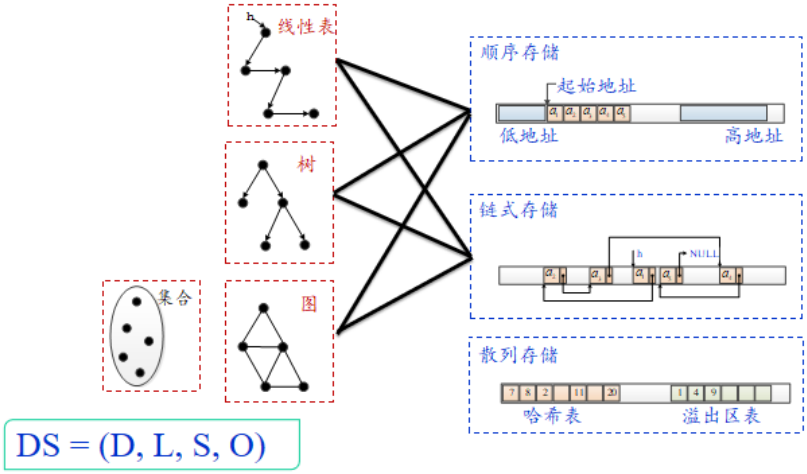
数据结构与算法

第一章 绪论

程序=数据结构+算法

数据结构

带结构的数据元素的集合。



数据结构二元组表达：

用集合的形式描述，数据结构是一个二元组：

$$DS=(D,R)$$

其中：D 是数据元素的集合，R 是 D 上关系的集合。

数据结构四元组表达：

$$Data\_Structure=(D,L,S,O)$$

D (Data)是数据元素的有限集，是存储和操作的对象；

L (Logical Structure)是数据元素集合D中数据元素之间客观存在的关系的有限集，称为逻辑结构；

S (Storage Structure)是数据元素集合D和数据元素之间的关系集合L在计算机中的存储表示，称为存储结构或物理结构；

O (Operation)是在数据元素集合D上规定的一组操作。

数据结构包含：数据元素、数据元素之间的逻辑关系、逻辑关系在计算机中的存储表示、以及所规定的操作这四部分。

数据 D

数据(Data)

数据是对客观事物的符号表示，是信息的载体；

在计算机科学中数据指所有能够被计算机识别的符号集合；

“识别”：输入、存储、处理、显示、输出；

“符号”：数字、字母、汉字、语音、图形、图像。

数据对象(Data Object)

数据对象是具有相同性质的数据元素的集合，是数据的一个子集。

数据元素(Data Element)

是数据(集合)中的一个“个体”，是数据结构中讨论的基本单位。

数据项(Data Item)

是数据结构中讨论的最小单位，数据元素可以是数据项的集合。

数据项 ⊆ 数据元素 ⊆ 数据对象 ⊆ 数据

逻辑结构 L

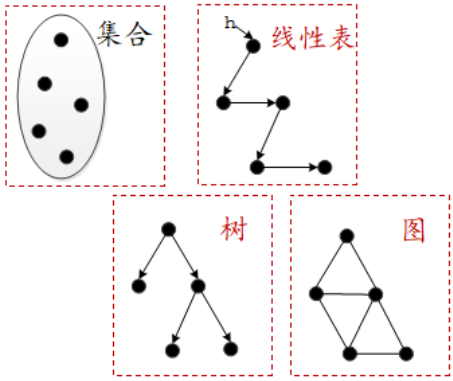
逻辑结构是指数据元素之间客观存在的关系，与数据在计算机中如何存储无关，主要用于人们理解和交流、以及指导算法的设计。

集合结构

线形结构

图形结构

树形结构



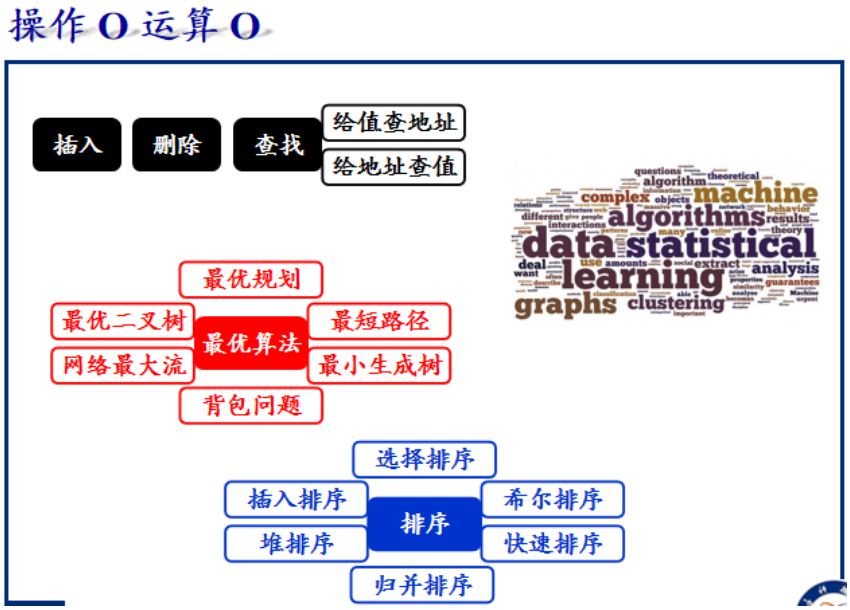
存储结构 S

数据的存储结构也称物理结构，指数据结构在计算机中的存储表示，包括数据结构中元素的表示及元素间关系的表示。

顺序存储结构：把逻辑上相邻的元素存储在物理位置相邻的存储单元中，特点是借助于数据元素在存储器中的相对位置来表示数据元素之间的逻辑关系。

链式存储结构：在数据元素中添加一些地址域或辅助结构，用于存放数据元素之间的关系，特点是借助于指示数据元素地址的指针表示数据元素之间的逻辑关系，通常借助于程序设计语言中的指针来实现。

散列存储结构：也称哈希(Hash)结构，通过对关键字直接计算得到数据元素的存储位置，特点数据元素的存储和查找都是通过对关键字直接计算得到的。



算法

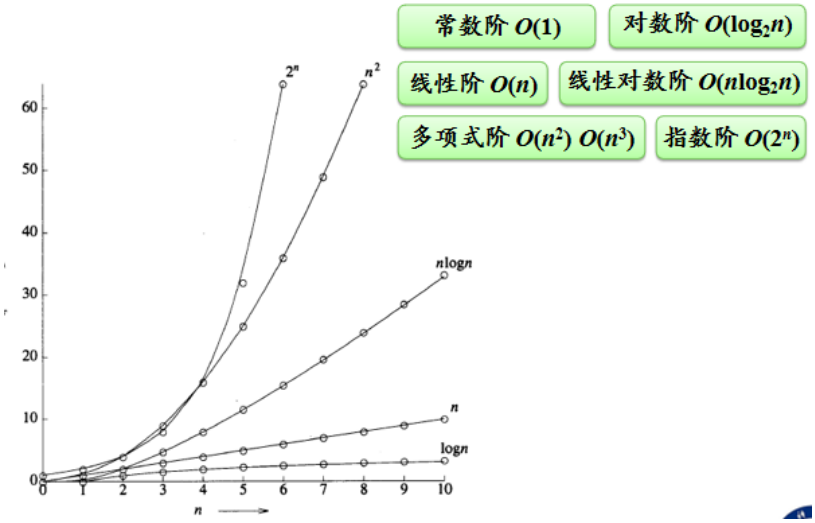
算法是计算机求解特定问题的有限步骤序列。

算法五特性：有输入、有输出、有穷性、确定性、可行性

算法四特点：正确性、可读性、健壮性、高效性

算法设计的好坏评价：时间复杂度 vs. 空间复杂度

时间复杂度



空间复杂度

$$S(n) = O(g(n))$$

算法的存储量包括有三：

① 输入数据占用空间、

② 程序本身占用空间、

③ 辅助变量占用空间。

第二章 线性表

线性表

线性表是一种最简单的线性结构。线性结构的基本特征为：线性结构是数据元素的有序(次序)集合。

1. 集合中必存在唯一的一个“第一元素”；
2. 集合中必存在唯一的一个“最后元素”；
3. 除最后元素在外，均有唯一的后继；
4. 除第一元素之外，均有唯一的前驱。

线性表求并集

题目：已知集合 A 和 B，求两个集合的并集，使 A = A∪B，且 B 不再单独存在。

分析：以线性表 LA 和 LB 分别表示集合 A 和 B，对集合 B 中的所有元素一个一个地检查，将存在于线性表 LB 中而不存在于线性表 LA 中的数据元素插入到线性表 LA 中去。

算法步骤：

1. 从线性表 LB 中取出一个数据元素
2. 依值在线性表 LA 中进行查询
3. 若不存在，则将它插入到 LA 中
4. 重复上述三步直至 LB 为空表止

必要的线性表基本操作：

1. ListDelete (LB, 1, e)
2. LocateElem (LA, e)
3. ListInsert (LA, n+1,e)

伪代码：

```
void ListUnion(List &LA, List &LB){
    La_len = ListLength(LA); //求得线性表LA的长度
    while (!ListEmpty(LB)) //依次处理LB中元素直至LB为空表止
    {
        ListDelete(LB,1,e); //从LB中删除第1个数据元素并赋给e
        // 当LA中不存在和e值相同的数据元素时进行插入
        if(!LocateElem(LA,e)
            ListInsert(LA,++La_len,e);
    }
    DestroyList(LB); // 销毁线性表LB
}
```

线性表有序表归并

有序表：若线性表中的数据元素相互之间可以比较，并且数据元素在线性表中依值非递减或非递增有序排列，即 $a_i \geq a_{i-1}$ 或 $a_i \leq a_{i-1}$ (i = 2,3,..., n)，则称该线性表为有序表。

题目：将非递减的有序表 La 和 Lb 归并为 Lc。

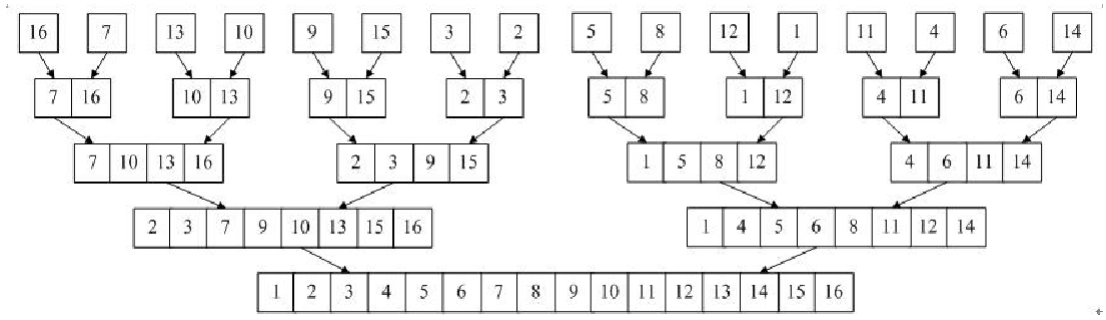
伪代码：

```
void MergeList(List La, List Lb, List &Lc)
```

```
{
    InitList(Lc);
    i = j = 1; k = 0;
    La_len = ListLength(La);
    Lb_len = ListLength(Lb);
    while ((i <= La_len) && (j <= Lb_len))
    {  // La 和 Lb 均不空
        GetElem(La, i, ai);
        GetElem(Lb, j, bj);
        if (ai <= bj) //将 ai 插入到 Lc 中
            {ListInsert(Lc, ++k, ai);  ++i;}
        else //将 bj 插入到 Lc 中
            {ListInsert(Lc, ++k, bj);  ++j;}
    }
    while (i <= La_len)
    { // 当La不空时插入 La 表中剩余元素
        GetElem(La, i++, ai);
        ListInsert(Lc, ++k, ai);
    }
    while (j <= Lb_len)
    { // 当Lb不空时插入 Lb 表中剩余元素
        GetElem(Lb, j++, bj);
        ListInsert(Lc, ++k, bj);
    }
}
```

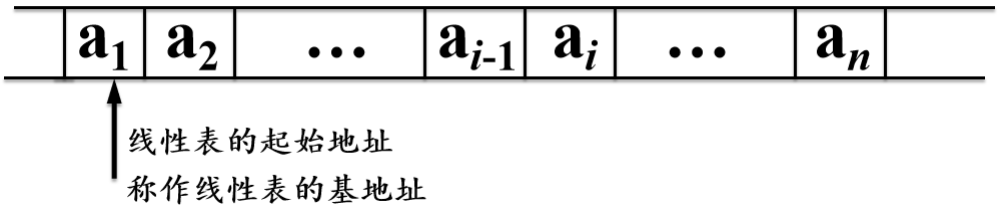
应用：

归并排序（时间复杂度：O(*n*log*n*)）



顺序表（Sequence List）

用一组地址连续的存储单元依次存放线性表中的数据元素。



所有数据元素的存储位置均取决于第一个数据元素的存储位置：

$$LOC(a_i) = \underbrace{LOC(a_1)}_{\text{基地址}} + (i-1) \times c$$

定义顺序表代码：

```
#define MAXSIZE 100
typedef int ElemType;
Typedef struct
{
   ElemType  elem[MAXSIZE];
   int       last;
}SeqList;
```

查值查找 ListLocate(·)

题目：在顺序表中查找某一个元素是否存在。

方法：将顺序表中的元素逐个和给定值 e 相比较。

时间复杂度：O(n)

伪代码：

```
#define MAXSIZE 100
typedef int ElemType;
typedef struct SeqList
{
  ElemType elem[MAXSIZE];
  int      last;
}SeqList;

int ListLocate(SeqList L, ElemType e)
{
   int i=0;
   while((i<=L.last)&&(L.elem[i]!=e))
      i++;
   if(i<=L.last)
      return(i+1);
   else
      return(-1);
}
```

插入操作 ListInsert(·)

题目：在顺序表的某个位置插入一个元素。

方法：目标位置之后的元素向后移动，再将元素插入留出的空位。

时间复杂度：O(n)

伪代码：

```
#define MAXSIZE  100
typedef int ElemType;
typedef struct SeqList
{
   ElemType elem[MAXSIZE];
   int      last;
}SeqList;
```

```
Int ListInsert (SeqList *L, int pos, ElemType e)
{
   if( pos<1 || pos>L->last+2 || L->last>= MAXSIZE-1)
      return(ERROR);
   for( int i=L->last; i>=pos-1; i-- )
      L->elem[i+1] = L->elem[i];
   L->elem[pos-1] = e;
   L->last++;
   return(OK);
}
```

删除操作 ListDelete (·)

题目：在顺序表的某个位置删除一个元素。

方法：目标位置之后的元素向前移动，覆盖要删除的元素。

时间复杂度：O(n)

伪代码：

```
#define MAXSIZE  100
typedef int ElemType;
typedef struct SeqList
{
   ElemType elem[MAXSIZE];
   int      last;
}SeqList;

int ListDelete(SeqList *L, int pos, ElemType *e)
{
   if( pos<1 || pos>L->last+1 )
        return(ERROR);
    *e = L->elem[pos-1];
    for(int i=pos; i<=L->last; i++)
        L->elem[i-1] = L->elem[i];
    L->last--;
    return(OK);
}
```

