Fundamentals of Data Structure

Mahesh Shirole

VJTI, Mumbai-19

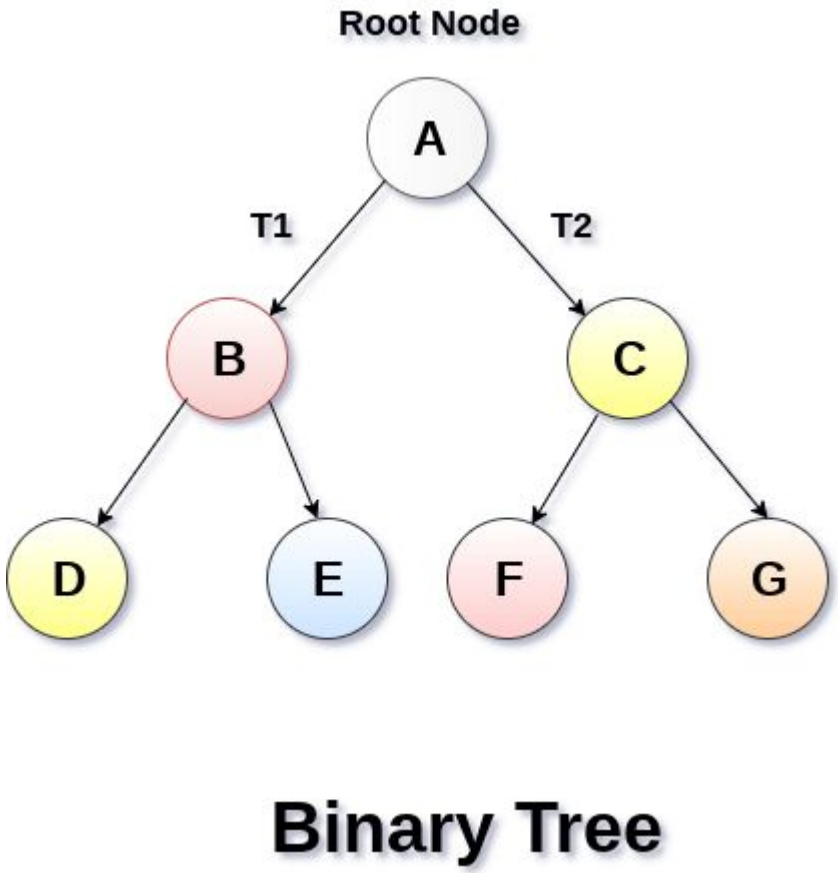
Slides are prepared from

1. Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

2.Data Structures and Algorithms in Java, by Robert Lafore, Second Edition, Sams Publishing

