Overview

• HW7 due on next Tuesday 11/19
• Midterm#3 in one week on 11/21
  – (algo/number/graph/tree), review next lecture.
• Last Lecture:
  – Graph Connectivity of Graphs, Shortest Path Problem,
• This Lecture: Trees & Counting/Combinatorics
  – Traveling salesman problem, Trees, Spanning Trees, Rooted Trees, Binary Trees
  – Rooted Tree Apps & Tree Representations & Tree Traversal Algorithms

Application: The Traveling Salesman Problem

• The traveling salesman problem is one of the classical problems in computer science.

  A traveling salesman wants to visit all major cities and then return to his starting point (any city!). Of course he wants to save time and energy, so he wants to determine the shortest path for his trip.

  We can represent the cities and the distances between them by a weighted, complete, undirected graph.

  The problem then is to find a Hamiltonian cycle of minimum total weight that visits each vertex exactly once.
• **Example:** What path would the traveling salesman take to visit the following cities?

![Diagram of cities Chicago, New York, Boston, Toronto with distances (600, 700, 650, 550, 200, 700)].

**Solution:** The shortest path is Boston, New York, Chicago, Toronto, Boston (2,000 miles).

• **Question:** Given \( n \) vertices, how many different cycles (with all \( n \) vertices) can we form by connecting these vertices with edges?

**Solution:** We first choose a starting point. Then we have \((n - 1)\) choices for the second vertex in the cycle, \((n - 2)\) for the third one, and so on, so there are \((n - 1)!\) choices for the whole cycle.

However, this number includes identical cycles that were constructed in **opposite directions**. Therefore, the actual number of different cycles is \((n - 1)!/2\).

• Unfortunately, no algorithm solving the traveling salesman problem with polynomial worst-case time complexity has been devised yet.

• This means that for large numbers of vertices, solving the traveling salesman problem is **not tractable (impractical)**.

• In these cases, we can use efficient **approximation algorithms** that determine a path whose length may be slightly larger than the traveling salesman’s path.
6.2 Introduction to Trees

- Definition: A graph $G$ is said to be a **tree** if it is connected and has no cycle (**acyclic**).
- $G$ is said to be a **forest** if it consists of several trees.
• Definition: a **spanning tree** $G'$ of an undirected graph $G$ satisfies:
  - $G'$ is a subgraph of $G$
  - $G'$ consists of all vertices in $G$
  - $G'$ is a tree (forest then you have **spanning forest**)

\[
G = (V, E) \quad G' = (V', E')
\]

Several Spanning trees $G'$ of $G$

Several Spanning forests $G'$ of $G$

---

**Kruskal Algorithm** (to find a spanning tree):

**Input**: a connected graph $G=(V, E)$

**Output**: a spanning tree $(V, T)$ of $G$

1. $T = \emptyset$
2. for each $e \in E$
   - if $(V, \{e\} \cup T)$ is acyclic then
     - $T = T \cup \{e\}$
3. return $(V, T)$

Note: above algorithm can be modified to obtain minimum cost spanning tree
Definition: A **rooted tree** is a connected acyclic graph with one node designated as the **root** of the tree

- The nodes $r_1, r_2, \ldots, r_t$ are **children** of $r$, and $r$ is a **parent** of $r_1, r_2, \ldots, r_t$
- A node with no children is called a **leaf**; all nonleaves are **internal** nodes.
- The **depth of a node** in a tree is the length of the path from the root to the node.
- The **height of the tree** is the maximum depth of any node in the tree.
- Branches: subtrees/subgraphs
Binary Trees is a rooted tree where each node has at most 2 children.

A full binary tree occurs when all internal nodes have 2 children and all leaves are the same depth.

A complete binary tree is an almost-full binary tree except for the deepest depth.

Note that a complete tree is not a complete graph!

Application of Rooted Trees: Counting problem

- A child can choose one jellybean out of two jellybeans (red, black), and one gummy bear out of three gummy bears (yellow, green, white). How many different sets of candy can the child have?

Y: R, Y ✓
G: R, G ✓
W: R, W ✓
B
choose
jellybean

Y: B, Y ✓
G: B, G ✓
W: B, W ✓
choose
gummy bear

∴ 6 outcomes
• Family tree - not only interesting but also useful for research in medical genetics.
• Files on your computer are organized in a hierarchical (treelike) structure (nested folders).
• Algebraic expression involving binary operations can be represented by a labeled binary tree.

\[
(2 + x) - (y \times 3)
\]

\[
2 + (x - y) \times 3
\]

**Binary Tree Representation**

Binary trees have special characteristics: the identity of the left and right child.

Binary tree

<table>
<thead>
<tr>
<th></th>
<th>left child</th>
<th>right child</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Left child-right child array representation

<table>
<thead>
<tr>
<th></th>
<th>left child</th>
<th>right child</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Pointer representation
## Tree Traversal Algorithms

**Traversal of Tree**: visit every nodes of a tree in a systematic order

The 3 common tree traversal algorithms are: **preorder**, **inorder**, and **postorder** traversal.

These terms refer to the order in which the root of a tree is visited compared to the subtree nodes.

In these traversal methods, it is helpful to use the recursive view of a tree, where the root of a tree is a parent of the roots of subtrees.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Preorder</strong></td>
<td>PR(T) //Writes the nodes of a tree with root r in preorder</td>
<td></td>
</tr>
<tr>
<td></td>
<td>write(r)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>for i = 1 to t do</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PR(T_i)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>end for</td>
<td></td>
</tr>
<tr>
<td><strong>Inorder</strong></td>
<td>IN(tree T) //Writes the nodes of a tree with root r in inorder</td>
<td></td>
</tr>
<tr>
<td></td>
<td>write(r)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>for i = 2 to t do</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IN(T_i)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>end for</td>
<td></td>
</tr>
<tr>
<td><strong>Postorder</strong></td>
<td>PO(tree T) //Writes the nodes of a tree with root r in Postorder</td>
<td></td>
</tr>
<tr>
<td></td>
<td>for i = 1 to t do</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PO(T_i)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>end for</td>
<td></td>
</tr>
</tbody>
</table>
e.g. Do a preorder, inorder, and postorder traversal of the tree.

Preorder: a, b, e, f, c, d, g, i, h
Inorder: e, b, f, a, c, i, g, d, h
Postorder: e, f, b, c, i, g, h, d, a

Algebraic expressions represented as binary trees

- **Infix notation**: operation symbol appears between the 2 operands. (Lisp)
- **Prefix notation**: operation symbol precedes its operands. (Lisp)
- **Postfix notation**: operation symbol follows its operands. (PS)