Tree Structures

- A tree is a hierarchical structure that places elements in nodes along branches that originate from a root.
- Nodes in a tree are subdivided into levels in which the topmost level holds the root node.
- Any node in a tree can have multiple successors at the next level. Hence, a tree is a nonlinear structure.
Tree Structures (2)

- Operating systems use a general tree to maintain file structures.

Tree Structures (3)

- In a binary tree each node has at most two successors. A compiler builds binary trees while parsing expressions in a program's source code.
Tree Terminology

- A tree structure is characterized as a collection of nodes that originate from a unique starting node called the root.
  - Each node consists of a value and a set of zero or more links to successor nodes.
  - The terms parent and child describe the relationship between a node and any of its successor nodes.

Tree Terminology (2)

- A path between a parent node P and any node N in its subtree is a sequence of nodes P=X₀, X₁, . . . , Xₖ = N where k is the length of the path. Each node Xᵢ in the sequence is the parent of Xᵢ₊₁ for 0 ≤ i ≤ k-1.
  - The level of a node is the length of the path from root to the node. Viewing a node as a root of its subtree, the height of a node is the length of the longest path from the node to a leaf in the subtree.
  - The height of a tree is the maximum level in the tree.
Tree Terminology (3)

In a binary tree, each parent has no more than two children.

A binary tree has a uniform structure that allows a simple description of its node structure and the development of a variety of tree algorithms.
Binary Trees (2)

- Each node of a binary tree defines a left and a right subtree. Each subtree is itself a tree.

Binary Trees (3)

- An alternative recursive definition of a binary tree:
  - T is a binary tree if T
    - has no node (T is an empty tree)
    - or
    - has at most two subtrees.
Binary Tree Nodes

- Define a binary tree node as an instance of the generic TNode class.
  - A node contains three fields.
    - The data value, called `nodeValue`.
    - The reference variables, `left` and `right` that identify the left child and the right child of the node respectively.

![TNode Object](image)

Binary Tree Nodes (2)

- The TNode class allows us to construct a binary tree as a collection of TNode objects.
TNode Class

```java
public class TNode<T>
{
    public T nodeValue; // node's value
    public TNode<T> left, right; // subtree references
    // create instance with a value and null subtrees
    public TNode(T item)
    {
        nodeValue = item;
        left = right = null;
    }
    // initialize the value and the subtrees
    public TNode(T item, TNode<T> left, TNode<T> right)
    {
        nodeValue = item;
        this.left = left;
        this.right = right;
    }
}
```

Building a Binary Tree

- A binary tree is a collection of TNode objects whose reference values specify links to their children. Build a binary tree one node at a time.

```java
TNode<Integer> p, q; // references to TNode objects with
                      // Integer data
// p is a leaf node with value 8;
p = new TNode<Integer>(8);

// q is a node with value 4 and p as a right child
q = new TNode<Integer>(4, null, p);
```

![Binary Tree Diagram]
Use the TNode class to build a binary tree from the bottom up.

```java
// references to Integer tree nodes
TNode<Integer> root, p, q, r;
// create leaf node p with value 20
// and leaf node q with value 40
p = new TNode<Integer>(20);
q = new TNode<Integer>(40);
// create internal node r with value 30,
// left child q, and a null right child
r = new TNode<Integer>(30, q, null);
// create root node with value 10,
// left child p, and right child r
root = new TNode<Integer>(10, p, r);
```

Building a Binary Tree (end)

```java
// n is in the range 0 to 2
public static TNode<Character> buildTree(int n) {
    ... }
```
Recursive Binary Tree-Scan Algorithms

To scan a tree recursively we must visit the node (N), scan the left subtree (L), and scan the right subtree (R). The order in which we perform the N, L, R tasks determines the scan algorithm.

Recursive Scanning Example

Preorder (NLR): A B D G C E H I F
Inorder (LNR): D G B A H E I C F
Postorder (LRN): G D B H I E F C A
Inorder Scan

- The inorder scan of a tree visits the left subtree L, visits the node N, then visits the right subtree R. To scan the entire tree, begin with the root.

![Inorder Scan Diagram]

Designing Recursive Scanning Methods

Recursive Scan Design Pattern (assuming an inorder scan (L N R) and a return value)

```java
public static <T> ReturnType scanMethod(TNode<T> t) {
    // check for empty tree (stopping condition)
    if (t == null) {
        // return information for an empty tree
    } else {
        // descend to left subtree and record return information
        valueLeft = scanMethod(t.left);
        // visit the node and record information
        < evaluate t.nodeValue >
        // descend to right subtree and record return information
        valueRight = scanMethod(t.right);
    }
    return <information from valueLeft, valueRight, and t.nodeValue >
}
```
Designing Scanning Methods (end)

Preorder Design Pattern:
<pre>
<evaluate t.nodeValue> // visit node first
valueLeft = scanMethod(t.left);  // go left
valueRight = scanMethod(t.right); // go right
</pre>

