Binary Trees

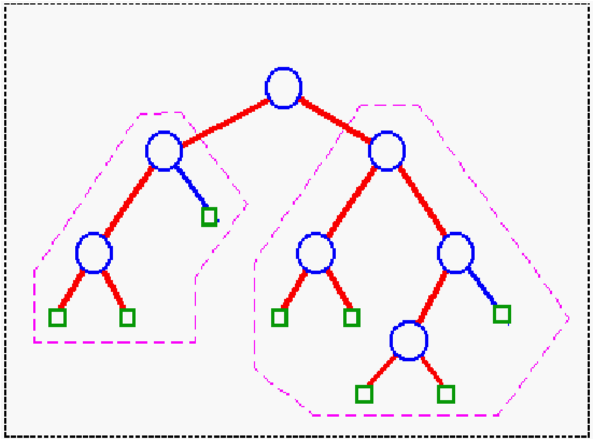
School of Computer Science and Informatics
University College Dublin, Ireland

Binary Tree

- A **Binary Tree** is a tree with the following properties:
  - Every node has at most 2 children (degree 2)
  - Each child node is labelled as being either a **left child** or a **right child**
  - A left child precedes a right child in the ordering of children of a node

- **Proper Binary Tree:** a Binary Tree in which all nodes have degree 0 or 2.

- A Binary Tree, T, is either empty or combines:
  - a node *r*, called the root of T, which stores an element
  - a binary tree, called the left subtree of T
  - a binary tree, called the right subtree of T

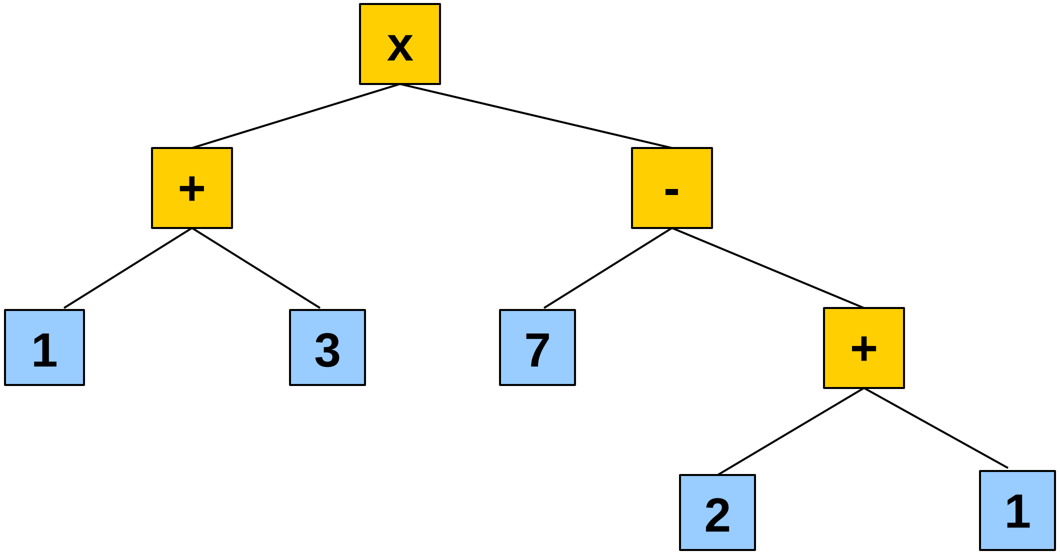


- **Level Property**: Level d of a binary tree is the set of all nodes with depth d, of which there are at most 2^d nodes.

Example: Arithmetic Operations

(1 + 3) x (7 – (2 + 1))

↓



Tree ADT

- Trees make use of the **<u>Position ADT</u>** and have the following operations:
  - **root()** returns the Position for the root
  - **parent(p)** returns the Position of p’s parent
  - **children(p)** returns an Iterator of the Positions of p’s children
  - **isInternal(p)** does p have children?
  - **isExternal(p)** is p a leaf?
  - **isRoot(p)** is p==root()?
  - **size()** number of nodes
  - **isEmpty()** tests whether or not the tree is empty
  - **iterator()** returns an Iterator of every element in the tree
  - **positions()** returns an Iterator of every Position in the tree
  - **replace(p, e)** replaces the element at Position p with e

Tree Interface

```
public interface Tree<T> {
    public Position<T> root();
    public Position<T> parent(Position<T> p);
    public Iterator<Position<T>> children(Position<T> p);
    public boolean isInternal(Position<T> p);
    public boolean isExternal(Position<T> p);
    public boolean isRoot(Position<T> p);
    public int size();
    public boolean isEmpty();
    public Iterator<T> iterator();
    public Iterator<Position<T>> positions();
    public T replace(Position<T> p, T t);
}
```

