UNIVERSIDAD NACIONAL DE COLOMBIA

Estructuras de Datos

Sesión 9
Tree Data Structure

Yoan Pinzón

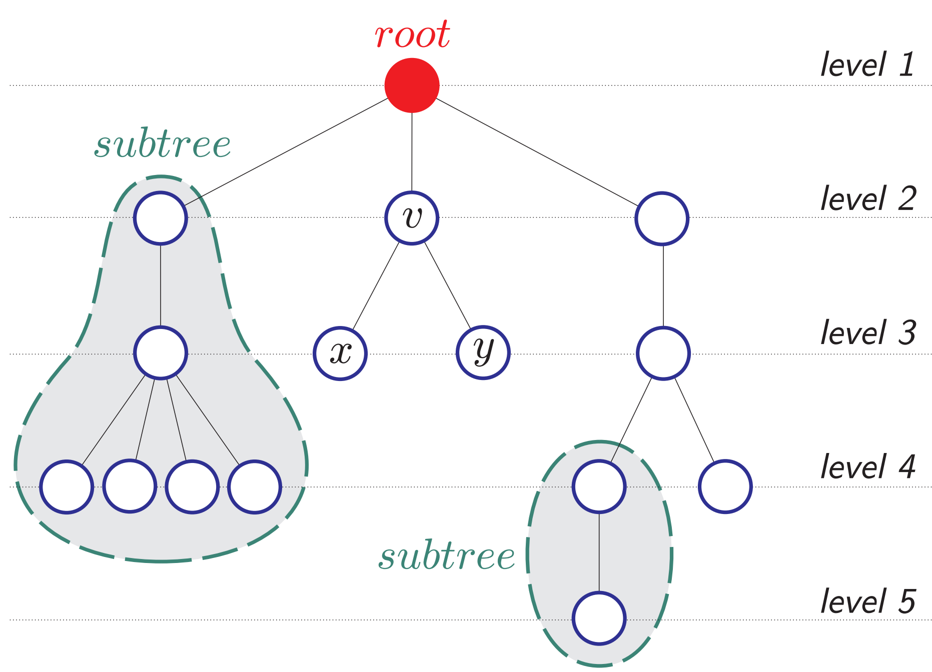
Table of Content Session 9

Tree Data Structure

- Until now: linear and tabular data
- How can we represent hierarchical data?
  - *E.g. somebody's descendants,*
  - *governmental/company subdivisions,*
  - *modular decomposition of programs, etc.*
- Answer: Tree Data Structure

Tree Terms

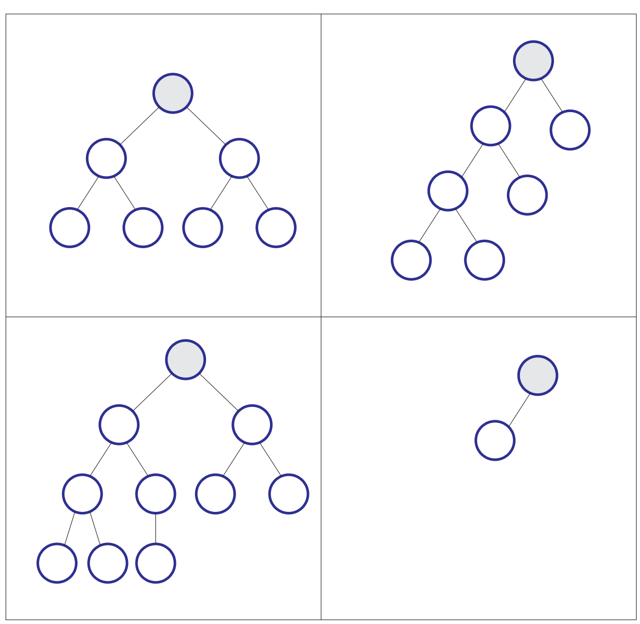
A **tree** is a finite nonempty set of elements



- x, y are **children** of v; v is a **parent** of x, y
- x, y are **siblings**
- Elements with no children are called **leaves**
- **Level:** root=level 1; children=level 2,3, ...
- **Degree of an element:** number of children
- **Height or Depth:** number of levels

Binary Trees

A **binary tree** is a tree (possible empty) in which every element has degree ≤ 2, except for the leaves.



