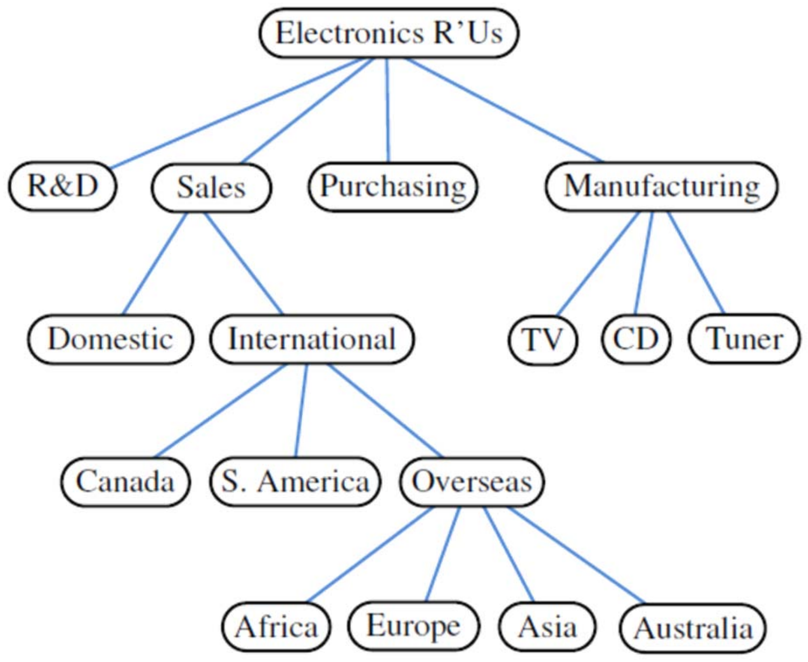
CSE214 Data Structures

Tree Structures

YoungMin Kwon

Trees

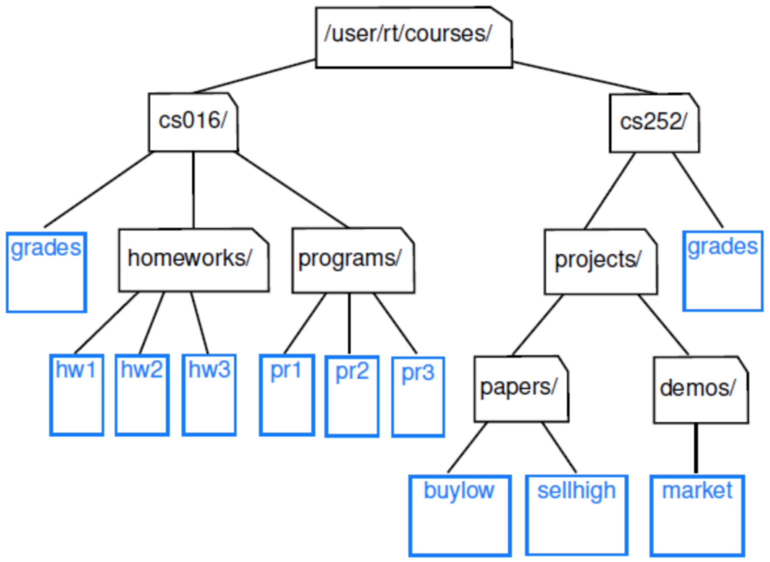
- Tree: an ADT that stores elements hierarchically
- Each element has
  - A parent element
  - Zero or more child elements
- Root: top element



Trees

- Definition
  - A *tree T* is a set of *nodes* storing elements and a *parent-child relation*
  - If T is non-empty, it has a special node, called *root* of T, that has no parent
  - Every non-root node has a unique *parent* w; every node with parent w is a *child* of w

Trees



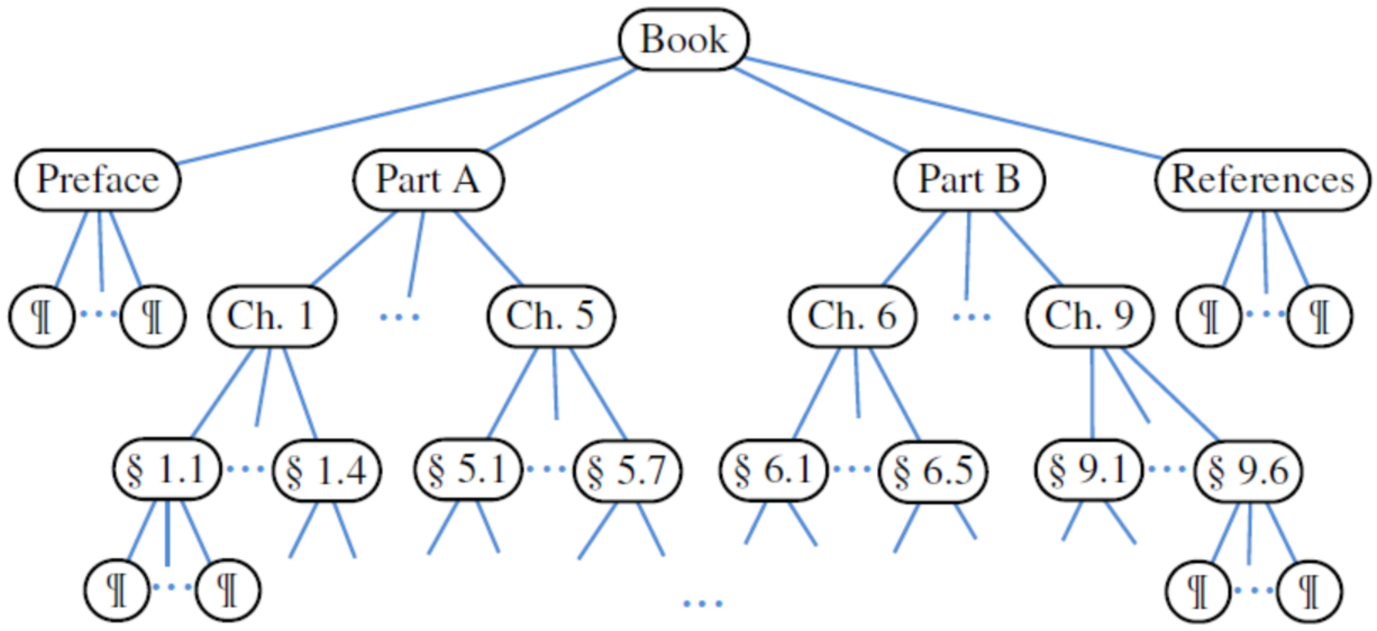
- Terms
  - Sibling: two nodes that are children of the same parent
  - External node (leaves): a node v is external if it has no children
  - Internal node: a node v is internal if it has one or more children

Trees

- Terms
  - Ancestor: u is an ancestor of v if u = v or u is an ancestor of the parent of v
  - Descendant: v is a descendant of u if u is an ancestor of v
  - Subtree of T rooted at v: the tree consisting of all the descendants of v in T (including v itself)
  - Edge: an edge of tree T is a pair of nodes (u, v) such that u is the parent of v, or vice versa
  - Path: a path of T is a sequence of nodes such that any two consecutive sequence form an edge

Ordered Trees

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



Tree Abstract Data Type

- Position of a tree

getElement( ):	Returns the element stored at this position.

- Accessor methods

root( ):	Returns the position of the root of the tree (or null if empty).
parent(p):	Returns the position of the parent of position p (or null if p is the root).
children(p):	Returns an iterable collection containing the children of position p (if any).
numChildren(p):	Returns the number of children of position p.

Tree Abstract Data Type

- Query methods

isInternal(p):	Returns true if position p has at least one child.
isExternal(p):	Returns true if position p does not have any children.
isRoot(p):	Returns true if position p is the root of the tree.

- Other methods

size( ):	Returns the number of positions (and hence elements) that are contained in the tree.
isEmpty( ):	Returns true if the tree does not contain any positions (and thus no elements).
iterator( ):	Returns an iterator for all elements in the tree (so that the tree itself is Iterable).
positions( ):	Returns an iterable collection of all positions of the tree.

