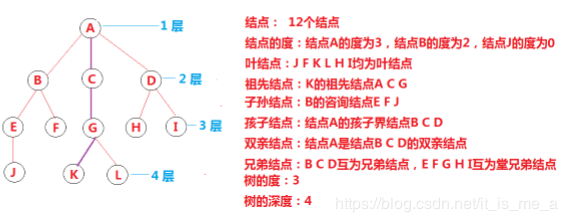
# 树是一种数据结构，有关树的概念如下：

https://blog.nowcoder.net/n/833201edc8cc42f58d1e2b84dbcae690

结点：结点包括一个数据元素及若干指向其他子树的分支(指针(索引))  
结点的度：结点所拥有子树的个数称为该结点的度  
树的度：树中所有结点的度的最大值称为该树的度  
叶子结点（终端节点）：度为0的结点称为叶子结点  
分支结点（非终端节点）：度不为0的结点称为分支结点，一棵树中除叶节点外的所有节点都是分支结点  
祖先结点：从根节点到该结点所经分支上的所有节点  
子孙结点：以某节点为根节点的子树中所有节点  
双亲结点（前驱结点）：一个结点称之为其后继结点的双亲节点。  
孩子结点（后继结点）：树中一个节点的子树的根节点称为该结点的孩子结点  
兄弟结点：具有同一双亲的结点之间互称为为兄弟结点  
结点层次：从根节点到树中某节点所经路径上的分支数称为该结点的层次，根结点为第一层，其孩子节点为第二层，以此类推。  
树的深度（高度）：树中所有节点层次的最大值称为该树的深度  
有序树和无序树：树中各结点从左到右是有次序的，即若交换了某结点各子树的相对位置则构成了不同的树，那么称之为有序树，反之则为无序树。  
森林：m棵树的集合(m大于等于0)。在自然界中树和森林是两个不同的概念，但在数据结构中，它们之间的差别很小。删去一棵非空树的根节点，树就变成森林；反之若增加一个节点，让森林中每一棵树的根节点都变成他的子女，森林就变成一棵树



## 树的存储结构

计算机中存储树的信息，要求既要存储结点的数据元素信息，又要存储结点之间的逻辑关系信息

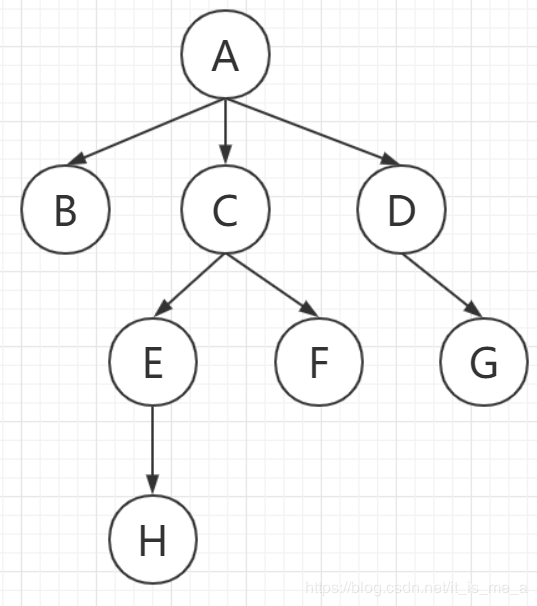
下面介绍常用的存储结构：

**1. 双亲表示法**

双亲表示法：

* 优点：寻找一个节点的双亲结点操作实现很方便
* 缺点：寻找一个节点的孩子结点很不方便

对于下图的树的存储，双亲表示法有两种实现方式：



| **1. 一维数组实现** | **2. 用指针表示出每个结点的双亲节点** |
| --- | --- |
| 用一维数组存储树中的各个结点，其中数组元素是个记录，包含data和parent两个字段，分别表示结点的数据值和其双亲在数组中的下标。 | https://uploadfiles.nowcoder.com/images/20190824/763669921_1566610860342_4DCEB2E392292C2584AED06C911A378E |
| https://uploadfiles.nowcoder.com/images/20190824/763669921_1566610860405_4DCEB2E392292C2584AED06C911A378E | https://uploadfiles.nowcoder.com/images/20190824/763669921_1566610860657_4DCEB2E392292C2584AED06C911A378E |

