BBM 202 - ALGORITHMS

HACETTEPE UNIVERSITY

DEPT. OF COMPUTER ENGINEERING

ERKUT ERDEM

STRING SORTS, TRIES

May. 9, 2013

Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

String Sorts, Tries

- String sorts
- Key-indexed counting
- LSD radix sort
- MSD radix sort
- 3-way radix quicksort
- Suffix arrays

- Tries

String processing

String. Sequence of characters.

Important fundamental abstraction.

- Information processing.
- Genomic sequences.
- Communication systems (e.g., email).
- Programming systems (e.g., Java programs).
- …

> *" The digital information that underlies biochemistry, cell biology, and development can be represented by a simple string of G's, A's, T's and C's. This string is the root data structure of an organism's biology. "* — *M. V. Olson*

The char data type

C char data type. Typically an 8-bit integer.

- Supports 7-bit ASCII.
- Need more bits to represent certain characters.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	“	#	$	%	&	‘	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Hexadecimal to ASCII conversion table

Java char data type. A 16-bit unsigned integer.

- Supports original 16-bit Unicode.
- Supports 21-bit Unicode 3.0 (awkwardly).

A	á	∂	𝔊
U+0041	U+00E1	U+2202	U+1D50A

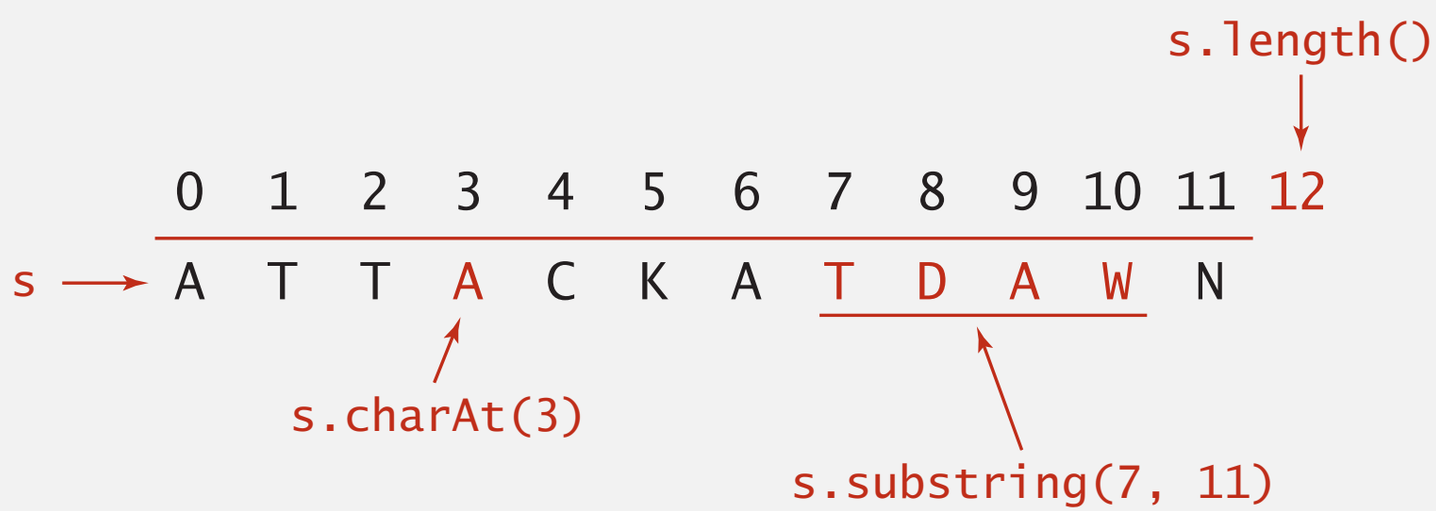
Unicode characters

I (heart) Unicode

The String data type

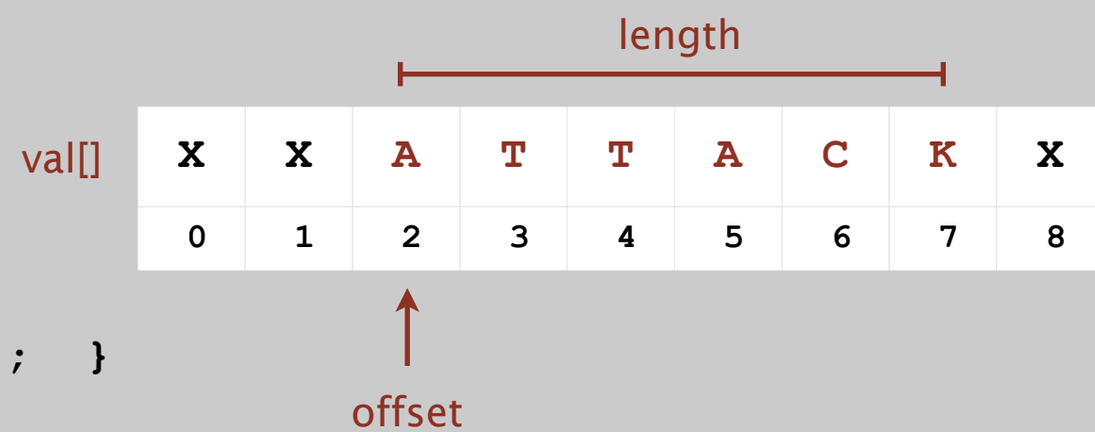
String data type. Sequence of characters (immutable).

Length. Number of characters.
Indexing. Get the ith character.
Substring extraction. Get a contiguous sequence of characters.
String concatenation. Append one character to end of another string.



The String data type: Java implementation

```
public final class String implements Comparable<String>
{
   private char[] val;    // characters
   private int offset;    // index of first char in array
   private int length;    // length of string
   private int hash;      // cache of hashCode()

   public int length()
   {  return length; }

   public char charAt(int i)
   {  return value[i + offset];  }

   private String(int offset, int length, char[] val)
   {
      this.offset = offset;
      this.length = length;
      this.val    = val;
   }

   public String substring(int from, int to)
   {  return new String(offset + from, to - from, val);  }
   …
```



copy of reference to original char array

The String data type: performance

String data type. Sequence of characters (immutable).

Underlying implementation. Immutable `char[]` array, offset, and length.

	String	
operation	guarantee	extra space
`length()`	1	1
`charAt()`	1	1
`substring()`	1	1
`concat()`	N	N

Memory. $40 + 2N$ bytes for a virgin `String` of length N.

can use `byte[]` or `char[]` instead of `String` to save space

(but lose convenience of `String` data type)

The StringBuilder data type

StringBuilder data type. Sequence of characters (mutable).

Underlying implementation. Resizing `char[]` array and length.

	String		StringBuilder	
operation	guarantee	extra space	guarantee	extra space
`length()`	1	1	1	1
`charAt()`	1	1	1	1
`substring()`	1	1	N	N
`concat()`	N	N	1 *	1 *

* amortized

Remark. `StringBuffer` data type is similar, but thread safe (and slower).

String vs. StringBuilder

Q. How to efficiently reverse a string?

A.

```
public static String reverse(String s)
{
   String rev = "";
   for (int i = s.length() - 1; i >= 0; i--)
      rev += s.charAt(i);
   return rev;
}
```

quadratic time

B.

```
public static String reverse(String s)
{
   StringBuilder rev = new StringBuilder();
   for (int i = s.length() - 1; i >= 0; i--)
      rev.append(s.charAt(i));
   return rev.toString();
}
```

linear time

String challenge: array of suffixes

Q. How to efficiently form array of suffixes?

input string

a	a	c	a	a	g	t	t	t	a	c	a	a	g	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

suffixes

0	a	a	c	a	a	g	t	t	t	a	c	a	a	g	c
1	a	c	a	a	g	t	t	t	a	c	a	a	g	c	
2	c	a	a	g	t	t	t	a	c	a	a	g	c		
3	a	a	g	t	t	t	a	c	a	a	g	c			
4	a	g	t	t	t	a	c	a	a	g	c				
5	g	t	t	t	a	c	a	a	g	c					
6	t	t	t	a	c	a	a	g	c						
7	t	t	a	c	a	a	g	c							
8	t	a	c	a	a	g	c								
9	a	c	a	a	g	c									
10	c	a	a	g	c										
11	a	a	g	c											
12	a	g	c												
13	g	c													
14	c														

String vs. StringBuilder

Q. How to efficiently form array of suffixes?

A.

```
public static String[] suffixes(String s)
{
   int N = s.length();
   String[] suffixes = new String[N];
   for (int i = 0; i < N; i++)
      suffixes[i] = s.substring(i, N);
   return suffixes;
}
```

linear time and linear space

B.

```
public static String[] suffixes(String s)
{
   int N = s.length();
   StringBuilder sb = new StringBuilder(s);
   String[] suffixes = new String[N];
   for (int i = 0; i < N; i++)
      suffixes[i] = sb.substring(i, N);
   return suffixes;
}
```

quadratic time and quadratic space

Longest common prefix

Q. How long to compute length of longest common prefix?

p	r	e	f	e	t	c	h
0	1	2	3	4	5	6	7
p	r	e	f	i	x		

```
public static int lcp(String s, String t)
{
   int N = Math.min(s.length(), t.length());
   for (int i = 0; i < N; i++)
      if (s.charAt(i) != t.charAt(i))
         return i;
   return N;
}
```

linear time (worst case)

sublinear time (typical case)

Running time. Proportional to length D of longest common prefix.

Remark. Also can compute `compareTo()` in sublinear time.

Alphabets

Digital key. Sequence of digits over fixed alphabet.

Radix. Number of digits R in alphabet.

name	R()	lgR()	characters
BINARY	2	1	01
OCTAL	8	3	01234567
DECIMAL	10	4	0123456789
HEXADECIMAL	16	4	0123456789ABCDEF
DNA	4	2	ACTG
LOWERCASE	26	5	abcdefghijklmnopqrstuvwxyz
UPPERCASE	26	5	ABCDEFGHIJKLMNOPQRSTUVWXYZ
PROTEIN	20	5	ACDEFGHIKLMNPQRSTVWY
BASE64	64	6	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdef ghijklmnopqrstuvwxyz0123456789+/
ASCII	128	7	*ASCII characters*
EXTENDED_ASCII	256	8	*extended ASCII characters*
UNICODE16	65536	16	*Unicode characters*

String Sorts

- Key-indexed counting
- LSD radix sort
- MSD radix sort
- 3-way radix quicksort
- Suffix arrays

Review: summary of the performance of sorting algorithms

Frequency of operations = key compares.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$N^2 / 2$	$N^2 / 4$	1	yes	`compareTo()`
mergesort	N lg N	N lg N	N	yes	`compareTo()`
quicksort	1.39 N lg N *	1.39 N lg N	c lg N	no	`compareTo()`
heapsort	2 N lg N	2 N lg N	1	no	`compareTo()`

* probabilistic

Lower bound. ~ $N \lg N$ compares required by any compare-based algorithm.

Q. Can we do better (despite the lower bound)?

A. Yes, if we don't depend on key compares.

Key-indexed counting: assumptions about keys

Assumption. Keys are integers between 0 and $R - 1$.

Implication. Can use key as an array index.

Applications.

- Sort string by first letter.
- Sort class roster by section.
- Sort phone numbers by area code.
- Subroutine in a sorting algorithm. [stay tuned]

Remark. Keys may have associated data $\Rightarrow$ can't just count up number of keys of each value.

input

name	*section*
Anderson	2
Brown	3
Davis	3
Garcia	4
Harris	1
Jackson	3
Johnson	4
Jones	3
Martin	1
Martinez	2
Miller	2
Moore	1
Robinson	2
Smith	4
Taylor	3
Thomas	4
Thompson	4
White	2
Williams	3
Wilson	4

keys are small integers

sorted result

(by section)	
Harris	1
Martin	1
Moore	1
Anderson	2
Martinez	2
Miller	2
Robinson	2
White	2
Brown	3
Davis	3
Jackson	3
Jones	3
Taylor	3
Williams	3
Garcia	4
Johnson	4
Smith	4
Thomas	4
Thompson	4
Wilson	4

Key-indexed counting demo

Goal. Sort an array `a[]` of N integers between 0 and $R - 1$.

R=6

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

use a for 0
b for 1
c for 2
d for 3
e for 4
f for 5

Key-indexed counting demo

Goal. Sort an array `a[]` of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

count frequencies

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

offset by 1
[stay tuned]

r	count[r]
a	0
b	2
c	3
d	1
e	2
f	1
-	3

Key-indexed counting demo

Goal. Sort an array `a[]` of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

compute cumulates

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	0
b	2
c	5
d	6
e	8
f	9
-	12

6 keys < d, 8 keys < e

so d's go in a[6] and a[7]

Key-indexed counting demo

Goal. Sort an array `a[]` of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

move items

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	0
b	2
c	5
d	6
e	8
f	9
-	12

i	aux[i]
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	

Key-indexed counting demo

Goal. Sort an array `a[]` of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

move items

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	0
b	2
c	5
d	7
e	8
f	9
-	12

i	aux[i]
0	
1	
2	
3	
4	
5	
6	d
7	
8	
9	
10	
11	

Key-indexed counting demo

Goal. Sort an array `a[]` of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

move items

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	1
b	2
c	5
d	7
e	8
f	9
-	12

i	aux[i]
0	a
1	
2	
3	
4	
5	
6	d
7	
8	
9	
10	
11	

Key-indexed counting demo

Goal. Sort an array `a[]` of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

move items

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	1
b	2
c	6
d	7
e	8
f	9
-	12

i	aux[i]
0	a
1	
2	
3	
4	
5	c
6	d
7	
8	
9	
10	
11	

Key-indexed counting demo

Goal. Sort an array `a[]` of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

move items

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	1
b	2
c	6
d	7
e	8
f	10
-	12

i	aux[i]
0	a
1	
2	
3	
4	
5	c
6	d
7	
8	
9	f
10	
11	

Key-indexed counting demo

Goal. Sort an array `a[]` of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

move items

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	1
b	2
c	6
d	7
e	8
f	11
-	12

i	aux[i]
0	a
1	
2	
3	
4	
5	c
6	d
7	
8	
9	f
10	f
11	

Key-indexed counting demo

Goal. Sort an array `a[]` of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

move items

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	1
b	3
c	6
d	7
e	8
f	11
-	12

i	aux[i]
0	a
1	
2	b
3	
4	
5	c
6	d
7	
8	
9	f
10	f
11	

Key-indexed counting demo

Goal. Sort an array `a[]` of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

move items

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	1
b	3
c	6
d	8
e	8
f	11
-	12

i	aux[i]
0	a
1	
2	b
3	
4	
5	c
6	d
7	d
8	
9	f
10	f
11	

Key-indexed counting demo

Goal. Sort an array `a[]` of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

move items

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	1
b	4
c	6
d	8
e	8
f	11
-	12

i	aux[i]
0	a
1	
2	b
3	b
4	
5	c
6	d
7	d
8	
9	f
10	f
11	

Key-indexed counting demo

Goal. Sort an array `a[]` of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

move items

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	1
b	4
c	6
d	8
e	8
f	12
-	12

i	aux[i]
0	a
1	
2	b
3	b
4	
5	c
6	d
7	d
8	
9	f
10	f
11	f

Key-indexed counting demo

Goal. Sort an array `a[]` of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

move items

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	1
b	5
c	6
d	8
e	8
f	12
-	12

i	aux[i]
0	a
1	
2	b
3	b
4	b
5	c
6	d
7	d
8	
9	f
10	f
11	f

Key-indexed counting demo

Goal. Sort an array `a[]` of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

move items

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	1
b	5
c	6
d	8
e	9
f	12
-	12

i	aux[i]
0	a
1	
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

Key-indexed counting demo

Goal. Sort an array `a[]` of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

move items

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	2
b	5
c	6
d	8
e	9
f	12
-	12

i	aux[i]
0	a
1	a
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

Key-indexed counting demo

Goal. Sort an array `a[]` of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

move items

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	2
b	5
c	6
d	8
e	9
f	12
-	12

i	aux[i]
0	a
1	a
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

Key-indexed counting demo

Goal. Sort an array `a[]` of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

copy back

i	a[i]
0	a
1	a
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

r	count[r]
a	2
b	5
c	6
d	8
e	9
f	12
-	12

i	aux[i]
0	a
1	a
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

Key-indexed counting demo

Goal. Sort an array `a[]` of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

count frequencies

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

offset by 1
[stay tuned]

r	count[r]
a	0
b	2
c	3
d	1
e	2
f	1
-	3

Key-indexed counting demo

Goal. Sort an array `a[]` of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

compute cumulates

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	0
b	2
c	5
d	6
e	8
f	9
-	12

6 keys < d, 8 keys < e

so d's go in a[6] and a[7]

Key-indexed counting demo

Goal. Sort an array `a[]` of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

move items

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	2
b	5
c	6
d	8
e	9
f	12
-	12

i	aux[i]
0	a
1	a
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

Key-indexed counting demo

Goal. Sort an array `a[]` of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

copy back

i	a[i]
0	a
1	a
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

r	count[r]
a	2
b	5
c	6
d	8
e	9
f	12
-	12

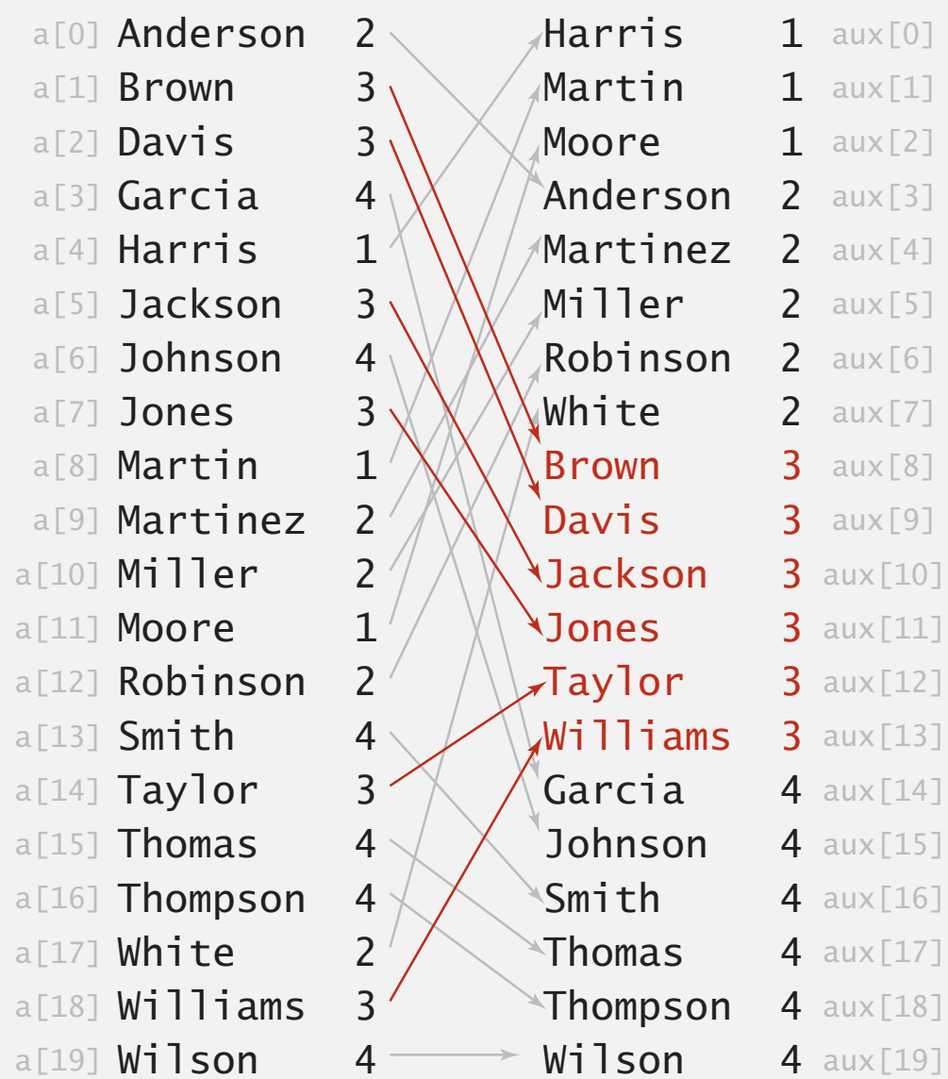
i	aux[i]
0	a
1	a
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

Key-indexed counting: analysis

Proposition. Key-indexed counting uses $\sim 11\,N + 4\,R$ array accesses to sort N items whose keys are integers between 0 and $R - 1$.

Proposition. Key-indexed counting uses extra space proportional to $N + R$.

Stable? ✔



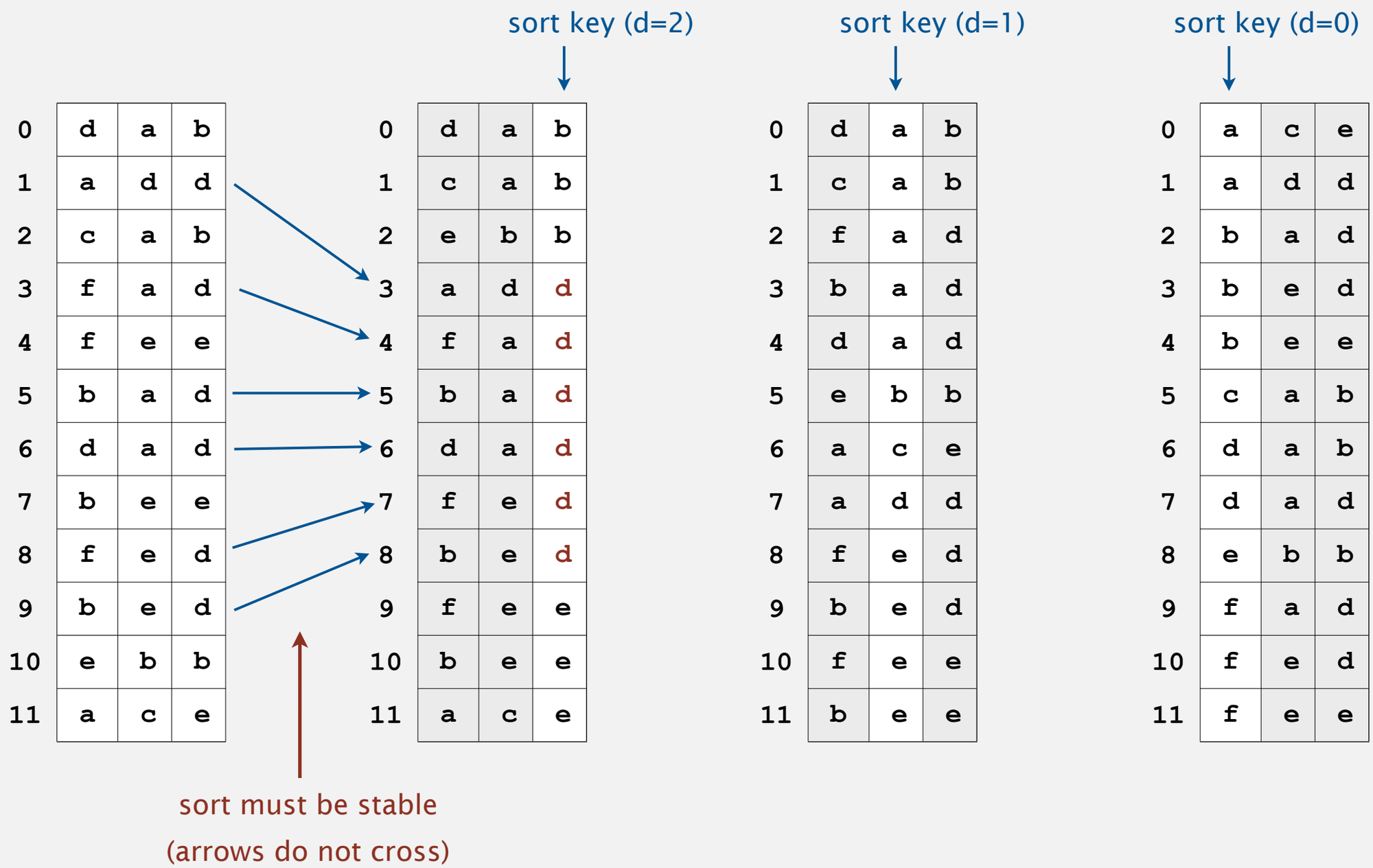
String Sorts

- Key-indexed counting
- LSD radix sort
- MSD radix sort
- 3-way radix quicksort
- Suffix arrays

Least-significant-digit-first string sort

LSD string (radix) sort.

- Consider characters from right to left.
- Stably sort using d^{th} character as the key (using key-indexed counting).



0	d	a	b
1	a	d	d
2	c	a	b
3	f	a	d
4	f	e	e
5	b	a	d
6	d	a	d
7	b	e	e
8	f	e	d
9	b	e	d
10	e	b	b
11	a	c	e

0	d	a	b
1	c	a	b
2	e	b	b
3	a	d	d
4	f	a	d
5	b	a	d
6	d	a	d
7	f	e	d
8	b	e	d
9	f	e	e
10	b	e	e
11	a	c	e

0	d	a	b
1	c	a	b
2	f	a	d
3	b	a	d
4	d	a	d
5	e	b	b
6	a	c	e
7	a	d	d
8	f	e	d
9	b	e	d
10	f	e	e
11	b	e	e

0	a	c	e
1	a	d	d
2	b	a	d
3	b	e	d
4	b	e	e
5	c	a	b
6	d	a	b
7	d	a	d
8	e	b	b
9	f	a	d
10	f	e	d
11	f	e	e

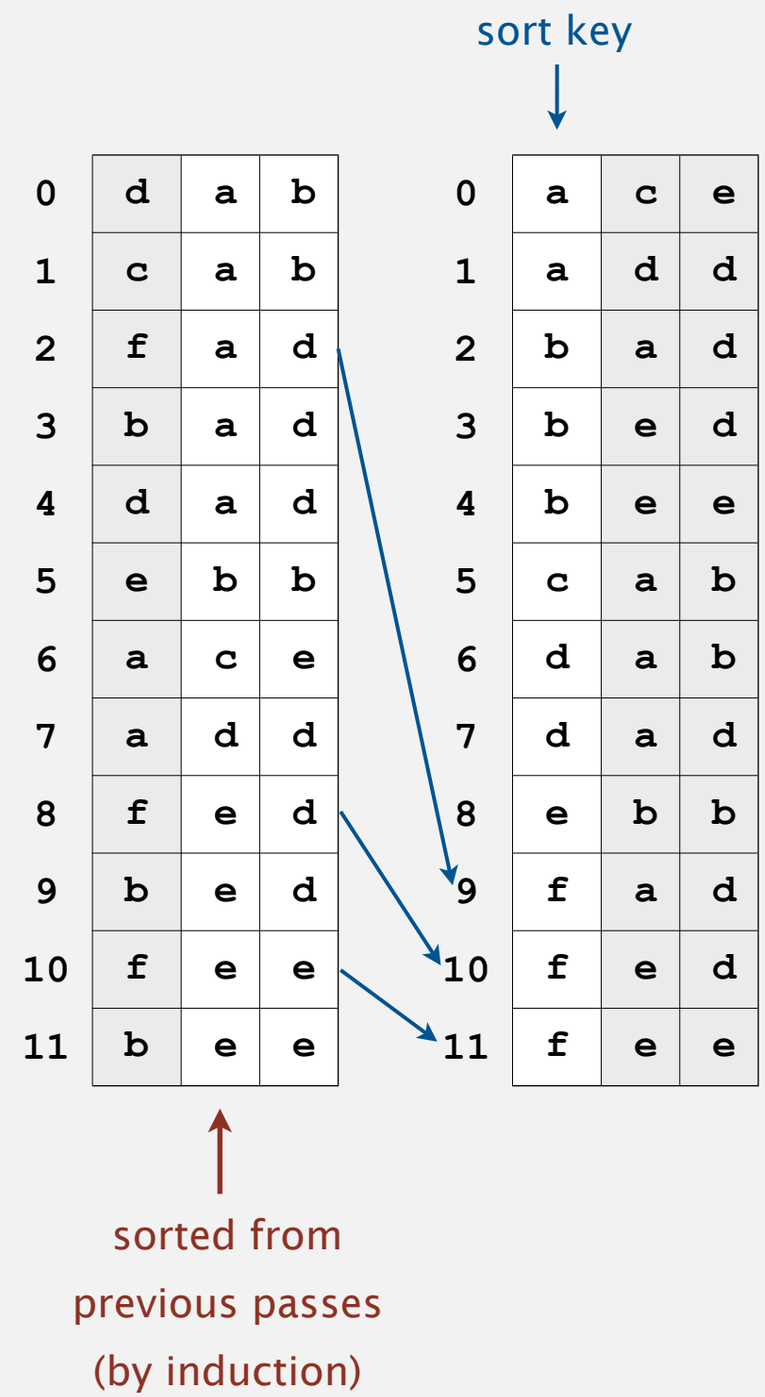
LSD string sort: correctness proof

Proposition. LSD sorts fixed-length strings in ascending order.

Pf. [by induction on i]

After pass i, strings are sorted by last i characters.

- If two strings differ on sort key, key-indexed sort puts them in proper relative order.
- If two strings agree on sort key, stability keeps them in proper relative order.



LSD string sort: Java implementation

```
public class LSD
{
   public static void sort(String[] a, int W)
   {
      int R = 256;
      int N = a.length;
      String[] aux = new String[N];

      for (int d = W-1; d >= 0; d--)
      {
         int[] count = new int[R+1];
         for (int i = 0; i < N; i++)
            count[a[i].charAt(d) + 1]++;
         for (int r = 0; r < R; r++)
            count[r+1] += count[r];
         for (int i = 0; i < N; i++)
            aux[count[a[i].charAt(d)]++] = a[i];
         for (int i = 0; i < N; i++)
            a[i] = aux[i];
      }
   }
}
```

fixed-length W strings

radix R

do key-indexed counting for each digit from right to left

key-indexed counting

Summary of the performance of sorting algorithms

Frequency of operations.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$N^2 / 2$	$N^2 / 4$	1	yes	`compareTo()`
mergesort	N lg N	N lg N	N	yes	`compareTo()`
quicksort	1.39 N lg N *	1.39 N lg N	c lg N	no	`compareTo()`
heapsort	2 N lg N	2 N lg N	1	no	`compareTo()`
LSD †	2 W N	2 W N	N + R	yes	`charAt()`

* probabilistic

† fixed-length W keys

Q. What if strings do not have same length?

String sorting challenge I

Problem. Sort a huge commercial database on a fixed-length key.

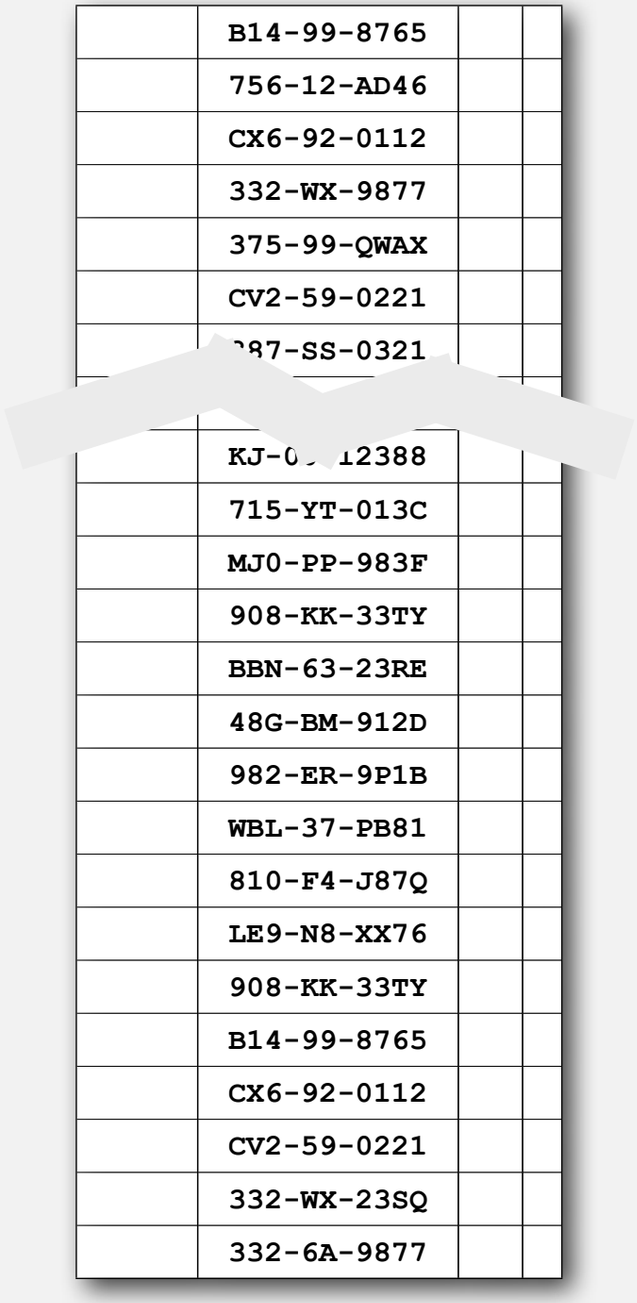
Ex. Account number, date, Social Security number, ...

Which sorting method to use?

- Insertion sort.
- Mergesort.
- Quicksort.
- Heapsort.
- ✓ LSD string sort.

256 (or 65,536) counters;

Fixed-length strings sort in W passes.



String sorting challenge 2a

Problem. Sort one million 32-bit integers.

Ex. Google (or presidential) interview.

Which sorting method to use?

