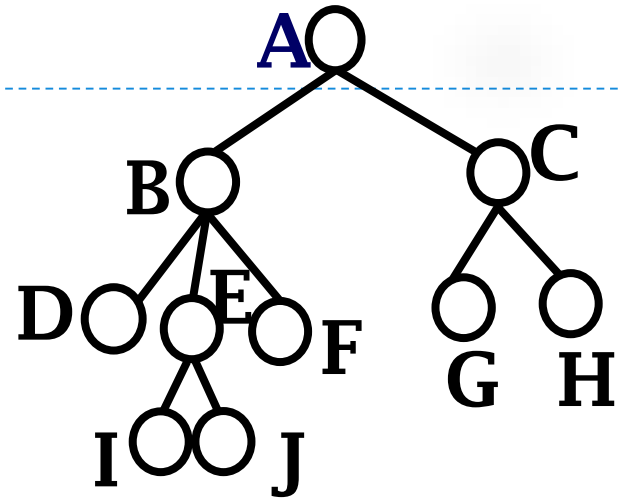
Data Structures and Algorithms (6)

Instructor: Ming Zhang
Textbook Authors: Ming Zhang, Tengjiao Wang and Haiyan Zhao
Higher Education Press, 2008.6 (the "Eleventh Five-Year" national planning textbook)

https://courses.edx.org/courses/PekingX/04830050x/2T2014/

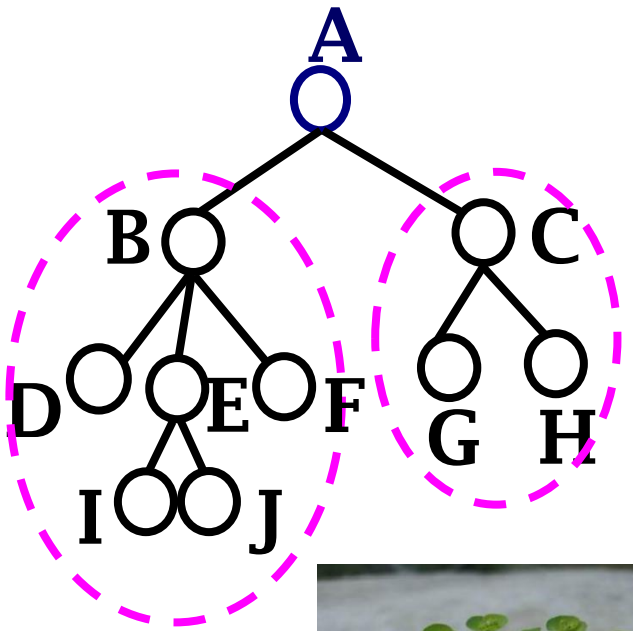
Chapter 6 Trees



- General Definitions and Terminology of Tree
  - Trees and Forest
  - Equivalent Transformation between a Forest and a Binary Tree
  - Abstract Data Type of Tree
  - General Tree Traversals
- Linked Storage Structure of Tree
- Sequential Storage Structure of Tree
- K-ary Trees

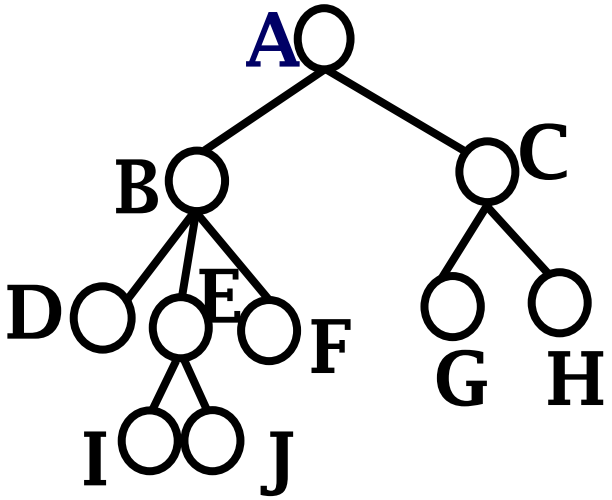
Trees and Forest

- A tree T is a finite set of one or more nodes :
  - there is one specific node R, called the root of T
  - If the set T-{R} is not empty, these nodes are partitioned into m > 0 disjoint finite subsets T_1 , T_2 , … , T_m, each of which is a tree. The subsets T_i are said to be subtrees of T.
  - Directed ordered trees: the relative order of subtrees is important
- An ordered tree with degree 2 is not a binary tree
  - After the first child node is deleted
  - The second child node will take the first child node's place

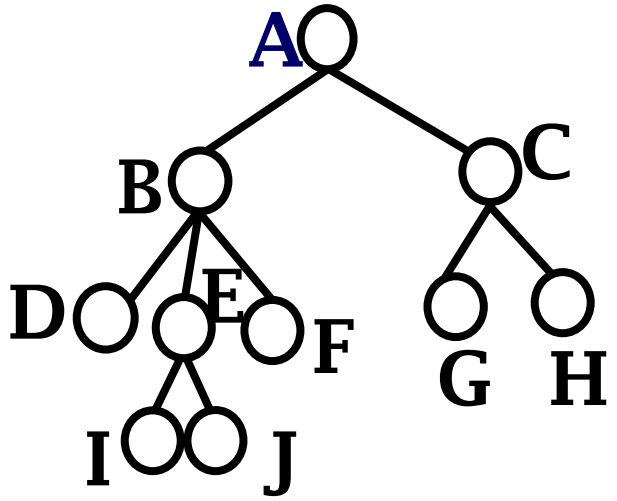


Logical Structure of Tree

- A finite set K of n nodes, and a relation r satisfying the following conditions:
  - There is a unique node $k_0 \in K$, who has no predecessor in relation r.
    - Node k_0 is called the root of the tree.
  - Except k_0, all the other nodes in K has a unique predecessor in relation r
- An example as in the figure on the right
  - Node set K = { A, B, C, D, E, F, G, H, I, J }
  - The relation on K: r = { <A, B>, <A, C>, <B, D>, <B, E>, <B, F>, <C, G>, <C, H>, <E, I>, <E, J> }

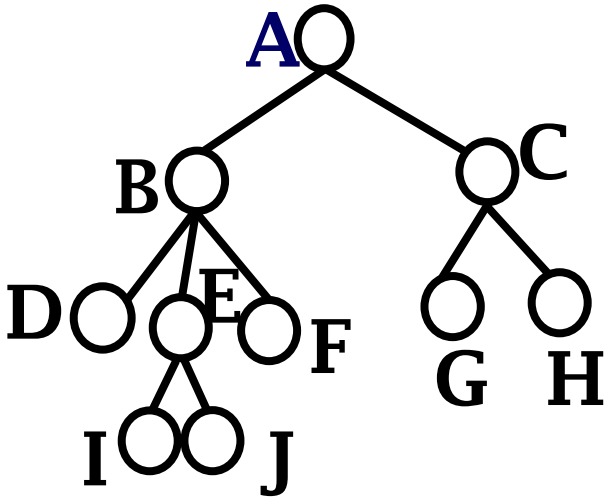


Terminology of Tree



- **Node**
  - **Child node, parent node, the first child node**
    - If <k, k′> ∈ r, we call that k is the parent node of k′, and k′ is the child node of k
  - **Sibling node, previous/next sibling node**
    - If <k, k′> ∈ r and <k, k″>∈ r, we call k′ and k″ are sibling nodes
  - **Branch node, leaf node**
    - Nodes who have no subtrees are called leaf nodes
    - Other nodes are called branch nodes

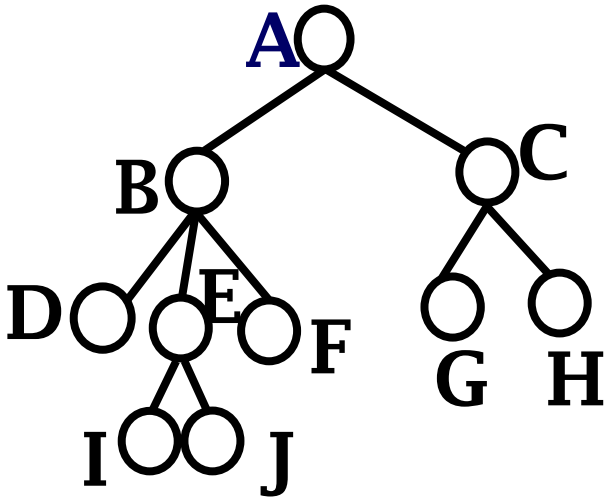
Terminology of Tree



- **Edge**
  - The ordered pair of two nodes is called an edge
- **Path, path length**
  - Except the node k_0, for any other node $k \in K$, there exists a node sequence $k_0, k_1, \ldots, k_s$, s.t. k_0 is the root node, $k_s=k$, and $<k_{i-1}, k_i> \in r$ $(1 \le i \le s)$.
  - This sequence is called a path from the root node to node k, and the path length (the total number of edges in the path) is s
- **Ancestor, descendant**
  - If there is a path from node k to node k_s, we call that k is an ancestor of k_s, and k_s is a descendant of k

Terminology of Tree

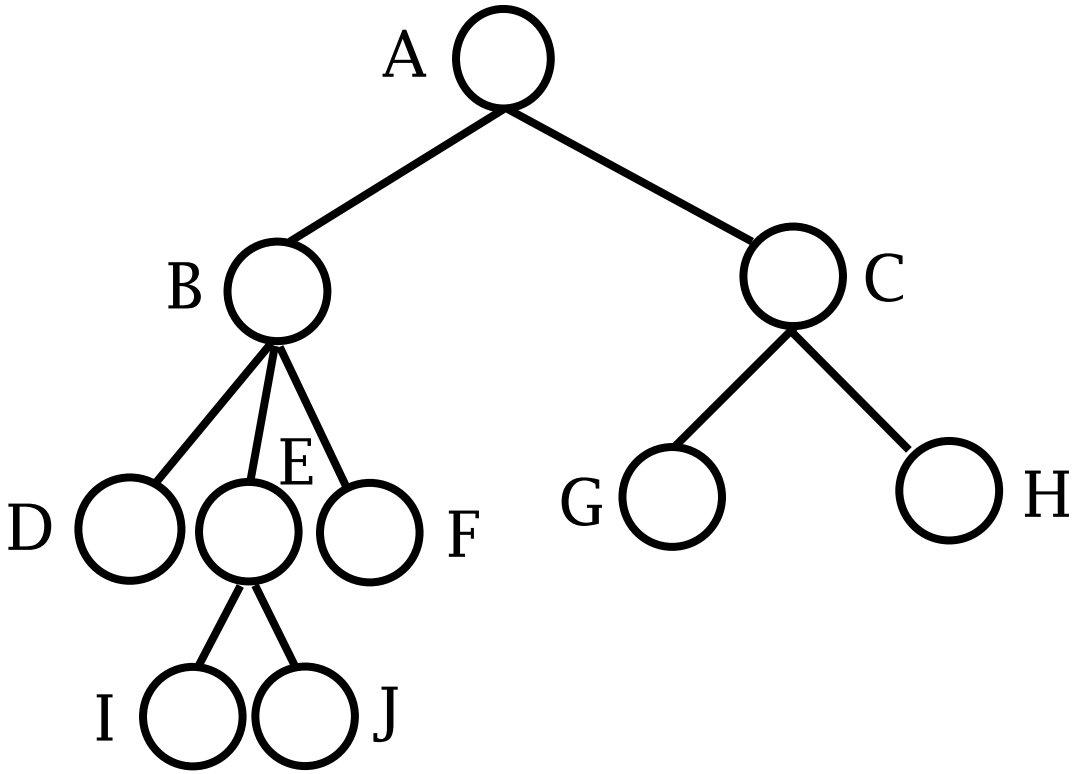
- **Degree**: The degree of a node is the number of children for that node.
- **Level**: The root node is at level 0
  - The level of any other node is the level of its parent node plus 1
- **Depth**: The depth of a node M in the tree is the path length from the root to M.
- **Height**: The height of a tree is the depth of the deepest node in the tree plus 1.



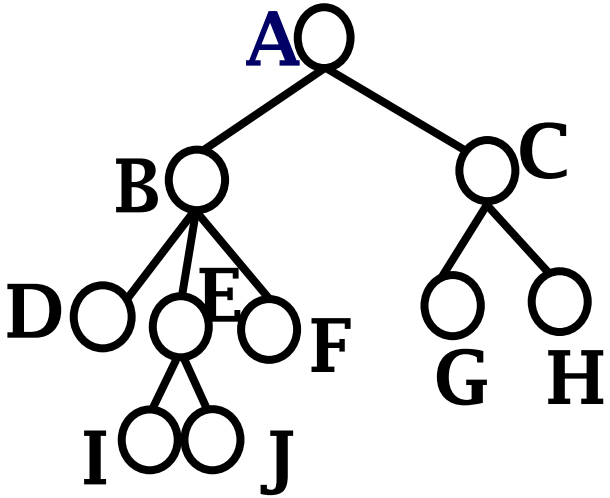
Different Representations of Trees

- Classic node-link representation
- Formal (set theory) representation
- Venn diagram representation
- Outline representation
- Nested parenthesis representation

Node-Link Representation



Formal Representation



The logical structure of a Tree is:

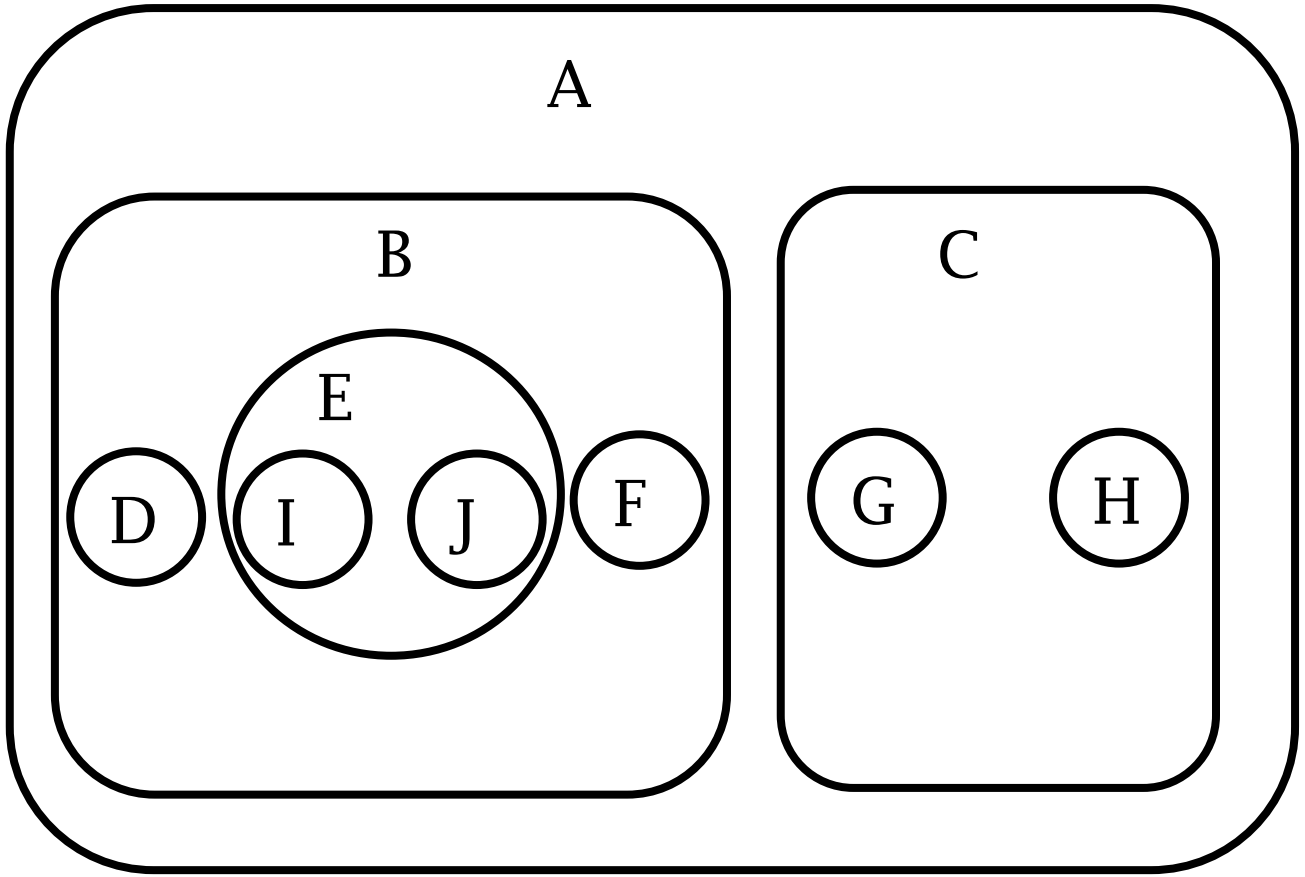
Node set:

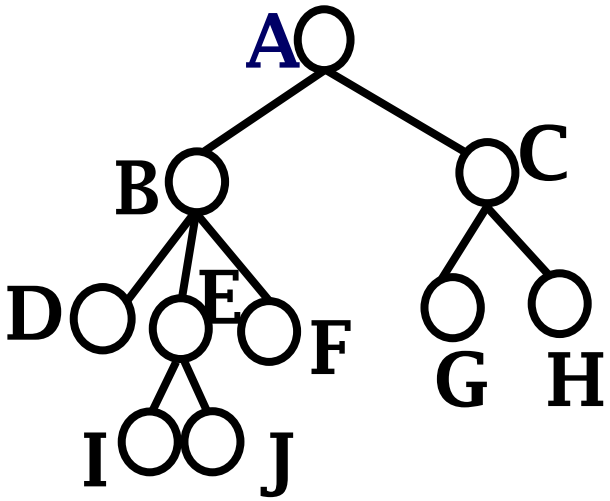
K = {A, B, C, D, E, F, G, H, I, J}

The relation on K:

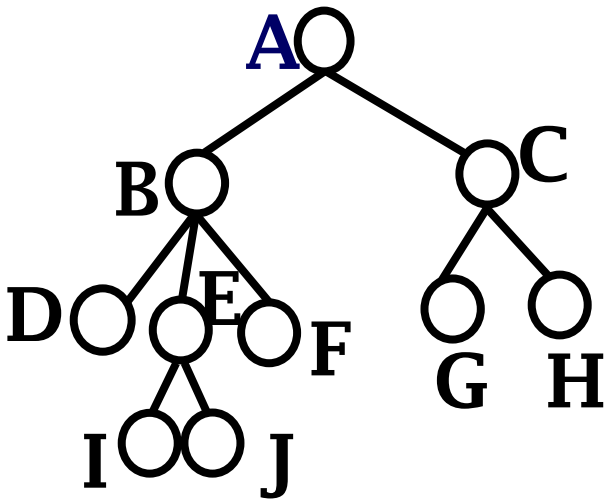
N = {<A, B>, <A, C>, <B, D>, <B, E>, <B, F>, <C, G>, <C, H>, <E, I>, <E, J>}