**2. 孩子表示法**

孩子表示法：

* 优点：寻找一个节点的孩子结点比较方便
* 缺点：寻找一个节点的双亲结点很不方便

孩子表示法的实现方式：用指针表示出每个结点的孩子结点，但是有个问题：结点的数据结构中应该给几个孩子结点指针呢？，看树的度。

数据结构：

typedef int DataType;

struct Node

{

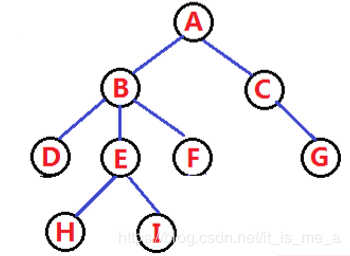
    struct Node\* \_pChild1;

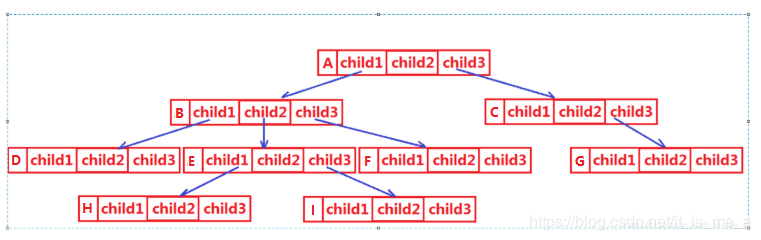
    struct Node\* \_pChild2;

    struct Node\* \_pChild3;

    DataType \_data;  //数据域

};





**3. 双亲孩子表示法**

双亲孩子表示法：用指针既表示出每个结点的双亲结点，也表示出每个结点的孩子结点，即：双亲表示法+孩子表示法

typedef int DataType;

struct Node

{

struct Node\* \_pParent; //指向双亲节点

struct Node\* \_pChild1;

struct Node\* \_pChild2;

struct Node\* \_pChild3;

DataType \_data; //数据域

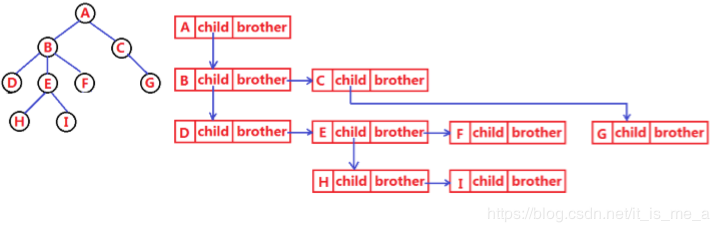
};

**4. 孩子兄弟表示法**

孩子兄弟表示法：即表示出每个结点的第一个孩子结点，也表示出每个结点的下一个兄弟结点。孩子兄弟链表存储结构是一种二叉链表，链表中的

每一个结点包含两个指针，

分别指向对应结点的第一个孩子结点和下一个孩子结点。



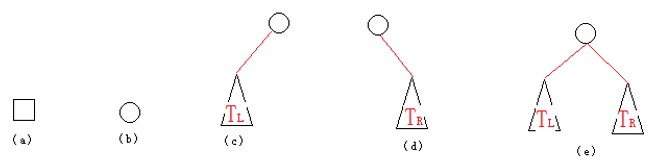
# 2. 二叉树

二叉树是一种特殊的树。二叉树的特点：

1.每个结点最多有两棵子树，即二叉树不存在度大于2的结点；

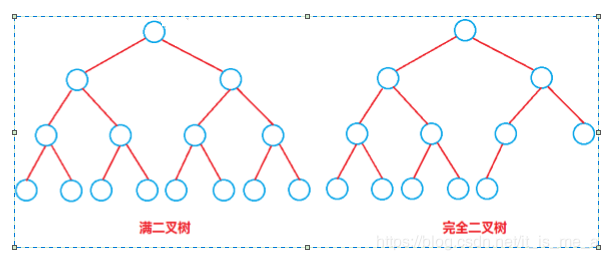
2. 二叉树的子树有左右之分，其子树的次序不能颠倒；

3. 二叉树即使只有一颗子树也要明确指出该子树是左子树还是右子树。



## 满二叉树&完全二叉树

* 满二叉树： 在一棵二叉树中，如果所有分支结点都存在左子树和右子树，并且所有叶子节点都在同一层上
* 完全二叉树：从根起，自上而下，自左而右，给满二叉树的每个结点从1到n连续编号，如果每个结点都与深度为k的满二叉树中编号从1至n的结点一一对应，
* 则称为完全二叉树



## 二叉树性质

1. 二叉树上叶子结点数等于度为2的节点数加1.

