累计发行超100万册

清华大学计算机系列教材

数据结构（C语言版）

严蔚敏　吴伟民　编著

清华大学出版社

前言

缘起

《数据结构》是一门计算机专业基础课，各类计算机考试都禁不住要考它，专升本考试自然也不例外。我给学生辅导这门课程已经有几个年头了，讲稿换了几次，逐渐丰富起来。加之看到学生们埋头记笔记时辛苦的样子，就产生了写一本小册子的想法。另外，还有一层意思就是对数次辅导进行总结，以便交流之用。

说明

首先，需要说明的是这本书在语言风格上不太讲究，常有些不严谨的表达，或调侃，或土得掉渣，难登大雅之堂，请勿在正规场合引用这些说法。这样做的目的，仅仅是为了更简练、更直接地描述思想，方便理解、记忆和使用。凡是这种情况，往往都用引号括起来，并加以脚注说明。

还有，本书需配合《数据结构》(严蔚敏)教材使用。由于篇幅有限，多数概念、术语没有详释。
另外，每章之后都配有习题，或多或少，难度不一，并没有局限于专升本的要求。对所有习题都提供了参考答案。

致谢

我要感谢所有给予我帮助的人。

张志老师的大力支持和帮助使得本书得以面世，他还提供了近年专升本试题。李永干老师的帮助使得本书顺利印刷。谭业武老师给了我很大支持，还提出了很多建议。

最后，我要感谢隆坤，她总是给我最大的支持，使那些本来只在我想象中的事情变成现实。

庄波
于滨州学院

2005 年 2 月 26 日

第0章 复习提示

一、教材内容

- 使用教材《数据结构》C 语言版 严蔚敏，清华大学出版社。
- 章节 去掉 第 5、8、11、12 章
  去掉 **部分
  去掉 1.3，2.4，4.4

二、复习提示

1. 经典算法

单链表：遍历、插入、删除
循环队列：队列空、队列满的条件
二叉树：递归遍历及应用
有序表的二分法查找
快速排序
简单选择排序

2. 绪论

掌握几个重要概念
　　数据结构、抽象数据类型、算法
　　时间复杂度的简单计算（C[1]）
掌握几种说法
　　数据元素是…，数据项是…
　　数据结构中关系的四种基本结构
　　数据结构的形式定义
算法的五个特征

[1] 记号 C，表示要求掌握计算方法，会计算。本节下同。

3. 线性表

线性表的概念和四个特征
顺序表和单链表的类型定义
在顺序表中查找、插入、删除，灵活运用
在单链表中查找、插入、删除，灵活运用
循环链表及双向链表的定义、插入、删除
算法：
　　单链表的算法，灵活运用、会编程（P[2]）

4. 栈和队列

栈和队列的概念、特点
入栈、出栈操作，灵活掌握
了解栈的实现：链栈和顺序栈（A[3]算法，P）
了解队列的实现，链队列和循环队列，注意链队列中的出队列操作
算法：
　　注意循环队列空和满的条件（A，P）
　　会运用栈和队列

5. 串

掌握相关概念
会运用串的基本操作（C），特别是 Concat()，Substring()，Index()和 Replace()
知道串的三种存储结构及其特点

6. 树和二叉树

树和二叉树的有关概念
二叉树的性质
熟练掌握遍历二叉树的递归算法，并灵活运用
知道线索二叉树，会对二叉树进行线索化
树、森林和二叉树的转化，会遍历树和森林
赫夫曼树及其应用
算法：
　　递归遍历二叉树及其应用（P）
　　构造赫夫曼树和赫夫曼编码（A）
　　树和二叉树的转换（A）
　　森林和二叉树的转换（A）
　　遍历树和森林（A）

---

[2] 记号 P，要求达到编写算法和程序的能力。本节下同。

[3] 记号 A，要求掌握算法思想，会演算。本节下同。

7. 图

图的有关概念
熟练掌握图的各种存储结构
图的遍历：深度优先、广度优先（A）
最小生成树算法（两个）及其特点（A）
拓扑排序（A）
关键路径算法（A）
最短路径算法（两个）（A，O^4：时间复杂度）

8. 查找表

查找的有关概念，ASL 等
顺序查找（A，P）
熟练掌握有序表的折半查找算法（A，P，C）
了解索引顺序表
熟练掌握二叉排序树的概念，建立（A），查找（A，P），删除（A），计算 ASL（C）
平衡二叉排序树的概念，建立（A），判断失去平衡的类型，平衡化（A），计算 ASL（C）
了解 B_树，B+树的概念和特点
知道键树（数字查找树）
哈希表的概念、特点、构造哈希表（A），计算 ASL 和装填因子 α （C）
了解各种查找表的性能（O）

9. 内部排序

直接插入排序（A）
折半插入排序（A，P）
希尔排序（A）
起泡排序（A）
快速排序（A，P，O）
简单选择排序（P，A，O）
堆的概念，调整成堆（A），堆排序（A，O）
归并排序（A，O）
链式基数排序（A，O）
各种排序算法的对比结论（O）

[4] 记号 O，要求掌握算法的时间复杂度。本节下同。

第1章 绪论

一、基础知识

概念和术语(黑体字部分)。
另外，注意：
1、**数据**元素是数据的基本单位。P4
2、**数据项**是数据不可分割的最小单位。P5
3、**数据结构**及其形式定义。P5
四种基本结构：①**集合**②**线性结构**③**树形结构**④**图（网）状结构**
4、数据结构的
逻辑结构（抽象的，与实现无关）
物理结构（存储结构） **顺序映像**（**顺序存储结构**）位置“相邻”
非顺序映像（**链式存储结构**）指针表示关系 P6
5、**数据类型** P7
抽象数据类型（ADT）P7
ADT=（数据对象，数据关系，基本操作）
ADT 细分为**原子类型**，**固定聚合**，**可变聚合类型**。P8
6、**算法**的概念 P13
7、算法的五个特征
①有穷性 ②确定性 ③可行性 ④输入（0 个或多个） ⑤输出（1 个或多个）
8、算法设计的要求：①正确性②可读性③健壮性④效率与低存储量
其中正确性的四个层次（通常要求达到 C 层）。
9、算法的**时间复杂度** P15
常见有： $O(1)$，$O(n)$，$O(n^2)$，$O(\log_2 n)^5$，$O(n \log_2 n)$，$O(2^n)$
语句频度，用归纳法计算。
10、算法的**空间复杂度** P17

二、算法

起泡排序。P16

[5] 分析算法的时间复杂度时，$\log_2 n$ 常简单记作 $\log n$。

另一种形式

```
void BubbleSort ( DataType a[], int n )
{
   for ( i=0; i<n-1; i++ )
      for ( j=0; j<n-i-1; j++ )
         if ( a[j]>a[j+1] )
            a[j]<—>a[j+1];
}
```

或

```
void BubbleSort ( DataType a[], int n )
{
   for ( i=1; i<n; i++ )
      for ( j=0; j<n-i; j++ )
         if( a[j]>a[j+1] )
            a[j]<—>a[j+1];
}
```

或

```
void BubbleSort ( DataType a[], int n )
{
   for ( i=0; i<n-1; i++ ) {
      change = fasle;
      for ( j=0; j<n-i-1; j++ )
         if ( a[j]>a[j+1] ) {
            a[j]<—>a[j+1];
            change = true;
         }
      if ( !change ) break;
   }
}
```

说明：

a) 考试中要求写算法时，可用类 C，也可用 C 程序。

b) 尽量书写算法说明，言简意赅。

c) 技巧：用“边界值验证法”检查下标越界错误。

如上第一个： 第二个循环条件若写作 j<n-i，则当 i=0 时 a[j+1]会越界。

d) 时间复杂度为 $O(n^2)$，第 3 个在最好情况下（待排记录有序），时间复杂度为 O(n)。

三、习题

1.1 编写冒泡排序算法，使结果从大到小排列。

1.2 计算下面语句段中指定语句的频度：

```
1)  for ( i=1; i<=n; i++ )
        for ( j=i; j<=n; j++ )
            x++;------------------------ // @
2)  i = 1;
```

```
while ( i<=n )
   i = i*2; ------------------------- // @
```

第2章 线性表

一、基础知识和算法

1. 线性表及其特点

线性表是 n 个数据元素的有限序列。
线性结构的特点： ①“第一个” ②“最后一个” ③**前驱** ④**后继**。[6]

2. 顺序表——线性表的顺序存储结构

（1）特点

a) 逻辑上相邻的元素在物理位置上相邻。
b) 随机访问。

（2）类型定义

简而言之，“数组+长度”[7]。

```
const int MAXSIZE = 线性表最大长度;
typedef struct{
    DataType elem[MAXSIZE];
    int length;
} SqList;
```

注：a) SqList 为类型名，可换用其他写法。
b) DataType 是数据元素的类型，根据需要确定。
c) MAXSIZE 根据需要确定。如

```
const int MAXSIZE=64;
```

d) 课本上的 SqList 类型可在需要时增加存储空间，在上面这种定义下不可以。(这样做避免了动态内存分配，明显减少了算法的复杂程度，容易理解。而且，原来 Pascal 版本的《数据结构》（严蔚敏）就是这样做的。)

[6] 这里太简炼了，只是为了便于记忆。

[7] 不准确的说法，只为便于理解和记忆，不要在正式场合引用。凡此情形，都加引号以示提醒。

e) 课本上的 SqList 类型定义中 listsize 表示已经分配的空间大小（容纳数据元素的个数）。当插入元素而遇到 L.length==L.listsize 时，用 realloc (L.elem, L.listsize+增量)重新分配内存，而 realloc()函数在必要的时候自动复制原来的元素到新分配的空间中。

（3）基本形态

1°. 顺序表空

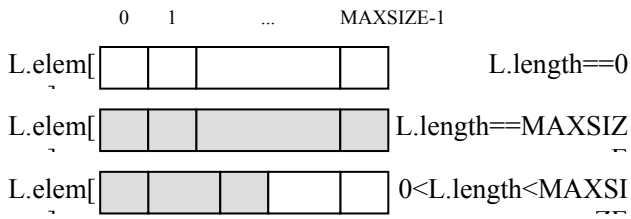
条件　L.length==0
不允许删除操作

2°. 顺序表满

条件　L.length==MAXSIZE
不允许插入操作

3°. 不空也不满

可以插入，删除



（4）基本算法——遍历

1°. 顺序访问所有元素

```
for ( i=0; i<L.length; i++ )
      visit ( L.elem[i] );
```

2°. 查找元素 x

```
for ( i=0; i<L.length; i++ )
   if ( L.elem[i]==x )    break;
if ( i<L.length )
   找到;
else
   未找到;
```

（5）插入算法 ListInsert(&L,i,x)

1°. 前提：表不满

2°. 合理的插入范围：1≤i≤L.length+1

注：位序 i 在 C/C++中对应于下标 i-1。

3°. 步骤

第 i 至最后所有元素后移一个元素
在第 i 个位置插入元素 x
表长增 1

4°. 算法

```
bool[8] ListInsert ( SqList& L, int i, DataType x )
{
  if ( L.length==MAXSIZE || i<1 || i>L.length+1 )   return false; // 失败
  // 元素后移
  for ( j=L.length-1; j>=i-1; j-- )              // 这里 j 为下标，从 L.length-1 到 i-1
        L.elem[j+1] = L.elem[j];                 //    若作为位序，有如何修改？
  // 插入 x
  L.elem[i-1] = x;
  // 表长增 1
  L.length++;
  return true;   // 插入成功
}
```

(6) 删除算法 ListDelete(&L,i,&x)

1°. 前提：表非空

2°. 合理的删除范围：1≤i≤L.length

3°. 步骤

取出第 i 个元素
第 i 个元素之后的元素向前移动一个位置
表长减 1

4°. 算法

```
bool ListDelete ( SqList& L, int i, DataType& x )
{
  if ( L.length==0 || i<1 || i>L.length )   return false; // 失败
  x = L.elem[i-1];
  for ( j=i; j<L.length; j++ )
        L.elem[j-1] = L.elem[j];
  L.length--;
  return true;   // 删除成功
}
```

[8] 这里返回 true 表示正确插入，返回 false 表示插入失败。以下常用来区分操作是否正确执行。

（7）算法分析

表 2.1 顺序表插入和删除算法的分析

	插入	删除
基本操作	移动元素	移动元素
平均移动次数	$\frac{1}{n+1}\sum_{i=1}^{n+1}(n-i+1)=\frac{n}{2}$	$\frac{1}{n}\sum_{i=1}^{n}(n-i)=\frac{n-1}{2}$
时间复杂度	O(n)	O(n)
尾端操作	插入第 n+1 个元素，不移动	删除第 n 个元素，不移动

插入、删除需移动大量元素 O(n)；但在尾端插入、删除效率高 O(1)。

（8）其他算法

1°. InitList (&L), ClearList (&L)

```
L.length = 0;
```

2°. ListEmpty (L)

```
return L.length == 0;
```

3°. ListLength (L)

```
return L.length;
```

4°. GetElem (L, i, &e)

```
e = L.elem[i-1];
```

3. 单链表——线性表的链式存储结构之一

(1) 概念

线性链表，单链表，结点；数据域，指针域；头指针，头结点。

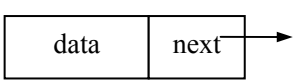
(2) 特点

用指针表示数据之间的逻辑关系（逻辑相邻的元素物理位置不一定相邻）。

(3) 类型定义

简而言之，“数据 + 指针”[9]。

```
typedef struct LNode {
        DataType data;
```



[9] 不准确的说法，只为便于理解和记忆，不要在正式场合引用。

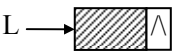
```
        struct LNode *next;
} LNode, *LinkList;
```

(4) 基本形态

带头结点的单链表的基本形态有：

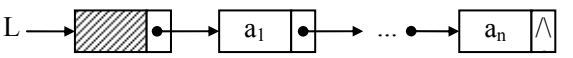
1°. 单链表空

条件： L->next == 0



2°. 单链表不空

条件：L->next != 0



(5) 基本算法 （遍历）

1°. 顺序访问所有元素

借助指针，“顺藤摸瓜”(沿着链表访问结点)。

```
p = L->next;                    // 注意起始位置的考虑
while ( p!=NULL ) {             // 判表尾，另外 (p!=0)或(p)均可
   visit( p->data );            // 访问:可以换成各种操作
   p = p->next;                 // 指针沿着链表向后移动
}
```

例：打印单链表中的数据。

```
void PrintLinkList ( LinkList L )
{
   p = L->next;
   while ( p!=NULL ) {
      print ( p->data );        // 访问：打印数据域
      p = p->next;
   }
}
```

2°. 查找元素 x

```
// 在单链表 L 中查找元素 x
// 若找到，返回指向该结点的指针；否则返回空指针
LinkList Find ( LinkList L, DataType x )
{
   p = L->next;
   while ( p!=NULL ) {
      if ( p->data == x )   return p;   // 找到 x
      p = p->next;
   }
   return NULL;   // 未找到
```

```
}

// 在单链表 L 中查找元素 x
// 若找到，返回该元素的位序；否则返回 0
int Find ( LinkList L, DataType x )
{
   p = L->next;   j = 1;
   while ( p!=NULL ) {
      if ( p->data == x )   return j;   // 找到 x
      p = p->next;   j++;        // 计数器随指针改变
   }
   return 0;   // 未找到
}
```

前一个算法的另一种写法：

```
p = L->next;
while ( p && p->data!=x )
   p = p->next;
if ( p && p->data==x ) return p;
else   return 0;
```

或者

```
p = L->next;
while ( p && p->data!=x )   p = p->next;
return   p;   // 为什么
```

3°. 查找第 i 个元素

```
LinkList Get ( LinkList L, int i )
{
   p = L->next;   j = 1;
   while ( p && j<i ) {
      p = p->next;   j++;
   }
   if ( p && j==i ) return p;
   else   return 0;
}
```

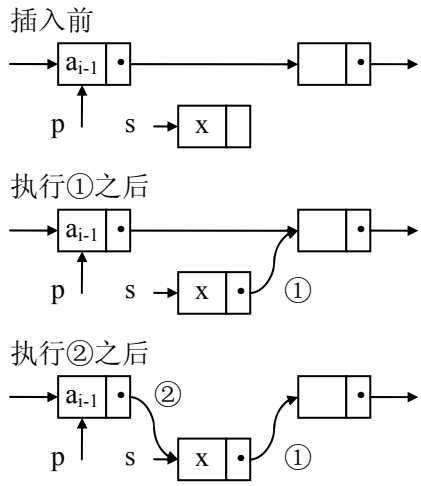
4°. 查找第 i-1 个元素

```
p = L;   j = 0;
while ( p && j<i-1 ) {
   p = p->next;   j++;
}
if ( p && j==i-1 ) return p;
else   return 0;
```

(6) 插入算法 ListInsert(&L,i,x)

技巧：画图辅助分析。
思路：
先查找第 i-1 个元素
若找到，在其后插入新结点

```
bool[10] ListInsert ( LinkList &L, int i, DataType x )
{
  // 查找第 i-1 个元素 p
  p = L;   j = 0;
  while ( p && j<i-1 ) {
    p = p->next;   j++;
  }
  // 若找到，在 p 后插入 x
  if ( p && j==i-1 ) {
    s = (LinkList) malloc(sizeof(LNode));
    s->data = x;
    s->next = p->next;              // ①
    p->next = s;                    // ②
    return true;   // 插入成功
  }
  else
    return false;   // 插入失败
}
```



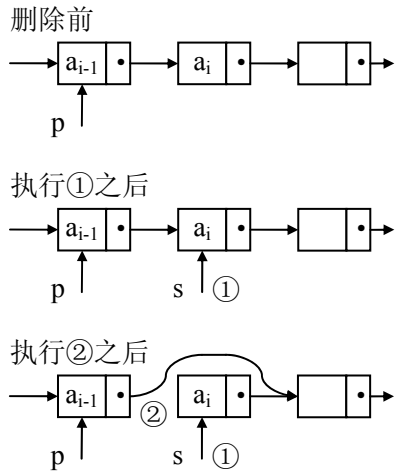
注意：
a) 要让 p 指向第 i-1 个而不是第 i 个元素（否则，不容易找到前驱以便插入）。
b) 能够插入的条件： p && j==i-1 。即使第 i 个元素不存在，只要存在第 i-1 个元素，仍然可以插入第 i 个元素。
c) 新建结点时需要动态分配内存。
s = (LinkList) malloc(sizeof(LNode));
若检查是否分配成功，可用
if (s==NULL) exit(1); // 分配失败则终止程序
d) 完成插入的步骤：①②。技巧：先修改新结点的指针域。

(7) 删除算法 ListDelete(&L,i,&x)

思路：
先查找第 i-1 个元素
若找到且其后存在第 i 个元素，则用 x 返回数据，并删除之

[10] 这里返回 true 表示正确插入，返回 false 表示插入失败。

```
bool ListDelete ( LinkList &L, int i, int &x )
{
  // 查找第 i-1 个元素 p
  p = L;   j = 0;
  while ( p && j<i-1 ) {
    p = p->next;   j++;
  }
  //若存在第 i 个元素，则用 x 返回数据，并删除之
  if ( p && j==i-1 && p->next ) { // 可以删除
    s = p->next;              // ①
    p->next = s->next;        // ②
    x = s->data;
    free (s);
    return true;
  }
  else
    return false;
}
```



注意：

a) 要求 p 找到第 i-1 个而非第 i 个元素。为什么？

b) 能够进行删除的条件： p && j==i-1 && p->next 。条件中的 p->next 就是要保证第 i 个元素存在，否则无法删除。若写成 p->next && j==i-1 也不妥，因为此时(循环结束时)可能有 p==NULL，所以必须先确定 p 不空。技巧：将条件中的“大前提”放在前面。该条件也不可以写成 p->next && p && j==i-1，因为先有 p!=0 才有 p->next，上式颠倒了这一关系。

c) 释放结点的方法。 free(s);

d) 完成删除的步骤：①②。

(8) 建立链表的两种方法

思路：

建立空表（头结点）；

依次插入数据结点（每次插入表尾得($a_1,a_2,\ldots,a_n$)，每次插入表头得($a_n,\ldots,a_2,a_1$)）。

1°. 顺序建表

```
void CreateLinkList ( LinkList &L, int n)
{
  // 建立空表
  L = (LinkList) malloc(sizeof(LNode));
  L->next = NULL;              // 空表
  p = L;                       // 用 p 指向表尾
  // 插入元素
  for ( i=0; i<n; i++ ) {
    scanf ( x );
```

```
        s = (LinkList) malloc(sizeof(LNode));
        s->data = x;
        // 插入表尾
        s->next = p->next;
        p->next = s;
        p = s;                          // 新的表尾
    }
}
```

2°. 逆序建表

```
void CreateLinkList ( LinkList &L, int n)
{
    // 建立空表
    L = (LinkList) malloc(sizeof(LNode));
    L->next = NULL;    // 空表
    // 插入元素
    for ( i=0; i<n; i++ ) {
        scanf ( x );
        s = (LinkList) malloc(sizeof(LNode));
        s->data = x;
        // 插入表头
        s->next = L->next;
        L->next = s;
    }
}
```

4. 循环链表

(1) 特点

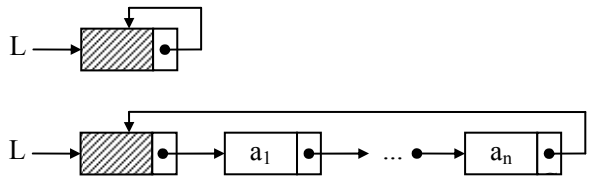
最后一个结点的指针指向头结点。

(2) 类型定义

同单链表。

(3) 基本形态

空表：L->next == L。
非空表。



(4) 与单链表的联系

判断表尾的方法不同：单链表用 p==NULL；循环链表用 p==L。
其余操作相同。

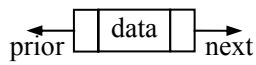
5. 双向循环链表

(1) 特点

一个结点包含指向后继(next)和指向前驱(prior)两个指针，两个方向又分别构成循环链表。

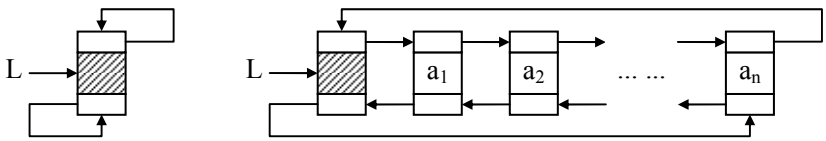
(2) 类型定义

```
typedef struct DuLNode {
    DataType data;
    struct DuLNode *prior, *next;    // 两个指针
} DuLNode, *DuLinkList;
```



