EECS560 Data Structures Kong

Topic 3: Review on Tree-Based Data Structures & their Implementations

Read: Chpt. 4, Weiss

Q: Why study tree-based data structures?

Standard linear data structures (arrays, lists, stacks, …) are no longer sufficient for many applications; most advance ADTs are implemented using tree-based data structures.

Recursive Definition of a (Rooted) Tree:

Let T be a set with $n \geq 0$ elements.

(i) If $n = 0$, T is an ***empty tree***,

(ii) If $n > 0$, then there exists a distinct element $r \in T$, called the ***root*** of T, such that $T-\{r\}$ can be partitioned into zero or more disjoint subsets T_1, T_2, ..., where each of these subsets also forms a tree.

Dfn: Elements of T are ***nodes*** in the tree T.
Each subset T_i is a ***subtree*** *of r*.
The roots of the subtrees of r are ***children*** *of r*.
The root r is the ***parent*** of the roots of its subtrees.
Nodes with the same parents are ***siblings***.
Nodes with no children are ***leaves***.

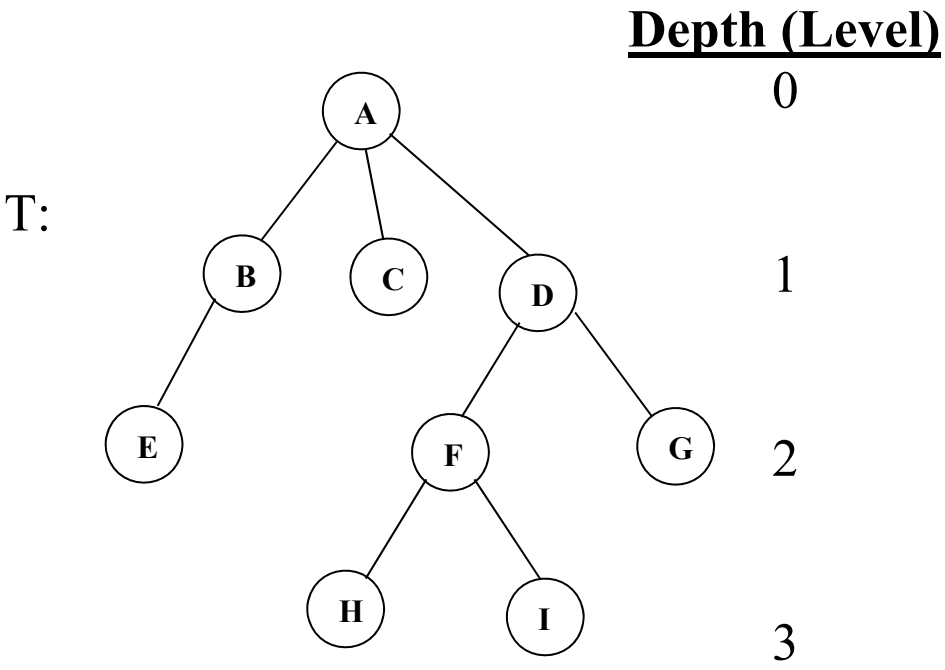
The # of children of a node is the ***degree*** of the node.
A ***path of length k*** (in a tree), $k \geq 0$, from node x to node y is a sequence of k+1 nodes $x = n_0, n_1, n_2, ..., n_k = y$ such that n_i is the parent of n_{i+1} for all $i = 0, 1, ..., k-1$.
If there is a path from x to y, then x is an ***ancestor*** of y and y is a ***descendant*** of x.
For any given node x in a tree, the ***height*** of x is the length of a longest path (in length) from x to any leaf.
The ***depth*** **(*level*)** of x is the length of the (unique) path from the root to x.
The ***height of a tree*** is the height of its root and the ***depth of a tree*** is the maximum depth of its nodes.

Remarks:

1. There is a unique path from the root to each node in the tree.
2. There is a path of zero length from any node to itself.
3. Depth of the root of a tree is 0 and height of a leaf is also 0.
4. Height of tree = Depth of tree.
5. For convenience, the height of an empty tree is defined to be -1.

Warning: The terminologies used for tree in data structures are not the same as for trees in graphs.

Graphical Representation of a Tree T:



A is the ***root*** of tree T.
D is the ***parent*** of F and G.
F and G are ***children*** of D.
F has ***degree*** 2.
B, C, D are ***siblings***.
C, E, G, H, I are ***leaves***.
D is an ***ancestor*** of I and I is a ***descendant*** of D.
There is no ancestor-descendant relation between H and G.
(A,D,F,H) is a ***path*** of length 3 from A to H.
The node H has ***height*** 0 and ***depth*** 3.
The ***height (depth) of the tree*** is 3.

Some Important Classes of Trees:

1. ***k-ary trees***, $k \geq 2$:

   A k-tree is a tree with each node having at most k children, $k \geq 2$.

2. ***Binary tree***:

   A binary tree is an ***ordered 2-ary tree***.

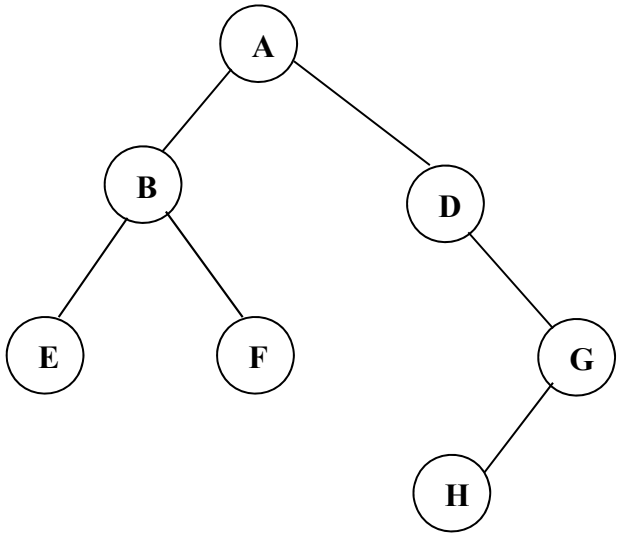
Recursive Definition of Binary Tree:

Let T be a set with $n \geq 0$ elements. T is a binary tree iff

(i) T is empty, or

(ii) if T is not empty, T has a root $r \in T$ such that $T-\{r\}$ can be partitioned into two disjoint binary trees T_L and T_R , called the ***left subtree*** and ***right subtree*** of r.

