



<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	$\log N$	$\log N$	$\log N$	✓	compareTo()
hash table	1^\dagger	1^\dagger	1^\dagger		equals() hashCode()

† 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

Task. Deduplication: finding distinct words.

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

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.



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

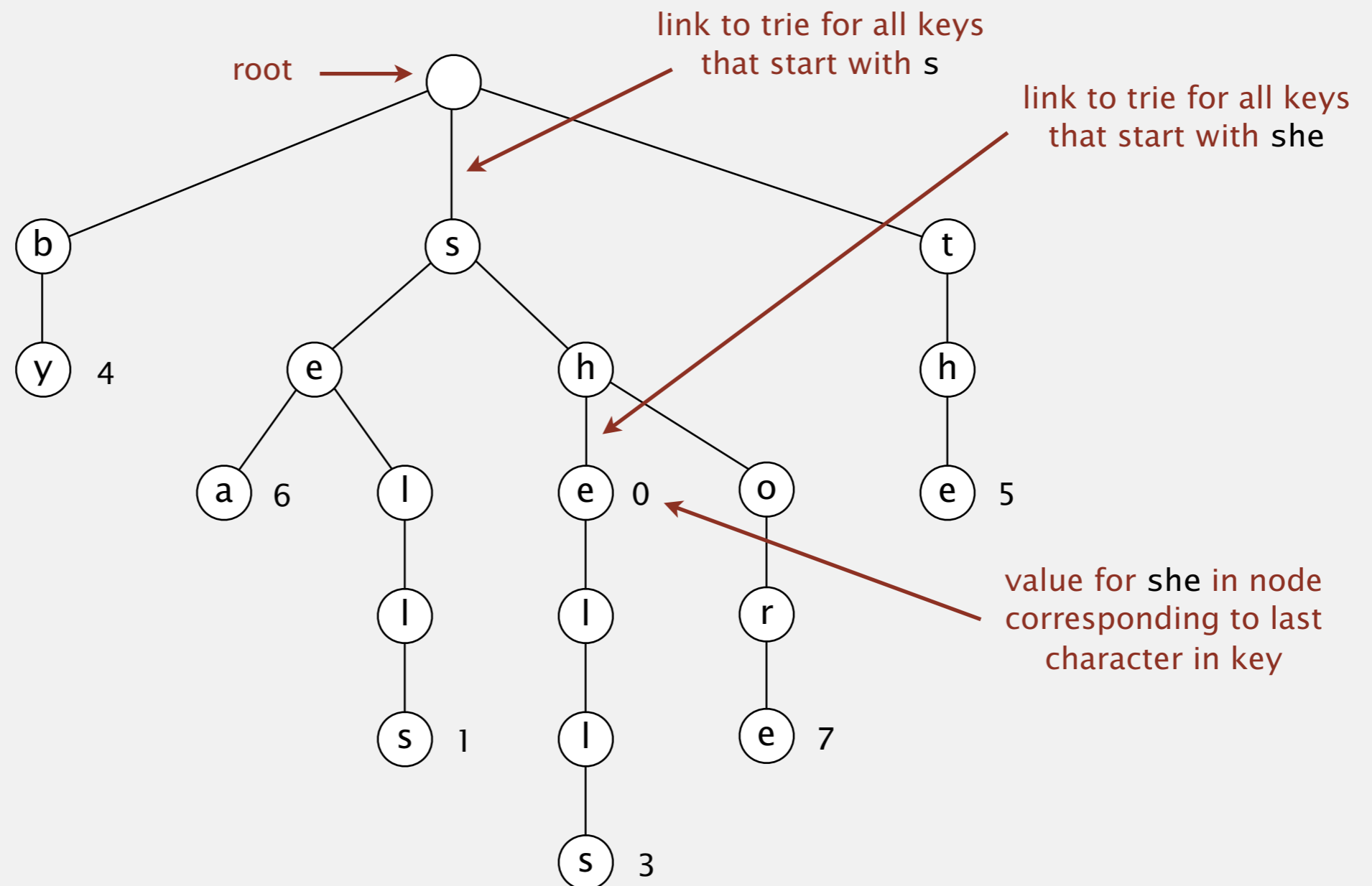
5.2 TRIES

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

Tries

Tries. [from re**trie**val, but we'll pronounce it "try"]

- Store keys' **characters** in nodes (not full strings).
- Each node has R children, one for each possible character.
(for now, we do not draw null links)



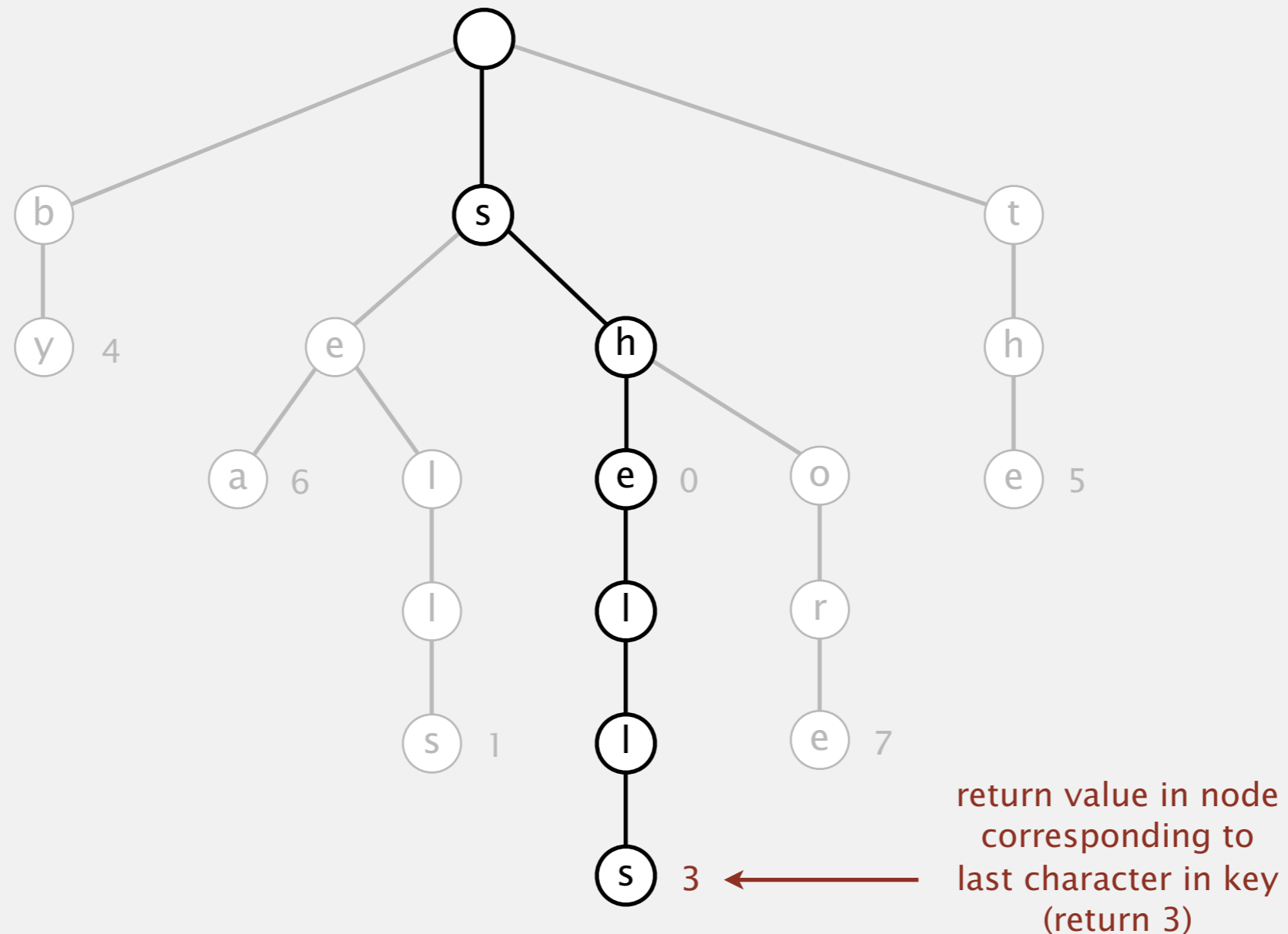
key	value
by	4
sea	6
sells	1
she	0
shells	3
shore	7
the	5

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")

