Algorithms

FOURTH EDITION

Robert Sedgewick | Kevin Wayne

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 †	1 †	1 †		`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

| 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.

5.2 TRIES

‣ *R-way tries*

‣ *ternary search tries*

‣ *character-based operations*

Algorithms

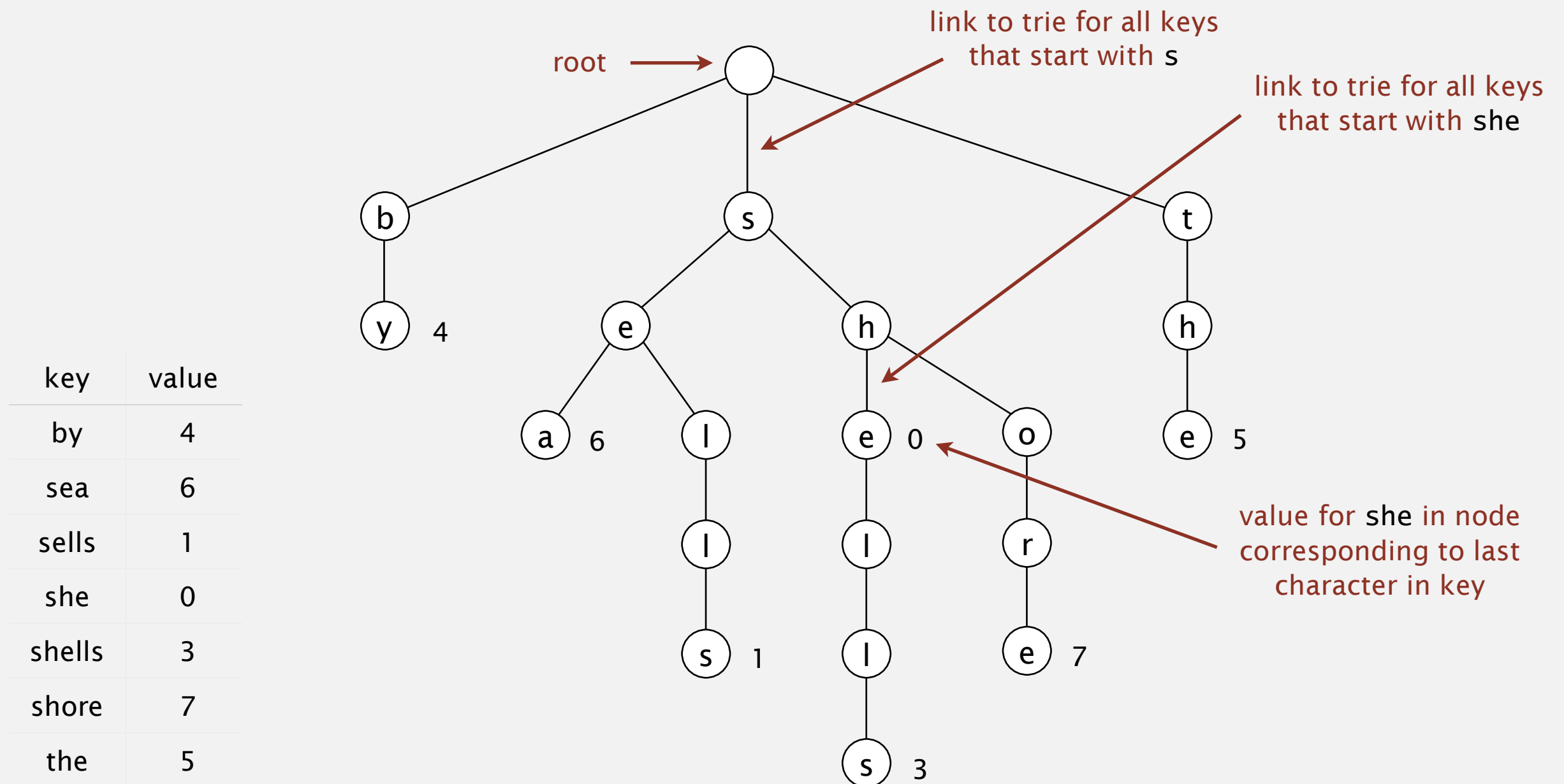ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

Tries

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)

root

link to trie for all keys that start with s

link to trie for all keys that start with she

value for she in node corresponding to last character in key

b

s

t

y 4

e

h

h

a 6

l

e 0

o

e 5

l

l

r

s 1

l

e 7

s 3

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

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)
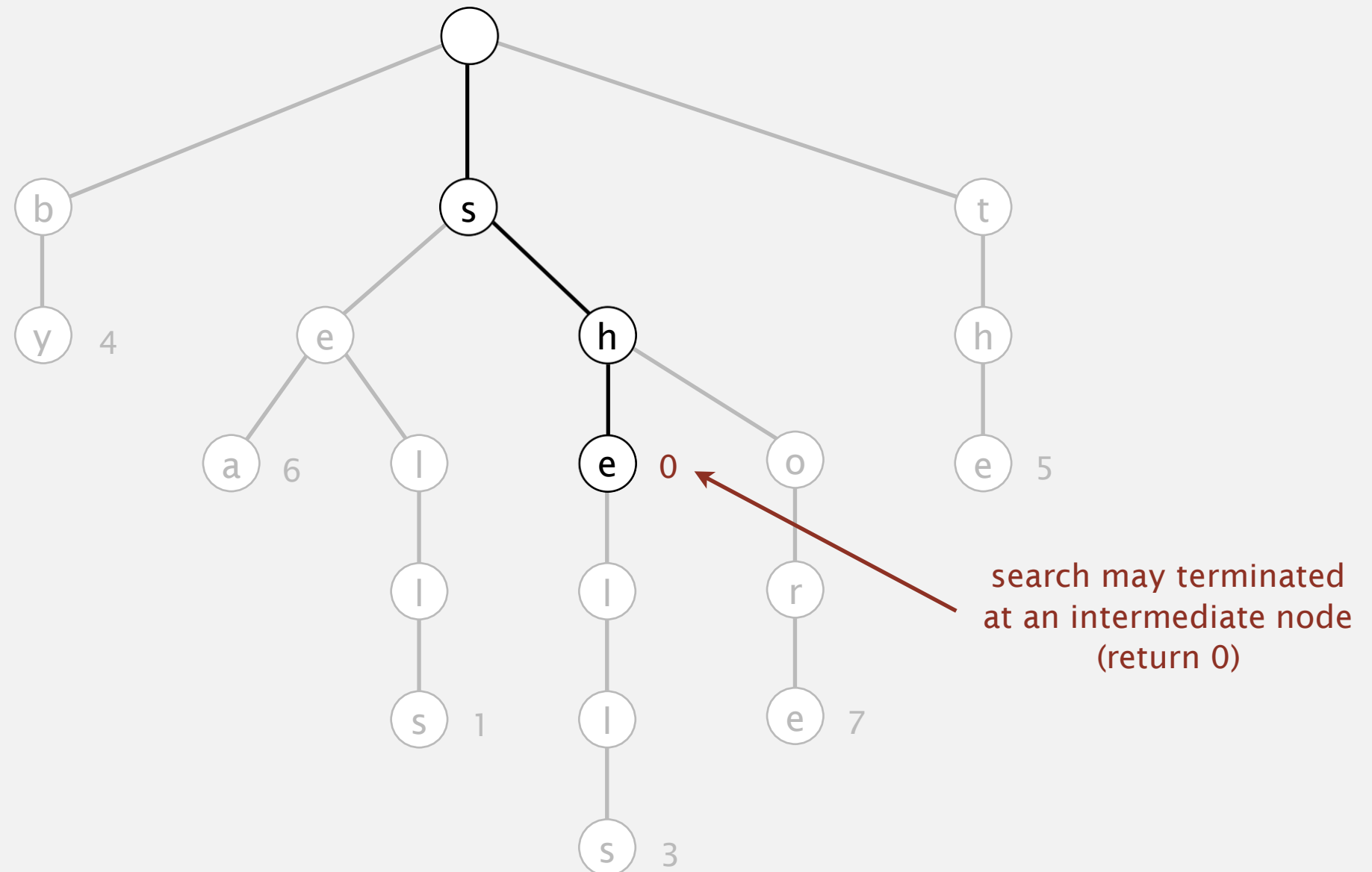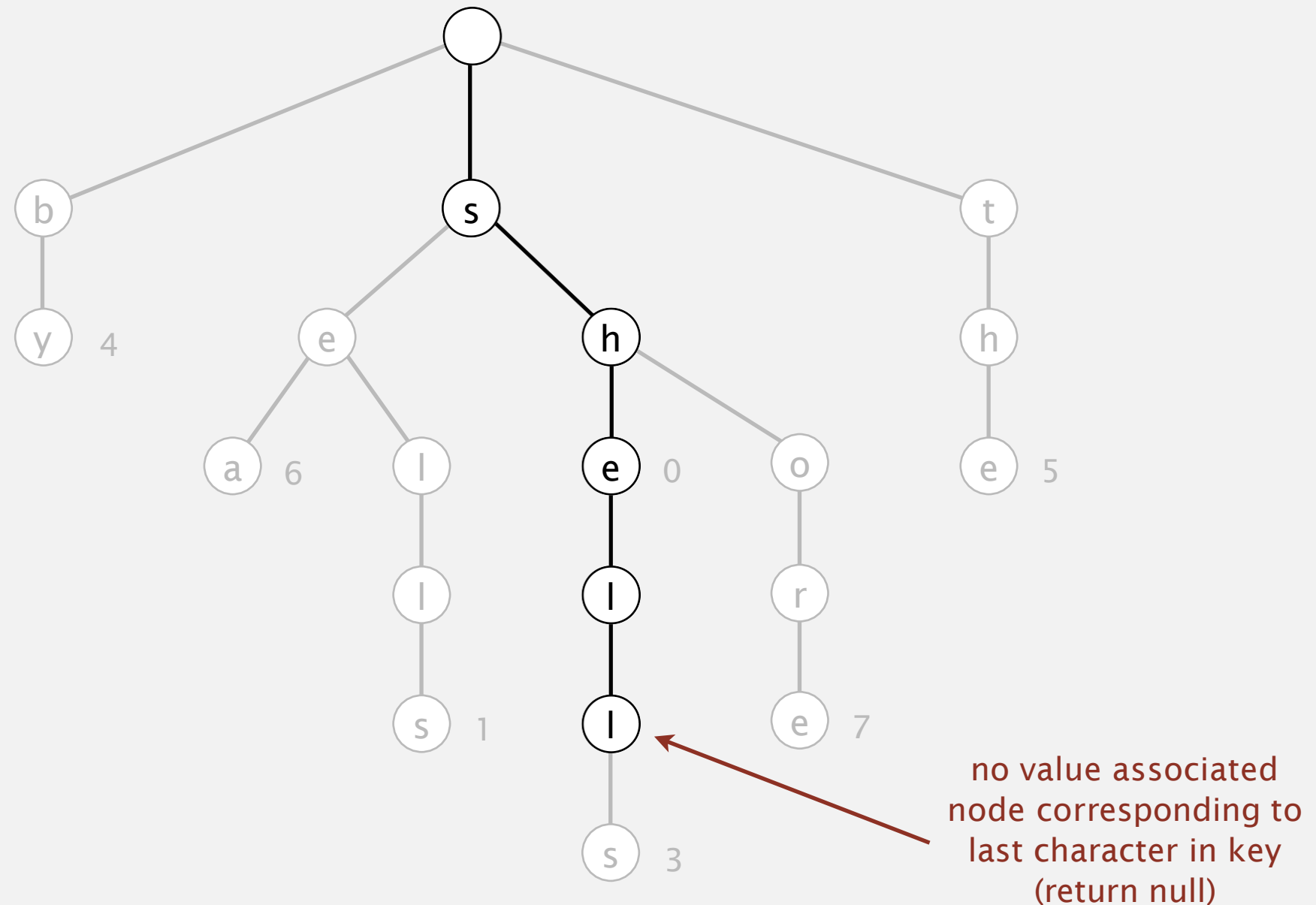
8
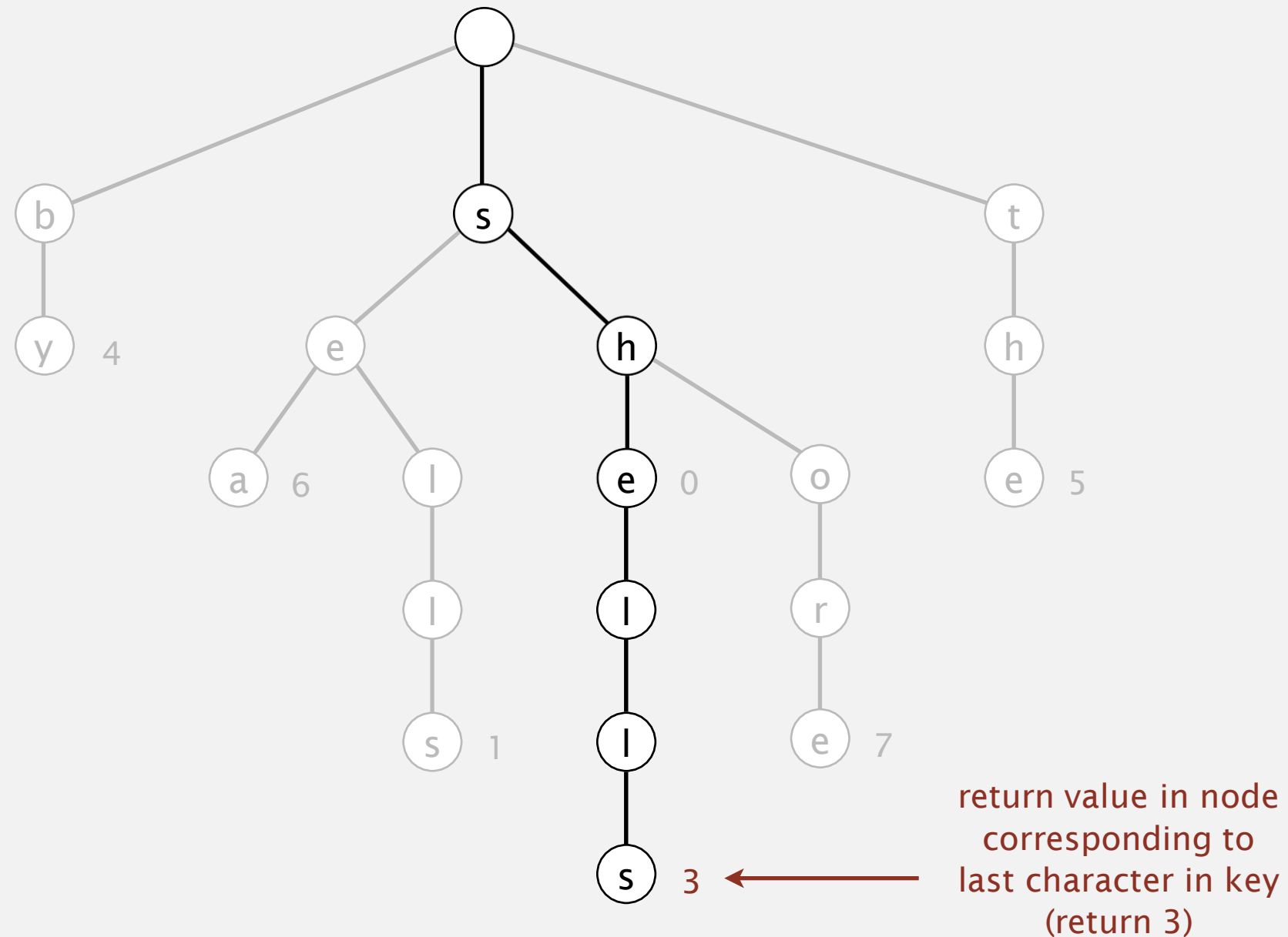
Search in a trie

