Ch.12 BINARY and OTHER TREES

BIRD'S-EYE VIEW (0)

- Chapter 12: Binary Tree
- Chapter 13: Priority Queue
  - Heap and Leftiest Tree
- Chapter 14: Tournament Trees
  - Winner Tree and Loser Tree

BIRD'S-EYE VIEW

- Can obtain improved run-time performance by using trees to represent the sets
- Tree and binary tree terminology
  - Height, Depth, Level
  - Root, Leaf
  - Child, Parent, Sibling
- Representation of binary trees
  - Array-based
  - Linked
- The four common ways to traverse a binary tree
  - Preorder
  - Inorder
  - Postorder
  - Level-order

Table of Contents

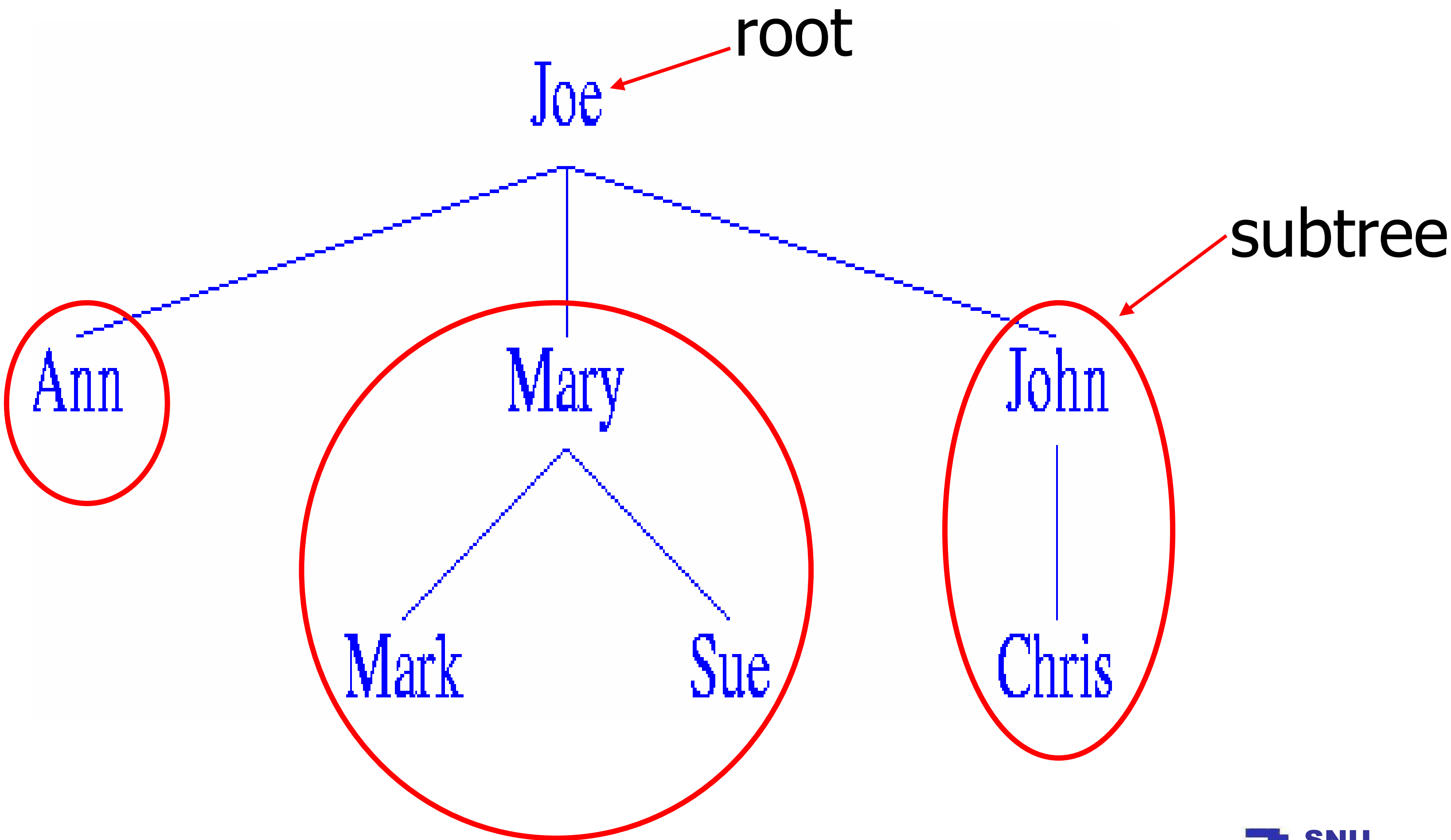
- Trees
- Binary Tree
  - Properties of Binary Trees
  - Representation of Binary Trees
  - Common Binary Tree Operations
  - Binary Tree Traversal
  - ADT Binary Tree
- Tree Applications

Trees (1)

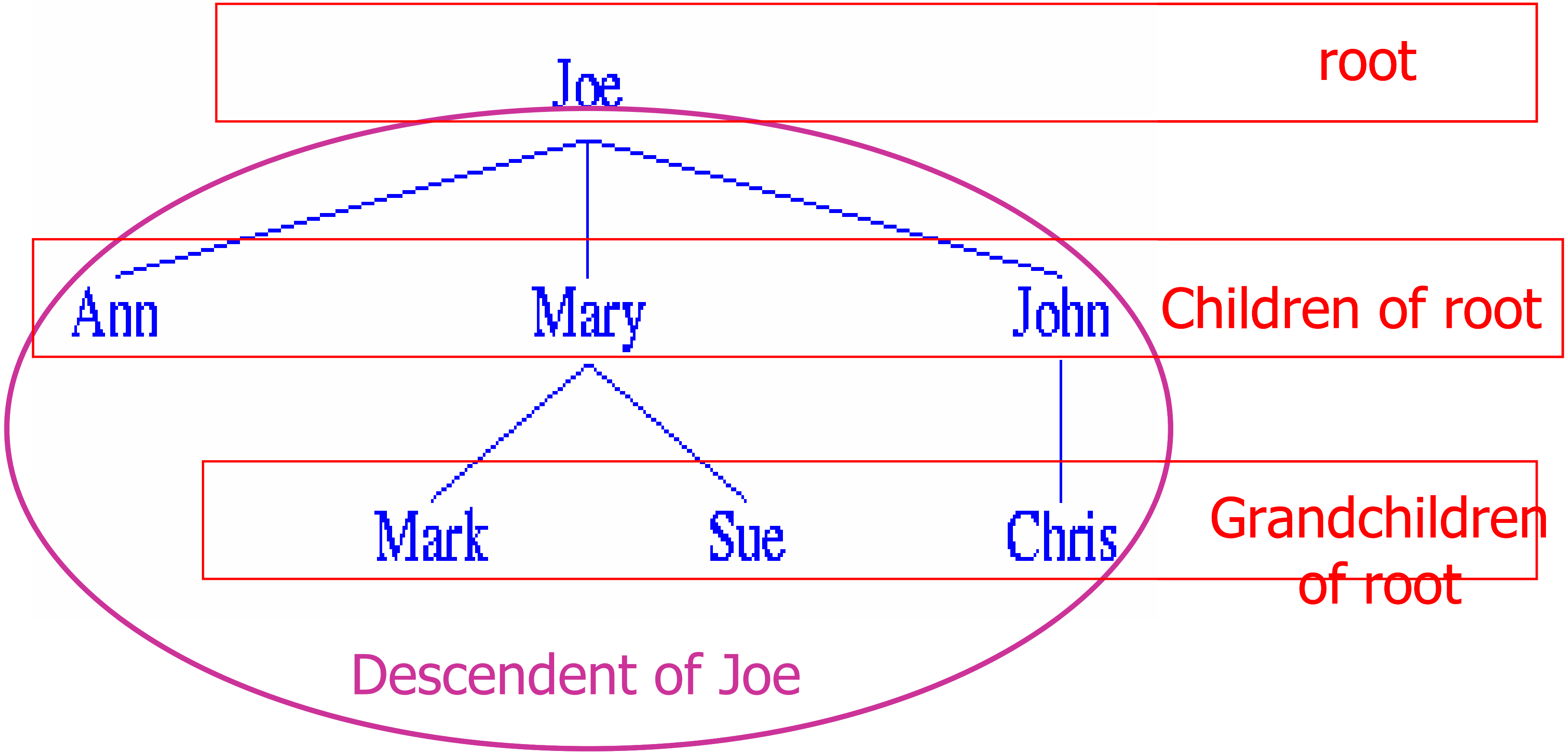
- A tree consists of finite nonempty set of elements
  - One of elements is called the **root**
  - Remaining elements are partitioned into trees, which are called the **subtrees**

- Terminology
  - Children, Grand Children, Descendent
  - Sibling
  - Parent, Grand Parent, Ancestor
  - Leaves
  - Level, Height, Degree

Trees (2)

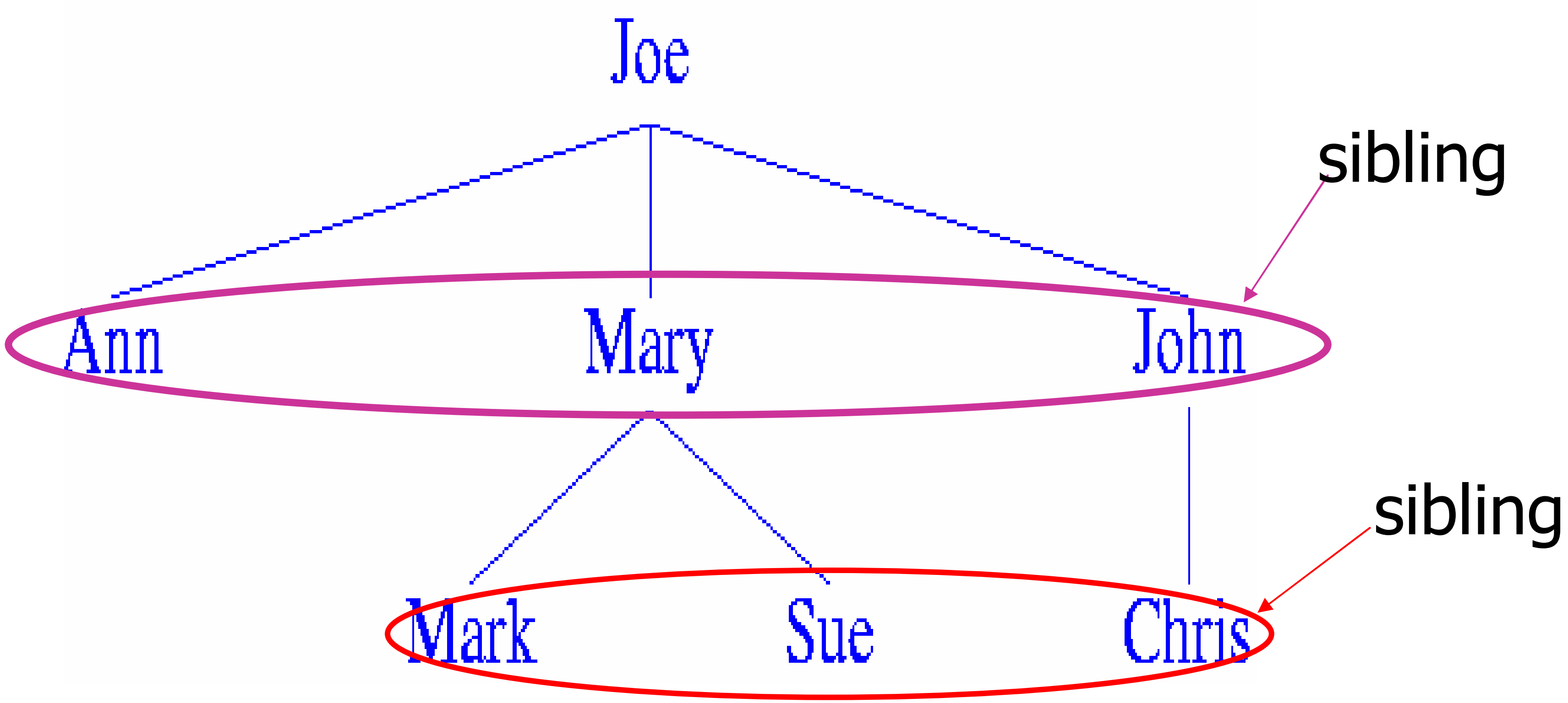


Trees (3)

