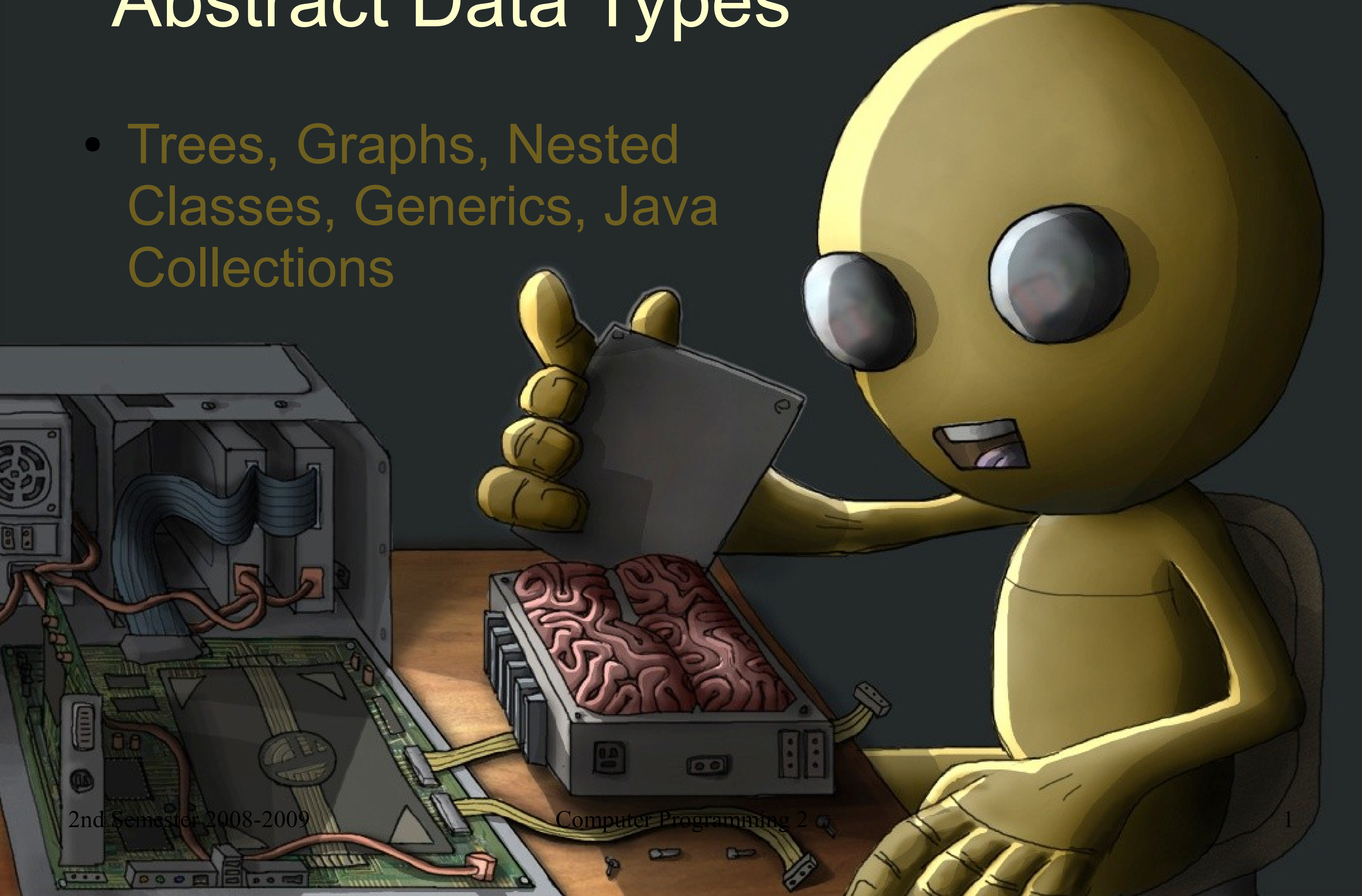
Background taken from: http://64bitllama.deviantart.com/art/The-Dissection-34940057

