The Fourth Course

Binary Trees

Binary Trees

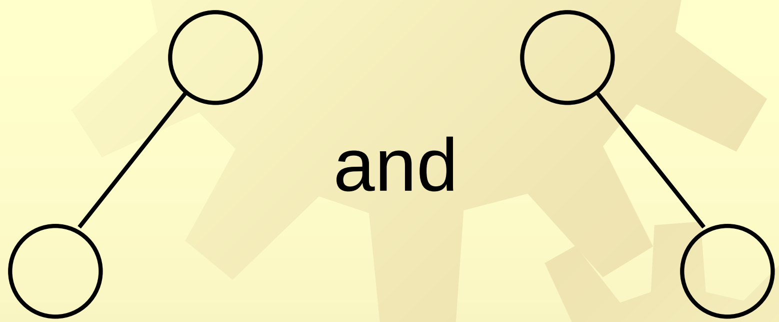
- DEFINITION:
  A binary tree is either *empty*, or it consists of a node called the *root* together with two binary trees called the *left subtree* and the *right subtree* of the root, , which are *disjoint* from each other and from the root.

Notation

- *Node*, *Children*, *Edge*,
- *Parent*, *Ancestor*, *Descendant*,
- *Path*, *Length (the number of edges)*,
- *Depth(0)*, *Height(1)*, *Level(0)*,
- *Leaf Node*, *Internal Node*,
- *Subtree*, *degree*

Binary Trees

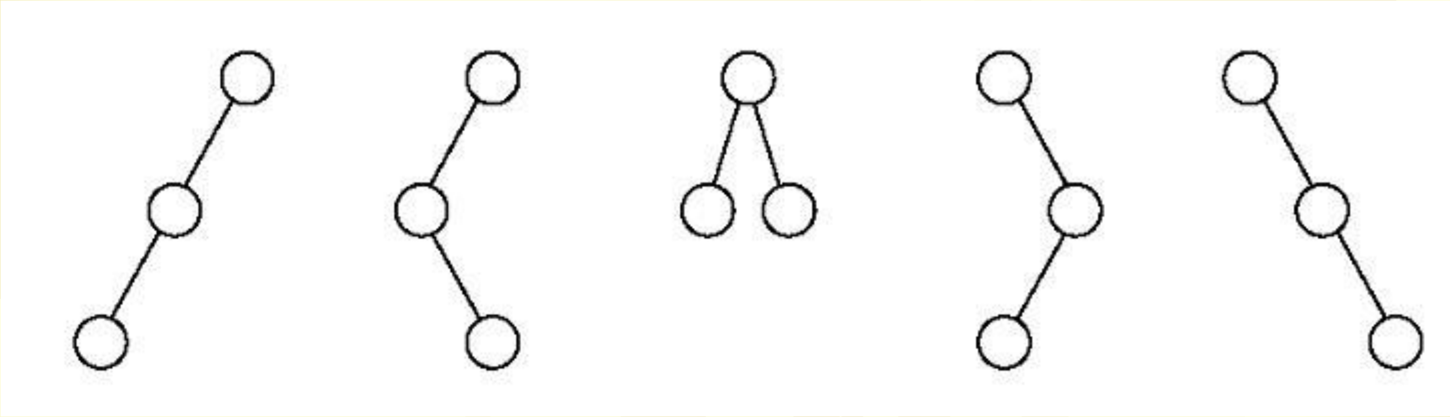
- There is one empty binary tree, one binary tree with one node, and two with two nodes:



- These are different from each other.

Binary Trees

- The binary trees with three nodes are:



计算具有 n 个结点的不同二叉树的棵数

- *Catalan*函数

$$b_n = \frac{1}{n+1} C_{2n}^{n} = \frac{1}{n+1} \frac{(2n)!}{n! \cdot n!}$$

- 例 $b_3 = \frac{1}{3+1} C_6^3 = \frac{1}{4} \frac{6 \cdot 5 \cdot 4}{3 \cdot 2 \cdot 1} = 5$

$$b_4 = \frac{1}{4+1} C_8^4 = \frac{1}{5} \frac{8 \cdot 7 \cdot 6 \cdot 5}{4 \cdot 3 \cdot 2 \cdot 1} = 14$$

二叉树的性质

性质1

若二叉树的层次从 0 开始，则在二叉树的第 i 层最多有 2^i 个结点。(i≥0)

[证明用数学归纳法]

- i = 0时，根结点只有1个，$2^i = 2^0 = 1$；
- 若设 $i = k$ 时性质成立，即该层最多有 2^k 个结点，则当 $i = k+1$ 时，由于第 k 层每个结点最多可有 2 个子女，第 $k+1$ 层最多结点个数可有 $2*2^k = 2^{k+1}$ 个，故性质成立。

性质2

高度为 h 的二叉树最多有 2^h-1个结点。(h≥1)

[证明用求等比级数前k项和的公式]

高度为 h 的二叉树有 h 层，各层最多结点个数相加，得到等比级数，求和得：

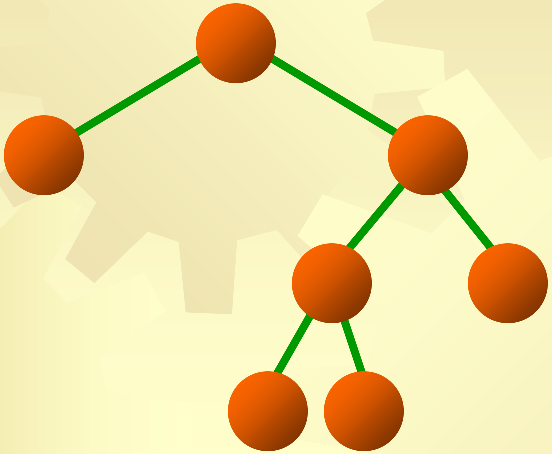
$$2^0 + 2^1 + 2^2 + \ldots + 2^{h-1} = 2^h - 1$$

- 空树的高度为 0，只有根结点的树的高度为 1。

Full and Complete Binary Trees

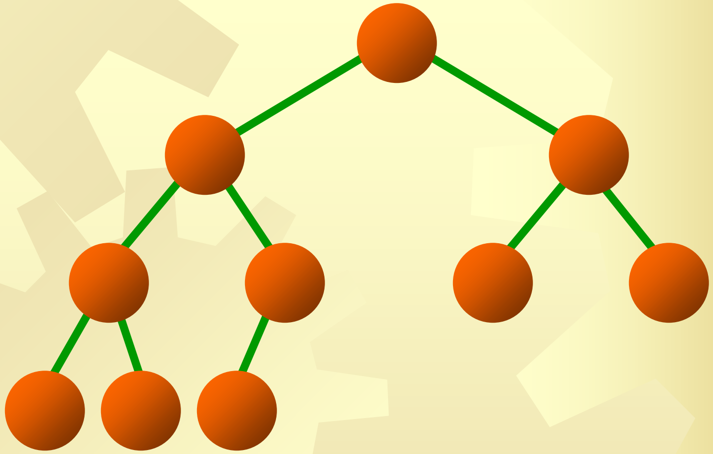
- *Full* binary tree: each node either is a leaf or is an internal node with exactly two non-empty children.

- *Complete* binary tree: If the height of the tree is d, then all levels except possibly level d are completely full. The bottom level has all nodes to the left side.



Full tree

Complete tree



Full Binary Tree Theorem

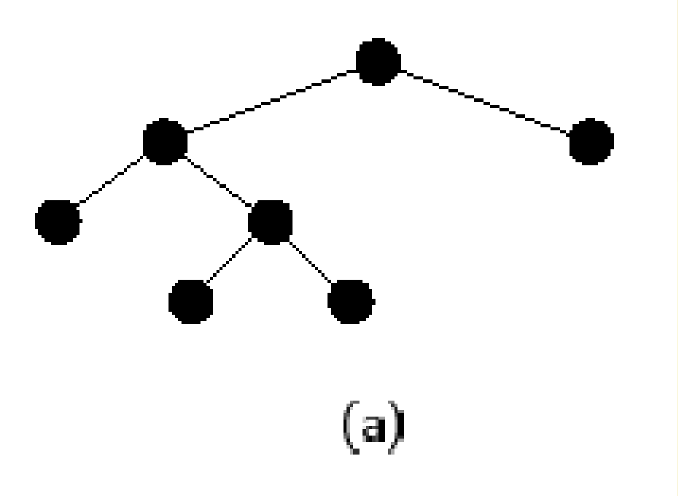
- Theorem: The number of leaves in a non-empty full binary tree is one more than the number of internal nodes.

That is :

$$n_0=n_2+n_1+1$$

$$n_1=0$$

$$\Rightarrow \quad n_0=n_2+1$$



Full tree

$n_0=4$

$n_2=3$

$n_0=n_2+1$

Full Binary Tree Theorem

- Proof (by Mathematical Induction):
  - **Base Case: A full binary tree with 1 internal node must have two leaf nodes.**
  - **Induction Hypothesis: Assume any full binary tree T containing n-1 internal nodes has n leaves.**

Full Binary Tree Theorem

- **Induction Step:** **Given tree T with n internal nodes, pick internal node I with two leaf children. Remove I's children, call resulting tree T'. By induction hypothesis, T' is a full binary tree with n leaves.**

  Restore i's two children. The number of internal nodes has now gone up by 1 to reach n. The number of leaves has also gone up by 1.

Full Binary Tree Theorem Corollary

- **Theorem:** The number of NULL pointers in a non-empty binary tree is one more than the number of nodes in the tree.

- **Proof:** Replace all null pointers with a pointer to an empty leaf node. This is a full binary tree.

性质4

具有 n ($n \geq 0$) 个结点的完全二叉树的高度为 $\lceil \log_2(n+1) \rceil$

证明：设完全二叉树的高度为 h，则有

$$\underbrace{2^{h-1}-1}_{\text{上面}h-1\text{层结点数}} < n \leqslant \underbrace{2^h-1}_{\text{包括第}h\text{层的最大结点数}}$$

