第6章 查找与排序

- 主要内容：
- 查找
  - 查找的基本概念
  - 静态查找表
  - 动态查找表
  - 哈希表
- 内部排序
  - 排序的基本概念
  - 插入排序
  - 交换排序
  - 选择排序
  - 归并排序

§6.1查找的概述

- 查找是一种常见的操作，其操作的对象叫查找表
- 查找表是一种松散的数据结构：集合
- 集合的特点是数据元素间只存在同属于一个集合的关系，这给查找带来不便，因此常常人为加上一些关系，例如线性关系，树的关系等。

查找表是一种非常灵活的数据结构

§6.1.1 基本概念

- 查找表：由同一类型的数据元素(记录)构成的集合
  - 该数据元素可以由若干个数据项组成
- 关键字：数据元素中某个数据项或组合项的值，用它可以标识一个数据元素。
  - 主关键字：能唯一地标识一个数据元素的值为主关键字
  - 次关键字：不能唯一确定一个记录，一般用以识别若干个记录的关键字
- 查找：也叫检索，是根据给定的某个值，在查找表中确定一个关键字等于给定值的记录或数据元素的过程
  - 查找成功：查到该记录，给出其在列表中的位置或其它信息。
  - 查找失败：查不到记录，输出相应信息或把元素插入表中

查找表的分类

- 查找表的操作有四个基本内容：
  - 查询某个“特定的”数据元素是否在查找表中
  - 检索某个“特定的”数据元素的各种属性
  - 在查找表中插入一个数据元素
  - 从查找表中删除某个数据元素
- 静态查找表：不涉及插入和删除操作的查找表，即在查找过程中其结构始终不发生变化的查找表。
  - 也适用于经过一段时间的查找之后，集中地进行插入和删除等修改操作的情况
- 动态查找表：能进行全部四种操作的查找表，即其结构在查找过程中要发生变化的查找表。

查找表的结构

- 静态查找表
  - 无序顺序表
    - 无监视哨
    - 有监视哨
  - 有序顺序表
    - 顺序查找
      - 无监视哨
      - 有监视哨
    - 折半查找
    - *插值查找
    - *斐波那契查找
    - *静态树表查找
  - 索引表
- 动态查找表
  - 二叉排序树
  - *平衡二叉树
  - *B-/B⁺树
  - *键树

查找算法的性能

- 算法效率评价：时间复杂度+空间复杂度
- 查找：性能通过对关键字的比较次数来度量
  - 最大查找长度：对关键字的最多比较次数。
  - 平均查找长度：为确定记录在表中的位置，需要和关键字进行比较的次数的期望值。
- 对含有n个记录的表，其平均查找长度为：

$$ASL=\sum_{i=1}^{n} p_i c_i$$

p_i为查找第i个元素的概率，$\sum_{i=1}^{n} p_i = 1$

c_i为找到表中第i个元素所需比较次数

注意：c_i取决于算法；p_i与算法无关，取决于具体应用。如果p_i是已知的，则平均查找长度只是问题规模的函数。

典型的关键字类型说明

- 类型说明

```
typedef  float  KeyType;
typedef  int  KeyType;
typedef  char  *KeyType;
```

- 数据元素类型定义

```
typedef struct {
   KeyType key;
   …
}ElemType;
```

- 两个关键字比较约定

```
#define EQ(a,b)  ((a)==(b))
#define LT(a,b)  ((a)<(b))
#define LQ(a,b)  ((a)<=(b))
#define EQ(a,b)  (!strcmp((a),(b))
#define LT(a,b)  (strcmp((a),(b))<0)
#define LQ(a,b)  (strcmp((a),(b))>0)
```


§6.1.2 静态查找表

- 抽象数据类型定义

```
ADT Staticsearchtable
{  数据对象D：……；
   数据关系R：……；
   基本操作P：
       create(&ST,n);    destroy(&ST);
       search(ST,key);   traverse(ST);
}ADT StaticSearchTable
```


1. 顺序表的查找

- 顺序查找又称线性查找，是最基本的查找方法之一
- 顺序表的数据结构

```
typedef struct{
   ElemType  *elem; //0为空单元
   int     Length; //表长度
}SSTable;
typedef struct
{  Keytype  key;
   其它域定义
}elemtype
```

- 查找基本思想：从表的一端，一般是第n个记录开始，逐个进行记录的关键字和给定值的比较。若相等，则查找成功；若整个表扫描结束后仍未找到与给定值相等的关键字，则查找失败。

改进的顺序查找算法

- 算法思路：把给定值K作为第0个元素(该元素不是查找表的元素)的关键字。这样不必判断是否已查找完整个表。第0个元素起到了监视哨的作用。这样可节省大量测试时间。

```
int search(SStable ST, Keytype key)//顺序表, 也可以用链表
{   ST.elem[0].key=key; //0位置本身为空单元
    for(i=ST.length; !EQ(ST.elem[i].key,key); --i);
    return i;
} //若存在返回在静态表中的位置i ，否则i=0
```

优点：对表的结构无任何要求(勿需排序)，算法简单且适应面广
缺点：效率低(当表很长时，线性查找的效率很低)

性能分析(平均查找长度ASL)

- 查找次数c_i取决于所查记录在表中的位置。
  - 查找最后一个记录时，比较1次；查找第1个记录，比较n次
  - 在查找成功情况下，$c_i=n-i+1$
- 设表中每个元素的查找概率相等 $p_i=\frac{1}{n}$

$$ASL=\sum_{i=1}^{n}p_ic_i=\frac{1}{n}\sum_{i=1}^{n}(n-i+1)=\frac{1}{n}\sum_{i=1}^{n}i=\frac{1}{n}\cdot\frac{n(n+1)}{2}=\frac{n+1}{2}$$

