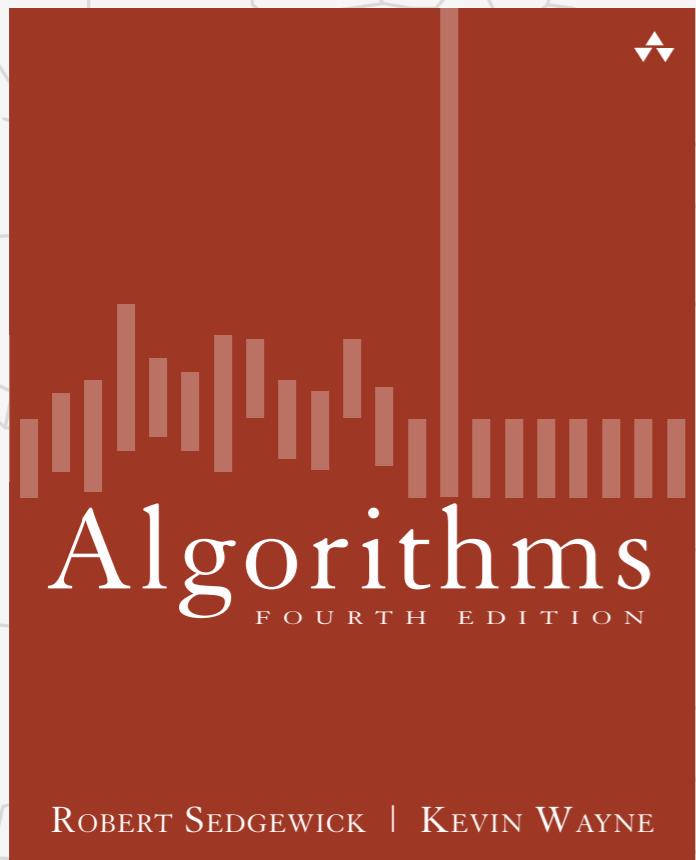


Algorithms

FARAAZ SARESHWALA



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

5.1 STRING SORTS

- ▶ strings in Java
- ▶ key-indexed counting
- ▶ LSD radix sort
- ▶ MSD radix sort
- ▶ 3-way radix quicksort
- ▶ suffix arrays

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

5.1 STRING SORTS

- ▶ strings in Java
- ▶ key-indexed counting
- ▶ LSD radix sort
- ▶ MSD radix sort
- ▶ 3-way radix quicksort
- ▶ suffix arrays

String processing

String. Sequence of characters.

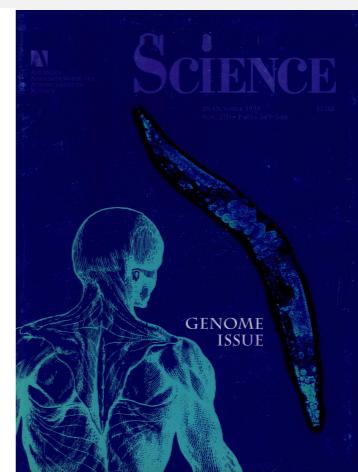
String processing

String. Sequence of characters.

Important fundamental abstraction.

- Genomic sequences.
- Information processing.
- 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



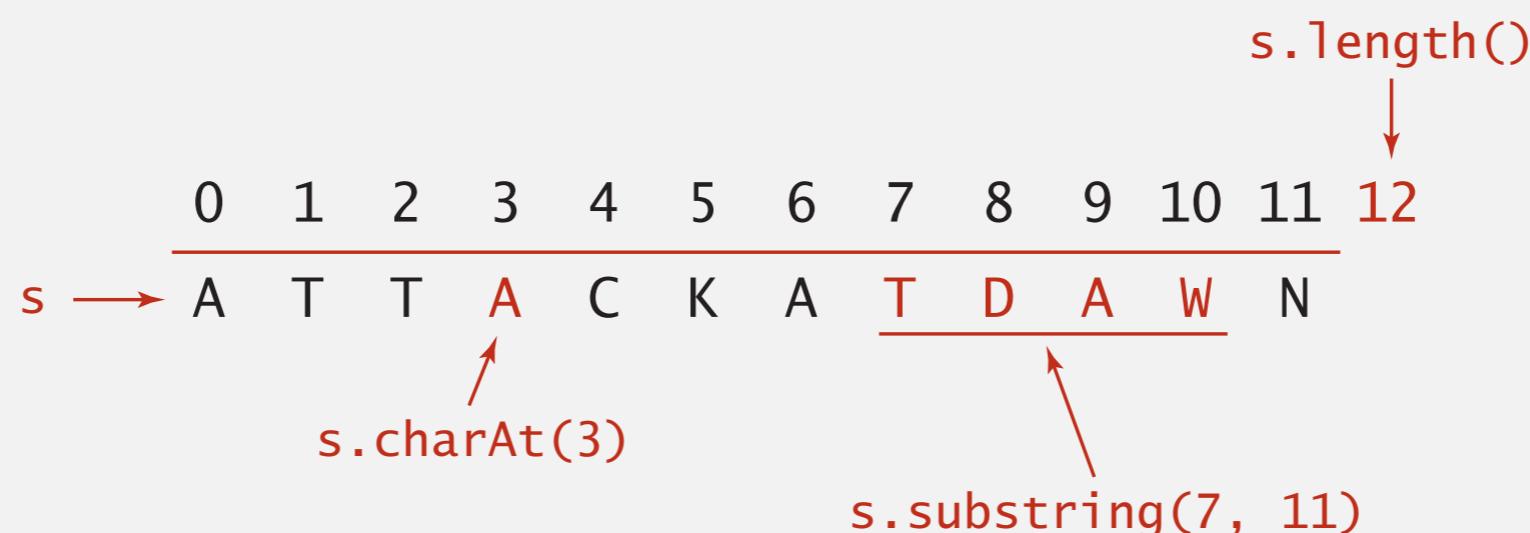
The String data type

String data type in Java. Immutable sequence of characters.

Length. Number of characters.

Indexing. Get the i^{th} character.

Concatenation. Concatenate one string to the end of another.



The String data type: immutability

The String data type: immutability

Q. Why immutable?

The String data type: immutability

Q. Why immutable?

A. All the usual reasons.

- Can use as keys in symbol table.
- Don't need to defensively copy.
- Ensures consistent state.
- Supports concurrency.

The String data type: immutability

Q. Why immutable?

A. All the usual reasons.

- Can use as keys in symbol table.
- Don't need to defensively copy.
- Ensures consistent state.
- Supports concurrency.
- Improves security.

```
public class FileInputStream
{
    private String filename;
    public FileInputStream(String filename)
    {
        if (!allowedToReadFile(filename))
            throw new SecurityException();
        this.filename = filename;
    }
    ...
attacker could bypass security if string type were mutable
}
```

String performance trap

Q. How to build a long string, one character at a time?

String performance trap

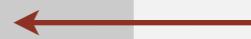
Q. How to build a long string, one character at a time?

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

String performance trap

Q. How to build a long string, one character at a time?

```
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

String performance trap

Q. How to build a long string, one character at a time?

```
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

A. Use StringBuilder data type (mutable char[] array).

String performance trap

Q. How to build a long string, one character at a time?

```
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

A. Use StringBuilder data type (mutable char[] array).

```
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

Comparing two strings

Q. How many character compares to compare two strings of length W ?

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

Comparing two strings

Q. How many character compares to compare two strings of length W ?

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

Running time. Proportional to length of longest common prefix.

- Proportional to W in the worst case.
- But, often sublinear in W .

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

5.1 STRING SORTS

- ▶ strings in Java
- ▶ 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.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$\frac{1}{2} N^2$	$\frac{1}{4} N^2$	1	✓	compareTo()
mergesort	$N \lg N$	$N \lg N$	N	✓	compareTo()
quicksort	$1.39 N \lg N$ *	$1.39 N \lg N$	$c \lg N$		compareTo()
heapsort	$2 N \lg N$	$2 N \lg N$	1		compareTo()

* probabilistic

Review: summary of the performance of sorting algorithms

Frequency of operations.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$\frac{1}{2} N^2$	$\frac{1}{4} N^2$	1	✓	compareTo()
mergesort	$N \lg N$	$N \lg N$	N	✓	compareTo()
quicksort	$1.39 N \lg N$ *	$1.39 N \lg N$	$c \lg N$		compareTo()
heapsort	$2 N \lg N$	$2 N \lg N$	1		compareTo()

* probabilistic

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

Review: summary of the performance of sorting algorithms

Frequency of operations.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$\frac{1}{2} N^2$	$\frac{1}{4} N^2$	1	✓	compareTo()
mergesort	$N \lg N$	$N \lg N$	N	✓	compareTo()
quicksort	$1.39 N \lg N$ *	$1.39 N \lg N$	$c \lg N$		compareTo()
heapsort	$2 N \lg N$	$2 N \lg N$	1		compareTo()

* probabilistic

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

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

Review: summary of the performance of sorting algorithms

Frequency of operations.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$\frac{1}{2} N^2$	$\frac{1}{4} N^2$	1	✓	compareTo()
mergesort	$N \lg N$	$N \lg N$	N	✓	compareTo()
quicksort	$1.39 N \lg N$ *	$1.39 N \lg N$	$c \lg N$		compareTo()
heapsort	$2 N \lg N$	$2 N \lg N$	1		compareTo()

* probabilistic

Lower bound. $\sim N \lg N$ 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.



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

Key-indexed counting: assumptions about keys

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

input		sorted result (by section)	
<i>name</i>	<i>section</i>		
Anderson	2	Harris	1
Brown	3	Martin	1
Davis	3	Moore	1
Garcia	4	Anderson	2
Harris	1	Martinez	2
Jackson	3	Miller	2
Johnson	4	Robinson	2
Jones	3	White	2
Martin	1	Brown	3
Martinez	2	Davis	3
Miller	2	Jackson	3
Moore	1	Jones	3
Robinson	2	Taylor	3
Smith	4	Williams	3
Taylor	3	Garcia	4
Thomas	4	Johnson	4
Thompson	4	Smith	4
White	2	Thomas	4
Williams	3	Thompson	4
Wilson	4	Wilson	4

*↑
keys are
small integers*

Key-indexed counting: assumptions about keys

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

Implication. Can use key as an array index.

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

*↑
keys are
small integers*

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]

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

↑
keys are
small integers

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

↑
*keys are
small integers*

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.

$\nearrow R = 6$

```
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++)
```

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

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++)
```

count
frequencies

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

a	0
b	2
c	3
d	1
e	2
f	1
-	3

	r count[r]
--	------------

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]++;  
  
compute  
cumulates  
→  
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++)
```

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	a[i]	r	count[r]
0	d	a	0
1	a	b	2
2	c	c	5
3	f	d	6
4	f	e	8
5	b	f	9
6	d	-	12
7	b	-	-
8	f	-	-
9	b	-	-
10	e	-	-
11	a	-	-

6 keys < d, 8 keys < e

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

13

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];
```

move
items

i	a[i]	r count[r]	aux[i]
0	d		0
1	a		1
2	c		2
3	f	a 0	3
4	f	b 2	4
5	b	c 5	5
6	d	d 6	6
7	b	e 8	7
8	f	f 9	8
9	b	- 12	9
10	e		10
11	a		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];  
  
move  
items →  
for (int i = 0; i < N; i++)  
    aux[count[a[i]]++] = a[i];  
  
for (int i = 0; i < N; i++)
```

i	a[i]	i	aux[i]
0	d	0	
1	a	1	
2	c	2	
3	f	3	
4	f	4	
5	b	5	
6	d	6	d
7	b	7	
8	f	8	
9	b	9	
10	e	10	
11	a	11	
			r count[r]
		a	0
		b	2
		c	5
		d	7
		e	8
		f	9
		-	12

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

i	a[i]	i	aux[i]
0	d	0	a
1	a	1	
2	c	2	
3	f	3	
4	f	4	
5	b	5	
6	d	6	d
7	b	7	
8	f	8	
9	b	9	
10	e	10	
11	a	11	
			r count[r]
		a	1
		b	2
		c	5
		d	7
		e	8
		f	9
		-	12

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

i	a[i]	i	aux[i]
0	d	0	a
1	a	1	
2	c	2	
3	f	3	
4	f	4	
5	b	5	
6	d	6	
7	b	7	
8	f	8	
9	b	9	
10	e	-	12
11	a	10	
		11	

move
items

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

i	a[i]	i	aux[i]
0	d	0	a
1	a	1	
2	c	2	
3	f	3	
4	f	4	
5	b	5	
6	d	6	
7	b	7	
8	f	8	
9	b	9	f
-	-	10	
10	e	10	
11	a	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];  
  
move  
items →  
for (int i = 0; i < N; i++)  
    aux[count[a[i]]++] = a[i];  
  
for (int i = 0; i < N; i++)
```

i	a[i]	i	aux[i]
0	d	0	a
1	a	1	
2	c	2	
3	f	3	
4	f	4	
5	b	5	
6	d	6	
7	b	7	
8	f	8	
9	b	9	
10	e	10	
11	a	11	
		r count[r]	
		a	1
		b	2
		c	6
		d	7
		e	8
		f	11
		-	12

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

i	a[i]	i	aux[i]
0	d	0	a
1	a	1	b
2	c	2	
3	f	3	
4	f	4	
5	b	5	c
6	d	6	d
7	b	7	
8	f	8	
9	b	9	f
-	-	10	
10	e	10	f
11	a	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];  
  
move  
items →  
for (int i = 0; i < N; i++)  
    aux[count[a[i]]++] = a[i];  
  
for (int i = 0; i < N; i++)
```

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

move
items

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

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

r count[*r*]

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

i	a[i]	i	aux[i]
0	d	0	a
1	a	1	b
2	c	2	b
3	f	3	b
4	f	4	
5	b	5	c
6	d	6	d
7	b	7	d
8	f	8	
9	b	9	f
-	-	10	
10	e	10	f
11	a	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];  
  
move  
items →  
for (int i = 0; i < N; i++)  
    aux[count[a[i]]++] = a[i];  
  
for (int i = 0; i < N; i++)
```

i	a[i]	i	aux[i]
0	d	0	a
1	a	1	b
2	c	2	b
3	f	3	b
4	f	4	b
5	b	5	c
6	d	6	d
7	b	7	d
8	f	8	d
9	b	9	f
10	e	10	f
11	a	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];  
  
move  
items →  
for (int i = 0; i < N; i++)  
    aux[count[a[i]]++] = a[i];  
  
for (int i = 0; i < N; i++)
```

i	a[i]	i	aux[i]
0	d	0	a
1	a	1	b
2	c	2	b
3	f	3	b
4	f	4	b
5	b	5	c
6	d	6	d
7	b	7	d
8	f	8	e
9	b	9	f
10	e	10	f
11	a	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];  
  
move  
items →  
for (int i = 0; i < N; i++)  
    aux[count[a[i]]++] = a[i];  
  
for (int i = 0; i < N; i++)
```

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

r count[*r*]

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

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

r count[*r*]

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];
```

copy
back →

```
for (int i = 0; i < N; i++)
```

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

r count[*r*]

a	2	0	a
b	5	1	a
c	6	2	b
d	8	3	b
e	9	4	b
f	12	5	c
-	12	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++)
```

count
frequencies

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

a	0
b	2
c	3
d	1
e	2
f	1
-	3

	r count[r]
--	------------

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]++;  
  
compute  
cumulates  
→  
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++)
```

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 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

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]

30

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

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

r count[*r*]

move
items →

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];
```

copy
back →

```
for (int i = 0; i < N; i++)
```

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

Key-indexed counting: analysis

Proposition. Key-indexed takes time proportional to $N + R$.

Key-indexed counting: analysis

Proposition. Key-indexed takes time proportional to $N + R$.

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

Key-indexed counting: analysis

Proposition. Key-indexed takes time proportional to $N + R$.

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

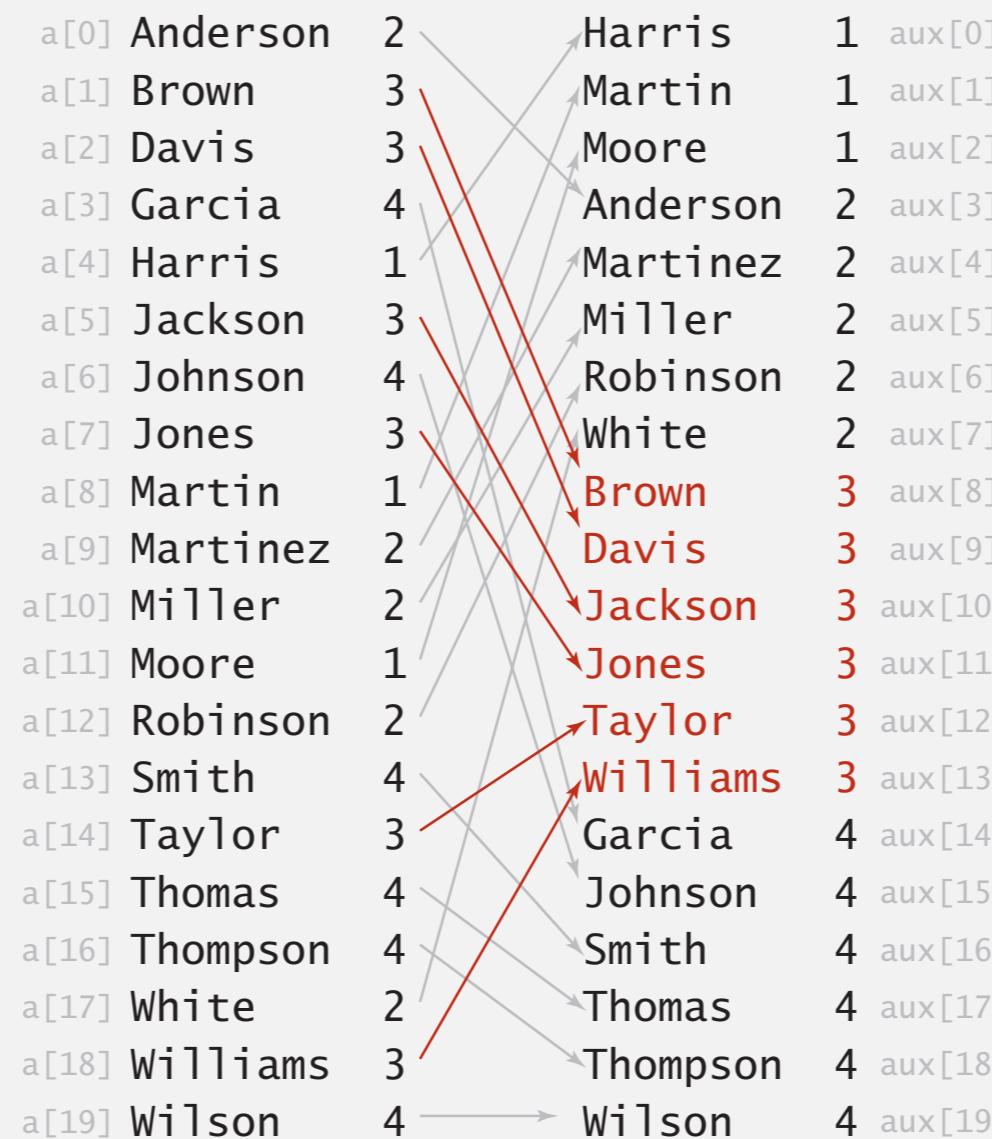
Stable?

Key-indexed counting: analysis

Proposition. Key-indexed takes time proportional to $N + R$.

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

Stable? ✓



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

5.1 STRING SORTS

- ▶ strings in Java
- ▶ 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.

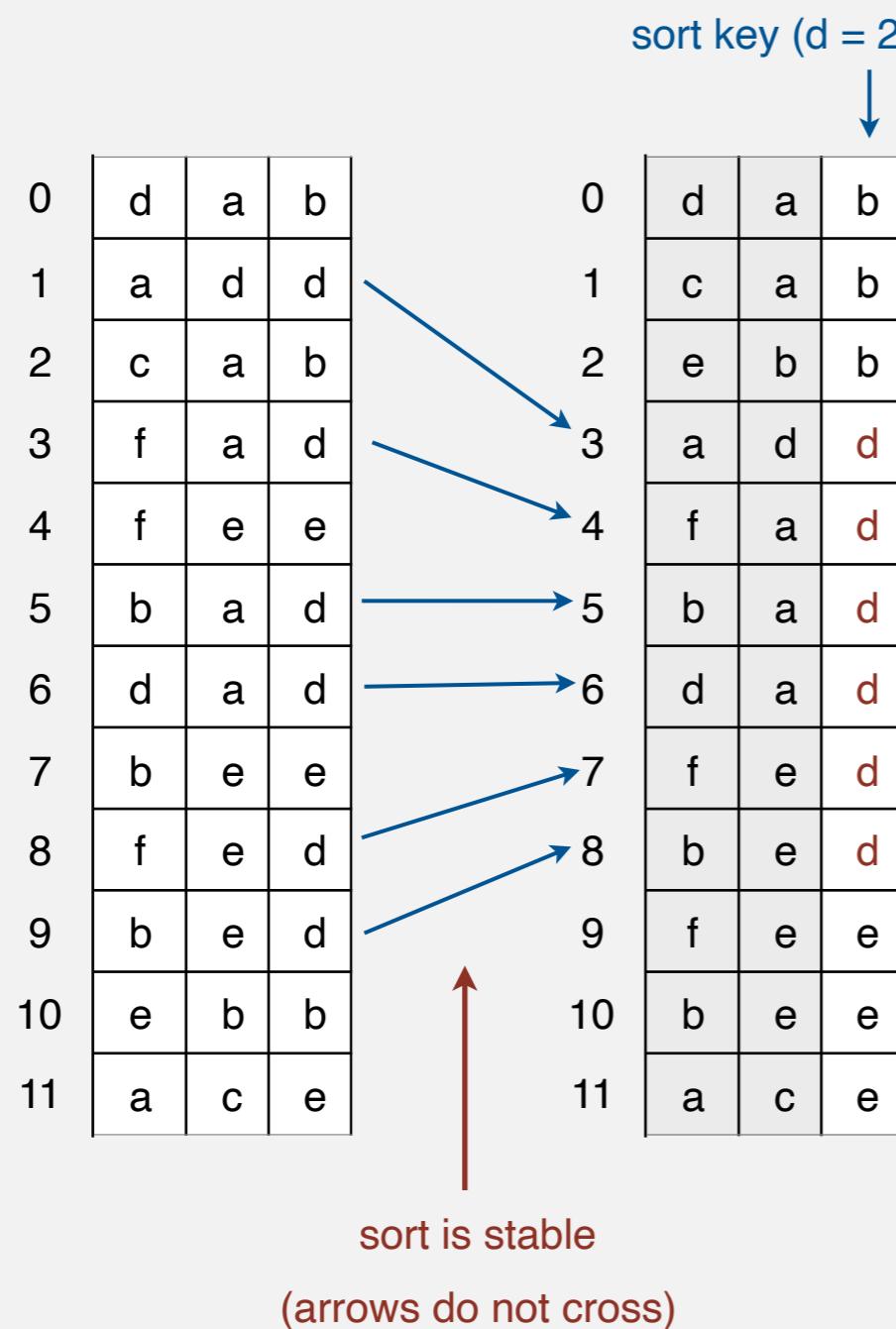
- Consider characters from right to left.
- Stably sort using d^{th} character as the key (using key-indexed counting).

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

Least-significant-digit-first string sort

LSD string (radix) sort.

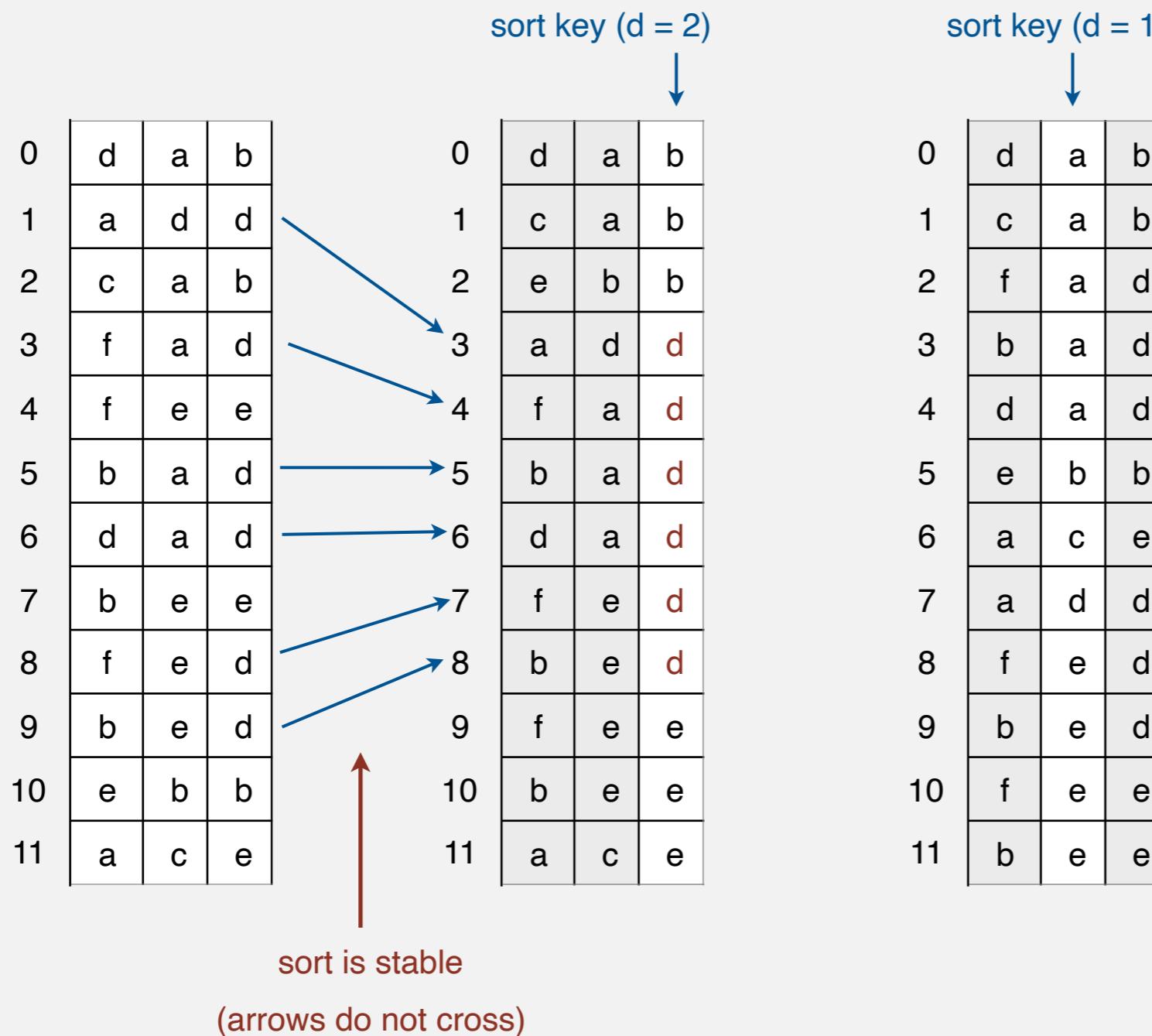
- Consider characters from right to left.
- Stably sort using d^{th} character as the key (using key-indexed counting).



Least-significant-digit-first string sort

LSD string (radix) sort.

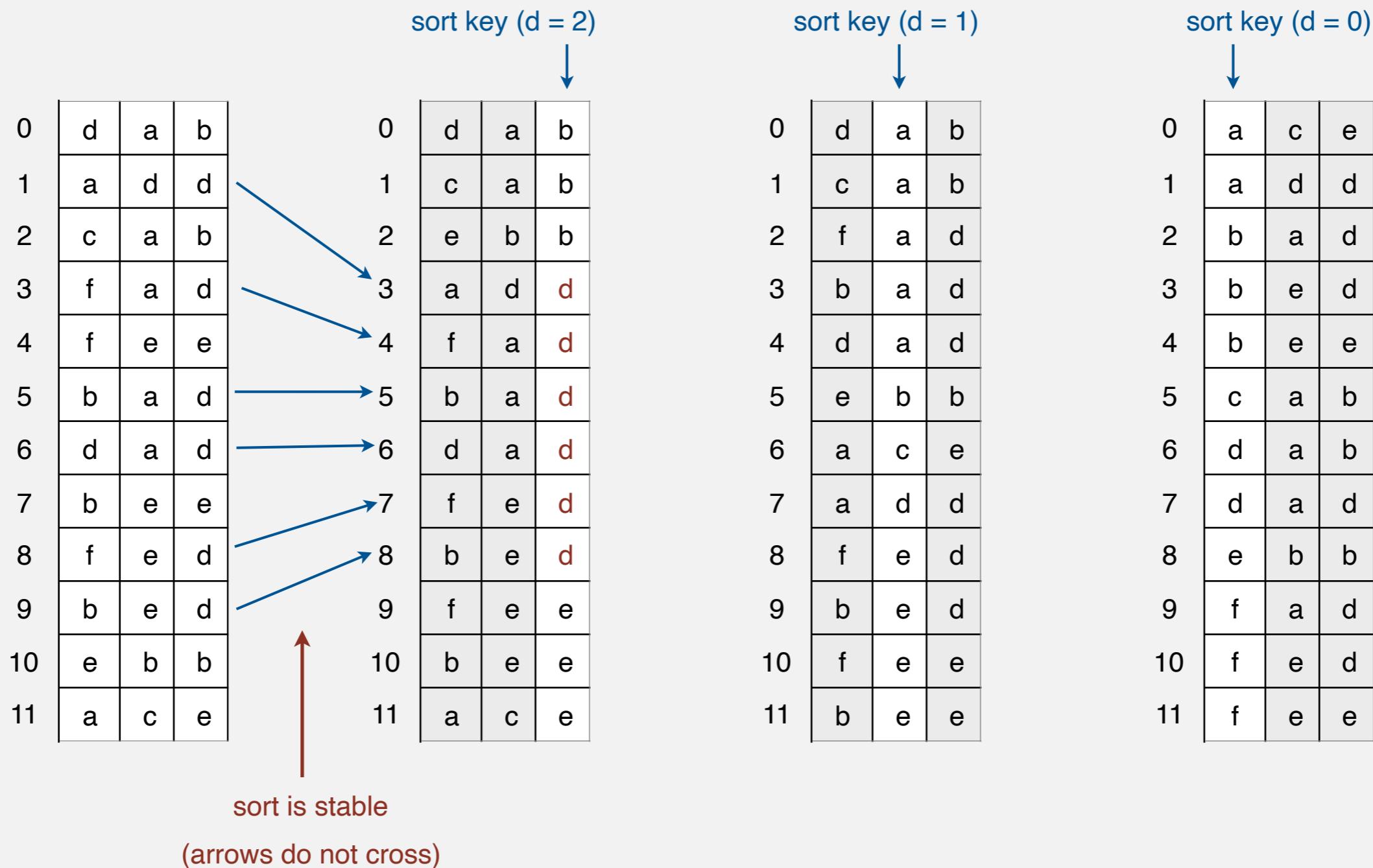
- Consider characters from right to left.
- Stably sort using d^{th} character as the key (using key-indexed counting).



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).



LSD string sort: correctness proof

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



LSD string sort: correctness proof

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

Pf. [by induction on i]

sort key
↓

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

sorted from
previous passes
(by induction)

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.

sort key ↓

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

sorted from
previous passes
(by induction)

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.

sort key ↓

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

sorted from
previous passes
(by induction)

36

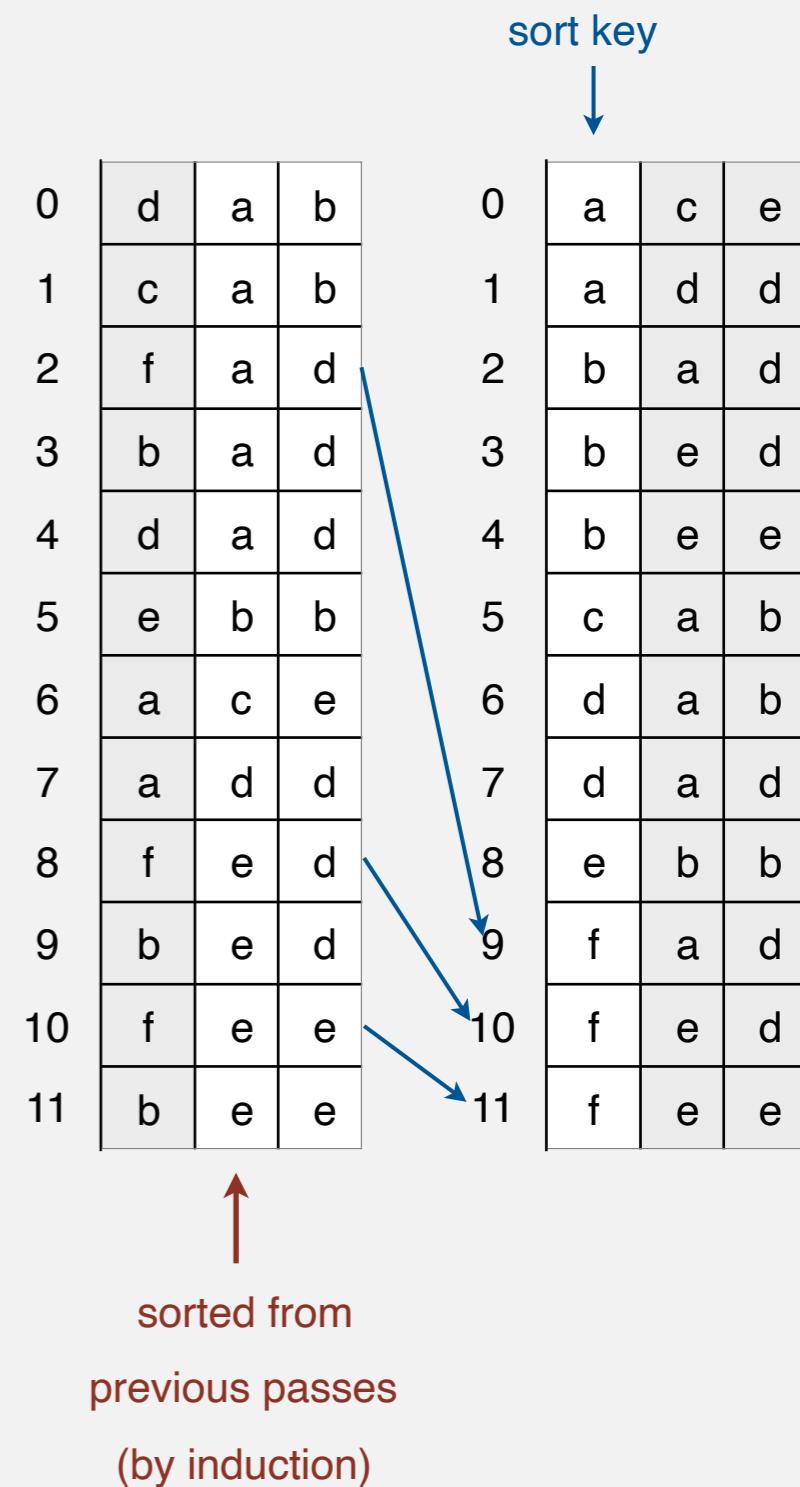
LSD string sort: correctness proof

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.



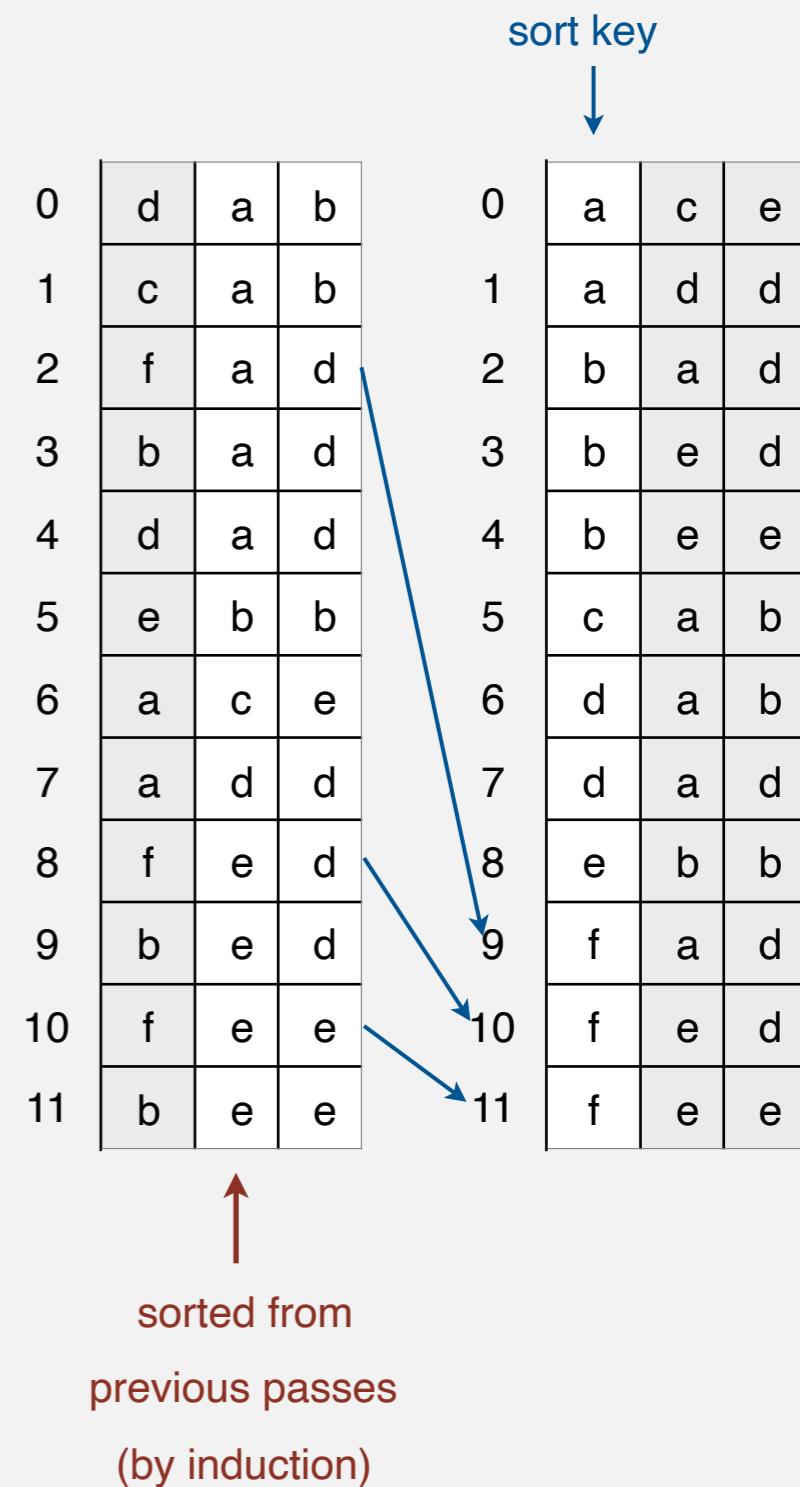
LSD string sort: correctness proof

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.



Proposition. LSD sort is stable.

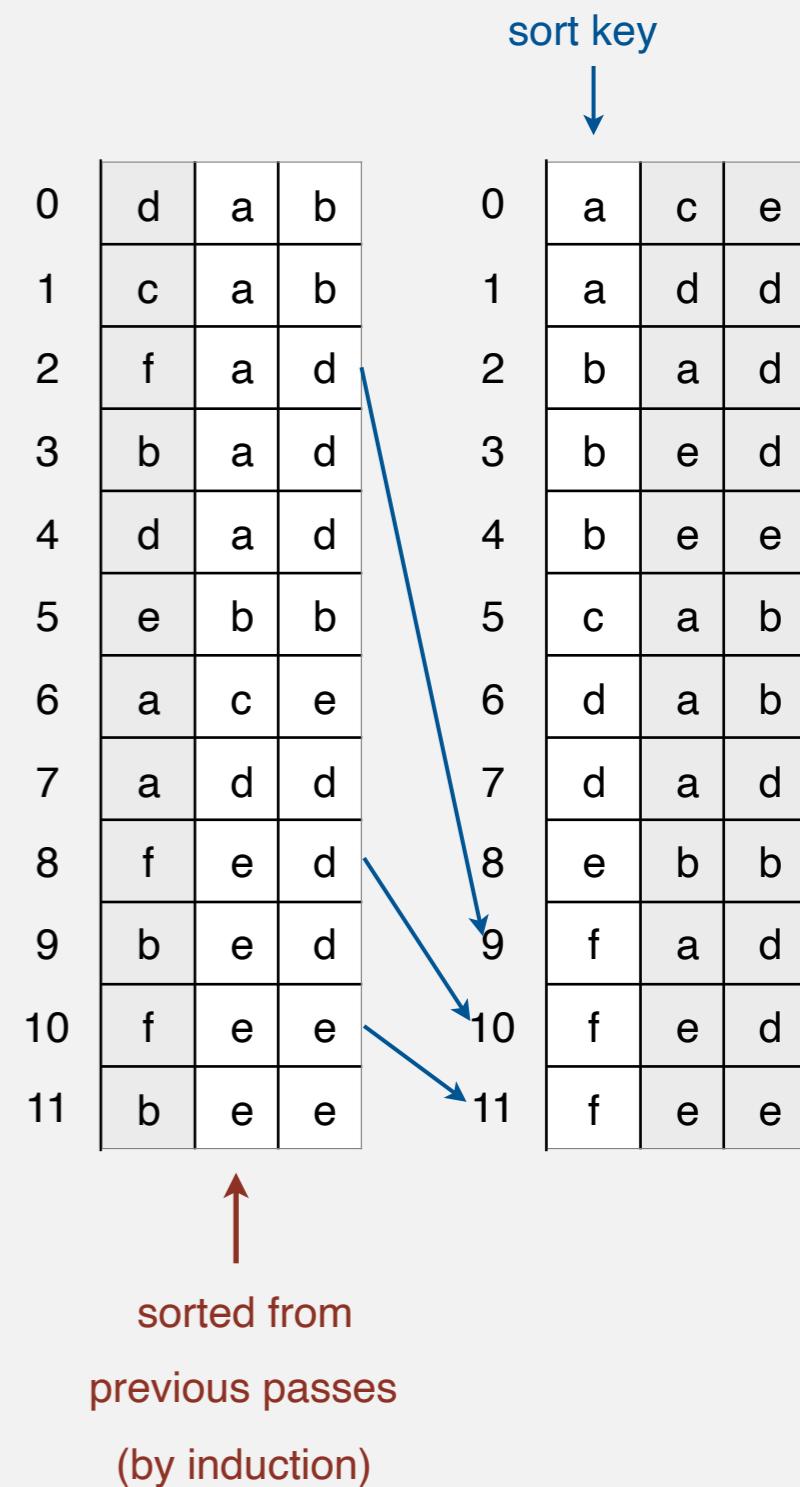
LSD string sort: correctness proof

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.



Proposition. LSD sort is stable.

Pf. Key-indexed counting is stable.

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	$\frac{1}{2} N^2$	$\frac{1}{4} N^2$	1	✓	compareTo()
mergesort	$N \lg N$	$N \lg N$	N	✓	compareTo()
quicksort	$1.39 N \lg N$ *	$1.39 N \lg N$	$c \lg N$		compareTo()
heapsort	$2 N \lg N$	$2 N \lg N$	1		compareTo()
LSD sort †	$2 W(N + R)$	$2 W(N + R)$	$N + R$	✓	charAt()

* probabilistic

† fixed-length W keys

Summary of the performance of sorting algorithms

Frequency of operations.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$\frac{1}{2} N^2$	$\frac{1}{4} N^2$	1	✓	compareTo()
mergesort	$N \lg N$	$N \lg N$	N	✓	compareTo()
quicksort	$1.39 N \lg N$ *	$1.39 N \lg N$	$c \lg N$		compareTo()
heapsort	$2 N \lg N$	$2 N \lg N$	1		compareTo()
LSD sort †	$2 W(N + R)$	$2 W(N + R)$	$N + R$	✓	charAt()

* probabilistic

† fixed-length W keys

Q. What if strings are not all of same length?

String sorting interview question

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.





Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

5.1 STRING SORTS

- ▶ strings in Java
- ▶ key-indexed counting
- ▶ LSD radix sort
- ▶ MSD radix sort
- ▶ 3-way radix quicksort
- ▶ suffix arrays

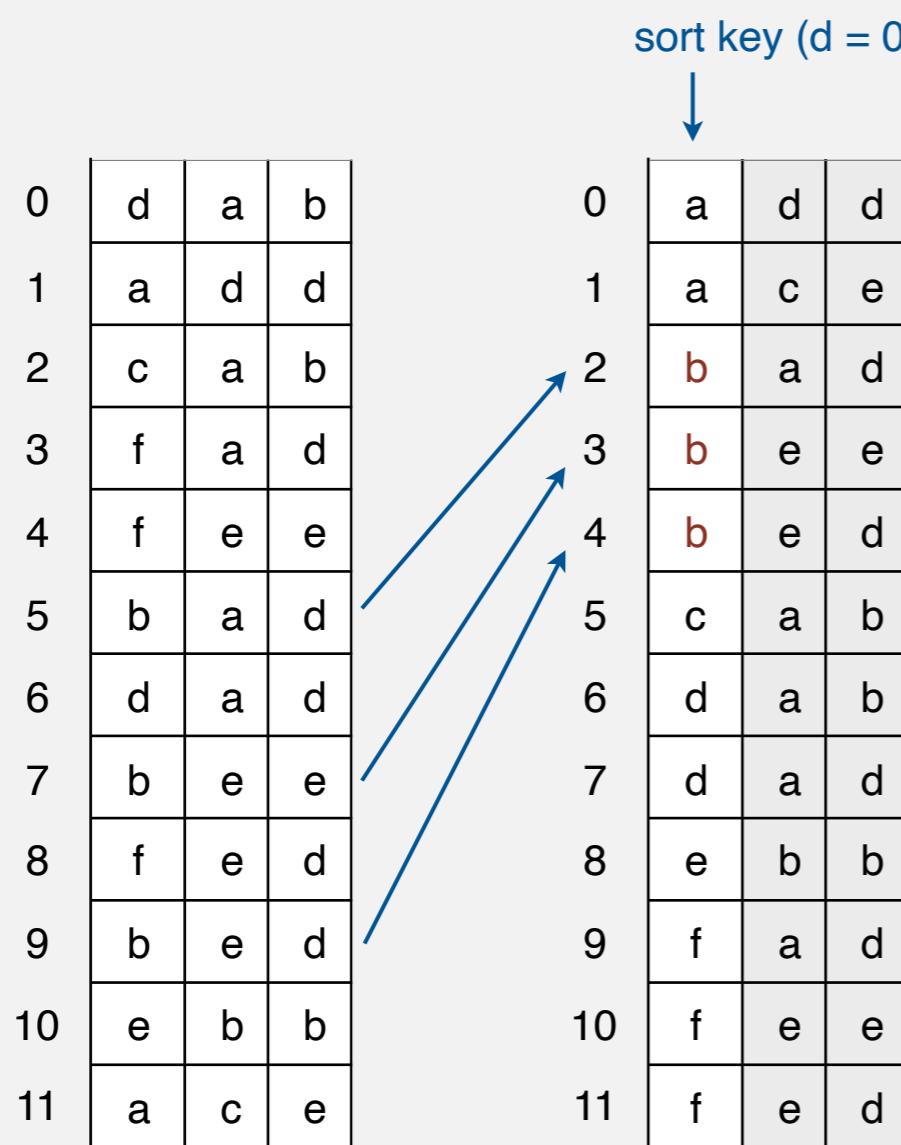
Reverse LSD

- Consider characters from left to right.
- 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

Reverse LSD

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



Reverse LSD

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

sort key (d = 0)

↓

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

sort key (d = 1)

↓

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

Diagram illustrating the Reverse LSD sorting process. The initial array (left) is sorted by the 0th character (d). The result is shown in the middle. The middle array is then sorted by the 1st character (a), resulting in the final sorted array (right).

Reverse LSD

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

sort key (d = 0)

sort key (d = 1)

sort key (d = 2)

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

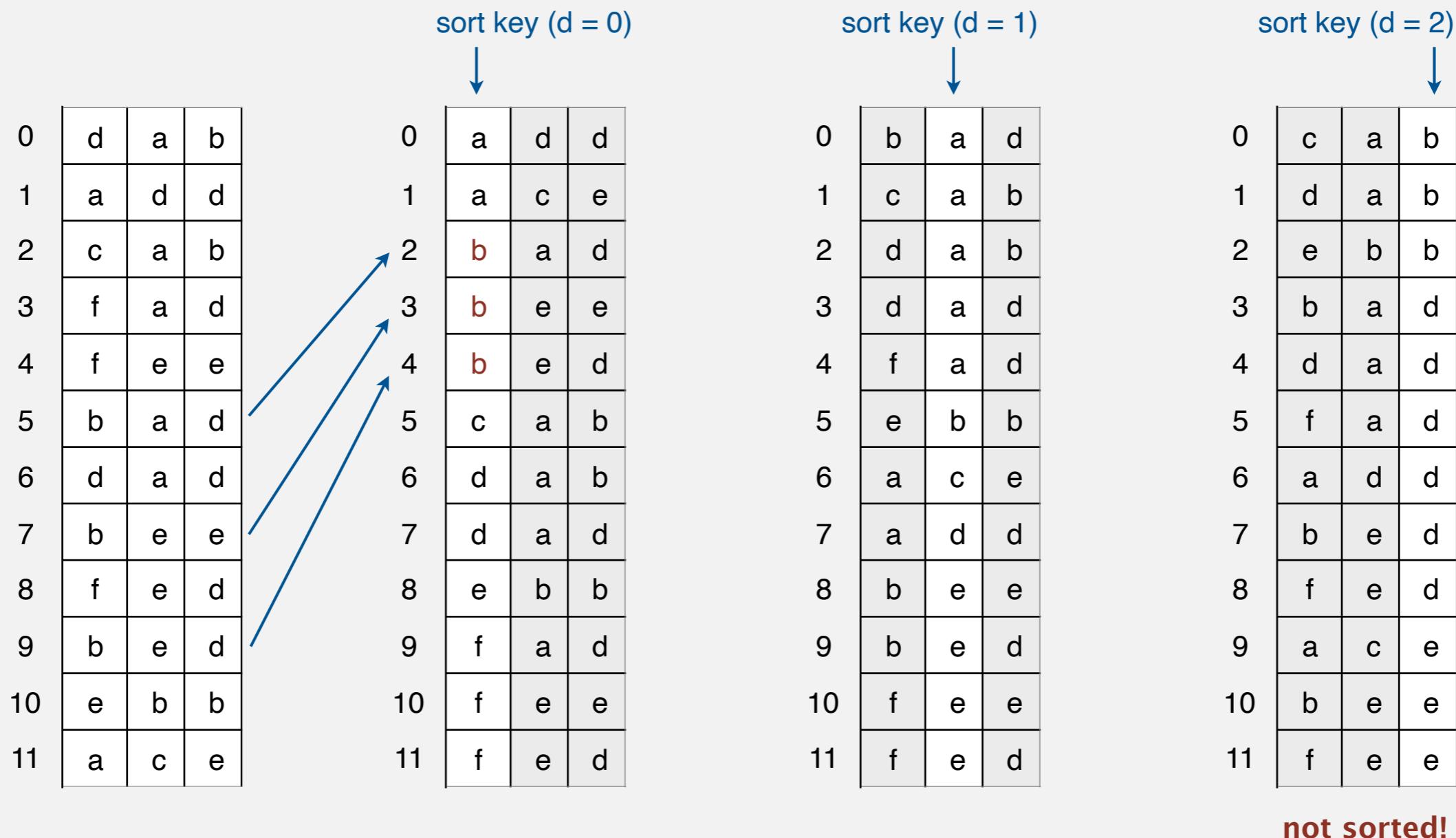
0	a	d	d
1	a	c	e
2	b	a	d
3	b	e	e
4	b	e	d
5	c	a	b
6	d	a	b
7	d	a	d
8	e	b	b
9	f	a	d
10	f	e	e
11	f	e	d

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

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

Reverse LSD

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



Most-significant-digit-first string sort

MSD string (radix) sort.

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

Most-significant-digit-first string sort

MSD string (radix) sort.

- Partition array into R pieces according to first character (use 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	a	d	d
1	a	c	e
2	b	a	d
3	b	e	e
4	b	e	d
5	c	a	b
6	d	a	b
7	d	a	d
8	e	b	b
9	f	a	d
10	f	e	e
11	f	e	d

↑
sort key

Most-significant-digit-first string sort

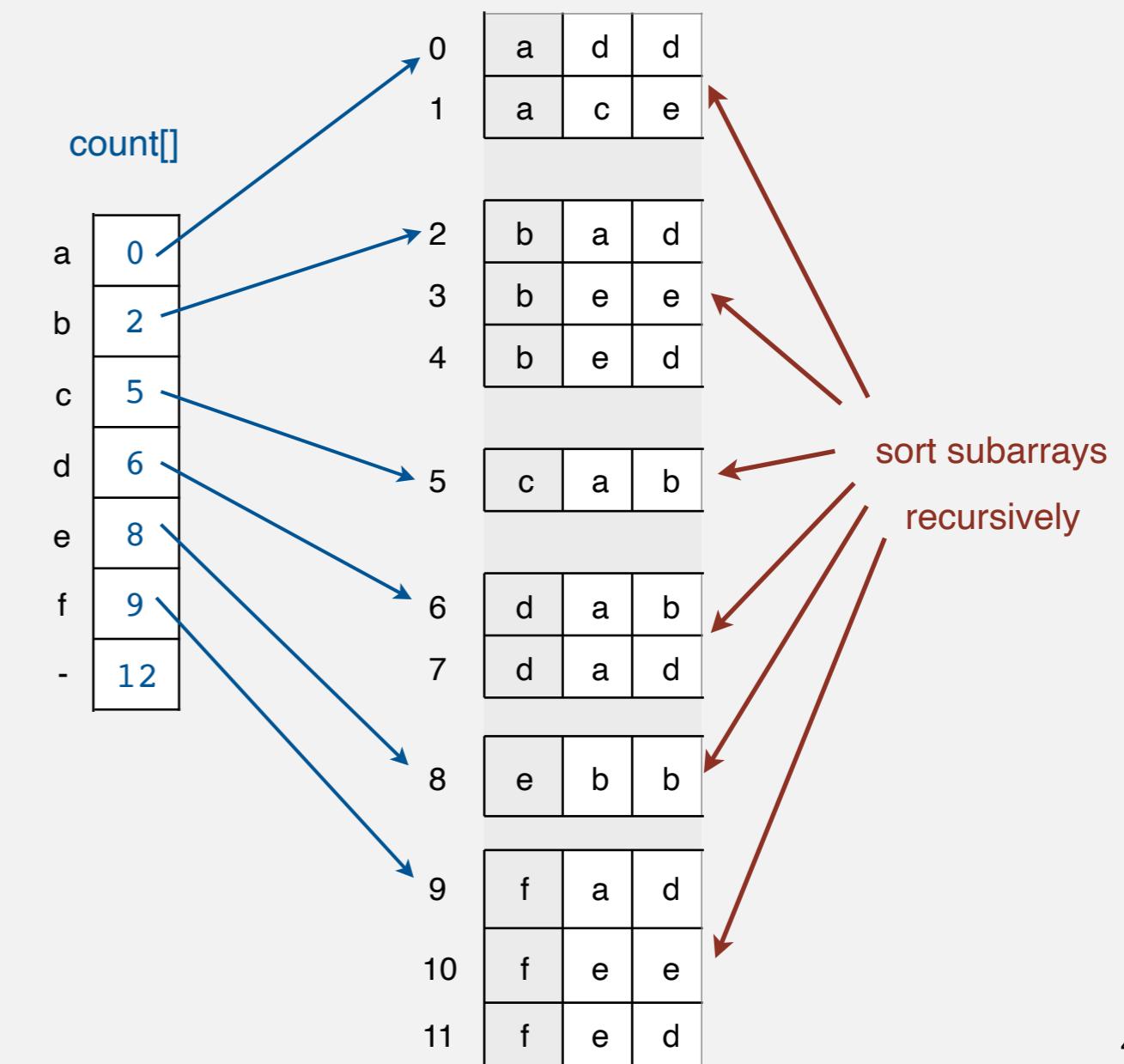
MSD string (radix) sort.

- 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).

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

sort key



MSD string sort: example

input	are								
she	are								
sells	by	to	by						
seashells	she	se	se	sea	sea	sea	sea	sea	sea
by	sells	seashells	sea	seashells	seashells	seashells	seashells	seashells	seashells
the	seashells	sea	seashells						
sea	sea	sells	se	sells	sells	sells	sells	sells	sells
shore	shore	seashells	se						
the	shells	she							
shells	she	shore							
she	sells	shells							
sells	surely	she							
are	seashells	surely							
surely	the	hi	the						
seashells	the								

<i>need to examine every character in equal keys</i>	are	output	
	are		
	by		
	sea		
	seashells		
	seashells		
	sells		
	sells		
	she		
	shore	sshore	shore
	shells	hells	shells
	she		
	surely		
	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;  
}
```

Variable-length strings

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

```
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 - 1, 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];
```

key-indexed counting

```
    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];
```

sort R subarrays recursively

```
    for (int r = 0; r < R; r++)
```

```
        sort(a, aux, lo + count[r], lo + count[r+1] - 1, d+1);
```

MSD string sort: Java implementation

```
public static void sort(String[] a){  
    aux = new String[a.length];  
    sort(a, aux, 0, a.length - 1, 0);  
}
```

recycles aux[] array
but not count[] array

```
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];
```

key-indexed counting

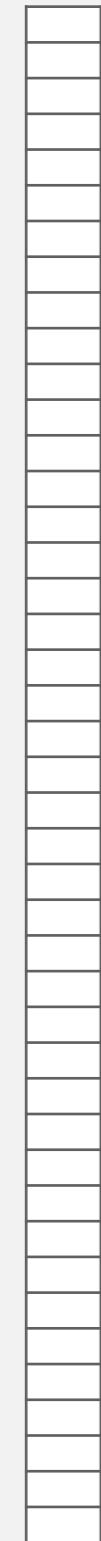
sort R subarrays recursively

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

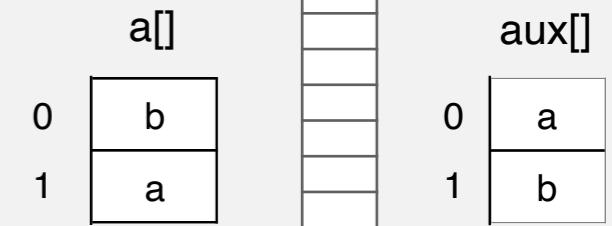
MSD string sort: potential for disastrous performance

Observation 1. Much too slow for small subarrays.

count[]



- 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 $N = 2$.



MSD string sort: potential for disastrous performance

Observation 1. Much too slow for small subarrays.

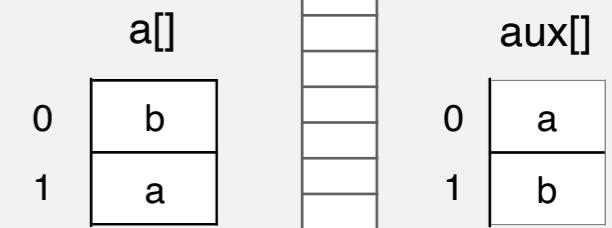
count[]



- 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.

Cutoff to insertion sort

Solution. Cutoff to insertion sort for small subarrays.

- Insertion sort, but start at d^{th} character.

```
private 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);
}
```

Cutoff to insertion sort

Solution. Cutoff to insertion sort for small subarrays.

- Insertion sort, but start at d^{th} character.

```
private 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);
}
```

- Implement `less()` so that it compares starting at d^{th} character.

```
private static boolean less(String v, String w, int d)
{
    for (int i = d; i < Math.min(v.length(), w.length()); i++)
    {
        if (v.charAt(i) < w.charAt(i)) return true;
        if (v.charAt(i) > w.charAt(i)) return false;
    }
    return v.length() < w.length();
}
```

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)
1EI0402	are	1DNB377
1HYL490	by	1DNB377
1R0Z572	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	$\frac{1}{2} N^2$	$\frac{1}{4} N^2$	1	✓	compareTo()
mergesort	$N \lg N$	$N \lg N$	N	✓	compareTo()
quicksort	$1.39 N \lg N$ *	$1.39 N \lg N$	$c \lg N$		compareTo()
heapsort	$2 N \lg N$	$2 N \lg N$	1		compareTo()
LSD sort †	$2 W(N + R)$	$2 W(N + R)$	$N + R$	✓	charAt()
MSD sort ‡	$2 W(N + R)$	$N \log_R N$	$N + D R$	✓	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.

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

MSD string sort vs. quicksort for strings

Disadvantages of MSD string sort.

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

Disadvantage of quicksort.

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

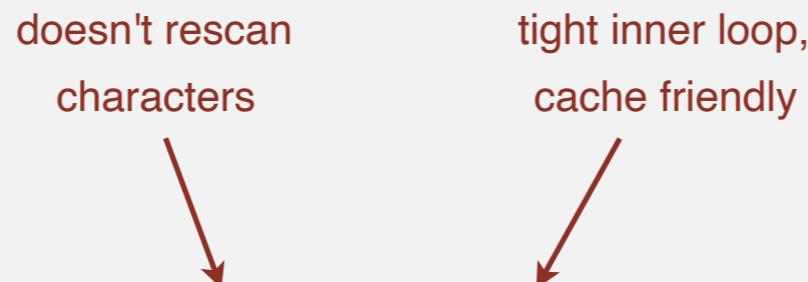
MSD string sort vs. quicksort for strings

Disadvantages of MSD string sort.

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

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.

Engineering a radix sort (American flag sort)

Engineering Radix Sort

Peter M. McIlroy and Keith Bostic
University of California at Berkeley;
and M. Douglas McIlroy
AT&T Bell Laboratories

ABSTRACT: Radix sorting methods have excellent asymptotic performance on string data, for which comparison is not a unit-time operation. Attractive for use in large byte-addressable memories, these methods have nevertheless long been eclipsed by more easily programmed algorithms. Three ways to sort strings by bytes left to right—a stable list sort, a stable two-array sort, and an in-place “American flag” sort—are illustrated with practical C programs. For heavy-duty sorting, all three perform comparably, usually running at least twice as fast as a good quicksort. We recommend American flag sort for general use.

Engineering a radix sort (American flag sort)

Optimization 0. Cutoff to insertion sort.

Engineering Radix Sort

Peter M. McIlroy and Keith Bostic
University of California at Berkeley;
and M. Douglas McIlroy
AT&T Bell Laboratories

ABSTRACT: Radix sorting methods have excellent asymptotic performance on string data, for which comparison is not a unit-time operation. Attractive for use in large byte-addressable memories, these methods have nevertheless long been eclipsed by more easily programmed algorithms. Three ways to sort strings by bytes left to right—a stable list sort, a stable two-array sort, and an in-place “American flag” sort—are illustrated with practical C programs. For heavy-duty sorting, all three perform comparably, usually running at least twice as fast as a good quicksort. We recommend American flag sort for general use.

Engineering a radix sort (American flag sort)

Optimization 0. Cutoff to insertion sort.

Optimization 1. Replace recursion with explicit stack.

- Push subarrays to be sorted onto stack.
- Now, one `count[]` array suffices.

Engineering Radix Sort

Peter M. McIlroy and Keith Bostic
University of California at Berkeley;
and M. Douglas McIlroy
AT&T Bell Laboratories

ABSTRACT: Radix sorting methods have excellent asymptotic performance on string data, for which comparison is not a unit-time operation. Attractive for use in large byte-addressable memories, these methods have nevertheless long been eclipsed by more easily programmed algorithms. Three ways to sort strings by bytes left to right—a stable list sort, a stable two-array sort, and an in-place “American flag” sort—are illustrated with practical C programs. For heavy-duty sorting, all three perform comparably, usually running at least twice as fast as a good quicksort. We recommend American flag sort for general use.

Engineering a radix sort (American flag sort)

Optimization 0. Cutoff to insertion sort.

Optimization 1. Replace recursion with explicit stack.

- Push subarrays to be sorted onto stack.
- Now, one `count[]` array suffices.

Optimization 2. Do R -way partitioning in place.

- Eliminates `aux[]` array.
- Sacrifices stability.



American national flag problem



Dutch national flag problem

Engineering Radix Sort

Peter M. McIlroy and Keith Bostic
University of California at Berkeley;
and M. Douglas McIlroy
AT&T Bell Laboratories

ABSTRACT: Radix sorting methods have excellent asymptotic performance on string data, for which comparison is not a unit-time operation. Attractive for use in large byte-addressable memories, these methods have nevertheless long been eclipsed by more easily programmed algorithms. Three ways to sort strings by bytes left to right—a stable list sort, a stable two-array sort, and an in-place “American flag” sort—are illustrated with practical C programs. For heavy-duty sorting, all three perform comparably, usually running at least twice as fast as a good quicksort. We recommend American flag sort for general use.

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

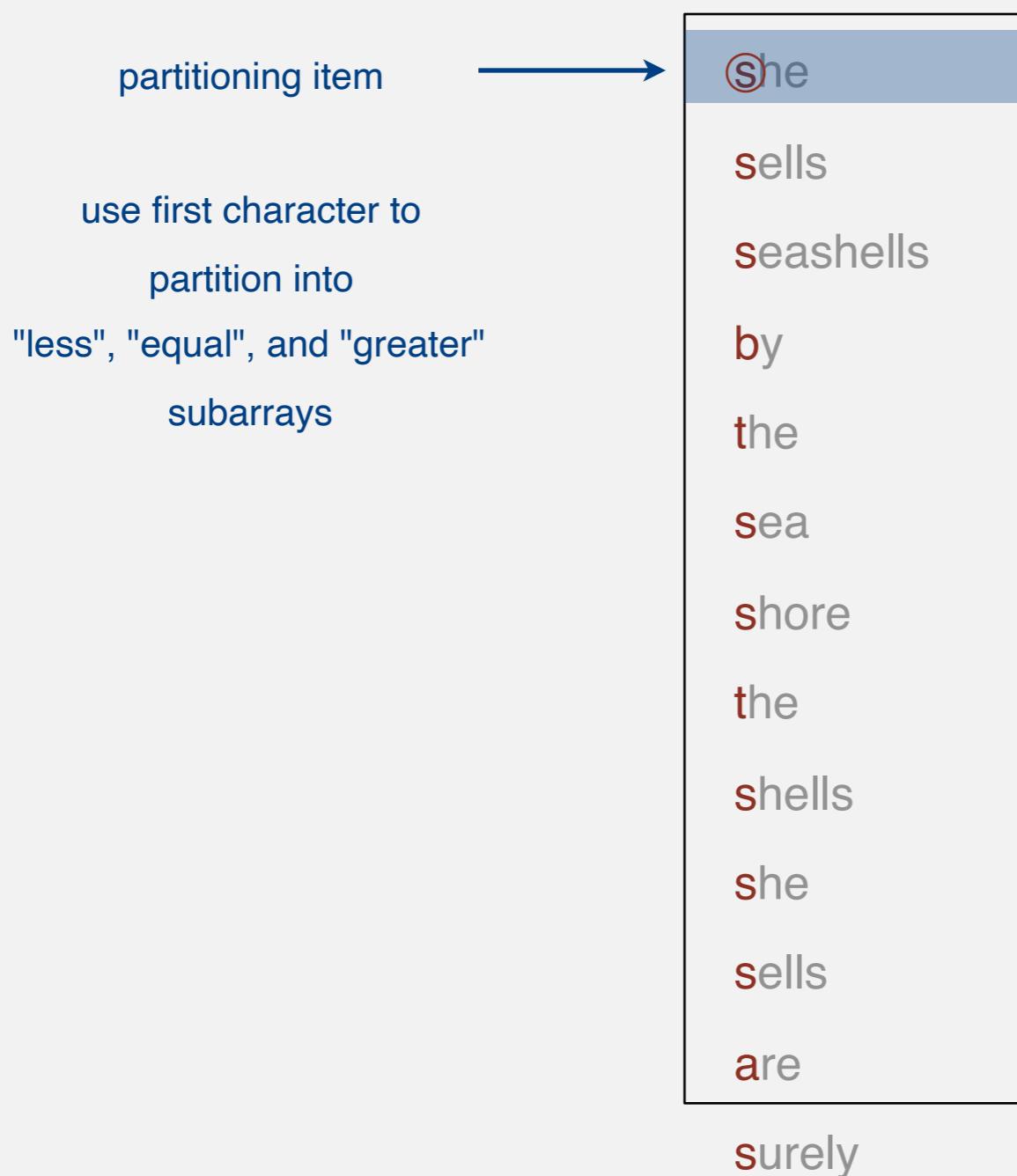
5.1 STRING SORTS

- ▶ strings in Java
- ▶ 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 d^{th} character.

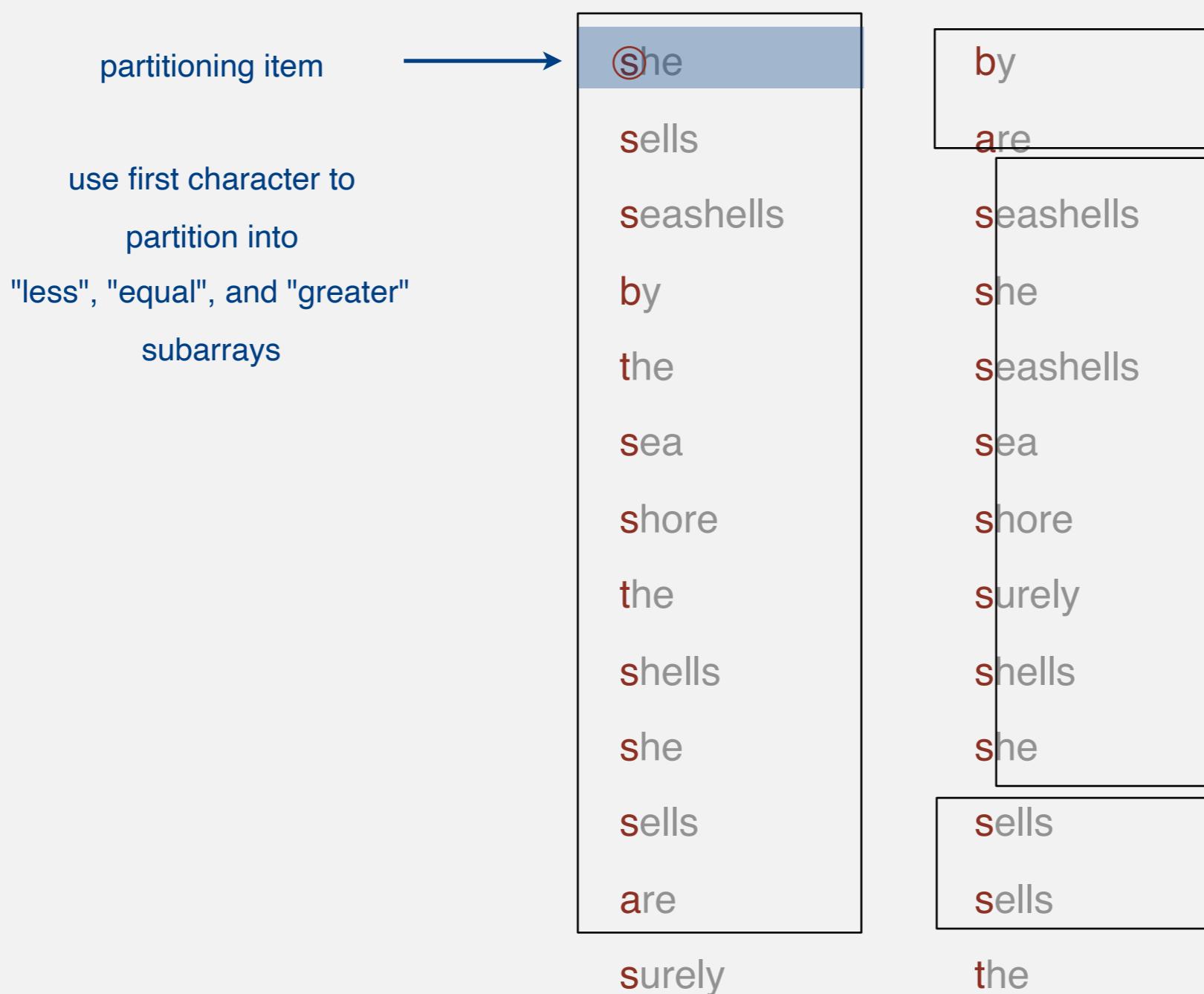
- Less overhead than R -way partitioning in MSD string sort.
- Does not re-examine characters equal to the partitioning char.
(but does re-examine characters not equal to the partitioning char)



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

Overview. Do 3-way partitioning on the d^{th} character.

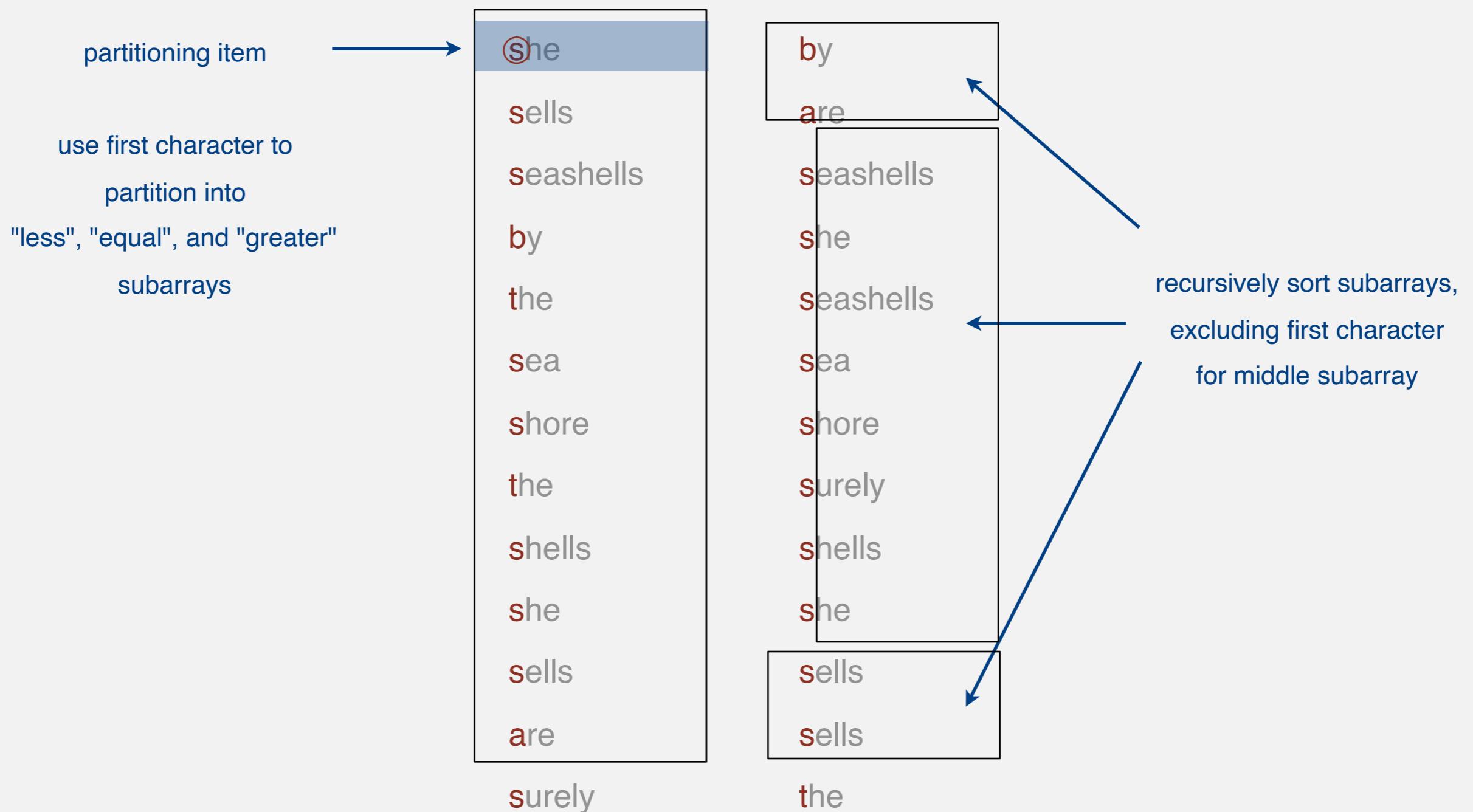
- Less overhead than R -way partitioning in MSD string sort.
- Does not re-examine characters equal to the partitioning char.
(but does re-examine characters not equal to the partitioning char)



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

Overview. Do 3-way partitioning on the d^{th} 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)



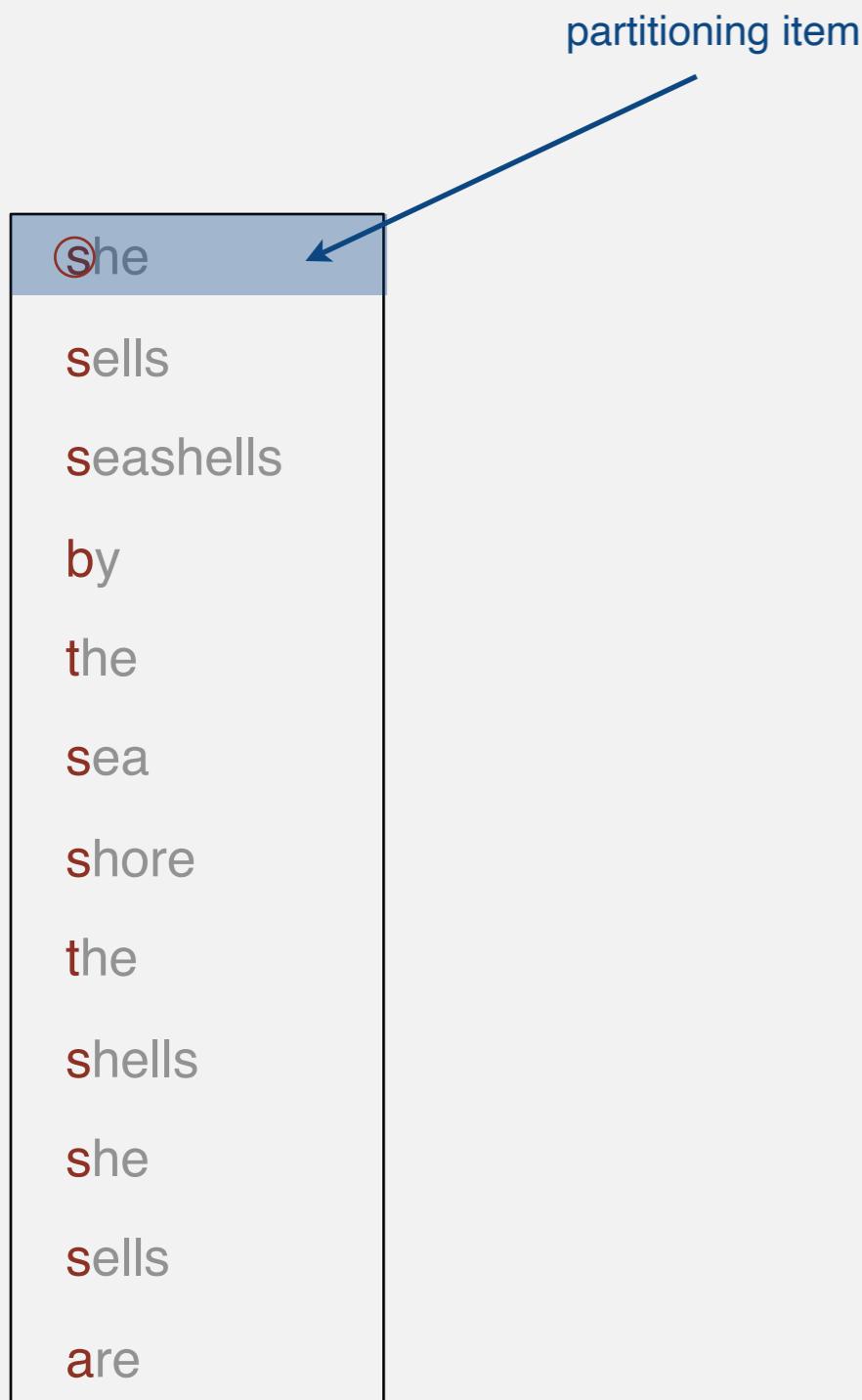
3-way string quicksort: trace of recursive calls

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

surely

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

3-way string quicksort: trace of recursive calls

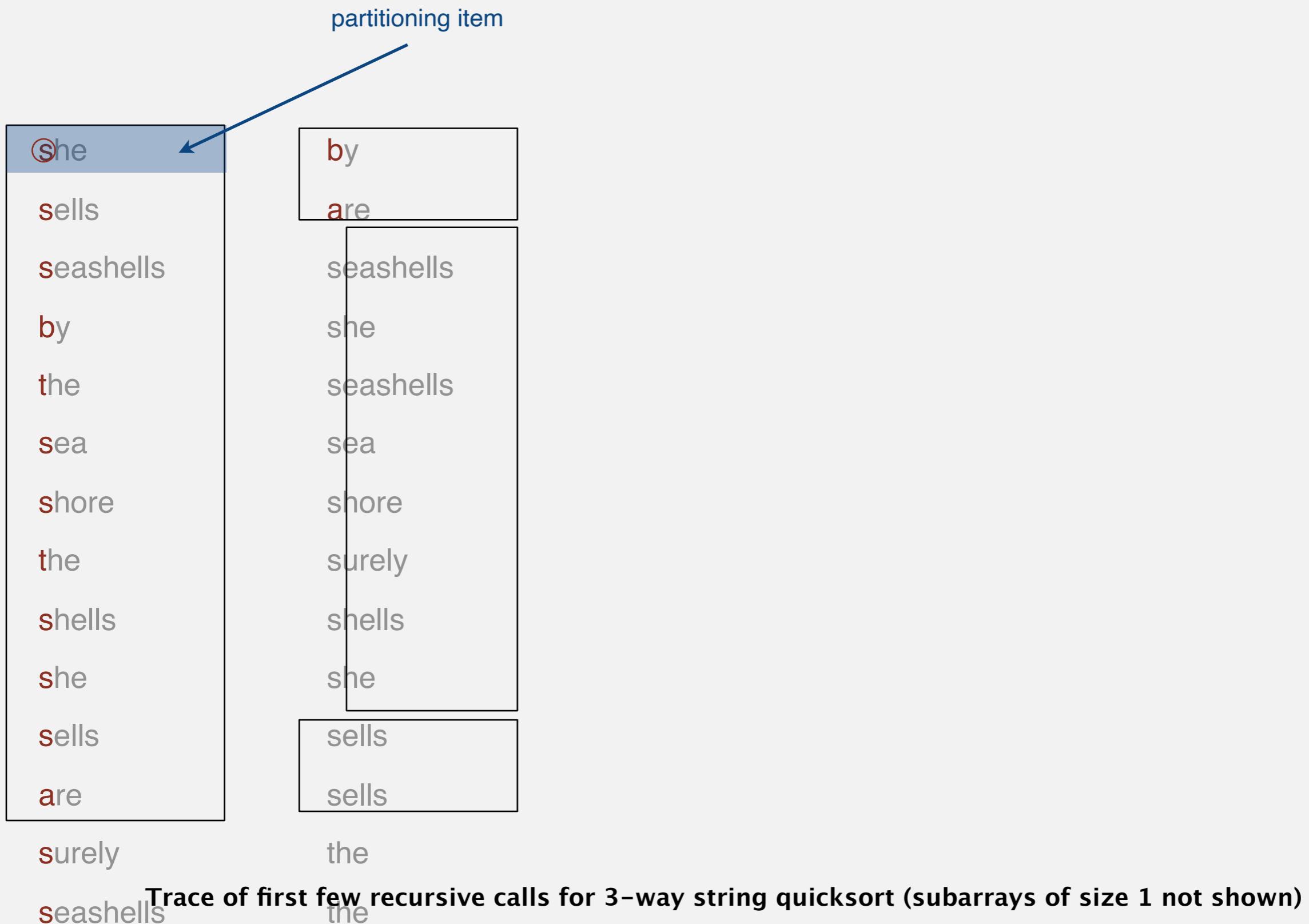


surely

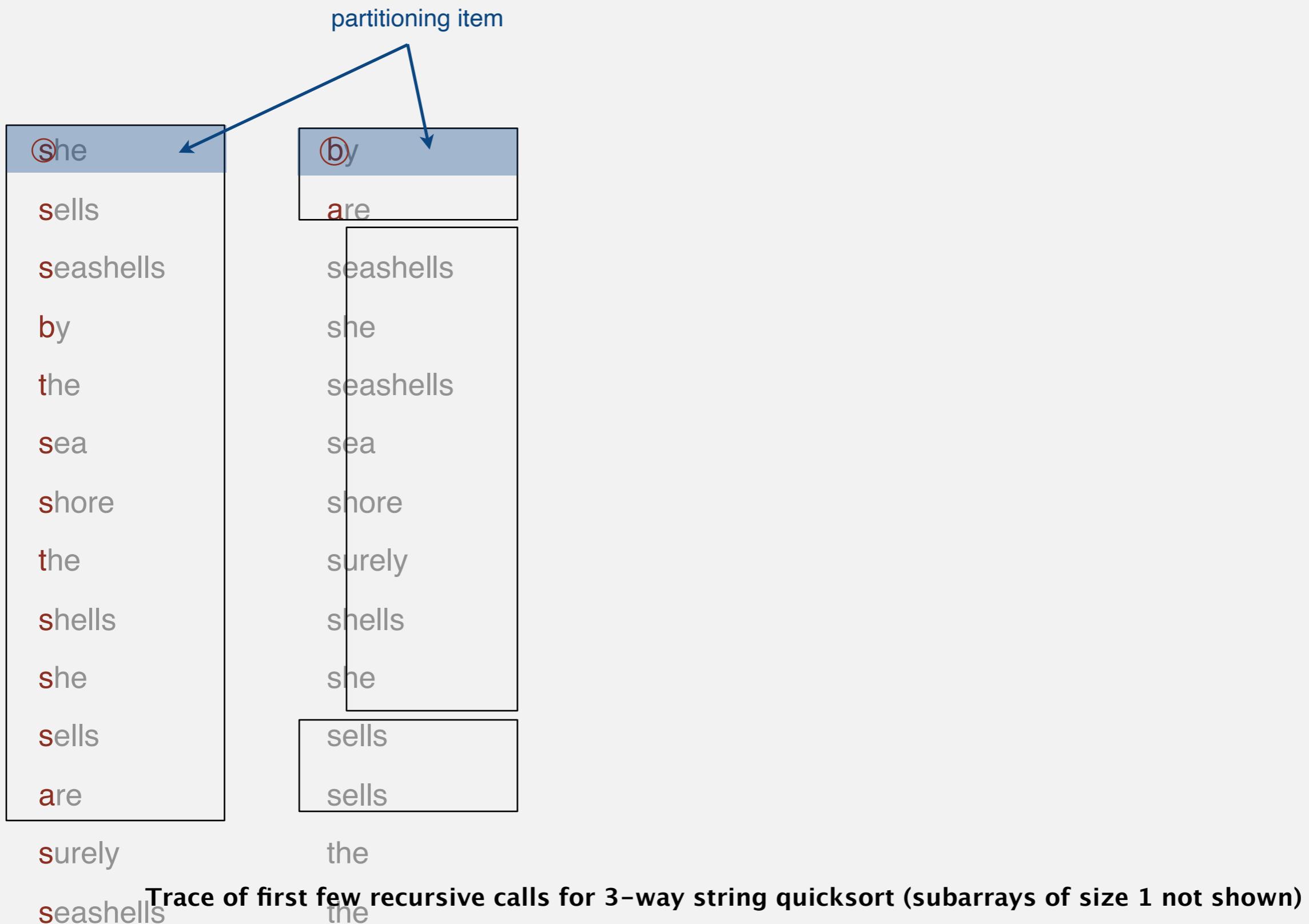
seashells

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

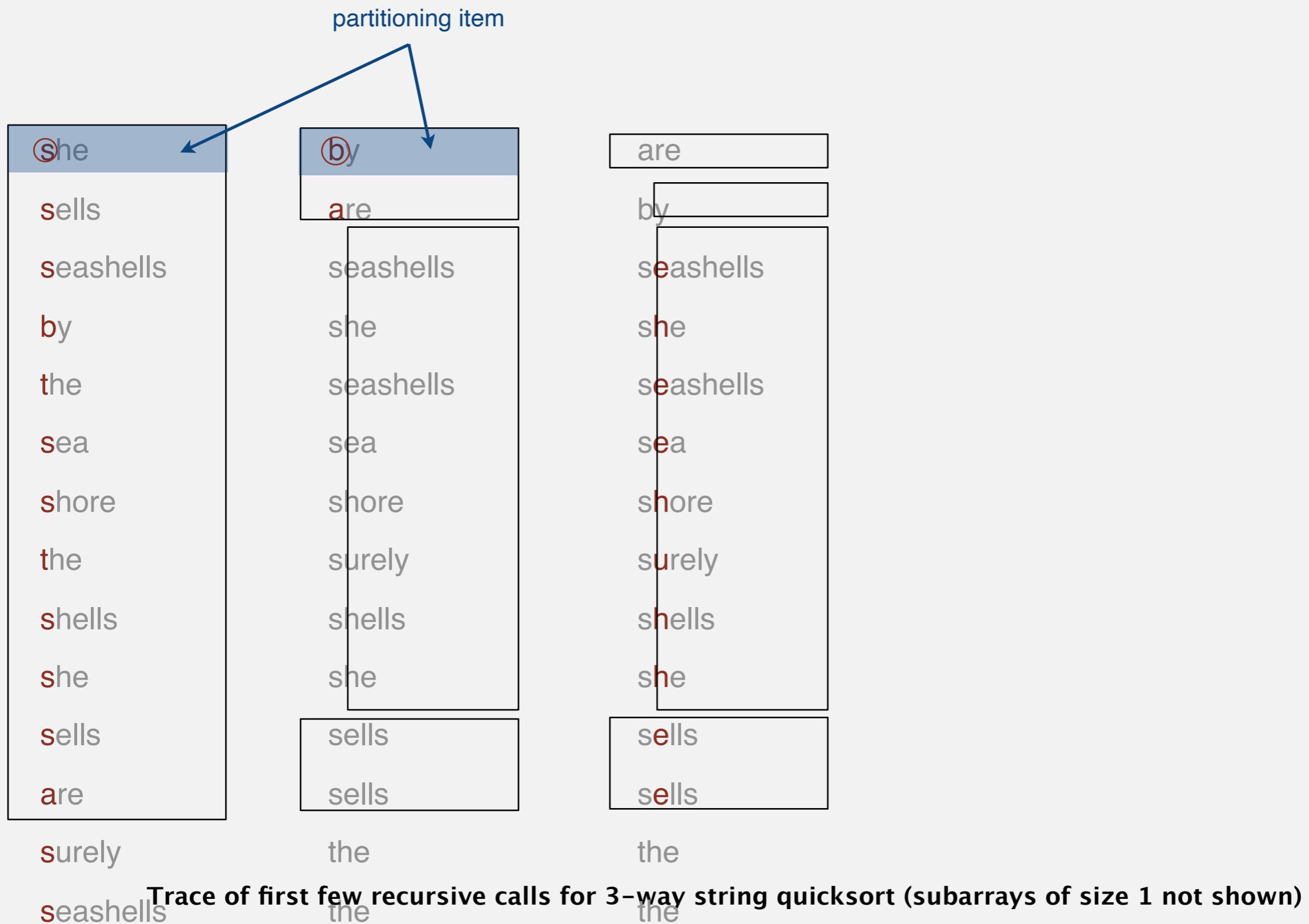
3-way string quicksort: trace of recursive calls



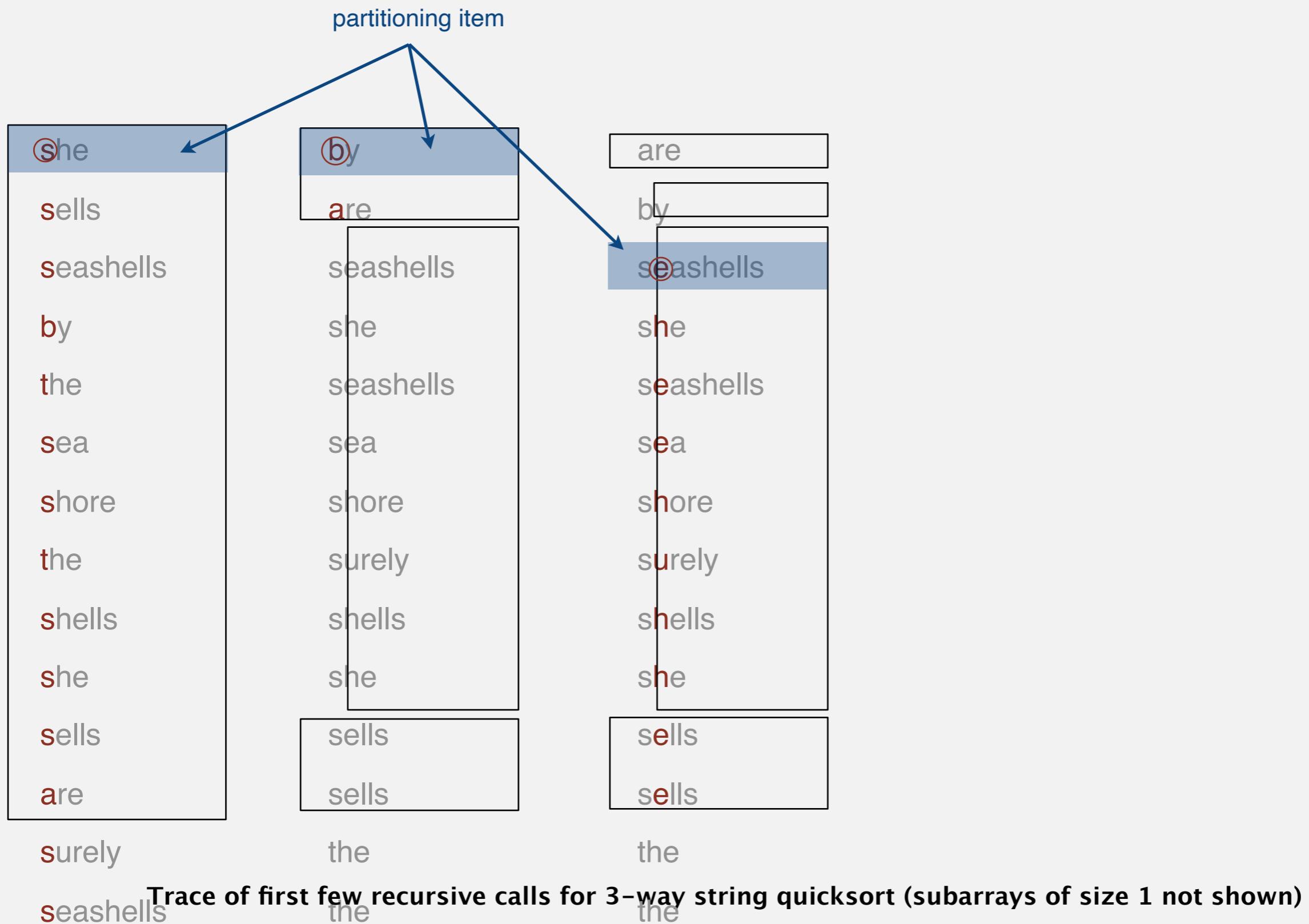
3-way string quicksort: trace of recursive calls



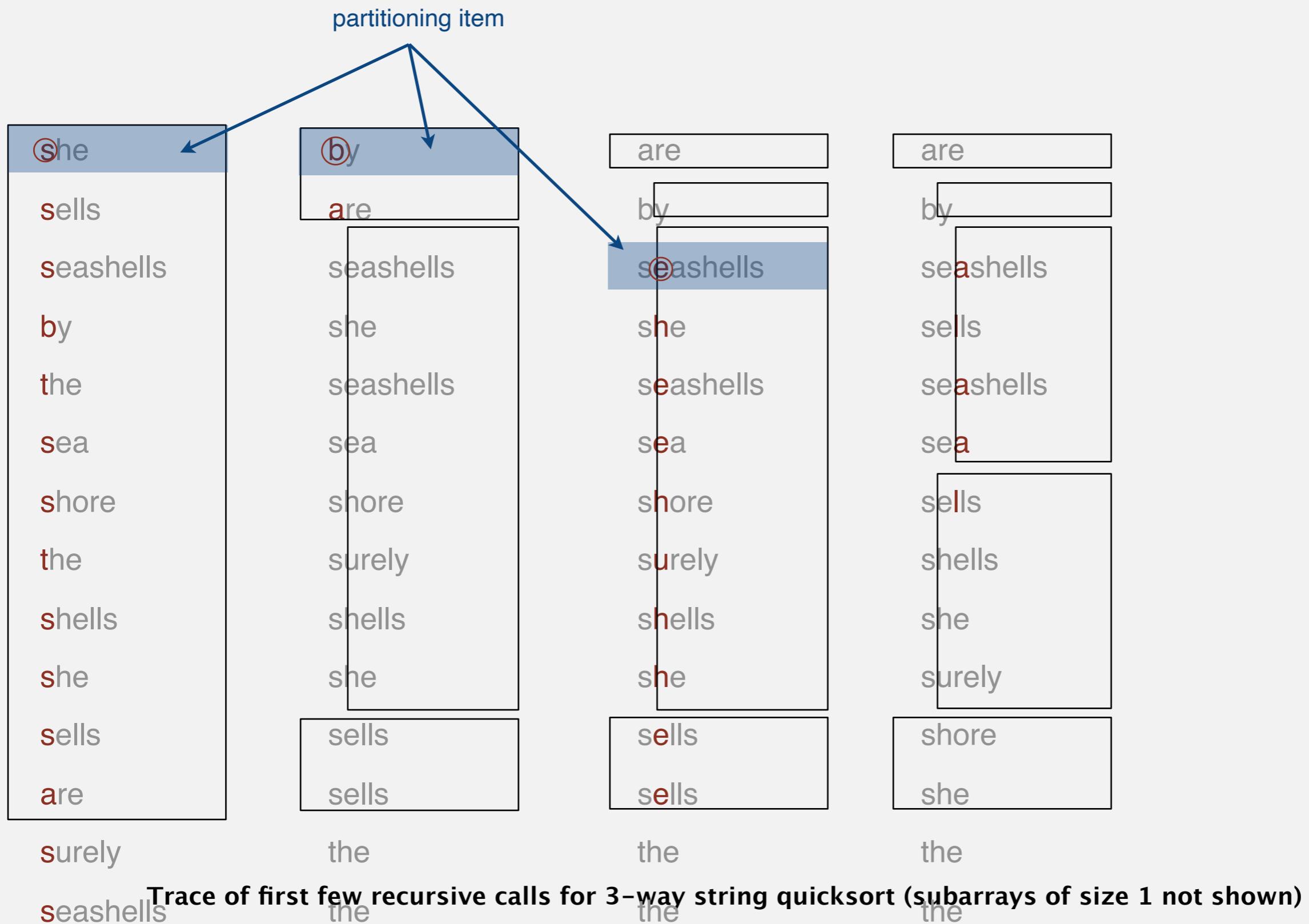
3-way string quicksort: trace of recursive calls



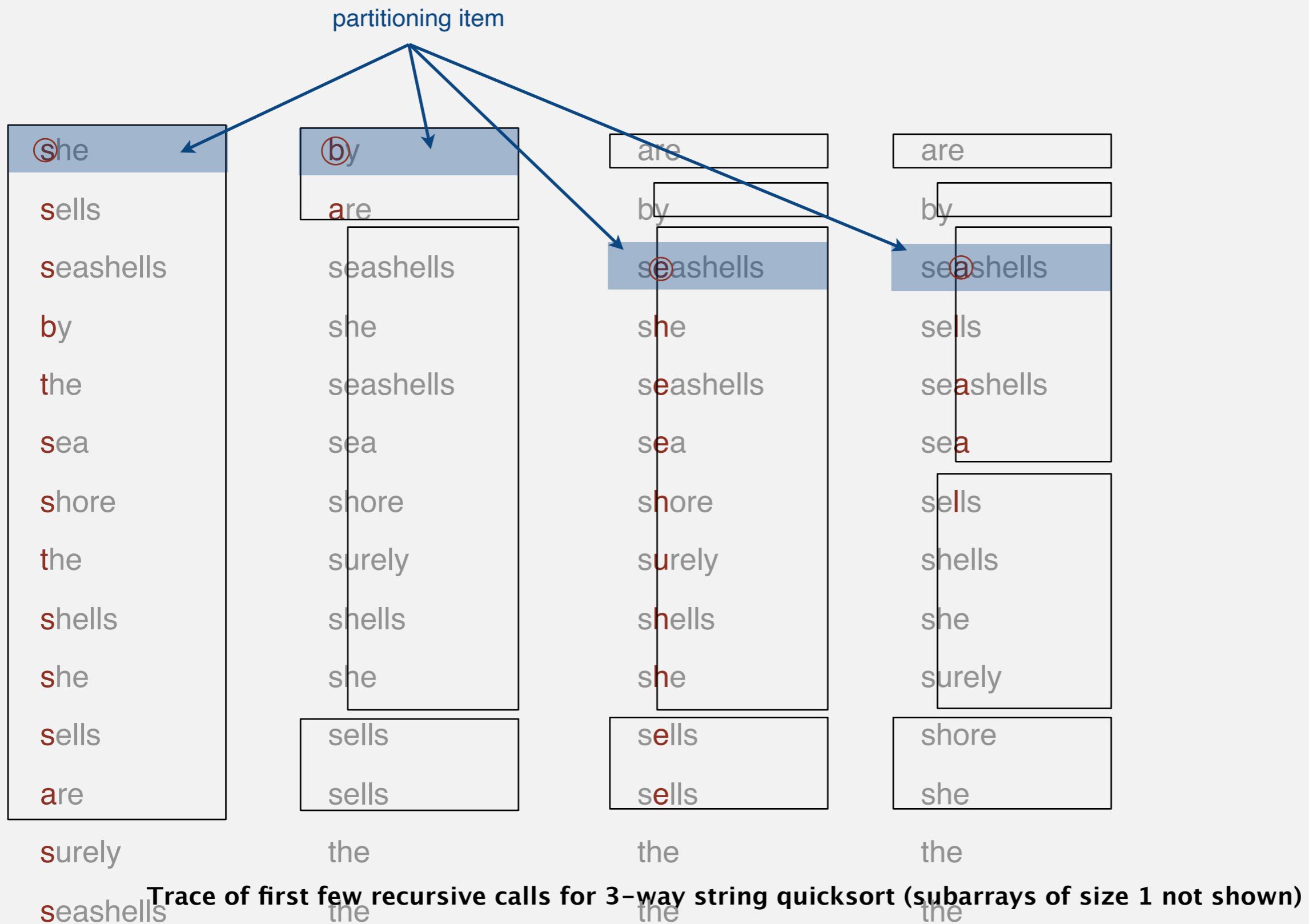
3-way string quicksort: trace of recursive calls



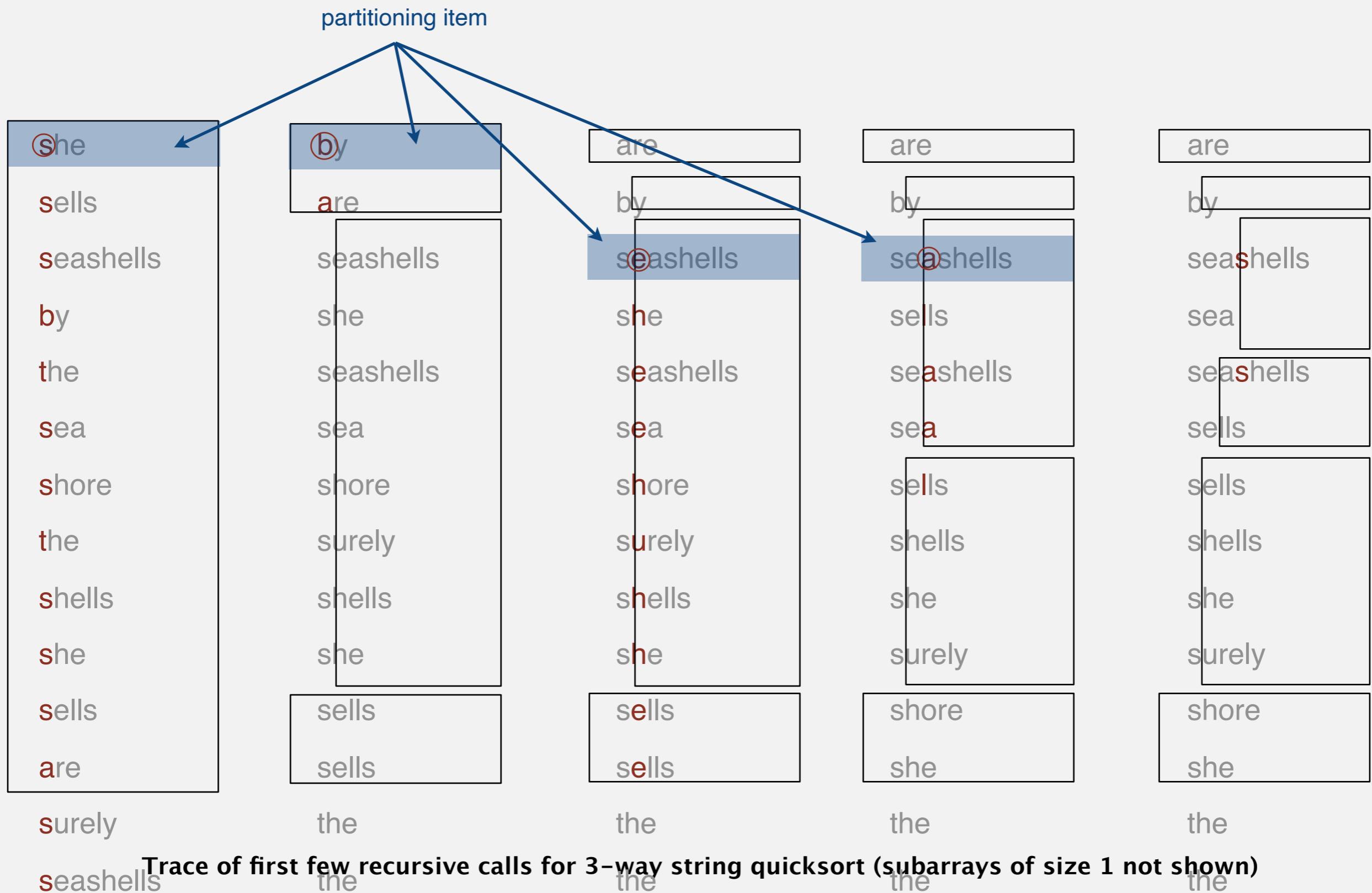
3-way string quicksort: trace of recursive calls



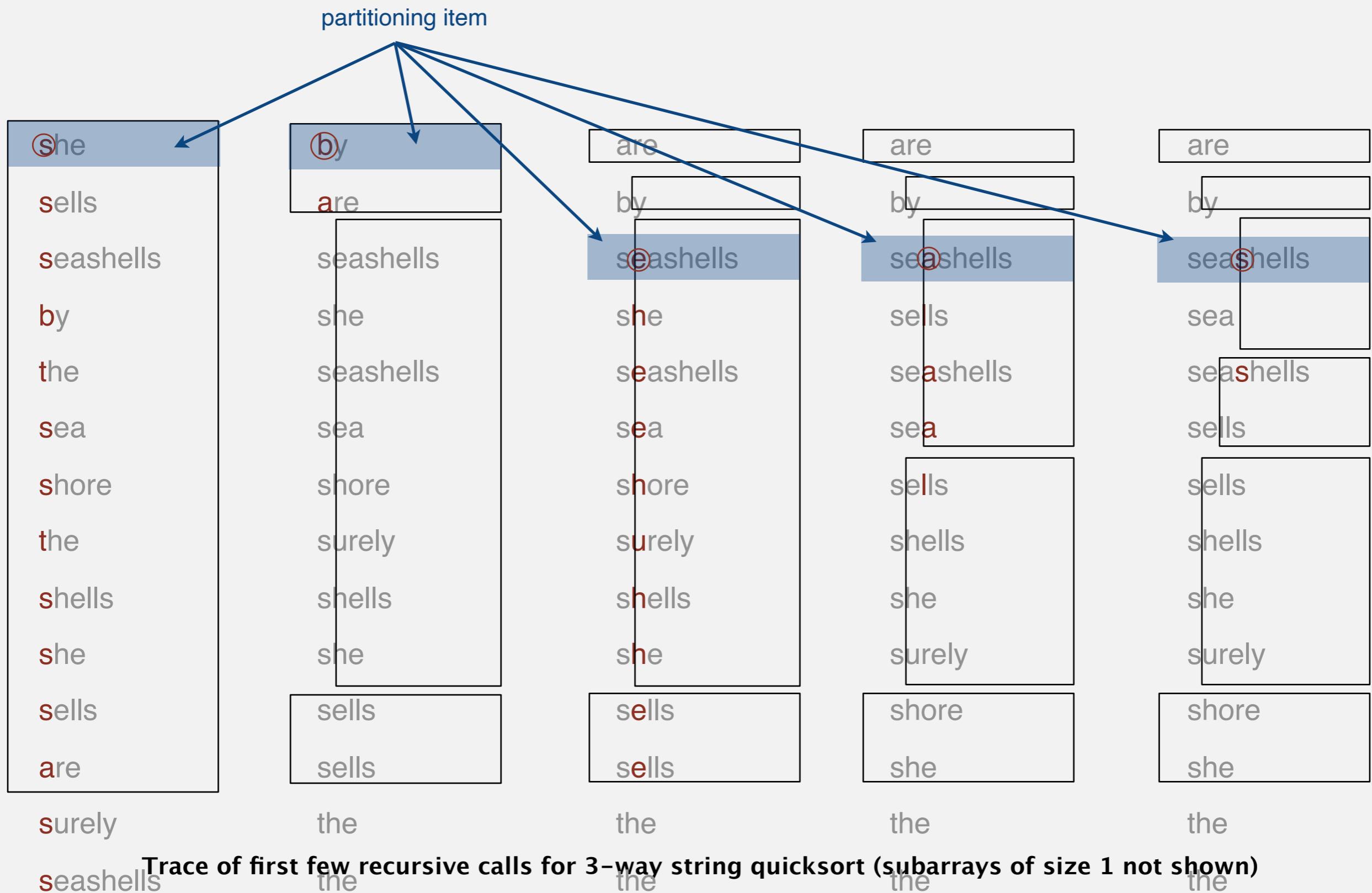
3-way string quicksort: trace of recursive calls



3-way string quicksort: trace of recursive calls



3-way string quicksort: trace of recursive calls



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

3-way string quicksort vs. standard quicksort

Standard quicksort.

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

3-way string (radix) quicksort.

- Uses $\sim 2N \ln N$ **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

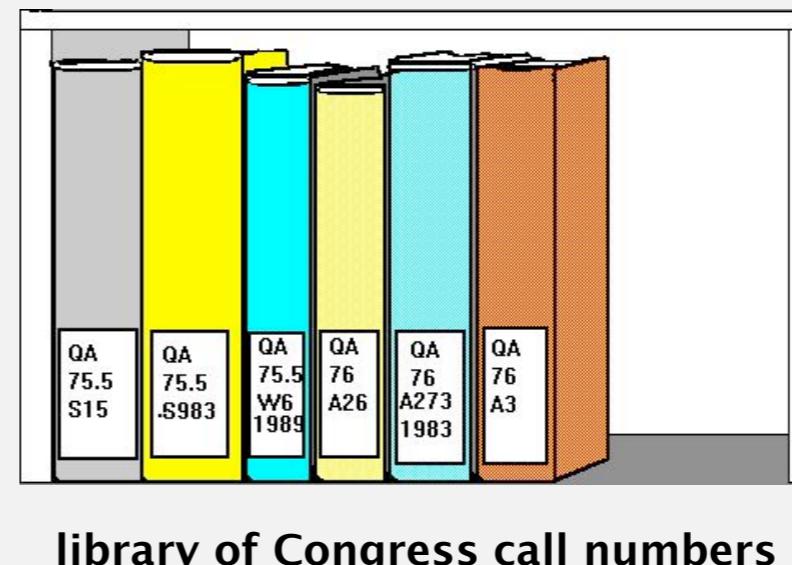
We present theoretical algorithms for sorting and searching multikey data, and derive from them practical C implementations for applications in which keys are character strings. The sorting algorithm blends Quicksort and radix sort; it is competitive with the best known C sort codes. The searching algorithm 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[].



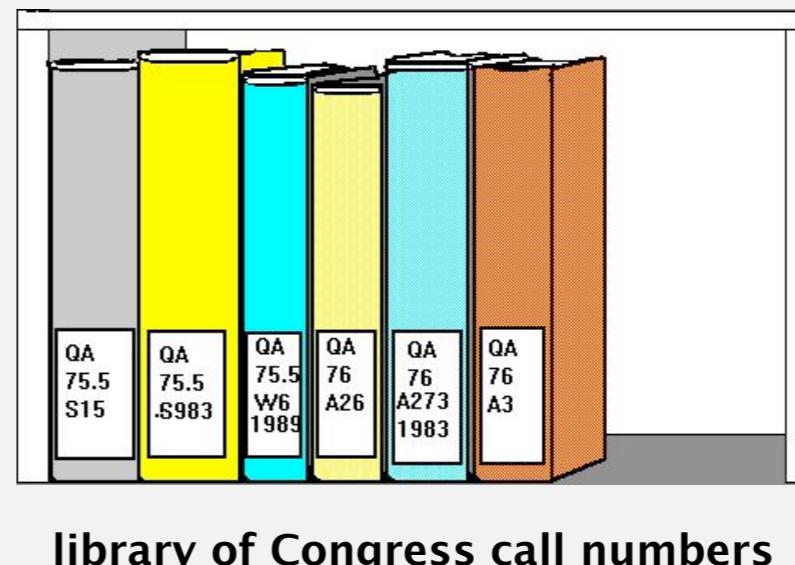
3-way string quicksort vs. MSD string sort

MSD string sort.

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

3-way string quicksort.

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



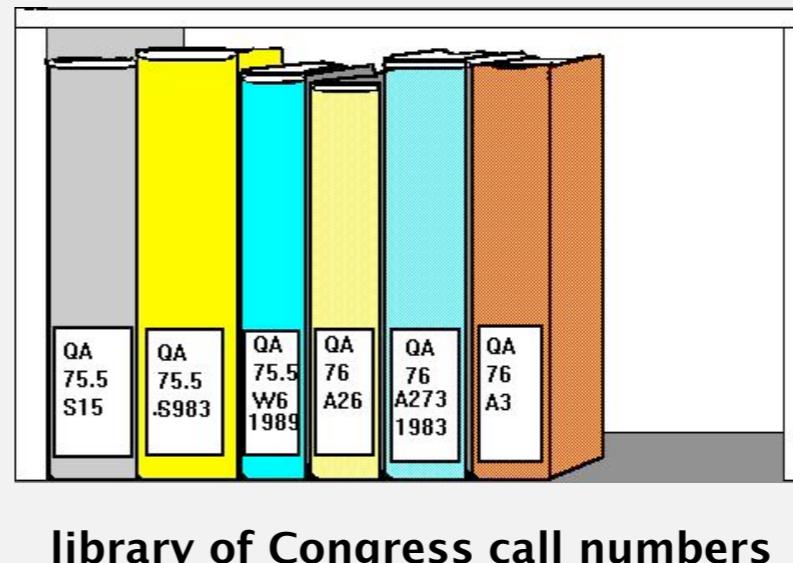
3-way string quicksort vs. MSD string sort

MSD string sort.

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

3-way string quicksort.

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



Bottom line. 3-way string quicksort is 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	$\frac{1}{2} N^2$	$\frac{1}{4} N^2$	1	✓	compareTo()
mergesort	$N \lg N$	$N \lg N$	N	✓	compareTo()
quicksort	$1.39 N \lg N$ *	$1.39 N \lg N$	$c \lg N$		compareTo()
heapsort	$2 N \lg N$	$2 N \lg N$	1		compareTo()
LSD sort †	$2 W(N + R)$	$2 W(N + R)$	$N + R$	✓	charAt()
MSD sort ‡	$2 W(N + R)$	$N \log_R N$	$N + D R$	✓	charAt()
3-way string quicksort	$1.39 W N \lg R$ *	$1.39 N \lg N$	$\log N + W$		charAt()

* probabilistic

† fixed-length W keys

‡ average-length W keys

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

5.1 STRING SORTS

- ▶ strings in Java
- ▶ 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).

```
% more tale.txt
it was the best of times
it was the worst of times
it was the age of wisdom
it was the age of foolishness
it was the epoch of belief
it was the epoch of incredulity
it was the season of light
it was the season of darkness
it was the spring of hope
it was the winter of despair
:
```

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

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

i	t	w	a	s	b	e	s	t	i	t	w	a	s	w
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

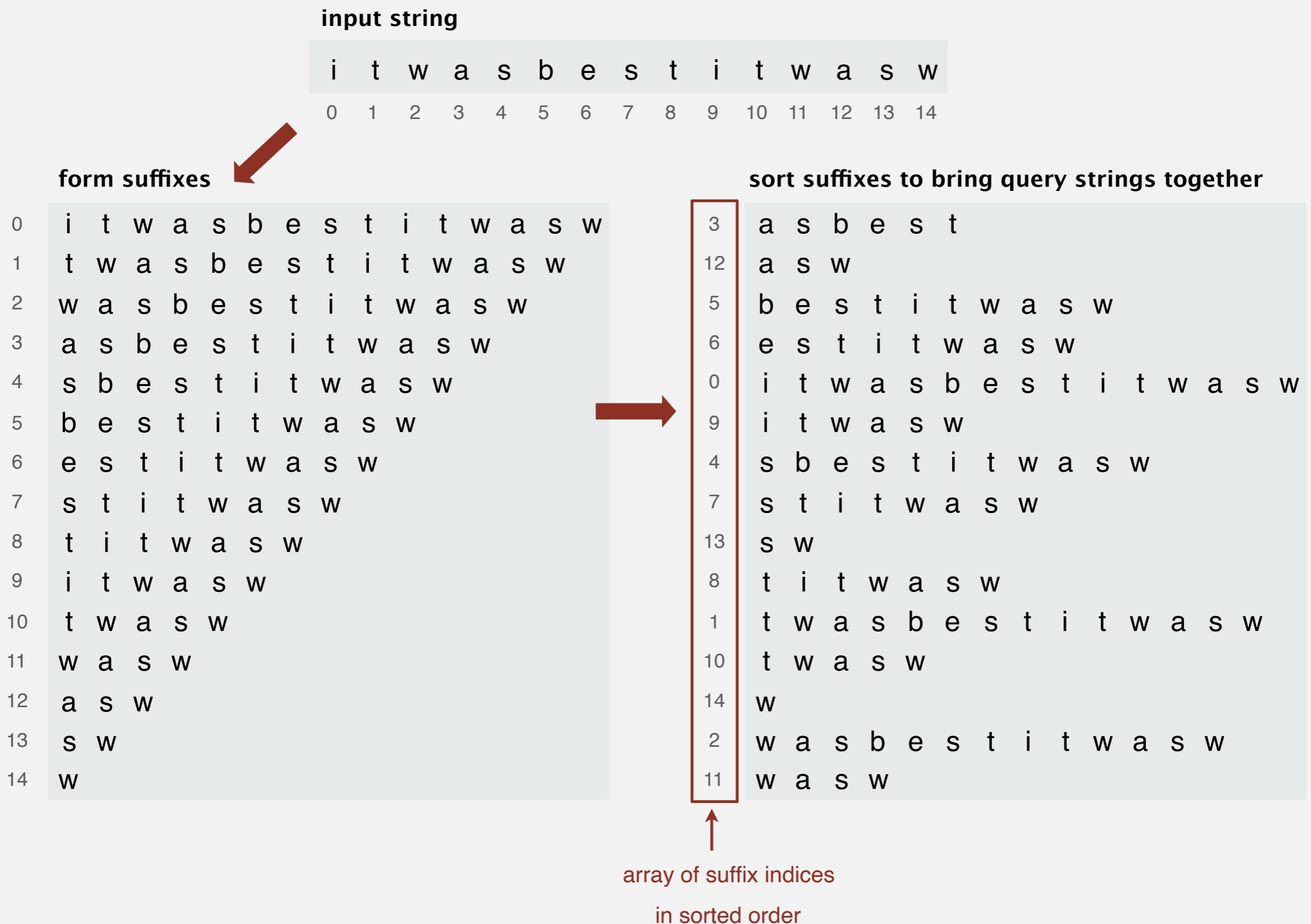
Suffix sort

input string															
	i	t	w	a	s	b	e	s	t	i	t	w	a	s	w
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	i	t	w	a	s	b	e	s	t	i	t	w	a	s	w
1	t	w	a	s	b	e	s	t	i	t	w	a	s	w	
2	w	a	s	b	e	s	t	i	t	w	a	s	w		
3	a	s	b	e	s	t	i	t	w	a	s	w			
4	s	b	e	s	t	i	t	w	a	s	w				
5	b	e	s	t	i	t	w	a	s	w					
6	e	s	t	i	t	w	a	s	w						
7	s	t	i	t	w	a	s	w							
8	t	i	t	w	a	s	w								
9	i	t	w	a	s	w									
10	t	w	a	s	w										
11	w	a	s	w											
12	a	s	w												
13	s	w													
14	w														

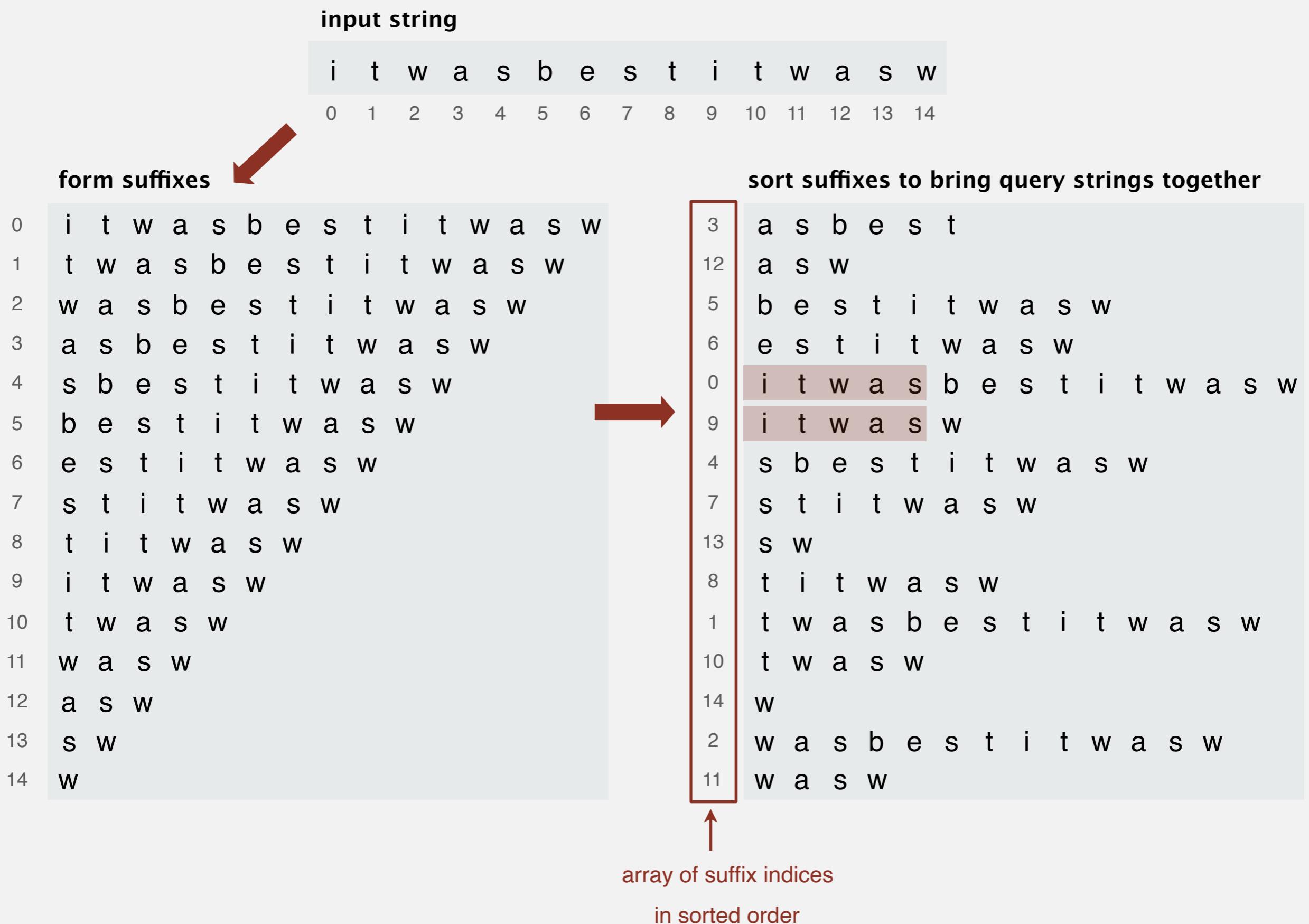
form suffixes



Suffix sort



Suffix sort



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 ...
	⋮

Suffix Arrays: theory

Q. What is worst-case running time of our suffix arrays algorithm?

- Quadratic.
- Linearithmic.
- Linear.
- None of the above.

Suffix Arrays: theory

Q. What is worst-case running time of our suffix arrays algorithm?

- Quadratic.
- Linearithmic.
- Linear.
- None of the above. ← $N^2 \log N$

suffixes

0	a a a a a a a a a a
1	a a a a a a a a a a
2	a a a a a a a a a a
3	a a a a a a a a a a
4	a a a a a a a a a a
5	a a a a a a a a a a
6	a a a a a a a a a a
7	a a a a a a a a a a
8	a a a a a a a a a a
9	a a a a a a a a a a

Suffix Arrays: theory

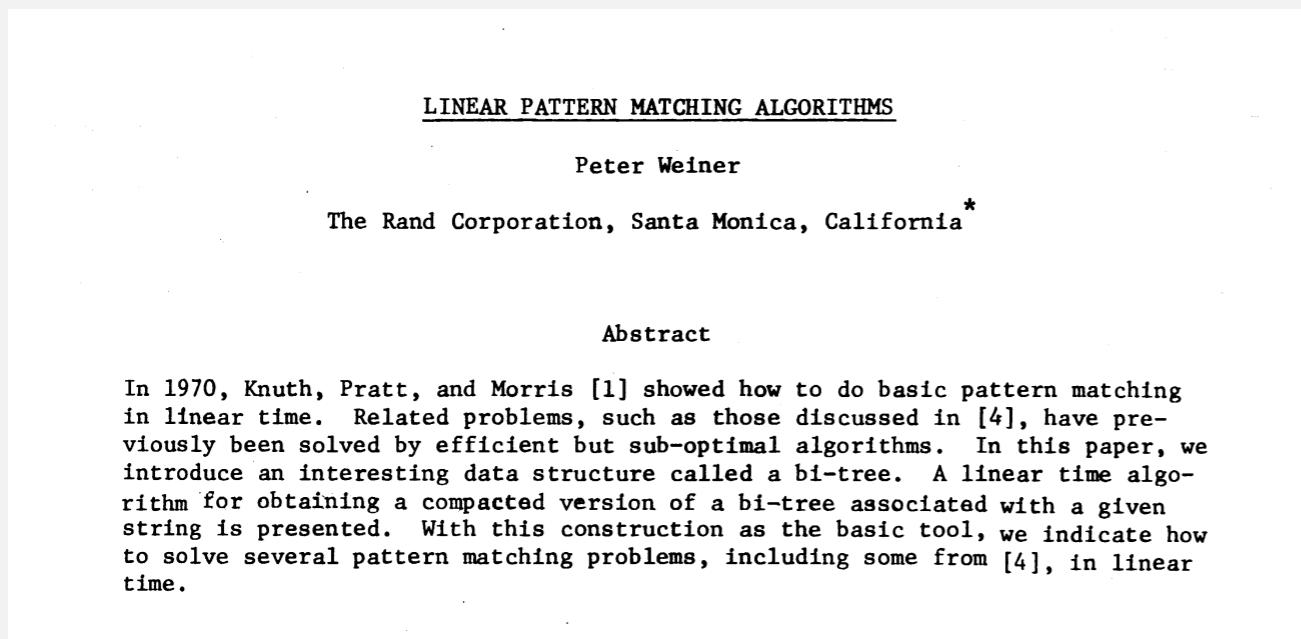
Q. What is space complexity of suffix arrays?

- Quadratic.
- Linearithmic.
- Linear.
- Nobody knows.

Suffix Arrays: theory

Q. What is space complexity of suffix arrays?

- Quadratic.
- Linearithmic.
- ✓ • Linear.  suffix trees (beyond our scope)
- Nobody knows.



Suffix Arrays: theory

Q. What is space complexity of suffix arrays?

- Quadratic.
- Linearithmic. ← Manber-Myers algorithm (see video)
- ✓ • Linear. ← suffix trees (beyond our scope)
- Nobody knows.

LINEAR PATTERN MATCHING ALGORITHMS

Peter Weiner

The Rand Corporation, Santa Monica, California*

Abstract

In 1970, Knuth, Pratt, and Morris [1] showed how to do basic pattern matching in linear time. Related problems, such as those discussed in [4], have previously been solved by efficient but sub-optimal algorithms. In this paper, we introduce an interesting data structure called a bi-tree. A linear time algorithm for obtaining a compacted version of a bi-tree associated with a given string is presented. With this construction as the basic tool, we indicate how to solve several pattern matching problems, including some from [4], in linear time.

Suffix arrays:
A new method for on-line string searches

Udi Manber¹
Gene Myers²

Department of Computer Science
University of Arizona
Tucson, AZ 85721

May 1989
Revised August 1991

Abstract

A new and conceptually simple data structure, called a suffix array, for on-line string searches is introduced in this paper. Constructing and querying suffix arrays is reduced to a sort and search paradigm that employs novel algorithms. The main advantage of suffix arrays over suffix trees is that, in practice, they use three to five times less space. From a complexity standpoint, suffix arrays permit on-line string searches of the type, “Is W a substring of A ?” to be answered in time $O(P + \log N)$, where P is the length of W and N is the length of A , which is competitive with (and in some cases slightly better than) suffix trees. The only drawback is that in those instances where the underlying alphabet is finite and small, suffix trees can be constructed in $O(N)$ time in the worst case, versus $O(N \log N)$ time for suffix arrays. However, we give an augmented algorithm that, regardless of the alphabet size, constructs suffix arrays in $O(N)$ **expected** time, albeit with lesser space efficiency. We believe that suffix arrays will prove to be better in practice than suffix trees for many applications.

Suffix Arrays: practice

Applications. Bioinformatics, information retrieval, data compression, ...

Suffix Arrays: practice

Applications. Bioinformatics, information retrieval, data compression, ...

Many ingenious algorithms.

- Memory footprint very important.
- State-of-the art still changing.

year	algorithm	worst case	memory
1990	Manber-Myers	$N \log N$	$8 N$
1999	Larsson-Sadakane	$N \log N$	$8 N$
2003	Kärkkäinen-Sanders	N	$13 N$
2003	Ko-Aluru	N	$10 N$

Suffix Arrays: practice

Applications. Bioinformatics, information retrieval, data compression, ...

Many ingenious algorithms.

- Memory footprint very important.
- State-of-the art still changing.

year	algorithm	worst case	memory
1990	Manber-Myers	$N \log N$	$8N$
1999	Larsson-Sadakane	$N \log N$	$8N$
2003	Kärkkäinen-Sanders	N	$13N$
2003	Ko-Aluru	N	$10N$
2008	divsufsort2	$N \log N$	$5N$
2010	sais	N	$6N$

good choices
(Yuta Mori)

Longest repeated substring

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

```
aacaagttacaagacatgatgctgtactaggagagttatactggtcg  
tcaaacctgaacctaattccttgtgtacacacactactactgtcgtc  
gtcatatatcgagatcatcgaacccggaaggccggacaaggcgggg  
ggtagatagatagaccctagatacacatacatagatctagta  
gctagctcatcgatacacactctcacactcaagagttatactggtca  
acacactactacgacagacgaccaaccagacagaaaaaaaactc  
tatatctataaaa
```

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

Longest repeated substring

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

```
aacaagttacaagacatgatgctgtactaggagagttatactggtcg  
tcaaaacctgaacctaatccttgtgtgttacacacactactactgtcgtc  
gtcatatatcgagatcatcgaaccggaggccggacaggcgggg  
ggtatagatagatagacccctagatacacacatacatagatctagca  
gctagctcatcgatacacactcacactcaaggatttatactggtca  
acacactactacgacagcaccaaccgacagaaaaaaaactc  
tatatctataaaa
```

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

Longest repeated substring: a musical application

Visualize repetitions in music. <http://www.bewitched.com>

Mary Had a Little Lamb



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.



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

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
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														

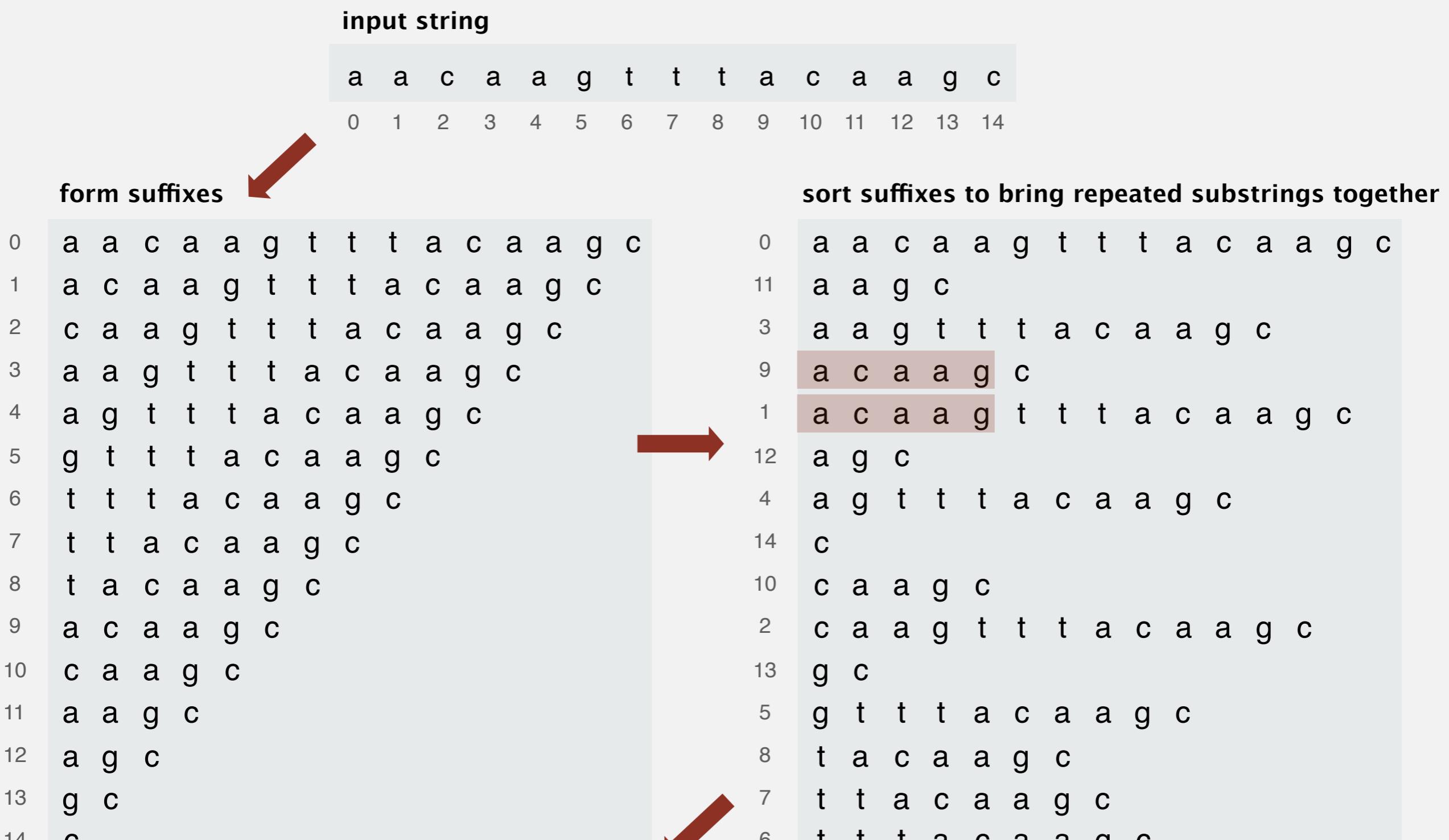
form suffixes



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	a	c	a	a	g	c			
13	g	c													
5	g	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						

Longest repeated substring: a sorting solution



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

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 arrays solution with insertion sort.
- D uses suffix arrays solution with LSD string sort.
- E uses suffix arrays solution with 3-way string quicksort.

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

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 arrays solution with insertion sort.
- D uses suffix arrays solution with LSD string sort.
- ✓ • E uses suffix arrays 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 Arrays: 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	sorted suffixes
0	t w i n s t w i n s	i n s
1	w i n s t w i n s	i n s t w i n s
2	i n s t w i n s	n s
3	n s t w i n s	n s t w i n s
4	s t w i n s	s
5	t w i n s	s t w i n s
6	w i n s	t w i n s
7	i n s	t w i n s t w i n s
8	n s	w i n s
9	s	w i n s t w i n s

Suffix Arrays: 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	sorted suffixes
0	t w i n s t w i n s	i n s
1	w i n s t w i n s	i n s t w i n s
2	i n s t w i n s	n s
3	n s t w i n s	n s t w i n s
4	s t w i n s	s
5	t w i n s	s t w i n s
6	w i n s	t w i n s
7	i n s	t w i n s t w i n s
8	n s	w i n s
9	s	w i n s t w i n s

LRS needs at least $1 + 2 + 3 + \dots + D$ character compares,
where $D = \text{length of longest match}$.

Suffix Arrays: 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	sorted suffixes
0	t w i n s t w i n s	7 i n s
1	w i n s t w i n s	2 i n s t w i n s
2	i n s t w i n s	8 n s
3	n s t w i n s	3 n s t w i n s
4	s t w i n s	9 s
5	t w i n s	4 s t w i n s
6	w i n s	5 t w i n s
7	i n s	0 t w i n s t w i n s
8	n s	6 w i n s
9	s	1 w i n s t w i n s

LRS needs at least $1 + 2 + 3 + \dots + D$ character compares,
where $D = \text{length of longest match}$.

Running time. Quadratic (or worse) in D for LRS (and also for sort).

Suffix Arrays challenge

Problem. Suffix sort a string of length N .

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

- Quadratic.
- Linearithmic.
- Linear.
- Nobody knows.

Suffix Arrays challenge

Problem. Suffix sort a string of length N .

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

- Quadratic.
- ✓ • Linearithmic.  Manber-Myers algorithm
- Linear.
- Nobody knows.

Suffix Arrays challenge

Problem. Suffix sort a string of length N .

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

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

Suffix Arrays in linearithmic time

Manber-Myers 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.

