Binary Search Trees

- When arbitrary elements are to be searched or deleted, heap is not suitable.
- A dictionary is a collection of pairs, each pair has a key and an associated element.
- Although natural dictionaries may have several pairs with the same key, we assume here that no two pairs have the same key.

ADT of Binary Search Trees

```
template <class K, class E>
class Dictionary {
public:
   virtual bool IsEmpty () const = 0;
      // return true iff the dictionary is empty
   virtual pair<K,E>* Get(const K&) const = 0;
      // return pointer to the pair with specified key;
      // return 0 if no such pair
   virtual void Insert(const pair<K,E>&) = 0;
      // insert the given pair; if key is a duplicate
      // update associated element
   virtual void Delete(const K&) = 0;
      // delete pair with specified key
};
```

The pair can be defined as:

```
template <class K, class E>
struct pair
{
    K first;
    E second;
};
```

Definition of Binary Search Trees

A binary search tree, if not empty, satisfies the following properties:

(1) The root has a key.

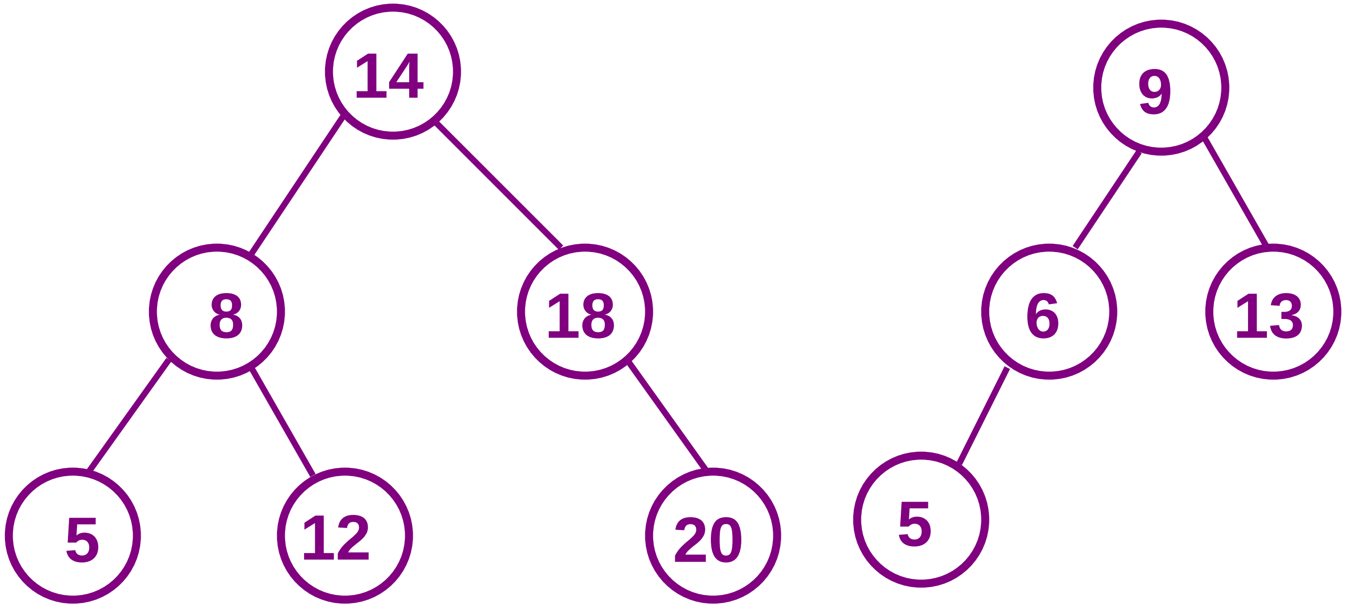
(2) The keys (if any) in the left subtree are smaller than that in the root.

(3) The keys (if any) in the right subtree are larger than that in the root.

(4) The left and right subtrees are also binary search trees.

A binary search tree has a better performance when the functions to be performed are search, insert and delete.

Note these properties imply that the keys must be distinct.



Binary search trees

Searching a Binary Search Tree

According to its properties, it is easy to search a binary search tree. Suppose search for an element with key k:

If k==the key in root, success;

If x<the key in root, search the left subtree;

If x>the key in root, search the right subtree.

```
template <class K, class E> // Driver
pair<K,E>* BST<K,E>::Get(const K& k)
{ // Search *this for a pair with key k.
    return Get(root, k);
}

template <class K, class E> // Workhorse
pair<K,E>* BST<K,E>::Get(treeNode<pair<K,E>>* p,
                                                    const K& k)
{
    if (!p) return 0;
    if (k < p→data.first) return Get(p→leftChild, k);
    if (k > p→data.first) return Get(p→rightChild, k);
    return &p→data;
}
```

The recursive version can be easily changed into an iterative one as in the following:

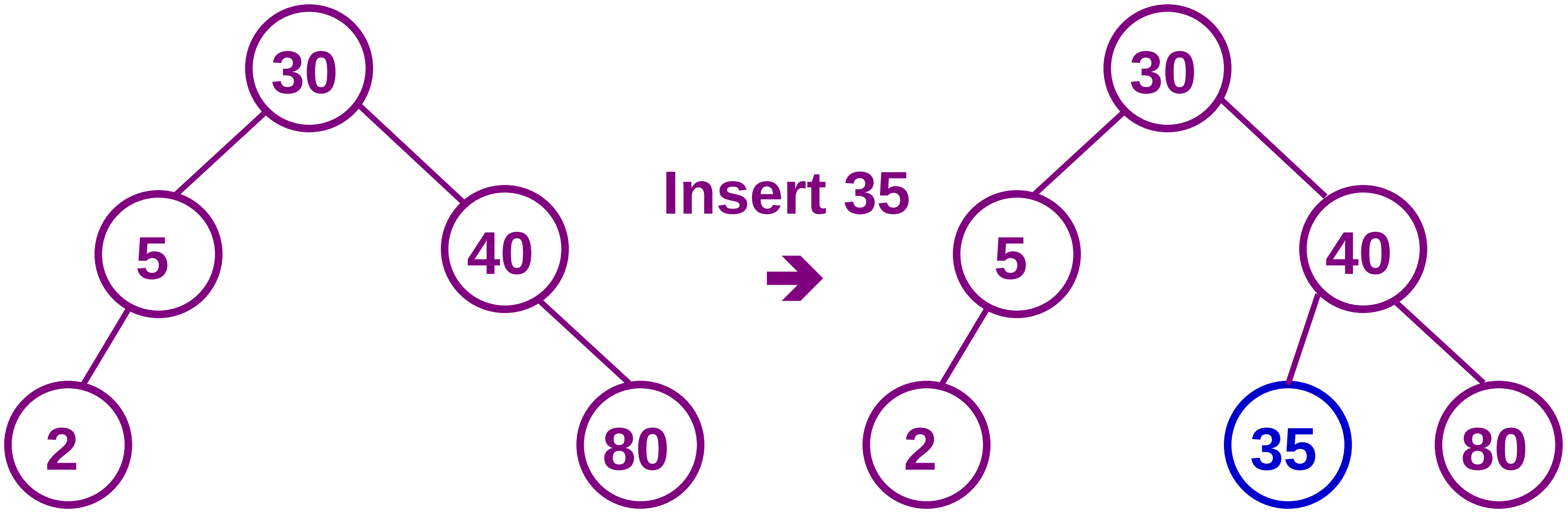
```
template <class K, class E>  // Iterative version
pair<K,E>* BST<K,E>::Get(const K& k)
{
    TreeNode<pair<K,E>>* currentNode = root;
    while (currentNode)
      if (k < currentNode→data.first)
          currentNode = currentNode→leftChild;
      else if (k > currentNode→data.first)
          currentNode = currentNode→rightChild;
      else return &currentNode→data;

    // no matching pairs
    return 0;
}
```

As can be seen, a binary search tree of height h can be search by key in O(h) time.

Insertion into a Binary Search Tree

To insert a new element, search is carried out. If unsuccessful, then the element is inserted at the point the search terminated.



```
template <class K, class E>
void BST<K,E>::Insert(const pair<K,E>& thePair)
{ // Insert thePair into the binary search tree
      // search for thePair.first, pp is parent of p
      TreeNode<pair<K,E>> *p=root, *pp=0;
      while (p) {
        pp=p;
        if (thePair.first < p→data.first) p=p→leftChild;
        else if (thePair.first > p→data.first) p=p→rightChild;
        else // duplicate, update associated element
             {p→data.second=thePair.second;return;}
      }
```

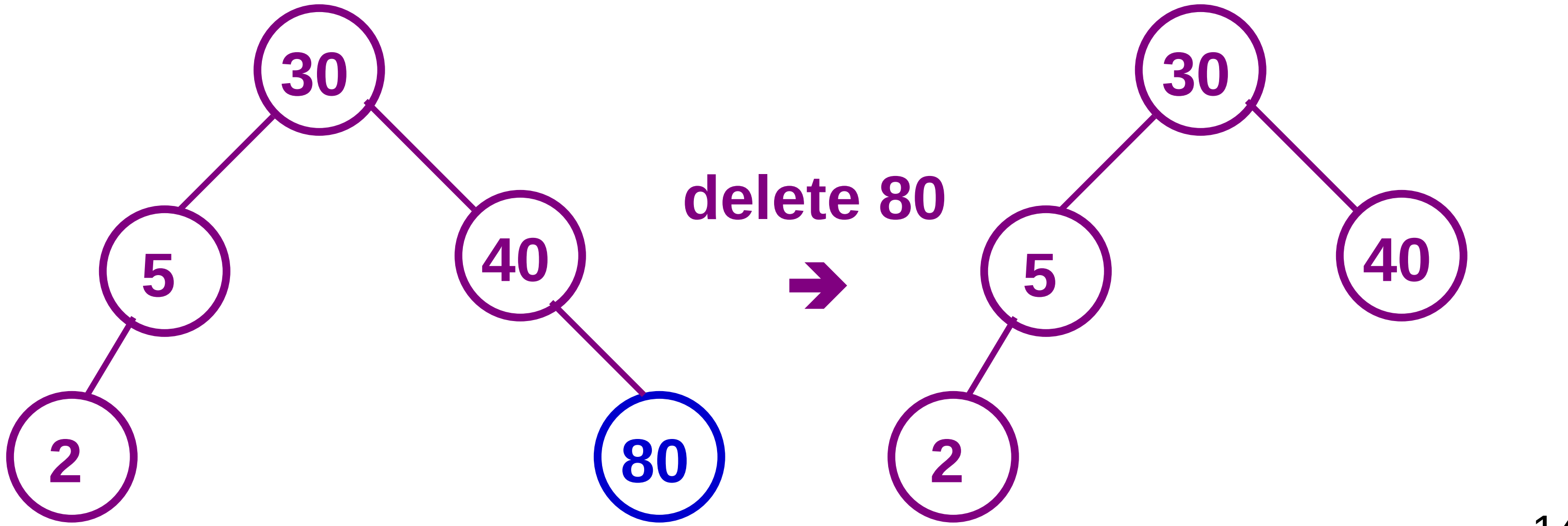
```
    // perform insertion
    p=new TreeNode<pair<K,E>>(thePair,0,0);
    if (root) // tree not empty
      if (thePair.first < pp→data.first) pp→leftChild=p;
        else pp→rightChild=p;
    else root=p;
}
```

The time is O(h).