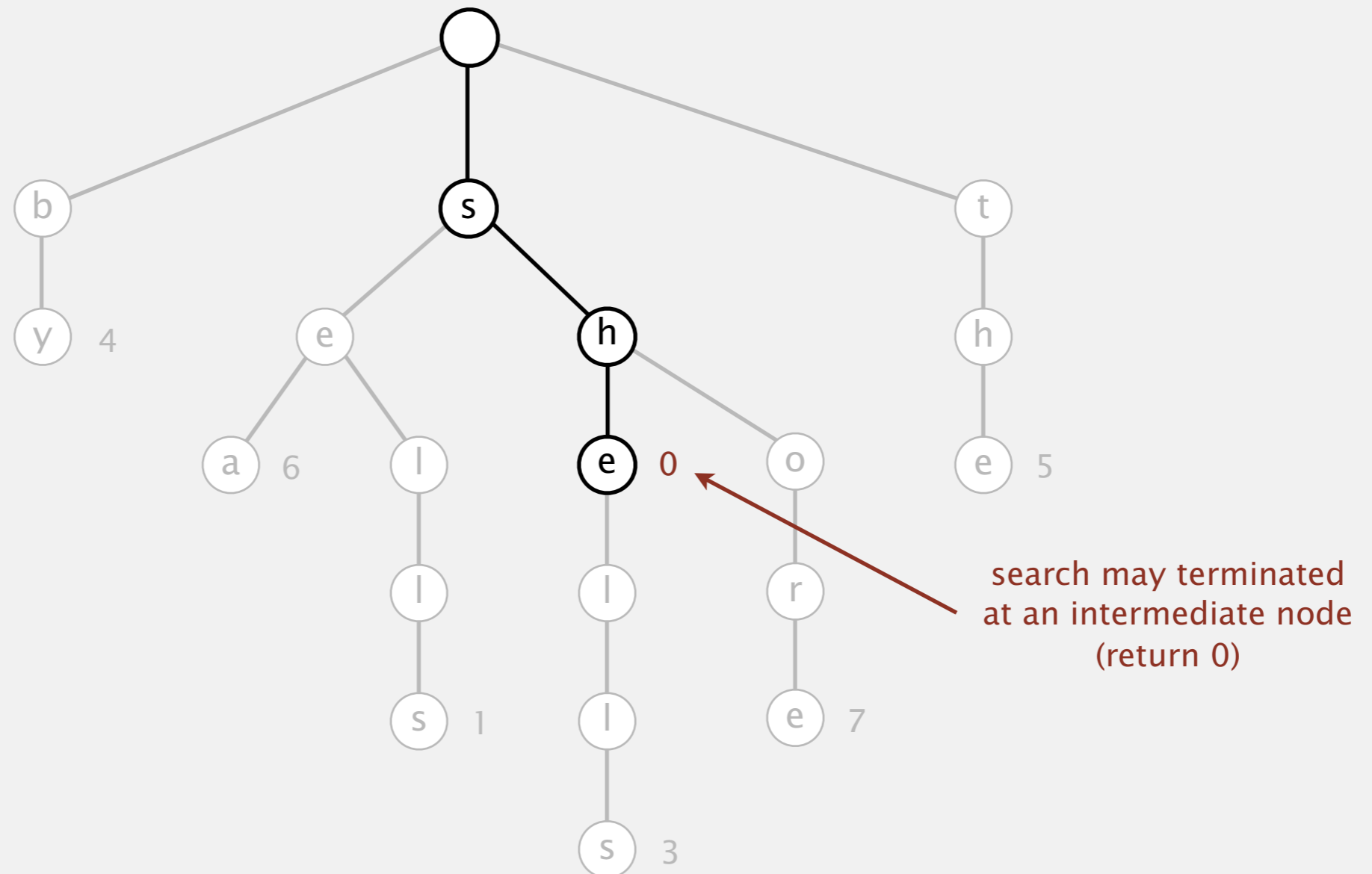


Search in a trie

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")

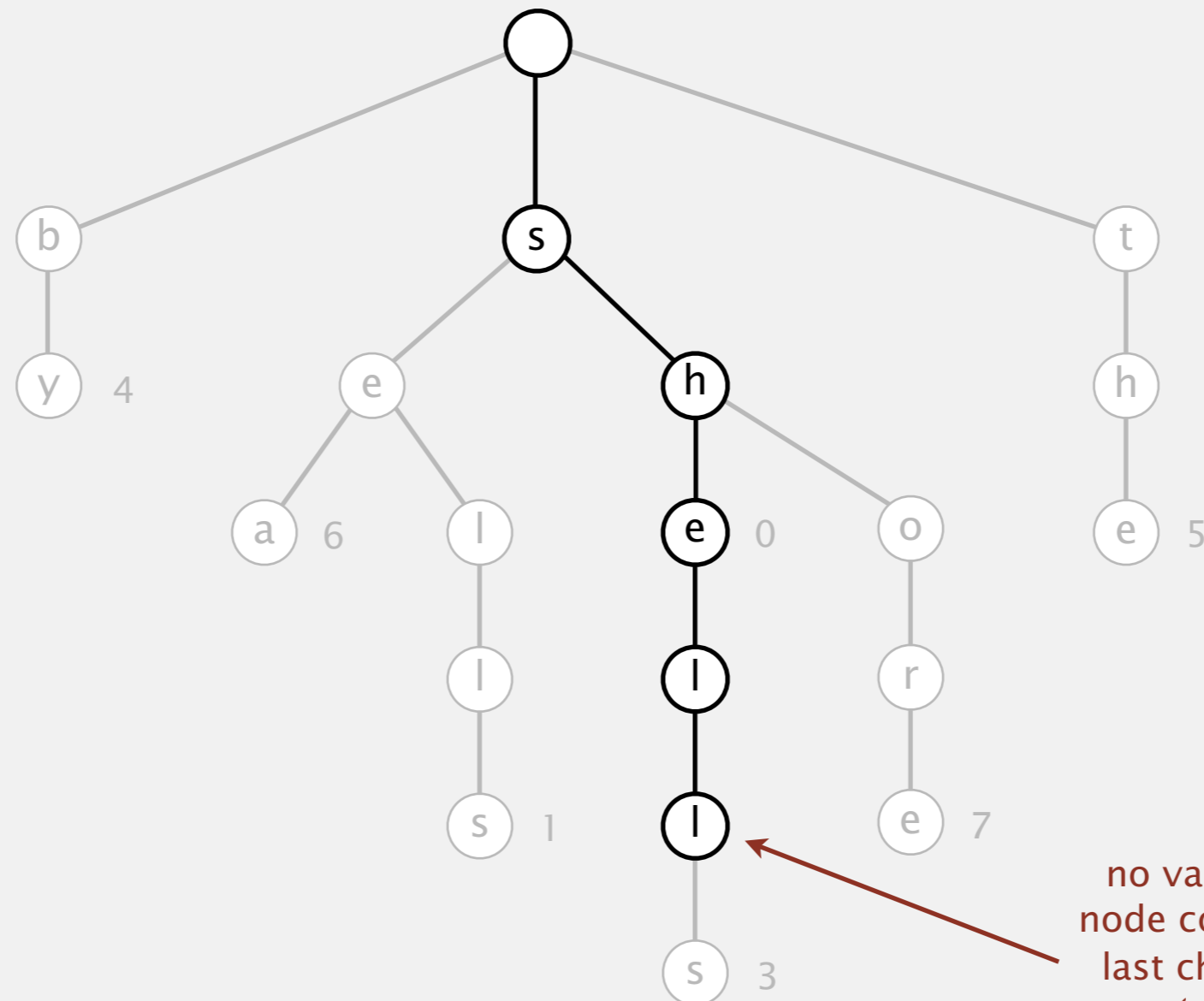


Search in a trie

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")



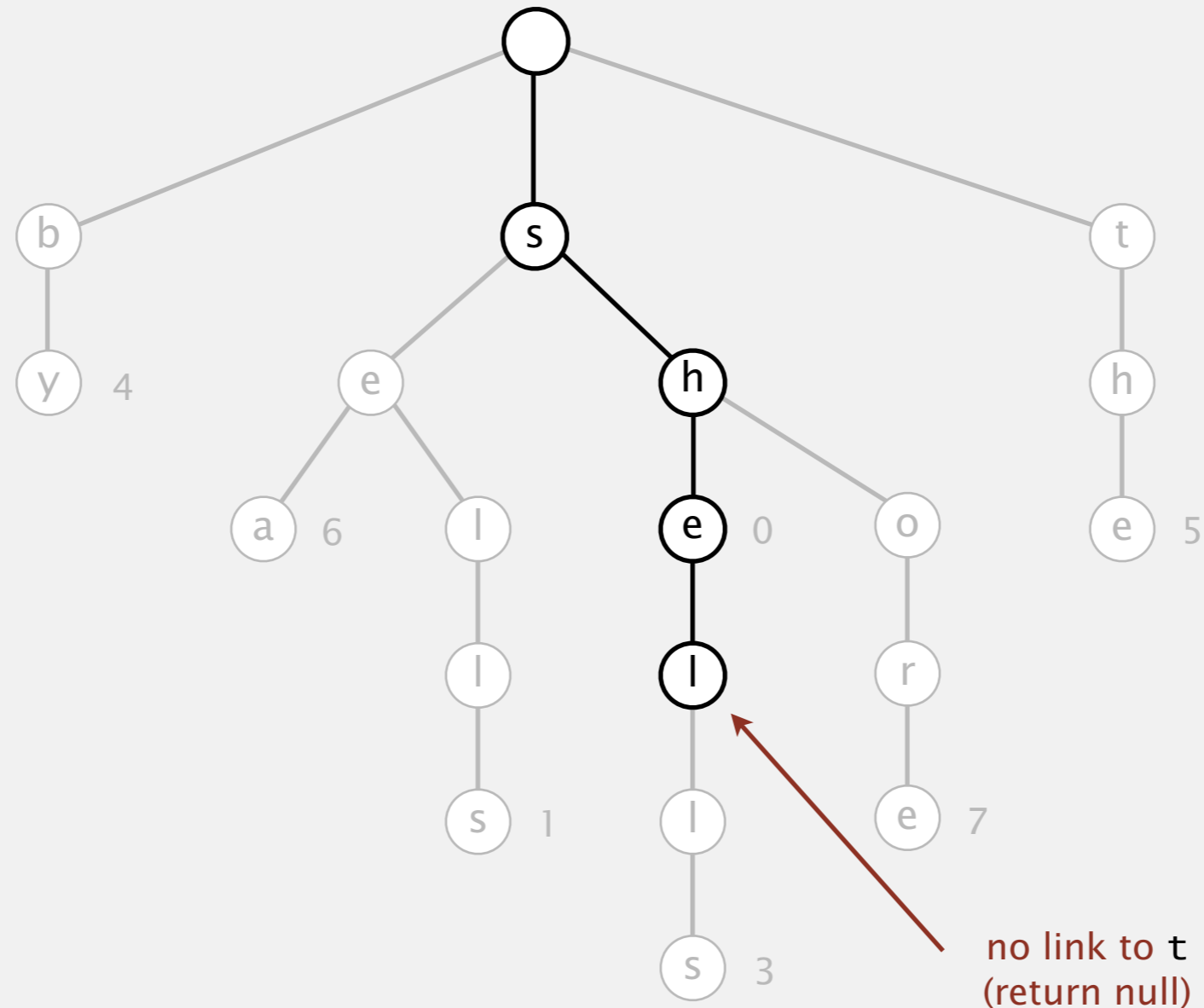
no value associated
node corresponding to
last character in key
(return null)

