CH 7: TREE

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN C++, GOODRICH, TAMASSIA AND MOUNT (WILEY 2004) AND SLIDES FROM NANCY M. AMATO
• General Trees (Ch. 7.1)
• Binary Trees (Ch. 7.3)
• Tree Traversals (Ch. 7.2)
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 relation.
- Applications:
  - Organization charts
  - File systems
  - Programming environments
FORMAL DEFINITION OF TREE

- A tree $T$ is a set of nodes storing elements in a parent-child relationship with the following properties:
  - If $T$ is nonempty, it has a special node called the root of $T$, that has no parent
  - Each node $v$ of $T$ different from the root has a unique parent node $w$; every node with parent $w$ is a child of $w$
- Note that trees can be empty and can be defined recursively!
- Note each node can have zero or more children
TREE TERMINOLOGY

- **Root**: node without parent (A)
- **Internal node**: node with at least one child (A, B, C, F)
- **Leaf** (aka External node): node without children (E, I, J, K, G, H, D)
- **Ancestors** of a node: parent, grandparent, great-grandparent, etc.
- **Siblings** of a node: Any node which shares a parent
- **Descendant** of a node: child, grandchild, great-grandchild, etc.
EXERCISE

- Answer the following questions about the tree shown on the right:
  - Classify each node of the tree as a root, leaf, or internal node
  - List the ancestors of nodes 3, 7, 8, and 1. Which are the parents?
  - List the descendants of nodes 3, 7, 8, and 1. Which are the children?
- **Depth** of a node: number of ancestors
- **Height** of a tree: maximum depth of any node (3)
Answer the following questions about the tree shown on the right:

- List the depths of nodes 3, 7, 8, and 1.
- What is the height of the tree?
□ **Subtree**: tree consisting of a node and its descendants

□ **Edge**: a pair of nodes \((u, v)\) such that \(u\) is a parent of \(v\) \(((C, H))\)

□ **Path**: A sequence of nodes such that any two consecutives nodes form an edge\((A, B, F, J)\)

□ A tree is **ordered** when there is a linear ordering defined for the children of each node
Draw the subtrees that are rooted at node 2 and at node 3.
We use positions to abstract nodes, as we don’t want to expose the internals of our structure.

Position functions:
- \( p.\text{parent}() \) – return parent
- \( p.\text{children}() \) – list of children positions
- \( p.\text{isRoot}() \)
- \( p.\text{isLeaf}() \)

Tree functions:
- size()
- empty()
- root() – return position for root
- positions() – return list of all positions

Additional functions may be defined by data structures implementing the Tree ADT, e.g., begin() and end().
A LINKED STRUCTURE FOR GENERAL TREES

- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes
- Node objects implement the Position ADT
BINARY TREE
A **binary tree** is a tree with the following properties:
- Each internal node has **two children**
- The children of a node are an ordered pair
- We call the children of an internal node **left child** and **right child**
- Alternative recursive definition: a binary tree is either
  - a tree consisting of a single node, or
  - a tree whose root has an ordered pair of children, each of which is a binary tree

**Applications**
- Arithmetic expressions
- Decision processes
- Searching
**TREE PROPERTIES**

- **Notation**
  - $n$ number of nodes
  - $l$ number of leaves
  - $i$ number of internal nodes
  - $h$ height

- **Properties:**
  - $l = i + 1$
  - $n = 2l - 1$
  - $h \leq i$
  - $h \leq \frac{n-1}{2}$
  - $l \leq 2^h$
  - $h \geq \log_2 l$
  - $h \geq \log_2 (n + 1) - 1$
PROOF BY INDUCTION

- Three parts:
  - Base Cases
  - Inductive hypothesis
  - Proof

- Property: \( l = i + 1 \), where \( l \) is number of leaves and \( i \) is number of internal nodes
  - Base Case: When the tree has a single node, \( l = 1 = 0 + 1, i = 0 \)
  - Induction hypothesis: Let assume \( l = i + 1 \) in \( T \)
  - Proof: \( l' = i' + 1 \), where \( l' = l + 1 \) and \( i' = i + 1 \) in \( T' \)
  - Replace one leaf in \( T \) by an internal node to make \( T' \)
  - \( l' = l - 1 + 2 = l + 1 = i + 1 + 1 = i' + 1 \)
**BINARY TREE TERMINOLOGY**

- **Full/Proper Tree**: All internal node has 2 children.
- **Improper tree**: Each internal node has 1 or 2 children.

**Complete Binary Tree**: let $h$ be the height of the tree.
- for $i = 0 \ldots h - 1$, there are $2^i$ nodes on level $i$.
- at level $h$, nodes are filled from left to right.
EXERCISE

- State whether each of these trees are proper, improper, complete
ARITHMETIC EXPRESSION TREE

- Binary tree associated with an arithmetic expression
  - Internal nodes: operators
  - Leaves: operands
- Example: arithmetic expression tree for the expression \((2 \times (a - 1) + (3 \times b))\)
DECISION TREE

- Binary tree associated with a decision process
  - Internal nodes: questions with yes/no answer
  - Leaves: decisions
- Example: dining decision

```
Want a fast meal?
  Yes
     How about coffee?
        Yes
            Starbucks
        No
            Spike’s
  No
     On expense account?
        Yes
            Al Forno
        No
            Café Paragon
```
The Binary Tree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT.