Venn Diagram Representation



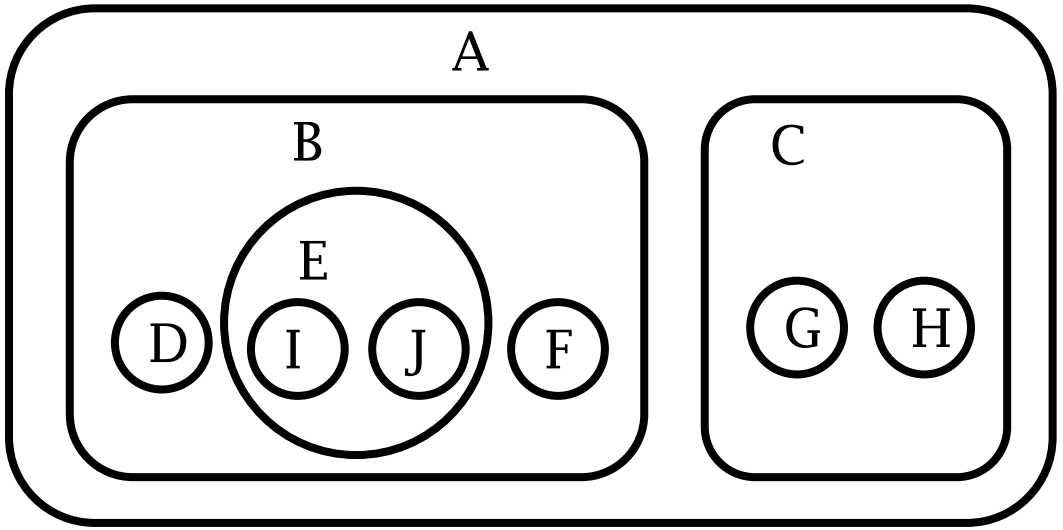


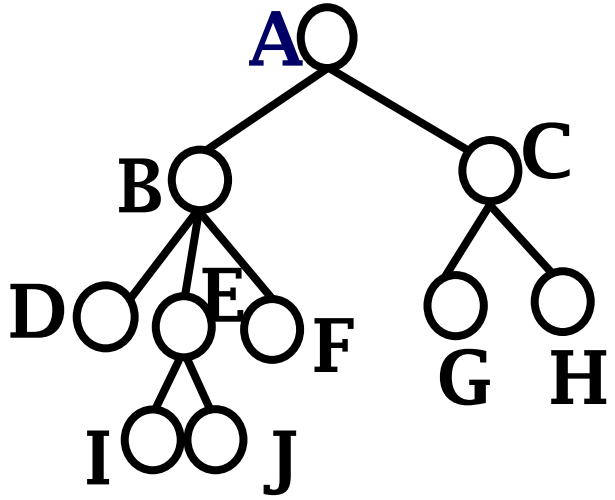
Nested Parenthesis Representation



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

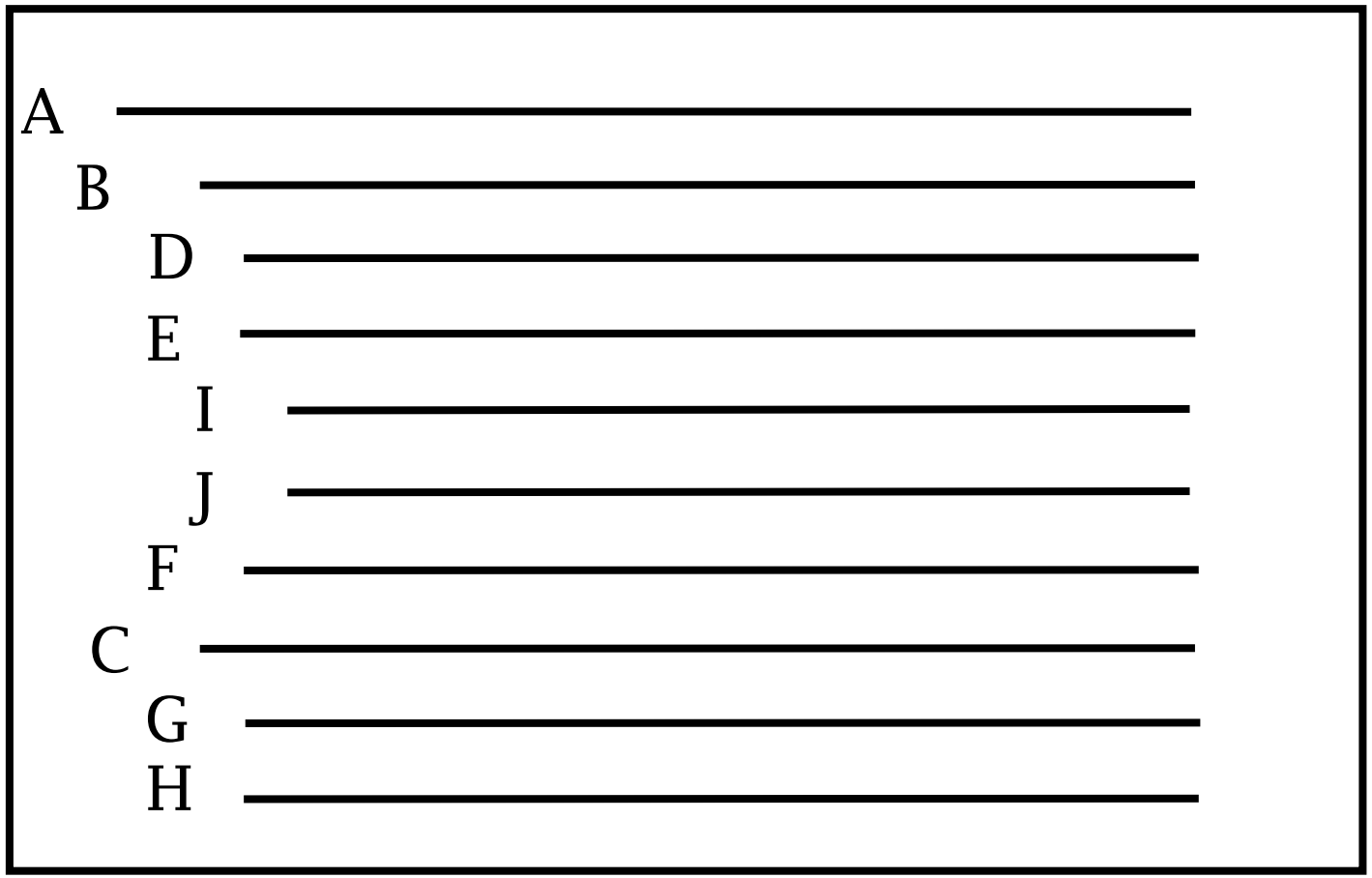
The conversion from Venn diagram to nested parenthesis

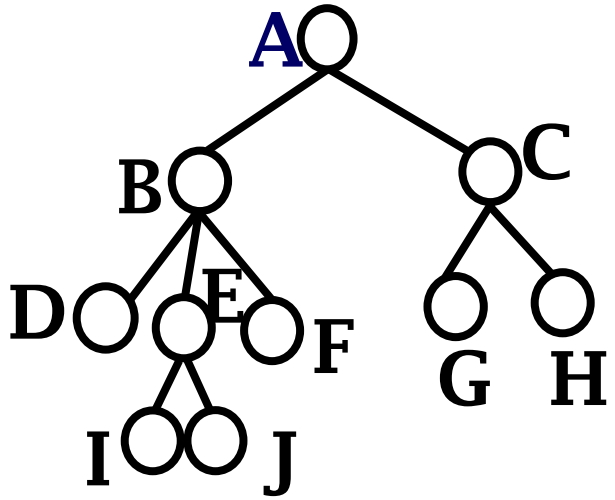




(A(B(D)E(I)(J))(F))(C(G)(H)))

Outline Representation

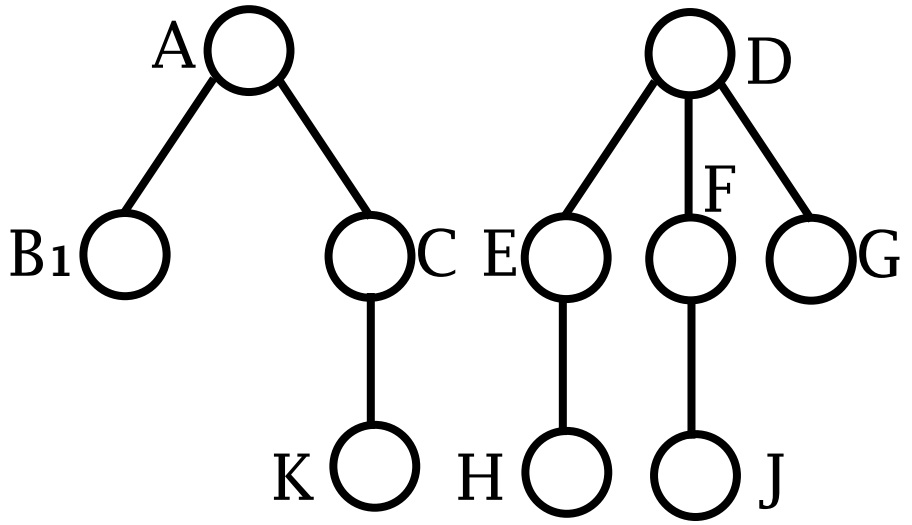




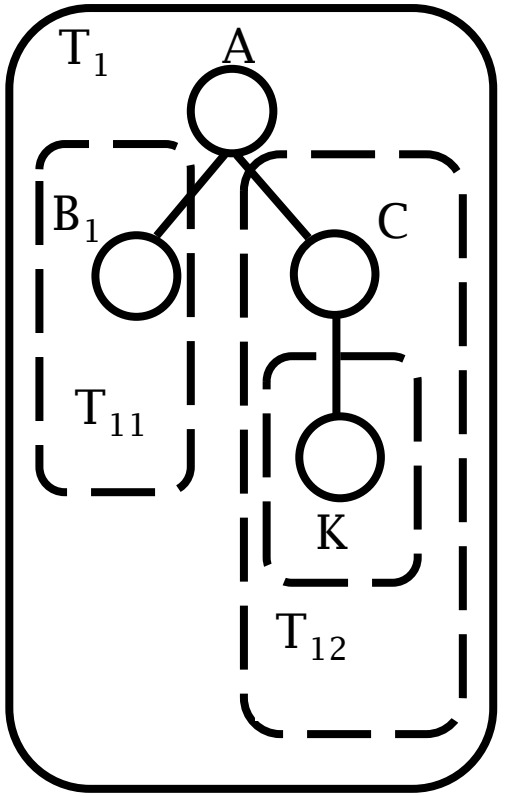
Book catalogue, Dewey representation

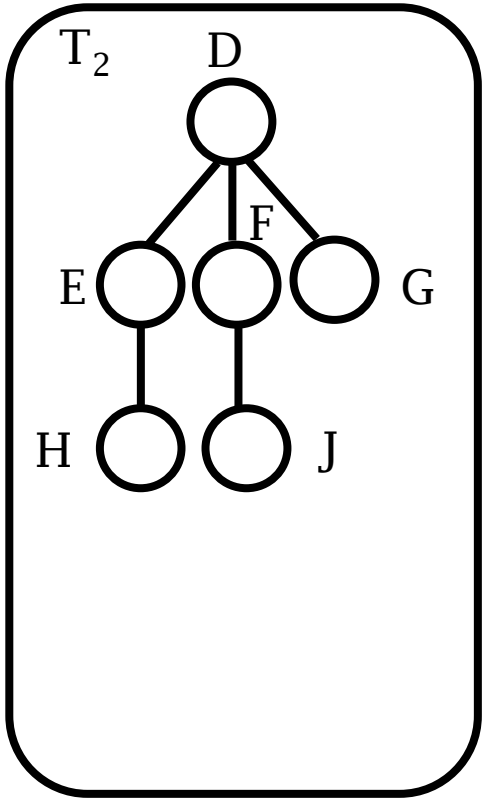
Equivalent Transformation between a Forest and a Binary Tree

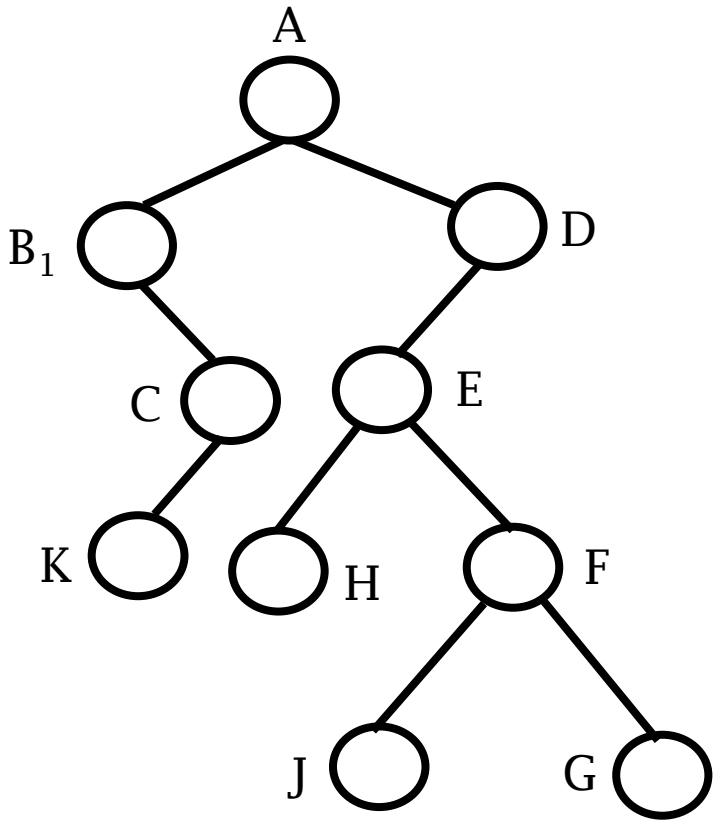
- **Forest**: A forest is a collection of one or more disjoint trees. (usually ordered)
- The correspondence between trees and a forests
  - Removing the root node from a tree, its subtrees become a forest.
  - Adding an extra node as the root of the trees in a forest, the forest becomes a tree.
- There is a one-to-one mapping between forests and binary trees
  - So that all the operations on forests can be transformed to the operations on binary trees



How to map a forest to a binary tree?