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")

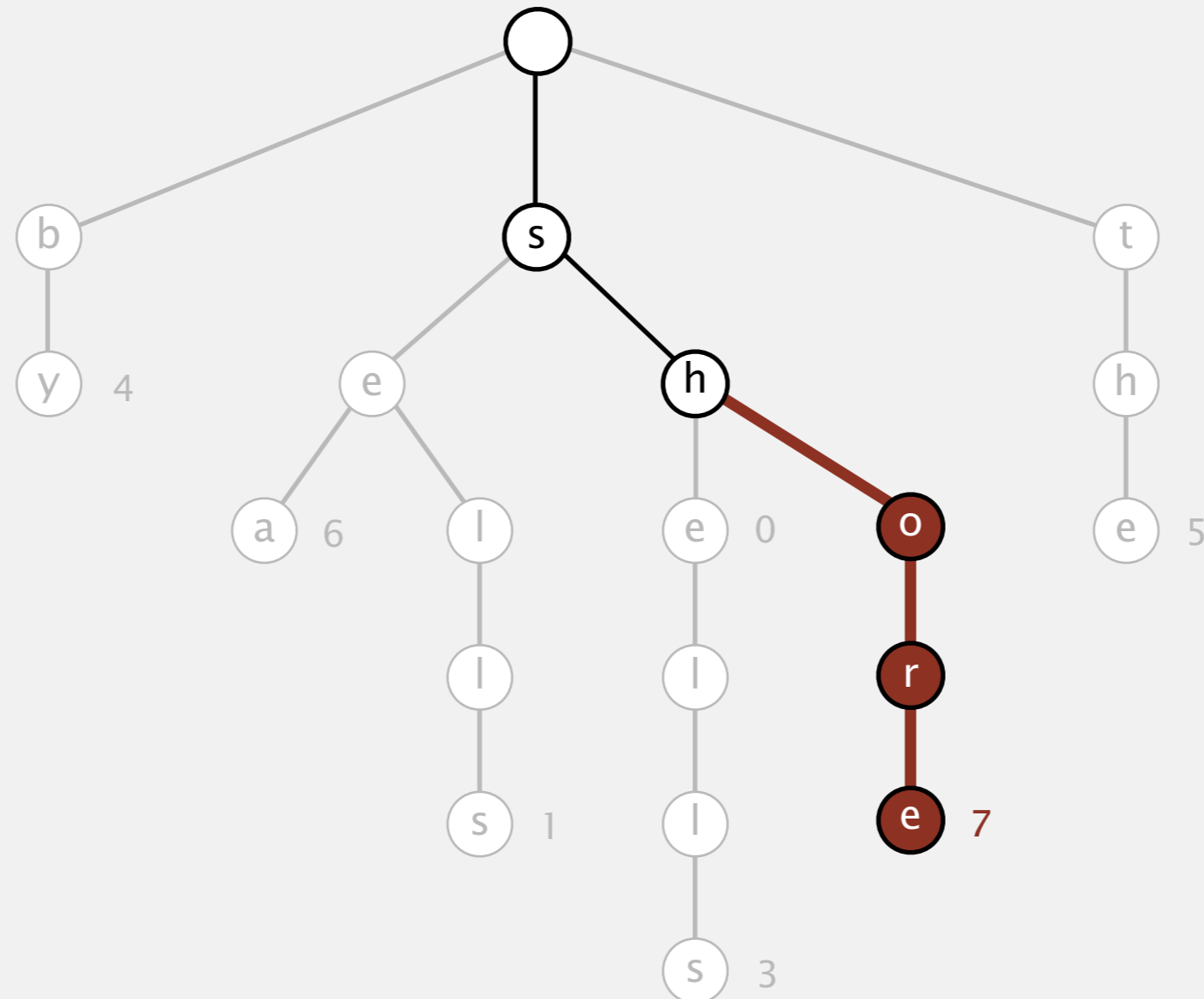


Insertion into a trie

Follow links corresponding to each character in the key.

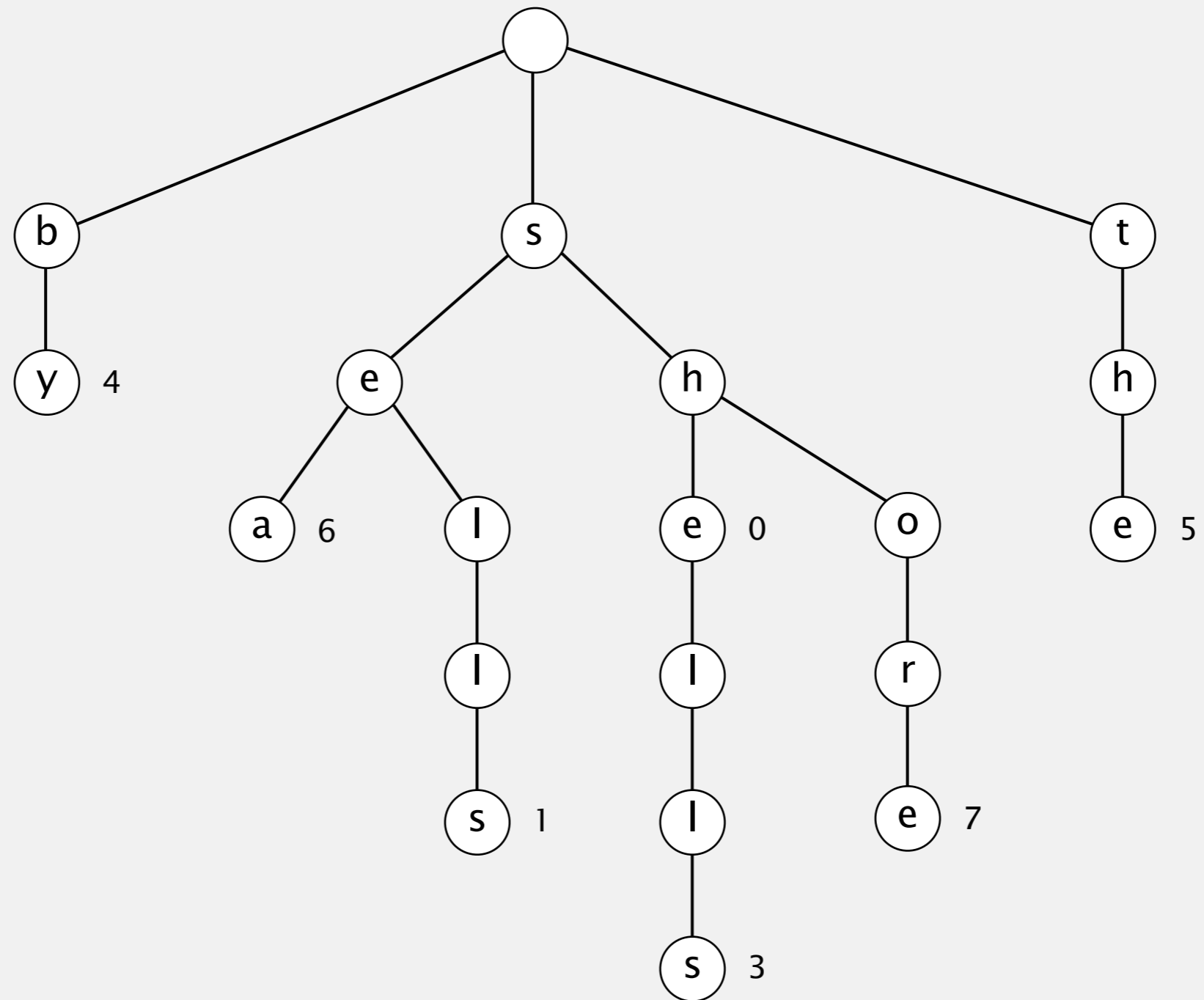
- Encounter a null link: create new node.
- Encounter the last character of the key: set value in that node.

