5.2 Tries

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

```java
public class StringST<Value>

    StringST()
    put(String key, Value val)
    get(String key)
    delete(String key)

```

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StringST()</td>
<td>create an empty symbol table</td>
</tr>
<tr>
<td>void put(String key, Value val)</td>
<td>put key-value pair into the symbol table</td>
</tr>
<tr>
<td>Value get(String key)</td>
<td>return value paired with given key</td>
</tr>
<tr>
<td>void delete(String key)</td>
<td>delete key and corresponding value</td>
</tr>
</tbody>
</table>

**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>$L + c \log^2 N$</td>
<td>$c \log^2 N$</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></th>
<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>
<td></td>
</tr>
<tr>
<td>actors.txt</td>
<td>82 MB</td>
<td>11.4 M</td>
<td>900 K</td>
<td></td>
</tr>
</tbody>
</table>

### Challenge
Efficient performance for string keys.
5.2 Tries

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

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

get("shells")

```
return value associated with last key character (return 3)
```

```
node where search ends has a non-null value.
```

```
reach null link or node where search ends has null value.
```
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.

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

![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 null link or node where search ends has null value.

```get("shell")```
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("shelter")
```
**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
Trie construction demo

trie

```
  b
  |   
  v   v
 y 4  e
  |   |   
 a 6  l  h
  |   |   |   |
 s 1  l  e  o
  |   |   |   |
 i 0  l  r
  |   |   |   |
 s 7  e
  |   |
 s 3
```
Node. A value, plus references to $R$ nodes.

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

- 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.
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;
    }

    ;
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;  // 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 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.
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")
```
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).

```plaintext
delete("shells")
```
## 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><strong>red-black BST</strong></td>
<td>$L + c\log^2 N$</td>
<td>$c\log^2 N$</td>
</tr>
<tr>
<td><strong>hashing (linear probing)</strong></td>
<td>$L$</td>
<td>$L$</td>
</tr>
<tr>
<td><strong>R-way trie</strong></td>
<td>$L$</td>
<td>$\log R N$</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!
5.2 Tries

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

- Store characters and values in nodes (not keys).
- Each node has 3 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 are

Fast Algorithms for Sorting and Searching Strings

Jon L. Bentley* Robert Sedgewick#
Ternary search tries

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

TST representation of a trie
get("sea")

return value associated with last key character
Search miss in a TST

get("shelter")

no link to t (return null)
Ternary search trie construction demo

ternary search trie
Ternary search trie construction demo

ternary search trie

```
  s
 /   \
|     |
b     t
 /     /
|     |
y     h
 /     /
|     |
e     e
 /     /
|     |
  e   h
 /   /  \
ll  /  /
|   |   |
ll  l   r
 / /   /
 | |   |
ss  e   ee

        4
        6
        1
        3
        0
        7
        5
```
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.
26-way trie vs. TST

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

TST. 3 null links in each leaf.
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;
}
```

Trie node representations

- Standard array of links ($R = 26$)
- Ternary search tree (TST)
- Link for keys that start with $s$
- Link for keys that start with $su$
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></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><strong>red-black BST</strong></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><strong>hashing</strong></td>
<td>$L$</td>
<td>0.76</td>
</tr>
<tr>
<td>(linear probing)</td>
<td>$L$</td>
<td>40.6</td>
</tr>
<tr>
<td><strong>R-way trie</strong></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><strong>TST</strong></td>
<td>$L + \ln N$</td>
<td>0.72</td>
</tr>
<tr>
<td></td>
<td>$\ln N$</td>
<td>38.7</td>
</tr>
<tr>
<td></td>
<td>$L + \ln N$</td>
<td></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></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>TST</td>
<td>$L + \ln N$</td>
<td>0.72</td>
</tr>
<tr>
<td></td>
<td>$\ln N$</td>
<td>38.7</td>
</tr>
<tr>
<td>TST with $R^2$</td>
<td>$L + \ln N$</td>
<td>0.51</td>
</tr>
<tr>
<td></td>
<td>$\ln N$</td>
<td>32.7</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.
- Only examines just enough key characters.
- 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]
5.2 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.
### String symbol table API

```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
}
```

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

<table>
<thead>
<tr>
<th>key</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td></td>
</tr>
<tr>
<td>by</td>
<td></td>
</tr>
<tr>
<td>sea</td>
<td>by</td>
</tr>
<tr>
<td>sel</td>
<td></td>
</tr>
<tr>
<td>sell</td>
<td></td>
</tr>
<tr>
<td>sells</td>
<td>by</td>
</tr>
<tr>
<td>sh</td>
<td></td>
</tr>
<tr>
<td>she</td>
<td>by</td>
</tr>
<tr>
<td>shell</td>
<td></td>
</tr>
<tr>
<td>shells</td>
<td>by</td>
</tr>
<tr>
<td>sho</td>
<td></td>
</tr>
<tr>
<td>shor</td>
<td></td>
</tr>
<tr>
<td>shore</td>
<td>by</td>
</tr>
<tr>
<td>t</td>
<td></td>
</tr>
<tr>
<td>th</td>
<td></td>
</tr>
<tr>
<td>the</td>
<td></td>
</tr>
</tbody>
</table>

Collecting the keys in a trie (trace)
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);
}
```

sequence of characters on path from root to node

or use StringBuilder
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.

```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;
}
```

<table>
<thead>
<tr>
<th>key</th>
<th>queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>sh</td>
<td>she</td>
</tr>
<tr>
<td>she</td>
<td>shell</td>
</tr>
<tr>
<td>shell</td>
<td>shells</td>
</tr>
<tr>
<td>shells</td>
<td>sho</td>
</tr>
<tr>
<td>sho</td>
<td>shor</td>
</tr>
<tr>
<td>shor</td>
<td>she</td>
</tr>
<tr>
<td>shore</td>
<td>she</td>
</tr>
<tr>
<td>she</td>
<td>shells</td>
</tr>
<tr>
<td>shells</td>
<td>shore</td>
</tr>
<tr>
<td>shore</td>
<td>she</td>
</tr>
<tr>
<td>she</td>
<td>shells</td>
</tr>
<tr>
<td>shells</td>
<td>shore</td>
</tr>
</tbody>
</table>
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: \( \text{floor}("128.112.100.16") = "128.112.055.15" \)
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.

Possibilities for `longestPrefixOf()`
Longest prefix in an R-way trie: 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.
Ex. hello: 4 4 3 3 5 5 5 5 5 5 6 6 6

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

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

Ex. hello: 4 3 5 5 6

Q. How to implement?
**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.
Suffix tree

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

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 (!!!)