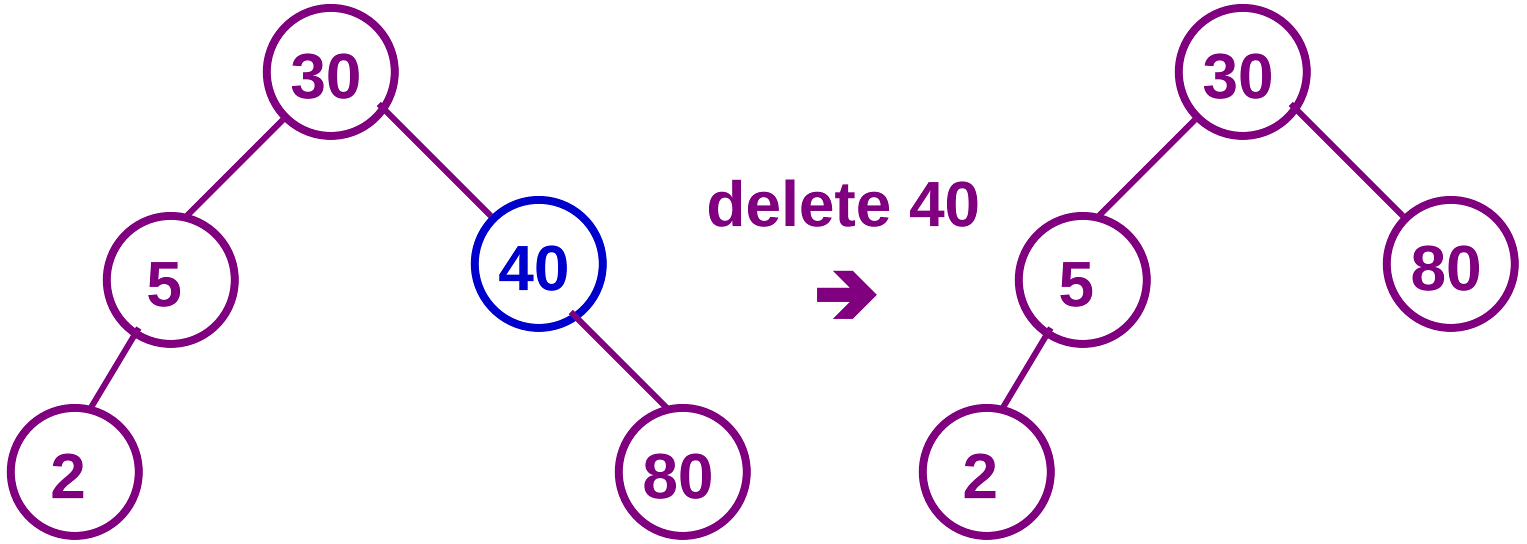
Deletion from a Binary Search Tree

3 cases:

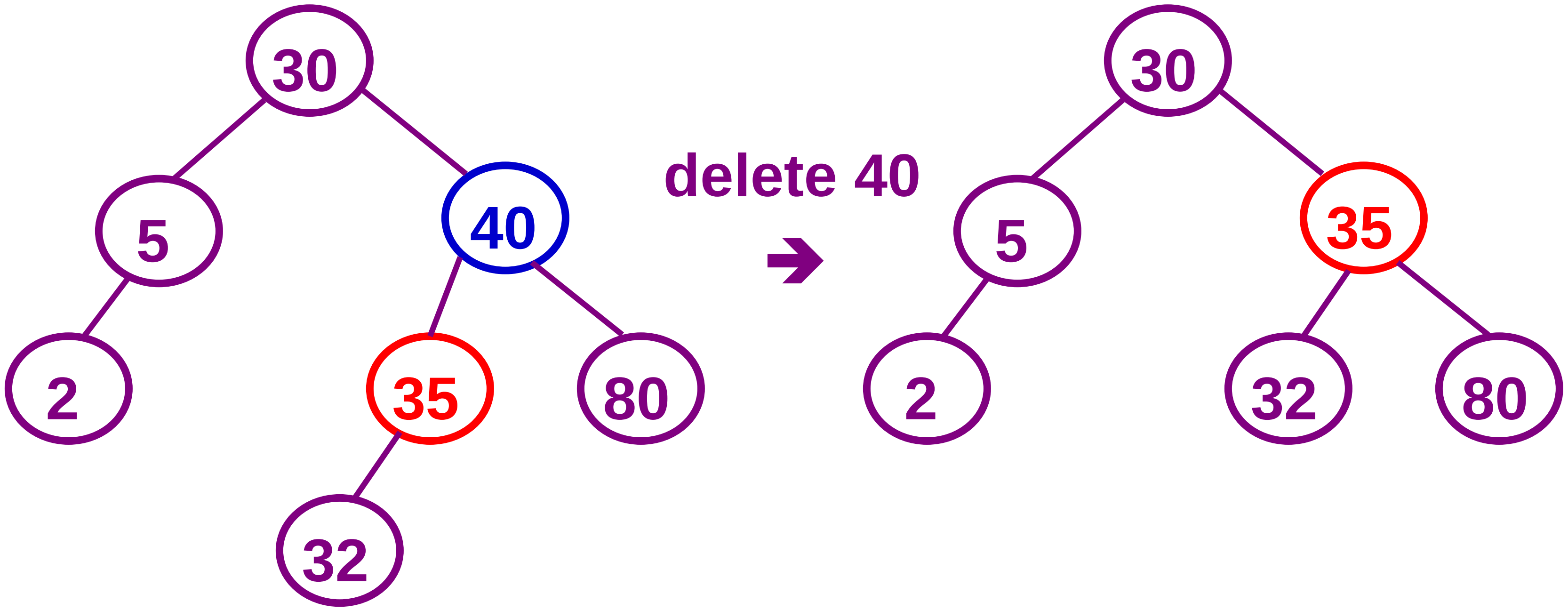
(1) deletion of a leaf: easy, set the corresponding child field of its parent to 0 and dispose the node.



(2) deletion of a nonleaf node with one child: easy, the child of the node takes the place of it and the node disposed.



(3) deletion of a nonleaf node with two children: the element is replaced by either the largest in its leftChild or the smallest in its rightChild. Then delete that node, which has at most one child, in the subtree.

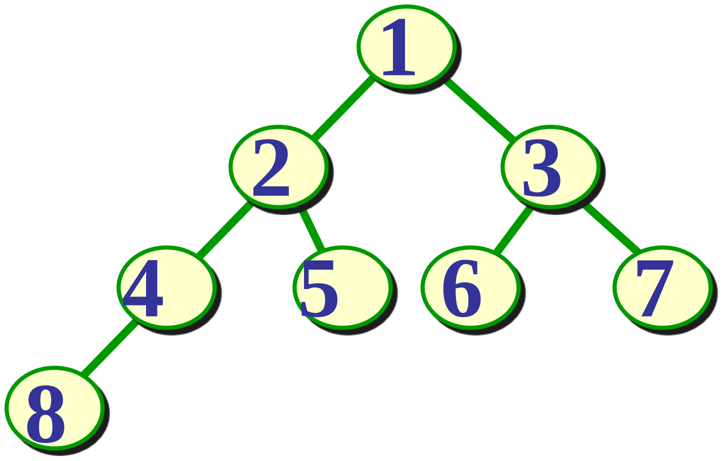


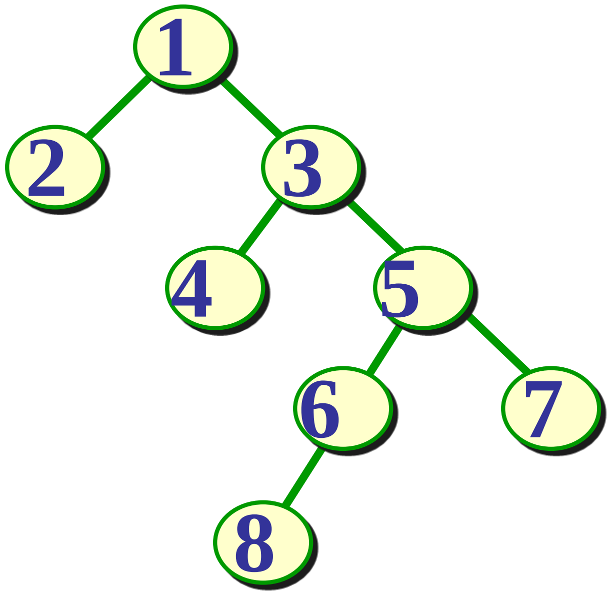
Trees

- Trees
- Binary Trees
  - Definition and Features
  - Binary Tree Representations
  - Traversal
  - Threaded Binary Trees
- Heaps and Binary Search Trees
  - Heaps
  - Huffman Trees
- Forrests
  - Tree Representations
  - Transforming between Trees and Forrests
  - Traversal