`put("shore", 7)`



Trie construction demo

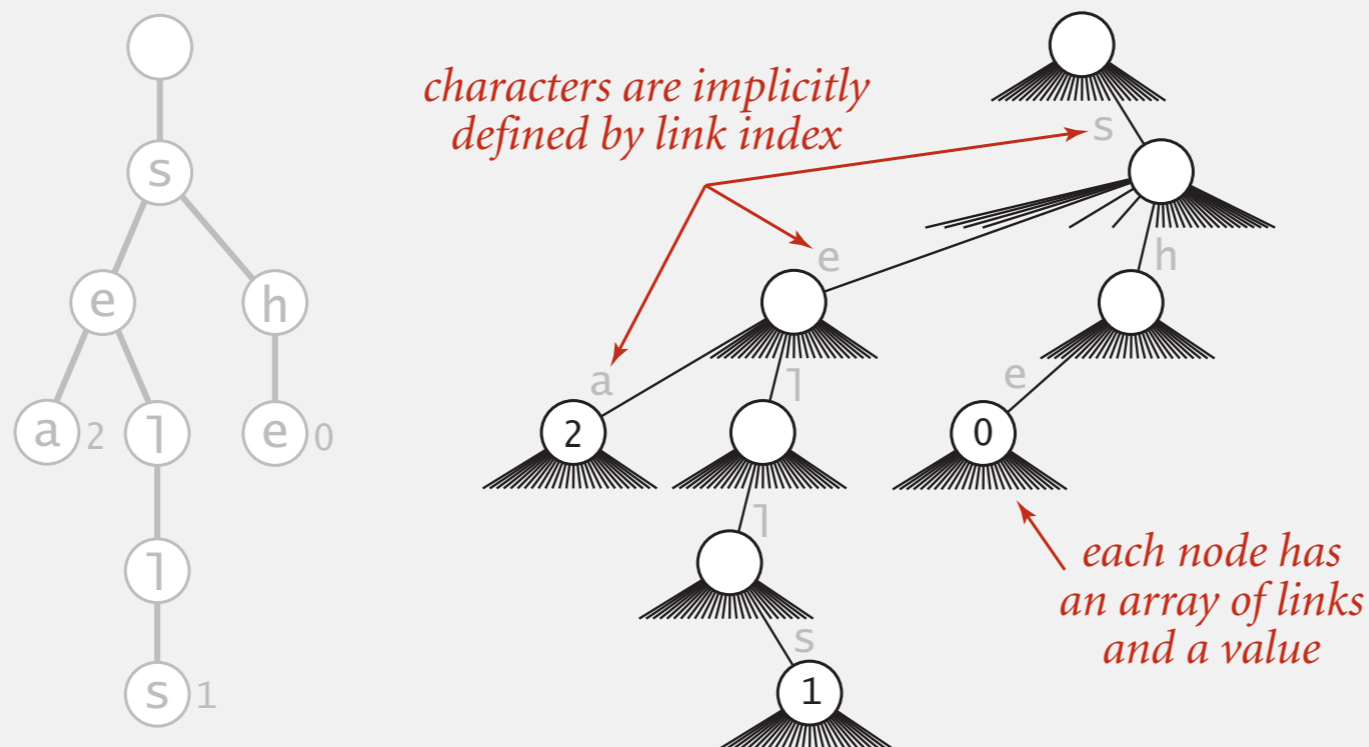
trie



Trie representation: Java implementation

Node. A value, plus references to R nodes.

```
private static class Node
{
    private Object val; // no generic array creation
    private Node[] next = new Node[R];
}
```



Remark. Neither keys nor characters are stored explicitly.

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 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 quiz 1

What is order of growth of the running time (in the worst case) to insert a key into an R -way trie, as a function of the number of strings N , the length L of the key, and the alphabet size R ?

- A. L
- B. $R + L$
- C. $N + L$
- D. $R L$
- E. *I don't know.*

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	$\log_R N$	$R + L$	$(R+1)N$	1.12	<i>out of memory</i>

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 2^{16} -way trie for Unicode!



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

5.2 TRIES

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

Ternary search tries

- Store keys' characters (not full strings) and values in nodes.
- Each node has **3** children: smaller (left), equal (middle), larger (right).
- MSD radix sort + 3-way quicksort = 3-way radix quicksort
R-way tries + Red-Black search trees = **TSTs**

Fast Algorithms for Sorting and Searching Strings

Jon L. Bentley*

Robert Sedgwick#

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.

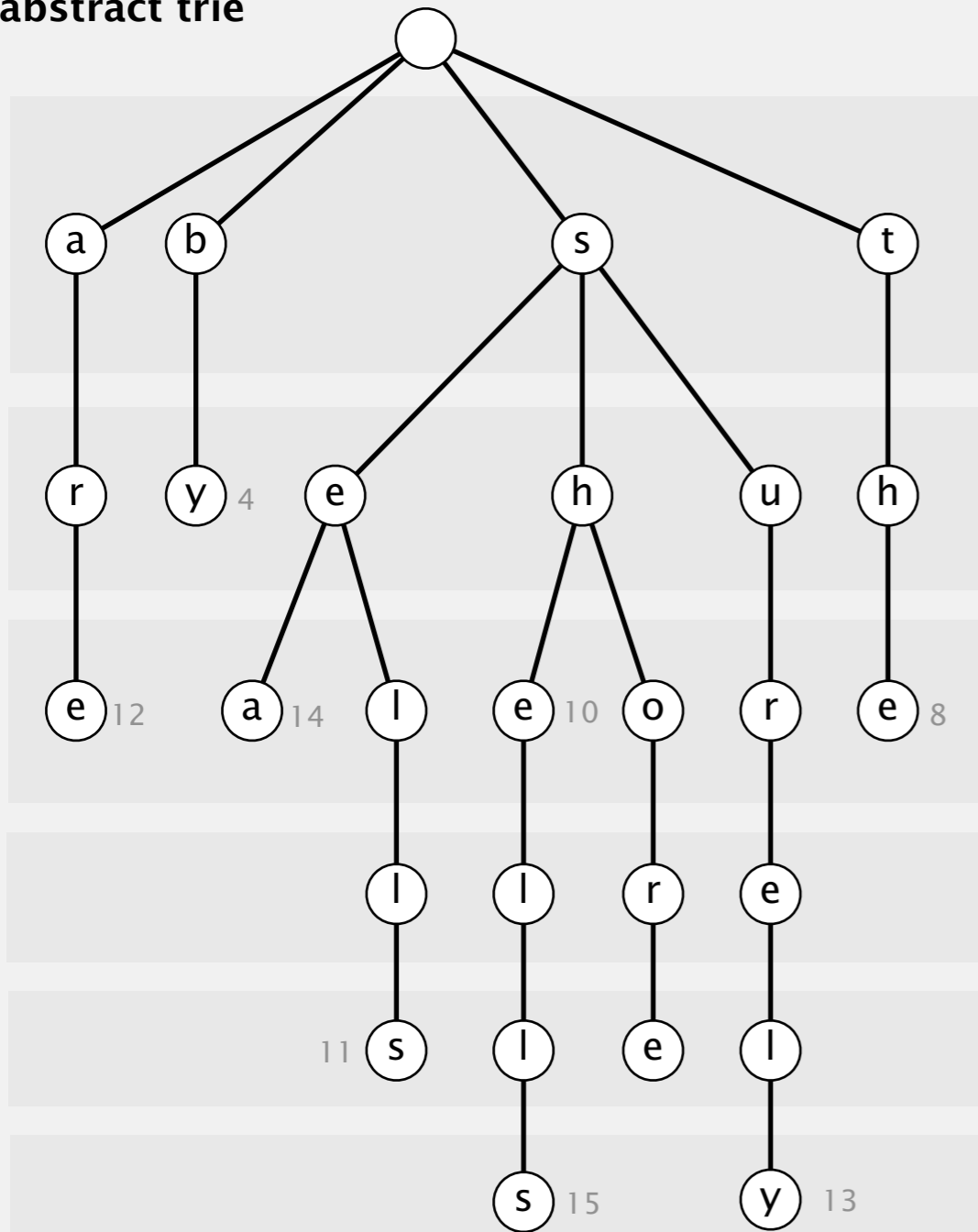
In many application programs, sorts use a Quicksort implementation based on an abstract compare operation,



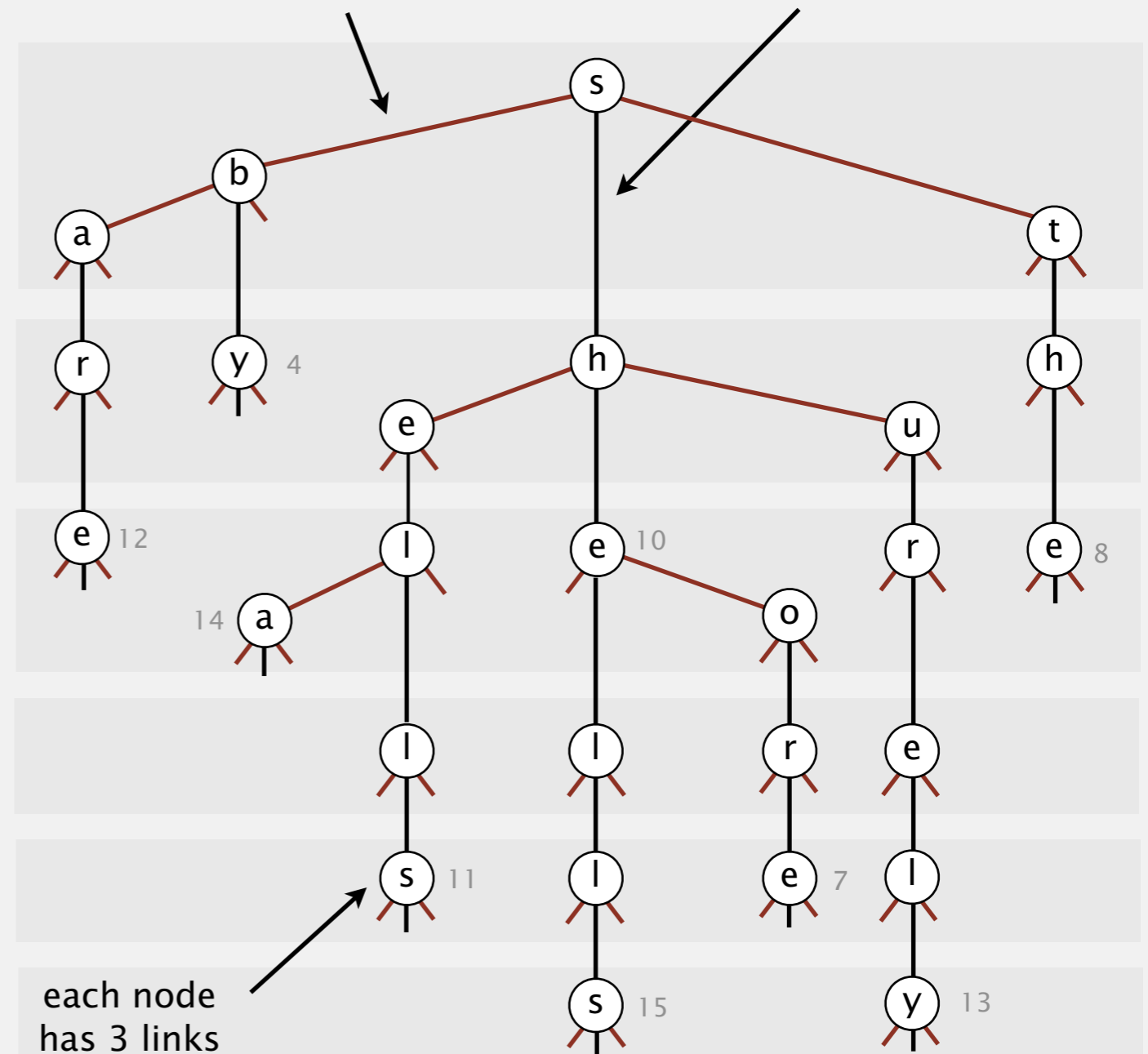
Ternary search tries

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

abstract trie



TST link to TST for all keys that start with a character less than s link to TST for all keys that start with s

