数据结构

夏天

xiat@ruc.edu.cn

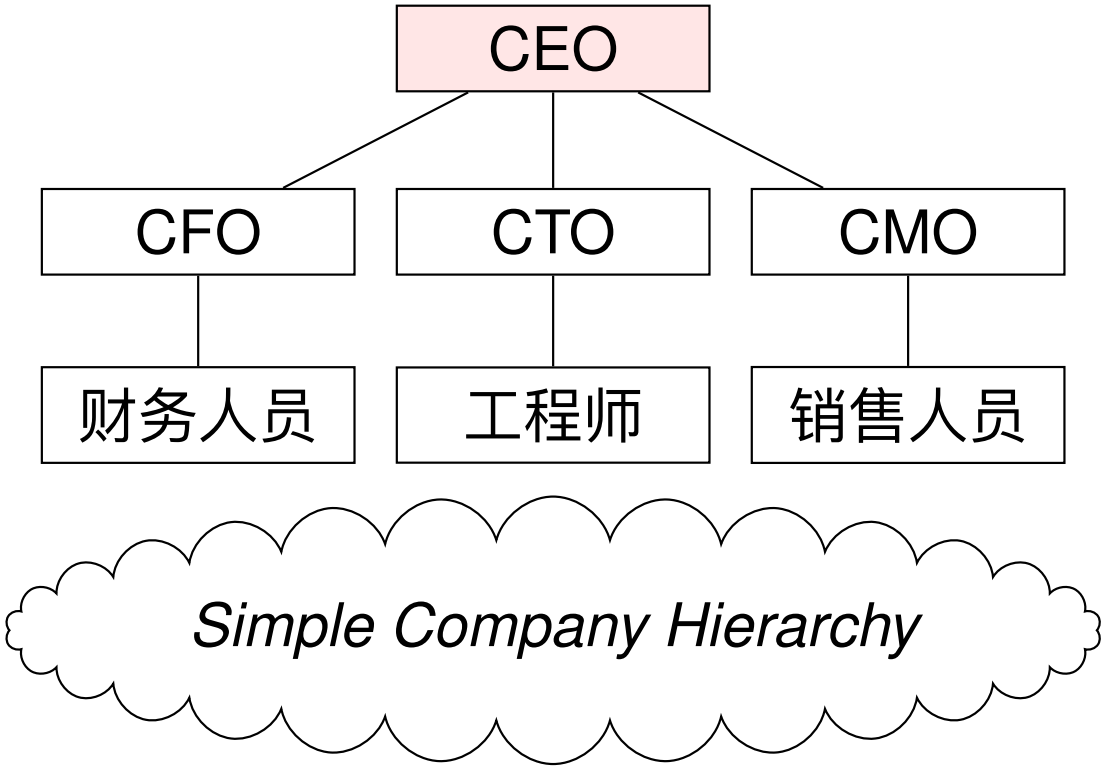
中国人民大学信息资源管理学院

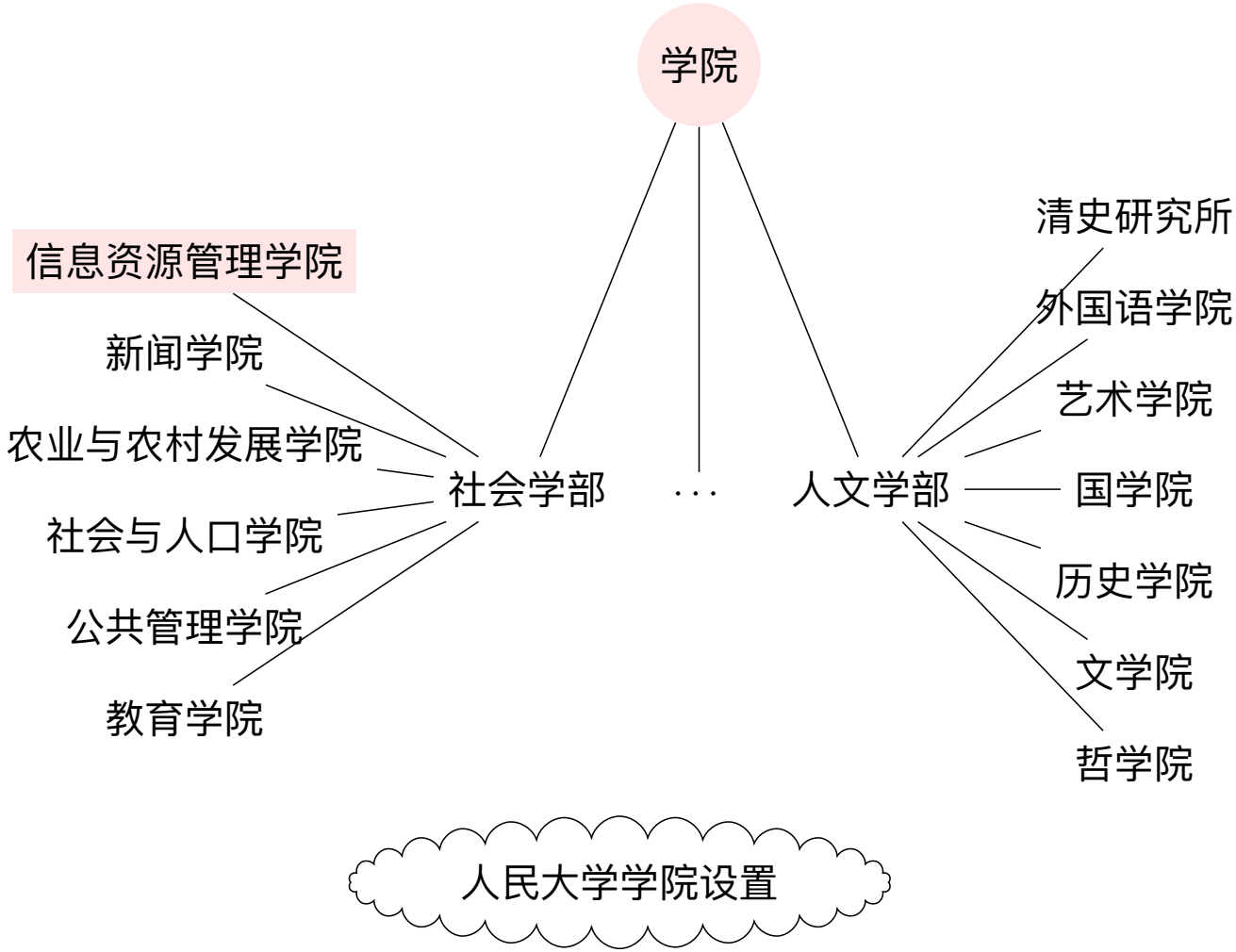
树和二叉树

树型结构是结点之间有分支, 并且具有层次关系的结构, 类似于自然界中的树。树有很多应用, 比如 Unix 等操作系统中的目录结构。

```
➜ github tree course_ds
course_ds
├── clean.py
├── dot
│   ├── tree-judge1.dot
│   ├── tree-judge1.pdf
│   ├── tree-judge2.dot
│   ├── tree-judge2.pdf
│   ├── tree-judge3.dot
│   ├── tree-judge3.pdf
│   ├── tree-judge4.dot
│   ├── tree-judge4.pdf
│   ├── tree-judge.pdf
│   ├── tree-represent1.dot
│   ├── tree-represent1.pdf
│   └── tree-term-demo.dot
├── ds.pdf
├── ds.tex
├── figs
│   └── search-block.tex
├── graph.tex
├── imgs
│   └── merge-sort.jpg
├── introduction.tex
├── LICENSE
├── _minted-ds
├── README.md
├── ruc_logo.png
```

例子




学院
信息资源管理学院
新闻学院
农业与农村发展学院
社会学部
社会与人口学院
公共管理学院
教育学院
· · ·
清史研究所
外国语学院
艺术学院
人文学部
国学院
历史学院
文学院
哲学院
人民大学学院设置

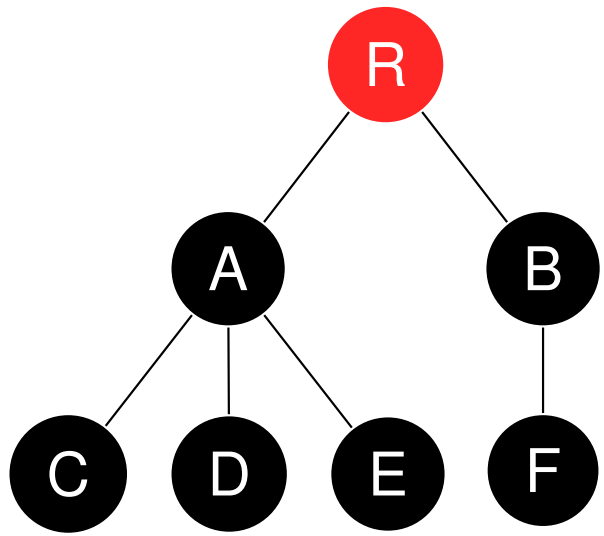
内容

- 树的基本术语
- 二叉树
- 遍历二叉树与线索二叉树
- 树和森林
- 哈夫曼树

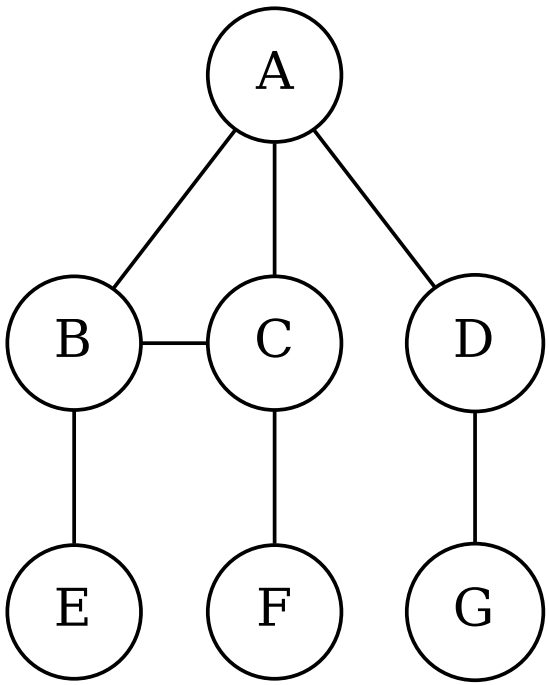
树 (TREE)

树 (Tree) 是 $\mathrm{n}(\mathrm{n} \geq 0)$ 个结点的有限集 T。T 为空时称为空树。当 $\mathrm{n} > 0$ 时, 树有且仅有一个特定的称为根 (Root) 的结点，其余结点可分为 $\mathrm{m}(\mathrm{m} \geq 0)$ 个互不相交的子集 $\mathrm{T}_1, \mathrm{T}_2, \cdots, \mathrm{T}_\mathrm{m}$，其中每个子集又是一棵树, 称为子树 (Subtree)。

- 各子树是互不相交的集合。
- 除根结点, 其它结点有唯一前驱。
- 一个结点可以有零个或多个后继。

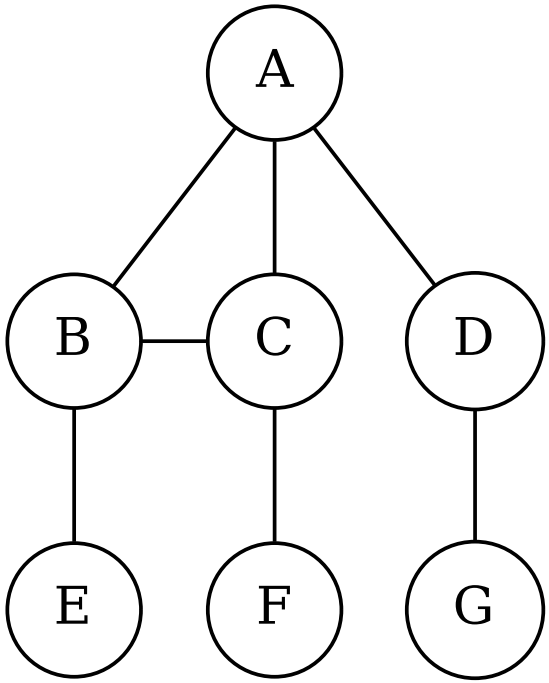


判断哪些是树结构

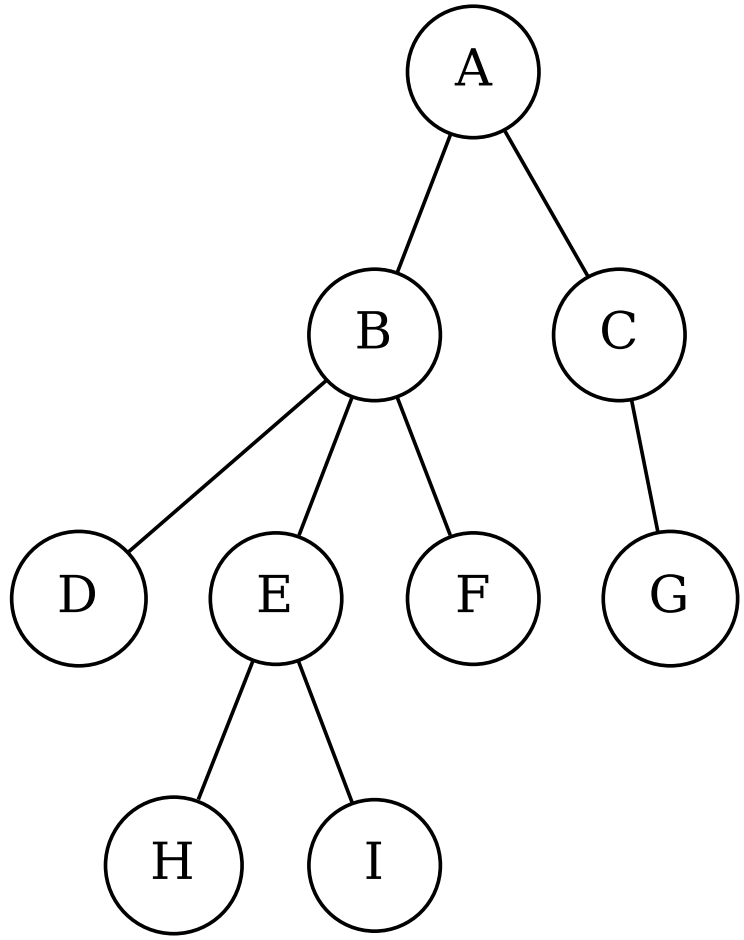


(1)

判断哪些是树结构

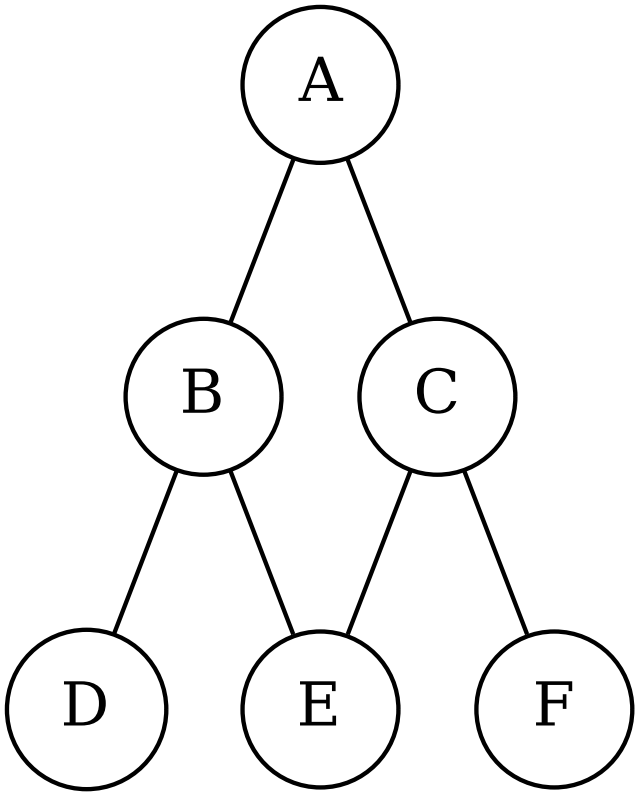


(1)



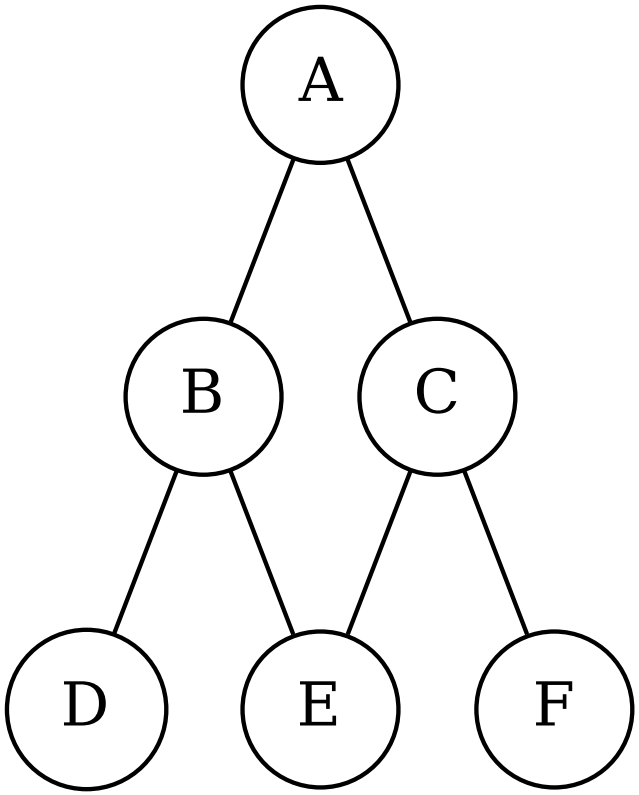
(2)