Abstract Data Types

- Trees, Graphs, Nested Classes, Generics, Java Collections

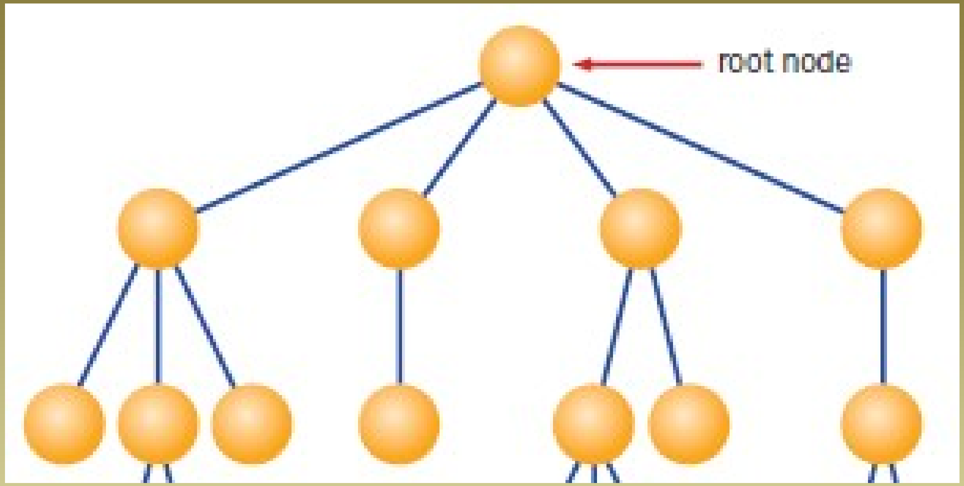


Trees

- Generalization of a linked structure
- Has 2 or more links instead of 1 link
  - Binary Tree – 2 links
- Operations vary depending on application but usually include:
  - Adding a node to the tree
  - Removing a node from the tree

Trees

- Similar to a linked list, contains a reference to the top called the root of the tree
- Recursive structure Tree is either
  - Empty / null
  - Node with N links, all of which are Trees



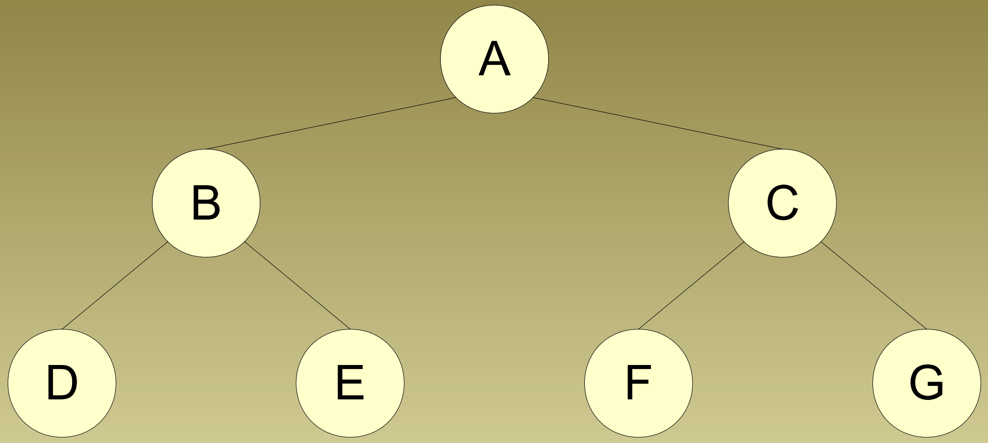
A visual representation of a tree.
The top node is also called the root node.
Java Software Solutions

Binary Trees

- A binary tree is either:
  - Empty / null
  - Binary tree node with left and right subtrees that are also binary trees
- Any binary tree node has
  - at most 2 children / subtrees
  - At most 1 parent / ancestor

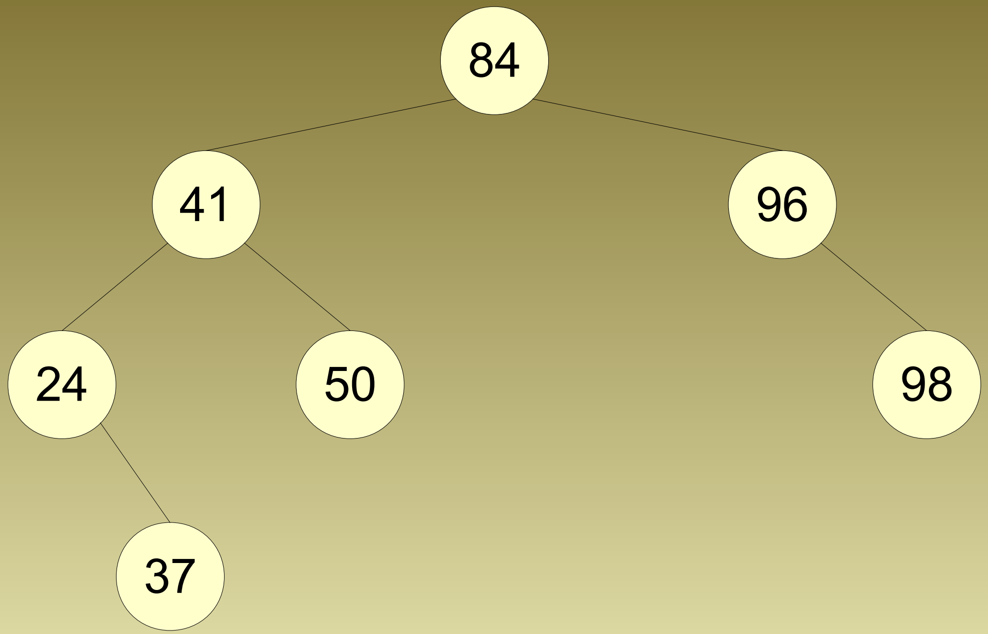
Binary Trees

- Root: A
- Parents: A B C
- Children: B C D E F G
- Leaf: D E F G
- Subtrees: (left root right)
  - (D B E)
  - (F C G)