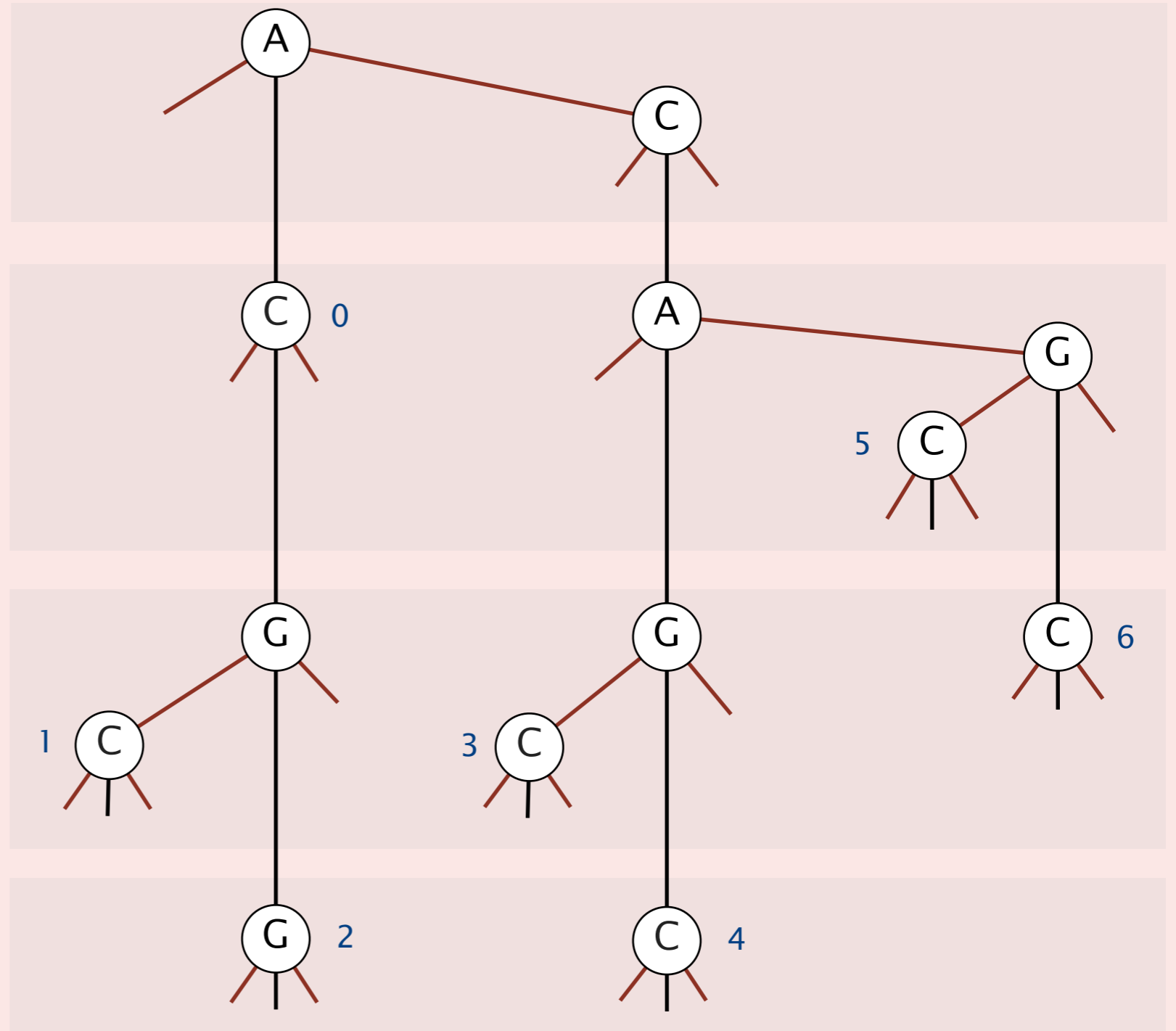


each node has 3 links

Trie quiz 2

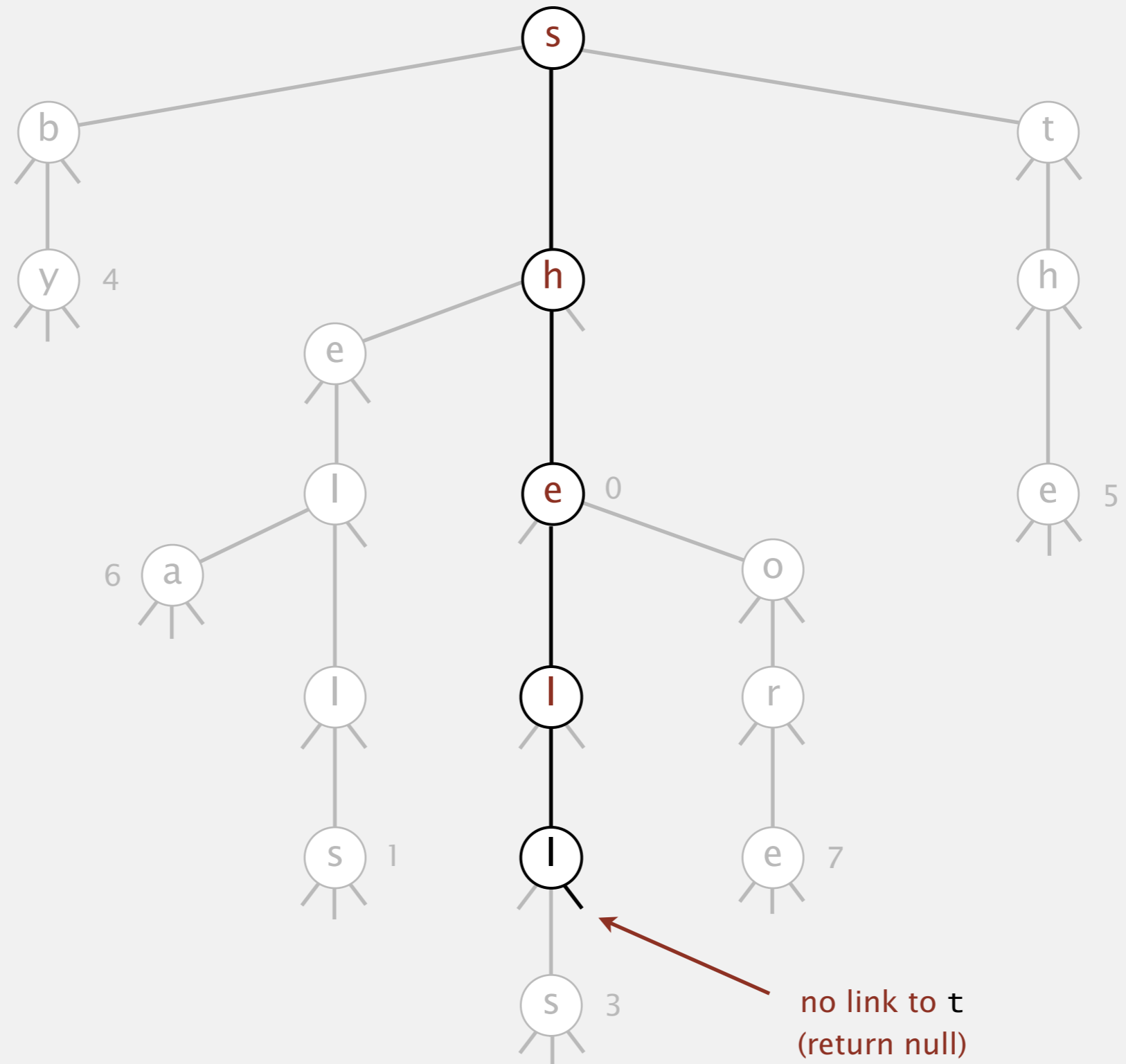
Which value is associated with the string key CAC ?

- A. 3
- B. 4
- C. 5
- D. *null*
- E. *I don't know.*



Search miss in a TST

get("shelter")



Trie quiz e

In which subtrie would the key CAC be inserted?

