Binary Trees

EECS2030: Advanced
Object Oriented Programming
Fall 2017

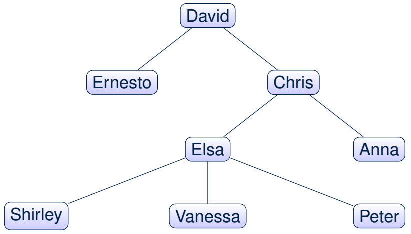
CHEN-WEI WANG

General Trees

- A *linear* data structure is a sequence, where stored objects can be related via the "*before*" and "*after*" relationships.

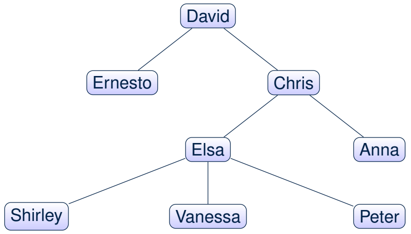
  e.g., arrays, singly-linked lists, and doubly-linked lists
- A *tree* is a *non-linear* collection of nodes.
  - Each node stores some data object.
  - Nodes stored in a *tree* is organized in a *non-linear* manner.
  - In a *tree*, the relationships between stored objects are *hierarchical* : some objects are "*above*" others, and some are "*below*" others.
- The main terminology for the *tree* data structure comes from that of family trees: parents, siblings, children, ancestors, descendants.

General Trees: Terminology (1)



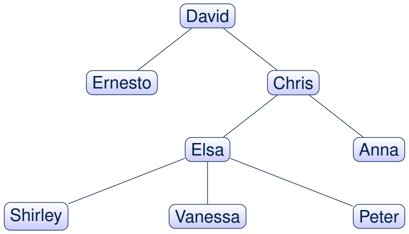
- *root of tree* : top element of the tree
  e.g., *root* of the above family tree: David
- *parent of node v* : node immediately above and connected to *v*
  e.g., *parent* of Vanessa: Elsa
- *children of node v* : nodes immediately below and connected to *v*
  e.g., *children* of Elsa: Shirley, Vanessa, and Peter
  e.g., *children* of Ernesto: ∅

General Trees: Terminology (2)



- *ancestors of node v* : *v* + *v*'s parent + *v*'s grand parent + . . .
  e.g., *ancestors* of Vanessa: *Vanessa*, Elsa, Chris, and David
  e.g., *ancestors* of David: David
- *descendants of node v* : *v* + *v*'s children + *v*'s grand children + . . .
  e.g., *descendants* of Vanessa: Vanessa
  e.g., *descendants* of David: the entire family tree

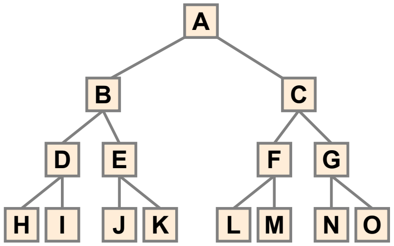
General Trees: Terminology (3)



- *siblings of node v* : nodes whose parents are the same as *v*'s
  e.g., *siblings* of Vanessa: Shirley and Peter
- *subtree rooted at v* : a tree formed by all descendant of *v*
- *external nodes (leaves)* : nodes that have no children
  e.g., *leaves* of the above tree: Ernesto, Anna, Shirley, Vanessa, Peter
- *internal nodes* : nodes that has at least one children
  e.g., *non-leaves* of the above tree: David, Chris, Elsa

Exercise: Identifying Subtrees

How many subtrees are there?

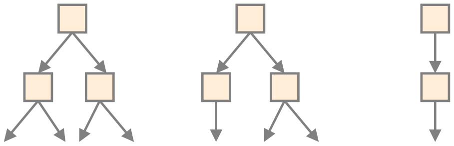


15 subtrees [i.e., subtrees rooted at each node]

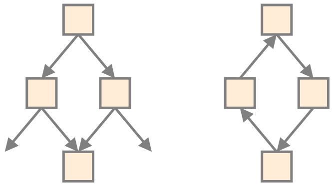
SIZE OF SUBTREE	ROOTS OF SUBTREES
1	H, I, J, K, L, M, N, O
3	D, E, F, G
7	B, C
15	A

General Tree: Important Characteristics

There is a *single unique path* along the edges from the *root* to any particular node.