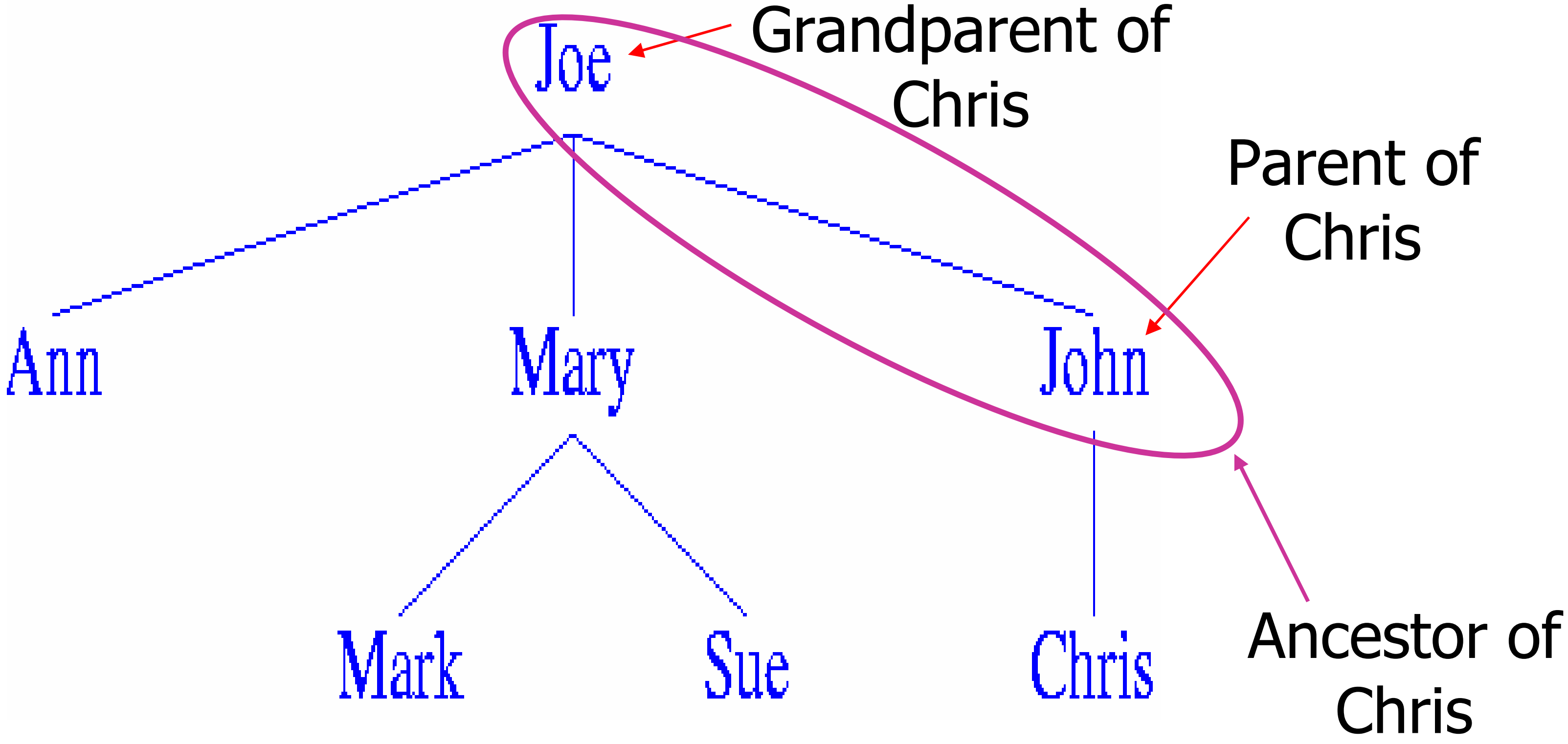


Joe
root
Ann
Mary
John
Children of root
Mark
Sue
Chris
Grandchildren of root
Descendent of Joe

Trees (4)

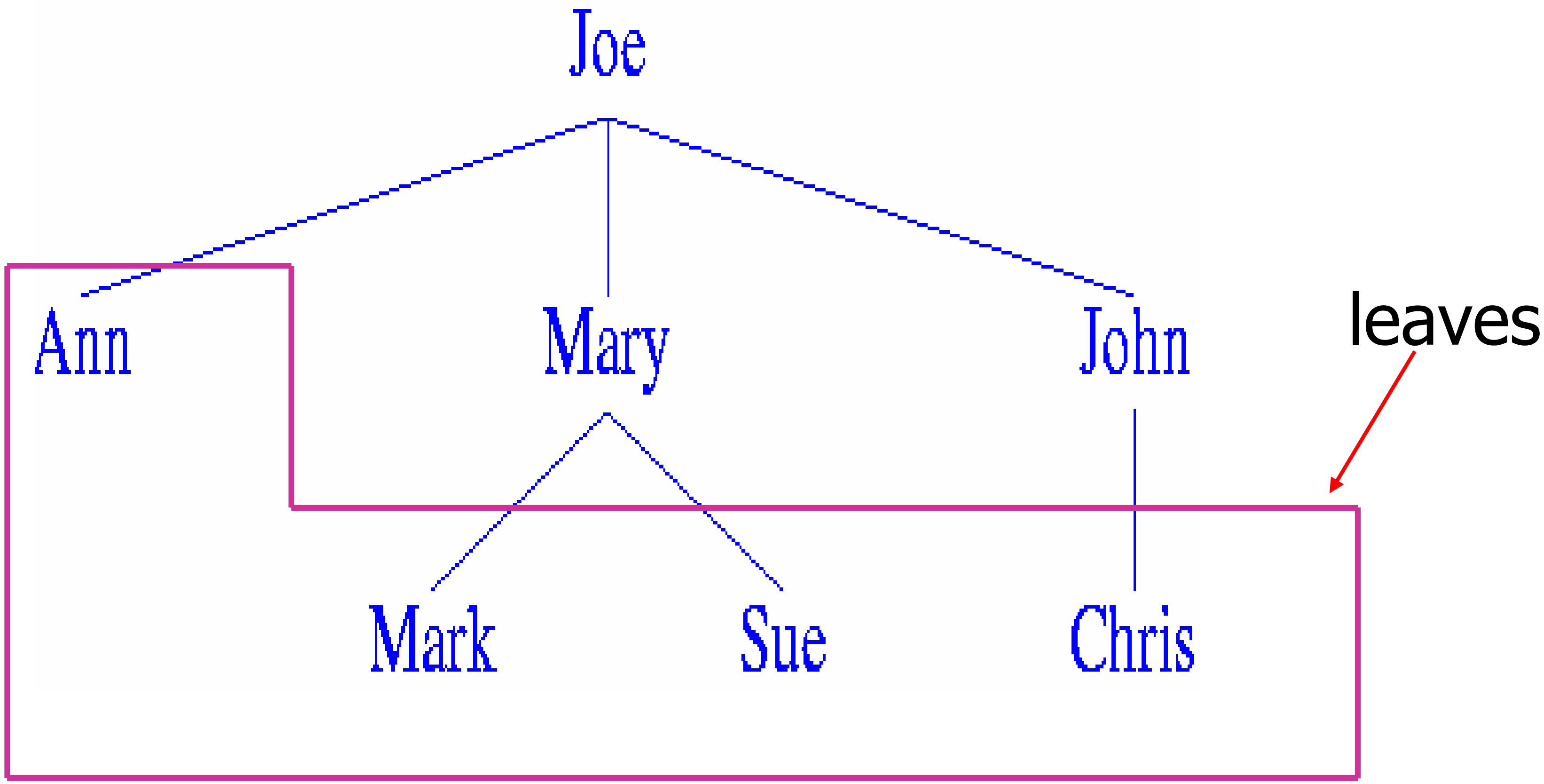


Trees (5)

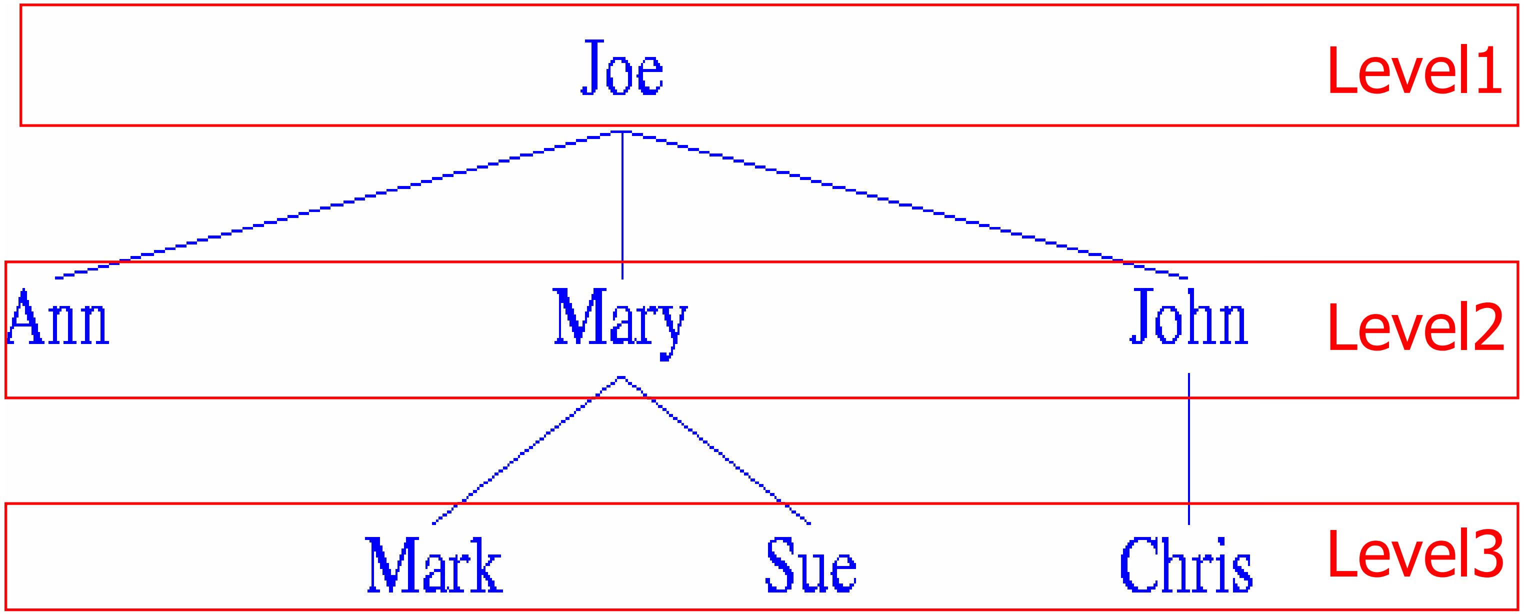


Joe
Ann
Mary
John
Mark
Sue
Chris
Grandparent of Chris
Parent of Chris
Ancestor of Chris

Trees (6)



Trees (7)



- Height = the number of levels = 3

Trees (8)

- Degree of an element
  - The number of children of an element
- Degree of a tree
  - The maximum of its element degrees