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 may terminated
at an intermediate node
(return 0)

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)

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.

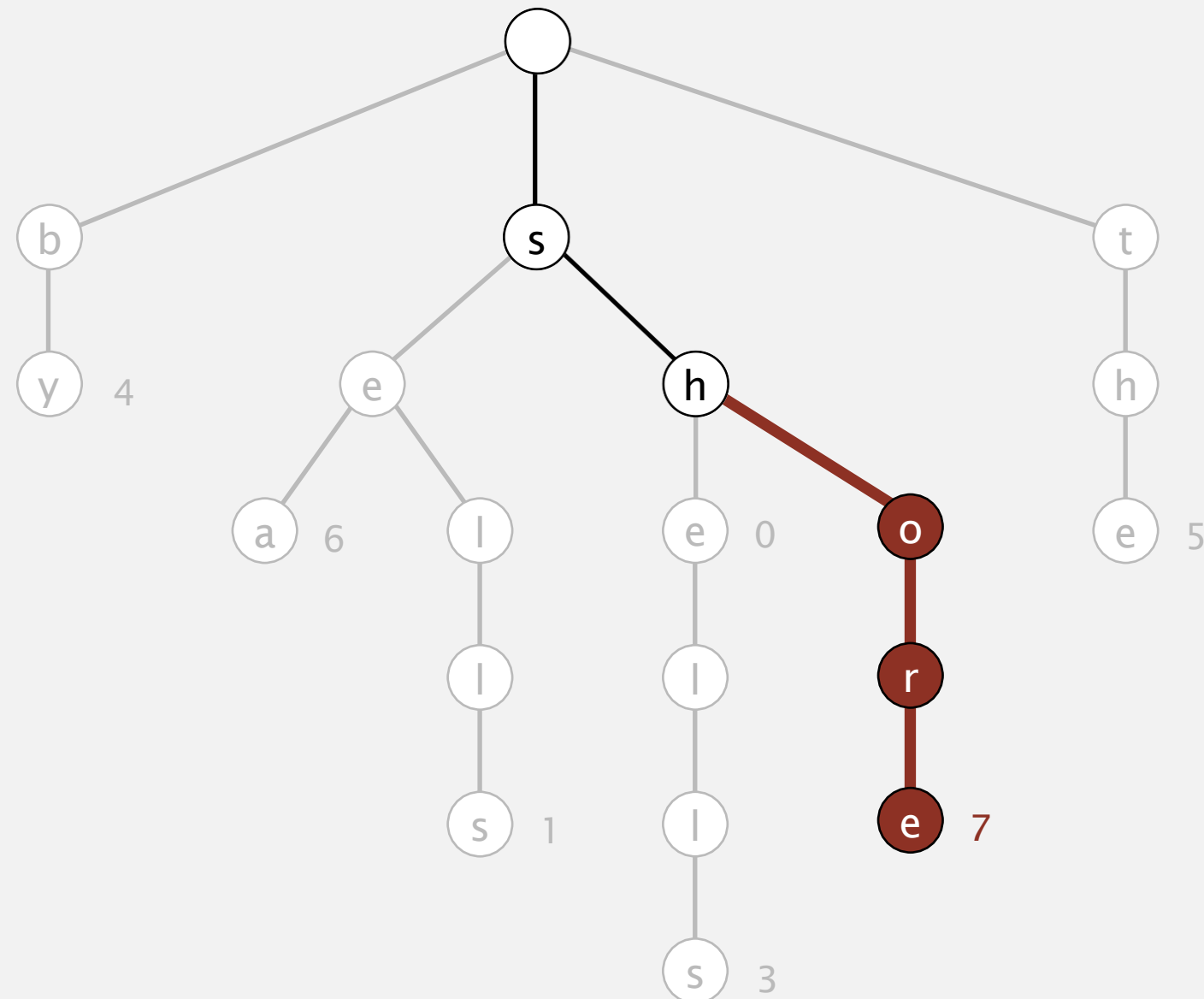get("shelter")

no link to t
(return null)
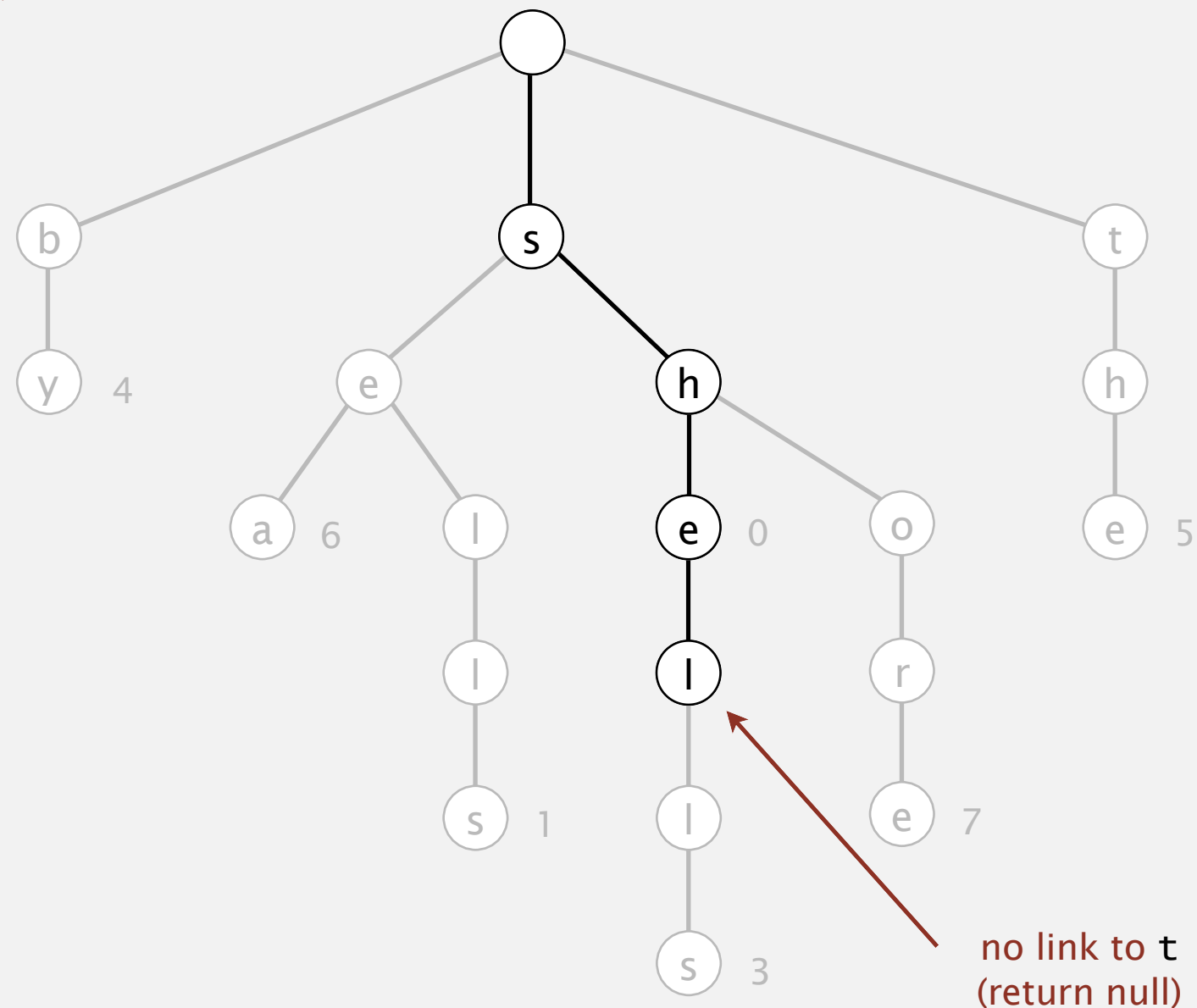
Insertion into a trie

