第4章 树和二叉树

- □ **主要内容：**
- □ 树的基本概念
- □ 二叉树的基本概念和存储结构
- □ 二叉树的遍历

树和二叉树概述

- □ 树是一种非线性结构。
- □ 非线性结构特点是：在结构中至少存在一个数据元素，有两个或两个以上的直接前驱(或直接后继)。
- □ 树中每个数据元素至多有一个直接前驱，但可以有多个直接后继。
- □ 二叉树是一种结构相对简单的树，其直接后继有左右之分，且限制在两个以内。

树是最重要的一种非线性结构。

由于多叉树可以按某种规则转换为二叉树，二叉树在应用中具有非常重要的地位 。

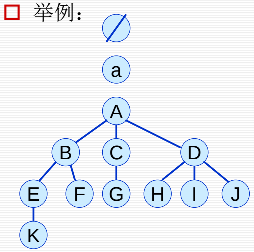
§4.1 树的定义和基本概念

- □ 1. 树(Tree)的定义
- □ 树是n(≥0)结点组成的有限集合，在非空树中：
  - ■ 有且仅有一个没有前驱的结点—根(root)。
  - ■ 当n>1时，其余结点有且仅有一个直接前驱。
  - ■ 所有结点都可以有0个或多个后继。
- □ 树是n(n≥0)个结点的有限集T，T为空时称为空树，否则满足两个条件：
  - ■ 有且仅有一个特定的称为根(Root)的结点。
  - ■ 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，其中每一个集合本身又是一棵树，称为根的子树(subtree)。

树是一个递归定义，即在树的定义中又用到树的概念

树的特点

- □ 非空树中至少有一个结点——根
- □ 树中各子树是互不相交的集合
- □ 举例：

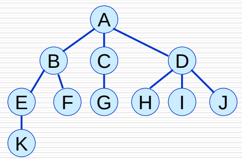


1) n=0; tree=Null

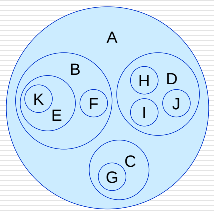
2) n=1; {a}

3) n=11;
T_1= {B, E, F, K}
T_2 ={C,G}
T_3 ={D,H,I,J}
$T_{1\text{-}3}$都是根A的子树，而本身又都是一棵树

2. 树的逻辑表示



- □ 直观表示法：即采用在结点和结点间的连线表示法。
- □ 嵌套集合表示法：将根结点视为一个大的集合，其若干棵子树构成这个大集合中若干个互不相交的子集，不断嵌套下去。

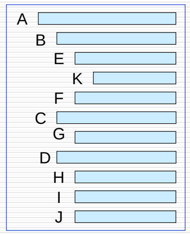


树的逻辑表示(.续)

- □ 广义表表示法：将根作为由子树森林组成的表的名字写在表的左边，然后递归表示

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

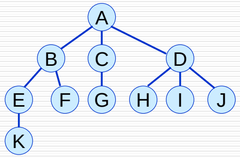
- □ 凹入表示法(目录表示法)：是一种层次表示法，它根据结点所处层次给予不同的长度，主要用于树的屏幕和打印输出



可见：只要分等级的分类方案，都可形成一棵树

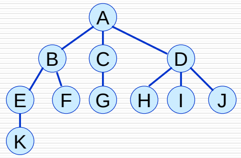
3. 树的基本术语

- 树的结点：即树中的元素，包括一个数据元素及若干指向其子树的分支。
- 结点的度：结点拥有的子树数。
- 树的度(次数)：树内各结点度的最大值。
- 叶子：度为0的结点，又称终端结点。
- 分支结点：度不为0的结点，又称非终端结点。
- 结点层次：从根结点到某结点 j 路径上结点的数目。根为第1层，其余结点的层数等于其双亲结点的层数加1
- 树的深度：树中结点的最大层次。



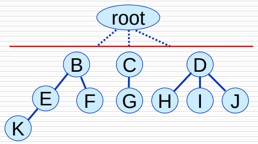
树的基本术语(.续)

- 结点孩子：结点子树的根称为该结点的孩子。
- 孩子双亲：由“结点孩子”定义，该结点称为孩子的双亲
- 兄弟结点：同一个双亲的孩子间互称兄弟。
- 结点祖先：从根到该结点所经分支上的所有结点。
- 子孙：以某结点为根的子树中任意结点称为子孙。
- 堂兄弟：其双亲在同一层的结点间互称堂兄弟。
- 有序树(无序树)：将树中结点的各子树看成从左至右是有序的(不能互换)，否则为无序树



