CSCI 241
Lecture 8
Introduction to Trees
Analyze **ALL** the Sorts!

<table>
<thead>
<tr>
<th></th>
<th>InsertionSort</th>
<th>SelectionSort</th>
<th>MergeSort</th>
<th>QuickSort</th>
<th>RadixSort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best-case</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average-case</td>
<td>same as worst</td>
<td>same as worst</td>
<td>same as worst</td>
<td></td>
<td>same as worst</td>
</tr>
<tr>
<td>Worst-case</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stable?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>In-place?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Announcements

• Nick’s office hours: Tuesdays 1-3pm, CF 167

• Events next week:
  • Monday, Oct. 15 – Tech Talk: Alaska Airlines – 5 pm in CF 110
  • Tuesday, Oct. 16 – Peer Lecture Series: CS Success Workshop – 4 pm in CF 420
  • Wednesday, Oct. 17 – Tech Talk: Integra Group – 5 pm in CF 125
  • Saturday & Sunday, Oct. 20 & 21 – Fall Game Jam! – 10 am in CF 105

• Regular club meetings:
  • Al Club - Tuesdays 6pm in PH 228 (talk to Sakari!)
  • Game Design Club - Mondays 6pm in CF 105 (talk to Kale!)
  • Others - see https://cse.wwu.edu/cs/cs-clubs
Goals:

• Understand the definition of a tree.

• Know the basic terminology associated with trees:
  • Root, child, parent, leaf, height, depth, subtree, descendent, ancestor

• Be able to write a tree class and some simple recursive processing methods.
public class ListNode {
    int value;
    ListNode next;
}

Linked List
Linked List

public class List {
    int value;
    List next;
}

The node is the list. Next points to the tail of the list (also a list!)
Binary Tree

public class Tree {
    int value;
    Tree left;
    Tree right;
}

The node is the tree.
Tree - Definition

**Tree**: like a linked list, but:

- Each node may have zero or more successors (children)
- Each node has exactly one predecessor (parent) except the root, which has none
- All nodes are reachable from root

**Binary tree**: A tree, but:

- Each can have at most two children (left child, right child)
Tree Terminology

*M* is the root of this tree
*G* is the root of the left subtree of *M*
*B*, *H*, *J*, *N*, *S* are leaves (have no children)
*N* is the left child of *P*
*S* is the right child of *P*
*P* is the parent of *N*
*M* and *G* are ancestors of *D*
*P*, *N*, *S* are descendants of *W*
*J* is at depth 2 (length of path from root)
The subtree rooted at *W* has height (length of longest path to a leaf) of 2
A collection of several trees is called a ____?
public class BinaryTreeNode {
    private int value;
    private BinaryTreeNode parent; // (null if no left child)
    private BinaryTreeNode left; // left subtree
    private BinaryTreeNode right; // right subtree
        (null if no right child)
}

public class GeneralTreeNode {
    private int value;
    private GeneralTreeNode parent;
    private List<GeneralTreeNode> children;
}
Why do we need these?
Why do we need these?

to represent hierarchical structure.
Why do we need these? to represent hierarchical structure.
Why do we need these? to represent hierarchical structure.
Why do we need these?
to represent hierarchical structure.
Why do we need these?

to represent **hierarchical structure**.

Syntax Trees:

- In textual representation, **parentheses** show hierarchical structure
- In tree representation, hierarchy is explicit in the tree’s **structure**

Also used for **natural languages** and **programming languages**!
Why do we need these? to implement various ADTs efficiently.

TreeSet, TreeMap

Height of a balanced binary tree is $O(\log n)$

Consequence: Many operations (find, insert, …) can be done in $O(\log n)$ in carefully-designed trees.
Thinking about trees recursively

• A binary tree is
  • Empty, or
  • Three things:
    • value
    • a left binary tree
    • a right binary tree
Thinking about trees recursively

• A binary tree is
  • Empty, or
  • Three things:
    • value
    • a left binary tree
    • a right binary tree
A binary tree is

- Empty, or
- Three things:
  - value
  - a left binary tree
  - a right binary tree
Operations on trees

often follow naturally from the definition of a tree:

• A **binary tree** is

  • Empty, or

  • Three things:
    • value
    • a left **binary tree**
    • a right **binary tree**

Find v in a binary tree:

  (base case - not found!)

  (base case - is this v?)

  (recursive call - is v in left?)

  (recursive call - is v in right?)
Operations on trees

often follow naturally from the definition of a tree:

• A binary tree is
  • Empty, or
  • Three things:
    • value
    • a left binary tree
    • a right binary tree

Find v in a binary tree:

```java
boolean findVal(Tree t, int v):
    if t == null:
        return false
    if t.value == v:
        return true
    return findVal(t.left) || findVal(t.right)
```

(base case - not found!)
```
if t == null:
    return false
```

(base case - is this v?)
```
if t.value == v:
    return true
```

(recursive call - is v in left?)
```
return findVal(t.left)
```

|| (recursive call - is v in right?)
```
|| findVal(t.right)
```

Tree Traversals

Print (or otherwise process) every node in a tree:

• **A binary tree is**

  • Empty, or

  • Three things:
    • value
    • a left binary tree
    • a right binary tree

Print all nodes in a binary tree:

```java
boolean printTree(TreeNode t):
  (base case - nothing to print)
  if t == null:
    return

  (print this node’s value)
  System.out.println(t.value)

  (recursive call - print left subtree)
  printTree(t.left)

  (recursive call - print left subtree)
  printTree(t.right)
```
Tree Traversals

Print (or otherwise process) every node in a tree:

Print all nodes in a binary tree:

```java
boolean printTree(Tree t):
    (base case - nothing to print)
    if t == null:
        return
    (print this node’s value)
    System.out.println(t.value)
    (recursive call - print left subtree)
    printTree(t.left)
    (recursive call - print left subtree)
    printTree(t.right)
```
Tree Traversals

Print (or otherwise process) every node in a tree:

- (base case - nothing to print)
- (print this node’s value)
- (recursive call - print left subtree)
- (recursive call - print left subtree)

Print all nodes in a binary tree:

```java
boolean printTree(Tree t):
    if t == null:
        return
    System.out.println(t.value)
    printTree(t.left)
    printTree(t.right)
```

ABCD: T is a reference to the node with value 5. What is printed by the call `printTree(T)`?

A. 5 4 2 7 8  
B. 7 4 8 5 2  
C. 7 8 4 2 5  
D. 5 4 7 8 2  

(print this node’s value)
(System.out.println(t.value))

(recursive call - print left subtree)
(printTree(t.left))

(recursive call - print left subtree)
(printTree(t.right))
“Walking” over the whole tree is called a tree traversal. This is done often enough that there are standard names. Previous example was a **pre-order traversal**:

1. Process root
2. Process left subtree
3. Process right subtree

Other common traversals:

**in-order traversal:**

1. Process left subtree
2. **Process root**
3. Process right subtree

**post-order traversal:**

1. Process left subtree
2. Process right subtree
3. **Process root**
Why do we need these?

to represent **hierarchical structure**.

Quadtrees in graphics and simulation:

https://www.youtube.com/watch?v=fuexOsLOfl0
Practice Exercise

• Write the values printed by a:
  • pre-order
  • in-order
  • post-order

traversal of this tree.
Terminology - Self-Quiz

- root
- subtree
- leaf
- child
- parent
- ancestor
- descendant
- depth
- height