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.
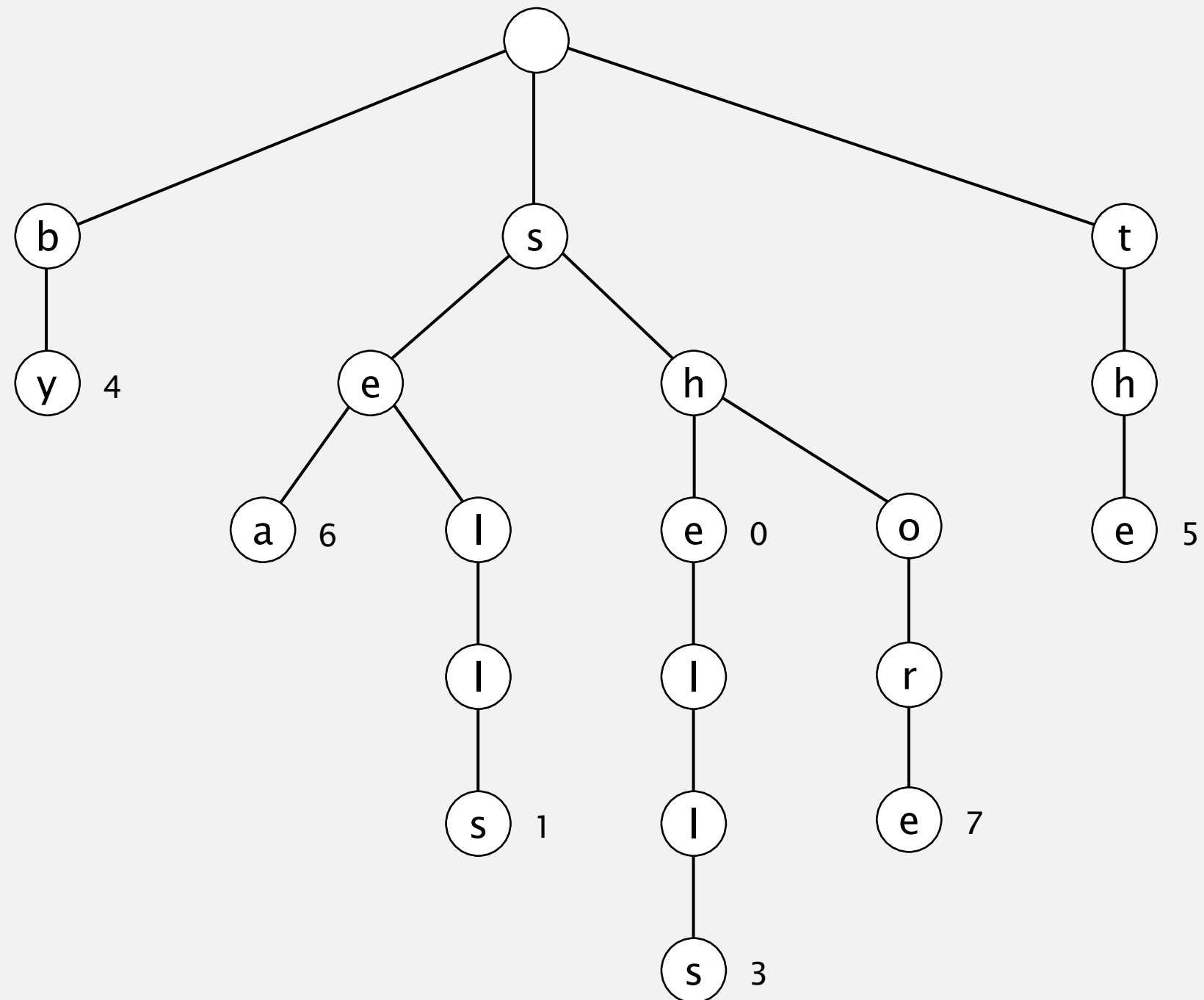
put("shore", 7)