A Tree Interface in Java

```java
public interface Tree<E> extends Iterable<E> {
    //accessor methods
    Position<E> root();
    Position<E> parent(Position<E> p) throws IllegalArgumentException;
    Iterable<Position<E>> children(Position<E> p)
                                      throws IllegalArgumentException;

    //query methods
    int numChildren(Position<E> p)    throws IllegalArgumentException;
    boolean isInternal(Position<E> p) throws IllegalArgumentException;
    boolean isExternal(Position<E> p) throws IllegalArgumentException;
    boolean isRoot(Position<E> p)     throws IllegalArgumentException;

    //other methods
    int size();
    boolean isEmpty();
    Iterator<E> iterator();            //iterator for elements
    Iterable<Position<E>> positions(); //iterable collection of positions
}
```

An AbstractTree Base Class

```java
public abstract class AbstractTree<E> implements Tree<E> {
    //Interface Tree
    //using numChildren, root, and size is a template pattern
    public boolean isInternal(Position<E> p) { return numChildren(p) >  0; }
    public boolean isExternal(Position<E> p) { return numChildren(p) == 0; }
    public boolean isRoot(Position<E> p)     { return p == root(); }
    public boolean isEmpty()                 { return size() == 0; }


    //Iterator of elements
    private class ElementIterator implements Iterator<E> {
        Iterator<Position<E>> posIterator = positions().iterator();
        public boolean hasNext() { return posIterator.hasNext(); }
        public E next()          { return posIterator.next().getElement(); }
    }
    public Iterator<E> iterator() {
        return new ElementIterator();
    }
}
```

Depth and Height

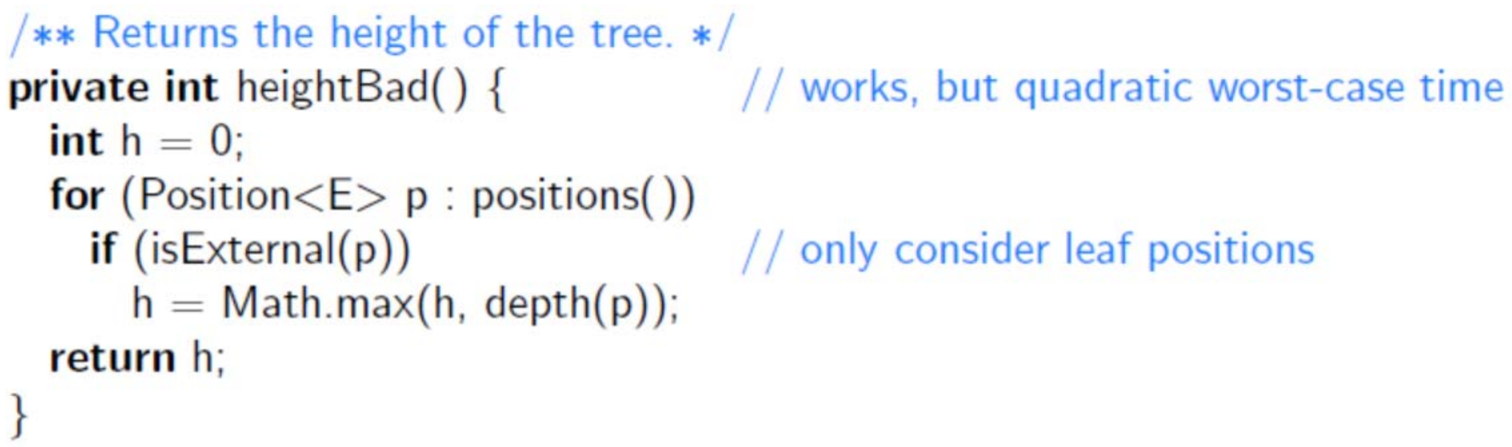
- Depth: the depth of a position p in T is the *number of ancestors of p,* other than p itself
  - If p is the root, the depth of p is 0
  - Otherwise, the depth of p is one plus the depth of its parent

```
/** Returns the number of levels separating Position p from the root. */
public int depth(Position<E> p) {
  if (isRoot(p))
    return 0;
  else
    return 1 + depth(parent(p));
}
```

  - depth is O(n)

Depth and Height

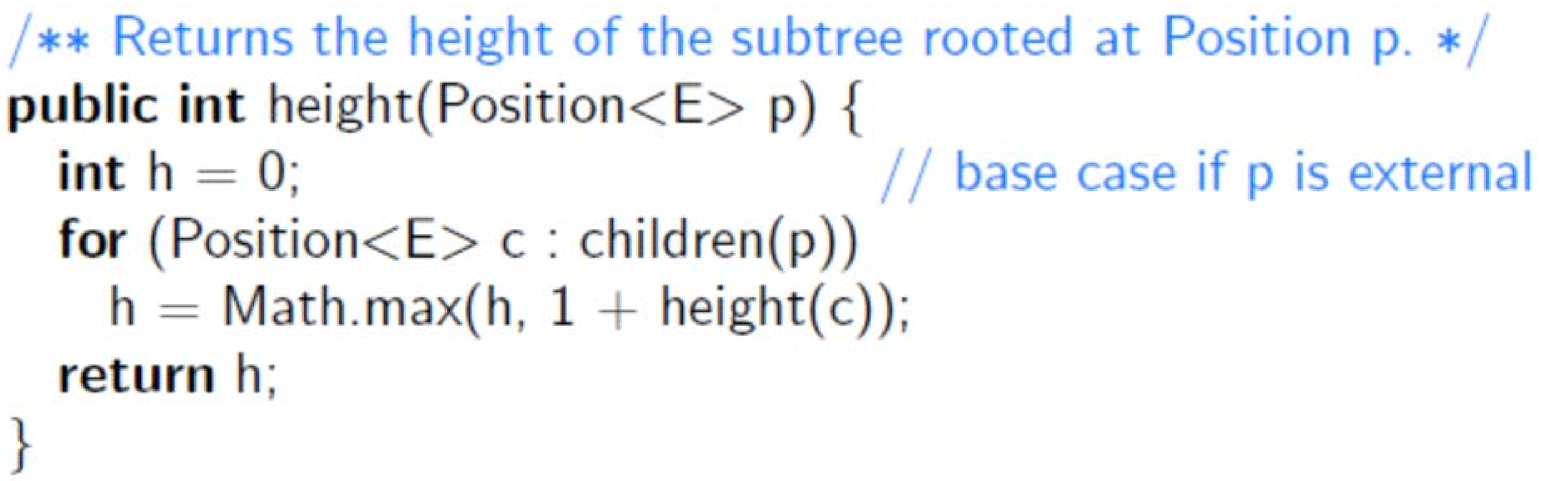
- Height: the height of a tree is the maximum depth of all positions



```
/** Returns the height of the tree. */
private int heightBad() {                  // works, but quadratic worst-case time
  int h = 0;
  for (Position<E> p : positions())
    if (isExternal(p))                     // only consider leaf positions
      h = Math.max(h, depth(p));
  return h;
}
```

  - Big-Oh of heightBad?

Depth and Height



```
/** Returns the height of the subtree rooted at Position p. */
public int height(Position<E> p) {
  int h = 0;                          // base case if p is external
  for (Position<E> c : children(p))
    h = Math.max(h, 1 + height(c));
  return h;
}
```

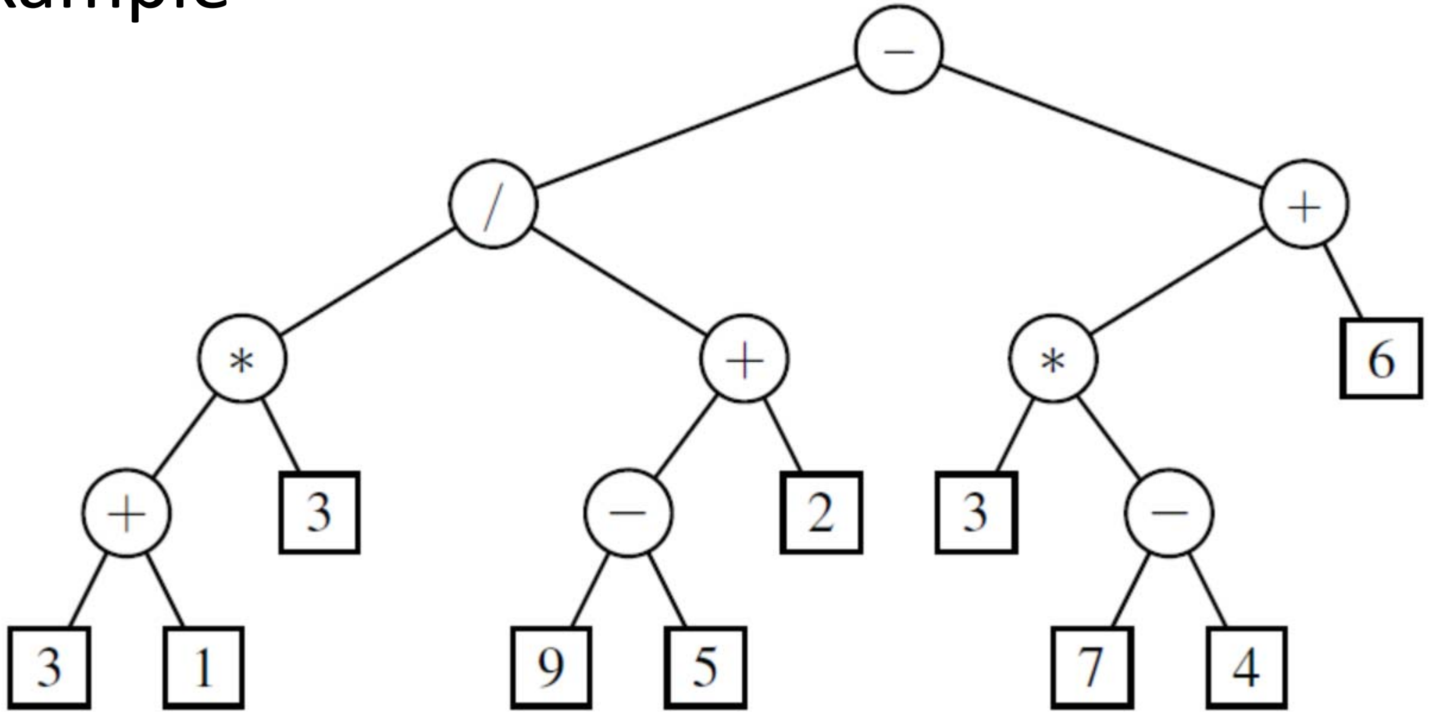
- height is O(n)
  - The total number of child node of another node is n-1

Binary Trees

- A *binary tree* is an *ordered tree* such that
  - Every node has *at most two children*
  - Each node is labeled as a *left child* or a *right child*
  - A *left child precedes a right child* in the order

Binary Trees

- Example



$$((((3+1)*3)/((9-5)+2))-((3*(7-4))+6))$$

Binary Trees

- Terms
  - Left (right) subtree: subtree rooted at the left(right) child
  - A binary tree is proper (full) if each node has either zero or two children
  - A binary tree that is not proper is improper

Binary Tree Abstract Data Type

- Has three additional accessors on top of the Tree ADT

left(p):	Returns the position of the left child of p (or null if p has no left child).
right(p):	Returns the position of the right child of p (or null if p has no right child).
sibling(p):	Returns the position of the sibling of p (or null if p has no sibling).