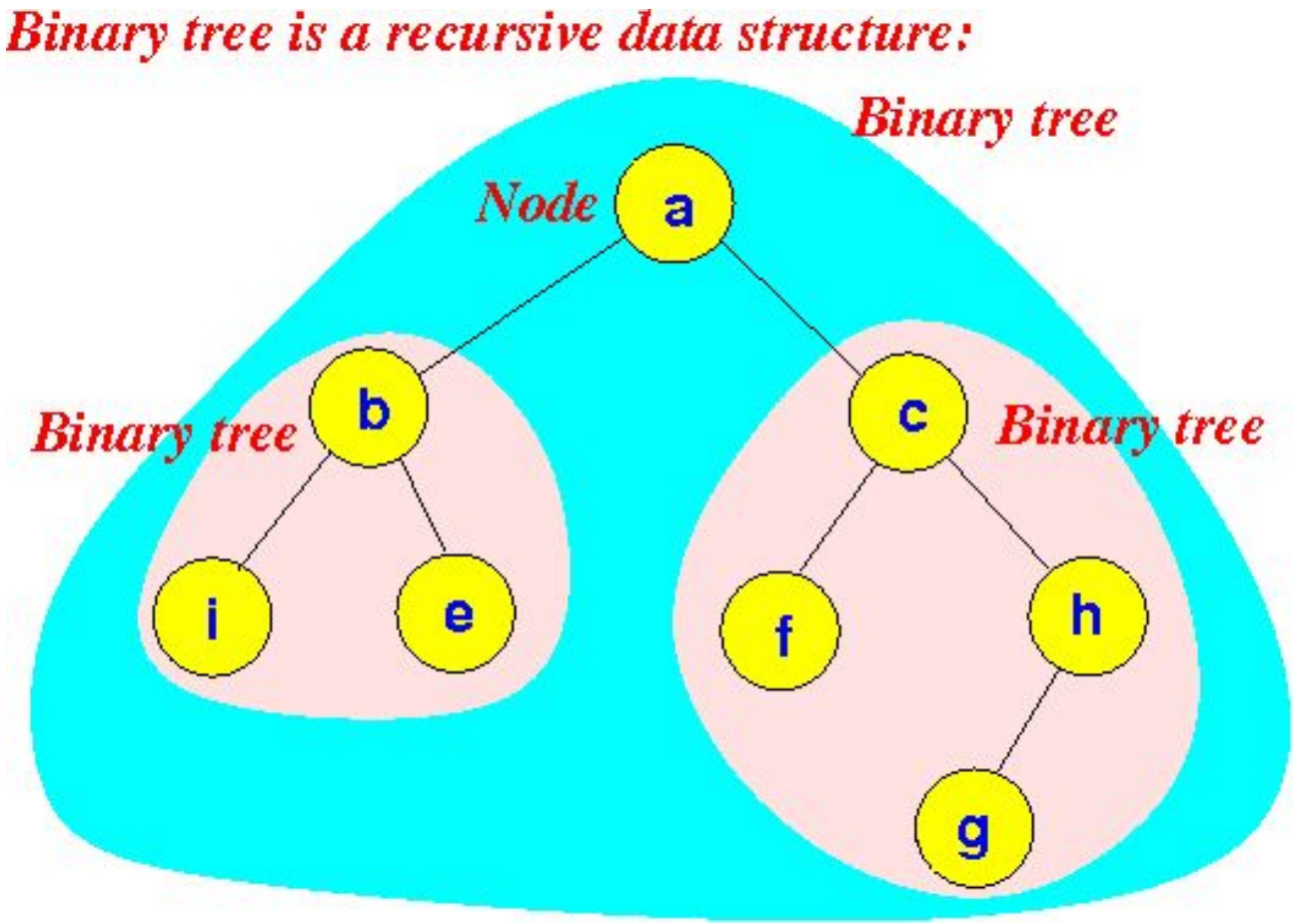
Binary Trees

- A binary tree is an ordered tree with the following properties:
  1. Every node has at most two children
  2. Each child node is labeled as being either a left child or a right child
  3. A left child precedes a right child in the order of children of a node
- The subtree rooted at a left or right child of an internal node v is called a *left subtree* or *right subtree*, respectively, of v
- A binary tree is **proper** if *each node has either zero or two children* it is also called as **full binary trees**
- Thus, in a proper binary tree, every internal node has exactly two children



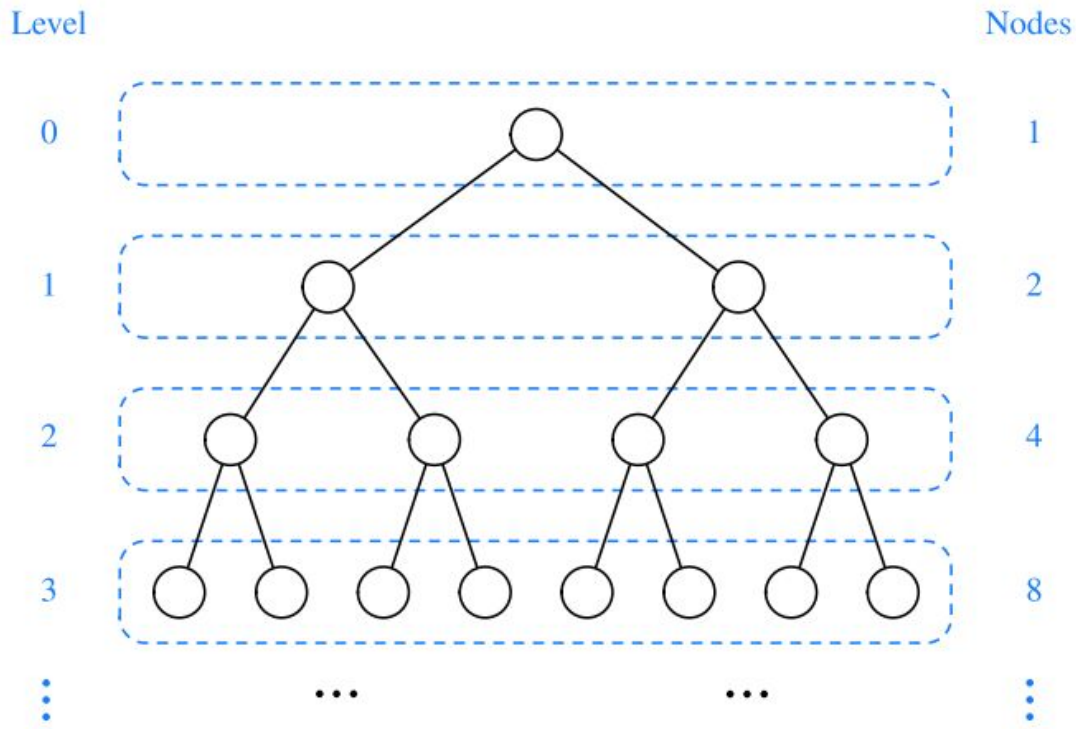
A Recursive Binary Tree Definition

- A binary tree is either:
  - An *empty tree*
  - A *nonempty tree* having a root node r, which stores an element, and *two binary trees that are respectively the left and right subtrees of r*.
- We note that one or both of those subtrees can be empty by this definition



Properties of Binary Trees

- In trees, we denote the set of all nodes of a tree T at the same depth d as ***level d*** of T
- In a binary tree, level 0 has at most one node (the root), level 1 has at most two nodes (the children of the root), level 2 has at most four nodes, and so on.
- In general, level d has at most 2^d nodes
- The *maximum number of nodes* on the levels of a binary tree *grows exponentially* as we go down the tree



Properties of Binary Trees

Proposition 8.7: *Let T be a nonempty binary tree, and let n, n_E, n_I, and h denote the number of nodes, number of external nodes, number of internal nodes, and height of T, respectively. Then T has the following properties:*

1. $h+1 \leq n \leq 2^{h+1}-1$
2. $1 \leq n_E \leq 2^h$
3. $h \leq n_I \leq 2^h-1$
4. $\log(n+1)-1 \leq h \leq n-1$