Additional position methods:
- $p.left()$
- $p.right()$

Update methods may also be defined by data structures implementing the Binary Tree ADT.
A LINKED STRUCTURE FOR BINARY TREE STRUCTURE

- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node
TREE TRAVERSALS
A traversal visits the nodes of a tree in a systematic manner.

In a preorder traversal, a node is visited before its descendants.

Application: print a structured document.

Algorithm preOrder(v):
1. visit(v)
2. for each child w of v
3. preOrder(w)
In a preorder traversal, a node is visited before its descendants.

List the nodes of this tree in preorder traversal order.

Algorithm preOrder(v)
1. visit(v)
2. for each child w of v
3. preOrder(w)
In a **postorder traversal**, a node is visited after its descendants

Application: compute space used by files in a directory and its subdirectories

**Algorithm postOrder(ν)**
1. **for each** child w of ν
2. postOrder(w)
3. visit(ν)

```
Algorithm postOrder(ν)
1. **for each** child w of ν
2. postOrder(w)
3. visit(ν)
```
In a *postorder traversal*, a node is visited after its descendants.

List the nodes of this tree in postorder traversal order.

**Algorithm postOrder(v)**
1. *for each* child $w$ of $v$
2. `postOrder(w)`
3. `visit(v)`
In an *inorder traversal* a node is visited after its left subtree and before its right subtree.

- Application: draw a binary tree
  - \( x(v) = \) inorder rank of \( v \)
  - \( y(v) = \) depth of \( v \)

**Algorithm** \texttt{inOrder}(\( v \))

1. \textbf{if} \( v \).isInternal()
2. \texttt{inOrder}(\( v \).left())
3. \texttt{visit}(\( v \))
4. \textbf{if} \( v \).isInternal()
5. \texttt{inOrder}(\( v \).right())
**EXERCISE**

- In an *inorder traversal* a node is visited after its left subtree and before its right subtree.
- List the nodes of this tree in inorder traversal order.

```plaintext
Algorithm inOrder(v)
1. if v.isInternal()
2. inOrder(v.left())
3. visit(v)
4. if v.isInternal()
5. inOrder(v.right())
```
EXERCISE

- Draw a (single) binary tree $T$, such that
  - Each internal node of $T$ stores a single character
  - A preorder traversal of $T$ yields `EXAMFUN`
  - An inorder traversal of $T$ yields `MAFXUEN`
Specialization of an inorder traversal
- print operand or operator when visiting node
- print "(" before traversing left subtree
- print ")" after traversing right subtree

Algorithm printExpression(\(v\))
1. if \(v\).isInternal()
2. print("(")
3. printExpression(\(v\).left())
4. print(\(v\).element())
5. if \(v\).isInternal()
6. printExpression(\(v\).right())
7. print(")")

\(((2 \times (a - 1)) + (3 \times b))\)
Specialization of a postorder traversal
- recursive method returning the value of a subtree
- when visiting an internal node, combine the values of the subtrees

Algorithm evalExpr($v$)
1. if $v$.isExternal()
2. return $v$.element()
3. $x \leftarrow$ evalExpr($v$.left())
4. $y \leftarrow$ evalExpr($v$.right())
5. $\circ \leftarrow$ operator stored at $v$
6. return $x \circ y$
EXERCISE

- Draw an expression tree that has
  - Four leaves, storing the values 1, 5, 6, and 7
  - 3 internal nodes, storing operations +, -, *, /
    operators can be used more than once, but each internal node stores only one
  - The value of the root is 21
- Generic traversal of a binary tree
- Includes as special cases the preorder, postorder and inorder traversals
- Walk around the tree and visit each node three times:
  - on the left (preorder)
  - from below (inorder)
  - on the right (postorder)
Algorithm eulerTour(v)
1. left_visit(v)
2. if v.isInternal()
3. eulerTour(v.left())
4. bottom_visit(v)
5. if v.isInternal()
6. eulerTour(v.right())
7. right_visit(v)
APPLICATION
PRINT ARITHMETIC EXPRESSIONS

- Specialization of an Euler Tour traversal
  - Left-visit: if node is internal, print "(" 
  - Bottom-visit: print value or operator stored at node 
  - Right-visit: if node is internal, print ")"

Algorithm printExpression(v)
1. if v.isExternal()
2. print v.element()
3. else
4. print "(" 
5. printExpression(v.left())
6. print operator at v
7. printExpression(v.right())
8. print ")"

((2 \times (a - 1)) + (3 \times b))