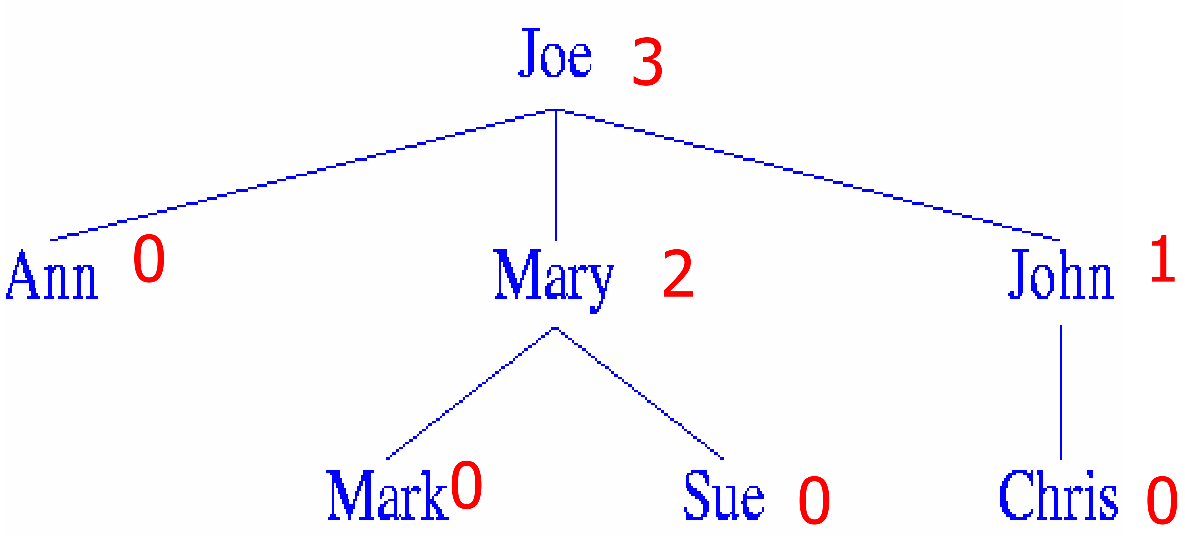


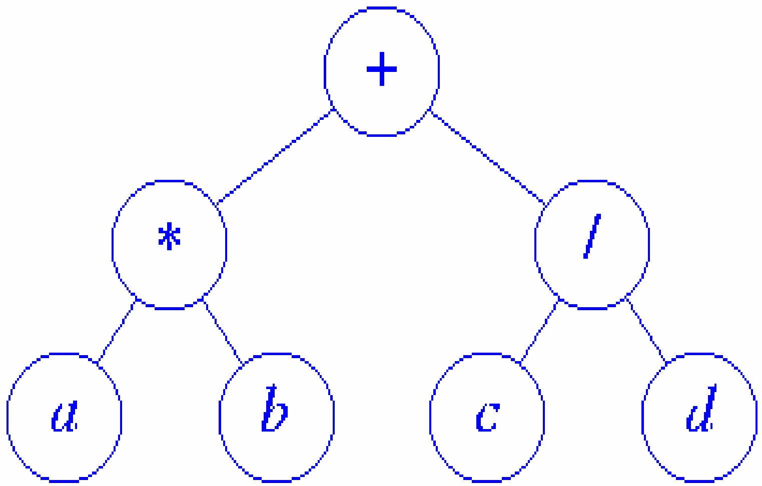
Table of Contents

- Trees
- Binary Trees
  - Properties of Binary Trees
  - Representation of Binary Trees
  - Common Binary Tree Operations
  - Binary Tree Traversal
  - ADT BinaryTree
- Tree Applications

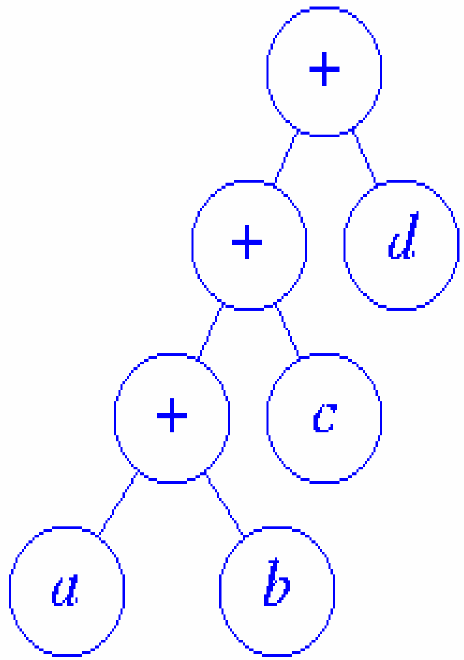
Binary Trees

- A Tree has any number of subtrees which are unordered
- A binary tree has two subtrees which are ordered
  - the left subtree and the right subtree
  - The subtrees are also binary tree
- Full Binary Tree
- Complete Binary Tree

Binary Trees for Arithmetic Expression



(a) (a * b) + (c / d)



(b) ((a + b) + c) + d

BT Property 12.1 & 12.2

- [12.1] The drawing of every binary tree with n elements, n > 0, has exactly n - 1 edges
- Proof
  - Every element in the binary tree(except the root) has exactly one parent
  - One edge between each child and its parent
  - So the number of edges is n-1

- [12.2] A binary tree of height, h >= 0, has at least h and at most $2^h - 1$ elements in it
- Proof
  - Since each level has one element, the number of elements is h
  - Since each element has two children, the number of elements is $2^h - 1$

BT Property 12.3

- [12.3] The height of a binary tree that contains n, n >= 0, elements is at most n and at least $\lceil \log_2(n+1) \rceil$
- Proof
  - Since there must be at least one element at each level, the height cannot exceed n
  - From property 12.2, a binary tree of height h can have no more than $2^h - 1$ elements
  - So $n \le 2^h - 1$.
  - Hence $h \ge \log_2(n+1)$
  - Since h is an integer, $h \ge \lceil \log_2(n+1) \rceil$

Full Binary tree

- FBT of height h that contains exactly $2^h - 1$ elements

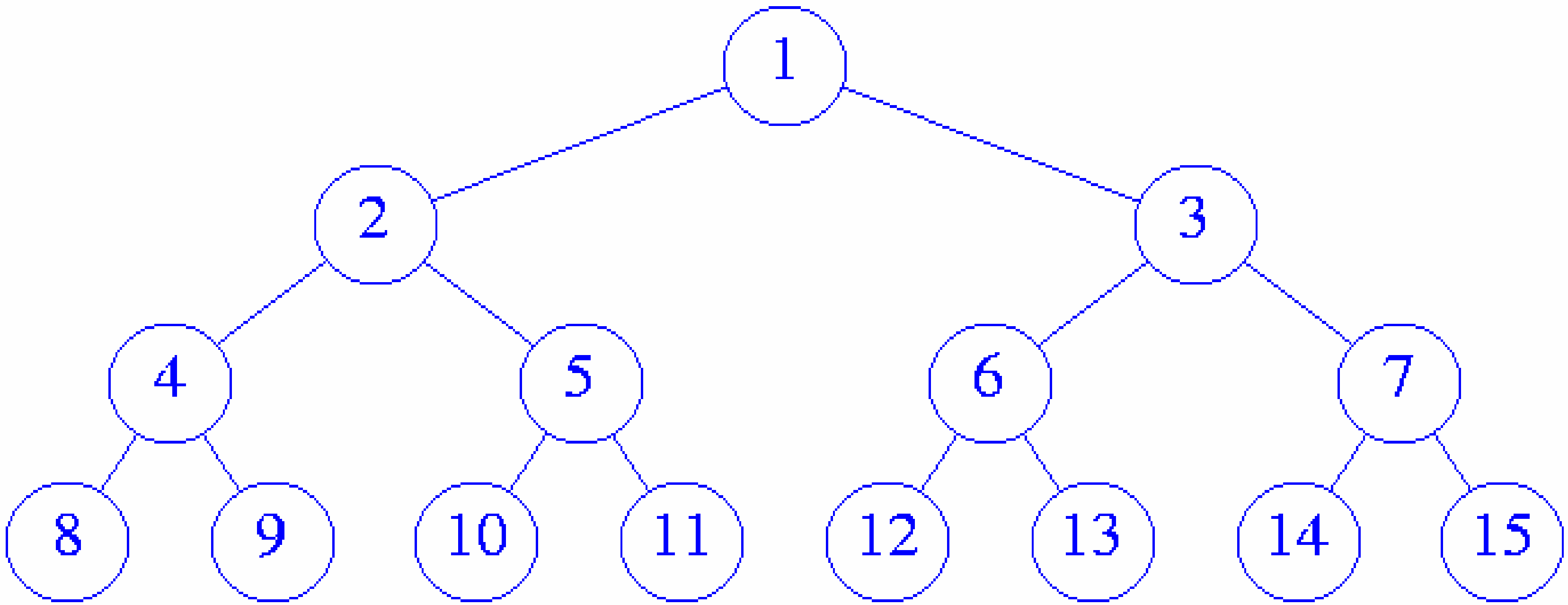


Figure 12.6 Full binary tree of height 4

Complete Binary Tree

- All resulting trees by deleting the k elements numbered $2^h - i$, $1 \le i \le k < 2^h$ from a full BT
  - $2^h - 1$, $2^h - 2$, $2^h - 3$,….. ➔ 8 – 1 , 8 – 2, 8 – 3, ➔ 7, 6, 5, ….

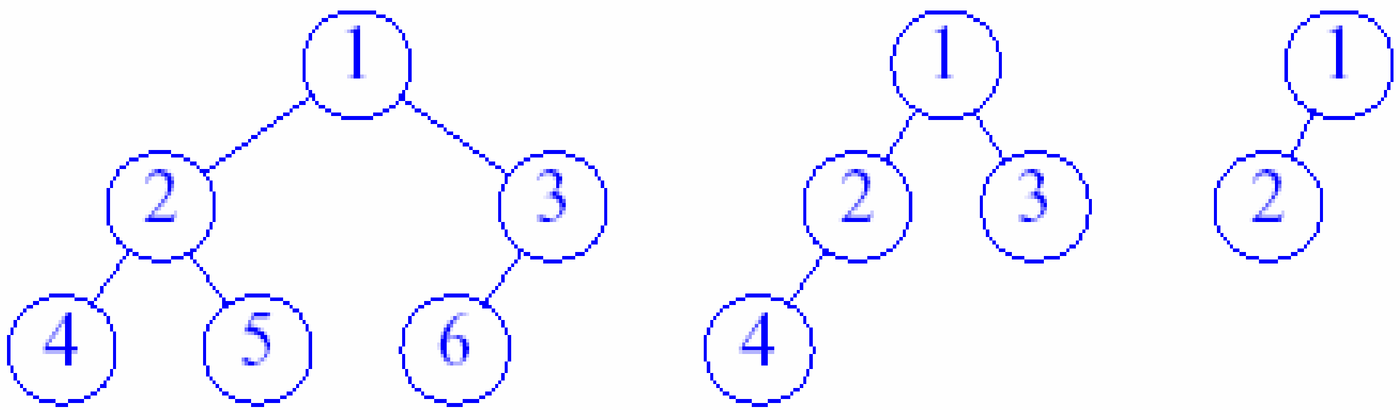


Figure 12.7 Complete binary trees

BT Property 12.4

- [12.4] Let i, 1 <= i <= n, be the number assigned to an element of a complete binary tree.

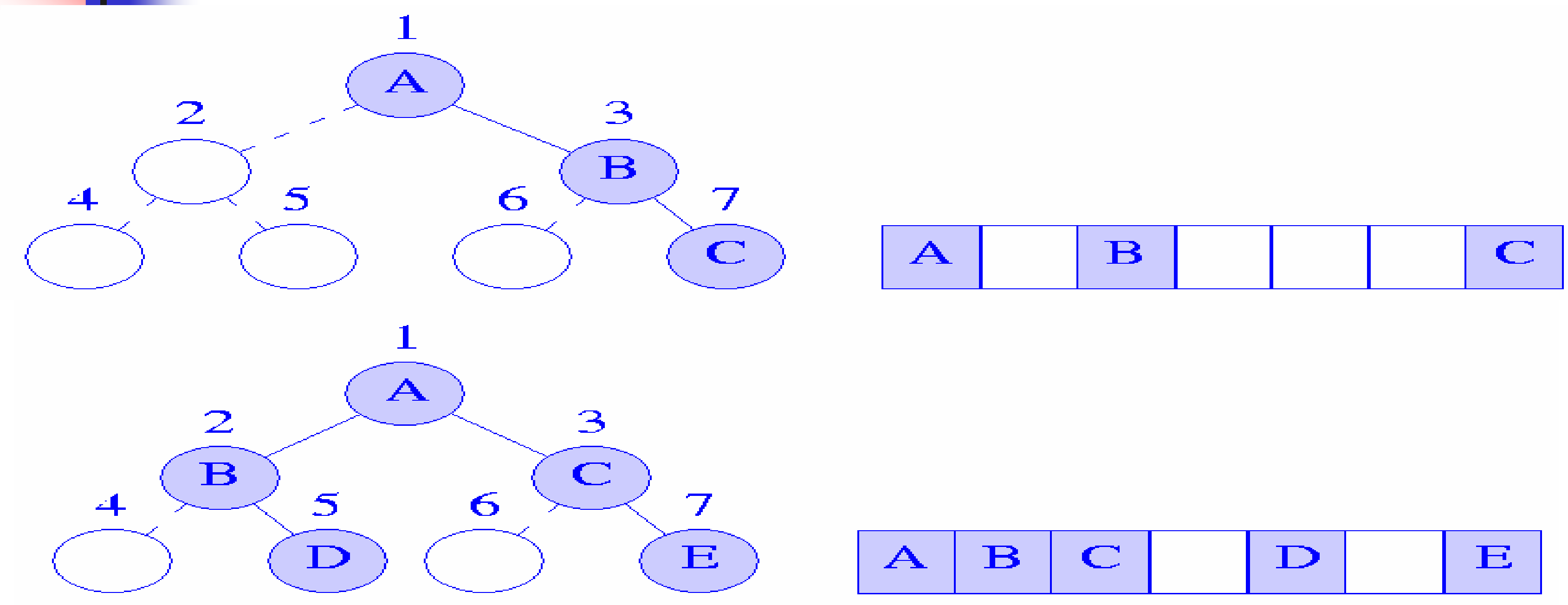
The followings are true:

- If i = 1, then element is the root of the binary tree.
- If i > 1, then the parent of this element has been assigned the number $\lfloor i/2 \rfloor$
- If i > n/2, then this element has no left child. Otherwise (i <= n/2), its left child has been assigned the number 2i
- If i > (n-1)/2, then this element has no right child. Otherwise (i <= (n-1)/2), its right child has been assigned the number 2i+1

- See Figure 12.7

Table of Contents

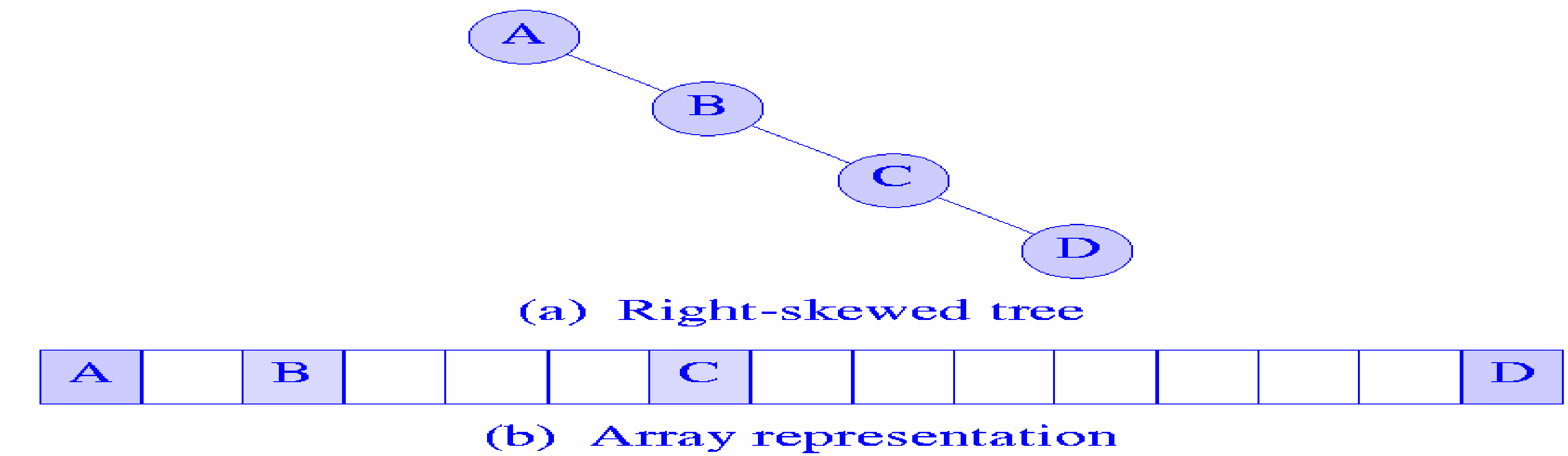
Array Based Binary Tree (1)



<Binary tree> <Array representation>

- The location of an element is calculated by the array index
  - Useful only when the number of missing elements is small

Array-Based Binary Tree (2)

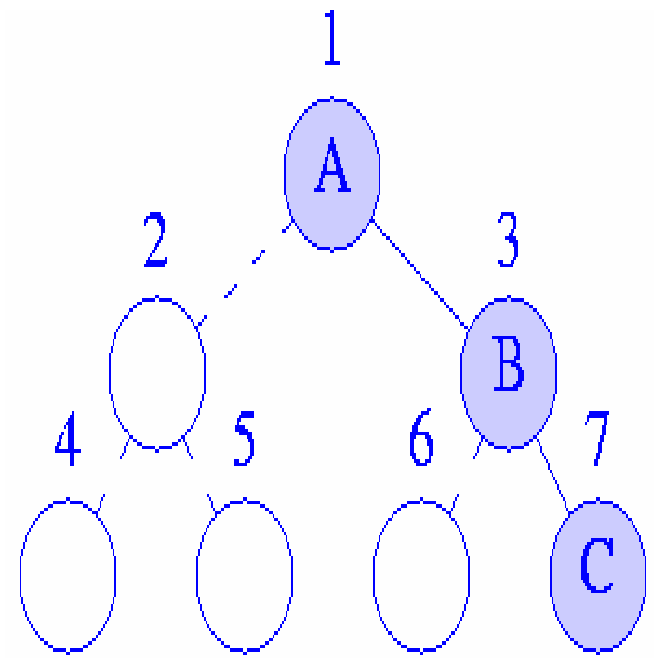


- Pitfalls
  - quite wasteful of space when many elements are missing in some cases like a right-skewed binary tree
  - A right-skewed binary tree that has n elements may require an array of size up to $2^n - 1$ for its representation

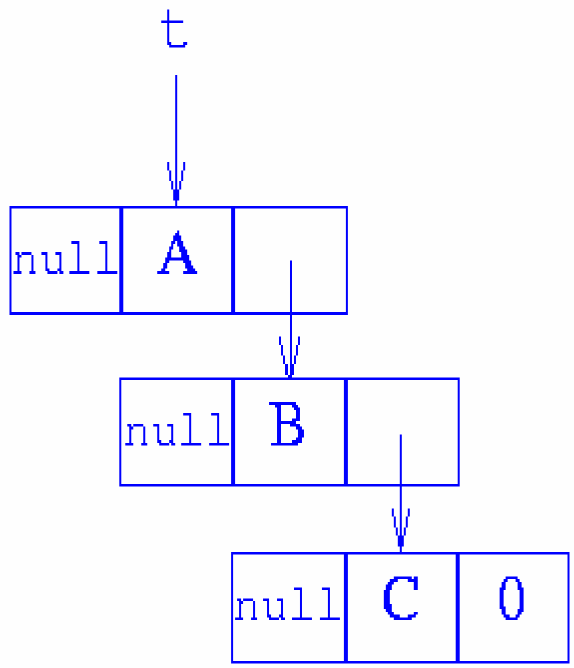
Linked Binary Tree (1)

- Two link fields represents each element
  - leftChild
  - rightChild
- Field-name: element
- Access all nodes in a binary tree by starting at the root and following leftChild and rightChild links recursively
  - inOrder
  - preOrder
  - postOrder
  - levelOrder

Linked Binary Tree (2)

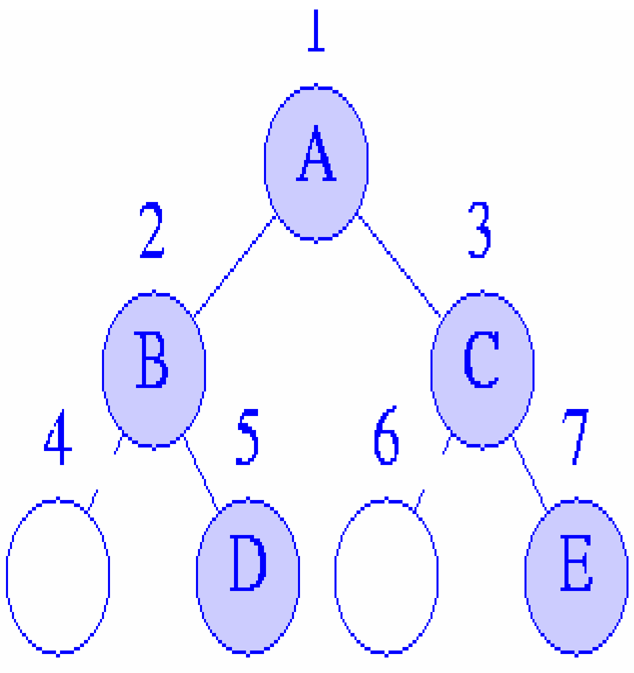


<Binary tree>

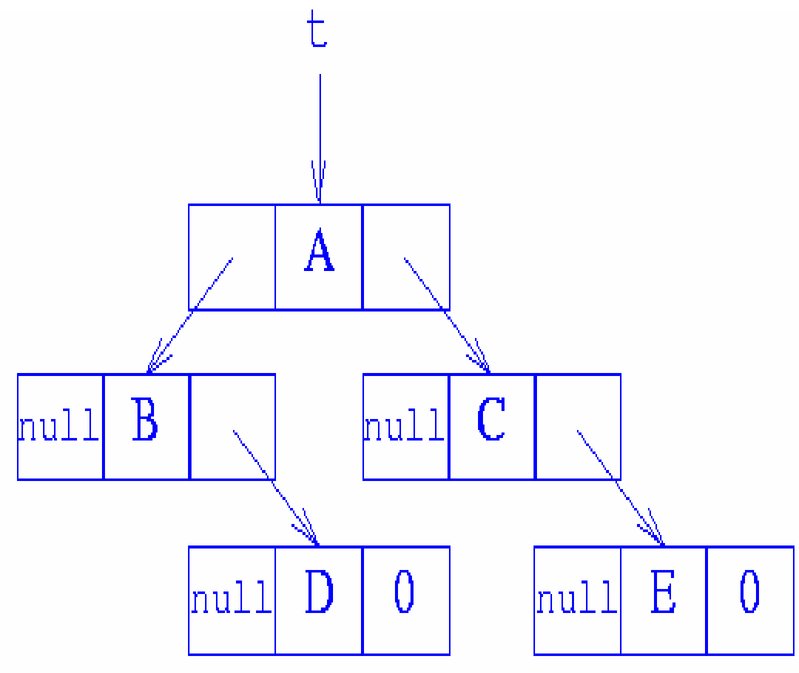


<Linked representation>

Linked Binary Tree (3)



<Binary tree>



<Linked representation>

Linked Binary Tree (4)

```
public class BinaryTreeNode {
 // package visible data members
Object element;
BinaryTreeNode leftChild;    // left subtree
BinaryTreeNode rightChild;   // right subtree
// constructors
public BinaryTreeNode () {}
public BinaryTreeNode (Object theElement)   {element = theElement;}
public BinaryTreeNode
    (Object theElement, BinaryTreeNode theleftChild, BinaryTreeNode therightChild)
     { element = theElement;
       leftChild = theleftChild;
       rightChild = therightChild;
      }
}
```

Table of Contents

- Trees
- Binary Trees
  - Properties of Binary Trees
  - Representation of Binary Trees
  - Traversal-based Operations in BT
  - ADT BinaryTree
- Tree Applications

Traversal-based Operations in BT

- These operations are performed by traversing the BT in a systematic manner
  - Determine its height of BT
  - Determine the number of elements in a BT
  - Make a copy of BT
  - Display the binary tree on a screen or on paper
  - Determine whether two binary trees are identical
  - Make the tree empty

Binary Tree Traversals

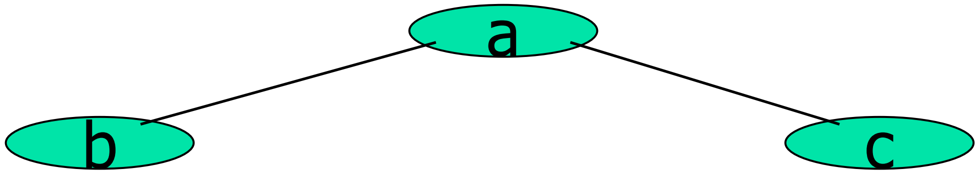
- In a BT traversal, each element is visited exactly once
  - Preorder traversal (depth first search)
  - Inorder traversal
  - Postorder traversal
  - Level-order traversal (breadth first search)