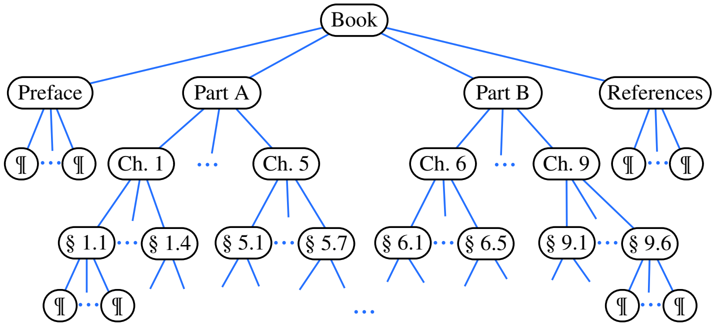
legal tree organization



illegal tree organization (nontrees)

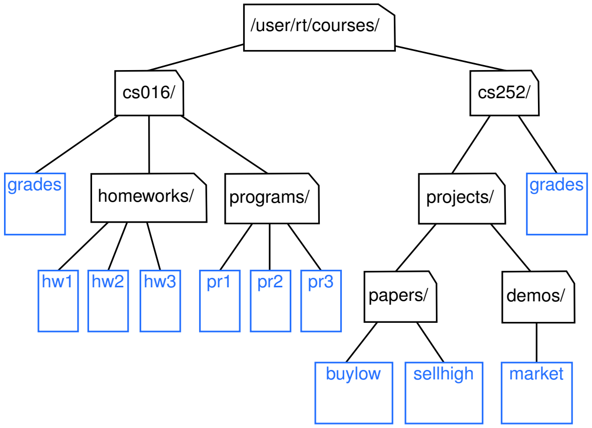
General Trees: Ordered Trees

A tree is *ordered* if there is a meaningful *linear* order among the *children* of each node.



General Trees: Unordered Trees

A tree is *unordered* if the order among the *children* of each node does not matter.



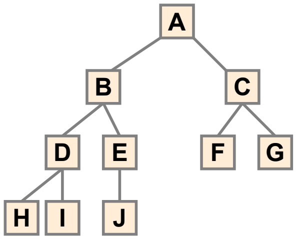
Binary Trees

- A *binary tree* is an *ordered* tree which satisfies the following properties:
  1. Each node has *at most two* children.
  2. Each child node is labeled as either a *left child* or a *right child*.
  3. A *left child* precedes a *right child* in the order of children of a node.

Binary Trees: Terminology (1)

For an *internal* node $\boxed{n}$:

- Subtree rooted at its *left child* is called *left subtree*.
  $\boxed{n}$ has no left child ⇒ $\boxed{n}$'s left subtree is **empty**
- Subtree rooted at its *right child* is called *right subtree*.
  $\boxed{n}$ has no right child ⇒ $\boxed{n}$'s right subtree is **empty**



$\boxed{\text{A}}$'s *left subtree* is rooted at $\boxed{\text{B}}$ and *right subtree* rooted at $\boxed{\text{C}}$.
$\boxed{\text{H}}$'s *left subtree* and *right subtree* are both empty.

Binary Trees: Recursive Definition

A *binary* tree is either:

- An *empty* tree; or
- A *nonempty* tree with a root node $\boxed{r}$ that
  - has a *left binary subtree* rooted at its left child
  - has a *right binary subtree* rooted at its right child

⇒ To solve problems *recursively* on a binary tree rooted at $\boxed{r}$:

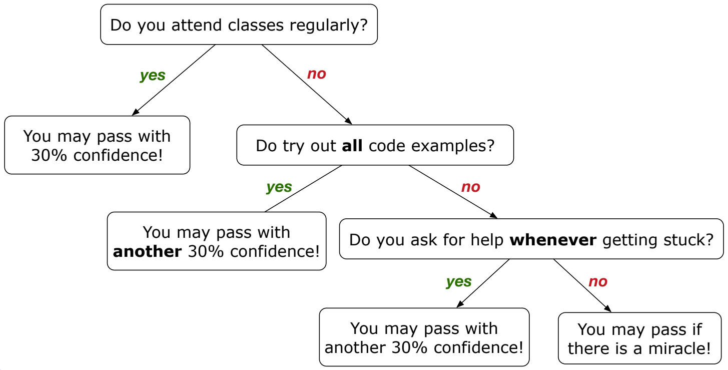
- Do something with root $\boxed{r}$.
- Recur on $\boxed{r}$'s ***left subtree***. [strictly smaller problem]
- Recur on $\boxed{r}$'s ***right subtree***. [strictly smaller problem]

Similar to how we *recur on* ***subarrays*** (by passing the `from` and `to` indices), we *recur on* ***subtrees*** by passing their $\boxed{\text{roots}}$ (i.e., the current root's left child and right child).

Binary Trees: Application (1)

A *decision tree* is a binary tree used to to express the decision-making process:

- Each *internal node* has two children (yes and no).
- Each *external node* represents a decision.



Do you attend classes regularly?

yes → You may pass with 30% confidence!

no → Do try out **all** code examples?

yes → You may pass with **another** 30% confidence!

no → Do you ask for help **whenever** getting stuck?

yes → You may pass with another 30% confidence!

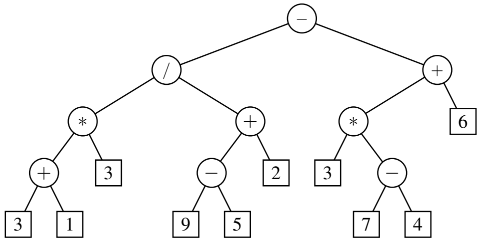
no → You may pass if there is a miracle!

Binary Trees: Application (2)

An *arithmetic expression* can be represented using a binary tree:

- Each *internal node* denotes an operator (unary or binary).
- Each *external node* denotes an operand (i.e., a number).
  e.g., Use a binary tree to represent the arithmetic expression

`( ((3 + 1) * 3) / ((9 − 5) + 2) ) − ( (3 * (7 − 4)) + 6 )`



- To print, or to evaluate, the expression that is represented by a binary tree, certain *traversal* over the entire tree is required.

Tree Traversal Algorithms: Definition

- A *traversal* of a tree T is a systematic way of visiting **all** the nodes of T.
- The visit of each node may be associated with an action: e.g.,
  - print the node element
  - determine if the node element satisfies certain property
  - accumulate the node element value to some global counter

Tree Traversal Algorithms: Common Types

- **Inorder**: Visit left subtree, then parent, then right subtree.

```
inorder (r) : if(r != null) {/*subtree with root r is not empty*/
  inorder (r's left child)
  visit and act on the subtree rooted at r
  inorder (r's right child) }
```

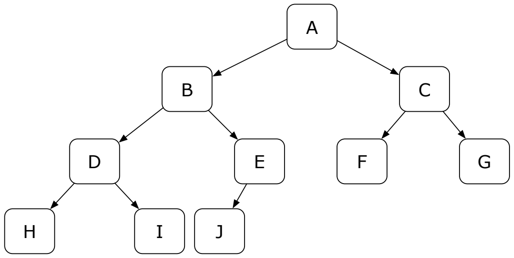
- **Preorder**: Visit parent, then left subtree, then right subtree.

```
preorder (r) : if(r != null) {/*subtree with root r is not empty*/
  visit and act on the subtree rooted at r
  preorder (r's left child)
  preorder (r's right child) }
```

- **Postorder**: Visit left subtree, then right subtree, then parent.

```
postorder (r) : if(r != null) {/*subtree with root r is not empty*/
  postorder (r's left child)
  postorder (r's right child)
  visit and act on the subtree rooted at r }
```