Trie construction demo
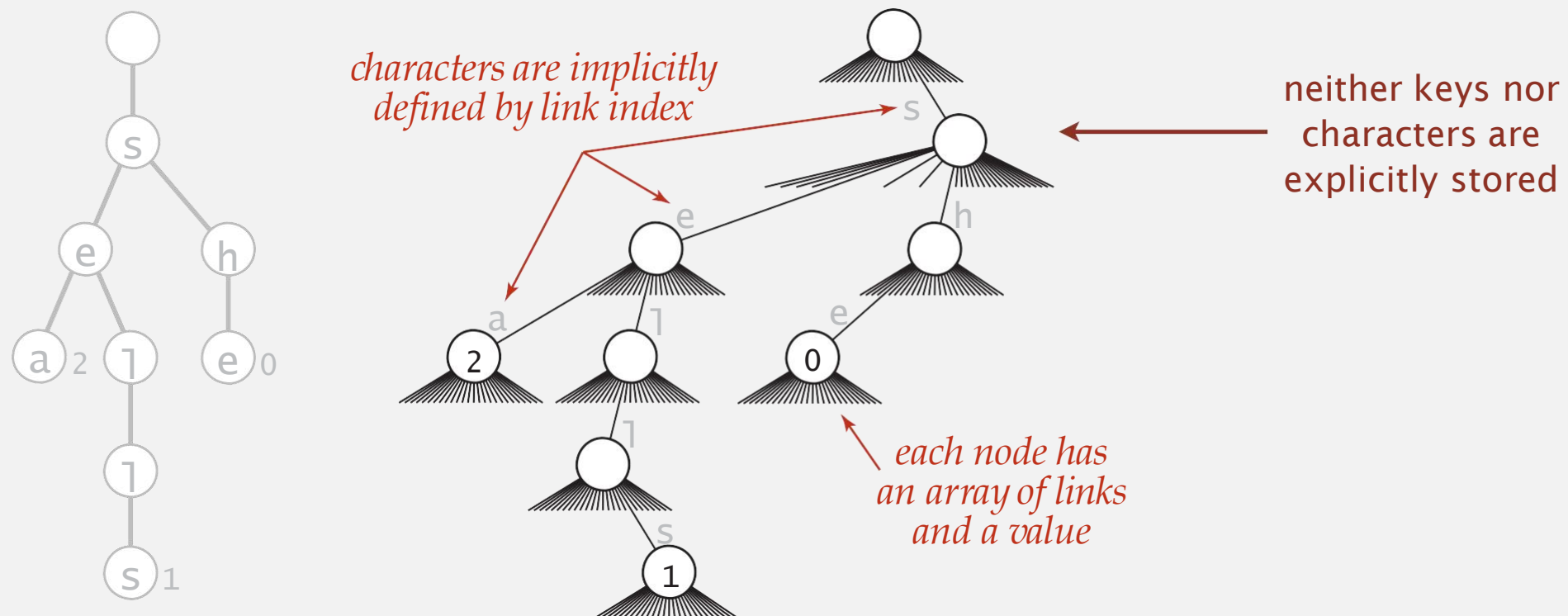
trie

trie

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];
}
```

no generic array creation in Java

characters are implicitly defined by link index

neither keys nor characters are explicitly stored

each node has an array of links and a value

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;
    }
    ⋮
```

```
   :

   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 strings share long common prefixes)

aracters are implicitly
defined by link index

each node h
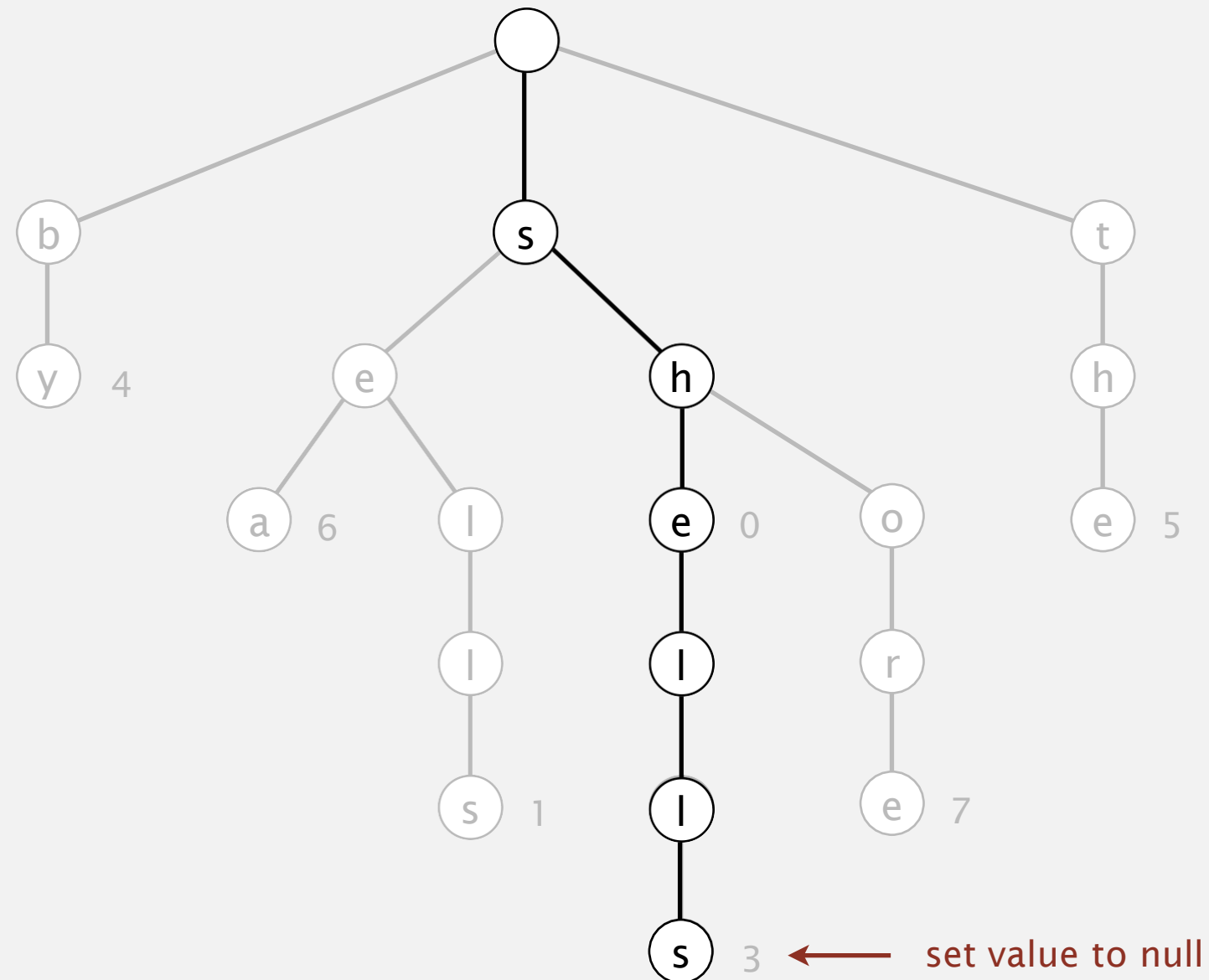*an array of li*
and a valu
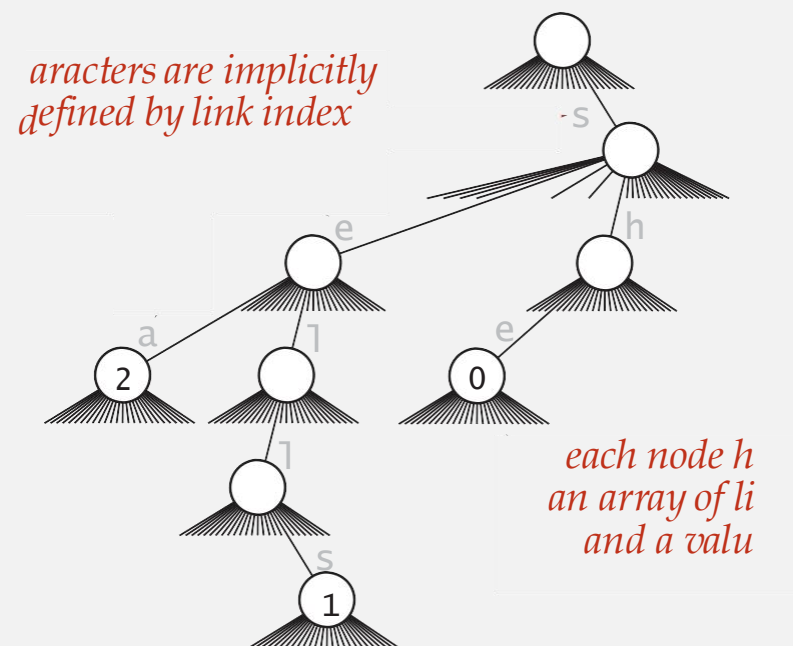
Bottom line. Fast search hit and even faster search miss, but wastes space.

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).

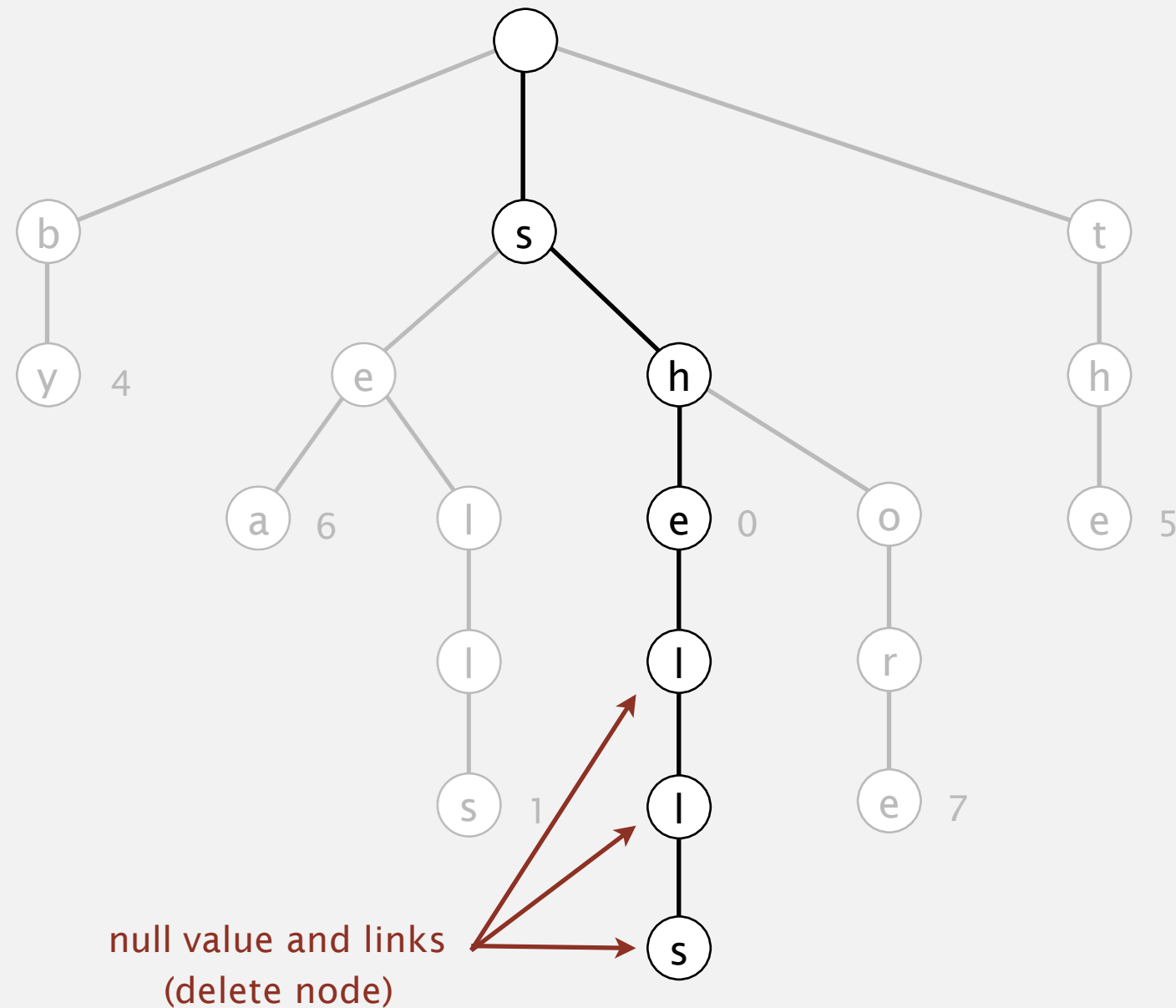**delete("shells")**

set value to null

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")

null value and links
(delete node)

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.
- Works well for medium 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

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

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-
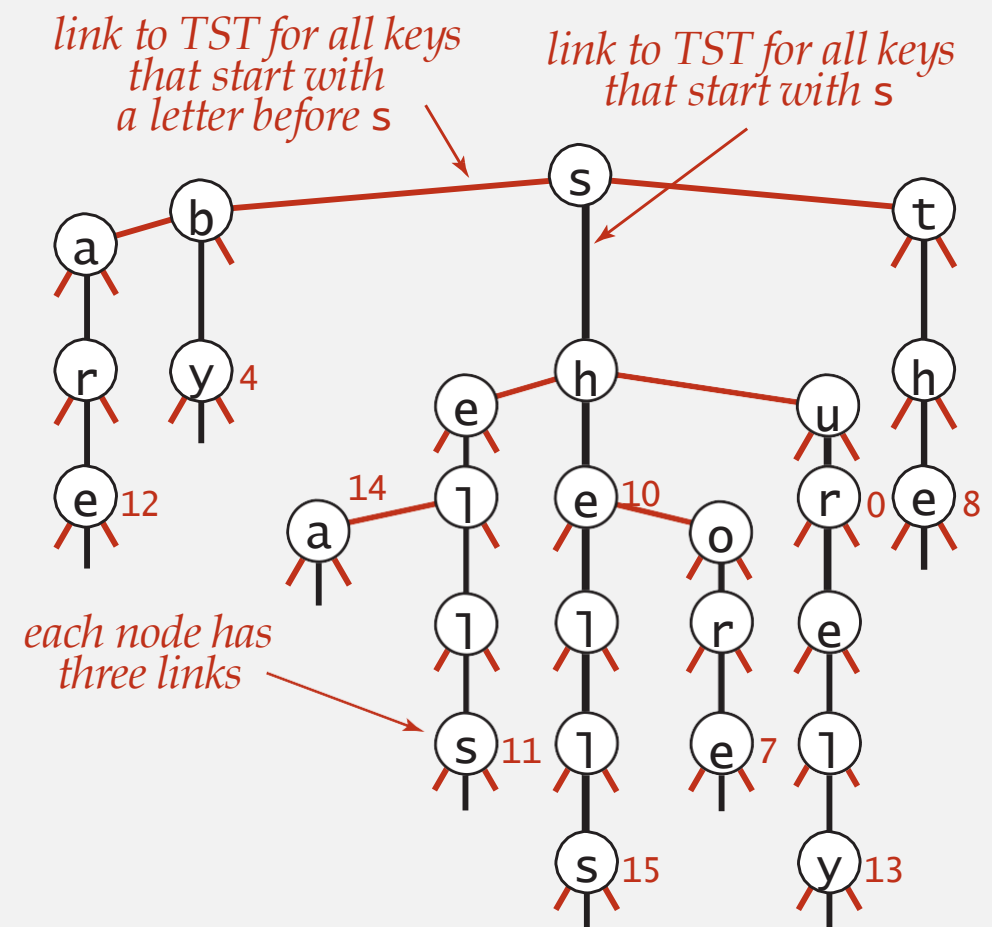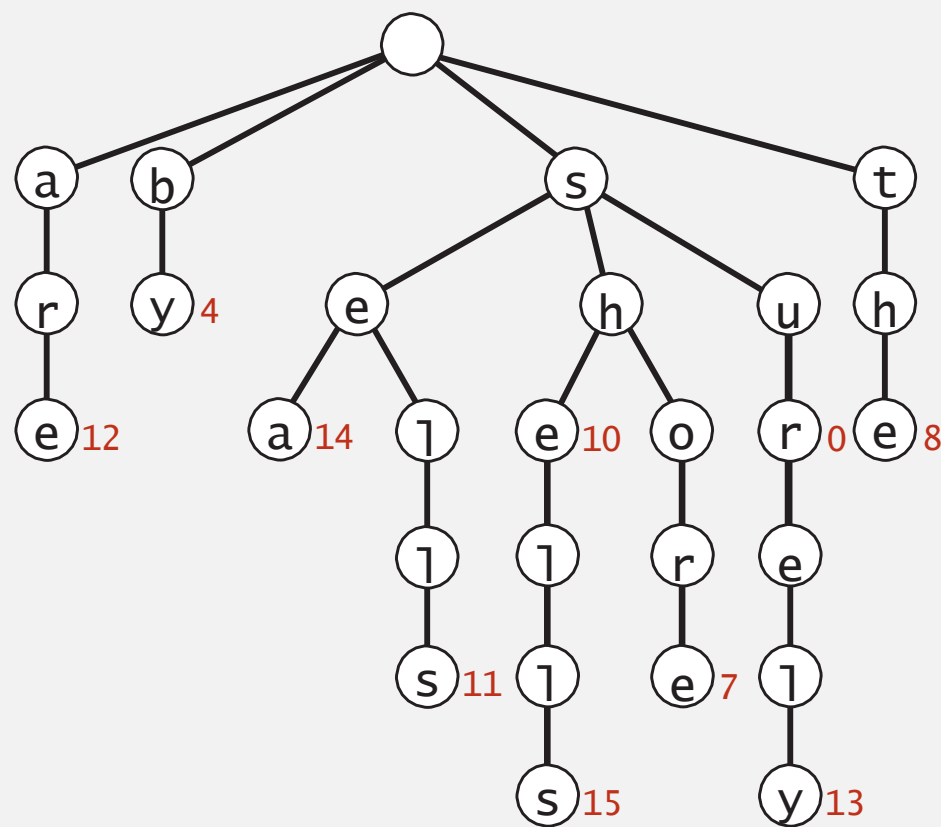
