IPL 电子信息学院 武汉大学 Wuhan University

数据结构与算法 (C#语言版)

DATA STRUCTURE & ALGORITHM IN C#

第6章 树与二叉树

王文伟 Wang Wenwei, Dr.-Ing.
Tel: 189-71562600
Email: wwwang@aliyun.com
Web: http://ipl.whu.edu.cn/sites/ced/st/

电子信息学院 Table of Contents 武汉大学 Wuhan University

本章位置

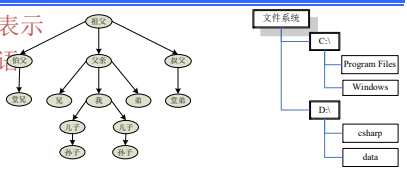
本章介绍具有层次关系的非线性数据结构—树和二叉树的概念和定义，重点讨论二叉树的性质、存储结构和遍历算法，并介绍线索二叉树的定义、存储结构和遍历算法。

电子信息学院 Table of Contents 武汉大学 Wuhan University

6.0 Introduction

- 树结构是一种数据元素之间具有层次关系的非线性结构。除根结点外，每个元素只有一个前驱元素，但可以有零个或若干个后继元素。
- 树结构有树和二叉树两种。二叉树是最多只有两个子树且两个子树有左右之分的有序树。
- 本章介绍具有层次关系的树结构，重点讨论二叉树的定义、性质、存储结构和遍历算法，并讨论线索二叉树的定义、存储结构和遍历算法。

6.1 树的定义与基本术语

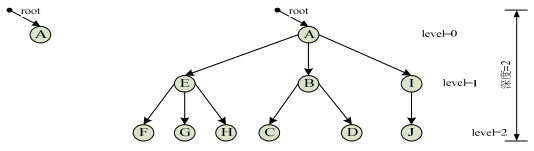


(a) 家谱树 (b) 文件系统

- 生活中的很多对象之间具有层次关系，如家族成员、机构管理部门、计算机文件系统等。这种层次关系类似于自然界中的树。
- 树结构从自然界中的树抽象而来，树的（树）根、枝杈和叶子分别对应于层次结构的起源、分支和终点。
- Windows等操作系统的文件系统是一个树型结构的数据结构，根目录是文件（目录）树的根节点，子目录（文件夹）是树中的分支节点，文件是树的叶子结点。

6.1.1 树的定义和表示

- 树(tree)是由n个结点组成的有限集合，它或者是棵空树，或者包含一个根结点和零或若干棵互不相交的子树。
- n=0的树称为空树；对n>0的**树T**有以下特点：
  - 树T有一个特殊的结点，它没有前驱结点，这个结点称为根结点（root结点）。
  - 当n>1时，除根结点之外的其他结点分为m个互不相交的集合T_1, T_2, …, T_m，其中每个集合T_i本身又是一棵树，称作子树(subtree)。



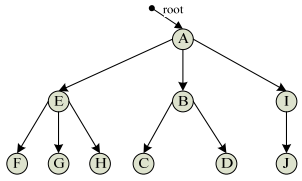
(a) n=1，仅有根结点的树 (b) n=10，深度为2的树

树可以分为无序树与有序树

- 在无序树（unorderd tree）中，结点的子树T_1，T_2，…之间没有次序。通常所说的树指的是无序树。
- 如果树中结点的子树T_1，T_2，…从左至右是有次序的，则称该树为有序树（orderd tree）。

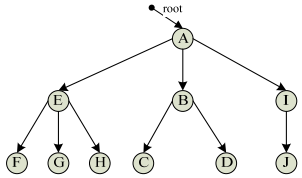
树与森林

- 若干棵互不相交的树的集合称为森林（forest）。
  - 给森林加上一个根结点就变成一棵树；
  - 将树的根结点删除就变成由子树组成的森林。



树的广义表形式表示

- 可以用广义表的形式表示树结构。例，如图所示树的广义表表示形式为：
  A（B（C，D），E（F，G，H），I（J））
- 树中的叶结点对应广义表中的原子，非叶结点对应子表。
- 树结构的广义表是一种纯表，其中没有共享和递归成分。



6.1.2 树的基本术语

- node: 结点表示树集合中的一个数据元素，它一般由元素的数据和指向它的子结点的指针构成。
- child node与parent node: 若结点*N*有子树，则子树的根结点称为结点*N*的子结点。结点*N*称为其孩子的父结点。父子结点相连接。
- sibling node: 同一双亲的子结点之间互称兄弟结点。
- ancestor node与descendant node: 结点的祖先是指从根到该结点所经过的所有结点。后代是指结点的所有子结点，以及各子结点的子结点。

树的基本术语(II)

- degree of node&tree ：结点的度定义为结点所拥有子树的棵数。树的度是指树中各结点度的最大值。
- leaf node与branch node: 叶子结点是指度为0的结点，又称为终端结点。除叶子结点以外的其他结点，称为分支结点或非叶子结点，又称为非终端结点。
- edge: 设树中*M* 结点是*N* 结点的父结点，有序对<*M*, *N*>称为连接这两个结点的边。

树的基本术语(II)

- path与path length: 如果（N_1，N_2，…，N_k）是由树中结点组成的一个序列，且<N_i，N_{i+1}>都是树的边，则该序列称为从N_1到N_k的一条路径。路径上边的数目称为该路径长度。
- level: 令根结点的层次为0，其余结点的层次等于它双亲结点的层次加1。显然，兄弟结点的层次相同。结点N的层次与从根到该结点的路径长度有关。
- depth: 树中结点的最大层次数称为树的深度或高度。

