IPL 电子信息学院 武汉大学 Wuhan University

算法与数据结构
(基于现代C++的方法及实践)
ALGORITHM & DATA STRUCTURE IN MODERN C++

第9章 树与二叉树

王文伟 Wang Wenwei, Dr.-Ing.
Tel: 189-71562600
Email: wwwang@aliyun.com
课程QQ群：珞珈EIS数据结构与算法，668792335

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。

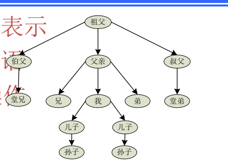
电子信息学院 Table of Contents 武汉大学 Wuhan University

9.0 Introduction

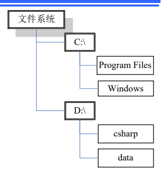
- 树结构是一种数据元素之间具有层次关系的非线性结构。树结构从自然树抽象而来，树的根、枝杈和叶子分别对应于层次结构的起源、分支和终点。
- 树结构有树和二叉树两种。二叉树是最多只有两个子树且两个子树有左右之分的有序树。
- 本章介绍具有层次关系的树结构，重点讨论二叉树的定义、性质、存储结构和遍历算法，并讨论线索二叉树的定义、存储结构和遍历算法。

9.1 树的定义与基本术语

9. 1.1 树的定义和表示
9. 1.2 树的基本术语
9. 1. 3 树的基本操作



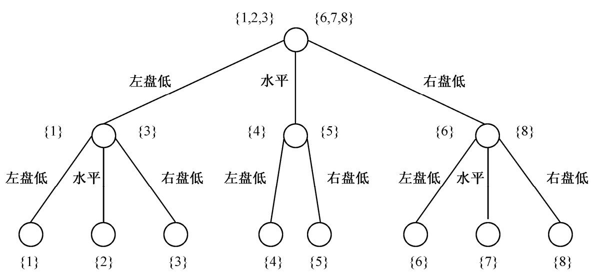
(a) 家谱树



(b) 文件系统

- 现实世界很多对象之间具有层次关系，如家族成员、机构部门、计算机文件系统等，类似于自然界中的树。
- 文件系统具有树型结构，根目录是树的根节点，子目录是树中的分支节点，文件是树的叶子结点。
- 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。

例：以树结构描述测试假币的称重策略



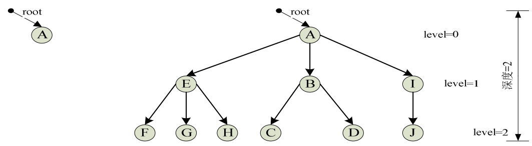
9.1.1 树的定义和表示

◆树(tree)是由n个结点组成的有限集合，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有一个称为根结点的特殊结点(root)。

➢ 当n>1时，根结点之外的其他结点分为m个互不相交的集合$T_1, T_2, \dots, T_m$，每个T_i本身就是一棵树，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。

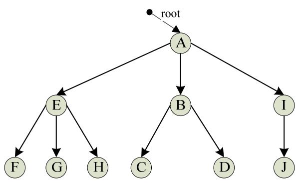
◆如果树中结点的子树T_1，T_2，…从左至右是有次序的，则称该树为有序树（orderd tree）。

树与森林

◆若干棵互不相交的树的集合称为森林（forest）。

➢将树的根结点删除就变成由子树组成的森林。

➢给森林加上一个根结点就变成一棵树。

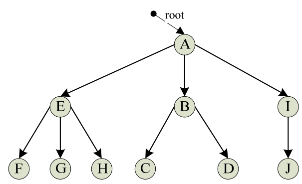


树的广义表形式表示

◆ 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。



9.1.2 树的基本术语

与树有关的术语常用家族成员之间的关系来定义与说明

◆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其子结点的指针构成。

◆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: 后代是指结点的所有子结点，以及各子结点的子结点。而从根到结点N所经过的所有结点，称为其祖先结点。

树的基本术语(II)