判断哪些是树结构

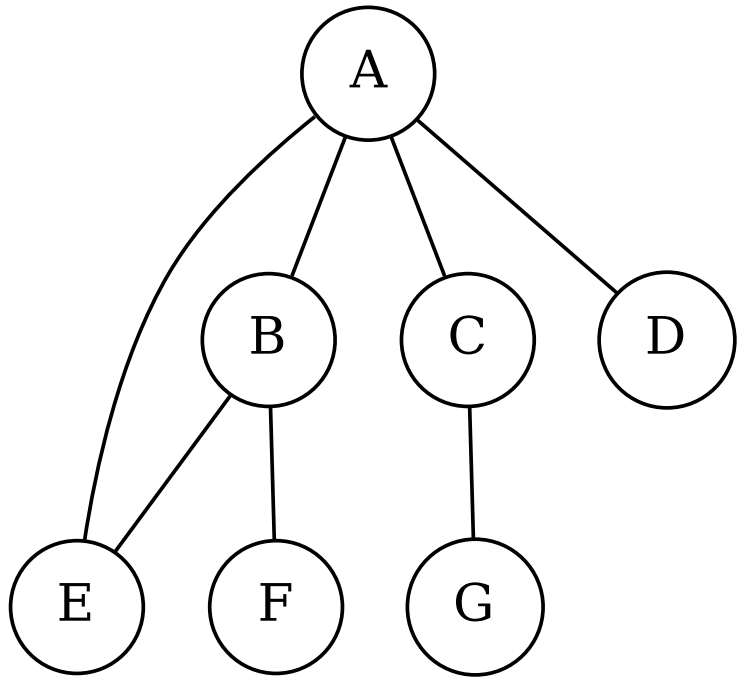


(3)

判断哪些是树结构

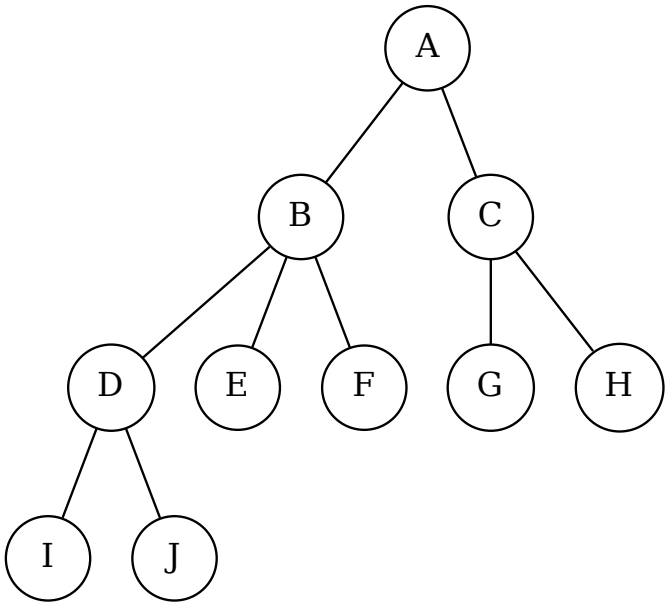


(3)



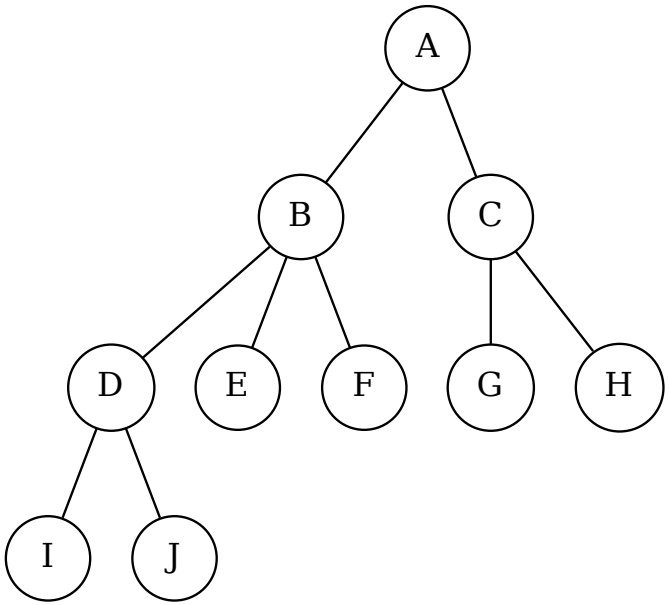
(4)

树的表示形式

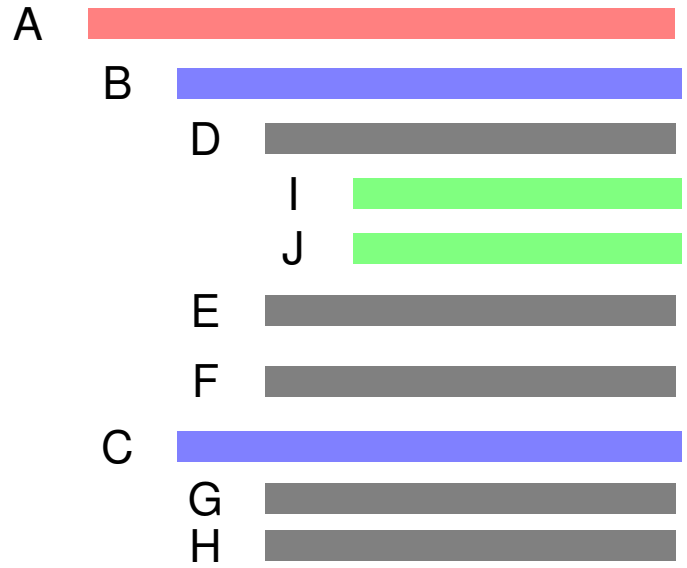


树形表示

树的表示形式

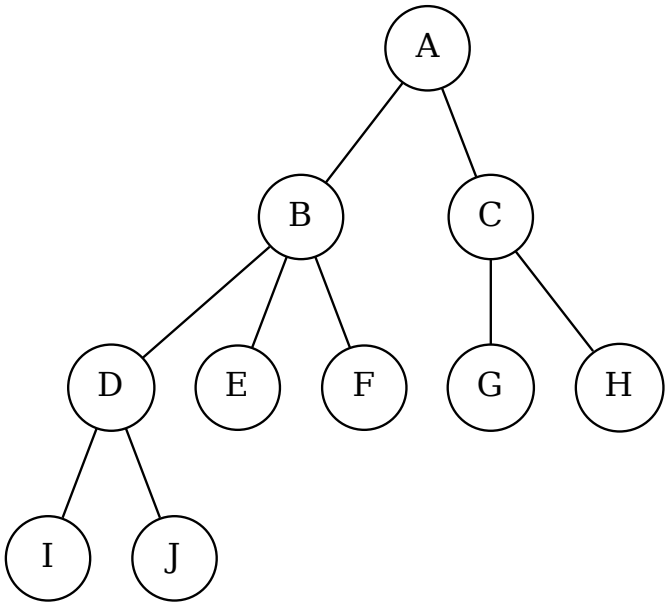


树形表示

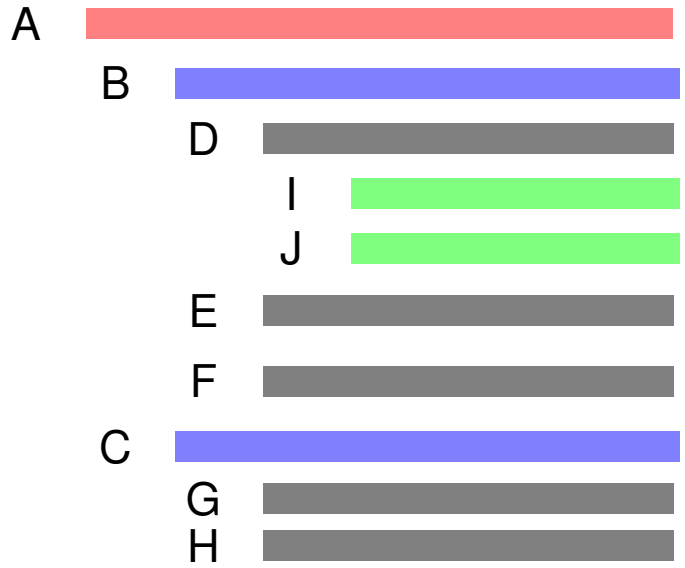


凹入表表示法

树的表示形式



树形表示

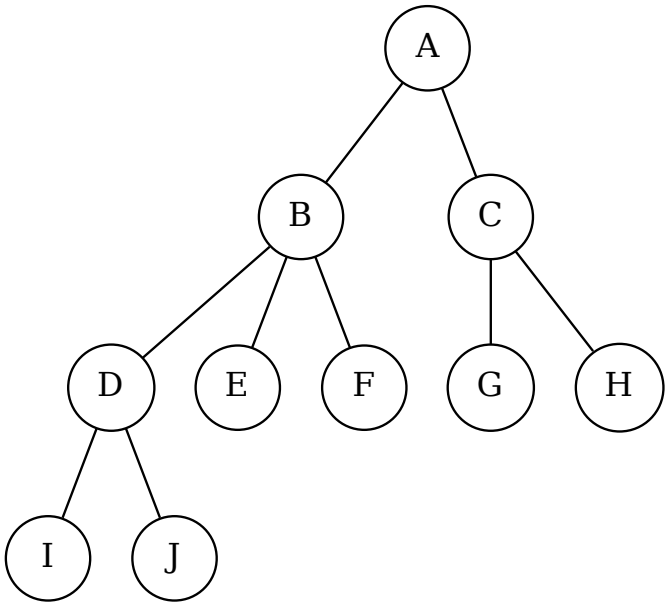


凹入表表示法

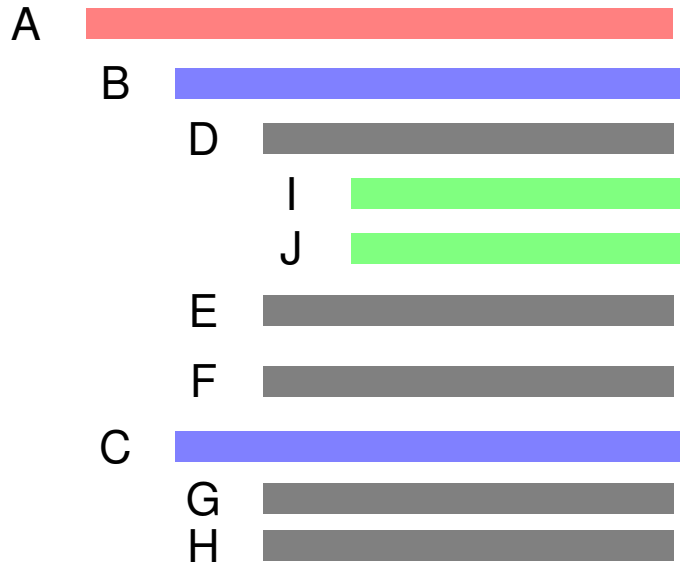
(A(B(D(I,J),E, F),C(G,H)))

广义表表示

树的表示形式



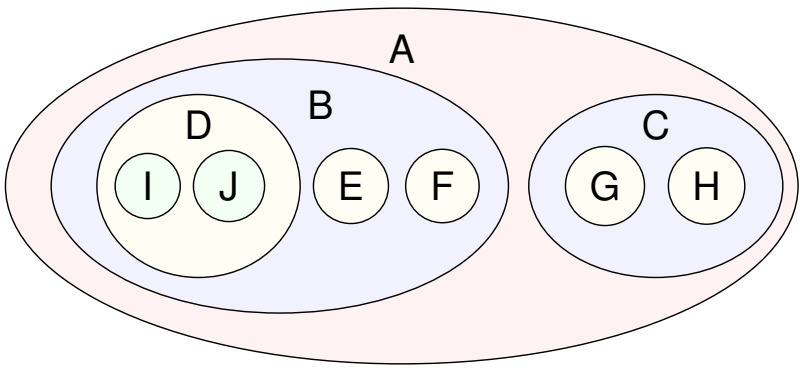
树形表示



凹入表表示法

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

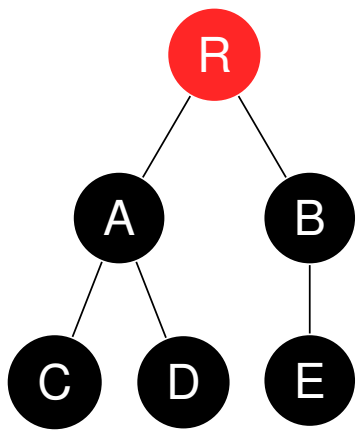
广义表表示



嵌套集合表示

基本术语

- 树 (tree)
- 子树 (sub-tree)
- 结点 (node)
- 结点的度 (degree)
- 叶子 (leaf)
- 孩子 (child)
- 父亲 (parents)
- 兄弟 (sibling)
- 祖先
- 子孙
- 树的度 (degree)
- 结点的层次 (level)
- 树的深度 (depth)
- 有序树
- 无序树
- 森林



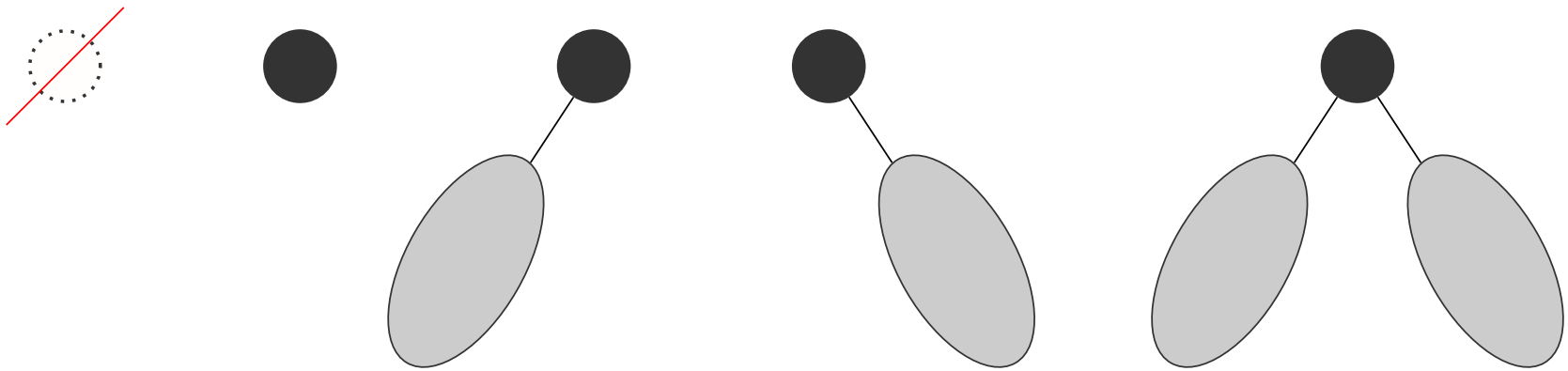
二叉树 (Binary Tree)

- 二叉树是一种树型结构, 它的每个结点至多只有两个子树, 分别称为左子树和右子树。二叉树是有序树。
- 二叉树是 $\mathrm{n}(\mathrm{n} \geq 0)$ 个结点构成的有限集合。二叉树或为空, 或是由一个根结点及两棵互不相交的左右子树组成, 并且左右子树都是二叉树。
- 在二叉树中要区分左子树和右子树, 即使只有一棵子树。这是二叉树与树的最主要的差别。

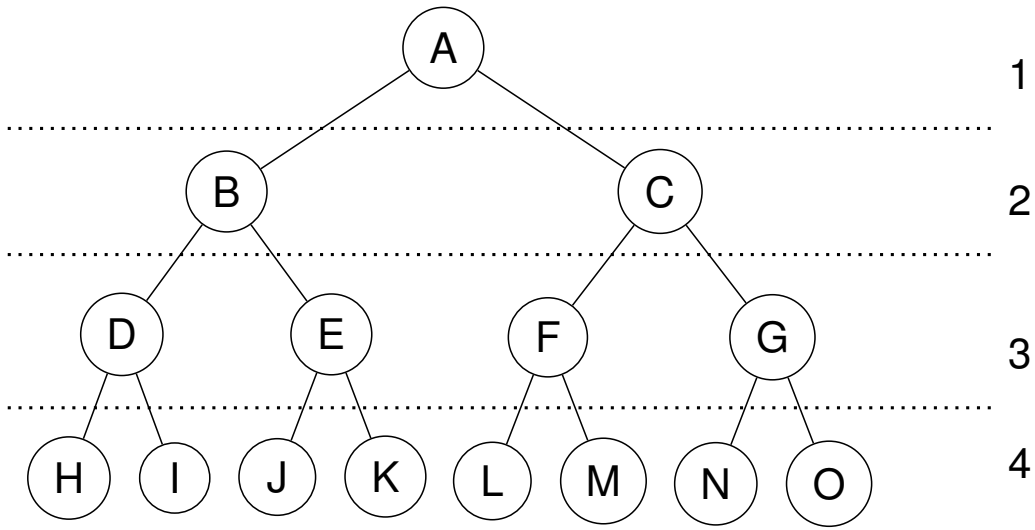
二叉树的一个重要应用是在查找中的应用。当然, 它还有许多与搜索无关的重要应用, 比如在编译器的设计领域。

二叉树的五种形态

1. 空二叉树;
2. 只有根结点 (左右子树都为空);
3. 只有左子树 (右子树为空);
4. 只有右子树 (左子树为空);
5. 左右子树均不空。



请观察二叉树, 并回答下列问题



1. 二叉树的第 i 层最多有多少个结点?
2. 二叉树深度为 k, 则它最多有多少个结点?
3. 二叉树有 n 个节点, 请问它最小深度是几?
4. 二叉树叶子的数目和度为 2 的节点的数目是否相等? 如果不等, 又是什么关系?

