

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.

implementation	typical case			ordered operations	operations on keys
	search	insert	delete		
red-black BST	$1.00 \lg N$	$1.00 \lg N$	$1.00 \lg N$	yes	<code>compareTo()</code>
hashing	1^\dagger	1^\dagger	1^\dagger	no	<code>equals()</code> <code>hashCode()</code>

† 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

	character accesses (typical case)				dedup	
implementation	search hit	search miss	insert	space (references)	moby.txt	actors.txt
red-black BST	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	$4N$	1.4	97.4
hashing	L	L	L	$4N$ to $16N$	0.76	40.6

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

Parameters

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

Challenge. Efficient performance for string keys.

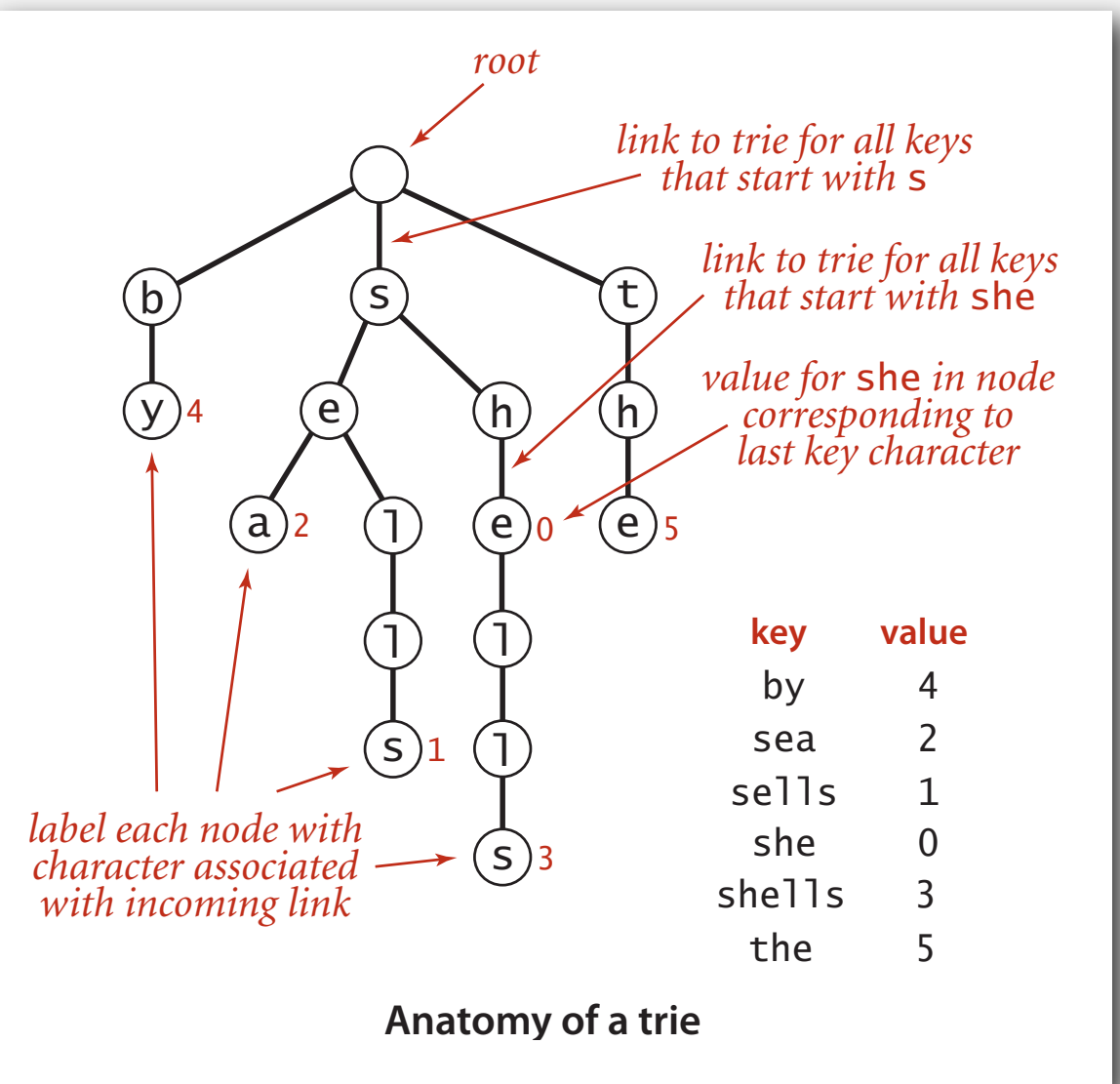
- ▶ R-way tries
- ▶ ternary search tries
- ▶ string symbol table API

Tries

Tries. [from re**trie**val, 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

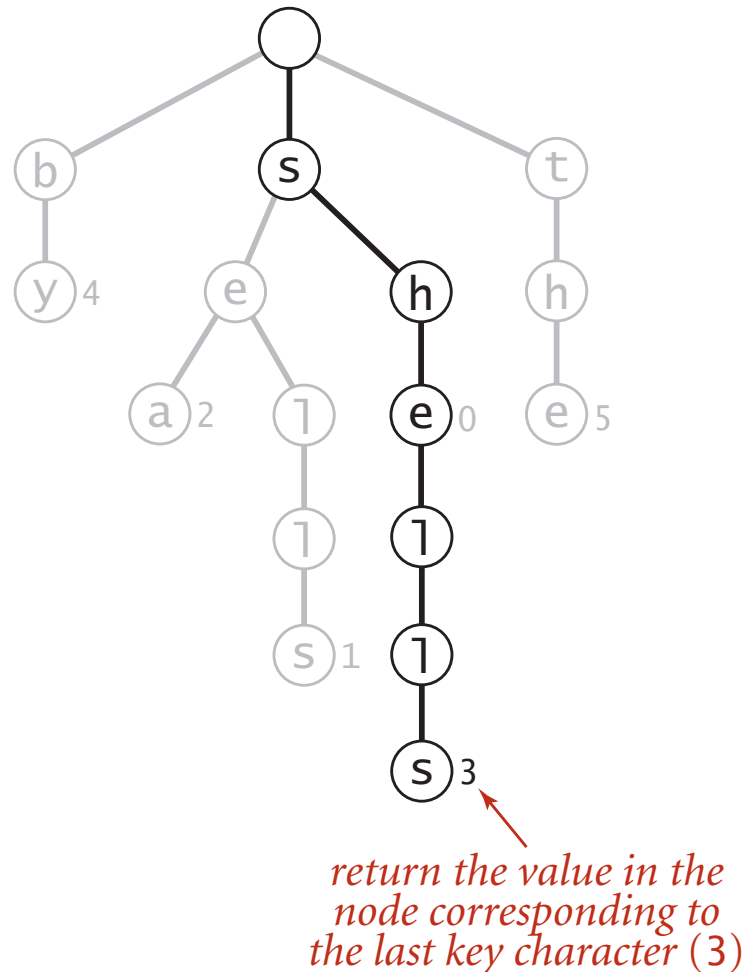


Search in a trie

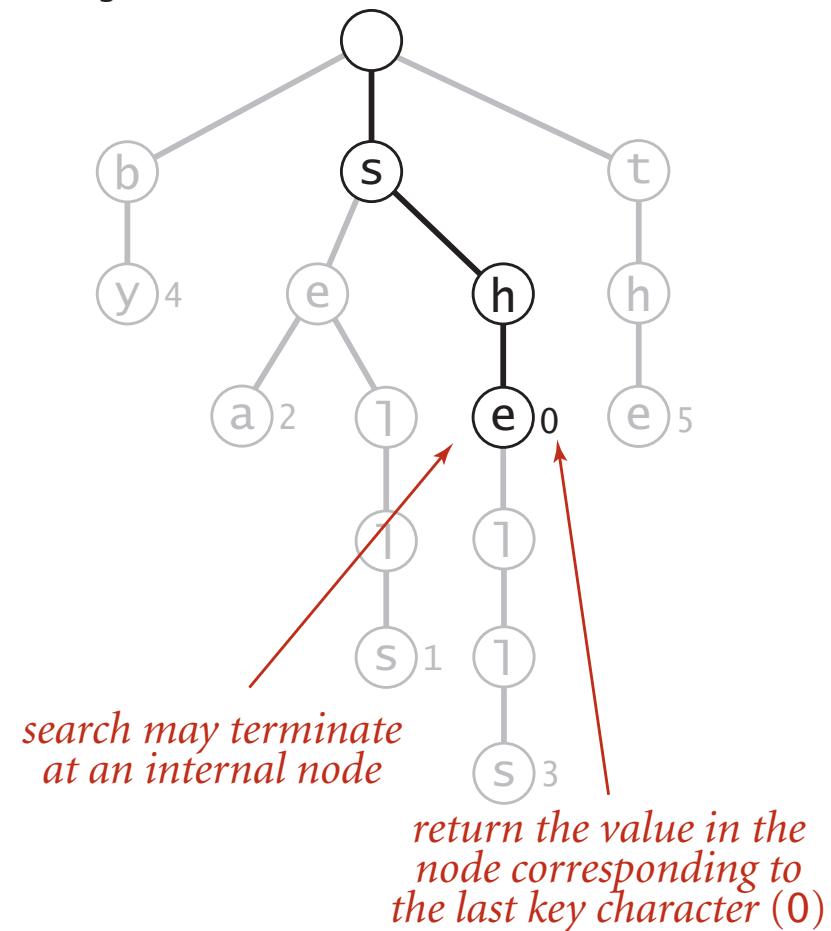
Follow links corresponding to each character in the key.

