

## 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

use array accesses to make R-way decisions  
(instead of binary decisions)

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

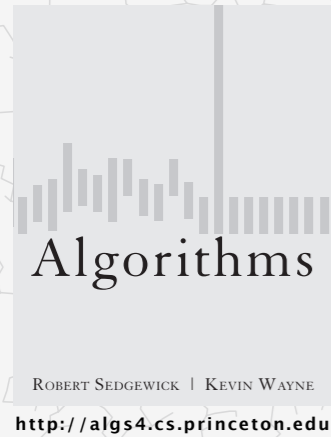
Parameters

- $N$  = 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.

## Tries



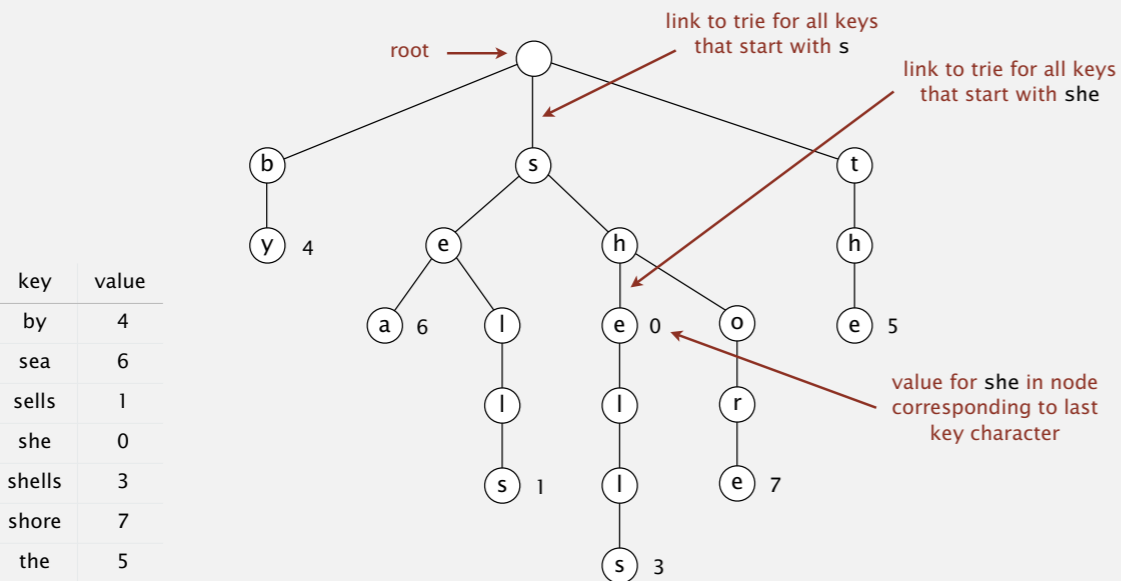
## 5.2 TRIES

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

## 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)

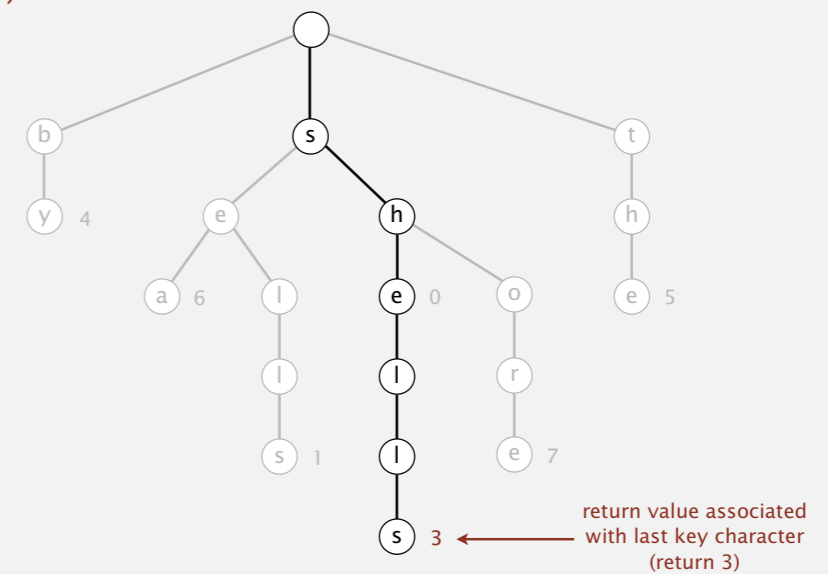


## Search in a trie

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")**

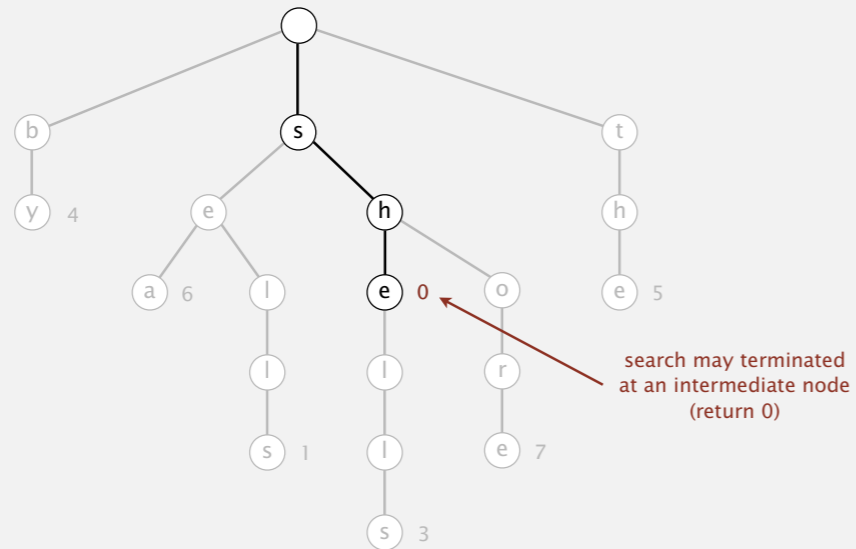


## Search in a trie

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")