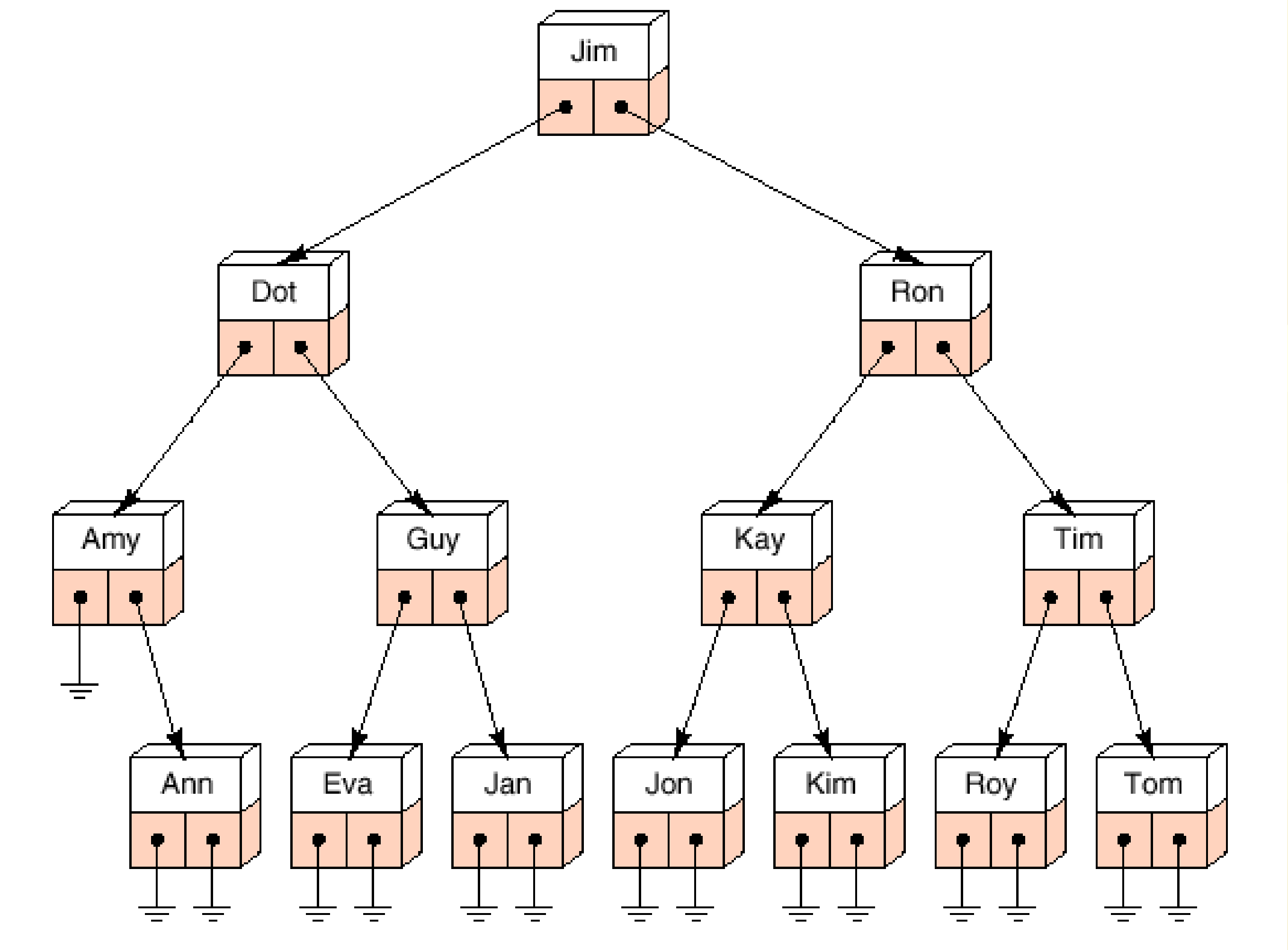
变形 $2^{h-1} < n+1 \leqslant 2^h$

取对数

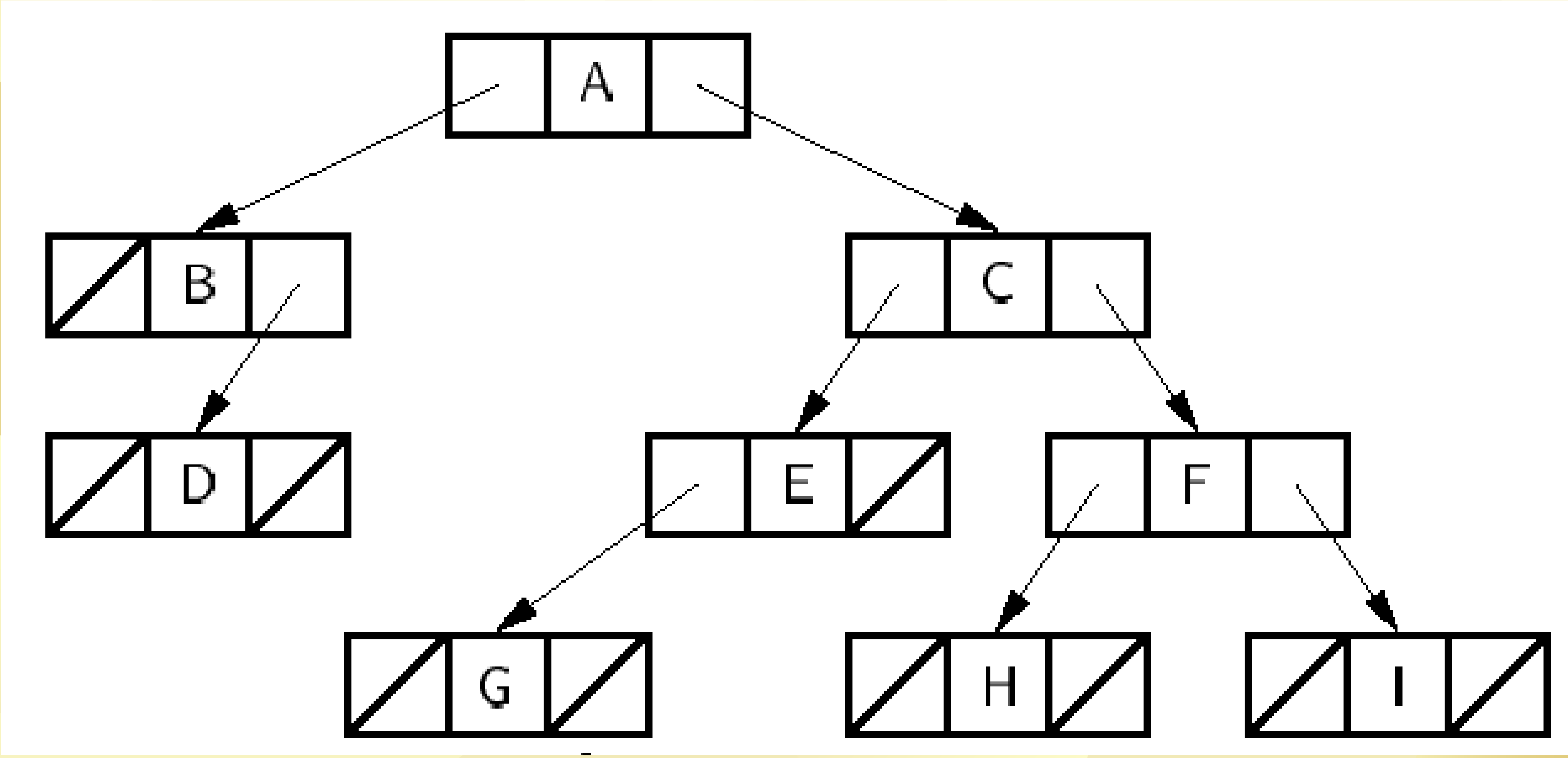
$$h-1 < \log_2(n+1) \leqslant h$$

有 $h = \lceil \log_2(n+1) \rceil$

Linked implementation of binary tree



Binary Tree Implementation

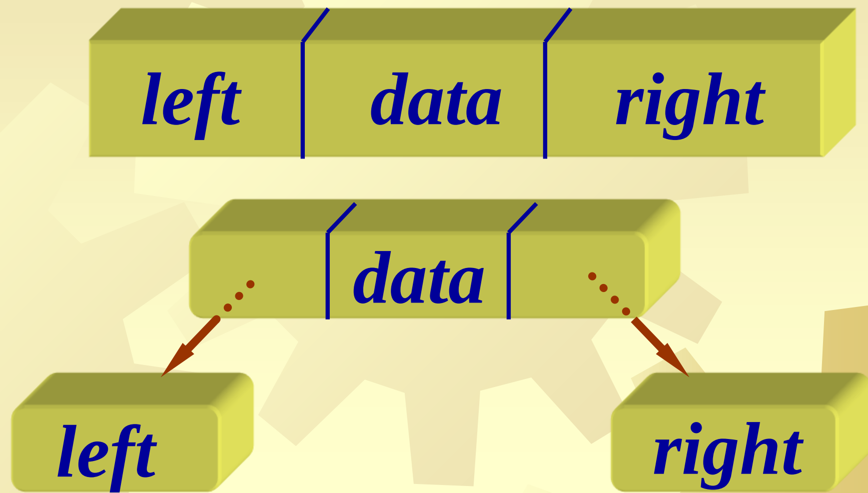


Linked Binary Tree Specifications

- Binary tree class
  -
```
template <class Entry>
class Binary_tree {
public:
// Add methods here.
protected:
        // Add auxiliary function prototypes here.
        Binary_node<Entry> *root;
};
```

Linked Binary Tree Specifications

- Binary node class




```
template <class Entry>
class Binary_node {
public:
  // data members
        Entry data;
        Binary_node<Entry> *left;
        Binary_node<Entry> *right;
  // constructors
        Binary_node( );
        Binary_node(const Entry &x);
};
```

Constructor

- template <class Entry>

```
Binary_tree<Entry> ::Binary_tree( )
/* Post: An empty binary tree has been created. */
{
    root = NULL;
}
```

Empty

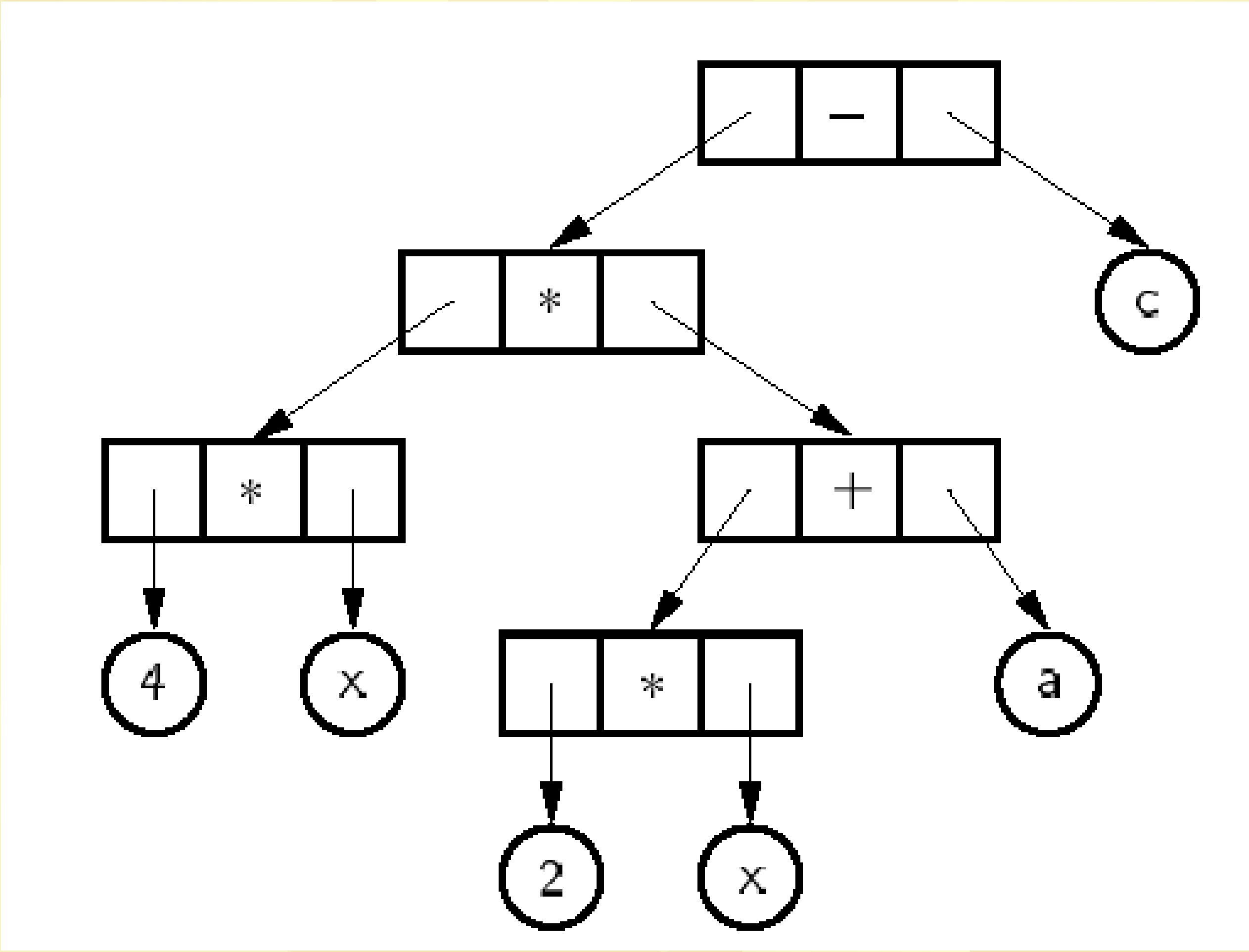
- template <class Entry>

```
bool Binary_tree<Entry> ::empty( ) const
/* Post: A result of true is returned if the binary tree is
empty. Otherwise, false is returned. */
{
    return root == NULL;
}
```

Space Overhead

- From Full Binary Tree Theorem:
  - **Half of pointers are NULL.**
  - **If leaves only store information, then overhead depends on whether tree is full.**
  - **All nodes the same, with two pointers to children:**
    - **Total space required is (2p +d)n.**
    - **Overhead: 2pn.**
    - **If p = d, this means 2p/(2p +d) =2/3 overhead.**

- (4x * (2x + a)) - c)



Space Overhead

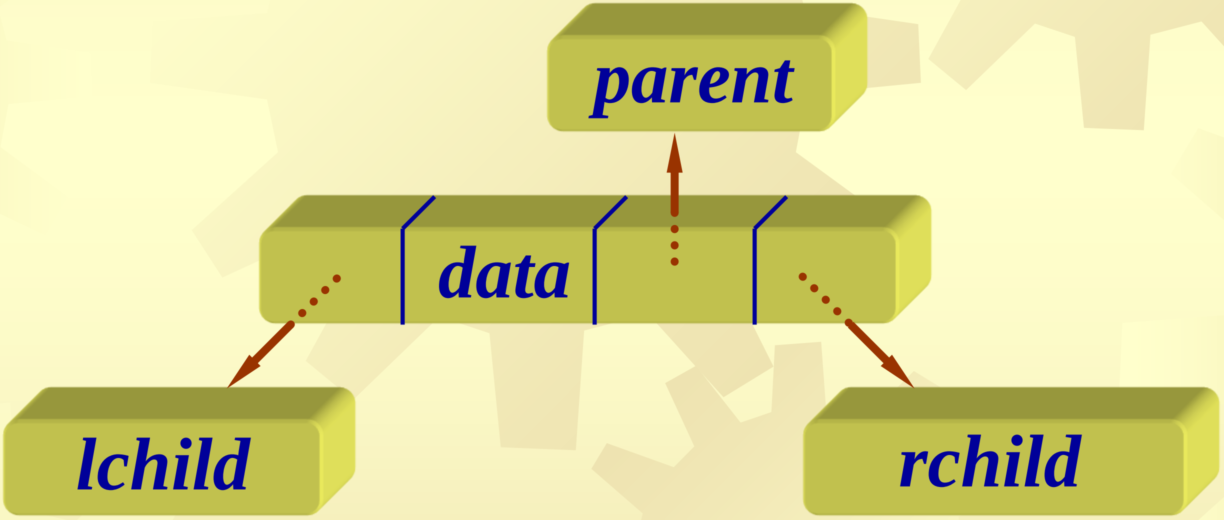
- Eliminate pointers from leaf nodes:

$$\frac{\frac{n}{2}(2p)}{\frac{n}{2}(2p)+dn} = \frac{p}{p+d}$$

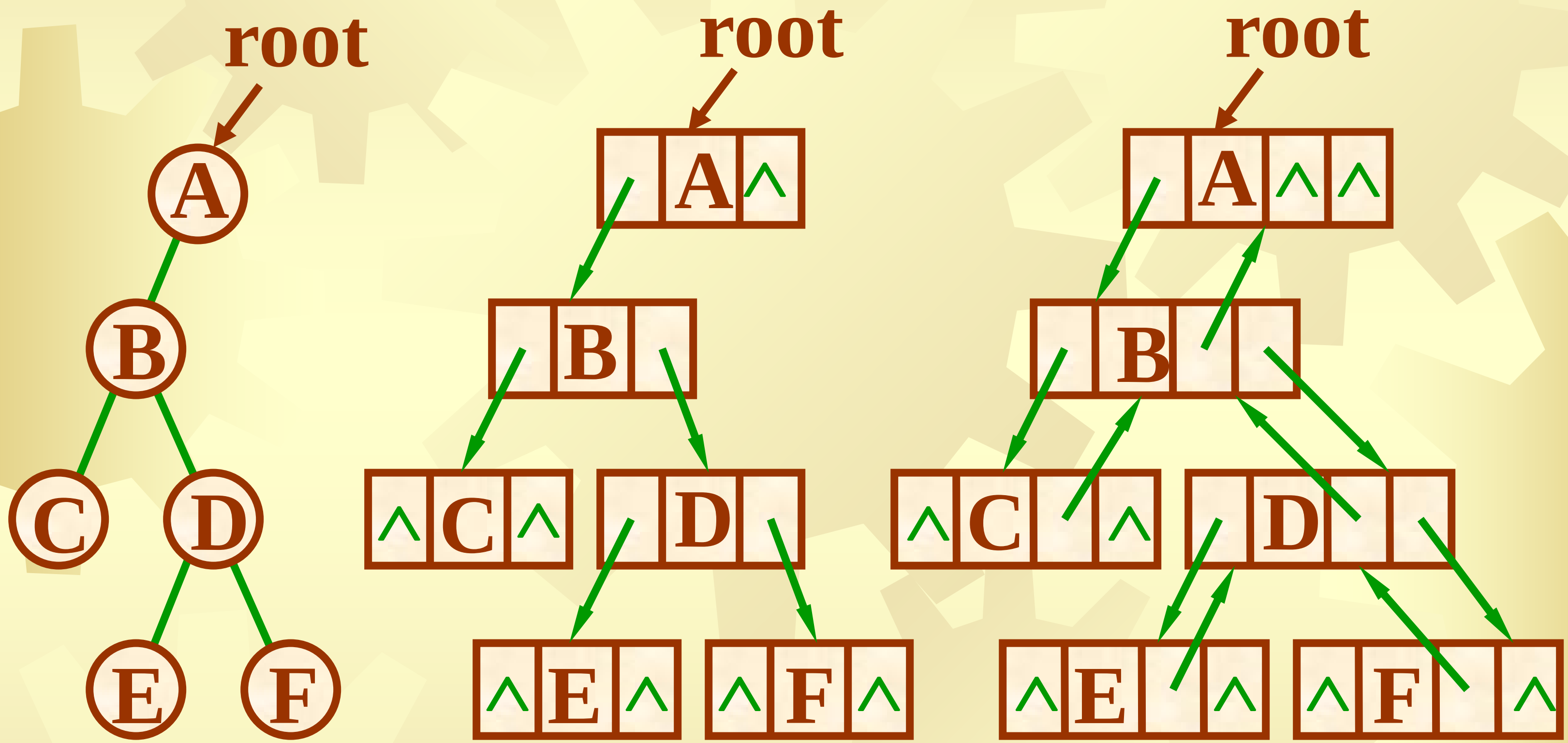
- **This is 1/2 if p = d.**
- **2p/(2p +d) if data only at leaves⇒2/3 overhead.**
- **Some method is needed to distinguish leaves from internal nodes.**

二叉树的三叉链表表示


lchild
data
parent
rchild
左子女指针
双亲指针
右子女指针


parent
data
lchild
rchild

二叉树链表表示的示例



二叉树　二叉链表　三叉链表

Traversals

- Any process for visiting the nodes in some order is called a *traversal*.
- A _traversal_ is a manner of _visiting_ each node in a tree once. What you do when visiting any particular node depends on the application;
  - **you might print a node's value**
  - **perform some calculation upon it.**
- There are several different traversals, each of which orders the nodes differently.
- Any traversal that lists every node in the tree exactly once is called an *enumeration* of the tree's nodes.

Traversal of Binary Trees

- At a given node there are three tasks to do in some order: Visit the node itself (V); traverse its left subtree (L); traverse its right subtree (R).
- There are six ways to arrange these tasks:
  - V L R
  - L V R
  - L R V
  - V R L
  - R V L
  - R L V.
  - **By standard convention, these are reduced to three by considering only the ways in which the left subtree is traversed before the right.**

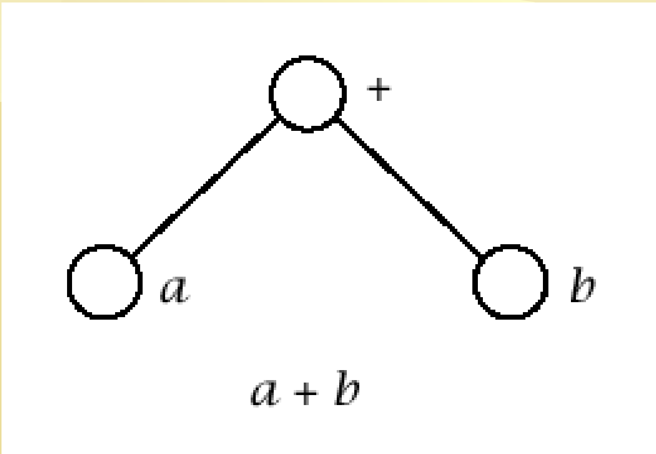
Traversal of Binary Trees

- Many traversals can be defined recursively.
- In a _preorder_ traversal, you visit each node before recursively visiting its children, which are visited from left to right. The root is visited first.
- Each node is visited only once, so a preorder traversal takes O(n) time, where n is the number of nodes in the tree. All the traversals we will consider take O(n) time.