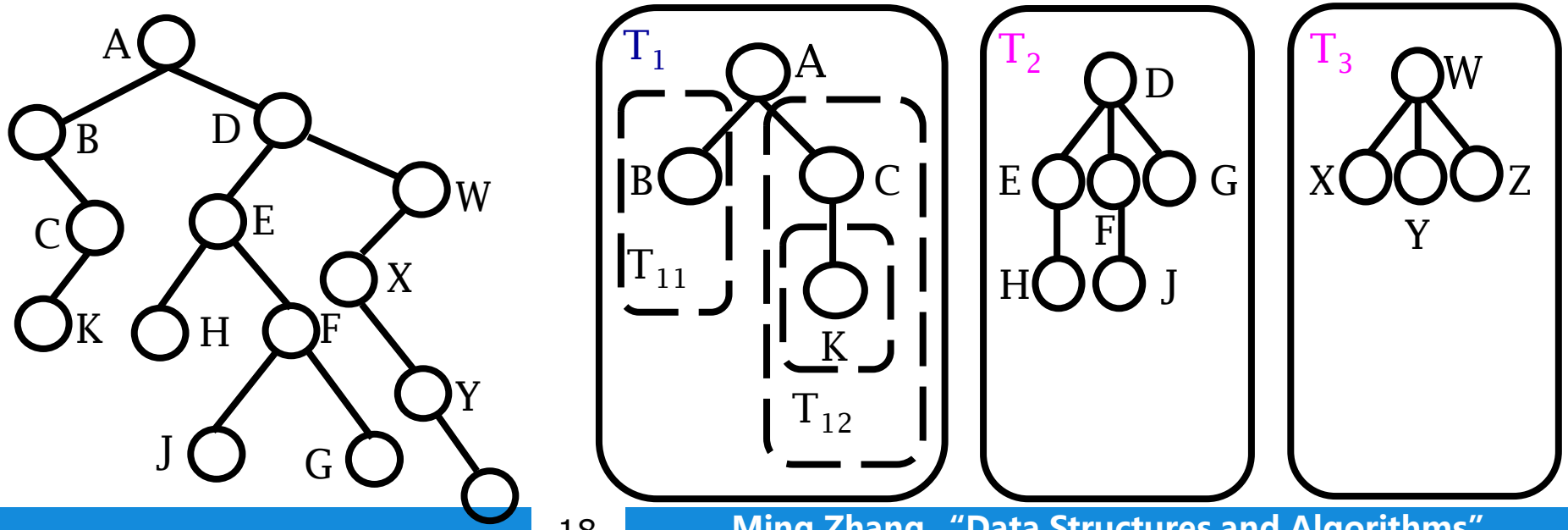




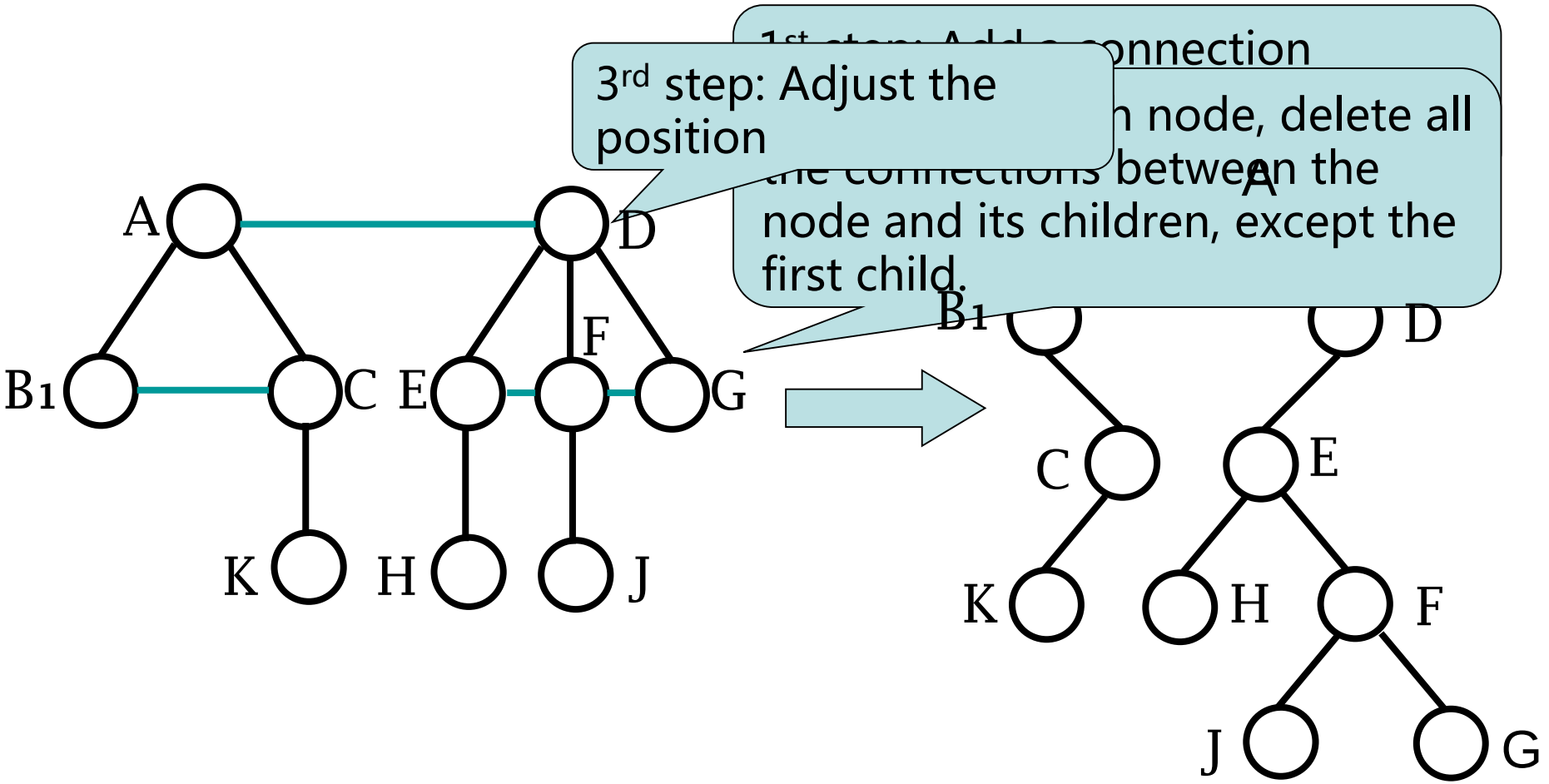


The transformation from a forest to a binary tree

- Ordered set F = {T_1, T_2, ..., T_n} is a forest with trees T_1, T_2, ..., T_n. We transform it to a binary tree B(F) recursively:
  - If F is empty (i.e., n=0), B(F) is an empty binary tree.
  - If F is not empty (i.e., n≠0), the root of B(F) is the root W_1 of the first tree T_1 in F;
  - the left subtree of B(F) is the binary tree B(F_{W1}), where F_{W1} is a forest consisting of W_1's subtrees in T_1;
  - the right subtree of B(F) is the binary tree B(F'), where F' = {T_2, ..., T_n}.

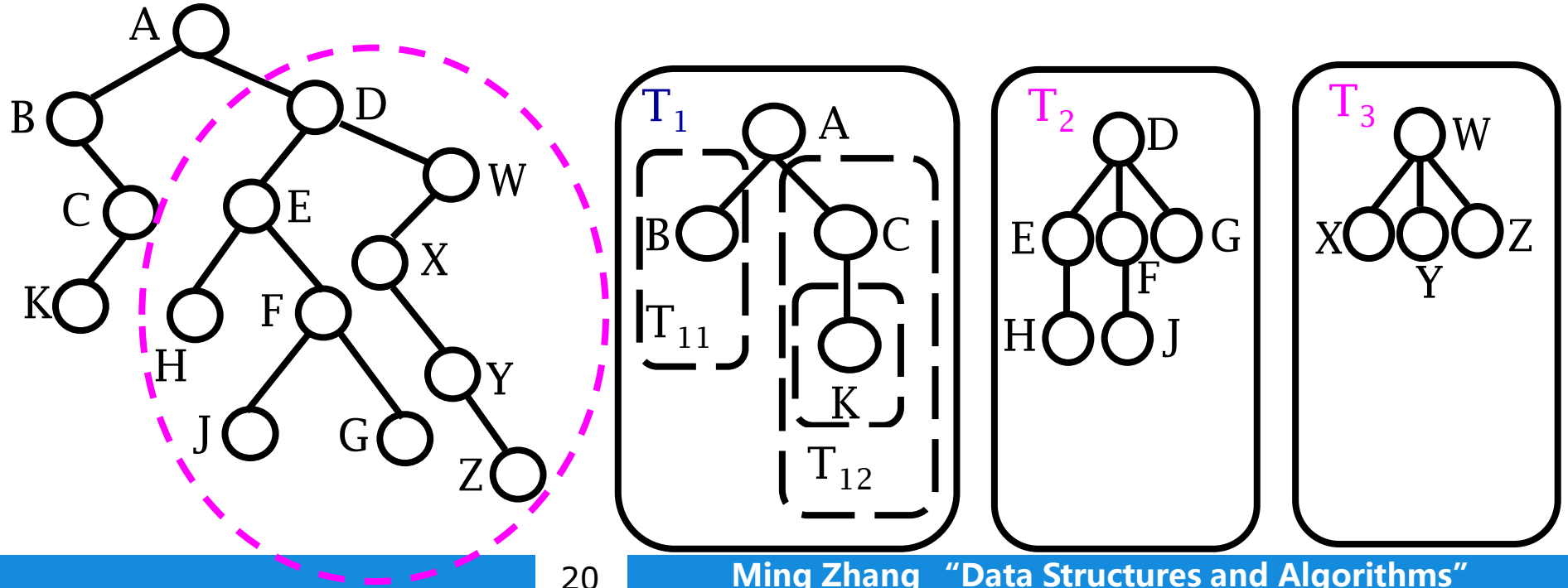


Convert a forest to a binary tree

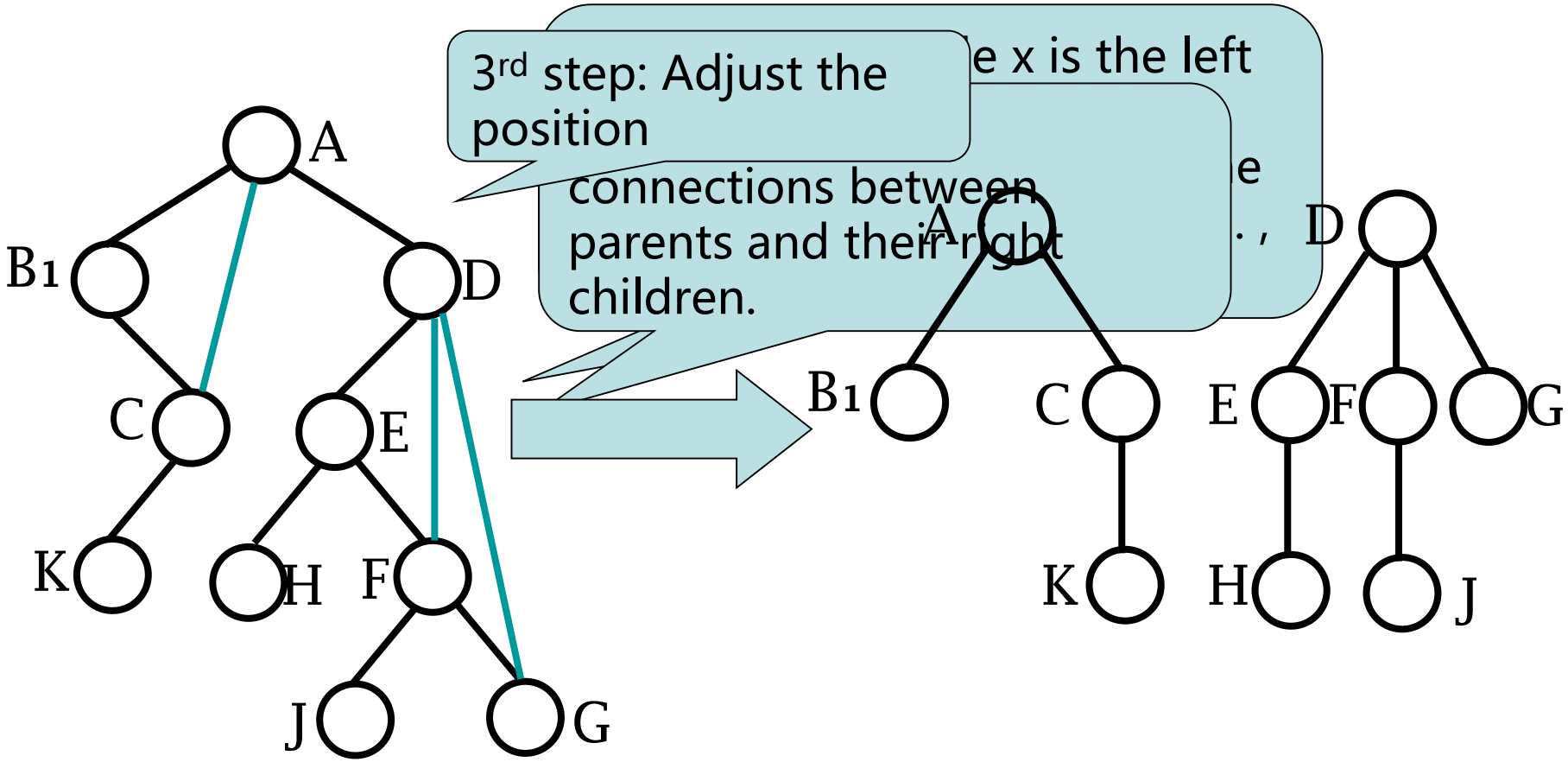


The transformation from a binary tree to a forest

- Assume B is a binary tree, r is the root of B, B_L is the left sub-tree of r, B_R is the right sub-tree of r. We can transform B to a corresponding forest F(B) as follows,
  - If B is empty, F(B) is an empty forest.
  - If B is not empty, F(B) consists of trees $\{T_1\} \cup F(B_R)$, where the root of T_1 is r, the subtrees of r are $F(B_L)$



Convert a binary tree to a forest

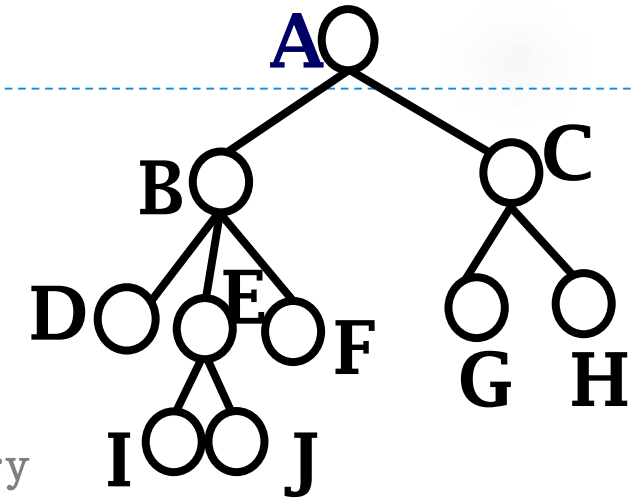


Questions

1. Is a tree also a forest?

1. Why do we establish the one-to-one mapping between binary trees and forests?

Chapter 6 Trees



- General Definitions and Terminology of Tree
  - Trees and Forest
  - Equivalence Transformation between a Forest and a Binary Tree
  - Abstract Data Type of Tree
  - General Tree Traversals
- Linked Storage Structure of Tree
- Sequential Storage Structure of Tree
- K-ary Trees

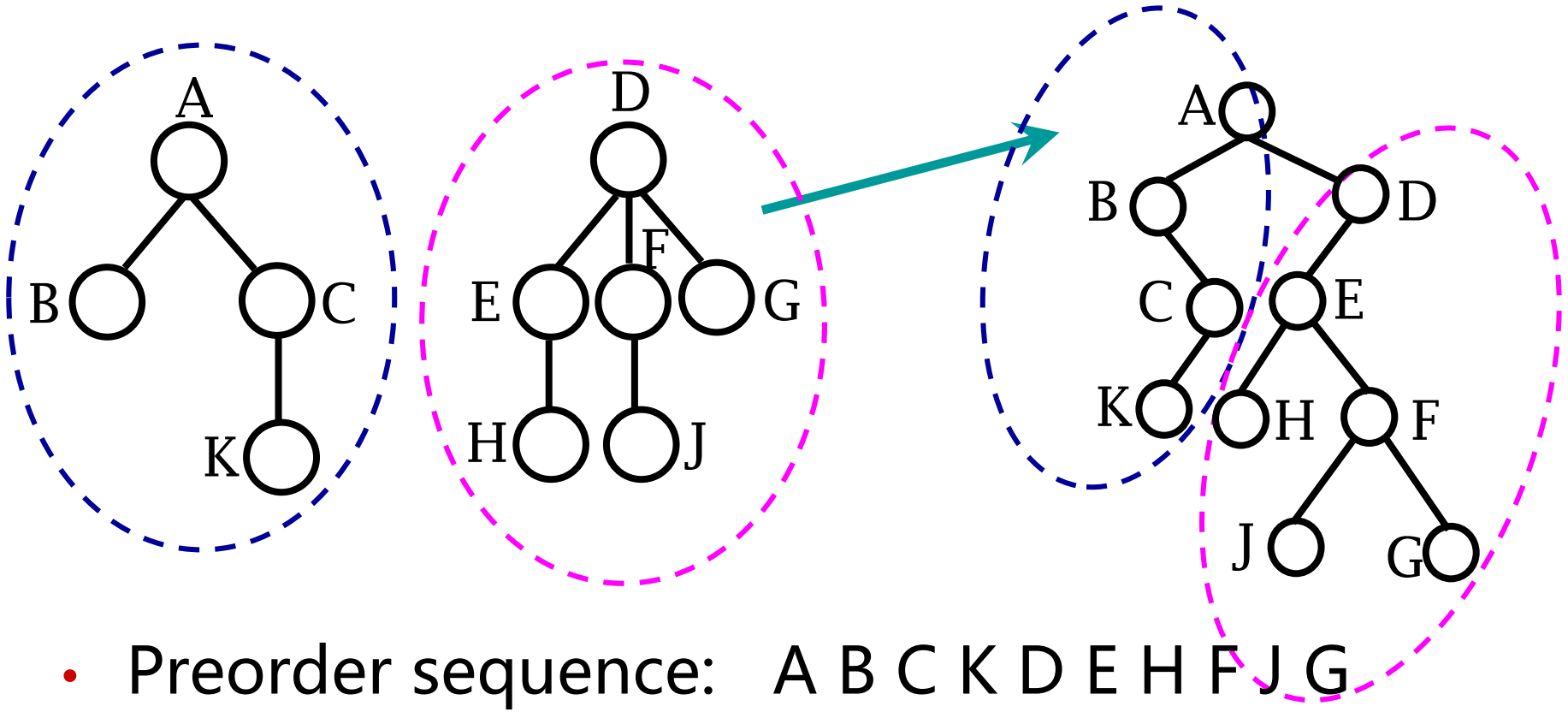
Abstract Data Type of Tree

```
template<class T>
class TreeNode  {                                  // The ADT of the tree node
public:
  TreeNode(const T& value);                        // Constructor
  virtual ~TreeNode() {};                          // Destructor
  bool isLeaf();                                   // Check whether the current node is the
                                                   //    leaf node or not
  T Value();                                       // Return the value of the node
  TreeNode<T> *LeftMostChild();                    // Return the left-most (first) child
  TreeNode<T> *RightSibling();                     // Return the right sibling
  void setValue(const T& value);                   // Set the value of the current node
  void setChild(TreeNode<T> *pointer);             // Set the left child
  void setSibling(TreeNode<T> *pointer);           // Set the right sibling
  void InsertFirst(TreeNode<T> *node);             // Insert a node as the left child
  void InsertNext(TreeNode<T> *node);              // Insert a node as the right sibling
};
```

Abstract Data Type of Tree

```
template<class T>
class Tree  {
public:
    Tree();                                          // Constructor
    virtual ~Tree();                                 // Destructor
    TreeNode<T>* getRoot();                          // Return the root node
    void CreateRoot(const T& rootValue);             // Create a root node whose value is rootValue
    bool isEmpty();                                  // Check whether it is an empty tree
    TreeNode<T>* Parent(TreeNode<T> *current);       // Return parent node
    TreeNode<T>* PrevSibling(TreeNode<T> *current);  // Return the previous sibling
    void DeleteSubTree(TreeNode<T> *subroot);        // Delete the subtree rooted at "subroot"
    void RootFirstTraverse(TreeNode<T> *root);       // Depth-first preorder traversal
    void RootLastTraverse(TreeNode<T> *root);        // Depth-first postorder traversal
    void WidthTraverse(TreeNode<T> *root);           // Breath-first traversal
};
```

Traversal of a Forest



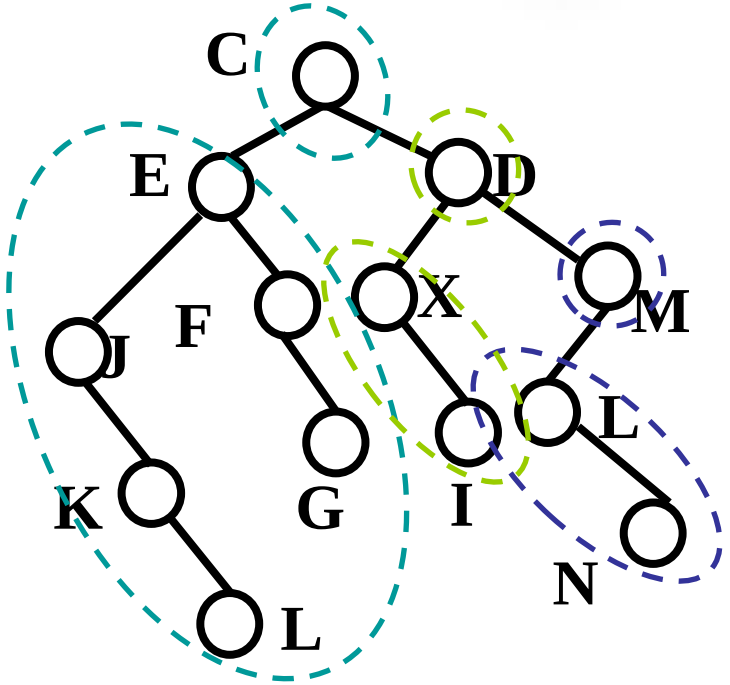
- Preorder sequence: A B C K D E H F J G
- Postorder sequence: B K C A H E J F G D

Forest Traversal vs. Binary Tree Traversal

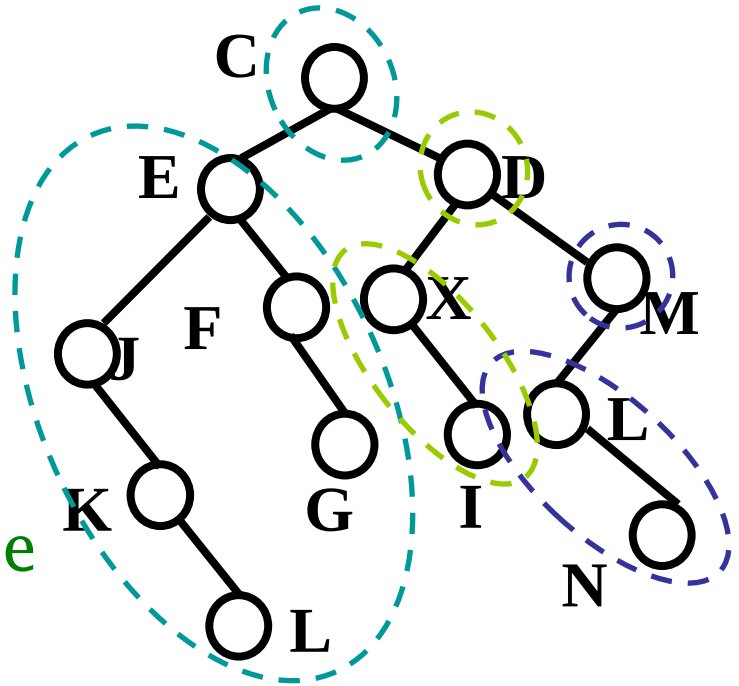
- Preorder forest traversal
  - Preorder binary tree traversal
- Postorder forest traversal
  - Inorder binary tree traversal
- Inorder forest traversal?
  - We cannot define between which two child nodes should the root locate.