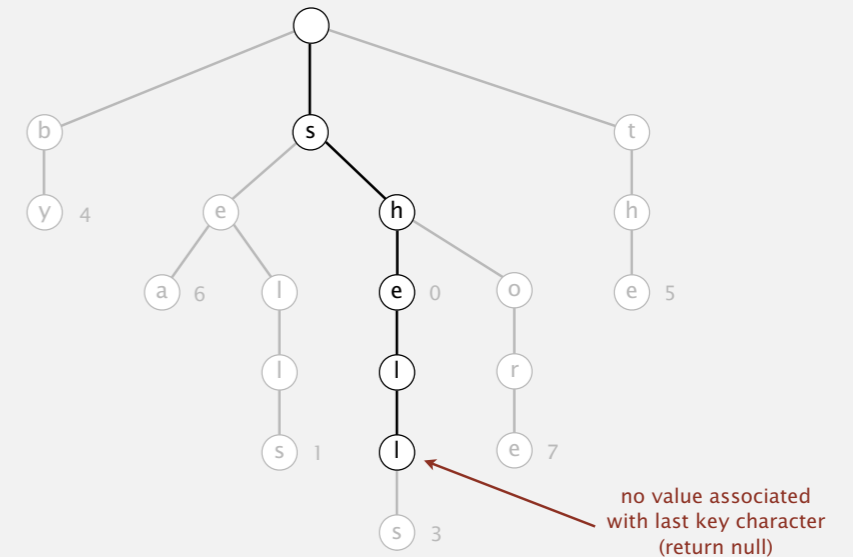
9

## 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")



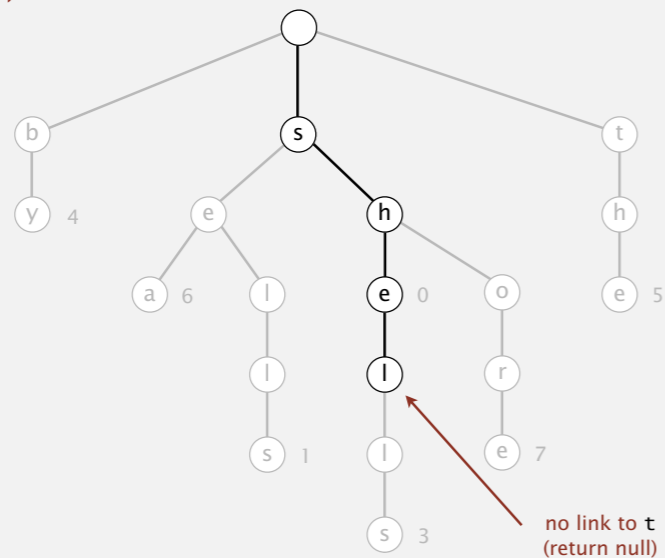
10

## 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")



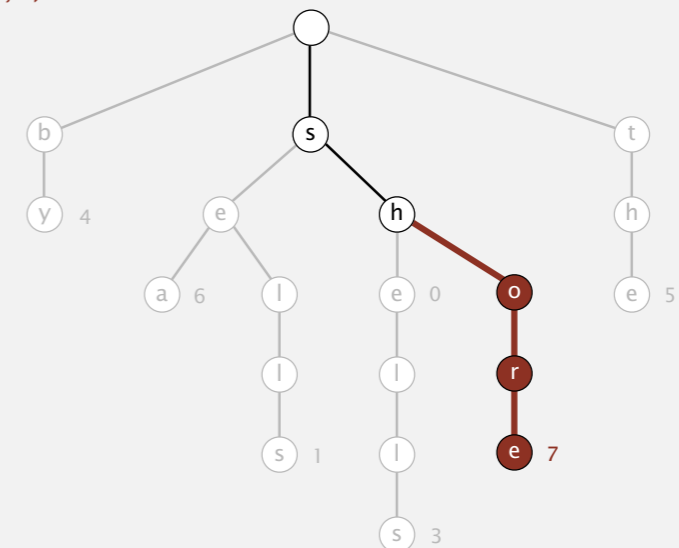
11

## 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)



12

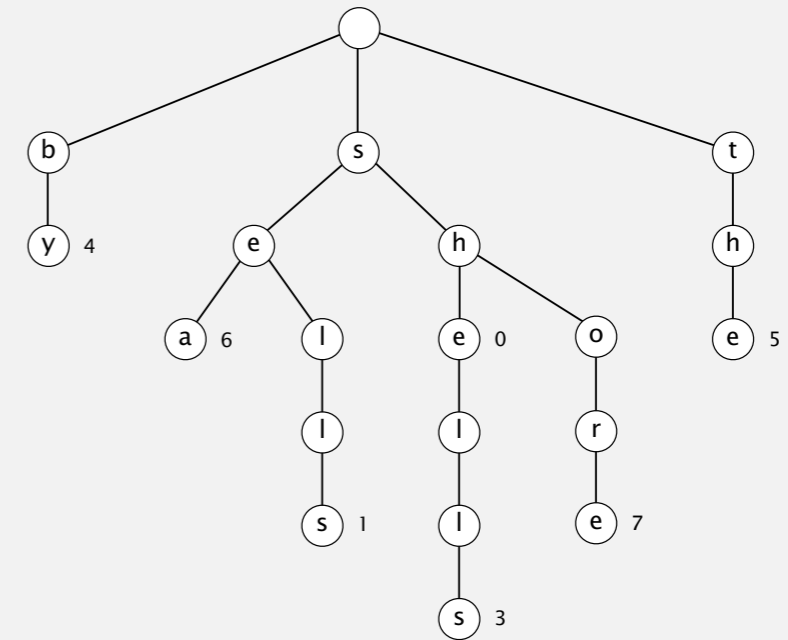
## Trie construction demo

trie



## Trie construction demo

trie

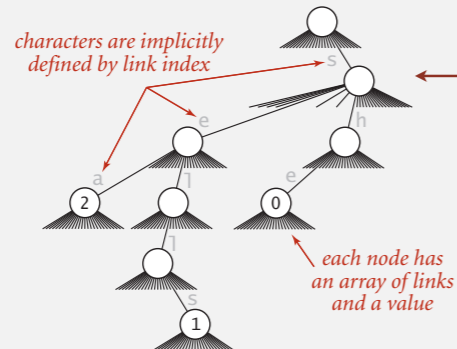
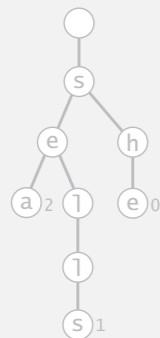


## Trie representation: Java implementation

**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

## 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;
    }
}
```











## 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;
}
}

```

33

## String symbol table implementation cost summary

implementation	character accesses (typical case)			space (references)	dedup	
	search hit	search miss	insert		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	$L + \ln N$	$\ln N$	$L + \ln N$	$4N$	0.72	38.7

