Trees

22c:19
Fall 2009

Task Tree

- Good for directed search tasks
- Subtree filtering (+/-)
- Not good for learning structure
- No attributes
- Apx 50 items visible
- Lose path to root for deep nodes
- Scroll bar!

Directory Tree
Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
- Internal nodes: operators
- External nodes: operands
- Example: arithmetic expression tree for the infix expression:
  \((2 \times (a - 1) + (3 \times b))\)
- Note: infix expressions require parentheses
- \((2 \times a) - (1+3) \times b\) is a different expression tree

Tree structure

Treemaps on the Web

- People Map: [http://www.truepeers.com/](http://www.truepeers.com/)
Definition of a tree

- A graph is called a **tree** if it is connected and has no cycles (acyclic, circuit-free).
- An acyclic graph is also called **forest**.

![Diagram of a tree](image)

Definition of a Rooted Tree

1. A single node, with no edges, is a tree. The root of the tree is its unique node.
2. Let \( T_1, \ldots, T_k \) \((k \geq 1)\) be trees with no nodes in common, and let \( r_1, \ldots, r_k \) be the roots of those trees, respectively. Let \( r \) be a new node. Then there is a tree \( T \) consisting of the nodes and edges of \( T_1, \ldots, T_k \), the new node \( r \) and the edges \( \langle r, r_1 \rangle, \ldots, \langle r, r_k \rangle \). The root of \( T \) is \( r \) and \( T_1, \ldots, T_k \) are called the subtrees of \( T \).

Recursive definition

Before:
The trees \( T_1, \ldots, T_k \), with roots \( r_1, \ldots, r_k \)

After:
new tree \( T \) rooted at \( r \) with subtrees \( T_1, \ldots, T_k \)
Tree Terminology

- $r$ is called the parent of $r_1, \ldots, r_k$
- $r_1, \ldots, r_k$ are the children of $r$
- $r_1, \ldots, r_k$ are the siblings of one another
- Node $v$ is a descendant of $u$ if
  - $u = v$ or
  - $v$ is a descendant of a child of $u$
  - A path exists from $u$ to $v$.

Tree terminology

- Every node is a descendant of itself.
- If $v$ is a descendant of $u$, then $u$ is an ancestor of $v$.
- Proper descendants:
  - The descendants of a node, other than the node itself.
- Proper ancestors
  - The ancestors of a node, other than the node itself.

Theorem 11.5.2

- Every tree of $n$ vertices has $n-1$ edges.

- Pf. Induction on $n$
- Base case: $n=1$: a single node has 0 edges.
- Induction Hypothesis: Any tree of $m < n$ nodes has $m-1$ edges.
- Inductive case: Let the root of tree $T$ of $n$ nodes be $r$. Suppose $r$ has $k$ subtrees $T_1, T_2, \ldots, T_k$. If each subtree $T_i$ has $a_i$ nodes, then by induction hypothesis, $T_i$ has $a_i - 1$ edges. So $T$ has $k \times (a_1 - 1) + (a_2 - 1) + \ldots + (a_k - 1) = a_1 + a_2 + \ldots + a_k = n - 1$ edges.
Tree terminology

A path from \( u \) to \( v \). Node \( u \) is the ancestor of node \( v \).

Tree terminology

- \( w \) has depth 2
- \( u \) has height 3
- tree has height 4
Important Properties

**Theorem:** Let $T$ be a graph of $n$ nodes. The following statements are logically equivalent:

1. $T$ is a tree.
2. There is a unique path connecting any pair of vertices in $T$.
3. $T$ is connected and has $n - 1$ edges.
4. $T$ is acyclic and has $n - 1$ edges.

**Pf.**

$(1) \Leftrightarrow (2)$

$\Rightarrow$ If $T$ is a tree, then $T$ is connected and there must be a path between any two vertices. If there are two paths between two vertices, then the two paths define a cycle, which is impossible.

$\Leftarrow$ If there is a unique path between any two vertices, then $T$ is connected and has no cycles.

$(1) \Leftrightarrow (3)$

$\Rightarrow$ If $T$ is a tree, then it is connected, acyclic and has $n - 1$ edges, so $(3)$ is true.