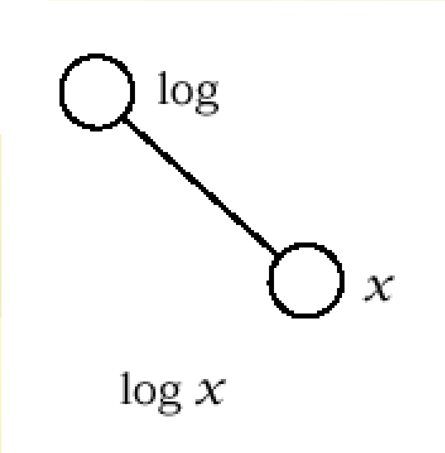
Traversal of Binary Trees

- These three names are chosen according to the step at which the given node is visited.
  - **With *preorder traversal* we first visit a node, then traverse its left subtree, and then traverse its right subtree.**
  - **With *inorder traversal* we first traverse the left subtree, then visit the node, and then traverse its right subtree.**
  - **With *postorder traversal* we first traverse the left subtree, then traverse the right subtree, and finally visit the node.**

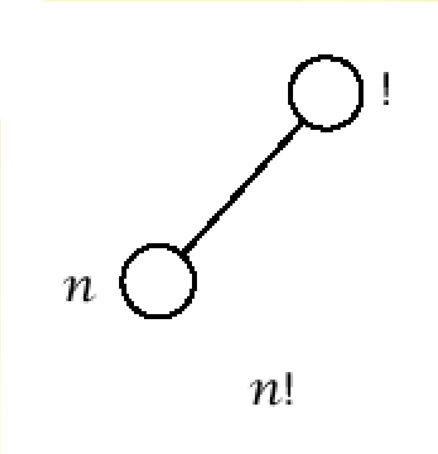
Expression Trees



Expression: a+b
Preorder : + a b
Inorder : a + b
Postorder : a b +

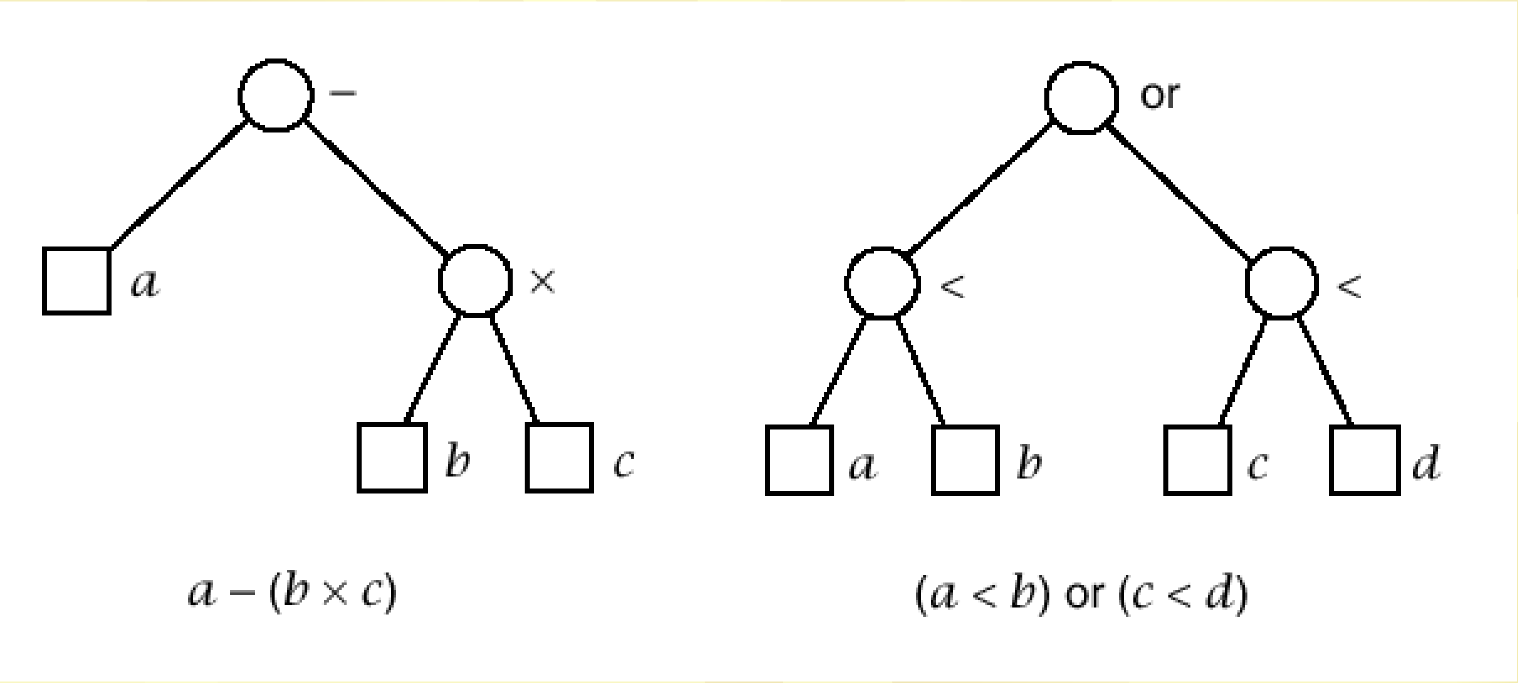


Expression: log x
Preorder : log x
Inorder : log x
Postorder : x log



Expression: n!
Preorder :! n
Inorder : n!
Postorder : n !

Expression Trees



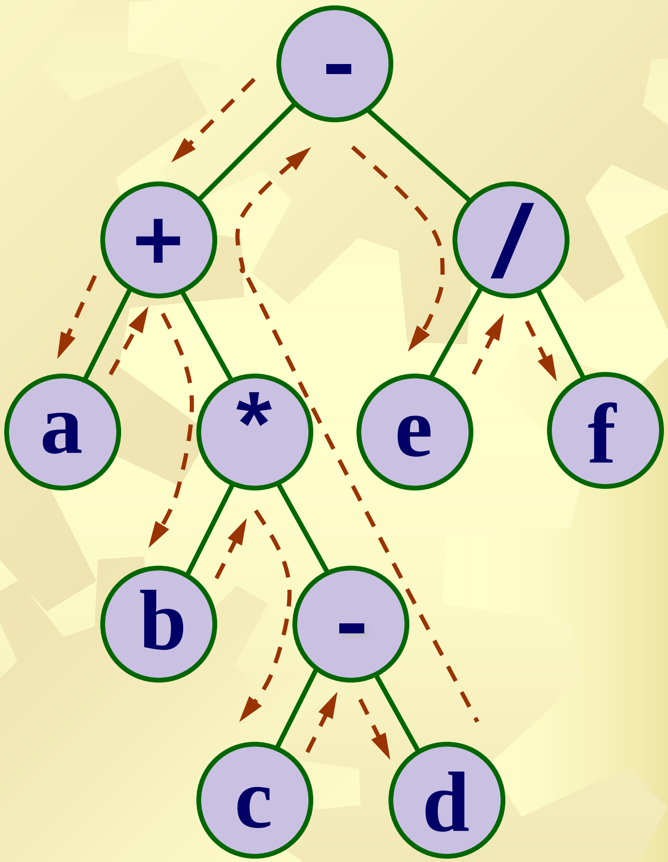
Expression: a - (b × c)
Preorder : - a × b c
Inorder : a - b × c
Postorder : a b c × -

Expression: (a < b) or (c < d)
Preorder : or < a b < c d
Inorder : a < b or c < d
Postorder : a b < c d < or

Inorder traversal

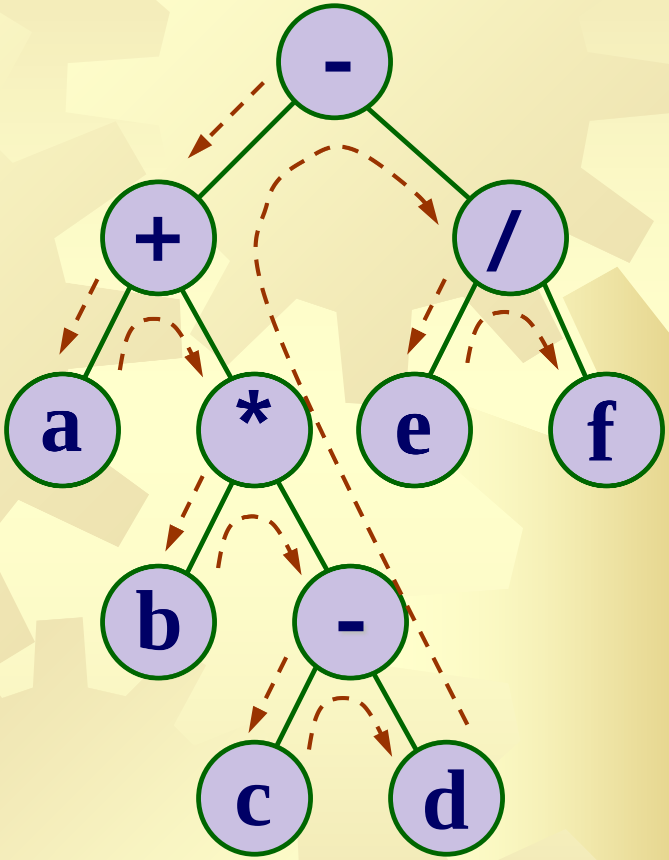
- Result:

  a + b * c - d - e / f



Preorder traversal

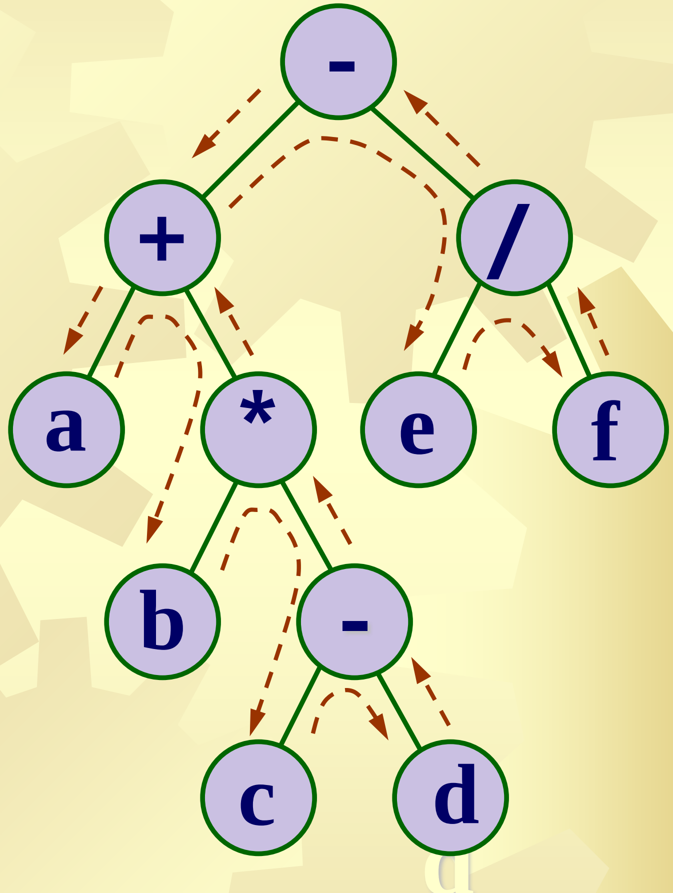
- Result:
  - + a * b - c d / e f



Postorder traversal

- Result:

  a b c d - * + e f / -

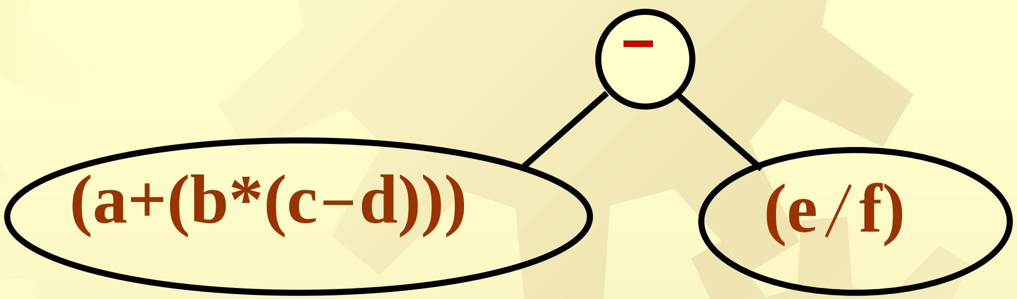


从 a+b*(c−d)−e/f 生成表达式树

(1) 先根据运算符的优先级对表达式加括号

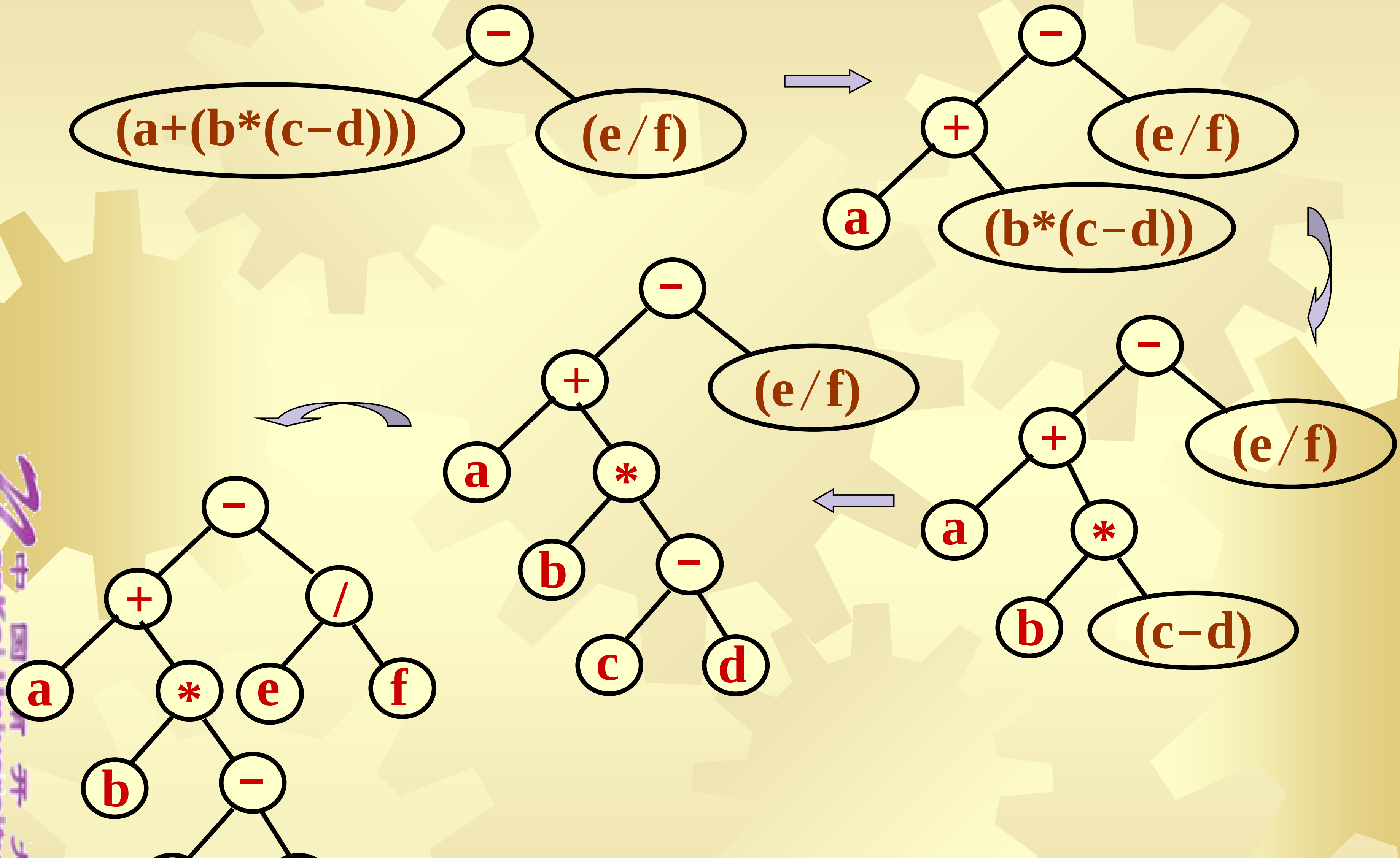
((a+(b*(c−d)))−(e/f))