Depth-First Preorder Forest Traversal

```
template<class T>
void Tree<T>::RootFirstTraverse(
        TreeNode<T> * root)  {
    while (root != NULL) {
        Visit(root->Value());     // Visit the current node
        // Traverse the subtree forest of the root node of
        // the first tree (except the root node)
        RootFirstTraverse(root->LeftMostChild());
        root = root->RightSibling();     // Traverse other trees
    }
}
```



Depth-First Postorder Traversal



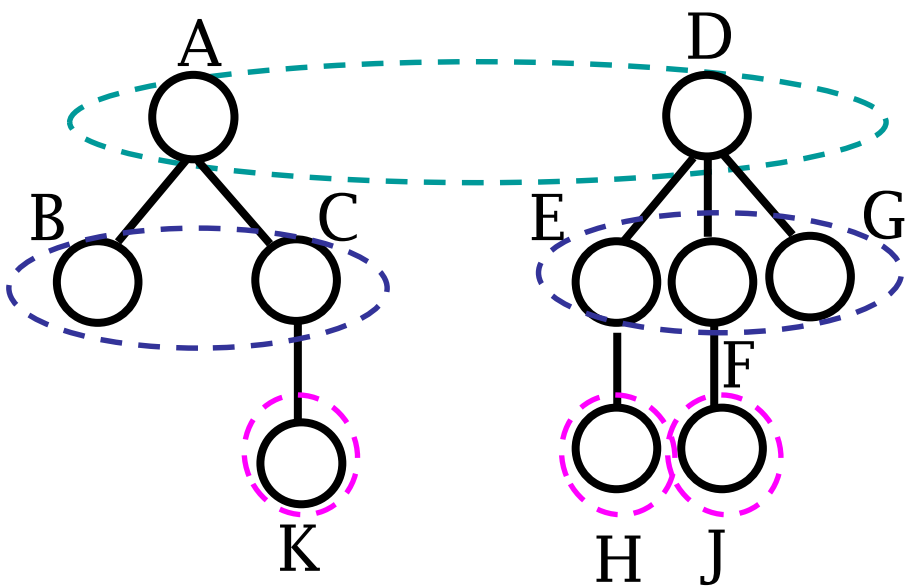

```
template<class T>
void Tree<T>::RootLastTraverse(
        TreeNode<T> * root)  {
    while (root != NULL) {
      // Traverse the subtree forest of the root node
      // of the first tree
      RootLastTraverse(root->LeftMostChild());
      Visit(root->Value());            // Visit the current node
      root = root->RightSibling();  // Traverse other trees
    }
}
```

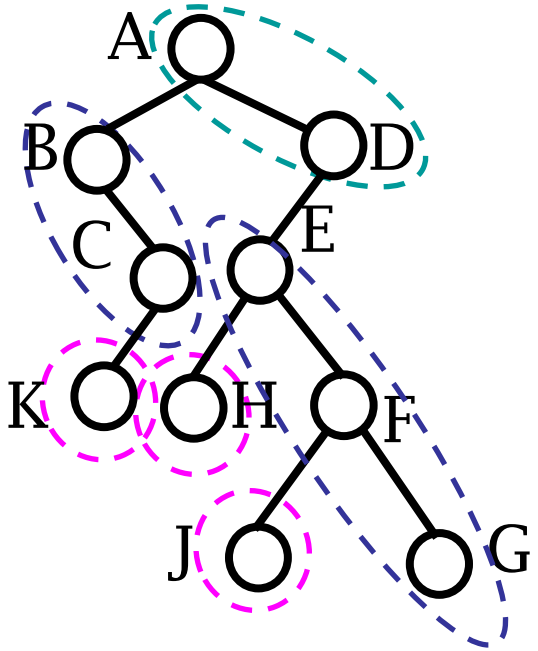
Breadth-First Forest Traversal

- Breadth-first forest traversal
  - Also be called level-order traversal

- a) First, visit the nodes who are at level 0
- b) Second, visit the nodes who are at level 1
- c) Continue until all the nodes at the deepest level are visited

Breadth-First Forest Traversal





- Breadth-first forest traversal sequence: A D B C E F G K H J
- **Look at the right diagonal of the binary tree storage structure**

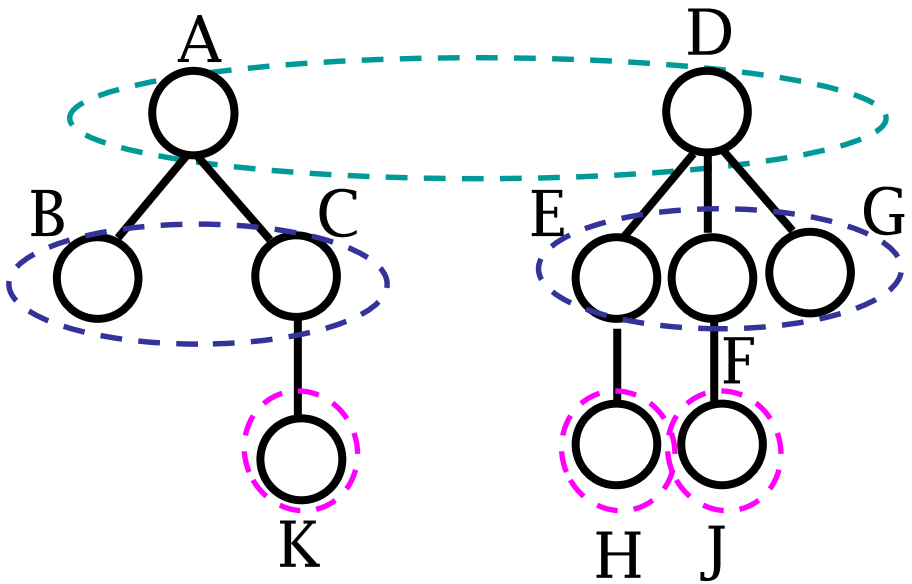
Breadth-first forest traversal

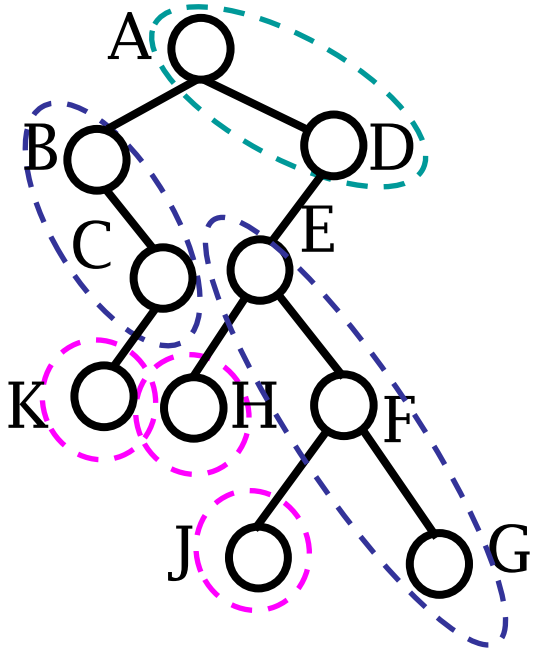
```
template<class T>
void Tree<T>::WidthTraverse(TreeNode<T> * root)  {
   using std::queue;                              // Use STL queue
   queue<TreeNode<T>*> aQueue;
   TreeNode<T> * pointer = root;
   while (pointer != NULL) {
      aQueue.push(pointer);                          // Put the current node go into the queue
      pointer = pointer->RightSibling();     // pointer pointing to right sibling
    }
 while (!aQueue.empty()) {
      pointer = aQueue.front();                 // Get the first element of the queue
      aQueue.pop();                          // Pop the current element out of the queue
      Visit(pointer->Value());                       // Visit the current node
      pointer = pointer-> LeftMostChild(); // pointer pointing to the first child
      while (pointer != NULL) {                 // Put the child nodes of the current node
                                    // into the queue
            aQueue.push(pointer);
            pointer = pointer->RightSibling();
    }  }  }
```

Questions

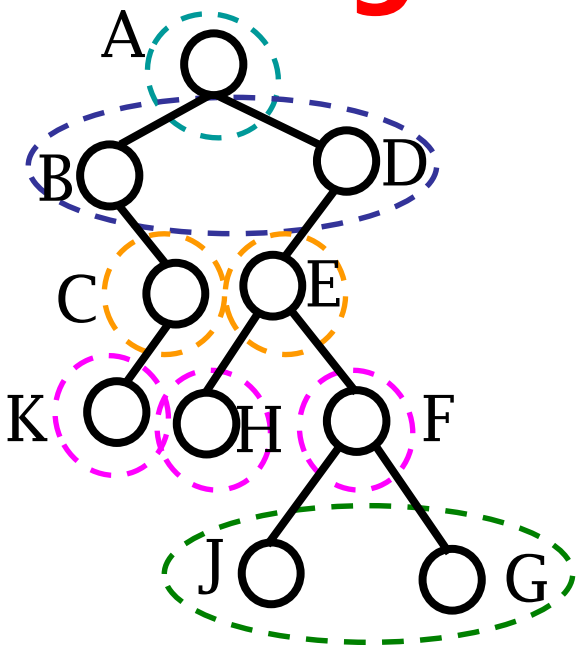
1. Can we use the template of preorder binary tree traversal to accomplish the preorder forest traversal?
2. Can we use the template of inorder binary tree traversal to accomplish the postorder forest traversal?
3. How to accomplish non-recursive depth-first forest search?

different opinions of breadth-first search



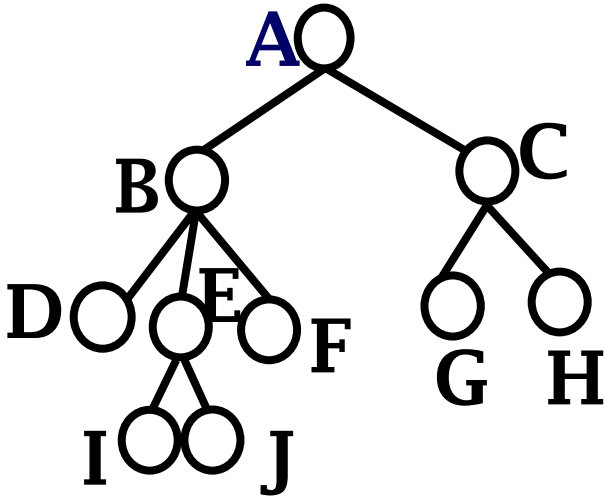


wrong



- Cannot use the template of breadth-first binary tree traversal. For example ,
  - In the left figure, the breadth-first forest traversal: A D B C E F G K H J
    - **Look at the tilt dotted circles in the middle**
  - In the right figure, the breadth-first binary tree traversal: A B D C E K H F J G
    - **Look at the parallel dotted circles on the right**

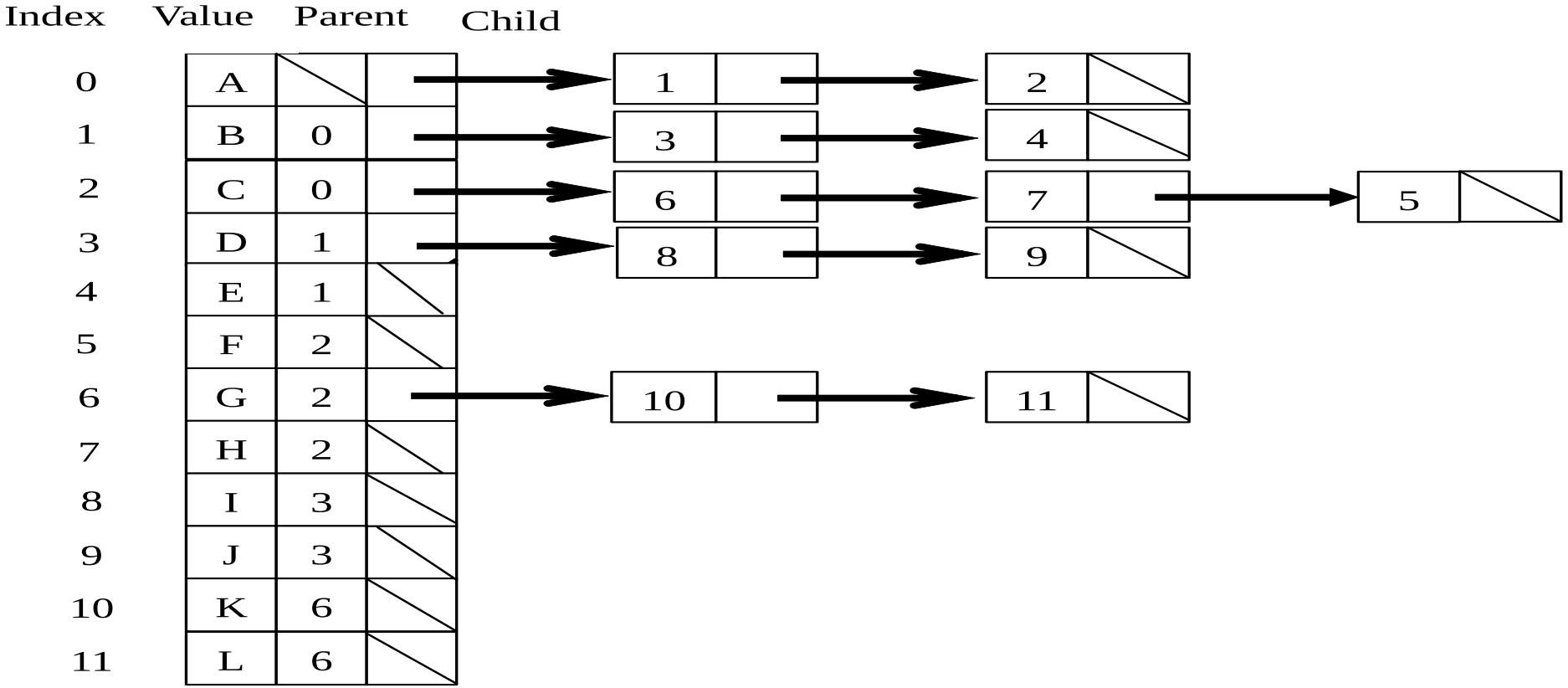
Chapter 6 Trees



- General Definitions and Terminology of Tree
- Linked Storage Structure of Tree
  - List of Children
  - Static Left-Child/Right-Sibling representation
  - Dynamic representation
  - Dynamic Left-Child/Right-Sibling representation
  - Parent Pointer representation and its Application in Union/Find Sets
- Sequential Storage Structure of Tree
- K-ary Trees

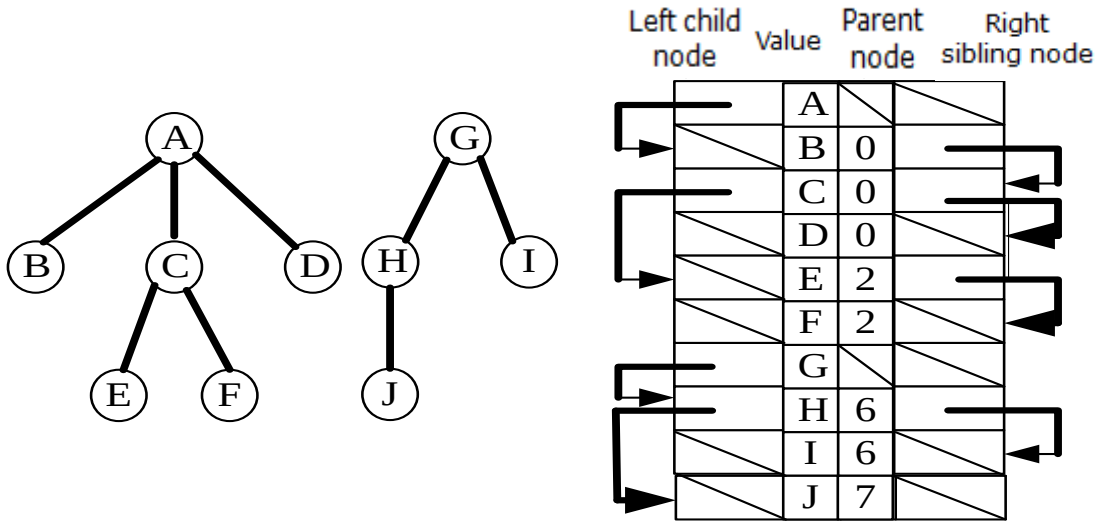
"List of Children" representation

List of children is the adjacency list of a directed graph.



