962	666	857	503	768
一次分配收集	270	121	962	503	324	666	857	768	768

稳定！

Q_0	270	Q_5	
Q_1	121	Q_6	666
Q_2	962	Q_7	857
Q_3	503	Q_8	768 768
Q_4	324	Q_9	

基数排序

	324	768	270	121	962	666	857	503	768
一次分配收集	270	121	962	503	324	666	857	768	768
两次分配收集	503	121	324	857	962	666	768	768	270

稳定!

Q_0	503	Q_5	857
Q_1		Q_6	962 666 768 768
Q_2	121 324	Q_7	270
Q_3		Q_8	
Q_4		Q_9	

基数排序

	324	768	270	121	962	666	857	503	768
一次分配收集	270	121	962	503	324	666	857	768	768
两次分配收集	503	121	324	857	962	666	768	768	270
三次分配收集	121	270	324	503	666	768	768	857	962

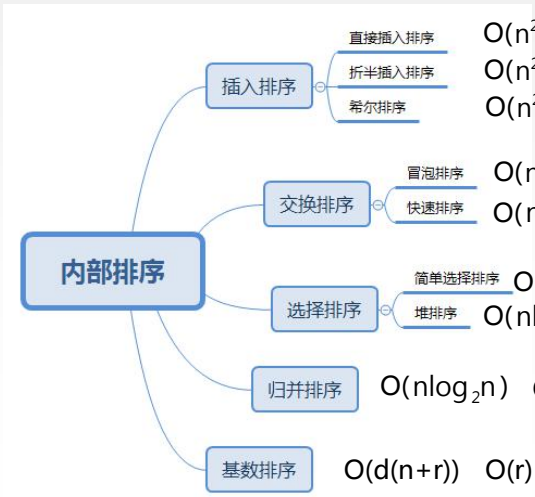
稳定!

Q_0		Q_5	503
Q_1	121	Q_6	666
Q_2	270	Q_7	768 768
Q_3	324	Q_8	857
Q_4		Q_9	962

O(d(n+r))

O(r)

排序



直接插入排序 $O(n^2)$ $O(1)$ 稳定 顺序存储和链式存储

折半插入排序 $O(n^2)$ $O(1)$ 稳定 顺序存储

希尔排序 $O(n^2)$ $O(1)$ 不稳定 顺序存储

冒泡排序 $O(n^2)$ $O(1)$ 稳定 顺序存储和链式存储

快速排序 $O(n\log_2 n)$ $O(\log_2 n)$ 不稳定 顺序存储（链式存储）

简单选择排序 $O(n^2)$ $O(1)$ 不稳定 顺序存储和链式存储

堆排序 $O(n\log_2 n)$ $O(1)$ 不稳定 顺序存储（链式存储）

归并排序 $O(n\log_2 n)$ $O(n)$ 稳定 顺序存储和链式存储

基数排序 $O(d(n+r))$ $O(r)$ 稳定 **不基于比较**

本节内容

排序

内部排序算法
比较和应用

王道考研/CSKAOYAN.COM

内部排序比较和应用

比较

时空复杂度　　稳定性　　一趟排序的特点

排序算法	时间复杂度			空间复杂度	稳定性
	最好情况	平均情况	最坏情况		
直接插入排序	O(n)	$O(n^2)$	$O(n^2)$	O(1)	稳定
冒泡排序	O(n)	$O(n^2)$	$O(n^2)$	O(1)	稳定
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	不稳定
希尔排序				O(1)	不稳定
快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	不稳定
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	O(1)	不稳定
2路归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	O(n)	稳定
基数排序	O(d(n+r))	O(d(n+r))	O(d(n+r))	O(r)	稳定

内部排序比较和应用

应用

考虑因素：
元素数目、元素大小、关键字结构及分布、稳定性、存储结构、辅助空间等

1、若n较小时（n≤50），可采用直接插入排序或简单选择排序
　若n较大时，则采用 快排、堆排或归并排序

2、若n很大，记录关键字位数较少且可分解，采用基数排序

3、当文件的n个关键字随机分布是，任何借助于“比较”的排序，至少需要 $O(n\log_2 n)$ 的时间

4、若初始基本有序，则采用直接插入或冒泡排序

5、当记录元素比较大，应避免大量移动的排序算法，尽量采用链式存储

本节内容

排序

外部排序
外部排序方法

外部排序

外部排序通常采用归并排序方法。

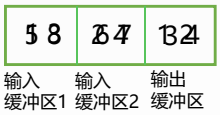
首先，根据缓冲区的大小将外存上含有n个记录的文件分成若干长度为h的子文件，依次读入内存并利用有限的内部排序算法对它们进行排序，并将排序后得到的有序子文件重新写回外存，通常称这些有序子文件为**归并段或顺串**

外部排序

外部排序通常采用归并排序方法。

首先，根据缓冲区的大小将外存上含有n个记录的文件分成若干长度为h的子文件，依次读入内存并利用有限的内部排序算法对它们进行排序，并将排序后得到的有序子文件重新写回外村，通常称这些有序子文件为**归并段或顺串**