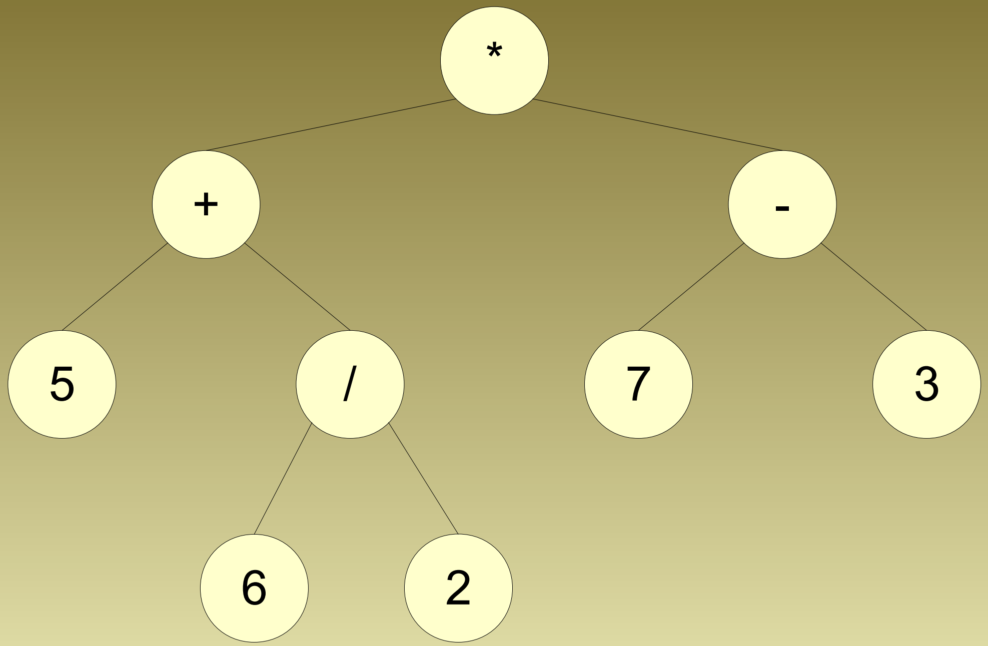
Binary Trees

- Binary Search Tree
  - All items in the left subtree of any node are smaller than the value at that node
  - All items in the right subtree of any node are larger than the value at that node



Binary Trees

- Expression Tree
  - Used to evaluate the value of an expression
  - Left sub tree and right subtree of a node are operands of the operator at that node



Binary Trees

- Implementation
  - Arrays
  - Linked Structures

Binary Trees

- ```
  public class BTNode {
    private BTNode left = null;
    private BTNode right = null;
    private Object data = null;
    public BTNode( Object data ) {
      this.left = null;
      this.right = null;
      this.data = data;
    }
  ```
  ```

Binary Trees

```
public BTNode getLeft() {
  return this.left;
}
public BTNode getRight() {
  return this.right;
}
public Object getData() {
  return this.data;
}
public void setLeft( BTNode left ) {
  this.left = left;
}
```

Binary Trees

```
  public void setRight( BTNode right ) {
    this.right = right;
  }
  public void setData( Object data ) {
    this.data = data;
  }
}
```

Binary Trees

```
– public class BinaryTree {
    private BTNode root;
    public BinaryTree() {
      this.root = null;
    }
    private BinaryTree( BTNode root ) {
      this.root = root;
    }
```

Private helper constructor.

Binary Trees

```
public BinaryTree( BinaryTree leftTree, BinaryTree rightTree,
                   Object data ) {
  root = new BTNode( data );
  if( leftTree != null ) {
    root.setLeft( leftTree.root );
  }
  if( rightTree != null ) {
    root.setRight( rightTree.root );
  }
}
```

Binary Trees

```
public boolean isEmpty() {
  return root == null;
}
public BinaryTree getLeftSubtree() {
  if( isEmpty() ) {
    return null;
  }
  return new BinaryTree( root.getLeft() );
}
```

Binary Trees