Binary Tree ADT

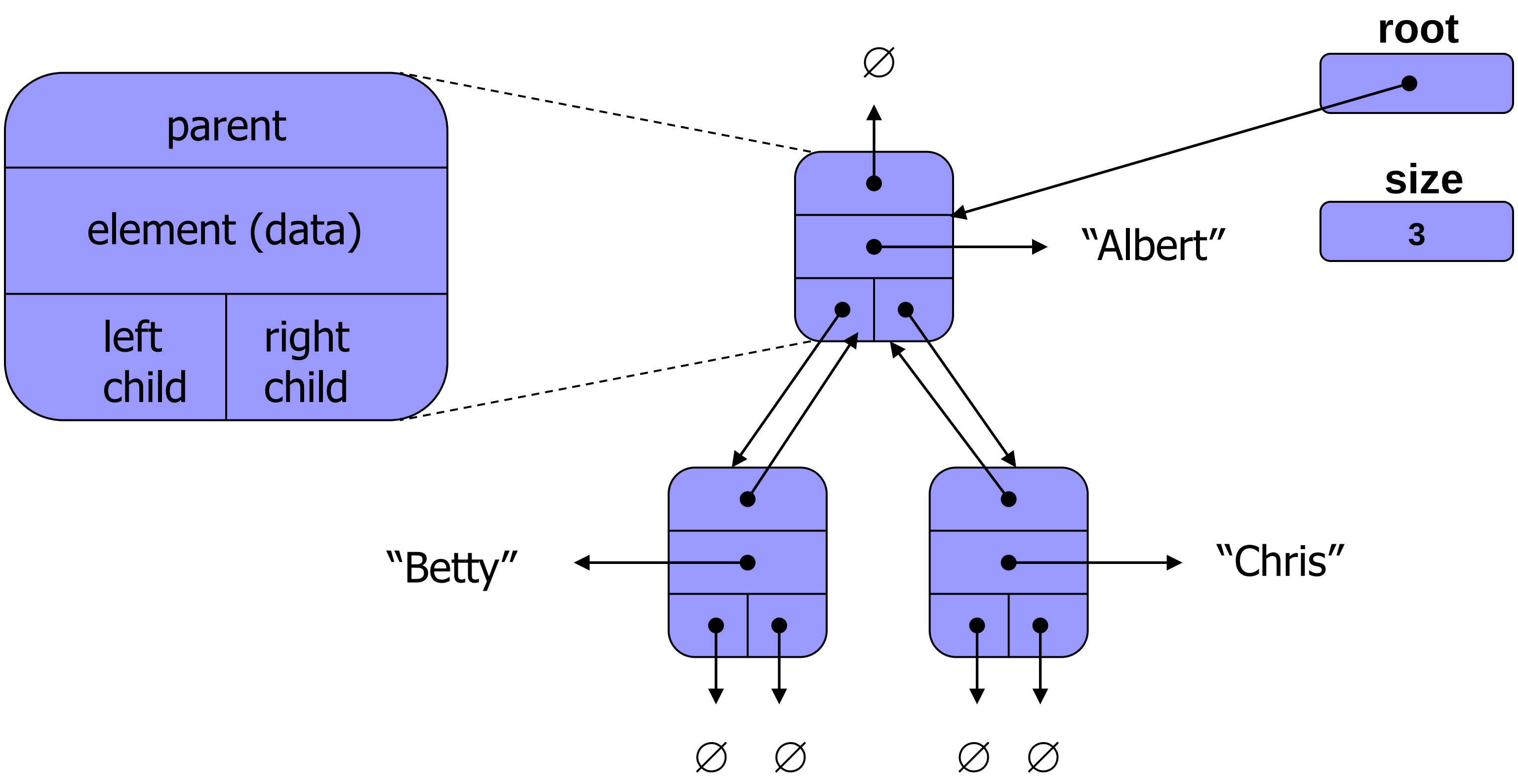
- **Binary Tree ADT** = Tree ADT + 4 extra operations:
  - left(p) return the Position of p's left child
  - right(p) return the Position of p's right child
  - hasLeft(p) test whether p has a left child
  - hasRight(p) test whether p has a right child
- The corresponding Java Interface mirrors this:

```
public interface BinaryTree<T> extends Tree<T> {
    public Position<T> left(Position<T> p);
    public Position<T> right(Position<T> p);
    public boolean hasLeft(Position<T> p);
    public boolean hasRight(Position<T> p);
}
```

- Again, no update methods are provided – these are left to the implementation strategy:
  - links vs arrays; proper vs improper

Link-Based Binary Tree

- Mirrors approach used in linear data types:
  - Nodes contain data (the element)
  - Key Relationships: parent / child (not previous / next)
  - Entry point: The root node
  - Additional Issues: the number of nodes in the tree (size)



Node Implementation

- Implement Node as an inner class:

```
private class Node<T> implements Position<T> {
      Node<T> parent;
      Node<T> left, right;
      T element;

      public Node(T e, Node<T> p) {
            this(e, p, null, null);
      }

      public Node(T e, Node<T> p, Node<T> l, Node<T> r) {
            element = e;
            parent = p;
            left = l;
            right = r;
      }

      public T element() {
            return element;
      }
}
```

Linked Binary Tree: Operations

Update Operations:

- addRoot(e) create and return a new root node storing e; an error should occur if the tree is not empty
- insertLeft(v, e) create and return a new node storing e as the left child of v; an error should occur if v has a left child.
- insertRight(v, e) create and return a new node storing e as the right child of v; an error should occur if v has a right child.
- remove(v) remove node v and replace it with its child, if any, and return the element stored at v; an error occurs if v has two children.
- attach(v, T1, T2) Attach T1 and T2 respectively, as the left and right subtrees of the external node v; an error occurs if v is not external.