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

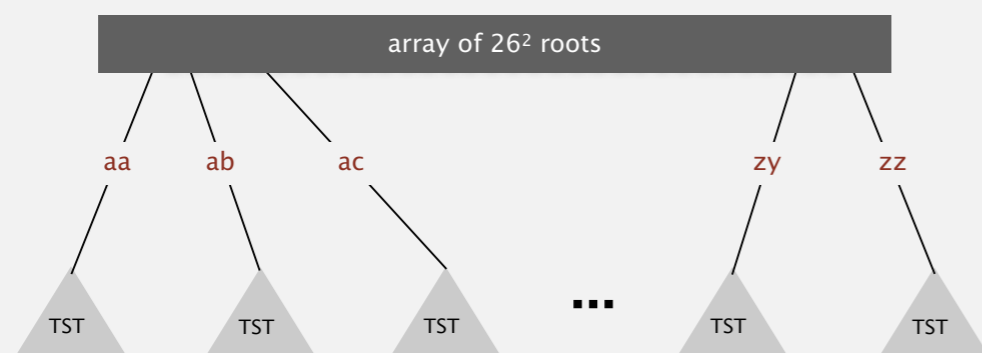
**Bottom line.** TST is as fast as hashing (for string keys), space efficient.

34

## 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?

35

## String symbol table implementation cost summary

implementation	character accesses (typical case)			space (references)	dedup	
	search hit	search miss	insert		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	$L + \ln N$	$\ln N$	$L + \ln N$	$4N$	0.72	38.7
TST with $R^2$	$L + \ln N$	$\ln N$	$L + \ln N$	$4N + R^2$	0.51	32.7

**Bottom line.** Faster than hashing for our benchmark client.

36

## 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.
- 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]

37

## 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.

39



## 5.2 TRIES

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

ROBERT SEDGWICK | KEVIN WAYNE  
<http://algs4.cs.princeton.edu>

## 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().

40

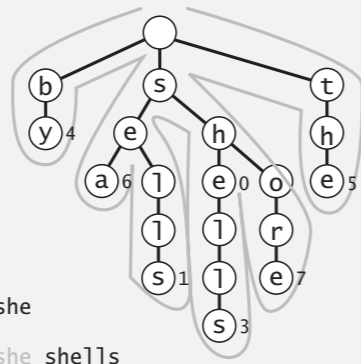
## Warmup: ordered iteration

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.

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



## 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

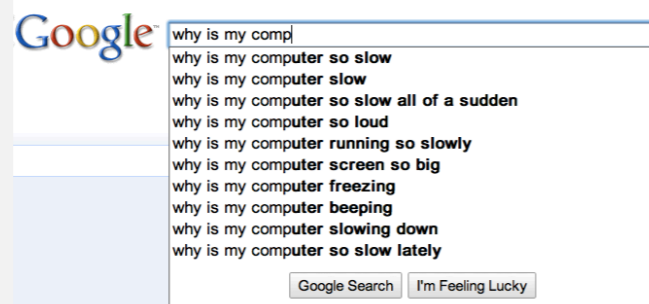
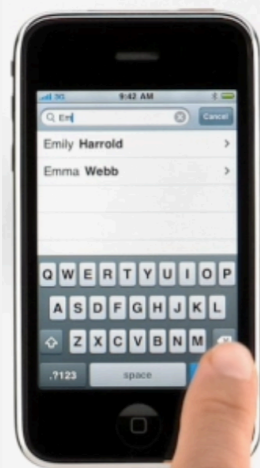
or use StringBuilder

## 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.

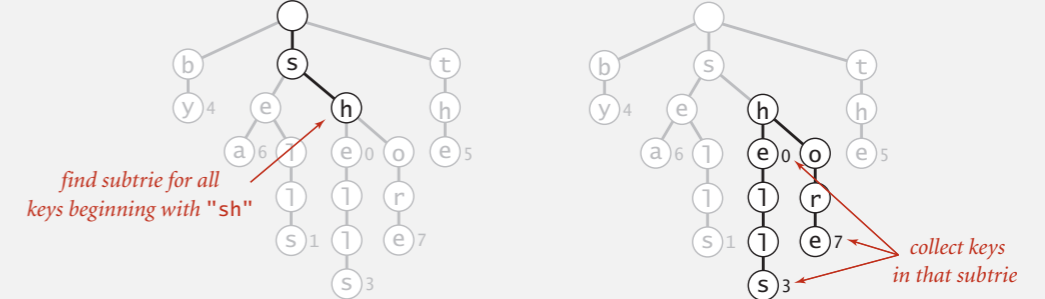
- User types characters one at a time.
- System reports all matching strings.