- **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")

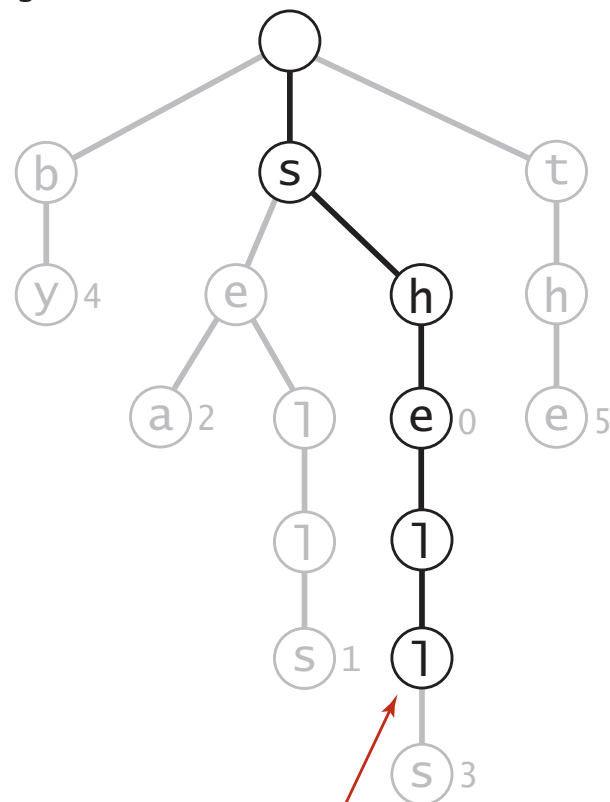


Search in a trie

Follow links corresponding to each character in the key.

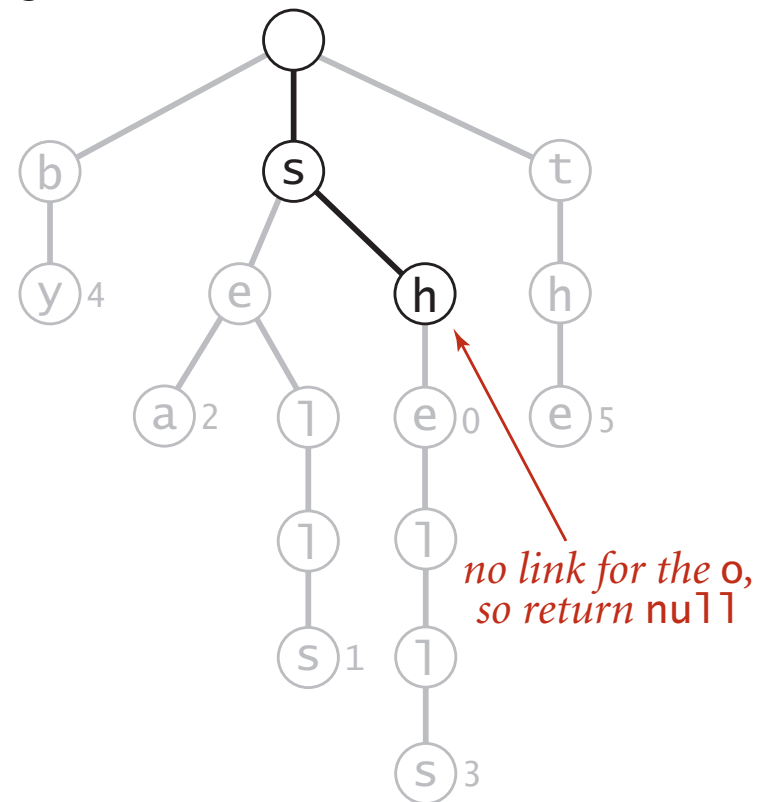
- Search hit: node where search ends has a non-null value.
- **Search miss:** reach a null link or node where search ends has null value.

get("shell")



*no value in the node
corresponding to the last key
character, so return null*

get("shore")

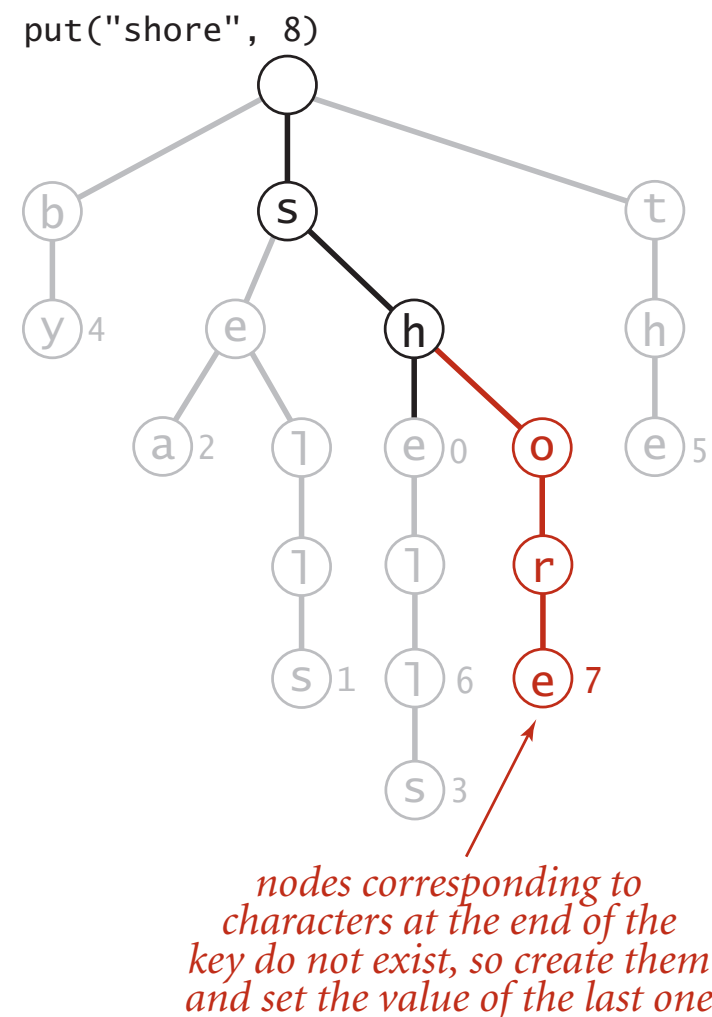
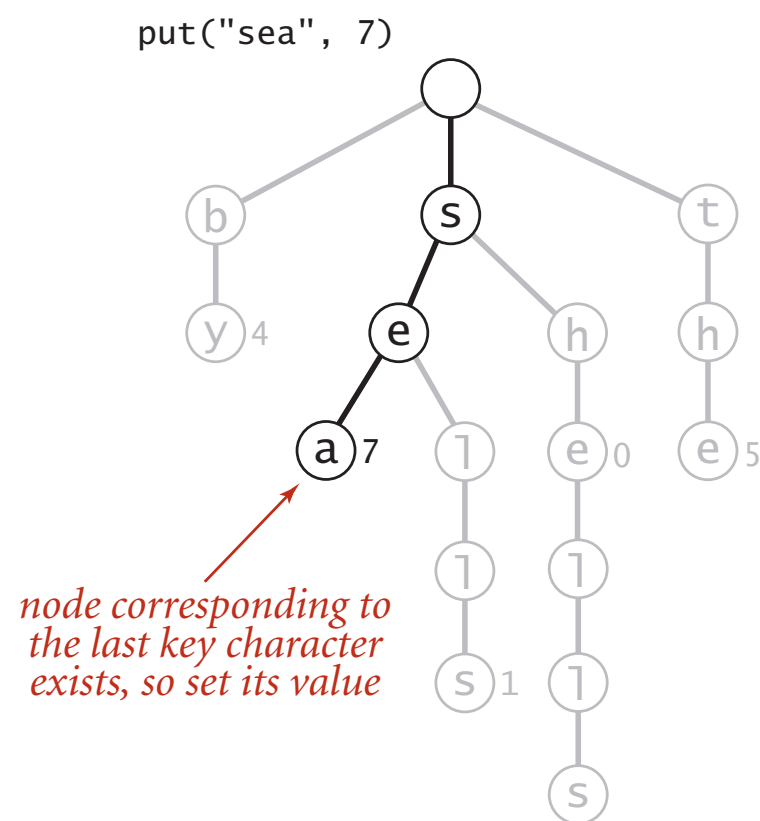


