CSCI 241

Abstract Data Types
Introduction to Trees
Announcements

• Submitting late (using slip days or otherwise) requires sending me email after you submit.

• Videos of Quicksort and Radix Sort runtime analysis will be posted soon after class.

• Today: onward to trees!

• Survey!

• There is a lab this week
Goals:

• Know the difference between an abstract data type and its implementation.

• Understand the motivation for trees:
  • To model **tree-structured data**.
  • To implement **abstract data types**.

• 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 simple recursive methods such as size, height, and traversals (lab 4).
Last Week:
Big-Deal CS Concept #1: Runtime
An abstract data type specifies only interface, not implementation.
Big-Deal CS Concept #2: Interface vs Implementation and Abstract Data Types

An abstract data type specifies only **interface**, not **implementation**.
Big-Deal CS Concept #2: Interface vs Implementation and Abstract Data Types

An abstract data type specifies only **interface**, not **implementation**

- **What** the operations do
- **How** they are accomplished
Abstract Data Types: Examples
Abstract Data Types:
Examples

Collection Interface
Abstract Data Types: Examples

- List, Queue, Stack
Abstract Data Types: Examples

- List, Queue, Stack
- Set
Abstract Data Types: Examples

- List, Queue, Stack
- Set
- Tree
Abstract Data Types: Examples

- List, Queue, Stack
- Set
- Tree
- Priority Queue
Abstract Data Types:
Examples

- List, Queue, Stack
- Set
- Tree
- Priority Queue
- Map

Collection Interface
Abstract Data Types: Examples

- List, Queue, Stack
- Set
- Tree
- Priority Queue
- Map
- Graph
Abstract Data Types: Examples

(145)

(Weeks 4,5,7)

(Weeks 4-6; A2)

(Week 6; A3)

(Week 7; A3)

(Weeks 8-9; A4)
Abstract Data Types: Examples

- List, Queue, Stack (145)
  (Weeks 4, 5, 7)
  (Weeks 4-6; A2)
  (Week 6; A3)
  (Week 7; A3)
  (Weeks 8-9; A4)
Abstract Data Types: Examples

- List, Queue, Stack (145)
- Set (Weeks 4, 5, 7)

  (Weeks 4-6; A2)

  (Week 6; A3)

  (Week 7; A3)

  (Weeks 8-9; A4)
Abstract Data Types: Examples

- List, Queue, Stack (145)
- Set (Weeks 4,5,7)
- Tree (Weeks 4-6; A2)
  (Week 6; A3)
  (Week 7; A3)
  (Weeks 8-9; A4)
Abstract Data Types: Examples

- List, Queue, Stack (145)
- Set (Weeks 4,5,7)
- Tree (Weeks 4-6; A2)
- Priority Queue (Week 6; A3)
  (Week 7; A3)
  (Weeks 8-9; A4)
Abstract Data Types: Examples

- List, Queue, Stack (145)
- Set (Weeks 4, 5, 7)
- Tree (Weeks 4-6; A2)
- Priority Queue (Week 6; A3)
- Map (Week 7; A3)

(Weeks 8-9; A4)
Abstract Data Types: Examples

- List, Queue, Stack  (145)
- Set  (Weeks 4,5,7)
- Tree  (Weeks 4-6; A2)
- Priority Queue  (Week 6; A3)
- Map  (Week 7; A3)
- Graph  (Weeks 8-9; A4)
Abstract Data Types: Examples

- List, Queue, Stack (145)
- Set (Weeks 4, 5, 7)
- Tree (Weeks 4-6; A2)
- Priority Queue (Week 6; A3)
- Map (Week 7; A3)
- Graph (Weeks 8-9; A4)
Interface vs Implementation: Example

Cabinet (interface)

FilingCabinet (Implementation 1)
PilingCabinet (Implementation 2)
Interface vs Implementation: Example

Cabinet:
- Contains(item) - returns true iff item is in the cabinet
- Add(item) - adds item to the cabinet
- Remove(item) - removes item from the cabinet if it exists

FilingCabinet implements Cabinet:
Contains(item):
- look up drawer by first letter range
- find folder by first letter
- search folder for item
- return true if item is found, false otherwise