(2) 脱一层括号，(a+(b*(c−d))) − (e/f)，取两个括号中间的“−”为根，将表达式分为两部分，左子树是(a+(b*(c−d)))，右子树为(e/f)



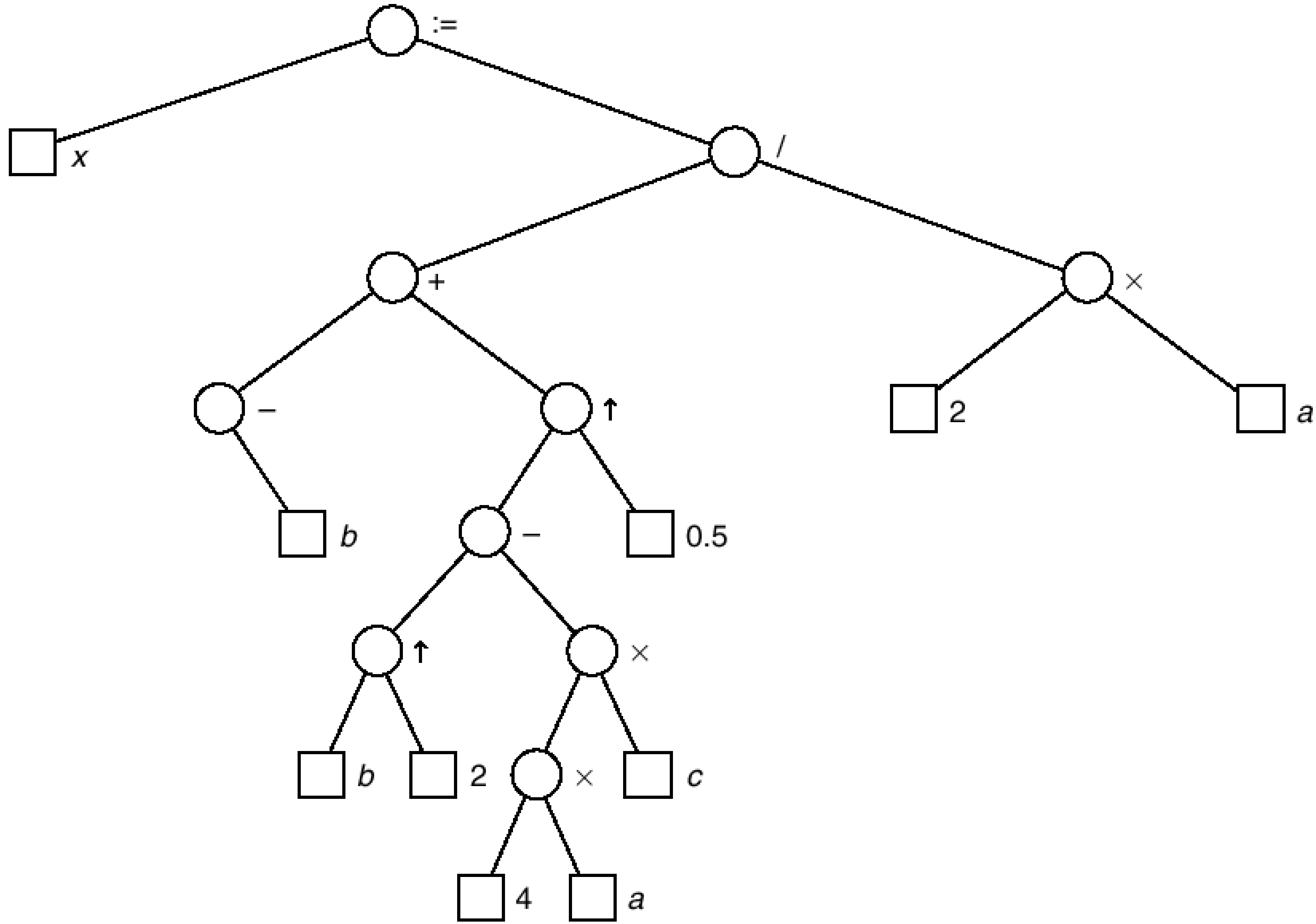
(3) 对左子树递归执行步骤(2)；//若树空则不再递归

(4) 对右子树递归执行步骤(2)；//若树空则不再递归

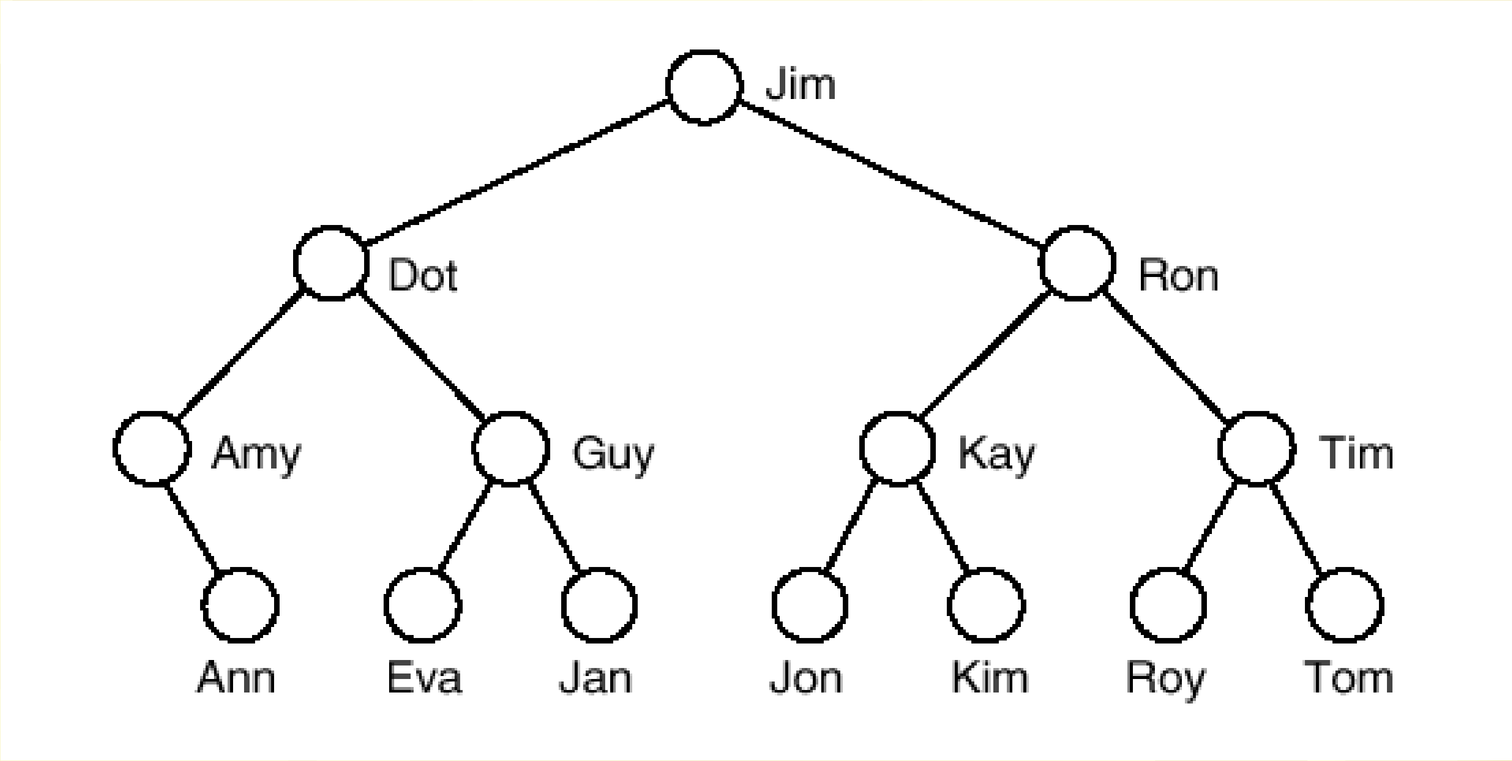


((a+(b*(c−d)))−(e/f))

$$x := (-b + (b \uparrow 2 - 4 \times a \times c) \uparrow 0.5) / (2 \times a)$$



Comparison tree



Inorder traversal

- template <class Entry>
void Binary_tree<Entry> :: recursive_inorder(
Binary_node<Entry> *sub_root, void (*visit)(Entry &))

```
/* Pre: sub_root is either NULL or points to a subtree of the Binary tree .
Post: The subtree has been traversed in inorder sequence.
Uses: The function recursive inorder recursively */

{
    if (sub_root != NULL) {
        recursive_inorder(sub_root->left, visit);
        (*visit)(sub_root->data);
        recursive_inorder(sub_root->right, visit);
    }
}
```

Preorder traversal

```
void preorder(BinNode* rt) // rt is root of a subtree
{
        if (rt == NULL) return; // Empty subtree
        visit(rt); // visit performs desired action
        preorder(rt->leftchild());
        preorder(rt->rightchild());
}
```

Traversal Example

```
// Return the number of nodes in the tree
template <class Elem>
int count(BinNode<Elem>* subroot) {
  if (subroot == NULL)
    return 0;  // Nothing to count
  return 1 + count(subroot->left())
           + count(subroot->right());
}
```

先序遍历二叉树的非递归算法

- 若T!=NIL，则：

  (1) 输出T->data；

  (2) 按先序次序输出左子树中各结点的值；

  (3) 按先序次序输出右子树中各结点的值。

- 若T==NIL，则表明以T为根指针的二叉树遍历完毕，应该返回（同时也表明对某一子树T1的左子树遍历完毕，下面应该对T1的右子树进行遍历了。此时，若栈不空，则应该根据存放在栈顶的指针找出T1的待遍历的右子树的根指针并赋给T，以继续遍历下去；若栈空则表明整个二叉树遍历完毕，结束。）