数组和指针

```
int array[100];

int n = 100;
int * parr1 = new int[n];
delete(parr1);

int * parr2 = (int *)malloc(n*sizeof(int));
free(parr2);

int * parr3 = array;
int a = sizeof(array);
int b = sizeof(parr3);
```

```
13
14 char str1[128] = "thisisafilename.txt";
15 char str2[128] = "";
16 char * pst1 = str1;
17 char * pst2 = str2;
18 while(*pst2++ = *pst1++);
```

应用：就地逆置-1

题目：将存放在动态数组array中的所有元素逆置，空间复杂度O(1)。

代码：

```
1  //方法1：需要一个额外空间
2  void SeqListReverse(int *array, int n)
3  {
4      int temp;
5      for(int i = 0; i < n/2; i++)
6      {
7          temp = array[i];
8          array[i] = array[n-1-i];
9          array[n-1-i] = temp;
10     }
11 }
12
13 //方法2：不需要任何额外空间
14 void SeqListReverse(int *array, int n)
15 {
16     int * p = &array[0];
17     int * q = &array[n-1];
18     while( p++ < q-- )
19     {
20         *p += *q;
21         *q  = *p - *q;
22         *p -= *q;
23     }
24 }
25
```

应用：就地逆置-2

题目：设存放在动态数组array中的元素分为前n个和后m个，设计算法将后m个元素前置将前n个元素后置。要求时间和空间复杂度尽可能低。

$$a_1 \quad a_2 \quad \cdots \quad a_n \quad b_1 \quad b_2 \quad \cdots \quad \cdots \quad b_{m-1} \quad b_m$$
$$b_1 \quad b_2 \quad \cdots \quad \cdots \quad b_{m-1} \quad b_m \quad a_1 \quad a_2 \quad \cdots \quad a_n$$

代码：

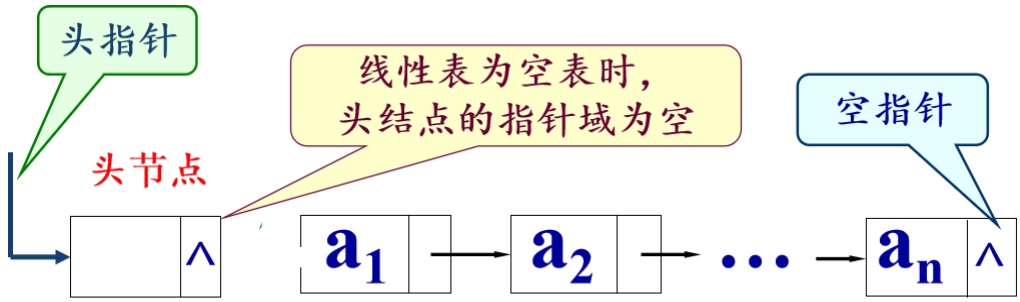
```
1  void SeqListReverse(int *array, int n)
2  {
3      int temp;
4      for(int i = 0; i < n/2; i++)
5      {
6          temp = array[i];
```

```
        array[i] = array[n-1-i];
        array[n-1-i] = temp;
    }
}
void SeqListMove(int *array, int n, int m)
{
    SeqListReverse(array,n+m);
    SeqListReverse(array,m);
    SeqListReverse(array+m,n);
}
```

单链表（Link List)

用一组地址任意的存储单元存放线性表中的数据元素。

元素(数据元素的映象) + 指针(指示后继元素存储位置) = 节点 (表示数据元素或数据元素的映象)



定义单链表代码：

```
typedef struct node
{
  DataType data;
  struct node * next;
} LinkListNode,*LinkList,**List;
```

查值查找 LinkedList_Locate(·)

题目：在单链表中找到第i个元素。

方法：移动指针，比较 i 和 pos ，令指针 p 始终指向线性表中第 i 个数据元素。

时间复杂度：O(n)

伪代码：

```
typedef int DataType;
typedef struct
{
  DataType data;
  struct node * next;
} LinkedList;

DataType LinkedList_Locate(LinkedList *L, int pos)
{
    LinkedList * p = L->next;
    int i = 0;
```

```
    while( p != NULL && i < pos-1)
    {
          i++;
          p = p->next;
    }
    if( p != NULL && i== pos -1)
        return p->data;
    return -1;
}

```

插入操作 LinkListInsert(·)

题目：在单链表的某个位置插入一个元素。

方法：目标位置前一个元素的指针指向要插入的元素，将要插入的元素的指针指向目标位置后一个元素。

时间复杂度：O(n)

伪代码：

```
typedef struct node
{
  DataType data;
  struct node * next;
} LinkListNode;

int LinkListInsert(LinkListNode L, int pos, ElemType e)
{
    LinkListNode * p = L;
    LinkListNode * s;
    int k = 0;
    while( p!=NULL && k < pos-1)
    {
        p = p->next; k++;
    }
    if(k==pos)
    {
        s=(LinkListNode*)malloc(sizeof(LinkListNode));
        s->data = e; s->next = p->next;
        p->next = s;
        return OK;
    }
    return ERROR;
}
```

删除操作 LinkListDelete(·)

题目：在单链表的某个位置删除一个元素。

方法：找到线性表中第i-1个结点，修改其指向后继的指针。

时间复杂度：O(n)

伪代码：

```c
typedef struct node
{
  DataType data;
  struct node * next;
} LinkListNode;

int LinkListDelete(LinkListNode * L, int pos)
{
    LinkListNode * p = L;
    LinkListNode * s;
    int k = 0;
    while( p!=NULL && k < pos-1)
    {
        p = p->next; k++;
    }
    if(k==pos)
    {
        q = p->next;   p->next = q->next;
        e = q->data;   free(q);
        return OK;
    }
    return ERROR;
}
```

应用：就地逆置

题目：将存放在单链表中的所有元素逆置。

代码：

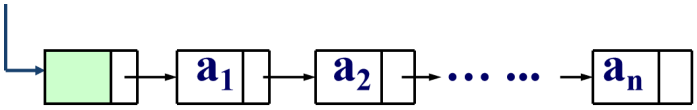
```c
typedef struct LinkListNode
{
    int data;
    struct LinkList* next;
} LinkListNode, *LinkList;

void Reverse(LinkList list)
{
    LinkList p, q;
    if ( list == NULL )
        return;
    p = list->next;
   list->next=NULL;
    while ( p != NULL )
    {
        q = p; // 将p指向旧链第一个元素赋给q控制
        p = p->next; // p始终指向旧链的第一个元素
        q->next = list->next; // q的next指向新链首
        list->next = q; // q被挂入新链
    }
}
```

链表的变形

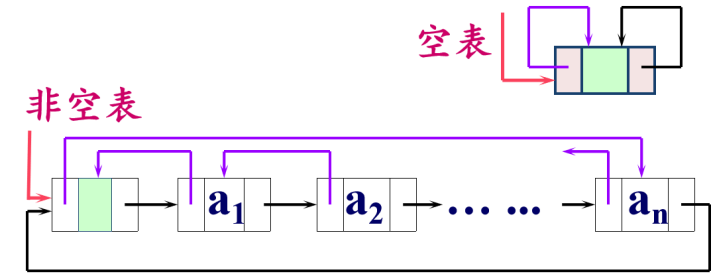
单循环链表

最后一个结点的指针域的指针又指回第一个结点的链表。

单循环链表和单链表的差别仅在于，判别链表中最后一个结点的条件不再是“后继是否为空”，而是“后继是否为头结点”。



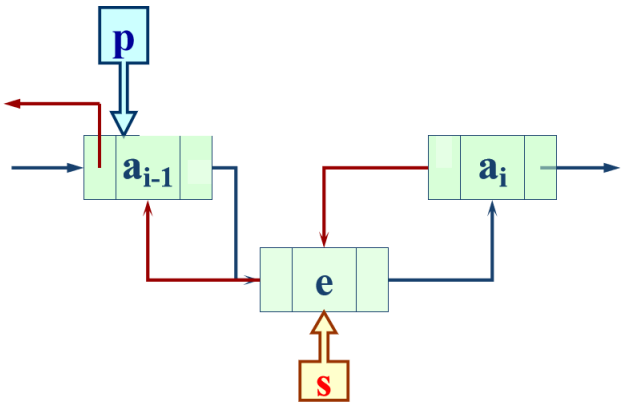
双向链表



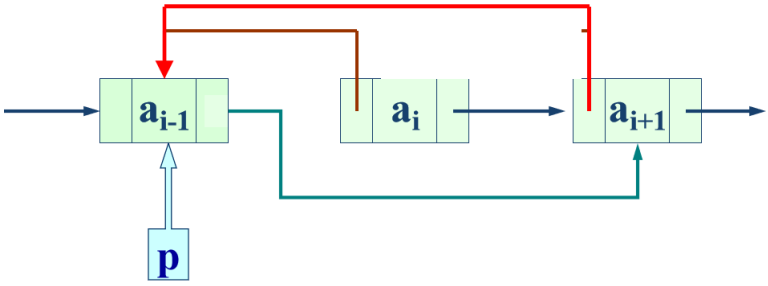
定义双向链表代码：

```
typedef struct  DLNode
{
    ElemType        data;    // 数据域
    struct DLNode * prior;   // 指向前驱的指针域
    struct DLNode * next;    // 指向后继的指针域
} DLNode, *DLinkList;
```

插入操作



删除操作




```
p->next = p->next->next;

p->next->prior = p;
```

栈和队列

栈和队列都是运算受限的线性表：

- 栈

  插入/删除操作只能在表的一端进行

  数据元素 后进先出(LIFO)/先进后出(FILO)

- 队列

  插入/删除操作分别在表的两端进行

  数据元素 先进先出(FIFO)

栈和队列的共同点：

- 逻辑结构为线性表
- 存储结构同样是有顺序和链式两种

栈

栈是特殊的线性表：

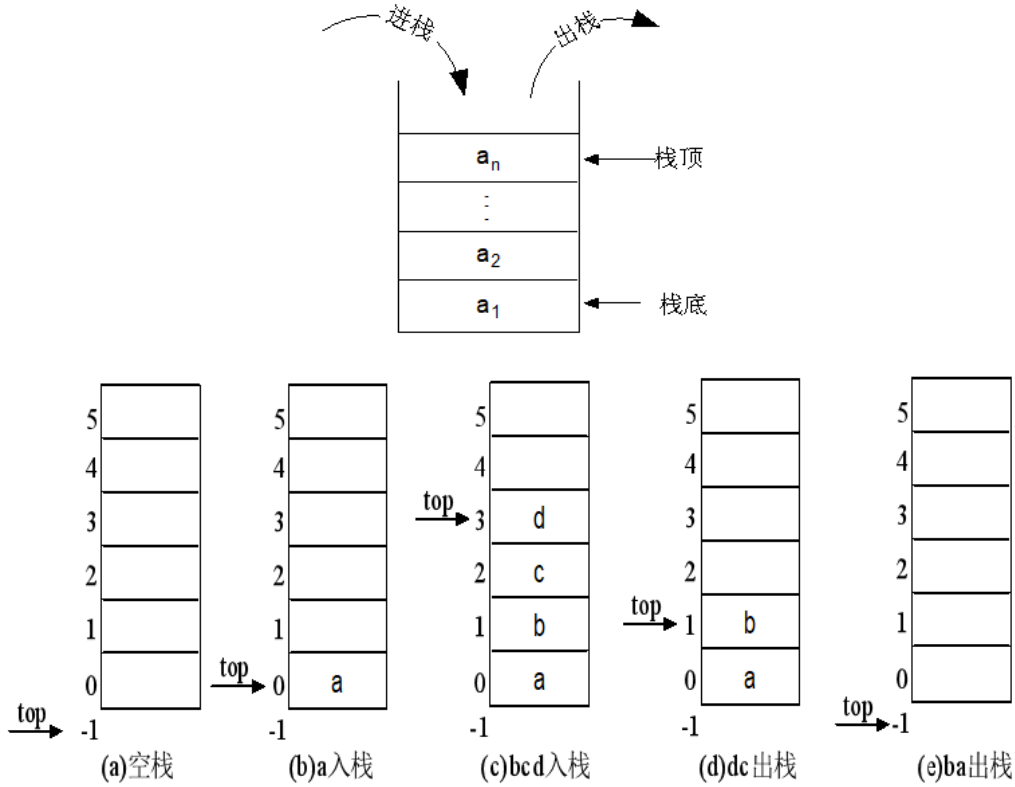
- 它有固定的一端，称为 栈底
- 它有操作的一端，称为 栈顶

定义栈代码：

```
#define MAXSTACK 100          // 数据空间
typedef StructStack
{
    int top ;                 // 栈顶位置
    StackEntry entry[MAXSTACK] ;     // 域entry用于存放数据元素
}  Stack, *StackPtr ;
//约定：top用于存放栈顶元素的位置，因此top == -1表示空栈，top==MAXSIZE-1表示栈满
```

栈的操作：

- 插入元素：入栈（入栈时top指针加1）
- 删除元素：出栈（出栈时top指针减1）



栈的空满：

- 栈中元素个数为零：空栈（空栈不能进行出栈操作）
- 栈中元素个数达到上限：满栈（满栈不能进行入栈操作）

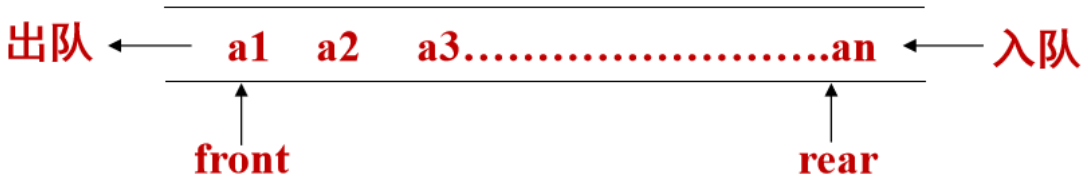
栈的溢出：

- 顺序栈的数据元素空间大小是预先分配
- 当空间满时再进行入栈操作产生的溢出称为上溢
- 当栈为空时再进行出栈操作产生的溢出称为下溢

队列

队列是特殊的线性表：

- 限定只能在表的一端(队头，front)进行删除
- 限定只能在表的另一端(队尾，rear)进行插入



定义队列代码：

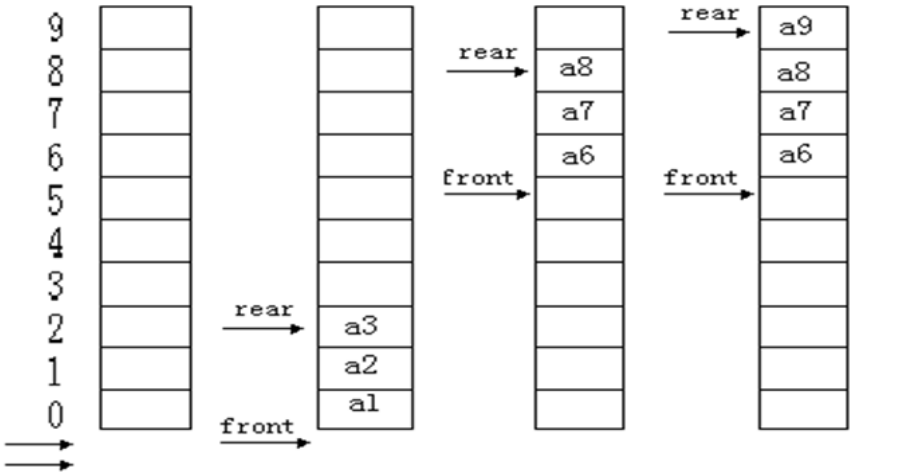
```
#define MAXQUEUE  10
typedef struct queue
{
    int  front, rear; /*分别指示对头和队尾数据元素的位置*/
    QueueEntry entry[MAXQUEUE]; /*数据元素存储空间 */
} Queue,*QueuePtr; /*定义为新的数据类型*/
//约定：队头指针指向队头元素前面一个位置，队尾指针指向队尾元素位置。
```