- 设查找成功与不成功的可能性相同，则查找成功时的概率修改为： $p_i=\frac{1}{2n}$
- 查找不成功查找长度 $ASL=\frac{1}{2}(n+1)$
- 总平均查找长度 $ASL_{SS}=\frac{1}{2n}\sum_{i=1}^{n}(n-i+1)+\frac{n+1}{2}=\frac{3(n+1)}{4}$

2. 折半查找(二分查找)

- 有序表：对于以顺序方式存储的记录，若元素按关键字非递减(或非递增)顺序排序则称为有序数组或有序表
- 查找思想：每次将待查记录所在范围缩小一半，直到找到或找不到该记录为止。
  - 设表长为n，low、high和mid分别指向待查元素所在区间的上界、下界和中点，k为给定值
  - 初始时，令low=1，high=n, $\mathrm{mid}=\lfloor(\mathrm{low}+\mathrm{high})/2\rfloor$，让k与mid指向的记录比较
    - 若k==elem[mid].key，查找成功
    - 若k<elem[mid].key，则在左半区比较：high=mid-1
    - 若k>elem[mid].key，则在右半区比较：low=mid+1
  - 重复上述操作，如果直至low>high，查找失败

算法程序

- 查找关键字为21的数据元素

①　②　③　④　⑤　⑥　⑦　⑧　⑨　⑩　⑪
05　13　19　21　37　56　64　75　80　88　92

↑low　　　　　↑mid　　　　　↑high
↑low　↑mid　↑high
↑low　↑high
↑mid

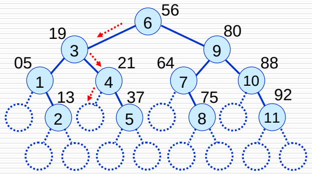
```
int search_bin(SStable ST, KeyType key)
{   low=1;  high=ST.length;
    while (low<=high)
    {   mid=(low+high)/2;
        if (EQ(key, ST.elem[mid].key))  return  mid;
        else if (LT(key, ST.elem[mid].key))  high=mid-1;
            else  low=mid+1;   }
    return 0;   }
```


性能分析

①　②　③　④　⑤　⑥　⑦　⑧　⑨　⑩　⑪
05　13　19　21　37　56　64　75　80　88　92

↑low　　　　　↑mid　　　　　↑high
↑low　↑mid　↑high
↑low　↑high
↑mid

$$ASL=\sum_{i=1}^{n}p_ic_i=\frac{1}{n}\sum_{i=1}^{n}c_i$$
$$=\frac{1+2\times2+3\times4+4\times4}{11}=3$$



成功的查找恰好是走了一条从根结点到被查找结点的路径
失败的查找则是经历了一条从根结点到某个外部结点的路径

二叉判定树

- 判定树：用来描述折半查找过程的二叉树。树中的每个结点对应有序表中的一个记录，结点的值为该记录在表中的位置。也称折半查找判定树或二叉判定树
  - 判定树的形态只与表结点的个数有关，与输入实例的关键字的取值无关。
- 特点：
  - 叶子结点所在层次之差最多为1
  - 具有n个结点的判定树的深度为$\lfloor\log_2 n\rfloor+1$。特别地，当n=2ᵈ-1 时，二叉树判断树一定是深度为d的满二叉树。
  - 和给定值进行比较的次数恰为该结点在判定树中的层次
  - 不成功查找的比较次数也不超过判定树的深度

性能分析(.续)

- 设表长n=2^h-1，h=$\log_2$(n+1)，即判定树是深度为h的满二叉树，则层次为h的结点有2^{h-1}个。设每个记录的查找概率相等：

$$ASL=\sum_{i=1}^{n}p_ic_i=\frac{1}{n}\sum_{i=1}^{n}c_i=\frac{1}{n}\sum_{j=1}^{h}j\cdot2^{j-1}=\frac{n+1}{n}\log_2(n+1)-1\approx\log_2(n+1)-1$$

- 结论：
  - 当元素有序时，折半查找比顺序查找效率高得多。但当元素无序时，需要先进行费时的排序，即使采用高效率的排序方法也要花费O(nlogn)的时间。
  - 折半查找只适用顺序存储结构。若要经常插入和删除元素，此时宜采用二叉排序树结构。当查找较少时，也可以用链表存储并进行顺序查找。

3. 索引顺序表的查找—分块查找

- 分块查找又称索引顺序查找
  - 表中元素均匀地分成块(子表)，块间按大小排序，块内不排序
  - 建立一个关键字表(也称索引表)，按大小顺序存放每块中最大或最小关键字值，其指针项给出每块起始地址。
- 分块有序：是指第二个子表中所有记录的关键字均大于第一个子表中的最大关键字。

索引表

最大关键字	22	48	86
起始地址	1	7	13

22	12	13	8	9	20	33	42	44	38	24	48	60	58	74	49	86	53

分块查找过程

- 算法思想：
  - 按折半查找获得所查关键字所对应的记录块
  - 按线性查找在块中找到key对应的记录
- 查找效率：设有n个元素，每块s个元素，
  - 平均查找长度=折半查找平均次数+线性查找平均次数

$$ASL_{bs}=\left\lfloor\log_2\left(\lceil n/s\rceil+1\right)\right\rfloor-1+\frac{s+1}{2}\approx\log_2\left(\frac{n}{s}+1\right)+\frac{s}{2}$$

  - 平均查找长度=顺序查找平均次数+线性查找平均次数

$$ASL_{bs}=\frac{\lceil n/s\rceil+1}{2}+\frac{s+1}{2}=\frac{1}{2}(\frac{n}{s}+s+2)$$

