# Data Structures and Algorithms ( 6 )

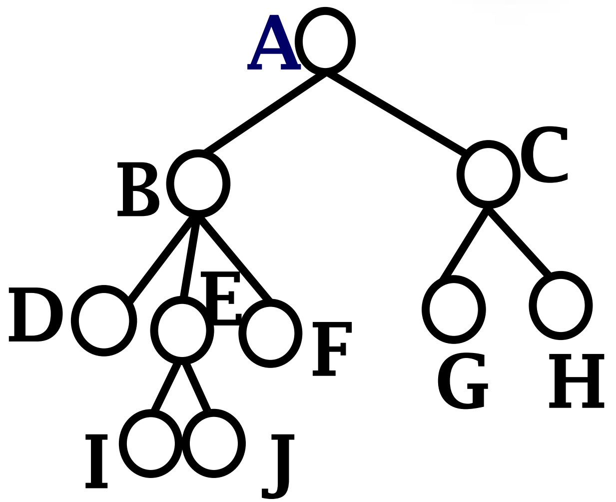
**Instructor: Ming Zhang**

**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
- 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

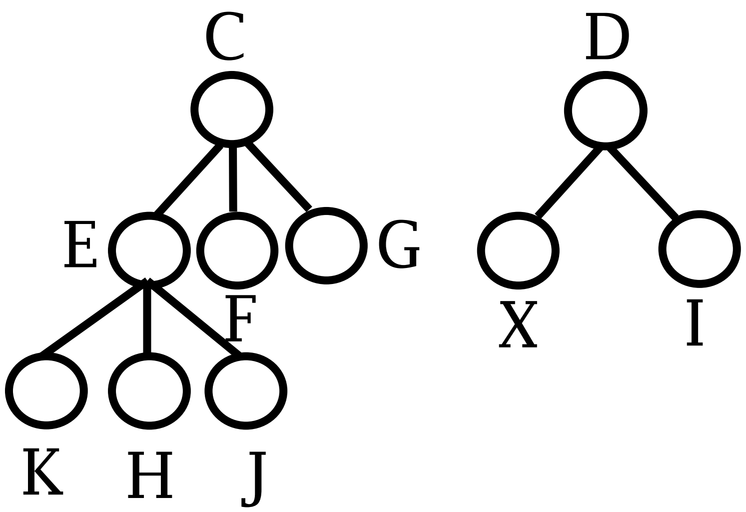
# Preorder sequence with right link representation

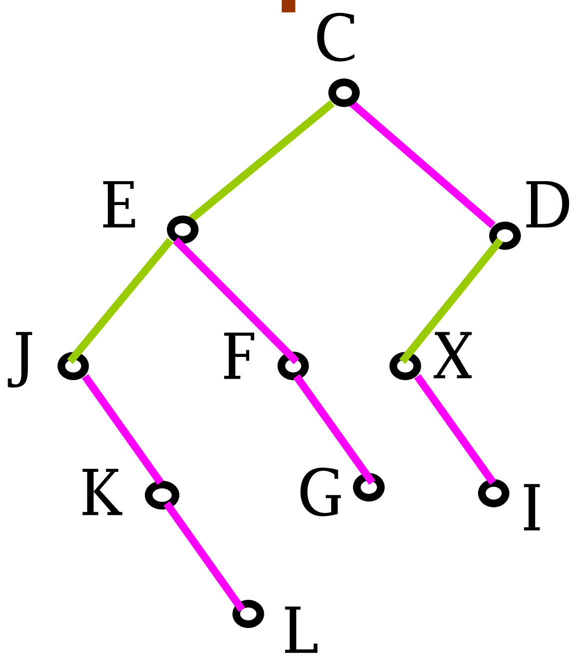
- Nodes are stored continuously according to preorder sequence

| 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.

