Review: summary of the performance of symbol-table implementations

Order of growth of the frequency of operations.

<table>
<thead>
<tr>
<th>implementation</th>
<th>typical case</th>
<th>ordered operations on keys</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>red-black BST</td>
<td>log N</td>
<td>log N</td>
<td>yes</td>
</tr>
<tr>
<td>hash table</td>
<td>1</td>
<td>1</td>
<td>no</td>
</tr>
</tbody>
</table>

• † under uniform hashing assumption

Q. Can we do better?
A. Yes, if we can avoid examining the entire key, as with string sorting.

String symbol table. Symbol table specialized to string keys.

Goal. Faster than hashing, more flexible than BSTs.
String symbol table implementations cost summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Character accesses (typical case)</th>
<th>dedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search hit</td>
<td>search miss</td>
</tr>
<tr>
<td>red-black BST</td>
<td>L + c lg</td>
<td>c lg</td>
</tr>
<tr>
<td>hashing (linear probing)</td>
<td>L</td>
<td>L</td>
</tr>
</tbody>
</table>

Parameters
• N = number of strings
• L = length of string
• R = radix

Challenge. Efficient performance for string keys.

Tries

Tries. [from retrieval, but pronounced "try"]
• Store characters in nodes (not keys).
• Each node has R children, one for each possible character.
• Store values in nodes corresponding to last characters in keys.

Follow links corresponding to each character in the key.
• Search hit: node where search ends has a non-null value.
• Search miss: reach a null link or node where search ends has null value.

Search in a trie

Get("shells")

Value for she in node corresponding to last key character

Return value associated with last key character (return 3)
Search in a trie

Follow links corresponding to each character in the key.
- **Search hit**: node where search ends has a non-null value.
- **Search miss**: reach a null link or node where search ends has null value.

```
get("she")
```

![Search in a trie](image1.png)

Search in a trie

Follow links corresponding to each character in the key.
- **Search hit**: node where search ends has a non-null value.
- **Search miss**: reach a null link or node where search ends has null value.

```
get("shell")
```

![Search in a trie](image2.png)

Search in a trie

Follow links corresponding to each character in the key.
- **Search hit**: node where search ends has a non-null value.
- **Search miss**: reach a null link or node where search ends has null value.

```
get("shelter")
```

![Search in a trie](image3.png)

Insertion into a trie

Follow links corresponding to each character in the key.
- **Encounter a null link**: create new node.
- **Encounter the last character of the key**: set value in that node.

```
put("shore", 7)
```

![Insertion into a trie](image4.png)
Trie construction demo

trie

Trie construction demo

put("she", 0)

key is sequence of characters from root to value
value is in node corresponding to last character

Trie construction demo

trie

Trie construction demo

trie

trie
Trie construction demo

put("sells", 1)

Trie construction demo

trie

Trie construction demo

trie

Trie construction demo

put("sea", 2)
Trie construction demo

```
trie
```

```
put("shells", 3)
```

```
trie
```

```
put("by", 4)
```
Trie construction demo

Trie construction demo

Trie construction demo

Trie construction demo

put("the", 5)

put("sea", 6)

case "sea":
    override old value with new value

Trie construction demo

put("shore", 7)

Trie construction demo
Trie representation: Java implementation

Node. A value, plus references to $R$ nodes.

```java
private static class Node {
    private Object value;
    private Node[] next = new Node[R];
}
```

Trie representation:

- Each node has an array of links and a value.
- Characters are implicitly defined by link index.
- Neither keys nor characters are explicitly stored.
- Use `Object` instead of `Value` since no generic array creation in Java.

R-way trie: Java implementation

```java
public class TrieST<Value> {
    private static final int R = 256;
    private Node root;

    private static class Node {
        /* see previous slide */
    }

    public void put(String key, Value val) {
        root = put(root, key, val, 0);
    }

    private Node put(Node x, String key, Value val, int d) {
        if (x == null) x = new Node();
        if (d == key.length()) { x.val = val; return x; }
        char c = key.charAt(d);
        x.next[c] = put(x.next[c], key, val, d+1);
        return x;
    }

    // ... (omitted)

    public boolean contains(String key) {
        return get(key) != null;
    }

    public Value get(String key) {
        Node x = get(root, key, 0);
        if (x == null) return null;
        return (Value) x.val;
    }

    private Node get(Node x, String key, int d) {
        if (x == null) return null;
        if (d == key.length()) return x;
        char c = key.charAt(d);
        return get(x.next[c], key, d+1);
    }
}
```