队列的空满：

- 队列中元素个数为零：空队列
- 队列中元素个数达到上限：满队列

队列的假上溢现象：

问题：初始为空队列；随后数据元素a1,a2,a3依次入队；然后数据元素a4, a5, a6, a7, a8依次入队；a1到a5依次出队；然后数据元素a9入队。



解决方法：

- 方法一

  固定队头指针永远指向数据区开始位置

  如果数据元素出队，则将队列中所有数据元素前移一个位置，同时修改队尾指针

- 方法二

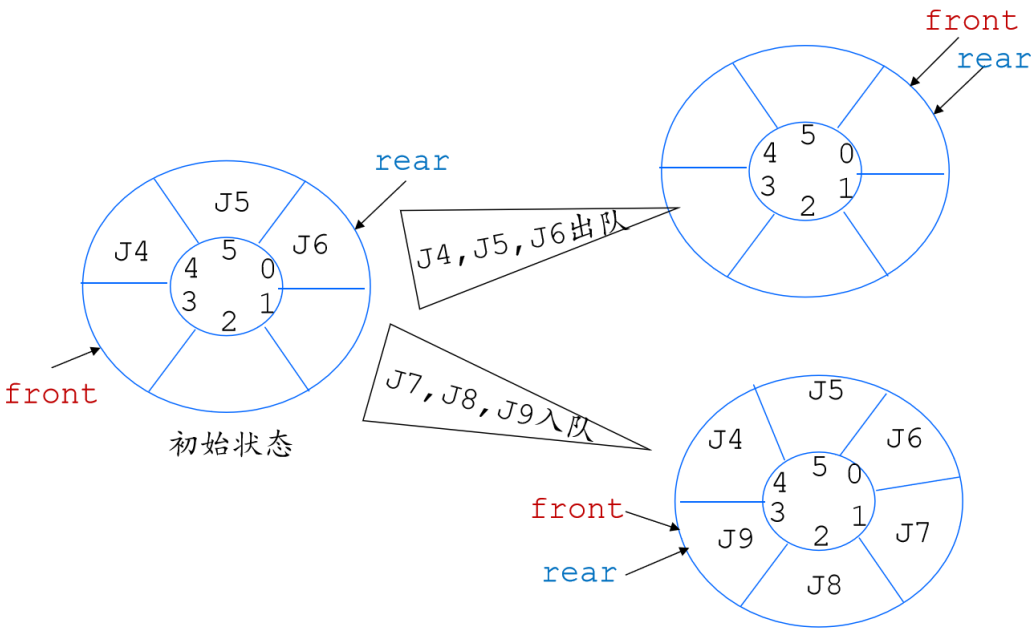
  视为“循环顺序队列”实现方法

  front=(front+1)% MAXQUEUE

  rear=(rear+1)% MAXQUEUE

循环队列假上溢现象：

问题：队满和队空时，均有sq-> front == sq-> rear。因此，只凭sq-> front == sq-> rear还无法区分是满还是空。



解决方法:

- 方法一

  设置一个标志位以区别队空/队满

```
初始化队列时
sq-> front = sq-> rear, 标志位flag=false

入队后, 置标志位 flag = true
出队后, 置标志位 flag = false

front == rear && flag == true      //队满
front == rear && flag == false     //队空
```

- 方法二

  用一个计数变量来记载队列中的元素个数

```
初始化队列时c=0;
当入队时, 计数变量+1 ( c++)
当出队时, 计数变量-1 (c--)
当计数变量 c=maxsize 时, 队满
当计数变量 c=0 时, 队空
```

- 方法三

  少用(牺牲)一个元素的存储空间

```
队空: front= =rear
队满: (rear+1)%M= =front
```

数组

数组是可以看作线性表的推广。

- 一维数组可以看作一个线性表
- 二维数组可以看作"数据元素是一维数组"的一维数组

- 三维数组可以看作“数据元素是二维数组”的一维数组，依此类推

数组的表示和实现：

数组是多维的结构，而存储空间是一个一维的结构。对多维数组分配时，要把它的元素映象存储在一维存储器中，一般有两种顺序映象的方式

- 以行为主序:如BASIC、PASCAL、COBOL、C等程序设计语言中用的是以行为主的顺序分配，即一行一行地分配
- 以列为主序:如FORTRAN语言中，用的是以列为主的顺序分配，即一列一列地分配

第三章 树和二叉树

> 如何构建平衡二叉树

二叉树基础

基本概念

二叉树是n($n \geq 0$)个具有相同类型的数据元素的有限集合

- 当n=0时称为空二叉树
- 当n>0时，数据元素分为：一个称为根的数据元素和两棵分别称为左子树和右子树的数据元素的集合，左、右子树互不相交，并且他们也都是二叉树

定义：二叉树或为空树，或是由一个根结点加上两棵分别称为左子树和右子树的、互不交的二叉树组成

特点：

1. 每个结点最多只有两棵子树，即不存结点度大于2的结点
2. 子树有左右之分，不能颠倒

二叉树术语

- 结点的度：结点所拥有的子树的个数
- 叶子：度为0的结点
- 孩子：结点子树的根
- 双亲：孩子结点的上层结点
- 子孙：以某结点为根的子树中的任一结点
- 祖先：从根到该结点所经分支上的所有结点
- 结点的层次：从根结点起，根为第一层，它的孩子为第二层 ，孩子的孩子为第三层
- 兄弟 ：同一双亲的孩子互为兄弟
- 堂兄弟：其双亲在同一层的结点互为堂兄弟
- 二叉树的度：二叉树中最大的结点度数
- 二叉树的深度：二叉树中结点的最大层次数
- 满二叉树：指的是深度为k且含有2^k-1个结点的二叉树。满二叉树中所有分支结点都存在左子树和右子树，并且所有叶子结点都在同一层上。
- 完全二叉树：树中所含的 n 个结点和满二叉树中编号为 1 至 n 的结点一一对应。

二叉树性质

性质1：二叉树的第 i 层上至多有2^{i-1} 个结点。 $(i \geq 1)$

证明(归纳法)：

归纳基 $i=1$ 层时，只有一个根结点：

$$2^{i-1}=2^0=1;$$

归纳假设：假设对所有的 j，$1 \leq j < i$，命题成立；

归纳证明：二叉树上每个结点至多有两棵子树，则第 i 层的结点数

$$2^{i-2} \times 2 = 2^{i-1}$$

证毕。

性质2：深度为 k 的二叉树上至多含 2^k-1个结点 。$(k \geq 1)$

证明：

基于性质1，深度为 k 的二叉树上的结点数至多为：

$$2^0+2^1+\cdots\cdots+2^{k-1}=2^k-1$$

证毕。

性质3：叶结点与双分支结点的关系为对任何一棵二叉树T，设叶子结点数为n_0，度为2的结点数为n_2，那么，$n_0=n_2+1$。

证明：

假设度为 1 的结点数为n_1，二叉树总结点数为n，那么：

$$n=n_0+n_1+n_2$$

结点个数 n 与边数 e 满足关系

$$e=n-1$$

分支边数又节点引出，1度节点引出1条边，2度节点引入2条边，0度节点引出0条边，那么总的边数 e 可以表达为：

$$e=0\times n_0+1\times n_1+2\times n_2=n_1+2n_2$$

联立三个等式，得

$$n_0=n_2+1$$

证毕。

性质4：具有 n 个结点的完全二叉树的深度为$k=\lfloor log_2^n \rfloor+1$

证明：

深度为k的完全二叉树最少有 2^{k-1} 个结点，最多有 2^k-1 个结点；

因此结点数 n 满足

$$2^{k-1}\le n<2^k$$

两边取对数得

$$k-1\le \log_2 n<k$$

则

$$k\le \log_2 n+1<k+1$$

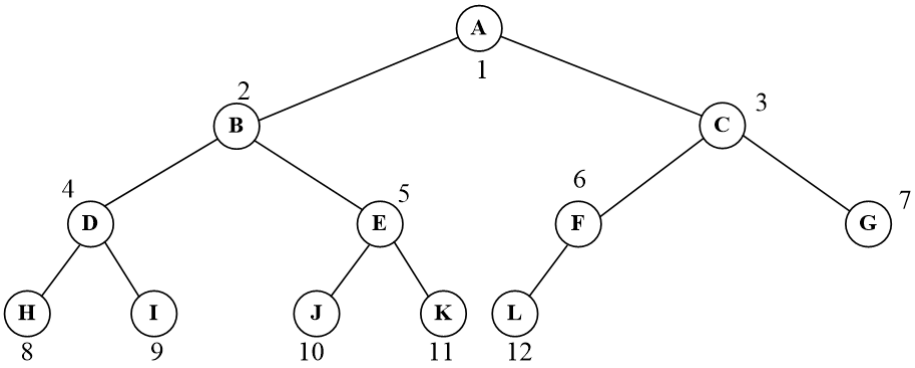
整理得

$$k=\lfloor \log_2 n \rfloor+1$$

证毕。

性质5：对有n个结点的完全二叉树的结点按层序编号，则对任一结点 i ($1\le i\le n$)

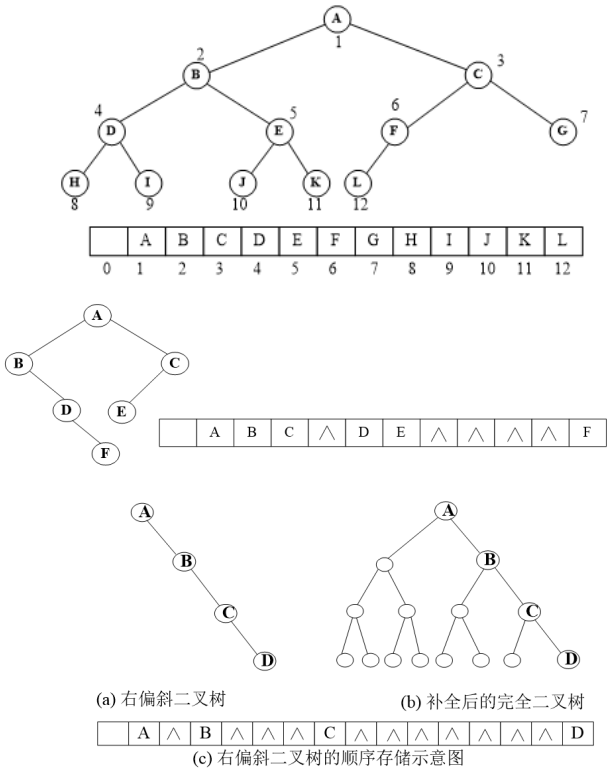
- 如果 $i=1$，则结点 i 是二叉树的根，无双亲；如果 i > 1，则其双亲是 $\lfloor i/2 \rfloor$
- 如果 $2i>n$，则结点 i 无左孩子；如果 $2i\le n$，则其左孩子是$2i$
- 如果 $2i+1>n$，则结点 i 无右孩子；如果$2i+1\le n$，则其右孩子是$2i+1$



二叉树的存储

顺序存储：

按完全二叉树顺序存储。

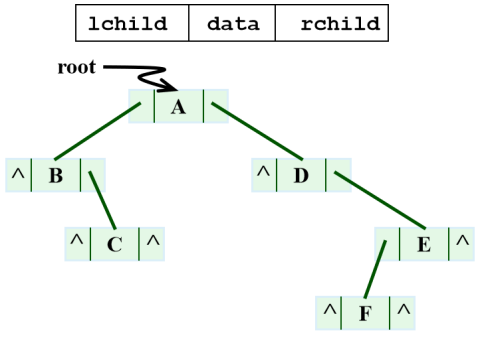


链式存储

二叉链表：

定义二叉链表代码：

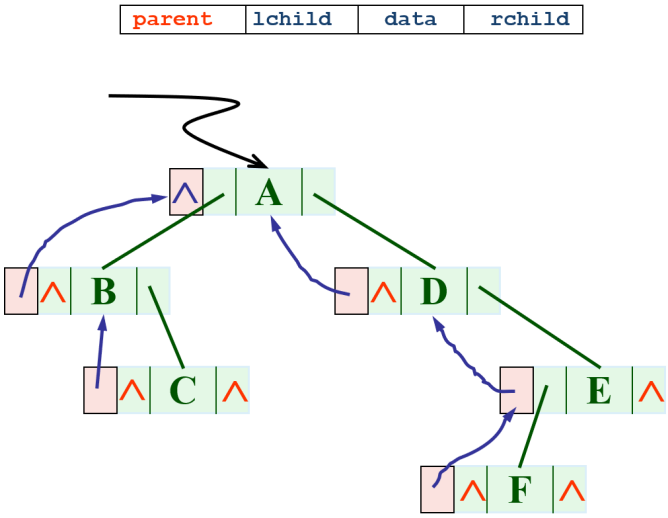
```
typedef struct BiTreeNode
{
    Datatype data;
    struct BiTreeNode * lchild;
    struct BiTreeNode * rchild;
}BiTreeNode, *BiTree;
```



三叉链表：

定义三叉链表代码：

```c
typedef struct BiTreeNode
{
    Datatype data;
    struct BiTreeNode * lchild;
    struct BiTreeNode * rchild;
    struct BiTreeNode * parent;
}BiTreeNode, *BiTree;
```



二叉树的遍历

问题：顺着某一条搜索路径巡访二叉树中的结点，使得每个结点均被访问一次，而且仅被访问一次。

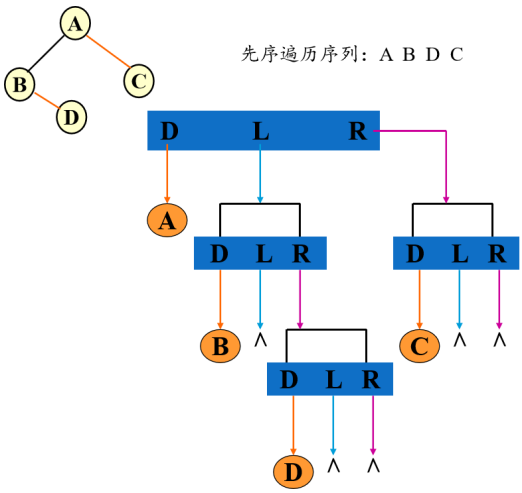
先序遍历

方法：若二叉树非空，则

1. 访问根结点
2. 先序遍历左子树
3. 先序遍历右子树

代码：

```c
void PreOrder( BiTree bt)
{
   if (bt!=NULL)
      {
         visit (bt->data);
         PreOrder( bt->lchild);
         PreOrder (bt->rchild);
      }
}
```