*no link for the o,
so return null*

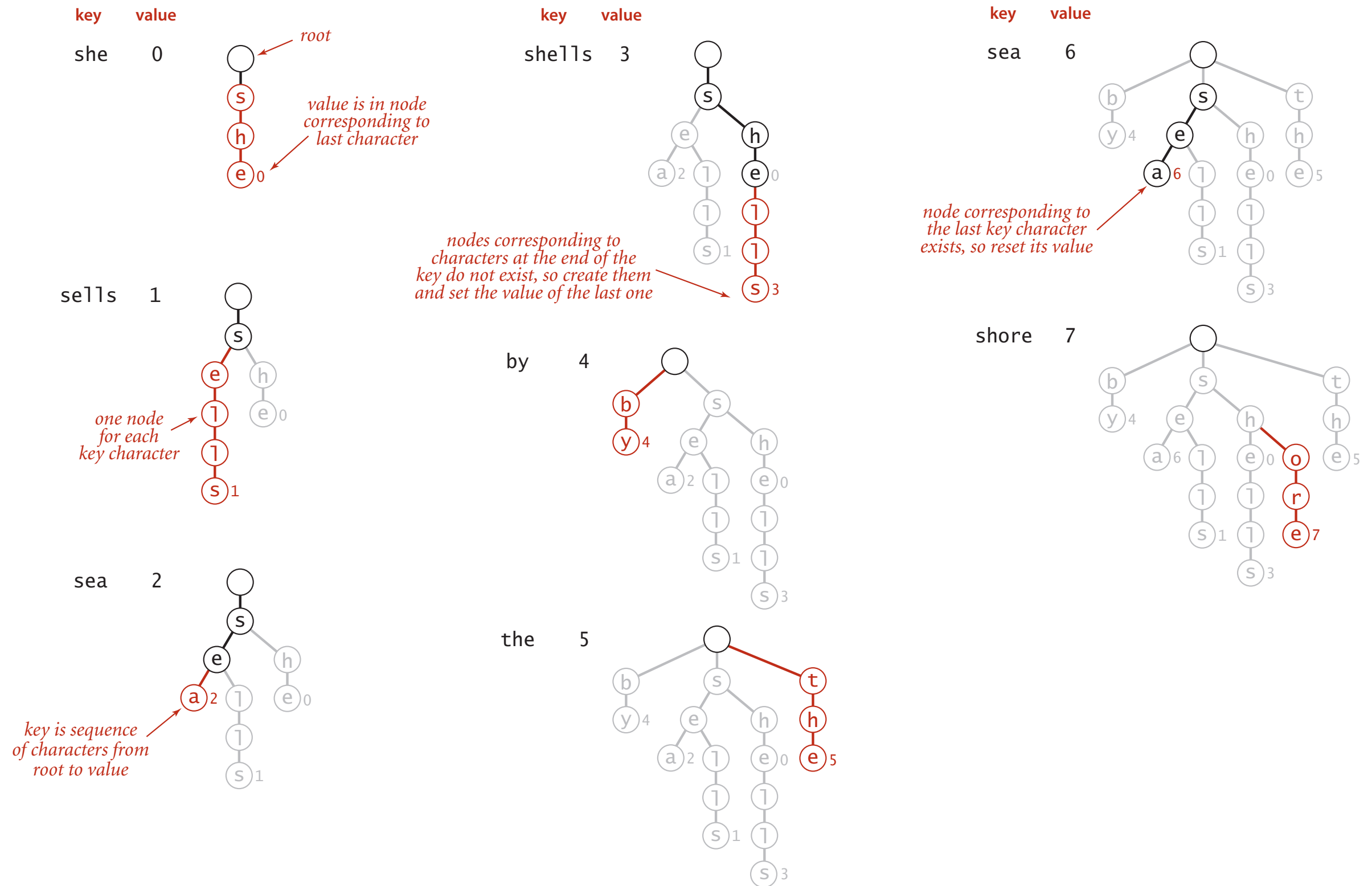
Insertion into a trie

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.