Properties of Binary Trees

P1: *Every binary tree with n elements, $n > 0$, has exactly $n - 1$ edges.*

Proof: Each element (except the root) has one parent. $\exists$ exactly one edge between each child and its parent. Hence, $\exists n - 1$ edges. $\square$

P2: *The number of elements at level i is $\leq 2^{i-1}, i > 0$.*

Proof: By induction on i.
Basis: $i = 1$; number of elements $= 1 = 2^0$
Ind. Hypothesis: $i = k$; number of elements at level $k \leq 2^{k-1}$.
Look at level $i = k + 1$
(number of elements at level $k + 1$) $\leq 2 \cdot$(number of elements at level k) $\leq 2 \times 2^{k-1} = 2^k$. $\square$

P3: *A binary tree of height h, $h > 0$, has at least h and at most $2^h - 1$ elements.*

Poof: Let n be the number of elements. $\exists$ must be ≥ 1 elements at each level, hence, $n \geq h$.
Now, if $h = 0$, then $n = 0 = 2^0 - 1$.
For $h > 0$, we have by P2 that

$$n \leq \sum_{i=1}^{h} 2^{i-1} = 2^h - 1$$

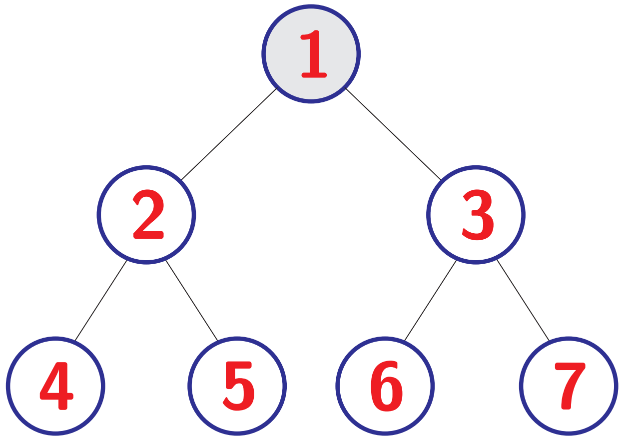
$\square$

P4: *Let h be the height of an n-elements binary tree, $n \geq 0$. Then, $\lceil \log_2(n+1) \rceil \leq h \leq n$*

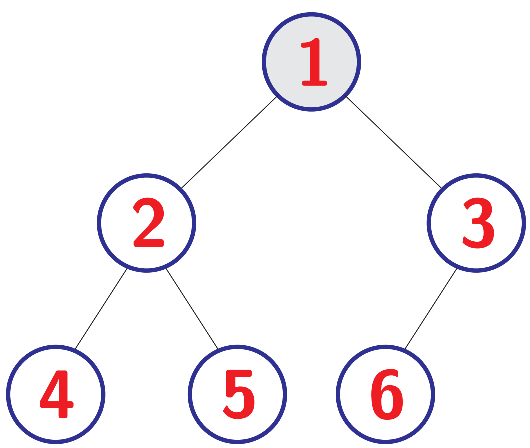
Proof: $\exists$ must be ≥ 1 element at each level, hence, $h \leq n$.
P3 $\Rightarrow n \leq 2^h - 1 \Rightarrow 2^h \geq n + 1 \Rightarrow h \geq \log_2(n+1)$.
Since h is an integer, we have that $h \geq \lceil \log_2(n+1) \rceil$. $\square$

Full Vs Complete

Full binary tree: A binary tree of height h is *full* if contains exactly 2^h-1 elements.



Complete binary tree: Is a binary tree of height h in which all levels (except perhaps for the last) have a maximum number of elements.



Number the elements from 1 through 2^h-k, starting from level 1 and proceed in a left-to-right fashion, for some $k \geq 1$

P5: *Let i, $1 \le i \le n$, be the number assigned to an element v of a complete binary tree. Then:*

(i) If $i = 1$, then v is the root. If $i > 1$, then the parent of v has been assigned the number $\lfloor i/2 \rfloor$.

(ii) If $2i > n$, then v has no left child. Otherwise, its left child has been assigned the number $2i$.

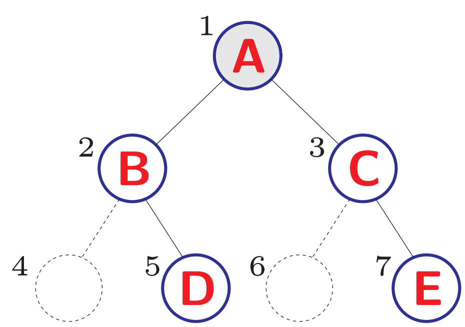
(iii) If $2i + 1 > n$, then v has no right child. Otherwise, its right child has been assigned the number $2i + 1$.

