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></td>
<td></td>
<td></td>
</tr>
<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>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 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>space (references)</td>
</tr>
<tr>
<td>red–black BST</td>
<td>$L + c \lg^2 n$</td>
<td>$4n$</td>
</tr>
<tr>
<td>hashing (linear probing)</td>
<td>$L$</td>
<td>$4n$ to $16n$</td>
</tr>
<tr>
<td></td>
<td>search miss</td>
<td>$c \lg^2 n$</td>
</tr>
<tr>
<td></td>
<td>insert</td>
<td>$c \lg^2 n$</td>
</tr>
</tbody>
</table>

$n = \text{number of string}$  
$L = \text{length of string}$  
$R = \text{radix}$

Challenge. Efficient performance for string keys.
String symbol table basic API

**String symbol table.** Symbol table specialized to string keys.

```java
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.
5.2 Tries

- 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>

```
key value
by 4
sea 6
sells 1
she 0
shells 3
shore 7
the 5
```
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 in node corresponding to last character in key (return 3)
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.

\[
\text{get("shell")}
\]
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.

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 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];
}
```

![Trie diagram with labels and annotations](image)

**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) {
    return get(root, key, 0);
}

private Value get(Node x, String key, int d) {
    if (x == null) return null;
    if (d == key.length())
        return (Value) x.val; // cast needed
    char c = key.charAt(d);
    return get(x.next[c], key, d+1);
}
```
What is order of growth of the running time (in the worst case) to insert a key of length $L$ into an $R$-way trie?

A. $L$
B. $R + L$
C. $n + L$
D. $RL$

$R = \text{alphabet size}$
$L = \text{length of key}$
$n = \text{number of keys}$
**Trie performance**

**Search hit.** Need to examine all $L$ characters for equality.

**Search miss.**
- Worst case: examine $L$ characters.
- Typical case: examine only a few characters before mismatch (sublinear).

**Space.** At least $R$ links per key.

**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).

```javascript
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).

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>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>
</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 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).

Fast Algorithms for Sorting and Searching Strings

Jon L. Bentley* Robert Sedgewick#

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 algo-

that 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 implementa-

tion. 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 3 children: smaller (left), equal (middle), larger (right).
Trie quiz 2

Which value is associated with the key CAC?

A. 3
B. 4
C. 5
D. null

![Trie diagram](image-url)
Search hit in a TST

get("sea")

return value in node corresponding to last character in key
Search miss in a TST

get("shelter")
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
In which subtrie would the key **CCC** be inserted?

A.  
B.  
C.  
D.  
E.
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.
- 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 Value get(String key)
    {
        return get(root, key, 0);
    }

    private Value 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.val;
    }

    public void put(String Key, Value val)
    {
        /* similar, see book or booksite */
    }
}
### 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>$c \log^2 n$</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.
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>

**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]
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.

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
    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.
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.

```java
keysWithPrefix("sh");
```

- **find subtrie for all keys beginning with "sh"**
- **collect keys in that subtrie**
Longest prefix

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

**Ex 1.** To send packet toward destination IP address, router chooses IP address in routing table that is longest prefix match.

```plaintext
"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)

```plaintext
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

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

**Ex 2.** LZW compression.

```
input  ...  R  A  B  R  A  B  R  A  B  R  O  C  C  O  L  I  ...  
value  89  82  88  88 
```
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()
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 6 3

“a much faster and more fun way to enter text”

**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.

http://www.t9.com
Q. How to implement T9 texting on a mobile phone?
To: info@t9support.com
Date: Tue, 25 Oct 2005 14:27:21 -0400 (EDT)

Dear T9 texting folks,

I enjoyed learning about the T9 text system from your webpage, and used it as an example in my algorithms and data structures class. However, one of my students noticed a bug in your phone keypad

   http://www.t9.com/images/how.gif

Somehow, it is missing the letter 's'. (!)

Just wanted to bring this information to your attention and thank you for your website.

Regards,
Kevin
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. uite 300
eattle, WA 98109

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

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.