Trie construction example

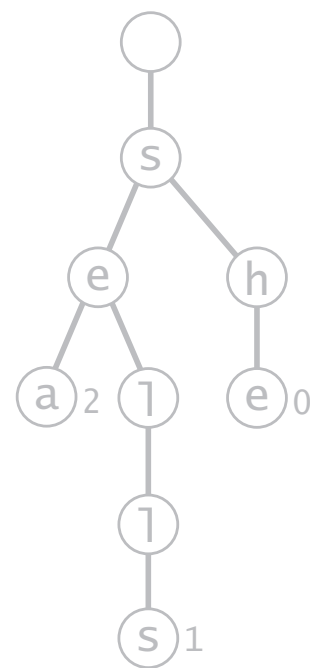


Trie representation: Java implementation

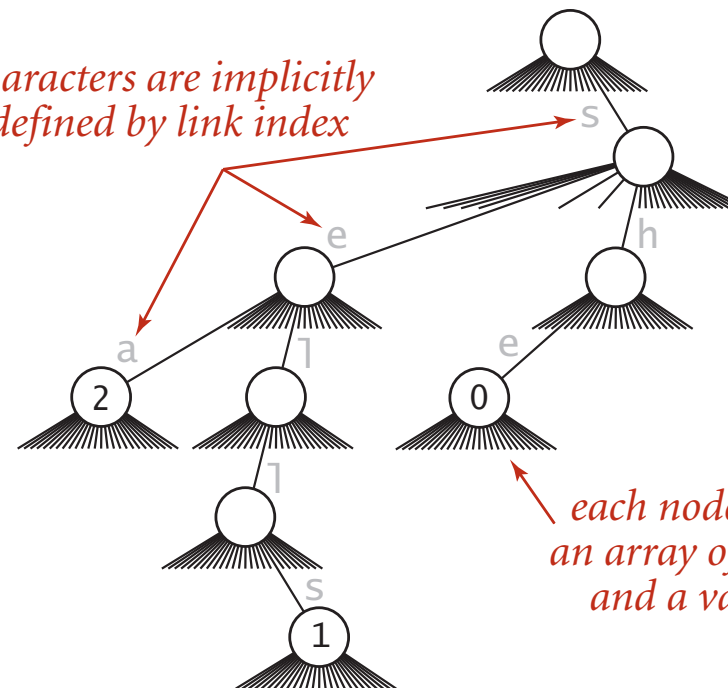
Node. A value, plus references to R nodes.

```
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



characters are implicitly
defined by link index



each node has
an array of links
and a value

keys are not
explicitly stored

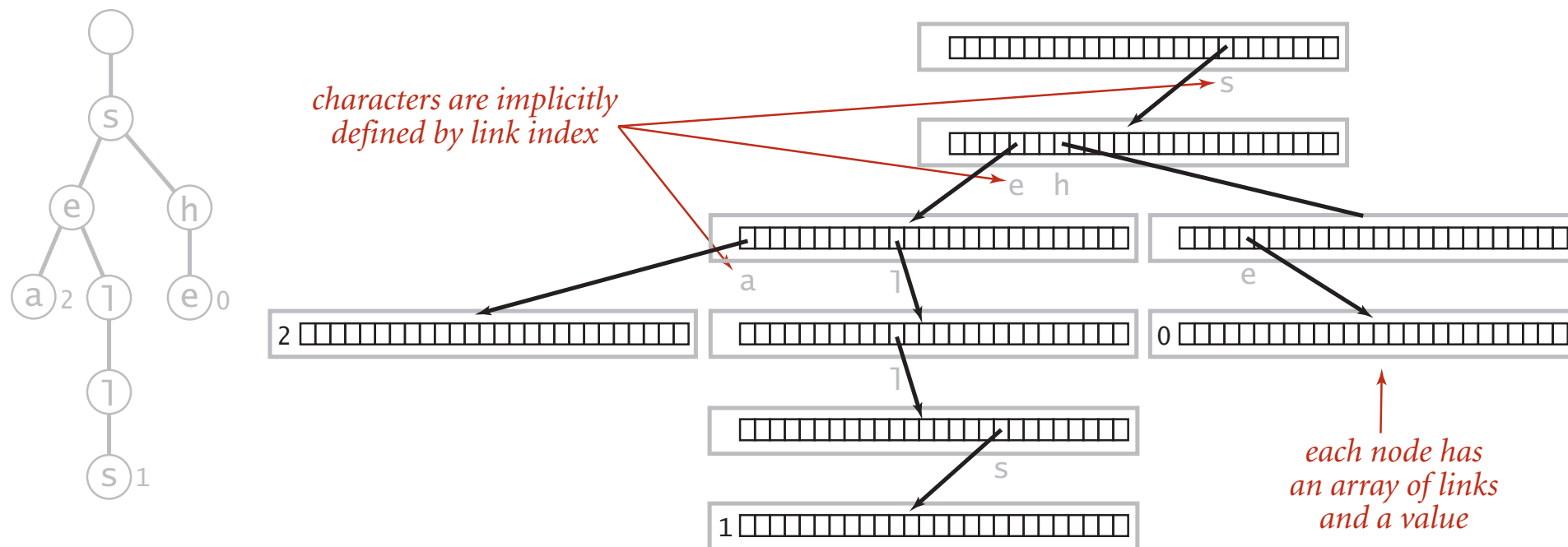
Trie representation

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



Trie representation ($R = 26$)

R-way trie: Java implementation

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

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 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

implementation	character accesses (typical case)				dedup	
	search hit	search miss	insert	space (references)	moby.txt	actors.txt
red-black BST	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	$4N$	1.4	97.4
hashing	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!

- ▶ R-way tries
- ▶ **ternary search tries**
- ▶ string symbol table API

Ternary search tries

TST. [Bentley-Sedgewick, 1997]

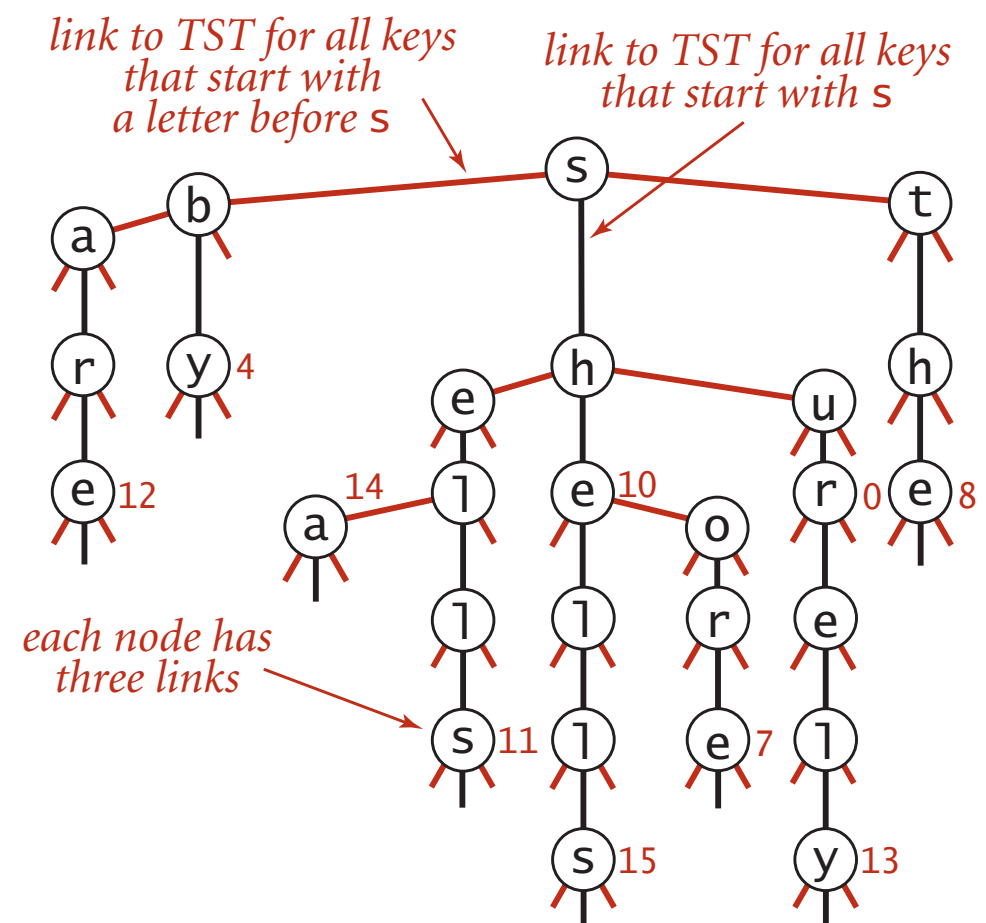
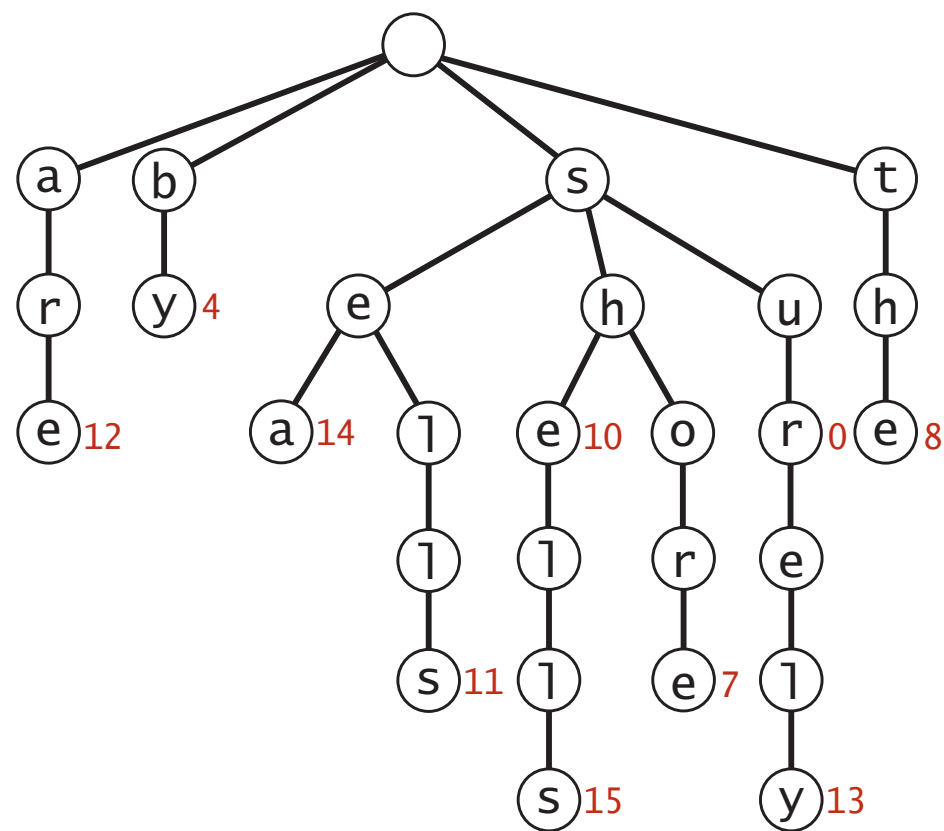
- Store characters and values in nodes (not keys).
- Each node has **three** children: smaller (left), equal (middle), larger (right).



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

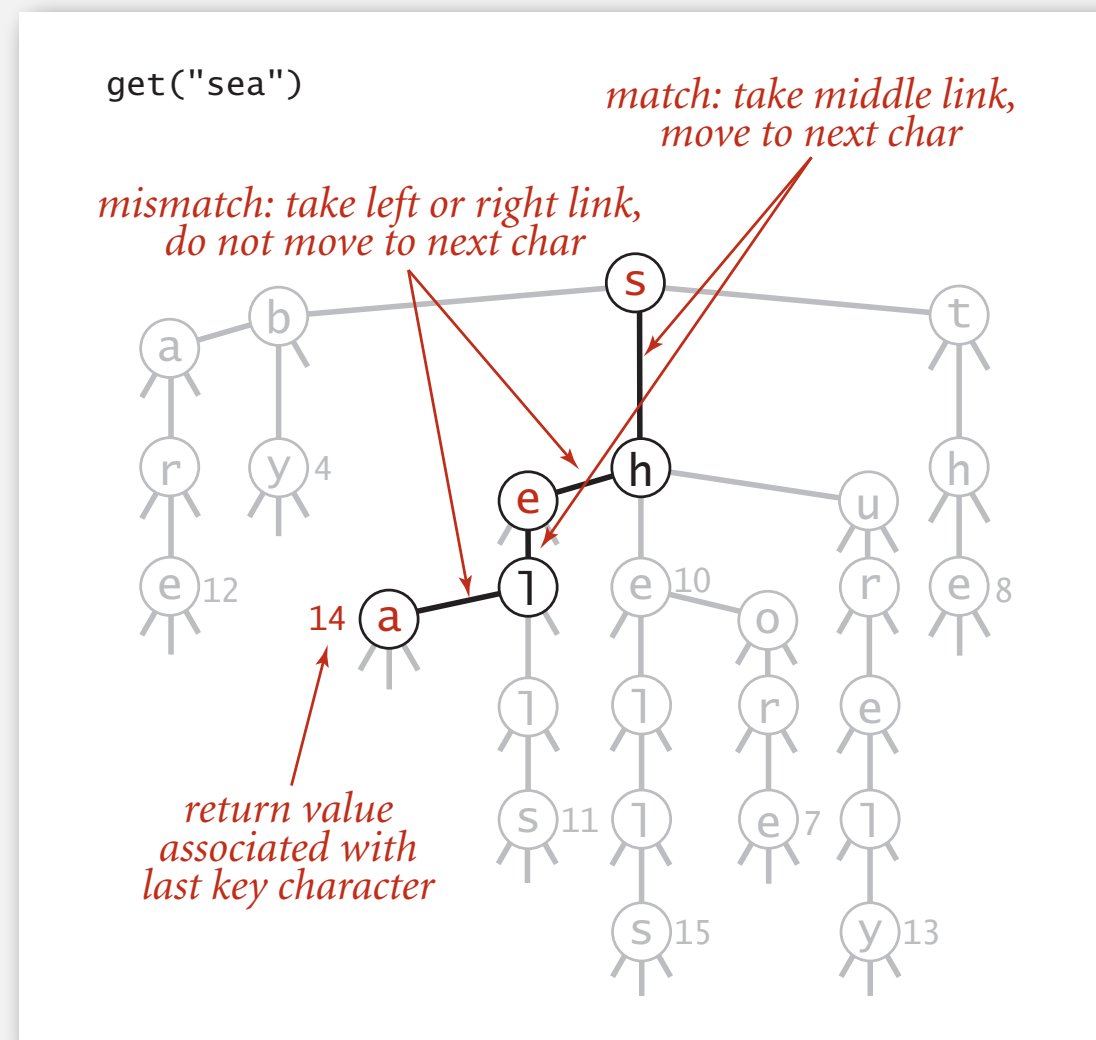
Search in a TST

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.