Proof: By induction on i. $\square$

Binary Tree Data Structure

Array-based Representation

Uses **P5**



1	2	3	4	5	6	7
A	B	C		D		E

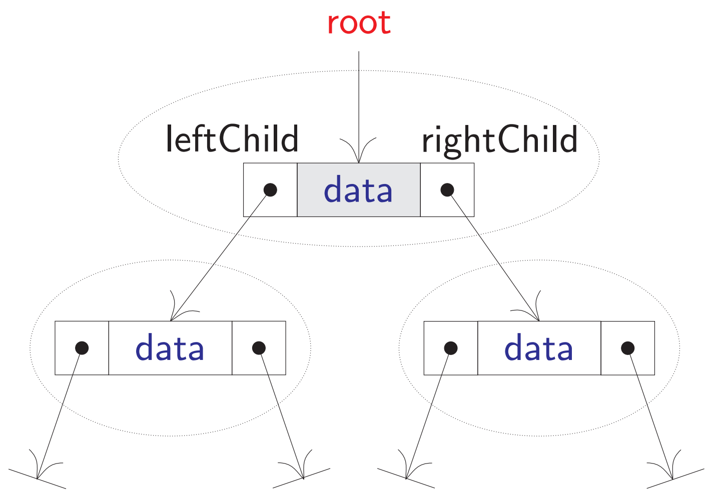
Note: An n–element binary tree may require an array of size $2^n - 1$ for its representation. $\Rightarrow$ Can be a waste of space

Binary Tree Data Structure

Linked Representation

The most popular way to represent a binary tree is by using links or pointers. Each node is represented by three fields:

- data
- leftChild
- rightChild



Binary Tree Traversal

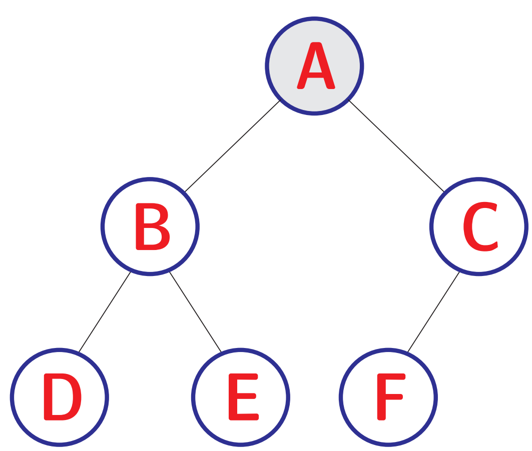
There are four common ways to traverse a binary tree:

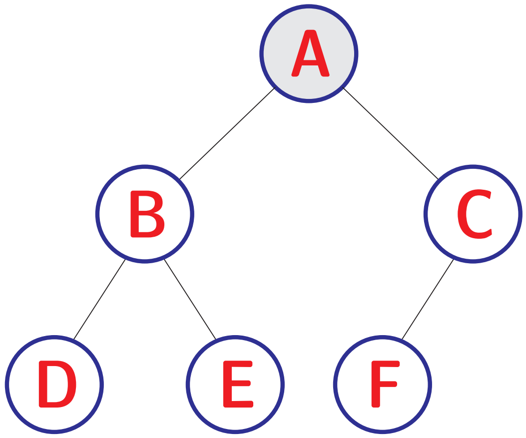
1) **Pre-order:** Visit-Left-Right

2) **In-order:** Left-Visit-Right

3) **Post-order:** Left-Right-Visit

4) **Level order**





1) **Pre-order:** ABDECF

2) **In-order:** DBEAFC

3) **Post-order:** DEBFCA

4) **Level order:** ABCDEF

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, `false` otherwise

`root()`:return the root element, returns `null` if the tree is empty

`makeTree(root, left, right)`:creates a binary tree root as the root element, and left (right) as the left (right) subtree.