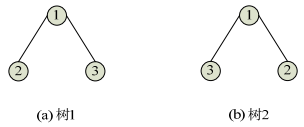
6.2 二叉树的定义与实现

- 树结构包括无序树和有序树两种类型
- 有序树中最常用的是二叉树

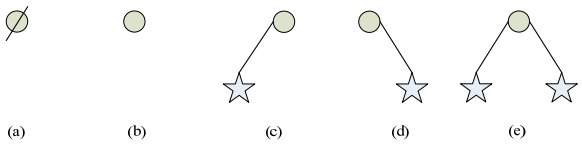
6.2.1 二叉树的定义

- 二叉树(binary tree)的递归定义：二叉树是n个结点组成的有限集合。n=0时称为空二叉树；n>0时，二叉树由一个根结点和两棵互不相交的、分别称为左子树和右子树的子二叉树构成。
- 二叉树是一种有序树，每个结点的两棵子树有左、右之分，即使只有一个子树，也要区分是左子树还是右子树。
- 二叉树的结点最多只有两棵子树，所以二叉树的度最多为2。

不同的二叉树与相同的度为2的树



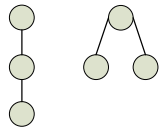
二叉树的五种基本形态



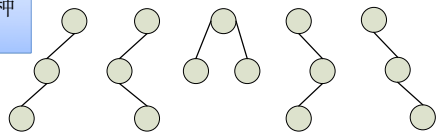
（a）为空的二叉树。
（b）为只有一个结点（根结点）的二叉树。
（c）为由根结点，非空的左子树和空的右子树组成的二叉树。
（d）为由根结点，空的左子树和非空的右子树组成的二叉树。
（e）为由根结点，非空的左子树和非空的右子树组成的二叉树。

【例6.1】画出3个结点的树与二叉树的基本形态

(a) 3个结点的树的2种基本形态



(b) 3个结点的二叉树的5种基本形态



6.2.2 二叉树的性质

- 性质一：二叉树第i层的结点数目最多为2^i ($i \geq 0$)
  用归纳法证明。
  - 归纳基础：根是i=0层上的唯一结点，故2^i=1，命题成立。
  - 归纳假设：对所有j($0 \leq j < i$)，j层上的最大结点数为2^j。
  - 归纳步骤：根据归纳假设，第i-1层上的最大结点数为2^{i-1}；由于二叉树中每个结点的度最大为2，故第i层上的最大结点数为$2 \times 2^{i-1}=2^i$。命题成立。
- 性质二：在深度为k的二叉树中，至多有$2^{k+1}-1$个结点

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

每一层的结点数目都达到最大值的二叉树称为满二叉树

二叉树的性质(II)

- 性质三：二叉树中，若叶子结点数为n_0，2度结点数为n_2，则有$n_0 = n_2+1$。

证明： 设二叉树结点数为n，1度结点数为n_1，则有：$n = n_0 + n_1 + n_2$

度为1的结点有1个子女，度为2的结点有2个子女，叶子结点没有子女，根结点不是任何结点的子女，从子结点数的角度看，有

$$n - 1 = 0 \times n_0 + 1 \times n_1 + 2 \times n_2$$

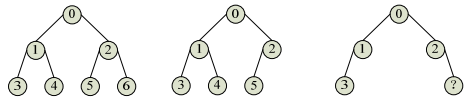
综合上述两式，可得$n_0=n_2+1$，即二叉树中叶子结点数比度为2的结点数多1。

二叉树的性质(III)

- 性质四：如果一棵完全二叉树有n个结点，则其深度
$$k=\lfloor \log_2 n \rfloor$$
  - 深度为k的满二叉树 (full binary tree) 具有2^{k+1}-1个结点。结点数目都达到最大值。
  - 对二叉树的结点进行连续编号，约定编号从根结点开始，自上而下，每层自左至右。
  - 具有n个结点深度为k的二叉树，如果它的每个结点都与深度为k的满二叉树中编号为0~n-1的结点一一对应，则称为完全二叉树 (complete binary tree)。

满二叉树与完全二叉树

- 满二叉树一定是完全二叉树，而完全二叉树不一定是满二叉树，它是具有满二叉树结构而不一定满的二叉树。
- 完全二叉树只有最下面一层可以不满，其上各层都可看成满二叉树。
- 完全二叉树最下面一层的结点都集中在该层最左边的若干位置上。
- 完全二叉树至多只有最下面两层结点的度可以小于2。



(a) 满二叉树 (b) 完全二叉树 (c) 非完全二叉树

二叉树的性质(续)

- 性质五：若将一棵具有n个结点的完全二叉树按顺序表示，编号为i的结点，有如下规律：
  - 若i=0，则结点i为根结点；若i≠0，则结点i的双亲是编号为(i-1) /2(取整)的结点。
  - 若2i+1≤n-1，则i的左孩子是编号为2i+1的结点；若2i+1>n-1，则i无左孩子。
  - 若2i+2≤n-1，则i的右孩子是编号为2i+2的结点；若2i+2＞n-1，则i无右孩子。

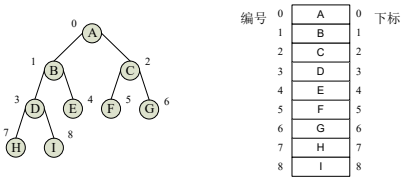
6.2.3 二叉树的存储结构

1. 二叉树的顺序存储结构
2. 二叉树的链式存储结构

- 二叉树结构是一种具有层次关系的数据结构，用链式存储结构实现二叉树更加灵活方便，所以一般情况下，采用链式存储结构来存储二叉树。
- 顺序存储结构适用于完全二叉树。

