The 5th Course

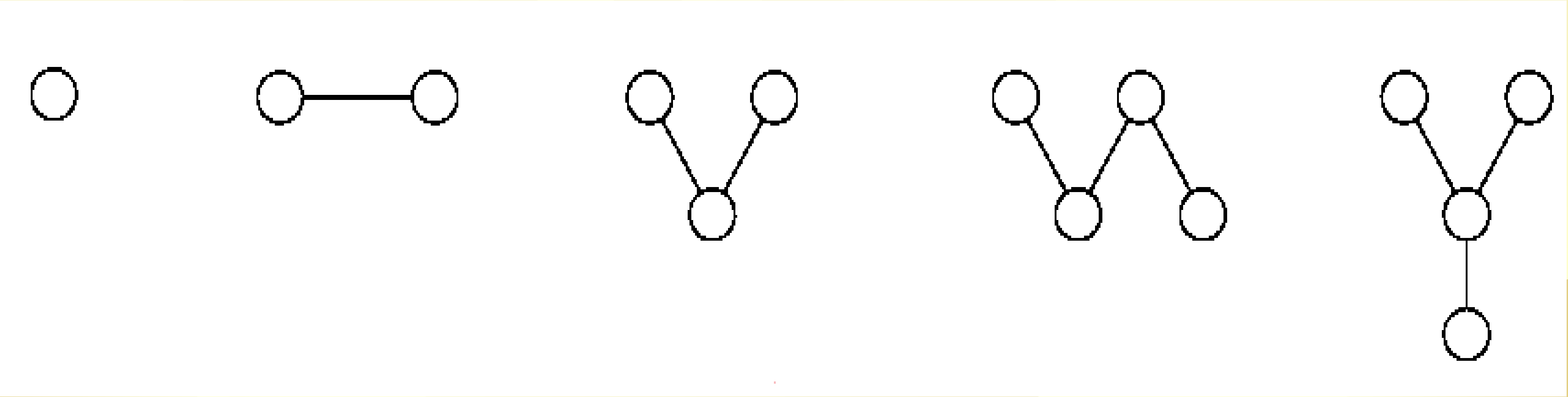
Multiway Trees

Definitions:

- ***A (free) tree*** **is any set of points (called *vertices*) and any set of pairs of distinct vertices (called *edges* or *branches*):**
  - **there is a sequence of edges (a *path*) from any vertex to any other &**
  - **there are no *circuits*, that is, no paths starting from a vertex and returning to the same vertex.**

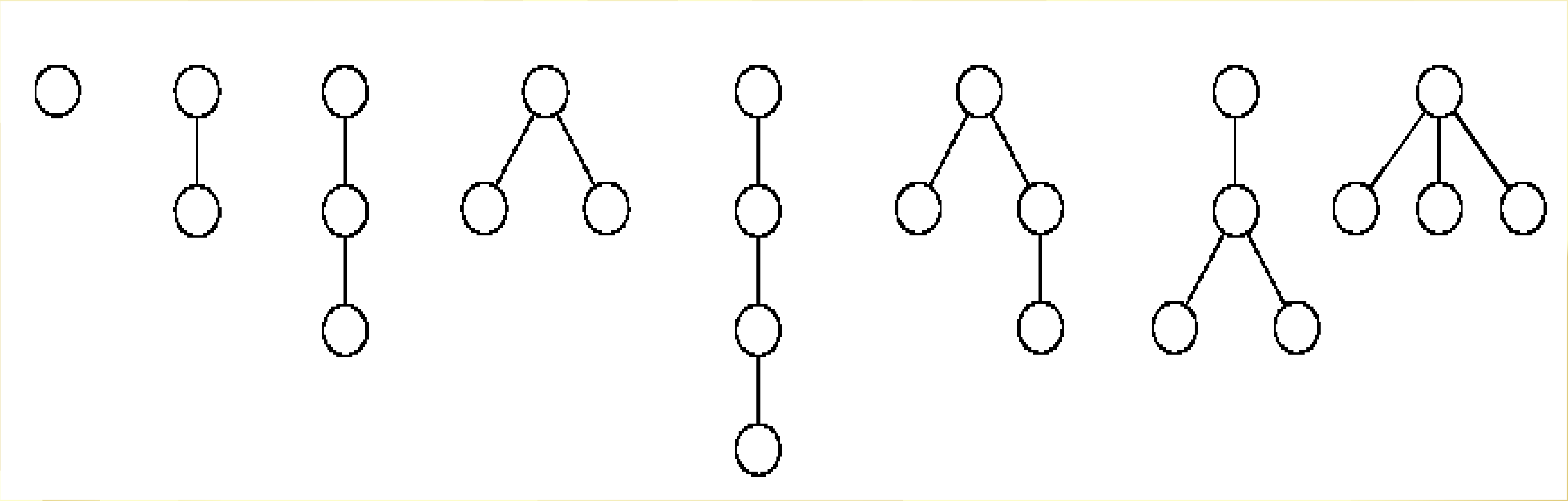
- *A rooted tree* is a tree in which one vertex, called the *root*, is distinguished.
- *An ordered tree* is a rooted tree in which the children of each vertex are assigned an order.
- *A forest* is a set of trees. We usually assume that all trees in a forest are rooted.
- *An orchard* (also called an ordered forest) is an ordered set of ordered trees.

Free trees



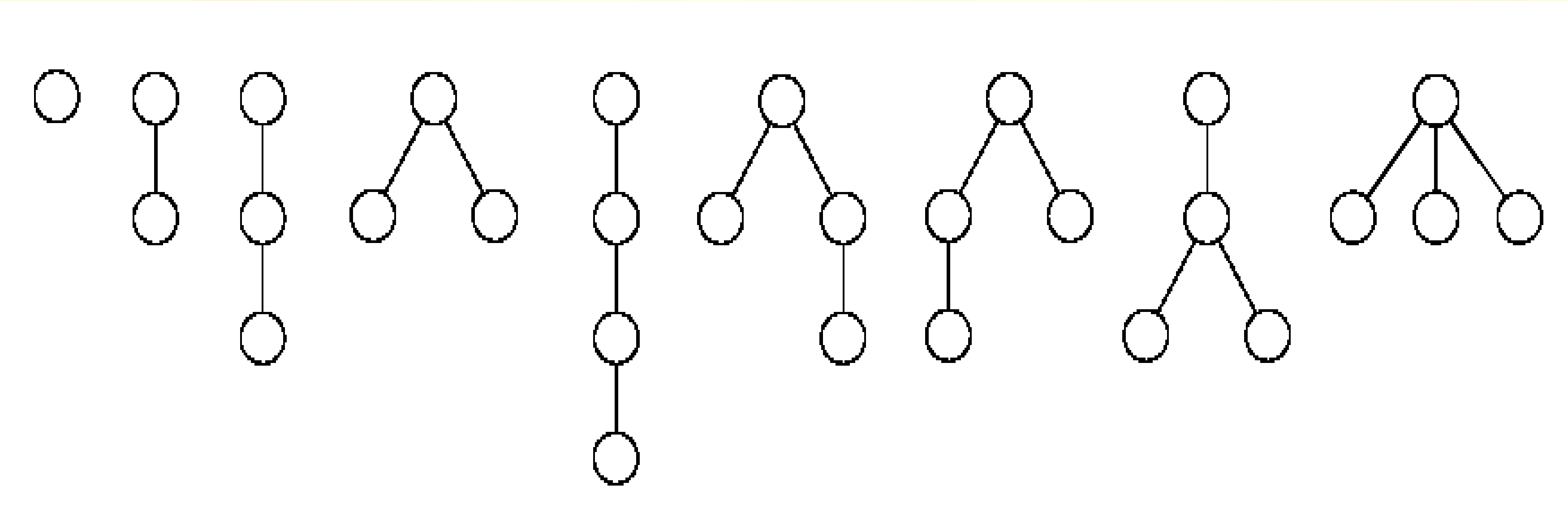
Free trees with four or fewer vertices
(Arrangement of vertices is irrelevant.)

Rooted tree



Rooted trees with four or fewer vertices
(Root is at the top of tree.)

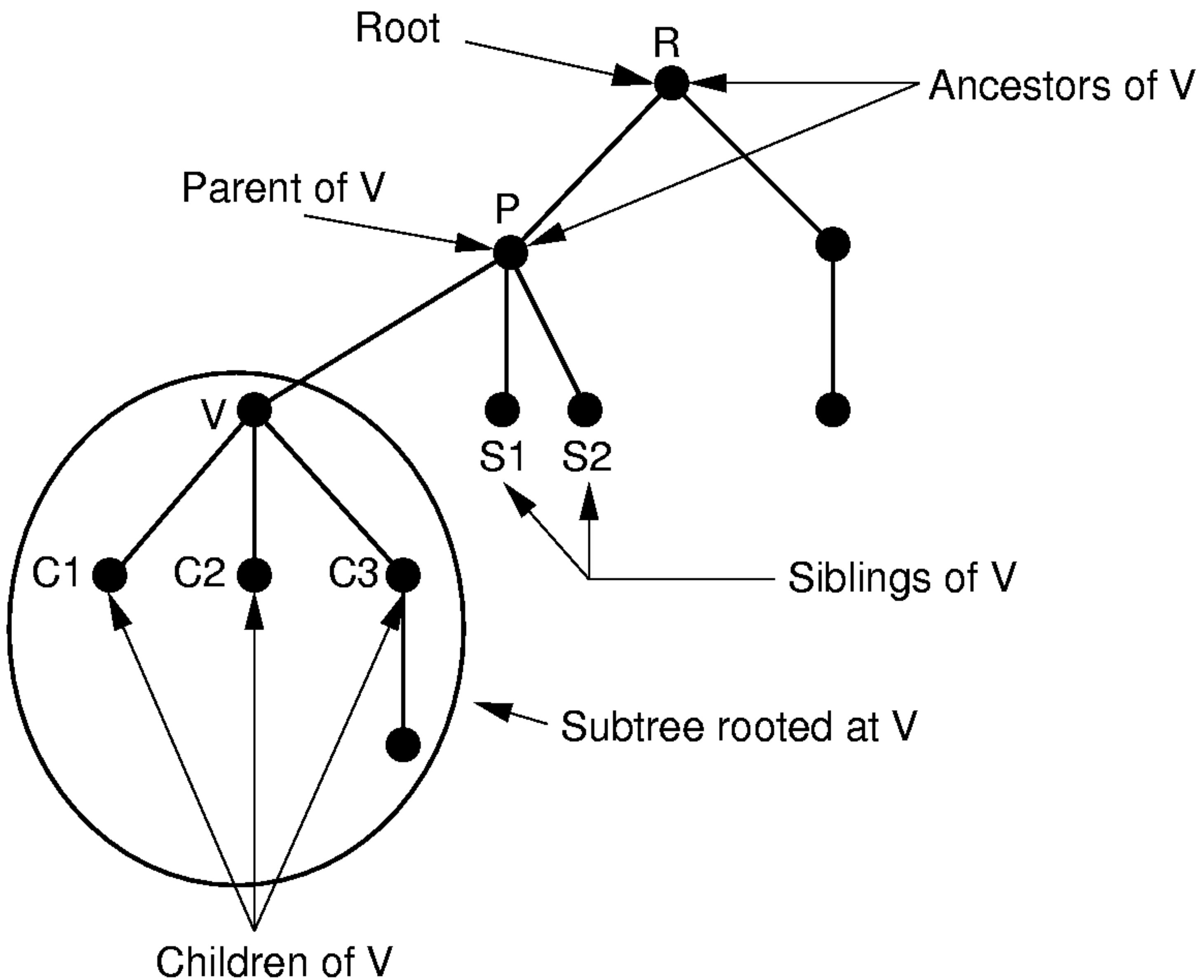
Ordered tree



Ordered trees with four or fewer vertices

Recursive Definitions

- ***A tree*** **T is a finite set of one or more nodes such that there is one designated node r called the root of T, and the remaining nodes in (T –{r}) are partitioned into n≥0 disjoint subsets T_1, T_2, ..., T_k, each of which is a tree, and whose roots r_1, r_2, ..., r_k, respectively, are children of r.**