that is competitive with the most efficient string sorting programs known. The second program is a symbol table implementation that is faster than hashing, which is commonly regarded as the fastest symbol table implementation. The symbol table implementation is much more space-efficient than multiway trees, and supports more advanced searches.

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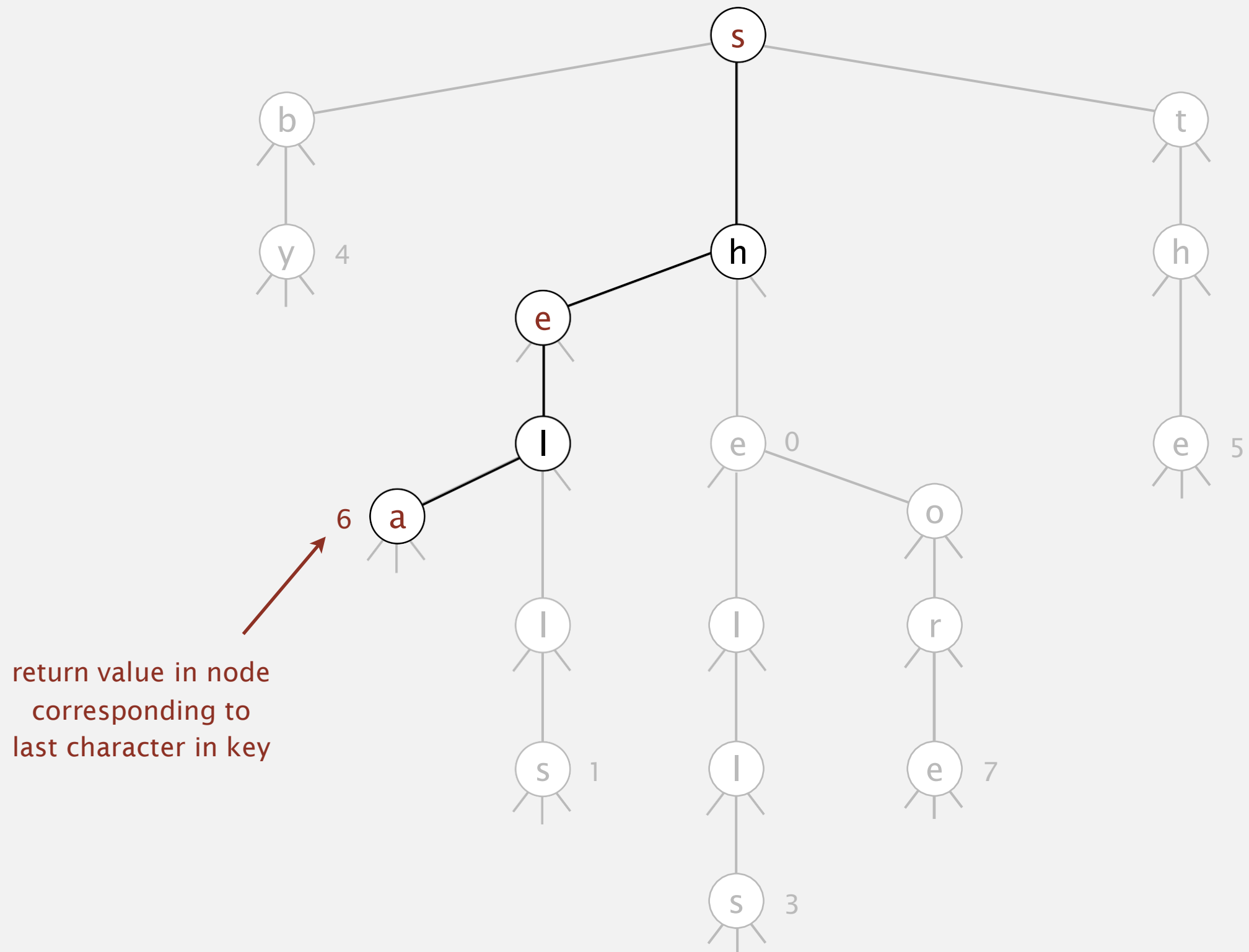
Ternary search tries

- Store characters and values in nodes (not keys).

- Each node has 3 children: smaller (left), equal (middle), larger (right).
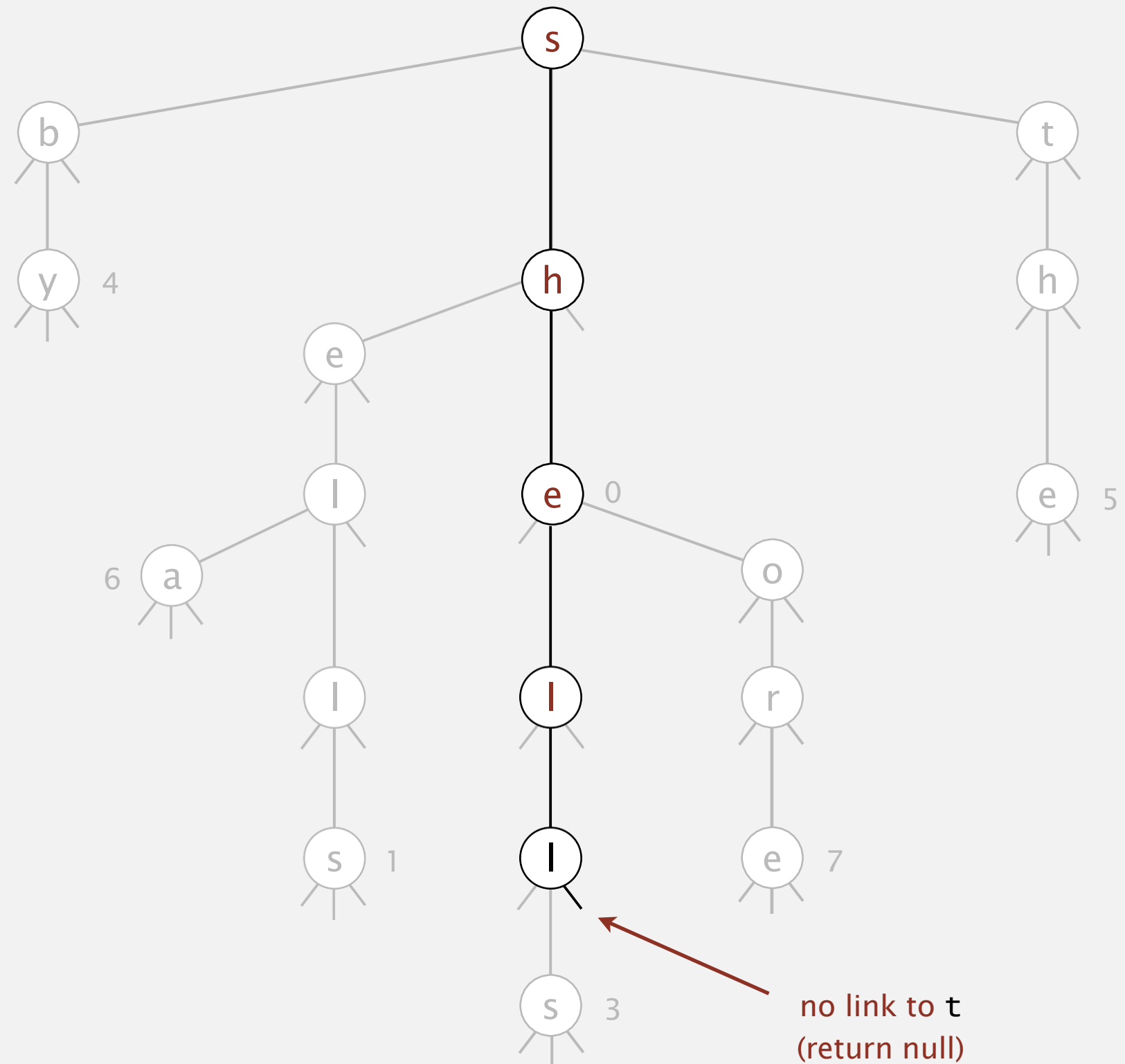

TST representation of a trie

get("sea")

return value in node
corresponding to
last character in key

get("shelter")
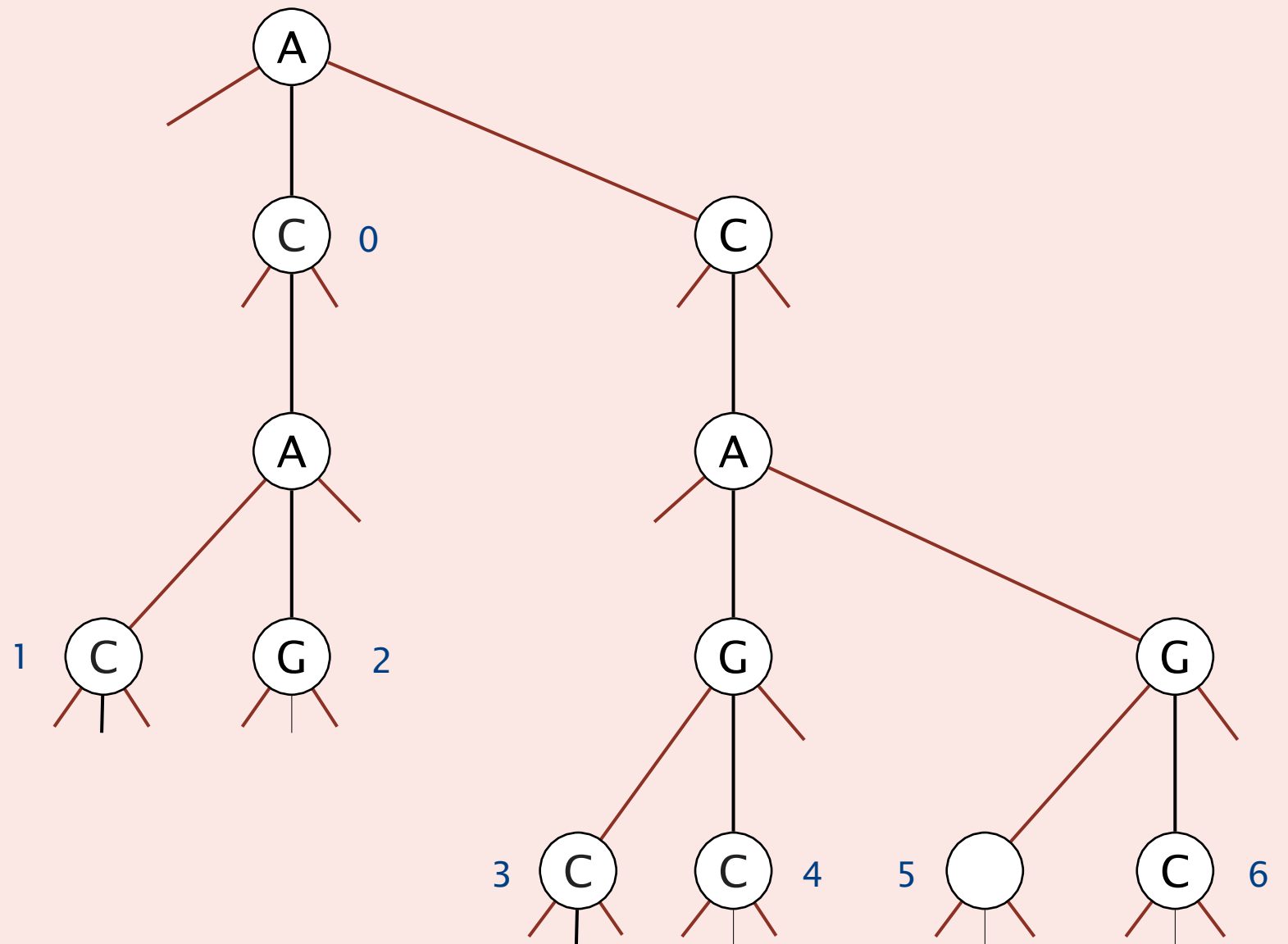

no link to t
(return null)

ternary search trie

ternary search trie
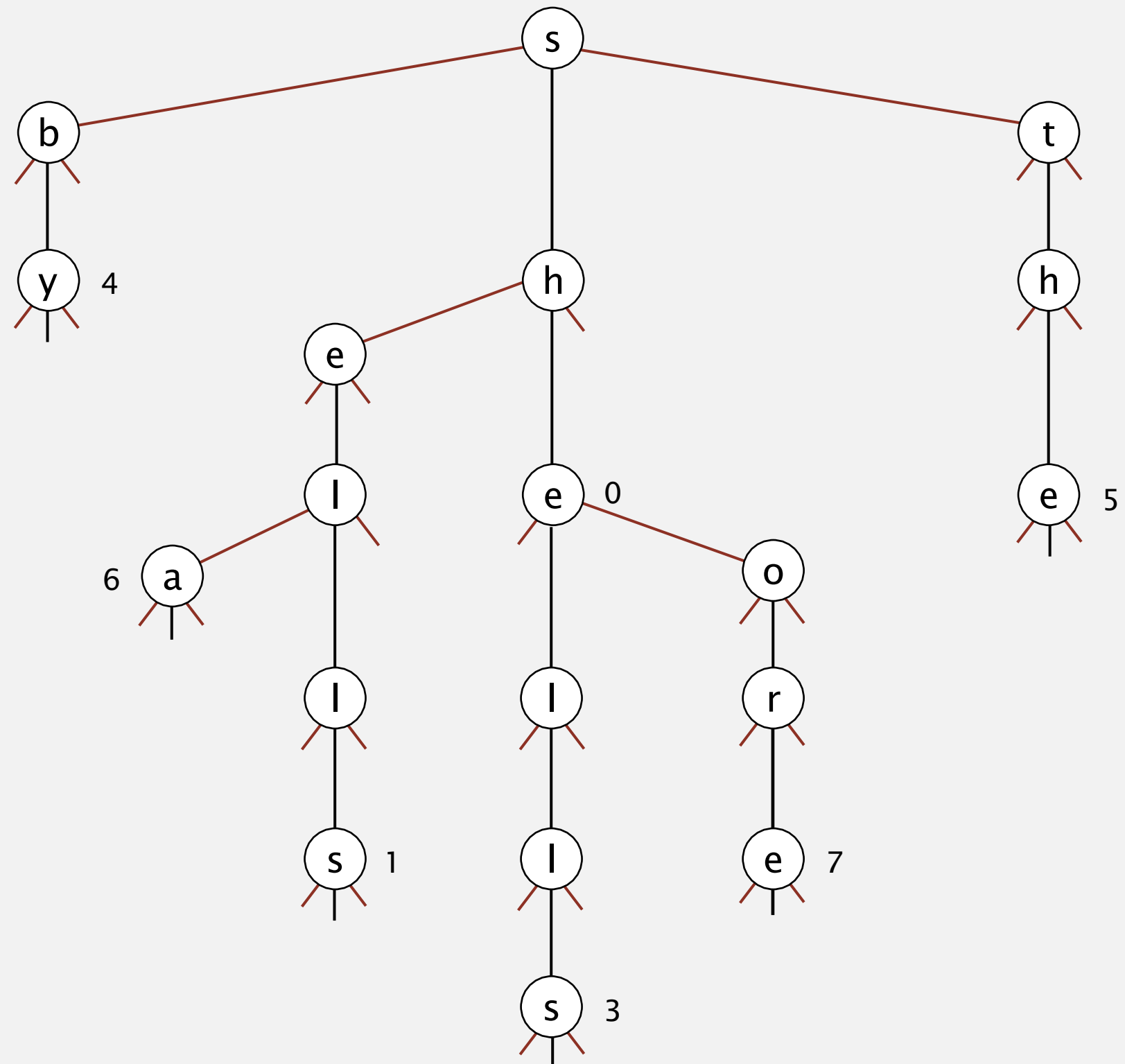
Trie quiz 1

Which value is associated with the string key "CAC" ?

A. 3

B. 4

C. 5

D. *null*

E. *I don't know.*

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.

get("sea")

match: take middle link, move to next char

mismatch: take left or right link, do not move to next char

return value associated with last key character

26-way trie vs. TST

26-way trie. 26 null links in each leaf.
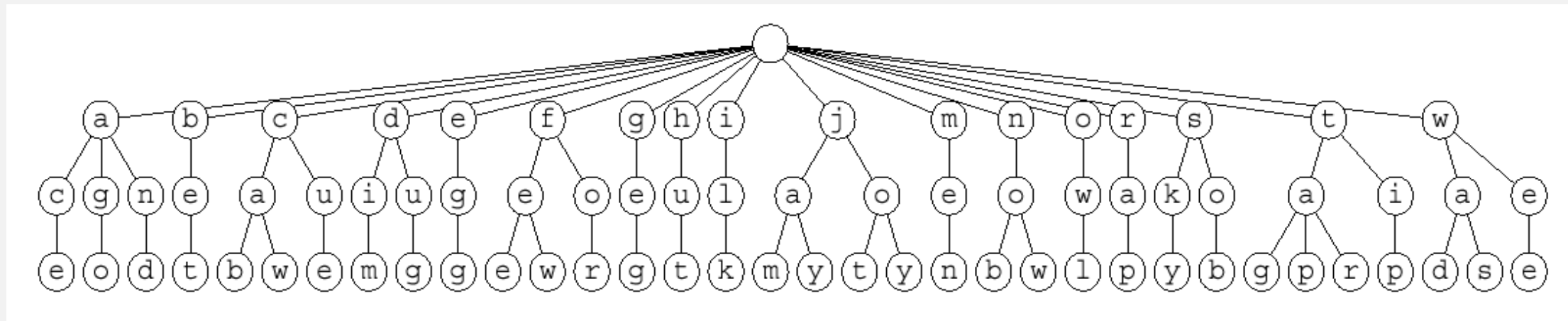
26- way trie (1035 null links, not shown)

TST. 3 null links in each leaf.

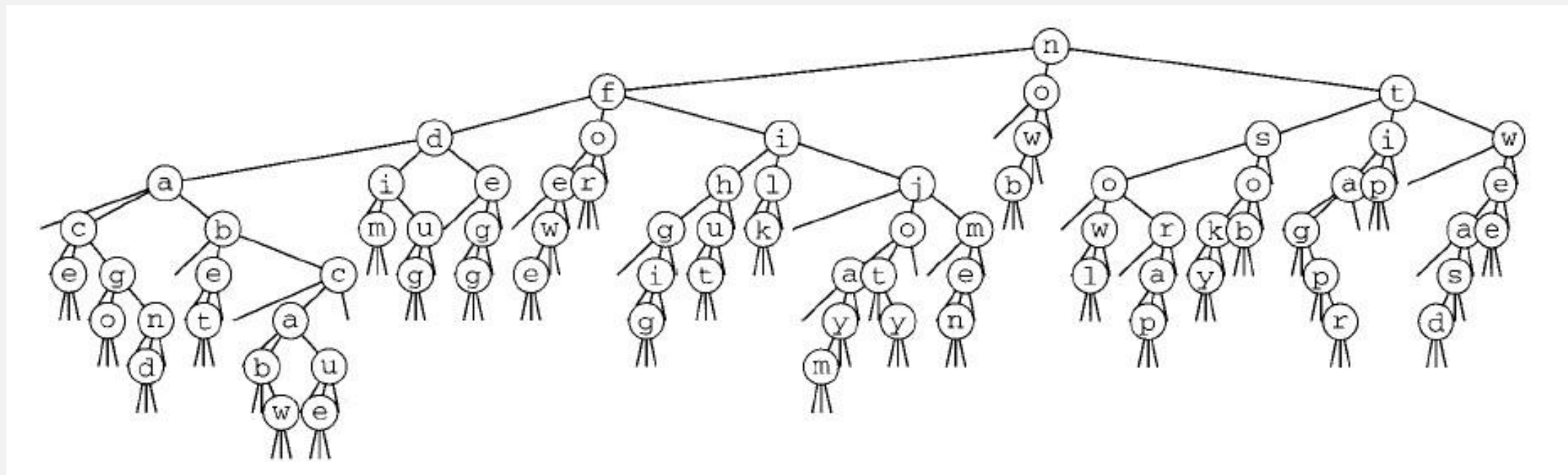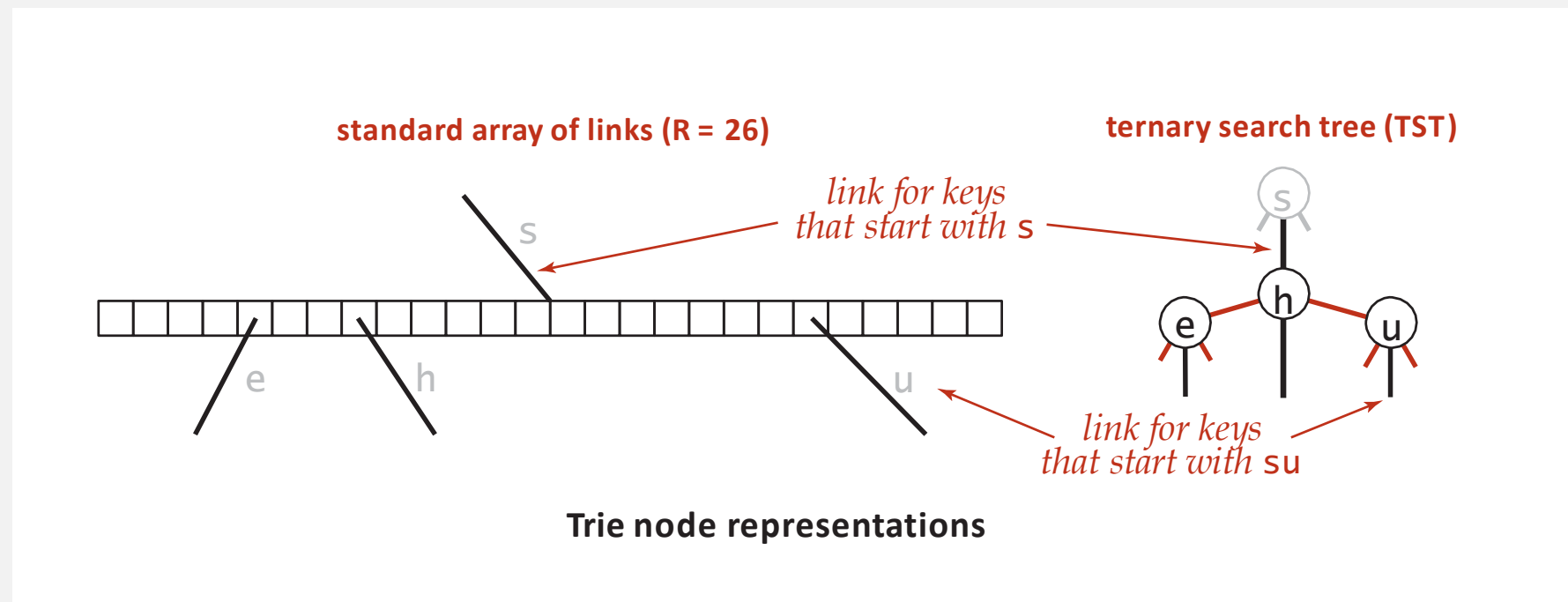**TST (155 null links)**

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;
}
```