`removeLeftSubtree()`:remove the left subtree and return it

`removeRightSubtree()`:remove the right subtree and return it

`preOrder`:preorder traversal of the binary tree

`inOrder`:inorder traversal of the binary tree

`postOrder`:postorder traversal of the binary tree

`levelOrder`:level-order traversal of the binary tree

}

Interface Definition of BinaryTree

```java
package unal.datastructures;

import java.lang.reflect.*;

interface BinaryTree<T>
{
   boolean isEmpty( );
   T root( );
   void makeTree( T root, BinaryTree<T> left, BinaryTree<T> right );
   BinaryTree<T> removeLeftSubtree( );
   BinaryTree<T> removeRightSubtree( );
   void preOrder( Method visit );
   void inOrder( Method visit );
   void postOrder( Method visit );
   void levelOrder( Method visit );
}
```


Class Definition of BinaryTreeNode

```java
package unal.datastructures;

public class BinaryTreeNode<T>
{
   // package visible fields
   T element;
   BinaryTreeNode<T> leftChild; // left subtree
   BinaryTreeNode<T> rightChild; // right subtree

   // constructors
   public BinaryTreeNode( ) { }

   public BinaryTreeNode( T theElement )
   {
      element = theElement;
   }
```

```java
    public BinaryTreeNode( T theElement,
                          BinaryTreeNode<T> theleftChild,
                          BinaryTreeNode<T> therightChild )
    {
        element = theElement;
        leftChild = theleftChild;
        rightChild = therightChild;
    }

    // accessor methods
    public BinaryTreeNode<T> getLeftChild( )
    {
       return leftChild;
    }

    public BinaryTreeNode<T> getRightChild( )
    {
```



```java
       return rightChild;
    }

    public T getElement( )
    {
       return element;
    }

    // mutator methods
    public void setLeftChild( BinaryTreeNode<T> theLeftChild )
    {
       leftChild = theLeftChild;
    }

    public void setRightChild( BinaryTreeNode<T> theRightChild )
    {
       rightChild = theRightChild;
    }
```

```java
    public void setElement( T theElement )
    {
        element = theElement;
    }

    @Override
    public String toString( )
    {
        return element.toString();
    }
}
```


Class Definition of LinkedBinaryTree

```java
package unal.datastructures;

import java.lang.reflect.*;

public class LinkedBinaryTree<T> implements BinaryTree<T>
{
    // instance fields
    BinaryTreeNode<T> root; // root node

    // class fields
    static Method visit;    // visit method to use during a traversal
    static Method theAdd1;  // method to increment count by 1
    static Method theOutput; // method to output node element
    static int count;       // counter

    // method to initialize class fields
```

```
static
{
   try
   {
      Class<LinkedBinaryTree> lbt = LinkedBinaryTree.class;
      theAdd1 = lbt.getMethod( "add1", BinaryTreeNode.class );
      theOutput = lbt.getMethod( "output", BinaryTreeNode.class );
   }
   catch( Exception e )
   {
      // exception not possible
   }
}

// constructor
public LinkedBinaryTree ( ) { /* ... */ }

// class methods
```



```
public static <T> void output( BinaryTreeNode<T> t ) { /* ... */ }
public static <T> void add1( BinaryTreeNode<T> t ) { /* ... */ }

// instance methods
public boolean isEmpty ( ) { /* ... */ }
public T root ( ) { /* ... */ }
public void makeTree ( T root, BinaryTree<T> left,
  BinaryTree<T> right ) { /* ... */ }
public BinaryTree<T> removeLeftSubtree ( ) { /* ... */ }
public BinaryTree<T> removeRightSubtree ( ) { /* ... */ }
public void preOrder ( Method visit ) { /* ... */ }
static <T> void thePreOrder ( BinaryTreeNode<T> t ) { /* ... */ }
public void inOrder ( Method visit ) { /* ... */ }
static <T> void theInOrder ( BinaryTreeNode<T> t ) { /* ... */ }
public void postOrder ( Method visit ) { /* ... */ }
static <T> void thePostOrder ( BinaryTreeNode<T> t ) { /* ... */ }
public void levelOrder ( Method visit ) { /* ... */ }
public void preOrderOutput ( ) { /* ... */ }
```

```
    public void inOrderOutput( ) { /* ... */ }
    public void postOrderOutput( ) { /* ... */ }
    public void levelOrderOutput( ) { /* ... */ }
    public int size( ) { /* ... */ }
    public int height( ) { /* ... */ }
    static <T> int theHeight( BinaryTreeNode<T> t ) { /* ... */ }
    public static void main( String[] args ) { /* ... */ }
}
```


constructor

```
    public LinkedBinaryTree( )
    {
        root = null;
    }
```

class methods

```java
/** visit method that outputs element */
public static <T> void output( BinaryTreeNode<T> t )
{
   System.out.print( t.element + "␣" );
}

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


isEmpty

