5.2 Tries

- R-way tries
- ternary search tries
- character-based operations
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</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
</tr>
<tr>
<td>red-black BST</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hash table</td>
<td>1 †</td>
<td>1 †</td>
<td>1 †</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></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()

void put(String key, Value val)

Value get(String key)

void delete(String key)

⋮

create an empty symbol table

put key-value pair into the symbol table

return value paired with given 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

<table>
<thead>
<tr>
<th>file</th>
<th>size</th>
<th>words</th>
<th>distinct</th>
</tr>
</thead>
<tbody>
<tr>
<td>moby.txt</td>
<td>1.2 MB</td>
<td>210 K</td>
<td>32 K</td>
</tr>
<tr>
<td>actors.txt</td>
<td>82 MB</td>
<td>11.4 M</td>
<td>900 K</td>
</tr>
</tbody>
</table>

**Challenge.** Efficient performance for string keys.
R-way tries
- ternary search tries
- character-based operations
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.

<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>
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("shells")

![Trie diagram](image)

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 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 a null link or node where search ends has null value.

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

![Trie diagram](image)
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")

![Trie diagram]

- 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.

\text{put}("\text{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.
  - Each node has an array of links and a value.
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;
    }
    ...
}
```
public boolean contains(String key)
{  return get(key) != null;  }

public Value get(String key)
{  ...
      if (d == key.length()) return x;
      char c = key.charAt(d);
      return get(x.next[c], key, d+1);
   }
}

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.
**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></td>
</tr>
<tr>
<td></td>
<td>search miss</td>
<td></td>
</tr>
<tr>
<td></td>
<td>insert</td>
<td></td>
</tr>
<tr>
<td></td>
<td>space (references)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>moby.txt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>actors.txt</td>
<td></td>
</tr>
<tr>
<td>red-black BST</td>
<td>(L + c \log^2 N)</td>
<td>1.40</td>
</tr>
<tr>
<td></td>
<td>(c \log^2 N)</td>
<td>97.4</td>
</tr>
<tr>
<td>hashing (linear probing)</td>
<td>(L)</td>
<td>0.76</td>
</tr>
<tr>
<td></td>
<td>(L)</td>
<td>40.6</td>
</tr>
<tr>
<td>R-way trie</td>
<td>(L)</td>
<td>1.12</td>
</tr>
<tr>
<td></td>
<td>log(_R) N</td>
<td>out of memory</td>
</tr>
<tr>
<td></td>
<td>(L)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>((R+1) N)</td>
<td></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
**Digression: out of memory?**

**A short (approximate) history.**

<table>
<thead>
<tr>
<th>machine</th>
<th>year</th>
<th>address bits</th>
<th>addressable memory</th>
<th>typical actual memory</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDP-8</td>
<td>1960s</td>
<td>12</td>
<td>6 KB</td>
<td>6 KB</td>
<td>$16K</td>
</tr>
<tr>
<td>PDP-10</td>
<td>1970s</td>
<td>18</td>
<td>256 KB</td>
<td>256 KB</td>
<td>$1M</td>
</tr>
<tr>
<td>IBM S/360</td>
<td>1970s</td>
<td>24</td>
<td>4 MB</td>
<td>512 KB</td>
<td>$1M</td>
</tr>
<tr>
<td>VAX</td>
<td>1980s</td>
<td>32</td>
<td>4 GB</td>
<td>1 MB</td>
<td>$1M</td>
</tr>
<tr>
<td>Pentium</td>
<td>1990s</td>
<td>32</td>
<td>4 GB</td>
<td>1 GB</td>
<td>$1K</td>
</tr>
<tr>
<td>Xeon</td>
<td>2000s</td>
<td>64</td>
<td>enough</td>
<td>4 GB</td>
<td>$100</td>
</tr>
<tr>
<td>??</td>
<td>future</td>
<td>128+</td>
<td>enough</td>
<td>enough</td>
<td>$1</td>
</tr>
</tbody>
</table>

“512-bit words ought to be enough for anybody.”

— Kevin Wayne, 1995
A modest proposal

Number of atoms in the universe (estimated). \( \leq 2^{266} \).

Age of universe (estimated). 14 billion years \( \sim 2^{59} \) seconds \( \leq 2^{89} \) nanoseconds.

Q. How many bits address every atom that ever existed?
A. Use a unique 512-bit address for every atom at every time quantum.

<table>
<thead>
<tr>
<th>266 bits</th>
<th>89 bits</th>
<th>157 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>atom</td>
<td>time</td>
<td>cushion for whatever</td>
</tr>
</tbody>
</table>

Ex. Use 256-way trie to map each atom to location.
• Represent atom as 64 8-bit chars (512 bits).
• 256-way trie wastes 255/256 actual memory.
• Need better use of memory.
- R-way tries
- ternary search tries
- character-based operations
Ternary search tries

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

**Abstract**

We present theoretical algorithms for sorting and searching multikey data, and derive from them practical C implementations for applications in which keys are character strings. The sorting algorithm blends Quicksort and radix sort, it is competitive with the best known C sort codes. The searching algorithm blends tries and binary search trees, it is faster than hashing and other commonly used search methods. The basic ideas behind the algorithms is competitive with the most efficient string sorting programs known. The second program is a symbol table implementation that is faster than hashing, which is commonly regarded as the fastest symbol table implementation. The symbol table implementation is much more space-efficient than multiway trees, and supports more advanced searches.

In many application programs, sorts use a Quicksort implementation based on an abstract compare operation,
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

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.

get("sea")

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
Search in a TST

get("sea")

return value associated with last key character
Search in a TST

get("shelter")

no link to 't' (return null)
TST construction demo
26-way trie vs. TST

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

TST. 3 null links in each leaf.
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