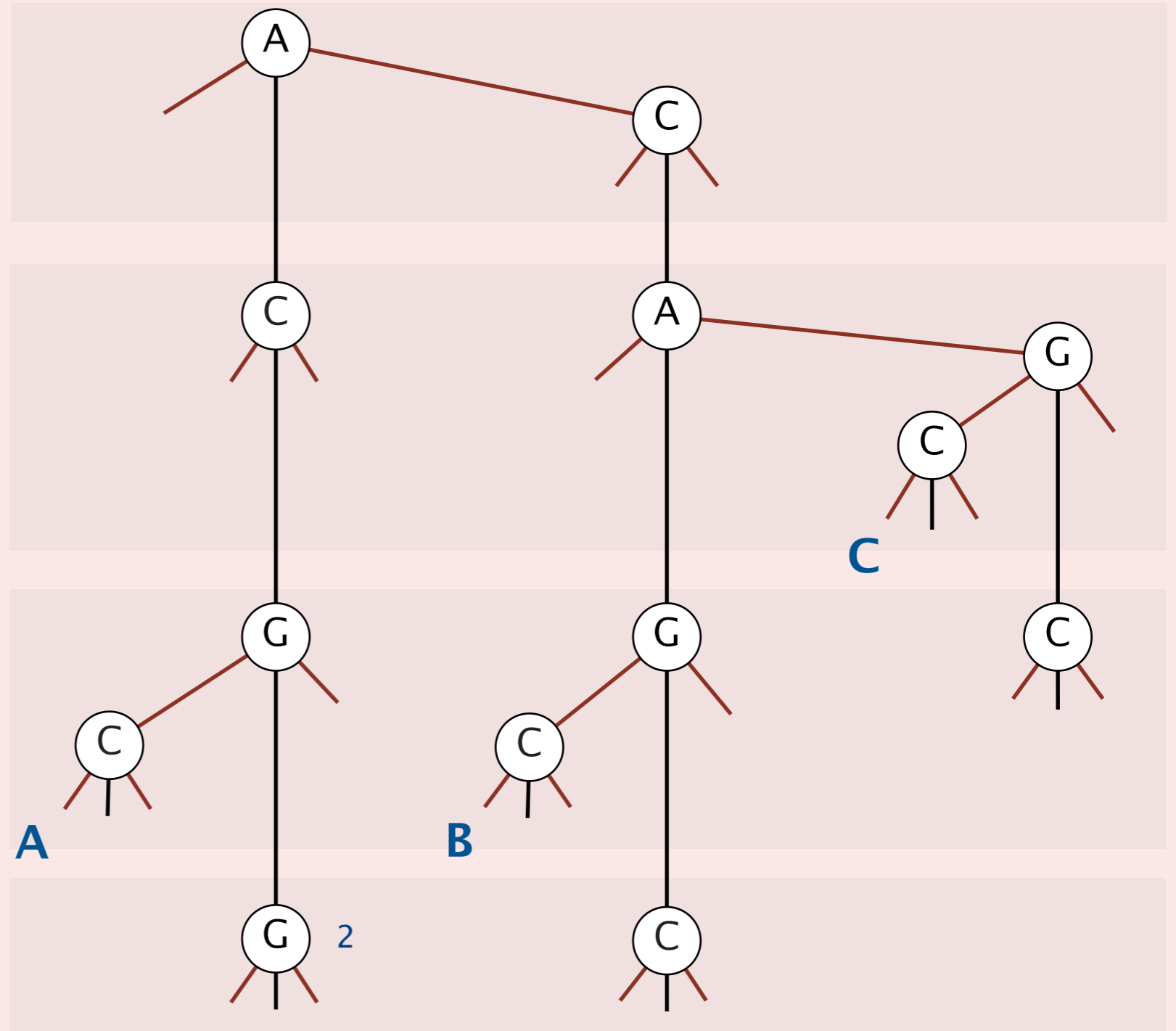
A.

B.

C.

D. *None of the above.*

E. *I don't know.*

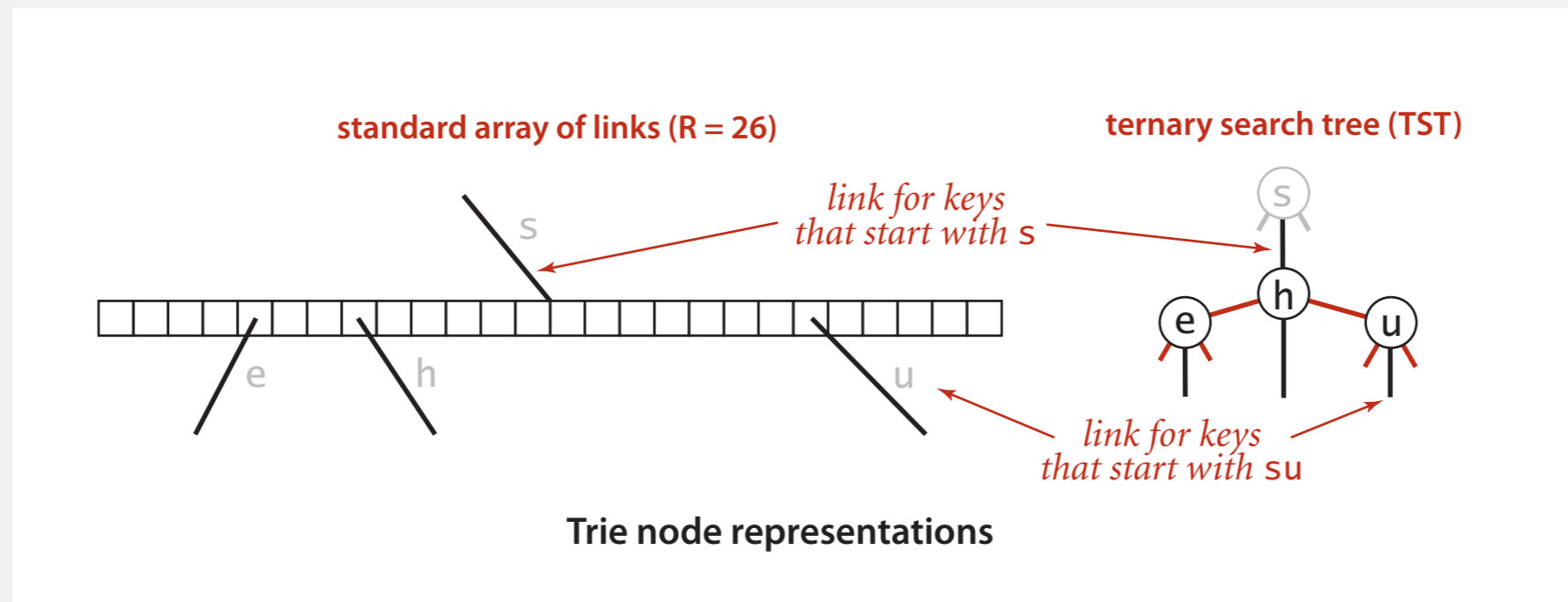


TST representation in Java

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 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;
    }

    public void put(String Key, Value val)
    { /* similar, see book or booksite */ }
}
```


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	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	$4N$	1.40	97.4
hashing (linear probing)	L	L	L	$4N$ to $16N$	0.76	40.6
R-way trie	L	$\log_R N$	$R + L$	$(R+1)N$	1.12	<i>out of memory</i>
TST	$L + \ln N$	$\ln N$	$L + \ln N$	$4N$	0.72	38.7

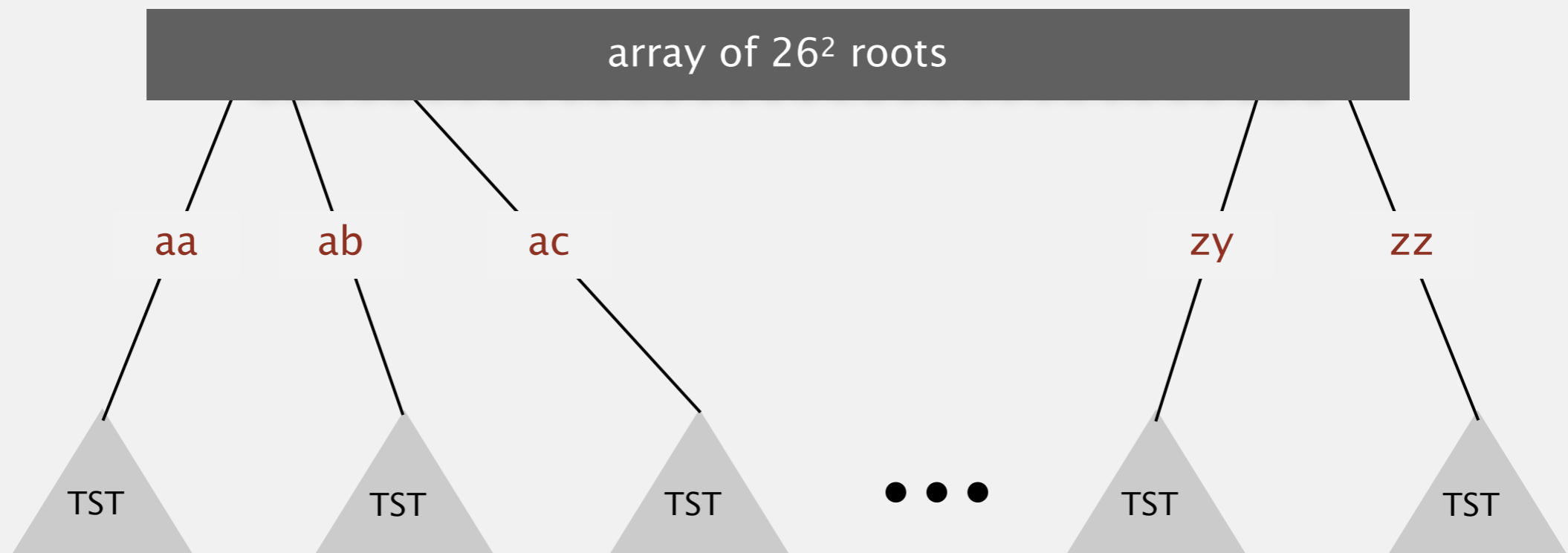
Remark. Can build balanced TSTs via rotations at each level (character) to achieve $L + \log N$ worst-case guarantees.

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	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	$4N$	1.40	97.4
hashing (linear probing)	L	L	L	$4N$ to $16N$	0.76	40.6
R-way trie	L	$\log_R N$	$R + L$	$(R+1) N$	1.12	<i>out of memory</i>
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 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]



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

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
se11s	1
she	0
she11s	3
shore	7
the	5

Prefix match. Keys with prefix sh: she, she11s, and shore.

Wildcard match. Keys that match .he: she and the.

Longest prefix. Key that is the longest prefix of she11sort: she11s.

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)
```

