Binary Tree ADT

Why (proper) binary trees?

- binary trees are a very common type of tree
- proper simplifies the implementation and is not limiting
  - in a proper binary tree, every non-leaf node has exactly two children
  - can have *dummy leaves* (no element is stored there)
- BinaryTree ADT / implementation ideas can easily be extended to general trees
- can implement general trees in terms of binary trees

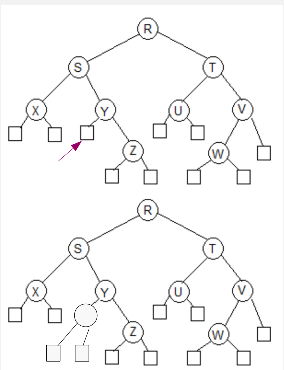
BinaryTree ADT

Note: this is representative of the concept – particular operations, names, parameters may vary.

- standard operations of any class
  - constructor – create a one-node tree
- standard operations for containers
  - size(), isEmpty()
- structural accessors
  - getRoot(), getParent(node), getLeftChild(node), getRightChild(node), sibling(node)
  - isRoot(node), isLeaf(node), isInternal(node)
- manipulating elements
  - setElement(node,elt), swapElements(node1,node2)
- structural mutators
  - expandLeaf(node), removeAboveLeaf(node)

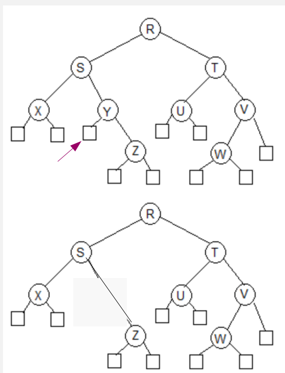
BinaryTree ADT

- expandLeaf(node)



follow with setElement(node,elt) to store an element in the new internal node

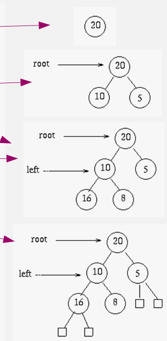
- removeAboveLeaf(node)



removes the leaf node and its parent

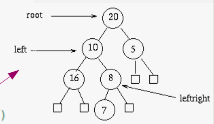
Working With Trees

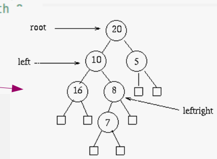
```
// create a tree with 20 at the root
BinaryTree<Integer> tree = new BinaryTree<Integer>(20);

// add 10 and 5 as the children of 20
Node<Integer> root = tree.getRoot(); // the node with 20
tree.expandLeaf(root,10,5);

Node<Integer> left = tree.getLeftChild(root); // the node with 10

// add 16 and 8 as the children of 10
tree.expandLeaf(left,16,8);

// add dummy nodes (no elements) as the children of 5 and 16
tree.expandLeaf(tree.getRightChild(root));
tree.expandLeaf(tree.getLeftChild(left));

// add 7 as the left child of 8 (and a dummy node as the right child)
Node<Integer> leftright = tree.getRightChild(left); // the node with 8
tree.expandLeaf(leftright);
tree.setElement(tree.getLeftChild(leftright),7);

// add dummy nodes (no elements) as the children of 7
tree.expandLeaf(tree.getLeftChild(leftright));
```



Working With Trees

```
// create a tree with 20 at the root
BinaryTree<Integer> tree = new BinaryTree<Integer>(20);

// add 10 and 5 as the children of 20
Node<Integer> root = tree.getRoot(); // the node with 20
tree.expandLeaf(root,10,5);

Node<Integer> left = tree.getLeftChild(root); // the node with 10

// add 16 and 8 as the children of 10
tree.expandLeaf(left,16,8);

// add dummy nodes (no elements) as the children of 5 and 16
tree.expandLeaf(tree.getRightChild(root));
tree.expandLeaf(tree.getLeftChild(left));

// add 7 as the left child of 8 (and a dummy node as the right child)
Node<Integer> leftright = tree.getRightChild(left); // the node with ^
tree.expandLeaf(leftright);
tree.setElement(tree.getLeftChild(leftright),7);

// add dummy nodes (no elements) as the children of 7
tree.expandLeaf(tree.getLeftChild(leftright));
```