```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;
}
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_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 with $R^2$ branching at root

Hybrid of R-way trie and TST.

- Do $R^2$-way branching at root.
- Each of $R^2$ root nodes points to a TST.

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 R 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>
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]
- R-way tries
- ternary search tries
- character-based operations
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**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>StringST()</code></td>
<td>create a symbol table with string keys</td>
</tr>
<tr>
<td><code>void put(String key, Value val)</code></td>
<td>put key-value pair into the symbol table</td>
</tr>
<tr>
<td><code>Value get(String key)</code></td>
<td>value paired with key</td>
</tr>
<tr>
<td><code>void delete(String key)</code></td>
<td>delete key and corresponding value</td>
</tr>
<tr>
<td><code>Iterable&lt;String&gt; keys()</code></td>
<td>all keys</td>
</tr>
<tr>
<td><code>Iterable&lt;String&gt; keysWithPrefix(String s)</code></td>
<td>keys having s as a prefix</td>
</tr>
<tr>
<td><code>Iterable&lt;String&gt; keysThatMatch(String s)</code></td>
<td>keys that match s (where . is a wildcard)</td>
</tr>
<tr>
<td><code>String longestPrefixOf(String s)</code></td>
<td>longest key that is a prefix of s</td>
</tr>
</tbody>
</table>

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

```plaintext
delete("shells")
```
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.

```python
keysWithPrefix(""");

key  q
  b
  by
  s
  sea
  sel
  sell
  sells
  sh
  she
  shell
  shells
  sho
  shor
  shore
  t
  th
  the
```
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> 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.
Find all keys in symbol table starting with a given prefix.

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

Prefix match in a trie

keysWithPrefix("sh");
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"
```

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

```
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

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

Possibilities for longestPrefixOf()
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.

```java
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.

**T9 text input.**
- Find all words that correspond to given sequence of numbers.
- Press 0 to see all completion options.

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

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

www.t9.com
Patricia trie

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.
Suffix tree

Suffix tree.
• Patricia trie of suffixes of a string.
• Linear-time construction: one steps beyond this lecture.

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.

**Bottom line.** You can get at anything by examining 50-100 bits (!!!)