Path Length

- 树的外部路径长度是各叶结点（外结点）到根结点的路径长度之和 EPL
- 树的内部路径长度是各非叶结点（内结点）到根结点的路径长度之和 IPL
- 树的路径长度 PL = EPL + IPL





IPL = 0+1+1+2 = 4

EPL = 2+2+2+3 = 9

PL = 13

IPL = 0+1+2+3 = 6

EPL = 1+2+3+4 = 10

PL = 16

- *n* 个结点的二叉树的路径长度不小于下述数列前 *n* 项的和，即：

$$PL = \sum_{i=1}^{n} \lfloor \log_2 i \rfloor$$

$$= 0 + 1 + 1 + 2 + 2 + 2 + 2 + 3 + 3 + \cdots$$

- 其最小路径长度等于PL，即：

$$PL = \sum_{i=1}^{n} \lfloor \log_2 i \rfloor$$

- 完全二叉树满足这个要求

Weighted Path Length

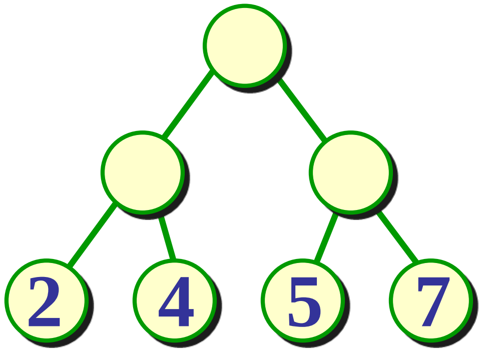
- 在很多应用问题中为树的叶结点赋予一个权值，用于表示出现频度、概率值等。
- 因此，在问题处理中把叶结点定义得不同于非叶结点，把叶结点看成“外结点”，非叶结点看成“内结点”。这样的二叉树称为扩充二叉树
- 扩充二叉树中只有度为 2 的内结点和度为 0的外结点
- 根据二叉树的性质，有 n 个外结点就有 $n-1$ 个内结点，总结点数为$2n-1$。

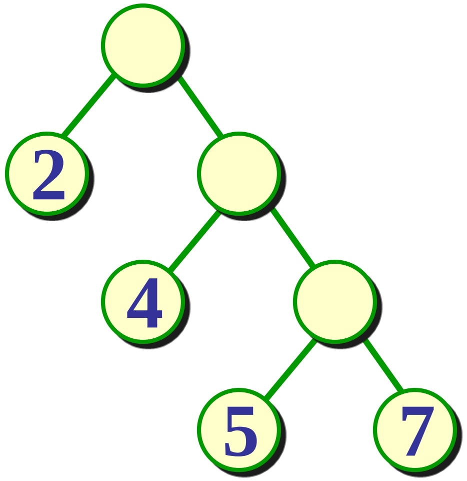
- 若一棵扩充二叉树有 n 个外结点，第 i 个外结点的权值为w_i，它到根的路径长度为l_i，则该外结点到根的带权路径长度为w_i*l_i
- 扩充二叉树的带权路径长度定义为树的各外结点到根的带权路径长度之和

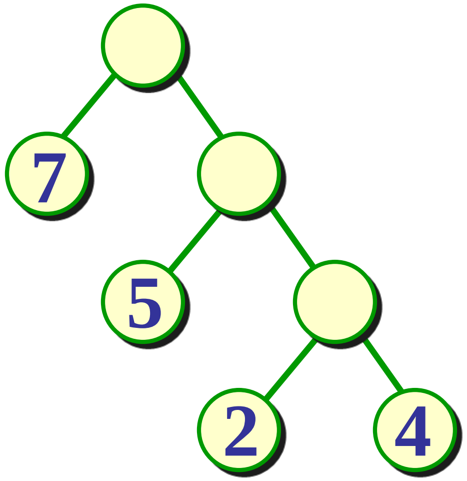
$$WPL = \sum_{i=1}^{n} w_i * l_i$$

- 对于同样一组权值，如果放在外结点上，组织方式不同，带权路径长度也不同

具有不同带权路径长度的扩充二叉树







WPL = 2*2+ 4*2+5*2+ 7*2 = 36

WPL = 2*1+ 4*2+5*3+ 7*3 = 46

WPL = 7*1+ 5*2+2*3+ 4*3 = 35

带权路径长度达到最小

Huffman Tree

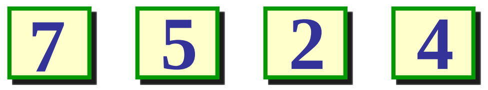
- Huffman树
  - 带权路径长度达到最小的扩充二叉树
  - 且权值大的结点离根最近

Construct a Huffman Tree

1. 由给定 n 个权值 $\{w_0, w_1, w_2, \ldots, w_{n-1}\}$，构造 具有 n 棵扩充二叉树的森林 $F = \{ T_0, T_1, T_2, \ldots, T_{n-1} \}$,
   其中, 每棵扩充二叉树 T_i 只有一个带权值 w_i 的根结点, 其左、右子树均为空。
2. 重复以下步骤, 直到 F 中仅剩一棵树为止：
   a) 在 F 中选取两棵根结点的权值最小的扩充二叉树, 做为左、右子树构造一棵新的二叉树。置新的二叉树的根结点的权值为其左、右子树上根结点的权值之和
   b) 在 F 中删去这两棵二叉树。
   c) 把新的二叉树加入 F。

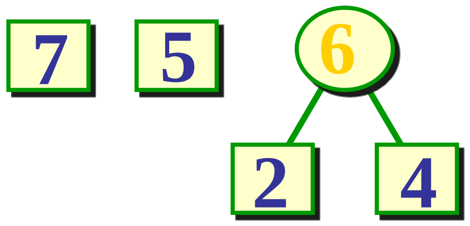
The Procedure

F : {7} {5} {2} {4}



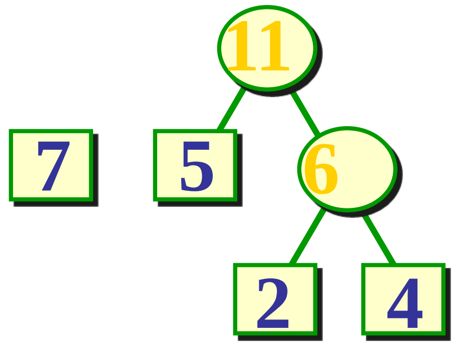
初始

F : {7} {5} {6}



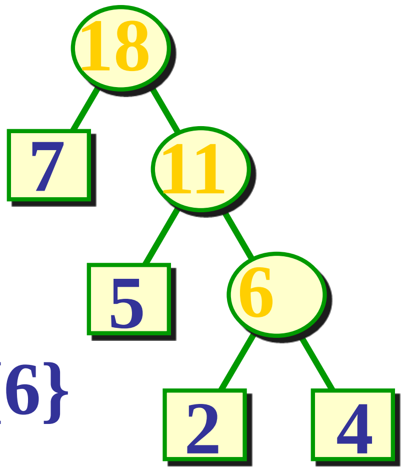
合并{2} {4}

F : {7} {11}



合并{5} {6}

F : {18}



合并{5} {6}

Definition

```
#include "heap.h"
const  int DefaultSize = 20;    //缺省权值集合大小
template <class T, class E>
struct HuffmanNode {            //树结点的类定义
    E data;                          //结点数据
     HuffmanNode<T, E> *parent;
    HuffmanNode<T, E> *leftChild, *rightChild;
                        //左、右子女和父结点指针
     HuffmanNode () : Parent(NULL), leftChild(NULL),
        rightChild(NULL) { }  //构造函数
    HuffmanNode (E elem,                    //构造函数
        HuffmanNode<T, E> *pr = NULL,
        HuffmanNode<T, E> *left = NULL,
        HuffmanNode<T, E> *right = NULL)
            : data (elem), parent (pr), leftChild (left),
              rightChild (right) { }
};
```

```
template <class T,  class E>
class HuffmanTree {              //Huffman树类定义
public:
    HuffmanTree (E w[], int n);         //构造函数
   ～HuffmanTree() { delete Tree(root);}  //析构函数
protected:
    HuffmanNode<T, E> *root;              //树的根
    void deleteTree (HuffmanNode<T, E> *t);
         //删除以 t 为根的子树
    void mergeTree (HuffmanNode<T, E>& ht1,
         HuffmanNode<T, E>& ht2,
        HuffmanNode<T, E> *& parent) ;
};
```

Building a Huffman Tree

```
template <class T, class E>
HuffmanTree<T, E>::HuffmanTree (E w[], int n)
{
//给出 n 个权值w[1]～w[n], 构造Huffman树
    minHeap<T, E> hp;          //使用最小堆存放森林
    HuffmanNode<T, E> *parent, &first, &second;
    HuffmanNode<T, E> *NodeList =
            new HuffmanNode<T,E>[n];    //森林
    for (int i = 0; i < n; i++) {
           NodeList[i].data = w[i+1];
           NodeList[i].leftChild = NULL;
```

```
        NodeList[i].rightChild = NULL;
        NodeList[i].parent = NULL;
        hp.Insert(NodeList[i]); //插入最小堆中
    }
    for (i = 0; i < n−1; i++) {   //n-1趟, 建Huffman树
        hp.Remove (first);                //根权值最小的树
        hp.Remove (second);    //根权值次小的树
         merge (first, second, parent);  //合并
        hp.Insert (*parent);              //重新插入堆中
        root = parent;          //建立根结点
    }
}
```

```
template <class T, class E>
void HuffmanTree<T, E>::
mergeTree (HuffmanNode<T, E>& bt1,
        HuffmanNode<T, E>& bt2,
        HuffmanNode<T, E> *& parent) {
    parent = new E;
    parent−>leftChild = &bt1;
    parent−>rightChild = &bt2;
    parent−>data.key =
          bt1.root−>data.key+bt2.root−>data.key;
    bt1.root−>parent = bt2.root−>parent = parent;
};
```