(3) 基本形态

空表：用后向指针判断 L->next == L，或者用前向指针判断 L->prior == L。
非空表。



(4) 与单链表和循环链表的联系

最大不同：前驱容易求得，可以向前遍历。
判断表尾的方法与循环链表相同：p==L。
插入和删除时需要修改两个方向的指针。

(5) 插入和删除

需要修改两个方向的指针。例如：(见下表)

表 2.2 双向循环链表的插入和删除

p 之后插入 s	p 之前插入 s	删除 p 之后继 s	删除 p

s->next = p->next;
p->next = s;
s->prior = p;
s->next->prior = s;
	s->prior = p->prior;
	p->prior = s;
	s->next = p;
	s->prior->next = s;
		s = p->next;
		p->next = s->next;
		p->next->prior = p;
			p->prior->next = p->next;
			p->next->prior = p->prior;

6. 顺序表与单链表的比较

表 2.3 顺序表和单链表的比较

顺序表	单链表
以地址相邻表示关系	用指针表示关系
随机访问，取元素 O(1)	顺序访问，取元素 O(n)
插入、删除需要移动元素 O(n)	插入、删除不用移动元素 O(n)(用于查找

	位置)

总结：需要反复插入、删除，宜采用链表；反复提取，很少插入、删除，宜采用顺序表。

二、习题

2.1 将顺序表中的元素反转顺序。
2.2 在非递减有序的顺序表中插入元素 x，并保持有序。
2.3 删除顺序表中所有等于 x 的元素。
2.4 编写算法实现顺序表元素唯一化(即使顺序表中重复的元素只保留一个)，给出算法的时间复杂度。
2.5 非递减有序的顺序表元素唯一化(参见习题 2.4)，要求算法的时间复杂度为O(n)。
2.6 将单链表就地逆置，即不另外开辟结点空间，而将链表元素翻转顺序。
2.7 采用插入法将单链表中的元素排序。
2.8 采用选择法将单链表中的元素排序。
2.9 将两个非递减有序的单链表归并成一个，仍并保持非递减有序。

第3章 栈和队列

一、基础知识和算法

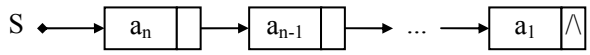
1. 栈

栈，栈顶，栈底，空栈，后进先出(LIFO)，入栈(Push)，出栈(Pop)。
顺序栈：栈的顺序存储结构；**链栈：**栈的链式存储结构。

2. 链栈

(1) 存储结构

用不带头结点的单链表实现。



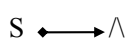
(2) 类型定义

同单链表。

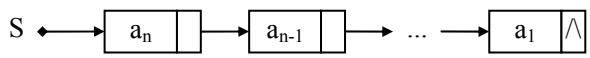
(3) 基本形态

1°. 栈空

条件: S == NULL



2°. 栈非空



3°. 栈满（一般不出现）

(4) 基本算法

1°. 入栈 Push (&s, x)

```
bool Push ( LinkList &s, DataType x )
{
  // 新建结点
```

```
   p = (LinkList) malloc (sizeof(LNode));
   if ( !p )    return false; //  失败
   p->data = x;
   //  插入栈顶
   p->next = s;
   s = p;
   return true;
}
```

2°. 出栈 Pop (&s, &x)

前提：栈非空。

```
bool Pop ( LinkList &s, DataType &x )
{
   if ( s==NULL )    return false;    //  栈空
   //  删除栈顶元素
   p = s;
   s = s->next;
   x = p->data;
   free ( p );
   return true;
}
```

3°. 栈顶元素

前提：栈非空。

```
bool Top ( LinkList &s, DataType &x )
{
   if ( s==NULL )    return false;    //  栈空
   x = s->data;
   return true;
}
```

3. 顺序栈

(1) 存储结构

类似于顺序表，插入和删除操作固定于表尾。

(2) 类型定义

简单说，“数组 + 长度”[11]。

```
const int MAXSIZE =  栈的最大容量;
typedef struct {
   DataType elem[MAXSIZE];
```

[11] 不准确的说法，只为便于理解和记忆，不要在正式场合引用。

```
   int top;
} SqStack;
```

(3) 基本形态

1°. 栈空

条件　s.top == 0;

2°. 栈满

条件　s.top == MAXSIZE

3°. 栈不空、不满

(4) 基本算法

1°. 入栈　Push (&s, x)

前提：栈不满

```
bool Push ( SqStack& s, DataType x )
{
   if ( s.top == MAXSIZE )    return false;    // 栈满
   s.elem[top] = x;                    // 或者 s.elem[top++] = x;
   top++;                              //     代替这两行
   return true;
}
```

2°. 出栈　Pop (&s, &x)

前提：栈非空

```
bool Pop ( SqStack &s, DataType &x )
{
   if ( s.top==0 )    return false;
   top--;                              // 可用 x=s.elem[--top];
   x = s.elem[top];                    //     代替这两行
   return true;
}
```

3°. 栈顶元素

前提：栈非空

s.elem[top-1] 即是。

4. 队列

队列，队头，队尾，空队列，先进先出(FIFO)。

链队列：队列的链式存储结构。
循环队列：队列的顺序存储结构之一。

5. 链队列

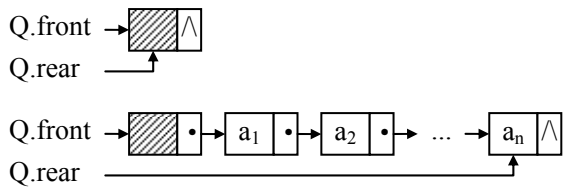
(1) 存储结构

简而言之，“单链表 + 尾指针”[12]。

(2) 类型定义

课本 P61。

```
typedef struct {
    LinkList front;
    LinkList rear;
} LinkQueue;
```



(3) 基本形态

队列空：Q.front==Q.rear。
非空队列。

(4) 基本算法

1°. 入队列

课本 P62。插入队尾，注意保持 Q.rear 指向队尾。

2°. 出队列

课本 P62。删除队头元素，
特别注意：如果队列中只有一个元素，则队头也同时是队尾，删除队头元素后也需要修改队尾指针。

6. 循环队列

(1) 存储结构

简单说，“数组 + 头、尾位置”[13]。

[12] 不准确的说法，只为便于理解和记忆，不要在正式场合引用。
[13] 不准确的说法，只为便于理解和记忆，不要在正式场合引用。

(2) 类型定义

```
const int MAXSIZE =  队列最大容量;
typedef struct {
    DataType elem[MAXSIZE];
    int front, rear;                    // 队头、队尾位置
} SqQueue;
```

(3) 基本形态

通常少用一个元素区分队列空和队列满，也可以加一标志。约定 front 指向队头元素的位置，rear 指向队尾的下一个位置，队列内容为 [front, rear)。

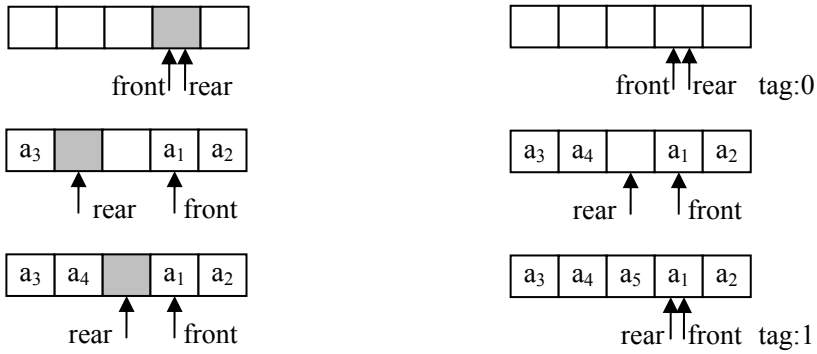
1°. 队列空

条件：Q.front == Q.rear。
不能出队列。

2°. 队列满

条件：(Q.rear+1)%MAXSIZE == Q.front （少用一个元素时）。
不能入队列。

3°. 队列不空也不满



4°. 加一标志区分队列空和队列满的情况

可以用满所有空间。队列空和队列满时都有 Q.front==Q.rear，再用标志区分。
队列空：Q.front==Q.rear **and** Q.tag==0；队列满：Q.front==Q.rear **and** Q.tag==1。

(4) 基本算法

1°. 入队列

前提：队列不满。

```
bool EnQueue ( SqQueue &Q, DataType x )
{
```

```
   if ( (Q.rear+1)%MAXSIZE==Q.front )    return false;   // 队列满
   // 入队列
   Q.elem [Q.rear] = x;
   Q.rear = (Q.rear+1)%MAXSIZE;
   return true;
}
```

2°. 出队列

前提：队列非空。

```
bool DeQueue ( SqQueue &Q, DataType &x )
{
   if ( Q.front==Q.rear )    return false;   // 队列空
   // 出队列
   x = Q.elem [Q.front];
   Q.front = (Q.front+1)%MAXSIZE;
   return true;
}
```

3°. 队列中元素个数

结论：(Q.rear-Q.front+MAXSIZE)%MAXSIZE。

注：Q.rear-Q.front 可能小于 0，需要加上 MAXSIZE。

```
int QueueLength ( SqQueue Q )
{
   return (Q.rear-Q.front+MAXSIZE)%MAXSIZE;
}
```

4°. 用标志区分队列空和满

用标志区分队列空和满时，队列初始化、入队列、出队列和队列长度的算法如下：

```
void InitQueue ( SqQueue &Q ) {
   Q.front = Q.rear = 0;    Q.tag = 0;
}
bool EnQueue ( SqQueue &Q, DataType x ) {
   if ( Q.front==Q.rear and Q.tag==1 )    return false;
   Q.elem[ Q.rear ] = x;
   Q.rear = (Q.rear+1)%MAXSIZE;
   if ( Q.tag==0 )    Q.tag = 1;                          // 队列非空
   return true;
}
bool DeQueue ( SqQueue &Q, DataType &x ) {
   if ( Q.front==Q.rear and Q.tag==0 )    return false;
   x = Q.elem[ Q.front ];
   Q.front = (Q.front+1)%MAXSIZE;
   if ( Q.front==Q.rear )    Q.tag = 0;                   // 队列空
   return true;
}
int QueueLength ( SqQueue Q )
```

```
{
  if ( Q.front==Q.rear and Q.tag==1 )
    return MAXSIZE;                              // 队列满
  else
    return (Q.rear-Q.front+MAXSIZE)%MAXSIZE;     // 队列不满(包含队列空的情
况)
}
```

7. 栈和队列比较

都是线形结构，栈的操作 LIFO（后进先出），队列操作 FIFO（先进先出）。

8. 简化的栈和队列结构

在算法中使用栈和队列时可以采用简化的形式。

表 3.1 简化的栈和队列结构

简化栈		简化队列	
结构	“s[] + top”	结构	“q[] + front + rear”
初始化	top = 0;	初始化	front=rear=0;
入栈	s[top++] = x;	入队列	q[rear] = x; rear = (rear+1)%MAXSIZE;
出栈	x = s[--top];	出队列	x = q[front]; front = (front+1)%MAXSIZE;
栈顶	s[top-1]	队列头	q[front]
栈空	top == 0	队列空	front == rear

说明：只要栈(队列)的容量足够大，算法中可以省去检查栈(队列)满的情况。

9. 栈和队列的应用

1°. 表达式求值

参见课本 P53。

2°. 括号匹配

例：检查表达式中的括号是否正确匹配。如{ ( ) [ ] }正确，([ )] }则错误。
分析：每个左括号都“期待”对应的右括号，匹配成功则可以消去。
思路：遇到左括号则入栈，遇到右括号则与栈顶括号相比较，如果匹配则消去，否则匹配失败。当然，如果栈中没有括号可以匹配，或者最后栈中还有未匹配的左括号，也都是匹配错误。

```
// 检查输入的表达式中括号是否匹配
bool MatchBrackets ()
{
  const int MAXSIZE = 1024;                      // 栈的最大容量
```

```
   char s [MAXSIZE];                              // 简化的栈结构
   int top;                                       // 栈顶
   // 栈初始化
   top = 0;
   // 检查括号是否匹配
   ch = getchar();
   while ( ch!=EOF ) {
      switch ( ch ) {
      case '(', '[', '{':
         s [top++] = ch;                          // 所有左括号入栈
         break;
      case ')':
         if ( top==0 or s [--top]!='(' ) return false;      // 栈空或右括号与栈顶左括号失
配
      case ']':
         if ( top==0 or s [--top]!='[' ) return false;
      case '}':
         if ( top==0 or s [--top]!='{' ) return false;
      }
      ch = getchar();                             // 取下一个字符
   }
   if ( top==0 )    return true;                  // 正好匹配
   else    return false;                          // 栈中尚有未匹配的左括号
}
```

3°. 递归程序的非递归化

将递归程序转化为非递归程序时常使用栈来实现。

4°. 作业排队

如操作系统中的作业调度中的作业排队，打印机的打印作业也排成队列。

5°. 按层次遍历二叉树

```
void LevelOrder ( BinTree bt, VisitFunc visit )
{
   const int MAXSIZE = 1024;     // 队列容量(足够大即可)
   BinTree q [MAXSIZE];     // 简化的队列结构
   int front, rear;                 //     队头、队尾

   if ( ! bt )    return ;
   // 初始化队列，根结点入队列
   front = rear = 0;
   q [rear] = bt;
   rear = (rear+1)%MAXSIZE;
   // 队列不空，则取出队头访问并将其左右孩子入队列
   while ( front!=rear ) {
```

```
        p = q [front];
        front = (front+1)%MAXSIZE;
        if ( p ) {
            visit ( p->data );        // 访问结点
            q [rear] = p->lchild;
            rear = (rear+1)%MAXSIZE;
            q [rear] = p->rchild;
            rear = (rear+1)%MAXSIZE;
        }
    }
}
```

二、习题

3.1 元素 1,2,3,4 依次入栈，不可能的出栈序列有哪些？
3.2 设循环队列 Q 少用一个元素区分队列空和队列满，MAXSIZE=5，Q.front=Q.rear=0，画出执行下列操作时队列空和队列满的状态。入队列 a,b,c，出队列 a,b,c，入队列 d,e,f,g。
3.3 编写算法利用栈将队列中的元素翻转顺序。

第4章 串

一、基础知识和算法

1. 概念

串，空串，空格串，串的长度；子串，子串在主串中的位置，主串；串相等。

2. 串的基本操作

表 4.1 串的基本操作

Assign (s, t), Create (s, cs)	Assign(s,t)将变量 t 赋值给 s，Create(s,cs)根据字串创建变量 s。
Equal (s, t), Length (s)	判断串相等，求串长度。如 Length(“”)=0。
Concat (s, t)	串连接。如 Concat(“ab”,”cd”)=”abcd”。
Substr (s, pos, len)	取子串，pos 为开始位置，len 为子串长度。
Index (s, t)	求子串 t 在主串 s 中的位置。如 Index(“abc”,”ab”)=1，Index(“a bc”,”bc”)=3。
Replace (s, t, v)	把串 s 中的字串 t 替换成 v。如 Replace(“aaa”,”aa”,”a”)=”aa”。
Delete (s, pos, len)	删除串 s 的一部分。

注：完成习题集 4.1~4.6。

3. 串的存储结构

表 4.2 串的存储结构

定长顺序串	最大长度固定，超过最大长度则作截断处理
堆分配存储表示	串的长度几乎没有限制
块链存储表示	块内存储空间连续，块间不连续

二、习题

4.1 长度为 n 的串的子串最多有多少个？

第6章 树和二叉树

一、基础知识和算法

1. 树及有关概念

树，根，子树；结点，结点的度，叶子（终端结点），分支结点（非终端结点），内部结点，树的度；孩子，双亲，兄弟，祖先，子孙，堂兄弟；层次（根所在层为第 1 层），**深度，高度；有序树，无序树**，二叉树是有序树；**森林**。

2. 二叉树

二叉树（二叉树与度为 2 的树不同，二叉树的度可能是 0，1，2）；**左孩子，右孩子**。二叉树的五种基本形态。

3. 二叉树的性质

1°. 二叉树的第i层[14]上至多有 2^{i-1} 个结点。

2°. 深度为 k 的二叉树至多有 2^k-1 个结点。

满二叉树：深度为 k，有 2^k-1 个结点。
完全二叉树：给满二叉树的结点编号，从上至下，从左至右，n 个结点的完全二叉树中结点在对应满二叉树中的编号正好是从 1 到 n。

3°. 叶子结点 n_0，度为 2 的结点为 n_2，则 $n_0 = n_2+1$。

考虑结点个数：$n = n_0 + n_1 + n_2$
考虑分支个数：$n-1 = 2n_2 + n_1$
可得 $n_0 = n_2+1$
例：1) 二叉树有 n 个叶子，没有度为 1 的结点，共有____个结点。 2) 完全二叉树的第 3 层有 2 个叶子，则共有____个结点。
分析：1) 度为 2 的结点有 n-1 个，所以共 2n-1 个结点。 2) 注意考虑到符合条件的二叉树的深度可能是 3 或 4，所以有 5、10 或 11 个结点。

[14] 本书中约定根结点在第 1 层，也有约定根在第 0 层的，则计算公式会有所不同。

4°. n 个结点的完全二叉树深度为$\lfloor \log n \rfloor + 1$。

5°. n 个结点的完全二叉树，结点按层次编号

有： i 的双亲是$\lfloor n/2 \rfloor$，如果 i = 1 时为根（无双亲）；
i 的左孩子是 2i，如果 2i>n，则无左孩子；
i 的右孩子是 2i + 1，如果 2i + 1>n 则无右孩子。

4. 二叉树的存储结构

1°. 顺序存储结构

用数组、编号 i 的结点存放在[i-1]处。适合于存储完全二叉树。

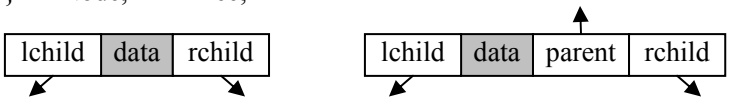
2°. 链式存储结构

二叉链表：

```
typedef struct BTNode {
   DataType data;
   struct BTNode *lchild, *rchild;
} BTNode, *BinTree;
```

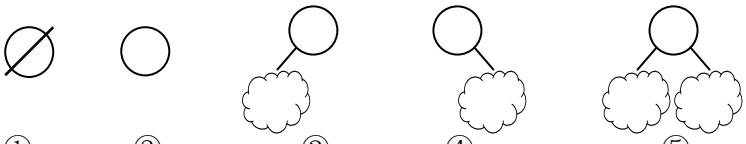
三叉链表：

```
typedef struct BTNode {
   DataType data;
   struct BTNode *lchild, *rchild, *parent;
} BTNode, *BinTree;
```



例：用二叉链表存储n个结点的二叉树(n>0)，共有(_a_)个空指针域；采用三叉链表存储，共有(_b_)个空指针域。[15]

5. 二叉树的五种基本形态



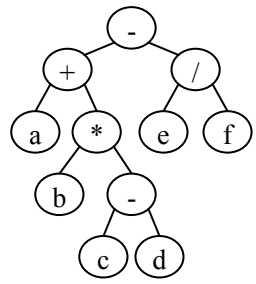
①空树：bt==NULL ②左右子树均空：bt->lchild==NULL **and** bt->rchild==NULL
③右子树为空：bt->rchild==NULL ④左子树为空：bt->lchild==NULL
⑤左右子树均非空。
前两种常作为递归结束条件，后三者常需要递归。

[15] 答案：(a) n+1 (b) n+2。提示：只有根结点没有双亲。

6. 遍历二叉树

1°. 常见有四种遍历方式

按层次遍历，先序遍历，中序遍历，后序遍历。
按层次遍历：“从上至下，从左至右”，利用队列。
先序遍历：DLR；**中序遍历**：LDR；**后序遍历** LRD。



例：写出 a+b*(c-d)-e/f 的前缀、中缀和后缀表达式。
画出二叉树，分别进行前序、中序、后序遍历即可得到。
前缀表达式：- + a * b - c d / e f
中缀表达式：a + b * c - d - e / f
后缀表达式：a b c d - * + e f / -

2°. 先序遍历算法

```
void Preorder ( BinTree bt )
{
   if ( bt ) {
      visit ( bt->data );
      Preorder ( bt->lchild );
      Preorder ( bt->rchild );
   }
}
```

3°. 中序遍历算法

```
void Inorder ( BinTree bt )
{
   if ( bt ) {
      Inorder ( bt->lchild );
      visit ( bt->data );
      Inorder ( bt->rchild );
   }
}
```

4°. 后序遍历

```
void Postorder ( BinTree bt )
{
   if ( bt ) {
      Postorder ( bt->lchild );
      Postorder ( bt->rchild );
      visit ( bt->data );
   }
}
```

5°. 按层次遍历

思路：利用一个队列，首先将根（头指针）入队列，以后若队列不空则取队头元素 p，如果 p 不空，则访问之，然后将其左右子树入队列，如此循环直到队列为空。

```
void LevelOrder ( BinTree bt )
{
  // 队列初始化为空
  InitQueue ( Q );
  // 根入队列
  EnQueue ( Q, bt );
  // 队列不空则继续遍历
  while ( ! QueueEmpty(Q) ) {
    DeQueue ( Q, p );
    if ( p!=NULL ) {
      visit ( p->data );
      // 左、右子树入队列
      EnQueue ( Q, p->lchild );
      EnQueue ( Q, p->rchild );
    }
  }
}
```

若队列表示为“数组 q[] + 头尾 front, rear”有：

```
void LevelOrder ( BinTree bt )
{
  const int MAXSIZE = 1024;
  BinTree q[MAXSIZE];
  int front,  rear;
  // 队列初始化为空
  front = rear = 0;
  // 根入队列
  q[rear] = bt;  rear = ( rear+1 ) % MAXSIZE;
  // 队列不空则循环
  while ( front != rear ) {
    p = q[front];  front = ( front+1 ) % MAXSIZE;
    if ( p ) {
      visit ( p->data );
      // 左、右子树入队列
      q[rear] = p->lchild;  rear = ( rear+1 ) % MAXSIZE;
      q[rear] = p->rchild;  rear = ( rear+1 ) % MAXSIZE;
    }
  }
}
```

