CS 106X, Lecture 21
Tries; Graphs

reading:
Programming Abstractions in C++, Chapter 18
Plan For Today

• Tries
• Announcements
• Graphs
• Implementing a Graph
• Representing Data with Graphs
Plan For Today

- Tries
- Announcements
- Graphs
- Implementing a Graph
- Representing Data with Graphs
Lexicons are good for storing words
- contains
- containsPrefix
- add
The Lexicon
The Lexicon

contains?
containsPrefix?
add?
The Lexicon

STARTING
START
The Lexicon

• We want to model a set of words as a tree of some kind
• The tree should be sorted in some way for efficient lookup
• The tree should take advantage of words containing each other to save space and time
Tries

**trie ("try")**: A tree structure optimized for "prefix" searches

```c
struct TrieNode {
    bool isWord;
    TrieNode* children[26]; // depends on the alphabet
};
```
<table>
<thead>
<tr>
<th>isWord: false</th>
<th>&quot;ace&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b c d e ...</td>
<td>&quot;ac&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;a&quot;</td>
</tr>
<tr>
<td>isWord: true</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>a b c d e ...</td>
<td></td>
</tr>
</tbody>
</table>
Reading Words

Yellow = word in the trie
Reading Words
Reading Words
Reading Words
Reading Words
Reading Words
Reading Words

What are all the words represented by this trie?
What are all the words represented by this trie?

a
as
ha
haha
he
she
What are all the words represented by this trie?

a
as
ha
haha
he
she

How can we write a function to print this?
```cpp
void printAllWords(TrieNode* root) {
    printAllWordsHelper(root, ";");
}

void printAllWordsHelper(TrieNode* root, string prefix) {
    if (root == nullptr) {
        return;
    }
    if (root->isWord) {
        cout << prefix << endl;
    }
    for (int i = 0; i < 26; i++) {
        printAllWordsHelper(root->children[i], prefix + char('a' + i));
    }
}
```
Prefixes

How can we write `containsPrefix`?

containsPrefix("a") => true
containsPrefix("se") => false
bool containsPrefix(TrieNode* node, string prefix) {
    if (node == nullptr) {
        return false;
    }
    if (prefix.length() == 0) {
        return true;
    }
    return containsPrefix(node->children[prefix[0] - 'a'], prefix.substr(1));
}
Plan For Today

• Tries
• Announcements
• Graphs
• Implementing a Graph
• Representing Data with Graphs
Announcements

- Brown Institute Data Visualization @ NY Times talk with Kevin Quealy Tues. 11/13 4:30PM in Packard 101
Plan For Today

- Tries
- Announcements
- Graphs
  - Implementing a Graph
  - Representing Data with Graphs
Linked Structures

2 → 5 → 8
Linked Structures
Linked Structures

What about bidirectional links?
What about representing cycles?
What about storing data in the links, like distances or costs?
Representing airline flights and distances:

- HNL to LAX: 2555
- LAX to SFO: 337
- SFO to ORD: 1843
- ORD to PVD: 849
- LAX to DFW: 1743
- DFW to MIA: 1387
- DFW to LGA: 1120
- LGA to PVD: 142
- MIA to HNL: 1099
Linked Structures: Shortfalls
Linked Structures: Shortfalls
Linked Structures: Shortfalls

- simple C++
- collections
- function calls
- recursion
  - fractals
  - exploration recursion
  - definition recursion
A graph consists of a set of **nodes** connected by **edges**.
A graph consists of a set of **nodes** connected by **edges**.
Nodes

Nodes
(“Vertices”)
Edges

Edges ("Arcs")

Directed
Edges

("Arcs")

Unweighted
Weights

A weight is a cost associated with a given edge.
Degree

Degree (# edges touching a node)

Diagram showing nodes with varying degrees: 5, 3, 2, 1, and 1.
Degree

In-degree (# edges coming into a node)
Out-degree (# edges leaving a node)
Degree

Degree = In-degree + Out-degree
Neighbours (nodes you can directly reach)
Neighbors
(nodes you can directly reach)
Neighbors

Nodes you can directly reach
Neighbors
(nodes you can directly reach)
Neighbors

(neighbors you can directly reach)
Paths

Path (sequence of edges that go from one node to another)

Paths are represented as nodes visited or edges taken

Path length = 3 (number of nodes or edges in the path)
Path (sequence of edges that go from one node to another)

Node \( x \) is reachable from node \( y \) if a path exists from \( y \) to \( x \).
Cycle (path that begins and ends at the same node)
Loop (an edge that connects a node to itself)
A graph is **connected** if every node is reachable from every other node.
A graph is **complete** if every node has a direct edge to every other node.
Graph Properties

A graph is **acyclic** if it does not contain any cycles.
A graph is *weighted* if its edges have weights, or *unweighted* if its edges do not have weights.
A graph is **directed** if its edges have direction, or **undirected** if its edges do not have direction (aka are bidirectional).
Graph Properties

- Connected or unconnected
- Acyclic
- Directed or undirected
- Weighted or unweighted
- Complete

Which of these properties do binary trees have?

What about linked lists?
• A *binary tree* is a graph with some restrictions:
  – The tree is an unweighted, directed, acyclic graph (DAG).
  – Each node's in-degree is at most 1, and out-degree is at most 2.
  – There is exactly one path from the root to every node.