Working With Trees – Patterns

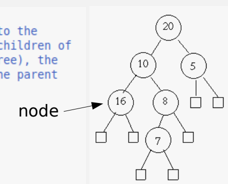
Three main ways of moving through trees:

- moving up the tree
  - loop with current node being updated to parent until the root is reached
- moving down the tree, interested in only one child
  - loop with current node being updated to child until leaf is reached
- moving down the tree, interested in both children
  - recursion (left child and right child), with leaf as base case

(note – these are general patterns; modify specifics like starting or ending point as needed for a particular task)

Working With Trees – Patterns

```
/**
 * Compute the depth of the specified node. The depth corresponds to the
 * number of ancestors - the root has depth 0 (no ancestors), the children of
 * the root have depth 1 (each has one ancestor, the root of the tree), the
 * grandchildren of the root have depth 2 (each has 2 ancestors, the parent
 * and the parent's parent), and so forth.
 *
 * @param node
 *          the node
 * @param tree
 *          the tree
 * @return the depth of the node
 */
public static int getDepth ( Node<Integer> node, BinaryTree<Integer> tree ) {
```

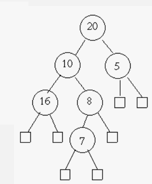


- moving up the tree
  - loop with current node being updated to parent until the root is reached

```
  int depth = 0;

  // pattern: moving up the tree
  for ( Node<Integer> current = node ; !tree.isRoot(current) ; current =
      tree.getParent(current) ) {
    depth++;
  }

  return depth;
}
```


Working With Trees – Patterns

```
/**
 * Return the leftmost internal node in the tree.
 *
 * @param tree
 *          the tree (size > 1)
 * @return the leftmost internal node
 */
public static Node<Integer> findLeftmost ( BinaryTree<Integer> tree ) {
```

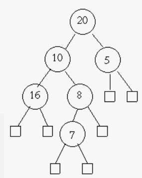


- moving down the tree, interested in only one child
  - loop with current node being updated to child until leaf is reached

```
  if ( tree.getSize() <= 1 ) {
    throw new IllegalArgumentException("tree must have more than one node; size "
        + tree.getSize());
  }

  // pattern: moving down the tree, interested in only one child
  Node<Integer> current = tree.getRoot();
  for ( ; !tree.isLeaf(tree.getLeftChild(current)) ; ) {
    current = tree.getLeftChild(current);
  }

  return current;
}
```

Working With Trees – Patterns

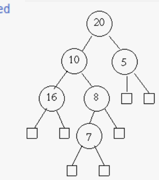
```java
/**
 * Compute the number of internal nodes in the tree.
 *
 * @param tree
 *          the tree
 * @return the number of internal nodes in the tree
 */
public static int getNumInternal ( BinaryTree<Integer> tree ) {
```



- moving down the tree, interested in both children
  - recursion (left child and right child), with leaf as base case

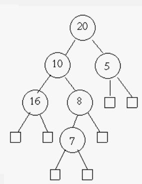
Working With Trees – Patterns

```java
/**
 * Compute the number of internal nodes in the subtree rooted at the specified
 * node.
 *
 * @param node
 *          the node
 * @param tree
 *          the tree
 * @return the number of internal nodes in the subtree rooted at node
 */

private static int getNumInternal ( Node<Integer> node,
                                    BinaryTree<Integer> tree ) {
  // pattern: moving down the tree, interested in both children
  if ( tree.isLeaf(node) ) {
    // base case - where the answer can be computed outright
    // (no internal nodes if there's only a leaf)
    return 0;

  } else {
    // recursive case - where the answer is computed for the left and right
    // subtrees and those answers are used to compute the whole answer
    int leftcount = getNumInternal(tree.getLeftChild(node),tree);
    int rightcount = getNumInternal(tree.getRightChild(node),tree);

    // number of internal nodes = number in left subtree + number in right
    // subtree + 1 for the current node, which is an internal node because
    // it's not a leaf
    return leftcount + rightcount + 1;
  }
}
```

