Trees, Binary Trees and Tries

Dynamic, nonlinear data structure are often useful in pruning information
A tree (a connected graph with no cycles) is a graph theoretical notion
A graph $G = (V, E)$ is a finite collection of vertices $V$ and edges $E$

- Usually $n = ||V||$, and $m = ||E||$

- Digraph: $E$ is a binary relation on $V$. Self loops are allowed. $0 \leq m \leq n^2$ and $\sum_{v \in V} \text{in} - \deg(v) = m$

- Undirected graph, or simply graph: $E$ is a set of unordered pairs of distinct vertices. $0 \leq m \leq \binom{n}{2} \in O(n^2)$ and $\sum_{v \in V} \deg(v) = 2m$

Tries are tree like structure whose shape is dependent on the space in which data lives, rather than the data itself (for example, points in a two dimensional world)
Binary trees are subtly different from trees and are used extensively
An Introduction to Tries

Name comes from retrieval, pronounced “trys”
Can be used to implement a dictionary $D$ on keys which are strings on an alphabet. Indeed

* Size of data structure is $O(s)$ where $s$ is the number of words in the dictionary
* Inserts and deletes take time proportional to the length of the largest string in $D$
* Search time is $O(m)$ where $m$ is the length of the query string

Also support prefix matching: Given a string $X$, find all strings in $D$ that contain $X$ as a prefix
Applications: command completion, supporting (after preprocessing) word matches (find words in an article), Internet routing
Basic computer data structure: Rooted ordered tree
Simple Tries

Let $S$ be a set of $s$ strings from an alphabet $\Sigma$ such that no string in $S$ is a prefix of another string. A simple trie for $S$ is an ordered tree $T$ with the following properties:

1. Each node of $T$ except the root is labeled with a character of $\Sigma$.

2. The ordering of the children of an internal node of $T$ is determined by the canonical ordering of $\Sigma$.

3. $T$ has $s$ external nodes, each associated with a string of $S$ such that the concatenation of the labels of the nodes on the path from the root to an external node $v$ of $T$ yields the appropriate string.
Simple Trie Implementation

Internal nodes have anywhere between 1 and $d$ children. The convention is to have a special marker character ($.$, or $\Lambda$) in the alphabet.

Impl 1: Array allocated at each node  
Impl 2: First-child, next-sibling

Space hungry: If the $s$ strings sum to length $n$, worst case number of nodes is $O(n)$. Incremental construction costs $O(dn)$. Operations costs $O(dm)$
Compressed or Patricia Tries

Main idea: Each internal node has at least two children

In a pessimistic version store string at each node AND position on which the node discriminates (useful for supporting dynamic inserts and deletes)

Alternatively, store the index of the character position on which that node discriminates, not the entire string.

Either way, number of nodes is proportional to the number of strings, not to the sum of the lengths.
Searching Patricia Tries

Search is easy (even if we don’t store the string in the nodes)

- If an internal node has index $i$ and the $i$th character of the search key is the $j$th possible symbol, follow the $j$th pointer from that node

- Upon reaching an external node, compare with the string stored. (This step is necessary; the algorithm optimistically assumes that matches have happened at all internal nodes. stick would be confused with stock)

- If the search key has fewer than $i$ characters, it is not in the trie

prefixQuery($x$) is also implemented easily. Idea: If the search terminates at an internal node $v$, return all external nodes stored in the subtree rooted at $v$

For a random set of input data, the average height is $O(\log_d n)$
The Search Algorithm for Compressed Tries

class TrieNode {
    int position; TrieNode[] pointers; String data;
    // next index to match, pointers based on alphabet
}

TrieNode find (string x, TrieNode v) {
    if (v == null) return v;
    else {
        Boolean fullMatch = match (x, v.data);
        // true if v.data is substring of x
        if (!fullMatch) {
            outputPartial (x, v.data);
            // also available at this point Y1 (Y2),
            // the part of v.data that matched (didn’t)
            // similarly X2 (not matched part of X)
            // We need these for insert!
            return v;
        }
    }
}
Search Algorithm Continued

def find(x, v[x[1]])
{
    if (isExternal(v)) return v;
    // also available (possibly null) X2 rest of X
    else {
        stripX(X, v.position);
        // remove parts of X we don’t care about.
        // Dereference the right pointer
        if (v[x[1]] == null) // nowhere to go
            return v; // remember we have X2 to return
        else find(x, v[x[1]]);
    }
}

The insert algorithm uses the status of the match step, as well as strings X2, Y1, Y2.
Patricia Insert Example

Insert Sequence: stock, sell, bear, bell, bid, buy, stop, bull
The Insert Algorithm

```java
insert (string x, TrieNode r) {
  v = find(x, r);
  if (v == null) ... // special case
  if (fullMatch) {
    if (X2 == null) return;
    w = new TrieNode();
    w.data = X2; w.position = pos(X2, x);
    w.pointers = nil; w.parent = v;
    v.child[X2[1]] = w;
  }
  else {
    u = new TrieNode();
    z = new TrieNode();
    u.data = Y1; u.position = ... z.data = X2; z.position =
    u[Y2[1]] = v;
    u.parent = v.parent;
    z.parent = v.parent;
    v.parent = u;
    v.data = y2;
  }
}
```
Use of Tries and Suffix Tries

Tries are best used as an auxiliary structure on strings (sort of index)

Tries efficiently support *word matches* and *prefix matches*

Pattern matching

- After preprocessing the pattern in time proportional to the pattern length, the Boyer-Moore algorithm searches an arbitrary English text in time proportional to the text length $O(n)$

- Suffix tries use $O(n)$ space performs pattern matching queries in time proportional to the pattern length
Word Match: Web Search Engines

- The index of a search engine (collection of all searchable words) is stored into a compressed trie
- Each leaf of the trie is associated with a word and has a list of pages (URLs) containing that word, called occurrence list
- The trie is kept in internal memory and the occurrence lists are kept in external memory and are ranked by relevance
- Boolean queries for sets of words (e.g., stock and stockings) correspond to set operations (such as intersection) on the occurrence lists
- Additional information retrieval techniques are used, such as
  - stopword elimination (ignore ‘the,’ ‘a,’ ‘is’)
  - stemming (add adding added),
  - link analysis (recognize authoritative pages)
Prefix Match: Internet Routers

- Computers on the Internet (hosts) are identified by a unique 32-bit IP (Internet protocol) address, usually written in dotted-quad-decimal notation (www.cs.umd.edu is 128.8.128.160)

- An organization uses a subset of IP addresses with the same prefix (umd uses 128.8.*, IIT Bombay uses 144.16.*.*

- Data is sent to a host by fragmenting it into packets. Each packet carries the IP address of its destination.

- The Internet is a graph whose nodes are routers, and whose edges are communication links.

- A router forwards packets to its neighbors using IP prefix matching rules (a packet with IP prefix 144.16 should be forwarded to the IIT gateway router)

- Routers use tries on the alphabet 0, 1 to do prefix matching.
Suffix Tries

The *suffix trie* for a text X of size n from an alphabet of size d

- Stores all the $n(n - 1)/2$ suffixes of X in $O(n)$ space
- Supports arbitrary pattern matching and prefix matching queries in $O(dm)$ time
- Can be constructed in $O(dn)$ time