# Preorder sequence with right link representation

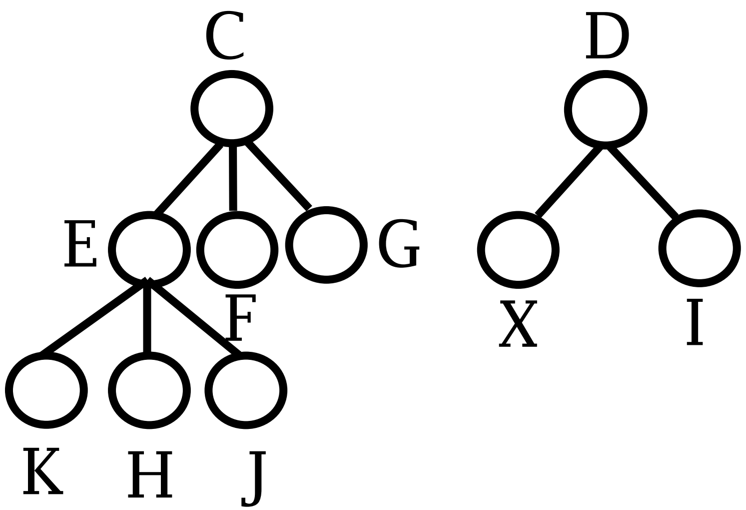


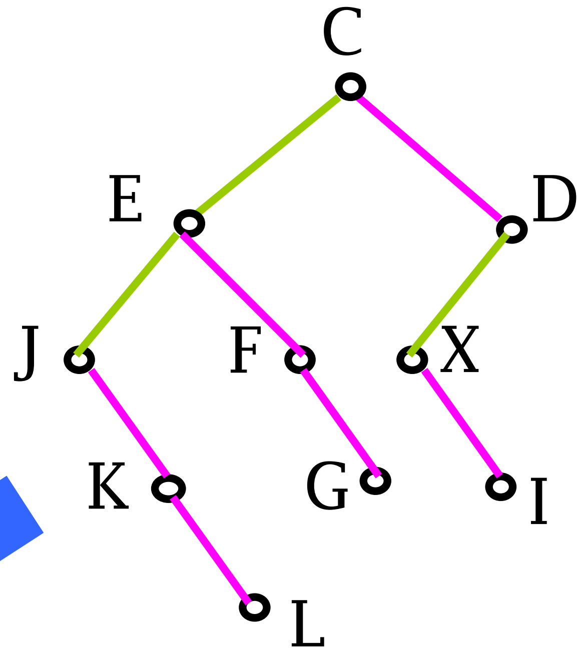




| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| rlink | 7 | 5 | 3 | 4 | -1 | 6 | -1 | -1 | 9 | -1 |
| info | C | E | J | K | L | F | G | D | X | I |
| ltag | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |

# From a preorder rlink-ltag to a tree

| 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 |



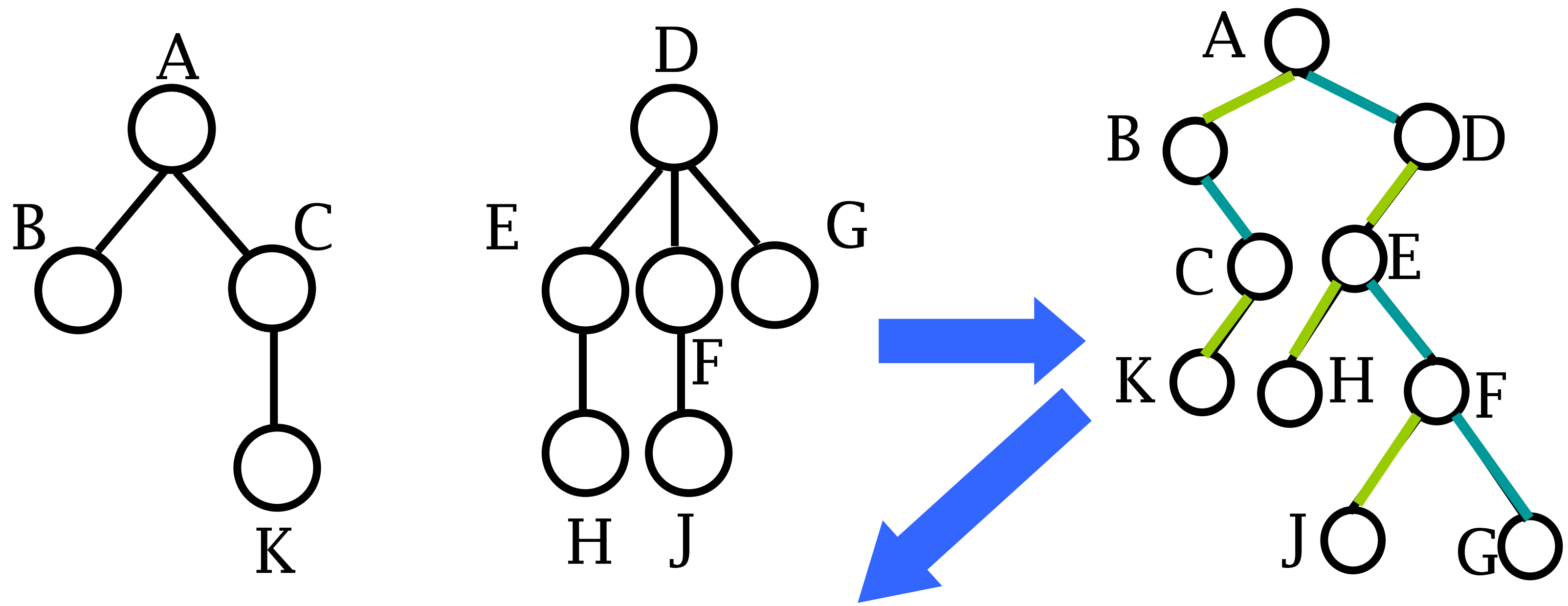
# Double-tagging preorder sequence representation