树的基本术语(..续)

- 森林：m[m≥0]棵互不相交的树的集合 $F=(T_1, T_2, ..., T_m)$。把树的根结点删除就可得到一个森林，反之将n棵树的森林加上一个根结点，使n棵树成为该结点的子树，就得到一棵树T=(root,F)



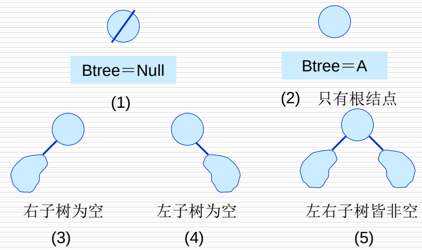
4. 树的抽象数据类型定义

```
ADT  Tree {
   数据对象: D
   数据关系: R
   基本操作:
        InitTree(&T); //初始化        Root(T); //返回根结点
        Parent(T,x); //返回结点的双亲
        Child(T,x,i);  //返回结点x的第i个孩子
        Right_Sibling(T,x);  //返回右兄弟
        CreatTree(&T);  //建树T
        Insertchild(&T,&p,i,c); //把c插入到p的第i棵子树处
        DeleteChild(&T,&p,i); //删除结点p的第i棵子树
        TraverseTree(T,visit( )); //遍历
        ClearTree(&T); //将树T清空
}ADT  Tree
```


§4.2 二叉树

- 1. 二叉树的基本概念
- 二叉树是由n(n≥0)个结点的有限集合构成的有序树。此集合或者为空集，或者由一个根结点及两棵互不相交的左右子树组成，且这两个子树有左右之分，次序不能颠倒，并且都是二叉树。
- 特点(二次、有序)：
  - 每个结点至多有二棵子树(即不存在度大于2的结点)
  - 二叉树的子树有左、右之分，且其次序不能任意颠倒

二叉树的基本形态



画出具有3个结点的树和具有3个结点的二叉树的形态？

二叉树的抽象数据类型定义

```
ADT BinaryTree{
   数据对象：D
   数据关系：R
   基本操作：initial(BT); //初始化
             root(BT); //返回根结点
             parent(BT,x); //返回结点 x 的双亲
             lchild(BT,x); //返回结点 x 的左孩子
             rchild(BT,x); //返回结点 x 的右孩子
             crt_bt(x,LBT,RBT); //生成一棵以 x 为根的树
             ins_lchild(BT,y,x); //把 x作为y的左子树插入
             ins_rchild(BT,y,x); //把 x作为y的右子树插入
             del_lchild(BT,y); //删除结点y的左子树
             del_rchild(BT,y); //删除结点y的右子树
             traverse(BT); //遍历
             clear(BT); //清除
}ADT BinaryTree
```


2. 二叉树的性质

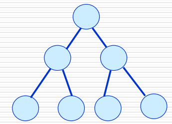
- **性质1**：在二叉树的第i层上至多有2^{i-1}个结点（$i \geq 1$）
- 归纳法证明
  - $i=1$时，只有一个根结点，$2^{i-1}=2^0=1$，命题成立。
  - 假定对所有的j，$1 \leq j < i$，命题成立，即第j层上至多有2^{j-1}个结点，那么可以证明$j=i$时命题也成立

在k叉树中，第*i*层上最多具有多少个结点？

- **性质2**：深度为k的二叉树至多有2^k-1个结点（$k \geq 1$），最少有k个结点。
- 等比求和证明 $\sum_{i=1}^{k}(\text{第}i\text{层上最大结点数}) = \sum_{i=1}^{k} 2^{i-1} = 2^k - 1$

二叉树的性质(.续)

- **性质3**：对任何一棵二叉树，如果其终端叶结点数为n_0，度为2的结点数为n_2，则$n_0=n_2+1$。
- 证明：总结点数＝总分支数(结点总度数)＋1
  - 总的结点数n：$n=n_0+n_1+n_2$
  - 结点总度数B：$B=n_1+2 \times n_2$