A BinaryTree Interface

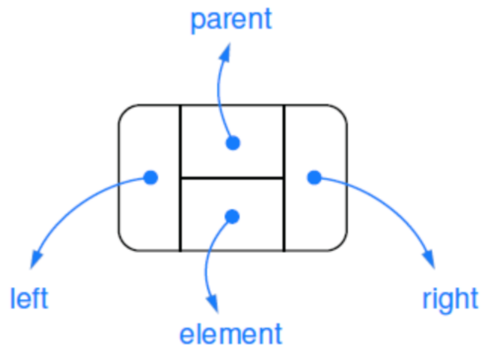
```
public interface BinaryTree<E> extends Tree<E> {
    Position<E> left(Position<E> p)    throws IllegalArgumentException;
    Position<E> right(Position<E> p)   throws IllegalArgumentException;
    Position<E> sibling(Position<E> p) throws IllegalArgumentException;
}
```

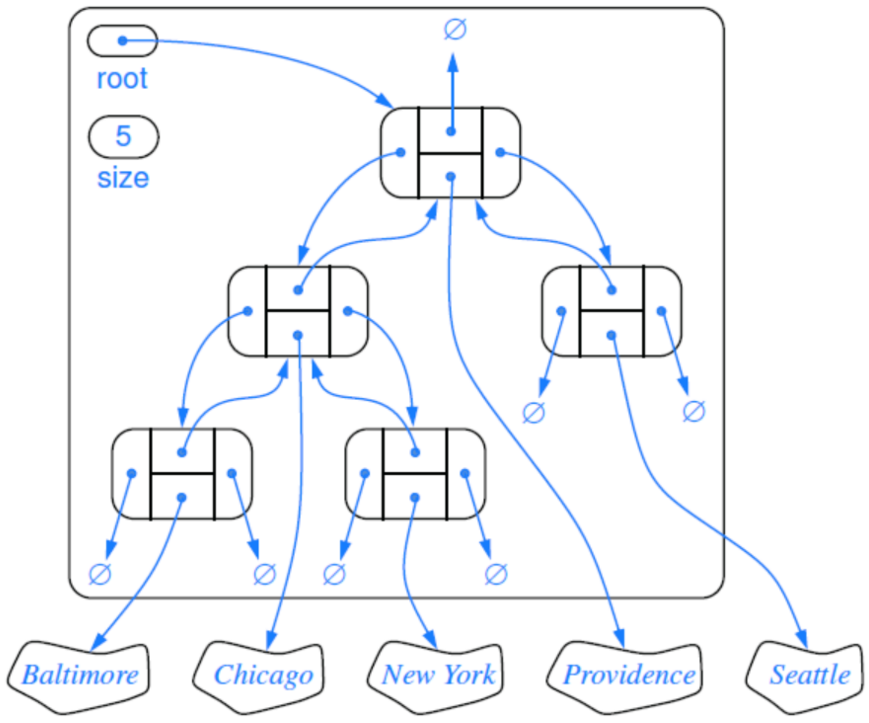
AbstractBinaryTree Base Class

```
public abstract class AbstractBinaryTree<E> extends AbstractTree<E>
                                            implements BinaryTree<E> {
    //Interface BinaryTree
    public Position<E> sibling(Position<E> p) {

        Position<E> parent = parent(p);

        return  parent == null     ? null //root node
            :   p == left(parent) ? right(parent)
            :   left(parent)
            ;

    }
    //Interface Tree
    public int numChildren(Position<E> p) {
        int count = 0;
        if(left(p) != null)
            count++;
        if(right(p) != null)
            count++;
        return count;
    }

    public Iterable<Position<E>> children(Position<E> p) {

        List<Position<E>> snapshot = new ArrayList<Position<E>>(2);

        if(left(p) != null)
            snapshot.add(0, left(p));
        if(right(p) != null)
            snapshot.add(snapshot.size(), right(p));

        return snapshot;
    }
}
```

Tree Representation

- Linked Structure for Binary Trees

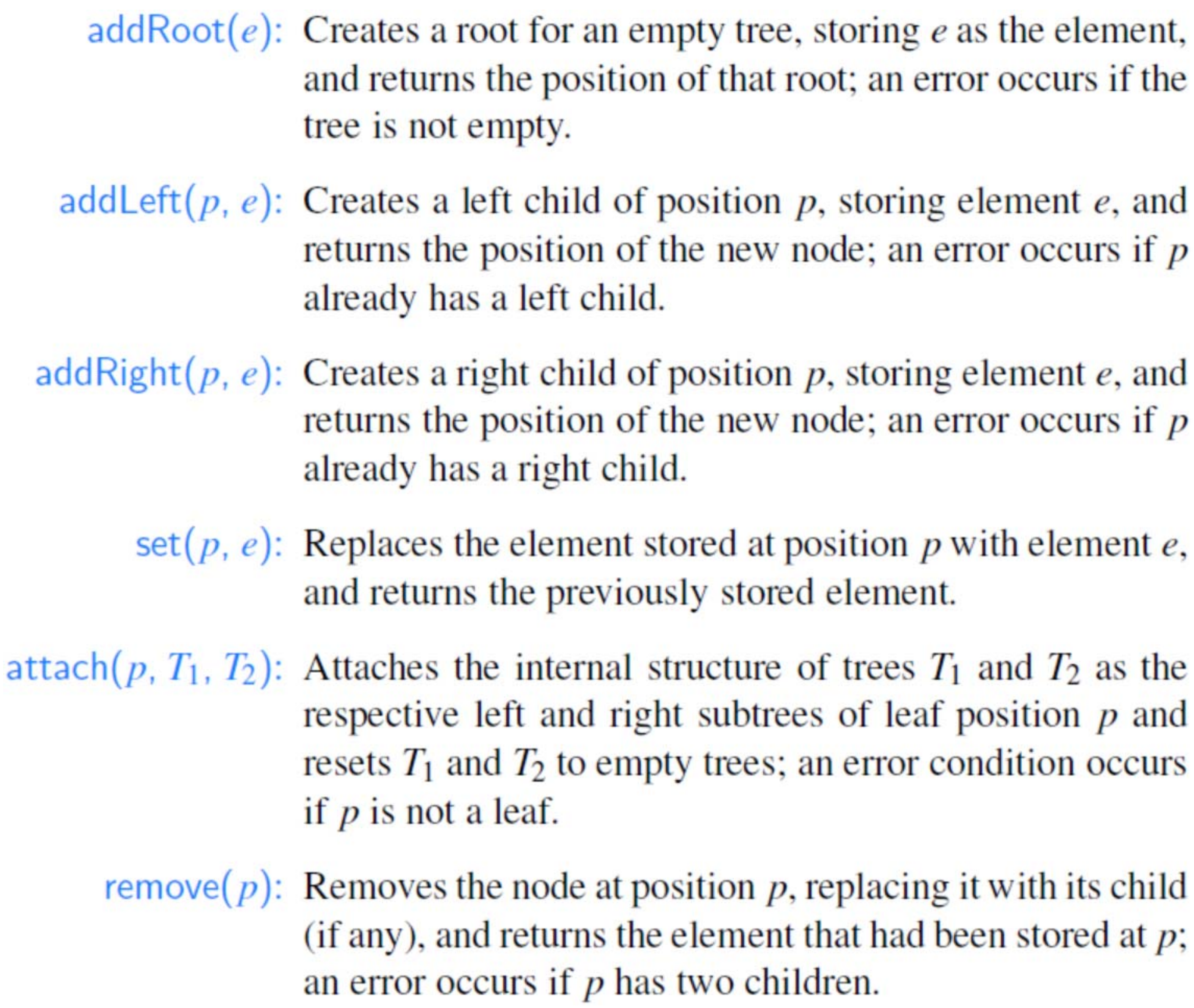




Linked Binary Tree

- Additional update methods
  - Compromising encapsulation for the efficiency
    - Exposing some O(1) methods that can improve the efficiency.
  - Conflicting goals
    - Encapsulation: the outward behaviors of an ADT need not depend on the internal representation
    - Efficiency: depends on the representation greatly