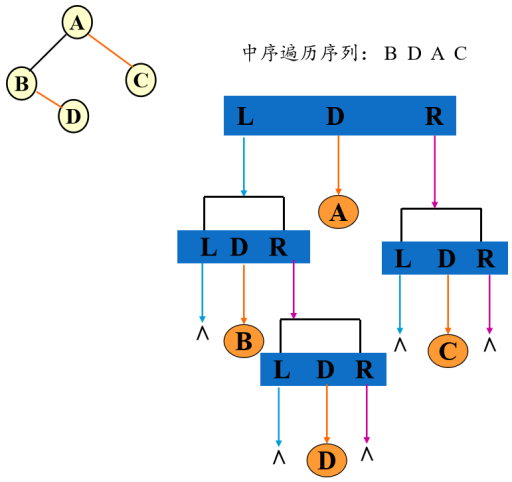
中序遍历

方法：若二叉树非空，则

1. 中序遍历左子树
2. 访问根结点
3. 中序遍历右子树

代码：

```
void InOrder( BiTree bt)
{
   if (bt!=NULL)
      {
         InOrder ( bt->lchild);
      visit (bt->data);
         InOrder (bt->rchild);
      }
}
```



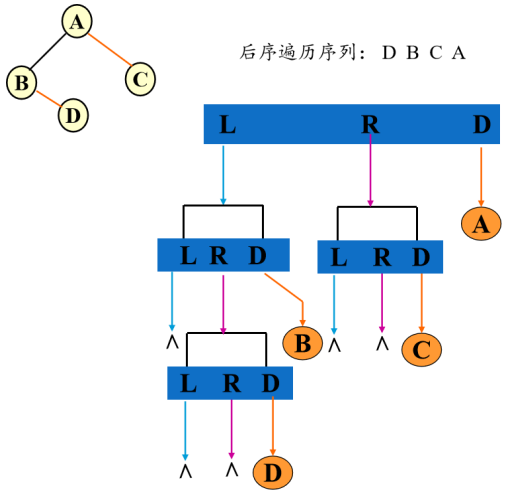
后序遍历

方法：若二叉树非空，则

1. 后序遍历左子树
2. 后序遍历右子树
3. 访问根结点

代码：

```c
void PostOrder( BiTree bt)
{
   if (bt!=NULL)
      {
        PostOrder( bt->lchild);
        PostOrder (bt->rchild);
        visit (bt->data);
      }
}
```



层次遍历

二叉树的层次遍历是指从二叉树的根结点开始，从上到下逐层遍历，同一层中从左到右访问二叉树的结点。

方法：

- 初始化：根结点入队
- 算法迭代：从队头取一个元素，每取一个元素，执行如下3个动作：

  访问该元素所指结点

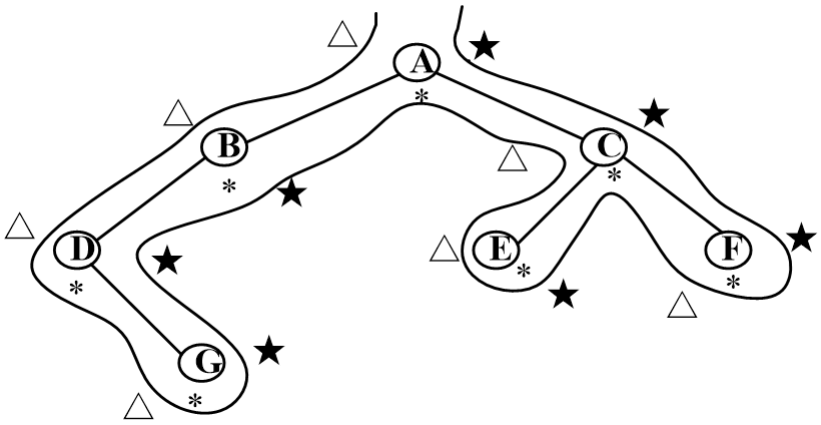
  如果该元素所指结点有左孩子，则左孩子指针入队

  如果该元素所指结点有右孩子，则右孩子指针入队

非递归遍历

先序，中序，后序都是沿着图中路线进行：从树根开始沿左子树一直深入，直到最左端无法深入时，返回，进入刚深入时遇到结点的右子树，再进行如此的深入和返回，直到最后从根结点的右子树返回到根结点为止。（深入返回的过程满足栈的特征，可用栈实现二叉树的遍历）

- 先序：遇到结点就访问 △
- 中序：左子树返回时访问 *
- 后序：右子树返回时访问 ☆



二叉树中序遍历非递归算法：

```
void NRInOrder(BiTree bt)
{
    BiTree S[MAXNODE], p=bt; /*定义栈S*/
    int top=-1;
    while(!(p==NULL && top==-1))
    {/*p空同时栈空结束*/
         while(p!=NULL)
            { /* 找最左端结点,沿途结点入栈 */
                    S[++top]=p;
                    p=p->lchild;
           }
        p=S[top--];           /*弹出栈顶元素*/
        visit(p->data);      /*访问结点*/
        p=p->rchild;         /*指向p的右孩子结点*/
    }
}
```

二叉树的应用

应用-1：求深度

问题：已知根节点指针，求二叉树的深度。

- 空树：深度 = 0
- 左右子树为空：深度 = 1
- 其它 ： 深度 = 1+max(左子树深度，右子树深度)

代码：

```c
int PostTreeDepth(BiTree bt)
{
    int hl,hr,max;
    if(bt!=NULL)
    {
        hl = PostTreeDepth(bt->LChild);
        hr = PostTreeDepth(bt->RChild);
        max = hl > hr ? hl : hr;
        return(max+1);
    }
    else
        return(0);
}
```

应用-2：求叶子数

问题：已知根节点指针，求二叉树的叶子数。

- 空树：叶子数 = 0
- 左右子树为空：叶子数 = 1
- 其它： 叶子数 = 左子树叶子数 + 右子树叶子数

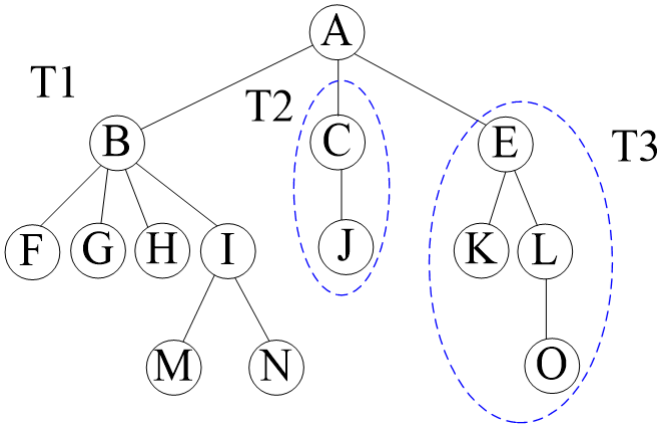
代码：

```c
int LeafCount(BiTree root)
{
    int leafcount;
    leafcount = 0;
    if(root==NULL)
        leafcount = 0;
    else
        if((root->LChild==NULL)&&(root->RChild==NULL))
            leafcount =1;
        else{
            leafcount = LeafCount(root->LChild) + LeafCountn(root->RChild);
        }
    return leafcount;
}
```

树和森林

普通的树

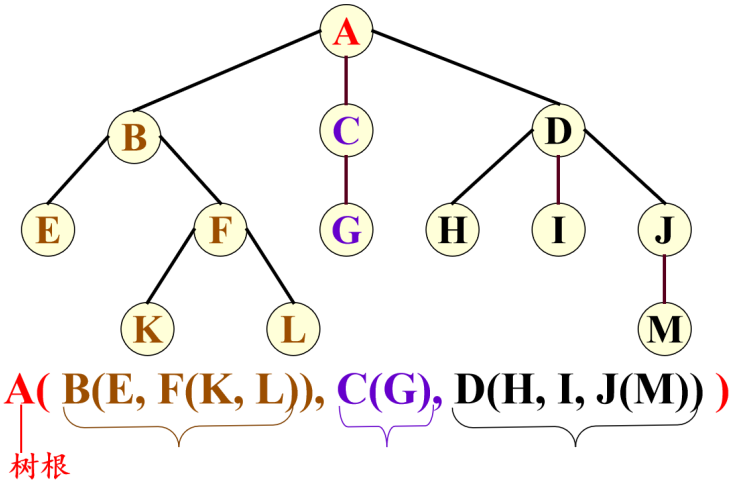
树的根结点没有双亲，除根结点以外，其他每个结点都有且仅有一个双亲。树中所有结点有零个或多个孩子结点。树是一种一对多的层次结构。



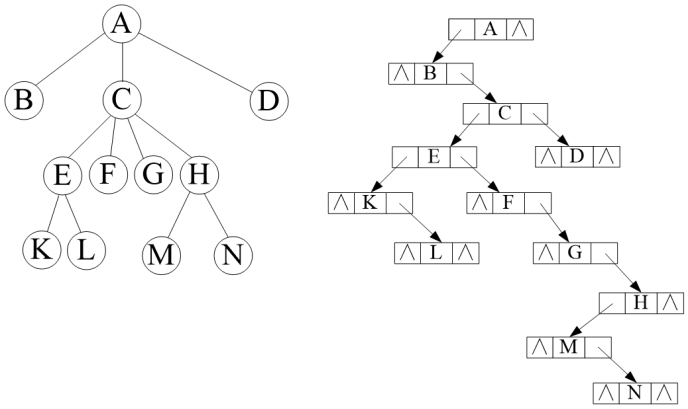
森林

森林是m（$m \geq 0$）棵互不相交的树的集合

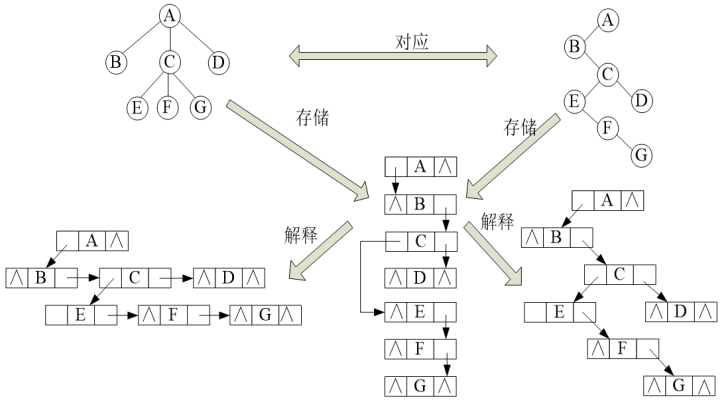
树的广义表表示方法：



树的孩子兄弟表示法：



树和二叉树的相互转换：



特殊的二叉树

线索二叉树（Thread Binary Tree, TBT）

问题：能够通过结点的两个链域查找出任一结点的前驱和后继

方法：

- 若结点有左孩子，则其lchild指示其左孩子，否则，令lchild域指示其前驱
- 若结点有右孩子，则其rchild指示其右孩子，否则，令rchild域指示其后继

定义线索二叉树代码：

```
typedef struct BiThrNode
{
    Datatype data;
    struct BiThrNode *lchild, *rchild;
    byte LTag, RTag;
}BiThrNode, *BiThrTree;
```

lchild	LTag	data	RTag	rchild

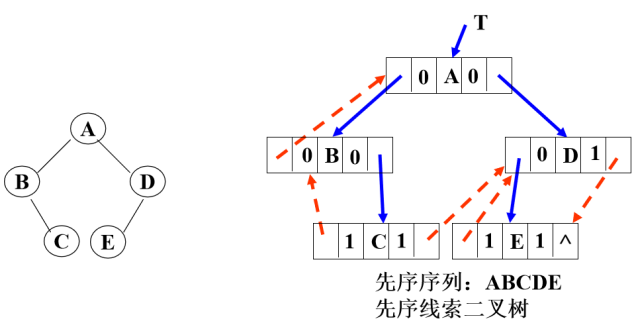
$$LTag=\begin{cases}0 & \text{lchild 域指示结点的左孩子}\\ 1 & \text{lchild域指示结点的前驱}\end{cases}$$

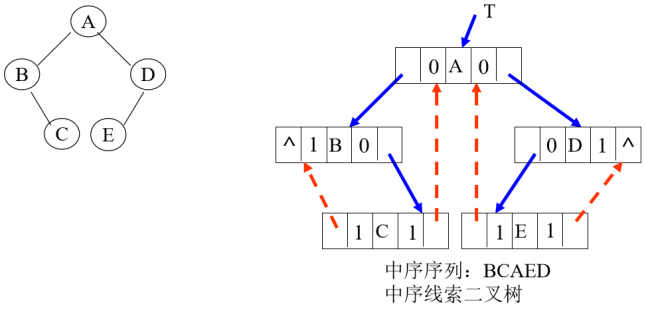
$$RTag=\begin{cases}0 & \text{rchild域指示结点的右孩子}\\ 1 & \text{rchild域指示结点的后继}\end{cases}$$

以这种结点结构构成的二叉链表作为二叉树的存储结构，叫做线索链表，其中指向前驱和后继的指针，叫做线索(Thread)。

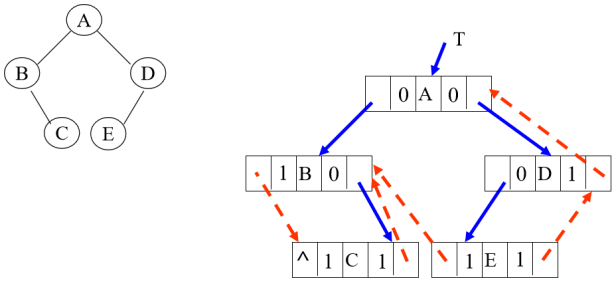
加上线索的二叉树叫做线索二叉树(Thread Binary Tree)。

对二叉树以某种次序遍历使其变为线索二叉树的过程叫做线索化。





中序序列：BCAED
中序线索二叉树



后序序列：CBEDA
后序线索二叉树

二叉排序树（Binary Sort Tree, BST）

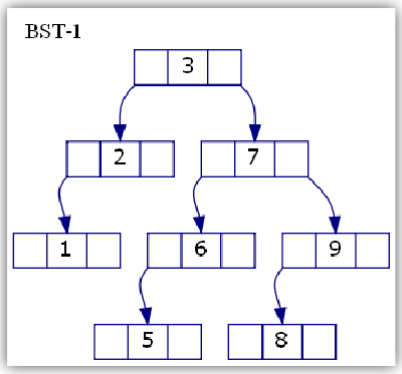
定义：二叉排序树或者是一棵空树；或者是具有如下特性的二叉树：

- 若根结点的左子树不空，则左子树上所有结点的值均小于根结点的值
- 若根结点的右子树不空，则右子树上所有结点的值均大于根结点的值
- 左、右子树本身也是一棵二叉排序树。