Trie performance

- **Search hit.** Need to examine all $L$ characters for equality.
- **Search miss.**
  - Could have mismatch on first character.
  - Typical case: examine only a few characters (sublinear).
- **Space.** $R$ null links at each leaf.
  (but sublinear space possible if many short strings share common prefixes)
- **Bottom line.** Fast search hit and even faster search miss, but wastes space.
Deletion in an R-way trie

To delete a key-value pair:
- Find the node corresponding to key and set value to null.
- If that node has all null links, remove that node (and recur).

```
delete("shells")
```

String symbol table implementations cost summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>search hit</th>
<th>search miss</th>
<th>insert</th>
<th>space (references)</th>
<th>moby.txt</th>
<th>actors.txt</th>
</tr>
</thead>
<tbody>
<tr>
<td>red-black BST</td>
<td>L + c lg</td>
<td>c lg</td>
<td>c lg</td>
<td>4N</td>
<td>1,4</td>
<td>97,4</td>
</tr>
<tr>
<td>hashing (linear probing)</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>4N to 16N</td>
<td>0.76</td>
<td>40.6</td>
</tr>
<tr>
<td>R-way trie</td>
<td>L</td>
<td>log</td>
<td>L</td>
<td>(R+1) N</td>
<td>1,12</td>
<td>out of memory</td>
</tr>
</tbody>
</table>

R-way trie.
- Method of choice for small $R$.
- Too much memory for large $R$.

Challenge. Use less memory, e.g., 65,536-way trie for Unicode!

Digression: out of memory?

- “640 K ought to be enough for anybody.”
  — (mis)attributed to Bill Gates, 1981
  (commenting on the amount of RAM in personal computers)

- “64 MB of RAM may limit performance of some Windows XP features; therefore, 128 MB or higher is recommended for best performance.”
  — Windows XP manual, 2002

- “64 bit is coming to desktops, there is no doubt about that. But apart from Photoshop, I can’t think of desktop applications where you would need more than 4GB of physical memory, which is what you have to have in order to benefit from this technology. Right now, it is costly.”
  — Bill Gates, 2003

- “512-bit words ought to be enough for anybody.”
  — Kevin Wayne, 1995
Ternary search tries

- Store characters and values in nodes (not keys).
- Each node has three children: smaller (left), equal (middle), larger (right).

Ternary search tries

- Store characters and values in nodes (not keys).
- Each node has three children: smaller (left), equal (middle), larger (right).
Search in a TST

get("sea")

return value associated with last key character

Ternary search trie insertion demo

ternary search trie

put("she", 0)

key is sequence of characters from root to value using middle links
value is in node corresponding to last character

Search in a TST

get("shelter")

no link to 't' (return null)
Ternary search trie insertion demo

```
put("she", 0)
```

Ternary search trie insertion demo

```
put("sells", 1)
```

Ternary search trie insertion demo

```
ternary search trie
```

Ternary search trie insertion demo

```
put("sea", 2)
```
Ternary search trie insertion demo

ternary search trie

put("shells", 3)

Ternary search trie insertion demo

Ternary search trie insertion demo

put("by", 4)
Ternary search trie insertion demo

Ternary search trie insertion demo

Ternary search trie insertion demo

Ternary search trie insertion demo

put("the", 5)

put("sea", 6)
Ternary search trie insertion demo

ternary search trie

put("shore", 7)

Ternary search trie insertion demo

ternary search trie

Ternary search trie insertion demo

ternary search trie

Ternary search trie insertion demo

ternary search trie
26-way trie vs. TST

26-way trie. 26 null links in each leaf.

TST. 3 null links in each leaf.

