Data Structures

CS284

Objectives

- To learn how to use a tree to represent a hierarchical organization of information
- To learn how to use recursion to process trees
- To understand the different ways of traversing a tree
- To understand the difference between binary trees, binary search trees, and heaps
- To learn how to implement binary trees, binary search trees, and heaps using linked data structures and arrays

Trees - Introduction

- All previous data organizations we've learned are linear—each element can have only one predecessor or successor
- Accessing all elements in a linear sequence is $\mathcal{O}(n)$
- Trees are nonlinear and hierarchical
- Tree nodes can have multiple successors (but only one predecessor)
- Trees are recursive data structures because they can be defined recursively

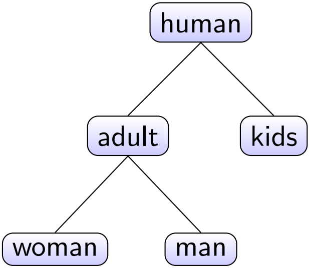
Binary Trees

- Definition and Terminology
- More Examples of Trees
- Binary Search Trees

Tree Traversals

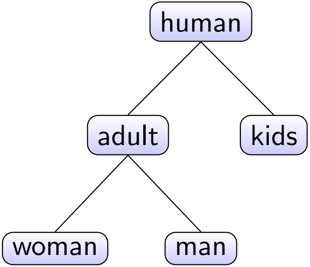
Binary Trees

- We first focus on binary trees
- In a binary tree each element has at most two successors



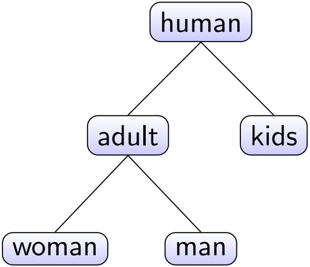
Binary Trees – Terminology

- Node
- Root
- Branches: links between nodes
- Children: successors of a node
- Parent (how many? root?): predecessor of a node
- Siblings: nodes with the same parent

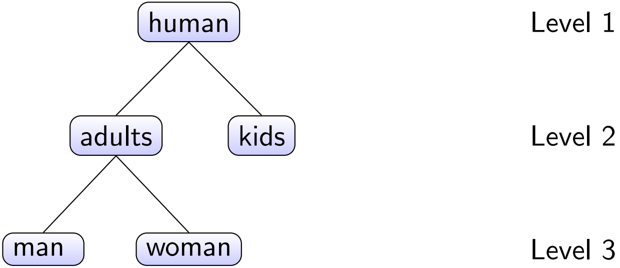


Binary Trees – Terminology (cont.)

- Internal node
- Leaf (= external node)
- Ancestor: generalization of parent-child
- Subtree (of a node): tree whose root is a child of that node



Binary Trees – Terminology (cont.)

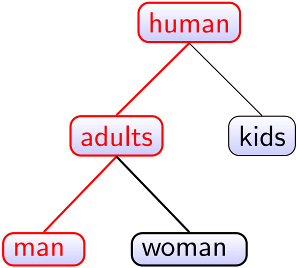


In words:

- If node n is the root of tree T, its level is 1
- If node n is not the root of tree T, its level is $1 +$ the level of its parent

Binary Trees – Terminology (cont.)

Height: number of nodes in the longest path the root to a leaf



Height is 3 in this example

Computing the Height of a Binary Tree

```
private class Node<E> {

    E value;

    Node<E> l_child;
    Node<E> r_child;

    private Node(E value, Node<E> l_child, Node<E> r_child) {
        this.value = value;
        this.l_child = l_child;
        this.r_child = r_child;
    }
}
```

Computing the Height of a Binary Tree (cont.)

```java
public Node<String> build_tree_str() {
    Node<String> men = new Node<String>("men", null, null);
    Node<String> women = new Node<String>("women", null, null);

    Node<String> boys = new Node<String>("boys", null, null);
    Node<String> girls = new Node<String>("girls", null, null);

    Node<String> adults = new Node<String>("adults", men, women);
    Node<String> kids = new Node<String>("kids", boys, girls);

    Node<String> human = new Node<String>("human", adults, kids);

    return human;
}
```

Computing the Height of a Binary Tree (cont.)