查找

BST的查找就是从根出发不断比较和分支的过程。

问题：在二叉排序树 BST-1 中分别查找元素3、5和10。



方法：

- 查找元素3：从根出发， 1次比较即命中。
- 查找元素5：从根出发，经过3, 7, 6, 5，共4次比较命中。
- 查找元素10：从根出发，经过3, 7, 9，发现9的右子树为空，共三次比较且查找失败 。

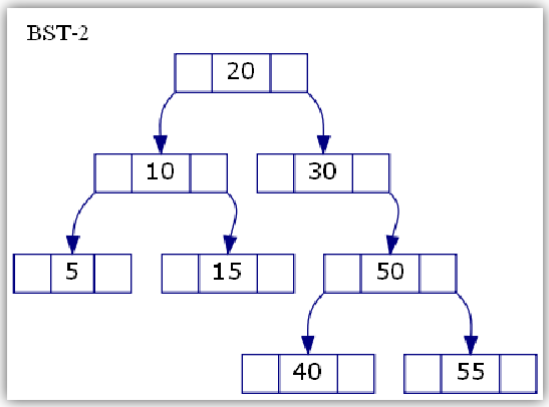
性能分析：如何在构造中降低BST的高度是改进的方向。

构造

BST的构造就是一个不断查找并添加叶子的过程。

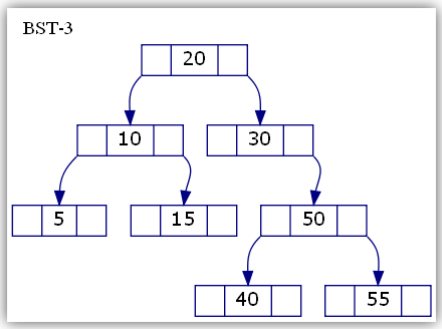
问题：给定数据序列{20 30 50 10 40 55 15 5}，请依输入序构造出一棵二叉排序树。

方法：



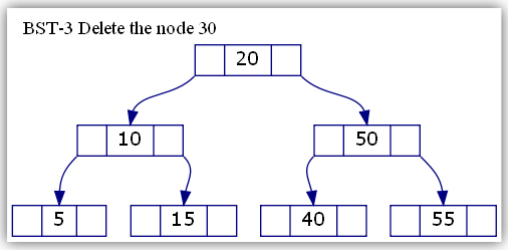
删除

问题：考虑在二叉排序树 BST-3 中分别删除节点15, 30, 20。



方法：

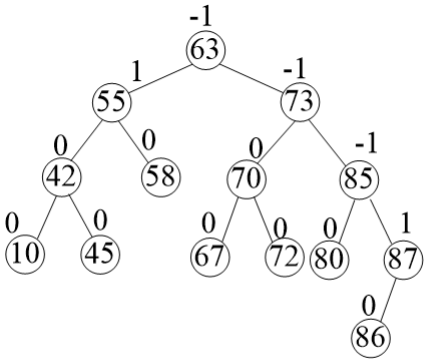
- 删除叶子节点，直接删除之。
- 删除度为1的节点，以其非空子节点替代之。
- 删除度为2的节点，以其前驱节点替代之。



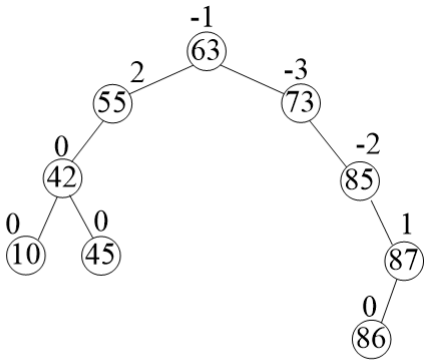
平衡二叉树（ADELSON-VELSKII and LANDIS, AVL）

定义：平衡二叉树或者是一棵空树，或者是具有下列性质的二叉树：

- 它的左、右子树都是平衡二叉树
- 它的左、右子树的深度之差不超过1



(a) 平衡二叉树



(b) 非平衡二叉树

平衡二叉树的构造：

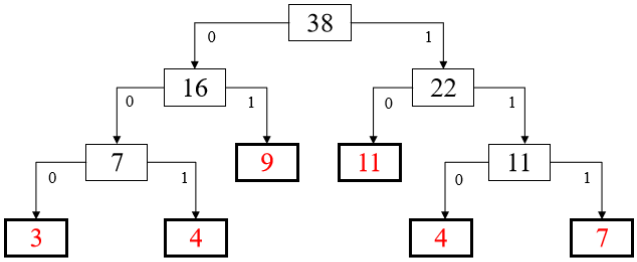
- 单向右旋（LL）
- 单向左旋（RR）
- 先左后右旋转（LR）
- 先右后左旋转（RL）

相同方向生长引发不平衡(LL或RR)，升级失衡节点的儿子节点为新的根。

不同方向生长引发不平衡(LR或RL)，升级失衡节点的孙子为儿子，新儿子升级为根。

最优二叉树（Optimal Binary Tree, Haffman Tree, Haffman-T)

定义：哈夫曼树又被称为最优二叉树，它是一棵带权路径长度(WPL)最短的树。



[David Huffman, 1952]

问题：构造哈夫曼树

方法：

- 从集合中选择两个最小的数，合并成根
- 用新根替换被选中的两个数

堆积树（Heap Tree, HeapT)

$$\begin{cases} value[i] \geq value[2 \times i] \\ value[i] \geq value[2 \times i + 1] \end{cases}$$

被称为大顶堆

$$\begin{cases} value[i] \leq value[2 \times i] \\ value[i] \leq value[2 \times i + 1] \end{cases}$$

被称为小顶堆

问题：构造堆积树

方法：

- 宏观上，将所有节点自下而上，进行筛选
- 微观上，将操作节点自上而下，进行梳理

第四章 图和广义表

图论基础

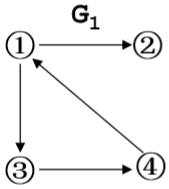
图的分类

有向图(Digragh)

G=(V, {A})

V为顶点的有穷非空集合

{A}为顶点之间的关系集合



G_1=(V，{A})

$V = \{v_1, v_2, v_3, v_4\}$

$A = \{<v_1,v_2>, <v_1,v_3>, <v_3,v_4>, <v_4,v_1>\}$

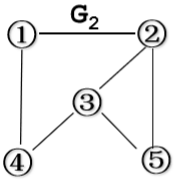
其中<x,y>表示从x到y的一条弧(arc)，A为弧集合，x为弧尾(tail)，y为弧头(head)

无向图(Undigraph)

G=(V, {E})

V为顶点的有穷非空集合

{E}为顶点之间的关系集合



G_2 = (V，{A})

$V = \{v_1, v_2, v_3, v_4, v_5\}$

$E = \{(v_1, v_2), (v_1, v_4), (v_2, v_3), (v_3, v_4), (v_2,v_5), (v_3,v_5)\}$

其中(x, y)表示x与y之间的一条连线，称为边(edge)

图的边与顶点的关系

设 n 为顶点数，e 为边或弧的条数，

无向图有：$0 \leq e \leq n \cdot (n-1)/2$

有向图有：$0 \leq e \leq n \cdot (n-1)$

完全图是弧或边达到最大的图：

- 无向完全图：边数为$n \cdot (n-1)/2$的无向图
- 有向完全图：弧数为$n \cdot (n-1)$的有向图

权：图的边或弧上搭载的权重数据

网：当图上的边或弧上带有权值时，图称为网

顶点的度

顶点的度(Total Degree，TD)定义：

- 无向图：为依附于顶点v的边数
- 有向图：等于以顶点v为弧头的弧数(称为v的入度，记为$ID(v)$)与以顶点v为弧尾的弧数(称为v的出度，记为$OD(v)$)之和。即$TD(v) = ID(v) + OD(v)$

那么对于无向图和有向图分别有:

$$e=\frac{1}{2}\sum_{i=1}^{n}\mathrm{TD}(v_i) \qquad e=\sum_{i=1}^{n}\mathrm{ID}(v_i)=\sum_{i=1}^{n}\mathrm{OD}(v_i)$$

图之路径

图的路径定义：

- 无向图：顶点v到v'的路径是一个顶点序列

$$(v,v_{i[0]},v_{i[1]},\cdots,v_{i[m]},v') \quad \text{s.t. } (v_{i[k-1]},v_{i[k]})\in \mathrm{E},\ 1\le k\le \mathrm{m}$$

- 有向图：顶点v到v'的路径是有向的顶点序列

$$(v,v_{i[0]},v_{i[1]},\cdots,v_{i[m]},v') \quad \text{s.t. } <v_{i[k-1]},v_{i[k]}>\in \mathrm{E},\ 1\le k\le \mathrm{m}$$

路径长度：路径上边或弧的数目。

回路或环：首尾顶点相同的路径，称为回路或环。即：

$$(v,v_{i[0]},v_{i[1]},\cdots,v_{i[m]},v) \quad \text{s.t. } <v_{i[k-1]},v_{i[k]}>\in \mathrm{E},\ 1\le k\le \mathrm{m}$$

简单路径：路径中不含相同顶点的路径。

简单回路：除首尾顶点外，路径中不含相同顶点的回路。

图之连通

顶点连通：若顶点v到顶点v'有路径，则称顶点v与v'是连通的。

连通图：若图中任意两个顶点vi, vj都是连通的，则称该图是连通图。

连通分量：

- 无向图中极大连通子图，称为连通分量
- 有向图中极大强连通子图，称为强连通分量

图之生成树和子图

树是图的极小连通子图，如果再增加一条边必构成环。

子图是图的一部分，它本身也是一个图。如果有图$G=(V，E)$和$G\prime=(V\prime,E\prime)$，且V'是V的子集，E'是E的子集，则称G'是G的子图。

图上顶点的邻接:

- 无向图中，两个顶点有边连接，则两个顶点互为邻接。
- 有向图中，两个顶点有弧连接，则接收方邻接发送方。

图的储存

邻接矩阵

设图$G=(V，E)$有n个顶点，则G的邻接矩阵定义为n阶方阵A。

$$A[i,j]=\begin{cases}1 & 若(\mathrm{v_i},\mathrm{v_j})或<\mathrm{v_i},\mathrm{v_j}>是图G的边\\ 0 & 若(\mathrm{v_i},\mathrm{v_j})或<\mathrm{v_i},\mathrm{v_j}>不是图的边\end{cases}$$

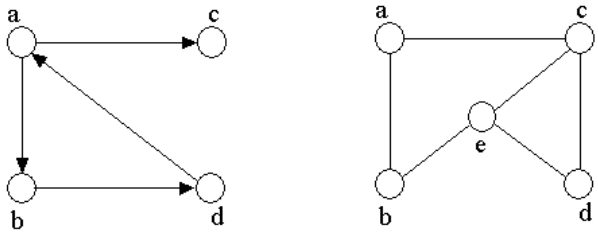


图4.8 有向图G1和无向图G2

$$A_1=\begin{bmatrix}0&1&1&0\\0&0&0&1\\0&0&0&0\\1&0&0&0\end{bmatrix}\qquad A_2=\begin{bmatrix}0&1&0&1&0\\1&0&1&0&1\\0&1&0&1&1\\1&0&1&0&0\\0&1&1&0&0\end{bmatrix}$$

判定两个顶点v_i与v_j是否关联：只需判$A[i][j]$是否为1

求顶点的度：

- 无向图中：

$$\mathrm{TD}(v_i)=\sum_{j=1}^{n}A[i][j]=\sum_{j=1}^{n}A[j][i]$$

即顶点v_i的度等于邻接矩阵中第i行(或第i列)的元素之和(非0元素个数)。

- 有向图中：

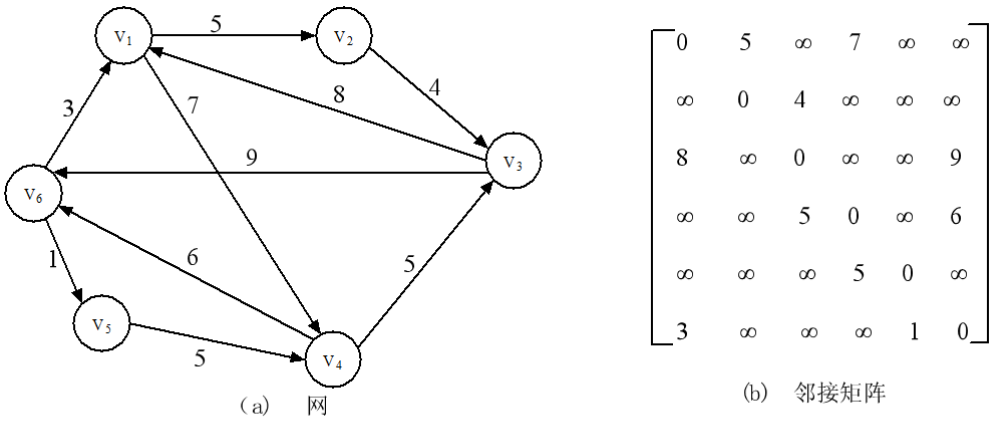
$$\mathrm{TD}(v_i)=\mathrm{OD}(v_i)+\mathrm{ID}(v_i)=\sum_{j=1}^{n}A[i][j]+\sum_{j=1}^{n}A[j][i]$$

即顶点v_i的出度为邻接矩阵中第i行元素之和顶点v_i的入度为邻接矩阵中第i列元素之和。

带权图邻接矩阵

如果G是带权图，$w_{i,j}$是边(v_i,v_j)或$<v_i,v_j>$的权，则其邻接矩阵定义为：

$$A[i][j]=\begin{cases}w_{i,j} & (v_i,v_j)或<v_i,v_j>是G的边(i\neq j)\\ \infty & (v_i,v_j)或<v_i,v_j>不是G的边(i\neq j)\\ 0 & (i=j)\end{cases}$$