LinkedBinaryTree Impl.

```
public class LinkedBinaryTree<T> implements BinaryTree<T> {
   private class Node<T> implements Position<T> { ... }
   private Node<T> root;
   private int size;

   public LinkedBinaryTree() {
        size = 0;
   }

   public Position<T> addRoot(T e) {
        if (size != 0) throw new DuplicateRootException();
        root = new Node<T>(e, null);
        size++;
        return root;
   }

   public Position<T> insertLeft(Position<T> p, T e) {
        Node<T> node = toNode(p);
        if (node.left != null) throw new ChildExistsException();
        node.left = new Node<T>(e, node);
        size++;
        return node.left;
   }
```

LinkedBinaryTree Impl.

```
public Position<T> left(Position<T> p) {
      Node<T> node = toNode(p);
      return node.left;
}

public boolean hasLeft(Position<T> p) {
      Node<T> node = toNode(p);
      return node.left != null;
}

public Position<T> root() {
      return root;
}

public boolean isRoot(Position<T> p) {
      return root == p;
}

public boolean isExternal(Position<T> p) {
      Node<T> node = toNode(p);
      return (node.left == null) && (node.right == null);
}
```

Proper Linked Binary Trees

- For a Proper Binary Tree:
  - Every node, *n*, has degree 0 (external node) or 2 (internal node).
  - Build the tree by expanding external nodes to become internal nodes.
  - Default Impl.: Only internal nodes hold data.
  - Less flexible but can simplify the implementation of data structures

- Key Operations:
  - expandExternal(v, e) — create two new null nodes and add them as the left and right children of v, and store data e at v; an error occurs if v is not external.
  - remove(v) — if the left child is external, remove it and v and replace v with the right child. If the right child is external, remove it and v and replace it with the left child.

ProperLinkedBinaryTree Impl.

```
public class ProperLinkedBinaryTree<T> implements BinaryTree<T> {
   private class Node<T> implements Position<T> { ... }
   private Node<T> root;
   private int size;

   public ProperLinkedBinaryTree(T e) {
         root = new Node<T>(e, null);
         root.left = new Node<T>(null, root);
         root.right = new Node<T>(null, root);
         size = 3;
   }

   public void expandExternal(Position<T> p, T e) {
         if (isInternal(p)) throw new InvalidNodeException();
         Node<T> node = toNode(p);
         node.element = e;
         node.left = new Node<T>(null, node);
         node.right = new Node<T>(null, node);
         size+=2;
   }
```

ProperLinkedBinaryTree Impl.

```
public T remove(Position<T> p) {
  Node node = toNode(p);
  if (isExternal(node.left)) {
    if (node.parent.left == node) {
      node.parent.left = node.right;
      node.right.parent = node.parent.left;
      node.left.parent = null;
    } else {
      node.parent.right = node.right;
      node.right.parent = node.parent.right;
      node.left.parent = null;
    }
    node.left = null;
    node.right = null;
    node.parent = null;
```

ProperLinkedBinaryTree Impl.

```
    } else if (isExternal(node.right)) {
      if (node.parent.left == node) {
        node.parent.left = node.right;
        node.left.parent = node.parent.left;
        node.right.parent = null;
      } else {
        node.parent.right = node.right;
        node.left.parent = node.parent.right;
        node.right.parent = null;
      }
      node.left = null;
      node.right = null;
      node.parent = null;
    } else {
      throw new InvalidNodeException();
    }

    T temp = node.element;
    node.element = null;
    size-=2;

    return temp;
  }

  ...
}
```

Visitor Pattern

- **Design Pattern**: a general reusable solution to a commonly occurring problem within a given context in software design.

- **Visitor Pattern**: a way of separating an algorithm from an object structure on which it operates.
  - Traversal Algorithms are used to visit nodes in a tree.
  - Same algorithm applies to general trees and binary trees.
  - Visitor Pattern allows us to implement once and reuse.

- **Example**: TreePrinter
  - Generates a String representation of a tree using indentation to indicate the level of each node.

Visitor Pattern Implementation

```
public interface TreeVisitor<T> {
    public void visit(Position<T> position, Object data);
}

public class TreePrinter<T> implements TreeVisitor<T> {
    BinaryTree<T> tree;
    String output = “”;

    public TreePrinter(BinaryTree<T> tree) { this.tree = tree; }

    @Override
    public void visit(Position<T> position, Object data) {
        output += data.toString() + position.element() + "\n";
        Iterator<Position<T>> it = tree.children(position);
        while (it.hasNext()) {
            visit(it.next(), data.toString() + "\t");
        }
    }

    public String toString() { return output; }
}
```