CS 228: Introduction to Data Structures
Lecture 28

Binary Trees

A ***binary tree*** is a tree in which every node has degree at most two, and every child is either a ***left child*** or a ***right child***, even if it is the only child its parent has. The following equivalent recursive definition is useful for devising algorithms for binary trees.

Definition. A ***binary tree*** is a structure T defined on a finite set of nodes, such that either

- T is empty (i.e., contains no nodes), or
- T is composed of three disjoint sets of nodes:
  - a ***root*** node,
  - a binary tree called the ***left subtree*** of T, and
  - a binary tree called the ***right subtree*** of T.

Note that in the preceding definition, one or both of the left and right subtrees of T can be empty. In particular, T can consist of a single node (the root).

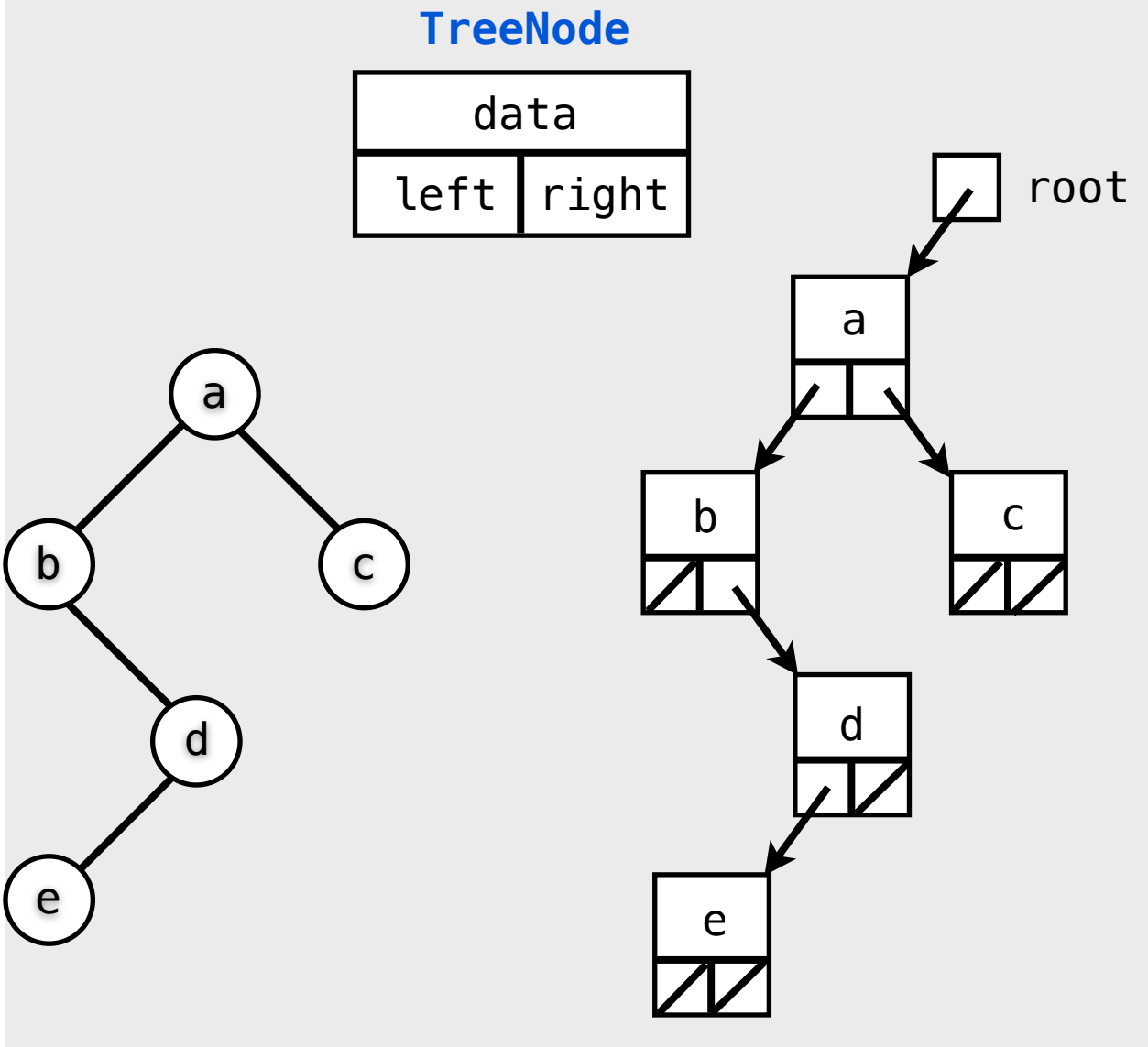
Representing Binary Trees

In the `TreeNode` class posted on Blackboard, each node has two successors, representing the left and right child. This particular implementation does not use child-to-parent links, but there are implementations that do.

```
public class TreeNode<E>
{
  protected TreeNode<E> left;
  protected TreeNode<E> right;
  protected E data;
```

The class includes various constructors; an `isLeaf()` method (whose purpose is obvious); getter methods `left()`, `right()`, and `data()`; and setter methods `setLeft()`, `setRight()`, and `setData()`.

Example. Here is a tree represented using `TreeNode`.



Tree Traversals

A ***traversal*** is a manner of ***visiting*** each node in a tree once. What you do when visiting any particular node depends on the application; for instance, you might print the data stored in the node, or perform some calculation

upon it. We will study four traversal algorithms, which differ from each other in the order they visit the nodes. In what follows, T denotes a binary tree.

