String Sets and Symbol Tables

Symbol Table Review

Symbol table.
- Associate a value with a key.
- Search for value given key.
- Balanced trees use $O(\log N)$ key comparisons.
- Hashing uses $O(1)$ probes, but probe proportional to key length.

Q. Are key comparisons necessary? No.
Q. Is time proportional to key length required? No.
Best possible. Examine $O(\log N)$ bits.

This lecture. Specialized symbol table/set for string keys.
- Faster than hashing.
- More flexible than BST.

Tries

Tries. [from retrieval, but pronounced “try”]
- Store characters in internal nodes, not keys.
- Store records in external nodes.
- Use the characters of the key to guide the search.

Ex: sells sea shells by the sea shore

Applications

Applications.
- Spell checkers.
- Auto-complete.
- Data compression. [stay tuned]
- Computational biology.
- Inverted index of Web.
- Routing tables for IP addresses.
- Storing and querying XML documents.
- T9 predictive text input for cell phones.
String Set: Operations

**Operations.**
- `st.add(s)`: insert string `s` into the set.
- `st.contains(s)`: is string `s` in the set?

**Challenge:** As fast as hashing, as flexible as BST.

```java
StringSET set = new StringSET();
while (!StdIn.isEmpty()) {
    String key = StdIn.readString();
    if (!set.contains(key)) {
        set.add(key);
        System.out.println(key);
    }
}
```

Removes duplicates from input stream

String Set: Implementations Cost Summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Typical Case</th>
<th>Dedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Search hit</td>
<td>Insert</td>
</tr>
<tr>
<td>Input *</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>Red-black</td>
<td>L + log N</td>
<td>log N</td>
</tr>
<tr>
<td>Hashing</td>
<td>L</td>
<td>L</td>
</tr>
</tbody>
</table>

**Actor:** 82MB, 11.4M words, 900K distinct.

**Moby:** 12MB, 210K words, 32K distinct.

N = number of strings
L = size of string
C = number of characters in input
R = radix

* only reads in data

**Challenge:** As fast as hashing, as flexible as BST.

**Keys**

- Key = sequence of “digits.”
- DNA: sequence of a, c, g, t.
- IPv6 address: sequence of 128 bits.
- English words: sequence of lowercase letters.
- Protein: sequence of 20 amino acids A, C, ..., Y.
- Credit card number: sequence of 16 decimal digits.
- International words: sequence of UNICODE characters.
- Library call numbers: sequence of letters, numbers, periods.

**This lecture:** key = string over ASCII alphabet.

R-Way Trie: Example

**Ex:** sells sea shells by the sea shore

R = 26
R-Way Trie: Java Implementation

R-way existence trie: a node.
Node: reference to R nodes.

R-way trie. Faster than hashing for small R, but slow and wastes memory for large R.

Challenge. Use less space.

Existence Symbol Table: Implementations Cost Summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Search hit</th>
<th>Insert</th>
<th>Space</th>
<th>Moby</th>
<th>Actors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>0.26</td>
<td>15.1</td>
</tr>
<tr>
<td>Red-Black</td>
<td>L + log N</td>
<td>log N</td>
<td>C</td>
<td>1.40</td>
<td>97.4</td>
</tr>
<tr>
<td>Hashing</td>
<td>L</td>
<td>L</td>
<td>C</td>
<td>0.76</td>
<td>40.6</td>
</tr>
<tr>
<td>R-Way Trie</td>
<td>L</td>
<td>L</td>
<td>RN + C</td>
<td>1.12</td>
<td>Memory</td>
</tr>
</tbody>
</table>

R-way trie. Faster than hashing for small R, but slow and wastes memory for large R.

Challenge. Use less space.
Ternary Search Trie

Ternary search trie. [Bentley-Sedgewick]
- Each node has 3 children:
  - Left (smaller), middle (equal), right (larger).

**Ex:** sells sea shells by the sea shore

**Observation:** Few wasted links!

TST Implementation

**TST String set:** a node.
**Node:** four fields:
- Character d.
- Reference to left TST. [smaller]
- Reference to middle TST. [equal]
- Reference to right TST. [larger]

private class Node {
    char c;
    Node l, m, r;
    boolean end;
}

TST: Java Implementation

```java
public boolean contains(String s) {
    if (s.length() == 0) return false;
    return contains(s, 0, s.length() - 1);
}
```

26-Way Trie vs. TST

**TST.** Collapses empty links in 26-way trie.

26-way trie (1035 null links, not shown)

TST (155 null links)
Hybrid of R-way and TST

- Do R-way or R\(^2\)-way branching at root.
- Each of R\(^2\) root nodes points to a TST.

Q. What about one-letter words?

TST

```java
public void add(String s) {
    root = add(root, s, 0);
}

private Node add(Node x, String s, int i) {
    if (i == s.length()) {
        x.end = true;
        return x;
    }
    else if (i < s.length()) {
        char c = s.charAt(i);
        Node l = add(x.l, s, i+1);
        Node r = add(x.r, s, i+1);
        if (c < x.key) x.l = l;
        else if (c > x.key) x.r = r;
        else x = x;
        return x;
    }
}
```
TST Summary

**Advantages.**
- Linear space.
- Very fast search hits.
- Search misses even faster.
- Adapts gracefully to irregularities in keys.
- Supports even more general symbol table ops.