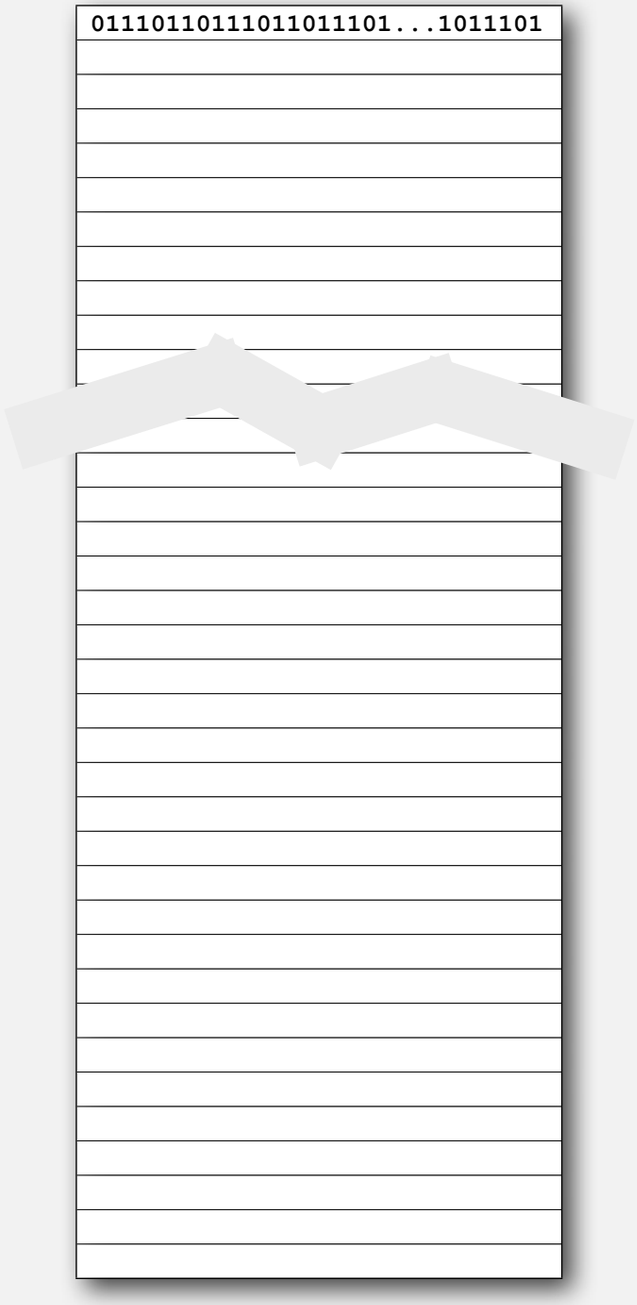
- Insertion sort.
- Mergesort.
- Quicksort.
- Heapsort.
- LSD string sort.

Google CEO Eric Schmidt interviews Barack Obama

String sorting challenge 2b

Problem. Sort huge array of random 128-bit numbers.

Ex. Supercomputer sort, internet router.



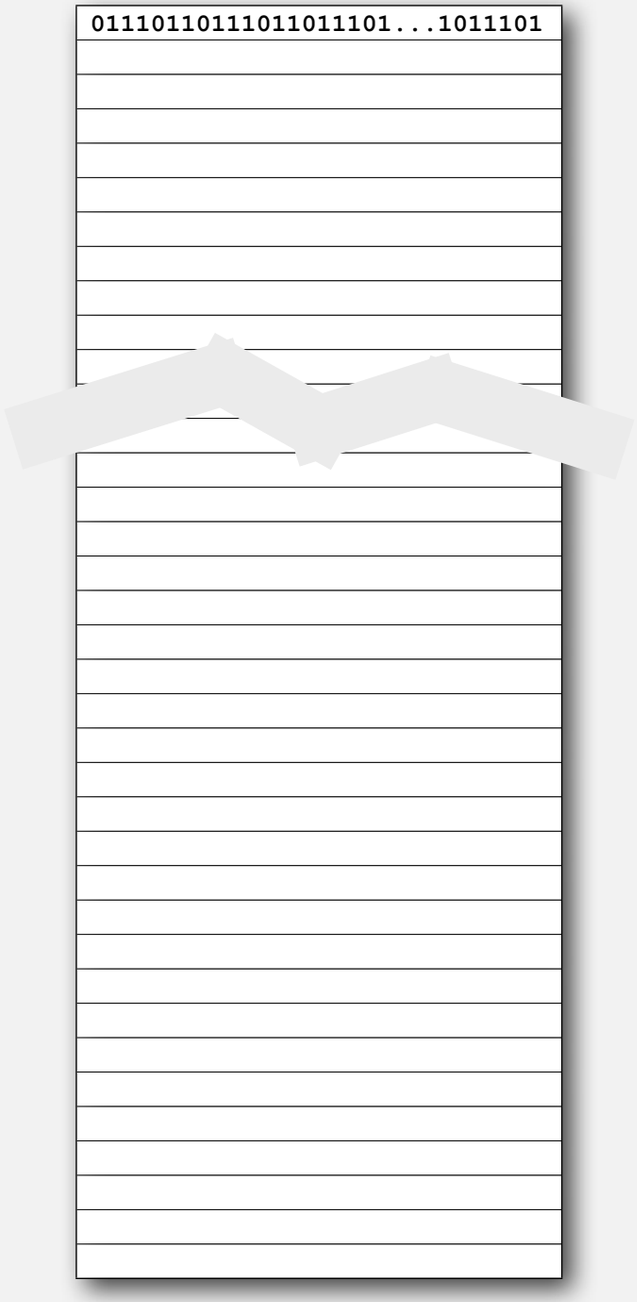
Which sorting method to use?

- Insertion sort.
- Mergesort.
- Quicksort.
- Heapsort.
- LSD string sort.

String sorting challenge 2b

Problem. Sort huge array of random 128-bit numbers.

Ex. Supercomputer sort, internet router.



Which sorting method to use?

- Insertion sort.
- Mergesort.
- Quicksort.
- Heapsort.
- ✓ LSD string sort.

Divide each word into eight 16-bit "chars"

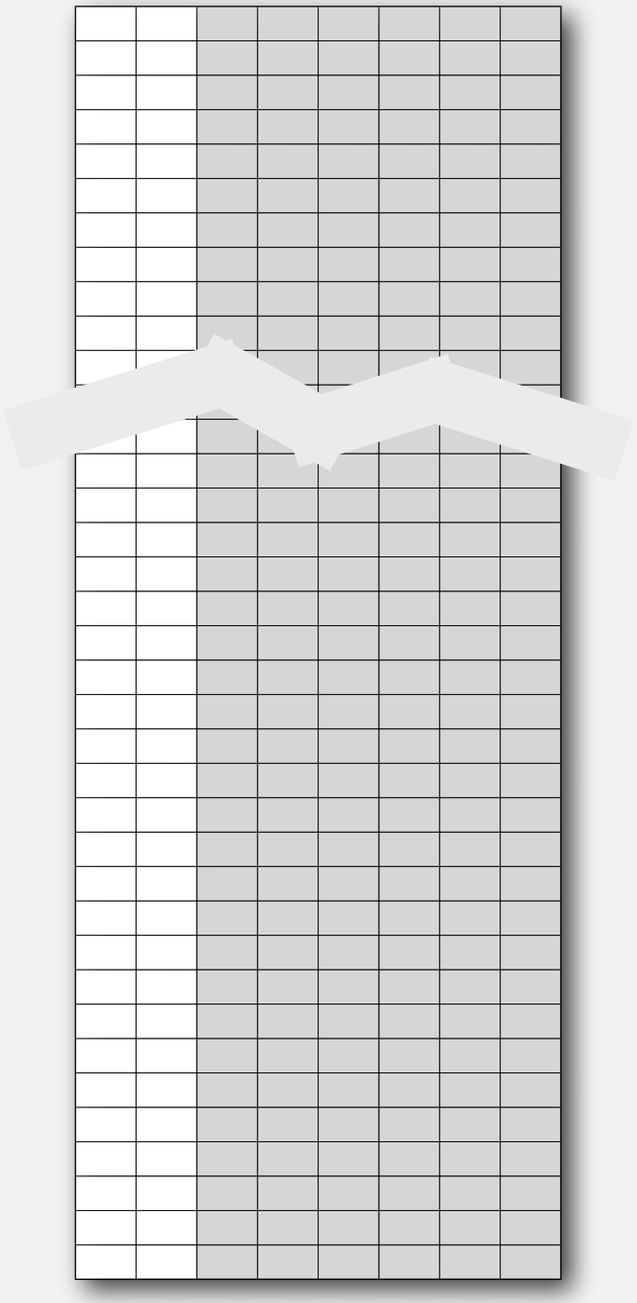
2^{16} = 65,536 counters.

Sort in 8 passes.

String sorting challenge 2b

Problem. Sort huge array of random 128-bit numbers.

Ex. Supercomputer sort, internet router.



Which sorting method to use?

- ✓ Insertion sort.
- Mergesort.
- Quicksort.
- Heapsort.
- ✓ LSD string sort.

Divide each word into eight 16-bit "chars"

2^{16} = 65,536 counters

LSD sort on leading 32 bits in 2 passes

Finish with insertion sort

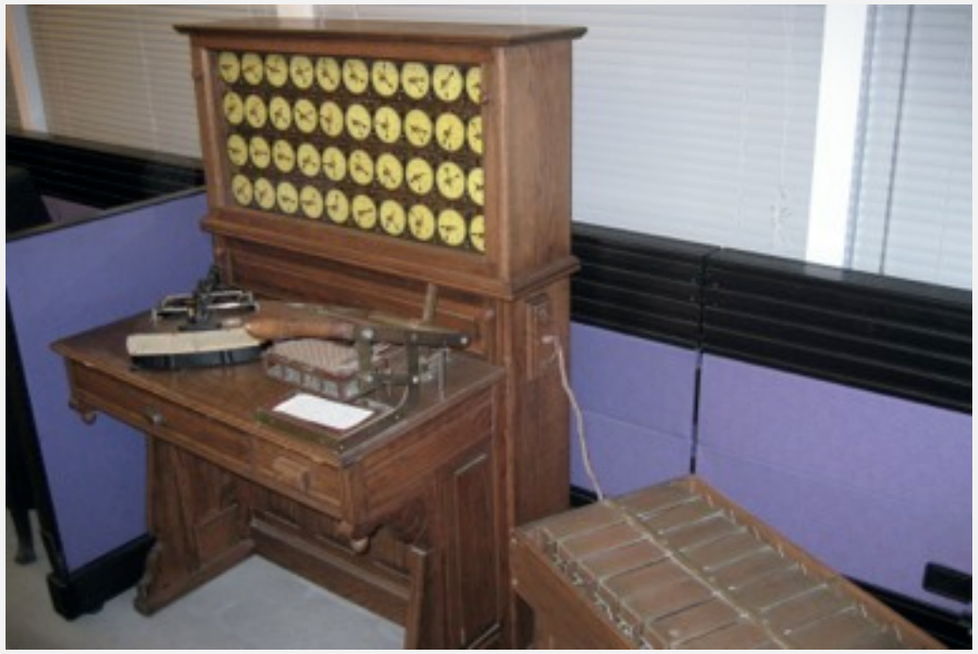
Examines only ~25% of the data

How to take a census in 1900s?

1880 Census. Took 1,500 people 7 years to manually process data.

Herman Hollerith. Developed counting and sorting machine to automate.

- Use punch cards to record data (e.g., gender, age).
- Machine sorts one column at a time (into one of 12 bins).
- Typical question: how many women of age 20 to 30?



Hollerith tabulating machine and sorter

punch card (12 holes per column)

1890 Census. Finished months early and under budget!

How to get rich sorting in 1900s?

Punch cards. [1900s to 1950s]

- Also useful for accounting, inventory, and business processes.
- Primary medium for data entry, storage, and processing.

Hollerith's company later merged with 3 others to form Computing Tabulating Recording Corporation (CTRC); the company was renamed in 1924.

IBM 80 Series Card Sorter (650 cards per minute)

LSD string sort: a moment in history (1960s)

card punch

punched cards

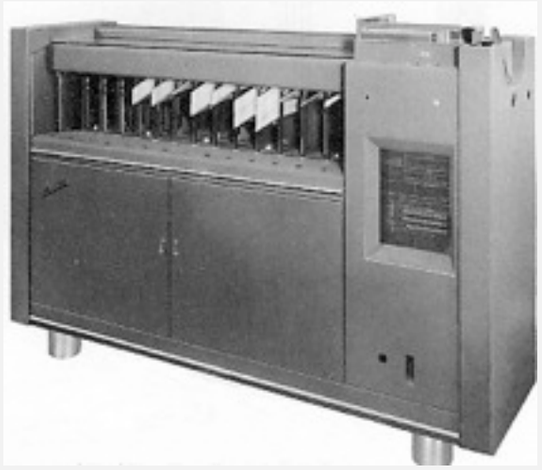
card reader

mainframe

line printer

To sort a card deck

- start on right column
- put cards into hopper
- machine distributes into bins
- pick up cards (stable)
- move left one column
- continue until sorted



card sorter

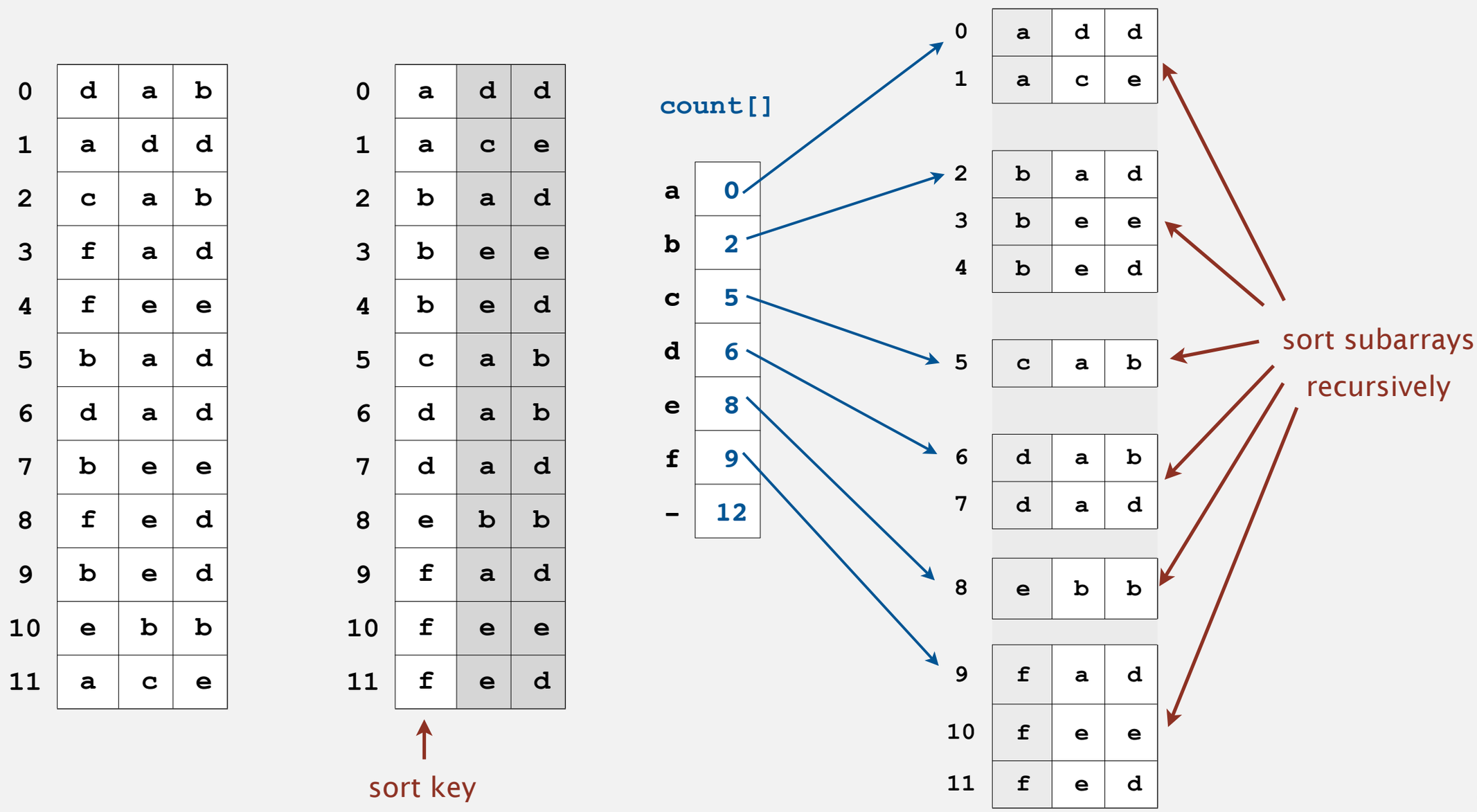
String Sorts

- Key-indexed counting
- LSD radix sort
- **MSD radix sort**
- 3-way radix quicksort
- Suffix arrays

Most-significant-digit-first string sort

MSD string (radix) sort.

- Partition array into R pieces according to first character (use key-indexed counting).
- Recursively sort all strings that start with each character (key-indexed counts delineate subarrays to sort).



MSD string sort: example

input								
she	are	are	are	are	are	are	are	are
sells	by	by	by	by	by	by	by	by
seashells	she	sells	seashells	sea	sea	sea	seas	sea
by	sells	seashells	sea	seashells	seashells	seashells	seashells	seashells
the	seashells	sea	seashells	seashells	seashells	seashells	seashells	seashells
sea	sea	sells	sells	sells	sells	sells	sells	sells
shore	shore	seashells	sells	sells	sells	sells	sells	sells
the	shells	she	she	she	she	she	she	she
shells	she	shore	shore	shore	shore	shore	shells	shells
she	sells	shells	shells	shells	shells	shells	shore	shore
sells	surely	she	she	she	she	she	she	she
are	seashells	surely	surely	surely	surely	surely	surely	surely
surely	the	the	the	the	the	the	the	the
seashells	the	the	the	the	the	the	the	the

lo

d

hi

need to examine every character in equal keys

end-of-string goes before any char value

							output
are	are	are	are	are	are	are	are
by	by	by	by	by	by	by	by
sea	sea	sea	sea	sea	sea	sea	sea
seashells	seashells	seashells	seashells	seashells	seashells	seashells	seashells
seashells	seashells	seashells	seashells	seashells	seashells	seashells	seashells
sells	sells	sells	sells	sells	sells	sells	sells
sells	sells	sells	sells	sells	sells	sells	sells
she	she	she	she	she	she	she	she
shells	shells	shells	shells	she	she	she	she
she	she	she	she	shells	shells	shells	shells
shore	shore	shore	shore	shore	shore	shore	shore
surely	surely	surely	surely	surely	surely	surely	surely
the	the	the	the	the	the	the	the
the	the	the	the	the	the	the	the

Trace of recursive calls for MSD string sort (no cutoff for small subarrays, subarrays of size 0 and 1 omitted)

Variable-length strings

Treat strings as if they had an extra char at end (smaller than any char).

why smaller?

0	s	e	a	-1						
1	s	e	a	s	h	e	l	l	s	-1
2	s	e	l	l	s	-1				
3	s	h	e	-1						
4	s	h	e	-1						
5	s	h	e	l	l	s	-1			
6	s	h	o	r	e	-1				
7	s	u	r	e	l	y	-1			

she before **shells**

```
private static int charAt(String s, int d)
{
   if (d < s.length()) return s.charAt(d);
   else return -1;
}
```

C strings. Have extra char `'\0'` at end $\Rightarrow$ no extra work needed.

MSD string sort: Java implementation

```
public static void sort(String[] a)
{
   aux = new String[a.length];
   sort(a, aux, 0, a.length, 0);
}

private static void sort(String[] a, String[] aux, int lo, int hi, int d)
{
   if (hi <= lo) return;
   int[] count = new int[R+2];
   for (int i = lo; i <= hi; i++)
      count[charAt(a[i], d) + 2]++;
   for (int r = 0; r < R+1; r++)
      count[r+1] += count[r];
   for (int i = lo; i <= hi; i++)
      aux[count[charAt(a[i], d) + 1]++] = a[i];
   for (int i = lo; i <= hi; i++)
      a[i] = aux[i - lo];

   for (int r = 0; r < R; r++)
      sort(a, aux, lo + count[r], lo + count[r+1] - 1, d+1);
}
```

can recycle `aux[]` array but not `count[]` array

key-indexed counting

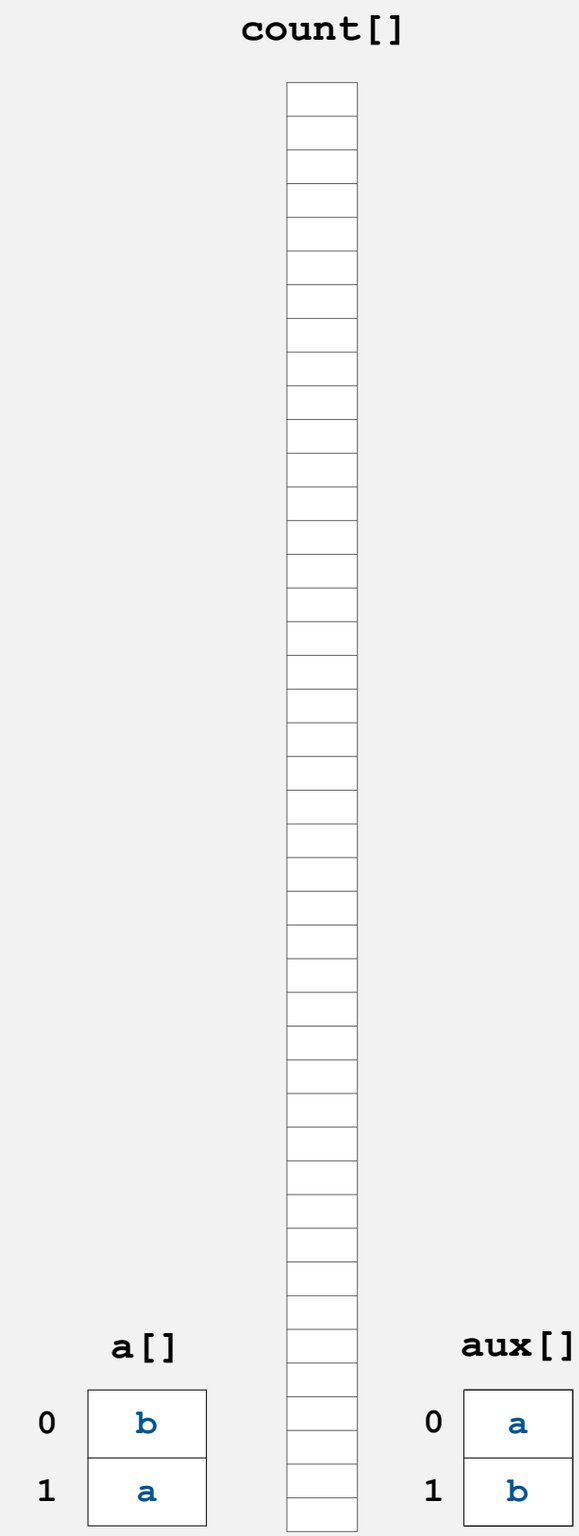
sort R subarrays recursively

MSD string sort: potential for disastrous performance

Observation 1. Much too slow for small subarrays.

- Each function call needs its own `count[]` array.
- ASCII (256 counts): 100x slower than copy pass for $N = 2$.
- Unicode (65,536 counts): 32,000x slower for $N = 2$.

Observation 2. Huge number of small subarrays because of recursion.



Cutoff to insertion sort

Solution. Cutoff to insertion sort for small subarrays.

- Insertion sort, but start at d^{th} character.
- Implement `less()` so that it compares starting at d^{th} character.

```
public static void sort(String[] a, int lo, int hi, int d)
{
   for (int i = lo; i <= hi; i++)
      for (int j = i; j > lo && less(a[j], a[j-1], d); j--)
         exch(a, j, j-1);
}
```

```
private static boolean less(String v, String w, int d)
{  return v.substring(d).compareTo(w.substring(d)) < 0;  }
```

in Java, forming and comparing substrings is faster than directly comparing chars with charAt()

MSD string sort: performance

Number of characters examined.

- MSD examines just enough characters to sort the keys.
- Number of characters examined depends on keys.
- Can be sublinear in input size!

compareTo() based sorts can also be sublinear!

Random (sublinear)	Non-random with duplicates (nearly linear)	Worst case (linear)
1EIO402	are	1DNB377
1HYL490	by	1DNB377
1ROZ572	sea	1DNB377
2HXE734	seashells	1DNB377
2IYE230	seashells	1DNB377
2XOR846	sells	1DNB377
3CDB573	sells	1DNB377
3CVP720	she	1DNB377
3IGJ319	she	1DNB377
3KNA382	shells	1DNB377
3TAV879	shore	1DNB377
4CQP781	surely	1DNB377
4QGI284	the	1DNB377
4YHV229	the	1DNB377

Characters examined by MSD string sort

Summary of the performance of sorting algorithms

Frequency of operations.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$N^2 / 2$	$N^2 / 4$	1	yes	`compareTo()`
mergesort	N lg N	N lg N	N	yes	`compareTo()`
quicksort	1.39 N lg N *	1.39 N lg N	c lg N	no	`compareTo()`
heapsort	2 N lg N	2 N lg N	1	no	`compareTo()`
LSD †	2 N W	2 N W	N + R	yes	`charAt()`
MSD ‡	2 N W	$N \log_R N$	N + D R	yes	`charAt()`

D = function-call stack depth
(length of longest prefix match)

* probabilistic
† fixed-length W keys
‡ average-length W keys

MSD string sort vs. quicksort for strings

Disadvantages of MSD string sort.

- Accesses memory "randomly" (cache inefficient).
- Inner loop has a lot of instructions.
- Extra space for `count[]`.
- Extra space for `aux[]`.

Disadvantage of quicksort.

- Linearithmic number of string compares (not linear).
- Has to rescan many characters in keys with long prefix matches.

Goal. Combine advantages of MSD and quicksort.

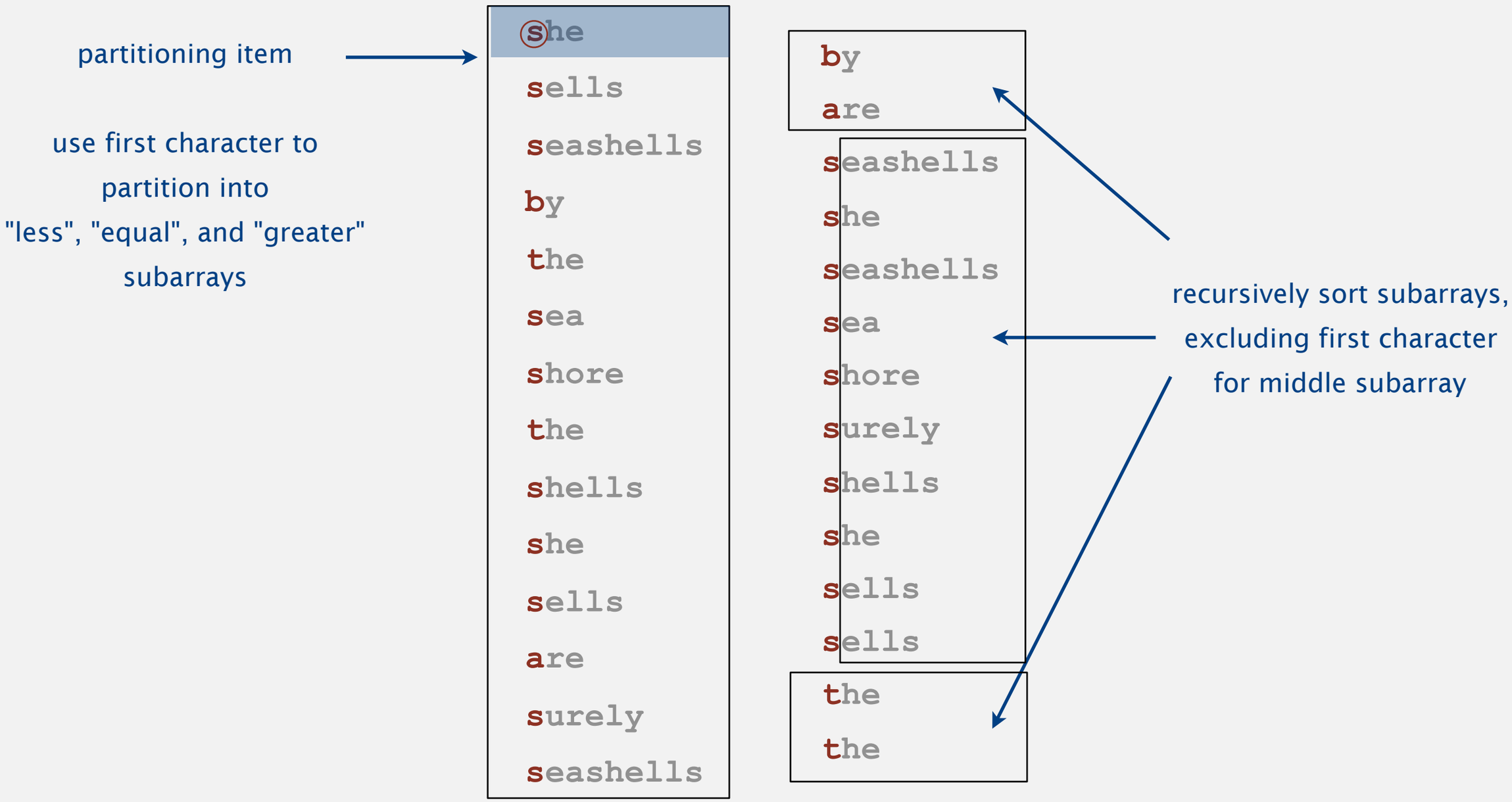
String Sorts

- Key-indexed counting
- LSD radix sort
- MSD radix sort
- **3-way radix quicksort**
- Suffix arrays

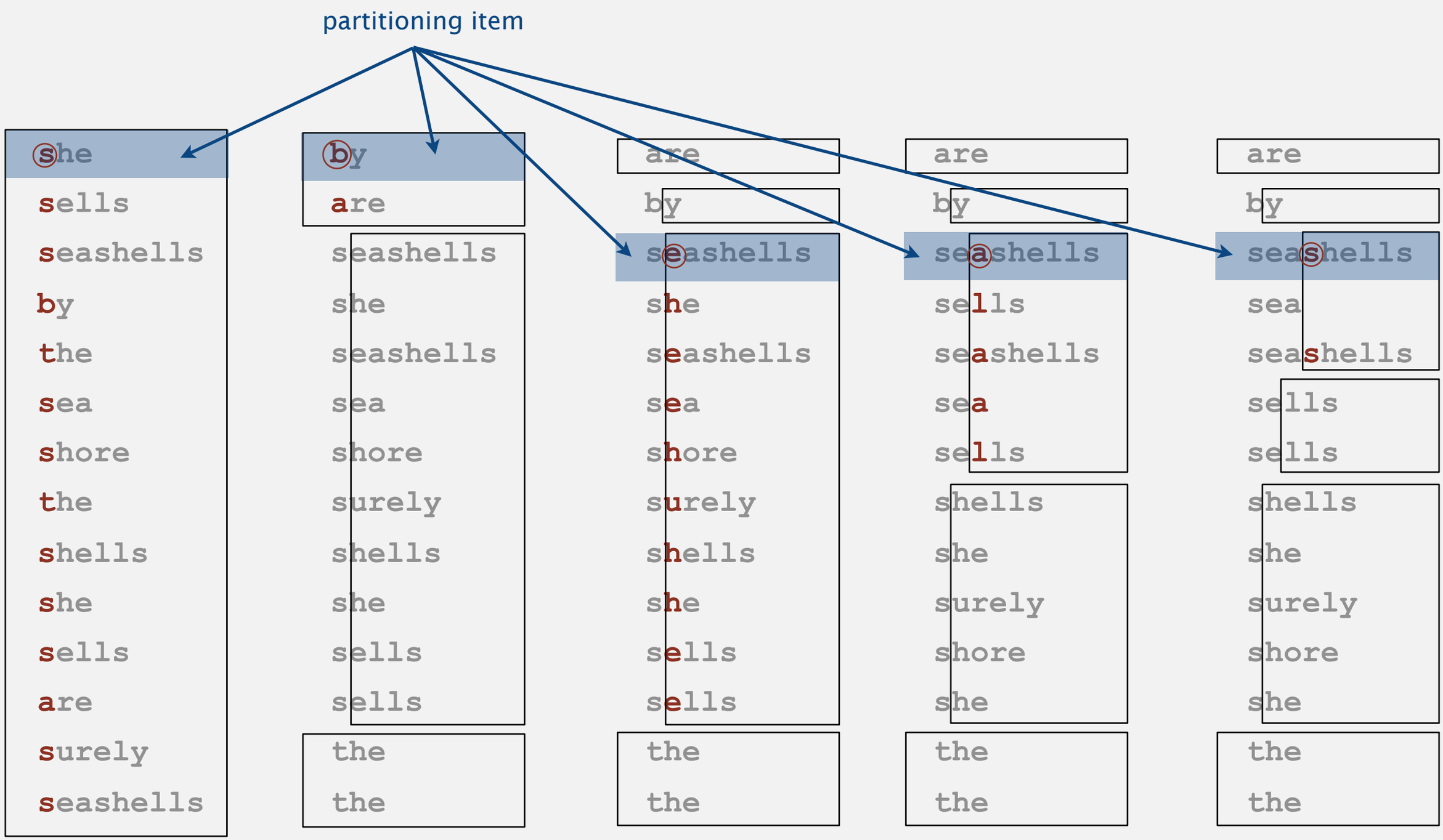
3-way string quicksort (Bentley and Sedgewick, 1997)

Overview. Do 3-way partitioning on the dth character.

- Less overhead than R-way partitioning in MSD string sort.
- Does not re-examine characters equal to the partitioning char (but does re-examine characters not equal to the partitioning char).



3-way string quicksort: trace of recursive calls



Trace of first few recursive calls for 3-way string quicksort (subarrays of size 1 not shown)

3-way string quicksort: Java implementation

```
private static void sort(String[] a)
{  sort(a, 0, a.length - 1, 0);  }

private static void sort(String[] a, int lo, int hi, int d)
{
   if (hi <= lo) return;
   int lt = lo, gt = hi;
   int v = charAt(a[lo], d);
   int i = lo + 1;
   while (i <= gt)
   {
      int t = charAt(a[i], d);
      if      (t < v) exch(a, lt++, i++);
      else if (t > v) exch(a, i, gt--);
      else            i++;
   }

   sort(a, lo, lt-1, d);
   if (v >= 0) sort(a, lt, gt, d+1);
   sort(a, gt+1, hi, d);
}
```

3-way partitioning (using d^{th} character)

to handle variable-length strings

sort 3 subarrays recursively

3-way string quicksort vs. standard quicksort

Standard quicksort.

