



<http://algs4.cs.princeton.edu>

## 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.

| implementation | typical case |             |             | ordered operations | operations on keys                               |
|----------------|--------------|-------------|-------------|--------------------|--|
|                | search       | insert      | delete      |                    |  |
| red-black BST  | $\log N$     | $\log N$    | $\log N$    | yes                | <code>compareTo()</code>                         |
| hash table     | $1^\dagger$  | $1^\dagger$ | $1^\dagger$ | no                 | <code>equals()</code><br><code>hashCode()</code> |

$^\dagger$  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

| implementation              | character accesses (typical case) |             |             |                    | dedup    |            |
|-----------------------------|-----------------------------------|-------------|-------------|--------------------|----------|------------|
|                             | search hit                        | search miss | insert      | space (references) | moby.txt | actors.txt |
| red-black BST               | $L + c \lg^2 N$                   | $c \lg^2 N$ | $c \lg^2 N$ | $4N$               | 1.40     | 97.4       |
| hashing<br>(linear probing) | $L$                               | $L$         | $L$         | $4N$ to $16N$      | 0.76     | 40.6       |

## Parameters

- $N$  = number of strings
- $L$  = length of string
- $R$  = radix

| file       | size   | words  | distinct |
|------------|--------|--------|----------|
| moby.txt   | 1.2 MB | 210 K  | 32 K     |
| actors.txt | 82 MB  | 11.4 M | 900 K    |

**Challenge.** Efficient performance for string keys.



## 5.2 TRIES

---

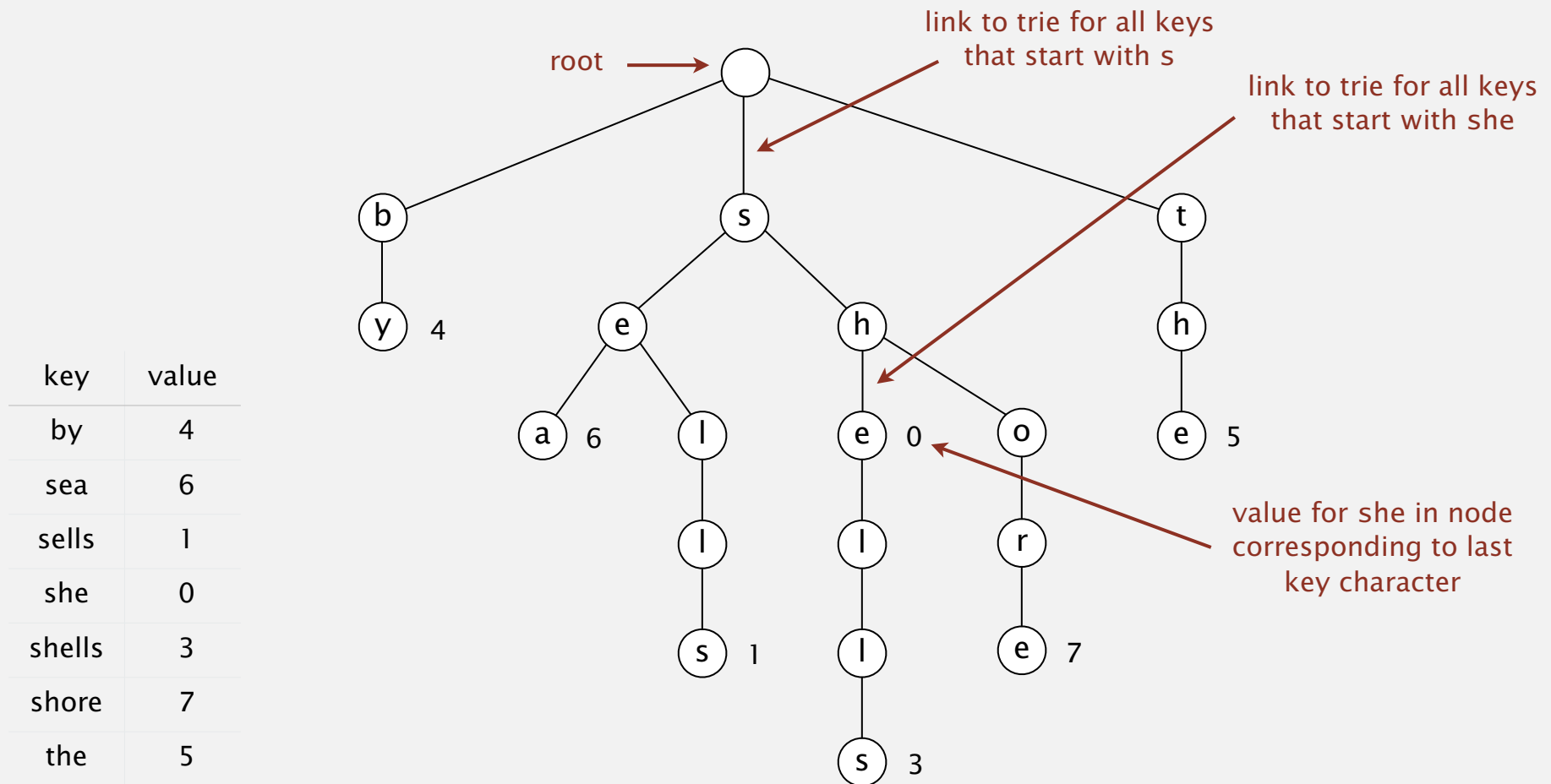
- ▶ *R-way tries*
- ▶ *ternary search tries*
- ▶ *character-based operations*

# Tries

**Tries.** [from **re**trieval, 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.

for now we do not  
draw null links

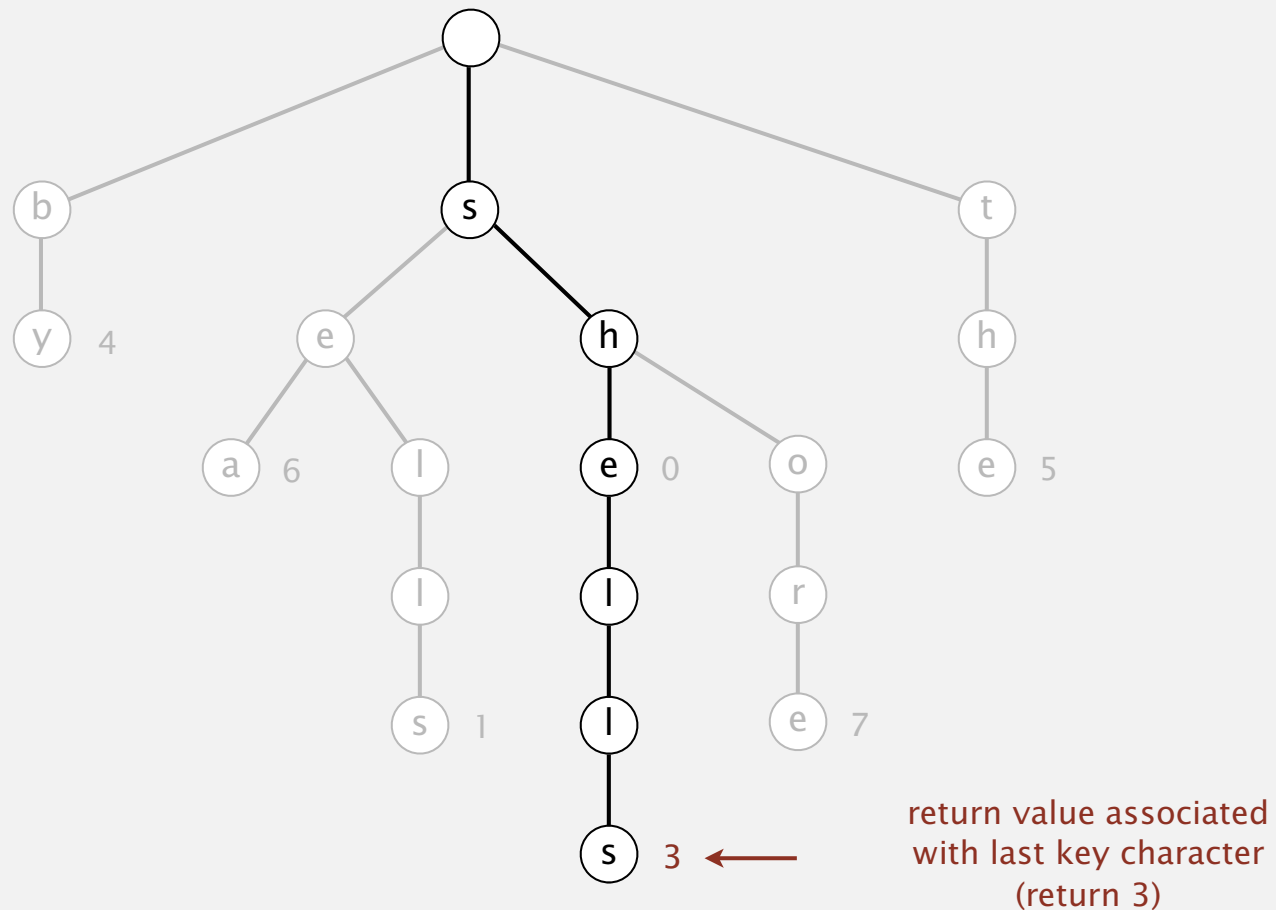


# Search in a trie

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.