- 结论：其效率介于折半查找和线性查找之间。
- 容易证明，当s取$\sqrt{n}$时，分块查找可取最小值$\sqrt{n}+1$

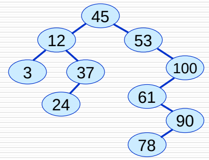
§6.1.3 动态查找表

- 特点：表结构是在查找过程中动态生成的，即对于给定值key，若表中存在其关键字等于key的记录，则在查找成功后返回，否则插入关键字等于key的记录。
- 抽象数据类型定义

```
ADT Dynamicsearchtable
{  数据对象D：……;
   数据关系R：……;
   基本操作P：
     initDtable(&DT);          destroyDtable(&DT);
     searchDtable(DT,key);   insertDtable(&DT,e);
     deleteDtable(&DT,key);  traverse(DT);
}ADT DynamicSearchTable
```


1. 二叉排序树

- 二叉排序树或者是一棵空树，或者是具有下列性质的二叉树
  - 若它的左子树不空，则左子树上所有结点的值均小于根结点的值
  - 若它的右子树不空，则右子树上所有结点的值均大于根结点的值
  - 它的左、右子树也分别为二叉排序树



中序遍历二叉排序树得到的序列是一个递增有序序列

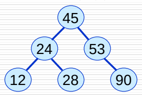
- 若允许相同关键字，可修改"大于"为"大于等于"，这可以保证其稳定性

数据结构类型定义

```
typedef  int  KeyType;
typedef struct  node{
    struct {
        KeyType key;
        InfoType otherinfo;
    }data;
    struct node  *lchild,*rchild;
}BSTNode;
typedef BSTNode *BSTree;
```


2. 二叉排序树的基本操作

- (1) 查找
- 当二叉排序树不空时，首先将给定值k与根结点的关键字比较，若相等，则查找成功；否则依k的大小在左子树或右子树上查找。查找终止条件：下层结点为空。
- 例如：在二叉排序树中查找90



- 算法程序

```
BiTree  SearchBST(BiTree T, keytype key)
{  if ((!T) || EQ(key,T->data.key)) return (T);
   else  if LT(key,T->data.key)
              return SearchBST(T->lchild,key);
        else  return SearchBST(T->rchild,key); }
```


(2) 插入

- 基本思想：只有当查找失败时，才将新结点插入到树中"适当位置"，使之仍然构成一棵二叉排序树。因此进行插入时首先比较原来二叉排序树上的结点，如果有该数据元素，则返回，否则判断新结点应该添加在哪一个叶子结点或度为1的节点上。
- 算法描述：
  - 在二叉树中查找所要插入的结点的关键字，若查找到，则不需插入
  - 若查找不成功
    - 动态生成值为k的新结点s
    - 若树为空，则直接将s作为根结点返回
    - 若k小于根结点的值，则在根的左子树中插入结点s
    - 若k大于根结点的值，则在根的右子树中插入结点s

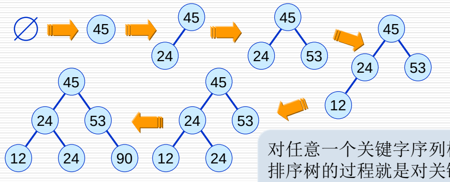
算法程序

```
status  SearchBST(BiTree T, keytype key, BiTree f, BiTree &p)
// f为T的父结点，p返回查找到的结点或未查找到的叶结点
{  if (!T) {  p=f;  return False;  }
   else if  EQ(key,T->data.key) {  p=T;   return True;  }
         else if LT(key,T->data.key)
                  return  SearchBST(T->lchild,key,T,p);
              else return SearchBST(T->rchild,key,T,p);   }
```

```
status  InsertBST(BiTree T, Elemtype e)
{  if(!SearchBST(T, e.key, Null, p))  //查找不成功
   {  new(s);    s->data=e;      s->lchild=s->rchild=Null;
      if (!p)  T=s;   //原树为空
      else if LT(e.key,p->data.key)  p->lchild=s;
           else p->rchild=s;
      return True;   }
   return False;  }
```

(3) 二叉排序树的生成

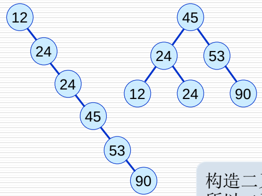
- 二叉排序树可由空树开始插入一个个数据元素来生成
  - 若二叉排序树为空树，则新结点为根结点
  - 若二叉排序树非空，则插入新结点形成某叶子结点
- 例1：45, 24, 53, 12, 24, 90



对任意一个关键字序列构造其二叉排序树的过程就是对关键字进行排序的过程，也称这种排序为树排序

二叉排序树的生成(.续)

- 例2：同样的序列45, 24, 53, 12, 24, 90，现输入顺序为12, 24, 24, 45, 53, 90，试构造二叉排序树



注意：含有n个结点的二叉排序树，其形态不唯一。随着结点的插入先后次序不同，所构成的二叉排序树也不同。

哪一个形态更好？

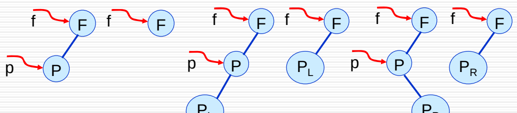
构造二叉排序树的目的是为了查找，所以二叉排序树又常称为二叉查找树

查找算法效率分析

- 二叉排序树的基本操作是查找。查找时，若查找成功，则是从根结点出发走了一条从根到某个叶子的路径。因此，类似折半查找，和关键字比较次数不超过该二叉树的深度h。
- 由于二叉排序树形态不唯一，其平均查找长度和二叉树在输入时所形成的形态有关。
  - 最好情况：类似二叉判定树，平均查找长度为$O(\log_2 n)$
  - 最坏情况：蜕变为一棵单支树(输入关键字值有序)，平均查找长度为$(n+1)/2$。
  - 一般情况下，则有n!棵二叉排序树(其中有的形态相同)，平均值也是$O(\log_2 n)$。
- 二叉排序树的生成操作类似，复杂度为$O(n\log_2 n)$

