数据结构与算法

课程总结

1. 什么是数据结构？
2. 为什么我们需要数据结构？
3. 数据结构都讲了哪些内容？
4. 数据结构在你的专业中重要吗？有什么作用？
5. ADT操作太多了，背不下来！
6. 书上的算法太多了，记不住！
7. 算法设计对我来说还比较难掌握，怎么办？
8. ……

一、数据结构的内容

数据结构用来定义存储到计算机中的数据对象中数据元素之间的关系，目的和作用除了完成数据存储，更重要的是方便数据处理，也称计算。数据元素之间的关系直接影响数据处理，如果存储到计算机中的数据不能很好的体现出这种关系，那么肯定要花费很多时间进行“搜索处理”，既浪费时间，又影响算法的执行效率。

数据结构是比较灵活的，要根据数据对象的特点和问题的需求，参考教材中的比较规范的“示例”灵活选择或定义，要做到以下几点：

1、避免机械地生搬硬套
2、要完整的形成整体，即完整性
3、要前后统一，不自相矛盾，即统一性
4、适用于广泛的数据对象，即通用性
5、不依赖于程序设计语言，即不依赖性
6、要满足问题的需求，方便算法的实施

1、线性结构

数据对象中的元素除第一个元素没有直接前驱，最后一个元素没有直接后继外，其它任何一个元素都有唯一的直接前驱元素和唯一的直接后继元素。

描述相邻两个数据元素的位置关系，谁前谁后。有三种存储结构，亦即：顺序存储、链式存储和静态链表。

（1）顺序存储

采用一组向量，也就是一维数组，通常根据数据元素的实际够成，形成结构体(ElementType)数组。

- 静态结构
- 动态操作（插入、删除）
- 空间利用率问题

```
# define  maxlength  100
Typedef struct {
  ElementType   data[maxlength];
                int   last;
} LIST;
位置类型：
        typedef  int  Position;
线性表 L:   LIST  L;
```

线性表的第*i*个数据元素a_i的存储位置为:$LOC(a_i)=LOC(a_1)+(i-1)*L$

线性表动态分配实现的顺序存储结构:

```
#define LIST_INIT_SIZE      100        //初始分配空间
#define LISTINCREMENET   10        //分配增量
typedef struct{
    ElemType   *elem;                    //元素空间
    int  length;                         //表的长度
    int listsize;                        //当前容量
}AqList ;
```

```
L.elem  = (ElemType*) malloc (LIST_INIT_SIZE*sizeof(ElemType));
```

```
newbase = (ElemType *)realloc(L.elem,
                              (L.listsize+LISTINCREMENT)*sizeof(ElemTYPE));
```

可调数组

顺序表中插入、删除元素时的平均移动元素次数的期望值：

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

ListInsert(*&L,i,e*)

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

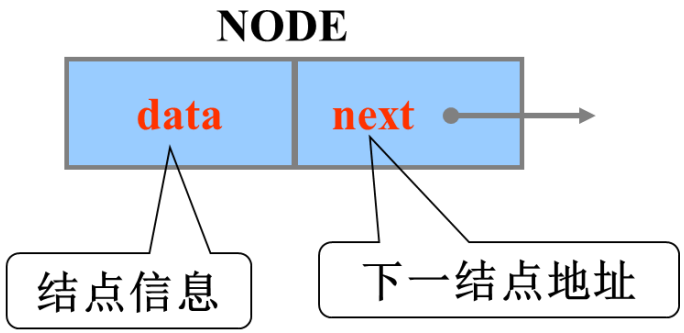
ListDelete(*&L,i,&e*)

（2）链式存储

元素先后的位置通过指针链接。也就是每个数据元素的存储空间内同时存储其直接后继元素的地址，形成线性链表。

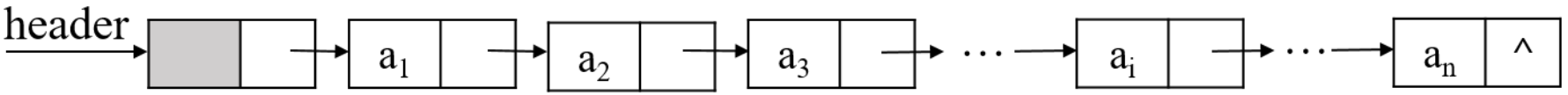
动态结构。

结点形式




```
struct   NODE  {
     ElementType  data ;
     struct NODE  *next ;
} ;

typedef  NODE *LIST;
typedef  NODE  *Position;
```

```
struct NODE *Header;
…… …
Header = (struct NODE *)malloc(sizeof(struct NODE *))
    或  Header = New NODE; //C++
…… …
free(Header)
    或 Delete Header;  //C++
```

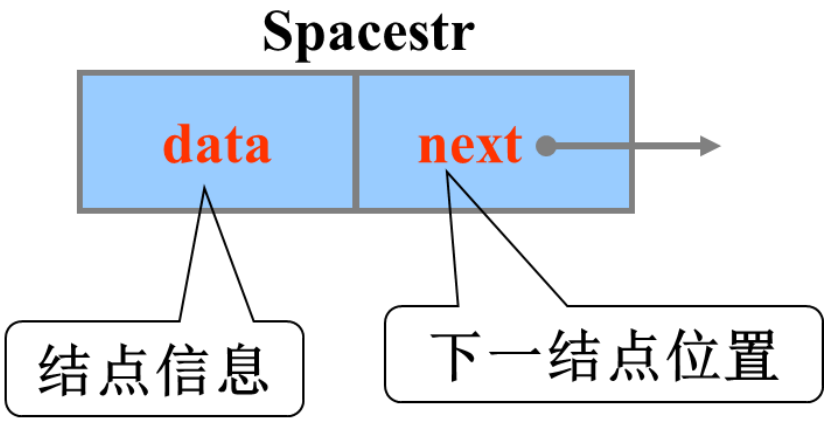


- 表头结点是线性链表第1个结点的前驱，使线性链表的插入和删除结点的操作统一起来；
- 通常，将线性链表中的元素位置超前一个结点，即指向结点前驱的指针；
- 头结点指针相当于一个地址常量。

（3）静态链表，也叫游标

意思是：静态存储+动态操作

结点形式




```
struct  Spacestr  {
    ElementType  data ;
    int    next ;
} ;
```

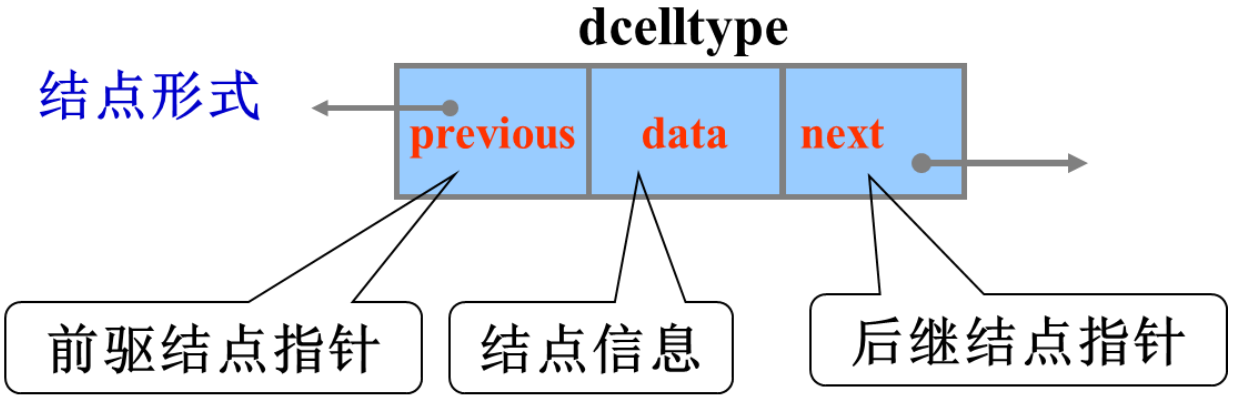
线性表：

```
Spacestr  SPACE[ maxsize ] ;
typedef  int  Position;
```

注意比较：

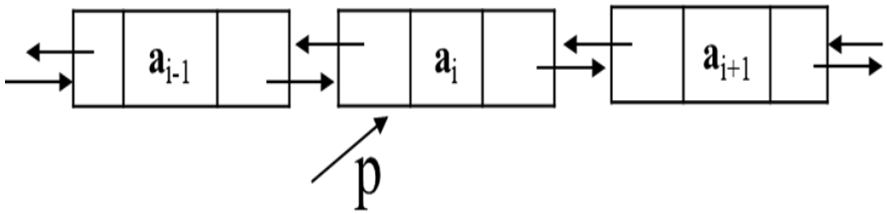
- 第*i*元素：L.data[i]，元素*i*的后继：L.data[i+1]
- 结点p的元素值：p->data，结点p的后继元素值：p->next->data
- 位置*i*的元素值：SPACE[i].data，其后继元素值：SPACE[SPACE[i].next].data

双向链表

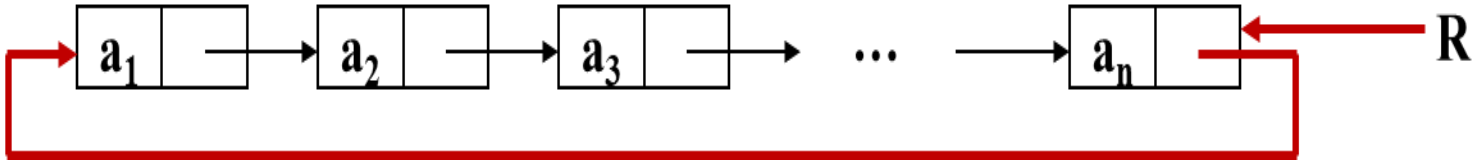


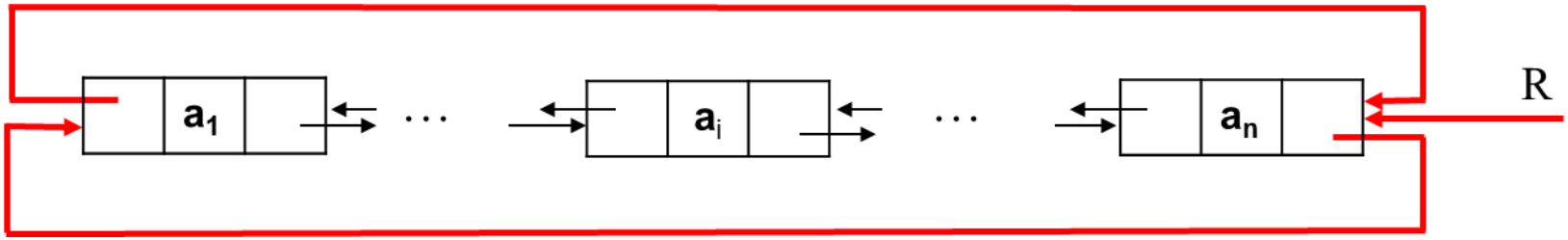
结点类型

```
struct  DnodeType  {
    ElementType  data ;
    DnodeType  *next, *previous ;
} ;

typedef   DnodeType  *DLIST ;
typedef   DnodeType  *Position ;
```



环形链表



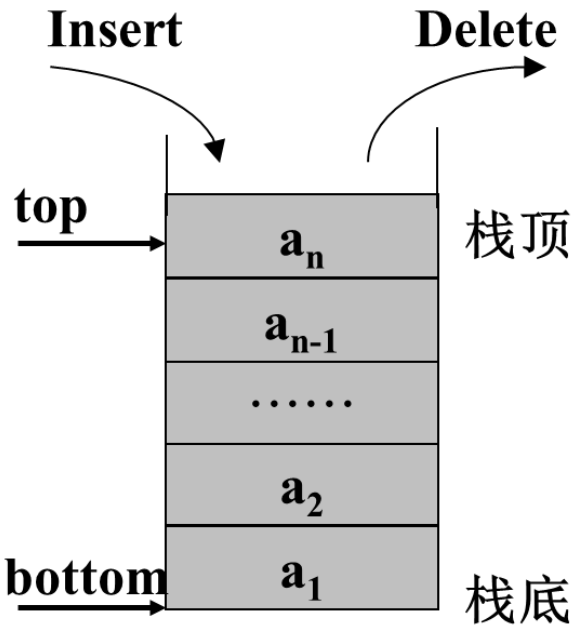


线性表 静态存储 与 动态存储的 比较

顺序存储	比较参数	链式存储
固定，不易扩充	←表的容量→	灵活，易扩充
随机存取	←存取操作→	顺序存取
插入删除费时间	←时间→	访问元素费时间
估算表长度，浪费空间	←空间→	实际长度，节省空间
…	…	…

栈是线性表的一种特殊形式，是一种**限定性数据结构**，也就是在对线性表的操作加以限制后，形成的一种新的数据结构。

【定义】**栈**是限定只在表尾进行插入和删除操作的线性表。
栈又称为**后进先出**(Last In First Out)的线性表。
简称LIFO结构。

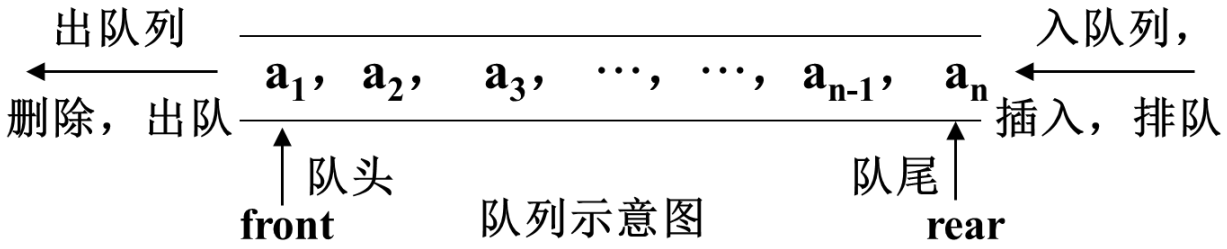


栈示意图

- 从栈的ADT反看栈的特性
  - InitStack(&S);
  - StackEmpty(S);
  - GetTop(S,&e);
  - Push(&s,e);
  - Pop(&s,&e)
- 要注意栈和递归的关系

队列是对线性表的插入和删除操作加以限定的另一种限定型数据结构。

【定义】 将线性表的插入和删除操作分别限制在表的两端进行，和栈相反，队列是一种先进先出（First In First Out，简称 FIFO 结构）的线性表。



假溢出

插入元素：

Q.rear = (Q.rear + 1) % maxlength

删除元素：

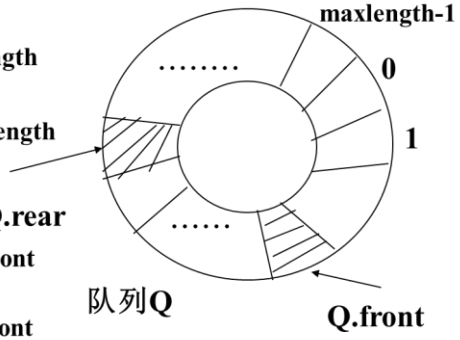
Q.front = (Q.front + 1) % maxlength

队空：

(Q.rear + 1) % maxlength == Q.front

队满：

(Q.rear + 1) % maxlength == Q.front



- 从队的ADT反看队的特性

InitQueue(&Q);

QueueEmpty(Q);

GetHead(Q,&e);

EnQueue(&Q,e);

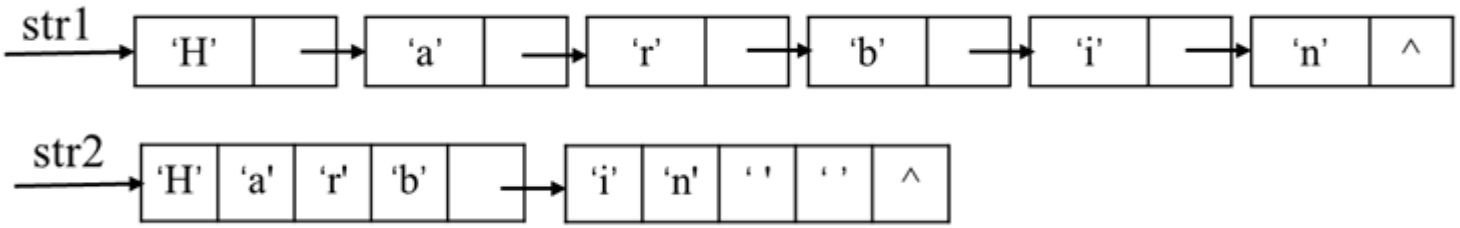
DeQueue(&Q,&e);

串 线性表的一种特殊形式，表中每个元素的类型为字符型，是一个有限的字符序列。

串的两种存储结构

```
struct node {
    char data;
    node *link;
};
typedef node *STRING1;
STRING1 str1;
```

```
struct node {
    char data[4];
    node *link;
};
typedef node *STRING2;
STRING2 str2;
```



KMP算法的核心思想：

利用已经得到的部分匹配信息来进行后面的匹配过程。

数组将数组理解成线性表，数组本身就是线性表的一种存储结构。

- 数组是由下标（index）和值（value）组成的序对（index，value）的集合。
- 也可以定义为是由相同类型的数据元素组成有限序列。
- 数组在内存中是采用一组连续的地址空间存储的，正是由于此种原因，才可以实现下标运算。
- 所有数组都是一个一维向量。