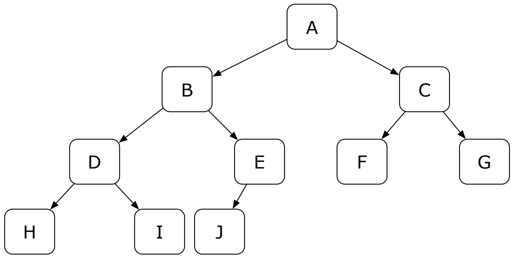
Tree Traversal: Inorder



inorder traversal from the root `A`:

$$\underbrace{\underbrace{\underbrace{\underbrace{H}_{inorder(H)}\ \boxed{D}\ \underbrace{I}_{inorder(I)}}_{inorder(D)}\ \boxed{B}\ \underbrace{\underbrace{J}_{inorder(J)}\ \boxed{E}}_{inorderE}}_{inorderB}\ \boxed{A}\ \underbrace{\underbrace{F}_{inorder(F)}\ \boxed{C}\ \underbrace{G}_{inorder(G)}}_{inorder(C)}}_{inorder(A)}$$

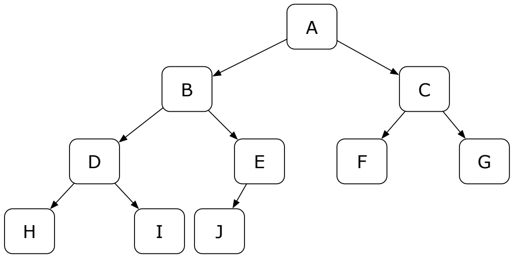
Tree Traversal: Preorder



preorder traversal from the root `A`:

$$\underbrace{\boxed{A}\ \underbrace{\boxed{B}\ \underbrace{\boxed{D}\ \underbrace{H}_{preorder(H)}\ \underbrace{I}_{preorder(I)}}_{preorder(D)}\ \underbrace{\boxed{E}\ \underbrace{J}_{preorder(J)}}_{preorder(E)}}_{preorder(B)}\ \underbrace{\boxed{C}\ \underbrace{F}_{preoder(F)}\ \underbrace{G}_{preoreder(G)}}_{preorder(C)}}_{preorder(A)}$$

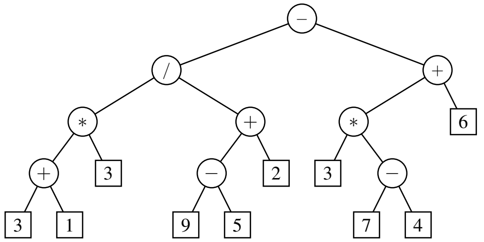
Tree Traversal: Postorder



postorder traversal from the root `A`:

$$\underbrace{\underbrace{\underbrace{\underbrace{H}_{postorder(H)}\ \underbrace{I}_{postorder(I)}\ \boxed{D}}_{postorder(D)}\ \underbrace{\underbrace{J}_{postorder(J)}\ \boxed{E}}_{postorder(E)}\ \boxed{B}}_{postorder(B)}\ \underbrace{\underbrace{F}_{postorder(F)}\ \underbrace{G}_{postorder(G)}\ \boxed{C}}_{postorder(C)}\ \boxed{A}}_{postorder(A)}$$

Tree Traversal: Exercises



- ***inorder*** traversal from the root:
  `3+1⋆3/9−5+2−3⋆7−4+6`
- ***preorder*** traversal from the root:
  `−/⋆+313+−952+⋆3−746`
- ***postorder*** traversal from the root:
  `31+3⋆95−2+/374−⋆6+−`

Binary Tree in Java: Linked Node

```
public class BTNode {
 private String element;
 private BTNode left;
 private BTNode right;

 BTNode(String element) {
  this.element = element;
 }

 public String getElement() { return element; }
 public BTNode getLeft() { return left; }
 public BTNode getRight() { return right; }

 public void setElement(String element) { this.element = element; }
 public void setLeft(BTNode left) { this.left = left; }
 public void setRight(BTNode right) { this.right = right; }
}
```

Binary Tree in Java: Root Note

```
public class BinaryTree {
 private BTNode root;

 public BinaryTree() {
  /* Initialize an empty binary tree with root being null. */
 }

 public void setRoot(BTNode root) {
  this.root = root;
 }

 ...
}
```