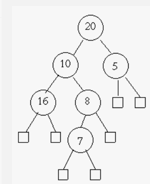


Working With Trees – Patterns

```java
/**
 * Compute the number of internal nodes in the tree.
 *
 * @param tree
 *          the tree
 * @return the number of internal nodes in the tree
 */
public static int getNumInternal ( BinaryTree<Integer> tree ) {
  // pattern: moving down the tree, interested in both children
  return getNumInternal(tree.getRoot(),tree);
}
```

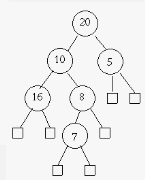



```java
/**
 * Print all of the elements contained in internal nodes in the tree.
 *
 * @param tree
 *          the tree
 */
public static void printTree ( BinaryTree<Integer> tree ) {
  // pattern: moving down the tree, interested in both children
  printTree(tree.getRoot(),tree);
}

/**
 * Print all of the elements contained in internal nodes in the subtree rooted
 * at the specified node.
 *
 * @param node
 *          the node
 * @param tree
 *          the tree
 */
private static void printTree ( Node<Integer> node,
                                BinaryTree<Integer> tree ) {
  // pattern: moving down the tree, interested in both children
  if ( tree.isLeaf(node) ) {
    // base case - where the answer can be computed outright
    // (nothing to print for a leaf since we are only printing internal nodes)

  } else {
    // recursive case - where the answer is computed for the left and right
    // subtrees and then those answers are used to compute the whole answer

    // preorder traversal - current node is handled (its element printed)
    // before the child subtrees
    System.out.println(node.getElement());
    printTree(tree.getLeftChild(node),tree);
    printTree(tree.getRightChild(node),tree);
  }
}
```



Working With Trees – Patterns

```
/**
 * Get the height of the tree. The height of a leaf is 0, the height of a
 * leaf's parents is 1, the height of a leaf's grandparents is 2, etc.
 *
 * @param tree
 *          the tree
 * @return the height of the tree
 */
public static int getHeight ( BinaryTree<Integer> tree ) {
```



- moving down the tree, interested in both children
  - recursion (left child and right child), with leaf as base case

Working With Trees – Patterns

Three main ways of traversing trees:

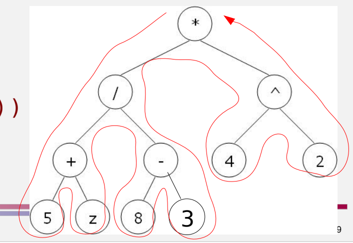
- preorder – visit node before children * / + 5 z – 8 3 ^ 4 2
- inorder – visit node between children 5 + z / 8 – 3 * 4 ^ 2
- postorder – visit node after children 5 z + 8 3 - / 4 2 ^ *

All three traversals are special cases of an Euler tour.

- visit, left, visit, right, visit

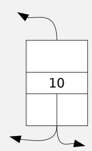
(((5 + z) / (8 – 3)) * (4 ^ 2))

print (on first visit,) on third for internal nodes



Implementing BinaryTree – TreeNode

operation	linked structure
instance variables	• element, parent, left child, right child
getElement()	O(1) – return element



Implementing BinaryTree

(parent pointers not shown)

operation	linked structure
instance variables	• root, size
size()	Th(1) – return size
isEmpty()	Th(1) – return size == 0
getParent(node) getLeftChild(node) getRightChild(node)	Th(1) – return value of instance variable in the node
expandLeaf(node)	Th(1) – create two new nodes, update links, size += 2
removeAboveLeaf(node)	Th(1) – relink grandparent to sibling, size -= 2
setElement(node,elt)	Th(1) – change instance var in node
swapElements(node1,node2)	Th(1) – essentially 2 setElements