6°. 非递归遍历二叉树

一般借助栈实现。设想一指针沿二叉树中序顺序移动，每当向上层移动时就要出栈。

(a) 中序非递归遍历

指针 p 从根开始，首先沿着左子树向下移动，同时入栈保存；当到达空子树后需要退栈访问结点，然后移动到右子树上去。

```
void InOrder ( BinTree bt, VisitFunc visit )
{
   InitStack ( S );
   p = bt;
   while ( p || ! StackEmpty(S) ) {
      if ( p ) {
         Push ( S, p );
         p = p->lchild;
      } else {
         Pop ( S, p );
         visit ( p );                          // 中序访问结点的位置
         p = p->rchild;
      }
   }
}
```

(b) 先序非递归遍历

按照中序遍历的顺序，将访问结点的位置放在第一次指向该结点时。

```
void Preorder ( BinTree bt, VisitFunc visit )
{
   InitStack ( S );
   p = bt;
   while ( p || ! StackEmpty(S) ) {
      if ( p ) {
         visit ( p );                    // 先序访问结点的位置
         Push ( S, p );
         p = p->lchild;
      } else {
         Pop ( S, p );
         p = p->rchild;
      }
   }
}
```

或者，由于访问过的结点便可以弃之不用，只要能访问其左右子树即可，写出如下算法。

```
void Preorder ( BinTree bt, VisitFunc visit )
{
   InitStack ( S );
   Push ( S, bt );
   while ( ! StackEmpty(S) ) {
      Pop ( S, p );
      if ( p ) {
         visit ( p );
         Push ( S, p->rchild );          // 先进栈，后访问，所以
         Push ( S, p->lchild );          //    这里先让右子树进栈
      }
```

```
    }
}
```

(c) 后序非递归遍历

后序遍历时，分别从左子树和右子树共两次返回根结点，只有从右子树返回时才访问根结点，所以增加一个栈标记到达结点的次序。

```
void PostOrder ( BinTree bt, VisitFunc visit )
{
   InitStack ( S ),   InitStack ( tag );
   p = bt;
   while ( p || ! StackEmpty(S) ) {
      if ( p ) {
         Push ( S, p ),   Push ( tag, 1);     // 第一次入栈
         p = p->lchild;
      } else {
         Pop ( S, p ),   Pop ( tag, f );
         if ( f==1 ) {
            // 从左子树返回，二次入栈，然后 p 转右子树
            Push ( S, p ),   Push ( tag, 2 );
            p = p->rchild;
         } else {
            // 从右子树返回(二次出栈)，访问根结点，p 转上层
            visit ( p );
            p = NULL;                        // 必须的，使下一步继续退栈
         }
      }
   }
}
```

注：后序非递归遍历的过程中，栈中保留的是当前结点的所有祖先。这是和先序及中序遍历不同的。在某些和祖先有关的算法中，此算法很有价值。

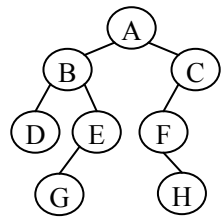
7°. 三叉链表的遍历算法

下面以中序遍历为例。

```
// 中序遍历三叉链表存储的二叉树
void Inorder ( BinTree bt, VisitFunc visit )
{
   if ( bt==NULL )   return;                 // 空树，以下考虑非空树
   // 找到遍历的起点
   p = bt;                                   // Note: p!=null here
   while ( p->lchild )   p = p->lchild;
   // 开始遍历
   while ( p ) {
      // 访问结点
      visit ( p );
      // p 转下一个结点
      if ( p->rchild ) {                     // 右子树不空，下一个在右子树
         p = p->rchild;
```

```
        while ( p->lchild )   p = p->lchild;   //    转右子树的最左下结点
      } else {                                 // 右子树为空，下一个在上层
        f = p->parent;
        while ( p == f->rchild ) {             //    若 p 是右子树则一直上溯
          p = f;   f = f->parent;
        }
      }
    }
}
```

7. 遍历二叉树的应用



1°. 写出遍历序列（前、中、后序）

2°. 根据遍历序列画出二叉树

(a) 已知前序和中序序列，唯一确定二叉树。
例：前序：ABDEGCFH，中序：DBGEAFHC，画出二叉树。
分析：前序序列的第一个是根，这是解题的突破口。
步骤：①前序序列的第一个是根 ②在中序序列中标出根，分成左右子树 ③在前序序列中标出左右子树(根据结点个数即可) ④分别对左右子树的前序和中序序列重复以上步骤直至完成。
(b) 已知后序和中序序列，唯一确定二叉树。
例：后序：DGEBHFCA，中序：DBGEAFHC，画出二叉树。
分析：后序序列的最后一个是根，这是解题的突破口。
步骤：①后序序列的最后一个是根 ②在中序序列中标出根，分成左右子树 ③在后序序列中标出左右子树(根据结点个数即可) ④分别对左右子树的后序和中序序列重复以上步骤直至完成。
(c) 已知前序和后序序列，不存在度为 1 的结点时能唯一确定二叉树。
例：前序：ABDEC，后序：DEBCA，画出二叉树。又前序 AB，后序 BA 则不能唯一确定二叉树。
注：对于不存在度为 1 的结点的二叉树，首先确定根结点，然后总可以将其余结点序列划分成左右子树，以此类推即可确定二叉树。

说明：画出二叉树后可以进行遍历以便验证。

3°. 编写算法

思路：按五种形态(①--⑤)分析，适度简化。
例：求二叉树结点的个数。
分析：① 0; ② 1; ③ L+1; ④ 1+R; ⑤ 1+L+R。
简化：② $1+L_{=0}+R_{=0}$ ③ $1+L+R_{=0}$ ④ $1+L_{=0}+R$ ⑤ 1+L+R 可合并成⑤一种情况。

```
int NodeCount ( BinTree bt )
{
   if ( bt==0 )    return 0;
   else    return 1 + NodeCount(bt->lchild) + NodeCount(bt->rchild);
}
```

例：求二叉树叶子结点的个数。
分析：① 0; ② 1; ③ L; ④ R; ⑤ L+R。简化：③④⑤可合并成⑤。

```
int LeafCount ( BinTree bt )
{
   if ( bt==0 )    return 0;
   else if ( bt->lchild==0 and bt->rchild==0 )    return 1;
   else    return LeafCount(bt->lchild) + LeafCount(bt->rchild);
}
```

例：求二叉树的深度。
分析：① 0; ② 1; ③ 1+L; ④ 1+R; ⑤ 1+max(L,R)。简化：②③④⑤可合并成⑤。

```
int Depth ( BinTree bt )
{
   if ( bt==0 )    return 0;
   else    return 1 + max(Depth(bt->lchild), Depth(bt->rchild));
}
```

8. 线索二叉树

1°. 线索

n 个结点的二叉链表中有 n+1 个空指针，可以利用其指向前驱或后继结点，叫**线索**，同时需附加一个标志，区分是子树还是线索。

lchild　有左子树，则指向左子树，标志 ltag == 0；
　　没有左子树，可作为前驱线索，标志 ltag == 1。
rchild　有右子树，则指向右子树，标志 rtag == 0；
　　没有右子树，可作为后继线索，标志 rtag == 1。

2°. 线索化二叉树

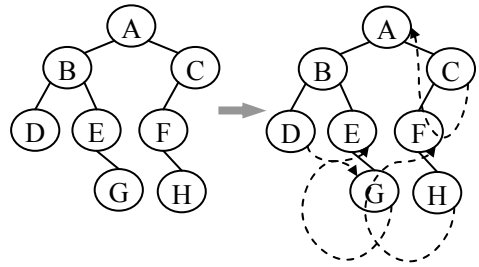
利用空指针作为线索指向前驱或后继。左边空指针可以作为前驱线索，右边空指针可以作为后继线索，可以全线索化或部分线索化。

表 6.1 线索化二叉树的类型

	前驱、后继线索	前驱线索	后继线索
中序线索化	中序全线索	中序前驱线索	中序后继线索
前序线索化	前序全线索	前序前驱线索	前序后继线索
后序线索化	后序全线索	后序前驱线索	后序后继线索

3°. 画出线索二叉树

思路：先写出遍历序列，再画线索。
步骤： 标出必要的空指针（前驱→左指针；后继→右指针，要点：“不要多标，也不要少标”）。
写出对应的遍历序列（前序，中序或后序）。
对照遍历结果画线索。
例：画出图中二叉树的后序后继线索。
步骤：先标出所有空的右指针(DGCH)；写出后序遍历结果：DGEBHFCA；标出后继线索：D→G, G→E, C→A, H→F。如图。



4°. 遍历线索二叉树

反复利用孩子和线索进行遍历，可以避免递归。

9. 树和森林

1°. 树的存储结构

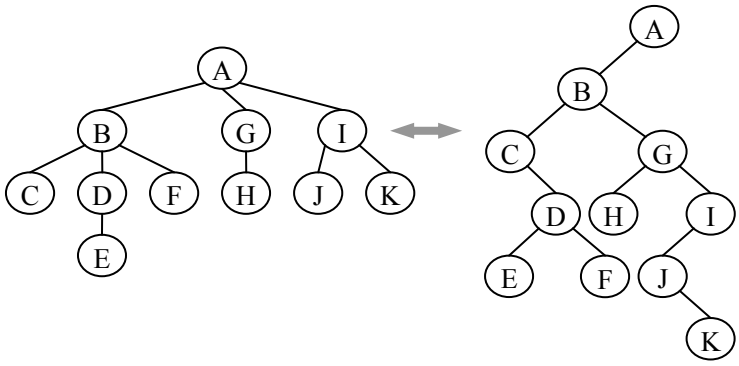
双亲表示法，孩子表示法，孩子兄弟表示法。
特点：双亲表示法容易求得双亲，但不容易求得孩子；孩子表示法容易求得孩子，但求双亲麻烦；两者可以结合起来使用。孩子兄弟表示法，容易求得孩子和兄弟，求双亲麻烦，也可以增加指向双亲的指针来解决。

2°. 树与二叉树的转换

表 6.2 树和二叉树的对应关系

树	对应的二叉树
根	根
第一个孩子	左孩子
下一个兄弟	右孩子

特点：由树转化成的二叉树，根结点没有右孩子
例：树转换成二叉树。



3°. 森林与二叉树的转换

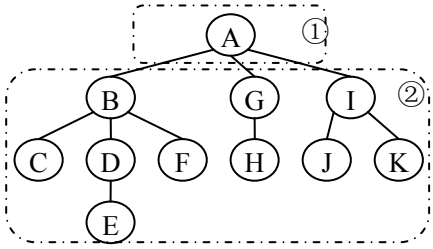
森林中第 1 棵树的根作为对应的二叉树的根；其他的树看作第 1 棵树的兄弟；森林中的树转换成对应的二叉树。则森林转换成对应的二叉树。
例：将森林转换成对应的二叉树。参见课本 P138。

4°. 树的遍历

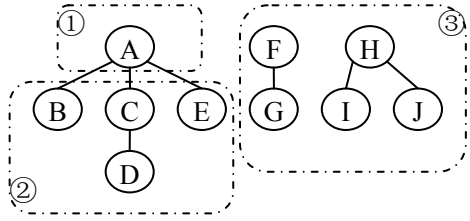
树的结构：①根，②根的子树。
先根遍历：①②。例：ABCDEFGHIJK。
后根遍历：②①。例：CEDFBHGJKIA。

5°. 遍历森林

森林的结构：①第一棵树的根，②第一棵树的根的子树森林，③ 其余树(除第一棵外)组成的森林。
先序遍历：①②③。例：ABCDEFGHIJ。
中序遍历：②①③。例：BDCEAGFIJH。
注：先序遍历森林，相当于依次先根遍历每一棵树；中根遍历森林相当于后根遍历每一棵树。



树的结构划



森林的结构划分

6°. 遍历树、森林与遍历二叉树的关系

表 6.3 遍历树、森林和二叉树的关系

树	森林	二叉树
先根遍历	先序遍历	先序遍历
后根遍历	中序遍历	中序遍历

10. 赫夫曼树及其应用

1°. 最优二叉树(赫夫曼树，哈夫曼树)

树的带权路径长度：所有叶子结点的带权路径长度之和。

$$WPL = \sum_{k=1}^{n} w_k l_k$$

路径长度 l_k 按分支数目计算。
带权路径长度最小的二叉树称为**最优二叉树**，或**赫夫曼树(哈夫曼树)**。

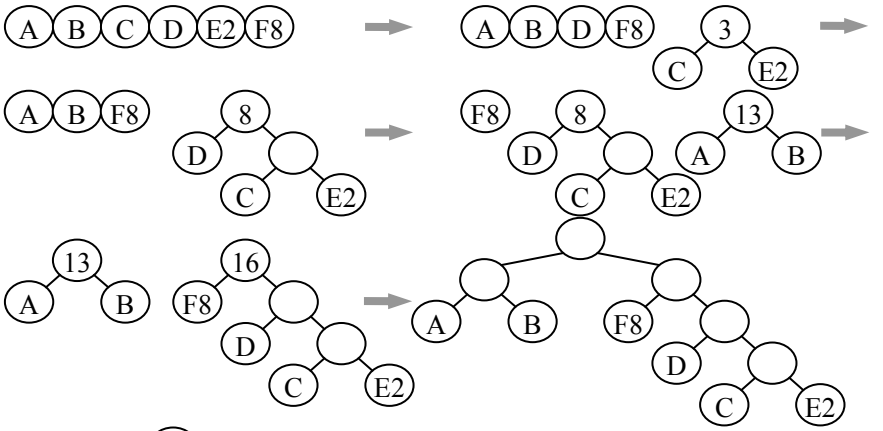
2°. 构造赫夫曼树

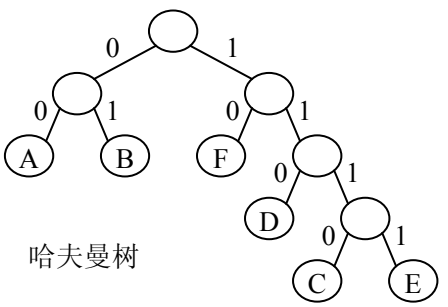
算法：参见课本 P145。
简单说，“每次取两个最小的树组成二叉树”[16]。

3°. 赫夫曼编码(前缀码)

向左分支为 0，向右分支为 1，从根到叶子的路径构成叶子的前缀编码。

例：字符及其权值如下：A(6), B(7), C(1), D(5), E(2), F(8)，构造哈夫曼树和哈夫曼编码，计算带权路径长度。





哈夫曼树

哈夫曼编码：
A:00
B:01
C:1110
D:110
E:1111
F:10

WPL = (6+7+8)*2+5*3+(1+2)*4 = 69

或采用课本上的算法计算，如下。

表 6.4 赫夫曼算法

		weight	parent	lchild	rchild

		weight	parent	lchild	rchild

[16] 不准确的说法，只为便于理解和记忆，不要在正式场合引用。

1	A	6	0	0	0
2	B	7	0	0	0
3	C	1	0	0	0
4	D	5	0	0	0
5	E	2	0	0	0
6	F	8	0	0	0
7		0	0	0	0
8		0	0	0	0
9		0	0	0	0
10		0	0	0	0
11		0	0	0	0

1	A	6	9	0	0
2	B	7	9	0	0
3	C	1	7	0	0
4	D	5	8	0	0
5	E	2	7	0	0
6	F	8	10	0	0
7		3	8	3	5
8		8	10	4	7
9		13	11	1	2
10		16	11	6	8
11		29	0	9	10

结果同上。

说明：同样的一组权值可能构造出不同的哈夫曼树，结果不一定唯一，但带权路径长度都是最小的。

技巧：要使前一种方法构造出的赫夫曼树和课本上算法产生的一样，只要把每次合并产生的二叉树放在树的集合的末尾，并且总是用两个最小树的前者作为左子树后者作为右子树。

二、习题

6.1 度为 k 的树中有 n_1 个度为 1 的结点，n_2 个度为 2 的结点，…，n_k 个度为 k 的结点，问该树中有多少个叶子结点。

6.2 有 n 个叶子结点的完全二叉树的高度是多少?

6.3 编写算法按照缩进形式打印二叉树。

6.4 编写算法按照逆时针旋转 90 度的形式打印二叉树。

6.5 编写算法判断二叉树是否是完全二叉树。

6.6 编写算法求二叉树中给定结点的所有祖先。

6.7 编写算法求二叉树中两个结点的最近的共同祖先。

6.8 编写算法输出以二叉树表示的算术表达式（中缀形式），要求在必要的地方输出括号。

6.9 树采用孩子-兄弟链表存储，编写算法求树中叶子结点的个数。

6.10 采用孩子-兄弟链表存储树，编写算法求树的度。

6.11 采用孩子-兄弟链表存储树，编写算法求树的深度。

6.12 已知二叉树的前序和中序序列，编写算法建立该二叉树。

6.13 树 T 的先根遍历序列为 GFKDAIEBCHJ，后根遍历序列为 DIAEKFCJHBG，画出树 T。

6.14 一森林 F 转换成的二叉树的先序序列为 ABCDEFGHIJKL，中序序列为 CBEFDGAJIKLH。画出森林 F。

6.15 某通信过程中使用的编码有 8 个字符 A,B,C,D,E,F,G,H，其出现的次数分别为 20, 6, 34, 11, 9, 7, 8, 5。若每个字符采用 3 位二进制数编码，整个通信需要多少字节？请给出哈夫曼编码，以及整个通信使用的字节数?

6.16 n 个权值构造的哈夫曼树共有多少个结点?

第7章 图

一、基础知识和算法

1. 图的有关概念

图，顶点，弧，弧头，弧尾；有向图（顶点集+弧集），0≤e≤n(n-1)，**无向图**（顶点集+边集），0≤e≤n(n-1)/2；**稀疏图**（e<nlogn），**稠密图**；**完全图** e=n(n-1)/2，**有向完全图** e=n(n-1)；**网，有向网，无向网。子图，邻接点，顶点的度，入度，出度；路径，路径长度**（经过边或弧的数目），**简单路径，回路**（环），**简单回路**（简单环）；**连通图，连通分量，强连通分量**。

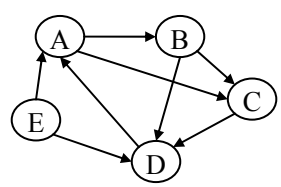
例：有 6 个顶点组成的无向图构成连通图，最少需要(_a_)条边；当边的数目大于(_b_)时，该图必定连通。
分析：a. 5。最少有 n-1 条边就可以构成连通图。 b. 10。考虑将 n 个顶点分成两组，一组有 n-1 个顶点，另一组只有 1 个顶点。首先在第一组中添加边，直到 n-1 个顶点构成全连通子图，共(n-1)(n-2)/2 条边，此后若再在图中任意添加一条边将必定连通两组顶点，从而构成连通图。
思考：对有向图有何结论。

2. 图的存储结构

(1) 图的存储结构

常见图的存储结构有：**邻接矩阵，邻接表，逆邻接表，十字链表，邻接多重表**。
邻接多重表只适用于存储无向图，其他存储结构可以存储无向图和有向图。

例：画出图的邻接矩阵、邻接表、逆邻接表和十字链表。



(2) 邻接矩阵

简言之，“数组(顶点)+二维数组(弧)+个数”[17]。
const int MAX_VERTEX = 最大顶点个数;

[17] 不准确的说法，只为便于理解和记忆，不要在正式场合引用。

```
typedef struct Graph {    // 图
   VertexType                                    vexs[MAX_VERTEX];    // 顶点向量
   ArcType     arcs[MAX_VERTEX] [MAX_VERTEX];    // 邻接矩阵
   int         vexnum, arcnum;                   // 顶点和弧的个数
} Graph;
```

图：有边(弧)为 1；否则为 0。网：有边(弧)为权值；否则为∞。
存储空间个数为 n^2，与边的数目无关。
无向图的邻接矩阵是对称的。

A	0	1	1	0	0
B	0	0	1	1	0
C	0	0	0	1	0
D	1	0	0	0	0
E	1	0	0	1	0

(3) 邻接表

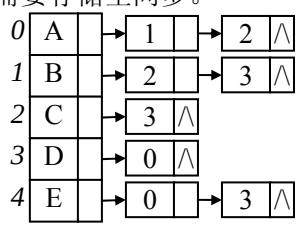
简言之，“数组(弧尾顶点)+链表(邻接点)+个数”[18]。

```
typedef struct ArcNode {    // 弧结点
   int         adjvex;                           // 邻接点
   struct ArcNode *nextarc;                      // 下一个邻接点
} ArcNode;

typedef struct VexNode {    // 顶点结点
   VertexType                                    data;  // 顶点信息
   ArcNode *firstarc;                            // 第一个邻接点
} VexNode;

const int MAX_VERTEX =  最大顶点个数;
typedef struct Graph {    // 图
   VexNode     vexs[MAX_VERTEX];                 // 顶点向量
   int         vexnum, arcnum;                   // 顶点和弧的个数
} Graph;
```

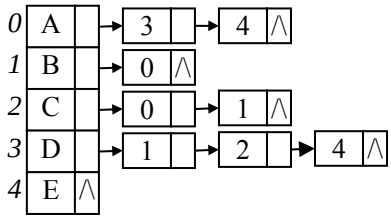
边(弧)多则需要存储空间多。



[18] 不准确的说法，只为便于理解和记忆，不要在正式场合引用。

(4) 逆邻接表

简言之，“数组(弧头顶点)+链表(逆邻结点)+个数”[19]。
类型定义类似邻接表。



(5) 十字链表

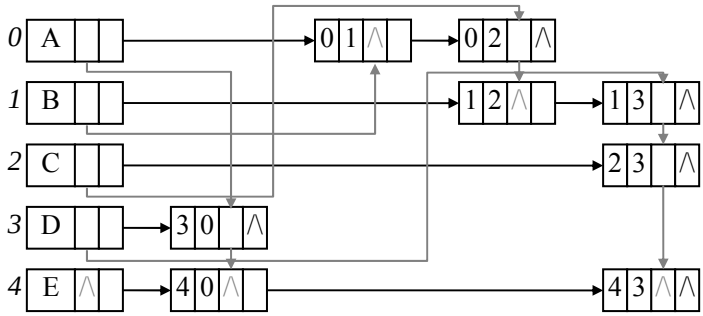
简言之，“数组(顶点)+弧结点(含头和尾)+个数”[20]。边可以看作两条弧。

```
typedef struct ArcNode {    // 弧结点
   int          vtail, vhead;                  // 弧尾和弧头顶点编号
   struct ArcNode   *nexttail, *nexthead;      // 指向同弧尾和同弧头的弧结点
} ArcNode;

typedef struct VexNode {    // 顶点结点
   VertexType                                  data;  // 顶点信息
   ArcNode     *firstin, *firstout;            // 指向第一条入弧和第一条出弧
} VexNode;

const int MAX_VERTEX =  最大顶点个数;
typedef struct Graph {    // 图
   VexNode     vexs[MAX_VERTEX];               // 顶点向量
   int         vexnum, arcnum;                 // 顶点和弧的个数
} Graph;
```