• A *linked list* is also a graph:
  – Unweighted DAG.
  – In/out degree of at most 1 for all nodes.
Graph examples

• For each, what are the vertices and what are the edges?
  – Web pages with links
  – Functions in a program that call each other
  – Road maps (e.g., Google maps)
  – Airline routes
  – Facebook friends
  – Course pre-requisites
  – Family trees
  – Paths through a maze
  – Chemical bonds
Plan For Today

- Tries
- Announcements
- Graphs
- Implementing a Graph
- Representing Data with Graphs
Adjacency List

- Map<\textit{Node}, \textit{Vector\<Node\>}>  
  - or Map<\textit{Node}, \textit{Set\<Node\>}>  

<table>
<thead>
<tr>
<th>Node</th>
<th>Set&lt;\textit{Node}&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Node1" /></td>
<td><img src="image" alt="Set1" /></td>
</tr>
<tr>
<td><img src="image" alt="Node2" /></td>
<td><img src="image" alt="Set2" /></td>
</tr>
<tr>
<td><img src="image" alt="Node3" /></td>
<td><img src="image" alt="Set3" /></td>
</tr>
<tr>
<td><img src="image" alt="Node4" /></td>
<td><img src="image" alt="Set4" /></td>
</tr>
</tbody>
</table>
**Adjacency Matrix**

- Store a boolean grid, rows/columns correspond to nodes
  - Alternative to Adjacency List

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
Edge List

• Store a Vector<\textit{Edge}> (or Set<\textit{Edge}>)
  – \textit{Edge} struct would have the two nodes

Vector<\textit{Edge}>
Stanford BasicGraph

• The Stanford C++ library includes a **BasicGraph** class representing a weighted, directed graph.
  – Based on an older library class named **Graph**

• You can construct a graph and add vertices/edges:

```cpp
#include "basicgraph.h"
...
BasicGraph graph;
graph.addVertex("a");
graph.addVertex("b");
graph.addVertex("c");
graph.addVertex("d");
graph.addEdge("a", "c");
graph.addEdge("b", "c");
graph.addEdge("c", "b");
graph.addEdge("b", "d");
graph.addEdge("c", "d");
```
#include "basicgraph.h"  // a directed, weighted graph

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>g.addEdge(v1, v2);</code></td>
<td>adds an edge between two vertexes</td>
</tr>
<tr>
<td><code>g.addVertex(name);</code></td>
<td>adds a vertex to the graph</td>
</tr>
<tr>
<td><code>g.clear();</code></td>
<td>removes all vertexes/edges from the graph</td>
</tr>
<tr>
<td><code>g.getEdgeSet()</code></td>
<td>returns all edges, or all edges that start at v, as a Set of pointers</td>
</tr>
<tr>
<td><code>g.getEdgeSet(v)</code></td>
<td>returns all edges, or all edges that start at v, as a Set of pointers</td>
</tr>
<tr>
<td><code>g.getNeighbors(v)</code></td>
<td>returns a set of all vertices that v has an edge to</td>
</tr>
<tr>
<td><code>g.getVertex(name)</code></td>
<td>returns pointer to vertex with the given name</td>
</tr>
<tr>
<td><code>g.getVertexSet()</code></td>
<td>returns a set of all vertexes</td>
</tr>
<tr>
<td><code>g.isNeighbor(v1, v2)</code></td>
<td>returns true if there is an edge from vertex v1 to v2</td>
</tr>
<tr>
<td><code>g.isEmpty()</code></td>
<td>returns true if queue contains no vertexes/edges</td>
</tr>
<tr>
<td><code>g.removeEdge(v1, v2);</code></td>
<td>removes an edge from the graph</td>
</tr>
<tr>
<td><code>g.removeVertex(name);</code></td>
<td>removes a vertex from the graph</td>
</tr>
<tr>
<td><code>g.size()</code></td>
<td>returns the number of vertexes in the graph</td>
</tr>
<tr>
<td><code>g.toString()</code></td>
<td>returns a string such as &quot;{a, b, c, a -&gt; b}&quot;</td>
</tr>
</tbody>
</table>
Using BasicGraph

• The graph stores a struct of information about each vertex/edge:

```cpp
struct Vertex {
    string name;
    Set<Edge*> edges;
    ...
};

struct Edge {
    Vertex* start;
    Vertex* finish;
    double weight;
    ...
};
```

• These are returned as pointers by various functions:

```cpp
// example usage
Vertex* vc = graph.getVertex("c");
for (Vertex* neighbor : graph.getNeighbors(vc)) {
    cout << neighbor->edges << endl;
}

Edge* edgeAC = graph.getEdge("a", "c");
cout << edgeAC->start->name << endl;   // a
cout << edgeAC->end->name   << endl;   // c
```
Plan For Today

• Tries
• Announcements
• Graphs
• Implementing a Graph
• Representing Data with Graphs
Twitter Influence

https://about.twitter.com/en_us/company/brand-resources.html
Twitter Influence

• Twitter lets a user follow another user to see their posts.
• Following is directional (e.g. I can follow you but you don’t have to follow me back 😞)
• Let’s define being influential as having a high number of followers-of-followers.
  – Reasoning: doesn’t just matter how many people follow you, but whether the people who follow you reach a large audience.

• Write a function mostInfluential that reads a file of Twitter relationships and outputs the most influential user.

https://about.twitter.com/en_us/company/brand-resources.html
Recap

• Tries
• Announcements
• Graphs
• Implementing a Graph
• Representing Data with Graphs

Next time: Graphs and Graph Algorithms