证明：  
设在一棵二叉树中，n为所有节点总数，n0为叶子结点个数，n1为度为1的节点个数，n2为度为2的结点个数。

则 n = n0 + n1 + n2 （1）  
再看二叉树中的分支数：除根节点外，所有结点都有一个分支进入，设b为分支数，  
则 n = b + 1  
又因为分支是由度为1或2的结点分出的，因此：  
b = n1 + 2 x n2  
于是： n = n1 + 2 x n2 + 1 （2）  
由（1）（2）得： n0 = n2 +1

1. 二叉树上第i层上至少有2^(i-1)个结点。（i>=1，根结点的层数为1）

证明：数学归纳法

1. 若规定只有根节点的二叉树的深度为1，则深度为K的二叉树的最大结点数是 2^k - 1 (k>=0)
2. 对于完全二叉树（n≥1，n为节点数），如果按照从上至下从左至右的顺序对所有节点从1开始编号，则对于序号为i的结点有：  
   **4.1** 除树的根节点外，若一个结点的编号为i，则双亲编号为：i/2。也就是说，当i为偶数的时候，其双亲结点编号为i/2，他是双亲节点的左孩子；当i为奇数的时候，其双亲结点编号为(i-1)/2，他是双亲节点的右孩子  
   **4.2** 如果 2 \* i ≤ n，则编号为i的结点为分支结点；否则为叶子结点。  
   **4.3** 若n为奇数，则每个分支结点都既有左孩子又有右孩子；若n为偶数，则编号最大的分支结点只有左孩子，没有右孩子。  
   **4.4** 若编号为i的结点有左孩子，则左孩子结点编号为2i；若编号为i的结点有右孩子，则右孩子结点编号为2i+1

## 二叉树的存储结构

与线性表一样，二叉树也有顺序存储结构和链式存储结构。

1. 顺序存储结构

|  |  |
| --- | --- |
| 1 | typedef DataType SqBinTree[MaxSize];   //其中DataType为二叉树结点的数据类型；MAxSize为顺序表的最大长度。 |

顺序存储一个完全二叉树，首先对该树的节点进行编号，然后以各结点的编号为下标，把各结点的值对应存储到一维数组中。完全二叉树的编号过程为：首先，把树的编号定为1，然后按照层次上从上到下、每层从左到右的顺序对每一结点进行编号。

对于一般二叉树，也可以用这种方法进行存储。首先，在二叉树上补上若干虚拟结点使其成为完全二叉树。然后按照上面的方法进行编号。

下图是个示例：

|  |
| --- |
| https://uploadfiles.nowcoder.com/images/20190824/763669921_1566610861703_4DCEB2E392292C2584AED06C911A378E | https://uploadfiles.nowcoder.com/images/20190824/763669921_1566610861750_066A36068D0AB88771592C2C8913E914 |

顺序存储结构的优点：存储完全二叉树，简单省空间 ；缺点：存储一般二叉树尤其单支树，存储空间利用不高 。

1. 二叉链存储结构

链表中每个结点包含两个指针，分别对应结点的左孩子和右孩子。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | struct BinTreeNode  {      struct BinTreeNode\* \_pLeft;      struct BinTreeNode\* \_pRight;      DataType \_data;  }; |

这种存储结构的优点是访问结点的孩子很方便，有时为了方便访问结点的双亲可在每个节点中再增加一个指向双亲的指针域。

1. 三叉链存储结构

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | struct BinTreeNode  {      struct BinTreeNode\* \_pParent;      struct BinTreeNode\* \_pLeft;      struct BinTreeNode\* \_pRight;      DataType \_data;  }; |

