树

树: 是被称为节点的实体的集合。节点通过边(`edge`)连接。每个节点都包含值或数据(`value/date`)，并且每个节点可能有也可能没有子结点。
节点：使用树结构存储的每一个数据元素都被称为“节点”。
节点的度: 节点所拥有的子树的数量。
根节点：树的首结点(root节点)。
叶节点：如果节点没有任何子节点，那么此结点称为叶子节点（叶节点）。
分支节点：除叶节点外的节点就是分支节点。
树的深度：树中距离根节点最远的节点所处的层次就是树的深度。
树的高度：叶节点的高度为1，非叶节点的高度是它的子女节点高度的最大值加1，高度与深度数值相等，但计算方式不一样。
树的度: 树中节点的度的最大值。

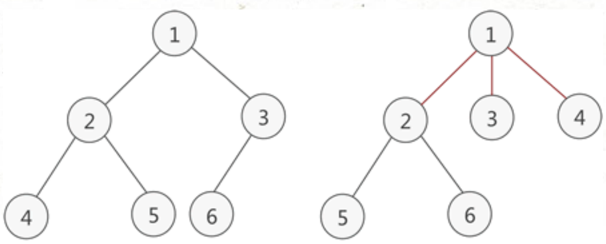
二叉树

特点：

1.本身是有序树

2.树中包含的各个节点的度不能超过2

例如，图 1a) 就是一棵二叉树，而图 1b) 则不是。



a) 二叉树　　b) 非二叉树

性质:

1.二叉树中，第 i 层最多有 2i-1 个结点。(2的i-1次方)；

2.如果二叉树的深度为 K，那么此二叉树最多有 2K-1 个节点。(2的k次方减一)；

3.二叉树中，终端节点数（叶子节点数）为 n0，度为 2 的结点数为 n2，则 n0=n2+1。

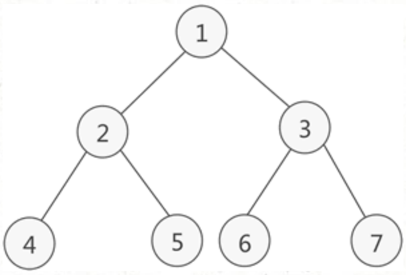
性质 3 的计算方法为：对于一个二叉树来说，除了度为 0 的叶子结点和度为 2 的结点，剩下的就是度为 1 的结点（设为 n1），那么总结点 n=n0+n1+n2。

同时，对于每一个结点来说都是由其父结点分支表示的，假设树中分枝数为 B，那么总结点数 n=B+1。而分枝数是可以通过 n1 和 n2 表示的，即 B=n1+2*n2。所以，n 用另外一种方式表示为 n=n1+2*n2+1。

两种方式得到的 n 值组成一个方程组，就可以得出 n0=n2+1。

满二叉树:

如果二叉树中除了叶子结点，每个结点的度都为 2，则此二叉树称为满二叉树。

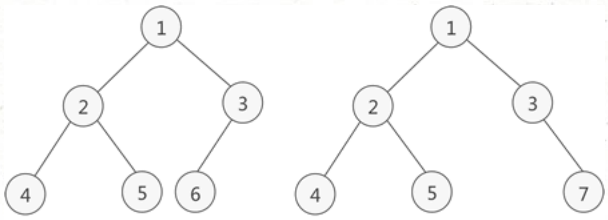


满二叉树除了满足普通二叉树的性质，还具有以下性质：

1. 满二叉树中第 i 层的节点数为 2n-1 个。(2的n-1次方)
2. 深度为 k 的满二叉树必有 2k-1 个节点 ，叶子数为 2k-1。(2的k次方减一)
3. 满二叉树中不存在度为 1 的节点，每一个分支点中都两棵深度相同的子树，且叶子节点都在最底层。
4. 具有 n 个节点的满二叉树的深度为 log2(n+1)。