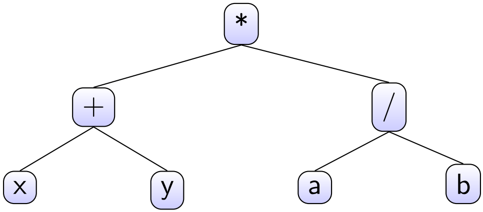
```
public int recursive_get_height(Node<E> root) {

    if (root.l_child == null && root.r_child == null)
        return 1;

    int left_height = 0;
    int right_height = 0;

    if (root.l_child != null)
        left_height = recursive_get_height(root.l_child);

    if (root.r_child != null)
        right_height = recursive_get_height(root.r_child);

    return 1 + Math.max(left_height, right_height);
}
```

Counting the Number of Nodes

```java
public int recursive_count_nodes(Node<E> root) {

    if (root.l_child == null && root.r_child == null)
        return 1;

    int left_count = 0;
    int right_count = 0;

    if (root.l_child != null)
        left_count = recursive_count_nodes(root.l_child);

    if (root.r_child != null)
        right_count = recursive_count_nodes(root.r_child);

    return 1 + left_count + right_count;
}
```

Arithmetic Expression Tree

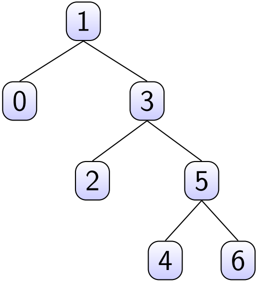
- Each node contains an operator or an operand
- Operands are stored in leaf nodes
- Parentheses are not stored in the tree because the tree structure dictates the order of operand evaluation
- Operators in nodes at higher levels are evaluated after operators in nodes at lower levels



(x + y) * (a/b)

Binary Search Tree

- All elements in the left subtree precede those in the right subtree
- A formal definition: A binary tree T is a *binary search tree* if either of the following is true:
  - $T = \mathit{Empty}$
  - If $T = \mathit{Node}(i, l, r)$, then
    - l and r are binary search trees and
    - i is greater than all values in l and i is less than all values in r



Check Whether Binary Tree is a BST

```
/**
 * Check whether a tree is a BST
 * Step 1: In-Order traversal of the binary tree, store
 * each element in a list
 * Step 2: Check whether the list monotonously increases
 * Pros and cons: inorder_isbst is easier to understand
 * than recursive_is_bst, but it requires O(n) storage space
 * @param root
 * @return
 */
public boolean inorder_isbst(Node<Integer> root) {
    if (root == null) return true;
    if (root.l_child == null && root.r_child == null)
        return true;

    ArrayList<Integer> value_list = new ArrayList<Integer>();

    inOrderTraversal(root, value_list);

    for (int i = 1; i < value_list.size(); i++) {
        if (value_list.get(i) <= value_list.get(i - 1))
            return false;
    }
    return true;
}
```

Check Whether Binary Tree is a BST (cont.)

```
public boolean recursive_is_bst(Node<Integer> root, Integer
lower_bound, Integer upper_bound) {
    if (root == null) return true;
    if (root.value <= lower_bound || root.value >= upper_bound)
    return false;

    return recursive_is_bst(root.l_child, lower_bound, root.value)
    && recursive_is_bst(root.r_child, root.value, upper_bound);
}
```

BST – Search

- Search for a target `key`
- Each probe has the potential to eliminate half the elements in the tree, so searching can be $\mathcal{O}(\log n)$
- In the worst case though, it is $\mathcal{O}(n)$