- Uses $\sim 2 N \ln N$ string compares on average.
- Costly for keys with long common prefixes (and this is a common case!)

3-way string (radix) quicksort.

- Uses $\sim 2 N \ln N$ character compares on average for random strings.
- Avoids re-comparing long common prefixes.

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 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.

3-way string quicksort vs. MSD string sort

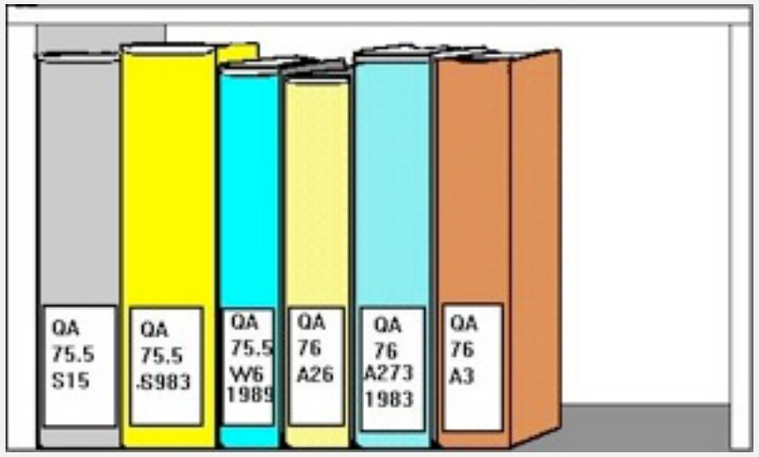
MSD string sort.

- Is cache-inefficient.
- Too much memory storing `count[]`.
- Too much overhead reinitializing `count[]` and `aux[]`.

3-way string quicksort.

- Has a short inner loop.
- Is cache-friendly.
- Is in-place.

library of Congress call numbers



Bottom line. 3-way string quicksort is the method of choice for sorting strings.

Summary of the performance of sorting algorithms

Frequency of operations.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$N^2 / 2$	$N^2 / 4$	1	yes	`compareTo()`
mergesort	N lg N	N lg N	N	yes	`compareTo()`
quicksort	1.39 N lg N *	1.39 N lg N	c lg N	no	`compareTo()`
heapsort	2 N lg N	2 N lg N	1	no	`compareTo()`
LSD †	2 N W	2 N W	N + R	yes	`charAt()`
MSD ‡	2 N W	$N \log_R N$	N + D R	yes	`charAt()`
3-way string quicksort	1.39 W N lg N *	1.39 N lg N	log N + W	no	`charAt()`

* probabilistic

† fixed-length W keys

‡ average-length W keys

String Sorts

- Key-indexed counting
- LSD radix sort
- MSD radix sort
- 3-way radix quicksort
- Suffix arrays

Keyword-in-context search

Given a text of N characters, preprocess it to enable fast substring search (find all occurrences of query string context).

```
% java KWIC tale.txt 15
search
o st giless to search for contraband
her unavailing search for your fathe
le and gone in search of her husband
t provinces in search of impoverishe
 dispersing in search of other carri
n that bed and search the straw hold

better thing
t is a far far better thing that i do than
 some sense of better things else forgotte
was capable of better things mr carton ent
```

characters of surrounding context

Applications. Linguistics, databases, web search, word processing, ….

Suffix sort

input string

a	a	c	a	a	g	t	t	t	a	c	a	a	g	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

form suffixes

0	a	a	c	a	a	g	t	t	t	a	c	a	a	g	c
1	a	c	a	a	g	t	t	t	a	c	a	a	g	c	
2	c	a	a	g	t	t	t	a	c	a	a	g	c		
3	a	a	g	t	t	t	a	c	a	a	g	c			
4	a	g	t	t	t	a	c	a	a	g	c				
5	g	t	t	t	a	c	a	a	g	c					
6	t	t	t	a	c	a	a	g	c						
7	t	t	a	c	a	a	g	c							
8	t	a	c	a	a	g	c								
9	a	c	a	a	g	c									
10	c	a	a	g	c										
11	a	a	g	c											
12	a	g	c												
13	g	c													
14	c														

sort suffixes to bring repeated substrings together

0	a	a	c	a	a	g	t	t	t	a	c	a	a	g	c
11	a	a	g	c											
3	a	a	g	t	t	t	a	c	a	a	g	c			
9	a	c	a	a	g	c									
1	a	c	a	a	g	t	t	t	a	c	a	a	g	c	
12	a	g	c												
4	a	g	t	t	t	a	c	a	a	g	c				
14	c														
10	c	a	a	g	c										
2	c	a	a	g	t	t	t	a	c	a	a	g	c		
13	g	c													
5	g	t	t	t	a	c	a	a	g	c					
8	t	a	c	a	a	g	c								
7	t	t	a	c	a	a	g	c							
6	t	t	t	a	c	a	a	g	c						

Keyword-in-context search: suffix-sorting solution

- Preprocess: suffix sort the text.
- Query: binary search for query; scan until mismatch.

KWIC search for "search" in Tale of Two Cities

								⋮														
632698	s	e	a	l	e	d	_	m	y	_	l	e	t	t	e	r	_	a	n	d	_	…
713727	s	e	a	m	s	t	r	e	s	s	_	i	s	_	l	i	f	t	e	d	_	…
660598	s	e	a	m	s	t	r	e	s	s	_	o	f	_	t	w	e	n	t	y	_	…
67610	s	e	a	m	s	t	r	e	s	s	_	w	h	o	_	w	a	s	_	w	i	…
4430	s	e	a	r	c	h	_	f	o	r	_	c	o	n	t	r	a	b	a	n	d	…
42705	s	e	a	r	c	h	_	f	o	r	_	y	o	u	r	_	f	a	t	h	e	…
499797	s	e	a	r	c	h	_	o	f	_	h	e	r	_	h	u	s	b	a	n	d	…
182045	s	e	a	r	c	h	_	o	f	_	i	m	p	o	v	e	r	i	s	h	e	…
143399	s	e	a	r	c	h	_	o	f	_	o	t	h	e	r	_	c	a	r	r	i	…
411801	s	e	a	r	c	h	_	t	h	e	_	s	t	r	a	w	_	h	o	l	d	…
158410	s	e	a	r	e	d	_	m	a	r	k	i	n	g	_	a	b	o	u	t	_	…
691536	s	e	a	s	_	a	n	d	_	m	a	d	a	m	e	_	d	e	f	a	r	…
536569	s	e	a	s	e	_	a	_	t	e	r	r	i	b	l	e	_	p	a	s	s	…
484763	s	e	a	s	e	_	t	h	a	t	_	h	a	d	_	b	r	o	u	g	h	…
								⋮														

Longest repeated substring

Given a string of N characters, find the longest repeated substring.

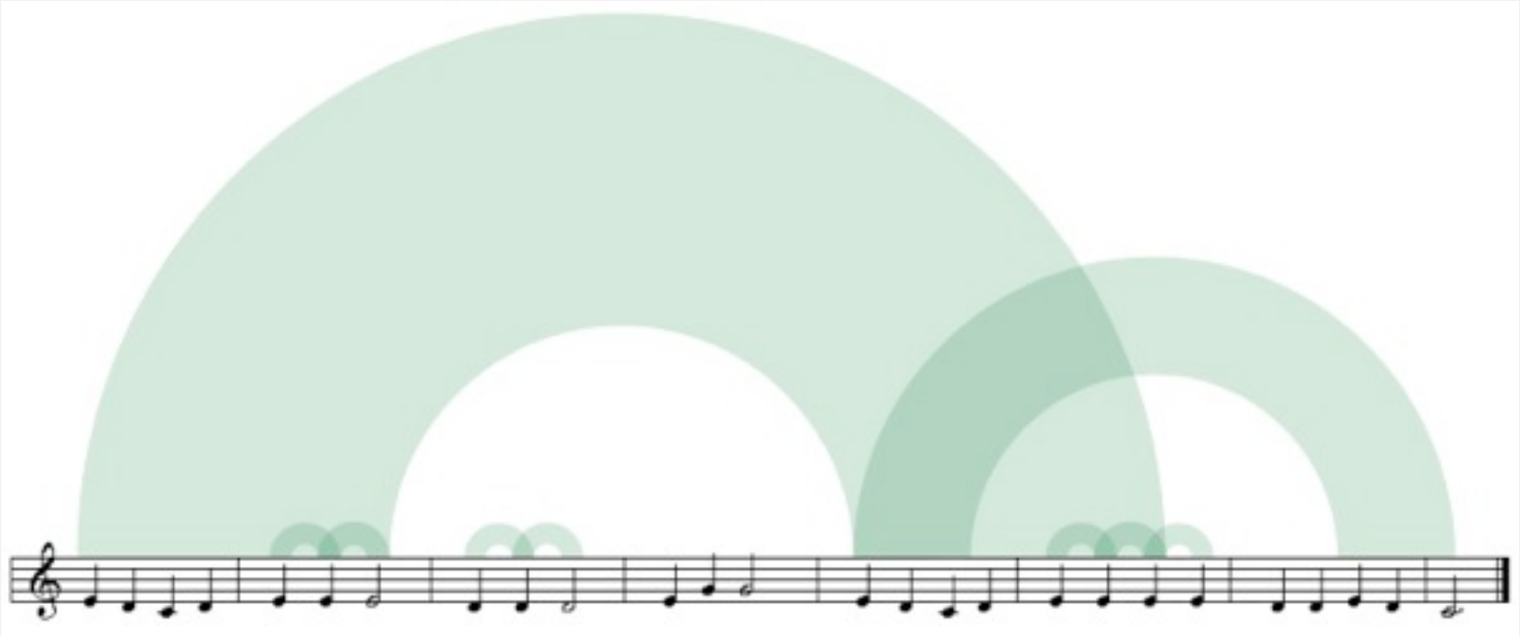
```
a a c a a g t t t a c a a g c a t g a t g c t g t a c t a
g g a g a g t t a t a c t g g t c g t c a a a c c t g a a
c c t a a t c c t t g t g t g t a c a c a c a c t a c t a
c t g t c g t c g t c a t a t a t c g a g a t c a t c g a
a c c g g a a g g c c g g a c a a g g c g g g g g g t a t
a g a t a g a t a g a c c c c t a g a t a c a c a t a c a
t a g a t c t a g c t a g c t a g c t c a t c g a t a c a
c a c t c t c a c a c t c a a g a g t t a t a c t g g t c
a a c a c a c t a c t a c g a c a g a c g a c c a a c c a
g a c a g a a a a a a a a a c t c t a t a t c t a t a a a a
```

Applications. Bioinformatics, cryptanalysis, data compression, ...

Longest repeated substring: a musical application

Visualize repetitions in music. `http://www.bewitched.com`

Mary Had a Little Lamb



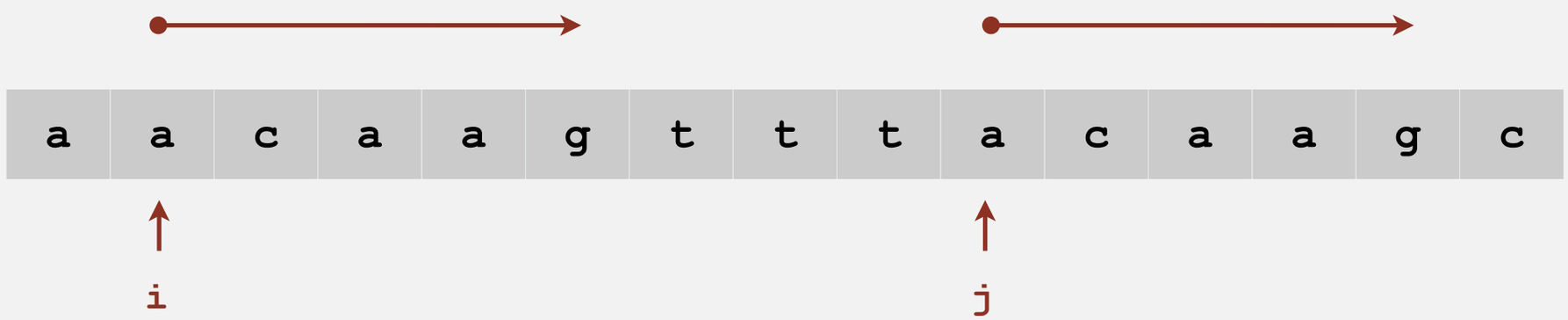
Bach's Goldberg Variations

Longest repeated substring

Given a string of N characters, find the longest repeated substring.

Brute-force algorithm.

- Try all indices i and j for start of possible match.
- Compute longest common prefix (LCP) for each pair.



Analysis. Running time $\leq D\,N^2$, where D is length of longest match.

Longest repeated substring: a sorting solution

input string

a	a	c	a	a	g	t	t	t	a	c	a	a	g	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

form suffixes

0	a	a	c	a	a	g	t	t	t	a	c	a	a	g	c
1	a	c	a	a	g	t	t	t	a	c	a	a	g	c	
2	c	a	a	g	t	t	t	a	c	a	a	g	c		
3	a	a	g	t	t	t	a	c	a	a	g	c			
4	a	g	t	t	t	a	c	a	a	g	c				
5	g	t	t	t	a	c	a	a	g	c					
6	t	t	t	a	c	a	a	g	c						
7	t	t	a	c	a	a	g	c							
8	t	a	c	a	a	g	c								
9	a	c	a	a	g	c									
10	c	a	a	g	c										
11	a	a	g	c											
12	a	g	c												
13	g	c													
14	c														

sort suffixes to bring repeated substrings together

0	a	a	c	a	a	g	t	t	t	a	c	a	a	g	c
11	a	a	g	c											
3	a	a	g	t	t	t	a	c	a	a	g	c			
9	a	c	a	a	g	c									
1	a	c	a	a	g	t	t	t	a	c	a	a	g	c	
12	a	g	c												
4	a	g	t	t	t	a	c	a	a	g	c				
14	c														
10	c	a	a	g	c										
2	c	a	a	g	t	t	t	a	c	a	a	g	c		
13	g	c													
5	g	t	t	t	a	c	a	a	g	c					
8	t	a	c	a	a	g	c								
7	t	t	a	c	a	a	g	c							
6	t	t	t	a	c	a	a	g	c						

compute longest prefix between adjacent suffixes

a	a	c	a	a	g	t	t	t	a	c	a	a	g	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Longest repeated substring: Java implementation

```
public String lrs(String s)
{
   int N = s.length();

   String[] suffixes = new String[N];
   for (int i = 0; i < N; i++)
      suffixes[i] = s.substring(i, N);

   Arrays.sort(suffixes);

   String lrs = "";
   for (int i = 0; i < N-1; i++)
   {
      int len = lcp(suffixes[i], suffixes[i+1]);
      if (len > lrs.length())
         lrs = suffixes[i].substring(0, len);
   }
   return lrs;
}
```

create suffixes (linear time and space)

sort suffixes

find LCP between adjacent suffixes in sorted order

```
% java LRS < mobydick.txt
,- Such a funny, sporty, gamy, jesty, joky, hoky-poky lad, is the Ocean, oh! Th
```

Sorting challenge

Problem. Five scientists A, B, C, D, and E are looking for long repeated substring in a genome with over 1 billion nucleotides.

- A has a grad student do it by hand.
- B uses brute force (check all pairs).
- C uses suffix sorting solution with insertion sort.
- D uses suffix sorting solution with LSD string sort.
- ✓ E uses suffix sorting solution with 3-way string quicksort.

but only if LRS is not long (!)

Q. Which one is more likely to lead to a cure cancer?

Longest repeated substring: empirical analysis

input file	characters	brute	suffix sort	length of LRS
`LRS.java`	2,162	0.6 sec	0.14 sec	73
`amendments.txt`	18,369	37 sec	0.25 sec	216
`aesop.txt`	191,945	1.2 hours	1.0 sec	58
`mobydick.txt`	1.2 million	43 hours †	7.6 sec	79
`chromosome11.txt`	7.1 million	2 months †	61 sec	12,567
`pi.txt`	10 million	4 months †	84 sec	14
`pipi.txt`	20 million	forever †	???	10 million

† estimated

Suffix sorting: worst-case input

Bad input: longest repeated substring very long.

- Ex: same letter repeated N times.
- Ex: two copies of the same Java codebase.

form suffixes

0	t w i n s t w i n s
1	w i n s t w i n s
2	i n s t w i n s
3	n s t w i n s
4	s t w i n s
5	t w i n s
6	w i n s
7	i n s
8	n s
9	s

sorted suffixes

9	i n s
8	i n s t w i n s
7	n s
6	n s t w i n s
5	s
4	s t w i n s
3	t w i n s
2	t w i n s t w i n s
1	w i n s
0	w i n s t w i n s

LRS needs at least 1 + 2 +3 + ... + D character compares, where D = length of longest match

Running time. Quadratic (or worse) in the length of the longest match.

Suffix sorting challenge

Problem. Suffix sort an arbitrary string of length N.

Q. What is worst-case running time of best algorithm for problem?

- Quadratic.
- ✓ Linearithmic. ← Manber's algorithm
- ✓ Linear. ← suffix trees (beyond our scope)
- Nobody knows.

Suffix sorting in linearithmic time

Manber's MSD algorithm overview.

- Phase 0: sort on first character using key-indexed counting sort.
- Phase i: given array of suffixes sorted on first 2^{i-1} characters, create array of suffixes sorted on first 2^i characters.

Worst-case running time. $N \lg N$.

- Finishes after $\lg N$ phases.
- Can perform a phase in linear time. (!) [ahead]

Linearithmic suffix sort example: phase 0

original suffixes

0	b a b a a a a b c b a b a a a a a 0
1	a b a a a a b c b a b a a a a a 0
2	b a a a a b c b a b a a a a a 0
3	a a a a b c b a b a a a a a 0
4	a a a b c b a b a a a a a 0
5	a a b c b a b a a a a a 0
6	a b c b a b a a a a a 0
7	b c b a b a a a a a 0
8	c b a b a a a a a 0
9	b a b a a a a a 0
10	a b a a a a a 0
11	b a a a a a 0
12	a a a a a 0
13	a a a a 0
14	a a a 0
15	a a 0
16	a 0
17	0

key-indexed counting sort (first character)

17	0	
1	a	b a a a a b c b a b a a a a a 0
16	a	0
3	a	a a a b c b a b a a a a a 0
4	a	a a b c b a b a a a a a 0
5	a	a b c b a b a a a a a 0
6	a	b c b a b a a a a a 0
15	a	a 0
14	a	a a 0
13	a	a a a 0
12	a	a a a a 0
10	a	b a a a a a 0
0	b	a b a a a a b c b a b a a a a a 0
9	b	a b a a a a a 0
11	b	a a a a a 0
7	b	c b a b a a a a a 0
2	b	a a a a b c b a b a a a a a 0
8	c	b a b a a a a a 0

sorted

Linearithmic suffix sort example: phase 1

original suffixes

0	b a b a a a a b c b a b a a a a a 0
1	a b a a a a b c b a b a a a a a 0
2	b a a a a b c b a b a a a a a 0
3	a a a a b c b a b a a a a a 0
4	a a a b c b a b a a a a a 0
5	a a b c b a b a a a a a 0
6	a b c b a b a a a a a 0
7	b c b a b a a a a a 0
8	c b a b a a a a a 0
9	b a b a a a a a 0
10	a b a a a a a 0
11	b a a a a a 0
12	a a a a a 0
13	a a a a 0
14	a a a 0
15	a a 0
16	a 0
17	0

index sort (first two characters)

17	0	
16	a 0	
12	a a	a a a 0
3	a a	a a b c b a b a a a a a 0
4	a a	a b c b a b a a a a a 0
5	a a	b c b a b a a a a a 0
13	a a	a a 0
15	a a	0
14	a a	a 0
6	a b	c b a b a a a a a 0
1	a b	a a a a b c b a b a a a a a 0
10	a b	a a a a a 0
0	b a	b a a a a b c b a b a a a a a 0
9	b a	b a a a a a 0
11	b a	a a a a 0
2	b a	a a a b c b a b a a a a a 0
7	b c	b a b a a a a a 0
8	c b	a b a a a a a 0

sorted

Linearithmic suffix sort example: phase 2

original suffixes

0	b a b a a a a b c b a b a a a a a 0
1	a b a a a a b c b a b a a a a a 0
2	b a a a a b c b a b a a a a a 0
3	a a a a b c b a b a a a a a 0
4	a a a b c b a b a a a a a 0
5	a a b c b a b a a a a a 0
6	a b c b a b a a a a a 0
7	b c b a b a a a a a 0
8	c b a b a a a a a 0
9	b a b a a a a a 0
10	a b a a a a a 0
11	b a a a a a 0
12	a a a a a 0
13	a a a a 0
14	a a a 0
15	a a 0
16	a 0
17	0

index sort (first four characters)

17	0
16	a 0
15	a a 0
14	a a a 0
3	a a a a b c b a b a a a a a 0
12	a a a a a 0
13	a a a a 0
4	a a a b c b a b a a a a a 0
5	a a b c b a b a a a a a 0
1	a b a a a a b c b a b a a a a a 0
10	a b a a a a a 0
6	a b c b a b a a a a a 0
2	b a a a a b c b a b a a a a a 0 a 0
11	b a a a a a 0
0	b a b a a a a b c b a b a a a a a 0
9	b a b a a a a a 0
7	b c b a b a a a a a 0
8	c b a b a a a a a 0

sorted

Linearithmic suffix sort example: phase 3

original suffixes

0	b a b a a a a b c b a b a a a a a 0
1	a b a a a a b c b a b a a a a a 0
2	b a a a a b c b a b a a a a a 0
3	a a a a b c b a b a a a a a 0
4	a a a b c b a b a a a a a 0
5	a a b c b a b a a a a a 0
6	a b c b a b a a a a a 0
7	b c b a b a a a a a 0
8	c b a b a a a a a 0
9	b a b a a a a a 0
10	a b a a a a a 0
11	b a a a a a 0
12	a a a a a 0
13	a a a a 0
14	a a a 0
15	a a 0
16	a 0
17	0

index sort (first eight characters)

17	0
16	a 0
15	a a 0
14	a a a 0
13	a a a a 0
12	a a a a a 0
3	a a a a b c b a b a a a a a 0
4	a a a b c b a b a a a a a 0
5	a a b c b a b a a a a a 0
10	a b a a a a a 0
1	a b a a a a b c b a b a a a a a 0
6	a b c b a b a a a a a 0
11	b a a a a a 0
2	b a a a a b c b a b a a a a a 0 a 0
9	b a b a a a a a 0
0	b a b a a a a b c b a b a a a a a 0
7	b c b a b a a a a a 0
8	c b a b a a a a a 0

finished (no equal keys)

Constant-time string compare by indexing into inverse

original suffixes

0	b a b a a a a b c b a b a a a a a 0
1	a b a a a a b c b a b a a a a a 0
2	b a a a a b c b a b a a a a a 0
3	a a a a b c b a b a a a a a 0
4	a a a b c b a b a a a a a 0
5	a a b c b a b a a a a a 0
6	a b c b a b a a a a a 0
7	b c b a b a a a a a 0
8	c b a b a a a a a 0
9	b a b a a a a a 0
10	a b a a a a a 0
11	b a a a a a 0
12	a a a a a 0
13	a a a a 0
14	a a a 0
15	a a 0
16	a 0
17	0

0 + 4 = 4

9 + 4 = 13

index sort (first four characters)

17	0
16	a 0
15	a a 0
14	a a a 0
3	a a a a b c b a b a a a a a 0
12	a a a a a 0
13	a a a a 0
4	a a a b c b a b a a a a a 0
5	a a b c b a b a a a a a 0
1	a b a a a a b c b a b a a a a a 0
10	a b a a a a a 0
6	a b c b a b a a a a a 0
2	b a a a a b c b a b a a a a a 0 a 0
11	b a a a a a 0
0	b a b a a a a b c b a b a a a a a 0
9	b a b a a a a a 0
7	b c b a b a a a a a 0
8	c b a b a a a a a 0

inverse

0	14
1	9
2	12
3	4
4	7
5	8
6	11
7	16
8	17
9	15
10	10
11	13
12	5
13	6
14	3
15	2
16	1
17	0

$\text{suffixes}_4[13] \le \text{suffixes}_4[4]$ (because $\text{inverse}[13] < \text{inverse}[4]$)

so $\text{suffixes}_8[9] \le \text{suffixes}_8[0]$

Suffix sort: experimental results

time to suffix sort (seconds)

algorithm	**mobydick.txt**	**aesopaesop.txt**
brute-force	36.000 †	4000 †
quicksort	9.5	167
LSD	not fixed length	not fixed length
MSD	395	out of memory
MSD with cutoff	6.8	162
3-way string quicksort	2.8	400
Manber MSD	17	8.5

† estimated

String sorting summary

We can develop linear-time sorts.

- Key compares not necessary for string keys.
- Use characters as index in an array.

We can develop sublinear-time sorts.

- Should measure amount of data in keys, not number of keys.
- Not all of the data has to be examined.

3-way string quicksort is asymptotically optimal.

- $1.39\, N \lg N$ chars for random data.

Long strings are rarely random in practice.

- Goal is often to learn the structure!
- May need specialized algorithms.

String Sorts, Tries

- String sorts

- Tries
- R-way tries
- Ternary search tries
- Character-based operations

Review: 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	`compareTo()`
hash table	1 †	1 †	1 †	no	`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

	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

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

- R-way tries
- Ternary search tries
- Character-based operations

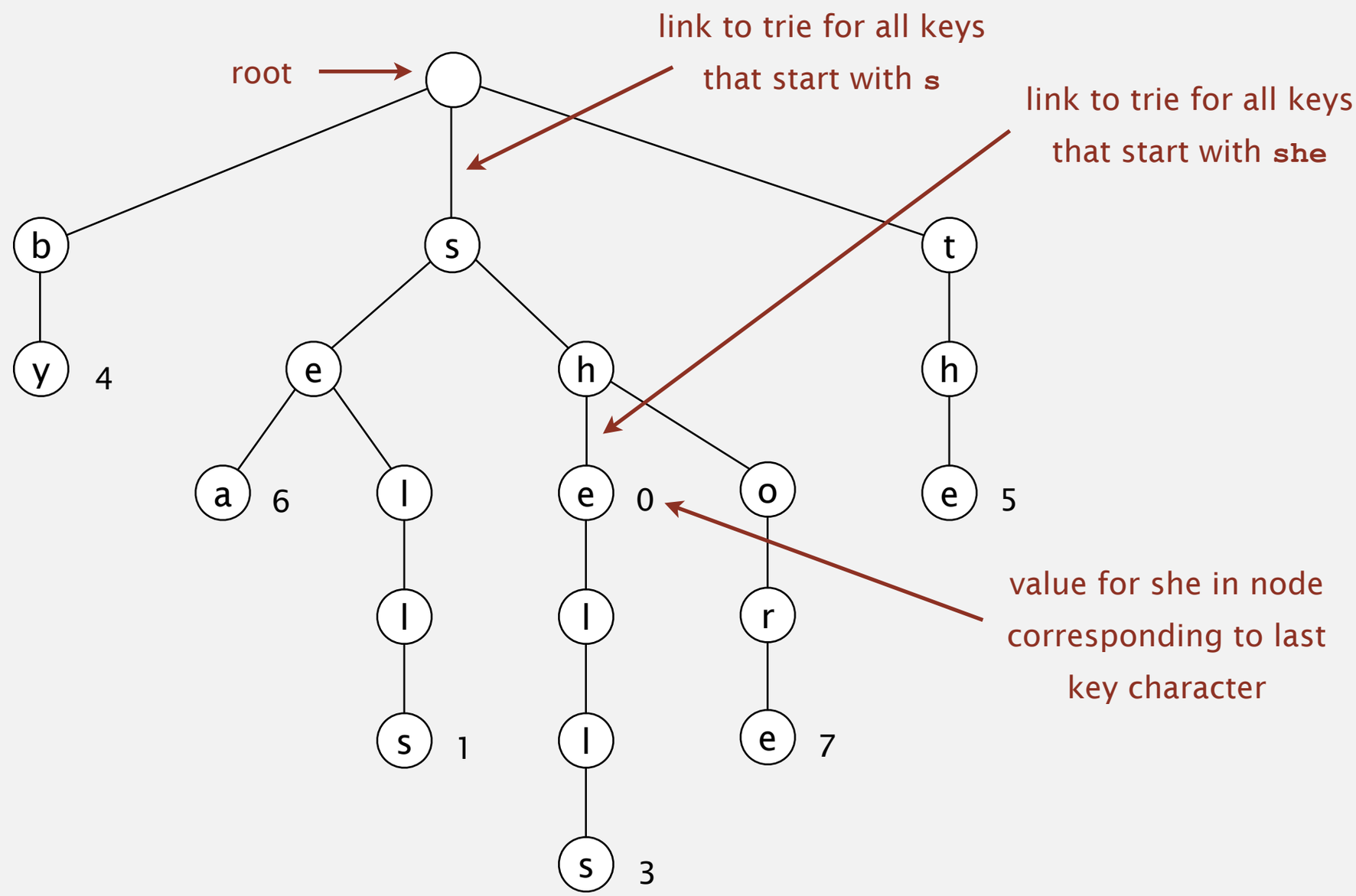
Tries

Tries. [from retrieval, but pronounced "try"]

- Store characters in nodes (not keys).
- Each node has R children, one for each possible character.
- Store values in nodes corresponding to last characters in keys.

for now, we do not draw null links

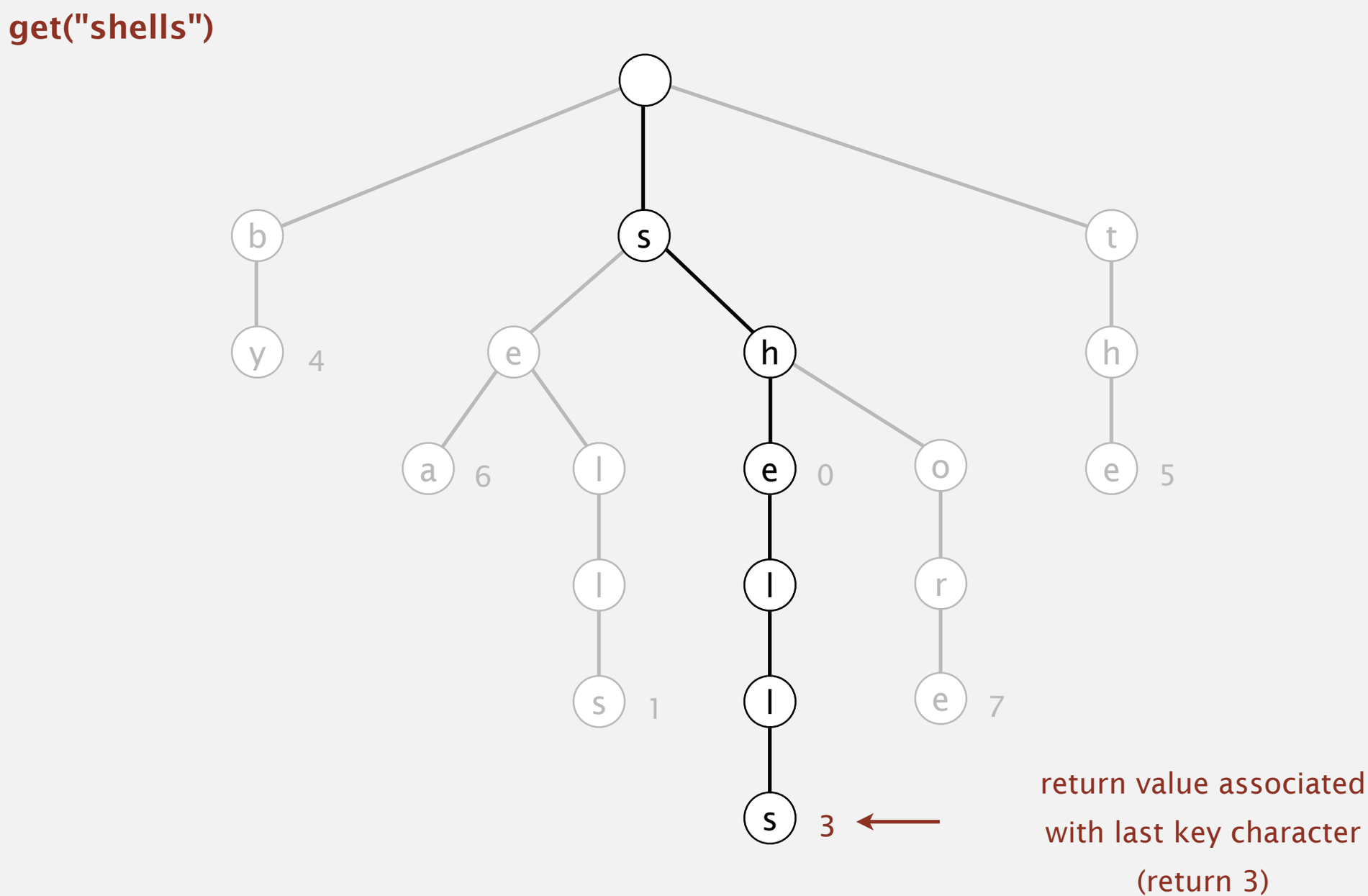
key	value
by	4
sea	6
sells	1
she	0
shells	3
shore	7
the	5



Search in a trie

Follow links corresponding to each character in the key.