Root
R
Ancestors of V
Parent of V
P
V
S1
S2
C1
C2
C3
Siblings of V
Subtree rooted at V
Children of V

Recursive Definitions

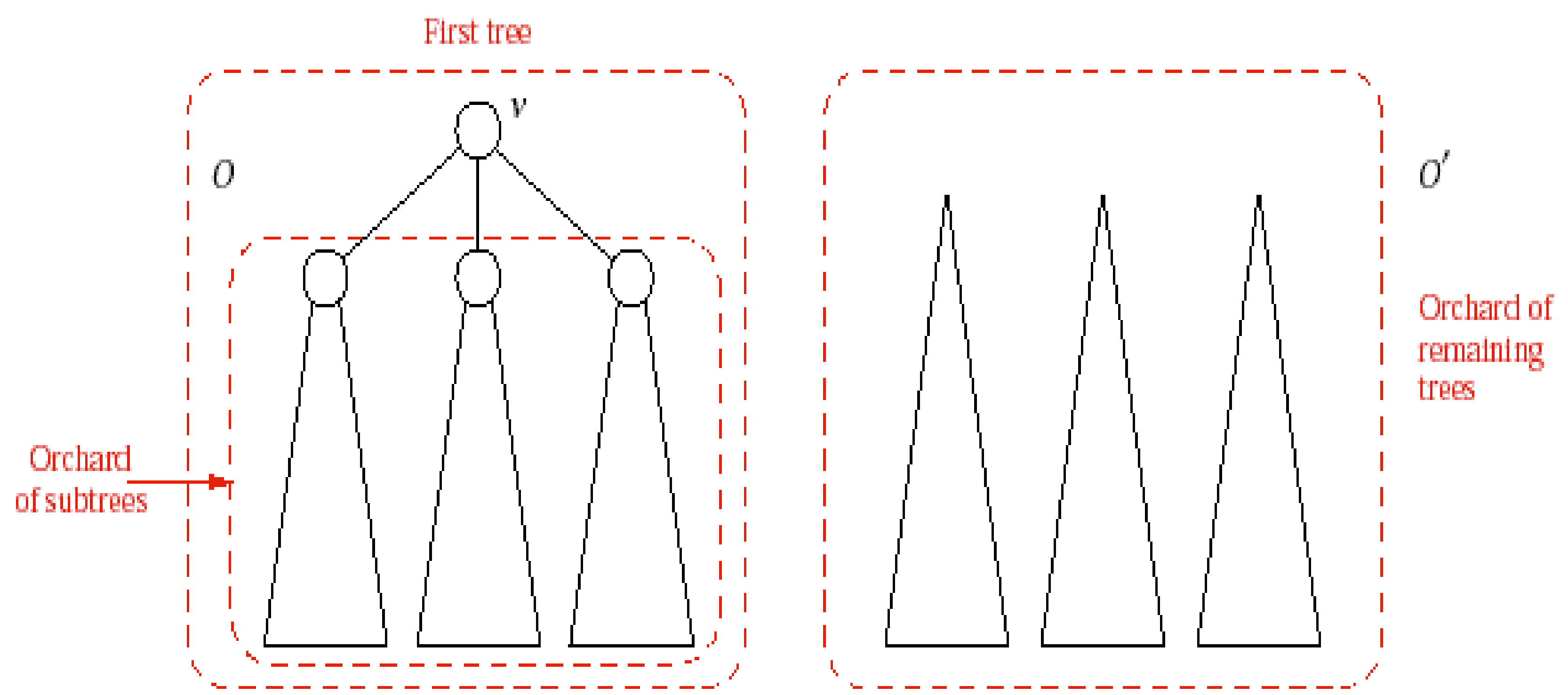
- ***A rooted tree*** **consists of a single vertex v, called the *root* of the tree, together with a *forest* F , whose trees are called the *subtrees* of the root.**

- ***A forest F*** **is a (possibly empty) set of rooted trees.**

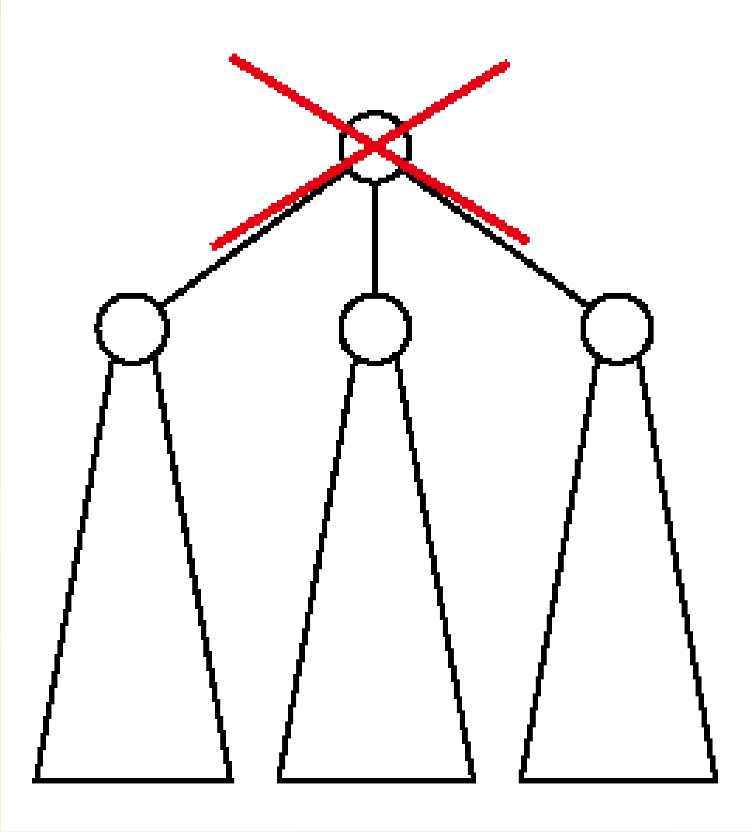
Recursive Definitions

- ***An ordered tree T*** consists of a single vertex v, called the root of the tree, together with an ***orchard*** O, whose trees are called the subtrees of the root *v*. We may denote the ordered tree with the ordered pair $T = \{ v, O \}$
- ***An orchard O*** is either the empty set ∅ , or consists of an ordered tree T , called the first tree of the orchard, together with another orchard O' (which contains the remaining trees of the orchard). We may denote the orchard with the ordered pair $O = \{ T, O' \}$