(a) 网　　(b) 邻接矩阵

图4.9　网及其邻接矩阵

邻接表

对图中每个顶点v_i建立一个单链表，链表中的结点表示依附于顶点v_i的边，每个链表结点为两个域：

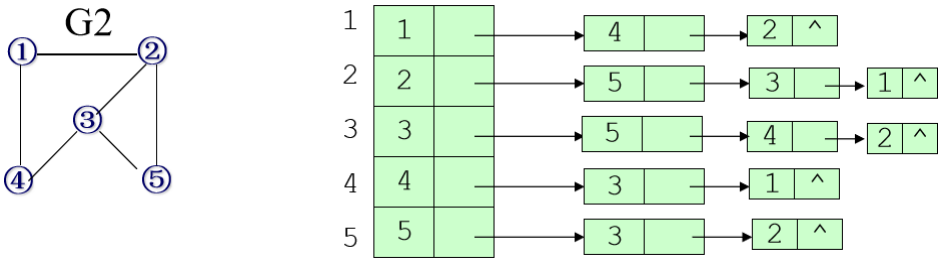
adjvex	nextarc

- 邻接点域(adjvex)：记载与顶点vi邻接的顶点信息
- 链域(nextarc)：指向下一个与顶点vi邻接的链表结点

每个链表附设一个头结点，头结点结构为：

vexdata	firstarc

- vexdata：存放顶点信息(姓名、编号等)
- fristarc：指向链表的第一个结点



无向图邻接表特点：

1. n个顶点，e条边的无向图，需n个头结点和2e个链表结点；
2. 顶点v_i的度$TD(v_i) = 链表i中的链表结点数$。

有向图邻接表特点：

1. n个顶点，e条弧的有向图，需n个表头结点，e 个链表结点；
2. 求顶点的出度易，求入度难。第 i 条链表上的链表结点数，为v_i的出度。

十字链表

十字链表是将有向图的邻接表和逆邻接表结合起来的一种有向图链式存储结构。有向图的每一条弧有一个弧结点，每一个顶点有一个顶点结点。

顶点结点

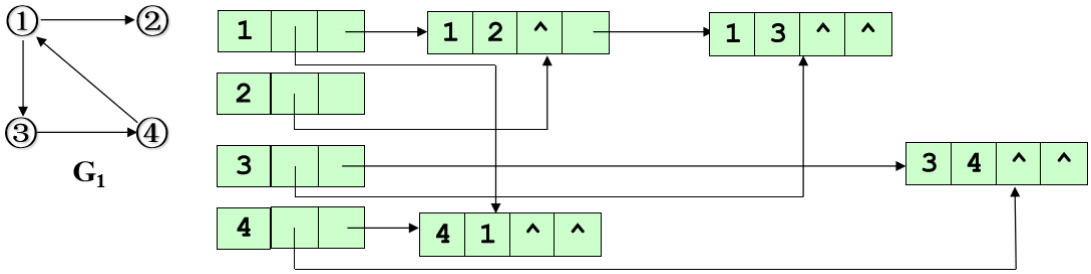
data	firstin	firstout

data:存放顶点的有关信息;
firstin:指向以该顶点为弧头的第一个弧结点;
firstout:指向以该顶点为弧尾的第一个弧结点。

弧结点

tailvex	headvex	hlink	tlink

tailvex:指示该弧的弧尾顶点;
headvex:指示该弧的弧头顶点;
hlink:指向弧头相同的下一条弧;
tlink: 指向弧尾相同的下一条弧。



图的遍历

从图中某个顶点出发，沿路径使图中每个顶点被访问且仅被访问一次的过程，称为图的遍历。

深度优先搜索(depth-first-search)

1. 访问指定的当前顶点顶点v
2. 设置新的当前顶点v为未被访问邻接点
3. 重复2直到当前顶点v的所有邻接点都被访问
4. 沿搜索路径回退，退到尚有邻接点未被访问过的某结点，将该结点作为新的当前结点v，重复2，直到所有顶点被访问过的为止

广度优先搜索(breadth-first-search)

1. 访问指定的当前顶点顶点v
2. 访问当前顶点的所有未访问过的邻接点
3. 依次将访问的这些邻接点作为当前顶点v
4. 重复2， 直到所有顶点被访问为止

图论算法

- Prim算法(贪心算法)
- Kruskal算法(贪心算法)
- 拓扑排序算法
- 关键路径(标号法)
- 单源Dijkestra算法(动态规划)
- 全源Floyd算法(矩阵自乘)

无向图求最小生成树

生成树：设无向连通图$G=(V，E)$，其子图$G'=(V，T)$满足：

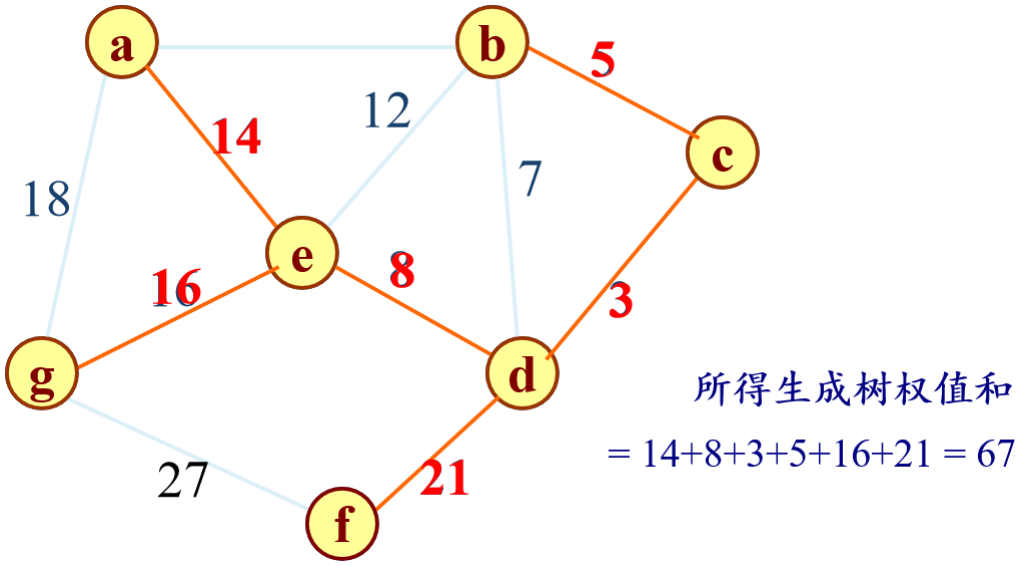
1. $V(G')=V(G)$ n个顶点;
2. G'是连通的;
3. G'中无回路，

则G'是G的生成树。

最小生成树（MST）：设$G=(V，E)$，U是顶点集V的一个非空子集。$u\in U，v\in V-U$，若(u,v)是一条具有最小权值的边，即：$(u,v)=argmincost(x,y)|x\in U,y\in V-U$，则必存在一棵包含边$(u,v)$的最小生成树。

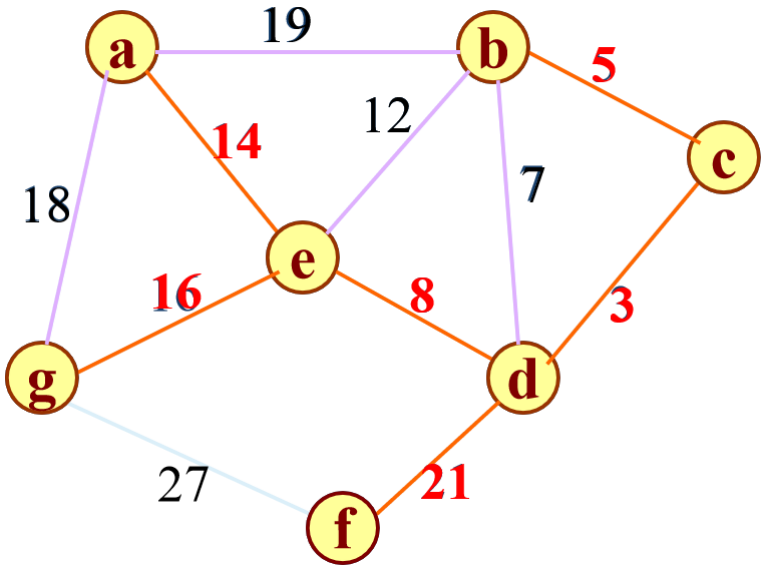
Prim算法

取图中任意一个顶点 v 作为生成树的根，之后往生成树上添加新的顶点 w。在添加的顶点 w 和已经在生成树上的顶点 u 之间必定存在一条边，并且该边的权值在所有连通顶点 u 和 w 之间的边中取值最小。之后继续往生成树上添加顶点，直至生成树上含有 n 个顶点为止。



Kruskal算法

假设连通网$G=（V，E）$，令最小生成树的初始状态为只有n个顶点而无边的非连通图$T=（V，）$，概述图中每个顶点自成一个连通分量。在E中选择代价最小的边，若该边依附的顶点分别在T中不同的连通分量上，则将此边加入到T中；否则，舍去此边而选择下一条代价最小的边。依此类推，直至T中所有顶点构成一个连通分量为止。



普里姆最小生成树算法

- 以连通为主；
- 选保证连通的代价最小的邻接边；
- 普里姆算法的时间复杂度与边无关，为$O(n^2)$；
- 适合于求边稠密网的最小生成树。

克鲁斯卡尔最小生成树算法

- 以最小代价边主；
- 添加不形成回路的当前最小代价边；
- 算法时间复杂度与边相关，为$O(e\log_2 e)$；
- 适合于求边稀疏网的最小生成树。

有向图环的检查

拓扑排序算法

拓扑排序：是一种对非线性结构的有向图进行线性化的重要手段。按照有向图给出的前驱后继关系，将图中顶点排成一个线性序列，对于有向图中没有限定次序关系的顶点，则可以人为加上任意的次序关系。由此所得顶点的线性序列称之为拓扑有序序列。

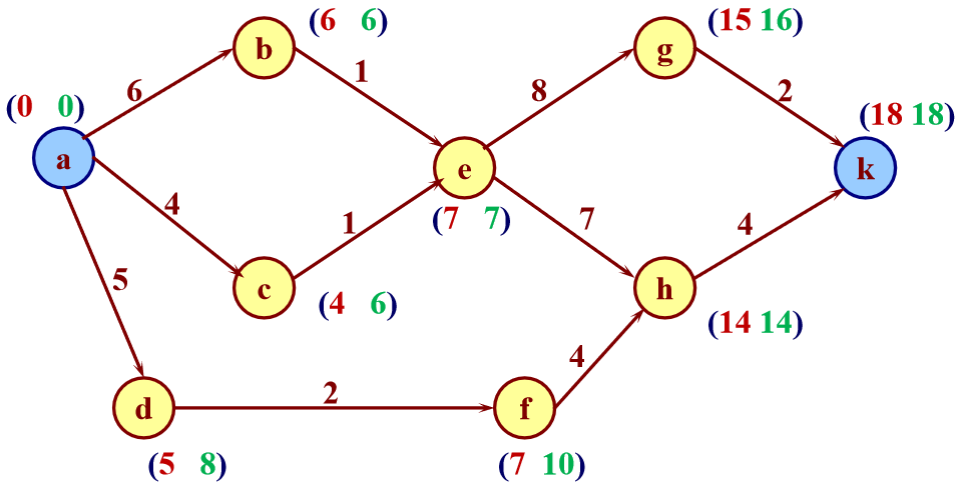
方法：没有前驱的顶点 = 入度为零的顶点，删除顶点及以它为尾的弧 = 弧头顶点的入度减1。

有向无环图(AOV网)

关键路径算法

方法：

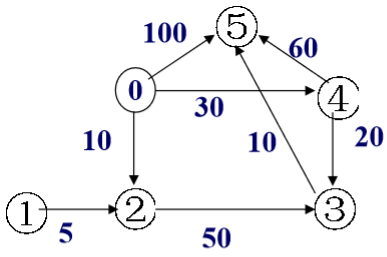
1. 正向求和之最大
2. 逆向求差之最小
3. 正逆二向相等的节点为关键节点



图的最短路径

单源Dijkestra算法

求从某个源点到其余各点的最短路径



cost

	0	1	2	3	4	5
0	∞	∞	10	∞	30	100
1	∞	∞	5	∞	∞	∞
2	∞	∞	∞	50	∞	∞
3	∞	∞	∞	∞	∞	10
4	∞	∞	∞	20	∞	60
5	∞	∞	∞	∞	∞	∞

选最小

终点	从v_0到各终点的dist值和最短路径				
v_1	∞	∞	∞	∞	∞
v_2	$10(v_0,v_2)$				
v_3	∞	$60(v_0,v_2,v_3)$	$50(v_0,v_4,v_3)$		
v_4	$30(v_0,v_4)$	$30(v_0,v_4)$			
v_5	$100(v_0,v_5)$	$100(v_0,v_5)$	$90(v_0,v_4,v_5)$	$60(v_0,v_4,v_3,v_5)$	
v_j	v_2	v_4	v_3	v_5	

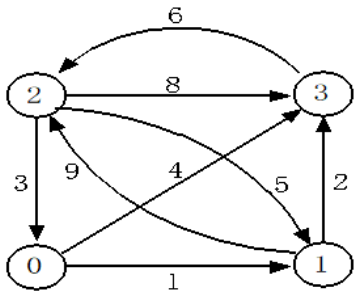
作更新

全源Floyd算法

每一对顶点之间的最短路径

代码：

```
void Floyd(int cost[][],Mgraph Gn,int path[][])
{
    int i,j,k;
    for (i=0;i<n;i++)
    for (j=0;j<n;j++)
    {
        Gn.arcs[i][j]=cost[i][j];
        if(i==j)
            path[i][j]=-1;
        else if(cost[i][j]<maxint)
                path[i][j]=i;
            else path[i][j]= -1;
    }
    for(k=0; k<n; k++)
    for(i=0; i<n; i++)
    for(j=0; j<n; j++)
        if(Gn.arcs[i][j]>( Gn.arcs[i][k]+Gn.arcs[k][j]))
        {
            Gn.arcs[i][j] = Gn.arcs[i][k]+Gn.arcs[k][j];
            path[i][j] = path[k][j];
        }
}
```



$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array}\begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 1 & \infty & 4 \\ \infty & 0 & 9 & 2 \\ 3 & 5 & 0 & 8 \\ \infty & \infty & 6 & 0 \end{bmatrix} \end{array}$$

表4.6 弗洛伊德Floyd算法求解过程

	$D^{(-1)}$				$D^{(0)}$				$D^{(1)}$				$D^{(2)}$				$D^{(3)}$			
	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	0	1	∞	4	0	1	∞	4	0	1	**10**	**3**	0	1	10	3	0	1	**9**	3
1	∞	0	9	2	∞	0	9	2	∞	0	9	2	**12**	0	9	2	**11**	0	**8**	2
2	3	5	0	8	3	**4**	0	**7**	3	4	0	**6**	3	4	0	6	3	4	0	6
3	∞	∞	6	0	∞	∞	6	0	∞	∞	6	0	**9**	**10**	6	0	9	10	6	0
	$Path^{(-1)}$				$Path^{(0)}$				$Path^{(1)}$				$Path^{(2)}$				$Path^{(3)}$			
	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	-1	0	-1	0	-1	0	-1	0	-1	0	**1**	**1**	-1	0	1	1	-1	0	**3**	1
1	-1	-1	1	1	-1	-1	1	1	-1	-1	1	1	**2**	-1	1	1	**2**	-1	**3**	1
2	2	2	-1	2	2	**0**	-1	**0**	2	0	-1	**1**	2	0	-1	1	2	0	-1	1
3	-1	-1	3	-1	-1	-1	3	-1	-1	-1	3	-1	**2**	**0**	3	-1	2	0	3	-1