BST – Search

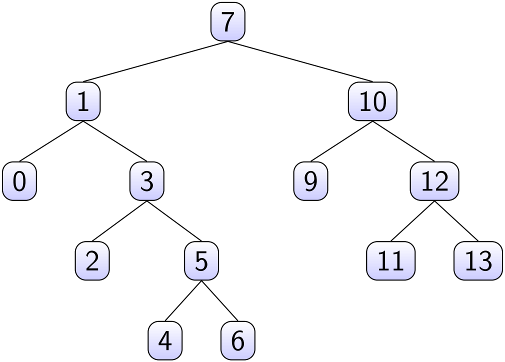
```
public Node<Integer> recursive_search(Node<Integer> root,
int target) {

    if (root == null)
        return null;

    if (root.value == target) {
        return root;
    }
    else if (root.value < target) {
        return recursive_search(root.r_child, target);
    }
    else {
        return recursive_search(root.l_child, target);
    }
}
```

Binary Search Tree Insertion

- A binary search tree never has to be sorted because its elements always satisfy the required order relations
- When new elements are inserted (or removed) properly, the binary search tree maintains its order
- In contrast, an array must be expanded whenever new elements are added, and compacted when elements are removed—expanding and contracting are both $\mathcal{O}(n)$

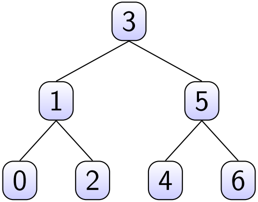
Full, Perfect, and Complete Binary Trees (cont.)

A full binary tree is a binary tree where all nodes have either 2 children or 0 children (the leaf nodes)



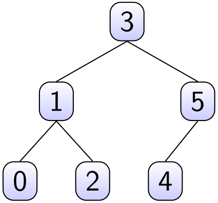
Full, Perfect, and Complete Binary Trees (cont.)

- A perfect binary tree is
  1. a full binary tree of height n
  2. all leaves have the same depth
- Item 2 is equivalent to requiring that the tree have exactly $2^n - 1$ nodes
- In this case, $n = 3$ and $2^n - 1 = 7$



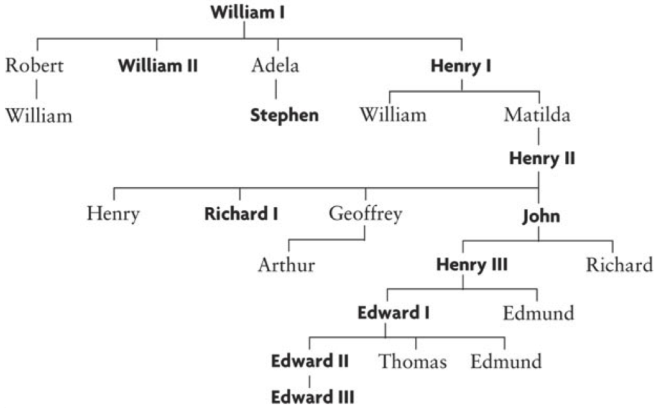
Full, Perfect, and Complete Binary Trees (cont.)

- A complete binary tree is a perfect binary tree through level $n - 1$ with some extra leaf nodes at level n (the tree height), all toward the left



General Trees

Nodes of a general tree can have any number of subtrees



Tree Traversals

- Often we want to determine the nodes of a tree and their relationship
- We can do this by walking through the tree in a prescribed order and visiting the nodes as they are encountered
- This process is called tree traversal
- Three common kinds of tree traversal
  - Inorder
  - Preorder
  - Postorder

Tree Traversals

- Preorder: visit root node, traverse TL, traverse TR
- Inorder: traverse TL, visit root node, traverse TR
- Postorder: traverse TL, traverse TR, visit root node

Algorithm for Preorder Traversal

1. `if` the tree is empty
2. Return.
   `else`
3. Visit the root.
4. Preorder traverse the left subtree.
5. Preorder traverse the right subtree.