- When an expression tree is output in preorder, inorder, and postorder, we get the prefix, infix, and postfix forms of the expression, respectively
  - Expression: A + B
    - Preorder tarversal ➔ Prefix form: + A B
    - Inorder traversal ➔ Infix form: A + B
    - Postorder traversal ➔ Postfix form: A B +

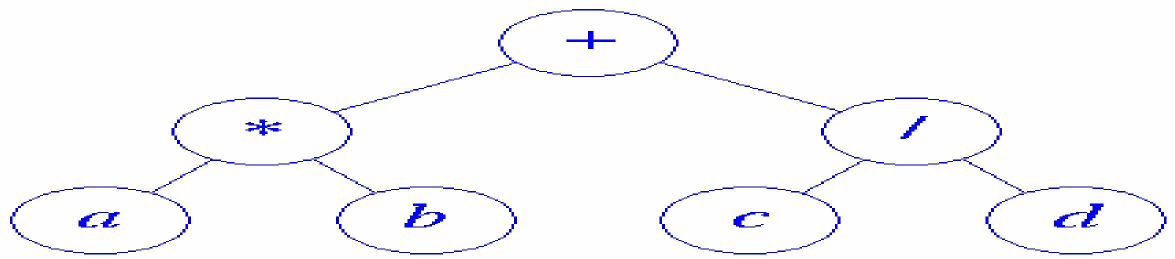
Preorder BT traversal (1)

```
public static void preOrder (BinaryTreeNode t) {
    if (t != null)
    {
        visit(t);                  //visit tree root
        preOrder(t.leftChild);     //do left subtree
        preOrder(t.rightChild);    //do right subtree
    }
}
```

Preorder BT traversal (2)

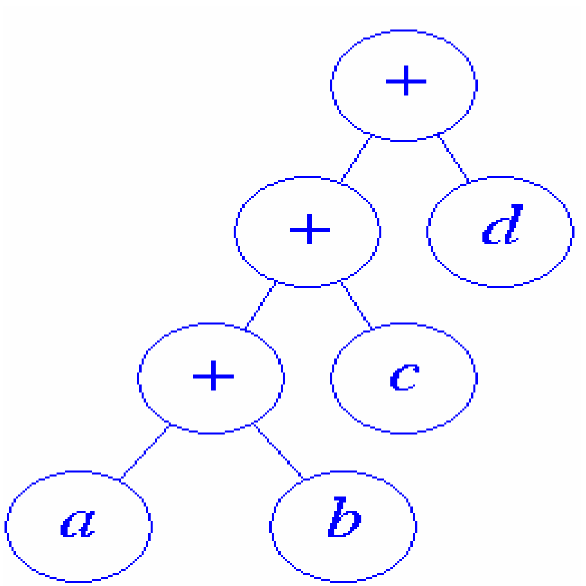


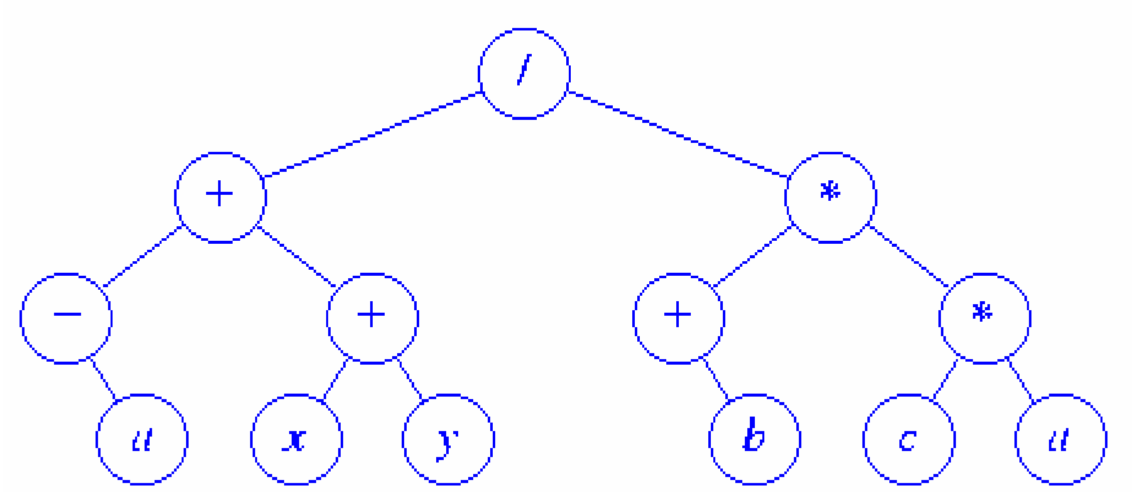
a b c



+ * a b / c d

Preorder BT traversal (3)





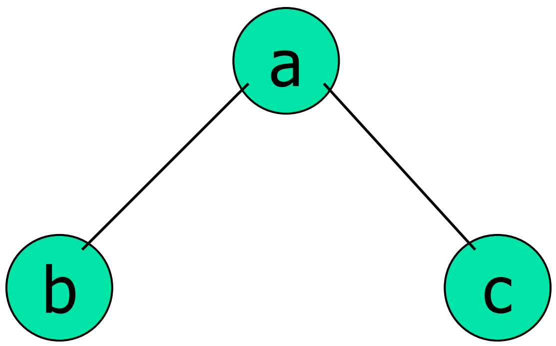
+ + + a b c d

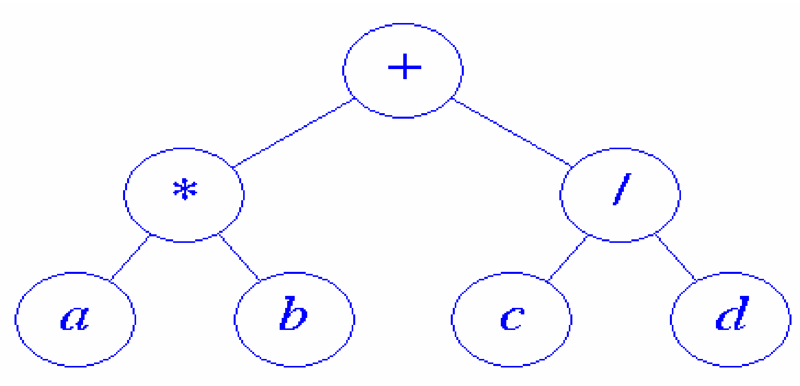
/ + - a + x y * + b * c d

Inorder BT traversal (1)

```
public static void inOrder (BinaryTreeNode t) {
   if (t != null)
   {
        inOrder(t.leftChild);        //do left subtree
        visit(t);                    //visit tree root
        inOrder(t.rightChild);       //do right subtree
   }
}
```

Inorder BT traversal (2)

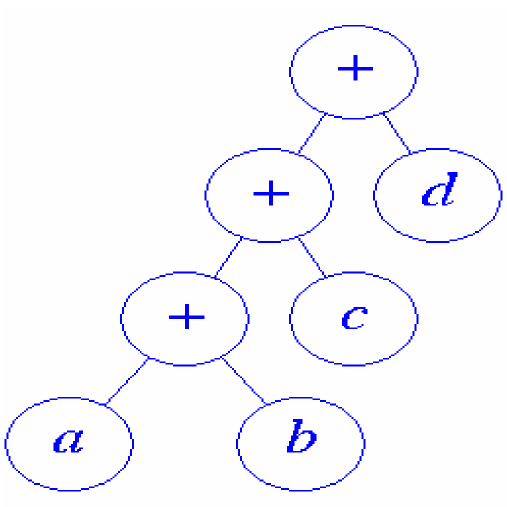


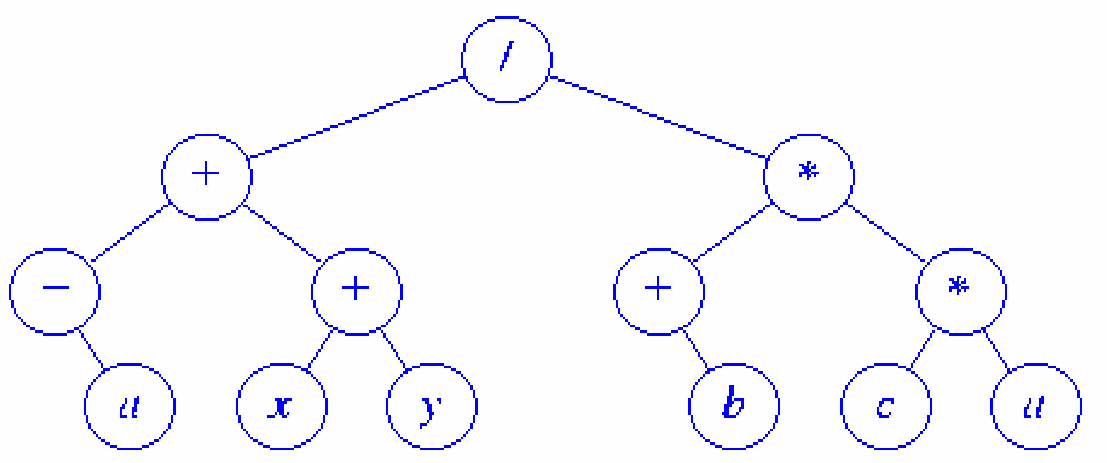


b a c

a * b + c / d

Inorder BT traversal (3)





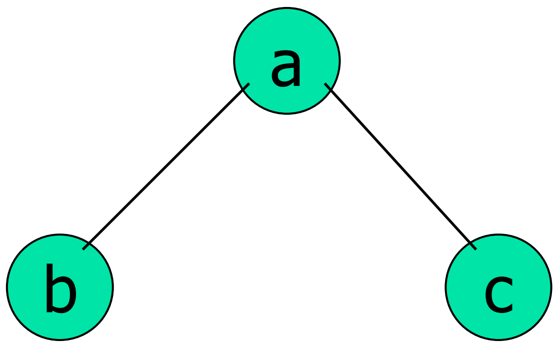
a + b + c + d

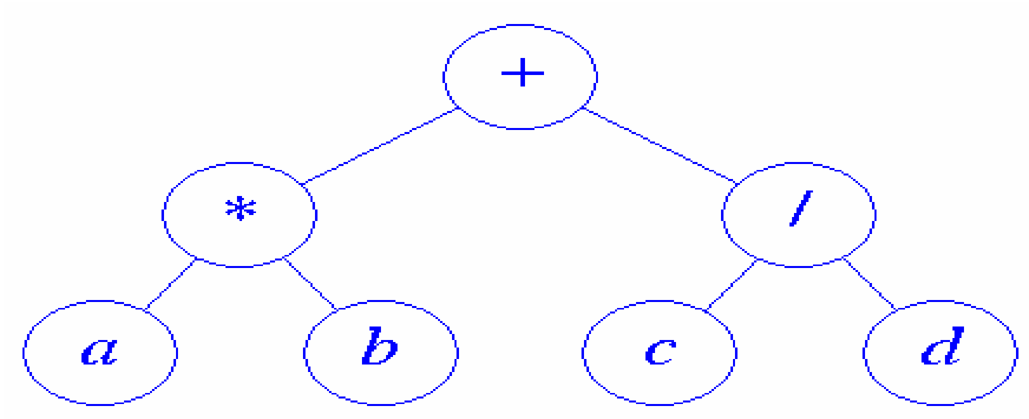
- a + x + y / + b * c * a

Postorder BT traversal (1)

```
public static void postOrder (BinaryTreeNode t) {
    if (t != null)
    {
        postOrder(t.leftChild);       //do left subtree
        postOrder(t.rightChild);      //do right subtree
        visit(t);                     //visit tree root
    }
}
```

Postorder BT traversal (2)