## 二叉树的基本操作（c++实现）

假设二叉树采用二叉链存储结构。

1. 建立二叉链

用ch扫描采用括号表示法表示二叉树的字符串str。分为以下几种情况：  
1）若 ch == ‘(’ ，则将前面刚创建的结点作为双亲节点进栈，并置k=1，表示其后创建的节点将作为这个节点的左孩子节点。  
2）若 ch == ‘(’，表示栈中结点的左右孩子结点处理完毕，退栈。  
3）若 ch == ‘,’，k=2，表示其后创建的节点为右孩子结点  
4）若 ch 为一个字母，创建一个节点，并根据k值建立它与栈中结点之间的联系，当k=1表示这个节点作为栈中栈顶结点的左孩子结点；当k=2表示这个节点作为栈中栈顶结点的右孩子结点；如此循环直至str处理完毕。算法中使用一个栈st保存双亲节点，top为其栈顶指针；k指定其后处理的结点是双亲节点的左孩子结点还是右孩子。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | #define MaxSize 20  typedef char DataType;    typedef struct tnode  {      DataType data;      struct tnode\* pchild1;      struct tnode\* pchild2;  }BTNode;    //建立二叉链  void CreateBTree(BTNode \*&bt, char \*str)  {      BTNode \*st[MaxSize], \*p = NULL;      int top = -1, k, j = 0;      char ch;      bt = NULL;//建立二叉树初始化为空      ch = str[j];      while ('\0' != ch)   //str未扫描完时循环      {          switch (ch)          {          case '(':top++; st[top] = p; k = 1; break;//左孩子节点          case ')':top--; break;          case ',':k = 2; break;  //右孩子结点          default:              p = (BTNode\*)malloc(sizeof(BTNode));              p->data = ch;              p->pchild1 = p->pchild2 = NULL;              if (bt == NULL)                  bt = p;              else              {                  switch (k)                  {                  case 1:st[top]->pchild1 = p; break;                  case 2:st[top]->pchild2 = p; break;                    }              }          }          j++;          ch = str[j];      }  } |

1. 求二叉树高度（递归求法）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | int BTHeight(BTNode \*bt)  {      int left, right;      if (NULL == bt)      {          return 0;      }      else      {          left = BTHeight(bt->pchild1);          right = BTHeight(bt->pchild2);          return (left > right) ? (left + 1) : (right + 1);      }  } |

1. 求二叉树结点个数（递归算法）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | int NodeCount(BTNode \*bt)  {      int left, right;      if (NULL == bt)      {          return 0;      }      else      {          left = NodeCount(bt->pchild1);          right = NodeCount(bt->pchild2);          return left + right + 1;      }  } |

1. 求二叉树叶子结点个数

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | int LeafCount(BTNode \*bt)  {      int left, right;      if (NULL == bt)          return 0;      else if (NULL == bt->pchild1 && NULL == bt->pchild2)          return 1;      else      {          left = LeafCount(bt->pchild1);          right = LeafCount(bt->pchild2);          return left + right;      }  } |

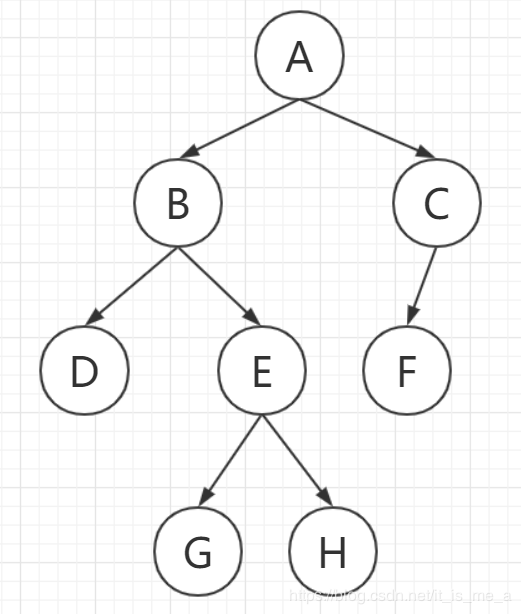
1. 以括号表示法输出二叉树

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | void DisBTree(BTNode \*bt)  {      if (NULL != bt)      {          cout << bt->data;          if (NULL != bt->pchild1 || NULL != bt->pchild2)          {              cout << '(';              DisBTree(bt->pchild1);              if (NULL != bt->pchild2) cout << ",";              DisBTree(bt->pchild2);              cout << ")";          }      }    } |

## 二叉树遍历（重要）–先序遍历、中序遍历、后序遍历和层次遍历

所谓二叉树的遍历，是指按照一定次序访问书中的所有节点，使得每个结点恰好被访问一 次。

二叉树常用的遍历有：先序遍历（先根遍历）、中序遍历、后序遍历和层次遍历。不同遍历方式区别在于访问根节点的顺序。

对于下面这个二叉树，它对应的遍历方式序列为：  


| **名称** | **特点** | **算法实现** | **对应的遍历方式序列** |
| --- | --- | --- | --- |
| 先序遍历 | 若二叉树为空，则： 1）访问根结点； 2）先序遍历左子树； 3）先序遍历右子树。 | https://uploadfiles.nowcoder.com/images/20190824/763669921_1566610862104_4DCEB2E392292C2584AED06C911A378E | ABDEGHCF |
| 中序遍历 | 若二叉树为空，则： 1）中序遍历左子树； 2）访问根节点； 3）中序遍历右子树。 | https://uploadfiles.nowcoder.com/images/20190824/763669921_1566610862157_4DCEB2E392292C2584AED06C911A378E | DBGEHAFC |
| 后序遍历 | 若二叉树为空，则： 1）后序遍历左子树； 2）后序遍历右子树； 3）访问根节点。 | https://uploadfiles.nowcoder.com/images/20190824/763669921_1566610862228_4DCEB2E392292C2584AED06C911A378E | DGHEBFCA |