二叉树的性质

- 性质 1: 二叉树的第 i 层至多有 $2^{\mathrm{i}-1}$ 个结点。
- 性质 2: 深度为 k 的二叉树至多有 $2^{\mathrm{k}}-1$ 个结点 $(\mathrm{k} \geq 1)$。
- 性质 3: 二叉树中终端结点数为 n_0, 度为 2 的结点数为 n_2, 则有 $\mathrm{n}_0=\mathrm{n}_2+1$ (试证明)

二叉树的性质

二叉树中终端结点数为 n_0, 度为 2 的结点数为 n_2, 则有 $n_0 = n_2 + 1$

- 设二叉树中度为 1 的结点数为 n_1, 二叉树中总结点数为 N, 则有:

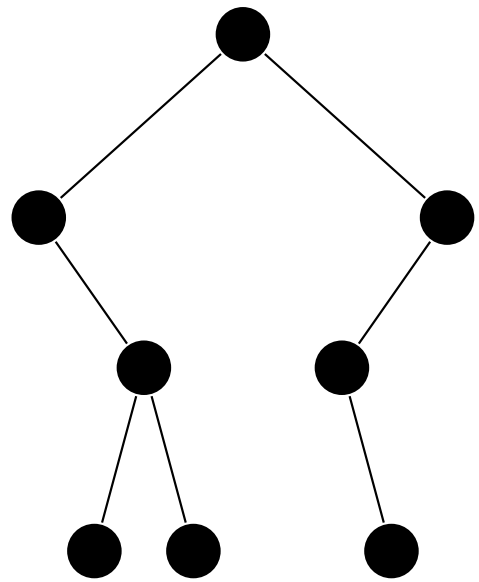
  $N = n_0 + n_1 + n_2$

- 再考虑二叉树中的分支数 (每个节点有唯一一个入的分支, 根节点除外; 再考虑出的分支数量), 则有:

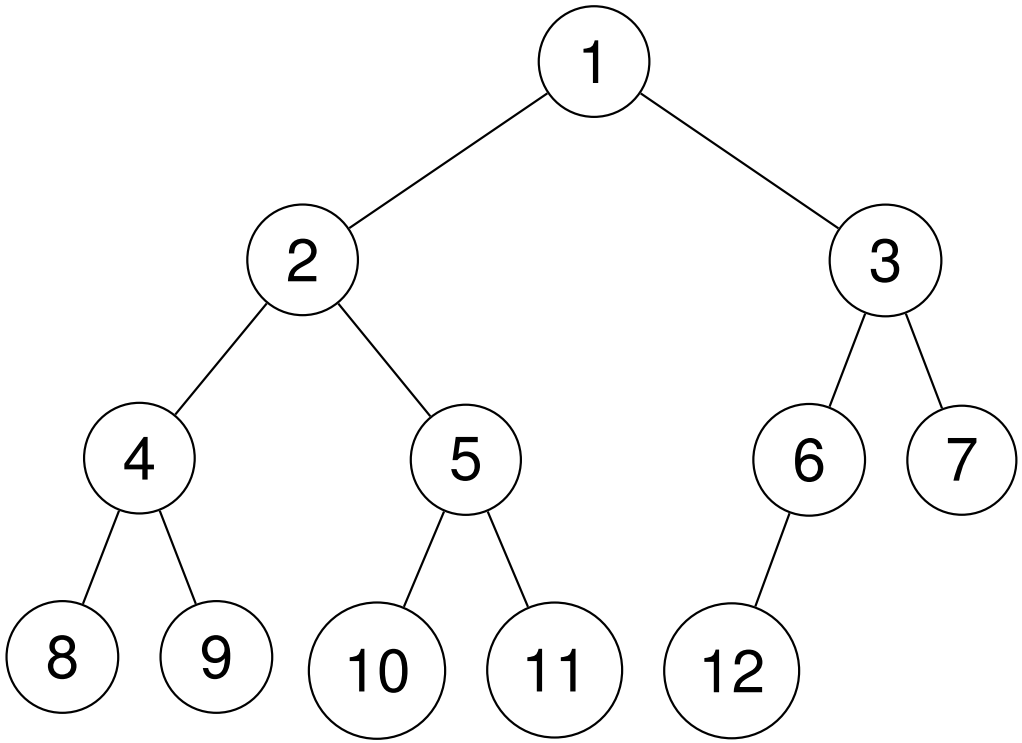
  $N - 1 = n_1 + 2 \times n_2$

- 整理可得:

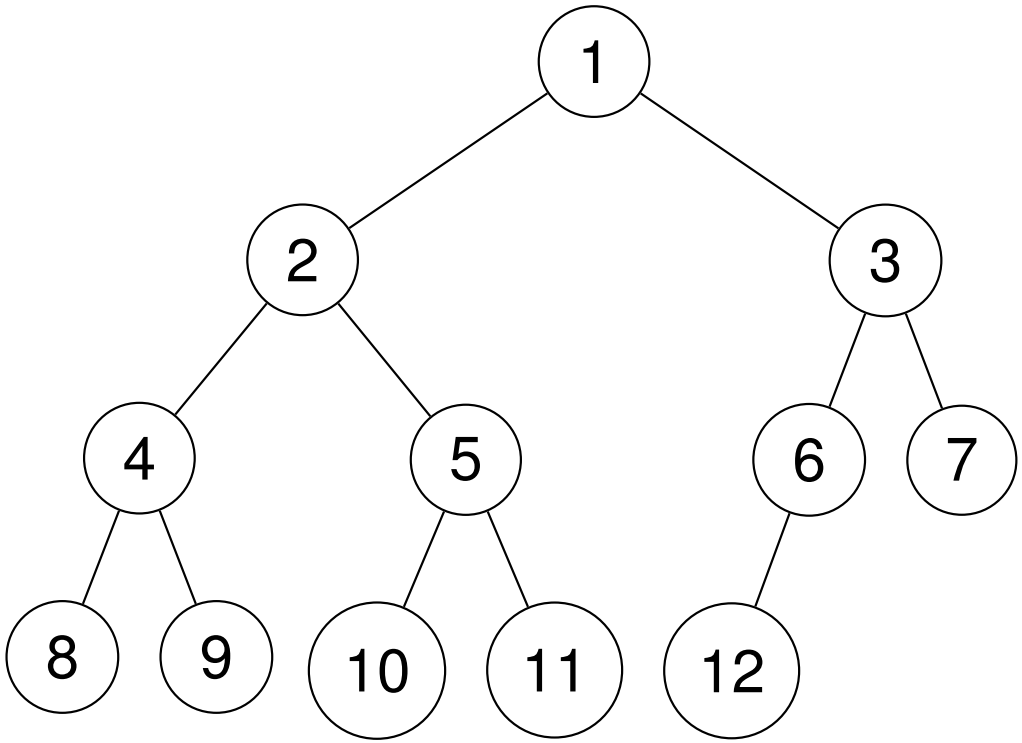
  $n_0 = n_2 + 1$



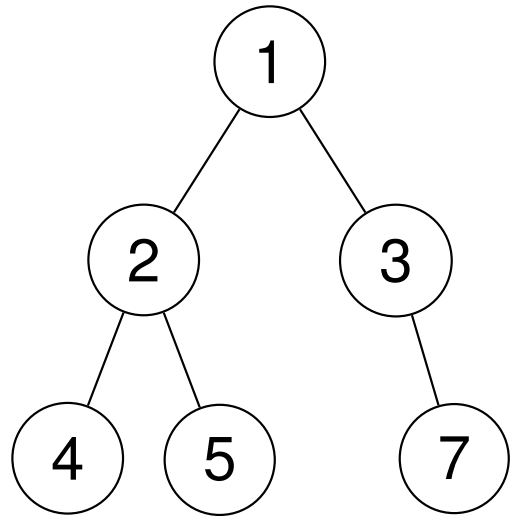
完全二叉树



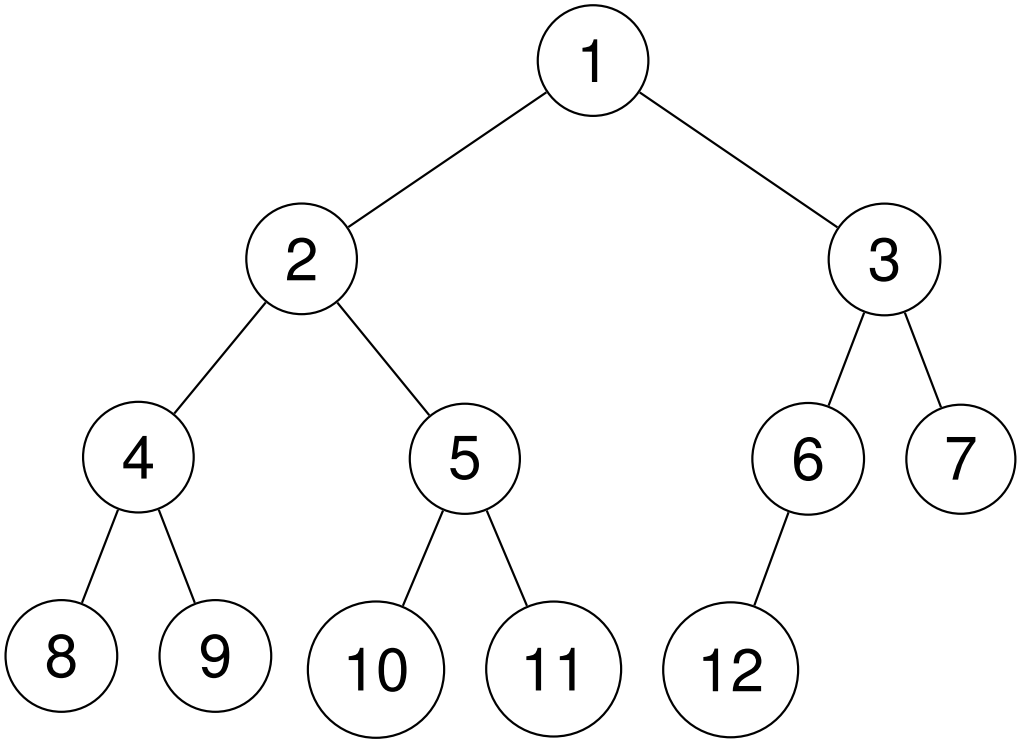
完全二叉树



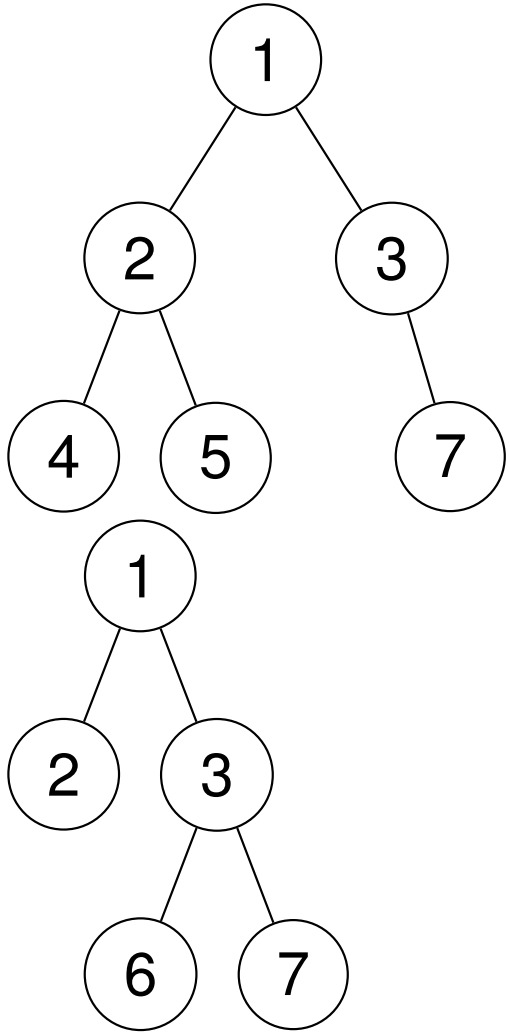
是否完全二叉树?



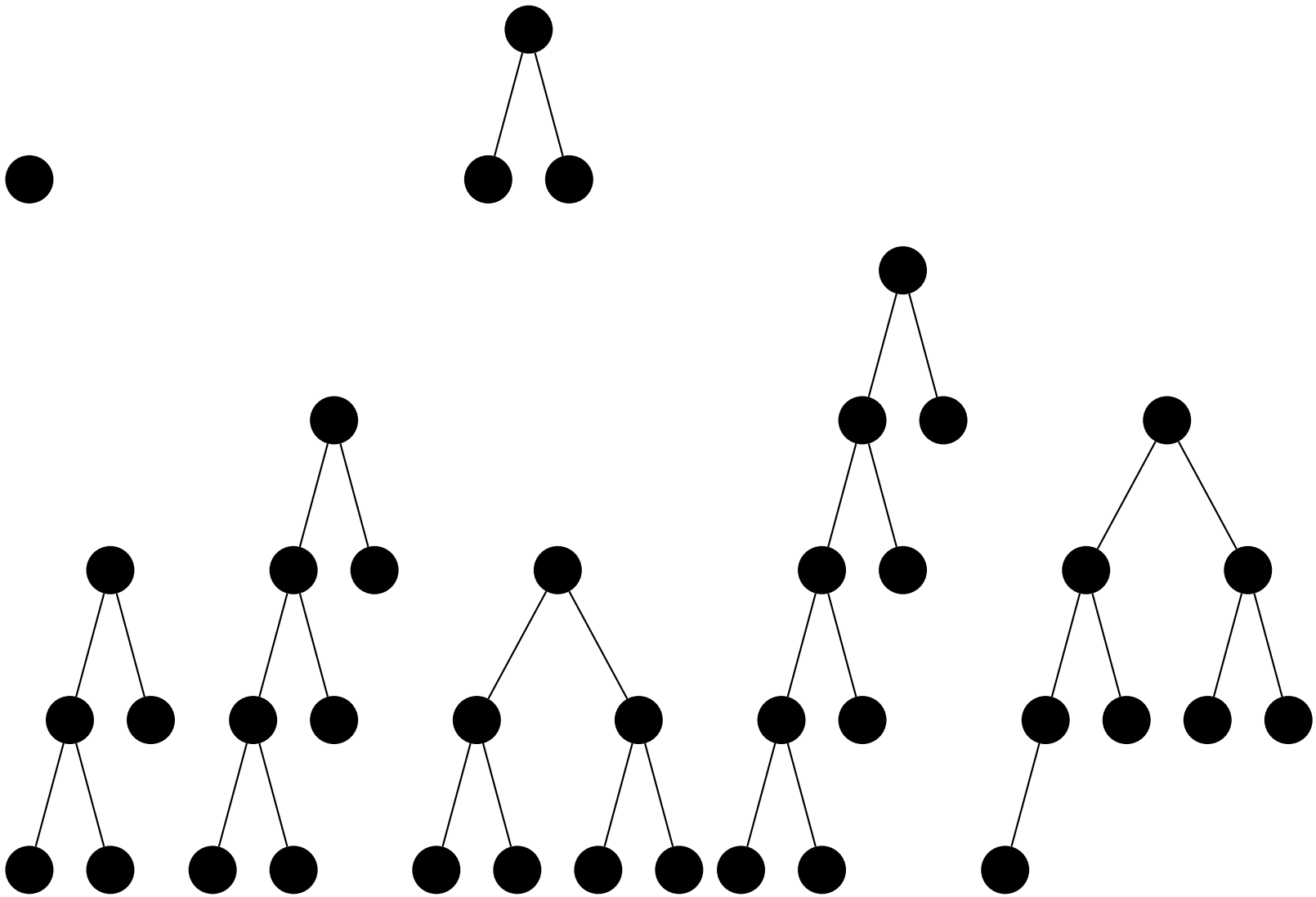
完全二叉树



是否完全二叉树?