Linked Binary Tree: additional methods



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, T_1, T_2):	Attaches the internal structure of trees T_1 and T_2 as the respective left and right subtrees of leaf position p and resets T_1 and T_2 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 LinkedBinaryTree

```java
public class LinkedBinaryTree<E> extends AbstractBinaryTree<E> {
    protected static class Node<E> implements Position<E> {
        private E e;
        private Node<E> parent, left, right;
        public Node(E e, Node<E> parent, Node<E> left, Node<E> right) {
            this.e = e;
            this.parent = parent; this.left = left; this.right = right;
        }

        //accessor
        public E getElement()       { return e; }
        public Node<E> getParent() { return parent; }
        public Node<E> getLeft()   { return left; }
        public Node<E> getRight()  { return right; }

        //update
        public void setElement(E e)           { this.e = e; }
        public void setParent(Node<E> parent) { this.parent = parent; }
        public void setLeft(Node<E> left)     { this.left= left; }
        public void setRight(Node<E> right)   { this.right = right; }
    }
```

```java
protected Node<E> root;
private int size;

//Constructor
public LinkedBinaryTree() {}

//factory method
protected Node<E> createNode(E e, Node<E> parent,
                                        Node<E> left, Node<E> right) {
    return new Node<E>(e, parent, left, right);
}

protected Node<E> validate(Position<E> p)
                  throws IllegalArgumentException {
    if(!(p instanceof Node))
        throw new IllegalArgumentException("Invalid Position type");

    Node<E> node = (Node<E>) p;
    if(node.getParent() == node)    //our convention to defunct a node
        throw new IllegalArgumentException("p is not in the tree");

    return node;
}
```

```
//Interface Tree
public int size()                        { return size; }
public Position<E> root()                { return root; }
public Position<E> parent(Position<E> p) { return validate(p).getParent(); }

//Interface BinaryTree
public Position<E> left(Position<E> p)   { return validate(p).getLeft();  }
public Position<E> right(Position<E> p)  { return validate(p).getRight(); }

//Class specific methods
public E set(Position<E> p, E e) {
    Node<E> node = validate(p);
    E temp = node.getElement();
    node.setElement(e);
    return temp;
}

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;
}
```

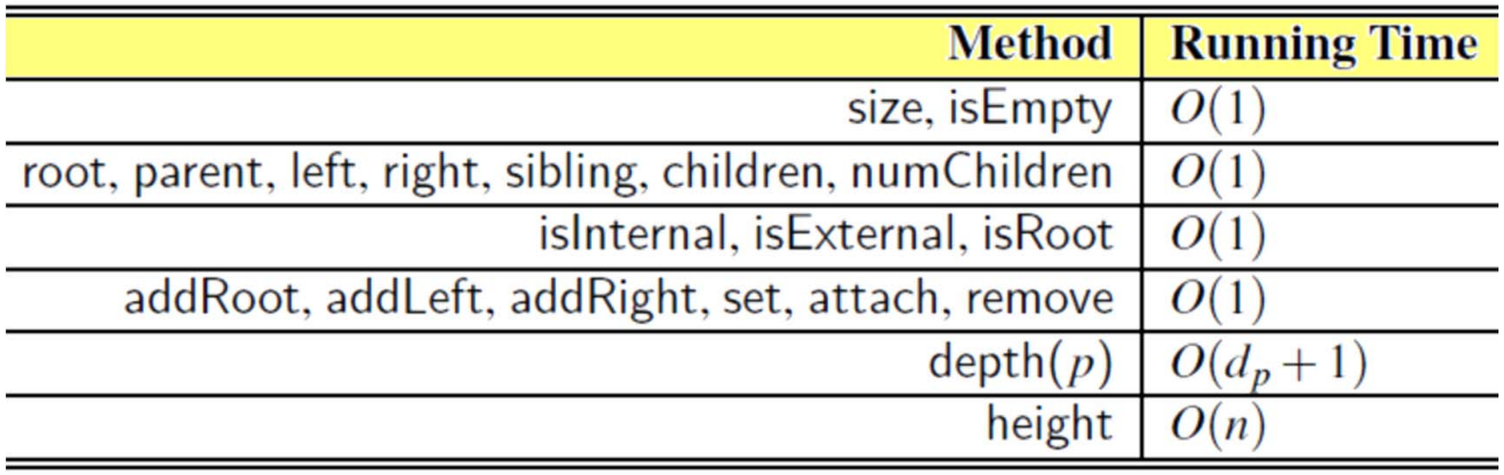
```java
public Position<E> addLeft(Position<E> p, E e)
                            throws IllegalArgumentException {
    Node<E> parent = validate(p);
    if(parent.getLeft() != null)
        throw new IllegalArgumentException("p already has a left child");

    //call the factory method
    Node<E> child = createNode(e, parent, null, null);
    parent.setLeft(child);
    size++;
    return child;
}

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");

    //call the factory method
    Node<E> child = createNode(e, parent, null, null);
    parent.setRight(child);
    size++;
    return child;
}
```

```
public void attach(Position<E> p,
                   LinkedBinaryTree<E> t1, LinkedBinaryTree<E> t2)
                   throws IllegalArgumentException {
    Node<E> node = validate(p);

    if(t1 != null && !t1.isEmpty()) {
        if(node.getLeft() != null)
            throw new IllegalArgumentException("p already has a left child");
        node.setLeft(t1.root);
        size += t1.size();
        t1.root.setParent(node);
        t1.root = null;
        t1.size = 0;
    }

    if(t2 != null && !t2.isEmpty()) {
        if(node.getRight() != null)
            throw new IllegalArgumentException("p already has a right child");
        node.setRight(t2.root);
        size += t2.size();
        t2.root.setParent(node);
        t2.root = null;
        t2.size = 0;
    }
}
```

```java
//remove the node at p and replace it with its child
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(); //find a possible child
    if(child == null)
        child = node.getRight();

    if(child != null)
        child.setParent(node.getParent()); //promote the child

    if(node == root)
        root = child;  //make child the root
    else {             //promote the child
        Node<E> parent = node.getParent();
        if(node == parent.getLeft())
            parent.setLeft(child);
        else
            parent.setRight(child);
    }
…
…
    size--;
    E temp = node.getElement();
    node.setElement(null);
    node.setLeft(null);
    node.setRight(null);
    node.setParent(node); //our convention for defunct node
    return temp;
}
```

LinkedBinaryTree

- Performance



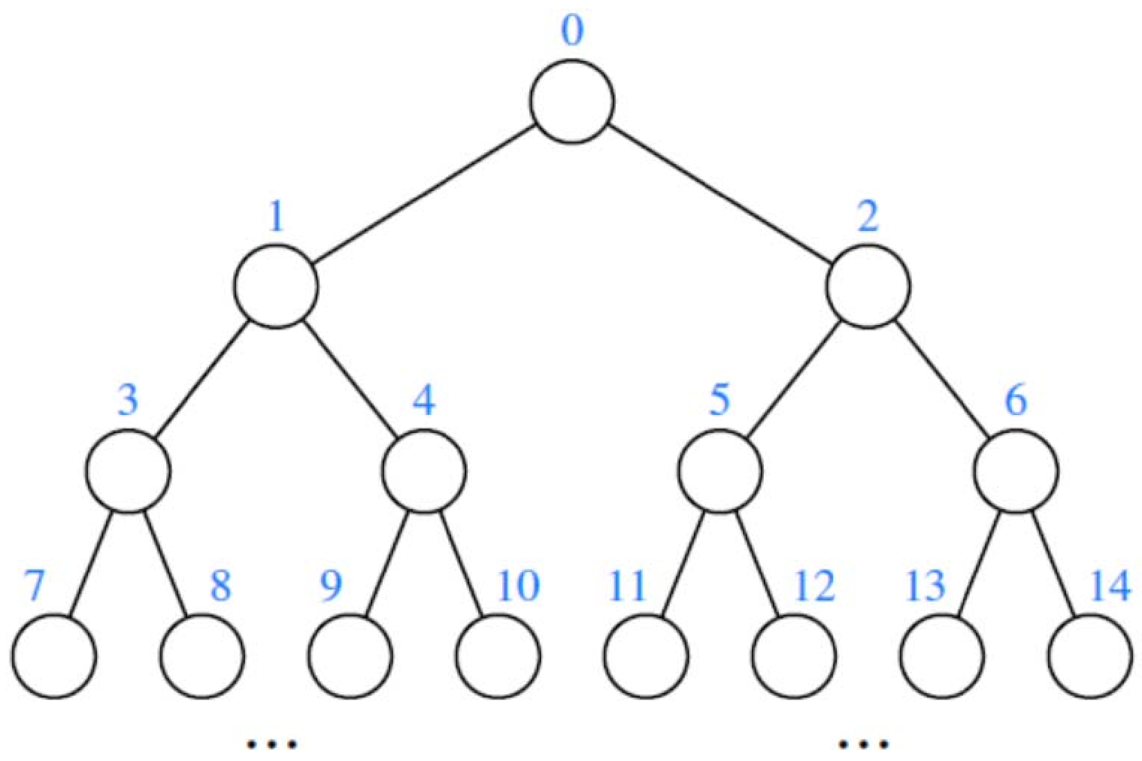
Method	Running Time
size, isEmpty	$O(1)$
root, parent, left, right, sibling, children, numChildren	$O(1)$
isInternal, isExternal, isRoot	$O(1)$
addRoot, addLeft, addRight, set, attach, remove	$O(1)$
depth(p)	$O(d_p + 1)$
height	$O(n)$