TST: Java implementation

```java
public class TST<Value> {
    private Node root;

    private class Node {
        // see previous slide
    }

    public void put(String key, Value val) {
        root = put(root, key, val, 0);
    }

    private Node put(Node x, String key, Value val, int d)
    {
        char c = key.charAt(d);
        if (x == null) {
            x = new Node();
            x.c = c;
        }
        if      (c < x.c)              x.left  = put(x.left,  key, val, d);
        else if (c > x.c)              x.right = put(x.right, key, val, d);
        else if (d < key.length() - 1) x.mid   = put(x.mid,   key, val, d+1);
        else                           x.val   = val;
        return x;
    }

    public boolean contains(String key) {
        return get(key) != null;
    }

    public Value get(String key) {
        Node x = get(root, key, 0);
        if (x == null) return null;
        return x.val;
    }

    private Node get(Node x, String key, int d)
    {
        if (x == null) return null;
        char c = key.charAt(d);
        if      (c < x.c)              return get(x.left,  key, d);
        else if (c > x.c)              return get(x.right, key, d);
        else if (d < key.length() - 1) return get(x.mid,   key, d+1);
        else                           return x;
    }
}
```

TST representation in Java

A TST node is five fields:
- A value.
- A character c.
- A reference to a left TST.
- A reference to a middle TST.
- A reference to a right TST.

```java
private class Node {
    private Value val;
    private char c;
    private Node left, mid, right;
}
```

TST: Java implementation (continued)

```java
public boolean contains(String key) {
    return get(key) != null;
}
```

```java
public Value get(String key) {
    Node x = get(root, key, 0);
    if (x == null) return null;
    return x.val;
}
```

```java
private Node get(Node x, String key, int d)
{
    if (x == null) return null;
    char c = key.charAt(d);
    if      (c < x.c) return get(x.left,  key, d);
    else if (c > x.c) return get(x.right, key, d);
    else if (d < key.length() - 1) return get(x.mid,   key, d+1);
    else return x;
}
String symbol table implementation cost summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Character accesses (typical case)</th>
<th>Dedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search hit</td>
<td>search miss</td>
</tr>
<tr>
<td>red-black BST</td>
<td>(L + c \log N)</td>
<td>(c \log N)</td>
</tr>
<tr>
<td>hashing (linear probing)</td>
<td>(L)</td>
<td>(L)</td>
</tr>
<tr>
<td>R-way trie</td>
<td>(L)</td>
<td>(\log (R+1)N)</td>
</tr>
<tr>
<td>TST</td>
<td>(L + \ln N)</td>
<td>(\ln N)</td>
</tr>
</tbody>
</table>

Remark. Can build balanced TSTs via rotations to achieve \(L + \log N\) worst-case guarantees.

Bottom line. TST is as fast as hashing (for string keys), space efficient.

TST vs. hashing

Hashing.
- Need to examine entire key.
- Search hits and misses cost about the same.
- Performance relies on hash function.
- Does not support ordered symbol table operations.

TSTs.
- Works only for strings (or digital keys).
- Only examines just enough key characters.
- Search miss may involve only a few characters.
- Supports ordered symbol table operations (plus others!).

Bottom line. TSTs are:
- Faster than hashing (especially for search misses).
- More flexible than red-black BSTs. [stay tuned]

Previously on BBM202...

Tries.
- Store characters in nodes (not keys).
- Each node has \(R\) children, one for each possible character.
- Store values in nodes corresponding to last characters in keys.

Ternary Search Trees (TSTs)
- Store characters and values in nodes (not keys).
- Each node has three children: smaller (left), equal (middle), larger (right).

Tries

- R-way tries
- Ternary search tries
- Character-based operations
Character-based operations. The string symbol table API supports several useful character-based operations.

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>by</td>
<td>4</td>
</tr>
<tr>
<td>sea</td>
<td>6</td>
</tr>
<tr>
<td>sells</td>
<td>1</td>
</tr>
<tr>
<td>she</td>
<td>0</td>
</tr>
<tr>
<td>shells</td>
<td>3</td>
</tr>
<tr>
<td>shore</td>
<td>7</td>
</tr>
<tr>
<td>the</td>
<td>5</td>
</tr>
</tbody>
</table>

Prefix match. Keys with prefix "sh": "she", "shells", and "shore".

Wildcard match. Keys that match ".he": "she" and "the".

Longest prefix. Key that is the longest prefix of "shellsort": "shells".

Warmup: ordered iteration

To iterate through all keys in sorted order:
- Do inorder traversal of trie; add keys encountered to a queue.
- Maintain sequence of characters on path from root to node.

Ordered iteration: Java implementation

To iterate through all keys in sorted order:
- Do inorder traversal of trie; add keys encountered to a queue.
- Maintain sequence of characters on path from root to node.