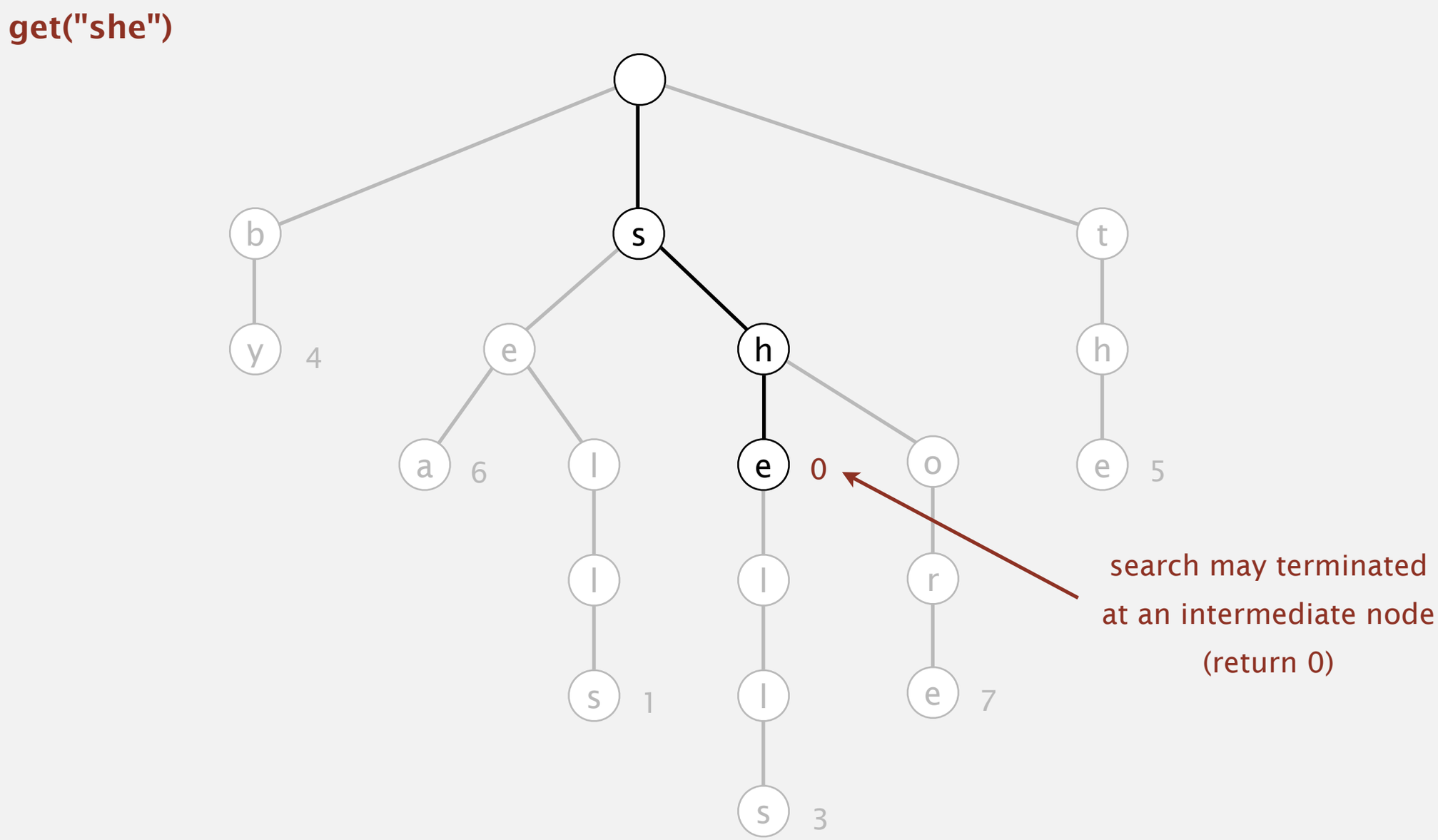
- Search hit: node where search ends has a non-null value.
- Search miss: reach a null link or node where search ends has null value.



Search in a trie

Follow links corresponding to each character in the key.

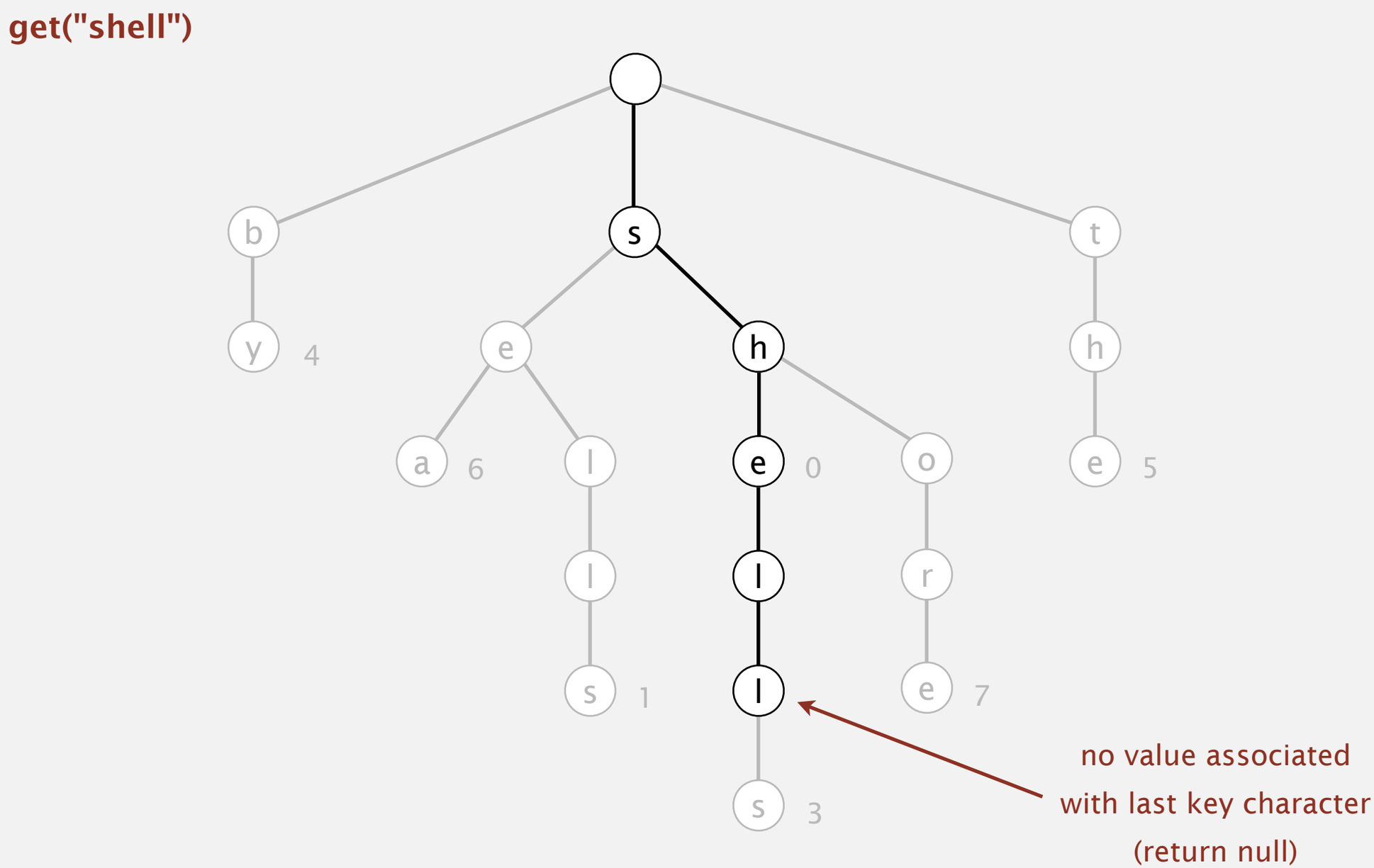
- Search hit: node where search ends has a non-null value.
- Search miss: reach a null link or node where search ends has null value.



Search in a trie

Follow links corresponding to each character in the key.

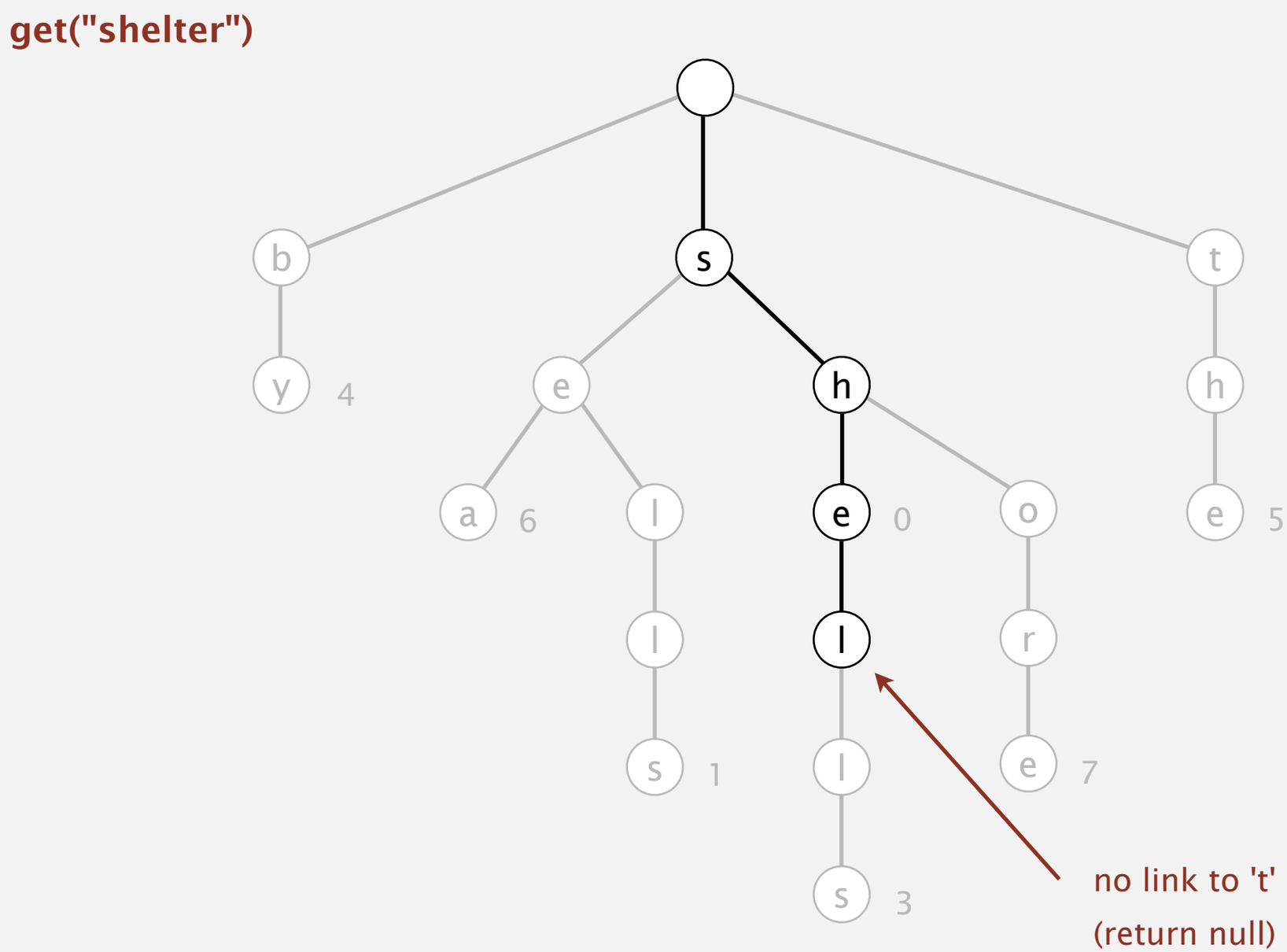
- Search hit: node where search ends has a non-null value.
- Search miss: reach a null link or node where search ends has null value.



Search in a trie

Follow links corresponding to each character in the key.

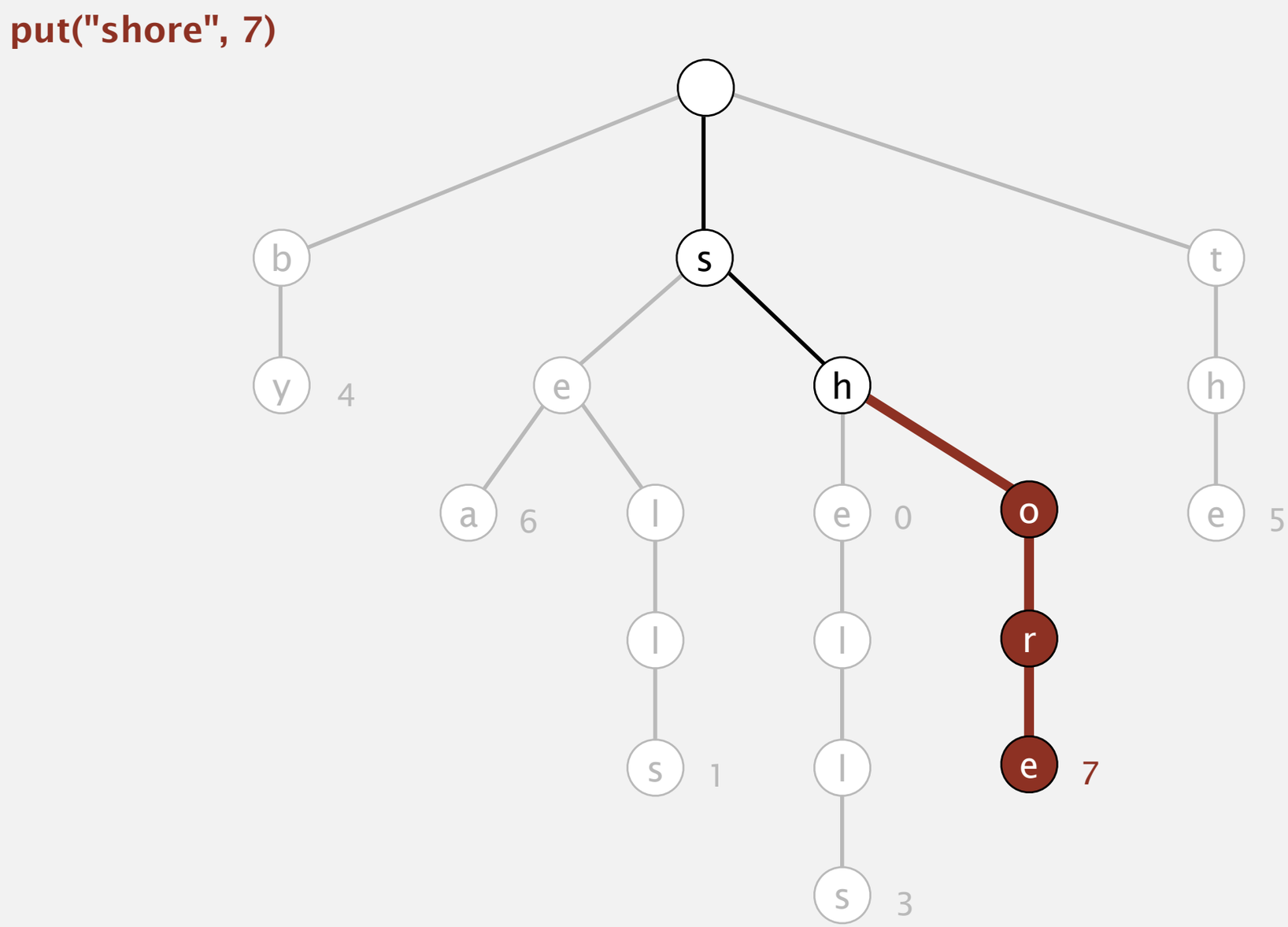
- Search hit: node where search ends has a non-null value.
- Search miss: reach a null link or node where search ends has null value.



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.

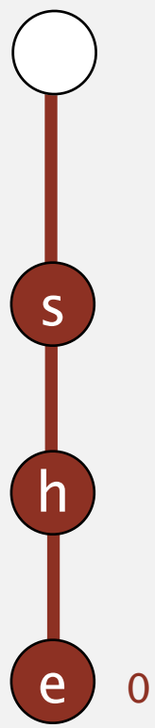


Trie construction demo

trie

Trie construction demo

put("she", 0)



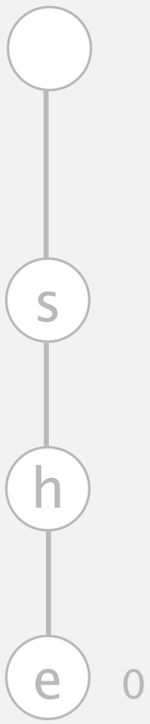
0

key is sequence of characters from root to value

value is in node corresponding to last character

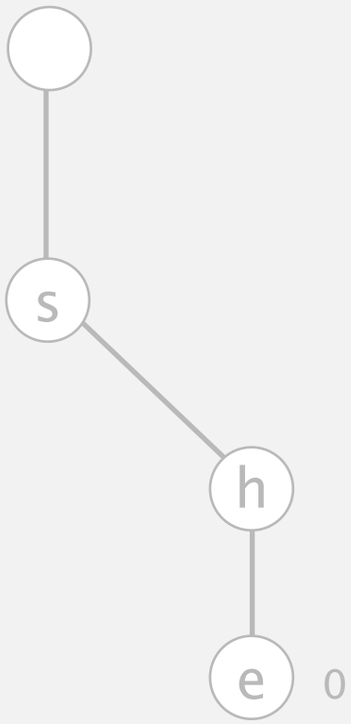
Trie construction demo

trie



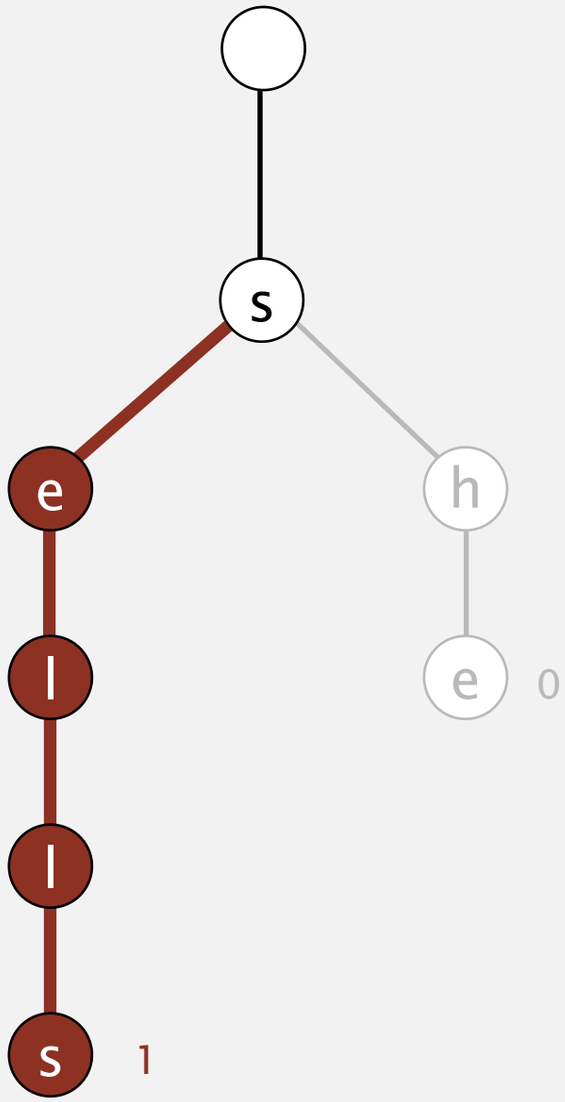
Trie construction demo

trie



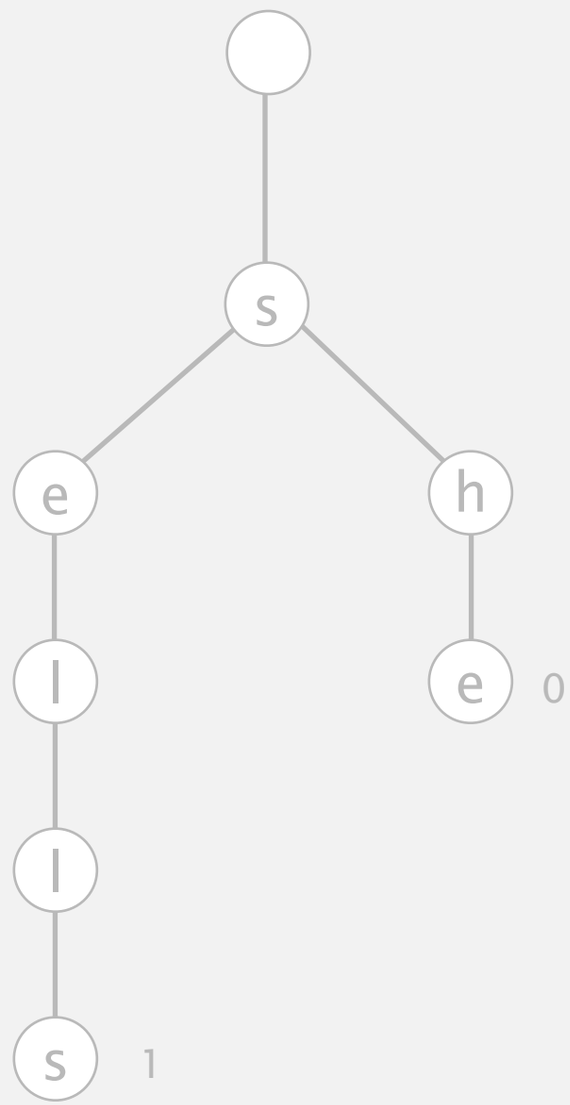
Trie construction demo

put("sells", 1)



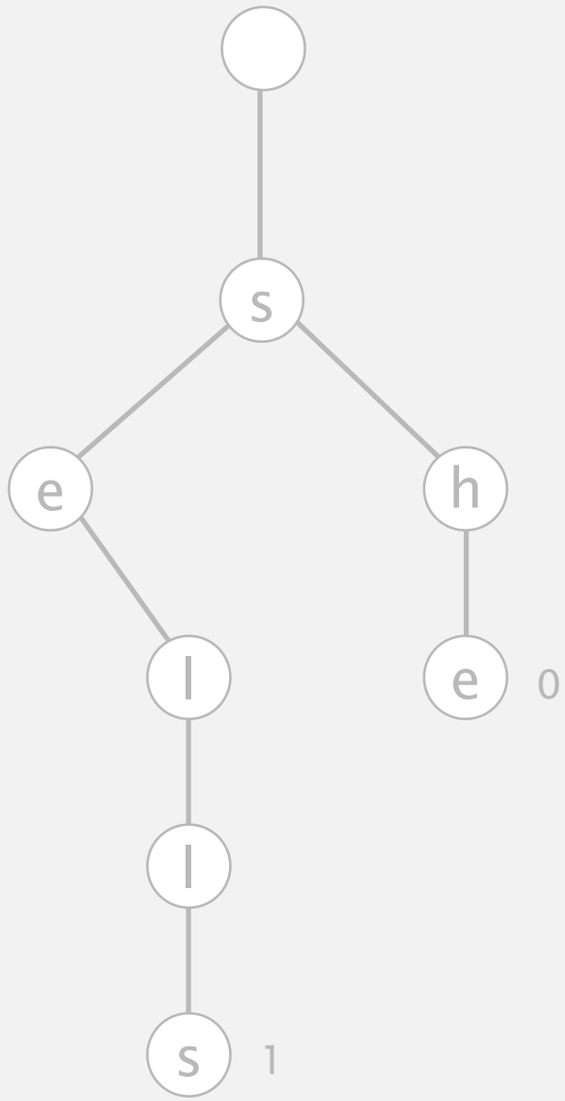
Trie construction demo

trie



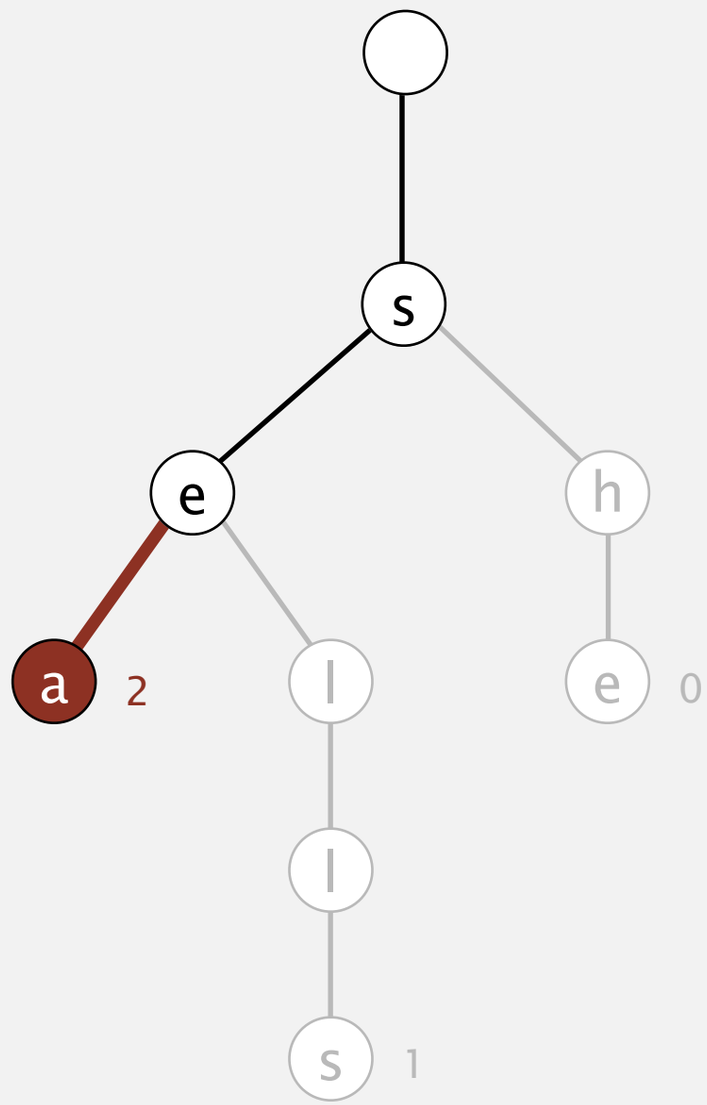
Trie construction demo

trie



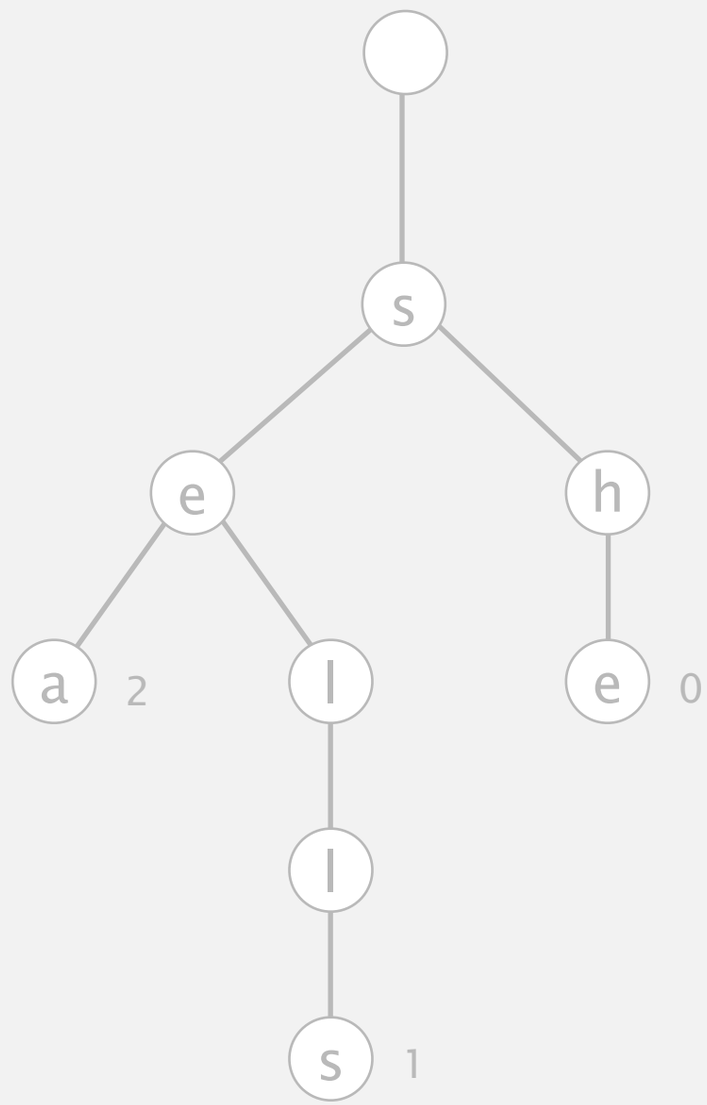
Trie construction demo

put("sea", 2)



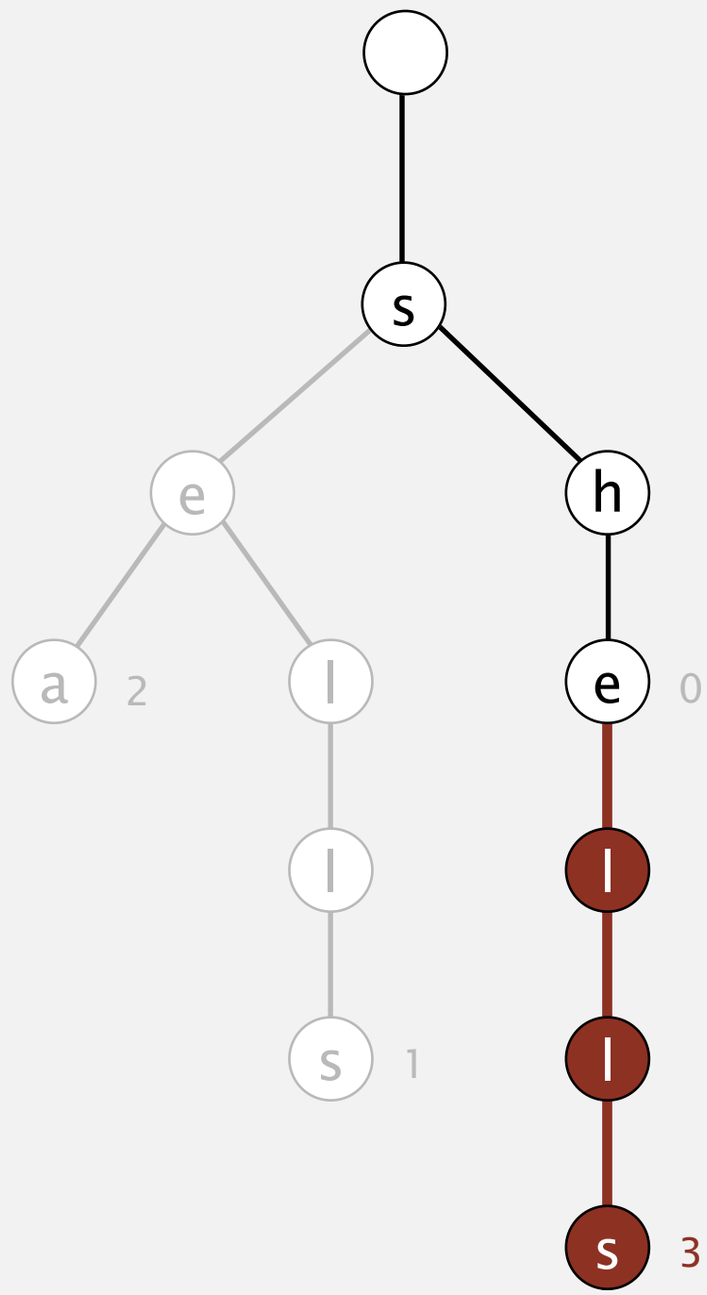
Trie construction demo

trie



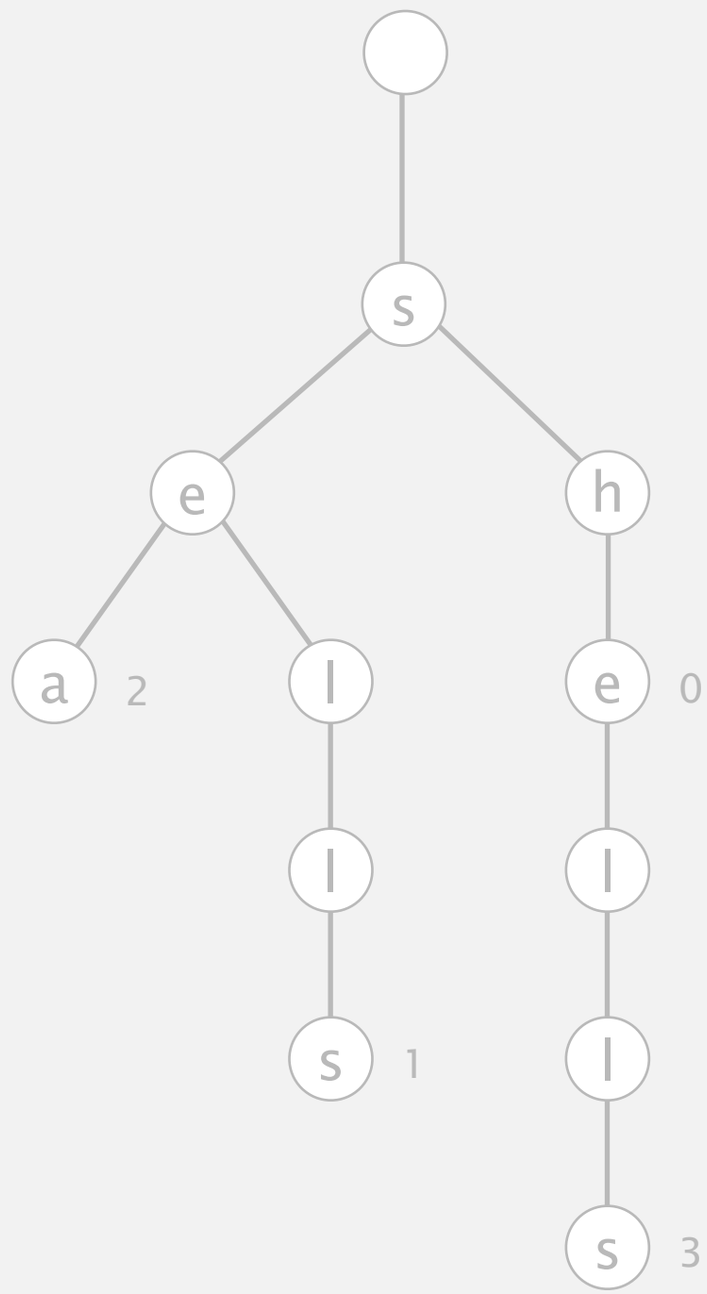
Trie construction demo

put("shells", 3)