**get("shells")**

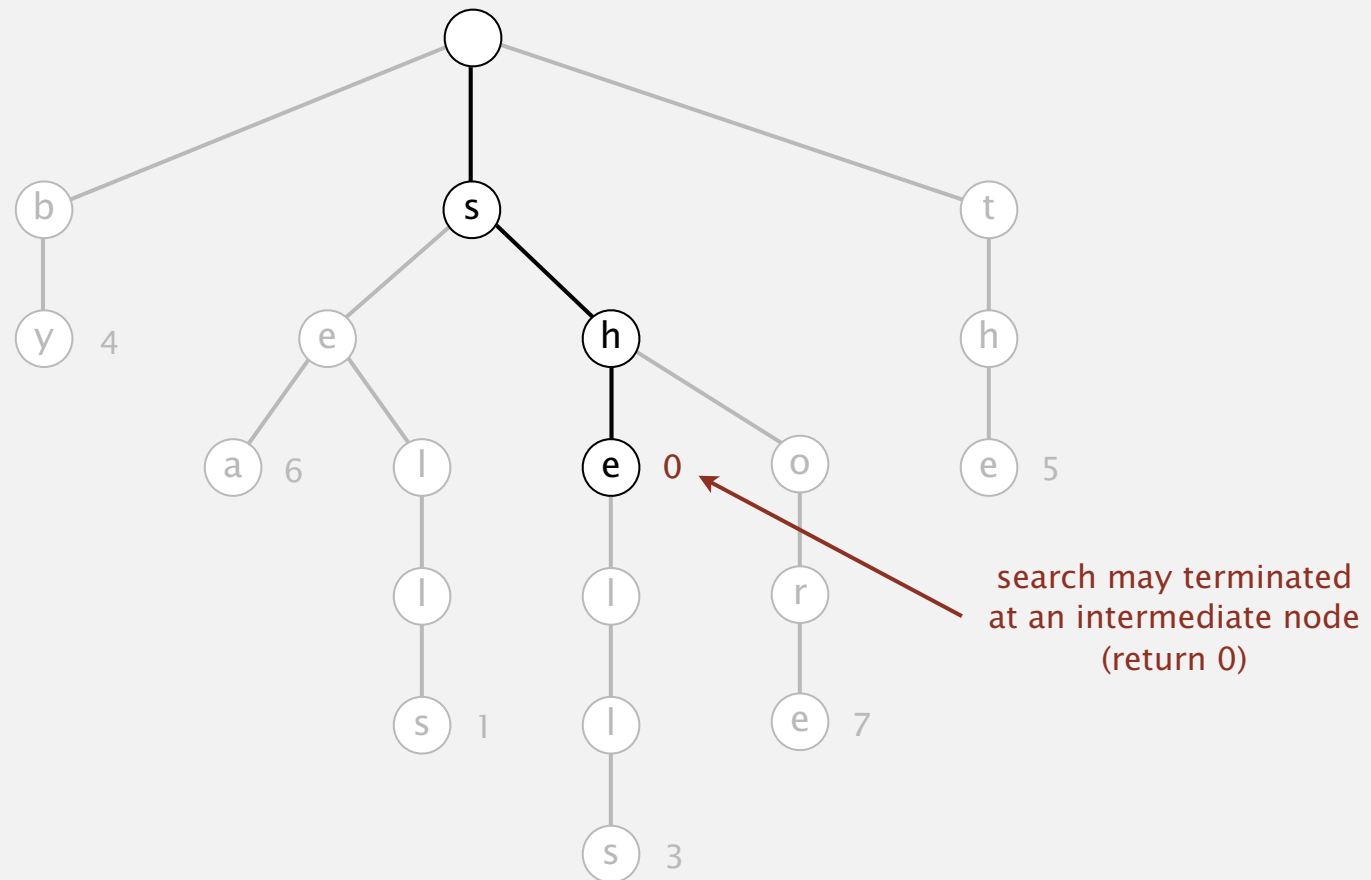


## Search in a trie

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")
```



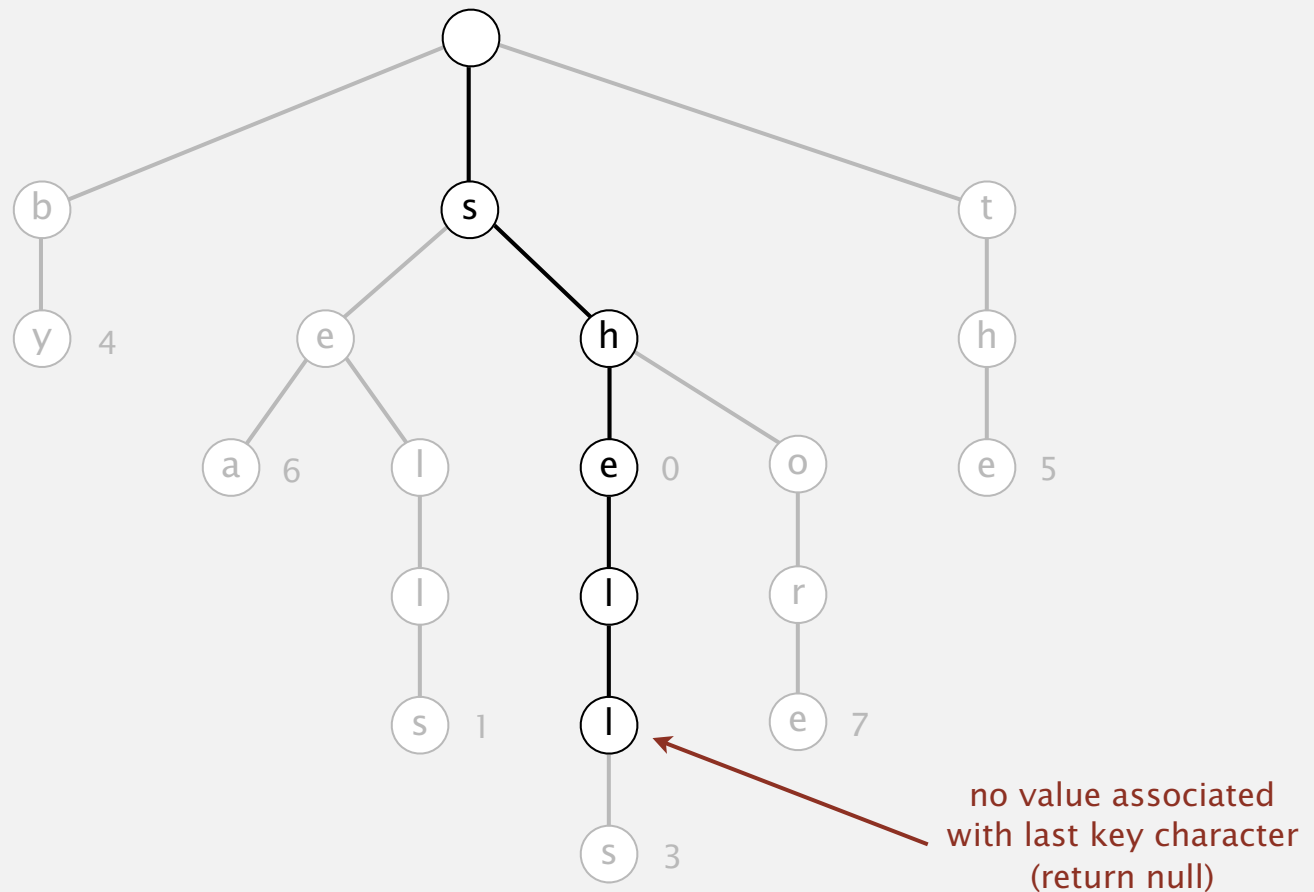


## Search in a trie

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")
```

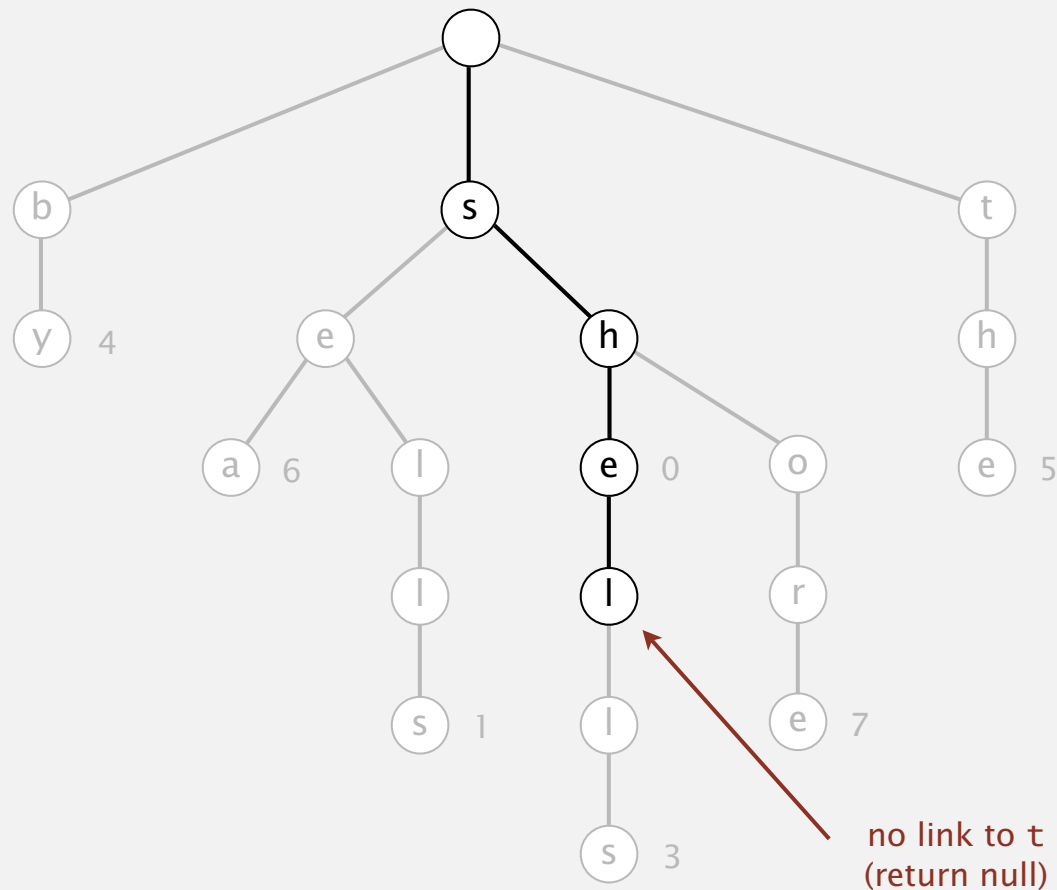


## Search in a trie

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.