(4) 二叉排序树的删除

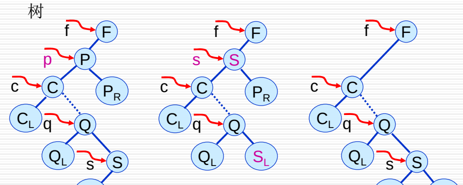
- 关键：确保删除后二叉排序树的特性不变。
- 若待删除的结点为*p，其双亲为*f，不失一般性，可设*p为*f的左子树。分三种情况：
  - 若*p没有左右孩子，则直接删除*p，修改*f指针域
  - 若*p结点只有左子树P_L或只有右子树P_R，令其左子树或右子树直接成为其双亲结点*f的左子树，并删除*p



(1) (2)

二叉排序树的删除(.续)

- 若*p结点的左子树和右子树均不空，有两种方法
  - 寻找*p的左子树中最右边结点*s，即左子树中关键字最大的结点S，令*s替代*p，然后删除*s，即用结点S取代P。若*s有左子树，则取代原*s位置
  - 令*p的左子树为其双亲*f的左子树，*p的右子树为*s的右子树



(3)

二叉判定树和二叉排序树总结

- 二叉判定树是用一个数组向量作为存储结构，它是唯一的，涉及插入、删除要维护表的有序性，其代价是O(n)。然而二叉排序树的存储结构是树，其插入和删除操作效率更高，因此：
  - 有序表是静态查找表时，采用向量存储结构，用二分查找。
  - 有序表是动态查找表时，则选择二叉排序树作为其存储结构。
- 然而，二叉排序树的算法效率与其形态有关，因此为了提高查询速度，必须使二叉排序树尽可能保持某种“平衡”，即构造所谓的平衡二叉树。

§6.1.4 哈希表

- 查找：待查值k ⟹ 确定k在存储结构中的位置
- 查找的方法
  - 关键字比较法：数据元素存储位置与关键字之间不存在确定的关系，查找时需要进行一系列对关键字的查找比较
    - “查找算法”是建立在比较的基础上的，查找效率由比较一次缩小的查找范围决定。

  能否不用比较，通过关键码直接确定存储位置？

  - 直接获得地址法：根据关键字直接得到对应的数据元素位置，要求关键字与数据元素存在一一对应关系。

1. 基本概念

- 哈希查找因使用哈希(Hash)函数而得名
- 哈希函数又叫散列函数，是一种能把关键字映射成记录存储地址的函数，也可以看成是一种映像。通过哈希函数确定元素地址的过程称为散列或哈希造表，所得存储位置称哈希地址或散列地址。
- $addr(a_i)=H(key_i)$

  关键字集合 $\xrightarrow{hash}$ 存储地址集合

  - a_i是表中的一个元素
  - $addr(a_i)$是a_i的存储地址，或表的向量结构的元素下标
  - key_i是a_i的关键字

为什么也称为哈希为散列？

哈希法不是按某种顺序依次存储记录到向量结构中，而是把各记录散列到相应存储单元中去，所以也称为散列地址法

基本概念(.续)

- 哈希表：根据哈希函数建立的关键字集合到地址空间的记录表就叫作对应于该哈希函数的哈希表。
- 哈希查找：在查找时，根据哈希函数找给定值的像，若存在，则必定在该像的存储位置上直接查到。这样不经过比较，一次存取就能得到所查元素的查找方法就叫哈希表查找。

注意：哈希既是一种查找技术，也是一种存储技术

哈希是一种完整的存储结构吗？

哈希只是通过记录的关键字定位该记录，没有完整地表达记录之间的逻辑关系，所以，哈希表主要是面向查找的存储结构

哈希表的冲突现象

- 冲突：给定H，若$key_1 \neq key_2$，而$H(key_1)=H(key_2)$,则称key_1,key_2发生地址冲突
- 同义词：具有同一哈希地址的关键字称为该哈希函数的同义词
- 冲突原因：一般关键字空间比地址空间大得多，而真正使用的关键字集合却很少，因此不能按关键字空间为其分配地址空间。哈希函数实际上是一个压缩函数，一般冲突不可避免。

  关键字集合 $\xrightarrow{hash}$ 存储地址集合

- 哈希表的填满因子a

  a=表中填入的记录数/哈希表长度

因此哈希法由哈希函数构造技术和冲突解决技术两部分组成

举例:各省信息的组织和查找

- 省记录：| 省名 | 相关信息 |
- 省份数：34，哈希表长度可定为34个记录
- 哈希函数：省份名拼音首字母在字母表中序号

key	北京	天津	山西	上海	…
H(key)	2	20	19	19	…

- 处理冲突：可将冲突的记录在表中的位置移动到下一个空的位置。

	北京	…	山西	天津	上海	…	…
1	2		19	20	21		34

哈希表关键问题：
1. 如何选择哈希函数
2. 当哈希表中发生冲突时，如何解决冲突

§6.2 排序概述

- 1. 基本概念
- 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的过程叫排序。
  - 设序列$\{R_1, R_2, \ldots, R_n\}$，相应关键字序列为$\{K_1, K_2, \ldots, K_n\}$，所谓排序是指重新排列$\{R_1, R_2, \ldots, R_n\}$为$\{R_{p1}, R_{p2}, \ldots, R_{pn}\}$，使得满足：$\{K_{p1} \leq K_{p2} \ldots\ldots \leq K_{pn}\}$ 或 $\{K_{p1} \geq K_{p2} \ldots\ldots \geq K_{pn}\}$
- 排序的稳定性：假设$K_i=K_j$，($1 \leq i \leq n$; $1 \leq j \leq n$; $i \neq j$)且在排序前的序列中R_i领先于R_j，如在排序后R_i仍领先于R_j，则称所用的排序方法是稳定的，反之则称排序方法是不稳定的。