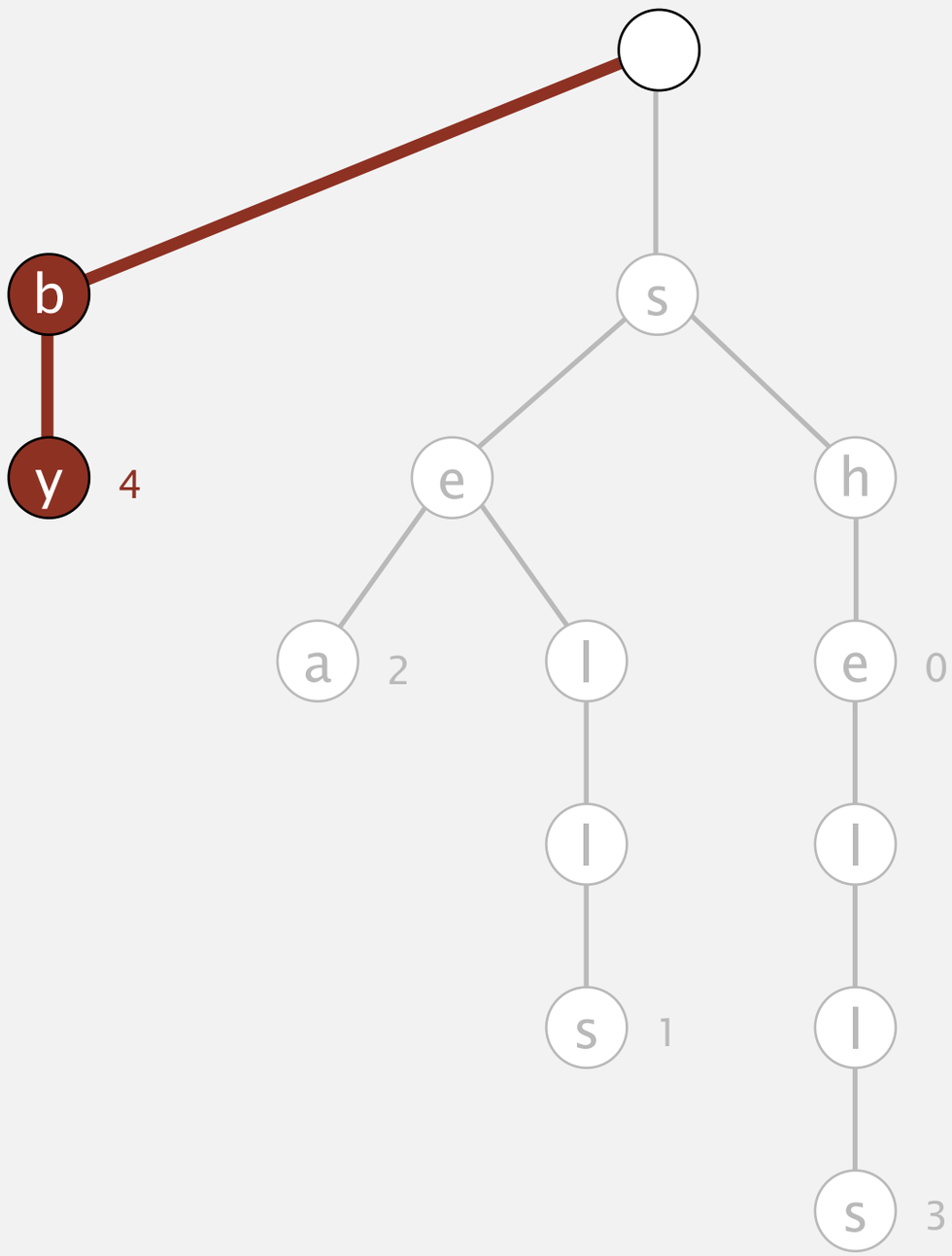
Trie construction demo

trie



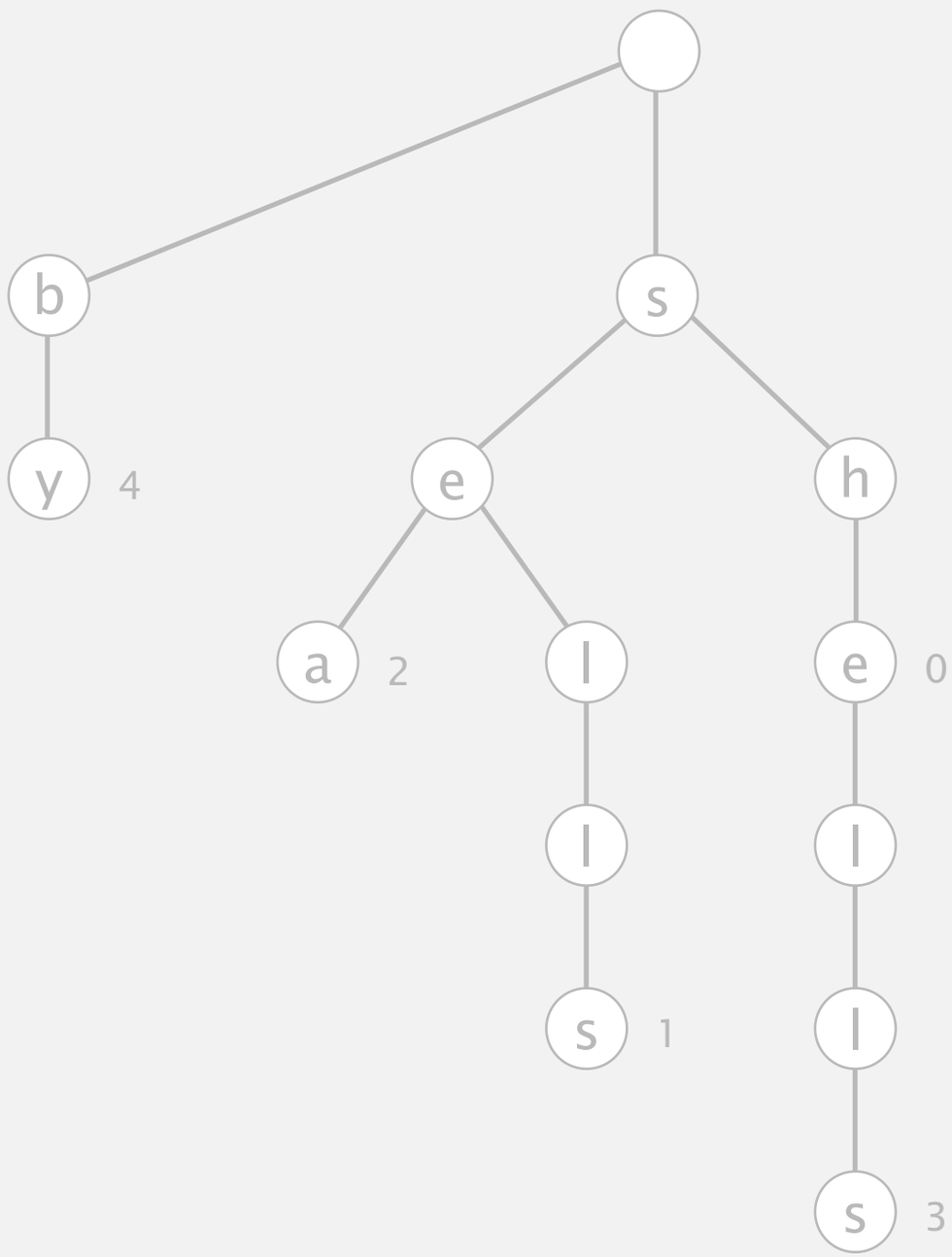
Trie construction demo

put("by", 4)



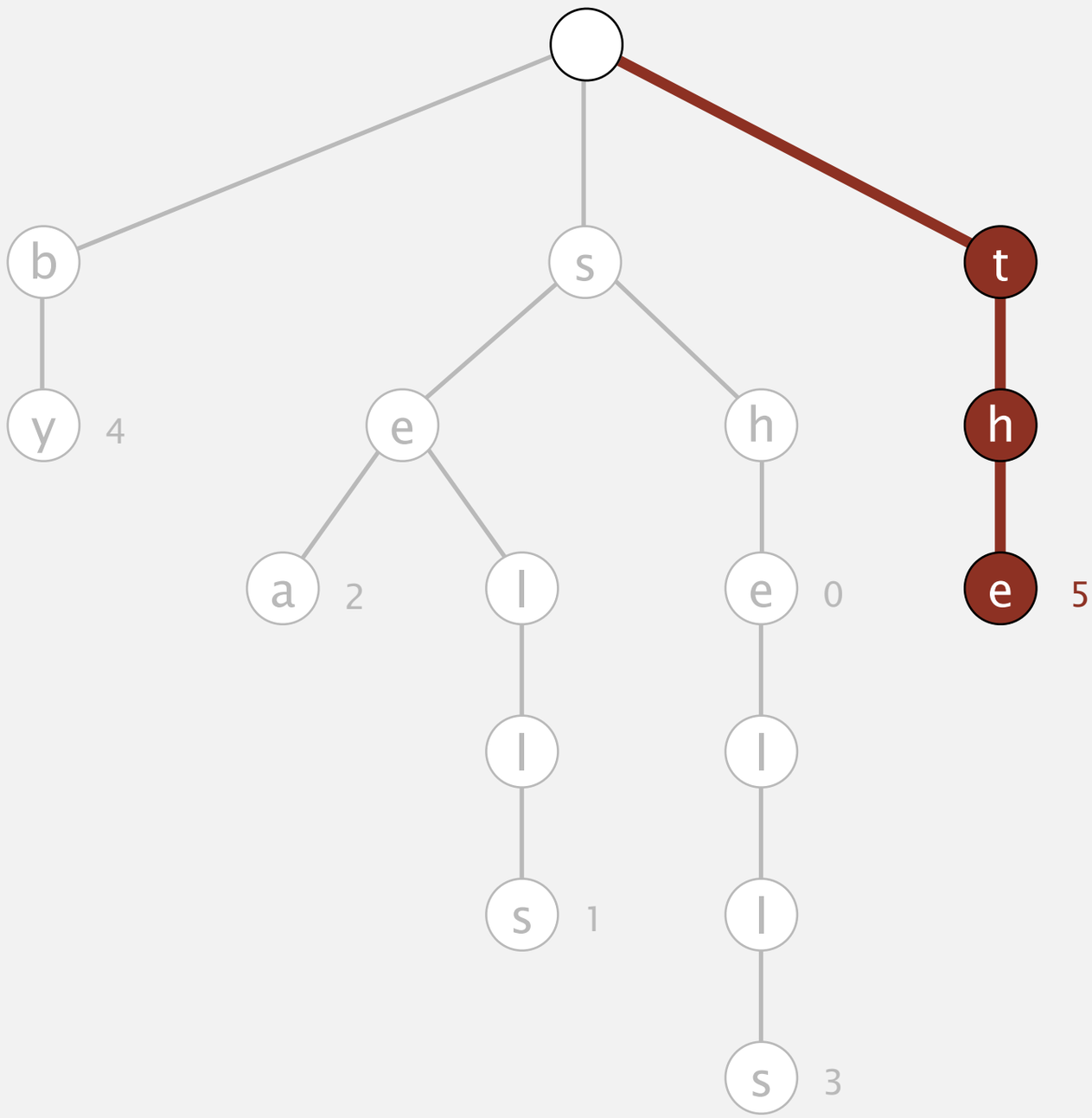
Trie construction demo

trie



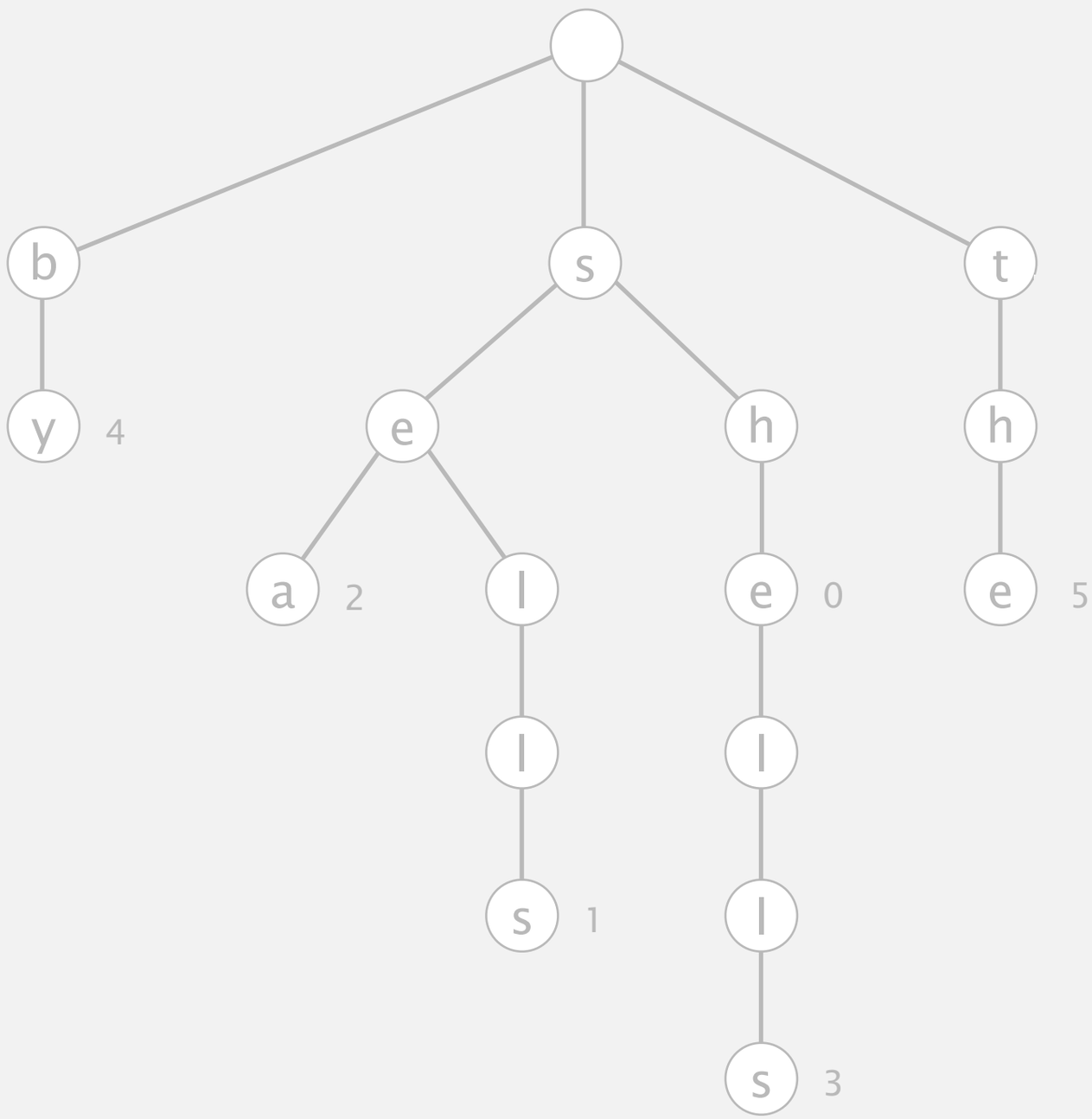
Trie construction demo

put("the", 5)



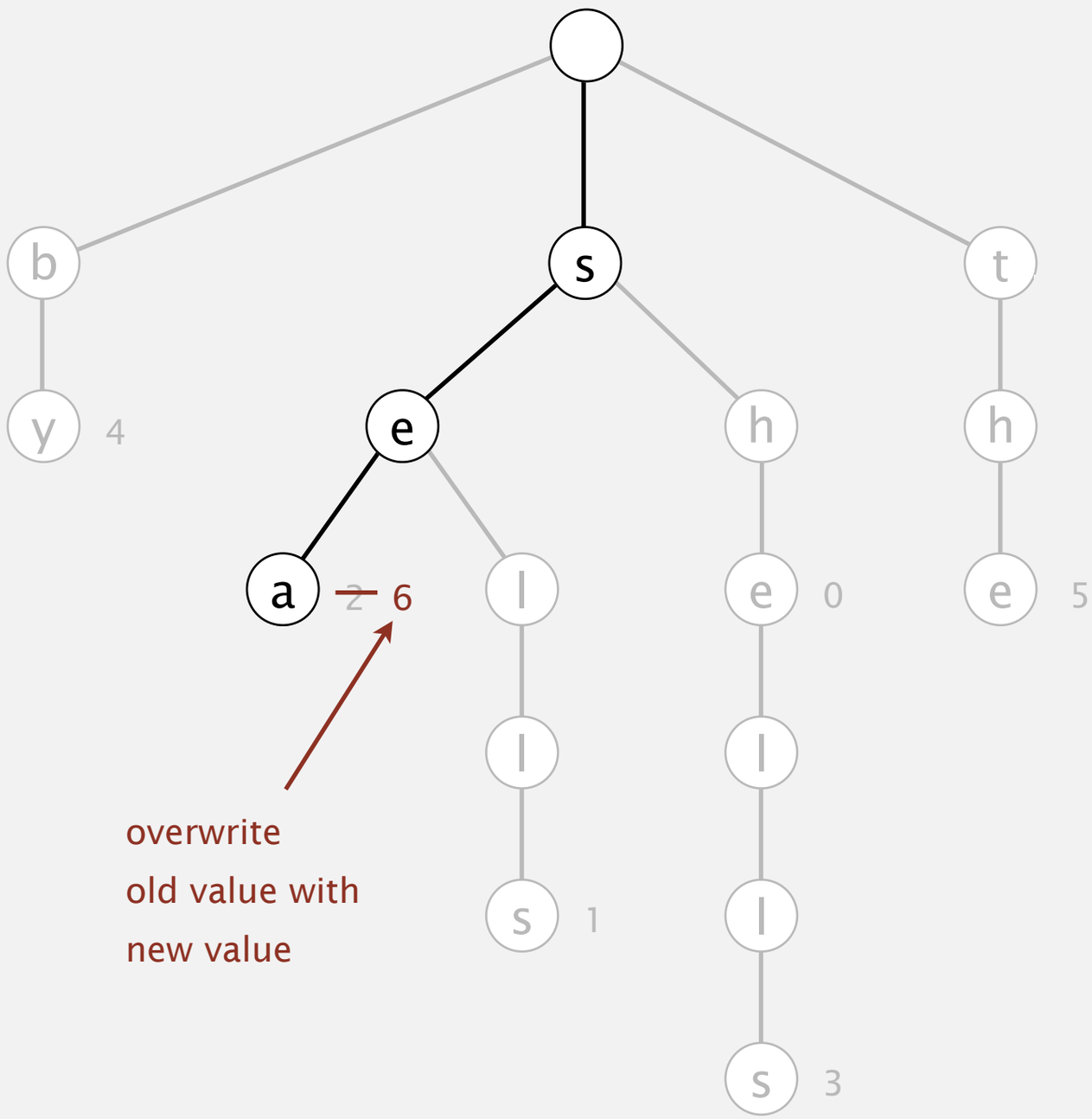
Trie construction demo

trie



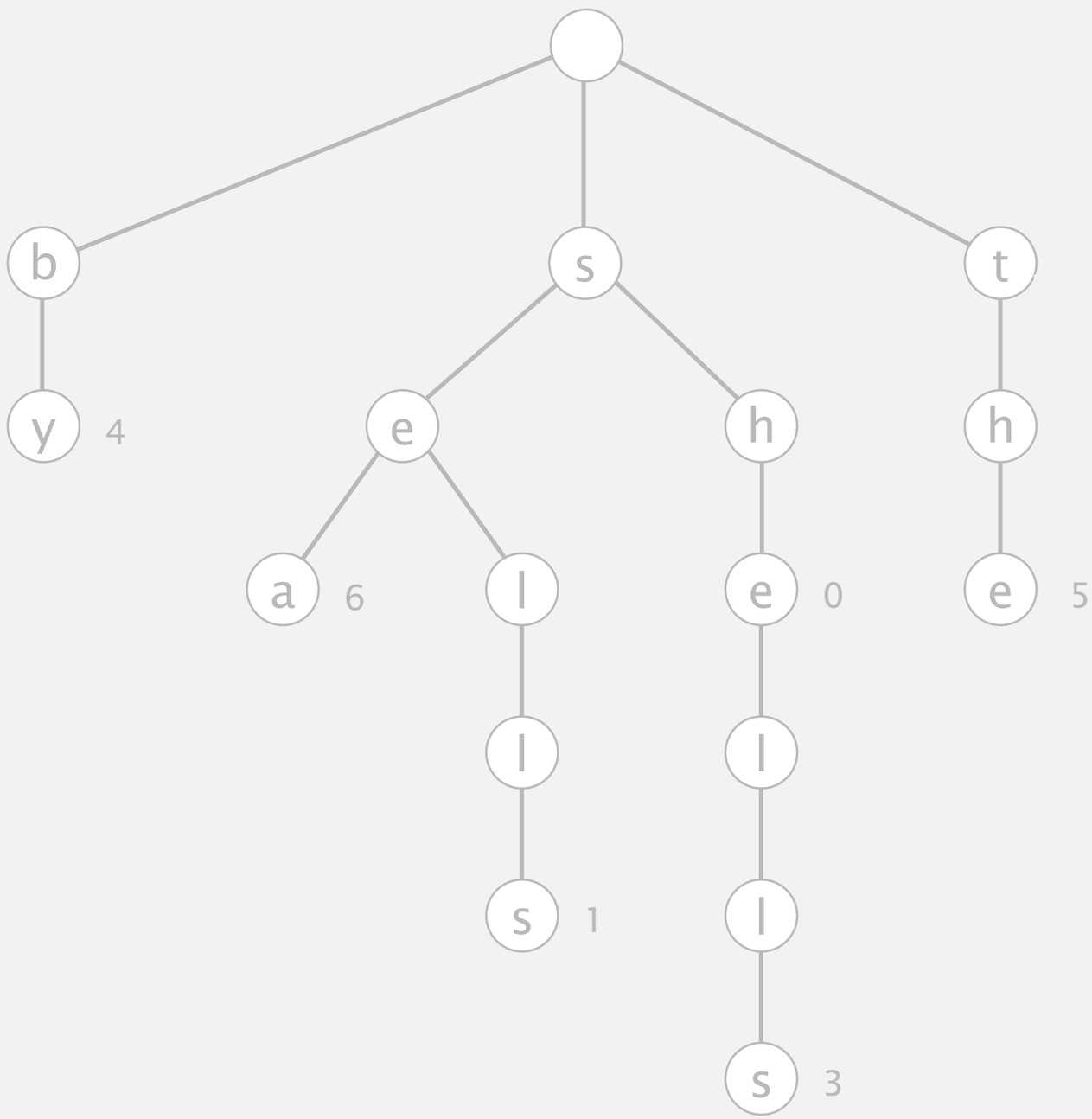
Trie construction demo

put("sea", 6)



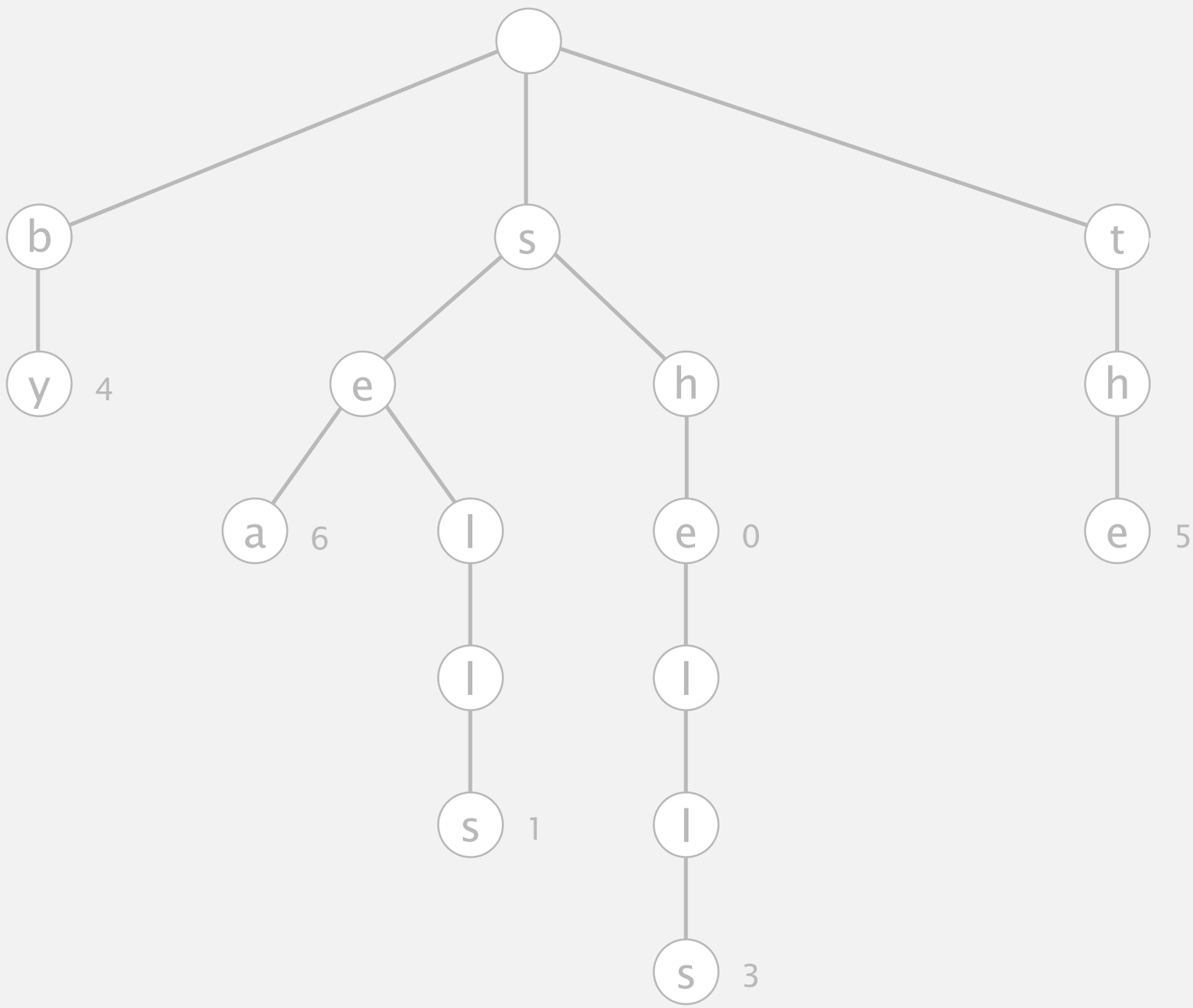
Trie construction demo

trie



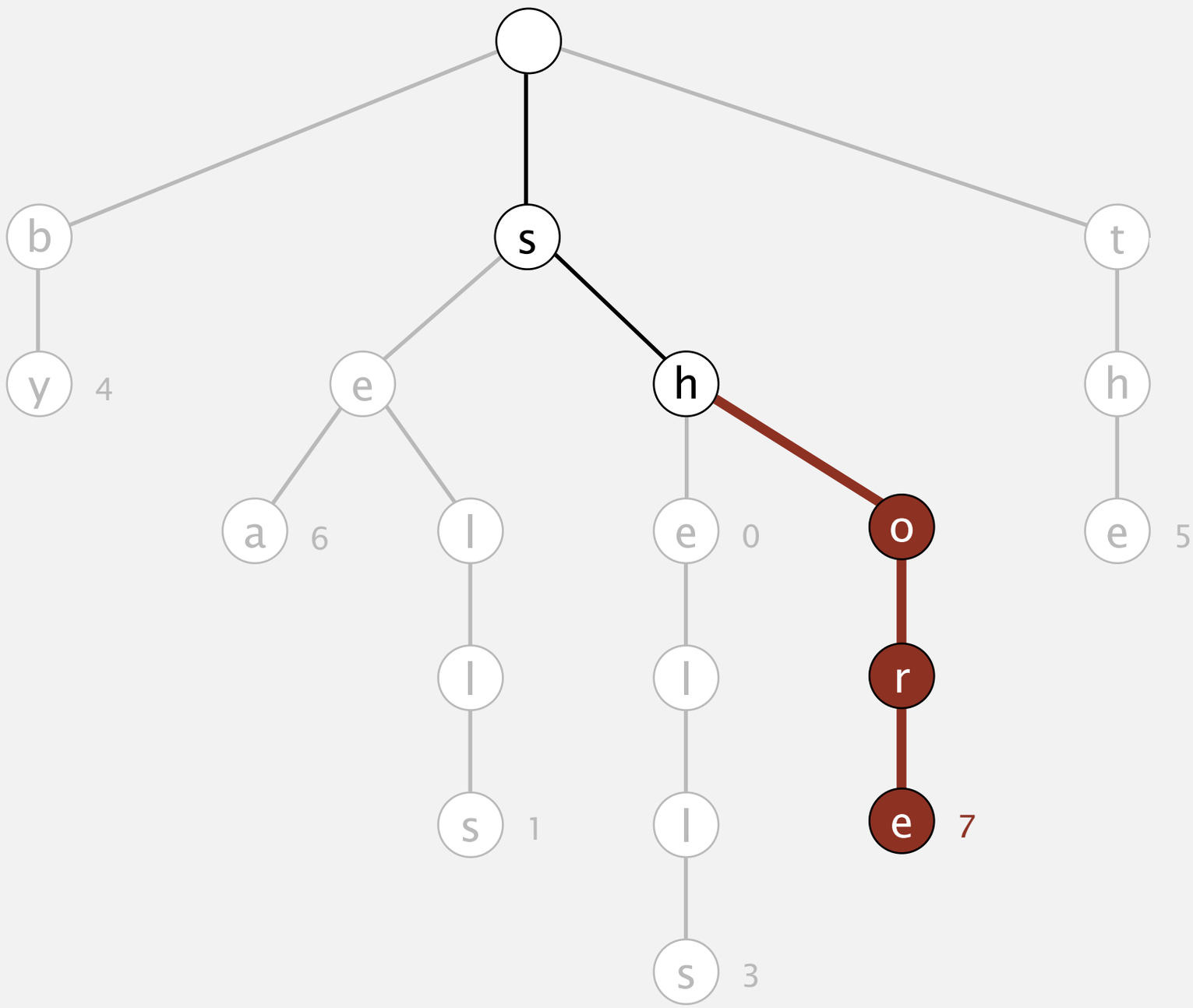
Trie construction demo

trie



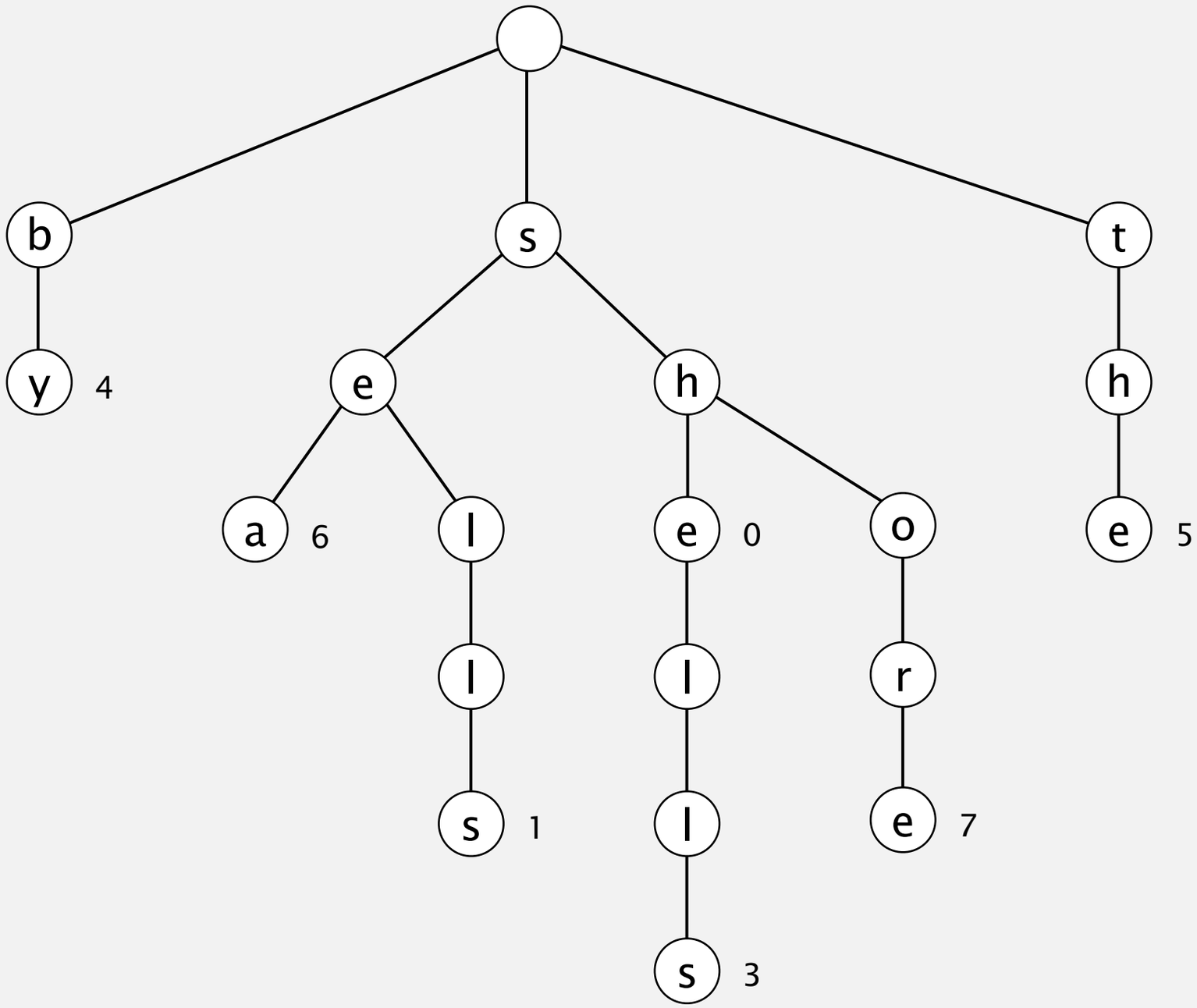
Trie construction demo

put("shore", 7)