```java
/** @return true iff tree is empty */
public boolean isEmpty( )
{
   return root == null;
}
```

root

```java
/** @return root element if tree is not empty
 * @return null if tree is empty */
public T root( )
{
   return ( root == null ) ? null : root.element;
}
```


makeTree

```java
/** set this to the tree with the given root and subtrees
 * CAUTION: does not clone left and right */
public void makeTree( T root, BinaryTree<T> left, BinaryTree<T> right )
{
   this.root = new BinaryTreeNode<T>( root,
      ((LinkedBinaryTree<T>) left).root,
      ((LinkedBinaryTree<T>) right).root );
}
```

removeLeftSubtree

```java
/** remove the left subtree
 * @throws IllegalArgumentException when tree is empty
 * @return removed subtree */
public BinaryTree<T> removeLeftSubtree( )
{
   if( root == null )
      throw new IllegalArgumentException( "tree␣is␣empty" );

   // detach left subtree and save in leftSubtree
   LinkedBinaryTree<T> leftSubtree = new LinkedBinaryTree<T>( );
   leftSubtree.root = root.leftChild;
   root.leftChild = null;

   return ( BinaryTree<T> ) leftSubtree;
}
```


removeRightSubtree

```java
/** remove the right subtree
 * @throws IllegalArgumentException when tree is empty
 * @return removed subtree */
public BinaryTree<T> removeRightSubtree( )
{
   if( root == null )
      throw new IllegalArgumentException( "tree␣is␣empty" );

   // detach right subtree and save in rightSubtree
   LinkedBinaryTree<T> rightSubtree = new LinkedBinaryTree<T>( );
   rightSubtree.root = root.rightChild;
   root.rightChild = null;

   return ( BinaryTree<T> ) rightSubtree;
}
```

preOrder

```java
/** preorder traversal */
public void preOrder( Method visit )
{
   LinkedBinaryTree.visit = visit;
   thePreOrder( root );
}
```


thePreOrder

```java
/** actual preorder traversal method */
static <T> void thePreOrder( BinaryTreeNode<T> t )
{
   if( t != null )
   {
      try
      {
         visit.invoke( null, t ); // visit tree root
      }
      catch ( Exception e )
      {
         System.out.println( e );
      }
      thePreOrder( t.leftChild ); // do left subtree
      thePreOrder( t.rightChild ); // do right subtree
   }
}
```

preOrderOutput

```java
/** output elements in preorder */
public void preOrderOutput( )
{
   preOrder( theOutput );
}
```


inOrder

```java
/** inorder traversal */
public void inOrder( Method visit )
{
   LinkedBinaryTree.visit = visit;
   theInOrder( root );
}
```

theInOrder

```java
/** actual inorder traversal method */
static <T> void theInOrder ( BinaryTreeNode<T> t )
{
   if( t != null )
   {
      theInOrder( t.leftChild ); // do left subtree
      try
      {
         visit.invoke( null, t ); // visit tree root
      }
      catch( Exception e )
      {
         System.out.println( e );
      }
      theInOrder( t.rightChild ); // do right subtree
   }
}
```


inOrderOutput

```java
/** output elements in inorder */
public void inOrderOutput ( )
{
   inOrder( theOutput );
}
```

postOrder

```java
/** postorder traversal */
public void postOrder( Method visit )
{
   LinkedBinaryTree.visit = visit;
   thePostOrder( root );
}
```


thePostOrder

```java
/** actual postorder traversal method */
static <T> void thePostOrder( BinaryTreeNode<T> t )
{
   if( t != null )
   {
      thePostOrder( t.leftChild );        // do left subtree
      thePostOrder( t.rightChild );       // do right subtree
      try
      {
         visit.invoke( null, t ); // visit tree root
      }
      catch ( Exception e )
      {
         System.out.println( e );
      }
   }
}
```

postOrderOutput

```java
/** output elements in postorder */
public void postOrderOutput( )
{
   postOrder( theOutput );
}
```


levelOrder