2. 排序的分类

- 待排序记录所在位置
  - 内部排序：待排序记录存放在内存中
  - 外部排序：排序过程中需对外存进行访问的排序

内部排序适用于记录个数不很多的小文件；外部排序则适用于记录个数太多，不能一次将其全部放入内存的大文件

- 排序依据策略
  - 插入排序：直接插入排序，折半插入排序，希尔排序
  - 交换排序：冒泡排序，快速排序
  - 选择排序：简单选择排序，堆排序
  - 归并排序：2-路归并排序
  - 基数排序

3. 排序的基本操作

- 排序的基本操作：
  - 比较操作：比较两个关键字的大小
  - 改变指向记录的指针(逻辑关系)或将一个记录从一个位置移动到另一个位置

是否需要移动，与待排序记录的存储方式有关

4. 数据的存储形式

- 待排记录一般有三种存储形式
  - 存放在一组地址连续的存储单元中，类似顺序表
    - 实现排序必须借助移动记录
  - 存放在静态链表中
    - 实现排序只需修改指针
  - 存放在一组地址连续的存储单元中，同时另设一个指示各个记录存储位置的地址向量
    - 实现排序时修改地址向量中的地址，排序结束时统一调整记录的存储位置

注意：本章中排序的记录以第一种方式存储

顺序存储结构定义

```
#define MAXSIZE  20    //顺序表的长度
typedef int KeyType;    //关键字类型为整数类型

typedef struct{
   KeyType  key;          //关键字项
   InfoType  otherinfo;   //其它数据项
}RedType;                 //记录类型

type struct {
   RedType r[MAXSIZE+1]; //r[0]空作为哨兵
   int  length;          //顺序表长度
}SqList;                 //顺序表类型
```


5. 算法复杂度分析

- 评价排序算法的标准：
  - 执行时间
  - 所需辅助空间
  - 稳定性

排序所需时间
- 简单排序：$T(n)=O(n^2)$
- 先进的排序：$T(n)=O(logn)$
- 基数排序：$T(n)=O(d.n)$

- 排序算法的时间开销
  - 主要是关键字的比较次数和记录的移动次数。
  - 有的排序算法其执行时间不仅依赖于问题的规模，还取决于输入实例中数据的状态。
- 排序算法的空间开销
  - 若所需辅助空间不依赖于问题的规模n，即辅助空间为O(1)，则称为**就地排序**。
  - 非就地排序所要求的辅助空间一般为O(n)

§6.2.1 插入排序

- 1. 插入排序的基本思想
- 基本思想：设R={R_1，R_2，……，R_n}为原始序列，R'={ }初始为空。插入排序就是依次取出R中的元素R_i，然后将R_i有序地插入到R'中。
- 例如：

	有序插入	
R={5,2,10,2}	→	R'={ }
R={2,10,2}	5	R'={5}
R={10,2}	2	R'={2,5}
R={2}	10	R'={2,5,10}
R={ }	2	R'={2,2,5,10}

2. 直接插入排序

- 排序过程：整个排序过程为n-1趟插入
  - 将序列中第1个记录看成是一个有序子序列
  - 从第2个记录开始，逐个进行插入，直至整个序列有序

R_1	R_2	……	R_n

R'={R_1}　R={ R_2，R_3，……，R_n }

- 若R[1..*i*-1]为有序区，寻找R[*i*]插入位置的方法：
  - 在R[0]处设置监视哨，即令：R[0]=R[*i*]
  - 从j=*i*-1的记录位置开始依次向前比较
  - 若R[*i*].key<R[j].key，R[j]后移，否则插入位置找到，将R[*i*]插入到第j+1个位置

R[0]既有哨兵的作用，也暂存了R[*i*]的值，使其不会因记录后移而丢失

举例

- 例：已知某下列的关键字为{49 38 65 97 76 13 27 49}，试采用直接插入排序方法进行排序

i=1:	(38)	(49)	38	65	97	76	13	27	49
i=2:	(65)	(38	49)	65	97	76	13	27	49
i=3:	(97)	(38	49	65)	97	76	13	27	49
i=4:	(76)	(38	49	65	97)	76	13	27	49
i=5:	(13)	(38	49	65	76	97)	13	27	49
i=6:	(27)	(13	38	49	65	76	97)	27	49
i=7:	(49)	(13	27	38	49	65	76	97)	49
i=8:		(13	27	38	49	49	65	76	97)

算法程序

```
void Dinsertsort(Sqlist  &L)
{  for (i=2;i<=L.length;++i)   //对每一个Ri∈R
     //小于时，需将L.r[i]插入到有序表
     if LT(L.r[i].key, L.r[i-1].key)
     {  L.r[0]=L.r[i];      //复制为哨兵
        /*寻找插入位置j*/
        for (j=i-1; LT(L.r[0].key, L.r[j].key); --j)
            L.r[j+1]=L.r[j];  //将j……i-1的记录后移一格
        L.r[j+1]=L.r[0];   }   //将Ri插入到位置j+1
}
```


算法效率

特点：n较小时，排序较快；待排序列基本正序时，排序时间为O(n)，是稳定的排序方法

- 时间复杂度
  - 待排序记录按关键字从小到大排列(正序)

    比较次数: $\sum_{i=2}^{n} 1 = n-1$　　移动次数: 0
  - 待排序记录按关键字从大到小排列(逆序)

    比较次数: $\sum_{i=2}^{n} i = \frac{(n+2)(n-1)}{2}$　　移动次数: $\sum_{i=2}^{n} (i+1) = \frac{(n+4)(n-1)}{2}$
  - 待排序记录随机，取平均值

    比较次数: $\approx \frac{n^2}{4}$　　移动次数: $\approx \frac{n^2}{4}$
  - 总的时间复杂度:T(n)=O(n^2)
- 空间复杂度：S(n)=O(1)