两种存储方式：

一是按行存储（C语言、PASCAL等）；

二是按列存储（FORTRAN等）；

数组的压缩存储：

对称矩阵（上下三角阵）：为实现节约存储空间，我们可以为每一对对称元素分配一个存储空间，这样，原来需要的 n^2 个元素压缩存储到 $n(n+1)/2$ 个元素空间。

B	a_{11}	a_{21}	a_{22}	a_{31}	a_{32}	a_{33}	…	a_{ij}	…	a_{n1}	…	a_{nn}
k =	1	2	3	4	5	6		i(i-1)/2+j		n(n-1)/2+1		n(n+1)/2

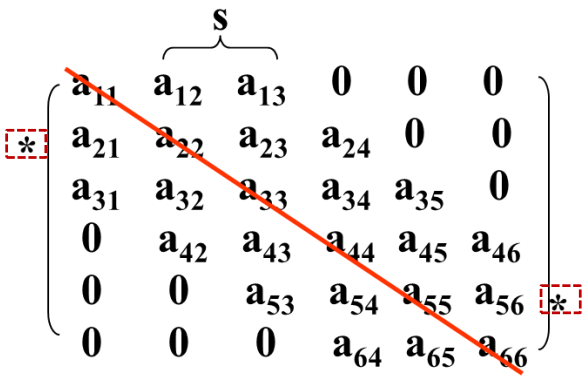
对角（带状）矩阵

所有非零元素都集中在以主对角线为中心的带状区域内。

称S为带宽，为实现压缩存储，我们只存储带区内的非零元素。

LOC[i，j] = LOC[1，1] + [(2s+1)*(i-1)+(j-i)]*L

L = 1;



稀疏矩阵

稀疏矩阵中，零元素的个数远远多于非零元素的个数。为实现压缩存储，只存储非零元素。

$$M=\begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{pmatrix}$$

i	j	v
1	2	12
1	3	9
3	1	-3
3	6	14
4	3	24
5	2	18
6	1	15
6	4	-7

```
#define MAXSIZE 12500
typedef struct {
    int i，j；
    ElementType e；
} Triple；
typedef struct {
     Triple data[MAXSIZE+1];
     int mu，nu，tu;
} TSMatrix;
```

2、层次结构：树和二叉树

树是n（n≥0）个结点的有限集。在任意一棵非空树中：

（1）有且仅有一个特定的称为根的结点；

（2）当n＞1时，其余结点可分为m（m＞0）个互不相交的有限集 T_1，T_2，…,T_m；

（3）其中每一个集合T_i（0＜i≤m）本身又是一棵树，称为根的子树。

数据对象D：D是具有相同类型的数据元素的集合。

数据关系R：若 D 为空集，则称为空树；

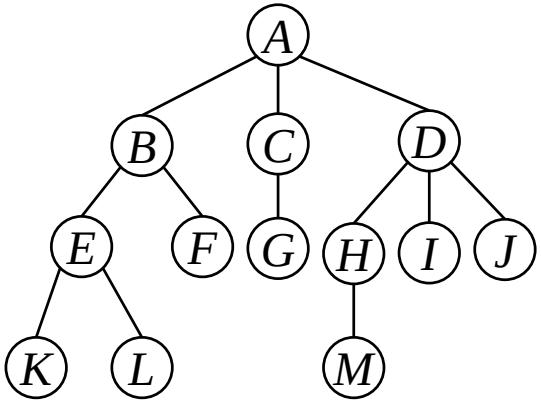
若 D 仅含一个数据元素，则 R 为空集，否则 R = { H }，H 是如下的二元关系：

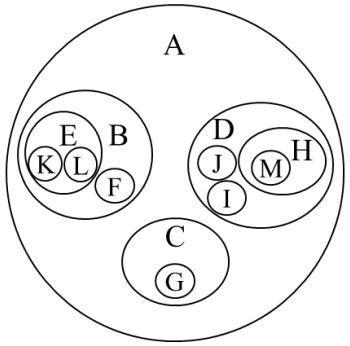
（1）在 D 中存在唯一的称为根的数据元素 root ，它在关系 H 下无前驱；

（2）若 D - { root } ≠Φ，则存在D-{root}的一个划分$D_1,D_2,\ldots,D_m(m>0)$，对任意 $j \neq k$（$1 \le j, k \le m$）有$D_j \cap D_k = \Phi$，且对任意的$i(1 \le i \le m)$，唯一存在数据元素 $x_i \in D_i$，有 $< root , x_i > \in H$;

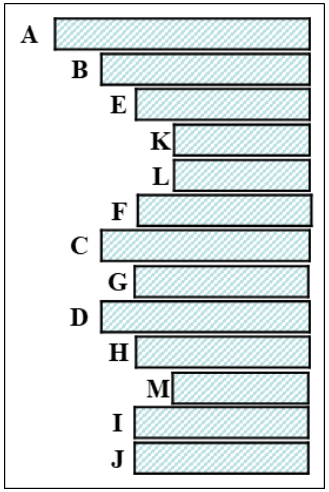
（3）对应于D-{ root }的划分，H-{<root,x_1>,…,<root,x_m>}有唯一的划分$H_1,H_2,\ldots,H_m$（m＞0）,对任意$j \neq k$（1≤j，k≤m）有$H_j \cap H_k \neq \Phi$,且对任意的i（$1 \le i \le m$），H_i是D_i上的二元关系，（D_i，{H_i}）是一棵符合本定义的树，称为根root的子树。

树的表示

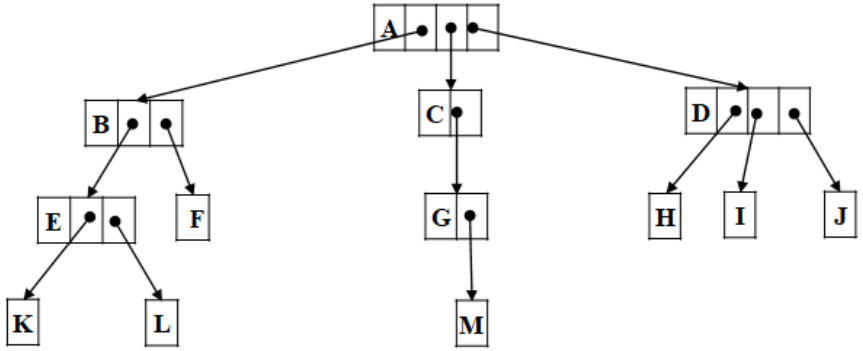


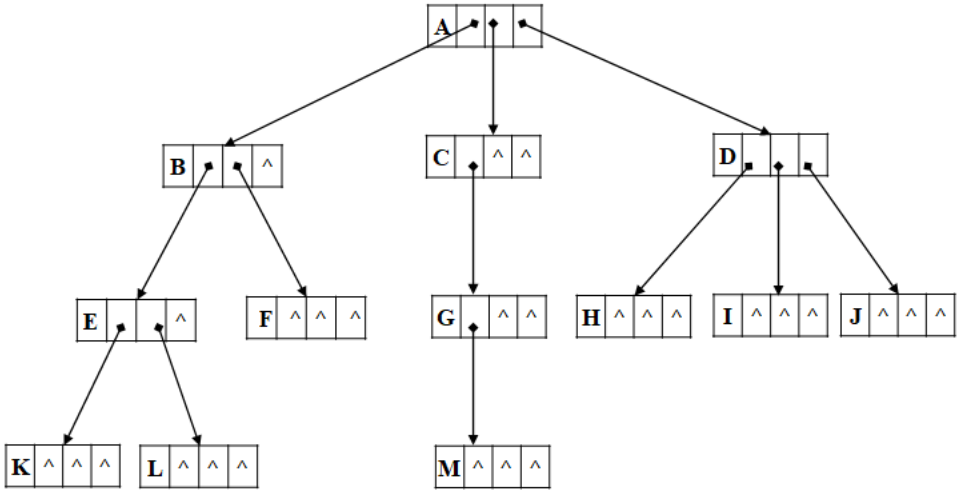


(A(B(E(K,L),F),C(G),D(H(M),I,J)))



树的存储





二叉树是有限个结点的集合，这个集合或者是空集，或者是由一个根结点和两棵互不相交的二叉树组成，其中一棵叫根的做左子树，另一棵叫做根的右子树。

数据对象D：D是具有相同类型的数据元素的集合。

数据关系R：

若 D=Φ，则R=Φ，称BinaryTree为空二叉树；

若 D≠Φ，则 R={ H }，H 是如下的二元关系：

（1）在 D 中存在唯一的称为根的数据元素 root ，它在关系H下无前驱；

（2）若 D - { root } ≠Φ，则存在D-{root}={D_l,D_r},且$D_l \cap D_r = \Phi$ ；

（3）若$D_l \neq \Phi$，则D_l中存在唯一的元素x_l，<root,x_l>∈H，且存在D_l上的关系$H_l \subset H$；若$D_r \neq \Phi$，则D_r中存在唯一的元素x_r，<root,x_r>∈H，且存在D_r上的关系$H_r \subset H$；H={<root, x_l>，<root, x_r>,H_l,H_r} ；

（4）（D_l，{H_l}）是一棵符合本定义的二叉树，称为根的左子树；

（D_r，{H_r}）是一棵符合本定义的二叉树，称为根的右子树。

二叉树的性质

性质1： 在二叉树中的第 i 层的结点数最多为：2^{i-1} 。

性质2： 高度为k的二叉树其结点总数最多为$2k-1\ (k \geq 1)$ 。

性质3： 对任意的非空二叉树 T，如果叶结点的个数为 n_0，而其度为 2 的结点数为 n_2，则：$n_0 = n_2 + 1$ 。

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

性质5： 如果对一棵有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有：

(1) 如果 $i=1$，则结点i是二叉树的根，无双亲；
如果 $i>1$，则其双亲结点是$\lfloor i/2 \rfloor$；

(2) 如果 $2i>n$，则结点i无左孩子结点，否则其左孩子结点是 $2i$；

(3) 如果 $2i+1>n$，则结点i无右孩子结点，否则其右孩子结点是 $2i+1$。

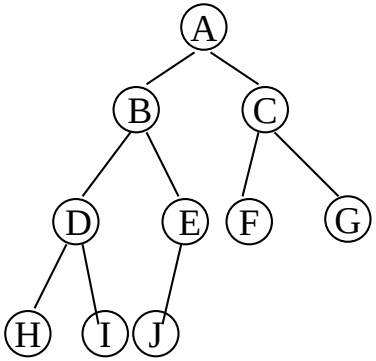
二叉树的存储结构

■ 顺序存储

根据性质5，如已知某结点的层序编号i,则可求得该结点的双亲结点、左孩子结点和右孩子结点。

（1）完全（或满）二叉树

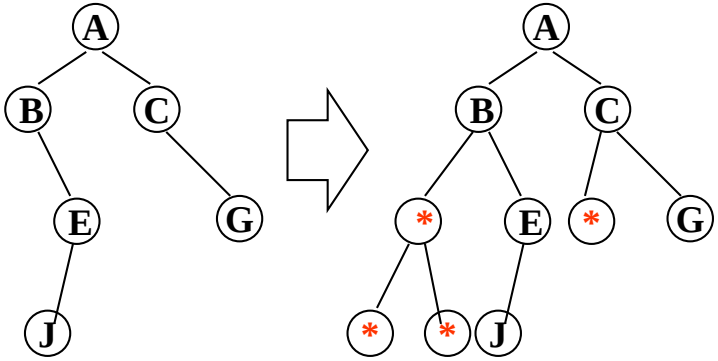
采用一维数组，按层序顺序依次存储二叉树的每一个结点。



A	B	C	D	E	F	G	H	I	J
1	2	3	4	5	6	7	8	9	10

（2）一般二叉树

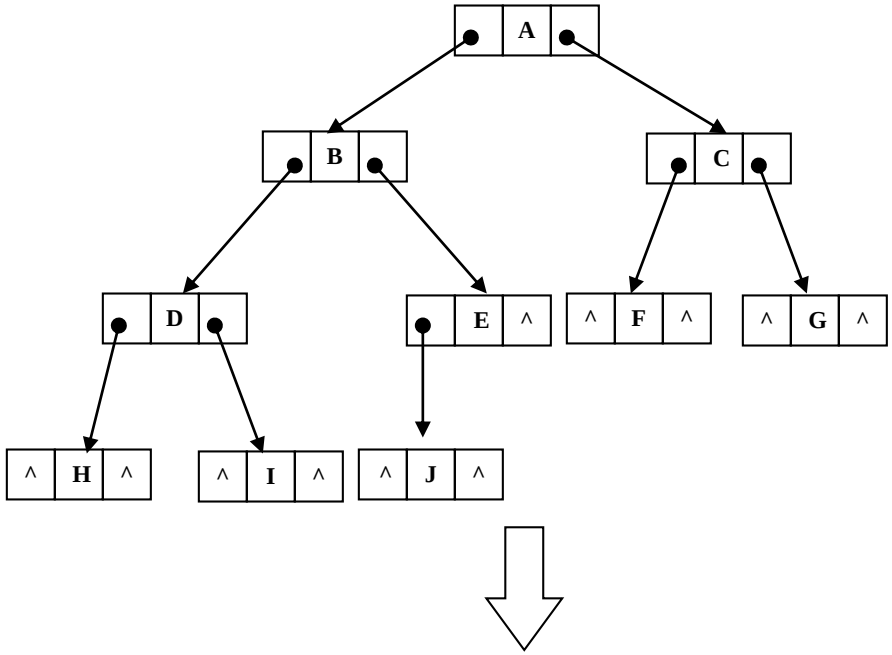
通过虚设部分结点，使其变成相应的完全二叉树。



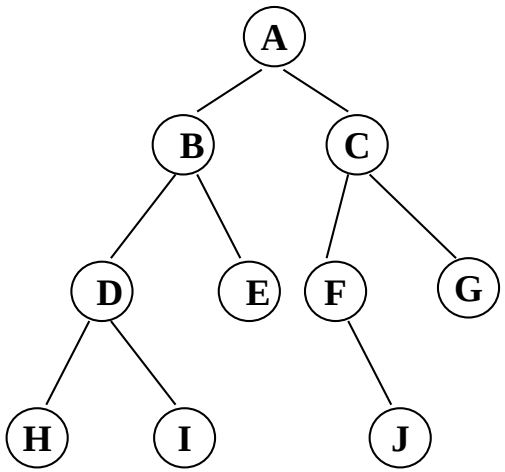
A	B	C	*	E	*	G	*	*	J
1	2	3	4	5	6	7	8	9	10

■ 二叉树的左右链表示

lchild	data	rchild

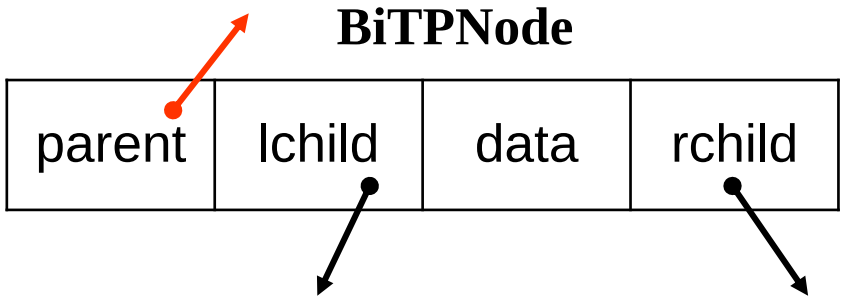



```
Struct  Node  {
    Struct  Node  *lchild ;
    Struct  Node  *rchild ;
          datatype  data ;  } ;
Typedef  struct   Node    * BTREE ;
```



■ 二叉树的三叉链表存储表示

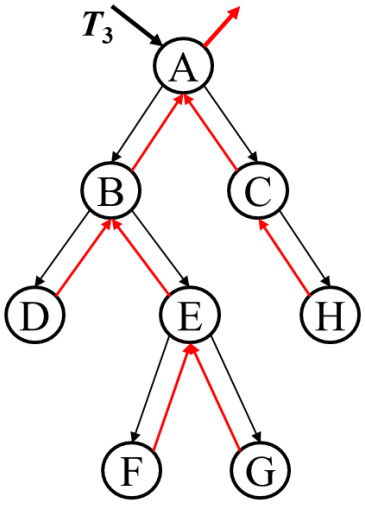
```
typedef struct BiTPNode
{
    ElementType data;
    struct BiTPNode *parent,*lchild,*rchild;
} *BiPTree;
```



