5.2 Tries

- R-way tries
- ternary search tries
- string symbol table API
**Review: summary of the performance of symbol-table implementations**

**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>1.00 lg N</td>
<td>1.00 lg N</td>
<td>1.00 lg N</td>
</tr>
<tr>
<td>hashing</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

boolean contains(String key) is there a value paired with the given key?

Goal. Faster than hashing, more flexible than binary search trees.
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</td>
<td>(L)</td>
<td>(L)</td>
</tr>
</tbody>
</table>

<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>

Parameters
- \(N\) = number of strings
- \(L\) = length of string
- \(R\) = radix

**Challenge.** Efficient performance for string keys.
- R-way tries
  - ternary search tries
  - string symbol table API
Tries

Tries. [from retrieval, but pronounced "try"]
• Store characters and values in nodes (not keys).
• Each node has $R$ children, one for each possible character.
• For now, we do not draw null links.

Ex. she sells sea shells by the
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")
```

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

**Diagram:**
- **Return the value in the node corresponding to the last key character (3)**
- **Return the value in the node corresponding to the last key character (0)**
- **Search may terminate at an internal node**
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.
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.
Trie construction example

key  value
she  0

sells 1

sea  2

key  value
shells 3

by  4

shore 7

the  5

root

value is in node corresponding to last character

nodes corresponding to characters at the end of the key do not exist, so create them and set the value of the last one

node corresponding to the last key character exists, so reset its value

key is sequence of characters from root to value

one node for each key character
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];
}
```

- **Use** `Object` instead of `Value` since no generic array creation in Java.
- **Characters** are implicitly defined by link index.
- **Keys** are not explicitly stored.
- **Each node** has an array of links and a value.
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 ($R = 26$)

- Each node has an array of links and a value.
- Characters are implicitly defined by link index.
- 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;

    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;  
}

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 miss.
- Could have mismatch on first character.
- Typical case: examine only a few characters (sublinear).

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

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>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</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 $R$.
- 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.”
— 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.”

— RS, 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.

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
- string symbol table API
Ternary search tries

**TST.** [Bentley-Sedgewick, 1997]

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

TST. [Bentley-Sedgewick, 1997]
• Store characters and values in nodes (not keys).
• Each node has three children: smaller (left), equal (middle), larger (right).

TST representation of a trie
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.
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;
}
```
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;
    }
}
TST: Java implementation (continued)

```java
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</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></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></td>
</tr>
<tr>
<td></td>
<td>(c \log^2 N)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4 (N)</td>
<td>97.4</td>
</tr>
<tr>
<td>hashing</td>
<td>(L)</td>
<td>0.76</td>
</tr>
<tr>
<td></td>
<td>(L)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(L)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4 (N) to 16 (N)</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></td>
</tr>
<tr>
<td></td>
<td>(L)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>((R + 1) N)</td>
<td></td>
</tr>
<tr>
<td>TST</td>
<td>(L + \ln N)</td>
<td>0.72</td>
</tr>
<tr>
<td></td>
<td>(\ln N)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(L + \ln N)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4 (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></td>
</tr>
<tr>
<td></td>
<td>(L + \ln N)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4 (N) + (R^2)</td>
<td>32.7</td>
</tr>
</tbody>
</table>
TST vs. hashing

Hashing.
- Need to examine entire key.
- Search hits and misses cost about the same.
- Need good hash function for every key type.
- No help for ordered symbol table operations.

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

Bottom line. TSTs are:
- Faster than hashing (especially for search misses).
  More flexible than red-black trees (next).
- R-way tries
- ternary search tries
- string symbol table API
Character-based operations. The string symbol table API supports several useful character-based operations.

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

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

Wildcard match. Keys that match ".he": "she" and "the".
String symbol table API

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StringST()</td>
<td>create a symbol table with string keys</td>
</tr>
<tr>
<td>StringST(Alphabet alpha)</td>
<td>create a symbol table with string keys whose characters are taken from alpha.</td>
</tr>
<tr>
<td>void put(String key, Value val)</td>
<td>put key-value pair into the symbol table (remove key from table if value is null)</td>
</tr>
<tr>
<td>Value get(String key)</td>
<td>value paired with key (null if key is absent)</td>
</tr>
<tr>
<td>void delete(String key)</td>
<td>remove key (and its value) from table</td>
</tr>
<tr>
<td>boolean contains(String key)</td>
<td>is there a value paired with key?</td>
</tr>
<tr>
<td>boolean isEmpty()</td>
<td>is the table empty?</td>
</tr>
<tr>
<td>String longestPrefixOf(String s)</td>
<td>return the longest key that is a prefix of s</td>
</tr>
<tr>
<td>Iterable&lt;String&gt; keysWithPrefix(String s)</td>
<td>all the keys having s as a prefix.</td>
</tr>
<tr>
<td>Iterable&lt;String&gt; keysThatMatch(String s)</td>
<td>all the keys that match s (where . matches any character).</td>
</tr>
<tr>
<td>int size()</td>
<td>number of key-value pairs in the table</td>
</tr>
<tr>
<td>Iterable&lt;String&gt; keys()</td>
<td>all the keys in the symbol table</td>
</tr>
</tbody>
</table>

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

Deleting a key (and its associated value) from a trie
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.

keysWithPrefix("");

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);
}
```
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.
Prefix matches

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

Prefix match in a trie

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

Use wildcard . to match any character in alphabet.

coalizer  acresce
coberger  acroach
codifier  acuracy
cofaster  octarch
cofather  science
cognizer  scranch
cohelper  scratch
colander  scrauch
coleader  screich
...  scrinch
compiler  scritch
...  scrunch
composer  scudick
computer  scutock
cowkeeper

c...er  .c...c.
Wildcard matches

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

```java
public Iterable<String> keysThatMatch(String pat) {
    Queue<String> queue = new Queue<String>();
    collect(root, "", 0, pat, queue);
    return queue;
}

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.** Search IP database for longest prefix matching destination IP, and route packets accordingly.

```
"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"
```

prefix("128.112.136.11") = "128.112.136"
prefix("128.166.123.45") = "128"

**Note.** Not the same as floor.

```
prefix("128.112.100.16") = "128.112"
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.** ["A much faster and more fun way to enter text."]

- 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 5 6 6 6 6
- T9: 4 3 5 5 6

[Diagram of phone keypad with T9 text input features]
Compressing a trie

Collapsing 1-way branches at bottom.
Internal node stores character; leaf node stores suffix (or full key).

Collapsing interior 1-way branches.
Node stores a sequence of characters.
A classic algorithm

**Patricia tries.** [Practical Algorithm to Retrieve Information Coded in Alphanumeric]

- Collapse one-way branches in binary trie.
- Thread trie to eliminate multiple node types.

![Diagram of Patricia trie]

**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.

**Implementation.** One step beyond this lecture.
Suffix tree

**Suffix tree.** Threaded trie with collapsed 1-way branching for string suffixes.

Applications.
- Linear-time longest repeated substring.
- Computational biology databases (BLAST, FASTA).

Implementation. One step beyond this lecture.
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 (!!!)