An Array-Based Binary Tree

- For every position *p* of *T*, let *index* (*p*) be the integer defined as follows
  - If p is the *root* of T, then
    index (*p*) = 0
  - If p is the *left child of q*, then
    index (*p*) = 2 · *index* (*q*) + 1
  - If p is the *right child of q*, then
    index (*p*) = 2 · *index* (*q*) + 2

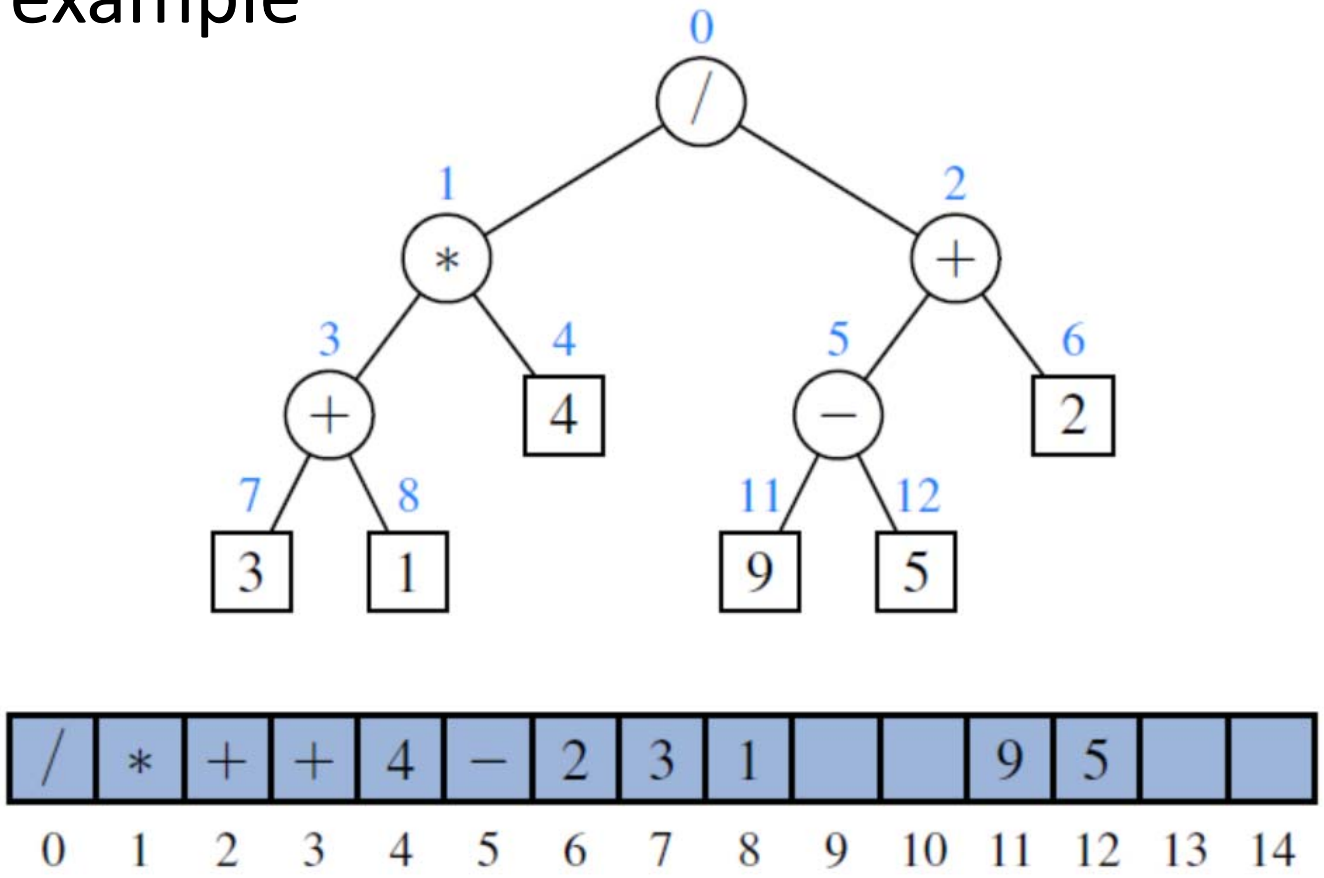
An Array-Based Binary Tree

- A general scheme

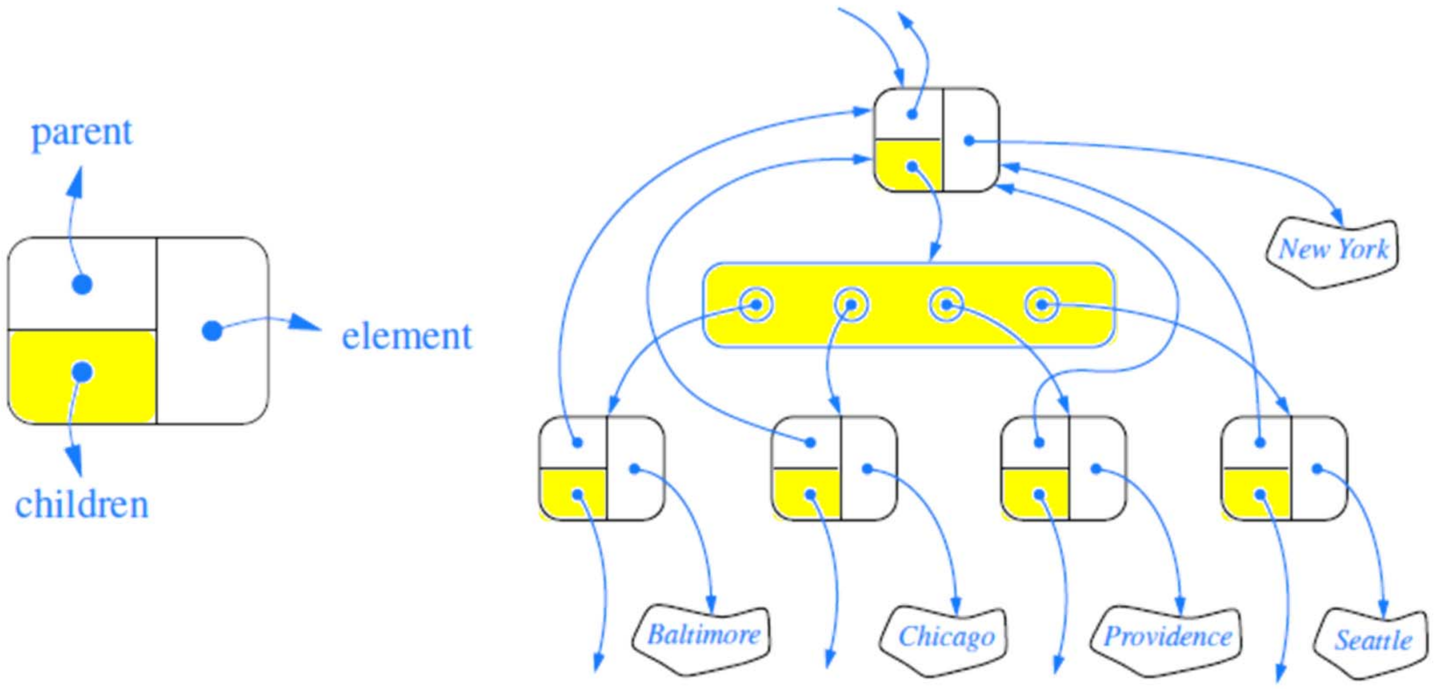


An Array-Based Binary Tree

- An example



Linked Structure for General Trees



Tree Traversal Algorithms

- Traversal of a tree T
  - A systematic way of accessing, or “visiting,” all the possible positions of T
  - Actions associated with the visit
    - E.g. a simple counting, printing, complex operations for p, …