弧结点中包含两个指针分别指向同一弧头的下一个弧和同一个弧尾的下一个弧。顶点结点则指向第一个同弧头和弧尾的弧。十字链表相当于邻接表和逆邻接表的结合。



[19] 不准确的说法，只为便于理解和记忆，不要在正式场合引用。
[20] 不准确的说法，只为便于理解和记忆，不要在正式场合引用。

技巧：把弧结点按行排整齐，然后画链表。同弧尾的弧组成链表，同弧头的弧组成链表。

(6) 邻接多重表

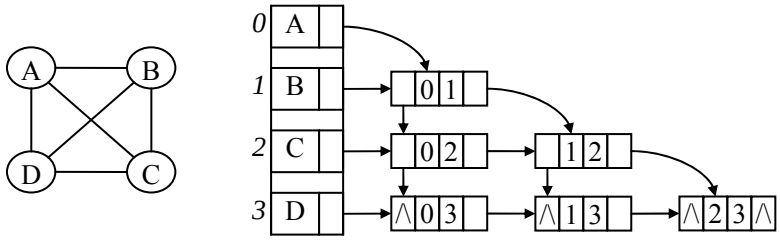
简言之，“数组(顶点)+边结点”[21]。

```
typedef struct EdgeNode {    // 边结点
   int            vexi, vexj;                    // 边的两个顶点
   struct EdgeNode    *nexti, *nextj;            // 两个顶点所依附的下一条边
} EdgeNode;

typedef struct VexNode {    // 顶点结点
   VertexType                                    data;  // 顶点信息
   EdgeNode   *firstedge;                        // 指向第一条边
} VexNode;

const int MAX_VERTEX =  最大顶点个数;
typedef struct Graph {    // 图
   VexNode     vexs[MAX_VERTEX];                 // 顶点向量
   int            vexnum, edgenum;               // 顶点和边的个数
} Graph;
```

只适合存储无向图，不能存储有向图。



技巧：把边结点按列排整齐，然后画链表。相同顶点组成链表，这里没有起点和终点的区别。

3. 图的遍历

(1) 深度优先搜索

1°. 遍历方法

从图中某个顶点出发，访问此顶点，然后依次从其未被访问的邻接点出发深度优先遍历图；若图中尚有顶点未被访问，则另选图中一个未被访问的顶点作为起始点，重复

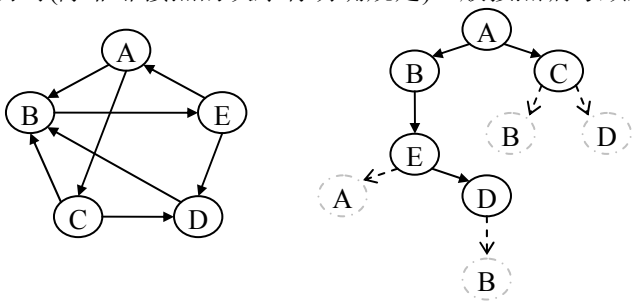
[21] 不准确的说法，只为便于理解和记忆，不要在正式场合引用。

上述过程，直到图中所有顶点都被访问为止。

2°. 分析方法

方法：画一棵“深度优先搜索树”。

例：下图从 A 出发深度优先搜索的结果是：ABEDC。
分析：画“深度优先搜索树”。从 A 出发，访问 A(画圈作标记)，A 的邻接点有 B 和 C(作为 A 的孩子)，B 未访问，访问 B(画圈)，B 的邻接点有 E(作 B 的孩子)，...，以此类推，画出搜索树。深度优先搜索的过程就是沿着该搜索树先根遍历的过程。
技巧：顶点的邻接点是无所谓次序的，所以同一个图的深度优先遍历序列可能不同，但在遍历时(除非邻接点的次序有明确规定)一般按照编号顺序安排邻接点的次序。



3°. 算法

课本上的算法稍加改动：

```
void DFSTraverse ( Graph G )
{
   visited [0 .. G.vexnum-1] = false;          // 初始化访问标志为未访问(false)
   for ( v=0; v<G.vexnum; v++ )
      if ( ! visited[v] )    DFS ( G, v );      // 从未被访问的顶点开始 DFS
}

void DFS ( Graph G, int v )
{
   visit ( v );    visited [v] = true;          // 访问顶点 v 并作标记
   for ( w=FirstAdjVex(G,v); w>=0; w=NextAdjVex(G,v,w) )
      if ( ! visited[w] )    DFS ( G, w );      // 分别从每个未访问的邻接点开始 DFS
}
```

其中的 FirstAdjVex(G,v)表示图 G 中顶点 v 的第一个邻接点，NextAdjVex(G,v,w)表示图 G 中顶点 v 的邻接点 w 之后 v 的下一个邻接点。
深度优先搜索算法有广泛的应用，以上算法是这些应用的基础。

(2) 广度优先搜索

1°. 遍历方法

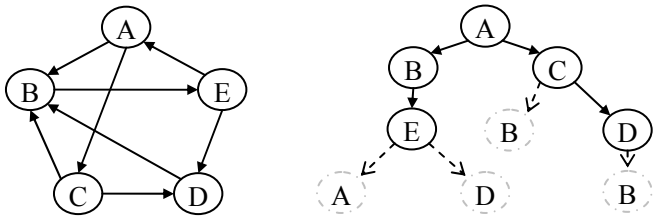
从图中某顶点出发，访问此顶点之后依次访问其各个未被访问的邻接点，然后从这些

邻接点出发依次访问它们的邻接点，并使“先被访问的顶点的邻接点”要先于“后被访问的顶点的邻接点”被访问，直至所有已被访问的顶点的邻接点都被访问。若图中尚有顶点未被访问，则另选图中未被访问的顶点作为起始点，重复以上过程，直到图中所有顶点都被访问为止。
广度优先搜索从某顶点出发，要依次访问路径长度为 1，2，… 的顶点。

2°. 分析方法

方法：画一棵“广度优先搜索树”。

例：下图从 A 出发广度优先遍历的结果是：ABCED。
分析：画“广度优先搜索树”。与深度优先搜索树类似，A 为根，其邻接点为其孩子，访问一个顶点，则扩展出其孩子。不过广度优先搜索的访问次序是对该树按层遍历的结果。



3°. 算法

利用队列(类似按层遍历二叉树)。

```
void BFSTraverse ( Graph G )
{
   visited [0 .. G.vexnum-1] = false;          // 初始化访问标志为未访问(false)
   InitQueue ( Q );
   for ( v=0; v<G.vexnum; v++ )
      if ( ! visited[v] )    {
         // 从 v 出发广度优先搜索
         visit ( v );    visited [v] = true;
         EnQueue ( Q, v );
         while ( ! QueueEmpty(Q) )    {
            DeQueue ( Q, u );
            for ( w=FirstAdjVex(G,u); w>=0; w=NextAdjVex(G,u,w) )
               if ( ! visited[w] )    {
                  visit ( w );    visited [w] = true;
                  EnQueue ( Q, w );
               }
         }
      }
}
```

(3) 时间复杂度分析

观察搜索树可以看出，无论是深度优先搜索还是广度优先搜索，其搜索过程就是对每

个顶点求所有邻接点的过程。当用邻接表存储图时，其时间复杂度为 O(n+e)；当采用邻接矩阵作为存储结构时，时间复杂度是 $O(n^2)$ (因为求一个顶点的所有邻接点就是搜索邻接矩阵的一行中的 n 个数，而顶点的个数为 n，总共就是 n^2)。

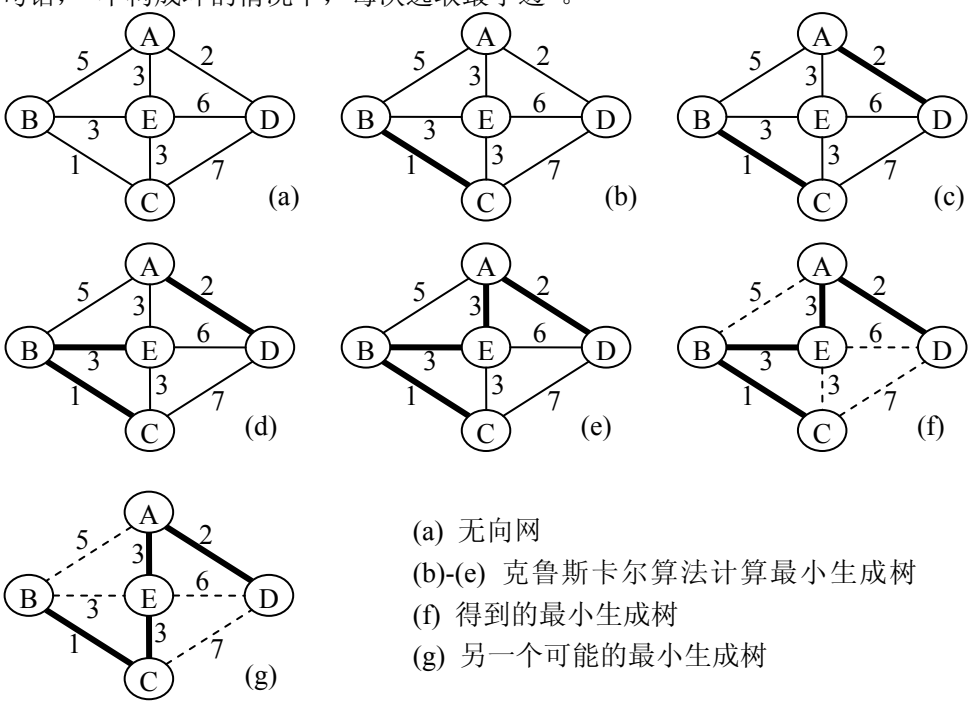
4. 最小生成树

(1) 最小生成树及MST性质

最小生成树。MST 性质。
注意：同一个连通网的最小生成树可能是不唯一的，但其代价都是最小(唯一的)。

(2) 克鲁斯卡尔算法

一句话，“不构成环的情况下，每次选取最小边”。[22]



(a) 无向网
(b)-(e) 克鲁斯卡尔算法计算最小生成树
(f) 得到的最小生成树
(g) 另一个可能的最小生成树

提示：在不要步骤、只要结果的情况下可采用，边较少时特别有效。

(3) 普里姆算法

记 V 是连通网的顶点集，U 是求得生成树的顶点集，TE 是求得生成树的边集。
普里姆算法：

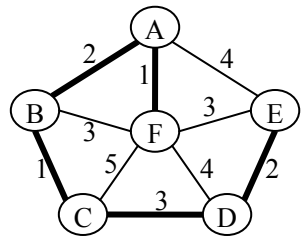
(a) 开始时，U={v_0}，TE=Φ；
(b) 计算 U 到其余顶点 V-U 的最小代价，将该顶点纳入 U，边纳入 TE；

[22] 不准确的说法，只为便于理解和记忆，不要在正式场合引用。

(c) 重复(b)直到 U=V。

例：用普里姆算法计算下图的最小生成树。

U	V-U	U 到 V-U 中各顶点的最小代价					最小代价边
		B	C	D	E	F	
{A}	{B,C,D,E,F}	AB/2	∞	∞	AE/4	AF/1	AF/1
{A,F}	{B,C,D,E}	AB/2	FC/5	FD/4	FE/3		AB/2
{A,F,B}	{C,D,E}		BC/1	FD/4	FE/3		BC/1
{A,F,B,C}	{D,E}			CD/3	FE/3		CD/3



(4) 两种算法的比较

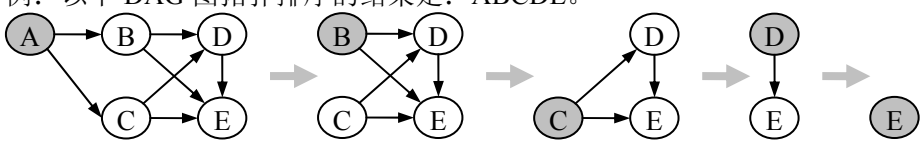
表 7.1 普里姆算法和克鲁斯卡尔算法的比较

算法	普里姆算法	克鲁斯卡尔算法
时间复杂度	$O(n^2)$	O(e loge)
特点	只与顶点个数 n 有关 与边的数目 e 无关 适用于稠密图	只与边的数目 e 有关 与顶点个数 n 无关 适用于稀疏图

5. 拓扑排序

有向无环图(DAG)，AOV 网；**拓扑排序**。
拓扑排序，一句话“每次删除入度为 0 的顶点并输出之”[23]。

例：以下 DAG 图拓扑排序的结果是：ABCDE。

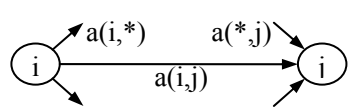


注意：拓扑排序的结果不一定是唯一的。如：ACBDE 也是以上 DAG 图的拓扑有序序列。

6. 关键路径

(1) AOE网，关键路径

AOE 网(活动在边上)，边代表活动或任务，顶点代表事件。
事件 i 发生后，其后继活动 a(i,*)都可以开始；只有所有先导活动 a(*,j)都结束后，事件 j 才发生。



[23] 不准确的说法，只为便于理解和记忆，不要在正式场合引用。

(2) 关键路径算法

问题：a) 整个工程完工需要多长时间？ b) 哪些活动影响工程的进度？或求关键路径。

事件(顶点) i： 最早发生时间 ve(i)，最晚发生时间 vl(i)；

活动(边) a(i,j)： 最早开始时间 e(i,j)，最晚开始时间 l(i,j)。

于是，整个工程完工的时间就是终点的最早发生时间；关键路径就是路径长度最长的路径。

求关键路径的算法：

(a) 按拓扑有序排列顶点：对顶点拓扑排序；

(b) 计算 ve(j)：

$$\begin{cases} ve(1) = 0, \\ ve(j) = \max\{ve(*) + a(*, j)\} \end{cases}$$ 其中*为任意前驱事件；

(c) 计算 vl(i)：

$$\begin{cases} vl(n) = ve(n), \\ vl(i) = \min\{vl(*) - a(i, *)\} \end{cases}$$ 其中*为任意后继事件；

(d) 计算 e(i,j)和 l(i,j)：

$$e(i, j) = ve(i),$$

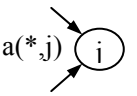
$$l(i, j) = vl(j) - a(i, j)$$

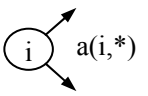
(e) 结论：工程总用时 ve(n)，关键活动是 e(i,j)=l(i,j)的活动 a(i,j)。

说明：

1^0. 若只求工程的总用时只要进行步骤(a)-(b)即可求得。

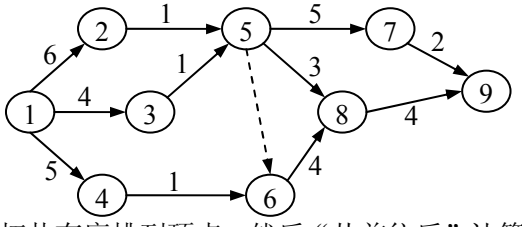
2^0. 如何理解计算 ve(j)和 vl(i)的公式：





事件 j 在所有前驱活动都完成后发生，所以其最早发生时间 ve(j) = max{ve(*)+a(*,j)}，即取决于最慢的前驱活动。另一方面，事件 i 发生后所有后继活动都可以开始了，所以其最晚发生时间 vl(i) = min{vl(*)-a(i,*)}，即不耽误最慢的后继活动。

例：某工程的 AOE 网如下，求 1) 整个工程完工需要多长时间，2) 关键路径。说明：图中的虚线仅表示事件的先后关系，不代表具体活动。



分析：按照拓扑有序排列顶点，然后“从前往后”计算事件的最早发生时间得到总时间，再“从后往前”计算事件的最晚发生时间，最后计算活动的最早和最晚开始时间得到关键活动和关键路径。

表 7.2 关键路径

事件	最早发生时间 ve	最晚发生时间 vl	活动	最早开始时间 e	最晚开始时间 l

v1	0	0	a(1,2)	0	0
v2	6	6	a(1,3)	0	2
v3	4	6	a(1,4)	0	1
v4	5	6	a(2,5)	6	6
v5	7	7	a(3,5)	4	6
v6	7	7	a(4,6)	5	6
v7	12	13	a(5,6)	7	7
v8	11	11	a(5,7)	7	8
v9	15	15	a(5,8)	7	8
			a(6,8)	7	7
			a(7,9)	12	13
			a(8,9)	11	11

所以，1) 工程完工需要时间 15，2) 关键路径是 1→2→5→6→8→9。

7. 最短路径

(1) 迪杰斯特拉算法

求一个顶点到其他各顶点的最短路径。

算法：(a) 初始化：用起点 v 到该顶点 w 的直接边(弧)初始化最短路径，否则设为∞；

(b) 从未求得最短路径的终点中选择路径长度最小的终点 u：即求得 v 到 u 的最短路径；

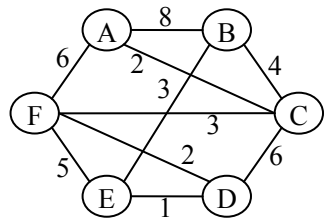
(c) 修改最短路径：计算 u 的邻接点的最短路径，若(v,…,u)+(u,w)<(v,…,w)，则以(v,…,u,w)代替。

(d) 重复(b)-(c)，直到求得 v 到其余所有顶点的最短路径。

特点：总是按照从小到大的顺序求得最短路径。

例：用迪杰斯特拉算法求下图中 A 到其余顶点的最短路径。

终点	从 A 到各顶点的最短路径				
B	8 AB	6 ACB	6 ACB		
C	2 AC				
D	∞	8 ACD	7 ACFD	7 ACFD	
E	∞		10 ACFE	9 ACBE	8 ACFDE
F	6 AF	5 ACF			
最短路径	2 AC	5 ACF	6 ACB	7 ACFD	8 ACFDE



说明：求得 v→u 的最短路径后，只要计算 u 到其余顶点是否(比原来)有更短路径即可，这样可以减少计算量。另外，注意合并路径。

(2) 弗洛伊德算法

求每对顶点之间的最短路径。
依次计算 $A^{(0)}$，$A^{(1)}$，…，$A^{(n)}$。$A^{(0)}$为邻接矩阵，计算 $A^{(k)}$时，$A^{(k)}(i,j)=\min\{A^{(k-1)}(i,j), A^{(k-1)}(i,k)+A^{(k-1)}(k,j)\}$。
技巧：计算$A^{(k)}$的技巧。第k行、第k列、对角线的元素保持不变，对其余元素，考查A(i,j)与A(i,k)+A(k,j) （“行+列”[24]），如果后者更小则替换A(i,j)，同时修改路径。

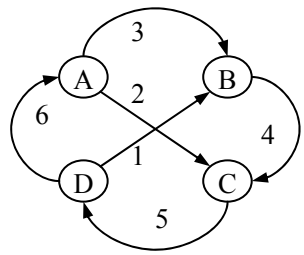
例：用弗洛伊德算法计算图中各顶点之间的最短路径。

$A^{(0)}$

0	3 AB	2 AC	∞
∞	0	4 BC	∞
∞	∞	0	5 CD
6 DA	1 DB	∞	0

$A^{(1)}$

0	3 AB	2 AC	∞
∞	0	4 BC	∞
∞	∞	0	5 CD
6 DA	1 DB	*8 DAC*	0



$A^{(2)}$

0	3 AB	2 AC	∞
∞	0	4 BC	∞
∞	∞	0	5 CD
6 DA	1 DB	*5 DBC*	0

$A^{(3)}$

0	3 AB	2 AC	*7 ACD*
∞	0	4 BC	*9 BCD*
∞	∞	0	5 CD
6 DA	1 DB	5 DBC	0

$A^{(4)}$

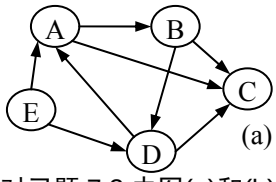
0	3 AB	2 AC	7 ACD
15 BCDA	0	4 BC	9 BCD
11 CDA	*6 CDB*	0	5 CD
6 DA	1 DB	5 DBC	0

技巧：当不变行或不变列(即第 k 行、第 k 列)某元素为∞时，其所在的列或行元素也不变。例如：计算 $A^{(1)}$时，A(2,1) = A(3,1) = ∞，所以第 2、3 行都不变，而 A(1,4) = ∞，所以第 4 列也不变，这样，只剩下 A(4,2)和 A(4,3)需要计算了。

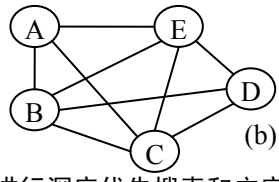
二、习题

7.1 具有 n 个顶点的有向图构成强连通图最少有多少条弧？当弧的数目超过多少时该图一定是强连通的？
7.2 给出下面有向图(a)的 1)邻接矩阵 2)邻接表 3)逆邻接表 4) 十字链表 和 无向图(b)的 5)邻接多重表。

[24] 第 k 列 i “行”元素加上第 k 行 j “列”元素。



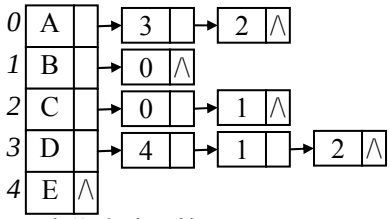
(a)



(b)