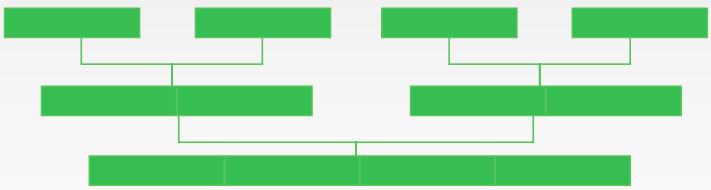
然后，对这些归并段进行逐趟归并，使归并段逐渐由小到大直至得到整个有序文件



1 3 5 8　　2 4 6 7

1 2 3 4 5 6 7 8

外部排序



外部排序的总时间=内部排序所需时间+外存信息读写时间+内部归并所需时间

$$t_{ES} = r * t_{IS} + d * t_{IO} + S(n-1)t_{mg}$$

20000个记录，初始归并段5000个记录

$$t_{ES} = 4 * t_{IS} + 3 * (4+4) * t_{IO} + 2 * 20000t_{mg}$$

外部排序

外部排序的总时间=内部排序所需时间+外存信息读写时间+内部归并所需时间

$$t_{ES} = r * t_{IS} + d * t_{IO} + S(n-1)t_{mg}$$

20000个记录，初始归并段5000个记录

$$2*(4+4)*t_{IO}$$

归并趟数= $\lceil \log_m r \rceil$

本节内容

排序

外部排序
失败树

失败树

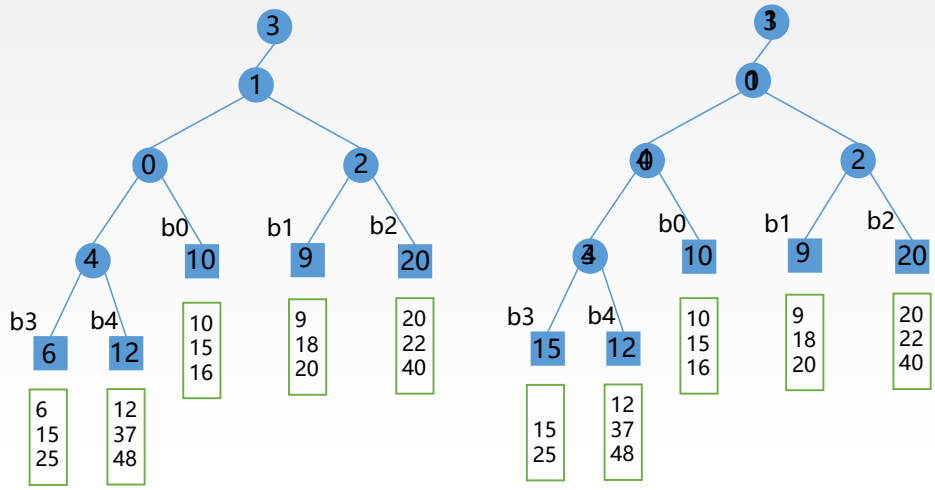
S趟归并总共需要比较的次数

$S(n-1)(m-1)=\lceil \log_m r \rceil (n-1)(m-1)$

失败树 树形选择排序的一种变体，可视为一棵完全二叉树

每个叶结点存放各归并段在归并过程中当前参加比较的记录，内部结点用来记忆左右子树中的“失败者”，胜利者向上继续进行比较，直到根结点。

失败树



失败树

S趟归并总共需要比较的次数

$S(n-1)(m-1)=\lceil \log_m r \rceil (n-1)(m-1)$

$S(n-1)(m-1)=\lceil \log_m r \rceil (n-1)\lceil \log_2 m \rceil=\lceil \log_2 r \rceil (n-1)$