Postorder Design Pattern:
<pre>
valueLeft = scanMethod(t.left);  // go left
valueRight = scanMethod(t.right); // go right
<evaluate t.nodeValue> // visit node last
</pre>

Console Output for an Inorder Scan

```java
// list the nodes of a binary tree using an LNR scan
public static &lt;T&gt; void inorderOutput(TNode&lt;T&gt; t)
{
    // the recursive scan terminates on an empty subtree
    if (t != null)
    {
        inorderOutput(t.left);  // descend left
        System.out.print(t.nodeValue + " ");
        inorderOutput(t.right);  // descend right
    }
}
```
inorderDisplay()

// list the nodes of a binary tree using an LNR scan
public static <T> String inorderDisplay(TNode<T> t)
{
    // return value
    String s = "";

    // the recursive scan terminates on a empty subtree
    if (t != null)
    {
        s += inorderDisplay(t.left); // descend left
        s += t.nodeValue + "  "; // display the node
        s += inorderDisplay(t.right); // descend right
    }
    return s;
}

Iterative Level-Order Scan

- A level-order scan visits the root, then nodes on level 1, then nodes on level 2, etc.
Iterative Level-Order Scan (2)

- A level-order scan is an iterative process that uses a queue as an intermediate storage collection.
  - Initially, the root enters the queue.
  - Pop a node from the queue, perform some action with the node, and then push its children onto the queue. Because siblings enter the queue during a visit of their parent, the siblings (on the same level) will exit the queue in successive iterations.

Iterative Level-Order Scan (3)

1. Initial setup: Create a queue with the root as the only element.
2. Pop D
3. Pop E
4. Pop A. Push B, C
5. Pop B. Push D
6. Pop C. Push E

Visit: A B C D E
levelorderDisplay()

// list the value of each node in a binary tree using a
// level order scan of the nodes
public static <T> String levelorderDisplay(TNode<T> t)
{
    // store siblings of each node in a queue
    // so that they are visited in order at the
    // next level of the tree
    LinkedQueue<TNode<T>> q =
        new LinkedQueue<TNode<T>>();
    TNode<T> p;
    // return value
    String s = "";

    // initialize the queue by inserting the
    // root in the queue
    q.push(t);

    // continue until the queue is empty
    while(!q.isEmpty())
    {
        // delete node from queue and output node value
        p = q.pop();
        s += p.nodeValue + " ";

        // if a left child exists, insert it in the queue
        if(p.left != null)
            q.push(p.left);

        // if a right child exists, insert next
        // to its sibling
        if(p.right != null)
            q.push(p.right);
    }

    return s;
}
Visitor Design Pattern

- The Visitor design pattern applies an action to each element of a collection.
- The Visitor interface defines the visit() method which denotes what a visitor does. For a specific visitor pattern, create a class that implements the Visitor interface. During traversal, call visit() and pass the current value as an argument.

```java
public interface Visitor<T>
{
    void visit(T item);
}
```

Visitor Design Pattern (2)

```java
public class VisitOutput<T> implements Visitor<T>
{
    public void visit(T obj)
    {
        System.out.print(obj + " ");
    }
}
```
Visitor Design Pattern (3)

```java
public class VisitMax<T extends Comparable<? super T>> implements Visitor<T>
{
    T max = null;
    public void visit(T obj)
    {
        if (max == null)
            max = obj;
        else if (obj.compareTo(max) > 0)
            max = obj;
    }
    public T getMax()
    {
        return max;
    }
}
```

scanInorder()

- The recursive method scanInorder() provides a generalized inorder traversal of a tree that performs an action specified by a visitor pattern.

```java
public static <T> void scanInorder (TNode<T> t, Visitor<T> v)
{
    if (t != null)
    {
        scanInorder(t.left, v);
        v.visit(t.nodeValue);
        scanInorder(t.right, v);
    }
}
```
Copying a Binary Tree

- In many applications, a programmer wants to duplicate a tree structure.
- Copy a tree using a postorder scan. This builds the duplicate from the bottom up.

```java
public static <T> T copyTree(TNode<T> t)
```

```plaintext
Original Tree 0
origRoot = TNode.buildTree (0)  
copyRoot = TNode.copyTree (origRoot)
```

Tree 0 and the copy of Tree 0 with roots origRoot and copyRoot respectively.