Orchard tree

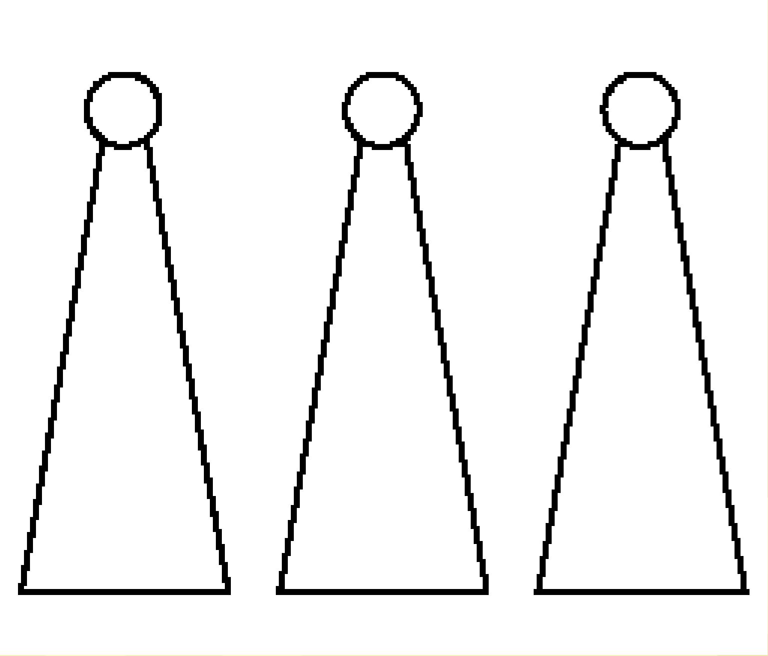


Ordered tree



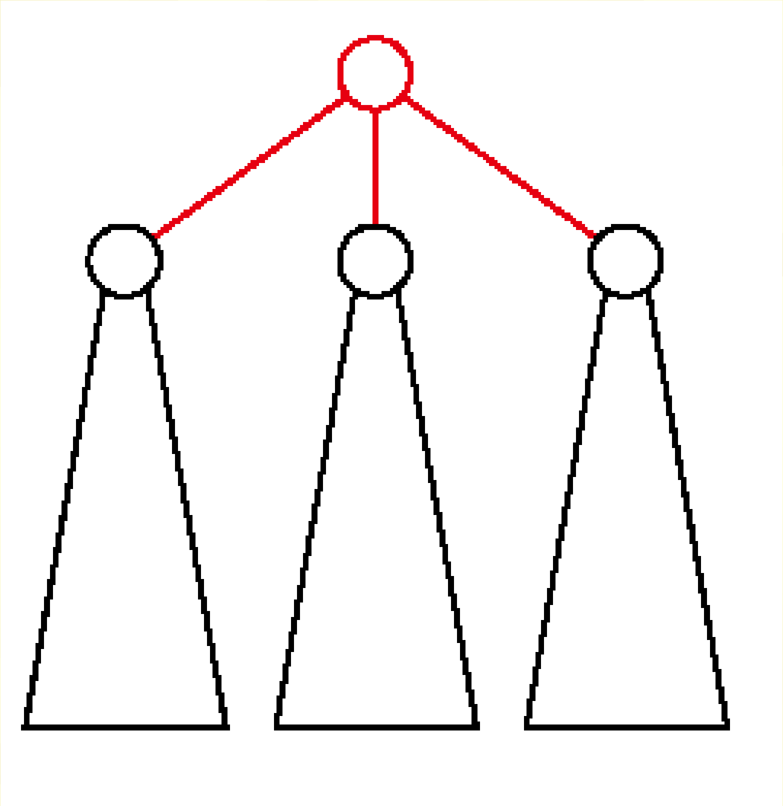
delete root

Orchard



Adjoin new root

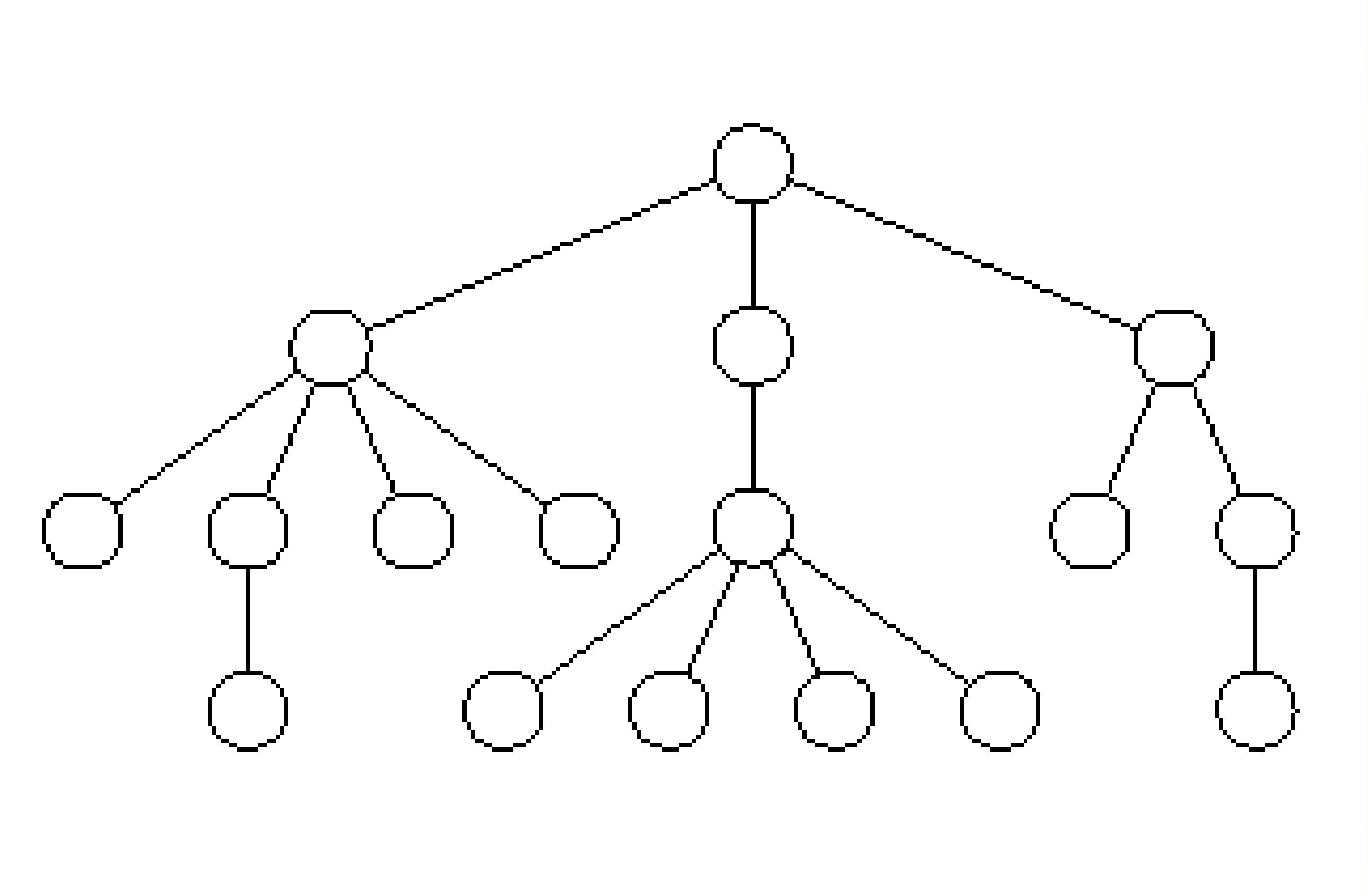
Ordered tree



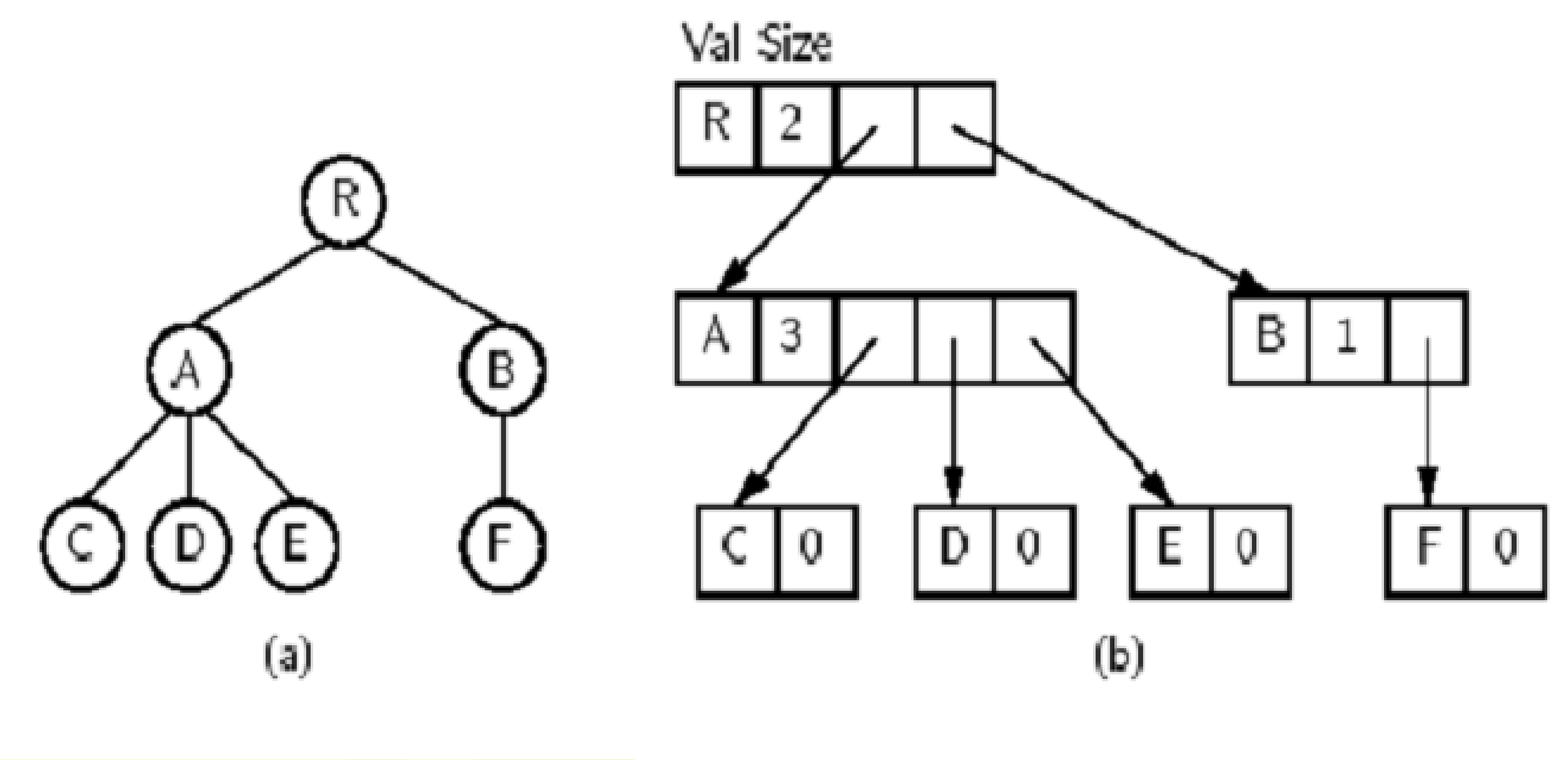
Implementations of Ordered Trees

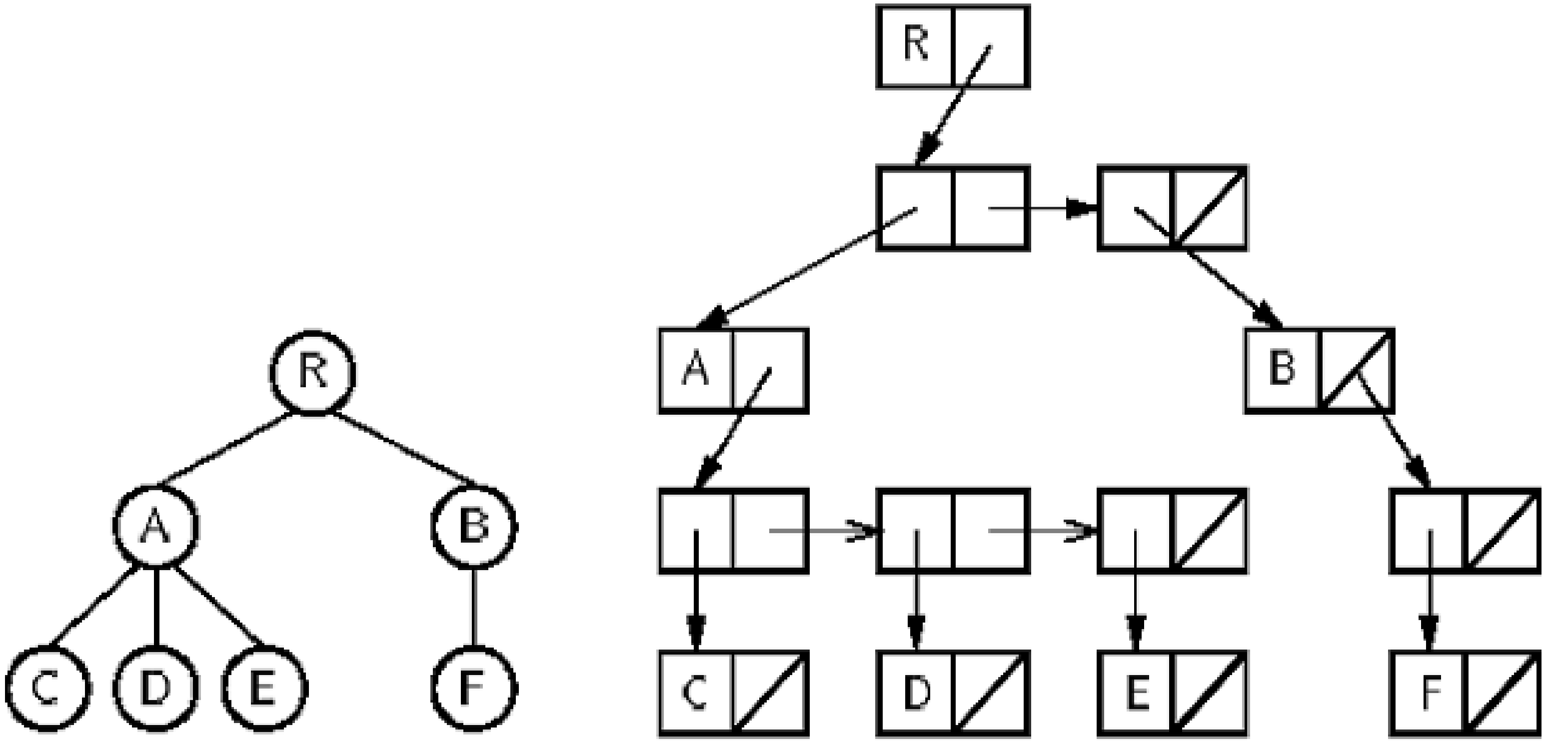
- Multiple links
- Parent Pointer
- Lists of Children
- First child and next sibling links
- Correspondence with binary trees