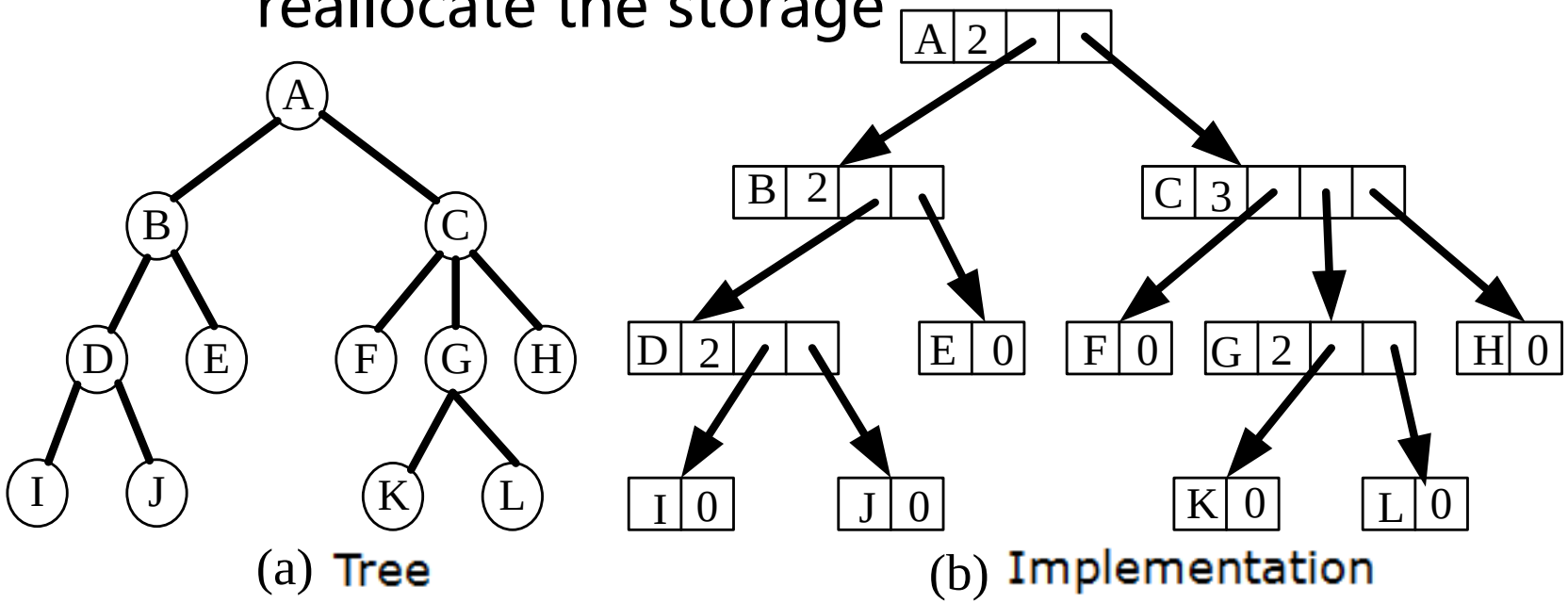
Static Left-Child/Right-Sibling representation

- Child node table stored in an array



Dynamic representation

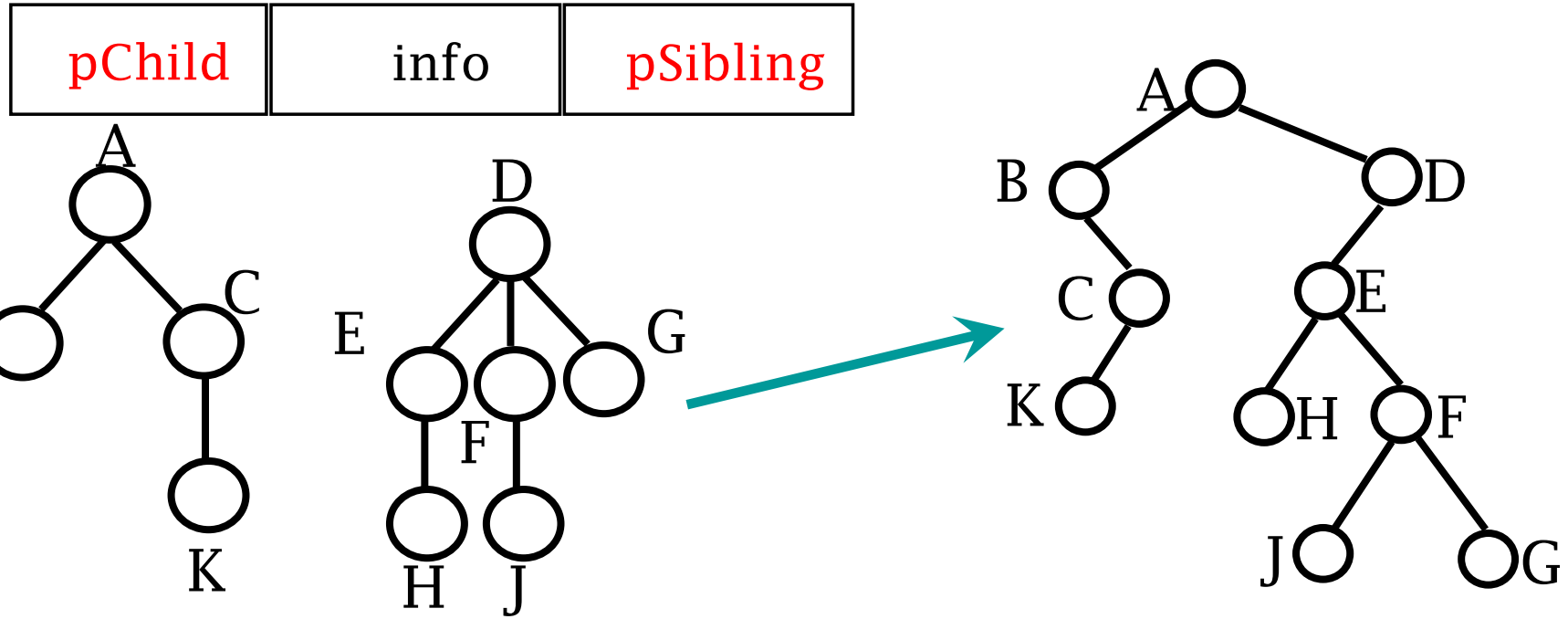
- Allocate dynamic storage to each node
  - If the number of child nodes changes, we should reallocate the storage



(a) Tree (b) Implementation

Dynamic Left-Child/Right-Sibling representation

- The left child in the tree is the first child of the node, the right child is the right sibling of that node.
- The right sibling of the root node is the root node of next tree in the forest



The key details of Dynamic Left-Child/Right-Sibling representation

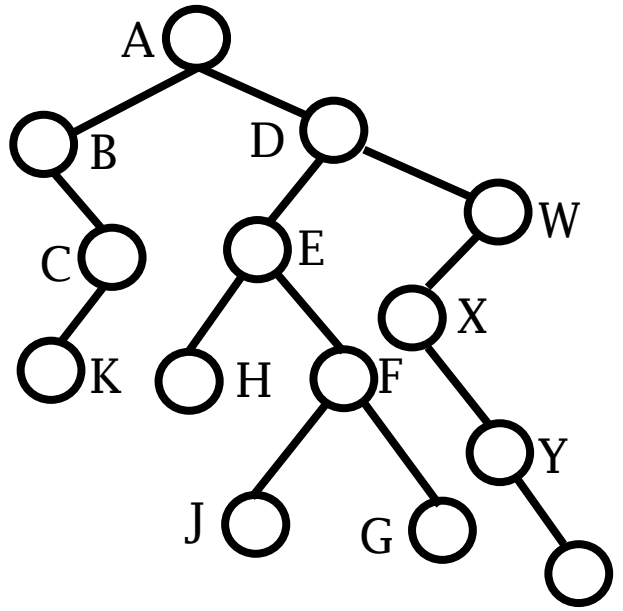
```
// Add following private variables to the class TreeNode
private:
T m_Value;                    // the value of the node
TreeNode<T> *pChild;          // the pointer of the first left child
TreeNode<T> *pSibling;        // the pointer of the right sibling
```

Find the parent node of the current node

```
template<class T>
TreeNode<T>* Tree<T>::Parent(TreeNode<T> *current)  {
using std::queue;                                          // use the queue of STL
    queue<TreeNode<T>*> aQueue;
TreeNode<T> *pointer = root;
TreeNode<T> *father = upperlevelpointer  = NULL;          // record the parent node
if (current != NULL && pointer != current)  {
while (pointer != NULL) {                    // Put all root nodes in the forest into the queue
      if (current == pointer)                // the parent node of the root node is empty
break;
      aQueue.push(pointer);                  // Put the current node into the queue
pointer=pointer-> RightSibling();            // let the pointer point to the right sibling node
}
```

Find the parent node of the current node

```
    while (!aQueue.empty()) {
        pointer = aQueue.front();            // get the pointer of the first node in the queue
        aQueue.pop();                        // Pop the current node out of the queue
        upperlevelpointer  = pointer;        // pointing to the node at upper level
        pointer = pointer-> LeftMostChild(); // pointing to the first child
        while (pointer) {
        // Put the child node of the current node into the queue
            if (current == pointer) {
               father = upperlevelpointer;  // return the parent node
               break;}
            else {
              aQueue.push(pointer);
              pointer = pointer->RightSibling();}
        }
    }
}
aQueue. clear( );                            // clear the queue, optional(local variable)
return father;
}
```



Delete all the nodes in a sub-forest

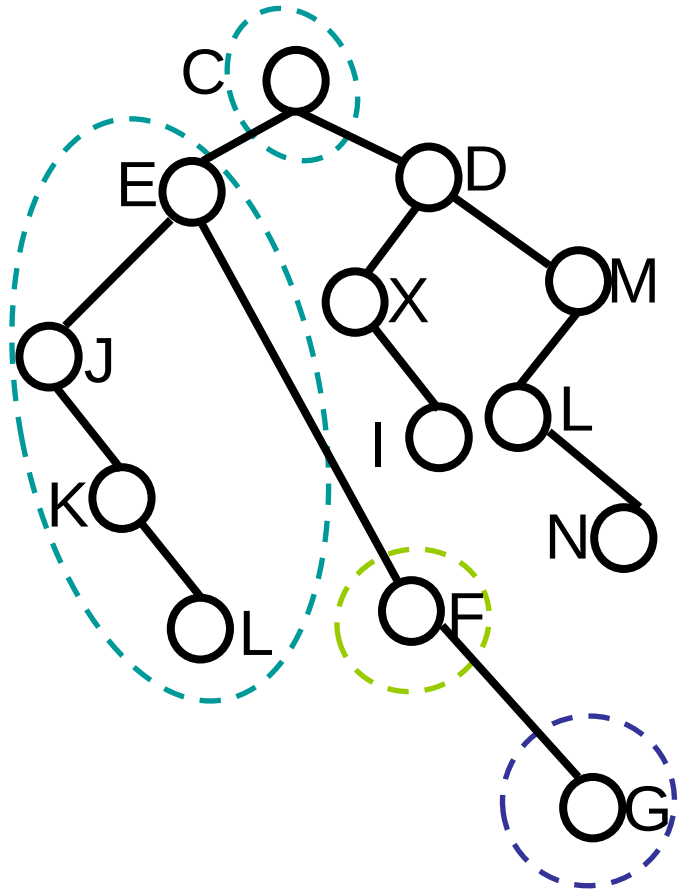
```
template <class T>
void Tree<T>::DestroyNodes(TreeNode<T>* root)  {
    if (root) {
        DestroyNodes(root->LeftMostChild());//delete the first subtree
        DestroyNodes(root->RightSibling());   //delete other subtrees recursively
        delete root; // delete the root node
    }
}
```

Delete the subtree whose root node is subroot

```
template<class T>
void Tree<T>::DeleteSubTree(TreeNode<T> *subroot)  {
    if (subroot == NULL) return;// if the subtree to be deleted is empty, then return
    TreeNode<T> *pointer = Parent (subroot); // find the parent node of subroot
    if (pointer == NULL) {// if subroot does not have a parent node, it is a root node
      pointer = root;
      while (pointer->RightSibling() != subroot)// find in the right siblings of subroot
        pointer = pointer->RightSibling();
      pointer->setSibling(subroot->RightSibling()); // renew the right sibling of pointer
    }
    else if (pointer->LeftMostChild() == subroot) // if subroot is the first child
      pointer->setChild(subroot->RightSibling()); // renew the right sibling of pointer
    else {// the condition where subroot has a left sibling
      pointer = pointer->LeftMostChild();   // sift down to the most left sibling
      while (pointer->RightSibling() != subroot))// find in the right siblings of subroot
        pointer = pointer->RightSibling();
      pointer->setSibling(subroot->RightSibling()); // renew the right sibling of pointer
    }
    subroot->setSibling(NULL);// very important. it will go wrong without this statement
    DestroyNodes(subroot);  // delete all the nodes of in the subforest rooted at subroot
}
```

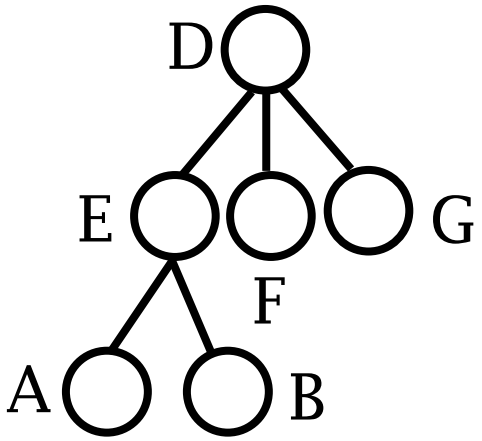
Thinking: Can the following algorithm traverse the forest?

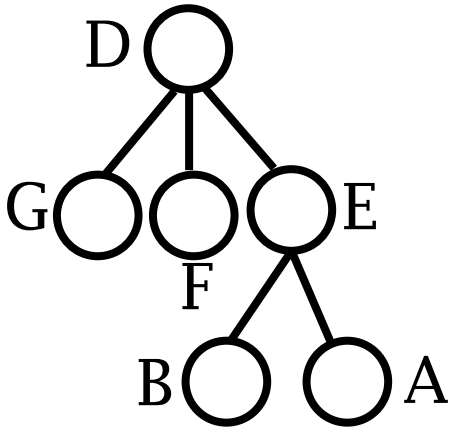
```
template <class T>
void Traverse(TreeNode <T> * rt) {
    if (rt==NULL)  return;
    Visit(rt);
    TreeNode * temp = rt-> LeftMostChild();
    while (temp != NULL) {
        Traverse(temp);
        temp = temp->RightSibling();
    }
}
```



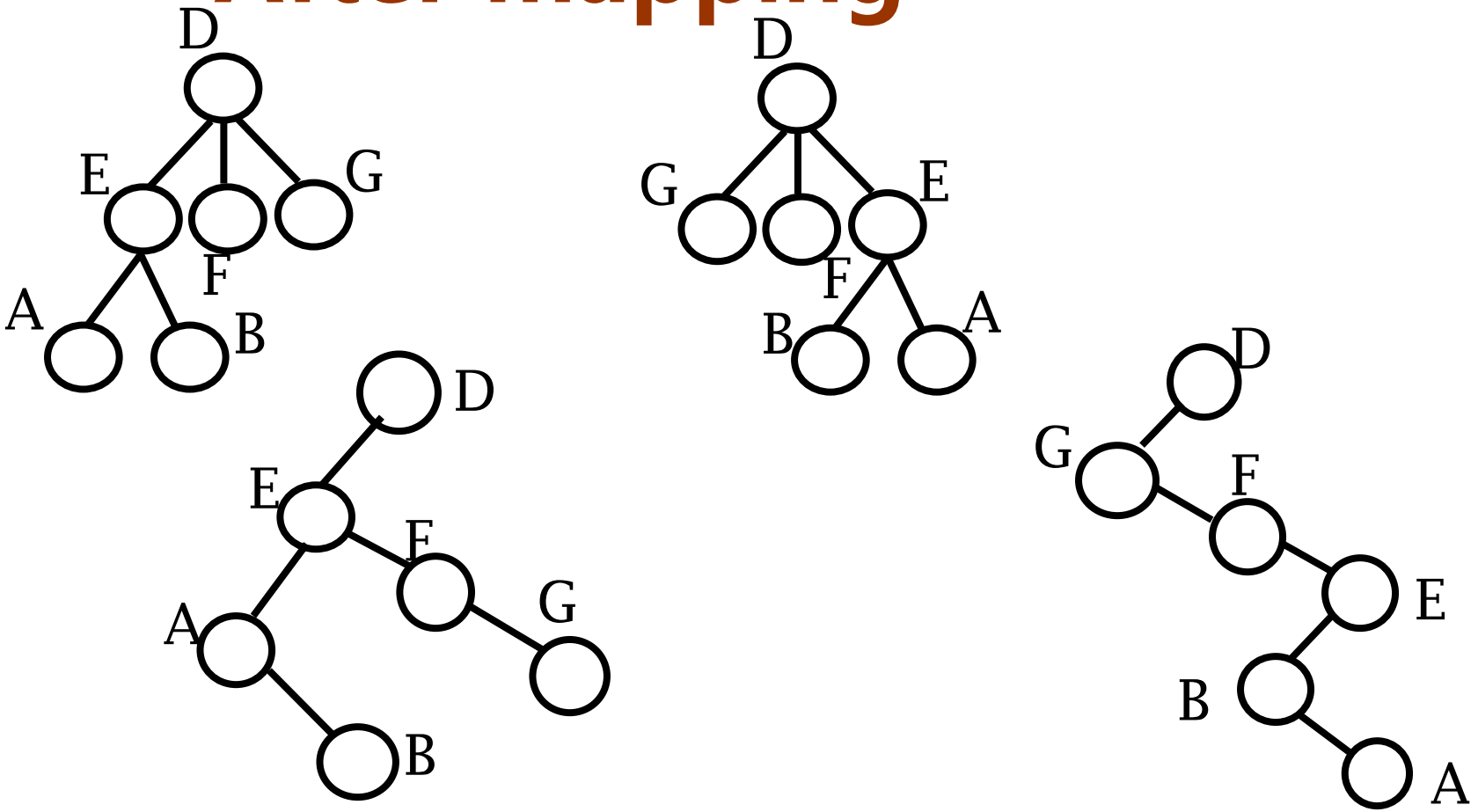
Thinking: use the traversal template flexibly

Example: Specular mapping of a forest





After mapping

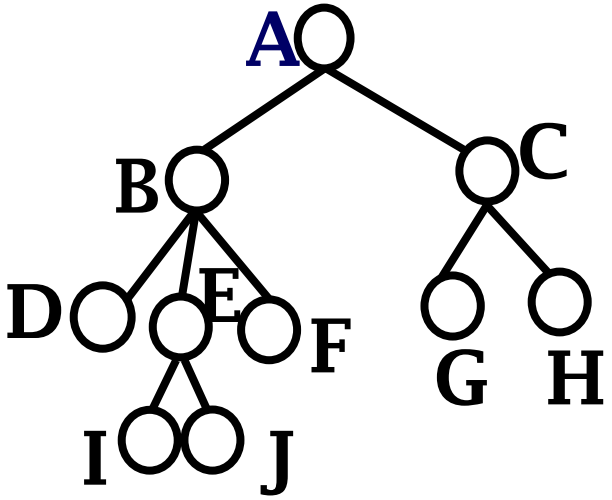


Thinking: delete the subtree rooted at "subroot"

Pay attention to checking whether the subtree to be deleted is empty or not, and whether the subroot have a parent pointer.