7.3 对习题 7.2 中图(a)和(b)从A出发进行深度优先搜索和广度优先搜索，写出遍历结果。

7.4 下面给出图 G 的邻接矩阵，请写出从顶点 A 出发深度优先遍历的结果。

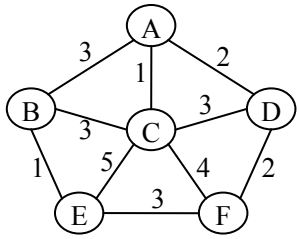


7.5 写出图的深度优先遍历算法。

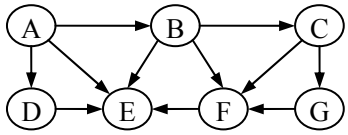
7.6 写出图的广度优先遍历算法。

7.7 证明最小生成树的 MST 性质。

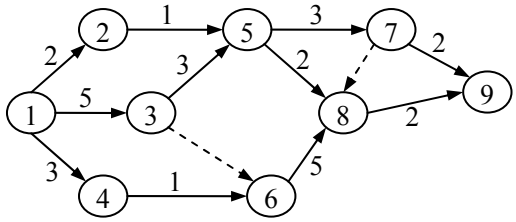
7.8 分别用普里姆算法和克鲁斯卡尔算法计算下图的最小生成树。



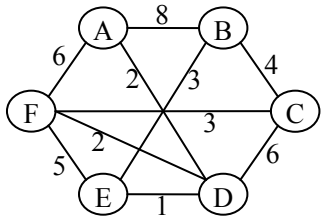
7.9 对下图进行拓扑排序,写出所有可能的拓扑有序序列。



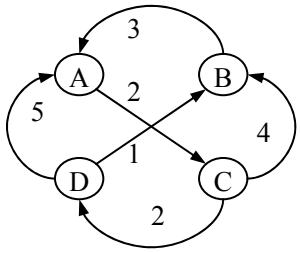
7.10 下面是某工程的 AOE 网，计算 1)整个工程完工需要多长时间(单位：天)？ 2)工程的关键路径。



7.11 用迪杰斯特拉算法计算下图中顶点 A 到其他顶点的最短路径。



7.12 用弗洛伊德算法计算下图中每对顶点之间的最短路径。


3
A
B
2
5
4
1
D
C
2

第9章 查找

一、基础知识和算法

1. 有关概念

查找表，静态查找表（只进行“查找”），**动态查找表**（可“查找”，“插入”，“删除”）。
关键字，平均查找长度

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

p_i 第 i 个关键字出现的概率，c_i 比较的关键字的个数。
静态查找表，**顺序查找表，折半查找表，静态树表，次优查找树，索引顺序表**。
动态查找表，**二叉排序树，平衡二叉树**（AVL 树），**B-树，B+树，键树，哈希表**。

2. 顺序查找

(1) 思路

按顺序逐个比较，直到找到或找不到。

(2) 算法

程序，要灵活应用。
例如：在数组 a 的前 n 个元素中查找 x

```
int Search ( int a[], int n, int x )
{
   for ( i=n-1; i>=0; i-- )
      if ( a[i]==x ) return i;
   return -1;   // -1 表示找不到
}
```

编程技巧：所有执行路径都要有正确的返回值，不要忘记最后那个 return 语句。
应试技巧：题目要求不明确时，按照方便的方法做，用适当的注释说明。

(3) 分析

顺序查找特点：思路简单(逐个比较)，适用面广(对查找表没有特殊要求)。

1°. 平均查找长度

一般在等概率情况下，查找成功时，平均查找长度

$$ASL = \frac{1+2+...+n}{n} = \frac{n+1}{2}$$

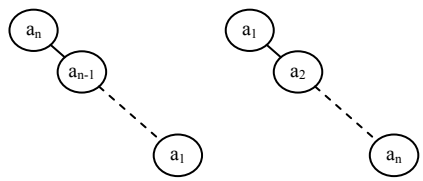
思路：假设对每个元素进行 1 次查找，共进行 n 次查找，计算出进行比较的关键字的个数，然后除以查找次数 n，就求得平均查找长度。

例：10 个元素的表等概率情况下查找成功时的平均查找长度

$$ASL = \frac{1+2+...+10}{10} = 5.5$$

2°. 判定树

判定树是一种描述查找中比较过程的直观形式，每个关键字所在层次就是其查找长度，有利于分析查找过程。顺序查找的判定树是一棵深度为 n 的单分支的树。课本上顺序查找从 a_n 开始，当然也可以从 a_1 开始。



3°. 时间复杂度

从平均查找长度看顺序查找的时间复杂度是 O(n)。

3. 折半查找

(1) 思路

待查找的表必须是有序的，先从中间开始比较，比较一次至少抛弃一半元素，逐渐缩小范围，直到查找成功或失败。

(2) 算法

要熟练掌握该算法。设 a[]升序有序，有以下算法：

```
int BinarySearch ( DataType a[], int n, DataType x )
{
   low = 0; high = n-1;
   while ( low <= high ) {
      mid = ( low + high )/2;      // 折半
      if ( a[mid]==x )
         return mid;   // 找到
      else if ( x<a[mid] )          // x 位于低半区  [low..mid-1]
         high = mid – 1;
      else                          // x 位于高半区  [mid+1..high]
         low = mid + 1;
```

```
    }
    return -1;   // -1 表示未找到
}
```

或者有递归版本：

```
int BinarySearch ( DataType a[], int low, int high, DataType x )
{
    if ( low>high )   return -1;       // 查找失败
    mid = (low+high)/2;              // 折半
    if ( a[mid]==x )
        return mid;   // 找到
    else if ( x<a[mid] )
        return BinarySearch (a, low, mid-1, x);
    else
        return BinarySearch (a, mid+1, high, x);
}
```

另外，程序可有多种写法。

例：a[]递减有序，折半查找，请填空。

```
int bs ( T a[], int n, T x )
{
    i = n-1; j = 0;
    while (____a____) {
        m = ____b____;
        if (____c____)
            return m;   // succeeded
        else if (____d____)
            i = ____e____;
        else
            ____f____;
    }
    return -1;   // not found
}
```

[25]

(3) 分析

特点：速度很快，要求查找表是有序的，而且随机访问(以便计算折半的下标)。所以，链表不能进行折半查找(但可以采用二叉排序树等形式进行快速的查找)。

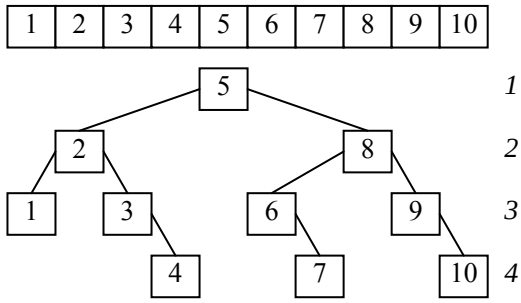
1°. 判定树

折半查找的判定树类似于完全二叉树，叶子结点所在层次之差最多为 1，其深度为 $\lfloor \log n \rfloor + 1$。

查找过程就是走了一条从根到该结点的路径。

如：表长 n=10 的有序表折半查找的判定树如下。

[25] 答案：a: j<=i b: (i+j)/2 c: a[m]==x d: x>a[m] e: m-1 f: j=m+1



2°. 平均查找长度

结论：等概率查找成功时的平均查找长度

$$ASL_{bs}=\frac{1}{n}\sum_{j=1}^{h}j2^{j-1}=\frac{n+1}{n}\log(n+1)-1\overset{n>50}{\approx}\log(n+1)-1$$

分析方法：对等概率情况，假设查找 n 次，且每个查找 1 次，共比较关键字 c 次，则平均 c/n 次。

例：表长为 n = 10，平均查找长度如下。

$$ASL=\frac{3+2+3+4+1+3+4+2+3+4}{10}=\frac{29}{10}=2.9$$

3°. 时间复杂度

结论：O(logn)，根据平均查找长度计算。

有时对需要反复查找的数据预先排序，再折半查找也是划算的。比如有 1000 个数据，顺序查找 100 次，平均比较约 100×500=50000 次；快排大约比较 1.44nlogn = 1.44×1000×10 = 14400，100 次折半查找比较不超过 100×9×2=1800 次(考虑到同一关键字的两次比较)，排序后折半查找合计比较不超过大约 16200 次。

4. 索引顺序表

分块，块间有序 + 块内无序，对应索引表有序 + 顺序表（无序）。

索引顺序表的查找性能介于顺序查找与折半查找之间。

分块的最佳长度是多少？规定条件：每块的大小相同，对块索引表和块内查找均采用顺序查找。

设表长为 n，等分成 b 块，采用顺序查找确定块需要比较(b+1)/2 次，块内顺序查找比较(n/b+1)/2 次，总共 C(b)=(b+1)/2+(n/b+1)/2，要使 C(b)最小，有 $b=\sqrt{n}$。

5. 二叉排序树

(1) 二叉排序树

二叉排序树或为空树；或者是这样一棵二叉树，若左子树不空，则左子树上所有结点

均小于根结点，若右子树不空，则右子树上所有结点均大于根结点，其左、右子树也是二叉排序树。
技巧：如果中序遍历二叉排序树，得到的结果将从小到大有序。手工判别二叉排序树的方法之一。

例：判断对错：二叉排序树或是空树，或是这样一棵二叉树，若左子树不空，则左孩子小于根结点，若右子树不空，则右孩子大于根结点，左、右子树也是这样的二叉排序树。（×）请自己举反例。

(2) 查找

思路：①若二叉树为空，则找不到，②先与根比较，相等则找到，否则若小于根则在左子树上继续查找，否则在右子树上继续查找。
递归算法：

```
BstTree BstSearch ( BstTree bst, DataType x )
{
   if ( bst==NULL )
      return NULL;
   else if ( bst->data==x )
      return bt;
   else if ( x<bst->data )
      return BstSearch ( bst->lchild, x);
   else
      return BstSearch ( bst->rchild, x);
}
```

非递归算法:

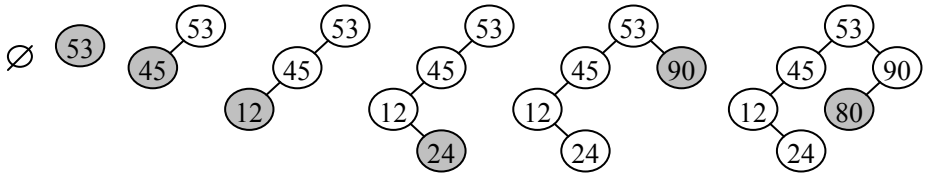
```
BstTree BstSearch ( BstTree bst, DataType x )
{
   p = bst;
   while ( p ) {
      if ( p->data==x )
         return p;
      else if ( x<p->data )
         p = p->lchild;
      else
         p = p->rchild;
   }
   return NULL;    // not found
}
```

(3) 插入

思路：先查找，若找不到则插入结点作为最后访问的叶子结点的孩子。
新插入的结点总是叶子。

(4) 建立

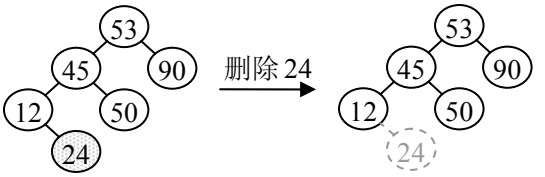
经过一系列插入操作可以建立二叉排序树。
给定关键字序列，建立二叉排序树。方法：①开始二叉树为空，②对每一个关键字，先进行查找，如果已存在，则不作任何处理，否则插入。
一句话，“从空树开始，每次插入一个关键字”。[26]
例：给定关键字序列{53，45，12，24，90，45，80}，建立二叉排序树。



(5) 删除

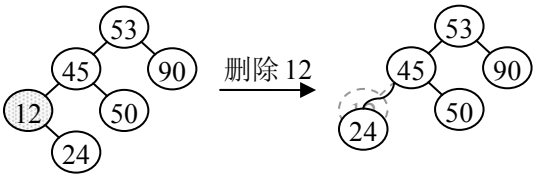
1°. 叶子

直接删除即可。



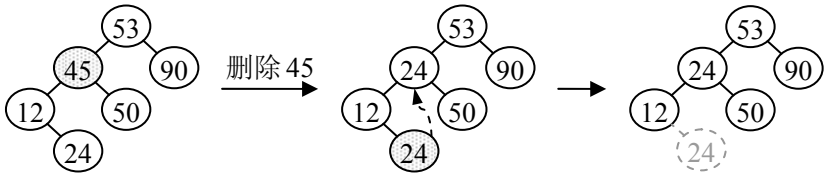
2°. 左子树或右子树为空

“移花接木”：将左子树或右子树接到双亲上。



3°. 左右子树都不空

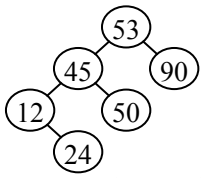
“偷梁换柱”：借左子树上最大的结点替换被删除的结点，然后删除左子树最大结点。（或者借用右子树上最小结点然后删除之亦可。）



[26] 不准确的说法，只为便于理解和记忆，不要在正式场合引用。

(6) 分析

判定树和二叉排序树相同。结点的层次等于查找时比较关键字的个数。



等概率情况下查找成功时

$$ASL = \frac{1}{n}\sum_{i=1}^{n} h_i = \frac{1+2+2+3+3+4}{6} = 2.5$$

若按照关键字有序的顺序插入结点建立二叉排序树，将得到一棵单支树，对其进行查找也退化为顺序查找，平均查找长度为(1+n)/2。一般地，如果在任一关键字 k 之后插入二叉排序树的关键字都大于或都小于 k，则该二叉排序树是单分支的，深度是 n，查找效率和顺序查找相同。

6. 平衡二叉树

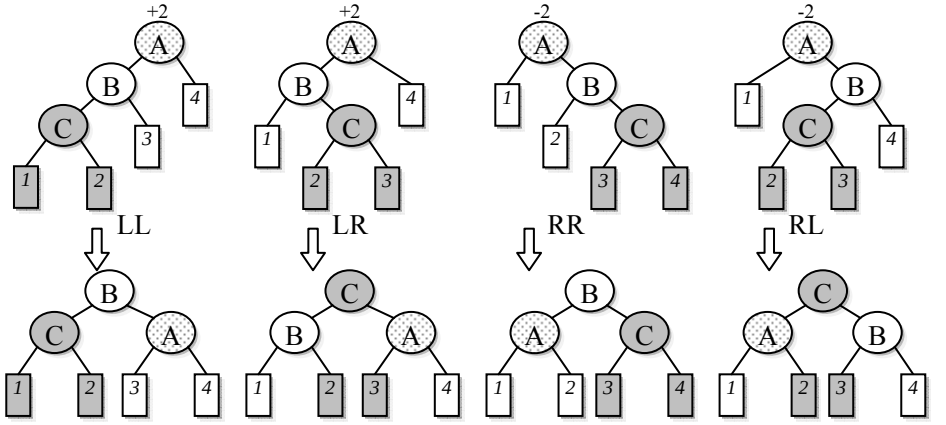
(1) 平衡因子和平衡二叉树(AVL树)

平衡因子：左子树深度 – 右子树深度。
平衡二叉树中各个结点的平衡因子只能是 0，1，-1。

(2) 构造平衡二叉排序树

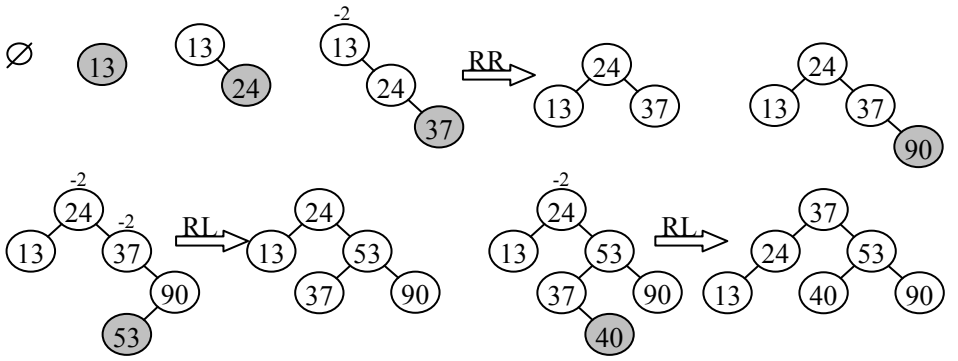
思路：按照建立二叉排序树的方法逐个插入结点，失去平衡时作调整。
失去平衡时的调整方法：[27]

1) 确定三个代表性结点。(A 是失去平衡的最小子树的根；B 是 A 的孩子；C 是 B 的孩子，也是新插入结点的子树)。关键是找到失去平衡的最小子树。
2) 根据三个代表性结点的相对位置(C 和 A 的相对位置)判断是哪种类型(LL, LR, RL, RR)。
3) 平衡化。“先摆好三个代表性结点(居中者为根)，再接好其余子树(根据大小)”。



[27] 这里没有使用课本上“旋转”的概念，这只是做题的方法，并不是正规的算法描述。

例：给定关键字的序列{13，24，37，90，53，40}，建立平衡二叉排序树。
注意：失去平衡时先确定失去平衡的最小子树，这是关键，然后判断类型（LL，LR，RL，RR），再平衡化处理。



(3) 分析

1°. 查找 (同二叉排序树)

2°. 平均查找长度ASL

结论：平均查找性能 O(log n)。
为求得 n 个结点的平衡二叉树的最大高度，考虑高度为 h 的平衡二叉树的最少结点数。

$$N_h = \begin{cases} 0, & h=0 \\ 1, & h=1 \\ N_{k-1}+N_{k-2}+1, & h\geq 2 \end{cases}$$

部分结果如下，F_h 表示斐波纳契数列第 h 项。

表 9.1 平衡二叉树的高度和结点个数

h	N_h	F_h	h	N_h	F_h
0	0	0	6	20	8
1	1	1	7	33	13
2	2	1	8	54	21
3	4	2	9	88	34
4	7	3	10	143	55
5	12	5	11	232	89

观察可以得出$N_h = F_{h+2} - 1, h\geqslant 0$。解得[28]

$$h = \log_{\varphi}(\sqrt{5}(n+1)) - 2 \approx 1.44 \log (n+1) - 0.328$$

其中 $\varphi = (\sqrt{5}+1)/2$。

[28] 解斐波纳契递推式代入得

$$N_h = n = \frac{1}{\sqrt{5}}\left[\left(\frac{\sqrt{5}+1}{2}\right)^{h+2} - \left(\frac{\sqrt{5}-1}{2}\right)^{h+2}\right] - 1 \xrightarrow{n\to\infty} \frac{1}{\sqrt{5}}\left(\frac{\sqrt{5}+1}{2}\right)^{h+2} - 1 = \frac{1}{\sqrt{5}}\varphi^{h+2} - 1, \text{其中 } \varphi = \frac{\sqrt{5}+1}{2}$$

然后整理，取对数可解得 h。

3°. 时间复杂度

一次查找经过根到某结点的路径，所以查找的时间复杂度是 O(log n)。

7. B-树和B^+树

(1) B-树

一棵 m 阶 B-树，或为空树，或满足：
(1) 每个结点至多有 m 棵子树；
(2) 若根结点不是叶子，则至少有两棵子树；
(3) 除根之外的所有非终端结点至少有$\lceil m/2 \rceil$棵子树；
(4) 所有非终端结点包含 n 个关键字和 n+1 棵子树：$(n, A_0, K_1, A_1, ..., K_n, A_n)$，其中关键字满足 $A_0<K_1<A_1<...<K_n<A_n$，关键字的个数$\lceil m/2 \rceil - 1 \le n \le m-1$。
(5) 所有叶子在同一层，不含信息，表示查找失败。

(2) B^+树

B+树和 B-树的差异：n 棵子树的结点中含有 n 个关键字；所有叶子结点中包含了全部关键字，且按大小顺序排列；所有非终端结点都是索引。
对 B+树既可以进行顺序查找又可以进行随机查找。

8. 键树

又叫**数字查找树**。
常见的两种存储结构：孩子兄弟链表，多重链表。

9. 哈希表

(1) 哈希表(散列表，杂凑表)

根据设定的哈希函数和处理冲突的方法，将一组关键字映像到一个有限的连续的地址集上，并以关键字在地址集中的象作为记录在表中的存储位置，这种表称为**哈希表**，又叫**散列表**，**杂凑表**。

(2) 哈希函数

常用**除留余数法**。H(key) = key MOD p。

(3) 冲突

什么是冲突？$H(key_1)=H(key_2)$，且 $key_1 \neq key_2$，称**冲突**。

处理冲突的方法：当 H(key)处已有记录，出现冲突，如何处理？

1°. 开放定址法

试用 $H(key) \oplus d_i$，常见以下三种。
(1) **线形探测再散列**：试用 $H(key) \oplus 1$，$H(key) \oplus 2$，...
(2) **二次探测再散列**：试用 $H(key) \oplus 1^2$，$H(key) \oplus -1^2$，$H(key) \oplus 2^2$，$H(key) \oplus -2^2$，...
(3) **伪随机探测再散列**：试用 $H(key) \oplus f(1)$，$H(key) \oplus f(2)$，...

2°. 再哈希法

$H_1(key)$冲突，试用 $H_2(key)$，$H_3(key)$，...