Example: A binary trees.



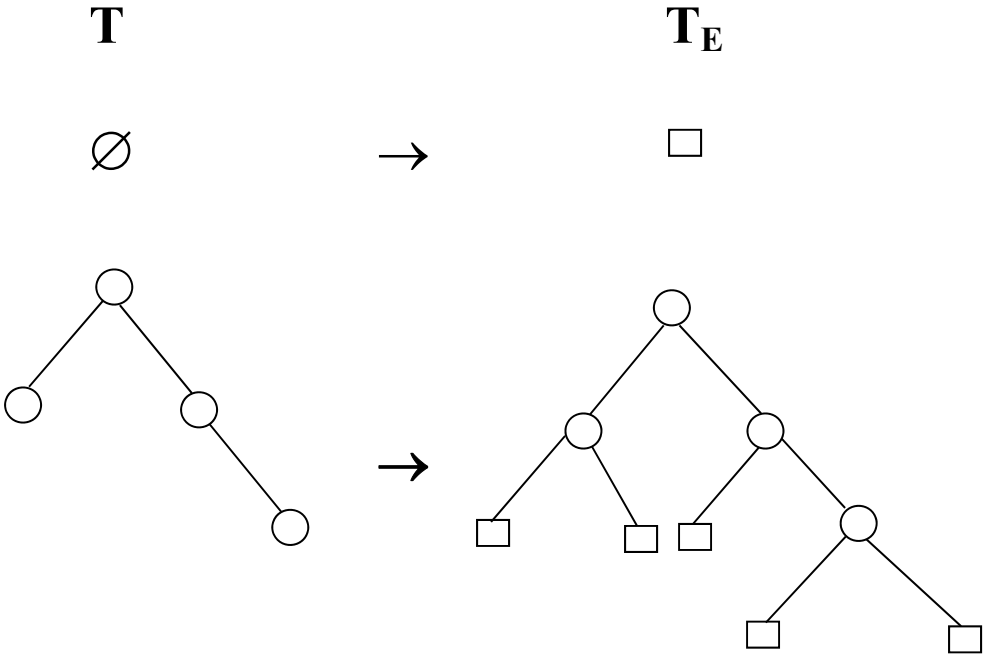
3. ***Extended binary tree***:

For any given binary tree T with n nodes, $n \geq 0$, the ***extended binary tree*** (EBT) T_E of T, is a binary tree constructed from T by introducing new leaf nodes to T_E such that each (original) node in T will have exactly two children in T_E.
The (original) nodes in T are called ***internal nodes*** and the (new) nodes in $T_E - T$ are called ***external nodes*** of the EBT T_E.

Special Case:

When n = 0, the EBT of an empty tree T consists of a single external node.

Example:



Consider a given extended binary tree T_E with n internal nodes.

Definition:

External path length of T_E:

$E(T_E)$ = sum of the level numbers (depth) of all external nodes.

Internal path length of T_E:

$I(T_E)$ = sum of the level numbers (depth) of all internal nodes.

Remarks:

1. #external nodes = #internal nodes + 1

2. $E(T_E) = I(T_E) + 2n$.

3. Average distance to an external node = $\frac{E(T_E)}{n+1}$.

4. Average distance to an internal node = $\frac{I(T_E)}{n}$.

Q. What kind of extended binary trees will have maximum (minimum) internal and external path length?

除了最后一层，其他层数必须满，最后最后一层从左往右开始生成nodes。

4. ***Complete binary tree***:

A complete binary tree T of height h is a binary tree satisfying the following two conditions:

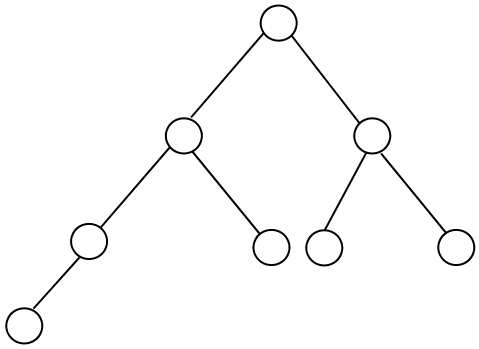
(i) Except possibly at level h, each level i must contain exactly 2^i nodes.

(ii) Nodes at level h must be left-justified.