Pay attention to the order of pointer updates after deletion.

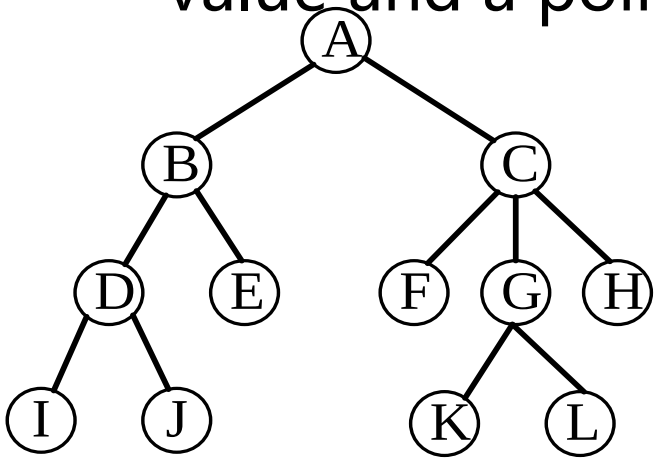
Chapter 6 Trees



- General Definitions and Terminology of Tree
- Linked Storage Structure of Tree
  - List of Children
  - Static Left-Child/Right-Sibling representation
  - Dynamic representation
  - Dynamic Left-Child/Right-Sibling representation
  - Parent Pointer representation and its Application in Union/Find Sets
- Sequential Storage Structure of Tree
- K-ary Trees

Parent Pointer representation

- A representation that you only need to know the parent node
- For each node, you only need to store a pointer which points to its parent node, so that we call it parent pointer representation
- Use an array to store the tree nodes, and each node includes a value and a pointer which points to its parent node



Node index	0	1	2	3	4	5	6	7	8	9	10	11
Value	A	B	C	D	E	F	G	H	I	J	K	L
Parent node index		0	0	1	1	2	2	2	3	3	6	6

Parent Pointer representation: Algorithm

- Find the root node of the current node
  - Start from a node, find a path from that node to its root node
    - O(k), k is the height of the tree
- Check whether two nodes are in the same tree
  - If these two nodes have the same root node, then they are sure to be in the same tree.
  - If these two nodes have different root nodes, then they are sure to be in different trees.

Union/Find Sets

- **Union/Find Sets** is a special kind of sets, consisted of some disjoint subsets. The basic operations of Union/Find Sets are:
  - Find: Find the set the node belongs to
  - Union: Merge two sets
- Union/Find Sets is an important abstract data types
  - The application of Union/Find sets is mainly to solve the problem of equivalence classes .

Equivalence relation

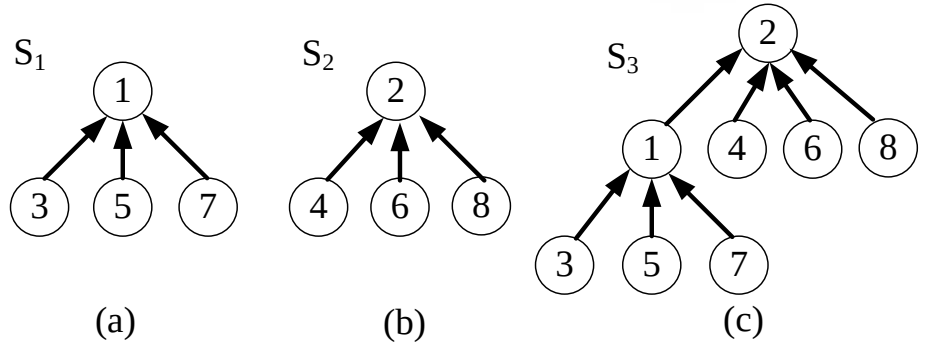
- There is a set S, having n elements, and a set R, having r relations, which are defined based on S. And x, y, z, are the elements of the set S.
- The relation R will be an equivalence relation, if and only if the following conditions are true:
  a) $(x, x) \in R$ for all (**reflexivity**)
  b) $(y, x) \in R$ if and only if $(x, y) \in R$ (**symmetry**)
  c) If $(x, y) \in R$ and $(y, z) \in R$, then $(x, z) \in R$ (**transitivity**)
- If $(x, y) \in R$, we say elements x and y are equivalent

Equivalence Classes

- Equivalence class is the largest set consisting of elements which are equivalent to each other. The 'largest' means that there is no other element equivalent to any element in the set.

- An equivalence class derived from $x \in S$ based on the relation R
  - $[x]_R = \{y \mid y \in S \wedge xRy\}$
  - R partition S into r disjoint sets $S_1, S_2, \ldots S_r$, and the union of these sets equals to S

Use tree to represent the Union/Find of equivalence classes

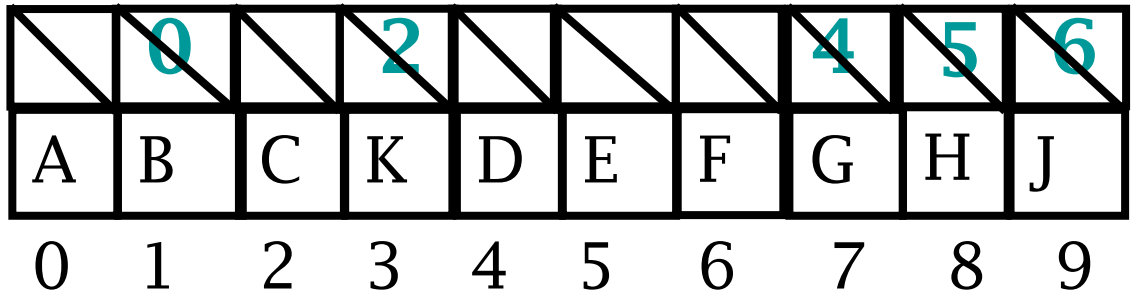


- Use a tree to represent a set
  - The set can be represented by the root node
  - If two nodes are in the same tree, they belong to the same set
- The representation of tree
  - Store in a static pointer array
  - A node only needs to store the information of its parent node

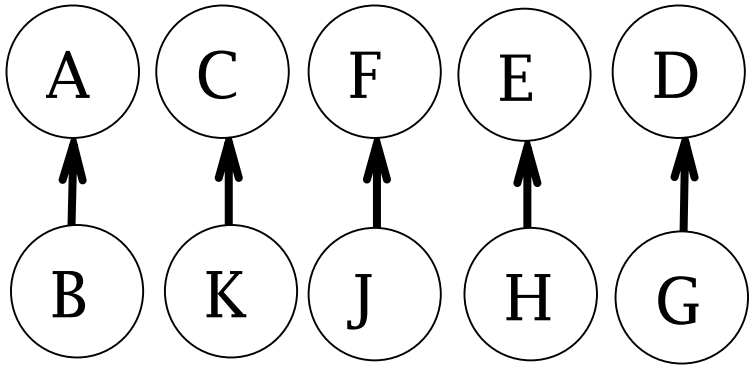
UNION/FIND Algorithm(1)

Process these 5 equivalence pairs (A, B),

(C, K) , (J, F), (H, E), (D, G)



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

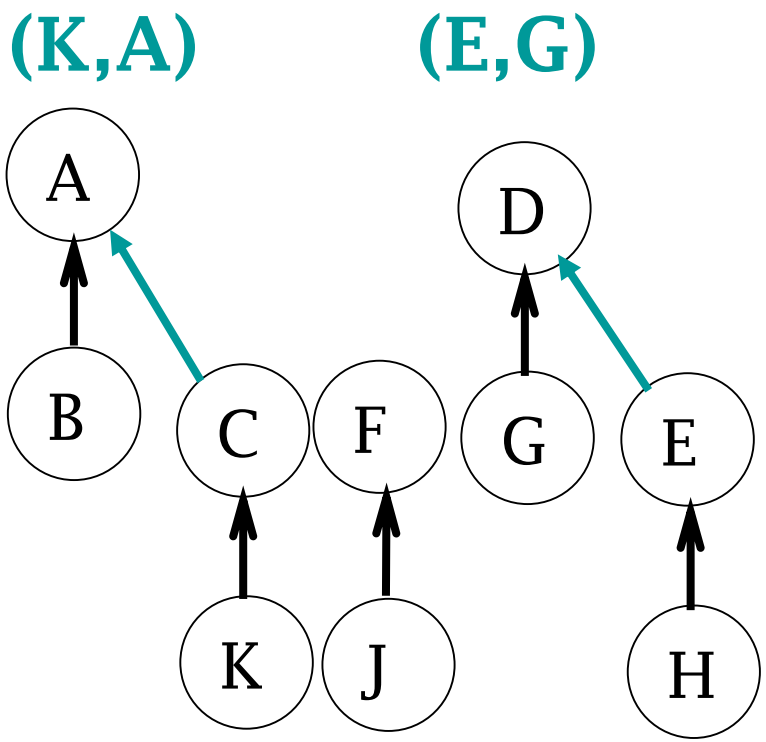


UNION/FIND Algorithm(1)

Then, process two equivalence pairs (K, A) and (E, G)

The root of the tree that K belongs to is C, A itself is a root node, and A ! =C, so merge these two trees.

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



6.2 Linked Storage Structure of Tree

Parent pointer representation and Union/Find algorithm representation

```
template<class T>
class ParTreeNode  {                                  //Definition of tree node
private:
  Tvalue;                                             //The value of the node
  ParTreeNode<T>* parent;                             //The parent node pointer
  int nCount;                                         //The number of the nodes in the set
public:
  ParTreeNode();                                      //Constructor
  virtual ~ParTreeNode(){};                           //Destructor
  TgetValue();                                        //Return the value of the node
  void  setValue(const T& val);                       //Set the value of the node
  ParTreeNode<T>*  getParent();                       //Return the parent node pointer
  void  setParent(ParTreeNode<T>* par);               //Set the parent node pointer
  int  getCount();                                    //Return the number of nodes
  void  setCount(const int count);                    //Set the number of nodes
};
```

6.2 Linked Storage Structure of Tree

Parent pointer representation and Union/Find algorithm representation

```
template<class T>
class ParTree  {                                  // Definition of the tree
public:
  ParTreeNode<T>* array;                          // the array used to store the tree node
  int Size;                                       // the size of the array
  ParTreeNode<T>*
  Find(ParTreeNode<T>* node) const;               // Find the root node of "node"
  ParTree(const int size);                        // Constructor
  virtual ~ParTree();                             // Destructor
  void Union(int i,int j);                        // Union set i and j, and merge them
                                                  //    into the same subtree
  bool Different(int i,int j);   // Check if node i and j belong to the same tree
};
```

Parent pointer representation and Union/Find algorithm representation

```
template <class T>
ParTreeNode<T>*
ParTree<T>::Find(ParTreeNode<T>* node) const
{
  ParTreeNode<T>* pointer=node;
  while ( pointer->getParent() != NULL )
    pointer=pointer->getParent();
  return pointer;
}
```

Parent pointer representation and Union/Find algorithm representation

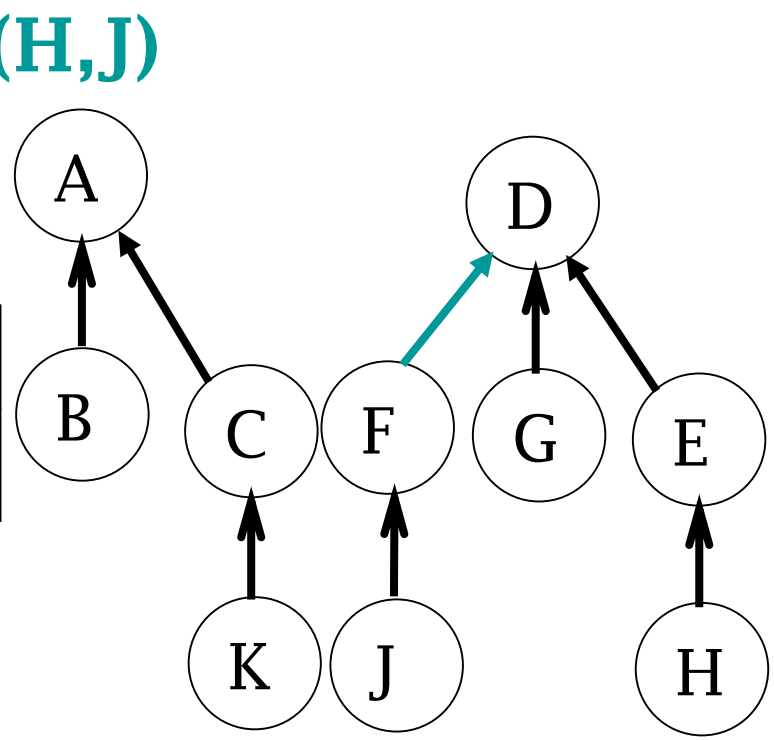
```
template<class T>
void ParTree<T>::Union(int i,int j) {
ParTreeNode<T>* pointeri = Find(&array[i]);      // find the root node of node i
ParTreeNode<T>* pointerj = Find(&array[j]);      // find the root node of node j
if (pointeri != pointerj) {
  if(pointeri->getCount() >= pointerj->getCount()) {
        pointerj->setParent(pointeri);
        pointeri->setCount(pointeri->getCount() +
                           pointerj->getCount());
  }
  else {
        pointeri->setParent(pointerj);
        pointerj->setCount(pointeri->getCount() +
                           pointerj->getCount());
   }
} }
```

UNION/FIND Algorithm(2)

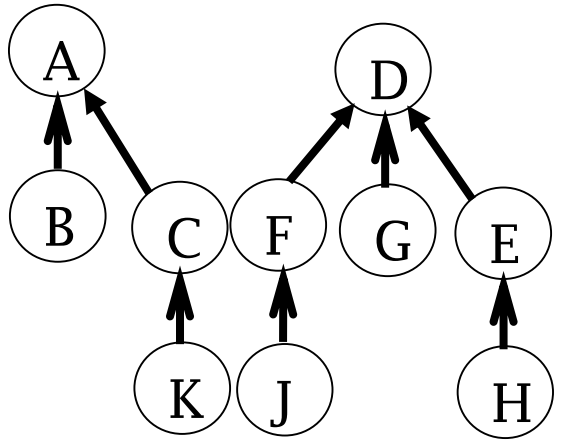
Use weighted union rule to process (H, J)

According to weighted union rule, the number of nodes of the tree whose root node is F is smaller, so let F point to D

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



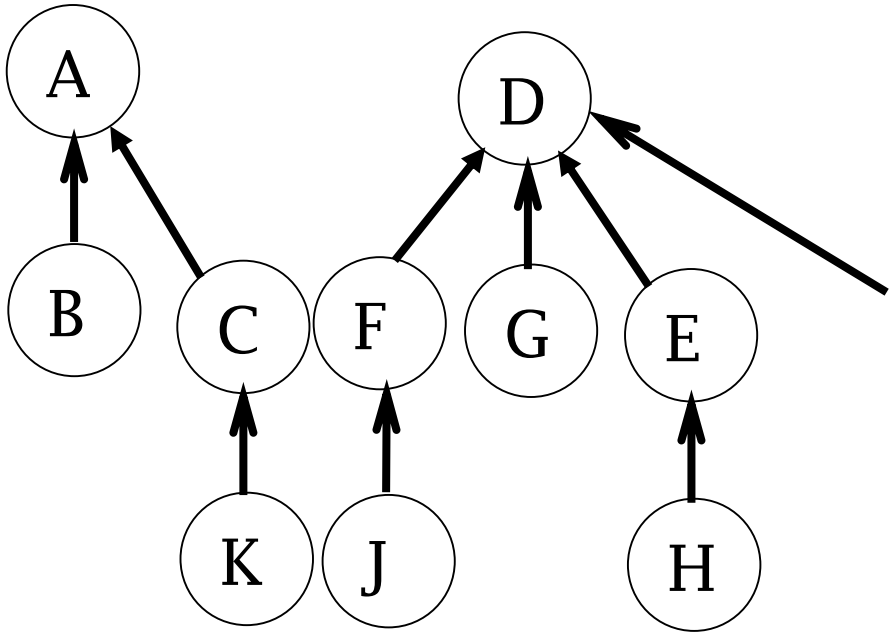
Path compression



- Find X
  - Assume that X finally reaches the root node R
  - Along the path from X to R, make parent pointer of every node point to R
- Low altitude trees come out

UNION/FIND Algorithm(3)

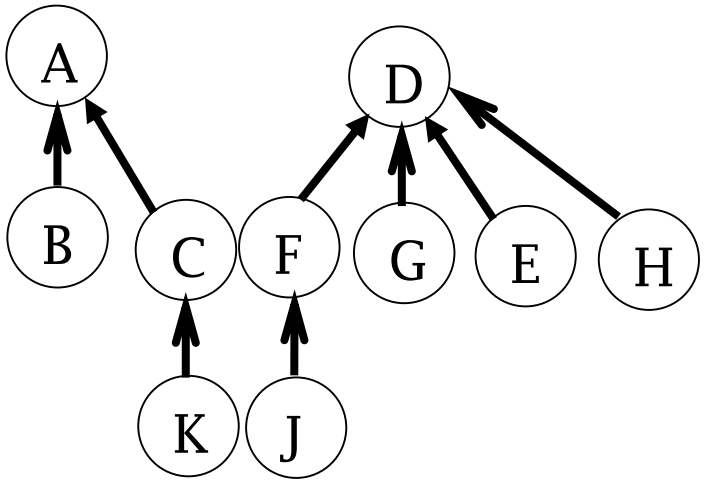
Use path compression to deal with Find(H)



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

Path compression

```
template <class T>
ParTreeNode<T>*
ParTree<T>::FindPC(ParTreeNode<T>* node) const
{
  if (node->getParent() == NULL)
    return node;
  node->setParent(FindPC(node->getParent()));
  return node->getParent();
}
```



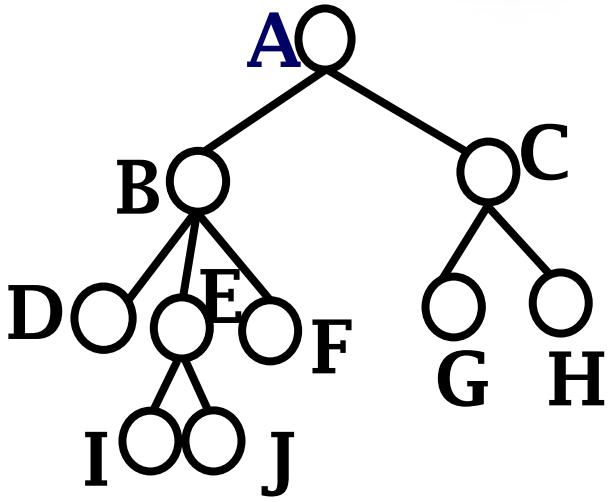
Path compression make the expenditure of Find close to a constant

- Weight + path compression
- The expenditure of n Find operations for n nodes is O(nα(n)), which is about Θ(nlog*n)
  - α(n) is the inverse of univariate Ackermann function, its growth rate is slower than logn, but not equals to constant
  - log*n is the number of operations that calculate log(n) before n = logn ≤ 1
  - log*65536 = 4 (4 log operations)
- The algorithm of find needs at most n Find operations, so it is very close to Θ(n)
  - In practical applications, α(n) is usually smaller than 4

Thinking

- Can we use dynamic pointer representation to accomplish parent pointer representation?
- Consult books or websites, read different optimizations of weighted union rule and path compression. Discuss their differences, advantages and disadvantages.