A ***preorder traversal*** of T first visits the root of T and then traverses the left and right subtrees of T.

```
PREORDER(T):

    if T is empty
        return

    let T_left and T_right be the left and right subtrees of T

    visit the root of T
    PREORDER(T_left)
    PREORDER(T_right)
```

Here's an implementation using `TreeNode`. This method just prints out the content of each node when it visits the node.

```
 public static void
 traversePreorder(TreeNode<?> node)
  {
    if (node == null) return;
    System.out.print(node.data().toString()
+ " ");
    traversePreorder(node.left());
    traversePreorder(node.right());
  }
```

A ***postorder traversal*** of T first traverses the left subtree of T, then the right subtree of T, and, finally, visits the root.

POSTORDER(T):

if T is empty
return

let T_{left} and T_{right} be the left and right subtrees of T

POSTORDER(T_{left})
POSTORDER(T_{right})
visit the root of T

An ***inorder traversal*** of T first traverses the left subtree of T, then visits the root, and, finally, traverses the right subtree of T.

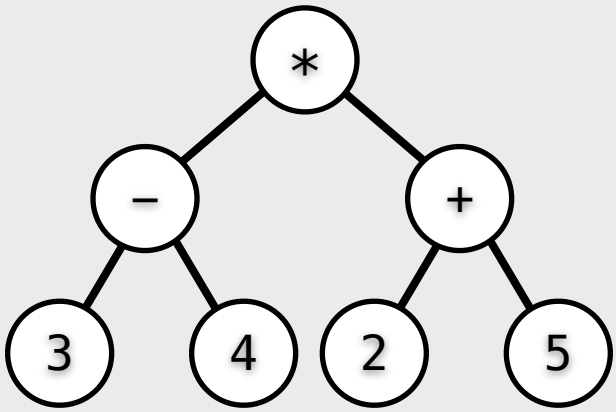
INORDER(T):

if T is empty
return

let T_{left} and T_{right} be the left and right subtrees of T

INORDER(T_{left})
visit the root of T
INORDER(T_{right})

Example. Consider the following expression tree.




```
Preorder:     * - 3 4 + 2 5
Inorder:      3 - 4 * 2 + 5
Postorder:    3 4 - 2 5 + *
```

Note that the preorder, inorder, and postorder traversals of an expression tree print a prefix, infix (without parentheses), and postfix expression, respectively.

A ***level order traversal*** first visits the root, then all the depth-1 nodes (from left to right), then all the depth-2 nodes, etc. It continues visiting nodes level by level, left-to-right within each level, until all nodes are visited.