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

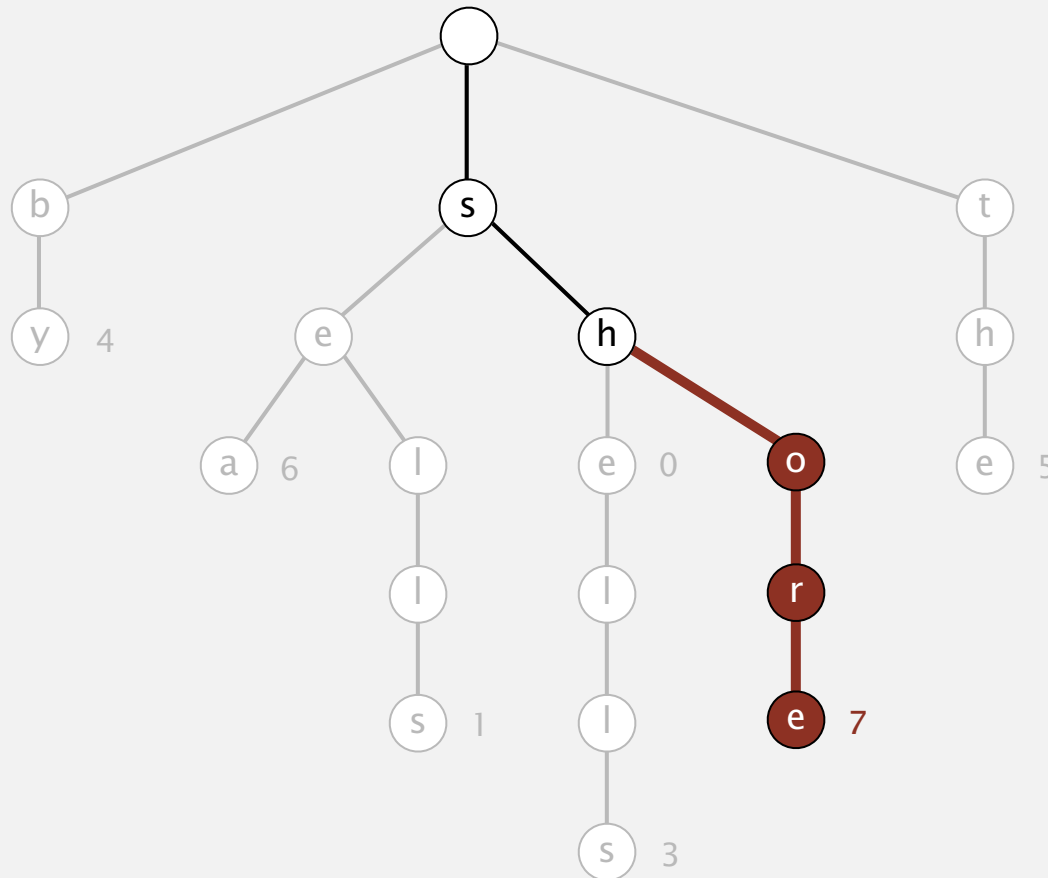


## Insertion into a trie

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

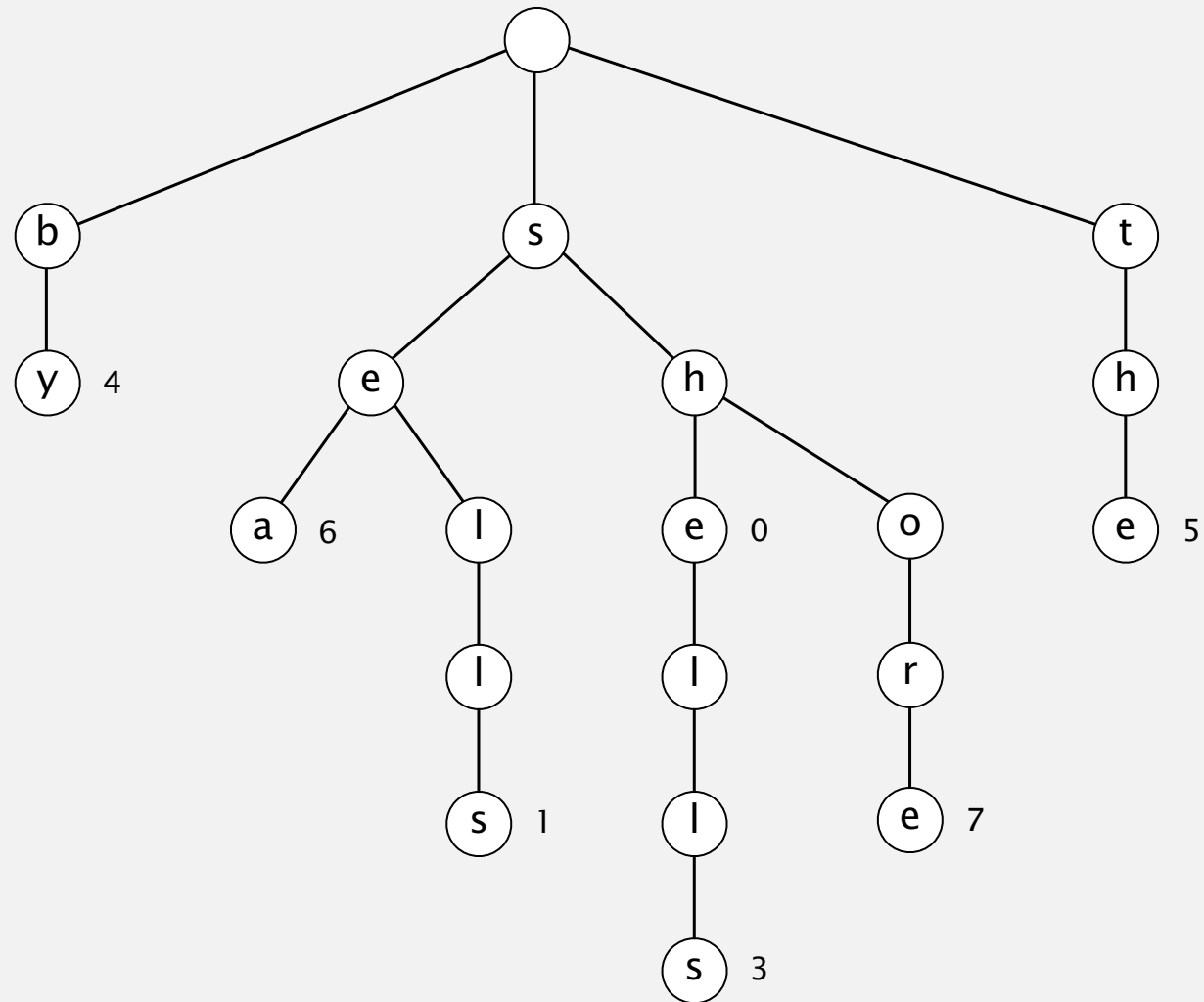
---

trie



# Trie construction demo

## trie

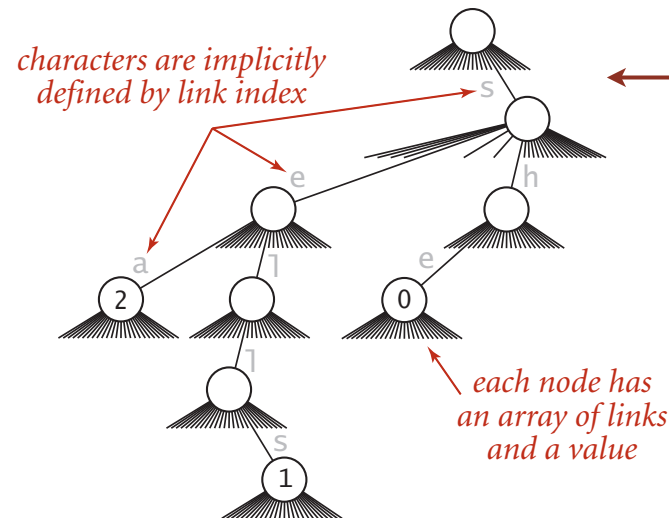
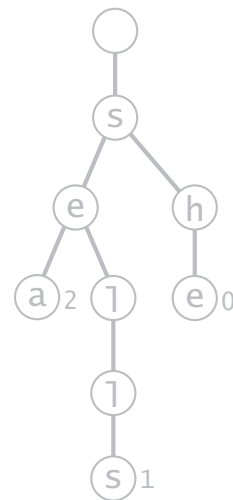


## Trie representation: Java implementation

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

```
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



- neither keys nor characters are explicitly stored

- each node has an array of links and a value

## Trie representation

## R-way trie: Java implementation

---

```
public class TrieST<Value>
{
    private static final int R = 256;    ← extended ASCII
    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)