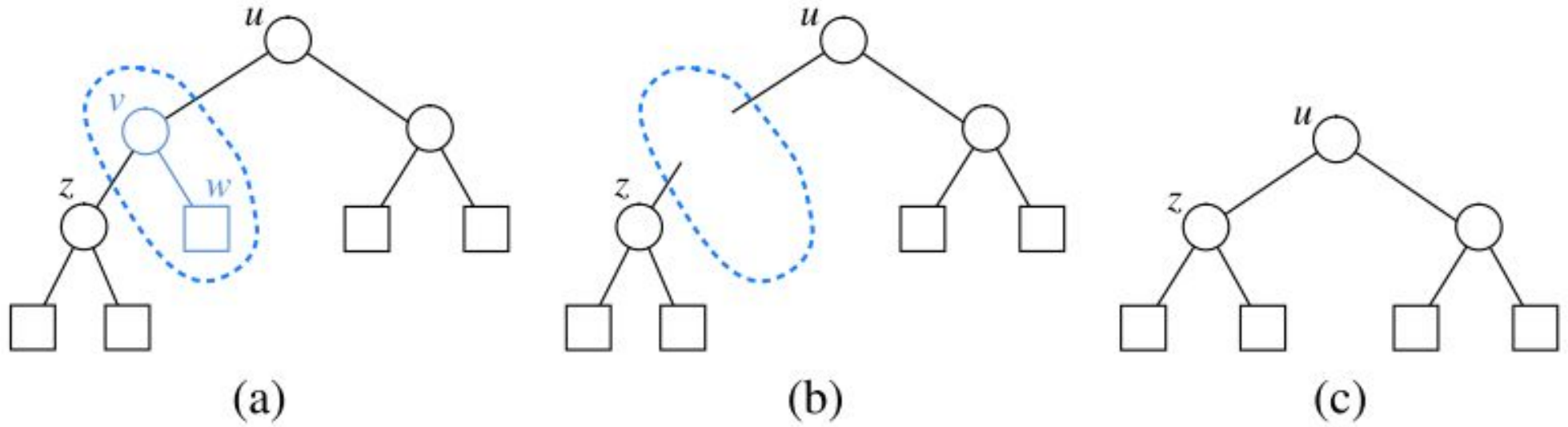
Also, if T is proper, then T has the following properties:

1. $2h+1 \leq n \leq 2^{h+1}-1$
2. $h+1 \leq n_E \leq 2^h$
3. $h \leq n_I \leq 2^h-1$
4. $\log(n+1)-1 \leq h \leq (n-1)/2$

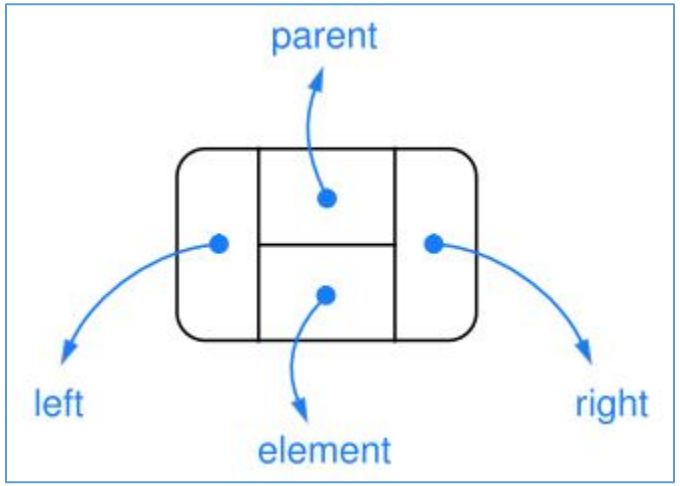
Properties of Binary Trees

Proposition 8.8: *In a nonempty proper binary tree T, with n_E external nodes and n_I internal nodes, we have $n_E = n_I + 1$.*

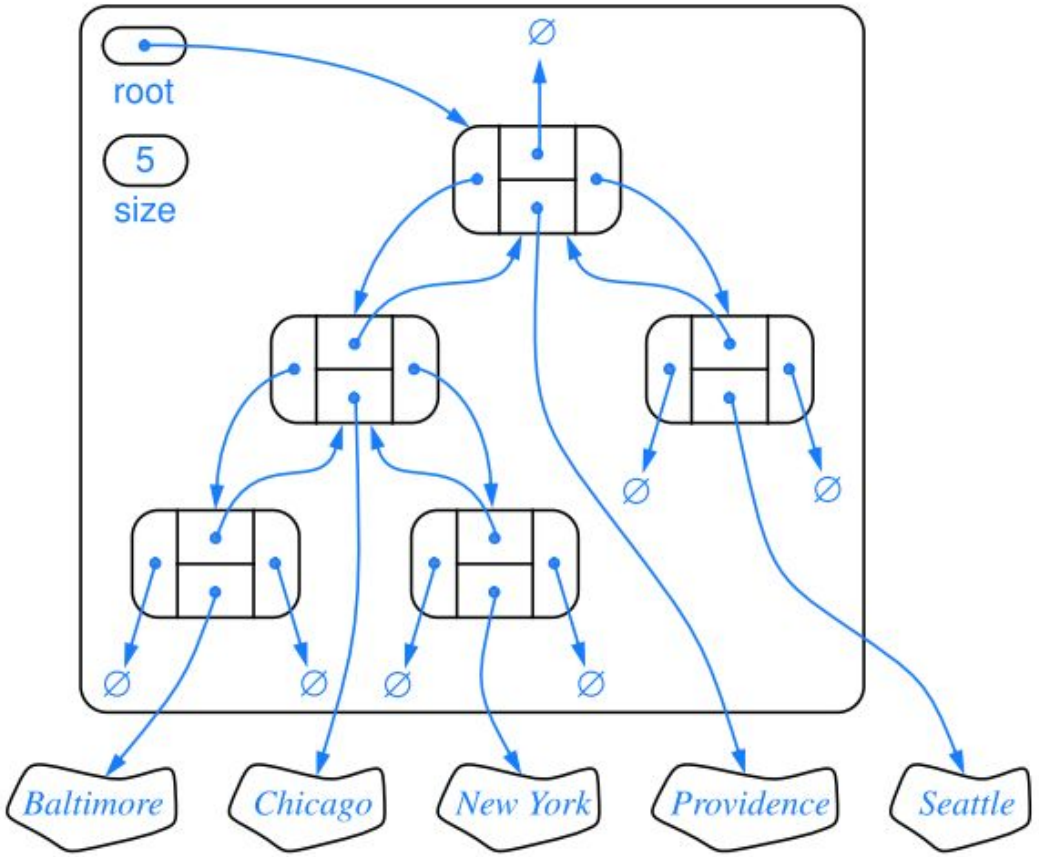
We justify this proposition by removing nodes from T and dividing them up into two “piles,” an internal-node pile and an external-node pile, until T becomes empty