Example. The level-order sequence for the expression tree of the preceding example is

```
* - + 3 4 2 5
```

Unlike the preorder, inorder, and postorder sequences,the level order sequence does not have a natural interpretation.

The easiest way to implement level order traversal is non-recursively, using a queue, as follows.

```
LevelOrder(T):
    Create an empty queue q
    if T is not empty
        q.enqueue(root)
    while !q.isEmpty()
        x = q.dequeue()
        let y_left be the left child of x
        let y_right be the right child of x
        if (y_left != null) q.enqueue(y_left)
        if (y_right != null) q.enqueue(y_right)
        visit x
```

We will not spend much time on the preceding algorithm now. You should study the above algorithm on your own, and try it out on a few examples, and convince yourselves that it is correct. We will return to this idea when we study non-binary trees and graphs, where level-order traversal — called ***breadth-first search*** in that context — is much more useful.

Application: Computing the Height of a Binary Tree

Tree traversals have several applications. Here we describe how to use post-order traversal to compute the ***height*** of a binary tree.

Recall that the height of a tree T is the length of the longest path from the root of T to a leaf. To devise an algorithm, it is easier to work with the following equivalent recursive definition.

1. If T is empty, then the height of T is –1.

2. Otherwise, the height of T is 1 + max {height left subtree, height right subtree}.

This recursive definition leads directly to the following implementation.

```
public static int height(TreeNode<?> node)
{
  if (node == null)
    return -1;
  int lHeight = height(node.left());
  int rHeight = height(node.right());
  if (lHeight > rHeight)
    return lHeight + 1;
  else
    return rHeight + 1;
}
```

Other examples of the applications of traversals are given in the class `ExampleTree`, posted on Blackboard.

Expression Trees

We now present algorithms to build and evaluate expression trees. Both are implemented in the `ExpressionTree` class posted on Blackboard.

From postfix to expression trees. The idea is to use a stack to store subtrees for subexpressions.

Algorithm

Scan the expression character by character from left to right. For each character `c`:

1. If `c` is a blank, skip it.

2. If `c` is a digit, create a node for it and push it on the stack.

3. If `c` is an operator,

   3.1. create a node n for `c`,

   3.2. pop the top of stack and make it n's right subtree,

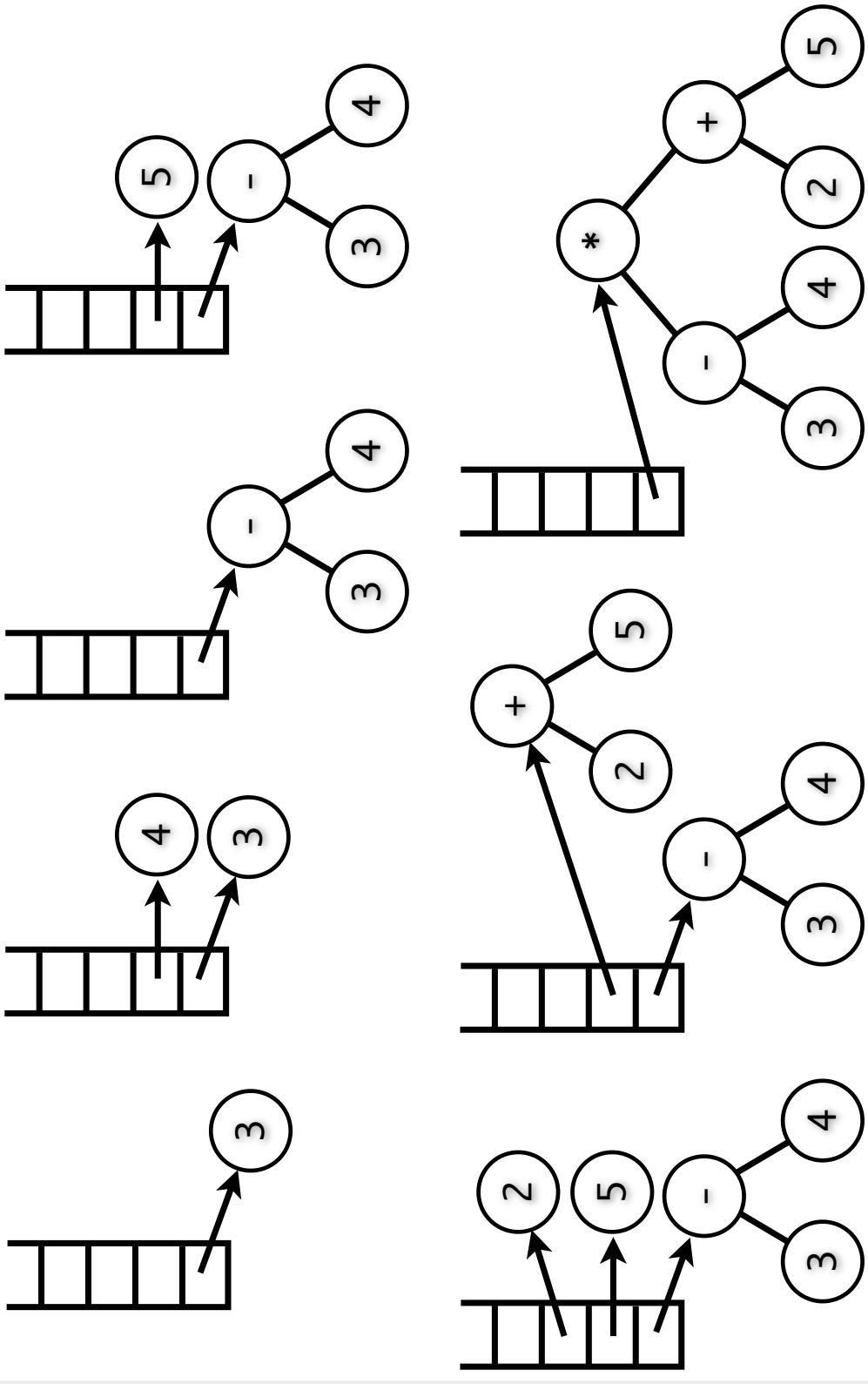
   3.3. pop again, and make it n's left subtree.

At the end, the stack contains a reference to the root of the expression tree.

The `makeFromPostfix()` method in `ExpressionTree` implements this algorithm.

Example. The next page shows how the algorithm handles the expression

```
3 4 - 5 2 + *
```


5
-
4
3
-
4
3
4
3
3
*
+
5
2
-
4
3
+
5
2
-
4
3
2
5
-
4
3

Evaluating an expression tree. We can use postorder traversal to evaluate an expression tree whose root is given:

1. If the root is a leaf, then it must be a number, so just return it.

2. Otherwise, the root must be a binary operator `op`, so

   2.1. Let x_L be the result of evaluating the left subtree.

   2.2. Let x_R be the result of evaluating the right subtree.

   2.3. Return x_L `op` x_R.

The `evaluate()` method in `ExpressionTree` implements this algorithm.

Sets

In mathematics, a ***set*** is a collection of elements with no duplicates. The Java `Set` interface, which extends `Collection`, models this mathematical abstraction. It defines methods for the standard set operations, such as containment, union, and set difference. Our implementations will build on Java's `AbstractSet`, an abstract class that provides a partial implementation of the

`Set` class. `AbstractSet` extends `AbstractCollection`, except that all of the methods and constructors in subclasses of this class must obey the additional constraints imposed by the `Set` interface; for instance, the `add` method must not permit addition of multiple instances of an object to a set.