Chapter 6 Trees



- General Definitions and Terminology of Tree
- Linked Storage Structure of Tree
- Sequential Storage Structure of Tree
- K-ary Trees

Sequential Storage Structure of Tree

- Preorder sequence with right link representation
- Double-tagging preorder sequence representation
- Double-tagging level-order sequence representation
- Postorder sequence with degree representation

Preorder sequence with right link representation

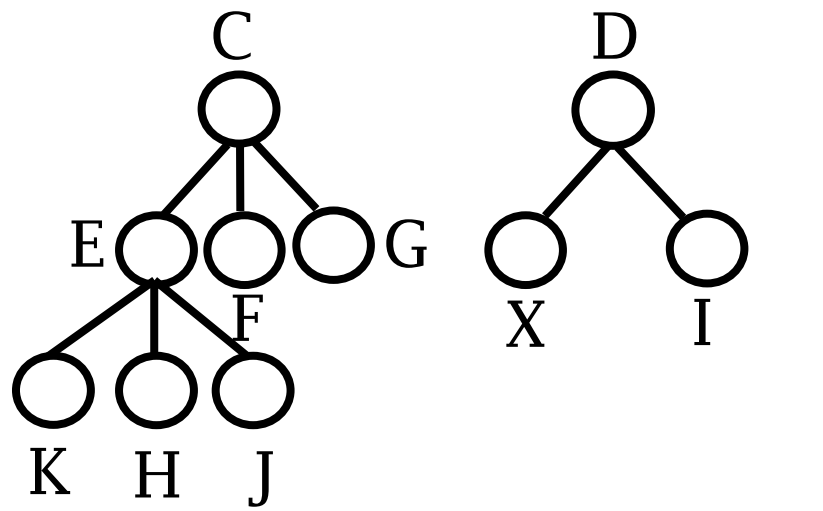
- Nodes are stored continuously according to preorder sequence

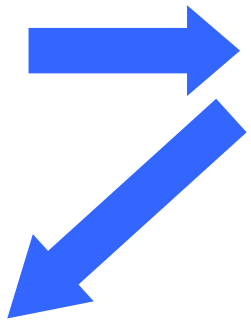
ltag	info	rlink

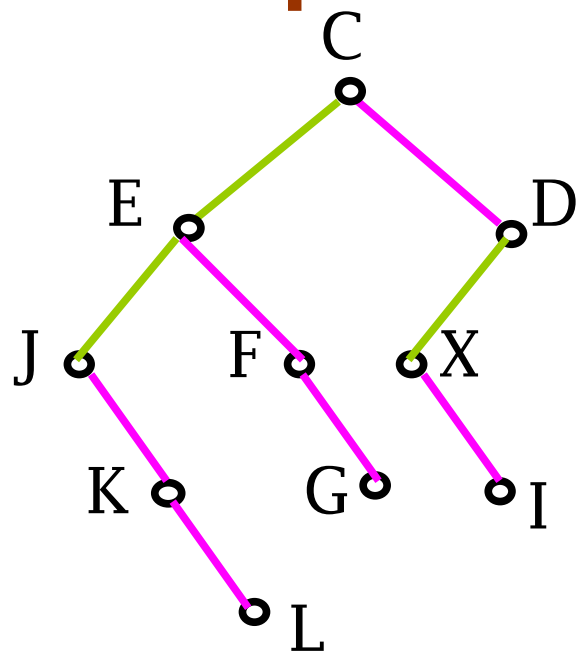
  - info : the data of the node
  - rlink : right link
    - Point to the next sibling of the node, which is corresponding to the right child node of the parent node in the binary tree
  - ltag : tag
    - If the node has no child node, which means the node doesn't have a left child node in the binary tree, and ltag will be 1.
    - Otherwise, ltag will be 0.

6.3 Sequential Storage Structure of Tree

Preorder sequence with right link representation

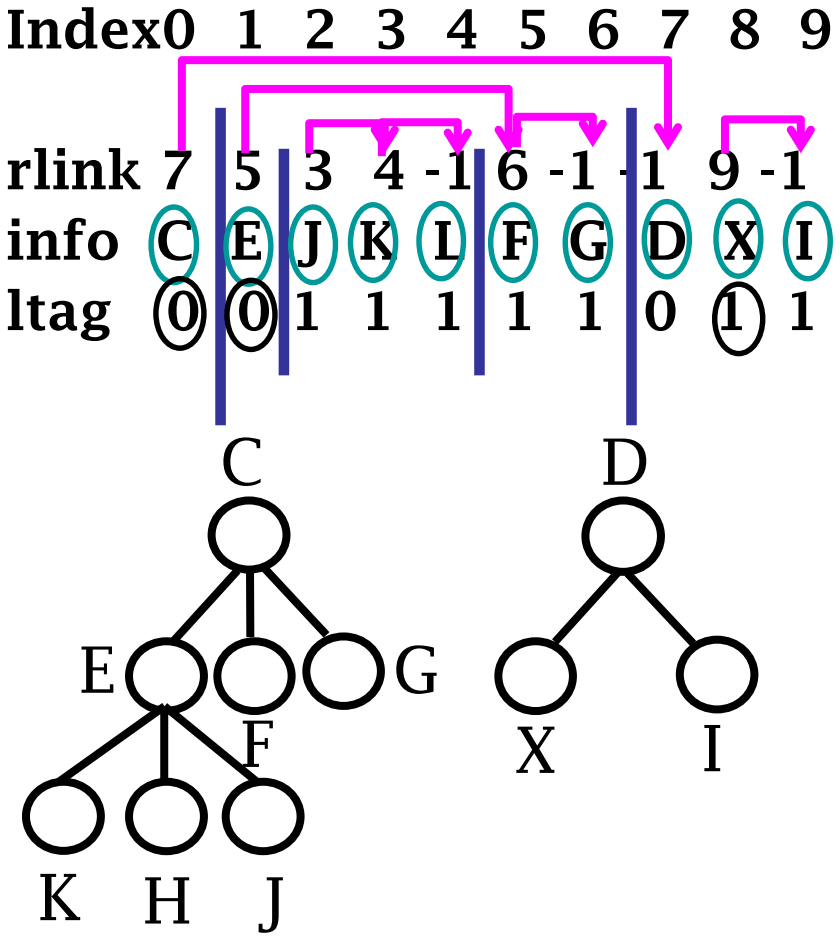


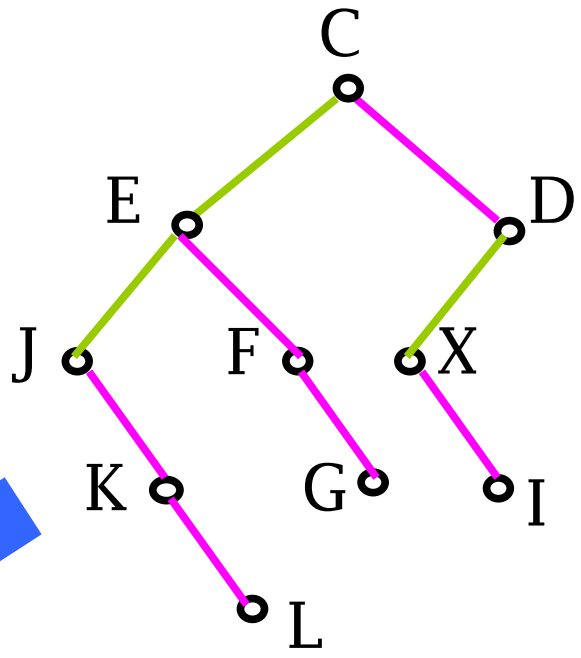




Index	0	1	2	3	4	5	6	7	8	9
rlink	7	5	3	4	-1	6	-1	-1	9	-1
info	C	E	J	K	L	F	G	D	X	I
ltag	0	0	1	1	1	1	1	0	1	1

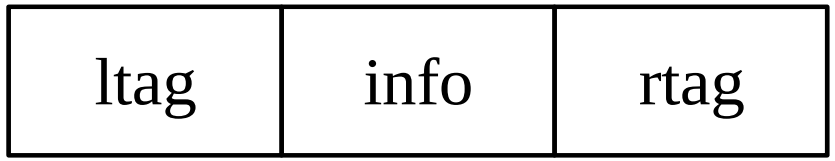
From a preorder rlink-ltag to a tree





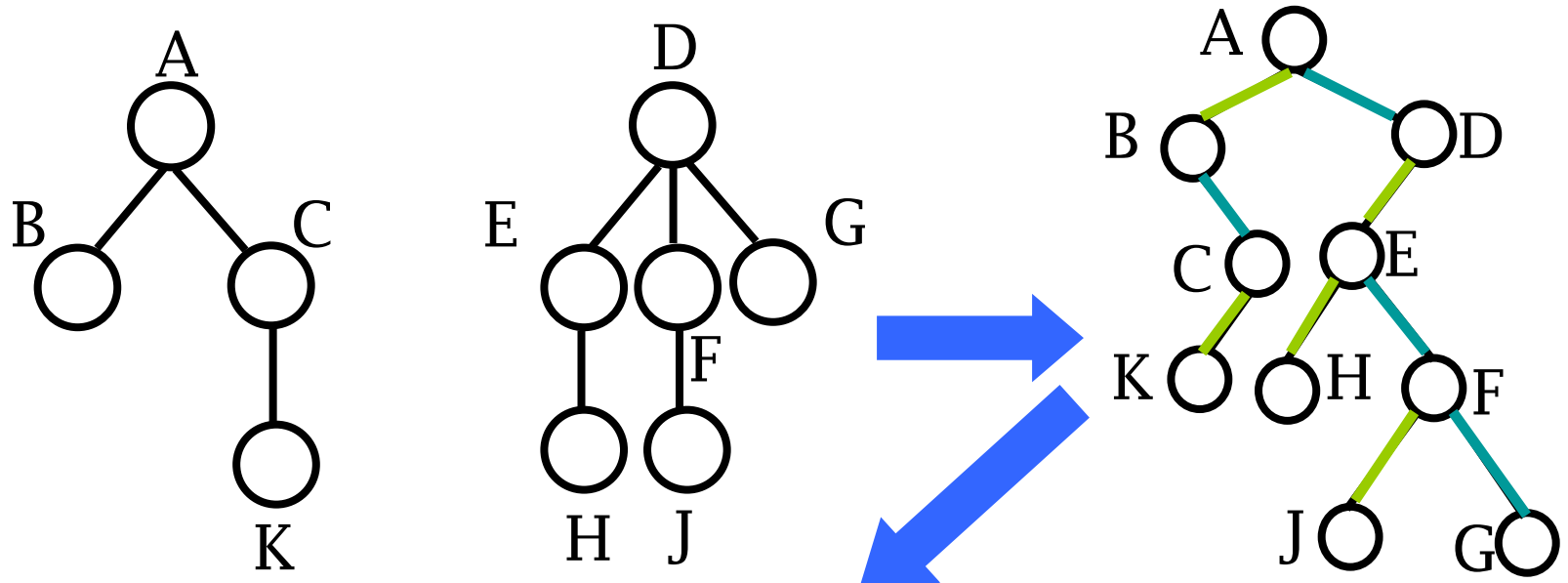
Double-tagging preorder sequence representation

- In preorder sequence with right link representation, rlink is still redundant, so we can replace the pointer rlink with a tag rtag, then it is called "double-tagging preorder sequence representation". Each node includes data and 2 tags(ltag and rtag), the form of the node is like:



According to the preorder sequence and 2 tags(ltag, rtag), we can calculate the value of llink and rlink of each node in the "Left-child/Right-sibling" list. And llink will be the same as that in preorder sequence with right link representation.

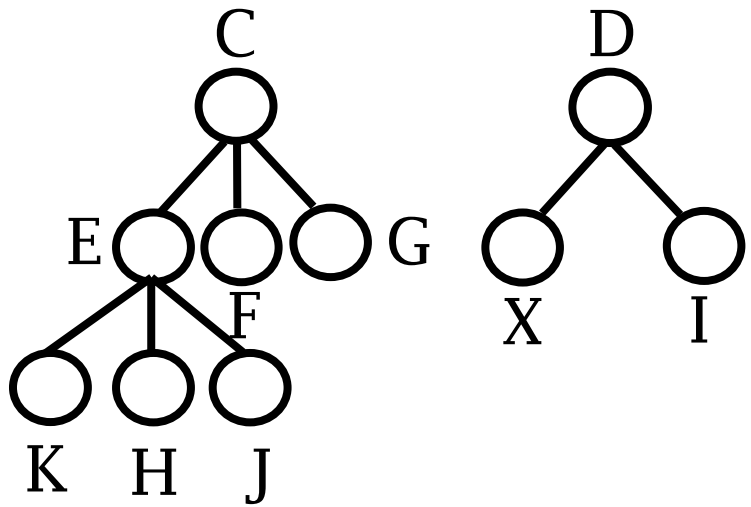
Double-tagging preorder sequence representation

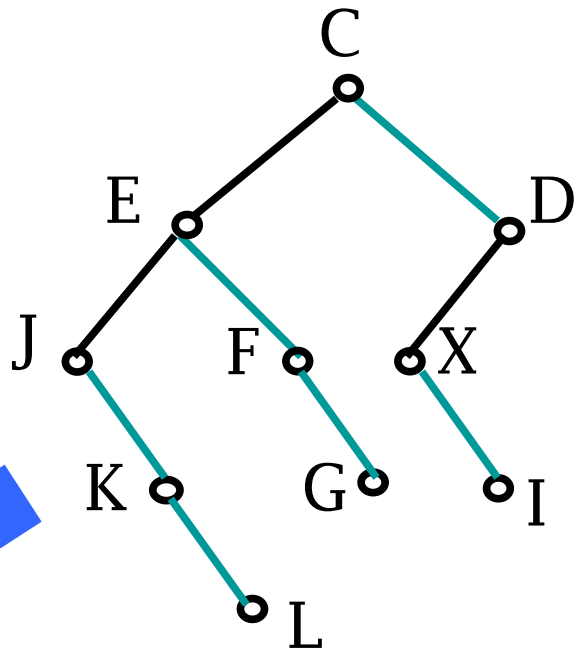


rtag	0	0	1	1	1	0	1	0	1	1
info	A	B	C	K	D	E	H	F	J	G
ltag	0	1	0	1	0	0	1	0	1	1

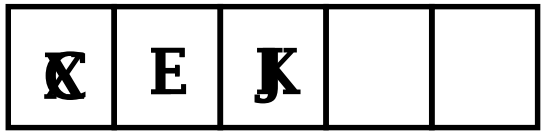
From a rtag-ltag preorder sequence to a tree

Index	0	1	2	3	4	5	6	7	8	9
rtag	0	0	0	0	1	0	1	1	0	1
info	C	E	J	K	L	F	G	D	X	I
ltag	0	0	1	1	1	1	1	0	1	1





stack



Rebuild the tree by double-tagging preorder sequence

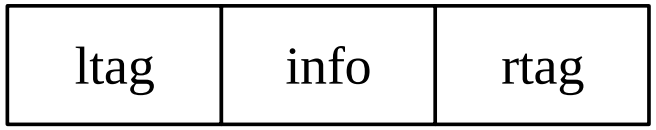
```
template<class T>
class DualTagTreeNode  {                          // class of double-tagging preorder sequence node
public:
    T info;                                       // data information of the node
    int ltag, rtag;                               // left/right tag
    DualTagTreeNode();                            // constructor
    virtual ~DualTagTreeNode();
 };

template <class T>
Tree<T>::Tree(DualTagTreeNode<T> *nodeArray, int count)  {
   // use double-tagging preorder sequence representation to build “Left-child/Right-sibling” tree
    using std::stack;                             // Use the stack of STL
    stack<TreeNode<T>* > aStack;
    TreeNode<T> *pointer = new TreeNode<T>;   // ready to set up root node
    root = pointer;
```

6.3 Sequential Storage Structure of Tree

```
for (int i = 0; i < count-1; i++) {            // deal with one node
  pointer->setValue(nodeArray[i].info);  // assign the value to the node
  if (nodeArray[i].rtag == 0)                  // if rtag equals to 0, push the node into the stack
     aStack.push(pointer);
  else pointer->setSibling(NULL);          // if rtag equals to 1, then right sibling pointer
                                                      // should be NULL
  TreeNode<T> *temppointer = new TreeNode<T>; // get ready for the next node
  if (nodeArray[i].ltag == 0)                  // if ltag equals to 0, then set the child node
     pointer->setChild(temppointer);
  else {                                              // if ltag equals to 1
     pointer->setChild(NULL);                 // set child pointer equal to NULL
     pointer = aStack.top();                    // get the top element of the stack
     aStack.pop();
     pointer->setSibling(temppointer); }  // set a sibling node for the top element of the stack
  pointer = temppointer; }
pointer->setValue(nodeArray[count-1].info); // deal with the last node
pointer->setChild(NULL);  pointer->setSibling(NULL);
}
```