Huffman Tree with Links

- 可以采用静态链表方式存储Huffman树，其类定义如下：

```
const  int n = 20;
const int m = 2*n−1;

typedef struct {
    float weight;
    int parent, lchild, rchild;
} HTNode;

typedef HTNode HuffmanTree[m];
```

Create a Huffman Tree

```
void CreateHuffmanTree (HuffmanTree T, float fr[ ], int n)    {
      for (int i = 0; i < n; i++) T[i].weight = fr[i];
      for (i = 0; i < m; i++) {
        T[i].parent = T[i].lchild = T[i].rchild = –1;
        }

      for (i = n; i < m; i++) {          //求n–1次根
        int min1 = min2 = MaxNum;
        int pos1, pos2;
        for (int j = 0; j < i; j++)    //检测前 i 棵树
```

```
          if (T[j].parent == −1)         //可参选的树根
            if (T[j].weight < min1) {         //选最小
                pos2 = pos1;  min2 = min1;
                pos1 = j;  min1 = T[j].weight;
            }
            else if (T[j].weight < min2)     //选次小
                { pos2 = j;  min2 = T[j].weight; }
      T[i].lchild = pos1;  T[i].rchild = pos2;
      T[i].weight = T[pos1].weight+T[pos2].weight;
      T[pos1].parent = T[pos2].parent = i;
   }
}
```

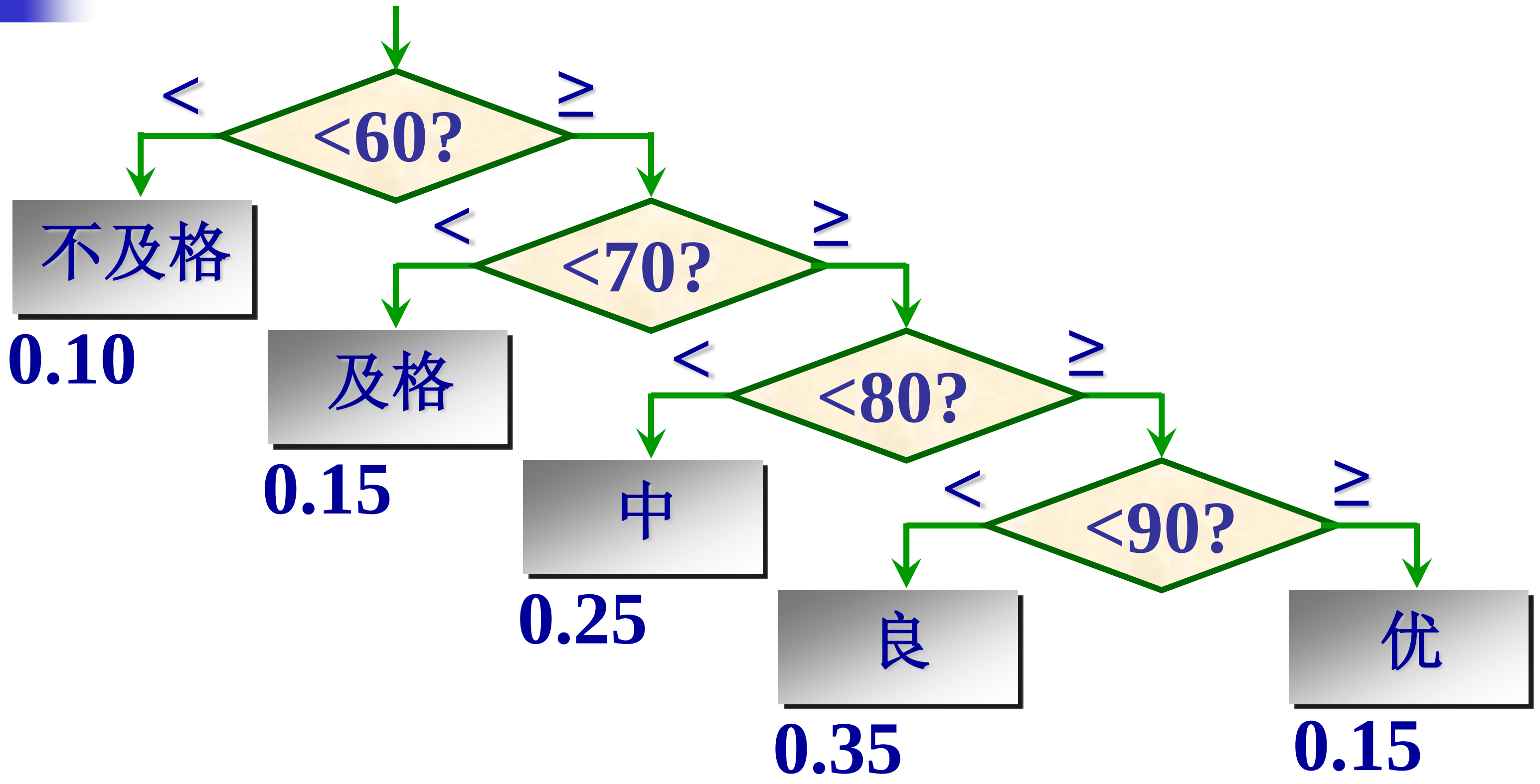
Decision Tree

- 利用Huffman树，可以在构造判定树（决策树）时让平均判定（比较）次数达到最小。
- 判定树是一棵扩展二叉树，外结点是比较结果，内结点是比较过程，外结点所带权值是概率。

考试成绩分布表

[0, 60)	[60, 70)	[70, 80)	[80, 90)	[90, 100)
不及格	及格	中	良	优
0.10	0.15	0.25	0.35	0.15