---

```
⋮  
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

---

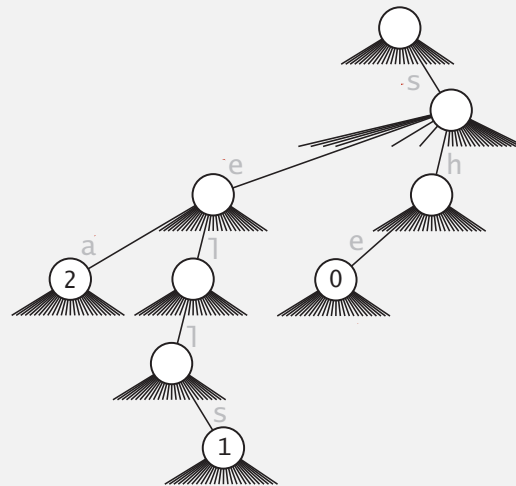
**Search hit.** Need to examine all  $L$  characters for equality.

**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)



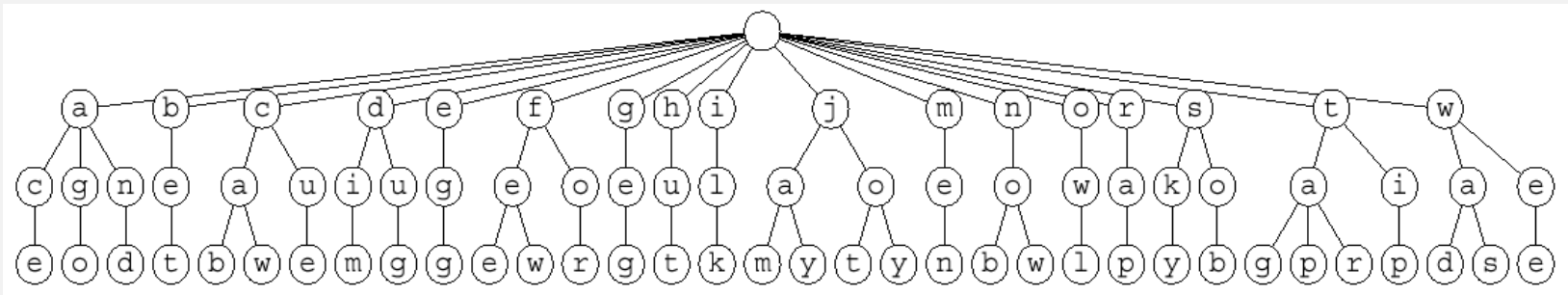
**Bottom line.** Fast search hit and even faster search miss, but wastes space.

# Popular interview question

**Goal.** Design a data structure to perform efficient spell checking.

**Solution.** Build a 26-way trie (key = word, value = bit).

English-language text



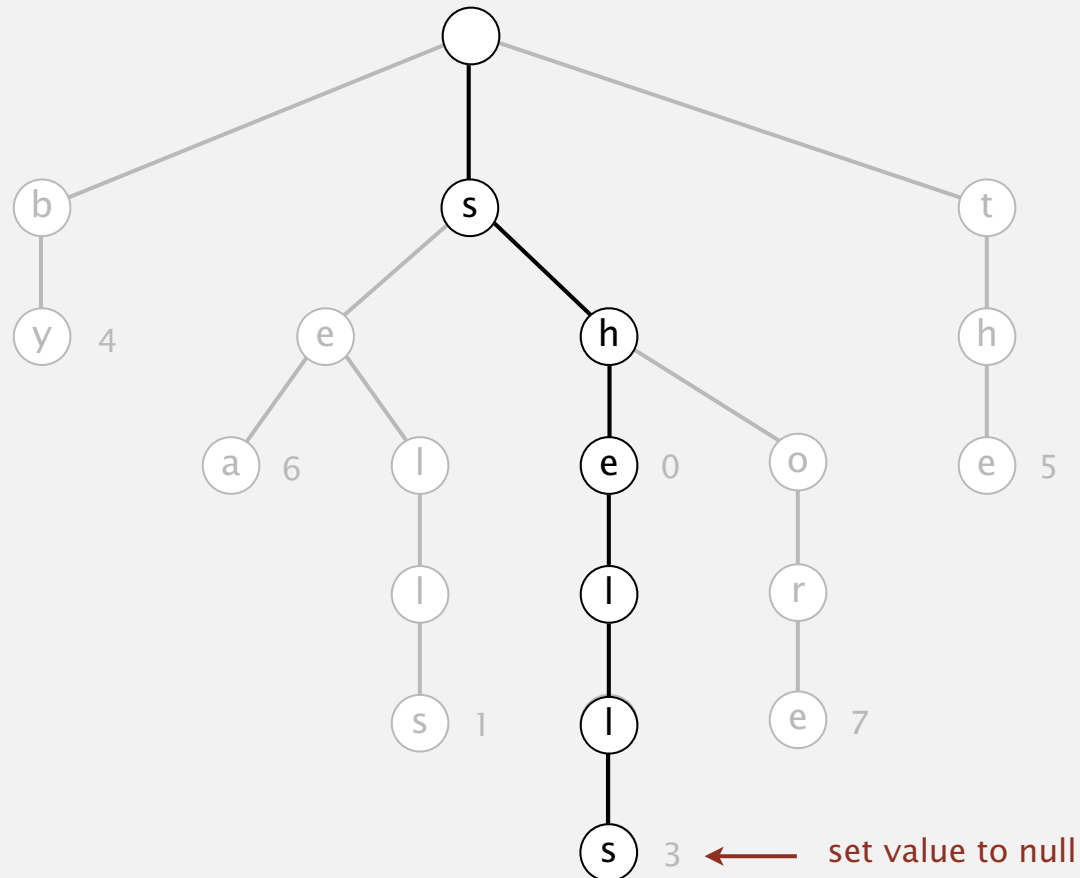
ace  
ago  
and  
bet  
cab  
caw  
cue  
dim  
dug  
egg  
fee  
few  
for  
gig  
hut  
ilk  
jam  
jay  
jot  
joy  
men  
nob  
now  
owl  
rap  
sky  
sob  
tag  
tap  
tar  
tip  
wad  
was  
wee

# 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")**

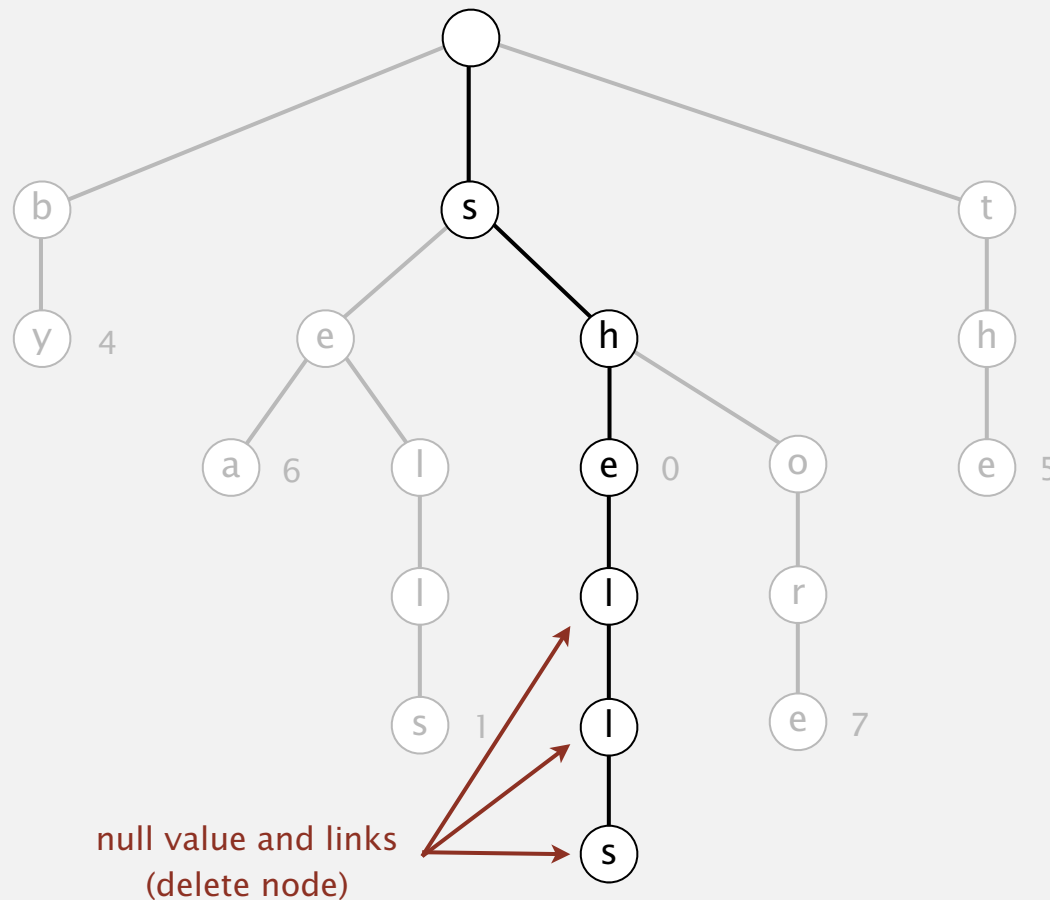


# Deletion in an R-way trie

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

| implementation           | character accesses (typical case) |             |             |                    | dedup    |               |