◆degree of node&tree ：结点的度定义为其所拥有子树的棵数。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，构成树的一条分支。

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 of node：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 of tree：　树中结点的最大层次数称为树的深度或高度。

9.1.3 树的基本操作

- Initialize：初始化。建立一个树实例并初始化它的结点集合和边的集合。
- AddNode /AddNodes：在树中设置、添加一个或若干个结点。
- Get/Set：访问。获取或设置树中的指定结点。
- Count：求树的结点个数。
- AddEdge：在树中设置、添加边，即结点之间的关联。
- Remove：删除。从树中删除一个数据结点及相关联的边。
- Contains/IndexOf：查找。在树中查找满足某种条件的结点（数据元素）。

9.2 二叉树的定义与实现

9.2.1 二叉树的定义
9.2.2 二叉树的性质
9.2.3 二叉树的存储结构
9.2.4 二叉树类的定义

➢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（binary tree），二叉树易于在计算机中表示和实现。

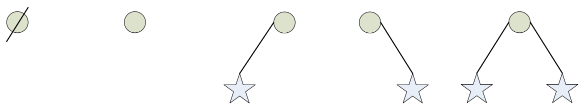
9.2.1 二叉树的定义

- 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。

(a) 树1　　(b) 树2

树：　两者相同
二叉树:两者不同

二叉树的五种基本形态

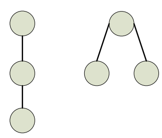


(a)　(b)　(c)　(d)　(e)

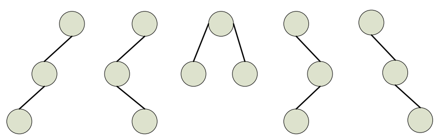
（a）为空的二叉树。
（b）为只有一个结点（根结点）的二叉树。
（c）为由根结点，非空的左子树和空的右子树组成的二叉树。
（d）为由根结点，空的左子树和非空的右子树组成的二叉树。
（e）为由根结点，非空的左子树和非空的右子树组成的二叉树。

【例9.1】画出3个结点的树与二叉树的基本形态

(a) 3个结点的树：
2种基本形态



(b) 3个结点的二叉树：
5种基本形态



9.2.2 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。
  - 基础：根是i=0层上的唯一结点，故2^i=1，命题成立。
  - 假设：对所有$j(0 \leq j < i)$，j层上的最大结点数为2^j。
  - 推理：第i-1层上的最大结点数为2^{i-1}；由于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)

根深叶茂

- 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：$n = n_0 + n_1 + n_2$

1度结点有1个子女，2度结点有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2度结点的数目多1。

二叉树的性质(III)

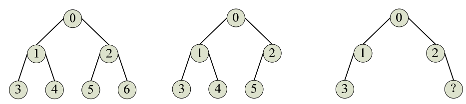
- 性质四：如果一棵完全二叉树有n个结点，则其深度

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

- 深度为k的满二叉树 (full binary tree) 具有2^{k+1}-1个结点。每层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，如果它的每个结点按自上而下、自左至右的顺序编号，并且与深度为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)。

完全二叉树与满二叉树

- 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。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$i \neq 0$，则结点i的双亲是编号为$j=(i-1)/2$(取整)的结点。
  - 若$2i+1 \leq n-1$，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$2i+2 \leq n-1$，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。

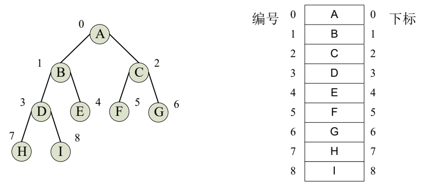
9.2.3 二叉树的存储结构

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

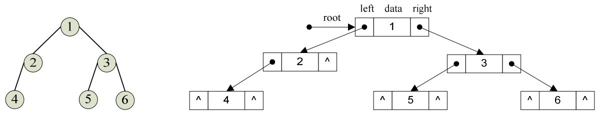
◆ 顺序存储结构适用于完全二叉树，对完全二叉树进行顺序编号，将编号为 *i* 的结点存放在数组下标为 *i* 的位置上。根据二叉树的性质五，对于结点*i*，可以直接计算得到其父结点、左子结点和右子结点的位置。