standard array of links (R = 26)   ternary search tree (TST)

link for keys that start with s

link for keys that start with su

Trie node representations

```
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 implementations cost summary

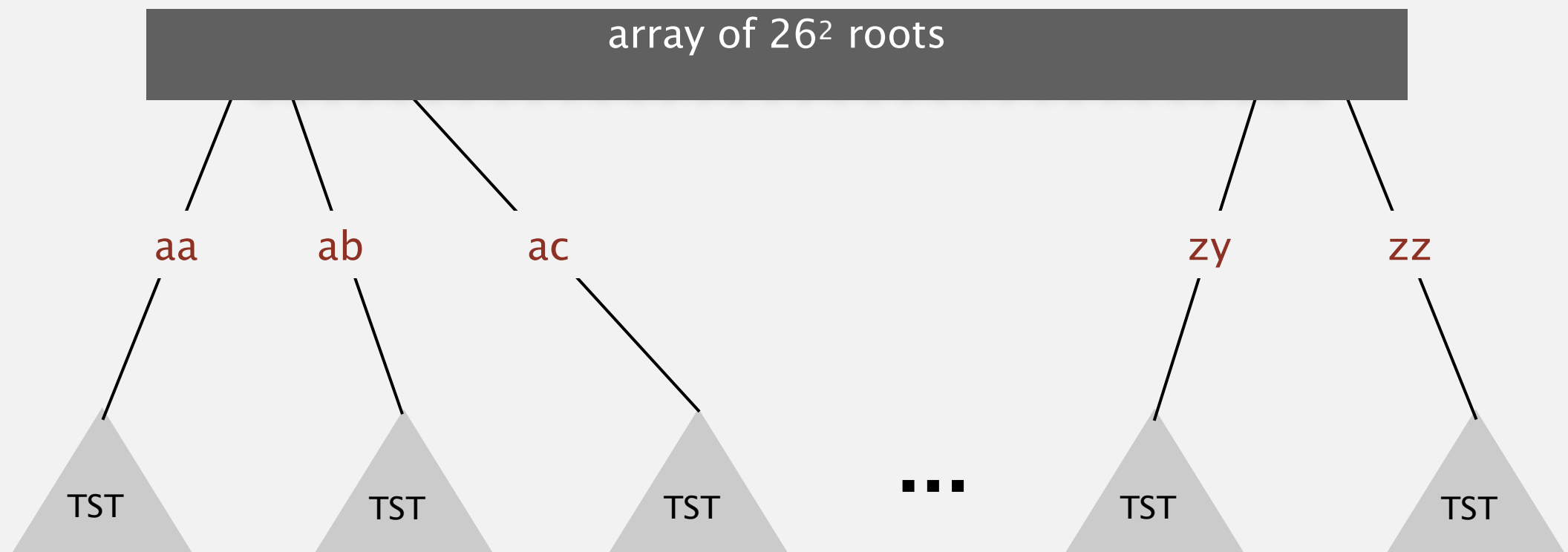| implementation	character accesses (typical case)				dedup	
	search hit	search miss	insert	space (references)	moby.txt	actors.txt
red-black BST	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	$4N$	1.40	97.4
hashing (linear probing)	L	L	L	$4N$ *to* $16N$	0.76	40.6
R-way trie	L	$\log_R N$	L	$(R+1)\,N$	1.12	*out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

	character accesses (typical case)				dedup	
implementation	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*
TST	$L + \ln N$	$\ln N$	$L + \ln N$	$4\,N$	0.72	38.7
TST with R²	$L + \ln N$	$\ln N$	$L + \ln N$	$4\,N + R^2$	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. | No balancing; poor behavior on sorted insertions. |

- 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] | But without the performance guarantees |

5.2 TRIES

‣ R-way tries

‣ ternary search tries

‣ character-based operations

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

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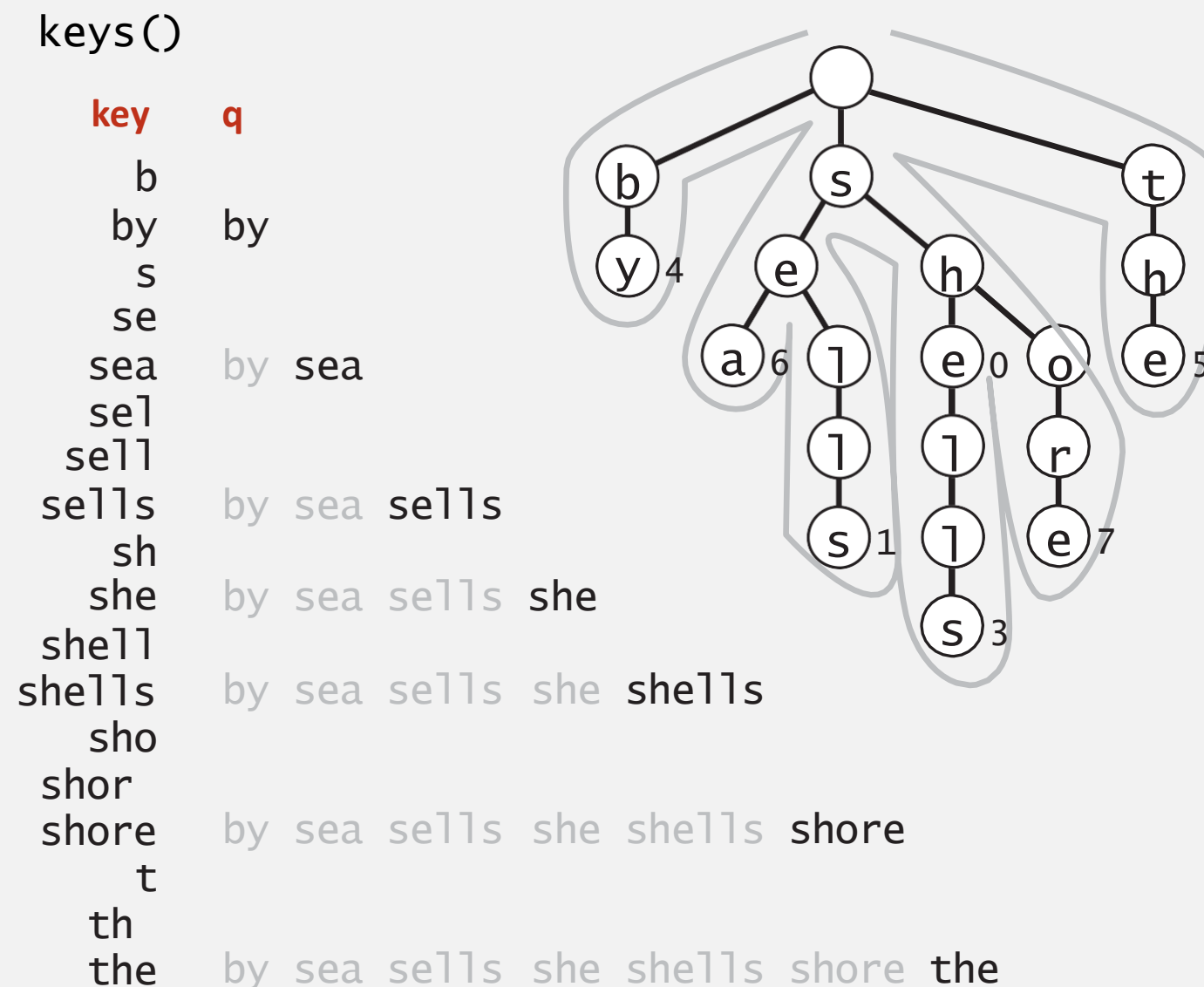
Remark. Can also add other ordered ST methods, e.g., `floor()` and `rank()`.

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.