Multiple links



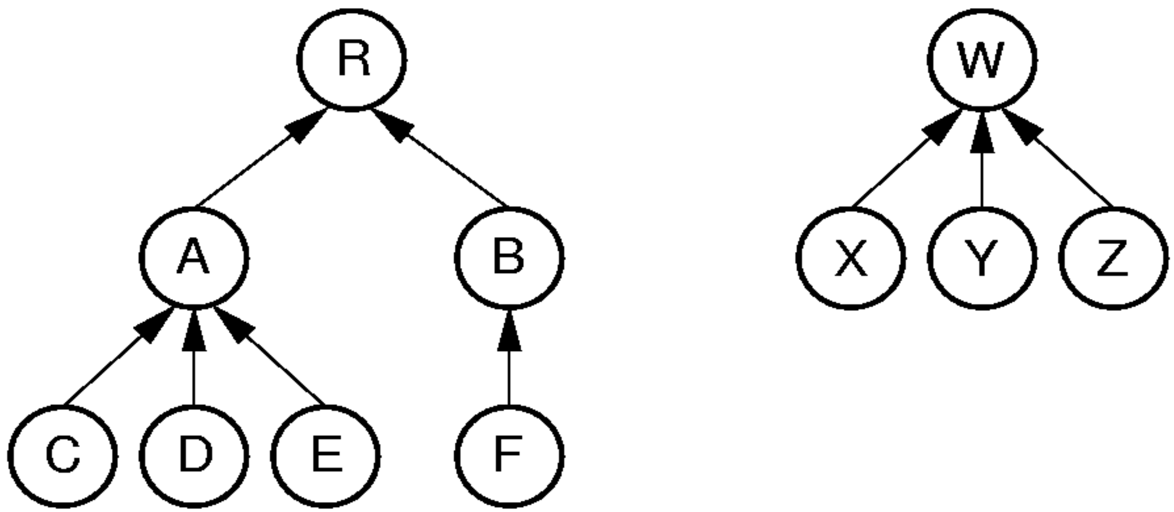
Linked Implementations





(a) (b)

Parent Pointer Implementation



Parent's Index		0	0	1	1	1	2		7	7	7
Label	R	A	B	C	D	E	F	W	X	Y	Z
Node Index	0	1	2	3	4	5	6	7	8	9	10

Union/Find

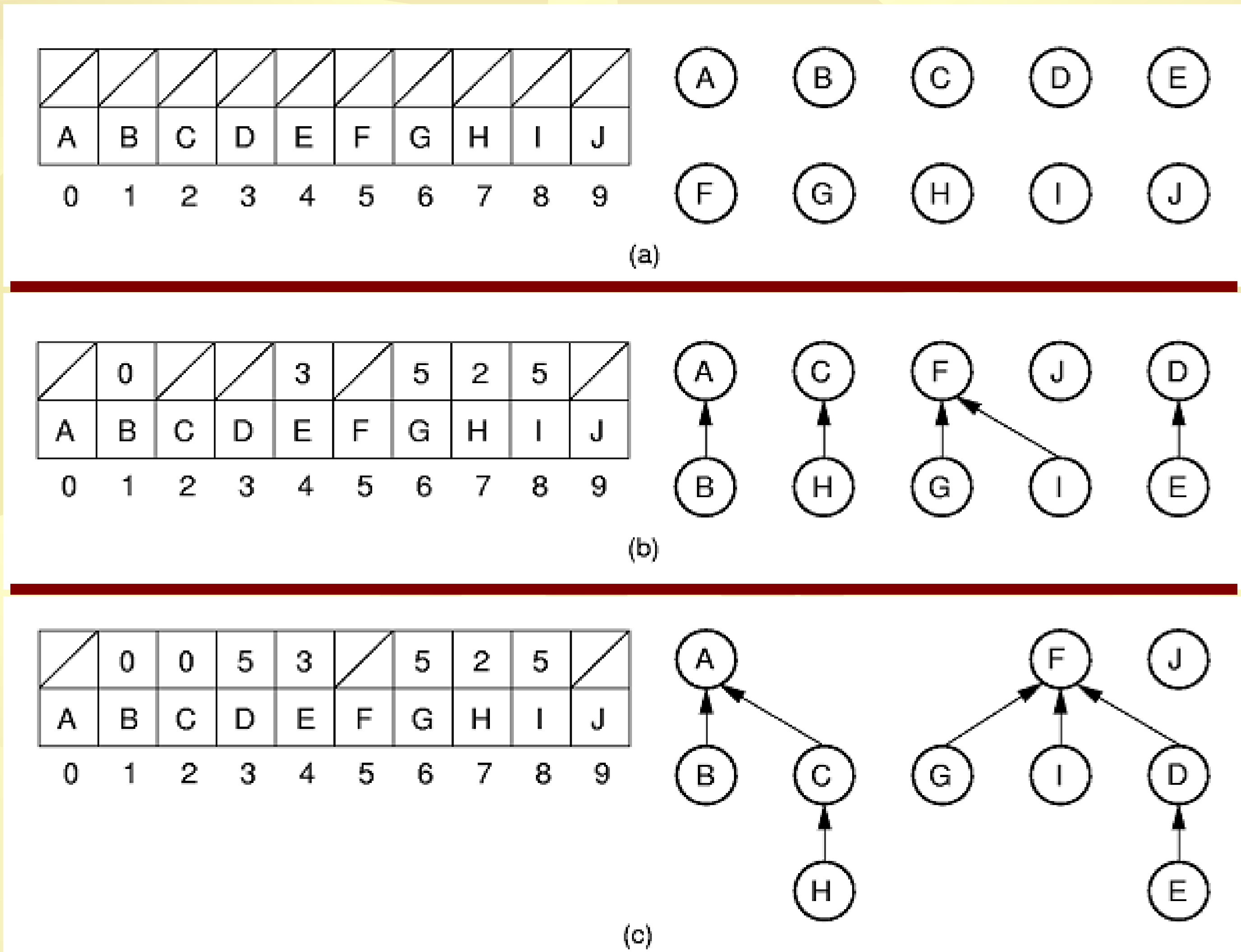
```
void Gentree::UNION(int a, int b) {
int root1 = FIND(a);    // Find root for a
  int root2 = FIND(b); // Find root for b
  if (root1 != root2) array[root2] = root1;
}

int Gentree::FIND(int curr) const {
while (array[curr]!=ROOT) curr = array[curr];
  return curr;  // At root
}
```

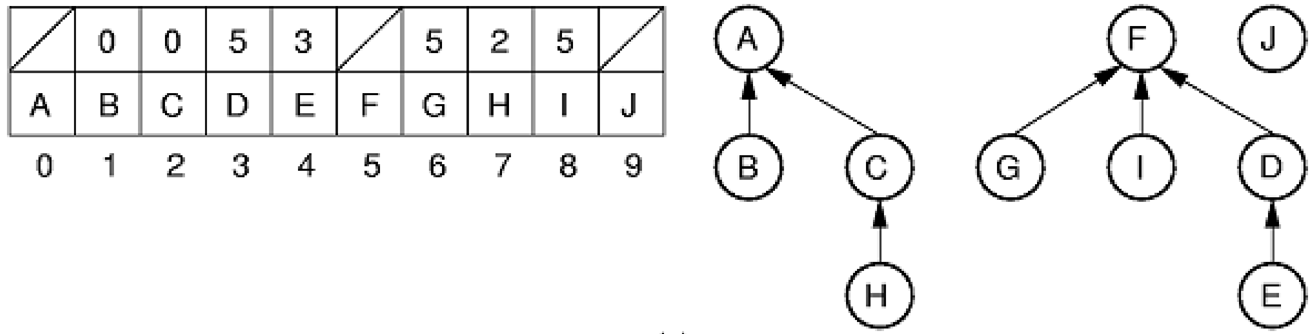
Want to keep the depth small.

Weighted union rule: Join the tree with fewer nodes to the tree with more nodes.

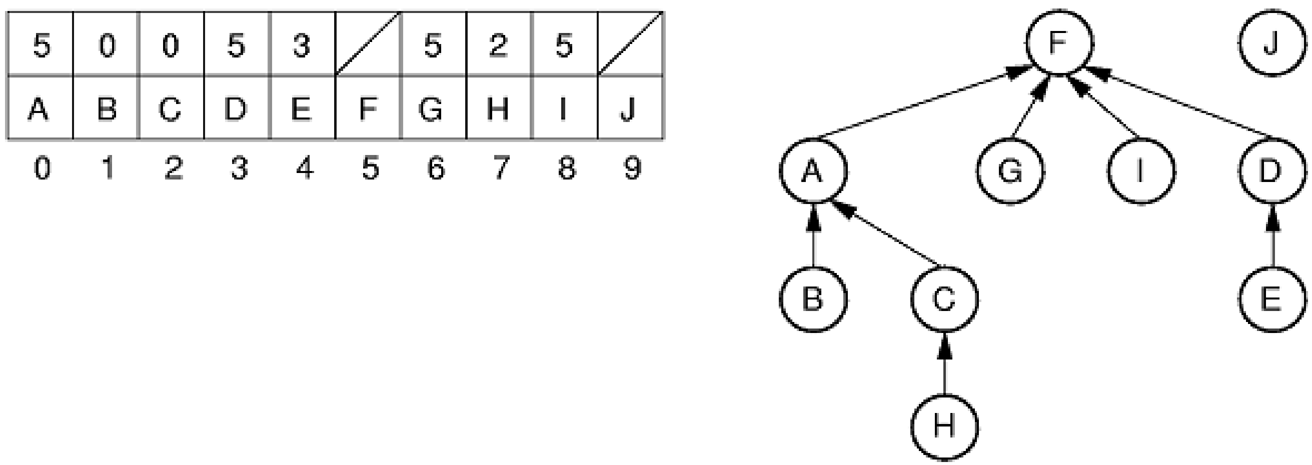
Equiv Class Processing (1)



Equiv Class Processing (2)



(c)

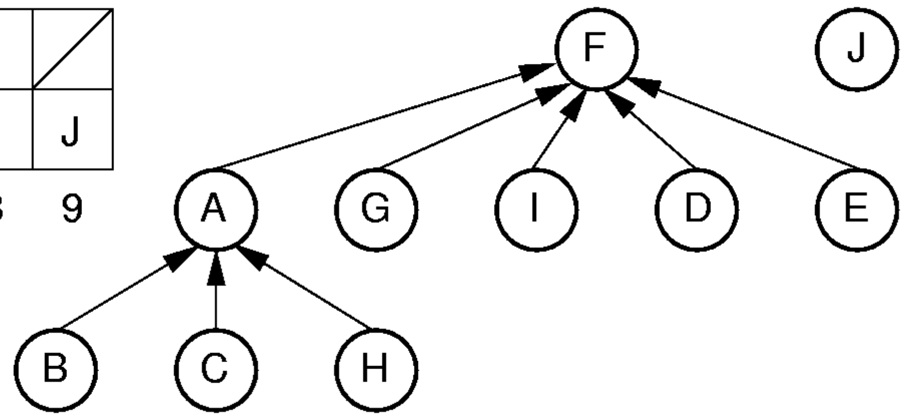


(d)