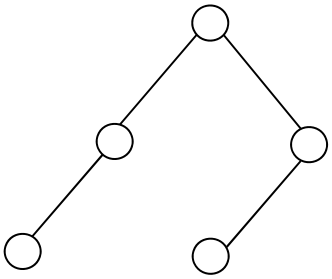
Examples:

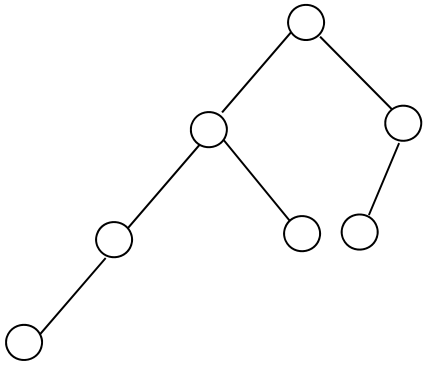
(a) A complete binary tree:

2^(level)+1



(b) Two incomplete binary trees:



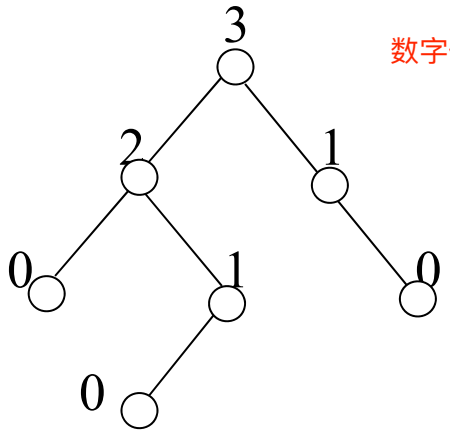


5. ***Balanced binary tree***:

A balanced binary tree T is a binary tree such that for any node x in the tree, the height of its left subtree differs by no more than one from the height of its right subtree. Hence, $|h(T_L(x)) - h(T_R(x))| \leq 1$, where $h(T_L(x))$, and $h(T_L(x))$, are the height of the left, and right, subtree rooted at x, respectively.

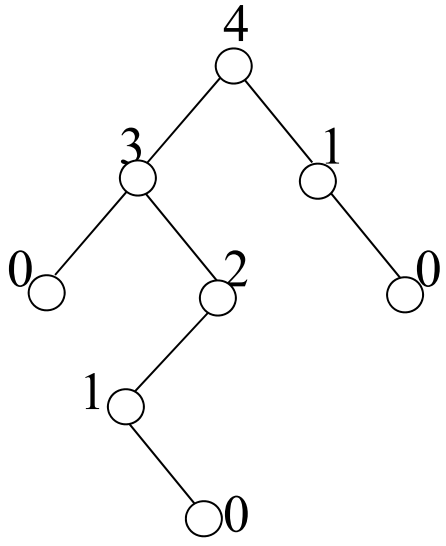
Example:

Balanced binary tree with height of nodes:



数字代表节点的高度

Unbalanced binary tree with height of nodes:

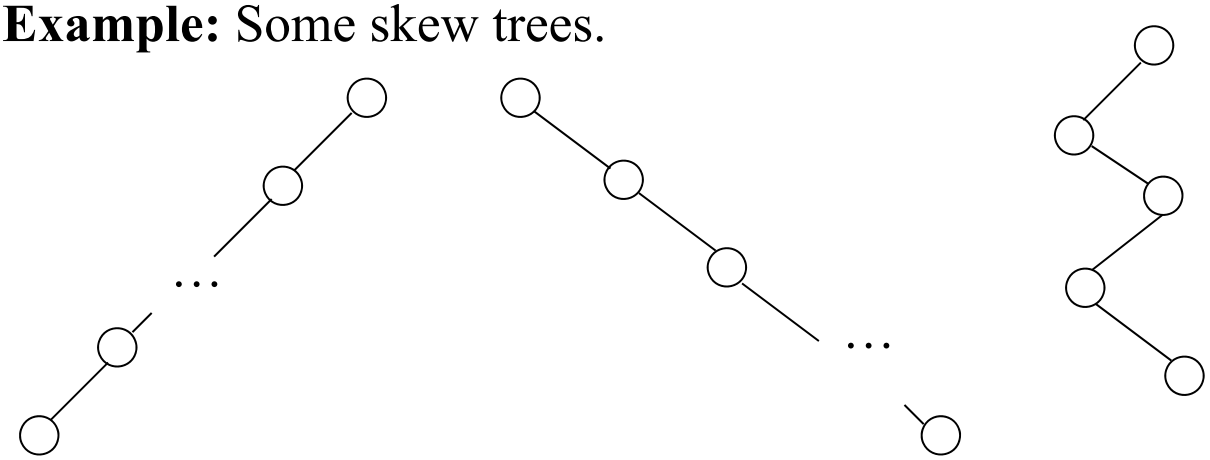


差评！只有一个孩子的树，效率最低，最好避免出现的，以为树的root选的不好

6. ***Skew tree***:

A skew tree T is a binary tree such that each non-leaf node in T is having exactly one child.

Example: Some skew trees.

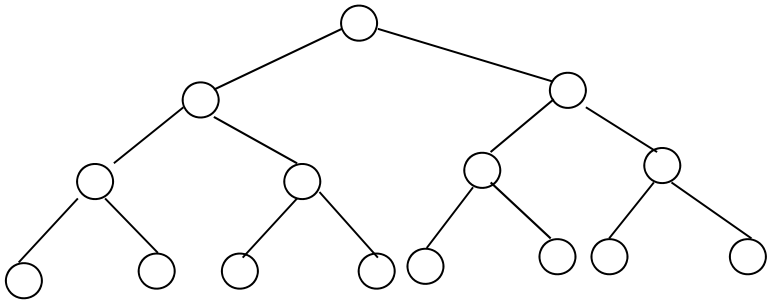


7. Full binary tree:

每一个节点都有两个children,高度决定节点数量

A full binary tree of height $h \geq 0$ is a binary tree such that each level i must contain exactly 2^i nodes, $0 \leq i \leq h$. Hence, each non-leaf node in T must have exactly two children and T has exactly $2^{h+1}-1$ nodes.

Example: A full binary tree with h = 3.



2^(3+1)-1 = 15 nodes

When implementation an ADT using a tree-based data structure, either non-leaf nodes or leaf-nodes can be used to hold the data objects of the ADT and the performance of an ADT operation will usually depend on the height of the tree.

Nodes, Leaves, and Heights of Binary Trees:
Consider a non-empty binary tree T.