Suffix Arrays in linearithmic time

Manber-Myers 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

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

use key-indexed counting to sort by first character

Linearithmic suffix sort example: phase 1

i	a[]	b[]	suffixes sorted by their first character	
0	17	12	0	
1	1	1	a b a a a a b c b a b a a a a a a 0	
2	16	16	a 0	
3	3	3	a a a b c b a b a a a a a a 0	
4	4	4	a a a b c b a b a a a a a a 0	
5	5	5	a a b c b a b a a a a a a 0	
6	6	6	a b c b a b a a a a a a 0	
7	15	15	a a 0	inverse[a[i]] = i
8	14	17	a a a 0	
9	13	13	a a a a 0	
10	12	11	a a a a a 0	
11	10	14	a b a a a a a 0	
12	0	10	b a b a a a a b c b a b a a a a a 0	
13	9	9	b a b a a a a a 0	
14	11	8	b a a a a a 0	
15	7	7	b c b a b a a a a a a 0	
16	2	2	b a a a a b c b a b a a a a a a 0	
17	8	0	c b a b a a a a a 0	

sort blue subarrays using second character as key

Linearithmic suffix sort example: phase 2

i	a[]	b[]	suffixes sorted by their first two characters
0	17	12	0
1	16	10	a 0
2	12	15	a a a a a a 0
3	3	3	a a a a b c b a b a a a a a a 0
4	4	4	a a a b c b a b a a a a a a 0
5	5	5	a a b c b a b a a a a a a 0
6	13	6	a a a a 0
7	15	9	a a 0
8	14	16	a a a 0
9	6	17	a b c b a b a a a a a a 0
10	1	13	a b a a a a b c b a b a a a a a a 0
11	10	11	a b a a a a a 0
12	0	14	b a b a a a a b c b a b a a a a a a 0
13	9	6	b a b a a a a a 0
14	11	8	b a b a a a a 0
15	2	7	b a a a a b c b a b a a a a a a 0
16	7	1	b c b a b a a a a a 0
17	8	0	c b a b a a a a a 0

sort blue subarrays using third and fourth characters as key

Linearithmic suffix sort example: phase 3

i	a[]	b[]	suffixes sorted by their first four characters
0	17	14	0
1	16	9	a 0
2	15	12	a a 0
3	14	4	a a a 0
4	3	7	a a a a b c b a b a a a a a 0
5	12	8	a a a a a 0
6	13	11	a a a a 0
7	4	16	a a a b c b a b a a a a a 0
8	5	17	a a b c b a b a a a a a 0
9	1	15	a b a a a a b c b a b a a a a a 0
10	10	10	a b a a a a a 0
11	6	13	a b c b a b a a a a a 0
12	2	5	b a a a a b c b a b a a a a a 0
13	11	6	b a a a a a 0
14	0	3	b a b a a a a b c b a b a a a a 0
15	9	2	b a b a a a a a 0
16	7	1	b c b a b a a a a a 0
17	8	0	c b a b a a a a a 0

sort blue subarrays using fifth through eighth characters as key

Linearithmic suffix sort example: phase 4

i	a[]	b[]	suffixes sorted by their first eight characters
0	17	15	0
1	16	10	a 0
2	15	13	a a 0
3	14	6	a a a 0
4	13	7	a a a a 0
5	12	8	a a a a a 0
6	3	11	a a a a b c b a b a a a a a 0
7	4	16	a a a b c b a b a a a a a 0
8	5	17	a a b c b a b a a a a a 0
9	10	14	a b a a a a a 0
10	1	9	a b a a a a b c b a b a a a a a 0
11	6	12	a b c b a b a a a a 0
12	11	5	b a a a a a 0
13	2	4	b a a a a b c b a b a a a a a 0
14	9	3	b a b a a a a a 0
15	0	2	b a b a a a a b c b a b a a a a a 0
16	7	1	b c b a b a a a a a 0
17	8	0	c b a b a a a a a 0

b[a[i]] = i

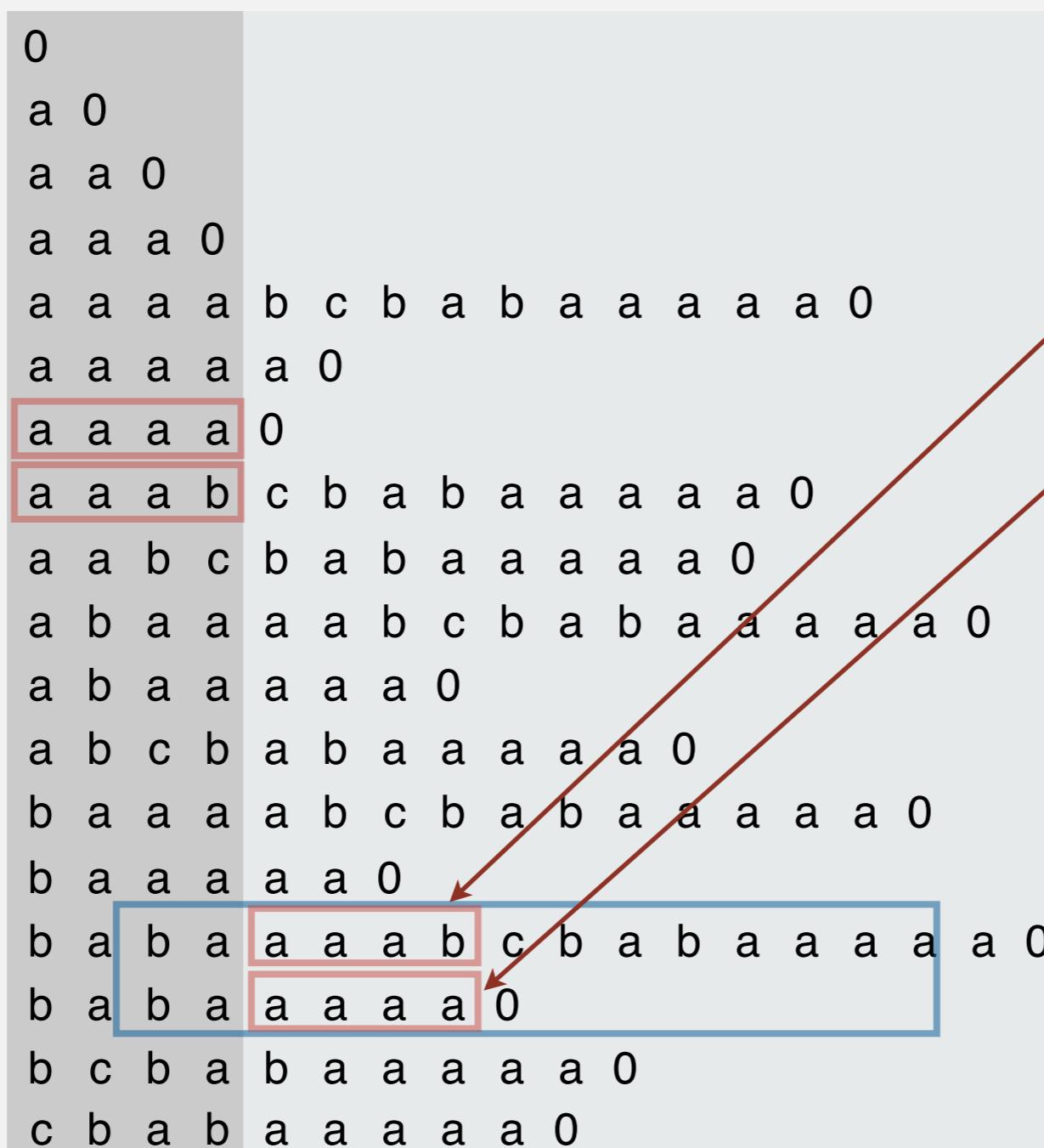
finished (no equal keys)

Linearithmic suffix sort example: inverse array

i	a[]	b[]	suffixes sorted by their first four characters
0	17	14	0
1	16	9	a 0
2	15	12	a a 0
3	14	4	a a a 0
4	3	7	a a a a b c b a b a a a a a 0
5	12	8	a a a a a 0
6	13	11	a a a a 0
7	4	16	a a a b c b a b a a a a a 0
8	5	17	a a b c b a b a a a a a 0
9	1	15	a b a a a a b c b a b a a a a 0
10	10	10	a b a a a a a a 0
11	6	13	a b c b a b a a a a a 0
12	2	5	b a a a a b c b a b a a a a 0
13	11	6	b a a a a a 0
14	0	3	b a b a a a a b c b a b a a a a 0
15	9	2	b a b a a a a a 0
16	7	1	b c b a b a a a a a 0
17	8	0	c b a b a a a a a 0

$0 + 4 = 4$
 $9 + 4 = 13$

suffixes₄[13] ≤ suffixes₄[4]
 (because b[13] < b[4])
so suffixes₈[9] ≤ suffixes₈[0]



use $b[a[i]] + 4$ as the sorting key, where $b[a[i]] = i$

String sorting summary

We can develop linear-time sorts.

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

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.

- Input size is amount of data in keys (not number of keys).
- Not all of the data has to be examined.

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.

- Input size is 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.

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.

- Input size is 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.

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

5.2 TRIES

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

Summary of the performance of symbol-table implementations

Order of growth of the frequency of operations.

implementation	typical case			ordered operations	operations on keys
	search	insert	delete		
red-black BST	$\log N$	$\log N$	$\log N$	✓	<code>compareTo()</code>
hash table	1^\dagger	1^\dagger	1^\dagger		<code>equals()</code> <code>hashCode()</code>

† under uniform hashing assumption

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

† under uniform hashing assumption

Q. Can we do better?

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

† under uniform hashing assumption

Q. Can we do better?

A. Yes, if we can avoid examining the entire key, as with string sorting.

String symbol table basic API

String symbol table. Symbol table specialized to string keys.

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

```
    StringST()
```

create an empty symbol table

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

put key-value pair into the symbol table

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

return value paired with given key

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

delete key and corresponding value

```
    :
```

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	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	$4N$	1.4	97.4
hashing (linear probing)	L	L	L	$4N \text{ to } 16N$	0.76	40.6

Parameters

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

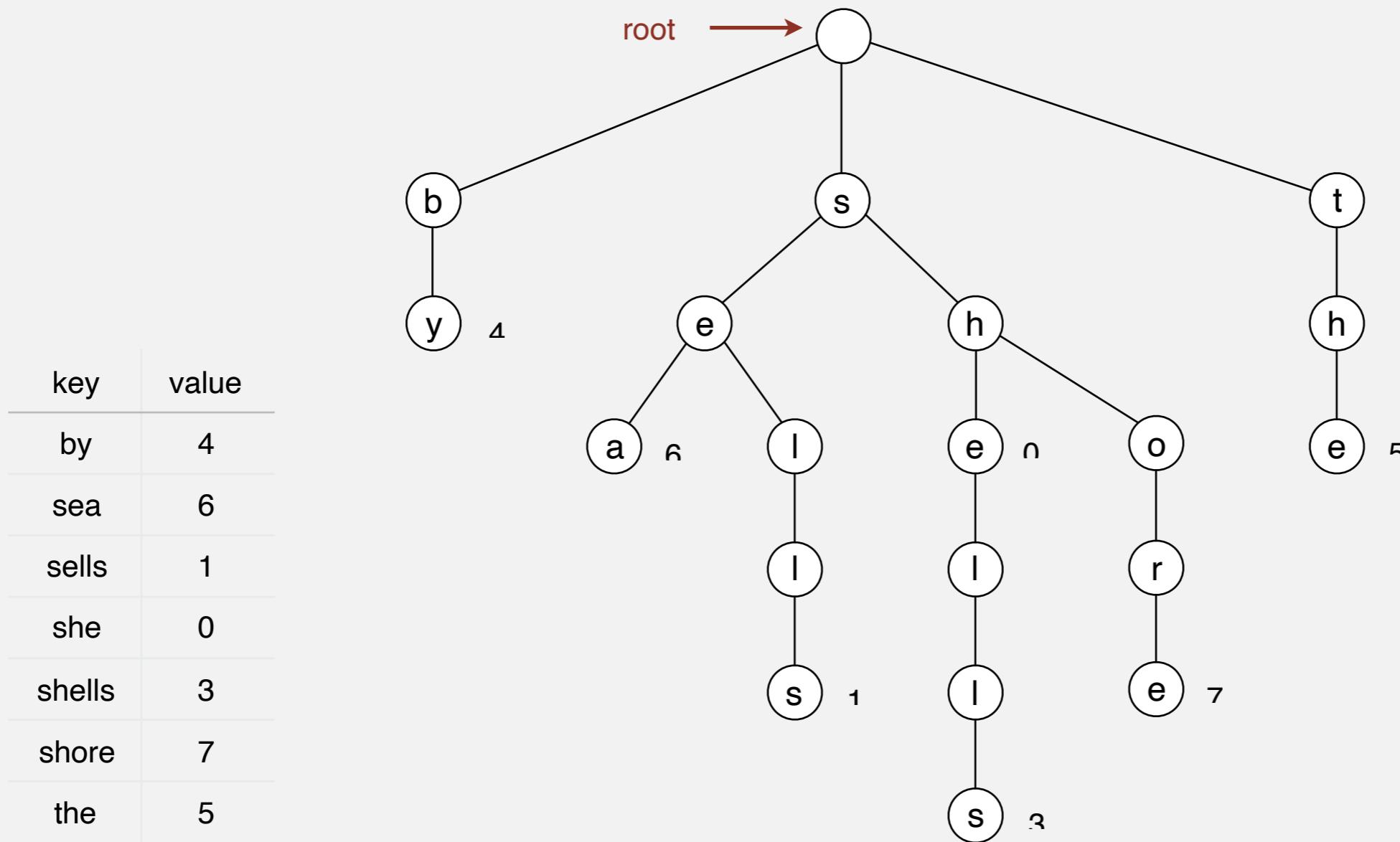
file	size	words	distinct
moby.txt	1.2 MB	210 K	32 K
actors.txt	82 MB	11.4 M	900 K

Challenge. Efficient performance for string keys.

Tries

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)

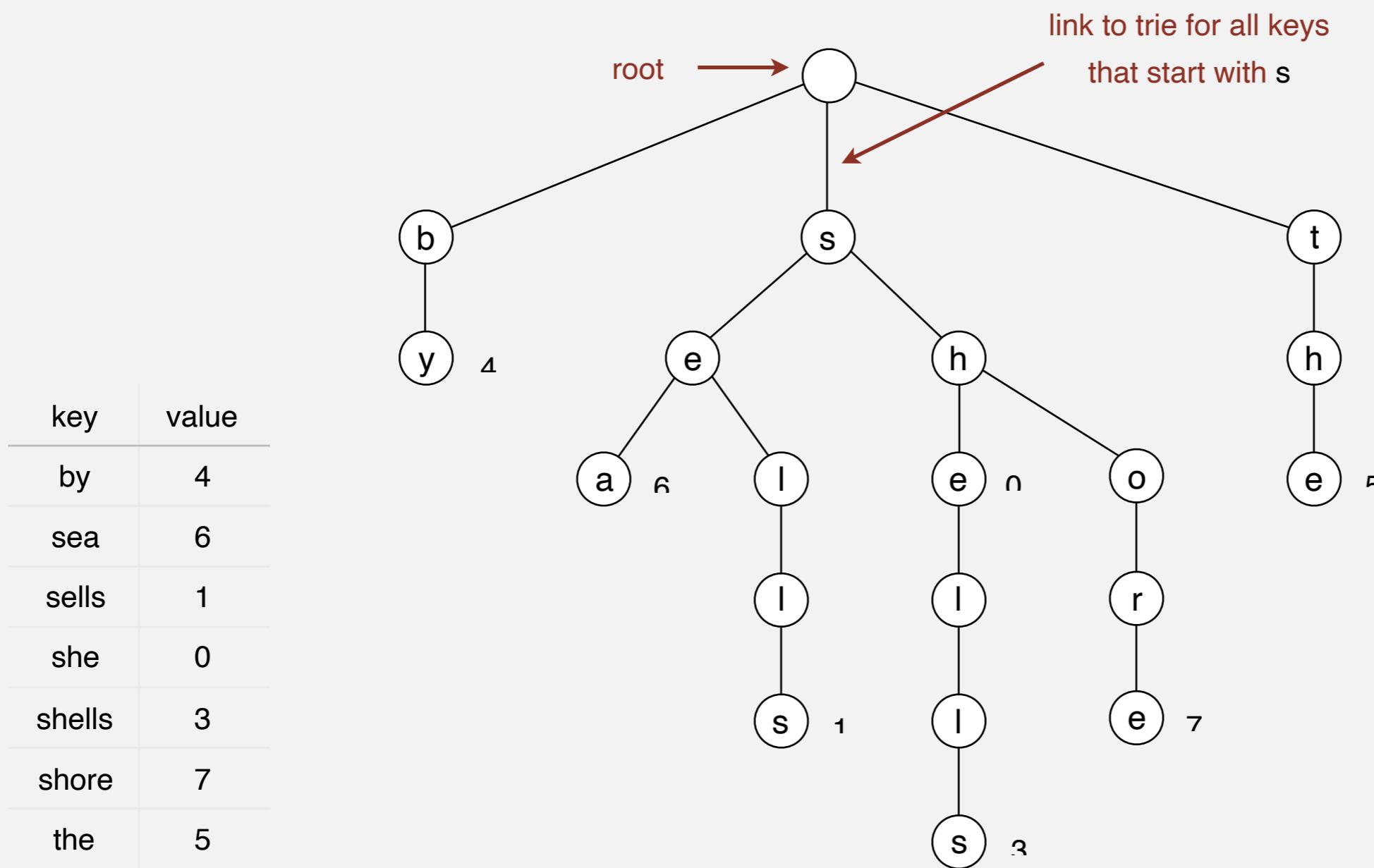


key	value
by	4
sea	6
sells	1
she	0
shells	3
shore	7
the	5

Tries

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

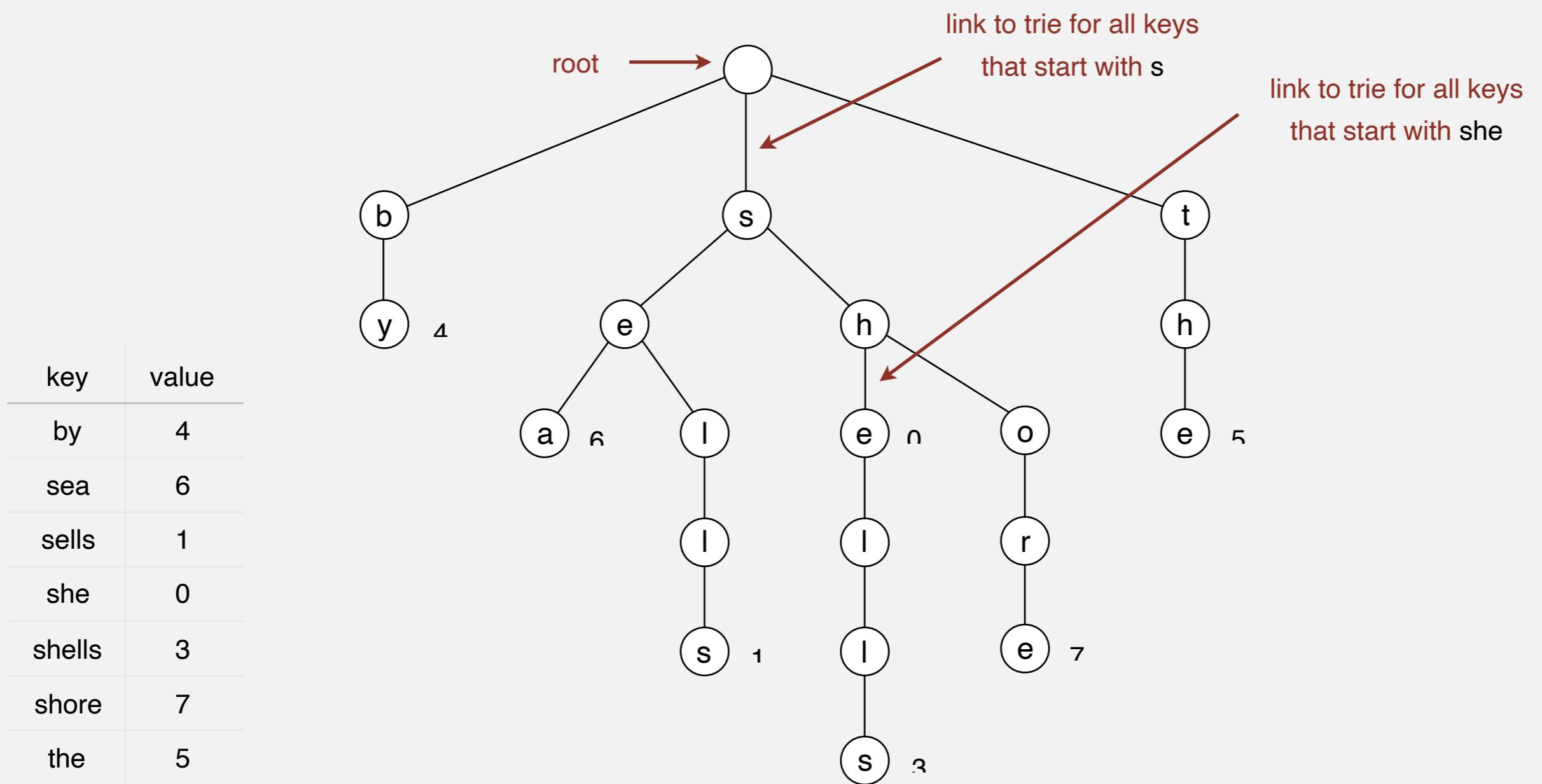
- Store characters in nodes (not keys).
- Each node has R children, one for each possible character.
(for now, we do not draw null links)



Tries

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

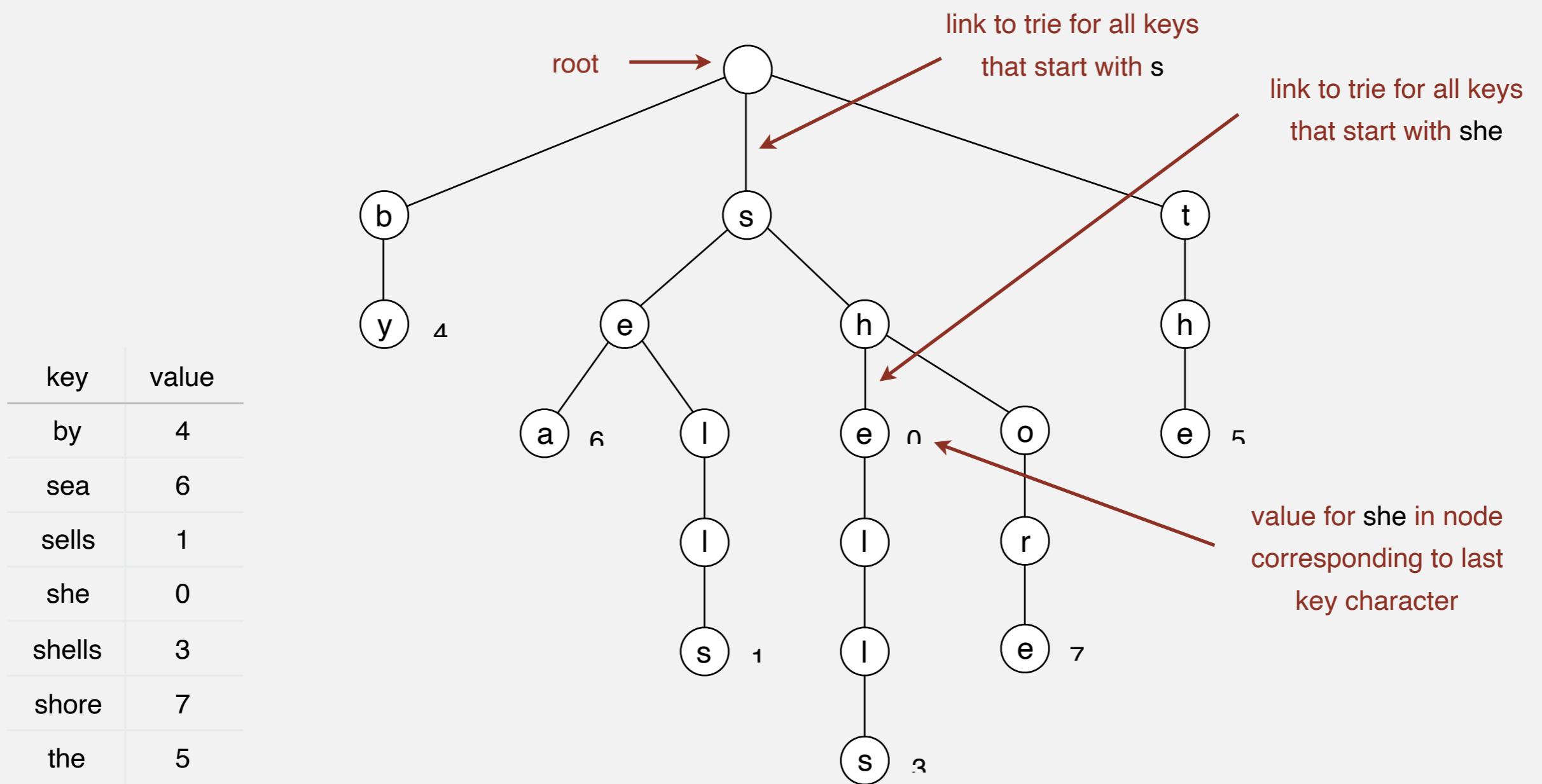
- Store characters in nodes (not keys).
- Each node has R children, one for each possible character.
(for now, we do not draw null links)



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)

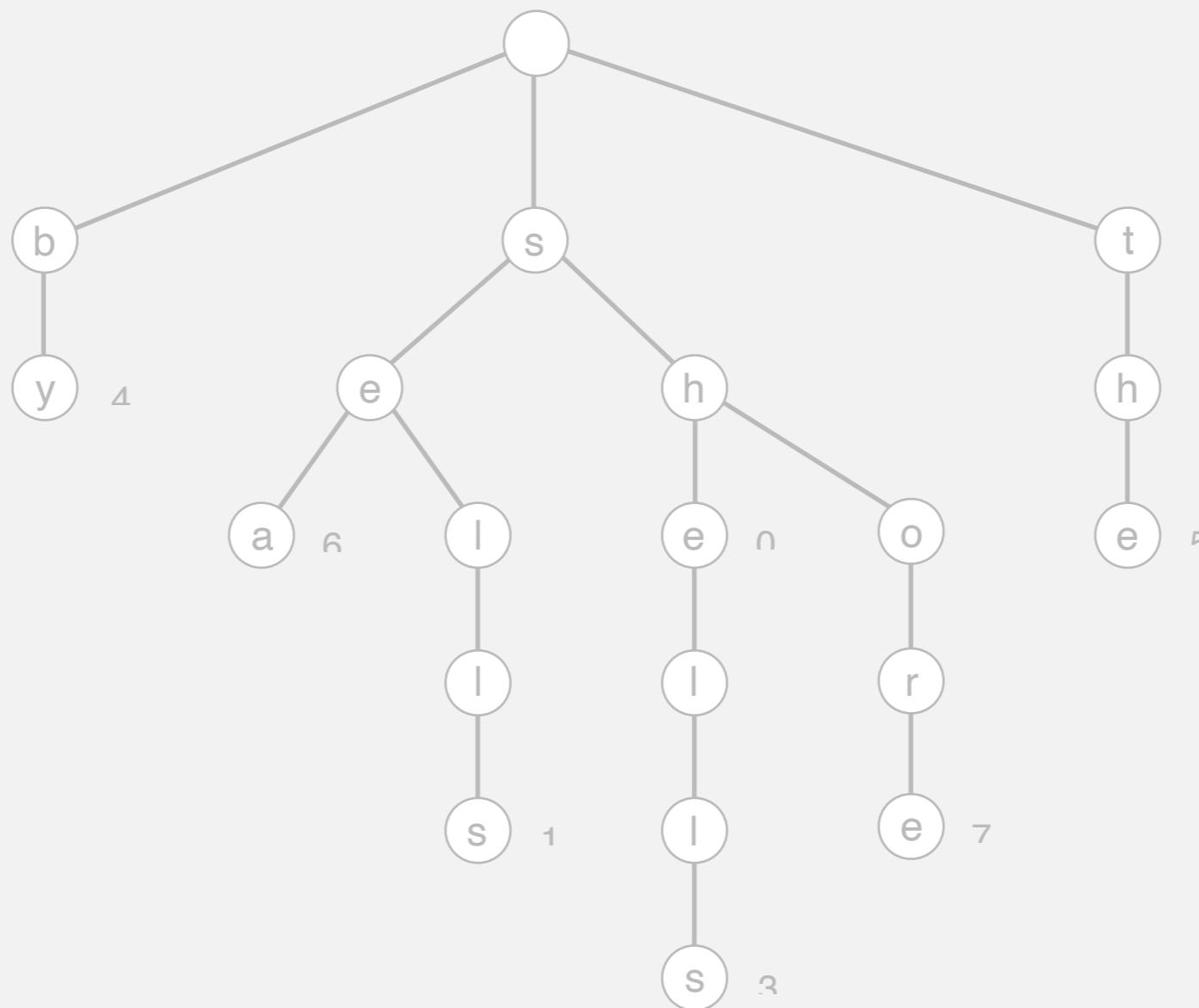


Search in a trie

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")

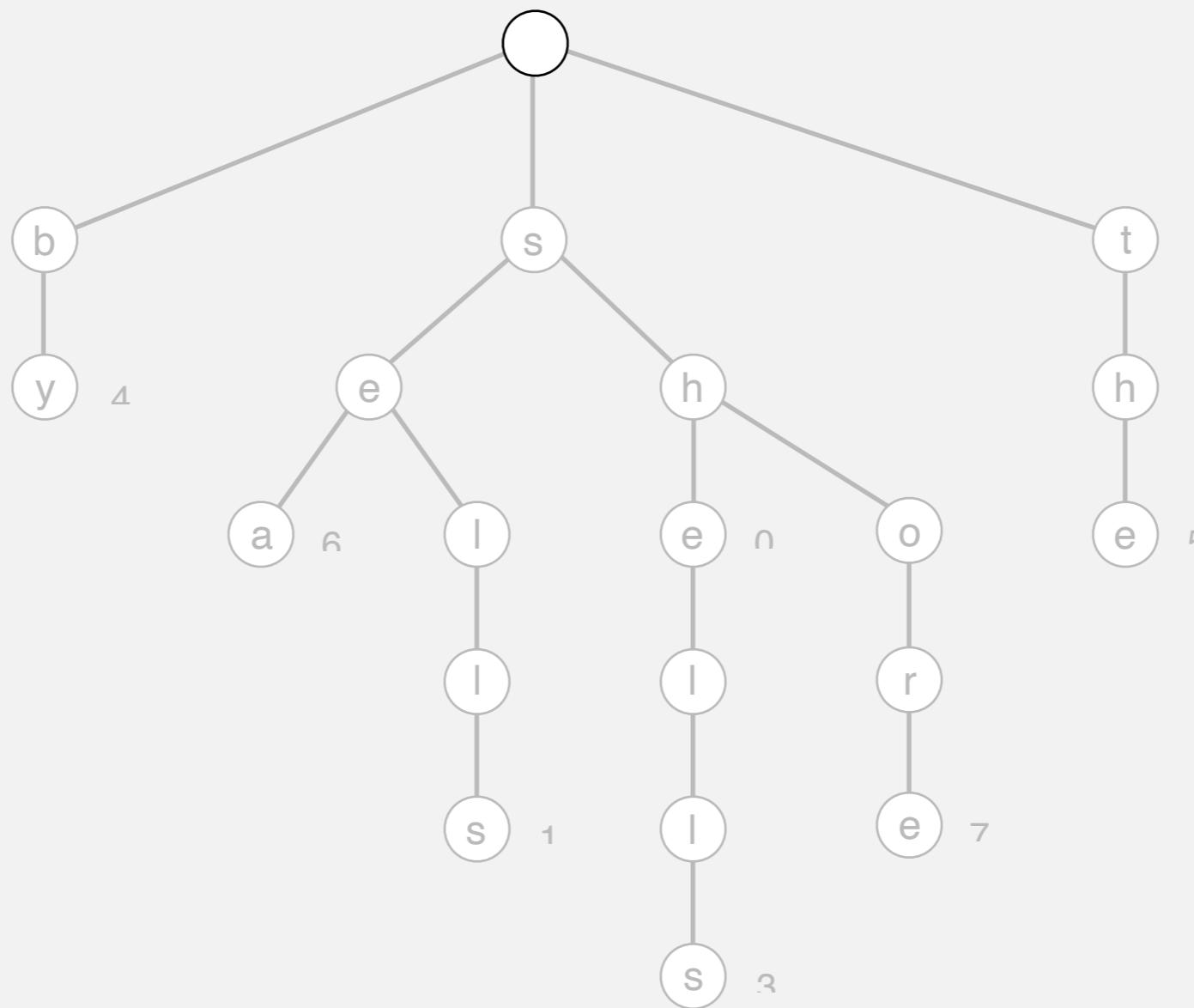


Search in a trie

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("shells")

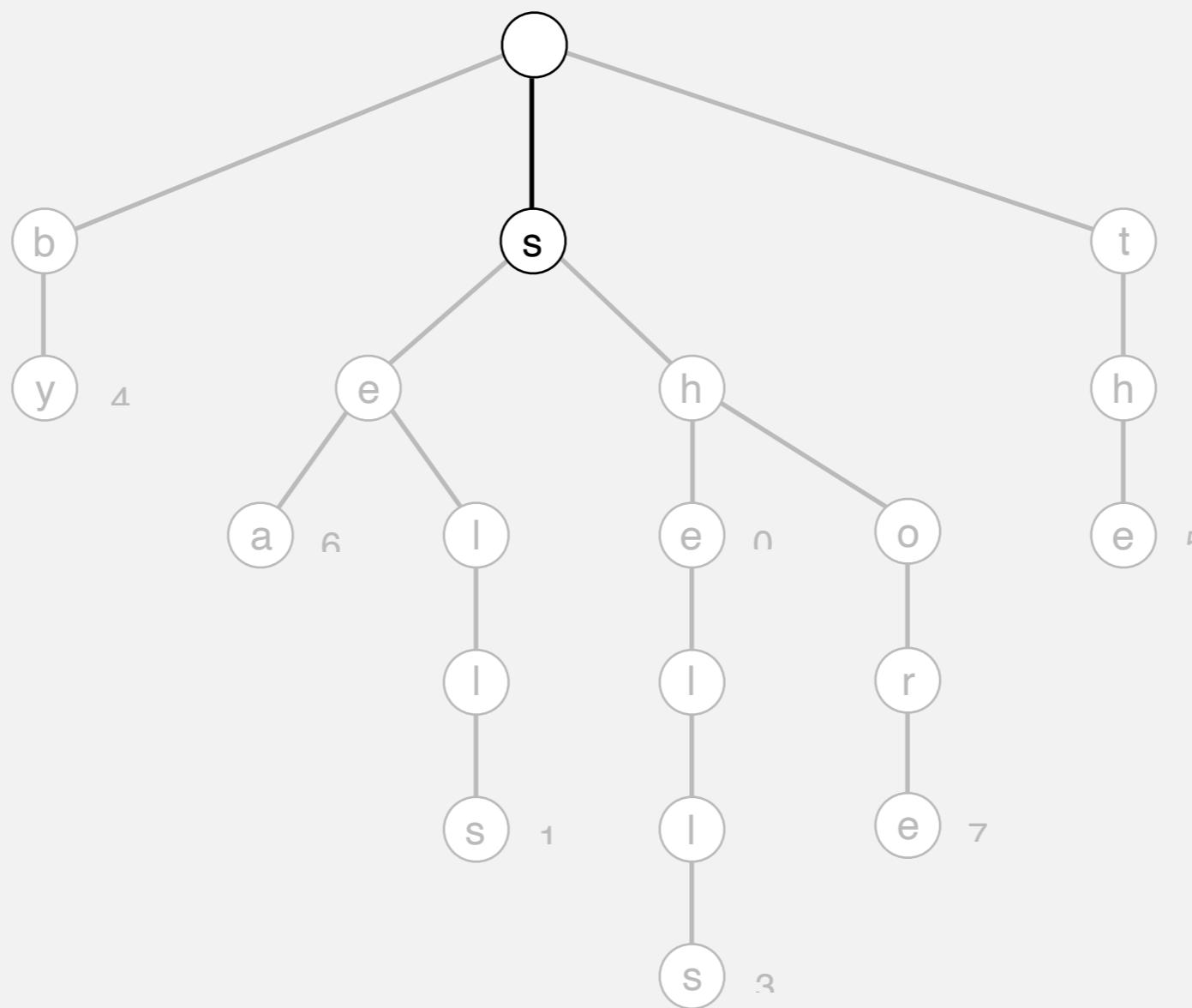


Search in a trie

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("shells")

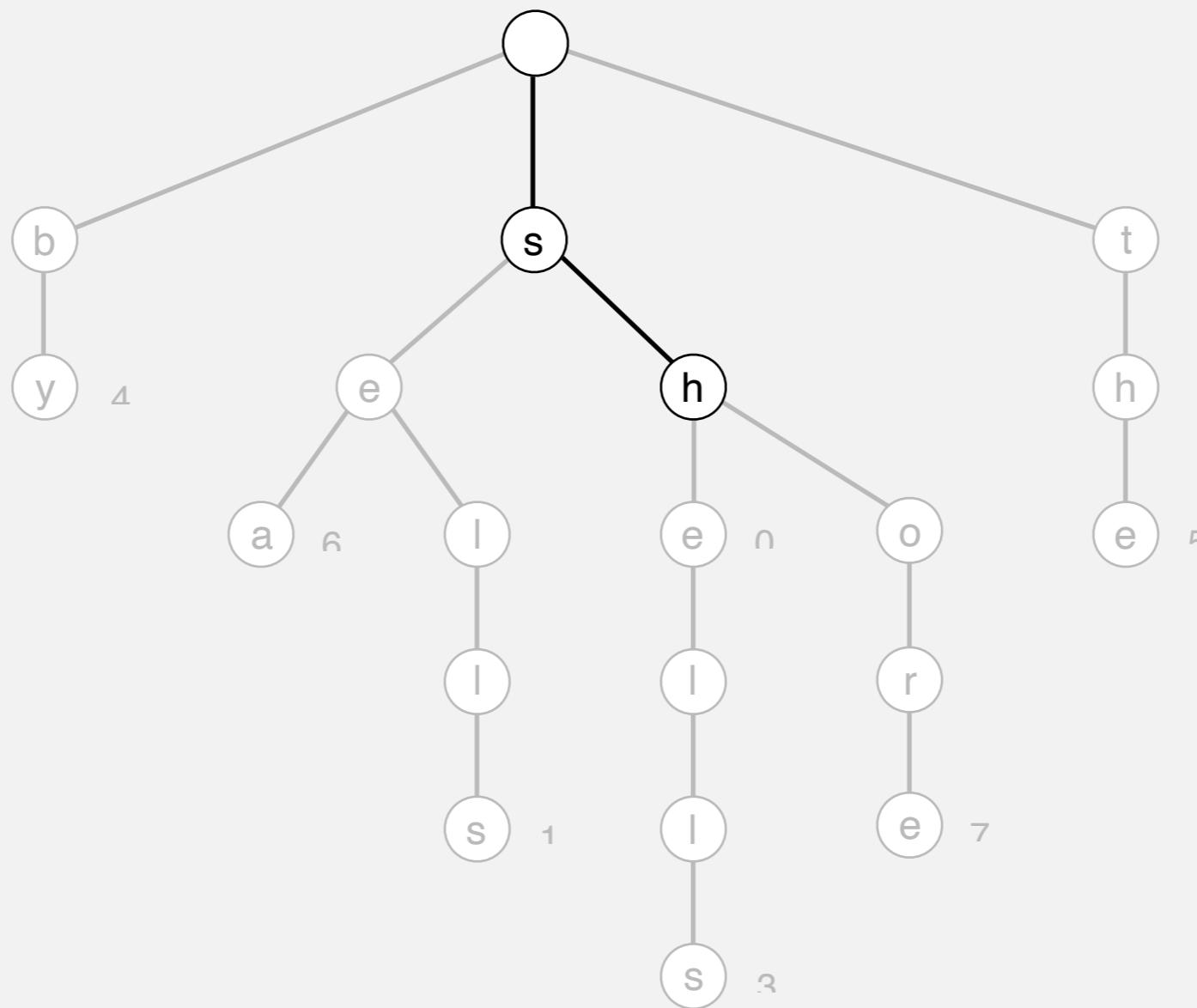


Search in a trie

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("shells")

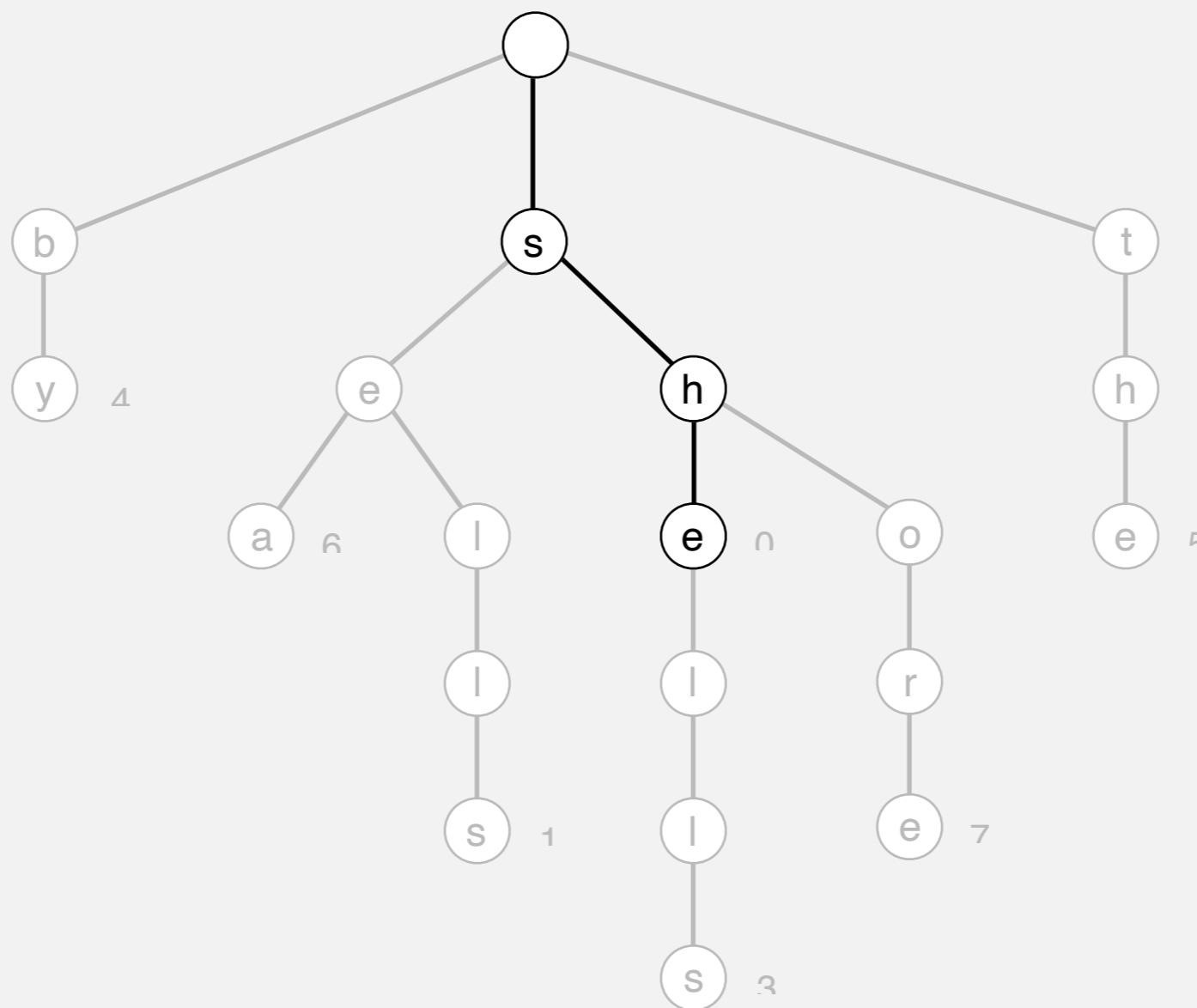


Search in a trie

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("shells")

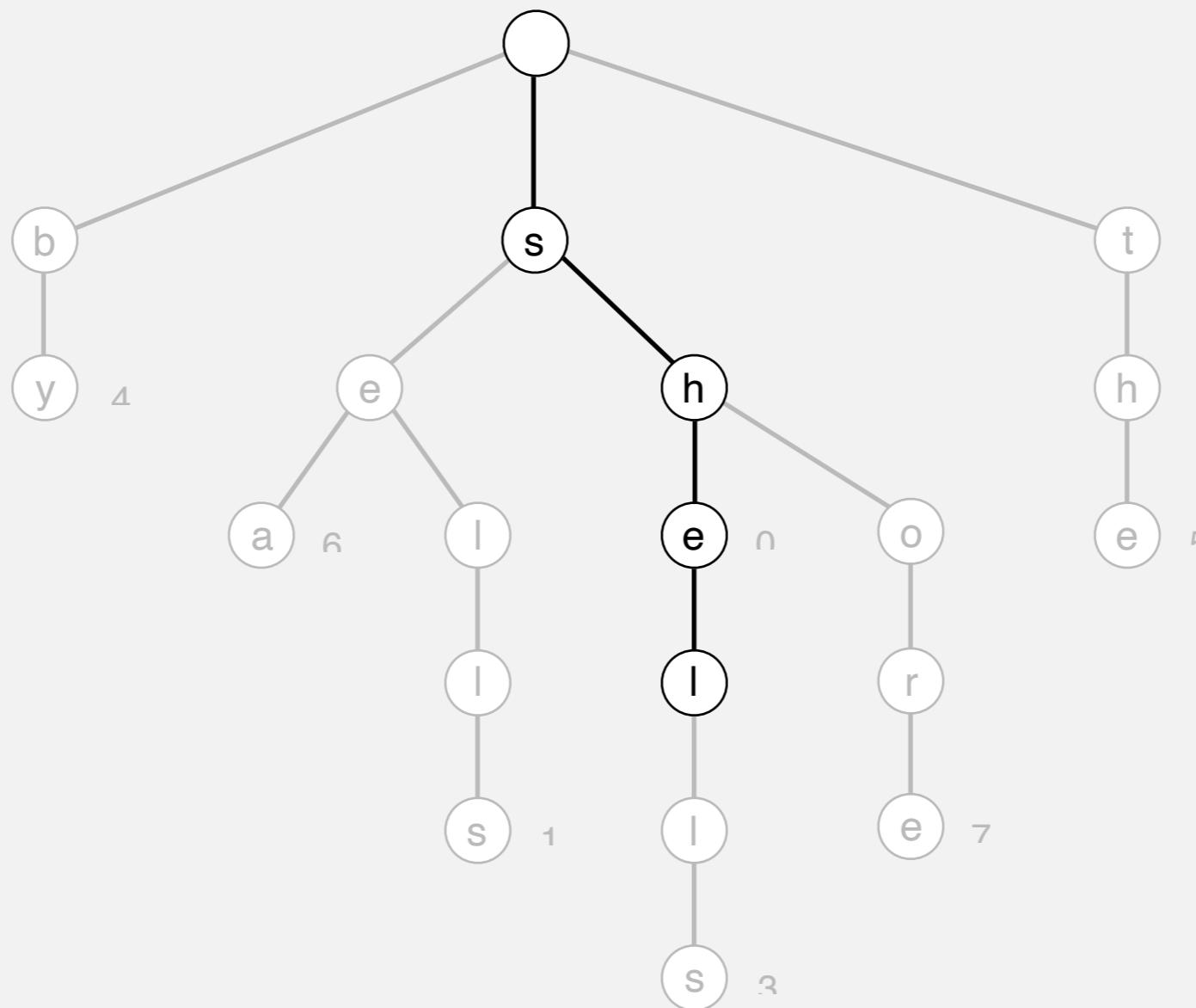


Search in a trie

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("shells")

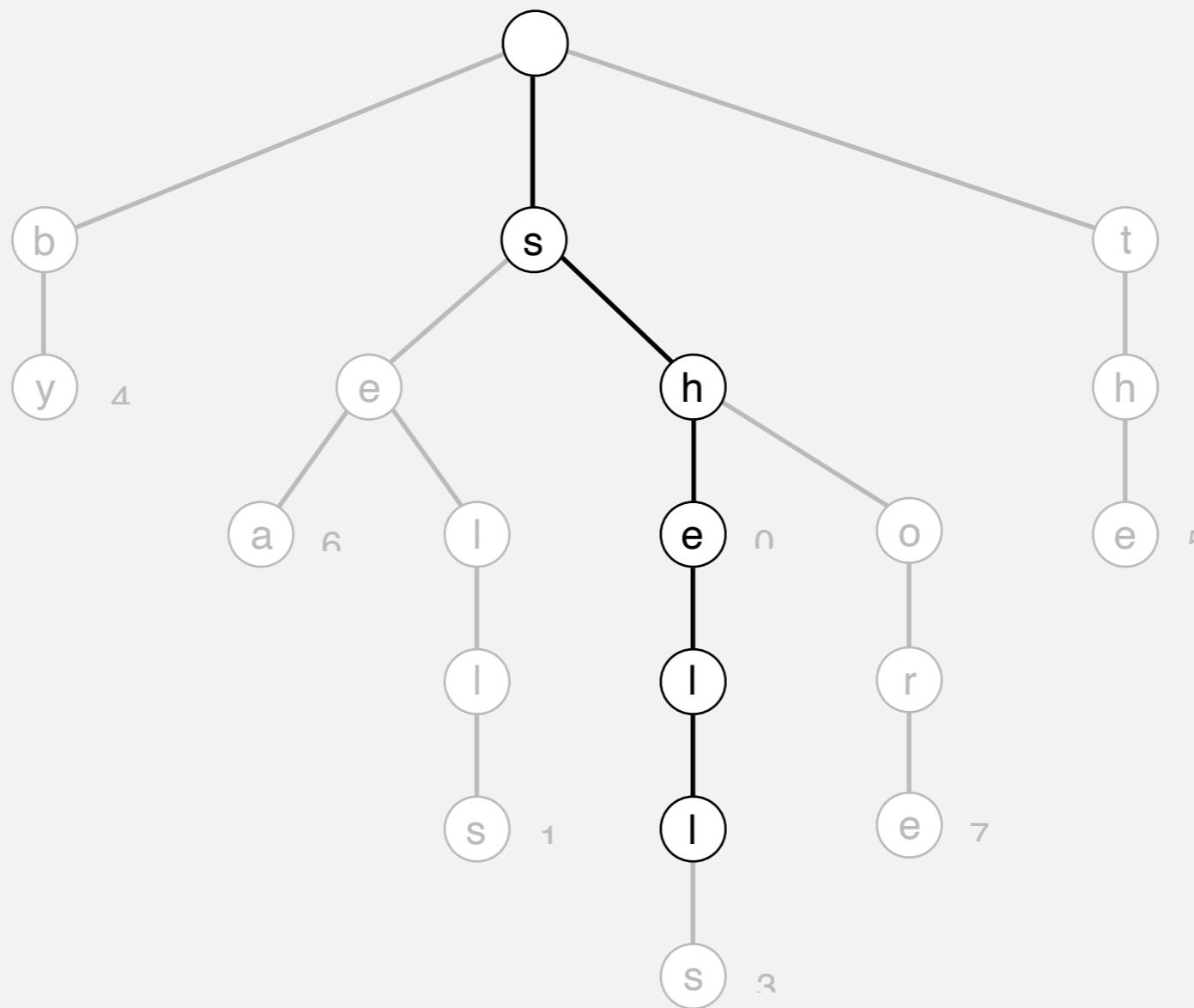


Search in a trie

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("shells")

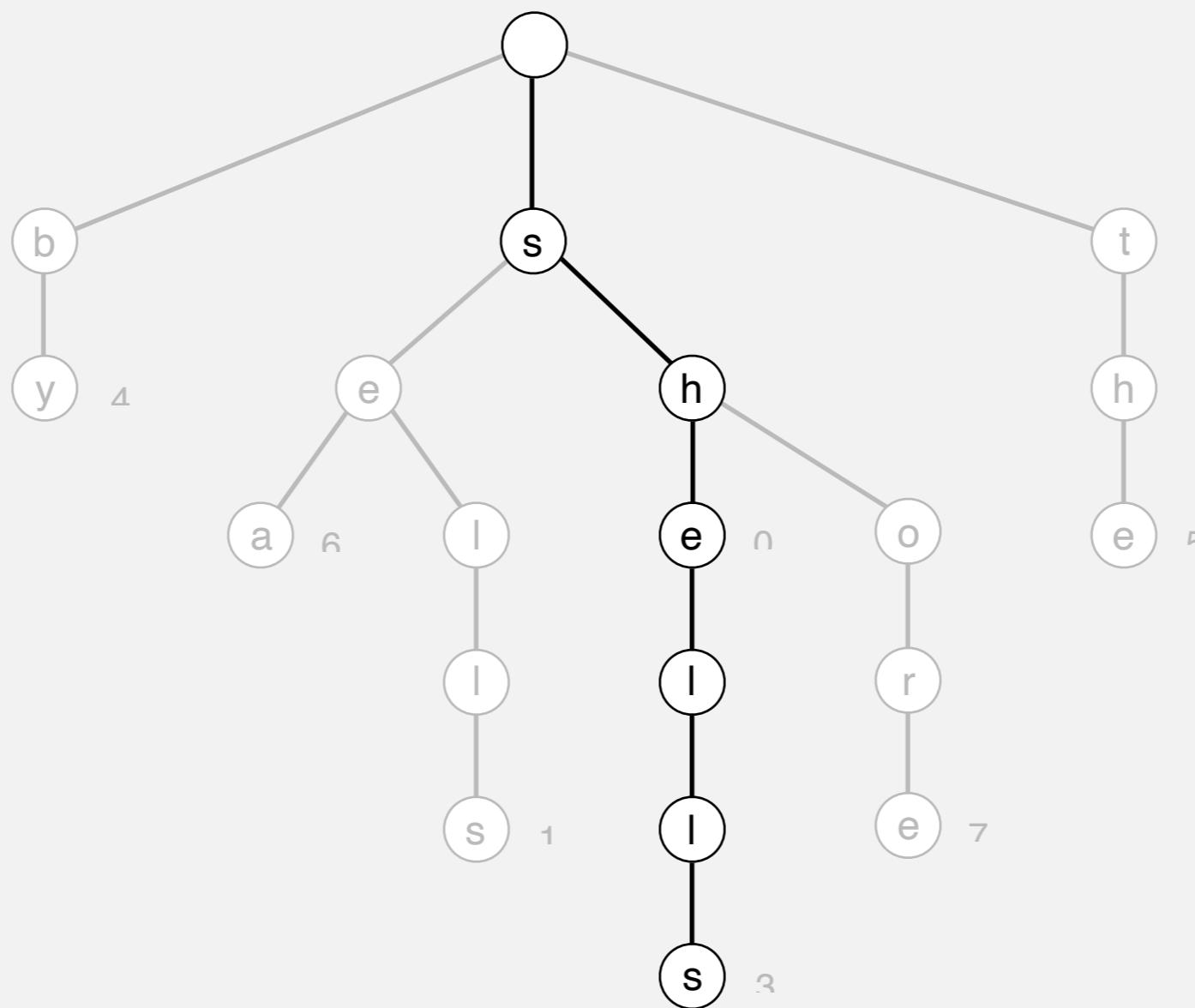


Search in a trie

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("shells")

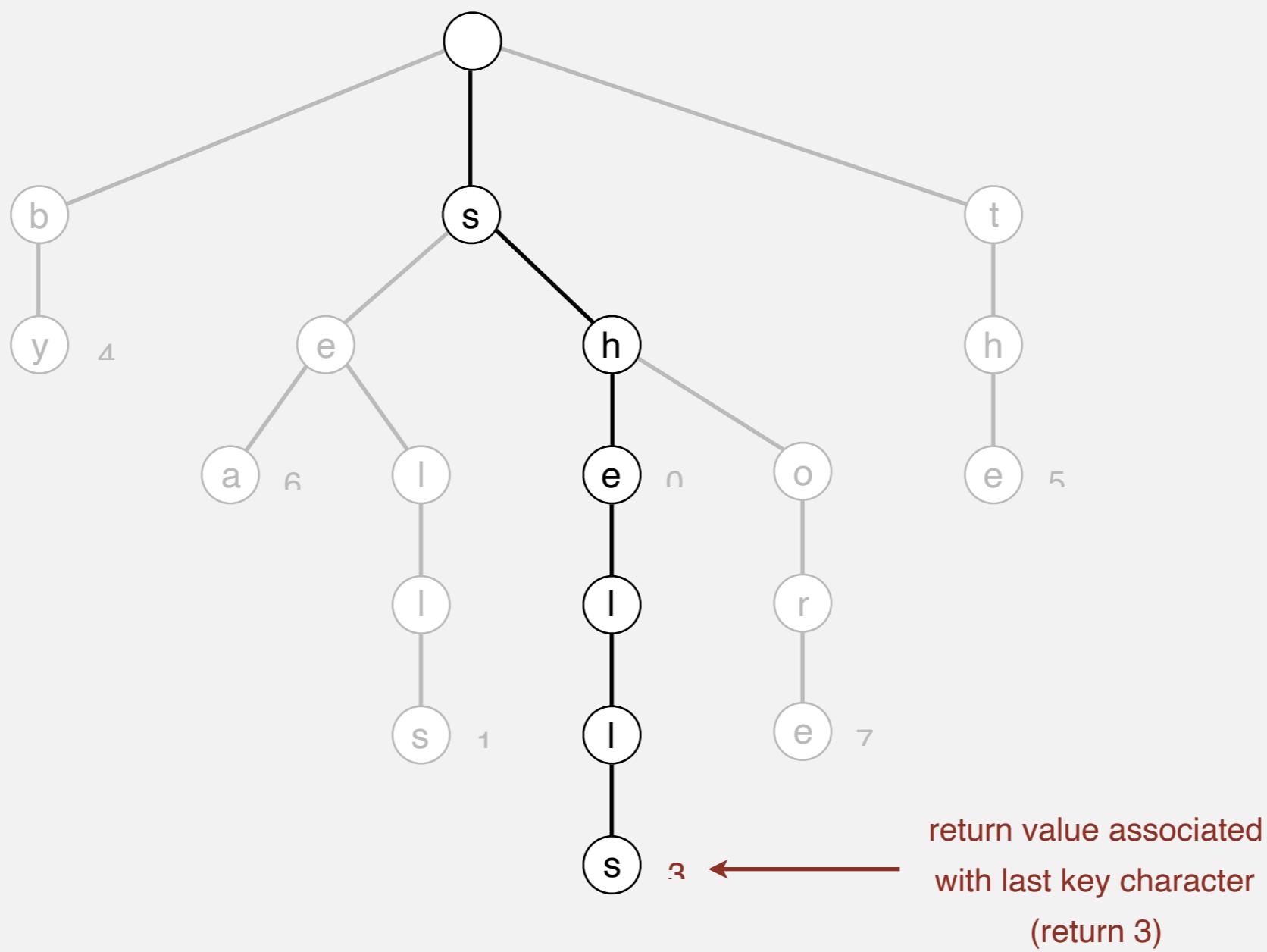


Search in a trie

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("shells")

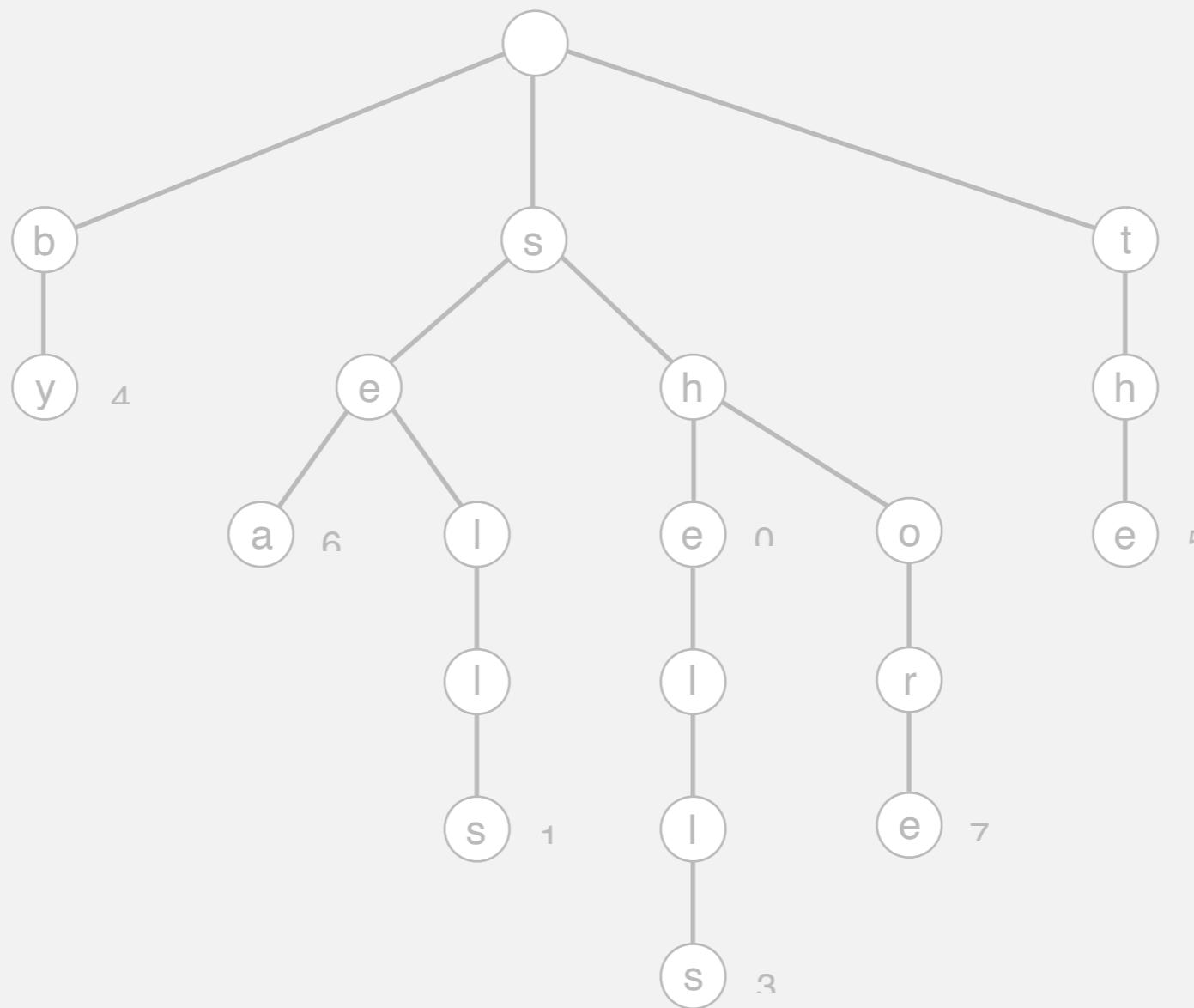


Search in a trie

Follow links corresponding to each character in the key.

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

get("she")

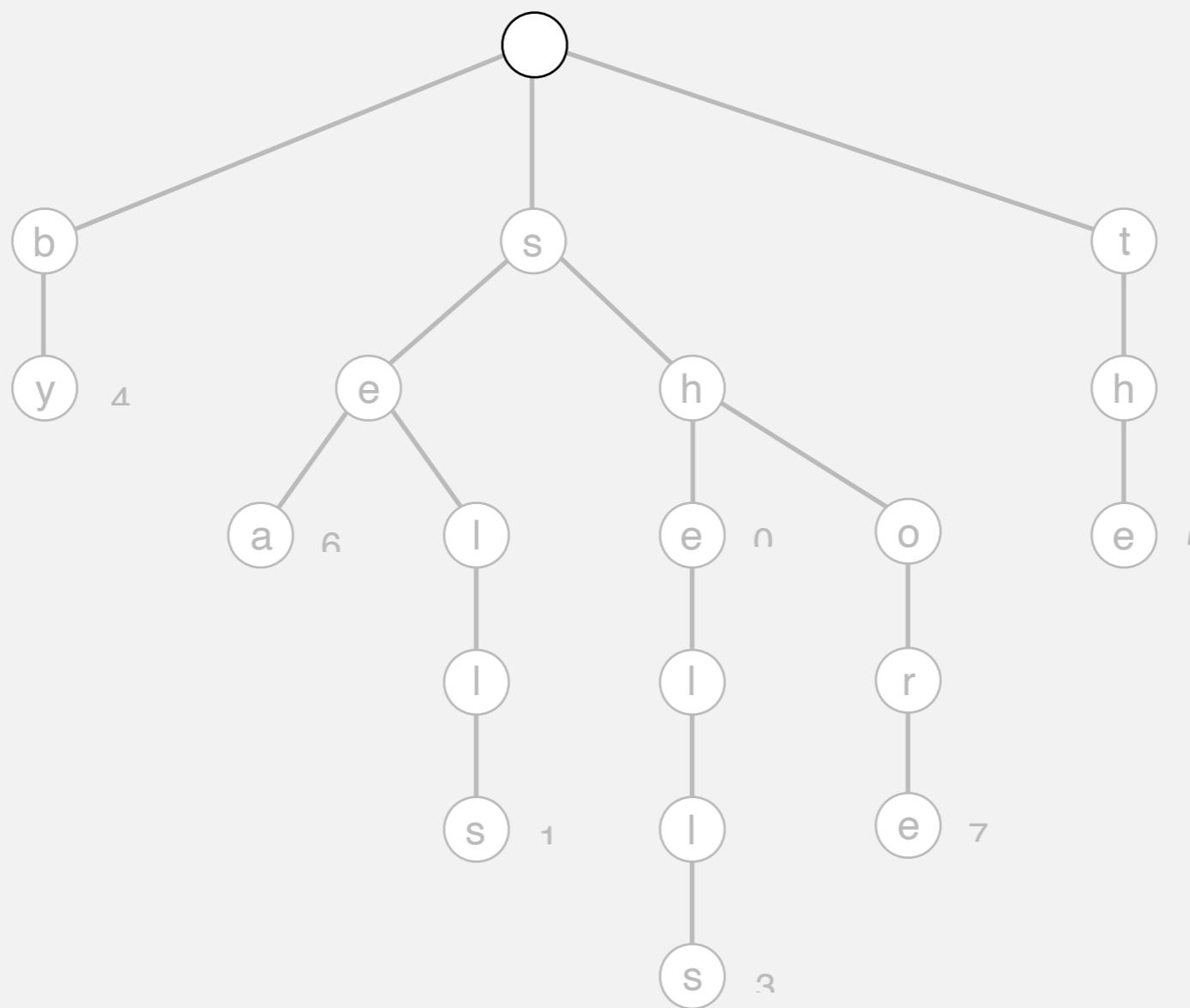


Search in a trie

Follow links corresponding to each character in the key.

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

get("she")

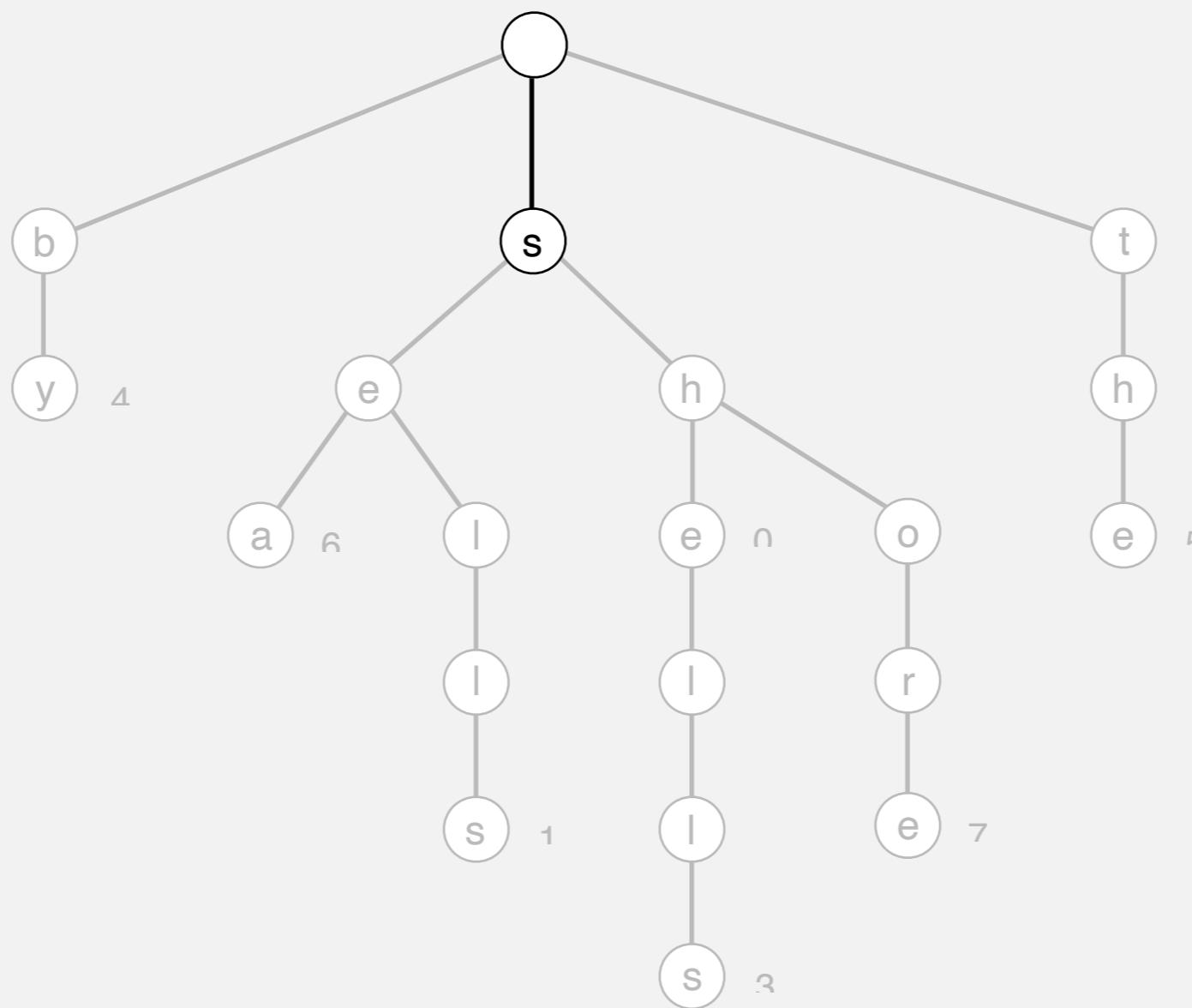


Search in a trie

Follow links corresponding to each character in the key.

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

get("she")

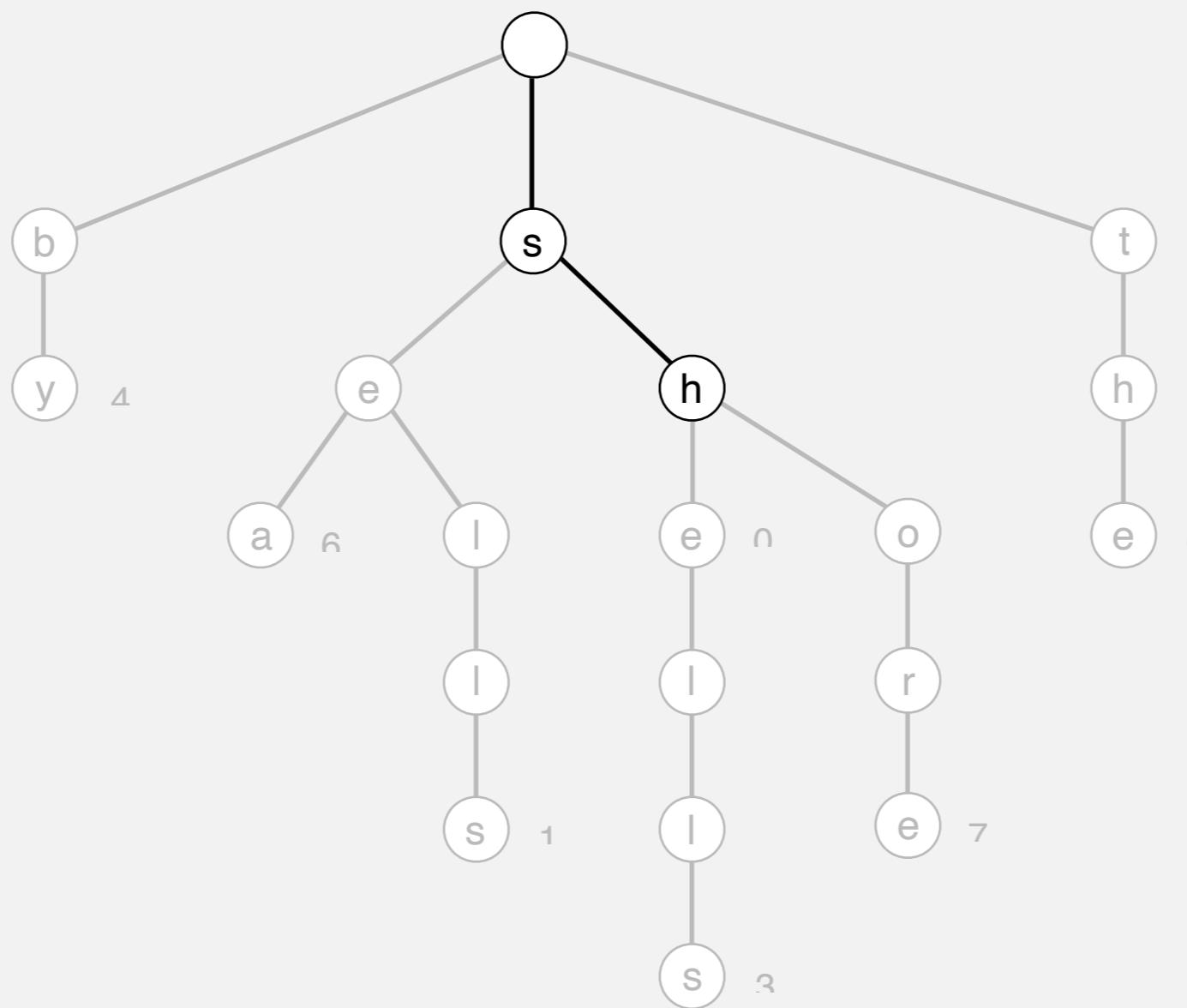


Search in a trie

Follow links corresponding to each character in the key.

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

get("she")

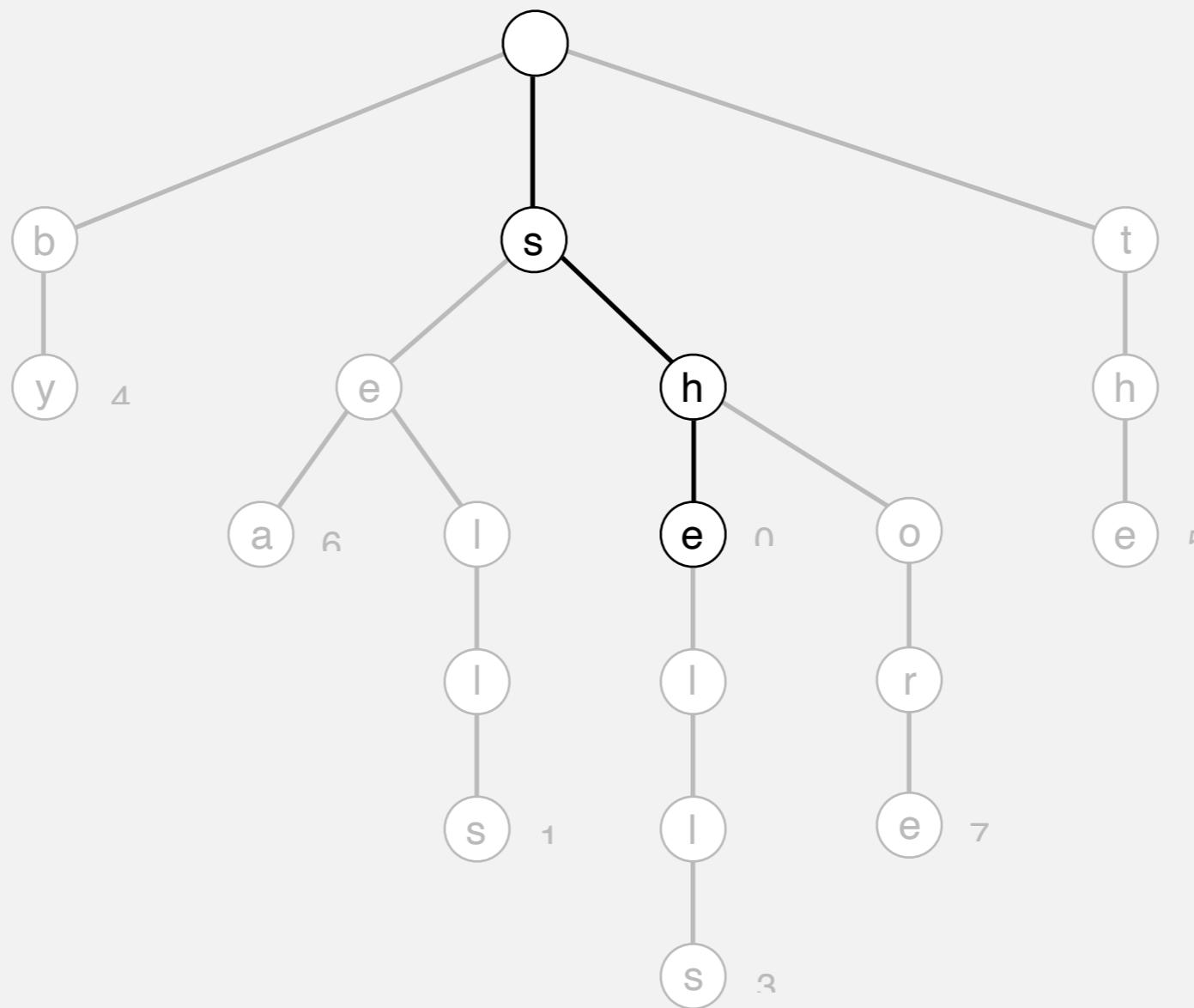


Search in a trie

Follow links corresponding to each character in the key.

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

get("she")

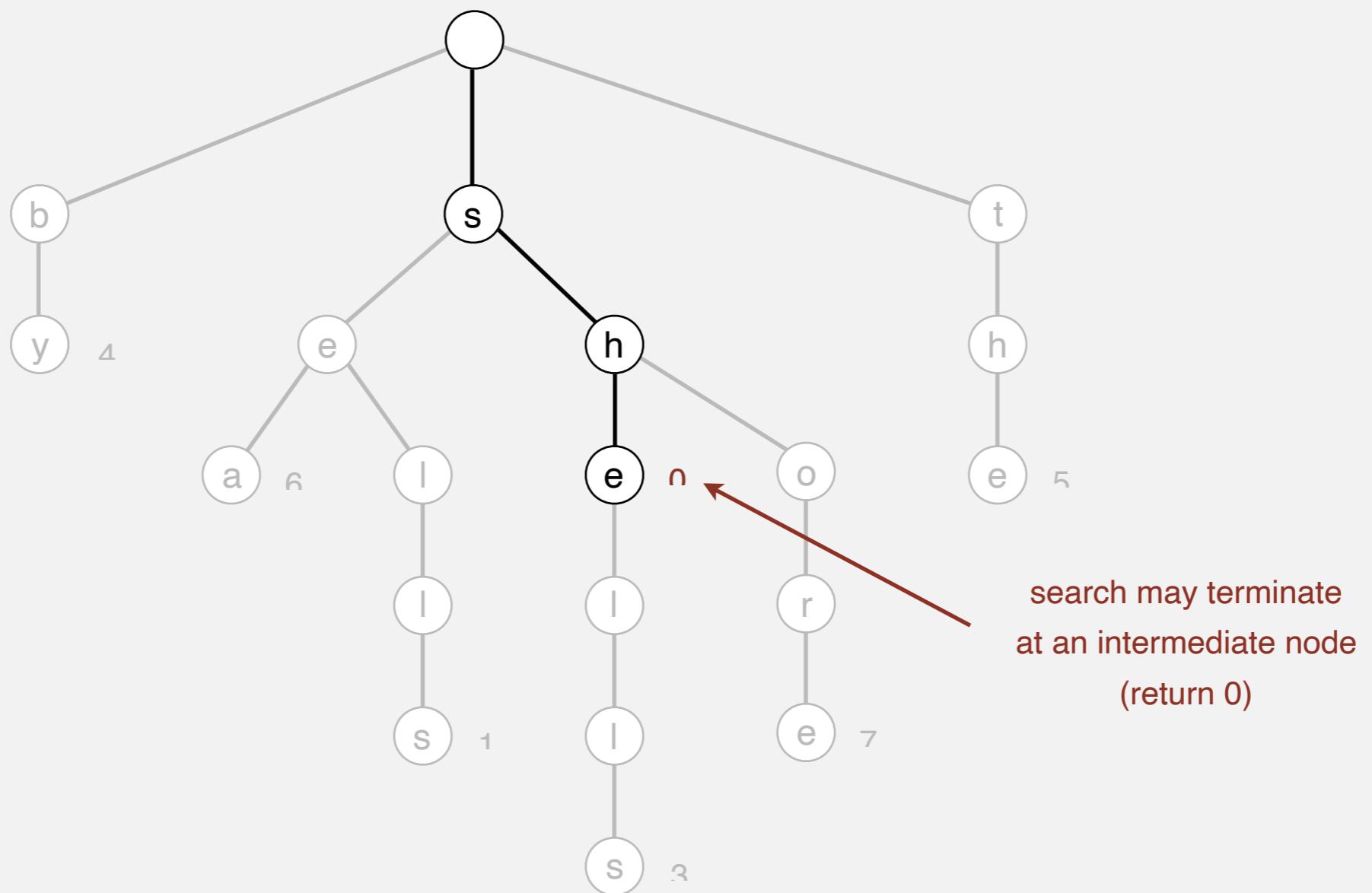


Search in a trie

Follow links corresponding to each character in the key.

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

get("she")

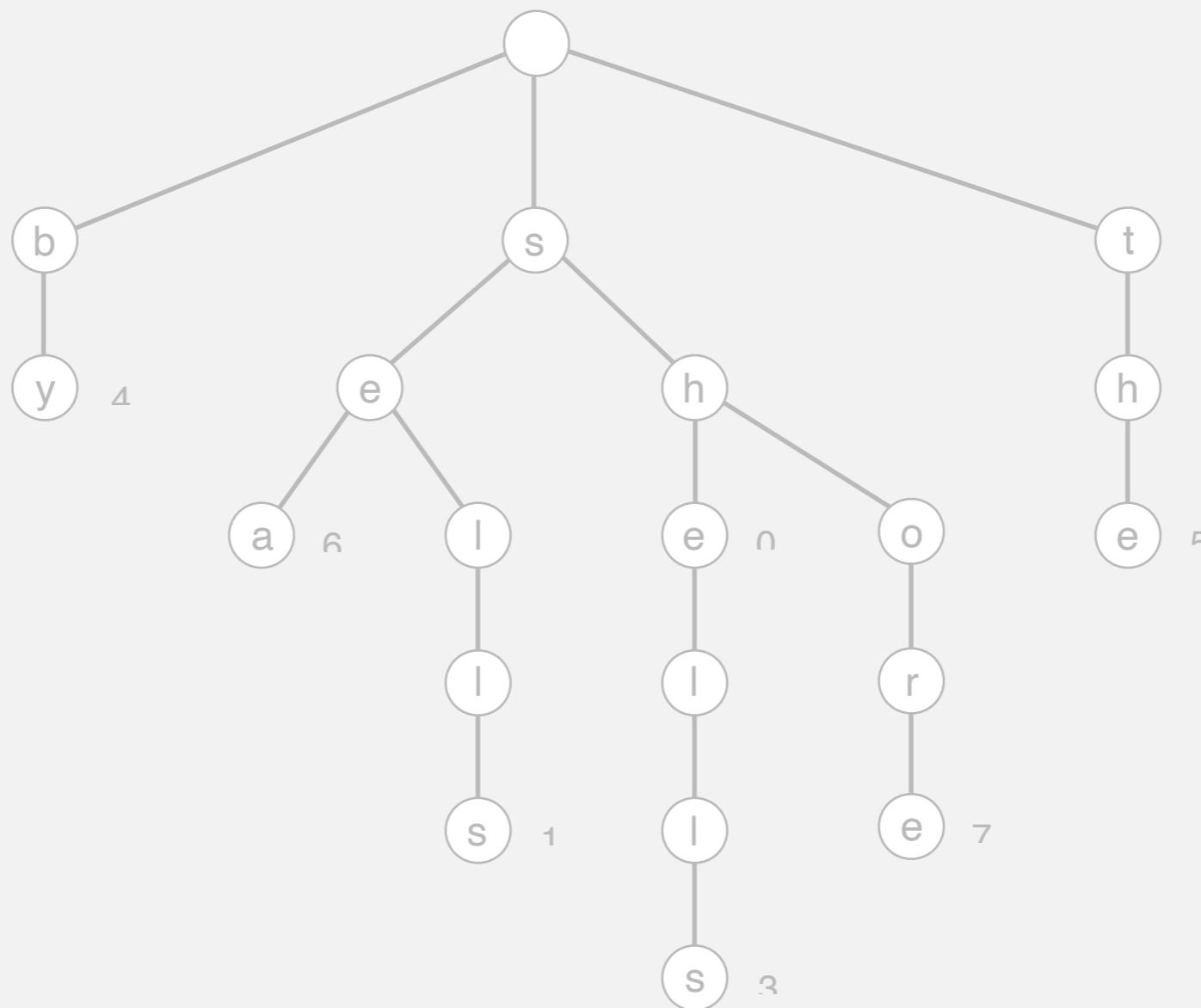


Search in a trie

Follow links corresponding to each character in the key.

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

get("shell")

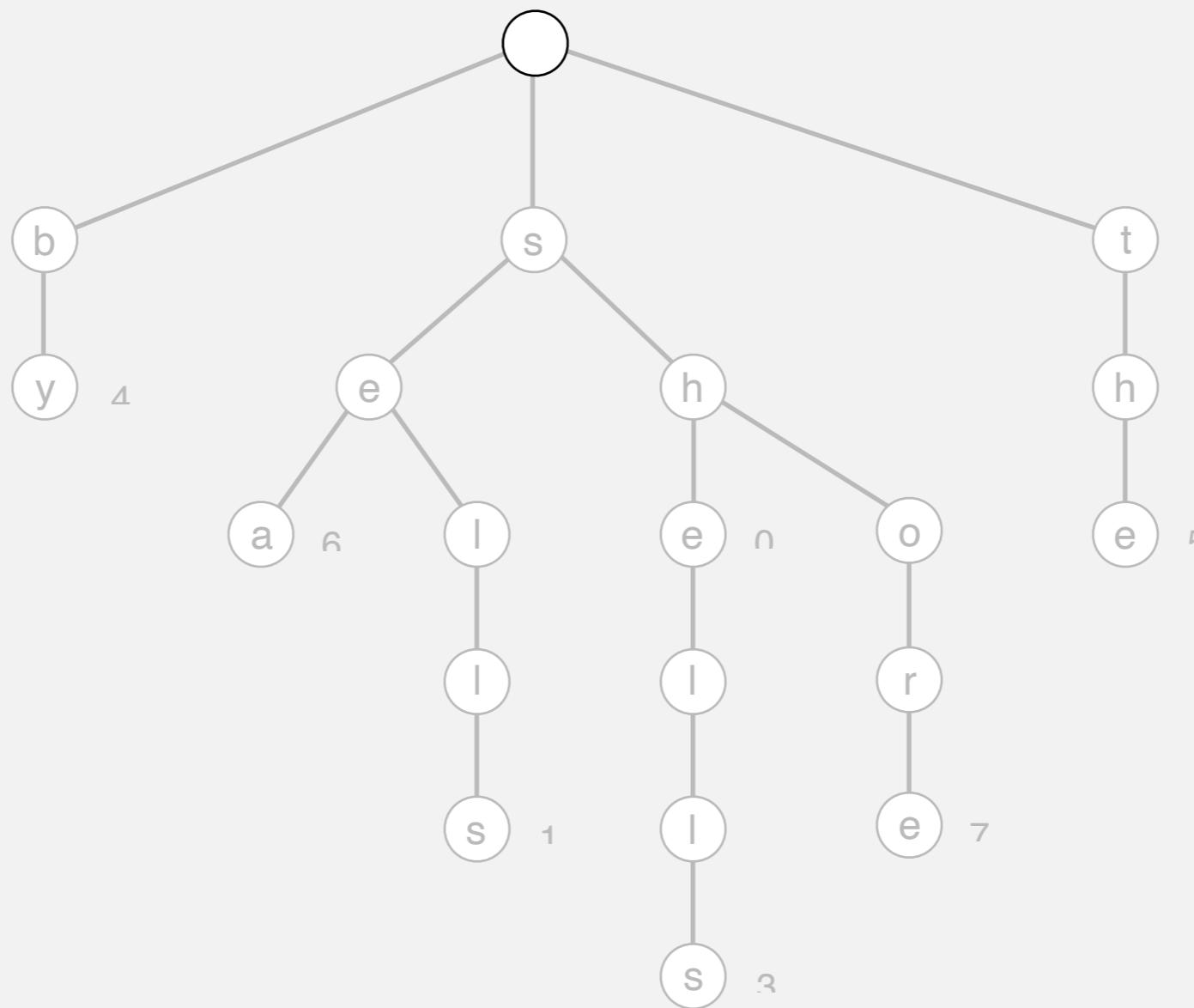


Search in a trie

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")

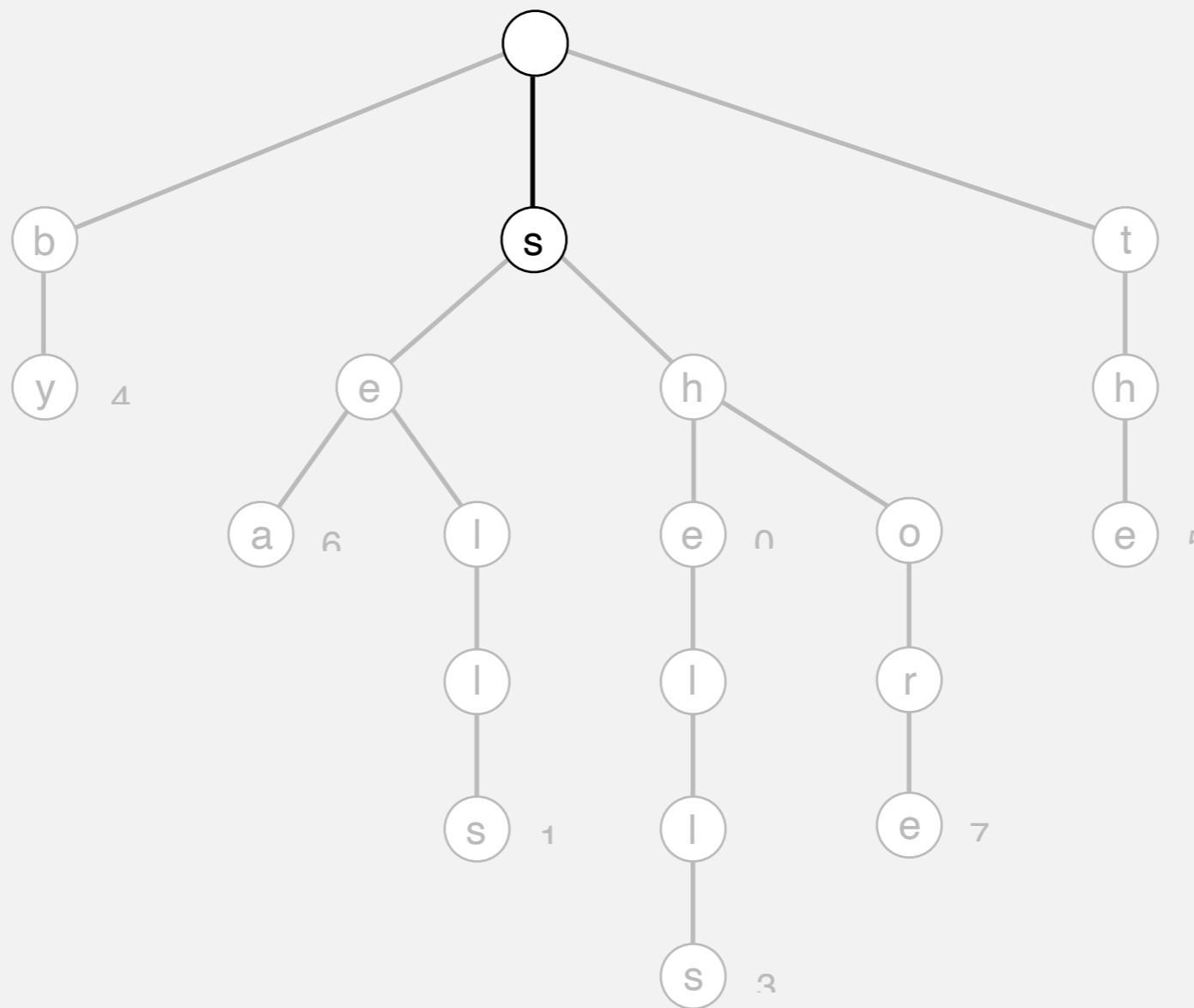


Search in a trie

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")

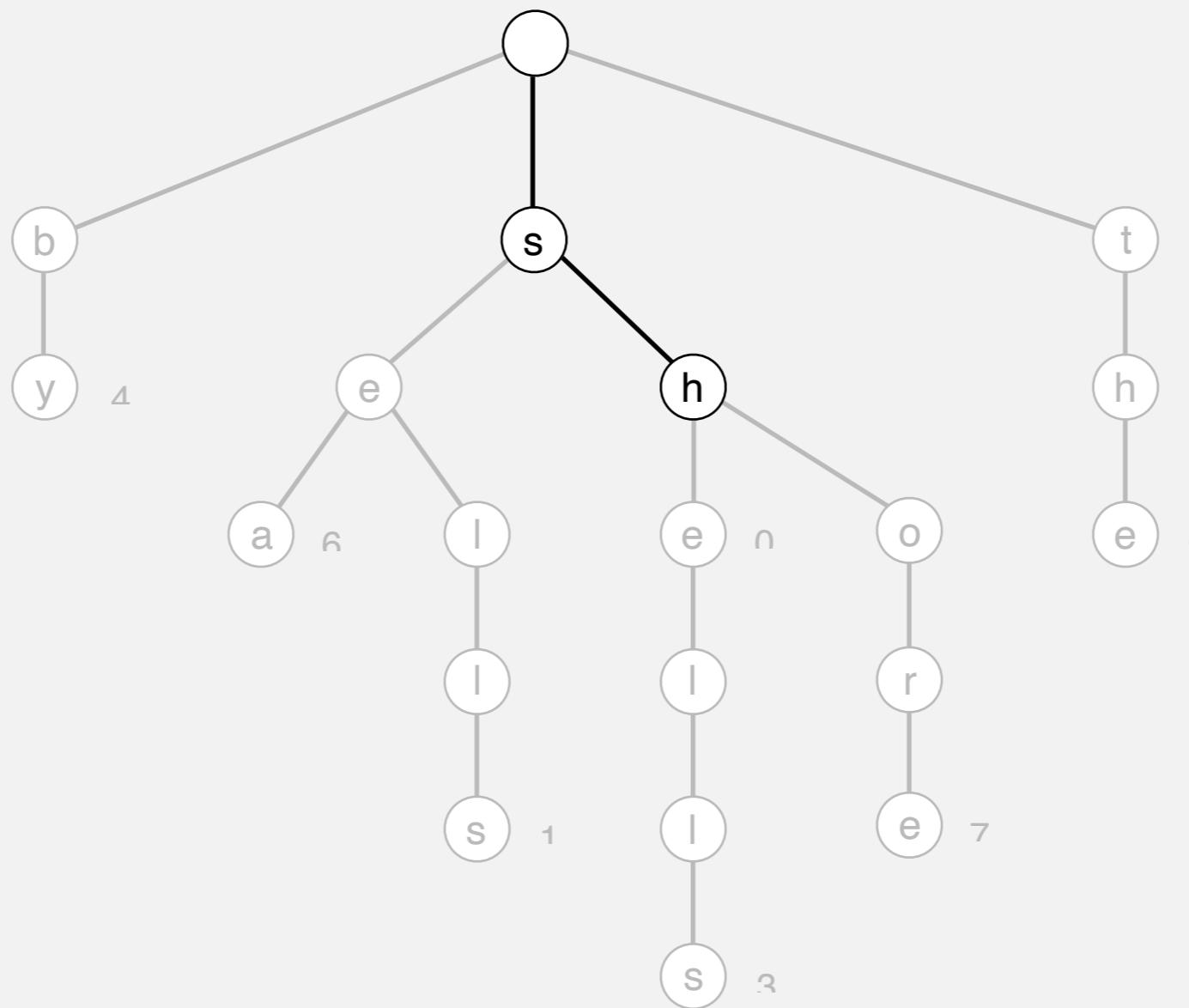


Search in a trie

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")

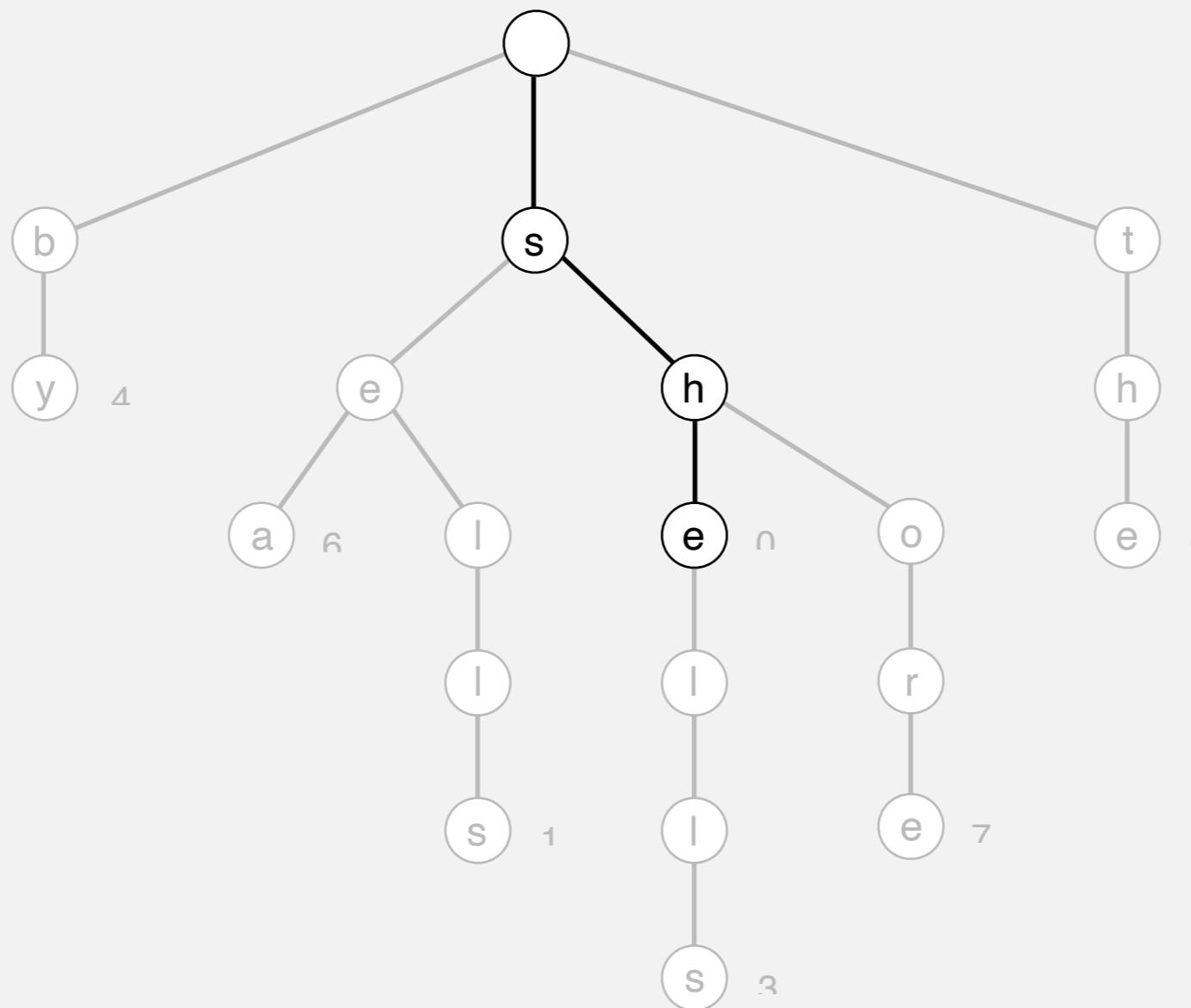


Search in a trie

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")

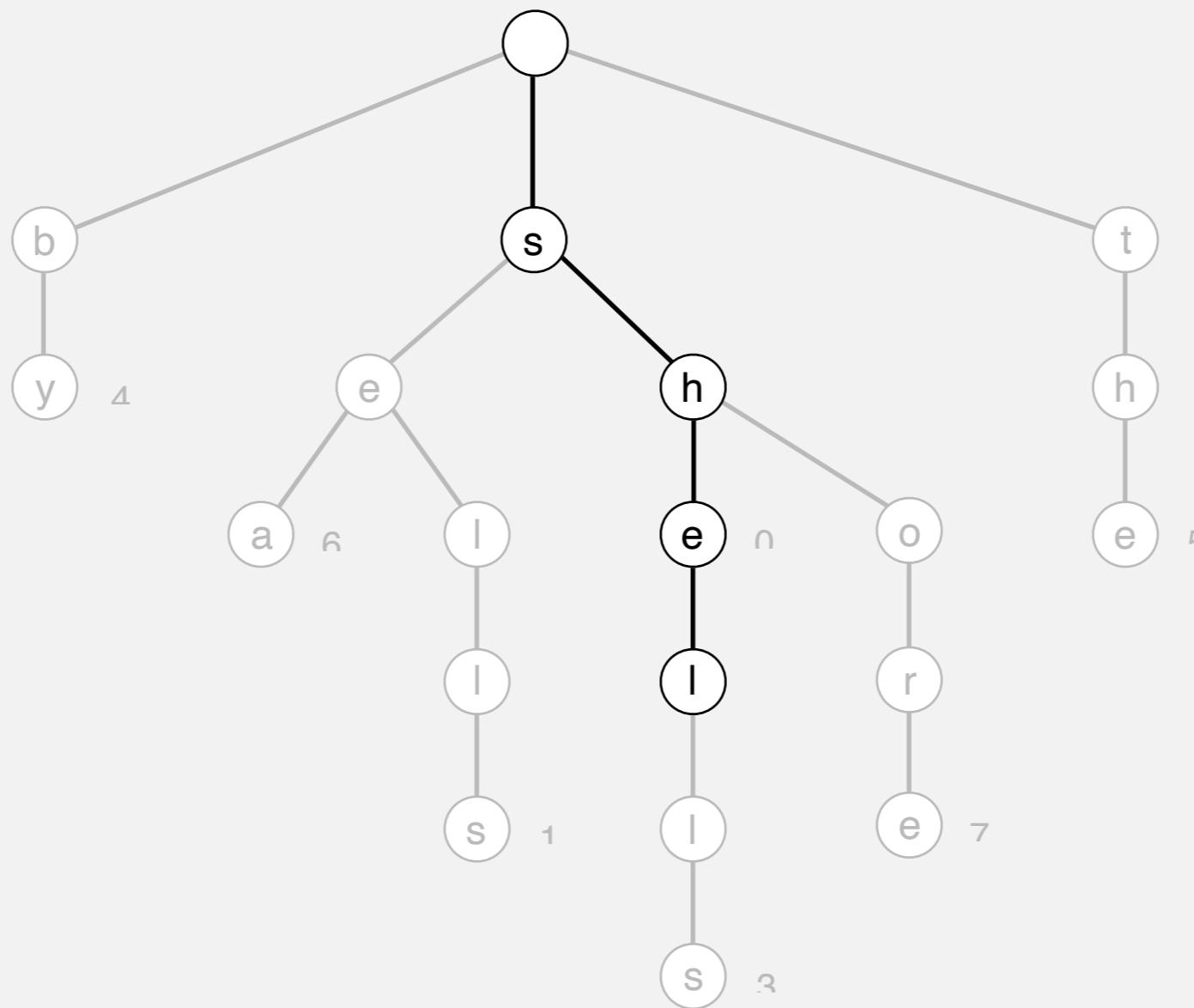


Search in a trie

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")

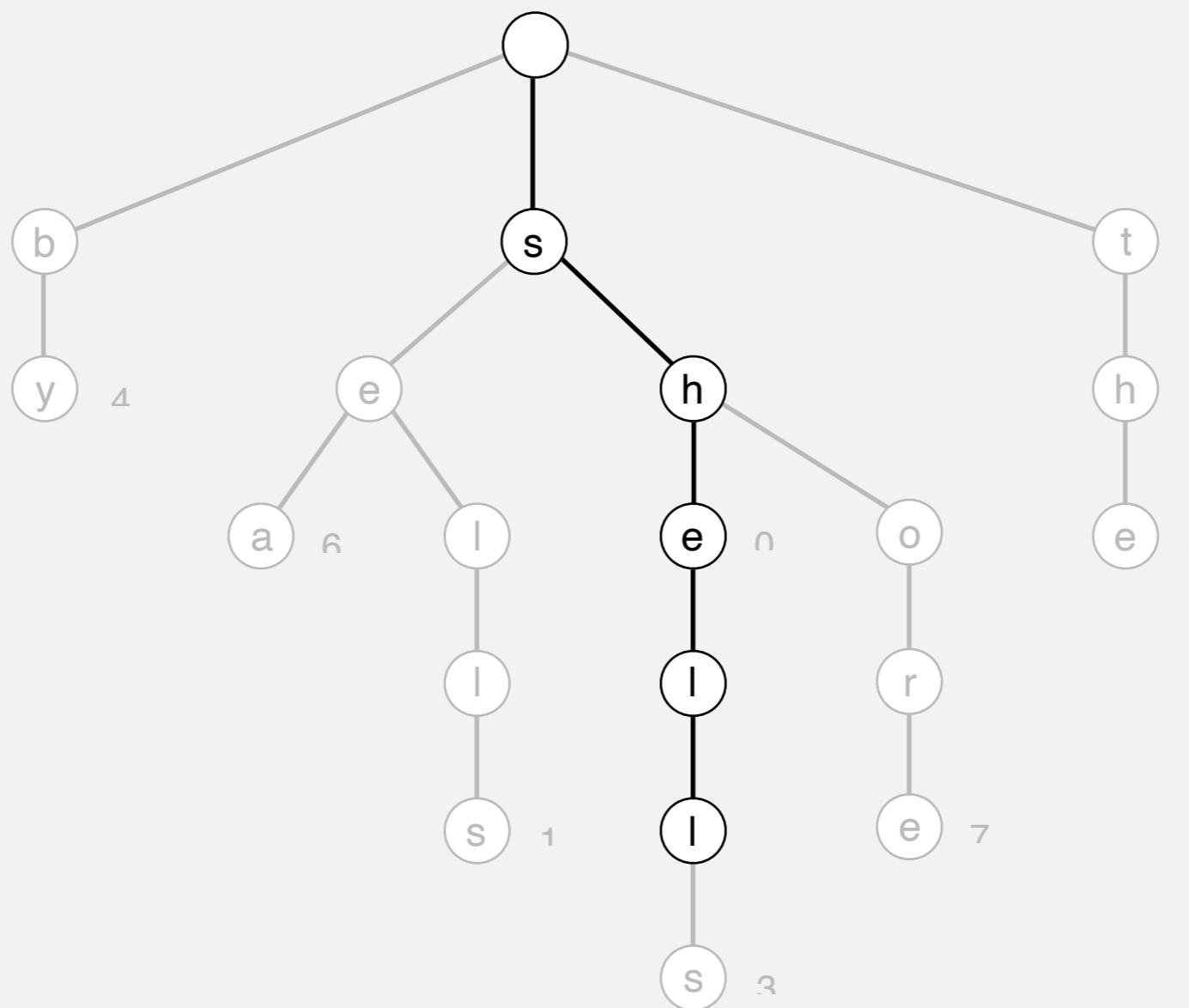


Search in a trie

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")

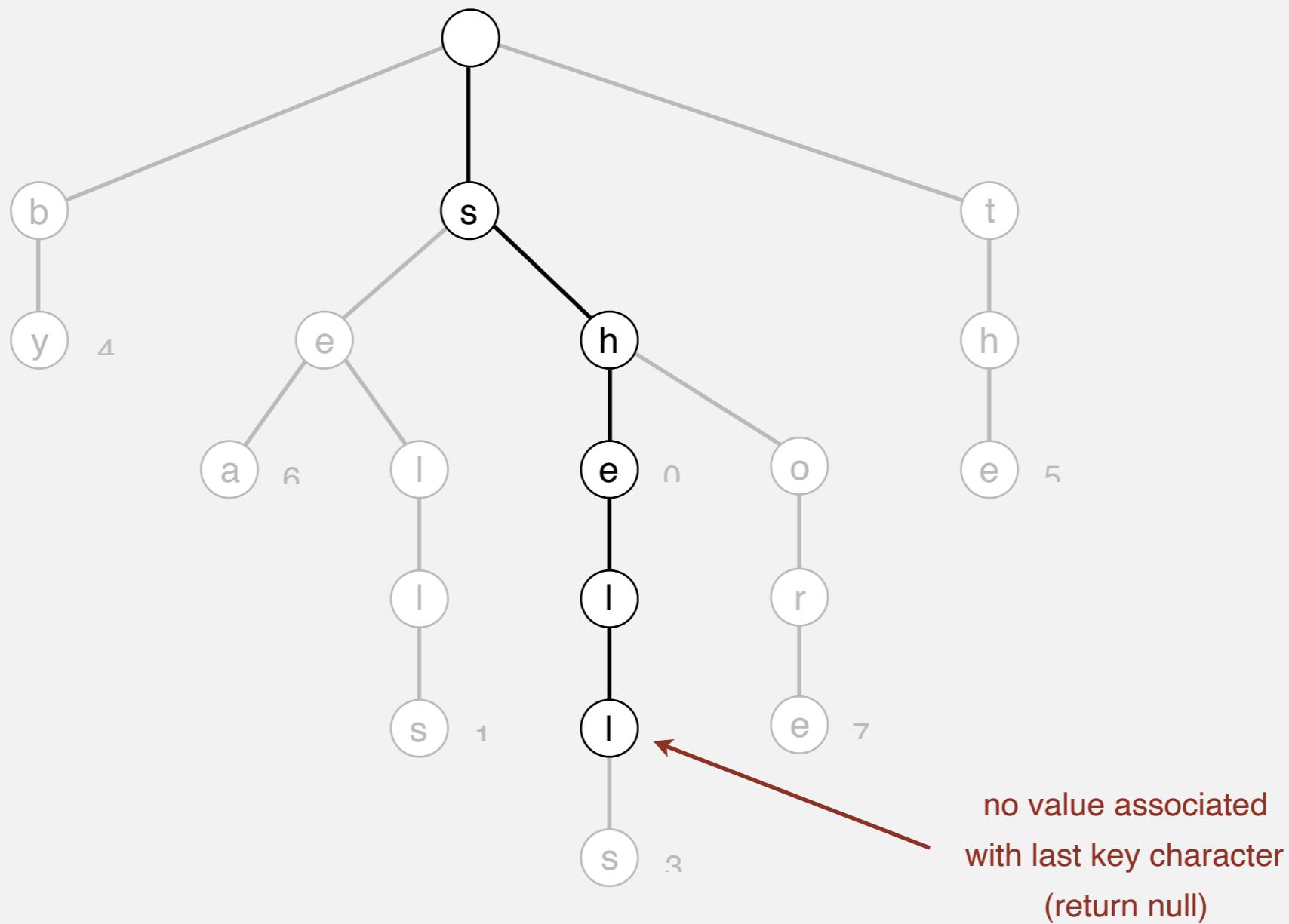


Search in a trie

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")

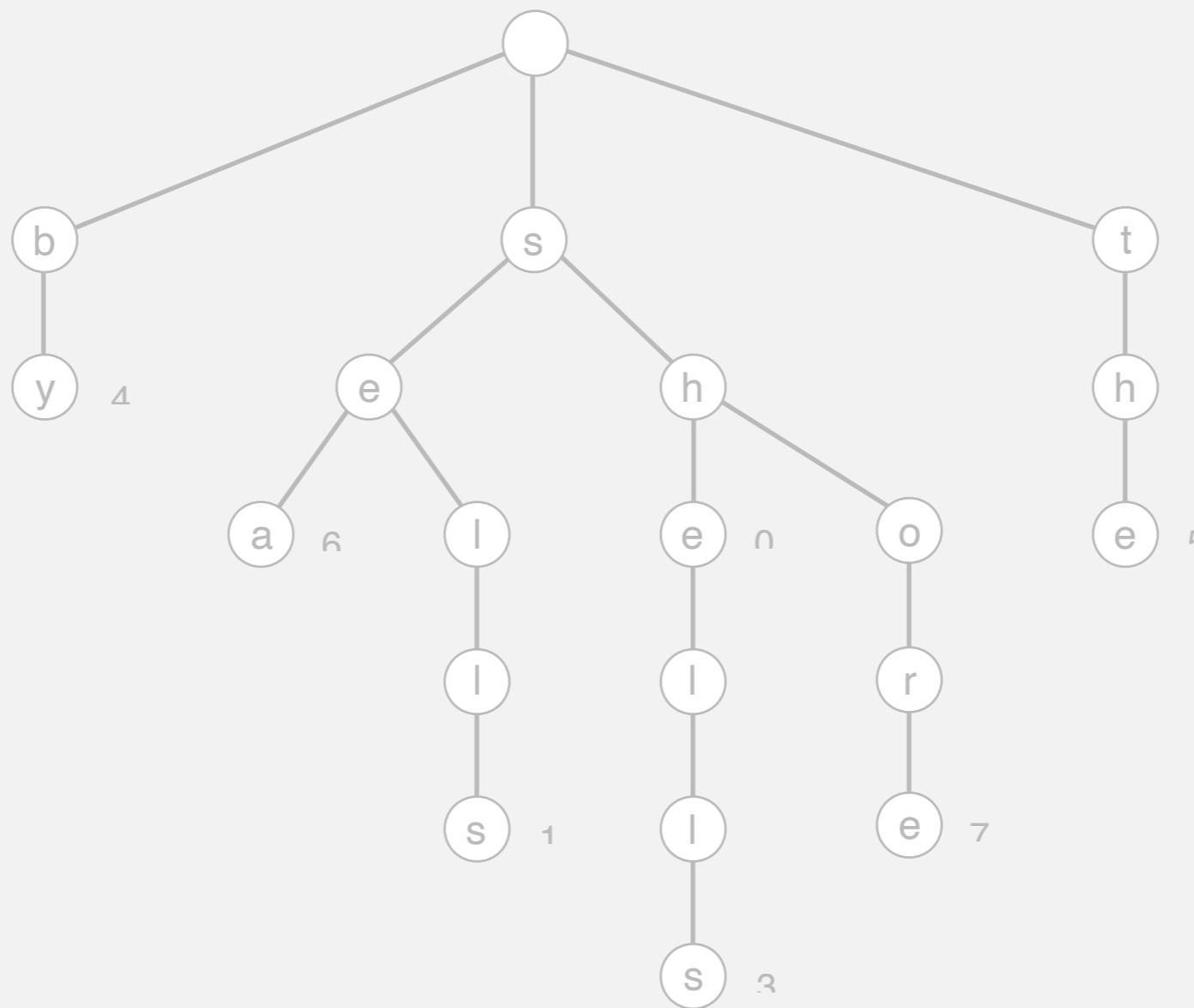


Search in a trie

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")

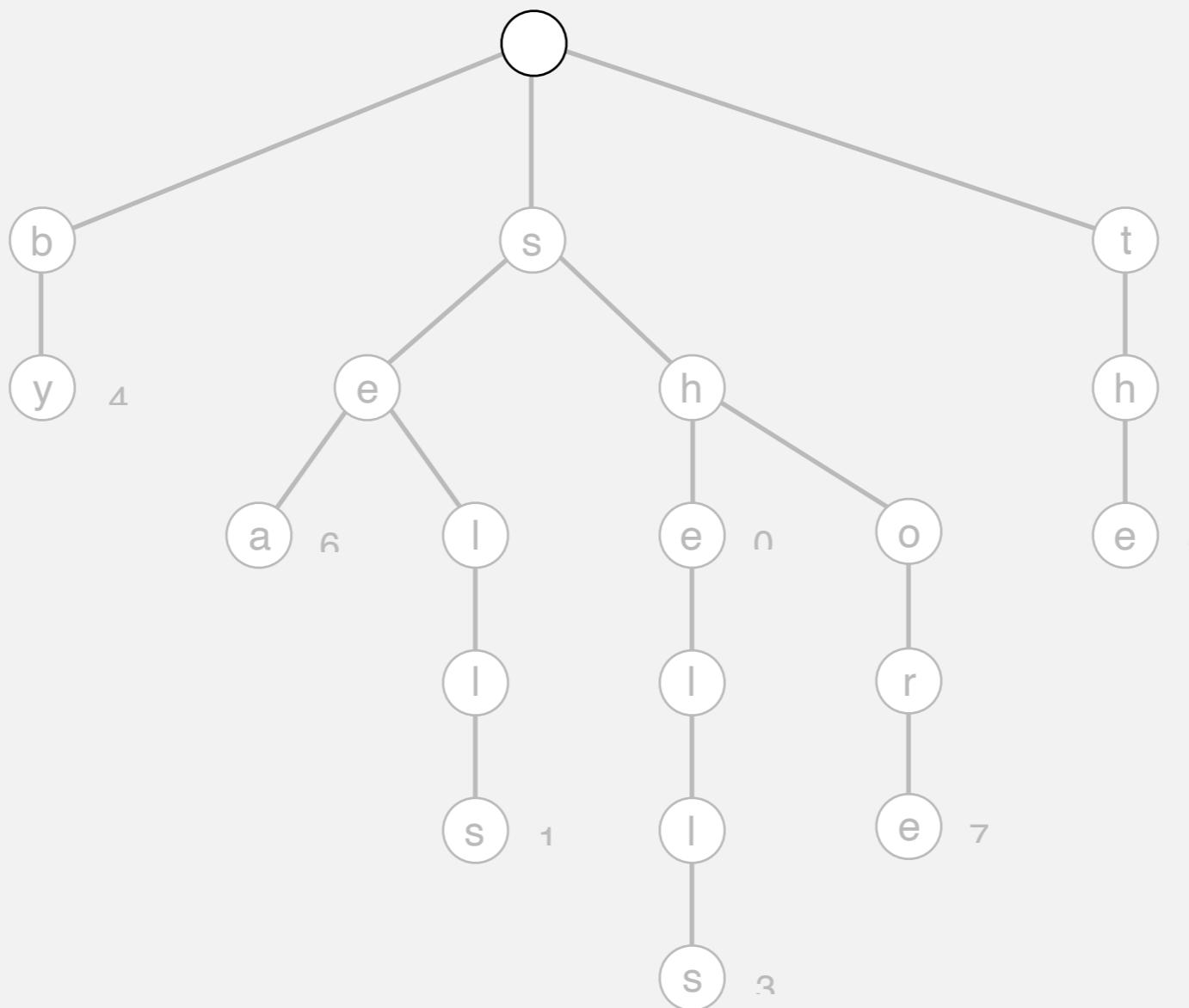


Search in a trie

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")

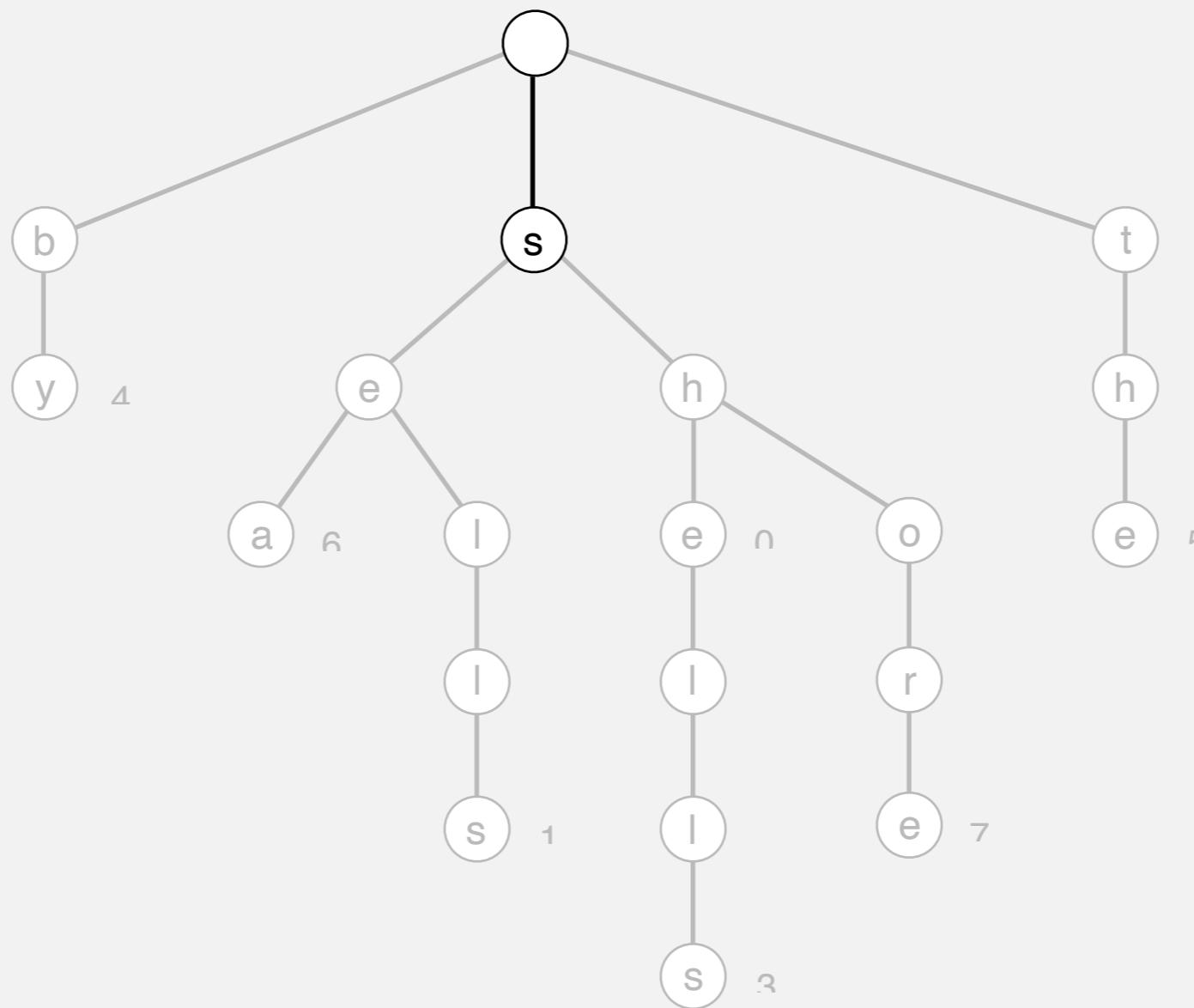


Search in a trie

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")

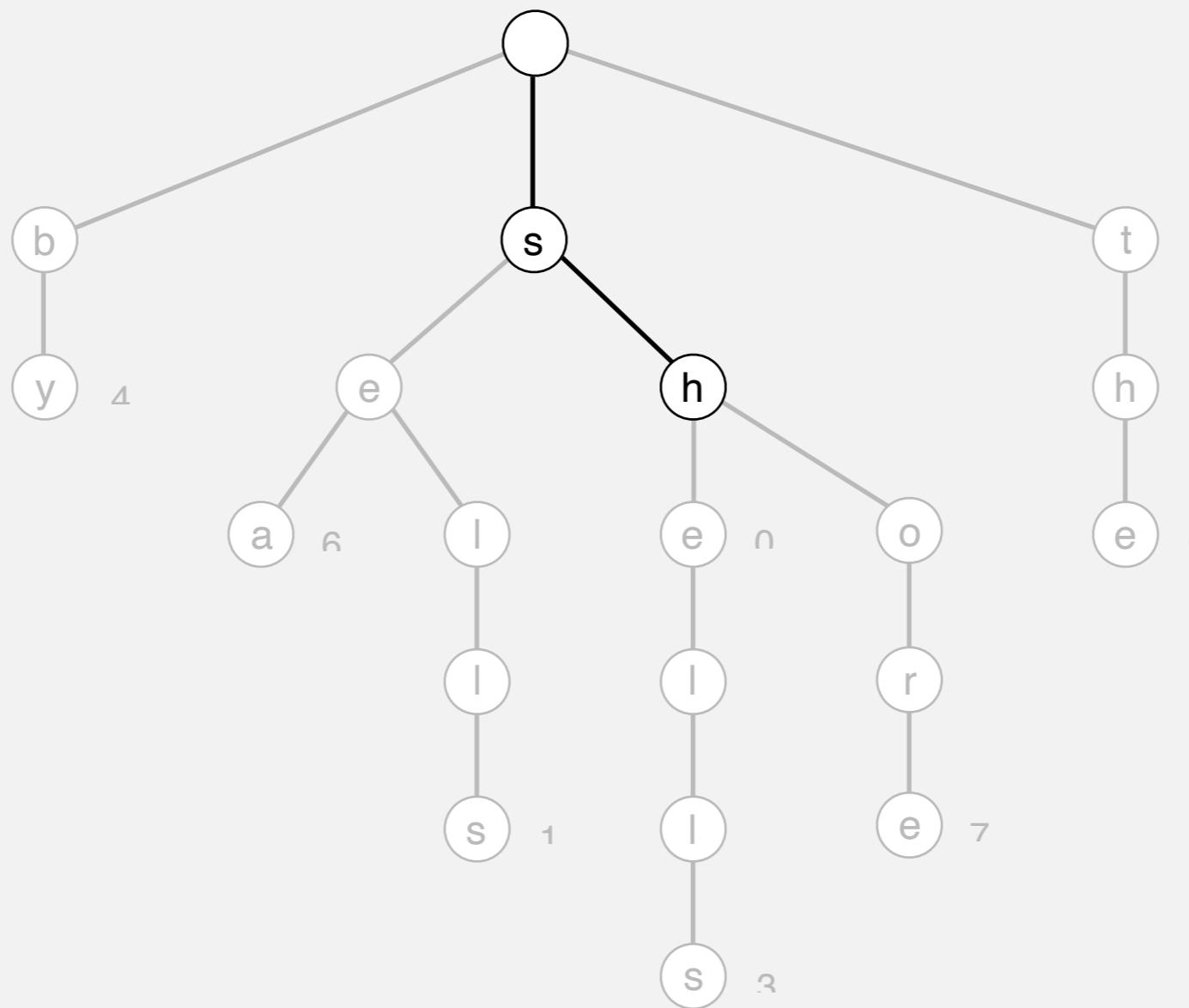


Search in a trie

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")

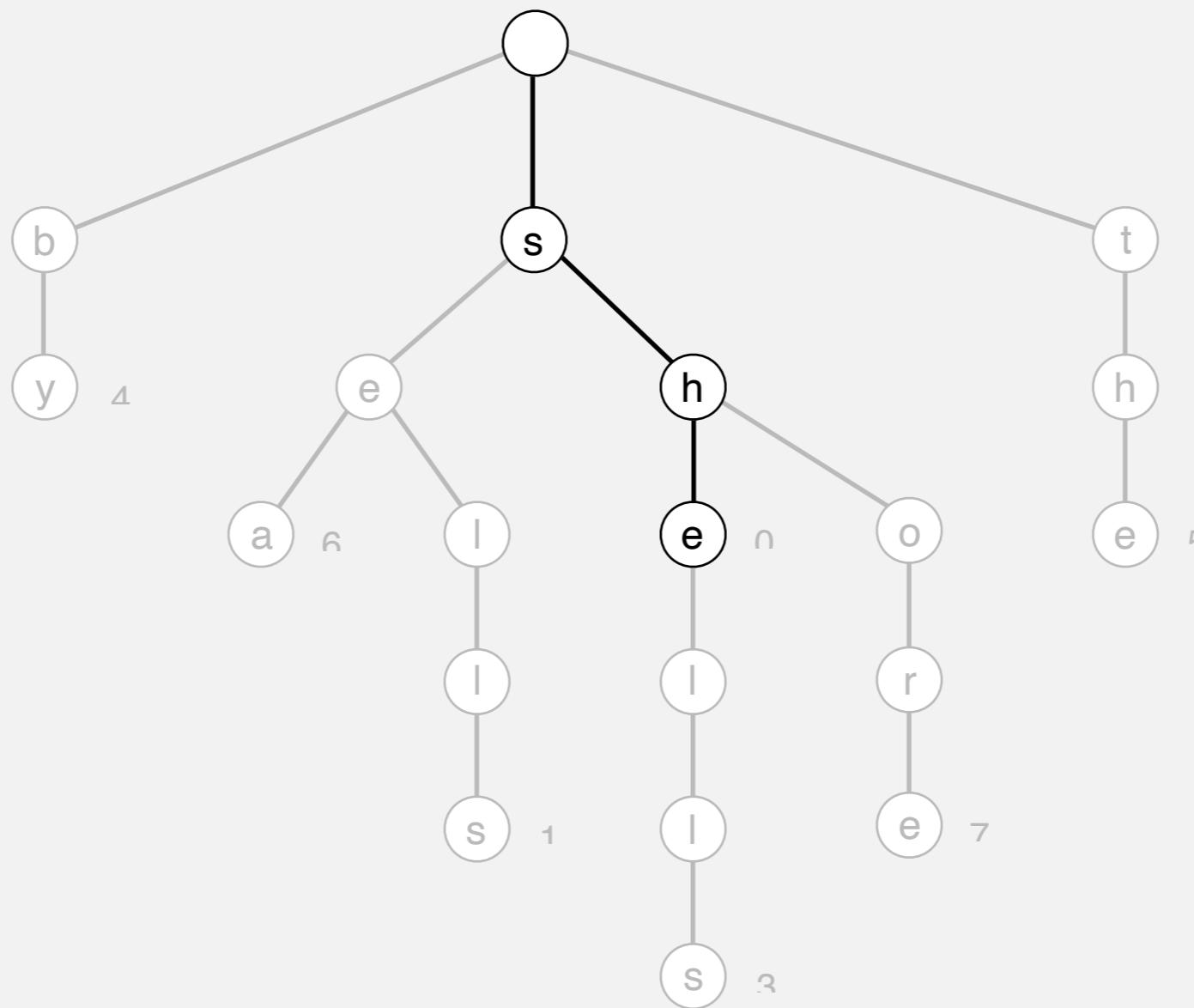


Search in a trie

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")

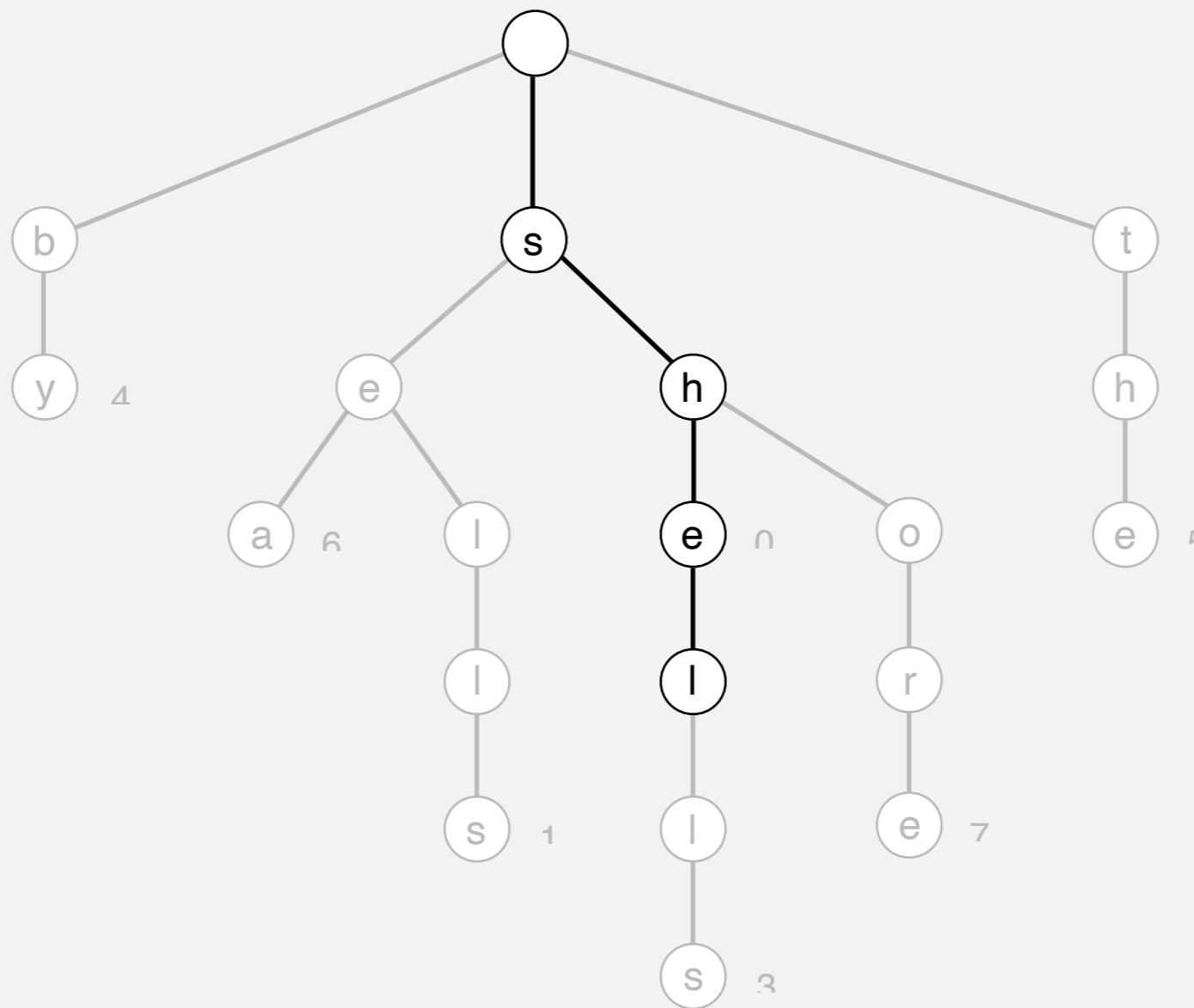


Search in a trie

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.

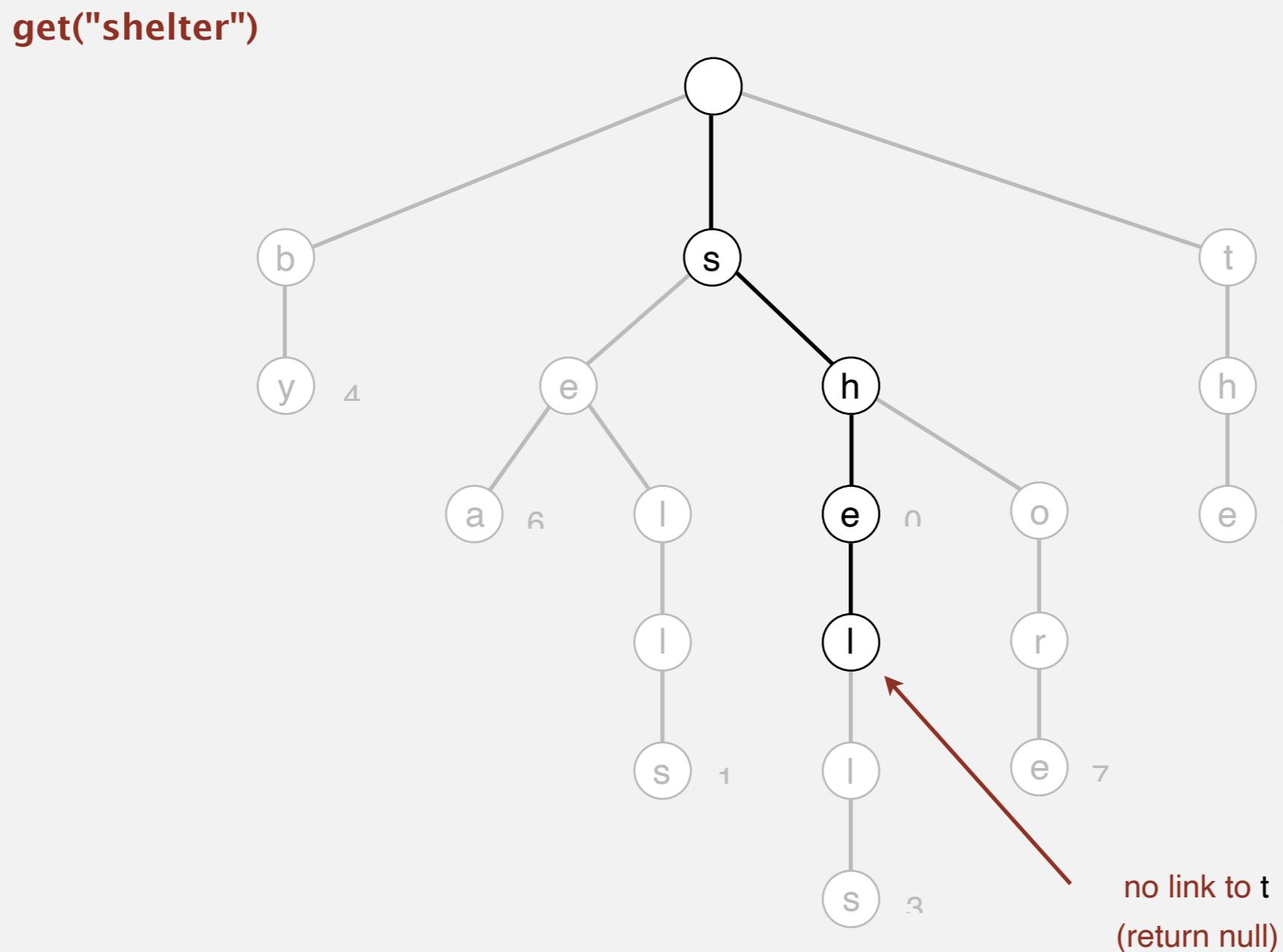
get("shelter")



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.