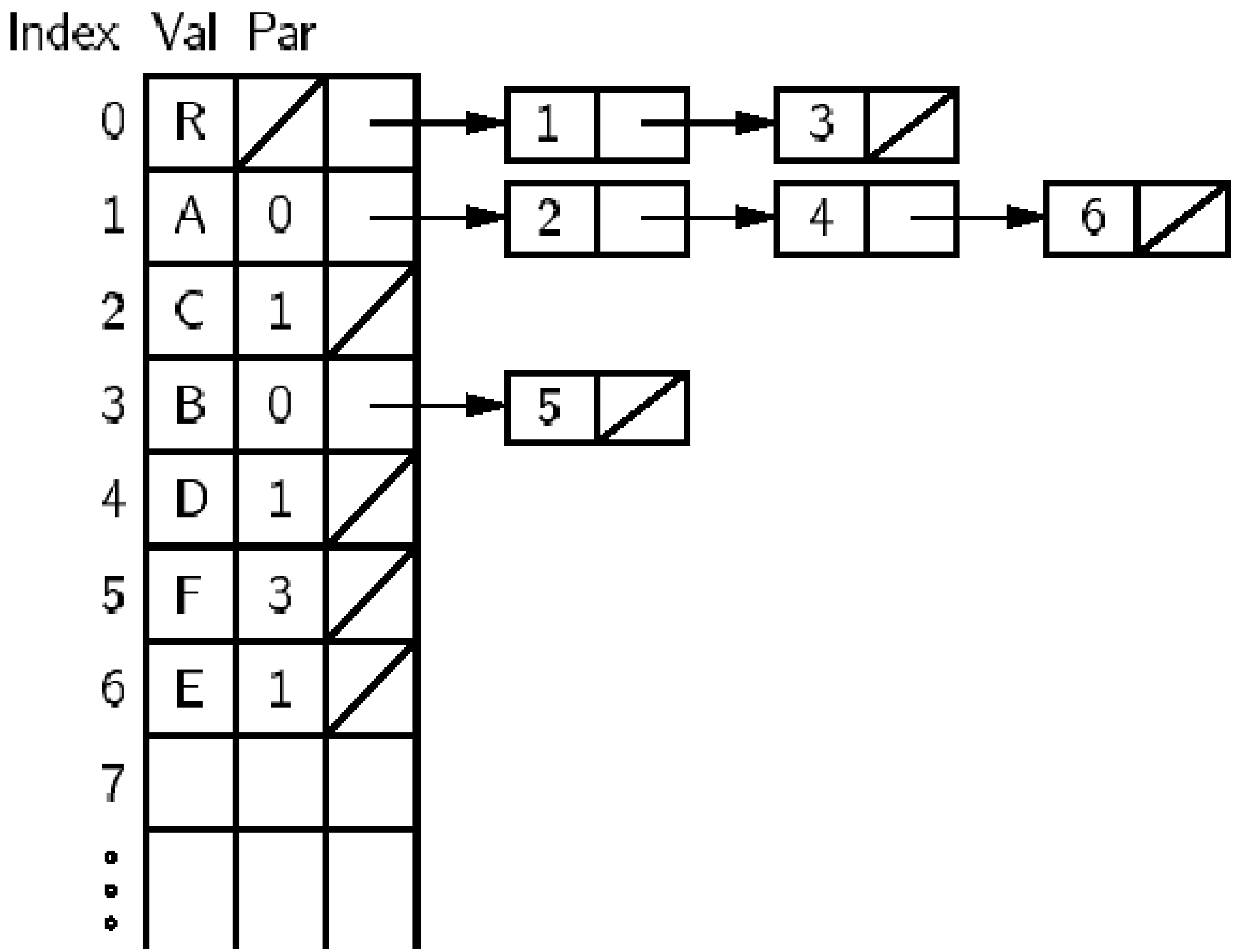
Path Compression

```
int Gentree::FIND(int curr) const {
  if (array[curr] == ROOT) return curr;
  return array[curr] = FIND(array[curr]);
}
```

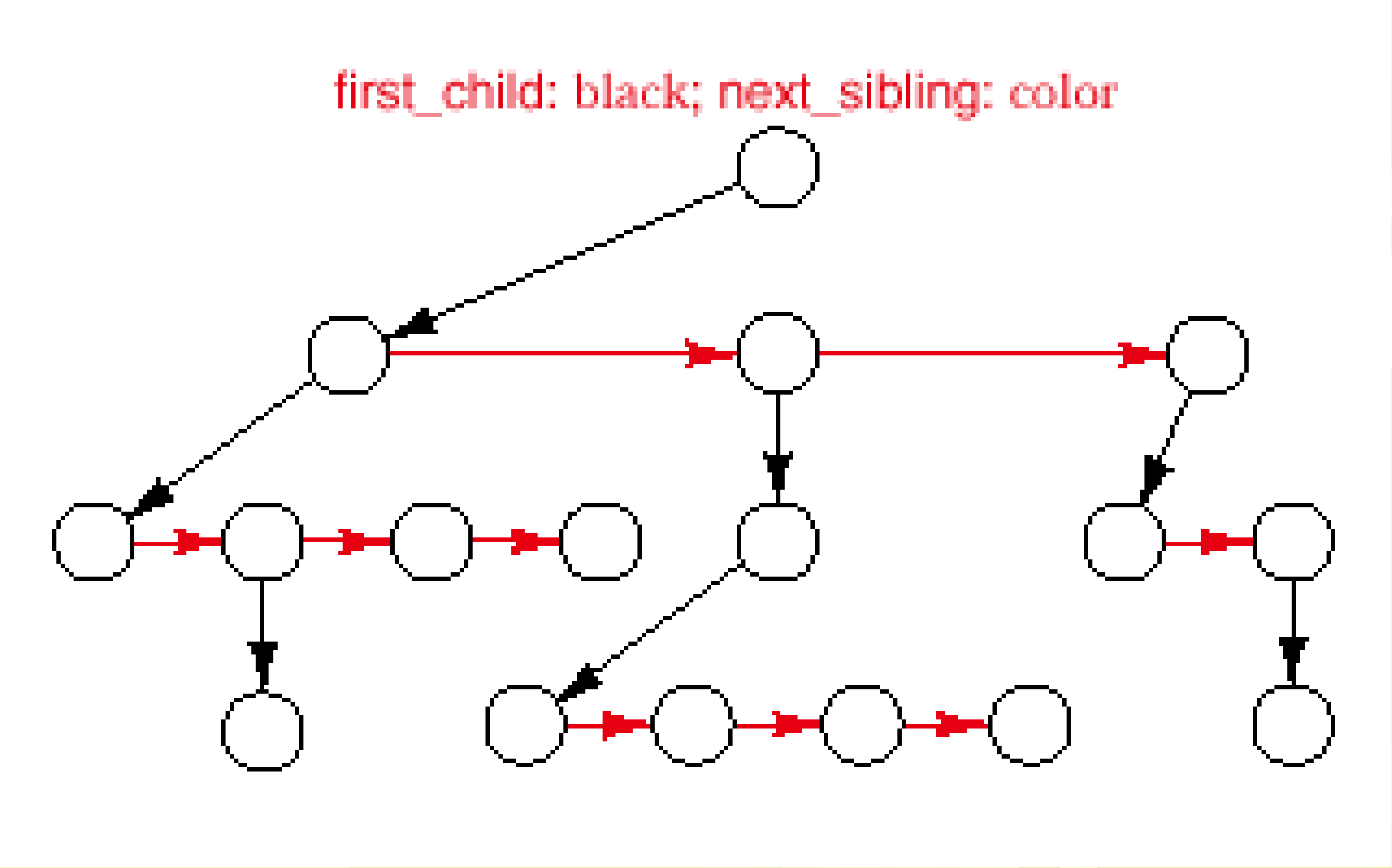
5	0	0	5	5		5	0	5	
A	B	C	D	E	F	G	H	I	J
0	1	2	3	4	5	6	7	8	9