Copying a Binary Tree (2)
Copying a Binary Tree (3)

Copying a Binary Tree (4)

```java
// create a duplicate of the tree with root t and return
// a reference to its root
public static <T> TNode<T> copyTree(TNode<T> t) {
    // newNode points at a new node that the algorithm
    // creates; newLptr and newRptr point to the subtrees
    // of newNode
    TNode<T> newLeft, newRight, newNode;

    // stop the recursive scan when we
    // arrive at empty tree
    if (t == null) {
        return null;
    }
```
Copying a Binary Tree (end)

// build new tree from the bottom up by building the two
// subtrees and then building the parent; at node t,
// make a copy of the left subtree and assign its root
// node reference to newLeft; make a copy of the right
// subtree and assign its root node reference to newRight
newLeft = copyTree(t.left);
newRight = copyTree(t.right);

// create a new node whose value is the same as the value
// in t and whose children are the copied subtrees
newNode = new TNode<T> (t.nodeValue, newLeft, newRight);

// return a reference to the root of the
// newly copied tree
return newNode;

Clearing a Binary Tree

- Clear a tree with a postorder scan. It
removes the left and right subtrees before
removing the node.

```
public static <T> void clearTree(TNode<T> t)
{
    // postorder scan: delete left and right
    // subtrees of t and then node t
    if (t != null)
    {
        clearTree(t.left);
        clearTree(t.right);
        t = null;
    }
}
```
Displaying a Binary Tree

- BinaryTree.displayTree() returns a string that has a layout of the node values in a binary tree. BinaryTree.drawTree() gives a graphical view of the tree.

```java
// return a string that displays a binary tree. output of
// a node value requires no more than maxCharacters
public static <T>
    String displayTree(TNode<T> t, int maxCharacters) 
    { ... }

// displays a tree in a graphical window
public static <T>
    void drawTree(TNode<T> t, int maxCharacters) 
    { ... }
```

Program 16.1

Integer tree to illustrate scanning algorithms and Visitor patterns.
import ds.util.TNode;
import ds.util.BinaryTree;

public class Program16_1
{
    public static void main(String[] args)
    {
        // root of the tree
        TNode<Integer> root;

        // create the Visitor objects
        VisitOutput<Integer> output =
            new VisitOutput<Integer>();
        VisitMax<Integer> max = new VisitMax<Integer>();

        // create the tree using buildTree16_1
        root = buildTree16_1();

        System.out.println("Scans of the tree");
        System.out.println("   Preorder scan:    " +
            BinaryTree.preorderDisplay(root));
        System.out.println("   Inorder scan:     " +
            BinaryTree.inorderDisplay(root));
        System.out.println("   Postorder scan:   " +
            BinaryTree.postorderDisplay(root));
        System.out.println("   Level order scan: " +
            BinaryTree.levelorderDisplay(root) + "\n");

        // use Vistor object and scanInorder() to traverse
        // the tree and determine the maximum value
        System.out.println("Call scanInorder() with VisitOutput: ");
        scanInorder(root, output);
        System.out.println();
Program 16.1 (4)

```java
scanInorder(root, max);
System.out.println(
  "Call scanInorder() with VisitMax: " +
  "Max value is " + max.getMax());
}

public static <T> void scanInorder(TNode<T> t,
                          Visitor<T> v)
{
  if (t != null)
  {
    scanInorder(t.left, v);
    v.visit(t.nodeValue);
    scanInorder(t.right, v);
  }
}
```

Program 16.1 (end)

```java
public static TNode<Integer> buildTree16_1()
{
  // TNode references; point to 8 items in the tree
  TNode<Integer> root20 = null, t45, t15, t30,
                      t5, t10, t25, t35;
  t35 = new TNode<Integer>(35);
  t25 = new TNode<Integer>(25);
  t10 = new TNode<Integer>(10, null, t35);
  t5 = new TNode<Integer>(5);
  t30 = new TNode<Integer>(30, t25, null);
  t15 = new TNode<Integer>(15, t5, t10);
  t45 = new TNode<Integer>(45, null, t30);
  root20 = new TNode<Integer>(20, t45, t15);

  return root20;
}
```
Program 16.1 (Run)

Scans of the tree

Preorder scan:  20  45  30  25  15  5  10  35
Inorder scan:   45  25  30  20  5  15  10  35
Postorder scan: 25  30  45  5  35  10  15  20
Level order scan: 20  45  15  30  5  10  25  35

Call scanInorder() with VisitOutput:
45  25  30  20  5  15  10  35
Call scanInorder() with VisitMax: Max value is 45

Program 16.2

```java
import ds.util.TNode;
import ds.util.BinaryTree;

public class Program16_2
{
    public static void main(String[] args)
    {
        // roots for two trees
        TNode<Character> root, copyRoot;

        // build the character Tree 2 with root root2
        root = BinaryTree.buildTree(2);

        // display the original tree on the console
        System.out.println(
            BinaryTree.displayTree(root, 1));
    }
}```
Program 16.2 (end)

// make a copy of root1 so its root is root2
copyRoot = BinaryTree.copyTree(root);

// graphically display the tree copy
BinaryTree.drawTree(copyRoot, 1);
}
}

Run of Program 16.2

Run:

```
   A
  /   \
 B     C
/     / \ 
D     E   F
|     |   |
G     H   I
```

```
A
B     C
\       \ 
D       E  F
|       |   |
G       H  I
```