(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：空二叉树，置root为nullptr。
◆ 二叉树中某结点的左子结点也代表该结点的左子树，若左子树为空，则其left链置为nullptr；同理，该结点的右子结点代表它的右子二叉树。若右子树为空，则其right链置为nullptr。



(a) 二叉树　　(b) 链式存储结构

8.2.4 二叉树类的定义

二叉树的结点类

```
template <typename T> struct BTNode {
  T data;           //存放结点值
  BTNode<T>* right; //指向右子结点的指针
  BTNode<T>* left;  //指向左子结点的指针
  // 构造函数，构造值为k的结点
  BTNode(const T& k) : data(k), left(nullptr),
          right(nullptr) {}
  // 缺省构造函数，构造缺省值的结点
  BTNode() : data{}, left(nullptr),
          right(nullptr) { }
  //析构函数
  ~BTNode() {}  ...... };
```


二叉树类

```
template <typename T>
class BTree {
protected:
  BTNode<T>* _root;//指向二叉树的根结点
public:
      //构造空二叉树
      BTree() : _root(nullptr) {  }
      // 获取和设置根结点
      const BTNode<T>* root() const {return _root;}
      BTNode<T>*& root() {return _root;}
      ...... };
```

◆ Btree模板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。

二叉树类

```
void dispose() {
  if (_root != nullptr) {
    dispose(_root); _root = nullptr;
  } }
void dispose(BTNode<T>* p) {
  if (p != nullptr) {
    dispose(p->left);
    dispose(p->right);
    delete p;
  } }
```

◆ dispose()函数用来销毁二叉树实例（包括根结点在内的所有结点）.

9.3. 二叉树的遍历

9.3.1. 二叉树遍历的过程
9.3.2. 二叉树遍历的递归算法
9.3.3. 二叉树遍历的非递归算法
9.3.4. 按层次遍历二叉树

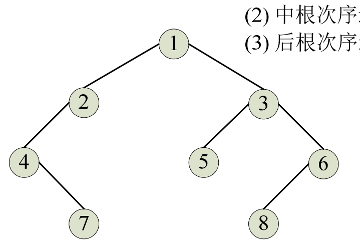
遍历二叉树就是按照一定规则和次序访问二叉树中的所有结点，并且每个结点仅被访问一次。

9.3.1 二叉树遍历的过程

- 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。
- 遍历可以按层次的高低次序进行，即从根结点开始，逐层深入，同层从左至右依次访问结点。
- 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，则遍历操作为空操作，直接返回；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。

9.3.2 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，则该操作为空操作，直接返回；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 BTNode

```
void showPreOrder() {
   cout << this->data << " ";
   BTNode<T>* q = this->left;
   if (q != nullptr)q->showPreOrder();
   q = this->right;
   if (q != nullptr)q->showPreOrder(); }
```

线性表

```
void traversalPreOrder(vector<T>& sql) {
   sql.push_back(this->data);
   BTNode<T>* q = this->left;
   if (q != nullptr)q->traversalPreOrder(sql);
   q = this->right;
   if (q != nullptr)q->traversalPreOrder(sql);  }
```


从根结点开始先根次序遍历二叉树 BTree

```
void showPreOrder() {
   if (_root == nullptr)    return;
   cout << "先根次序：  ";
   _root->showPreOrder();
   cout << endl;
}
```

```
void traversalPreOrder(vector<T>& sql) {
   _root->traversalPreOrder(sql);
}
```

中根次序遍历二叉树的递归算法 BTNode

```
void showInOrder() {
  BTNode<T>* q = this->left;
  if (q != nullptr)q->showInOrder();
  cout << this->data << " ";
  q = this->right;
  if (q != nullptr)q->showInOrder();        }
```

```
void traversalInOrder(vector<T>& sql) {
  BTNode<T>* q = this->left;
  if (q != nullptr)q->traversalInOrder(sql);
  sql.push_back(this->data);
  q = this->right;
  if (q != nullptr)q->traversalInOrder(sql);     }
```