```
public BinaryTree getRightSubtree() {
  if( isEmpty() ) {
    return null;
  }
  return new BinaryTree( root.getRight() );
}
public Object getData() {
  if( isEmpty() ) {
    return null;
  }
  return root.getData();
}}
```

Traversals

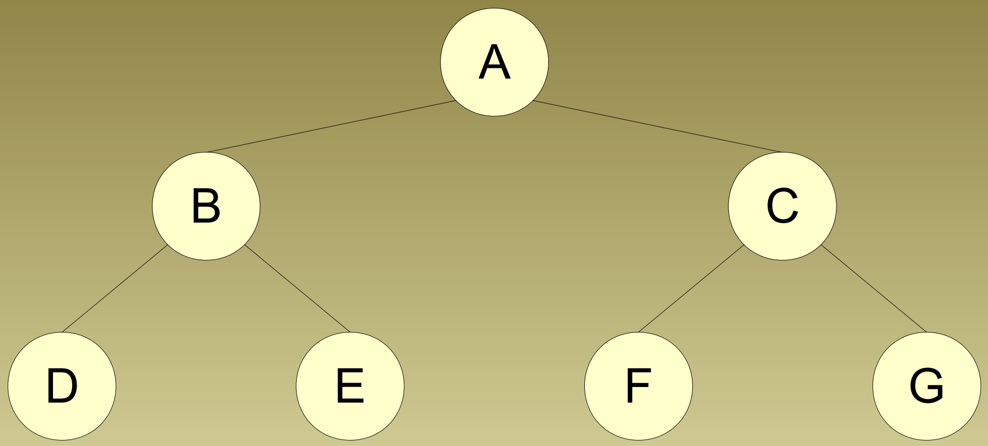
- Preorder
  - Visit the root
  - Perform preorder traversal of the left subtree
  - Perform preorder traversal of the right subtree
- Inorder
  - Perform inorder traversal of the left subtree
  - Visit the root
  - Perform inorder traversal of the right subtree

Traversals

- Postorder
  - Perform postorder traversal of the left subtree
  - Perform postorder traversal of the right subtree
  - Visit the root
- Traversals are recursive!

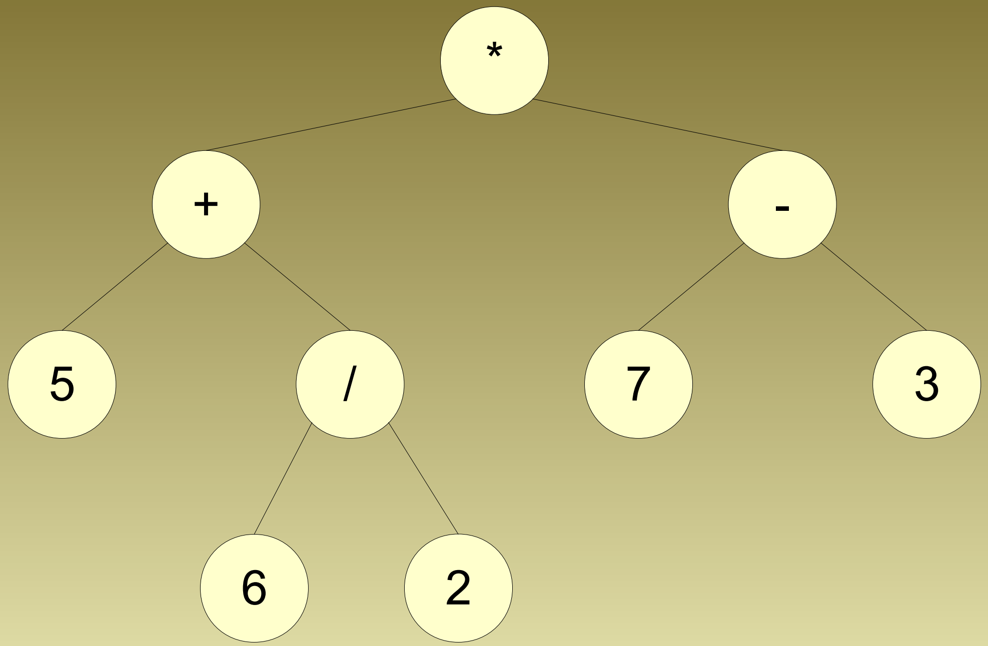
Traversals

- Example:
  - Preorder: A B D E C F G
  - Inorder: D B E A F C G
  - Postorder: D E B F G C A



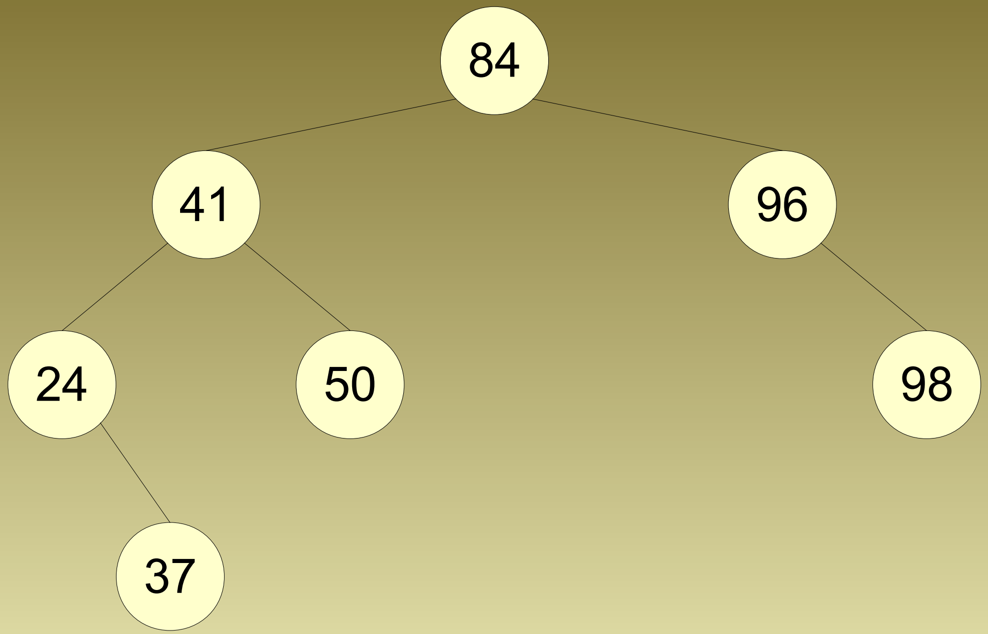
Traversals

- What does the post order traversal of an expression tree yield?



Traversals

- What does the inorder traversal of a binary search tree yield?



Traversals