试找出非完全二叉树

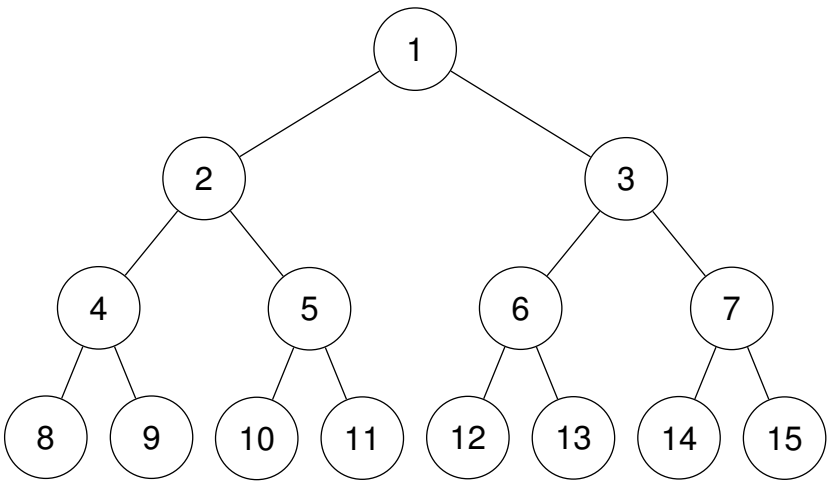


二叉树的性质

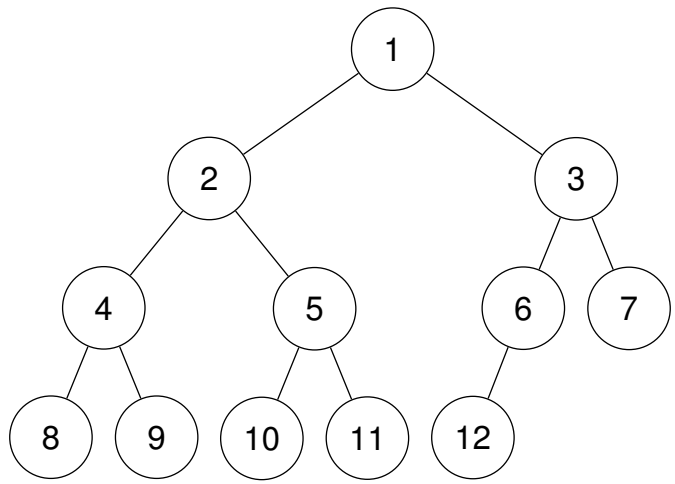
- 性质 4: 具有 n 个结点的完全二叉树的深度为:

$$\lfloor \log_2 n \rfloor + 1$$

对于完全二叉树，设深度为 k, 由 $2^{k-1} - 1 < n \leq 2^k - 1$ 可知, $2^{k-1} \leq n < 2^k$, 则 $k - 1 \leq \log_2 n < k$ (参考性质 2)



满二叉树

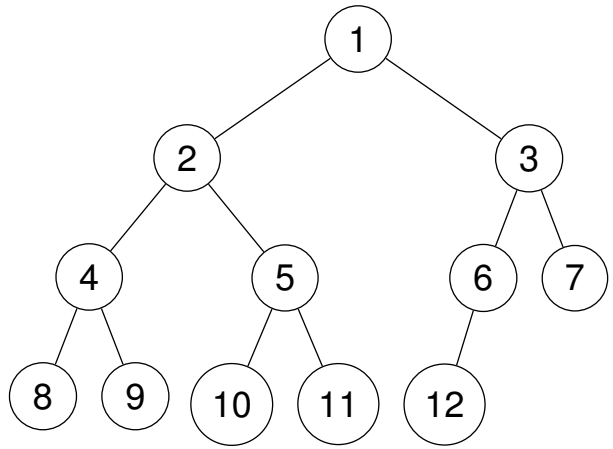


完全二叉树

测试

对于完全二叉树:

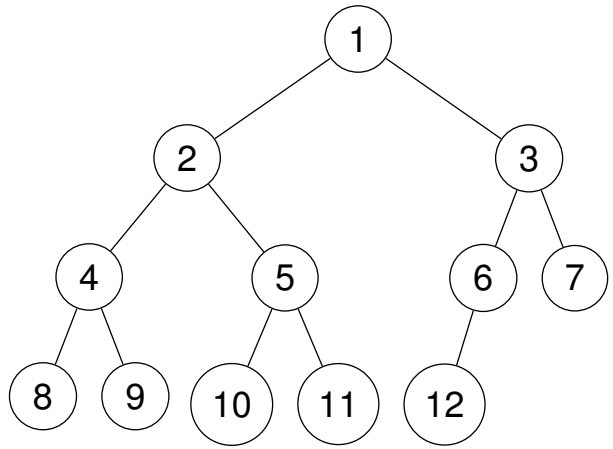
1. 若完全二叉树有叶子结点出现在第 k 层, 它可能还有 (　　) 层的叶子结点;
2. 若某结点的右子树的最大层次为 L, 则其左子树的最大层次为 (　　);
3. 若按如图所示的编号方式, 试求出编号为 i 的节点的父节点和子节点的编号.



测试

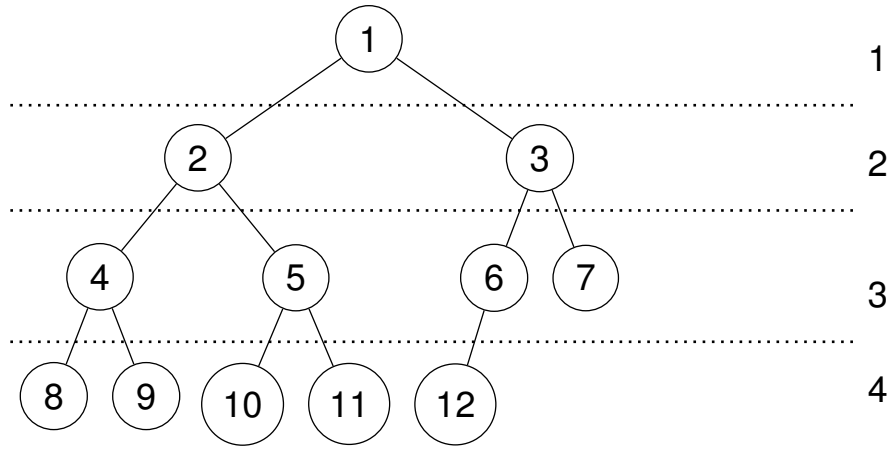
对于完全二叉树:

1. 若完全二叉树有叶子结点出现在第 k 层, 它可能还有 (　　) 层的叶子结点;
2. 若某结点的右子树的最大层次为 L, 则其左子树的最大层次为 (　　);
3. 若按如图所示的编号方式, 试求出编号为 i 的节点的父节点和子节点的编号.



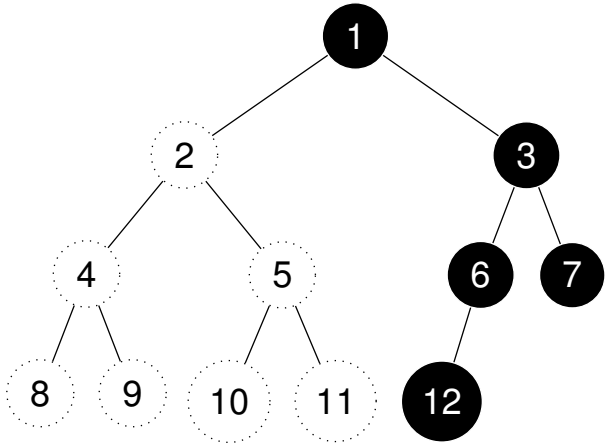
1. $k-1, k+1$
2. L or $L+1$

二叉树的性质



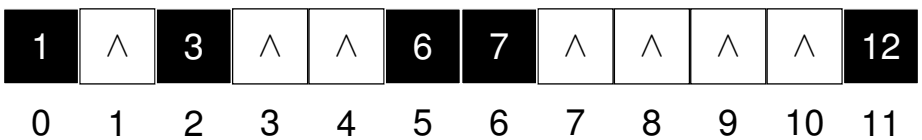
- 性质 5: 对具有 n 个结点的完全二叉树的结点按层次顺序编号, 对任意结点 i 有:
  - (有关结点 i 的双亲) 若 $i = 1$, 则为二叉树的根结点, 没有双亲; 否则双亲结点的编号为: $\left\lfloor \frac{i}{2} \right\rfloor$
  - (有关结点 i 的孩子) 若 $n < 2 \cdot i$, 则结点 i 无左孩子; 否则左孩子编号是 $2 \cdot i$。
  - 若 $n < 2 \cdot i + 1$, 则结点 i 无右孩子; 否则右孩子编号为 $2 \cdot i + 1$

二叉树的存储结构：顺序存储




```
#define MAX_SIZE 100
typedef int SqBiTree[MAX_SIZE];
SqBiTree bt;
```

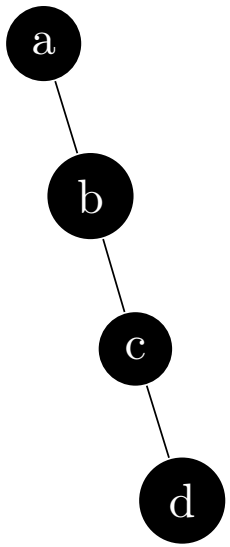
```
class SqBiTree {
  //static int MAX_SIZE = 100;
  //int[] data = new int[MAX_SIZE];
  List<Integer> data = new ArrayList<>()
}
```

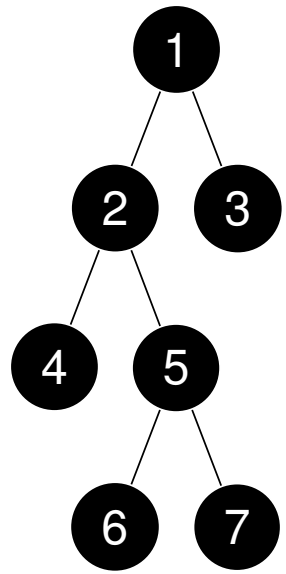


- 把结点安排成一个恰当的序列 (编号), 存储在数组中
- 便于“随机存取”

二叉树的存储结构：顺序存储

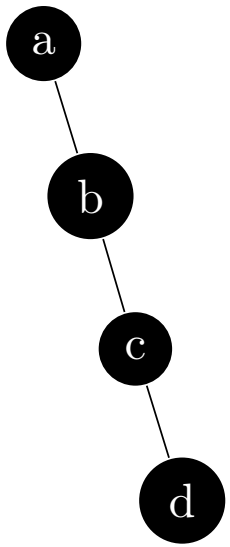
- 适用于完全二叉树

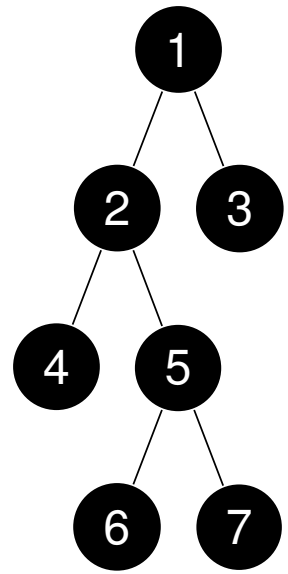




二叉树的存储结构：顺序存储

- 适用于完全二叉树

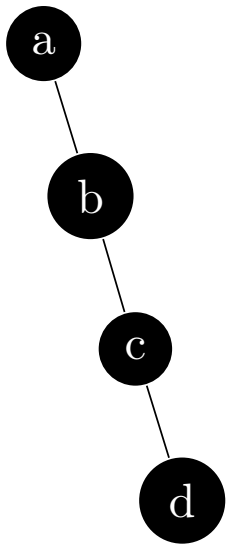


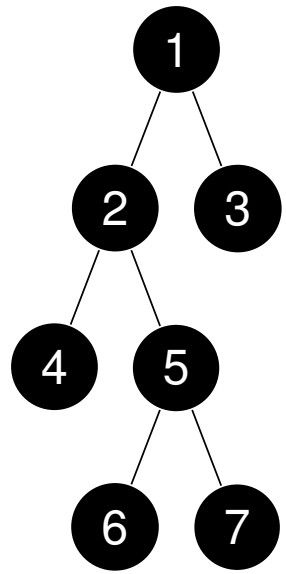


a	0	b	0	0	0	c	0	0	0	0	0	0	0	d
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

二叉树的存储结构：顺序存储

- 适用于完全二叉树

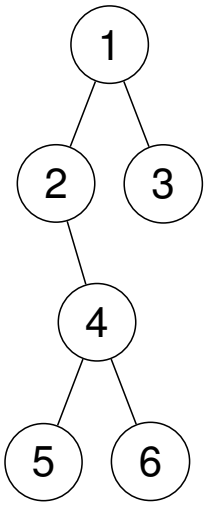


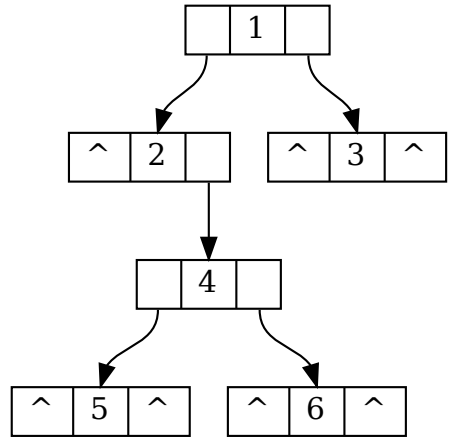


a	0	b	0	0	0	c	0	0	0	0	0	0	0	d
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

1	2	3	4	5	0	0	0	0	6	7
0	1	2	3	4	5	6	7	8	9	10

二叉树的链式存储: 二叉链表



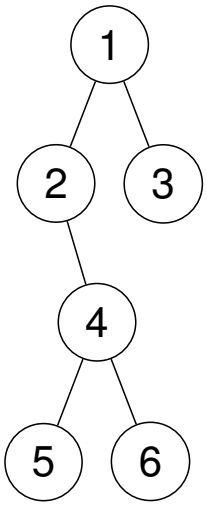


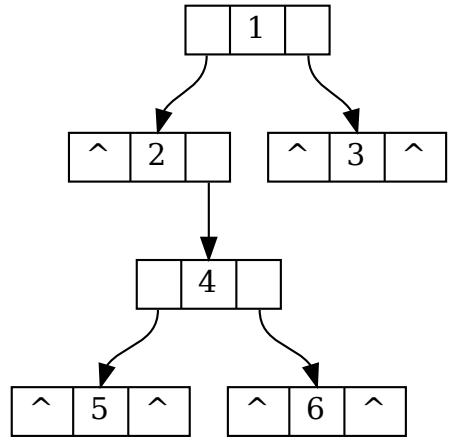

```c
typedef struct BiTNode  { // C Code
  ElemType data;
  struct BiTNode *lchild, *rchild;
} BiTNode, *BiTree;
```

```java
class BiTNode<T> { //Java Code
  T data;
  BiTNode lchild, rchild;
}
```

思考: 含 n 个结点的二叉链表中有多少个空指针?

二叉树的链式存储: 二叉链表






```c
typedef struct BiTNode  { // C Code
  ElemType data;
  struct BiTNode *lchild, *rchild;
} BiTNode, *BiTree;
```

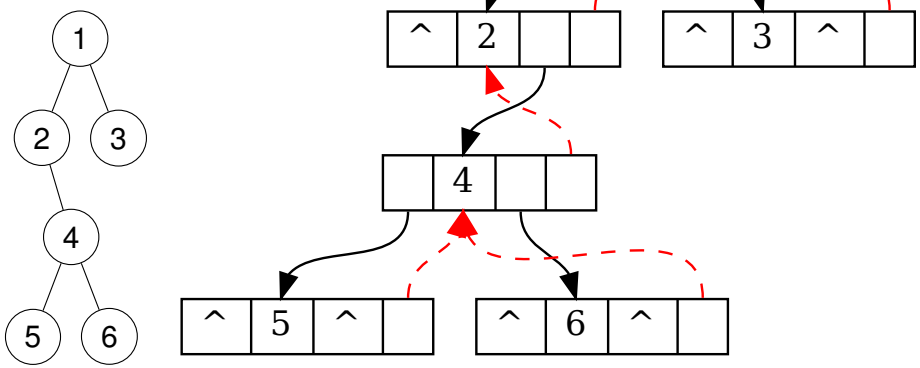
```java
class BiTNode<T> { //Java Code
  T data;
  BiTNode lchild, rchild;
}
```

思考: 含 n 个结点的二叉链表中有多少个空指针?