Trie construction demo

trie

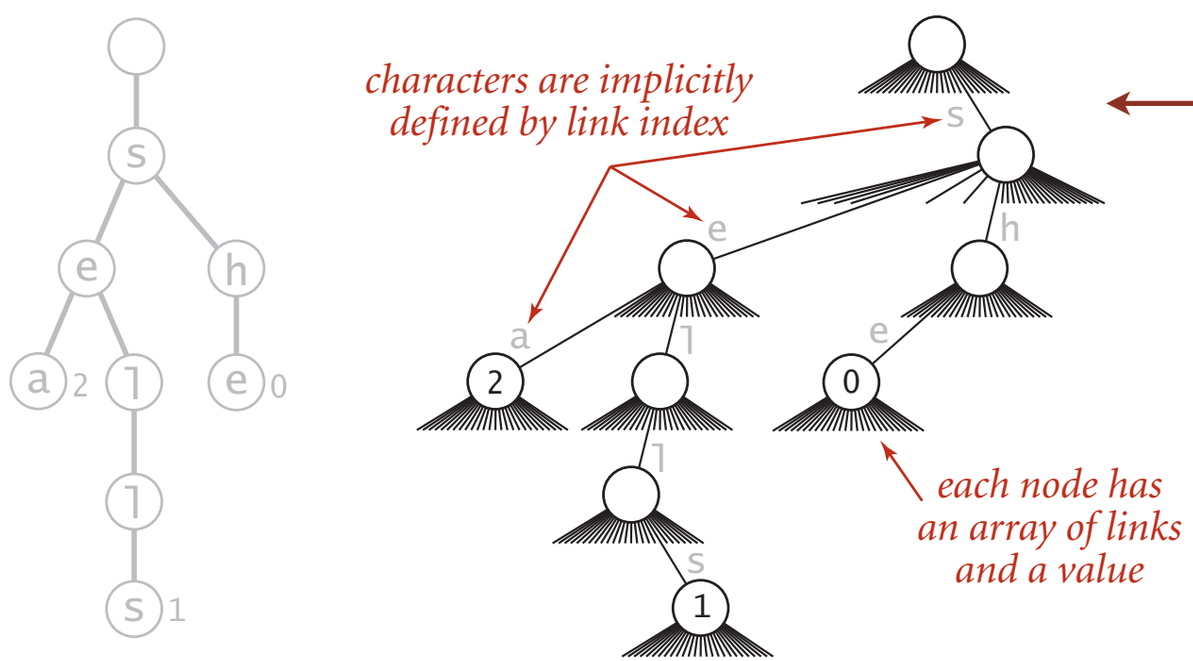


Trie representation: Java implementation

Node. A value, plus references to R nodes.

```
private static class Node
{
   private Object value;
   private Node[] next = new Node[R];
}
```