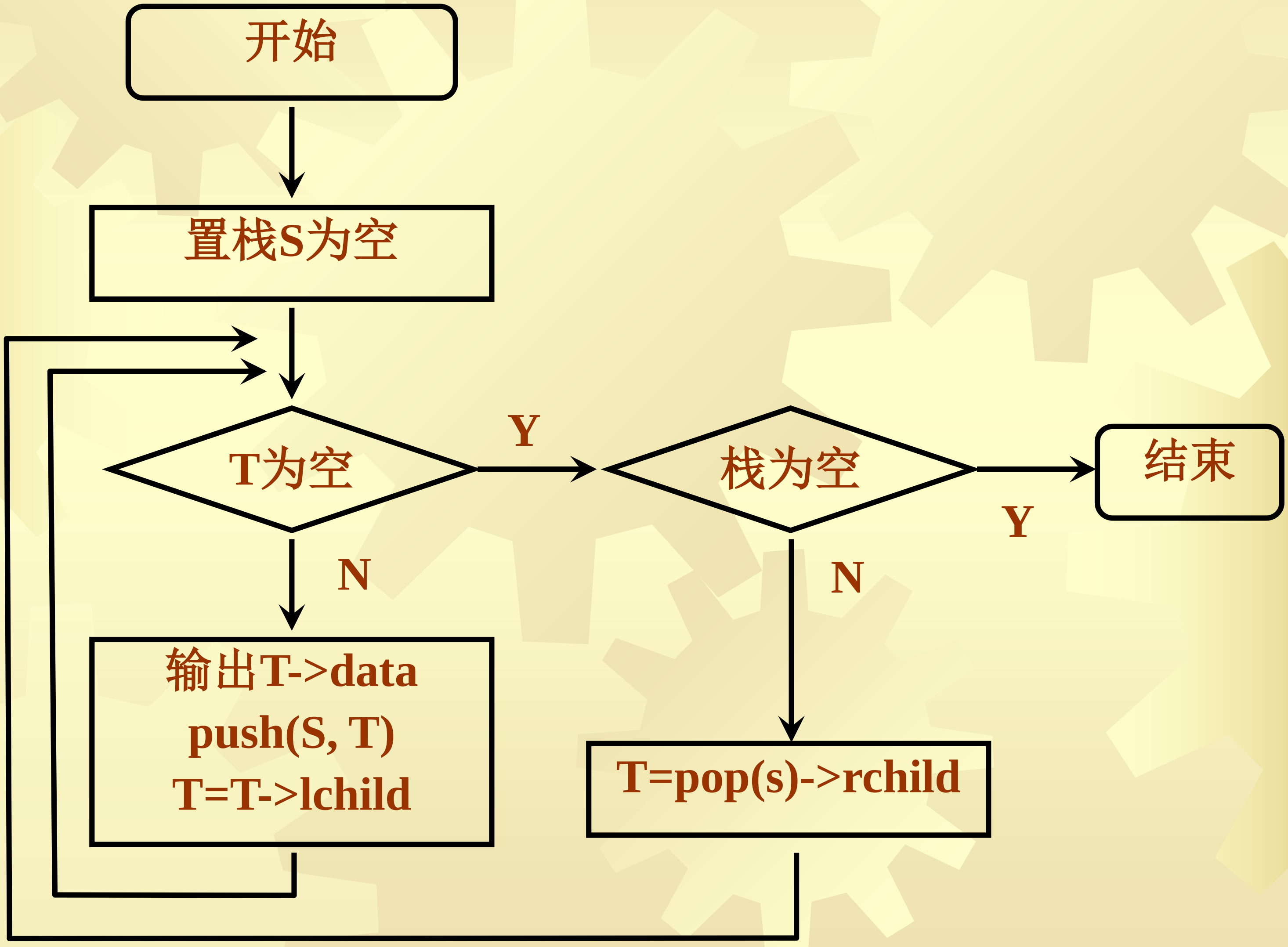
找右子树的方法

- 在先序遍历过T的整个左子树后，如何找到T的右子树的根指针呢？有以下两种方法：

(1) 在输出过T->data后，将指针T的值保存到栈中，接着遍历T的左子树。在遍历完T的左子树并返回时，退出栈顶元素到T，再对T的右子树进行先序遍历。

(2) 输出T->data后，保存到栈中的不是T而是结点T的右孩子指针，接着遍历左子树，遍历完左子树并返回时，退出栈顶元素到T，然后先序遍历以T为根的子树。

算法框图


开始
置栈S为空
T为空
Y
栈为空
结束
Y
N
N
输出T->data
push(S, T)
T=T->lchild
T=pop(s)->rchild

中序遍历二叉树的非递归算法

- 若T!=NIL，则：

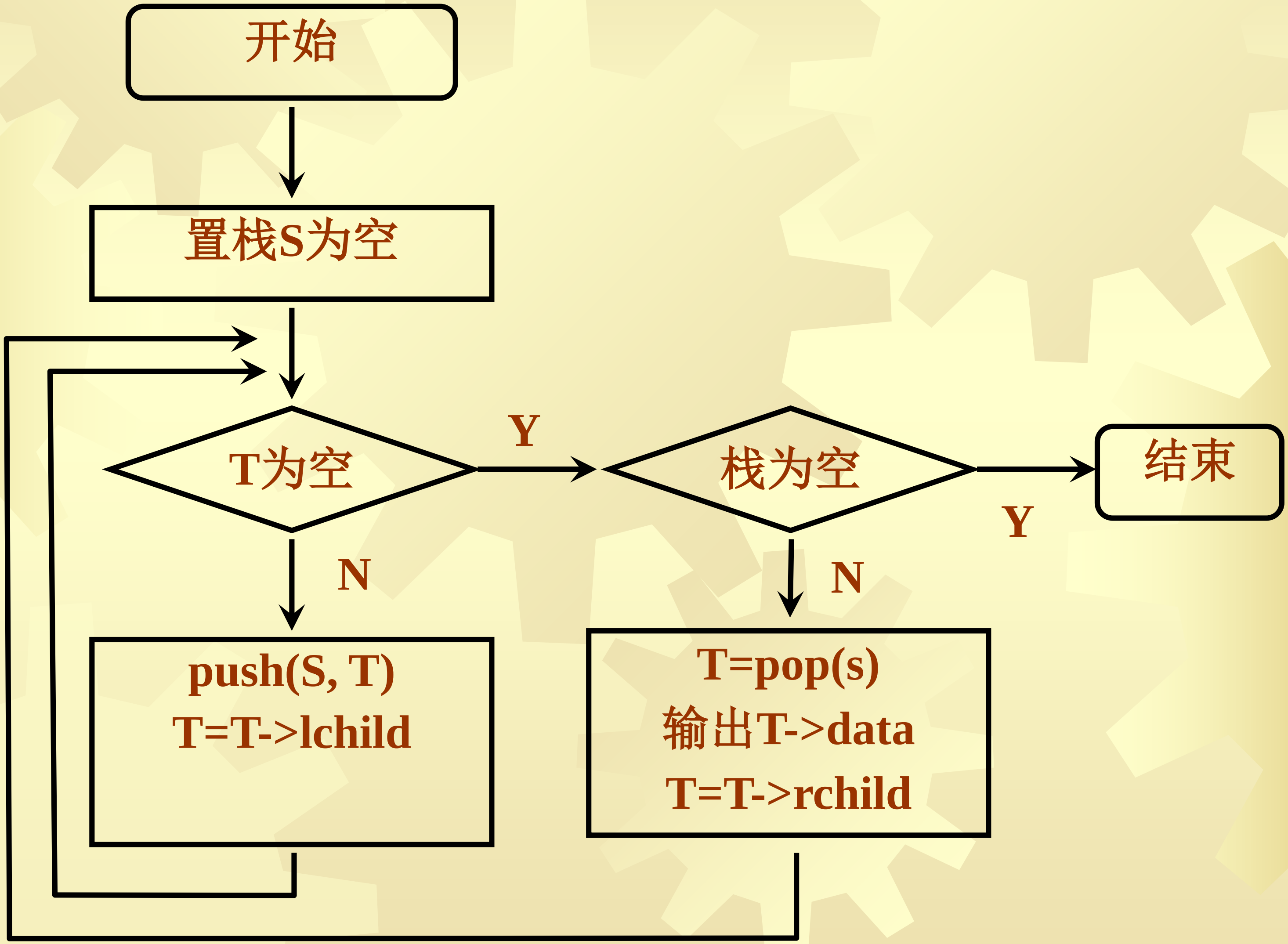
  (1) 按中序次序输出左子树中各结点的值；

  (2) 输出T->data；

  (3) 按中序次序输出右子树中各结点的值。

- 若T==NIL，则表明以T为根指针的二叉树遍历完毕，应该返回（同时也表明对某一子树T1的左子树遍历完毕，下面应该访问根结点，并对其右子树进行遍历了。此时，若栈不空，则栈顶元素一定为T1，取出栈顶元素并赋给T，在访问结点T以后继续遍历其右子树；若栈空则表明整个二叉树遍历完毕，结束。）

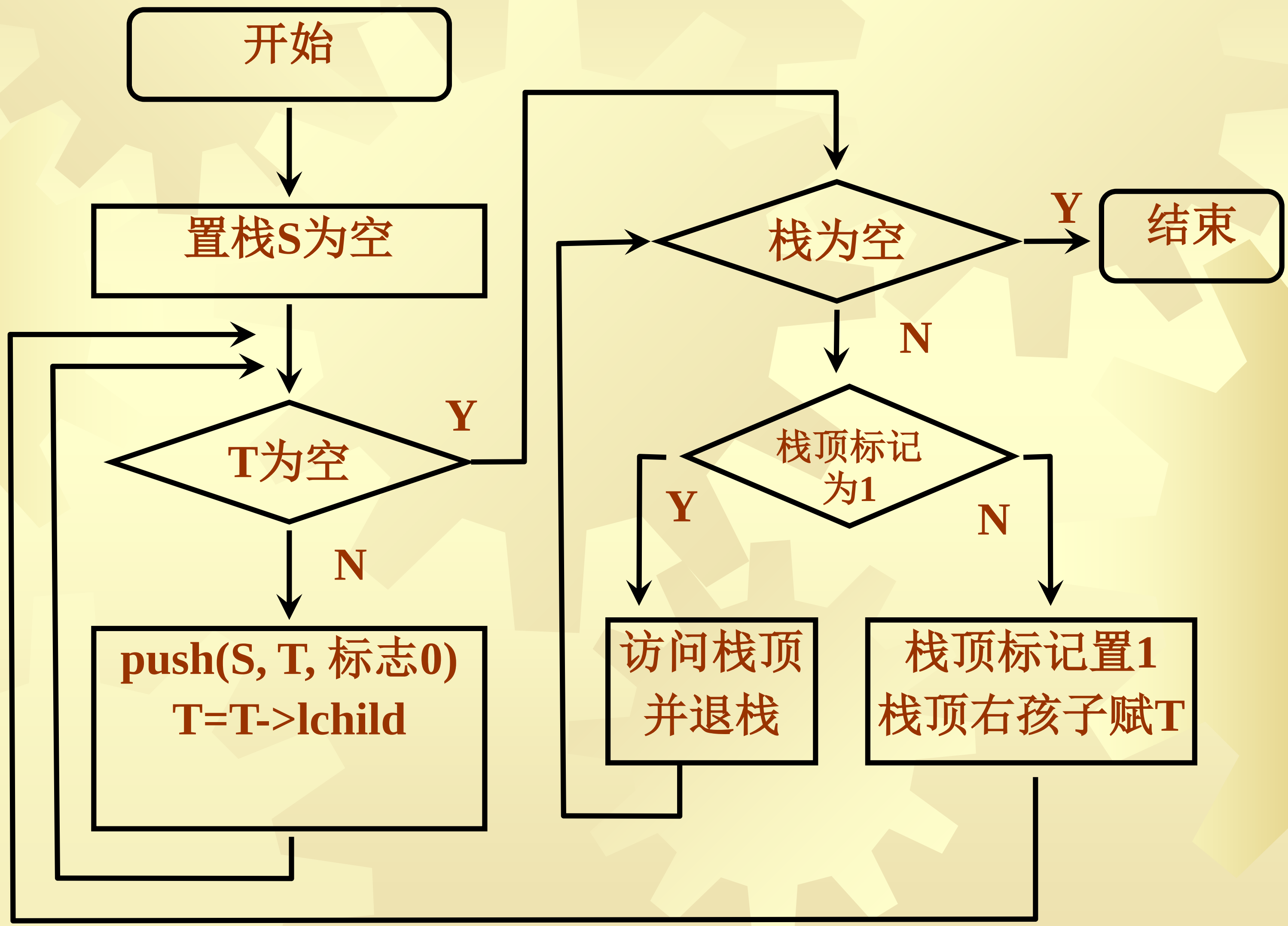
算法框图


开始
置栈S为空
T为空
Y
栈为空
Y
结束
N
N
push(S, T)
T=T->lchild
T=pop(s)
输出T->data
T=T->rchild

后序遍历二叉树的非递归算法

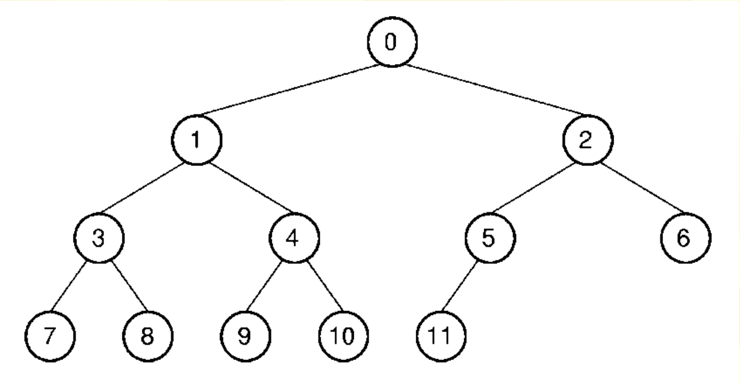
- (1) 若T!=NIL，则T及标志tag(0)入栈，遍历其左子树(左孩子赋T，转(1))；
- (2)如果T=NIL，则返回，此时：
  - **(a) 若栈空，则整个遍历过程结束；**
  - **(b) 若栈不空，表明栈顶结点的左子树或右子树已遍历完毕，此时，若栈顶结点的标志tag为0，则修改为1，并遍历其右子树(右孩子赋T，转(1))；否则，退出并输出栈中右子树已遍历过后所的栈顶结点，再转(1)。**

算法框图



开始
置栈S为空
T为空
Y
N
push(S, T, 标志0)
T=T->lchild
栈为空
Y
结束
N
栈顶标记
为1
Y
N
访问栈顶
并退栈
栈顶标记置1
栈顶右孩子赋T

Array Implementation

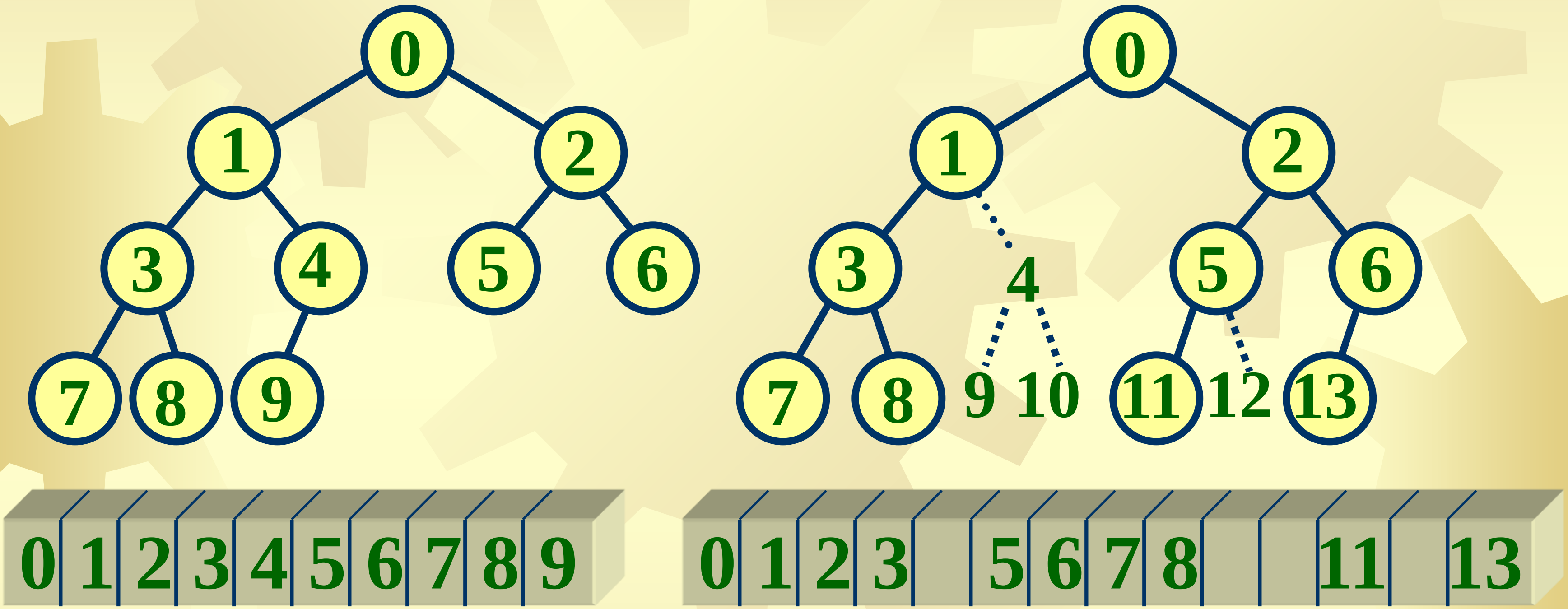


Position	0	1	2	3	4	5	6	7	8	9	10	11
Parent	--	0	0	1	1	2	2	3	3	4	4	5
Left Child	1	3	5	7	9	11	--	--	--	--	--	--
Right Child	2	4	6	8	10	--	--	--	--	--	--	--
Left Sibling	--	--	1	--	3	--	5	--	7	--	9	--
Right Sibling	--	2	--	4	--	6	--	8	--	10	--	--

Array Implementation

- Parent (r) = (r - 1) / 2 $\quad$ $0<r<n$
- Leftchild(r) = 2r + 1 $\quad$ $2r+1<n$
- Rightchild(r) =2r + 2 $\quad$ $2r+2<n$
- Leftsibling(r) = r - 1 $\quad$ $0<r<n$ and $r=2t$
- Rightsibling(r) = r + 1 $\quad$ $r+1<n$ and $r=2t+1$