```
– public String preorder( BinaryTree aTree ) {
    String result = “”;
    if( aTree.isEmpty() ) {
    }
    else {
      result = aTree.root.getData().toString();
      result += preorder( aTree.getLeftSubtree() );
      result += preorder( aTree.getRightSubtree() );
    }
    return result;
  }
```

Binary Trees

- ```
  public String preorder() {
    return preorder( this );
  }
  ```
- ```
  public String preorder2() {
    if( isEmpty() ) {
      return ““;
    }
    else {
      return this.getLeftSubtree().preorder() + this.getData().toString() +
                this.getRightSubtree().preorder();
    }}
  ```

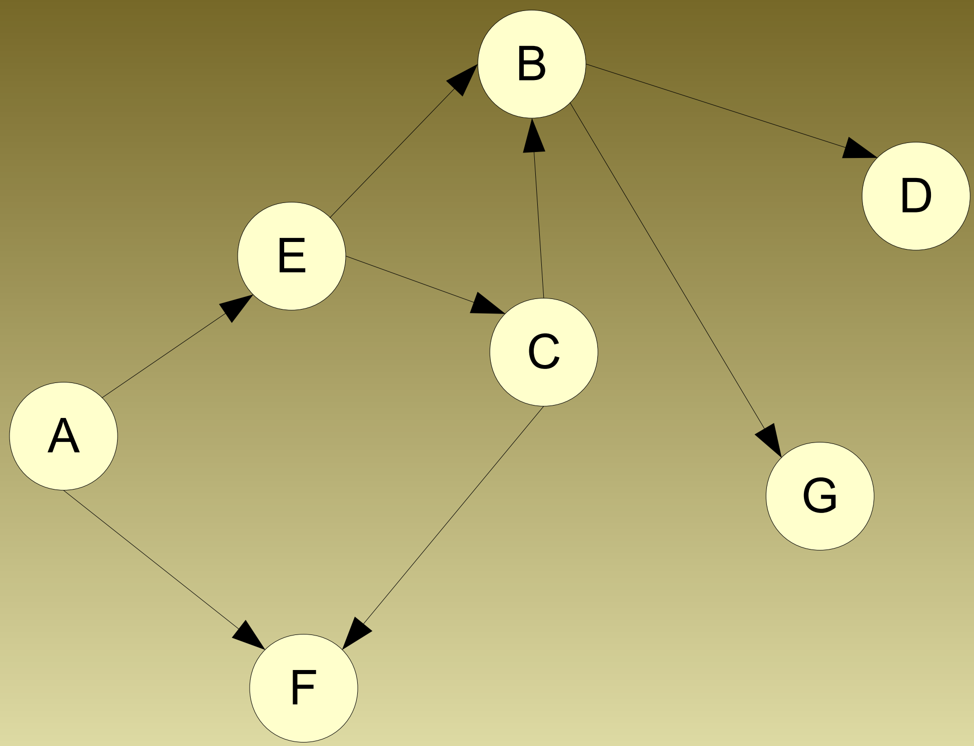
Graphs

- Like a tree but has no primary entry point / root node
- Composed of:
  - Nodes / Vertices
  - Links / Edges
- Generally no restriction on the number of edges that connect nodes

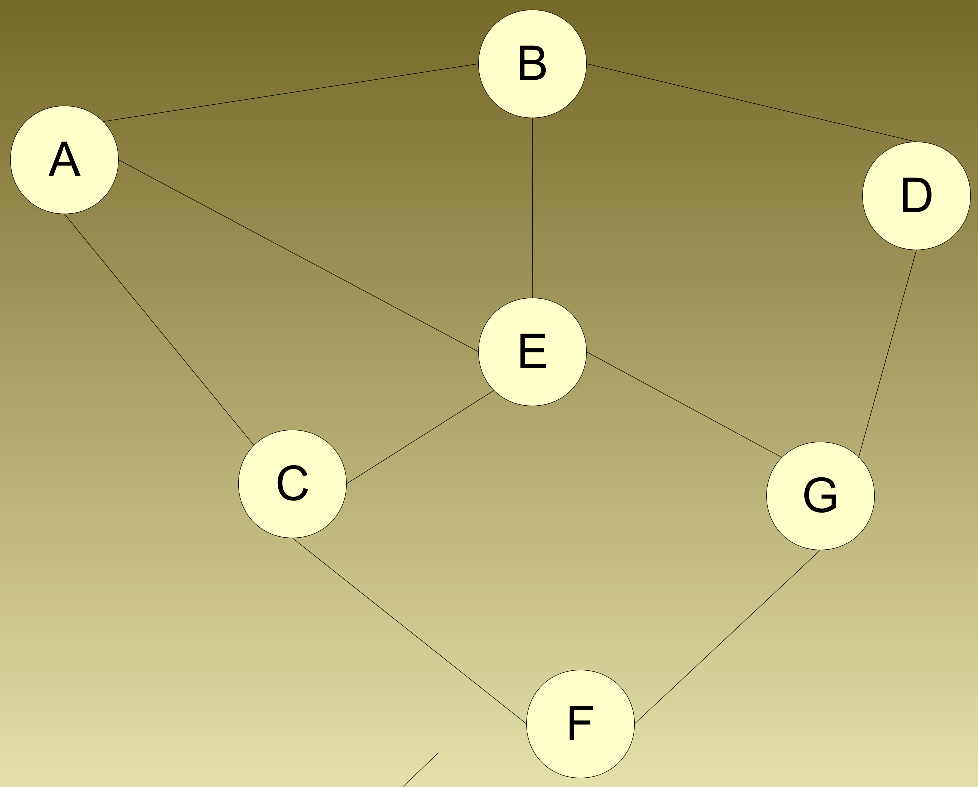
Graphs

- 2 types according to connection:
  - Undirected: if Node 1 is connected to Node 2, Node 2 is connected to Node 1
  - Directed: if Node 1 is connected to Node 2, Node 2 is NOT necessarily connected to Node 1
- Applications:
  - Transportation
  - Networks

Graphs



A directed graph



An undirected graph

Graphs

- Implementation
  - Arrays: use an adjacency matrix
    - Value at row i and column j indicates connection or cost of connection between node i and column j
    - Usually adjMatrix[i][j] > 0 if node i is connected to node j
    - If undirected adjMatrix[i][j] and adjMatrix[j][i] are equal
  - Linked Structures

Graphs

