Trees

Ioan Despi
despi@turing.une.edu.au

University of New England

July 31, 2013
Outline

1. Trees
2. Theorems
3. Rooted Tree
4. Binary Tree
5. Traversal of Binary Trees
6. Polish postfix notation
Recall

- A path is a sequence of consecutive edges in a graph.
Recall

- A **path** is a sequence of consecutive edges in a graph.
  - The **length** of the path is the number of edges traversed.
Recall

- A **path** is a sequence of consecutive edges in a graph.
  - The **length** of the path is the number of edges traversed.
Recall

- A **path** is a sequence of consecutive edges in a graph.
  - The **length** of the path is the number of edges traversed.
Recall

- A **path** is a sequence of consecutive edges in a graph.
  - The **length** of the path is the number of edges traversed.

- A **circuit** is a path which ends at the vertex it begins.
Recall

- A **path** is a sequence of consecutive edges in a graph.
  - The **length** of the path is the number of edges traversed.

- A **circuit** is a path which ends at the vertex it begins.
  - A **loop** is a circuit of length one.
Recall

- A **path** is a sequence of consecutive edges in a graph.
  - The **length** of the path is the number of edges traversed.

![Graph with a path and a circuit]

- A **circuit** is a path which ends at the vertex it begins.
  - A **loop** is an circuit of length one.
- A **nontrivial circuit** (a **cycle**) is a circuit with at least one edge.
A tree is a special type of graph.
Trees

- A tree is a special type of graph.
  - A tree is a connected graph without nontrivial circuits.
Trees

• A tree is a special type of graph.
  ▶ A tree is a connected graph without nontrivial circuits.
  ▶ A tree is a connected circuitless graph.
Trees

- A tree is a special type of graph.
  - A tree is a connected graph without nontrivial circuits.
  - A tree is a connected circuitless graph.
- A forest is composed of one tree or some disconnected trees.
Trees

- A tree is a special type of graph.
  - A **tree** is a connected graph without nontrivial circuits.
  - A **tree** is a connected circuitless graph.
- A **forest** is composed of one tree or some disconnected trees.
- A **terminating vertex** (or a **leaf**) in a tree is a vertex of degree 1.
Trees

- A tree is a special type of graph.
  - A tree is a connected graph without nontrivial circuits.
  - A tree is a connected circuitless graph.
- A forest is composed of one tree or some disconnected trees.
- A terminating vertex (or a leaf) in a tree is a vertex of degree 1.
- An internal vertex (or a branch vertex) in a tree is a vertex of degree greater than 1.
Trees

- A tree is a special type of graph.
  - A tree is a connected graph without nontrivial circuits.
  - A tree is a connected circuitless graph.
- A forest is composed of one tree or some disconnected trees.
- A terminating vertex (or a leaf) in a tree is a vertex of degree 1.
- An internal vertex (or a branch vertex) in a tree is a vertex of degree greater than 1.
- Vertices are sometimes referred to as nodes, particularly when dealing with graph trees.
Example

The following two graphs are trees:
Example

This graph is not a tree:
Lemma

Any tree with at least one edge must have at least one vertex of degree 1.

Proof.

Proof by contradiction: Let us suppose all vertices have degree $\geq 2$.

- We start from any vertex $v_0$ and walk along the edges not walked before, thus arriving at vertices $v_1, v_2, \cdots, v_n, \cdots$ in sequence.
- If all vertices of the tree had degree $\geq 2$, then such walking wouldn’t terminate at any vertex (because its degree $\geq 2$) without going back to one of the vertices walked over before.
- Since the walking has to terminate because we are only allowed to walk on the (finite number of) edges not walked before, we must go back to at least one of the vertices already walked over before.
- A nontrivial circuit would thus be found.
- This would contradict the definition of a tree.
- Hence not all vertices have degree $\geq 2$, i.e., the Lemma is true.
Some Theorems
Some Theorems

**Theorem**

For any $n \geq 1$, a connected graph with $n$ vertices is a tree if and only if it has exactly $n - 1$ edges.

**Proof.**

The theorem is equivalent to the following two parts:

(i) Statement $S_n$: A tree $T_n$ with $n$ vertices has $n - 1$ edges.

(ii) A connected graph with $n$ vertices and $n - 1$ edges is a tree.
Some Theorems

Proof.

Proof of (i) by induction on statement $S_n$.

For $n = 1$, $S_1$ is true because $T_1$ has 1 vertex and 0 edges. Now assume $S_k$ is true. From the Lemma for $n = k + 1$, we can find a vertex $v_0$ of the tree $T_{k+1}$ such that $\delta(v_0) = 1$, see figure on the left. We then remove $v_0$ and its edge to obtain $T_k$, which is obviously still a tree (with $k$ vertices). Hence $T_k$ has $k - 1$ edges because of the induction assumption $S_k$ is true. Thus $T_{k+1}$ has $1 + (k - 1) = k$ edges, i.e., $S_{k+1}$ is also true, proving (i).
Some Theorems

Proof.