use `Object` instead of `Value` since no generic array creation in Java



Trie representation

neither keys nor characters are explicitly stored

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;
}

   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);
   }


}
```

cast needed

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).

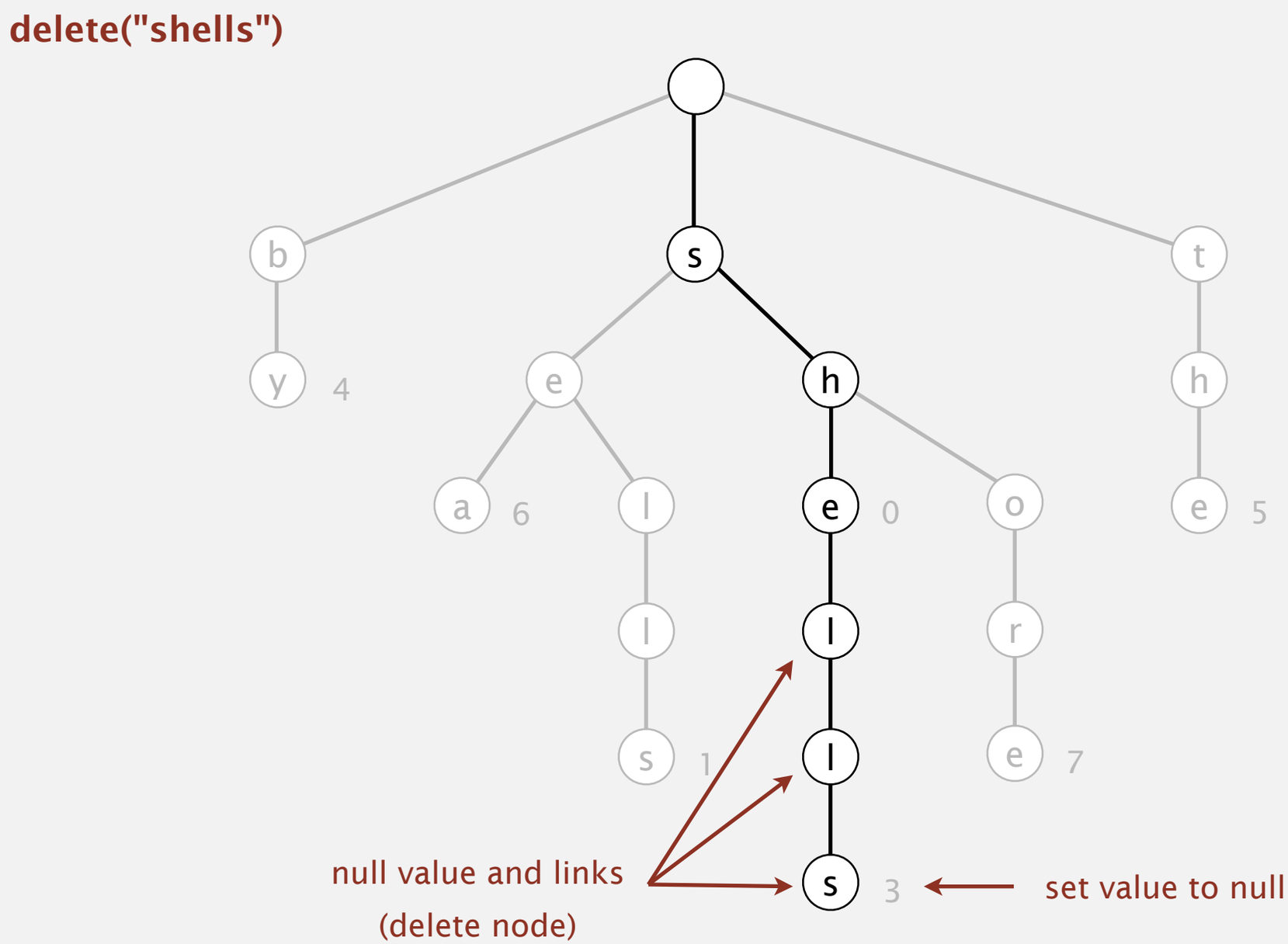
Space. R null links at each leaf.
(but sublinear space possible if many short strings share common prefixes)

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

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).



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

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!

Digression: out of memory?

" 640 K ought to be enough for anybody. "
— (mis)attributed to Bill Gates, 1981
(commenting on the amount of RAM in personal computers)

" 64 MB of RAM may limit performance of some Windows XP features; therefore, 128 MB or higher is recommended for best performance. " *— Windows XP manual, 2002*

" 64 bit is coming to desktops, there is no doubt about that. But apart from Photoshop, I can't think of desktop applications where you would need more than 4GB of physical memory, which is what you have to have in order to benefit from this technology. Right now, it is costly. " *— Bill Gates, 2003*

Digression: out of memory?

A short (approximate) history.

machine	year	address bits	addressable memory	typical actual memory	cost
PDP-8	1960s	12	6 KB	6 KB	$16K
PDP-10	1970s	18	256 KB	256 KB	$1M
IBM S/360	1970s	24	4 MB	512 KB	$1M
VAX	1980s	32	4 GB	1 MB	$1M
Pentium	1990s	32	4 GB	1 GB	$1K
Xeon	2000s	64	enough	4 GB	$100
??	future	128+	enough	enough	$1

" 512-bit words ought to be enough for anybody. "

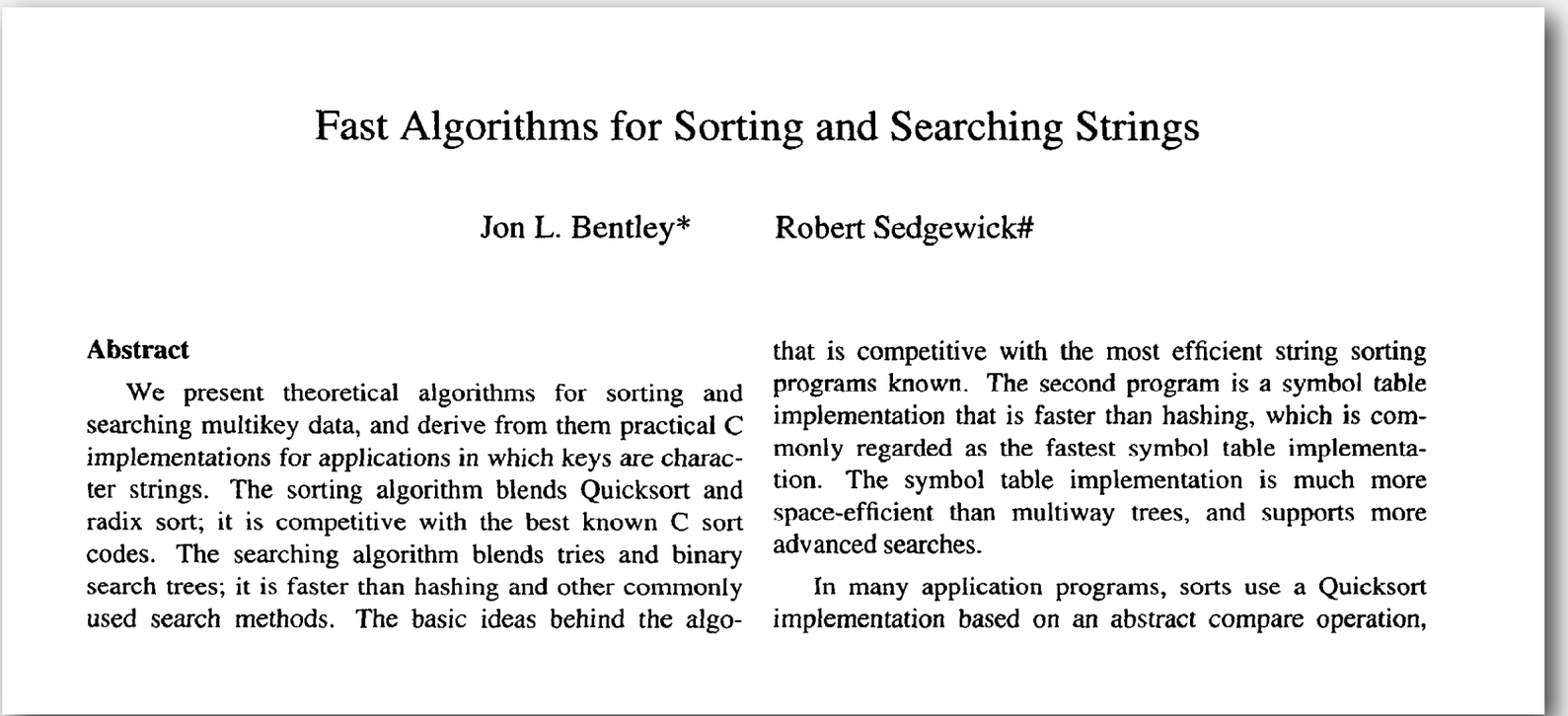
— Kevin Wayne, 1995

Tries

- R-way tries
- Ternary search tries
- Character-based operations

Ternary search tries

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



Fast Algorithms for Sorting and Searching Strings

Jon L. Bentley* Robert Sedgewick#

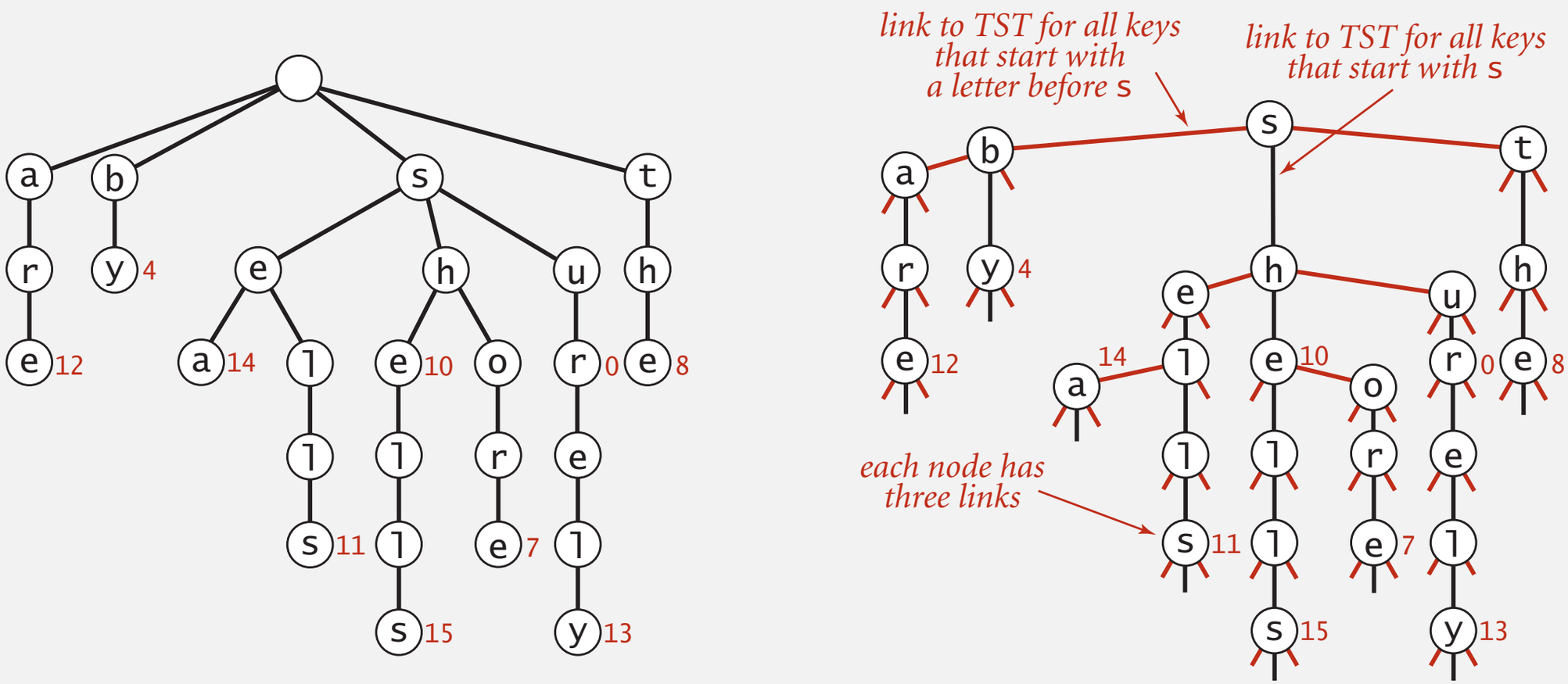
Abstract

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 three children: smaller (left), equal (middle), larger (right).



TST representation of a trie

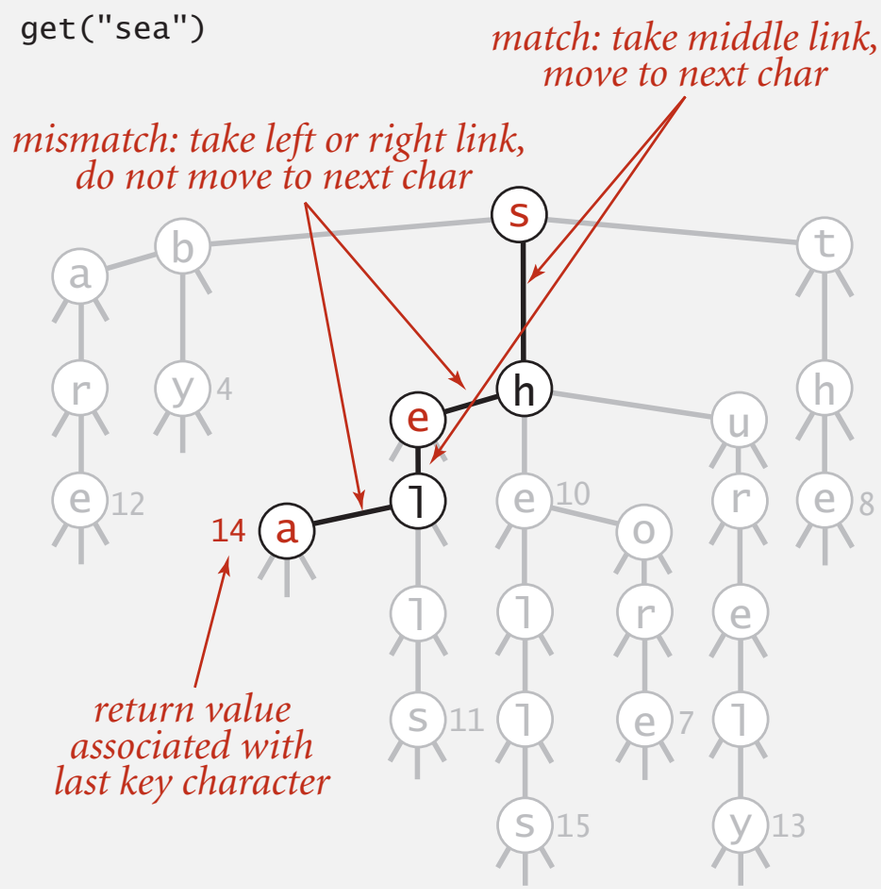
Search in a TST

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.

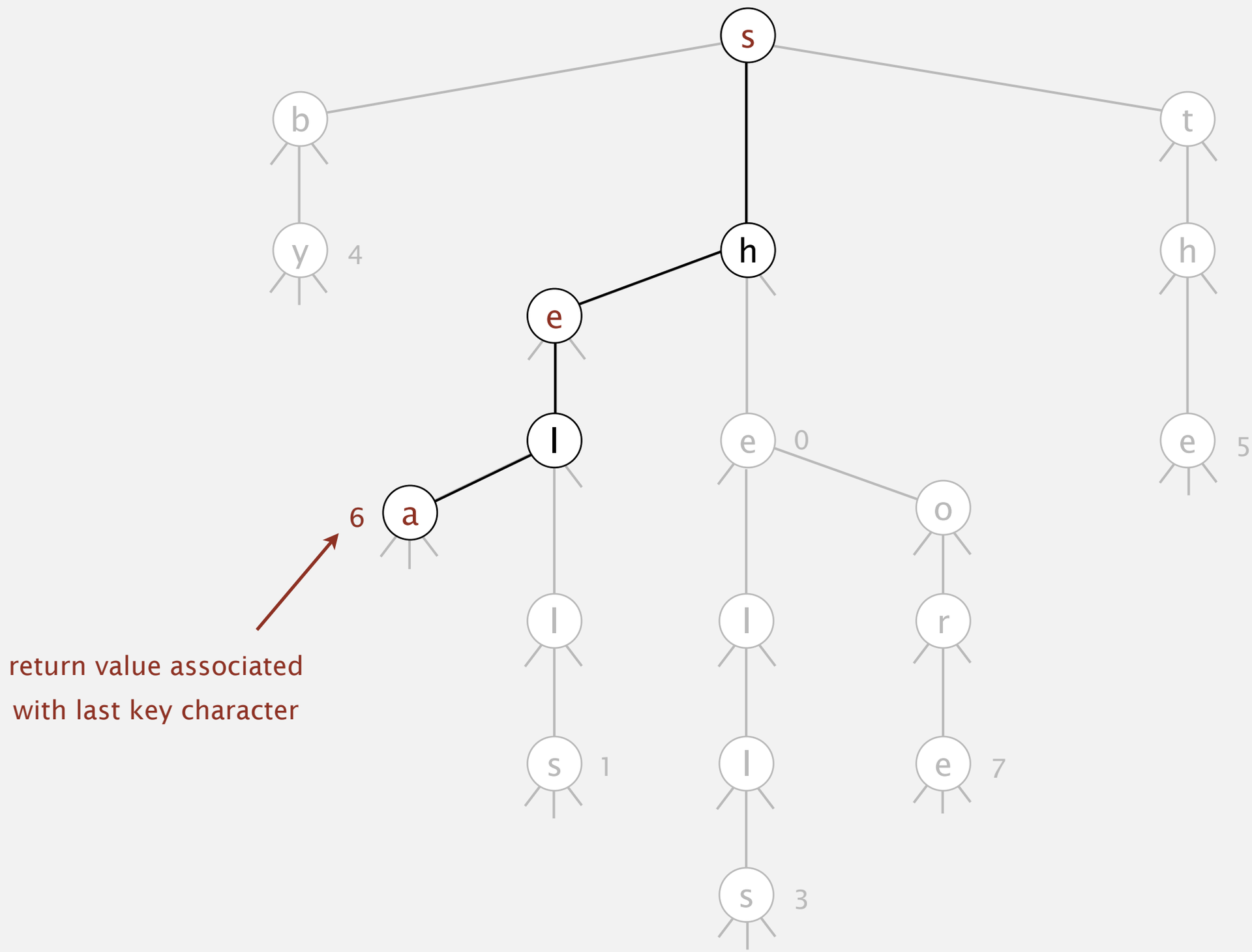
Search hit. Node where search ends has a non-null value.

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



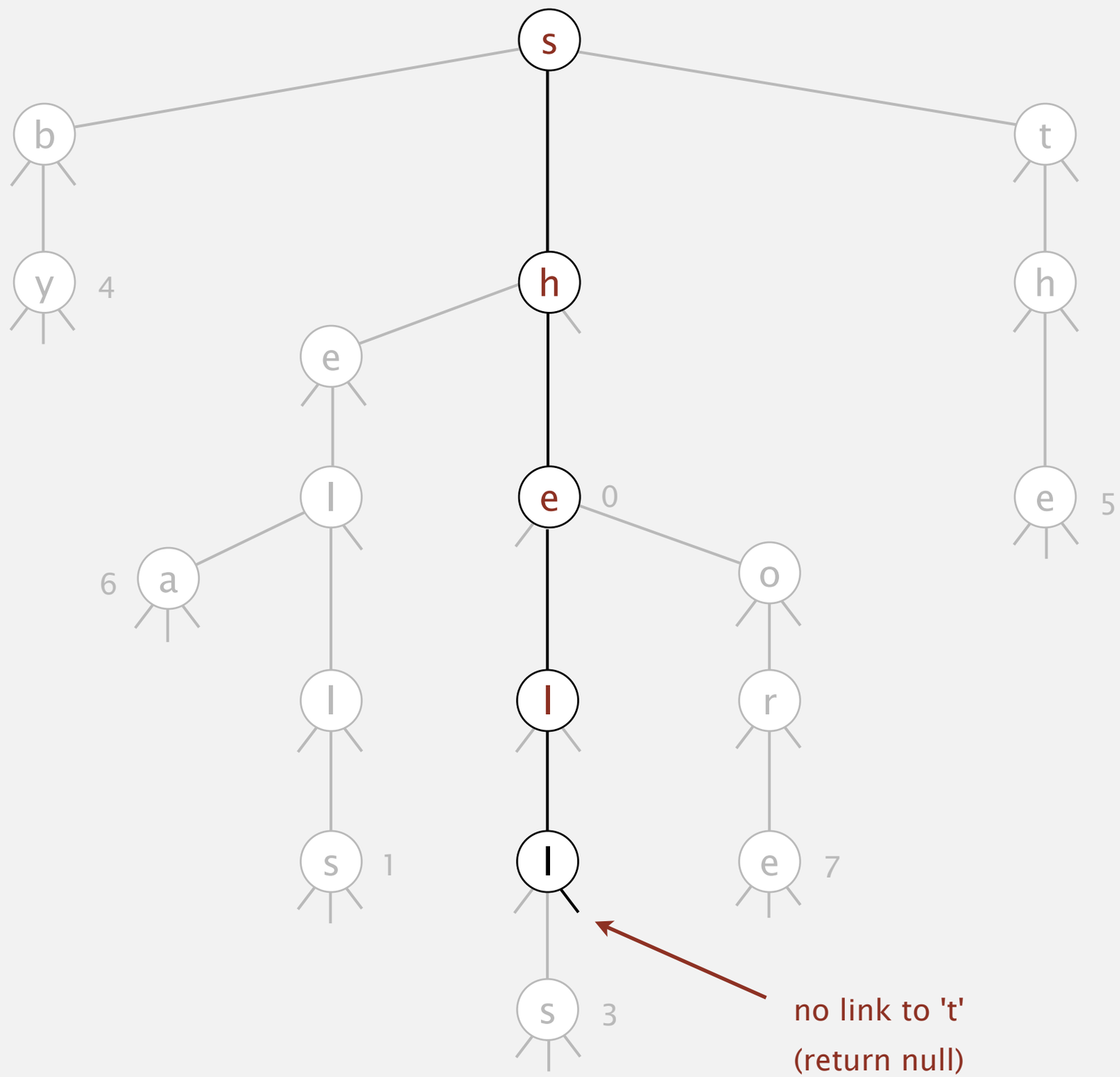
Search in a TST

get("sea")



Search in a TST

get("shelter")

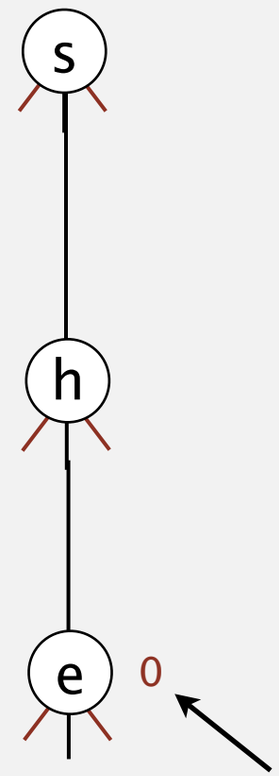


Ternary search trie insertion demo

ternary search trie

Ternary search trie insertion demo

put("she", 0)



key is sequence
of characters from
root to value
using middle links

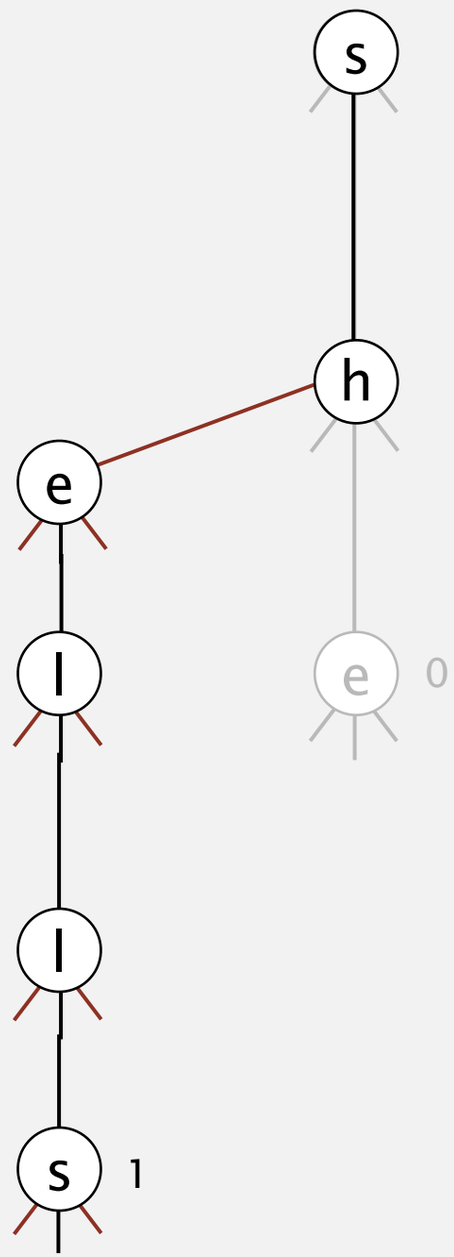
value is in node
corresponding to
last character

Ternary search trie insertion demo

put("she", 0)

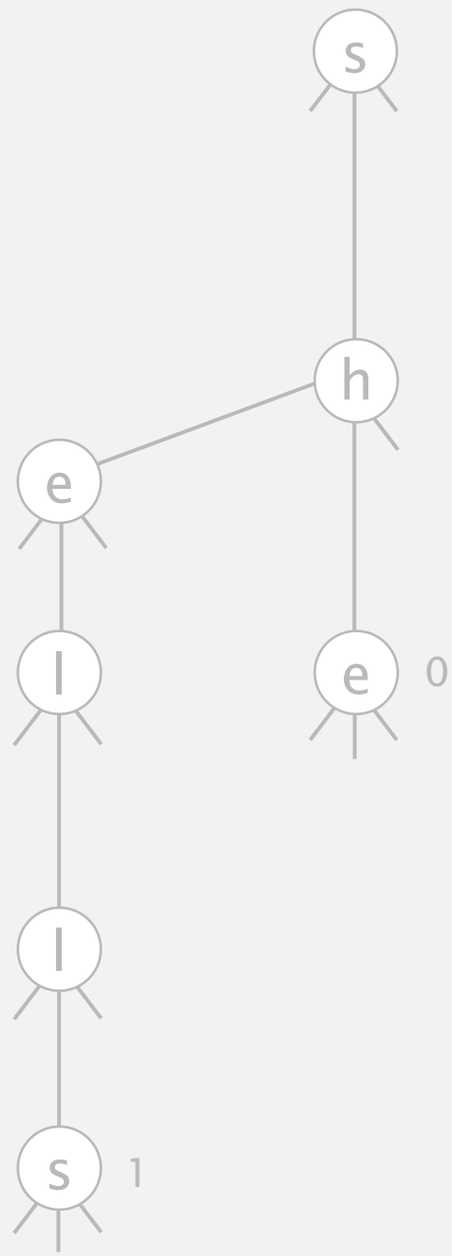
Ternary search trie insertion demo

put("sells", 1)



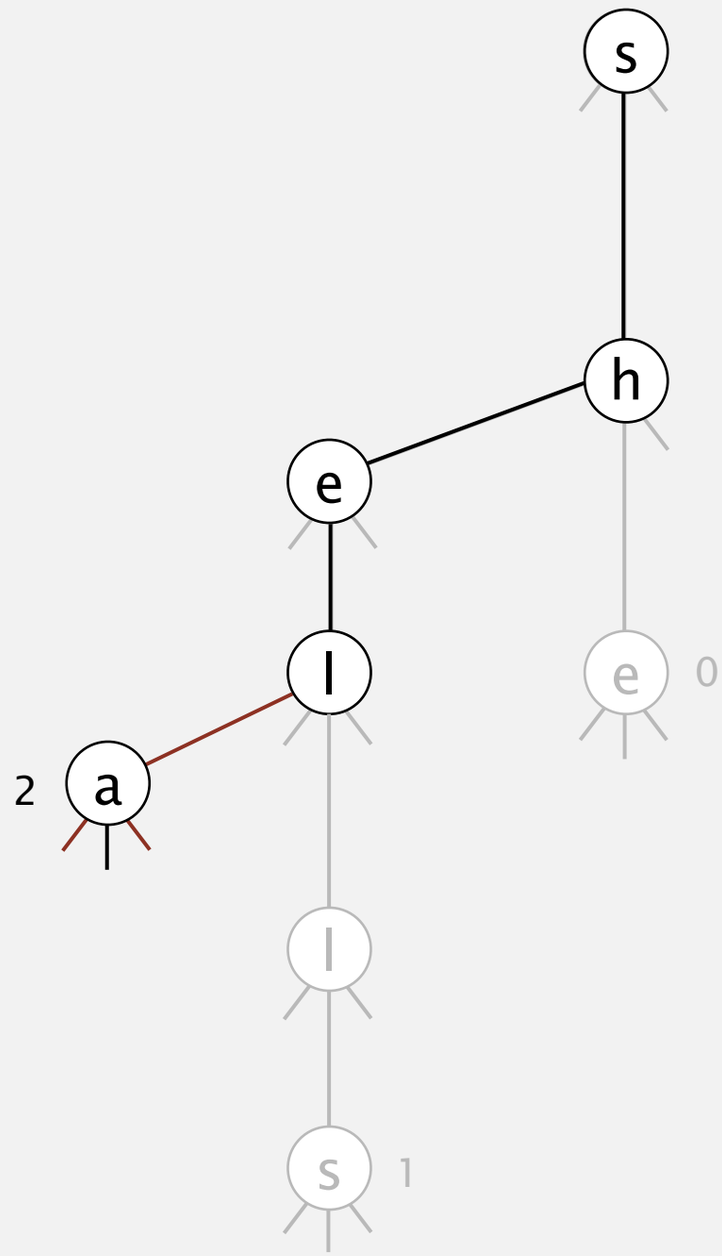
Ternary search trie insertion demo

ternary search trie



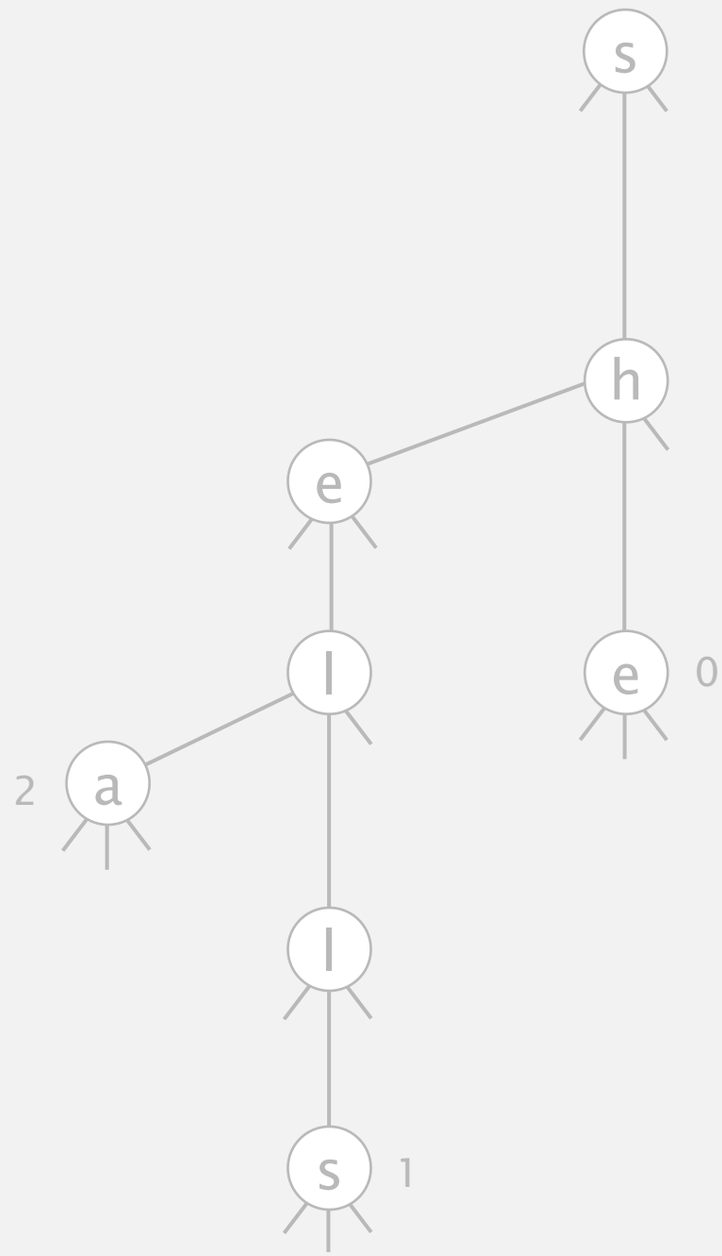
Ternary search trie insertion demo

put("sea", 2)



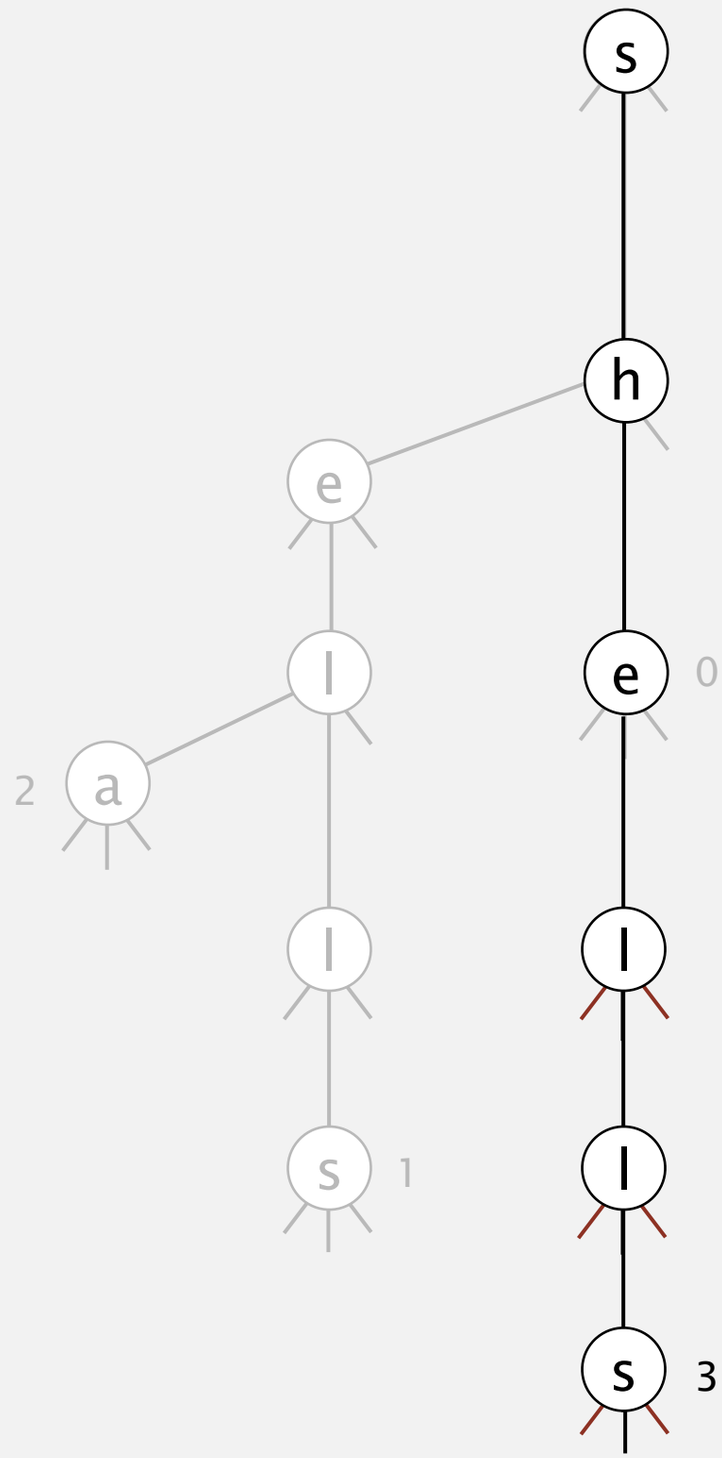
Ternary search trie insertion demo

ternary search trie



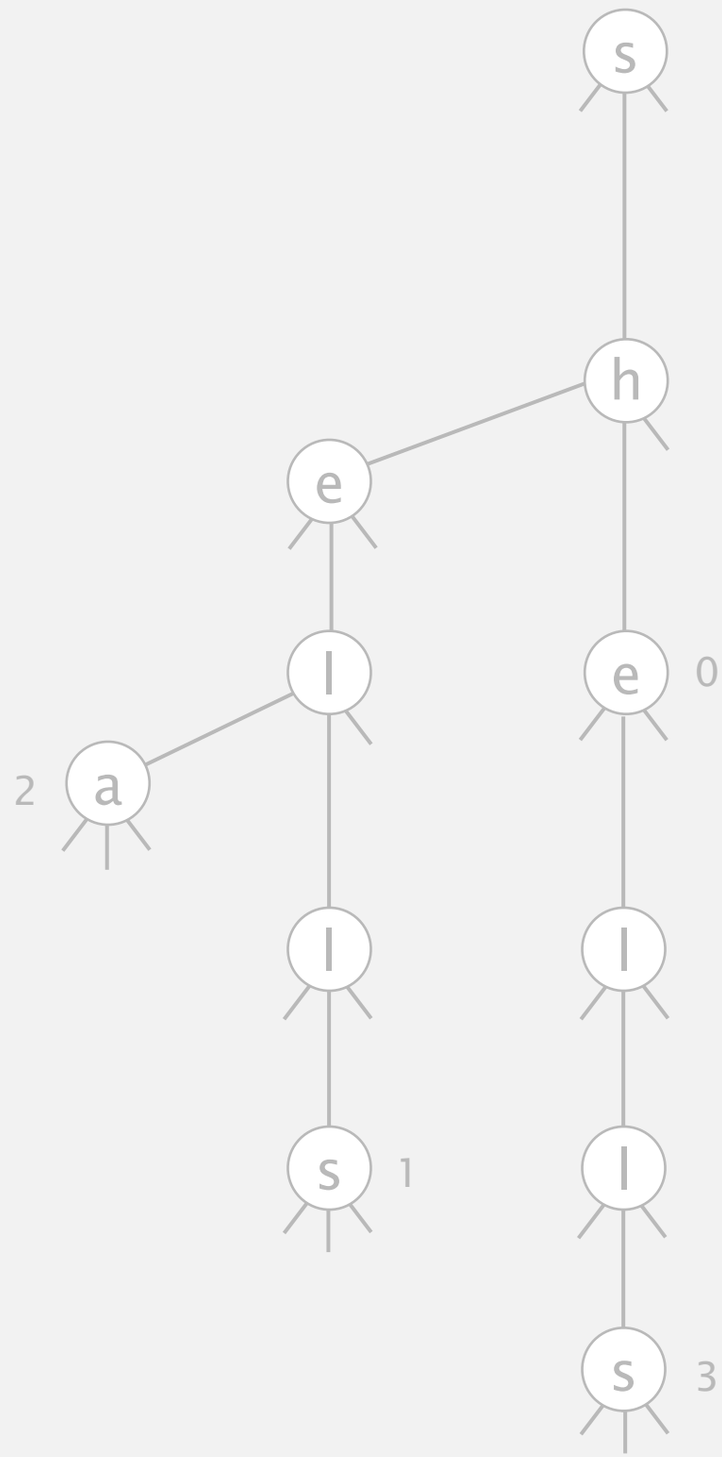
Ternary search trie insertion demo

put("shells", 3)



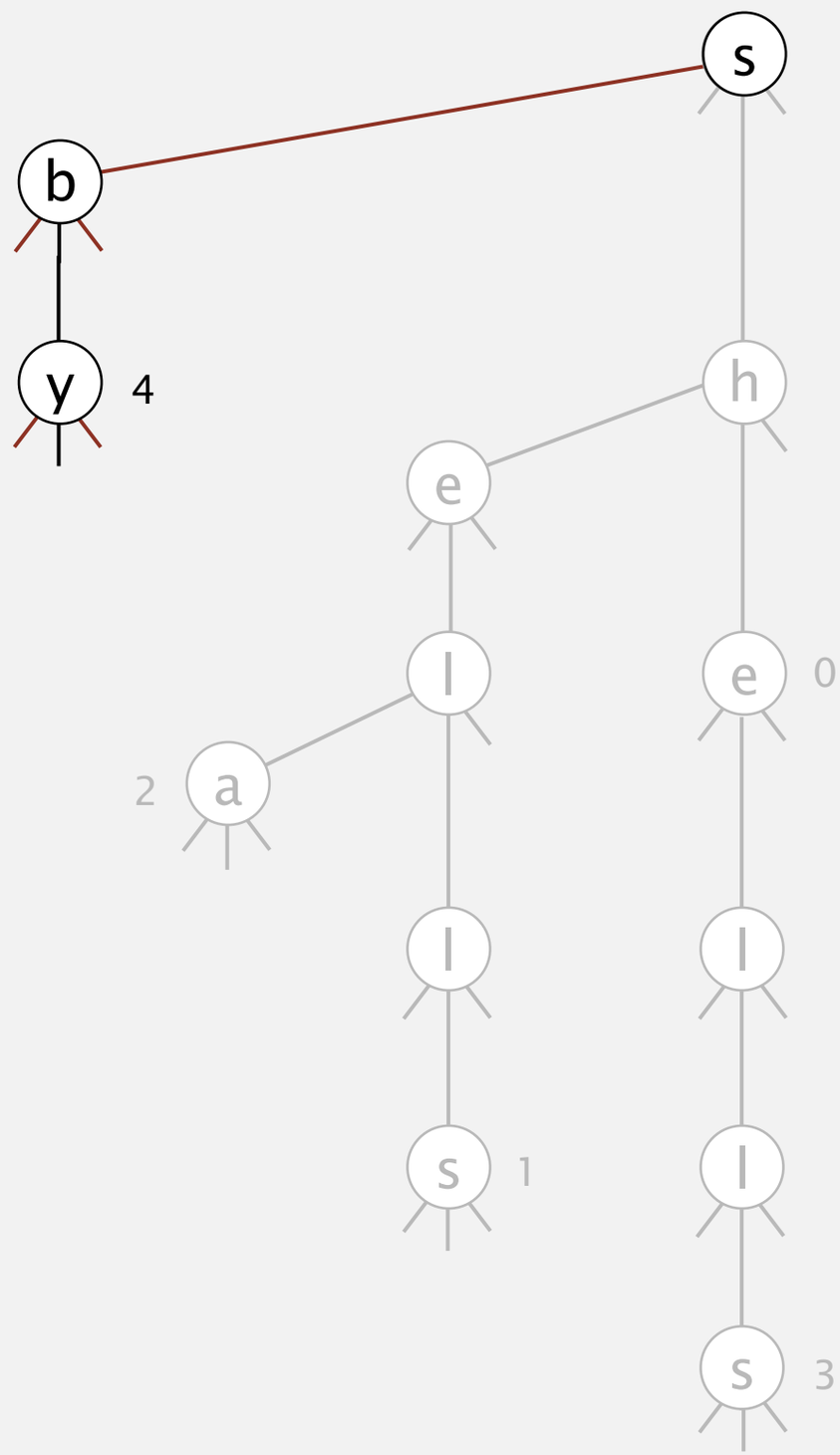
Ternary search trie insertion demo

ternary search trie



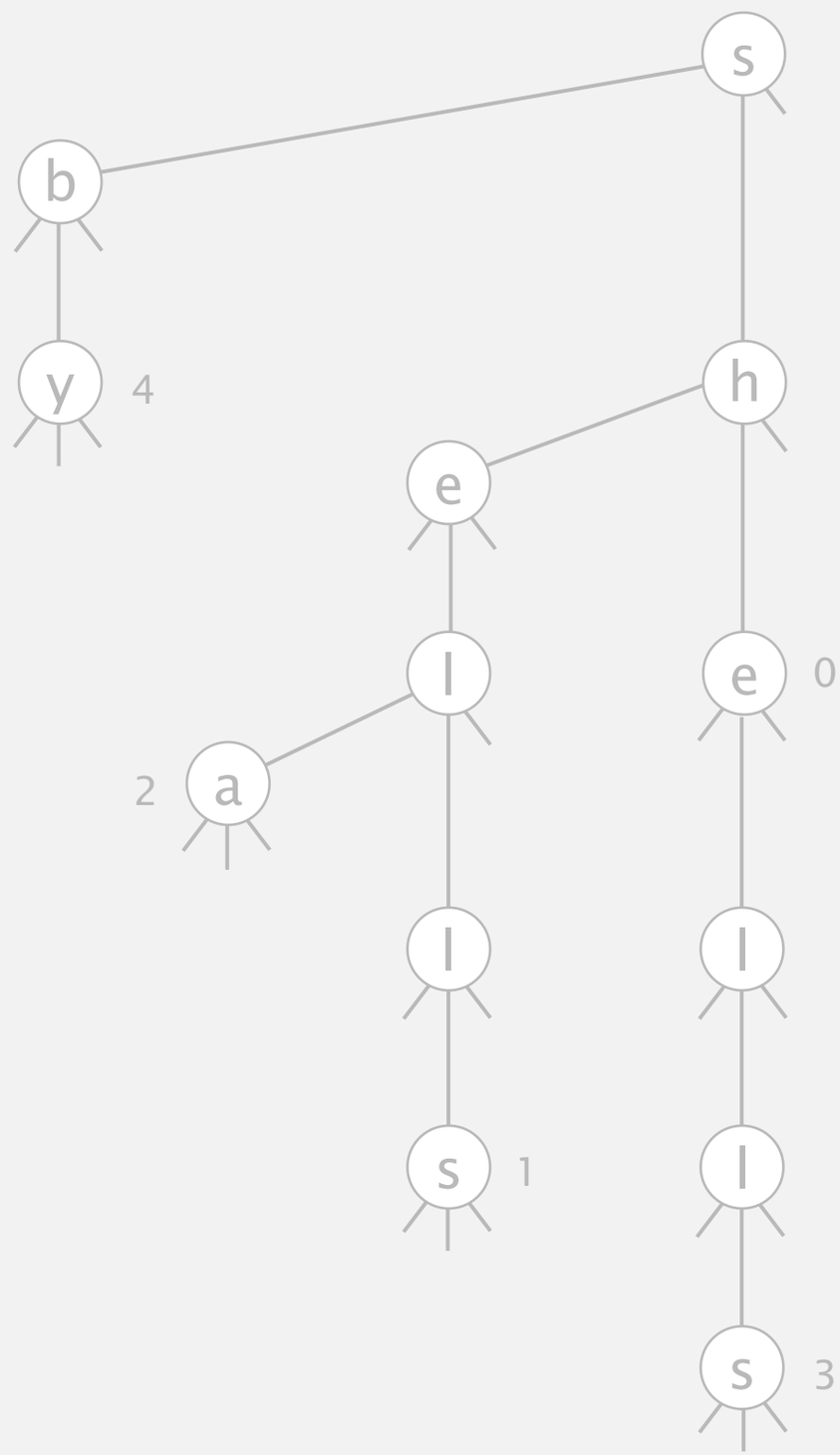
Ternary search trie insertion demo

put("by", 4)



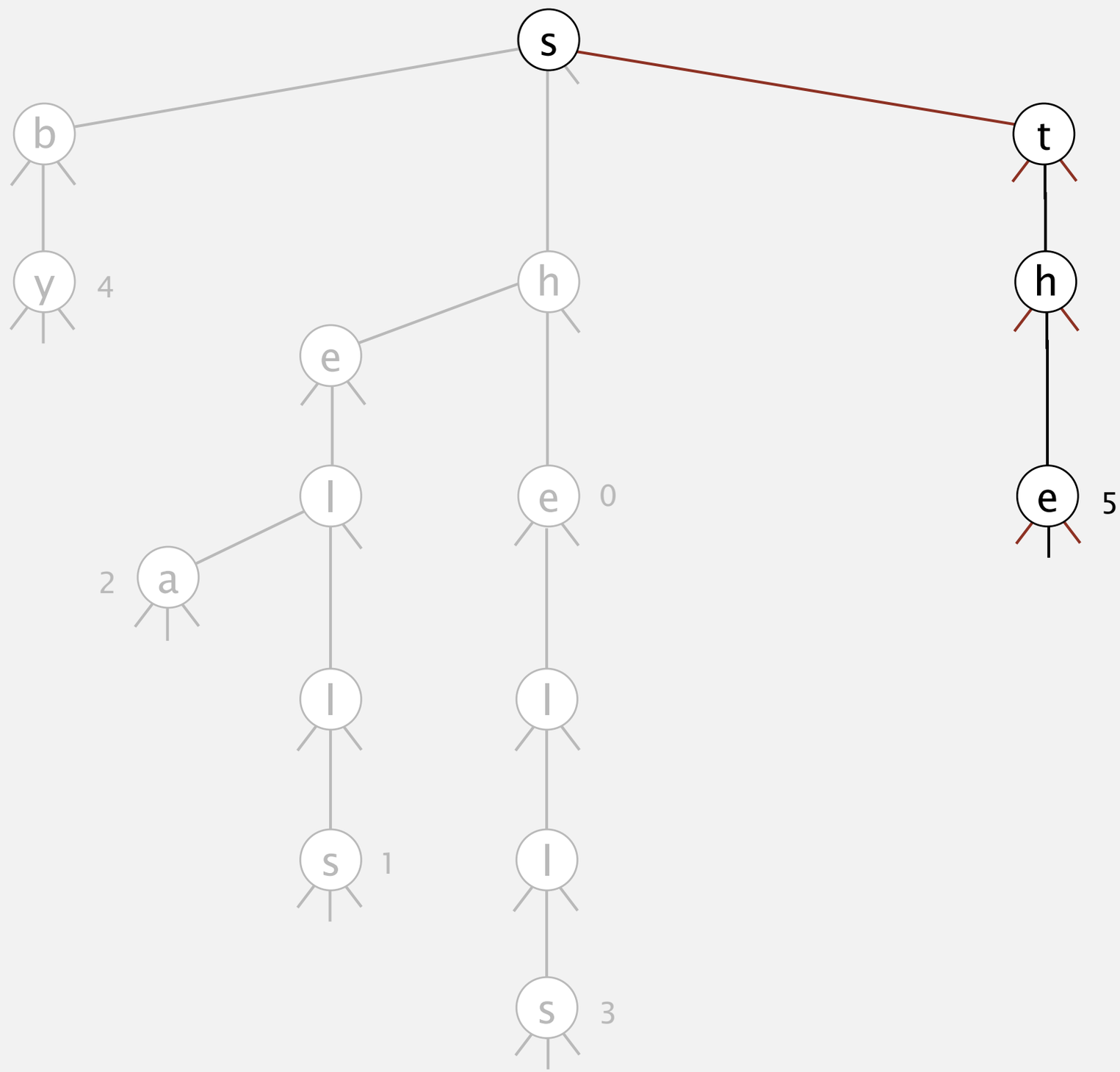
Ternary search trie insertion demo

ternary search trie



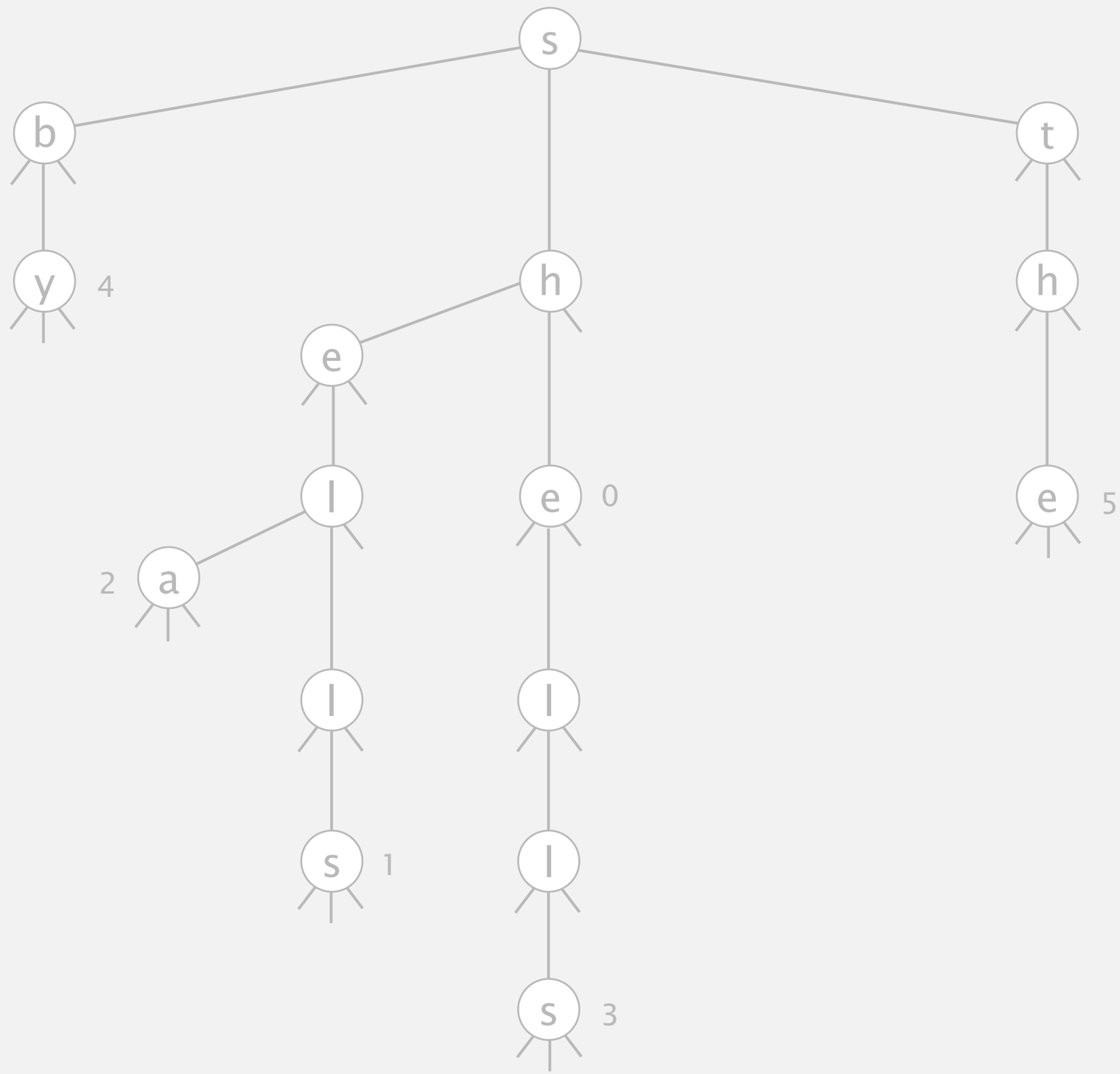
Ternary search trie insertion demo

put("the", 5)



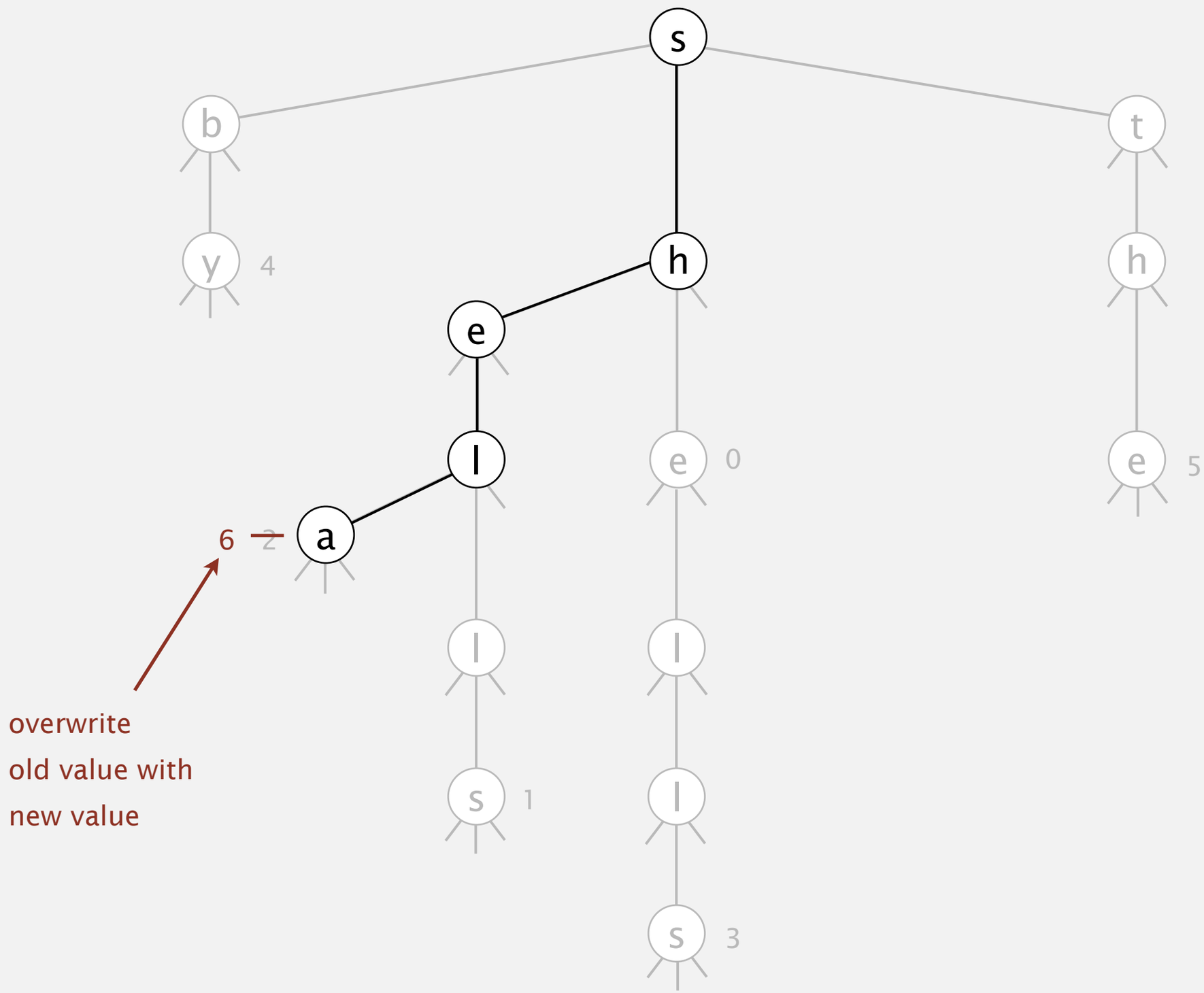
Ternary search trie insertion demo

ternary search trie



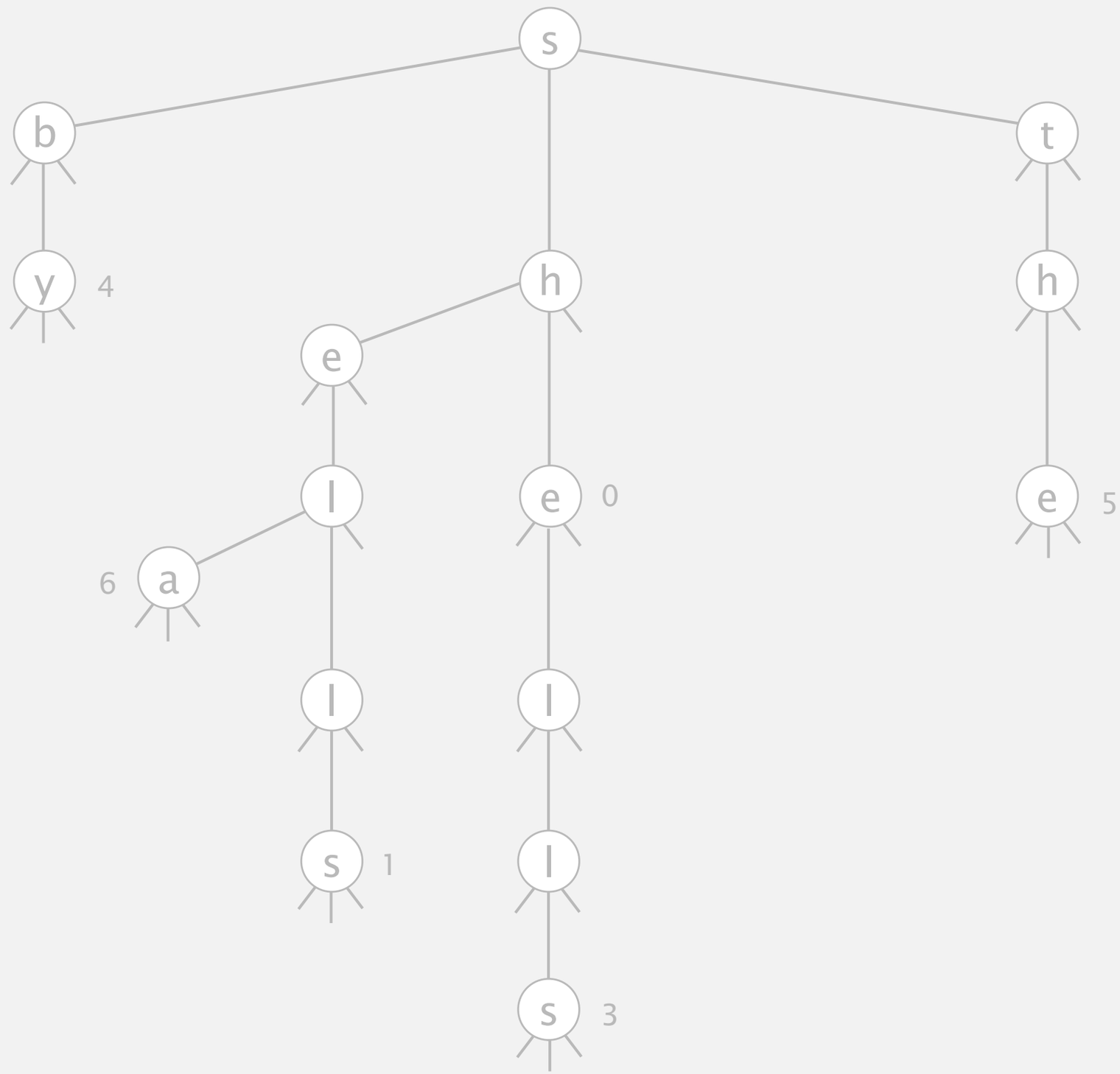
Ternary search trie insertion demo

put("sea", 6)



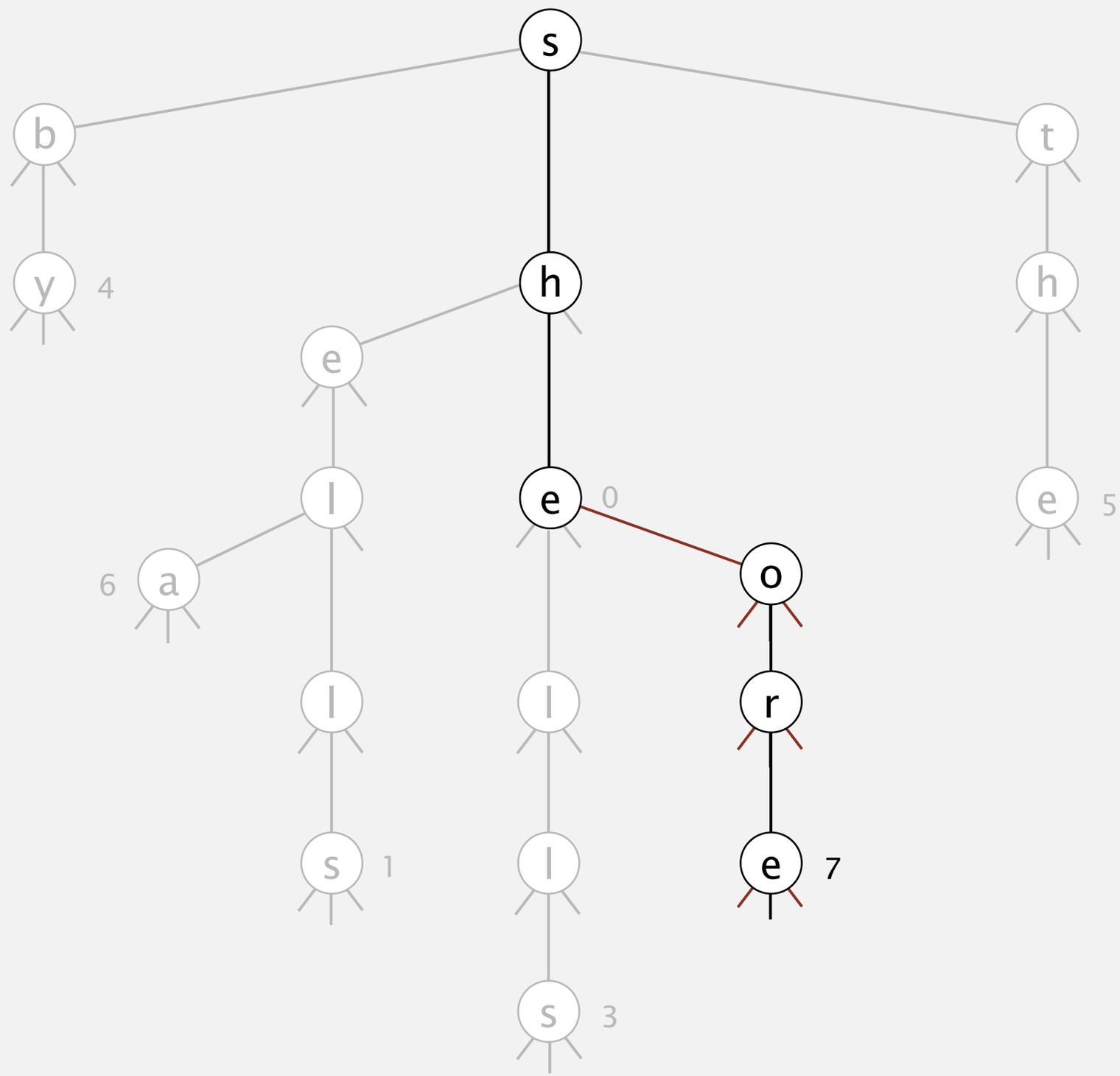
Ternary search trie insertion demo

ternary search trie



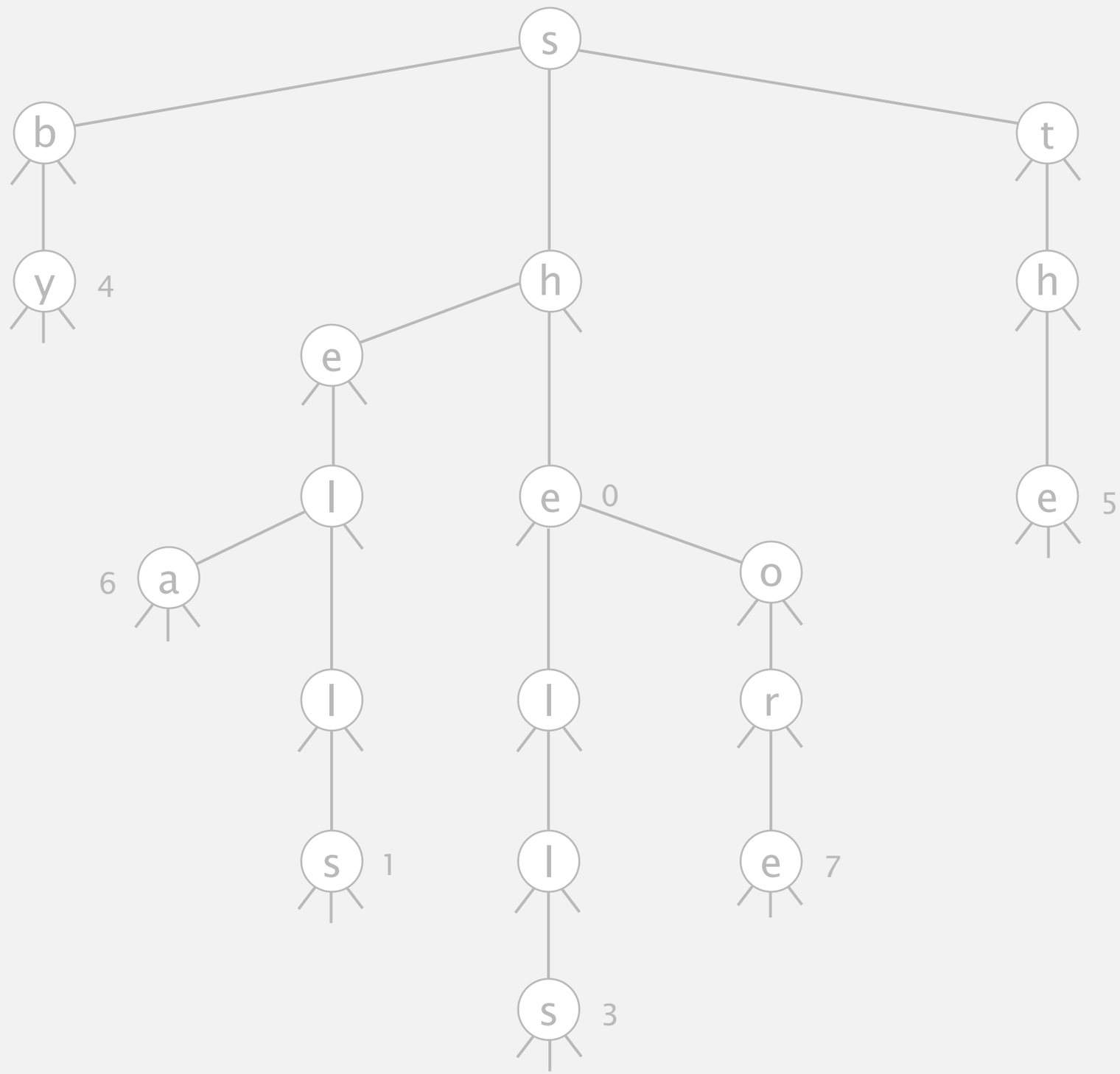
Ternary search trie insertion demo

put("shore", 7)



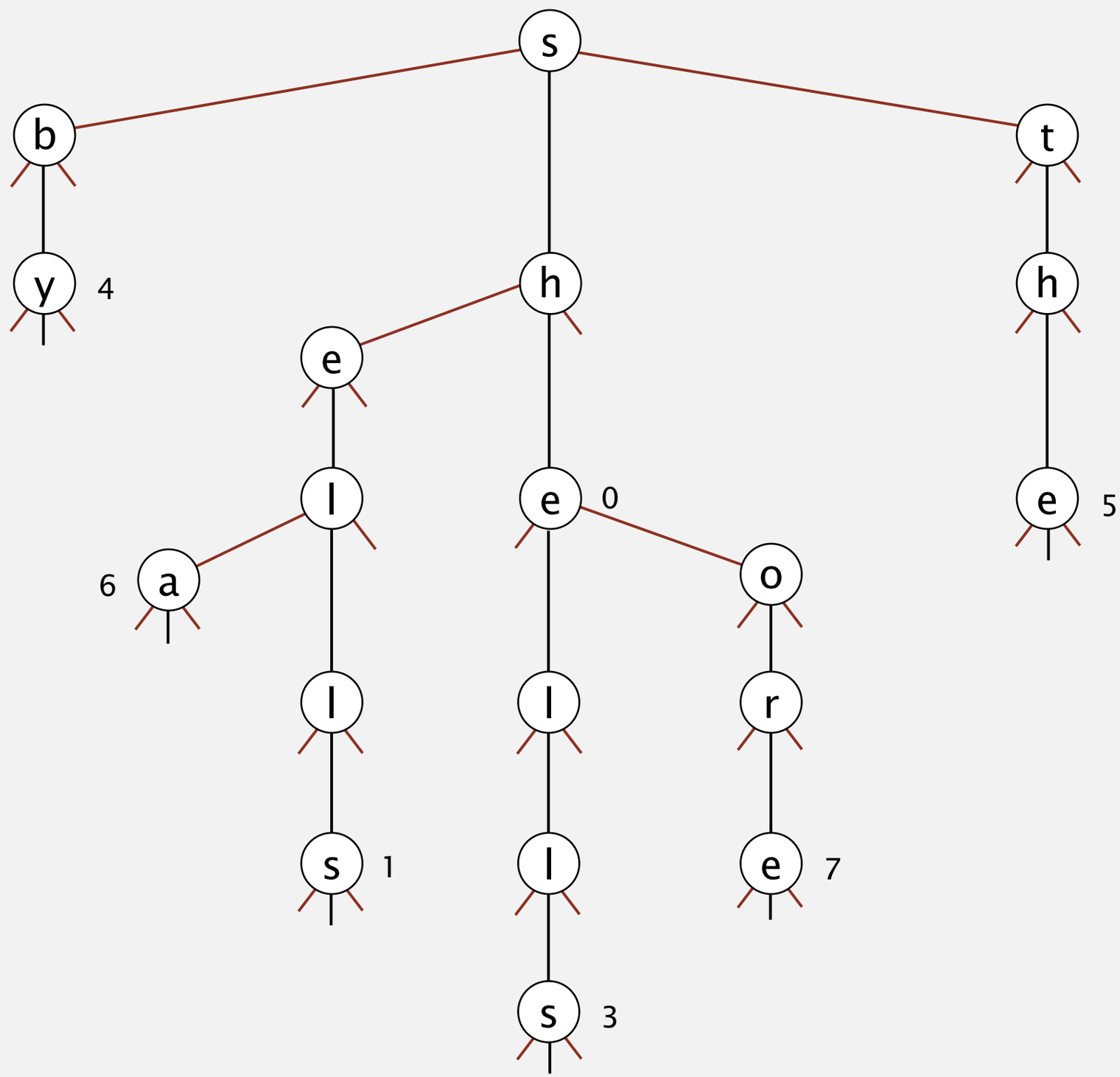
Ternary search trie insertion demo

ternary search trie



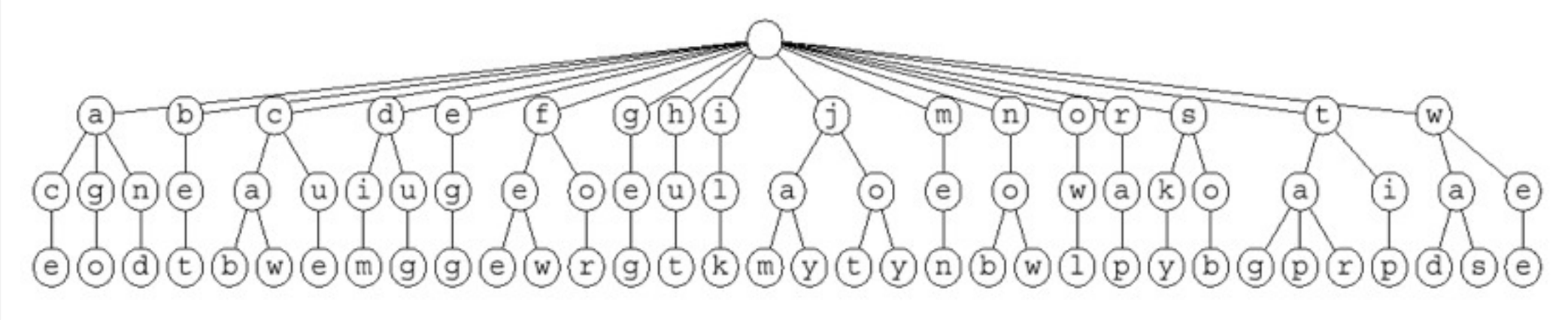
Ternary search trie insertion demo

ternary search trie



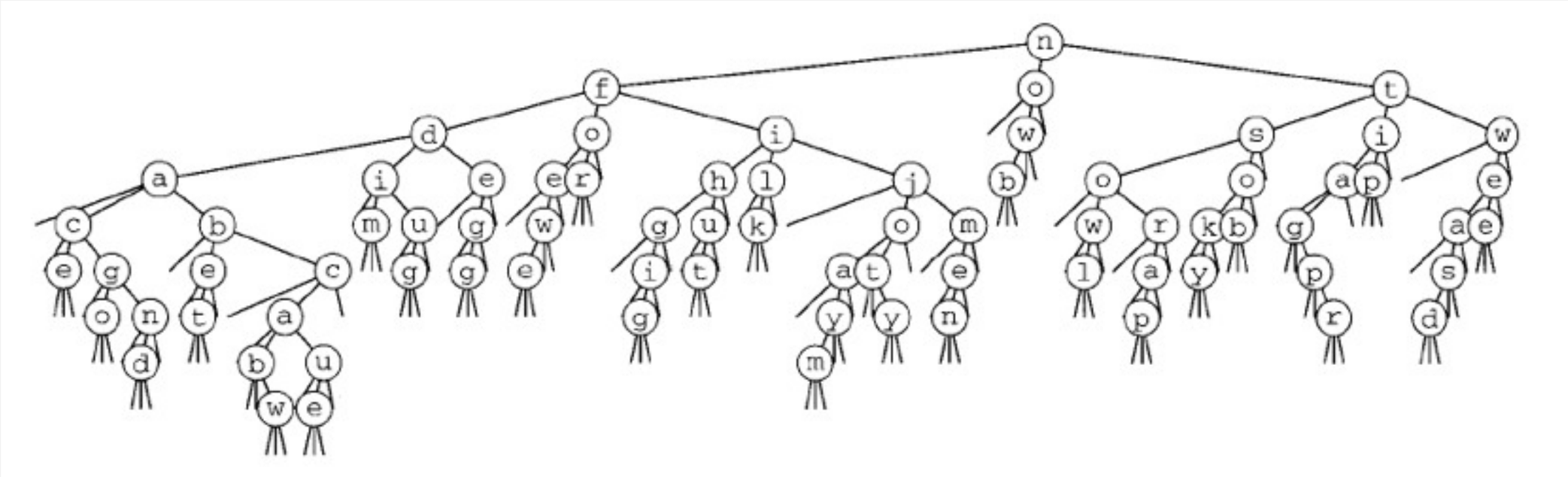
26-way trie vs. TST

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)

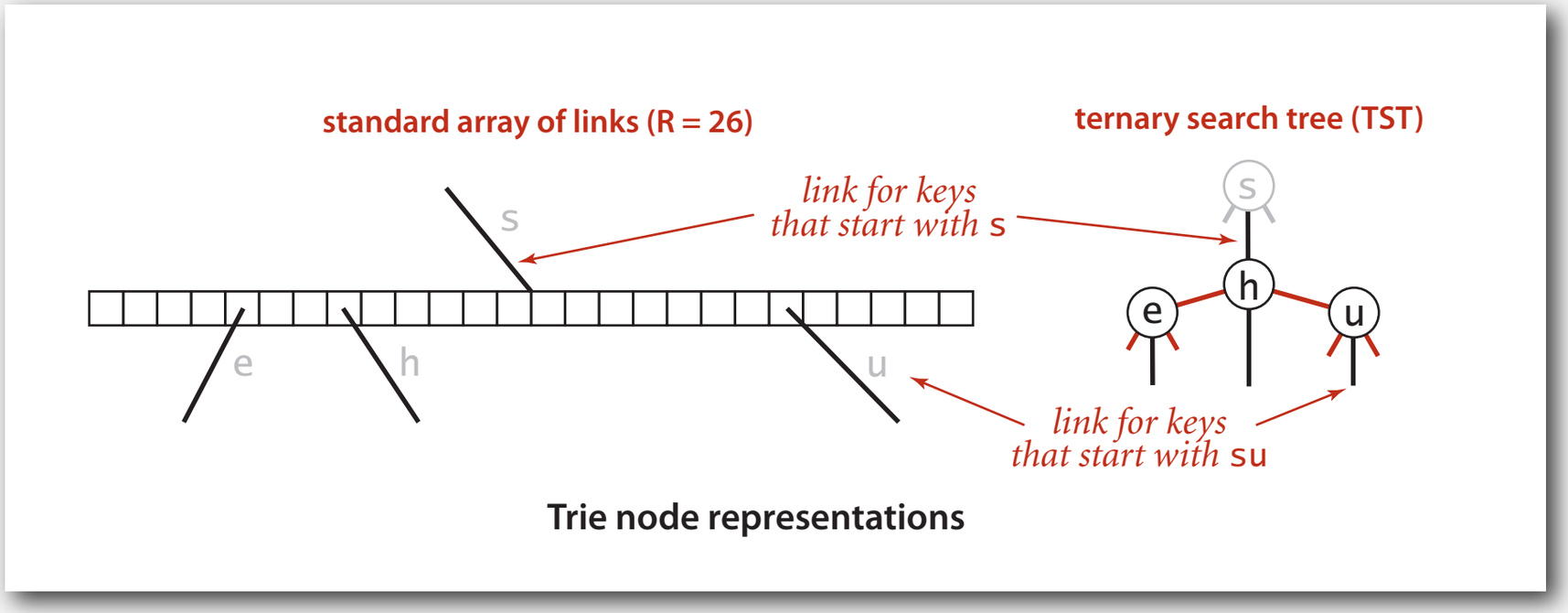
now
for
tip
ilk
dim
tag
jot
sob
nob
sky
hut
ace
bet
men
egg
few
jay
owl
joy
rap
gig
wee
was
cab
wad
caw
cue
fee
tap
ago
tar
jam
dug
and

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

	character accesses (typical case)				dedup	
implementation	search hit	search miss	insert	space (references)	moby.txt	actors.txt
red-black BST	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	$4 N$	1.40	97.4
hashing (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 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]

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*

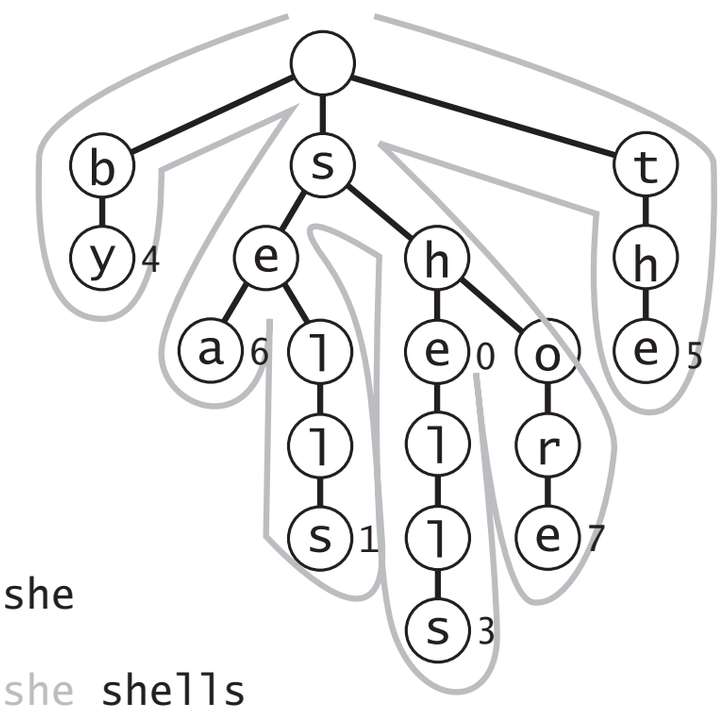
Remark. Can also add other ordered ST methods, e.g., `floor()` and `rank()`.

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.