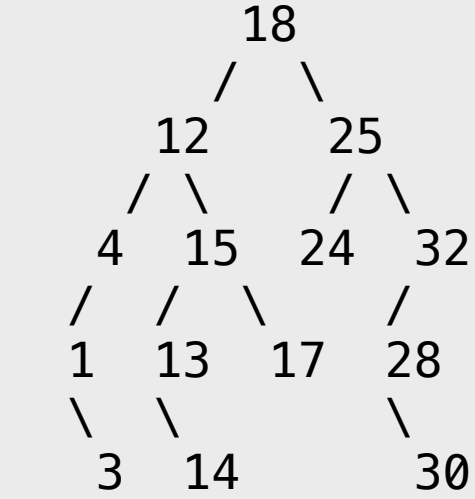
Binary Search Trees

We begin by considering sets where the element type is `Comparable`. Thus, the elements can be ordered, so that there is a minimum element, a maximum element, and any element, other than the min and max, has a successor (next) and a predecessor (previous). In sets like this, the elements are often called ***keys***.

A ***binary search tree*** (BST) is one way to implement a set of comparable elements. A BST is a binary tree whose nodes hold the keys; the keys must satisfy the following.

Binary Search Tree Property: For any node X, every key in the left subtree of X is less than X's key, and every key in the right subtree of X is greater than X's key.

Example. Consider the BST below.



For instance, the root is 18, its left subtree (rooted at 12) contains numbers from 1 to 17, and its right subtree (rooted at 25) contains numbers from 24 to 30.

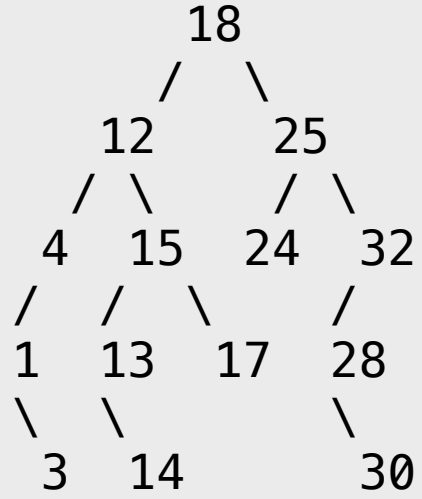
Binary Search Trees

Suppose we want to represent sets where the element type is `Comparable`. That is, the elements have a natural ordering, so that there is a minimum element, a maximum element, and any element, other than the min and max, has a successor (next) and a predecessor (previous). In sets like this, the elements are often called ***keys***.

A ***binary search tree*** (BST) is one way to implement a set of comparable elements. A BST is a binary tree whose nodes hold the keys; the keys must satisfy the following.

Binary Search Tree Property: For any node X, every key in the left subtree of X is less than X's key, and every key in the right subtree of X is greater than X's key.

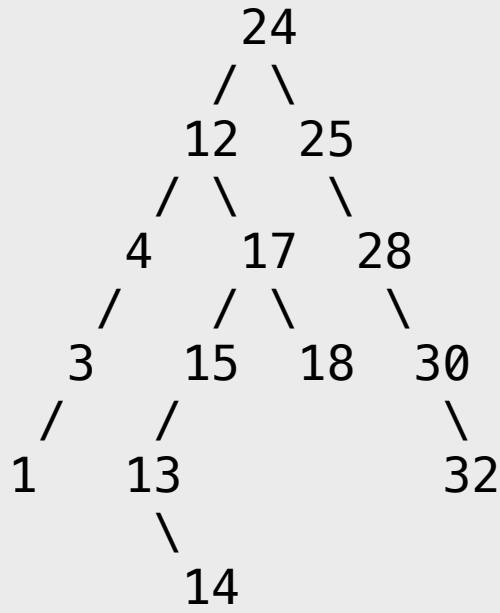
Example. Consider the BST below.



For instance, the root is 18, its left subtree (rooted at 12) contains numbers from 1 to 17, and its right subtree (rooted at 25) contains numbers from 24 to 30.

When a node has only one child, that child is either a left child or a right child, depending on whether its key is smaller or larger than its parent's key.

Note. The BST representation of a set is not unique. Here's another way to represent the set of keys from the previous example:



You should be able to verify the following:

Fact. An inorder traversal of a binary search tree visits the nodes in sorted order.

In this sense, a search tree maintains a sorted list of entries. However, operations on a binary search tree are usually much faster than the same operations on a sorted linked list because you typically only traverse a small fraction of the nodes of a tree. Before explaining why, we need to provide a few implementation details.