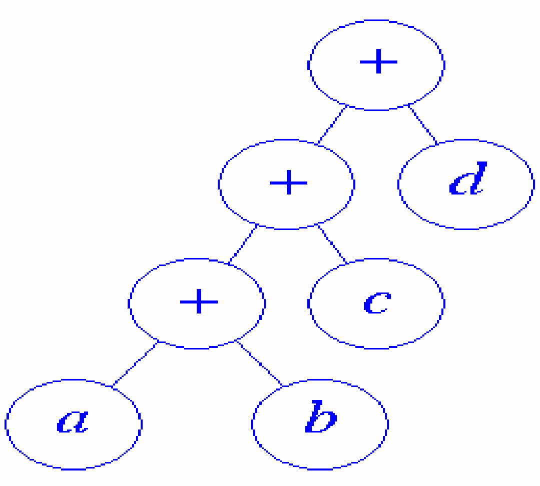


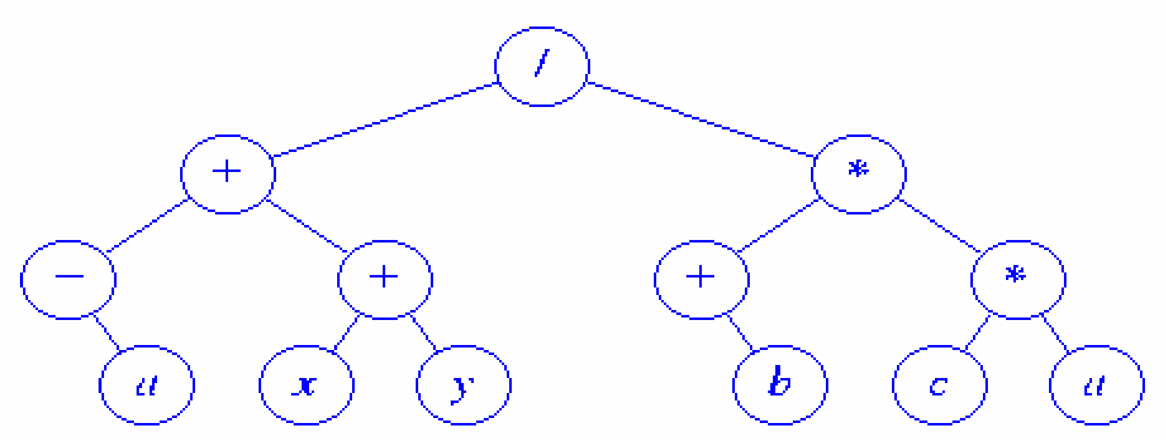


b c a

a b * c d / +

Postorder BT traversal (3)





a b + c + d +

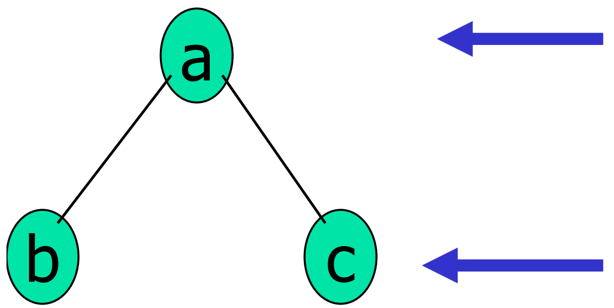
a − x y + + b + c d * * /

Level-order BT traversal (1)

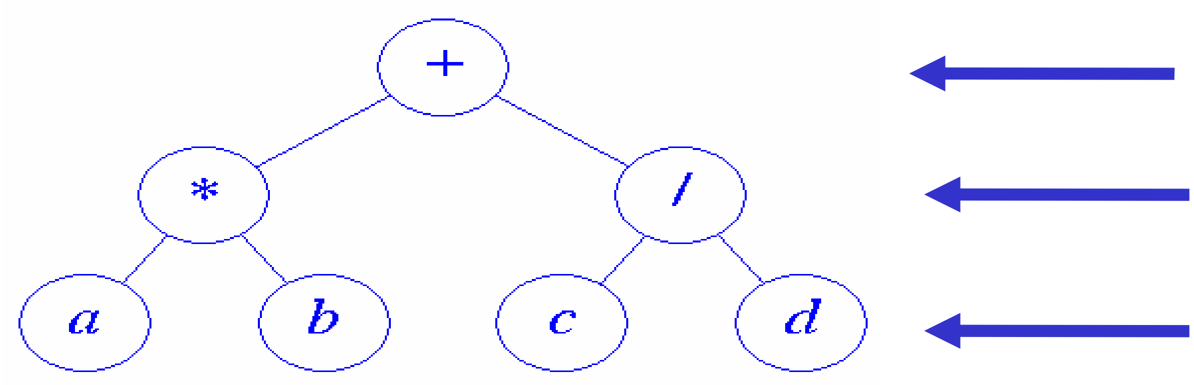
- Visit by level from top to bottom
- Within levels, elements are visited from left to right.

```
public static void levelOrder (BinaryTreeNode t) {
    ArrayQueue q = new ArrayQueue();  // need array queue
    while (t != null) {
        visit(t);   // visit t
        // put t's children on queue
        if (t.leftChild != null)    q.put(t.leftChild);
        if (t.rightChild != null)  q.put(t.rightChild);
      // get next node to visit
        t =  (BinaryTreeNode)q.remove();
     }
}
```

Level-order BT traversal (2)

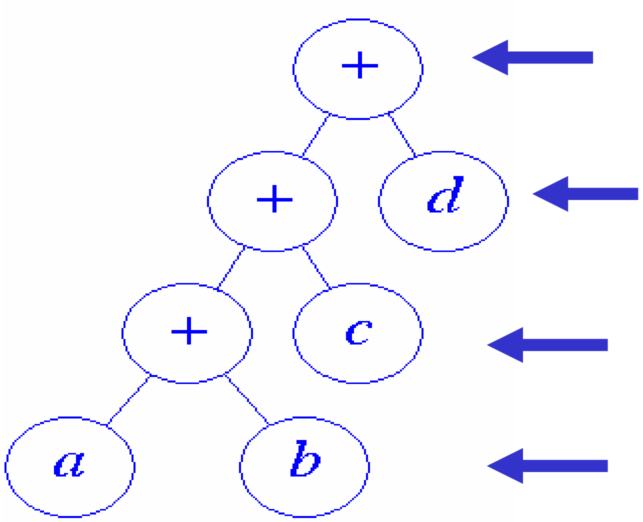


a b c

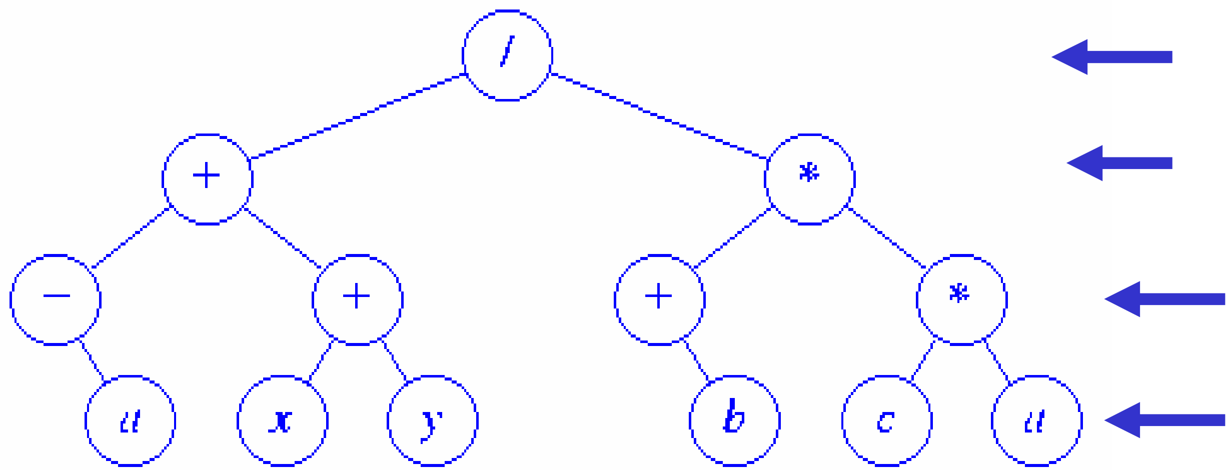


+ * / a b c d

Level-order BT traversal (3)



+ + d + c a b



/ + * - + + * a x y b c a

Table of Contents

- Trees
- Binary Trees
  - Properties of Binary Trees
  - Representation of Binary Trees
  - Traversal-based Operations in BT
  - ADT BinaryTree
- Tree Applications

The ADT BinaryTree

AbstractDataType BinaryTree {
// instances
collection of elements;
if not empty, the collection is partitioned into a root, left subtree, and right subtree;
each subtree is also a binary tree;
// operations
isEmpty() : return true if empty, return false otherwise;
root() : return the root element; return null if the tree is empty;
makeTree(root, left, right) : create a binary tree with root, left subtree, right subtree
removeLeftSubtree() : remove the left subtree and return it;
removeRightSubtree() : remove the right subtree and return it;
preOrder(visit) : preorder traversal of binary tree
inOrder(visit) : inorder traversal of binary tree
postOrder(visit) : postorder traversal of binary tree
levelOrder(visit) : level-order traversal of binary tree
}

Data Members in LinkedBinaryTree

```
// instance data member
BinaryTreeNode root; // root node
// class data members
Static Method visit;
Static Object [] visitArgs = new Object [1]
Static int count;
Static Class [] paramType = {BinaryTreeNode.class}
Static Method the Add1;
Static Method theOutput;
// method to initialize class data members
Static {
  try {
    class lbt = LinkedBinaryTree.class;
    theAdd1 = lbt.getMethod(“add1”,paramType);
    theOutput = lbt.getMethod(“output”, paramType);
  }
  catch (Exception e) {} // exception not possible
}
```

Visit methods in LinkedBinaryTree

```
// only default constructor available

// class methods
/** visit method that outputs element
public static void output (BinaryTreeNode t) {System.out.print(t.element + " ");}

/** visit method to count nodes */
public static void add1 (BinaryTreeNode t) {count++;}
```

makeTree() & removeLeftSubtree() methods of LinkedBinaryTree

```
public void makeTree (Object root, Object left, Object right)  {
This.root = new BinaryTreeNode(root, ((LinkedBinaryTree) left).root,
    ((LinkedBinaryTree) right).roor);
}

public BinaryTree removeLeftSubtree ( ) {
If(root==null) throw new illegalArgumentException("tree is empty");
// detach left subtree and save in leftSubtree
LinkedBinaryTree leftSubtree = new LinkedBinaryTree();
leftSubtree.root = root.leftChild;
Root.leftChild = null;
Return (BinaryTree) leftSubtree;
}
```

Preorder Methods of LinkedBinaryTree

```
public void preOrder (Method visit) {
    this.visit = visit;
    thePreOrder(root);
}
Static void thePreOrder (BianryTreeNode t) {
    if (t != null)  {
      visitArgs[0] = t;
      try { visit.invoke(null, visitArgs); };
      catch (Exception e) { System.out.println(e) };
      thePreOrder (t.leftChild);
      thePreOrder (t.rightChild);
   }
}

public void preOrderOutput() { preOrder (theOutput); }
```

Table of Contents

- Trees
- Binary Trees
  - Properties of Binary Trees
  - Representation of Binary Trees
  - Common Binary Tree Operations
  - Binary Tree Traversal
  - ADT BinaryTree
- Tree Applications
  - Placement of Signal Boosters (PSB)
  - Union-Find Problem (UFP)

Placement of Signal Booster

- Signal p → v degrades the signal strength by 5 because p → r → v
- Signal q → x degrades the signal strength by 3 because q → s → x