(short for “if and only if”)
Comparing Implementations

class FilingCabinet:
  • `Contains(item):`
    look up drawer by first letter range
    find folder by first letter
    search folder for item
    return true if item is found, false otherwise

class PilingCabinet:
  • `Contains(item):`
    for each drawer:
      exhaustively search drawer
      if found, return true
    return false
Comparing Implementations

class FilingCabinet:
    • Add(item):
        look up drawer by first letter range
        find folder by first letter
        insert item into folder

class PilingCabinet:
    • Add(item):
        open random drawer
        insert item into drawer
Is an array an ADT?

abstract data type
ADTs and Runtime: Why we care

Runtime comparison of **List** implementations:

<table>
<thead>
<tr>
<th>Class</th>
<th>ArrayList</th>
<th>LinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backing storage</td>
<td>array</td>
<td>chained nodes</td>
</tr>
<tr>
<td>addAt(i, val)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>addFirst(val)</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
<tr>
<td>addLast(val)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>get(i)</td>
<td>O(1) (circled)</td>
<td>O(n) (circled)</td>
</tr>
<tr>
<td>getFirst()</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>getLast()</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

Assume: \(i = \) arbitrary index \(\) \(n = \) array's length
public class ListNode {
    int value;
    ListNode next;
}

Linked List
public class List {
    int value;
    List next;
}
public class List {
    int value;
    List next;
}

The node *is the list.*
Next points to the **tail** of the list (also a list!)
public class Tree {
    int value;
    Tree left;
    Tree right;
}

Binary Tree
Binary Tree

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

The node is the tree. 
left points to the left child of the tree (also a tree!) 
right points to the right child of the tree (also a 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 node can have at most two children (*left child, right child*)

---

**General tree**

**Binary tree**

**Not a tree**

**List-like tree**
Tree Terminology

$M$ is the **root** of this tree

$N$ is the **left child** of $P$

$S$ is the **right child** of $P$

$P$ is the **parent** of $N$
G is the root of M's left subtree
A leaf has no children
B's ancestors are D, G, M
P's descendants are N, S
Height of a tree

Length of the path from root to deepest leaf

Height 3
Depth of a node

Depth of a node is the length of the path from the root to the node.

G has depth 1
Tree Terminology

M is the root of this tree
N is the left child of P
S is the right child of P
P is the parent of N
G is the root of the left subtree of M
B, H, J, N, S are leaves
M and G are ancestors of D
P, N, S are descendants of W
J is at depth 2
The subtree rooted at W has height 2
A collection of several trees is called a forest?
```java
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.
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**.
Why do we need these?

to implement various ADTs **efficiently**.

TreeSet, TreeMap

- unordered collection of unique items
- unordered collections of key-value pairs
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

```java
public class BinaryTreeNode {
    private int value;
    private BinaryTreeNode parent;
    private BinaryTreeNode left;
    private BinaryTreeNode right;
}
```
Thinking about trees recursively

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

```java
public class BinaryTreeNode {
    private int value;
    private BinaryTreeNode parent;
    private BinaryTreeNode left;
    private BinaryTreeNode right;
}
```
Thinking about trees recursively

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

```java
public class BinaryTreeNode {
    private int value;
    private BinaryTreeNode parent;
    private BinaryTreeNode left;
    private BinaryTreeNode right;
}
```
Thinking about trees recursively

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

```java
class BinaryTreeNode {
    int value;
    BinaryTreeNode parent;
    BinaryTreeNode left;
    BinaryTreeNode right;
}
```
Thinking about trees recursively

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

public class BinaryTreeNode {
    private int value;
    private BinaryTreeNode parent;
    private BinaryTreeNode left;
    private BinaryTreeNode 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
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:
Operations on trees

often follow naturally from the definition of a tree:

• **A binary tree** is

  Find v in a binary tree:

  • Empty, or

  • Three things:

    • value

    • a left **binary tree**

    • a right **binary tree**

(base case - not found!)
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?)
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?)
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:

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

|| findVal(t.right)

(recursive call - is v in 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**
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:
boolean printTreeTree(Tree t):
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:

\[
\text{boolean } \text{printTree(Tree } t) : \\
\text{(base case - nothing to print)} \\
\text{if } t == \text{null:} \\
\quad \text{return }
\]
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(Tree t):
  if t == null:
    return
  System.out.println(t.value)
```

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(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)
```
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(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:

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 right subtree)
    printTree(t.right)
```
Tree Traversals

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

- Print this node’s value
- (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
“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
Tree Traversals

“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:
Tree Traversals

“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
Tree Traversals

“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