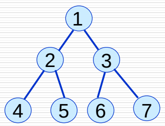


已知一棵度为m的树中有n_1个度为1的结点，n_2个度为2的结点，…n_m个度为m的结点，问该树中有多少片叶子？

$n_0=1+n_2+\ldots+(m-1)n_m$

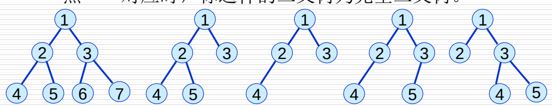
3. 特殊形态的二叉树

- (1) 满二叉树
- **满二叉树**：一棵深度为k且有2^k-1个结点的二叉树称为满二叉树。
  - 每一层上的结点数都是最大结点数。
  - 叶子结点只可能在最后一层。
  - 只有度为0和度为2的结点。
  - 和同样深度下的二叉树比，满二叉树叶子结点个数最多
- 满二叉树的编号：可从根结点开始，自上而下，自左至右对结点进行连续编号



(2) 完全二叉树

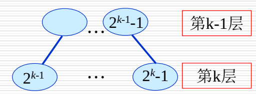
- **完全二叉树**：深度为k，有n个结点的二叉树当且仅当其每一个结点都与深度为k的满二叉树中编号从1至n的结点一一对应时，称这样的二叉树为完全二叉树。



- 特点：k层完全二叉树的前k-1层必须为满二叉树，其第k层上，任意结点的左边不能有空位。
- 叶子结点只可能在层次最大的两层上出现
- 对任一结点，若其右分支下子孙的最大层次为*l*，则其左分支下子孙的最大层次必为*l* 或*l*+1
- n为偶数时有1个度为1的结点，且该结点只有左孩子；n为奇数时没有度为1的结点。

完全二叉树的性质

- **性质4**：具有n个结点的完全二叉树的深度为：$\lfloor \log_2 n \rfloor + 1$，符号$\lfloor x \rfloor$ 表示不大于x的最大整数。
  - 证明：假设此二叉树的深度为k，则根据性质2及完全二叉树的定义得到：



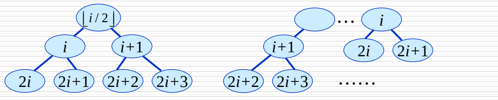
$2^{k-1}-1 < n \leq 2^k-1$ 或 $2^{k-1} \leq n < 2^k$

最少结点数　最多结点数

- 取对数得到：$k-1 \leq \log_2 n < k$

完全二叉树的性质(.续)

- **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(从第1层到第 $\lfloor \log_2 n \rfloor+1$ 层，每层从左到右)，则对任一结点i(1≤i≤n)，有：
  - ① $i=1$，则i是根结点，无双亲；
  - $i>1$，则其双亲parent(i)是结点 $\lfloor i/2 \rfloor$
  - ② $2i>n$，则i为叶子结点，无左孩子；否则lchild(i)=$2i$
  - ③ $2i+1>n$，则i无右孩子；否则rchild(i)=$2i+1$

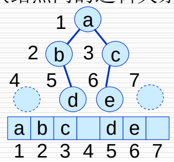


结点i和$i+1$在同一层上　　　结点i和$i+1$不在同一层上

性质5表明，完全二叉树结点的层序编号反映了结点之间的逻辑关系。

4. 二叉树的存储结构

- (1) 顺序存储结构
- 顺序存储结构：用一维数组存储二叉树中的结点，并且结点的存储位置能反映结点间的逻辑关系
  - 分配2^k-1个空间
  - 结点按满二叉树或完全二叉树编号存储
  - 物理位置确定了结点间关系



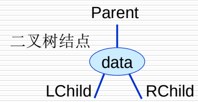
- 存储格式：

```
#define MAX_TREE_SIZE 100
typedef TElemType SqBitree[MAX_TREE_SIZE];
SqBiTree bt;
```

单支树？

(2) 链式存储结构

- 链式存储：令二叉树的每个结点对应一个链表结点，结点除了存放与二叉树结点有关的数据信息外，还要设置指针来指示元素间的逻辑关系



- 结点：两个指针域或三个指针域

lchild	data	rchild	二叉链表的结点

lchild	data	parent	rchild	三叉链表的结点

存储结构类型定义

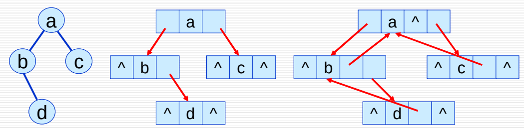
- 二叉链表

```
typedef struct BiTNode{
TelemType  data;          //结点的值
struct BiTNode *lchild, *rchild;  //左右孩子
}BiTNode, *BiTree;
```