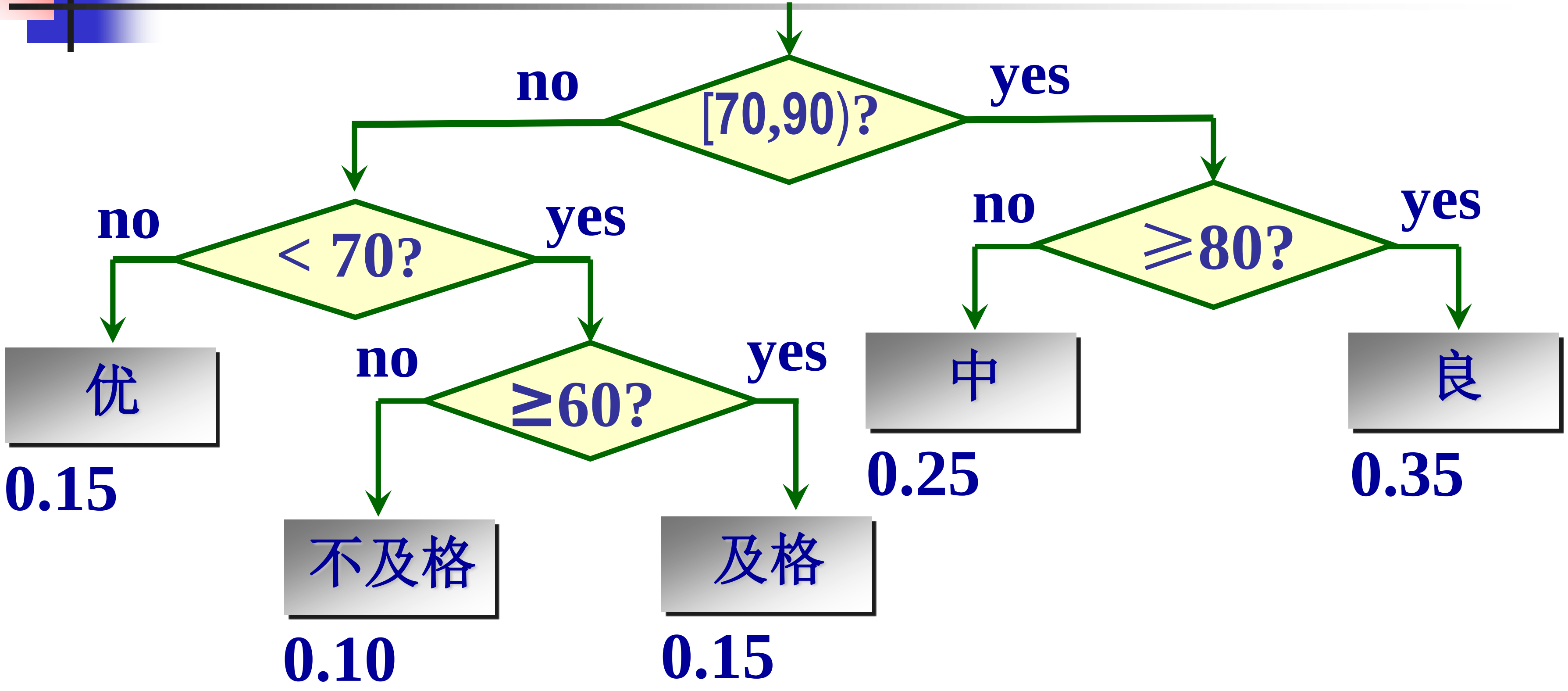
Decision Tree



WPL = 0.10*1+0.15*2+0.25*3+0.35*4+0.15*4

= 3.15

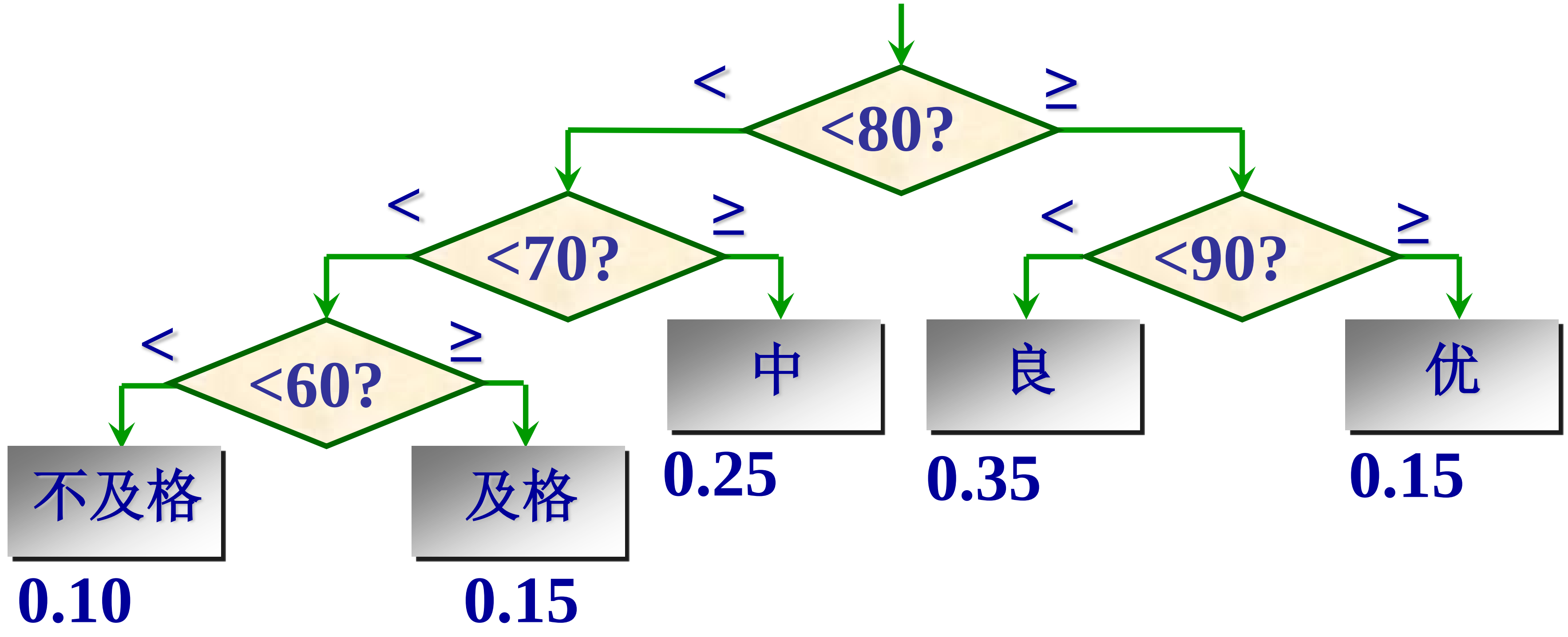
Adjusting Decision Tree



WPL = 0.10*3+0.15*3+0.25*2+0.35*2+0.15*2

= 0.3+0.45+0.5+0.7+0.3 = 2.25

Decision Tree



WPL = 0.10*3+0.15*3+0.25*2+0.35*2+0.15*2

= 0.3+0.45+0.5+0.7+0.3 = 2.25

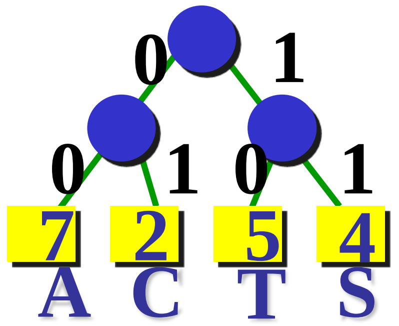
Huffman Coding

- 主要用途是实现数据压缩

- 设给出一段报文：

  CAST CAST SAT AT A TASA
- 字符集合是 { C, A, S, T }，各个字符出现的频度（次数）是 W＝{ 2, 7, 4, 5 }。
- 若给每个字符以等长编码（2位二进制足够）

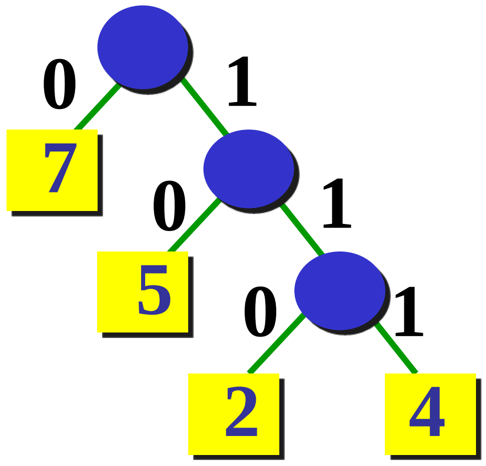
  A : 00 T : 10 C : 01 S : 11
- 则总编码长度为 (2+7+4+5) * 2 = 36
- *能否减少总编码长度，使得发出同样报文，可以用最少的二进制代码？*



- 若按各个字符出现的概率不同而给予不等长编码，可望减少总编码长度。
- 各字符出现概率为{ 2/18, 7/18, 4/18, 5/18 }, 化整为 { 2, 7, 4, 5 }。
- 以它们为各叶结点上的权值，建立Huffman树。左分支赋0，右分支赋 1，得Huffman编码(变长编码)：

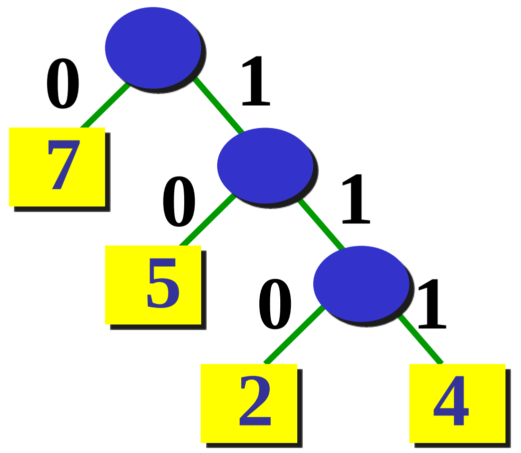
  A : 0　T : 10　C : 110　S : 111

- 它的总编码长度：7*1+5*2+(2+4)*3 = 35。比等长编码的情形要短。
- *总编码长度正好等于Huffman树的带权路径长度WPL。*
- Huffman编码是一种前缀编码，即任一个二进制编码不是其他二进制编码的前缀。解码时不会混淆。



Huffman编码树

- 它的总编码长度：7*1+5*2+(2+4)*3 = 35。比等长编码的情形要短。
- 总编码长度正好等于Huffman树的带权路径长度WPL。
- Huffman编码是一种前缀编码，即任一个二进制编码不是其他二进制编码的前缀。解码时不会混淆。



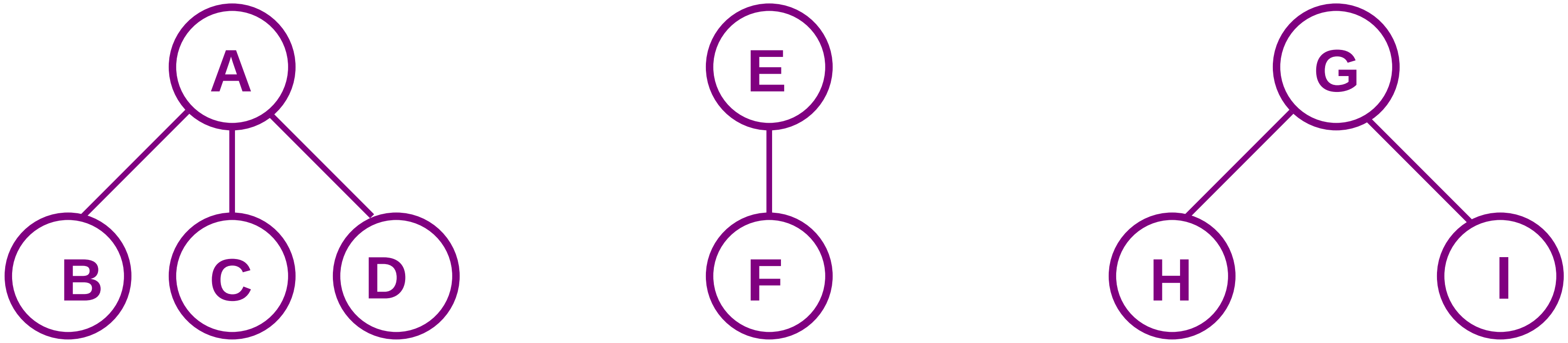
Huffman编码树

Trees

- Trees
- Binary Trees
  - Definition and Features
  - Binary Tree Representations
  - Traversal
  - Threaded Binary Trees
- Heaps and Binary Search Trees
  - Heaps
  - Huffman Trees
- Forrests
  - Tree Representations
  - Transforming between Trees and Forrests
  - Traversal

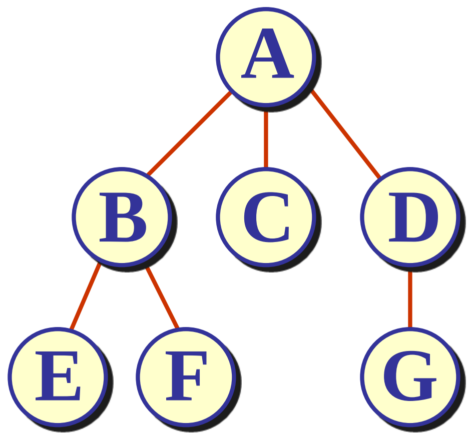
Forests

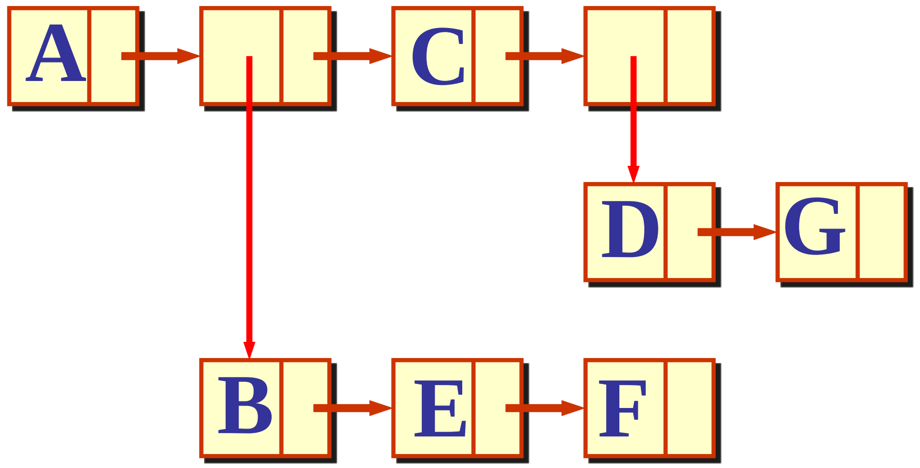
A forest is a set of n≥0 disjoint trees.