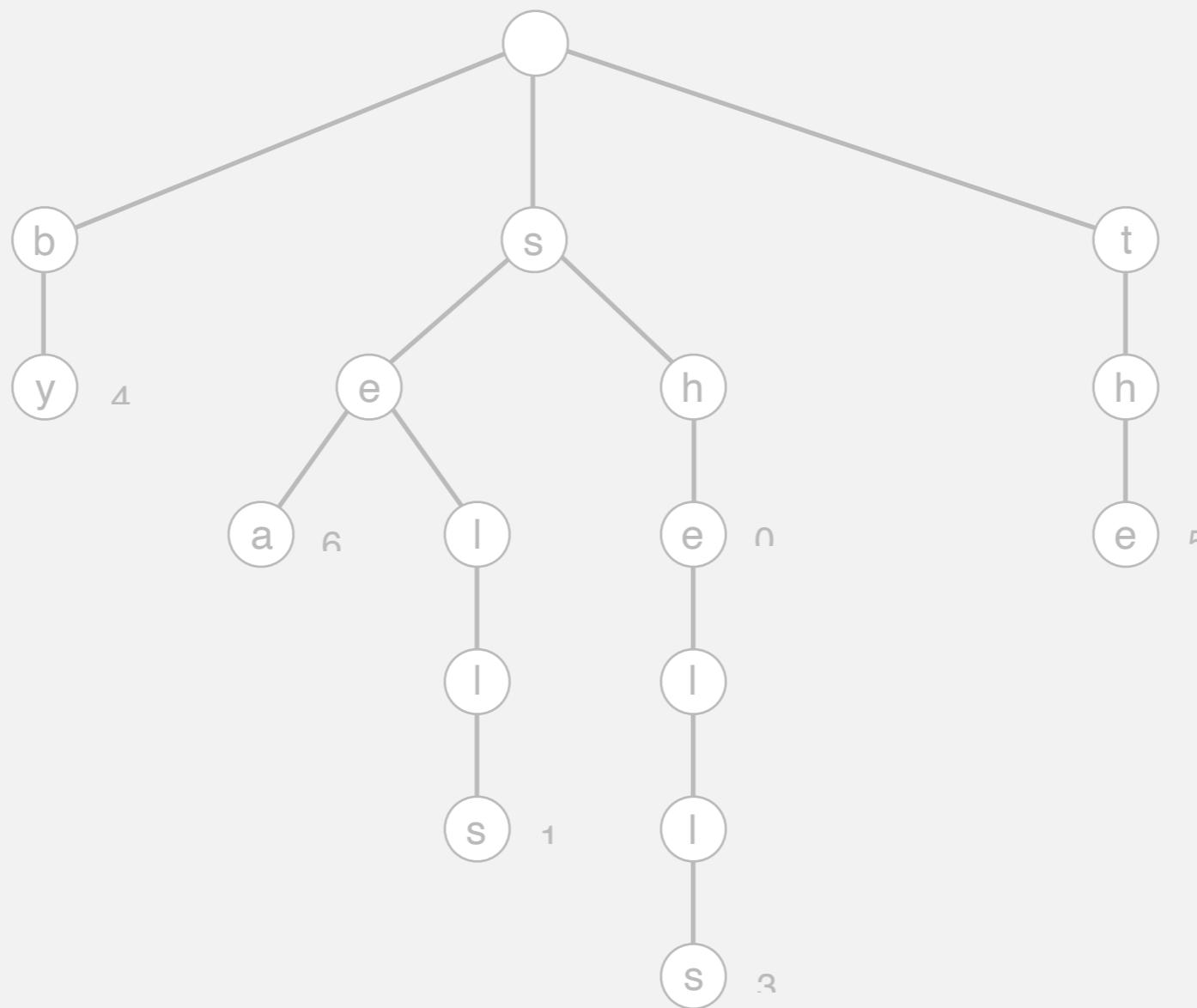


Insertion into a trie

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)

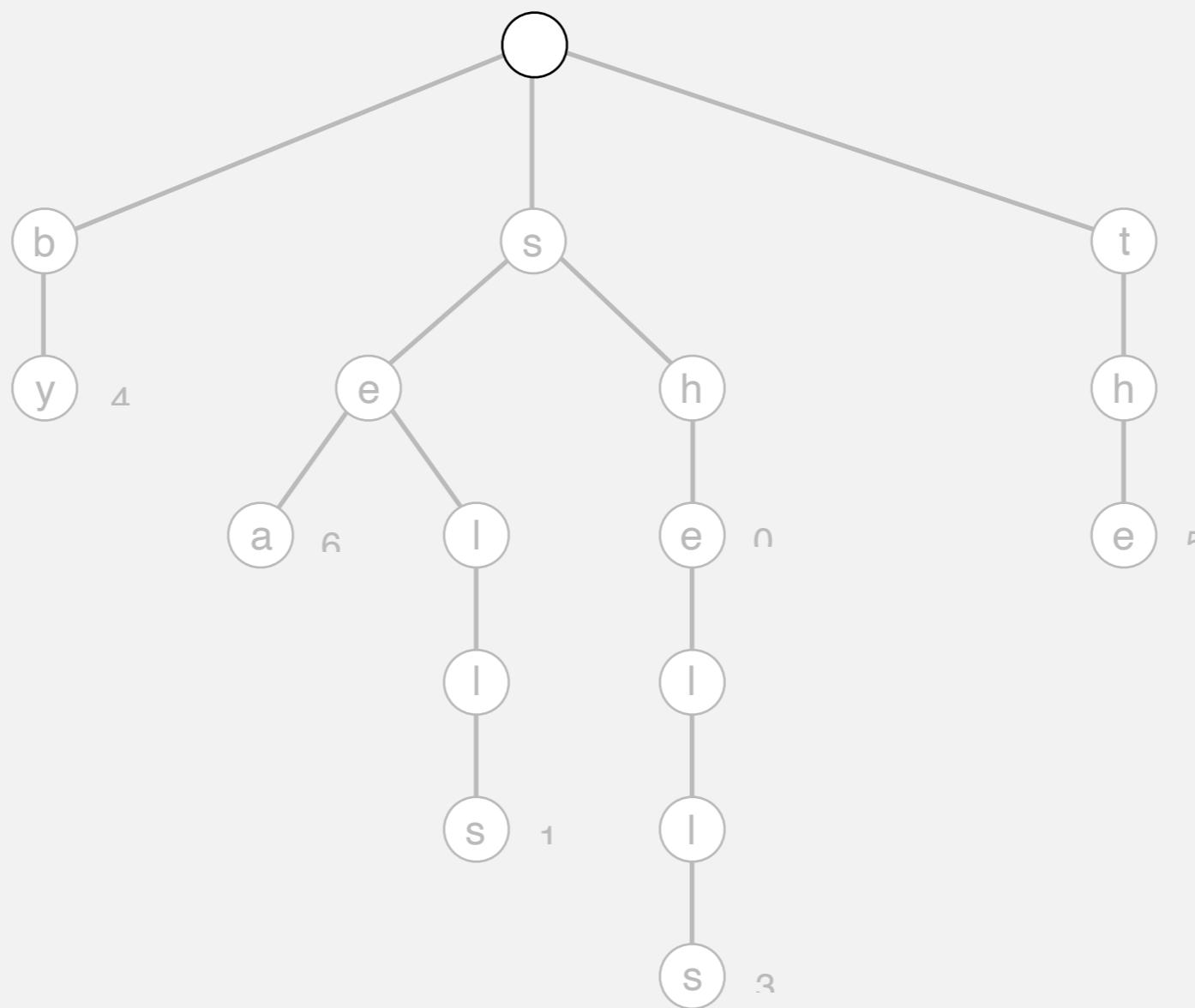


Insertion into a trie

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)

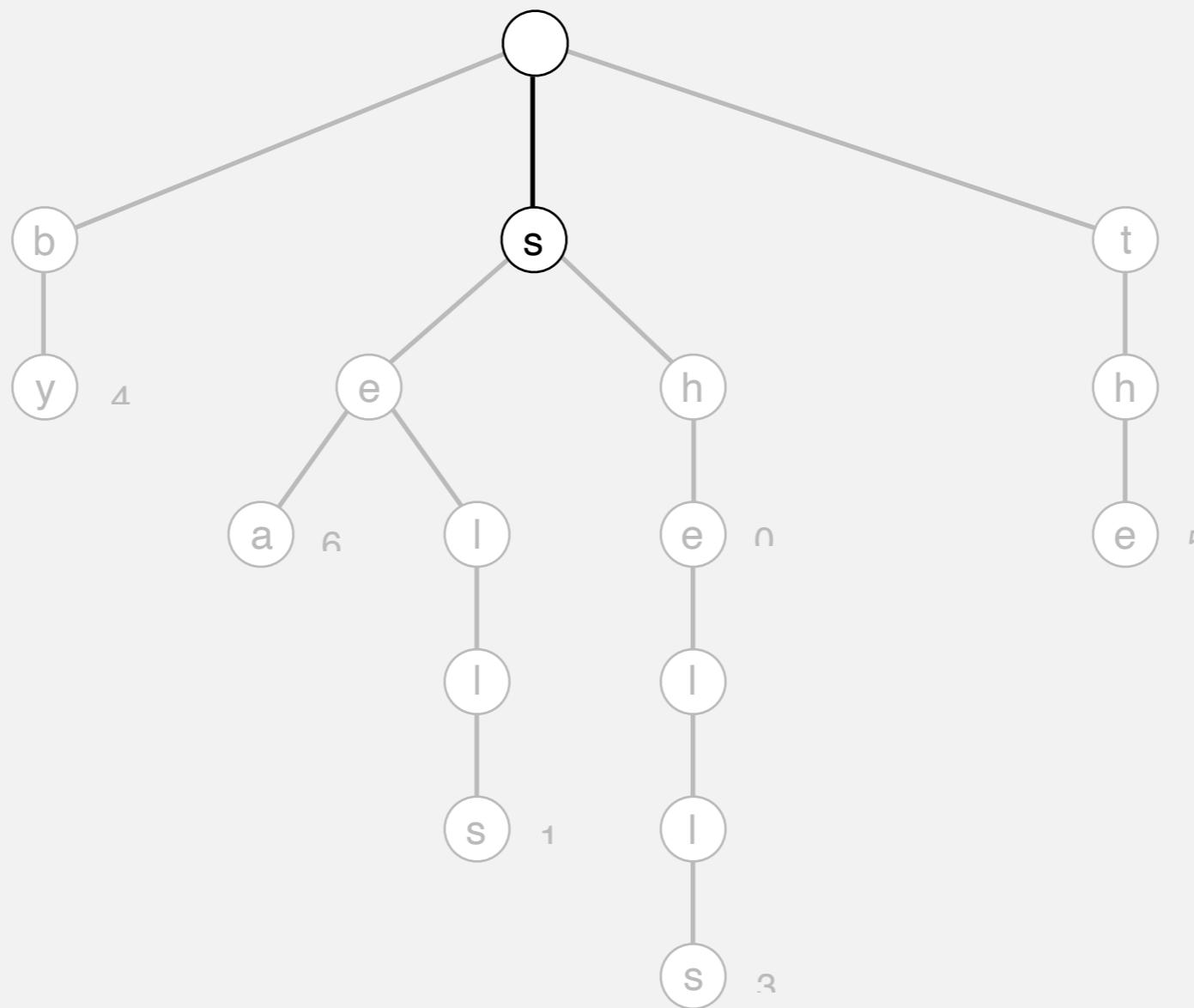


Insertion into a trie

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)

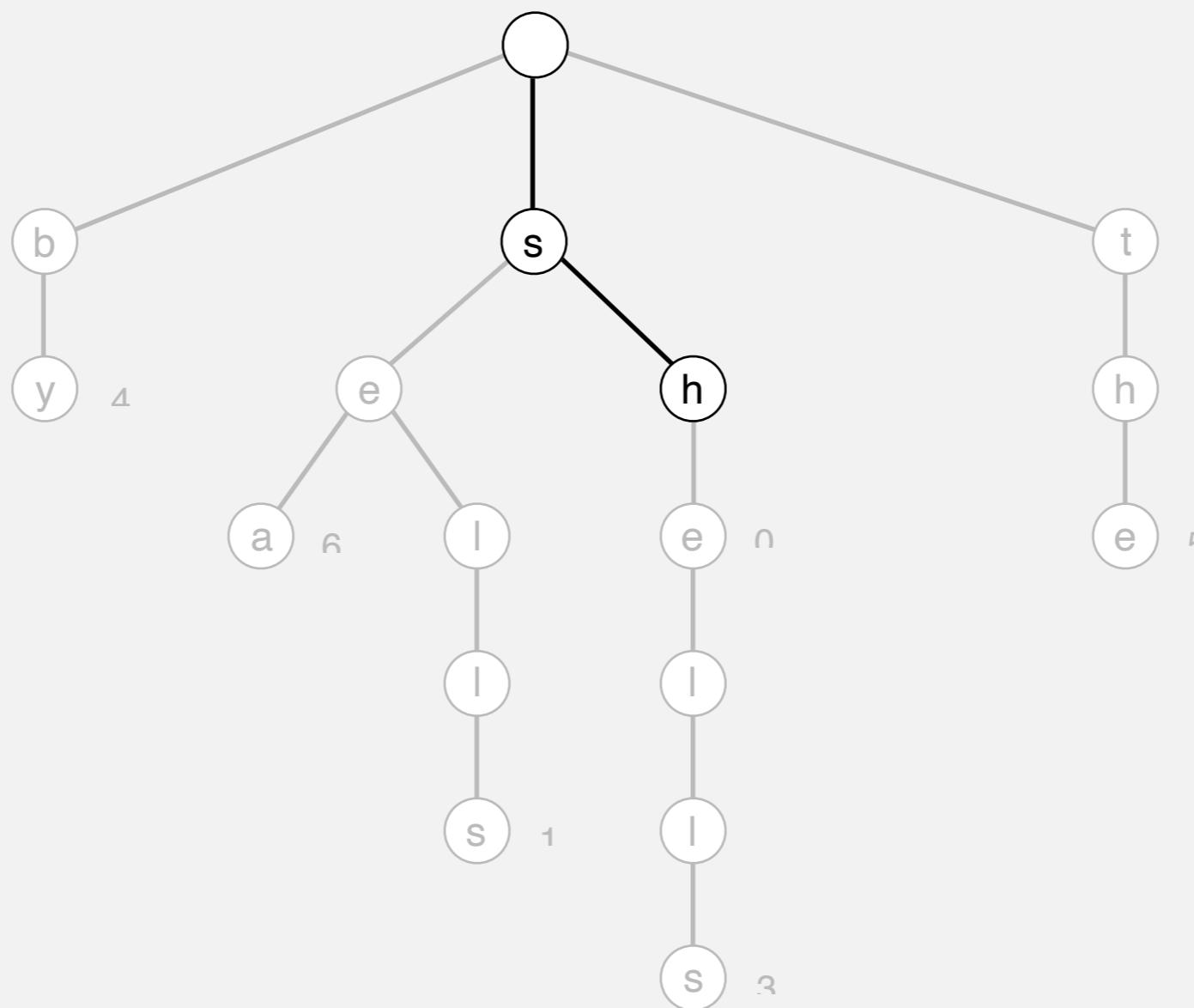


Insertion into a trie

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)

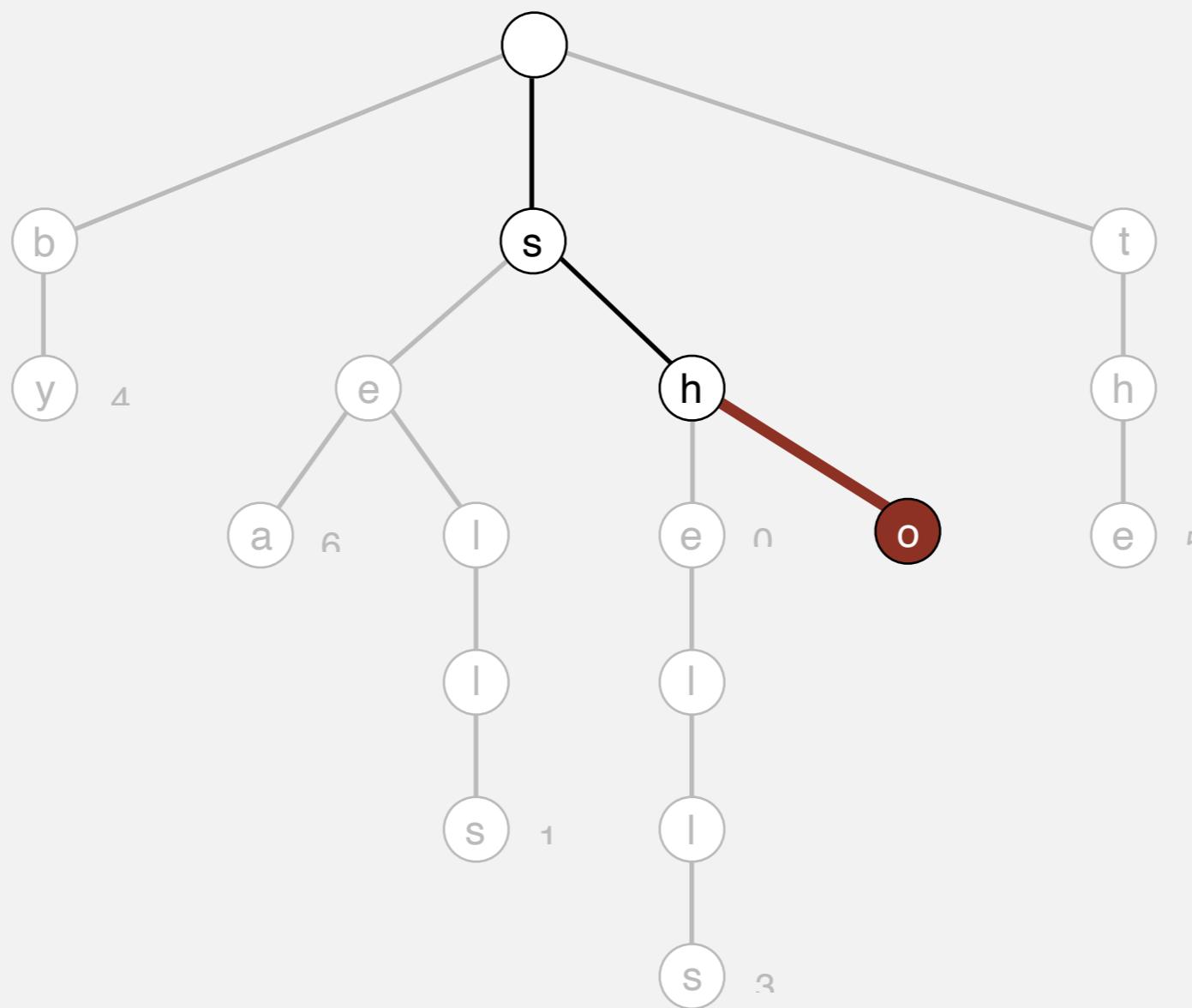


Insertion into a trie

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)

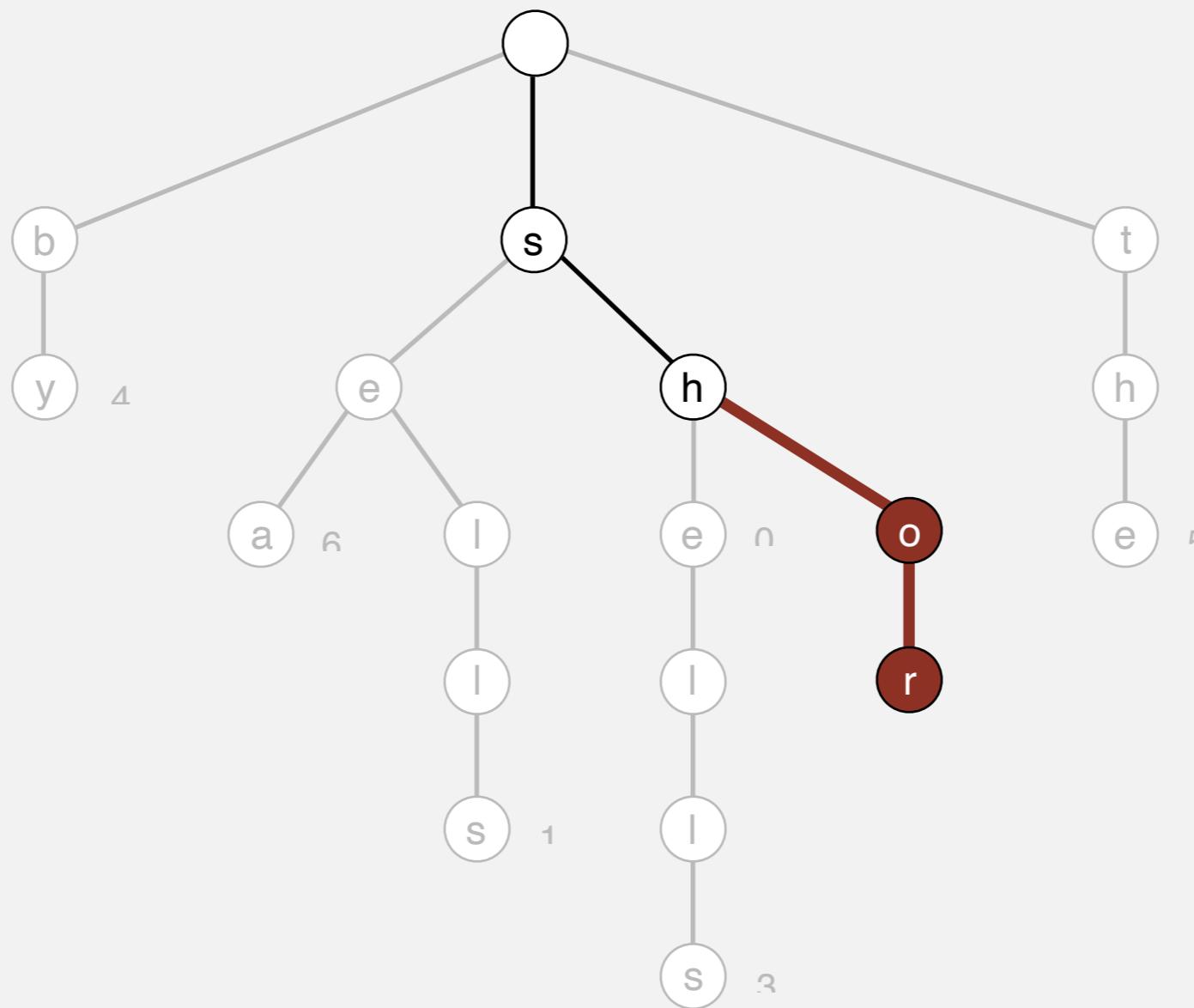


Insertion into a trie

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)

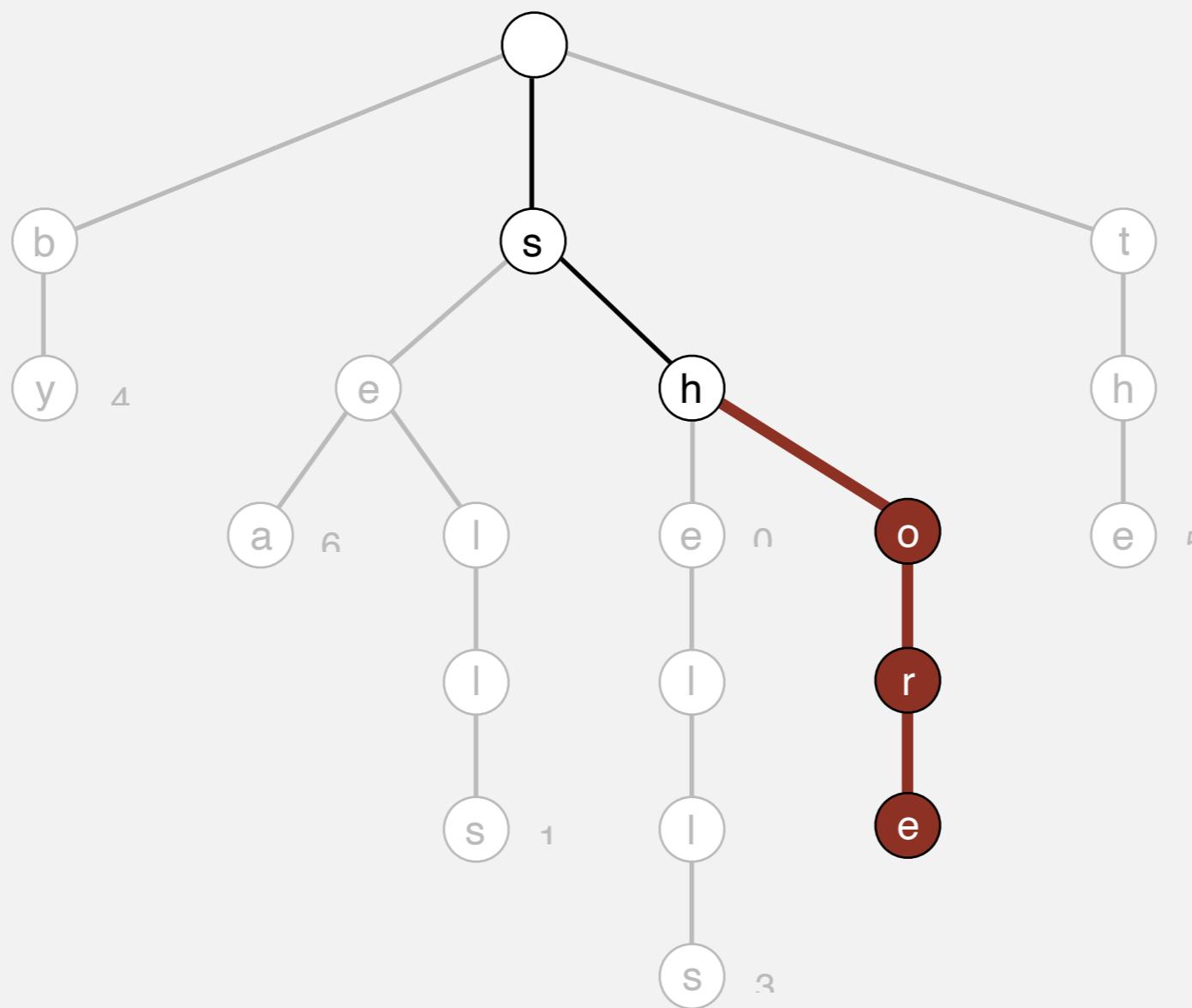


Insertion into a trie

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)

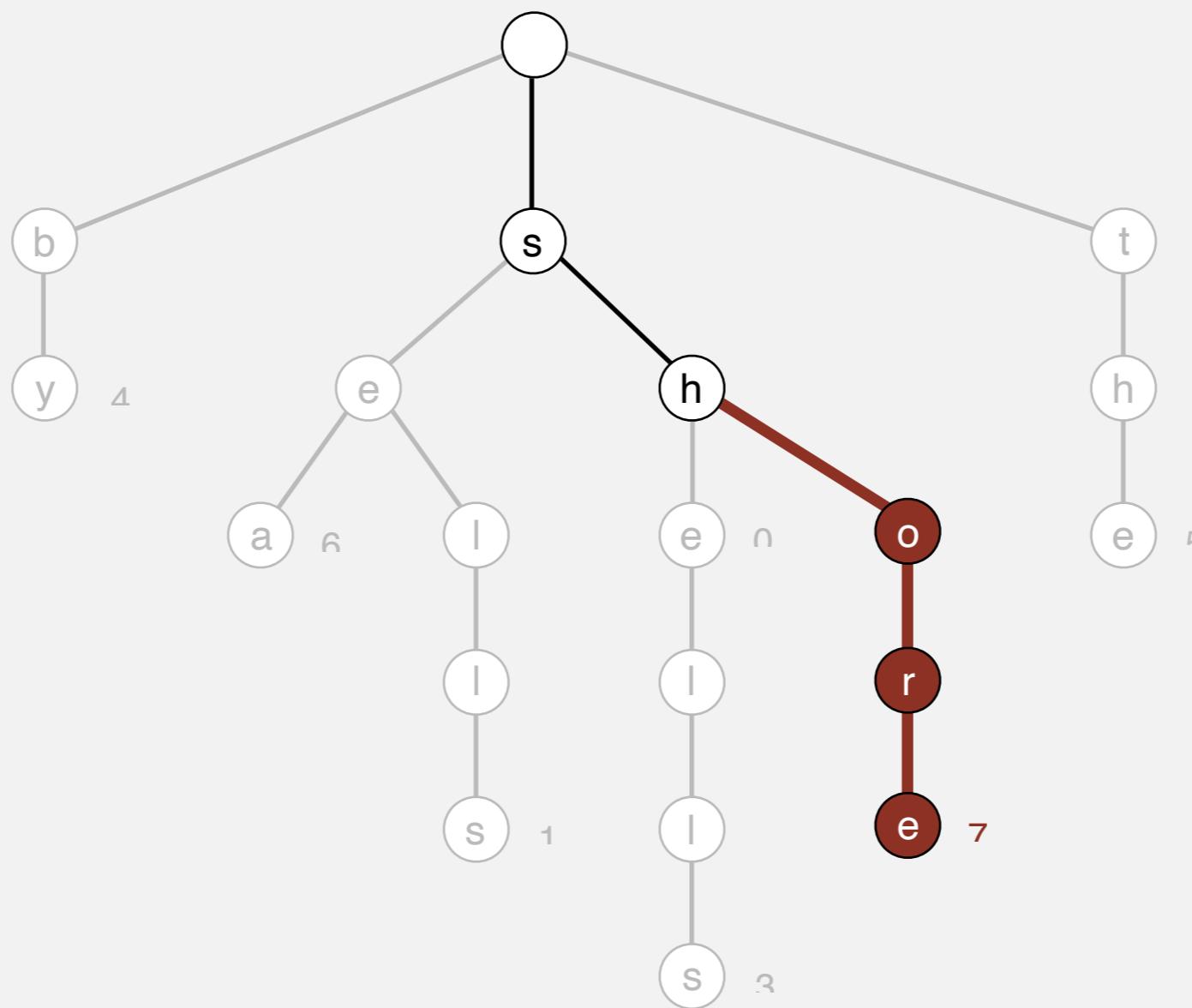


Insertion into a trie

Follow links corresponding to each character in the key.

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

put("shore", 7)



Trie construction demo

trie



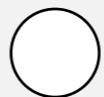
Trie construction demo

put("she", 0)



Trie construction demo

put("she", 0)



Trie construction demo

put("she", 0)



Trie construction demo

put("she", 0)



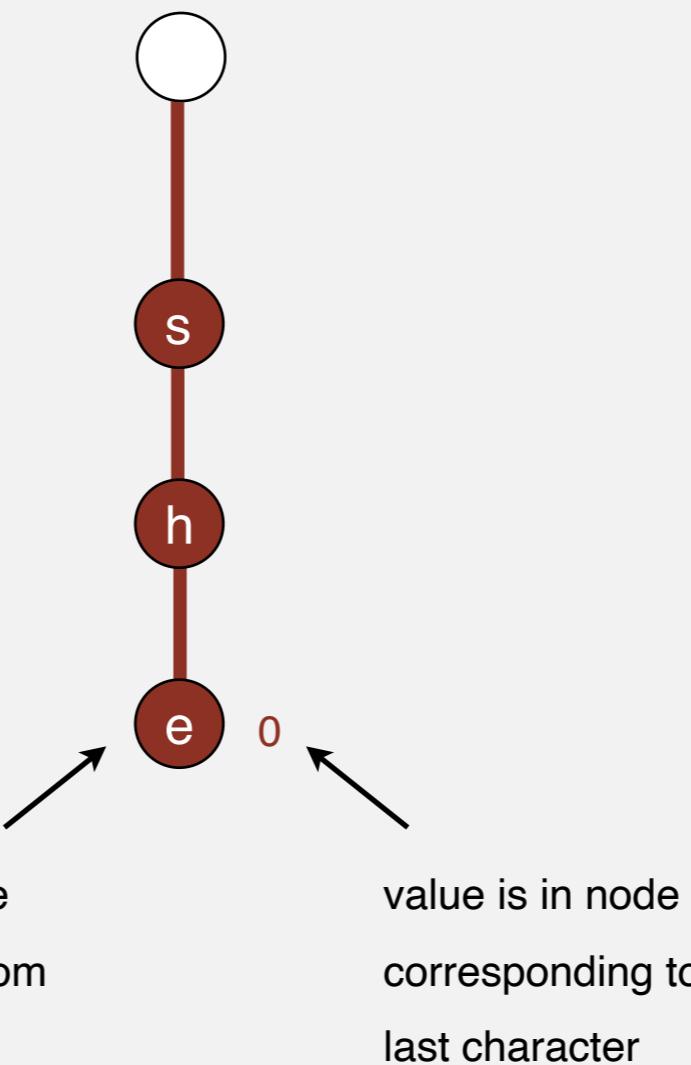
Trie construction demo

put("she", 0)



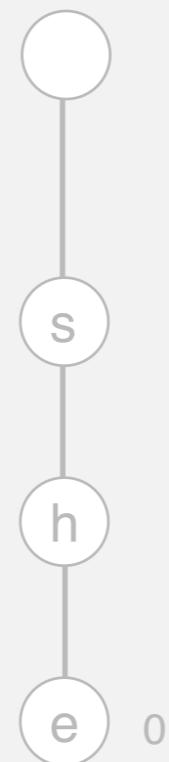
Trie construction demo

`put("she", 0)`



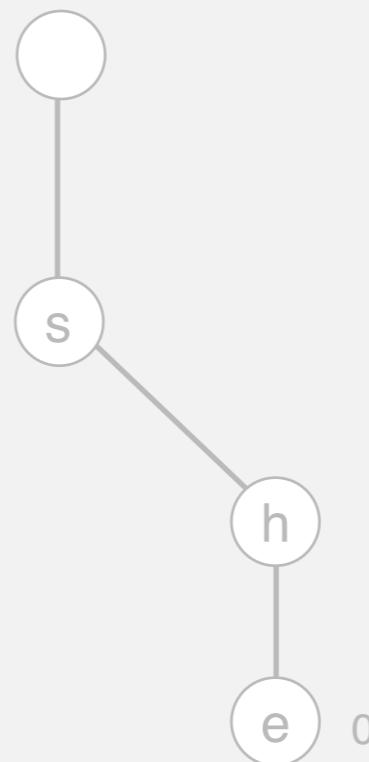
Trie construction demo

trie



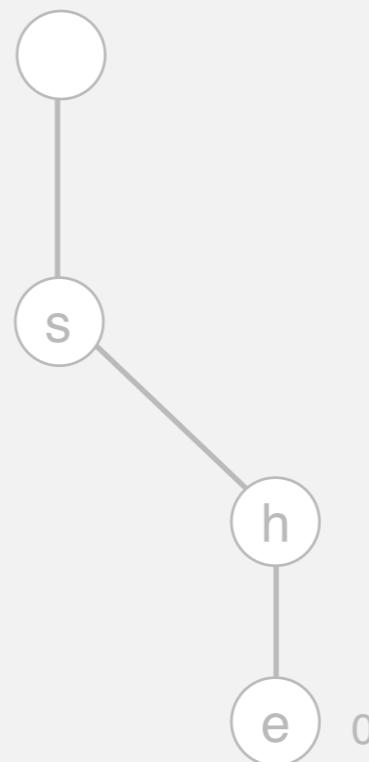
Trie construction demo

trie



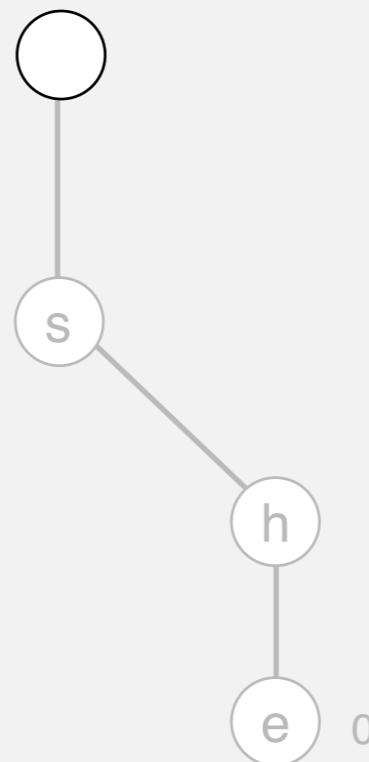
Trie construction demo

put("sells", 1)



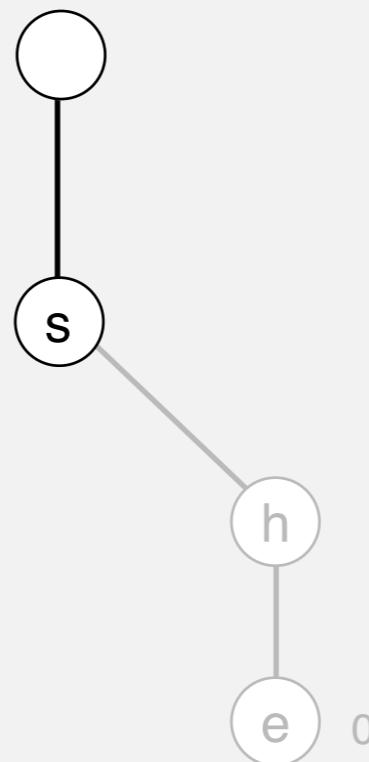
Trie construction demo

put("sells", 1)



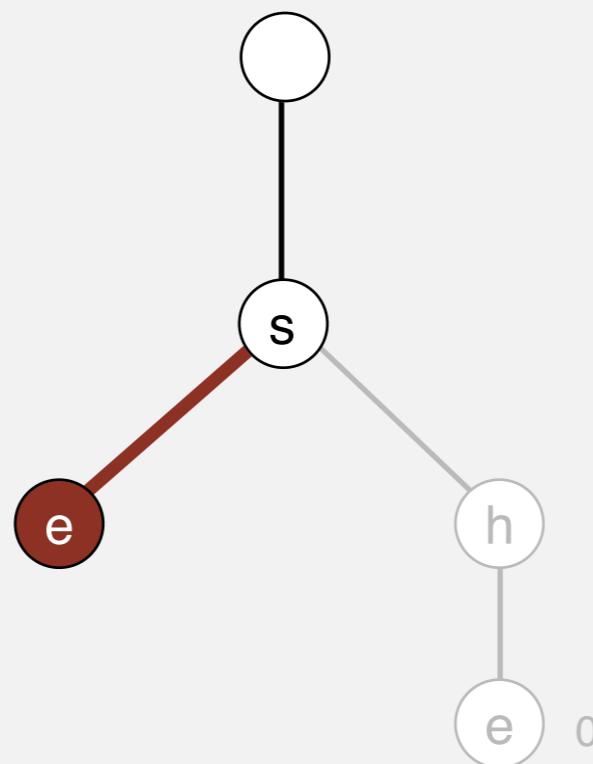
Trie construction demo

put("sells", 1)



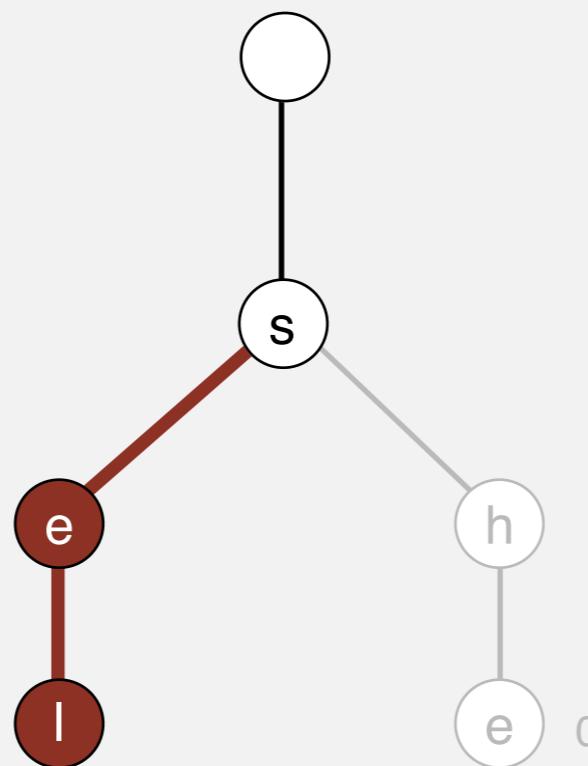
Trie construction demo

put("sells", 1)



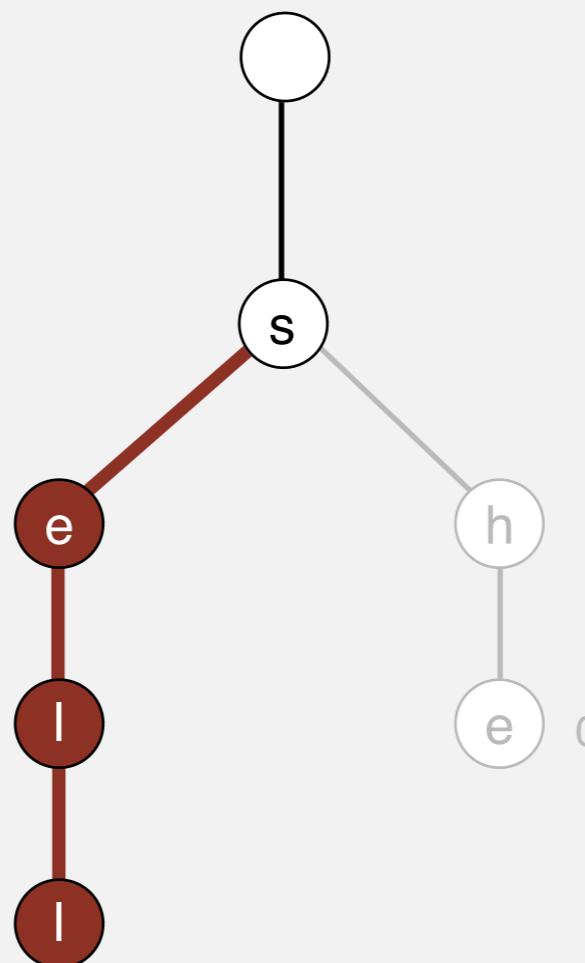
Trie construction demo

put("sells", 1)



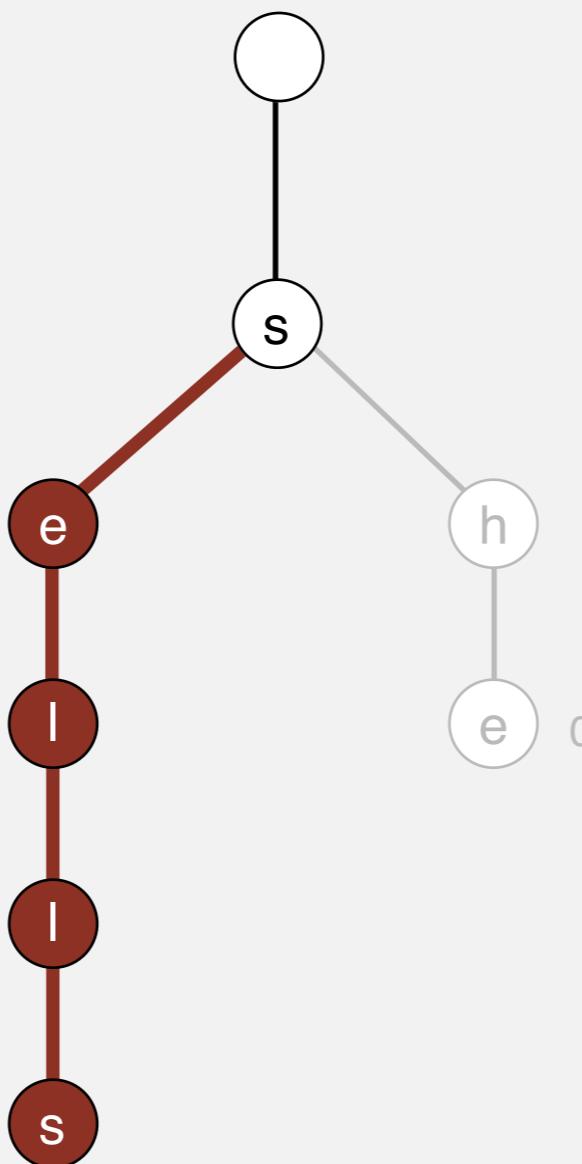
Trie construction demo

put("sells", 1)



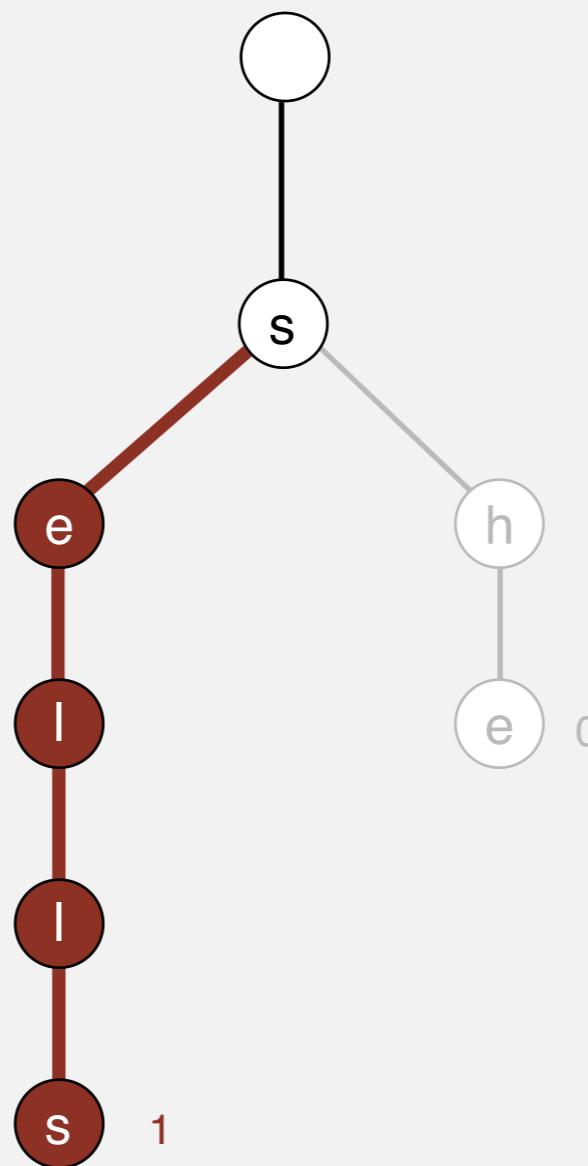
Trie construction demo

put("sells", 1)



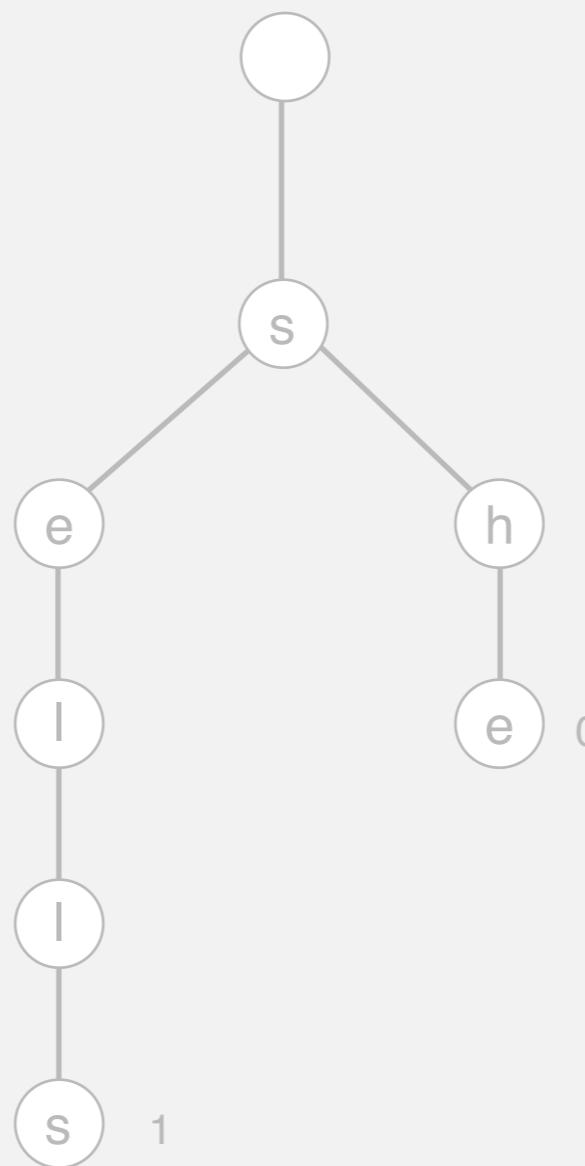
Trie construction demo

```
put("sells", 1)
```



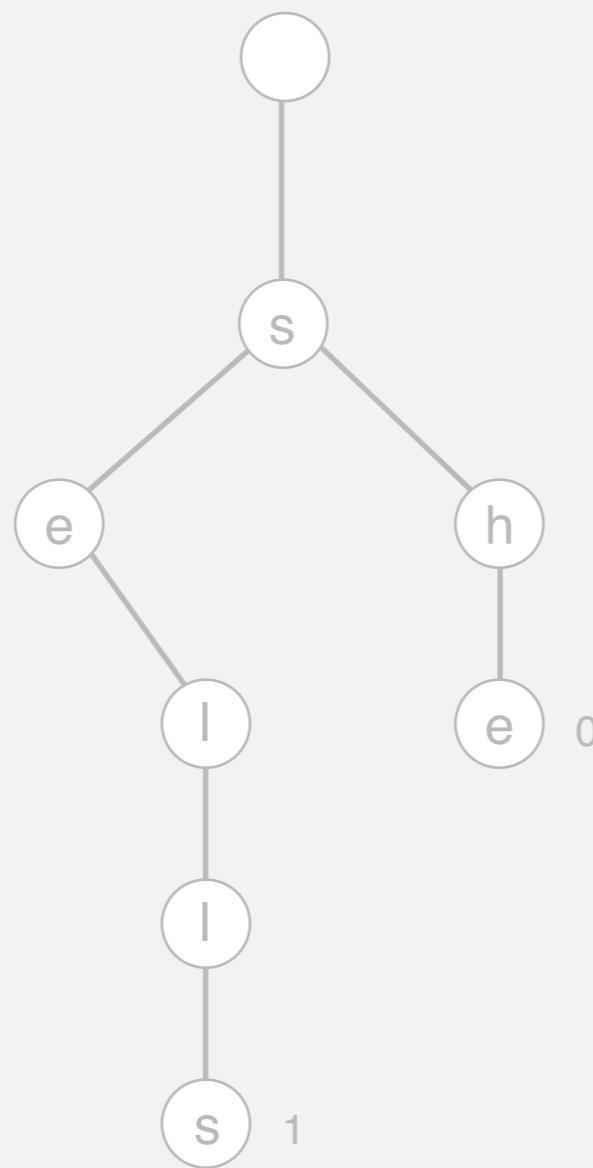
Trie construction demo

trie



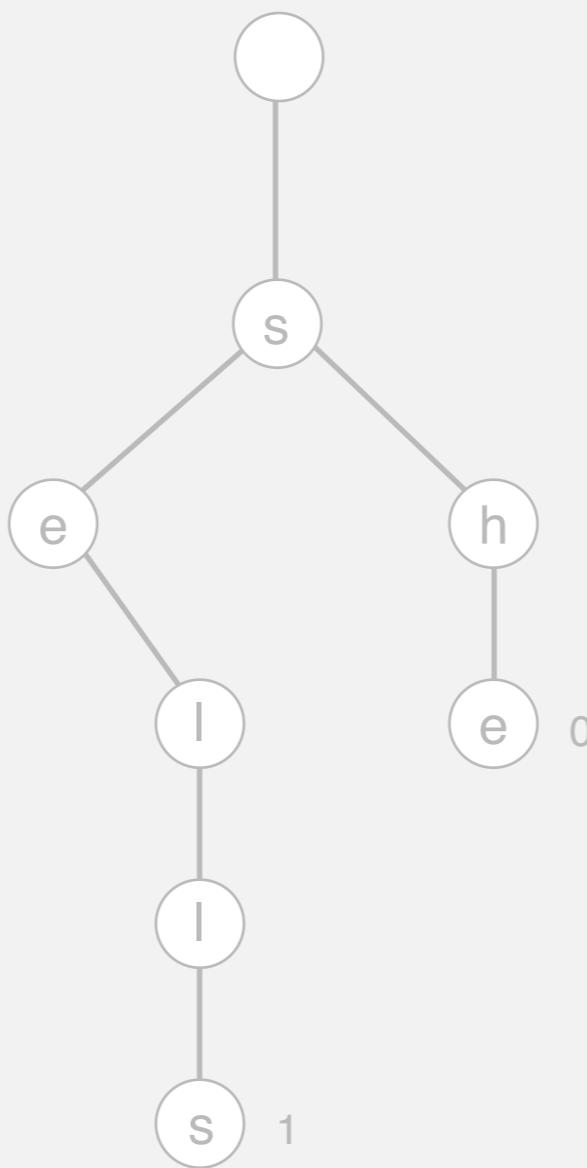
Trie construction demo

trie



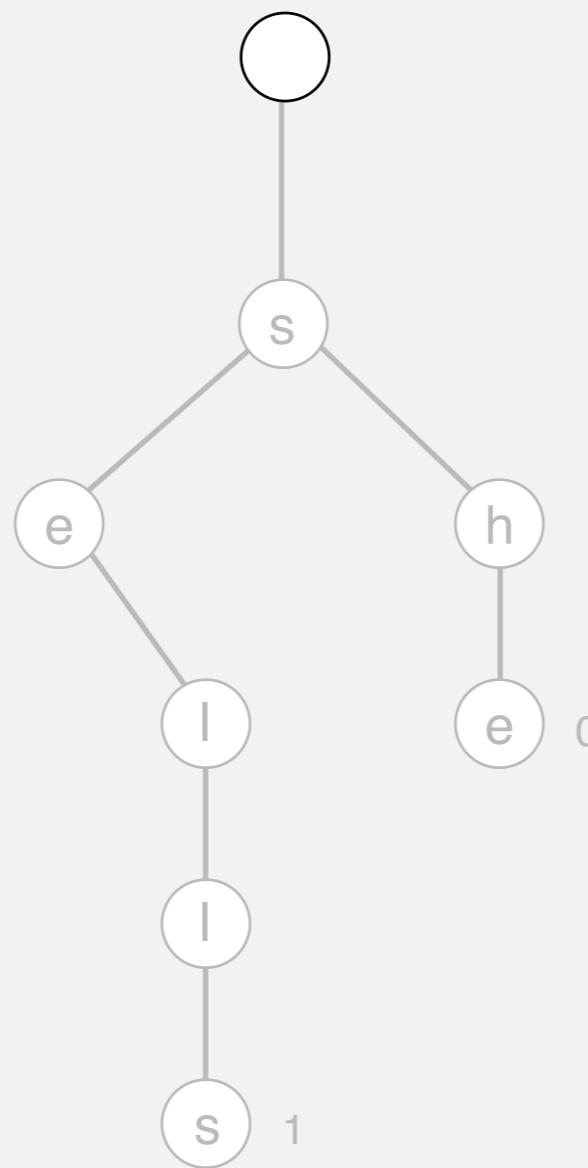
Trie construction demo

put("sea", 2)



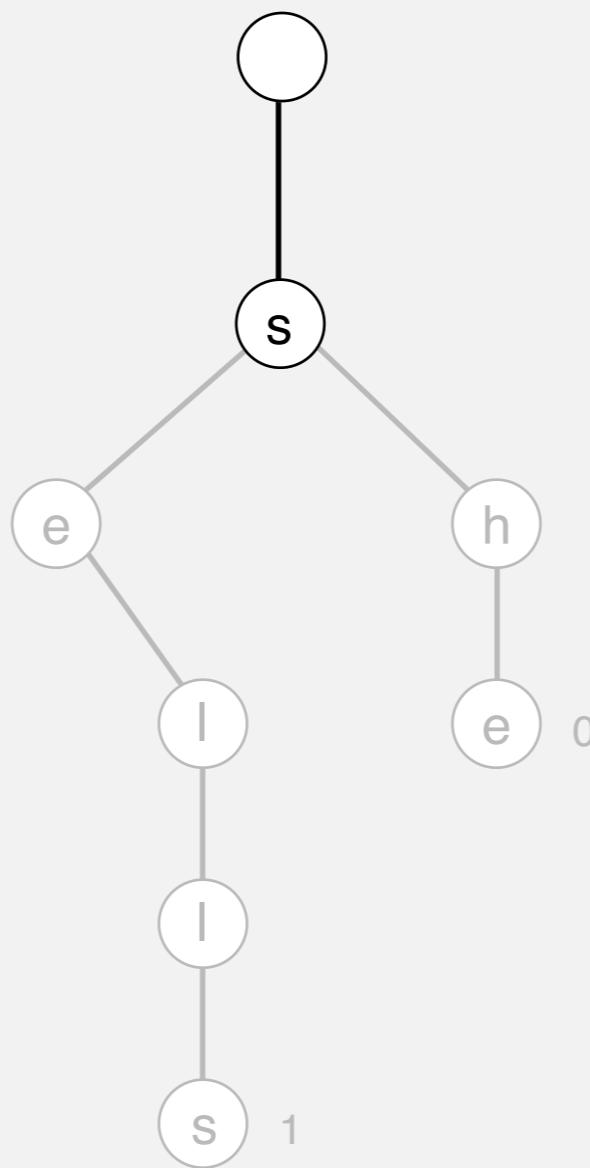
Trie construction demo

put("sea", 2)



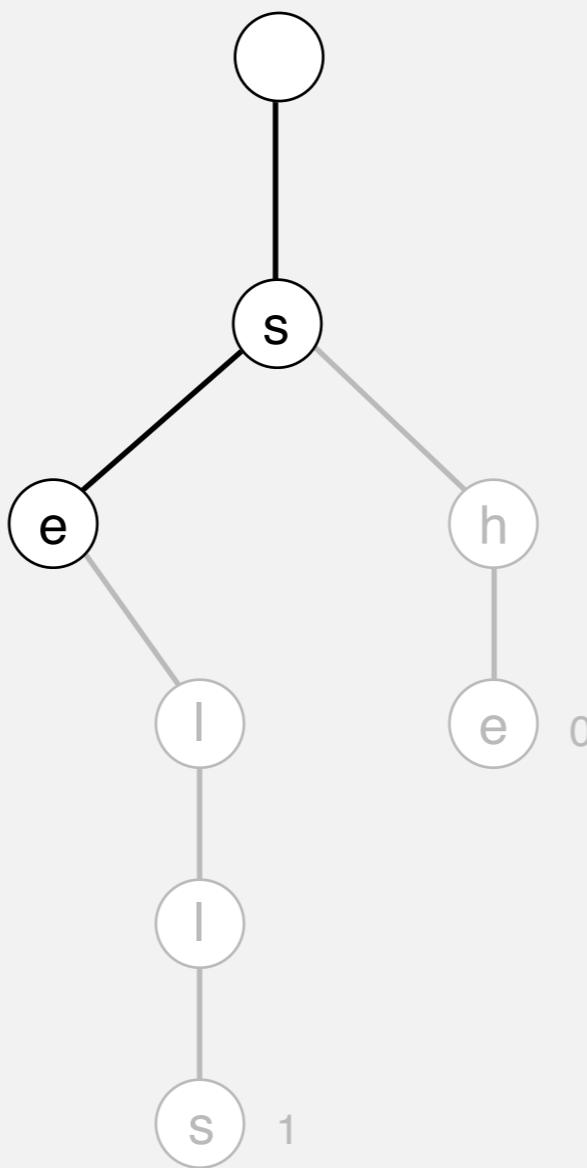
Trie construction demo

put("sea", 2)



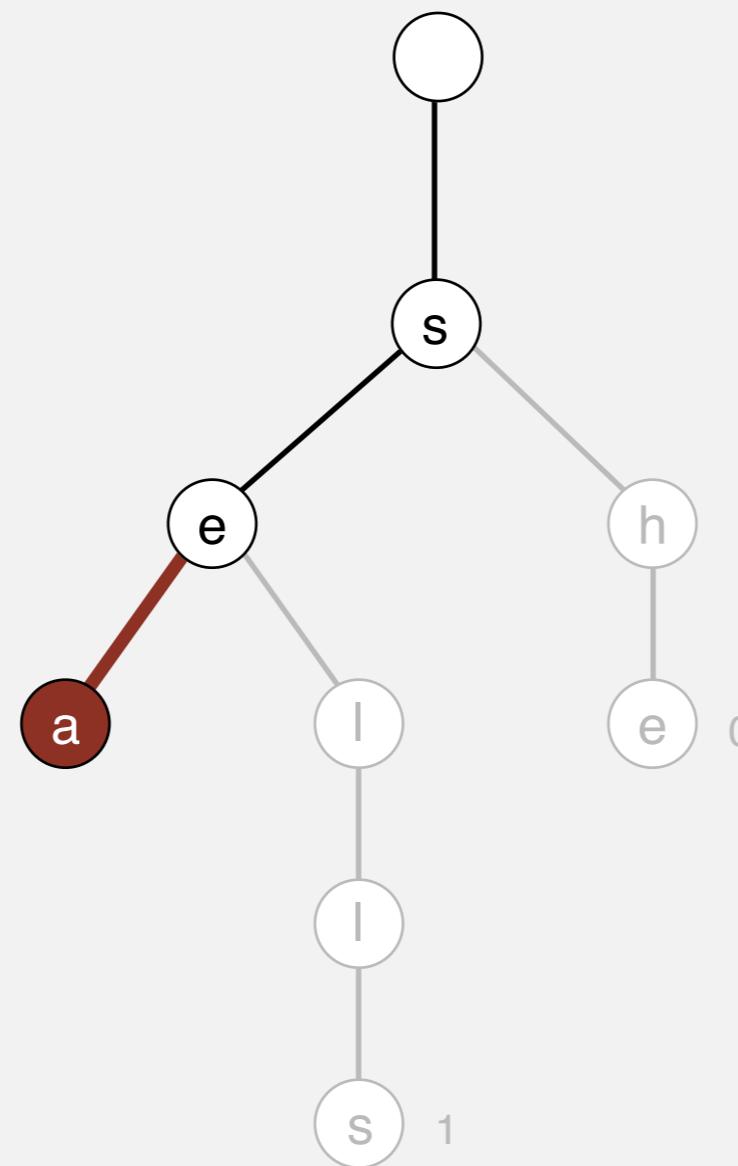
Trie construction demo

put("sea", 2)



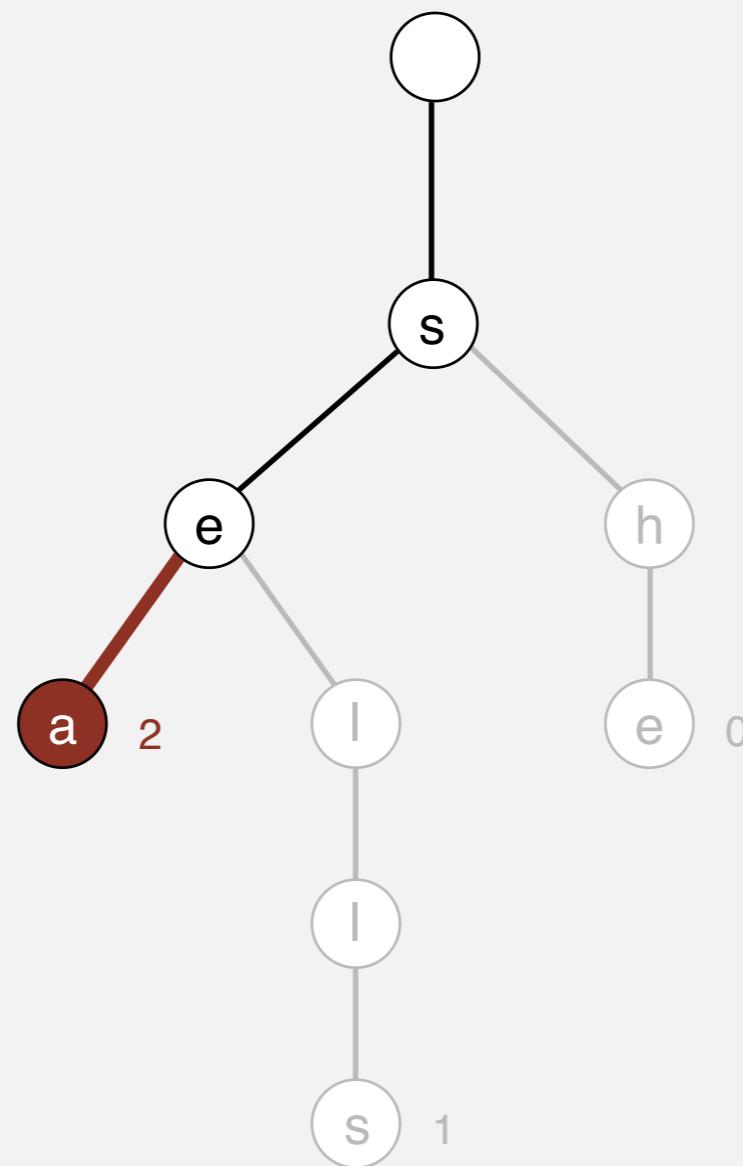
Trie construction demo

put("sea", 2)



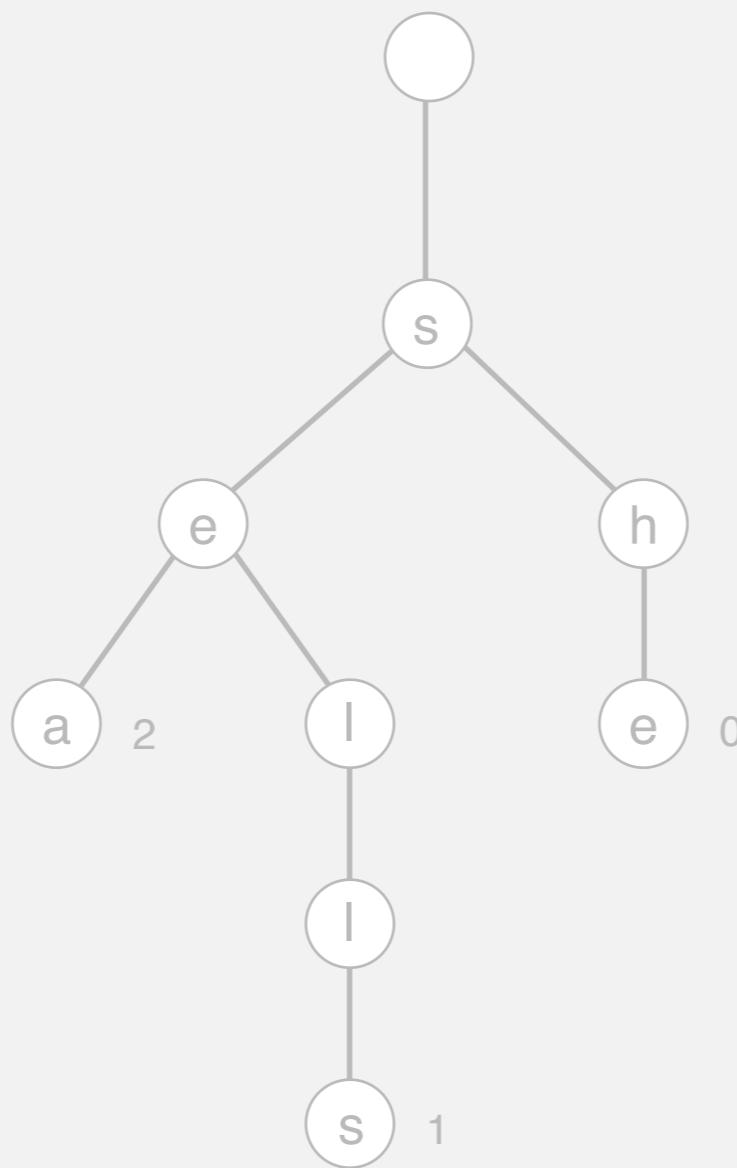
Trie construction demo

put("sea", 2)



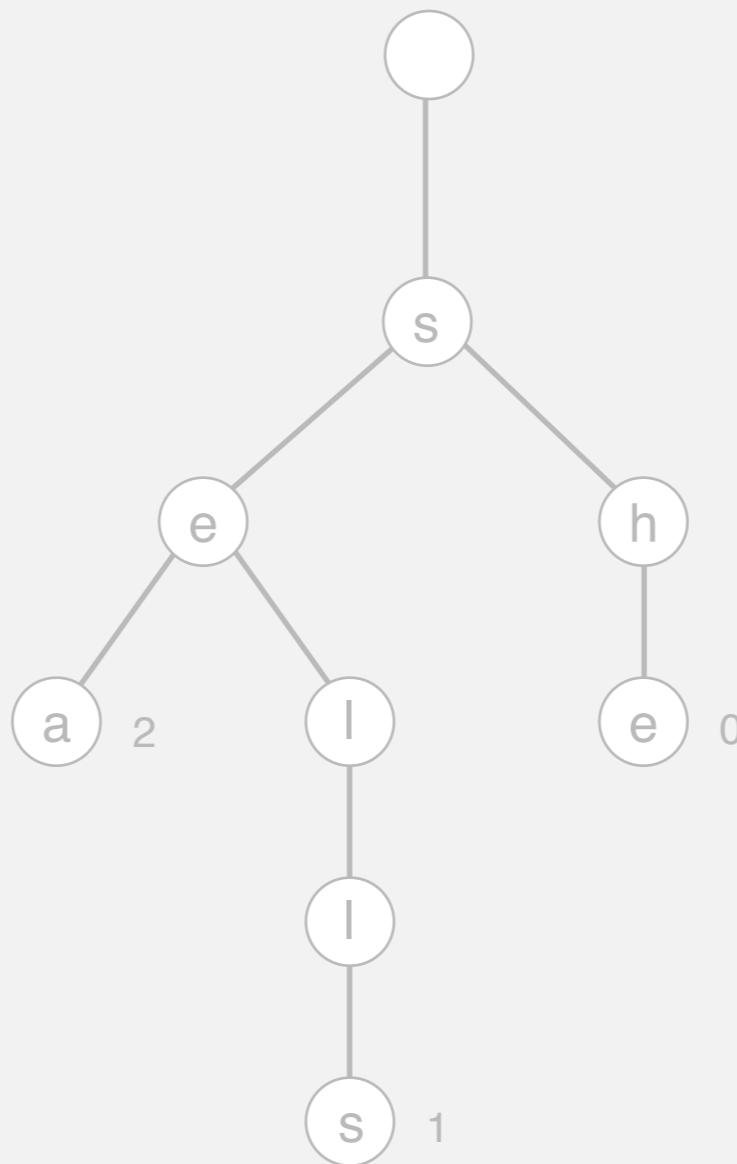
Trie construction demo

trie



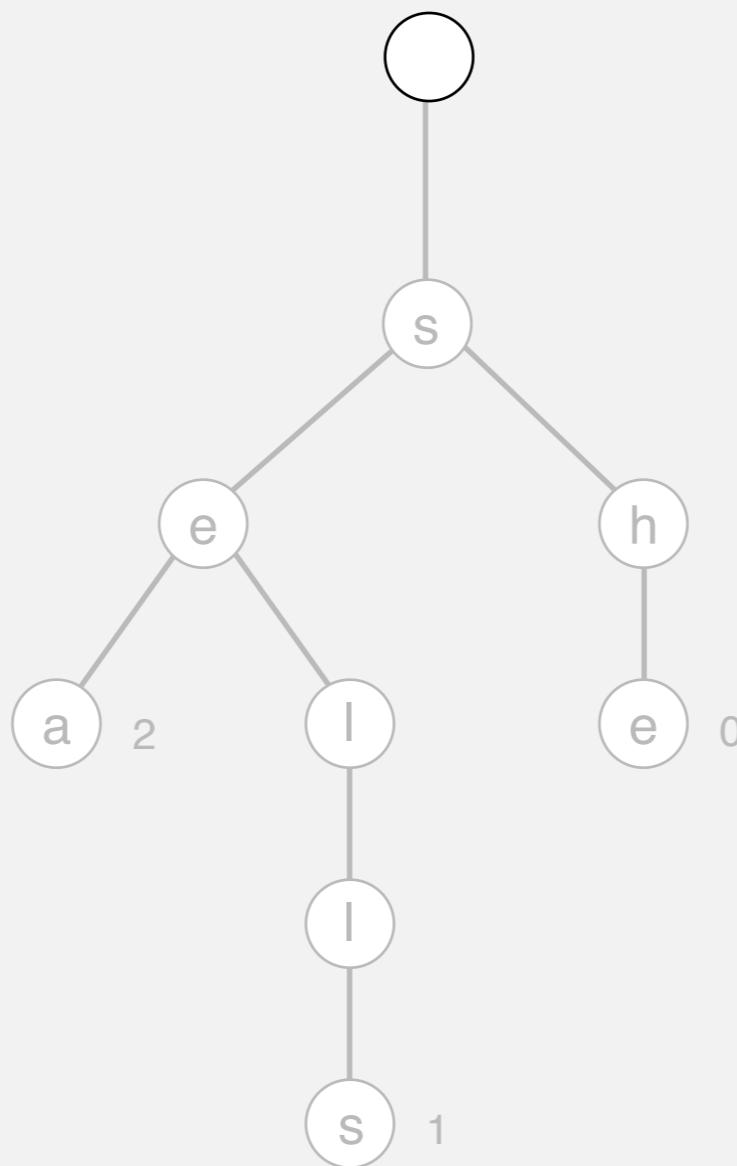
Trie construction demo

put("shells", 3)



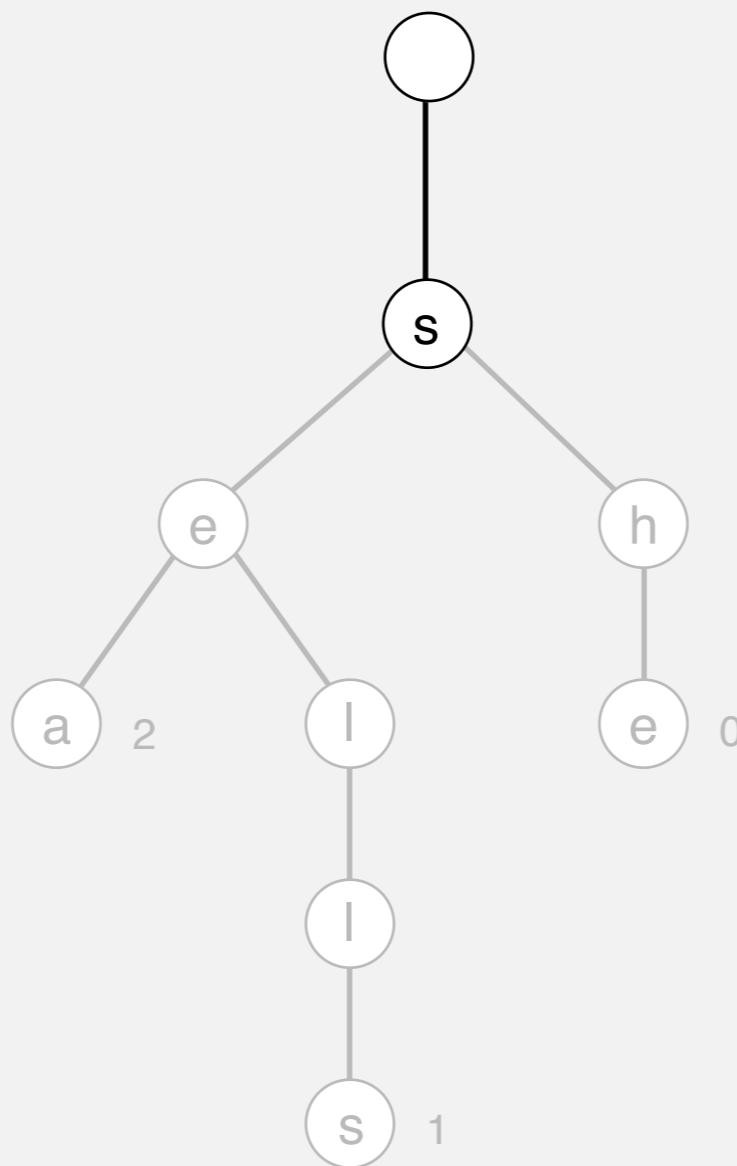
Trie construction demo

put("shells", 3)



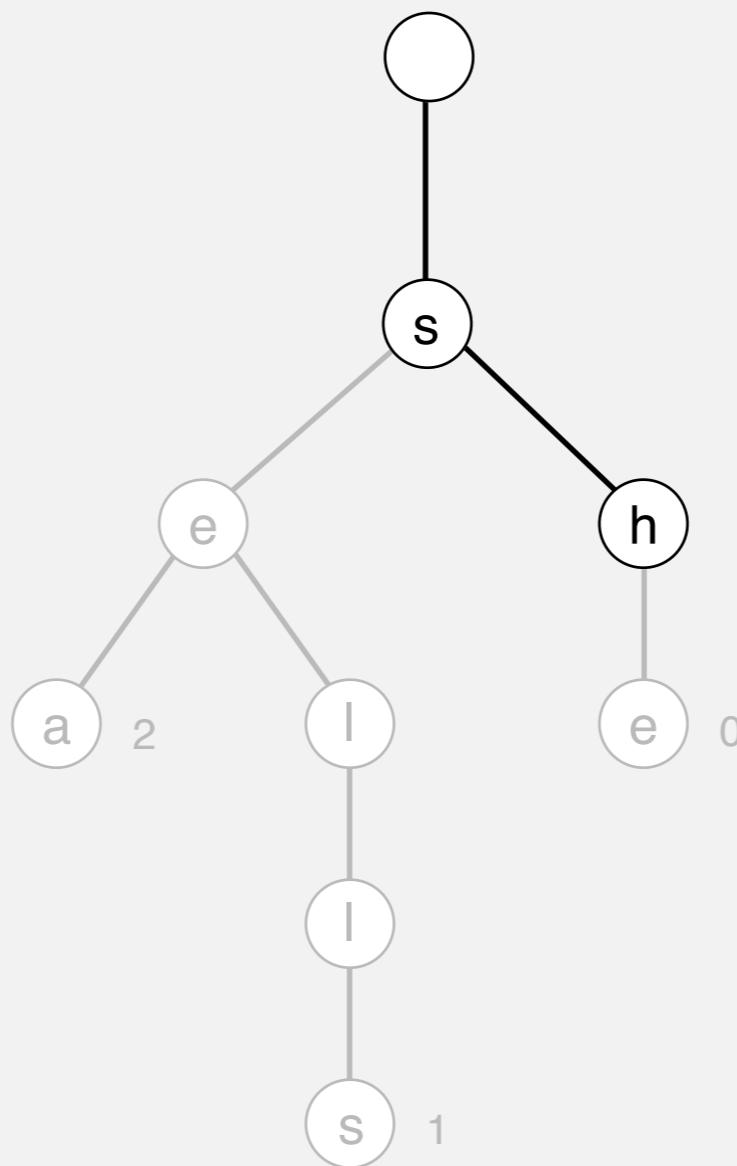
Trie construction demo

put("shells", 3)



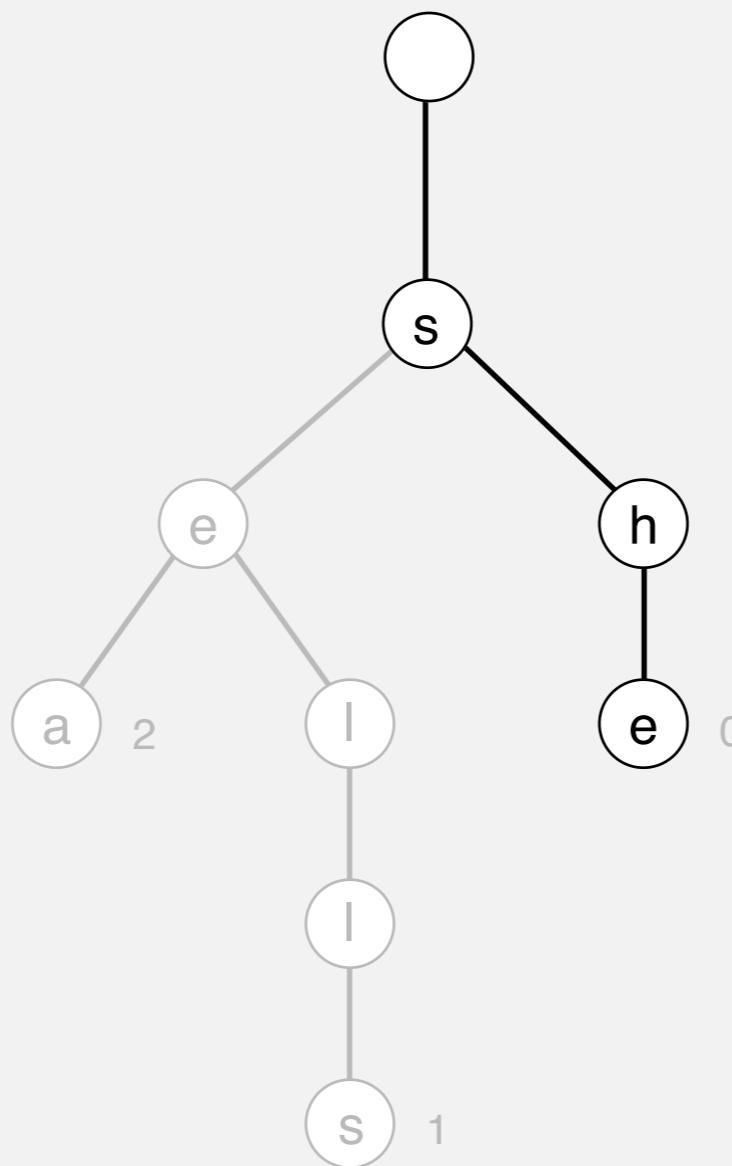
Trie construction demo

put("shells", 3)



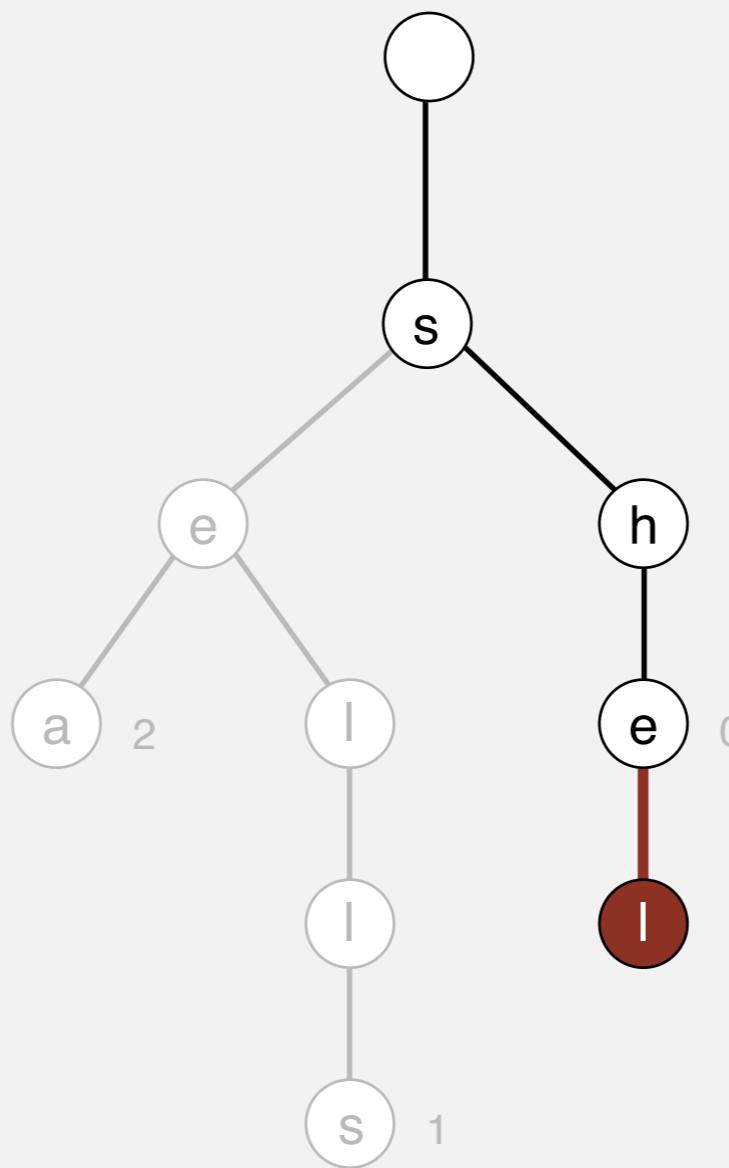
Trie construction demo

put("shells", 3)



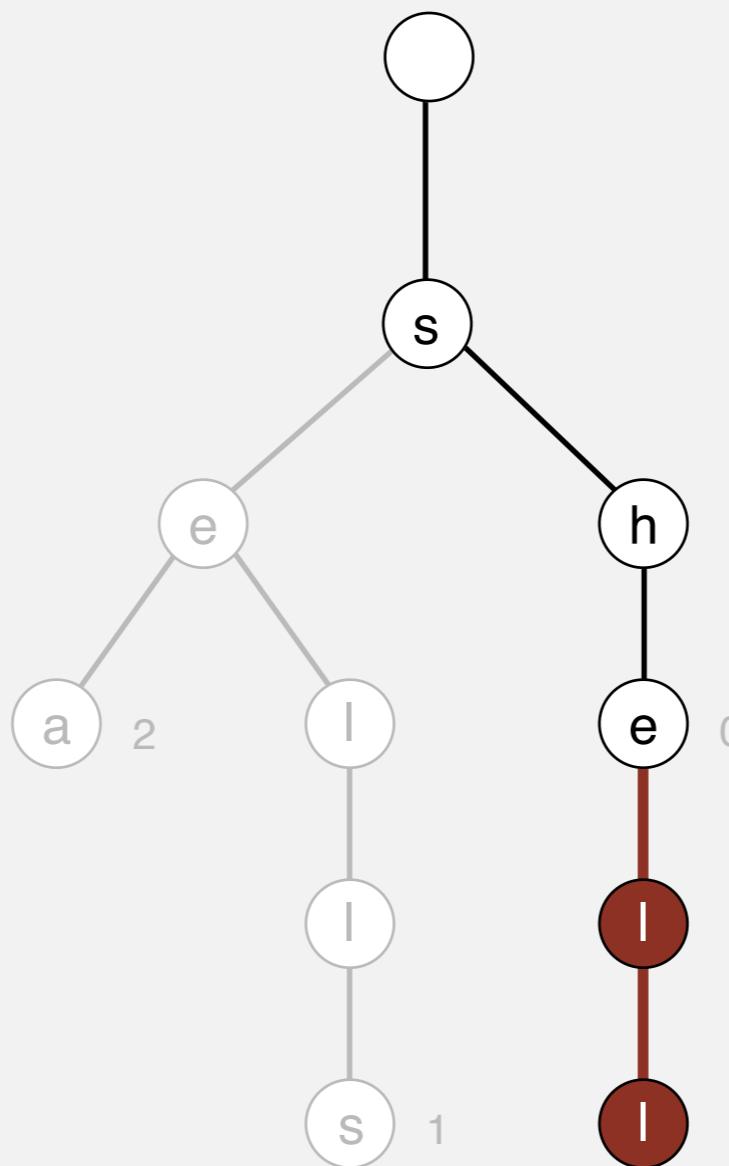
Trie construction demo

put("shells", 3)



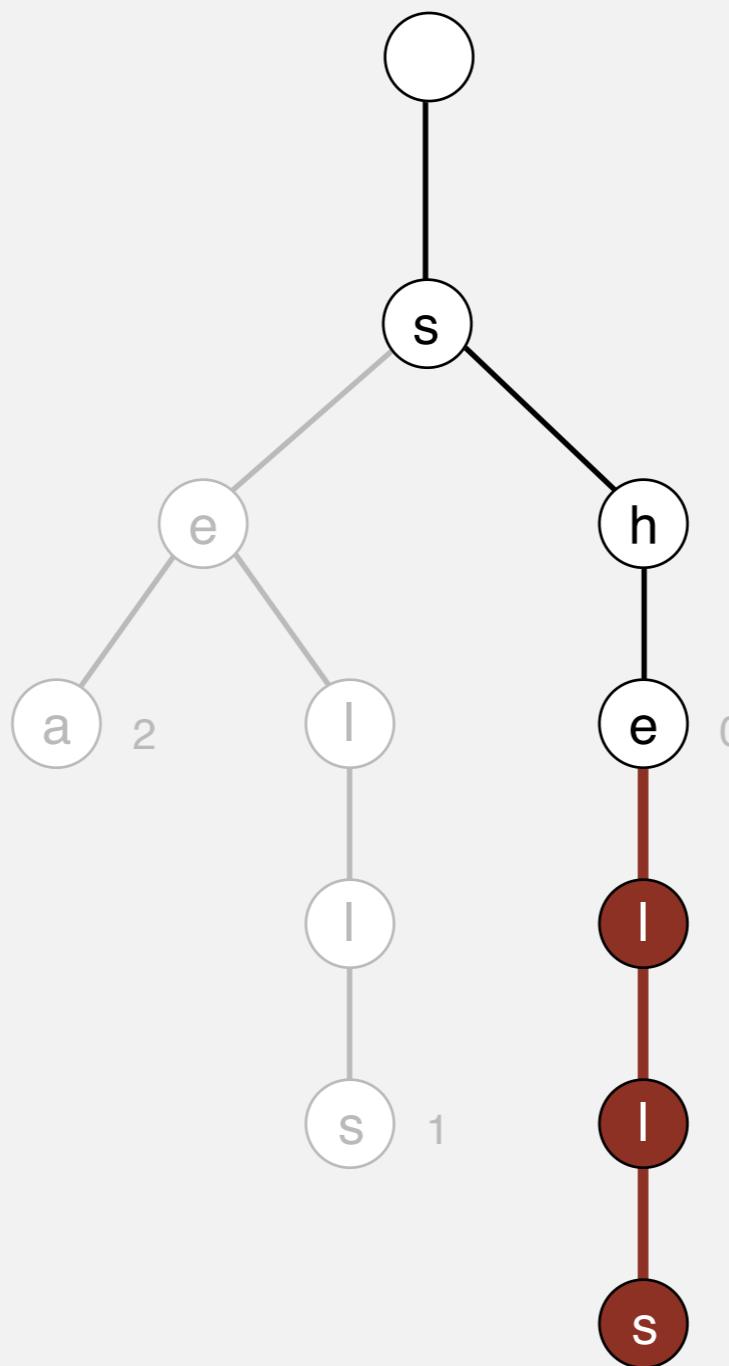
Trie construction demo

put("shells", 3)



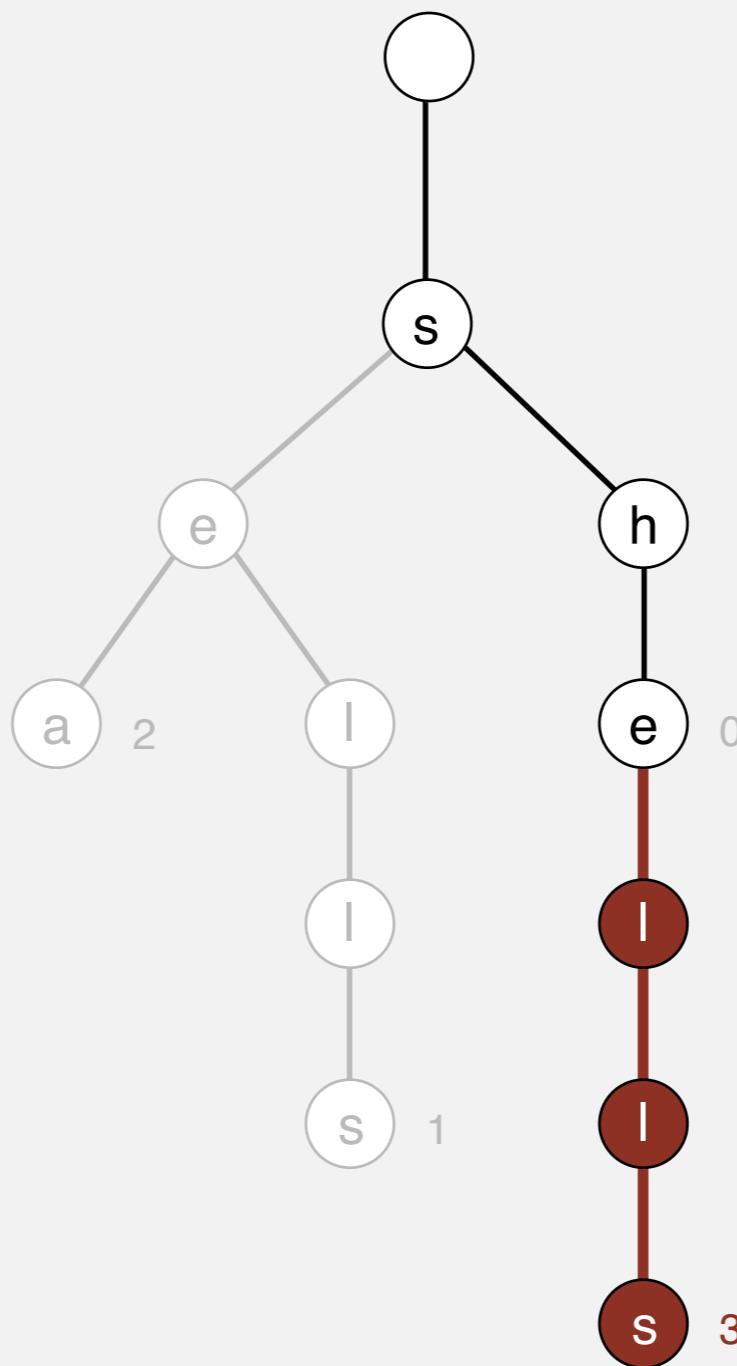
Trie construction demo

put("shells", 3)



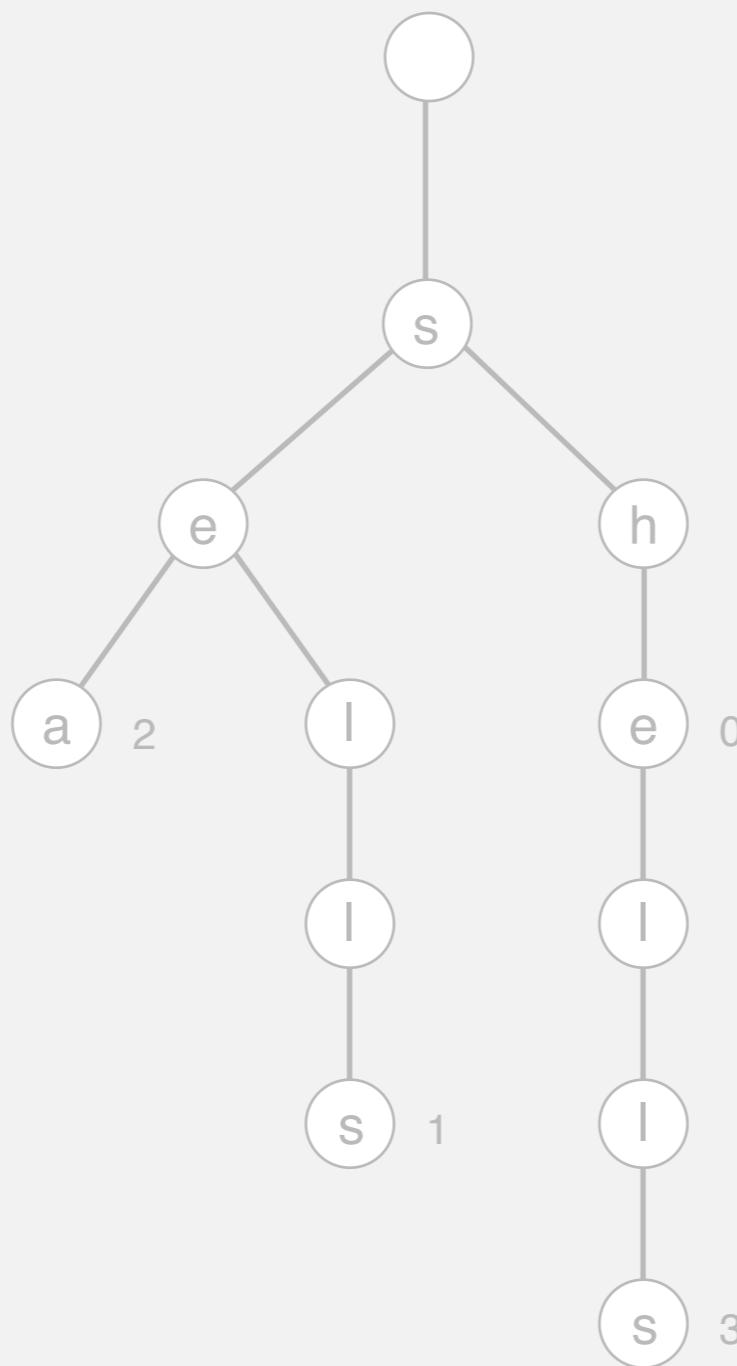
Trie construction demo

put("shells", 3)



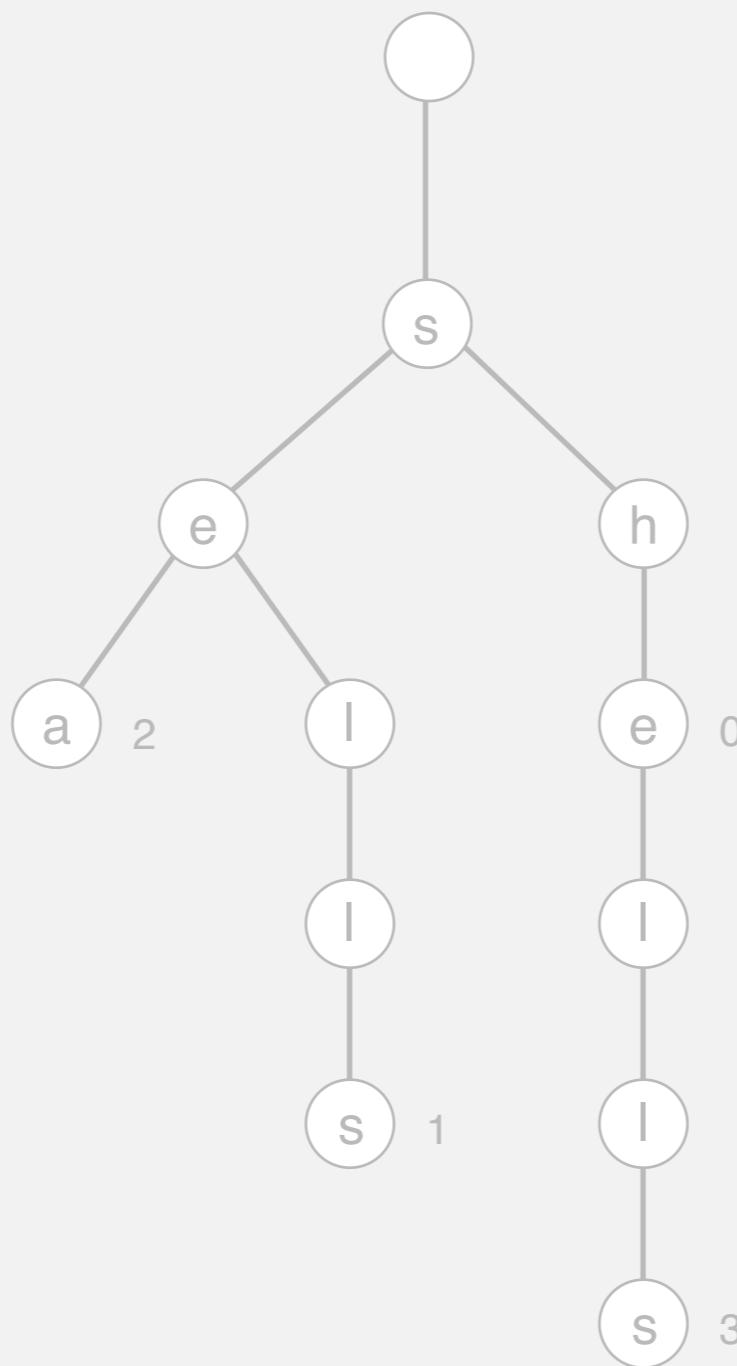
Trie construction demo

trie



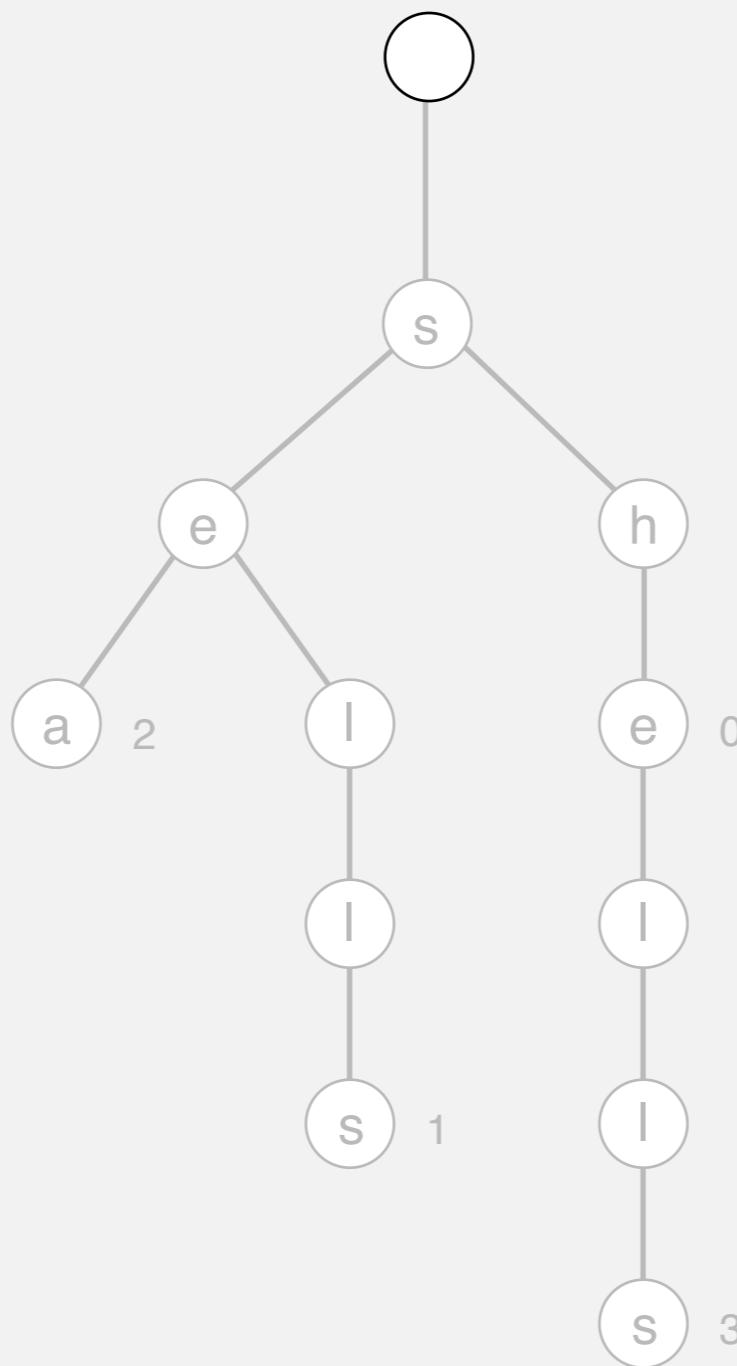
Trie construction demo

put("by", 4)



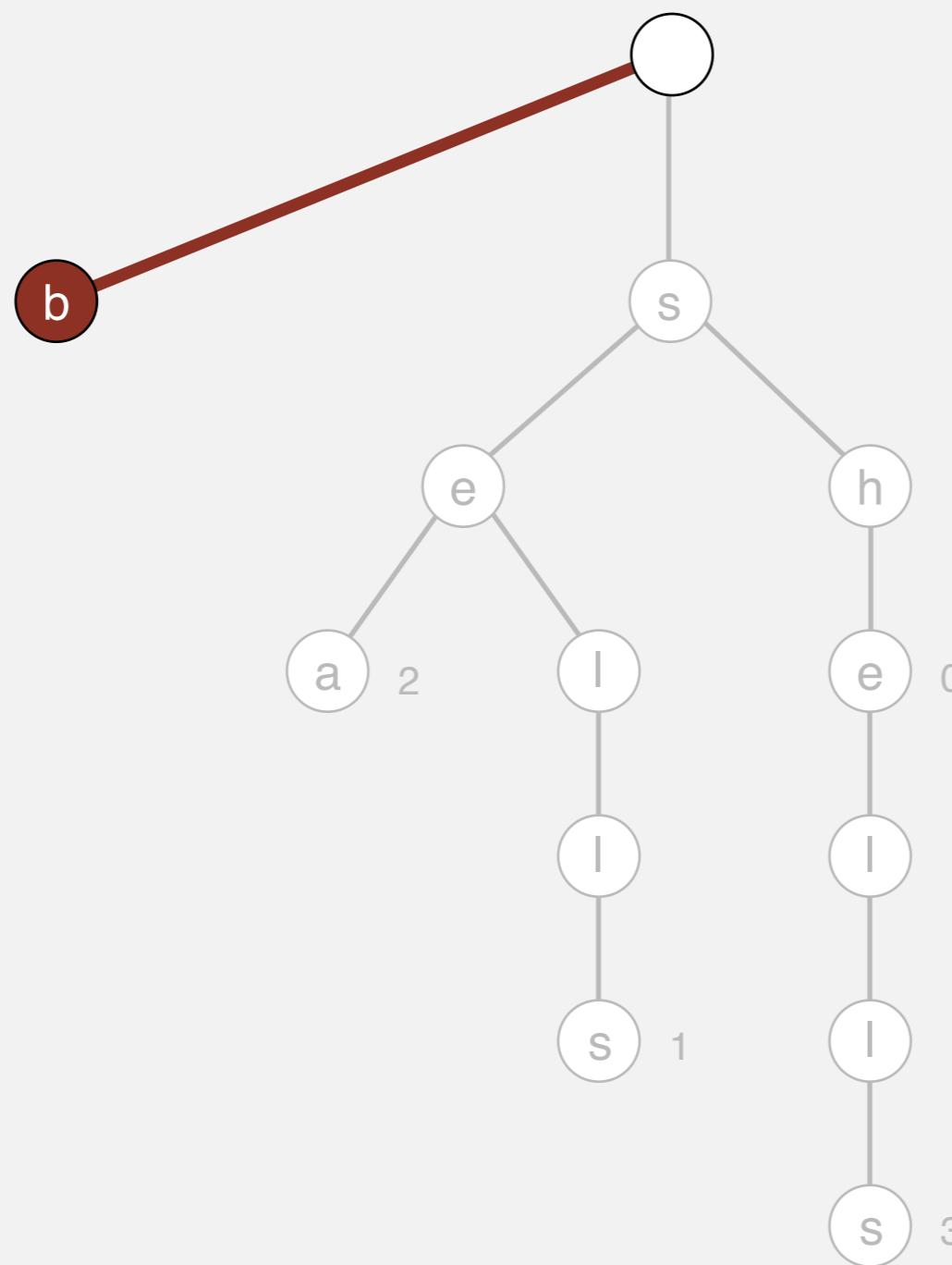
Trie construction demo

put("by", 4)



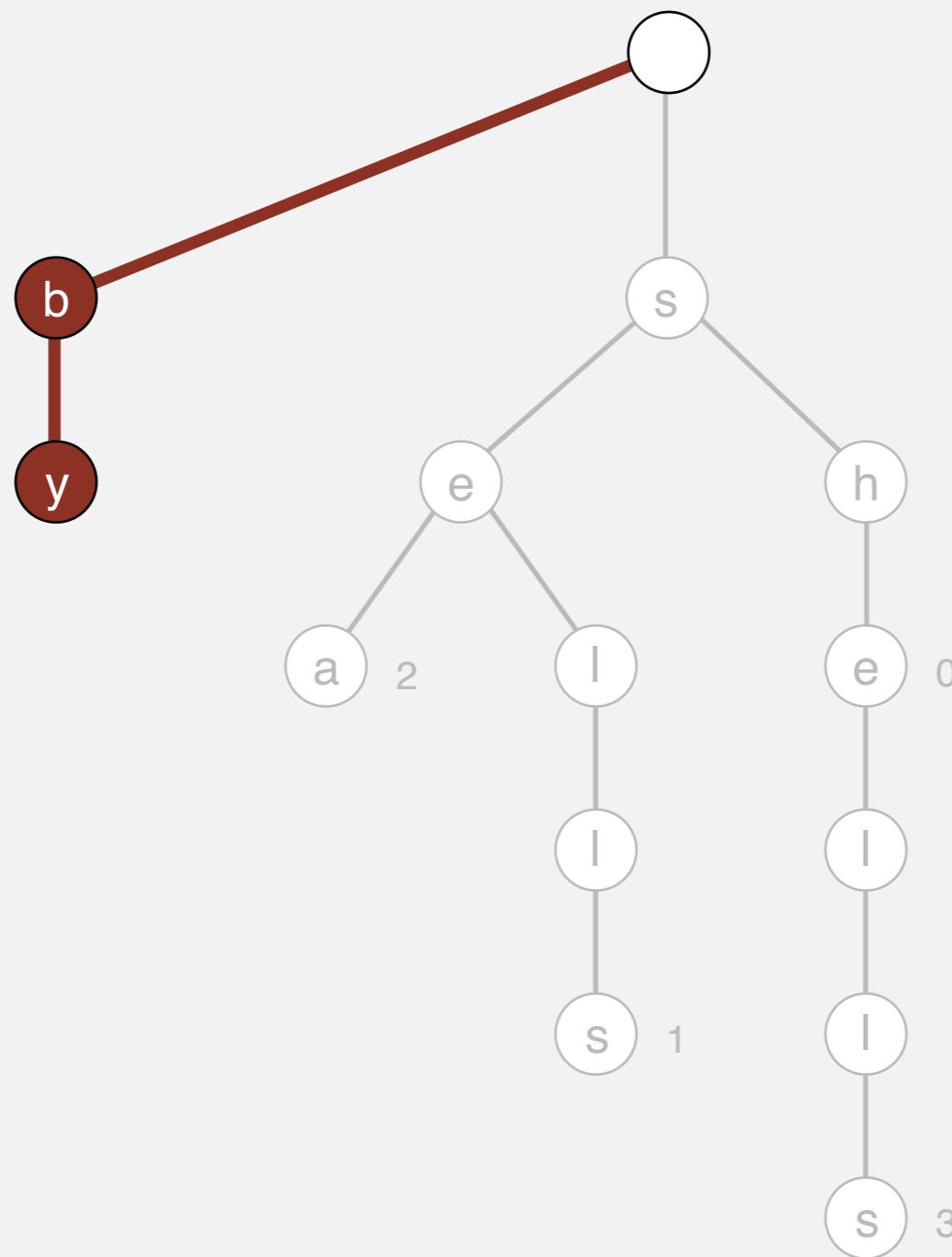
Trie construction demo

put("by", 4)



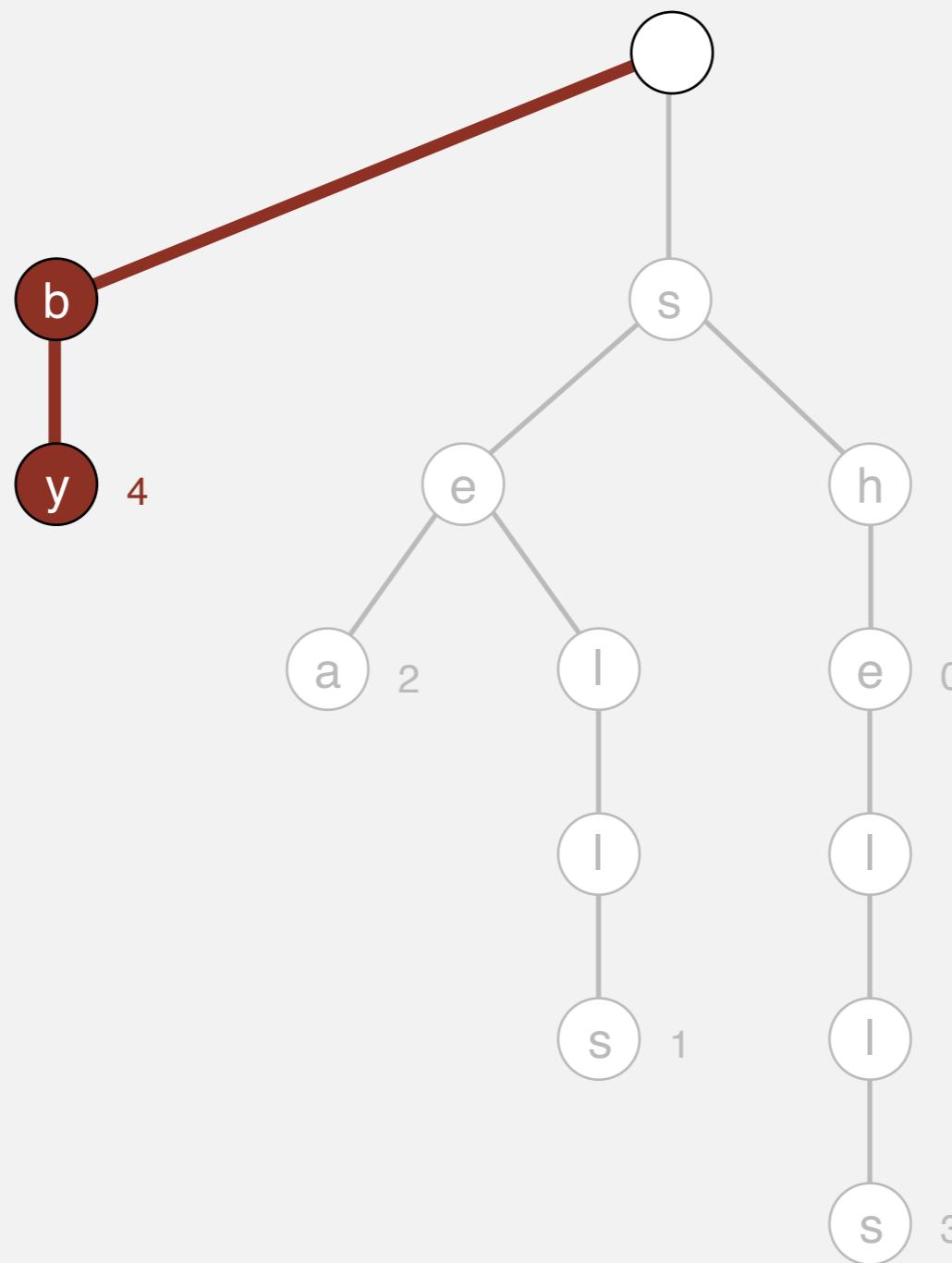
Trie construction demo

put("by", 4)



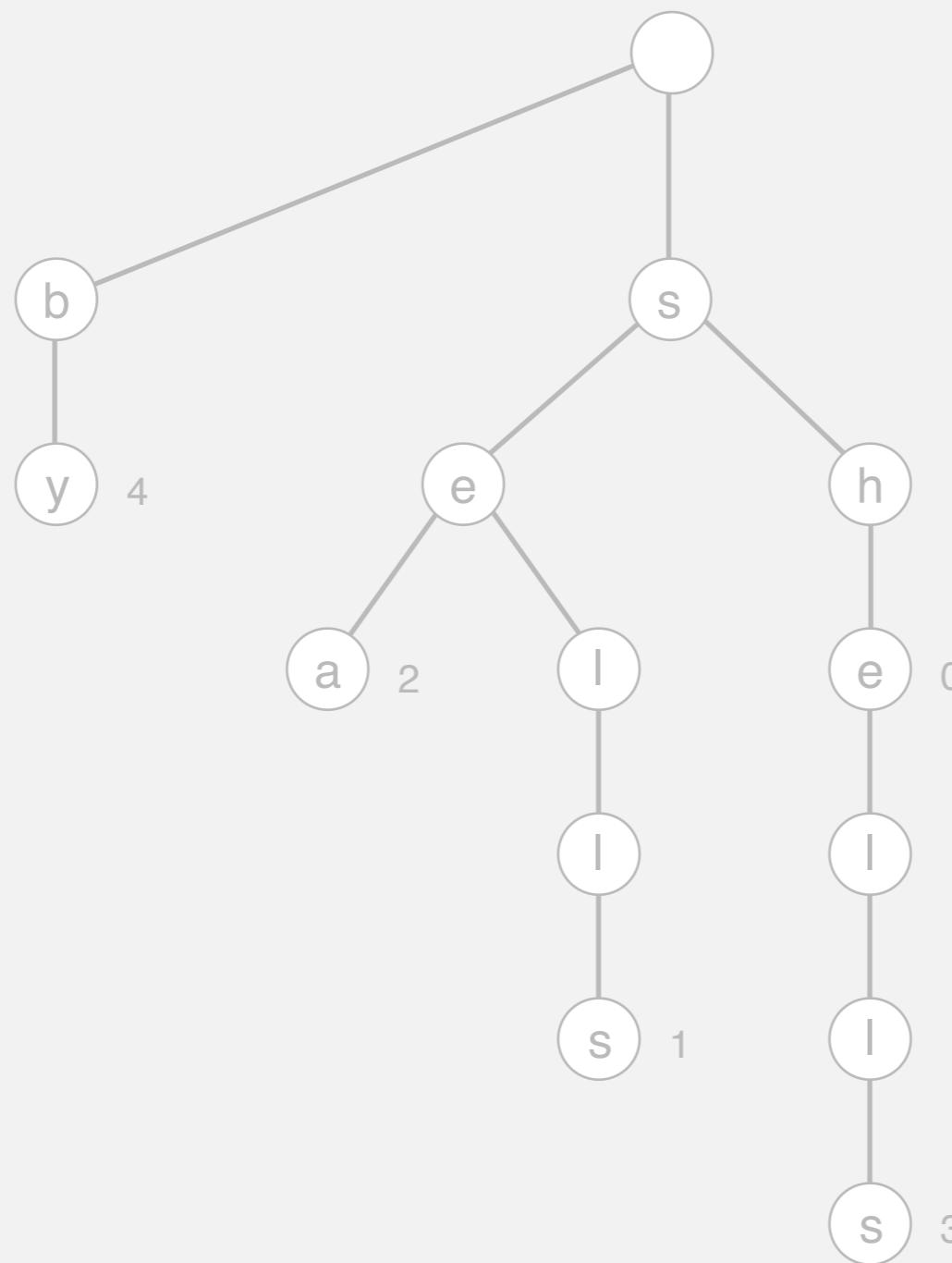
Trie construction demo

`put("by", 4)`



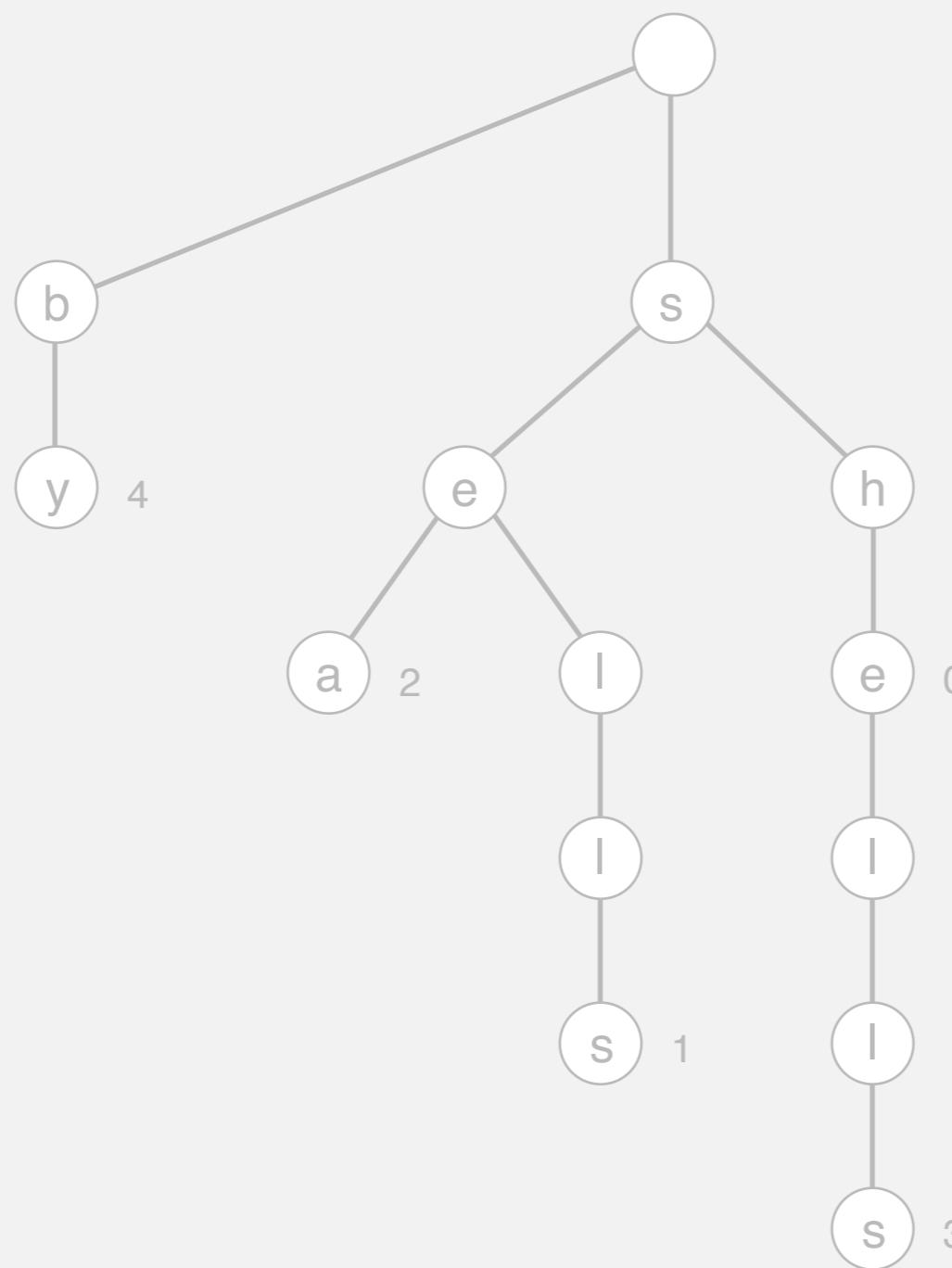
Trie construction demo

trie



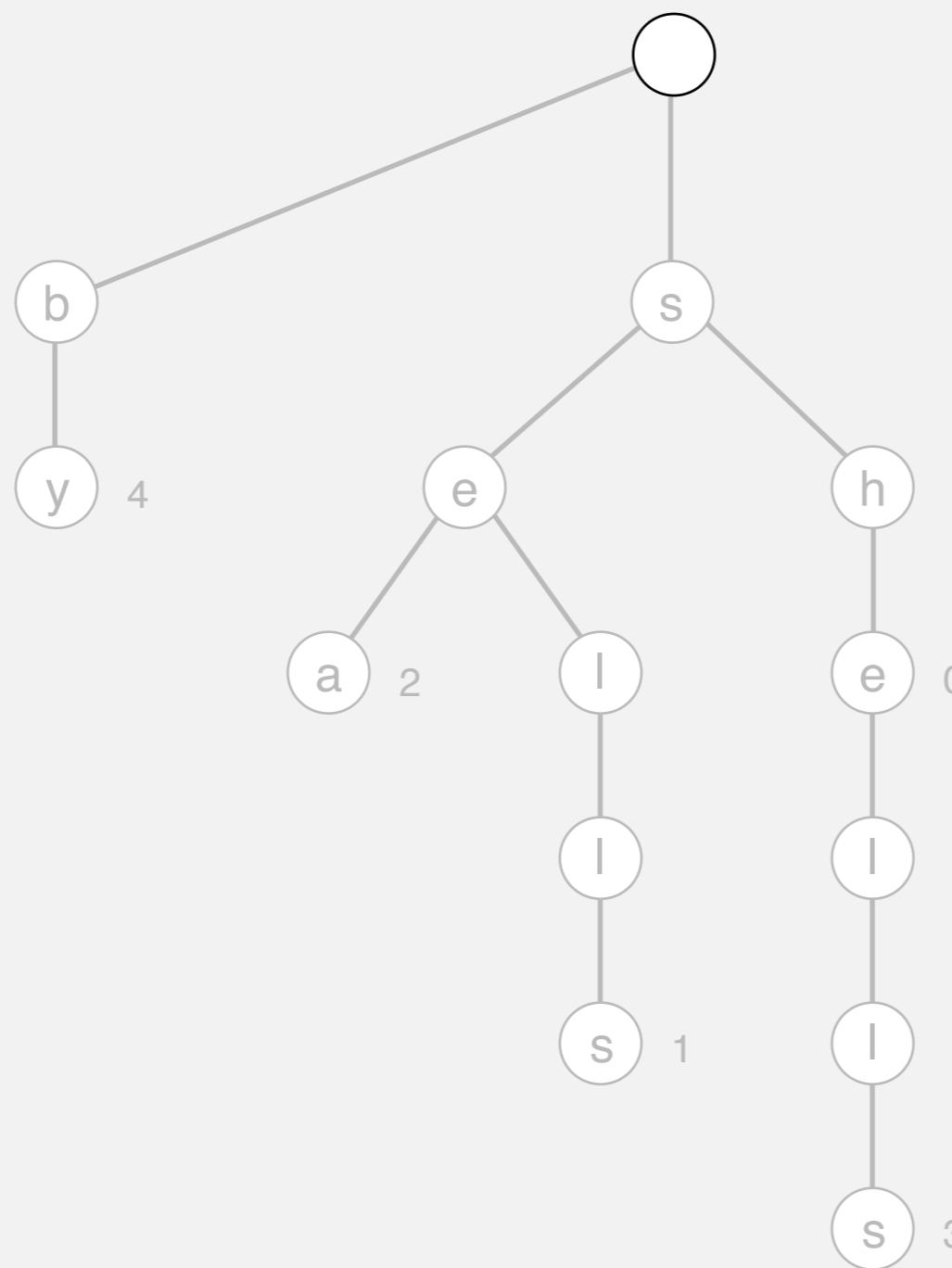
Trie construction demo

put("the", 5)



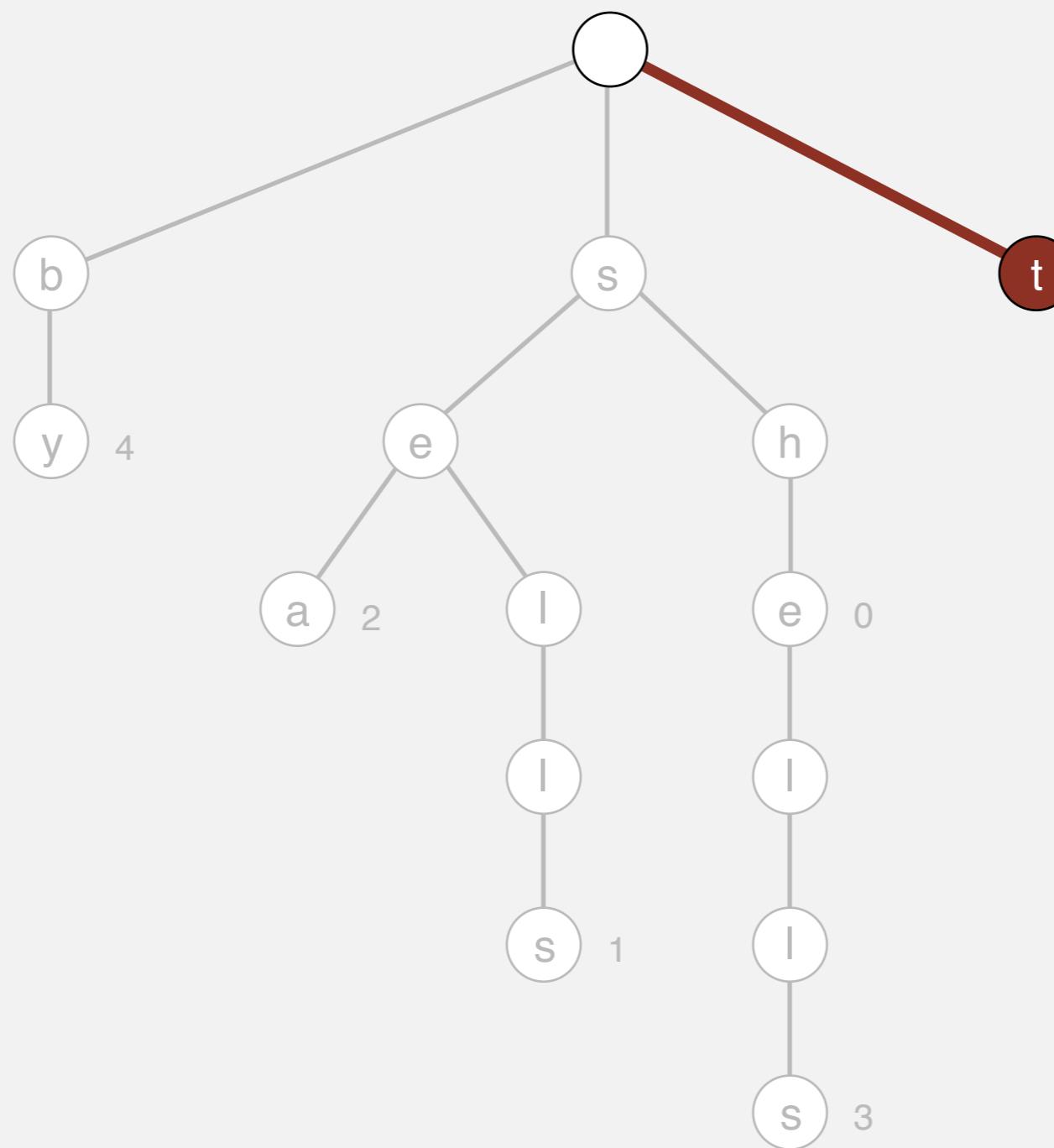
Trie construction demo

put("the", 5)



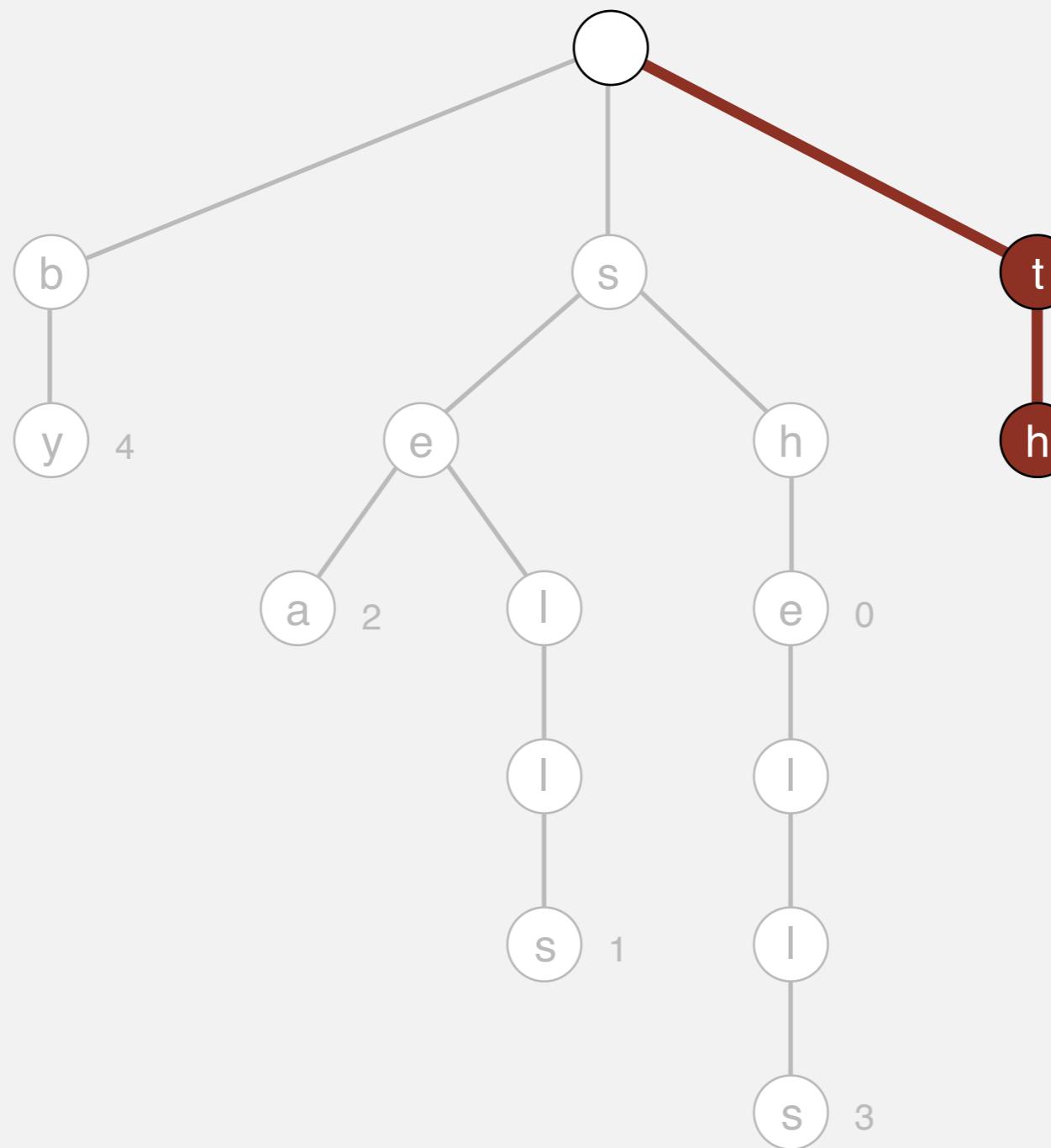
Trie construction demo

`put("the", 5)`



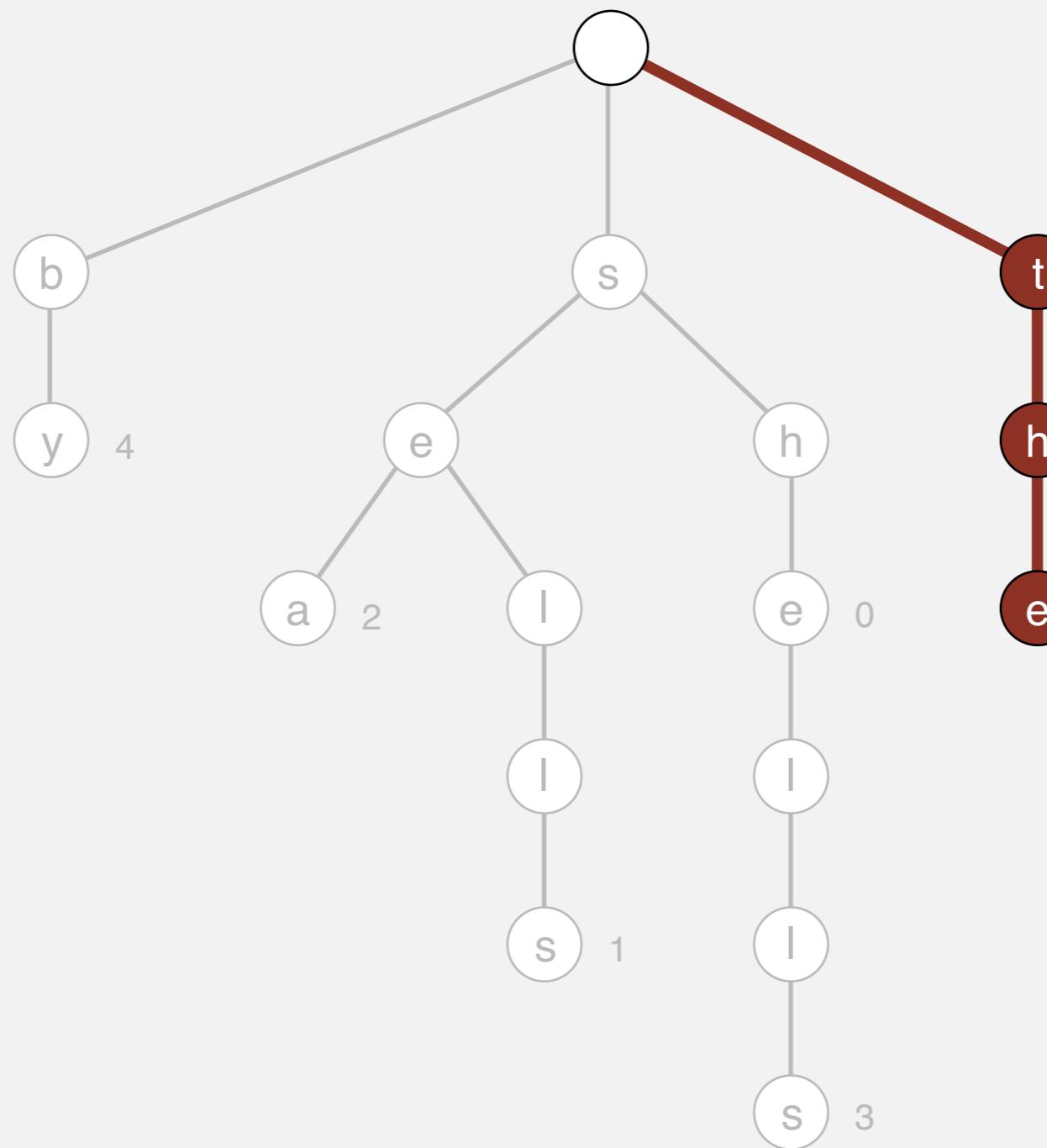
Trie construction demo

put("the", 5)



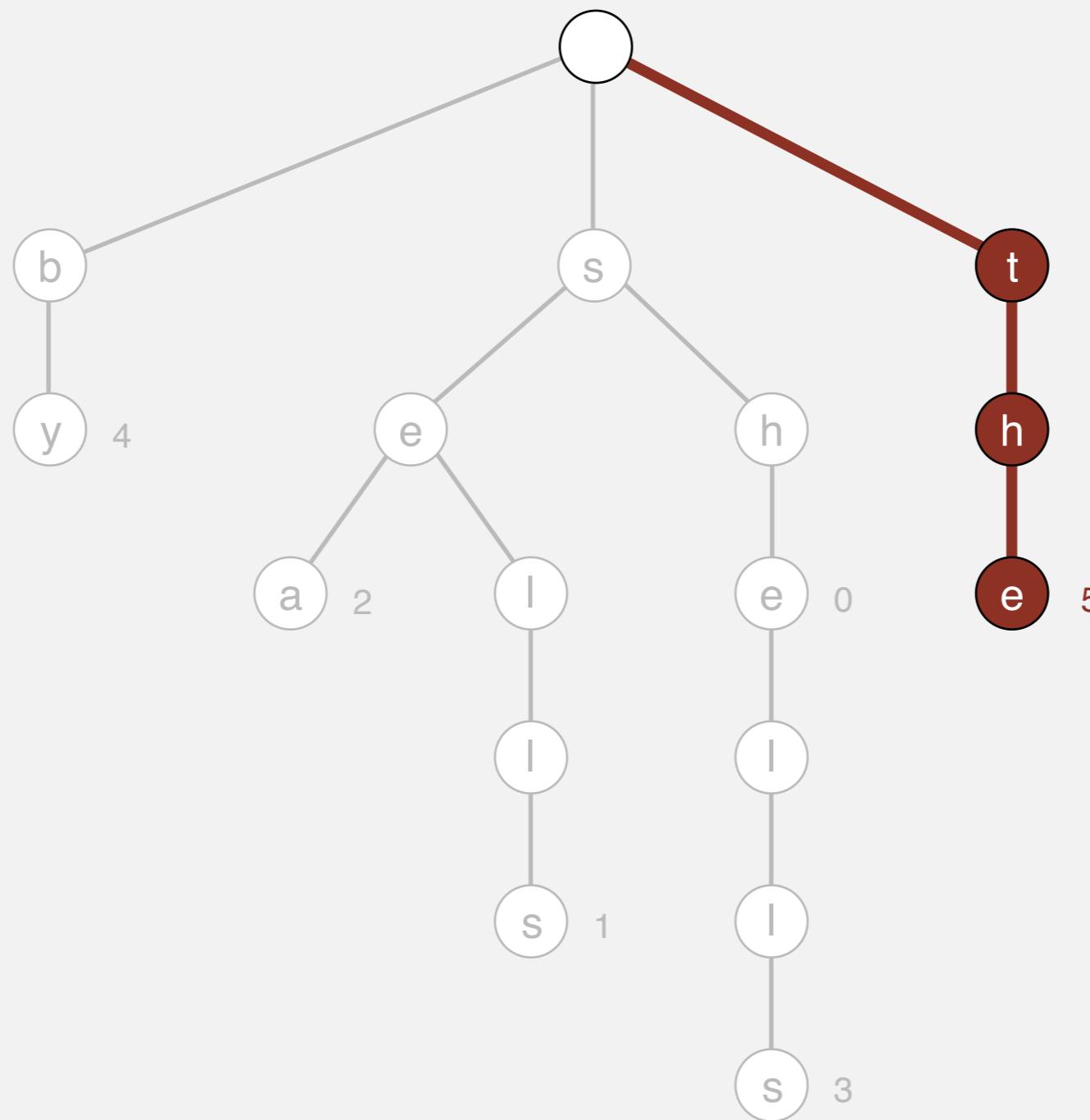
Trie construction demo

`put("the", 5)`



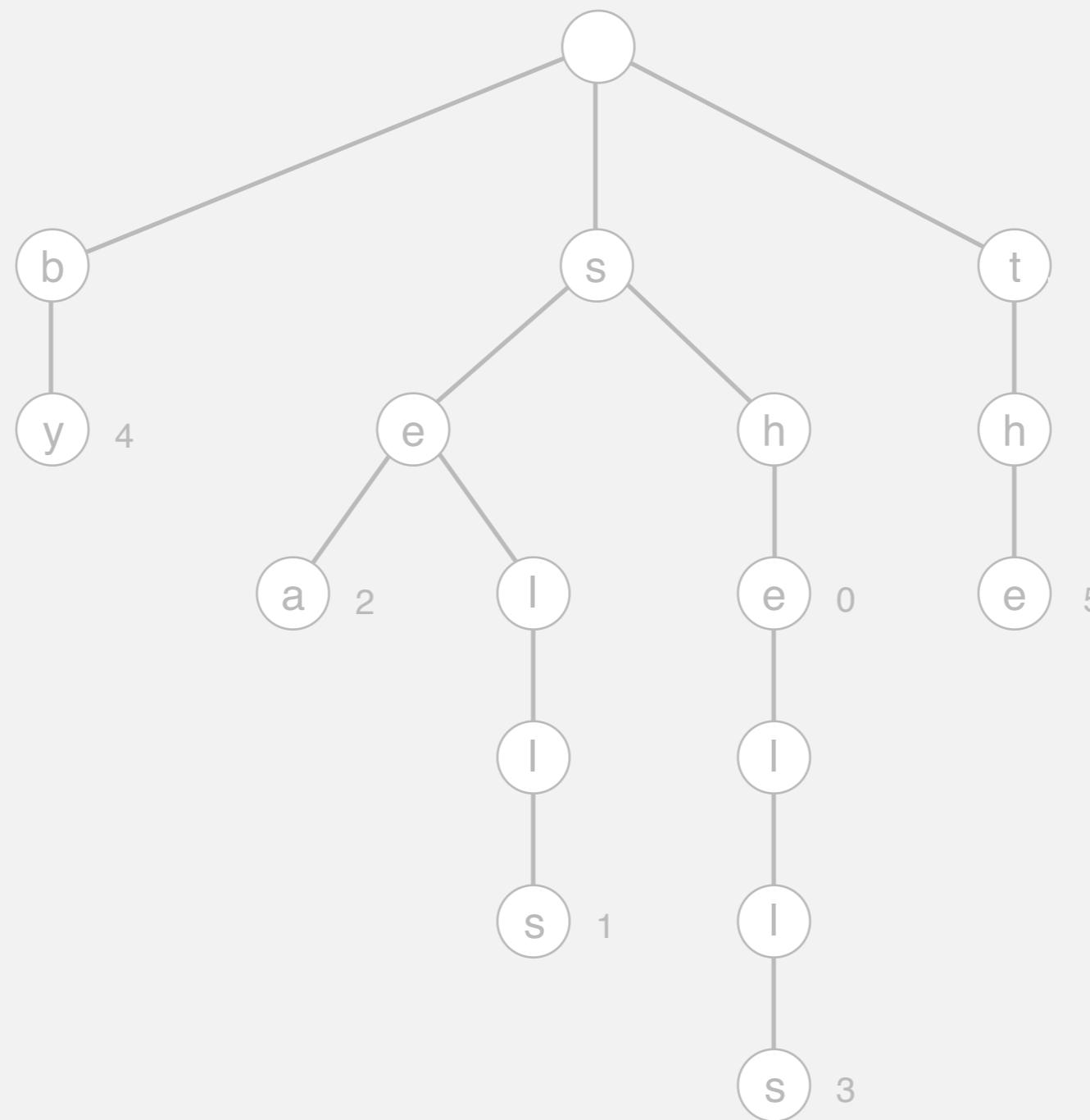
Trie construction demo

`put("the", 5)`



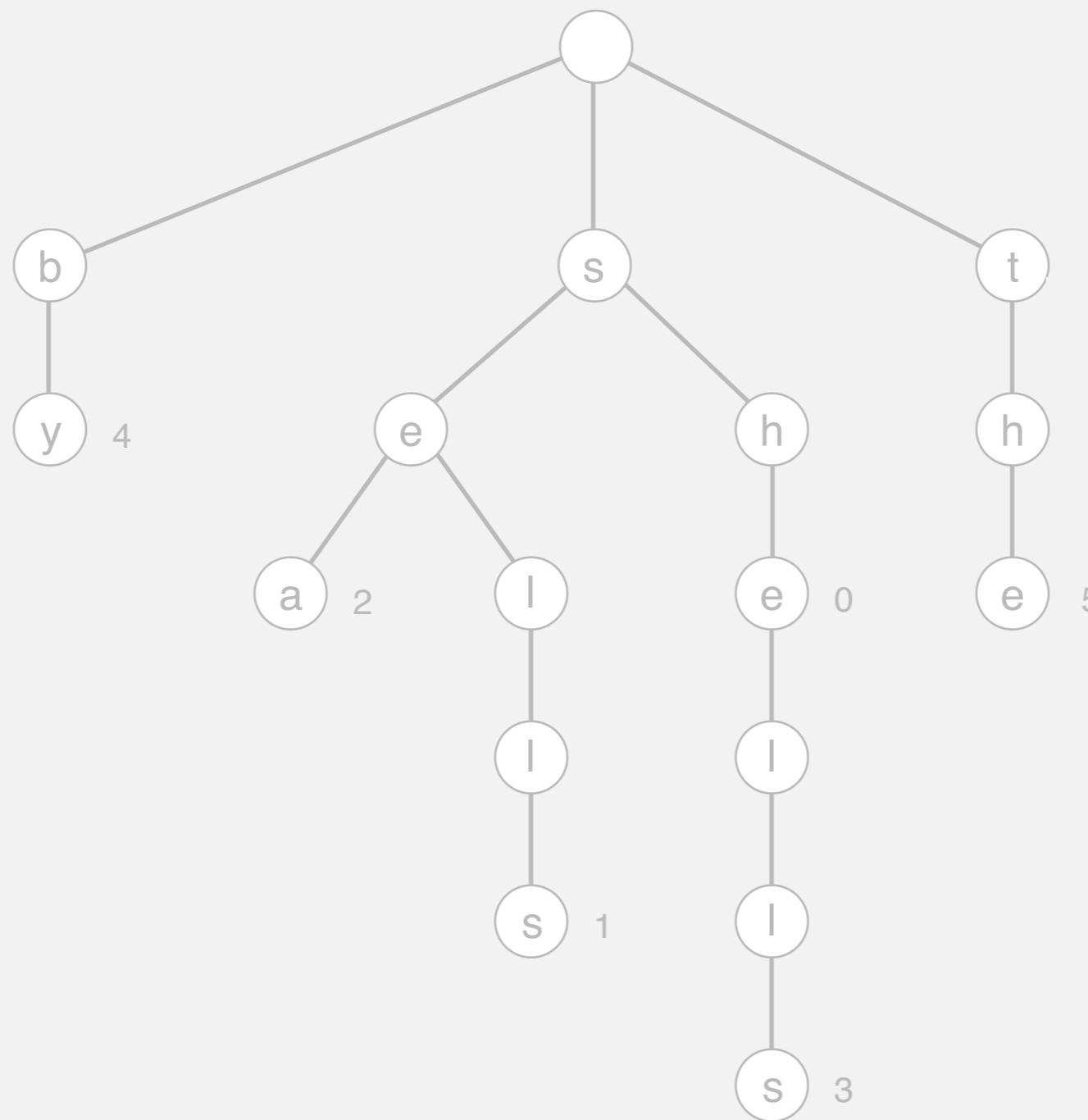
Trie construction demo

trie



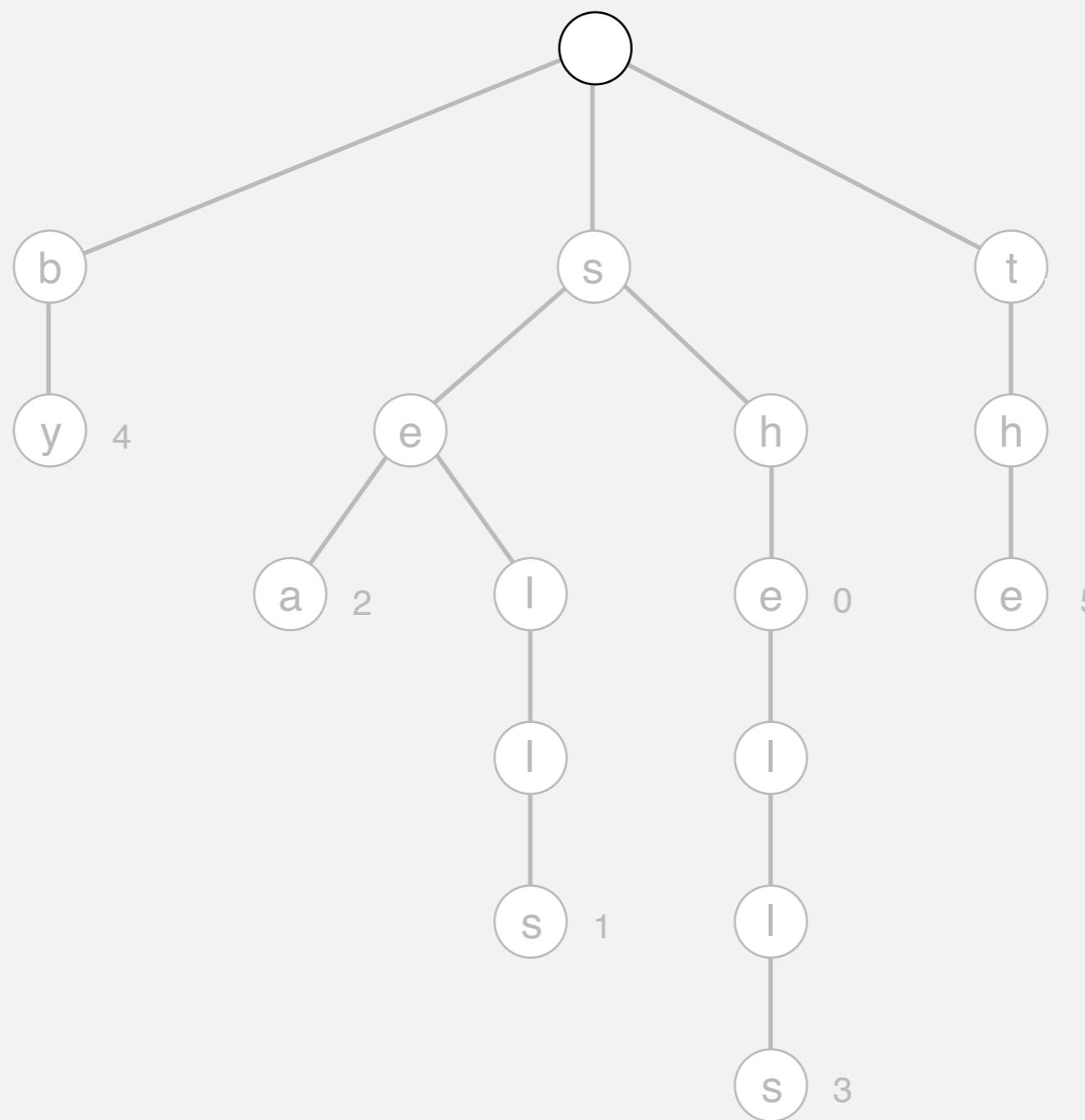
Trie construction demo

put("sea", 6)