1. T with height h, $h \geq 0$:

Skew tree Min # nodes = h+1. Max # nodes $= 2^{h+1} - 1$ Full Binary tree

2. T with height h, $h \geq 0$:

Min # leaves = 1. Max # leaves $= 2^{h}$.

3. T with n nodes, $n \geq 1$:

height 是从0开始计算的

Complete Binary tree Min height $= \lfloor \log_2 n \rfloor$. 向下舍 Max height = n–1. Skew tree

4. T with n nodes, $n \geq 1$:

Skew tree Min # leaves = 1. Max # leaves $= \lceil n/2 \rceil$. 向上进位

5. T with m leaves, $m \geq 1$:

Min height $= \lceil \log_2 m \rceil$. Max height = ∞.

6. T with m leaves, $m \geq 1$:

Min # nodes = 2m–1. Max # nodes = ∞.

HW: Verify the above results.
Can you generalize these results to k-ary trees with $k \geq 2$?

Applications of Tree Traversals:

1. To explore the underlying hierarchical structures.
2. To retrieve information stored in the tree.
3. To restore the topological structure of a tree-based data structure.

Basic Binary Tree Traversal Algorithms:

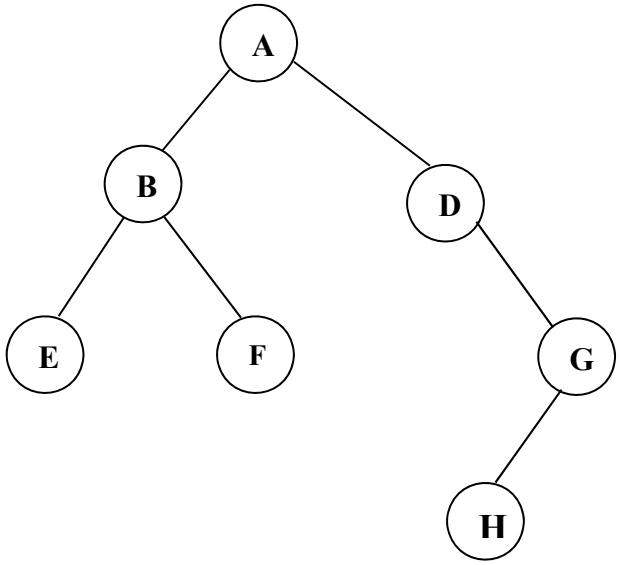
1. ***Preorder traversal:***
   Traverse/retrieve root,
   Traverse left subtree recursively in preorder,
   Traverse right subtree recursively in preorder.

2. ***Postorder traversal:***
   Traverse left subtree recursively in postorder,
   Traverse right subtree recursively in postorder,
   Traverse/retrieve root.

3. ***Inorder traversal:***
   Traverse left subtree recursively in inorder,
   Traverse/retrieve root,
   Traverse right subtree recursively in inorder.

4. ***Level-order traversal:***
   Starting at level 0, traverse the nodes at each level from left to right.

Example: Binary tree traversals.



Preorder: A B E F D G H
Postorder: E F B H G D A
Inorder: E B F A D H G
Level-order: A B D E F G H

HW: Review and implement the above tree traversal algorithms.

More on binary tree traversals:

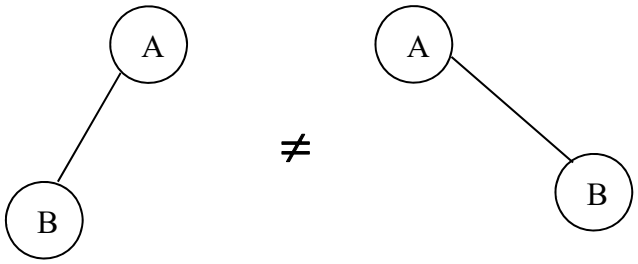
Observe that for a given binary tree T, one can traverse T easily.

Binary tree is a tree with the root, get left children, and right children

Q: If a traversal of a binary tree T is given, can we reconstruct the given binary tree T?

No!

Consider the following binary trees:



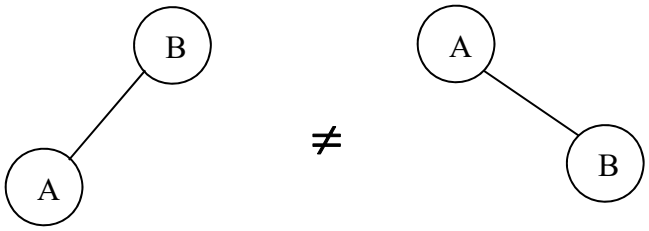
On traversing both trees, we have

Preorder traversal: A B
Postorder traversal: B A
Level-order traversal: A B

Hence, these two binary trees are indistinguishable if only one of these three tree traversals is given!

Q: How about inorder traversal?

Consider the following binary trees:



Both binary trees have the same inorder traversal: AB and are indistinguishable using only inorder traversal.

Q: How can we reconstruct the binary tree T from its traversal(s)?

One must know the root, the left and the right subtrees of the binary tree.

If we are given any one of the following pairs of traversals of T, T can be reconstructed if exists.

1. Preorder and inorder traversals.
2. Postorder and inorder traversals.
3. Level-order and inorder traversals.

HW: Can you reconstruct a binary tree T having the following pairs of traversals? Algorithms?