指针域一共有 $2 * n$ 个，分支共有 $N - 1$, 每个分支占用一个指针域，所以空指针数量为: $2 * n - (n - 1) = n + 1$

二叉树的链式存储: 三叉链表

- 二叉链表不便查找父节点, 可加一个指向双亲的指针



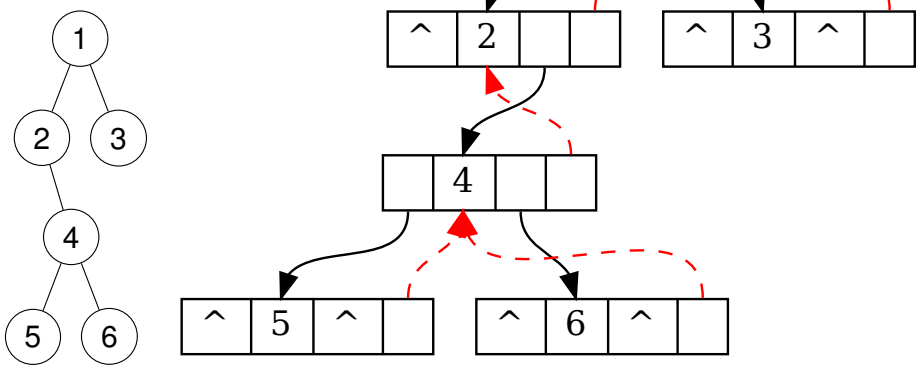

```
typedef struct BiTNode  { // C Code
  ElemType data;
  struct BiTNode *lchild, *rchild;
  struct BiTNode *parent;
} BiTNode, *BiTree;
```

```
class BiTNode<T> { //Java Code
  T data;
  BiTNode lchild, rchild, parent;
}
```

思考: 有 n 个结点的三叉链表有多少个空指针?

二叉树的链式存储: 三叉链表

- 二叉链表不便查找父节点, 可加一个指向双亲的指针




```c
typedef struct BiTNode  { // C Code
  ElemType data;
  struct BiTNode *lchild, *rchild;
  struct BiTNode *parent;
} BiTNode, *BiTree;
```

```java
class BiTNode<T> { //Java Code
  T data;
  BiTNode lchild, rchild, parent;
}
```

思考: 有 n 个结点的三叉链表有多少个空指针?

对于二叉链表，指针域一共有 $2 * n$ 个，分支共有 $N - 1$, 每个分支占用一个指针域，所以空指

遍历二叉树和线索二叉树

在二叉树的一些应用中, 常常要求在树中查找具有某种特征的结点, 或者对树中全部结点逐一进行某种处理。这就引入了遍历二叉树的问题, 即如何按某条搜索路径巡访树中的每一个结点, 使得每一个结点均被访问一次且仅访问一次。

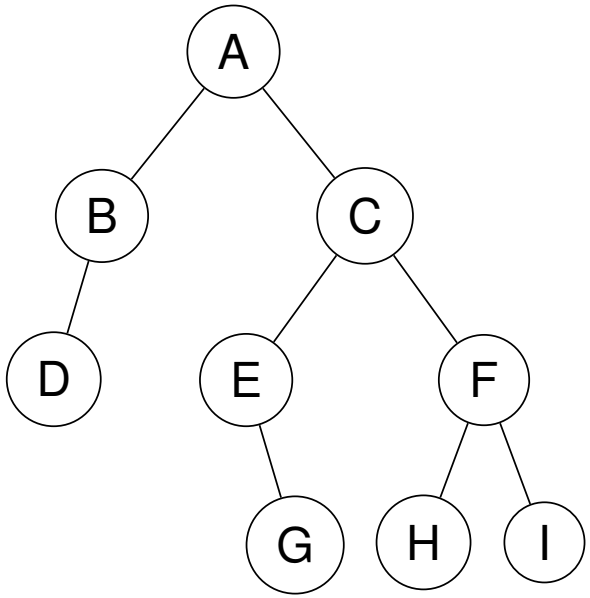
二叉树的遍历

- 遍历是指按某种方式访问所有结点, 使每个结点被访问一次且只被访问一次。
- 二叉树的遍历是按一定规则将二叉树的结点排成一个线性序列, 即非线性序列线性化。
- 遍历的方式: 深度优先和广度优先，深度优先又分为三种：
  - 先序次序
  - 中序次序 (对称序次序)
  - 后序次序

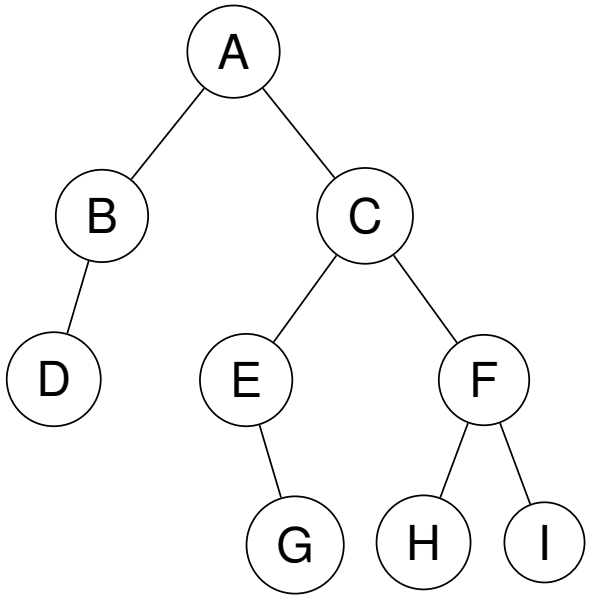
二叉树的遍历

- 1、先序遍历二叉树的操作定义为: 若二叉树为空, 则空操作; 否则
  - (1) 访问根结点;
  - (2) 先序遍历左子树;
  - (3) 先序遍历右子树。
- 2、中序遍历二叉树的操作定义为: 若二叉树为空, 则空操作; 否则
  - (1) 中序遍历左子树;
  - (2) 访问根结点;
  - (3) 中序遍历右子树。
- 3、后序遍历二叉树的操作定义为: 若二叉树为空, 则空操作; 否则

示例



示例

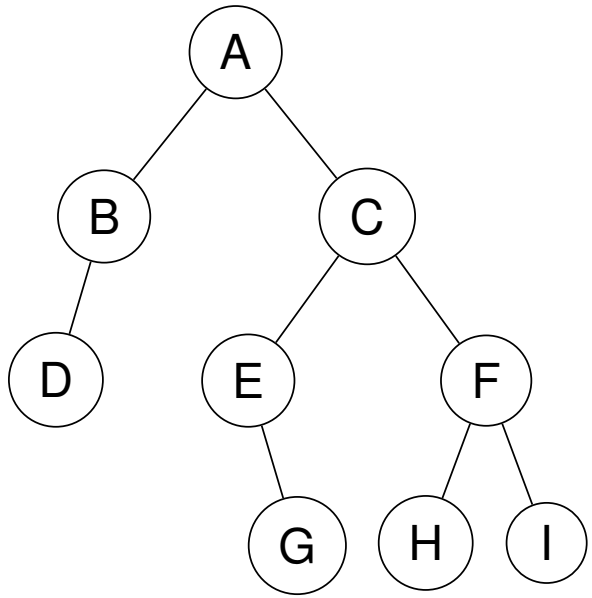


- 先序:ABDCEGFHI
- 中序:DBAEGCHFI
- 后序:DBGEHIFCA
- 广度优先：ABCDEFGHI

二叉树的遍历

//先序遍历二叉树递归算法 C 伪代码

```
status preOrderTraverse(BiTree T){
  if(T){
    printf(T->data);
    preOrderTraverse(T->lchild);
    preOrderTraverse(T->rchild);
  }
}
```

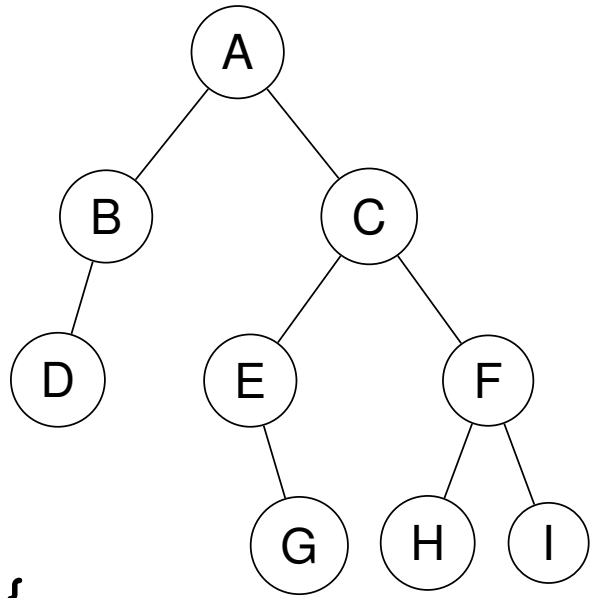


- status 代表什么?
- printf 代表什么?

二叉树的遍历

先序遍历递归算法的 Java 实现

```
class Node {
  String data;
  Node left, right;
}
class Tree {
  void preOrderTraverse(Node node) {
    if(node != null) {
      System.out.println(node.data);
      preOrderTraverse(node.left);
      preOrderTraverse(node.right);
    }
  }
}
```



中序遍历的 Java 实现 I

```java
public class Tree {
  Node root;

  public Tree(Node root) {
    this.root = root;
  }

  void inOrderTraverse() {
    inOrderTraverse(root);
  }

  void inOrderTraverse(Node node) {
    if(node != null) {
      inOrderTraverse(node.lc);
      //visit(node);
      System.out.println(node);
      inOrderTraverse(node.rc);
```

中序遍历的 Java 实现 II

```
    }
}

public static void main(String[] args) {
    Node a  =new Node("A",
    new Node("B", new Node("D"), null),
    new Node("C", new Node("E"), new Node("F"))
    );
    Tree tree = new Tree(a);
    tree.inOrderTraverse();
}

static class Node {
    String data;
    Node lc, rc;

    public Node(String data, Node lc, Node rc) {
```

中序遍历的 Java 实现 III

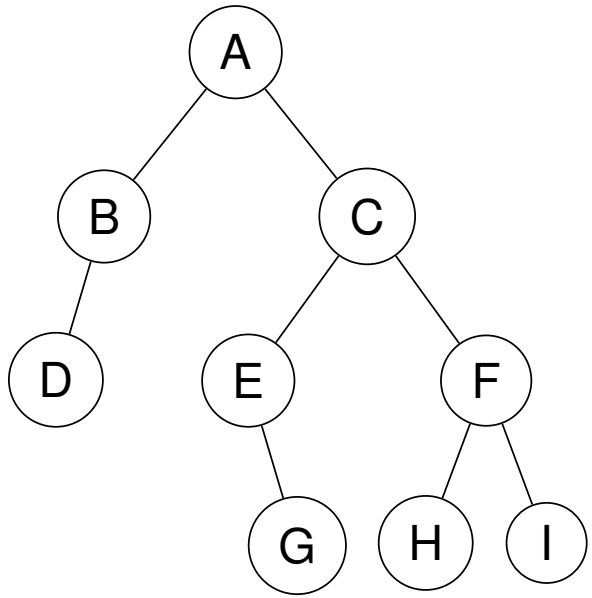
```
      this.data = data;
      this.lc = lc;
      this.rc = rc;
    }

    public Node(String data) {
      this.data = data;
      this.lc = null;
      this.rc = null;
    }

    public String toString(){
      return data;
    }
  }
}
```

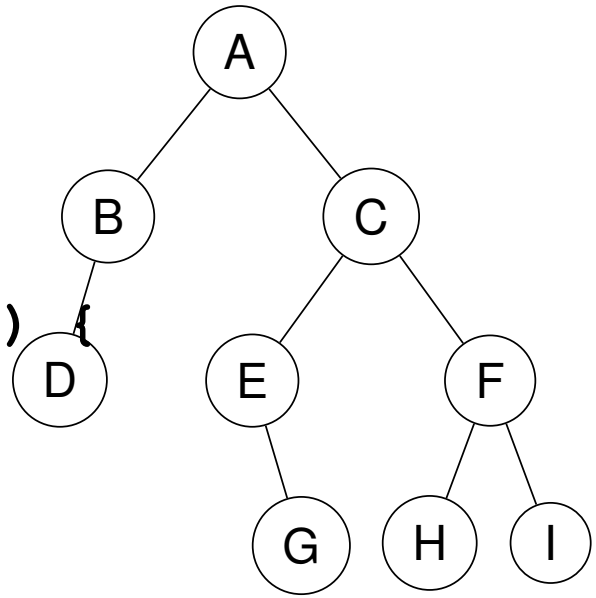
如下将得到树的什么序列?

```
status traverse(BiTree T) {
  InitStack(S);
  p = T;
  while(p || !StackEmpty(S)) {
    if(p) {
      push(S, p); p = p->lchild;
    } else {
      pop(S, p);
      printf(p->data);
      p = p->rchild;
    }
  }
  return OK;
}
```



如下将得到树的什么序列?

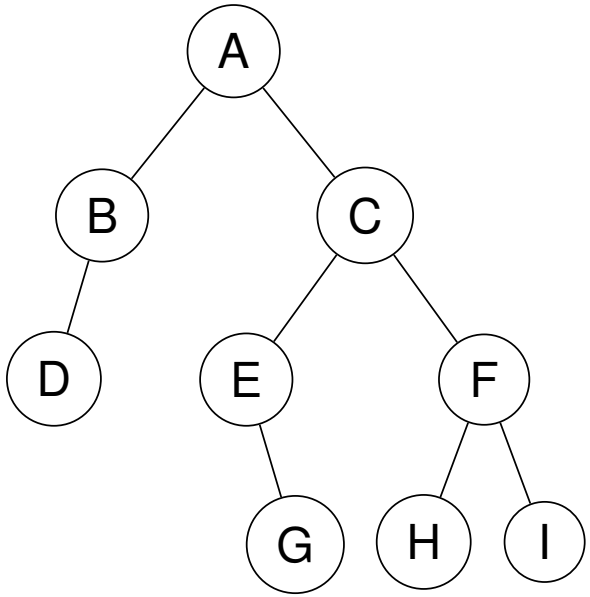
```
void traverse(TreeNode root) {
  Stack stack = new Stack();
  TreeNode p = root;
  while(p!=null || stack.notEmpty()) {
    if(p!=null) {
      stack.push(p);
      p = p.lchild;
    } else {
      p = stack.pop();
      printf(p.data);
      p = p.rchild;
    }
  }
}
```



二叉树的遍历: 广度优先

算法步骤:

- 访问节点
- 从左到右依次访问儿子节点
- 重复前一步骤



A						
	B	C				
		C	D			

.....