```
– public class ArrayGraph {
    private int adjMatrix[][];
    public ArrayGraph( int numVertices ) {
      adjMatrix = new int[numVertices][numVertices];
    }
    public int getEdge( int node1, int node2 ) {
      return adjMatrix[node1][node2];
    }
    public void setEdge( int node1, int node2, int value ) {
      adjMatrix[node1][node2] = value;
    }
  }
```

Graphs

- Problems:
  - Path finding
    - Node A and Node B are connected by a path if there exists a set of edges that can be followed from Node A to Node B
  - Shortest Path
    - Minimum cost of path from Node A to Node B
  - Minimum Spanning Tree
    - Minimum cost/number of edges needed to connect all vertices in a graph. (i.e. all other vertices are reachable from any vertex)

Nested Classes

- Class can be declared inside a class
  - Similar to how a loop can be declared inside a loop (nested loop)
- Produces separate bytecode file with the following name: EnclosingClass$NestedClass.class
- Enclosing class can access attributes
- Used to put implementation classes inside class which use them (encapsulation)

Nested Classes

- Two types:
  - Inner Classes (Non-static)
    - Associated with an instance of the enclosing class
    - No member inside can be declared static (child can only exist if parent exists)
  - Static nested classes
    - NOT associated with an instance of the enclosing class
    - Cannot access instance (non-static) attributes and functions

Nested Classes

- Redefine BinaryTree:
  - public class BinaryTree {

```
private static class BTNode {
  private Object data = null;
  private BTNode left = null;
  private BTNode right = null;
  private BTNode( Object data ) {
    this.left = null;
    this.right = null;
    this.data = data;
}
```

Use static nested class if enclosing class depends on nested class but not vice versa. BTNode is a SHARED blueprint among BinaryTrees

If class is private, attributes may be declared public without breaking information hiding

Nested Classes

```
}
private BTNode root;
public BinaryTree() {
  root = null;
}
private BinaryTree( BTNode root ) {
  this.root = root;
}
```

End of BTNode class definition. No need for getters and setters since enclosing class can access the attributes directly.

Nested Classes

```
public BinaryTree( BinaryTree leftTree, BinaryTree rightTree,
                   Object data ) {
  root = new BTNode( data );
  if( leftTree != null ) {
    root.left = leftTree.root;
  }
  if( rightTree != null ) {
    root.right = rightTree.right;
  }
}
```

If BTNode were declared not as a nested class, root.left and root.right wouldn't be accessible unless they were declared public

Nested Classes

```
public boolean isEmpty() {
  return root == null;
}
public BinaryTree getRightSubtree() {
  if( isEmpty() ) {
    return null;
  }
  return new BinaryTree( root.right );
}
```

Nested Classes

