CSCI 104
Tries

Mark Redekopp
David Kempe
TRIES
Review of Set/Map Again

- Recall the operations a set or map performs...
  - Insert(key)
  - Remove(key)
  - find(key) : bool/iterator-pointer [Map only]
  - Get(key) : value [Map only]
- We can implement a set or map using a binary search tree
  - Search = O(_________)
- But what work do we have to do at each node?
  - Compare (i.e. string compare)
  - How much does that cost?
    - Int = O(1)
    - String = O( m ) where m is length of the string
  - Thus, search costs O( ____________ )
Review of Set/Map Again

• Recall the operations a set or map performs...
  – Insert(key)
  – Remove(key)
  – find(key) : bool/iterator/pointer
  – Get(key) : value  [Map only]

• We can implement a set or map using a binary search tree
  – Search = $O(\log(n))$

• But what work do we have to do at each node?
  – Compare (i.e. string compare)
  – How much does that cost?
    • Int = $O(1)$
    • String = $O(m)$ where m is length of the string
  – Thus, search costs $O(m \times \log(n))$
Review of Set/Map Again

• We can implement a set or map using a hash table
  – Search = $O(1)$

• But what work do we have to do once we hash?
  – Compare (i.e. string compare)
  – How much does that cost?
    • Int = $O(1)$
    • String = $O(m)$ where $m$ is length of the string
  – Thus, search costs $O(m)$
Tries

- Assuming unique keys, can we still achieve $O(m)$ search but not have collisions?
  - $O(m)$ means the time to compare is independent of how many keys (i.e. $n$) are being stored and only depends on the length of the key

- Trie(s) (often pronounced "try" or "tries") allow $O(m)$ retrieval
  - Sometimes referred to as a radix tree or prefix tree

- Consider a trie for the keys
  - "HE", "HEAP", "HEAR", "HELP", "ILL", "IN"
Tries

- Rather than each node storing a full key value, each node represents a prefix of the key
- Highlighted nodes indicate terminal locations
  - For a map we could store the associated value of the key at that terminal location
- Notice we "share" paths for keys that have a common prefix
- To search for a key, start at the root consuming one unit (bit, char, etc.) of the key at a time
  - If you end at a terminal node, SUCCESS
  - If you end at a non-terminal node, FAILURE
Tries

• To search for a key, start at the root consuming one unit (bit, char, etc.) of the key at a time
  – If you end at a terminal node, SUCCESS
  – If you end at a non-terminal node, FAILURE

• Examples:
  – Search for "He"
  – Search for "Help"
  – Search for "Head"

• Search takes $O(m)$ where $m = \text{length of key}$
  – Notice this is the same as a hash table

A "value" type could be stored for each non-terminal node
Your Turn

• Construct a trie to store the set of words
  – Ten
  – Tent
  – Then
  – Tense
  – Tens
  – Tenth
Application: IP Lookups

• Network routers form the backbone of the Internet
• Incoming packets contain a destination IP address (128.125.73.60)
• Routers contain a "routing table" mapping some prefix of destination IP address to output port
  – 128.125.x.x => Output port C
  – 128.209.32.x => Output port B
  – 128.x.x.x => Output port D
  – 132.x.x.x => Output port A
• Keys = Match the longest prefix
  – Keys are unique
• Value = Output port

<table>
<thead>
<tr>
<th>Octet 1</th>
<th>Octet 2</th>
<th>Octet 3</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000000</td>
<td>01111101</td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>10000000</td>
<td>11010001</td>
<td>00100000</td>
<td>B</td>
</tr>
<tr>
<td>10000000</td>
<td></td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>10000100</td>
<td></td>
<td></td>
<td>A</td>
</tr>
</tbody>
</table>
IP Lookup Trie

• A binary trie implies that the
  – Left child is for bit '0'
  – Right child is for bit '1'

• Routing Table:
  – 128.125.x.x => Output port C
  – 128.209.32.x => Output port B
  – 128.209.44.x => Output port D
  – 132.x.x.x => Output port A

<table>
<thead>
<tr>
<th>Octet 1</th>
<th>Octet 2</th>
<th>Octet 3</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000000</td>
<td>01111101</td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>10000000</td>
<td>11010001</td>
<td>00100000</td>
<td>B</td>
</tr>
<tr>
<td>10000000</td>
<td></td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>10000100</td>
<td></td>
<td></td>
<td>A</td>
</tr>
</tbody>
</table>
Structure of Trie Nodes

• What do we need to store in each node?
• Depends on how "dense" or "sparse" the tree is?
  • Dense (most characters used) or small size of alphabet of possible key characters
    – Array of child pointers
    – One for each possible character in the alphabet
  • Sparse
    – (Linked) List of children
    – Node needs to store ______

```
template < class V >
struct TrieNode{
  V* value; // NULL if non-terminal
  TrieNode<V>* children[26];
};
```

```
template < class V >
struct TrieNode{
  char key;
  V* value;
  TrieNode<V>* next;
  TrieNode<V>* children;
};
```
Search

- Search consumes one character at a time until
  - The end of the search key
    - If value pointer exists, then the key is present in the map
    - Or no child pointer exists in the TrieNode
- Insert
  - Search until key is consumed but trie path already exists
    - Set v pointer to value
  - Search until trie path is NULL, extend path adding new TrieNodes and then add value at terminal

```cpp
V* search(char* k, TrieNode<V>* node)
{
    while(*k != '\0' && node != NULL){
        node = node->children[*k - 'a'];
        k++;
    }
    if(node){
        return node->v;
    }
}
```

```cpp
void insert(char* k, Value& v)
{
    TrieNode<V>* node = root;
    while(*k != '\0' && node != NULL){
        node = node->children[*k - 'a'];  k++;
    }
    if(node){
        node->v = new Value(v);
    } else {
        // create new nodes in trie
        // to extend path
        // updating root if trie is empty
    }
}
```
SUFFIX TREES (TRIES)
Prefix Trees (Tries) Review

• What problem does a prefix tree solve
  – Lookups of keys (and possible associated values)

• A prefix tree helps us match 1-of-n keys
  – "He"
  – "Help"
  – "Hear"
  – "Heap"
  – "In"
  – "Ill"

• Here is a slightly different problem:
  – Given a large text string, T, can we find certain substrings or answer other queries about patterns in T
  – A suffix tree (trie) can help here
Suffix Trie Slides

Suffix Trie Wrap-Up

• How many nodes can a suffix trie have for text, T, with length $|T|$?
  – $|T|^2$
  – Can we do better?

• Can compress paths without branches into a single node

• Do we need a suffix trie to find substrings or answer certain queries?
  – We could just take a string and search it for a certain query, q
  – But it would be slow => $O(|T|)$ and not $O(|q|)$
What Have We Learned

• [Key Point]: Think about all the data structures we've been learning?
  – There is almost always a trade-off of memory vs. speed
    • i.e. Space vs. time
  – Most data structures just exploit different points on that time-space tradeoff continuum
  – Think about searches in your project...Do we need a map?
  – No, we could just search all items each time a keyword is provided
    • But think how slow that would be
  – So we build a data structure (i.e. a map) that replicates data and takes a lot of memory space...
  – ...so that we can find data faster