CMP-5014Y Data Structures and Algorithms

Geoff McKeown

Binary Trees

A Fundamental Non-linear data Structure

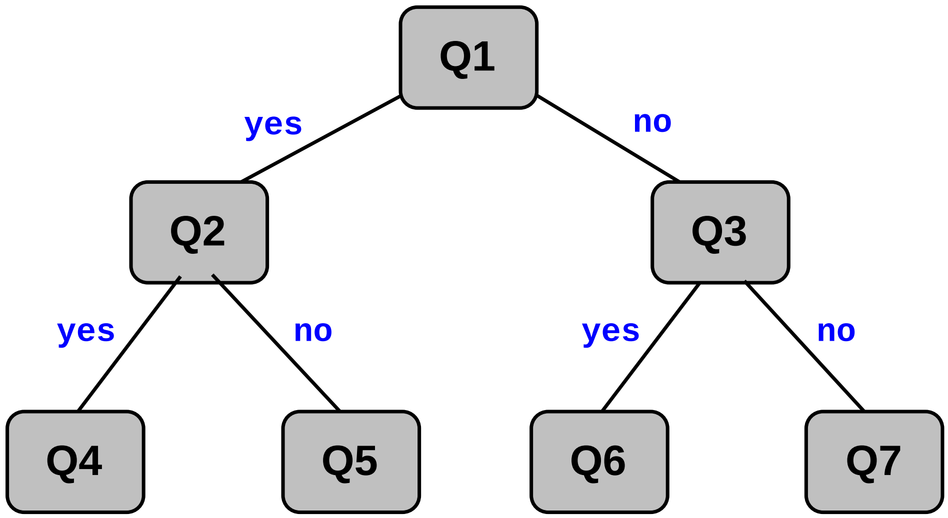
Lecture Objectives

- To introduce the concept of tree data structures.
- To give a Java implementation of a binary tree abstract data type.
- To introduce a number of binary tree traversal algorithms.

Binary Trees

- All forms of lists,
  - e.g. stacks, queues, arrays, ArrayLists, linked lists,

  are *linear* structures
  - the elements in such structures are held one after the other in a chain-like fashion.
- Tree structures are *non-linear*
  - the elements are held in a hierarchical fashion which does not, in general, form a chain.
- A familiar everyday example of a tree structure is a family tree.
- Another important type of tree is a *decision tree*:



◇ A decision tree is a particular type of *binary tree*.

A binary tree is defined recursively as follows:

Definition A ***binary tree***, T, on a set of elements, E, is either

(i) empty, or

(ii) consists of a finite collection of nodes, each containing an element of E, and which contains a particular node called the ***root*** of T, with the remaining nodes of T partitioned into two binary trees, called the ***left sub-tree*** and the ***right sub-tree***, respectively.

Terminology

nodes or ***vertices*** - contain elements of T.

parent (node) - every node except for the root has a unique parent node:

if p is the root of a binary tree, T, and c is the root of either the left or the right sub-tree of T, then p is the parent of c

child (node) - if p is the parent of c then c is a child of p

siblings - two nodes are siblings if they have the same parent node

ancestor (node) - node a is an ancestor of node d if either a is the parent of d or a is the parent of an ancestor of d.

descendant (node) - node d is a descendant of node a if a is an ancestor of d.

leaf (node) - a node with no child.

external node - another name for a leaf node

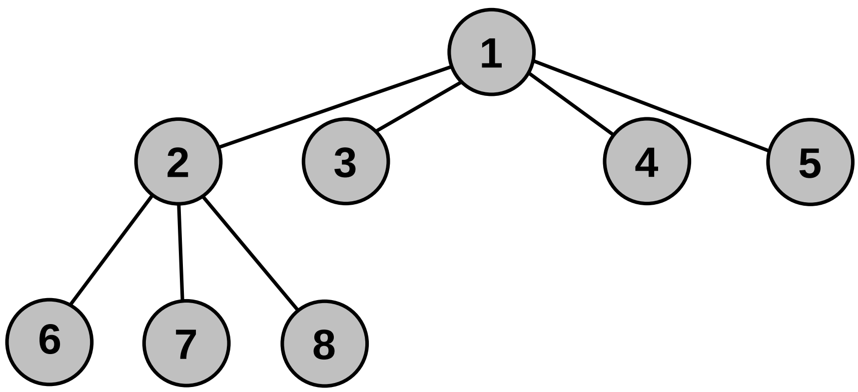
internal node - a non-leaf node, i.e. a node with at least one child.

level of a node - if n is the root node then $\textbf{\textit{level}}(n) = 0$ otherwise $\textbf{\textit{level}}(n) = \textbf{\textit{level}}(\textbf{\textit{parent}}(n)) + 1$.