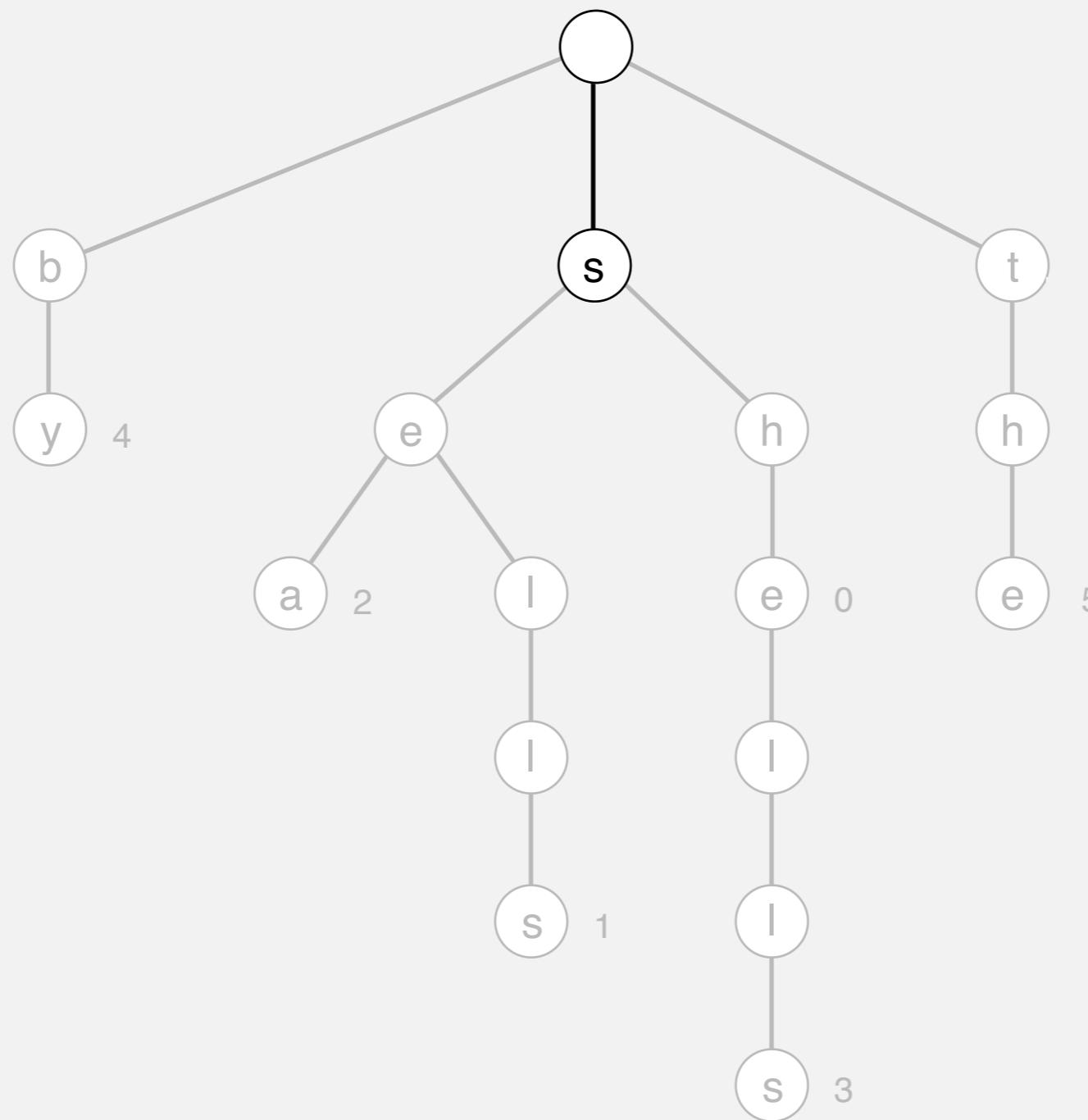
Trie construction demo

put("sea", 6)



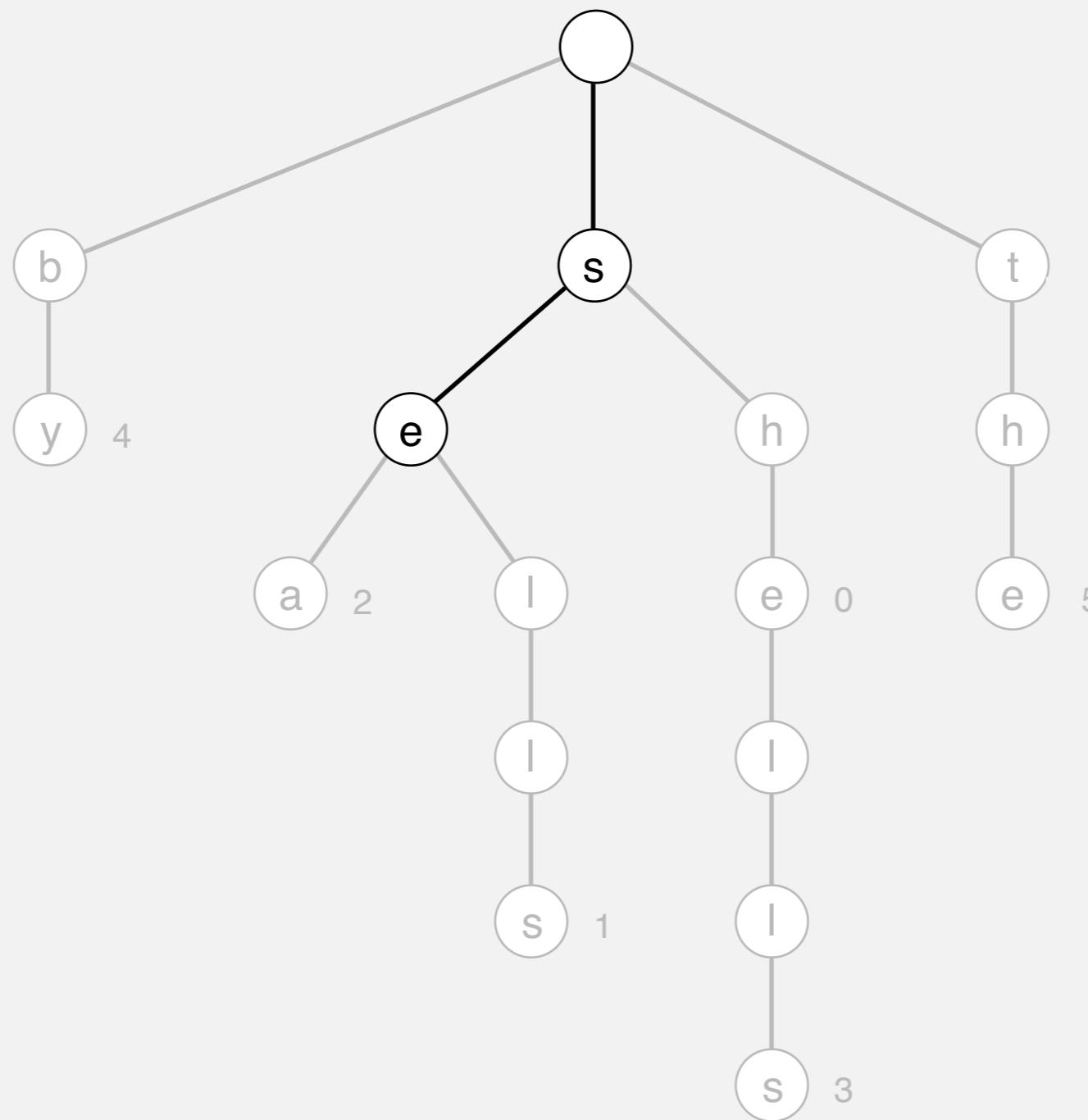
Trie construction demo

put("sea", 6)



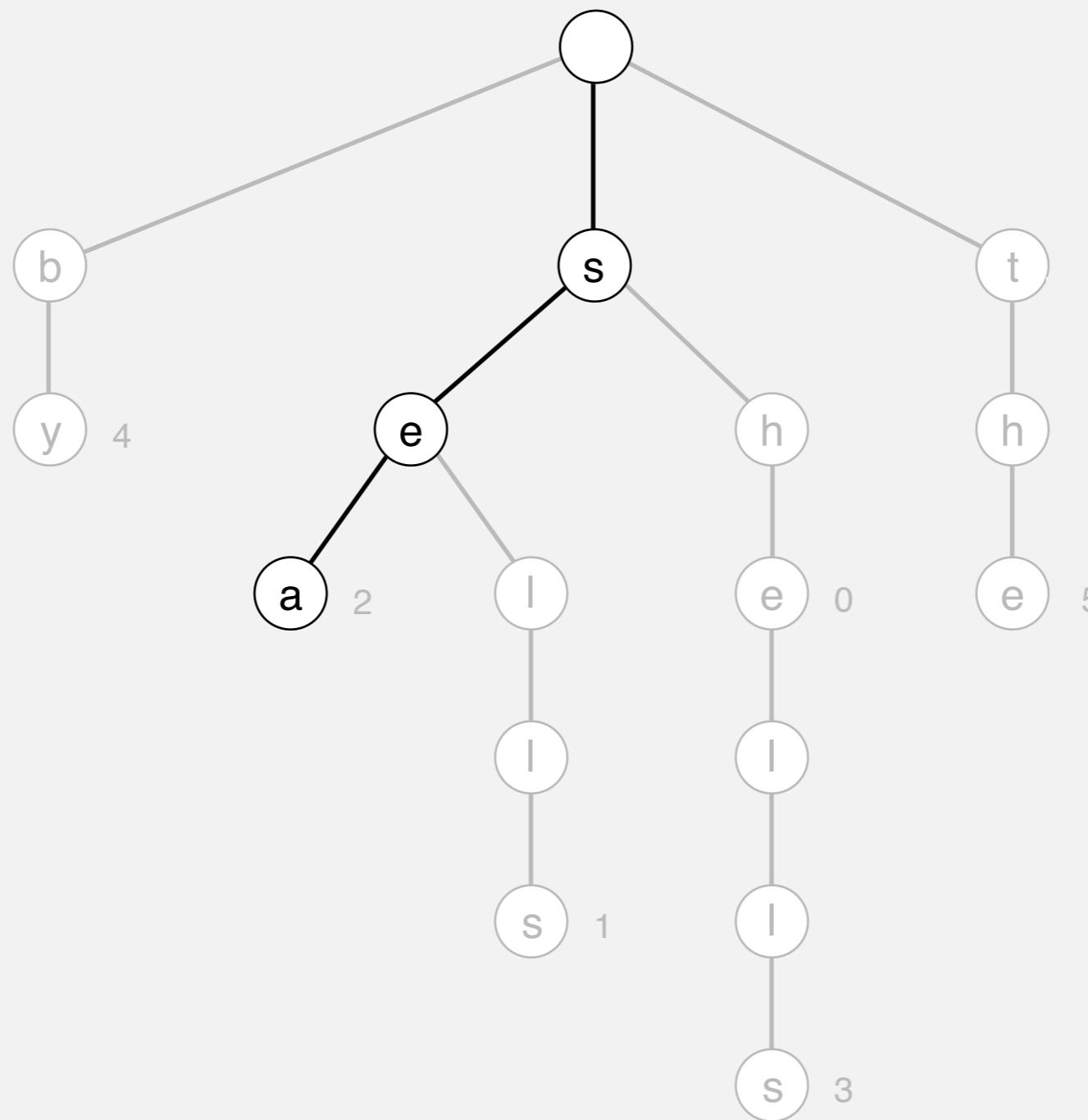
Trie construction demo

put("sea", 6)



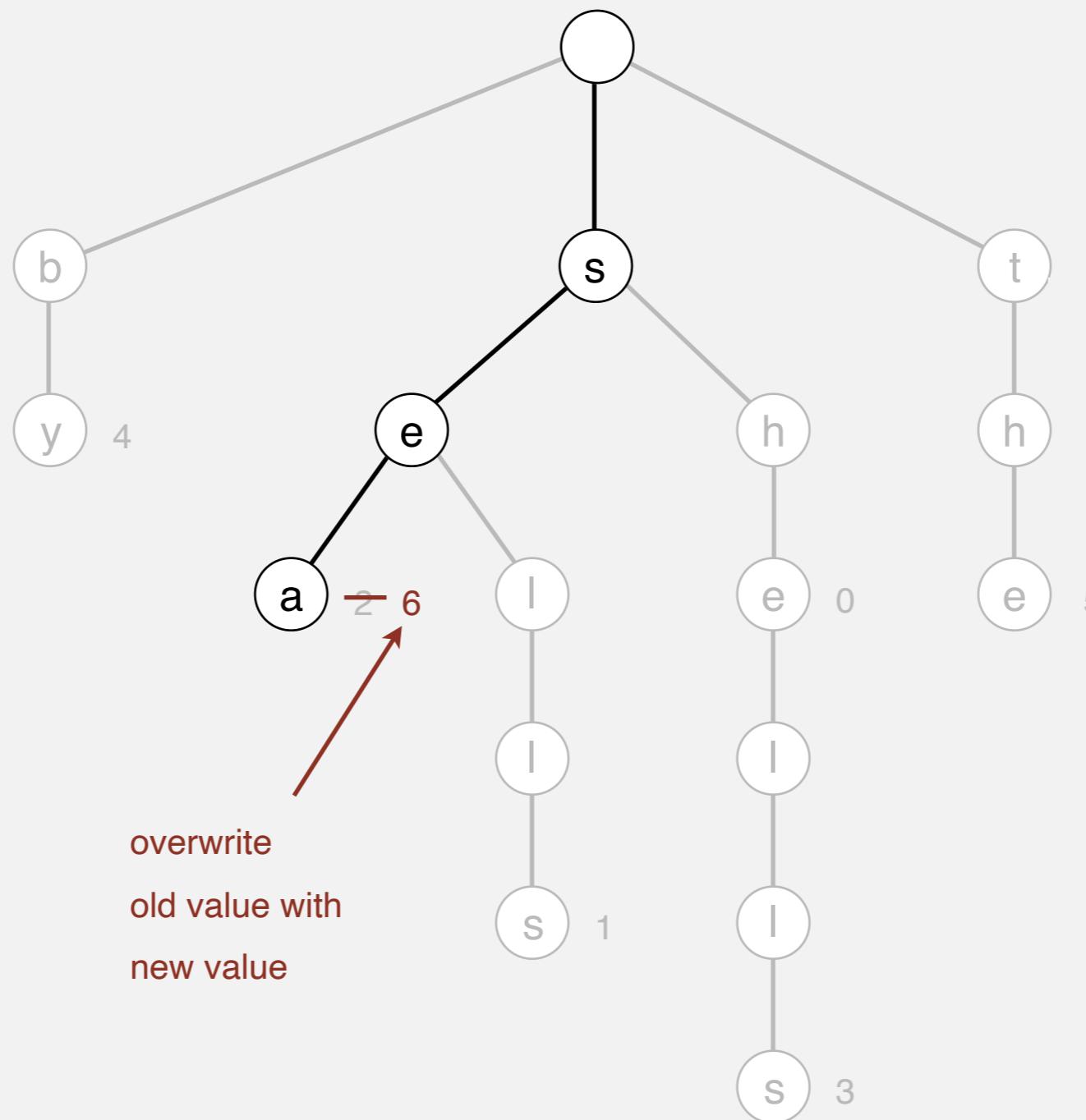
Trie construction demo

put("sea", 6)



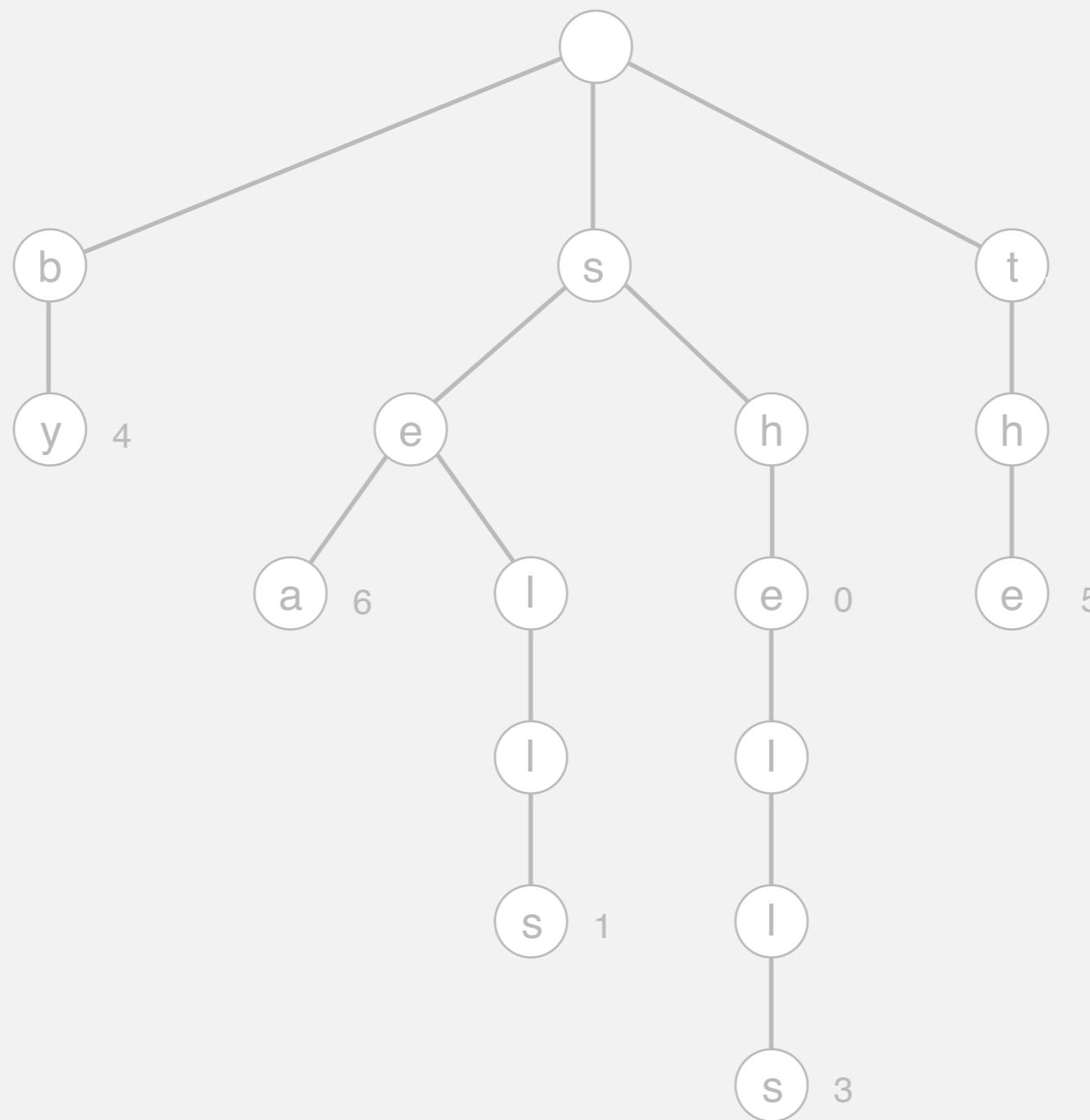
Trie construction demo

put("sea", 6)



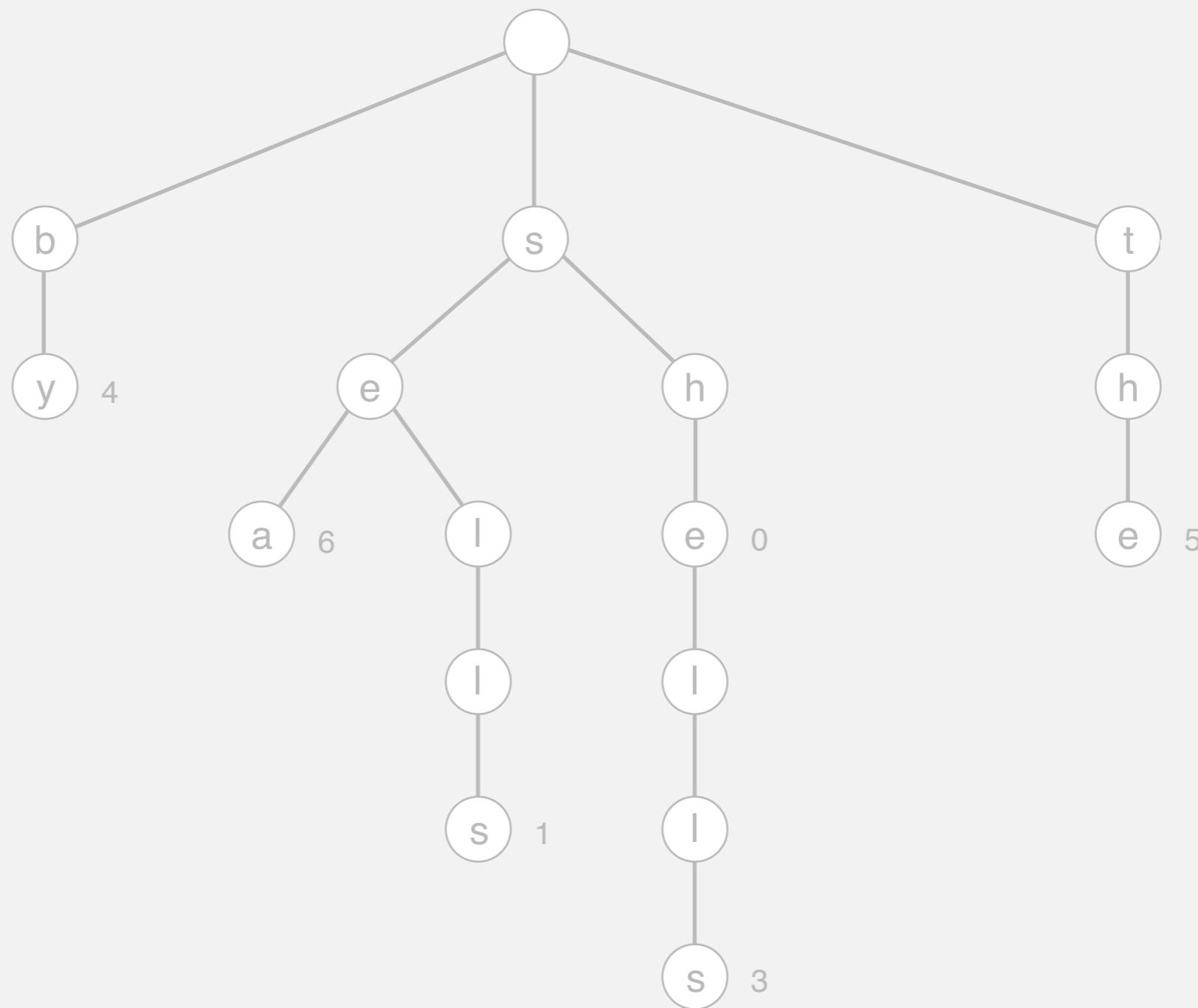
Trie construction demo

trie



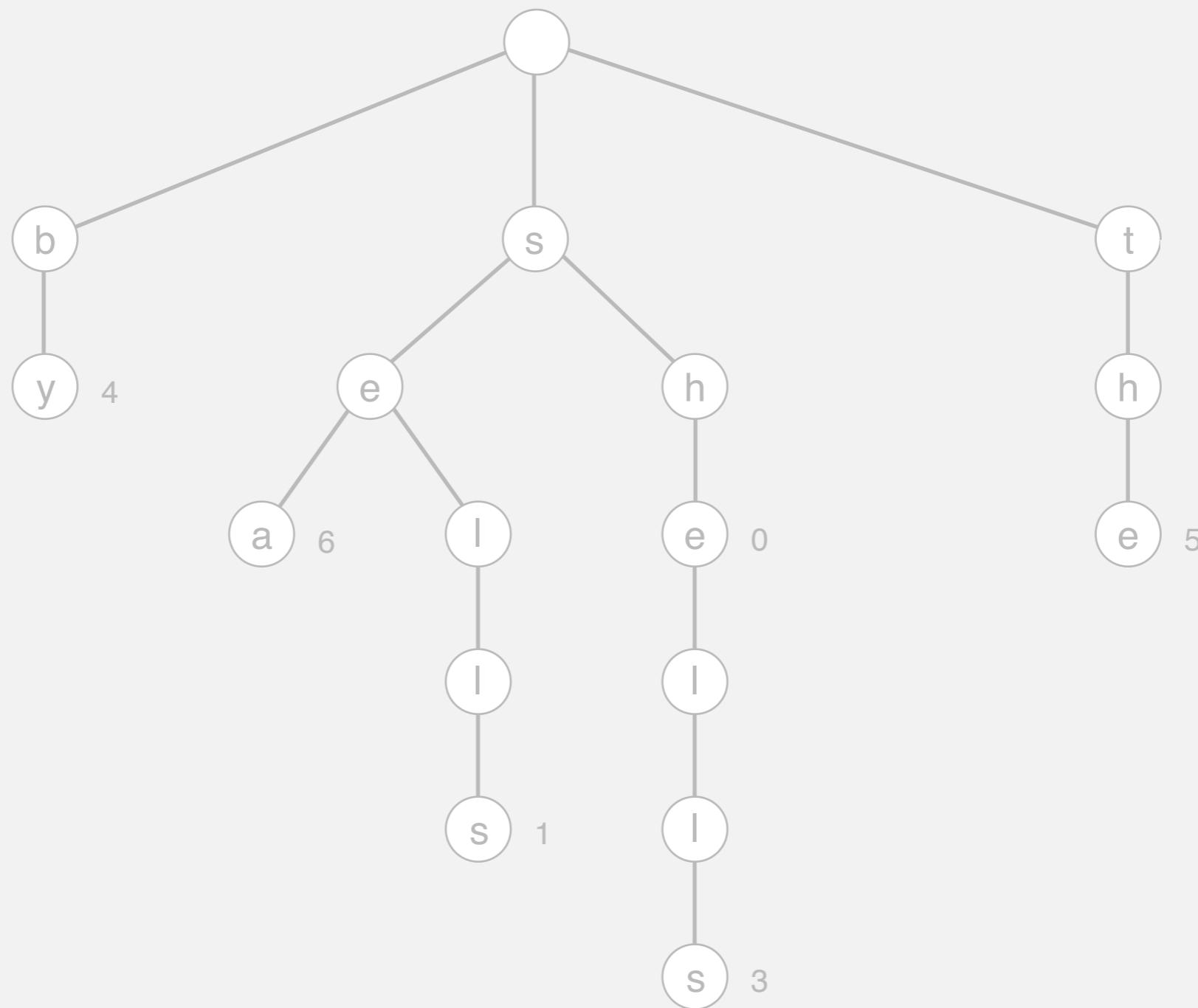
Trie construction demo

trie



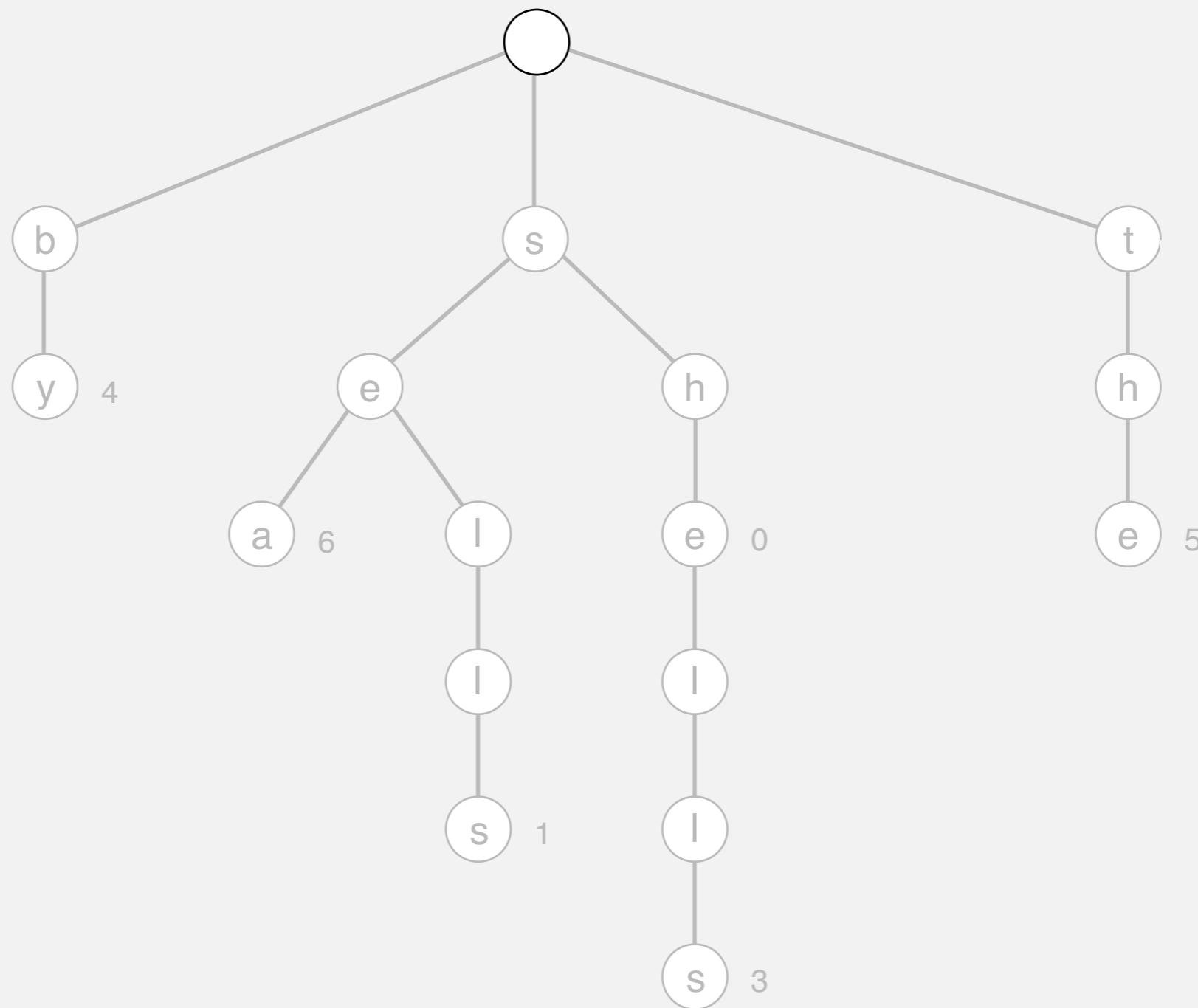
Trie construction demo

put("shore", 7)



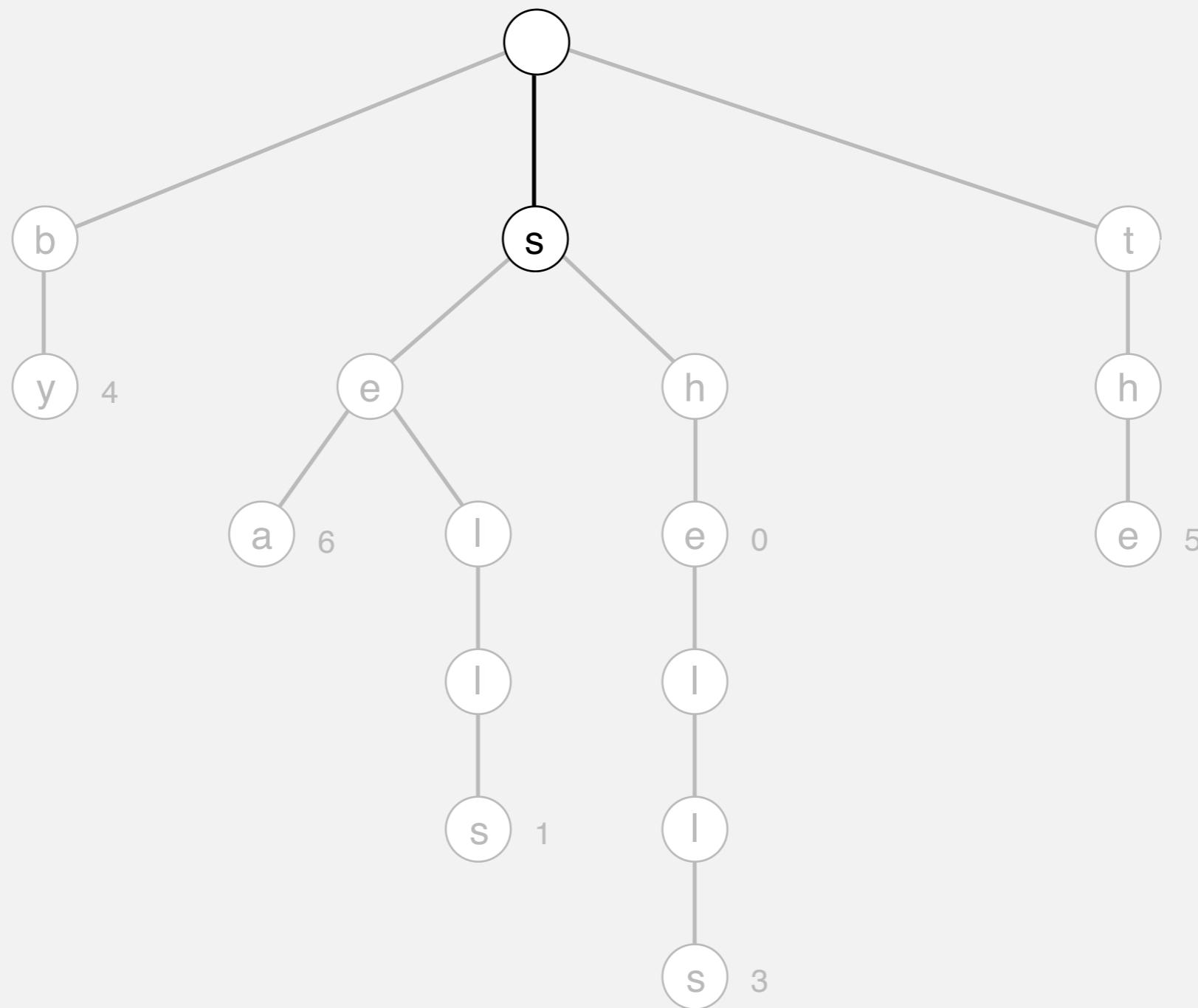
Trie construction demo

put("shore", 7)



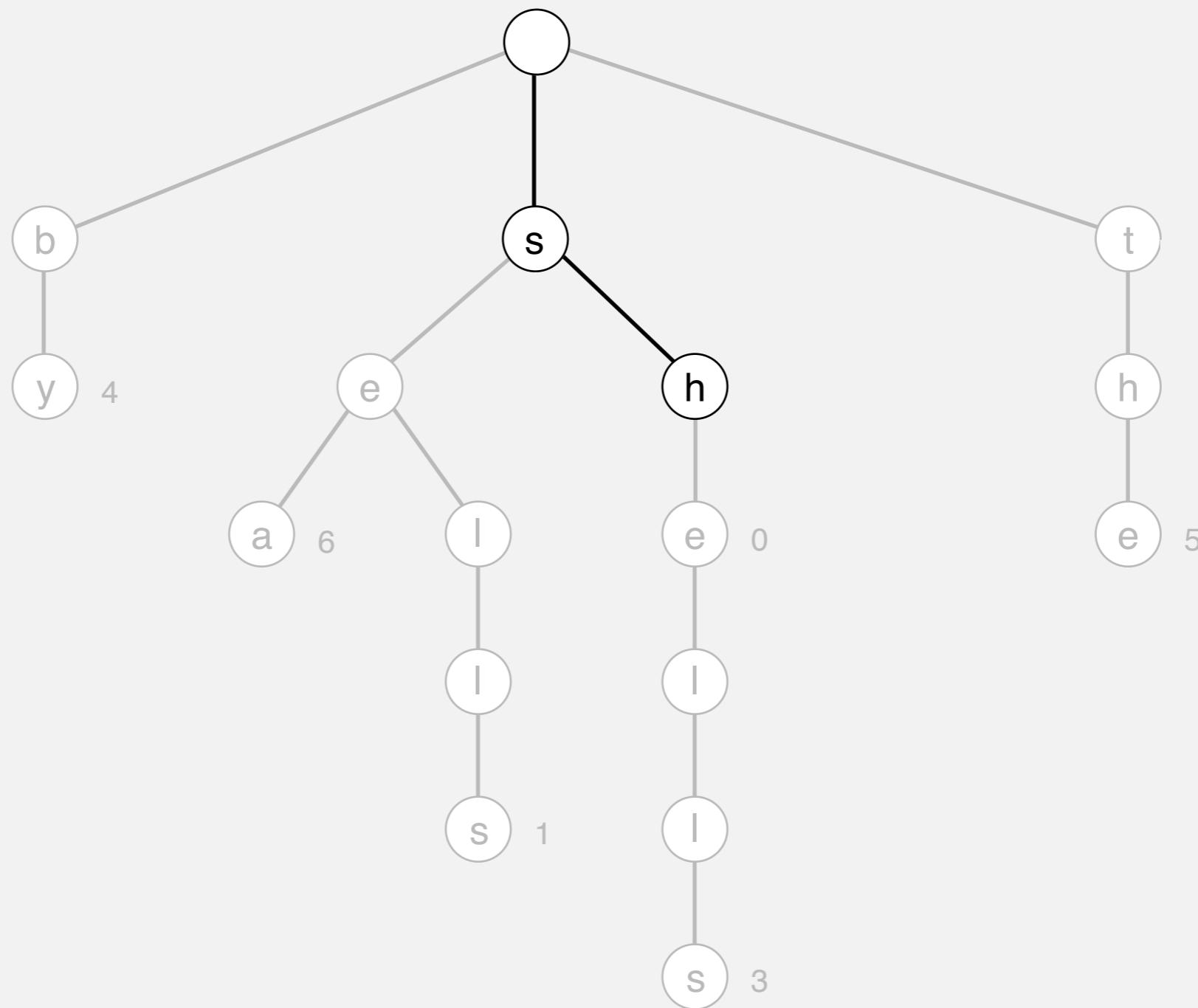
Trie construction demo

put("shore", 7)



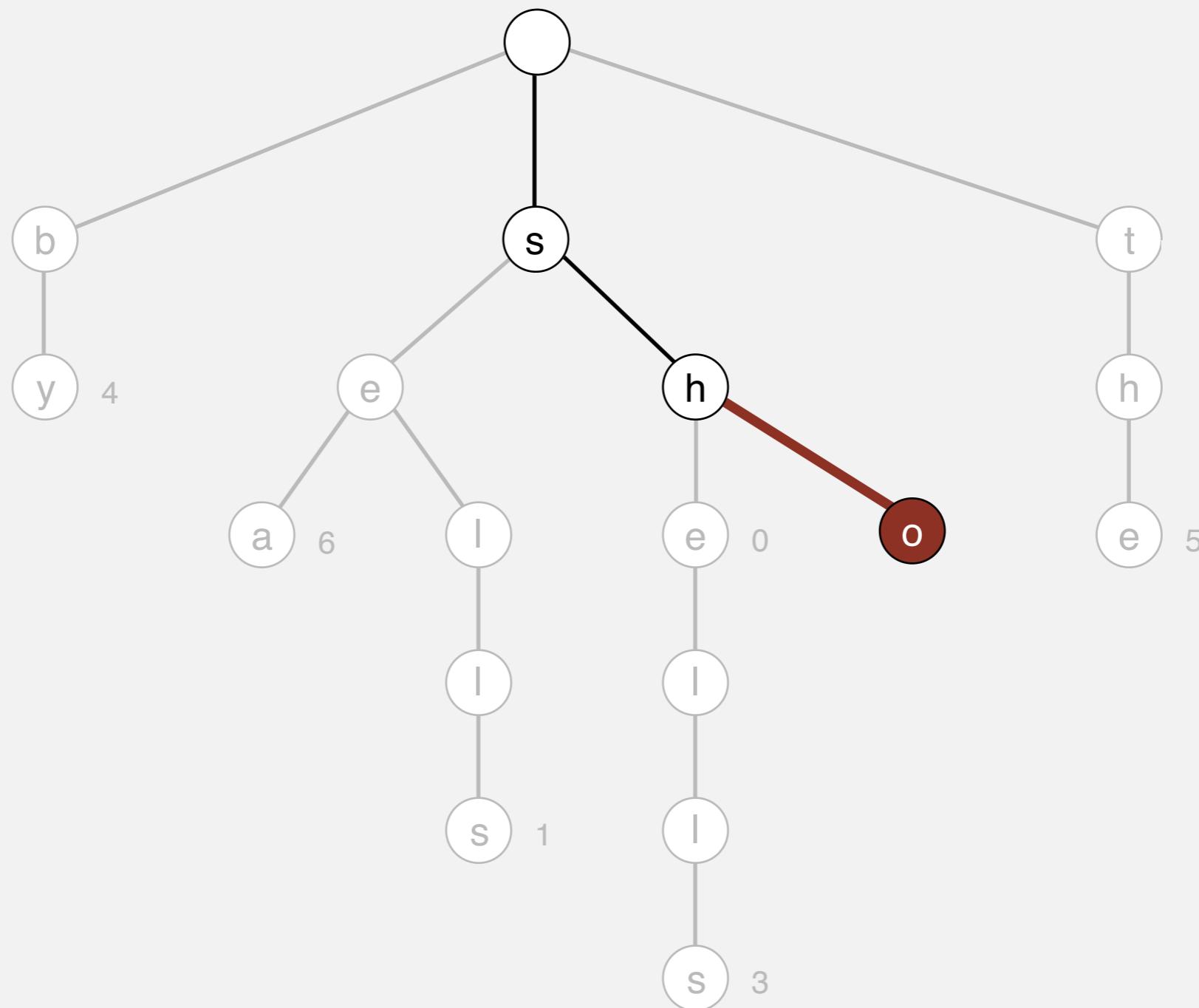
Trie construction demo

put("shore", 7)



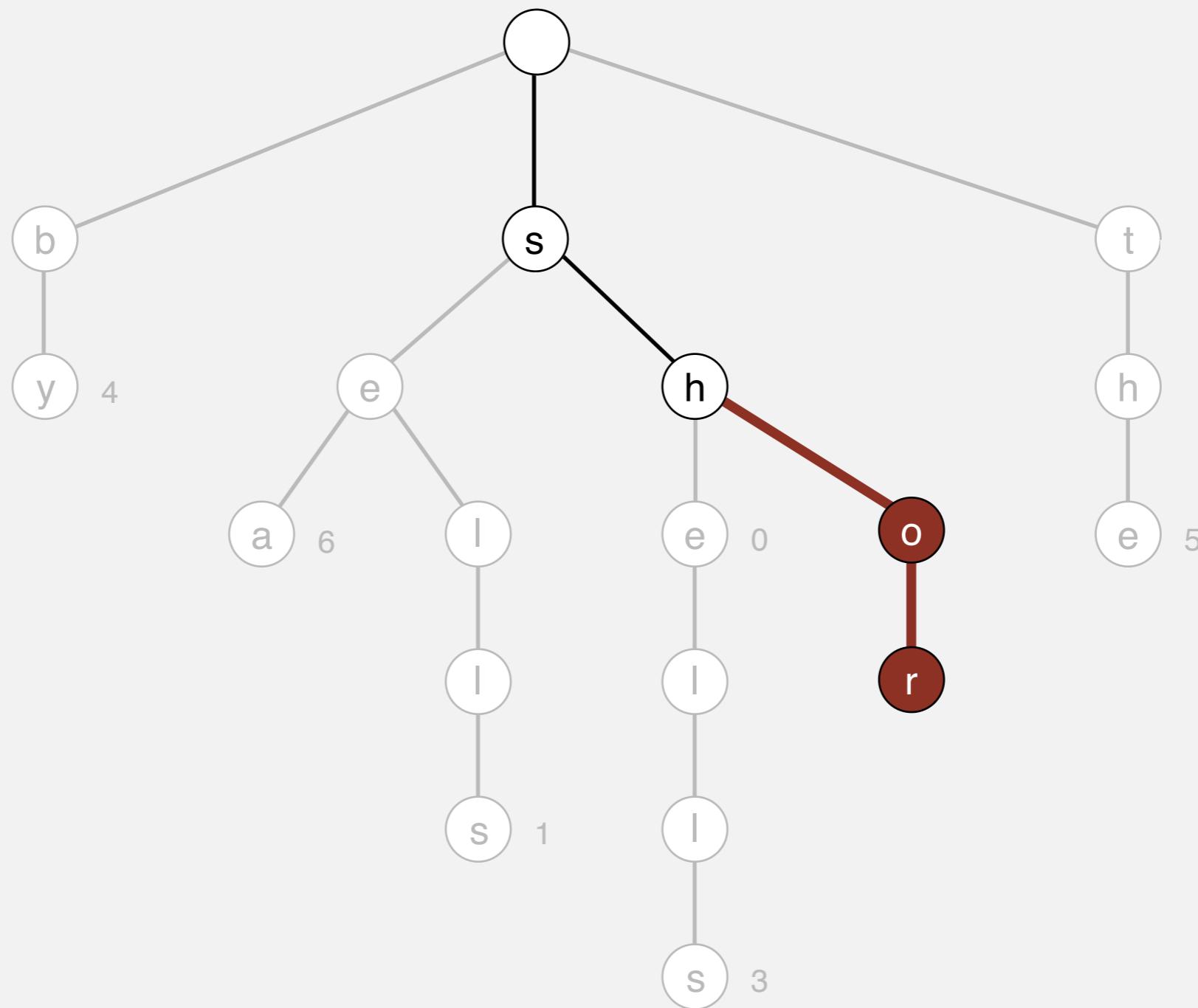
Trie construction demo

`put("shore", 7)`



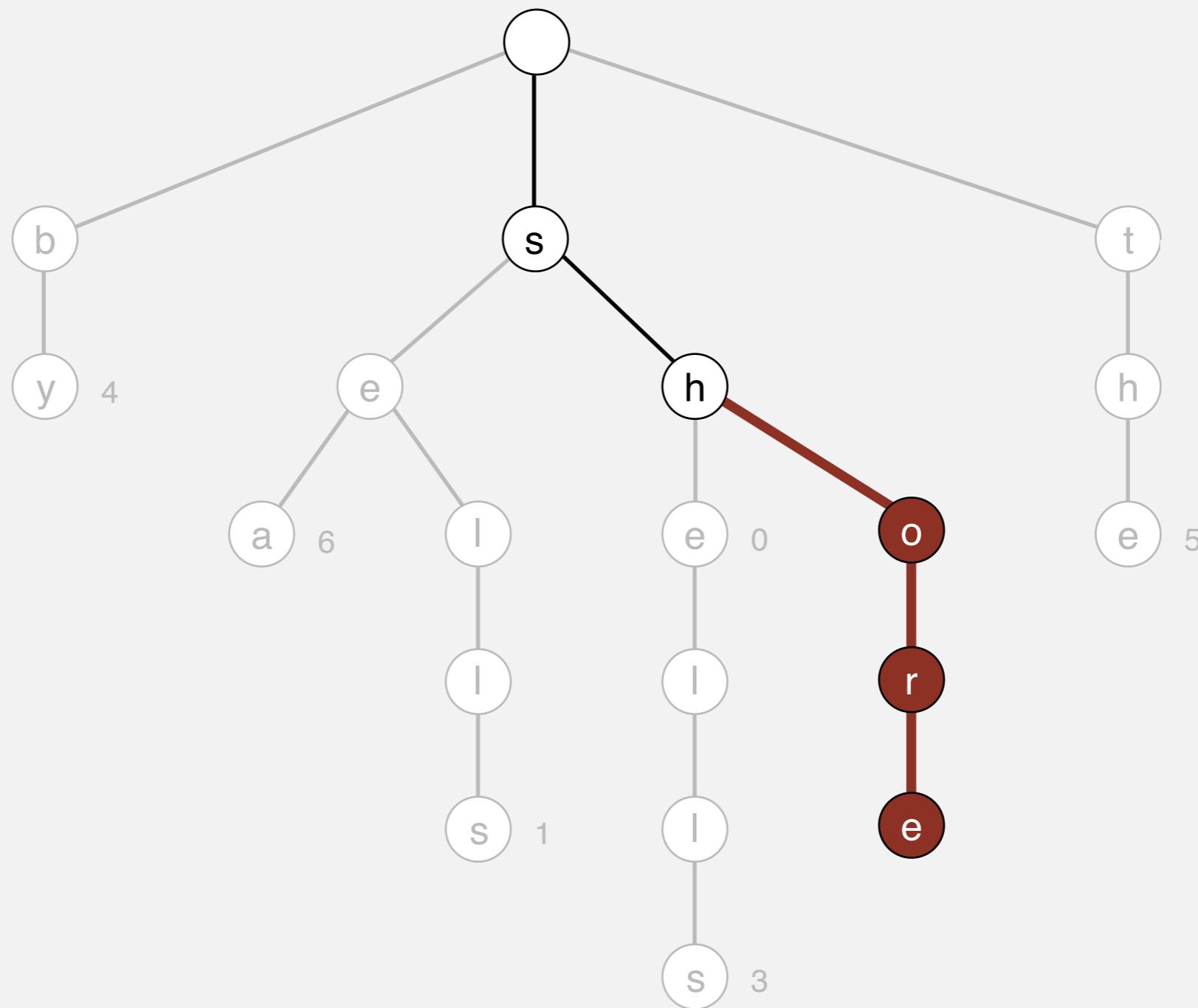
Trie construction demo

`put("shore", 7)`



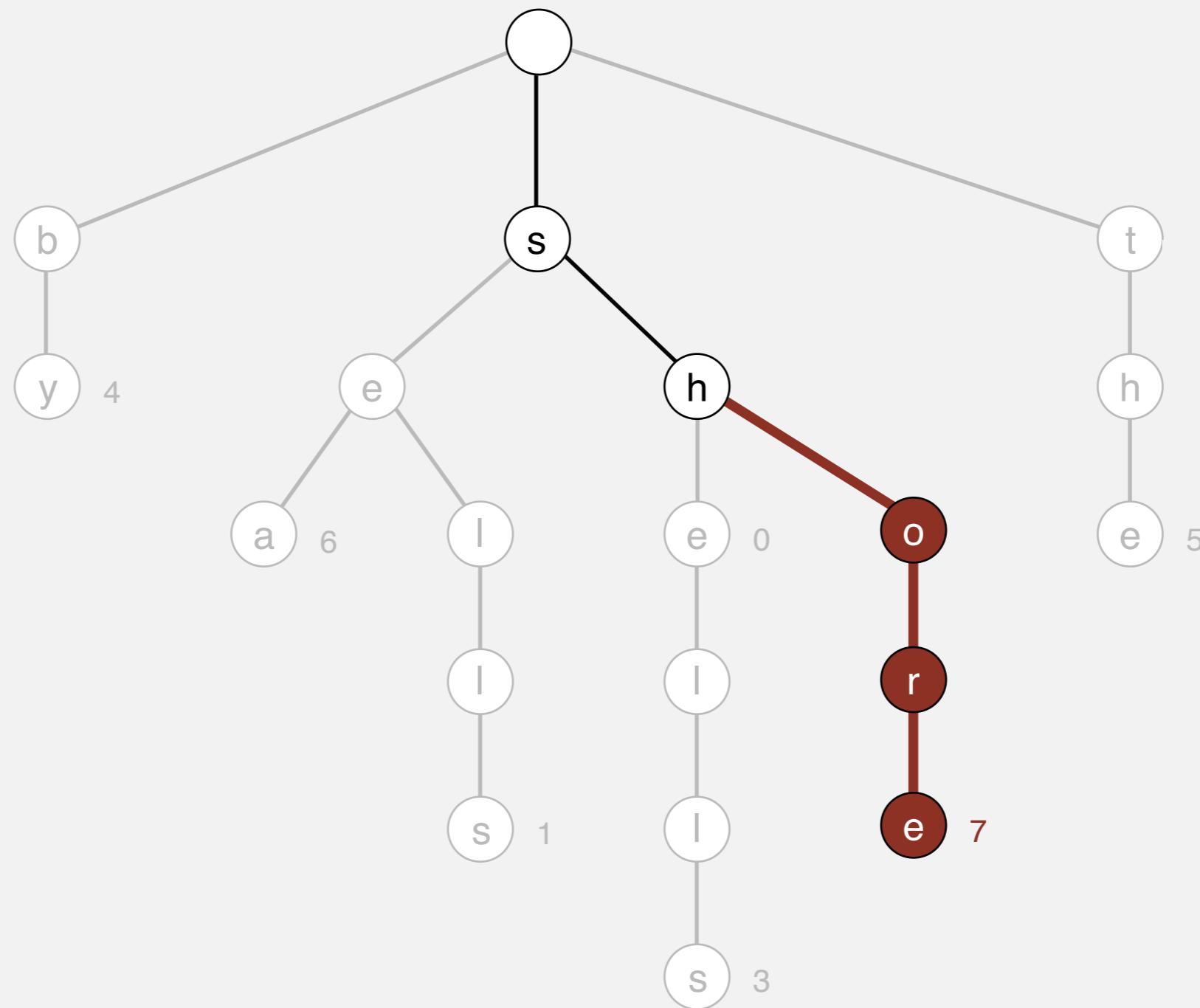
Trie construction demo

`put("shore", 7)`



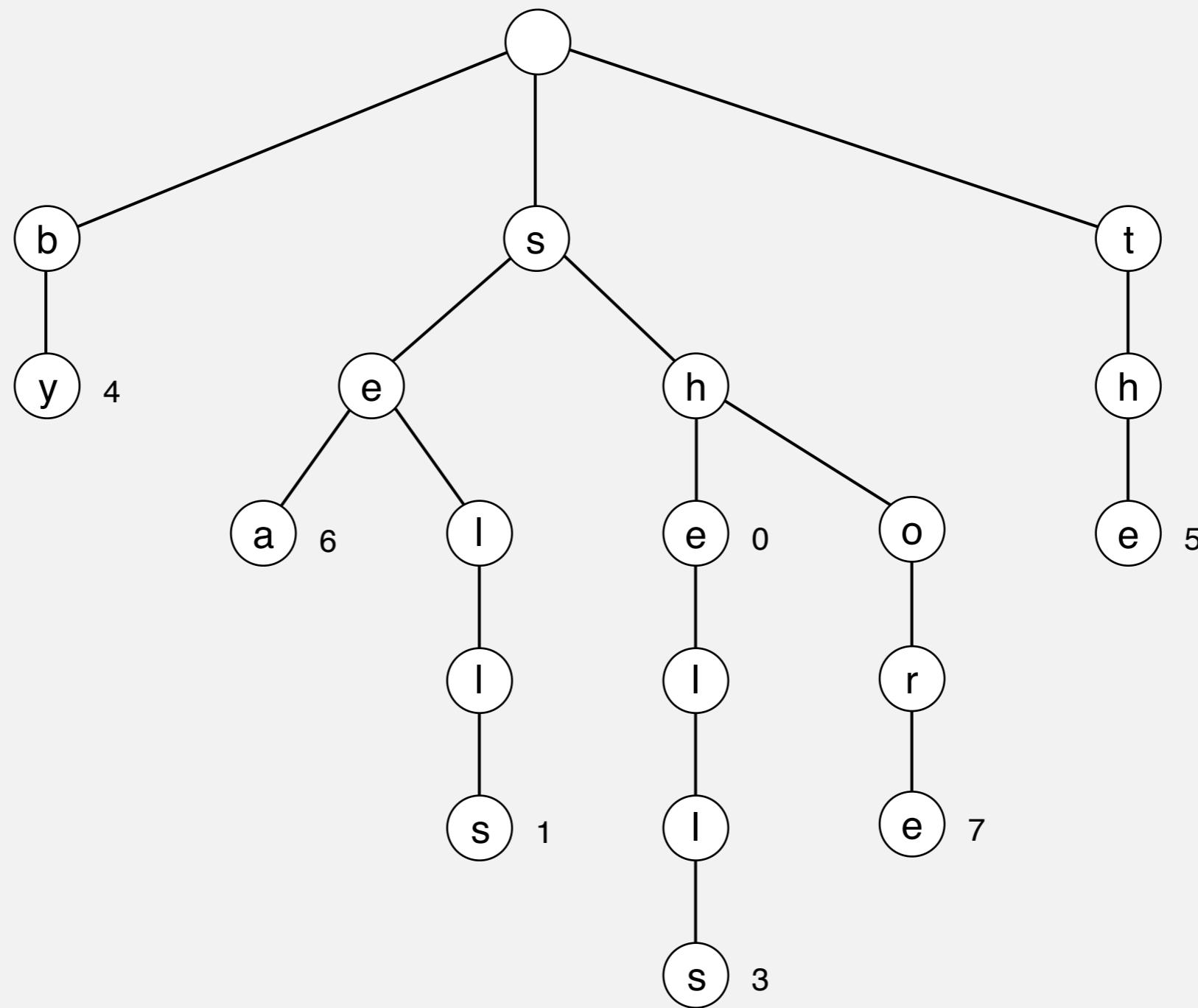
Trie construction demo

`put("shore", 7)`



Trie construction demo

trie

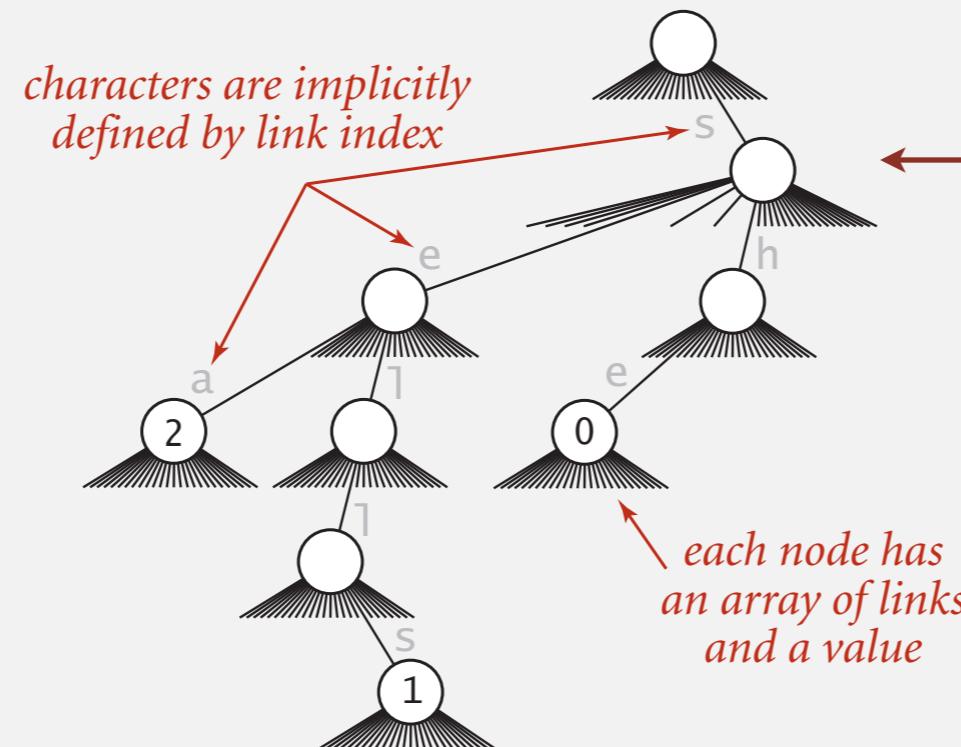
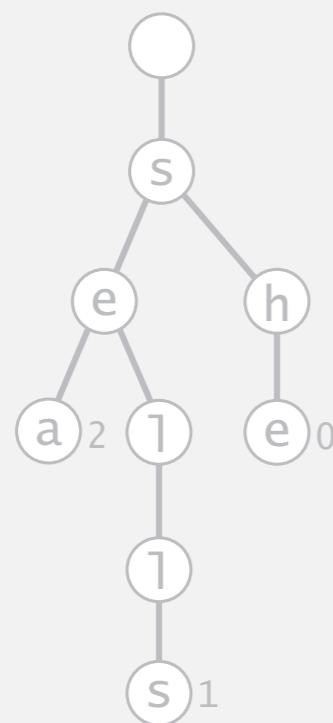


Trie representation: Java implementation

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



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;  
    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;  
    }
```

← extended ASCII

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);
}
```

```
}
```

Trie performance

Search hit. Need to examine all L characters for equality.

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).

Trie performance

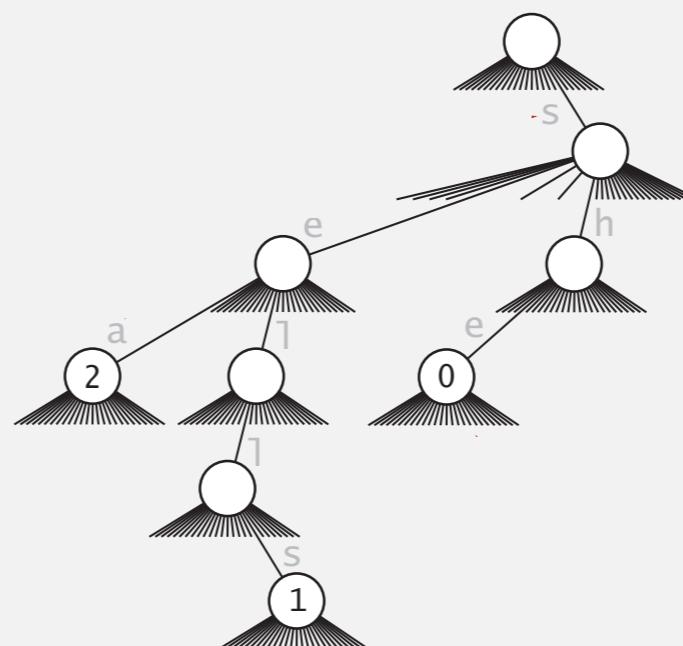
Search hit. Need to examine all L characters for equality.

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)



Trie performance

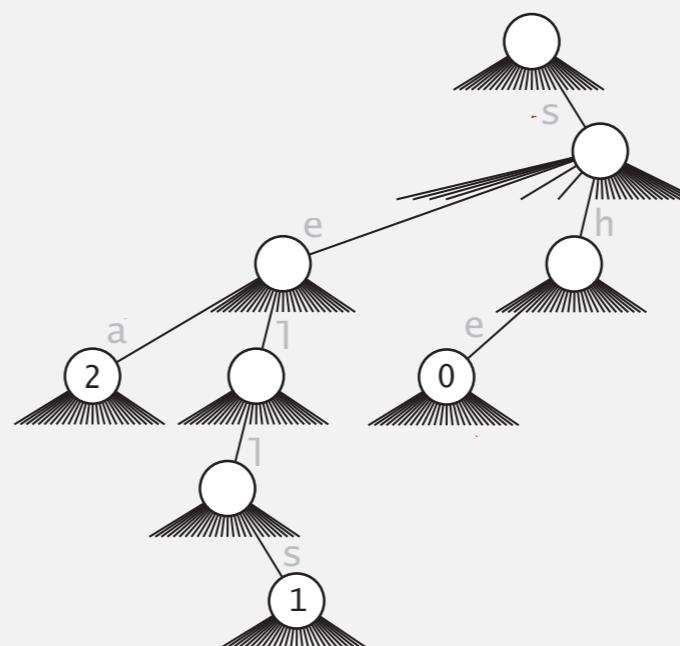
Search hit. Need to examine all L characters for equality.

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)



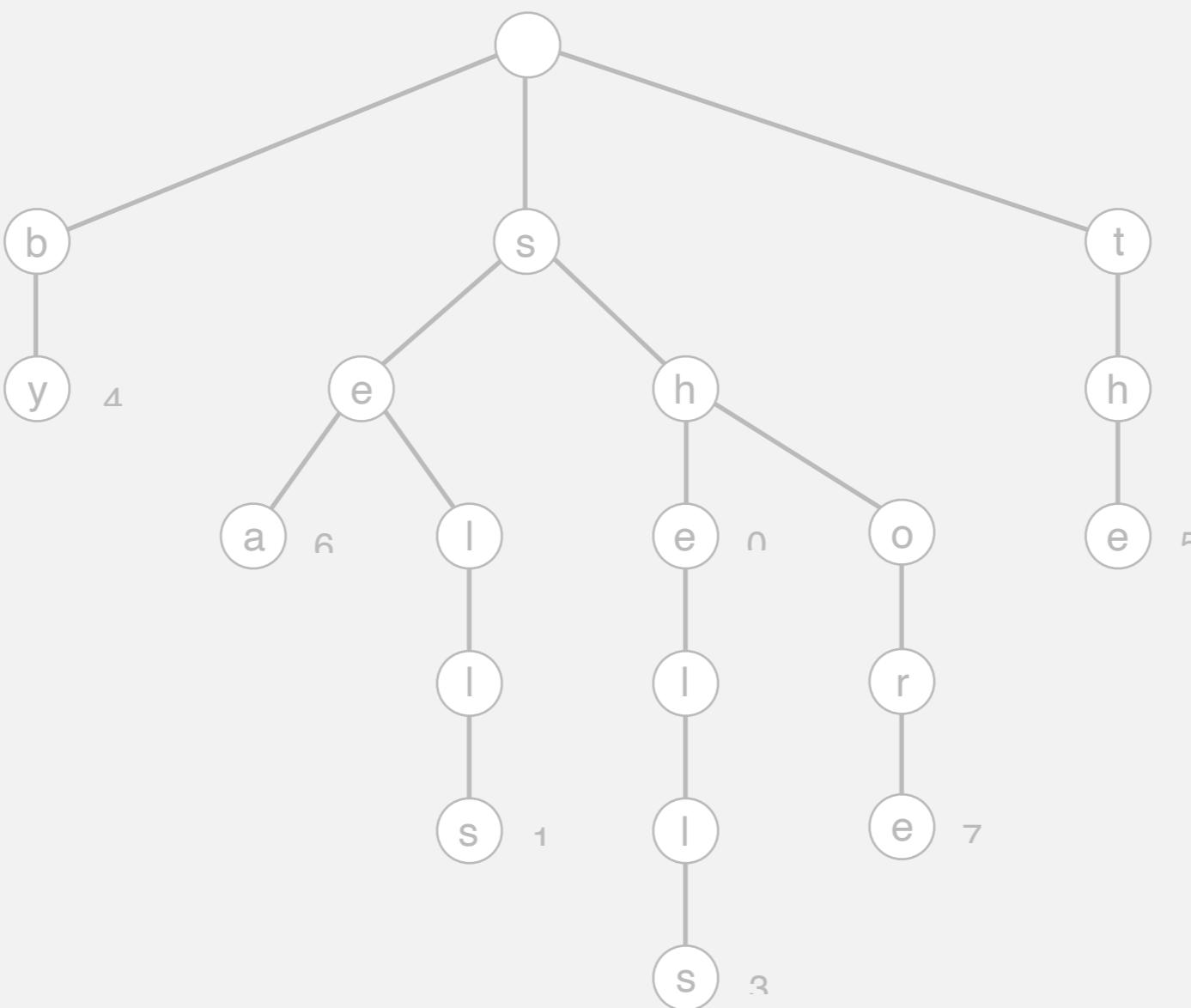
Bottom line. Fast search hit and even faster search miss, but wastes space.

Deletion in an R-way trie

To delete a key-value pair:

- Find the node corresponding to key and set value to null.
- If node has null value and all null links, remove that node (and recur).

delete("shells")

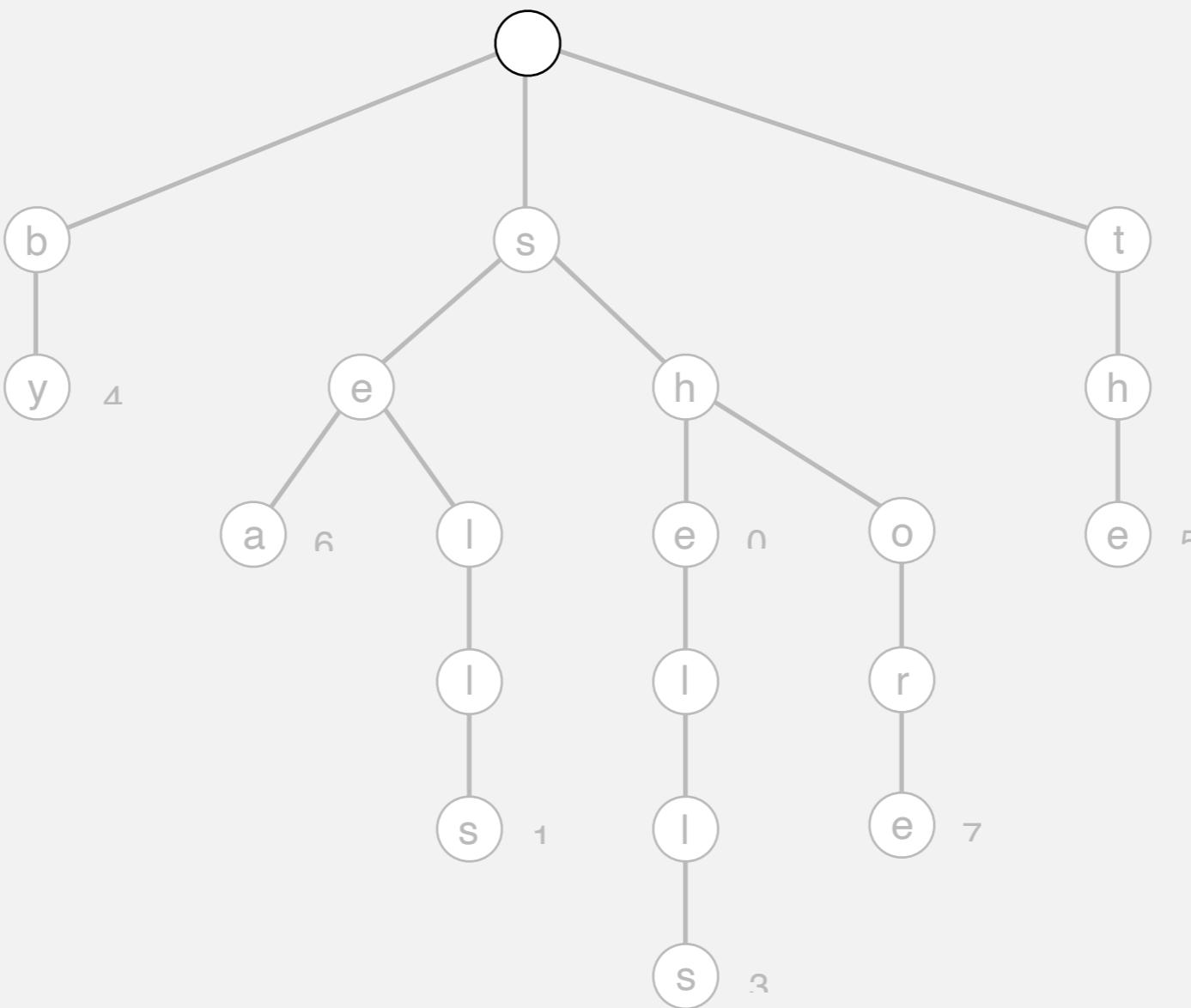


Deletion in an R-way trie

To delete a key-value pair:

- Find the node corresponding to key and set value to null.
- If node has null value and all null links, remove that node (and recur).

delete("shells")

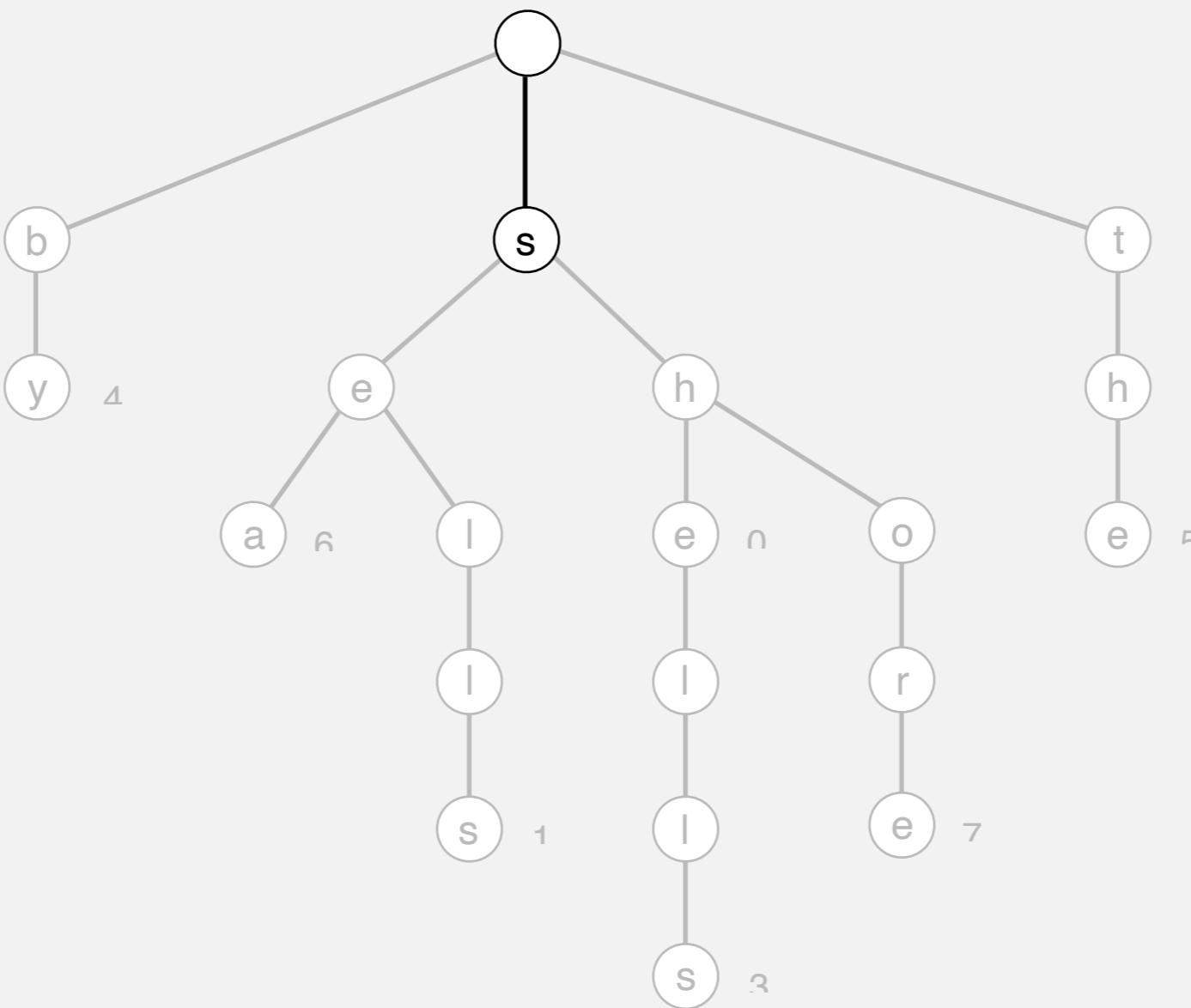


Deletion in an R-way trie

To delete a key-value pair:

- Find the node corresponding to key and set value to null.
- If node has null value and all null links, remove that node (and recur).

delete("shells")

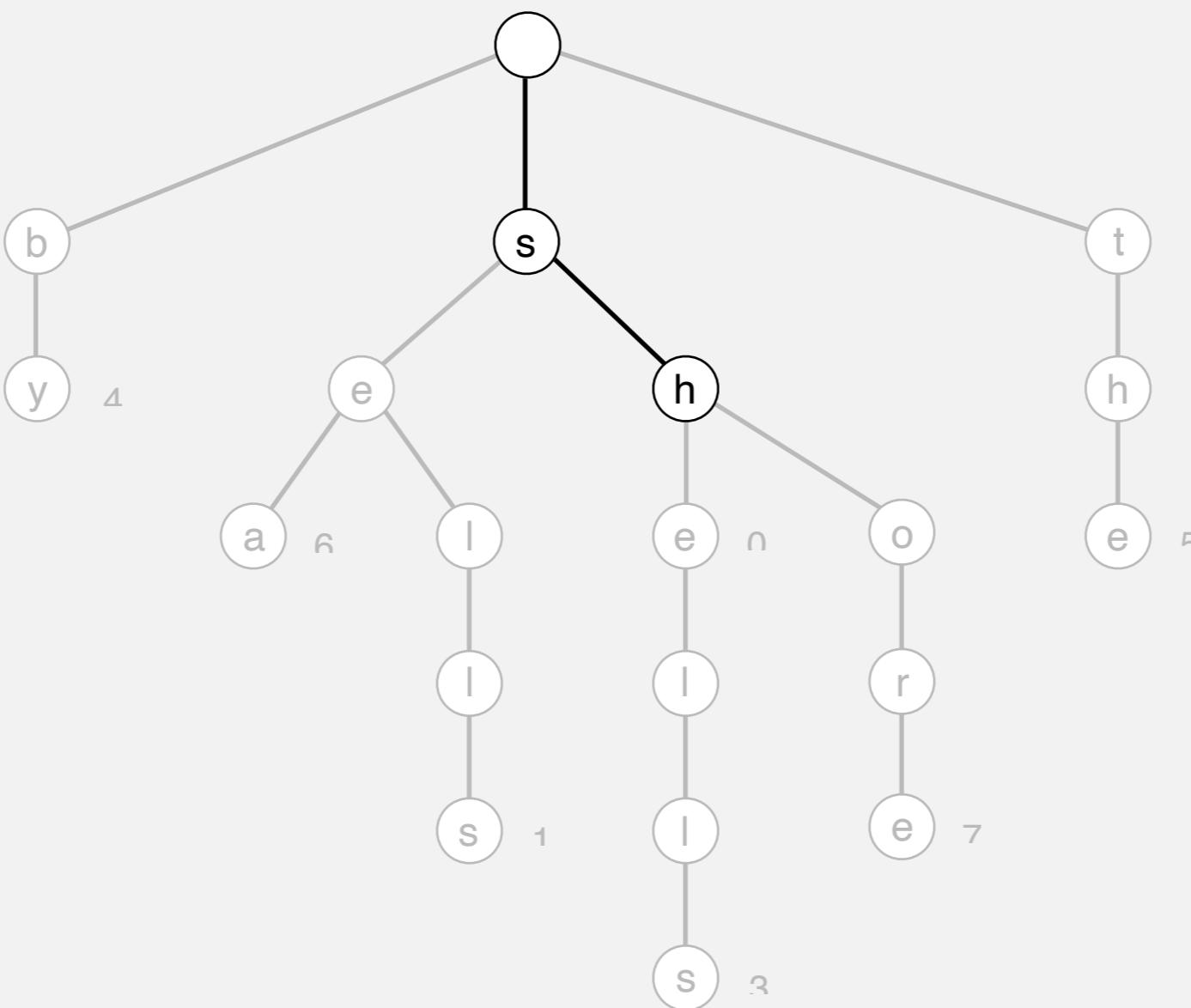


Deletion in an R-way trie

To delete a key-value pair:

- Find the node corresponding to key and set value to null.
- If node has null value and all null links, remove that node (and recur).

delete("shells")

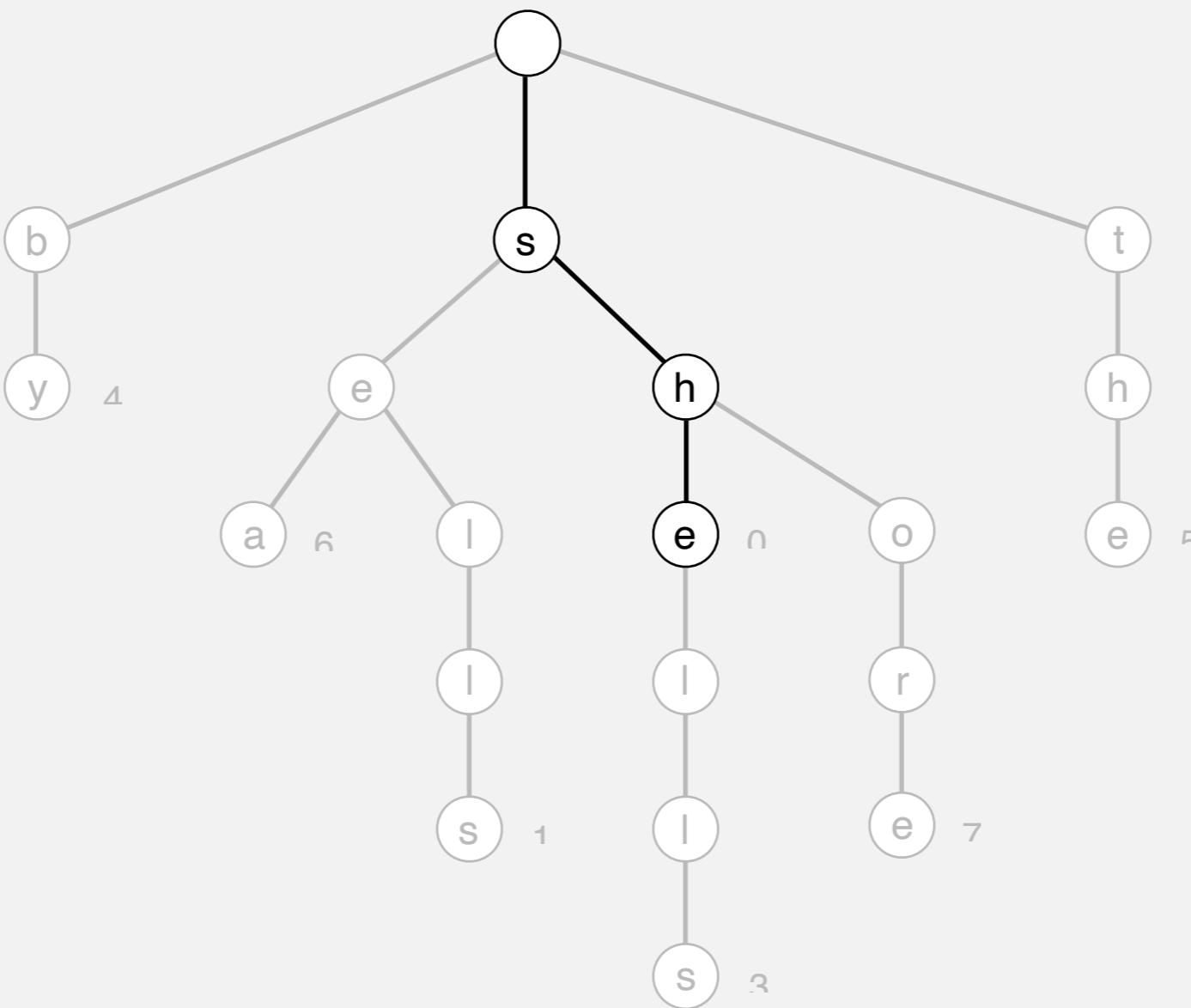


Deletion in an R-way trie

To delete a key-value pair:

- Find the node corresponding to key and set value to null.
- If node has null value and all null links, remove that node (and recur).

delete("shells")

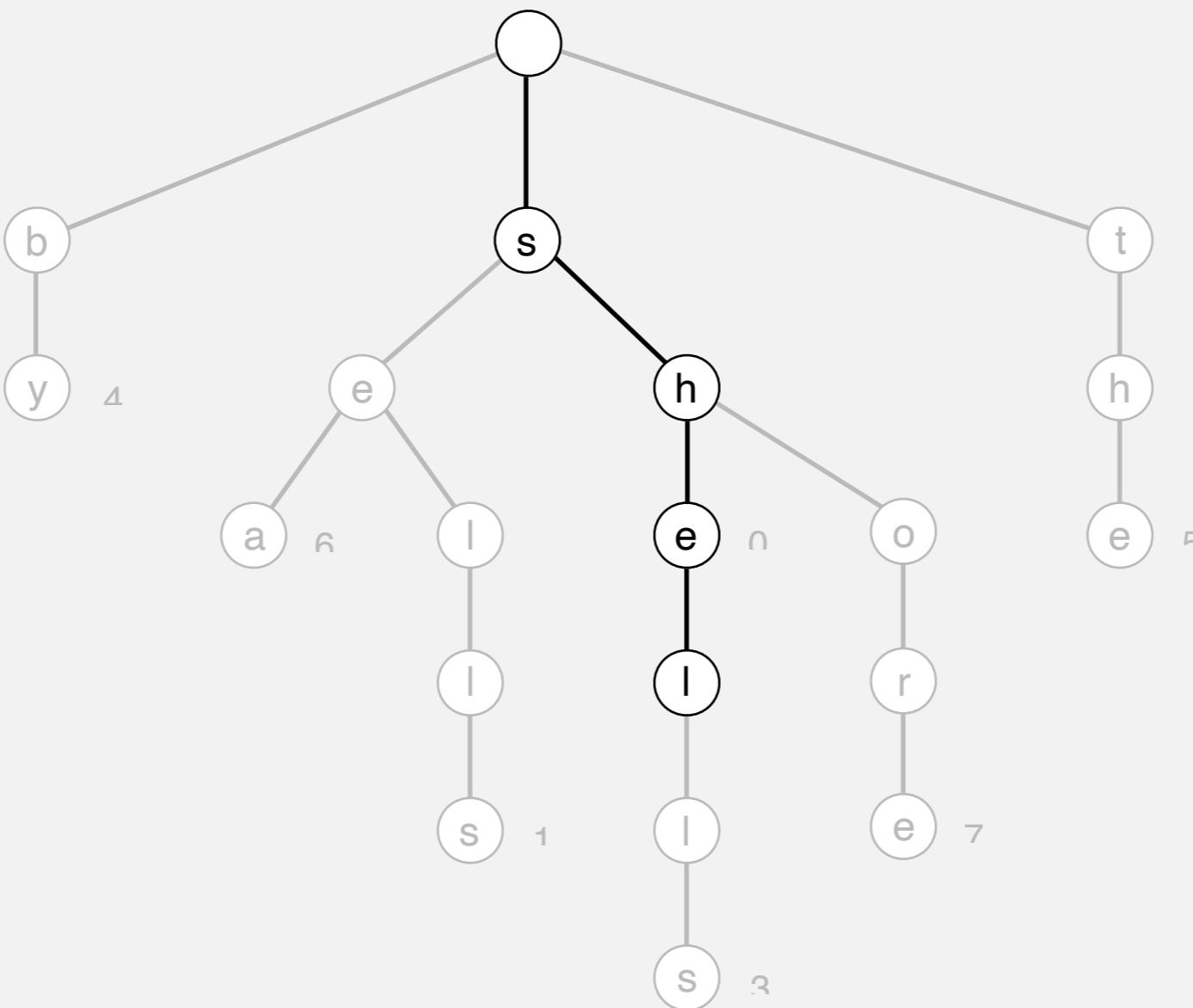


Deletion in an R-way trie

To delete a key-value pair:

- Find the node corresponding to key and set value to null.
- If node has null value and all null links, remove that node (and recur).

delete("shells")

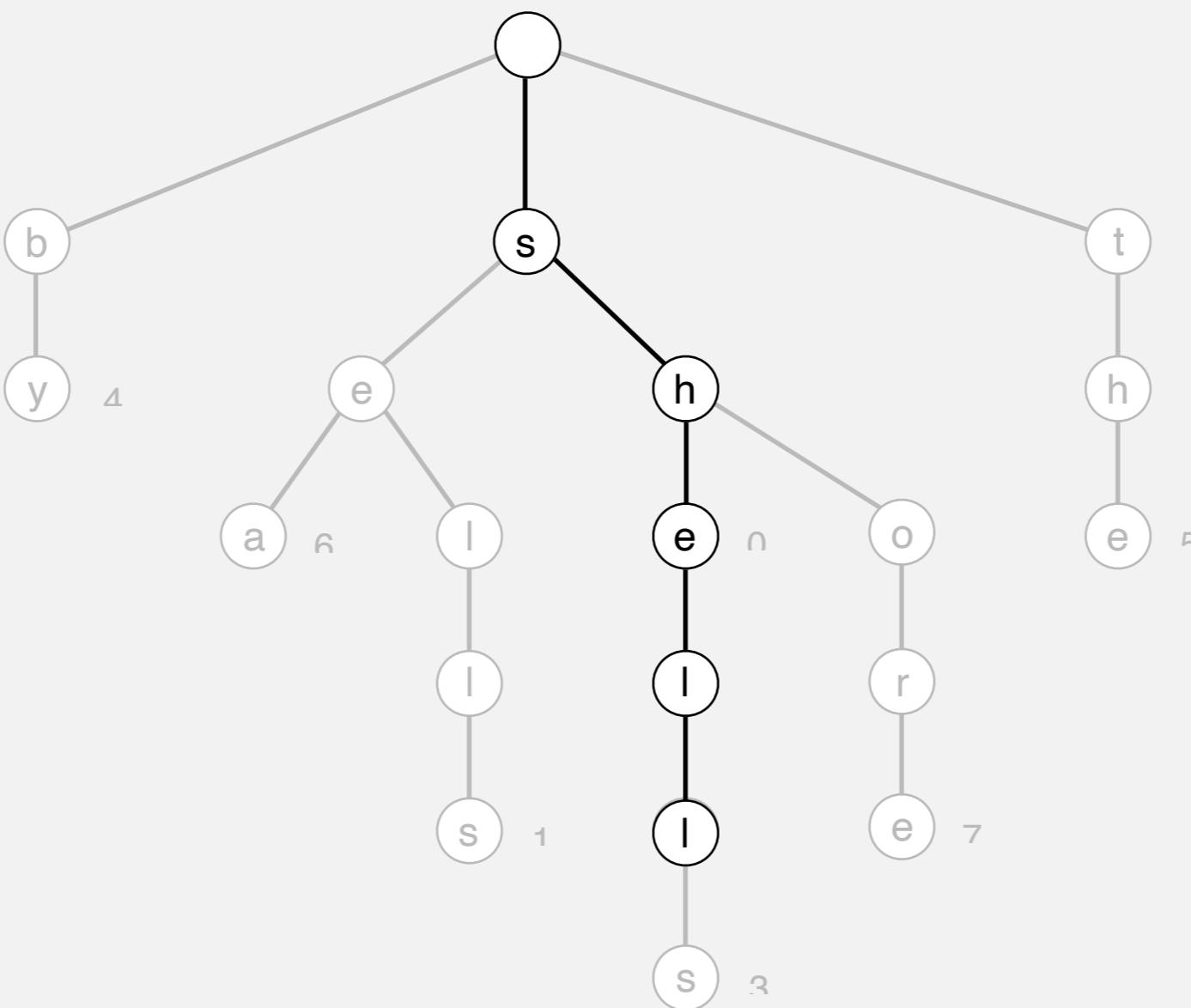


Deletion in an R-way trie

To delete a key-value pair:

- Find the node corresponding to key and set value to null.
- If node has null value and all null links, remove that node (and recur).

delete("shells")

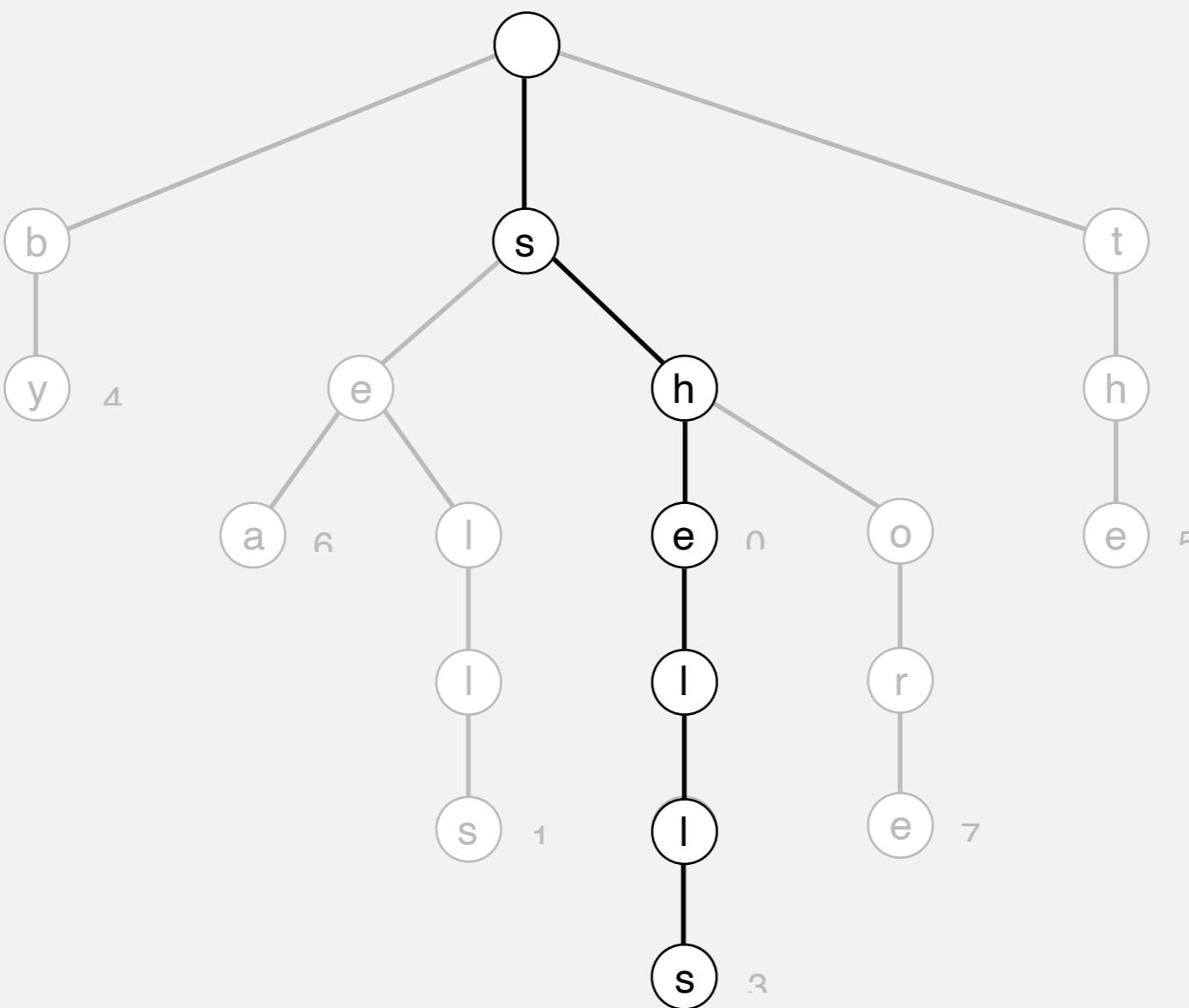


Deletion in an R-way trie

To delete a key-value pair:

- Find the node corresponding to key and set value to null.
- If node has null value and all null links, remove that node (and recur).

delete("shells")

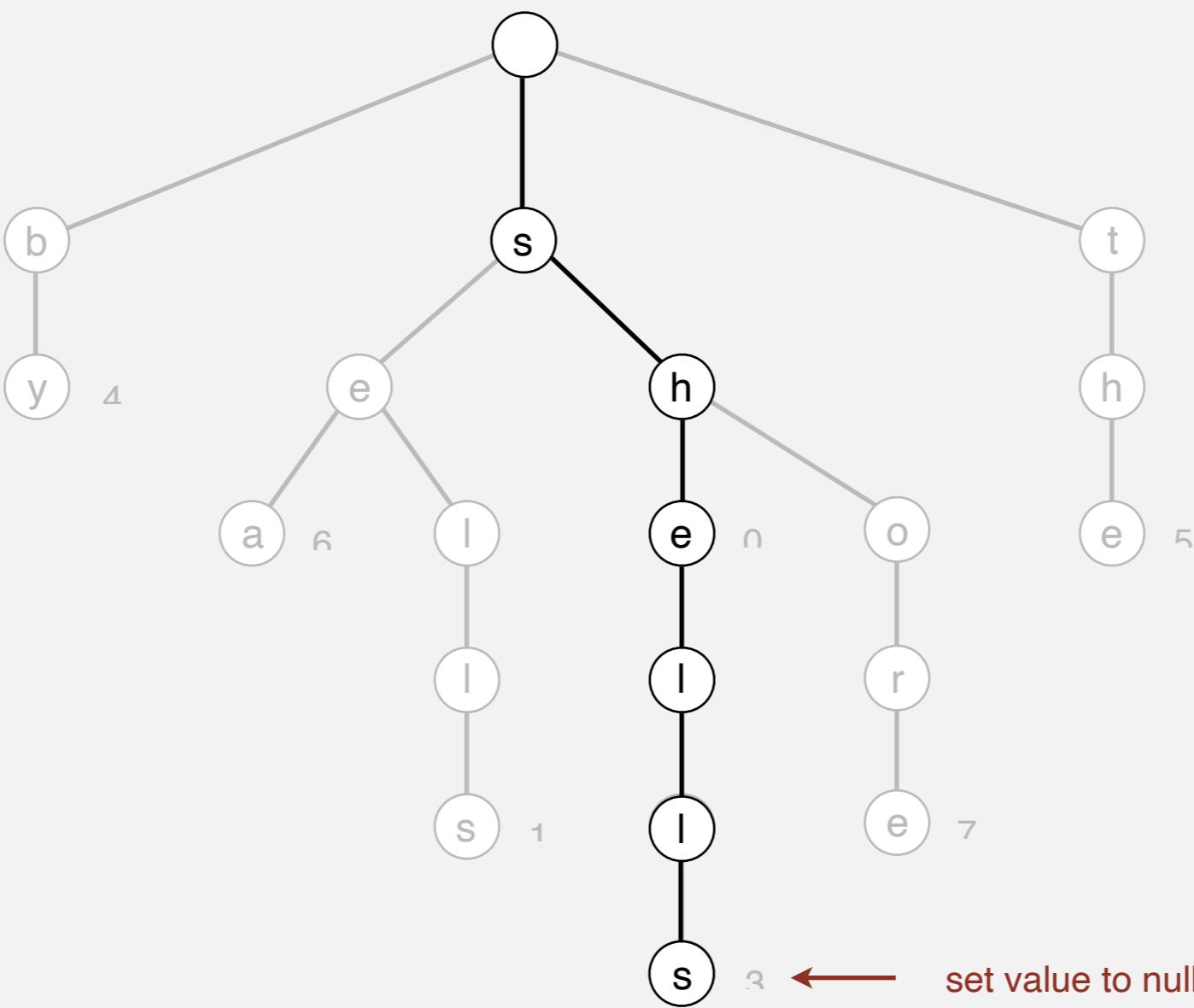


Deletion in an R-way trie

To delete a key-value pair:

- Find the node corresponding to key and set value to null.
- If node has null value and all null links, remove that node (and recur).

delete("shells")

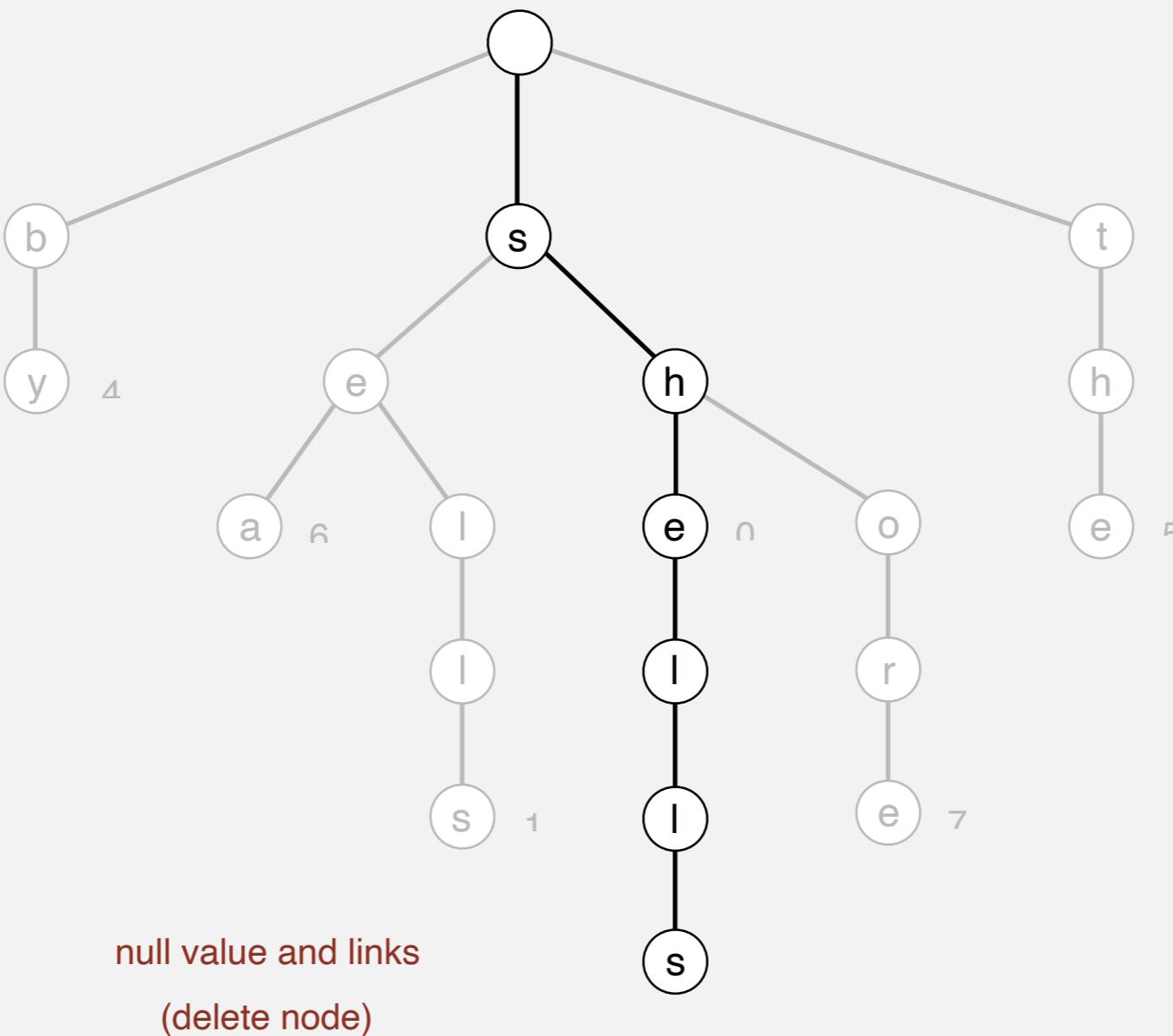


Deletion in an R-way trie

To delete a key-value pair:

- Find the node corresponding to key and set value to null.
 - If node has null value and all null links, remove that node (and recur).

delete("shells")

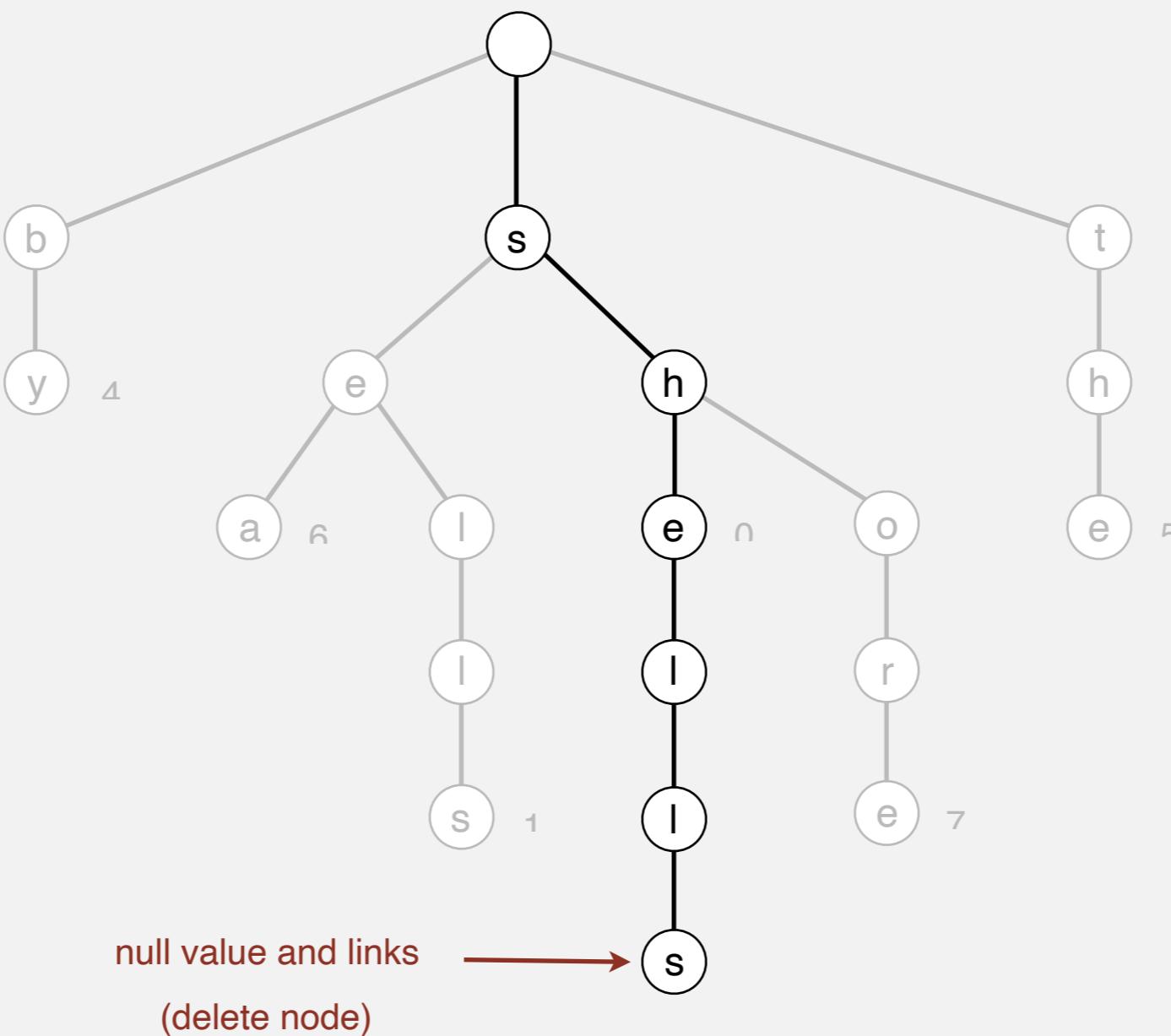


Deletion in an R-way trie

To delete a key-value pair:

- Find the node corresponding to key and set value to null.
- If node has null value and all null links, remove that node (and recur).

delete("shells")

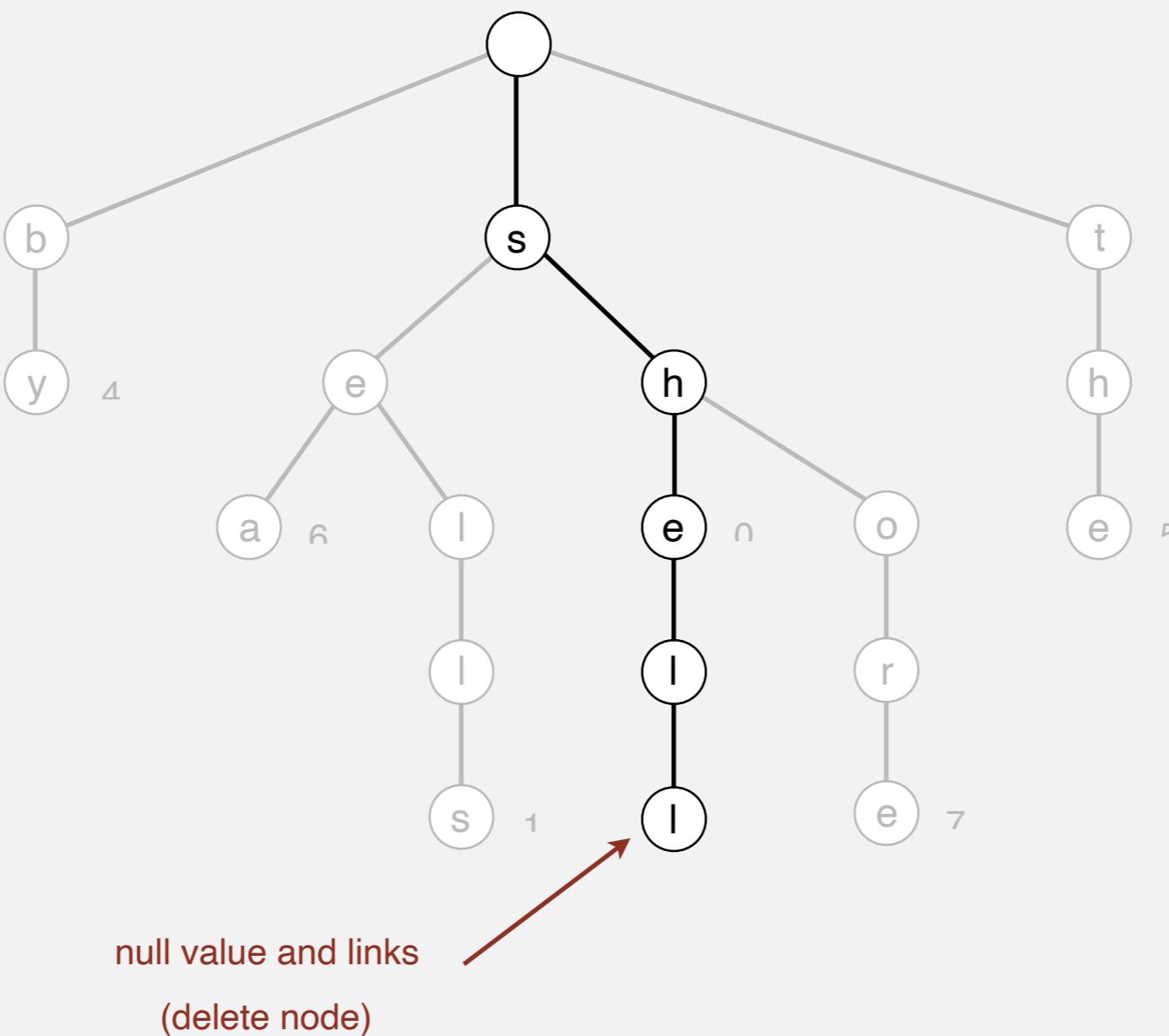


Deletion in an R-way trie

To delete a key-value pair:

- Find the node corresponding to key and set value to null.
- If node has null value and all null links, remove that node (and recur).

delete("shells")

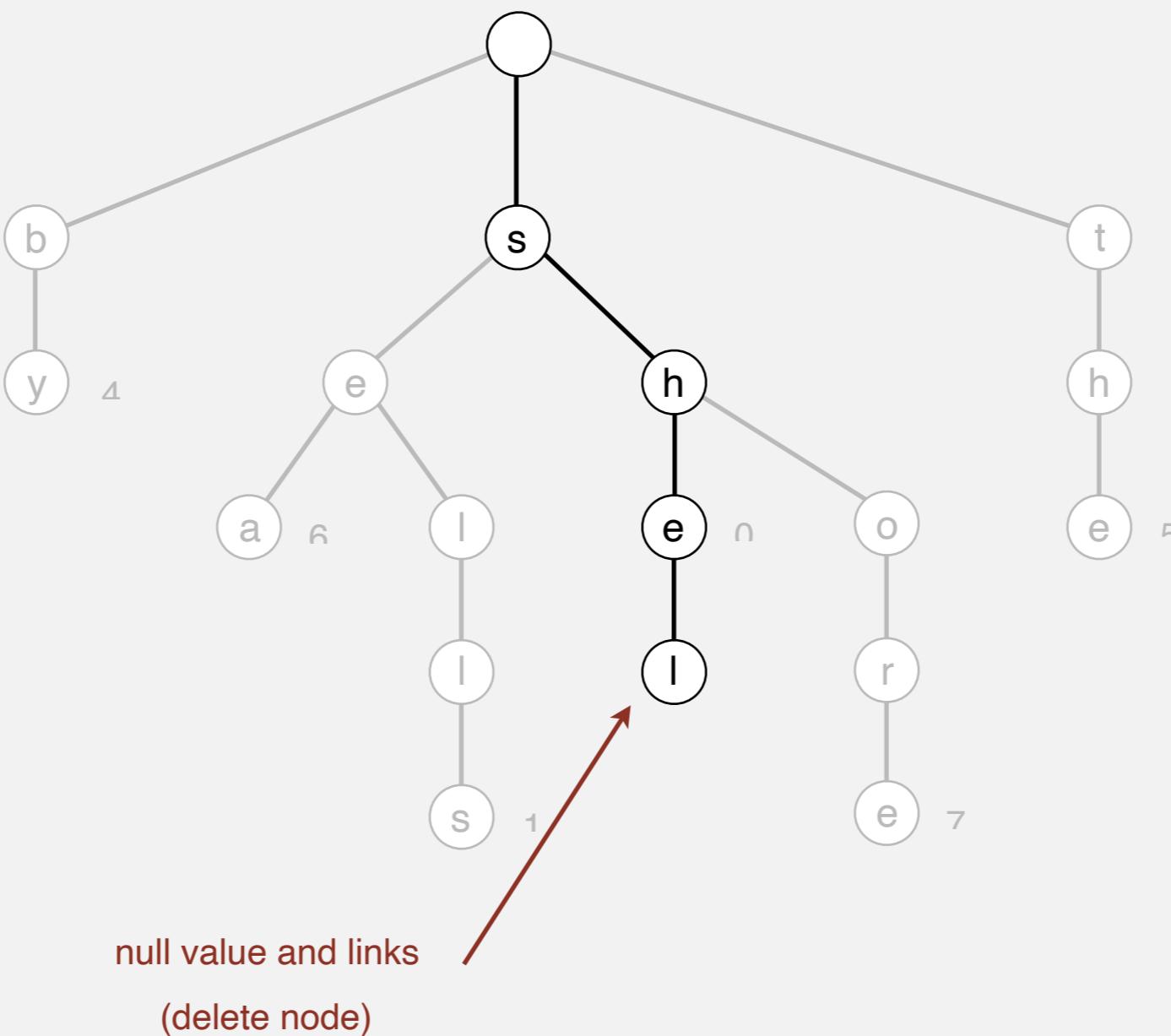


Deletion in an R-way trie

To delete a key-value pair:

- Find the node corresponding to key and set value to null.
- If node has null value and all null links, remove that node (and recur).

delete("shells")

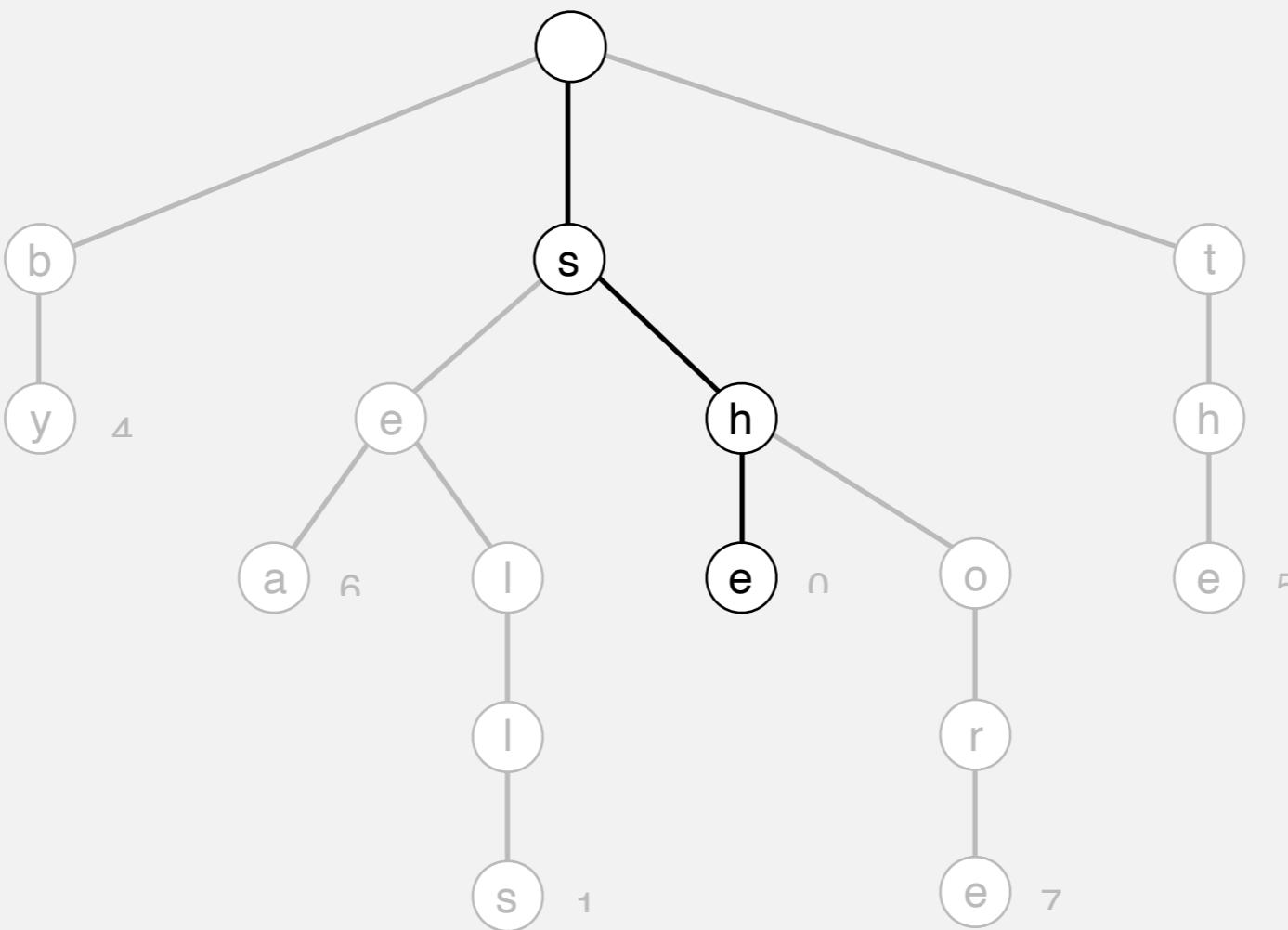


Deletion in an R-way trie

To delete a key-value pair:

- Find the node corresponding to key and set value to null.
- If node has null value and all null links, remove that node (and recur).

delete("shells")



String symbol table implementations cost summary

implementation	character accesses (typical case)				dedup	
	search hit	search miss	insert	space (references)	moby.txt	actors.txt
red-black BST	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	$4N$	1.4	97.4
hashing (linear probing)	L	L	L	$4N \text{ to } 16N$	0.76	40.6
R-way trie	L	$\log_R N$	L	$(R+1) N$	1.12	<i>out of memory</i>

R-way trie.

- Method of choice for small R .
- Too much memory for large R .

String symbol table implementations cost summary

implementation	character accesses (typical case)				dedup	
	search hit	search miss	insert	space (references)	moby.txt	actors.txt
red-black BST	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	$4N$	1.4	97.4
hashing (linear probing)	L	L	L	$4N \text{ to } 16N$	0.76	40.6
R-way trie	L	$\log_R N$	L	$(R+1) N$	1.12	<i>out of memory</i>

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!

Ternary search tries

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

Fast Algorithms for Sorting and Searching Strings

Jon L. Bentley* Robert Sedgewick#

Abstract

We present theoretical algorithms for sorting and searching multikey data, and derive from them practical C implementations for applications in which keys are character strings. The sorting algorithm blends Quicksort and radix sort; it is competitive with the best known C sort codes. The searching algorithm blends tries and binary search trees; it is faster than hashing and other commonly used search methods. The basic ideas behind the algo-

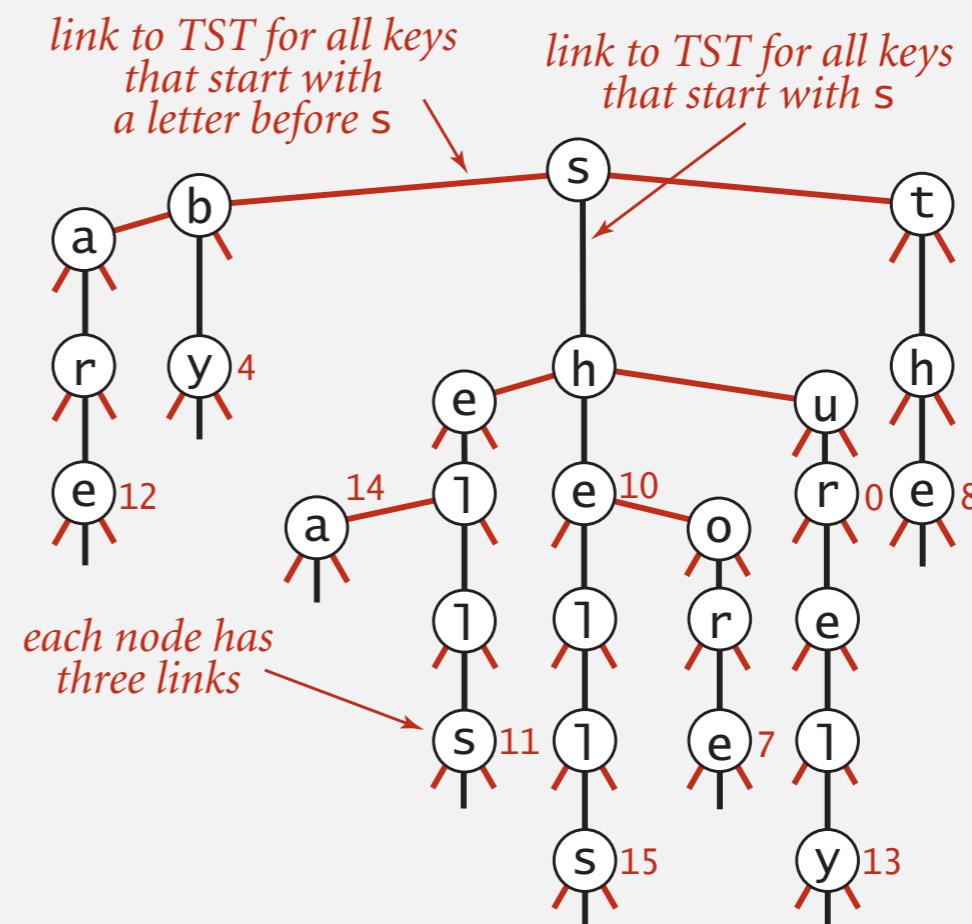
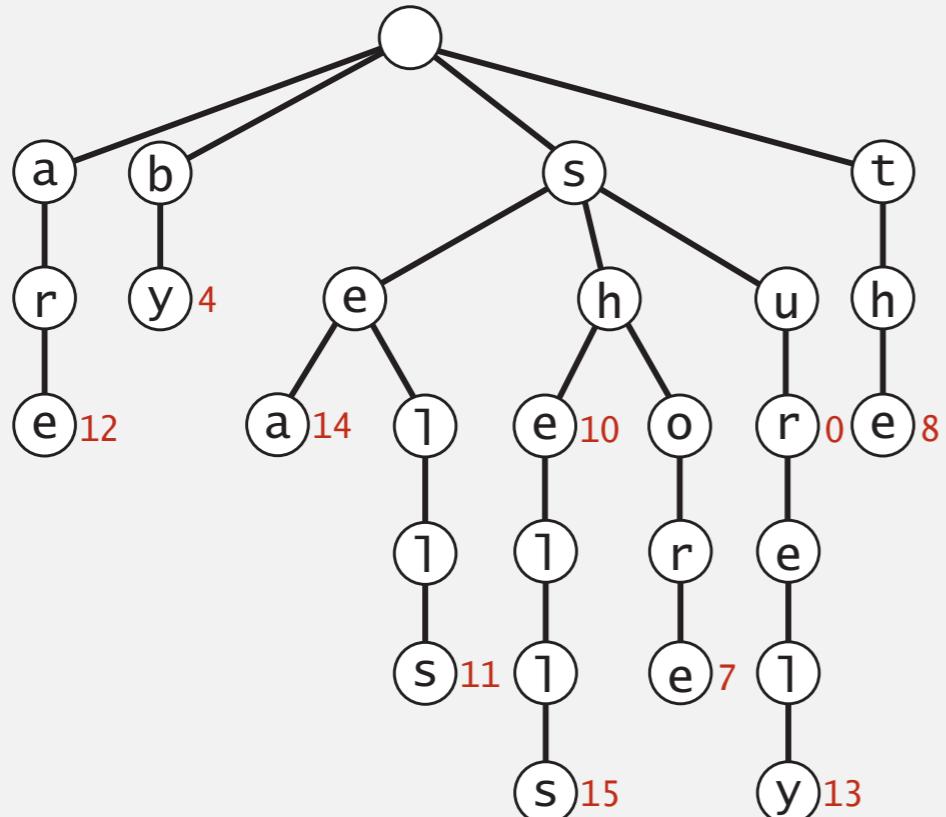
that is competitive with the most efficient string sorting programs known. The second program is a symbol table implementation that is faster than hashing, which is commonly regarded as the fastest symbol table implementation. The symbol table implementation is much more space-efficient than multiway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

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



Ternary search tries

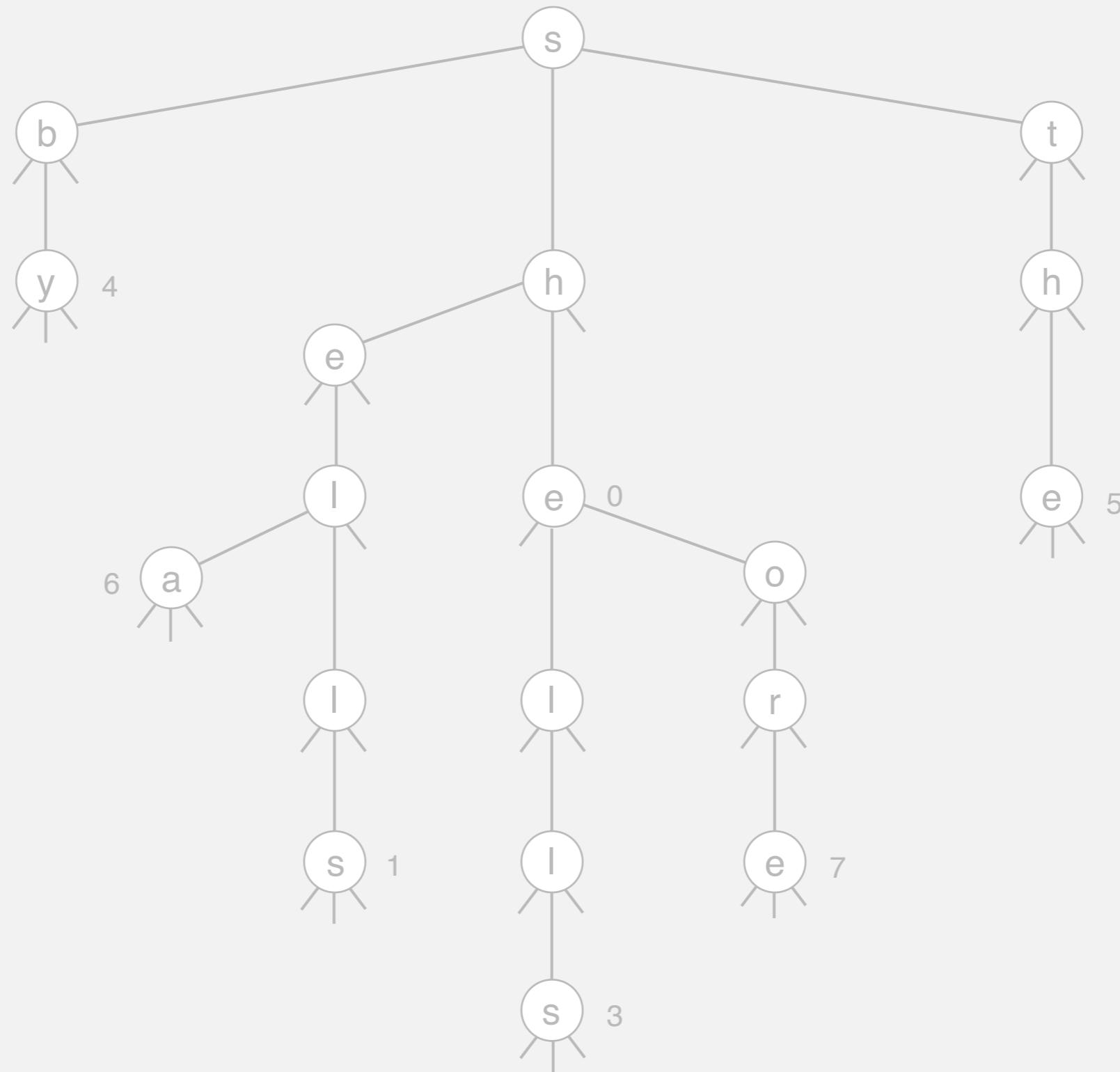
- Store characters and values in nodes (not keys).
- Each node has **3** children: smaller (left), equal (middle), larger (right).