- 三叉链表

```
typedef struct BitNode{
   TelemType  data;    //结点的值
   struct BiTNode *lchild, *rchild, *parent;  //左右孩子及双亲
}BiTNode, *BiTree;
```


举例



具有n个结点的二叉链表中，有多少个空指针？

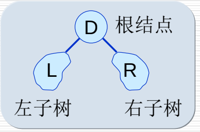
n个结点的二叉链表一定有n+1个空指针域
度为k, n个结点的k叉链表树有n(k-1)+1个空指针域

存储结构总结

- 顺序存储
  - 按完全二叉树分配存储空间，存储一般二叉树时会造成空间浪费，因此一般仅用于存储完全二叉树
- 链式存储
  - 三叉链表便于查找孩子结点和双亲结点；但相对于二叉链表，增加了空间开销。
  - 二叉链表虽然不能直接访问到结点双亲，但由于二叉链表结构灵活，操作方便，对于一般情况的二叉树，甚至比顺序存储结构还节省空间。
- 因此，二叉链表是最常用的二叉树存储方式。

§4.3 二叉树的遍历

- 二叉树的遍历：按一定规律走遍树的各个结点，且使每一结点仅被访问一次。访问可以理解为对结点进行某种处理。
- 遍历操作的结果：使非线性结构线性化



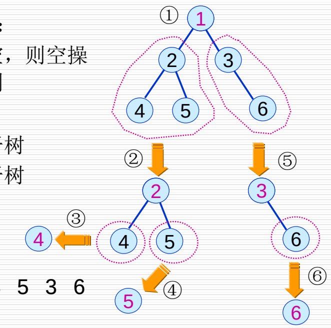
二叉树的基本组成

- 规定对子树的访问先左后右，则方法为：
  - DLR—先(根)序遍历
  - LDR—中(根)序遍历
  - LRD—后(根)序遍历

层序遍历：按二叉树的层序编号的次序访问各结点

先序遍历算法

- 先序遍历算法：
  - 若二叉树为空，则空操作返回；否则
  - 访问根结点
  - 先序遍历左子树
  - 先序遍历右子树



- 先序：1 2 4 5 3 6

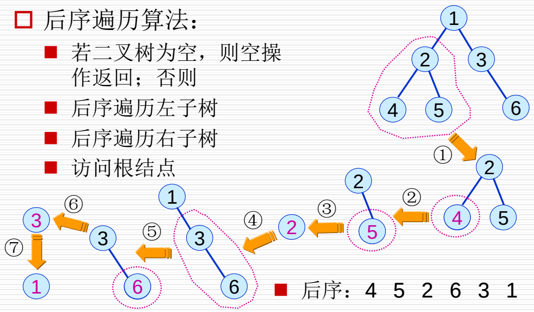
中序遍历算法

- 中序遍历算法：
  - 若二叉树为空，则空操作返回；否则
  - 中序遍历左子树
  - 访问根结点
  - 中序遍历右子树

- 中序：4 2 5 1 3 6

后序遍历算法

- 后序遍历算法：
  - 若二叉树为空，则空操作返回；否则
  - 后序遍历左子树
  - 后序遍历右子树
  - 访问根结点



- 后序：4 5 2 6 3 1

遍历算法的递归程序

先序遍历

```
void  preorder(bt)
{  if (bt)
    {  visit(bt->data);        //访问结点
       preorder(bt->lchild);   //先序访问左子树
       preorder(bt->rchild); } } //先序访问右子树
```