|--------------------------|-----------------------------------|-------------|-------------|--------------------|----------|---------------|
|                          | search hit                        | search miss | insert      | space (references) | moby.txt | actors.txt    |
| red-black BST            | $L + c \lg^2 N$                   | $c \lg^2 N$ | $c \lg^2 N$ | $4N$               | 1.40     | 97.4          |
| hashing (linear probing) | $L$                               | $L$         | $L$         | $4N$ to $16N$      | 0.76     | 40.6          |
| R-way trie               | $L$                               | $\log_R N$  | $L$         | $(R+1) N$          | 1.12     | out of memory |

## 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*



## 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 implementation. The symbol table implementation is much more space-efficient than multiway trees, and supports more advanced searches.

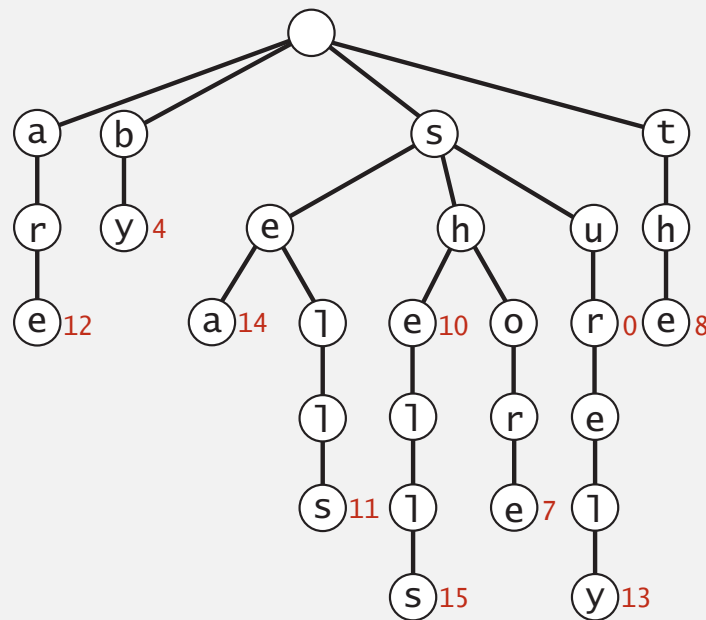
In many application programs, sorts use a Quicksort implementation based on an abstract compare operation,



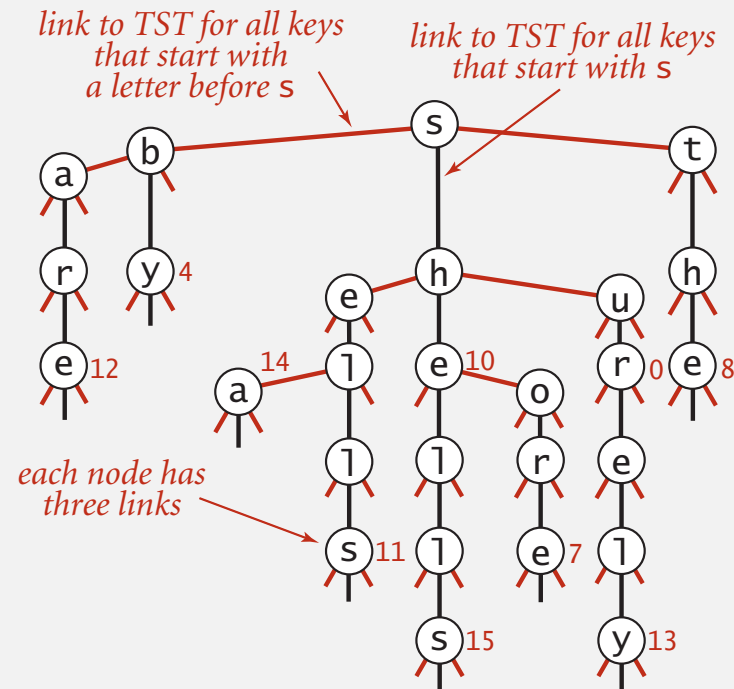


# Ternary search tries

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