Implementing Binary Trees



- A natural way to realize a binary tree T is to use a linked structure
- Node Structure: maintains references to the ***element stored at a position p*** and to the ***nodes associated with the children*** and ***parent of p***
- If *p is the root of T*, then the *parent field of p is null*
- Similarly, if *p does not have a left child* (respectively, right child), the *associated field is null*
- The tree itself maintains an instance variable storing a reference to the **root node** (if any), and a variable, called **size**, that represents the overall number of nodes of T

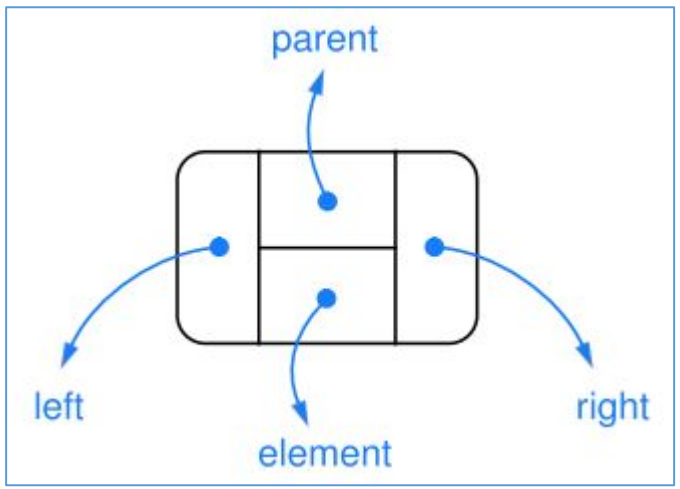


Operations for Updating a Linked Binary Tree

- In the case of a linked binary tree, we suggest that the following update methods be supported:
- **addRoot(e)**: Creates a root for an empty tree, storing ***e*** as the element, and returns the position of that root; an error occurs if the tree is not empty.
- **addLeft(p, e)**: Creates a left child of position ***p***, storing element ***e***, and returns the position of the new node; an error occurs if p already has a left child.
- **addRight(p, e)**: Creates a right child of position ***p***, storing element ***e***, and returns the position of the new node; an error occurs if p already has a right child.
- **set(p, e)**: Replaces the element stored at position ***p*** with element ***e***, and returns the previously stored element.
- **attach(p, T1, T2)**: Attaches the internal structure of trees ***T1*** and ***T2*** as the respective left and right subtrees of leaf position ***p*** and resets ***T1*** and ***T2*** to empty trees; an error condition occurs if ***p*** is not a leaf.
- **remove(p)**: Removes the node at position ***p***, replacing it with its child (if any), and returns the element that had been stored at ***p***; an error occurs if ***p*** has two children

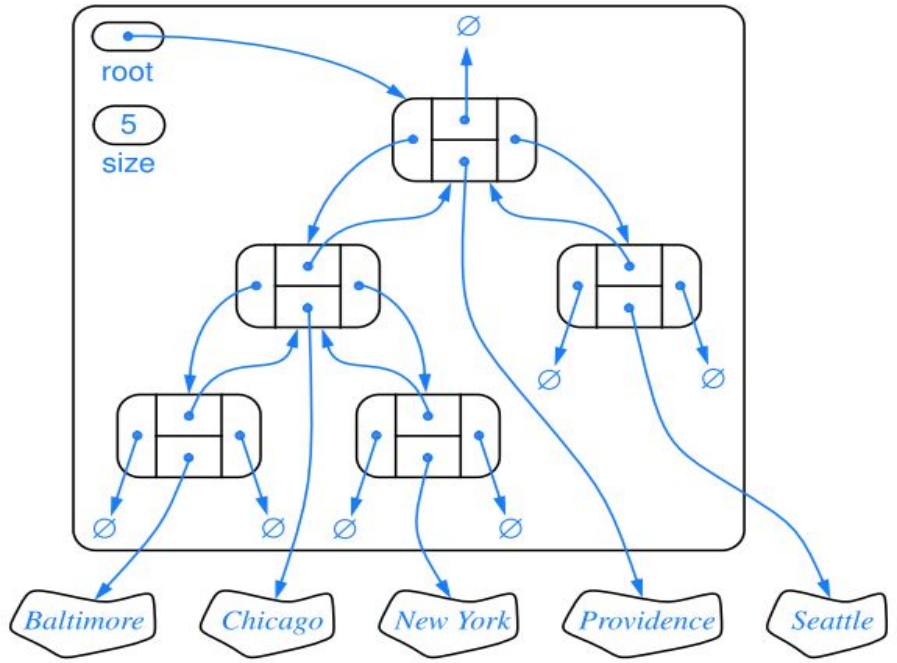
Java Implementation of a Linked Binary Tree Structure

```
//---------------- nested Node class ----------------
protected static class Node<E> implements Position<E> {
  private E element;                    // an element stored at this node
  private Node<E> parent;               // a reference to the parent node (if any)
  private Node<E> left;                 // a reference to the left child (if any)
  private Node<E> right;                // a reference to the right child (if any)
  /** Constructs a node with the given element and neighbors. */
  public Node(E e, Node<E> above, Node<E> leftChild, Node<E> rightChild) {
    element = e;
    parent = above;
    left = leftChild;
    right = rightChild;
  }
  // accessor methods
  public E getElement( ) { return element; }
  public Node<E> getParent( ) { return parent; }
  public Node<E> getLeft( ) { return left; }
  public Node<E> getRight( ) { return right; }
  // update methods
  public void setElement(E e) { element = e; }
  public void setParent(Node<E> parentNode) { parent = parentNode; }
  public void setLeft(Node<E> leftChild) { left = leftChild; }
  public void setRight(Node<E> rightChild) { right = rightChild; }
} //---------- end of nested Node class ----------
```