完全二叉树：

如果二叉树中除去最后一层节点为满二叉树，且最后一层的结点依次从左到右分布，则此二叉树被称为完全二叉树。



a) 完全二叉树　　　　b) 非完全二叉树

如图 3a) 所示是一棵完全二叉树，图 3b) 由于最后一层的节点没有按照从左向右分布，因此只能算作是普通的二叉树。

完全二叉树除了具有普通二叉树的性质，它自身也具有一些独特的性质。

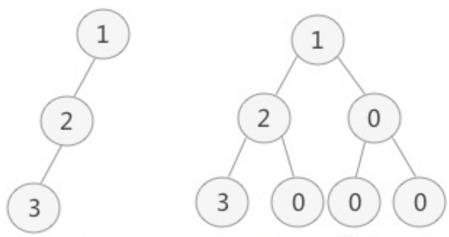
比如说，n 个结点的完全二叉树的深度为 ⌊log2n⌋+1。⌊log2n⌋ 表示取小于 log2n 的最大整数。例如，⌊log24⌋ = 2，而 ⌊log25⌋ 结果也是 2。

对于任意一个完全二叉树来说，如果将含有的结点按照层次从左到右依次标号（如图 3a)），对于任意一个结点 i ，完全二叉树还有以下几个结论成立：

1. 当 i>1 时，父亲结点为结点 [i/2] 。（i=1 时，表示的是根结点，无父亲结点）
2. 如果 2*i>n（总结点的个数），则结点 i 肯定没有左孩子（为叶子结点）；否则其左孩子是结点 2*i 。
3. 如果 2*i+1>n ，则结点 i 肯定没有右孩子；否则右孩子是结点 2*i+1 。

二叉树的顺序存储结构:

二叉树的顺序存储，指的是使用[顺序表]（[数组]）存储二叉树。需要注意的是，顺序存储只适用于完全二叉树。换句话说，只有完全二叉树才可以使用顺序表存储。因此，如果我们想顺序存储普通二叉树，需要提前将普通二叉树转化为完全二叉树。



普通二叉树转完全二叉树的方法很简单，只需给二叉树额外添加一些节点，将其"拼凑"成完全二叉树即可。

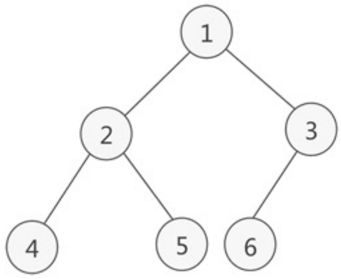


图 2 完全二叉树示意图

所示的完全二叉树，其存储状态:

图 3 完全二叉树存储状态示意图

同样，存储由普通二叉树转化来的完全二叉树也是如此。普通二叉树的数组存储状态

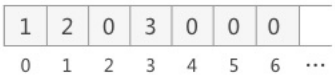
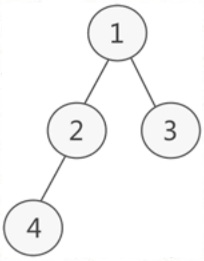


图 4 普通二叉树的存储状态

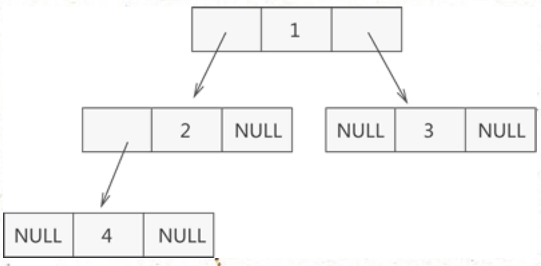
不仅如此，从顺序表中还原完全二叉树也很简单。我们知道，完全二叉树具有这样的性质，将树中节点按照层次并从左到右依次标号（1,2,3,...），若节点 i 有左右孩子，则其左孩子节点为 2*i，右孩子节点为 2*i+1。此性质可用于还原数组中存储的完全二叉树。

二叉树的链式存储结构:

其实二叉树并不适合用数组存储，因为并不是每个二叉树都是完全二叉树，普通二叉树使用顺序表存储或多或多会存在空间浪费的现象。



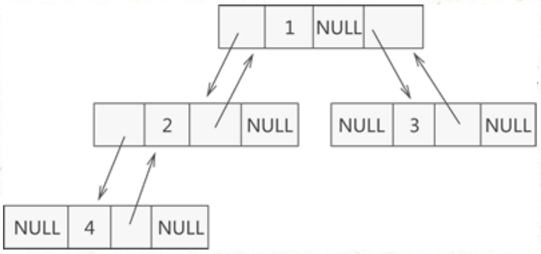
此为一棵普通的二叉树，若将其采用链式存储，则只需从树的根节点开始，将各个节点及其左右孩子使用链表存储即可。