◆ ***n*个结点的二叉树有*n*+2个空指针！**

很显然：

- 相对二叉链表表示的二叉树，除了找父结点的操作变得很容易外，其它基本操作没有什么变化。
- 对二叉树的先序/中序/后序的非递归遍历，不需要再使用栈。



类型题：

（1）如何判断一颗任意二叉树是否为满二叉树？

（2）如何判断一颗任意二叉树是否为完全二叉树？

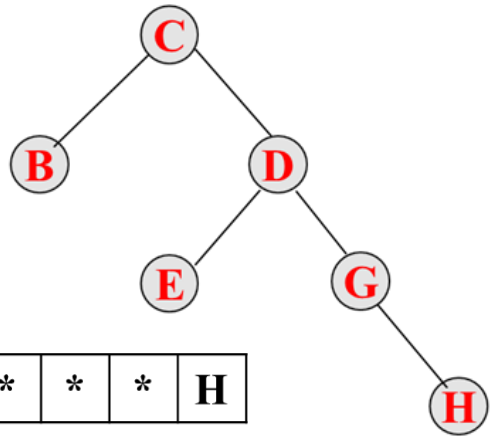
（3）求二叉树任意结点所在的层？

（4）求任意结点的所有祖先结点（根到该结点的路径）

（5）统计任意二叉树中的结点个数？

总结点、度为2、度为1、度为0的结点个数。

（6）二叉链表存储的二叉树转换到按照完全二叉树存储的数组中。‘*’表示空结点。



C	B	D	*	*	E	G	*	*	*	*	*	*	*	H

■ 线索二叉树

（1）在n个结点的二叉树左右链表示中，有n+1个空链域。如何利用n+1个空链域，使二叉树的操作更加方便；

（2）在二叉树左右链表示中，为求某个结点的（中序）前驱 $p 或（中序）后继 p$，每次都要从树根开始进行查找，很不方便。

【中序线索二叉树定义】

若结点p有左孩子，则p->lchild指向其左孩子结点，

否则令其指向其（中序）前驱；

若结点p有右孩子，则p->rchild指向其右孩子结点，

否则令其指向其（中序）后继。

lchild	ltag	data	rchild	rtag

Typdef Struct LNode * THTREE;

```
Struct  LNode {
            ElementType   data ;
            Struct  LNode
                    *lchild , *rchild ;
             int  ltag , rtag ; }
```

树的存储结构

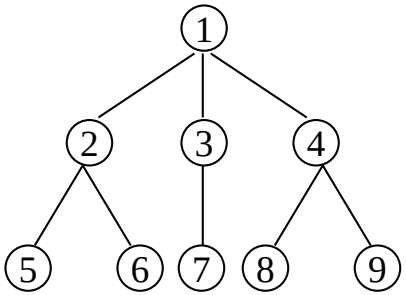
面向特定的操作，设计合适的存储结构

- 树的双亲表示法（~~数组实现方法~~）

树的结点依次编号为1，2，3，… ，n；设数组A[i]

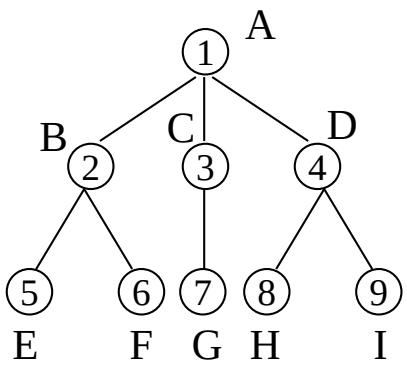
$$A[i] = \begin{cases} j & \text{若结点 i 的父亲是 j} \\ 0 & \text{若结点 i 是根} \end{cases}$$

A		0	1	1	1	2	2	3	4	4
	0	1	2	3	4	5	6	7	8	9




```
Struct  Node  {
        int  parent ;
        char  data ;  } ;
```

```
Typdef  Node  TREE[11];
TREE  T ;
```

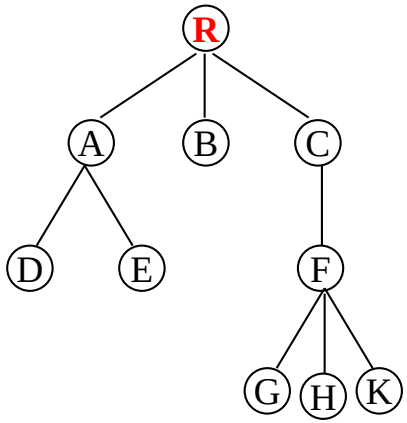


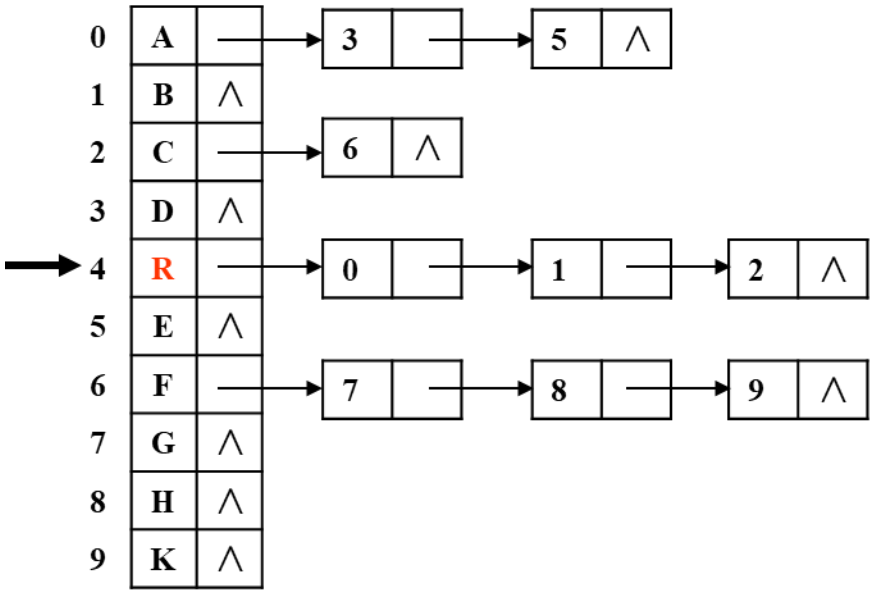
T

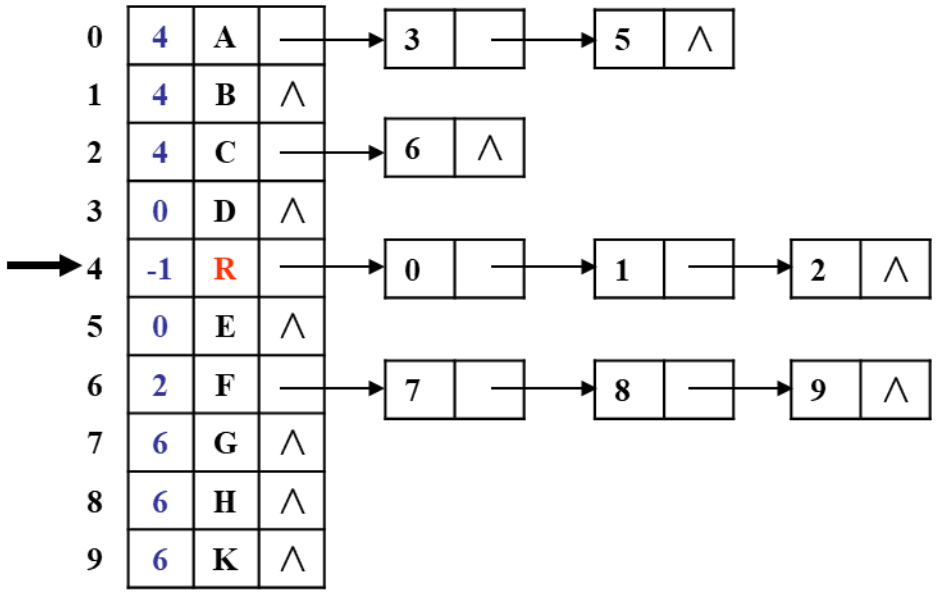
parent	0	1	1	1	2	2	3	4	4
data	A	B	C	D	E	F	G	H	I
	0	1	2	3	4	5	6	7	8	9

■ 树的孩子表示法（~~邻接表表示法~~）

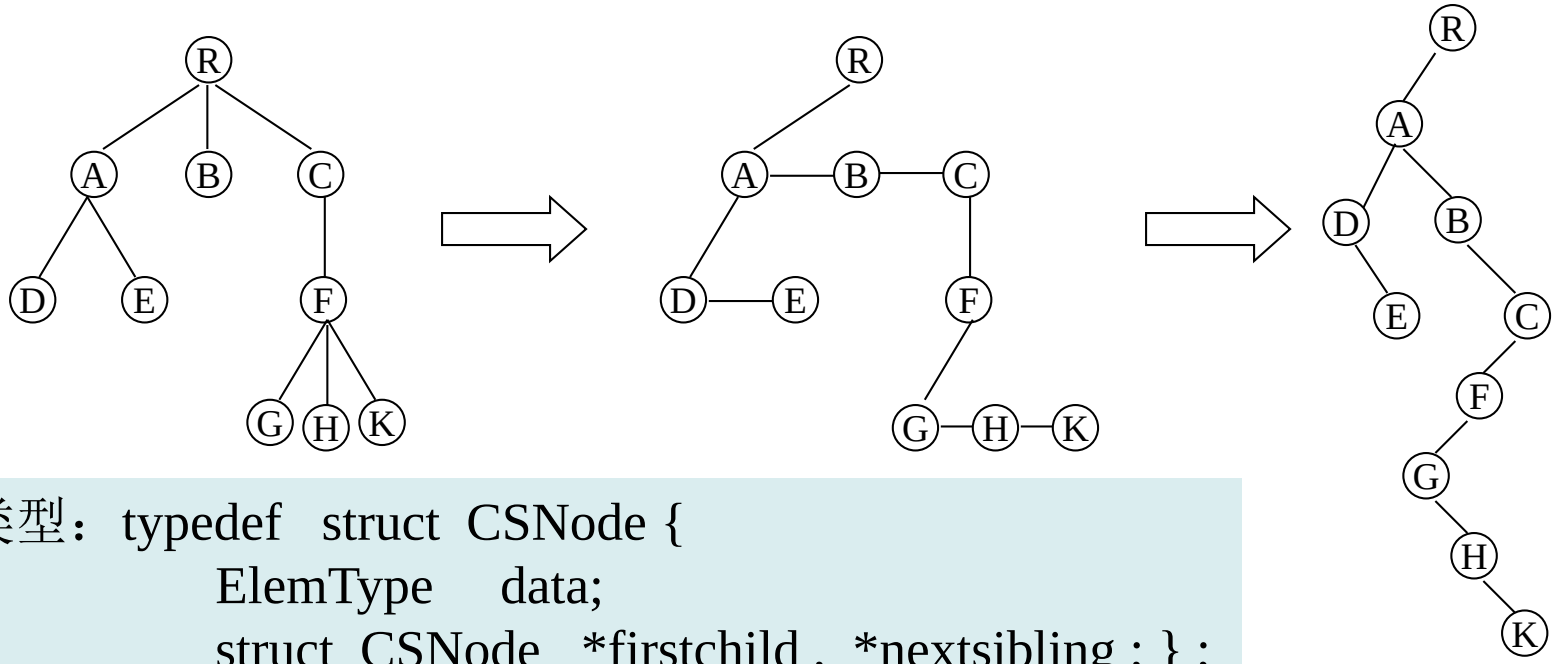
```
typedef  struct  CTNode  {
  int    child ;
  struct  CTNode  *next ; }   *ChildPtr ;
typedef  struct  {
  Telementtype  data ;
   ChildPtr  firstchild ;      }     CTBox ;
typedef  struct  {
   CTBox  Nodes[MAX_TREE_SIZE] ;  int   n , r ; }  Ctree ;
```







■ 树的孩子兄弟表示法（二叉树表示法）

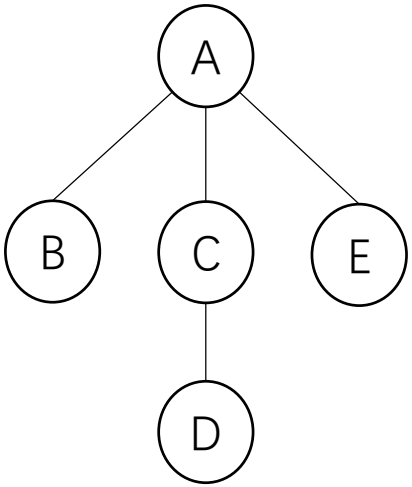



```
类型：typedef   struct  CSNode {
            ElemType     data;
            struct  CSNode   *firstchild ,  *nextsibling ; } ;
```

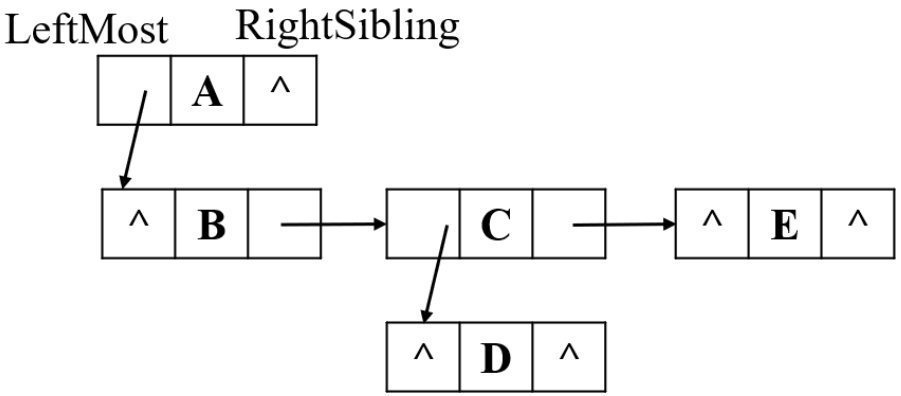
遍历	树		二叉树
先序	RADEBCFGHK	⟷	RADEBCFGHK
中序	DAERBGFHKC		DEABGHKFCR
后序	DEABGHKFCR	⟷	EDKHGFCBAR

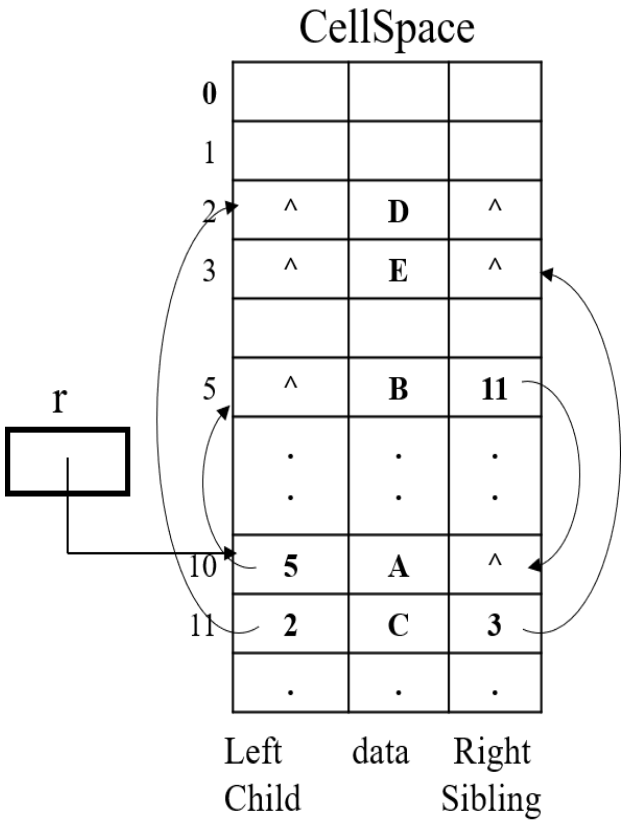
左孩子右兄弟

■ 树的一种静态结构存储




```
Struct {
   DataType data;
   int LeftChild;
   int RightSibling;
}CellSpace[MaxNodes];
```

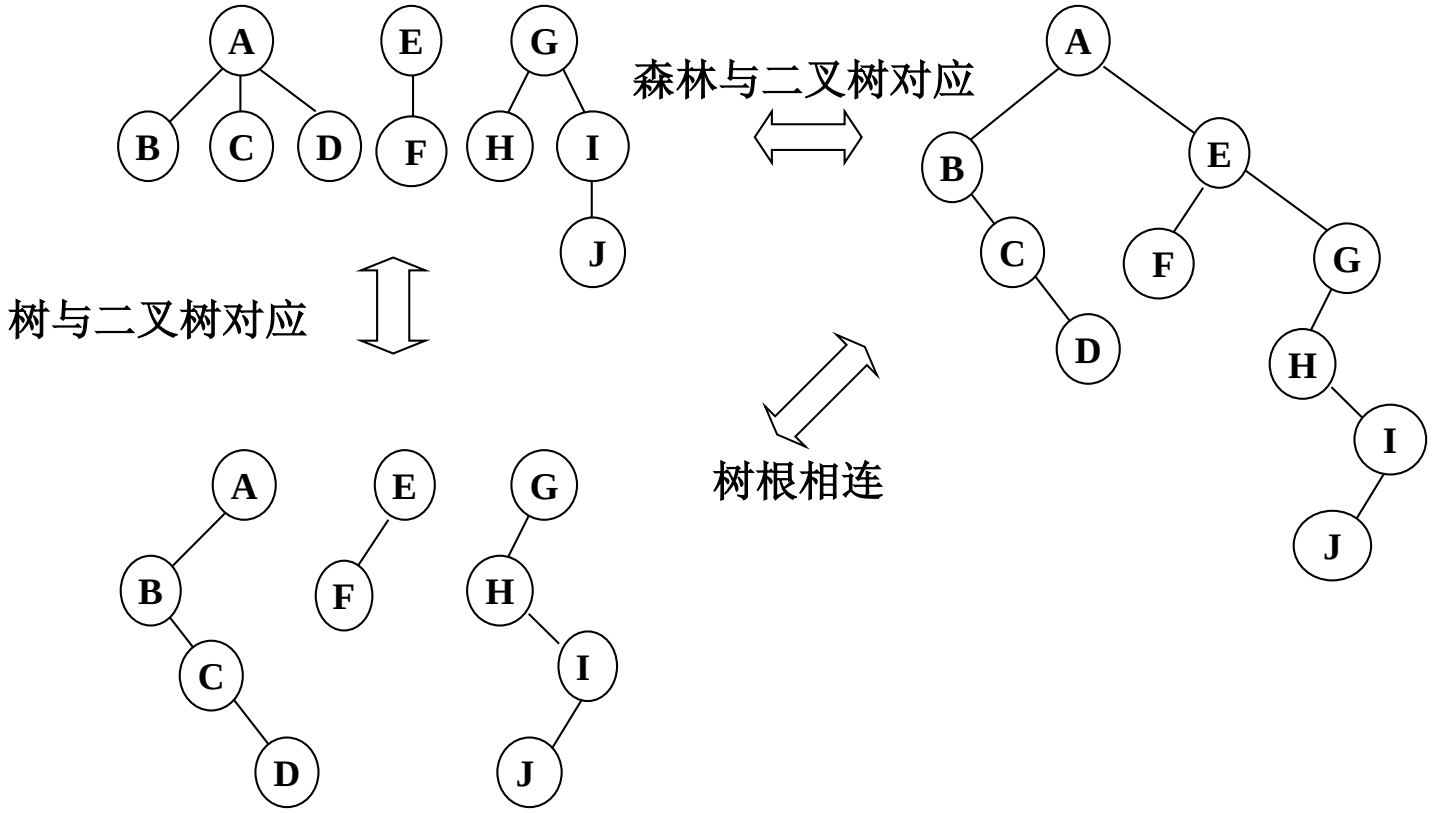




森林与二叉树

森林转换为二叉树：

（1）先将森林中每棵树转换成二叉树；
（2）二叉树的树根连接起来。



树、森林、二叉树的遍历对应关系		
树	**森林**	**对应的二叉树**
先序遍历	先序遍历	先序遍历
后序遍历	中序遍历	中序遍历
层次遍历		后序遍历

3、网状结构：图

【定义】一个图G=（V，E）是一个由非空的有限集 V和一个边集E所组成的。若E中的每条边都是顶点的有序对（v，w），就说该图是有向图。若E中的每条边是两个不同顶点无序对，就说该图是无向图，其边仍表示成（v, w）。