课堂练习：编程实现广度优先遍历。

广度优先遍历

```
void bfs(TreeNode root) {
  Queue Q = new Queue();
  Q.enqueue(root);
  while(Q.notEmpty()) {
    TreeNode node = Q.dequeue();
    printf(node);
    if(node.left!=null) Q.enqueue(node.left);
    if(node.right !=null) Q.enqueue(node.right);
  }
}
```

补充

- 由一棵给定的二叉树可以获得三种遍历序列, 同样, 也可以由这些遍历序列来重新构造二叉树。
- 举例

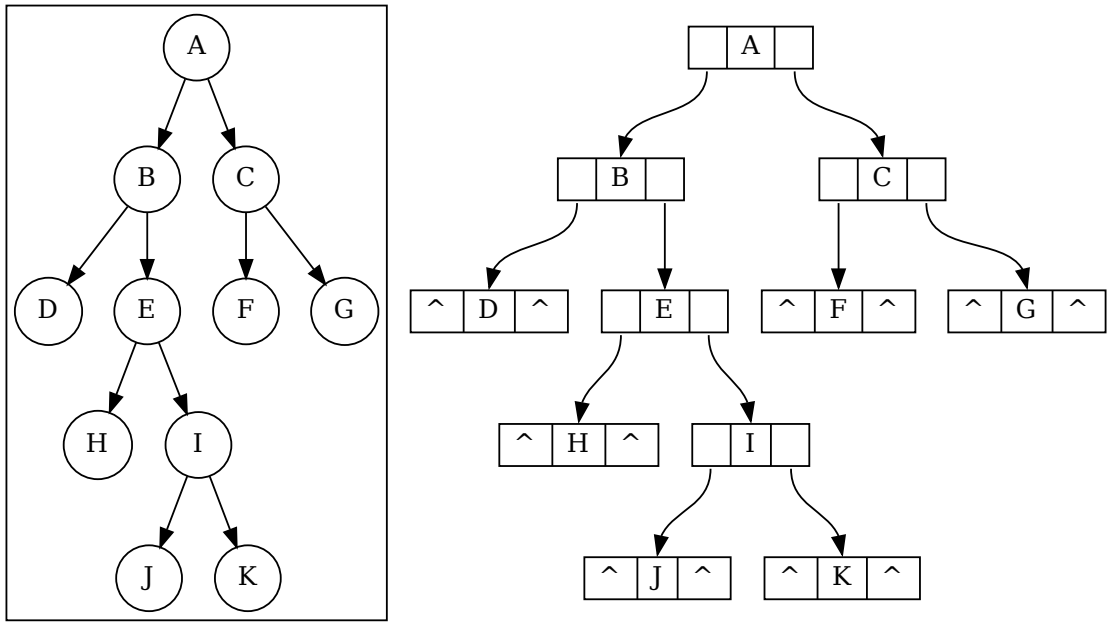
  先序:ABCDEFGHIJ

  中序:CBDEAFHIGJ
- 由于不能从遍历序列中区分二叉树的左、右子树, 因此单用一个遍历序列是无法构造二叉树的。利用中序遍历序列, 并结合先序遍历序列或后序遍历序列就能重新构造二叉树。

二叉树的线索化

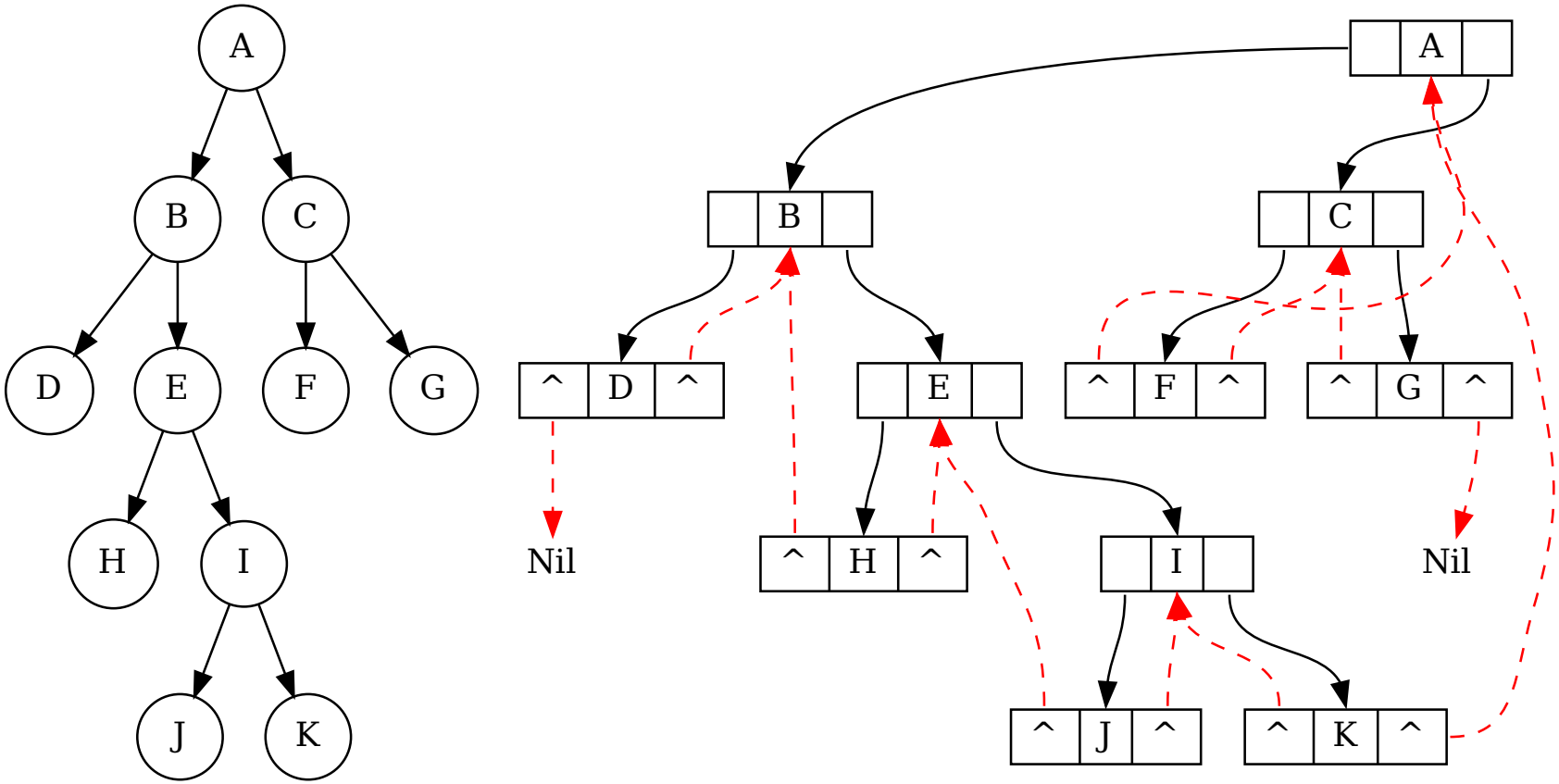
- 基于二叉链表可以很方便地查找节点的左、右孩子。
- 问题: 如何快速查找给定结点在遍历所得线性序列中的前驱和后继。
- 该信息在遍历的动态过程中才能得到。如果经常需要查找前驱和后继, 需要在二叉链表的结点上添加指向前驱和后继的指针, 叫做线索。这样得到的二叉树称为线索二叉树。

举例: 中序线索二叉树



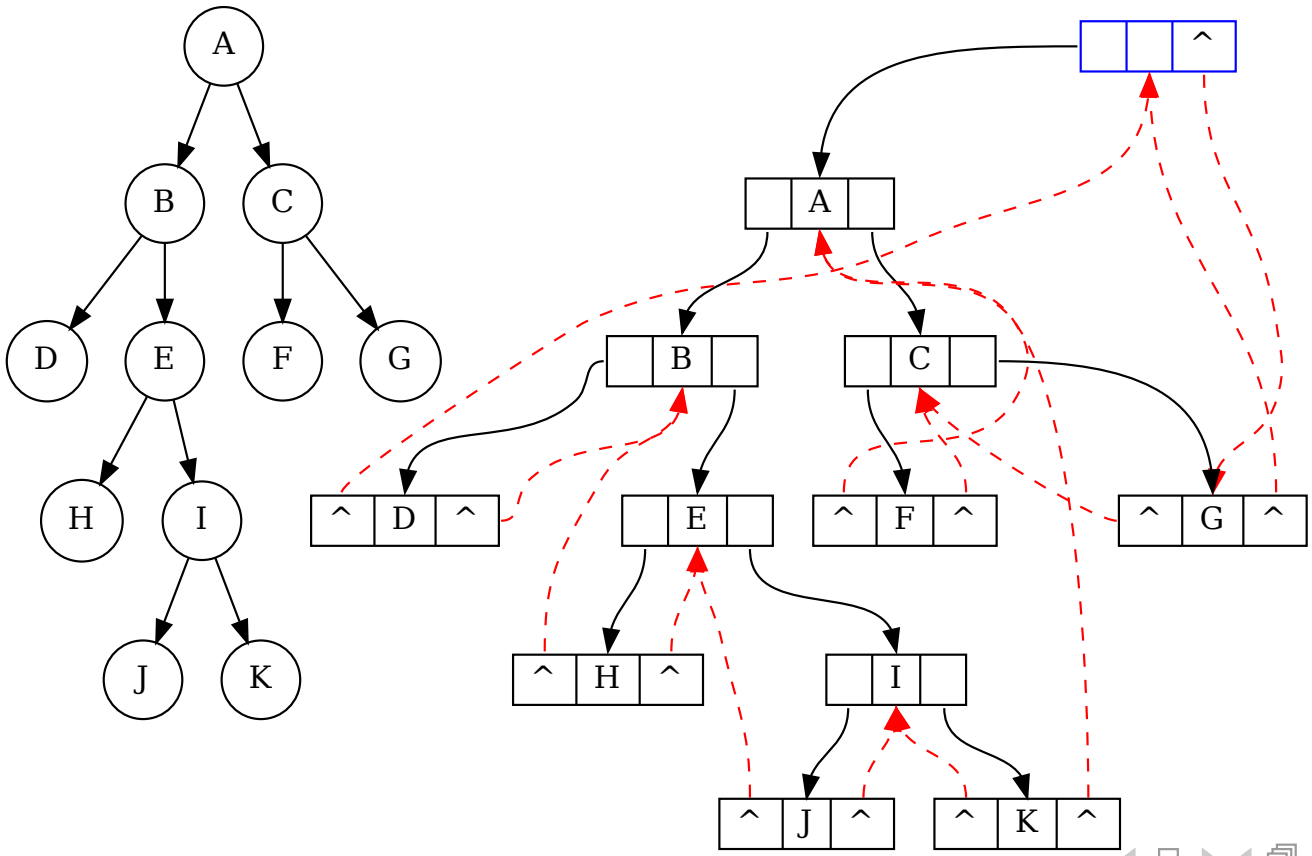
- 以右图二叉树为例, 中序序列为 (　　).
- 线索的表示: 若结点有左子树, 则指向其左孩子, 否则指向其前驱; 若节点有右子树, 则指向其右孩子, 否则指向其后继。
- 请问结点 A 和 F 的前驱和后继分别是?

举例: 中序线索二叉树



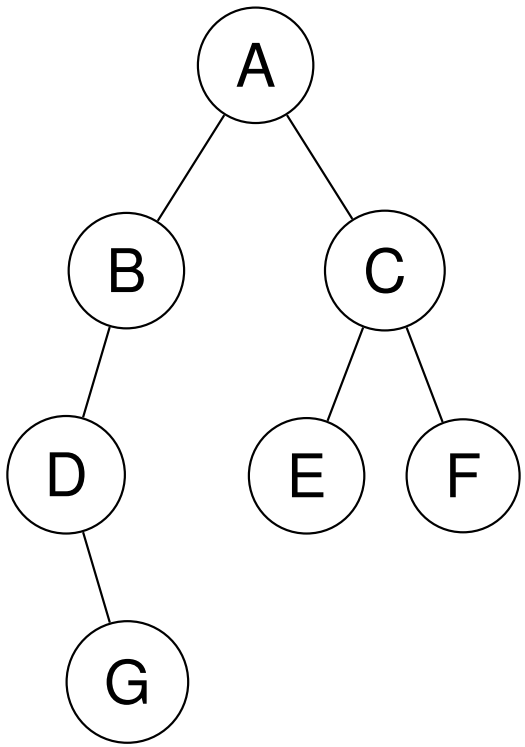
带头节点的中序线索链表

参照双向链表，在二叉树线索链表上添加头节点，令头节点的左孩子指向二叉树的根节点。优点：既可从第一个结点起顺后继遍历，也可从最后一个结点起顺前驱遍历。



练习

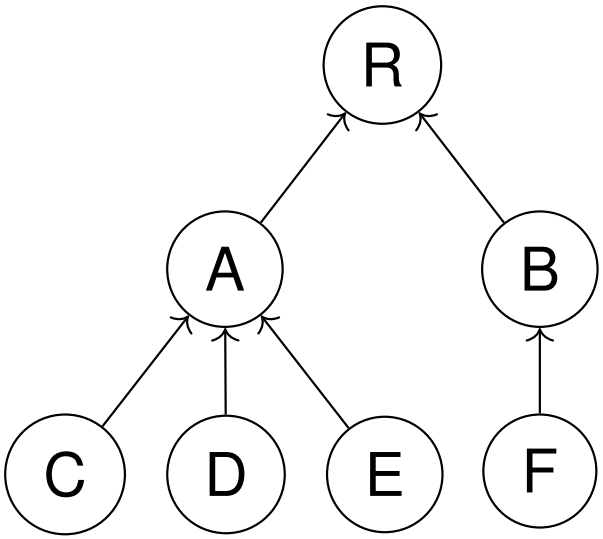
画出下图的先序、中序和后序线索二叉树。



树和森林

- 树的存储结构
- 树和二叉树的转换
- 森林和二叉树的转换
- 树和森林的遍历

树的存储结构：双亲表示法



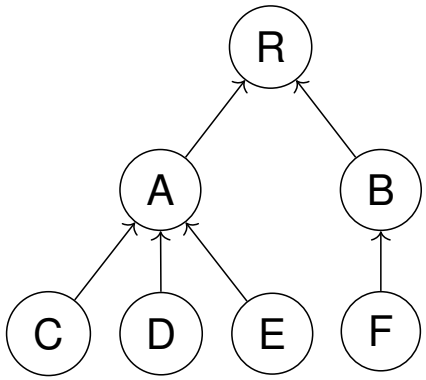
父结点索引	−	0	0	1	1	1	2
标记	R	A	B	C	D	E	F
结点索引	0	1	2	3	4	5	6

```
class PtNode<T> {
  T data;
  int parent;
}
```

```
typedef struct{
  telemtype data;
  int parent;
} PtNode;
```

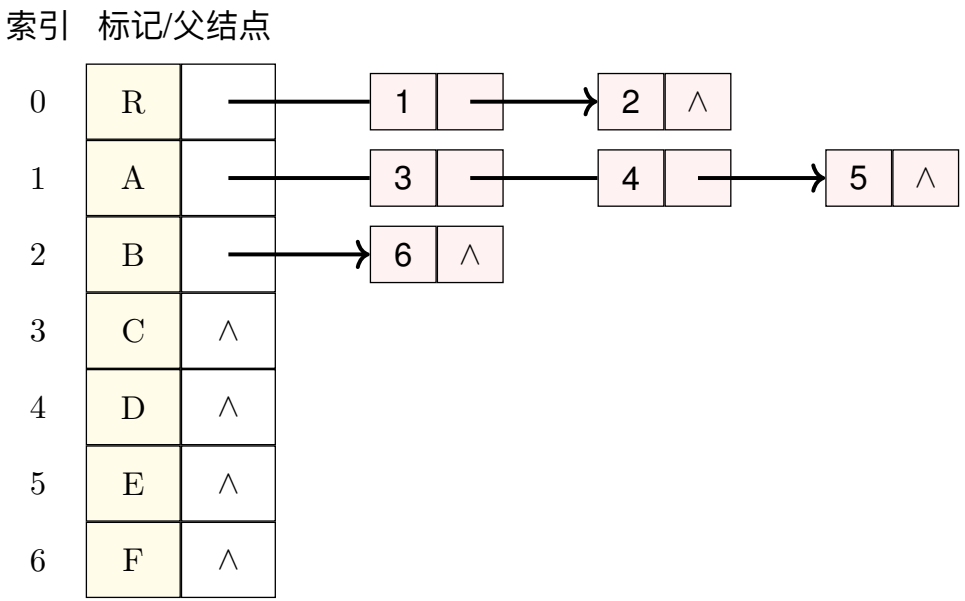
因节点的度各异, 从父亲指向孩子极不方便! 若经常需要查找孩子结点, 双亲表示法就不方便了。

树的存储结构：孩子表示法

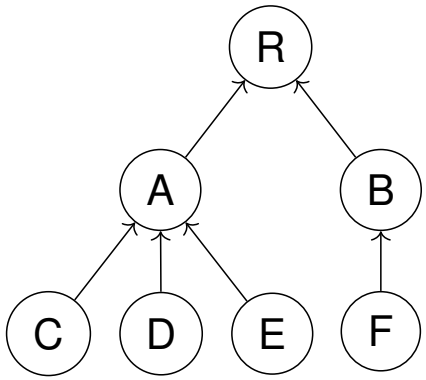


试写出 ChildNode, CTree

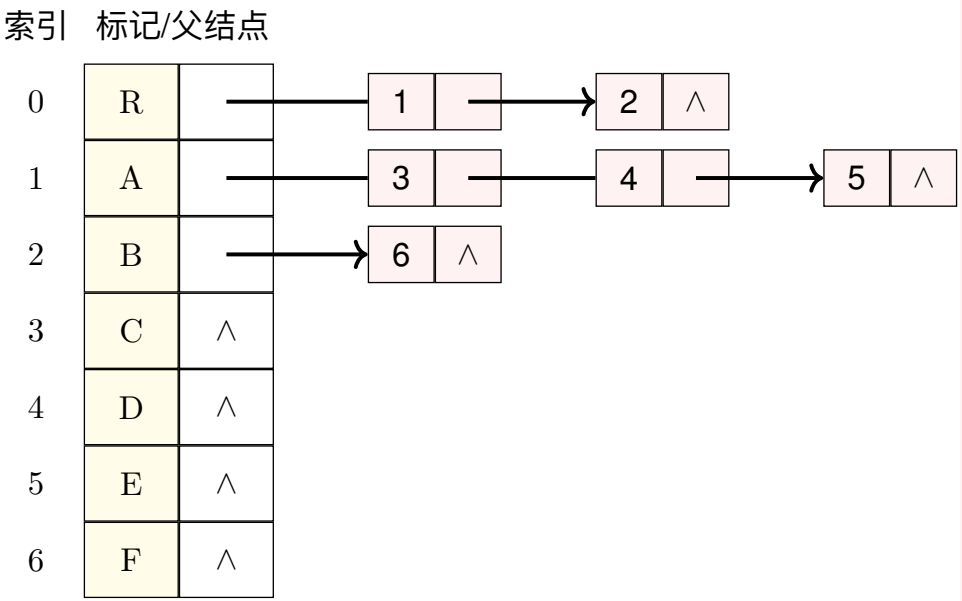
把每个结点的孩子结点排列成一个线性表, 以单链表做存储结构.