失败树 树形选择排序的一种变体，可视为一棵完全二叉树

每个叶结点存放各归并段在归并过程中当前参加比较的记录，背部结点用来记忆左右子树中的“失败者”，胜利者向上继续进行比较，直到根结点。

本节内容

排序

外部排序
置换-选择排序

王道考研/CSKAOYAN.COM

置换-选择排序

置换-选择排序

设初始待排序文件为FI，初始归并段文件为FO，内存工作区为WA，内存工作区可容纳w个记录。

算法思想：

1）从待排序文件FI输入w个记录到工作区WA；

2）从内存工作区WA中选出其中关键字取最小值的记录，记为MINIMAX；

3）将MINIMAX记录输出到FO中；

4）若FI未读完，则从FI输入下一个记录到WA中；

5）从WA中所有关键字比MINIMAX记录的关键字大的记录中选出最小的关键字记录，作为新的MINIMAX；

6）重复3）~5），直到WA中选不出新的MINIMAX记录位置，由此得到一个初始归并段，输出一个归并段的结束标志到FO中；

7）重复2）~6），直到WA为空。由此得到全部初始归并段。

置换-选择排序

置换-选择排序

设待排序文件FI={17，21，05，44，10，12，56，32，29}，内存工作区的容量w为3

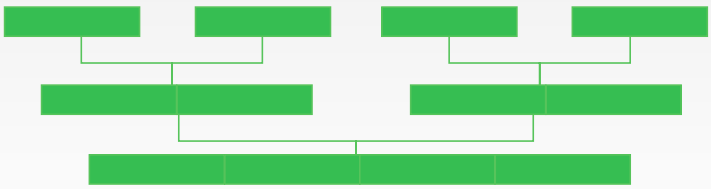
输出文件FO	工作区	输入文件FI
		17 21 05 44 10 12 56 32 29
	17 21 05	44 10 12 56 32 29
05	17 21 44	10 12 56 32 29
05 17	10 21 44	12 56 32 29
05 17 21	10 12 44	56 32 29
05 17 21 44	10 12 56	32 29
05 17 21 44 56	10 12 32	29
05 17 21 44 56 #	10 12 32	29
10	29 12 32	
10 12	29 32	
10 12 29	32	
10 12 29 32		
10 12 29 32 #		

本节内容

排序

外部排序
最佳归并树

最佳归并树

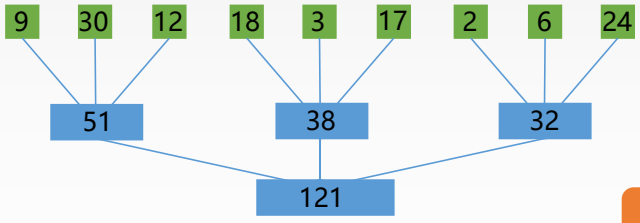


m路归并排序可用一棵m叉树描述。

归并树 用来描述m归并，并只有度为0和度为m的结点的严格m叉树

最佳归并树

设由置换-选择排序得到9个初始归并段，
其记录的长度依次为9， 30， 12， 18， 3， 17， 2， 6， 24



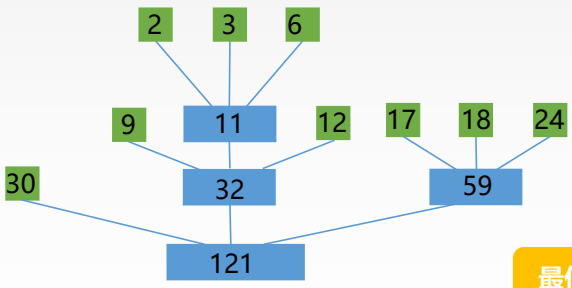
哈夫曼树

总IO次数为2*WPL=484

★带权路径长度之和为归并过程中的总读记录数

最佳归并树

设由置换-选择排序得到9个初始归并段，
其记录的长度依次为9， 30， 12， 18， 3， 17， 2， 6， 24



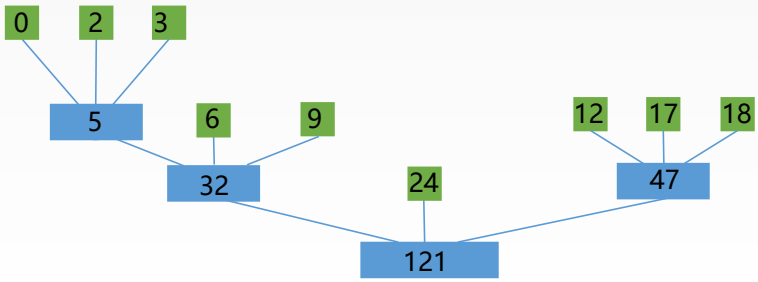
最佳归并树

总IO次数为2*WPL=446

最佳归并树

设由置换-选择排序得到8个初始归并段，
其记录的长度依次为9， 12， 18， 3， 17， 2， 6， 24

★ 当叶子节点数不够时，增加权值为0的结点用来构造哈夫曼树



最佳归并树

? 补充的虚段个数

设度为0的结点有n_0个，度为m的结点有n_m个，

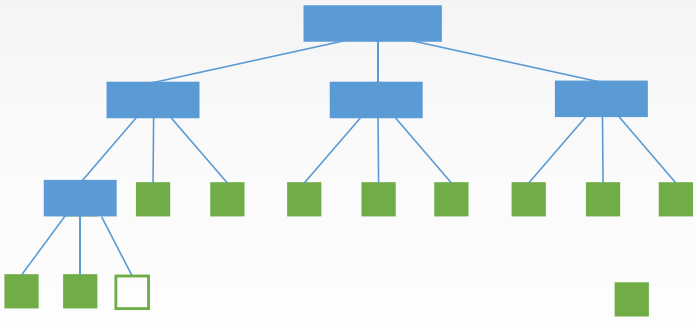
则对严格m叉树有$n_0=(m-1)\ n_m+1$，即得$n_m=\frac{n_0-1}{m-1}$

·若$(n_0-1)\%(m-1)==0$，则说明对于这个n_0个叶结点（初始归并段）可以构造m叉树归并树

·若$(n_0-1)\%(m-1)=u\neq 0$，则说明对于这个n_0个叶结点（初始归并段）其中有u个叶结点是多余的

最佳归并树

设现在有10个叶结点



★ 多出u个结点，需要补充m-u-1个结点

数据结构