Preorder: B A D C E F G I H
Inorder: A F E G C D I B H

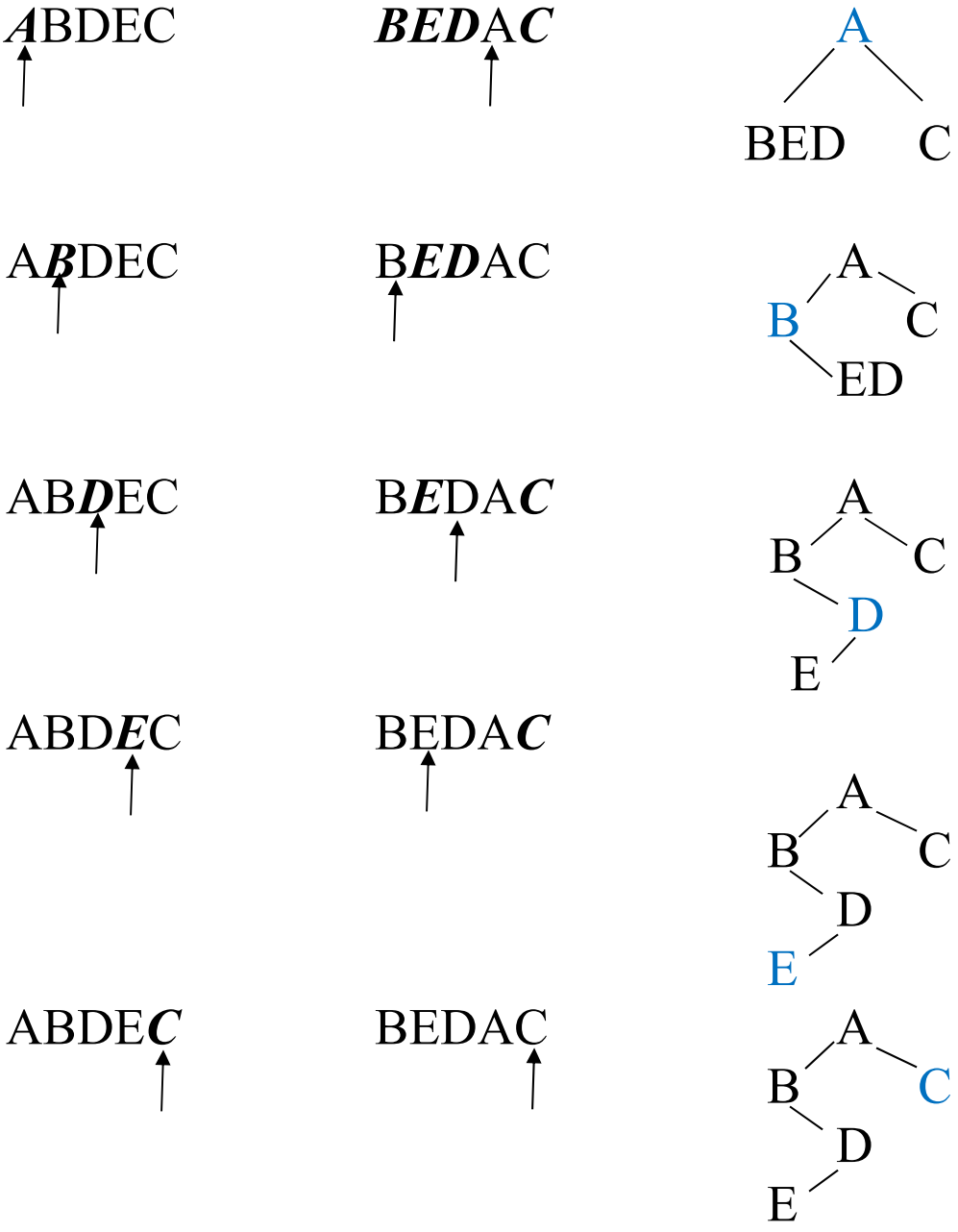
Preorder: A C D H I E B F G
Inorder: D I H C E A B G F

Preorder: A C D H I E B F G
LevelOrder: A C B D E F H G I

Postorder: A K J G B D H F I E C
Inorder: A D K G J B C F H E I

Example: Let preorder = ABDEC and inorder = BEDAC.

Approach: Scan preorder traversal from left to right to determine root followed by scanning inorder traversal to determine left and right subtree.



Extension: General Tree Traversals

1. ***Preorder traversal***:

   Traverse/retrieve root,
   Traverse subtree T_1 recursively in preorder,
   Traverse subtree T_2 recursively in preorder,
   • • •
   Traverse subtree T_k recursively in preorder.

2. ***Postorder traversal***:

   Traverse subtree T_1 recursively in postorder,
   Traverse subtree T_2 recursively in postorder,
   • • •
   Traverse subtree T_k recursively in postorder,
   Traverse/retrieve root.

3. ***Level order traversal***:

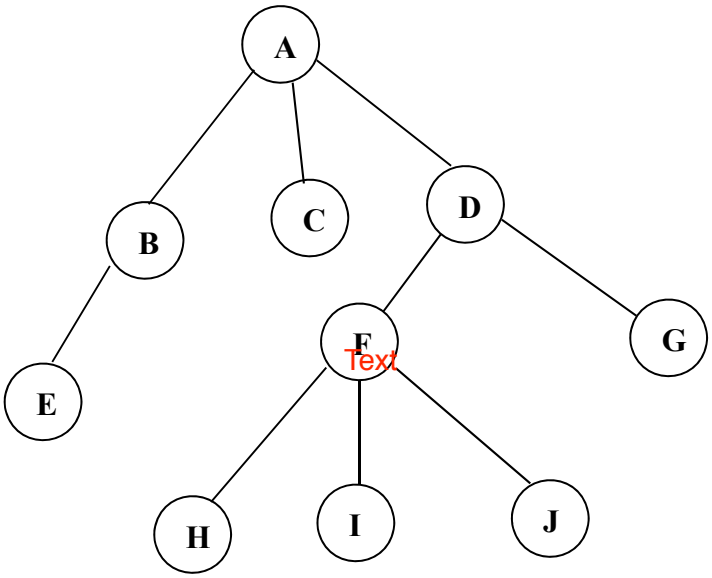
   Starting at level 0, traverse the nodes at each level from left to right.