Tree Representation

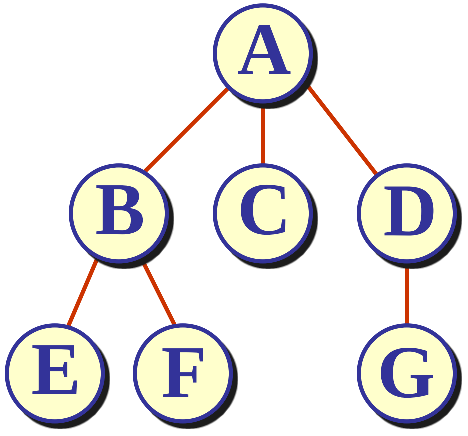
- 广义表表示





A(B(E, F), C, D(G)) 结点的utype域没有画出

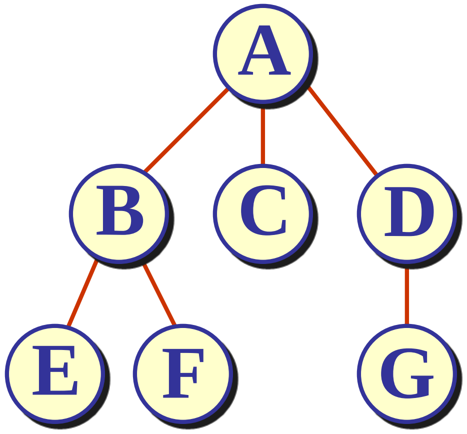
双亲表示

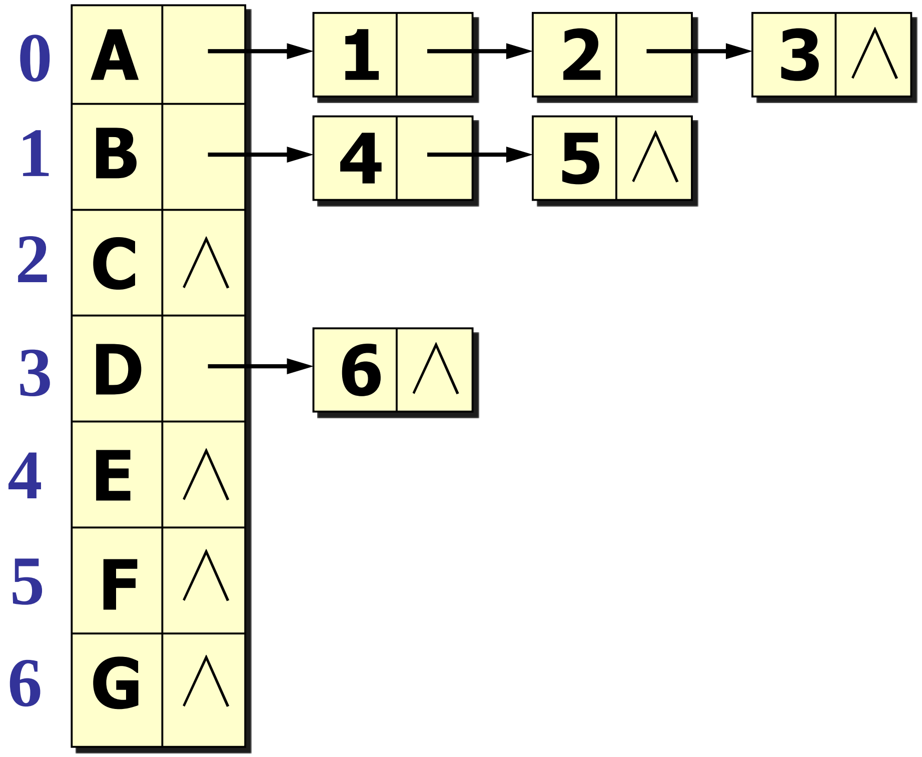


	0	1	2	3	4	5	6
data	A	B	C	D	E	F	G
parent	−1	0	0	0	1	1	3

- 树中结点的存放顺序一般不做特殊要求，但为了操作实现的方便，有时也会规定结点的存放顺序。例如，可以规定按树的前序次序存放树中的各个结点，或规定按树的层次次序安排所有结点。

子女链表表示



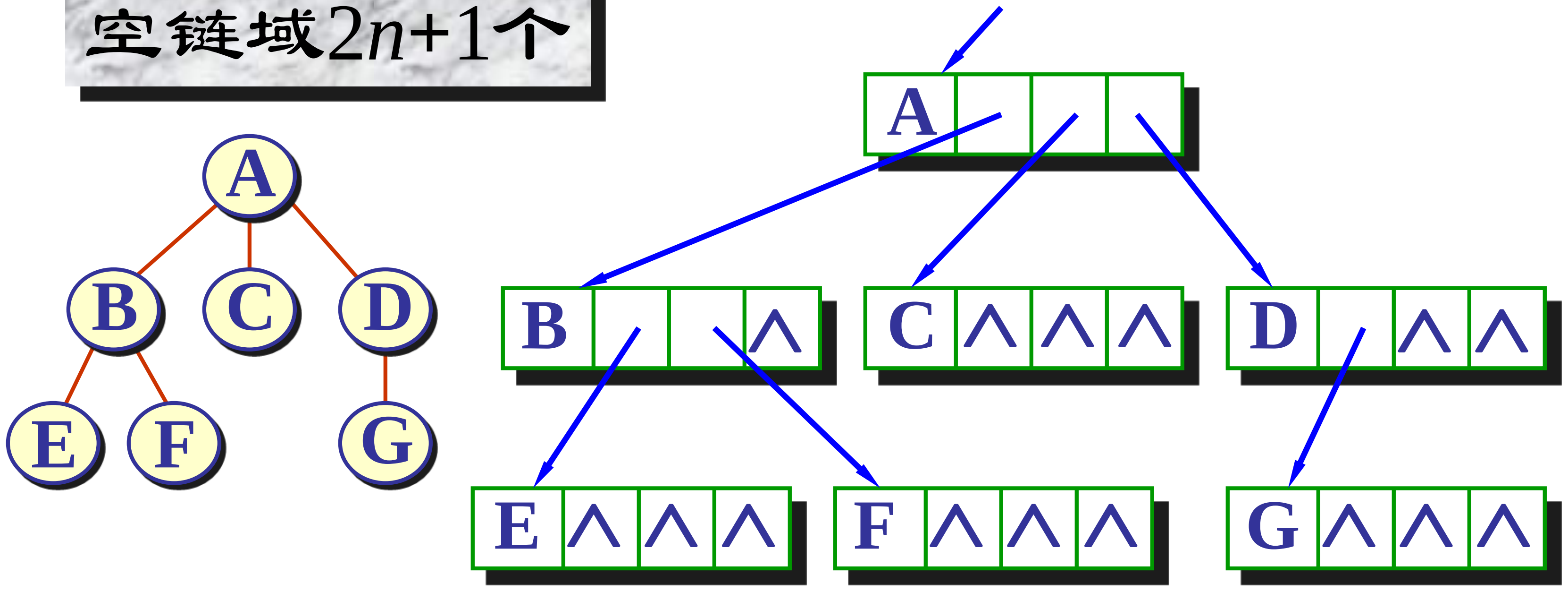


- 无序树情形链表中各结点顺序任意，有序树必须自左向右链接各个子女结点。

子女指针表示

- 一个合理的想法是在结点中存放指向每一个子女结点的指针。但由于各个结点的子女数不同，每个结点设置数目不等的指针，将很难管理
- 设置等长的结点，每个结点包含的指针个数相等
- 保证结点有足够的指针指向它的所有子女结点。但可能产生很多空闲指针，造成存储浪费