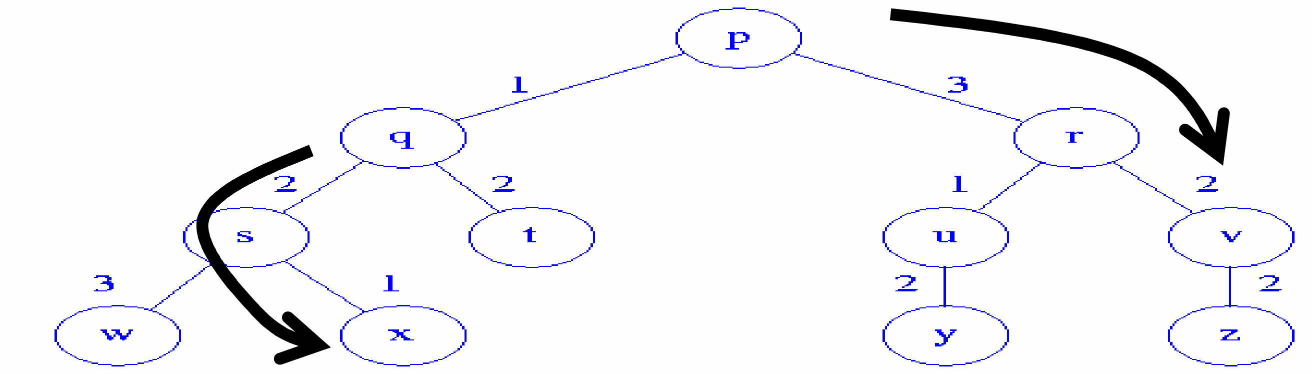


Figure 12.12 Tree distribution network

- If a signal booster is placed at node r which is a descendant of p
  - The strength of the signal that arrives at r is supposed to be 3 units less than that of the signal that leaves source p without the signal booster
  - But the signal that leaves r has the same strength as the signal that leaves source p

PSB Solution (1)

- degradeFromParent(i)
  - degradation between node i and its parent
  - The value of incoming edge of a node
    - degradeFromParent(w) = 3
    - degradeFromParent(p) = 0
    - degradeFromParent(r) = 3

- degradeToLeaf(i)
  - maximum signal degradation from node i to any leaf in the subtree rooted at i
  - If i is a leaf node, then degradeToLeaf(i) = 0
  - For the remaining nodes
    - degradeToLeaf(i) =
      max {degradeToLeaf(j) + degradeFromParent(j)}, j is a child of i
    - degradeToLeaf(s) = 3
    - degradeToLeaf(q) = 5

PSB Solution (2)

- Consider degradeToLeaf(q) when P → (1) q → (2) s → (3) w
  - degradeToLeaf(q) = degradeToLeaf(s) + degradeFromParent(s) = 5
  - Suppose tolerance = 3
  - Placing a booster at q or p does not help because it cannot tolerate signal degradation between q and its descendents
  - If a booster is placed at s, then degradeToLeaf(q) = 3

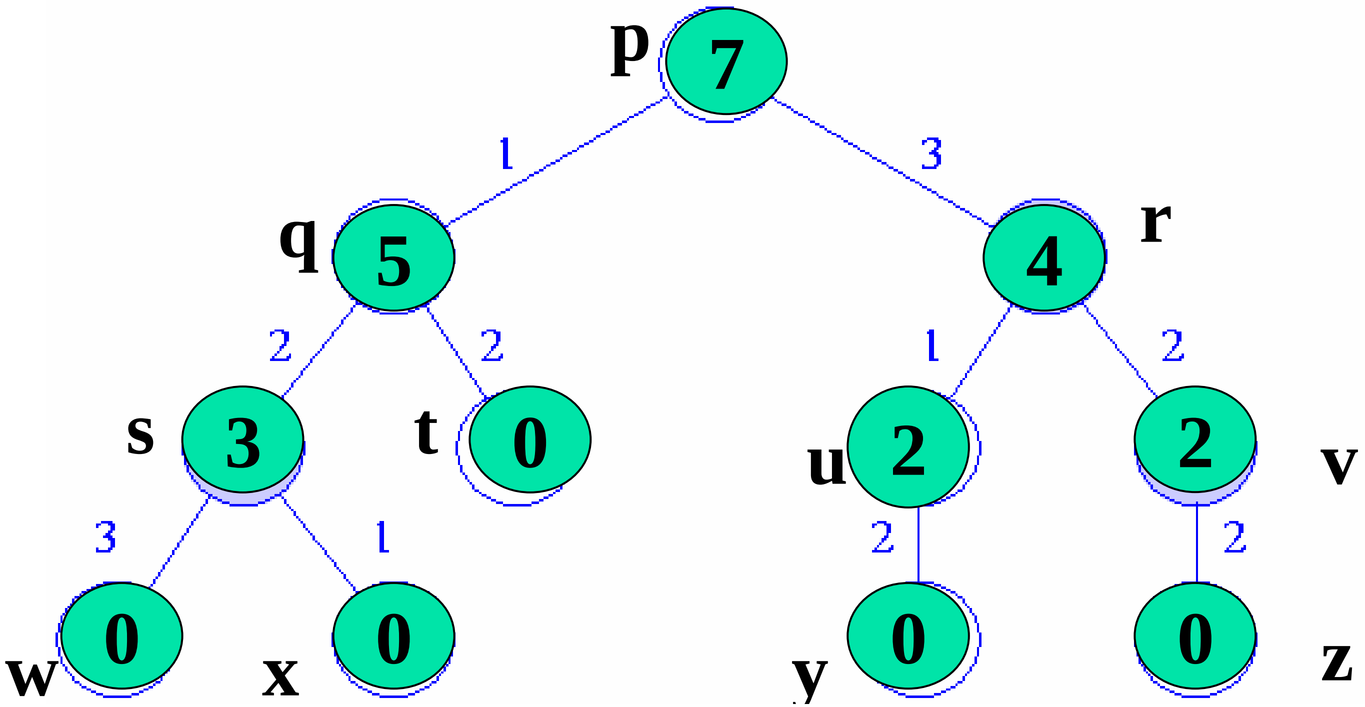
- From the root, check every path by node traversal
  - Sum degradeFromParent and degradeToLeaf of the nodes in the path
  - If the sum ≥ tolerance, put the booster there

PSB Solution (3)

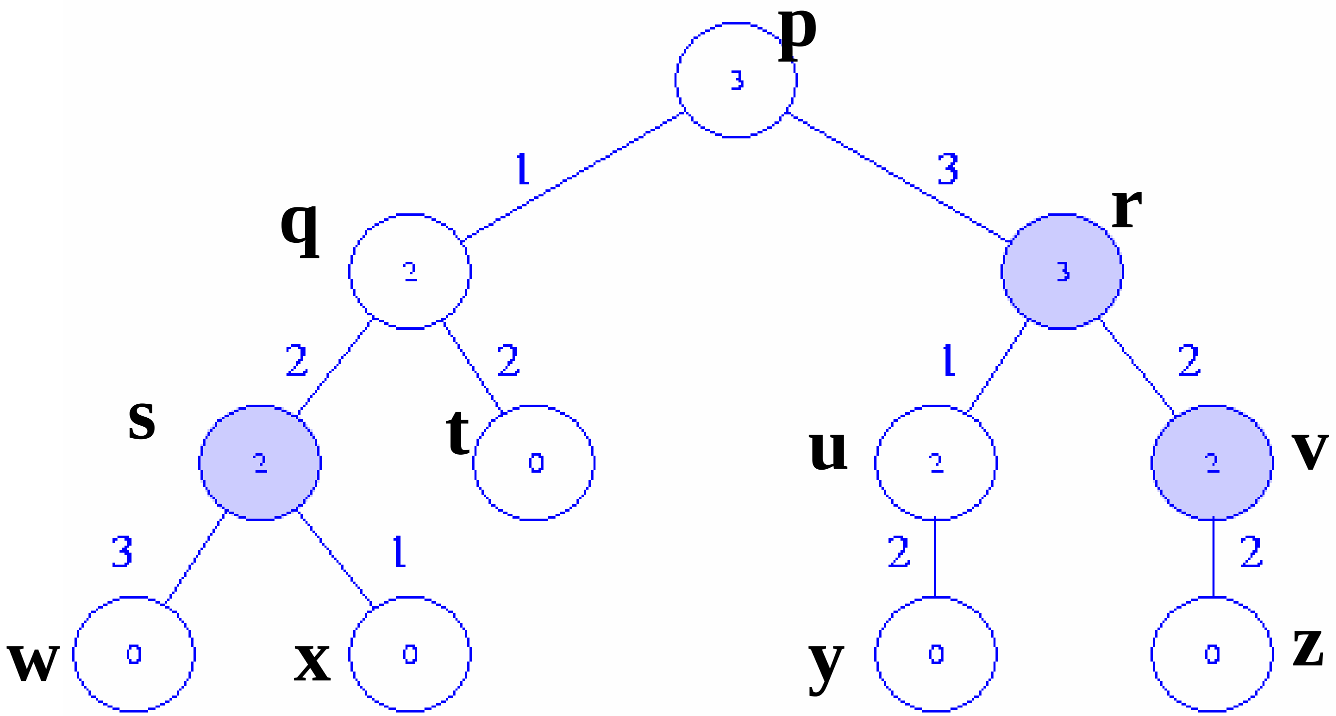
```
degradeToLeaf(i) = 0;
for (each child j of i)
    if (degradeToLeaf(j) + degradeFromParent(j)) > tolerance)
    {
        place a booster at j;
        degradeToLeaf(i) = max{degradeToLeaf(i),
                               degradeFromParent(j)};
    }
    else
        degradeToLeaf(i) = max{degradeToLeaf(i),
                               degradeToLeaf(j) + degradeFromParent(j)};
```

Figure 12.13 Pseudocode to place boosters and compute *degradeToLeaf*

PSB Solution (4)



PSB Solution (5)



Signal boosters are at shaded nodes

Numbers inside nodes are degradeToLeaf values

Figure 12.14 Distribution network with signal boosters

Union-Find Problem

- Given a set {1, 2, …, n} of n elements
- Initially each element is in a different set
  - {1}, {2}, …, {n}
- An intermixed sequence of union & find operations is performed
- A union operation combines two sets into one set
  - Each of the n elements is in exactly one set at any time
- A find operation identifies the set that contains a particular element