Proof of (ii).

- Let $G$ be a connected graph with $n$ vertices and $n - 1$ edges.
- We show $G$ is a tree by showing it has no nontrivial circuits.
- Assume otherwise, i.e., $G$ has a nontrivial circuit $H$. We show that this will lead to a contradiction.
- Since the removal of an edge from circuit $H$ won’t disconnect $G$, we can remove from $G$ sufficient edges ($\geq 1$) so that the resulting subgraph $G^*$ has no nontrivial circuits while remaining connected with $n$ vertices.
- $G^*$ now by definition is a tree, and should have $n - 1$ edges from (i).
- Hence $G$ must have more than $n - 1$ edges when the removed edges are added back. Thus $G$ has no nontrivial circuit.
- Consequently $G$ is a tree, proving (ii).
Example

List all (up to isomorphism) trees of 4 vertices.

- From the Lemma, there exists a vertex, call it \( v_0 \), of degree 1.
- We denote by \( v_1 \) the only vertex which is connected to \( v_0 \).
- Since there are exactly 3 edges in a tree of 4 vertices, the highest degree \( v_1 \) can attain is 3.
- It is also obvious that \( v_1 \) must have at least 2 edges because otherwise edge \( \{v_0, v_1\} \) would be disconnected from the remaining vertices, which would contradict the definition of a tree. Hence we conclude \( 2 \leq \delta(v_1) \leq 3 \) and will enumerate the two cases below:
  - (a) \( \delta(v_1) = 3 \).
  - (b) \( \delta(v_1) = 2 \). This means \( v_1 \) is connected to another vertex \( v_2 \). Since only one extra edge is needed, the edge has to be attached to \( v_2 \).

Hence there are exactly 2 different trees, which are (a) and (b) respectively in the following figure:
Examples

(a) v0 v1
(b) v0 v1 v2
A rooted tree is a tree in which one vertex is designated as the root and has no parent.
Rooted tree

- A **rooted tree** is a tree in which one vertex is designated as the **root** and has no parent.
- Every other node has exactly one parent.
Rooted tree

- A **rooted tree** is a tree in which one vertex is designated as the **root** and has no parent.
- Every other node has exactly one parent.
- The **level** of a vertex is the number of edges in the **unique** walk between the vertex and the root.
A rooted tree is a tree in which one vertex is designated as the root and has no parent.

Every other node has exactly one parent.

The level of a vertex is the number of edges in the unique walk between the vertex and the root.

The height (or depth) of a tree is the maximum level of any vertex there.
Rooted tree

- $u$ is **parent** of $v$ and $w$
- $v$, $w$ are **children** of $u$
- $v$ and $w$ are **siblings**
- $u$ is an **ancestor** of each
- descendants of $u$ which is an ancestor of each
A binary tree is a rooted tree in which each vertex has at most two children.
Binary tree

- A **binary tree** is a rooted tree in which each vertex has at most two children.
- Each child there is designated either a **left child** or a **right child**.
A binary tree is a rooted tree in which each vertex has at most two children.

Each child there is designated either a left child or a right child.

Algebraic expressions with binary operators have an inherent tree-like structure:
Binary tree

- A **binary tree** is a rooted tree in which each vertex has at most two children.
- Each child there is designated either a **left child** or a **right child**.
- Algebraic expressions with binary operators have an inherent tree-like structure:
  - the terminal nodes (leaves) of the tree are the variables or constants in the expression (a, b, c, d, ...), while
A **binary tree** is a rooted tree in which each vertex has at most two children.

Each child there is designated either a **left child** or a **right child**.

Algebraic expressions with binary operators have an inherent tree-like structure:

- the terminal nodes (leaves) of the tree are the variables or constants in the expression (a, b, c, d, ...), while
- the non-terminal nodes are operators (+, −, ×, /).
Binary tree

v is the left child of u

left subtree of w

right subtree of w
Example

Draw a binary tree to represent \(((a - b) \cdot c) + (d/e)\).

Solution.

- The expression is made up of two parts separated by +, so that + becomes the root.
- Look at the expression on the left of the +, order of operation is important, and again see that the expression is made up of two parts separated by a \cdot or multiplication.
- This operation becomes the left child of +.
- Continue in this fashion to complete the left side of the tree then do the same for the right side.
Example

\[(a - b) \cdot c + (d/e)\]
To traverse a tree means to visit every one of its nodes once.
Traversing Binary Trees

- To traverse a tree means to visit every one of its nodes once.
- At each step, we distinguish a root (the current node) and its left and right subtrees.

Three classical ways of traversing a binary tree:

- **Preorder traversal**
- **Inorder traversal**
- **Postorder traversal**
Traversal of Binary Trees

- To \textit{traverse} a tree means to visit every one of its nodes once.
- At each step, we distinguish a root (the current node) and its left and right subtrees.
- These two subtrees are visited recursively, and the distinguishing feature is when the root is visited.
Traversing Binary Trees

- To **traverse** a tree means to visit every one of its nodes once.
- At each step, we distinguish a root (the current node) and its left and right subtrees.
- These two subtrees are visited recursively, and the distinguishing feature is when the root is visited.
- Let \( r, R, L \) denote the root, the right subtree and the left subtree, respectively.
To **traverse** a tree means to visit every one of its nodes once.