空链域$2n+1$个



等数量的链域

data	$child_1$	$child_2$	$child_3$	······	$child_d$

子女-兄弟表示

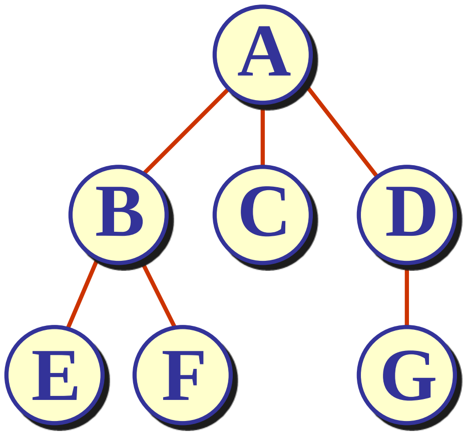
- 也称为树的二叉树表示。结点构造为：

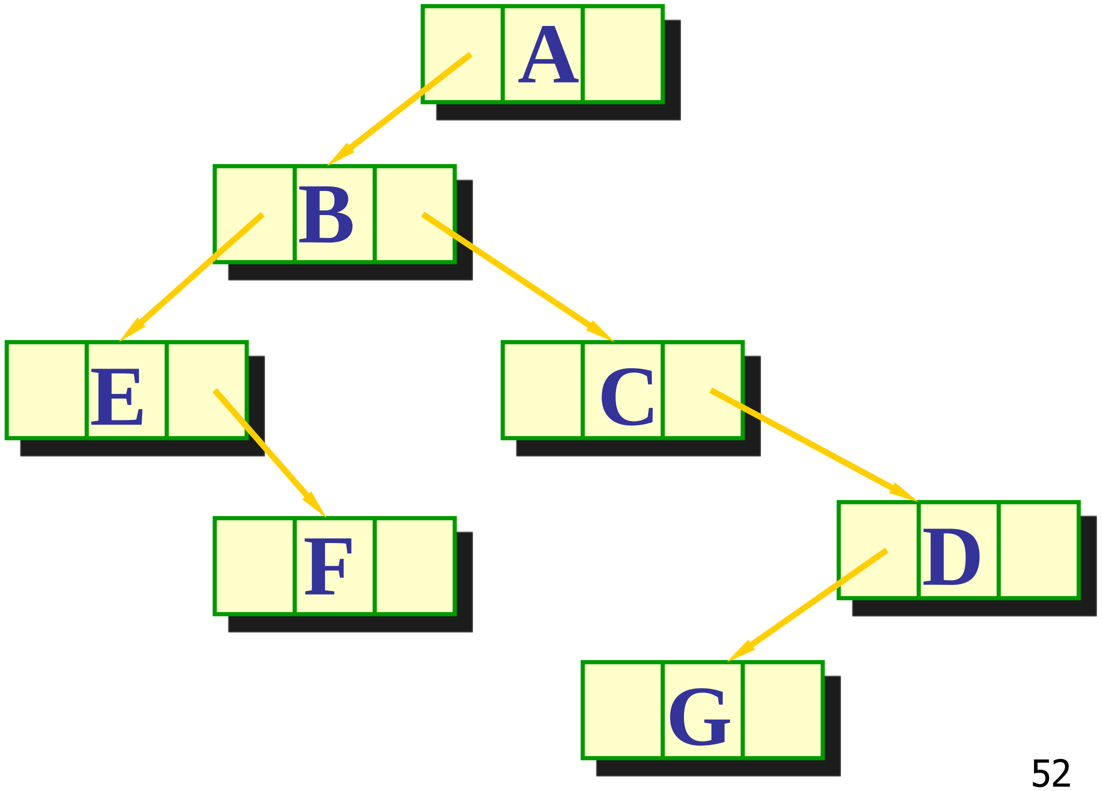
data	**firstChild**	**nextSibling**

- firstChild 指向该结点的第一个子女结点。无序树时，可任意指定一个结点为第一个子女。
- nextSibling 指向该结点的下一个兄弟。任一结点在存储时总是有顺序的。
- 若想找某结点的所有子女，可先找firstChild,再反复用nextSibling 沿链扫描。

树的子女-兄弟表示

data	firstChild	nextSibling





Trees

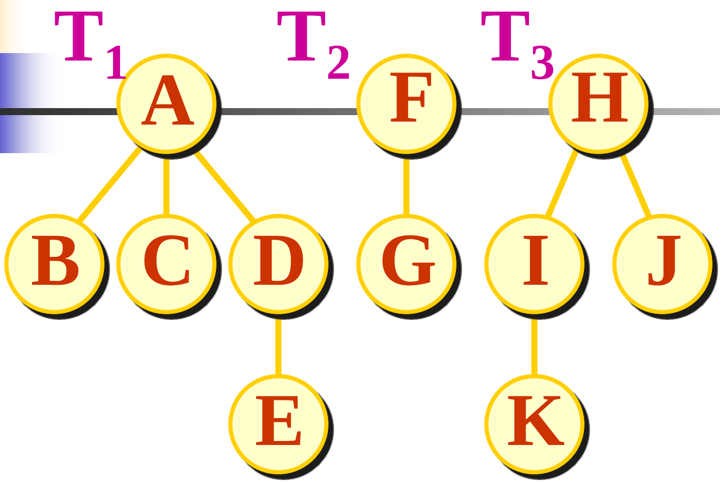
- Trees
- Binary Trees
  - Definition and Features
  - Binary Tree Representations
  - Traversal
  - Threaded Binary Trees
- Heaps and Binary Search Trees
  - Heaps
  - Huffman Trees
- Forrests
  - Tree Representations
  - Transforming between Trees and Forrests
  - Traversal

Transforming a Forrest into a Binary Tree

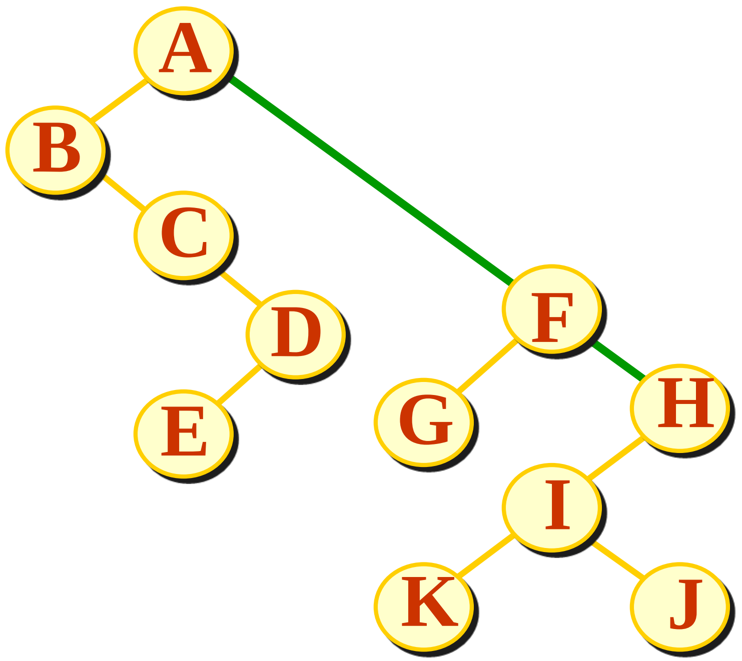
Definition: If $T_1, \ldots, T_n$ is a forest of trees, then the binary tree corresponding to it , denoted by $B(T_1, \ldots, T_n)$,

(1) is empty if n=0

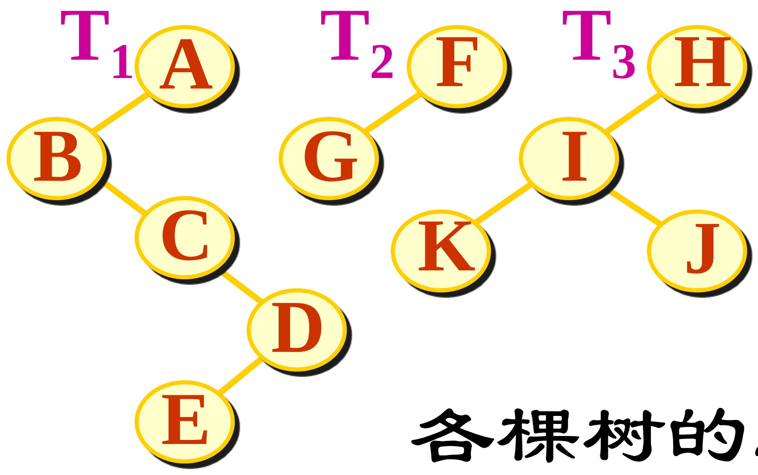
(2) has root equal to root(T_1); has left subtree equal to $B(T_{11}, \ldots, T_{1m})$, where $T_{11}, \ldots, T_{1m}$ are the subtrees of root(T_1); and has right subtree $B(T_2, \ldots, T_n)$.



3 棵树的森林



森林的二叉树表示



各棵树的二叉树表示

Transforming from Forrest to Trees

1. 若 F 为空，即 $n = 0$，则对应的二叉树 B 为空树
2. 若 F 不空，则
   - ✓ 二叉树 B 的根是 F 第一棵树 T_1 的根；
   - ✓ 其左子树为$B(T_{11}, T_{12}, \ldots, T_{1m})$，其中，$T_{11}, T_{12}, \ldots, T_{1m}$ 是 T_1 的根的子树；
   - ✓ 其右子树为 $B(T_2, T_3, \ldots, T_n)$，其中，$T_2, T_3, \ldots, T_n$ 是除 T_1 外其它树构成的森林。

Transforming from Trees to Forrest

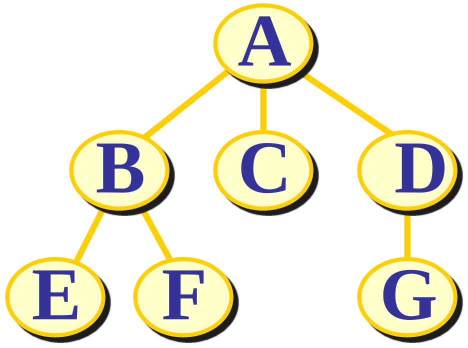
1. 如果 B 为空，则对应的森林 F 也为空。
2. 如果 B 非空，则
   - ✓ F 中第一棵树 T_1 的根为 B 的根；
   - ✓ T_1 的根的子树森林 $\{ T_{11}, T_{12}, \ldots, T_{1m} \}$ 是由 B 的根的左子树 LB 转换而来；
   - ✓ F 中除了 T_1 之外其余的树组成的森林 $\{ T_2, T_3, \ldots, T_n \}$ 是由 B 的根的右子树 RB 转换而成的森林。

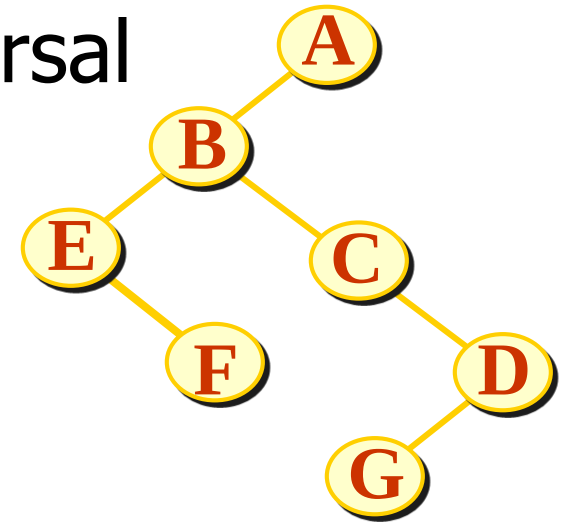
Trees

- Trees
- Binary Trees
  - Definition and Features
  - Binary Tree Representations
  - Traversal
  - Threaded Binary Trees
- Heaps and Binary Search Trees
  - Heaps
  - Huffman Trees
- Forrests
- Tree Representations
  - Transforming between Trees and Forrests
  - Traversal