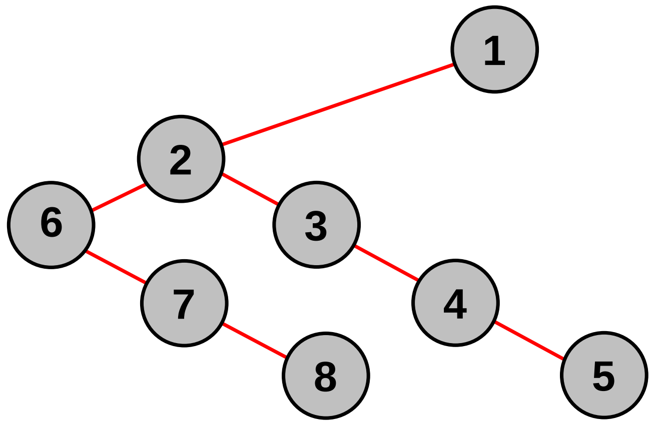
height of tree - $\textbf{\textit{height}}(T) = \max_{n \in T} \textbf{\textit{level}}(n)$ (height of T is also called the ***level of the tree***, T).

General Trees

- ◇ More generally, nodes in a tree may have more than two children.
- ◇ But: any such tree may be represented by a binary tree.
  - ▷ This is not always an effective thing to do, however.
- ◇ An example of a non-binary tree:



- ◇ To create a binary tree equivalent to a given general tree:
  - ▷ the leftmost child of a node in the general tree becomes the left child of that node in the binary tree;
  - ▷ the remaining children of that node in the general tree form a chain of right descendants of the left child in the binary tree.
- ◇ The binary tree corresponding to the tree above:



Some Properties of Binary Trees

- ◇ A binary tree of height h can have at most $2^{h+1} - 1$ nodes.
- ◇ A binary tree of n nodes has level h, where

$$\lceil \log_2(n+1) \rceil - 1 \leq h \leq n - 1.$$

Some Binary Tree Traversals

(i) *PREORDER*

visit root
visit left sub-tree in preorder
visit right sub-tree in preorder

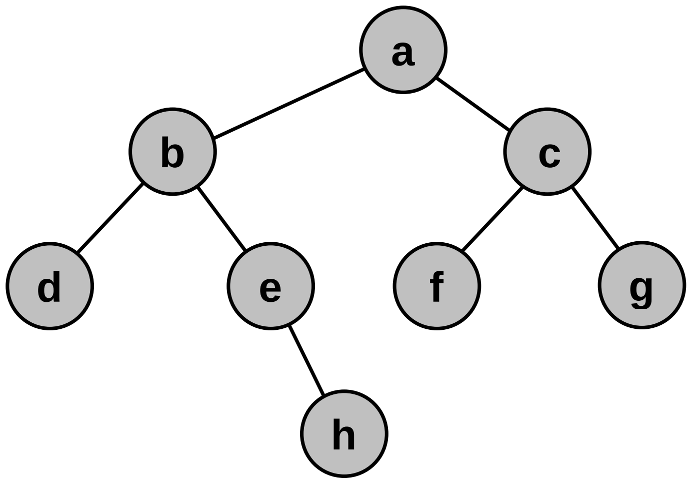
(ii) *INORDER*

visit left sub-tree in inorder
visit root
visit right sub-tree in inorder

(iii) *POSTORDER*

visit left sub-tree in postorder
visit right sub-tree in postorder
visit root

Example



- ◇ *preorder*:

  `a, b, d, e, h, c, f, g`

- ◇ *inorder*:

  `d, b, e, h, a, f, c, g`

- ◇ *postorder*:

  `d, h, e, b, f, g, c, a`

Implementation

◇ We begin by specifying an *abstract data type* (ADT) by a Java interface.