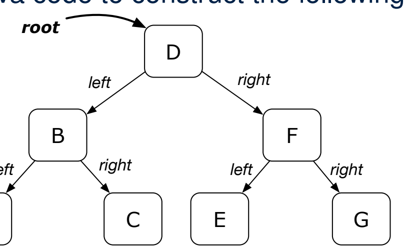
Binary Tree in Java: Adding Nodes (1)

```
public class BinaryTree {
 private BTNode root;
 public void addToLeft(BTNode n, String element) {
  if(n.getLeft() != null) {
   throw new IllegalArgumentException("Left is already there");
  }
  n.setLeft(new BTNode(element));
 }
 public void addToRight(BTNode n, String element) {
  if(n.getRight() != null) {
   throw new IllegalArgumentException("Right is already there");
  }
  n.setRight(new BTNode(element));
 }
}
```

- The way we implement the `add` methods is ***not*** recursive.
- These two `add` methods assume that the caller calls them by *passing references* of the ***parent nodes***.

Binary Tree in Java: Adding Nodes (2)

Exercise: Write Java code to construct the following binary tree:



```
BinaryTree bt = new BinaryTree(); /* empty binary tree */
BTNode root = new BTNode("D"); /* node disconnected from BT */
bt.setRoot(root); /* node connected to BT */
bt.addToLeft(root, "B");
bt.addToRight(root, "F");
bt.addToLeft(root.getLeft(), "A");
bt.addToRight(root.getLeft(), "C");
bt.addToLeft(root.getRight(), "E");
bt.addToRight(root.getRight(), "G");
```

Binary Tree in Java: Counting Size (1)

Size of a tree rooted at *r* is 1 (counting *r* itself) plus the size of *r*'s left subtree and plus the size of *r*'s right subtree.

```
public class BinaryTree {
  private BTNode root;
  public int size() { return sizeHelper(root); }
  private int sizeHelper(BTNode root) {
    if(root == null) {
      return 0;
    }
    else {
      return
          1
        + sizeHelper(root.getLeft())
        + sizeHelper(root.getRight());
    }
  }
}
```

Binary Tree in Java: Counting Size (2)

```
@Test
public void testBTSize() {
  BinaryTree bt = new BinaryTree();
  assertEquals(0, bt.size());

  BTNode root = new BTNode("D");
  bt.setRoot(root);
  assertEquals(1, bt.size());

  bt.addToLeft(root, "B");
  bt.addToRight(root, "F");
  bt.addToLeft(root.getLeft(), "A");
  bt.addToRight(root.getLeft(), "C");
  bt.addToLeft(root.getRight(), "E");
  bt.addToRight(root.getRight(), "G");
  assertEquals(7, bt.size());
}
```

Binary Tree in Java: Membership (1)

An element *e* exists in a tree rooted at *r* if either *r* contains *e*, or *r*'s left subtree contains *e*, or *r*'s right subtree contains *e*.

```
public class BinaryTree {
  private BTNode root;

  public boolean has(String e) { return hasHelper(root, e); }
  private boolean hasHelper(BTNode root, String e) {
    if(root == null) {
      return false;
    }
    else {
      return
           root.getElement().equals(e)
        || hasHelper(root.getLeft(), e)
        || hasHelper(root.getRight(), e);
    }
  }
}
```

Binary Tree in Java: Membership (2)

```
@Test
public void testBTMembership() {
  BinaryTree bt = new BinaryTree();
  assertFalse(bt.has("D"));
  BTNode root = new BTNode("D");
  bt.setRoot(root);
  assertTrue(bt.has("D"));
  assertFalse(bt.has("A"));
  bt.addToLeft(root, "B");
  bt.addToRight(root, "F");
  bt.addToLeft(root.getLeft(), "A");
  bt.addToRight(root.getLeft(), "C");
  bt.addToLeft(root.getRight(), "E");
  bt.addToRight(root.getRight(), "G");
  assertTrue(bt.has("A")); assertTrue(bt.has("B"));
  assertTrue(bt.has("C")); assertTrue(bt.has("D"));
  assertTrue(bt.has("E")); assertTrue(bt.has("F"));
  assertTrue(bt.has("G"));
  assertFalse(bt.has("H"));
  assertFalse(bt.has("I"));
}
```