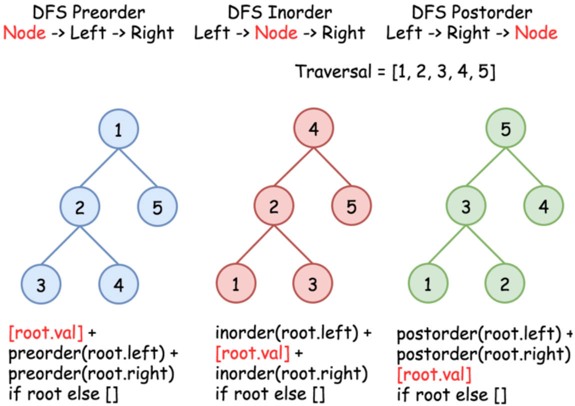
Algorithm for Inorder Traversal

1. `if` the tree is empty
2. Return.
   `else`
3. Inorder traverse the left subtree.
4. Visit the root.
5. Inorder traverse the right subtree.

Algorithm for Postorder Traversal

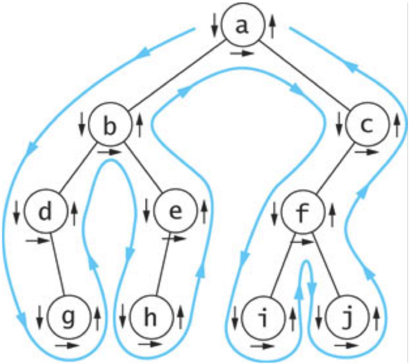
1. `if` the tree is empty
2. Return.
   `else`
3. Postorder traverse the left subtree.
4. Postorder traverse the right subtree.
5. Visit the root.

Traversals


DFS Preorder
Node -> Left -> Right
DFS Inorder
Left -> Node -> Right
DFS Postorder
Left -> Right -> Node
Traversal = [1, 2, 3, 4, 5]
1
2
5
3
4
4
2
5
1
3
5
3
4
1
2
[root.val] +
preorder(root.left) +
preorder(root.right)
if root else []
inorder(root.left) +
[root.val] +
inorder(root.right)
if root else []
postorder(root.left) +
postorder(root.right)
[root.val]
if root else []

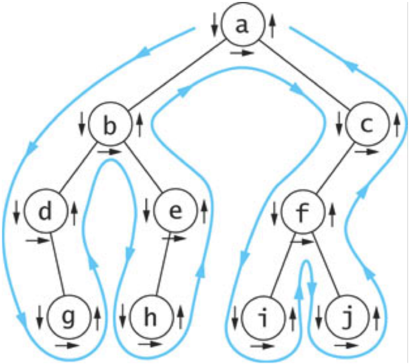
Visualizing Tree Traversals

- You can visualize a tree traversal by imagining a mouse that walks along the edge of the tree
- If the mouse always keeps the tree to the left, it will trace a route known as the Euler tour
- The Euler tour is the path traced in blue in the figure on the right



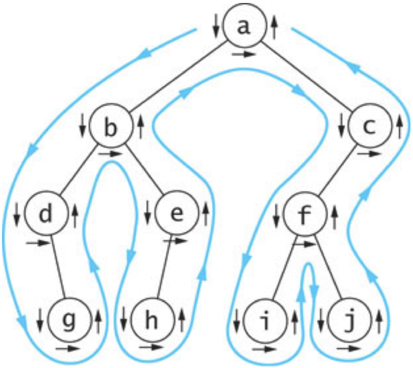
Visualizing Tree Traversals

- An Euler tour (blue path) is a preorder traversal
- The sequence in this example is `a b d g e h c f i j`
- The mouse visits each node before traversing its subtrees (shown by the downward pointing arrows)



Visualizing Tree Traversals

- If we record a node as the mouse returns from traversing its left subtree (horizontal black arrows in the figure) we get an inorder traversal
- The sequence is
  `d g b h e a i f j c`



Visualizing Tree Traversals

- If we record each node as the mouse last encounters it, we get a postorder traversal (shown by the upward pointing arrows)
- The sequence is
  `g d h e b i j f c a`