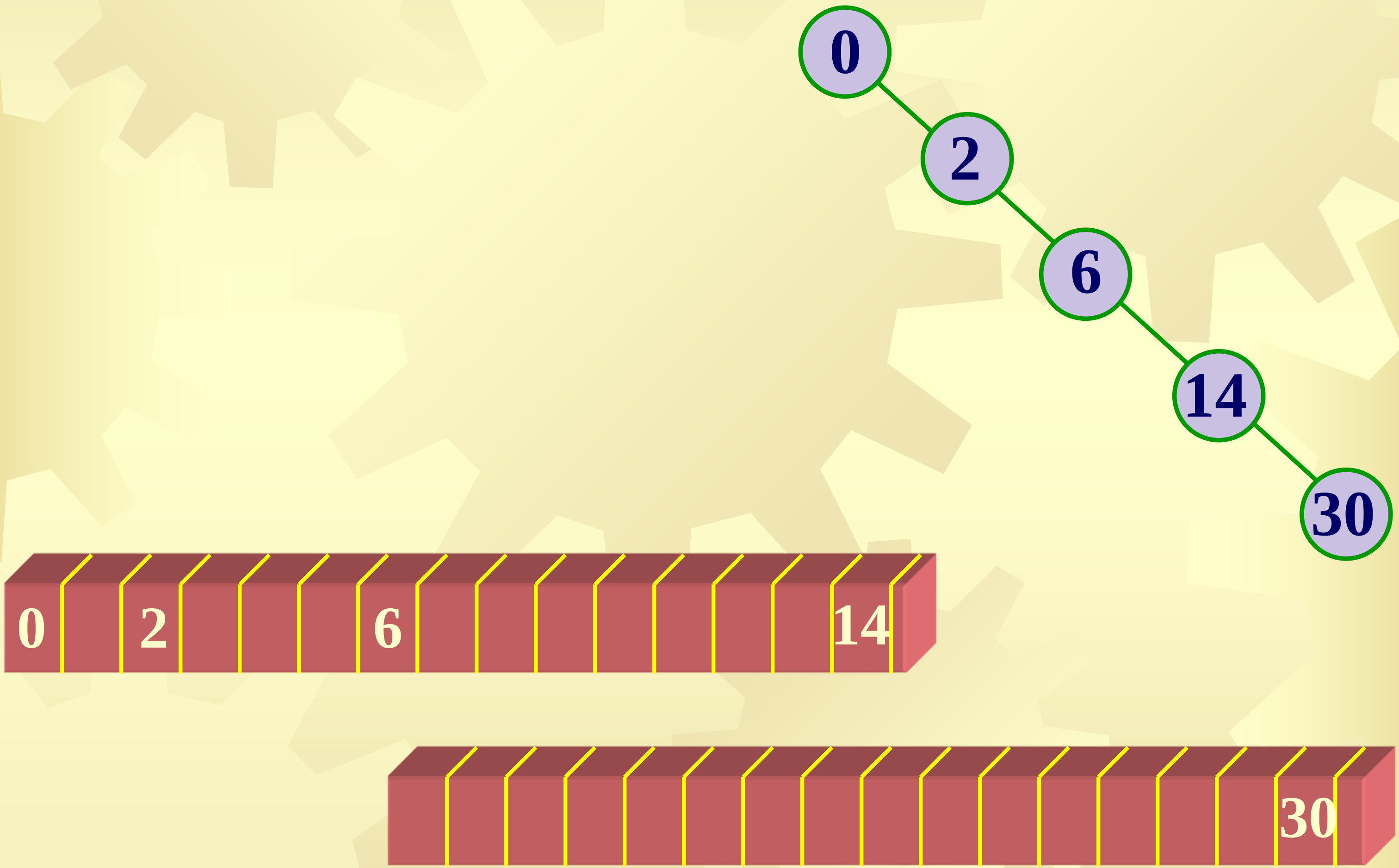
Array Implementation



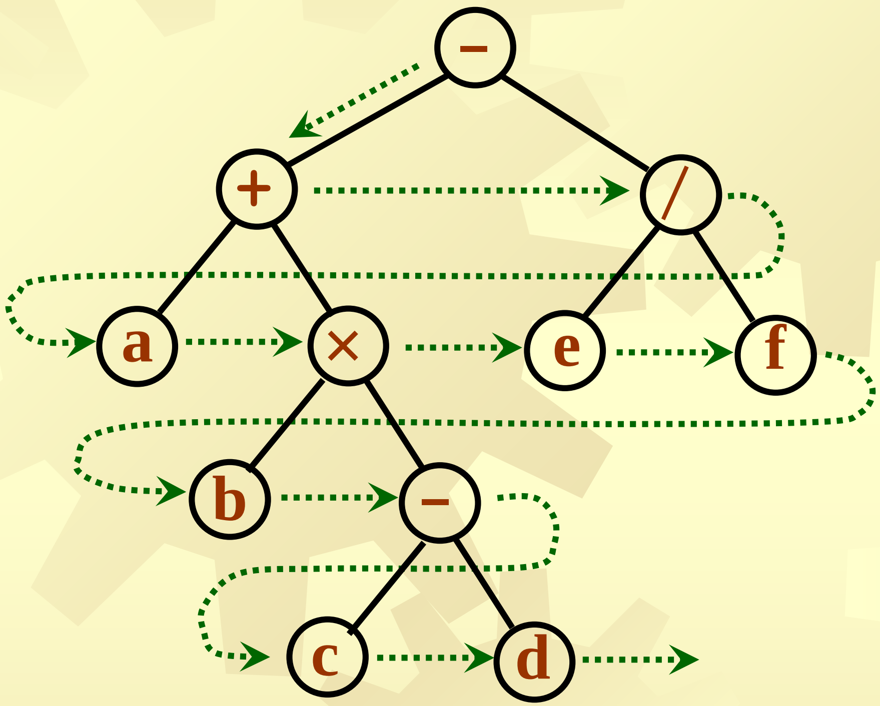
Complete tree

Tree

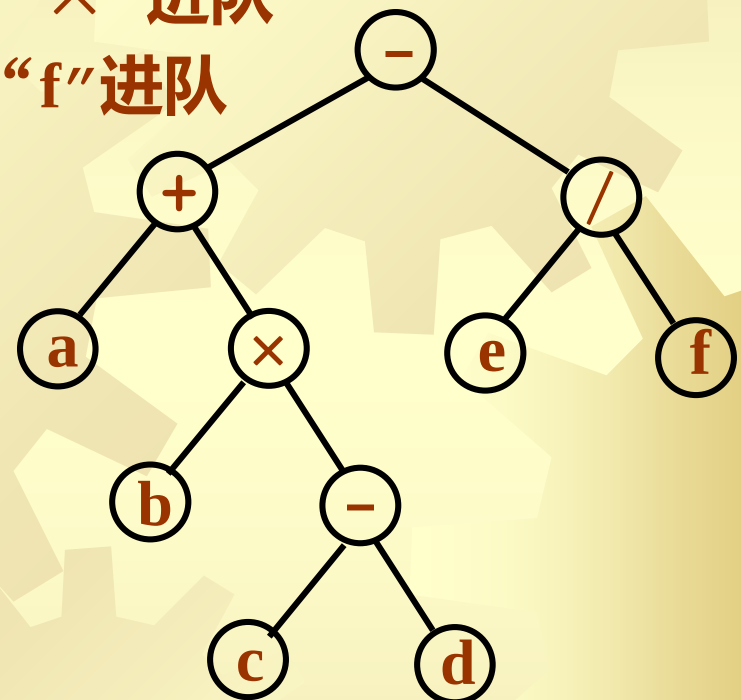
极端情形: 只有右单支的二叉树



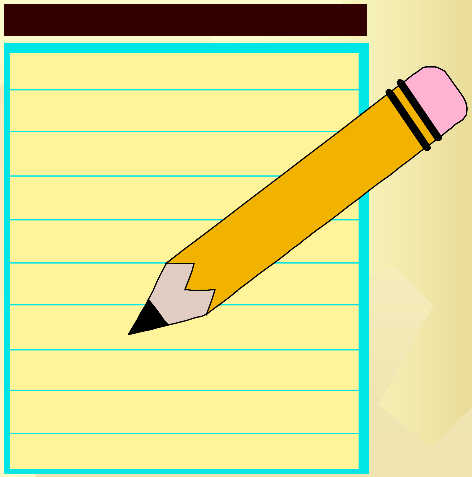
Level order



Q	-				根“-”进队
Q	+	/			“-”出队，“+” “/”进队
Q	/	a	×		“+”出队，“a” “×”进队
Q	a	×	e	f	“/”出队，“e” “f”进队
Q	×	e	f		“a”出队，无进队
Q	e	f	b	-	“×”出队，“b” “-”进队
Q	f	b	-		“e”出队，无进队
Q	b	-			“f”出队，无进队
Q	-				“b”出队，无进队
Q	c	d			“-”出队，“c” “d”进队
Q	d				“c”出队，无进队
Q					“d”出队，无进队



Questions?



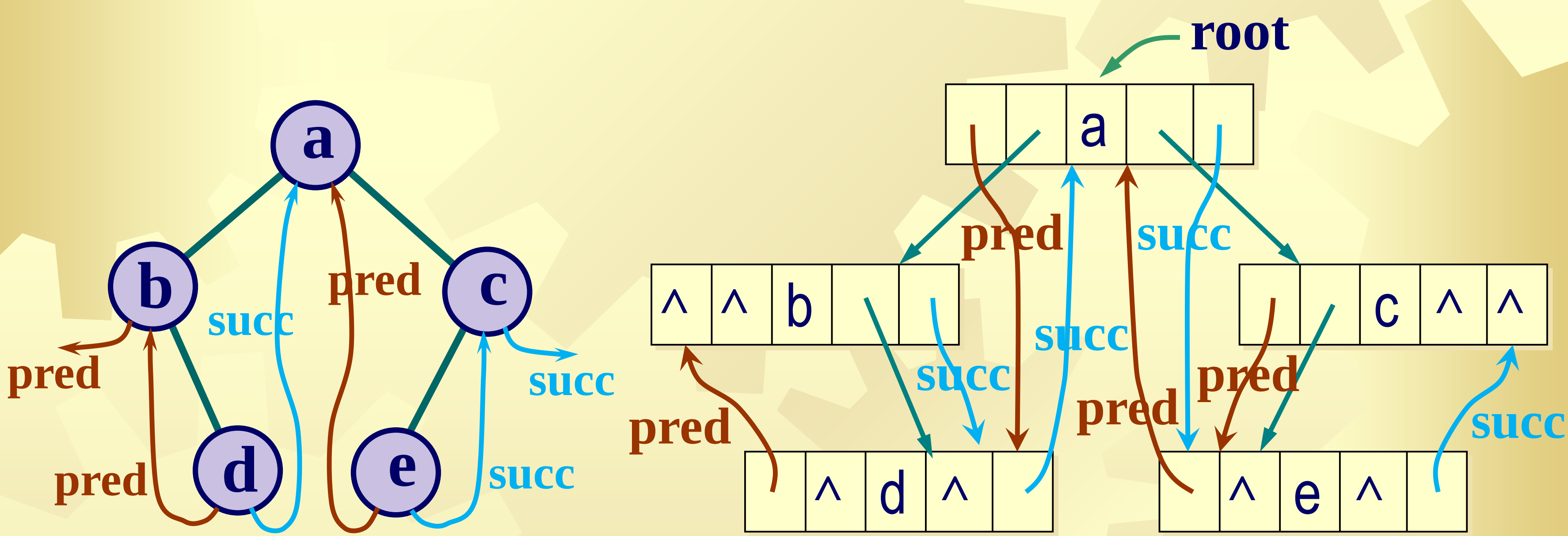
线索二叉树 (Threaded Binary Tree)

- 通过二叉树遍历，可将二叉树中所有结点的数据排列在一个线性序列中，可以找到某数据在这种排列下它的前驱和后继。
- 希望不必每次都通过遍历找出这样的线性序列。只要事先做预处理，将某种遍历顺序下的前驱、后继关系记在树的存储结构中，以后就可以高效地找出某结点的前驱、后继。
- 为此，在二叉树存储结点中增加线索信息。

线索 (Thread)

pred	**left**	**data**	**right**	**succ**

- 增加前驱pred指针和后继succ指针的二叉树



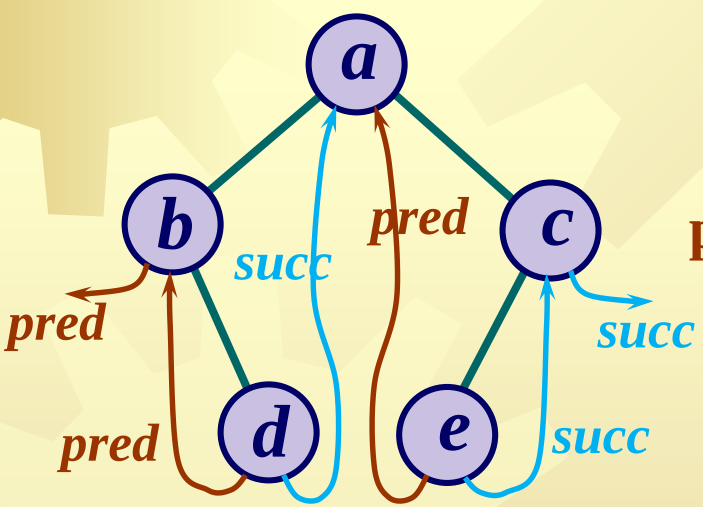
- 改造树结点，将 pred 指针和 succ 指针压缩到 left 和 right 的空闲指针中，并增设两个标志 ltag 和 rtag，指明指针是指示子女还是前驱 / 后继。后者称为线索。

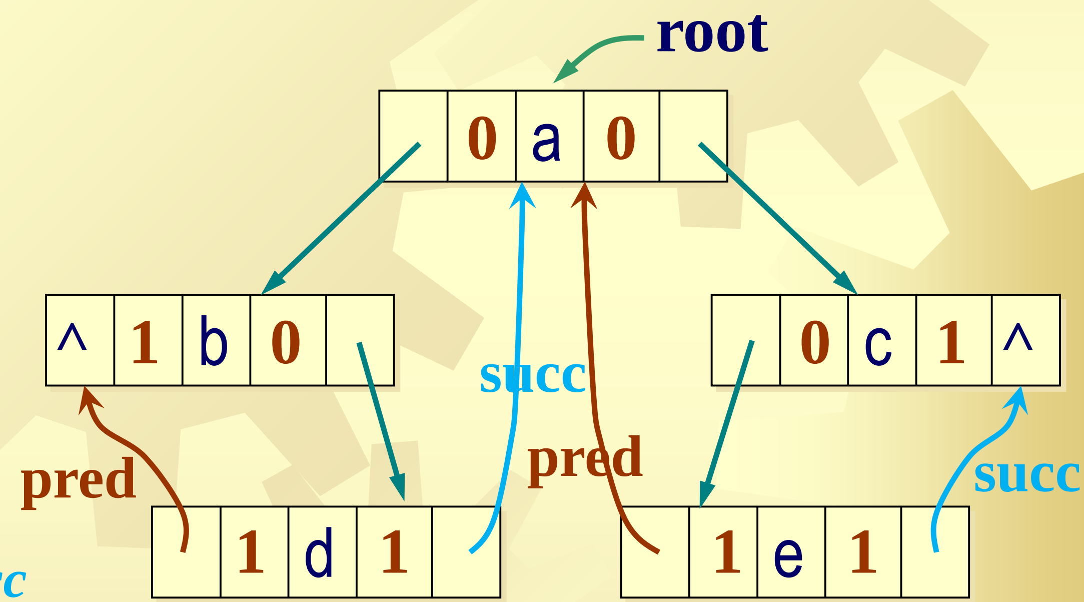
left	**ltag**	**data**	**rtag**	**right**

- ltag (或rtag) = 0，表示相应指针指示左子女（或右子女结点）；
- ltag (或rtag) = 1，表示相应指针为前驱（或后继）线索。

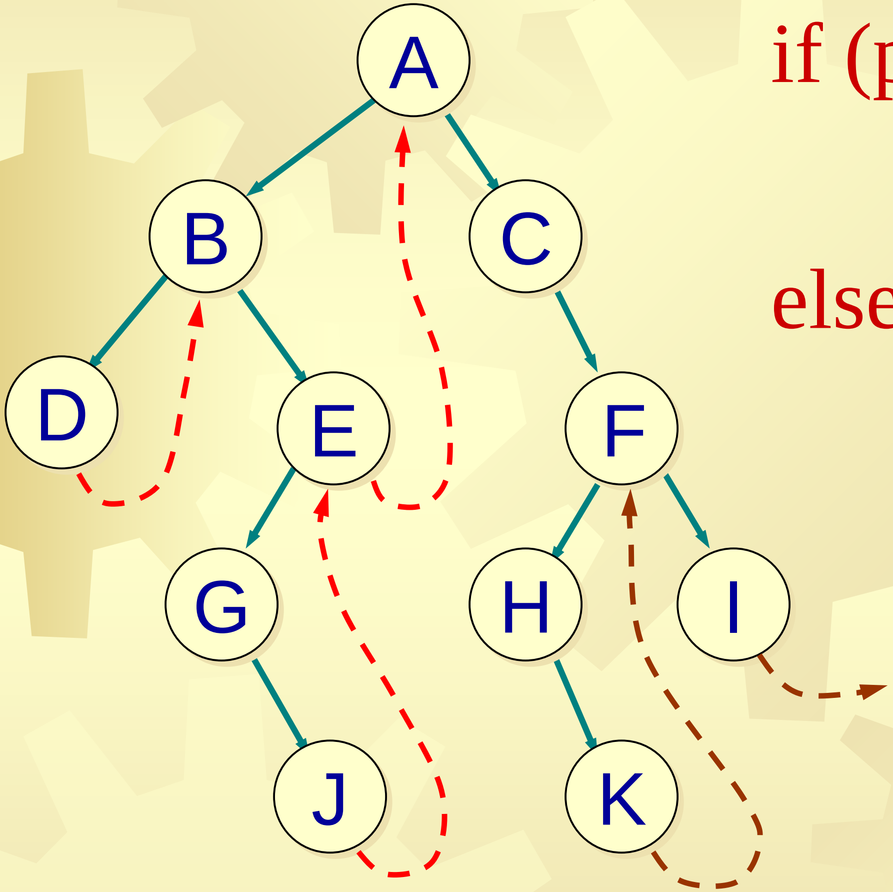
中序线索二叉树及其链表表示

left	ltag	data	rtag	right





寻找结点 p 在中序下的后继



if (p->rtag ==1)