Binary Tree in Java: Inorder Traversal (1)

```
public class BinaryTree {
  private BTNode root;

  public ArrayList<String> inorder() {
    ArrayList<String> list = new ArrayList<>();
    inorderHelper(root, list);
    return list;
  }
  private void inorderHelper(BTNode root, ArrayList<String> list) {
    if(root != null) {
      inorderHelper(root.getLeft(), list);
      list.add(root.getElement());
      inorderHelper(root.getRight(), list);
    }
  }
}
```

Binary Tree in Java: Inorder Traversal (2)

```
@Test
public void testBT_inorder() {
  BinaryTree bt = new BinaryTree();
  BTNode root = new BTNode("D");
  bt.setRoot(root);
  bt.addToLeft(root, "B");
  bt.addToRight(root, "F");
  bt.addToLeft(root.getLeft(), "A");
  bt.addToRight(root.getLeft(), "C");
  bt.addToLeft(root.getRight(), "E");
  bt.addToRight(root.getRight(), "G");
  ArrayList<String> list = bt.inorder();
  assertEquals(list.get(0), "A");
  assertEquals(list.get(1), "B");
  assertEquals(list.get(2), "C");
  assertEquals(list.get(3), "D");
  assertEquals(list.get(4), "E");
  assertEquals(list.get(5), "F");
  assertEquals(list.get(6), "G");
}
```

Binary Tree in Java: Preorder Traversal (1)

```
public class BinaryTree {
  private BTNode root;

  public ArrayList<String> preorder() {
    ArrayList<String> list = new ArrayList<>();
    preorderHelper(root, list);
    return list;
  }
  private void preorderHelper(BTNode root, ArrayList<String> list) {
    if(root != null) {
      list.add(root.getElement());
      preorderHelper(root.getLeft(), list);
      preorderHelper(root.getRight(), list);
    }
  }
}
```

Binary Tree in Java: Preorder Traversal (2)

```
@Test
public void testBT_inorder() {
  BinaryTree bt = new BinaryTree();
  BTNode root = new BTNode("D");
  bt.setRoot(root);
  bt.addToLeft(root, "B");
  bt.addToRight(root, "F");
  bt.addToLeft(root.getLeft(), "A");
  bt.addToRight(root.getLeft(), "C");
  bt.addToLeft(root.getRight(), "E");
  bt.addToRight(root.getRight(), "G");
  ArrayList<String> list = bt.preorder();
  assertEquals(list.get(0), "D");
  assertEquals(list.get(1), "B");
  assertEquals(list.get(2), "A");
  assertEquals(list.get(3), "C");
  assertEquals(list.get(4), "F");
  assertEquals(list.get(5), "E");
  assertEquals(list.get(6), "G");
}
```

Binary Tree in Java: Postorder Traversal (1)

```
public class BinaryTree {
  private BTNode root;

  public ArrayList<String> preorder() {
    ArrayList<String> list = new ArrayList<>();
    postorderHelper (root, list);
    return list;
  }
  private void postorderHelper (BTNode root, ArrayList<String> list) {
    if(root != null) {
      list.add(root.getElement());
      postorderHelper (root.getLeft(), list);
      postorderHelper (root.getRight(), list);
    }
  }
}
```

Binary Tree in Java: Postorder Traversal (2)

```
@Test
public void testBT_inorder() {
  BinaryTree bt = new BinaryTree();
  BTNode root = new BTNode("D");
  bt.setRoot(root);
  bt.addToLeft(root, "B");
  bt.addToRight(root, "F");
  bt.addToLeft(root.getLeft(), "A");
  bt.addToRight(root.getLeft(), "C");
  bt.addToLeft(root.getRight(), "E");
  bt.addToRight(root.getRight(), "G");
  ArrayList<String> list = bt.postorder();
  assertEquals(list.get(0), "A");
  assertEquals(list.get(1), "C");
  assertEquals(list.get(2), "B");
  assertEquals(list.get(3), "E");
  assertEquals(list.get(4), "G");
  assertEquals(list.get(5), "F");
  assertEquals(list.get(6), "D");
}
```

Index (1)

Index (2)

Index (3)