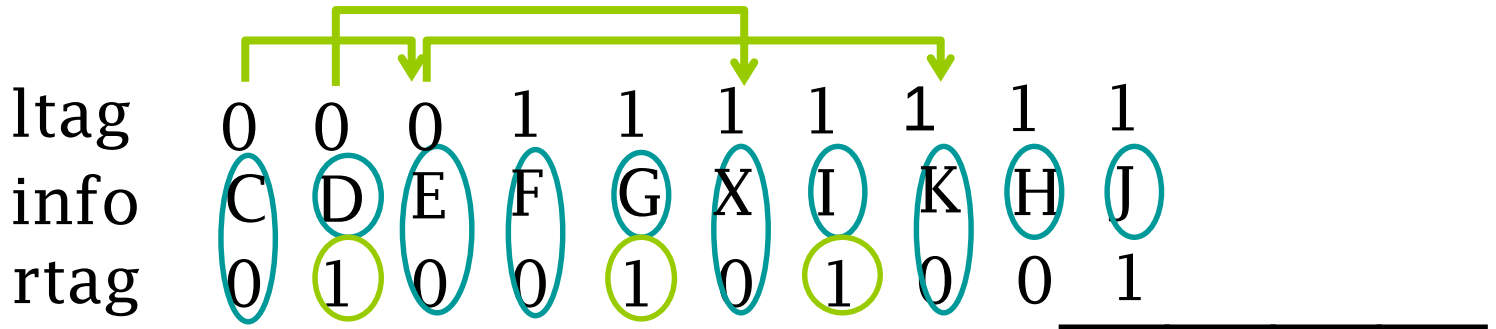
Double-tagging level-order sequence representation

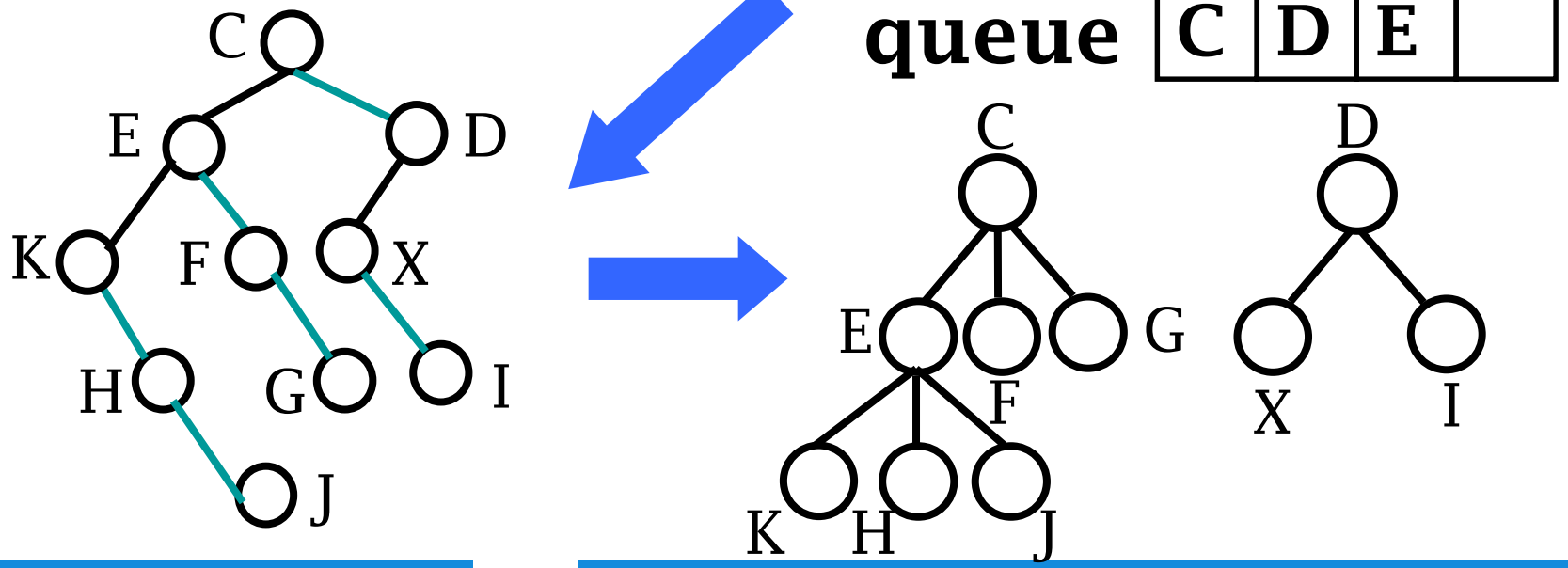
- Nodes are stored continuously according to level-order sequence



  - Info represents the data of the node.
  - ltag is a 1-bit tag, if the node doesn't have a child node, which means the node of the corresponding binary tree doesn't have a left child node, then ltag equals to 1, otherwise, ltag equals to 0.
  - rtag is a 1-bit tag, if the node doesn't have a right sibling node , which means the node of the corresponding binary tree doesn't have a right child node, then rtag equals to 1, otherwise, rtag equals to 0.

From a double-tagging level-order sequence to a tree





queue

C	D	E	

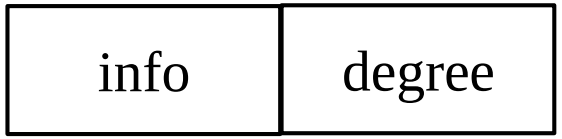
From a double-tagging level-order sequence to a tree

```
template <class T>
Tree<T>::Tree(DualTagWidthTreeNode<T>* nodeArray, int count)  {
  using std::queue;                                  // use the queue of STL
  queue<TreeNode<T>*> aQueue;
  TreeNode<T>* pointer=new TreeNode<T>; // build the root node
  root=pointer;
  for(int i=0;i<count-1;i++) {              // deal with each node
    pointer->setValue(nodeArray[i].info);
    if(nodeArray[i].ltag==0)
          aQueue.push(pointer);          // push the pointer into the queue
      else pointer->setChild(NULL);    // set the left child node as NULL
    TreeNode<T>* temppointer=new TreeNode<T>;
```

```
    if(nodeArray[i].rtag == 0)
      pointer->setSibling(temppointer);
    else {
        pointer->setSibling(NULL);     // set the right sibling node as NULL
        pointer=aQueue.front();        // get the pointer of the first node in the queue
        aQueue.pop();                  // and pop it out of the queue
        pointer->setChild(temppointer);
    }
    pointer=temppointer;
  }
  pointer->setValue(nodeArray[count-1].info); // the last node
  pointer->setChild(NULL);  pointer->setSibling(NULL);
}
```

Postorder sequence with degree representation

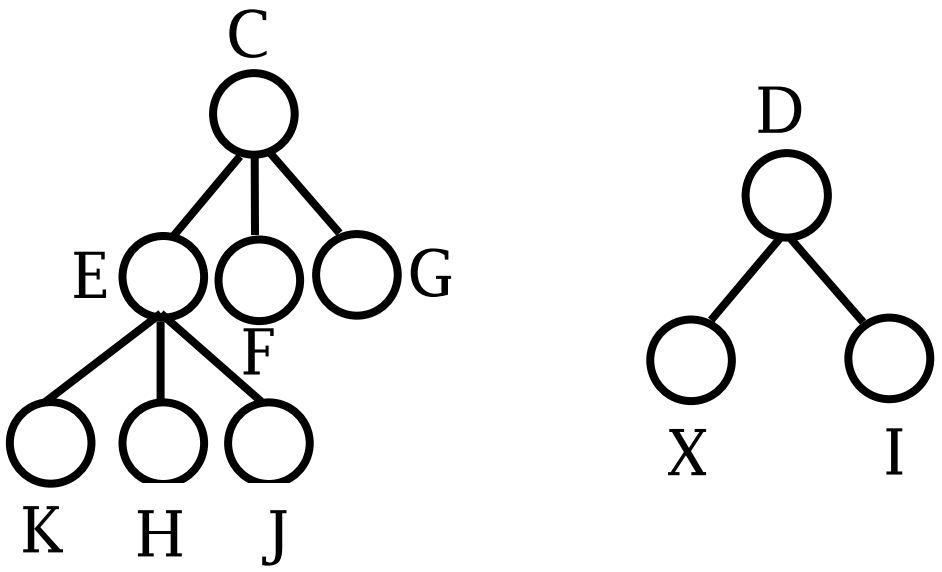
- In postorder sequence with degree representation, nodes are stored contiguously according to postorder sequence, whose form are like:



- info represents the data of the node, and degree represents the degree of the node

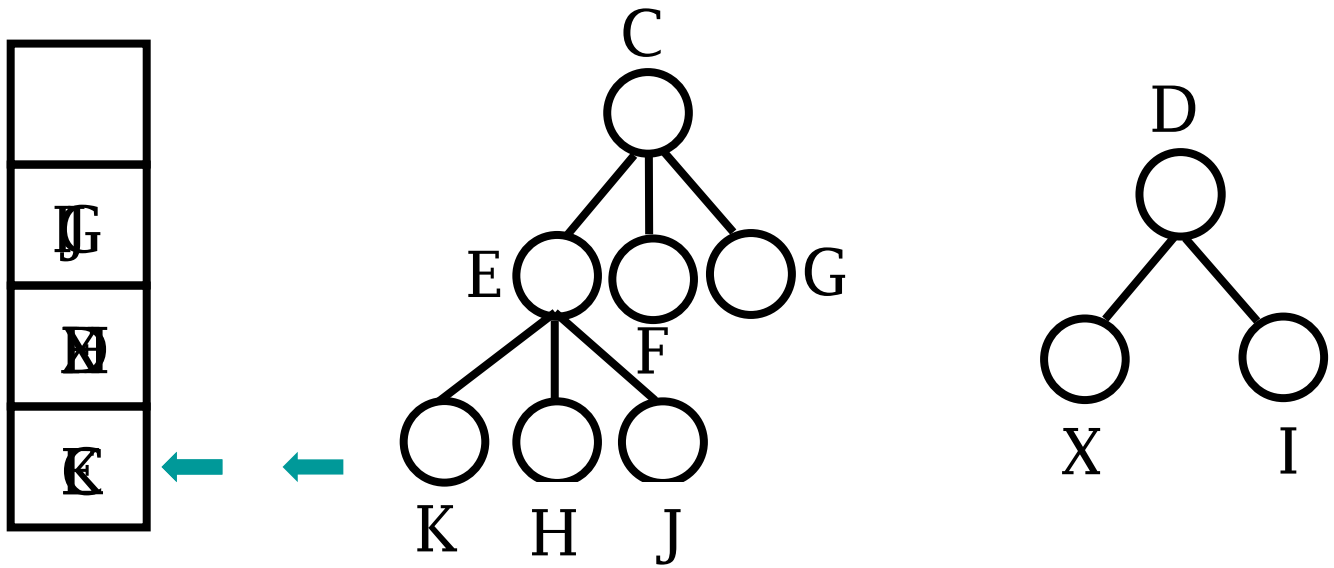
Postorder sequence with degree representation

degree	0	0	0	3	0	0	3	0	0	2
info	K	H	J	E	F	G	C	X	I	D



Postorder sequence with degree representation

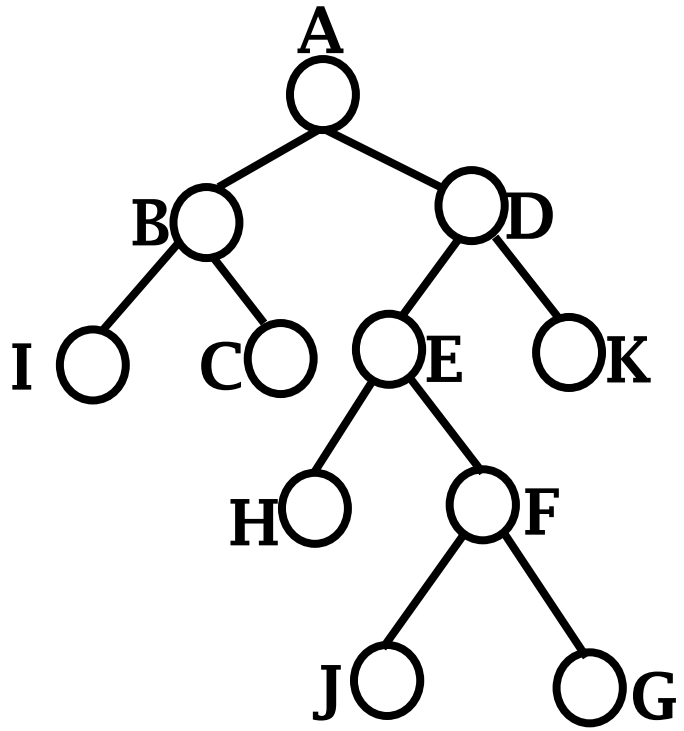
degree	0	0	0	3	0	0	3	0	0	2
info	K	H	J	E	F	G	C	X	I	D



6.3 Sequential Storage Structure of Tree

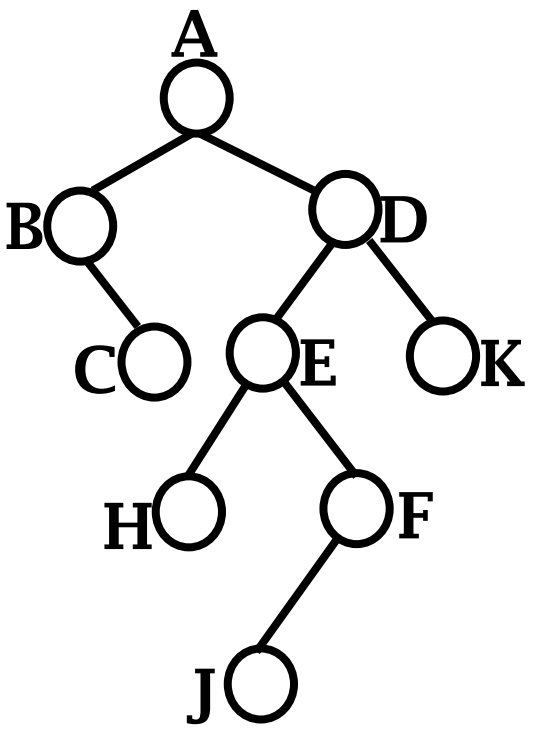
- Preorder sequence of the full tagging binary tree

  A' B' I C D' E' H F' J G K



- Preorder sequence of the virtual full tagging binary tree

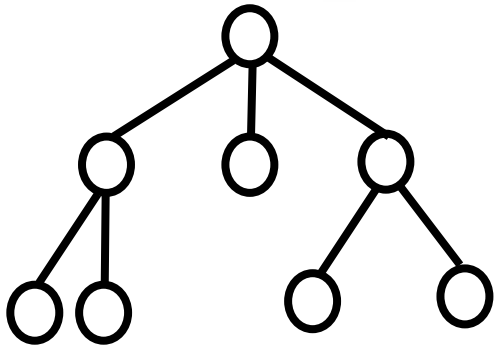
  A' B' / C D' E' H F' J / K



Thinking: Sequential Storage of the Forest

- Information redundancy
- Other sequential storage of tree
  - Preorder sequence with degree?
  - Level-order sequence with degree?
- Sequential storage of the binary tree?
  - The binary tree is corresponding to the tree, but their semantemes are different
    - Preorder sequence of the binary tree with right link
    - Level-order sequence of the binary tree with left link

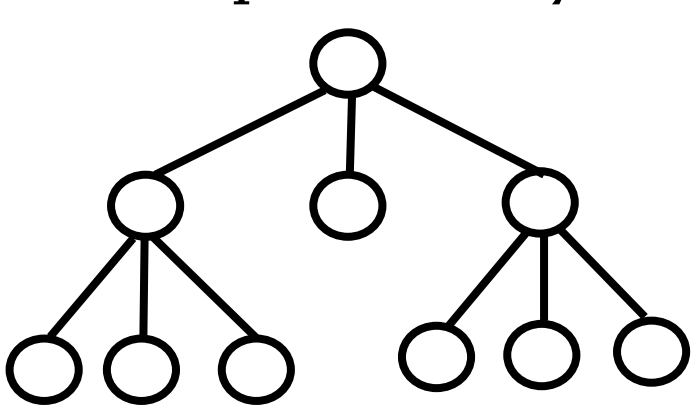
Definition of K-ary tree



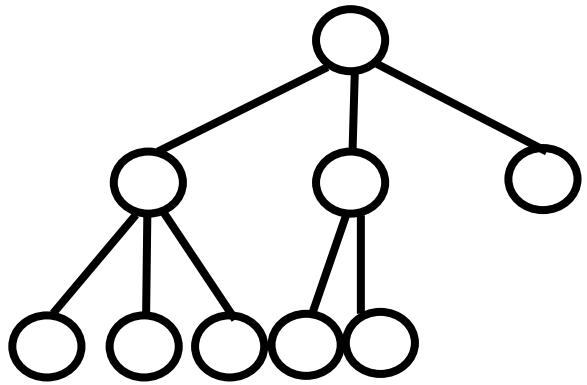
- K-ary tree T is a finite node set, which can be defined recursively:
  - (a) T is an empty set.
  - (b) T consists of a root node and K disjoint K-ary subtrees.
- Nodes except the root R are devided into K subsets (T_0, T_1, ..., T_{K-1}), and each subset is a K-ary tree, such that T = {R, T_0, T_1, ..., T_{K-1}}.
- Each branch node of K-ary tree has K child nodes.

Full K-ary trees and complete K-ary trees

- The nodes of K-ary tree have K child nodes
- Many properties of binary tree can be generalized to K-ary tree
  - Full K-ary trees and complete K-ary trees are similar to full binary trees and complete binary trees
  - Complete K-ary trees can also be storeed in an array



Full 3-ary tree



Complete 3-ary tree

Data Structures and Algorithms

Thanks

the National Elaborate Course (Only available for IPs in China)
http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/

Ming Zhang, Tengjiao Wang and Haiyan Zhao

Higher Education Press, 2008.6 (awarded as the "Eleventh Five-Year" national planning textbook)