从根结点开始中根次序遍历二叉树 BinaryTree

```
void showInOrder() {
  if (_root == nullptr)   return;
  cout << "中根次序：  ";
  _root->showInOrder();
  cout << endl;
}
```

```
void traversalInOrder(vector<T>& sql) {
  _root->traversalInOrder(sql);
}
```


后根次序遍历二叉树的递归算法 BTNode

```
void showPostOrder() {
  BTNode<T>* q = this->left;
  if (q != nullptr)q->showPostOrder();
  q = this->right;
  if (q != nullptr)q->showPostOrder();
  cout << this->data << " ";    }
```

```
void traversalPostOrder(vector<T>& sql) {
  BTNode<T>* q = this->left;
  if (q != nullptr)q->traversalPostOrder(sql);
  q = this->right;
  if (q != nullptr)q->traversalPostOrder(sql);
  sql.push_back(this->data);  }
```


从根结点开始后根次序遍历二叉树 BTree

```
void showPostOrder() {
  if (_root == nullptr)   return;
  cout << "后根次序：  ";
  _root->showPostOrder();
  cout << endl;
}
```

```
void traversalPostOrder(vector<T>& sql) {
  _root->traversalPostOrder(sql);
}
```


【例9.2】按先根、中根和后根次序遍历二叉树

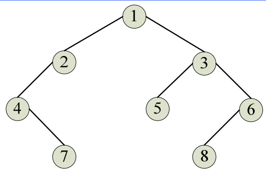
```
BTree<int> btree;
BTNode<int> nodes[9]{0,1,2,3,4,5,6,7,8};
btree.root() = nodes+1;
nodes[1].left = nodes+2;
nodes[1].right = nodes+3;
nodes[2].left = nodes+4;
nodes[3].left = nodes+5;
nodes[3].right = nodes+6; ……
btree.showPreOrder(); btree.showInOrder();
btree.showPostOrder();
```

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。

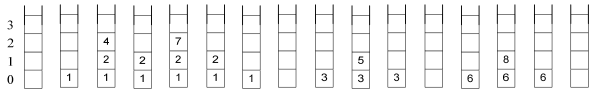
9.3.3 二叉树遍历的非递归算法

- 中根次序遍历规则: 在每个结点处，先选择遍历左子树，其后必须返回该结点，对其进行访问，然后遍历右子树。
- 设置一个栈s来记录经过的路径。结点指针变量*p*从根结点开始，如果*p*不空或栈*s*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，将当前结点指针*p*入栈（s.push(p)），进入其左子树（p=p->left）。
  2. 如果*p*为空并且栈*s*不空，表示已走过一条路径，必须返回一步以寻找另一条路径。置*p*指向出栈的结点（p=s.top()），访问*p*结点，再进入*p*的右子树（p=p->right）。

二叉树的非递归中根遍历过程



(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



Pop: 4 7 2 1 5 8 6

非递归中根遍历序列按出栈顺序：4 7 2 1 5 3 8 6

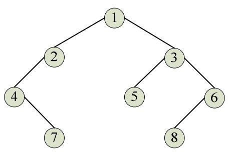
定义在二叉树类中

```
void showInOrderNR() {
  stack<BTNode<T>*> s; BTNode<T>* p = _root;
  cout << "非递归中根次序：  ";
  while(p!= nullptr||s.size()!= 0){
         //p非空或栈非空时
    if (p != nullptr) {
    s.push(p);    //当前结点指针p入栈
    p = p->left; //进入左子树
    }  else {//p为空而栈非空时
      p = s.top(); s.pop(); //出栈的结点由p指向
      cout << p->data << " ";  //访问结点
      p = p->right; //进入右子树
    }  }
  cout << endl;}
```

