5.2 TRIES

- R-way tries
- ternary search tries
- character-based operations

Summary of the performance of symbol-table implementations

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Typical Case</th>
<th>Ordered Operations</th>
<th>Operations on Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>red-black BST</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
</tr>
<tr>
<td>hash table</td>
<td>1 †</td>
<td>1 †</td>
<td>1 †</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 basic API

String symbol table. Symbol table specialized to string keys.

public class StringST<Value>

StringST() create an empty symbol table
void put(String key, Value val) put key-value pair into the symbol table
Value get(String key) return value paired with given key
void delete(String key) delete key and corresponding value

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^2 N</td>
<td>c lg^2 N</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

Goal. Efficient performance for string keys.
“More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason—including blind stupidity.”  — William A. Wulf

Tries

Tries.  [from retrieval, but pronounced “try”]
- Store characters in nodes (not keys).
- Each node has \( R \) children, one for each possible character.
  (for now, we do not draw null links)
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 null link or node where search ends has null value.

```plaintext
get("shells")
```

![Diagram of search for "shells"]

- return value in node corresponding to last character in key (return 3)

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

![Diagram of search for "she"]

- search may terminated at an intermediate node (return 0)

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 null link or node where search ends has null value.

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

![Diagram of search for "shell"]

- no value associated node corresponding to last character in key (return null)

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

![Diagram of search for "shelter"]

- no link to t (return null)
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)
```

Trie construction demo

Trie representation: Java implementation

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

```java
private static class Node {
    private Object val; // no generic array creation
    private Node[] next = new Node[R];
}
```

Remark. Neither keys nor characters are stored explicitly.

R-way trie: Java implementation

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

    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;
    }
}
```
**R-way trie: Java implementation (continued)**

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

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 quiz 1**

What is order of growth of the running time (in the worst case) to insert a key into an R-way trie, as a function of the number of strings \( N \), the length \( L \) of the key, and the alphabet size \( R \)?

A. \( L \)
B. \( R + L \)
C. \( N + L \)
D. \( R \times L \)
E. I don't know.

---

**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 strings share long 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 node has null value and all null links, remove that node (and recur).

`delete("shells")`

![Trie representation](image)
Deletion in an R-way trie

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

\[ \text{delete("shells")} \]

null value and links (delete node)

---

5.2 TRIES

- R-way tries
- ternary search tries
- character-based operations

---

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^2 N )</td>
<td>( c \lg^2 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 ( aN )</td>
</tr>
</tbody>
</table>

R-way trie.
- Method of choice for small \( R \).
- Works well for medium \( R \).
- Too much memory for large \( R \).

Challenge. Use less memory, e.g., a 216-way trie for Unicode!

Terinary search tries

- Store characters and values in nodes (not keys).
- Each node has 3 children: smaller (left), equal (middle), larger (right).
Ternary search tries

- Store characters and values in nodes (not keys).
- Each node has 3 children: smaller (left), equal (middle), larger (right).

**Trie quiz 2**

Which value is associated with the string key CAC?

A. 3  
B. 4  
C. 5  
D. null  
E. I don’t know.

**Search hit in a TST**

get("sea")

return value in node corresponding to last character in key

**Search miss in a TST**

get("shelter")

no link to t (return null)
Search in a TST

Follow links corresponding to each character in the key.
• If less, take left link; if greater, take right link.
• If equal, take the middle link and move to the next key character.

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.

Ternary search trie construction demo

match: take middle link, move to next char
mismatch: take left or right link, do not move to next char
return value associated with last key character

Trie quiz e

In which subtrie would the key CAC be inserted?

A.  
B.  
C.  
D. None of the above.  
E. I don't know.

26-way trie vs. TST

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

TST. 3 null links in each leaf.
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^2 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 ) ( 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: Java implementation

```java
public class TST<Value, Node extends Node>
{
    private Node root;
    private class Node
    {
        private Value val;
        private char c;
        private Node left, mid, right;
    }

    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);
        return x;
    }

    public void put(String Key, Value val)
    {
        // similar, see book or booksite /*
    }
}
```

Q. What about one- and two-letter words?
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 lg^2 N</td>
<td>c lg^2 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_2 N</td>
</tr>
<tr>
<td>TST</td>
<td>L + ln N</td>
<td>ln N</td>
</tr>
<tr>
<td>TST with R^2</td>
<td>L + ln N</td>
<td>ln N</td>
</tr>
</tbody>
</table>

**Bottom line.** Faster than hashing for our benchmark client.

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 string (or digital) keys.
- Search miss may involve only a few characters.
- Supports ordered symbol table operations (plus extras!).

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

String symbol table API

**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.
String symbol table API

**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

**Remark.** Can also add other ordered ST methods, e.g., floor() and rank().

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

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

Prefix matches

Find all keys in a 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.
Prefix matches in an R-way trie

Find all keys in a 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;
}
```

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"
"128.112.136.11"
"128.112.136.45"
```

longestPrefixOf("128.112.136.11") = "128.112.136"
longestPrefixOf("128.112.100.16") = "128.112"
longestPrefixOf("128.166.123.45") = "128"

Note. Not the same as floor: floor("128.112.100.16") = "128.112.055.15"

Longest prefix in an R-way trie

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);
}

```

Possibilities for longestPrefixOf()
T9 texting (predictive texting)

Goal. Type text messages on a phone keypad.

Multi-tap input. Enter a letter by repeatedly pressing a key.
Ex. good: 4 6 6 6 6 6 3

T9 text input.
- Find all words that correspond to given sequence of numbers.
  4663: good, home, gone, hoof.
- Press * to select next option.
- Press 0 to see all completion options.
- System adapts to user's tendencies.

"a much faster and more fun way to enter text"

Patricia trie

Predictive Text Error Leads to Fatal Stabbing

ANDY CHALK | 10FEBRUARY 2011 9:36 PM
A U.K. man has been convicted of manslaughter for stabbing his friend to death after a predictive text error ballooned into an argument and ultimately a vicious knife attack.

33-year-old Neil Brook and his neighbor, 27-year-old Josef Wilkowski, had known each other for about six months before getting into a beef over a text message misunderstanding. Brook told police he'd sent Wilkowski a text message asking "What are you on about mutter?", "mutter" being "a local colloquialism for a person who behaves in an antisocial or vulgar manner." But thanks to the predictive text on his phone, "mutter" was corrected to "nutter," a slang term meaning "deranged."


Caveat

Patricia trie. [Practical Algorithm to Retrieve Information Coded in Alphanumeric]
- Remove one-way branching.
- Each node represents a sequence of characters.
- Implementation: one step beyond this course.

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.

Also known as: crit-bit tree, radix tree.

http://www.t9.com
### Suffix tree

**Suffix tree.**
- Patricia trie of suffixes of a string.
- Linear-time construction: well beyond scope of this course.

---

**Suffix tree for BANANAS**

---

**Applications.**
- Linear-time: longest repeated substring, longest common substring, longest palindromic substring, substring search, tandem repeats, ....
- Computational biology databases (BLAST, FASTA).

---

### String symbol tables summary

**A success story in algorithm design and analysis.**

**Red-black BST.**
- Performance guarantee: $\log N$ key compares.
- Supports ordered symbol table API.

**Hash tables.**
- Performance guarantee: constant number of probes.
- Requires good hash function for key type.

**Tries.** R-way, TST.
- Performance guarantee: $\log N$ characters accessed.
- Supports character-based operations.