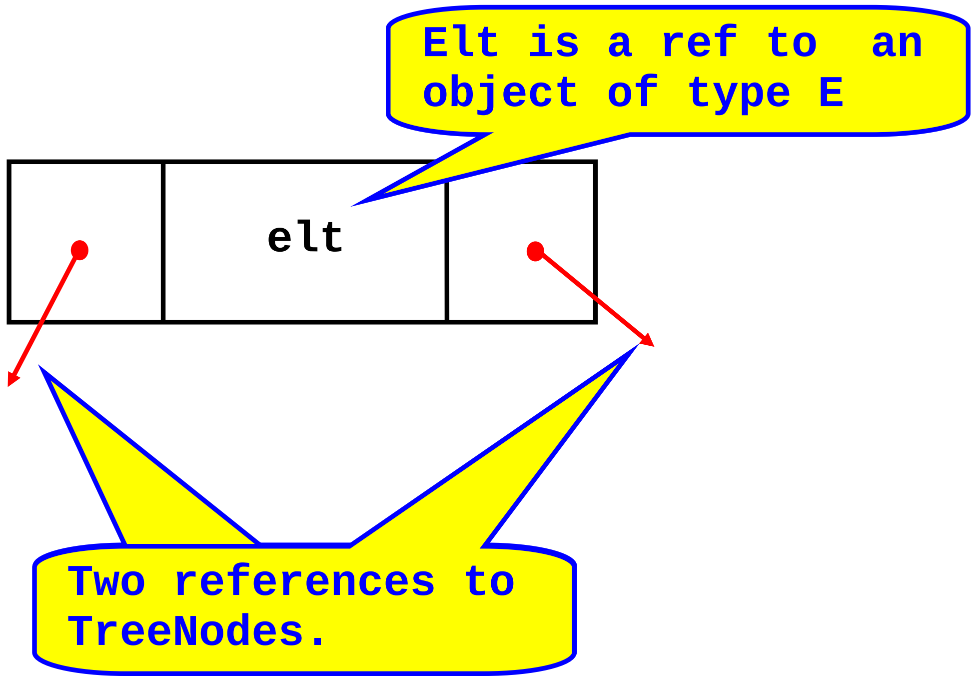
Java Interface

3 accessor methods, one for each of the 3 parts that make up any binary tree. Plus a method that tests for an empty tree.

```
package BinaryTree;

public interface ADT_BinaryTree<E>
{
  public boolean isEmpty();

  public E getRootElt();

  public BinaryTree getLeftTree();

  public BinaryTree getRightTree();

}
```

```
package BinaryTree;

public class BinaryTree<E> implements ADT_BinaryTree<E>

{
// BinaryTree has only one data member (field)

  private TreeNode root;
```