```
private BinaryTree getLeftSubtree() {
  if( isEmpty() ) {
    return null;
  }
  return new BinaryTree( root.left );
}
public Object getData() {
  if( isEmpty() ) {
    return null;
  }
  return root.data;
}}
```

Generics

- Allow writing safe and easy code without Objects and casts
  - Problems with using Object
    - Need typecast
    - Allow adding any type. No automatic checking
- Write classes that are type customizable
- Particularly useful for abstract data types / collections
- Way to implement/use homogeneous collections

Generics

- Without generics:
  - ArrayList names = new ArrayList();
  - names.add(“Rose”);
  - names.add(“Enzo”);
  - String aName1 = (String) names.get(0);
  - names.add(new Integer(1234567890));
  - String aName2 = (String) names.get(2);

Need to typecast objects retrieved since their type is Object.

Causes a ClassCastException. Since ArrayList uses Object, all types may be added and sometimes they are not the expected type.

Generics

- With generics:
  - ArrayList<String> names = new ArrayList<String>();
  - names.add(“Enzo”);
  - names.add(“Rose”);
  - String aName = names.get(0);
  - names.add(new Integer(1234567890));

A homogenous ArrayList. Can only work on Strings.

No need for a typecast.

Causes a compiler error. Neat.

Generics

- Defining a generic class:
  - Append type variable(s) to class name

    class. Type names can be any valid name but typically use only 1 letter.

    ClassName<type1,type2,...,typeN>
  - Use the type to define:
    - Type of attributes/variables
    - Type of method parameters
    - Method return type
    - Object allocation

Generics

- Example:

```
– public class Pair<T> {
    private T first;
    private T second;
    public Pair( T first, T second ) {
      setFirst( first )
      setSecond( second );
    }
    public T getFirst() {
      return this.first;
    }
```

T used to define type of attributes.

T used to define return type of method.

Generics

```
  public T getSecond() {
    return this.second;
  }
  public void setFirst( T first ) {
    this.first = first;
  }
  public void setSecond( T second ) {
    this.second = second;
  }
}
```

T used to define type of method parameters.

Generics

- Redefining BinaryTree again:

```
– public class BinaryTree<E> {
    private class BTNode<E> {
      private E data = null;
      private BTNode<E> left = null;
      private BTNode<E> right = null;
      private BTNode(E data) {
        this.data = data;
      }
    }
```

Generics

```
private BTNode<E> root;

public BinaryTree() {
  root = null;
}

private BinaryTree( BTNode<E> root ) {
  this.root = root;
}
```

Generics

```
public BinaryTree( BinaryTree<E> leftTree,
                   BinaryTree<E> rightTree<E>, E data ) {
  root = new BTNode<E>(data);
  if( leftTree != null ) {
    root.left = leftTree.root;
  }
  if( rightTree != null ) {
    root.right = rightTree.root;
  }
}
```

Generics

```
public boolean isEmpty() {
  return root == null;
}
public BinaryTree<E> getRightSubtree() {
  if( isEmpty() ) {
    return null;
  }
  return new BinaryTree<E>(root.right);
}
```

Generics

```
public BinaryTree<E> getLeftSubtree() {
  if( isEmpty() ) {
    return null;
  }
  return new BinaryTree<E>(root.left);
}
public E getData() {
  if( isEmpty() ) {
    return null;
  }
    return root.data;
}}
```

Generics

- ```java
  public class GenericTreeTest {
    public static void main( String args[] ) {
      BinaryTree<Integer> left = new BinaryTree<Integer>(null,null,2);
      BinaryTree<Integer> right = new BinaryTree<Integer>(null,null,3);
      BinaryTree<Integer> aTree = new BinaryTree<Integer>(left,right,1);
      System.out.println(aTree.getData());
      System.out.println(aTree.getLeftSubtree().getData());
      System.out.println(aTree.getRightSubtree().getData());
    }
  }
  ```
  ```

Generics

- In general, GenericClass<A> cannot be assigned to GenericClass<B> even if B is a subclass of A (B extends A).

  Example:

  - ArrayList<String> stringList = new ArrayList<String>();
  - ArrayList<Object> objList = stringList; — Compiler will be upset.
- Also, parametized type cannot be a primitive.
  - ArrayList<int> intList = new ArrayList<int>(); — Error. Use wrapper.

Generics

- Proof: (Assume class Charizard extends Pokemon)
  - ArrayList<Charizard> cList = new ArrayList<Charizard>();
  - ArrayList<Pokemon> pList = cList;
  - pList.add(new Pokemon());
  - Charizard aCharzard = cList.get(0);