4. *Inorder traversal:*

important:Inorder 在普通树不行，因为是左中右

   Since the concept of *in-between* is not well-defined for a general tree, we don't usually use inorder traversal for general tree!

Example: General tree traversals.



Preorder: A B E C D F H I J G
Postorder: E B C H I J F G D A
Level order: A B C D E F G H I J

Basic Binary Tree Operations:

(1) createTree()
(2) destroyTree()
(3) isEmpty()
(4) setRootData()
(5) getRootData()
(6) attachLeftTree()
(7) attachRightTree()
(8) detachLeftTree()
(9) detachRightTree()
(10) getLeftTree()
(11) getRightTree()
(12) preorderTraversal()
(13) postorderTraversal()
(14) inorderTraversal()
(15) levelorderTraversal()

Binary Tree Implementations:

Depend on speed and memory requirements.

Movement in tree:

up (require parent info)

down (require children info)

sideway (require siblings info)

Basic Implementations of Trees: 考点

Using array, pointer, or hybrid implementations.

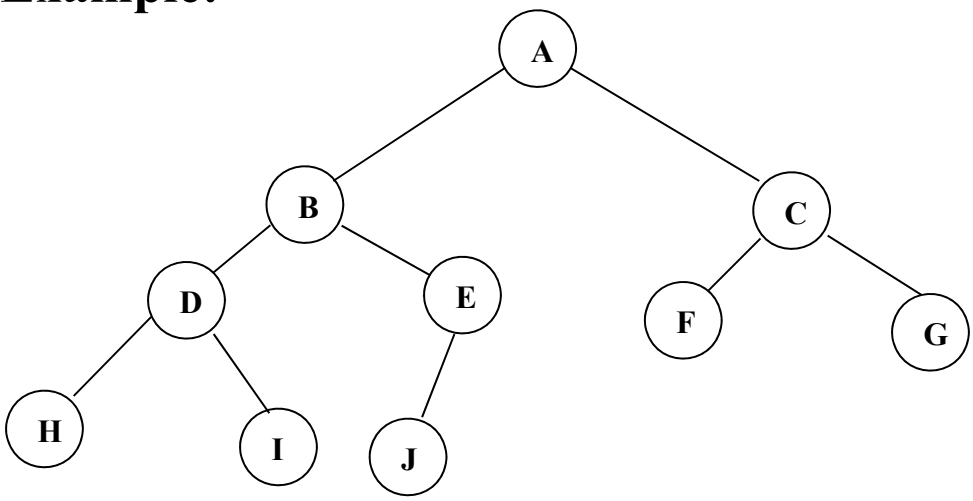
1. Array-based Implementation of (Complete) Binary Tree:

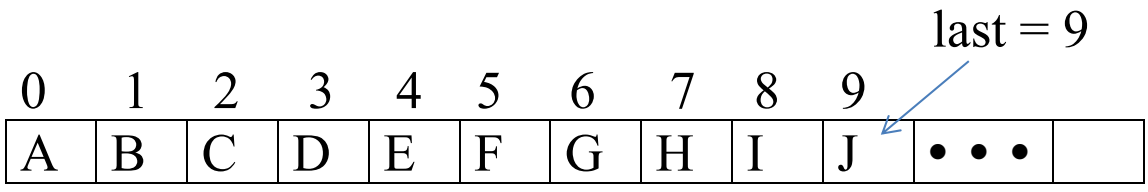
   Given a complete binary tree T with n nodes. T can be implemented using an array A[0:n−1] such that

   (1) Root of T at A[0],
   (2) Parent of a node A[i] at A[(i−1)/2] if exists,
   (3) Left child (right child) of a node A[i] at A[2i+1] (A[2i+2]) if exist.

   Observe that, for n ≥ 1, A[i] is a leaf node iff 2i ≥ n−1.

Example:





Advantages: Fastest, Θ(1) time, in accessing parent and children locations.