UFP Tree Representation

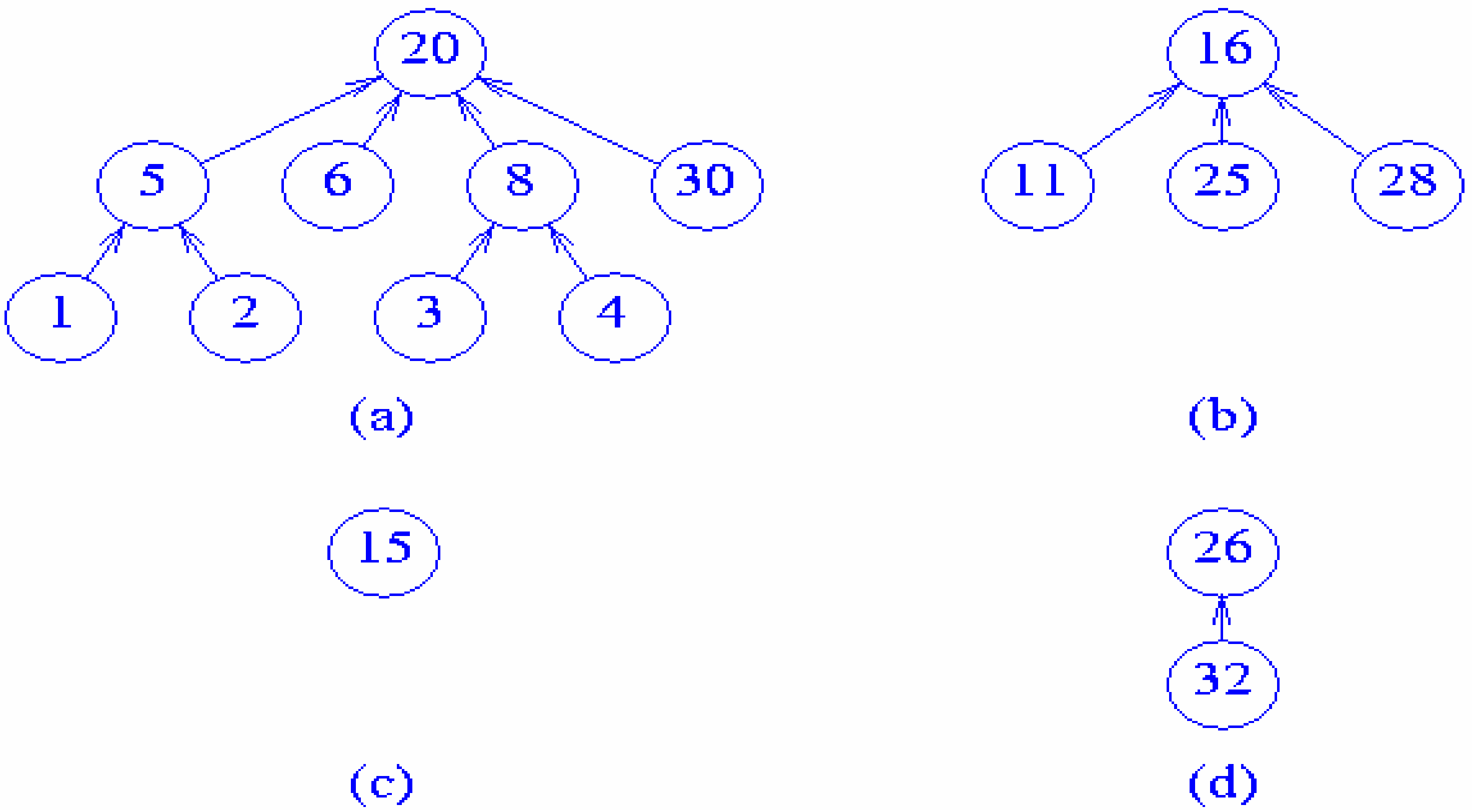
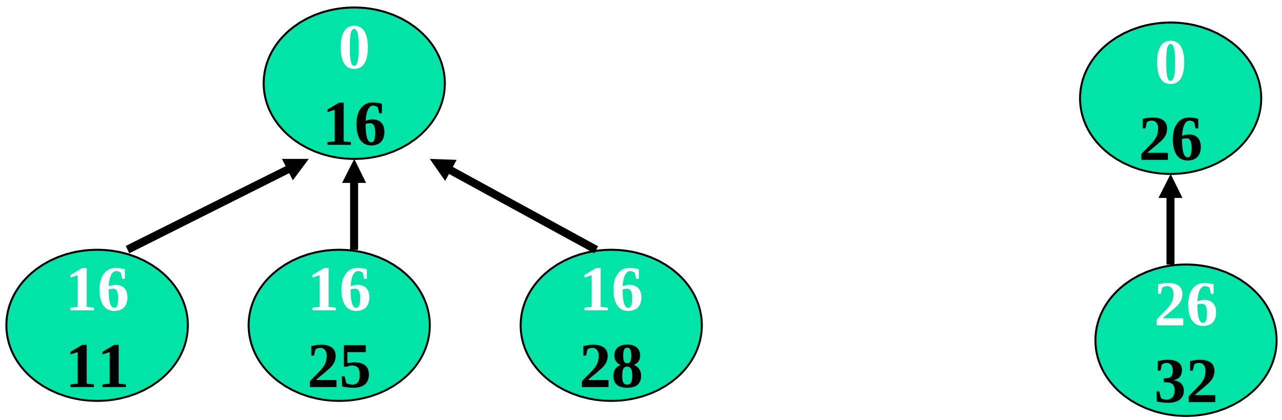


Figure 12.16 Tree representation of disjoint sets

Representing a Set as a Tree

- **ClassA: 16, 11,25,28**
- **ClassB: 26,32**



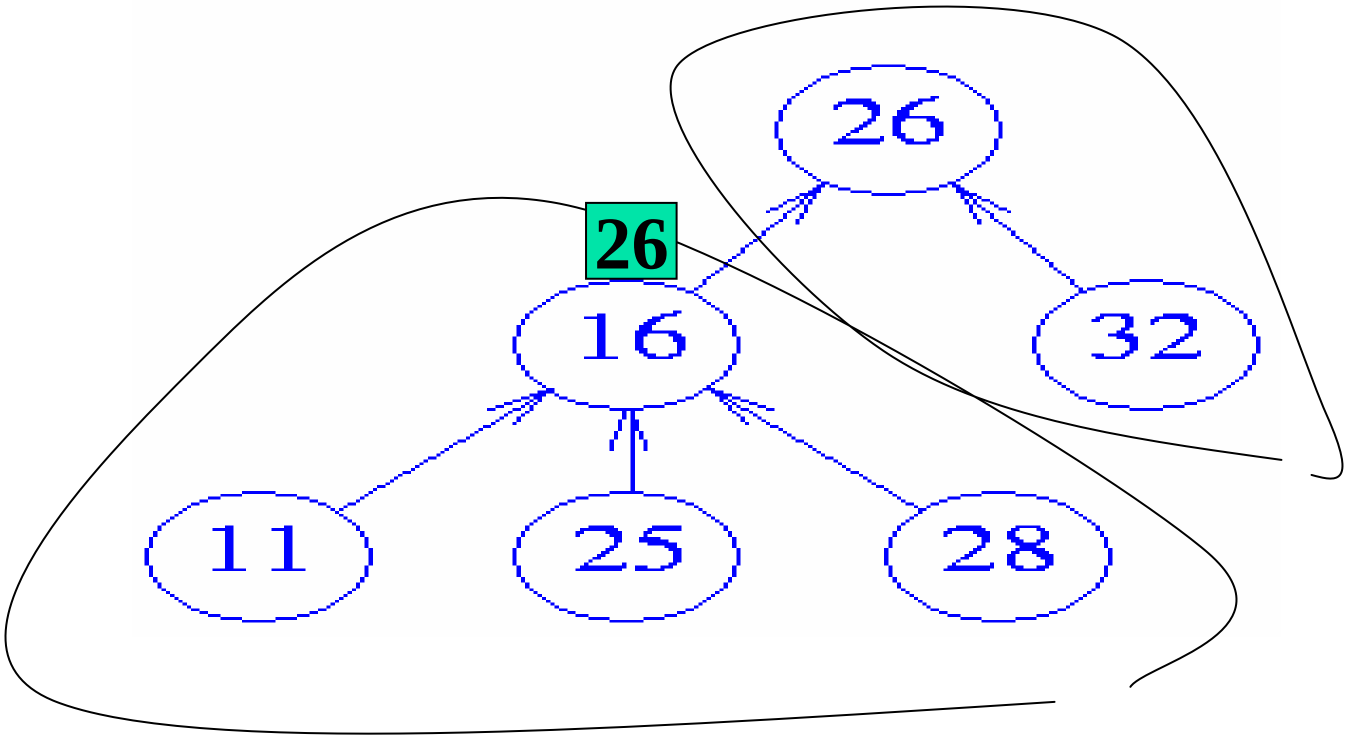
UFP Tree Solution

- Represent each set as a tree
- Find operation
  - Use the element in the root as the set identifier
  - find(3) returns the value 20
  - find(1) returns the value 20
  - find(26) returns the value 26
  - find(i) = find(j) iff i and j are in the same set

- Union operation: Union(classA, classB)
  - To unite the two trees, make one tree a subtree of the other
  - If classA = 16 and classB = 26
    - classA is made a subtree of classB
    - classB is made a subtree of classA

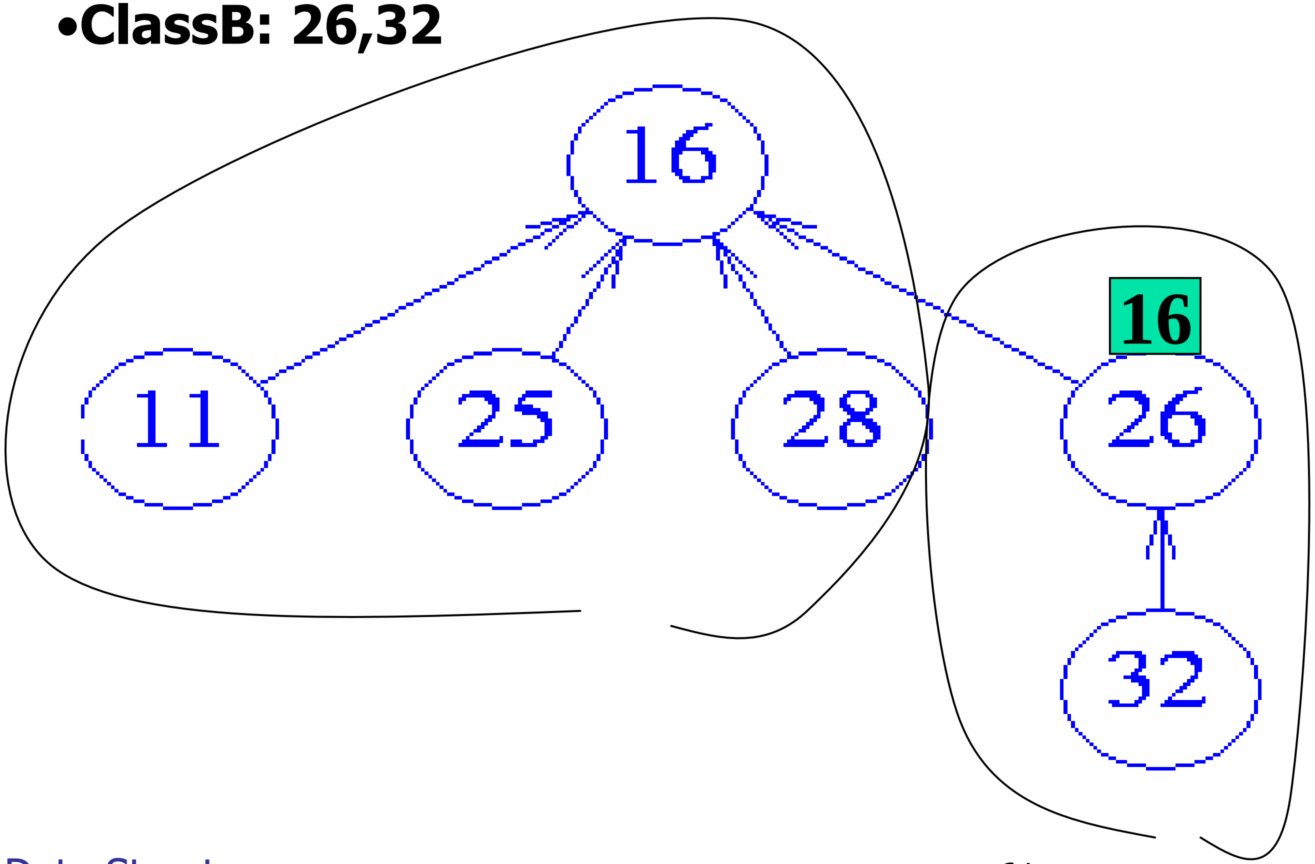
Union(classA, classB): classA is made a subtree of classB

- **ClassA: 16, 11,25,28**
- **ClassB: 26,32**



Union(classA, classB): classB is made a subtree of classA

- **ClassA: 16, 11,25,28**
- **ClassB: 26,32**



UnionFindWithTrees

```
public class UnionFindWithTrees {
    int [] parent;   // pointer to parent in tree
    /** initialize n trees, one element per tree/class/set */
    public UnionFindWithTrees (int n) {
            parent = new int [n + 1];
            for (int e = 1; e <= n; e++)  parent[e] = 0;
    }
    /** @return root of the tree that contains theElement */
    public int find (int theElement) {
            while ( parent[theElement] != 0 )  // move up one level
                        theElement = parent[theElement];
          return theElement;
    }
    /** combine trees with distinct roots rootA and rootB */
     public void union (int rootA, int rootB) { parent[rootB] = rootA; }
}
```

Summary

- Can obtain improved run-time performance by using trees to represent the sets

- Tree terminology
  - Height, Depth, Level
  - Root, Leaf
  - Child, Parent, Sibling
- Representation of BT: array-based vs linked
- The 4 common ways to traverse a BT
  - Preorder/Inorder/Postorder/Level-order

- Tree Applications
  - Placement of Signal Boosters (PSB)
  - Union-Find Problem (UFP)

Sahni class: dataStructures.BinaryTree(p.474)

public interface ***BinaryTree*** {

methods

boolean isEmpty(): Returns *true* if empty, *false* otherwise

Object root(): Returns the root element

void makeTree(Object root, Object left, Object right): Creates a binary tree with *root* as the root element, *left* as the left subtree, *right* as the right subtree

BinaryTree removeLeftSubtree(): Removes the left subtree and returns it

BinaryTree removeRightSubtree(): Removes the right subtree and returns it

void preOrder(Method visit): Carries out preorder traversal

void inOrder(Method visit): Carries out inorder traversal

void postOrder(Method visit): Carries out postorder traversal

void levelOrder(Method visit): Carries out level-order traversal

}