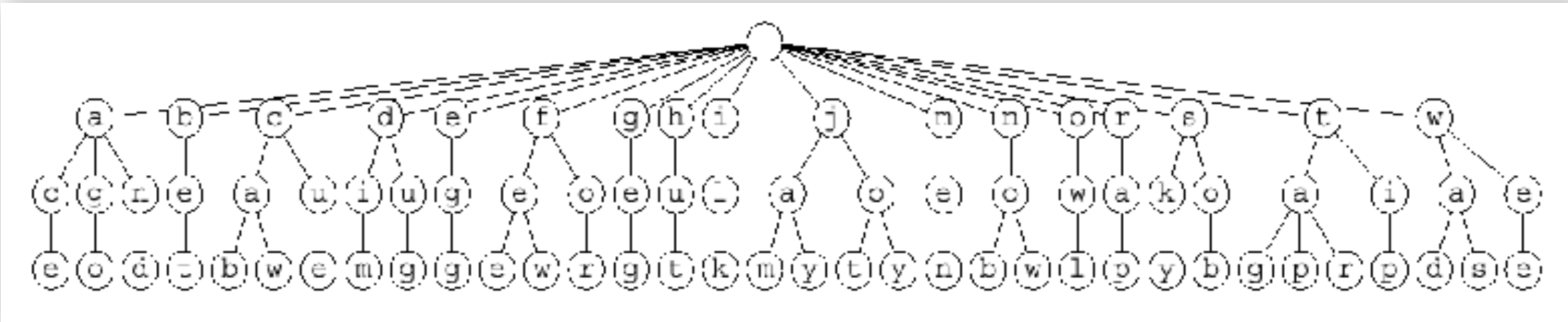
Search hit. Node where search ends has a non-null value.

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



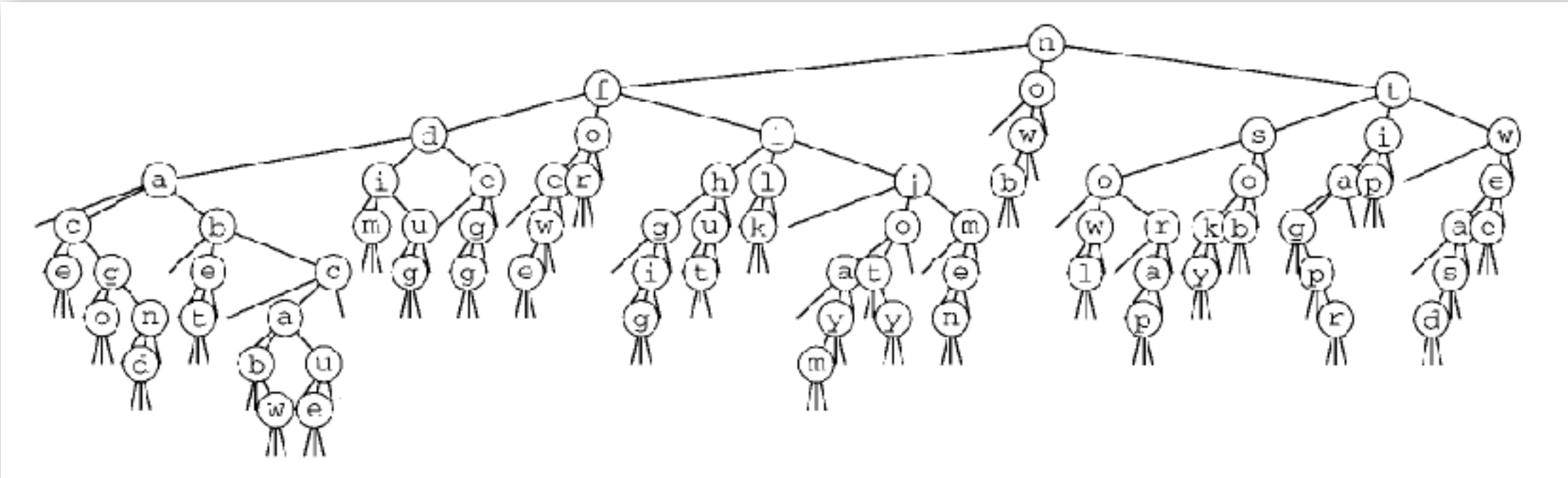
26-way trie vs. TST

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



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

TST. 3 null links in each leaf.



TST (155 null links)

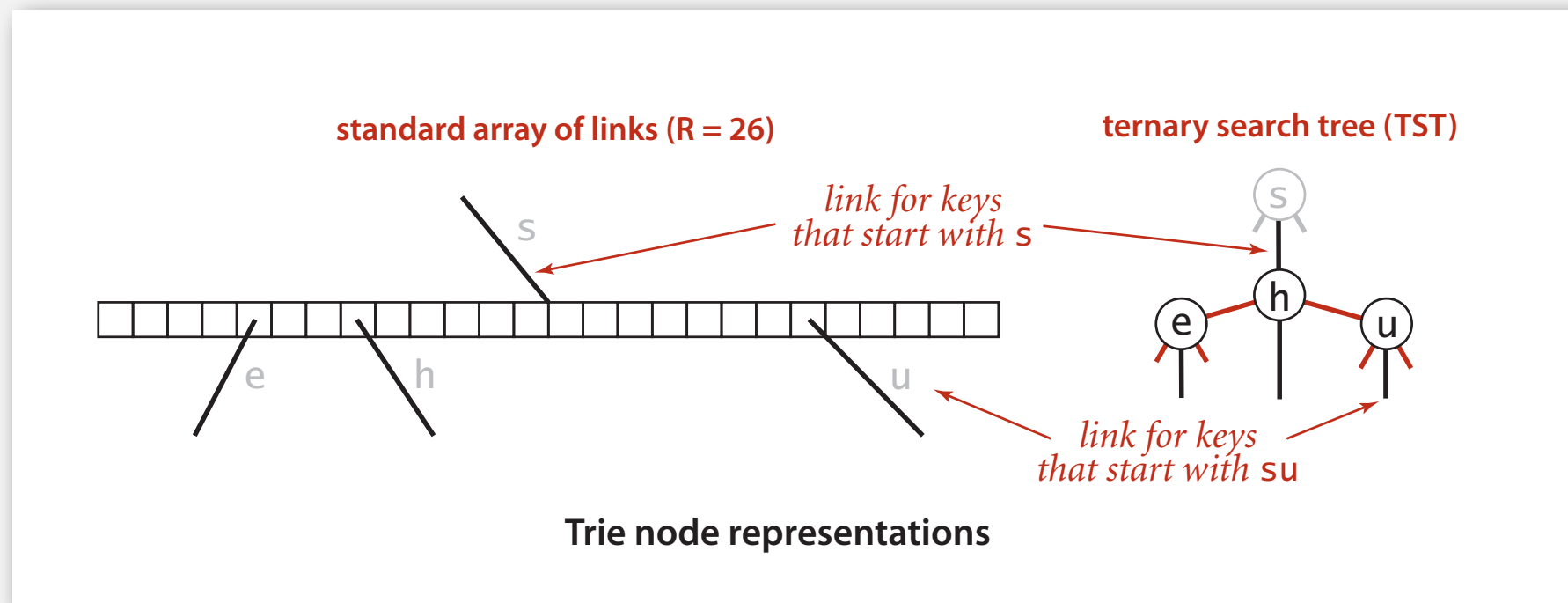
now
for
tip
ilk
dim
tag
jot
sob
nob
sky
hut
ace
bet
men
egg
few
jay
owl
joy
rap
gig
wee
was
cab
wad
caw
cue
fee
tap
ago
tar
jam
dug
and

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

	character accesses (typical case)				dedup	
implementation	search hit	search miss	insert	space (references)	moby.txt	actors.txt
red-black BST	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	$4 N$	1.4	97.4
hashing	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

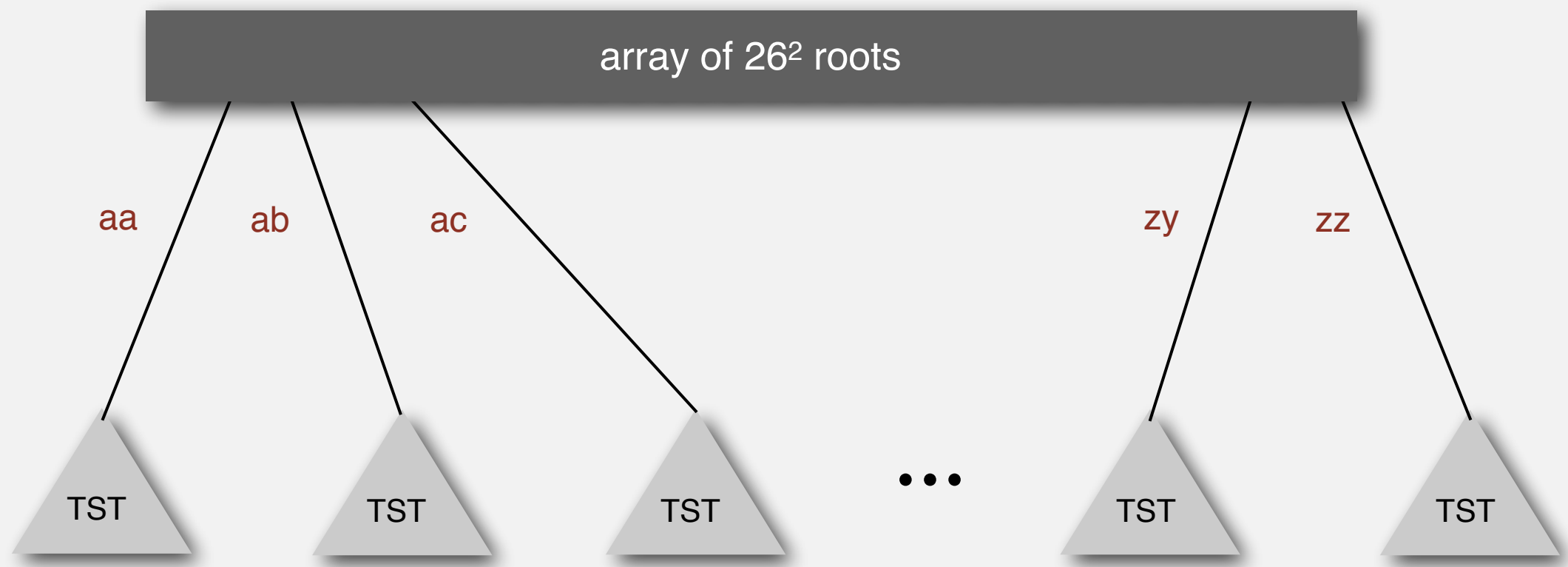
Remark. Can build balanced TSTs via rotations to achieve $L + \log N$ worst-case guarantees.

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

	character accesses (typical case)				dedup	
implementation	search hit	search miss	insert	space (references)	moby.txt	actors.txt
red-black BST	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	$4 N$	1.4	97.4
hashing	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

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**

String symbol table API

Character-based operations. The string symbol table API supports several useful character-based operations.