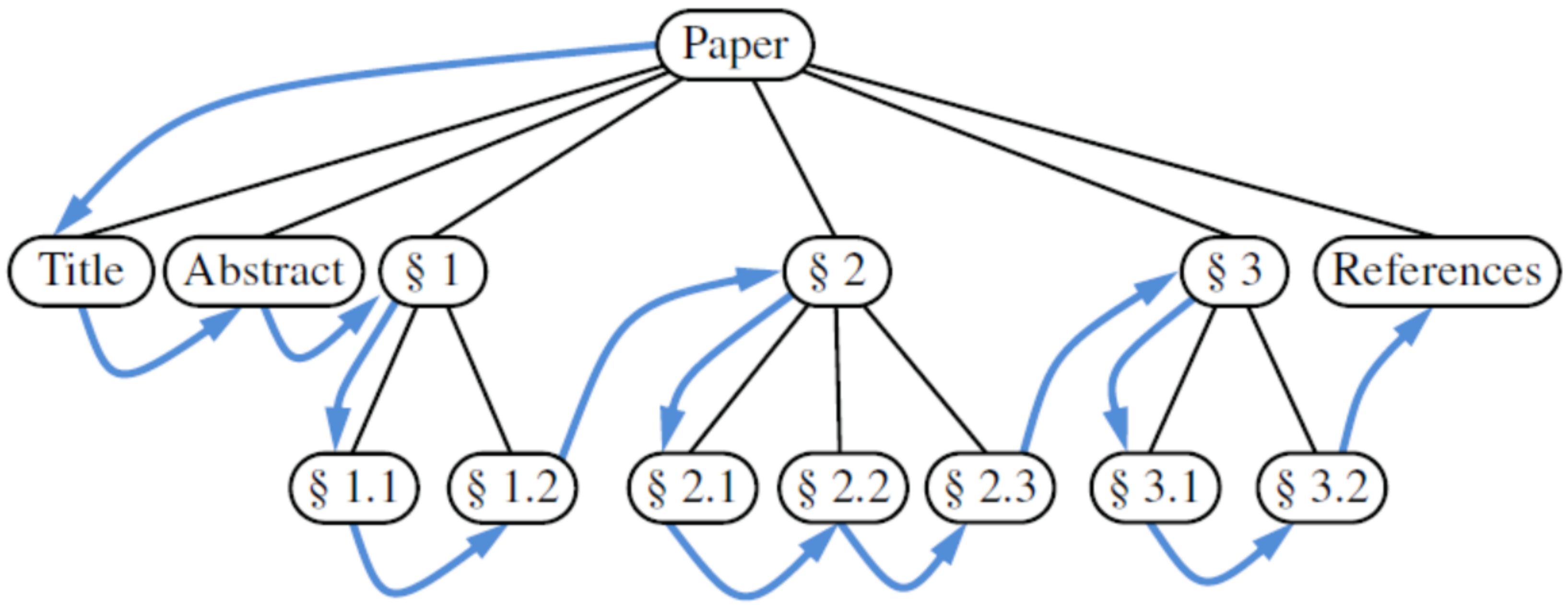
Preorder Traversal

- Preorder traversal of a tree T
  - The *root of T is visited first* and *then the subtrees rooted at its children* are traversed recursively

```
Algorithm preorder(p):
    perform the "visit" action for position p    { this happens before any recursion }
    for each child c in children(p) do
        preorder(c)                                { recursively traverse the subtree rooted at c }
```

Preorder Traversal

- Example



Java Implementation

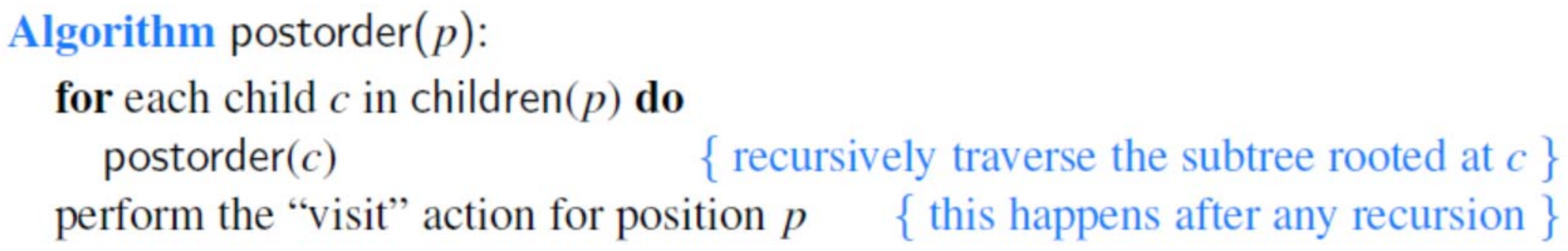
```java
//Tree traversal (add below to Tree.java)
public Iterable<Position<E>> preorder();
public Iterable<Position<E>> postorder();
public Iterable<Position<E>> breadthFirst();


//tree traversal (add below to BinaryTree.java)
public Iterable<Position<E>> inorder();


//Iterable collection of positions (add the code to AbstractTree.java)
public Iterable<Position<E>> positions() {
    return preorder();
}
```

Preorder Traversal

```
//
// preorder traversal
//

// add the positions to snapshot while traversing the tree
private void preorderSubtree(Position<E> p, List<Position<E>> snapshot) {

    snapshot.add(p);

    for(Position<E> c: children(p))
        preorderSubtree(c, snapshot);
}

public Iterable<Position<E>> preorder() {
    List<Position<E>> snapshot = new ArrayList<>();
    if(!isEmpty())
        preorderSubtree(root(), snapshot);
    return snapshot;
}
```

Postorder Traversal

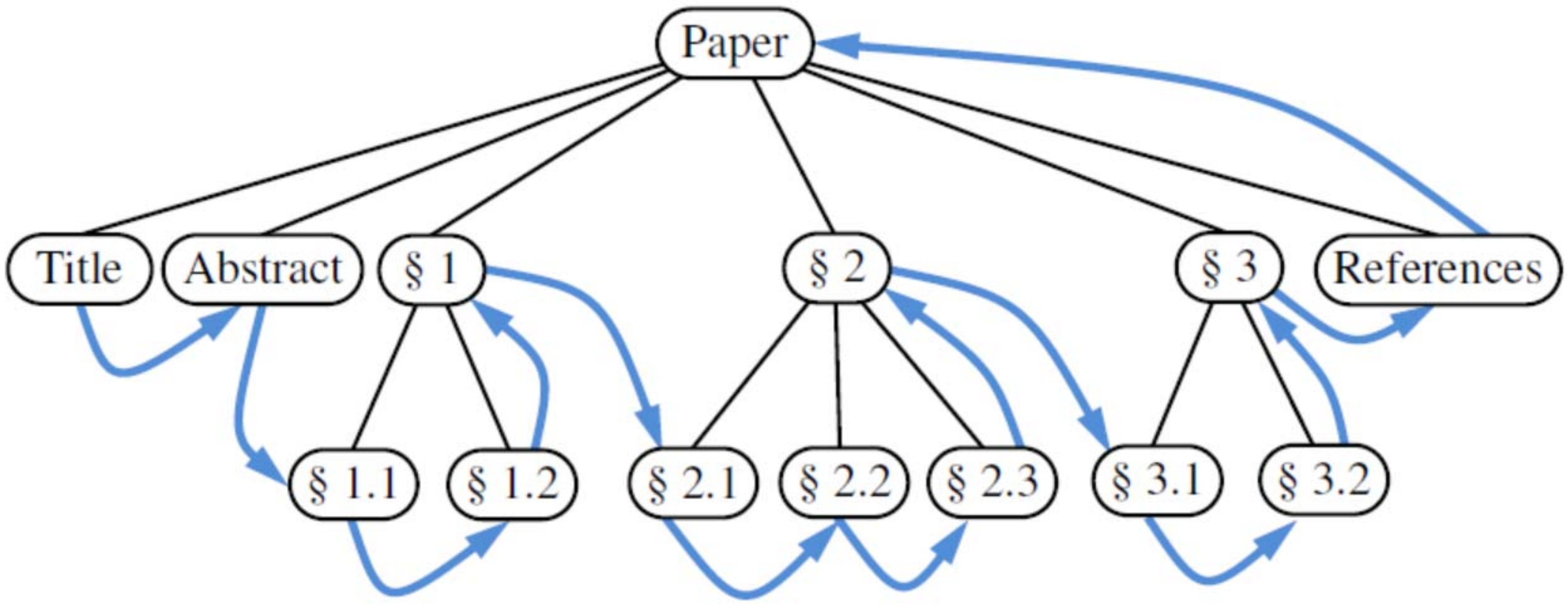
- Postorder traversal of a tree T
  - *Recursively traverses the subtrees* rooted at the *children of the root first*, and *then visits the root*



```
Algorithm postorder($p$):
  for each child $c$ in children($p$) do
    postorder($c$)                                  { recursively traverse the subtree rooted at $c$ }
  perform the "visit" action for position $p$      { this happens after any recursion }
```

Postorder Traversal

- Example

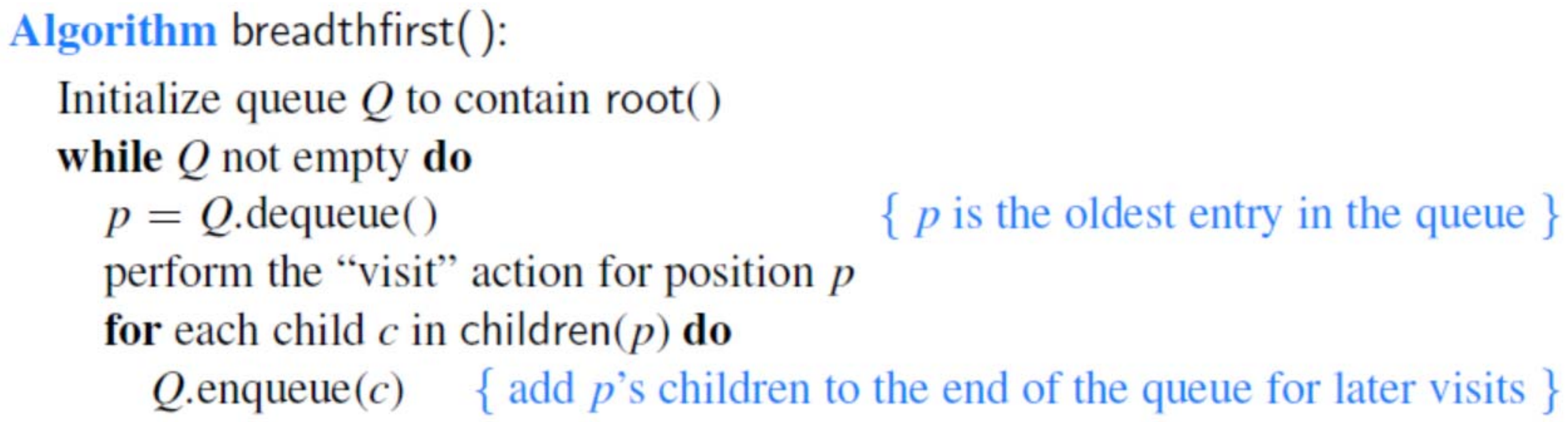


Java Implementation

```
//
// postorder traversal
//

// add the positions to snapshot while traversing the tree
private void postorderSubtree(Position<E> p, List<Position<E>> snapshot) {

    for(Position<E> c: children(p))
        postorderSubtree(c, snapshot);

    snapshot.add(p);
}

public Iterable<Position<E>> postorder() {
    List<Position<E>> snapshot = new ArrayList<>();
    if(!isEmpty())
        postorderSubtree(root(), snapshot);
    return snapshot;
}
```