```
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
```



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);
}
```

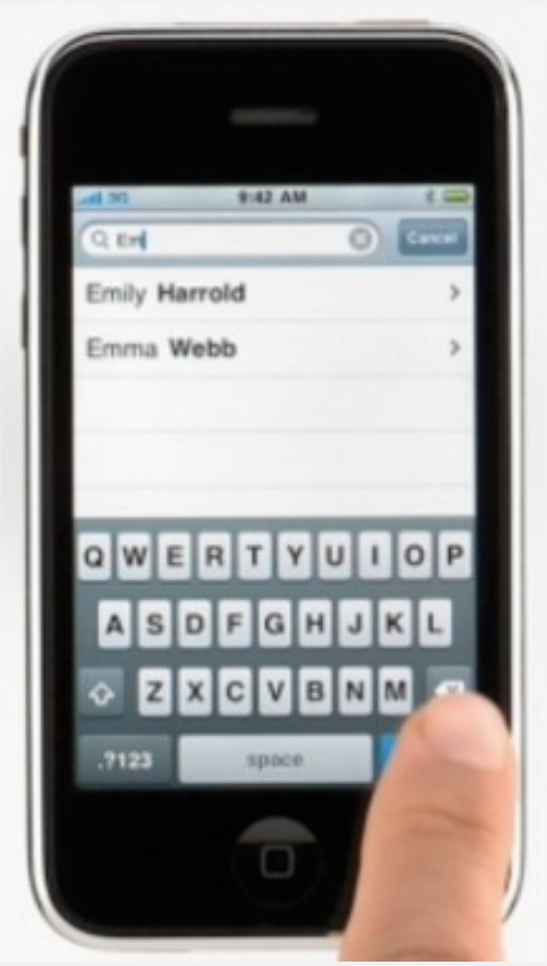
sequence of characters on path from root to x

Prefix matches

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

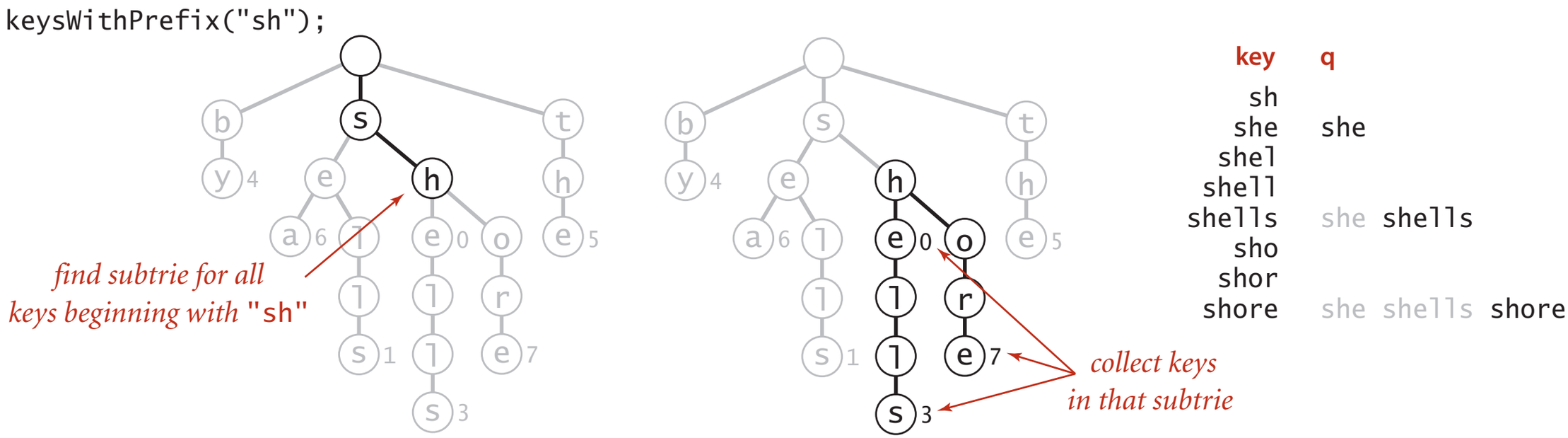
Ex. Autocomplete in a cell phone, search bar, text editor, or shell.

- 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.



Prefix match in a trie

```
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.

co....er	.c...c.
coalizer	acresce
coberger	acroach
codifier	acuracy
cofaster	octarch
cofather	science
cognizer	scranch
cohelper	scratch
colander	scrauch
coleader	screich
...	scrinch
compiler	scritch
...	scrunch
composer	scudick
computer	scutock
cowkeper	

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. 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)

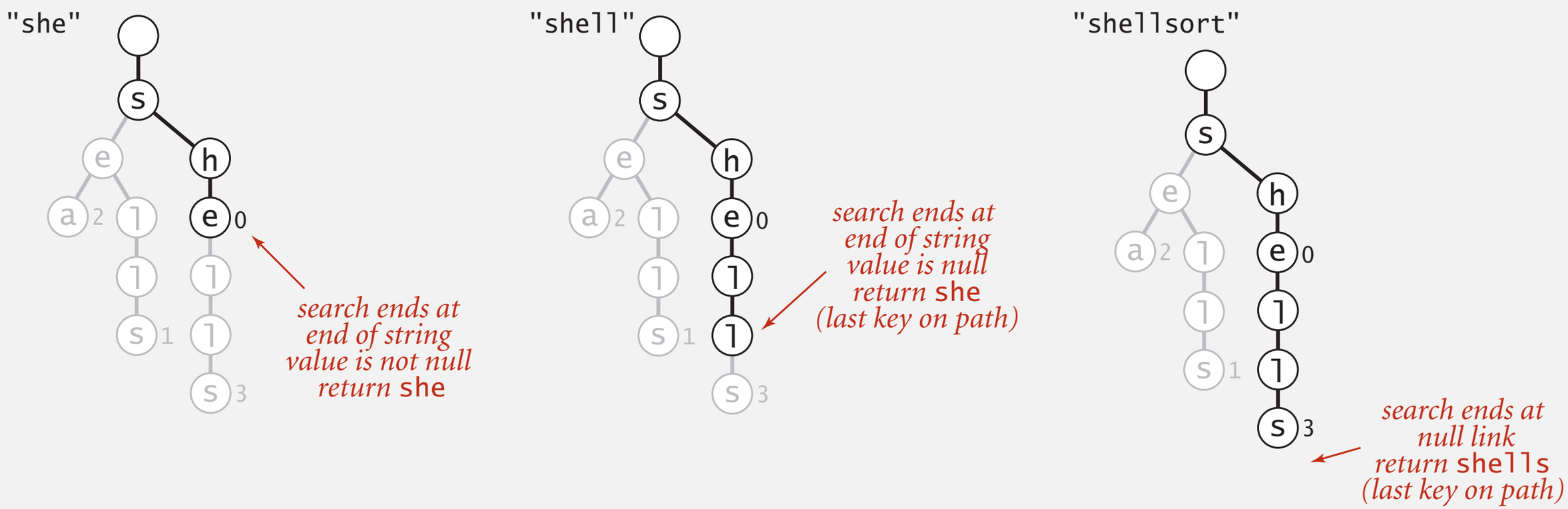
```
longestPrefixOf("128.112.136.11") = "128.112.136"
longestPrefixOf("128.112.100.16") = "128.112"
longestPrefixOf("128.166.123.45") = "128"
```

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

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.



Possibilities for `longestPrefixOf()`

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.

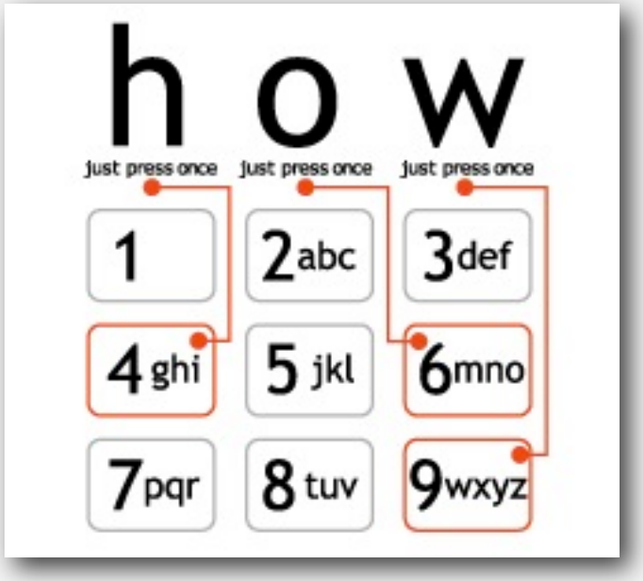
"a much faster and more fun way to enter text"

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`



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

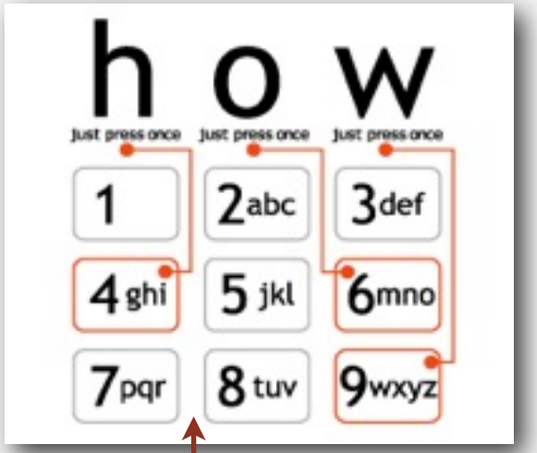
http://www.t9.com/images/how.gif

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" ???

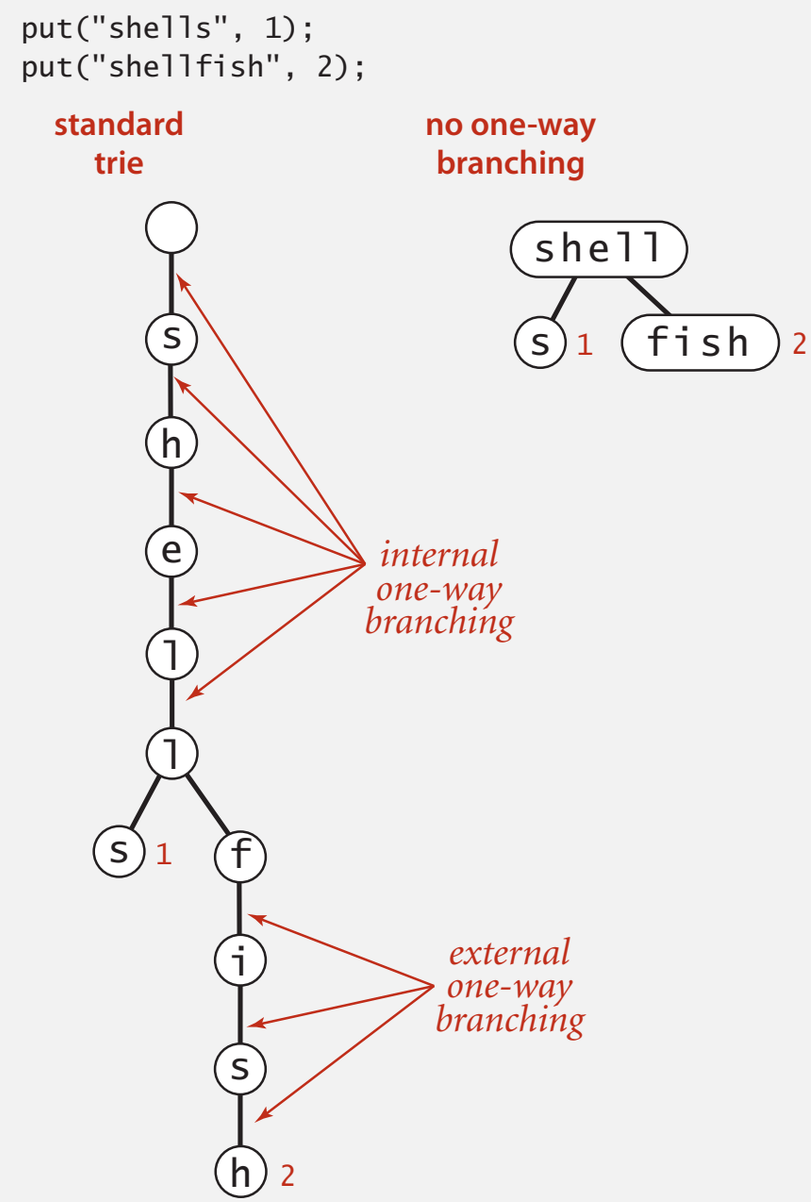
A world without “s” ??

```
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 uppor t
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.



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.

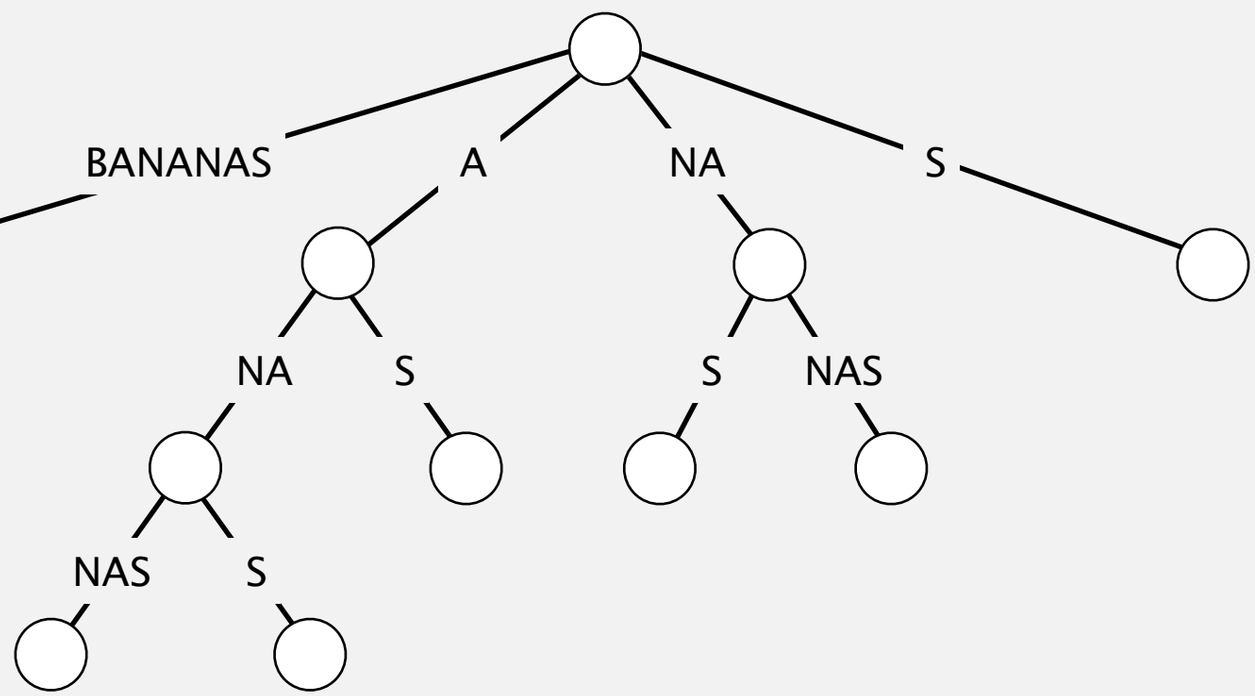
Also known as: crit-bit tree, radix tree.

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 (!!!)