3°. 链地址法

发生冲突的记录链成单链表。

4°. 建立公共溢出区

所有冲突记录存入溢出区。

(4) 装填因子

$$\alpha = \frac{n}{m}$$

n 个记录，m 个地址空间。
哈希表的平均查找长度与记录个数 n 不直接相关，而是取决于装填因子和处理冲突的方法。

(5) 举例

例：已知一组关键字{19, 14, 23, 1, 68, 20, 85, 9}，采用哈希函数 H(key)=key MOD 11，请分别采用以下处理冲突的方法构造哈希表，并求各自的平均查找长度。

1) 采用线性探测再散列；
2) 采用伪随机探测再散列，伪随机函数为 f(n) = - n；
3) 采用链地址法。

思路：开始时表为空，依次插入关键字建立哈希表。

1) 线性探测再散列

0	1	2	3	4	5	6	7	8	9	10
9	23	1	14	68				19	20	85
3	*1*	*2*	*1*	*3*				*1*	*1*	*3*

key	19	14	23	1	68	20	85	9	← 关键字
H(key)	*8*	*3*	*1*	*1*	*2*	*9*	*8*	*9*	← H(key)
				2	*3*		*9*	*10*	← 冲突时计算下一地址
					4		*10*	*0*	←
查找长度	1	1	1	2	3	1	3	3	← 每个关键字的查找长度

ASL = (1+1+1+2+3+1+3+3)/8 = 15/8

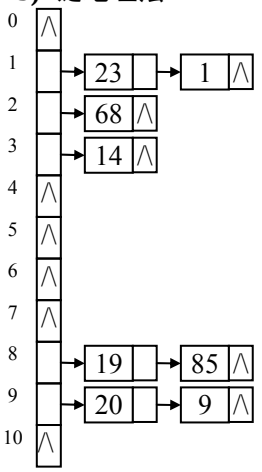
2) 伪随机探测再散列

0	1	2	3	4	5	6	7	8	9	10
1	23	68	14			9	85	19	20	
2	*1*	*1*	*1*			*4*	*2*	*1*	*1*	

key	19	14	23	1	68	20	85	9
H(key)	*8*	*3*	*1*	*1*	*2*	*9*	*8*	*9*
				0			*7*	*8*
								7
								6
查找长度	1	1	1	2	1	1	2	4

ASL = (1+1+1+2+1+1+2+4)/8 = 13/8

3) 链地址法



ASL = (1×5+2×3)/8 = 11/8

注：关键字在链表中的插入位置可以在表头或表尾，也可以在中间以便保持关键字有序。

最后，此哈希表的装填因子是 α = 8/11。

二、习题

9.1 对 n=10 个元素的表顺序查找，在等概率情况下(对每个元素查找的概率相同)，查找成功时的平均查找长度是多少？如果查找失败的概率为 0.2，平均查找长度又是多少？

9.2 表长 15 的有序表进行折半查找，计算等概率情况下查找成功时的平均查找长度和查找第 3 个元素需要比较关键字的个数。

9.3 对表长 1023 的有序表折半查找，比较 8 个关键字才能找到的元素有多少个？至多比较 8 次就能找到的元素有多少个？

9.4 长度为 10000 的表进行分块查找，用顺序查找确定所在块，块内元素无序，为使平均查找长度最小应该分几块？

9.5 编写算法判断给定的二叉树是否是二叉排序树。

9.6 给定一组关键字(13，24，37，90，53，40，20)，建立二叉排序树。

9.7 给定一组关键字(13，24，37，90，53，40，20)，建立平衡二叉排序树。

9.8 编写平衡二叉排序树的查找算法。

9.9 具有 20 个结点的二叉排序树的最大深度是(___)，最小深度是(___)；20 个结点的平衡二叉排序树最大深度是(___)。

9.10 在一棵非空的 5 阶 B-树中，非叶子结点中最少有(___)棵子树，最多有(___)个关键字；除根以外的非终端结点中最少有(___)棵子树。

9.11 已知一组关键字{19, 14, 23, 1, 68, 20, 85, 9}，采用哈希函数 H(key)=key MOD 13，请分别采用以下处理冲突的方法构造哈希表，并求各自的平均查找长度。

1) 采用线性探测再散列；
2) 采用伪随机探测再散列，伪随机函数为 $f(n) = n^2+2n+3$；
3) 采用链地址法。

第10章 内部排序

一、基础知识和算法

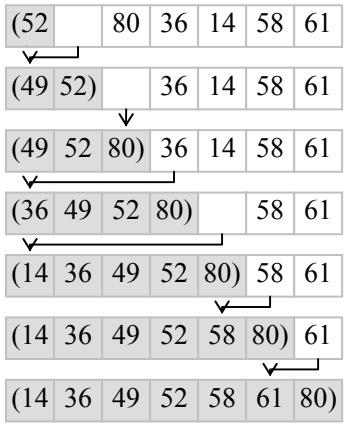
1. 排序的有关概念

排序（按关键字大小顺序排列数据）。
排序方法：**内部排序，外部排序**；简单的排序方法 $O(n^2)$，先进的排序方法 O(nlogn)，**基数排序** O(dn)；**插入排序，交换排序，选择排序，归并排序，计数排序**。
排序方法的稳定性：取决于该方法采取的策略，不是由一次具体的排序结果决定的。但是通过列举不稳定的排序实例可以说明该排序算法的不稳定性。

2. 直接插入排序

(1) 思路

将待排序记录插入已排好的记录中，不断扩大有序序列
一句话，“将待排序记录插入有序序列，重复n-1 次”[29]。
例：52，49，80，36，14，58，61 进行直接插入排序。



[29] 不准确的说法，只为便于理解和记忆，不要在正式场合引用。

(2) 分析

表 10.1 直接插入排序

	比较	移动	
记录顺序有序时	n-1	0	最好
记录逆序有序时	((n+2)(n-1))/2	((n+4)(n-1))/2	最坏

平均 $n^2/4$，算法的时间复杂度 $O(n^2)$。直接插入排序是稳定的排序算法。

3. 折半插入排序

(1) 思路

在直接插入排序中，查找插入位置时采用折半查找的方法。

(2) 程序

```
void BinInsertSort ( T a[], int n )
{
  for ( i=1; i<n; i++ ) {
    // 在 a[0..i-1]中折半查找插入位置使 a[high]≤a[i]<a[high+1..i-1]
    low = 0;   high = i-1;
    while ( low<=high ) {
      m = ( low+high )/2;
      if ( a[i]<a[m] )
        high = m-1;
      else
        low = m+1;
    }
    // 向后移动元素 a[high+1..i-1]，在 a[high+1]处插入 a[i]
    x = a[i];
    for ( j=i-1; j>high; j-- )
      a[j+1] = a[j];
    a [high+1] = x;              // 完成插入
  }
}
```

(3) 分析

时间复杂度 $O(n^2)$。比直接插入排序减少了比较次数。折半插入排序是稳定的排序算法。

4. 希尔排序（缩小增量排序）

(1) 思路

先将待排序列分割成若干个子序列，分别进行直接插入排序，基本有序后再对整个序列进行直接插入排序。

步骤：

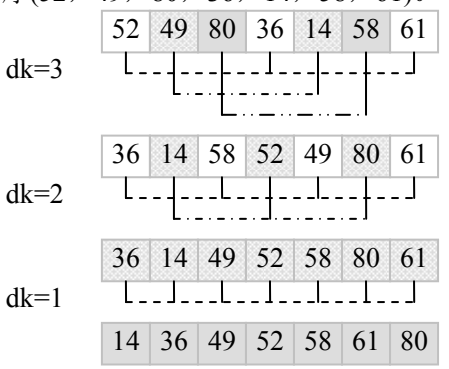
1^0. 分成子序列(按照增量 dk)；

2^0. 对子序列排序(直接插入排序)；

3^0. 缩小增量，重复以上步骤，直到增量 dk=1。

增量序列中最后一个增量一定是 1，如：... 9, 5, 3, 2, 1 和... 13, 4, 1。如没有明确说明增量序列可以选择... 3, 2, 1 或... 5, 3, 2, 1。

例：希尔排序(52，49，80，36，14，58，61)。



注意：希尔排序是不稳定的。时间复杂度大约为 $O(n^{3/2})$。

(2) 程序

```
void ShellSort ( T a[], int n )
{
   dk = n/2;
   while ( dk>=1 ) {
      // 一趟希尔排序，对 dk 个序列分别进行插入排序
      for ( i=dk; i<n; i++ ) {
         x = a[i];
         for ( j=i-dk; j>=0 and x<a[j]; j-=dk )
            a[j+dk] = a[j];
         a[j+dk] = x;
      }
      // 缩小增量
      dk = dk/2;
   }
}
```

5. 起泡排序

(1) 思路

一句话，“依次比较相邻元素，‘逆序’则交换，重复n-1 次”[30]。
例：冒泡排序(52，49，80，36，14，58，61)。

52	49	80	36	14	58	61
49	52	36	14	58	61	80
49	36	14	52	58	61	80
36	14	49	52	58	61	80
14	36	49	52	58	61	80
14	36	49	52	58	61	80
14	36	49	52	58	61	80

(2) 程序

请参考 第 1 章 二、BubbleSort算法。

(3) 分析

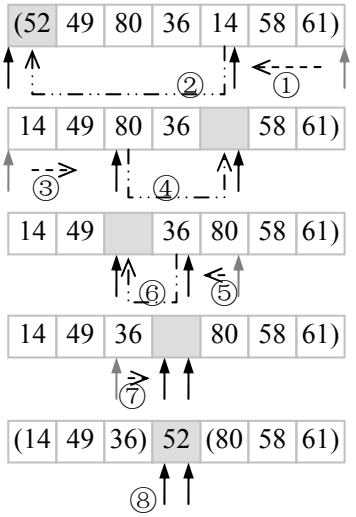
比较和交换总是发生在相邻元素之间，是稳定的排序算法。时间复杂度 $O(n^2)$。

6. 快速排序

(1) 思路

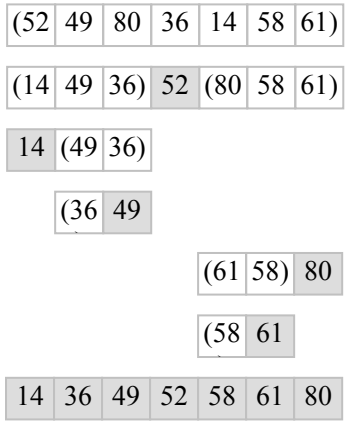
一趟排序把记录分割成独立的两部分，一部分关键字均比另一部分小，然后再分别对两部分快排。
例：{52 49 80 36 14 58 61} 快速排序。
下面是一次划分的详细步骤：

[30] 不准确的说法，只为便于理解和记忆，不要在正式场合引用。



技巧：
选第 1 个记录为轴，分别从后向前，从前向后扫描记录，后面“抓大放小”(如：①②)，前面“抓小放大”(如：③④)，交替进行(⑤-⑦)，最后将轴记录放在中间(⑧)，划分

整个快速排序过程如下：



(2) 程序

```
void QuickSort ( T a[], int low, int high )
{
   if ( low < high ) {
      // 划分
      pivot = a[low];
      i = low; j = high;
      while ( i < j ) {
         while ( i<j && a[j] >= pivot )    j--;
         a[i] = a[j];
         while ( i<j && a[i] <= pivot )    i++;
         a[j] = a[i];
      }
      a[i] = pivot;
```

```
        // 对子序列快排
        QuickSort ( a, low, i-1);
        QuickSort ( a, i+1, high);
    }
}
```

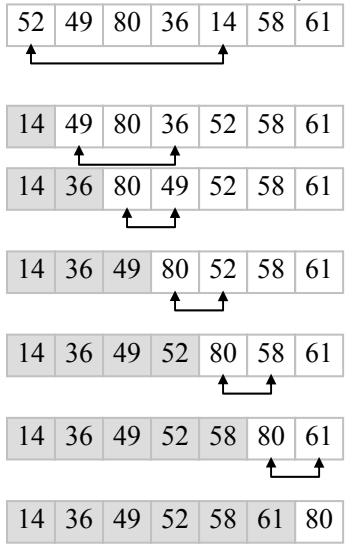
(3) 分析

平均情况下，时间复杂度 O(nlogn)。记录本来有序时为最坏情况，时间复杂度为 $O(n^2)$。空间复杂度(考虑递归调用的最大深度)在平均情况下为 O(logn)，在最坏情况下为 O(n)。
快速排序是不稳定的。

7. 简单选择排序

(1) 思路

第 i 趟排序过程是在剩余的待排记录中选一个最小（大）的，放在第 i 个位置。
一句话，“在待排记录中选取最小的，交换到合适位置，重复n-1 次”[31]。
例：{52 49 80 36 14 58 61}简单选择排序。



(2) 程序

```
void SelectionSort ( T a[], int n )
{
    for ( i=0; i<n-1; i++ ) {
        k = i;
```

[31] 不准确的说法，只为便于理解和记忆，不要在正式场合引用。

```
    for ( j=i+1; j<n; j++)
       if ( a[j]<a[k] )   k=j;   // 最小记录
    if ( k!=i )   a[i]←→a[k];
  }
}
```

(3) 分析

时间复杂度 $O(n^2)$，耗费在比较记录上，比较次数始终为 n(n-1)/2，移动次数最小为 0，最大 3(n-1)，即 n-1 次交换。
注意：简单选择排序是不稳定的。反例：$(10_1, 10_2, 9) \rightarrow (9, 10_2, 10_1)$。

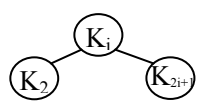
8. 堆排序

(1) 堆及其特点

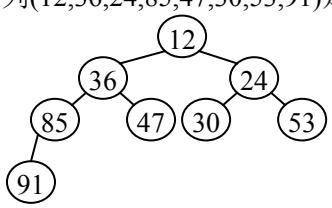
堆，小顶堆，大顶堆。
序列$\{K_1,K_2, ..., K_n\}$满足 $K_i \leqslant K_{2i}$，$K_i \leqslant K_{2i+1}$，称为小顶堆；若满足 $K_i \geqslant K_{2i}$，$K_i \geqslant K_{2i+1}$，称为大顶堆，其中 i=1, 2, ..., n/2。
特点：小顶堆的堆顶(第一个元素)为最小元素，大顶堆的堆顶为最大元素。

(2) 判断序列是否构成堆

方法：用 K_i 作为编号为 i 的结点，画一棵完全二叉树，比较双亲和孩子容易判断是否构成堆。



例：判断序列(12,36,24,85,47,30,53,91)是否构成堆。

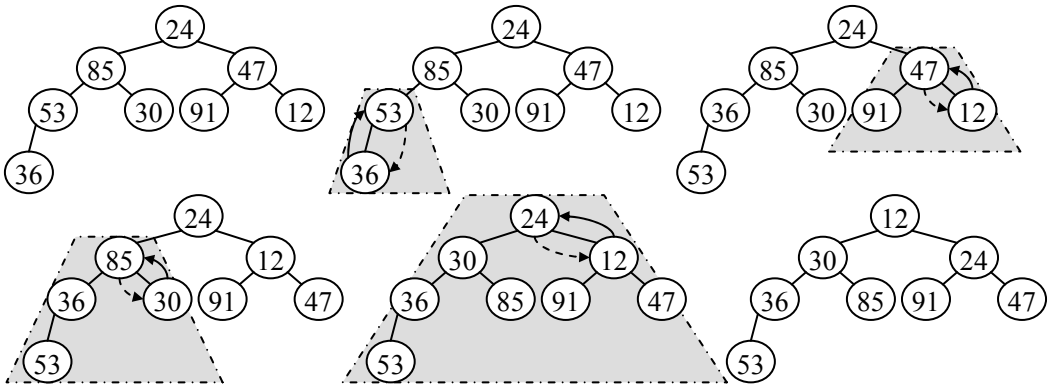


根据上图判断，该序列构成小顶堆。

(3) 建立堆

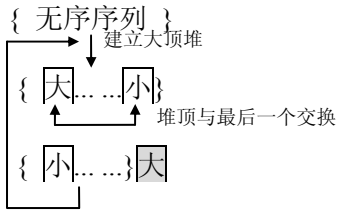
一句话，“‘小堆’变‘大堆’，从$\lfloor n/2 \rfloor$变到 1”[32]。第$\lfloor n/2 \rfloor$个是最后一个分支结点。
例：把(24,85,47,53,30,91,12,36)调整成小顶堆。

[32] 不准确的说法，只为便于理解和记忆，不要在正式场合引用。

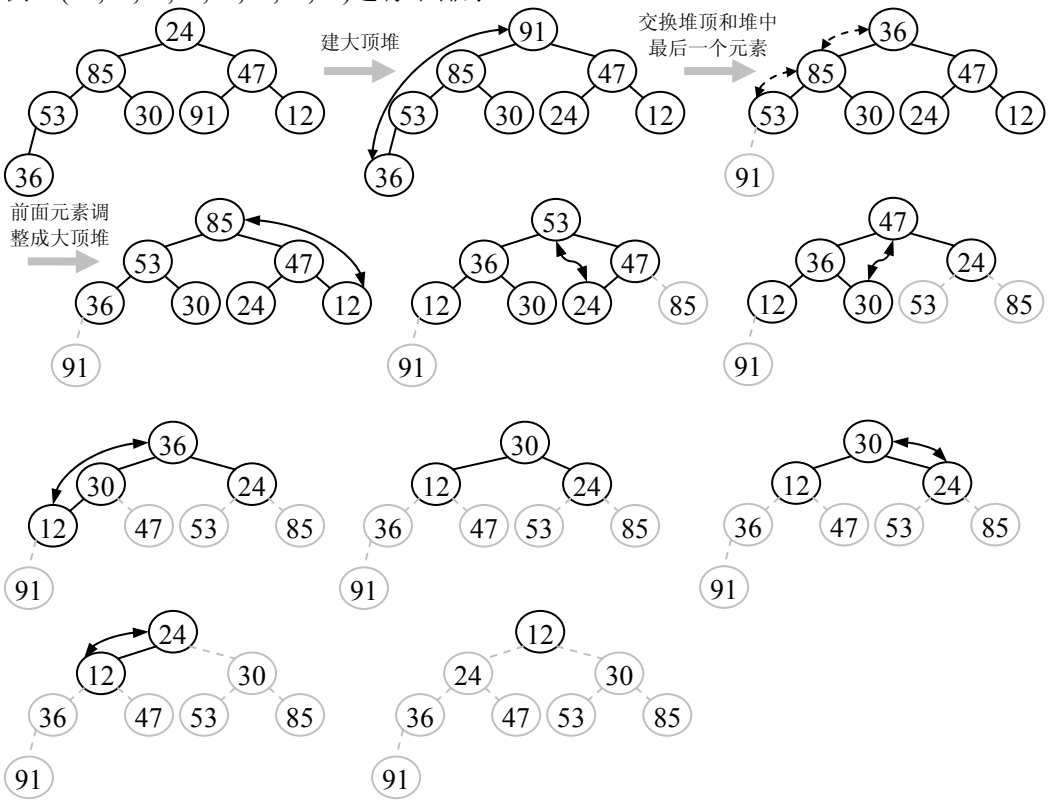


(4) 堆排序

思路：



例：(24,85,47,53,30,91,12,36)进行堆排序。



整个排序步骤如下：

24	85	47	53	30	91	12	36
91	85	47	53	30	24	12	36
85	53	47	36	30	24	12	91
53	36	47	12	30	24	85	91
47	36	24	12	30	53	85	91
36	30	24	12	47	53	85	91
30	12	24	36	47	53	85	91
24	12	30	36	47	53	85	91
12	24	30	36	47	53	85	91

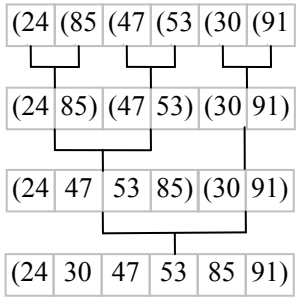
堆排序是不稳定的。时间复杂度是 O(nlogn)。

9. 归并排序

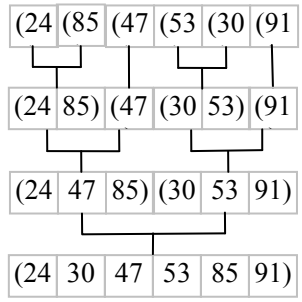
(1) 思路

归并，两个或多个有序表合并成一个有序表。**归并排序**。

例：对{24, 85, 47, 53, 30, 91}归并排序。



自底向上归并排序



归并排序

(2) 程序

归并排序：

```
void MergeSort ( T a[], int low, int high )
{
   if ( low>=high )    return;
   else {
```

```
    mid = (low+high)/2;
    MergeSort ( a, low, mid );
    MergeSort ( a, mid+1, high );
    Merge ( a, low, mid, high );
  }
}
```

自底向上的归并排序：

```
void MergeSort ( T a[], int n )
{
  t = 1;
  while ( t<n ) {
    s = t;   t = s*2;
    for ( i=0; i+t<=n; i+=t )
      Merge ( a, i, i+s-1, i+t-1 );
    if ( i+s<n )
      Merge ( a, i, i+s-1, n-1 );
  }
}
```

附：Merge()，将有序序列 a[low..mid]和 a[mid+1..high]归并到 a[low..high]。

```
void Merge ( T a[], int low, int mid, int high )
{
  // 归并到 b[]
  i = low;   j = mid+1;   k = low;
  while ( i<=mid and j<=high ) {
    if ( a[i]<=a[j] ) { b[k] = a[i];   i++; }
    else   { b[k] = a[j];   j++; }
    k++;
  }
  // 归并剩余元素
  while ( i<=mid )   b[k++] = a[i++];
  while ( j<=high )   b[k++] = a[j++];
  // 从 b[]复制回 a[]
  a[low..high] = b[low..high];
}
```

(3) 分析

时间复杂度 O(nlogn)。需要空间多，空间复杂度 O(n)。归并排序是稳定的排序。

10. 基数排序

(1) 思路

1°. 多关键字排序

最高位优先（MSD），**最低位优先**（LSD）。

2°. 链式基数排序

链式基数排序采用“分配”和“收集”策略。
例：{503，087，512，061，908，170，897，275，653}
分析：数字 0~9 共 10 种情况，rd=10。
每个关键字都有 3 位数字，d=3。
共有 9 个记录，n=9。
先“收集”成一个链表，
按最低位（个位）“分配”到 10 个链表中（0 号~9 号）：

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
170	061	512	503		275		087	908	
			653				897		

按个位顺序“收集”成一个链表：
(170, 061, 512, 503, 653, 275, 087, 897, 908)
再按第 2 位数字（十位）“分配”到 10 个链表中：

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
503	512				653	061	170		897
908							275		

“收集”成一个链表：
(503, 908, 512, 653, 061, 170, 275, 897)
按第 3 位数字（百位）“分配”到 10 个链表中：

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
061	170	275			503	653		897	908
					512				

“收集”成一个链表：
(061, 170, 275, 503, 512, 653, 897, 908)
完成排序。

(2) 分析

对 n 个数据进行基数排序，每个数据基数为 rd，有 d 位数字。那么，一趟分配和收集用时 n+rd(分配用 n，收集用 rd)，共需 d 趟，总的时间复杂度为 O(d(n+rd))。
基数排序是稳定的排序算法。

11. 各种排序方法比较

表 10.2 排序方法的比较

排序方法	时间复杂性			空间复杂性	稳定	特点
	平均	最好	最坏			
简单插入排序	$O(n^2)$	O(n)	$O(n^2)$	O(1)	是	元素少或基本有序时高效
希尔排序	$O(n^{3/2})$			O(1)	否	
冒泡排序	$O(n^2)$	O(n)	$O(n^2)$	O(1)	是	
快速排序	O(nlogn)	O(nlogn)	$O(n^2)$	O(logn)	否	平均时间性能最好
简单选择排序	$O(n^2)$			O(1)	否	比较次数最多
堆排序	O(nlogn)			O(1)	否	辅助空间少
归并排序	O(nlogn)			O(n)	是	稳定的
基数排序	O(d(n+rd))			O(rd)	是	适合个数多关键字较小