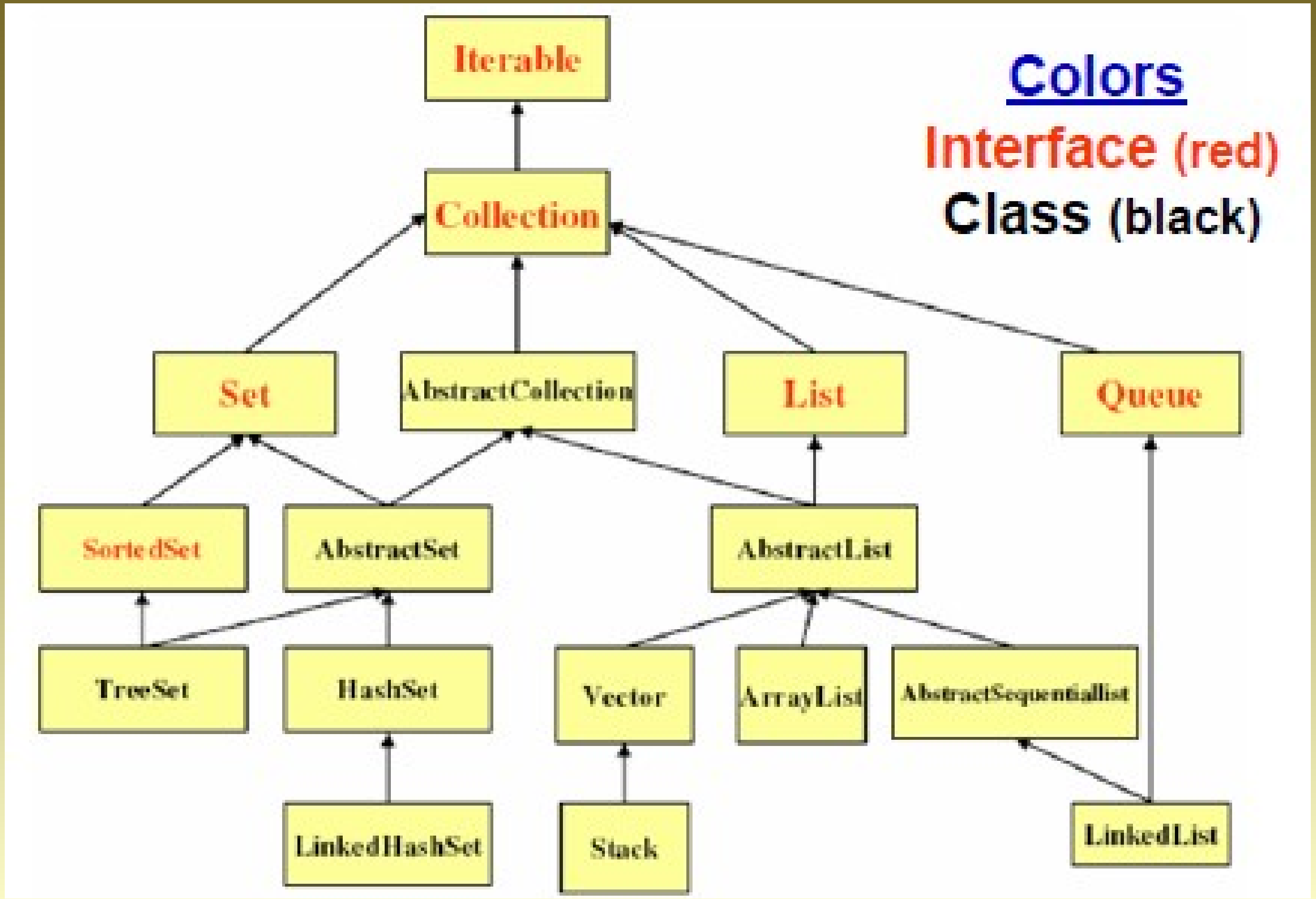
Pokemon will be added to the list of Charizards.

Error! A Pokemon will be retrieved and assigned to a Charizard object but not all Pokemon's are Charizards.

Java Collections

- Generic capable collections
- Consists of:
  - Interfaces/Abstract Classes
    - Define behaviors of a collection
  - Implementation
    - Concrete classes that extend/implement the interfaces
  - Algorithms
    - Generic operations that can be performed on collections

Java Collections



Java Collections Framework Class Hierarchy.

Java Collections

- Separating interface from implementation
  - Use concrete class only when constructing the object. Use interface class to hold the object reference
    - List<Integer> aList1 = new LinkedList<Integer>();
    - List<Double> aList2 = new ArrayList<Double>();
    - Queue<String> aQueue = new LinkedList<String>();

Java Collections

- All collections have an iterator. Use iterator interface to traverse a whole Collection. It has three methods:

```
public interface Iterator<E> {
  E next();
  boolean hasNext();
  void remove();
}
```

An iterator can process an element only after “skipping” (see example)

Java Collections

Example (traversing a collection):

```
Collection<String> aCollection = new LinkedList<String>();
Iterator<String> anIter = aCollection.iterator();
while( anIter.hasNext() ) {
  String curString = anIter.next();
  //do something with curString
  System.out.println( curString );
}
```

Java Collections

Example (traversing a collection using a for each loop):

```
for( String curString: aCollection ) {
  //do something with curString
  System.out.println( curString );
}
```

Read as, for each string in aCollection. Call the current element as curString.

Java Collections

Example (removing an element):

```
anIter.next();
anIter.remove();
```

Example (removing two elements):

```
anIter.next();
anIter.remove();
anIter.next();
anIter.remove();
```

Can't call remove immediately. Must "skip" before removing.

Java Collections

- Collection vs Collections
  - Collection
    - Parent/root interface of collection classes
    - Methods: add, contains, remove, size
  - Collections
    - Class which contains static methods that operate on collections
    - Methods: binarySearch, copy, fill, max, min, sort, shuffle