TST representation of a trie

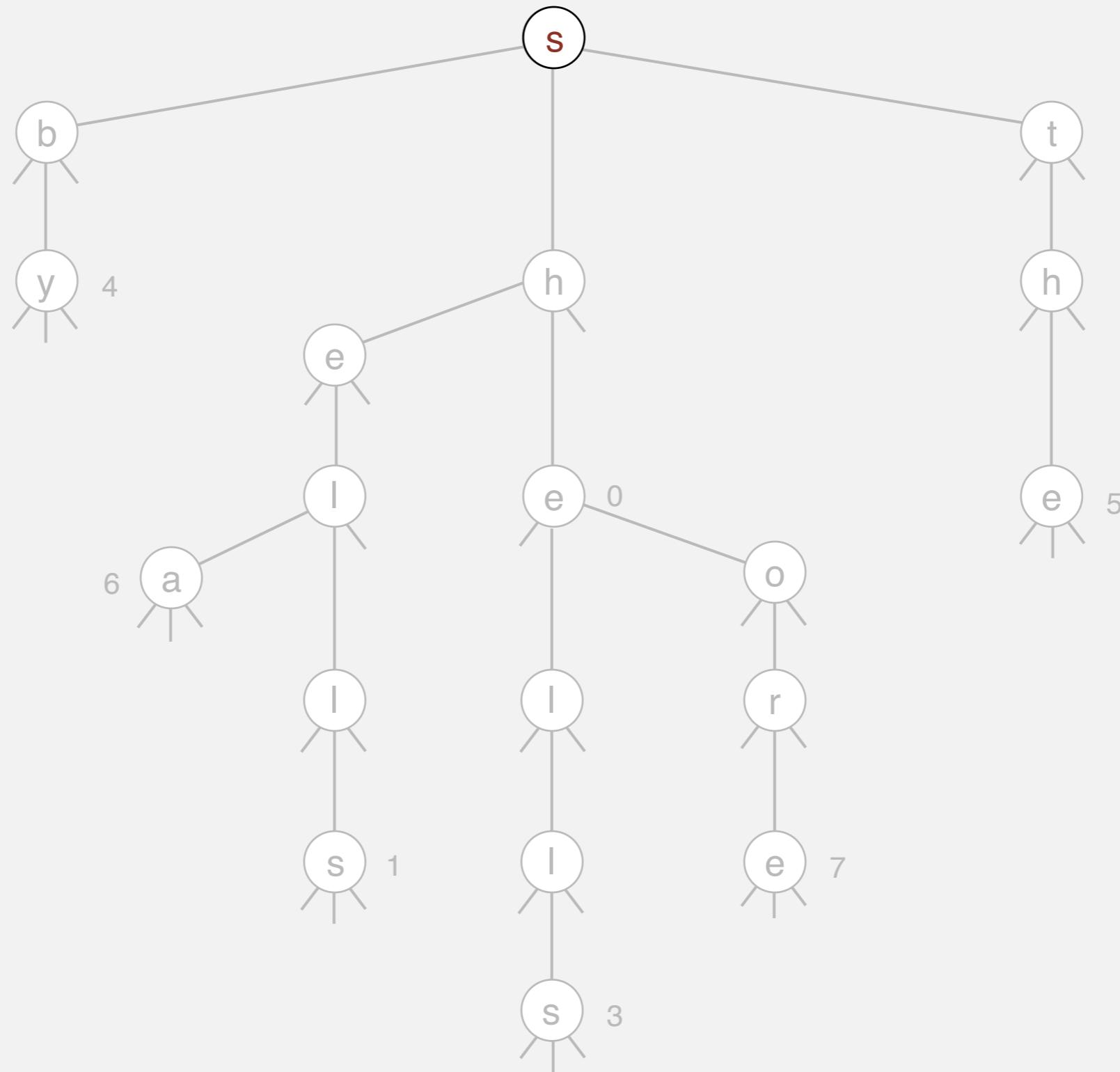
Search hit in a TST

get("sea")



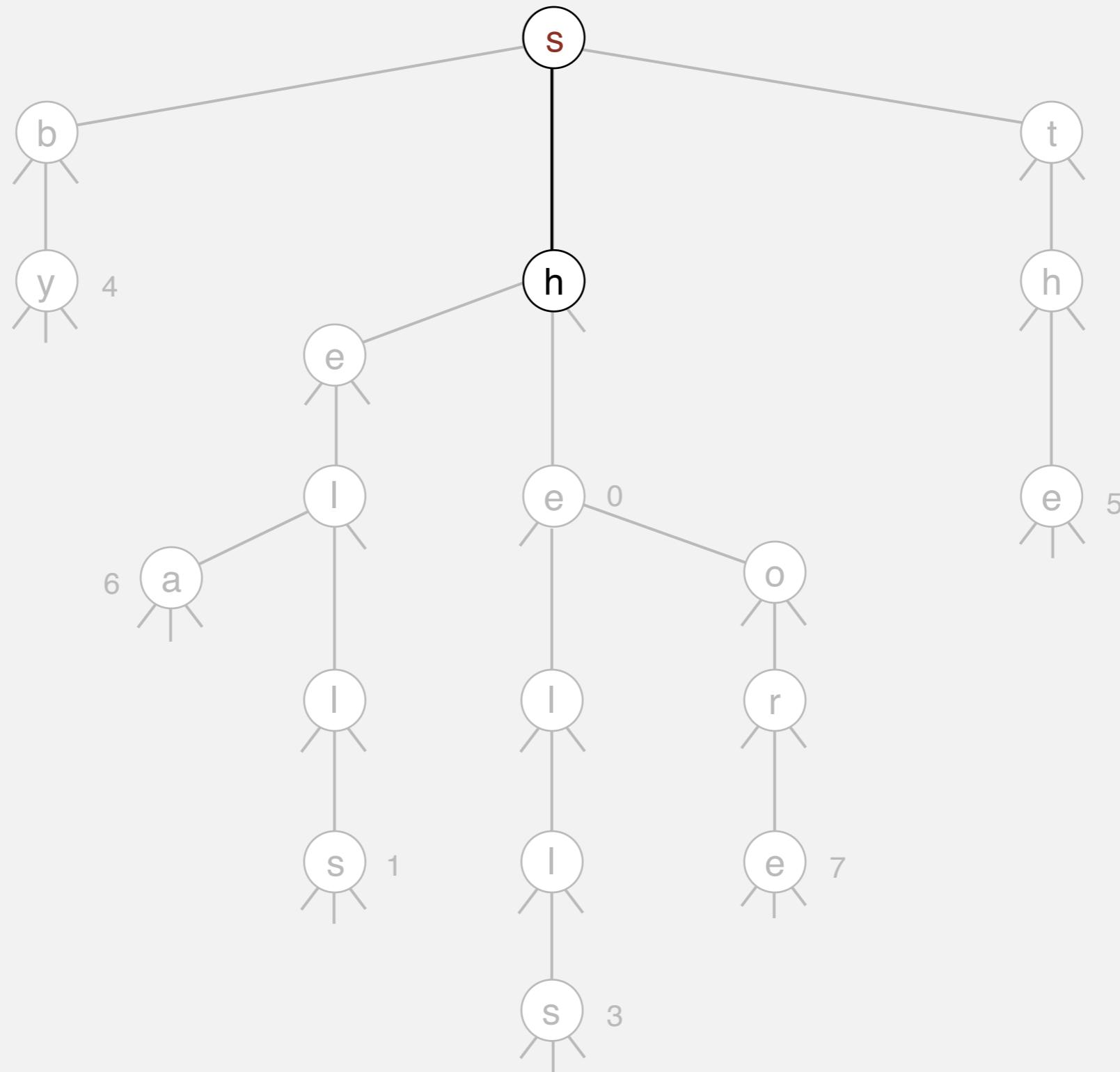
Search hit in a TST

get("sea")



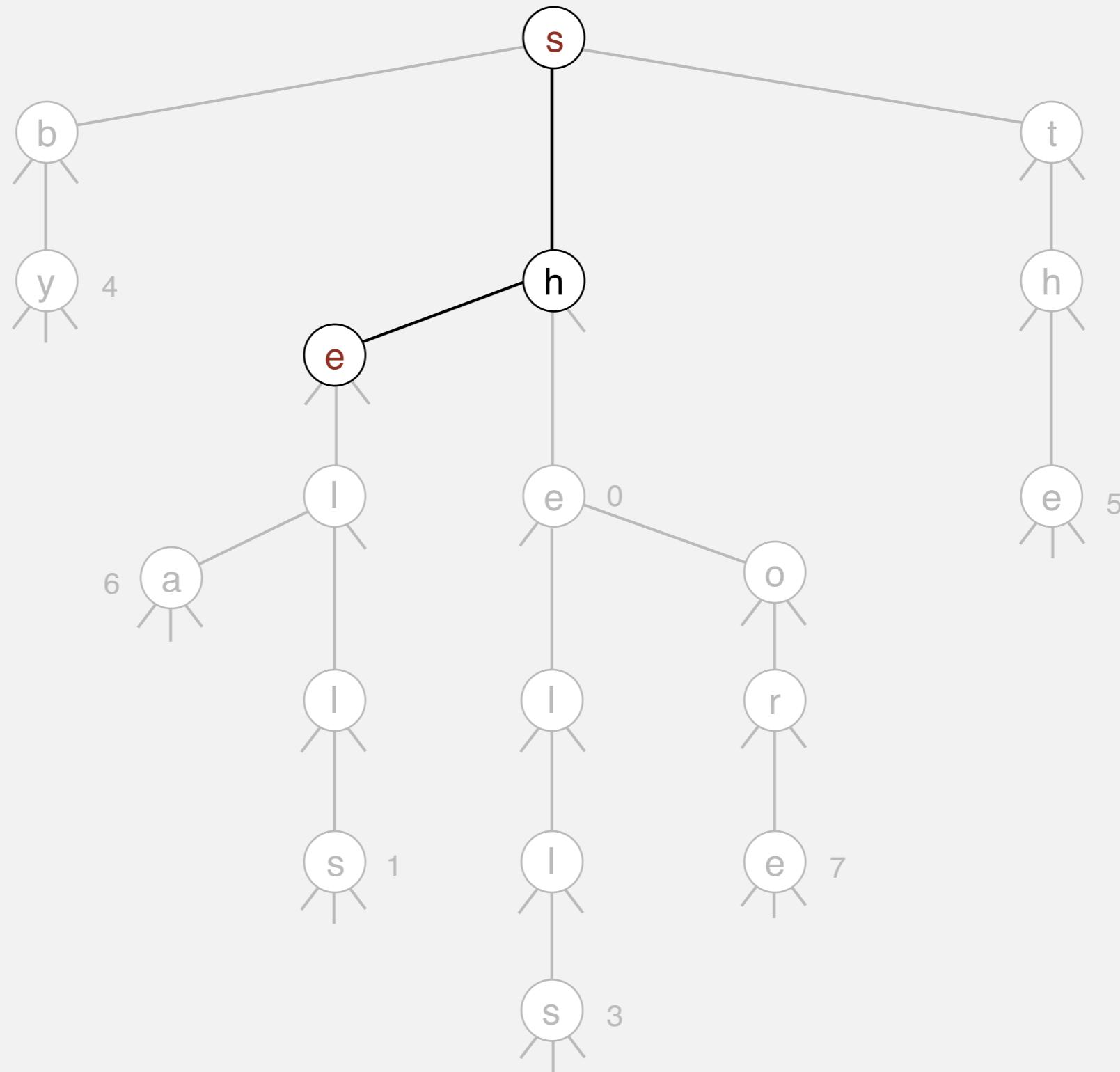
Search hit in a TST

get("sea")



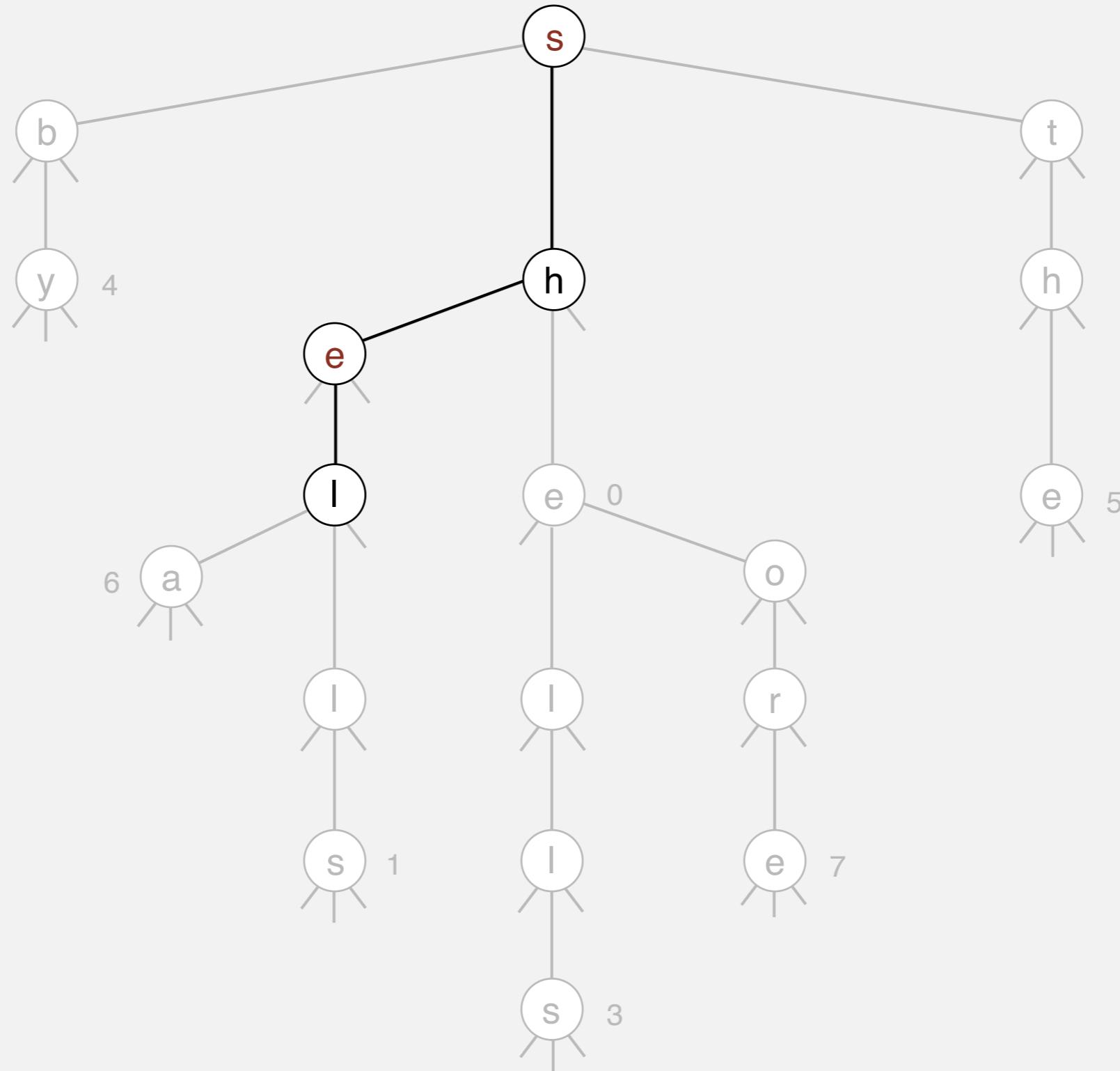
Search hit in a TST

get("sea")



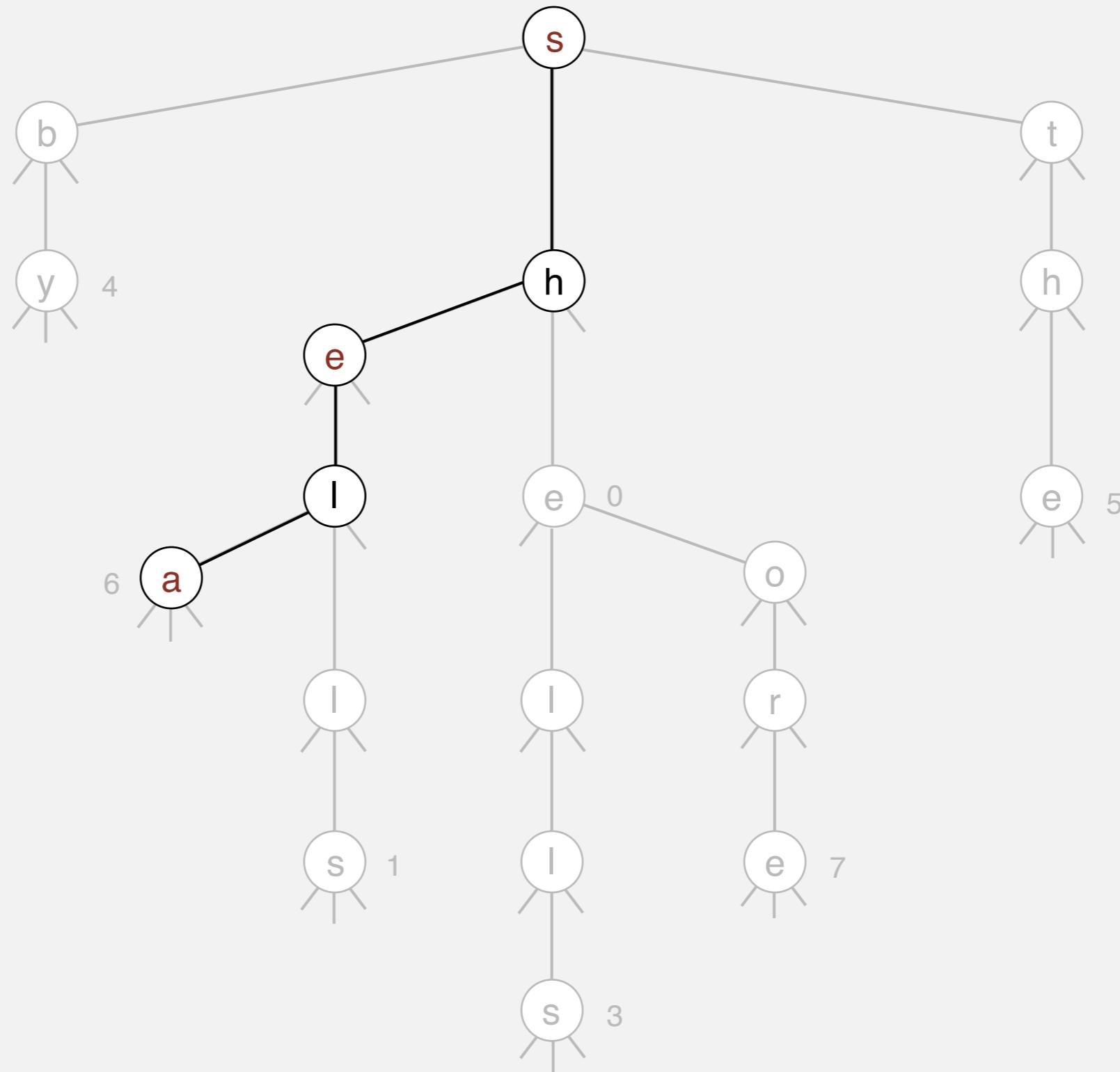
Search hit in a TST

get("sea")



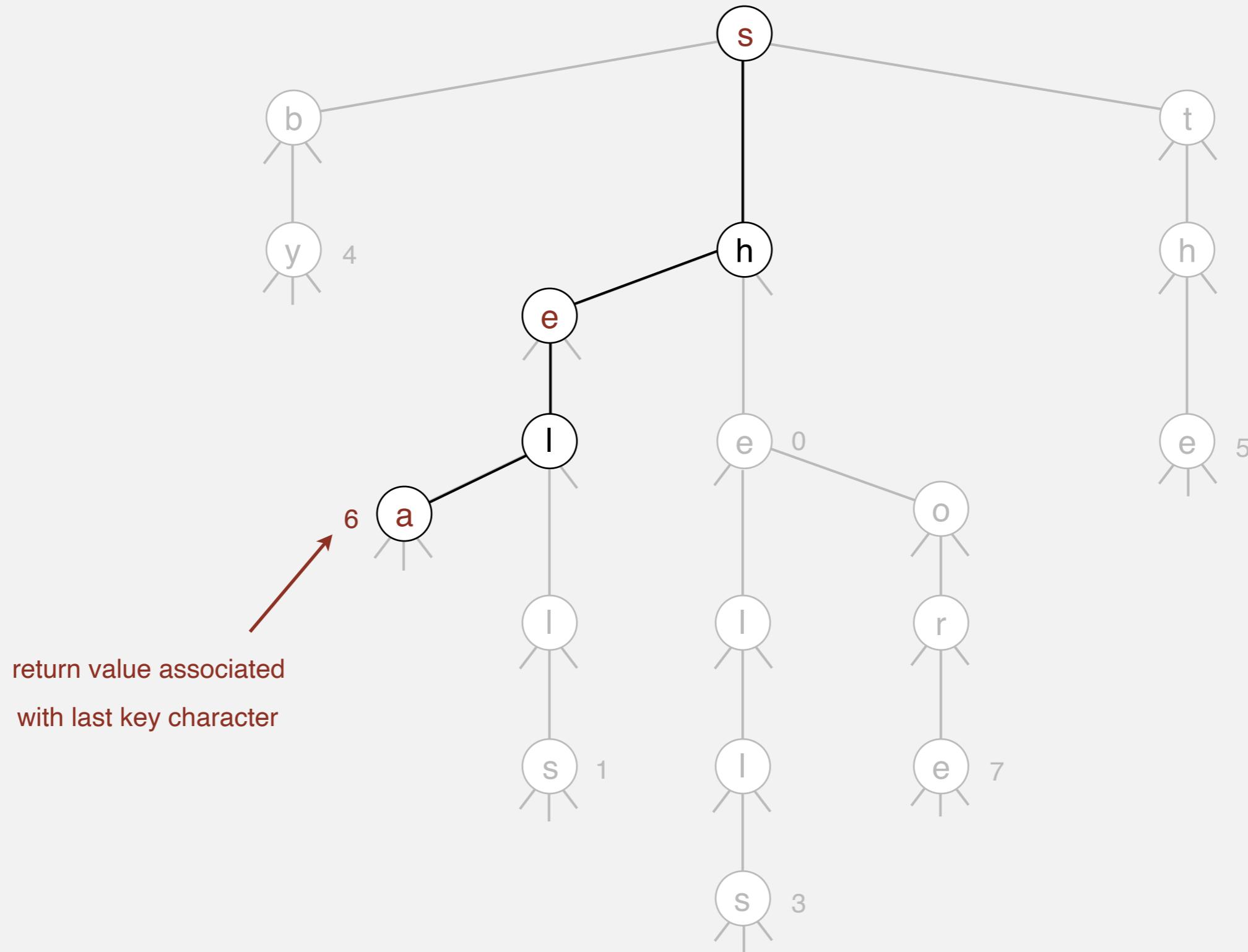
Search hit in a TST

get("sea")



Search hit in a TST

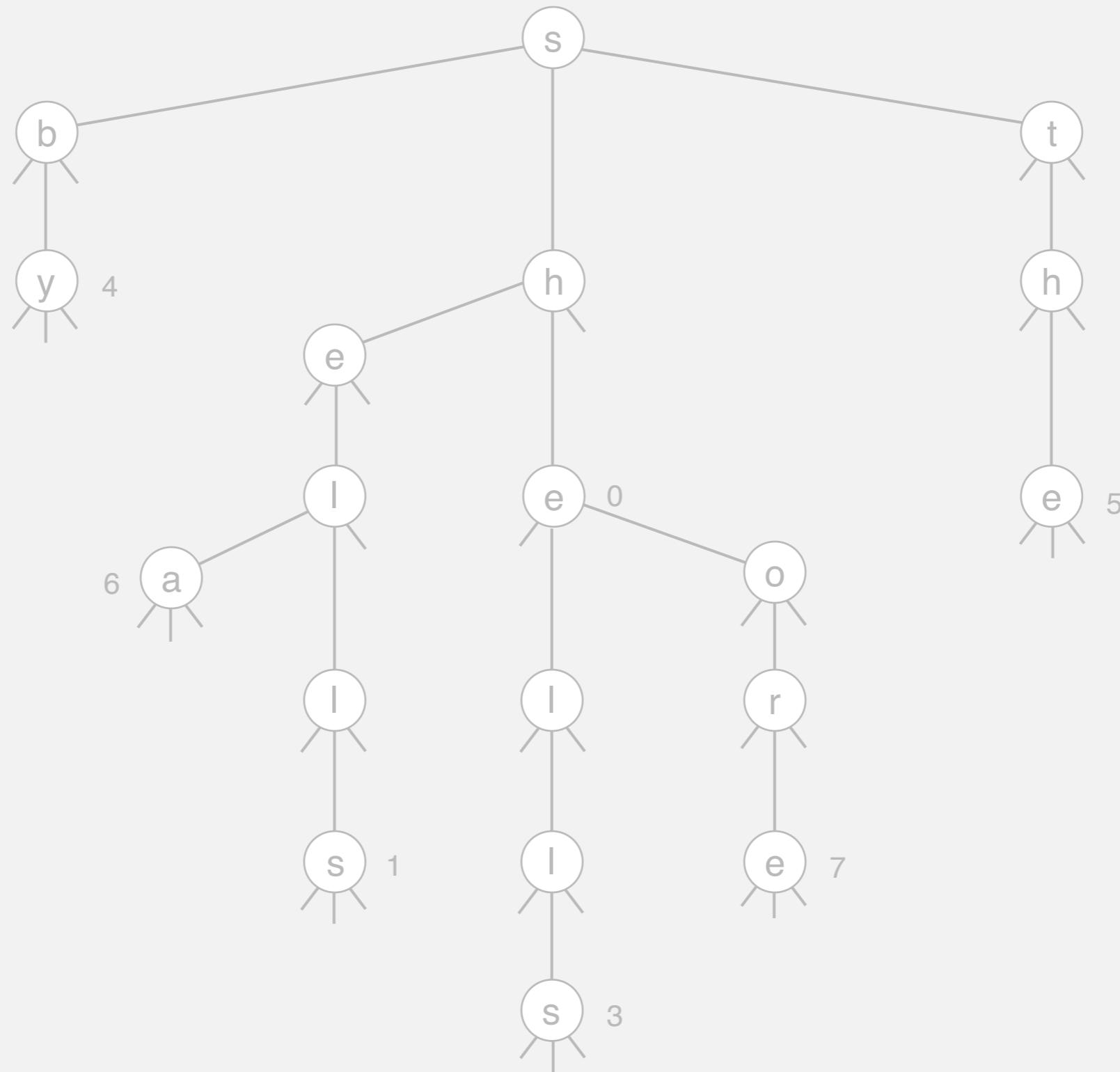
get("sea")



return value associated
with last key character

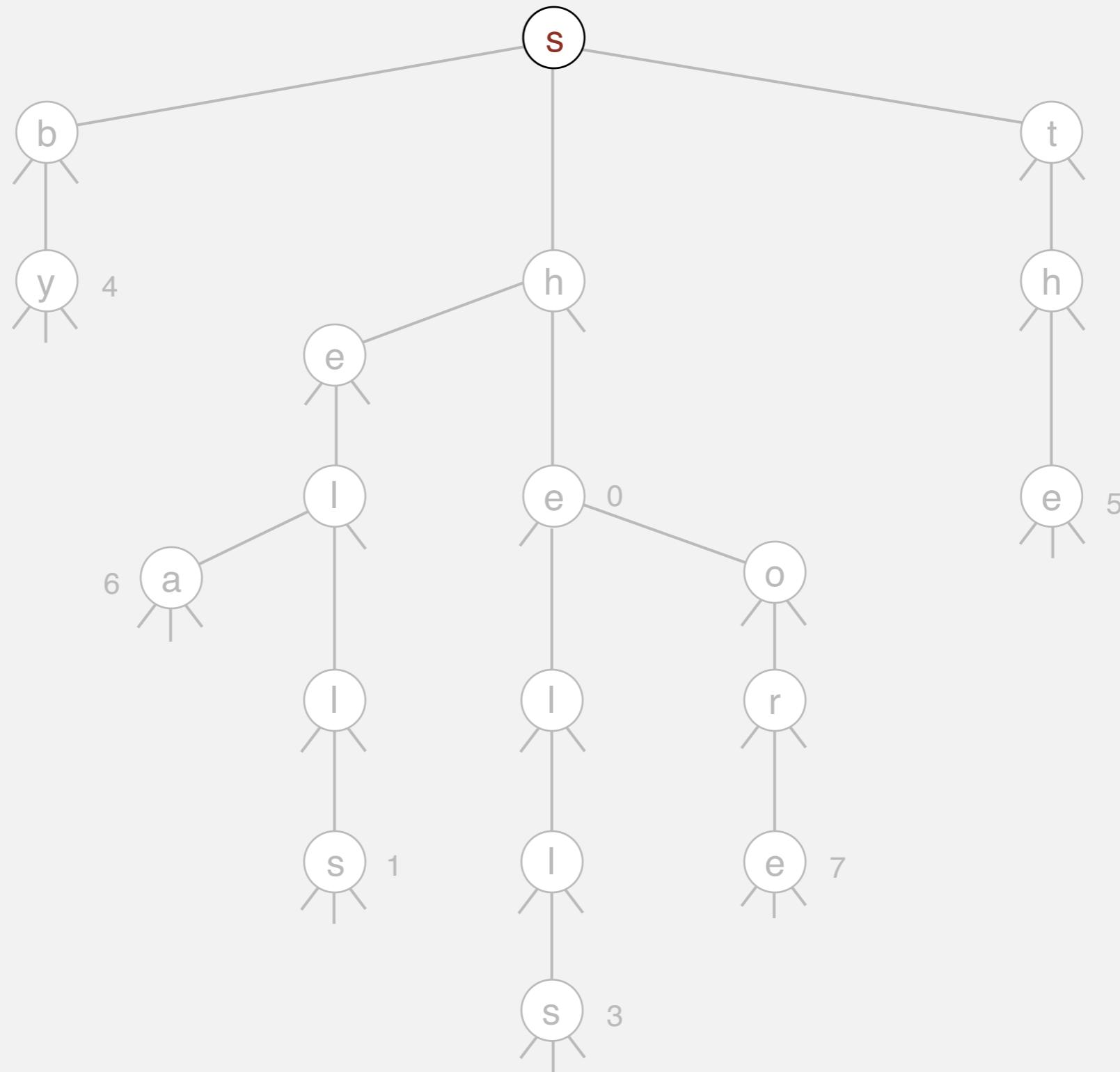
Search miss in a TST

get("shelter")



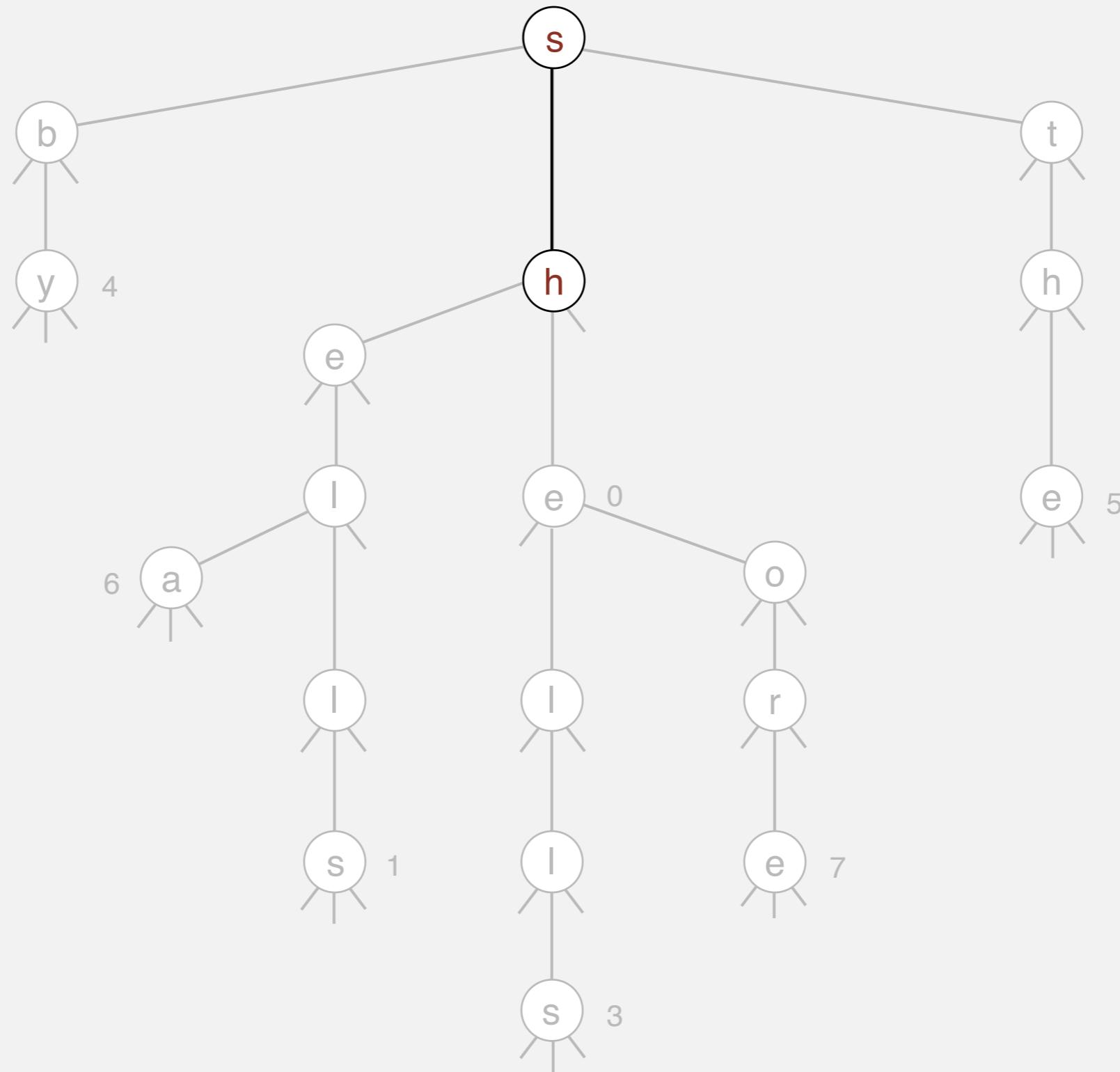
Search miss in a TST

get("shelter")



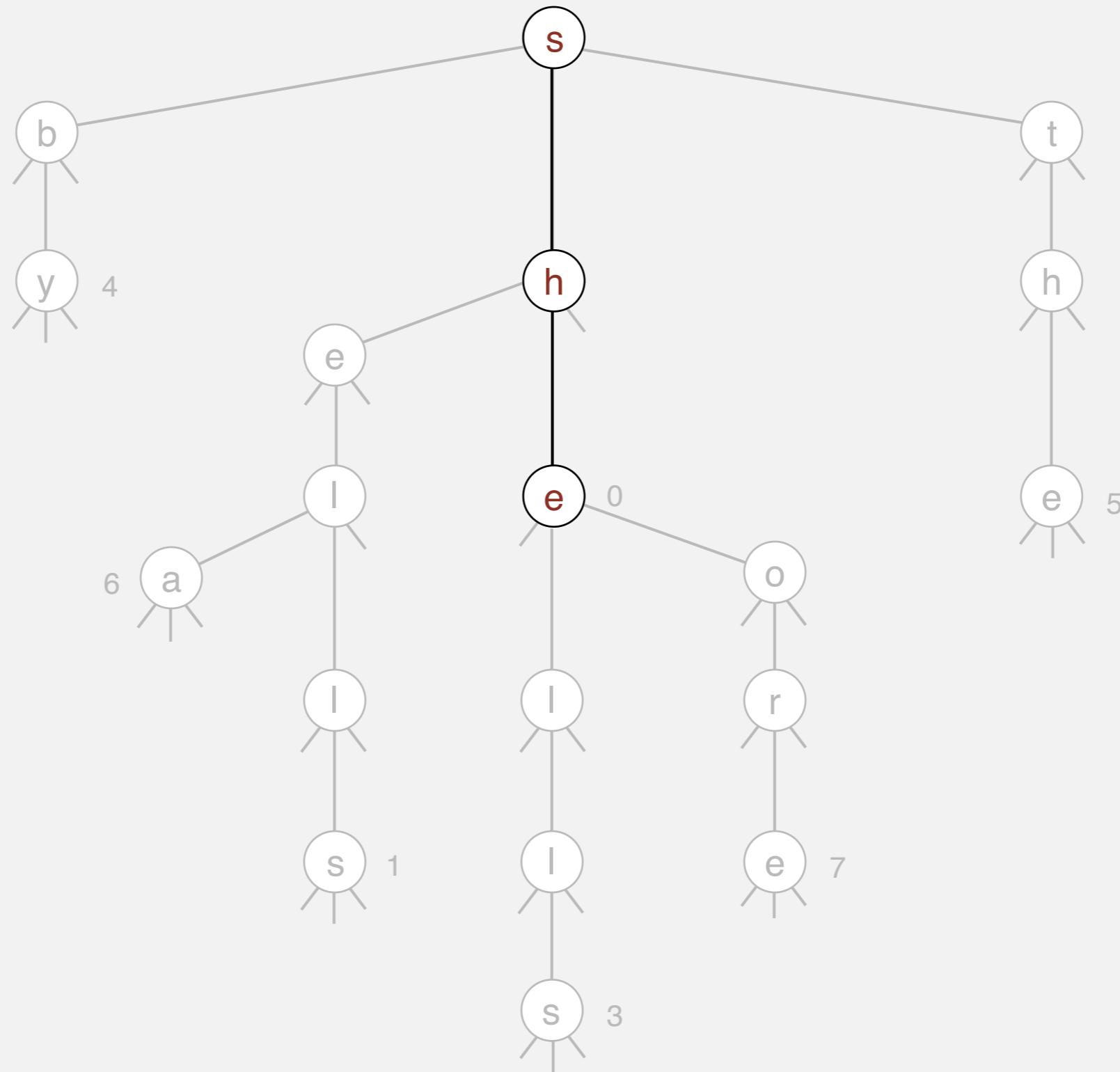
Search miss in a TST

get("shelter")



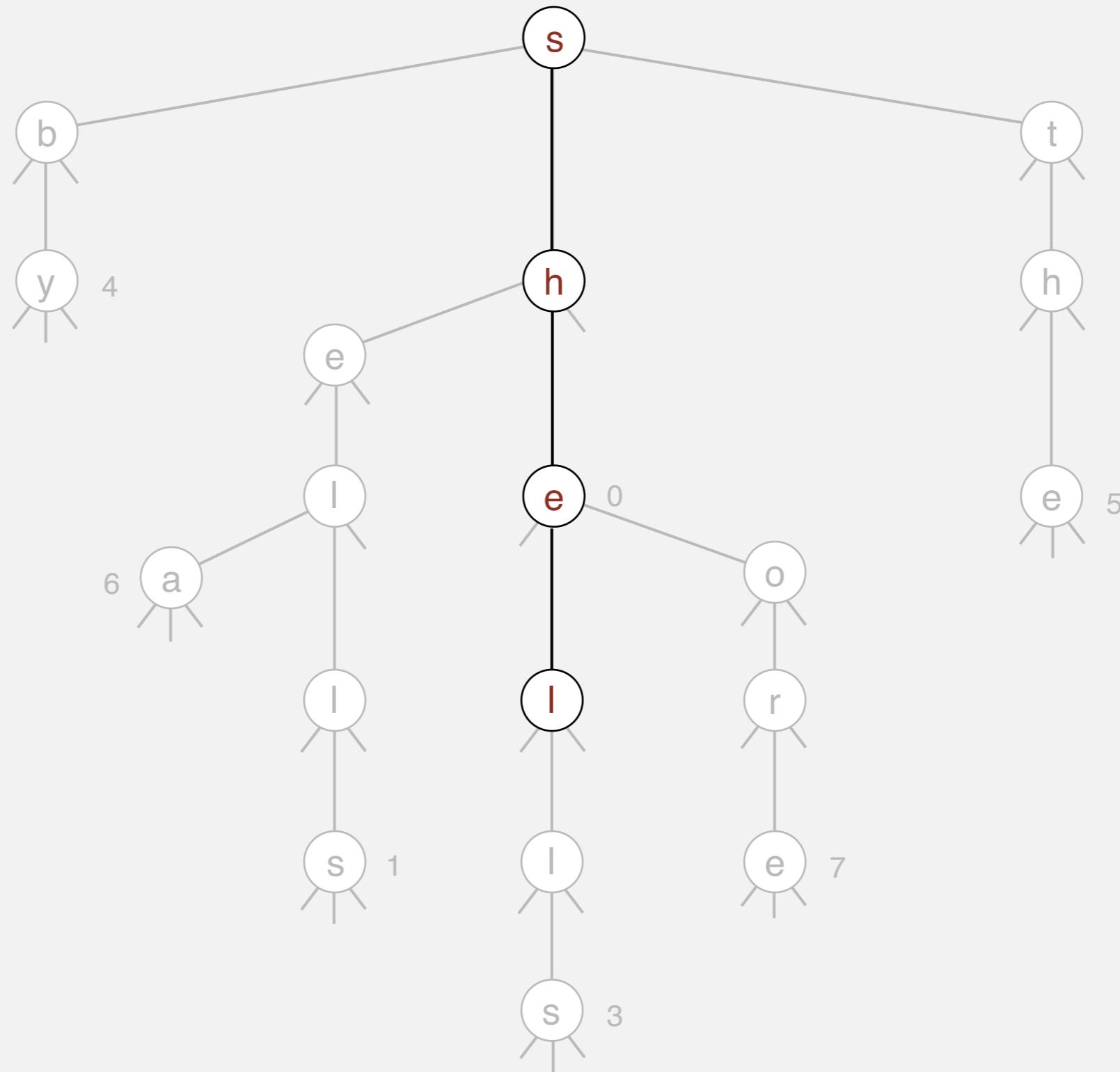
Search miss in a TST

get("shelter")



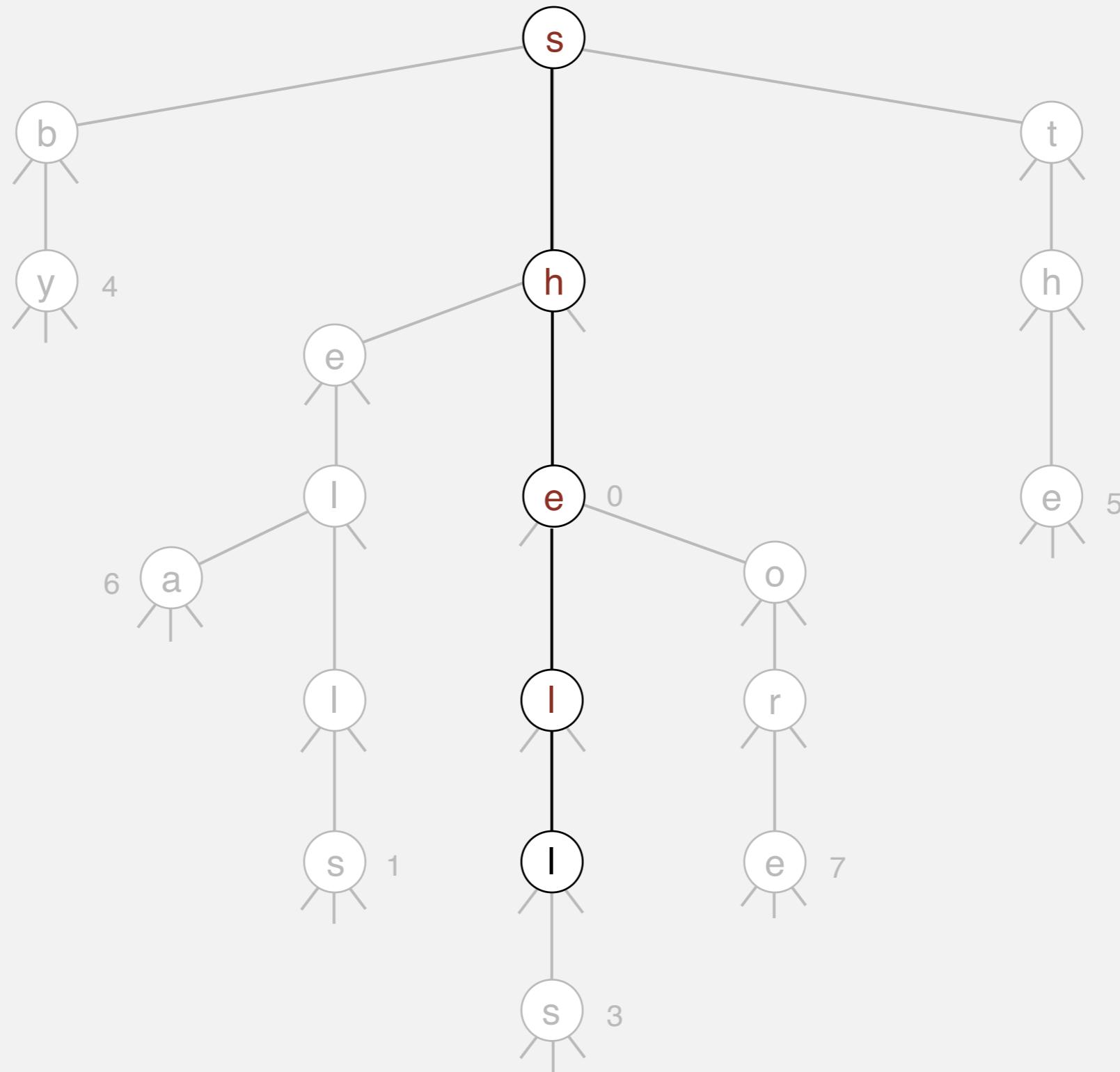
Search miss in a TST

get("shelter")



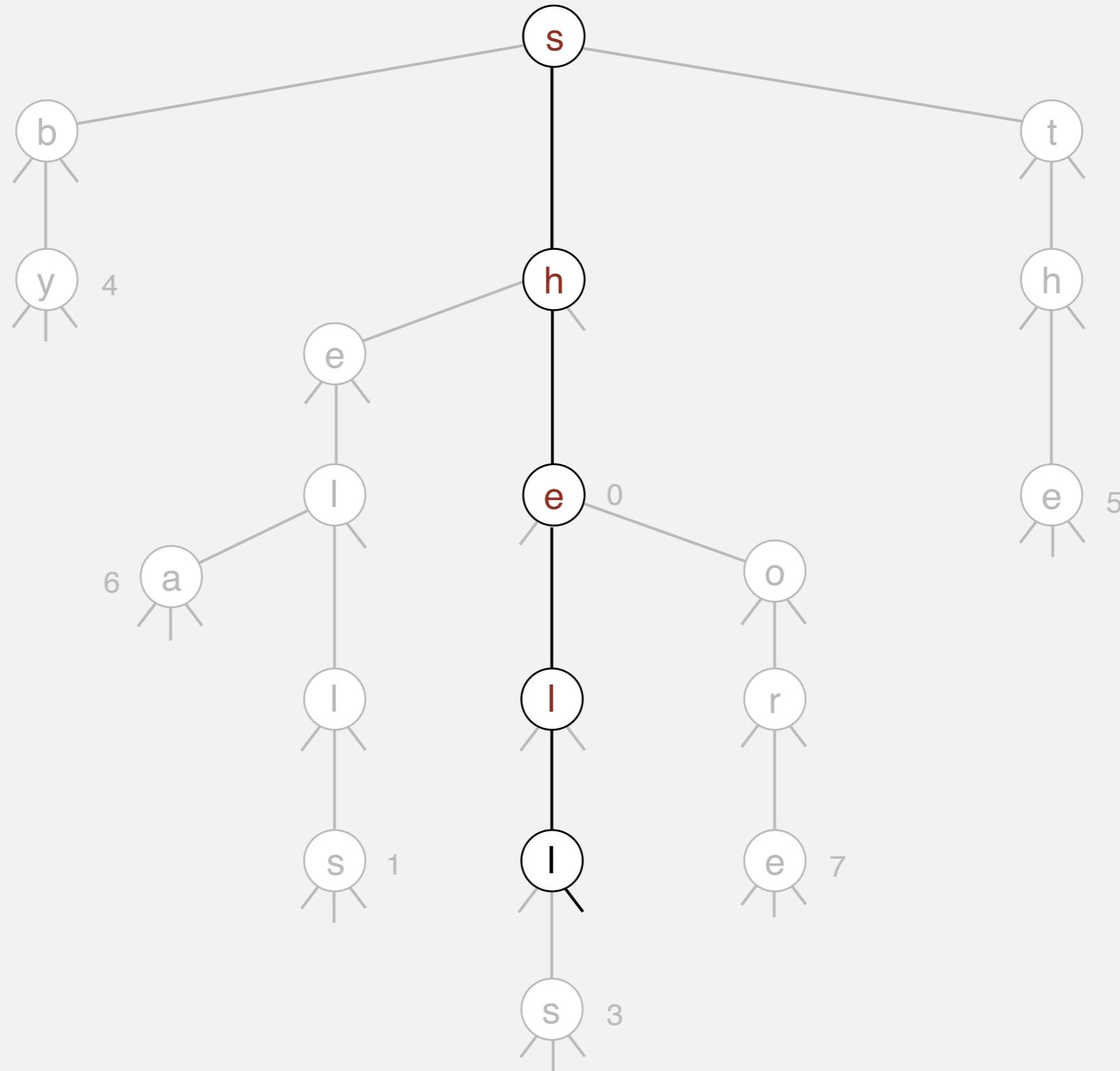
Search miss in a TST

get("shelter")



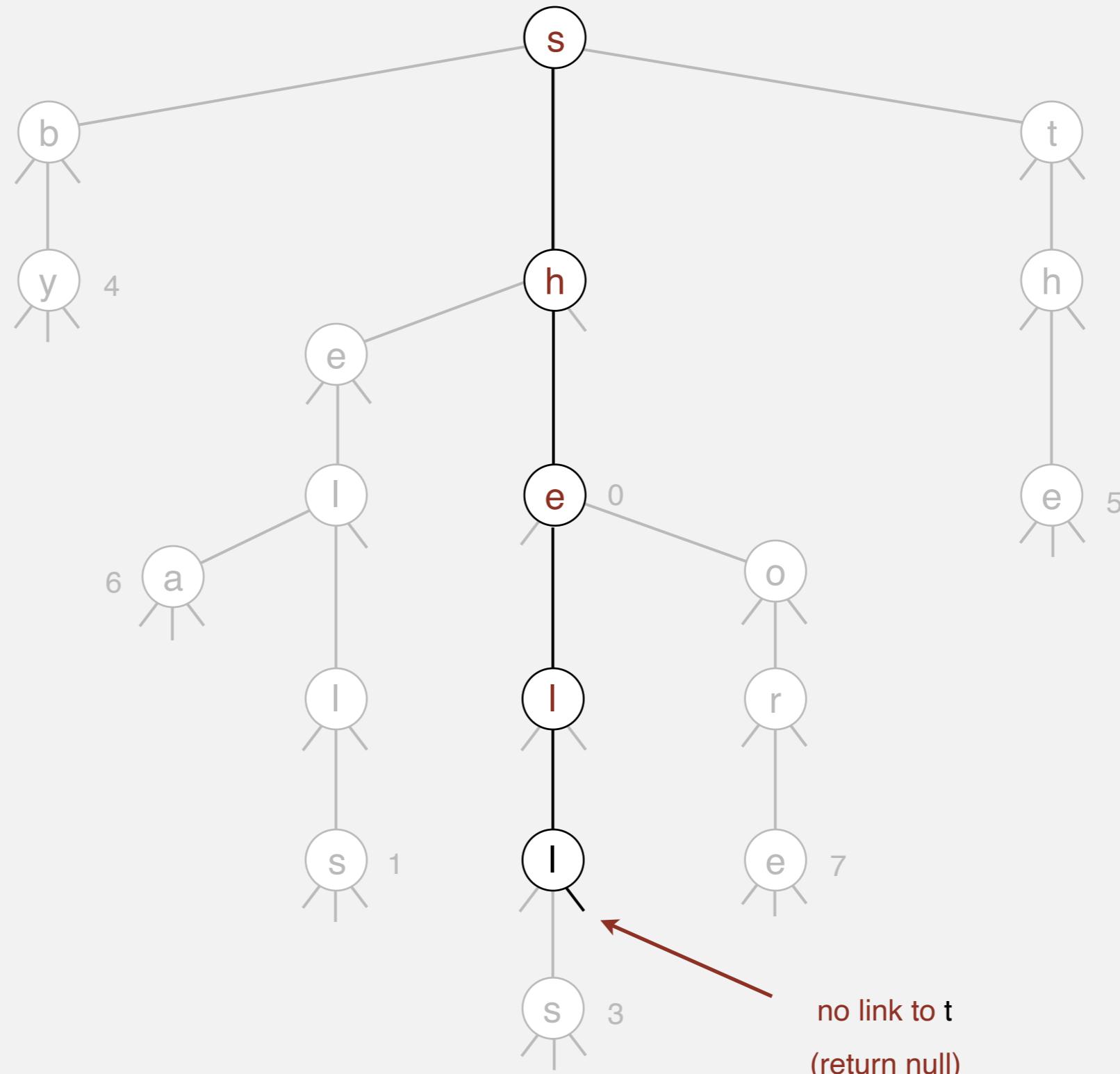
Search miss in a TST

get("shelter")



Search miss in a TST

get("shelter")



Ternary search trie construction demo

ternary search trie

Ternary search trie construction demo

```
put("she", 0)
```

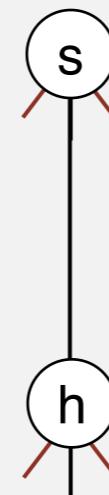
Ternary search trie construction demo

put("she", 0)



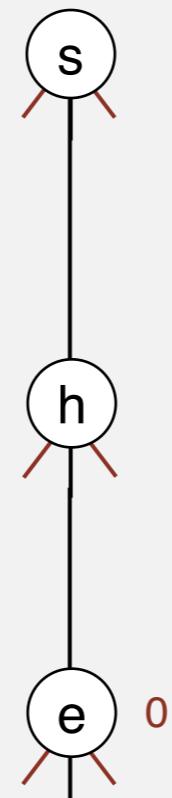
Ternary search trie construction demo

put("she", 0)



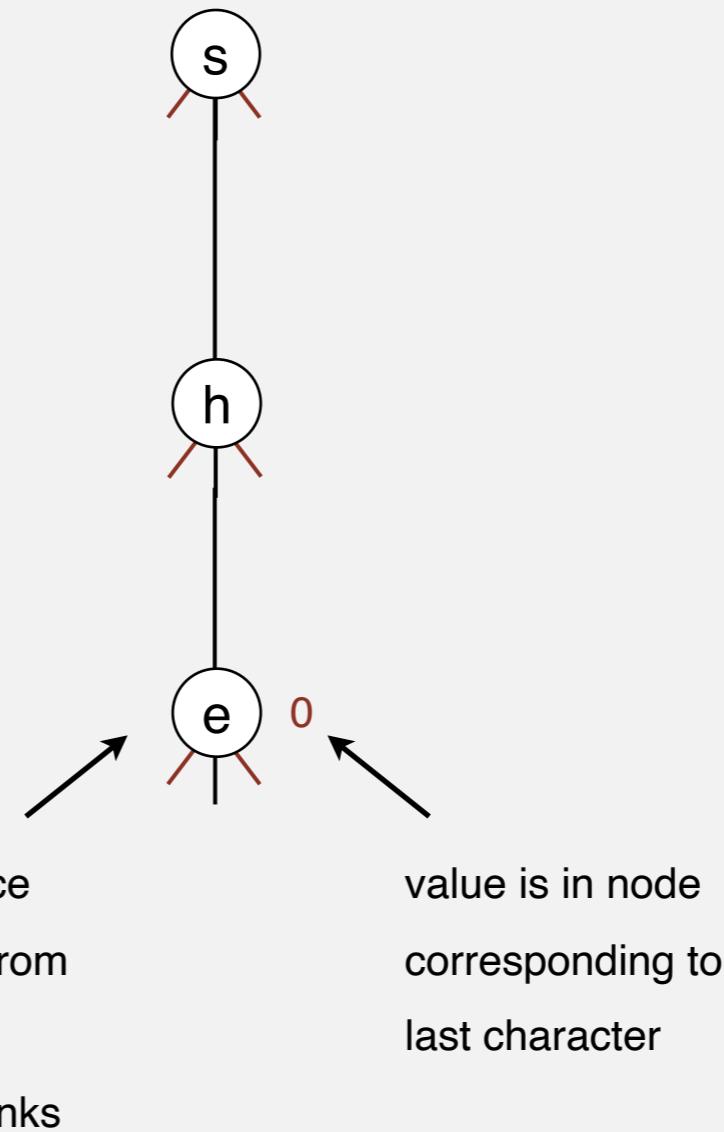
Ternary search trie construction demo

put("she", 0)



Ternary search trie construction demo

put("she", 0)



Ternary search trie construction demo

put("she", 0)



Ternary search trie construction demo

put("sells", 1)



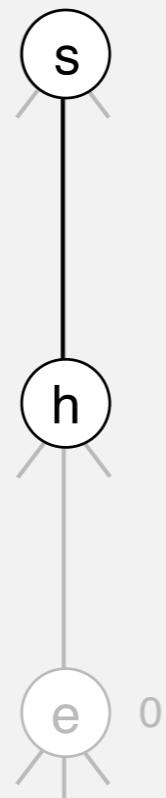
Ternary search trie construction demo

put("sells", 1)



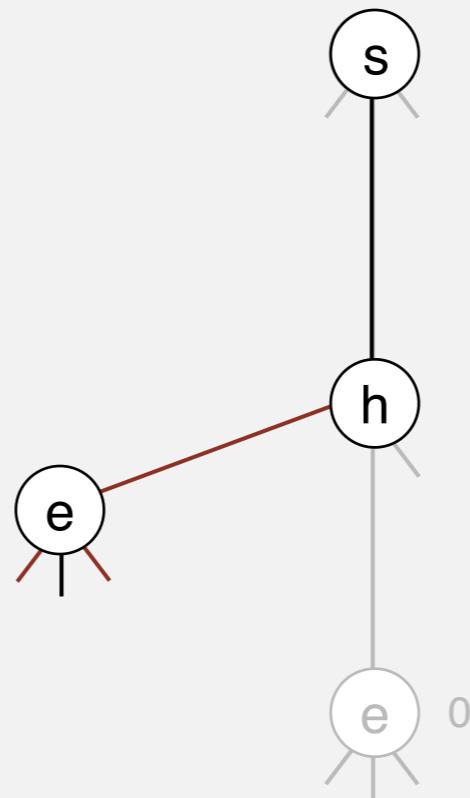
Ternary search trie construction demo

put("sells", 1)



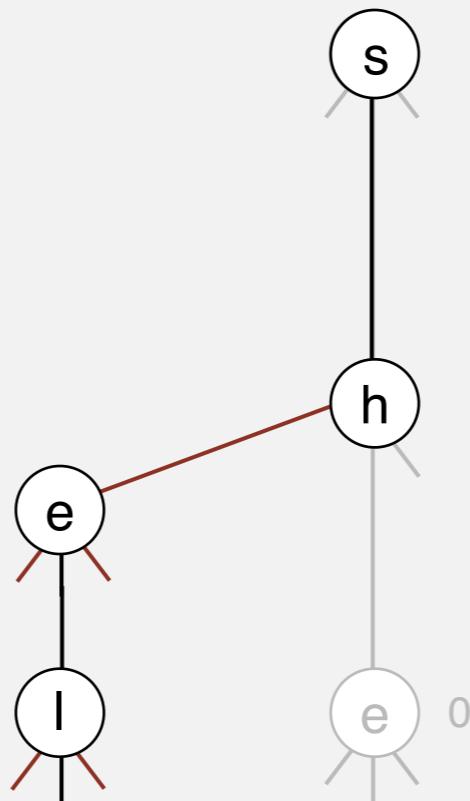
Ternary search trie construction demo

put("sells", 1)



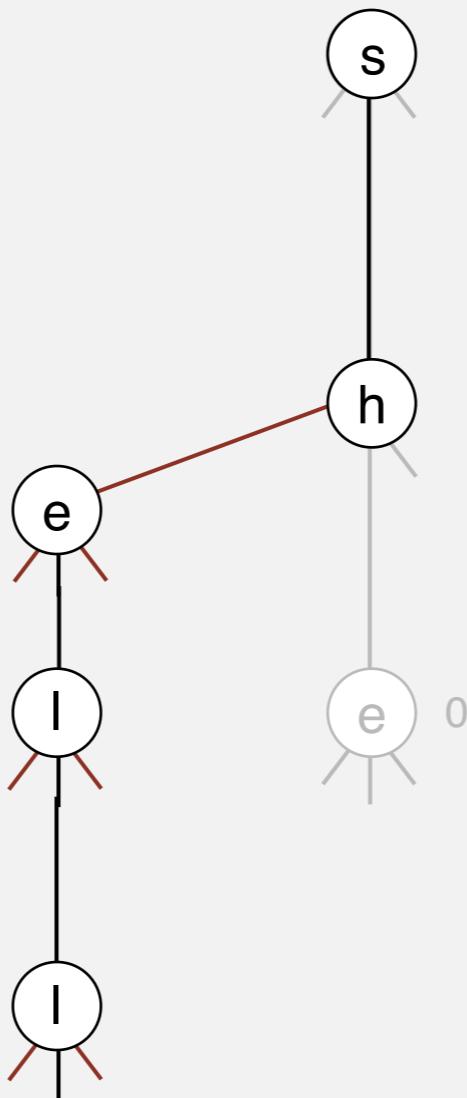
Ternary search trie construction demo

`put("sells", 1)`



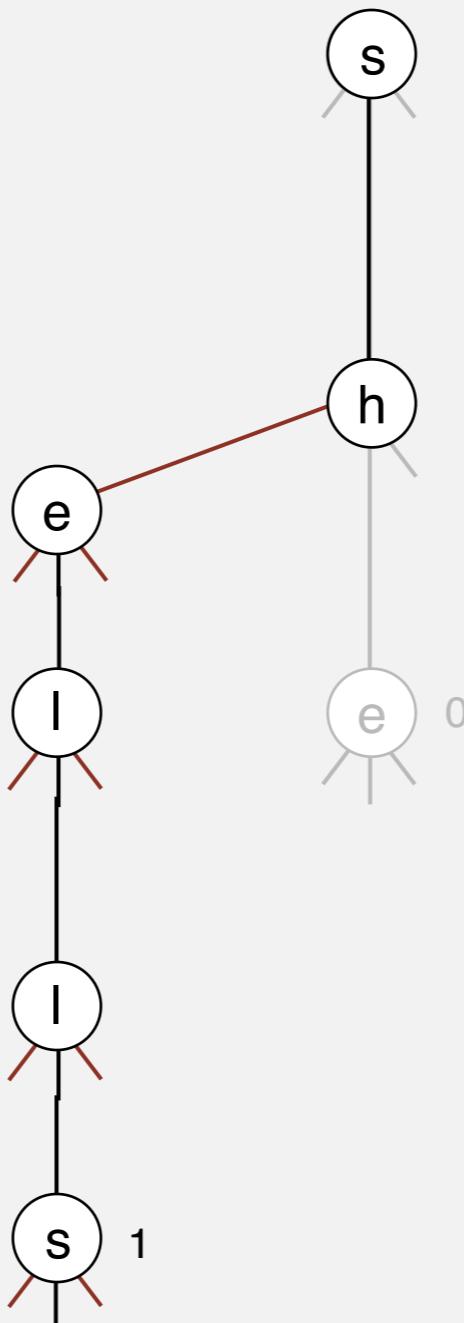
Ternary search trie construction demo

`put("sells", 1)`



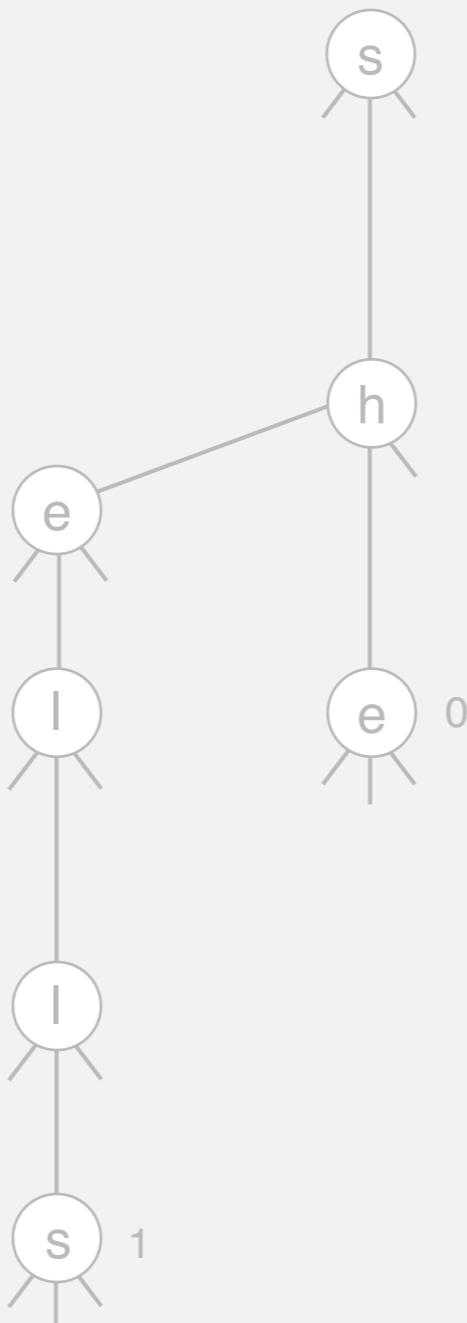
Ternary search trie construction demo

`put("sells", 1)`



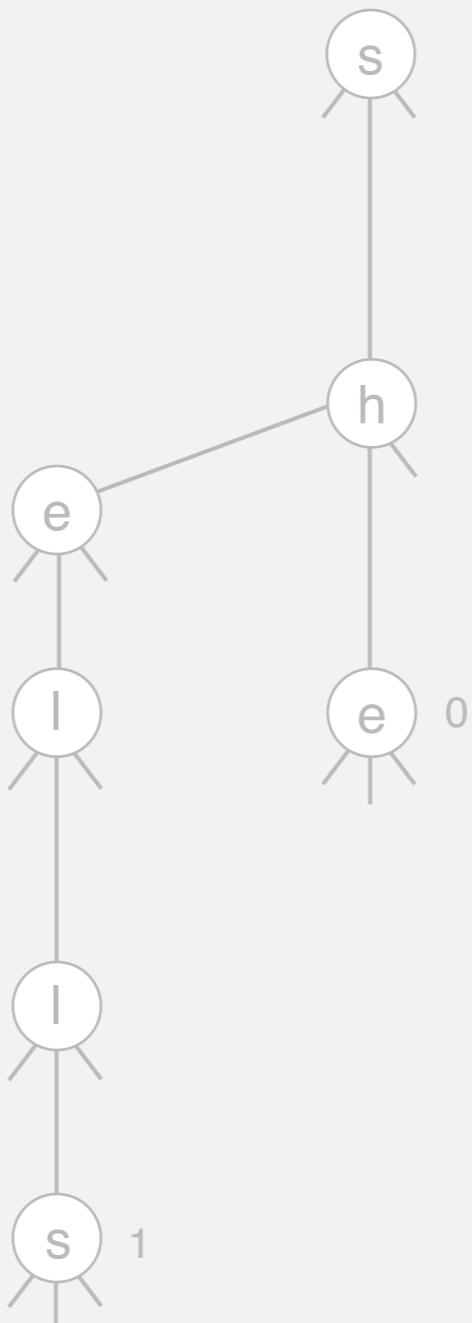
Ternary search trie construction demo

ternary search trie



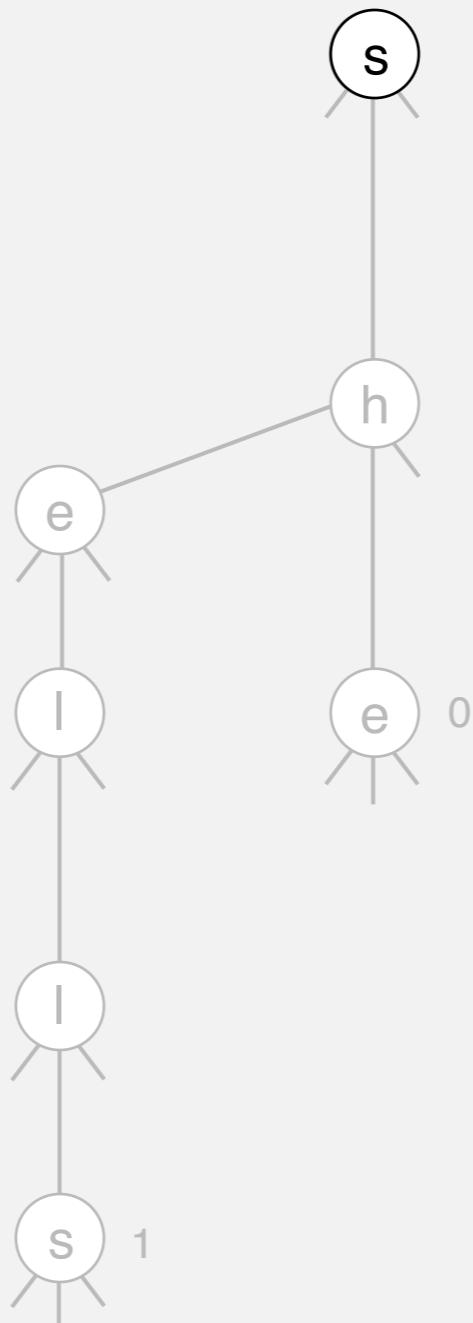
Ternary search trie construction demo

put("sea", 2)



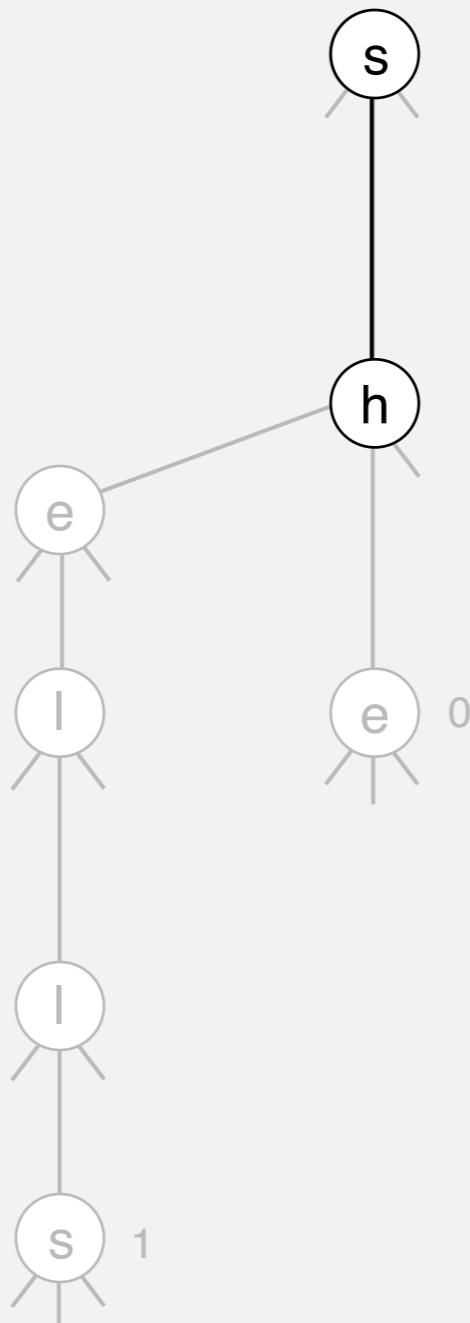
Ternary search trie construction demo

`put("sea", 2)`



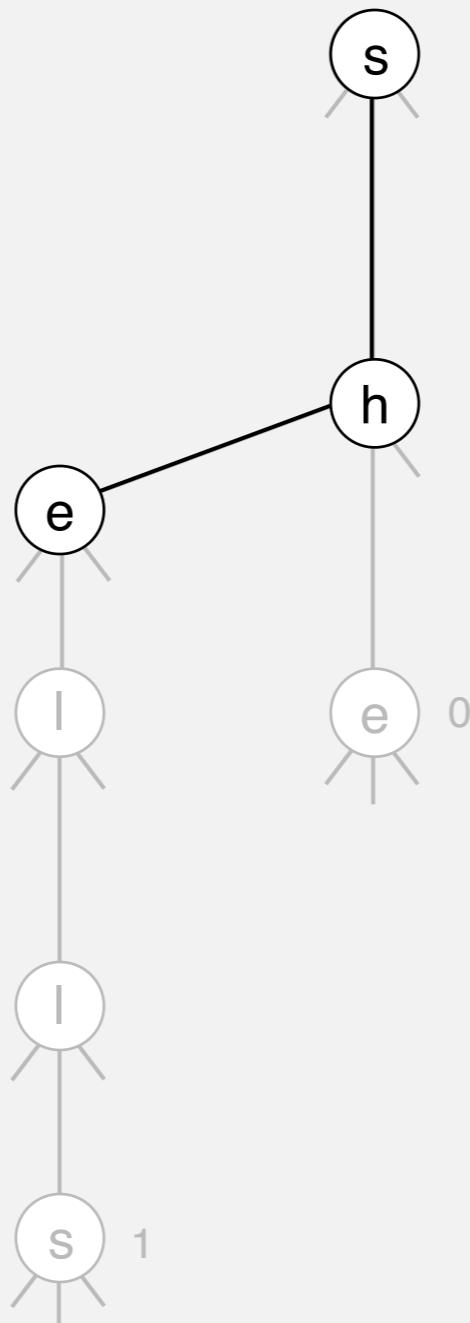
Ternary search trie construction demo

put("sea", 2)



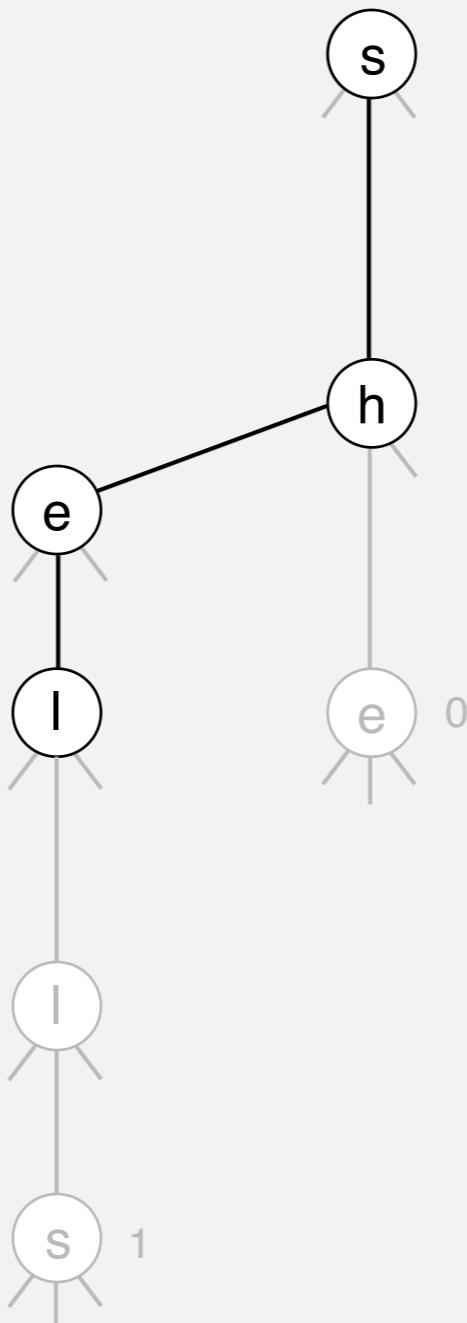
Ternary search trie construction demo

`put("sea", 2)`



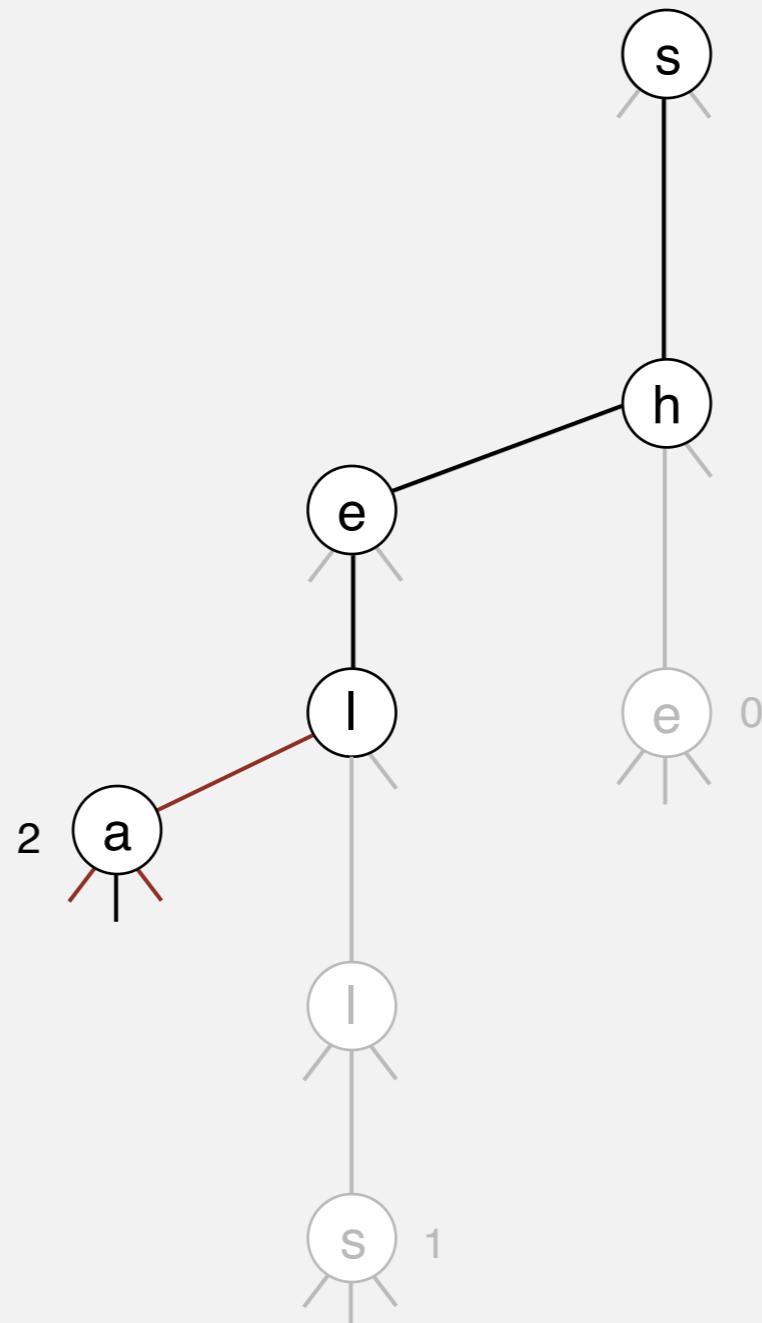
Ternary search trie construction demo

`put("sea", 2)`



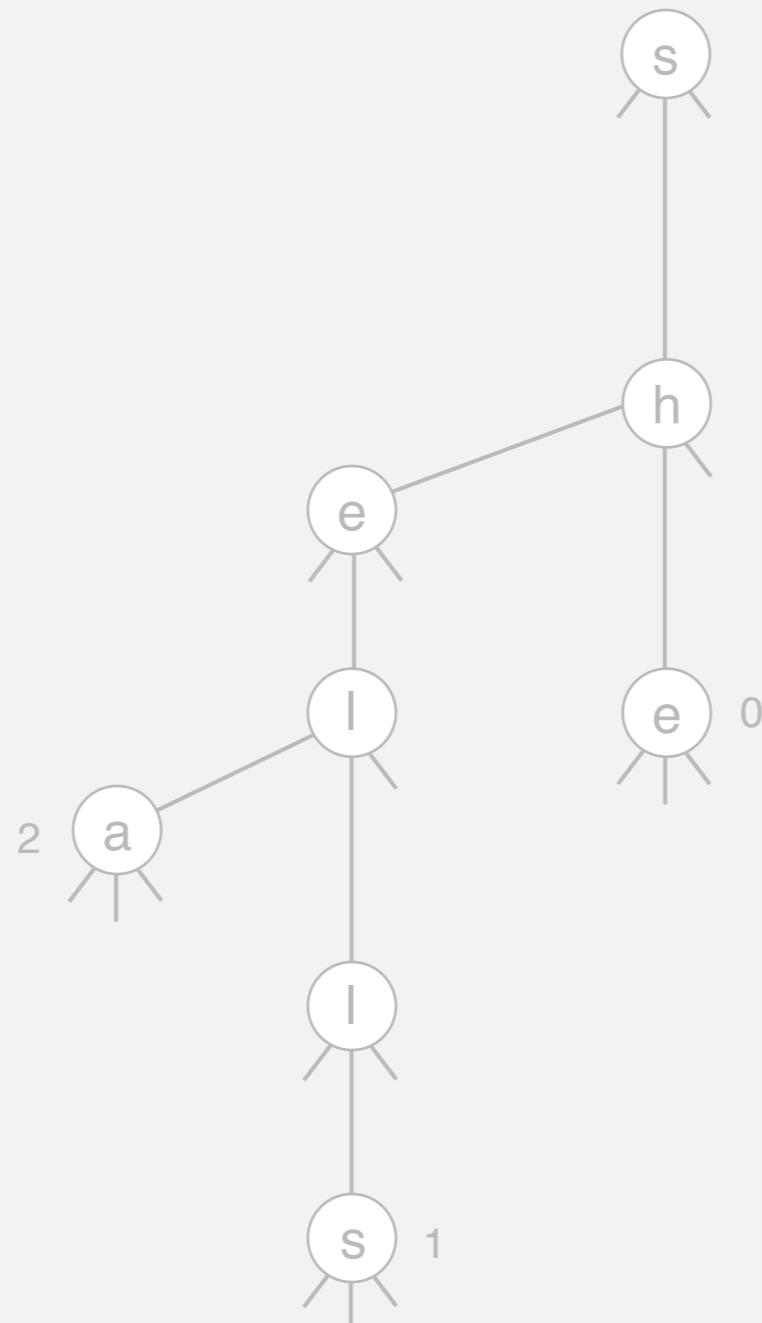
Ternary search trie construction demo

`put("sea", 2)`



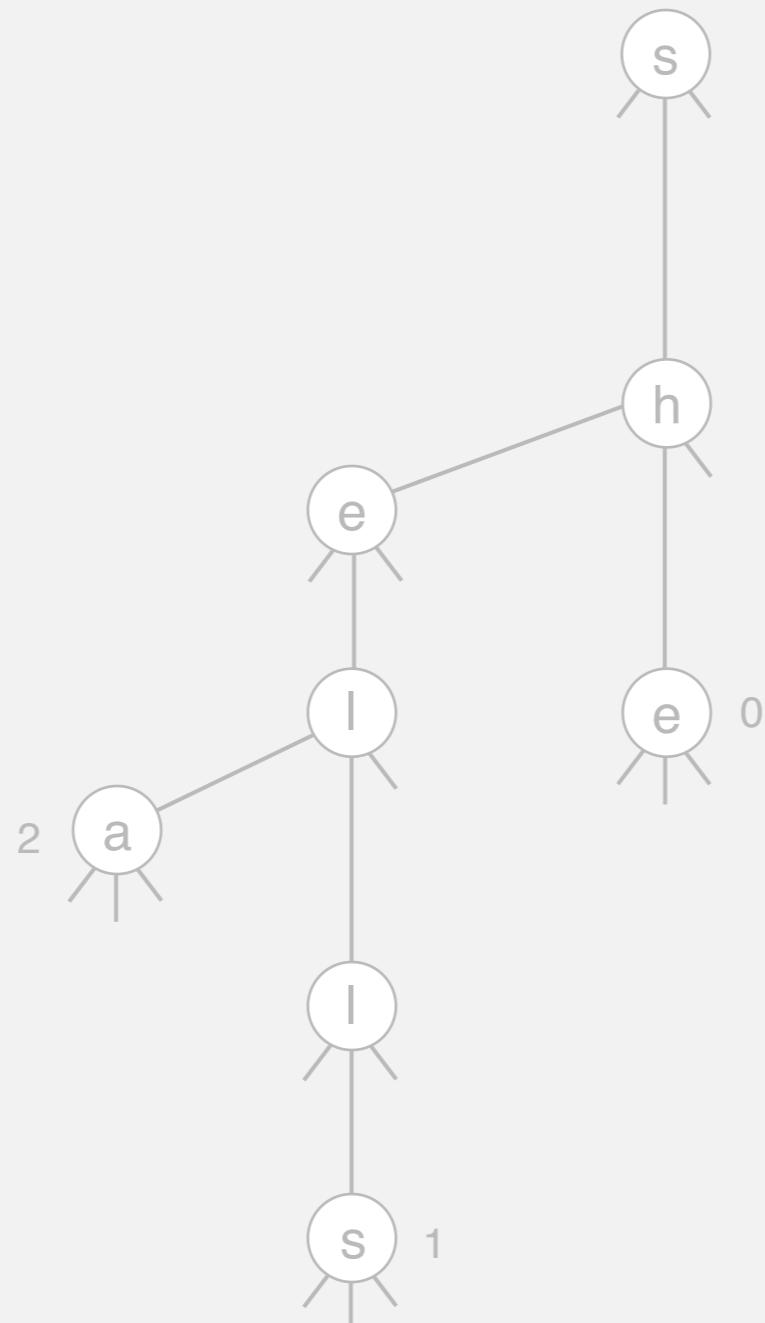
Ternary search trie construction demo

ternary search trie



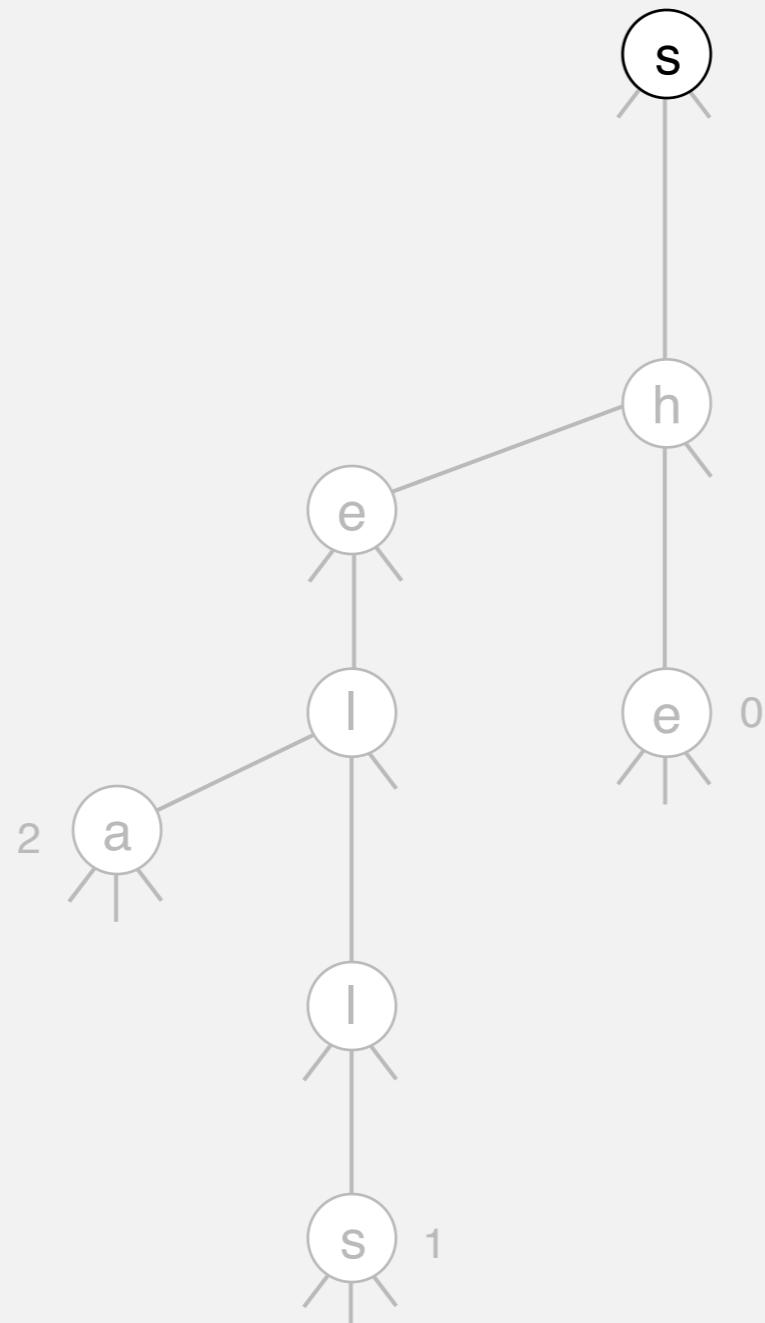
Ternary search trie construction demo

put("shells", 3)



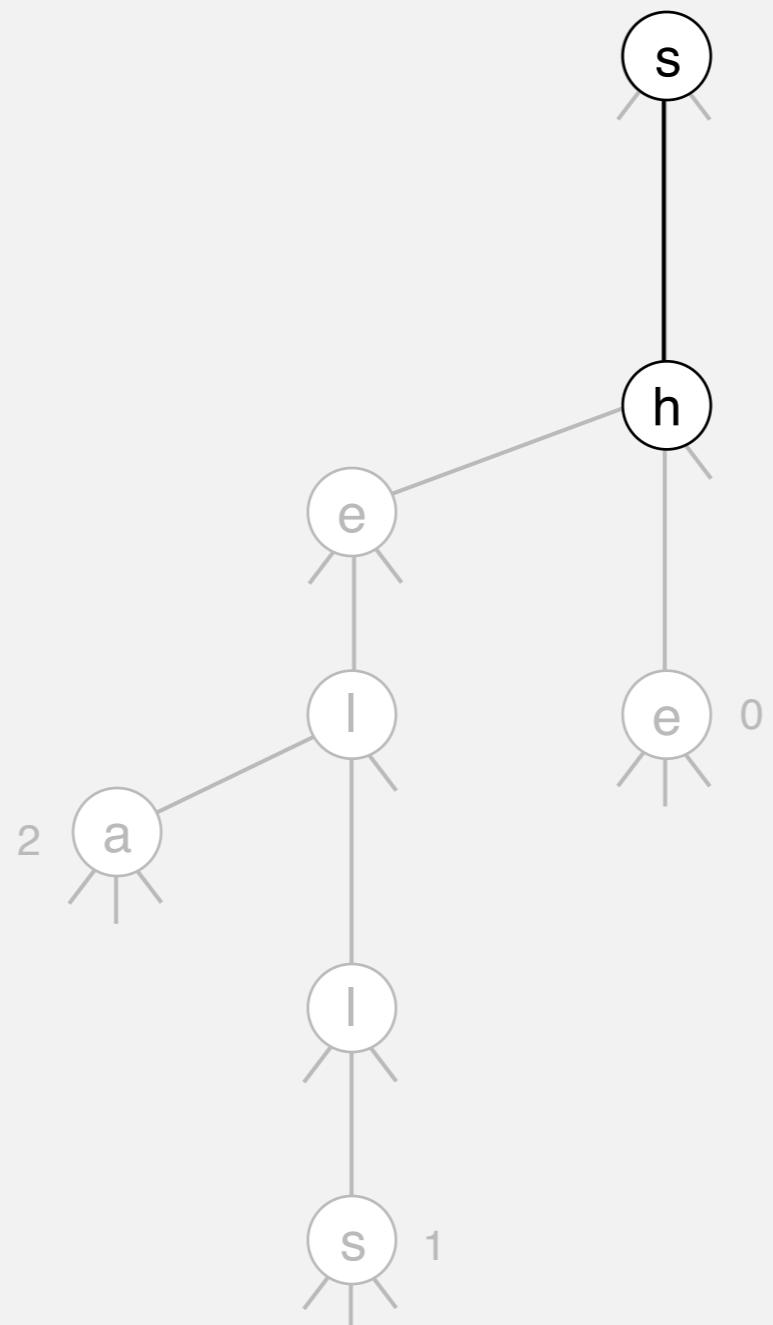
Ternary search trie construction demo

put("shells", 3)



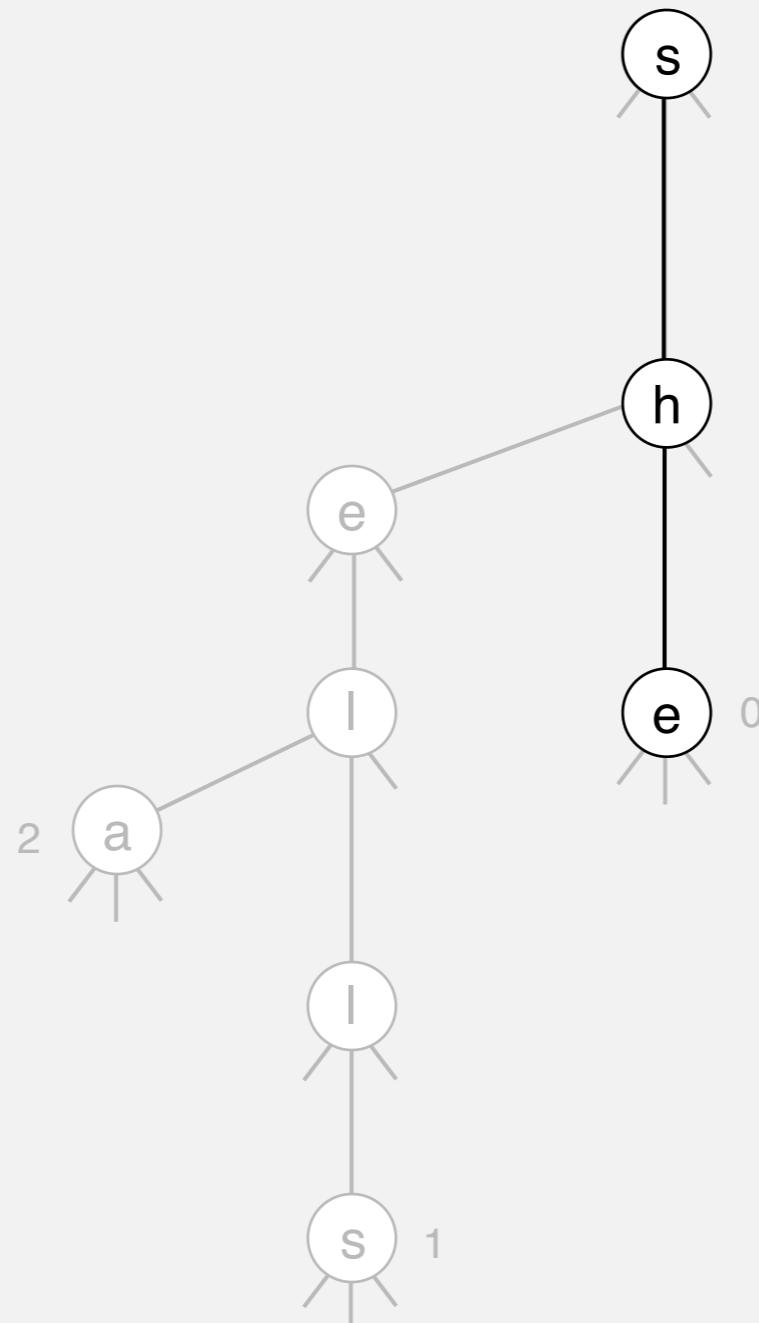
Ternary search trie construction demo

```
put("shells", 3)
```



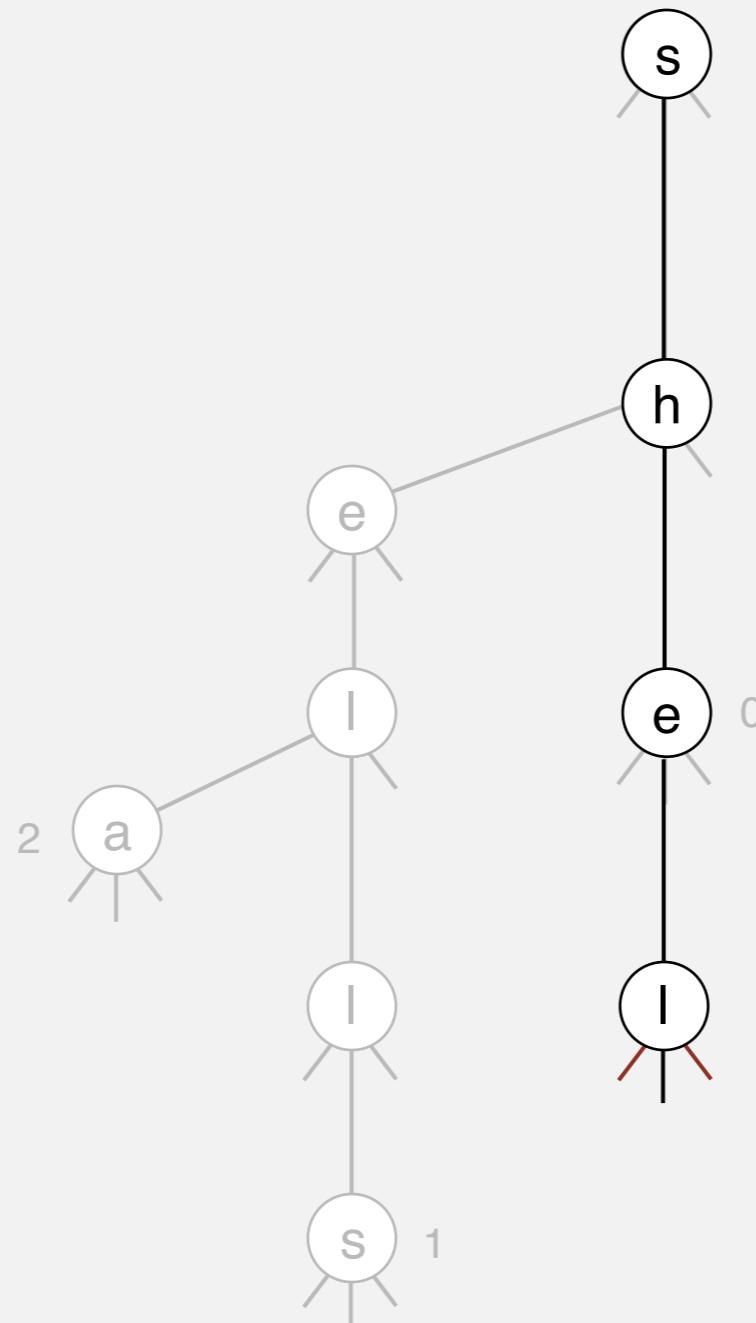
Ternary search trie construction demo

put("shells", 3)



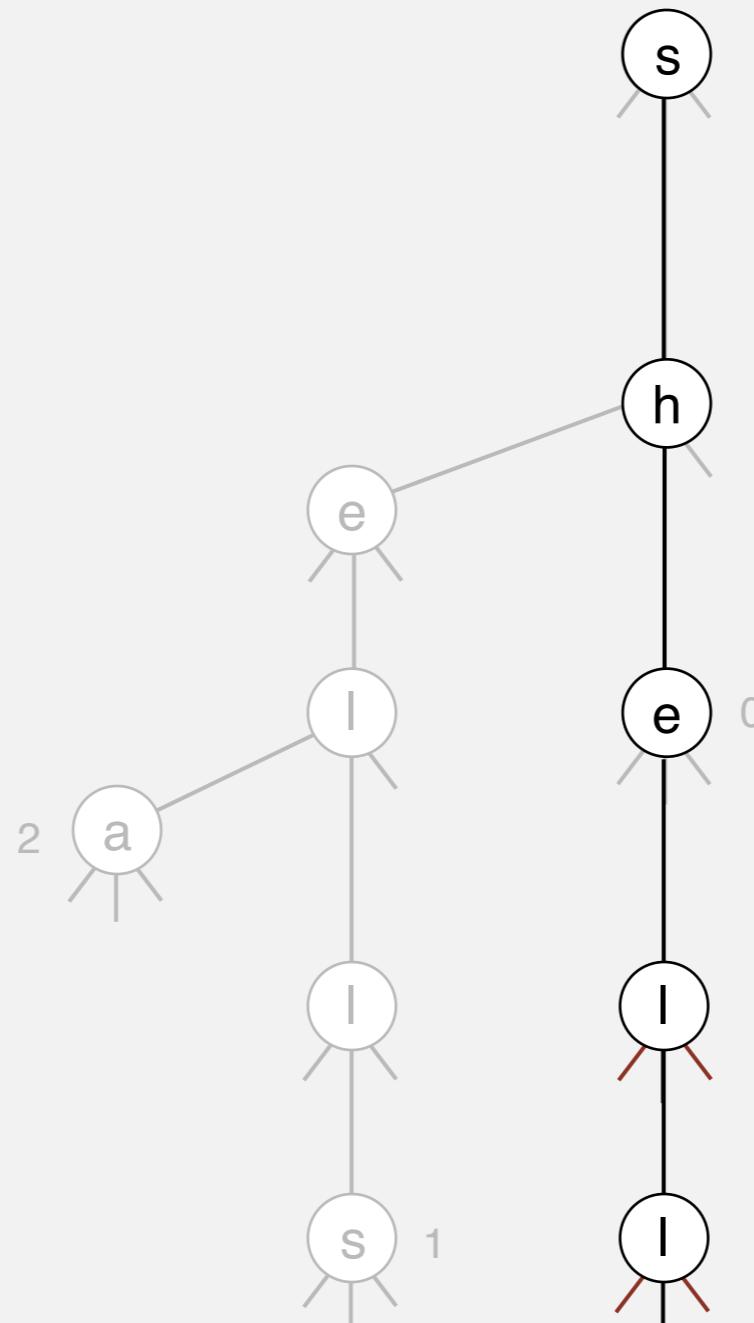
Ternary search trie construction demo

put("shells", 3)



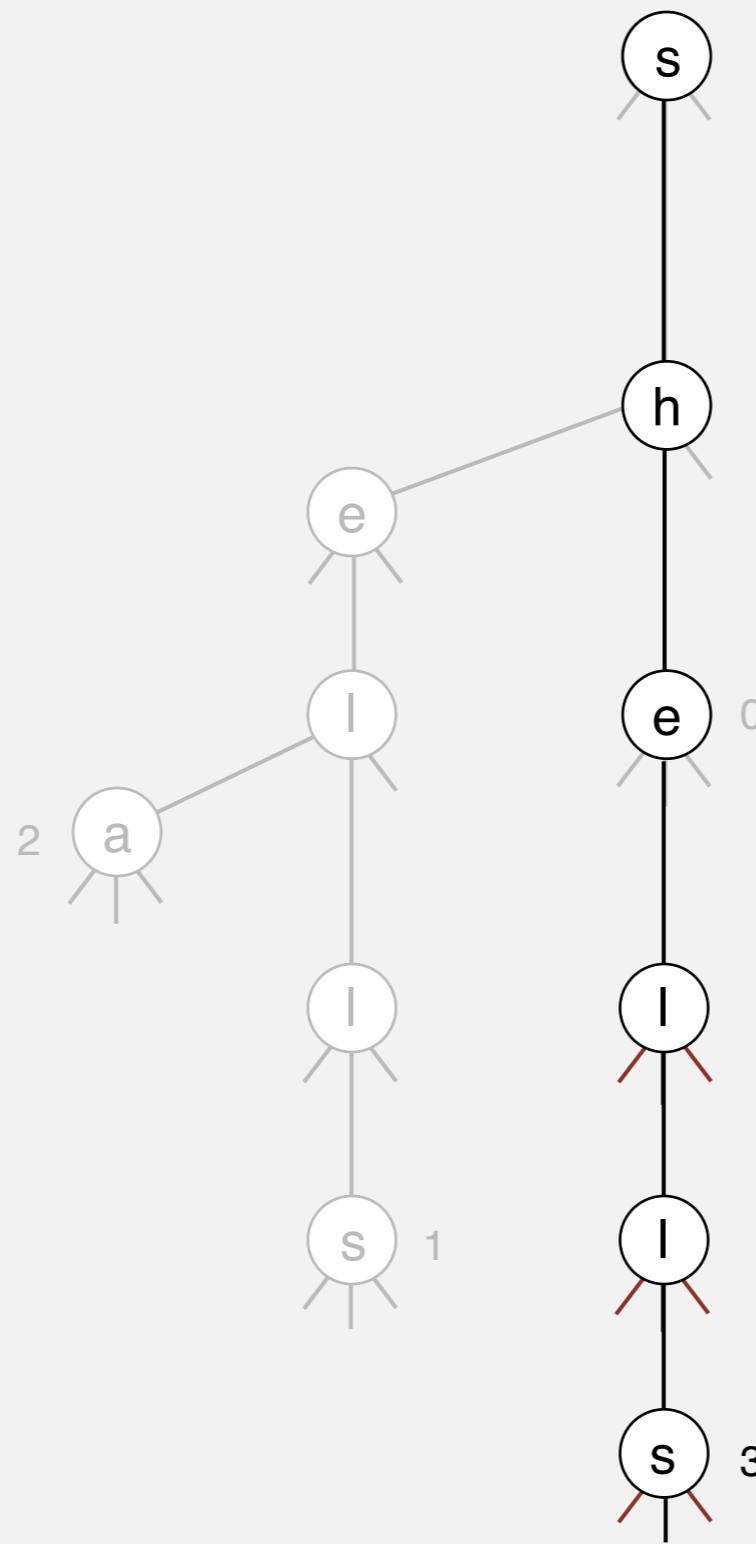
Ternary search trie construction demo

```
put("shells", 3)
```



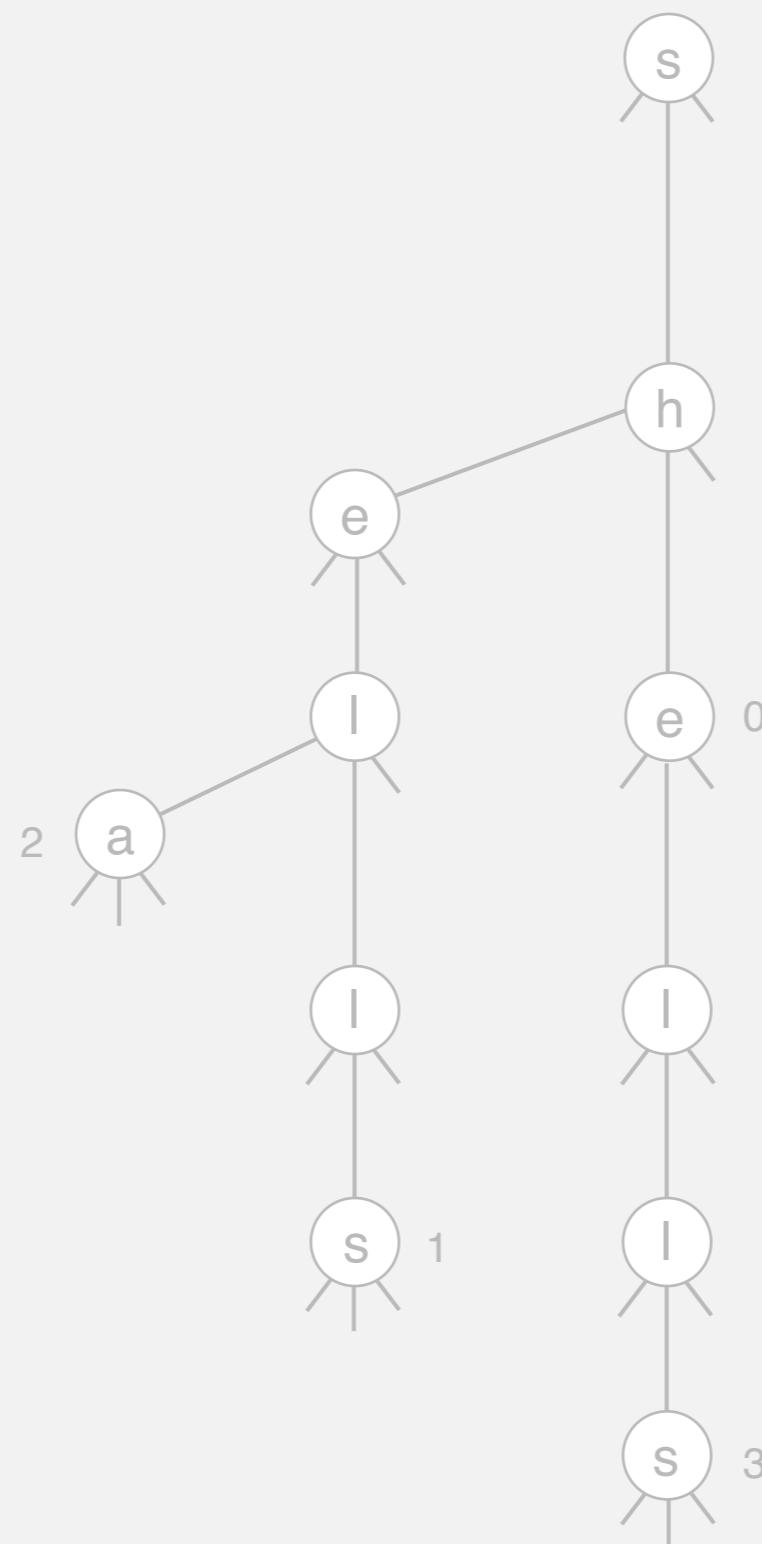
Ternary search trie construction demo

put("shells", 3)



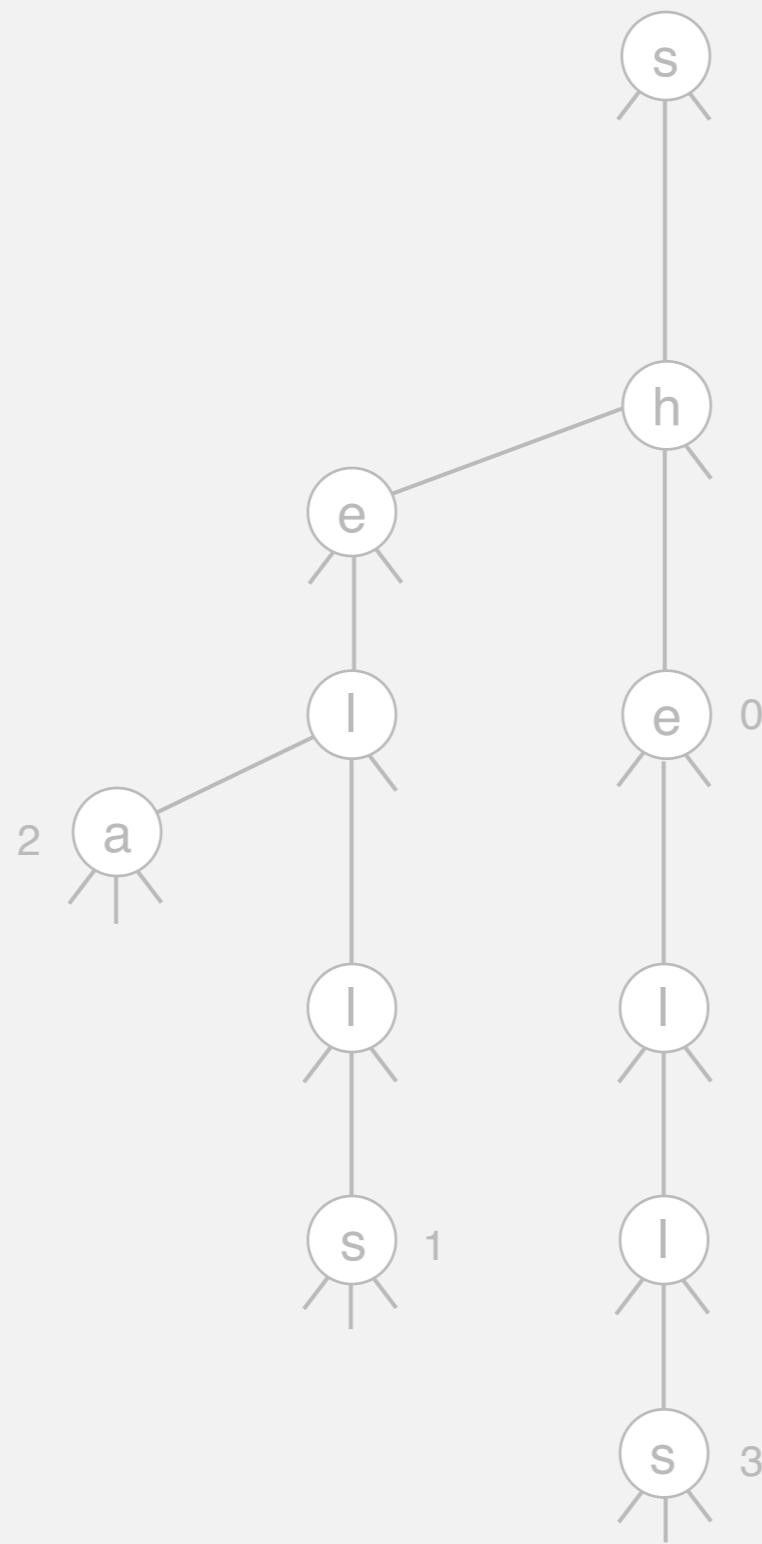
Ternary search trie construction demo

ternary search trie



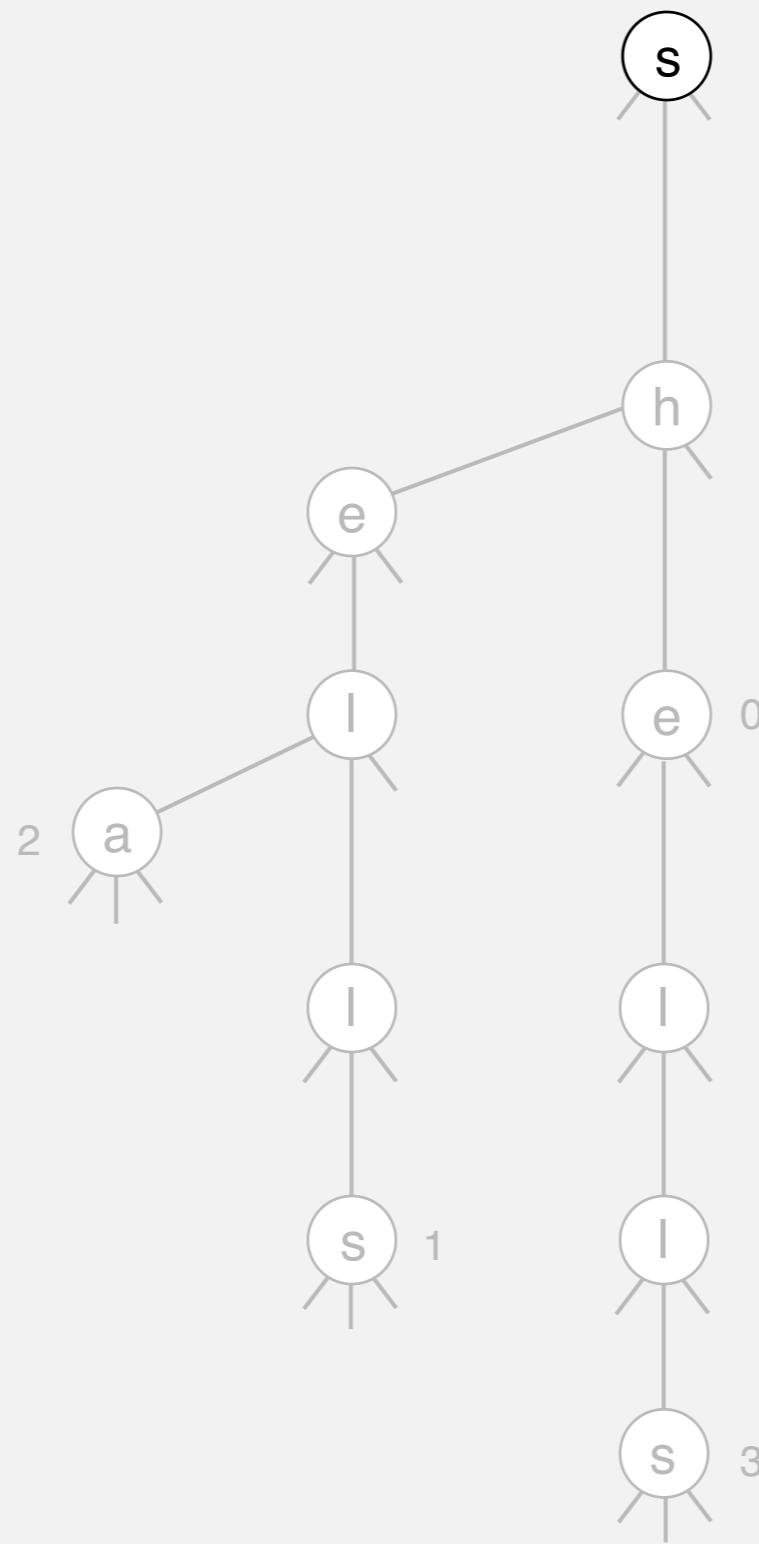
Ternary search trie construction demo

put("by", 4)



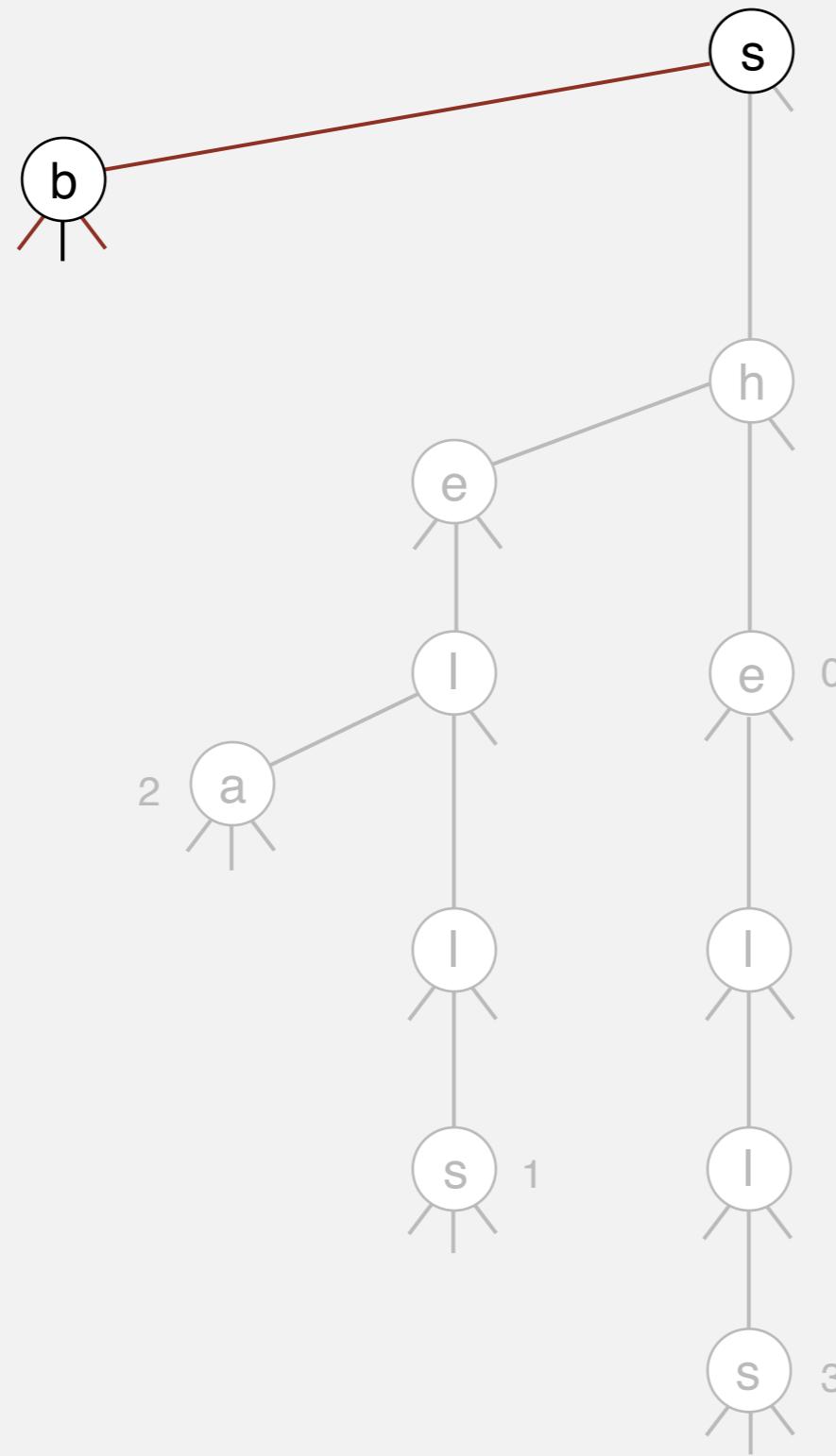
Ternary search trie construction demo

put("by", 4)



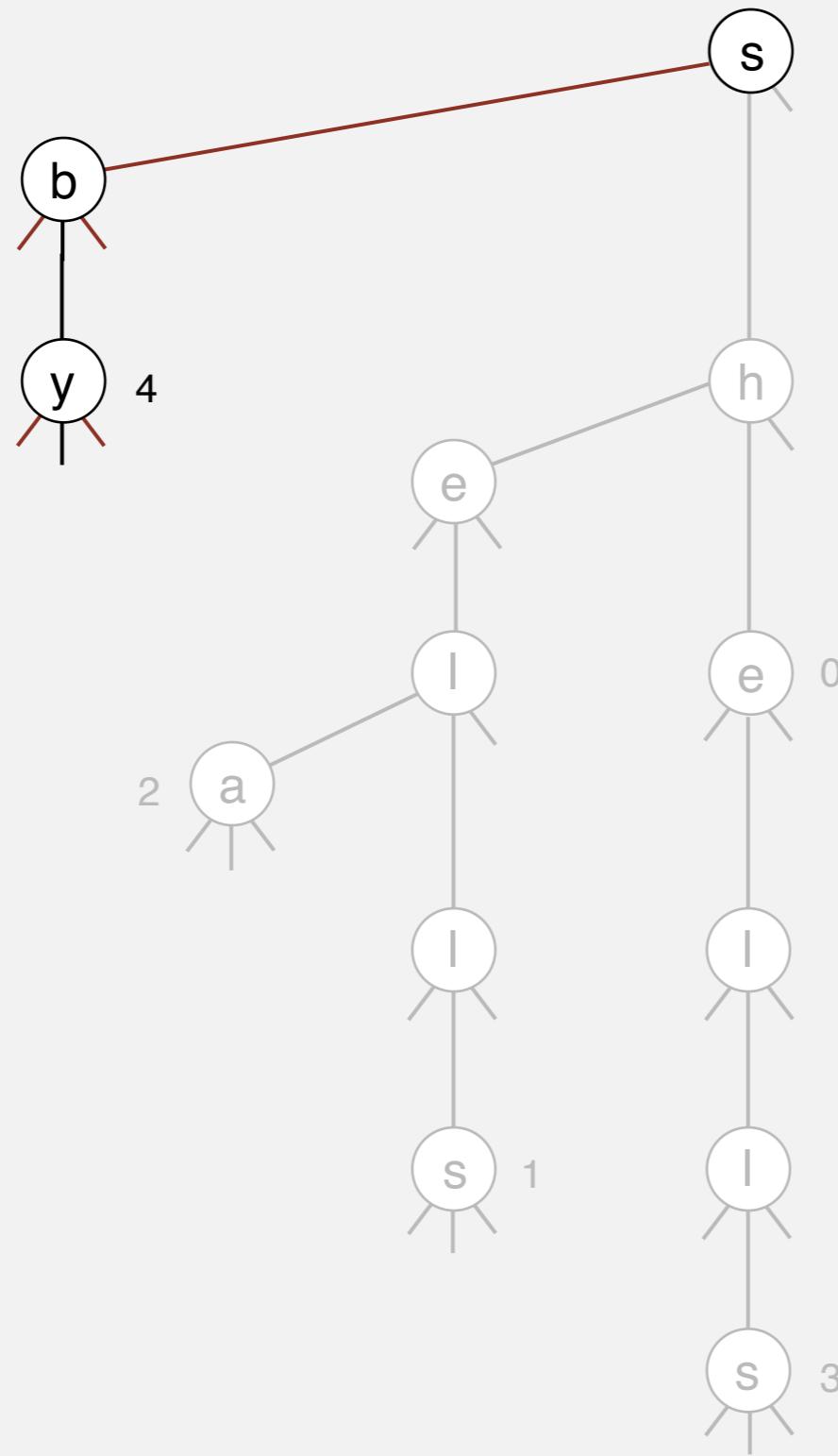
Ternary search trie construction demo

put("by", 4)



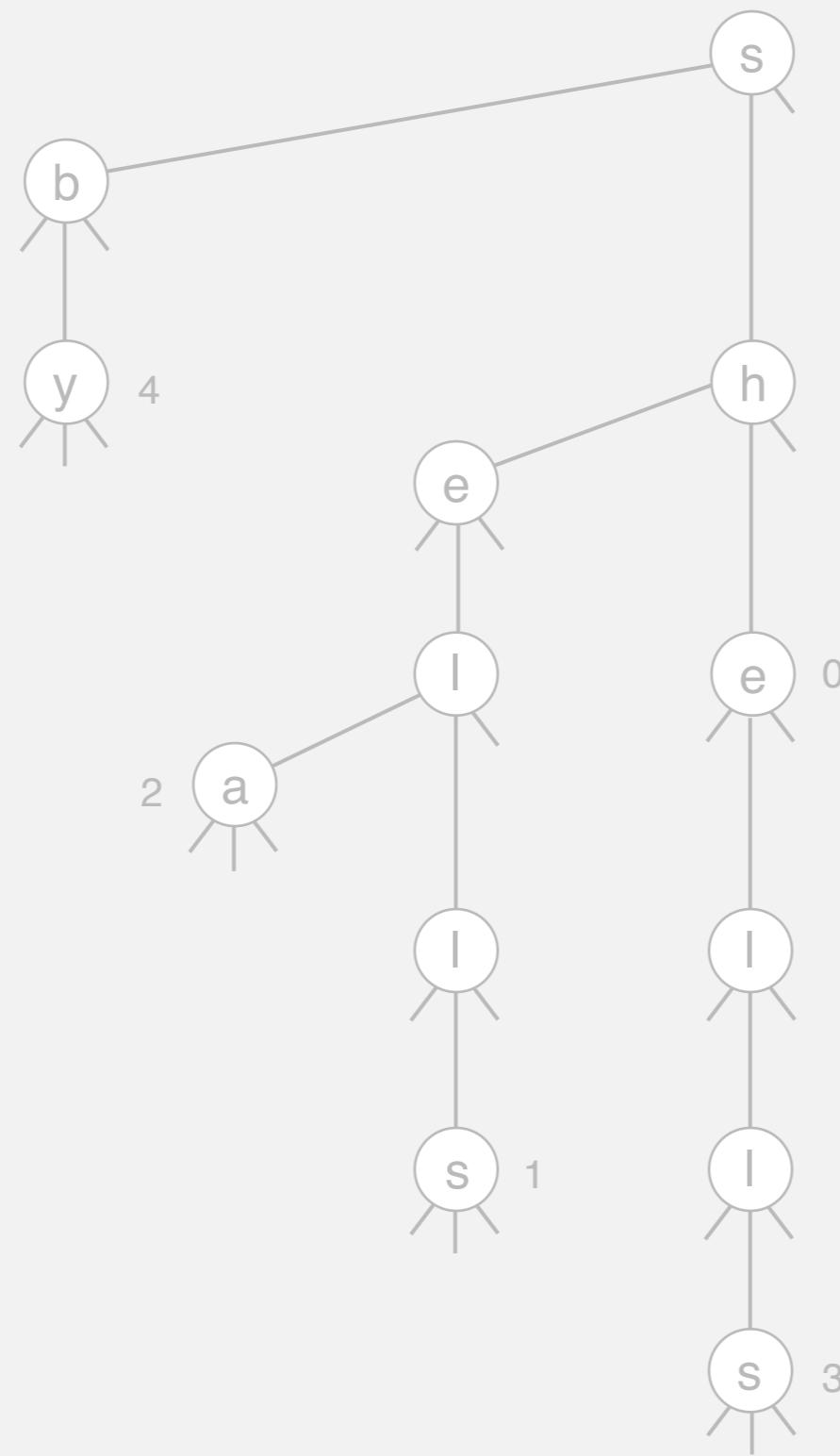
Ternary search trie construction demo

put("by", 4)



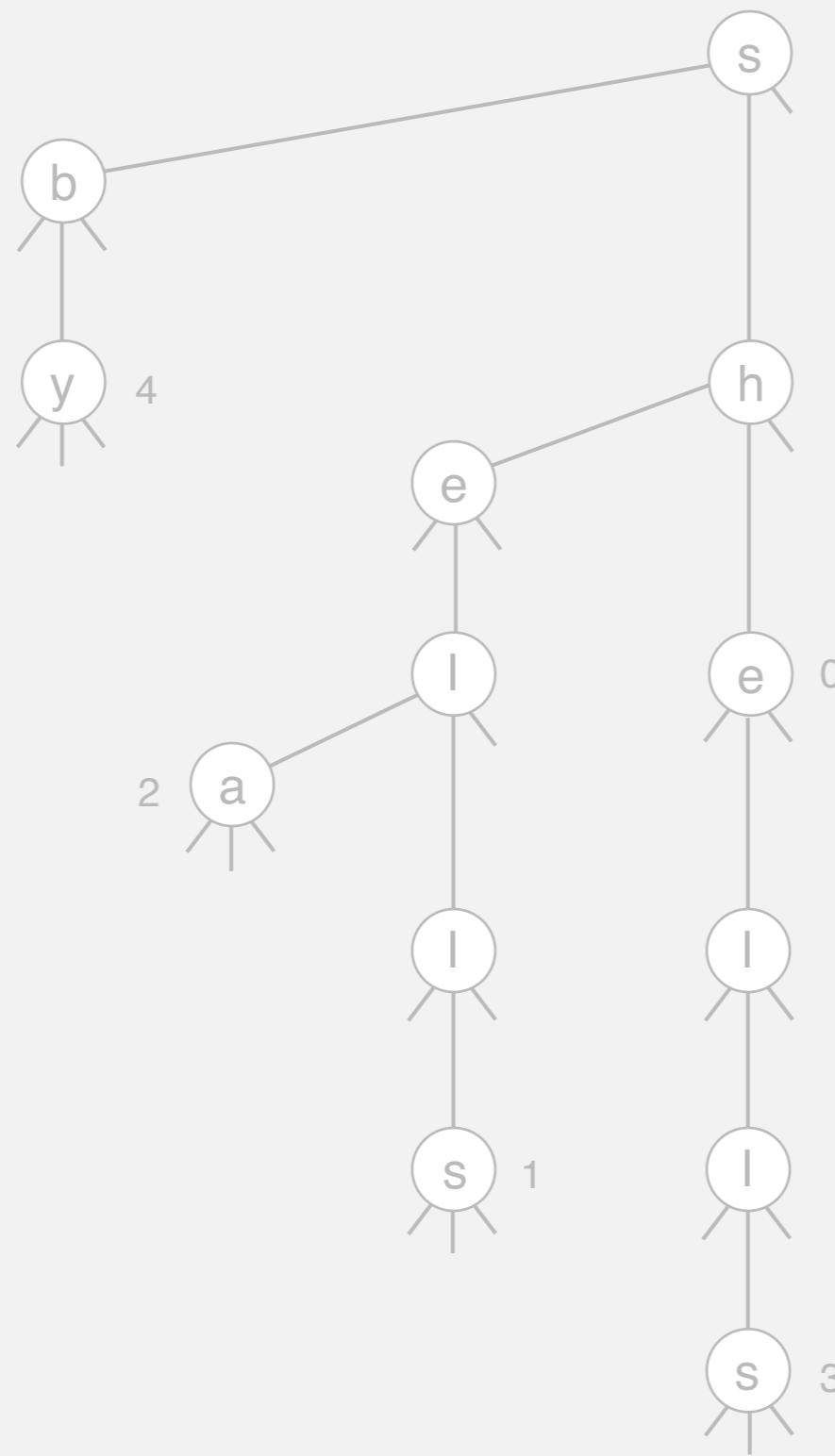
Ternary search trie construction demo

ternary search trie



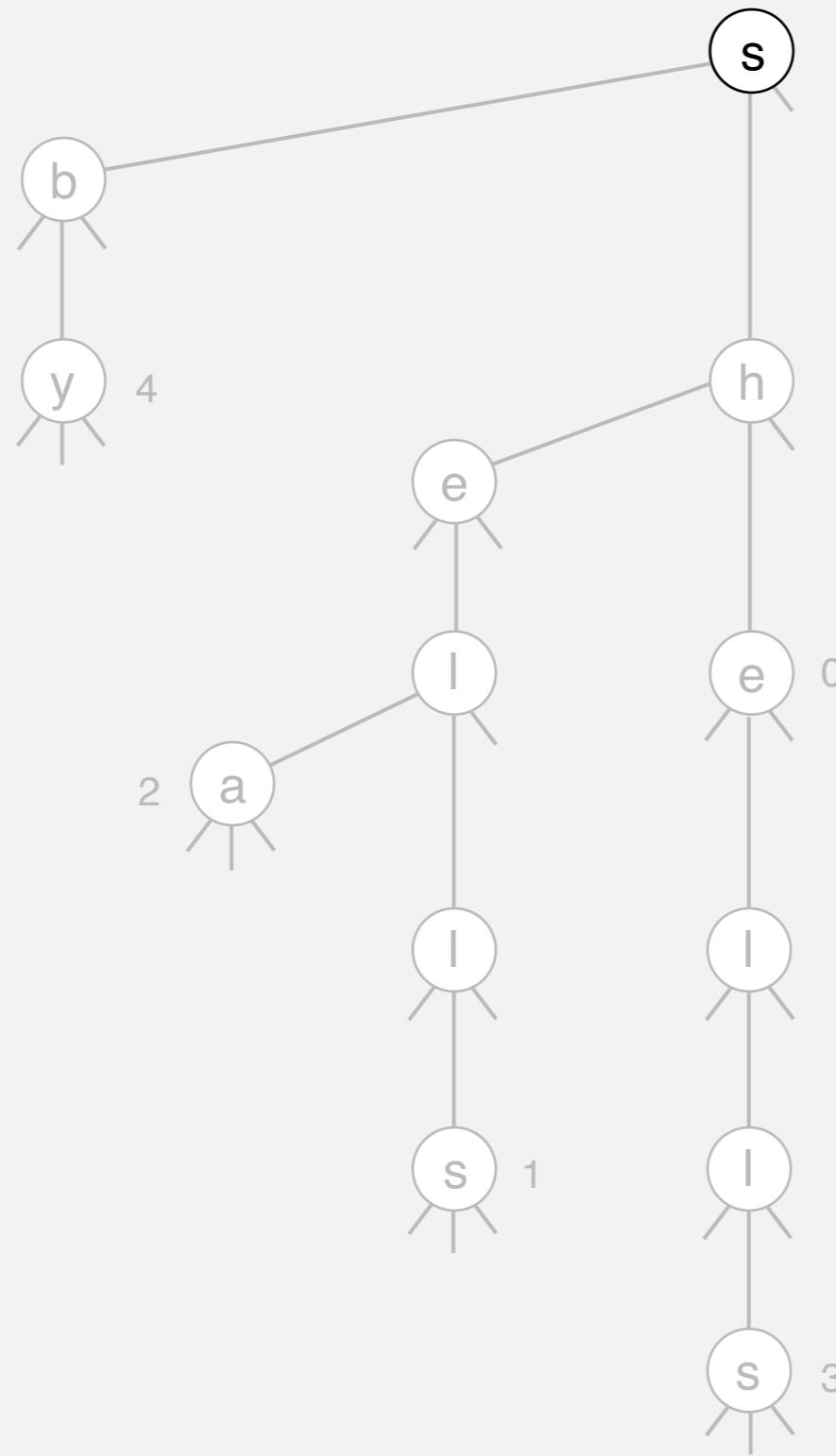
Ternary search trie construction demo

put("the", 5)



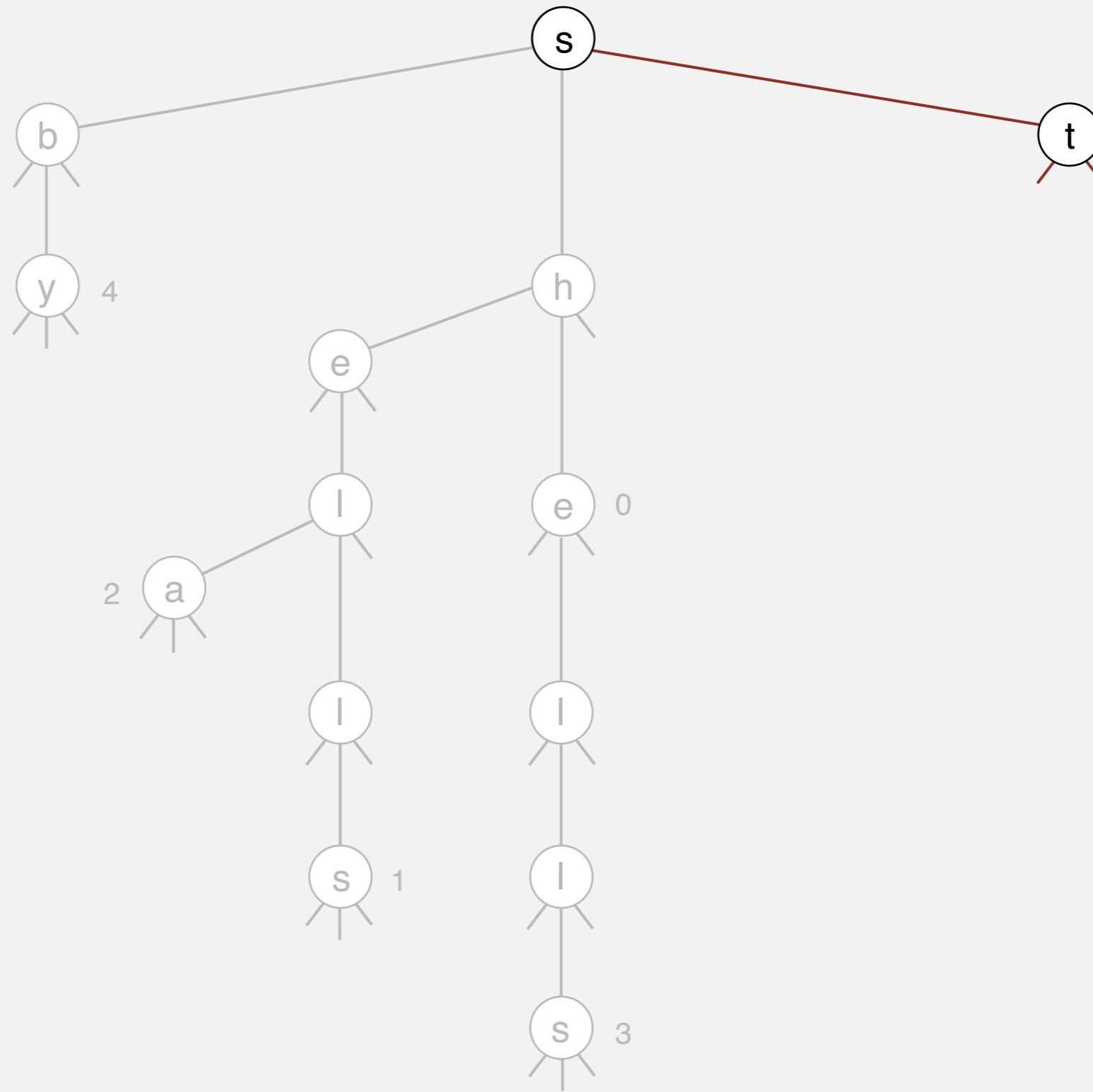
Ternary search trie construction demo

put("the", 5)



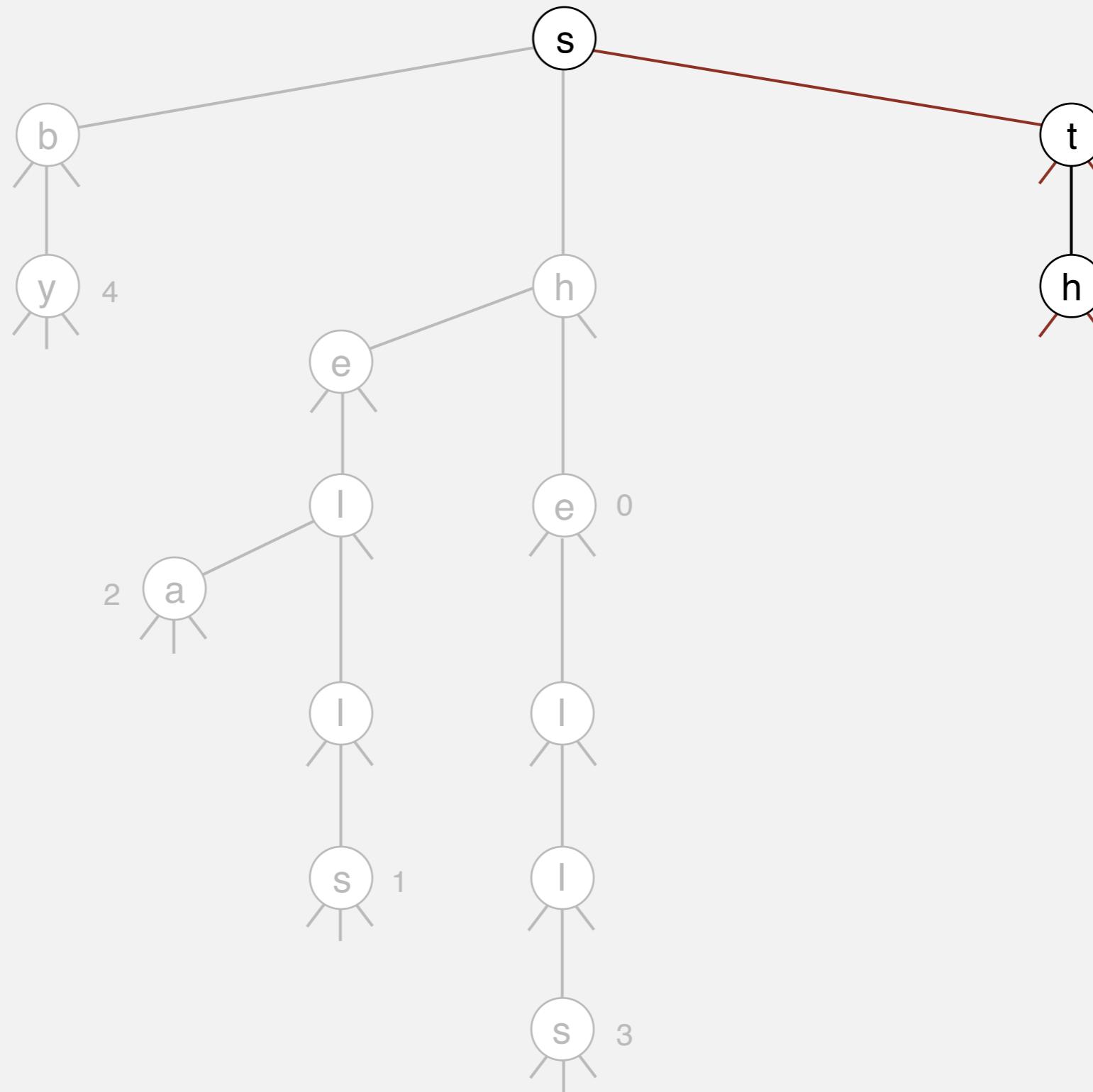
Ternary search trie construction demo

put("the", 5)



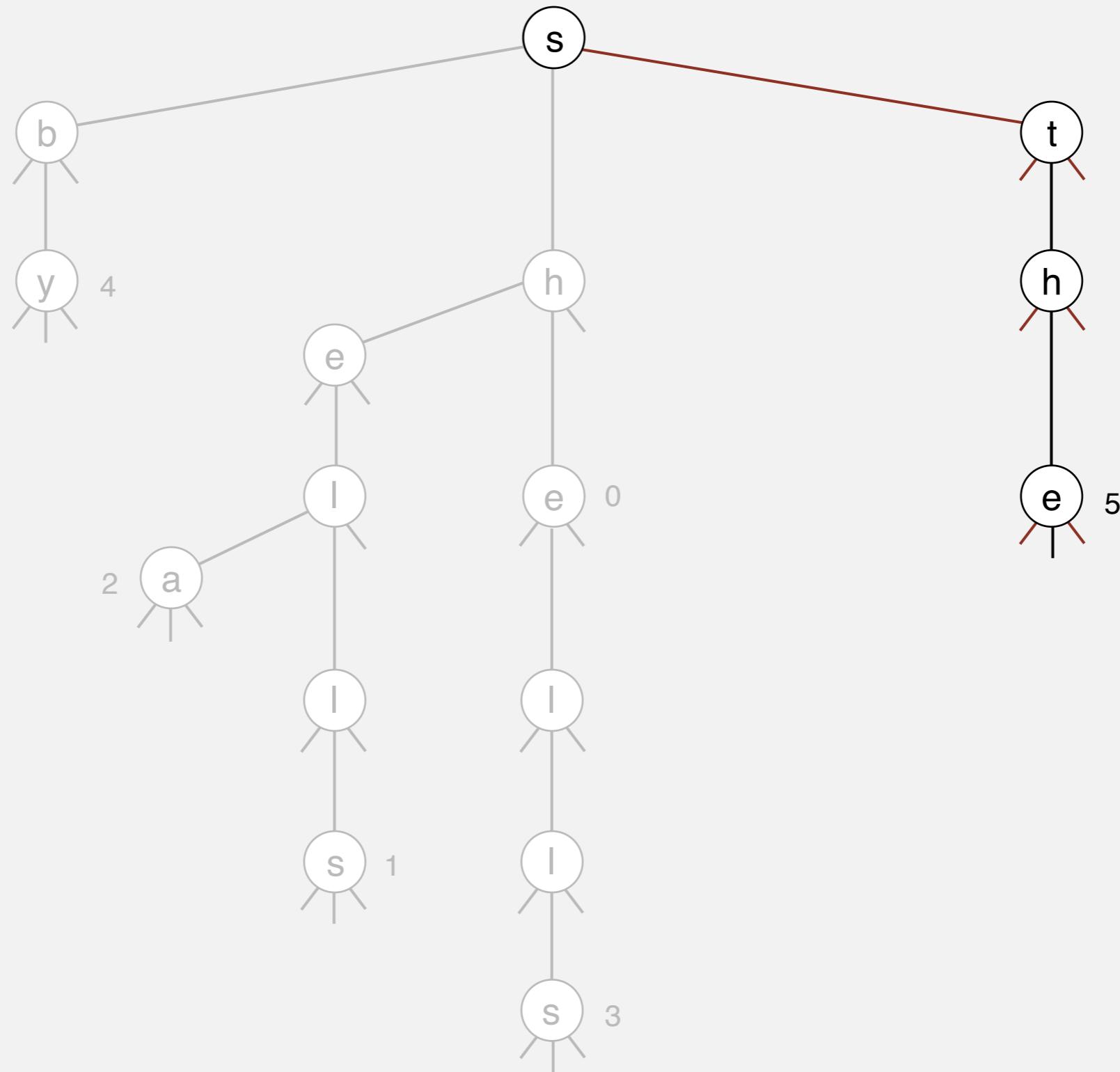
Ternary search trie construction demo

put("the", 5)



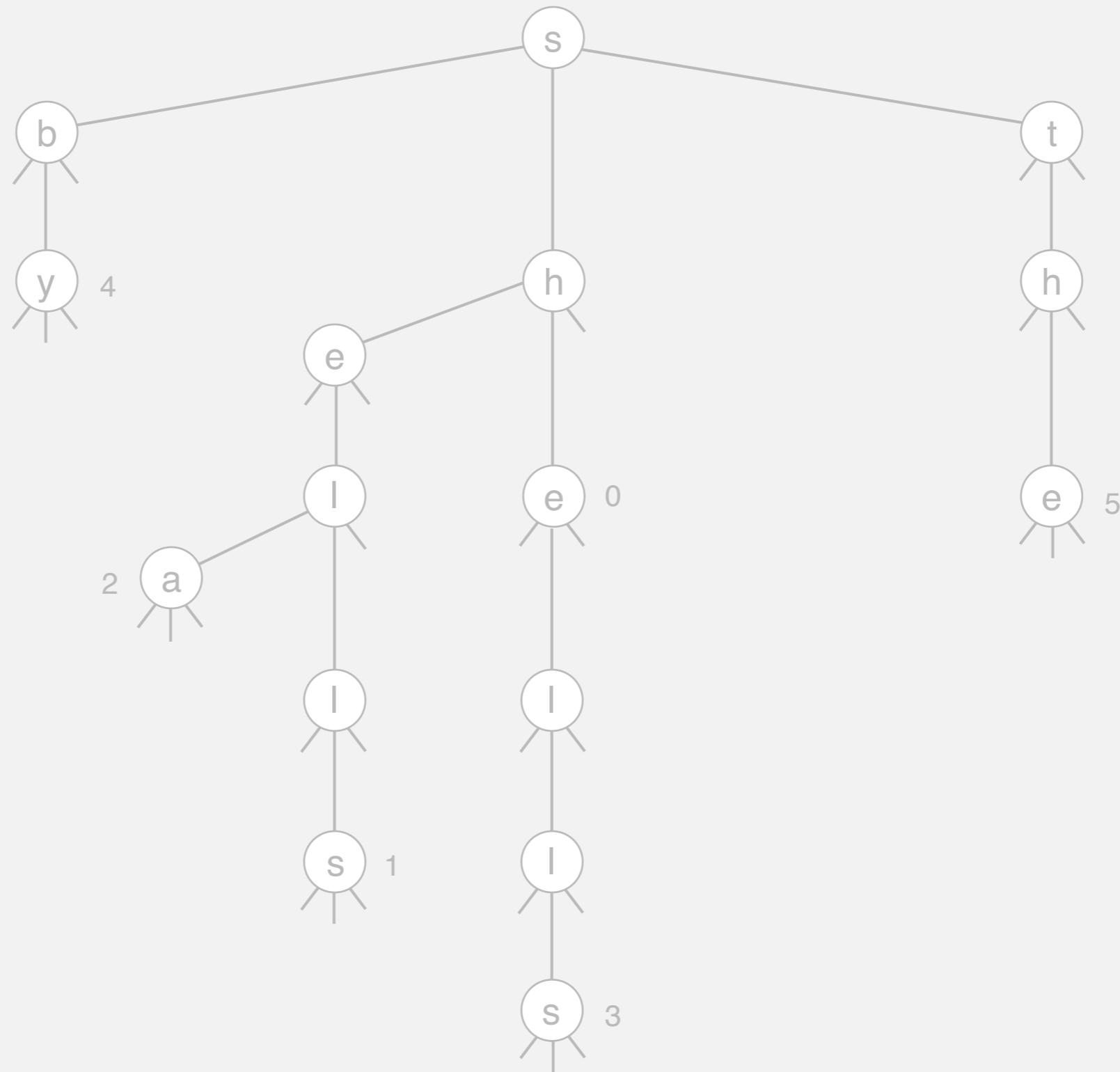
Ternary search trie construction demo

put("the", 5)



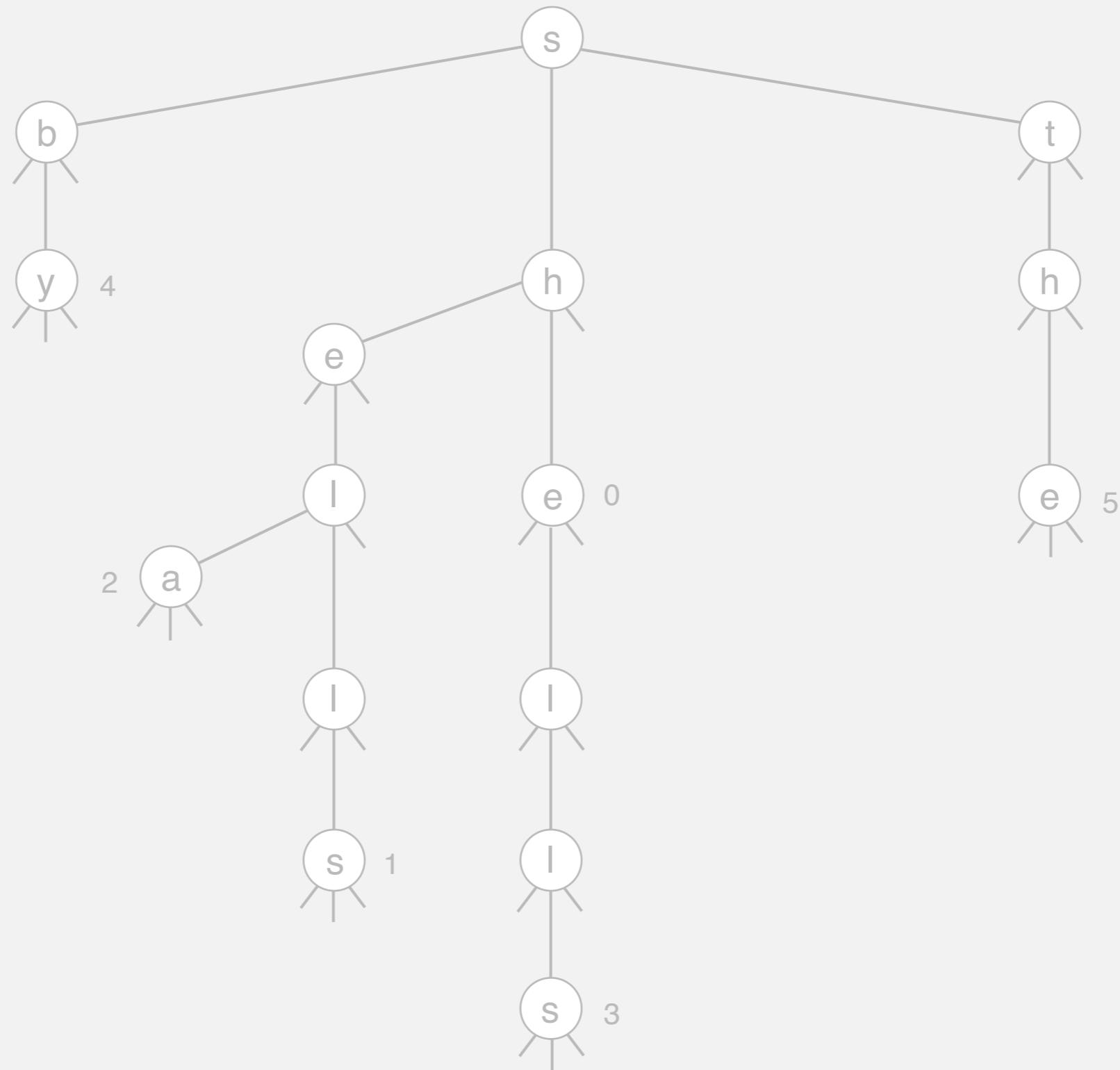
Ternary search trie construction demo

ternary search trie



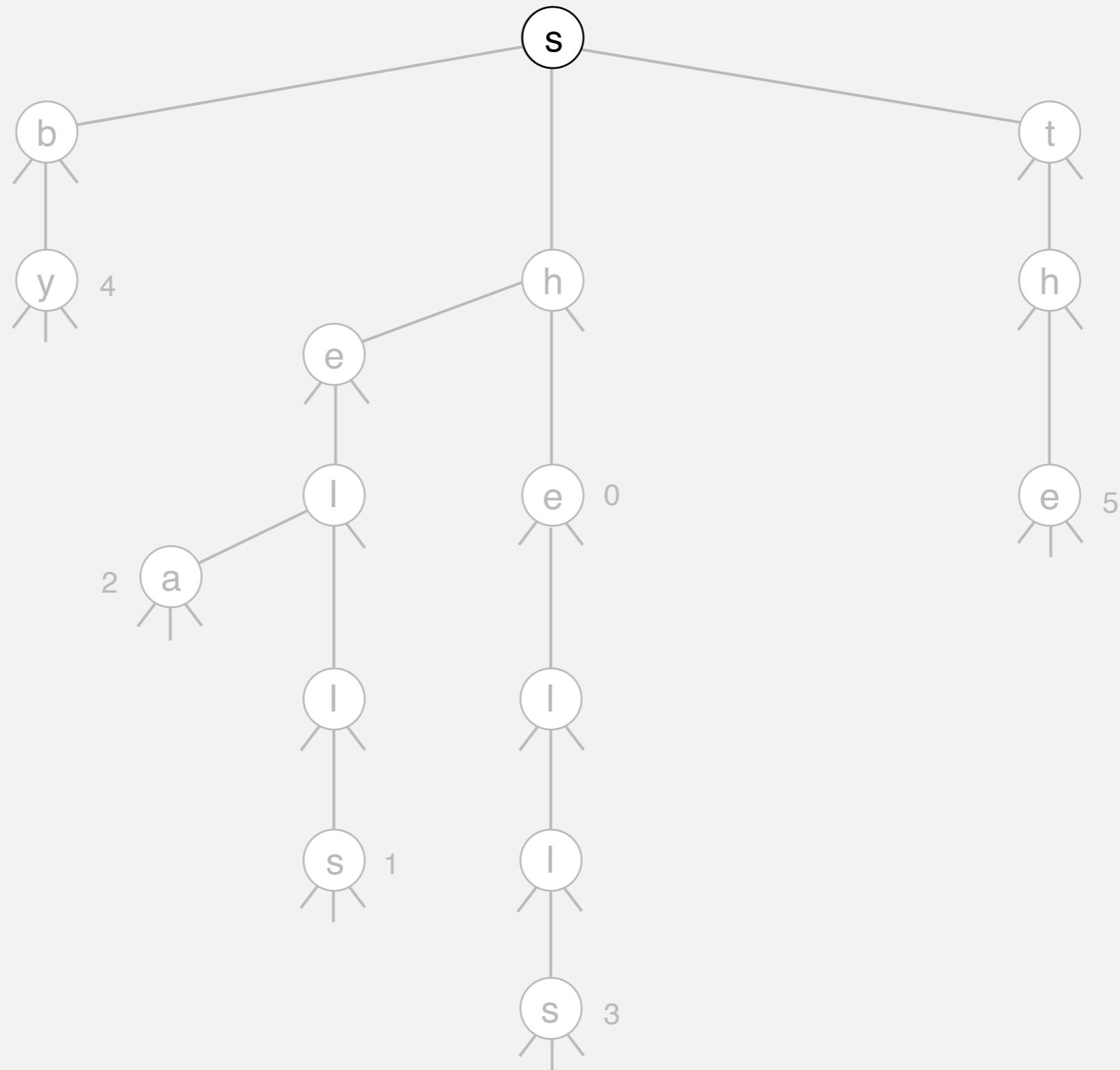
Ternary search trie construction demo

put("sea", 6)