3. 折半插入排序

- 排序过程：用折半查找方法确定插入位置。
- 举例：

i=1:　(38)　(49)　38　65　97　76　13　27　49

H　LMH　LM　H+1:插入位置

i=2:　(65)　(38　49)　65　97　76　13　27　49

LM　H　LMH　MH　L　……　H+1:插入位置

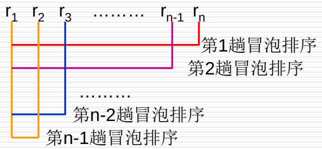
i=8:　(13　27　38　49　49　65　76　97)

§6.2.2 交换排序

- 1. 交换排序的基本思想
- 基本思想：在待排序列中选两个记录，将它们的关键码相比较，如果反序(即排列顺序与排序后的次序正好相反)，则交换它们的存储位置。
  - 特点：通过交换，将关键字值较大的记录向序列的后部移动，关键字较小的记录向前移动。
- 典型算法
  - 冒泡排序
  - 快速排序

2. 冒泡排序

- 基本思想
  - 在n个记录中，将相邻两个记录进行比较，若r[*i*].key>r[*i*+1].key (*i*=1,2,...,n)，则交换(第一趟冒泡排序)，结果关键字最大的记录被放在r_n。
  - 在n-1个记录中，若r[*i*].key>r[*i*+1].key (*i*=1,2,...,n-1)，则交换，结果关键字次大的记录被安置在r_{n-1}
  - 以此类推，直到“在一趟排序过程中没有进行过交换记录的操作”为止



举例

- 已知某序列的关键字为{ <u>49</u> 38 65 97 76 13 27 49 }，试采用冒泡排序法对其进行排序

初始关键字	<u>49</u>	38	65	97	76	13	27	49
第1趟排序后	38	<u>49</u>	65	76	13	27	49	97
第2趟排序后	38	<u>49</u>	65	13	27	49	76	97
第3趟排序后	38	<u>49</u>	13	27	49	65	76	97
第4趟排序后	38	13	27	<u>49</u>	49	65	76	97
第5趟排序后	13	27	38	<u>49</u>	49	65	76	97
第6趟排序后	13	27	38	<u>49</u>	49	65	76	97

结束标志：在一趟排序过程中没有进行过交换记录的操作

算法程序

```
void Bubble_sort(Sqlist  &L)
{  for (i= L.length; i>1; --i)
   {   exchange=false;    //设置开关
        for(j=1; j<i; ++j)
             if GT(L.r[j].key, L.r[j+1].key)//一趟冒泡
             {  L.r[j] ←→ L.r[j+1];   //移动3次
                 exchange=true;   }
        if (!exchange) exit;
   }
}
```


性能分析

- 时间复杂度
  - 最好情况(正序)：比较1趟n-1次，不移动
  - 最坏情况(逆序)：
    - 比较次数：$\sum_{i=1}^{n-1}(n-i)=\frac{1}{2}(n^2-n)$
    - 移动次数：$3\sum_{i=1}^{n}(n-i)=\frac{3}{2}(n^2-n)$
  - 总的时间复杂度：O(n²)
- 空间复杂度：S(n)=O(1)
- 冒泡排序是稳定的排序

冒泡排序的改进

- 改进不对称性

1 2 3 4 5	需扫描1趟	5 4 3 2 1	需扫描n-1趟
5 1 2 3 4	需扫描2趟	2 3 4 5 1	需扫描n-1趟

造成不对称的原因是什么？

每趟扫描仅能使最轻气泡“下沉”一个位置，因此使位于顶端的最轻气泡下沉到底部时，需做n-1趟扫描

- **双向冒泡排序**：在排序过程中交替改变扫描方向

2 3 4 5 1

2 3 4 1 5

1 2 3 4 5

改进方案2：记录的比较和移动是在相邻单元中进行，记录的每次交换只能移动一个单元。因此要减少扫描次数，可以增大比较和移动的距离，并缩小序列规模

2. 快速排序

- 基本思想：选择一个枢轴，通过一趟排序，将待排序记录分割成独立的两部分，其中一部分记录的关键字均比另一部分记录的关键字小，然后分别对这两部分记录进行排序，以达到整个序列有序。

快速排序方法的实质是将一组关键字进行分区交换排序

- 排序过程：
  - 设序列为r_1，r_2，…，r_n，定r_1为枢轴
  - 设low, high指针分别从两端与枢轴比较，把比r_1小的记录放前，比r_1大的记录放后，得到一次划分：

    r_{s1}，r_{s2}，…，r_{sk}，r_1，r_{t1}，r_{t2}，…，r_{tj}
  - 然后分别对两序列r_s和r_t再进行划分,直到划分后的序列剩一个元素为止，这是一个递归的过程

一趟分割过程

- 先从high所指记录向前搜索，找到第1个小于key的记录与枢轴交换。然后从low起向后搜索，找到第一个比key大的记录与low互换。重复上面两步，直到low=high为止(该位置即为枢轴的位置)

初始关键字	49	38	65	97	76	13	27	49
	L						↑H	↑H
1次交换之后	27	38	65	97	76	13	49	49
	↑L	↑L	↑L				H	
2次交换之后	27	38	49	97	76	13	65	49
			L			↑H	↑H	
3次交换之后	27	38	13	97	76	49	65	49
			↑L	↑L		H		
4次交换之后	27	38	13	49	76	97	65	49
				L↑H	↑H	↑H		
完成一趟排序	27	38	13	49	76	97	65	49

快速排序整个过程

初始关键字	49	38	65	97	76	13	27	49
第1趟排序后	27	38	13	49	76	97	65	49
第2趟排序后	13	27	38	49	49	65	76	97
	结束		结束					结束
第3趟排序后	13	27	38	49	49	65	76	97
						结束		
有序序列	13	27	38	49	49	65	76	97