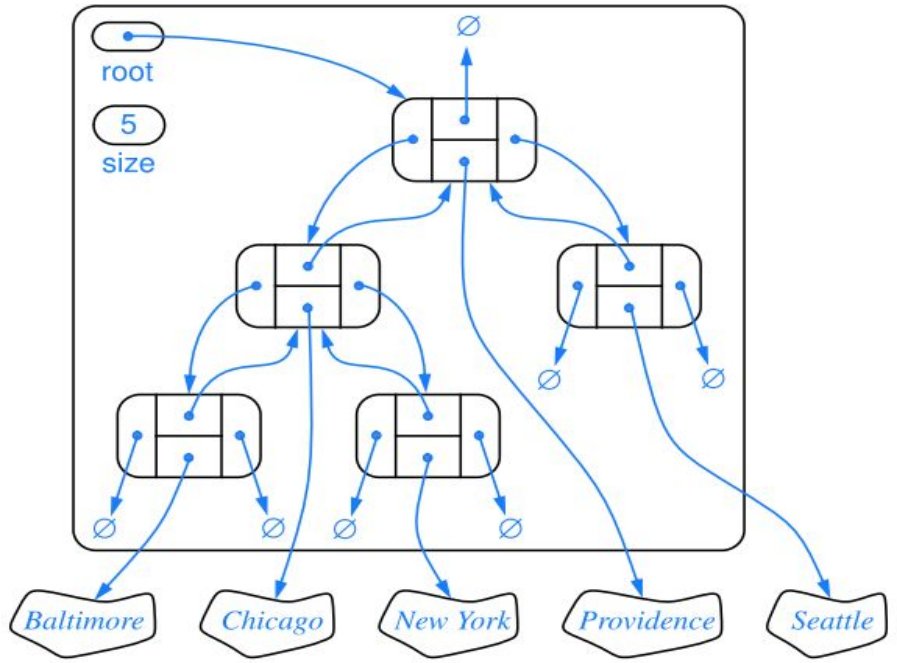
Java Implementation of a Linked Binary Tree Structure

```
/** Concrete implementation of a binary tree using a node-based, linked structure. */
public class LinkedBinaryTree<E> extends AbstractBinaryTree<E> {

  //---------------- nested Node class ----------------

  /** Factory function to create a new node storing element e. */
  protected Node<E> createNode(E e, Node<E> parent,
                               Node<E> left, Node<E> right)
    return new Node<E>(e, parent, left, right);
  }

  // LinkedBinaryTree instance variables
  protected Node<E> root = null;          // root of the tree
  private int size = 0;                   // number of nodes in the tree

  // constructor
  public LinkedBinaryTree( ) { }          // constructs an empty binary tree
```



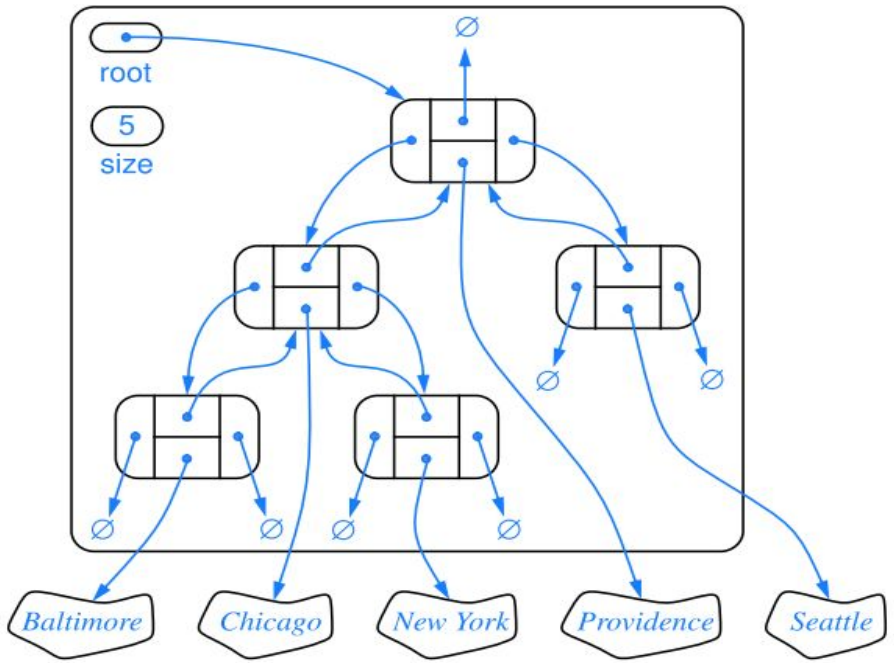
Java Implementation of a Linked Binary Tree Structure

```
// nonpublic utility
/** Validates the position and returns it as a node. */
protected Node<E> validate(Position<E> p) throws IllegalArgumentException {
  if (!(p instanceof Node))
    throw new IllegalArgumentException("Not valid position type");
  Node<E> node = (Node<E>) p;              // safe cast
  if (node.getParent( ) == node)           // our convention for defunct node
    throw new IllegalArgumentException("p is no longer in the tree");
  return node;
}

// accessor methods (not already implemented in AbstractBinaryTree)
/** Returns the number of nodes in the tree. */
public int size( ) {
  return size;
}
```