基于关键字比较的排序方法，在最坏情况下所能达到的最好的时间复杂度是O(nlogn)。

二、习题

10.1 对待排序列(24, 86, 48, 56, 72, 36)进行 1) 直接插入排序，2) 希尔排序，3) 起泡排序，4) 快速排序，5) 简单选择排序，6) 堆排序，7) 归并排序，8) 链式基数排序。

10.2 证明：借助比较进行的排序方法，在最坏情况下所能达到的最好的时间复杂度是O(nlogn)。

附录A: 习题解答

1.1 编写冒泡排序算法，使结果从大到小排列。

```
void BubbleSortDec ( DataType a[], int n )
{
  for ( i=0; i<n-1; i++ ) {
     change = false;
     for( j=0; j<n-i-1; j++ )
        if ( a[j]<a[j+1] ) {          // 逆序
           swap( a[j], a[j+1] );
           change = true;
        }
     if ( not change ) break;        // 没有交换则完成排序
  }
}
```

1.2 计算下面语句段中指定语句的频度：

```
1)  for ( i=1; i<=n; i++ )
        for ( j=i; j<=n; j++ )
            x++;  // @
2)  i = 1;
    while ( i<=n )
        i = i*2;  // @
```

分析：计算语句频度是计算时间复杂度的基础。可以用观察和归纳进行简单的计算。

1)

问题规模 n	执行次数
1	1
2	2+1
3	3+2+1
...	...
n	n+...+2+1=n(n+1)/2

结论：语句频度为 n(n+1)/2。

2)

问题规模 n	执行次数
1	1
2	2
3	2
4	3
...	...
2^k	k+1

结论：语句频度为$\lfloor \log n \rfloor + 1$。

2.1 将顺序表中的元素反转顺序。

```
void Reverse ( SqList& L )
```

```
{
   for ( i=0; i<L.length/2; i++)          // ㊀ < 还是 <= [33]
      L.elem[i]←→L.elem[L.length-i-1];
}
```

2.2 在非递减有序的顺序表中插入元素x，并保持有序。

思路 1：先查找适合插入的位置 i
　　向后移动元素（从后向前处理）
　　插入元素，表长加 1

思路 2：从后向前查找插入位置，同时向后移动大于 x 的元素
　　插入元素，表长加 1

注意：表满时不能插入。

```
// 顺序表结构
const int MAXSIZE = 1024;
typedef struct {
      DataType elem[MAXSIZE];
      int length;
} SqList;

// 向非递减有序的顺序表 L 中插入元素 x，仍保持非递减有序
// 插入成功返回 true，否则返回 false
bool OrderListInsert ( SqList &L, DataType x )
{
   if ( L.length==MAXSIZE )   return false;   // 表满，插入失败
   // 将大于 x 的元素后移
   for ( i=L.length-1; i>=0 && L.elem[i]>x; i--)
      L.elem[i+1] = L.elem[i];
   // 插入 x (因为最后执行 i--，故应在 i+1 处)
   L.elem[i+1] = x;
   L.length++;
   return true;
}
```

2.3 删除顺序表中所有等于x的元素。

```
void Remove ( SqList &L, DataType x )
{
   i = 0;                              // 剩余元素个数，同时是下一个元素的插入位置
   for ( j=0; j<L.length; j++ )
      if ( L.elem[j]!=x ) {            // 复制不等于 x 的元素组成新表
         if ( i!=j )   L.elem[i] = L.elem[j];   // 当 i==j 时不必复制
         i++;
      }
   L.length = i;                       // 剩余元素个数
}
```

[33] 用边界值验证法说明对于偶数个元素的表会有何种情况。

本算法的时间复杂度为 O(n)；若改用反复调用查找和删除算法，时间复杂度会达到 $O(n^2)$。

2.4 编写算法实现顺序表元素唯一化(即使顺序表中重复的元素只保留一个)，给出算法的时间复杂度。
思路：设已经唯一化的序列是$(a_0, \ldots, a_{i-1})$，剩余序列是$(a_j, \ldots, a_n)$。所要做的就是在已经唯一化的序列 L.elem[0..i-1]中查找 L.elem[j]，如果未找到则复制到 L.elem[i]处。如此重复直到剩余序列为空。

```
void Unique ( SqList &L )
{
  if ( L.length<=1 )   return;          // 空表或只有一个元素的表已经唯一化了
  i = 1;                                // 开始 L.elem[0..0]是唯一化序列
  for ( j=1; j<L.length; j++ ) {
    // 在 L.elem[0..i-1]中查找 L.elem[j]
    for ( k=0; k<i; k++ )
      if ( L.elem[k]==L.elem[j] )   break;
    if ( k==i ) {                       // L.elem[j]未出现过，复制到 L.elem[i]处
      if ( j!=i )   L.elem[i] = L.elem[j];
      i++;
    }
  }
  L.length = i;   // 表长为 i
}
```

以上算法的时间复杂度为 $O(n^2)$。当然，可以反复将重复元素删除达到唯一化，时间复杂度仍是 $O(n^2)$，但是与以上算法相比要移动更多元素。

2.5 非递减有序的顺序表元素唯一化(参见习题 2.4)，要求算法的时间复杂度为O(n)。
分析：由于该表是有序的，相等的元素必然靠在一起，不必从头开始查找，所以算法的时间复杂度可以降低。
思路：类似习题 2.4，但是查找部分只要与L.elem[i-1]比较就可以了。

```
void Unique ( SqList &L )
{
  i = 0;   // 开始的唯一化序列为空(㊀对比习题 2.4 思考为什么不用i=1 开始 [34])
  for ( j=1; j<L.length; j++ )
    if ( L.elem[j]!=L.elem[i-1] ) {      // Note：写成 L.elem[j]!=L.elem[j-1]亦可
      if ( j!=i )   L.elem[i] = L.elem[j];
      i++;
    }
  L.length = i;   // 表长
}
```

[34] 原因是这里不能确定表是否为空，而习题 2.4 则用开始的 if 语句排除了空表的情况。事实上，习题 2.4 也可以仿照此处修改，请读者自己完成。

2.6 将单链表就地逆置，即不另外开辟结点空间，而将链表元素翻转顺序。
思路：从链上依次取下结点，按照逆序建表的方法(参见 2.3. (8) 2°. 节)重新建表。

```
void Reverse ( LinkList &L )
{
   p = L->next;                    // 原链表
   L->next = NULL;                 // 新表(空表)
   while ( p ) {
      // 从原链表中取下结点 s
      s = p;
      p = p->next;
      // 插入 L 新表表头
      s->next = L->next;
      L->next = s;
   }
}
```

2.7 采用插入法将单链表中的元素排序。

```
void InsertionSort ( LinkList &L )
{
   h = L->next;                    // 原链表
   L->next = NULL;                 // 新空表
   while ( h ) {
      // 从原链表中取下结点 s
      s = h;   h = h->next;
      // 在新表中查找插入位置
      p = L;
      while ( p->next && p->next->data<=s->data )
         p = p->next;
      // 在 p 之后插入 s
      s->next = p->next;
      p->next = s;
   }
}
```

2.8 采用选择法将单链表中的元素排序。

```
void SelectionSort ( LinkList &L )
{
   p = L;
   while ( p->next ) {
      // 选择最小(从 p->next 至表尾)
      q = p;                       // 最小元素的前驱 q
      s = p;
      while ( s->next ) {
         if ( s->next->data < q->next->data )   q = s;
         s = s->next;
      }
      m = q->next;                 // 找到最小 m
```

```
      // 最小元素 m 插入有序序列末尾(p 之后)
      if ( q!=p ) {
         q->next = m->next;    // 解下最小 m
         m->next = p->next;    // 插入 p 之后
         p->next = m;
      }
      p = p->next;                  // L->next 至 p 为有序序列
   }
}
```

2.9 将两个非递减有序的单链表归并成一个，仍并保持非递减有序。

```
// 将非递减有序的单链表 lb 合并入 la，保持非递减有序
// 结果 la 中含有两个链表中所有元素，lb 为空表
void Merge ( LinkList &la, LinkList &lb )
{
   p = la,   q = lb;
   while ( p->next and q->next ) {
      // 跳过 la 中较小的元素
      while ( p->next and (p->next->data <= q->next->data) )
         p = p->next;
      // 把 lb 中较小的元素插入 la 中
      while ( p->next and q->next and (q->next->data < p->next->data) ) {
         s = q->next;
         q->next = s->next;
         s->next = p->next;
         p->next = s;
         p = s;
      }
   }
   if ( lb->next ) {       // 表 lb 剩余部分插入 la 末尾
      p->next = lb->next;
      lb->next = NULL;
   }
}
```

3.1 元素 1,2,3,4 依次入栈，不可能的出栈序列有哪些？

分析：什么是不可能的出栈序列？如果后入栈的数(如 4)先出栈，则此前入栈元素(如 1,2,3)在栈中的顺序就确定了，它们的出栈顺序一定是逆序(如 3,2,1)，否则就是不可能的出栈序列(如 2,1,3)。

不可能的出栈序列有：4123，4132，4213，4231，4312，3412，3142，3124。其中后 3 种都含 312 这一不可能序列。

3.2 设循环队列Q少用一个元素区分队列空和队列满，MAXSIZE=5，Q.front=Q.rear=0，画出执行下列操作时队列空和队列满的状态。入队列a,b,c，出队列a,b,c，入队列d,e,f,g。[fa][r][][][]→[fa][b][r][][]→[fa][b][c][r][]→[][fb][c][r][]→[][][fc][r][]→ [][][][fr][] 队列空 → ... →[f][r][][fd][e]→[f][g][r][fd][e] 队列满。

3.3 编写算法利用栈将队列中的元素翻转顺序。
思路：先将队列中的元素入栈，然后从栈中取出重新入队列。

```
void Reverse ( SqQueue &Q )
{
  InitStack ( S );
  while ( ! QueueEmpty(Q) ) {
    DeQueue ( Q, x ); Push ( S, x );
  }
  while ( ! StackEmpty(S) ) {
    Pop ( S, x ); EnQueue ( Q, x );
  }
}
```

4.1 长度为n的串的子串最多有多少个？
思路：对子串长度归纳。
子串的长度是 0，1，2，...，n，对应子串的个数分别是 1(空串)，n，n-1，...，1，总起来就是 1+n+(n-1)+...+2+1=1+n(n+1)/2。

6.1 度为k的树中有n_1个度为 1 的结点，n_2个度为 2 的结点，…，n_k个度为k的结点，问该树中有多少个叶子结点。
分析：分别从结点个数和分支个数考虑。
设叶子个数为 n0，结点总数：$n = n_0+n_1+n_2+...+n_k$，分支数目：$n-1 = n_1+2\ n_2+...+k\ n_k$，于是得到叶子个数

$$n_0 = 1 + \sum_{i=1}^{k} (i-1) n_i$$

6.2 有n个叶子结点的完全二叉树的高度是多少?
分析：完全二叉树中度为 1 的结点至多有一个。
完全二叉树中的结点数 n+(n-1) ≤ N ≤ n + (n-1) + 1，即 2n-1 ≤ N ≤ 2n，二叉树的高度是

$$\lfloor \log(2n-1) \rfloor + 1 \le h \le \lfloor \log(2n) \rfloor + 1$$

于是，(1) 当 $n=2^k$ 时，$h = \lfloor \log n \rfloor + 1$，当没有度为 1 的结点时；$h = \lfloor \log n \rfloor + 2$，当有 1 个度为 1 的结点时。(2) 其他情况下，$h = \lfloor \log n \rfloor + 2$。

6.3 编写算法按照缩进形式打印二叉树。

```
void PrintBinaryTree ( BinTree bt, int indent )
{
  if ( ! bt )   return;
  for ( i=0; i<indent; i++ )   print ( "  ");   // 缩进
  print ( bt->data );
  PrintBinaryTree ( bt->lchild, indent+1 );
  PrintBinaryTree ( bt->rchild, indent+1 );
}
```

6.4 编写算法按照逆时针旋转 90 度的形式打印二叉树。

```
void PrintBinaryTree ( BinTree bt, int level )
{
   if ( ! bt )    return;
   PrintBinaryTree ( bt->rchild, level+1 );   // 旋转后先打印右子树
   for ( i=0; i<level; i++ )    print ( "    " );   // 缩进
   print ( bt->data );
   PrintBinaryTree ( bt->lchild, level+1 );
}
```

6.5 编写算法判断二叉树是否是完全二叉树。

分析：按层遍历完全二叉树，当遇到第一个空指针之后应该全都是空指针。

```
bool IsComplete ( BinTree bt )
{
   // 按层遍历至第一个空指针
   InitQueue ( Q );
   EnQueue ( Q, bt );
   while ( ! QueueEmpty(Q) ) {
      DeQueue ( Q, p );
      if ( p ) {
         EnQueue ( Q, p->lchild );
         EnQueue ( Q, p->rchild );
      } else
         break;   // 遇到第一个空指针时停止遍历
   }
   // 检查队列中剩余元素是否全部是空指针
   while ( ! QueueEmpty(Q) ) {
      DeQueue ( Q, p );
      if ( ! p )    return false;   // 不是完全二叉树
   }
   return true;   // 完全二叉树
}
```

6.6 编写算法求二叉树中给定结点的所有祖先。

分析：进行后序遍历时，栈中保存的是当前结点的所有祖先。所以，后序遍历二叉树，遇到该结点时，取出栈中的内容即是所有祖先。

```
// 求二叉树 bt 中结点 xptr 的所有祖先
vector Ancestors ( BinTree bt, BinTree xptr )
{
   stack s;    stack tag;
   p = bt;
   while ( p || ! s.empty() ) {
      if ( p ) {
         s.push ( p );    tag.push ( 1 );
         p = p->lchild;
      } else {
         p = s.pop();    f = tag.pop();
```

```
        if ( f==1 ) {
            s.push ( p );    tag.push ( 2 );
            p = p->rchild;
        } else {
            if ( p==xptr ) {
                v = s;   // 当前栈的内容就是 xptr 的所有祖先
                return v;
            }
            p = NULL;
        }
    }
  } // while
  return vector();    // return a null vector
}
```

注：这里为描述方便借助了 C++中的某些描述方式。

6.7 编写算法求二叉树中两个结点的最近的共同祖先。

思路：用后序遍历求出两者的所有祖先，依次比较。

```
// 求二叉树 bt 中两个结点 q 和 r 的最近的共同祖先
BinTree LastAncestor ( BinTree bt, BinTree q, BinTree r )
{
  stack sq, sr;
  stack s;    stack tag;
  // 求 q 和 r 的所有祖先
  p = bt;
  while ( p || ! s.empty() ) {
    if ( p ) {
        s.push ( p );    tag.push ( 1 );
        p = p->lchild;
    } else {
        p = s.pop();    f = tag.pop();
        if ( f==1 ) {
            s.push ( p );    tag.push ( 2 );
            p = p->rchild;
        } else {
            if ( p==q )    sq = s;        // q 的所有祖先
            if ( p==r )    sr = s;              // s 的所有祖先
            p = NULL;
        }
    }
  }
  // 先跳过不同层的祖先，然后依次比较同一层的祖先
  if ( sq.size()>sr.size() )    while ( sq.size()>sr.size() )    sq.pop();
  else    while ( sr.size()>sq.size() )    sr.pop();
  // 求 q 和 r 的最近的共同祖先
  while ( !sq.empty() and (sq.top()!=sr.top()) ) {    //寻找共同祖先
    sq.pop();    sr.pop();
```

```
    }
    if ( !sq.empty() )
        return sq.top();
    else
        return NULL;
}
```

6.8 编写算法输出以二叉树表示的算术表达式（中缀形式），要求在必要的地方输出括号。
分析：当左孩子的优先级低于根时需要加括号，根的优先级大于右孩子时也需要加括号。

```
void PrintExpression ( BinTree bt )
{
    if ( bt==NULL )    return ;
    if ( bt->lchild==NULL and bt->rchild==NULL )
        print ( bt->data );                // 叶子结点直接打印
    else {
        // 左子树
        brackets = bt->lchild and is_operator(bt->lchild->data)
                and comp_operator(bt->lchild->data, bt->data)<0;   // 左孩子优先级低于根
        if ( brackets )    print ("(");
        PrintExpression ( bt->lchild );
        if ( brackets )    print (")");
        // 根结点
        print ( bt->data );
        // 右子树
        brackets = bt->rchild and is_operator(bt->lchild->data)
                and comp_operator(bt->data, bt->rchild->data)>0;   // 根的优先级大于右孩子
        if ( brackets )    print ("(");
        PrintExpression ( bt->rchild );
        if ( brackets )    print (")");
    }
}
```

注：is_operator(c)判断 c 是否是运算符；comp_operator(a,b)比较两个运算符的优先级。

```
bool is_operator(char c) {          // 判断 c 是否是运算符
    return c=='+' || c=='-' || c=='*' || c=='/';
}
int comp_operator(char opl, char opr) {      // 比较两个运算符的优先级
    return (opl=='*' || opl=='/' || opr=='+' || opr=='-') ? +1 : -1;
}
```

6.9 树采用孩子-兄弟链表存储，编写算法求树中叶子结点的个数。
分析：树中的叶子没有孩子，即 firstchild 为空。

```
// 求树 t 中叶子结点的个数
int LeafCount ( CSTree t )
```

```
{
   if ( t==NULL )    return 0;          // 空树
   if ( t->firstchild==NULL )          // 没有孩子
      return 1 + LeafCount(t->nextsibling);
   else
      return LeafCount(t->firstchild) + LeafCount(t->nextsibling);
}
```

6.10 采用孩子-兄弟链表存储树，编写算法求树的度。
分析：度最大的结点的度数。

```
int Degree ( CSTree t )
{
   if ( t==NULL )    return 0;
   else
      return max( Degree(t->firstchild), 1+Degree(t->nextsibling));
}
```

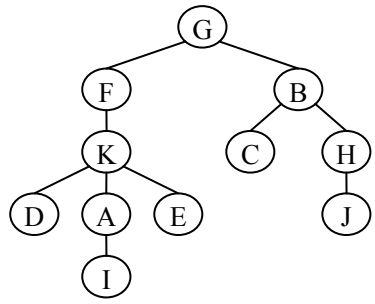
6.11 采用孩子-兄弟链表存储树，编写算法求树的深度。

```
int Depth ( CSTree t )
{
   if ( t==NULL )    return 0;
   else {
      depchild = Depth(t->firstchild);      // 孩子的深度
      depsibling = Depth(t->nextsibling);      // 兄弟的深度
      return max(depchild+1, depsibling);    // 取较大者
   }
}
```

6.12 已知二叉树的前序和中序序列，编写算法建立该二叉树。
分析：划分先序序列 a=(D,(L),(R))和后序序列 b=((L),D,(R))，然后对子序列(L)和(R)递归。

```
// 根据先序序列 a[si..ti]和中序序列 b[sj..tj]构造二叉树
BinTree CreateBinaryTree ( T a[], int si, int ti, T b[], int sj, int tj )
{
   if ( n<=0 )    return 0;                                    // 空树
   // 建立根结点
   p = new BinNode(a[si]);                                     // 以 a[si]为数据域建立新结点
   // 根据根结点划分中序序列为 b[sj..k-1]和 b[k+1..tj]
   k = sj;
   while ( b[k]!=a[si] )    k++;    // 在 b[]中搜索根结点 a[si]
   // 建立左右子树
   p->lchild = CreateBinaryTree ( a, si+1, si+k-sj, b, sj, k-1 );     // 建立左子树
   p->rchild = CreateBinaryTree ( a, si+k-sj+1, b, k+1, tj);   // 建立右子树
   return p;
}
```

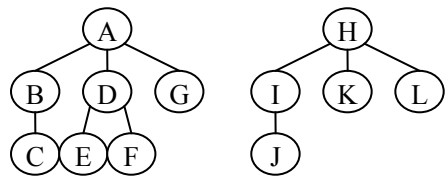
6.13 树T的先根遍历序列为GFKDAIEBCHJ，后根遍历序列为DIAEKFCJHBG，画出树T。



分析：根据先根和后根序列可以唯一确定一棵树。先根序列中的第一个是树的根，后根序列中最后一个是根，然后根据先根序列和后根序列，将其余序列划分成若干不相交的子序列，就是根的子树，对每一个子树，重复前面的步骤，最终就可以得到整个树。

先根 GFKDAIEBCHJ ➔根为 G，第一棵子树的根为 F，又后根 DIAEKFCJHBG，所以第一棵子树为(DIAEKF)，同样第二棵子树为(CJHB)，依此类推，可得该树。

6.14 一森林F转换成的二叉树的先序序列为ABCDEFGHIJKL，中序序列为CBEFDGAJIKLH。画出森林F。



分析：根据森林和二叉树的对应关系，可知森林的先序序列和中序序列。划分出每一棵树，正好得到每棵树的先序和后序序列，最终得到整个森林。

6.15 某通信过程中使用的编码有 8 个字符A,B,C,D,E,F,G,H，其出现的次数分别为 20, 6, 34, 11, 9, 7, 8, 5。若每个字符采用 3 位二进制数编码，整个通信需要多少字节？请给出哈夫曼编码，以及整个通信使用的字节数？

分析：由于每个字符出现的频率不同，使用固定长度的编码往往比哈夫曼编码使得整个通信量增多。这里先建立哈夫曼树，得出哈夫曼编码，然后计算通信所需的字节数。每字节含 8 位。

使用固定长度的编码所需字节数为(20+6+34+11+9+7+8+5)*3/8=37.5 字节。一种可能的哈夫曼编码是：A:00, B:1100, C:10, D:010, E:011, F:1110, G:1111, H:1101，通信的总字节数是[(20+34)*2+(11+9)*3+(6+5+7+8)*4]/8=34 字节。