1. 二叉树的顺序存储结构

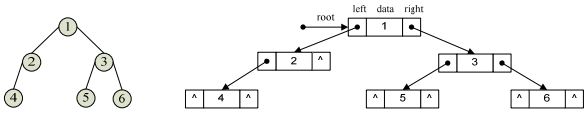
- 顺序存储结构适用于完全二叉树，对完全二叉树进行顺序编号，将编号为 i 的结点存放在数组下标为 i 的位置上。根据二叉树的性质五，对于结点i，可以直接计算得到其父结点、左子结点和右子结点的位置。



(a) 完全二叉树 (b) 顺序存储完全二叉树

2. 二叉树的链式存储结构

- 结点结构：每个结点有3个域：
  - 数据域data，表示结点的数据元素；
  - 左链域left，指向该结点的左子结点；
  - 右链域right，指向该结点的右子结点。
- 二叉树结构：需记录其根结点root，若二叉树为空，则root==null。
- 树中某结点的左子结点也代表该结点的左子树，同理，该结点的右子结点代表它的右子树。



(a) 二叉树 (b) 链式存储结构

6.2.4 二叉树类的定义

二叉树的结点类

```
public class BinaryTreeNode<T> {
  private T data;   //数据元素
  private BinaryTreeNode<T> left, right;
  //指向左、右孩子结点的链
  public BinaryTreeNode(){
    left = right = null;
  }
  public BinaryTreeNode(T d) {//构造有值结点
    data = d;  left = right = null;
  }
  …… }
```


二叉树结点类的公有属性

```
public T Data {
  get { return data; }
  set { data = value; }
}
public BinaryTreeNode<T> Left {
  get { return left; }
  set { left = value; }
}
public BinaryTreeNode<T> Right {
  get { return right; }
  set { right = value;}
}
```


二叉树类

```
public class BinaryTree<T> {
  protected BinaryTreeNode<T> root;
          //指向二叉树的根结点
  public BinaryTree(){//构造空二叉树
    root = null;
  }
  ……
}
```

- BinaryTree类表示链式存储结构的二叉树，成员变量root指向二叉树的根结点。

6.3. 二叉树的遍历

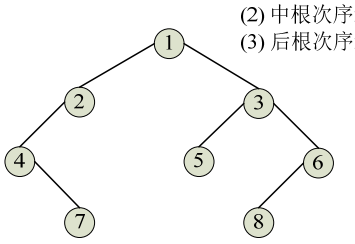
遍历二叉树就是按照一定规则和次序访问二叉树中的所有结点，并且每个结点仅被访问一次。

6.3.1 二叉树遍历的概念

- traversal: 遍历二叉树就是按照一定次序访问二叉树中的所有结点，并且每个结点仅被访问一次。例：按层次高低次序遍历二叉树。
- 二叉树是由根结点、左子树和右子树三个部分组成的，依次遍历这三个部分，便是遍历整个二叉树。。若规定对子树的访问按“先左后右”的次序进行，则遍历二叉树有3种次序：
  - 先根次序：访问根结点，遍历左子树，遍历右子树。
  - 中根次序：遍历左子树，访问根结点，遍历右子树。
  - 后根次序：遍历左子树，遍历右子树，访问根结点。

遍历二叉树的3种次序

(1) 先根次序遍历序列：1 2 4 7 3 5 6 8
(2) 中根次序遍历序列：4 7 2 1 5 3 8 6
(3) 后根次序遍历序列：7 4 2 5 8 6 3 1



- 先根次序或后根次序反映双亲与孩子结点的层次关系，中根次序反映兄弟结点间的左右次序。

先根次序遍历二叉树的过程

若二叉树为空，则该操作为空操作，返回；否则从根结点开始，

1. 访问当前结点；
2. 若当前结点的左子树不空，则沿着left链进入该结点的左子树进行遍历。
3. 若当前结点的右子树不空，则沿着right链进入该结点的右子树进行遍历。

6.3.2 二叉树遍历的递归算法

- 按先根次序遍历二叉树的递归算法

  若二叉树为空，则空操作，返回；否则从根结点开始，

  1. 访问当前结点。
  2. 按先根次序遍历当前结点的左子树。
  3. 按先根次序遍历当前结点的右子树。

先根次序遍历二叉树的递归算法

BinaryTreeNode

```
public void ShowPreOrder() {
  Console.Write(this.Data + " ");
  BinaryTreeNode<T> q = this.Left;
  if (q != null) q.ShowPreOrder();
  q = this.Right;
  if (q != null) q.ShowPreOrder();
}
```

数组或线性表

```
public void TraversalPreOrder(IList<T> sql) {
  sql.Add(this.Data);
  BinaryTreeNode<T> q = this.Left;
  if (q != null)q.TraversalPreOrder(sql);
  q = this.Right;
  if (q != null)q.TraversalPreOrder(sql); }
```


从根结点开始先根次序遍历二叉树

BinaryTree

```
public void ShowPreOrder() {
    Console.Write("先根次序: ");
    if (root != null)
        root.ShowPreOrder();
    Console.WriteLine();
}
```

```
public List<T> TraversalPreOrder() {
    List<T> sql = new List<T>();
    if (root != null)
        root.TraversalPreOrder(sql);
    return sql; }
```