```
by sea sells she shells shore the
```

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

```
public class StringST<Value>
```

```
    StringST()
```

create a symbol table with string keys

```
    StringST(Alphabet alpha)
```

*create a symbol table with string keys
whose characters are taken from alpha.*

```
    void put(String key, Value val)
```

*put key-value pair into the symbol table
(remove key from table if value is null)*

```
    Value get(String key)
```

*value paired with key
(null if key is absent)*

```
    void delete(String key)
```

remove key (and its value) from table

```
    boolean contains(String key)
```

is there a value paired with key?

```
    boolean isEmpty()
```

is the table empty?

```
    String longestPrefixOf(String s)
```

return the longest key that is a prefix of s

```
    Iterable<String> keysWithPrefix(String s)
```

all the keys having s as a prefix.

```
    Iterable<String> keysThatMatch(String s)
```

*all the keys that match s (where .
matches any character).*

```
    int size()
```

number of key-value pairs in the table

```
    Iterable<String> keys()
```

all the keys in the symbol table

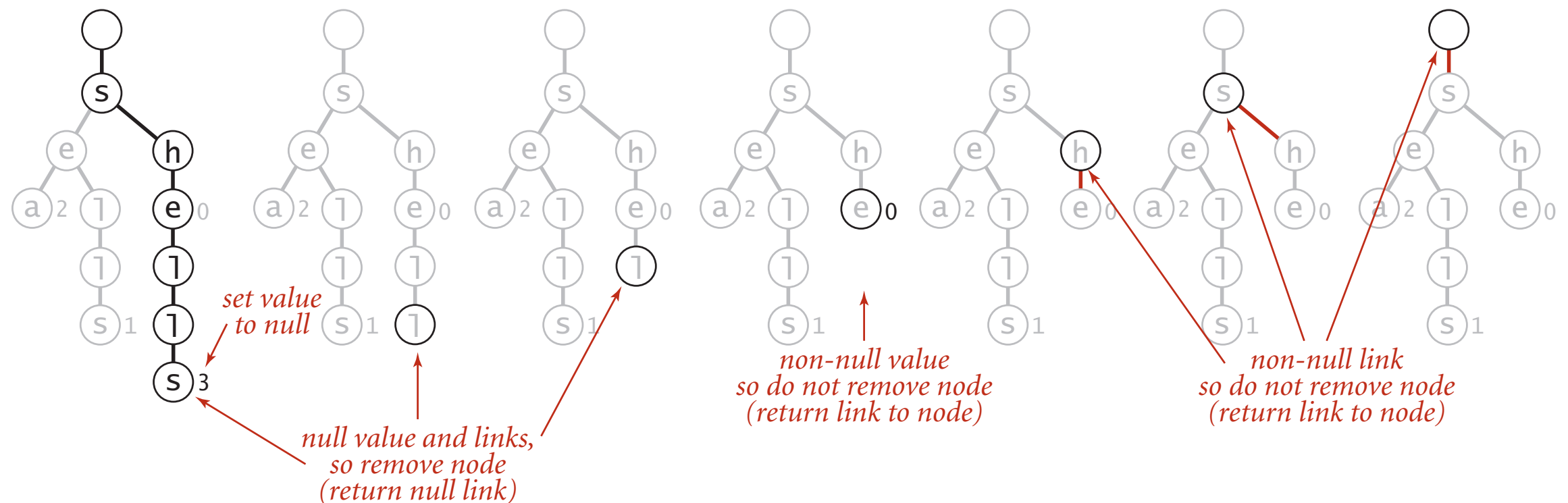
Remark. Can also add other ordered ST methods, e.g., `floor()` and `rank()`.

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

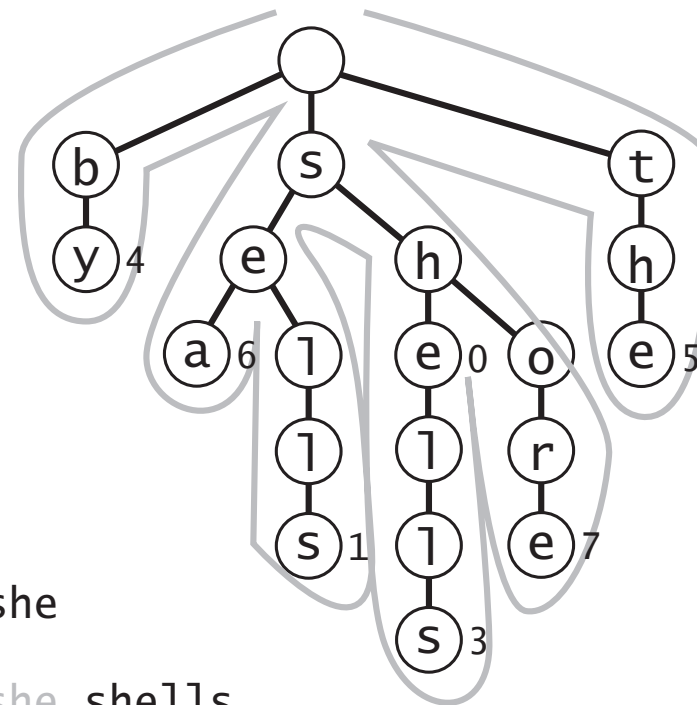
Ordered iteration

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("");
```