采用链式存储二叉树时，其节点结构由 3 部分构成

- 指向左孩子节点的指针（Lchild）；
- 节点存储的数据（data）；
- 指向右孩子节点的指针（Rchild）；

在某些实际场景中，可能会做 "查找某节点的父节点" 的操作，这时可以在节点结构中再添加一个指针域，用于各个节点指向其父亲节点



- 这样的链表结构，通常称为三叉链表。

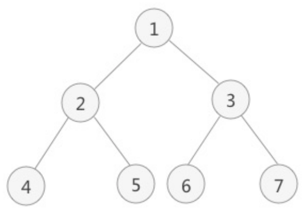
广义表创建二叉树:

```
//A(B(D,E(G,)),C(,F))#
function init_tree(str) {
  let item;
  let k = 0;
  let node = null;
  let root = null;
  const stack = new Stack();
  for (let i = 0; i < str.length; i++) {
    item = str[i];
     if (item === '#') {
       break;
     }
     if (item === '(') {
       k = 1;
       node && stack.push(node);
     } else if (item === ')') {
       if (!stack.isEmpty()) {
         stack.pop();
       }
     } else if (item === ',') {
       k = 2;
     } else {
       node = new Node(item);
       if (!root) {
         root = node;
         continue;
       }
       const parent_node = stack.top();
       if (k === 1) {
         parent_node.left_node = node;
       } else if (k === 2) {
         parent_node.right_node = node;
```

```
      }
      node.parent_node = parent_node;
    }
  }
  return root;
}
```

二叉树遍历:

二叉树的先序、中序和后序的遍历算法，运用了栈的数据结构，主要思想就是按照先左子树后右子树的顺序依次遍历树中各个结点



先序遍历:

二叉树先序遍历的实现思想是：

1. 访问根节点；
2. 访问当前节点的左子树；
3. 若当前节点无左子树，则访问当前节点的右子树；

遍历二叉树的过程:

1. 访问该二叉树的根节点，找到 1；
2. 访问节点 1 的左子树，找到节点 2；
3. 访问节点 2 的左子树，找到节点 4；
4. 由于访问节点 4 左子树失败，且也没有右子树，因此以节点 4 为根节点的子树遍历完成。但节点 2 还没有遍历其右子树，因此现在开始遍历，即访问节点 5；
5. 由于节点 5 无左右子树，因此节点 5 遍历完成，并且由此以节点 2 为根节点的子树也遍历完成。现在回到节点 1，并开始遍历该节点的右子树，即访问节点 3；
6. 访问节点 3 左子树，找到节点 6；
7. 由于节点 6 无左右子树，因此节点 6 遍历完成，回到节点 3 并遍历其右子树，找到节点 7；
8. 节点 7 无左右子树，因此以节点 3 为根节点的子树遍历完成，同时回归节点 1。由于节点 1 的左右子树全部遍历完成，因此整个二叉树遍历完成；

1.递归:

```
function in_order(node) {
  if (!node) {
    return null;
  }
  console.log(node.data);
  in_order(node.left_node);
  in_order(node.right_node);
}
```

2.非递归:

```
function in_order(node) {
  const stack = new Stack();
  let cur_node = node;
  while (cur_node) {
    if (cur_node.right_node) {
      stack.push(cur_node.right_node)
```

```
    }
    cur_node = cur_node.left_node;
    if (!cur_node && !stack.isEmpty()) {
      cur_node = stack.pop();
     }
  }
}
```

中序遍历:

二叉树中序遍历的实现思想是:

1. 访问当前节点的左子树;
2. 访问根节点;
3. 访问当前节点的右子树;

过程:

1. 访问该二叉树的根节点，找到 1;
2. 遍历节点 1 的左子树，找到节点 2;
3. 遍历节点 2 的左子树，找到节点 4;
4. 由于节点 4 无左孩子，因此找到节点 4，并遍历节点 4 的右子树;
5. 由于节点 4 无右子树，因此节点 2 的左子树遍历完成，访问节点 2;
6. 遍历节点 2 的右子树，找到节点 5;
7. 由于节点 5 无左子树，因此访问节点 5 ，又因为节点 5 没有右子树，因此节点 1 的左子树遍历完成，访问节点 1 ，并遍历节点 1 的右子树，找到节点 3;
8. 遍历节点 3 的左子树，找到节点 6;
9. 由于节点 6 无左子树，因此访问节点 6，又因为该节点无右子树，因此节点 3 的左子树遍历完成，开始访问节点 3 ，并遍历节点 3 的右子树，找到节点 7;
10. 由于节点 7 无左子树，因此访问节点 7，又因为该节点无右子树，因此节点 1 的右子树遍历完成，即整棵树遍历完成;