```
keys()

    key     q
      b
     by     by
      s
     se
    sea     by sea
    sel
   sell
  sells     by sea sells
     sh
    she     by sea sells she
   shell
  shells    by sea sells she shells
    sho
   shor
  shore     by sea sells she shells shore
      t
     th
    the     by sea sells she shells shore the
```

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> results = new Queue<String>();
   collect(root, "", results);
   return results;
}


private void collect(Node x, String prefix, Queue<String> results)
{
   if (x == null) return;
   if (x.val != null) results.enqueue(prefix.toString());
   for (char c = 0; c < R; c++)
      collect(x.next[c], prefix + c, results);
}
```

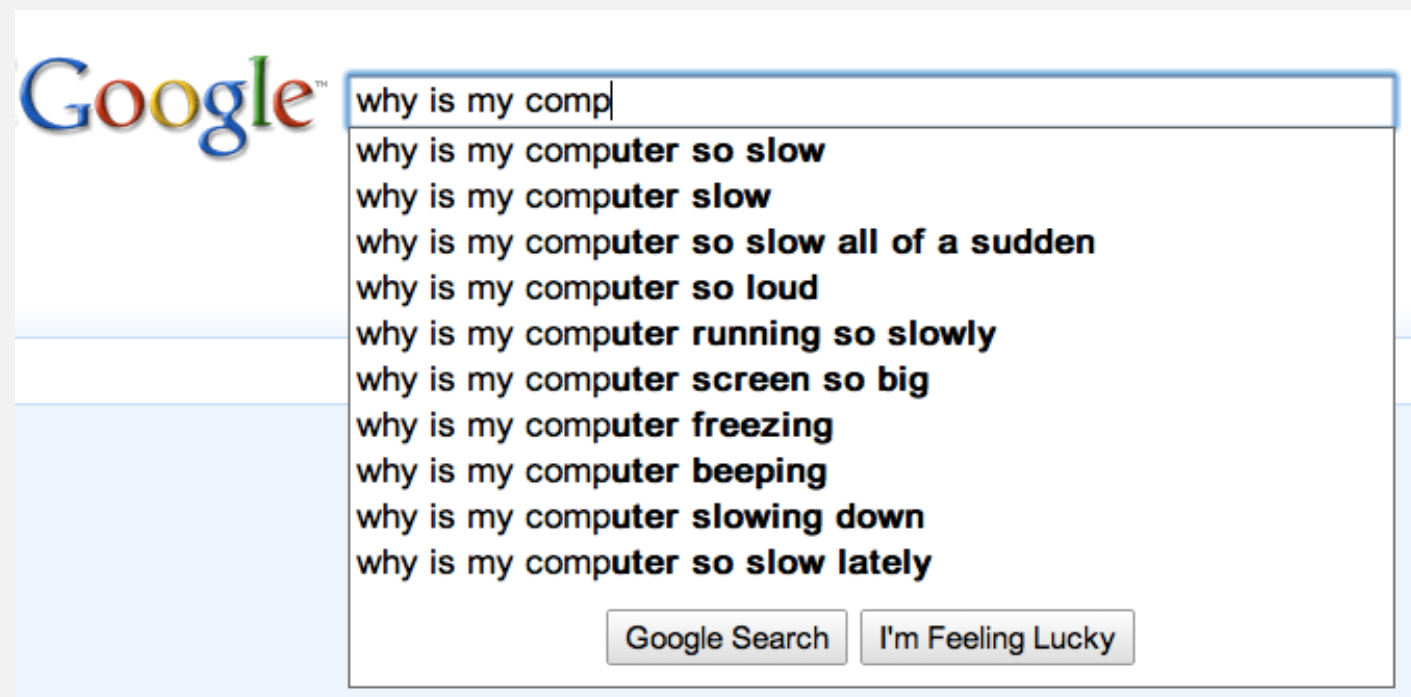sequence of characters
on path from root to x

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.
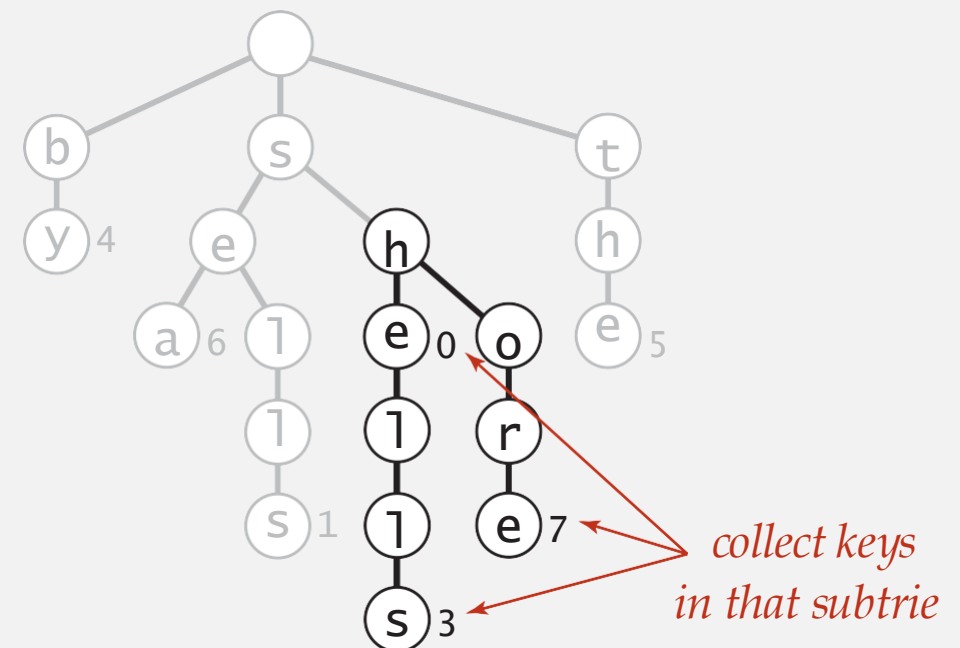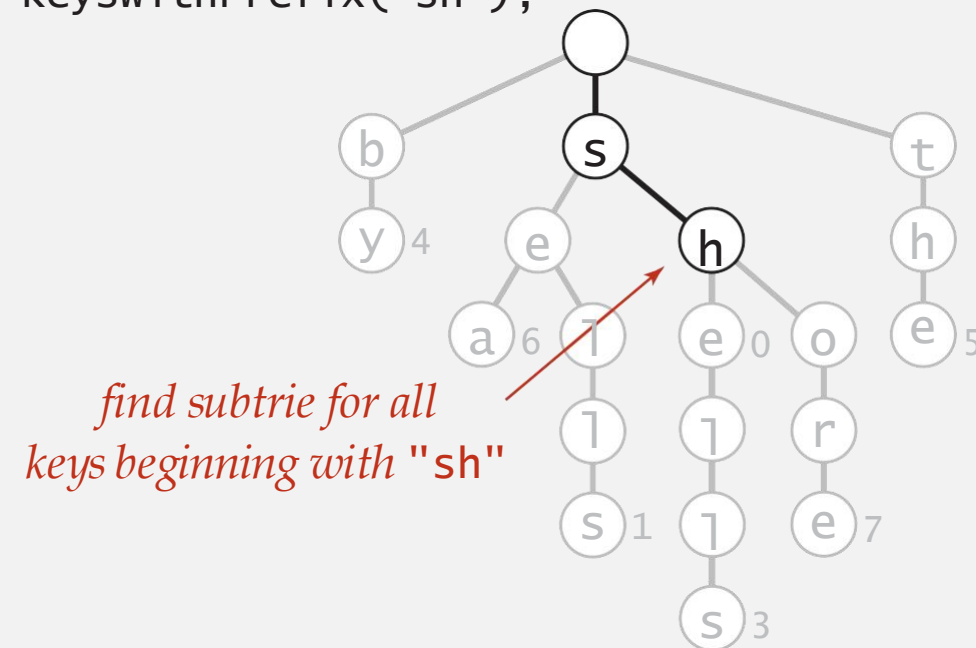
- User types characters one at a time.