广义表基础

定义：广义表也是常用的一类数据结构(第四类)。它与图一样，可以反映数据元素之间多对多的关系。广义表也称为列表，甚至可以简称为表。

- 广义表可记为：$LS=(d1,d2,\ldots,dn)$

  LS为表名，n为表的长度，当长度 n = 0 时称为空表；非空表的第一个元素d_1称为表头，其余元素组成的表$(d_2,\ldots,d_n)$称为表尾。

  数据元素$d_i(i=1,2,\ldots,n)$，可以是单元素(用小写字母表示)；也可以是广义表(称为子表，用大写字母表示)

- 广义表的深度是指该广义表展开后所含括号的层数

广义表可以是空的，可以是嵌套的，可以是递归的，可以是无穷的。

A = ( )

空表A，其长度为0。

B = (a，(b，c))

非空表B，长度为2，深度为2，
其中第一个数据元素是单元素a，第二个数据元素是一个子表(b，c)。

C =(A，A，B)

非空表C，长度为3，深度为3，
其前两个元素为表A，第三个元素为B

D=(a，D)

非空表D，长度为2，递归定义的广义表，深度为无穷
D相当于无穷表D=(a，(a，(a，(…))))

运算：

1. 取表头 head(L)
2. 取表尾 tail(L)

广义表 B = (a, (b, c))

head(*B*) = a

tail(*B*) = ((b, c))

第五章 查找

在数据集合上的查找涉及到两个主要问题：

- 一是数据及其结构是如何组织的：查找表
- 二是在查找表上如何进行查找运算：查找方法

在顺序表上的查找：①顺序查找、②折半查找

在索引表上的查找：③索引查找

在散列表上的查找：④哈希查找

查找分类

静态查找：仅作查询和检索操作的查找。

动态查找：将查询结果“不在查找表中”的数据元素插入到查找表中；或者，从查找表中删除其查询结果为“在查找表中”的数据元素。

关键字：是数据元素中某个数据项的值，用以标识一个数据元素。

- 若关键字能标识唯一的一个数据元素，则称谓主关键字。
- 若关键字能标识若干个数据元素，则称谓次关键字。

查找算法性能的评价

平均查找长度 ASL

$$ASL = p_1 \cdot C_1 + p_2 \cdot C_2 + \ldots + p_n \cdot C_n$$

p_i—— 查找第i个元素的概率；

C_i—— 查找第i个元素需要的比较次数。

不失一般性，我们考虑等概率情况，则

$$p_i = 1 / n$$

那么，对ASL性能的评价重点都在C_i上。

顺序表的查找

顺序查找

查找成功等概率下需要比较$\frac{n+1}{2}$次

查找失败需要比较n次

折半查找

查找成功至多需要比较$log_2 n$次，至少1次

查找失败需要比较$log_2 n$次

- 折半查找的查找效率高；平均查找性能和最坏性能相当接近
- 折半查找要求查找表为有序表；并且，折半查找只适用于顺序存储结构

索引表的查找

索引表的建立

1. 分块：按查找表中数据按关键字分成若干块：$R_1, R_2, ..., R_L$，使得数据“分块有序”；
2. 建立索引项：为每一个块建立一个索引项：

   关键字项(记录块中最大关键字值)

   指针项(记录块的起始地址)
3. 建立索引表：将所有索引项管理起来组成索引表。

索引查找

1. 顺序查找方法在索引表上确定待查数据所在块(块间顺序查找)
2. 在已确定的块中进行顺序查找(快内顺序查找)

索引查找平均查找长度为块间及块内平均查找长度之和。n为表长，均匀分为b块，每块含有s个记录，则索引查找平均查找长度为：

$$\mathrm{ASL}_{bs} = L_b + L_s = \frac{1}{b}\sum_{j=1}^{b} j + \frac{1}{s}\sum_{i=1}^{s} i = \frac{1}{2}\left(\frac{n}{s} + s\right) + 1$$

散列表的查找

将关键字与存储位置之间建立一个函数关系，以 hash(key) 或 H(key) 作为关键字为 key 的记录在表中的位置。通常称这个函数 hash (key) 或 H(key) 为哈希函数。

根据设定的哈希函数 H(key) 和提供的处理冲突的方法，将一组关键字映象到一个地址连续的地址空间上，并以关键字在地址空间中的“象”作为相应记录在表中的存储位置，如此构造所得的查找表称之为哈希表。

哈希函数构造方法

直接哈希函数

取关键字本身或关键字的某个线性函数值作为哈希地址，即：$H(key) = key$或$H(key) = a \cdot key + b$，其中a、b为常数。

数字分析法

关键字各位出现的频率不一定相同，可能在某些位上均匀分布，而在另一些位上分布不均匀。则选择其中分布均匀的s位作为哈希地址，即：$H(key) =$“key中数字均匀分布的s位”。

平方取中法

取关键字平方后的中间几位作为哈希地址，即哈希函数为：$H(key) =$“$key2$的中间几位”，所取的位数由哈希表的大小确定。 “扩大差别”和“贡献均衡”。

移位折叠法

将关键字分割成位数相等的几部分，取这几部分的叠加和作为哈希地址。位数由存储地址的位数确定。适合于: 关键字的数字位数特别多。

$$
\begin{array}{r}
d_r\cdots\cdots\ d_2\ \ d_1 \\
d_{2r}\cdots\cdots d_{r+2}\ \ d_{r+1} \\
+)\ d_{3r}\cdots\cdots d_{2r+2}\ d_{2r+1} \\
\hline
S_r\cdots\cdots\ S_2\ \ S_1
\end{array}
$$

(a) 移位叠加法

$$
\begin{array}{r}
d_r\cdots\cdots\ d_2\ \ d_1 \\
d_{r+1}\cdots\cdots d_{2r-1}d_{2r} \\
+)\ d_{3r}\cdots\cdots d_{2r+2}\ d_{2r+1} \\
\hline
S_r\cdots\cdots\ S_2\ \ S_1
\end{array}
$$

(b) 边界叠加法

除留余数法

取关键字被某个不大于哈希表长度 n 的素数 p 除后所得余数作为哈希地址，即：
$H(key) = key\ MOD\ p \quad s.t.\ p \leq n$。

p 的选择很重要，如果选得不好会产生很多冲突。选择 p ≤ n 的某个素数。

随机数法

选择一个随机函数，取关键字的随机函数值作为哈希地址，即：
$H(key) = random(key) \quad s.t.\ p \leq n$，其中random为随机函数。

冲突处理方法

开放地址法

为产生冲突的地址 H(key) 求得一个地址序列：$H_0, H_1, H_2, \ldots, H_s \quad 1 \leq s \leq m-1$，其中：
$H_0 = H(key), H_i = (H(key) + d_i)\ MOD\ m \quad (i = 1, 2, \ldots, s)$m，为Hash表表长，$d_i$为增量序列。

增量d_i有三种取法：

1. 线性探测再散列：$d_i = c \times i$ 最简单的情况 $c = 1$
2. 平方探测再散列：$d_i = 1^2, -1^2, 2^2, -2^2, \ldots$
3. 随机探测再散列：d_i 是一组伪随机数列

在m=16的哈希表中已有关键字分别为19，70，33三个记录，
哈希函数取为H（key）= key mod 13，
现要把第四个关键字为18的数据存入表中，
由哈希函数得地址为5，产生冲突，三种增量处理冲突。

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
						70	19	33								
线性						70	19	33	18							
							H_1	H_2	H_3							
二次					18	70	19	33								
					H_2		H_1									
伪随机						70	19	33							18	

设伪随机数序列 9，13．．．H_1 = 14 mod 16 = 14

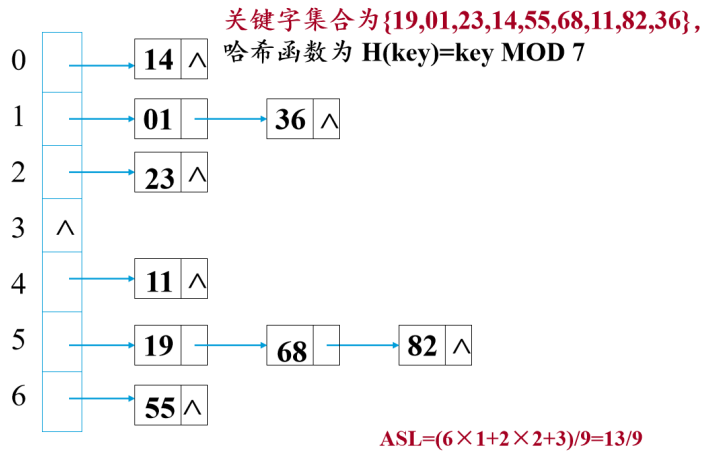
再哈希法

将n个不同哈希函数排成一个序列，当发生冲突时，由RH_i确定第i次冲突的地址H_i。即：$H_i = RH_i(key) \quad i = 1, 2, \ldots, n$，其中：$RH_i$为不同哈希函数。

这种方法不会产生“聚类”，但会增加计算时间。

链地址法

将所有哈希地址相同的记录都链接在同一链表中。



公共溢出区法

假设某哈希函数的值域[0, m-1]，向量 HashTable[0, m-1] 为基本表，每个分量存放一个记录，另设一个向量 OverTable[0, v] 为溢出表。将与基本表中的关键字发生冲突的所有记录都填入溢出表中。

查找方法比较

顺序查找、折半查找、索引查找：

平均查找长度都不为零，差别在于：关键字和给定值进行比较的顺序不同。

	ASL	结构要求	时间复杂度
顺序查找	大	无	O(*n*)
折半查找	小	有序表	O(log *n*)
索引查找	中	分块有序表	O(log (*n* / *s*)+*s*)

哈希查找：

线性探测再散列

$$S_{\mathrm{SUC}} \approx \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$$

线性探测再散列

$$S_{\mathrm{UNS}} \approx \frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$$

随机探测再散列

$$S_{\mathrm{SUC}} \approx -\frac{1}{\alpha}\ln(1-\alpha)$$

随机探测再散列

$$S_{\mathrm{UNS}} \approx \frac{1}{1-\alpha}$$

链地址法

$$S_{\mathrm{SUC}} \approx 1+\frac{\alpha}{2}$$

链地址法

$$S_{\mathrm{UNS}} \approx \alpha + e^{\alpha}$$

哈希表的平均查找长度是装填因子 α 的函数，而不是 n 的函数。这是哈希表所特有的特点。

第六章 排序

排序是计算机内经常进行的一种操作，其目的是将一组“无序”的记录序列调整为“有序”的记录序列。

- 若整个排序过程不需要访问外存便能完成则称此类排序问题为内部排序
- 若参加排序的记录数量很大，整个序列的排序过程不可能在内存中完成，则称此类排序问题为外部排序

排序四分类

1. 插入类：将无序子序列中的一个或几个记录“插入”到有序序列中，从而增加记录的有序子序列长度。
2. 交换类：通过“交换”无序序列中的记录从而得到其中关键字最小或最大的记录，并将它加入到有序子序列中，以此方法增加记录的有序子序列的长度。
3. 选择类：从记录的无序子序列中“选择”关键字最小或最大的记录，并将它加入到有序子序列中，以此方法增加记录的有序子序列的长度。
4. 归并类：通过“归并”两个或两个以上的记录有序子序列，逐步增加记录有序序列的长度。

插入排序

1. 在$R[1\ldots i-1]$中查找$R[i]$的插入位置，
   $R[1\ldots j].key <= R[i].key < R[j+1\ldots i-1].key$
2. 将$R[j+1\ldots i-1]$中的所有记录均后移一个位置
3. 将$R[i]$插入(复制)到$R[j+1]$的位置上

直接插入排序

算法思想：

- 从R[i-1]起向前进行顺序查找，监视哨设置在R[0]
- 对于在查找过程中找到的那些关键字不小于R[i].key的记录，并在查找的同时实现记录向后移动

```
void InsertSort(int r[],int n)
{
    for(int i=2; i<=n; i++) //进行n-2趟插入排序
    {
        r[0] = r[i];    //设置哨兵
            int j;
        for(j=i-1; r[0]<r[j]; j--) //寻找位置
        {
            r[j+1] = r[j]; //记录后移
        }
            r[j+1] = r[0];
    }
}
```

最好的情况（关键字在记录序列中顺序有序）：

“比较”的次数：

$$\sum_{i=2}^{n} 1 = n-1$$

“移动”的次数：

$$0$$

最坏的情况（关键字在记录序列中逆序有序）：

“比较”的次数：

$$\sum_{i=2}^{n} (i+1) = \frac{(n+4)(n-1)}{2}$$

“移动”的次数：

$$\sum_{i=2}^{n} (i+1) = \frac{(n+4)(n-1)}{2}$$

折半插入排序

```
1  void BinSort(int r[],int n)
2  {
3      for(int i=2; i<=n; i++)
4      {
5          r[0]=r[i];
6          low=1;
7          high=i-1;
8          while(low <= high) //折半查找插入位置
9          {
10             mid=(low+high)/2;
11             if(r[0] < r[mid])
12                 high=mid-1;
13             else
14                 low=mid+1;
15         }//high+1即为第i个元素的插入位置
16         for(int j=i-1; j>=high+1; j--)
17             r[j+1]=r[j];
18         r[high+1]=r[0];
19     }
20 }
```

时间复杂度：$O（n^2）$

希尔排序

算法思想：

对待排记录序列先作“宏观”调整，再作“微观”调整。“宏观”调整指的是“跳跃式”的插入排序。

例如：将数组 R[] 的 n 个记录分成 d 个子序列：

$R[1], R[1+d], R[1+2d], \ldots, R[1+kd]$

$R[2], R[2+d], R[2+2d], \ldots, R[2+kd]$

...