【ADT】 Graph G = (V , R)

数据对象V：V是具有相同特性的数据元素的集合，称为顶点集。

数据关系R:

R = { VR }

VR = { <v , w >|v, w ∈V,且P(v,w),<v, w>表示从v到w的弧，
谓词P(v, w)定义了弧 <v, w>的意义或信息 }

int First Adj Vex (G, v)
//返回值为图G中与顶点v邻接的第一个邻接点，0为没有邻接点

int Next Adj Vex (G, v, w)
//返回值为图G中与顶点v邻接的w之后的邻接点，0为无下一个邻接点

■ 图的顺序存储—邻接矩阵

设图G = (V, E) ， V = { 0， 1， …， n-1 }则表示G的邻接矩阵 A 是其元素按下式定义的n×n矩阵:

$$A[i][j] = \begin{cases} 1 & 若\ (i,j) \in E \\ 0 & 若\ (i,j) \notin E \end{cases}$$

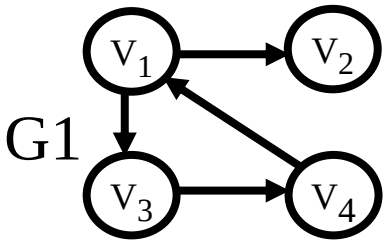
网的邻接矩阵可定义为：

$$A[i][j] = \begin{cases} W_{ij} & 若\ (i,j) \in E \\ \infty & 若\ (i,j) \notin E \end{cases}$$

$$TD(v_i) = \sum_{j=0}^{n-1} A[i][j] = \sum_{j=0}^{n-1} A[j][i]$$ （ n :顶点个数，无向图 ）

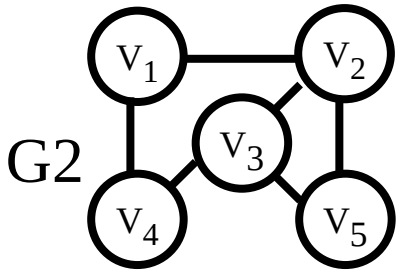
$$TD(v_i) = OD(v_i) + ID(v_i) = \sum_{j=0}^{n-1} A[i][j] + \sum_{i=0}^{n-1} A[i][j]$$ (n : 顶点个数，有向图）

图的表示 ---邻接矩阵



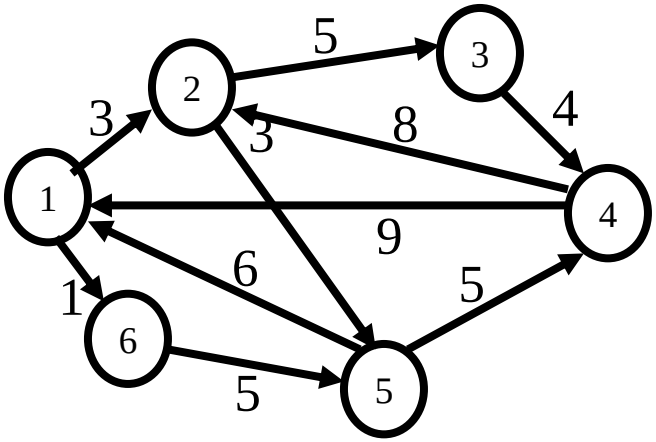
G1.arcs

	v_1	v_2	v_3	v_4
v_1	0	1	1	0
v_2	0	0	0	0
v_3	0	0	0	1
v_4	1	0	0	0



G2.arcs

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$



$$\begin{pmatrix} \infty & 3 & \infty & \infty & \infty & 1 \\ \infty & \infty & 5 & \infty & 3 & \infty \\ \infty & \infty & \infty & 4 & \infty & \infty \\ 9 & 8 & \infty & \infty & \infty & \infty \\ 6 & \infty & \infty & 5 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 5 \end{pmatrix}$$

图的存储至少存储两个内容：顶点数据和顶点间的关系。

```
#define   INFINITY   INT_MAX
#define   MAX_VERTEX_NUM   20
Typedef   enum { DG, DN, AG, AN }  GraphKind ;
Typedef   struct  ArcCell {
            VRType        adj ; // 顶点的邻接关系
            InfoType      *info ; //弧相关信息的指针
 } ArcCell  ,  AdjMatrix[ MAX_VERTEX_NUM][MAX_VERTEX_NUM] ;

Typedef   struct  {
            VertexType    vex[MAX_VERTEX_NUM] ;
            AdjMatrix     arcs ;
            int           vexnum , arcnum ;
            GraphKind     kind;
} Mgraph ;
```

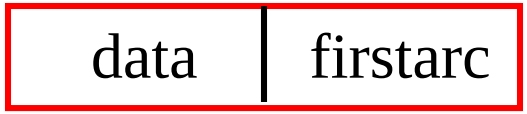
如何建立简单的邻接矩阵？

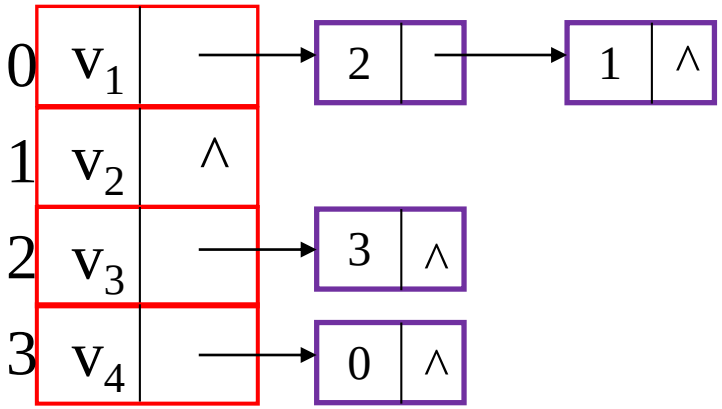
图的链式存储 ---邻接表法

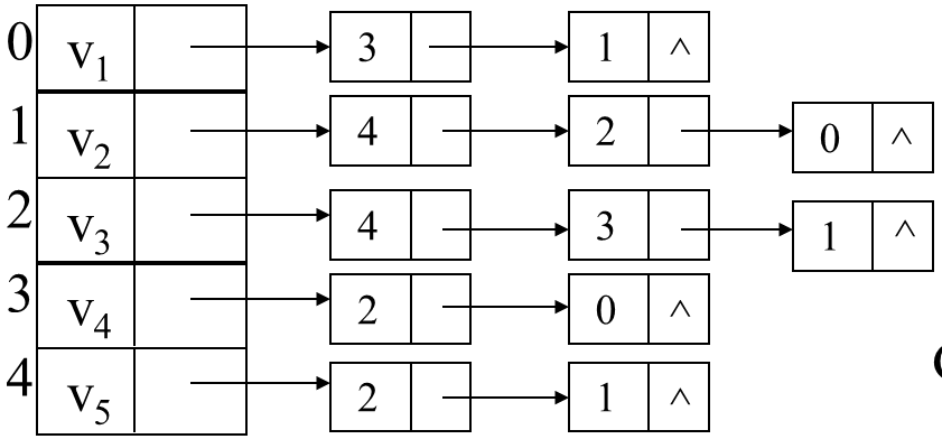
```
#define  MAX_VERTEX_NUM  20
Typedef  struct  ArcNode  {
         int                adjvex ;//位置
         struct  ArcNode   *nextarc ;
         InfoType          *info ;
}  ArcNode ;

Typedef  struct  Vnode  {
         VertexType     data ;
         ArcNode       *firstarc ;
}  Vnode, AdjList[MAX_VERTEX_NUM] ;

Typedef  struct {
         AdjList    vertices ;
         Int        vexnum ;
         Int        kind ;
}  ALGraph ;
```

表结点（边）

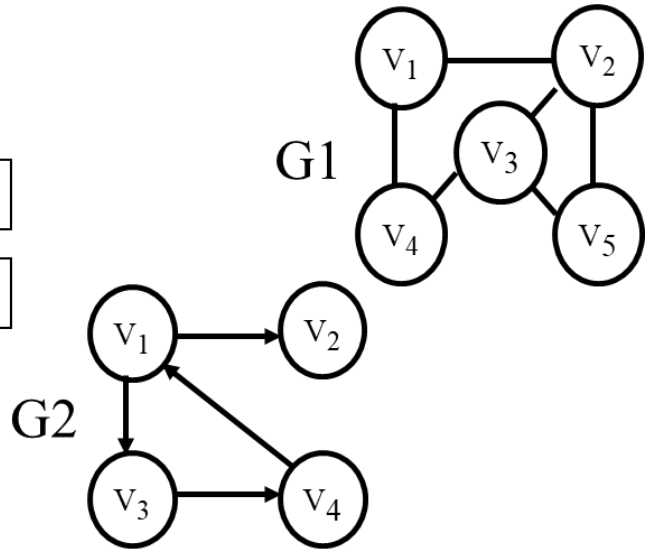
头结点(顶点)

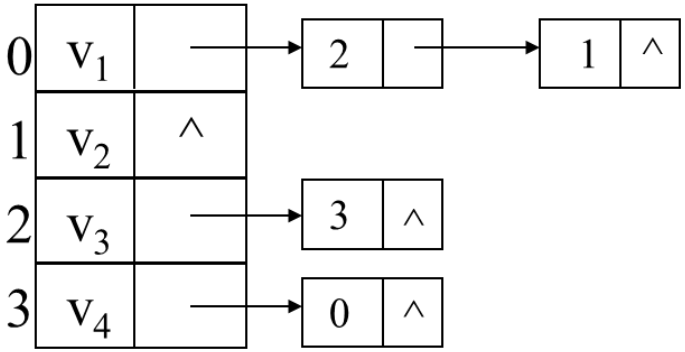






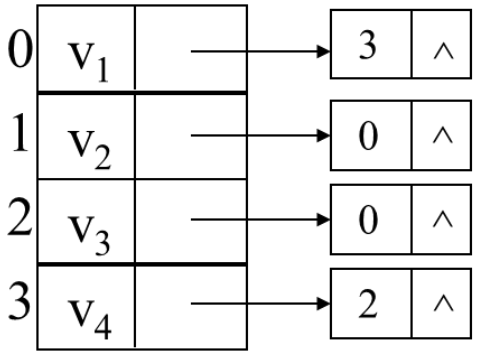
无向图G1邻接表





有向图G2邻接表

+



G2的逆邻接表

4、查找中的数据结构

线性查找：顺序表 和 线性链表
　　无序表的查找

折半查找：顺序表 和 线性链表+（一至四级）索引，称为跳表
　　有序表的查找

分块查找：无序表 与 有序表 的结合，所谓分块有序

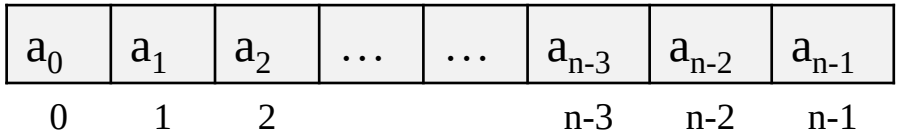
动态查找：二叉查找树　平衡二叉树
　　m-路从查找树　m-路平衡查找树
　　B-树、B+树、键树、红黑树、…

地址散列法：哈希表，顺序表（+链表，链地址法）

哈希表在文件组织结构中的应用

5、排序中的数据结构

- 简单的排序方法（$O(n^2)$）
- 希尔排序（插入）
- 快速排序（交换）
- 归并排序



顺序表

- 堆排序(选择)

完全二叉树

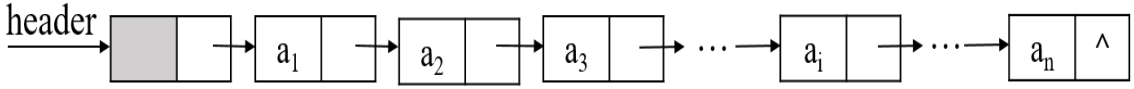
- 基数排序

队列结构

S(n)=O(n+rd)

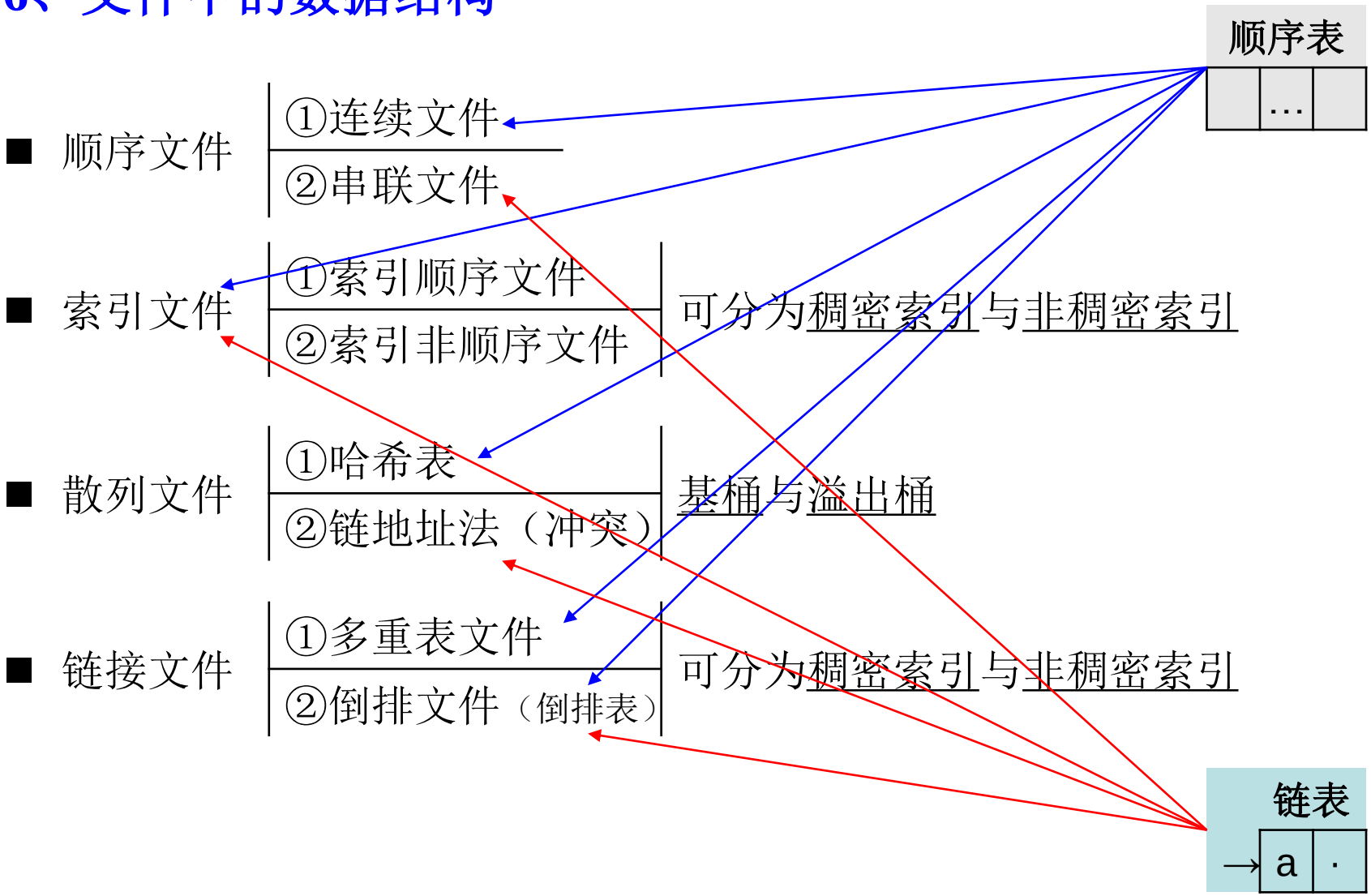
S(n)=O(rd)