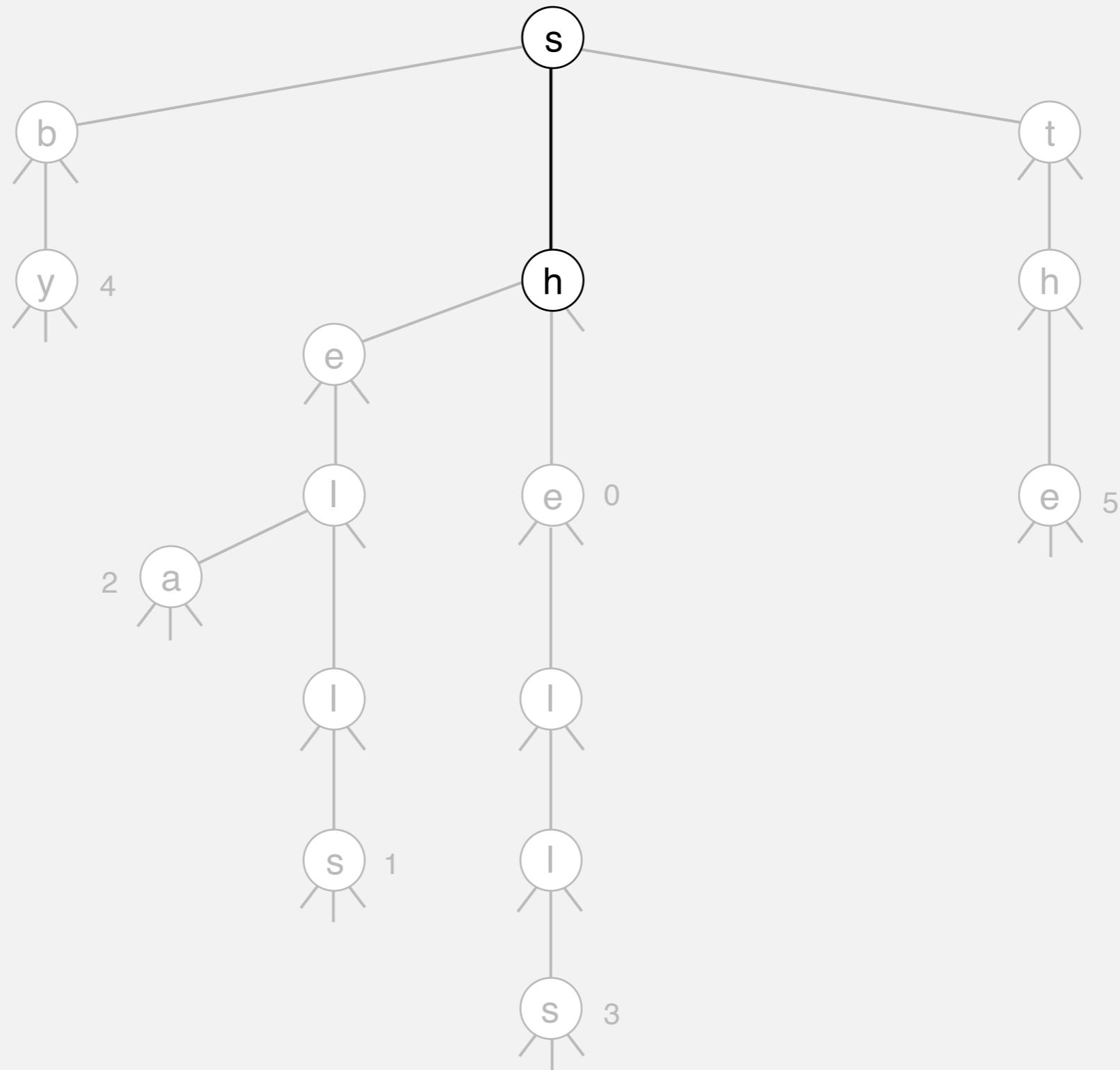
Ternary search trie construction demo

put("sea", 6)



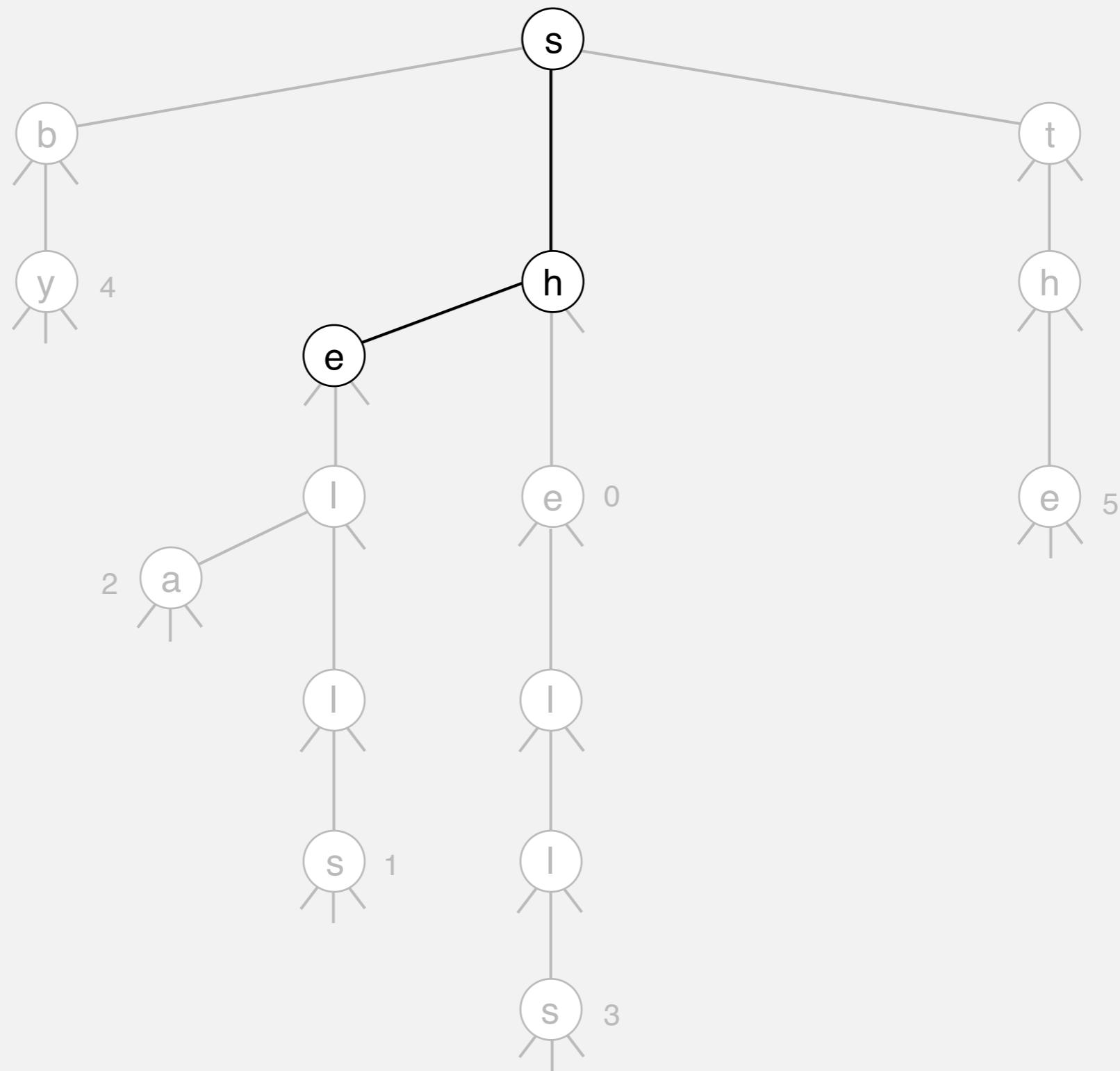
Ternary search trie construction demo

put("sea", 6)



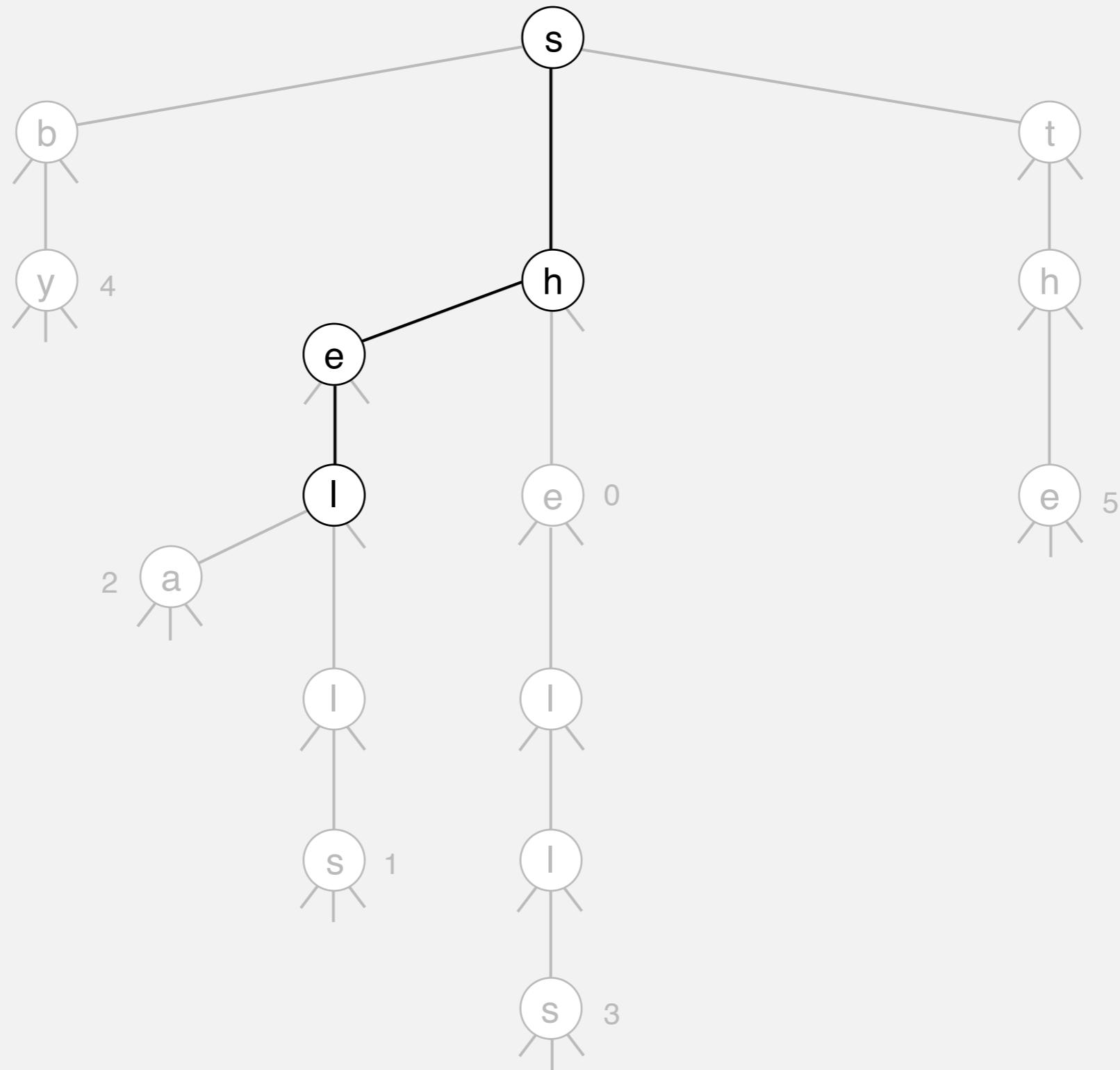
Ternary search trie construction demo

`put("sea", 6)`



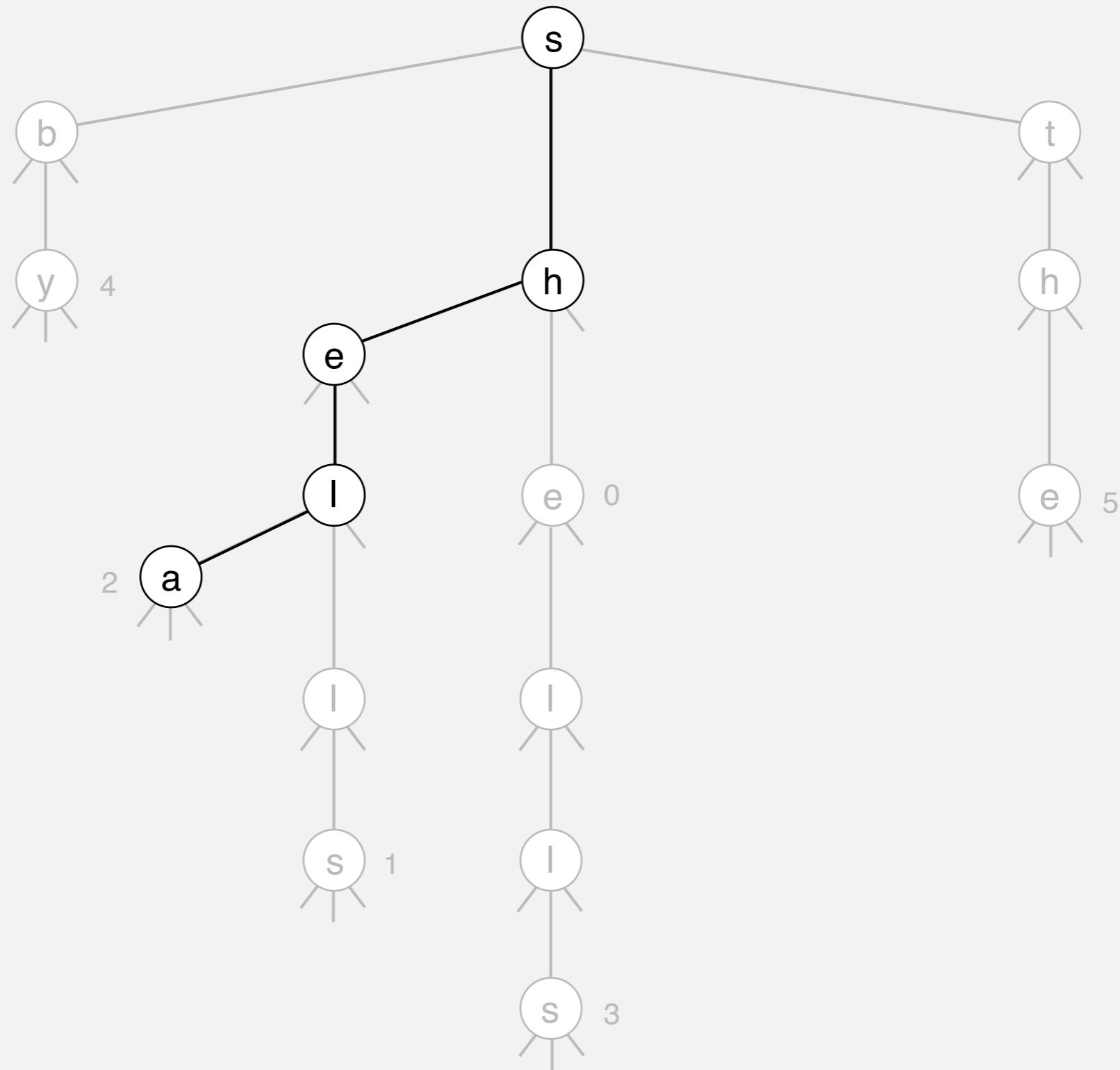
Ternary search trie construction demo

`put("sea", 6)`



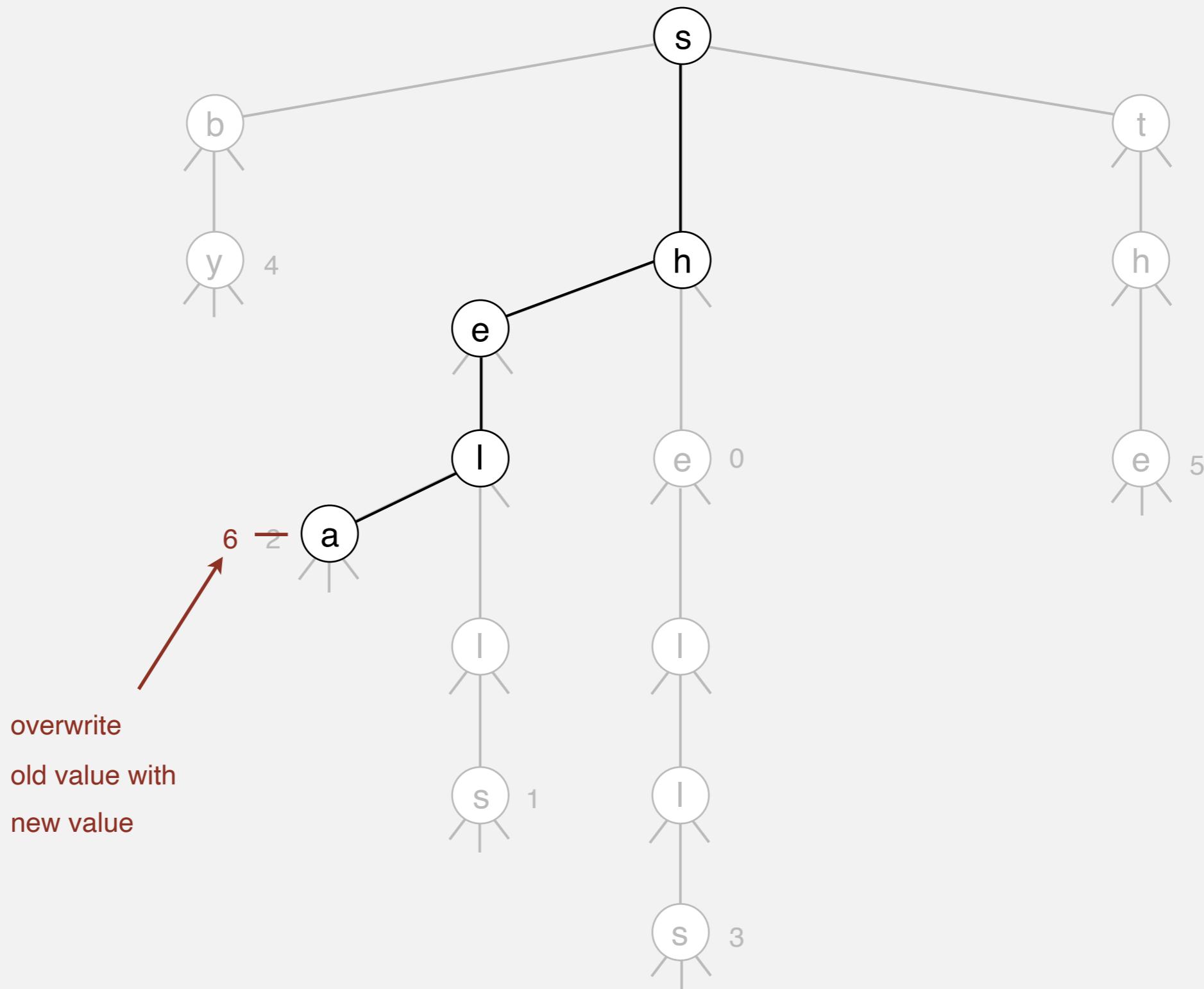
Ternary search trie construction demo

`put("sea", 6)`



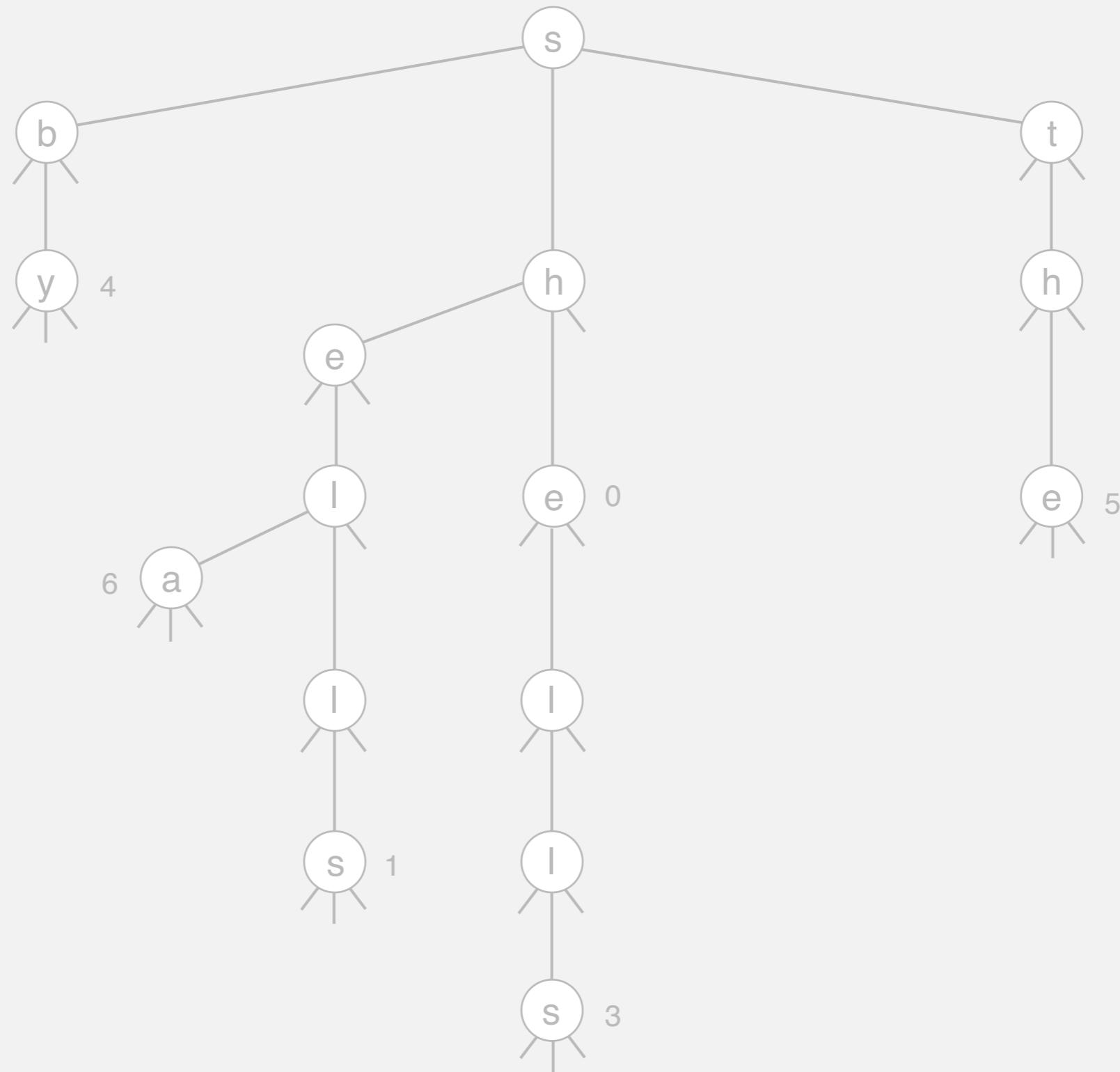
Ternary search trie construction demo

`put("sea", 6)`



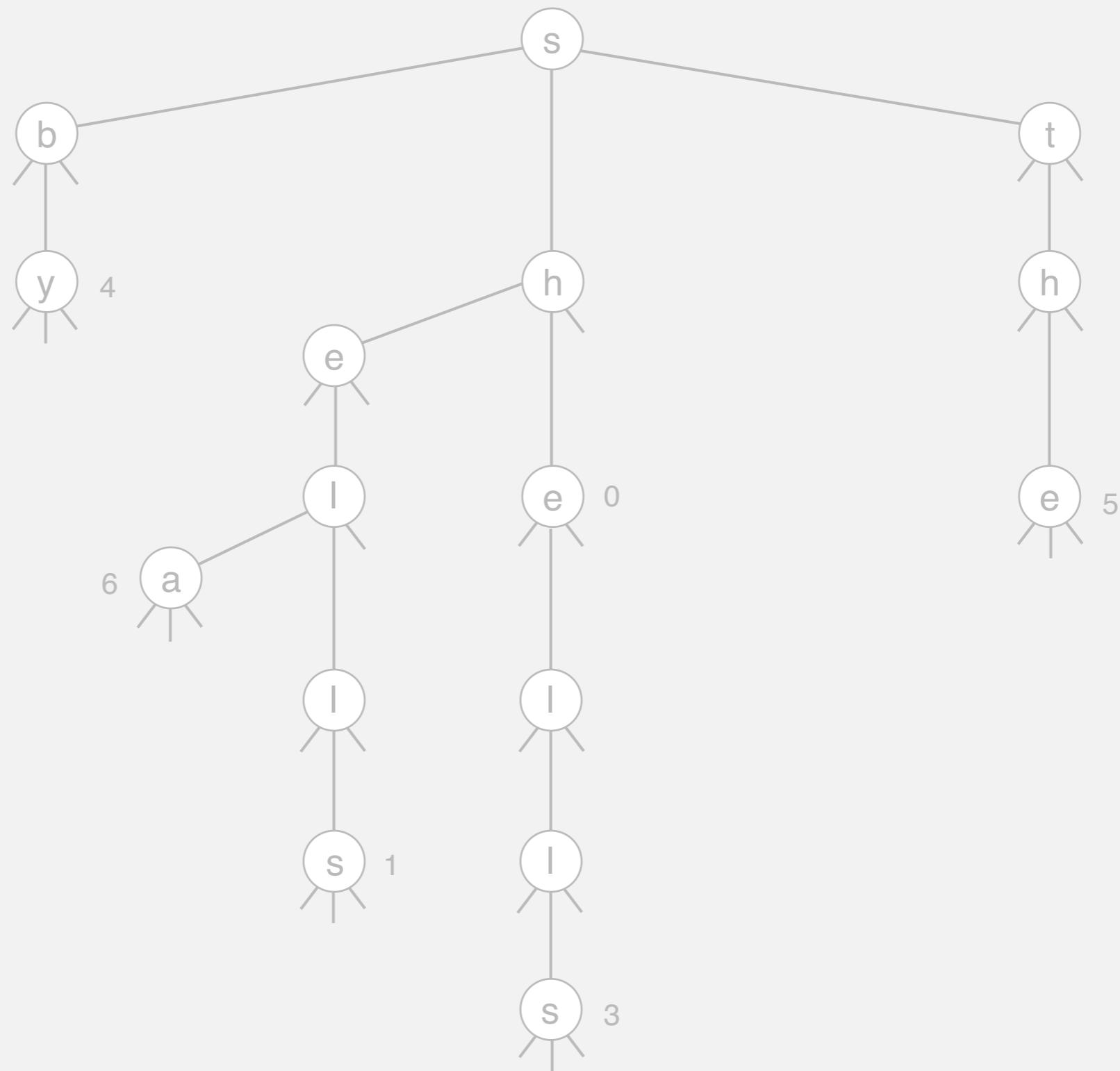
Ternary search trie construction demo

ternary search trie



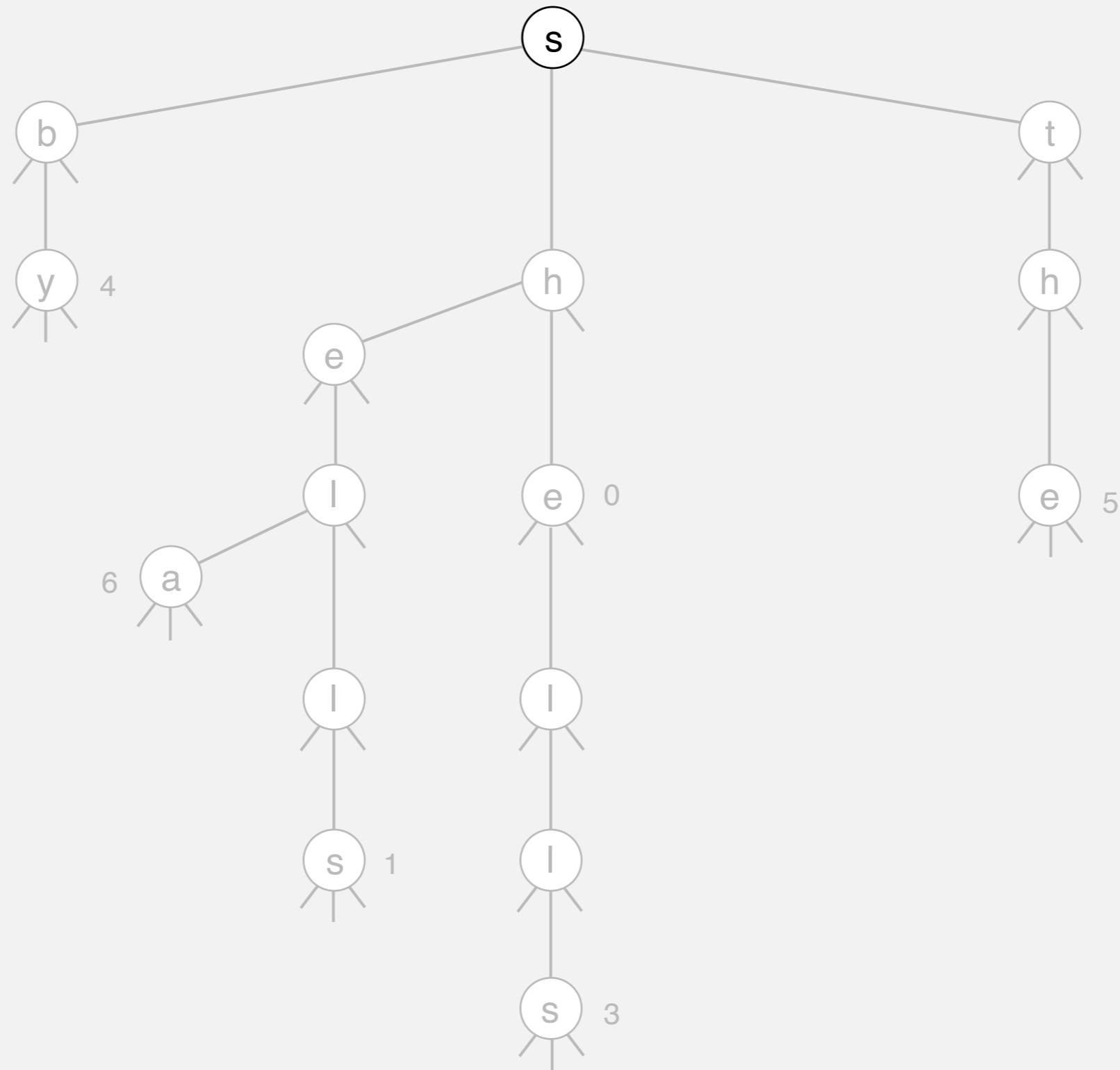
Ternary search trie construction demo

put("shore", 7)



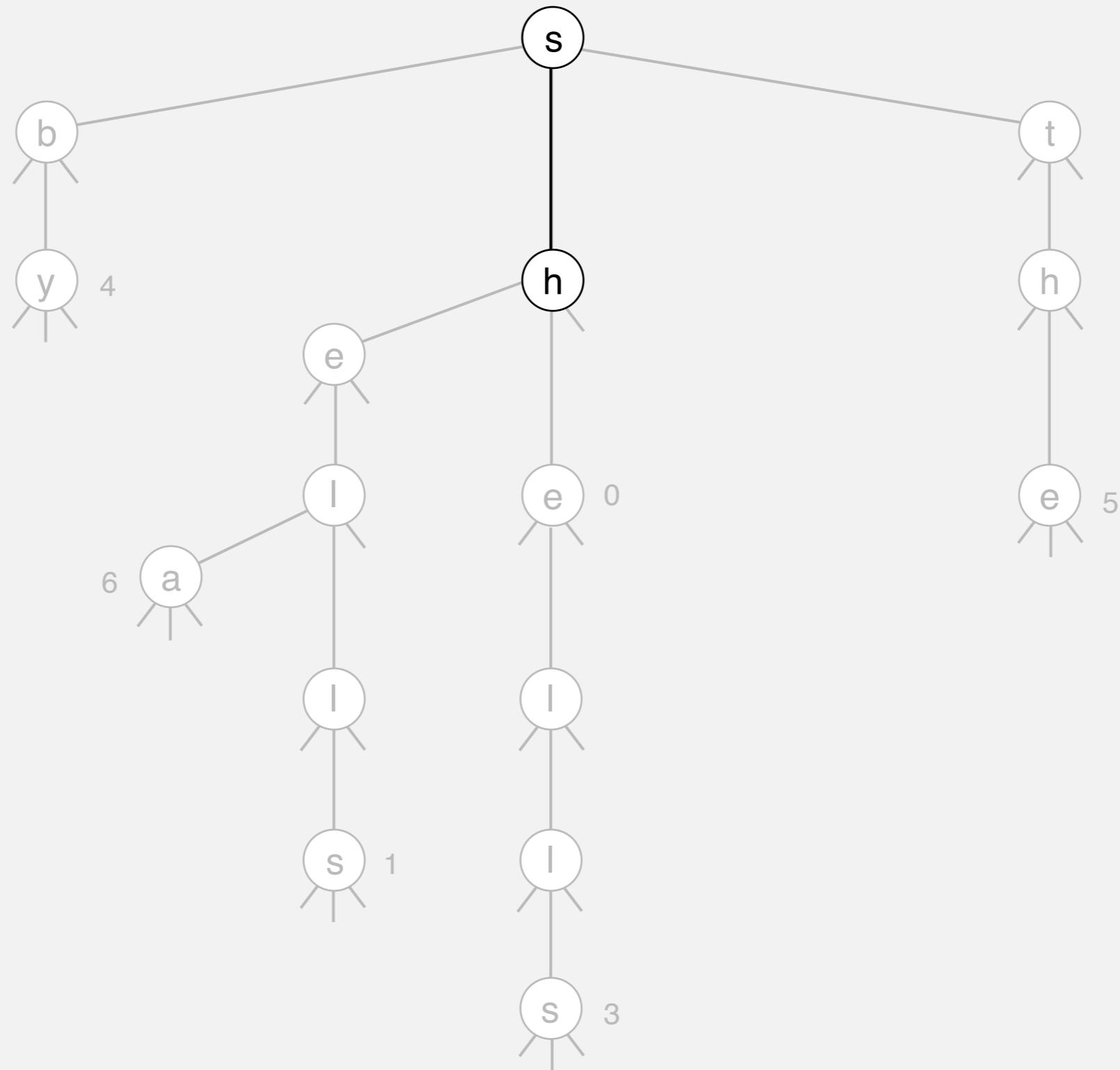
Ternary search trie construction demo

put("shore", 7)



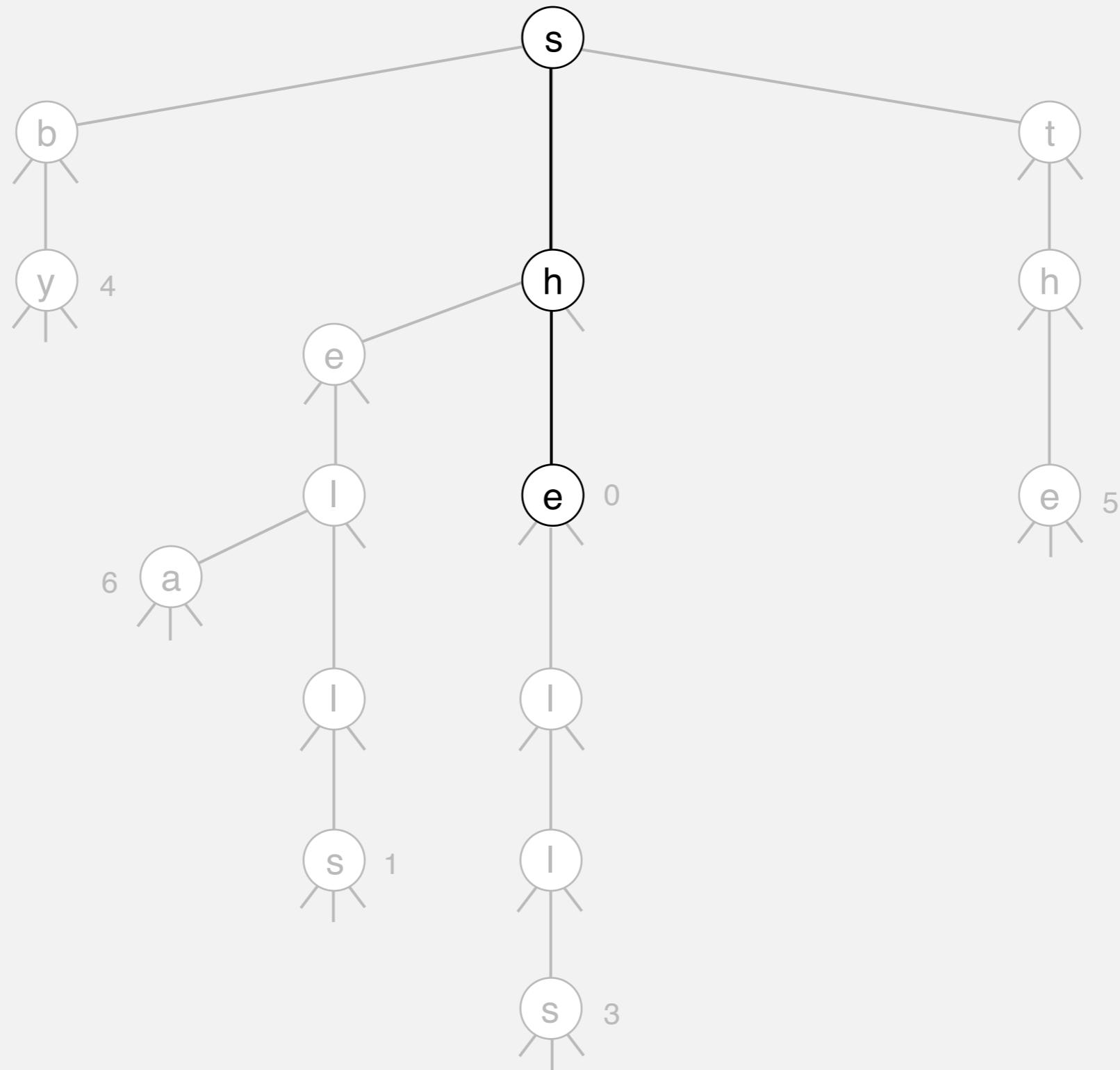
Ternary search trie construction demo

put("shore", 7)



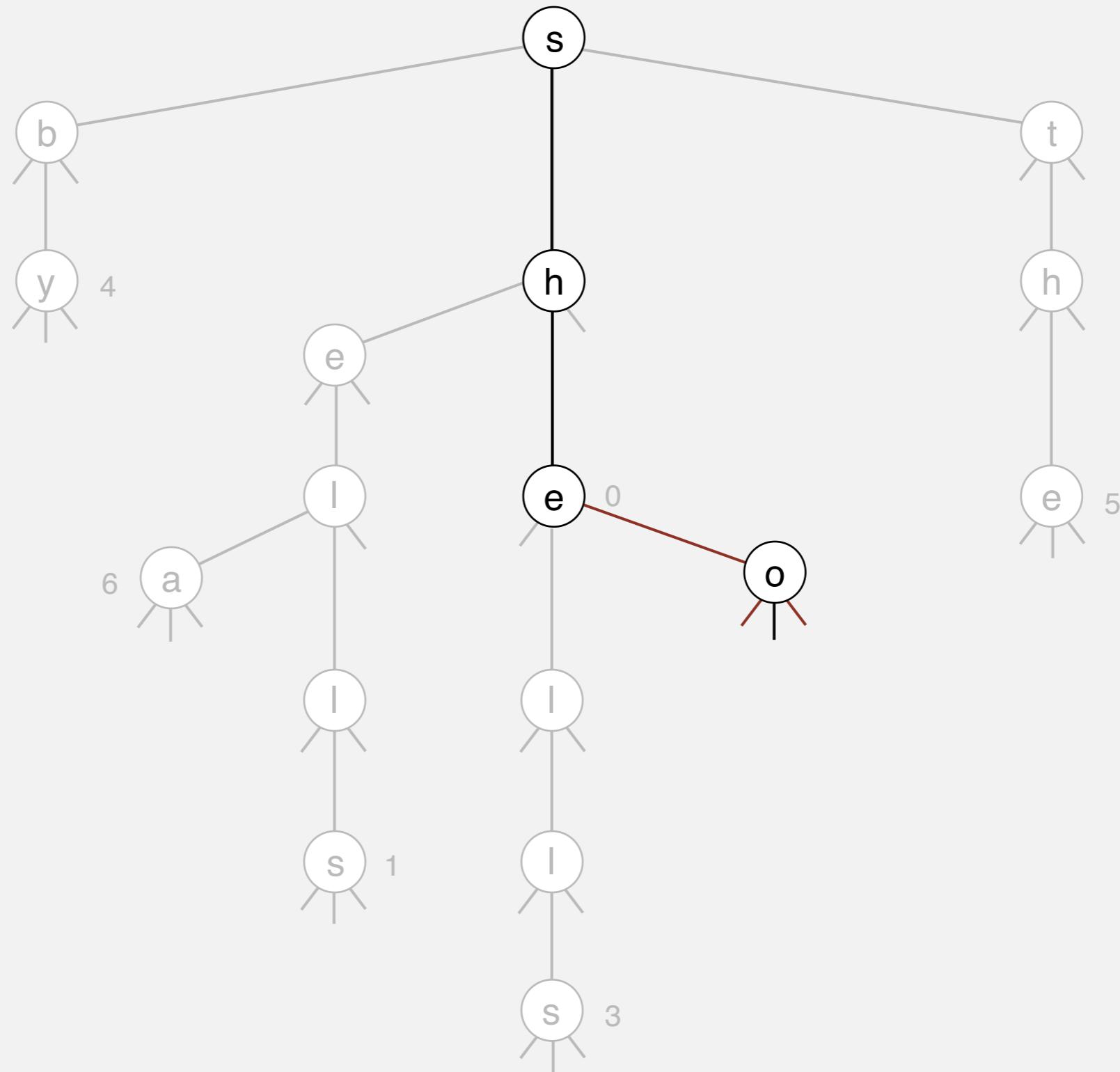
Ternary search trie construction demo

put("shore", 7)



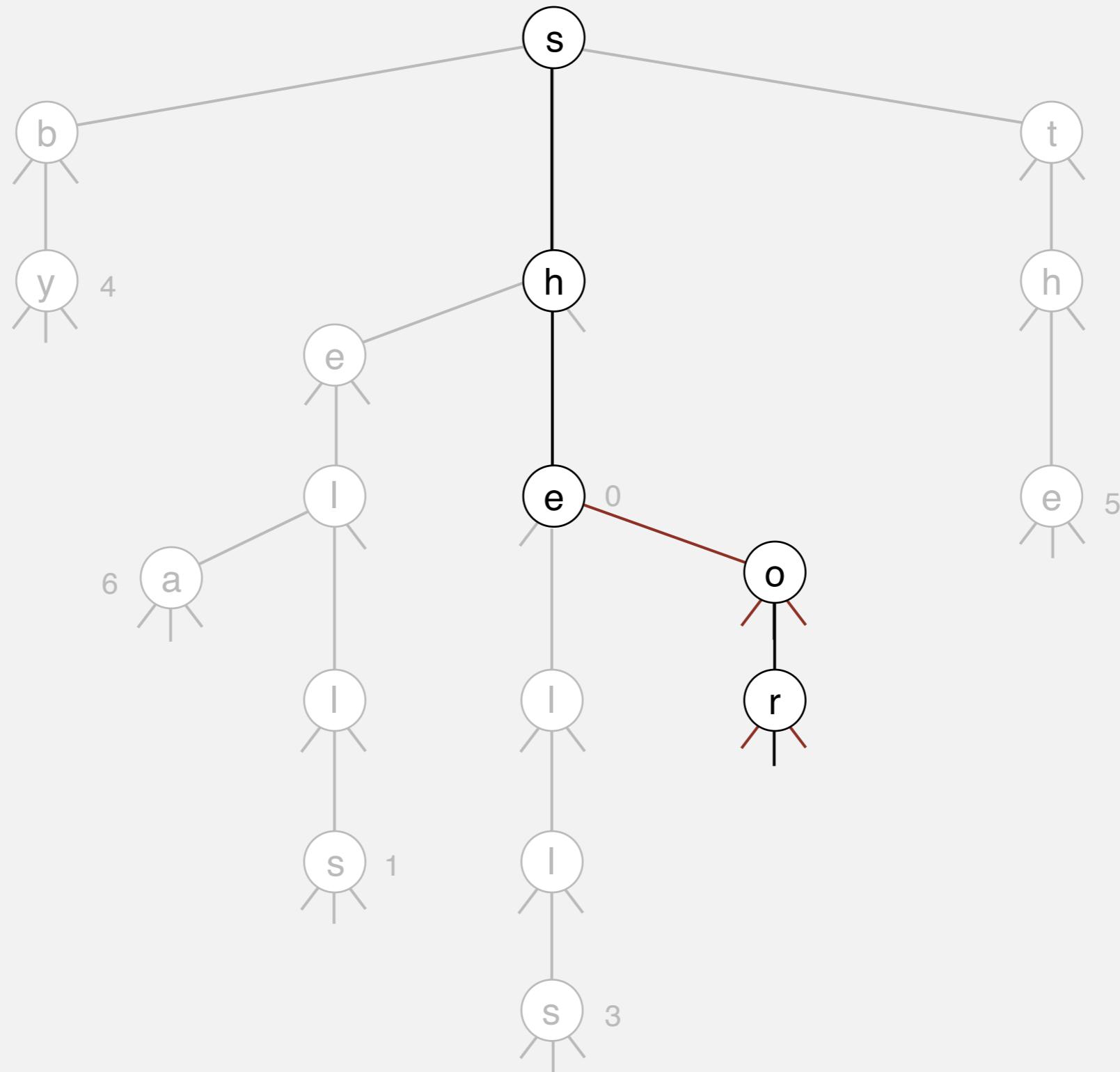
Ternary search trie construction demo

`put("shore", 7)`



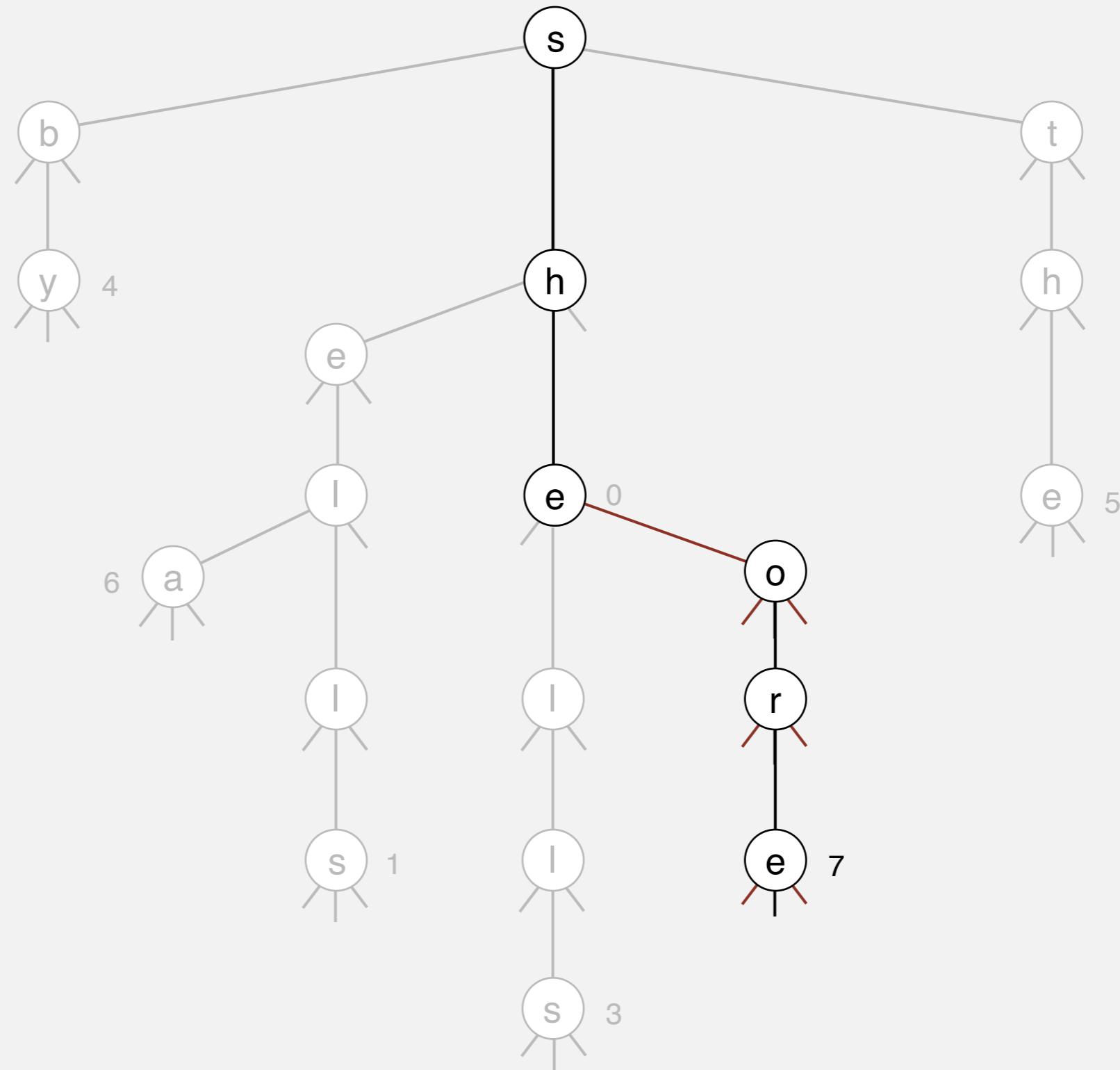
Ternary search trie construction demo

`put("shore", 7)`



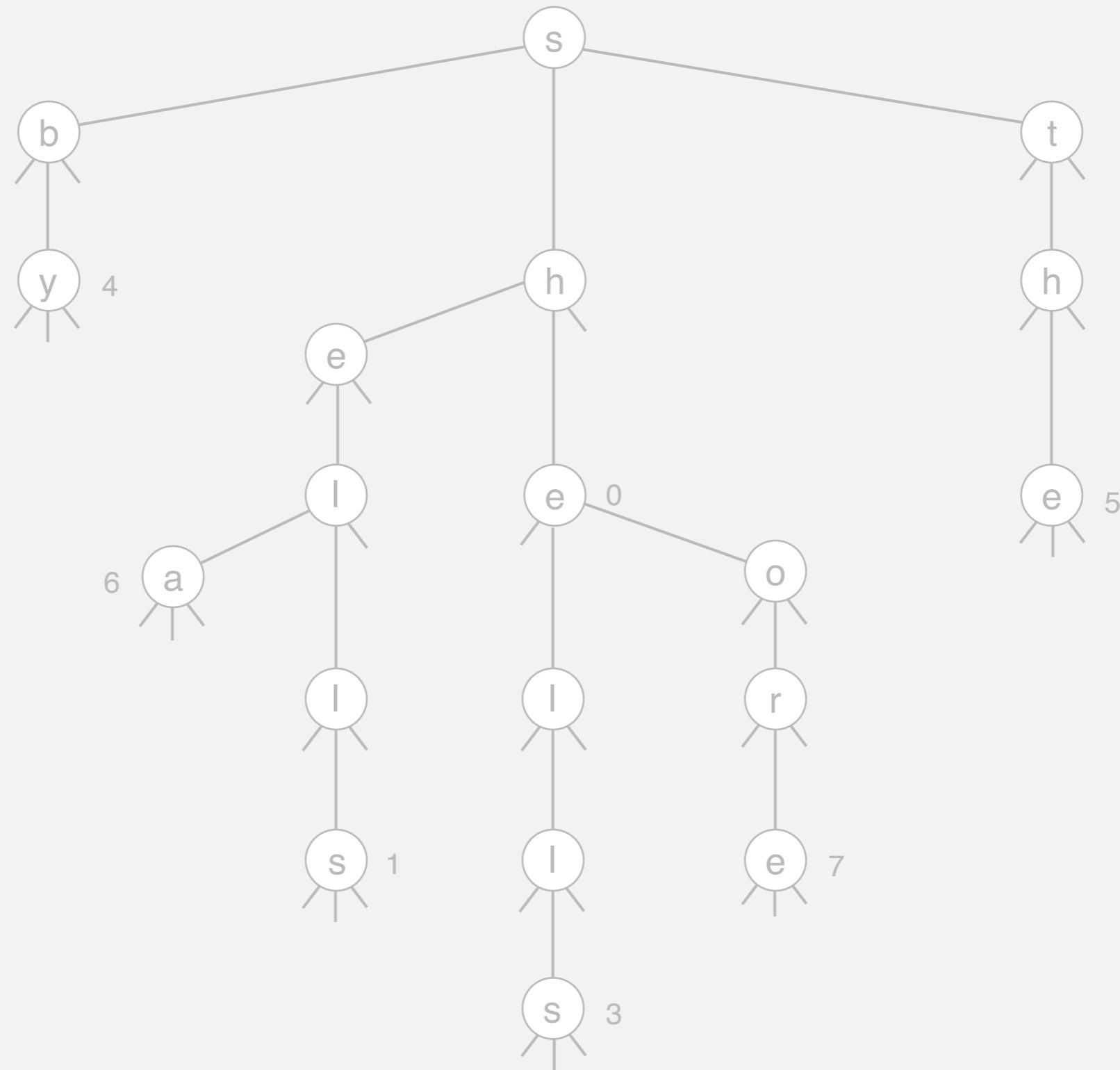
Ternary search trie construction demo

`put("shore", 7)`



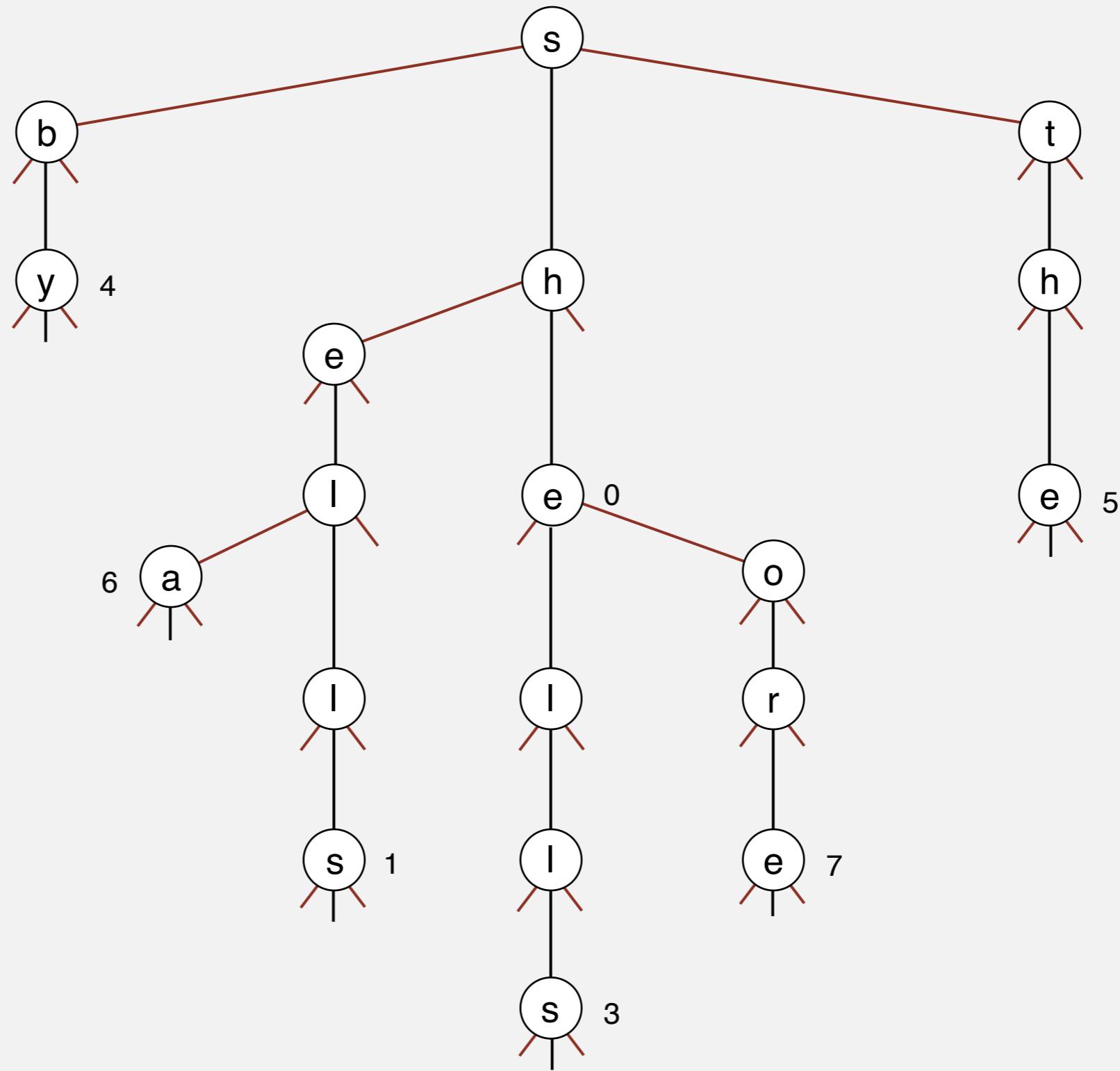
Ternary search trie construction demo

ternary search trie



Ternary search trie construction demo

ternary search trie



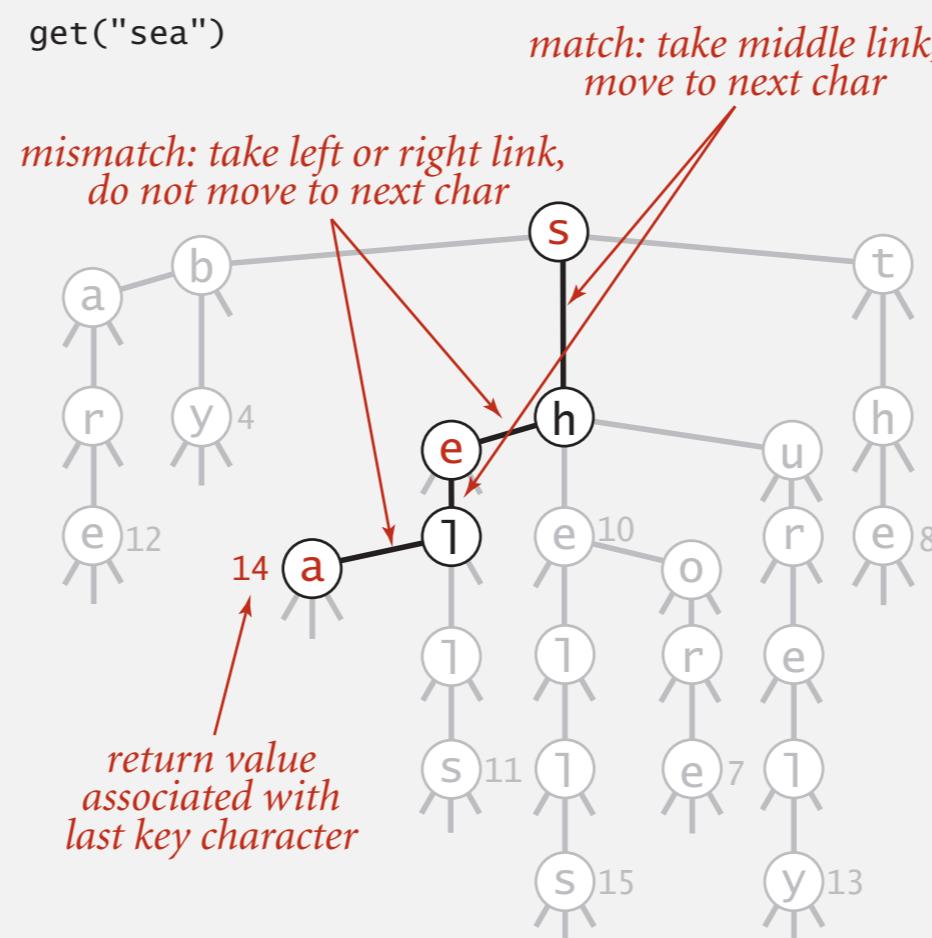
Search in a TST

Follow links corresponding to each character in the key.

- If less, take left link; if greater, take right link.
- If equal, take the middle link and move to the next key character.

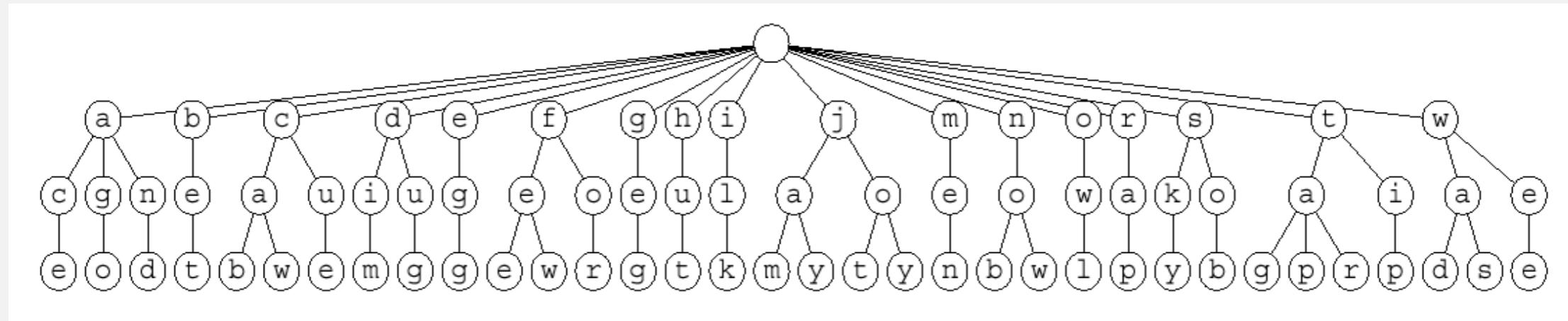
Search hit. Node where search ends has a non-null value.

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



26-way trie vs. TST

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

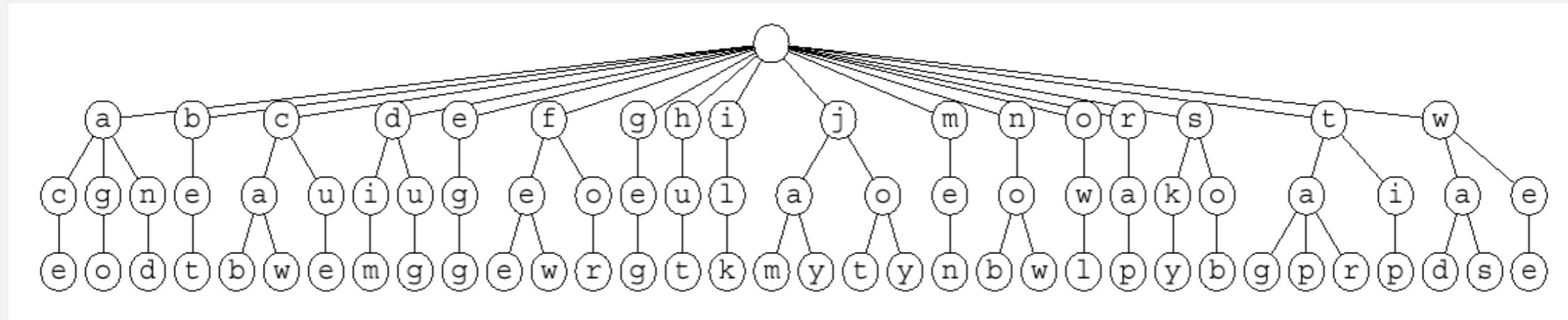


26-way trie (1035 null links, not shown)

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

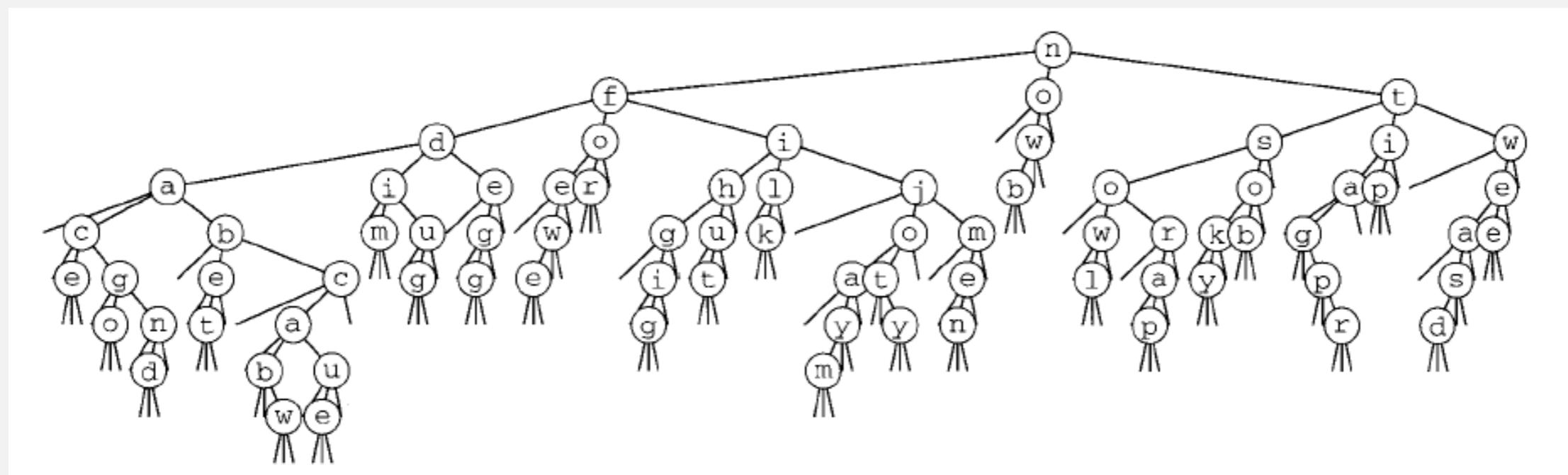
26-way trie vs. TST

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)

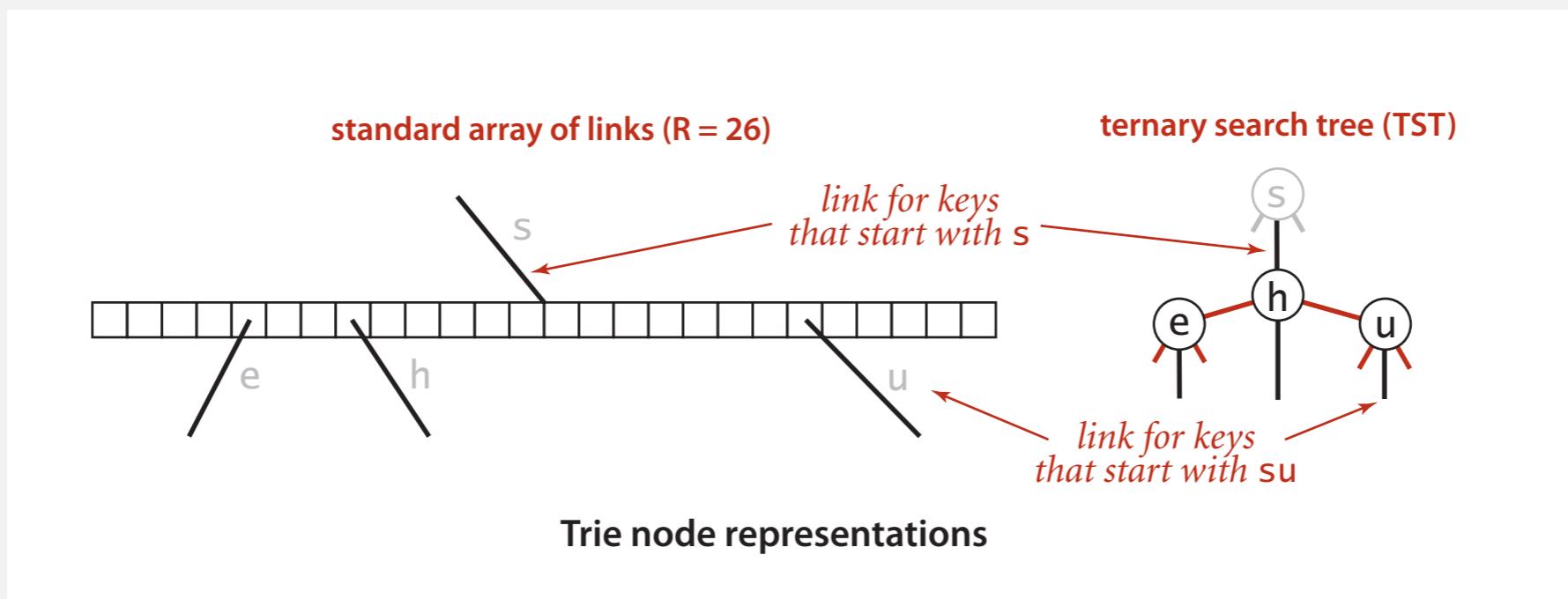
now
for
tip
ilk
dim
tag
jot
sob
nob
sky
hut
ace
bet
men
egg
few
jay
owl
joy
rap
gig
wee
was
cab
wad
caw
cue
fee
tap
ago

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

implementation	character accesses (typical case)				dedup	
	search hit	search miss	insert	space (references)	moby.txt	actors.txt
red-black BST	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	$4N$	1.4	97.4
hashing (linear probing)	L	L	L	$4N \text{ to } 16N$	0.76	40.6
R-way trie	L	$\log_R N$	L	$(R+1) N$	1.12	<i>out of memory</i>
TST	$L + \ln N$	$\ln N$	$L + \ln N$	4 N	0.72	38.7

String symbol table implementation cost summary

implementation	character accesses (typical case)				dedup	
	search hit	search miss	insert	space (references)	moby.txt	actors.txt
red-black BST	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	$4N$	1.4	97.4
hashing (linear probing)	L	L	L	$4N \text{ to } 16N$	0.76	40.6
R-way trie	L	$\log_R N$	L	$(R+1) N$	1.12	<i>out of memory</i>
TST	$L + \ln N$	$\ln N$	$L + \ln N$	4 N	0.72	38.7

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

String symbol table implementation cost summary

implementation	character accesses (typical case)				dedup	
	search hit	search miss	insert	space (references)	moby.txt	actors.txt
red-black BST	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	$4N$	1.4	97.4
hashing (linear probing)	L	L	L	$4N \text{ to } 16N$	0.76	40.6
R-way trie	L	$\log_R N$	L	$(R+1) N$	1.12	<i>out of memory</i>
TST	$L + \ln N$	$\ln N$	$L + \ln N$	4 N	0.72	38.7

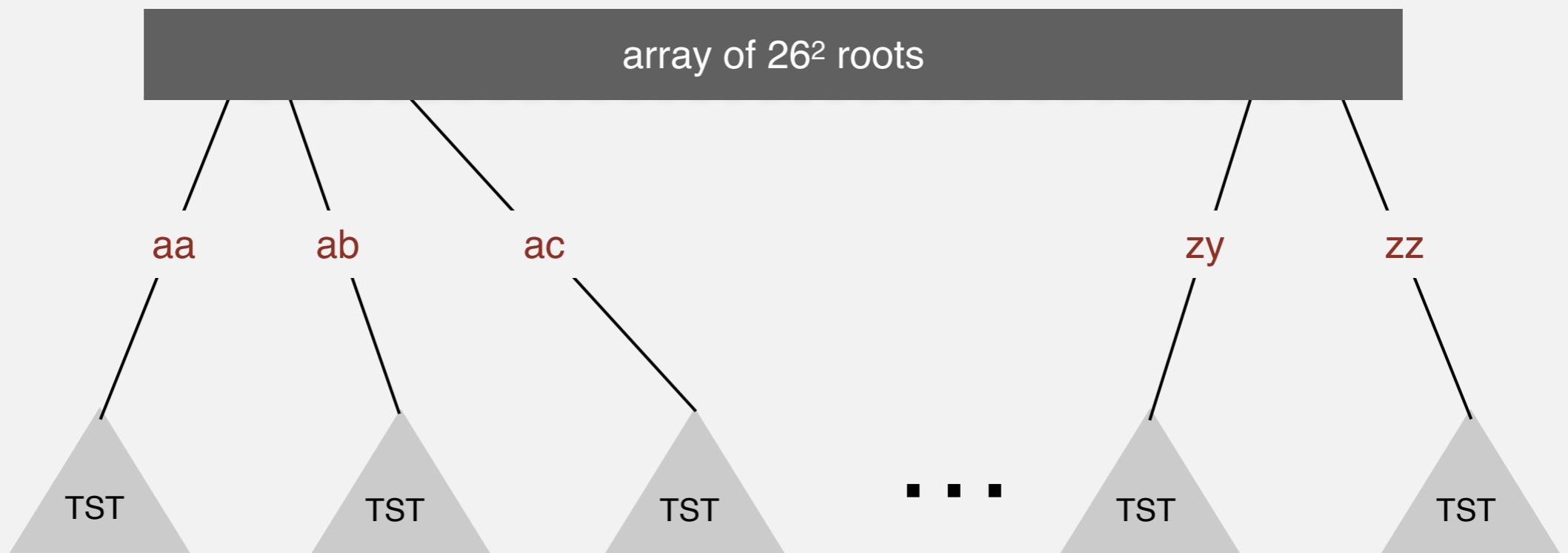
Remark. Can build balanced TSTs via rotations to achieve $L + \log N$ worst-case guarantees.

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

TST with R^2 branching at root

Hybrid of R-way trie and TST.

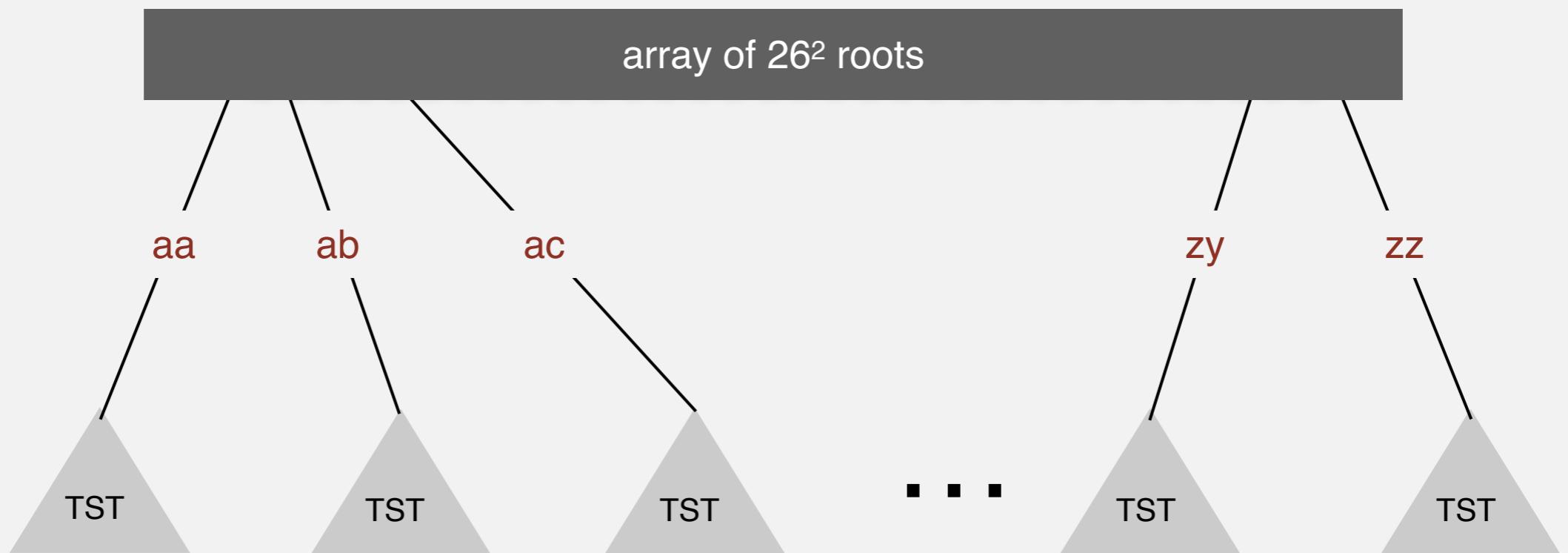
- Do R^2 -way branching at root.
- Each of R^2 root nodes points to a TST.



TST with R^2 branching at root

Hybrid of R-way trie and TST.

- Do R^2 -way branching at root.
- Each of R^2 root nodes points to a TST.



Q. What about one- and two-letter words?

String symbol table implementation cost summary

implementation	character accesses (typical case)				dedup	
	search hit	search miss	insert	space (references)	moby.txt	actors.txt
red-black BST	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	$4N$	1.4	97.4
hashing (linear probing)	L	L	L	$4N \text{ to } 16N$	0.76	40.6
R-way trie	L	$\log_R N$	L	$(R+1) N$	1.12	<i>out of memory</i>
TST	$L + \ln N$	$\ln N$	$L + \ln N$	$4 N$	0.72	38.7
TST with R²	$L + \ln N$	$\ln N$	$L + \ln N$	$4 N + R^2$	0.51	32.7

Bottom line. Faster than hashing for our benchmark client.

TST vs. hashing

Hashing.

- Need to examine entire key.
- Search hits and misses cost about the same.
- Performance relies on hash function.
- Does not support ordered symbol table operations.

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!).

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]

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

5.2 TRIES

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

String symbol table API

[Character-based operations.](#) The string symbol table API supports several useful character-based operations.

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

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.

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.

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

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

```
    :
```

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

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.

keys()

key q

b by

s se

sea by sea

sel

sell

sells by sea sells

sh

she

shell

shells

sho

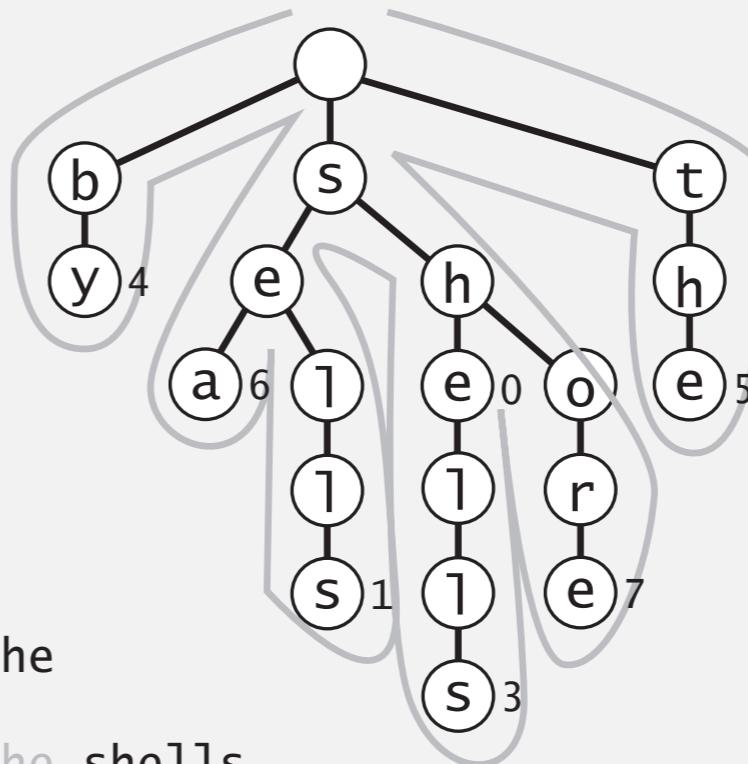
shor

shore

t

th

the



by sea sells she shells

shore

the

Prefix matches

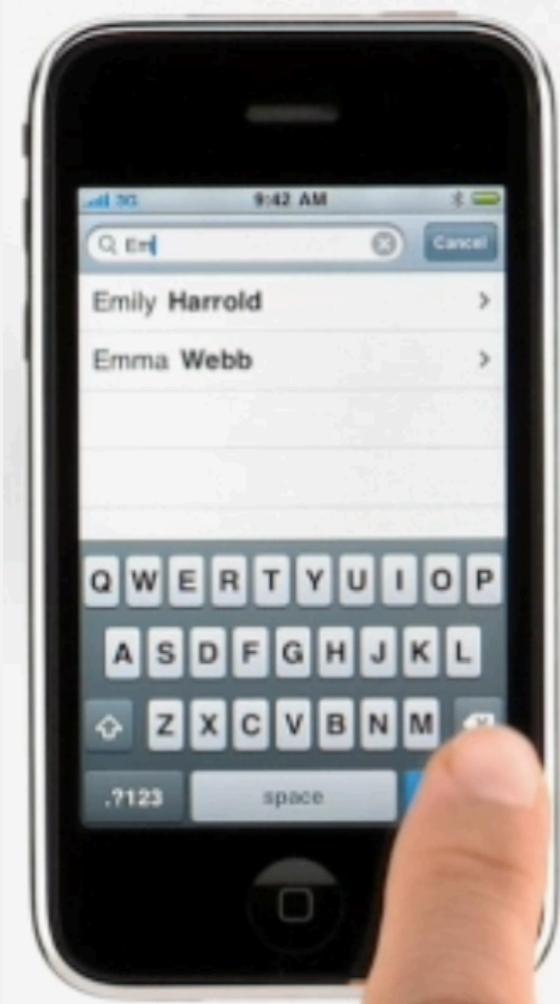
Find all keys in a symbol table starting with a given prefix.

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.

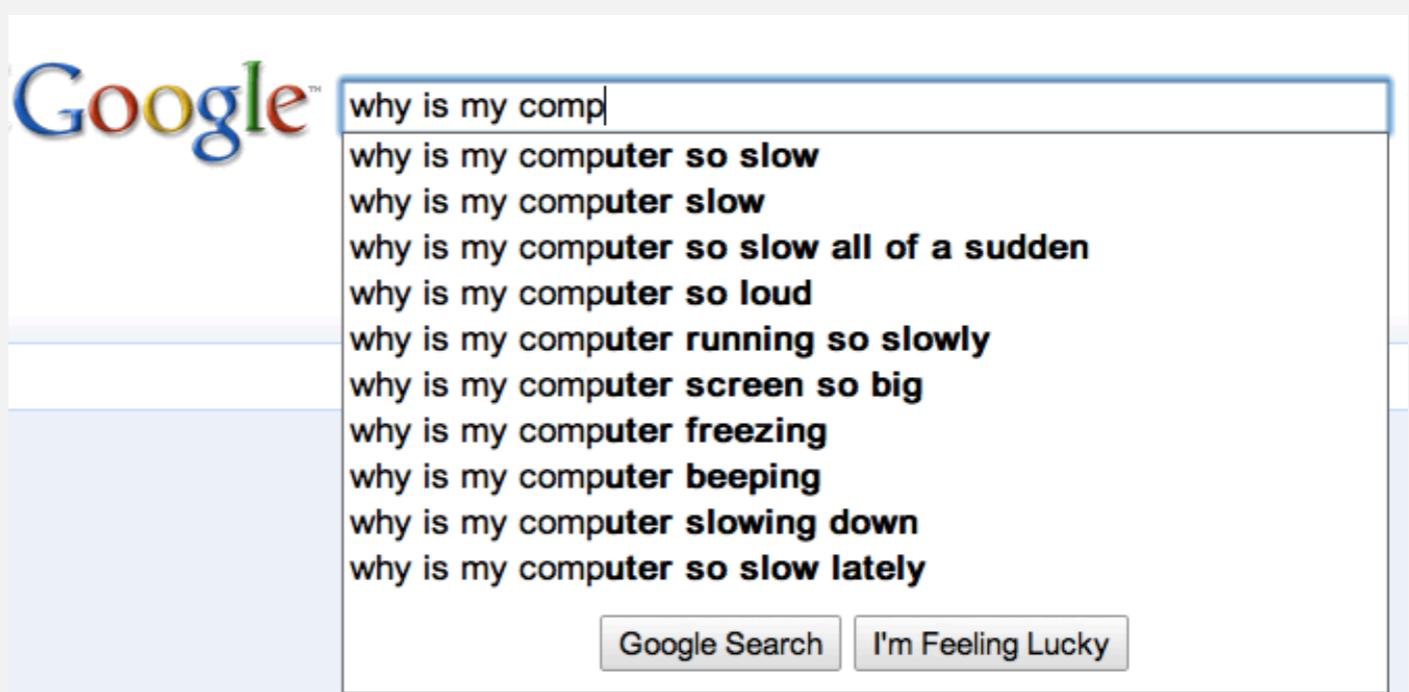
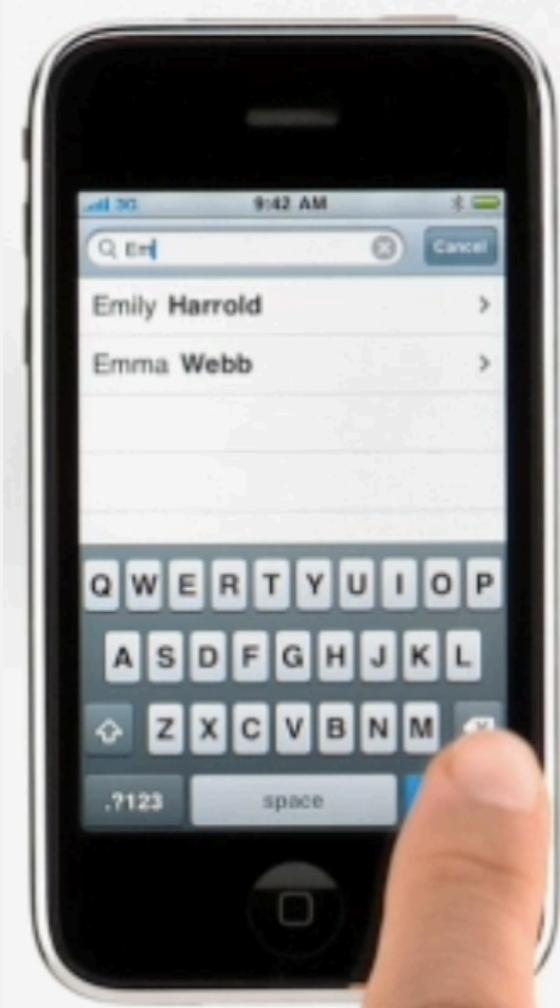


Prefix matches

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

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

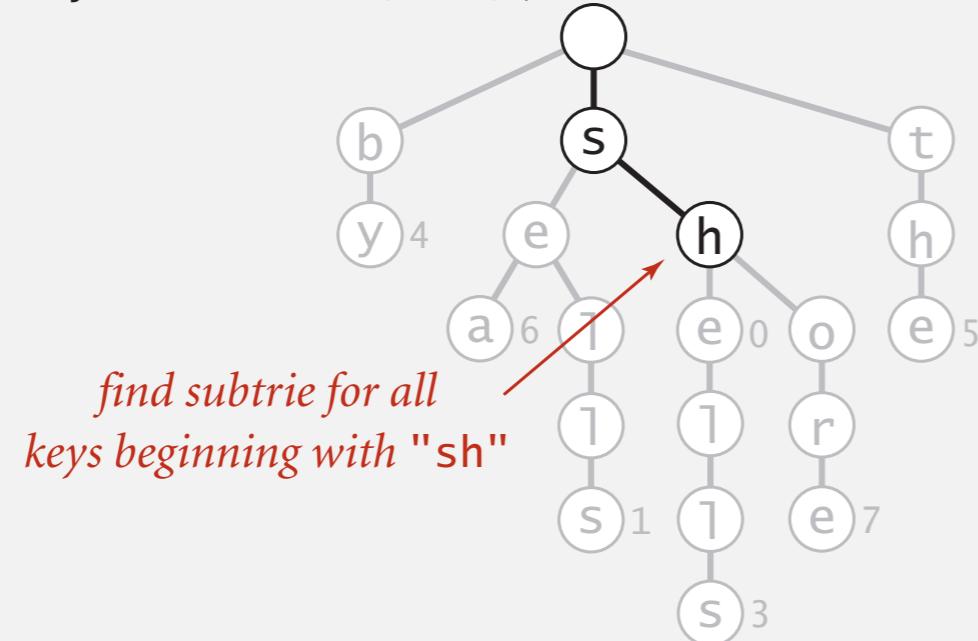
- User types characters one at a time.
- System reports all matching strings.



Prefix matches in an R-way trie

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

keysWithPrefix("sh");

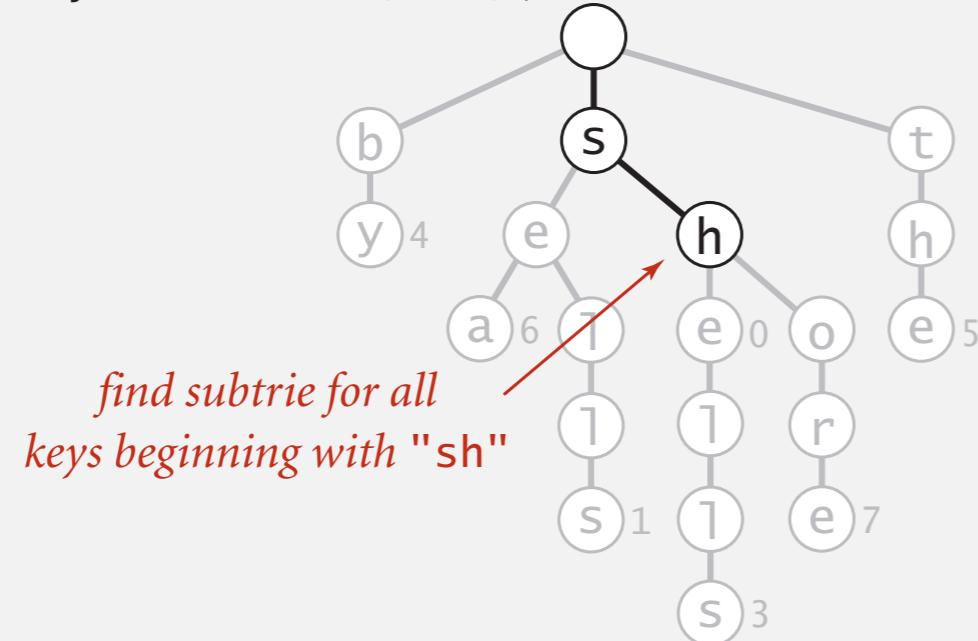


*find subtrie for all
keys beginning with "sh"*

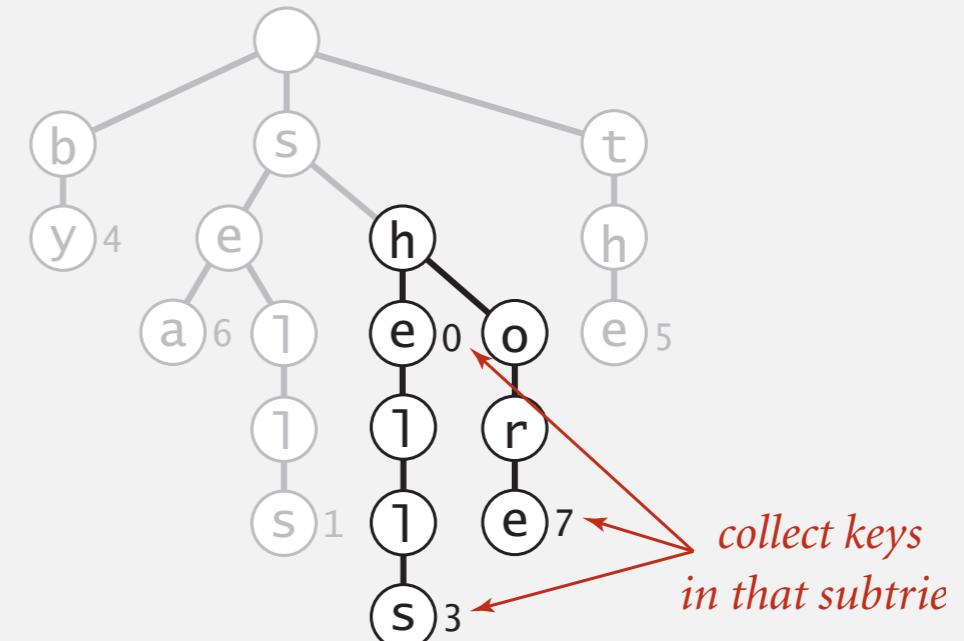
Prefix matches in an R-way trie

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

keysWithPrefix("sh");



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

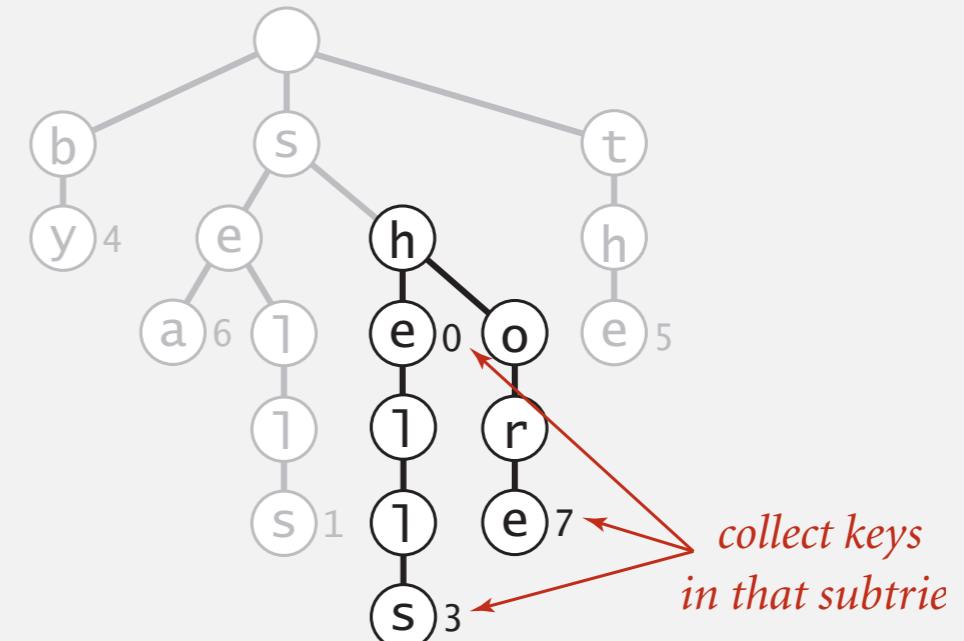
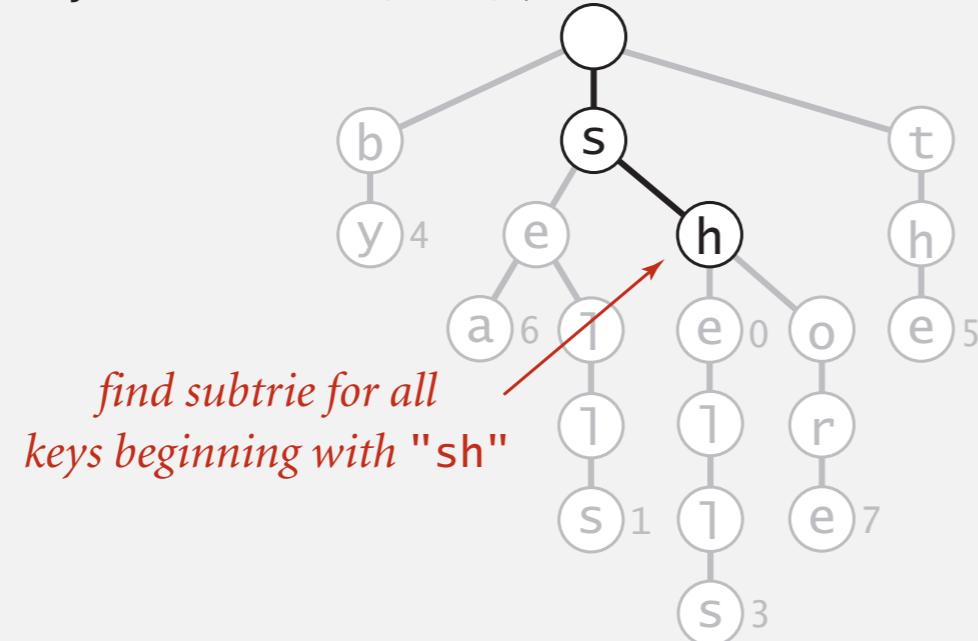


*collect keys
in that subtrie*

Prefix matches in an R-way trie

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

keysWithPrefix("sh");



```
public Iterable<String> keysWithPrefix(String prefix)
{
    Queue<String> queue = new Queue<String>();
    Node x = get(root, prefix, 0);
    collect(x, prefix, queue);
    return queue;
}
```

root of subtrie for all strings
beginning with given prefix

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

Wildcard matches

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

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

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

Longest prefix

Find longest key in symbol table that is a prefix of query string.

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)

Longest prefix

Find longest key in symbol table that is a prefix of query string.

Ex. To send packet toward destination IP address, router chooses IP address in routing table that is longest prefix match.

"128"	represented as 32-bit binary number for IPv4 (instead of string)
"128.112"	
"128.112.055"	
"128.112.055.15"	
"128.112.136"	<code>longestPrefixOf("128.112.136.11") = "128.112.136"</code>
"128.112.155.11"	<code>longestPrefixOf("128.112.100.16") = "128.112"</code>
"128.112.155.13"	<code>longestPrefixOf("128.166.123.45") = "128"</code>
"128.222"	
"128.222.136"	

Longest prefix

Find longest key in symbol table that is a prefix of query string.

Ex. To send packet toward destination IP address, router chooses IP address in routing table that is longest prefix match.

"128"	represented as 32-bit binary number for IPv4 (instead of string)
"128.112"	
"128.112.055"	
"128.112.055.15"	
"128.112.136"	<code>longestPrefixOf("128.112.136.11") = "128.112.136"</code>
"128.112.155.11"	<code>longestPrefixOf("128.112.100.16") = "128.112"</code>
"128.112.155.13"	<code>longestPrefixOf("128.166.123.45") = "128"</code>
"128.222"	
"128.222.136"	

Note. Not the same as floor:
 $\text{floor}("128.112.100.16") = "128.112.055.15"$

Longest prefix in an R-way trie

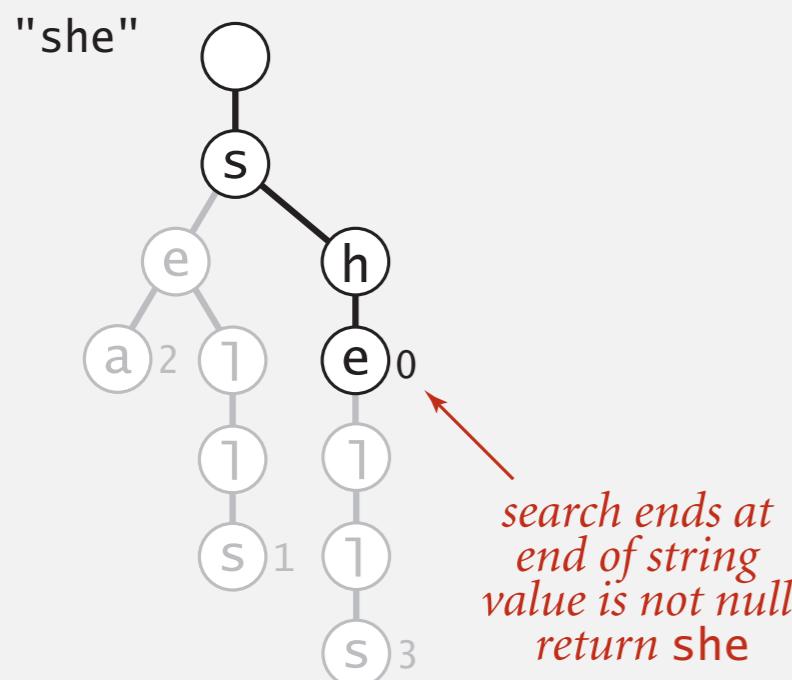
Find longest key in symbol table that is a prefix of query string.

- Search for query string.
- Keep track of longest key encountered.

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.

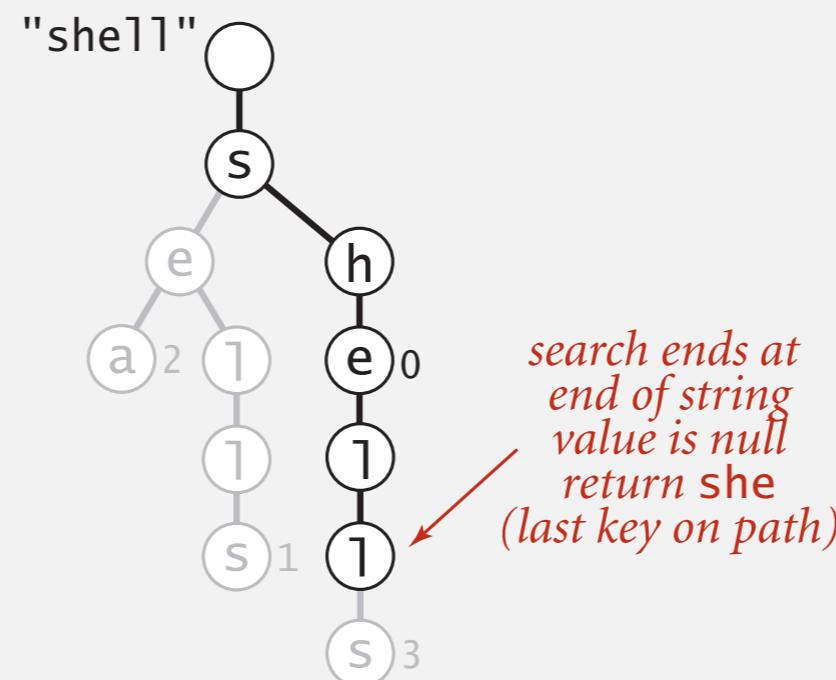
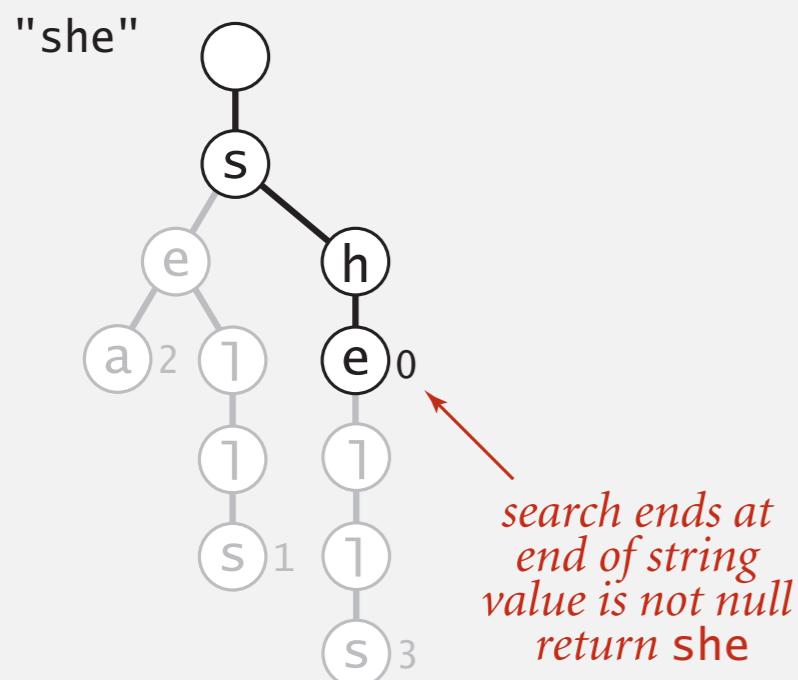


Possibilities for longestPrefixOf()

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.

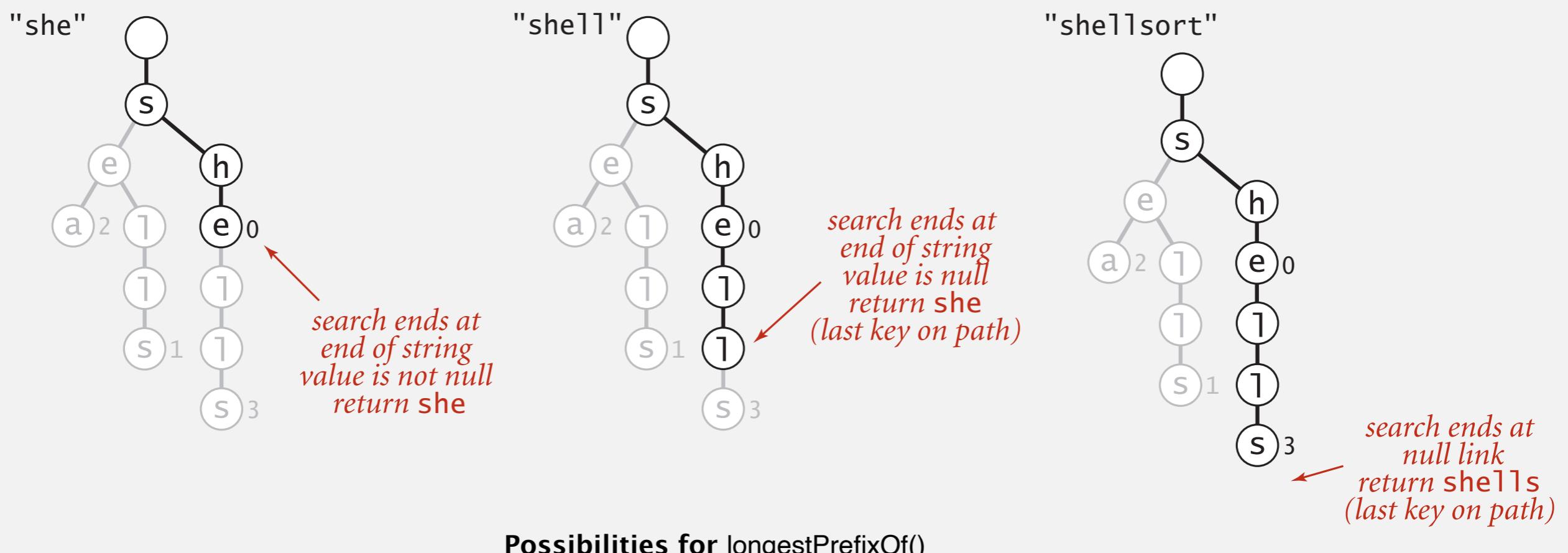


Possibilities for `longestPrefixOf()`

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.



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);
}
```

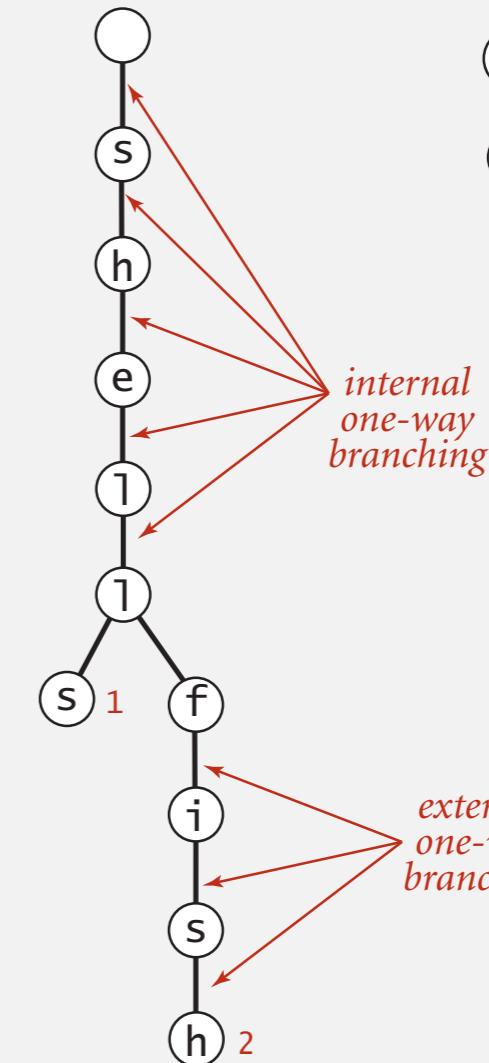
Patricia trie

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

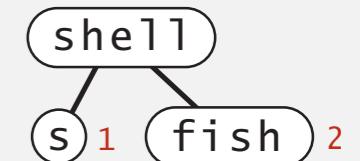
- Remove one-way branching.
- Each node represents a sequence of characters.
- Implementation: one step beyond this course.

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

standard
trie



no one-way
branching



*internal
one-way
branching*

*external
one-way
branching*

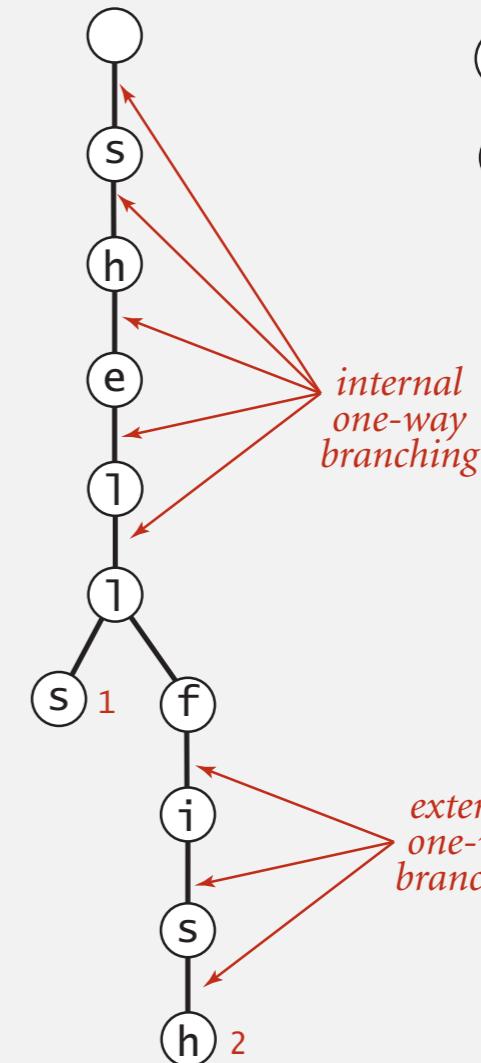
Patricia trie

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

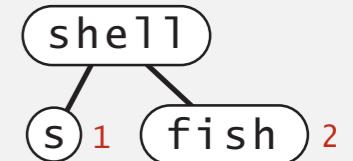
- Remove one-way branching.
- Each node represents a sequence of characters.
- Implementation: one step beyond this course.

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

standard
trie



no one-way
branching



Applications.

- Database search.
- P2P network search.
- IP routing tables: find longest prefix match.
- Compressed quad-tree for N-body simulation.
- Efficiently storing and querying XML documents.

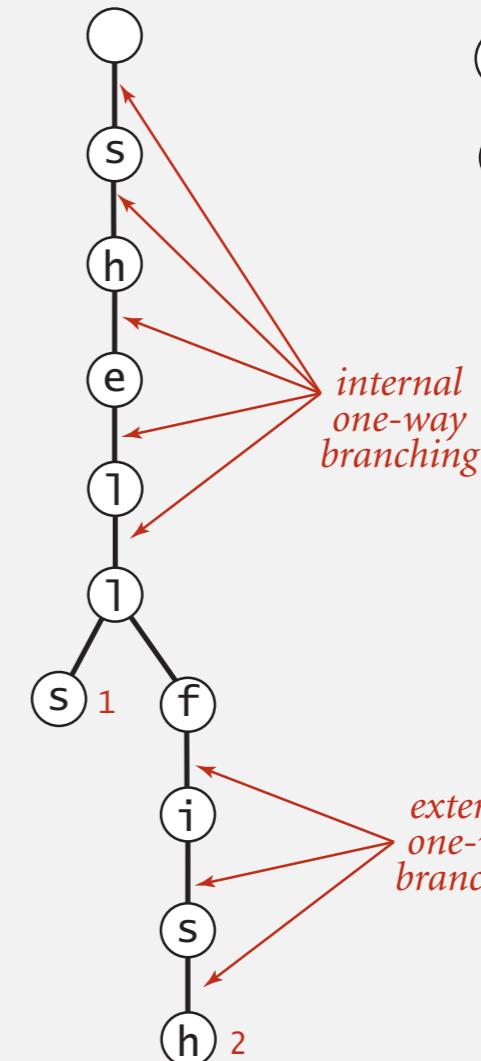
Patricia trie

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

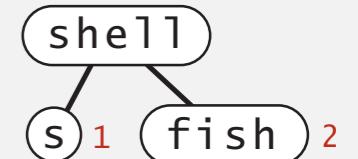
- Remove one-way branching.
- Each node represents a sequence of characters.
- Implementation: one step beyond this course.

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

standard
trie



no one-way
branching



Applications.

- Database search.
- P2P network search.
- IP routing tables: find longest prefix match.
- Compressed quad-tree for N-body simulation.
- Efficiently storing and querying XML documents.

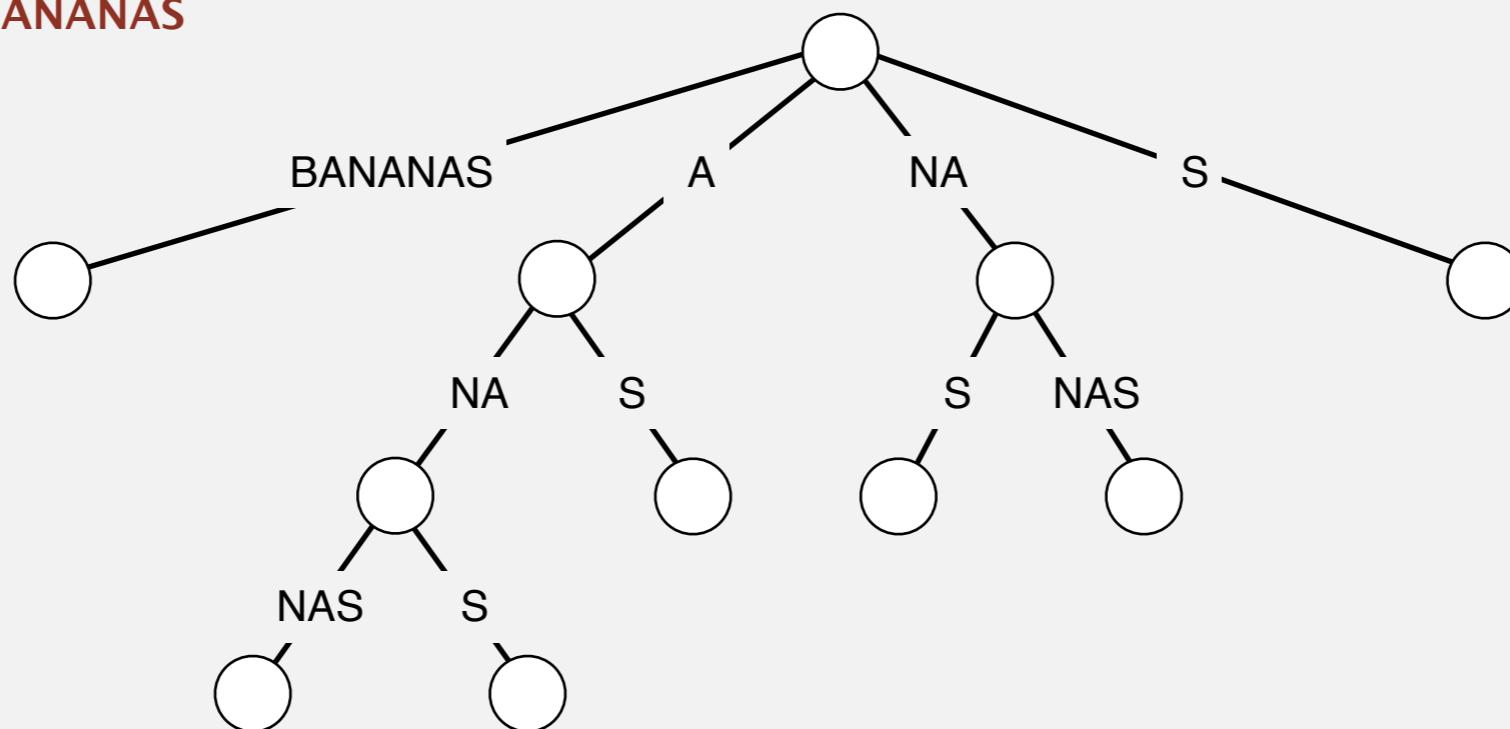
Also known as: crit-bit tree, radix tree.

Suffix tree

Suffix tree.

- Patricia trie of suffixes of a string.
- Linear-time construction: well beyond scope of this course.

suffix tree for BANANAS

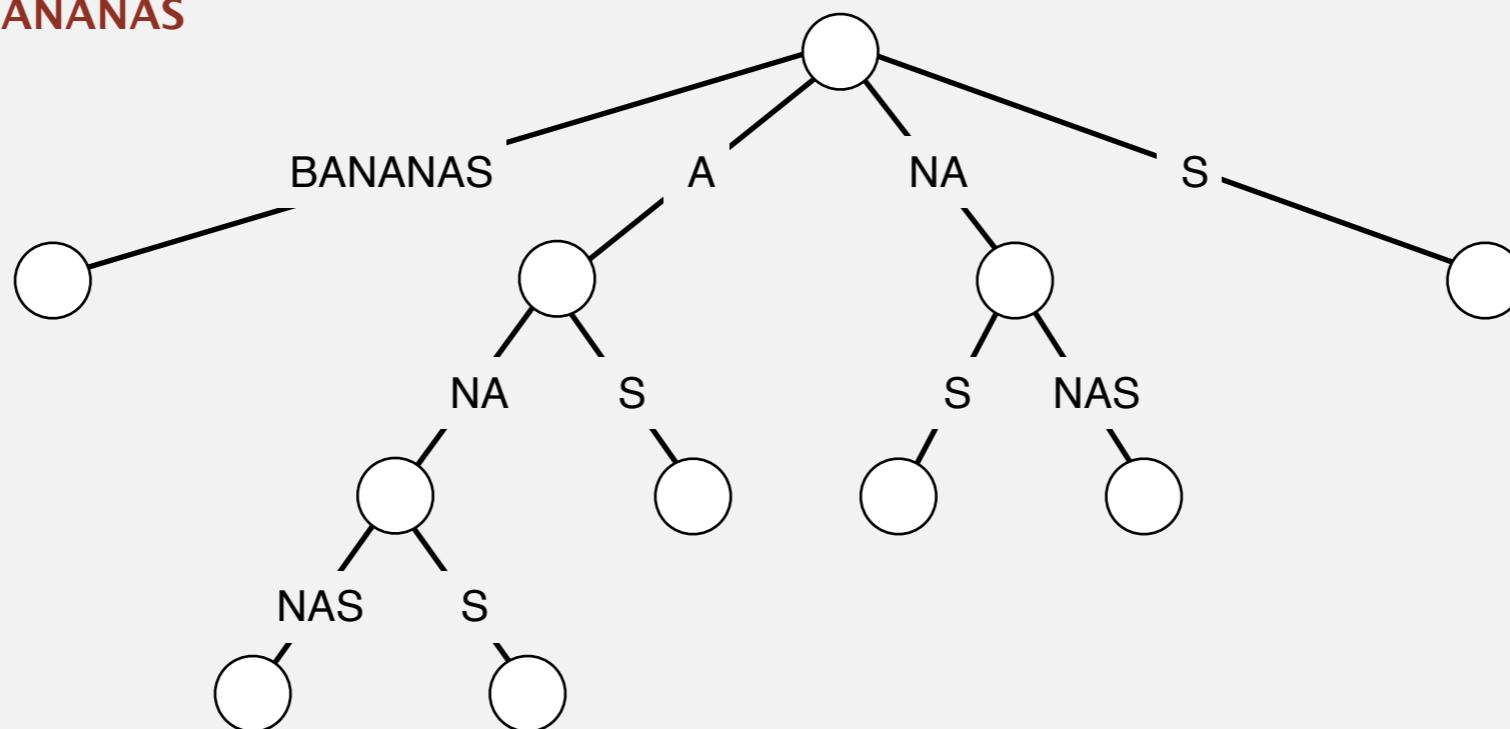


Suffix tree

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).

String symbol tables summary

A success story in algorithm design and analysis.

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.

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.

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.

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 (!!!)

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

5.3 SUBSTRING SEARCH

► introduction

► brute force

► Knuth-Morris-Pratt

► Boyer-Moore

► Rabin-Karp

Substring search

Goal. Find pattern of length M in a text of length N .

typically $N \gg M$

pattern → N E E D L E

text → I N A H A Y S T A C K N E E D L E I N A

↑
match

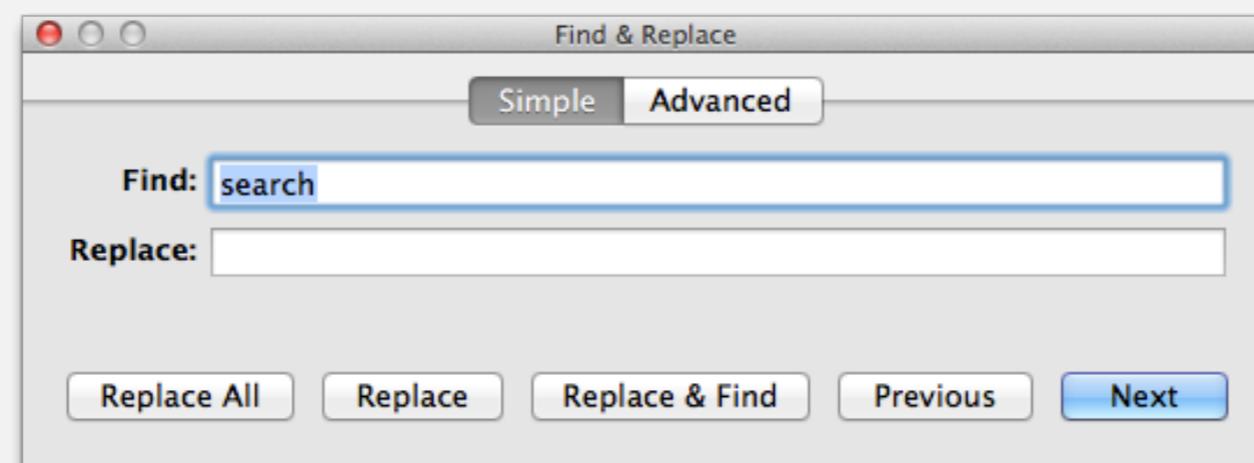
Substring search applications

Goal. Find pattern of length M in a text of length N .

pattern → N E E D L E

text → I N A H A Y S T A C K N E E D L E I N A

typically $N \gg M$



Substring search applications

Goal. Find pattern of length M in a text of length N .

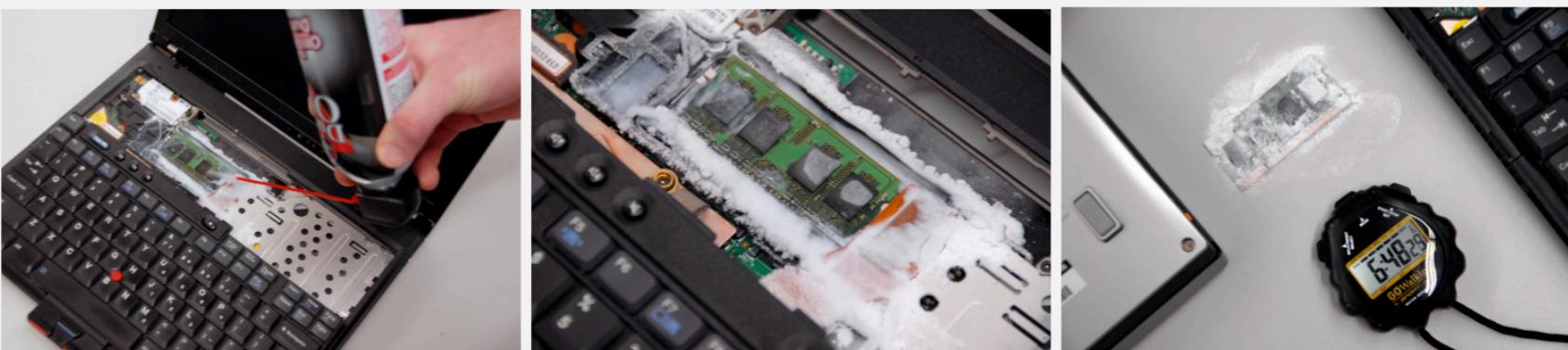
pattern → N E E D L E

text → I N A H A Y S T A C K N E E D L E I N A

typically $N \gg M$

match

Computer forensics. Search memory or disk for signatures, e.g., all URLs or RSA keys that the user has entered.



<http://citp.princeton.edu/memory>

Screen scraping: Java implementation

[Java library](#). The `indexOf()` method in Java's string library returns the index of the first occurrence of a given string, starting at a given offset.

```
public class StockQuote
{
    public static void main(String[] args)
    {
        String name = "http://finance.yahoo.com/q?s=";
        In in = new In(name + args[0]);
        String text = in.readAll();
        int start  = text.indexOf("Last Trade:", 0);
        int from   = text.indexOf("<b>", start);
        int to     = text.indexOf("</b>", from);
        String price = text.substring(from + 2, to);
    }
}
```

```
% java StockQuote goog
```

```
582.93
```

```
% java StockQuote msft
```

```
24.84
```

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

5.3 SUBSTRING SEARCH

- ▶ introduction
- ▶ brute force
- ▶ Knuth-Morris-Pratt
- ▶ Boyer-Moore
- ▶ Rabin-Karp

Brute-force substring search

Check for pattern starting at each text position.

i	j	i+j	0	1	2	3	4	5	6	7	8	9	10
			A	B	A	C	A	D	A	B	R	A	C
0	2	2	A	B	R	A							
1	0	1		A	B	R	A						
2	1	3			A	B	R	A					
3	0	3				A	B	R	A				
4	1	5					A	B	R	A			
5	0	5						A	B	R	A		
6	4	10							A	B	R	A	

txt → A B A C A D A B R A C

entries in red are mismatches

entries in gray are for reference only

entries in black match the text

return i when j is M

match

Brute-force substring search: worst case

Brute-force algorithm can be slow if text and pattern are repetitive.

i	j	$i+j$	0	1	2	3	4	5	6	7	8	9
			A	A	A	A	A	A	A	A	A	B
0	4	4	A	A	A	A	B	← pat				
1	4	5		A	A	A	A	B				
2	4	6			A	A	A	A	B			
3	4	7				A	A	A	A	B		
4	4	8					A	A	A	A	B	
5	5	10						<u>A</u>	A	A	A	B

\uparrow
match

Brute-force substring search: worst case

Brute-force algorithm can be slow if text and pattern are repetitive.

i	j	$i+j$	0	1	2	3	4	5	6	7	8	9
			A	A	A	A	A	A	A	A	A	B
0	4	4	A	A	A	A	B	← pat				
1	4	5		A	A	A	A	B				
2	4	6			A	A	A	A	B			
3	4	7				A	A	A	A	B		
4	4	8					A	A	A	A	B	
5	5	10						<u>A</u>	A	A	A	B

\uparrow
match

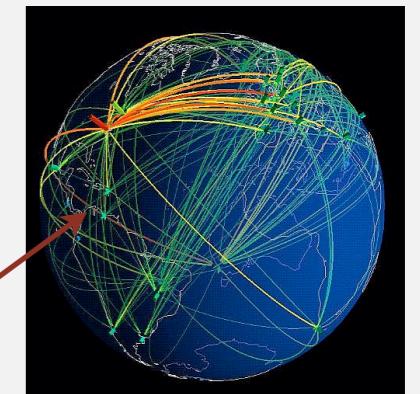
Worst case. $\sim M N$ char compares.

Backup

In many applications, we want to avoid **backup** in text stream.

- Treat input as stream of data.
- Abstract model: standard input.

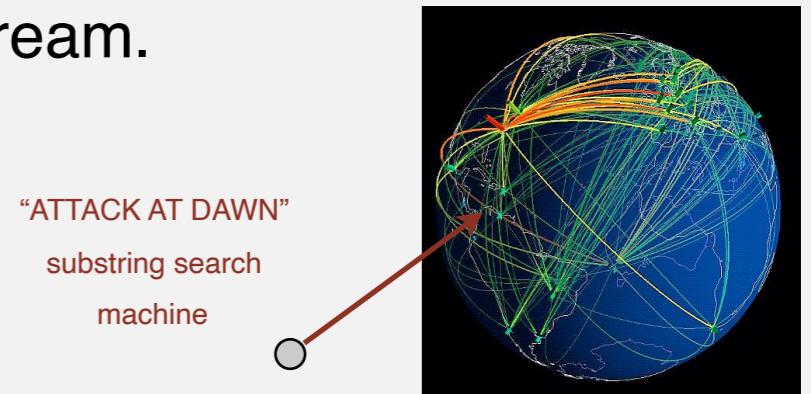
“ATTACK AT DAWN”
substring search
machine



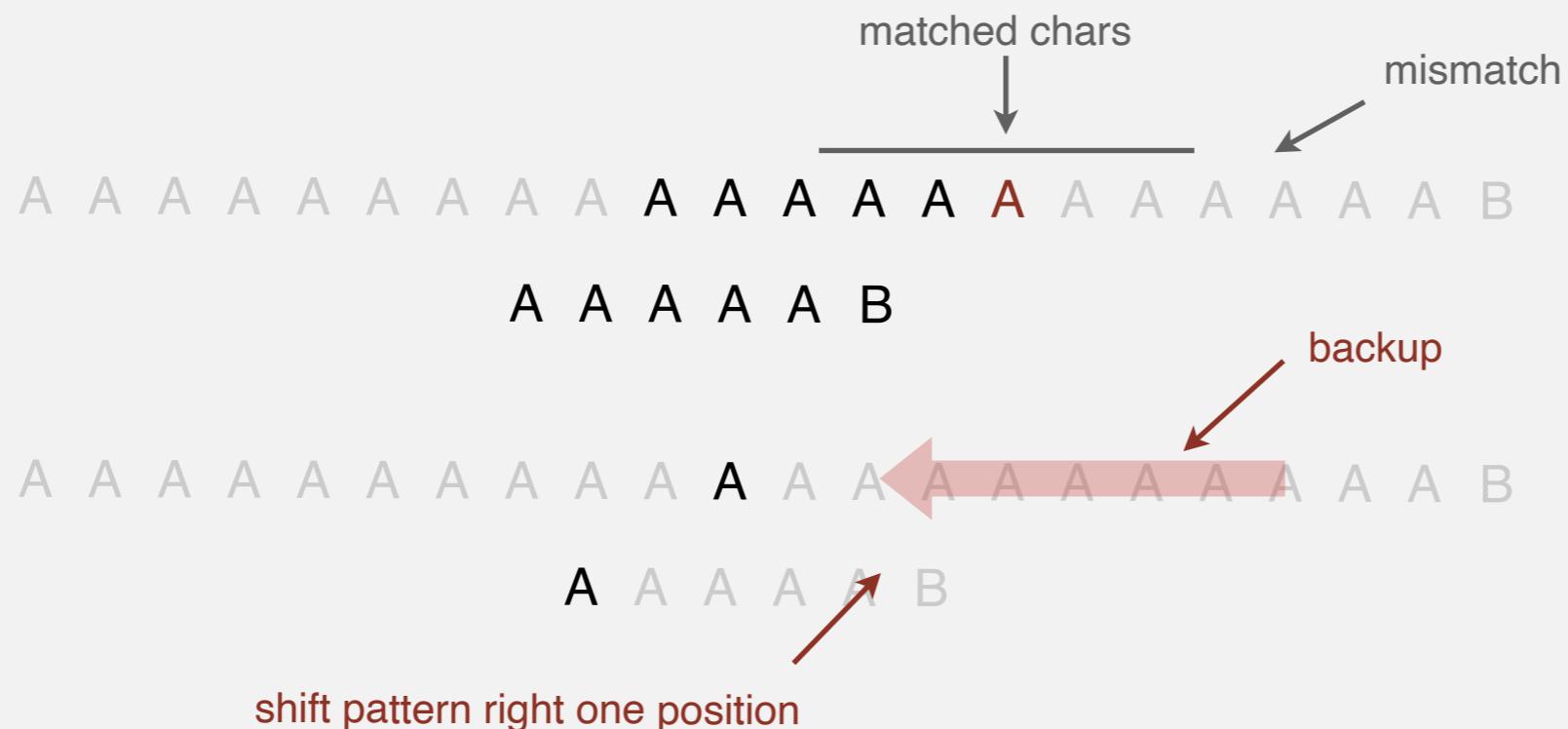
Backup

In many applications, we want to avoid backup in text stream.

- Treat input as stream of data.
 - Abstract model: standard input.



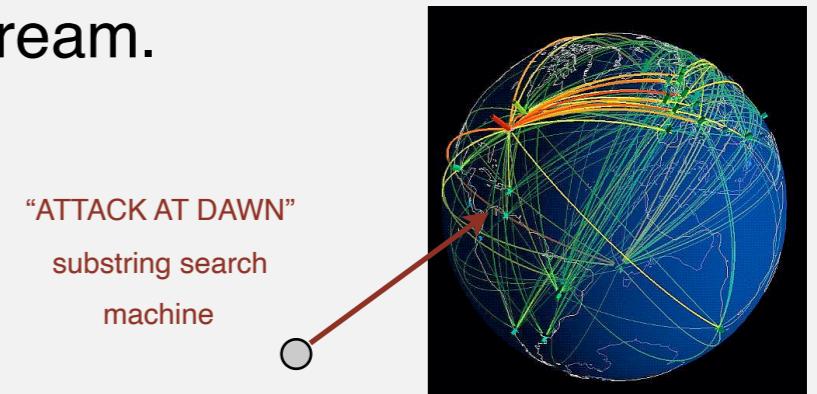
Brute-force algorithm needs backup for every mismatch.



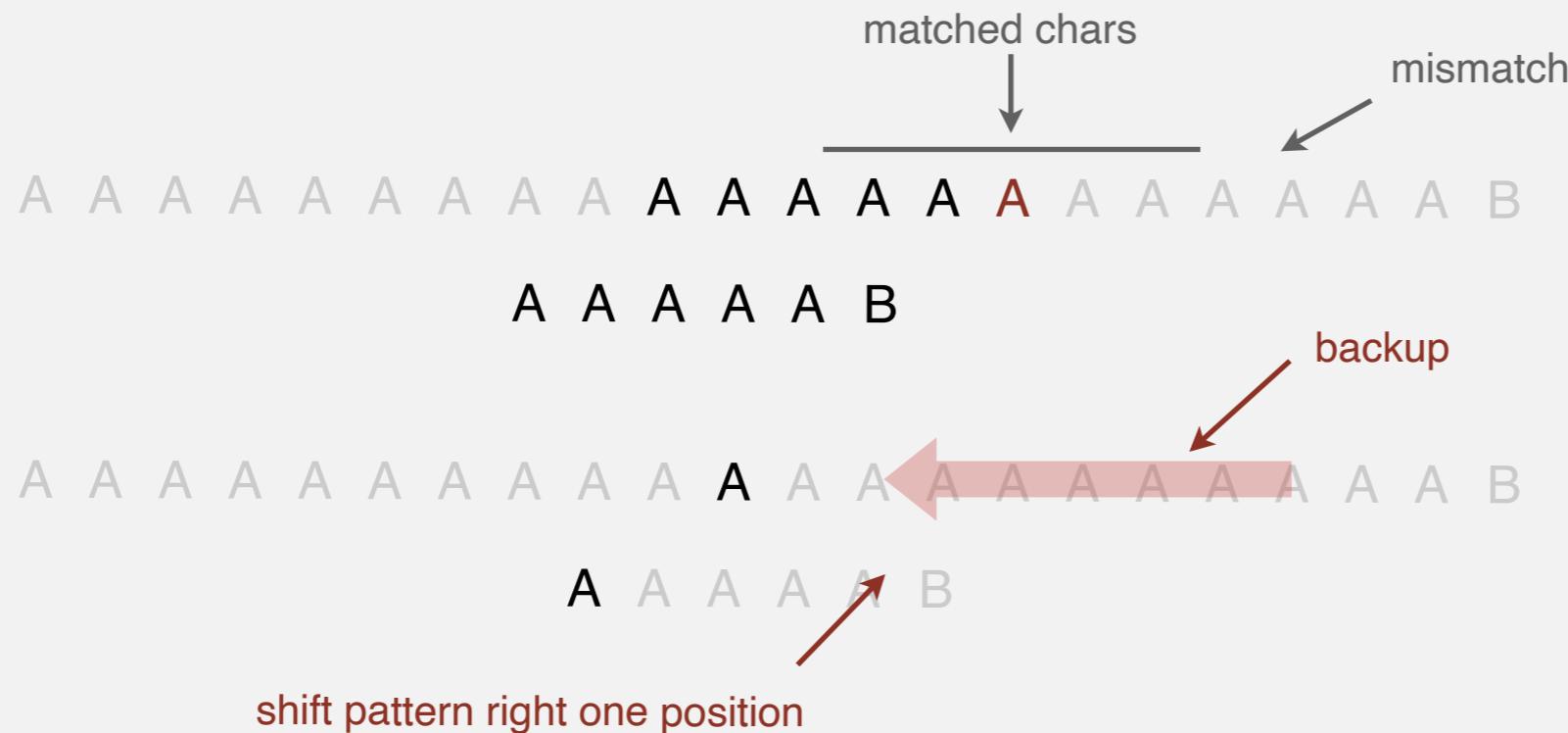
Backup

In many applications, we want to avoid **backup** in text stream.

- Treat input as stream of data.
- Abstract model: standard input.



Brute-force algorithm needs backup for every mismatch.

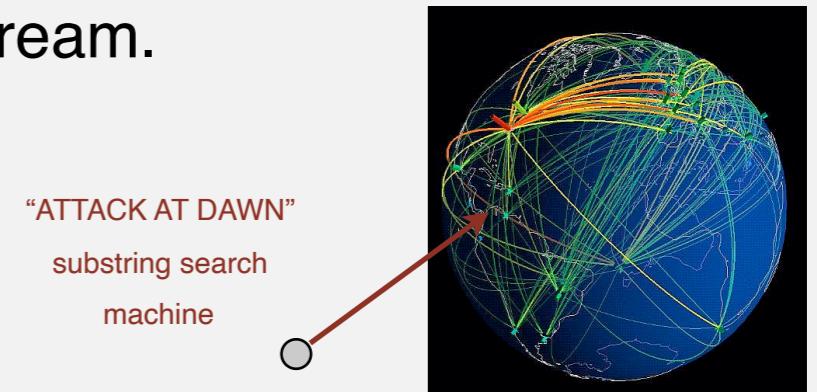


Approach 1. Maintain buffer of last M characters.

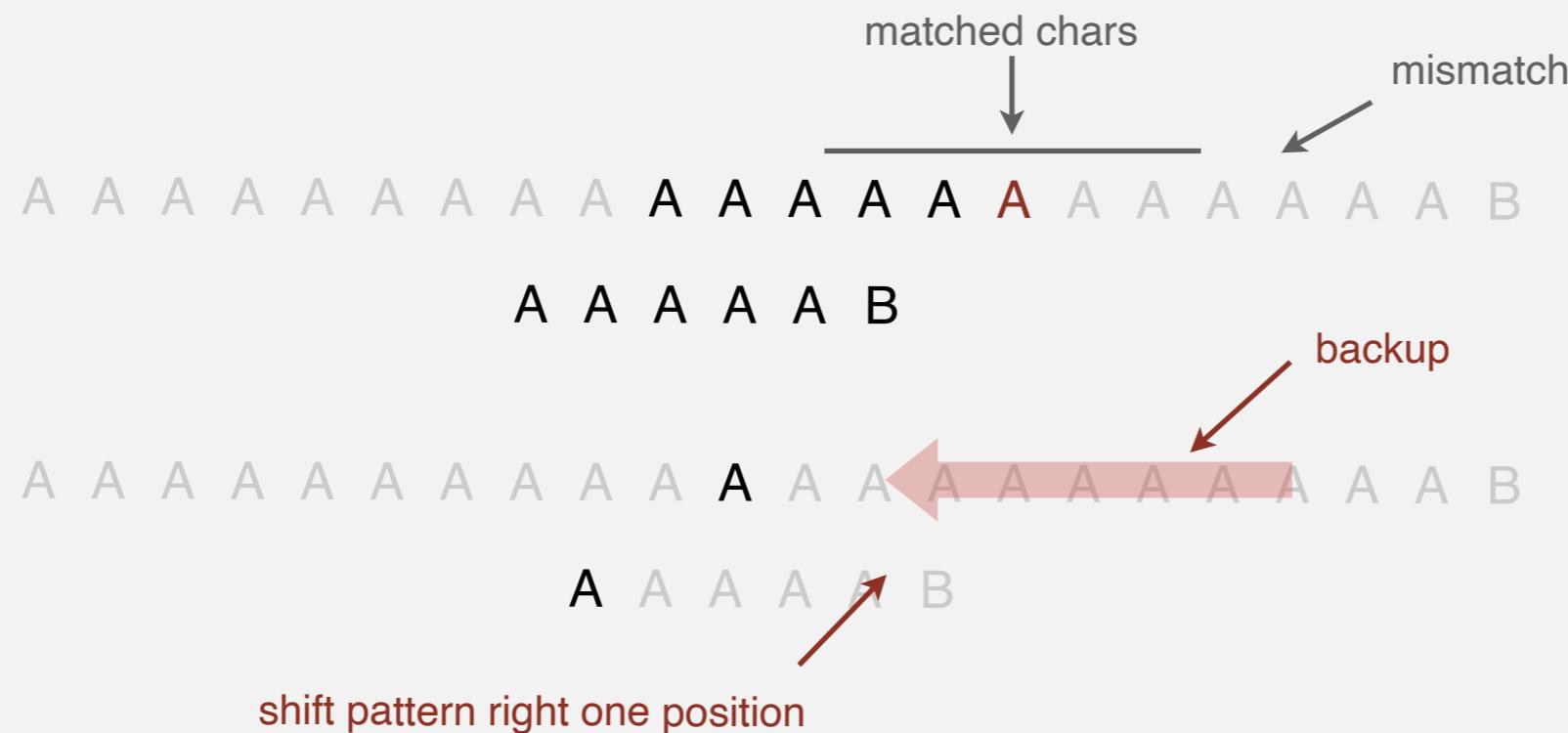
Backup

In many applications, we want to avoid **backup** in text stream.

- Treat input as stream of data.
- Abstract model: standard input.



Brute-force algorithm needs backup for every mismatch.



Approach 1. Maintain buffer of last M characters.

Approach 2. Stay tuned.

Algorithmic challenges in substring search

Brute-force is not always good enough.

Now is the time for all people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many good people to come to the aid of their party. Now is the time for a lot of good people to come to the aid of their party. Now is the time for all of the good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for each good person to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Republicans to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many or all good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Democrats to come to the aid of their party. Now is the time for all people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many good people to come to the aid of their party. Now is the time for a lot of good people to come to the aid of their party. Now is the time for all of the good people to come to the aid of their party. Now is the time for all good people to come to the aid of their **attack at dawn** party. Now is the time for each person to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Republicans to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many or all good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Democrats to come to the aid of their party.

Algorithmic challenges in substring search

Brute-force is not always good enough.

Theoretical challenge. Linear-time guarantee.

← fundamental algorithmic problem

Now is the time for all people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for a lot of good people to come to the aid of their party. Now is the time for all of the good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for each good person to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Republicans to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many or all good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Democrats to come to the aid of their party. Now is the time for all people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for a lot of good people to come to the aid of their party. Now is the time for all of the good people to come to the aid of their party. Now is the time for all good people to come to the aid of their **attack at dawn** party. Now is the time for each person to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Republicans to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many or all good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Democrats to come to the aid of their party.

Algorithmic challenges in substring search

Brute-force is not always good enough.

Theoretical challenge. Linear-time guarantee.

← fundamental algorithmic problem

Practical challenge. Avoid backup in text stream.

← often no room or time to save text

Now is the time for all people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for a lot of good people to come to the aid of their party. Now is the time for all of the good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for each good person to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Republicans to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many or all good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Democrats to come to the aid of their party. Now is the time for all people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for a lot of good people to come to the aid of their party. Now is the time for all of the good people to come to the aid of their party. Now is the time for all good people to come to the aid of their **attack at dawn** party. Now is the time for each person to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Republicans to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many or all good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Democrats to come to the aid of their party.

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

5.3 SUBSTRING SEARCH

- ▶ introduction
- ▶ brute force
- ▶ Knuth-Morris-Pratt
- ▶ Boyer-Moore
- ▶ Rabin-Karp



**Michael Rabin
Dick Karp**

Rabin-Karp fingerprint search

Basic idea = modular hashing.

Rabin-Karp fingerprint search

Basic idea = modular hashing.

- Compute a hash of $\text{pat}[0..M-1]$.

$$\begin{array}{r} \text{pat.charAt}(i) \\ \hline i & 0 & 1 & 2 & 3 & 4 \\ & 2 & 6 & 5 & 3 & 5 \end{array} \% 997 = 613$$

modular hashing with $R = 10$ and $\text{hash}(s) = s \pmod{997}$

Rabin-Karp fingerprint search

Basic idea = modular hashing.

- Compute a hash of $\text{pat}[0..M-1]$.
- For each i , compute a hash of $\text{txt}[i..M+i-1]$.

pat.charAt(i)																
i	0	1	2	3	4											
	2	6	5	3	5											
					% 997 = 613											
txt.charAt(i)																
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3
0	3	1	4	1	5	% 997		= 508								
1		1	4	1	5	9	% 997		= 201							
2			4	1	5	9	2	% 997		= 715						
3				1	5	9	2	6	% 997		= 971					
4					5	9	2	6	5	% 997		= 442				
5						9	2	6	5	3	% 997		= 929			
6	← return i = 6						2	6	5	3	5	% 997		= 613		

modular hashing with $R = 10$ and $\text{hash}(s) = s \pmod{997}$

Rabin-Karp fingerprint search

Basic idea = modular hashing.

- Compute a hash of $\text{pat}[0..M-1]$.
- For each i , compute a hash of $\text{txt}[i..M+i-1]$.
- If pattern hash = text substring hash, check for a match.

pat.charAt(i)																
i	0	1	2	3	4											
	2	6	5	3	5											
% 997 = 613																
txt.charAt(i)																
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3
0	3	1	4	1	5	% 997 = 508										
1		1	4	1	5	9	% 997 = 201									
2			4	1	5	9	2	% 997 = 715								
3				1	5	9	2	6	% 997 = 971							
4					5	9	2	6	5	% 997 = 442						
5						9	2	6	5	3	% 997 = 929					
6	$\leftarrow \text{return } i = 6$					2	6	5	3	5	match % 997 = 613					

modular hashing with $R = 10$ and $\text{hash}(s) = s \pmod{997}$

Modular arithmetic

Math trick. To keep numbers small, take intermediate results modulo Q .

Ex.

$$\begin{aligned} & (10000 + 535) * 1000 \pmod{997} \\ &= (30 + 535) * 3 \pmod{997} \quad 1000 \pmod{997} = 3 \\ &= 1695 \pmod{997} \\ &= 698 \pmod{997} \end{aligned}$$

$10000 \pmod{997} = 30$

\nearrow

$1000 \pmod{997} = 3$

Modular arithmetic

Math trick. To keep numbers small, take intermediate results modulo Q .

Ex.

$$\begin{aligned} & (10000 + 535) * 1000 \pmod{997} \\ &= (30 + 535) * 3 \pmod{997} \quad 1000 \pmod{997} = 3 \\ &= 1695 \pmod{997} \\ &= 698 \pmod{997} \end{aligned}$$

$$10000 \pmod{997} = 30$$

$$(a + b) \pmod{Q} = ((a \pmod{Q}) + (b \pmod{Q})) \pmod{Q}$$

$$(a * b) \pmod{Q} = ((a \pmod{Q}) * (b \pmod{Q})) \pmod{Q}$$

two useful modular arithmetic identities

Efficiently computing the hash function

Modular hash function. Using the notation t_i for `txt.charAt(i)`, we wish to compute

$$x_i = t_i R^{M-1} + t_{i+1} R^{M-2} + \dots + t_{i+M-1} R^0 \pmod{Q}$$

Intuition. M -digit, base- R integer, modulo Q .

$$26535 = 2*10000 + 6*1000 + 5*100 + 3*10 + 5$$

Efficiently computing the hash function

Modular hash function. Using the notation t_i for `txt.charAt(i)`, we wish to compute

$$x_i = t_i R^{M-1} + t_{i+1} R^{M-2} + \dots + t_{i+M-1} R^0 \pmod{Q}$$

Intuition. M -digit, base- R integer, modulo Q .

Horner's method. Linear-time method to evaluate degree- M polynomial.

pat.charAt()							
i	0	1	2	3	4		
	2	6	5	3	5		
0	2	%	997	=	2		
1	2	6	%	997	= $(2*10 + 6) \% 997 = 26$		
2	2	6	5	%	997 = $(26*10 + 5) \% 997 = 265$		
3	2	6	5	3	%	997 = $(265*10 + 3) \% 997 = 659$	
4	2	6	5	3	5	%	997 = $(659*10 + 5) \% 997 = 613$

$$\begin{aligned} 26535 &= 2*10000 + 6*1000 + 5*100 + 3*10 + 5 \\ &= (((2 * 10 + 6) * 10 + 5) * 10 + 3) * 10 + 5 \end{aligned}$$

Efficiently computing the hash function

Modular hash function. Using the notation t_i for `txt.charAt(i)`, we wish to compute

$$x_i = t_i R^{M-1} + t_{i+1} R^{M-2} + \dots + t_{i+M-1} R^0 \pmod{Q}$$

Intuition. M -digit, base- R integer, modulo Q .

Horner's method. Linear-time method to evaluate degree- M polynomial.

pat.charAt()							
i	0	1	2	3	4		
	2	6	5	3	5		
0	2	%	997	=	2		
1	2	6	%	997	= $(2 * 10 + 6) \% 997 = 26$		
2	2	6	5	%	997 = $(26 * 10 + 5) \% 997 = 265$		
3	2	6	5	3	%	997 = $(265 * 10 + 3) \% 997 = 659$	
4	2	6	5	3	5	%	997 = $(659 * 10 + 5) \% 997 = 613$

```
// Compute hash for M-digit key
private long hash(String key, int M)
{
    long h = 0;
    for (int j = 0; j < M; j++)
        h = (h * R + key.charAt(j)) % Q;
    return h;
}
```

$$26535 = 2 * 10000 + 6 * 1000 + 5 * 100 + 3 * 10 + 5$$

$$= (((((2 * 10 + 6) * 10 + 5) * 10 + 3) * 10 + 5$$

Efficiently computing the hash function

Challenge. How to efficiently compute x_{i+1} given that we know x_i .

$$x_i = t_i R^{M-1} + t_{i+1} R^{M-2} + \dots + t_{i+M-1} R^0$$

$$x_{i+1} = t_{i+1} R^{M-1} + t_{i+2} R^{M-2} + \dots + t_{i+M} R^0$$

i	...	2	3	4	5	6	7	...
<i>current value</i>		1	4	1	5	9	2	6
<i>new value</i>		4	1	5	9	2	6	5

text

Efficiently computing the hash function

Challenge. How to efficiently compute x_{i+1} given that we know x_i .

$$x_i = t_i R^{M-1} + t_{i+1} R^{M-2} + \dots + t_{i+M-1} R^0$$

$$x_{i+1} = t_{i+1} R^{M-1} + t_{i+2} R^{M-2} + \dots + t_{i+M} R^0$$

i	...	2	3	4	5	6	7	...
<i>current value</i>		1	4	1	5	9	2	6
<i>new value</i>		4	1	5	9	2	6	5

text

	4	1	5	9	2	<i>current value</i>
-	4	0	0	0	0	
	1	5	9	2		<i>subtract leading digit</i>
	*	1	0			<i>multiply by radix</i>
	1	5	9	2	0	
			+	6		<i>add new trailing digit</i>
	1	5	9	2	6	<i>new value</i>

Efficiently computing the hash function

Challenge. How to efficiently compute x_{i+1} given that we know x_i .

$$x_i = t_i R^{M-1} + t_{i+1} R^{M-2} + \dots + t_{i+M-1} R^0$$

$$x_{i+1} = t_{i+1} R^{M-1} + t_{i+2} R^{M-2} + \dots + t_{i+M} R^0$$

Key property. Can update "rolling" hash function in constant time!

$$x_{i+1} = (x_i - t_i R^{M-1}) R + t_{i+M}$$

↑ ↑ ↑ ↑
current subtract multiply add new
value leading digit by radix trailing digit (can precompute R^{M-1})

i	...	2	3	4	5	6	7	...
current value	1	4	1	5	9	2	6	5
new value		4	1	5	9	2	6	5

\Rightarrow text

$4 \quad 1 \quad 5 \quad 9 \quad 2 \quad$ current value

$- \quad 4 \quad 0 \quad 0 \quad 0 \quad 0$

$1 \quad 5 \quad 9 \quad 2 \quad$ subtract leading digit

$* \quad 1 \quad 0 \quad$ multiply by radix

$1 \quad 5 \quad 9 \quad 2 \quad 0$

$+ \quad 6 \quad$ add new trailing digit

$1 \quad 5 \quad 9 \quad 2 \quad 6 \quad$ new value

Rabin-Karp substring search example

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

Rabin-Karp substring search example

First R entries: Use Horner's rule.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3
0	3	%	997	=	3											
1	3	1	%	997	=	(3*10 + 1)	%	997	=	31						
2	3	1	4	%	997	=	(31*10 + 4)	%	997	=	314					
3	3	1	4	1	%	997	=	(314*10 + 1)	%	997	=	150				
4	3	1	4	1	5	%	997	=	(150*10 + 5)	%	997	=	508			

Horner's
rule

Rabin-Karp substring search example

First R entries: Use Horner's rule.

Remaining entries: Use rolling hash (and % to avoid overflow).

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3
0	3	%	997	=	3											
1	3	1	%	997	=	(3*10 + 1)	%	997	=	31						
2	3	1	4	%	997	=	(31*10 + 4)	%	997	=	314					
3	3	1	4	1	%	997	=	(314*10 + 1)	%	997	=	150				
4	3	1	4	1	5	%	997	=	(150*10 + 5)	%	997	=	508	RM	R	
5	1	4	1	5	9	%	997	=	((508 + 3*(997 - 30)) * 10 + 9)	%	997	=	201			
6	4	1	5	9	2	%	997	=	((201 + 1*(997 - 30)) * 10 + 2)	%	997	=	715			
7	1	5	9	2	6	%	997	=	((715 + 4*(997 - 30)) * 10 + 6)	%	997	=	971			
8	5	9	2	6	5	%	997	=	((971 + 1*(997 - 30)) * 10 + 5)	%	997	=	442	match		
9	9	2	6	5	3	%	997	=	((442 + 5*(997 - 30)) * 10 + 3)	%	997	=	929			
10	←	return	i-M+1	=	6											613
		2	6	5	3	5	%	997	=	((929 + 9*(997 - 30)) * 10 + 5)	%	997	=	613		

Horner's rule

rolling hash

$-30 \pmod{997} = 997 - 30$

$10000 \pmod{997} = 30$

Rabin-Karp: Java implementation

```
public class RabinKarp {  
    private long patHash; // pattern hash value  
    private int M; // pattern length  
    private long Q; // modulus  
    private int R; // radix  
    private long RM1; //  $R^{M-1} \bmod Q$ 
```

```
public RabinKarp(String pat) {  
    M = pat.length();  
    R = 256;  
    Q = longRandomPrime();
```

a large prime
(but avoid overflow)

precompute $R^{M-1} \bmod Q$

```
    RM1 = 1;  
    for (int i = 1; i <= M-1; i++)  
        RM1 = (R * RM1) % Q;  
    patHash = hash(pat, M);  
}
```

```
private long hash(String key, int M)  
{ /* as before */ }
```

Rabin-Karp: Java implementation (continued)

Monte Carlo version. Return match if hash match.