1.递归:

```
function infix_order(node) {
  if (!node) {
    return null;
  }
  infix_order(node.left_node);
   console.log(node.data);
  infix_order(node.right_node);
}
```

2.非递归:

```
function infix_order(node) {
  let cur_node = node;
  const stack = new Stack();
  while (true) {
    while (cur_node) {
      stack.push(cur_node);
      cur_node = cur_node.left_node;
    }
    const top_node = stack.pop();
     console.log(top_node.data);
     cur_node = top_node.right_node;
     if (!cur_node && stack.isEmpty()) {
       break;
     }
```

```
  }
}
```

后序遍历:

二叉树后序遍历的实现思想是：从根节点出发，依次遍历各节点的左右子树，直到当前节点左右子树遍历完成后，才访问该节点元素。

1. 对此二叉树进行后序遍历的操作过程为：
2. 从根节点 1 开始，遍历该节点的左子树（以节点 2 为根节点）；
3. 遍历节点 2 的左子树（以节点 4 为根节点）；
4. 由于节点 4 既没有左子树，也没有右子树，此时访问该节点中的元素 4，并回退到节点 2，遍历节点 2 的右子树（以 5 为根节点）；
5. 由于节点 5 无左右子树，因此可以访问节点 5，并且此时节点 2 的左右子树也遍历完成，因此也可以访问节点 2;
6. 此时回退到节点 1，开始遍历节点 1 的右子树（以节点 3 为根节点）；
7. 遍历节点 3 的左子树（以节点 6 为根节点）；
8. 由于节点 6 无左右子树，因此访问节点 6，并回退到节点 3，开始遍历节点 3 的右子树（以节点 7 为根节点）；
9. 由于节点 7 无左右子树，因此访问节点 7，并且节点 3 的左右子树也遍历完成，可以访问节点 3；节点 1 的左右子树也遍历完成，可以访问节点 1;

1.递归:

```
function epilogue(node) {
    if(!node) {
        return null;
    }
    epilogue(node.left_node);
    epilogue(node.right_node);
    console.log(node.data)
}
```

2.非递归:

```
function epilogue(node) {
  let cur_node = node;
  const stack = new Stack();
  while (true) {
    while (cur_node) {
      cur_node.isFinish = 0;
      stack.push(cur_node);
      cur_node = cur_node.left_node;
    }
     let top_node = stack.pop();
     let right_node = top_node.right_node;
     if (top_node.isFinish === 1 || !right_node) {
       top_node.isFinish = 1;
       console.log(top_node.data)
     } else {
       top_node.isFinish = 1;
       stack.push(top_node);
       cur_node = right_node;
     }
     if (!cur_node && stack.isEmpty()) {
       break;
     }
```

```
  }
}
```

层次遍历:

按照二叉树中的层次从左到右依次遍历每层中的结点。具体的实现思路是：通过使用队列的数据结构，从树的根结点开始，依次将其左孩子和右孩子入队。而后每次队列中一个结点出队，都将其左孩子和右孩子入队，直到树中所有结点都出队，出队结点的先后顺序就是层次遍历的最终结果。

层次遍历的实现过程:

1. 首先，根结点 1 入队；
2. 根结点 1 出队，出队的同时，将左孩子 2 和右孩子 3 分别入队；
3. 队头结点 2 出队，出队的同时，将结点 2 的左孩子 4 和右孩子 5 依次入队；
4. 队头结点 3 出队，出队的同时，将结点 3 的左孩子 6 和右孩子 7 依次入队；
5. 不断地循环，直至队列内为空。

```
function tier_order(node) {
  let cur_node = node;
  let line = '';
  const queue = new Queue();
  queue.push(cur_node);
  queue.push(0); // 0 为结束条件
  while (true) {
    const top_node = queue.pop();
    if (top_node === 0) {
      console.log(line);
      line = '';
      if (queue.isEmpty()) {
        break
      }
      queue.push(0);
      continue;
    }
    line += `${top_node.data} `;
    top_node.left_node && queue.push(top_node.left_node);
    top_node.right_node && queue.push(top_node.right_node)
  }
}
```