树的存储结构：孩子表示法



把每个结点的孩子结点排列成一个线性表, 以单链表做存储结构.



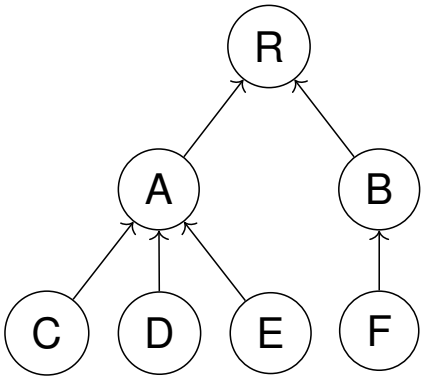
试写出 ChildNode, CTree

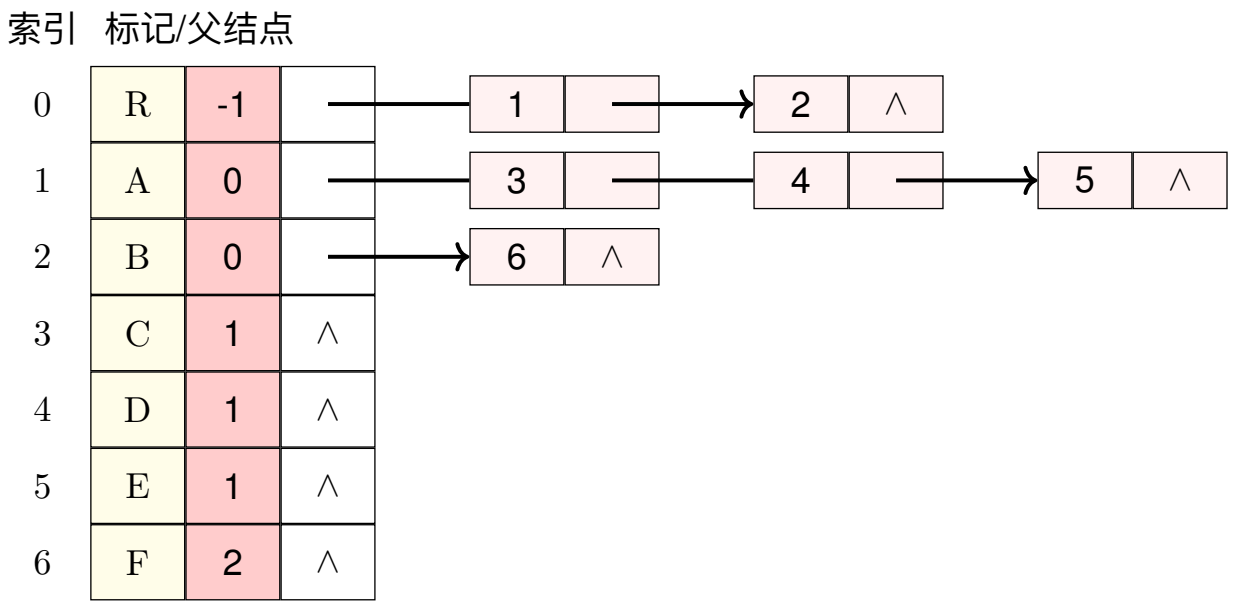
```
class TNode<T> {
  T data;
  ChildNode firstChild;
}

class ChildNode<T> {
  int index;
  ChildNode nextSibling;
}

class Tree {
  TNode<String>[] nodes = new
  ↪  TNode<String>[]{new
  ↪  TNode("R"), new TNode("A")};
  TNode root = nodes[0];
}
```

树的存储结构：双亲＋孩子表示法





树的存储结构：孩子兄弟表示法

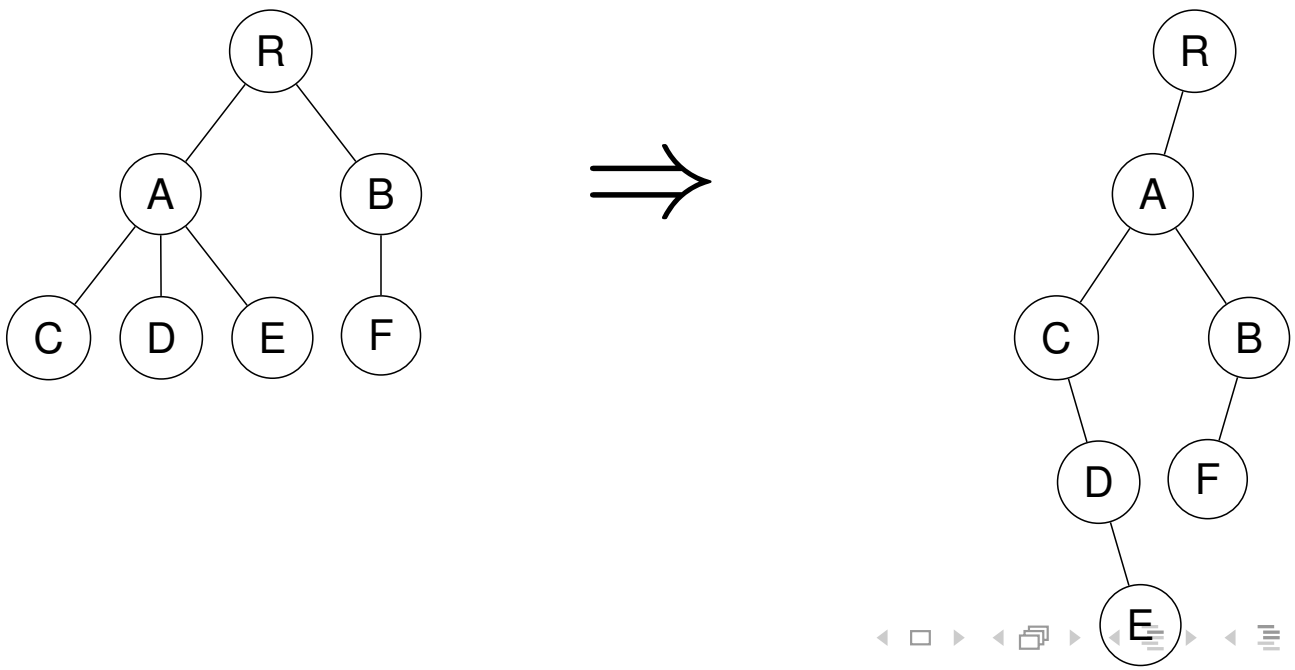
试想如何借鉴二叉树的存储方法?

树和森林的存储结构与算法都很复杂, 如果能够将它们转换为二叉树, 不但可以采用二叉树的存储结构, 而且可以利用二叉树的有关算法来实现有关操作。幸运的是, 树或森林与二叉树之间存在一一对应关系。

树的存储结构：孩子兄弟表示法

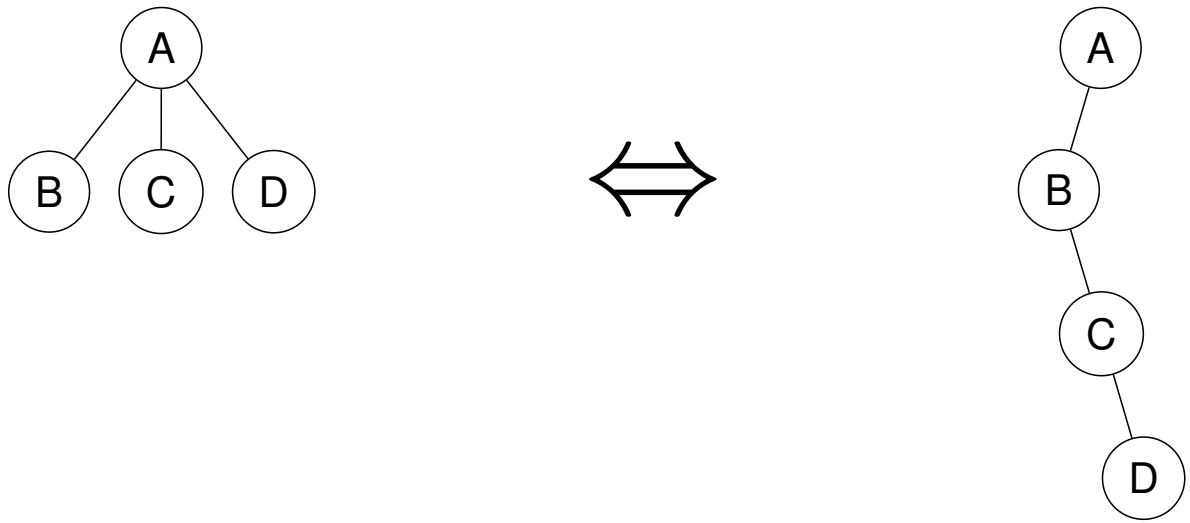
使用二叉链表, 两个指针分别指向第一个孩子和下一个兄弟.

```
class CSNode<T> {
  T data;
  CSNode<T> firstChild, nextSibling;
}
```



树与二叉树的转换

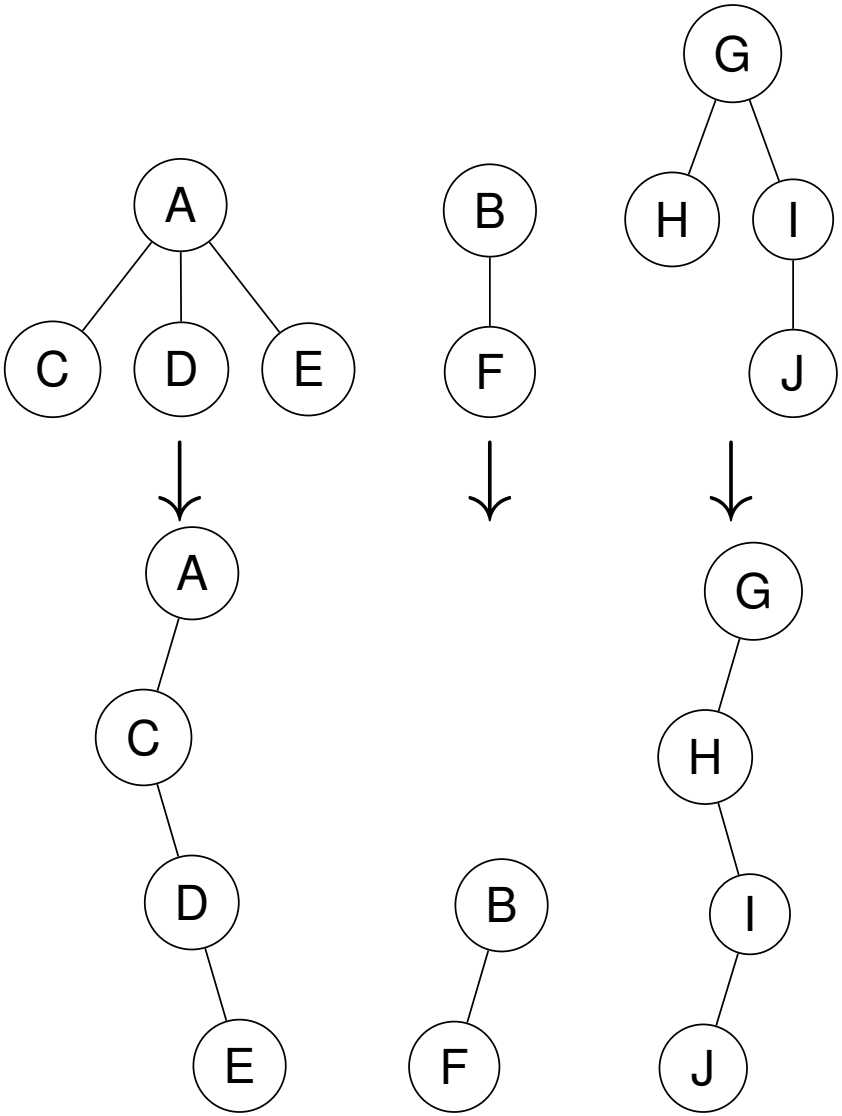
- 孩子作左子树的根, 兄弟作右子树的根
- 任何一棵树对应的二叉树, 其右子树必空, 也就是说, 所有的树都可以转化为二叉树, 但不是所有的二叉树都可以转化为树。



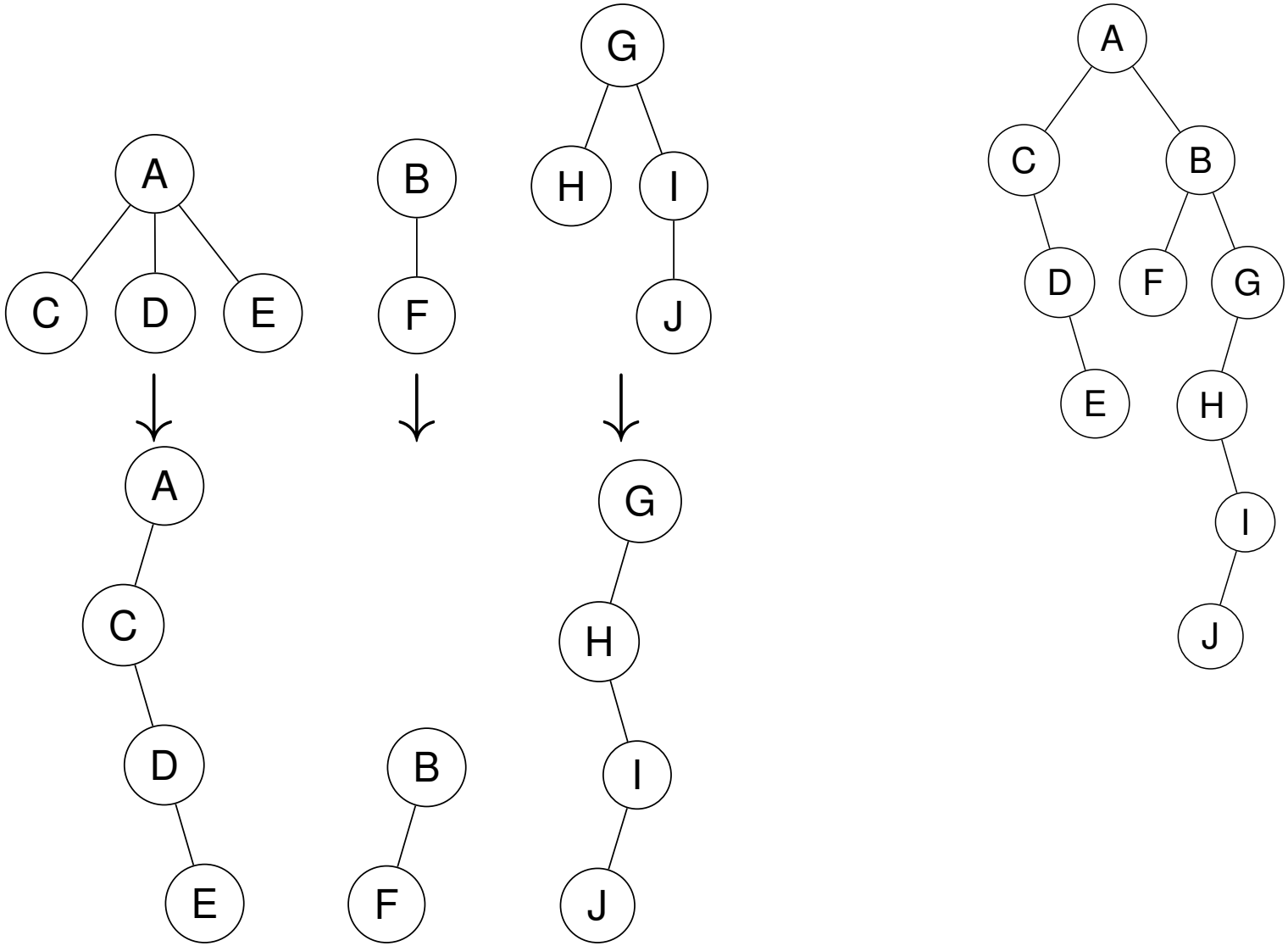
森林与二叉树的转换

- 零棵或有限棵不相交的树的集合称为森林。自然界中树和森林是不同的概念, 但在数据结构中, 树和森林只有很小的差别——任何一棵树, 删去根结点就变成了森林。
- 根据树的二叉链表表示, 任何一棵和树对应的二叉树的右子树必空。若将森林中第二棵树的根结点看成第一棵树的根结点的兄弟, 则可以导出森林和二叉树的对应关系。