$\Leftarrow$ If $T$ is connected and has a cycle, we remove one edge in the cycle without disconnecting $T$. After breaking all cycles, we obtain a tree $T'$. Since $T'$ has $n - 1$ edges, the same as $T$, so no edges are ever removed from $T$, i.e., $T$ cannot have any cycle.
Important Properties

**Theorem:** Let T be a graph of n nodes. The following statements are logically equivalent:

1. T is a tree.
2. There is a unique path connecting any pair of vertices in T.
3. T is connected and has n – 1 edges.
4. T is acyclic and has n – 1 edges.

**Pf.** (1) ⇔ (4)

⇒ If T is a tree, then it is connected, acyclic and has n – 1 edges, so (4) is true.

⇐ If T is acyclic and has k components, say T₁, T₂, ..., Tₖ. Each Tᵢ is a tree which has aᵢ nodes and aᵢ – 1 edges. So T has

\[(a₁ – 1) + (a₂ – 1) + ... + (aₖ – 1) = a₁ + a₂ + ... + aₖ – k = n – k \text{ edges.}\]

Since we also know T has n – 1 edges. n – 1 = n – k, thus k = 1. So T has only one component.

Special kinds of trees

- Ordered vs. unordered
- Binary tree
- Empty vs. non-empty
- Full
- Perfect
- Complete

Ordered trees

- Have a linear order on the children of each node.
- That is, we can clearly identify a 1ˢᵗ, 2ⁿᵈ, ..., kᵗʰ child.
- An unordered tree doesn’t have this property
Binary tree

- An ordered tree with at most two children for each node.
- If node has two child, the 1st is called the left child, the 2nd is called the right child
- If only one child, it is either the right child or the left child

Two trees are not equivalent...One has only a left child; the other has only a right child

Empty binary tree

- Convenient to define an empty binary tree, written \( \Lambda \), for use in this definition:
  - A binary tree is either \( \Lambda \) or is a node with left and right subtrees, each of which is a binary tree.

Full binary tree

- No nodes with only one child
  - Each node either is a leaf or has 2 children
- Theorem 11.5.5: \( \# \) leaves = \( \# \) non-leaves + 1

- Pf. Let \( n \) be the total number of nodes and \( k \) be \( \# \) non-leaves. Since each non-leaf node has two children, the \( \# \) of nodes having a parent is \( 2k \). Each node either is the root or has a parent. So \( n = 2k + 1 \). So \( \# \) leaves = \( n - k = (2k+1) - k = k + 1 \).
Full binary tree
• No nodes with only one child
  — Each node either is a leaf or has 2 children
• Theorem 11.5.5: \( \# \text{leaves} = \# \text{non-leaves} + 1 \)

  
  Another Proof: Induction on the structure of a tree.
  • Base case: \( T \) has a single node: 1 leaf and 0 non-leaf.
  • Induction Hypothesis: For any tree with less than \( n \) nodes, the theorem is true.
  • Inductive case: \( T \) has two subtrees, \( T_1 \) and \( T_2 \). By the hypothesis, \( T_1 \) has \( a \) non-leaves and \( a+1 \) leaves. \( T_2 \) has \( b \) non-leaves and \( b+1 \) leaves. So \( T \) has \( a+b+1 \) non-leaves and \( a+b+2 \) leaves.

Tree terminology & properties
• Height of a node in a tree
  — Length of the longest path from that node to a leaf
  — The height of the root is the height of the tree.
• Depth of a node in a tree
  — Length of the path from the root of the tree to the node

Perfect binary trees
A full binary tree in which all leaves have the same depth
Perfect binary tree

- A full binary tree in which all leaves have the same depth
- Perfect binary tree of height $h$ has:
  - $2^{h+1}-1$ nodes
  - $2^h$ leaves
  - $2^h-1$ non-leaves
- Interesting because it is “fully packed”, the shallowest tree that can contain that many nodes. Often, we’ll be searching from the root. Such a shallow tree gives us minimal number of accesses to nodes in the tree.

Tree terminology

- Forest
  - A finite set of trees
  - In the case of ordered trees, the trees in the forest must have a distinguishable order as well.