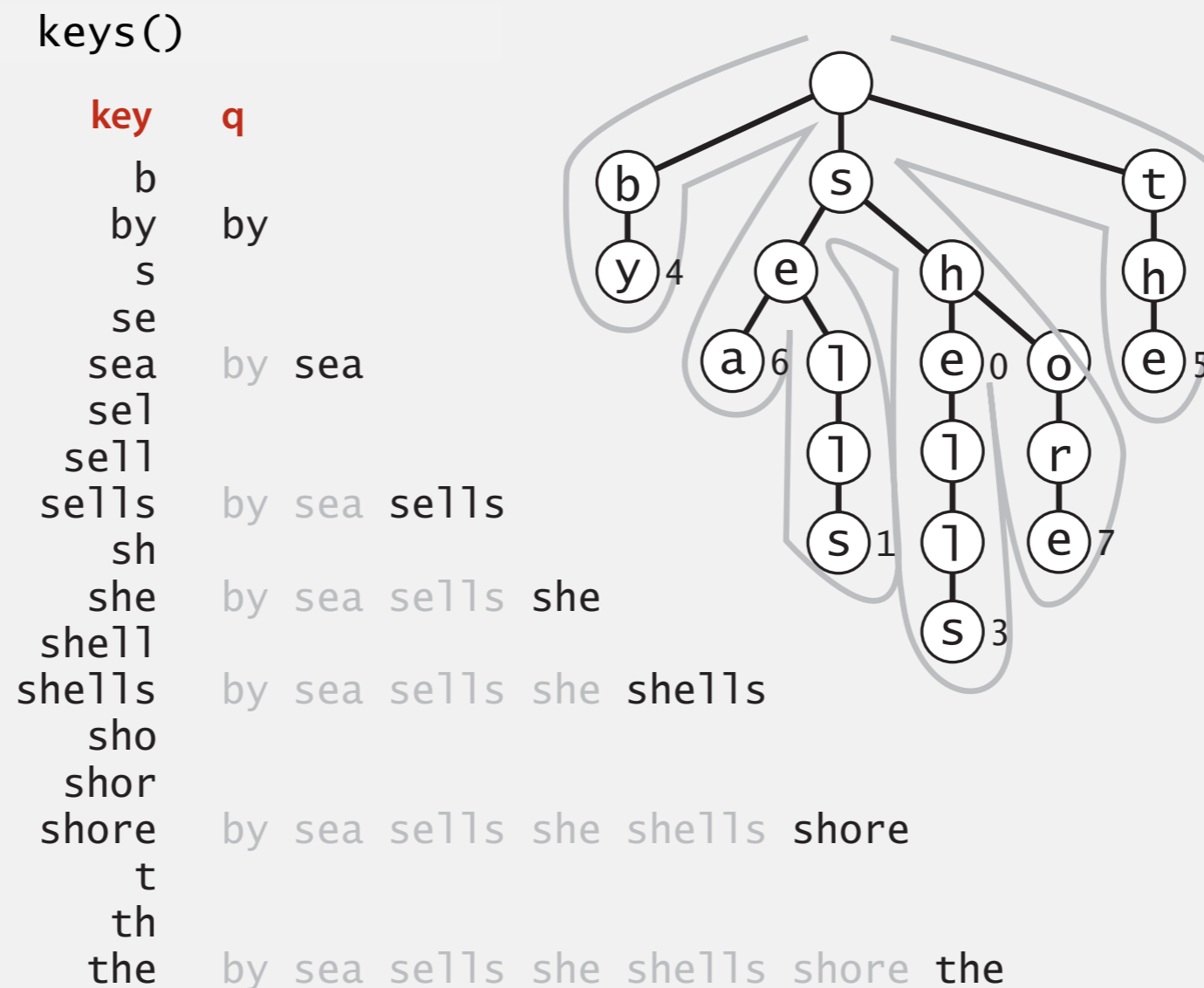
longest key that is a prefix of s

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> 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);
}
```

sequence of characters
on path from root to x

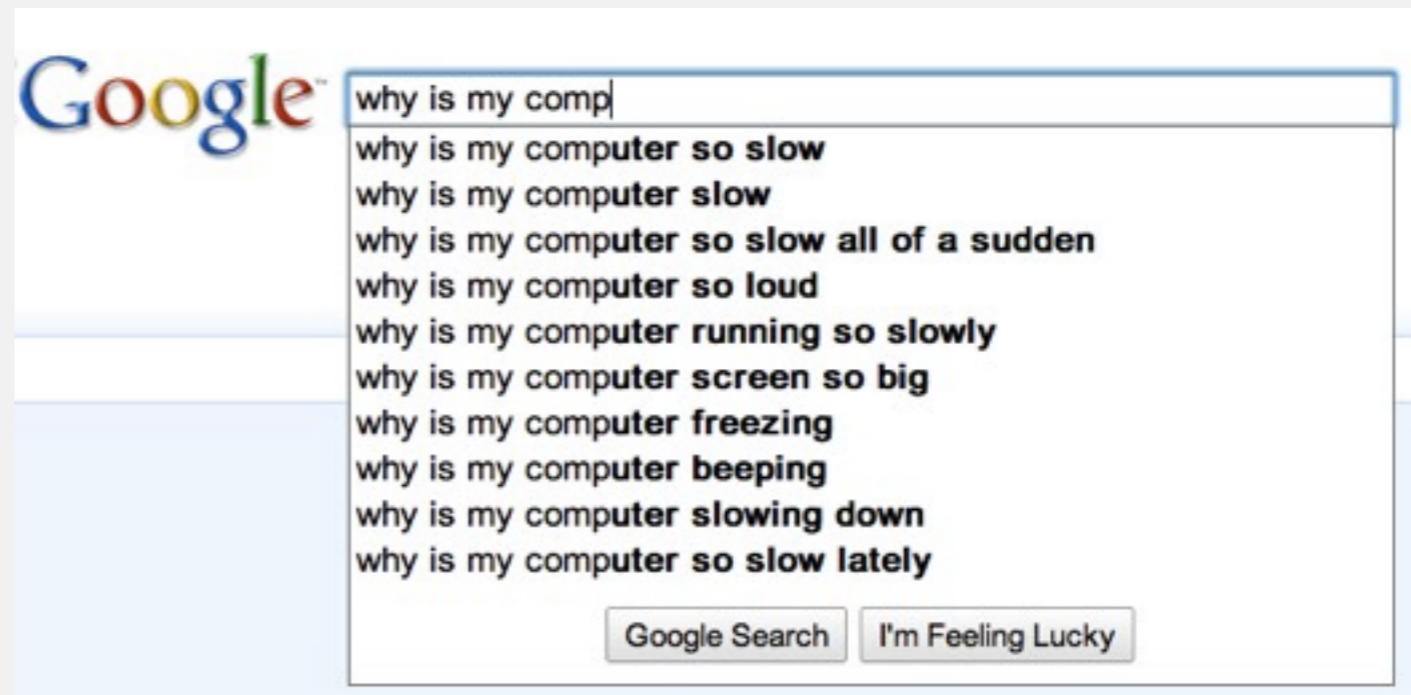
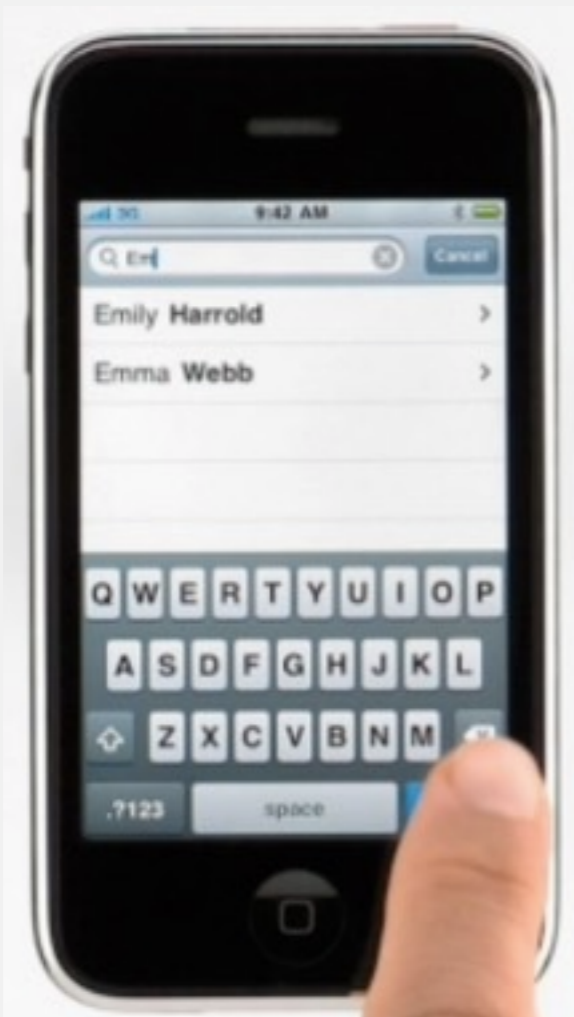
or use `StringBuilder`

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.

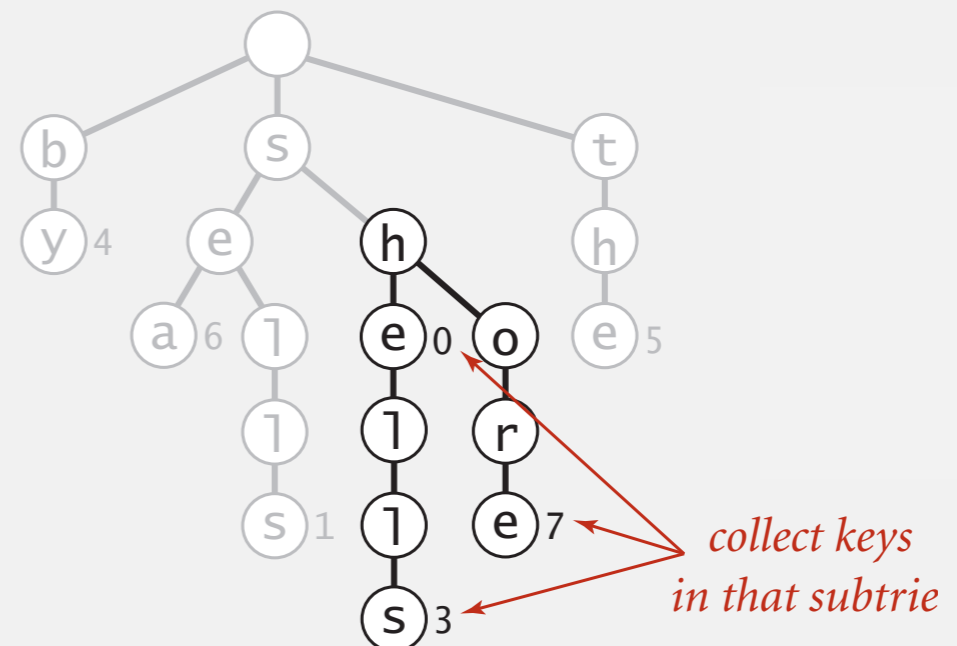
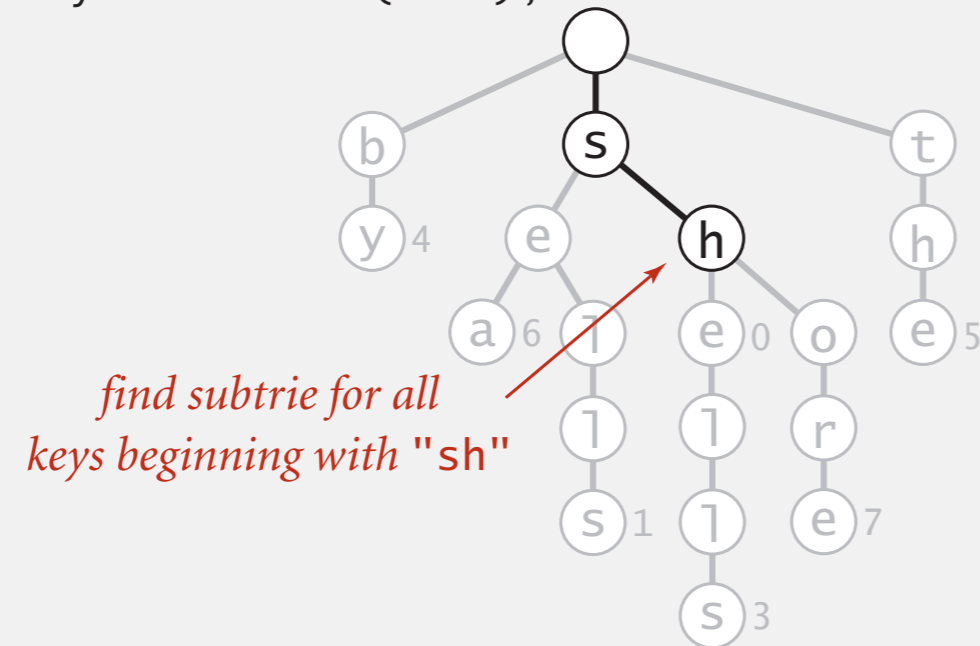
- User types characters one at a time.
- System reports all matching strings.



Prefix matches in an R-way trie

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

keysWithPrefix("sh");