$R[d], R[2d], R[3d], \ldots, R[kd], R[(k+1)d]$

其中，d 称为增量，它的值在排序过程中从大到小逐渐缩小，直至最后一趟排序减为 1。

插入排序评述

- 直接插入排序外循环 n 趟，内循环顺序查找 $O(n/2)$，数据移动$O(n/2)$：总的时间复杂度 $O(n^2)$。
- 折半插入排序外循环 n 趟，内循环顺序查找 $O(logn)$，数据移动$O(n/2)$：总的时间复杂度 $O(n^2)$。
- 希尔排序在外循环之外额外增加有限次迭代(三次)。因为宏观的调整，使得数据移动次数从 $O(n/2)$大量降低，逼近但不等于$O(logn)$：总的时间复杂度介于 $O(nlogn)$ 和 $O(n2)$ 之间。

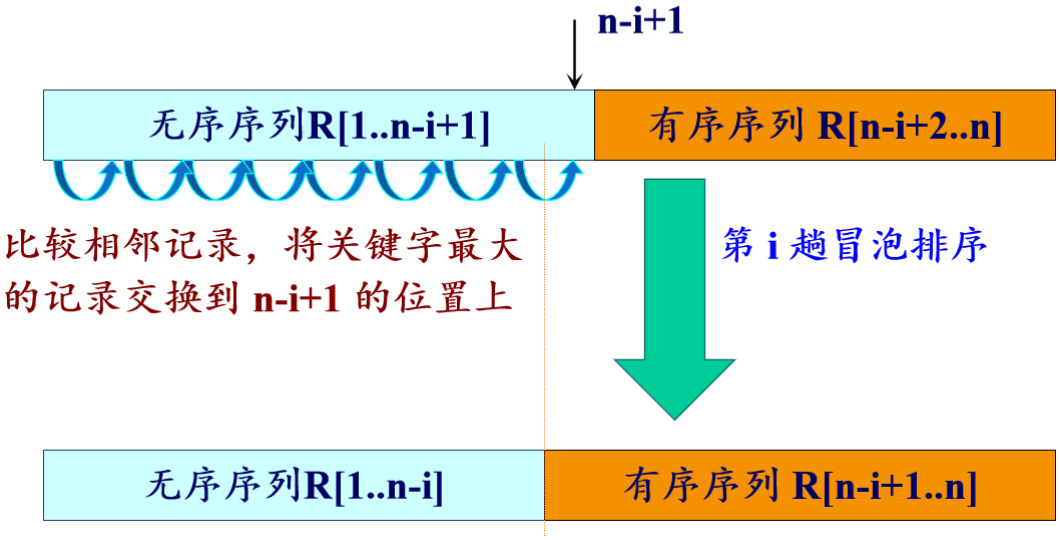
交换排序

通过“交换”无序序列中的记录从而得到其中关键字最小或最大的记录，并将它加入到有序子序列中，以此方法增加记录的有序子序列的长度。

冒泡排序

算法思想：

假设在排序过程中，记录序列R[1..n]的状态为：




```
1  void  BubbleSort(RecordType r[], int length )
2  {
3      int change;
4      int n = length;
5      int change = 1;
6      for (i = 1; i <= n-1 && change; ++i)
7      {
8          change = 0;
9          for ( j = 1; j <= n-i; ++j)
10         if (r[j].key > r[j+1].key )
11         {
12             RecordType x = r[j];
13             r[j]= r[j+1];
14             r[j+1] = x;
15             change = 1;
16         }
17     }
18 }
```

时间复杂度：$O（n^2）$

最好情况(单调升序)

比较次数

$$(n-1)$$

移动次数

$$0$$

最坏情况(单调降序)

比较次数

$$\sum_{i=n}^{2}(i-1)=\frac{n(n-1)}{2}$$

移动次数

$$3\sum_{i=n}^{2}(i-1)=\frac{3n(n-1)}{2}$$

快速排序

算法思想：

找一个记录，以它的关键字作为“枢轴”，凡其关键字小于枢轴的记录均移动至该记录之前，反之，凡关键字大于枢轴的记录均移动至该记录之后。致使一趟排序之后，记录的无序序列$R[s\dots t]$将分割成两部分：$R[s\dots i-1]$和$R[i+1\dots t]$，且 $R[s\dots i-1].key \le R[i].key \le R[i+1\dots t].key$。

首先对无序的记录序列进行“一次划分”，之后分别对分割所得两个子序列“递归”进行快速排序。

```
//快速排序一次划分
int QKPass(RecordType r[],int left, int right)
{
  RecordType x = r[left];  int low, high;
  while(low < high)
  {
    while(low < high && r[high].key >= x.key)
      high--;
    if(low < high)
    { r[low] = r[high]; low++; }
    while(low < high && r[low].key < x.key)
      low++;
    if(low < high)
    { r[high]= r[low]; high--; }
  }
  r[low] = x;
  return low;
}
//快速排序
void QKSort(RecordType r[],int low, int high )
{
  int pos;
  if(low < high)
  {
    pos = QKPass(r, low, high);
    QKSort(r, low, pos-1);
    QKSort(r, pos+1, high);
  }
}
```

时间复杂度：$O（nlogn）$

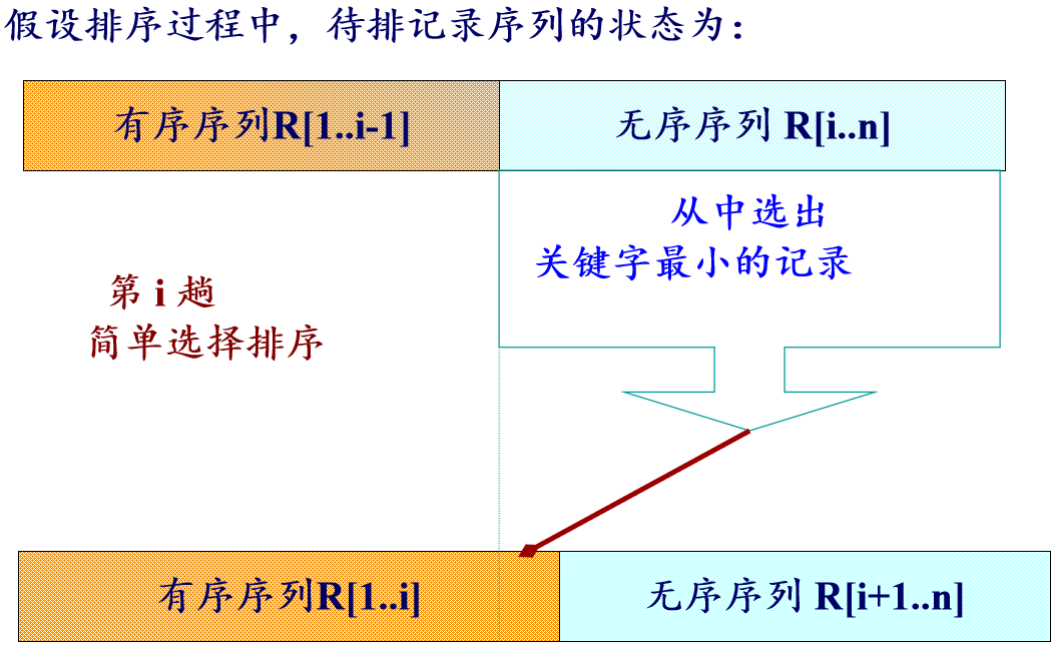
修正改进：若待排记录的初始状态为按关键字有序时，快速排序将蜕化为起泡排序，其时间复杂度为O(n^2)。为避免出现这种情况，需在进行一次划分之前，进行“预处理”，即：先对 $R(s).key$，$R(t).key$和 $R[(s+t)/2].key$三者进行相互比较，然后取关键字为“三者之中”的记录为枢轴记录，以此避免性能的恶化。

选择排序

算法思想：

选择排序(Selection sort)是一种简单直观的排序算法。它的工作原理是每一次从待排序的数据元素中选出最小(或最大)的一个元素，存放在序列的起始位置，直到全部待排序的数据元素排完。 选择排序是不稳定的排序方法。

简单选择排序




```
void SelectSort(RecordType r[], int length)
{
  RecordType   x;
  int n = length;
  for(int i=1 ; i<= n-1; ++i)
  {
    int k = i;
    for ( j = i+1 ; j <= n ; ++j)
      if(r[j].key < r[k].key )
        k = j;
    if ( k != i)
    {
      x = r[i];
      r[i] = r[k];
      r[k] = x;
    }
  }
}
```

时间复杂度：$O（n^2）$

堆排序

大顶堆是具有这样性质的完全二叉树：每个节点的值都大于或等于其左右孩子的值。大顶堆的调整称为筛选：将根节点与左右子树根节点比较，选择最大者调整交换作为当前的根。

算法思想：

将待排记录构造成为一个堆，筛选使其满足堆的性质。选取堆顶元素，从堆中移出，将堆中最后一个元素填入堆顶，继续进行筛选。反复进行这样的操作直到堆为空。

1. 对 n 个关键字，建成深度为$\lfloor \log n \rfloor+1$的堆，所需进行的关键字比较的次数至多 $4n$；

2. 调整“堆顶” n-1 次，总共进行的字比较的次数不超过

$$2(\lfloor \log(n-1) \rfloor+\lfloor \log(n-2) \rfloor+\ldots+\log 2)<2n(\lfloor \log n \rfloor)$$

因此，堆排序的时间复杂度为$O(n\log n)$。

时间复杂度$O(nlogn)$，空间复杂度$O(1)$

归并排序

通过“归并”两个或两个以上的记录有序子序列，逐步增加记录有序序列的长度。

2路归并排序

算法思想：

将两个位置相邻的记录有序子序列归并为一个记录的有序序列。

外层迭代：2路归并排序总共要进行$\log n$趟；

内层迭代：2路归并排序的每一趟都要进行n个数据的大小比较。

因此， 2路归并排序的时间复杂度$O(n\log n)$ 。

时间复杂度：O（$nlogn$）

排序算法总结

时间复杂度归类

$O(n^2)$：直接插入排序、冒泡排序、简单选择排序

$O(nlogn)$：快速排序、堆排序、归并排序

希尔排序介于 $O(n^2)$ 和 $O(nlogn)$ 之间。

空间复杂度归类

$O(1)$：直接插入排序、冒泡排序、简单选择排序、堆排序

$O(logn)$：快速排序

$O(n)$：归并排序

稳定性

空间复杂度×时间复杂度去和 $O(n^2)$ 做比较。

只有简单选择排序是特例，所以有“简单”二字前缀。

排序思想

冒泡排序：好实现，速度不慢，使用于轻量级的数据排序。

插入排序：在已排序列中新增加元素，不用再排序了，直接查找插入。

选择排序：学会怎么去获得最大值，最小值等方法。

归并排序：学会分而治之的方法，而且在合并两个数组的时候很适用。

堆排序：思想加持了很多内力。

快速排序：用的最多的排序。

希尔排序：分治的另类优美，原来还有这种思想的存在。

第七章 算法设计策略

算法分析

包含时间复杂度和空间复杂度的分析。算法的程序实现后，其执行时间主要花费在循环和递归上。

循环的时间代价分析方法：

- 单个循环，循环次数直接决定
- 并列循环，循环次数按加法规则求和
- 嵌套循环，循环次数按乘法规则求积

递归的时间代价分析方法：

对于递归算法，一般可以把时间代价表示为一个递推方程，根据递推方程求解。

- 决定问题规模的度量，确定初始条件
- 找出算法的基本操作，建立一个递推方程
- 求解递推方程

主方法求解递归方程

主方法是一类求解递归方程的快速便捷方法。它适用于求下面类型的递归形式：

$$T(n)=a\cdot T(\frac{n}{b})+f(n)$$

其中 $a>0,b>0$ ，$f(n)$ 为渐进函数。

求解方法是比较渐进函数 $f(n)$ 与 a,b 的指数形式。分三种情况进行讨论。约定一个变量 $\varepsilon\geq 0$ 。

$$T(n)=a\cdot T(\frac{n}{b})+f(n)$$

第一种情况，如果 $f(n)$ 可以被写作：

$$f(n)=n^{\log_b a-\varepsilon}=n^{\log_b a}\cdot\frac{1}{n^{\varepsilon}}\quad\Rightarrow T(n)=O(n^{\log_b a})$$

第二种情况，如果 $f(n)$ 可以被写作：

$$f(n)=n^{\log_b a+\varepsilon}=n^{\log_b a}\cdot n^{\varepsilon}\quad\Rightarrow T(n)=O(f(n))$$

第三种情况，如果 $f(n)$ 可以被写作：

$$f(n)=C\cdot n^{\log_b a}\quad\Rightarrow T(n)=O((\log n)^{k+1}\cdot f(n))$$
$$C=(\log n)^k,k\geq 0$$

算法设计策略

直接法：查找、删除、插入、遍历、拓扑排序

贪心法：最小生成树、关键路径、哈夫曼树

分治法：快速排序、归并排序

动态规划：0/1背包问题、活动安排问题

分制法和动态规划非常类似，区别在于：

- 分制法：子问题独立
- 动态规划：重叠子问题

能采用动态规划求解的问题都需要满足两个限制条件：

- 最优化原理（Principle of optimality）：假设为了解决某一优化问题，需要依次作出n个决策D_1，D_2，…，D_n，则一个最优决策的任一子决策，对于它的当时状态必是最优的
- 无后效性：在最优决策序列中，对于任何一个整数k，1 < k < n，子决策D_k只与由前面k-1个子决策所确定的当前状态有关

动态规划两项显著特点：

- 状态转换方程
- 填表法运算

背包问题

普通背包问题: 在选择物品 i 装入背包时，可以选择物品 i 的一部分，而不一定要全部装入背包，1≤ i ≤n。（直接用贪心算法求解）

0/1背包问题： 在选择装入背包的物品时，对每种物品i只有两种选择，即装入背包或不装入背包。不能将物品i装入背包多次，也不能只装入部分的物品i 。（不能用贪心算法求解）

给定n种物品和一个背包。物品i的重量是w_i价值为v_i，背包容量为C。问应如何选择装入背包中的物品，使得装入背包中物品的总价值最大。

动态规划最核心之处在于构造状态转移函数optimal(), o()

$o(i, j)$表示在前i件物品装入重量j的背包取得的最优值；那么，$o(i\text{-}1, j\text{-}w_i)$表示前$i$-1件物品装入$j\text{-}w_i$重量背包的最优值。

$$o(i, j) = \max\{o(i-1, j-w_i) + v_i, o(i-1, j)\}$$

腾个空间放入物品i　　不放物品i保持最优解

有编号分别为{a,b,c,d,e}的五件物品，它们的重量分别是{2,2,6,5,4}，它们的价值分别是{6,3,5,4,6}。承重为10的背包，如何让背包里装入的物品具有最大的价值总和。

i是外循环，j是内循环，填表按行横填。看着上一行，填写下一行。

		w	v	1	2	3	4	5	6	7	8	9	10
i	a	2	6	0	6	6	6	6	6	6	6	6	6
	b	2	3	0	6	6	9	9	9	9	9	9	9
	c	6	5	0	6	6	9	9	9	9	11	11	14
	d	5	4	0	6	6	9	9	9	10	11	13	14
	e	4	6	0	6	6	9	9	12	12	15	15	15

$$o(i, j) = \max\{o(i-1, j-w_i) + v_i, o(i-1, j)\}$$

活动安排问题

某同学想利用悠长寒假中的连续14天安排一系列激动人心旅行计划。他拟出的旅行活动目的地有五个{a,b,c,d,e}，以14天为单位，五项活动的启动日期分别是{6,1,7,6,11}，结束日期分别是{8,5,9,13,14}，将获得经验值{2,3,5,11,7}。请安排其旅行计划，以使得获得经验值最多。要求写出动态规划的递归关系式，并求解其最优解。

定式解法：

定义 **f(*j*)** 表示与日期 ***j*** 不冲突的活动最晚结束日期 ***i***，即 ***i<j***。定义 **o(*j*)** 表示在日期 ***j*** 时能够获取的最大经验值，即最优解。

第一，如果选择当前日期结束的活动，则不能选取不兼容的活动，且必须包括兼容活动构成的早前最优解。

第二，如果不选择当前日期结束的活动，则必定包括上一日期最优解。

$$o(j)=\begin{cases}0 & j=0\\ \max\{o(j-1),o(f(i))+v_j\} & j>0\end{cases}$$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a						2								
b	3													
c							5							
d						11								
e											7			
o(j)	0	0	0	0	3	3	3	5	8	8	8	8	14	15