Tree Operations

- $\text{Parent}(v)$
- $\text{Children}(v)$
- $\text{FirstChild}(v)$
- $\text{RightSibling}(v)$
- $\text{LeftSibling}(v)$
- $\text{LeftChild}(v)$
- $\text{RightChild}(v)$
- $\text{isLeaf}(v)$
- $\text{Depth}(v)$
- $\text{Height}(v)$
**Parent(v)**

- Return the parent of node \( v \), or \( \Lambda \) if \( v \) is the root.

**Children(v)**

- Return the set of children of node \( v \) (the empty set, if \( v \) is a leaf).

**FirstChild(v)**

- Return the first child of node \( v \), or \( \Lambda \) if \( v \) is a leaf.
RightSibling(v)
• Return the right sibling of v, or Λ if v is the root or the rightmost child of its parent.

LeftSibling(v)
• Return the left sibling of v, or Λ if v is the root or the leftmost child of its parent.

LeftChild(v)
• Return the left child of node v;
• Return Λ if v has no left or right child.
**RightChild(v)**

- Return the right child of node v;
- return $\Lambda$ if v has no right child

**isLeaf(v)**

- Return **true** if node v is a leaf, **false** if v has a child

**Depth(v)**

- Return the depth of node v in the tree.
**Height(v)**

- Return the height of node v in the tree.

---

**Arithmetic Expression Tree**

- Binary tree associated with an arithmetic expression
- Internal nodes: operators
- External nodes: operands
- Example: arithmetic expression tree for the infix expression:
  - \((2 \times (a - 1) + (3 \times b))\)
- Note: infix expressions require parentheses
- \((2 \times a) - (1+3) \times b)\) is a different expression tree

---

**Evaluating Expression Trees**

```plaintext
function Evaluate(ptr P): integer
/* return value of expression represented by tree with root P */
if isLeaf(P) return Label(P)
else
    xL <- Evaluate(LeftChild(P))
    xR <- Evaluate(RightChild(P))
    op <- Label(P)
    return ApplyOp(op, xL, xR)
```

---
**Example**

- Evaluate (A)
  - Evaluate(B) \( \Rightarrow \) 2
  - Evaluate(D)
    - Evaluate(H) \( \Rightarrow \) a
    - Evaluate(I) \( \Rightarrow \) 1

\[ \text{ApplyOp}(\cdot, a, 1) \Rightarrow a - 1 \]
\[ \text{ApplyOp}(x, 2, (a - 1)) \Rightarrow 2 \times (a - 1) \]

- Evaluate(C)
  - Evaluate(F) \( \Rightarrow \) 3
  - Evaluate(G) \( \Rightarrow \) b
  - ApplyOp(\cdot, (2 \times (a - 1)), (3 \times b)) \( \Rightarrow \) 

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

**Traversals**

- Traverse = any well-defined ordering of the visits to the nodes in a tree
  - **PostOrder** traversal
    - Node is considered after its children have been considered
  - **PreOrder** traversal
    - Node is considered before its children have been considered
  - **InOrder** traversal (for binary trees)
    - Consider left child, consider node, consider right child

**PostOrder Traversal**

**procedure** PostOrder(ptr P):

```plaintext```

foreach child Q of P, in order, do

PostOrder(Q)

Visit(P)
```

PostOrder traversal gives:

2, a, 1, -, x, 3, b, x, +

Nice feature: unambiguous ... doesn't need parentheses

-- often used with scientific calculators
-- simple stack-based evaluation
Evaluating PostOrder Expressions

**procedure** PostOrderEvaluate($e_1, ..., e_n$):

  for $i$ from 1 to $n$ do

    if $e_i$ is a number, then push it on the stack

    else

      • Pop the top two numbers from the stack
      • Apply the operator $e_i$ to them, with the right operand being the first one popped
      • Push the result on the stack

PreOrder Traversal

**procedure** PreOrder(ptr $P$):

  Visit($P$)

  foreach child $Q$ of $P$, in order, do

    PreOrder($Q$)

PreOrder Traversal:

+ x 2, -, a, 1, x, 3, b

Nice feature: unambiguous, don’t need parentheses

— Outline order

InOrder Traversal

**procedure** InOrder(ptr $P$):

  // $P$ is a pointer to the root of a binary tree

  if $P = \Lambda$ then return

  else

      InOrder(LeftChild($P$))
      Visit($P$)
      InOrder(RightChild($P$))

InOrder Traversal:

$2 \times a – 1 + 3 \times b$