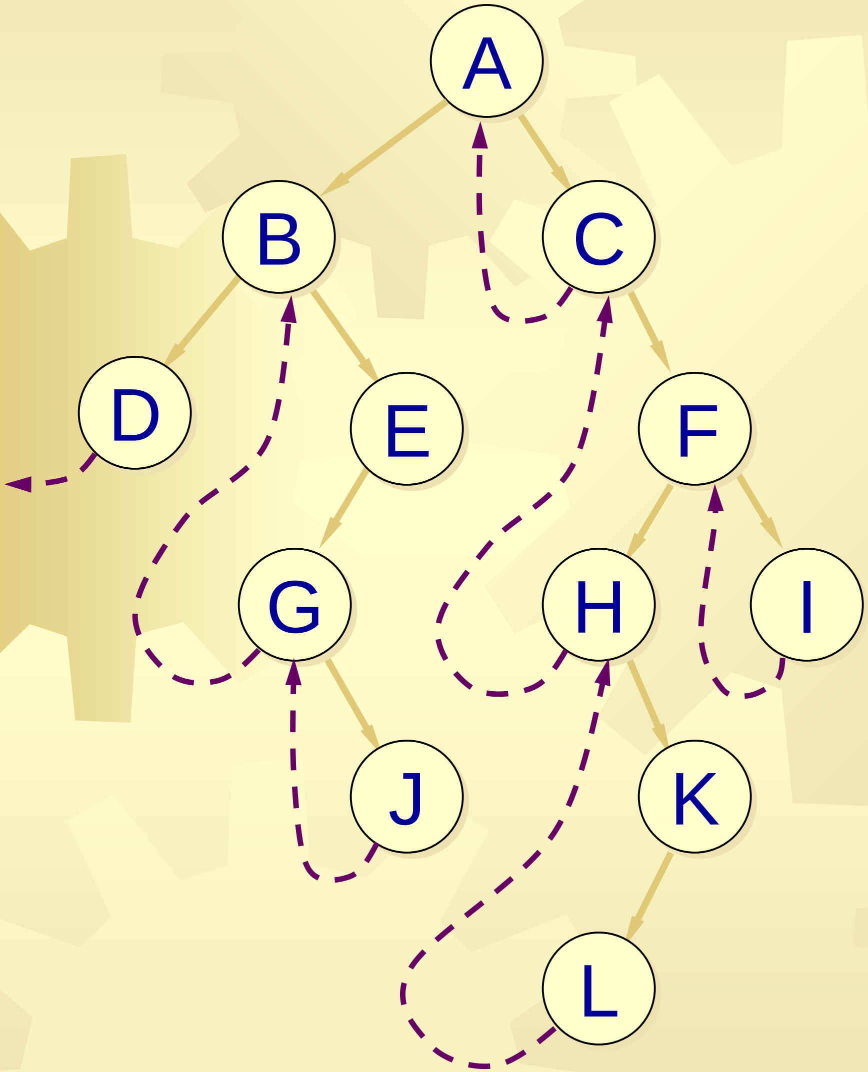
后继为 p->rchild

else //p->rtag != 1

后继为结点 p 右子树

q 中的中序下的第一个结点

寻找结点 p 在中序下的前驱



if (p->ltag==1)

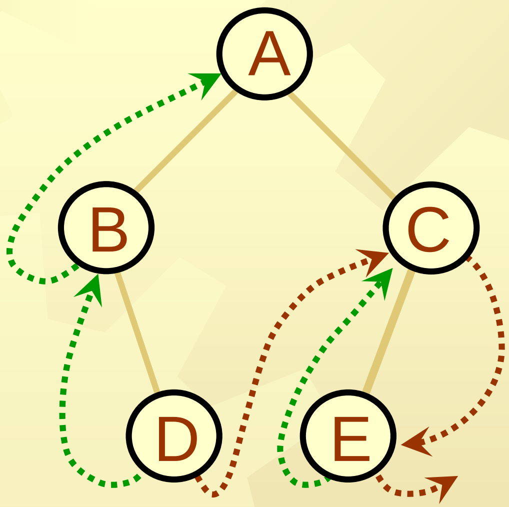
前驱为 p->lchild

else //p->leftThread==0

前驱为结点 p 左子树中序下的最后一个结点

通过前序遍历建立前序线索二叉树

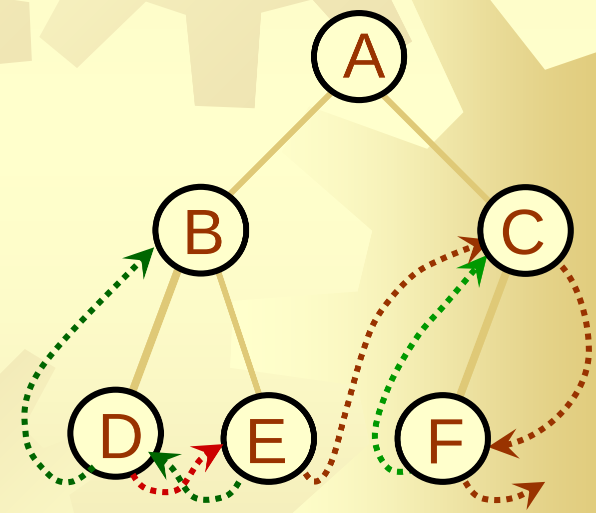
- 在前序线索二叉树上寻找指定结点前序下的**后继**比较容易
  - 如果结点有左子女，左子女是前序下的后继；
  - 如果没有左子女，则右子女是前序下的后继。



前序序列 *ABDCE*

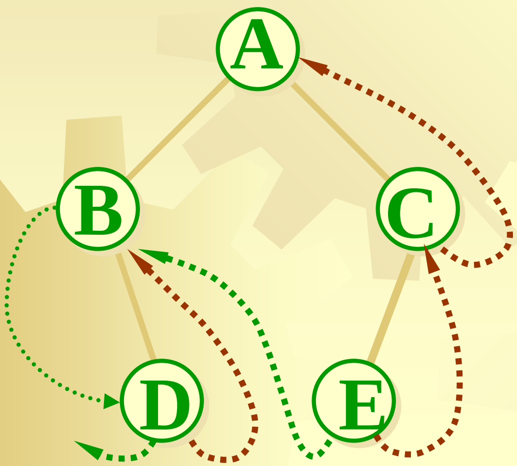
通过前序遍历建立前序线索二叉树

- 在前序线索二叉树中寻找指定结点*p的前序下的**前驱**
  - 如果结点*p有前驱线索，则可直接找到前序下的前驱结点，否则
  - 寻找结点*p的双亲*q：
  - 如果*q不存在，则*p无前驱；
  - 如果*p是*q的左子女，则*q是*p的前驱结点；
  - 如果*p是*q的右子女，则前驱是*q左子树中前序下的最后一个结点。



前序序列 *ABDECF*

后序线索二叉树

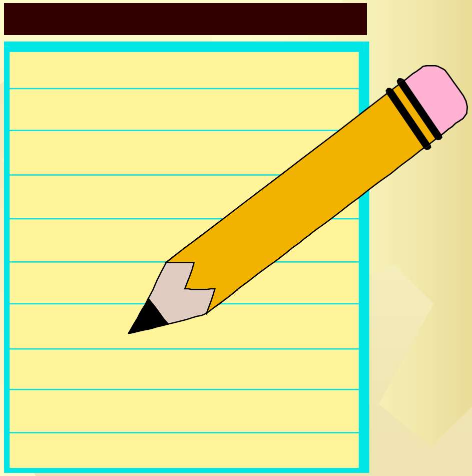


后序序列

D B E C A

- 后序线索二叉树与前序线索二叉树是对称的，只要把左、右互换即可。故不再详细讨论。
- 后序线索二叉树的建立通过后序遍历二叉树得到，只需把建立线索的操作移到两个递归语句后面，其他与中序、前序线索二叉树建立算法的代码基本一致。

Questions?



Huffman Coding Trees

- ASCII codes: 8 bits per character.
- Fixed-length coding.
- Can take advantage of relative frequency of letters to save space.
- Variable-length coding

Z	K	F	C	U	D	L	E
2	7	24	32	37	42	42	120

- Weighted path length=weight * depth   for leaves
- Build the tree with *minimum external path weight.*

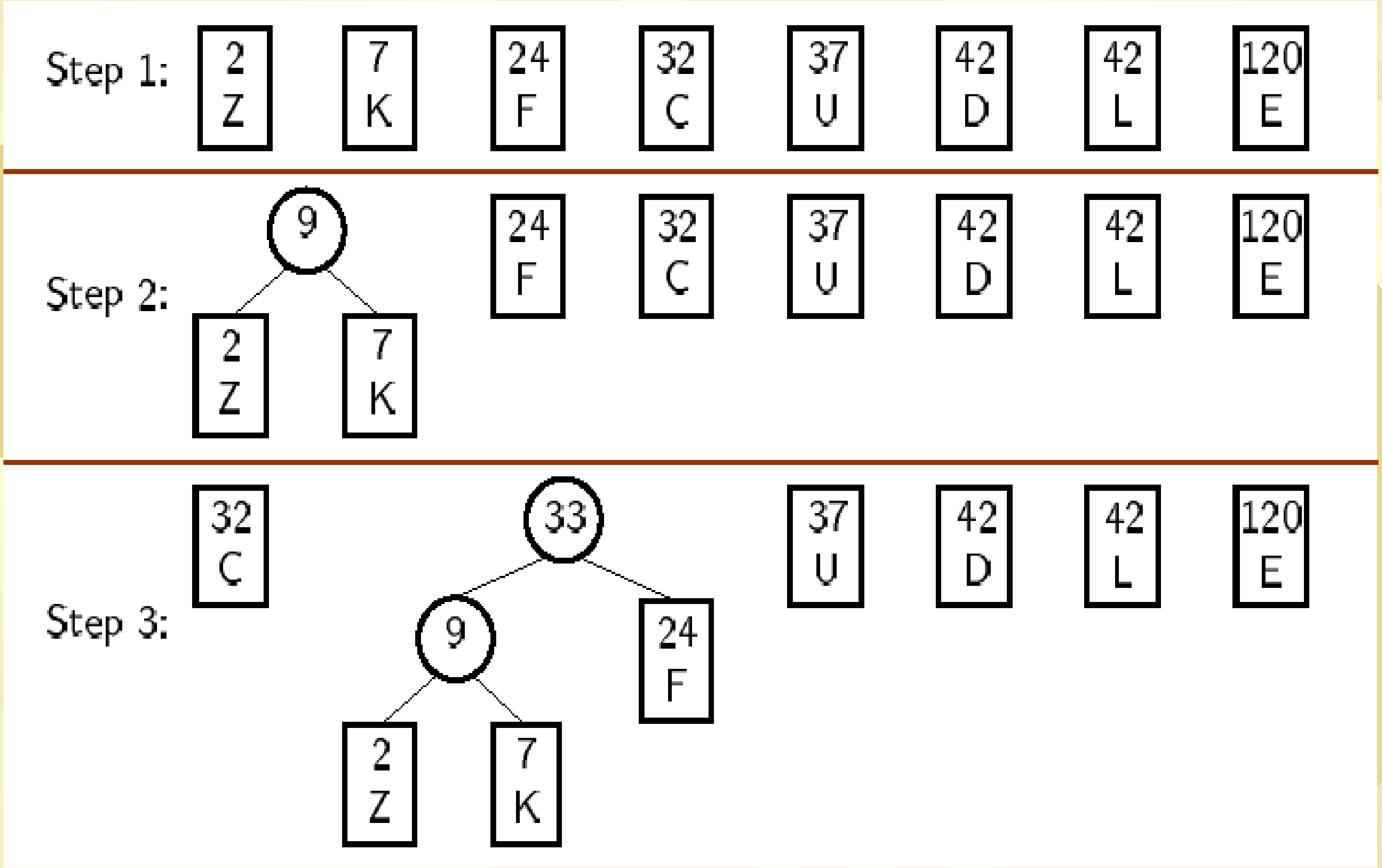
  min(∑(weighted path length))

  leaves

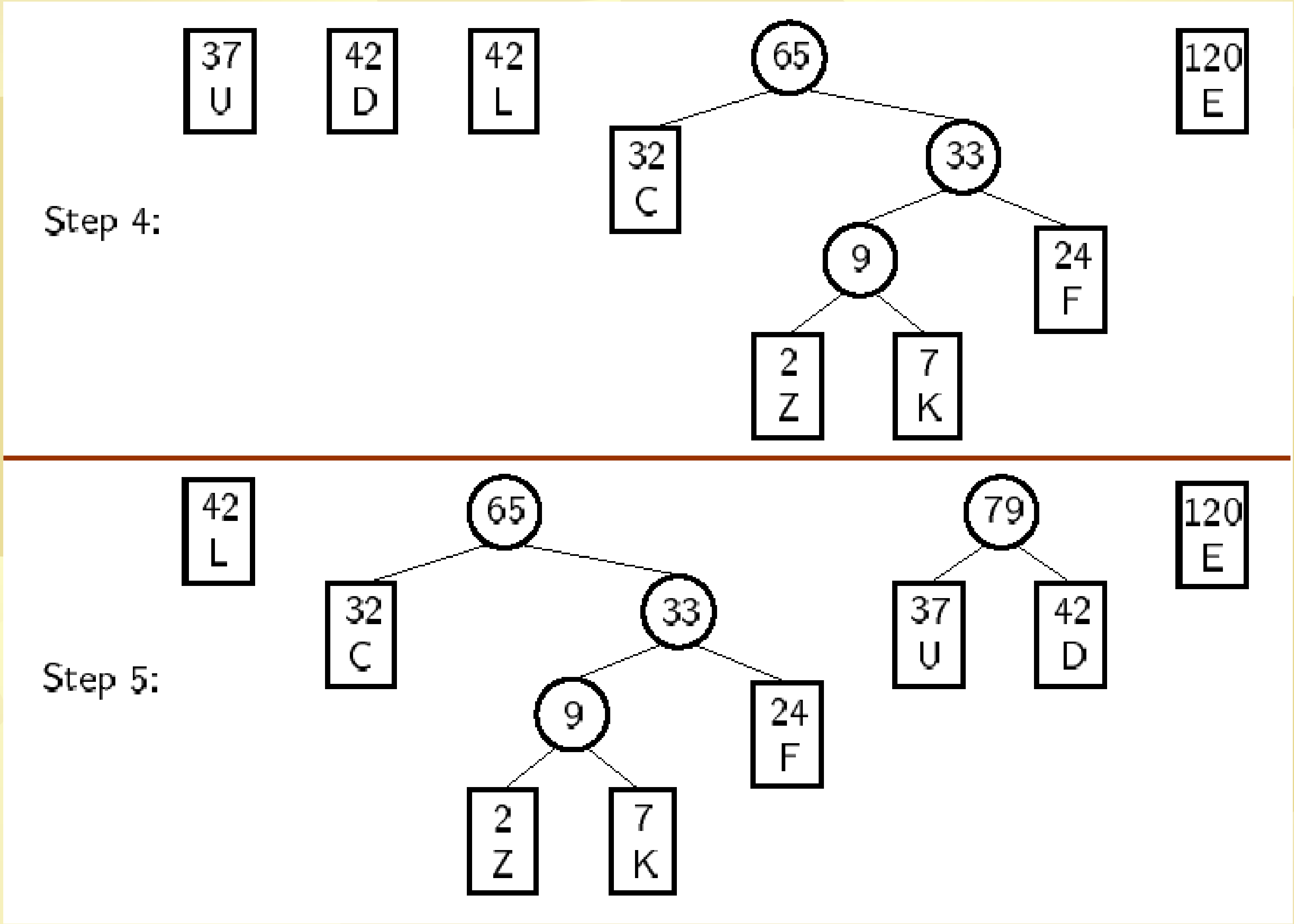
- Definition :
  Weighted path length(带权路径长度)
  =weight * depth (for leaves)
  （权） （深度）

- 建立一棵有***minimum external path weight.***
  （最小外部路径权重）的树
  min($\sum^{leaves}$(weighted path length))