6.16 n个权值构造的哈夫曼树共有多少个结点？

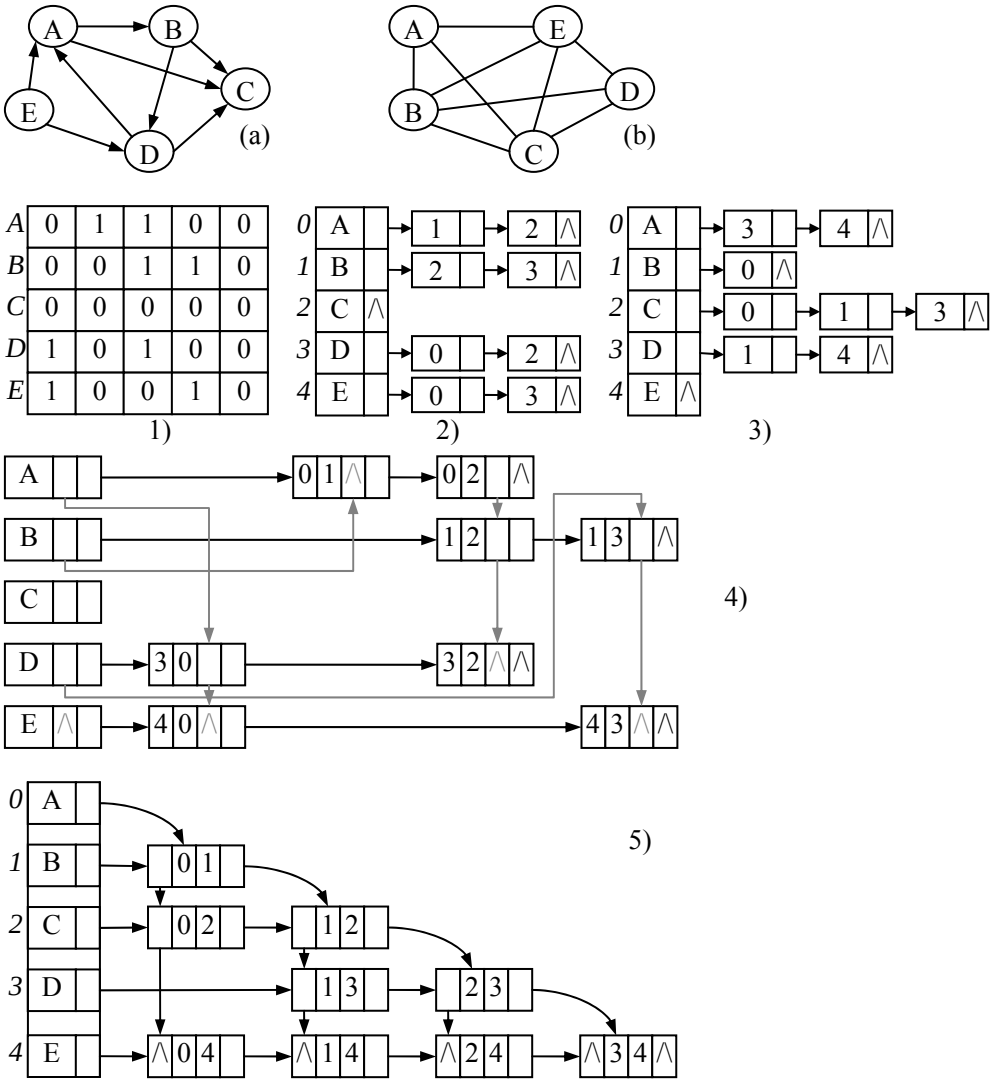
分析：哈夫曼树总是取两个最小的子树合并成一棵二叉树，共需 n-1 步完成算法，共增加 n-1 个结点，故总结点个数为 n+(n-1)=2n-1。

7.1 具有n个顶点的有向图构成强连通图最少有多少条弧？当弧的数目超过多少时该图一定是强连通的？

分析：如果 n 条弧恰好使 n 个顶点构成环的话，这是构成强连通图所需弧最少的情况。类似无向图的情况，n-1 个顶点最多有(n-1)(n-2)条弧，再增加 n-1 条指向另外一点顶点的弧(或者相反方向的弧)，此时该图恰好不能构成强连通图，若再增加一条弧则必定强连通。

因此，n 个顶点的有向图最少需要 n 条弧就可以构成强连通图；当弧的数目超过 $(n-1)(n-2)+(n-1) = (n-1)^2$ 时，必定构成强连通图。

7.2 给出下面有向图(a)的 1)邻接矩阵 2)邻接表 3)逆邻接表 4) 十字链表 和 无向图(b)的 5)邻接多重表。



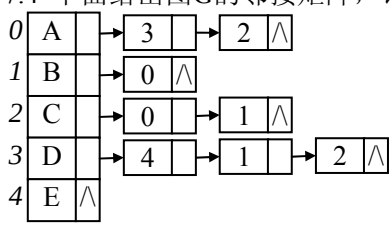
7.3 对习题 7.2 中图(a)和(b)从A出发进行深度优先搜索和广度优先搜索，写出遍历结果。

分析：画出搜索树，写出结果。

(a) 深度优先搜索：ABCDE，广度优先搜索：ABCDE；

(b) 深度优先搜索：ABCDE，广度优先搜索：ABCED。

7.4 下面给出图G的邻接矩阵，请写出从顶点A出发深度优先遍历的结果。



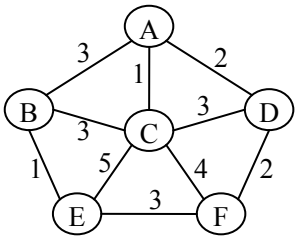
分析：这里用邻接表给出了图的存储结构，同时确定了邻接点的先后关系。仍然采用搜索树分析。
深度优先遍历结果：ADEBC。

7.5 写出图的深度优先遍历算法。
参考 第 7 章 一、3. (1) 3°．。

7.6 写出图的广度优先遍历算法。
参考 第 7 章 一、3. (2) 3°．。

7.7 证明最小生成树的MST性质。
参考课本 P173。

7.8 分别用普里姆算法和克鲁斯卡尔算法计算下图的最小生成树。

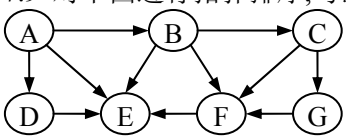


普里姆算法：

U	V-U	U 到 V-U 中各顶点的最小代价					最小代价边
		B	C	D	E	F	
{A}	{B,C,D,E,F}	AB/3	AC/1	AD/2	∞	∞	AC/1
{A,C}	{B,D,E,F}	AB/3		AD/2	CE/5	CF/4	AD/2
{A,C,D}	{B,E,F}	AB/3			CE/5	DF/2	DF/2
{A,C,D,F}	{B,E}	AB/3			FE/3		AB/3
{A,C,D,F,B}	{E}				BE/1		BE/1
{A,C,D,F,B,E}	{}						

克鲁斯卡尔算法：
依次选择代价最小的边：AC, BE, AD, DF，然后，可以选择 AB 或 BC 或 EF 即可。
最小生成树的总代价是 9。

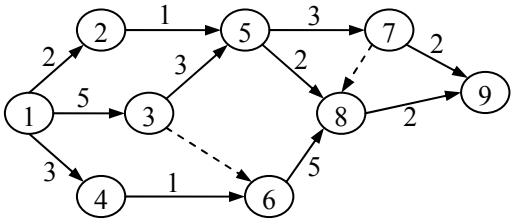
7.9 对下图进行拓扑排序,写出所有可能的拓扑有序序列。



分析：可能的拓扑有序序列有多种，A 之后可能是 B 或 D，逐渐类推可得到所有可能的拓扑序列：ADBCGFE, ABDCGFE, ABCDGFE, ABCGDFE, ABCGFDE, ABCGFED，

共 6 种。

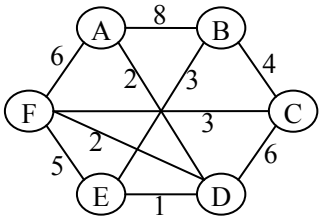
7.10 下面是某工程的AOE网，计算 1)整个工程完工需要多长时间(单位：天)？ 2)工程的关键路径。



事件	最早发生时间 ve	最晚发生时间 vl	活动	最早开始时间 e	最晚开始时间 l
v1	0	0	a(1,2)	0	5
v2	2	7	a(1,3)	0	0
v3	5	5	a(1,4)	0	2
v4	3	5	a(2,5)	2	7
v5	8	8	a(3,5)	5	5
v6	5	6	a(3,6)	5	6
v7	11	11	a(4,6)	3	5
v8	11	11	a(5,7)	8	8
v9	13	13	a(5,8)	8	9
			a(6,8)	5	6
			a(7,8)	11	11
			a(7,9)	11	11
			a(8,9)	11	11

整个工程完工需要 13 天，关键路径有 1→3→5→7→8→9 和 1→3→5→7→9。

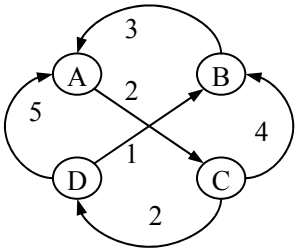
7.11 用迪杰斯特拉算法计算下图中顶点A到其他顶点的最短路径。



终点	从 A 到各顶点的最短路径				
B	8 AB	8 AB	6 ADEB	6 ADEB	
C	∞	8 ADC	8 ADC	7 ADFC	7 ADFC
D	2 AD				
E	∞	3 ADE			

F	6 AF	4 ADF	4 ADF		
最短路径	2 AD	3 ADE	4 ADF	6 ADEB	7 ADFC

7.12 用弗洛伊德算法计算下图中每对顶点之间的最短路径。



0	∞	2 AC	∞
3 BA	0	∞	∞
∞	4 CB	0	2 CD
5 DA	1 DB	∞	0

$A^{(0)}$

0	∞	2 AC	∞
3 BA	0	*5* *BAC*	∞
∞	4 CB	0	2 CD
5 DA	1 DB	*7* *DAC*	0

$A^{(1)}$

0	∞	2 AC	∞
3 BA	0	5 BAC	∞
7 *CBA*	4 CB	0	2 CD
4 *DBA*	1 DB	*6* *DBAC*	0

$A^{(2)}$

0	*6* *ACB*	2 AC	*4* *ACD*
3 BA	0	5 BAC	*7* *BACD*
7 CBA	4 CB	0	2 CD
4 DBA	1 DB	6 DBAC	0

$A^{(3)}$

0	*5* *ACDB*	2 AC	4 ACD
3 BA	0	5 BAC	7 BACD
6 *CDBA*	*3* *CDB*	0	2 CD
4 DBA	1 DB	6 DBAC	0

$A^{(4)}$

9.1 对n=10 个元素的表顺序查找，在等概率情况下(对每个元素查找的概率相同)，查找成功时的平均查找长度是多少？如果查找失败的概率为 0.2，平均查找长度又是多少？

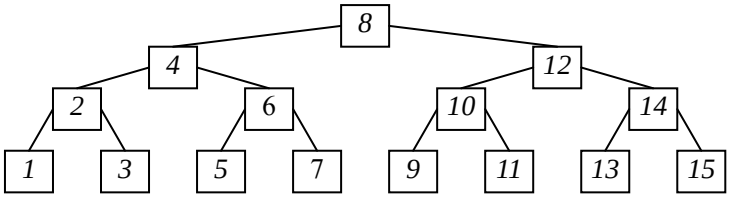
分析：等概率情况下查找成功时 ASL = (n+1)/2 = 5.5；若查找失败的概率为 0.2，则 ASL = 0.8×(10+1)/2 + 0.2×11 = 6.6。

技巧：可以用归纳法。查找第 1 个元素比较 10 个关键字，查找第 2 个元素比较 9 个关键字，……，查找第 10 个元素比较 1 个关键字，总共比较 10+9+…+1=55 个关键字，ASL=55/10=5.5。

提示：以上分析都是按照课本上的算法，如果遇到其他具体算法还需要具体分析。[35]

[35] 按照课本上的算法，查找从后往前进行，所以查找第 1 个元素要比较 10 个关键字。查找失败时，由于和“哨兵”记录多比较了 1 次，所以需要比较 11 个关键字。

9.2 表长 15 的有序表进行折半查找，计算等概率情况下查找成功时的平均查找长度和查找第 3 个元素需要比较关键字的个数。
分析：画出判定树进行分析、计算。



技巧：不断计算区间的中点下标(即两端下标之和除以 2)作为根结点，最终画出判定树。
ASL = (1×1+2×2+4×3+8×4)/15 = 49/15。查找第 3 个元素需要比较 4 个关键字，即第 8、4、2 和第 3 个。

9.3 对表长 1023 的有序表折半查找，比较 8 个关键字才能找到的元素有多少个？至多比较 8 次就能找到的元素有多少个？
分析：比较 8 个关键字才能找到的元素位于判定树的第 8 层。对长度为 1023 的有序表折半查找的判定树有 10 层，第 1 至 8 层应该是满二叉树，所以恰好比较 8 次才找到的有 $2^7 = 128$ 个，至多比较 8 次就能找到的是位于判定树第 1 至 8 层所有结点，共 $2^8-1 = 255$ 个。

9.4 长度为 10000 的表进行分块查找，用顺序查找确定所在块，块内元素无序，为使平均查找长度最小应该分几块？
分析：相当于索引顺序表的查找，当采用顺序查找时为使平均查找长度最小，应该划分成sqrt(n) = 100 块。具体分析参见“第 9 章 一、4. 索引顺序表”。

9.5 编写算法判断给定的二叉树是否是二叉排序树。
思路：根据中序遍历序列是否升序来判断。
非递归算法：

```
bool IsBinarySortTree ( BinTree bt )
{
  InitStack(S);
  p = bt;   pre = 0;                    // pre 保持为 p 的中序前驱
  while ( p or ! StackEmpty(S) ) {
    if ( p ) {
      Push(S, p);
      p = p->lchild;
    } else {
      p = Pop(S);
      if ( pre and (p->data <= pre->data) )   return false;   // 不是二叉排序树
      pre = p;                          // 记下前驱结点
      p = p->rchild;
    }
  }
  return true;   // 二叉排序树
```

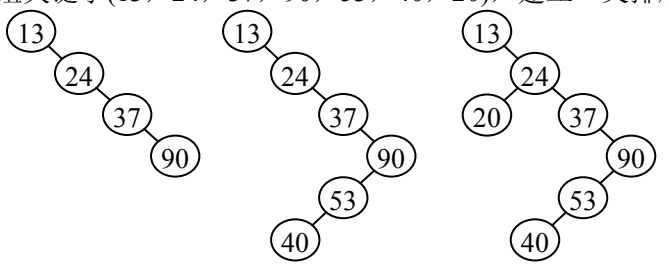
```
}
```

递归算法：

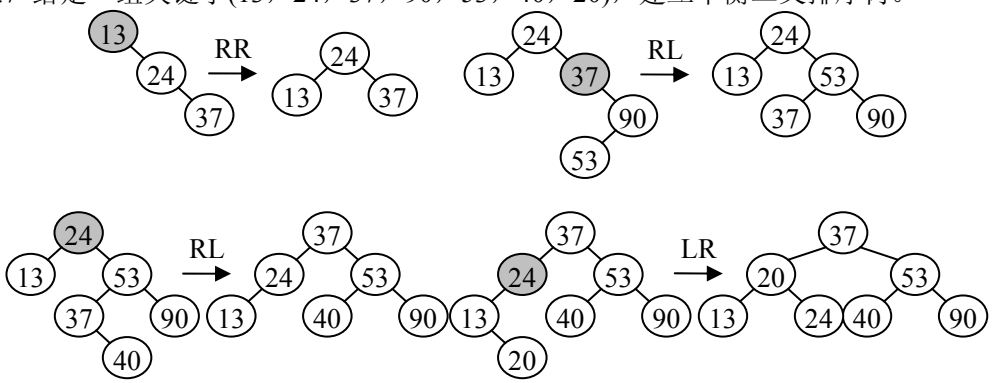
```
bool IsBinarySortTree ( BinTree bt )
{
   pre = NULL;
   return IsBST ( bt, pre );   // 调用递归算法
}

// 判断二叉树 bt 是否是二叉排序树，pre 是 bt 的前驱结点
bool IsBST (BinTree bt, BinTree &pre )
{
   if ( bt==0 )   return true;       // 空树是二叉排序树
   else {
      if ( not IsBST(bt->lchild, pre) )   return false; // 左子树
      if ( pre and (bt->data <= pre->data) )   return false;   // 根结点
      pre = bt;                          // 保持前驱结点
      if ( not IsBST(bt->rchild, pre ) )   return false;   // 右子树
      return true;
   }
}
```

9.6 给定一组关键字(13，24，37，90，53，40，20)，建立二叉排序树。



9.7 给定一组关键字(13，24，37，90，53，40，20)，建立平衡二叉排序树。



9.8 编写平衡二叉排序树的查找算法。

思路：实际上就是对二叉排序树的查找。具体算法参见“第 9 章 一、5. ”二叉排序

树的查找算法(递归和非递归算法)。

9.9 具有 20 个结点的二叉排序树的最大深度是(___)，最小深度是(___)；20 个结点的平衡二叉排序树最大深度是(___)。
结果：20，5，6。
分析：最坏情况下，二叉排序树的深度为 n(结点个数)，最小深度等于完全二叉树的深度$\lfloor \log n \rfloor + 1$。
平衡二叉排序树的最大深度和结点数的关系为 $N_h=N_{h-1}+N_{h-2}+1$，$N_0=0$，$N_1=1$。

9.10 在一棵非空的 5 阶B-树中，非叶子结点中最少有(___)棵子树，最多有(___)个关键字；除根以外的非终端结点中最少有(___)棵子树。
结果：2，4，3。
分析：非空 B-树中的根结点最少可以有 2 棵子树，其他非终端结点至少含有$\lceil m/2 \rceil$=3 棵子树。结点最多有 m=5 棵子树，m-1=4 个关键字。

9.11 已知一组关键字{19, 14, 23, 1, 68, 20, 85, 9}，采用哈希函数H(key)=key MOD 13，请分别采用以下处理冲突的方法构造哈希表，并求各自的平均查找长度。
1) 采用线性探测再散列；
2) 采用伪随机探测再散列，伪随机函数为$f(n) = n^2+2n+3$；
3) 采用链地址法。
思路：依次插入关键字建立哈希表。详细步骤参见“第 9 章 一、9. (5) 举例”。

1)

0	1	2	3	4	5	6	7	8	9	10	11	12
	14_1	1_2	68_1			19_1	20_1	85_2	9_1	23_1		

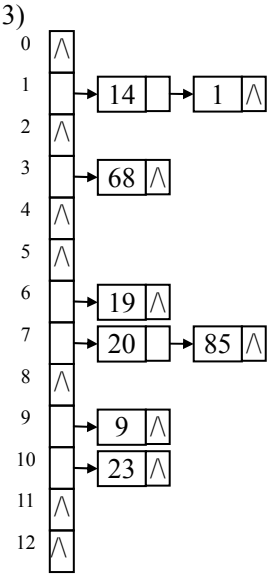
ASL = (1+2+1+1+1+2+1+1)/8 =5/4=1.25

2)

0	1	2	3	4	5	6	7	8	9	10	11	12
20_2	14_1		68_1		85_3	19_1	1_2		9_1	23_1		

ASL = (2+1+1+3+1+2+1+1)/8 =3/2=1.5

3)



ASL = (1+2+1+1+1+2+1+1)/8 =5/4=1.25

10.1 对待排序列(24, 86, 48, 56, 72, 36)进行 1) 直接插入排序，2) 希尔排序，3) 起泡排序，4) 快速排序，5) 简单选择排序，6) 堆排序，7) 归并排序，8) 链式基数排序。

1) 直接插入排序
   (24), 86, 48, 56, 72, 36
   (24, 86), 48, 56, 72, 36
   (24, 48, 86), 56, 72, 36
   (24, 48, 56, 86), 72, 36
   (24, 48, 56, 72, 86), 36
   (24, 36, 48, 56, 72, 86)
2) 希尔排序
   24, 86, 48, 56, 72, 36 dk=3
   24, 72, 36, 56, 86, 48 dk=2
   24, 48, 36, 56, 86, 72 dk=1
   24, 36, 48, 56, 72, 86
3) 起泡排序
   24, 86, 48, 56, 72, 36
   24, 48, 56, 72, 36,(86)
   24, 48, 56, 36,(72, 86)
   24, 48, 36,(56, 72, 86)
   24, 36,(48, 56, 72, 86)
   24,(36, 48, 56, 72, 86)
4) 快速排序
   {24, 86, 48, 56, 72, 36}

   {}24{86, 48, 56, 72, 36}
      {36, 48, 56, 72}86{}
     {}36{48, 56, 72}
       {}48{56, 72}
         {}56{72}
   (24, 36, 48, 56, 72, 86)
5) 简单选择排序
   24, 86, 48, 56, 72, 36
   (24), 86, 48, 56, 72, 36
   (24, 36), 48, 56, 72, 86
   (24, 36, 48), 56, 72, 86
   (24, 36, 48, 56), 72, 86
   (24, 36, 48, 56, 72), 86
6) 堆排序

24, 86, 48, 56, 72, 36
(86, 72, 48, 56, 24, 36) (建立堆)
(72, 56, 48, 36, 24), 86
(56, 36, 48, 24), 72, 86
(48, 36, 24), 56, 72, 86
(36, 24), 48, 56, 72, 86
(24), 36, 48, 56, 72, 86

7) 归并排序

(24),(86),(48),(56),(72),(36)
(24, 86), (48, 56), (36, 72)
(24, 48, 56, 86), (36, 72)
(24, 36, 48, 56, 72, 86)

8) 链式基数排序

(24, 86, 48, 56, 72, 36)
分配：
[0][1][2][3][4][5][6][7][8][9]
72 24 86 48
56
36
收集：72, 24, 86, 56, 36, 48
分配：
[0][1][2][3][4][5][6][7][8][9]
24 36 48 56 72 86
收集：(24, 36, 48, 56, 72, 86)

10.2 证明：借助比较进行的排序方法，在最坏情况下所能达到的最好的时间复杂度是O(nlogn)。
分析：含有 n 个记录的序列排序可能的初始状态有 n!个，所以描述 n 个记录排序过程的判定树必有 n!个叶子，二叉树的高度 $h \geqslant \log_2(n!)+1$。该判定树上必定存在长度为 $\log_2(n!)$的路径。所以借助比较的排序方法在最坏情况下的所需要比较次数至少为 $\log_2(n!)$。时间复杂度为 $O(\log_2(n!)) = O(n\log n)$。