At each step, we distinguish a root (the current node) and its left and right subtrees.

These two subtrees are visited recursively, and the distinguishing feature is when the root is visited.

Let $r$, $R$, $L$ denote the root, the right subtree and the left subtree, respectively.

There are three classical ways of traversing a binary tree:
Traversals of Binary Trees

- To **traverse** a tree means to visit every one of its nodes once.
- At each step, we distinguish a root (the current node) and its left and right subtrees.
- These two subtrees are visited recursively, and the distinguishing feature is when the root is visited.
- Let $r$, $R$, $L$ denote the root, the right subtree and the left subtree, respectively.
- There are three classical ways of traversing a binary tree:
  - **preorder traversal**
Traversing of Binary Trees

- To **traverse** a tree means to visit every one of its nodes once.
- At each step, we distinguish a root (the current node) and its left and right subtrees.
- These two subtrees are visited recursively, and the distinguishing feature is when the root is visited.
- Let \( r \), \( R \), \( L \) denote the root, the right subtree and the left subtree, respectively.
- There are three classical ways of traversing a binary tree:
  - preorder traversal
  - inorder traversal
Traversing Binary Trees

- To traverse a tree means to visit every one of its nodes once.
- At each step, we distinguish a root (the current node) and its left and right subtrees.
- These two subtrees are visited recursively, and the distinguishing feature is when the root is visited.
- Let \( r, R, L \) denote the root, the right subtree and the left subtree, respectively.
- There are three classical ways of traversing a binary tree:
  - preorder traversal
  - inorder traversal
  - postorder traversal
Preorder traversal (rLR)

- Visit the root;
- Traverse left subtree in pre-order;
- Traverse right subtree in pre-order.
Inorder traversal (LrR)

- Traverse left subtree in in-order;
- Visit the root;
- Traverse right subtree in in-order.
Postorder traversal (LRr)

- Traverse left subtree in post-order;
- Traverse right subtree in post-order;
- Visit the root.
Examples

For binary tree

```
 a
 / 
 b   c
 / 
 d   e
 / 
 f   g
```

the traversals are as follows

- Preorder: \( a, b, d, c, e, f, g \)
- Inorder: \( d, b, a, f, e, g, c \)
- Postorder: \( d, b, f, g, e, c, a \)
Examples

The binary tree representation of \((a + b \cdot c)/d\) is easily seen as

```
    /
   /\   \\
  +  d  \\
 /\  \\
 a  .
  b\  \\
    c
```

But how is the calculation actually done?
It may be processed via a postorder traversal

\[ (a, b, c, \cdot, +, d, /) \tag{†} \]

by interpreting each binary operation as acting on the 2 quantities immediately to its left.
Polish postfix notation

- That is, $\alpha, \beta, \gamma, \delta, *, \cdots$ means $\alpha, \beta, (\gamma * \delta), \cdots$, if $\alpha, \beta, \gamma, \delta$ are numbers and $*$ is any binary operation.
- This way, the formula is processed successively in following sequence

$$a, \ b, \ c, \ \cdot, \ +, \ d, \ /$$
$$a, \ b \cdot c, \ +, \ d, \ /$$
$$a + b \cdot c, \ d, \ /$$
$$(a + b \cdot c)/d$$

- The sequence (†) is said to be in the **Polish postfix notation**.
Example

Use a binary tree to sort the following list of numbers

15, 7, 24, 11, 27, 13, 18, 19, 9

We note that when a binary tree is used to sort a list, the inorder traversal will be automatically assumed in this unit.

Solution.

To sort a list, just create a binary tree by adding to the tree one item after another, because the insertion is essentially a sorting process.

The binary tree constructed during the course of sorting the above list then reads
Examples

```
15
  7 24
11 27
  9 13
  18
  19
  27
```

Diagram:
```
15
  7 24
11 18
  9 13
  19
  27
```
Examples

- The sorted list is thus
  
  \[7, 9, 11, 13, 15, 18, 19, 24, 27.\]

- This is because, according to the inorder traversal, we have to visit the left subtree of (the root) 15 containing vertices 7, 11, 9 and 13 before visiting 15 itself and then its right subtree.

- However, to visit the subtree containing exactly the vertices 7, 11, 9 and 13 we have to visit first the (empty) left subtree of 7, then 7, then the right subtree of 7 containing vertices 11, 9 and 13.

- Hence 7 is the very first vertex to be visited.

- To visit the right subtree of 7 containing vertices 11, 9 and 13 we visit first 9, then 11 and then 13.

- Hence the first 4 vertices visited are 7, 9, 11 and 13.

- The rest is similar.

- We note that a better tree sorting algorithm will involve balancing the trees. However such additional features fall beyond the scope of the current unit.