TreeNode is a private class within BinaryTree


Elt is a ref to an object of type E
elt
Two references to TreeNodes.

```
package BinaryTree;

private class TreeNode
{
  // Data members

  TreeNode left;

  TreeNode right;

  E elt;


  // Constuctors

  TreeNode( ) { this( null ); }

  TreeNode( E anElt )
    { this( anElt, null, null ); }

  TreeNode( E anElt, TreeNode lt, TreeNode rt)
    { elt = anElt; left = lt; right = rt; }
```

```
// pre:  this.left == null
void attachLeft( TreeNode lt )
  { if ( lt != null ) left = lt.copy( ); }


// pre:  this.right == null
void attachRight( TreeNode rt )
  { if ( rt != null ) right = rt.copy( ); }
```

```java
// creates a new TreeNode, identical to this

TreeNode copy( )
{
  TreeNode root = new TreeNode ( elt );
  if ( left != null )
    root.left = left.copy();
  if ( right != null )
    root.right = right.copy();
  return root;
}
```

```java
public String toString()
{
  return(elt.toString());
}
```

```
  void preOrder(Processing p){
    p.process(elt);
    if (left != null)
      left.preOrder(p);
    if (right != null)
      right.preOrder(p);
  }
} // End of class TreeNode
```

```
  void inOrder(Processing p){
    p.process(elt);
    if (left != null)
      left.inOrder(p);
    if (right != null)
      right.inOrder(p);
  }
} // End of class TreeNode
```

```java
// Constructors

  public BinaryTree()
    { root = null; }

  public BinaryTree( E rootElt )
    { root = new TreeNode( rootElt ); }

  public BinaryTree( E rElt, BinaryTree lt, BinaryTree rt )
  {
    root = new TreeNode( rElt );
    if ( lt != null )
      root.attachLeft( lt.root );
    if ( rt != null )
      root.attachRight( rt.root );
  }
```

```java
public boolean isEmptyTree( )
  { return root == null; }

// Accessor for the root element
public E getRootElt()
  { return root.elt; }
```

```java
public BinaryTree getLeftTree( )
{
  if ( this == null )
    throw new IllegalArgumentException
      ( "Attempt to apply leftTree to the empty tree" );
  else
  {
    BinaryTree t = new BinaryTree();
    if ( root.left != null )
      t.root = root.left.copy();
    return t;
  }
}
```

```
public BinaryTree getRightTree( )
{
  if ( this == null )
    throw new IllegalArgumentException
      ( "Attempt to apply rightTree to the empty tree" );
  else
  {
    BinaryTree t = new BinaryTree( );
    if ( root.right != null )
      t.root = root.right.copy( );
    return t;
  }
}
```

```java
public void preOrder(Processing p){
  if (root != null)
    root.preOrder(p);
  else
    p.handleEmpty();
}
```

```java
public void inOrder(Processing p){
  if (root != null )
    root.inOrder(p);
  else
    p.handleEmpty();
}
```

```
public int height(){
  return rootHeight(root); }
```

```
private int rootHeight(TreeNode t){
  if ( t == null )
    return -1;
  else
    return 1+Math.max(rootHeight(t.left), rootHeight(t.right));
} }// End of BinaryTree
```

Processing Nodes in a Binary Tree

```java
package BinaryTree;

public interface Processing<E> {
  // A processing object can apply its process to
  // an element.
  public void process( E elt );
  public void handleEmpty();
}
```

One obvious way of processing a node in a binary tree is to print out the value of the element in the node:

```java
package BinaryTree;

public class PrintElement<E> implements Processing
{
  public void process( E e )
  {
    System.out.print( e.toString() + " ");
  }
  public void handleEmpty(){
    System.out.println("No elements to print.");
  }
}
```

Another way of processing the nodes is simply to count them:

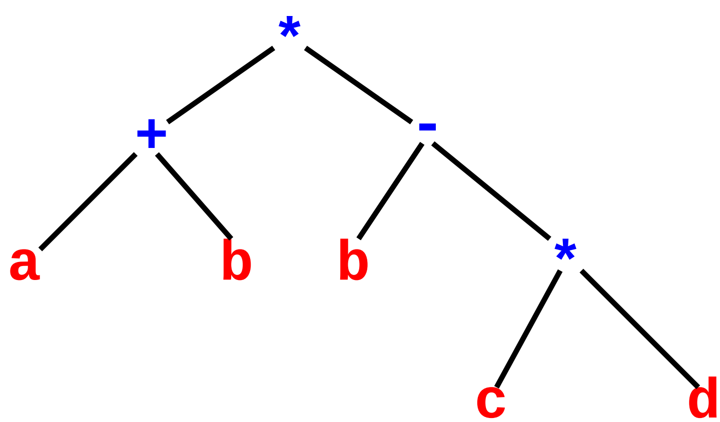
```
package BinaryTree

public class CountNodes<E> implements Processing
{
  static int count = 0;

  public void process( E e )

    { count++;}

  public void handleEmpty(){

    ;//nothing to do

  }

  public String toString() { return "Count = " + count;}
}
```

Representing Arithmetic Expressions: an application of Binary Trees

- An arithmetic expression may be represented by a binary tree called an *expression tree*.
- For example, the arithmetic expression

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

has the following expression tree:



Postorder traversal gives *postfix* or *reverse polish* expression:

$$ab{+}bcd * - *$$