Trees

Saikrishna Arcot
M. Hudachek-Buswell

August 19, 2018

What is a Tree?

- In computer science, a tree is an abstract model of a hierarchical structure.

What is a Tree?

- In computer science, a tree is an abstract model of a hierarchical structure.
- A tree consists of nodes with a parent-child relationship.

What is a Tree?

- In computer science, a tree is an abstract model of a hierarchical structure.
- A tree consists of nodes with a parent-child relationship.
- Applications:

What is a Tree?

- In computer science, a tree is an abstract model of a hierarchical structure.
- A tree consists of nodes with a parent-child relationship.
- Applications:
  - Organization charts

What is a Tree?

- In computer science, a tree is an abstract model of a hierarchical structure.
- A tree consists of nodes with a parent-child relationship.
- Applications:
  - Organization charts
  - File systems

What is a Tree?

- In computer science, a tree is an abstract model of a hierarchical structure.
- A tree consists of nodes with a parent-child relationship.
- Applications:
  - Organization charts
  - File systems
  - Programming environments

What is a Tree?

- In computer science, a tree is an abstract model of a hierarchical structure.
- A tree consists of nodes with a parent-child relationship.
- Applications:
  - Organization charts
  - File systems
  - Programming environments
  - Your file system

What is a Tree?

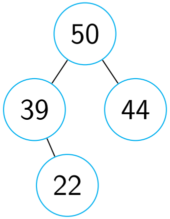
- In computer science, a tree is an abstract model of a hierarchical structure.
- A tree consists of nodes with a parent-child relationship.
- Applications:
  - Organization charts
  - File systems
  - Programming environments
  - Your file system
- Linked lists are trees!

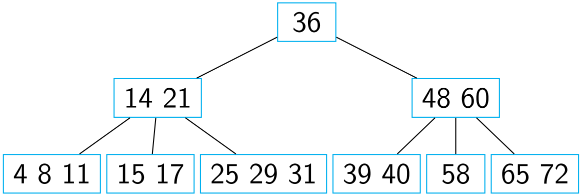
What is a Tree?

- In computer science, a tree is an abstract model of a hierarchical structure.
- A tree consists of nodes with a parent-child relationship.
- Applications:
  - Organization charts
  - File systems
  - Programming environments
  - Your file system
- Linked lists are trees!
- Trees are limited graphs with NO cycles; they are highly recursive.

Example Trees

For our purposes, we will assume no duplicates in the tree(s).





Terminology

Root node: node without a parent, where you enter the tree

Terminology

Root node:	node without a parent, where you enter the tree
Internal node:	node with *at least* one child

Terminology

Root node: node without a parent, where you enter the tree

Internal node: node with *at least* one child

Leaf node/External node: node with *no* children

Terminology

Root node:	node without a parent, where you enter the tree
Internal node:	node with *at least* one child
Leaf node/External node:	node with *no* children
Parent of a node:	node immediately above a node

Terminology

Root node:	node without a parent, where you enter the tree
Internal node:	node with *at least* one child
Leaf node/External node:	node with *no* children
Parent of a node:	node immediately above a node
Ancestors of a node:	parent, grandparent, great-grandparent, etc. of a node

Terminology

Root node:	node without a parent, where you enter the tree
Internal node:	node with *at least* one child
Leaf node/External node:	node with *no* children
Parent of a node:	node immediately above a node
Ancestors of a node:	parent, grandparent, great-grandparent, etc. of a node
Children of a node:	node(s) immediately below a node

Terminology

Root node:	node without a parent, where you enter the tree
Internal node:	node with *at least* one child
Leaf node/External node:	node with *no* children
Parent of a node:	node immediately above a node
Ancestors of a node:	parent, grandparent, great-grandparent, etc. of a node
Children of a node:	node(s) immediately below a node
Descendants of a node:	children, grandchildren, great-grandchildren, etc. of a node

Terminology

Height of a node: maximum number of nodes needed to be traversed to reach a leaf node. In this class, the height of a leaf node is 0. (Bottom-up)

Terminology

Height of a node: maximum number of nodes needed to be traversed to reach a leaf node. In this class, the height of a leaf node is 0. (Bottom-up)

Depth of a node: number of nodes needed to be traversed to reach the root node. In this class, the depth of the root node is 0 (which means the depth of the root node's children is 1). (Top-down) Change from previous version

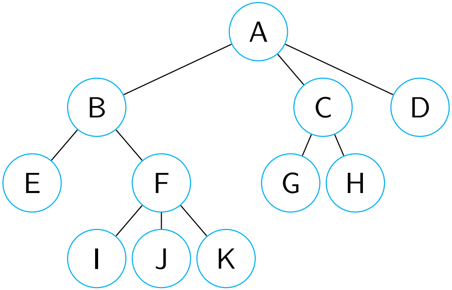
Terminology

Height of a node: maximum number of nodes needed to be traversed to reach a leaf node. In this class, the height of a leaf node is 0. (Bottom-up)

Depth of a node: number of nodes needed to be traversed to reach the root node. In this class, the depth of the root node is 0 (which means the depth of the root node's children is 1). (Top-down) Change from previous version

Subtree of a node: tree consisting of a node and its descendants

Terminology



Which node is the root?
Which nodes are internal nodes?
Which nodes are the leaves?
What is the height of the root? a leaf?
What is the depth of the root? a leaf?
Where are the subtrees?