快速排序是不稳定的排序

初始关键字	38	49	65	97	76	13	27	49
有序序列	13	27	38	49	49	65	76	97

算法效率分析

- 时间复杂度
  - 最好情况：每次总是选到中间值作枢轴，则形成二叉递归树：$T(n)=O(n\log_2 n)$
  - 最坏情况：每次总是选到最小或最大元素作枢轴，则退化为冒泡排序：$T(n)=O(n^2)$
  - 平均时间复杂度：$T_{avg}(n)= kn\ln n$ ；k为某个常数
  - 经验证明，在所有同数量级O(nlogn)的此类排序中，快速排序的常数因子k最小。因此，就平均时间而言，快速排序是速度最快的排序
  - 若枢轴记录取r[low]、r[(low+high)/2]和r[high]中关键字的中值的记录，并与r[low]互换，可以大大改善最坏情况的快速排序
- 空间复杂度：使用栈空间实现递归
  - 最坏情况：$S(n)=O(n)$；一般情况：$S(n)=O(\log_2 n)$

§6.2.3 选择排序

- 基本思想：每一趟在n-i+1(i=1, 2, ..., n-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. 简单选择排序
- 排序过程：
  - 设待排序列为：$a_1, a_2, \ldots, a_n$
  - 在$a_1, a_2, \ldots, a_n$中，找最小值记录与a_1交换
  - 在$a_2, a_3, \ldots, a_n$中，找最小值记录与a_2交换
  - 重复上述操作，共进行n-1趟排序后，排序结束
- 简单选择排序是不稳定的排序

5	5	1	2	3	➡	1	2	3	5	5

算法程序和效率分析

```
void  Selectsort(Sqlist &L)
{  for(i=1; i<L.length; ++i)
   {   for (j=i+1, k=i; j<=L.length; ++j)
           if (L.r[i].key>L.r[j].key)  k=j;
        if(k!=i) L.r[i] ↔L.r[k];   } }
```

- 时间复杂度
  - 移动次数：最好(正序)：0；最坏(逆序)：3(n-1)
  - 比较次数：$\sum_{i=1}^{n-1}(n-i)=\frac{1}{2}(n^2-n)$
  - 总的时间复杂度：$T(n)=O(n^2)$
- 空间复杂度：$S(n)=O(1)$

改进方案：若在每趟比较时，既找出键值最小的记录，也找出键值较小的记录，则可减少后面的选择中所用的比较次数，并缩小序列规模

2. 堆排序

- (1) 堆和堆排序
- 堆：n个元素的序列$(a_1, a_2, \ldots, a_n)$，其关键值序列为：$(k_1, k_2, \ldots, k_n)$，当且仅当其关键值满足下式称之为堆：

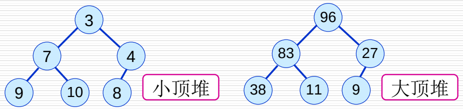
$$\begin{cases} k_i \le k_{2i} \\ k_i \le k_{2i+1} \end{cases} \quad 或 \quad \begin{cases} k_i \ge k_{2i} \\ k_i \ge k_{2i+1} \end{cases} \quad 其中 i=1, 2, \ldots, \lfloor n/2 \rfloor$$

- 满足前一表达式的堆称为小顶堆，即堆顶元素为所有元素中最小值；符合后一组不等式的堆称为大顶堆，即堆顶元素为所有元素中最大值。

考虑完全二叉树的顺序存储，若将其解释成堆如何？

举例

- {3, 7, 4, 9, 10, 8}，{96, 83, 27, 38, 11, 9}



注意：小顶堆中较小结点靠近根结点，但不绝对。大顶堆中较大结点靠近根结点，但也不绝对

- 堆是具有如下性质的完全二叉树：若树中每个结点的值都小于等于(或大于等于)其左、右孩子结点的值，则从根结点开始按结点编号排列所得的结点序列就是一个堆。

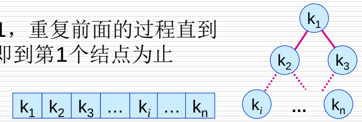
堆排序

- 堆排序：将无序序列建成一个堆，得到关键字最小(或最大)的记录；输出堆顶的最小(大)值后，使剩余的n-1个元素重又建成一个堆，则可得到n个元素的次小值；重复执行，得到一个有序序列，这个过程就叫堆排序。
- 排序过程(小顶堆)
  - 若a_1，a_2，……，a_n为堆，则a_1最小，交换a_1，a_n，输出a_n
  - 将剩余a_1，a_2，……，a_{n-1}调整为堆，当作新的序列
  - 重复这个过程直到序列只剩一个元素

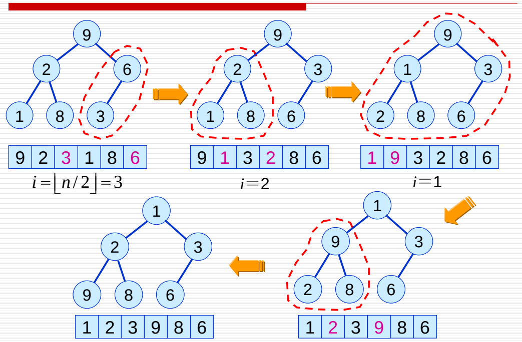
面临问题：
1. 如何由一个n个元素的无序序列建成一个堆？
2. 输出堆顶元素后，剩下的n-1个元素如何调整才能成为一个新的堆？

(2) 建立堆

- 算法描述(小顶堆)
  - 首先把给定序列看成是一棵完全二叉树
  - 从第$i=\lfloor n/2 \rfloor$结点(最后一个非终端结点)开始与其子树结点比较，若其直接子树结点中较小者小于i结点，则交换。若该直接子树结点交换后大于其孩子结点，继续交换，直到叶结点或不再交换
  - 令$i=i-1$，重复前面的过程直到$i=1$，即到第1个结点为止