中根次序遍历二叉树的递归算法

BinaryTreeNode

```
public void ShowInOrder() {
  BinaryTreeNode<T> q = this.Left;
  if (q != null) q.ShowInOrder();
  Console.Write(this.Data + " ");
  q = this.Right;
  if (q != null) q.ShowInOrder(); }
```

```
public void TraversalInOrder(IList<T> sql) {
  BinaryTreeNode<T> q = this.Left;
  if (q != null)q.TraversalInOrder(sql);
  sql.Add(this.Data);
  q = this.Right;
  if (q != null)q.TraversalInOrder(sql);
}
```


从根结点开始中根次序遍历二叉树

BinaryTree

```
public void ShowInOrder() {
    Console.Write("中根次序: ");
    if (root != null)
        root.ShowInOrder();
    Console.WriteLine();
}
```

```
public List<T> TraversalInOrder() {
    List<T> sql = new List<T>();
    if (root != null)
        root.TraversalInOrder(sql);
    return sql;
}
```

后根次序遍历二叉树的递归算法

BinaryTreeNode

```
public void ShowPostOrder() {
  BinaryTreeNode<T> q = this.Left;
  if (q != null) q.ShowPostOrder();
  q = this.Right;
  if (q != null) q.ShowPostOrder();
  Console.Write(this.Data + " "); }
```

```
public void TraversalPostOrder(IList<T> sql) {
  BinaryTreeNode<T> q = this.Left;
  if (q != null) q.TraversalPostOrder(sql);
  q = this.Right;
  if (q != null) q.TraversalPostOrder(sql);
  sql.Add(this.Data);   }
```


从根结点开始后根次序遍历二叉树

BinaryTree

```
public void ShowPostOrder() {
    Console.Write("后根次序： ");
    if (root != null)
        root.ShowPostOrder();
    Console.WriteLine();
}
```

```
public List<T> TraversalPostOrder() {
    List<T> sql = new List<T>();
    if (root != null)root.TraversalPostOrder(sql);
    return sql;
}
```


【例6.2】按先根、中根和后根次序遍历二叉树

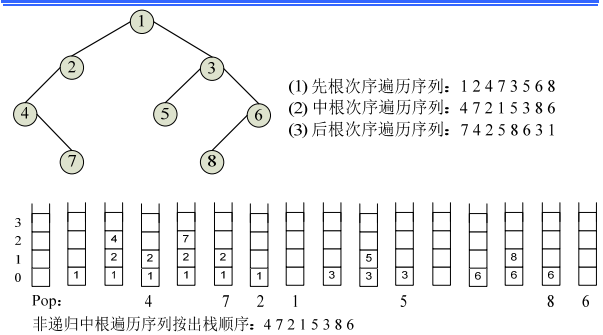
```
BinaryTree<int> btree = new BinaryTree<int>();
BinaryTreeNode<int>[] nodes= new BinaryTreeNode<int>[9];
for(int i=1;i<=8;i++)nodes[i]=new BinaryTreeNode<int>(i);
btree.Root = nodes[1];
nodes[1].Left = nodes[2]; nodes[1].Right = nodes[3];
nodes[2].Left = nodes[4];
nodes[3].Left = nodes[5]; nodes[3].Right = nodes[6];
nodes[4].Right = nodes[7];
nodes[6].Left = nodes[8];btree.ShowPreOrder();
btree.ShowInOrder(); btree.ShowPostOrder();

BinaryTree<string> btree2 = new BinaryTree<string>();
btree2.Root = new BinaryTreeNode<string>("大学");
btree2.Root.Left = new BinaryTreeNode<string>("学院1");
```

递归方式的思路直接清晰，但是算法的空间复杂度和时间复杂度，相比非递归方式增加了许多。

6.3.3 二叉树遍历的非递归算法

- 中根次序遍历: 在每个结点处，先选择遍历左子树，其后必须返回该结点，访问该结点后，再遍历右子树。
- 设置一个栈s来记录经过的路径。从二叉树的根结点p开始，如果p不空或栈不空时，循环执行以下操作，直到扫描完二叉树且栈为空。
  1. 如果p不空，表示扫描到一个结点，将p结点入栈（s.Push(p)），进入其左子树。
  2. 如果p为空并且栈不空，表示已走过一条路径，必须返回寻找另一条路径。设置p指向出栈的结点（p=s.Pop()），访问p结点，再进入p的右子树。

二叉树的非递归中根遍历过程



非递归中根遍历序列按出栈顺序：4 7 2 1 5 3 8 6

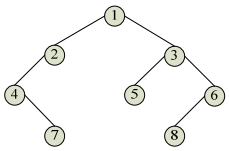
定义在二叉树类中

```
public void ShowInOrderNR() {
  Stack<BinaryTreeNode<T>> s =
      new Stack<BinaryTreeNode<T>>(100);
  BinaryTreeNode<T> p = root;
  Console.Write("非递归中根次序： ");
  while(p!= null||s.Count!= 0){//p非空或栈非空时
    if (p != null) {
      s.Push(p);  //p结点入栈
      p = p.Left; //进入左子树
    } else {    //p为空且栈非空时
      p = s.Pop();//p指向出栈的结点
      Console.Write(p.Data + " ");//访问结点
      p = p.Right;                //进入右子树
    }   }
  Console.WriteLine();  }
```

6.3.4 按层次遍历二叉树

◆按层次遍历二叉树: 从根结点开始，逐层深入，同层从左至右依次访问结点。

图示的二叉树，其层次遍历序列为1，2，3，4，5，6，7，8。首先访问根结点1，再访问根结点的孩子2和3，然后应该访问2的孩子4，再访问3的孩子5结点……必须设立辅助的“先进先出”数据结构，用来指示下一个要访问的结点。