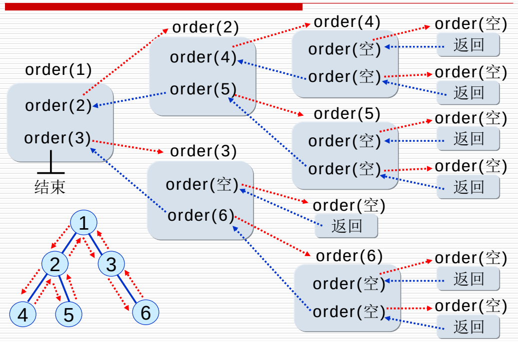
中序遍历

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

后序遍历

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


调用过程



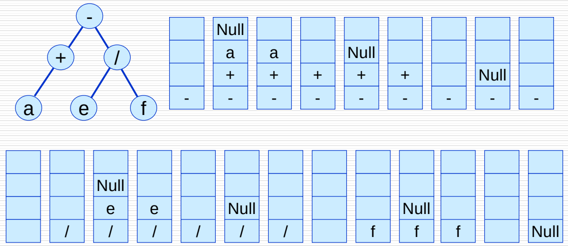
递归遍历算法总结

- 先序、中序和后序遍历都是从根结点开始的，且在遍历过程中经过结点的路线是一样的，只是访问的时机不同而已。
- 遍历路线
  - 从根结点开始沿左子树往下走
  - 当到达最左端，无法再往下时，则返回
  - 再逐一进入上一层遇到的结点的右子树，再进行往下走和返回
  - 直到最后从根结点的右子树返回到根结点为止。
- 先序遍历是在深入时遇到结点就访问
- 中序遍历是在从左子树返回时遇到结点访问
- 后序遍历是在从右子树返回时遇到结点访问

非递归算法

- 遍历过程中，遇见结点的顺序与返回的顺序相反，即后遇见先返回，正好符合栈结构后进先出的特点，可以利用栈实现非递归算法。
- 遍历过程如下：
  - 沿左子树往下走，每遇见一个结点入栈一个结点，若为先序遍历，则在入栈之前访问之
  - 当沿左分支走不下去时，则返回，即从堆栈中弹出前面压入的结点。若为中序遍历，则此时访问该结点，然后从该结点的右子树继续往下走
  - 若为后序遍历，则将此结点再次入栈，然后从该结点的右子树继续往下走，与前面类似，仍为遇见一个结点入栈一个结点，往下走不下去再返回，直到第二次从栈里弹出该结点，才访问之。

举例



- 其先序遍历为：－＋a/ef；中序遍历为：a＋－e/f

非递归先序遍历程序

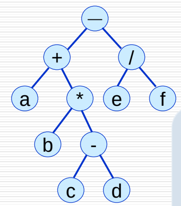
```
void  preorder(bt)
{  initstack(s);  push(s,bt);  //初始化栈
   if (!bt)  visit(bt);     //遍历根结点
   while(!empty(s))
   {  while(gettop(s,p)&&p) //取出栈顶元素，且该元素不为空
      {  push(s,p->lchild); //向左走到尽头
         if (!p->lchild)  visit(p->lchild );  }
      pop(s,p);    //去掉栈顶空指针
      if (!empty(s))
      {  pop(s,p); //回到上层的非空结点
         push(s,p->rchild);  //右子树进栈
         if (!p->rchild) visit(p->rchild );  }
   } }
```


非递归中序遍历程序

```
void inorder(bt)
{  initstack(s);  push(s,bt);       //初始化栈
   while (!empty(s))
   {  while(gettop(s,p)&&p)
         push (s,p->lchild);  //寻找最左下端的结点
      pop(s,p);              //去掉栈顶空指针
      if (!empty(s))
      {  pop(s,p);
         visit(p->data);       //访问结点
         push(s,p->rchild);  }  //右子树进栈
} }
```


二叉树遍历练习

- 二叉树表达式(a＋b*(c-d)-e/f)
  - 先序：－＋a*b－cd/ef
  - 中序：a＋b*c－d－e/f
  - 后序：abcd－*＋ef/－



先序访问，根结点总在最前面
后序访问，根结点总在最后面
中序访问，第1访问结点为最左边第1个无左子树的结点，最后一个访问结点为最右边第1个无右子树的结点

注意：遍历是二叉树各种操作的基础

二叉链表的建立

- 二叉树存储结构的建立可通过遍历实现，一般以先序遍历方式输入。在输入时，可约定一种结点来表示该结点无左孩子或无右孩子，例如空格。将约定的结点数据称为虚结点数据。

```
status CreateBinTree(BiTree  &T)
{   scanf(&ch);
    if(ch==' ')  T=NULL;
    else {
        if(!(T=(BiTNode *)malloc(sizeof(BiTNode))) exit(0);
        T->data=ch;
        CreateBinTree((BiTree *)&(T->Lchild));
        CreateBinTree((BiTree *)&(T->Rchild));    }
    return(OK);     }
```