```
public int search(String txt)
{
    int N = txt.length();
    int txtHash = hash(txt, M);
    if (patHash == txtHash) return 0;
    for (int i = M; i < N; i++)
    {
        txtHash = (txtHash + Q - RM*txt.charAt(i-M) % Q) % Q;
        txtHash = (txtHash*R + txt.charAt(i)) % Q;
        if (patHash == txtHash) return i - M + 1;
    }
    return N;
}
```

check for hash collision
using rolling hash function

Las Vegas version. Check for substring match if hash match;
continue search if false collision.

Rabin-Karp analysis

Theory. If Q is a sufficiently large random prime (about $M N^2$), then the probability of a false collision is about $1 / N$.

Rabin-Karp analysis

Theory. If Q is a sufficiently large random prime (about $M N^2$), then the probability of a false collision is about $1 / N$.

Practice. Choose Q to be a large prime (but not so large to cause overflow). Under reasonable assumptions, probability of a collision is about $1 / Q$.

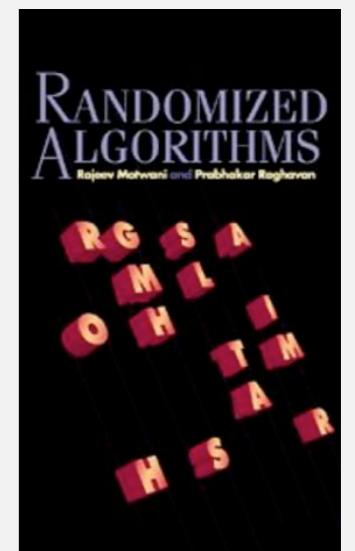
Rabin-Karp analysis

Theory. If Q is a sufficiently large random prime (about $M N^2$), then the probability of a false collision is about $1 / N$.

Practice. Choose Q to be a large prime (but not so large to cause overflow). Under reasonable assumptions, probability of a collision is about $1 / Q$.

Monte Carlo version.

- Always runs in linear time.
- Extremely likely to return correct answer (but not always!).



Las Vegas version.

- Always returns correct answer.
- Extremely likely to run in linear time (but worst case is $M N$).

Rabin-Karp fingerprint search

Advantages.

- Extends to 2d patterns.
- Extends to finding multiple patterns.

Rabin-Karp fingerprint search

Advantages.

- Extends to 2d patterns.
- Extends to finding multiple patterns.

Disadvantages.

- Arithmetic ops slower than char compares.
- Las Vegas version requires backup.
- Poor worst-case guarantee.

Rabin-Karp fingerprint search

Advantages.

- Extends to 2d patterns.
- Extends to finding multiple patterns.

Disadvantages.

- Arithmetic ops slower than char compares.
- Las Vegas version requires backup.
- Poor worst-case guarantee.

Q. How would you extend Rabin-Karp to efficiently search for any one of P possible patterns in a text of length N ?



Substring search cost summary

Cost of searching for an M -character pattern in an N -character text.

algorithm	version	operation count		backup in input?	correct?	extra space
		guarantee	typical			
brute force	—	MN	$1.1 N$	yes	yes	1
Knuth-Morris-Pratt	<i>full DFA</i> (Algorithm 5.6)	$2N$	$1.1 N$	no	yes	MR
	<i>mismatch transitions only</i>	$3N$	$1.1 N$	no	yes	M
Boyer-Moore	<i>full algorithm</i>	$3N$	N / M	yes	yes	R
	<i>mismatched char heuristic only</i> (Algorithm 5.7)	MN	N / M	yes	yes	R
Rabin-Karp [†]	<i>Monte Carlo</i> (Algorithm 5.8)	$7N$	$7N$	no	yes^{\dagger}	1
	<i>Las Vegas</i>	$7N^{\dagger}$	$7N$	yes	yes	1

[†] probabilistic guarantee, with uniform hash function

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

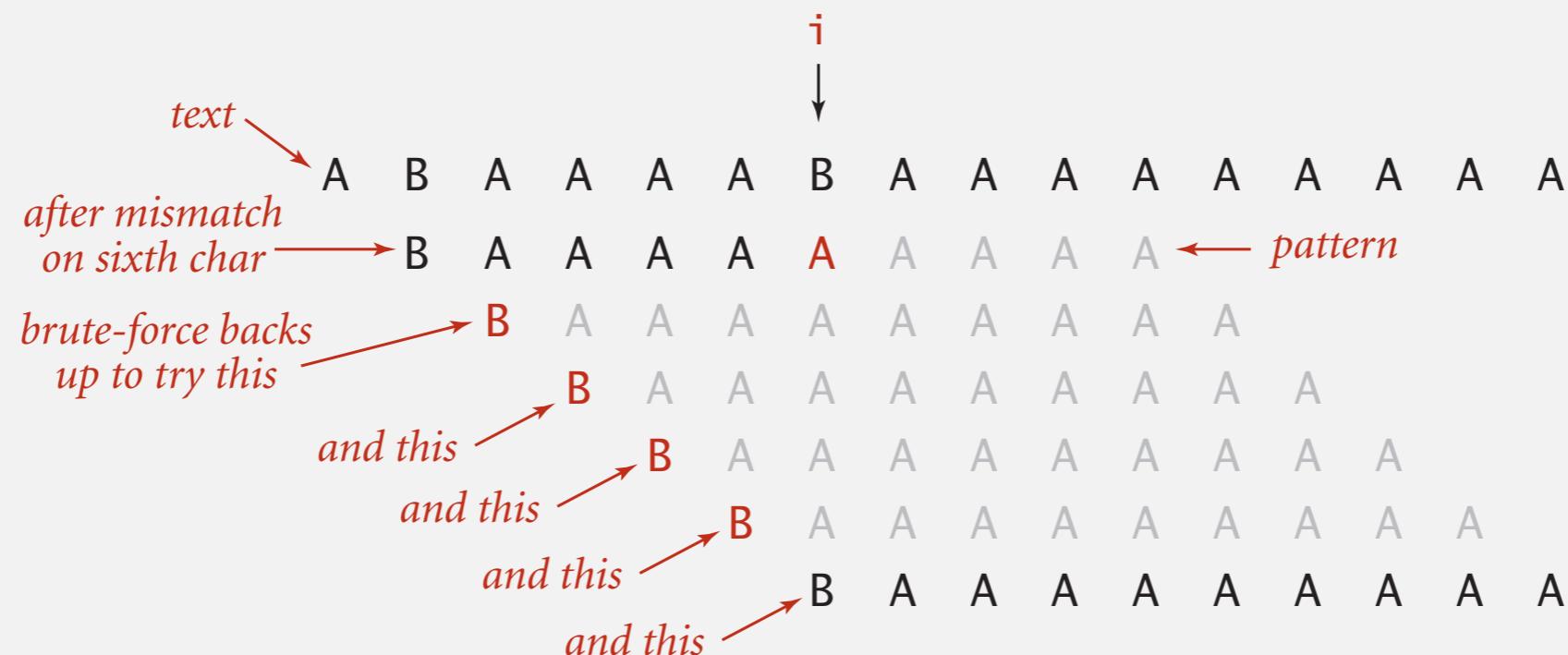
<http://algs4.cs.princeton.edu>

5.3 SUBSTRING SEARCH

- ▶ introduction
- ▶ brute force
- ▶ Knuth-Morris-Pratt
- ▶ Boyer-Moore
- ▶ Rabin-Karp

Knuth-Morris-Pratt substring search

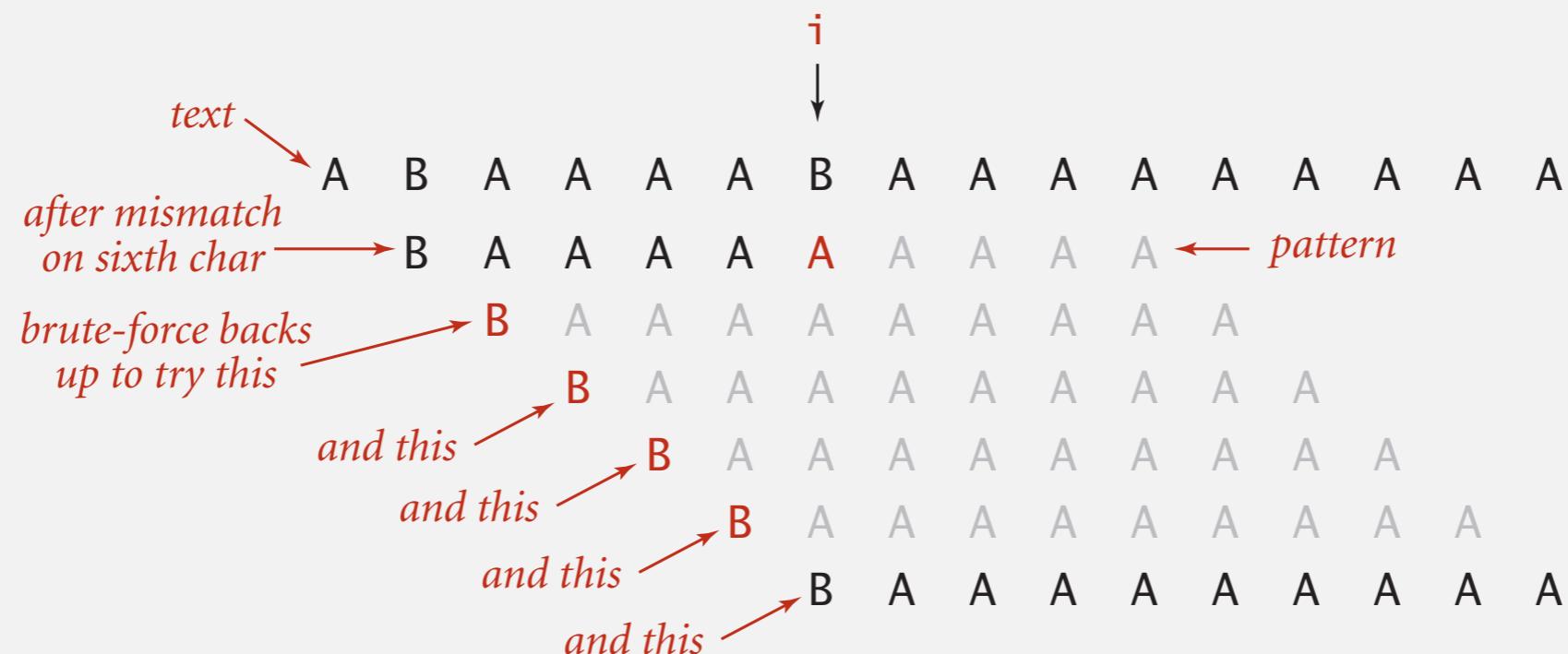
Intuition. Suppose we are searching in text for pattern BAAAAAAAAAA.



Knuth-Morris-Pratt substring search

Intuition. Suppose we are searching in text for pattern BAAAAAAAAAA.

- Suppose we match 5 chars in pattern, with mismatch on 6th char.

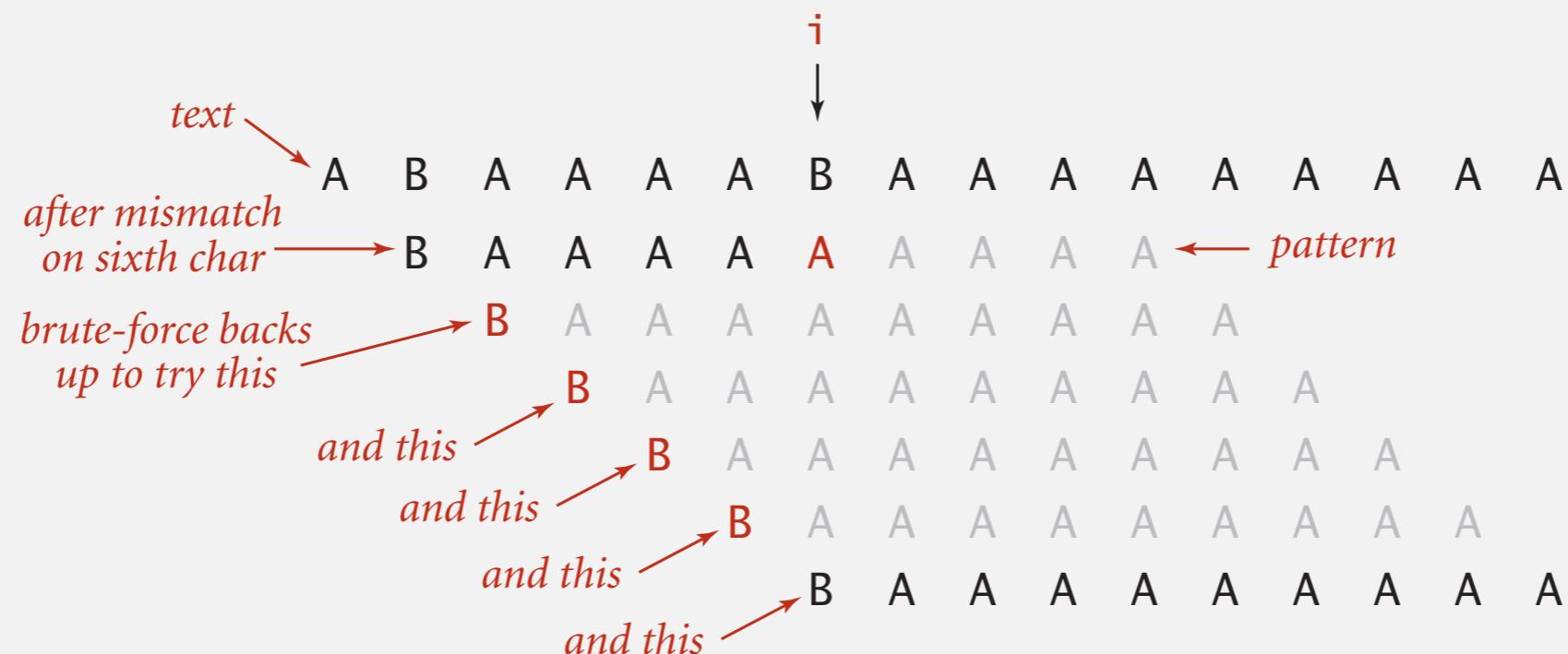


Knuth-Morris-Pratt substring search

Intuition. Suppose we are searching in text for pattern BAAAAAAAAAA.

- Suppose we match 5 chars in pattern, with mismatch on 6th char.
- We know previous 6 chars in text are BAAAAB.

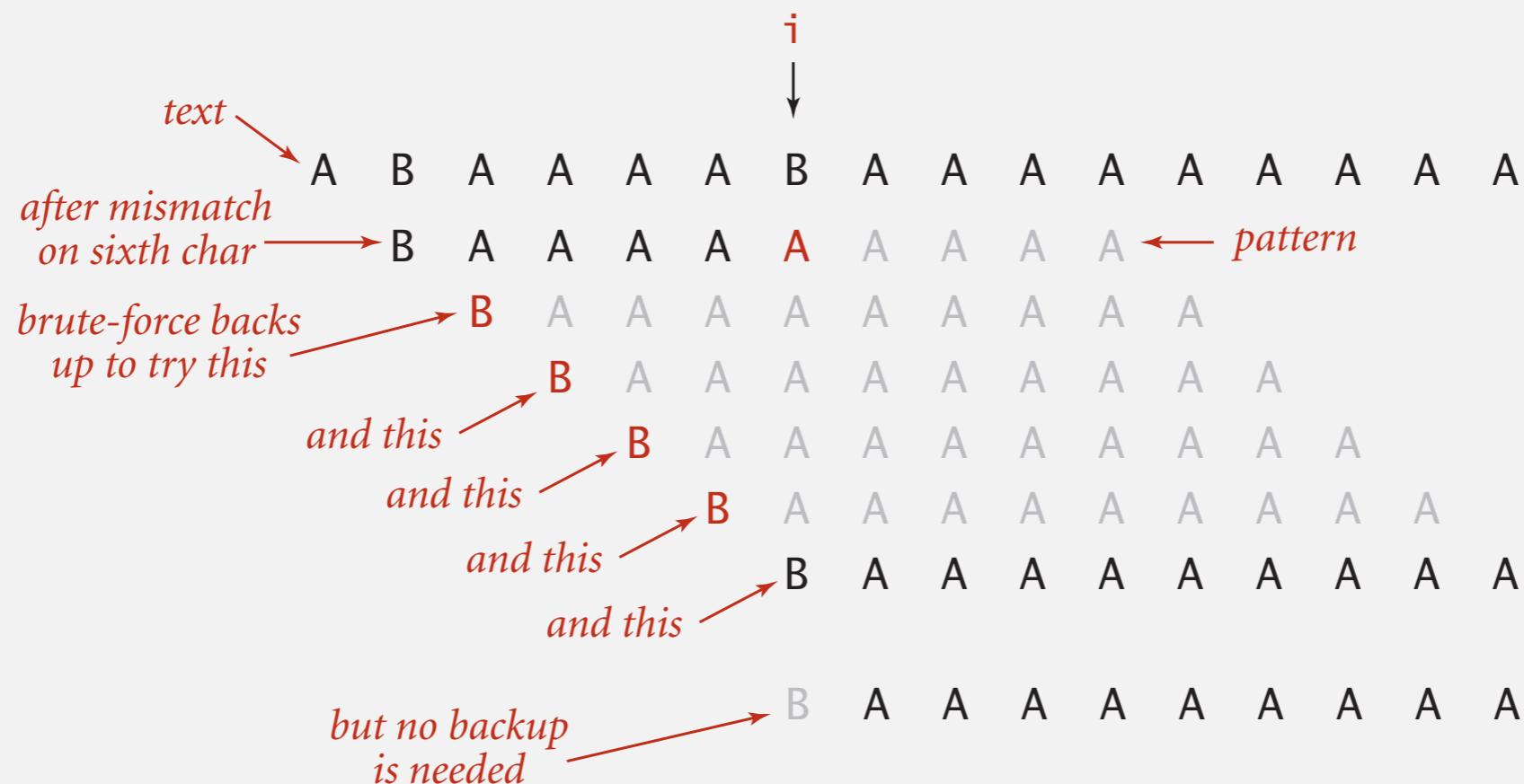
assuming { A, B } alphabet



Knuth-Morris-Pratt substring search

Intuition. Suppose we are searching in text for pattern BAAAAAAAAAA.

- Suppose we match 5 chars in pattern, with mismatch on 6th char.
 - We know previous 6 chars in text are BAAAAAB.
 - Don't need to back up text pointer!

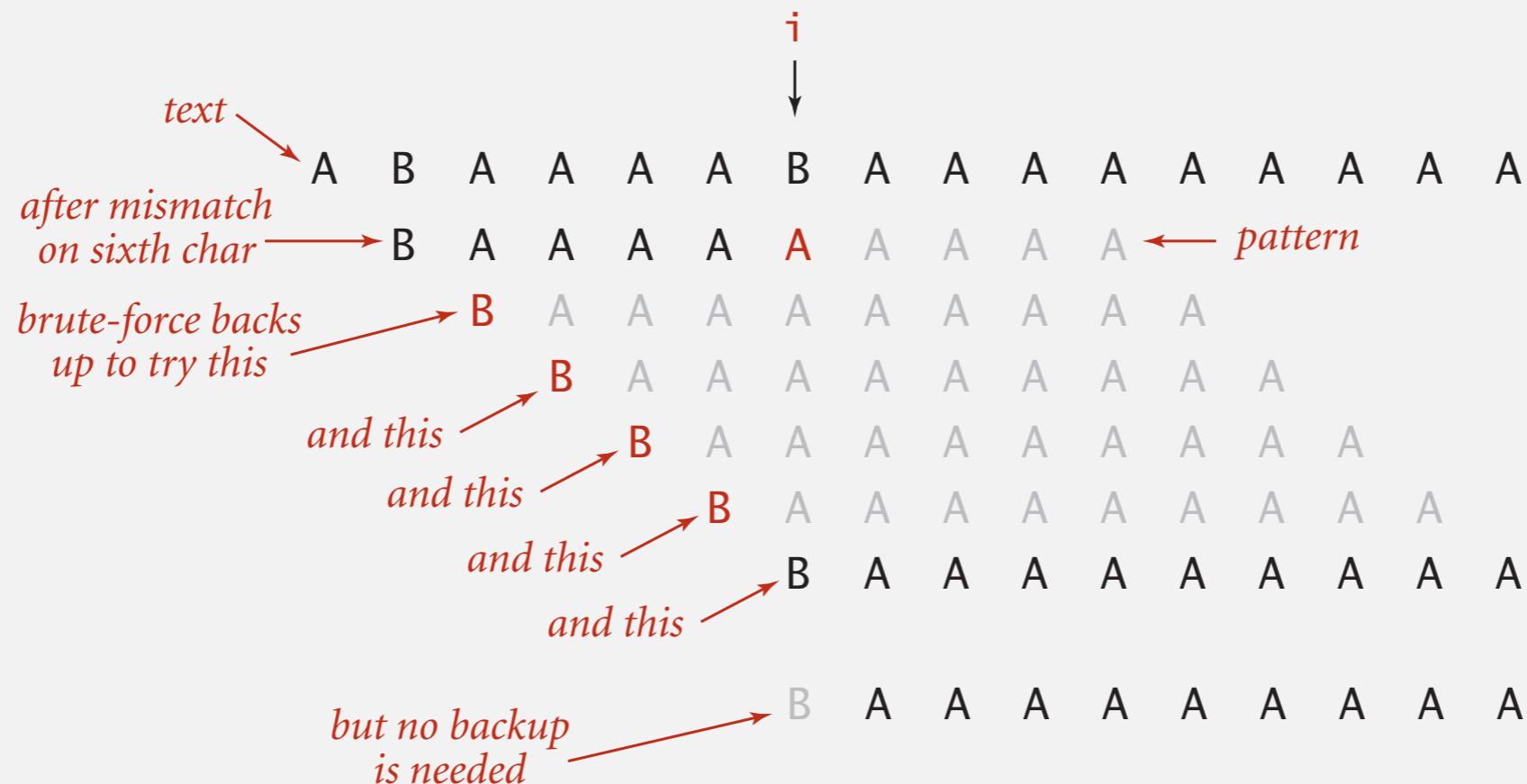


Knuth-Morris-Pratt substring search

Intuition. Suppose we are searching in text for pattern BAAAAAAAAAA.

- Suppose we match 5 chars in pattern, with mismatch on 6th char.
- We know previous 6 chars in text are BAAAAB.
- Don't need to back up text pointer!

assuming { A, B } alphabet



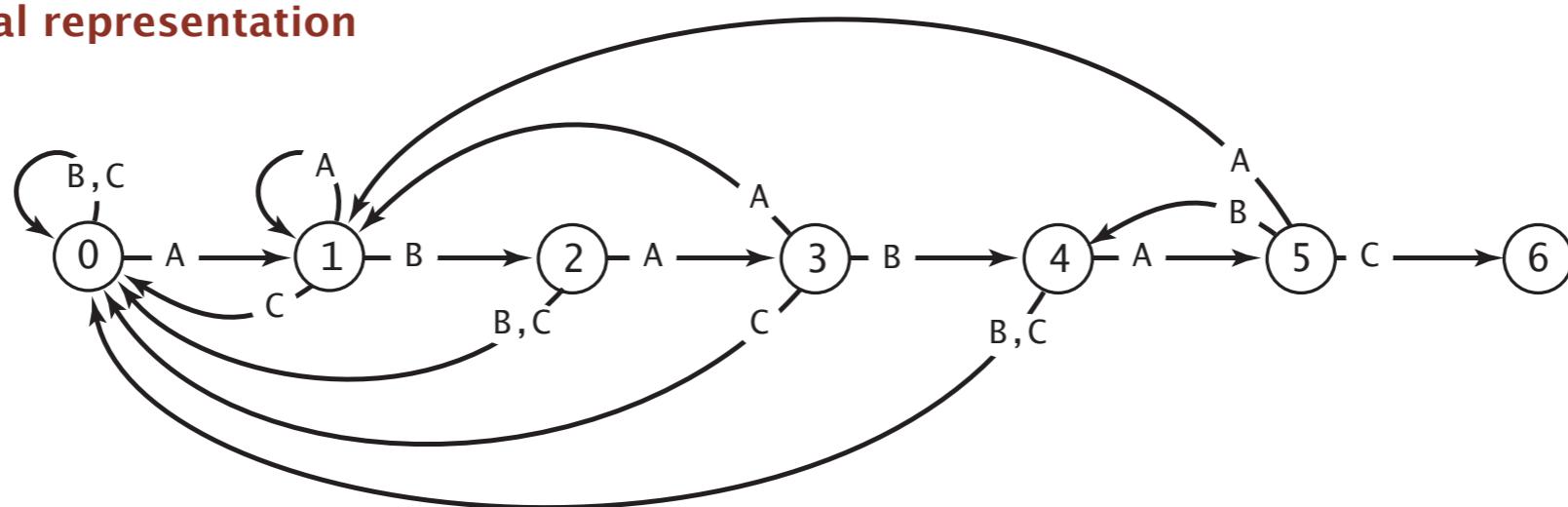
Knuth-Morris-Pratt algorithm. Clever method to always avoid backup. (!)

Deterministic finite state automaton (DFA)

DFA is abstract string-searching machine.

- Finite number of states (including start and halt).
- Exactly one transition for each char in alphabet.
- Accept if sequence of transitions leads to halt state.

graphical representation



Deterministic finite state automaton (DFA)

DFA is abstract string-searching machine.

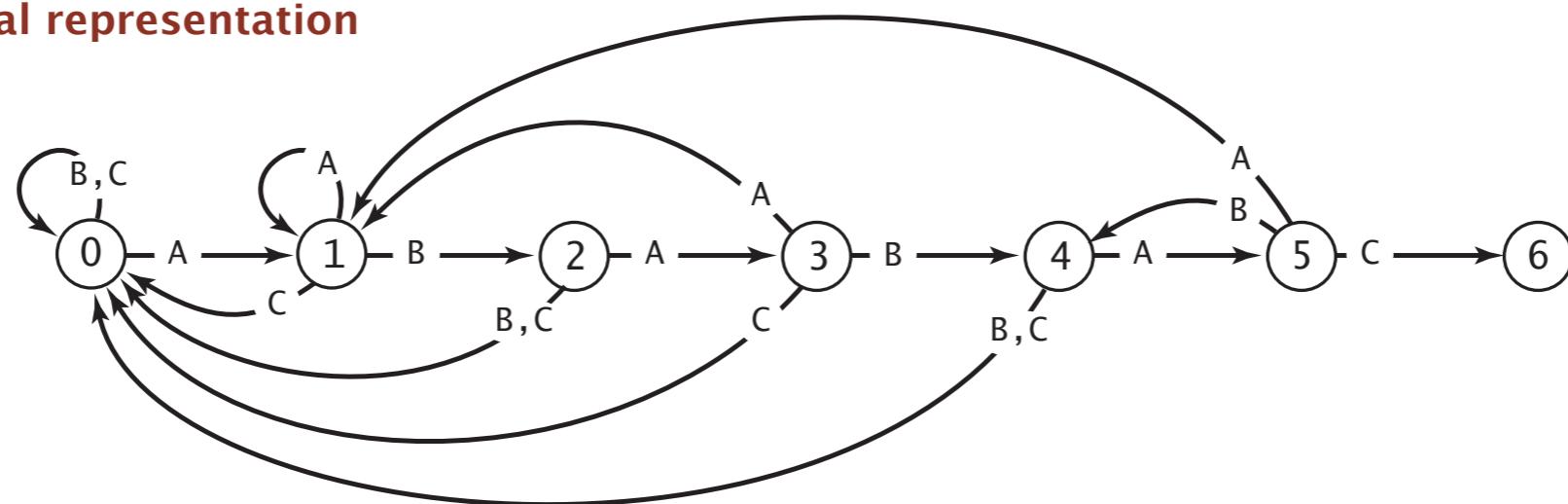
- Finite number of states (including start and halt).
- Exactly one transition for each char in alphabet.
- Accept if sequence of transitions leads to halt state.

internal representation

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
dfa[] [j]	1	1	3	1	5	1
A	0	2	0	4	0	4
B	0	0	0	0	0	6
C	0	0	0	0	0	6

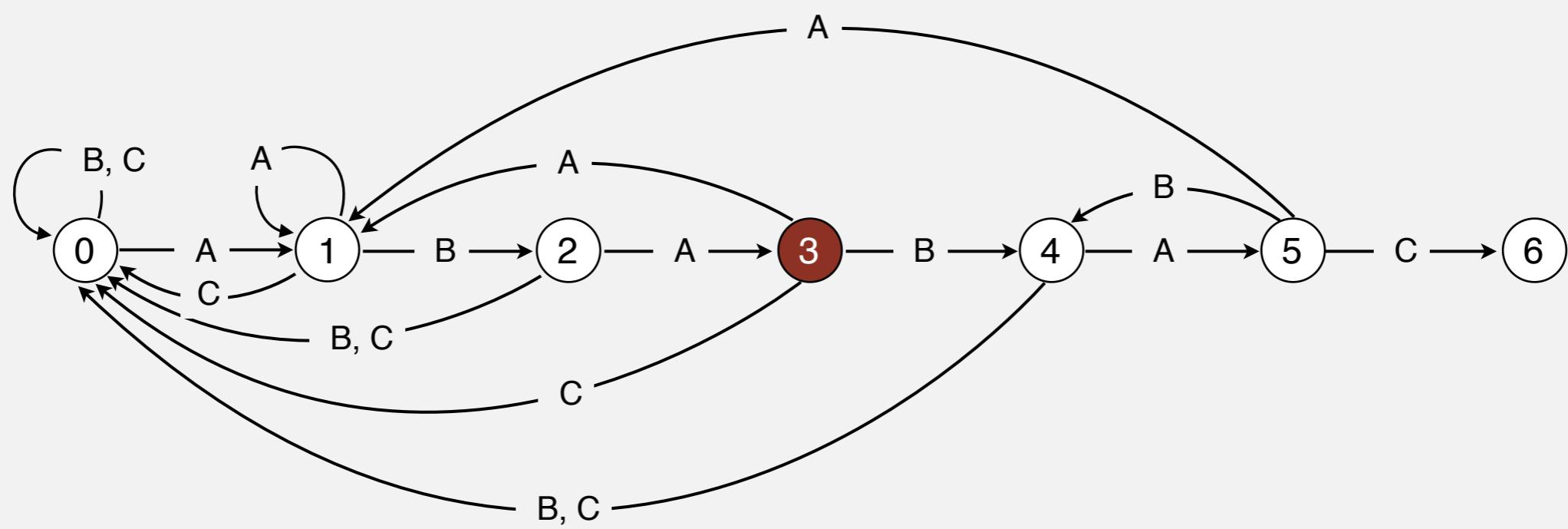
If in state j reading char c:
if j is 6 halt and accept
else move to state dfa[c][j]

graphical representation



Interpretation of Knuth-Morris-Pratt DFA

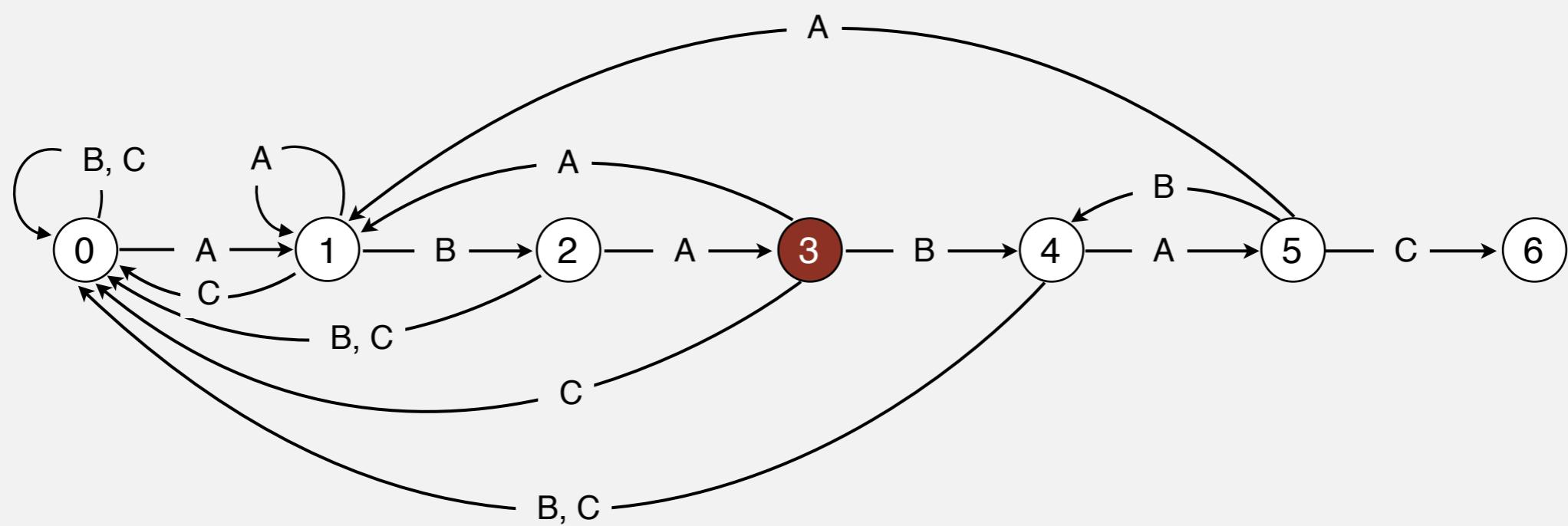
Q. What is interpretation of DFA state after reading in $\text{txt}[i]$?



Interpretation of Knuth-Morris-Pratt DFA

Q. What is interpretation of DFA state after reading in $\text{txt}[i]$?

A. State = number of characters in pattern that have been matched.

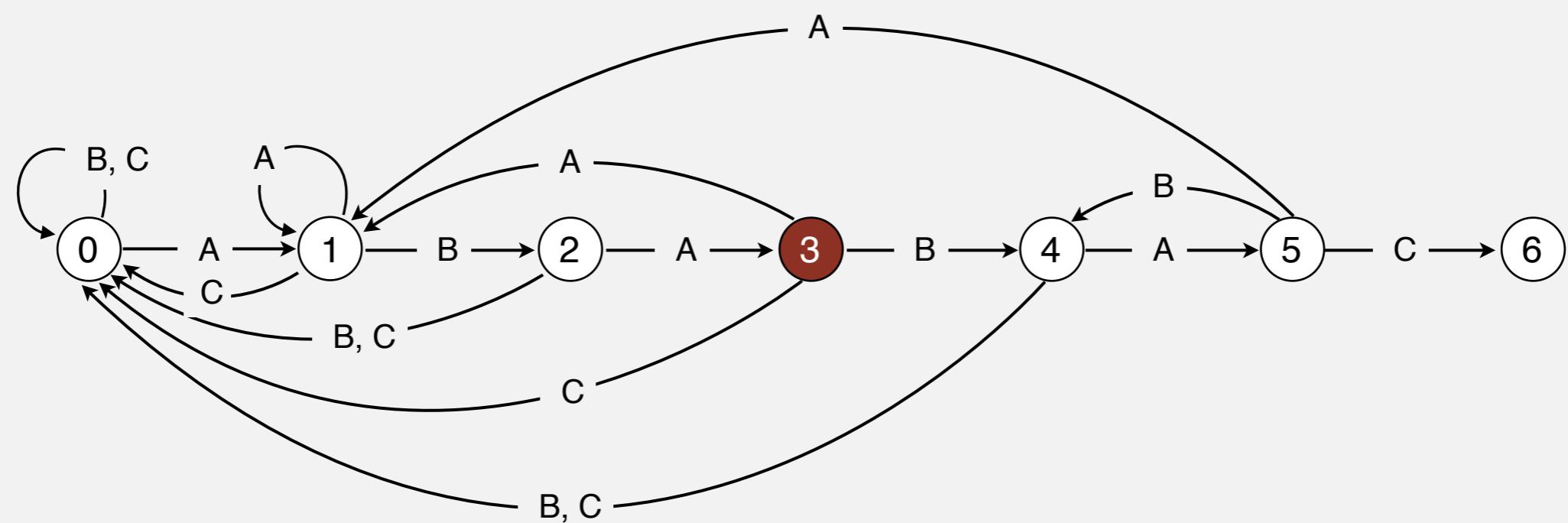


Interpretation of Knuth-Morris-Pratt DFA

Q. What is interpretation of DFA state after reading in $\text{txt}[i]$?

A. State = number of characters in pattern that have been matched.

Ex. DFA is in state 3 after reading in $\text{txt}[0..6]$.



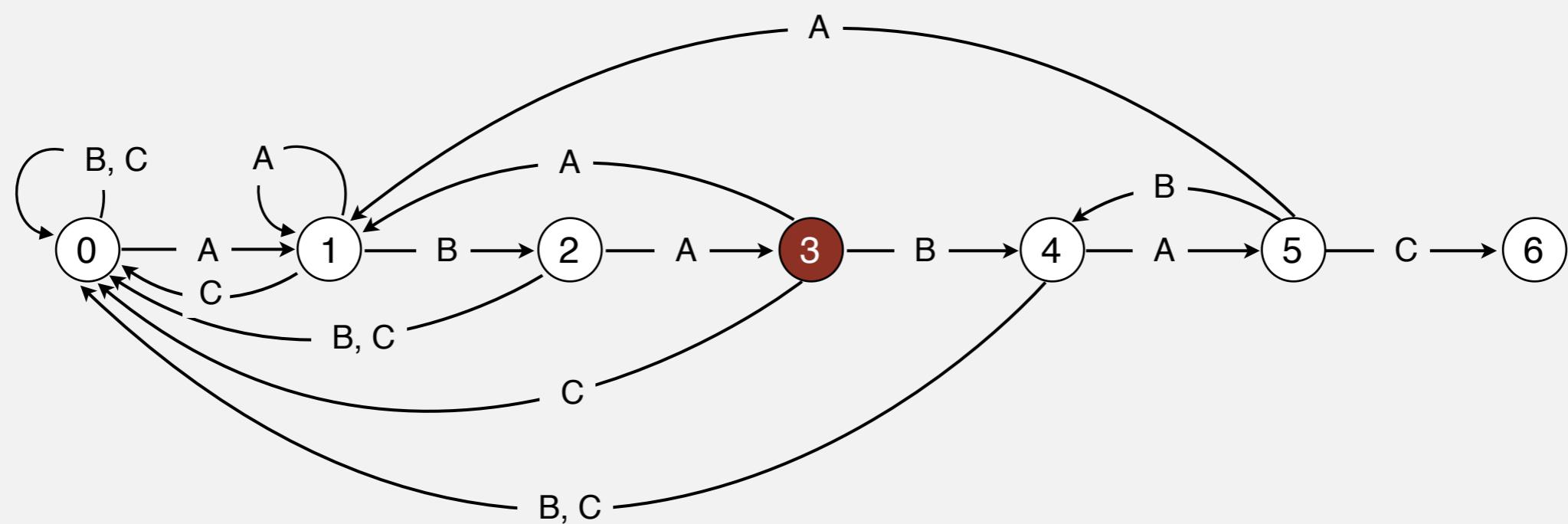
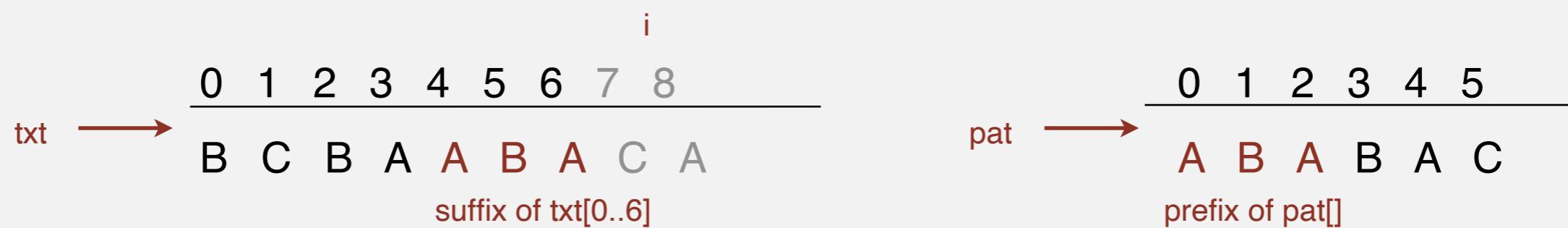
Interpretation of Knuth-Morris-Pratt DFA

Q. What is interpretation of DFA state after reading in $\text{txt}[i]$?

A. State = number of characters in pattern that have been matched.

length of longest prefix of $\text{pat}[]$
that is a suffix of $\text{txt}[0..i]$

Ex. DFA is in state 3 after reading in $\text{txt}[0..6]$.

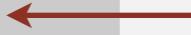


Knuth-Morris-Pratt substring search: Java implementation

Key differences from brute-force implementation.

- Need to precompute `dfa[][]` from pattern.
- Text pointer `i` never decrements.

```
public int search(String txt)
{
    int i, j, N = txt.length();
    for (i = 0, j = 0; i < N && j < M; i++)
        j = dfa[txt.charAt(i)][j];
        if (j == M) return i - M;
        else      return N;
}
```



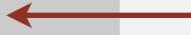
← no backup

Knuth-Morris-Pratt substring search: Java implementation

Key differences from brute-force implementation.

- Need to precompute `dfa[][]` from pattern.
- Text pointer `i` never decrements.

```
public int search(String txt)
{
    int i, j, N = txt.length();
    for (i = 0, j = 0; i < N && j < M; i++)
        j = dfa[txt.charAt(i)][j];
        if (j == M) return i - M;
        else      return N;
}
```



← no backup

Running time.

Knuth-Morris-Pratt substring search: Java implementation

Key differences from brute-force implementation.

- Need to precompute $\text{dfa}[][]$ from pattern.
- Text pointer i never decrements.

```
public int search(String txt)
{
    int i, j, N = txt.length();
    for (i = 0, j = 0; i < N && j < M; i++)
        j = dfa[txt.charAt(i)][j];
        if (j == M) return i - M;
        else      return N;
}
```



← no backup

Running time.

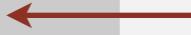
- Simulate DFA on text: at most N character accesses.

Knuth-Morris-Pratt substring search: Java implementation

Key differences from brute-force implementation.

- Need to precompute $\text{dfa}[][]$ from pattern.
- Text pointer i never decrements.

```
public int search(String txt)
{
    int i, j, N = txt.length();
    for (i = 0, j = 0; i < N && j < M; i++)
        j = dfa[txt.charAt(i)][j];
        if (j == M) return i - M;
        else      return N;
}
```



← no backup

Running time.

- Simulate DFA on text: at most N character accesses.
- Build DFA: how to do efficiently? [warning: tricky algorithm ahead]

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

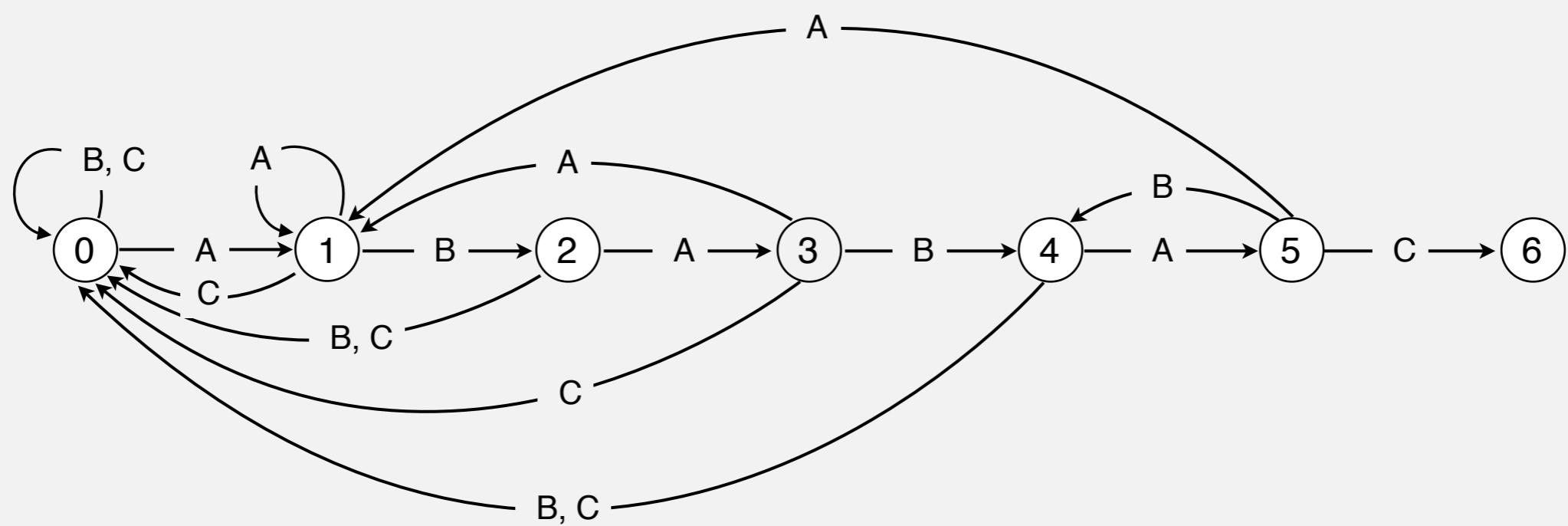
5.3 KNUTH-MORRIS-PRATT

- ▶ DFA simulation
- ▶ DFA construction
- ▶ DFA construction in linear-time

Knuth-Morris-Pratt demo: DFA simulation

A A B A C A A B A B A C A A

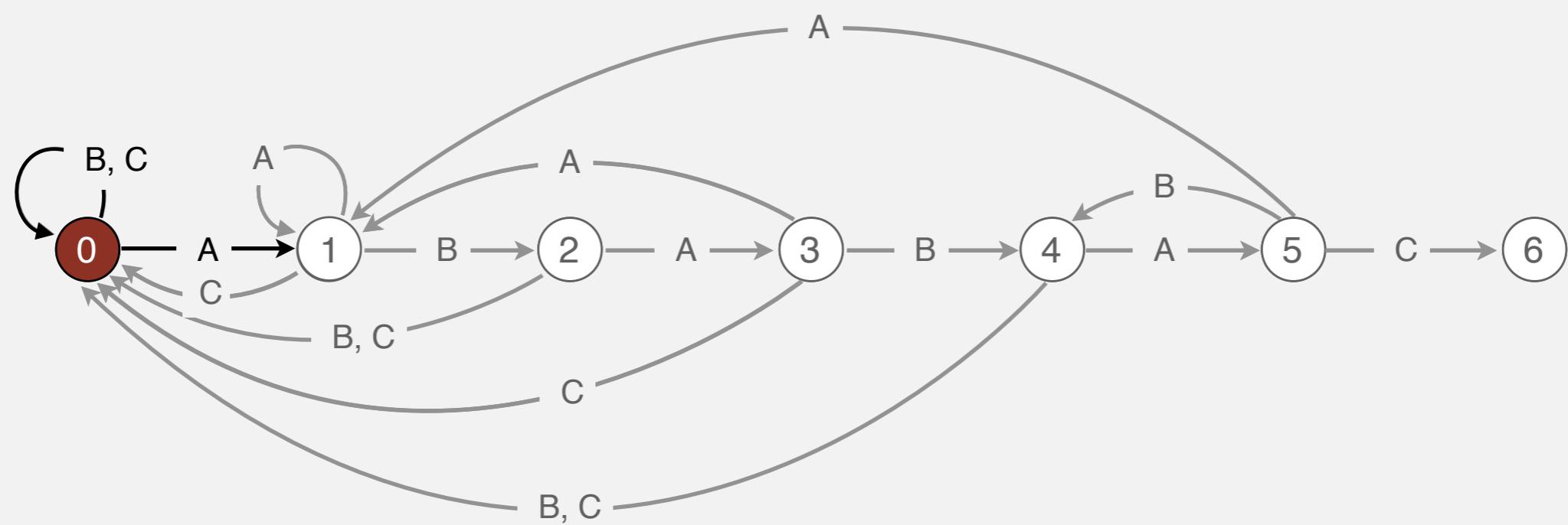
	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
dfa[][][j]	A	1	1	3	1	5
B	0	2	0	4	0	4
C	0	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

A A B A C A A B A B A C A A

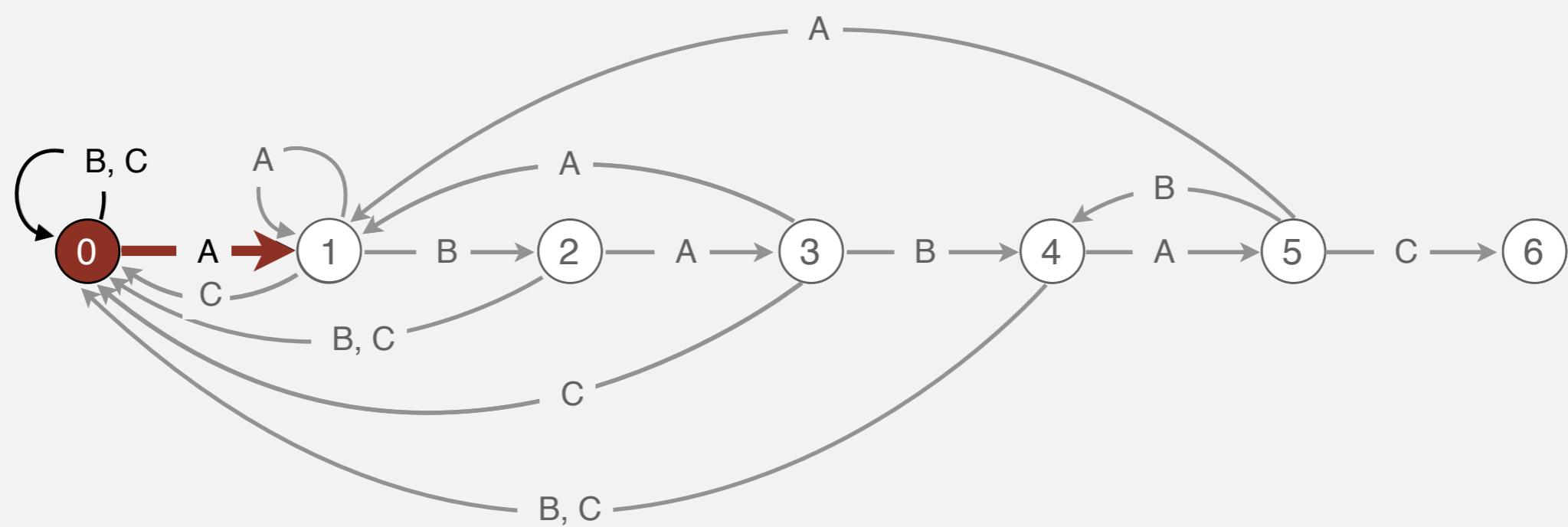
0	1	2	3	4	5	
pat.charAt(j)	A	B	A	B	A	C
dfa[][][j]	A	1	3	1	5	1
B	0	2	0	4	0	4
C	0	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

A A B A C A A B A B A C A A

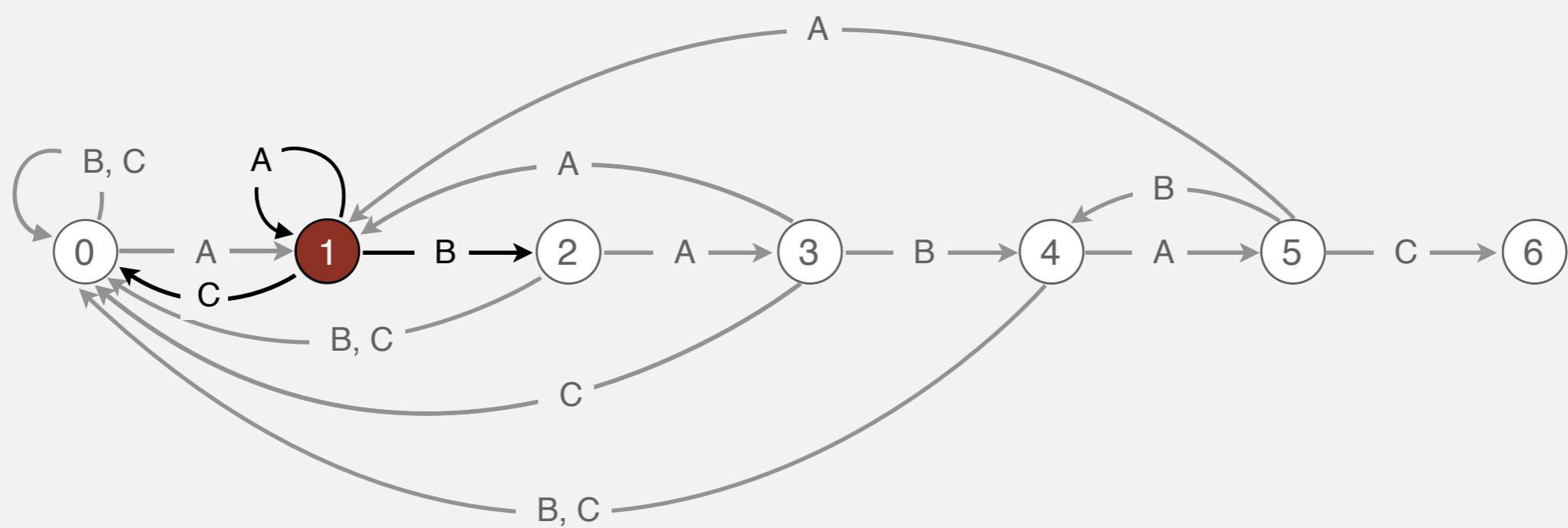
0	1	2	3	4	5	
pat.charAt(j)	A	B	A	B	A	C
dfa[][][j]	A	1	1	3	1	5
	B	0	2	0	4	0
	C	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

A A B A C A A B A B A C A A

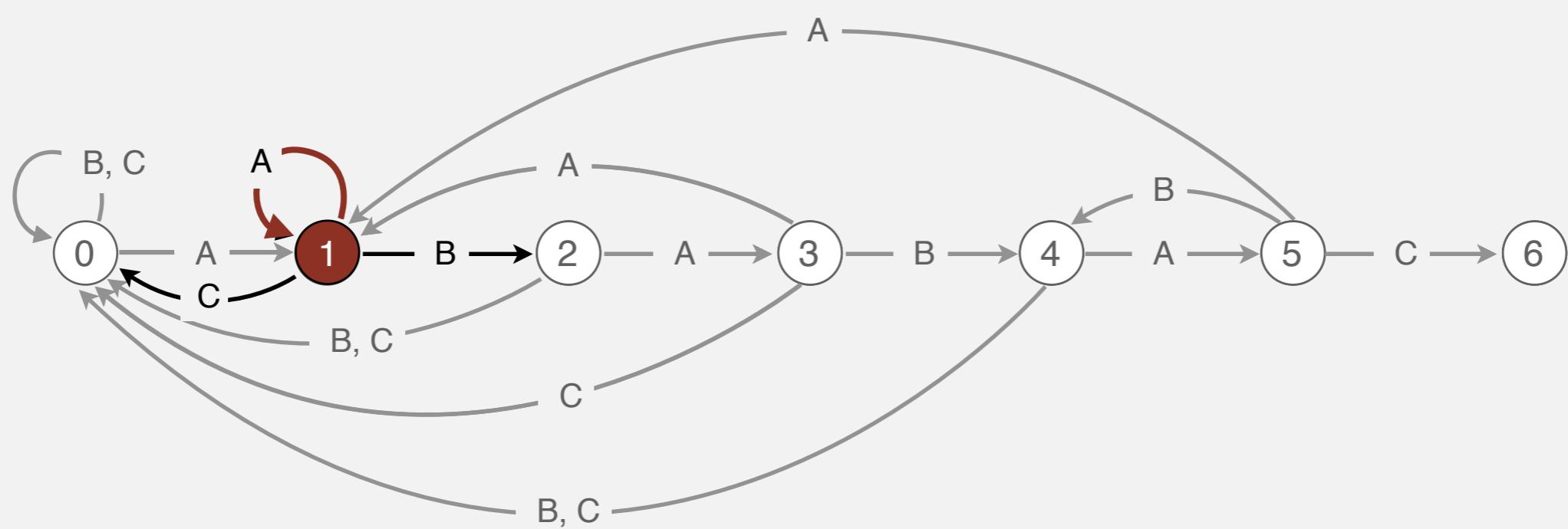
		0	1	2	3	4	5
pat.charAt(j)	dfa[][][j]	A	B	A	B	A	C
		1	1	3	1	5	1
A	0	2	0	4	0	4	
B	0	0	0	0	0	0	
C	0	0	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

A A B A C A A B A B A C A A

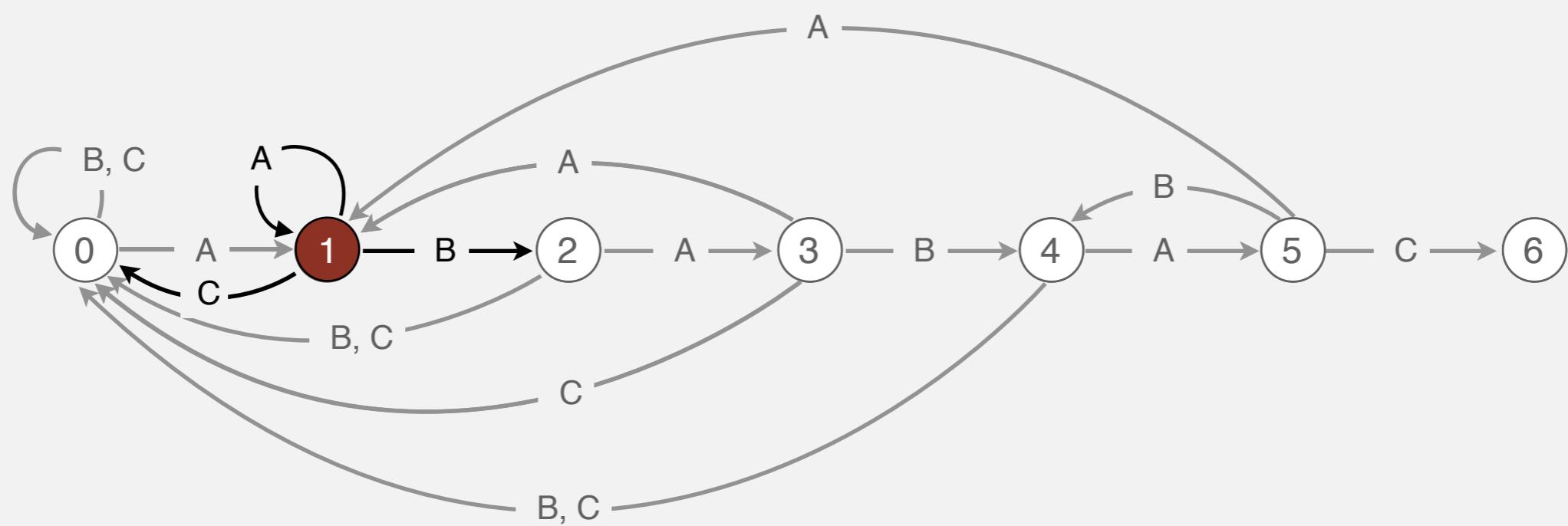
		0	1	2	3	4	5
pat.charAt(j)	dfa[][][j]	A	B	A	B	A	C
		1	1	3	1	5	1
A	0	2	0	4	0	4	
B	0	0	0	0	0	0	
C	0	0	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

A A B A C A A B A B A C A A

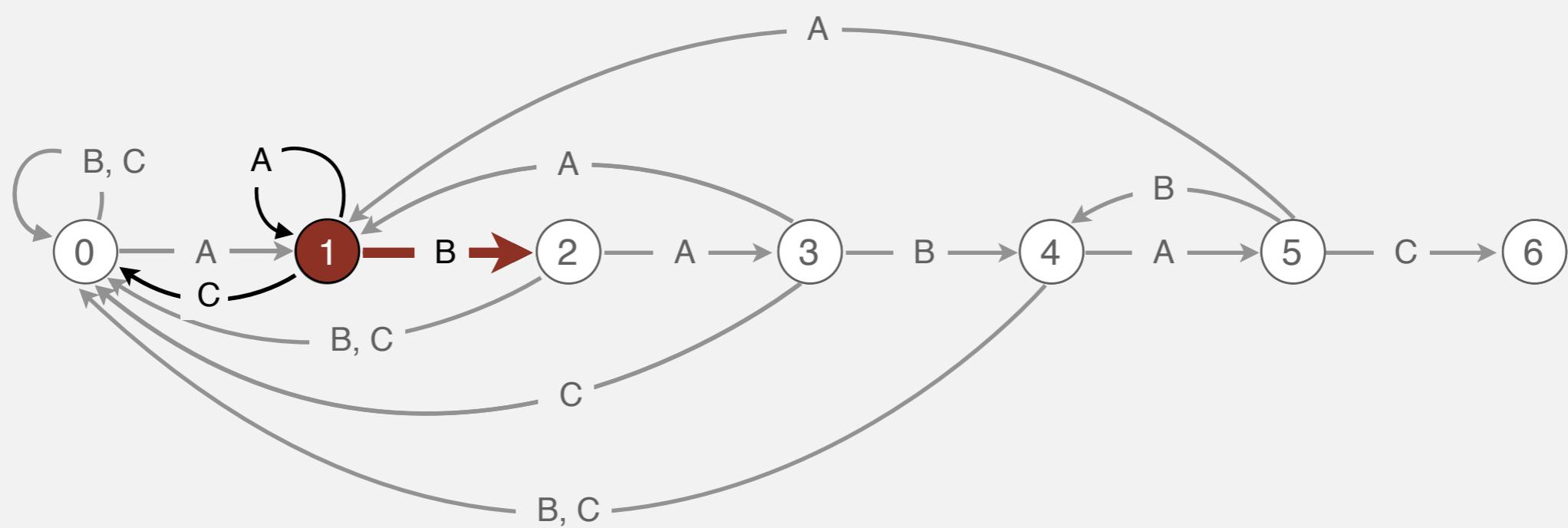
		0	1	2	3	4	5
pat.charAt(j)	A	A	B	A	B	A	C
	B	1	1	3	1	5	1
dfa[][][j]	A	0	2	0	4	0	4
C	0	0	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

A A B A C A A B A B A C A A

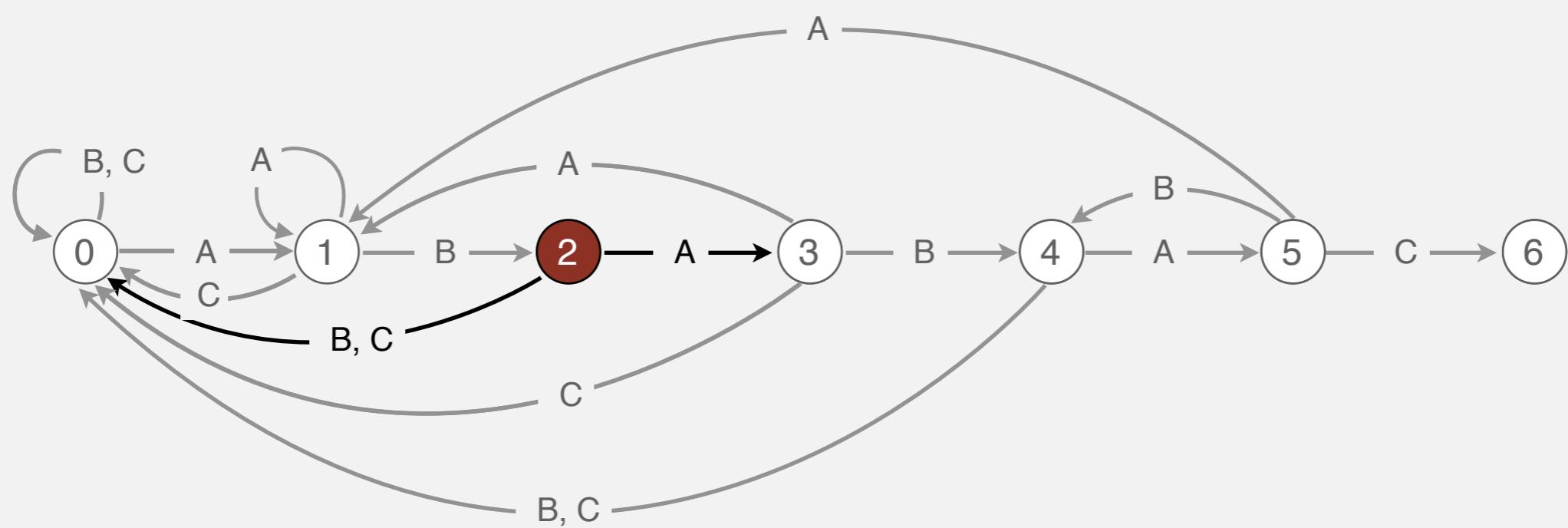
		0	1	2	3	4	5
pat.charAt(j)	A	A	B	A	B	A	C
	B	1	1	3	1	5	1
dfa[][][j]	A	0	2	0	4	0	4
B	0	0	0	0	0	0	6
C	0	0	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

A A B A C A A B A B A C A A

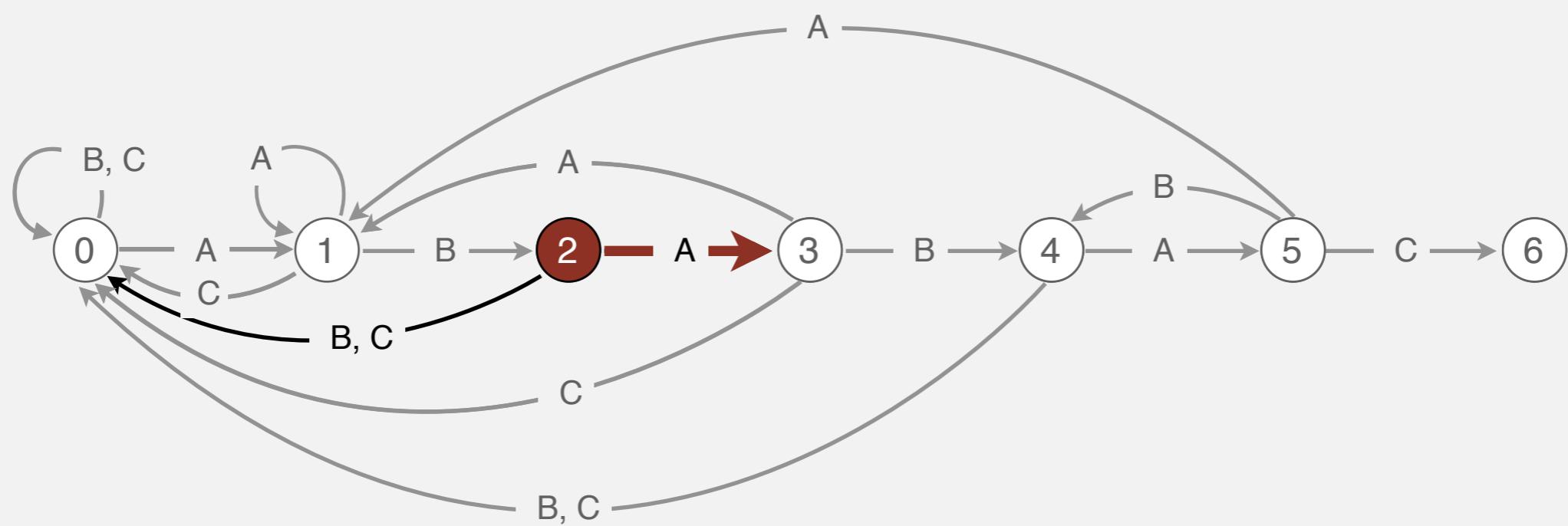
		0	1	2	3	4	5
pat.charAt(j)	dfa[][][j]	A	B	A	B	A	C
		1	1	3	1	5	1
A	0	2	0	4	0	4	
B	0	0	0	0	0	6	
C	0	0	0	0	0	6	



Knuth-Morris-Pratt demo: DFA simulation

A A B A C A A B A B A C A A

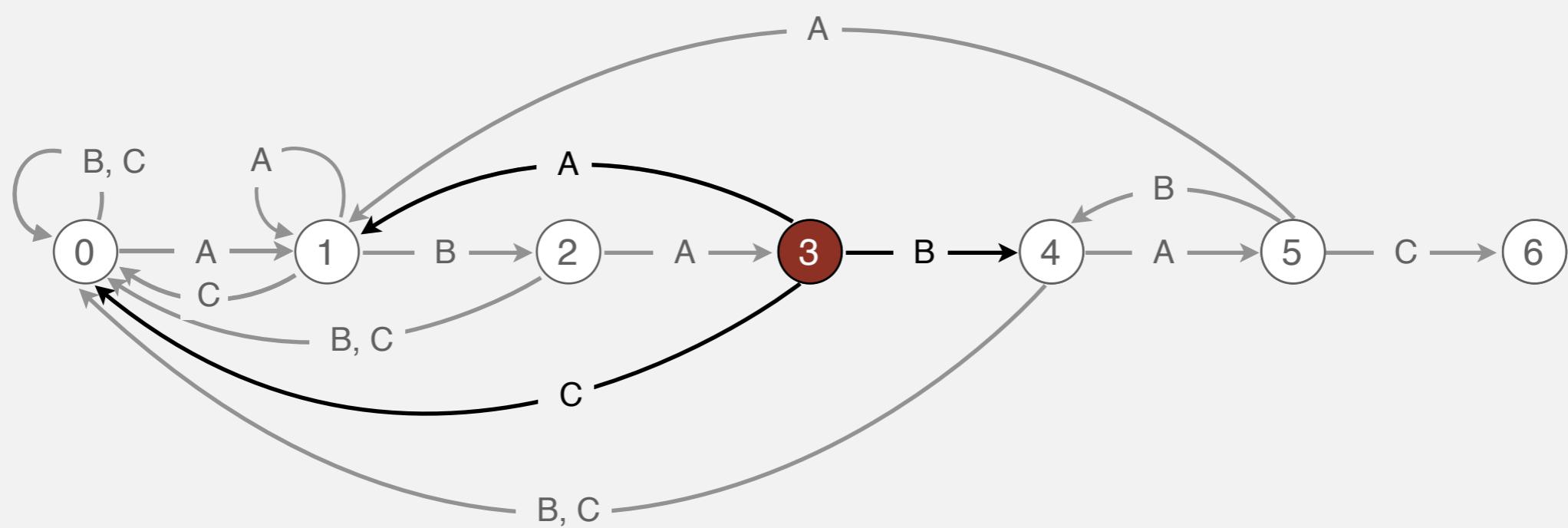
		0	1	2	3	4	5
pat.charAt(j)	dfa[][][j]	A	B	A	B	A	C
		1	1	3	1	5	1
A	0	2	0	4	0	4	
B	0	0	0	0	0	6	
C	0	0	0	0	0	6	



Knuth-Morris-Pratt demo: DFA simulation

A A B A C A A B A B A C A A
↑

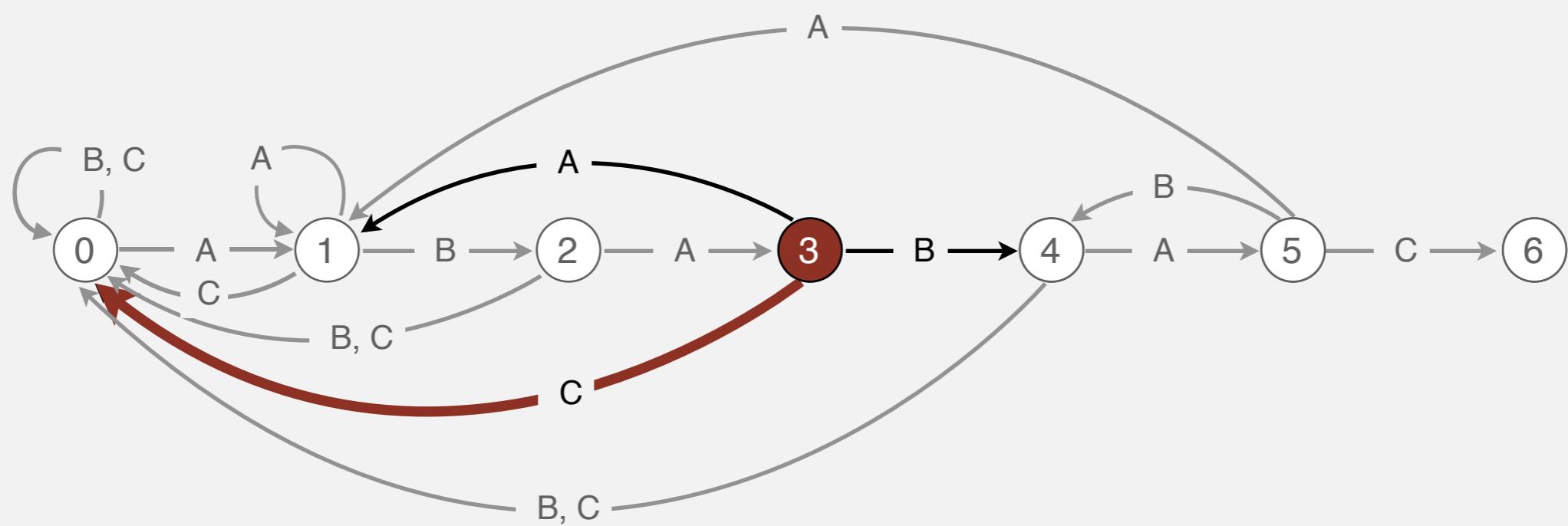
		0	1	2	3	4	5
pat.charAt(j)	dfa[][][j]	A	B	A	B	A	C
		1	1	3	1	5	1
A	0	2	0	4	0	4	
B	0	0	0	0	0	6	
C	0	0	0	0	0	6	



Knuth-Morris-Pratt demo: DFA simulation

A A B A C A A B A B A C A A
↑

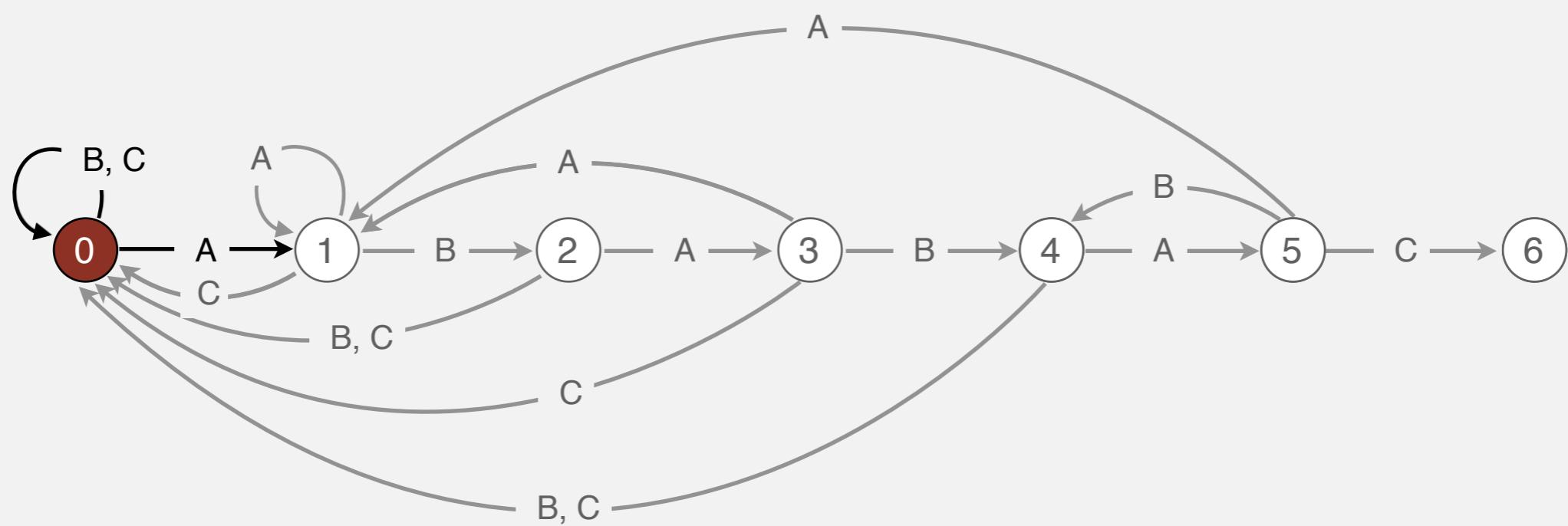
		0	1	2	3	4	5
pat.charAt(j)	dfa[][][j]	A	B	A	B	A	C
		1	1	3	1	5	1
A	0	2	0	4	0	4	
B	0	0	0	0	0	6	
C	0	0	0	0	0	6	



Knuth-Morris-Pratt demo: DFA simulation

A A B A C A A B A B A C A A

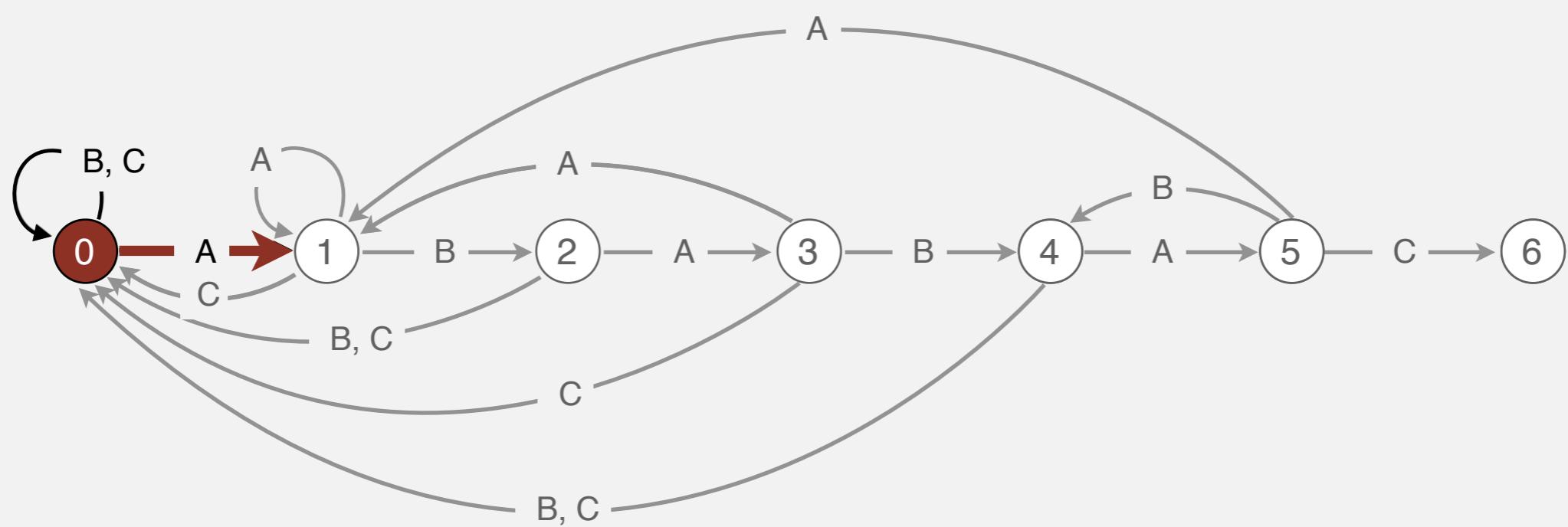
0	1	2	3	4	5	
pat.charAt(j)	A	B	A	B	A	C
dfa[][][j]	A	1	3	1	5	1
	B	0	0	4	0	4
	C	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

A A B A C A A B A B A C A A

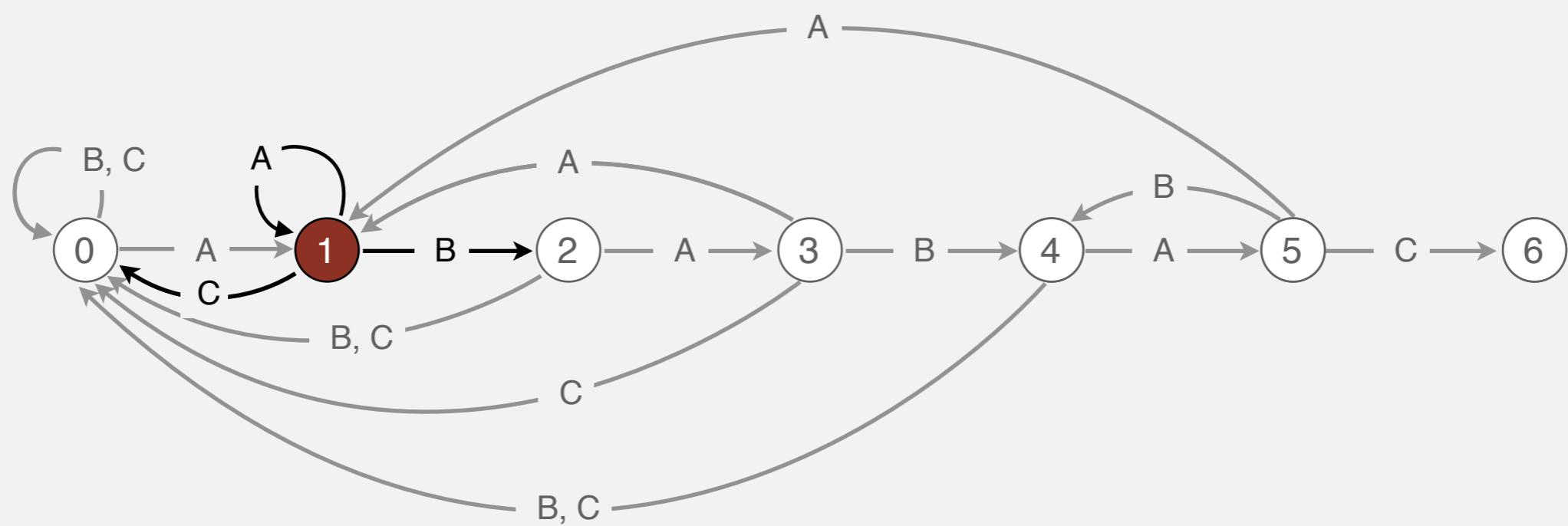
0	1	2	3	4	5	
pat.charAt(j)	A	B	A	B	A	C
dfa[][][j]	A	1	3	1	5	1
B	0	2	0	4	0	4
C	0	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

A A B A C **A** A B A B A C A A
↑

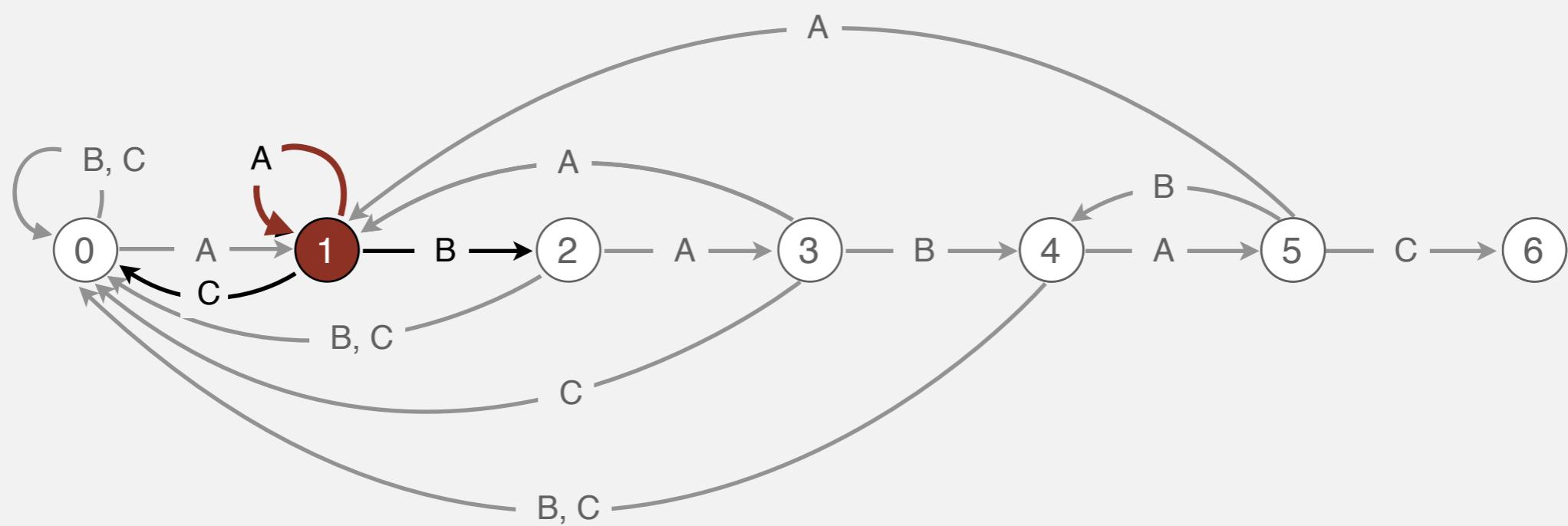
		0	1	2	3	4	5
pat.charAt(j)	A	A	B	A	B	A	C
	B	1	1	3	1	5	1
dfa[][]j	C	0	2	0	4	0	4
		0	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

A A B A C **A** A B A B A C A A
↑

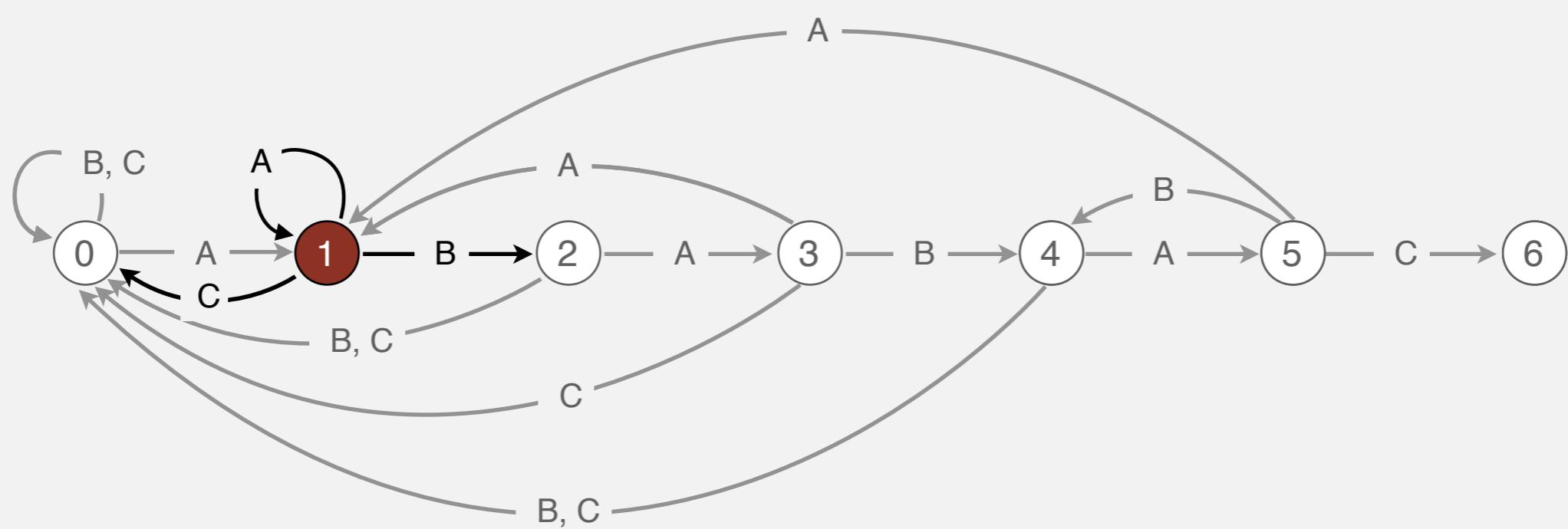
		0	1	2	3	4	5
pat.charAt(j)	dfa[][][j]	A	B	A	B	A	C
		1	1	3	1	5	1
A	0	2	0	4	0	4	
B	0	0	0	0	0	0	
C	0	0	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

A A B A C A **A** B A B A C A A
↑

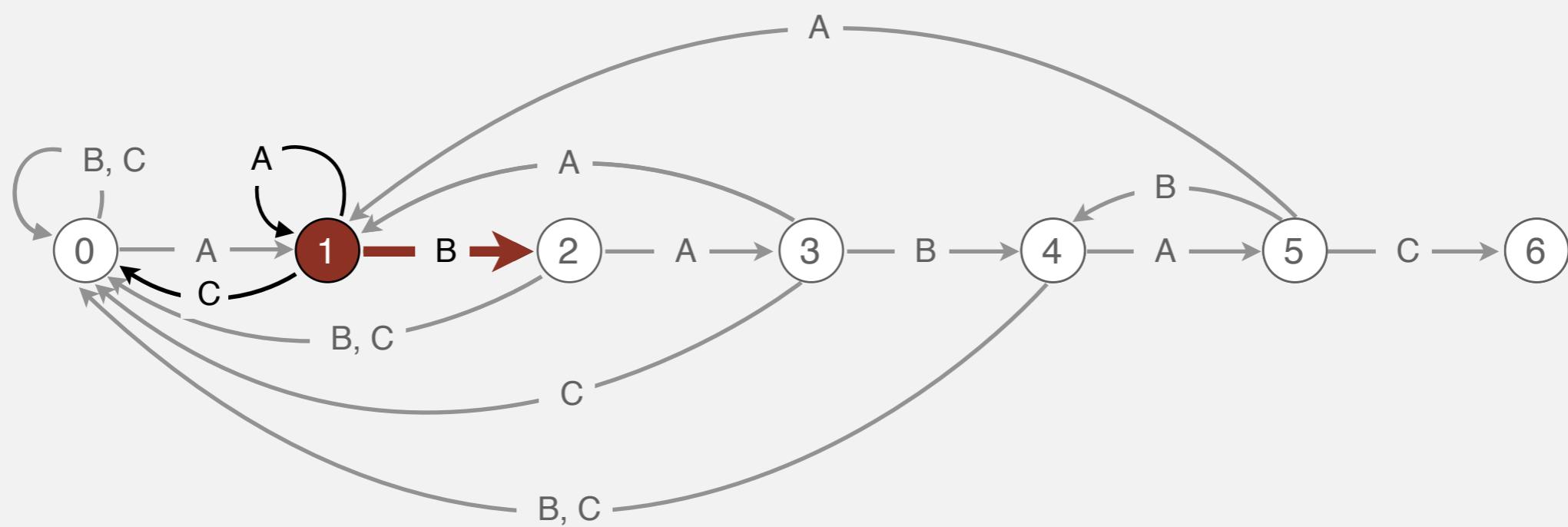
		0	1	2	3	4	5
pat.charAt(j)	A	A	B	A	B	A	C
	B	1	1	3	1	5	1
dfa[][]j	C	0	2	0	4	0	4
	0	0	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

A A B A C A A B A B A C A A
↑

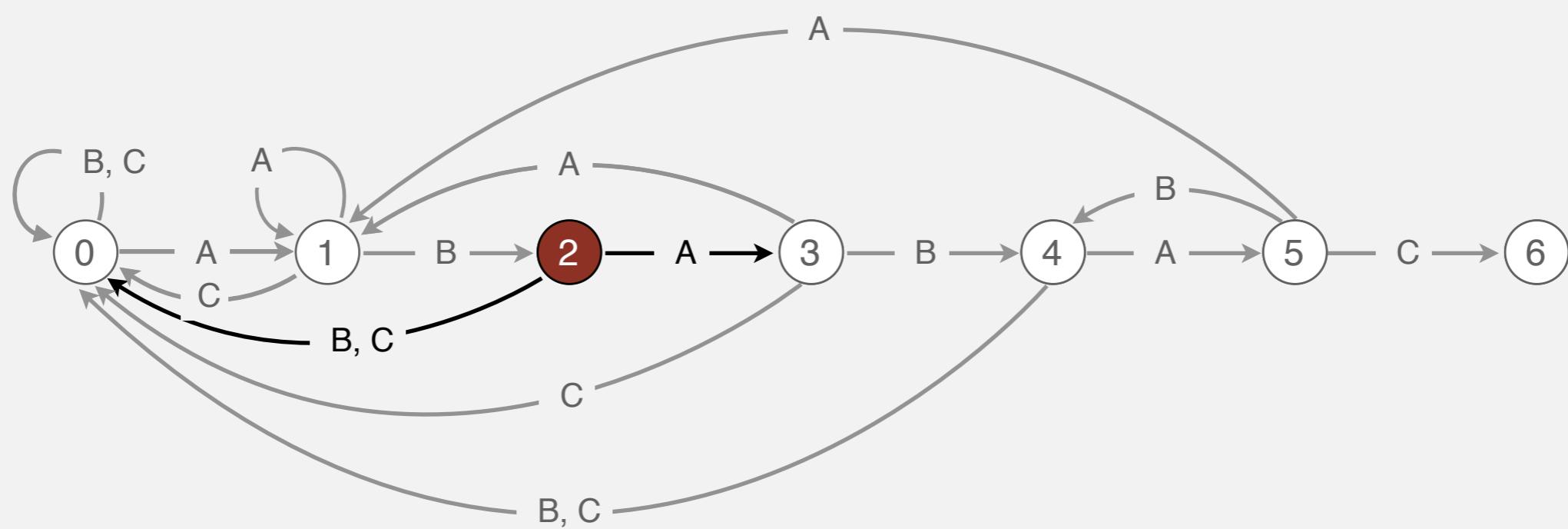
		0	1	2	3	4	5
pat.charAt(j)	A	A	B	A	B	A	C
	B	1	1	3	1	5	1
dfa[][]j	A	0	2	0	4	0	4
	B	0	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

A A B A C A **A** B A B A C A A
↑

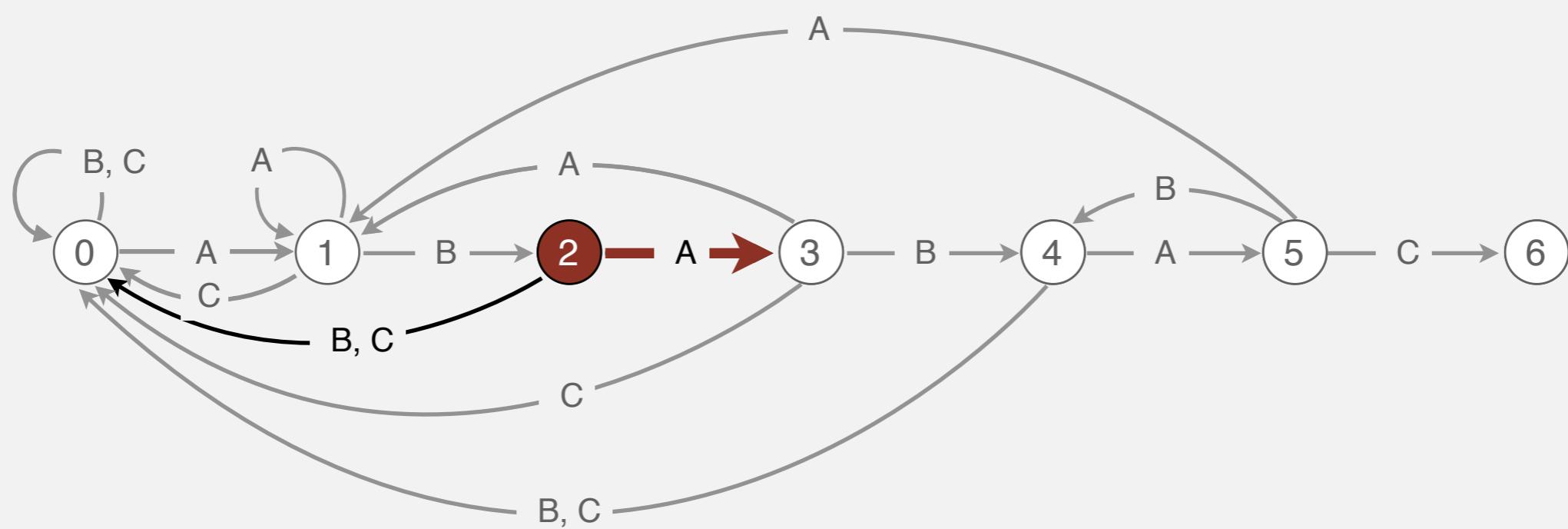
		0	1	2	3	4	5
pat.charAt(j)	A	A	B	A	B	A	C
	B	1	1	3	1	5	1
dfa[][]j	C	0	2	0	4	0	4
		0	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

A A B A C A **A** B A B A C A A
↑

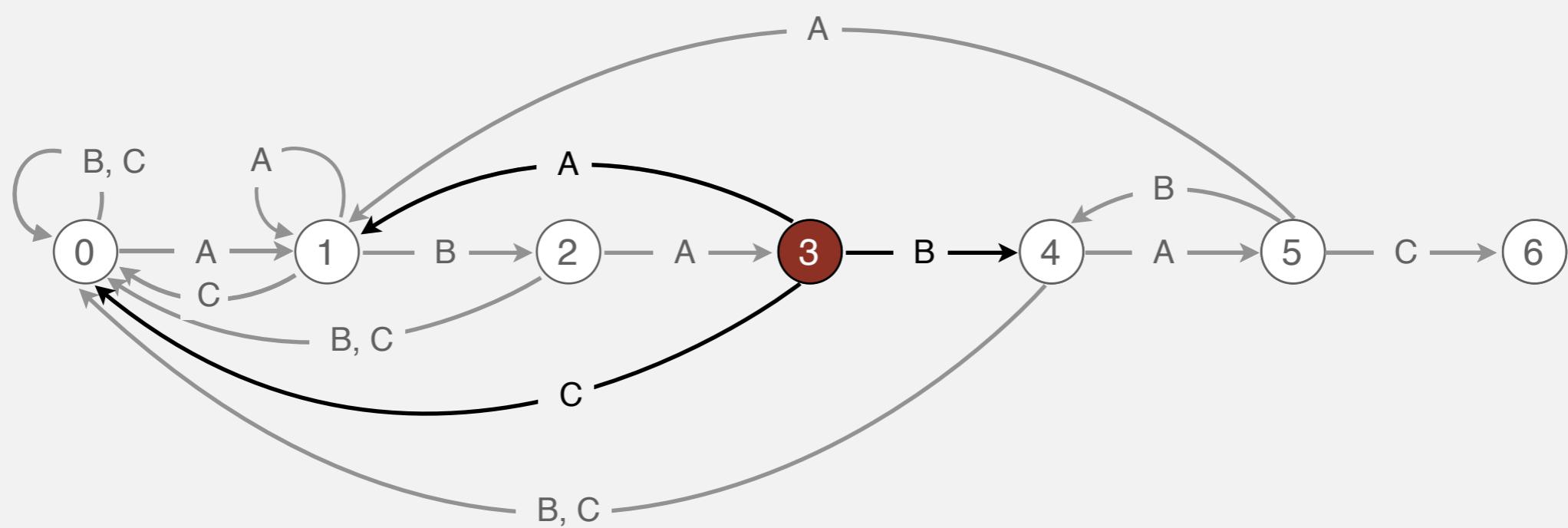
		0	1	2	3	4	5
pat.charAt(j)	A	A	B	A	B	A	C
	B	1	1	3	1	5	1
dfa[][][j]	C	0	2	0	4	0	4
		0	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

A A B A C A **A** B A B A C A A
↑

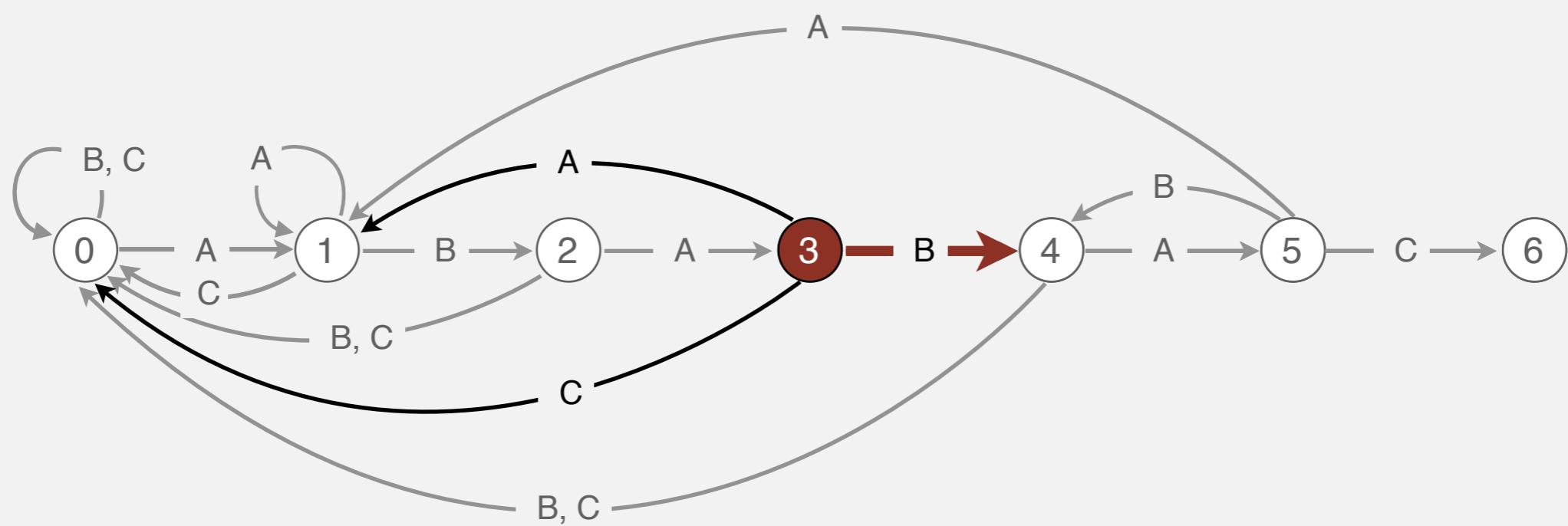
		0	1	2	3	4	5
pat.charAt(j)	A	A	B	A	B	A	C
	B	1	1	3	1	5	1
dfa[][]j	C	0	2	0	4	0	4
		0	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

A A B A C A **A** B A B A C A A
↑

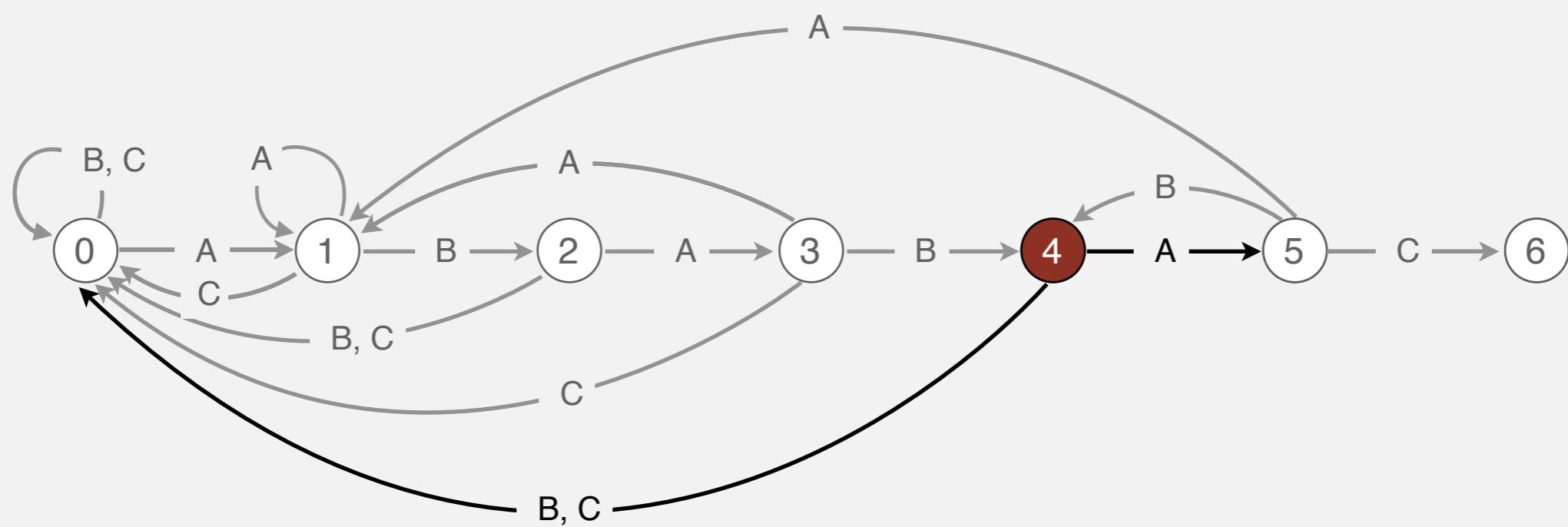
		0	1	2	3	4	5
pat.charAt(j)	A	A	B	A	B	A	C
	B	1	1	3	1	5	1
dfa[][]j	C	0	2	0	4	0	4
		0	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

A A B A C A **A** B A B A C A A
↑

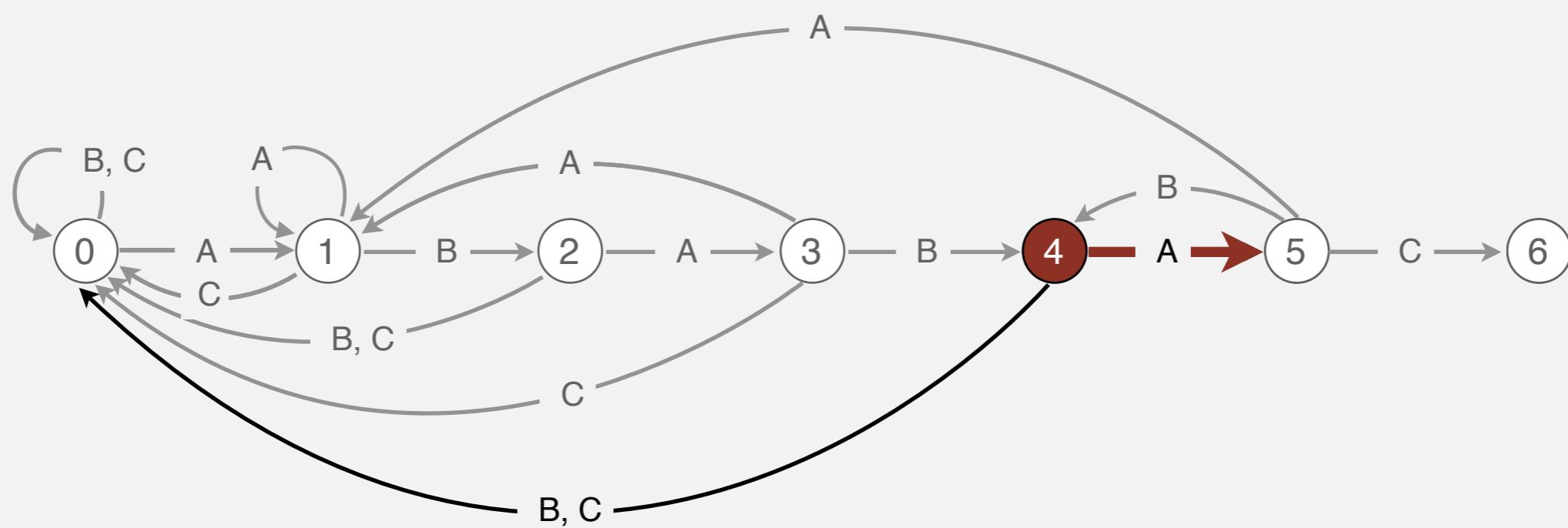
		0	1	2	3	4	5
pat.charAt(j)	A	A	B	A	B	A	C
	B	1	1	3	1	5	1
dfa[][]j	C	0	2	0	4	0	4
		0	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

A A B A C A **A** B A B A C A A
↑

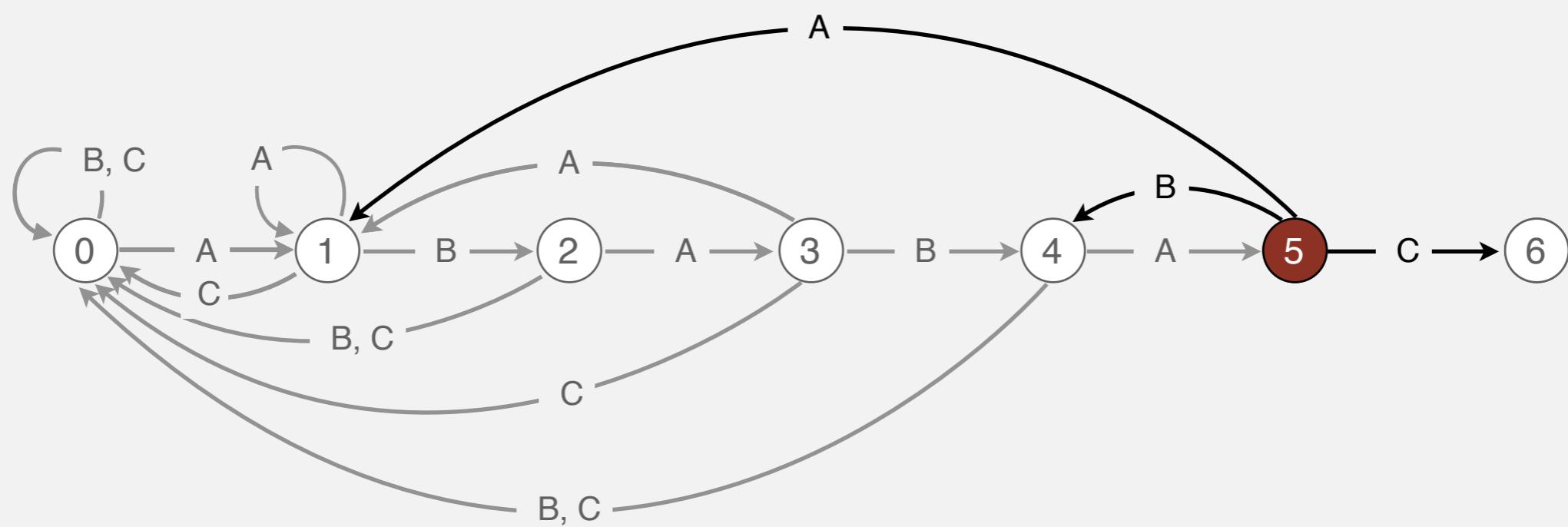
		0	1	2	3	4	5
pat.charAt(j)	A	A	B	A	B	A	C
	B	1	1	3	1	5	1
dfa[][][j]	C	0	2	0	4	0	4
		0	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

A A B A C A **A** B A B A C A A
↑

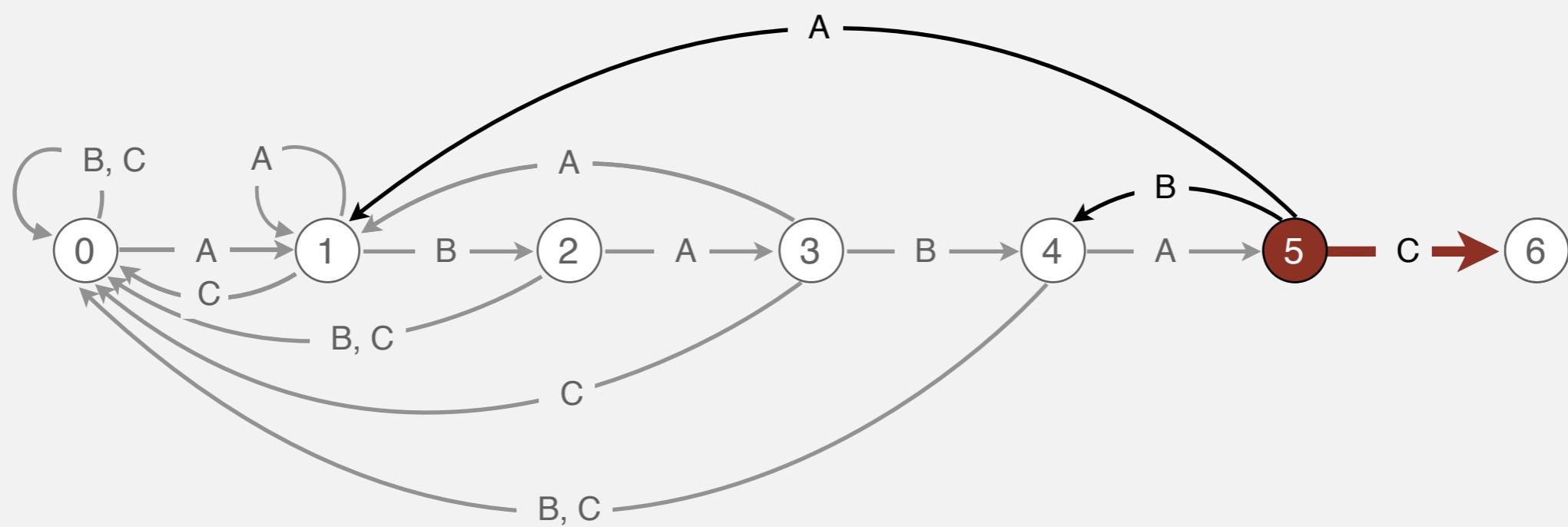
	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
dfa[][][j]	A	1	1	3	1	5
	B	0	2	0	4	0
	C	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

A A B A C A **A** B A B A C A A
↑

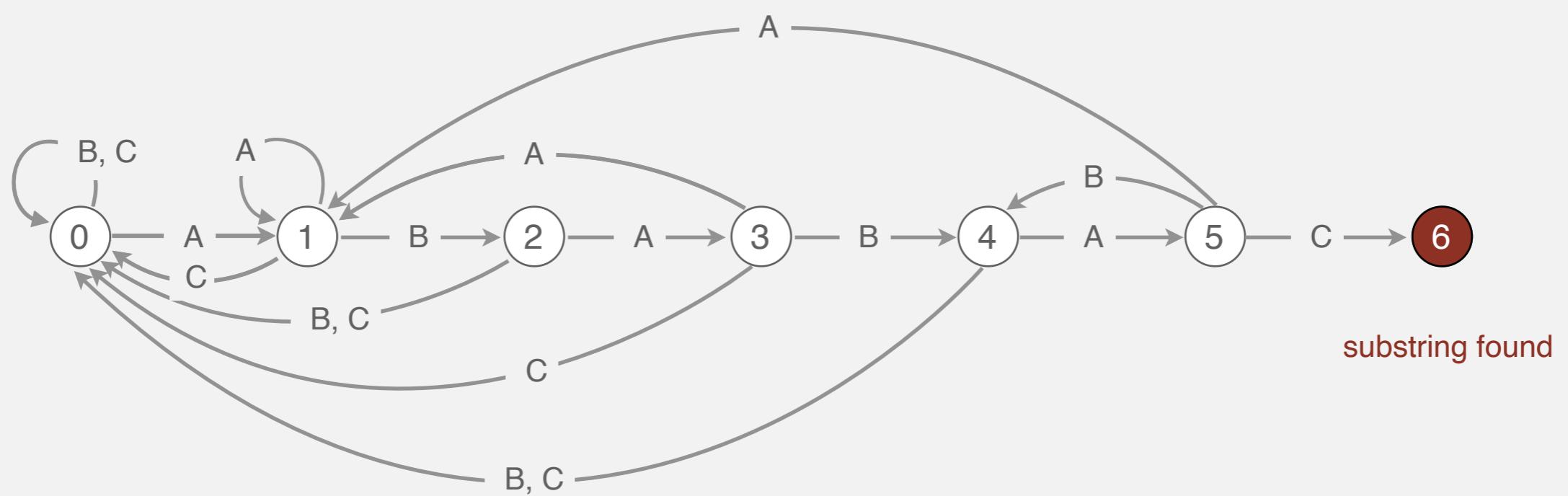
	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
dfa[][][j]	A	1	1	3	1	5
	B	0	2	0	4	0
	C	0	0	0	0	6



Knuth-Morris-Pratt demo: DFA simulation

A A B A C A **A** B A B A C A A
↑

		0	1	2	3	4	5
pat.charAt(j)	dfa[][][j]	A	B	A	B	A	C
		1	1	3	1	5	1
A	0	2	0	4	0	4	
B	0	0	0	0	0	6	
C	0	0	0	0	0	6	



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

5.3 KNUTH-MORRIS-PRATT

- ▶ DFA simulation
- ▶ DFA construction
- ▶ DFA construction in linear-time

Knuth-Morris-Pratt demo: DFA construction

Include one state for each character in pattern (plus accept state).

pat.charAt(j)	0	1	2	3	4	5
dfa[][]j]	A	B	A	B	A	C
A						
B						
C						

Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt demo: DFA construction

Include one state for each character in pattern (plus accept state).

pat.charAt(j)	0	1	2	3	4	5
dfa[][]j]	A	B	A	B	A	C
A						
B						
C						

Constructing the DFA for KMP substring search for A B A B A C



Knuth-Morris-Pratt demo: DFA construction

Match transition. If in state j and next char $c == \text{pat.charAt}(j)$, go to $j+1$.

↑
first j characters of pattern
have already been matched

↑
next char matches

↑
now first $j+1$ characters of
pattern have been matched

	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
dfa[][]j]	A	B				

Constructing the DFA for KMP substring search for A B A B A C



Knuth-Morris-Pratt demo: DFA construction

Match transition. If in state j and next char $c == \text{pat.charAt}(j)$, go to $j+1$.

↑
first j characters of pattern
have already been matched

↑
next char matches

↑
now first $j+1$ characters of
pattern have been matched

	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
dfa[][][j]	A	1	3	5		
	B		2	4		
	C				6	

Constructing the DFA for KMP substring search for A B A B A C



Knuth-Morris-Pratt demo: DFA construction

Mismatch transition: back up if $c \neq \text{pat.charAt}(j)$.

		0	1	2	3	4	5
		A	B	A	B	A	C
pat.charAt(j)	dfa[][][j]	1	3		5		
A			2		4		
B							
C							6

Constructing the DFA for KMP substring search for A B A B A C



Knuth-Morris-Pratt demo: DFA construction

Mismatch transition: back up if $c \neq \text{pat.charAt}(j)$.

		0	1	2	3	4	5
		A	B	A	B	A	C
pat.charAt(j)		A	1	3		5	
		B	0	2	4		
		C	0				6

Constructing the DFA for KMP substring search for A B A B A C



Knuth-Morris-Pratt demo: DFA construction

Mismatch transition: back up if $c \neq \text{pat.charAt}(j)$.

		0	1	2	3	4	5
		A	B	A	B	A	C
pat.charAt(j)	A	1		3		5	
	B	0	2		4		
	C	0					6

Constructing the DFA for KMP substring search for A B A B A C

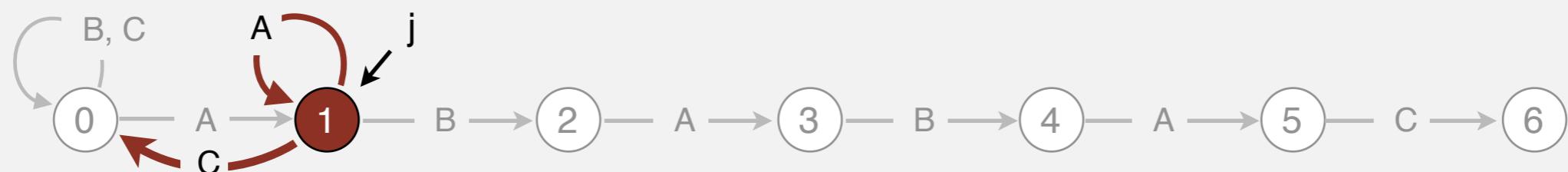


Knuth-Morris-Pratt demo: DFA construction

Mismatch transition: back up if $c \neq \text{pat.charAt}(j)$.

		0	1	2	3	4	5
		A	B	A	B	A	C
pat.charAt(j)	A	1	1	3		5	
	B	0	2		4		
dfa[][][j]	C	0	0				6

Constructing the DFA for KMP substring search for A B A B A C

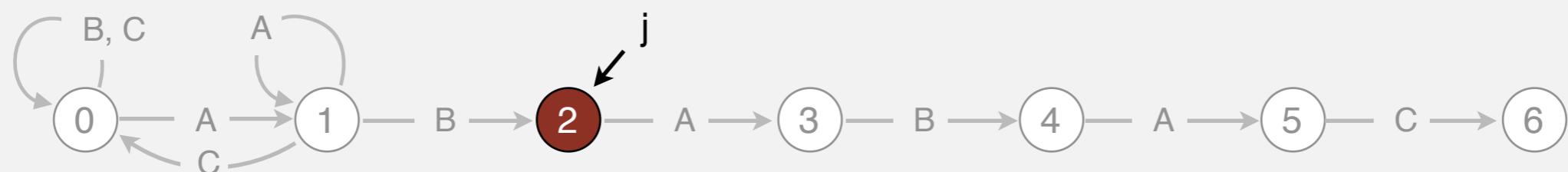


Knuth-Morris-Pratt demo: DFA construction

Mismatch transition: back up if $c \neq \text{pat.charAt}(j)$.

		0	1	2	3	4	5
		A	B	A	B	A	C
pat.charAt(j)	A	1	1	3		5	
	B	0	2		4		
	C	0	0				6

Constructing the DFA for KMP substring search for A B A B A C



Knuth-Morris-Pratt demo: DFA construction

Mismatch transition: back up if $c \neq \text{pat.charAt}(j)$.

		0	1	2	3	4	5
pat.charAt(j)	A	A	B	A	B	A	C
	B	1	1	3		5	
dfa[][][j]	B	0	2	0	4		
C	0	0	0			6	

Constructing the DFA for KMP substring search for A B A B A C

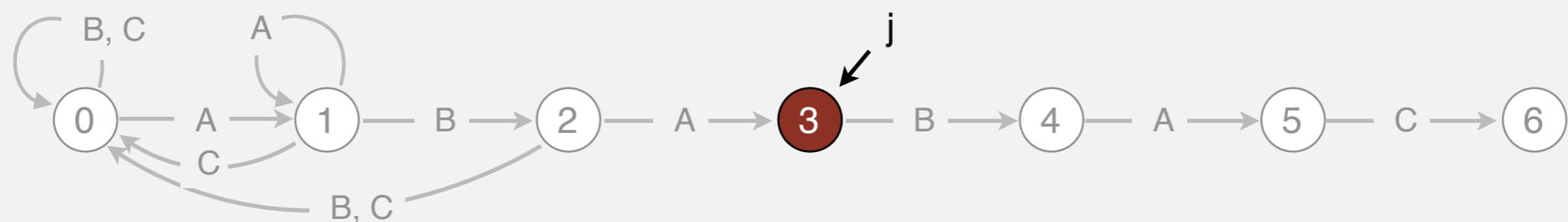


Knuth-Morris-Pratt demo: DFA construction

Mismatch transition: back up if $c \neq \text{pat.charAt}(j)$.

		0	1	2	3	4	5
		A	B	A	B	A	C
pat.charAt(j)	A	1	1	3		5	
	B	0	2	0	4		
dfa[][][j]	C	0	0	0			6

Constructing the DFA for KMP substring search for A B A B A C

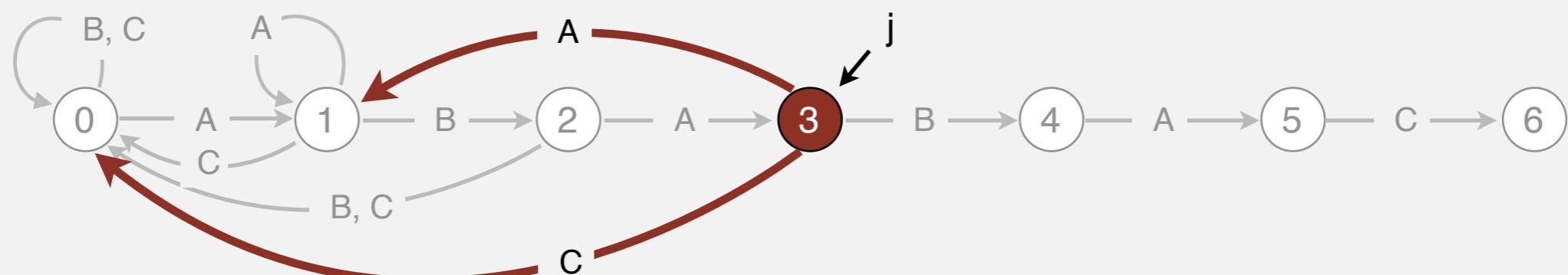


Knuth-Morris-Pratt demo: DFA construction

Mismatch transition: back up if $c \neq \text{pat.charAt}(j)$.

		0	1	2	3	4	5
		A	B	A	B	A	C
pat.charAt(j)	A	1	1	3	1	5	
	B	0	2	0	4		
dfa[][][j]	C	0	0	0	0		6

Constructing the DFA for KMP substring search for A B A B A C

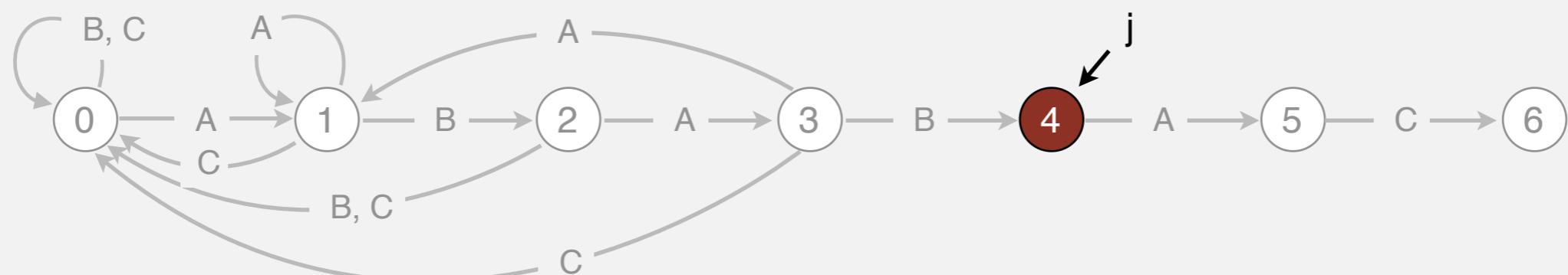


Knuth-Morris-Pratt demo: DFA construction

Mismatch transition: back up if $c \neq \text{pat.charAt}(j)$.

		0	1	2	3	4	5
		A	B	A	B	A	C
pat.charAt(j)	A	1	1	3	1	5	
	B	0	2	0	4		
dfa[][][j]	C	0	0	0	0		6

Constructing the DFA for KMP substring search for A B A B A C

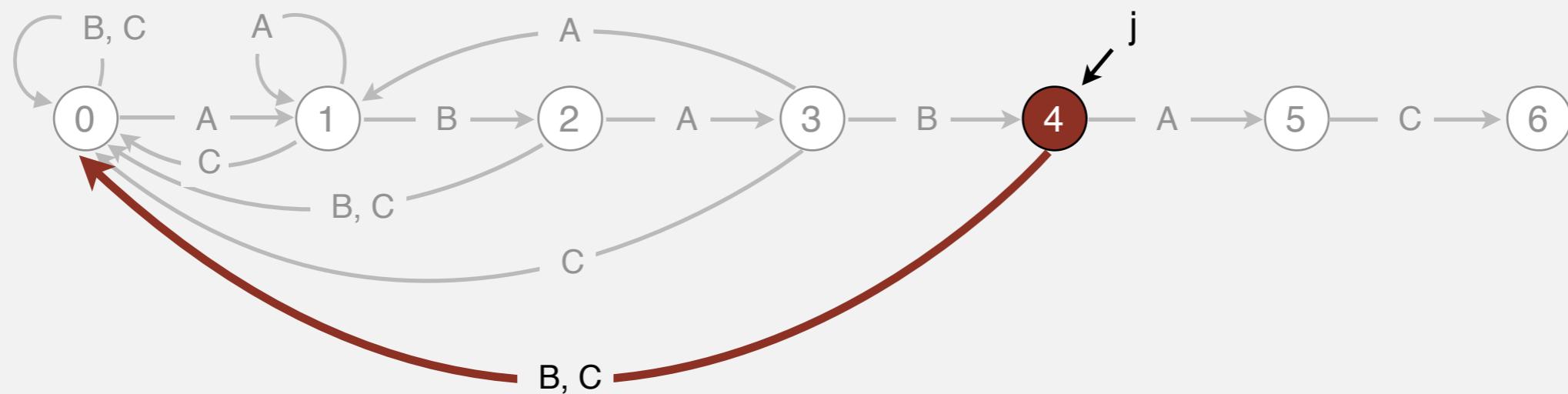


Knuth-Morris-Pratt demo: DFA construction

Mismatch transition: back up if $c \neq \text{pat.charAt}(j)$.

		0	1	2	3	4	5
		A	B	A	B	A	C
pat.charAt(j)	A	1	1	3	1	5	
	B	0	2	0	4	0	
C	0	0	0	0	0	0	6

Constructing the DFA for KMP substring search for A B A B A C

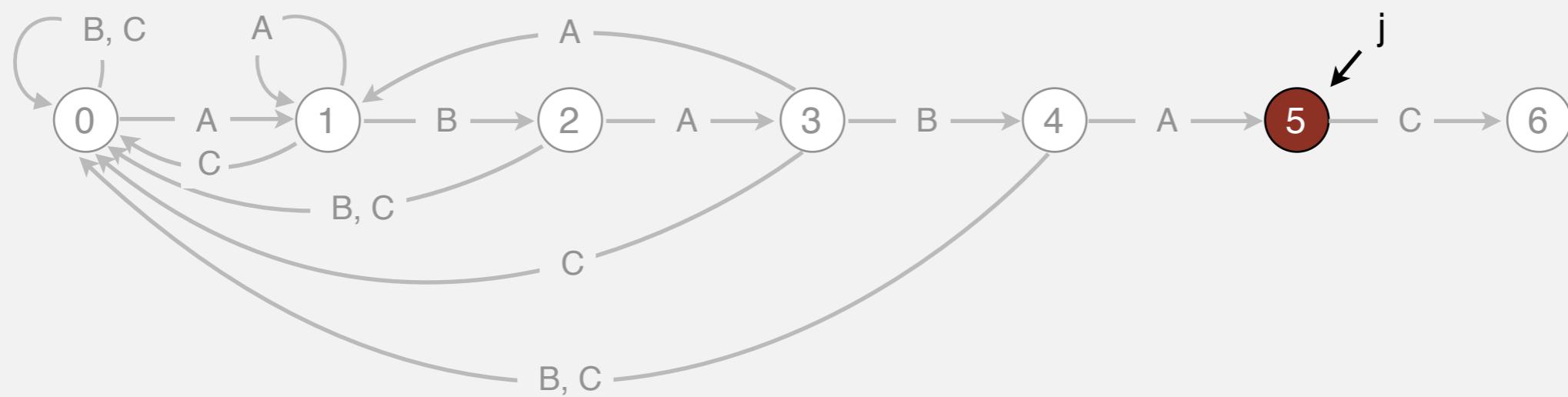


Knuth-Morris-Pratt demo: DFA construction

Mismatch transition: back up if $c \neq \text{pat.charAt}(j)$.

		0	1	2	3	4	5
		A	B	A	B	A	C
pat.charAt(j)	A	1	1	3	1	5	
	B	0	2	0	4	0	
C	0	0	0	0	0	6	

Constructing the DFA for KMP substring search for A B A B A C

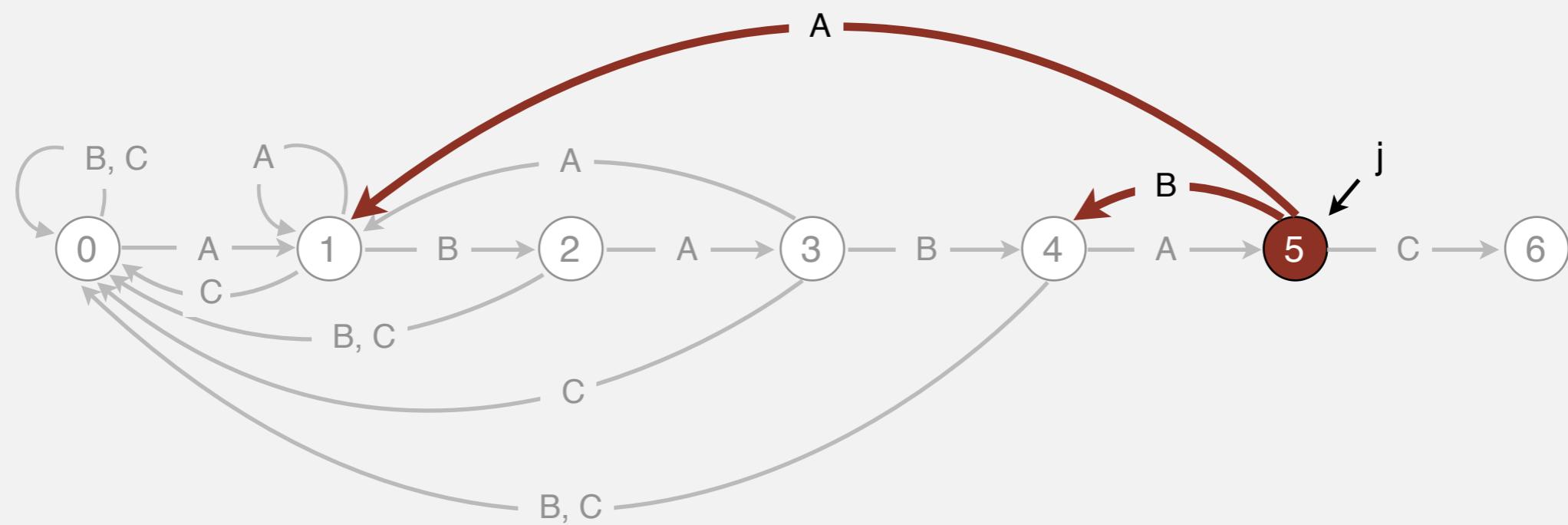


Knuth-Morris-Pratt demo: DFA construction

Mismatch transition: back up if $c \neq \text{pat.charAt}(j)$.

		0	1	2	3	4	5
		A	B	A	B	A	C
pat.charAt(j)	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
C	0	0	0	0	0	0	6

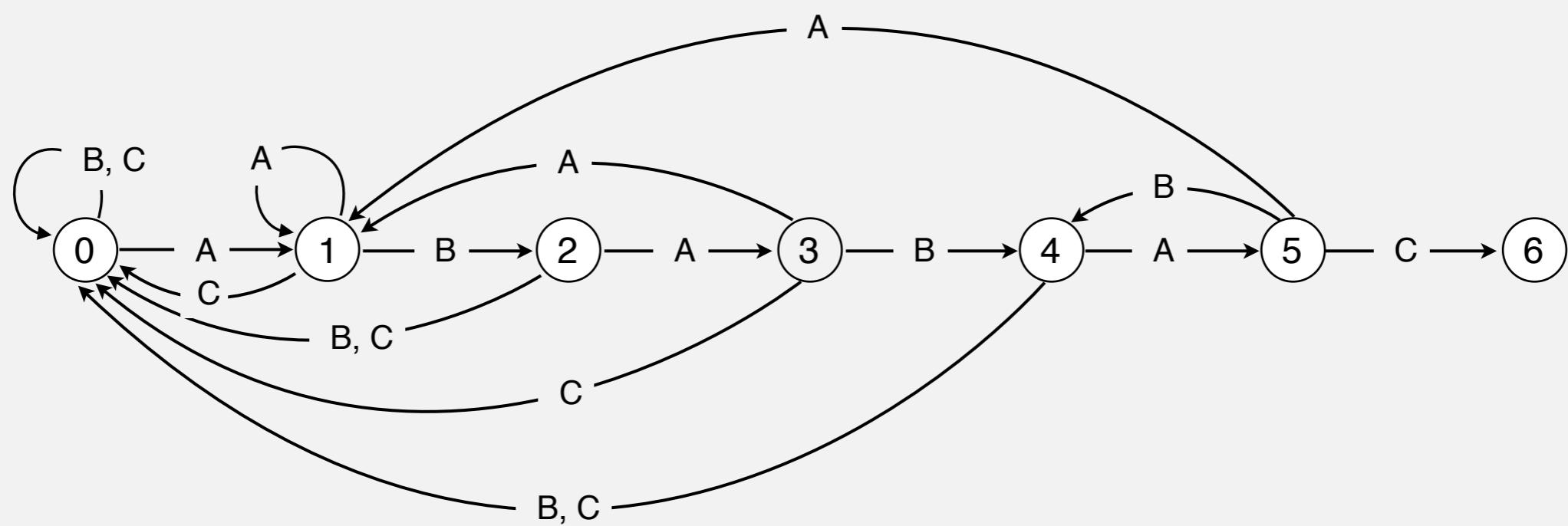
Constructing the DFA for KMP substring search for A B A B A C



Knuth-Morris-Pratt demo: DFA construction

		0	1	2	3	4	5
pat.charAt(j)	A	A	B	A	B	C	
	B	1	1	3	1	5	1
dfa[][][j]	C	0	2	0	4	0	4
		0	0	0	0	0	6

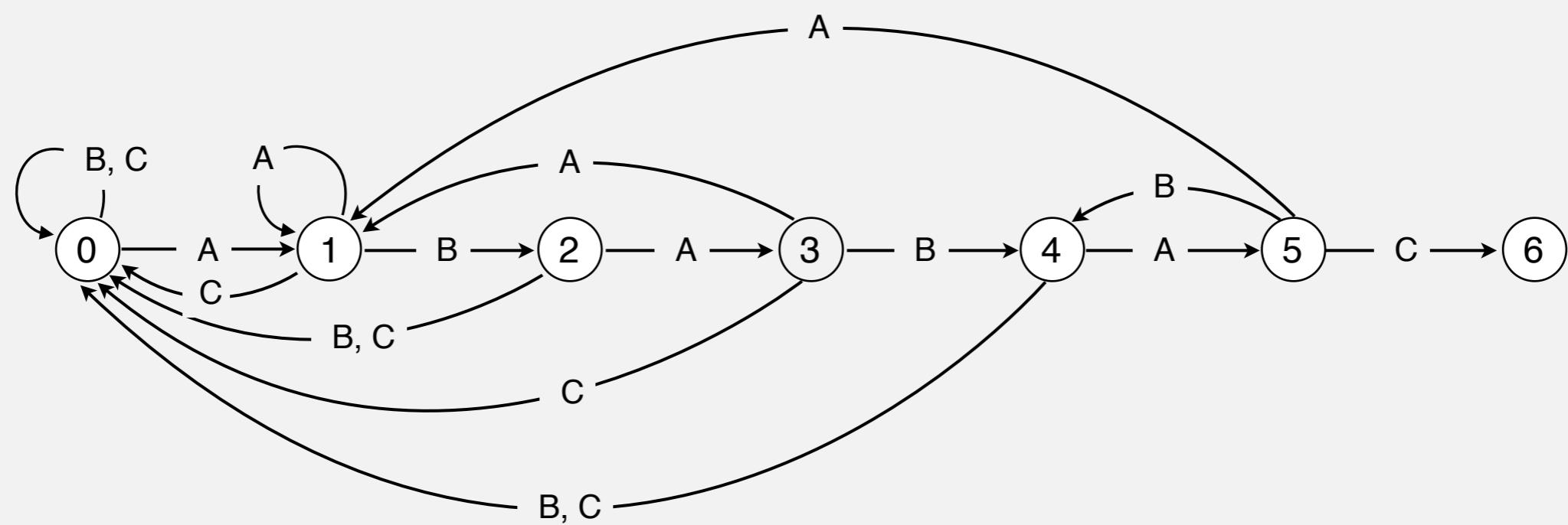
Constructing the DFA for KMP substring search for A B A B A C



Knuth-Morris-Pratt demo: DFA construction in linear time

		0	1	2	3	4	5
pat.charAt(j)	A	A	B	A	B	A	C
	B	1	1	3	1	5	1
dfa[][]j	C	0	2	0	4	0	4
		0	0	0	0	0	6

Constructing the DFA for KMP substring search for A B A B A C

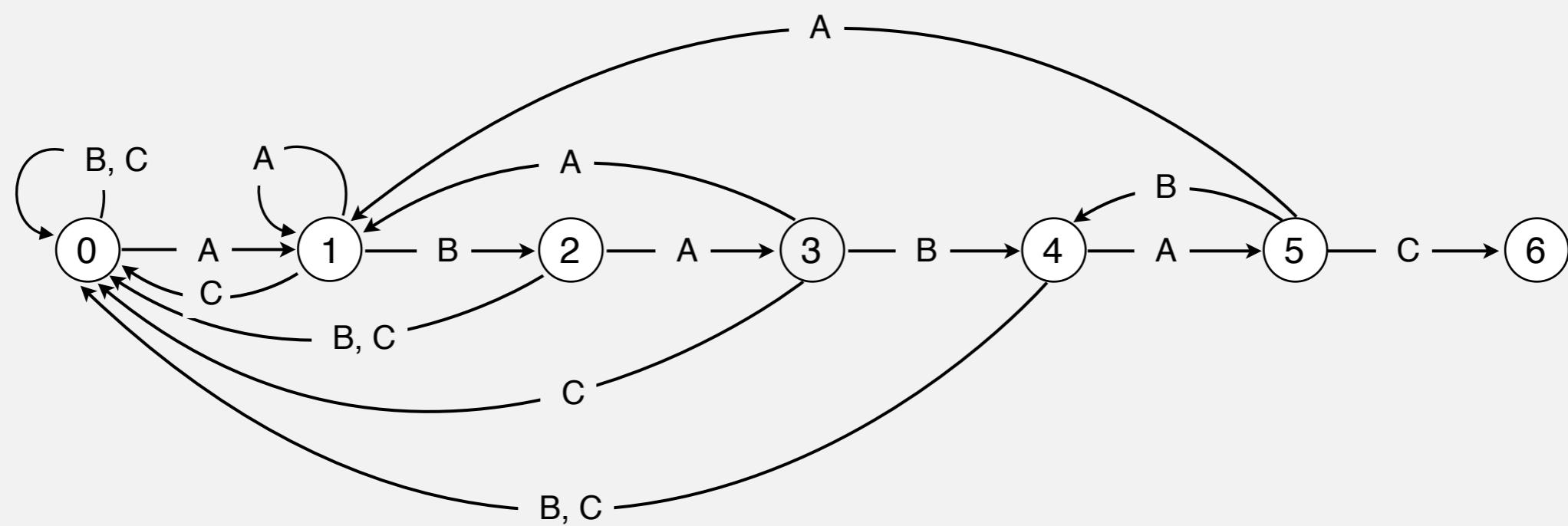


Knuth-Morris-Pratt demo: DFA construction in linear time

Linear Time Algorithm. There is a known algorithm to build the Knuth-Morris-Pratt DFA in linear time. We won't be covering it in this class.

pat.charAt(j)	0	1	2	3	4	5
A	A	B	A	B	A	C
dfa[][]j	1	1	3	1	5	1
B	0	2	0	4	0	4
C	0	0	0	0	0	6

Constructing the DFA for KMP substring search for A B A B A C



Constructing the DFA for KMP substring search: Java implementation

For each state j :

- Copy $dfa[][\mathbf{X}]$ to $dfa[][\mathbf{j}]$ for mismatch case.
- Set $dfa[\mathbf{pat.charAt(j)}][\mathbf{j}]$ to $\mathbf{j+1}$ for match case.
- Update X .