9.3.4 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各结点。

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。



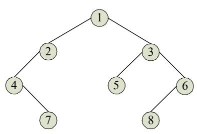
按层次遍历二叉树的非递归算法

- 设置一个队列变量q。结点指针变量*p*从根开始，当*p*不为空时，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=q.front()），否则置p为nullptr。

定义在二叉树类中

```
void showByLevel() {
  BTNode<T>* p = _root;
  QueueSP<BTNode<T>*> q;//设立一个空队列
  cout << "层次遍历：  ";
  while (p != nullptr) {
    cout << p->data << " ";
    if (p->left != nullptr)
      q.enqueue(p->left);  //p的左链入队
    if (p->right != nullptr)
      q.enqueue(p->right);//p的右链入队
    if (q.size() != 0){
      p = q.front(); q.dequeue();
      //队首赋值给p，且出队
    }else p = nullptr;
  } cout << endl;   }
```

按层次遍历二叉树时队列内容的变化



Dequeue	2	3						
2		3						
		3	4					
3			4					
			4	5	6			
4				5	6			
				5	6	7		
5					6	7		
6						7		
						7	8	
7							8	
8								

按层次遍历序列=根+出队序列，即：12345678

9.4 构建二叉树

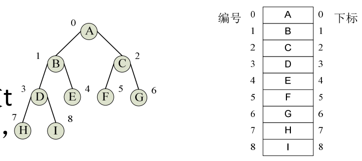
- 给定一定的条件，可唯一建立二叉树。例如对于完全二叉树，如果各结点值已按顺序存储在数组，即可唯一建立一颗二叉树。
- 一般情况下，建立一棵二叉树必须明确以下两点：
  - 结点与其父结点及子结点间的层次关系。
  - 兄弟结点间的左右顺序关系。
- 广义表形式有时不能唯一表示二叉树，需用特殊的广义表形式才能唯一描述，例如在二叉树的广义表表示式中清楚地标明空子树，给定这种形式的广义表表示式，可以唯一地建立一颗二叉树。
- 对于给定的一棵二叉树，遍历产生的先根、中根、后根序列是唯一的；反之，仅知一种遍历序列，并不能唯一确定一棵二叉树。先根次序或后根次序反映双亲与孩子结点的层次关系，中根次序反映兄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

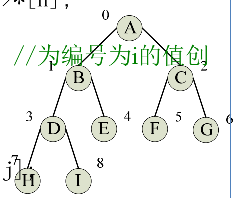
1. 建立链式存储结构的完全二叉树

- 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Btree.中，增加全局函数byArray，它的参数t指向线性表或数组，用以表示顺序存储的完全二叉树结点值的序列。



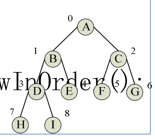
(a) 完全二叉树　(b) 顺序存储完全二叉树


```
template <typename T>
void byArray(const T* t, int cnt, BTree<T>* bt)
{ int n = cnt;
  if (n == 0) bt->root() = nullptr; int i, j;
  BTNode<T>** q = new BTNode<T>*[n];
  for (i = 0; i < n; i++)
    q[i]= new BTNode<T>(t[i]); //为编号为i的值创建结点
  for (i = 0; i < n; i++) {
    j = 2 * i + 1;
    if (j < n) q[i]->left = q[j];
    else q[i]->left = nullptr;
    j++; if (j < n)q[i]->right = q[j];
    else q[i]->right = nullptr;
  } bt->root() = q[0]; }
```



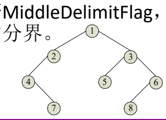
【例9.3】根据给定数组建立完全二叉树

```
const int CNT = 8;
int it[CNT] = { 0,1,2,3,4,5,6,7 };
BTree<int> btree; byArray(it, CNT, &btree);
btree.showPreOrder();btree.showInOrder();
btree.showPostOrder(); btree.showByLevel();
btree.dispose();
char ct[CNT]= {'A','B','C','D','E','F','G','H' };
BTree<char> btree2;
byArray(ct, CNT, &btree2);
btree2.showPreOrder();btree2.showInOrder();
btree2.dispose();
```