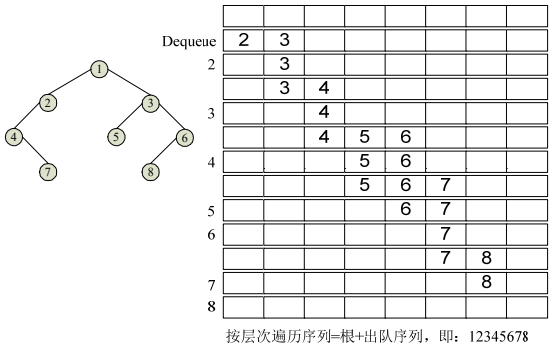


按层次遍历二叉树的非递归算法

◆ 设置一个队列q。结点变量p从根开始，当p不空时，循环顺序执行以下操作：

1. 访问p结点。
2. 如果p的left链不空，将p结点的左孩子加入队列q（入队操作q.Enqueue(p.Left)）。
3. 如果p的right链不空，将p结点的右孩子加入队列q（入队操作q.Enqueue(p.Right)）。
4. 如果队列q为非空，设置p指向队列q出队的结点（p=Dequeue()），否则设置p指向null。

按层次遍历二叉树时队列内容的变化



按层次遍历序列=根+出队序列，即：12345678

定义在二叉树类中

```
public void ShowByLevel() {
  var q = new Queue<BinaryTreeNode<T>>(100);
  //设立一个空队列
  BinaryTreeNode<T> p = root;
  Console.Write("层次遍历: ");
  while (p != null) {
    Console.Write(p.Data + " ");
    if (p.Left != null)
      q.Enqueue(p.Left); //p的左孩子结点入队
    if (p.Right != null)
      q.Enqueue(p.Right); //p的右孩子结点入队
    if (q.Count != 0)
       p = q.Dequeue();//p指向出队的结点
    else
       p = null;
  }
  Console.WriteLine(); }
```

6.3.5 建立二叉树

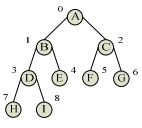
- 给定一定的条件，可唯一建立二叉树。例如对于完全二叉树，如果各结点值已按顺序存储，即可唯一建立一颗二叉树。
- 一般情况下，建立一棵二叉树必须明确以下两点：
  - 结点与其父结点及子结点间的层次关系。
  - 兄弟结点间的左右顺序关系。
- 广义表形式有时不能唯一表示二叉树，需用特殊的广义表才能唯一描述，例如在二叉树的广义表表示式中清楚地标明空子树，给定这种形式的广义表表示式，可以唯一地建立一颗二叉树。
- 对于给定的一棵二叉树，遍历产生的先根、中根、后根序列是唯一的；反之，仅知一种遍历序列，并不能唯一确定一棵二叉树。先根次序或后根次序反映双亲与孩子结点的层次关系，中根次序反映兄弟结点间的左右次序。所以，已知先根和中根两种遍历序列，或中根和后根两种遍历序列才能够唯一确定一棵二叉树。

建立二叉树举例

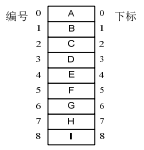
1. 建立链式存储结构的完全二叉树
2. 以广义表表示式建立二叉树
3. 按先根和中根次序遍历序列建立二叉树

1. 建立链式存储结构的完全二叉树

- 对于一棵已经顺序存储的完全二叉树，由二叉树的性质五可知，第0个结点为根结点，第i个结点的左孩子为第$2i+1$个结点，右孩子为第$2i+2$个结点。
- 在二叉树类BinaryTree<T>中，增加静态方法ByOneList，它的参数t是一个线性表或数组，用以表示顺序存储的完全二叉树结点值的序列。

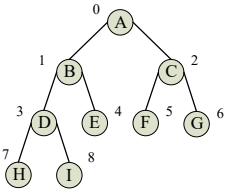


(a) 完全二叉树



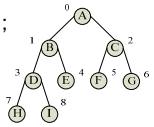
(b) 顺序存储完全二叉树


```
public static BinaryTree<T> ByOneList(IList<T> t) {
  int n = t.Count;
  BinaryTree<T> bt = new BinaryTree<T>();
  if(n==0){bt.Root=null; return bt;}
  int i, j; T v;
  BinaryTreeNode<T>[] q = new BinaryTreeNode<T>[n];
  for (i = 0; i < n; i++) {
    v = t[i]; //取编号为i的结点值
    q[i] = new BinaryTreeNode<T>(v);   }
  for (i = 0; i < n; i++) {
    j = 2 * i + 1;
    if (j < n) q[i].Left = q[j];
    else q[i].Left = null;
    j++;
    if (j < n) q[i].Right = q[j];
    else q[i].Right = null; }
  bt.Root = q[0]; return bt; }
```



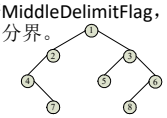
【例6.3】根据给定数组建立完全二叉树

```
static void Main(string[] args) {
  int[] it = { 0,1,2,3,4,5,6,7};
  BinaryTree<int> btree =
      BinaryTree<int>.ByOneList(it);
  btree.ShowPreOrder();
  btree.ShowInOrder();

  char[] ct = { 'A', 'B', 'C',
      'D', 'E', 'F', 'G', 'H' };
  BinaryTree<char> btree2 =
      BinaryTree<char>.ByOneList(ct);
}
```