* 层次遍历

在进行层次访问的时候，对某一层的结点访问完，再按照他们的访问次序对各个结点的左孩子、右孩子顺序访问。层次遍历序列为ABCDEFGH。

层次访问需要一个环形队列qu来实现：先将根结点进队，在队不空时循环；从队列中出列一个节点\*p，访问它；若它有左孩子节点，将左孩子节点进队；若它有右孩子结点，将右孩子结点进队。如此操作直到队空为止。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | void LevelOrder(BTNode \*bt)  {      BTNode \*p;      BTNode \*qu[MaxSize];//定义环形队列      int front, rear;//对头与队尾指针      front = rear = -1;//队列为空      rear++;      qu[rear] = bt;   //根节点进入队列      while (front != rear){         //队列不为空          front = (front + 1) % MaxSize;          p = qu[front];//队头出队列          printf("%c ", p->data);//访问结点          if (NULL != p->leftchild)//有左孩子时将其入队          {              rear = (rear + 1) % MaxSize;              qu[rear] = p->leftchild;          }          if (NULL != p->rightchild)//有右孩子时将其入队          {              rear = (rear + 1) % MaxSize;              qu[rear] = p->rightchild;          }      }  } |

注意

* 对于不同的二叉树，先序遍历序列可能相同
* 对于不同的二叉树，中序遍历序列可能相同
* 对于不同的二叉树，先序遍历序列可能相同
* 对于不同的二叉树，先序遍历序列和后序遍历可能都相同
* 由先序遍历和中序遍历序列能够唯一确定一棵二叉树
* 由后序遍历和中序遍历序列能够唯一确定一棵二叉树

## 线索二叉树

* 线索&线索二叉树

对于n个结点的二叉链，在二叉链存储结构中有n+1个空链域，利用这些空链域存放在某种遍历次序下该节点的前驱结点和后继结点的指针，这些指针称为线索。加上线索的二叉树称为线索二叉树。

在线索二叉树中，除出了相应序列的第一个结点和最后一个节点，二叉树的遍历序列中每个节点都有一个前驱和后驱结点。

下图给出了三种不同的线索二叉树，图中虚线为线索。

[外链图片转存失败(img-nlwdR8El-1566562300545)(1)]

为什么说“对于n个结点的二叉链，在二叉链存储结构中有n+1个空链域？”

在二叉链中，一个结点有两个指针域，所以n个结点共有2n个链域。又因为在二叉树中，除根结点外，每一个结点有且仅有一个双亲，所以只有n-1个结点的链域存放指向非空子结点的指针，这样就还有n+1个空指针。

* 线索二叉树的存储结构

| **算法实现** | **图解** |
| --- | --- |
| https://uploadfiles.nowcoder.com/images/20190824/763669921_1566610862270_46839FD0025AEEAF31C8AB56E3DC5073 | https://uploadfiles.nowcoder.com/images/20190824/763669921_1566610862392_4DCEB2E392292C2584AED06C911A378E |

## 二叉查找树\*

二叉查找树：二叉查找树又称二叉排序树，或者是一课空二叉树，或者是具有如下特征的二叉树：  
a、若它的左子树不空，则左子树上所有结点的值均小于根结点的值  
b、若它的右子树不空，则右子树上所有结点的值均大于根结点的值  
c、它的左、右子树也分别是二叉查找树

## 平衡二叉树

平衡二叉树：平衡二叉查找树简称平衡二叉树，平衡二叉树或者是棵空树，或者是具有下列性质的二叉查找树：它的左子树和右子树都是平衡二叉树，且左子树和右子树的高度之差的绝对值不超过1  
平衡二叉树的失衡及调整主要可归纳为下列四种情况：LL型、RR型、LR型、RL型

# 3. 堆

堆是具有以下特性的完全二叉树，其所有非叶子结点均不大于（或不小于）其左右孩子结点。若堆中所有非叶子结点均不大于其左右孩子结点，则称为小顶堆（小根堆），若堆中所有非叶子结点均不小于其左右孩子结点，则称为大顶堆（大根堆）

[举报](javascript:void(0);)