Forrest Traversal

- Depth First Traversal
  - Preorder Traversal
  - Postorder Traversal
- Breadth First Traversal





森林的二叉树表示

DFT

- 给定森林 F，若 $F = \varnothing$，则遍历结束。
- 否则，

  若$F = \{\{T_1 = \{ r_1, T_{11}, \ldots, T_{1k} \}, T_2, \ldots, T_m\}$，则可以导出先根遍历、后根遍历两种方法。

其中， r_1是第一棵树的根结点，

$\{T_{11}, \ldots, T_{1k}\}$是第一棵树的子树森林，

$\{T_2, \ldots,T_m\}$是除去第一棵树之后剩余的树构成的森林。

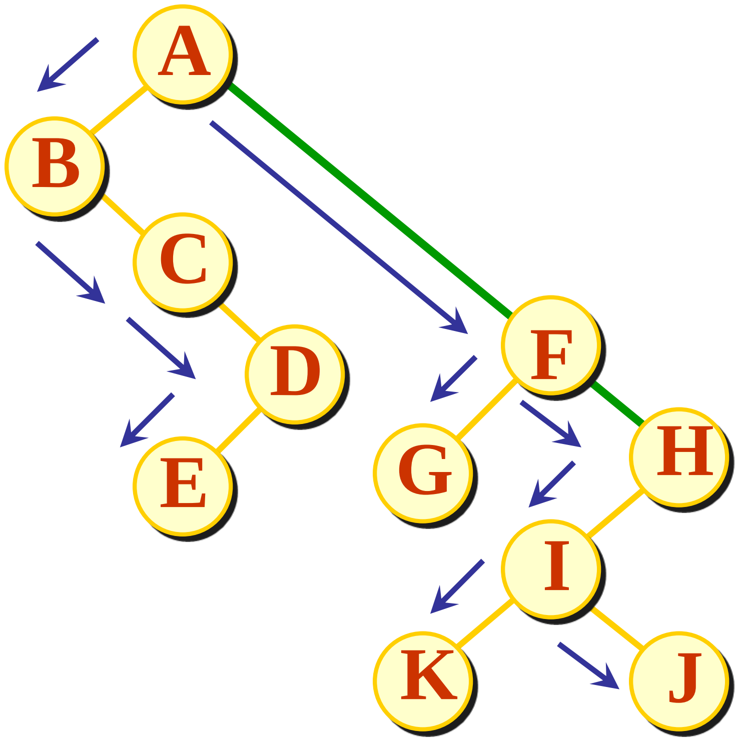
Preorder Traversal

- 若森林F = Ø，返回；
- 否则
  - ✓ 访问森林的根（也是第一棵树的根）r_1；
  - ✓ 先根遍历森林第一棵树的根的子树森林$\{T_{11}, \ldots, T_{1k}\}$；
  - ✓ 先根遍历森林中除第一棵树外其他树组成的森林$\{T_2, \ldots, T_m\}$。

- 森林的先根次序遍历的结果序列

  ABCDE FG HIKJ

- 这相当于对应二叉树的前序遍历结果



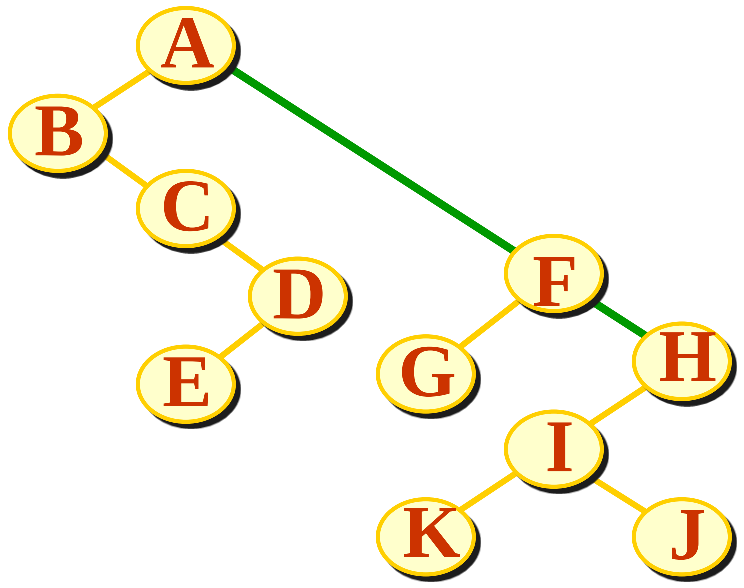
Postorder Traversal

- 若森林 $F = \varnothing$，返回；
- 否则
  - ✓ 后根遍历森林 F 第一棵树的根结点的子树森林$\{T_{11}, \ldots, T_{1k}\}$；
  - ✓ 访问森林的根结点 r_1；
  - ✓ 后根遍历森林中除第一棵树外其他树组成的森林$\{T_2, \ldots, T_m\}$。

- 森林的后根次序遍历的结果序列

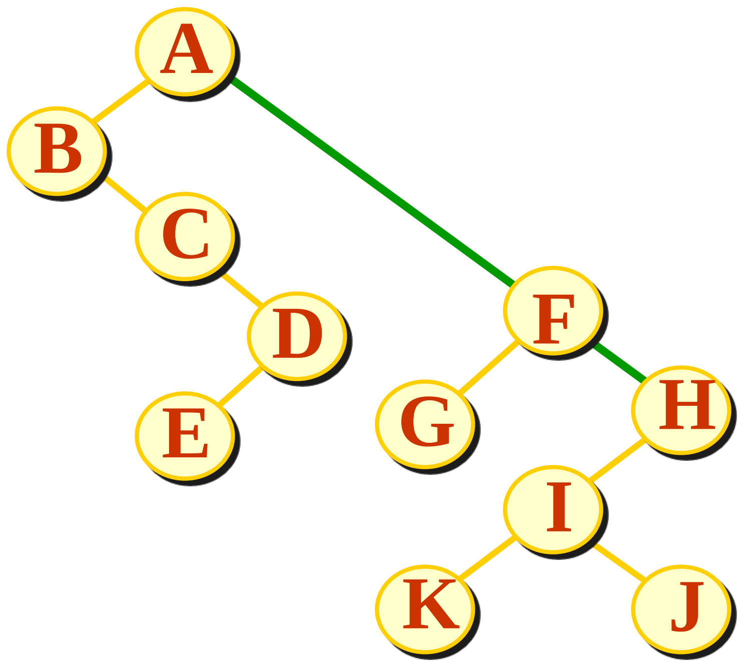
  BCEDA GF KIJH

- 这相当于对应二叉树中序遍历的结果



BFT

- 若森林 *F* 为空，返回；否则
  - ✓ 依次遍历各棵树的根结点；
  - ✓ 依次遍历各棵树根结点的所有子女；
  - ✓ 依次遍历这些子女结点的子女结点；
  - ✓ ……



AFH BCDGIJ EK

并查集

- 是一种简单的集合表示，支持以下3种操作：

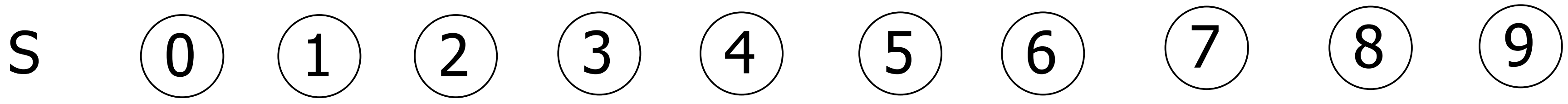
（1）Union(S,Root1,Root2):把集合S中的子集合Root2并入子集合Root1。要求Root1和Root2互不相交，否则不执行合并。

（2）Find(S,X):查找集合S中单元素x所在的子集合，并返回该子集合的名字。

（3）Initial(S):将集合S中的每个元素都初始化为只有一个单元素的子集合。

并查集的存储结构

- 通常用树（森林）的双亲表示
- 每个子集合用一棵树表示
- 所有表示子集合的树，构成表示全集合的森林，存放在双亲表示数组内
- 用数组元素的下标代表元素名
- 用根节点的下标代表子集合名
- 根节点的双亲结点为负数

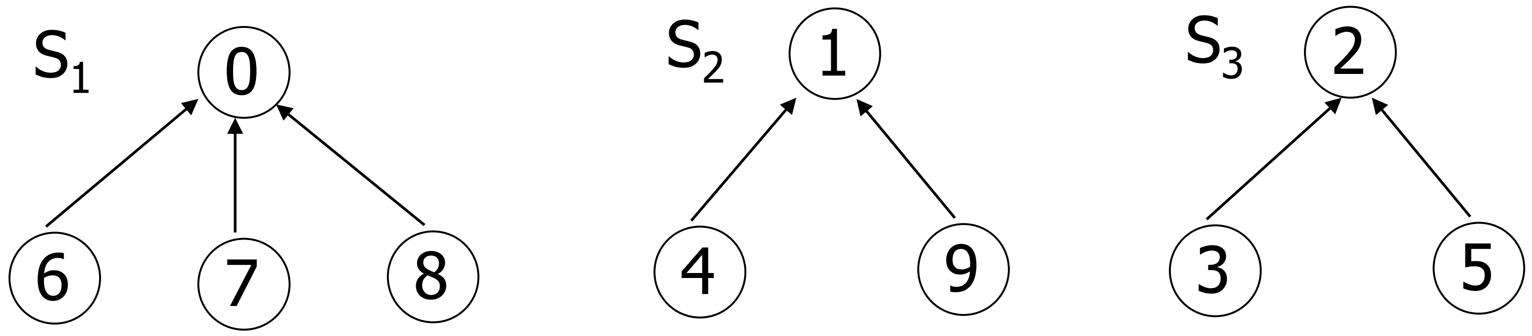


全集合S初始化时形成一个森林

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

初始化时形成一个森林双亲表示

- 形成3个更大的子集合S_1={0，6，7，8} S_2={1，4，9}，S_3={2，3，5}



集合的树形表示

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

双亲表示

The END