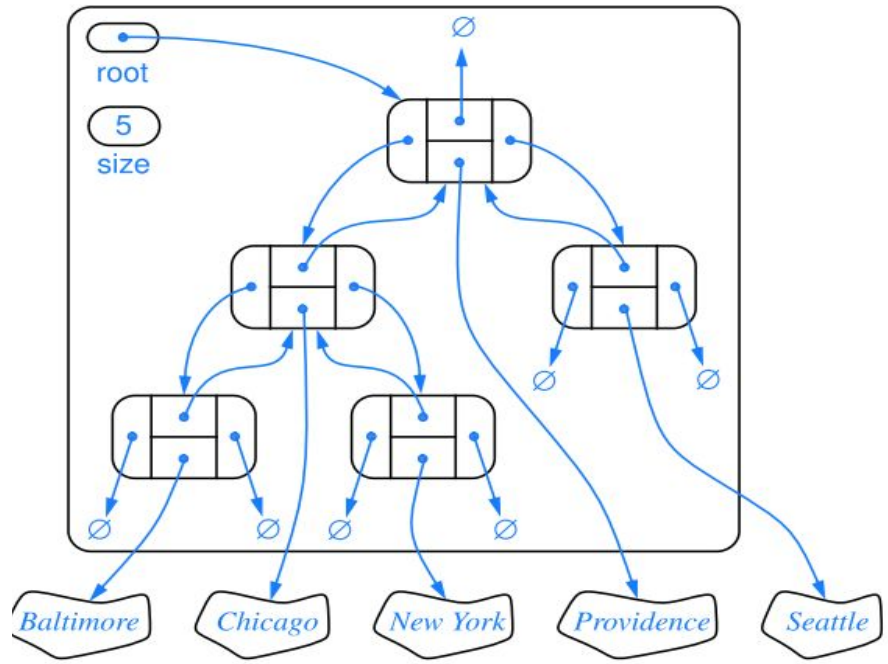
Java Implementation of a Linked Binary Tree Structure

```
/** Returns the root Position of the tree (or null if tree is empty). */
public Position<E> root( ) {
  return root;
}

/** Returns the Position of p's parent (or null if p is root). */
public Position<E> parent(Position<E> p) throws IllegalArgumentException {
  Node<E> node = validate(p);
  return node.getParent( );
}

/** Returns the Position of p's left child (or null if no child exists). */
public Position<E> left(Position<E> p) throws IllegalArgumentException {
  Node<E> node = validate(p);
  return node.getLeft( );
}

/** Returns the Position of p's right child (or null if no child exists). */
public Position<E> right(Position<E> p) throws IllegalArgumentException {
  Node<E> node = validate(p);
  return node.getRight( );
}
```



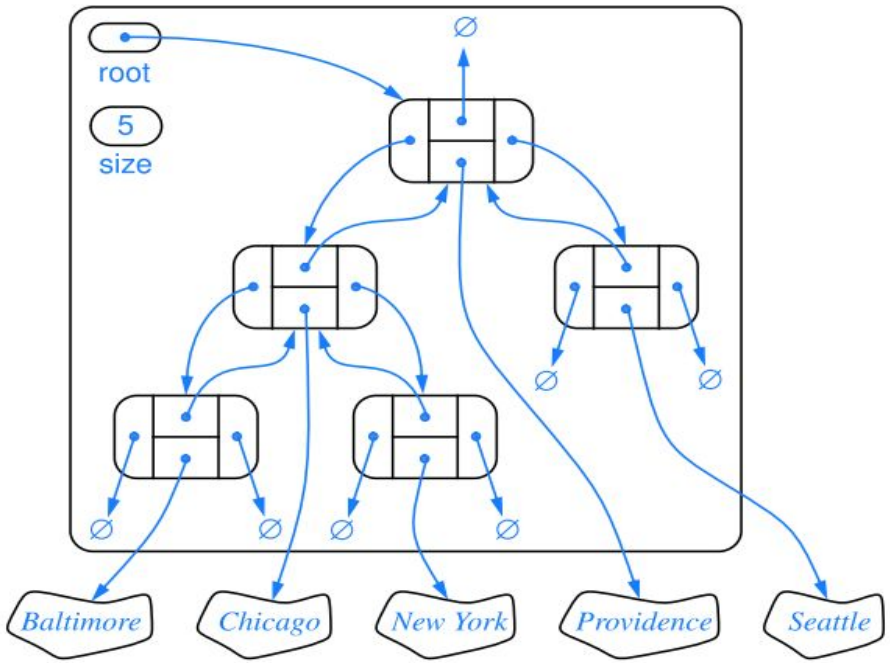
Java Implementation of a Linked Binary Tree Structure

```
// update methods supported by this class
/** Places element e at the root of an empty tree and returns its new Position. */
public Position<E> addRoot(E e) throws IllegalStateException {
  if (!isEmpty( )) throw new IllegalStateException("Tree is not empty");
  root = createNode(e, null, null, null);
  size = 1;
  return root;
}

 /** Replaces the element at Position p with e and returns the replaced element. */
 public E set(Position<E> p, E e) throws IllegalArgumentException {
   Node<E> node = validate(p);
   E temp = node.getElement( );
   node.setElement(e);
   return temp;
 }
```