Tree Characteristics

There are two concepts which distinguish trees into groups: Order and Shape.

Order: refers to the relationship between the parent and child, as well as that between the siblings.

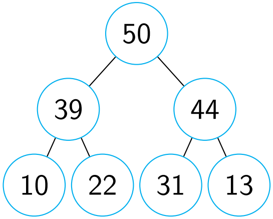
Shape: refers to how the tree is populated and what restrictions are in place for leaves. Shapes to be considered in trees are

- complete
- full
- balanced

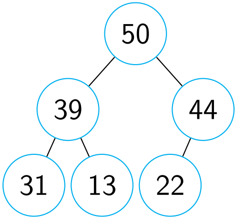
Tree Shapes

Full Trees - every node, except the leaves, has two children.
Complete Trees - all levels, exceot the last, are full, and the leaves are filled left to right.
Balanced Trees - the heights of sibling nodes can differ by at most by 1.

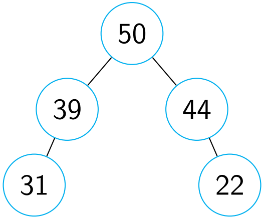
Full



Complete



Balanced



Tree ADT

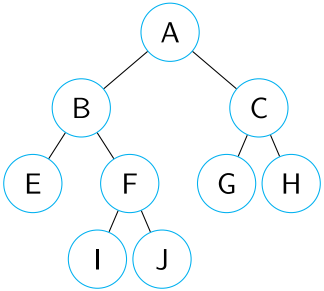
- Use positions to abstract the nodes
- Generic methods:
  - integer size()
  - boolean isEmpty()
  - Iterator iterator()
  - Iterable positions()
- Accessor methods:
  - position root()
  - position parent(p)
  - Iterable children(p)
  - integer numChildren(p)
- Query methods:
  - boolean isExternal(p)
  - boolean isInternal(p)
  - boolean isRoot(p)
- Additional update methods may be defined by data structures implementing the Tree ADT
- Think about the exceptions thrown and the edge cases involved in the Tree ADT

A Note on Coding

Trees are a recursive data structure. For example, the left child of the root of a tree is also a tree on its own. Because of this and other reasons, most operations on trees (adding, removing, searching, etc.) are done recursively instead of iteratively, as using recursion is far easier. While it may be possible to code these methods iteratively, doing so may cause the implementation to be messy and/or inefficient, and will probably make your head hurt.

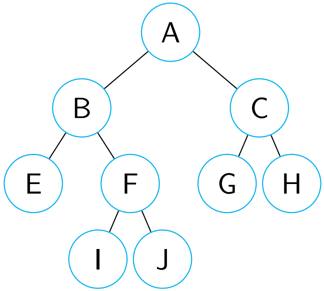
Binary Trees

- A *binary* tree is a tree where each node has at most two children (a left and right child) that are an ordered pair.



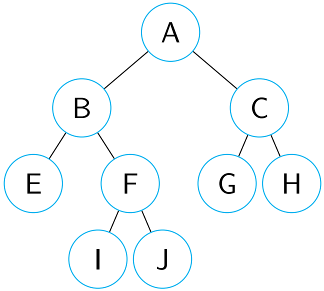
Binary Trees

- A *binary* tree is a tree where each node has at most two children (a left and right child) that are an ordered pair.
- A binary tree is a tree containing a single node, or a tree whose root has an ordered pair of children that are themselves binary trees.



Binary Trees

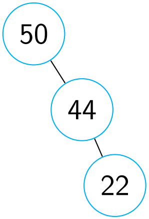
- A *binary* tree is a tree where each node has at most two children (a left and right child) that are an ordered pair.
- A binary tree is a tree containing a single node, or a tree whose root has an ordered pair of children that are themselves binary trees.
- Applications are arithmetic expressions, decision processes, searching



Properties of Binary Trees

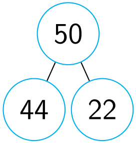
Notation:

- n number of nodes
- e number of external nodes
- i number of internal nodes
- h height



Properties for Proper Binary Trees:

- $h \leq i$
- $e \leq 2^h$



Binary Tree ADT

- The BinaryTree ADT extends the Tree ADT, and inherits all the methods of the Tree ADT
- Position methods:
  - position left(p)
  - position right(p)
  - position sibling(p)
- These additional methods return *null* when there is no left, right, or sibling of p, respectively
- Additional update methods may be defined by data structures implementing the BinaryTree ADT

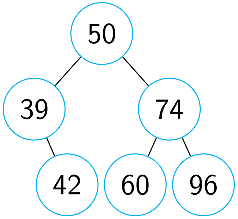
Binary Search Trees

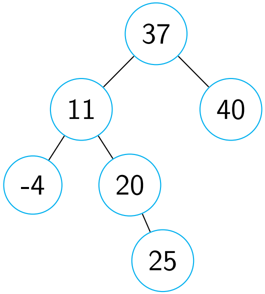
- A *binary search* tree is a special case of a binary tree where the left child of a node is smaller than the current node and the right child of a node is larger than the current node.