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



Collecting the keys in a trie (trace)

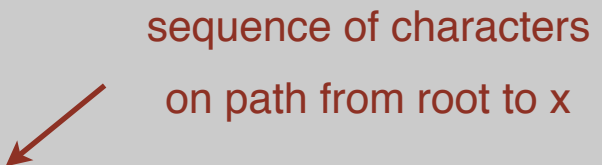
Ordered iteration: Java implementation

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.

```
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);
}
```



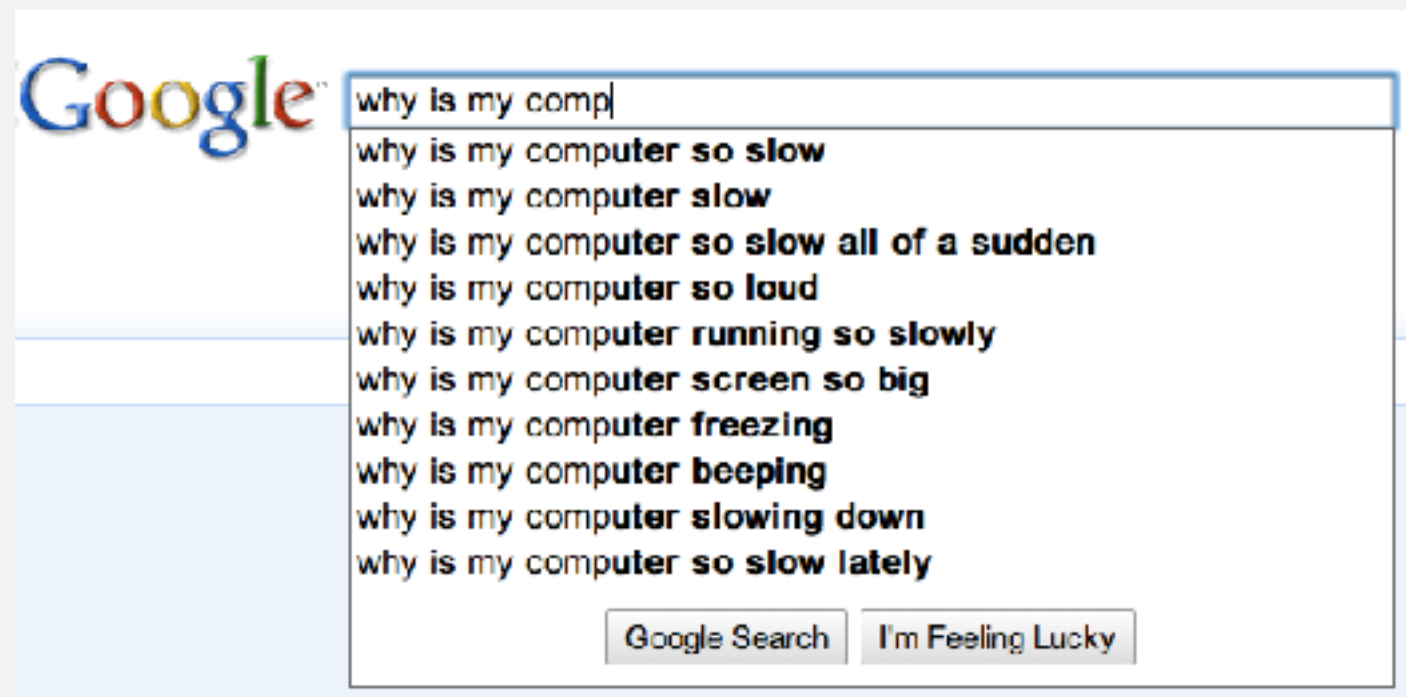
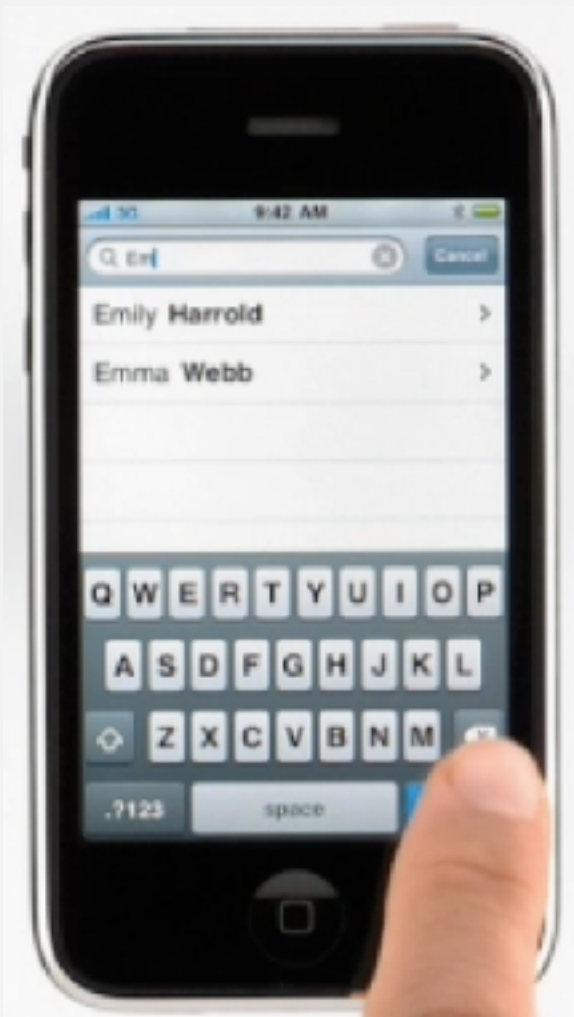
sequence of characters
on path from root to x

Prefix matches

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

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

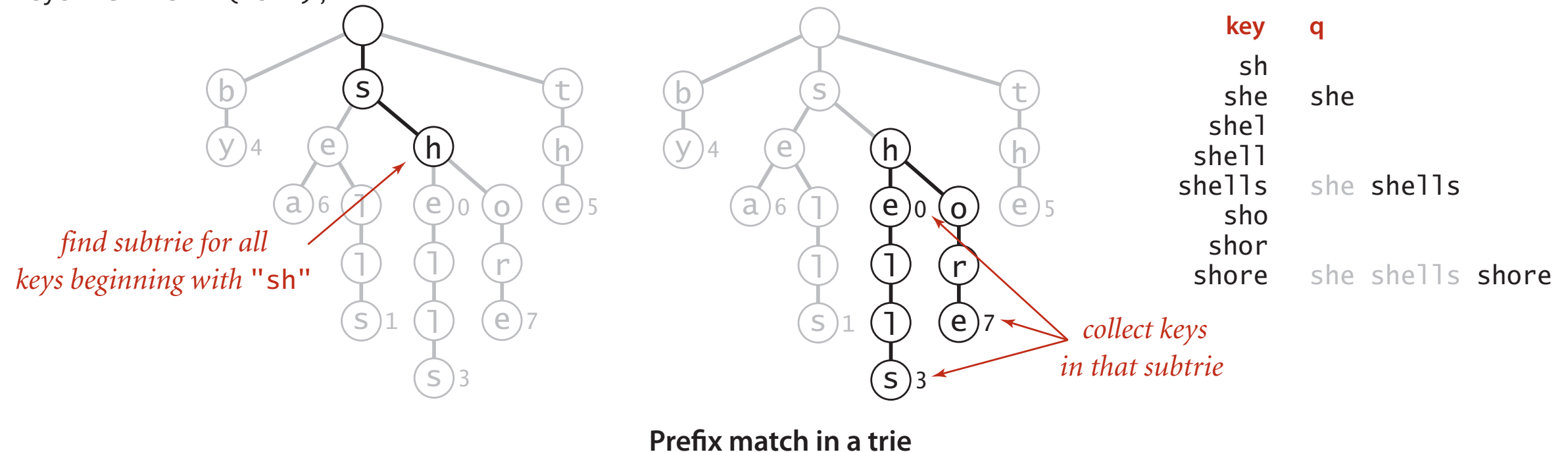
- User types characters one at a time.
- System reports all matching strings.