链 表



- 外部排序

6、文件中的数据结构


顺序表
…
■ 顺序文件
①连续文件
②串联文件
■ 索引文件
①索引顺序文件
②索引非顺序文件
可分为稠密索引与非稠密索引
■ 散列文件
①哈希表
②链地址法（冲突）
基桶与溢出桶
■ 链接文件
①多重表文件
②倒排文件（倒排表）
可分为稠密索引与非稠密索引
链表
a

二、算法的内容

对算法的基本要求：

- 算法针对的数据对象
- 算法解决的问题是什么？亦即算法的功能
- 算法的输入、输出
- 算法采用的数据结构
- 算法的基本思想
- 算法实现的程序结构
- 算法描述与分析方法

学习算法的目的：

- 验证数据结构
- 积累经验
- 能够模仿
- 学会应用
- 设计算法
- 解决问题
- … …

1、线性表-线性结构上的算法

- **基本操作：**
  - 线性表遍历操作（顺序存储、线性链表）
  - 线性表的反向及逆序遍历（顺序表、线性链表）
  - 线性表的插入和删除操作（顺序表、线性链表）
  - 有序表上的相关操作（二分查找降低时间复杂度）
  - 有序表的归并操作（顺序表、线性链表，O(m+n)）
  - 顺序表的元素划分操作（时间复杂度O(n)）
  - 数组（顺序表）循环移位

- **线性链表：**
  - 单向链表环的问题
    如何判断一个单向链表是否有环？找到环的入口结点。
  - 线性链表，求倒数第k个数
  - 线性链表，求中间位置的元素
  - 单向链表交叉问题
    如何判断两个单向链表是否交叉？找到交叉结点。

- **栈的应用：**
  - 栈的基本操作
  - 栈与递归
  - 表达式处理
  - 迷宫求解

- **队列的应用：**
  - 队列的基本操作
  - 约瑟夫环
  - 舞伴问题
  - 多项式计算

- **串及其操作：**
  - 串的基本操作
  - 朴素模式匹配算法及其分析
  - KMP算法及分析O(m+n)

- **数组及其操作：**
  - 数组的基本操作
  - 对称矩阵压缩存储及操作
  - 上/下三角阵的存储及操作
  - 带状矩阵的存储及操作
  - 稀疏矩阵的存储及操作
    三元组转置的O(m+n)

2、树及二叉树-层次结构上的算法

- **二叉树基本算法：**
  - 二叉树的先序、中序和后序遍历的递归与非递归算法
  - 二叉树的层序遍历
  - 二叉树的建立
  - 二叉树镜像（左右孩子交换）
  - 统计任意二叉树中的结点个数？（全部、或按结点度数统计）

- **二叉树其它操作-1：**
  - 二叉树的深度
  - 二叉树的宽度
  - 判断一颗任意二叉树是否为满二叉树
  - 判断一颗任意二叉树是否为完全二叉树
  - 求二叉树任意结点所在的层号
  - 求任意结点的所有祖先结点（根到该结点的路径）
  - 求二叉树中两个结点的最近公共祖先
  - 二叉链表存储的二叉树转换到按照完全二叉树存储的数组中

- **二叉树其它操作-2：**
  - 任意两个结点的路径
  - 二叉树最大/最短路径
  - 不遍历二叉树，求中序/先序/后序遍历第一个结点或最后一个结点
  - 不用递归，中序遍历线索二叉树
  - 二叉树的复制
  - 二叉树等价判断
  - 堆的插入和删除操作
  - 哈夫曼树的建立
  - 二叉排序树的建立、结点的插入与删除

3、图-网状结构上的算法

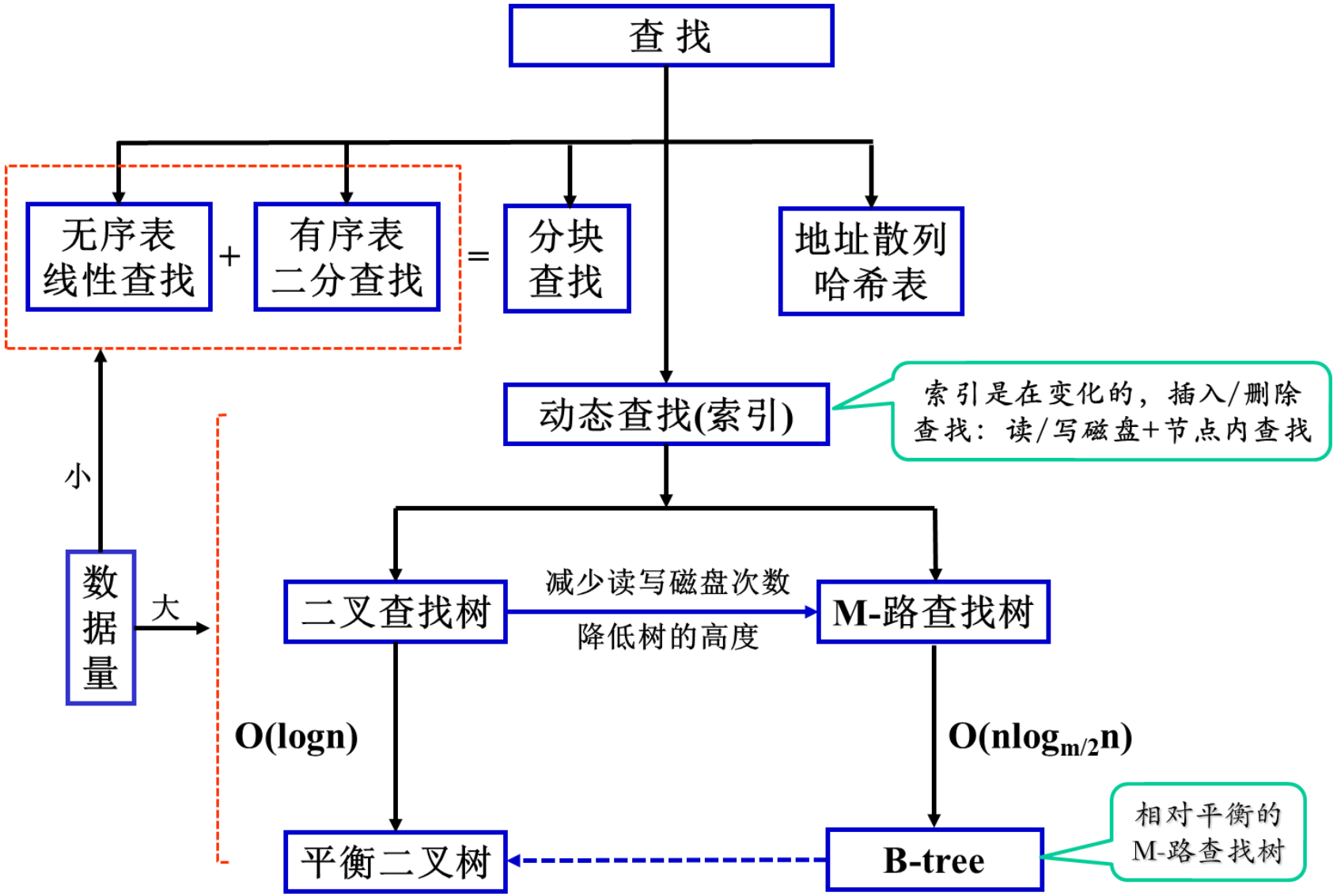
- **图的基本操作：**
  - 图的遍历（邻接表、邻接矩阵，深度优先DFS、广度优先BFS）
  - 图的顶点入度（ID）和出度（OD）的计算（邻接表、邻接矩阵）
  - 图的存储结构相互转换

- **图的其它算法：**
  - 两顶点之间的路径，判断两顶点是否连通
  - 判断图是否连通
  - 求包含顶点*v*的环路
  - 判断图是否有环路

- **图结构上的经典算法：**
  - 最小生成树（prim算法和Kruskal 算法）及其比较分析
  - 关结点及其求解步骤
  - 连通分量、强分量求解方法及步骤
  - 拓扑分类算法及其比较分析
  - 关键路径求解方法及步骤
  - 单源最短路径算法（Dijkstra）及其分析
  - 每一对顶点之间的最短路径算法（Floyd）及其分析

4、查找算法

- 线性查找算法
- 有序表的二分查找算法
- 分块查找的思想及查找步骤
- 二叉查找树的相关操作
- 平衡二叉树（AVL）的判断、构造过程及其算法
- B-树结构特点及其查找过程，插入及删除结点的过程分析
- 地址散列法（哈希表）构造及其查找成功和查找失败的平均查找长度计算


查 找
无序表
线性查找
+
有序表
二分查找
=
分块
查找
地址散列
哈希表
动态查找(索引)
索引是在变化的，插入/删除
查找：读/写磁盘+节点内查找
小
数据量
大
二叉查找树
减少读写磁盘次数
降低树的高度
M-路查找树
O(logn)
O(nlog$_{m/2}$n)
平衡二叉树
B-tree
相对平衡的
M-路查找树

5、排序算法

序号	排序方法	平均时间	最好情况	最坏情况	辅助空间	稳定性
1	简单选择排序	$O(n^2)$	O（n）	$O(n^2)$	O(1)	不稳定
2	直接插入排序	$O(n^2)$	O(n)	$O(n^2)$	O(1)	稳定
3	折半插入排序	$O(n^2)$	$O(n\cdot\log_2 n)$	$O(n^2)$	O(1)	稳定
4	冒泡排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	稳定
5	希尔排序	$O(n^{1.3})$	O（n）	$O(n^2)$	O(1)	不稳定
6	堆排序	$O(n\cdot\log_2 n)$	$O(n\cdot\log_2 n)$	$O(n\cdot\log_2 n)$	O(1)	不稳定
7	归并排序	$O(n\cdot\log_2 n)$	$O(n\cdot\log_2 n)$	$O(n\cdot\log_2 n)$	O(n)	稳定
8	快速排序	$O(n\cdot\log_2 n)$	$O(n\cdot\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	不稳定
9	**基数排序**	**$O(d\cdot(n+r))$** **$O(d\cdot(n+rd))$**	**$O(d\cdot(n+r\cdot d))$**	**$O(d\cdot(n+r))$** **$O(d\cdot(n+rd))$**	**$O(n+r\cdot d)$** **$O(r\cdot d)$**	**稳定**

重点：各类排序算法的设计思想，比较分析

外部排序——归并思想，归并算法的策略

磁盘文件的归并分类

解决的问题：

(1) 多路归并——减少归并遍数,**最佳归并树**
(2) 并行操作的缓冲区处理，**2K个输入缓冲区**
——使输入、输出和CPU处理尽可能重叠
(3) 初始归并段的生成，**置换-选择法**

磁带文件的归并分类

K路平衡归并分类需要2K磁带机数量。

三、关于ADT操作

ADT List {

数据对象：D={ $a_i|a_i \in ElemSet, i=1,2,\ldots,n, n\geq 0$ }

数据关系：R1={ $<a_{i-1},a_i> | a_{i-1},a_i \in D, i=2,\ldots,n$ }

基本操作：

- InitList(&L)
  操作结果：构造一个空的线性表L。

- ListEmpty（L）
  初始条件：线性表L已存在。
  操作结果：若L为空表，则返回TRUE ，否则返回FALSE。

- LocateElem(L, e, compare())
  初始条件：线性表L已存在，compare()是数据元素判定函数。
  操作结果：返回L中第1个与e满足关系compare()的数据元素的位序。
  若这样的数据元素不存在，则返回值为0。

- PriorElem(L, cur_e, &pre_e)
  初始条件：线性表L已存在。
  操作结果：若cur_e是L中的数据元素，且不是第1个元素，则用pre_e返回它的前驱，否则操作失败，pre_e无定义。

- NextElem(L, cur_e, &next_e)
  初始条件：线性表L已存在。
  操作结果：若cur_e是L中的数据元素，且不是最后1个元素，则用next_e返回它的后继，否则操作失败，next_e无定义。

- ListInsert(&L, i, e ）
  初始条件：线性表L已存在，1≤i≤ListLength（L）+1。
  操作结果：在L中第i个位置之前插入新的数据元素e, 线性表L的长度加1。

- ListDelete(&L, i, &e ）
  初始条件：线性表L已存在且非空，1≤i≤ListLength（L）。
  操作结果：删除线性表L中第i个数据元素，并用e返回, L的长度减1。
- ……

}ADT List

ADT Stack {

数据对象：D={ $a_i|a_i \in ElemSet, i=1,2,\ldots,n, n\geq 0$ }

数据关系：R1={ $<a_{i-1},a_i> | a_{i-1},a_i \in D, i=2,\ldots,n$ }

约定a_n端为栈顶， a_1端为栈底。

基本操作：

- InitStack(&S)

  操作结果：构造一个空栈 S。

- StackEmpty（S）

  初始条件：栈S已存在。

  操作结果：若栈S为空栈，返回TRUE ， 否则返回FALSE。

- GetTop(S, &e)

  初始条件：栈S已存在且非空。

  操作结果：用e返回栈S的栈顶元素。

- Push(&S, e)

  初始条件：栈S已存在。

  操作结果：插入元素e 为新的栈顶元素。

- Pop(&S, &e)

  初始条件：栈S已存在且非空。

  操作结果：删除S的栈顶元素，并用e返回其值。

} ADT Stack

ADT Queue {

数据对象：D={ $a_i|a_i \in ElemSet, i=1,2,\ldots,n, n\geq 0$ }

数据关系：R1={ $<a_{i-1},a_i> | a_{i-1},a_i \in D, i=2,\ldots,n$ }

约定其中a_1端为队列头， a_n端为队列尾。

基本操作：

- InitQueue(&Q)
  操作结果：构造一个空队列 Q。
- QueueEmpty（Q）
  初始条件：队列Q已存在。
  操作结果：若Q为空队列，则返回TRUE ， 否则返回FALSE。
- GetHead(Q, &e)
  初始条件：队列Q已存在且非空。
  操作结果：用e返回队列Q的队头元素。
- EnQueue(&Q, e)
  初始条件：队列Q已存在。
  操作结果：插入元素e 为队列Q新的队尾元素。
- DeQueue(&Q, &e)
  初始条件：队列Q已存在且非空。
  操作结果：删除Q的队头元素，并用e返回其值。

} ADT Queue

ADT Tree {

数据对象D：D是具有相同类型的数据元素的集合。

数据关系R：若 D 为空集，则称为空树；

若 D 仅含一个数据元素，则 R 为空集，否则 R = { H }，H 是如下的二元关系：

（1）在 D 中存在唯一的称为根的数据元素 root ，它在关系 H 下无前驱；

（2）若 D - { root } ≠Φ，则存在D-{root}的一个划分$D_1,D_2,\ldots,D_m(m>0)$，对任意 $j \neq k$（$1 \le j, k \le m$）有$D_j \cap D_k = \Phi$ ，且对任意的$i(1 \le i \le m)$，唯一存在数据元素 $x_i \in D_i$，有 $< root , x_i > \in H$;

（3）对应于D-{ root }的划分，H-{<root,x_1>,…,<root,x_m>}有唯一的划分$H_1,H_2,\ldots,H_m$（m＞0）,对任意$j \neq k$（$1 \le j, k \le m$）有$H_j \cap H_k \neq \Phi$,且对任意的i（$1 \le i \le m$），H_i是D_i上的二元关系，（D_i，{H_i}）是一棵符合本定义的树，称为根root的子树。

基本操作：

- InitTree(&T)

  操作结果：构造一棵空树T。

Tree基本操作（续）：

- TreeEmpty（ T）
  初始条件：树T已存在。
  操作结果：若T为空树，则返回TRUE，否则返回FALSE 。
- Root(T)
  初始条件：树T已存在。
  操作结果：返回树T的根。
- Parent(T, cur_e)
  初始条件：树T已存在，cur_e是树T中的某个结点。
  操作结果：若cur_e是T的非根结点，则返回它的双亲 ，否则返回空值。
- LeftChild(T, cur_e)
  初始条件：树T已存在，cur_e是树T中的某个结点。
  操作结果：若cur_e是T的非叶子结点，返回它的最左孩子，否则返回空值。
- RightSibling(T, cur_e)
  初始条件：树T已存在，cur_e是树T中的某个结点。
  操作结果：若cur_e有右兄弟，返回它的右兄弟，否则返回空值。

} ADT Tree

ADT BinaryTree {

数据对象D：D是具有相同类型的数据元素的集合。

数据关系R：

若 D=Φ，则R=Φ，称BinaryTree为空二叉树；

若 D≠Φ，则 R={ H }，H 是如下的二元关系：

（1）在 D 中存在唯一的称为根的数据元素 root ，它在关系H下无前驱；

（2）若 D - { root } ≠Φ，则存在D-{root}={D_l,D_r},且$D_l \cap D_r = \Phi$ ；

（3）若$D_l \neq \Phi$，则D_l中存在唯一的元素x_l，<root,x_l>∈H，且存在D_l上的关系$H_l \subset H$；若$D_r \neq \Phi$，则D_r中存在唯一的元素x_r，<root,x_r>∈H，且存在D_r上的关系$H_r \subset H$；H={<root, x_l>，<root, x_r>,H_l,H_r} ；