TST representation of a trie



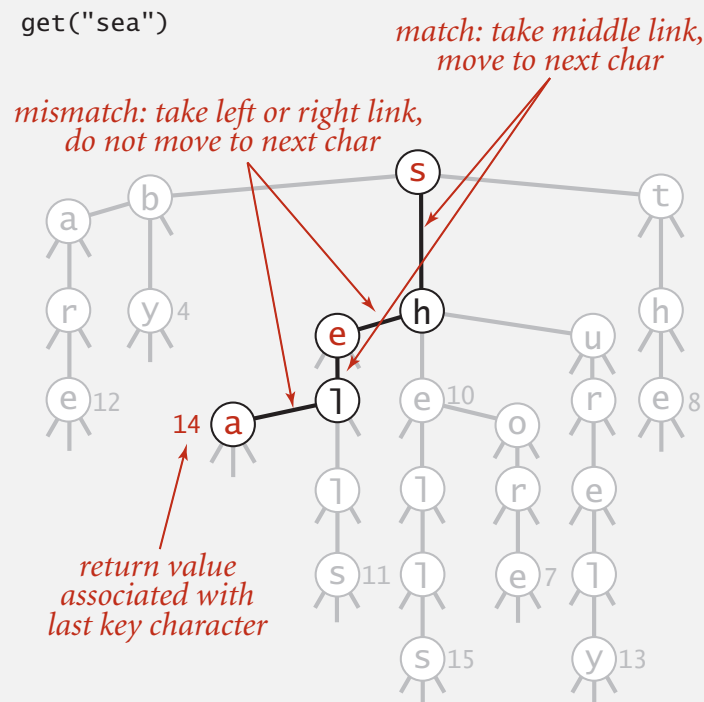
## Search in a TST

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.

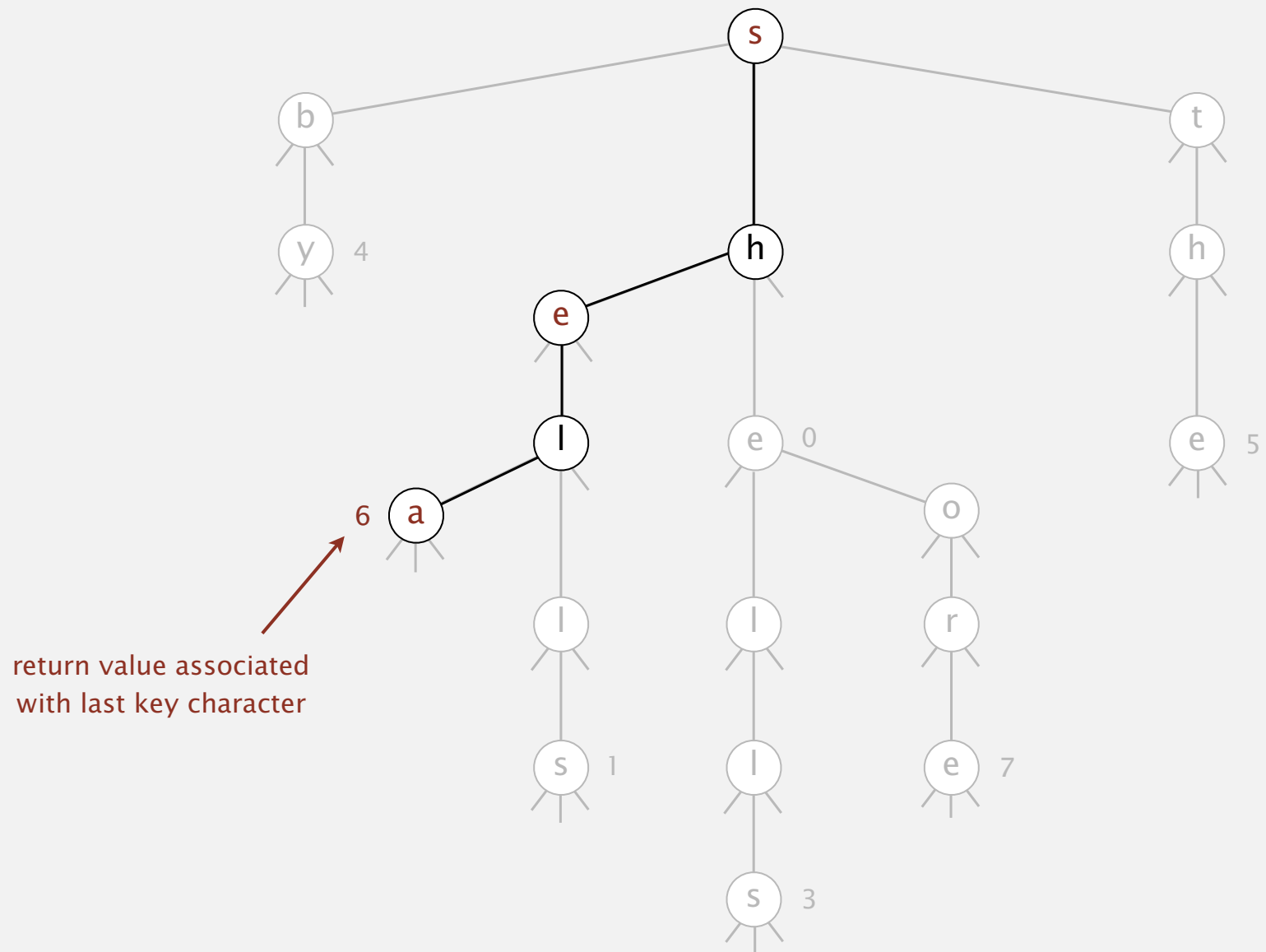
**Search hit.** Node where search ends has a non-null value.

**Search miss.** Reach a null link or node where search ends has null value.



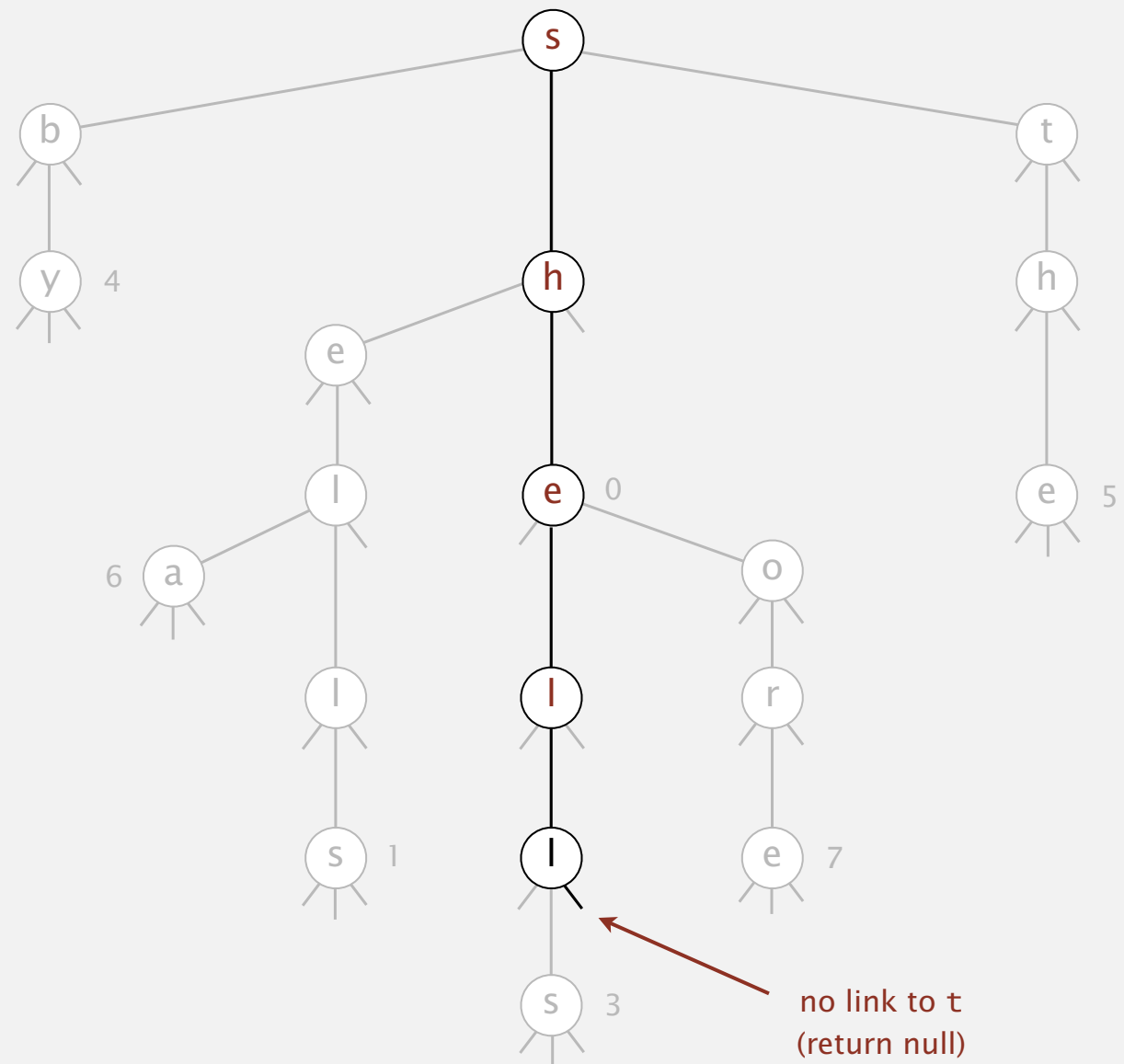
# Search hit in a TST

get("sea")



# Search miss in a TST

`get("shelter")`



# Ternary search trie construction demo

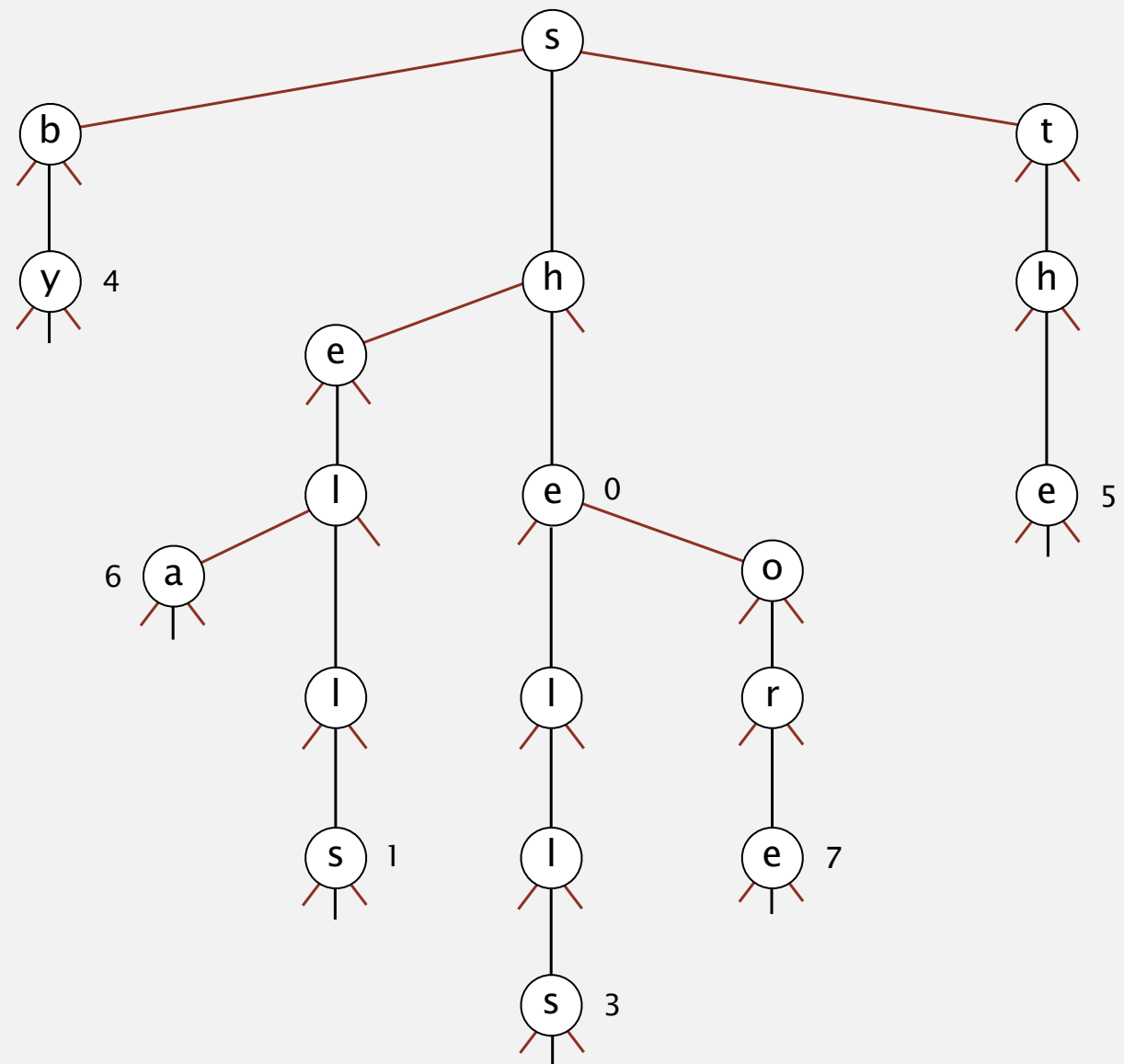
---

ternary search trie



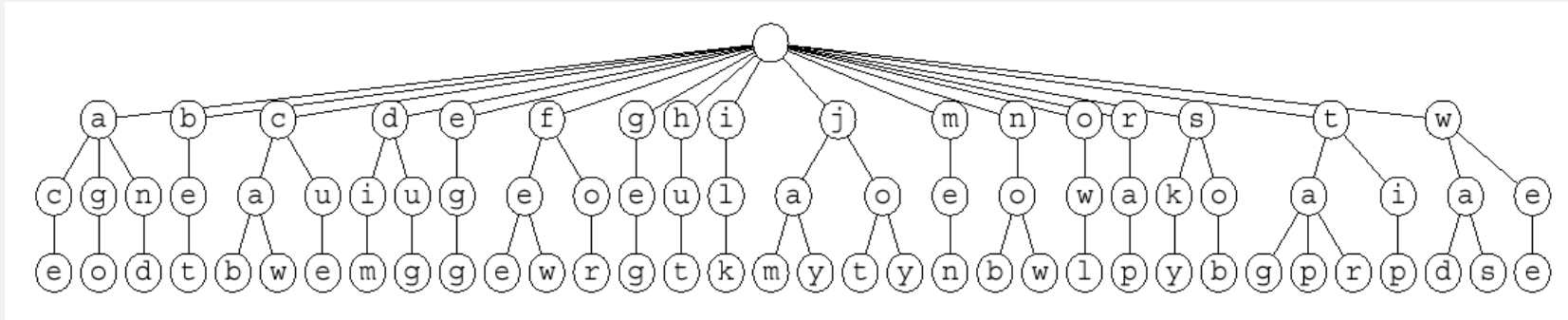
# Ternary search trie construction demo

ternary search trie



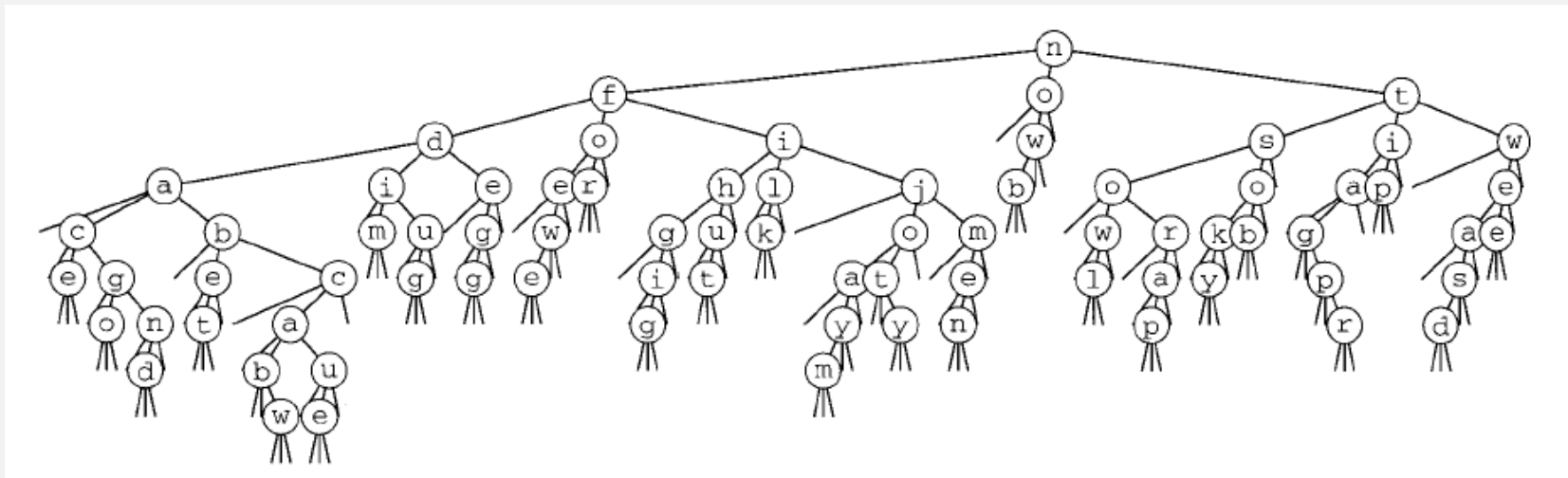
# 26-way trie vs. TST

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



26-way trie (1035 null links, not shown)

TST. 3 null links in each leaf.



TST (155 null links)

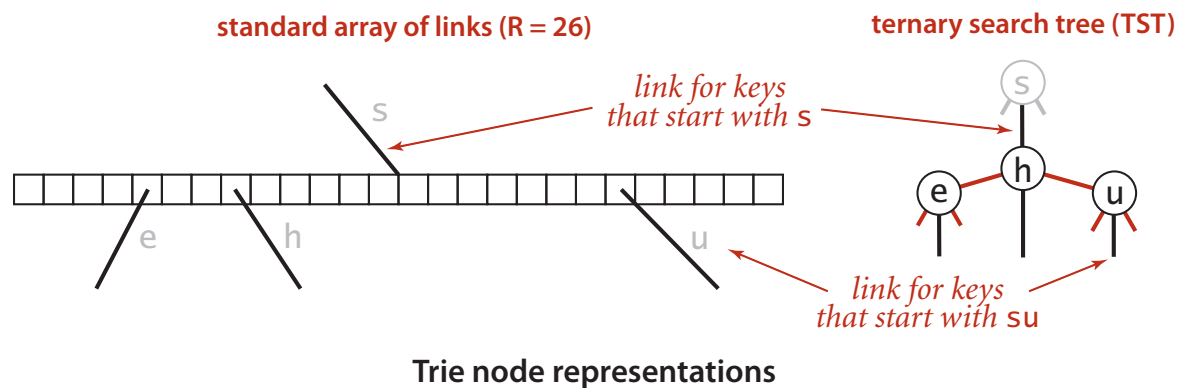
now  
for  
tip  
ilk  
dim  
tag  
jot  
sob  
nob  
sky  
hut  
ace  
bet  
men  
egg  
few  
jay  
owl  
joy  
rap  
gig  
wee  
was  
cab  
wad  
caw  
cue  
fee  
tap  
ago  
tar  
jam  
dug  
and

# 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.