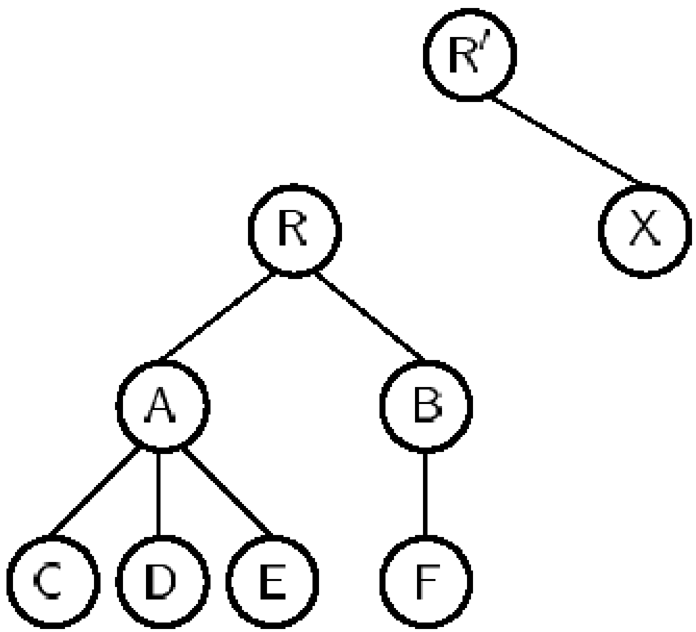
Lists of Children



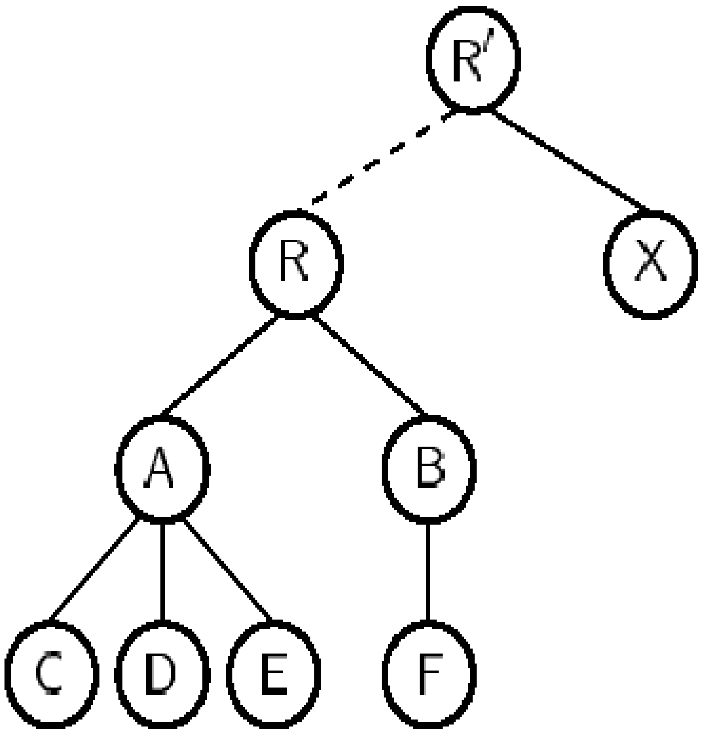
First child and next sibling links



Leftmost Child/Right Sibling

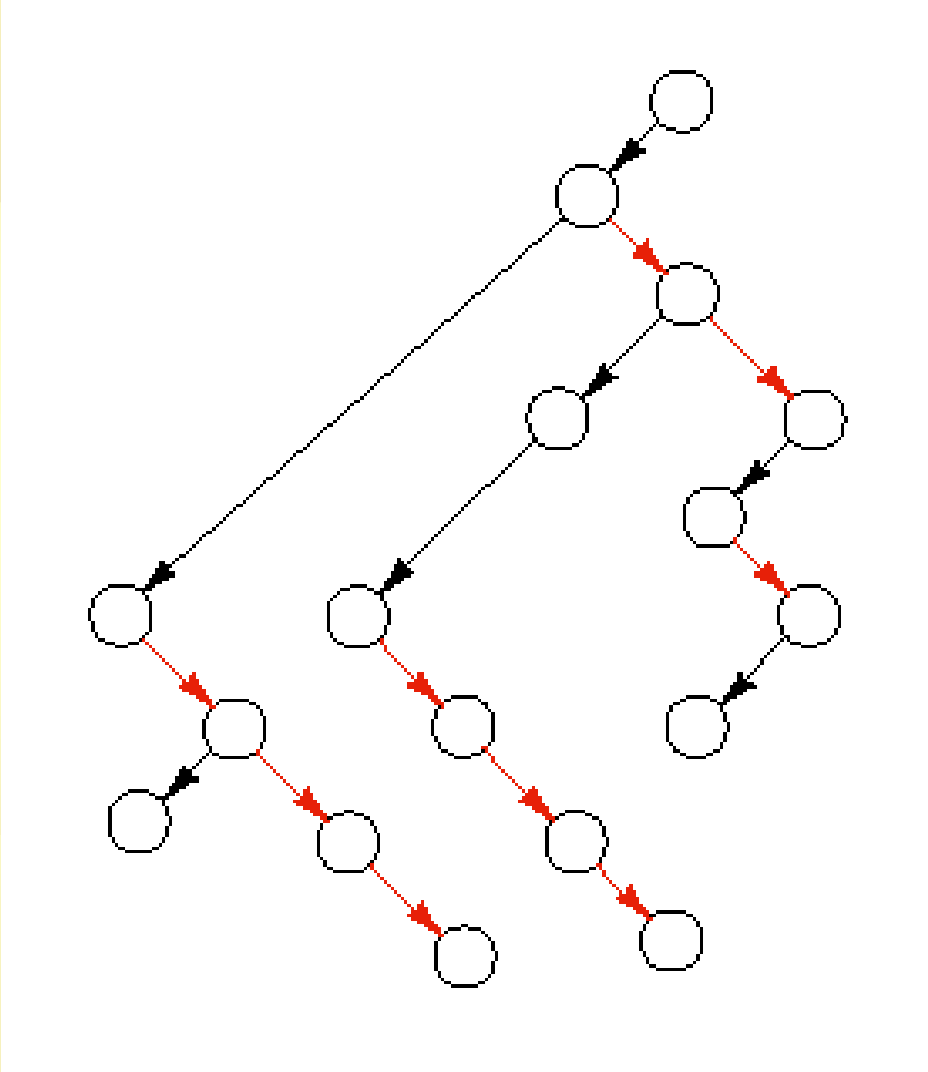


Left	Val	Par	Right
1	R		
3	A	0	2
6	B	0	
	C	1	4
	D	1	5
	E	1	
	F	2	
8	R′		
	X	7	



Left	Val	Par	Right
1	R	7	8
3	A	0	2
6	B	0	
	C	1	4
	D	1	5
	E	1	
	F	2	
0	R′	-1	
	X	7	

Correspondence with binary trees

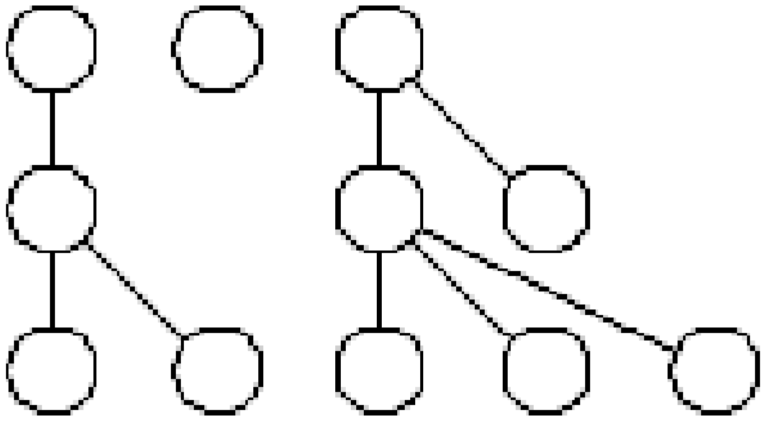


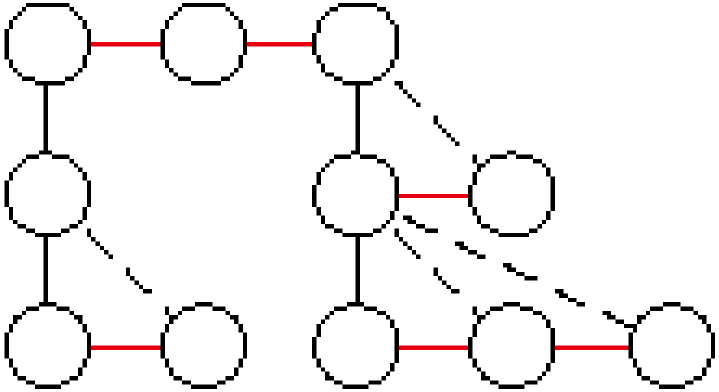
The Formal Correspondence

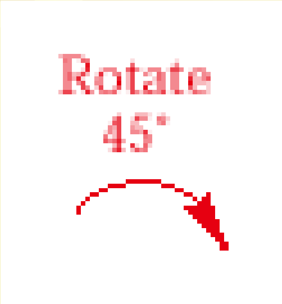
- **DEFINITION:** ***A binary tree B*** **is either the empty set ; or consists of a root vertex v with two binary trees B_1 and B_2 . We may denote the binary tree with the ordered triple B = [v; B_1; B_2].**
- **THEOREM: Let S be any finite set of vertices. There is a one-to-one correspondence f *from* the set of orchards whose set of vertices is S *to* the set of binary trees whose set of vertices is S.**

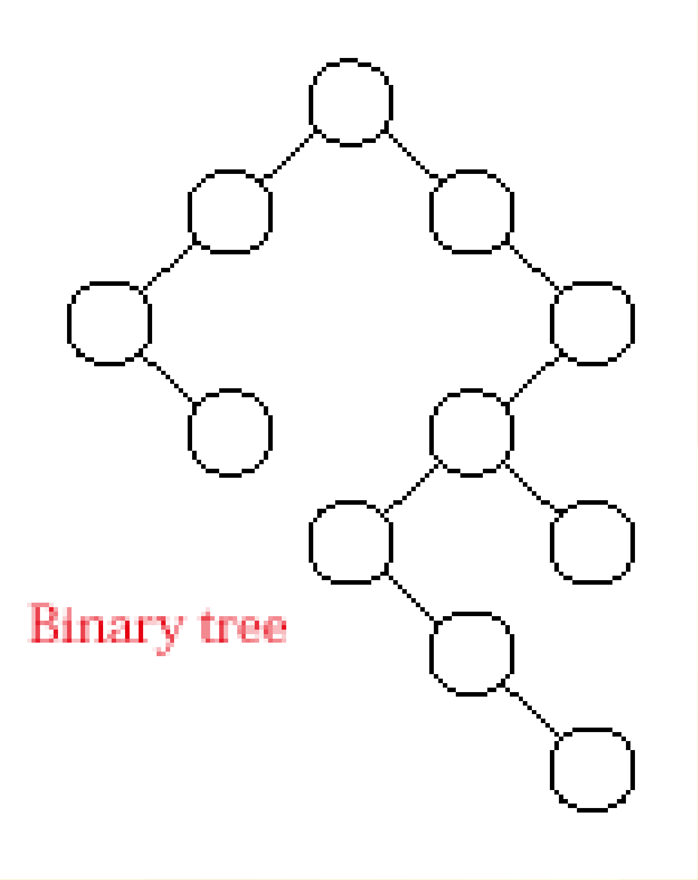
Rotations

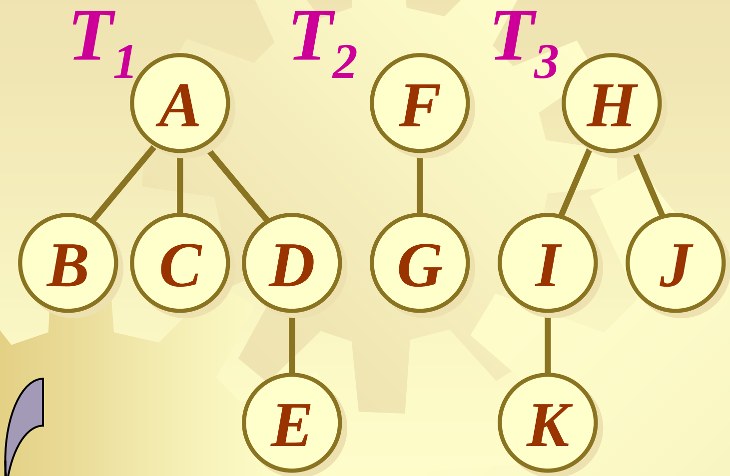
- Draw the orchard so that the first child of each vertex is immediately below the vertex.
- Draw a vertical link from each vertex to its first child, and draw a horizontal link from each vertex to its next sibling.
- Remove the remaining original links.
- Rotate the diagram 45 degrees clockwise, so that the vertical links appear as left links and the horizontal links as right links.