（4）（D_l，{H_l}）是一棵符合本定义的二叉树，称为根的左子树；

（D_r，{H_r}）是一棵符合本定义的二叉树，称为根的右子树。

基本操作：

- InitBiTree(&T)

  操作结果：构造一棵空的二叉树T。

- BiTreeEmpty(T)

  初始条件：二叉树T已存在。

  操作结果：若T为空二叉树，则返回TRUE，否则返回FALSE。

BinaryTree基本操作（续）：

- Root(T)
  初始条件：二叉树T已存在。
  操作结果：返回二叉树T的根。

- Parent(T, cur_e)
  初始条件：二叉树T已存在，cur_e是二叉树T中的某个结点。
  操作结果：若cur_e是T的非根结点，则返回它的双亲，否则返回空值。

- LeftChild(T, cur_e)
  初始条件：二叉树T已存在，cur_e是二叉T中某结点。
  操作结果：返回cur_e的左孩子，若cur_e无左孩子则返回空值。

- RightChild(T, cur_e)
  初始条件：二叉树T已存在，cur_e是二叉T中某结点。
  操作结果：返回cur_e的右孩子，若cur_e无右孩子则返回空值。

} ADT BinaryTree

ADT Graph {

数据对象V：V是具有相同特性的数据元素的集合，称为顶点集。

数据关系R：

R = { VR }

VR = { <v , w >|v, w ∈V,且P(v,w),<v, w>表示从v到w的弧，
谓词P(v, w)定义了弧 <v, w>的意义或信息 }

基本操作：

- FirstAdjVex (G, v)
  初始条件：图G存在，v是G中的某个顶点。
  操作结果：返回图G中与顶点v邻接的第一个邻接点，若顶点v没有邻接点，则返回“空”。

- NextAdjVex (G, v, w)
  初始条件：图G存在，v是G中的某个顶点，w是v的邻接顶点。
  操作结果：返回图G中与顶点v邻接的w之后的邻接点，若w是v的最后一个邻接点，则返回“空”。

} ADT Graph

四、这门课的知识点

第一章 绪论

A数据结构研究对象

信息　　数据　　数据元素　　数据项

数据结构　　数据对象　　数据类型

B数据结构

逻辑结构　　存储结构（物理结构）　　数据结构分类

C数据结构发展概况

D抽象数据型(ADT)

数据型、数据结构与抽象数据型

抽象数据型的规格描述（语法、语义）

抽象数据型的实现

抽象数据型的优点

多层次抽象技术

E算法

什么叫算法？　算法的特征　“好”的算法的评价标准　对算法的正确性的要求

算法的描述　　类语言

F算法分析

算法的时间特性　　时间复杂度T(n)　空间复杂度S(n)

第二章 线性表

A线性表的概念
什么叫线性表　　抽象数据型线性表

B线性表的实现
静态数据结构　　动态数据结构
顺序存储（数组实现）　链式存储（指针）　游标（静态链表）

C线性链表
表头结点　单向链表　双向链表　单向循环量表　双向循环链表

D限定性数据结构：　栈　&　队列

E栈
栈的概念　　ADT栈　　栈的存储结构
栈的应用:　　栈与递归　迷宫求解　　表达式转换与求值

F队列
队列的概念　ADT队列　队列的存储结构　　循环队列

G线性表的应用：　多项式的表示　多项式相加运算

H串

串的基本概念　ADT串　　串的存储结构　　存储密度

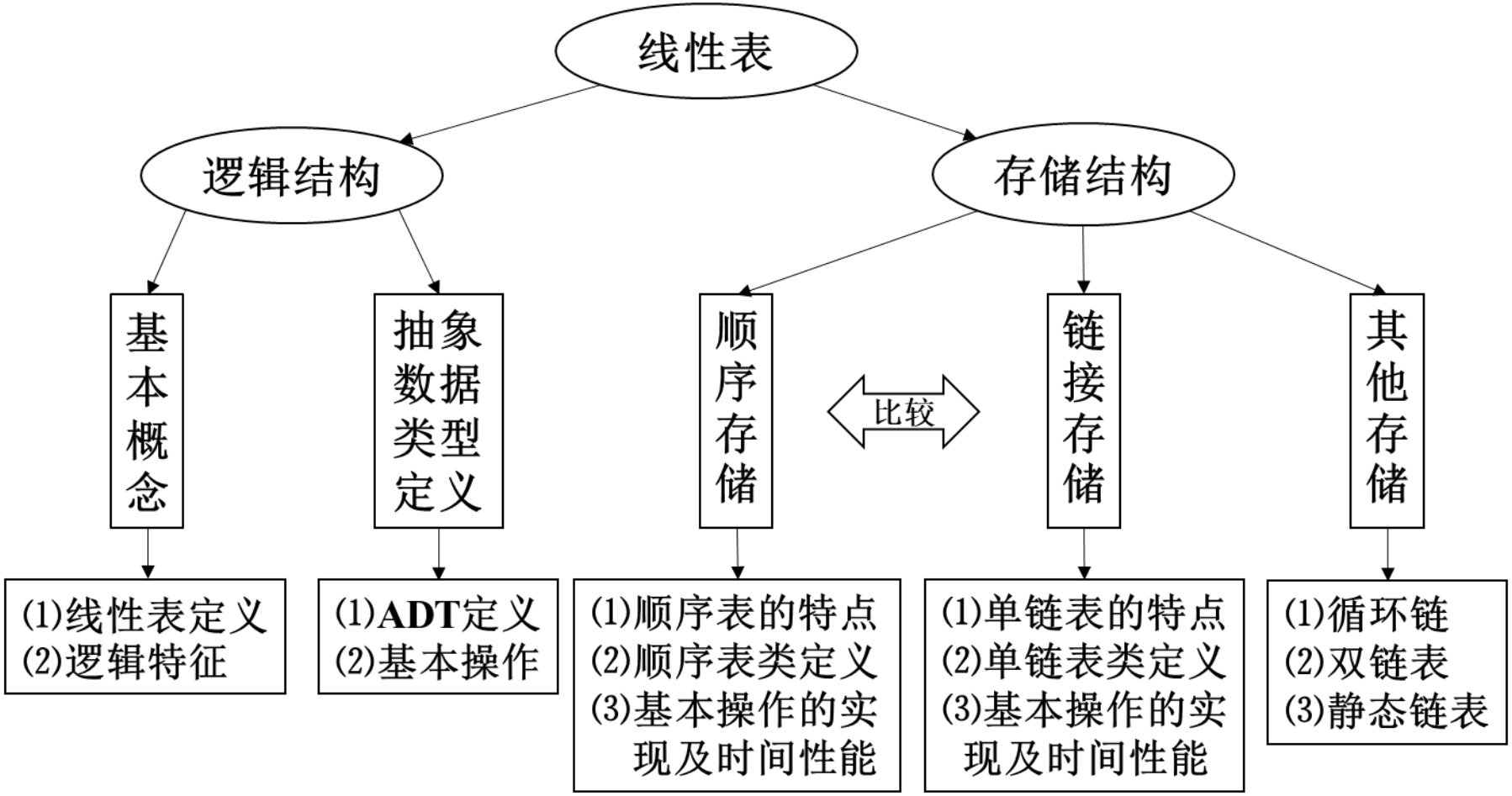
I 数组

数组的概念　ADT数组　数组的存储结构

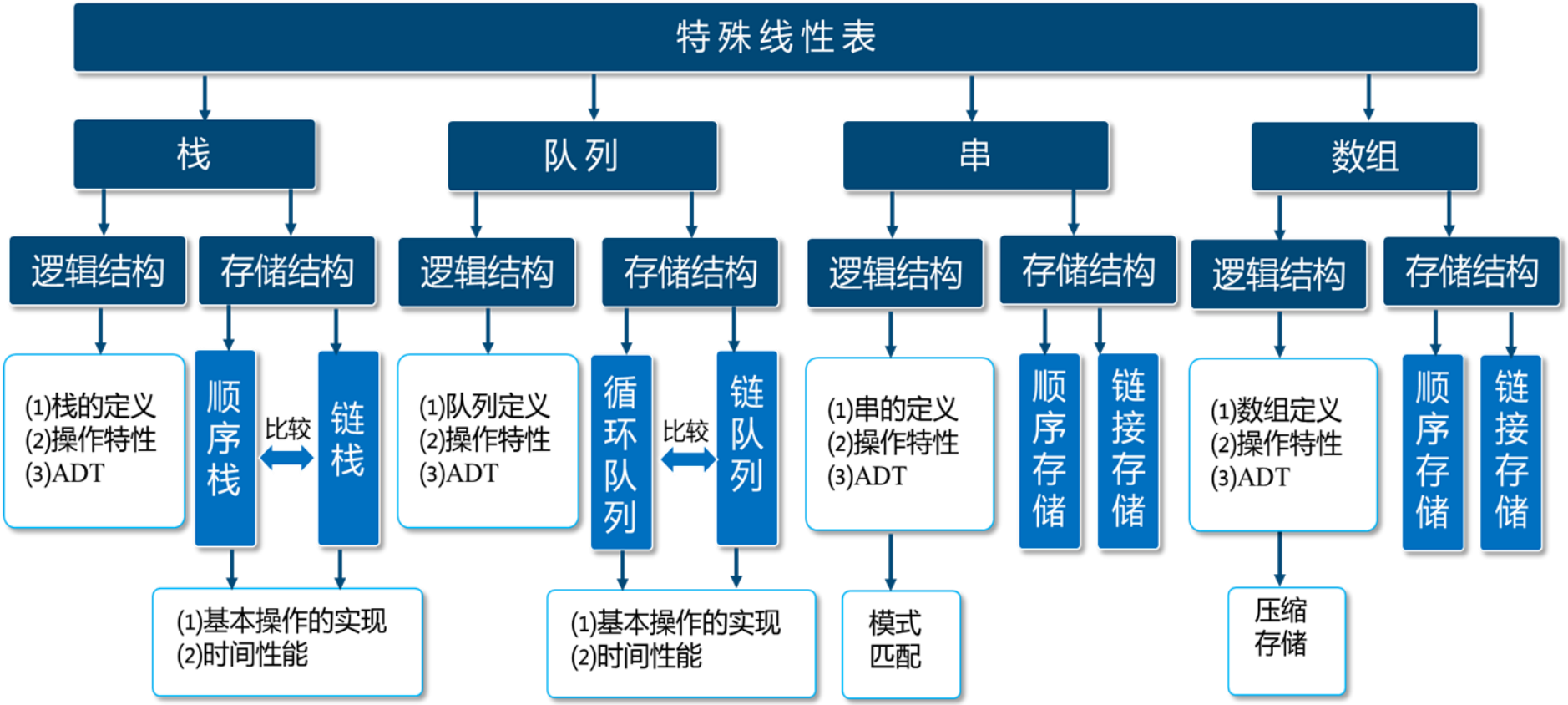
数组的压缩存储：特殊矩阵、对角或带状矩阵、稀疏矩阵

J广义表

基本概念　　广义表的存储结构


线性表
逻辑结构
存储结构
基本概念
抽象数据类型定义
顺序存储
比较
链接存储
其他存储
(1)线性表定义
(2)逻辑特征
(1)ADT定义
(2)基本操作
(1)顺序表的特点
(2)顺序表类定义
(3)基本操作的实现及时间性能
(1)单链表的特点
(2)单链表类定义
(3)基本操作的实现及时间性能
(1)循环链
(2)双链表
(3)静态链表

线性表小结


特殊线性表
栈
队列
串
数组
逻辑结构
存储结构
逻辑结构
存储结构
逻辑结构
存储结构
逻辑结构
存储结构
(1)栈的定义
(2)操作特性
(3)ADT
顺序栈
比较
链栈
(1)队列定义
(2)操作特性
(3)ADT
循环队列
比较
链队列
(1)串的定义
(2)操作特性
(3)ADT
顺序存储
链接存储
(1)数组定义
(2)操作特性
(3)ADT
顺序存储
链接存储
(1)基本操作的实现
(2)时间性能
(1)基本操作的实现
(2)时间性能
模式匹配
压缩存储

第三章 树与二叉树

A树的基本术语

树　子树　结点　分支　度　路
叶子　非终端结点　终端结点　儿子　父亲　兄弟
堂兄弟　祖先　子孙结点　层　高度（深度）
结点的顺序　层序　有序树　无序树
森林　等等

B二叉树

二叉树的定义，ADT二叉树，满二叉树，完全二叉树
二叉树的遍历：先序遍历、中序遍历和后序遍历，层序遍历
二叉树遍历的非递归算法（先序、中序和后序）
二叉树的性质：（1～5）
二叉树的存储结构：顺序存储、链式存储（二叉链表）
线索二叉树：基本概念，先序、中序与后序线索
求线索二叉树的（先序、中序、后序）前驱与后继结点
线索二叉树的遍历
线索二叉树中插入、删除结点的讨论

C二叉树的相似与等价，二叉树的复制算法

D树

ADT树

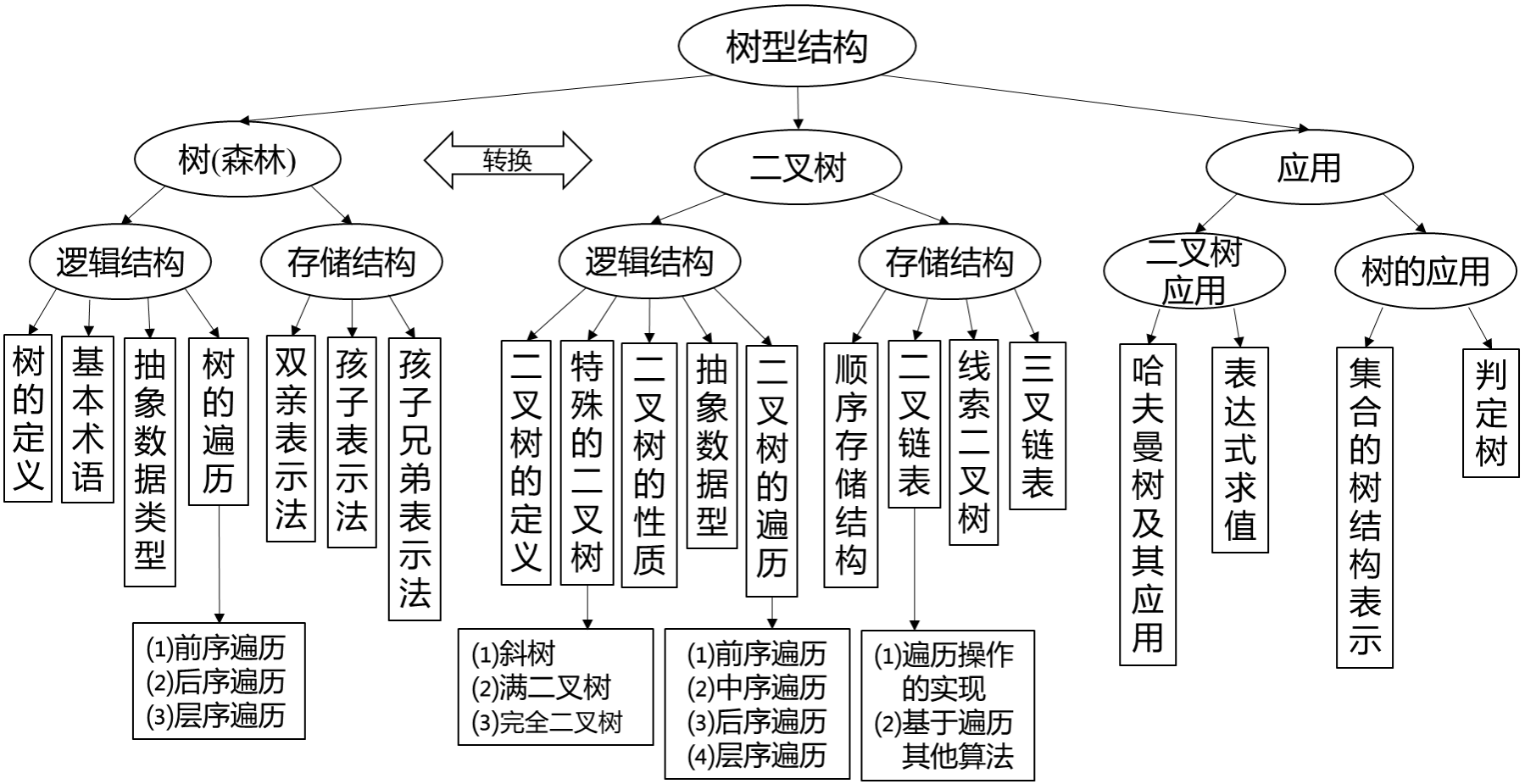
树的存储结构：双亲表示法，孩子表示法（邻接表），树的左右链表示

树与二叉树的转换，森林与二叉树的转换

树的遍历：先序、中序和后序遍历树

E树的应用

用树表示集合、判定树、哈夫曼树及其应用、最优编码


树型结构
树(森林)
转换
二叉树
应用
逻辑结构
存储结构
逻辑结构
存储结构
二叉树应用
树的应用
树的定义
基本术语
抽象数据类型
树的遍历
双亲表示法
孩子表示法
孩子兄弟表示法
二叉树的定义
特殊的二叉树
二叉树的性质
抽象数据型
二叉树的遍历
顺序存储结构
二叉链表
线索二叉树
三叉链表
哈夫曼树及其应用
表达式求值
集合的树结构表示
判定树
(1)前序遍历
(2)后序遍历
(3)层序遍历
(1)斜树
(2)满二叉树
(3)完全二叉树
(1)前序遍历
(2)中序遍历
(3)后序遍历
(4)层序遍历
(1)遍历操作的实现
(2)基于遍历其他算法