```
private class Node
{
    private Value val;
    private char c;
    private Node left, mid, right;
}
```





## TST: Java implementation

---

```
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)

---

```
⋮  
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

| implementation              | character accesses (typical case) |             |             |                    | dedup    |               |
|-----------------------------|-----------------------------------|-------------|-------------|--------------------|----------|---------------|
|                             | search hit                        | search miss | insert      | space (references) | moby.txt | actors.txt    |
| red-black BST               | $L + c \lg^2 N$                   | $c \lg^2 N$ | $c \lg^2 N$ | $4 N$              | 1.40     | 97.4          |
| hashing<br>(linear probing) | $L$                               | $L$         | $L$         | $4 N$ to $16 N$    | 0.76     | 40.6          |
| R-way trie                  | $L$                               | $\log_R N$  | $L$         | $(R + 1) N$        | 1.12     | out of memory |
| TST                         | $L + \ln N$                       | $\ln N$     | $L + \ln N$ | $4 N$              | 0.72     | 38.7          |

**Remark.** Can build balanced TSTs via rotations to achieve  $L + \log N$  worst-case guarantees.

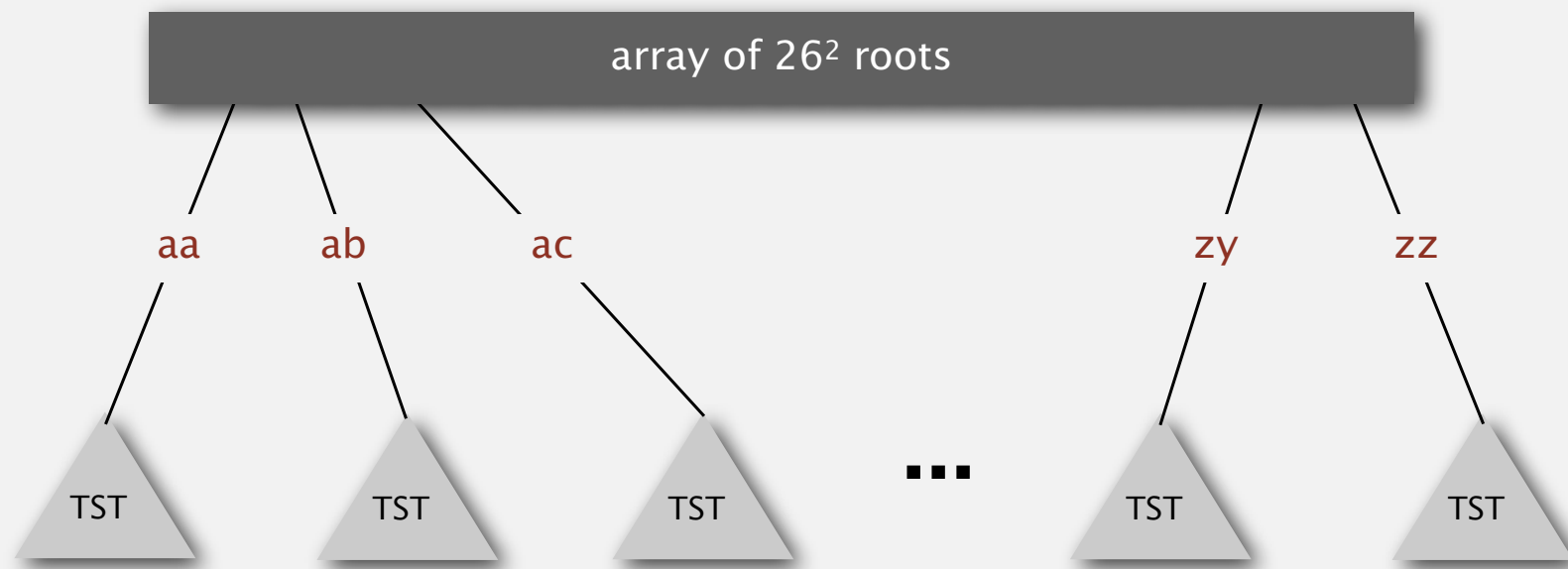
**Bottom line.** TST is as fast as hashing (for string keys), space efficient.

# 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

| implementation              | character accesses (typical case) |             |             |                    | dedup    |               |
|-----------------------------|-----------------------------------|-------------|-------------|--------------------|----------|---------------|
|                             | search hit                        | search miss | insert      | space (references) | moby.txt | actors.txt    |
| red-black BST               | $L + c \lg^2 N$                   | $c \lg^2 N$ | $c \lg^2 N$ | $4 N$              | 1.40     | 97.4          |
| hashing<br>(linear probing) | $L$                               | $L$         | $L$         | $4 N$ to $16 N$    | 0.76     | 40.6          |
| R-way trie                  | $L$                               | $\log_R N$  | $L$         | $(R + 1) N$        | 1.12     | out of memory |
| TST                         | $L + \ln N$                       | $\ln N$     | $L + \ln N$ | $4 N$              | 0.72     | 38.7          |
| TST with $R^2$              | $L + \ln N$                       | $\ln N$     | $L + \ln N$ | $4 N + R^2$        | 0.51     | 32.7          |

**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 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]



## 5.2 TRIES

---

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



## 5.2 TRIES

---

- ▶ *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.

| key    | value |
|--------|-------|
| by     | 4     |
| sea    | 6     |
| sells  | 1     |
| she    | 0     |
| shells | 3     |
| shore  | 7     |
| the    | 5     |

**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)
```

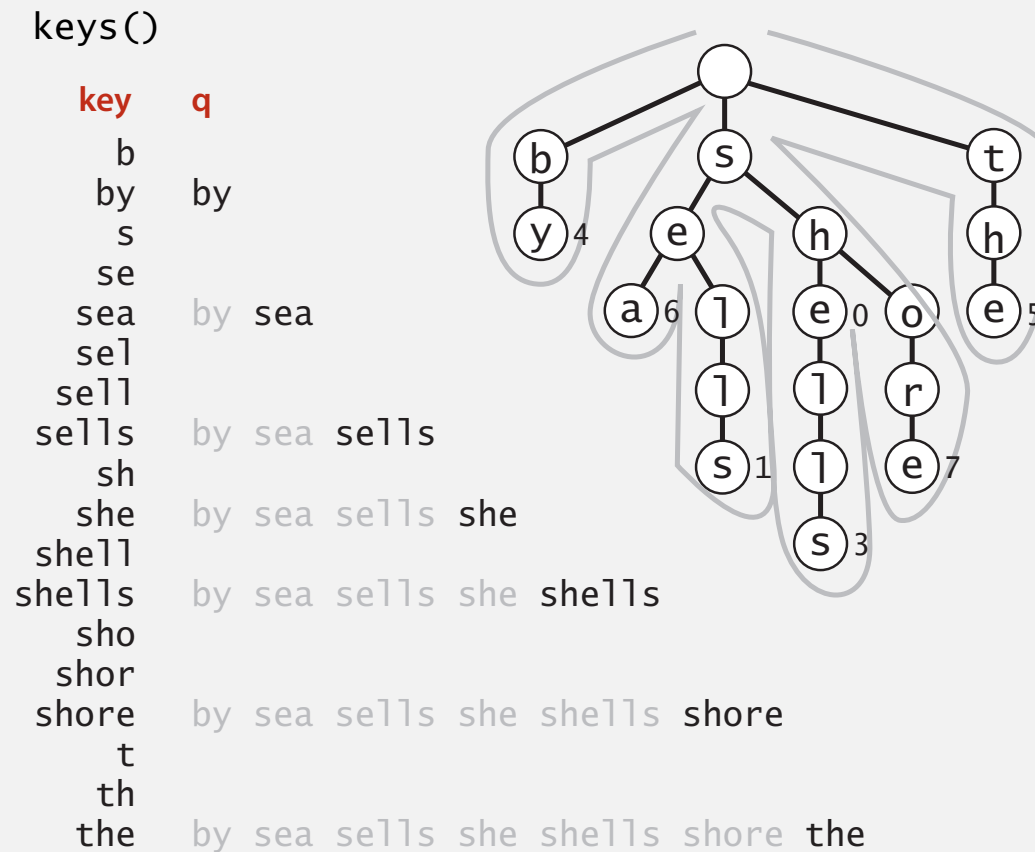
*longest key that is a prefix of s*

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

## 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

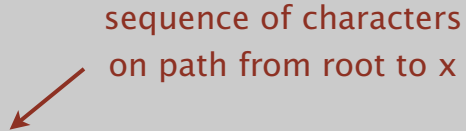
---

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.

```
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 x

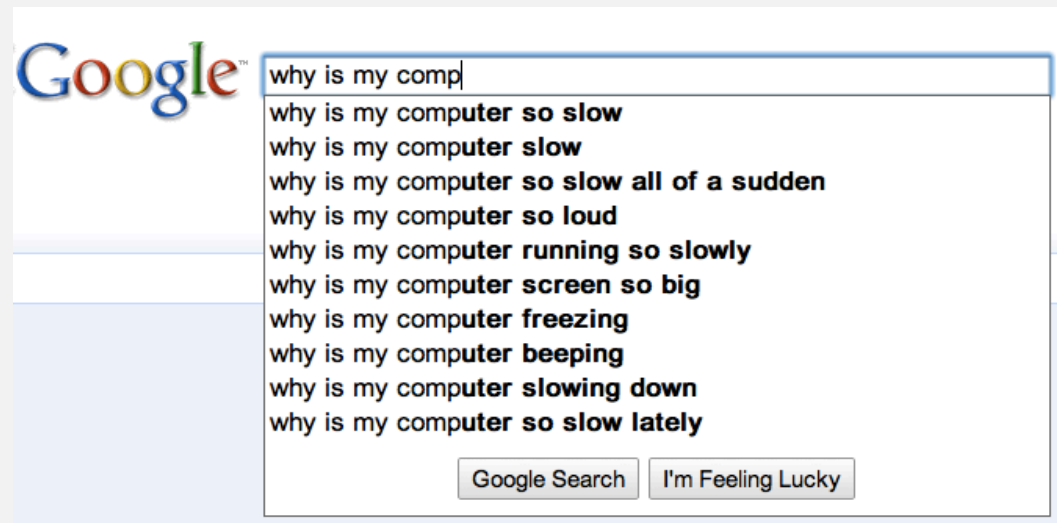
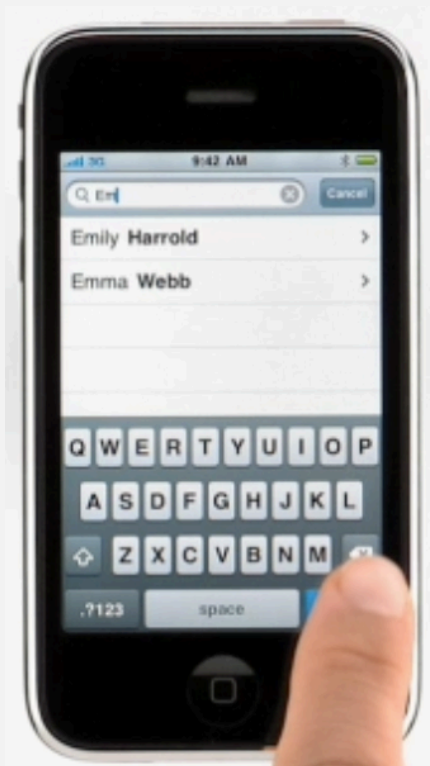
# Prefix matches

---

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

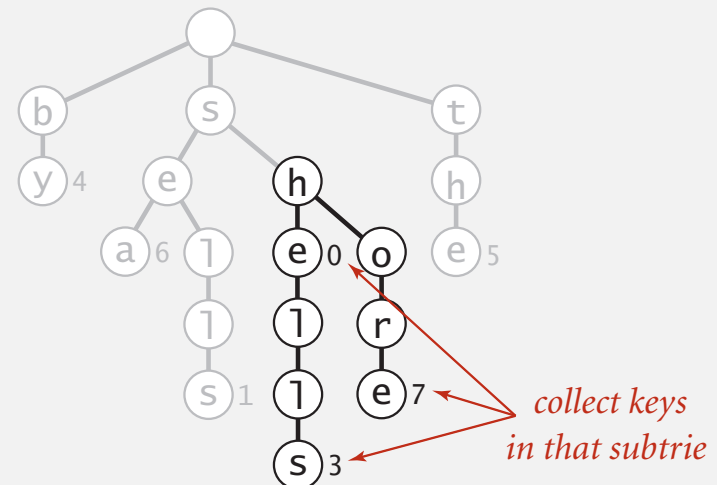
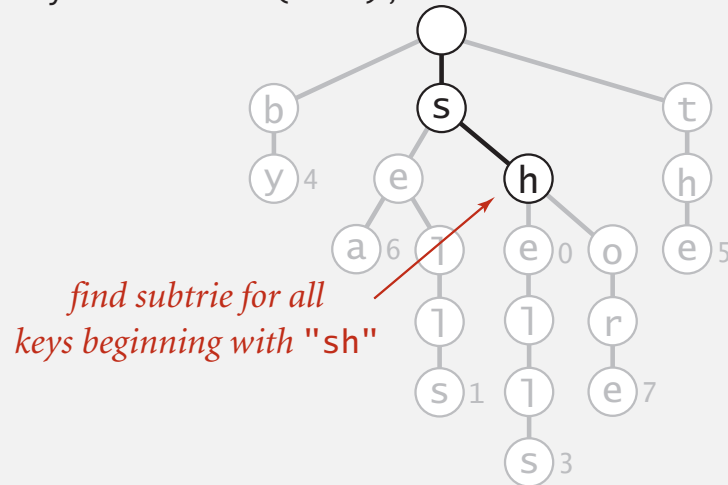
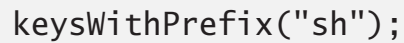
**Ex.** Autocomplete in a cell phone, search bar, text editor, or shell.

- 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;
}
```

root of subtree for all strings  
beginning with given prefix

| key    | queue            |
|--------|------------------|
| sh     |                  |
| she    | she              |
| shell  |                  |
| shells | she shells       |
| shore  | she shells shore |