Prefix matches

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

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

Wildcard matches

Use wildcard `.` to match any character in alphabet.

coalizer
coberger
codifier
cofaster
cofather
cognizer
cohelper
colander
coleader
...
compiler
...
composer
computer
cowkeeper

`co....er`

acresce
acroach
acuracy
octarch
science
scranch
scratch
scrauch
screich
scrinch
scritch
scrunch
scudick
scutock

`.c...c.`

Wildcard matches

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

```
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"

← represented as 32-bit binary number
for IPv4 (instead of string)

`prefix("128.112.136.11") = "128.112.136"`

`prefix("128.166.123.45") = "128"`

Note. Not the same as floor.

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"
```

← represented as 32-bit binary number
for IPv4 (instead of string)

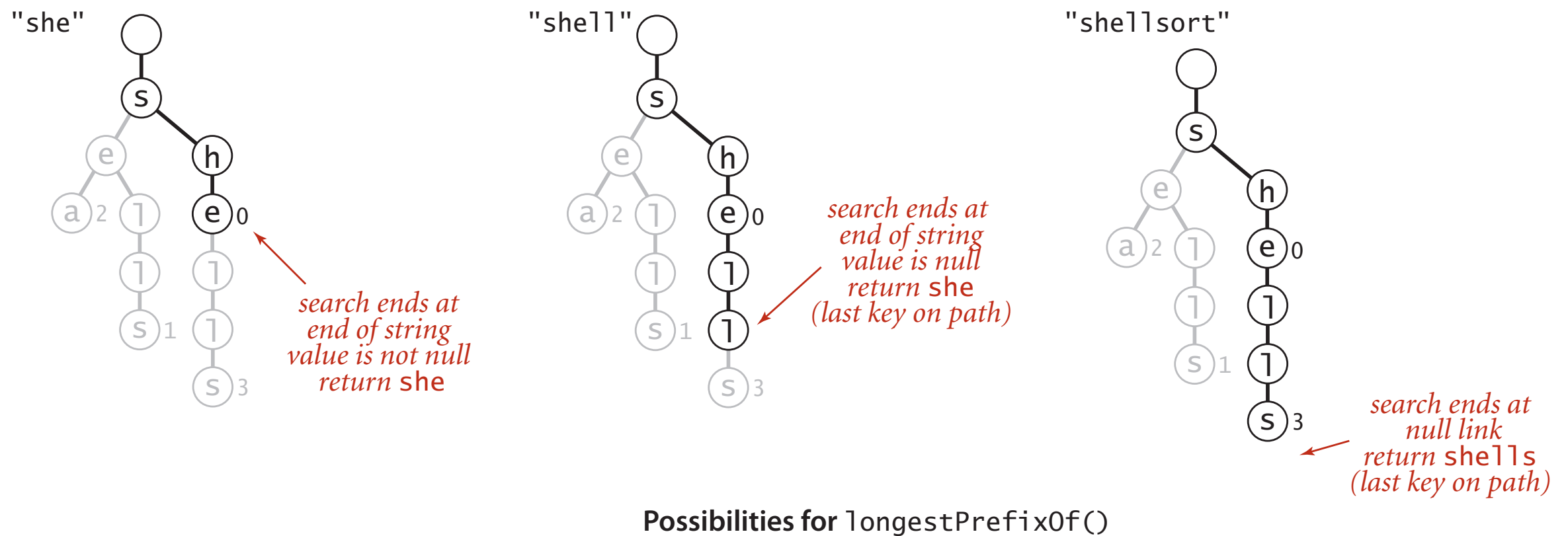
Note. Not the same as floor.

```
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.



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.

```
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.

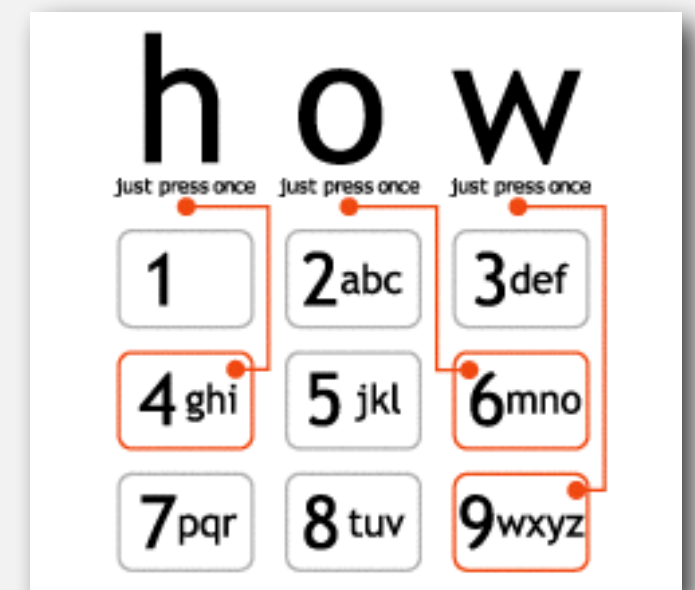
Multi-tap input. Enter a letter by repeatedly pressing a key until the desired letter appears.

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
- T9: 4 3 5 5 6



www.t9.com

Compressing a trie

Collapsing 1-way branches at bottom.

Internal node stores character; leaf node stores suffix (or full key).

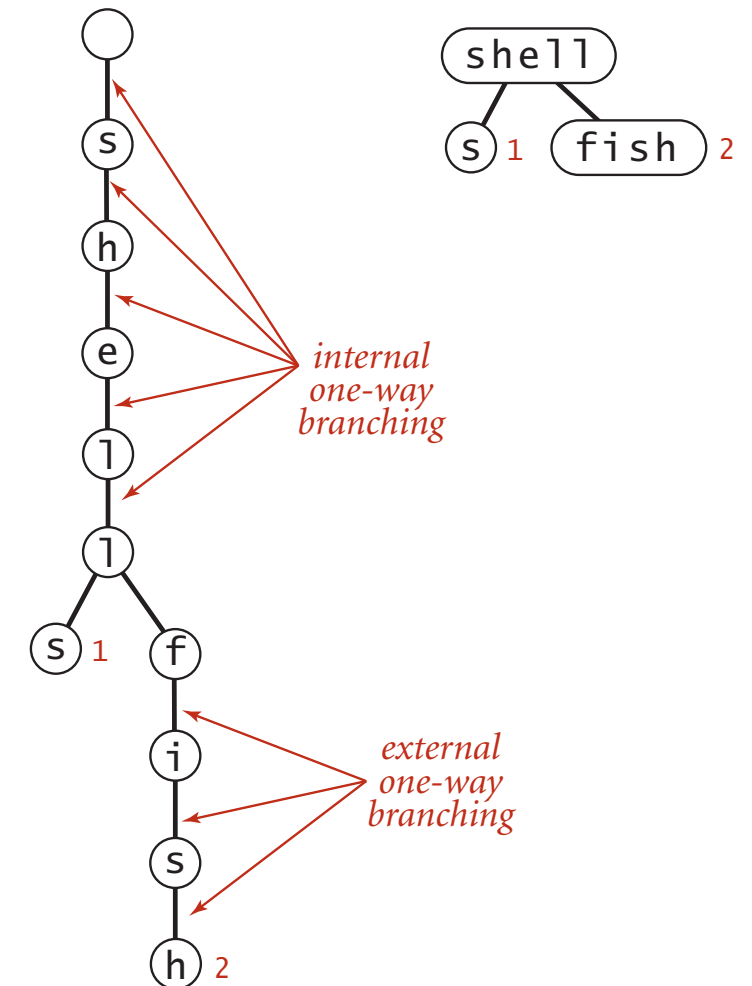
Collapsing interior 1-way branches.

Node stores a sequence of characters.

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

standard
trie

no one-way
branching



Removing one-way branching in a trie

String symbol table implementation cost summary

implementation	character accesses (typical case)			
	search hit	search miss	insert	space (links)
red-black BST	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	$4 N$
hashing	L	L	L	$4 N$ to $16 N$
R-way trie	L	$\log_R N$	L	$(R + 1) N$
TST	$L + \ln N$	$\ln N$	$L + \ln N$	$4 N$
TST with R^2	$L + \ln N$	$\ln N$	$L + \ln N$	$4 N + R^2$
R-way with no 1-way	$L + \lg_R N$	$\log_R N$	$\log_R N$	$R N$
TST with no 1-way	$L + \ln N$	$\ln N$	$\ln N$	$4 N$

Challenge met.

- Efficient performance for arbitrarily long keys.
- Search time is independent of key length!

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 (!!!)