2.以广义表表示式建立二叉树

- 广义表形式有时不能唯一表示一棵二叉树，原因在于无法明确左右子树。例如，广义表A(B)没有表达出结点B是结点A的左子结点还是右子结点。为了唯一表示一棵二叉树，必须重新定义广义表的形式。
- 在广义表中，除数据元素外还定义四个边界符号：
  1. 空子树符NullSubtreeFlag，如 '^'，以标明非叶子结点的空子树。
  2. 左界符LeftDelimitFlag，如 '('，以标明下一层次的左边界；
  3. 右界符RightDelimitFlag，如 ')'，以标明下一层次的右边界。
  4. 中界符MiddleDelimitFlag，如 ','，以标明某一层次的左右子树的分界。



1(2(4(^,7),^),3(5,6(8,^)))

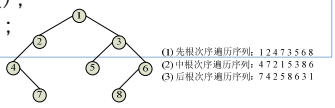
以广义表表示式建立二叉树的算法

- 依次读取二叉树的广义表表示序列中的每个元素，
  - ➢如果遇到有效数据值，建立一个二叉树结点对象；移到下一元素，
    - 如果它为LeftDelimitFlag，则LeftDelimitFlag和RightDelimitFlag之间是该结点的左子树与右子树，再递归调用，分别建立左、右子树，返回结点对象。
    - 如果没有遇到LeftDelimitFlag，表示该结点是叶子结点。
  - ➢如果遇到NullSubtreeFlag，表示空子树，返回null值。
- 在二叉树类BinaryTree<T>中，增加静态方法ByOneList，它的第一个参数表示顺序存储的广义表表示式，第二个参数定义广义表表示式所用的分界符。


```
public static BinaryTree<T> ByOneList(
  IList<T> sList,ListFlagsStruc<T> ListFlags){
  BinaryTree<T>.ListFlags = ListFlags;
  BinaryTree<T>.idx = 0; //初始化递归变量
  BinaryTree<T> bt = new BinaryTree<T>();
  if (sList.Count > 0)
    bt.Root = RootByOneList(sList);
  else
    bt.Root = null;
  return bt;
}
```

```
private static BinaryTreeNode<T> RootByOneList(IList<T> sList) {
  BinaryTreeNode<T> p = null;
  T nodeData = sList[idx];
  if (isData(nodeData)) {
    p = new BinaryTreeNode<T>(nodeData);//有效数据，建立结点
    idx++; nodeData = sList[idx];
    if (nodeData.Equals(ListFlags.LeftDelimitFlag)){
      idx++;  //左边界，如'('，跳过
      p.Left = RootByOneList(sList);//建立左子树，递归
      idx++;  //跳过中界符，如','
      p.Right = RootByOneList(sList);//建立右子树，递归
      idx++;      //跳过右边界，如')'
    } }
  if(nodeData.Equals(ListFlags.NullSubtreeFlag))
      idx++;//空子树符，跳过，返回null
  return p;      }
```

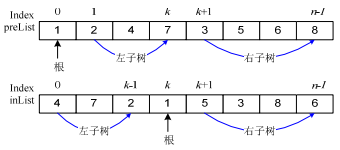
```
static void Main(string[] args) {
  string s = "1(2(4(^,7),^),3(5,6(8,^)))";
  Console.WriteLine("Generalized List: " + s);
  ListFlagsStruc<char> ListFlags;
  ListFlags.NullSubtreeFlag = '^';
  ListFlags.LeftDelimitFlag = '(';
  ListFlags.RightDelimitFlag = ')';
  ListFlags.MiddleDelimitFlag = ',';
  BinaryTree<char> btree = BinaryTree<char>.
   ByOneList(s.ToCharArray(0,s.Length), ListFlags);
  btree.ShowPreOrder();
  btree.ShowInOrder();
    }
```



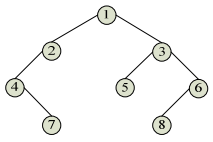
3.按先根和中根次序遍历序列建立二叉树

已知二叉树的一种遍历序列，并不能唯一确定一棵二叉树。如果已知二叉树的先根和中根两种遍历序列，或中根和后根两种遍历序列，则可唯一地确定一棵二叉树。

设二叉树的先根及中根次序遍历序列分别为线性表或数组preList和inList 。



(a) 先根与中根遍历序列



(b) 所建立的二叉树

按先根和中根次序遍历序列建立二叉树算法

- 设二叉树的先根及中根遍历序列分别为preList和inList 。
  1. 确定根元素。由先根次序知，二叉树的根为preList[0]。查找它在inList中的位置k = inList.IndexOf(rootData);
  2. 确定根的左子树的相关序列。由中根次序知， inList[k]之前的结点在根的左子树上， inList[k]之后的结点在根的右子树上。因此根的左子树由k个结点组成：
     - 先根序列——preList[1]，…，preList [k]。
     - 中根序列——inList[0]，…，inList [k-1]。
  3. 根据左子树的先根序列和中根序列建立左子树，递归。
  4. 确定根的右子树的相关序列。右子树由n-k-1个结点组成：
     - 先根序列——preList [k+1]，…，preList [n-1]。
     - 中根序列——inList [k+1]，…，inList [n-1]。
  5. 根据右子树的先根序列和中根序列建立右子树，递归。