## 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.

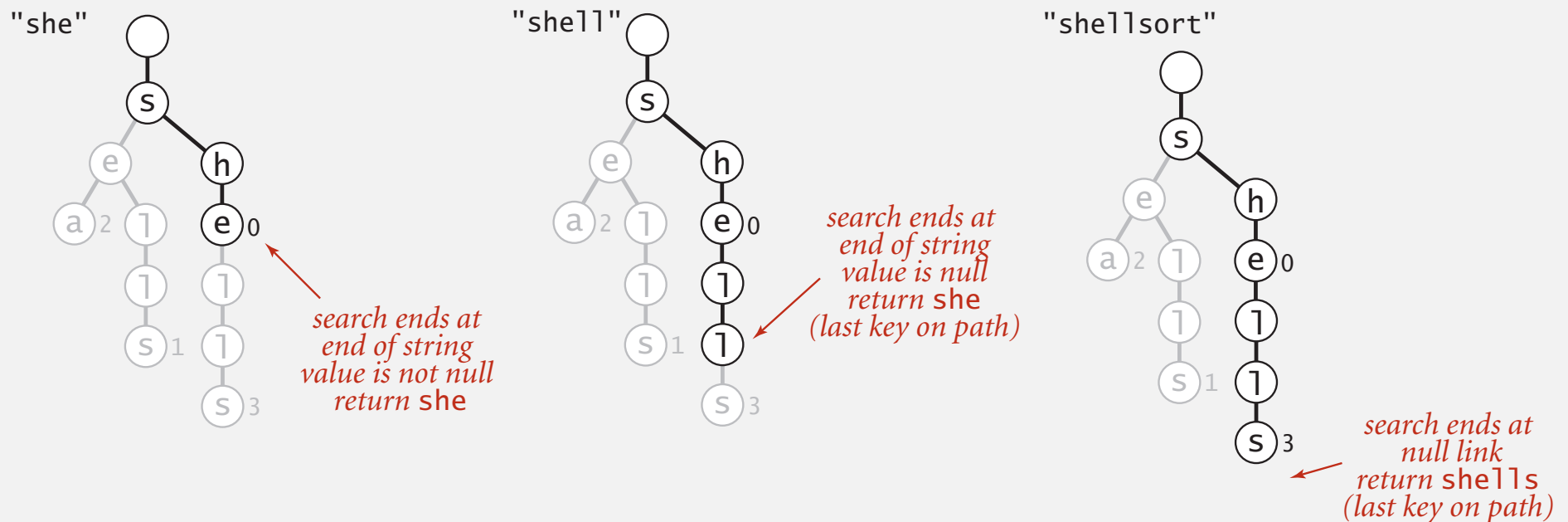
|                  |  |
|------------------|--|
| "128"            | ← represented as 32-bit<br>binary number for IPv4<br>(instead of string) |
| "128.112"        |  |
| "128.112.055"    |  |
| "128.112.055.15" |  |
| "128.112.136"    | LongestPrefixOf("128.112.136.11") = "128.112.136"                        |
| "128.112.155.11" | LongestPrefixOf("128.112.100.16") = "128.112"                            |
| "128.112.155.13" | LongestPrefixOf("128.166.123.45") = "128"                                |
| "128.222"        |  |
| "128.222.136"    |  |

**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.



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.

```
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.

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

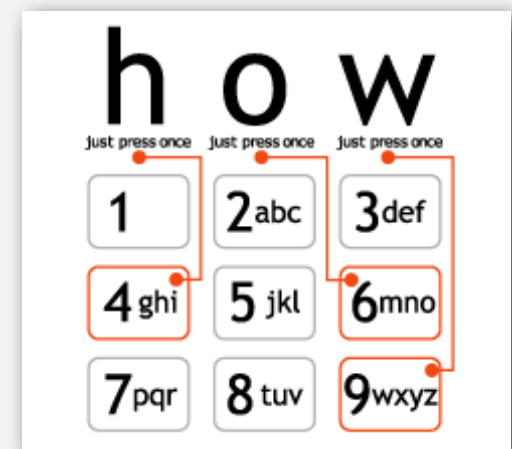
**T9 text input.**

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

**Ex.** hello

- Multi-tap: 4 4 3 3 5 5 5 5 5 5 6 6 6
- T9: 4 3 5 5 6

**Q.** How to implement?



[www.t9.com](http://www.t9.com)

# A letter to t9.com

---

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 data structures and algorithms class. However, one of my students noticed a bug in your phone keypad

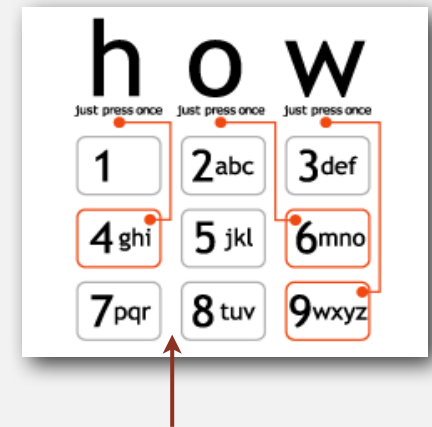
<http://www.t9.com/images/how.gif>

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

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

Regards,

Kevin



where the @#\$% is the "s" ???

## 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

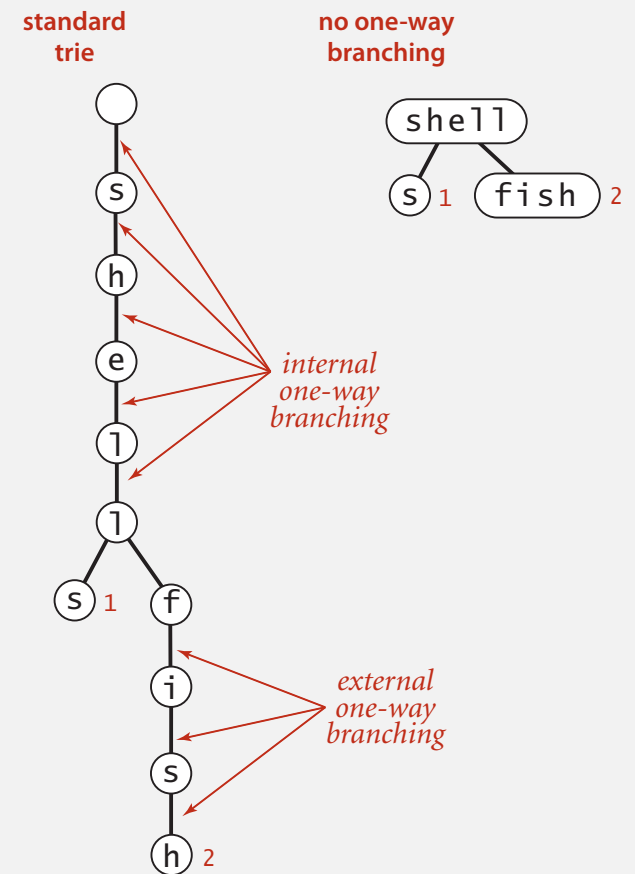
**Patricia trie.** [Practical Algorithm to Retrieve Information Coded in Alphanumeric]

- Remove one-way branching.
- Each node represents a sequence of characters.
- Implementation: one step beyond this course.

```
put("shells", 1);  
put("shellfish", 2);
```

## 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.

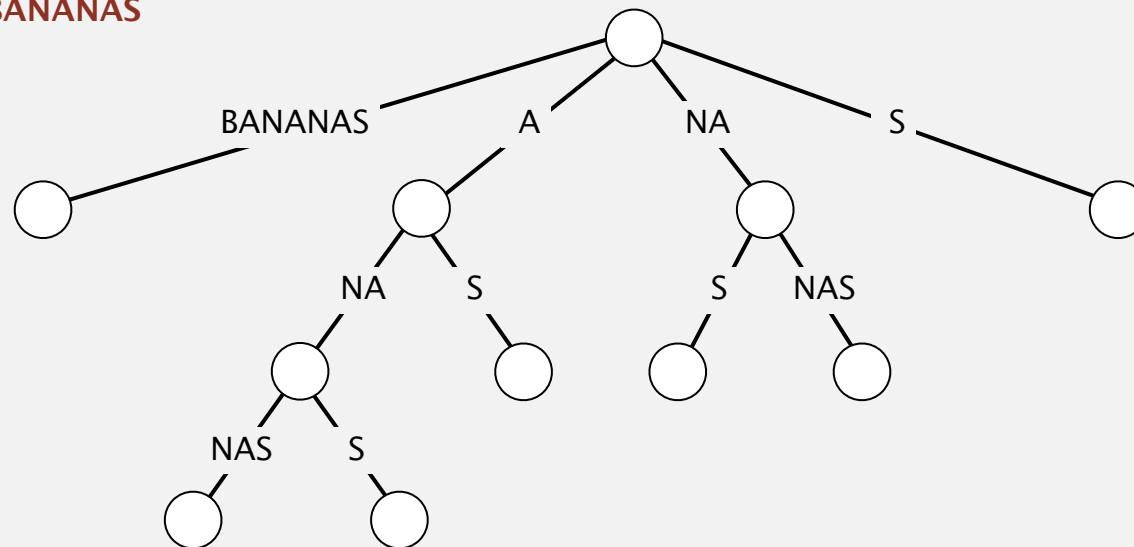
# Suffix tree

---

## Suffix tree.

- Patricia trie of suffixes of a string.
- Linear-time construction: beyond 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.

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



## 5.2 TRIES

---

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





<http://algs4.cs.princeton.edu>

## 5.2 TRIES

---

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