- 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:

| ltag | info | rtag |
|---|---|---|

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.

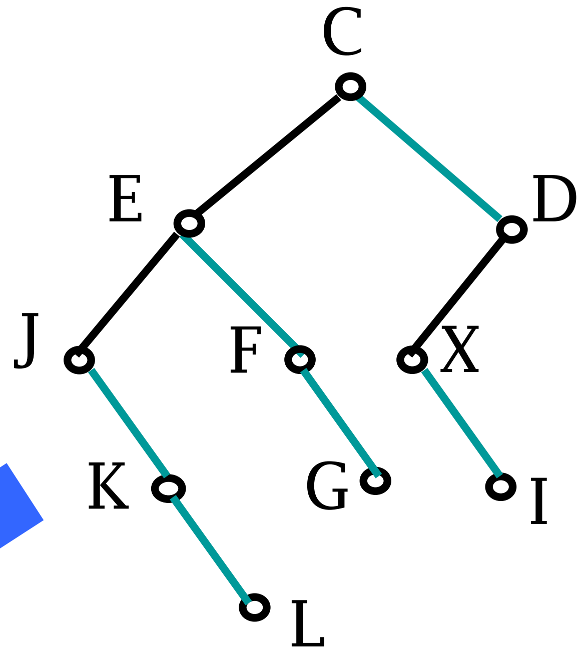
# Double-tagging preorder sequence representation

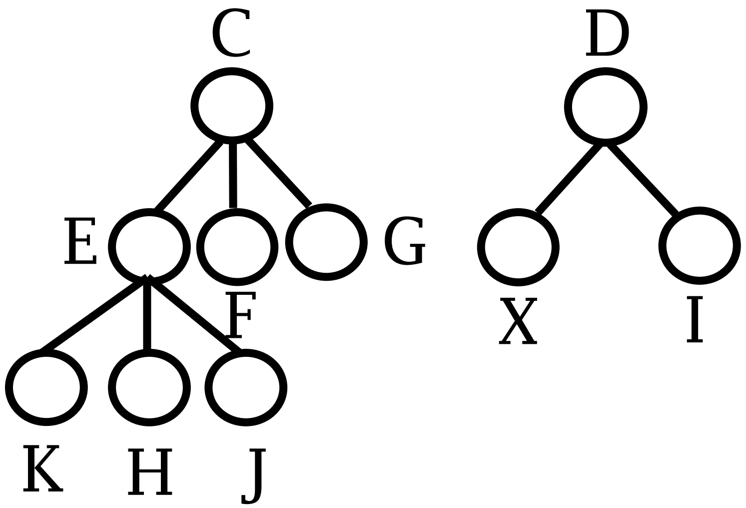



| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| rtag | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 |
| info | A | B | C | K | D | E | H | F | J | G |
| ltag | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |

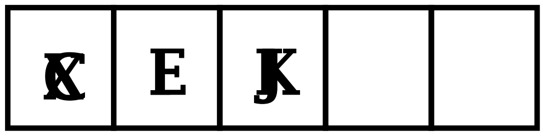
# From a rtag-ltag preorder sequence to a tree

| 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

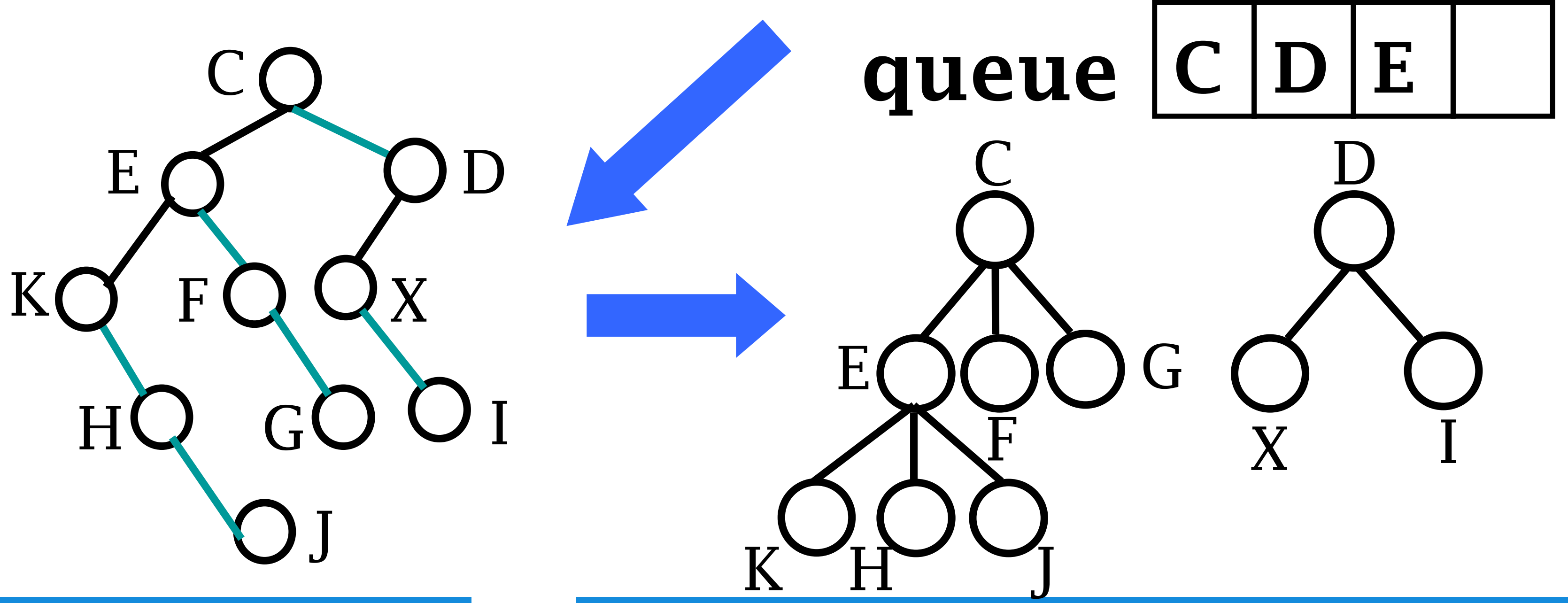
- Nodes are stored continuously according to level-order sequence

| | | |
|---|---|---|
| ltag | info | rtag |

  - 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

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| ltag | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| info | C | D | E | F | G | X | I | K | H | J |
| rtag | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |



# 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

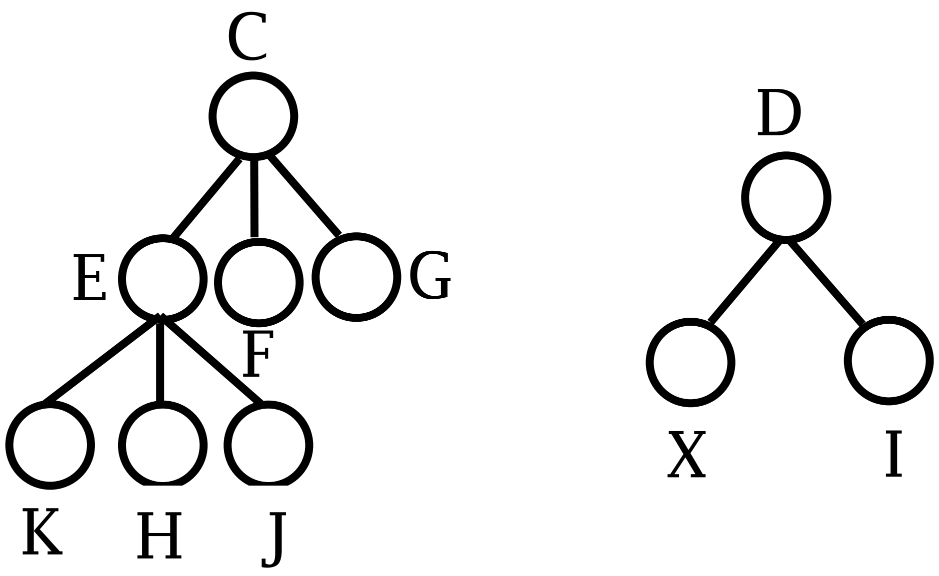
- In postorder sequence with degree representation, nodes are stored contiguously according to postorder sequence, whose form are like:

| info | degree |
| --- | --- |

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

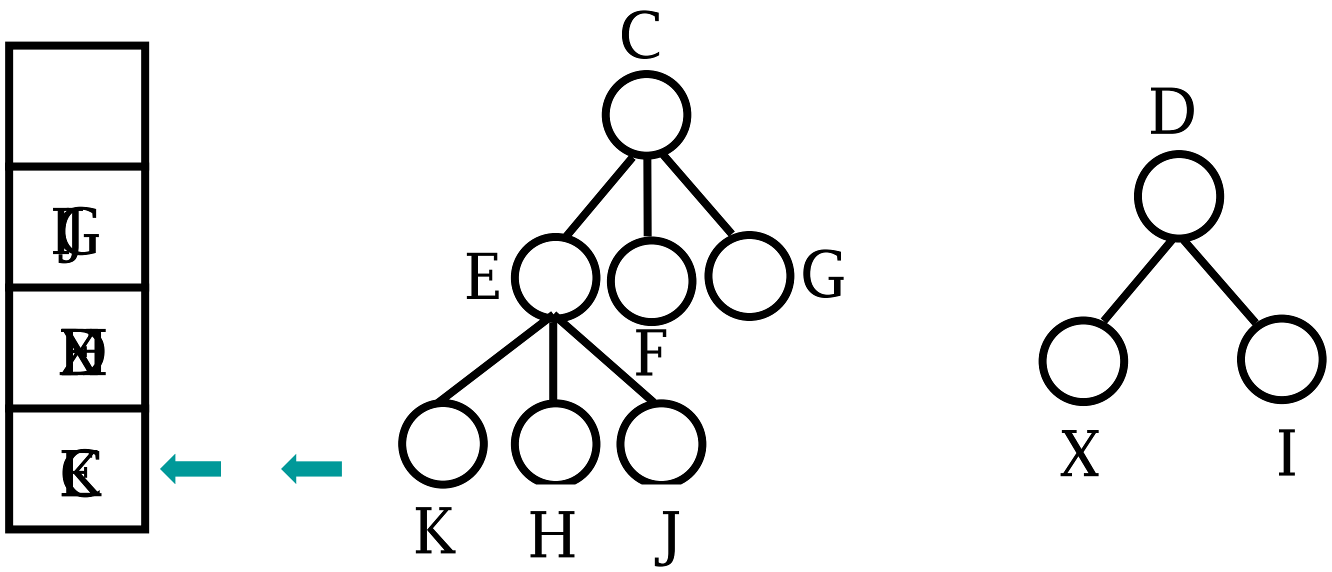
# Postorder sequence with degree representation

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 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