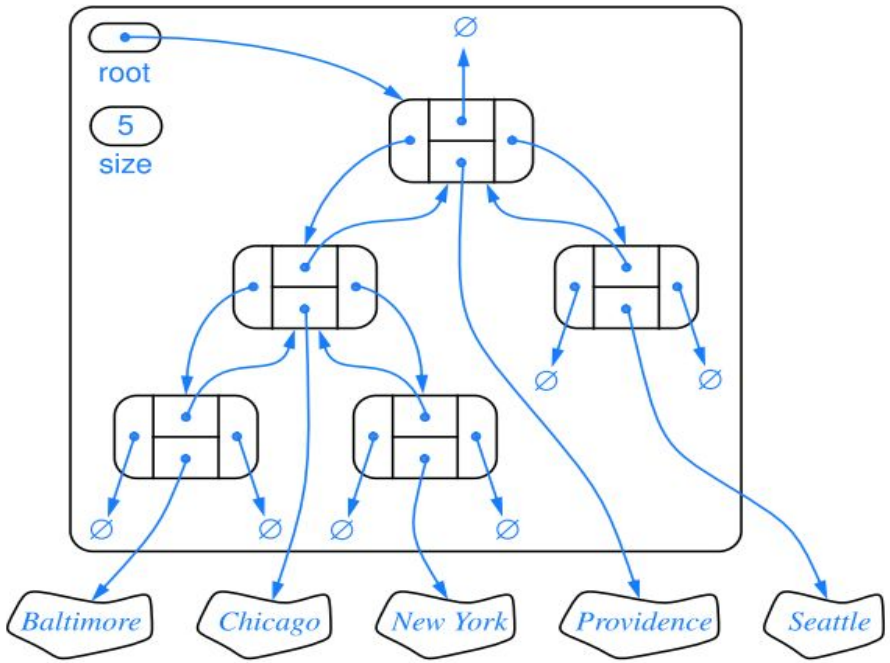
Java Implementation of a Linked Binary Tree Structure

```
/** Creates a new left child of Position p storing element e; returns its Position. */
public Position<E> addLeft(Position<E> p, E e)
                          throws IllegalArgumentException {
  Node<E> parent = validate(p);
  if (parent.getLeft( ) != null)
    throw new IllegalArgumentException("p already has a left child");
  Node<E> child = createNode(e, parent, null, null);
  parent.setLeft(child);
  size++;
  return child;
}

/** Creates a new right child of Position p storing element e; returns its Position. */
public Position<E> addRight(Position<E> p, E e)
                          throws IllegalArgumentException {
  Node<E> parent = validate(p);
  if (parent.getRight( ) != null)
    throw new IllegalArgumentException("p already has a right child");
  Node<E> child = createNode(e, parent, null, null);
  parent.setRight(child);
  size++;
  return child;
}
```



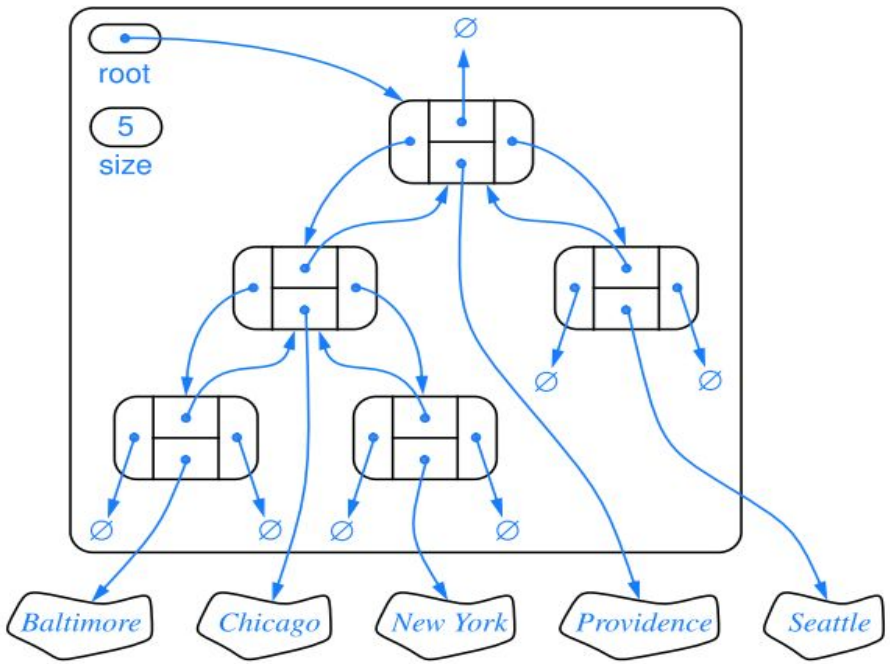
Java Implementation of a Linked Binary Tree Structure

```
/** Attaches trees t1 and t2 as left and right subtrees of external p. */
public void attach(Position<E> p, LinkedBinaryTree<E> t1,
                   LinkedBinaryTree<E> t2) throws IllegalArgumentException {
  Node<E> node = validate(p);
  if (isInternal(p)) throw new IllegalArgumentException("p must be a leaf");
  size += t1.size( ) + t2.size( );
  if (!t1.isEmpty( )) {                    // attach t1 as left subtree of node
    t1.root.setParent(node);
    node.setLeft(t1.root);
    t1.root = null;
    t1.size = 0;
  }
  if (!t2.isEmpty( )) {                    // attach t2 as right subtree of node
    t2.root.setParent(node);
    node.setRight(t2.root);
    t2.root = null;
    t2.size = 0;
  }
}
```



Java Implementation of a Linked Binary Tree Structure

```
/** Removes the node at Position p and replaces it with its child, if any. */
public E remove(Position<E> p) throws IllegalArgumentException {
  Node<E> node = validate(p);
  if (numChildren(p) == 2)
    throw new IllegalArgumentException("p has two children");
  Node<E> child = (node.getLeft( ) != null ? node.getLeft( ) : node.getRight( ) );
  if (child != null)
    child.setParent(node.getParent( ));      // child's grandparent becomes its parent
  if (node == root)
    root = child;                            // child becomes root
  else {
    Node<E> parent = node.getParent( );
    if (node == parent.getLeft( ))
      parent.setLeft(child);
    else
      parent.setRight(child);
  }
  size--;
  E temp = node.getElement( );
  node.setElement(null);                     // help garbage collection
  node.setLeft(null);
  node.setRight(null);
  node.setParent(node);                      // our convention for defunct node
  return temp;
 }
} //---------- end of LinkedBinaryTree class ----------
```