```java
/** level order traversal */
public void levelOrder( Method visit )
{
   ArrayQueue<BinaryTreeNode<T>> q = new ArrayQueue<>( );
   BinaryTreeNode<T> t = root;
   while( t != null )
   {
      try
      {
         visit.invoke( null, t ); // visit tree root
      }
      catch ( Exception e )
      {
         System.out.println( e );
      }

      // put t's children on queue
      if( t.leftChild != null )
         q.put( t.leftChild );
```

```java
        if( t.rightChild != null )
           q.put( t.rightChild );

        // get next node to visit
        t = ( BinaryTreeNode<T> ) q.remove( );
     }
  }
```


levelOrderOutput

```java
/** output elements in level order */
public void levelOrderOutput( )
{
   levelOrder( theOutput );
}
```

size

```java
/** count number of nodes in tree */
public int size( )
{
   count = 0;
   preOrder( theAdd1 );
   return count;
}
```


height

```java
/** @return tree height */
public int height( )
{
   return theHeight( root );
}

/** @return height of subtree rooted at t */
static <T> int theHeight( BinaryTreeNode<T> t )
{
   if( t == null ) return 0;
   int hl = theHeight( t.leftChild ); // height of left subtree
   int hr = theHeight( t.rightChild ); // height of right subtree
   if( hl > hr ) return ++hl;
   else return ++hr;
}
```

main

```java
/** test program */
public static void main( String[] args )
{
   LinkedBinaryTree<Integer> a = new LinkedBinaryTree<>( ),
                       x = new LinkedBinaryTree<>( ),
                       y = new LinkedBinaryTree<>( ),
                       z = new LinkedBinaryTree<>( );
   y.makeTree( new Integer( 1 ), a, a );
   z.makeTree( new Integer( 2 ), a, a );
   x.makeTree( new Integer( 3 ), y, z );
   y.makeTree( new Integer( 4 ), x, a );

   System.out.println( "Preorder␣sequence␣is␣" );
   y.preOrderOutput( );
   System.out.println( );

   System.out.println( "Inorder␣sequence␣is␣" );
   y.inOrderOutput( );
   System.out.println( );
```



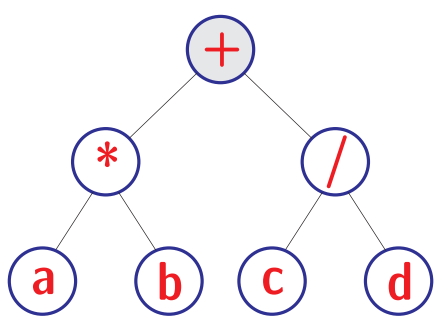
```java
   System.out.println( "Postorder␣sequence␣is␣" );
   y.postOrderOutput( );
   System.out.println( );

   System.out.println( "Level␣order␣sequence␣is␣" );
   y.levelOrderOutput( );
   System.out.println( );

   System.out.println( "Number␣of␣nodes␣=␣" + y.size( ) );

   System.out.println( "Height␣=␣" + y.height( ) );
}
```

Compiling `LinkedBinaryTree.java`

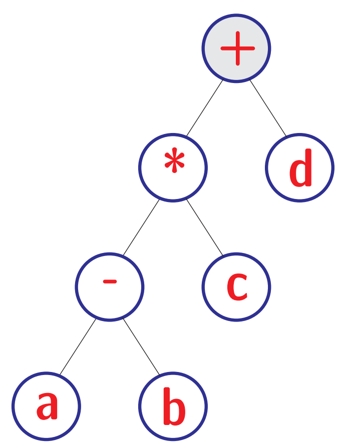
```
C:\2016699\code> javac unal\datastructures\LinkedBinaryTree.java
C:\2016699\code> java unal.datastructures.LinkedBinaryTree
Preorder sequence is
4 3 1 2
Inorder sequence is
1 3 2 4
Postorder sequence is
1 2 3 4
Level order sequence is
4 3 1 2
Number of nodes = 4
Height = 3
```


Binary Tree Application

Expression Trees



(a*b)+(c/d)



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

infix form:	a*b+c/d	a-b*c+d
prefix form:	+*ab/cd	+*-abcd
postfix form:	ab*cd/+	ab-c*d+

- infix: ambiguous; pre/postfix: unambiguous

- postfix evaluation:
  - Scan left-to-right
  - If an operand is encountered, it is stacked in to a stack of operands
  - If an operator is encountered, apply operator to the correct number of operands in the top of the stack and replace them for the result produced by the operator