第四章 图以及与图有关的算法

A图的基本概念

图的定义　ADT图　有向图　无向图　弧　边 顶点　邻接点　相邻　依附
环路　权　子图　带标号的图（网） 路径　简单路径　连通图　强连通图
连通分量　强连通分量　完全图　稀疏图　稠密图　度 入度 出度 生成树

B图的表示（存储结构）：　邻接矩阵　邻接表

C图的遍历（搜索）算法：　先深搜索（DFS）　先广搜索（BFS）

D图与树的关系

生成树　先深生成森林　先广生成森林　树边与回退边　开放树
最小生成树及其算法（MST性质、Prim、Kruskal算法）

E无向图的双连通性

关结点 双连通图　双连通分量

F有向图的搜索

生成树　生成森林　如何区别树边、向前边、回退边和 横边

G强连通性：强连通分量，归约图 ，图的中心点的概念及求解方法

- 图（有向图、无向图）与树的联系
  深度优先（先深dfs）/广度优先(先广bfs)遍历(算法)
  先深生成森林和先广生成森林
  树边、非树边

- 无向图/网
  开放树、最小生成树（算法）
  由边的等价，引出双连通分量
  关节点、双连通图

- 有向图/网
  树边、向前边、回退边、和横边
  由顶点的等价，引出强分量
  强连通图
  归约图

- 有向无环图
  拓扑排序(算法)、关键路径(算法)

H拓扑分类

有向无环图及其应用　拓扑分类　拓扑分类算法

I 关键路径

AOE网　AOV网事件　活动　路径长度　关键活动　关键路径

AOE网问题：(1)完成整个工程至少需要多少时间？

(2)哪些活动是影响工程进度的关键活动？

关键路径算法中的关键变量：

① 事件 V_j 的最早可能发生时间 $VE(j)$；

② 活动 a_i 的最早可能开始时间 $E(k)$；

③ 事件 V_k 的最迟发生时间 $VL(k)$；

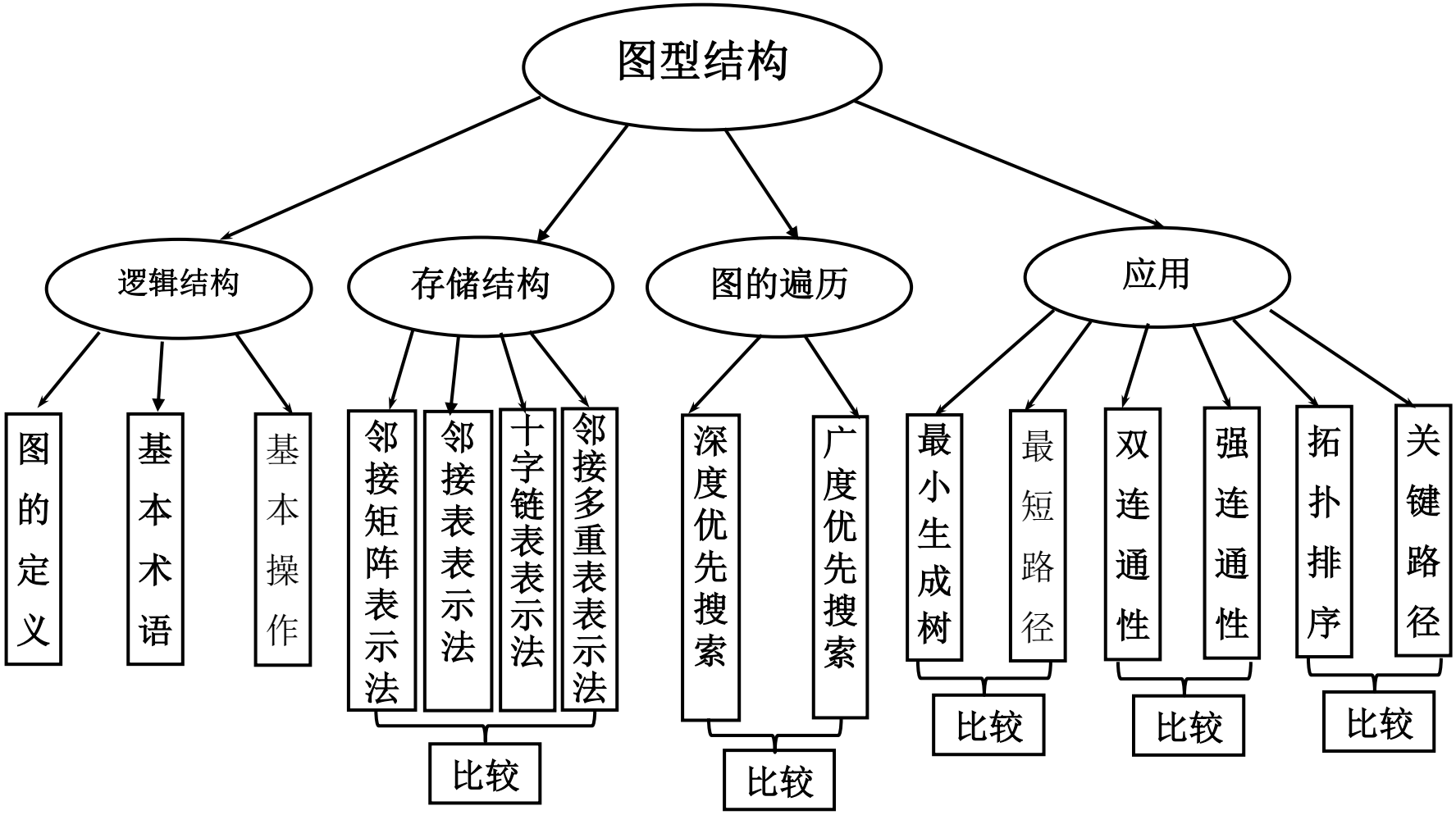
④ 活动 a_i 的最迟允许开始时间 $L(i)$；

⑤ 时间余量 $L(i)-E(i)$。

J最短路径问题

单源最短路径：Dijkstra算法

任意顶点间的最短路径：Floyd算法、Warshall算法


图型结构
逻辑结构
存储结构
图的遍历
应用
图的定义
基本术语
基本操作
邻接矩阵表示法
邻接表表示法
十字链表表示法
邻接多重表表示法
比较
深度优先搜索
广度优先搜索
比较
最小生成树
最短路径
比较
双连通性
强连通性
比较
拓扑排序
关键路径
比较

第五章 查找

A基本概念

查找（检索） 查找表 关键字 静态查找 动态查找 平均查找长度

B线性查找

C折半查找：条件

D分快查找

EAVL树

FB-树B+树

G二叉查找树：什么叫二叉查找树、插入结点、删除结点、查找结点

H散列法：哈稀函数 冲突 哈希表的长度

哈希函数：直接定址法 质数除余法 平方取中法

折叠法 数字分析法 随机法

处理冲突：开放定址法（线性探测、二次探测）

再散列法 链地址法 建立公共溢出区

I装填因子：$\alpha=\dfrac{\text{表中装入的记录数}}{\text{哈希表的长度}}$

装填因子α标志着哈希表的装满程度，α越小，发生冲突的可能性越小，反之，发生冲突的可能性越大。

J成功查找平均查找长度：ASL_s

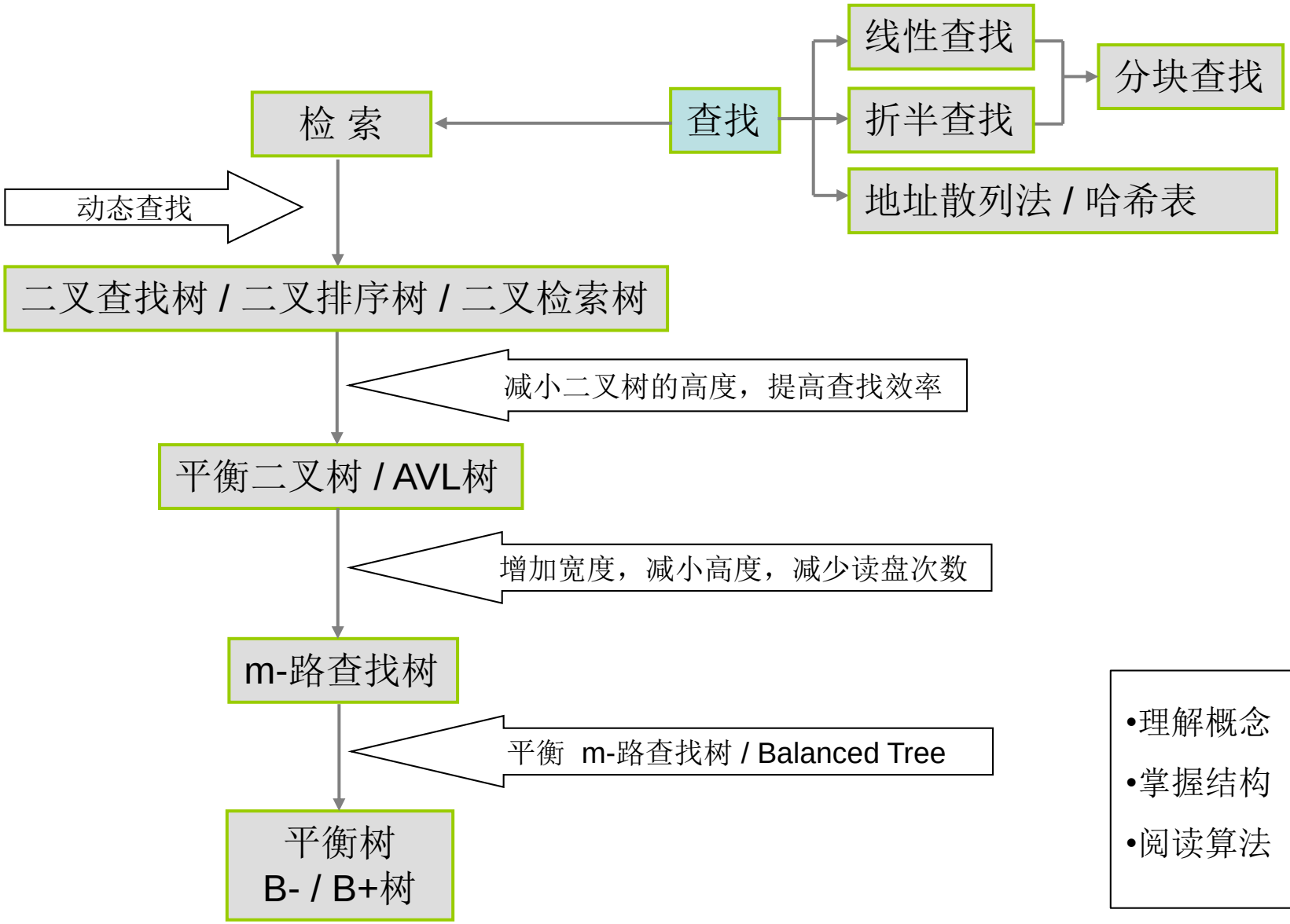
查找到散列表中已存在结点的平均比较次数。

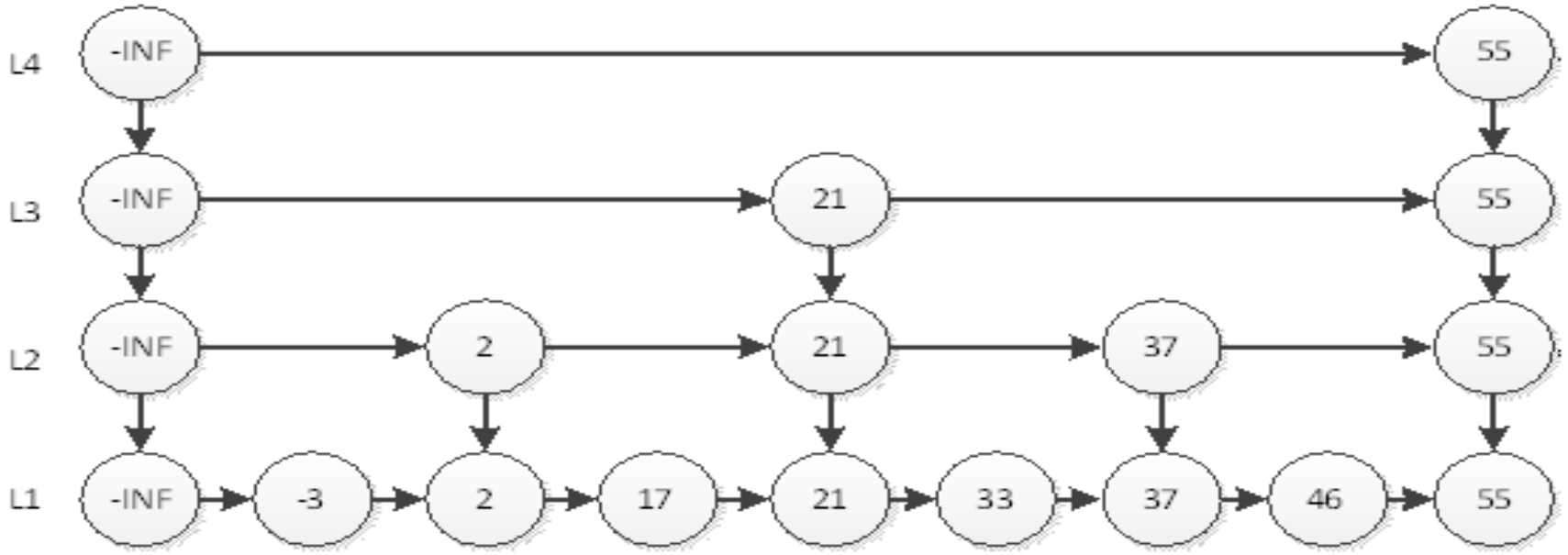
K失败查找平均查找长度：ASL_u

查找失败，但找到插入位置的平均比较次数。

L几种处理冲突方法的平均查找长度

处理冲突方法	查找成功	查找失败（插入记录）
线性探测法	$\frac{1}{2}(1+\frac{1}{1-\alpha})$	$\frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$
二次探测法 再散列法	$-(\frac{1}{\alpha})\ln(1-\alpha)$	$\frac{1}{1-\alpha}$
链地址法	$1+\frac{\alpha}{2}$	$\alpha+e^{-\alpha}$


查找
检 索
线性查找
折半查找
分块查找
地址散列法 / 哈希表
动态查找
二叉查找树 / 二叉排序树 / 二叉检索树
减小二叉树的高度，提高查找效率
平衡二叉树 / AVL树
增加宽度，减小高度，减少读盘次数
m-路查找树
平衡 m-路查找树 / Balanced Tree
平衡树
B- / B+树
•理解概念
•掌握结构
•阅读算法



Skip List

又称跳跃表

简称跳表

- redis和levelDB都是用了它；
- 他在有序链表的基础上进行扩展；
- 解决了有序链表结构查找特定值困难的问题；
- 查找特定值的时间复杂度为O(logn)；
- 一种可以代替平衡树的数据结构；

第六章 内部排序（分类）

A排序（分类）

排序　内部排序　外部排序　稳定与不稳定的排序方法

B影响分类性能的因素：

比较关键字的次数—当关键字是字符串时，是主要因素；

交换记录位置和移动记录的次数—当记录很大时，是主要因素。

C简单的分类算法：

气泡排序　插入排序　冒泡（选择）排序　共同特点 $O(n^2)$

DShell分类：缩小增量法

E快速分类: 快速分类的递归算法与非递归算法

F归并分类

G堆 分 类：堆的概念　整理堆的算法

H基数分类（多关键字）

第七章 外部排序（分类）

A归并方法：首先将文件中的数据输入到内存，采用内部分类方法进行分类（归并段），然后将有序段写回外存；对多归并段（已有序）进行多遍合并（归并），最后形成一个有序序列。

B磁盘文件的归并分类

(1) 多路归并——减少归并遍数

(2) 并行操作的缓冲区处理——使输入、输出和CPU处理尽可能重叠

(3) 初始归并段的生成

◇ *m*个初始段进行 *2* 路归并，需要 $log_2 m$ 遍归并；

一般地，*m* 个初始段，采用*k*路归并，需要 $log_k m$ 遍归并。显然，*k*越大，归并遍数越少，可提高归并的效率。

◇在 *k* 路归并时，从 *k*个关键字中选择最小记录时，要比较*K-1*次。若记录总数为*n*，每遍要比较的次数为：*n*(k-1)[log2m/log2k]*