Orchard

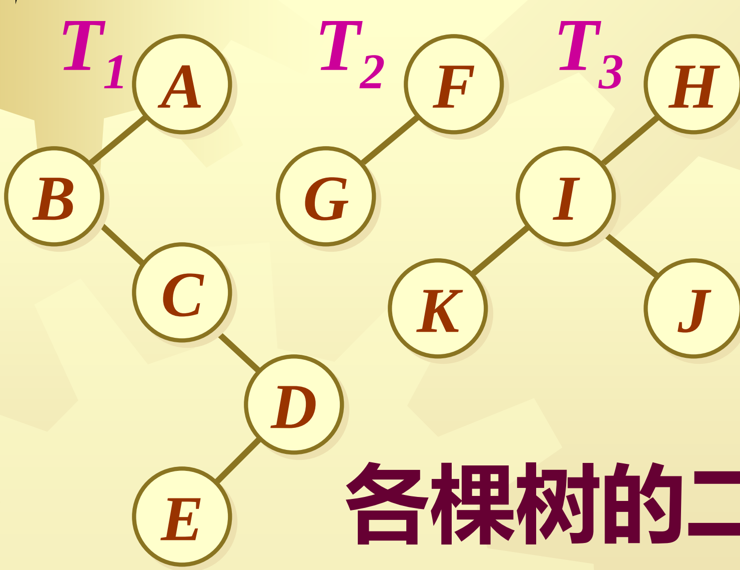

Colored links added,
broken links removed


Rotate
45°

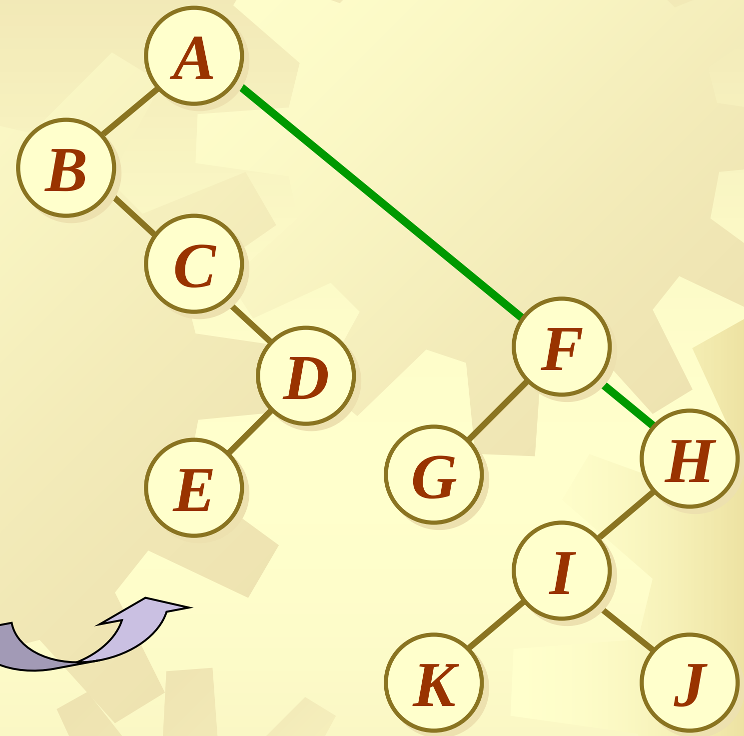

Binary tree



3 棵树的森林



各棵树的二叉树表示



森林的二叉树表示

森林的遍历_深度优先遍历

- 给定森林 F，若 $F = \emptyset$，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、中根遍历两种方法。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。

森林的先根次序遍历

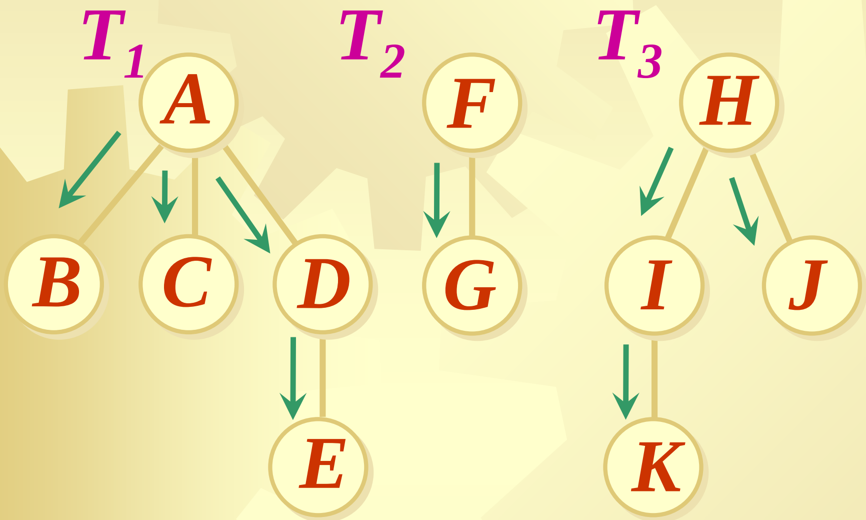
- 若森林F = Ø，返回；否则

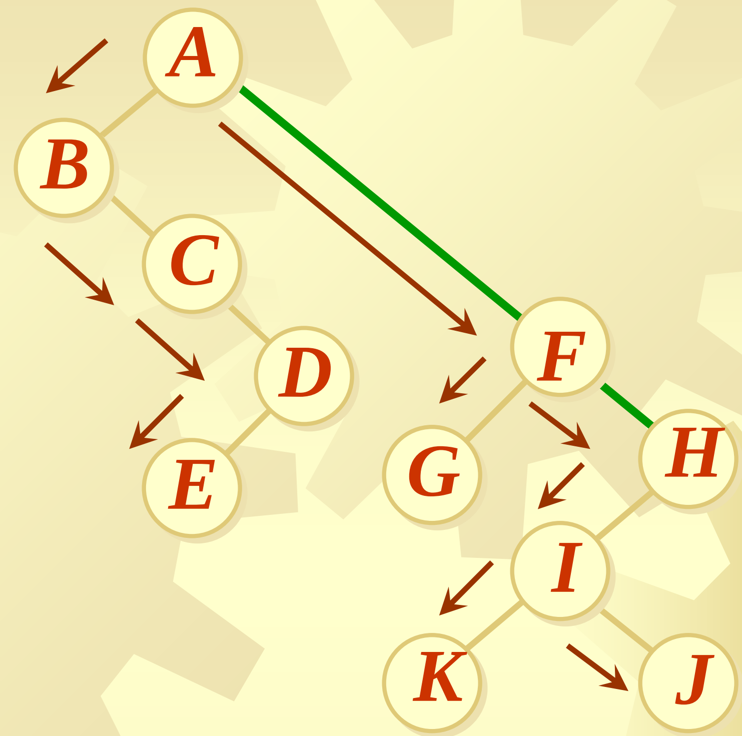
① 访问森林的根（也是第一棵树的根）r_1;

② 先根遍历森林第一棵树的根的子树森林 $\{T_{11}, \ldots, T_{1k}\}$;

③ 先根遍历森林中除第一棵树外其他树组成的森林$\{T_2, \ldots, T_m\}$。

- 注意，② 是一个循环，对于每一个T_{1i}，执行树的先根次序遍历；③ 是一个递归过程，重新执行 T_2 为第一棵树的森林的先根次序遍历。





- 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。

森林的中根次序遍历

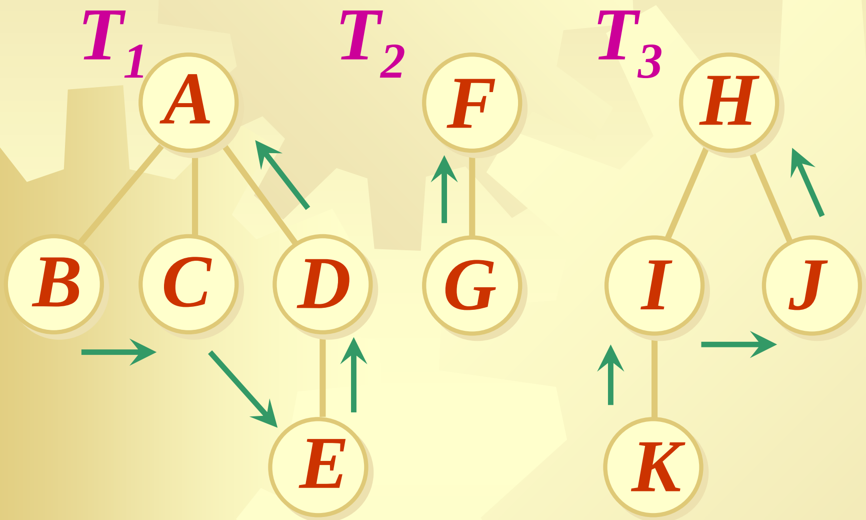
- 若森林 $F = \varnothing$，返回；否则

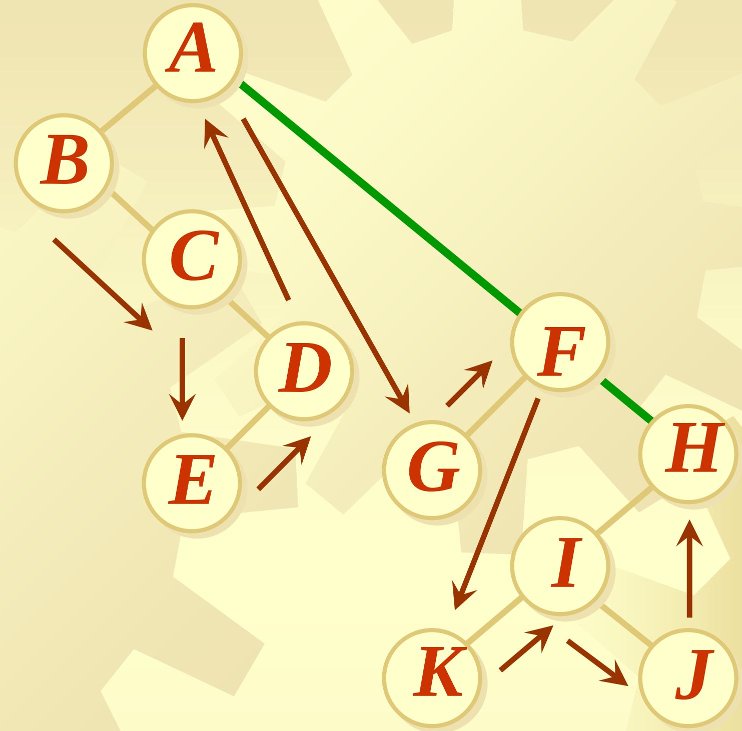
① 中根遍历森林 F 第一棵树的根结点的子树森林$\{T_{11}, \ldots, T_{1k}\}$；