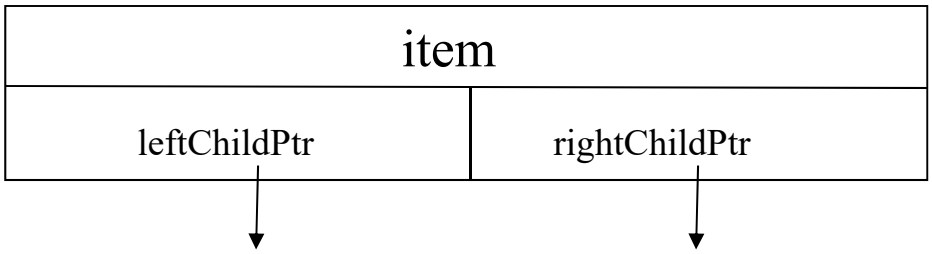
size与高度有关系

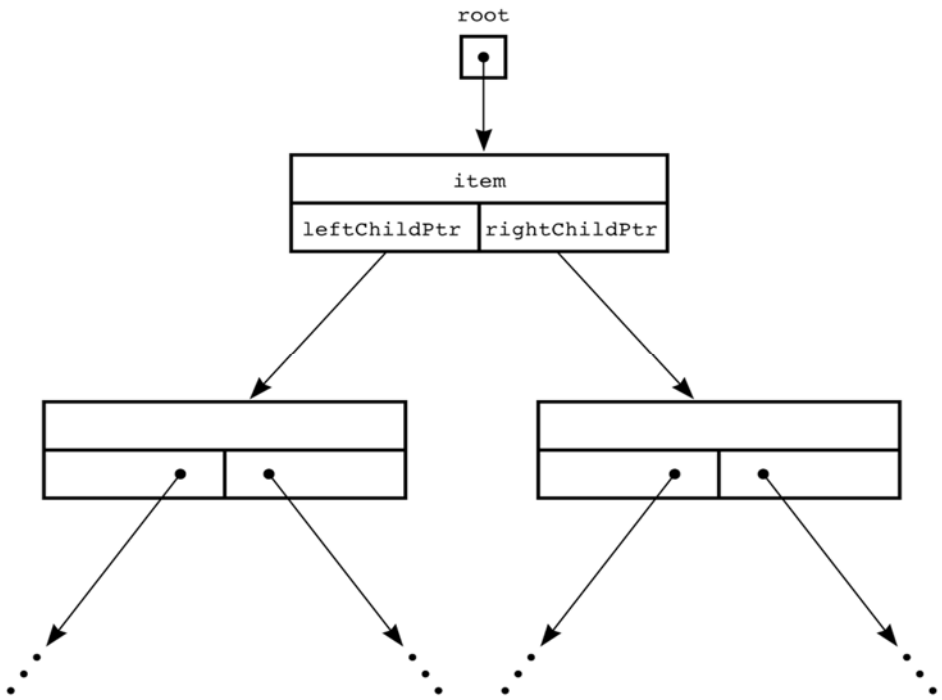
Disadvantage: Only useful when tree is complete (or "almost complete"); otherwise, very memory intensive. For a (right-skew) tree with height h, it requires an array of size $2^{h+1}-1$. Hence, a skew tree with 10 nodes with height 9 will require an array of size $2^{10} - 1 = 1023$ for its implementation.

HW: What if root of T is stored at A[1] instead? For a node stored in A[i], i ≥ 1, compute its parent, left child, and right child location.

2. Children-Pointer Implementation of Binary Tree:

NodeType:





The external pointer root points at the root r of the tree. If the tree is empty, root is NULL; otherwise, root→leftChildPtr (root→rightChildPtr) points to the root of the left (right) subtree of r.

TreeNode Class:

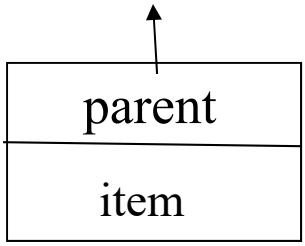
```
typedef string TreeItemType;

class TreeNode                          // node in the tree
{
private:
  TreeNode() {};
  TreeNode(const TreeItemType& nodeItem,
                          TreeNode *left = NULL,
                          TreeNode *right = NULL) :
                          item(nodeItem), leftChildPtr(left),
                          rightChildPtr(right){ }
  TreeItemType item;                    // data portion
  TreeNode *leftChildPtr;               // pointer to left child
  TreeNode *rightChildPtr;              // pointer to right child

  friend class BinaryTree;              //  friend class

}; // end TreeNode class
```

3. Parent-Pointer Implementation of Tree:

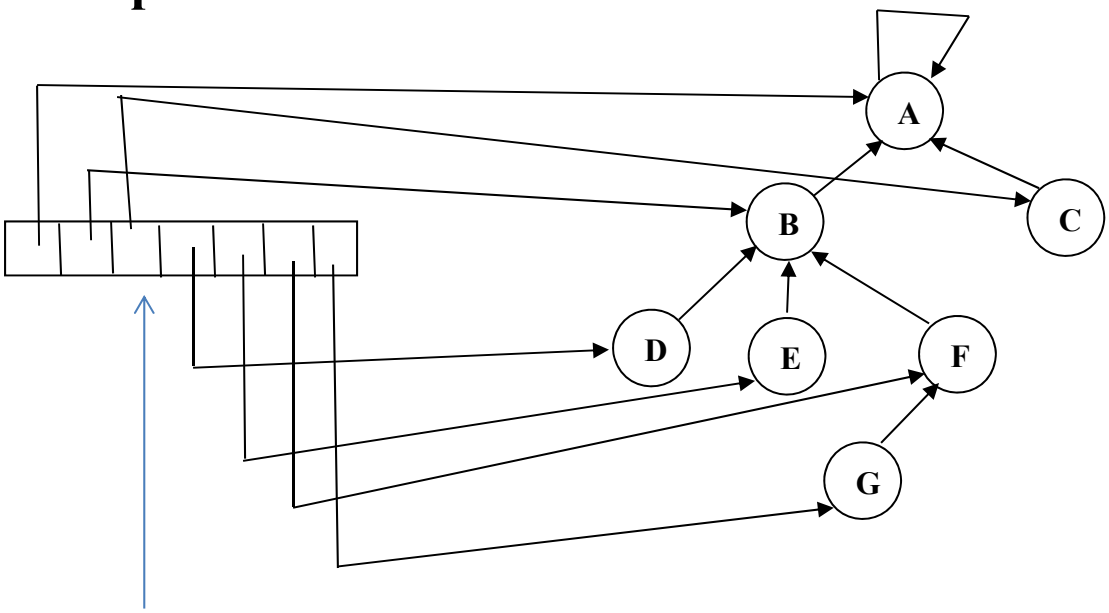
Observe that every node in a tree can have at most one parent. Hence, we can implement any tree by using the following hybrid data structure.

Root的parent是自己

NodeType:



Example:



auxiliary array (to access elements in T)

Remark: The array of pointers will be used to access the nodes of the tree.