2.根据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,^)))

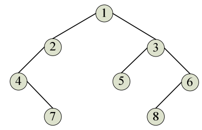
根据给定的广义表表示式建立二叉树的算法

- ◆ 依次读取二叉树的广义表表示序列中的每个符号元素，检查其内容，如果
  - ➢遇到有效数据值，则建立一个二叉树结点对象；扫描下一元素，如果
    - 它为LeftDelimitFlag，则LeftDelimitFlag和RightDelimitFlag之间是该结点的左子树与右子树，递归调用，分别建立左、右子树，返回结点对象。
    - 没有遇到LeftDelimitFlag，表示该结点是叶子结点。
  - ➢遇到NullSubtreeFlag，表示空子树，返回null值。
- ◆ 在二叉树Btree.h中， 增加全局函数byGList，它的前两个参数表示顺序存储的广义表表示式，最后一个参数定义广义表表示式所用的分界符。


```
template <typename T>
void byGList(T* sList,int cnt,BTree<T>* bt, const
      ListFlags<T>* pCustomListFlags = nullptr){
  idx = 0;                    // 初始化递归变量
  ListFlags<T> DefaultFlags{'^', '(', ')', ',' };
  if (cnt > 0) {
    ListFlags<T>* p;
    if (pCustomListFlags != nullptr)
      p = (ListFlags<T>*)pCustomListFlags;
    else p = &DefaultFlags;
    pListFlags = p;// 全局静态指针变量记录界符结构
    bt->root() = rootByGList(sList);
  }
  else bt->root() = nullptr;
  return; }
```

```
BTNode<T>* rootByGList(T* sList){
  BTNode<T>* p = nullptr;
  T nodeData = sList[idx];        // 序列当前元素的值
  ListFlags<T>* pFlags = (ListFlags<T>*)pListFlags;
  if (isData(nodeData, pFlags)) {
    p = new BTNode<T>(nodeData);// 有效数据，建立结点
    idx++;
    nodeData = sList[idx];
    if (nodeData== pFlags->LeftDelimitFlag) {
      idx++;                 //左边界，如'(', 跳过
      p->left = rootByGList(sList); //建立左子树，递归
      idx++;                 //跳过中界符，如','
      p->right= rootByGList(sList); //建立右子树，递归
      idx++;                 //跳过右边界，如')'
    } }
  if (nodeData==pFlags->NullSubtreeFlag)
    idx++;//空子树符，跳过，返回nullptr
  return p;  }
```

```
ListFlags<char> flags{'^', '(', ')', ','};
string s = "1(2(4(^,7),^),3(5,6(8,^)))";
cout<<"Generalized List: "<<s << endl;
BTree<char> btree;
byGList((char*)s.data(),s.length(), &btree);
// byGList((char*)s.data(),s.length(), &btree, &flags);
btree.showPreOrder(); btree.showInOrder();
btree.dispose();
```

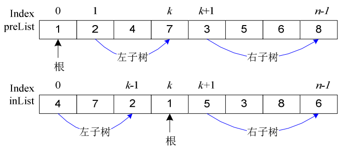


(1) 先根次序遍历序列：1 2 4 7 3 5 6 8
(2) 中根次序遍历序列：4 7 2 1 5 3 8 6
(3) 后根次序遍历序列：7 4 2 5 8 6 3 1

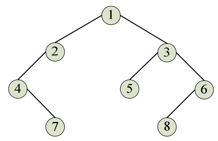
3.根据先根和中根次序遍历序列建立二叉树

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存储在线性表或数组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;
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9.5】按先根和中根次序遍历序列建立二叉树

```
vector<int> prelist { 1, 2, 4, 7, 3, 5, 6, 8 };
vector<int> inlist { 4, 7, 2, 1, 5, 3, 8, 6 };
BTree<int> btree;
by2Lists(prelist, inlist, &btree);
btree.showPreOrder(); btree.showInOrder();
```


