Administrative Details

- Lab 9: Super Lexicon is online
  - Partners are permitted this week!
Last Time

- **Efficient Binary search trees (Ch 14)**
  - AVL Trees
    - Height is $O(\log n)$, so all operations are $O(\log n)$
  - Red-Black Trees
    - Different height-balancing idea: height is $O(\log n)$
    - All operations are $O(\log n)$
Today’s Outline

• Lab 9: Super Lexicon
• Introduction To Graphs
  • Basic Definitions and Properties
  • Applications and Problems
Lab 9 : Lexicon

- Goal: Build a data structure that can efficiently store and search a large set of words
- A special kind of tree called a trie

A trie is a letter-tree that efficiently stores strings. A node in a trie represents a letter. A path in the trie traces out a sequence of letters that represent a prefix or word in the lexicon. You can store and search a large set of words in a trie, but you can only add a word to the lexicon—there are no delete operations.
Lab 9 : Tries

• A trie is a tree that stores words where
  • Each node holds a letter
  • Some nodes are “word” nodes (dark circles)
  • Any path from the root to a word node describes one of the stored words
  • All paths from the root form prefixes of stored words (a word is considered a prefix of itself)
Tries

Now add “dot” and “news”
Now remove “not” and “zen”
All paths in the trie must eventually lead to a word. If the word being removed was the only valid word along this path, the nodes along that path must be deleted from the trie along with the word. For example, if you removed the words *zen* and *not* from the trie shown previously, you should have the result below.

As a general observation, there should never be a leaf node whose `isWord` field is false. If a node has no children and does not represent a valid word (i.e., `isWord` is false), then this node is not part of any path to a valid word in the trie and such nodes should be deleted when removing a word.

In some cases, removing a word from the trie may not require removing any nodes. For example, if we were to remove the word *new* from the above trie, it turns off `isWord` but all nodes along that path are still in use for other words.

**Important note:** when removing a word from the trie, the only nodes that may require deallocation are nodes on the path to the word that was removed. It would be extremely inefficient if you were to traverse the whole trie to check for deallocating nodes every time a word was removed, and you should not use such an inefficient strategy.

**Other trie operations**

There are few remaining odds and ends to the trie implementation. Creating an iterator and writing the words to a file both involve a recursive exploration of all paths through the trie to find all of the contained words.

Remember that in both cases it is only words (not prefixes) that you want to operate on and that these operations need to access the words in alphabetical order.

Once you have a working lexicon, you're ready to implement the snazzy spelling correction features. There are two additional Lexicon member functions, one for suggesting simple corrections and the second for regular expressions matching:

- `Set<string> *SuggestCorrections(string target, int maxDistance);`
- `Set<string> *MatchRegex(string pattern);`

**Suggesting corrections**

First consider the member function `SuggestCorrections`. Given a (potentially misspelled) target string and a maximum distance, this function gathers the set of words from the lexicon that have a distance to the target string less than or equal to the given maxDistance. We define the distance between two equal-length strings to be the total number of character positions in which the strings differ. For example, "place" and "peace" have distance 1, "place" and "plank" have distance 2. The returned set contains all words in the lexicon that are the same length as the target string and are within the maximum distance.
Lab 9 : Lexicon

An interface that provides the methods

```
public interface Lexicon {
    public boolean addWord(String word);
    public int addWordsFromFile(String filename);
    public boolean removeWord(String word);
    public int numWords();
    public boolean containsWord(String word);
    public boolean containsPrefix(String prefix);
    public Iterator<String> iterator();
    public Set<String> suggestCorrections(String target, int maxDistance);
    public Set<String> matchRegex(String pattern);
}
```
Lab 9

- Implement a program that creates, updates, and searches a Lexicon
  - Based on a LexiconTrie class
    - Each node of the Trie is a LexiconNode
    - Analogous to a SLL consisting of SLLNodes
  - LexiconTrie implements the Lexicon Interface
  - Supports
    - adding/removing words
    - searching for words and prefixes
    - reading words from files
    - Iterating over all words
Graphs Describe the World

- Transportation Networks
- Communication Networks
- Molecular structures
- Dependency structures
- Scheduling
- Matching
- Graphics Modeling
- ....

\(^1\)But don’t tell Tom Garrity---he’ll just be sad....
Nodes = subway stops; Edges = track between stops
Nodes = cities; Edges = rail lines connecting cities
Note: Connections in graph matter, not precise locations of nodes
Internet (~1972)
Internet (~1998)
Word Game
CS Pre-requisite Structure (subset)

Nodes = courses; Edges = prerequisites

Nodes:
- Discrete Math
- Data Structures
- Programming Languages
- Operating Systems
- Graphics
- Theory of comp.
- AI

Edges:
- Java
- Organization
Wire-Frame Models
Def’n: An *undirected graph* \( G = (V,E) \) consists of two sets

- \( V \): the vertices of \( G \), and \( E \): the edges of \( G \)
- Each edge \( e \) in \( E \) is defined by a set of two vertices: its *incident vertices*. We write \( e = \{u,v\} \) and say that \( u \) and \( v \) are *adjacent*. 

---

**Basic Definitions & Concepts**
Walking Along a Graph

- A walk from $u$ to $v$ in a graph $G = (V,E)$ is an alternating sequence of vertices and edges
  $$u = v_0, e_1, v_1, e_2, v_2, \ldots, v_{k-1}, e_k, v_k = v$$
  such that each $e_i = \{v_i, v_{i+1}\}$ for $i = 1, \ldots, k$
- Note a walk starts and ends on a vertex
- If no edge appears more than once then the walk is called a path
- If no vertex appears more than once then the walk is a simple path
Walking In Circles

• A closed walk in a graph G = (V,E) is a walk

   v₀, e₁, v₁, e₂, v₂, ... , vₖ₋₁, eₖ, vₖ

   such that each v₀ = vₖ

• A circuit is a path where v₀ = vₖ
  • No repeated edges

• A cycle is a simple path where v₀ = vₖ
  • No repeated vertices (uhm, except for v₀!)

• The length of any of these is the number of edges in the sequence
Little Tiny Theorems

• If there is a walk from u to v, then there is a walk from v to u.
• If there is a walk from u to v, then there is a path from u to v (and from v to u)
• If there is a path from u to v, then there is a simple path from u to v (and v to u)
• Every circuit through v contains a cycle through v
• Not every closed walk through v contains a cycle through v! [Try to find an example!]
Another Useful Graph Fact

• Degree of a vertex $v$
  • Number of edges incident to $v$
  • Denoted by $\deg(v)$

• Thm: For any graph $G = (V,E)$

$$\sum_{v \in V} \deg(v) = 2 |E|$$

where $|E|$ is the number of edges in $G$

• Proof Hint: Induction on $|E|$: How does removing an edge change the equation?
  • Or: Count pairs $(v,e)$ where $v$ is incident with $e$
Reachability and Connectedness

- Def’n: A vertex $v$ in $G$ is reachable from a vertex $u$ in $G$ if there is a path from $u$ to $v$
- $v$ is reachable from $u$ iff $u$ is reachable from $v$
- Def’n: An undirected graph $G$ is connected if for every pair of vertices $u, v$ in $G$, $v$ is reachable from $u$ (and vice versa)
- The set of all vertices reachable from $v$, along with all edges of $G$ connecting any two of them, is called the connected component of $v$