**Bottom line:** TST more flexible than BST; can be faster than hashing.

especially if lots of search misses

Longest Prefix Match

**Longest prefix match.** Find string in set with longest prefix match.

**Ex:** Search IP database for longest prefix matching destination IP, and route packets accordingly.

```java
public String prefix(String s) {
    int len = prefix(root, s, 0);
    return s.substring(0, len);
}
```

```java
private int prefix(Node x, String s, int i) {
    if (x == null) return 0;
    int len = 0;
    if (x.end) len = i;
    if (i == s.length()) return len;
    char c = s.charAt(i);
    return Math.max(len, prefix(x.next[c], s, i+1));
}
```

R-way Trie: Longest Prefix Match

**Longest prefix match.** Search, returning the length of longest prefix match seen so far.

```java
"128"
"128.112"
"128.112.136"
"128.112.055"
"128.112.055.15"
"128.112.155.11"
"128.112.155.13"
"128.222"
"128.222.136"

prefix("128.112.136.11") = "128.112.136"
prefix("128.166.123.45") = "128"
```
**T9 Texting**

**Goal.** Type text messages on a phone keypad.

**Multi-tap input.** Enter a letter by repeatedly pressing a key until the desired letter appears.

**T9 text input.** "A much faster and more fun way to enter text."
- Find all words that correspond to given sequence of numbers. (sorted by frequency)
- Press 0 to see all completion options.

**Ex:** hello
- Multi-tap: 4 4 3 5 5 5 5 5 6 6 6
- T9: 4 3 5 5 6

---

**Wildcard Match**

**Wildcard match.** Use wildcard . to match any character.

```
coalizer
coberger
codifier
cofaster
cofather

cognizer
coheler
colander
coleader
... compiler
... composer
computer
cowkeeper
```

---

**TST: Wildcard Match**

**Wildcard match.** Use wildcard . to match any character.
- Search as usual if query character is not a period.
- Go down all three branches if query character is a period.

```java
public void wildcard(String s) {
    wildcard(root, s, 0, "");
    for printing matches
    \n    private void wildcard(Node x, String s, int i, String prefix) {
        if (x == null) return;
        char c = s.charAt(i);
        if (c == '.') { | c < x.c) wildcard(x.left, s, i, prefix);
            if (c == '.') | c == x.c) {
                if (i < s.length() - 1)
                    wildcard(x.mid, s, i+1, prefix + x.c);
                else if (x.end)
                    System.out.println(prefix + x.c);
            }
            if (c == '.') | c > x.c) wildcard(x.right, s, i, prefix);
    }
```

---

**TST Symbol Table**

**TST implementation of symbol table ADT.**
- Store key-value pairs in leaves of trie.
- Search hits end at leaf with key-value pair.
- Search misses end at null or leaf with different key.
- Internal node stores char; external node stores key-value pair.
- use separate internal and external nodes?
- collapse (and split) 1-way branches at bottom?
Existence Symbol Table: Implementations Cost Summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Search hit</th>
<th>Insert</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>Red-Black</td>
<td>L + log N</td>
<td>log N</td>
<td>C</td>
</tr>
<tr>
<td>Hashing</td>
<td>L</td>
<td>L</td>
<td>C</td>
</tr>
<tr>
<td>R-Way Trie</td>
<td>L</td>
<td>R N + C</td>
<td></td>
</tr>
<tr>
<td>TST</td>
<td>L + log N</td>
<td>L + log N</td>
<td>C</td>
</tr>
<tr>
<td>TST with R^2</td>
<td>L + log N</td>
<td>L + log N</td>
<td>C</td>
</tr>
<tr>
<td>R-way collapse 1-way</td>
<td>log N</td>
<td>log N</td>
<td>RN + C</td>
</tr>
<tr>
<td>TST collapse 1-way</td>
<td>log N</td>
<td>log N</td>
<td>C</td>
</tr>
</tbody>
</table>

Key property. Search, insert time is independent of key length!
Consequence. Can use with very long keys.

Suffix Tree

**Suffix tree.** Patricia trie of suffixes of a string.

- Longest common substring.
- Longest repeated substring.
- Longest palindromic substring.
- Longest common prefix of two substrings.
- Computational biology databases (BLAST, FASTA).
- Search for music by melody.

PATRICIA Tries

**Patricia tries.** [Practical Algorithm to Retrieve Information Coded in Alphanumeric.]
- Collapse one-way branches in binary trie.
- Thread trie to eliminate multiple node types.

Applications.
- Database search.
- P2P network search.
- IP routing tables: find longest prefix match.
- Compressed quad-tree for N-body simulation.
- Efficiently storing and querying XML documents.

Symbol Table Summary

**Hash tables.** Separate chaining, linear probing.

**Binary search trees.** Randomized, splay, red-black.

**Tries.** R-way, TST.

Lessons.
- Determine the needed ST ops for your application, and choose the best data structure.
- You can get at anything (if organized properly) in 40 or 100 bits!