森林与二叉树的转换示例



森林与二叉树的转换示例



树和森林的遍历

请按下述原则写出该树的遍历序列

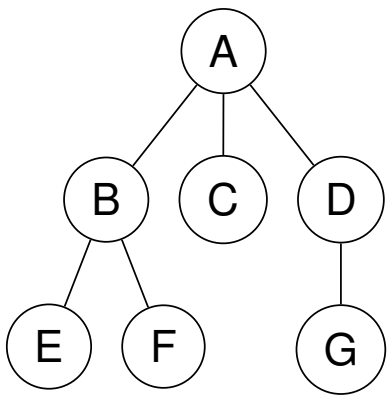
- 先根遍历: = 相应二叉树的先序遍历

  首先访问根结点; 再从左到右遍历根结点的每一棵子树。

- 后根遍历: = 相应二叉树的中序遍历

  首先按照从左到右的顺序后根遍历根结点的每一棵子树, 再访问根结点。

请画出该树对应的二叉树, 并写出该二叉树的先、中、后序遍历



树和森林的遍历

请按下述原则写出该树的遍历序列

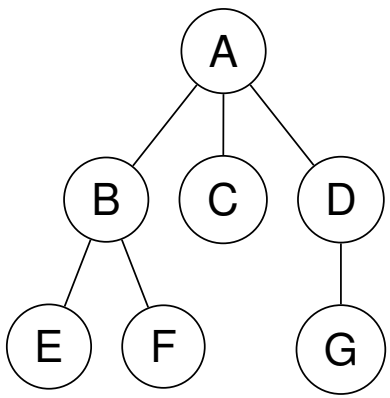
- 先根遍历: = 相应二叉树的先序遍历

  首先访问根结点; 再从左到右遍历根结点的每一棵子树。

- 后根遍历: = 相应二叉树的中序遍历

  首先按照从左到右的顺序后根遍历根结点的每一棵子树, 再访问根结点。

请画出该树对应的二叉树, 并写出该二叉树的先、中、后序遍历

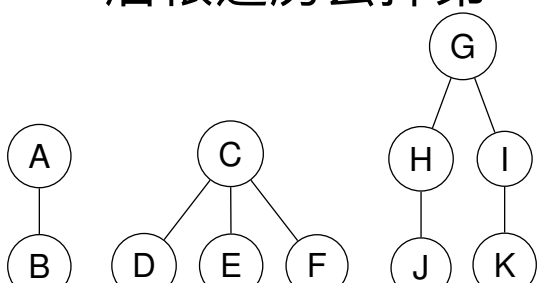


上图的先根遍历序列为:A B E F C D G

上图的后根遍历序列为:E F B C G D A

森林的遍历

- 前序遍历 (相应二叉树的先序遍历)
  - 访问森林中第一棵树的根结点;
  - 先根遍历第一棵树的根结点的子树;
  - 先根遍历去掉第一棵树后的子森林。
- 中序遍历 (相应二叉树的中序遍历)
  - 后根遍历第一棵树的根结点的子树;
  - 访问森林中第一棵树的根结点;
  - 后根遍历去掉第一棵树后的子森林。



前序遍历结果: A B C D E F G H J I K

中序遍历结果:B A D E F C J H K I G

哈夫曼树

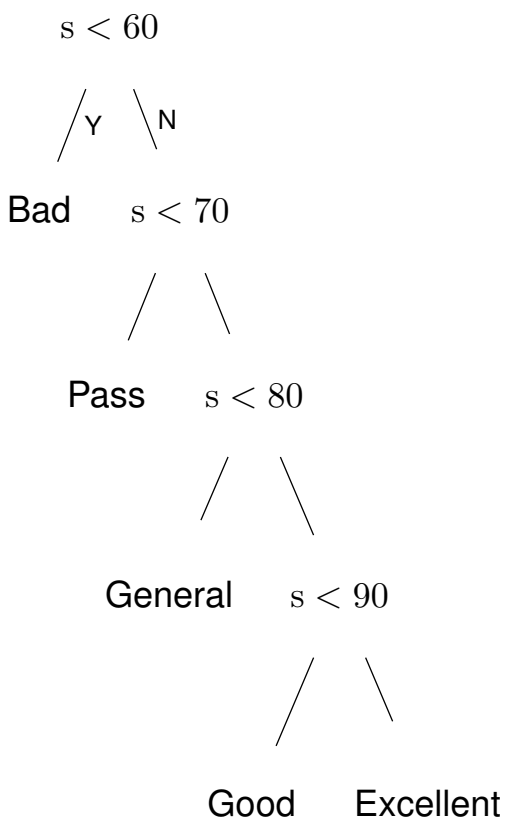
- Huffman coding (哈夫曼编码) 是 David A. Huffman 于 1952 年在麻省理工攻读博士时发明的一种编码方法, 可用于无损数据压缩的熵编码。Huffman 编码广泛应用在涉及数字压缩和传输的领域, 如 fax、modem、计算机网络和 HDTV。
- 原理：使用一张特殊的编码表将源字符进行编码, 它的特殊之处在于它是根据每一个源字符出现的估算概率建立起来的——出现概率高的字符使用较短的编码, 反之出现概率低的则使用较长的编码, 使编码后的字符串的平均期望长度降低, 从而达到无损压缩数据的目的。若能比较准确的估计英文中各个字母出现概率, 就能大幅提高无损压缩的效率。
- 例如, 在英文中, 用普通的表示方法时, 每个英文字母均占用一个字节 (byte), 即 8 位。实际上,e 的出现概率很高, 而 z 的出现概率则最低。当利用哈夫曼编码进行压缩时,e 可能用 1 位来表示,z 可能用 25 位。

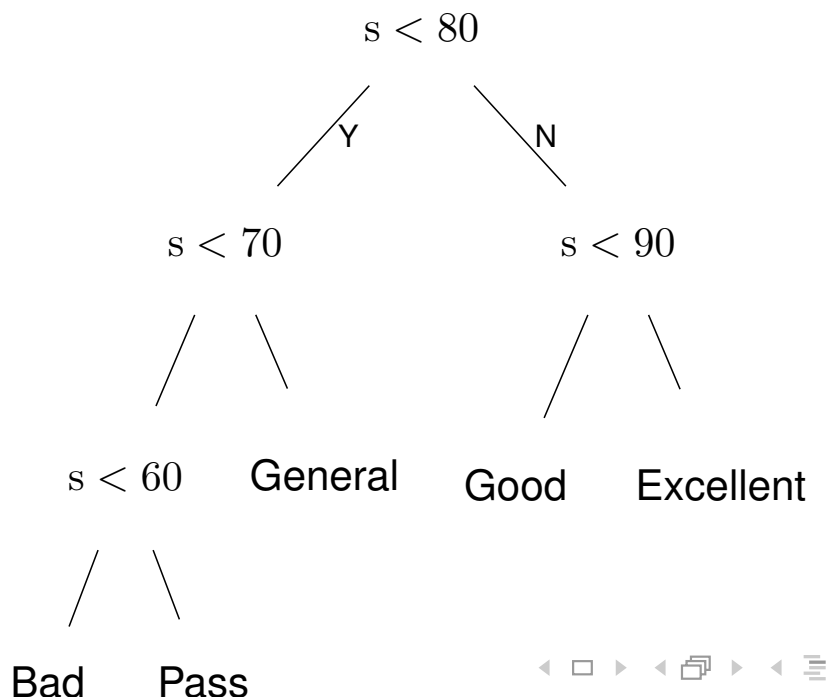
David Huffman (1925/08/09 – 1999/10/07)

- 1950 年,Huffman 在 MIT 的信息理论与编码研究生班学习。Robert Fano 教授让学生们自己决定是参加期末考试还是做一个大作业, 因为大作业比期末考试可能更容易通过,Huffman 选择了后者。正是一个大作业促使了 Huffman 算法的诞生!
- 离开 MIT 后,Huffman 来到 University of California, Santa Cruz 的计算机系任教, 并为此系的学术做出了许多杰出的工作。而他的算法也广泛应用于传真机, 图象压缩和计算机安全领域。但是 Huffman 却从未为此算法申请专利或其它能够为他带来经济利益的东西, 他将他全部的精力放在教学上, 用他自己的话来说,“My products are my students.”
- David Huffman 对于有限状态自动机, 开关电路, 异步过程和信号设计也做出了杰出贡献。

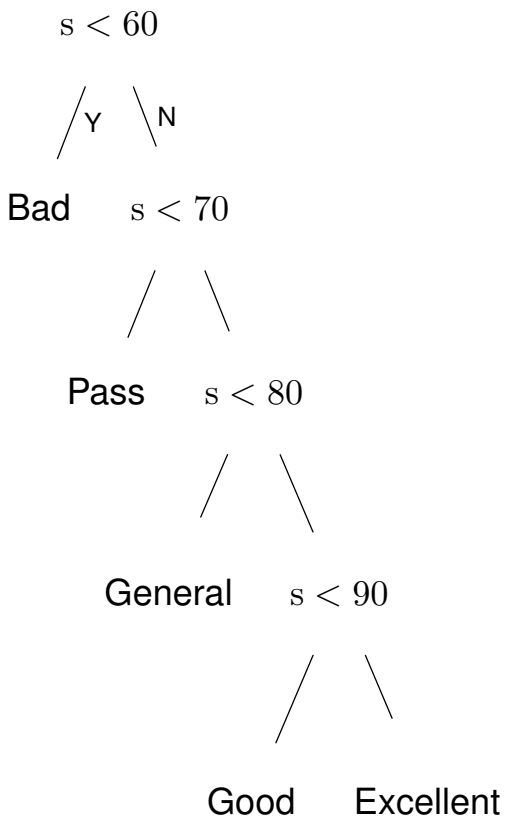
举例: 假设学生成绩分布如下, 试构造一棵树

Grade	Bad	Pass	General	Good	Excellent
Score	0 – 59	60 – 69	70 – 79	80 – 89	90 – 100
Ratio	0.05	0.15	0.40	0.30	0.10





Grade	Bad	Pass	General	Good	Excellent
Score	0 – 59	60 – 69	70 – 79	80 – 89	90 – 100
Ratio	0.05	0.15	0.40	0.30	0.10



- 结点的带权路径长度 (Weighted Path Length)：

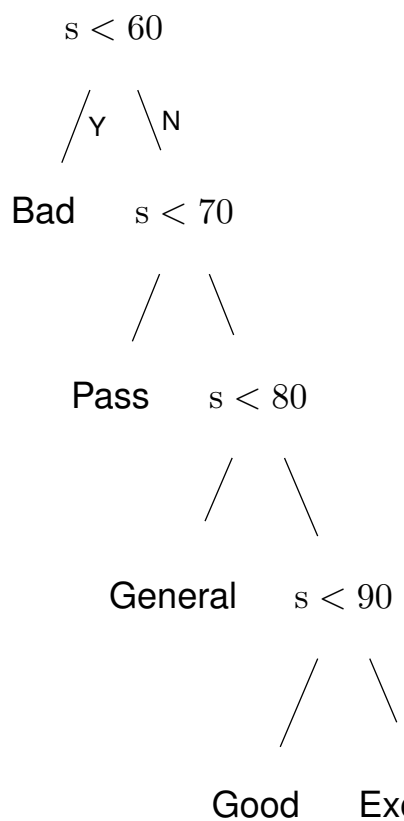
$$\mathrm{WPL}(\mathrm{good}) = 4 * 0.3 = 1.2$$

- 树的带权路径长度:

$$\begin{aligned}\mathrm{WPL}(\mathrm{TREE}) &= 1 \times 0.05 \\ &\quad +2 \times 0.15 \\ &\quad +3 \times 0.40 \\ &\quad +4 \times 0.30 \\ &\quad +4 \times 0.10 \\ &= 3.15\end{aligned}$$

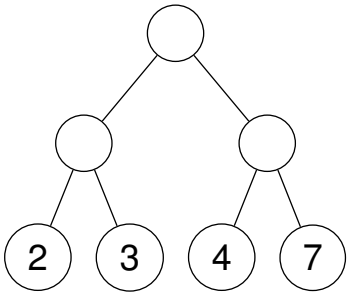
练习：求树的带权路径长度

Grade	Bad	Pass	General	Good	Excellent
Score	0 – 59	60 – 69	70 – 79	80 – 89	90 – 100
Ratio	0.05	0.15	0.40	0.30	0.10

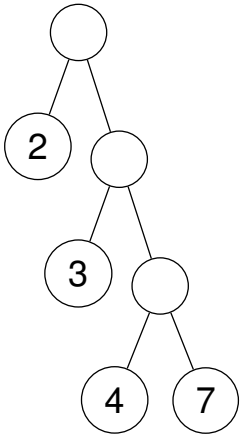


哈夫曼树

具有最小带权路径长度的二叉树称为哈夫曼树 (最优二叉树)。



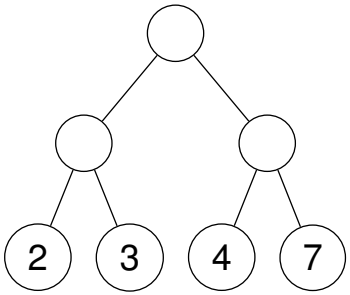
$$\mathrm{WPL} = 2\times 2 + 2\times 3 + 2\times 4 + 2\times 7 = 32$$



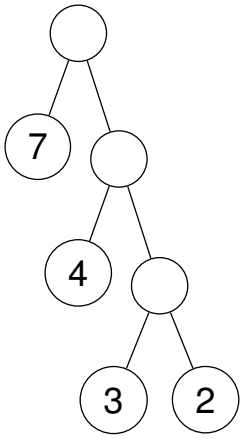
$$\mathrm{WPL} = 1\times 2 + 2\times 3 + 3\times 4 + 3\times 7 = 41$$

哈夫曼树

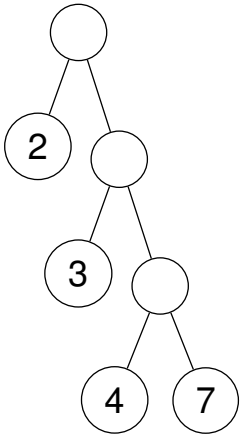
具有最小带权路径长度的二叉树称为哈夫曼树 (最优二叉树)。



$$\mathrm{WPL} = 2\times 2 + 2\times 3 + 2\times 4 + 2\times 7 = 32$$



$$\mathrm{WPL} = 30$$



$$\mathrm{WPL} = 1\times 2 + 2\times 3 + 3\times 4 + 3\times 7 = 41$$

哈夫曼树的构造

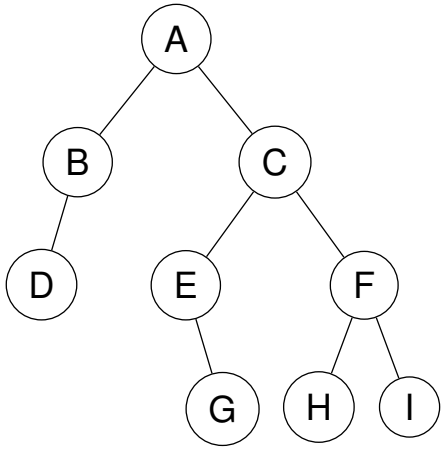
由给定的 n 个权值 $\{w_1, w_2, \cdots, w_n\}$, 构造 n 棵只有一个叶子结点的二叉树, 从而得到一个二叉树的集合 $F = \{T_1, T_2, \cdots, T_n\}$.

1. 在 F 中选取根结点的权值最小和次小的两棵二叉树作为左、右子树构造一棵新的二叉树, 这棵新的二叉树根结点的权值为其左、右子树根结点权值之和;
2. 在集合 F 中删除作为左、右子树的两棵二叉树, 并将新建立的二叉树加入到集合 F 中;
3. 重复 (2)、(3) 两步, 当 F 中只剩下一棵二叉树时, 这棵二叉树便是所要建立的哈夫曼树。

例子

1, 3, 5, 7

课堂作业




```
class TNode {
   String data;
   int lc, rc;
}
```

0	A	1	2
1	B	3	-1
2	C	4	5
3	D	-1	-1
4	E	-1	6
5	F	7	8
6	G	-1	-1
7	H	-1	-1
8	I	-1	-1

- 给定树 T 如图所示, 请写递归程序实现如下功能:
- (1) 遍历树 T, 显示其先序/中序/后序序列;
- (2) 求结点数、叶子数;
- (3) 求树 T 的深度.

本章作业

哈夫曼编码

- 根据 26 个英文字母的频度, 为每一个字母赋予一个唯一的最优编码
- 输出要求, 每行一个字母, 及其对应编码, 如:
  - A: 00100
  - B: 0001
  - ...

英文字母的频率表

字母	出现频率	字母	出现频率	字母	出现频率
a	8.167%	j	0.153%	s	6.327%
b	1.492%	k	0.772%	t	9.056%
c	2.782%	l	4.025%	u	2.758%
d	4.253%	m	2.406%	v	0.978%
e	12.702%	n	6.749%	w	2.360%
f	2.228%	o	7.507%	x	0.150%
g	2.015%	p	1.929%	y	1.974%
h	6.094%	q	0.095%	z	0.074%
i	6.966%	r	5.987%		