Breadth-First Tree Traversal

- Breadth-first traversal of a tree T
  - Visit *all the positions at depth d* before visiting the positions at depth d+1

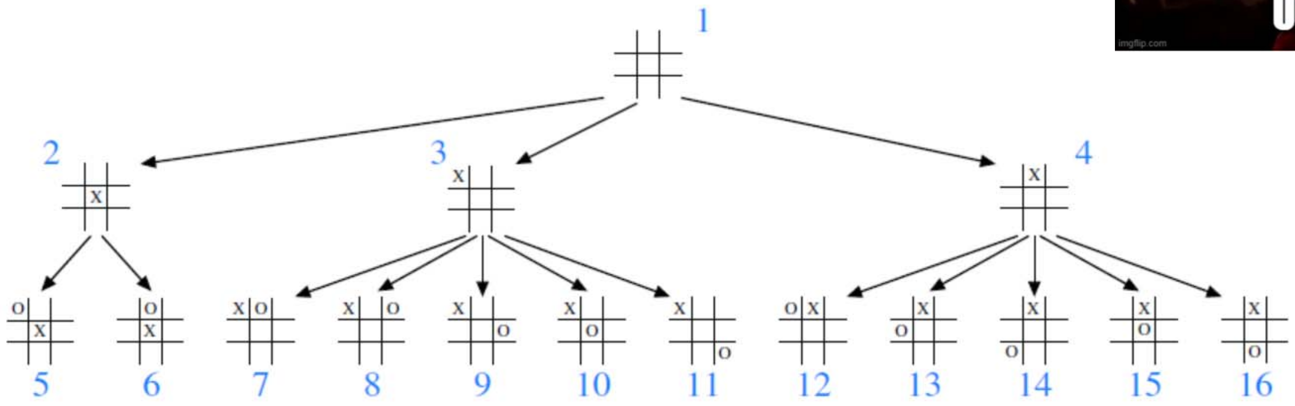


```
Algorithm breadthfirst( ):
  Initialize queue $Q$ to contain root( )
  while $Q$ not empty do
    $p = Q$.dequeue( )                    { $p$ is the oldest entry in the queue }
    perform the "visit" action for position $p$
    for each child $c$ in children($p$) do
      $Q$.enqueue($c$)     { add $p$'s children to the end of the queue for later visits }
```

Breadth-First Tree Traversal

- Example: game tree
  - Possible choices of moves that might be made by a player (or a computer) during a game
  - Root is the initial configuration

I looked forward in time, I saw 14,000,605 futures
Did you run Breadth-First Search?
Umm...
imgflip.com



Java Implementation

```
//breadth first traversal
public Iterable<Position<E>> breadthFirst() {
    List<Position<E>> snapshot = new ArrayList<>();
    if(!isEmpty()) {

        Queue<Position<E>> queue = new LinkedList<>();
        queue.add(root());
        while(!queue.isEmpty()) {
            Position<E> p = queue.remove();
            snapshot.add(p);
            for(Position<E> c: children(p))
                queue.add(c);
        }

    }
    return snapshot;
}
```

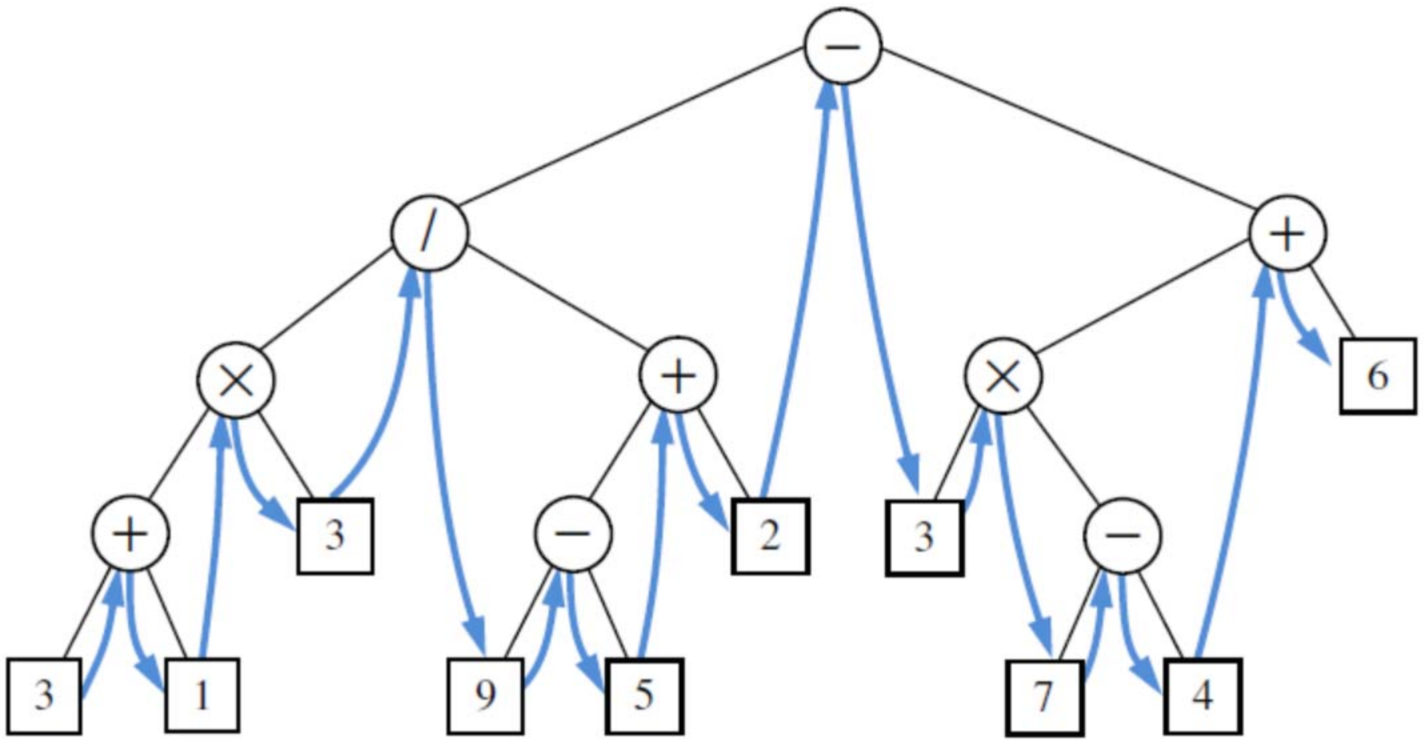
Inorder Traversal of a Binary Tree

- Inorder traversal of a tree T
  - *Visit a position between the recursive traversals* of its left and right subtrees

```
Algorithm inorder($p$):
    if $p$ has a left child $lc$ then
        inorder($lc$)                    { recursively traverse the left subtree of $p$ }
    perform the "visit" action for position $p$
    if $p$ has a right child $rc$ then
        inorder($rc$)                    { recursively traverse the right subtree of $p$ }
```

Inorder Traversal of a Binary Tree

- Example: (3+1)*3 / ((9-5)+2) – (3*(7-4) + 6)



Java Implementation

```
//
// inorder traversal
//
private void inorderSubtree(Position<E> p, List<Position<E>> snapshot) {
    if(left(p) != null)
        inorderSubtree(left(p), snapshot);

    snapshot.add(p);

    if(right(p) != null)
        inorderSubtree(right(p), snapshot);
}

public Iterable<Position<E>> inorder() {
    List<Position<E>> snapshot = new ArrayList<>();
    if(!isEmpty())
        inorderSubtree(root(), snapshot);
    return snapshot;
}
```