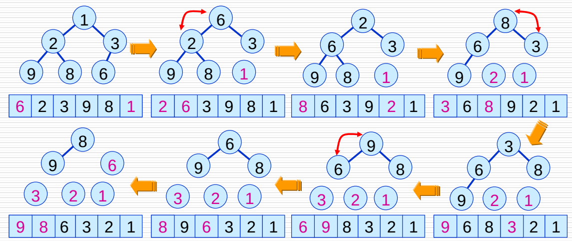
举例：9 2 6 1 8 3



(3) 调整堆—筛选

- 筛选：输出堆顶元素后，剩余元素重新调整为堆的过程。
- 调整步骤(小顶堆)
  - 将该完全二叉树中最后一个元素替代已输出的结点
  - 若新的完全二叉树的根结点小于左右子树的根结点，则直接输出。反之，则比较左右子树根结点的大小。若左子树的根结点小于右子树的根结点，则将左子树的根结点与该完全二叉树的根结点交换，否则将右子树与其交换。
  - 重复上述过程，调整左子树(或右子树)，直至叶子结点，则新的二叉树满足堆的条件。

举例



堆排序的过程就是一个不断从堆顶到叶子的筛选过程

当堆顶元素为所有元素中最小值时，输出的序列为逆序，为了符合正序，可采用大顶堆，即堆顶元素(根结点)为最大值。

算法程序

```
void  Heapadjust(Sqlist  &H,int s,int m)
// 除结点H.r[s]外,H.r[s+1..m]为大顶堆,将s结点调整到适当位置
{  temp=H.r[s];
   for (j=2*s; j<=m; j*=2) //沿k值较大的孩子筛选
   {  if ( j<m && LT(H.r[j].key,H.r[j+1].key)  ++j; //结点值较大的为j
      if (LT(temp.key,H.r[j].key)
      {  H.r[s]=H.r[j];  s=j;  }   //若s结点小于j,则j覆盖s,s下移一层
      else  exit;   }
   H.r[s]=temp;   }  //将s放入合适的位置

void  Heapsort(Sqlist  &H)
{  for (i=H.length/2; i>0; --i;)
      Heapadjust(H,i,H.length); //建立初始堆
      for (i=H.length; i>1; --i;)
      {   H.r[1] ↔ H.r[i];      //输出第i个小元素
          Heapadjust(H,1,i-1);  } } //剩余1..i-1个元素调整为堆
```


算法效率分析

- 时间复杂度
  - 对深度为k的堆，筛选算法中进行关键字的比较次数至多为2(k-1)。建堆时，n个元素所需的比较次数不超过4n。总的比较次数：$T(n) < 2n\lfloor \log_2 n \rfloor$
  - 在最坏情况下，其时间复杂度为O(n log n)
- 空间复杂度
  - 仅需一个记录大小的辅助存储空间。
- 堆排序是不稳定的排序。

堆排序的运行时间主要耗费在建立初始堆和筛选上，当记录较少时，不值得提倡。当n很大时效率高，是常用算法。

§6.2.4 归并排序

- 基本思想：将若干有序序列逐步归并，最终得到一个有序序列。
- 归并：将两个或两个以上的有序表合并成一个新的有序表的过程。

归并排序有多路归并排序、两路归并排序，可用于内排序，也可以用于外排序。

- 2-路归并排序
  - 设初始序列含有n个记录，看成n个有序的子序列，每个子序列长度为1
  - 两两合并，得到$\lceil n/2 \rceil$个长度为2或1的有序子序列
  - 对n/2个有序表两两合并，得到n/4个长度为4或3的有序表
  - 重复这个过程，直至得到一个长度为n的有序序列

举例

- 已知某序列为{49, 38, 65, 97, 76, 13, 27}，试采用归并排序，将其按从小到大顺序排列。

初始关键字	[49]	[38]	[65]	[97]	[76]	[13]	[27]	7个序列
1趟归并后	[38	49]	[65	97]	[13	76]	[27]	4个序列
2趟归并后	[38	49	65	97]	[13	27	76]	2个序列
3趟归并后	[13	27	38	49	65	76	97]	1个序列

归并排序可否就地进行？

对顺序表进行归并有可能破坏原来的表，因此需要一个与原来长度相同的表存放归并结果，如果采用链表存储可以就地排序

算法效率

- 时间复杂度
  - 对任意两个有序序列，长度分别为 m，n，可以在O(m+n)时间内完成有序归并，因此每趟归并的时间复杂度为O(n)，整个算法需$\log_2 n$趟。总的时间复杂度：O(n $\log_2$ n)
- 空间复杂度
  - S(n)=O(n)
- 归并排序是稳定的排序

归并排序算法虽简单，但占用辅助空间大，实用性差

各种内部排序方法的比较

排序方法	平均时间	最坏情况	辅助空间	稳定性
插入排序	$O(n^2)$	$O(n^2)$	O(1)	稳定
冒泡排序	$O(n^2)$	$O(n^2)$	O(1)	稳定
简单选择排序	$O(n^2)$	$O(n^2)$	O(1)	不稳定
希尔排序	$O(n^2)$	$O(n^2)$	O(1)	不稳定
快速排序	O(n log n)	$O(n^2)$	O(log n)	不稳定
堆排序	O(n log n)	O(n log n)	O(1)	不稳定
归并排序	O(n log n)	O(n log n)	O(n)	稳定
基数排序	O(d(n+rd))	O(d(n+rd))	O(rd)	稳定

总结

- 从平均时间看，快速排序最佳，所需时间最省，但快速排序在最坏情况下的时间性能不如堆排序和归并排序。而后两者相比较，在n较大时，归并排序所需时间较堆排序省
- 简单排序中，直接插入最简单，当序列基本有序或n较小，其最佳
- 基数排序的时间复杂度可写成O(d×n)，它最适用于n值很大而关键字较小的序列
- 从稳定性来比较，基数排序是稳定的，一般来说，时间复杂度为$O(n^2)$的简单排序也是稳定的，而快速排序、堆排序和希尔排序等时间性能较好的排序方法是不稳定的