- 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");
```


find subtrie for all keys beginning with **"sh"**

collect keys in that subtrie

```
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
shel	
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.
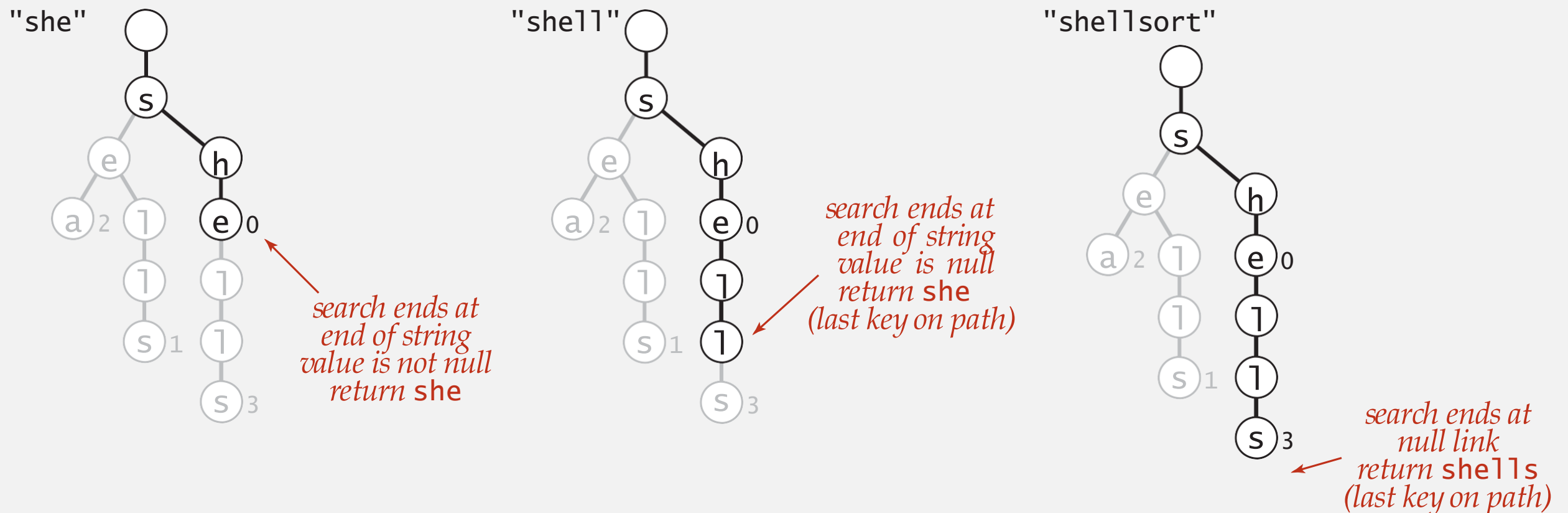
```
"128"
"128.112"                    ←  represented as 32-bit
                                binary number for IPv4
"128.112.055"                     (instead of string)

"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"
                       longestPrefixOf("128.166.123.45") = "128"
"128.112.155.13"

"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.

*search ends at
end of string
value is not null
return* **she**

*search ends at
end of string
value is null
return* **she**
(last key on path)

*search ends at
null link
return* **shells**
(last key on path)

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 (predictive texting)

Goal. Type text messages on a phone keypad.

Multi-tap input. Enter a letter by repeatedly pressing a key.
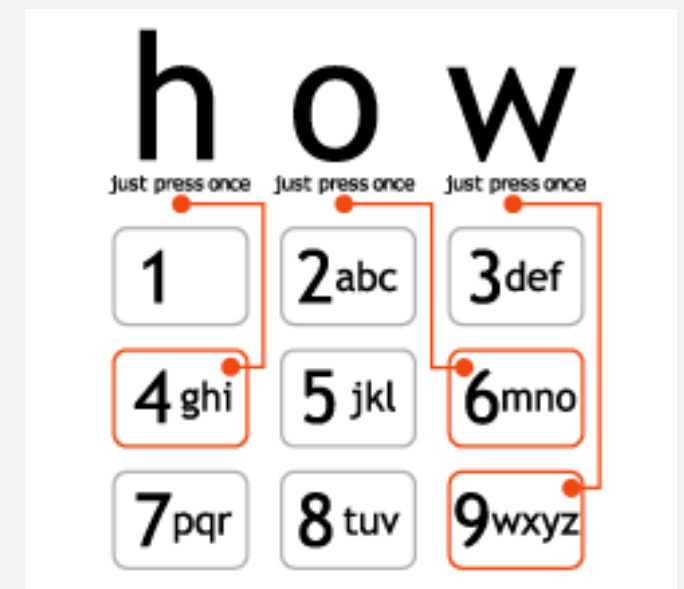
Ex. good: 4 6 6 6 6 6 6 3

"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.

T9 TEXTING

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

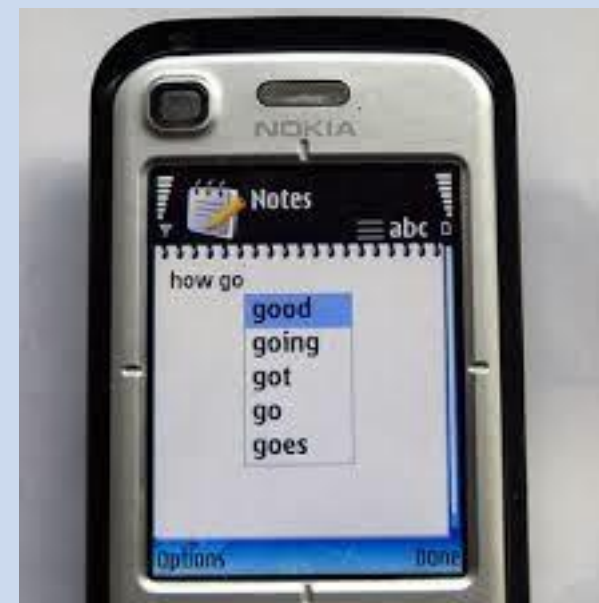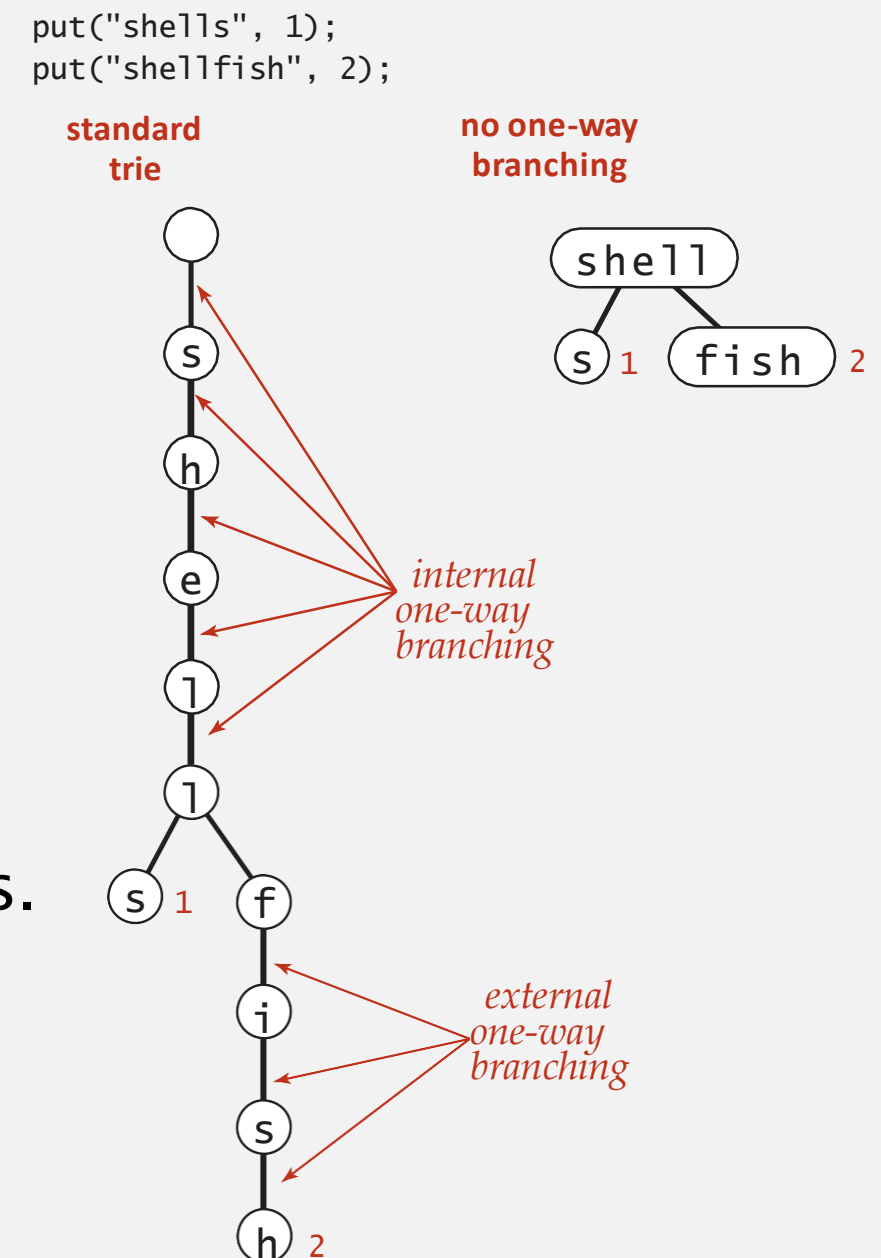# Patricia trie

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.
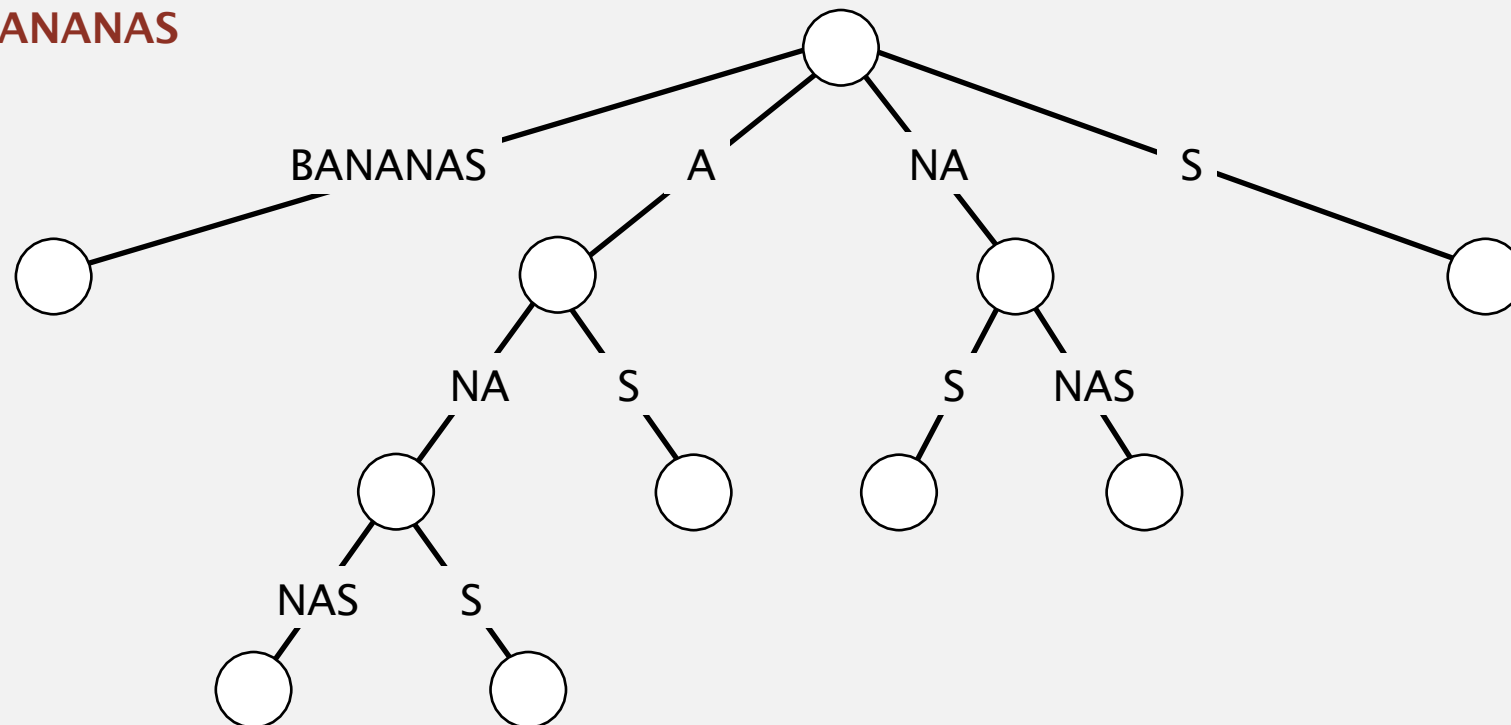- Efficiently storing and querying XML documents.

shell

s 1   fish 2

s

h

e

l   *internal one-way branching*

l

s 1   f

i   *external one-way branching*

s

h 2

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.