9.5 用二叉树表示树与森林

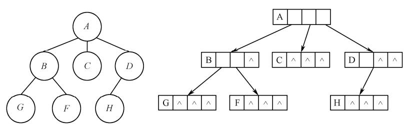
- 二叉树是一种特殊的树，它的实现相对容易；一般的树和森林实现起来比较麻烦，树和森林可以转换为二叉树进行处理。
  1. 树与森林转化为二叉树
  2. 二叉树还原为树与森林

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：

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

可能造成大量存储空间的浪费

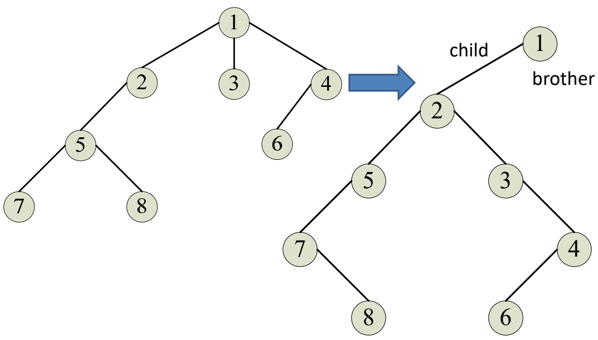


(a) 树　(b) 多重链表存储的树

2）树的“孩子-兄弟”存储结构

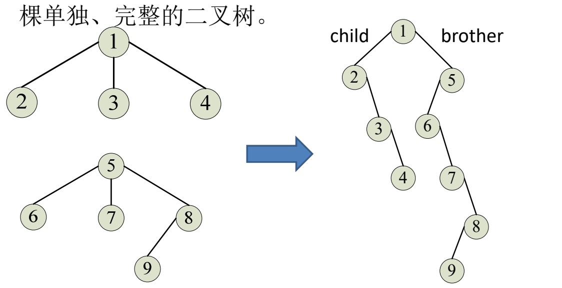
- 树的“孩子-兄弟”存储结构将一棵树转换成了一棵二叉树。该结构的结点有3个域：
  - 数据域data，存放结点数据。
  - 左链域child，指向该结点的第一个孩子结点。
  - 右链域brother，指向该结点的下一个兄弟结点。
- 对于给定的一棵树，按照以上规则，可以得到唯一的二叉树表达式，也就是有唯一的一棵二叉树与原树结构相对应。
- 由于树的根结点没有兄弟结点，所以相应的二叉树表达式中的根结点没有右子树。

树转化为二叉树



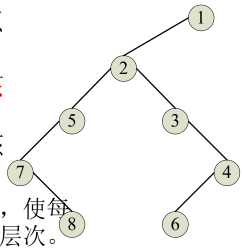
3）森林转化成二叉树的形式存储

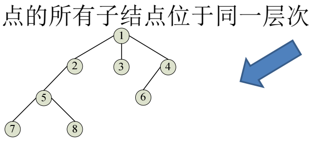
- 将森林中的每棵树转化成二叉树。
- 由每颗树的根结点的brother链将若干颗树连接成一棵单独、完整的二叉树。



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。

实习9

- 实验目的
  理解树与二叉树的基本概念及其基本操作，熟练掌握二叉树的性质、存储结构、遍历原理与实现方法。熟练掌握Visual Studio进行调试和多项目管理。
- 题意
  熟练掌握Visual Studio进行调试与多项目和类的创建和管理。完成本次全部实验，需三个项目：1）一个“静态库（C++/Windows）”型项目（如称作dsa），在其中增加二叉树及其结点模块（.cpp和.h文件），用于封装各种自定义的二叉树数据结构。2）一个“控制台应用（C++/Windows）”类型的项目（如称作exp7app），用于二叉树结构的测试、演示和应用。项目exp7app需要引用dsa类库项目。3）一个“Windows窗体应用（C#/.Net Framework）”类型的项目（如称作exp7xapp），用于最后一个实验的设计（选做）。