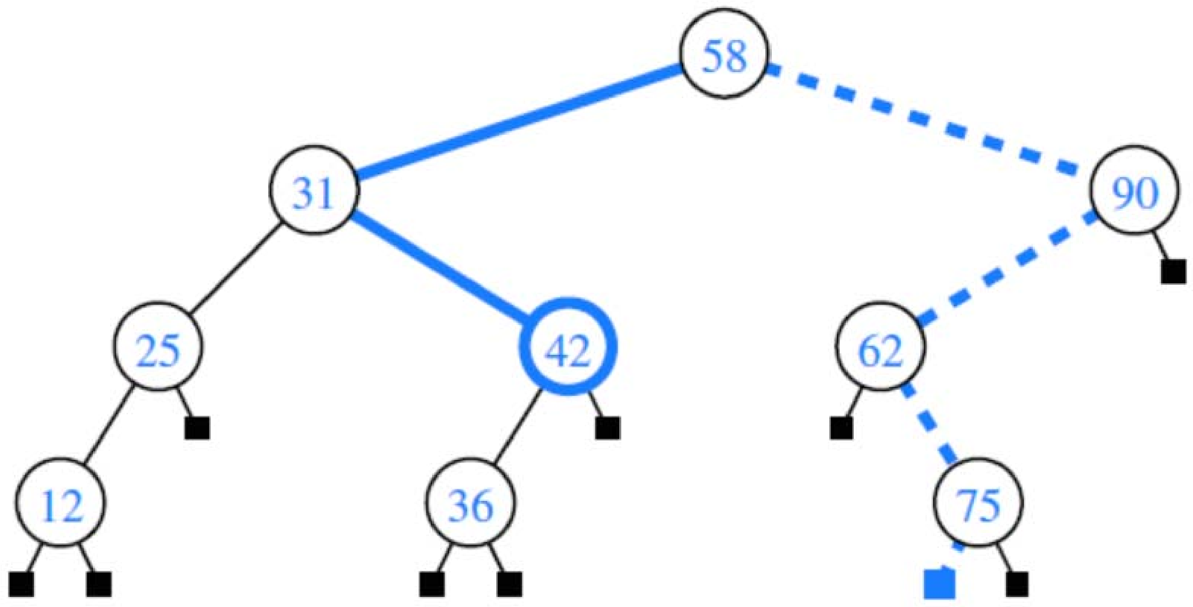
Binary Search Trees

- Let *S* be a set whose unique *elements have an order relation*

- Binary search tree for *S*
  - Position *p* stores an element of S, denoted as *e*(*p*)
  - Elements stored in the *left subtree of p are less than e*(*p*)
  - Elements stored in the *right subtree of p are greater than e*(*p*)

Binary Search Trees

- Example

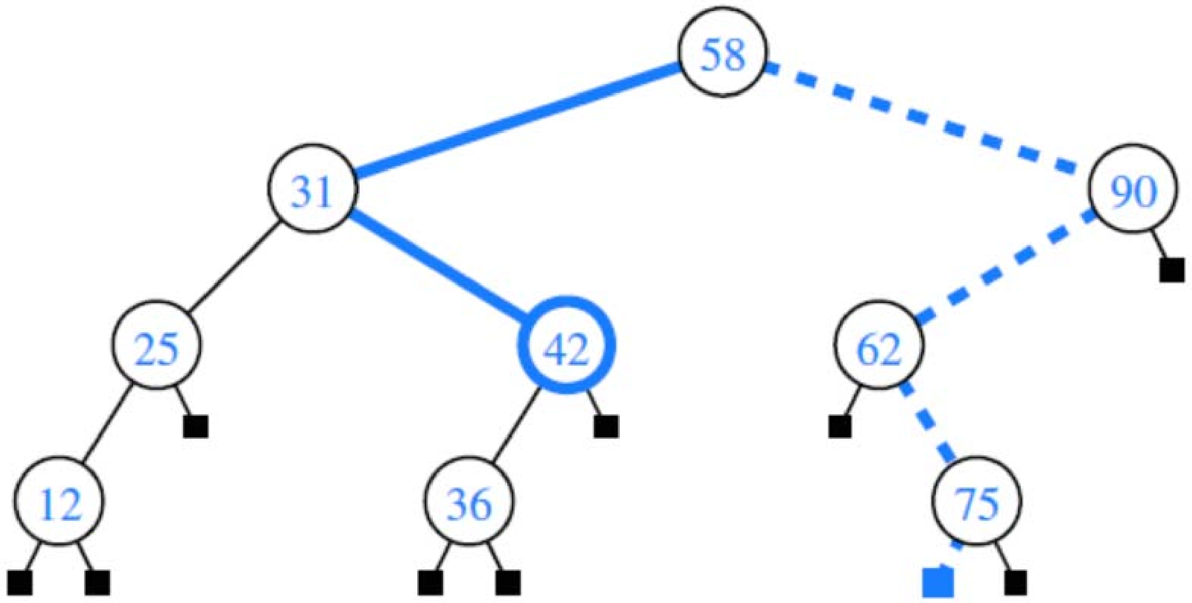


Binary Search Trees

- To check whether *v* is in *S*
  - Search starts from the root node
  - On encountering an *internal node* p
    - If v = e(p), the search terminates successfully
    - If v < e(p), the search continues in the left subtree of p
    - If v > e(p), the search continues in the right subtree of p
  - On reaching a *leaf node*
    - The search terminates unsuccessfully

Binary Search Trees

- Example
  - Searching for 42 (solid line) and 75 (dashed line)



Binary Search Trees

- Performance
  - Running time of the search is proportional to the height of T
  - The height of a binary tree with n nodes can be
    - As small as log(n+1)-1
    - As large as n-1
  - Binary search trees are most efficient when their height is small

```
public class LinkedBinarySearchTree<E extends Comparable<E>>
                                   extends LinkedBinaryTree<E> {
    //Constructor
    public LinkedBinarySearchTree() {}

    private Position<E> find(Position<E> p, E e) {
        Node<E> node = validate(p);
        int cmp = e.compareTo(node.getElement());
        if(cmp == 0)
            return node;
        else if(cmp < 0)
            return left(node)  == null ? null : find(left(node), e);
        else
            return right(node) == null ? null : find(right(node), e);
    }

    public Position<E> find(E e) {
        return root() == null ? null: find(root(), e);
    }
```

```
private Position<E> add(Position<E> p, E e) {
    Node<E> node = validate(p);
    int cmp = e.compareTo(node.getElement());
    if(cmp <= 0)
        return left(node)  != null ? add(left(node), e)
                                   : addLeft(node, e);
    else
        return right(node) != null ? add(right(node), e)
                                   : addRight(node, e);
}

public Position<E> add(E e) {
    return root() == null ? addRoot(e) : add(root(), e);
}
```

```
public static void main(String[] args) {
    LinkedBinarySearchTree<Integer> bst =
                                new LinkedBinarySearchTree<>();
    int[] arr = {4, 5, 1, 7, 6, 3, 2, 8};
    for(int i : arr)
        bst.add(i);

    System.out.println("find");
    for(int i : arr)
        System.out.print(bst.find(i).getElement() + ", ");
    System.out.println("");

    System.out.println("postorder");
    for(Position<Integer> i : bst.postorder())
        System.out.print(i.getElement() + ", ");
    System.out.println("");

    System.out.println("preorder");
    for(Position<Integer> i : bst.preorder())
        System.out.print(i.getElement() + ", ");
    System.out.println("");
…
```

```
        System.out.println("breadth first");
        for(Position<Integer> i : bst.breadthFirst())
            System.out.print(i.getElement() + ", ");
        System.out.println("");

        System.out.println("inorder");
        for(Position<Integer> i : bst.inorder())
            System.out.print(i.getElement() + ", ");
        System.out.println("");

        /* expected result
            find
            4, 5, 1, 7, 6, 3, 2, 8,
            postorder
            2, 3, 1, 6, 8, 7, 5, 4,
            preorder
            4, 1, 3, 2, 5, 7, 6, 8,
            breadth first
            4, 1, 5, 3, 7, 2, 6, 8,
            inorder
            1, 2, 3, 4, 5, 6, 7, 8,
        */
    }
}
```

Assignment 7

- In this assignment, we will
  - Build a parse tree from a given expression
  - Evaluate the expression using the parse tree

- Download hw7.zip
  - Implement the TODO lines in Parser.java

- Due date: 5/10/2022

Assignment 7

```
public static BinaryTree<Node> parseExpr(String expr) {
    Scanner scan = new Scanner(expr + "$");
    Stack<LinkedBinaryTree<Node>> stack_tree = new StackByArray<>();
    Stack<Character> stack_oper = new StackByArray<>();

    //TODO: - parseExpr will be similar to evalExpr function that evaluates
    //        infix expressions.
    //      - Here, instead of using the operand stack, we push/pop subtrees
    //        of the parse tree to/from the tree stack.
    //      - When popping an operator, pop one or two parse-trees from the
    //        tree stack; build a parse-tree rooted at the operator; and push
    //        the resulting tree onto the tree stack

}
```

Assignment 7

```
public static double evalExpr(BinaryTree<Node> tree) {
    Stack<Double> num = new StackByArray<Double>();

    //TODO: - evalExpr will be similar to evalPostfixExpr function that
    //        evaluates postfix expressions.
    //      - While traversing the nodes of the parseTree in the post-order,
    //        evaluate the expression by pushing/popping operands to/from
    //        the stack num

}
```

Assignment 7

- Expected result:

```
Enter an expression: ~1 * ~ 2 * ~ 3 / (~4 - ~5) + 6
eval:      0.0
preorder:  + / * * ~ 1 ~ 2 ~ 3 - ~ 4 ~ 5 6
postorder: 1 ~ 2 ~ * 3 ~ * 4 ~ 5 ~ - / 6 +
inorder:   ~ 1 * ~ 2 * ~ 3 / ~ 4 - ~ 5 + 6
```