```
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 subtrie for all strings beginning with given prefix

key	queue
sh	
she	she
shell	
shell	
shells	she shells
sho	
shor	
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
"128.112" ← binary number for IPv4
"128.112.055" ← (instead of string)
"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: 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 (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

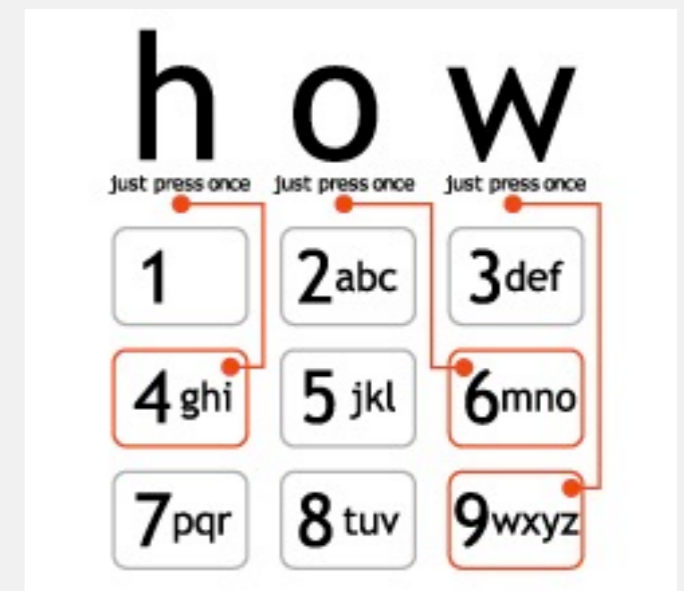
"a much faster and more fun way to enter text"

T9 text input.

- Find all words that correspond to given sequence of numbers.

4663: good, home, gone, hoof. ← *textonyms*

- Press * to select next option.
- Press 0 to see all completion options.
- System adapts to user's tendencies.



<http://www.t9.com>

T9 TEXTING

Q. How to implement T9 texting on a mobile phone?

SONY



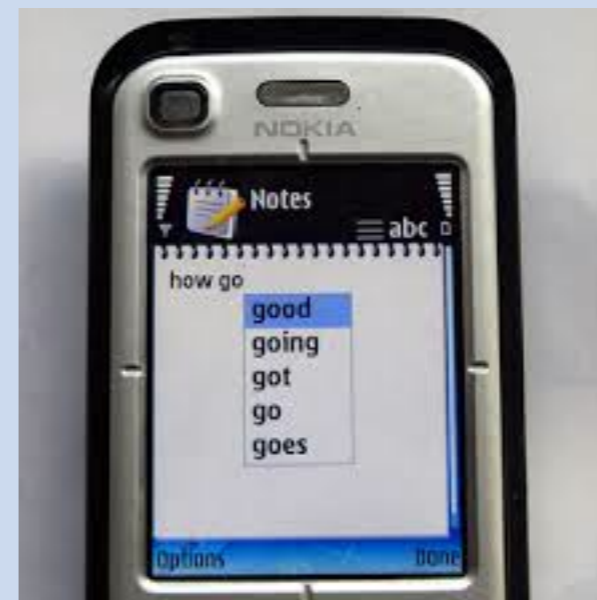
SIEMENS

NEC

SANYO



1	2 ABC	3 DEF	-
4 GHI	5 JKL	6 MNO	.
7 PRQS	8 TUV	9 WXYZ	DEL X
* # (0 +	⏪	Next



Patricia trie

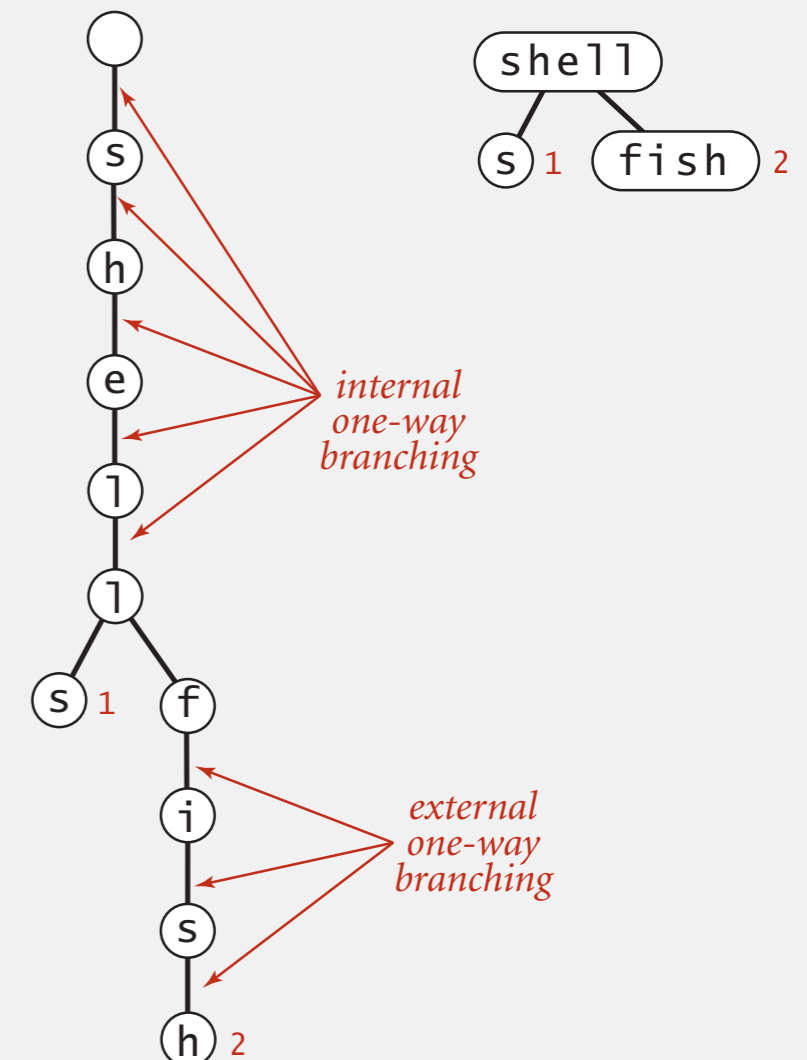
Patricia trie. [Practical Algorithm to Retrieve Information Coded in Alphanumeric]

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

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

standard
trie

no one-way
branching



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.

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.