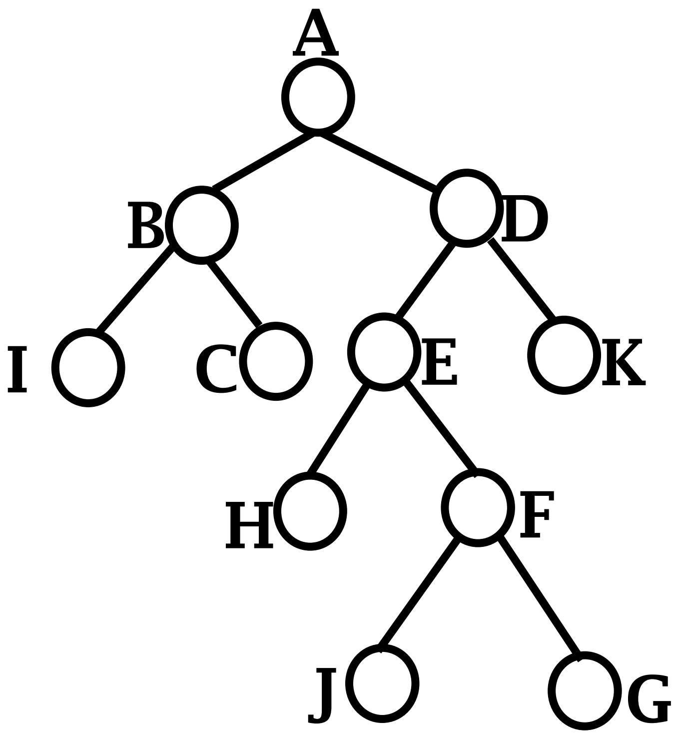
| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| degree | 0 | 0 | 0 | 3 | 0 | 0 | 3 | 0 | 0 | 2 |
| info | K | H | J | E | F | G | C | X | I | D |



# 6.3 Sequential Storage Structure of Tree

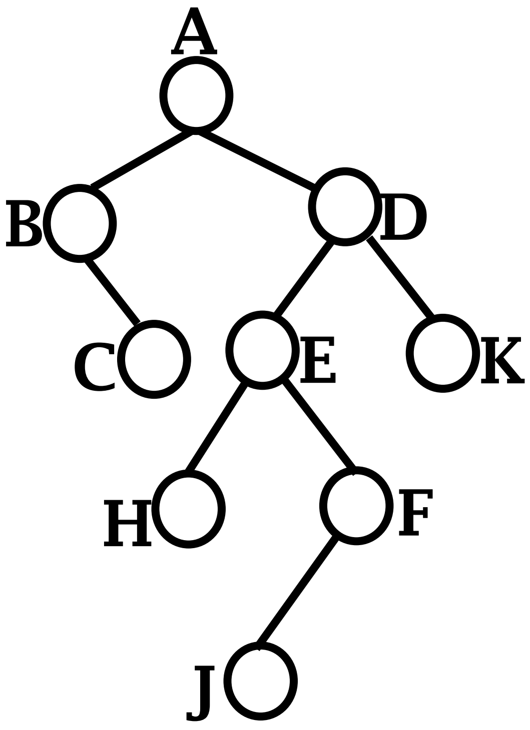
- Preorder sequence of the full tagging binary tree

A’ B’ I C D’ E’ H F’ J G K




- Preorder sequence of the virtual full tagging binary tree

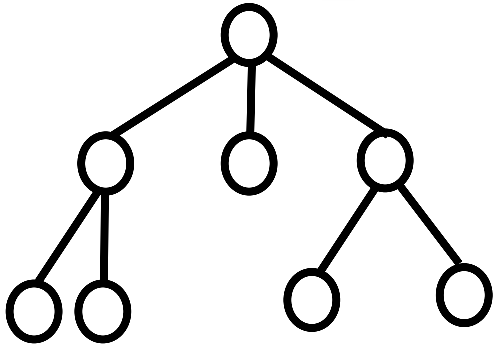
A’ B’ / C D’ E’ H F’ J / K



# 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

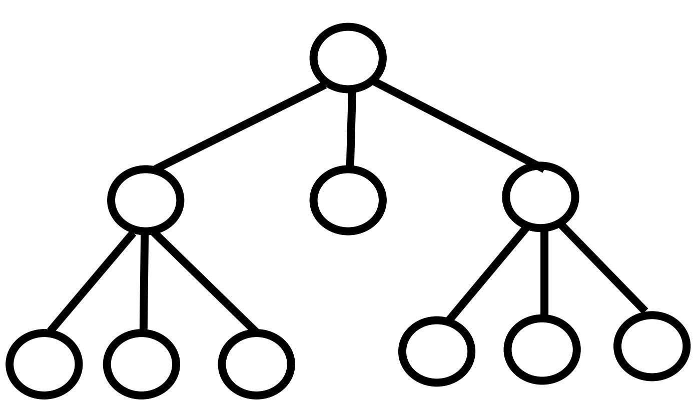
# Definition of K-ary tree



- 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)**