## 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;
}
```

key	queue
sh	
she	she
shell	
shells	she shells
sho	
shor	
shore	she shells shore

root of subtrie for all strings beginning with given prefix

## 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"  
 "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)

`LongestPrefixOf("128.112.136.11") = "128.112.136"`  
`LongestPrefixOf("128.112.100.16") = "128.112"`  
`LongestPrefixOf("128.166.123.45") = "128"`

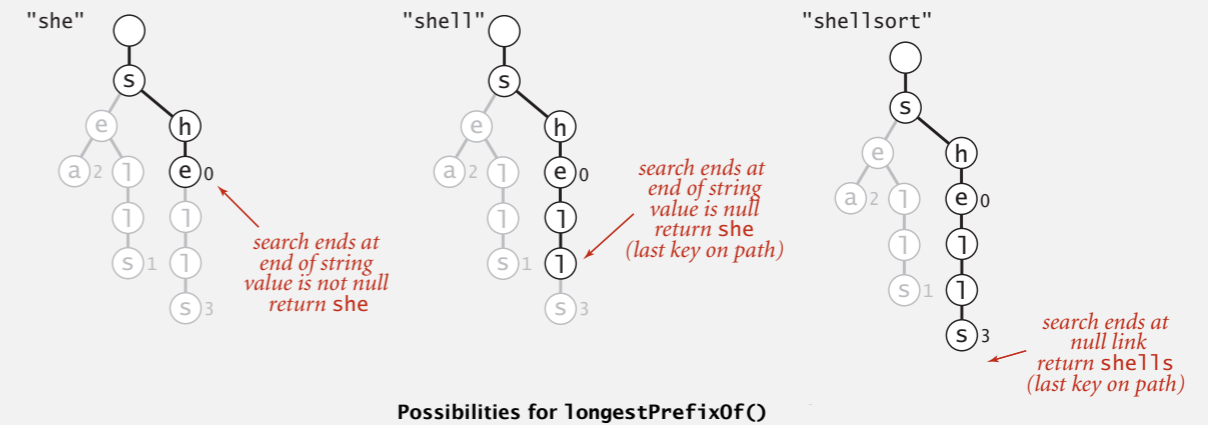
**Note.** Not the same as floor: `floor("128.112.100.16") = "128.112.055.15"`

45

## 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.



46

## 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);
}
```

47

## T9 texting

**Goal.** Type text messages on a phone keypad.

**Multi-tap input.** Enter a letter by repeatedly pressing a key.

**Ex.** hello: 4 4 3 3 5 5 5 5 5 5 6 6 6

**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: 4 3 5 5 6



www.t9.com

**Q.** How to implement?

48

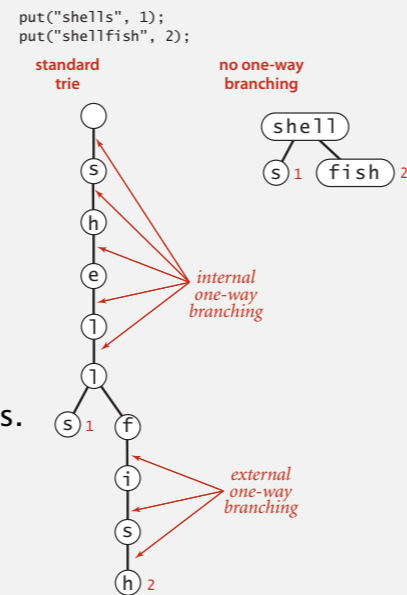
## Patricia trie

**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.

### 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.

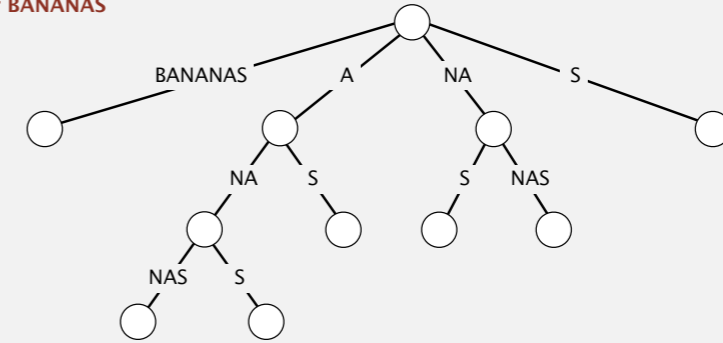
51

## Suffix 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).

52

## 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 (!!!)

53