Performance of the Linked Binary Tree Implementation

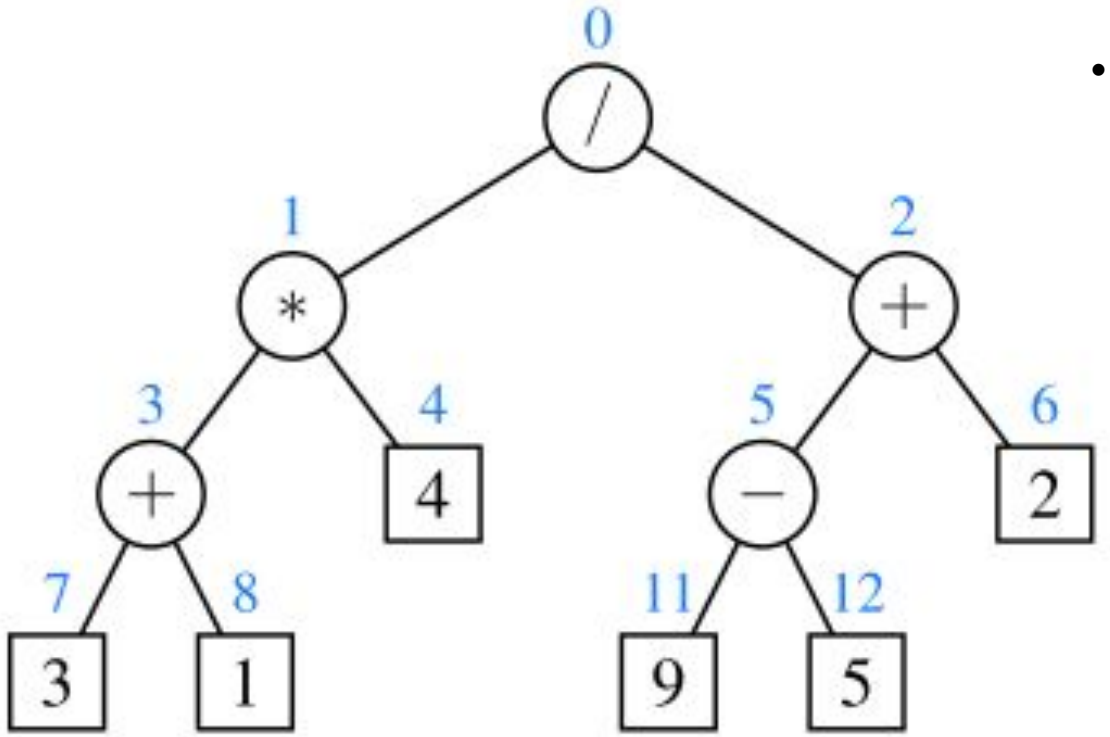
- To summarize the efficiencies of the linked structure representation, we analyze the running times of the LinkedBinaryTree methods, including derived methods that are inherited from the AbstractTree and AbstractBinaryTree classes:
- The overall space requirement of this data structure is O(n), for a tree with '*n*' nodes, as there is an instance of the Node class for every node, in addition to the top-level size and root fields

Method	Running Time
size, isEmpty	$O(1)$
root, parent, left, right, sibling, children, numChildren	$O(1)$
isInternal, isExternal, isRoot	$O(1)$
addRoot, addLeft, addRight, set, attach, remove	$O(1)$
depth(p)	$O(d_p + 1)$
height	$O(n)$

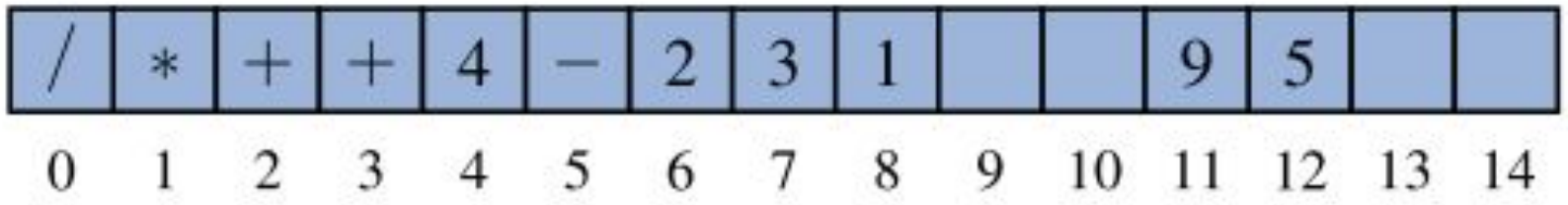
Array-Based Representation of a Binary Tree

- An alternative representation of a binary tree ***T*** is based on a way of numbering the positions of ***T***
- For every position ***p*** of T, let ***f(p)*** be the integer defined as follows
  - If ***p*** is the root of ***T***, then ***f(p) = 0***
  - If ***p*** is the left child of position ***q***, then ***f(p) = 2 f(q)+1***
  - If ***p*** is the right child of position ***q***, then ***f(p) = 2 f(q)+2***
- The numbering function ***f*** is known as a level numbering of the positions in a binary tree ***T***
- The level numbering is based on potential positions within a tree, not the actual shape of a specific tree, so they are not necessarily consecutive

Array-Based Representation of a Binary Tree



- Based on our formula for the level numbering,
  - the left child of ***p*** has index ***2 f(p)+1,***
  - the right child of ***p*** has index ***2 f(p)+2***, and
  - the parent of ***p*** has index ***⌊(f(p)−1)/2⌋***



Array-Based Representation of a Binary Tree

- The space usage of an array-based representation depends greatly on the shape of the tree
- Let ***n*** be the number of nodes of ***T***, and let f_M be the maximum value of ***f(p)*** over all the nodes of ***T***
- The array *A* requires length $N = 1 + f_M$, since elements range from *A[0] to A[fM]*
- Note that *A* may have a number of empty cells that do not refer to existing positions of ***T***
- In fact, in the worst case, $N = 2^n - 1$