General Tree Implementations:

1. k-tree implementation, k ≥ 2:

(a) Array-based Implementation of (Complete) k-Tree:

Given a complete k-tree T with n nodes. T can be implemented using an array A[0:n–1] such that

(1) Root of T at A[0],

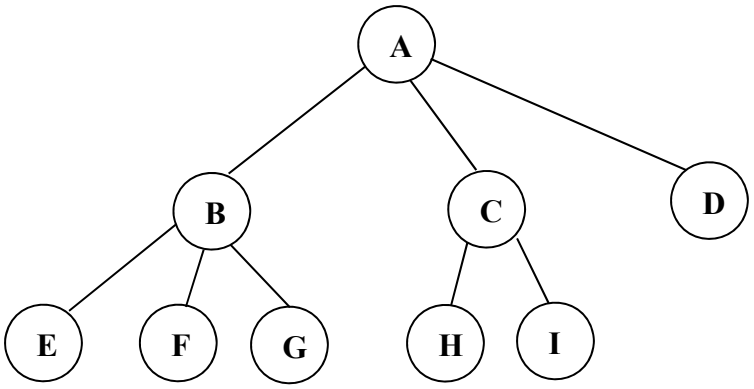
(2) Parent of a node A[i] at A[(i–1)/k] if exists,

(3) The jth chid of a node A[i], 1 ≤ j ≤ k, at A[ki+j] if exist.

Observe that, for n ≥ 1, A[i] is a leaf iff ki ≥ n–1.

J代表第几个child；

Example:



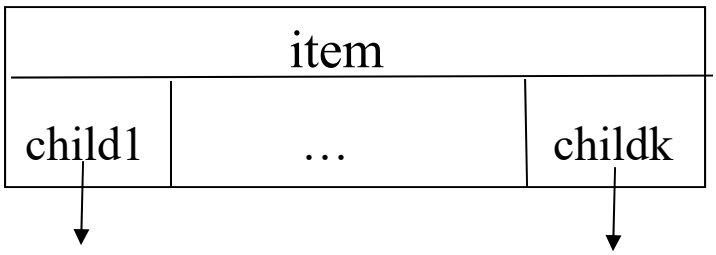
last = 8

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

(b) Children-Pointer Implementation:

Same as before except using k children pointers.

NodeType:



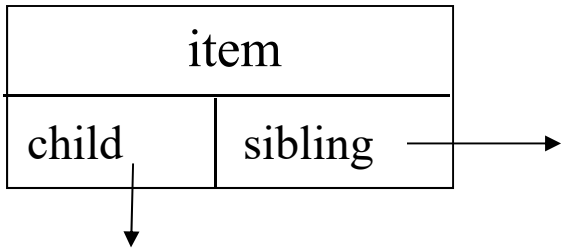
2. Parent-Pointer Implementation for Tree:

Same as above.

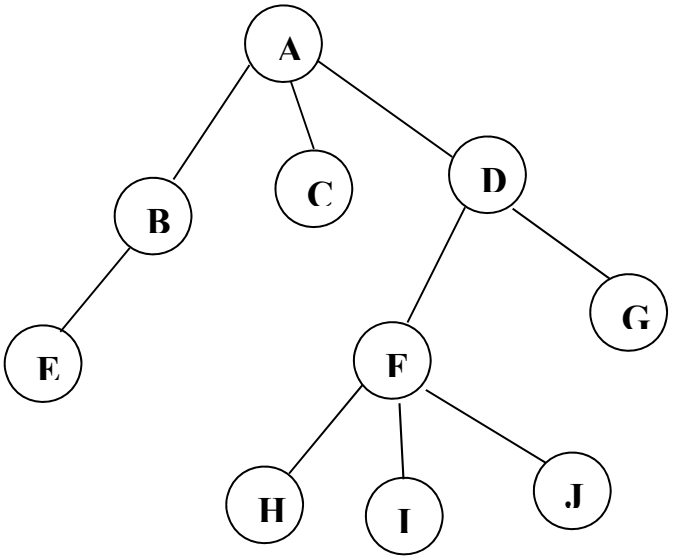
3. Left-Child List-of-Siblings Implementation for Tree:

For each node N in T, the leftmost child of N, say x, will become the **only** child of N and the siblings of x will be linked together to form a chain of siblings of x.

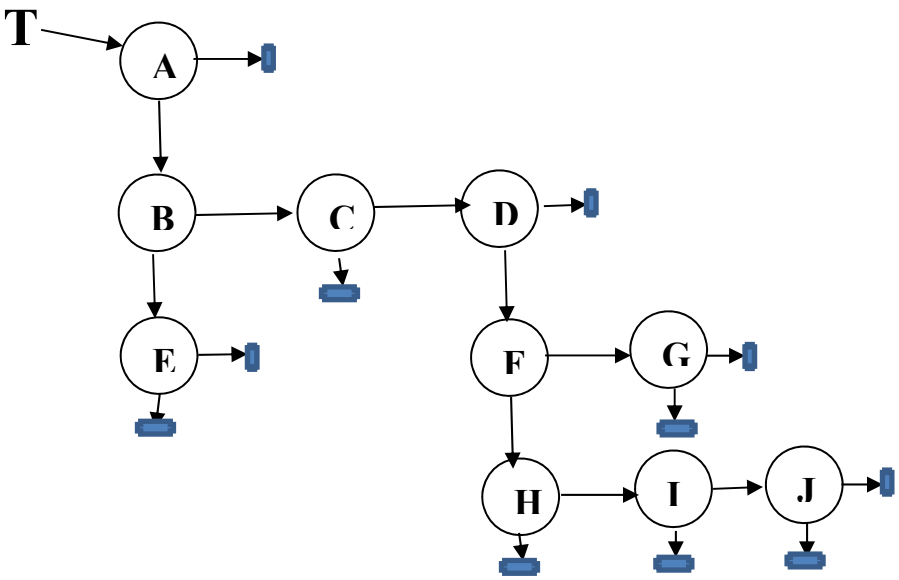
NodeType:



Example:



Left-Child List-of-Siblings Implementation:



Remark: Based on applications, siblings can be linked together using a suitable linked structure such as singly, doubly, or circular doubly linked list.