◇选择树或败者树，*k*路归并时间 $O(n \cdot log_2 m)$ ，与*k*无关，最佳归并树。

C磁带文件的归并分类： *k* 路平衡归并分类。

复习时注意几个问题

知识的连贯性：认真读书、**尊重教材**、同时要注意参考其它教材 或资料

注意基本概念：名词的理解、**问题的研究对象**

存储结构：线性表、树与二叉树、图 ， 查找、排序算法的数据组织方式

注意顺序结构与链式结构的比较（各种应用）

算法设计：能够按要求设计算法（方法步骤、伪码、代码）

算法分析：时间复杂度T(n)，空间复杂度S(n)，（排序）ASL、稳定性

题目类型：选择填空、应用/简答、算法设计

卷面70/100分，平时成绩=作业10分+实验20分

答疑

最后一次线下答疑：

时间：2021年5月29日

14:00-16:00

地址：信息楼15楼中庭

授课老师：张正

地址：信息楼1519

邮件：zhengzhang@hit.edu.cn

课程助教：

黄裕涛（13班），QQ：278580633

李逢君（14班），QQ：543877815

李浩宇（15班），QQ：412689719

何素妍（16班），QQ：1872954023

赵书光，QQ：1329777148

希望通过本课的学习，为同学们打下专业基础！

欢迎大家提出宝贵的意见或建议！

祝同学们取得好的成绩！

谢谢大家！

张 正
zhengzhang@hit.edu.cn
http://faculty.hitsz.edu.cn/zhangzheng

附录：ADT 描述

抽象数据类型线性表的定义如下：

ADT List {

数据对象：$D=\{ a_i \mid a_i \in ElemSet,\ i=1,2,\cdots,n,\ n\geqslant 0 \}$

数据关系：$R1=\{ <a_{i-1},a_i> \mid a_{i-1},a_i \in D,\ i=2,\cdots,n \}$

基本操作：

InitList(&L)

操作结果：构造一个空的线性表 L。

DestroyList(&L)

初始条件：线性表 L 已存在。

操作结果：销毁线性表 L。

ClearList(&L)

初始条件：线性表 L 已存在。

操作结果：将 L 重置为空表。

ListEmpty(L)

初始条件：线性表 L 已存在。

操作结果：若 L 为空表，则返回 TRUE，否则返回 FALSE。

ListLength(L)

初始条件：线性表 L 已存在。

操作结果：返回 L 中数据元素个数。

GetElem(L, i, &e)

初始条件：线性表 L 已存在，1≤i≤ListLength(L)。

操作结果：用 e 返回 L 中第 i 个数据元素的值。

LocateElem(L, e, compare())

初始条件：线性表 L 已存在，compare()是数据元素判定函数。

操作结果：返回 L 中第 1 个与 e 满足关系 compare()的数据元素的位序。若这样的数据元素不存在，则返回值为 0。

PriorElem(L, cur_e, &pre_e)

初始条件：线性表 L 已存在。

操作结果：若 cur_e 是 L 的数据元素，且不是第一个，则用 pre_e 返回它的前驱，否则操作失败，pre_e 无定义。

NextElem(L, cur_e, &next_e)

初始条件:线性表 L 已存在。

操作结果:若 cur_e 是 L 的数据元素,且不是最后一个,则用 next_e 返回它的后继,否则操作失败,next_e 无定义。

ListInsert(&L, i, e)

初始条件:线性表 L 已存在,1≤i≤ListLength(L) + 1。

操作结果:在 L 中第 i 个位置之前插入新的数据元素 e,L 的长度加 1。

ListDelete(&L, i, &e)

初始条件:线性表 L 已存在且非空,1≤i≤ListLength(L)。

操作结果:删除 L 的第 i 个数据元素,并用 e 返回其值,L 的长度减 1。

ListTraverse(L, visit())

初始条件:线性表 L 已存在。

操作结果:依次对 L 的每个数据元素调用函数 visit()。一旦 visit()失败,则操作失败。

} **ADT** List

抽象数据型：栈

ADT Stack {

数据对象：D = { a_i | $a_i \in$ ElemSet, i = 1,2,…,n, n≥0 }

数据关系：R1 = { <a_{i-1}, a_i> | a_{i-1}, $a_i \in$ D, i = 2,…,n }

约定 a_n 端为栈顶，a_1 端为栈底。

基本操作：

InitStack(&S)

操作结果：构造一个空栈 S。

DestroyStack(&S)

初始条件：栈 S 已存在。

操作结果：栈 S 被销毁。

ClearStack(&S)

初始条件：栈 S 已存在。

操作结果：将 S 清为空栈。

StackEmpty(S)

初始条件：栈 S 已存在。

操作结果：若栈 S 为空栈，则返回 TRUE，否则 FALSE。

StackLength(S)

初始条件：栈 S 已存在。

操作结果：返回 S 的元素个数，即栈的长度。

GetTop(S, &e)

初始条件：栈 S 已存在且非空。

操作结果：用 e 返回 S 的栈顶元素。

Push(&S, e)

初始条件：栈 S 已存在。

操作结果：插入元素 e 为新的栈顶元素。

Pop(&S, &e)

初始条件：栈 S 已存在且非空。

操作结果：删除 S 的栈顶元素，并用 e 返回其值。

StackTraverse(S,visit())

初始条件：栈 S 已存在且非空。

操作结果：从栈底到栈顶依次对 S 的每个数据元素调用函数 visit()。一旦 visit()失败，则操作失效。

}ADT Stack

抽象数据型：队

ADT Queue {

数据对象：$D = \{a_i \mid a_i \in ElemSet,\ i = 1,2,\cdots,n,\ n \geqslant 0\}$

数据关系：$R1 = \{<a_{i-1}, a_i> \mid a_{i-1}, a_i \in D,\ i = 2,\cdots,n\}$

约定其中 a_1 端为队列头，a_n 端为队列尾。

基本操作：

InitQueue(&Q)

操作结果：构造一个空队列 Q。

DestroyQueue(&Q)

初始条件：队列 Q 已存在。

操作结果：队列 Q 被销毁，不再存在。

ClearQueue(&Q)

初始条件：队列 Q 已存在。

操作结果：将 Q 清为空队列。

QueueEmpty(Q)

初始条件：队列 Q 已存在。

操作结果：若 Q 为空队列，则返回 TRUE，否则 FALSE。

QueueLength(Q)

初始条件：队列 Q 已存在。

操作结果：返回 Q 的元素个数，即队列的长度。

GetHead(Q, &e)

初始条件:Q 为非空队列。

操作结果:用 e 返回 Q 的队头元素。

EnQueue(&Q, e)

初始条件:队列 Q 已存在。

操作结果:插入元素 e 为 Q 的新的队尾元素。

DeQueue(&Q, &e)

初始条件:Q 为非空队列。

操作结果:删除 Q 的队头元素,并用 e 返回其值。

QueueTraverse(Q,visit())

初始条件:Q 已存在且非空。

操作结果:从队头到队尾,依次对 Q 的每个数据元素调用函数 visit()。一旦 visit() 失败,则操作失败。

}ADT Queue

抽象数据型：树

ADT Tree {

数据对象 D:D 是具有相同特性的数据元素的集合。

数据关系 R:若 D 为空集,则称为空树;

若 D 仅含一个数据元素,则 R 为空集,否则 R = {H},H 是如下二元关系:

(1) 在 D 中存在惟一的称为根的数据元素 root,它在关系 H 下无前驱;

(2) 若 D - {root} ≠ Φ,则存在 D - {root} 的一个划分 $D_1, D_2, \cdots, D_m (m>0)$,对任意 $j \neq k (1 \leqslant j,k \leqslant m)$ 有 $D_j \cap D_k = \Phi$,且对任意的 $i (1 \leqslant i \leqslant m)$,惟一存在数据元素 $x_i \in D_i$,有 $<root, x_i> \in H$;

(3) 对应于 D - {root} 的划分,$H - \{<root, x_1>, \cdots, <root, x_m>\}$ 有惟一的一个划分 $H_1, H_2, \cdots, H_m (m>0)$,对任意 $j \neq k (1 \leqslant j,k \leqslant m)$ 有 $H_j \cap H_k = \Phi$,且对任意 $i (1 \leqslant i \leqslant m)$,$H_i$ 是 D_i 上的二元关系,$(D_i, \{H_i\})$ 是一棵符合本定义的树,称为根 root 的子树。

基本操作 P:

InitTree(&T);

操作结果:构造空树 T。

DestroyTree(&T);

初始条件:树 T 存在。

操作结果:销毁树 T。

CreateTree(&T, definition);

初始条件:definition 给出树 T 的定义。

操作结果:按 definition 构造树 T。

ClearTree(&T);

初始条件:树 T 存在。

操作结果:将树 T 清为空树。

TreeEmpty(T);

初始条件:树 T 存在。

操作结果:若 T 为空树,则返回 TRUE,否则 FALSE。

TreeDepth(T);

初始条件:树 T 存在。

操作结果:返回 T 的深度。

Root(T);

初始条件:树 T 存在。

操作结果:返回 T 的根。

Value(T, cur_e);

初始条件:树 T 存在,cur_e 是 T 中某个结点。

操作结果:返回 cur_e 的值。

Assign(T, cur_e, value);

初始条件：树 T 存在，cur_e 是 T 中某个结点。

操作结果：结点 cur_e 赋值为 value。

Parent(T, cur_e);

初始条件：树 T 存在，cur_e 是 T 中某个结点。

操作结果：若 cur_e 是 T 的非根结点，则返回它的双亲，否则函数值为“空”。

LeftChild(T, cur_e);

初始条件：树 T 存在，cur_e 是 T 中某个结点。

操作结果：若 cur_e 是 T 的非叶子结点，则返回它的最左孩子，否则返回“空”。

RightSibling(T, cur_e);

初始条件：树 T 存在，cur_e 是 T 中某个结点。

操作结果：若 cur_e 有右兄弟，则返回它的右兄弟，否则函数值为“空”。

InsertChild(&T, &p, i, c);

初始条件：树 T 存在，p 指向 T 中某个结点，1≤i≤p 所指结点的度 + 1，非空树 c 与 T 不相交。

操作结果：插入 c 为 T 中 p 指结点的第 i 棵子树。

DeleteChild(&T, &p, i);

初始条件：树 T 存在，p 指向 T 中某个结点，1≤i≤p 指结点的度。

操作结果：删除 T 中 p 所指结点的第 i 棵子树。

TraverseTree(T, Visit());

初始条件：树 T 存在，Visit 是对结点操作的应用函数。

操作结果：按某种次序对 T 的每个结点调用函数 visit()一次且至多一次。一旦 visit()失败，则操作失败。

}ADT Tree

抽象数据型：二叉树

ADT BinaryTree {

数据对象 D：D 是具有相同特性的数据元素的集合。

数据关系 R：

若 D = Φ，则 R = Φ，称 BinaryTree 为空二叉树；

若 D≠Φ，则 R = {H}，H 是如下二元关系：

(1) 在 D 中存在惟一的称为根的数据元素 root，它在关系 H 下无前驱；

(2) 若 D − {root} ≠ Φ，则存在 D − {root} = {D_l, D_r}，且 $D_l \cap D_r = \Phi$；

(3) 若 $D_l \neq \Phi$，则 D_l 中存在惟一的元素 x_l，<root, x_l> ∈ H，且存在 D_l 上的关系 $H_l \subset H$；若 $D_r \neq \Phi$，则 D_r 中存在惟一的元素 x_r，<root, x_r> ∈ H，且存在 D_r 上的关系 $H_r \subset H$；H = {<root, x_l>, <root, x_r>, H_l, H_r}；

(4) (D_l, {H_l}) 是一棵符合本定义的二叉树，称为根的左子树，(D_r, {H_r}) 是一棵符合本定义的二叉树，称为根的右子树。

基本操作 P：

InitBiTree(&T)；

操作结果：构造空二叉树 T。

DestroyBiTree(&T)；

初始条件：二叉树 T 存在。

操作结果：销毁二叉树 T。

CreateBiTree(&T, definition);

初始条件：definition 给出二叉树 T 的定义。

操作结果：按 definition 构造二叉树 T。

ClearBiTree(&T);

初始条件：二叉树 T 存在。

操作结果：将二叉树 T 清为空树。

BiTreeEmpty(T);

初始条件：二叉树 T 存在。

操作结果：若 T 为空二叉树，则返回 TRUE，否则 FALSE。

BiTreeDepth(T);

初始条件：二叉树 T 存在。

操作结果：返回 T 的深度。

Root(T);

初始条件：二叉树 T 存在。

操作结果：返回 T 的根。

Value(T, e);

初始条件：二叉树 T 存在，e 是 T 中某个结点。

操作结果：返回 e 的值。

Assign(T, &e, value);

初始条件：二叉树 T 存在，e 是 T 中某个结点。

操作结果：结点 e 赋值为 value。

Parent(T, e);

初始条件：二叉树 T 存在，e 是 T 中某个结点。

操作结果：若 e 是 T 的非根结点，则返回它的双亲，否则返回“空”。

LeftChild(T, e);

初始条件：二叉树 T 存在，e 是 T 中某个结点。

操作结果：返回 e 的左孩子。若 e 无左孩子，则返回“空”。

RightChild(T, e);

初始条件：二叉树 T 存在，e 是 T 中某个结点。

操作结果：返回 e 的右孩子。若 e 无右孩子，则返回“空”。

LeftSibling(T, e);

初始条件：二叉树 T 存在，e 是 T 中某个结点。

操作结果：返回 e 的左兄弟。若 e 是 T 的左孩子或无左兄弟，则返回“空”。

RightSibling(T, e);

初始条件：二叉树 T 存在，e 是 T 中某个结点。

操作结果：返回 e 的右兄弟。若 e 是 T 的右孩子或无右兄弟，则返回“空”。

InsertChild(T, p, LR, c);

初始条件：二叉树 T 存在，p 指向 T 中某个结点，LR 为 0 或 1，非空二叉树 c 与 T 不相交且右子树为空。

操作结果：根据 LR 为 0 或 1，插入 c 为 T 中 p 所指结点的左或右子树。p 所指结点的原有左或右子树则成为 c 的右子树。

DeleteChild(T, p, LR);

初始条件：二叉树 T 存在，p 指向 T 中某个结点，LR 为 0 或 1。

操作结果：根据 LR 为 0 或 1，删除 T 中 p 所指结点的左或右子树。

PreOrderTraverse(T, Visit());

初始条件：二叉树 T 存在，Visit 是对结点操作的应用函数。

操作结果：先序遍历 T，对每个结点调用函数 Visit 一次且仅一次。一旦 visit()失败，则操作失败。

InOrderTraverse(T, Visit());

初始条件：二叉树 T 存在，Visit 是对结点操作的应用函数。

操作结果：中序遍历 T，对每个结点调用函数 Visit 一次且仅一次。一旦 visit()失败，则操作失败。

PostOrderTraverse(T, Visit());

初始条件：二叉树 T 存在，Visit 是对结点操作的应用函数。

操作结果：后序遍历 T，对每个结点调用函数 Visit 一次且仅一次。一旦 visit()失败，则操作失败。

LevelOrderTraverse(T, Visit());

初始条件：二叉树 T 存在，Visit 是对结点操作的应用函数。

操作结果：层序遍历 T，对每个结点调用函数 Visit 一次且仅一次。一旦 visit()失败，则操作失败。

}**ADT** BinaryTree

抽象数据型：图

ADT Graph {

数据对象 V:V 是具有相同特性的数据元素的集合,称为顶点集。

数据关系 R:

R = {VR}

VR = {<v,w> | v,w∈V 且 P(v,w),<v,w>表示从 v 到 w 的弧,
谓词 P(v,w)定义了弧<v,w>的意义或信息 }

基本操作 P:

CreateGraph(&G,V,VR);

初始条件:V 是图的顶点集,VR 是图中弧的集合。

操作结果:按 V 和 VR 的定义构造图 G。

DestroyGraph(&G);

初始条件:图 G 存在。

操作结果:销毁图 G。

LocateVex(G, u);

初始条件:图 G 存在,u 和 G 中顶点有相同特征。

操作结果:若 G 中存在顶点 u,则返回该顶点在图中位置;否则返回其他信息。

GetVex(G, v);

初始条件:图 G 存在,v 是 G 中某个顶点。

操作结果:返回 v 的值。

PutVex(&G, v, value);

初始条件:图 G 存在,v 是 G 中某个顶点。

操作结果:对 v 赋值 value。

FirstAdjVex(G, v);

初始条件:图 G 存在,v 是 G 中某个顶点。

操作结果:返回 v 的第一个邻接顶点。若顶点在 G 中没有邻接顶点,则返回“空”。

NextAdjVex(G, v, w);

初始条件:图 G 存在,v 是 G 中某个顶点,w 是 v 的邻接顶点。

操作结果:返回 v 的(相对于 w 的)下一个邻接顶点。若 w 是 v 的最后一个邻接点,则返回“空”。

InsertVex(&G, v);

初始条件:图 G 存在,v 和图中顶点有相同特征。

操作结果:在图 G 中增添新顶点 v。

DeleteVex(&G, v);

初始条件:图 G 存在,v 是 G 中某个顶点。

操作结果:删除 G 中顶点 v 及其相关的弧。

InsertArc(&G, v, w);

初始条件:图 G 存在,v 和 w 是 G 中两个顶点。

操作结果:在 G 中增添弧<v,w>,若 G 是无向的,则还增添对称弧<w,v>。

DeleteArc(&G, v, w);

初始条件:图 G 存在,v 和 w 是 G 中两个顶点。

操作结果:在 G 中删除弧<v,w>,若 G 是无向的,则还删除对称弧<w,v>。

DFSTraverse(G, Visit());

初始条件:图 G 存在,Visit 是顶点的应用函数。

操作结果:对图进行深度优先遍历。在遍历过程中对每个顶点调用函数 Visit 一次且仅一次。一旦 visit()失败,则操作失败。

BFSTraverse(G, Visit());

初始条件:图 G 存在,Visit 是顶点的应用函数。

操作结果:对图进行广度优先遍历。在遍历过程中对每个顶点调用函数 Visit 一次且仅一次。一旦 visit()失败,则操作失败。

}ADT Graph