Binary Search Trees

- A *binary search* tree is a special case of a binary tree where the left child of a node is smaller than the current node and the right child of a node is larger than the current node.
- This also means that the left subtree contains items that are smaller than the current node, and the right subtree contains items that are larger than the current node.

Binary Search Trees





Tree Traversals

- A traversal visits the nodes of a tree in a systematic manner.

Tree Traversals

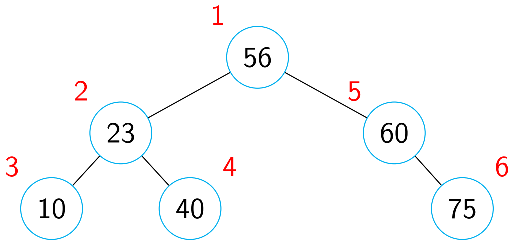
- A traversal visits the nodes of a tree in a systematic manner.
- For all trees, you can perform a preorder traversal, postorder traversal, or a level-order traversal.

Tree Traversals

- A traversal visits the nodes of a tree in a systematic manner.
- For all trees, you can perform a preorder traversal, postorder traversal, or a level-order traversal.
- For binary search trees, you can also perform an inorder traversal, which traverses the data in the tree in ascending order.

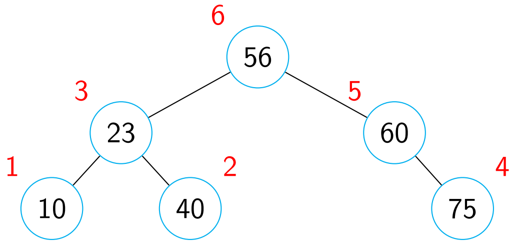
Preorder Traversal

In a preorder traversal, a node is visited first, then the child nodes are visited (from left to right). Mark when visited.
Parent-Left-Right pattern



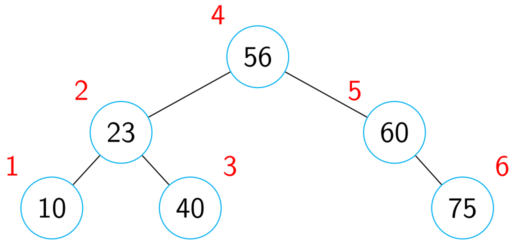
Postorder Traversal

In a postorder traversal, the child nodes are visited first (from left to right), then the node itself is visited. Left-Right-Parent pattern



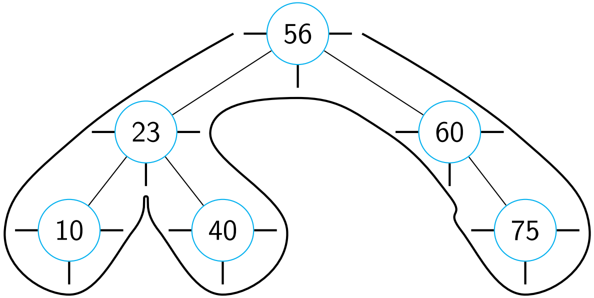
Inorder Traversal

In an inorder traversal, the left subtree of the node is visited first, then the node itself is visited, then the right subtree is visited. This traversal applies only to binary search trees and visits the data items in the tree in ascending order. Left-Parent-Right pattern



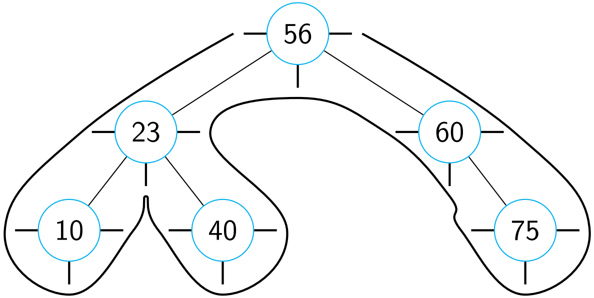
Doing the Traversals on Paper

There is an easy way to do these three traversals on paper. For each node, add three "spokes": one to the left, one downward, and one to the right. Then, starting from the left spoke of the root node, draw a path that goes around the tree and hits each spoke.



Doing the Traversals on Paper

To determine the preorder traversal order, use the left spoke; when the path being drawn hits the left spoke of a node, the node gets added to the preorder traversal. Similarly, use the bottom spoke for an inorder traversal, and the right spoke for a postorder traversal.



Traversal Pseudocode

```
procedure PREORDER(node)
    if node is valid then
        mark this node as visited
        call PREORDER on each of its children
    end if
end procedure
procedure POSTORDER(node)
    if node is valid then
        call POSTORDER on each of its children
        mark this node as visited
    end if
end procedure
```

Traversal Pseudocode

```
procedure INORDER(node)
    if node is valid then
        INORDER(node.left)
        mark this node as visited
        INORDER(node.right)
    end if
end procedure
```