```java
public class StringST<Value>

StringST() create a symbol table with string keys
void put(String key, Value val) put key-value pair into the symbol table
Value get(String key) value paired with key
void delete(String key) delete key and corresponding value

Iterable<String> keys() all keys
Iterable<String> keysWithPrefix(String s) keys having s as a prefix
Iterable<String> keysThatMatch(String s) keys that match s (where . is a wildcard)
String longestPrefixOf(String s) longest key that is a prefix of s

public Iterable<String> keys()
{
    Queue<String> queue = new Queue<String>();
    collect(root, "", queue);
    return queue;
}

private void collect(Node x, String prefix, Queue<String> q)
{
    if (x == null) return;
    if (x.val != null) q.enqueue(prefix);
    for (char c = 0; c < R; c++)
    {
        collect(x.next[c], prefix + c, q);
    }
}
```
Prefix matches

Find all keys in symbol table starting with a given prefix.

Ex. Autocomplete in a cell phone, search bar, text editor, or shell.

- User types characters one at a time.
- System reports all matching strings.

Wildcard matches

Use wildcard . to match any character in alphabet.

codex...  c...c.
collizer acresce
coberger acroach
codifier acracy
cofaster octarch
cofather science
cognizer scratch
cohelper scratch
colander scratch
coleader screich
... 
srinch
... 
compiler scrich
... 
scrunch
composer scudick
computer scutock
cowkeeper

Prefix matches

Find all keys in symbol table starting with a given prefix.

public Iterable<String> keysWithPrefix(String prefix) {
    Queue<String> queue = new Queue<String>();
    Node x = get(root, prefix, 0);
    collect(x, prefix, queue);
    return queue;
}

Wildcard matches

Search as usual if character is not a period; go down all R branches if query character is a period.

private void collect(Node x, String prefix, String pat, Queue<String> q) {
    if (x == null) return;
    int d = prefix.length();
    if (d == pat.length() && x.val != null) q.enqueue(prefix);
    if (d == pat.length()) return;
    char next = pat.charAt(d);
    for (char c = 0; c < R; c++)
        if (next == '.' || next == c)
            collect(x.next[c], prefix + c, pat, q);
}
Longest prefix

Find longest key in symbol table that is a prefix of query string.

Ex. To send packet toward destination IP address, router chooses IP address in routing table that is longest prefix match.

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

Note. Not the same as floor:  

```
represented as 32-bit binary number for IPv4  
(instead of string)
```

```
floor("128.112.136.11") = "128.112.136"
longestPrefixOf("128.112.136.11") = "128.112.136"
```

```
longestPrefixOf("128.112.100.16") = "128.112"
```

```
longestPrefixOf("128.166.123.45") = "128"
```

Longest prefix: Java implementation

Find longest key in symbol table that is a prefix of query string.
• Search for query string.
• Keep track of longest key encountered.

```
public String longestPrefixOf(String query) {
    int length = search(root, query, 0, 0);
    return query.substring(0, length);
}
```

```
private int search(Node x, String query, int d, int length) {
    if (x == null) return length;
    if (x.val != null) length = d;
    if (d == query.length()) return length;
    char c = query.charAt(d);
    return search(x.next[c], query, d+1, length);
}
```

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.

```
"she"
"shell"
"shellsort"
```

```
search ends at end of string  
value is not null  
return she  
(last key on path)
```

```
search ends at null link  
return shells  
(last key on path)
```

```
search ends at null link  
return she  
(last key on path)
```

Possibilities for longestPrefixOf()

```
"shellers"
```

Ex. hello
• Multi-tap: 4 4 3 3 5 5 5 5 6 6 6
• T9: 4 3 5 5 6

Goal. Type text messages on a phone keypad.

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

```
how
```

"a much faster and more fun way to enter text"
A world without “s” ??

To: "Kevin Wayne" <wayne@CS.Princeton.EDU>
Date: Tue, 25 Oct 2005 12:44:42 -0700

Thank you Kevin.

I am glad that you find T9 o valuable for your cla.  I had not noticed thi before.  Thank for writing in and letting u know.

Take care,
Brooke nyder
OEM Dev upport
AOL/Tegic Communication
1000 Dexter Ave N. suite 300
eattle, WA 98109

ALL INFORMATION CONTAINED IN THIS EMAIL IS CONSIDERED CONFIDENTIAL AND PROPERTY OF AOL/TEGIC COMMUNICATIONS