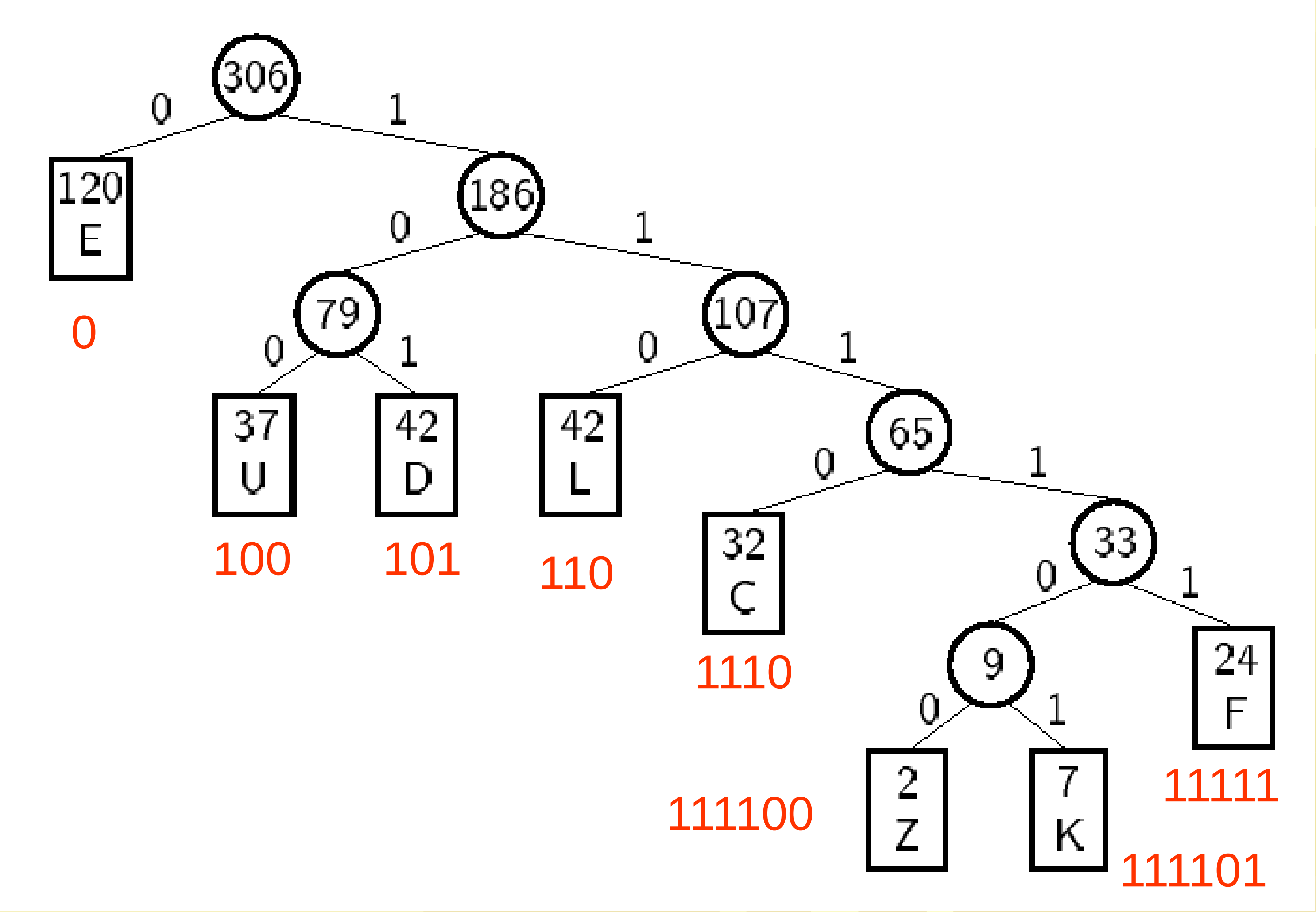
Huffman Tree Construction



Huffman Tree Construction



Huffman Tree



Assigning Codes

Letter	Freq	Code	Bits
C	32	1110	4
D	42	101	3
E	120	0	1
F	24	11111	5
K	7	111101	6
L	42	110	3
U	37	100	3
Z	2	111100	6

Coding and Decoding

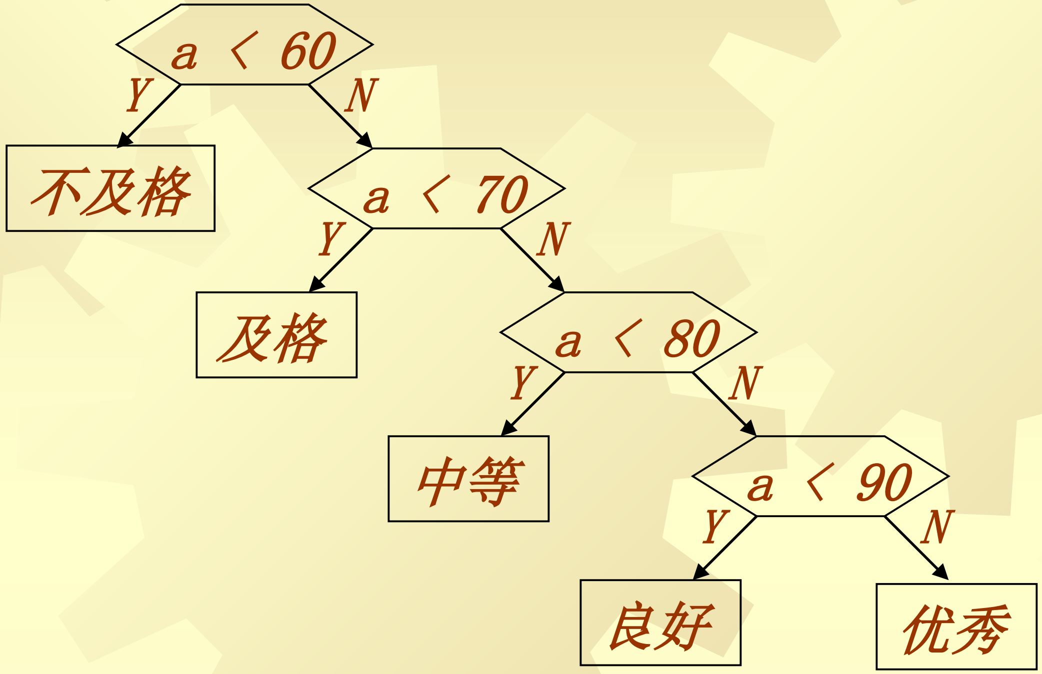
- A set of codes is said to meet the prefix property if no code in the set is the prefix of another.

- Code for DEED: 10100101
- Decode 1011001110111101: DULK

- Expected cost per letter:
  $\sum(c_i * f_i) / f_T = 785 / 306 = 2.56536$

由百分制转换为五分制

```
if (a<60)
  b=‘bad’;
else if (a<70)
        b=‘pass’;
      else if (a<80)
            b=‘genneral’;
          else if (a<90)
                b=‘good’;
              else b=‘excellent’;
```

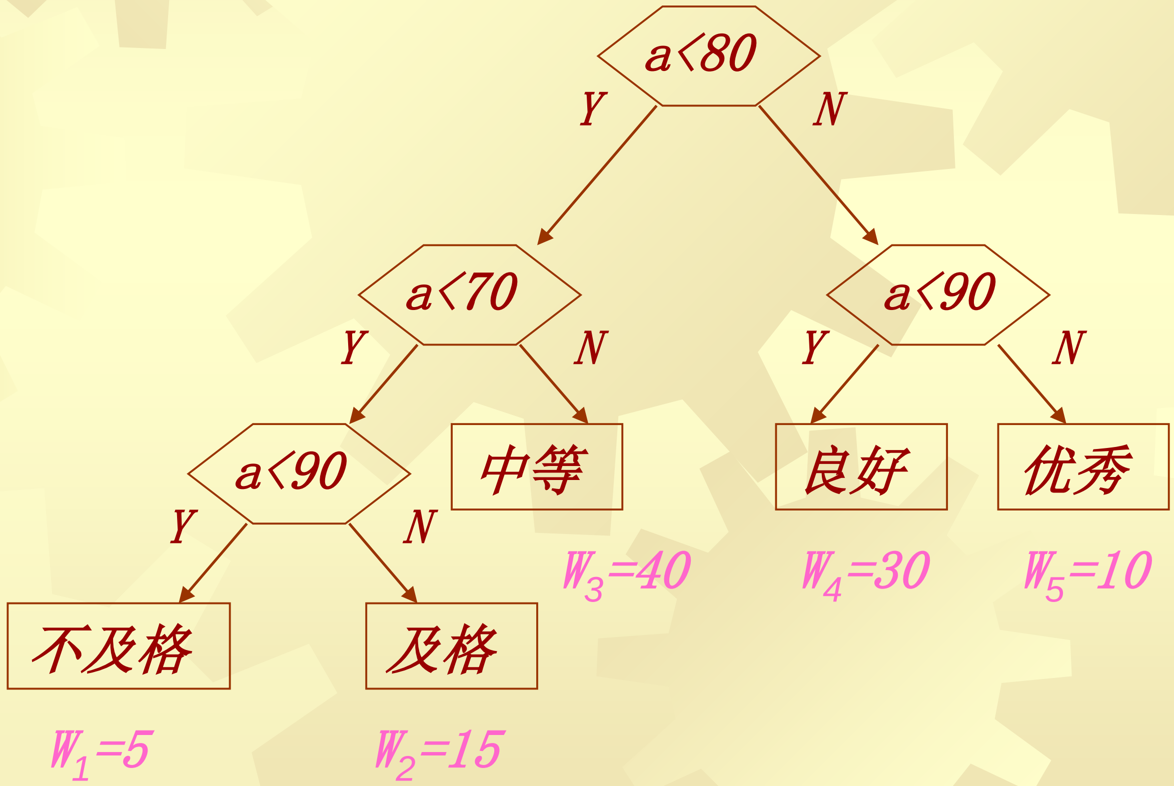


假设一批学生成绩的分布规律如下

分数	0-59	60-69	70-79	80-89	90-100
比例(%)	5	15	40	30	10

- 该算法将有80%的数据需要进行3次或3次以上的比较，才能得出结果。
- 若有10,000个数据进行换算，则总共需要31,500次判断。
- 改进：
  - **将出现次数多的数据（如分数在70~79，占40%），尽早进行判定，出现次数少的数据（如60分以下或90以上）较晚进行判定，算法的比较次数有望降低。**

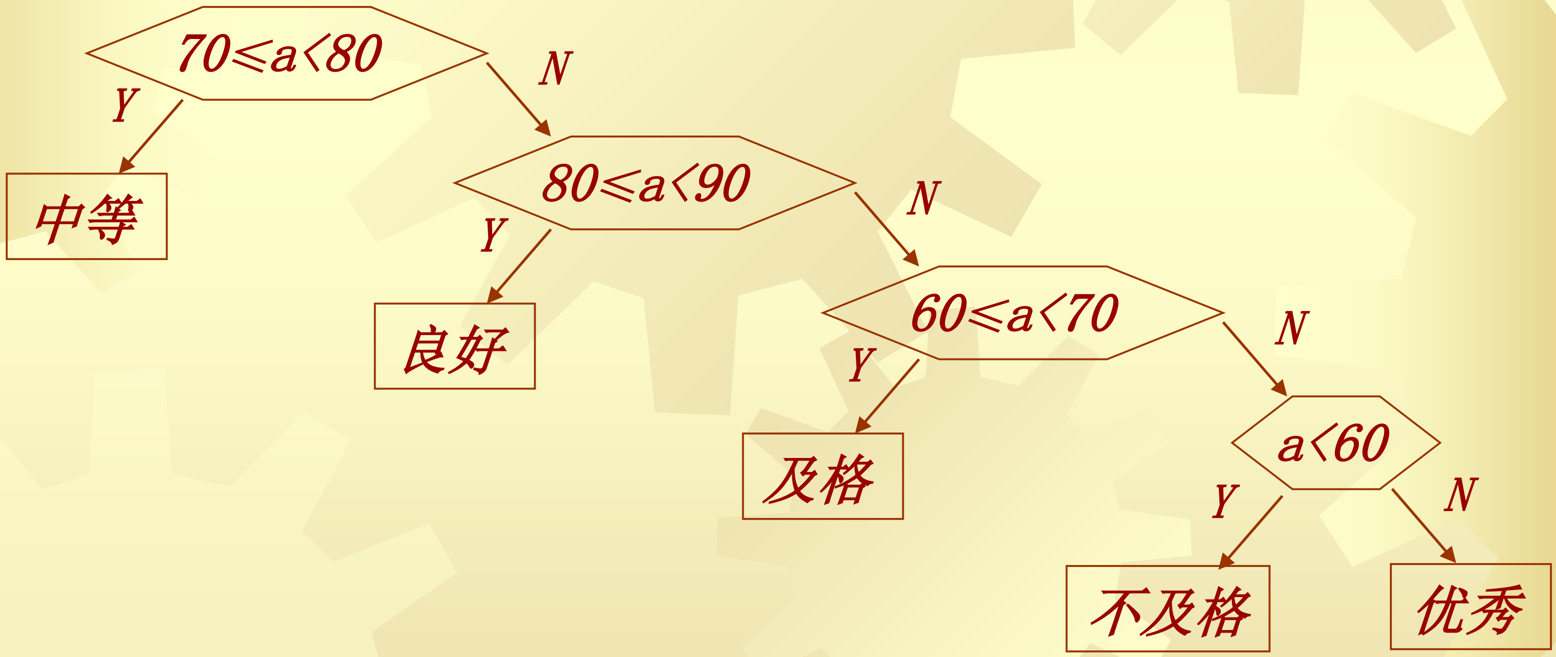
带权路径长度(WPL)



a<80
Y
N
a<70
a<90
Y
N
Y
N
a<90
中等
良好
优秀
Y
N
W3=40
W4=30
W5=10
不及格
及格
W1=5
W2=15

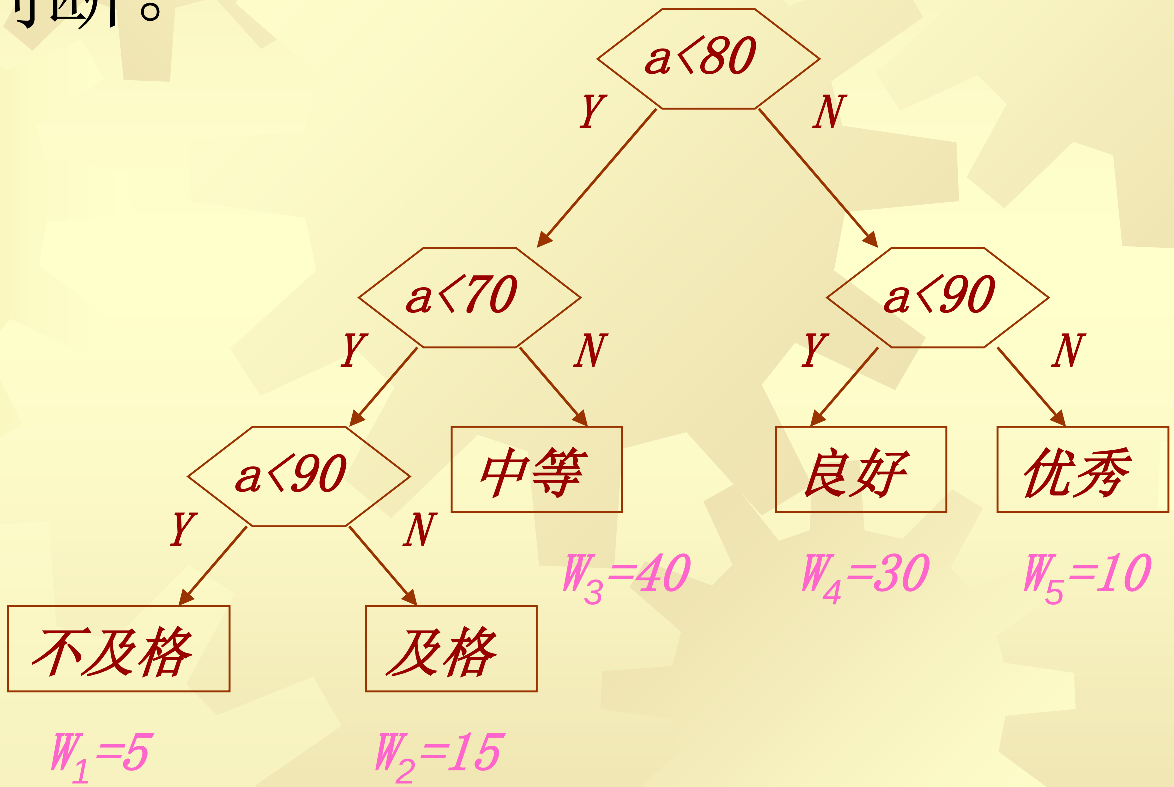
改进算法

- 对于10,000个数据该算法共需20,500次判断。



改进算法

- 10,000个数据进行换算，则共需要20,500次判断。



Questions?