② 访问森林的根结点 r_1；

③ 中根遍历森林中除第一棵树外其他树组成的森林$\{T_2, \ldots, T_m\}$。

- 注意，与先根次序遍历相比，访问根结点的时机不同。所以在③的情形，对以T_2为第一棵树的森林遍历时，重复执行①②③的操作。



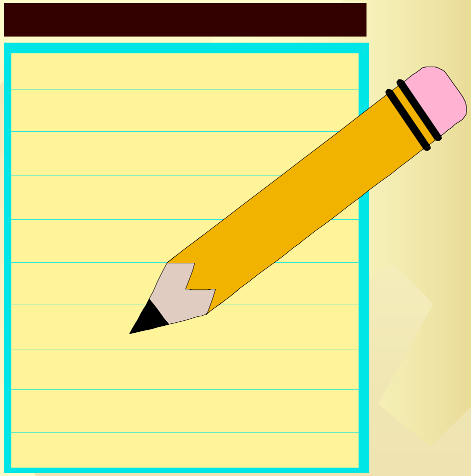


- 森林的中根次序遍历的结果序列

  BCEDA GF KIJH

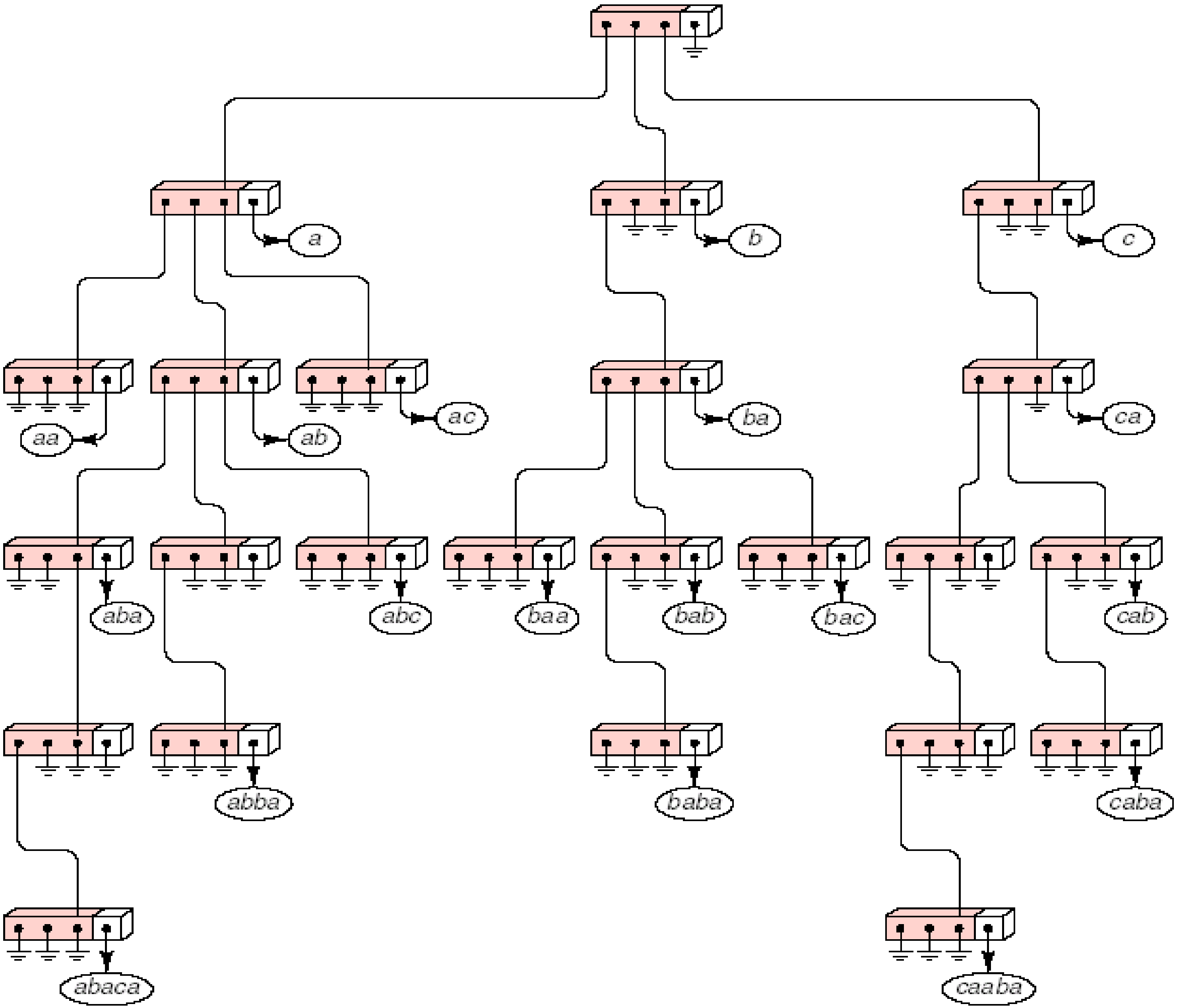
- 这相当于对应二叉树中序遍历的结果。

Questions?



Lexicographic Search Trees: Tries

- **A tries of order m is either empty or consists of an ordered sequence of exactly m tries of order m**



a
b
c
aa
ab
ac
ba
ca
aba
abc
baa
bab
bac
cab
abba
baba
caba
abaca
caaba

C++ Trie Declarations

- **Every Record has a Key that is an alphanumeric string.**
- **Method char *key_letter*(int position) returns the character in the given position of the key or returns a blank, if the key has length less than position.**
- **Auxiliary function int *alphabetic_order*(char symbol) returns the alphabetic position of the character symbol, or 27 for nonblank/non alphabetic characters, or 0 for blank characters.**

C++ Trie Declarations

```
class Trie{
    public:              // Add method prototypes here.
    private:             // data members
        Trie_node*root;
    };

    const int num_chars = 28;

    struct Trie_node{
        Record *data;                // data members
        Trie_node *branch[num_chars];
        Trie_node( );                // constructors};
```

Searching a Trie

- **Error code Trie ::trie_search(const Key &target, Record &x) const**

  / Post: If the search is successful, a code of success is returned, and the output parameter x is set as a copy of the Trie's record that holds target . Otherwise, a code of not_present is returned.*
  *Uses: Methods of class Key . */*

```
{
   int position = 0;
   char next_char;
   Trie_node *location = root;
   while (location != NULL &&
   (next_char = target.key_letter(position)) != ‘ ’)
    {              // Terminate search for a NULL location
                   // or a blank in the target.
        location = location->branch
                        [alphabetic_order (next_char)];
        // Move down the appropriate branch of the trie.
        position ++ ;
        // Move to the next character of the target.
   }
```

```
if ( location != NULL &&
        location->data != NULL) {
        x = *(location->data);
        return success;
    }
    else
        return not_present;
}
```

Insertion into a Trie

```
Error_code Trie ::insert(const Record
    &new_entry)
    /* Post: If the Key of new_entry is already in the Trie,
    a code of duplicate_error is returned. Otherwise, a
    code of success is returned and the Record
    new_entry is inserted into the Trie.
    Uses: Methods of classes Record and Trie_node. */
    {
        Error_code result = success;
        if (root == NULL) root = new Trie_node;
                                    // Create a new empty Trie.
        int position = 0; //indexes letters of new entry
        char next_char;
        Trie_node *location = root; //moves through
    the Trie
```

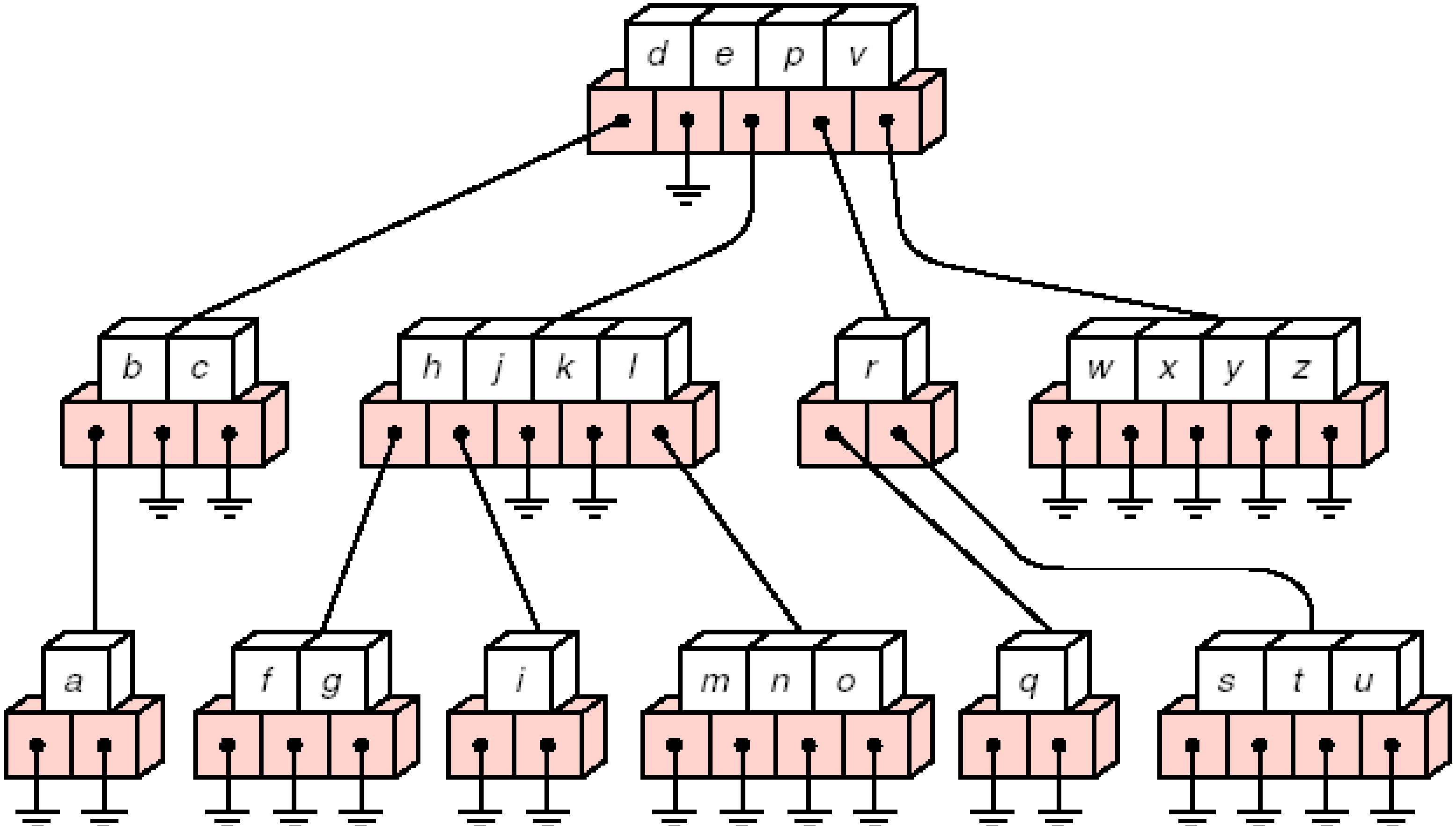
```
while (location != NULL && (next_char =
    new_entry.key_letter(position)) != ' ') {
    int next_position = alphabetic_order(next_char);
    if (location->branch[next_position] == NULL)
        location->branch[next_position] = new
                                        Trie_node;
        location = location->branch[next_position];
        position ++;
    }
    // At this point, we have tested for all nonblank characters of
    new entry .
    if (location->data != NULL) result =
                            duplicate_error;
    else location->data = new Record(new_entry);
    return result;
}
```

- The number of steps required to search a trie or insert into it is proportional to the number of characters making up a key, not to a logarithm of the number of keys as in other tree-based searches.

Multiway Search Trees

- ***An m-way search tree*** **is a tree in which, for some integer m called the order of the tree, each node has at most m children.**
- **If k≤ m is the number of children, then the node contains exactly k -1 keys, which partition all the keys into k subsets consisting of all the keys less than the first key in the node, all the keys between a pair of keys in the node, and all keys greater than the largest key in the node.**

Multiway Search Trees



Questions?