【例6.5】按先根和中根次序遍历序列建立二叉树

```
using System; using DSAGL;
namespace treetest {
  class ByTwoListTest {
    static void Main(string[] args) {
      int[] prelist = { 1, 2, 4, 7, 3, 5, 6, 8};
      int[] inlist = {4, 7, 2, 1, 5, 3, 8, 6};
      BinaryTree<int> btree = BinaryTree<int>.
        ByTwoList(prelist, inlist);
      btree.ShowPreOrder();
      btree.ShowInOrder();
  }  } }
```


6.4 线索二叉树

- 从上节的讨论可知：二叉树的遍历将得到一个二叉树结点集合按一定规则排列的线性序列，在这个遍历序列中，除第一个和最后一个结点外，每个结点有且只有一个前驱结点和一个后继结点。
- 在上节定义的二叉树的链式存储结构中，每个结点很容易到达其左、右子结点，而不能直接到达该结点在任意一个遍历序列中的前驱或后继结点，这种信息只能在遍历的动态过程中才能得到。下面介绍的线索树结构，在不增加很多存储空间的条件下，能够存储遍历过程中得到的信息，并在后续的使用中解决直接访问前驱结点和后继结点的问题。

6.4.1 线索与线索二叉树

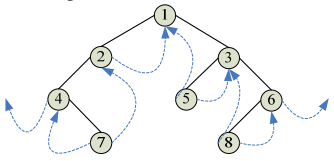
- *n*个结点的二叉树，总共有2*n*个链，仅需要*n*-1个链来指明各结点间的关系，其余*n*+1个链均为空值。利用空链作线索来指明结点在某种遍历次序下的前驱和后继结点，构成**线索二叉树**。对二叉树以某种次序进行遍历并加上线索的过程称为**线索化**。
- 线索二叉树中，原先非空的链保持不变，仍然指向该结点的左、右子结点，它记录的是结点间的层次关系。原先空的左链用来指向遍历中该结点的前驱结点，原先空的右链指向后继结点，它记录的是结点间在遍历时的顺序关系。为了区别每条链是否是一个线索，可以在二叉树的结点结构中设置两个状态字段lefttag和righttag，用以标记相应链的状态。

线索二叉树的结点结构

由5个域构成：data, left, right, lefttag和righttag

$$\text{lefttag} = \begin{cases} \text{true} & \text{left为线索，指向前驱结点} \\ \text{false} & \text{left为普通链，指向左子结点} \end{cases}$$

$$\text{righttag} = \begin{cases} \text{true} & \text{right为线索，指向后继结点} \\ \text{false} & \text{right为普通链，指向右子结点} \end{cases}$$



中序线索二叉树

6.4.2 线索二叉树类的实现

1. **定义线索二叉树结点类**: ThreadBinaryTreeNode<T>，其中有5个成员变量：data用于表示数据元素，left和right分别是指向左、右孩子结点的链，lefttag和righttag为线索标记。线索二叉树结点类也是自引用的泛型类，结点的值类型在对象实例化时决定。

2. 定义线索二叉树类: ThreadBinaryTree，其中成员变量root指向二叉树的根结点。

6.4.3 二叉树的中序线索化

设*p*指向一棵二叉树的某个结点，front指向*p*的前驱结点，它的初值为*null*。当*p*非空时，执行以下操作：

- 中序线索化*p*结点的左子树。
- 如果*p*的左子树为空，设置*p*的lefttag标记为true，它的left链为指向前驱结点front的线索。
- 如果*p*的右子树为空，设置*p*的righttag标记为true
- 如果前驱结点front非空并且它的右链为线索，设置front的right链为指向*p*的线索。
- 移动front，使front指向*p*。
- 中序线索化*p*结点的右子树。

如果一开始让*p*指向二叉树的根结点root，则上述过程线索化整个二叉树。

6.4.4 线索二叉树的遍历

1. **中序线索二叉树中查找中根次序的后继结点。**
2. **中序线索二叉树的中根次序遍历。**
3. 中序线索二叉树中查找先根次序的后继结点。
4. 中序线索二叉树的先根次序遍历。

1. 中序线索二叉树中查找中根次序的后继结点

- 设p指向当前结点，执行以下操作：
  - 如果p结点的右子树为线索，则p的right链为其后继结点，设置p为该结点。
  - 否则说明p的右子树为非空，则p的后继结点是p的右子树上第一个中序访问的结点，即p的右孩子的最左边的子孙结点，设置p为该结点。
- 返回p，作为当前结点在中根次序下的后继结点。

在线索二叉树结点类ThreadBinaryTreeNode中，增加NextNodeInOrder方法，以查找某结点在中根次序下的后继结点。

```
public ThreadBinaryTreeNode<T> NextNodeInOrder() {
  ThreadBinaryTreeNode<T>  p = this;
  if (p.RightTag) //右子树为空时
    p = p.Right;  //right指向后继结点
  else {//右子树非空时
     p = p.Right; //进入右子树
     while (!p.LeftTag)//找到最左边的子孙结点
       p = p.Left;
  }
  return p;
}
```

2. 中序线索二叉树的中根次序遍历

非递归算法：

- 寻找第一个访问结点。它是根的左子树上最左边的子孙结点，用p指向该结点。
- 访问p结点。
- 找到p的后继结点，用p指向该结点，跳转到上一步，直至p为null，说明已访问了序列的最后一个结点。

在线索二叉树ThreadBinaryTree类中，增加ShowUsingThreadInOrder方法，它首先找到遍历序列的第一个结点，然后调用结点类中的NextNodeInOrder方法依次找到后继结点，在中序线索二叉树中完成中根次序遍历。


```
public void ShowUsingThreadInOrder() {
  ThreadBinaryTreeNode<T> p = root;
  if (p != null) {
    Console.Write("中根次序： ");
    while (!p.LeftTag)
      p = p.Left; //找到根的最左边子孙结点
    do {
      Console.Write(p.Data + " ");
      p = p.NextNodeInOrder();//返回p的后继结点
    } while (p != null);
    Console.WriteLine();
  }
}
```

6.5 用二叉树表示树与森林

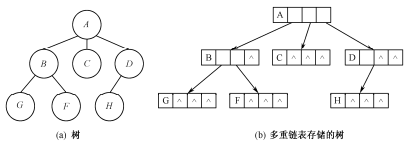
- 二叉树是一种特殊的树，它的实现相对容易；一般的树和森林实现起来比较麻烦，树和森林可以转换为二叉树进行处理。
  1. 树与森林转化为二叉树
  2. 二叉树还原为树与森林

1）树的多重链表结构

- n个结点、度为k的树，每个结点需用k个链指向子结点，在这种多重链链式结构中，总链数为$n\times k$，其中只有n-1个非空的链指向除根以外的n-1个结点。空链与链总数之比为：

$$\frac{\text{空链数}}{\text{总链数}}=\frac{nk-(n-1)}{n\times k}\approx 1-\frac{1}{k}$$

可能造成大量存储空间的浪费



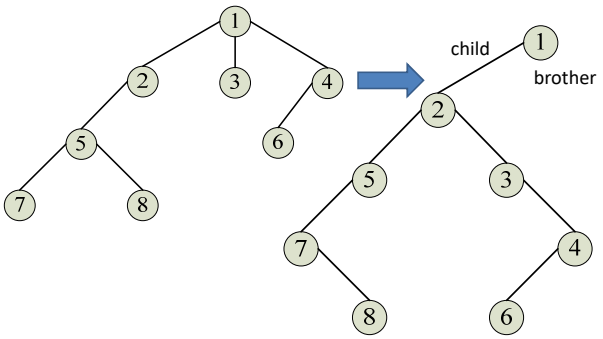
(a) 树

(b) 多重链表存储的树

2）树的“孩子-兄弟”存储结构

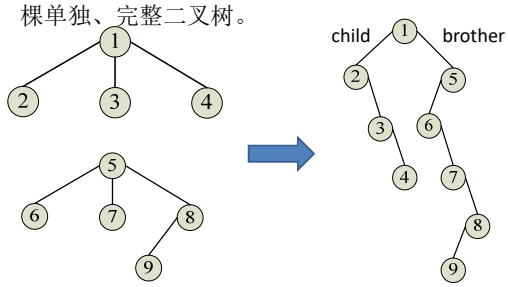
- 树的“孩子-兄弟”存储结构将一棵树转换成了一棵二叉树。该结构的结点有3个域：
  - data存放结点数据。
  - child指向该结点的第一个孩子结点。
  - brother指向该结点的下一个兄弟结点。
- 对于给定的一棵树，按照以上规则，可以得到唯一的二叉树表达式，也就是有唯一的一棵二叉树与原树结构相对应。
- 由于树的根结点没有兄弟结点，所以相应的二叉树表达式中的根结点没有右子树。

树转化为二叉树



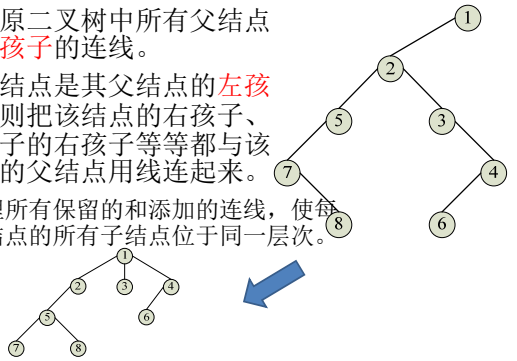
3）森林转化成二叉树的形式存储

- 将森林中的每棵树转化成二叉树。
- 由每颗树的根结点的brother链将若干颗树连接成一棵单独、完整二叉树。



4）二叉树还原为树

- 删除原二叉树中所有父结点与右孩子的连线。
- 若某结点是其父结点的左孩子，则把该结点的右孩子、右孩子的右孩子等等都与该结点的父结点用线连起来。
- 整理所有保留的和添加的连线，使每个结点的所有子结点位于同一层次。



本章学习要点

1. 熟练掌握二叉树的结构特性。熟悉二叉树的各种存储结构的特点及适用范围。
2. 熟悉树的各种存储结构及其特点，掌握树和森林与二叉树的转换方法。
3. 遍历二叉树是二叉树各种操作的基础。实现二叉树遍历的具体算法与所采用的存储结构有关。掌握各种遍历策略的递归算法，灵活运用遍历算法实现二叉树的其它操作。
4. 建立存储结构是进行其它操作的前提，因此应掌握 2 至 3 种建立二叉树和树的存储结构的方法。

作业6

6.1 二叉树是否就是度为2的树？为什么？

6.2 什么样的二叉树，它的先根与后根次序相同？

6.3 先根遍历序列为ABCD的二叉树有多少种形态？

6.4 对于如图6.15所示的二叉树，求先、中、后三种次序的遍历序列。

6.5 给定一棵链式存储的二叉树，完成若干功能，并比较是否可用先、中、后三种次序的遍历算法。

实习6

- 实验目的
  掌握二叉树的定义、性质、存储结构、遍历原理与实现方法。
- 题意
  在一棵二叉树中，求每个结点所在的层次。
- 实验要求：建立链式存储的二叉树，并研究是否可用多种次序的遍历算法。