```
public KMP(String pat) {  
    this.pat = pat;  
    M = pat.length();  
    dfa = new int[R][M];  
    dfa[pat.charAt(0)][0] = 1;  
    for (int X = 0, j = 1; j < M; j++) {  
        for (int c = 0; c < R; c++)  
            dfa[c][j] = dfa[c][X];  
        dfa[pat.charAt(j)][j] = j+1;  
        X = dfa[pat.charAt(j)][X];  
    }  
}
```

← copy mismatch cases
← set match case
← update restart state

Constructing the DFA for KMP substring search: Java implementation

For each state j :

- Copy $\text{dfa}[][]X$ to $\text{dfa}[]][j]$ for mismatch case.
- Set $\text{dfa}[\text{pat.charAt}(j)][j]$ to $j+1$ for match case.
- Update X .

```
public KMP(String pat) {  
    this.pat = pat;  
    M = pat.length();  
    dfa = new int[R][M];  
    dfa[pat.charAt(0)][0] = 1;  
    for (int X = 0, j = 1; j < M; j++) {  
        for (int c = 0; c < R; c++)  
            dfa[c][j] = dfa[c][X];  
        dfa[pat.charAt(j)][j] = j+1;  
        X = dfa[pat.charAt(j)][X];  
    }  
}
```

← copy mismatch cases
← set match case
← update restart state

Running time. M character accesses (but space/time proportional to $R M$).

KMP substring search analysis

Proposition. KMP substring search accesses no more than $M + N$ chars to search for a pattern of length M in a text of length N .

KMP substring search analysis

Proposition. KMP substring search accesses no more than $M + N$ chars to search for a pattern of length M in a text of length N .

Pf. Each pattern char accessed once when constructing the DFA; each text char accessed once (in the worst case) when simulating the DFA.

KMP substring search analysis

Proposition. KMP substring search accesses no more than $M + N$ chars to search for a pattern of length M in a text of length N .

Pf. Each pattern char accessed once when constructing the DFA; each text char accessed once (in the worst case) when simulating the DFA.

Proposition. KMP constructs `dfa[][]` in time and space proportional to $R M$.

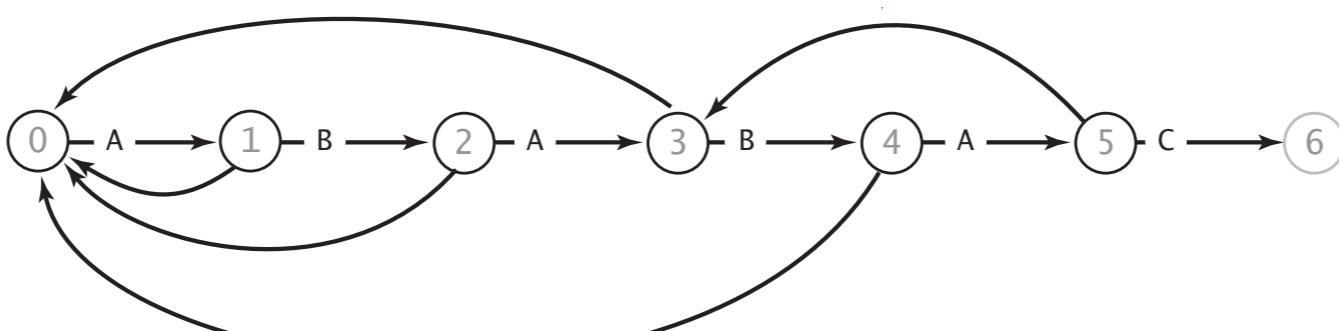
KMP substring search analysis

Proposition. KMP substring search accesses no more than $M + N$ chars to search for a pattern of length M in a text of length N .

Pf. Each pattern char accessed once when constructing the DFA; each text char accessed once (in the worst case) when simulating the DFA.

Proposition. KMP constructs $\text{dfa}[][]$ in time and space proportional to $R M$.

Larger alphabets. Improved version of KMP constructs $\text{nfa}[]$ in time and space proportional to M .



KMP NFA for ABABAC

Knuth-Morris-Pratt: brief history

- Independently discovered by two theoreticians and a hacker.

Knuth-Morris-Pratt: brief history

- Independently discovered by two theoreticians and a hacker.
 - Knuth: inspired by esoteric theorem, discovered linear algorithm



Don Knuth

Knuth-Morris-Pratt: brief history

- Independently discovered by two theoreticians and a hacker.
 - Knuth: inspired by esoteric theorem, discovered linear algorithm
 - Pratt: made running time independent of alphabet size



Don Knuth



Vaughan Pratt

Knuth-Morris-Pratt: brief history

- Independently discovered by two theoreticians and a hacker.
 - Knuth: inspired by esoteric theorem, discovered linear algorithm
 - Pratt: made running time independent of alphabet size
 - Morris: built a text editor for the CDC 6400 computer



Don Knuth



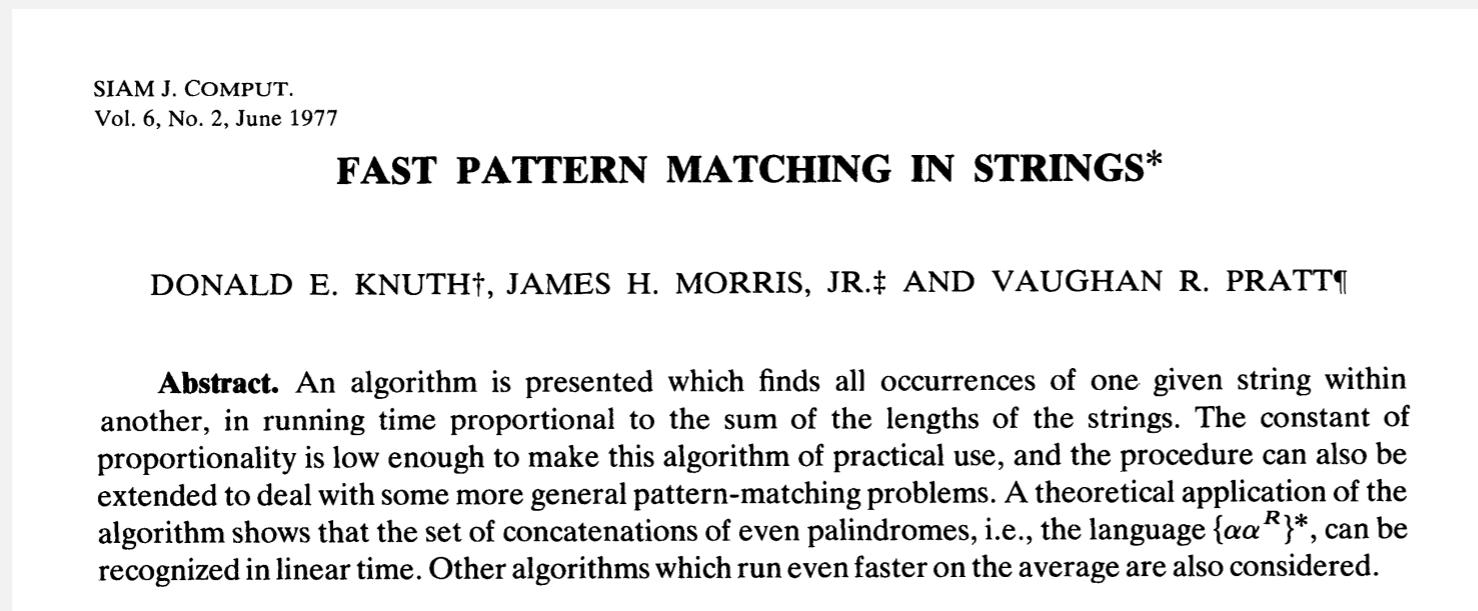
Jim Morris



Vaughan Pratt

Knuth-Morris-Pratt: brief history

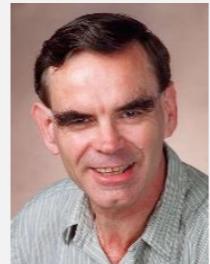
- Independently discovered by two theoreticians and a hacker.
 - Knuth: inspired by esoteric theorem, discovered linear algorithm
 - Pratt: made running time independent of alphabet size
 - Morris: built a text editor for the CDC 6400 computer
- Theory meets practice.



Don Knuth



Jim Morris



Vaughan Pratt

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

5.3 SUBSTRING SEARCH

- ▶ introduction
- ▶ brute force
- ▶ Knuth-Morris-Pratt
- ▶ Boyer-Moore
- ▶ Rabin-Karp



Robert Boyer J. Strother Moore

Boyer-Moore: mismatched character heuristic

Intuition.

- Scan characters in pattern from right to left.
- Can skip as many as M text chars when finding one not in the pattern.

i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
		<i>text</i> → F	I	N	D	I	N	A	H	A	Y	S	T	A	C	K	N	E	E	D	L	E	I	N	A
0	5	N	E	E	D	L	E ← <i>pattern</i>																		
5	5						N	E	E	D	L	E													
11	4												N	E	E	D	L	E							
15	0															N	E	E	D	L	E				

Won't Be Covering.

- Not enough time to cover this algorithm in depth

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

5.5 DATA COMPRESSION

- ▶ introduction
- ▶ run-length coding
- ▶ Huffman compression
- ▶ LZW compression

Data compression

Compression reduces the size of a file:

- To save **space** when storing it.

Data compression

Compression reduces the size of a file:

- To save **space** when storing it.
- To save **time** when transmitting it.

Data compression

Compression reduces the size of a file:

- To save **space** when storing it.
- To save **time** when transmitting it.
- Most files have lots of redundancy.

Data compression

Compression reduces the size of a file:

- To save **space** when storing it.
- To save **time** when transmitting it.
- Most files have lots of redundancy.

Who needs compression?

Data compression

Compression reduces the size of a file:

- To save **space** when storing it.
- To save **time** when transmitting it.
- Most files have lots of redundancy.

Who needs compression?

- Moore's law: # transistors on a chip doubles every 18–24 months.

Data compression

Compression reduces the size of a file:

- To save **space** when storing it.
- To save **time** when transmitting it.
- Most files have lots of redundancy.

Who needs compression?

- Moore's law: # transistors on a chip doubles every 18–24 months.
- Parkinson's law: data expands to fill space available.
- Text, images, sound, video, ...

“Everyday, we create 2.5 quintillion bytes of data—so much that 90% of the data in the world today has been created in the last two years alone. ” — IBM report on big data (2011)

Data compression

Compression reduces the size of a file:

- To save **space** when storing it.
- To save **time** when transmitting it.
- Most files have lots of redundancy.

Who needs compression?

- Moore's law: # transistors on a chip doubles every 18–24 months.
- Parkinson's law: data expands to fill space available.
- Text, images, sound, video, ...

“Everyday, we create 2.5 quintillion bytes of data—so much that 90% of the data in the world today has been created in the last two years alone. ” — IBM report on big data (2011)

Basic concepts ancient (1950s), best technology recently developed.

Applications

Applications

Generic file compression.

- Files: GZIP, BZIP, 7z.
- Archivers: PKZIP.
- File systems: NTFS, ZFS, HFS+, ReFS, GFS.



Applications

Generic file compression.

- Files: GZIP, BZIP, 7z.
- Archivers: PKZIP.
- File systems: NTFS, ZFS, HFS+, ReFS, GFS.



Multimedia.

- Images: GIF, JPEG.
- Sound: MP3.
- Video: MPEG, DivX™, HDTV.



Applications

Generic file compression.

- Files: GZIP, BZIP, 7z.
- Archivers: PKZIP.
- File systems: NTFS, ZFS, HFS+, ReFS, GFS.



Multimedia.

- Images: GIF, JPEG.
- Sound: MP3.
- Video: MPEG, DivX™, HDTV.



Communication.

- ITU-T T4 Group 3 Fax.
- V.42bis modem.
- Skype, Google hangout.



Applications

Generic file compression.

- Files: GZIP, BZIP, 7z.
- Archivers: PKZIP.
- File systems: NTFS, ZFS, HFS+, ReFS, GFS.



Multimedia.

- Images: GIF, JPEG.
- Sound: MP3.
- Video: MPEG, DivX™, HDTV.



Communication.

- ITU-T T4 Group 3 Fax.
- V.42bis modem.
- Skype, Google hangout.



Databases. Google, Facebook,



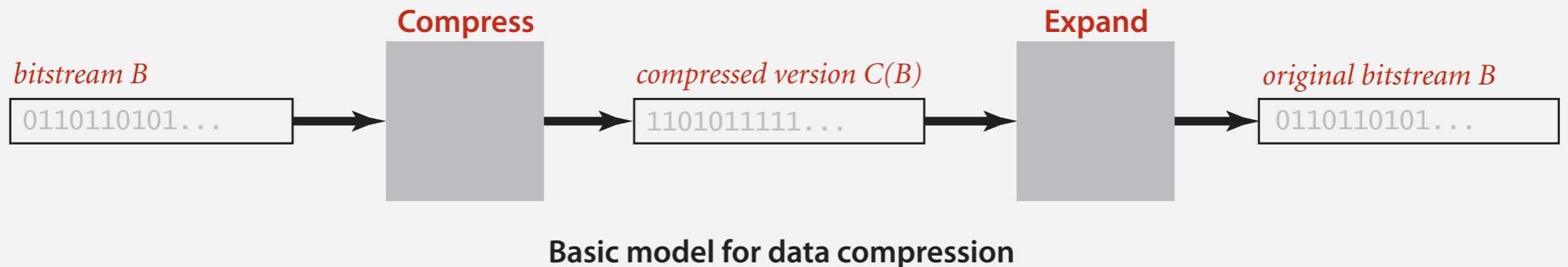
Lossless compression and expansion

Message. Binary data B we want to compress.

Compress. Generates a "compressed" representation $C(B)$.

Expand. Reconstructs original bitstream B .

uses fewer bits
(you hope)



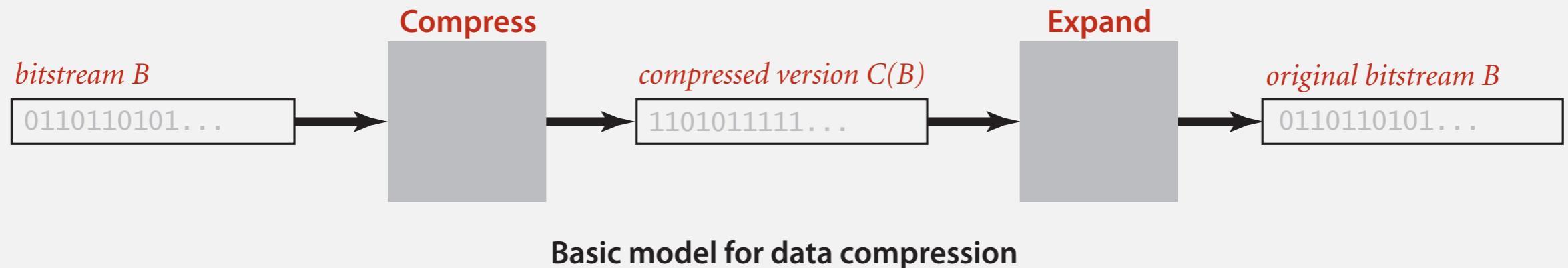
Lossless compression and expansion

Message. Binary data B we want to compress.

Compress. Generates a "compressed" representation $C(B)$.

Expand. Reconstructs original bitstream B .

uses fewer bits
(you hope)



Compression ratio. Bits in $C(B)$ / bits in B .

Ex. 50–75% or better compression ratio for natural language.

Food for thought

Data compression has been omnipresent since antiquity:

- Number systems.
- Natural languages.
- Mathematical notation.

|||| |

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$$

Food for thought

Data compression has been omnipresent since antiquity:

- Number systems.
- Natural languages.
- Mathematical notation.

||||| |

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$$

has played a central role in communications technology,

- Grade 2 Braille.
- Morse code.
- Telephone system.

b	r	a	i	I	I	e
●○ ●○ ○○	●○ ●○ ○○	●○ ○○ ○○	○● ●○ ○○	●○ ●○ ○○	●○ ●○ ○○	●○ ○○ ○○
but	rather	a	I	like	like	every

Food for thought

Data compression has been omnipresent since antiquity:

- Number systems.
- Natural languages.
- Mathematical notation.

||||| |

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$$

has played a central role in communications technology,

- Grade 2 Braille.
- Morse code.
- Telephone system.

b	r	a	i	I	I	e
●○ ●○ ○○	●○ ●○ ○○	●○ ○○ ○○	○● ○○ ○○	●○ ○○ ○○	●○ ●○ ○○	●○ ●○ ○○
but	rather	a	I	like	like	every

and is part of modern life.

- MP3.
- MPEG.



Food for thought

Data compression has been omnipresent since antiquity:

- Number systems.
- Natural languages.
- Mathematical notation.

||||| |

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$$

has played a central role in communications technology,

- Grade 2 Braille.
- Morse code.
- Telephone system.

b	r	a	i	I	I	e
●○ ●○ ○○	●○ ●○ ○○	○○ ○○ ○○	○● ○○ ○○	●○ ○○ ○○	●○ ●○ ○○	●○ ●○ ○○
but	rather	a	I	like	like	every

and is part of modern life.

- MP3.
- MPEG.



Q. What role will it play in the future?

Data representation: genomic code

Genome. String over the alphabet { A, C, T, G }.

Goal. Encode an N -character genome: ATAGATGCATAG...

Data representation: genomic code

Genome. String over the alphabet { A, C, T, G }.

Goal. Encode an N -character genome: A T A G A T G C A T A G ...

Standard ASCII encoding.

- 8 bits per char.
- $8N$ bits.

char	hex	binary
A	41	01000001
C	43	01000011
T	54	01010100
G	47	01000111

Data representation: genomic code

Genome. String over the alphabet { A, C, T, G }.

Goal. Encode an N -character genome: A T A G A T G C A T A G ...

Standard ASCII encoding.

- 8 bits per char.
- $8N$ bits.

char	hex	binary
A	41	01000001
C	43	01000011
T	54	01010100
G	47	01000111

Two-bit encoding.

- 2 bits per char.
- $2N$ bits.

char	binary
A	00
C	01
T	10
G	11

Data representation: genomic code

Genome. String over the alphabet { A, C, T, G }.

Goal. Encode an N -character genome: A T A G A T G C A T A G ...

Standard ASCII encoding.

- 8 bits per char.
- $8N$ bits.

char	hex	binary
A	41	01000001
C	43	01000011
T	54	01010100
G	47	01000111

Two-bit encoding.

- 2 bits per char.
- $2N$ bits.

char	binary
A	00
C	01
T	10
G	11

Fixed-length code. k -bit code supports alphabet of size 2^k .

Data representation: genomic code

Genome. String over the alphabet { A, C, T, G }.

Goal. Encode an N -character genome: A T A G A T G C A T A G ...

Standard ASCII encoding.

- 8 bits per char.
- $8N$ bits.

char	hex	binary
A	41	01000001
C	43	01000011
T	54	01010100
G	47	01000111

Two-bit encoding.

- 2 bits per char.
- $2N$ bits.

char	binary
A	00
C	01
T	10
G	11

Fixed-length code. k -bit code supports alphabet of size 2^k .

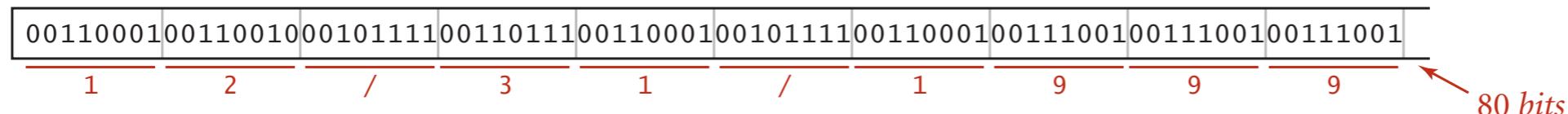
Amazing but true. Some genomic databases in 1990s used ASCII.

Writing binary data

Date representation. Three different ways to represent 12/31/1999.

A character stream (StdOut)

```
StdOut.print(month + "/" + day + "/" + year);
```



Writing binary data

Date representation. Three different ways to represent 12/31/1999.

A character stream (StdOut)

```
StdOut.print(month + "/" + day + "/" + year);
```

0011000100110010001011110011011100110001001011110011000100111001001110010011100100111001

1 2 / 3 1 / 1 9 9 9 ↗
80 bits

Three ints (BinaryStdOut)

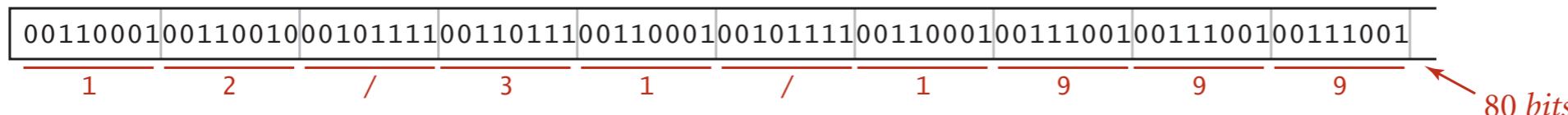
```
BinaryStdOut.write(month);  
BinaryStdOut.write(day);  
BinaryStdOut.write(year);
```

Writing binary data

Date representation. Three different ways to represent 12/31/1999.

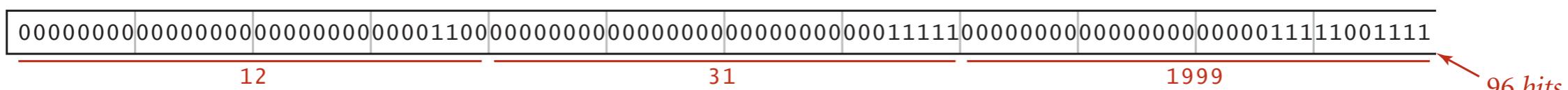
A character stream (StdOut)

```
StdOut.print(month + "/" + day + "/" + year);
```



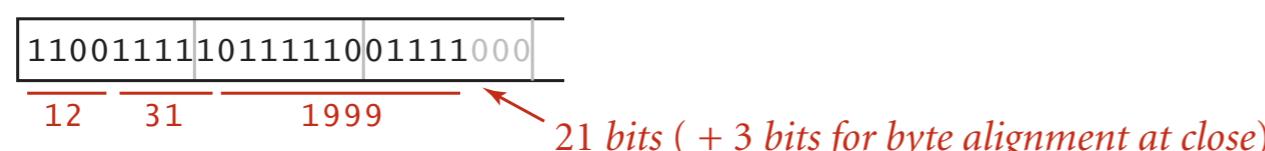
Three ints (BinaryStdOut)

```
BinaryStdOut.write(month);  
BinaryStdOut.write(day);  
BinaryStdOut.write(year);
```



A 4-bit field, a 5-bit field, and a 12-bit field (BinaryStdOut)

```
BinaryStdOut.write(month, 4);
BinaryStdOut.write(day, 5);
BinaryStdOut.write(year, 12);
```



Universal data compression

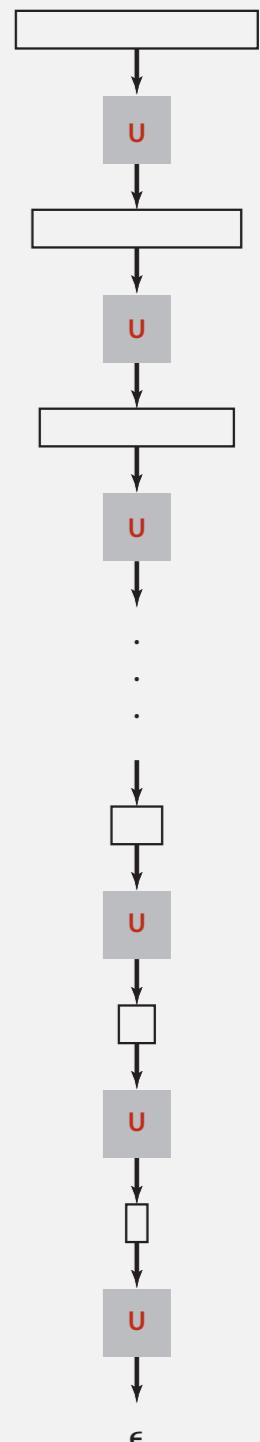
Proposition. No algorithm can compress every bitstring.

Universal data compression

Proposition. No algorithm can compress every bitstring.

Pf 1. [by contradiction]

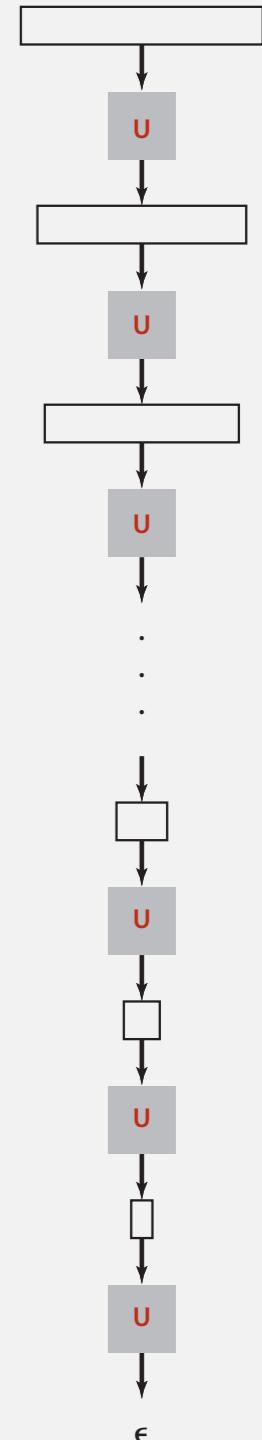
- Suppose you have a universal data compression algorithm U that can compress every bitstream.
- Given bitstring B_0 , compress it to get smaller bitstring B_1 .
- Compress B_1 to get a smaller bitstring B_2 .
- Continue until reaching bitstring of size 0.
- Implication: all bitstrings can be compressed to 0 bits!



Universal
data compression?

Universal data compression

Proposition. No algorithm can compress every bitstring.



Pf 1. [by contradiction]

- Suppose you have a universal data compression algorithm U that can compress every bitstream.
 - Given bitstring B_0 , compress it to get smaller bitstring B_1 .
 - Compress B_1 to get a smaller bitstring B_2 .
 - Continue until reaching bitstring of size 0.
 - Implication: all bitstrings can be compressed to 0 bits!

Pf 2. [by counting]

- Suppose your algorithm that can compress all 1,000-bit strings.
 - 2^{1000} possible bitstrings with 1,000 bits.
 - Only $1 + 2 + 4 + \dots + 2^{998} + 2^{999}$ can be encoded with ≤ 999 bits.
 - Similarly, only 1 in 2^{499} bitstrings can be encoded with ≤ 500 bits!

Universal data compression?

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

5.5 DATA COMPRESSION

- ▶ introduction
- ▶ run-length coding
- ▶ Huffman compression
- ▶ LZW compression

Run-length encoding

Simple type of redundancy in a bitstream. Long runs of repeated bits.

0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1



40 bits

Run-length encoding

Simple type of redundancy in a bitstream. Long runs of repeated bits.

0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
↑ 40 bits

Representation. 4-bit counts to represent alternating runs of 0s and 1s:
15 0s, then 7 1s, then 7 0s, then 11 1s.

1 1 1 1 0 1 1 1 0 1 1 1 1 0 1 1
15 7 7 11 ← 16 bits (instead of 40)

Run-length encoding

Simple type of redundancy in a bitstream. Long runs of repeated bits.

0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
 40 bits

Representation. 4-bit counts to represent alternating runs of 0s and 1s:
15 0s, then 7 1s, then 7 0s, then 11 1s.

1 1 1 1 0 1 1 1 0 1 1 1 1 0 1 1 ← 16 bits (instead of 40)
15 7 7 11

Q. How many bits to store the counts?

Run-length encoding

Simple type of redundancy in a bitstream. Long runs of repeated bits.

0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
 40 bits

Representation. 4-bit counts to represent alternating runs of 0s and 1s:
15 0s, then 7 1s, then 7 0s, then 11 1s.

1 1 1 1 0 1 1 1 0 1 1 1 1 0 1 1 ← 16 bits (instead of 40)
15 7 7 11

Q. How many bits to store the counts?

A. We'll use 8 (but 4 in the example above).

Run-length encoding

Simple type of redundancy in a bitstream. Long runs of repeated bits.

0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
↑ 40 bits

Representation. 4-bit counts to represent alternating runs of 0s and 1s:
15 0s, then 7 1s, then 7 0s, then 11 1s.

1 1 1 1 0 1 1 1 0 1 1 1 1 0 1 1 ← 16 bits (instead of 40)
15 7 7 11

- Q. How many bits to store the counts?
 - A. We'll use 8 (but 4 in the example above).

- Q. What to do when run length exceeds max count?

Run-length encoding

Simple type of redundancy in a bitstream. Long runs of repeated bits.

0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
↑ 40 bits

Representation. 4-bit counts to represent alternating runs of 0s and 1s:
15 0s, then 7 1s, then 7 0s, then 11 1s.

1 1 1 1 0 1 1 1 0 1 1 1 1 0 1 1 ← 16 bits (instead of 40)
15 7 7 11

Q. How many bits to store the counts?

A. We'll use 8 (but 4 in the example above).

Q. What to do when run length exceeds max count?

A. If longer than 255, intersperse runs of length 0.

Run-length encoding

Simple type of redundancy in a bitstream. Long runs of repeated bits.

0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
↑ 40 bits

Representation. 4-bit counts to represent alternating runs of 0s and 1s:
15 0s, then 7 1s, then 7 0s, then 11 1s.

1 1 1 1 0 1 1 1 0 1 1 1 1 0 1 1 ← 16 bits (instead of 40)
15 7 7 11

Q. How many bits to store the counts?

A. We'll use 8 (but 4 in the example above).

Q. What to do when run length exceeds max count?

A. If longer than 255, intersperse runs of length 0.

Applications. JPEG, ITU-T T4 Group 3 Fax, ...

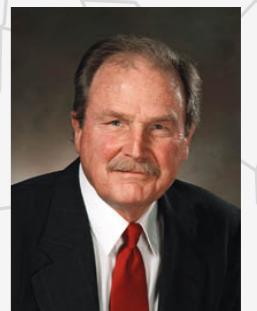
Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

5.5 DATA COMPRESSION

- ▶ introduction
- ▶ run-length coding
- ▶ Huffman compression
- ▶ LZW compression



David Huffman

Variable-length codes

Use different number of bits to encode different chars.

Ex. Morse code: • • • - - - • • •

Letters	Numbers
A	•—
B	—•••
C	—•—•
D	—••
E	•
F	••—•
G	——•
H	••••
I	••
J	•——
K	—•—
L	•—••
M	——
N	—•
O	———
P	•——•
Q	——•—
R	•—•
S	•••
T	—
U	••—
V	•••—
W	•—
X	—••—
Y	—•—
Z	——••

Variable-length codes

Use different number of bits to encode different chars.

Ex. Morse code: • • • - - - • • •

Issue. Ambiguity.

SOS ?

V7 ?

IAMIE ?

EENI ?

Letters	Numbers
A	•—
B	—•••
C	—•—•
D	—••
E	•
F	••—•
G	——•
H	••••
I	••
J	•——
K	—•—
L	•—••
M	——
N	—•
O	———
P	•——•
Q	——•—
R	•—•
S	•••
T	—
U	••—
V	•••—
W	•—
X	—••—
Y	—•—
Z	——••

Variable-length codes

Use different number of bits to encode different chars.

Ex. Morse code: • • • - - - • • •

Issue. Ambiguity.

SOS ?

V7 ?

I AMIE ?

E EWNI ?

In practice. Use a medium gap to separate codewords.

codeword for S is a prefix
of codeword for V

Letters	Numbers
A	•—
B	—•••
C	—•—•
D	—••
E	•
F	••—•
G	——•
H	••••
I	••
J	•——
K	—•—
L	•—••
M	——
N	—•
O	———
P	•——•
Q	——•—
R	•—•
S	•••
T	—
U	••—
V	•••—
W	•——
X	—••—
Y	—•—
Z	——••

Variable-length codes

Q. How do we avoid ambiguity?

Variable-length codes

Q. How do we avoid ambiguity?

A. Ensure that no codeword is a **prefix** of another.

Variable-length codes

Q. How do we avoid ambiguity?

A. Ensure that no codeword is a **prefix** of another.

Ex 1. Fixed-length code.

Variable-length codes

Q. How do we avoid ambiguity?

A. Ensure that no codeword is a **prefix** of another.

Ex 1. Fixed-length code.

Ex 2. Append special stop char to each codeword.

Variable-length codes

Q. How do we avoid ambiguity?

A. Ensure that no codeword is a **prefix** of another.

Ex 1. Fixed-length code.

Ex 2. Append special stop char to each codeword.

Ex 3. General prefix-free code.

Codeword table

<i>key</i>	<i>value</i>
!	101
A	0
B	1111
C	110
D	100
R	1110

Compressed bitstring

011111110011001000111111100101 ← 30 bits
A B RA CA DA B RA !

Variable-length codes

Q. How do we avoid ambiguity?

A. Ensure that no codeword is a **prefix** of another.

Ex 1. Fixed-length code.

Ex 2. Append special stop char to each codeword.

Ex 3. General prefix-free code.

Codeword table

<i>key</i>	<i>value</i>
!	101
A	0
B	1111
C	110
D	100
R	1110

Compressed bitstring

01111111001100100011111100101
A B RA CA DA B RA ! ← 30 bits

Codeword table

<i>key</i>	<i>value</i>
!	101
A	11
B	00
C	010
D	100
R	011

Compressed bitstring

11000111101011100110001111101
A B R A C A D A B R A ! ← 29 bits

Prefix-free codes: trie representation

Q. How to represent the prefix-free code?

Codeword table

<i>key</i>	<i>value</i>
!	101
A	0
B	1111
C	110
D	100
R	1110

Compressed bitstring

01111111001100100011111100101
A B RA CA DA B RA ! ← 30 bits

Codeword table

<i>key</i>	<i>value</i>
!	101
A	11
B	00
C	010
D	100
R	011

Compressed bitstring

11000111101011100110001111101
A B RA CA DA B RA ! ← 29 bits

Prefix-free codes: trie representation

Q. How to represent the prefix-free code?

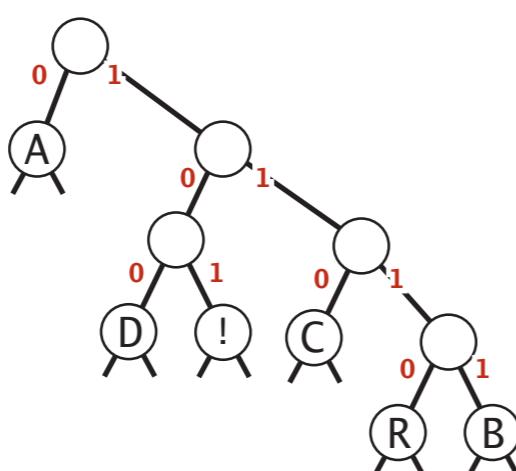
A. A binary trie!

- Chars in leaves.
- Codeword is path from root to leaf.

Codeword table

key	value
!	101
A	0
B	1111
C	110
D	100
R	1110

Trie representation



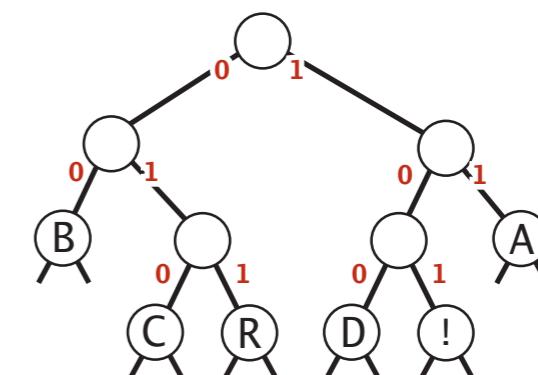
Compressed bitstring

01111111001100100011111100101
A B RA CA DA B RA ! ← 30 bits

Codeword table

key	value
!	101
A	11
B	00
C	010
D	100
R	011

Trie representation



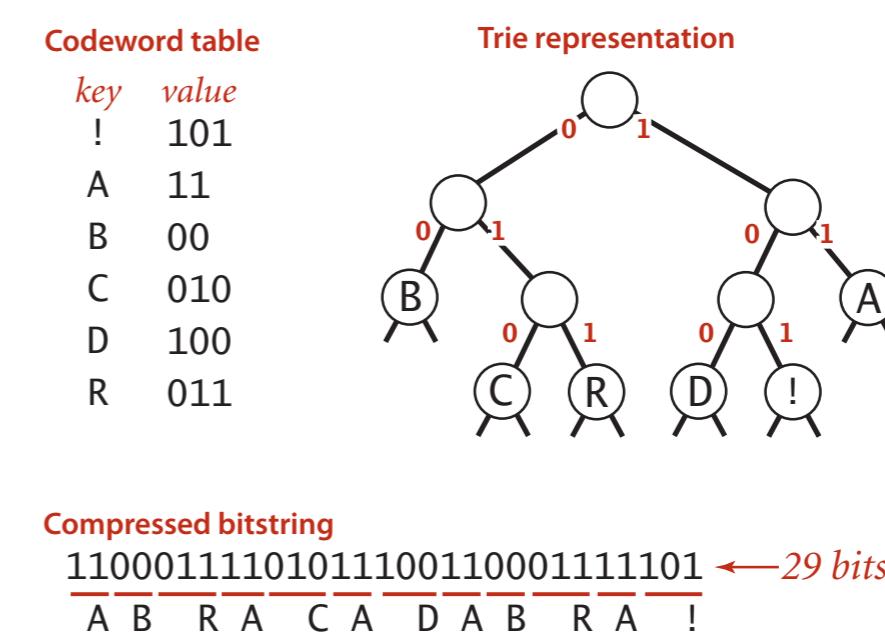
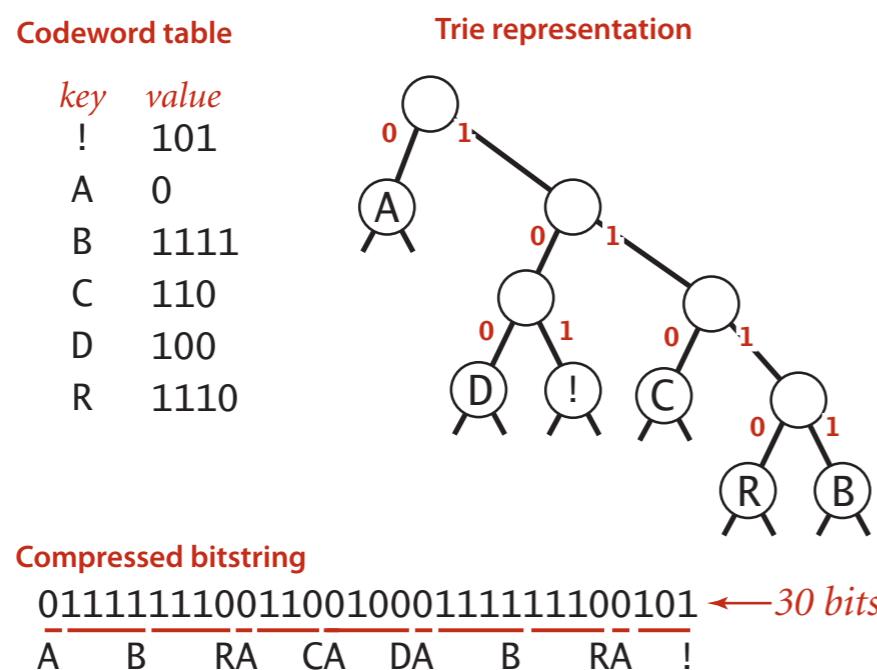
Compressed bitstring

11000111101011100110001111101 ← 29 bits
A B RA CA DA B RA !

Prefix-free codes: compression and expansion

Compression.

- Method 1: start at leaf; follow path up to the root; print bits in reverse.
- Method 2: create ST of key-value pairs.



Prefix-free codes: compression and expansion

Compression.

- Method 1: start at leaf; follow path up to the root; print bits in reverse.
- Method 2: create ST of key-value pairs.

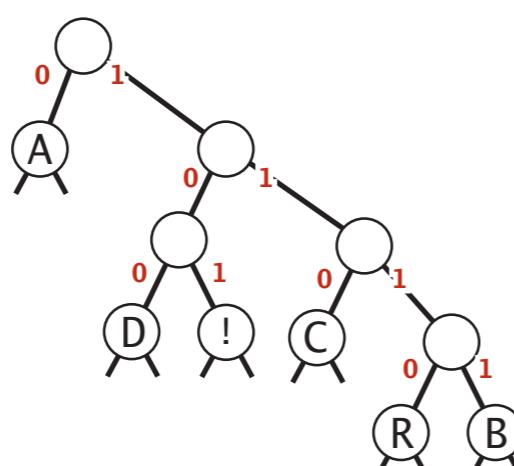
Expansion.

- Start at root.
- Go left if bit is 0; go right if 1.
- If leaf node, print char and return to root.

Codeword table

key	value
!	101
A	0
B	1111
C	110
D	100
R	1110

Trie representation



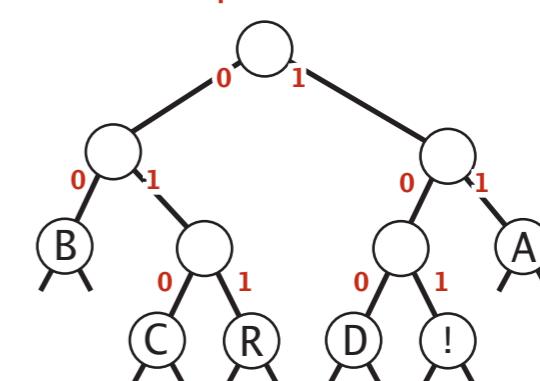
Compressed bitstring

01111111001100100011111100101
A B RA CA DA B RA ! ← 30 bits

Codeword table

key	value
!	101
A	11
B	00
C	010
D	100
R	011

Trie representation



Compressed bitstring

11000111101011100110001111101 ← 29 bits
A B RA CA DA B RA !

Huffman coding overview

Dynamic model. Use a custom prefix-free code for each message.

Huffman coding overview

Dynamic model. Use a custom prefix-free code for each message.

Compression.

- Read message.
- Built **best** prefix-free code for message. How?
- Write prefix-free code (as a trie) to file.
- Compress message using prefix-free code.

Huffman coding overview

Dynamic model. Use a custom prefix-free code for each message.

Compression.

- Read message.
- Built **best** prefix-free code for message. How?
- Write prefix-free code (as a trie) to file.
- Compress message using prefix-free code.

Expansion.

- Read prefix-free code (as a trie) from file.
- Read compressed message and expand using trie.

Huffman trie node data type

```
private static class Node implements Comparable<Node> {  
    private final char ch; // used only for leaf nodes  
    private final int freq; // used only for compress  
    private final Node left, right;
```

```
public Node(char ch, int freq, Node left, Node right) {  
    this.ch = ch;  
    this.freq = freq;  
    this.left = left;  
    this.right = right;  
}
```

← initializing constructor

```
public boolean isLeaf()  
{ return left == null && right == null; }
```

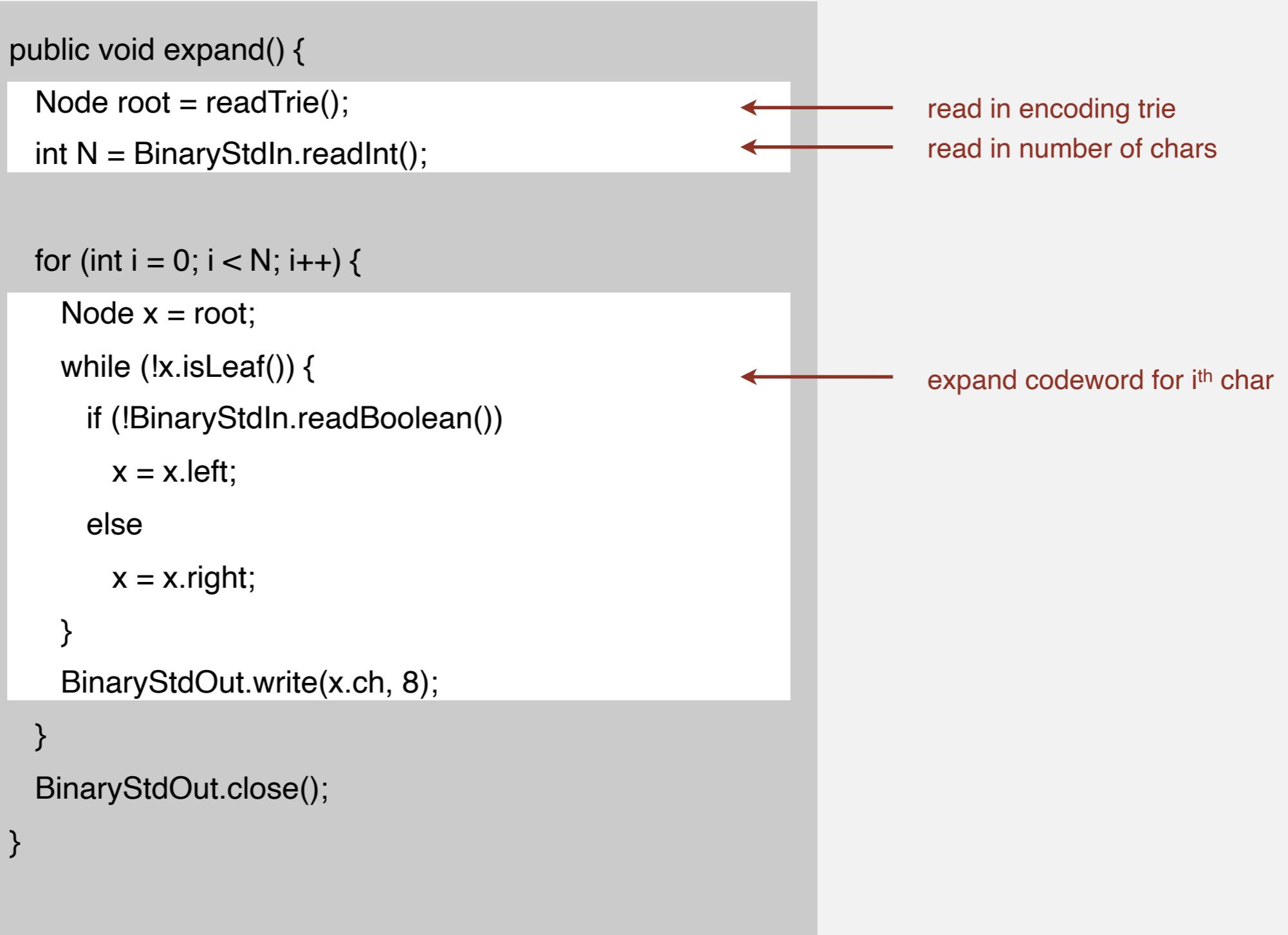
← is Node a leaf?

```
public int compareTo(Node that)  
{ return this.freq - that.freq; }  
}
```

← compare Nodes by frequency
(stay tuned)

Prefix-free codes: expansion

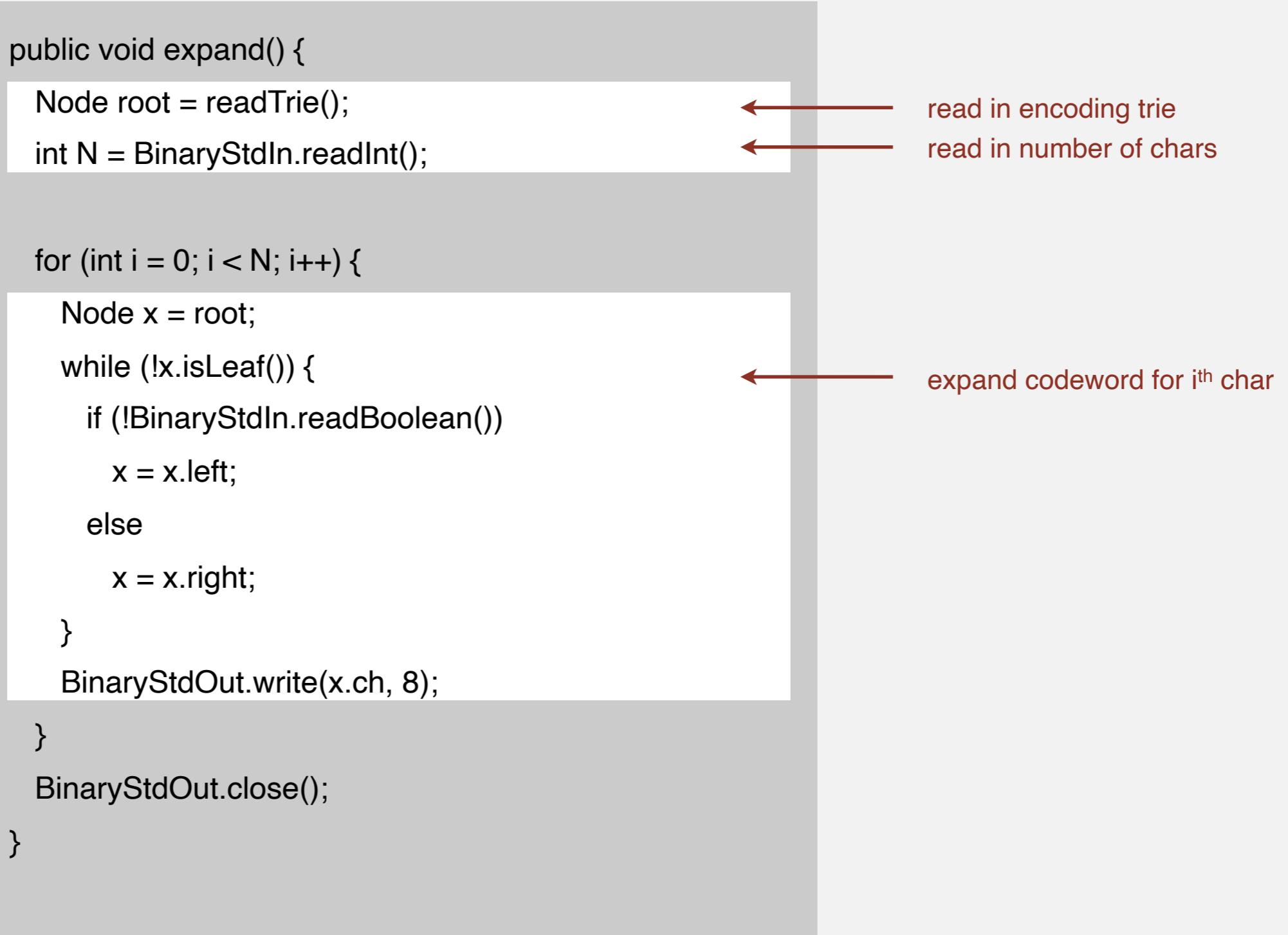
```
public void expand() {  
    Node root = readTrie();  
    int N = BinaryStdIn.readInt();  
  
    for (int i = 0; i < N; i++) {  
        Node x = root;  
        while (!x.isLeaf()) {  
            if (!BinaryStdIn.readBoolean())  
                x = x.left;  
            else  
                x = x.right;  
        }  
        BinaryStdOut.write(x.ch, 8);  
    }  
    BinaryStdOut.close();  
}
```



read in encoding trie
read in number of chars
expand codeword for i^{th} char

Prefix-free codes: expansion

```
public void expand() {  
    Node root = readTrie();  
    int N = BinaryStdIn.readInt();  
  
    for (int i = 0; i < N; i++) {  
        Node x = root;  
        while (!x.isLeaf()) {  
            if (!BinaryStdIn.readBoolean())  
                x = x.left;  
            else  
                x = x.right;  
        }  
        BinaryStdOut.write(x.ch, 8);  
    }  
    BinaryStdOut.close();  
}
```



Running time. Linear in input size N .

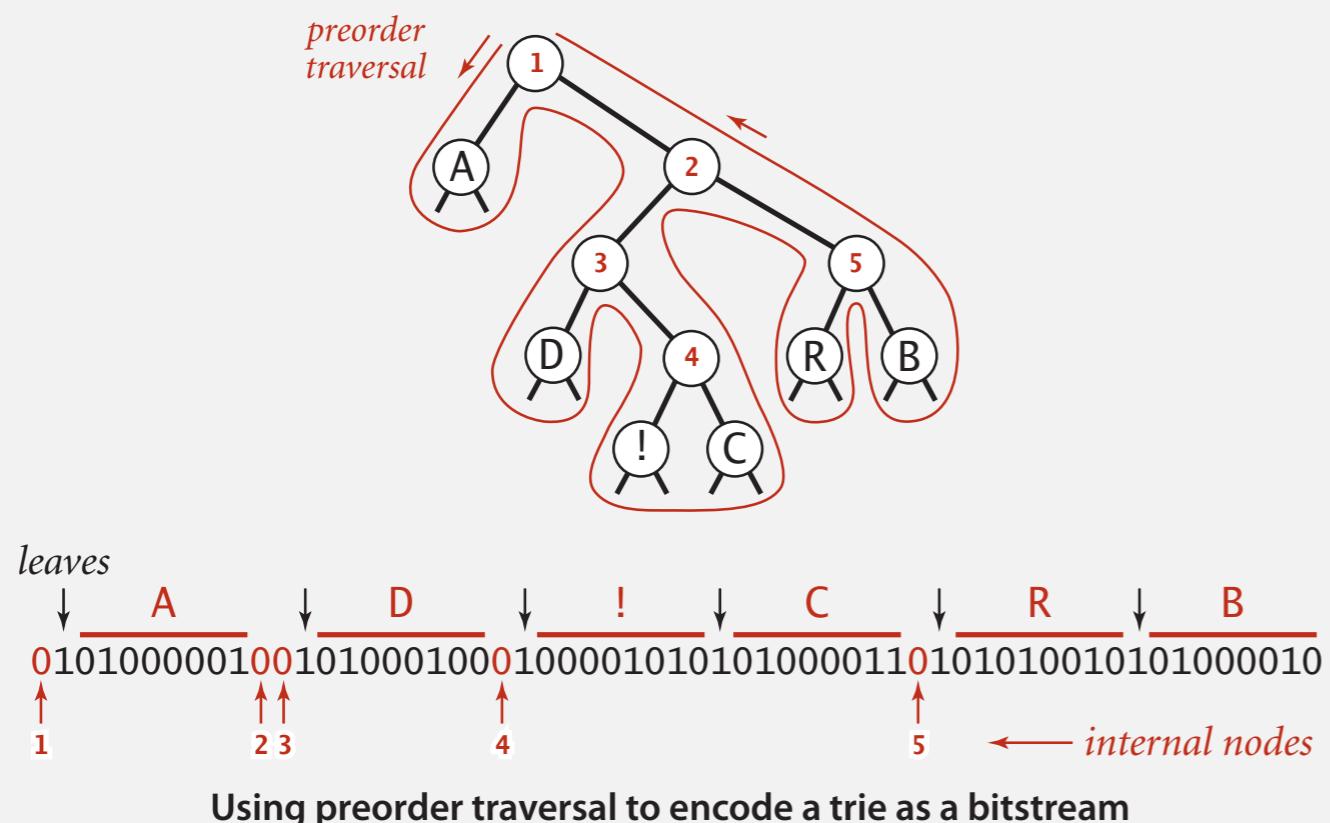
Prefix-free codes: how to transmit

Q. How to write the trie?

Prefix-free codes: how to transmit

Q. How to write the trie?

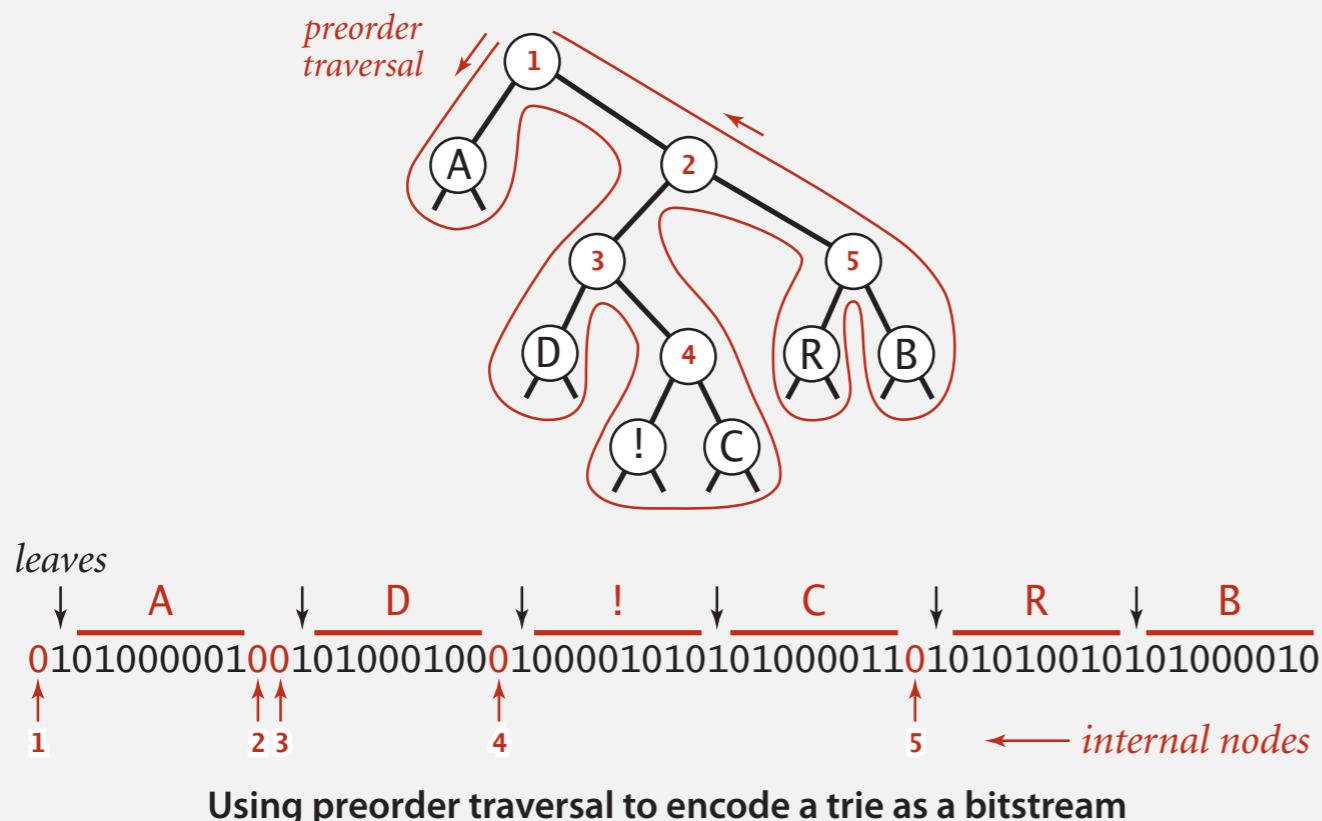
A. Write preorder traversal of trie; mark leaf and internal nodes with a bit.



Prefix-free codes: how to transmit

Q. How to write the trie?

A. Write preorder traversal of trie; mark leaf and internal nodes with a bit.

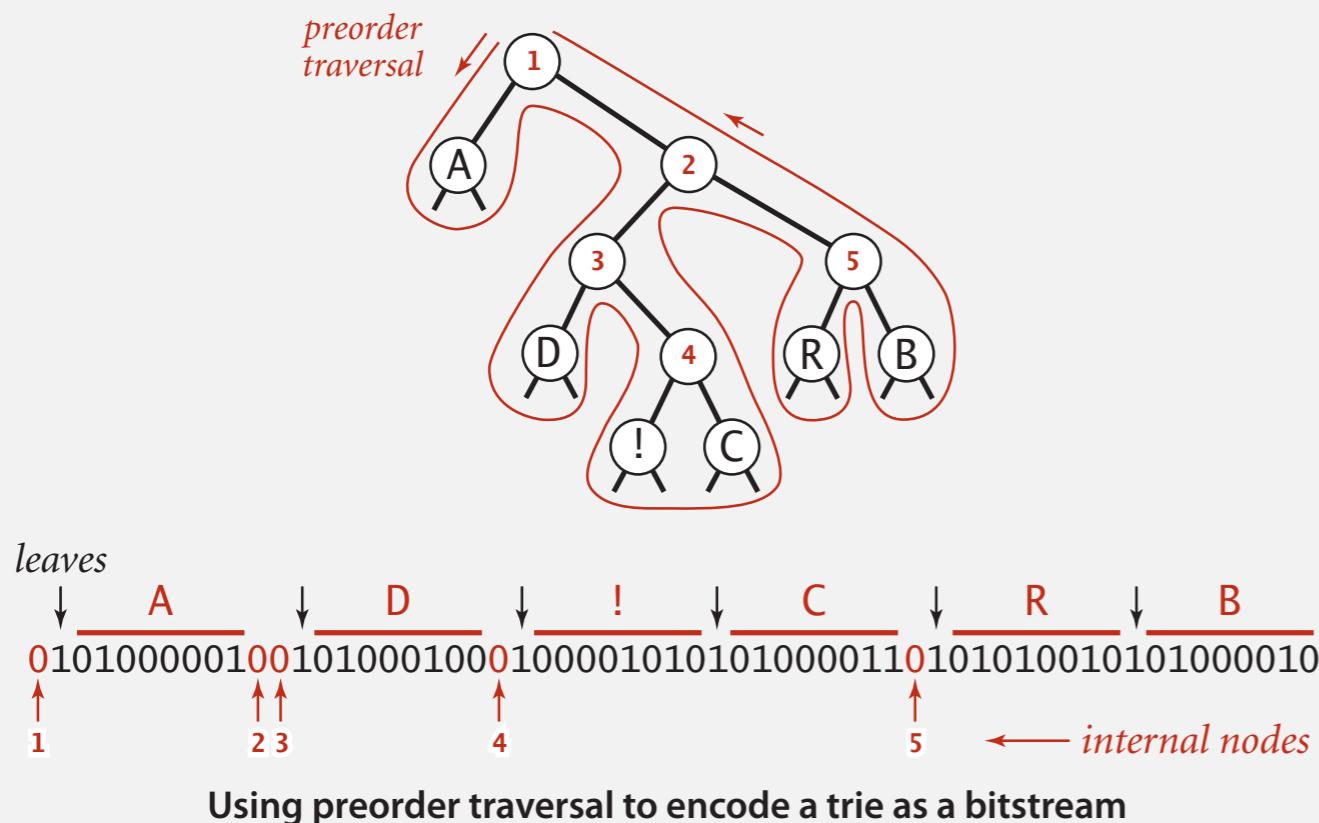


```
private static void writeTrie(Node x)
{
    if (x.isLeaf())
    {
        BinaryStdOut.write(true);
        BinaryStdOut.write(x.ch, 8);
        return;
    }
    BinaryStdOut.write(false);
    writeTrie(x.left);
    writeTrie(x.right);
}
```

Prefix-free codes: how to transmit

Q. How to write the trie?

A. Write preorder traversal of trie; mark leaf and internal nodes with a bit.

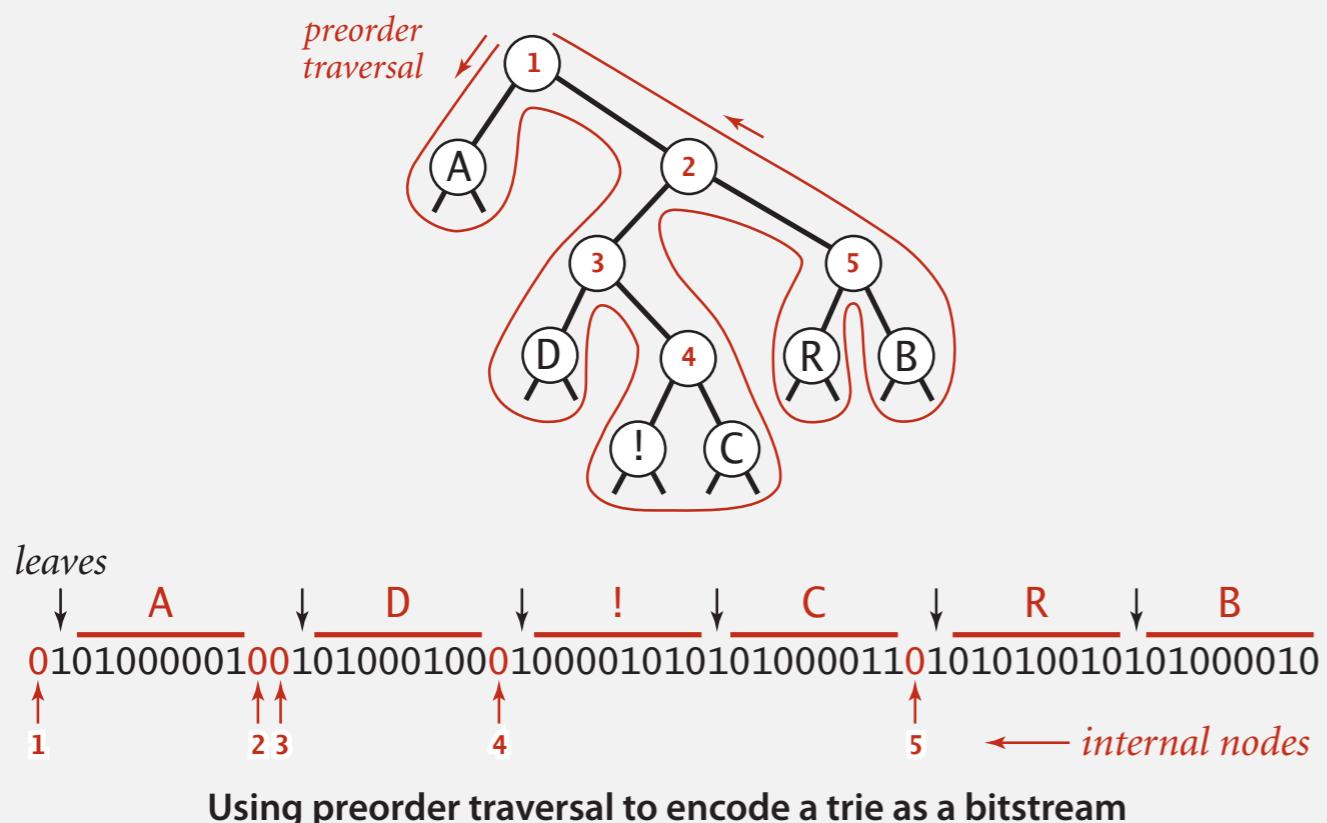


```
private static void writeTrie(Node x)
{
    if (x.isLeaf())
    {
        BinaryStdOut.write(true);
        BinaryStdOut.write(x.ch, 8);
        return;
    }
    BinaryStdOut.write(false);
    writeTrie(x.left);
    writeTrie(x.right);
}
```

Note. If message is long, overhead of transmitting trie is small.

Prefix-free codes: how to transmit

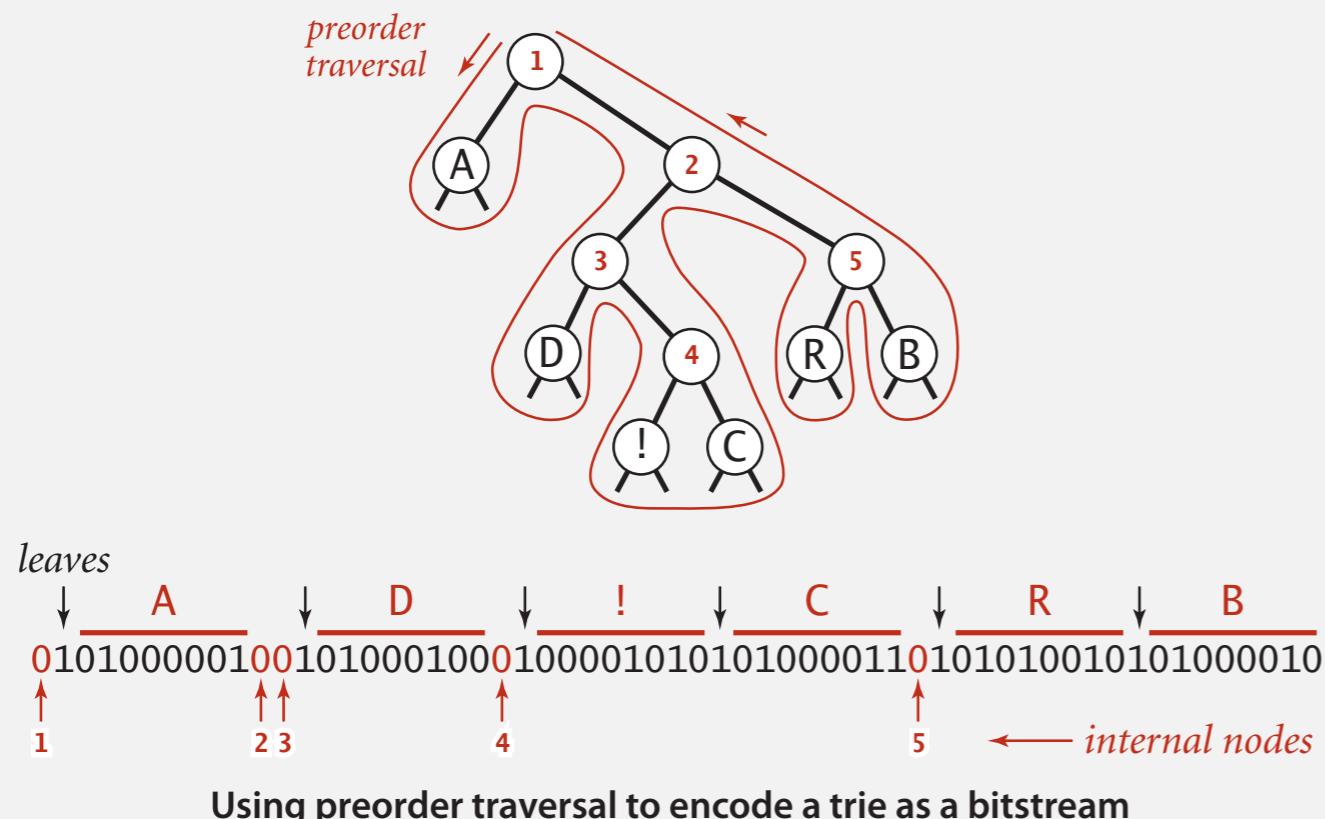
Q. How to read in the trie?



Prefix-free codes: how to transmit

Q. How to read in the trie?

A. Reconstruct from preorder traversal of trie.



```
private static Node readTrie()
{
    if (BinaryStdIn.readBoolean())
    {
        char c = BinaryStdIn.readChar(8);
        return new Node(c, 0, null, null);
    }
    Node x = readTrie();
    Node y = readTrie();
    return new Node('\\0', 0, x, y);
}
```

arbitrary value
(value not used with internal nodes)

Shannon-Fano codes

Q. How to find best prefix-free code?

Shannon-Fano codes

Q. How to find best prefix-free code?

Shannon-Fano algorithm:

- Partition symbols S into two subsets S_0 and S_1 of (roughly) equal freq.
- Codewords for symbols in S_0 start with 0; for symbols in S_1 start with 1.
- Recur in S_0 and S_1 .

char	freq	encoding
A	5	0...
C	1	0...

$S_0 = \text{codewords starting with 0}$

char	freq	encoding
B	2	1...
D	1	1...
R	2	1...
!	1	1...

$S_1 = \text{codewords starting with 1}$

Shannon-Fano codes

Q. How to find best prefix-free code?

Shannon-Fano algorithm:

- Partition symbols S into two subsets S_0 and S_1 of (roughly) equal freq.
- Codewords for symbols in S_0 start with 0; for symbols in S_1 start with 1.
- Recur in S_0 and S_1 .

char	freq	encoding
A	5	0...
C	1	0...

$S_0 = \text{codewords starting with 0}$

char	freq	encoding
B	2	1...
D	1	1...
R	2	1...
!	1	1...

$S_1 = \text{codewords starting with 1}$

Problem 1. How to divide up symbols?

Shannon-Fano codes

Q. How to find best prefix-free code?

Shannon-Fano algorithm:

- Partition symbols S into two subsets S_0 and S_1 of (roughly) equal freq.
- Codewords for symbols in S_0 start with 0; for symbols in S_1 start with 1.
- Recur in S_0 and S_1 .

char	freq	encoding
A	5	0...
C	1	0...

$S_0 = \text{codewords starting with 0}$

char	freq	encoding
B	2	1...
D	1	1...
R	2	1...
!	1	1...

$S_1 = \text{codewords starting with 1}$

Problem 1. How to divide up symbols?

Problem 2. Not optimal!

Huffman coding demo

- Count frequency for each character in input.

char	freq	encoding
A		
B		
C		
D		
R		
!		

input

ABRACADABRA!

Huffman coding demo

- Count frequency for each character in input.

char	freq	encoding
A	5	
B	2	
C	1	
D	1	
R	2	
!	1	

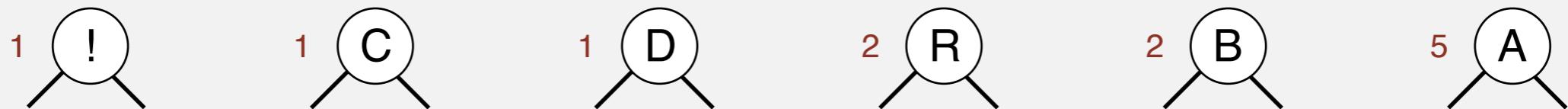
input

ABRACADABRA!

Huffman coding demo

- Start with one node corresponding to each character with weight equal to frequency.

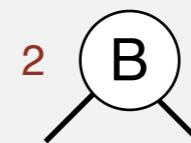
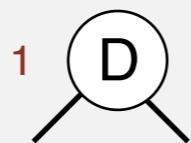
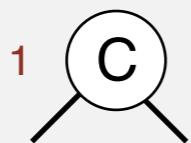
char	freq	encoding
A	5	
B	2	
C	1	
D	1	
R	2	
!	1	



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

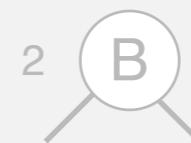
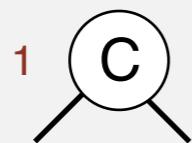
char	freq	encoding
A	5	
B	2	
C	1	
D	1	
R	2	
!	1	



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

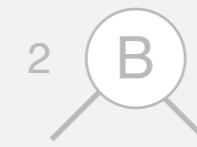
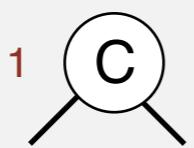
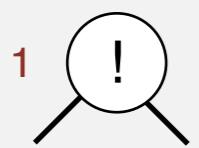
char	freq	encoding
A	5	
B	2	
C	1	
D	1	
R	2	
!	1	



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

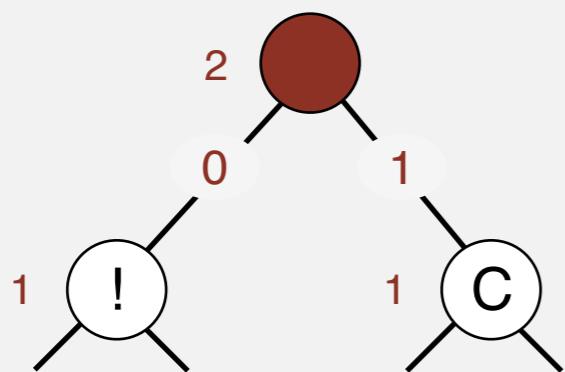
char	freq	encoding
A	5	
B	2	
C	1	
D	1	
R	2	
!	1	



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

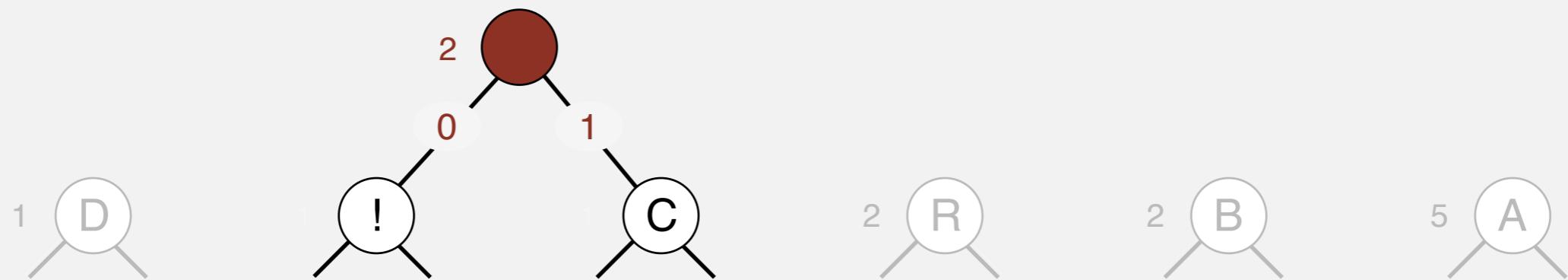
char	freq	encoding
A	5	
B	2	
C	1	1
D	1	
R	2	
!	1	0



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

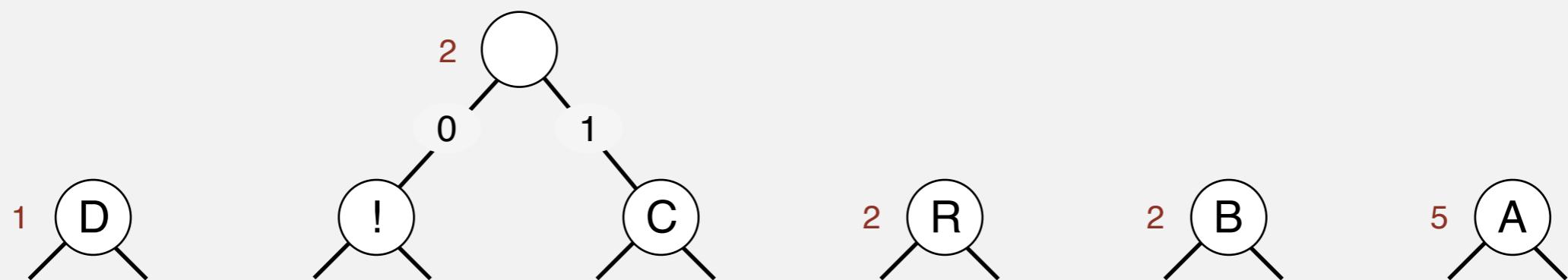
char	freq	encoding
A	5	
B	2	
C	1	1
D	1	
R	2	
!	1	0



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

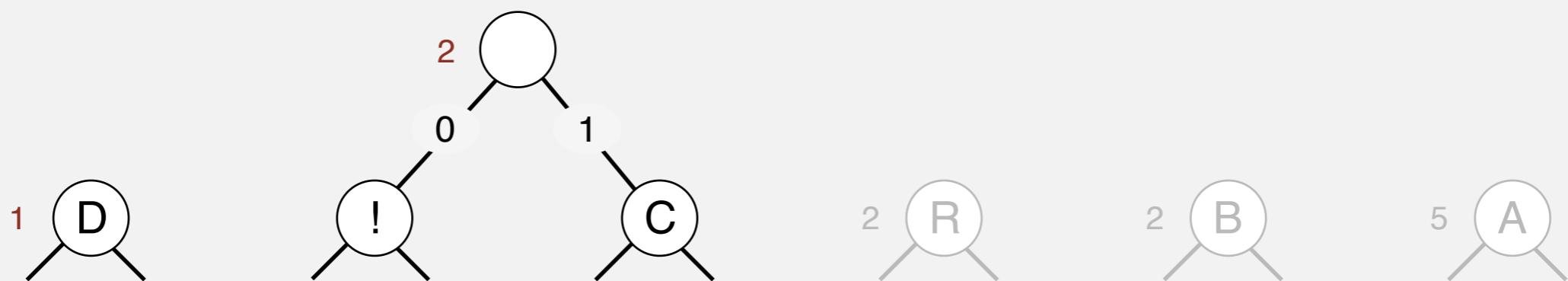
char	freq	encoding
A	5	
B	2	
C	1	1
D	1	
R	2	
!	1	0



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

char	freq	encoding
A	5	
B	2	
C	1	1
D	1	
R	2	
!	1	0



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

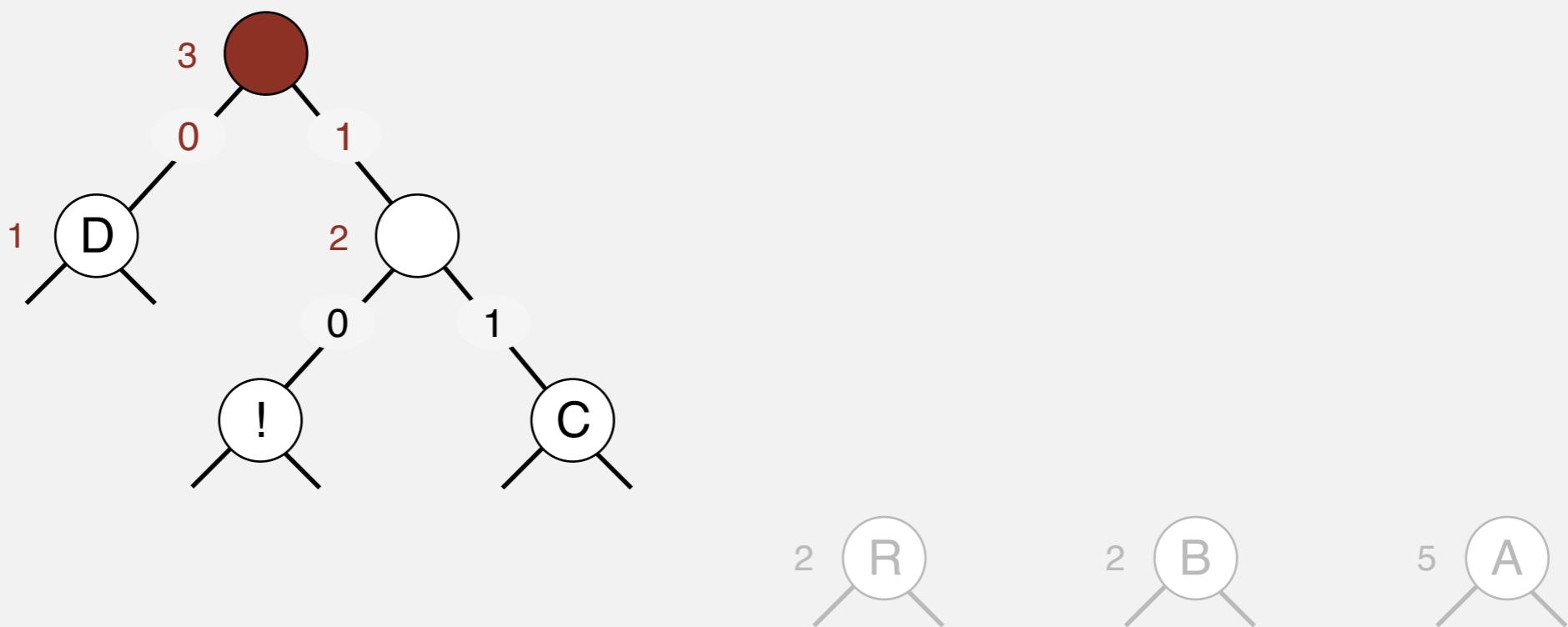
char	freq	encoding
A	5	
B	2	
C	1	1
D	1	
R	2	
!	1	0



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

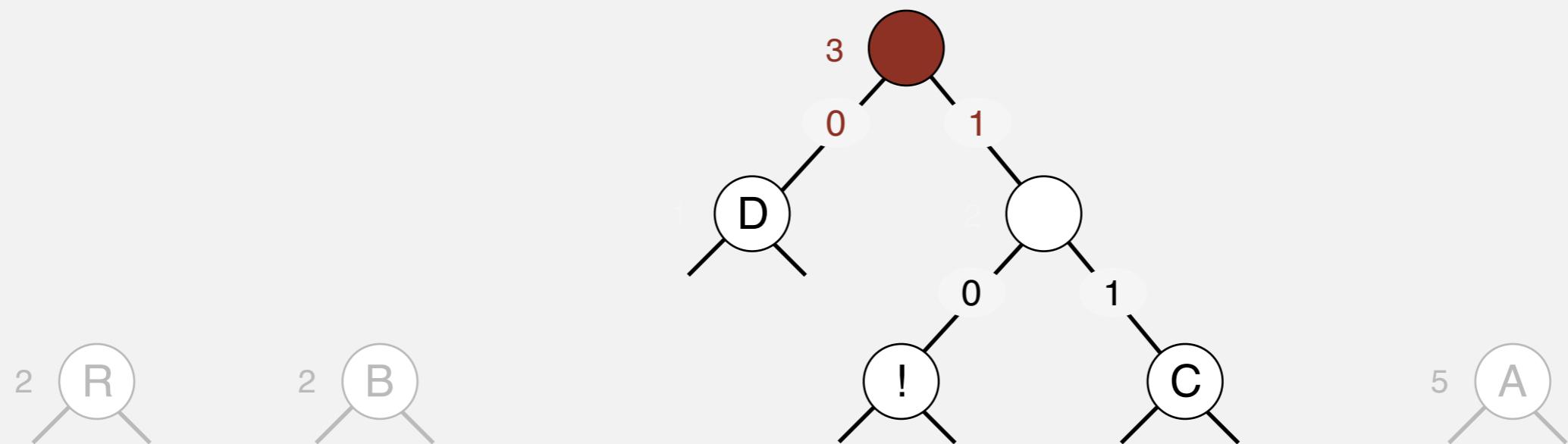
char	freq	encoding
A	5	
B	2	
C	1	1
D	1	0
R	2	
!	1	1 0



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

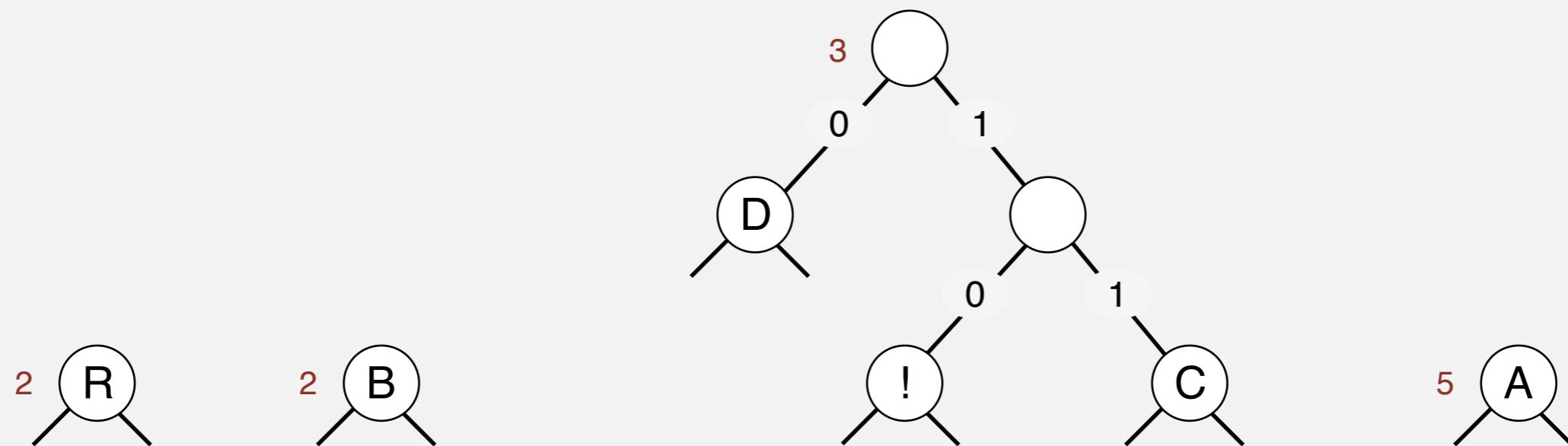
char	freq	encoding
A	5	
B	2	
C	1	1
D	1	0
R	2	
!	1	1 0



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

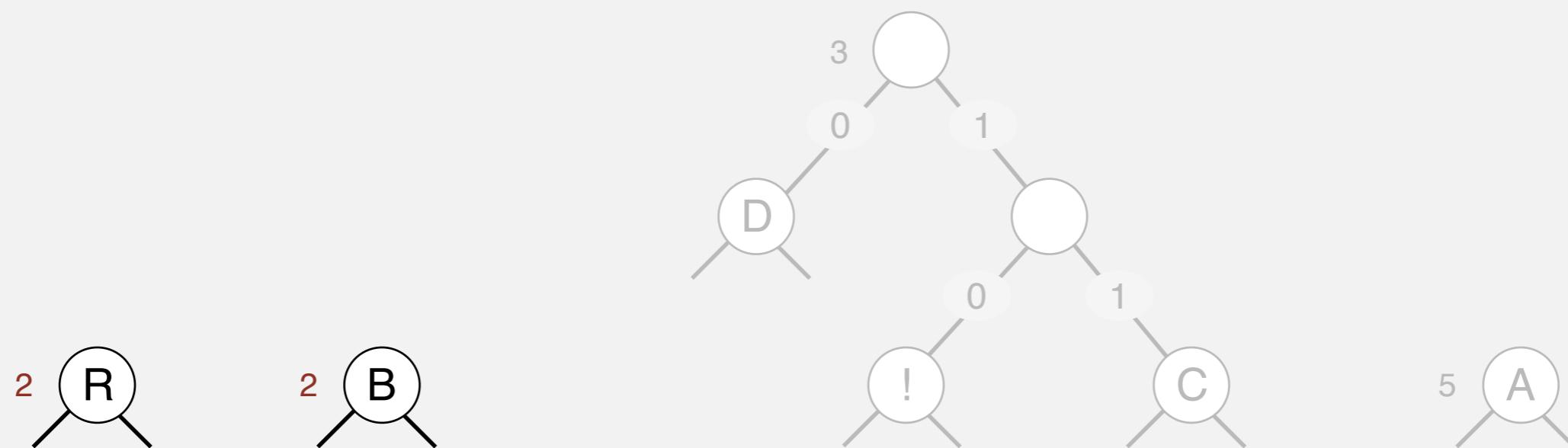
char	freq	encoding
A	5	
B	2	
C	1	1 1
D	1	0
R	2	
!	1	1 0



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

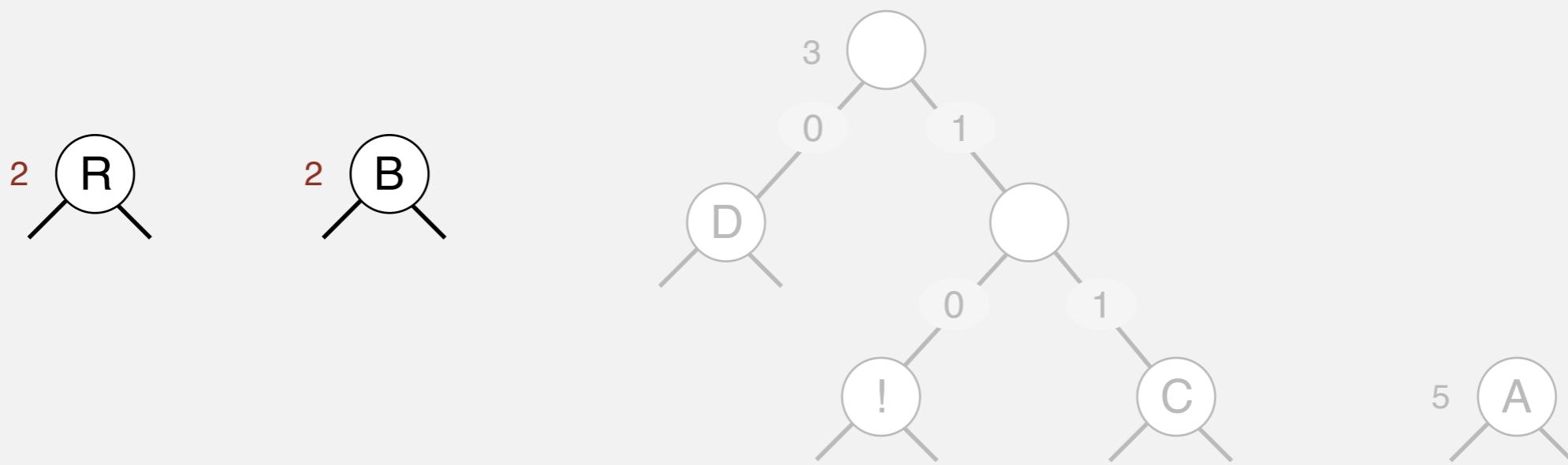
char	freq	encoding
A	5	
B	2	
C	1	1 1
D	1	0
R	2	
!	1	1 0



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

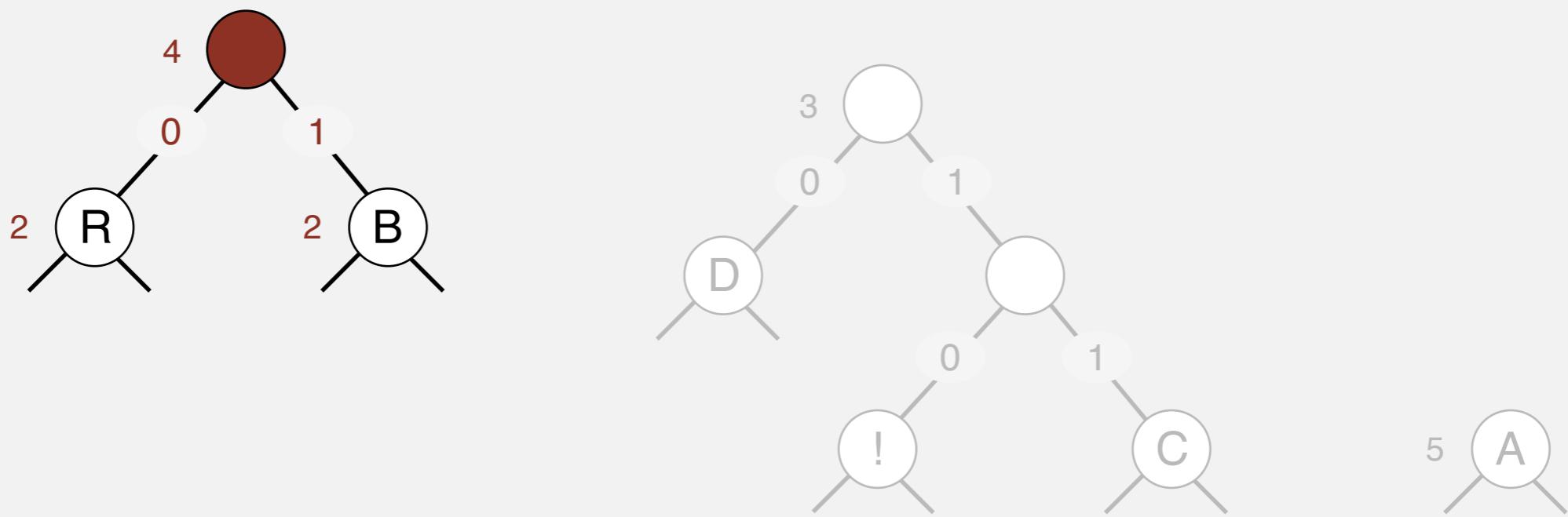
char	freq	encoding
A	5	
B	2	
C	1	1 1
D	1	0
R	2	
!	1	1 0



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

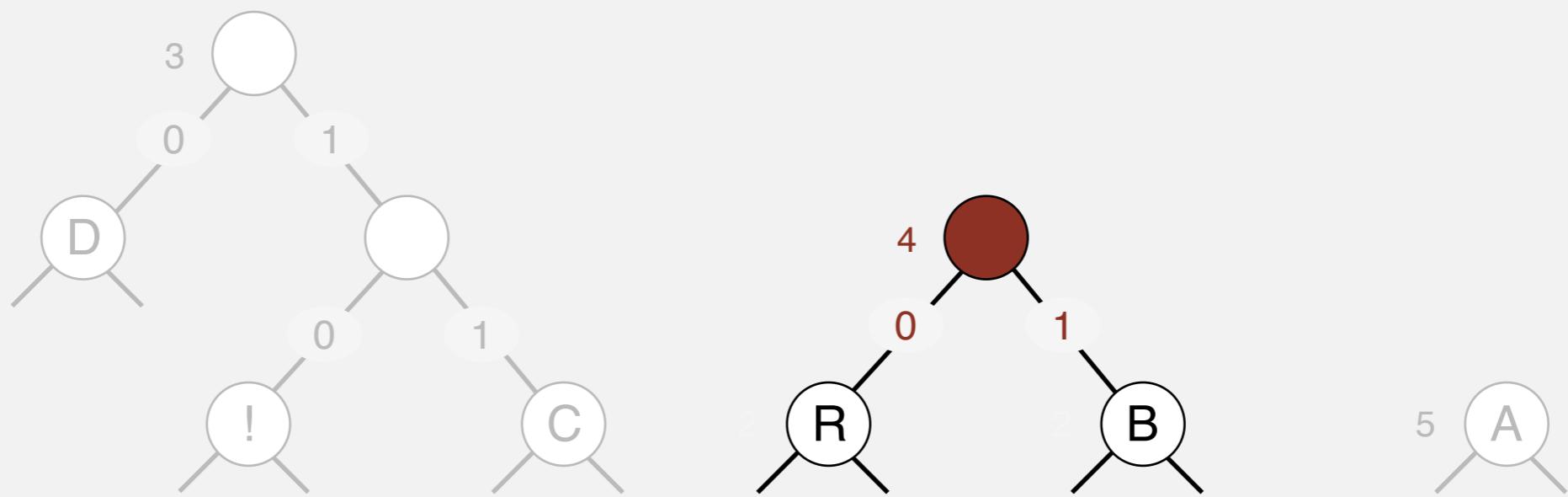
char	freq	encoding
A	5	
B	2	1
C	1	1 1
D	1	0
R	2	0
!	1	1 0



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

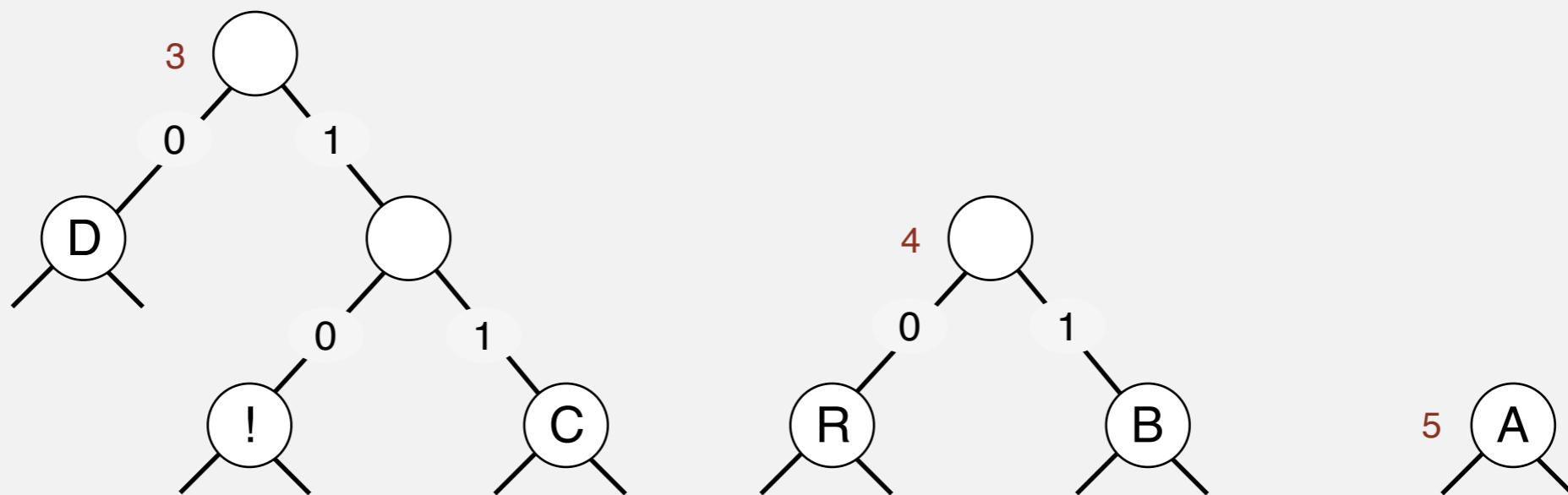
char	freq	encoding
A	5	
B	2	1
C	1	1 1
D	1	0
R	2	0
!	1	1 0



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

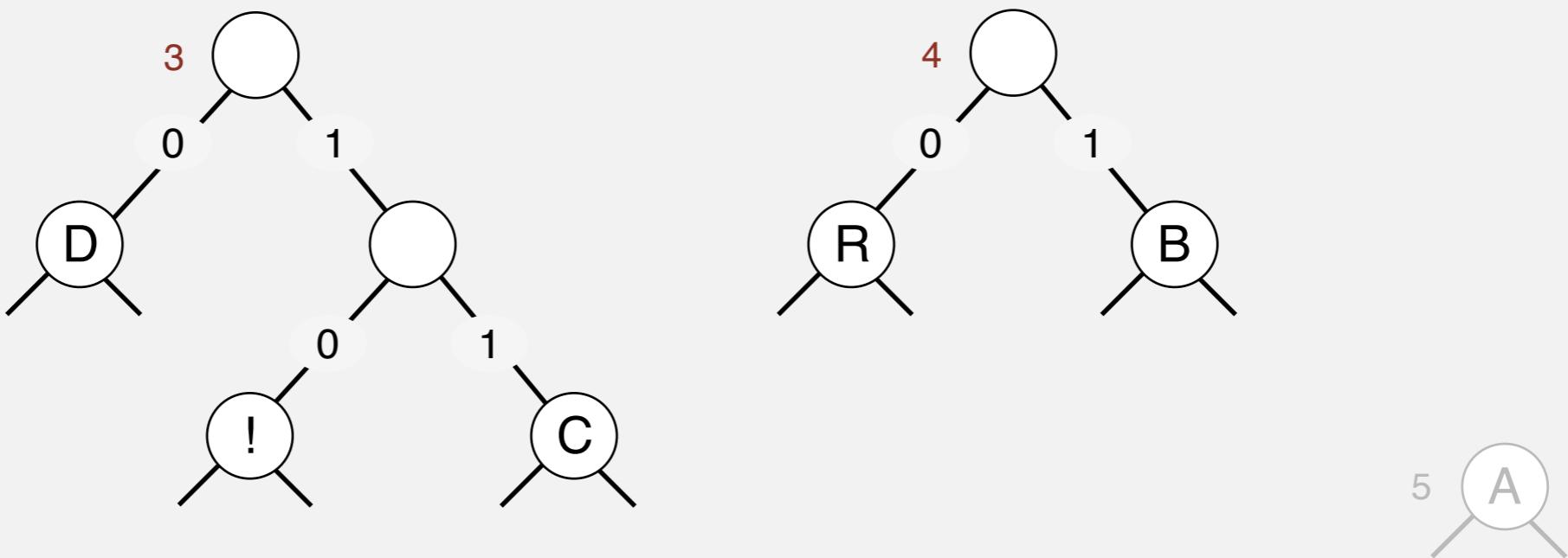
char	freq	encoding
A	5	
B	2	1
C	1	1 1
D	1	0
R	2	0
!	1	1 0



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

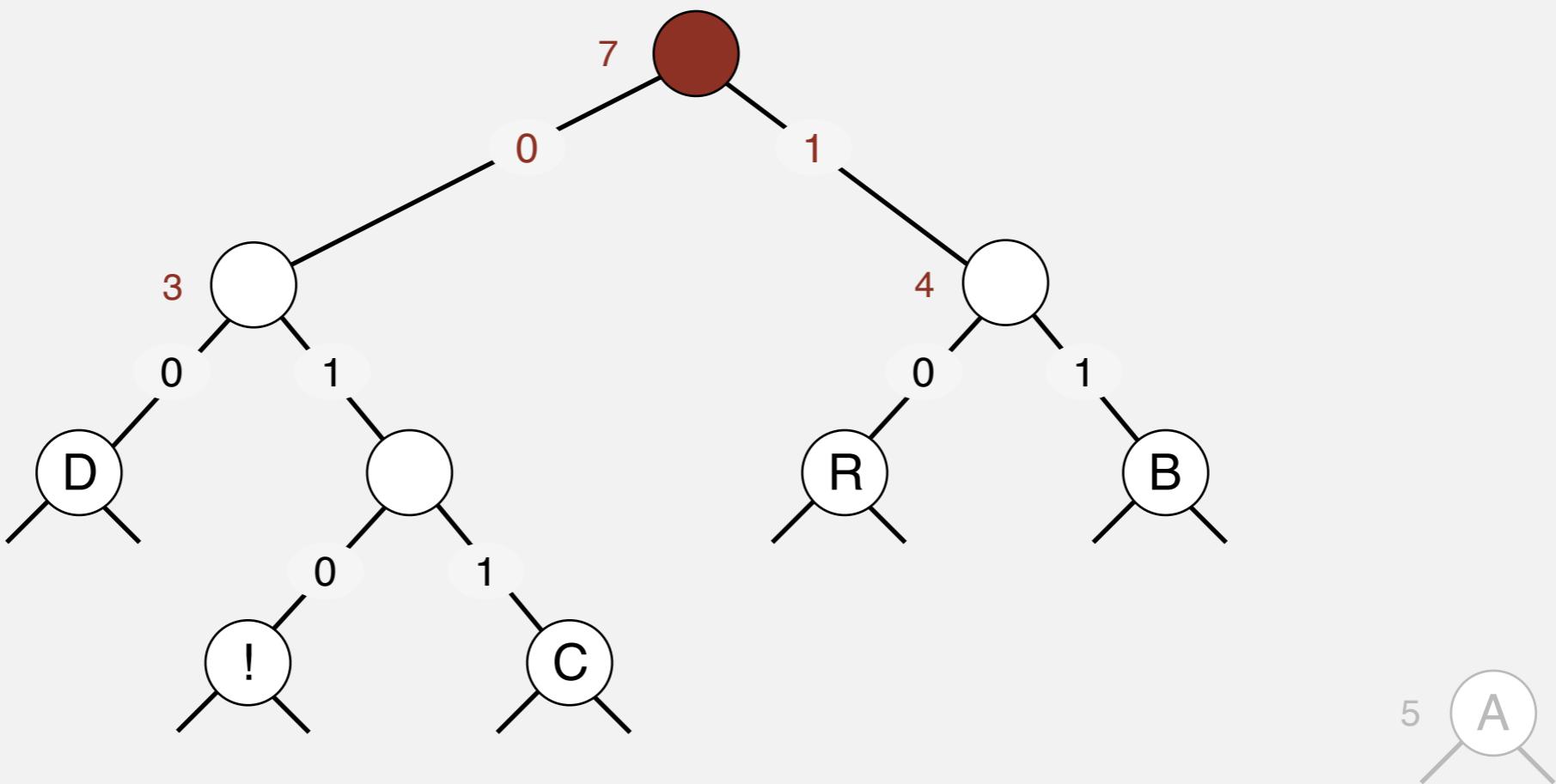
char	freq	encoding
A	5	
B	2	1
C	1	1 1
D	1	0
R	2	0
!	1	1 0



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

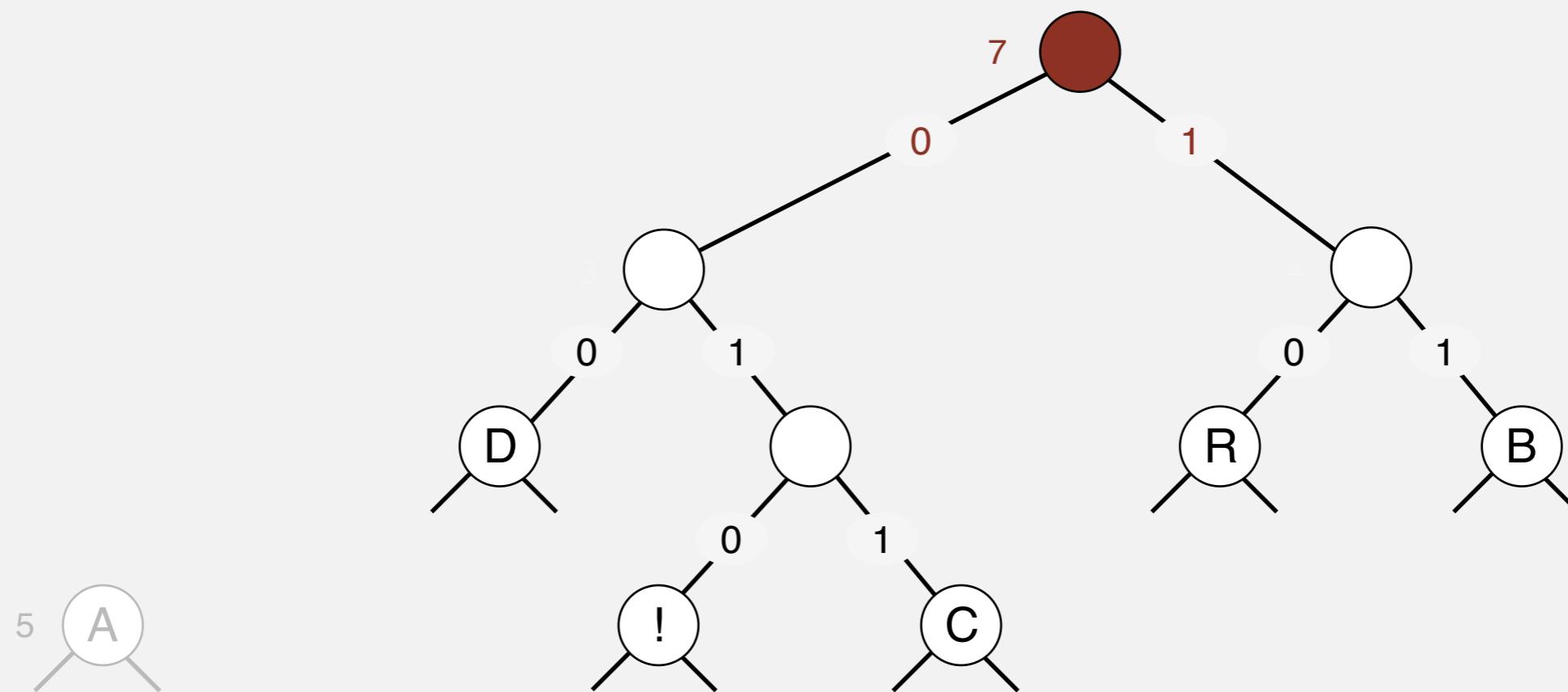
char	freq	encoding
A	5	
B	2	1 1
C	1	0 1 1
D	1	0 0
R	2	1 0
!	1	0 1 0



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

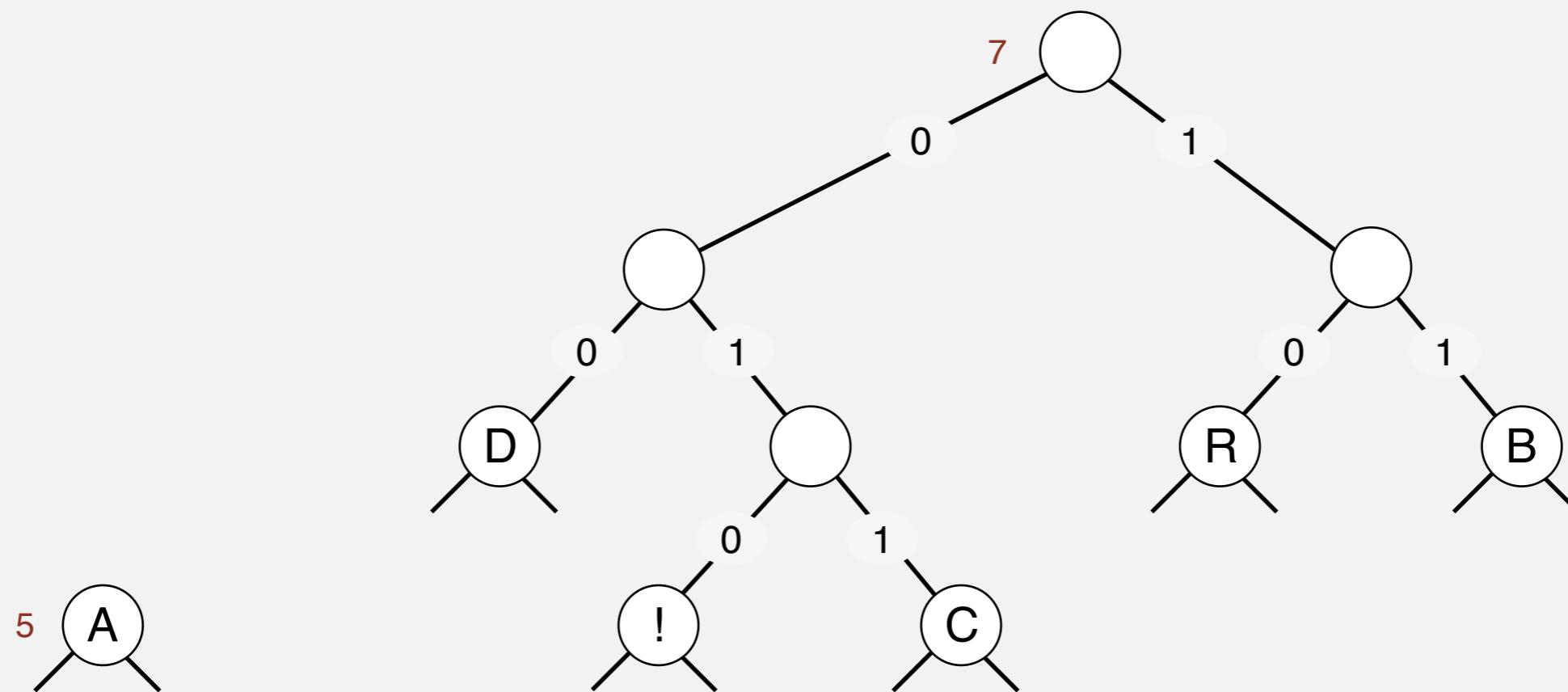
char	freq	encoding
A	5	
B	2	1 1
C	1	0 1 1
D	1	0 0
R	2	1 0
!	1	0 1 0



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

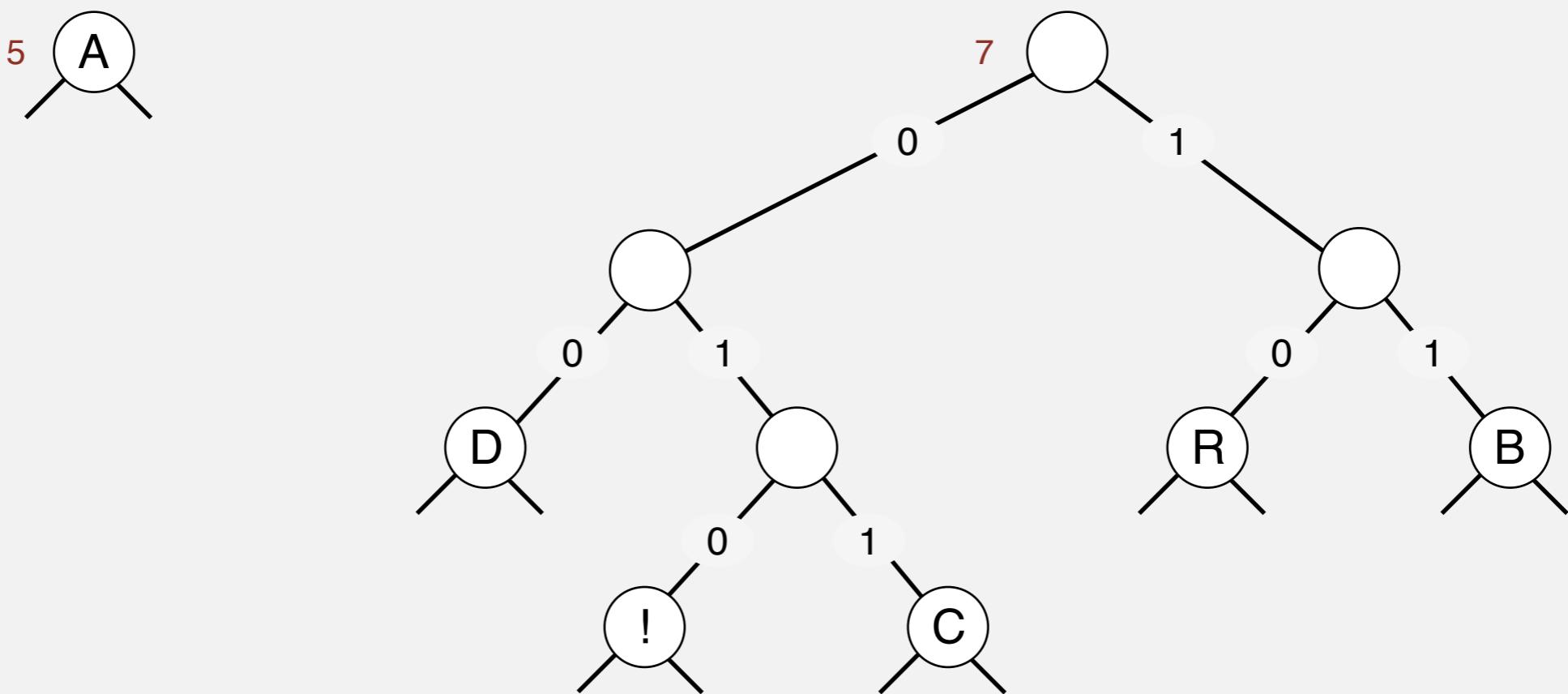
char	freq	encoding
A	5	
B	2	1 1
C	1	0 1 1
D	1	0 0
R	2	1 0
!	1	0 1 0



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

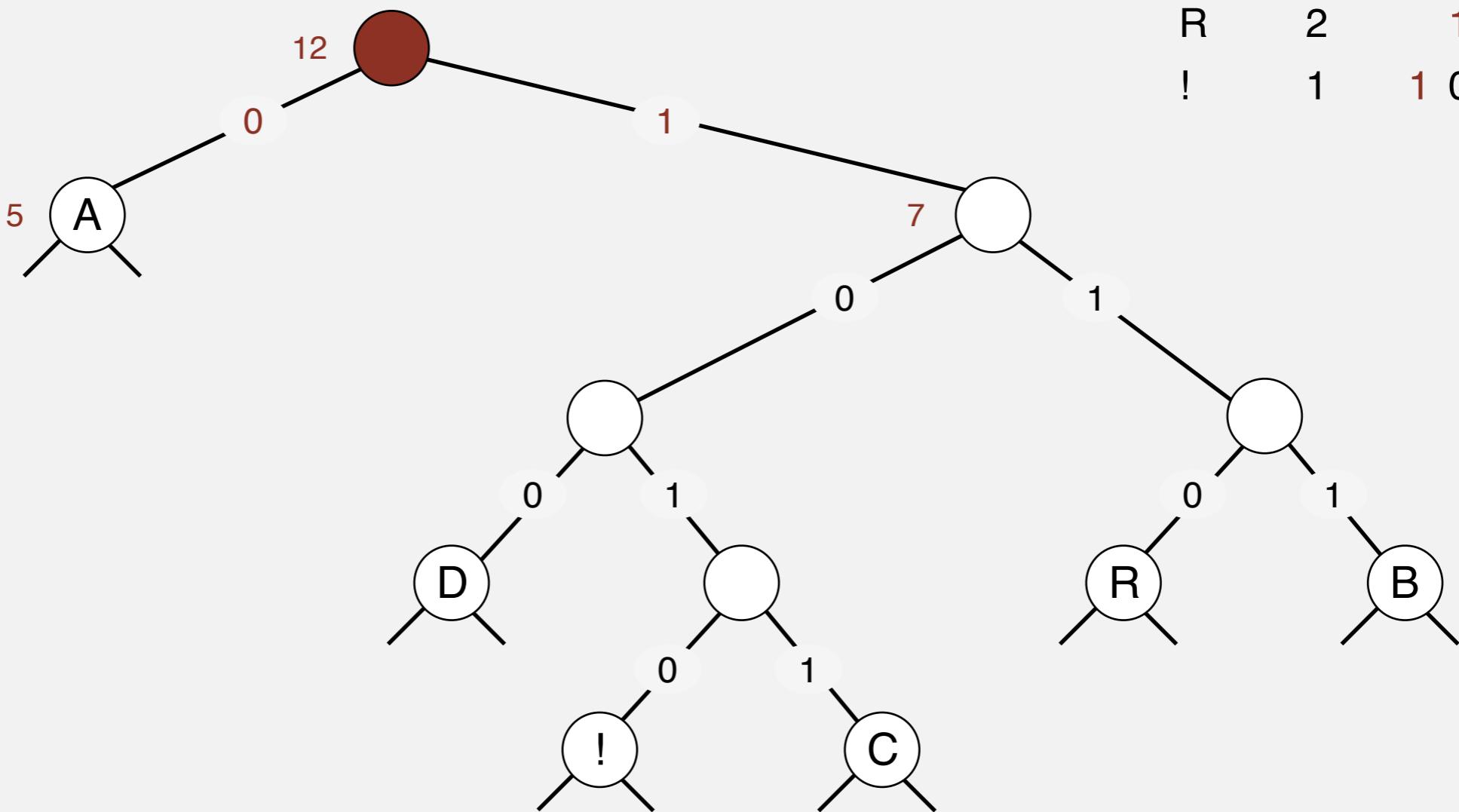
char	freq	encoding
A	5	
B	2	1 1
C	1	0 1 1
D	1	0 0
R	2	1 0
!	1	0 1 0



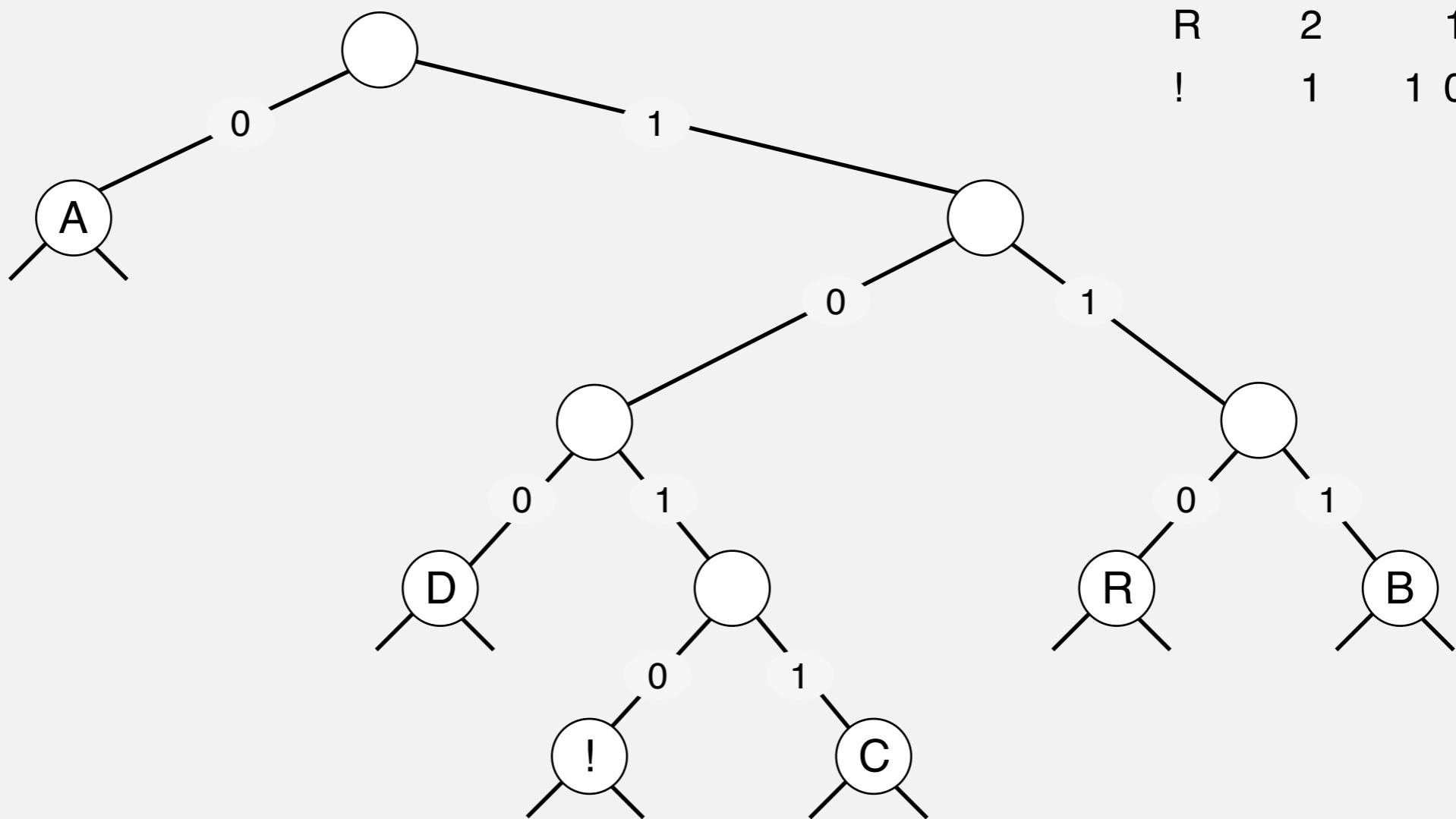
Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

char	freq	encoding
A	5	0
B	2	1 1 1
C	1	1 0 1 1
D	1	1 0 0
R	2	1 1 0
!	1	1 0 1 0



Huffman algorithm demo



Huffman codes

Q. How to find best prefix-free code?

Huffman algorithm:

- Count frequency $\text{freq}[i]$ for each char i in input.
- Start with one node corresponding to each char i (with weight $\text{freq}[i]$).
- Repeat until single trie formed:
 - select two tries with min weight $\text{freq}[i]$ and $\text{freq}[j]$
 - merge into single trie with weight $\text{freq}[i] + \text{freq}[j]$

Huffman codes

Q. How to find best prefix-free code?

Huffman algorithm:

- Count frequency $\text{freq}[i]$ for each char i in input.
- Start with one node corresponding to each char i (with weight $\text{freq}[i]$).
- Repeat until single trie formed:
 - select two tries with min weight $\text{freq}[i]$ and $\text{freq}[j]$
 - merge into single trie with weight $\text{freq}[i] + \text{freq}[j]$

Applications:



Constructing a Huffman encoding trie: Java implementation

```
private static Node buildTrie(int[] freq) {  
    MinPQ<Node> pq = new MinPQ<Node>();  
  
    for (char i = 0; i < R; i++)  
        if (freq[i] > 0)  
            pq.insert(new Node(i, freq[i], null, null));
```

initialize PQ with
singleton tries

```
    while (pq.size() > 1) {  
        Node x = pq.delMin();  
        Node y = pq.delMin();  
  
        Node parent = new Node('\\0', x.freq + y.freq, x, y);  
        pq.insert(parent);  
    }  
  
    return pq.delMin();  
}
```

merge two
smallest tries

not used for
internal nodes

two subtrees

total frequency

Huffman encoding summary

Proposition. [Huffman 1950s] Huffman algorithm produces an optimal prefix-free code.

Pf. See textbook.

↑
no prefix-free code
uses fewer bits

Huffman encoding summary

Proposition. [Huffman 1950s] Huffman algorithm produces an optimal prefix-free code.

Pf. See textbook.

↑
no prefix-free code
uses fewer bits

Implementation.

- Pass 1: tabulate char frequencies and build trie.
- Pass 2: encode file by traversing trie or lookup table.

Huffman encoding summary

Proposition. [Huffman 1950s] Huffman algorithm produces an optimal prefix-free code.

Pf. See textbook.

↑
no prefix-free code
uses fewer bits

Implementation.

- Pass 1: tabulate char frequencies and build trie.
- Pass 2: encode file by traversing trie or lookup table.

Running time. Using a binary heap $\Rightarrow N + R \log R$.

↑ ↑
input size alphabet size

Huffman encoding summary

Proposition. [Huffman 1950s] Huffman algorithm produces an optimal prefix-free code.

Pf. See textbook.

↑
no prefix-free code
uses fewer bits

Implementation.

- Pass 1: tabulate char frequencies and build trie.
- Pass 2: encode file by traversing trie or lookup table.

Running time. Using a binary heap $\Rightarrow N + R \log R$.

↑ ↑
input size alphabet size

Q. Can we do better? [stay tuned]

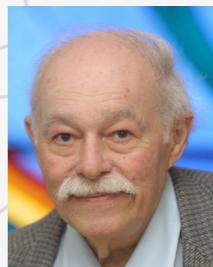
Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

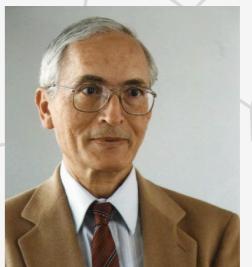
<http://algs4.cs.princeton.edu>

5.5 DATA COMPRESSION

- ▶ introduction
- ▶ run-length coding
- ▶ Huffman compression
- ▶ LZW compression



Abraham Lempel



Jacob Ziv

Statistical methods

Static model. Same model for all texts.

- Fast.
- Not optimal: different texts have different statistical properties.
- Ex: ASCII, Morse code.

Statistical methods

Static model. Same model for all texts.

- Fast.
- Not optimal: different texts have different statistical properties.
- Ex: ASCII, Morse code.

Dynamic model. Generate model based on text.

- Preliminary pass needed to generate model.
- Must transmit the model.
- Ex: Huffman code.

Statistical methods

Static model. Same model for all texts.

- Fast.
- Not optimal: different texts have different statistical properties.
- Ex: ASCII, Morse code.

Dynamic model. Generate model based on text.

- Preliminary pass needed to generate model.
- Must transmit the model.
- Ex: Huffman code.

Adaptive model. Progressively learn and update model as you read text.

- More accurate modeling produces better compression.
- Decoding must start from beginning.
- Ex: LZW.

LZW compression demo

input A B R A C A D A B R A B R A B R A

matches

value

LZW compression for A B R A C A D A B R A B R A B R A

key	value
:	:
A	41
B	42
C	43
D	44
:	:

codeword table

LZW compression demo

input A B R A C A D A B R A B R A B R A

matches A

value 41

LZW compression for A B R A C A D A B R A B R A B R A

key	value
:	:
A	41
B	42
C	43
D	44
:	:

codeword table

LZW compression demo

input A B R A C A D A B R A B R A B R A
matches A
value 41

LZW compression for A B R A C A D A B R A B R A B R A

key	value
:	:
A	41
B	42
C	43
D	44
:	:

key	value
AB	81

codeword table

LZW compression demo

input	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
matches	A	B															
value	41	42															

LZW compression for A B R A C A D A B R A B R A B R A

key	value
:	:
A	41
B	42
C	43
D	44
:	:
AB	81

codeword table

LZW compression demo

input	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
matches	A	B															
value	41	42															

LZW compression for A B R A C A D A B R A B R A B R A

key	value
:	:
A	41
B	42
C	43
D	44
:	:

key	value
AB	81
BR	82

codeword table

LZW compression demo

input	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
matches	A	B	R														
value	41	42	52														

LZW compression for A B R A C A D A B R A B R A

key	value
:	:
A	41
B	42
C	43
D	44
:	:

key	value
AB	81
BR	82

codeword table

LZW compression demo

input	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
matches	A	B	R														
value	41	42	52														

LZW compression for A B R A C A D A B R A B R A

key	value
:	:
A	41
B	42
C	43
D	44
:	:

key	value
AB	81
BR	82
RA	83

codeword table

LZW compression demo

input	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
matches	A	B	R	A													
value	41	42	52	41													

LZW compression for A B R A C A D A B R A B R A

key	value
:	:
A	41
B	42
C	43
D	44
:	:

key	value
AB	81
BR	82
RA	83

codeword table

LZW compression demo

input	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
matches	A	B	R	A													
value	41	42	52	41													

LZW compression for A B R A C A D A B R A B R A B R A

key	value	key	value
:	:	AB	81
A	41	BR	82
B	42	RA	83
C	43	AC	84
D	44		
:	:		

codeword table

LZW compression demo

input	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
matches	A	B	R	A	C												
value	41	42	52	41	43												

LZW compression for A B R A C A D A B R A B R A B R A

key	value	key	value
:	:	AB	81
A	41	BR	82
B	42	RA	83
C	43	AC	84
D	44		
:	:		

codeword table

LZW compression demo

input	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
matches	A	B	R	A	C												
value	41	42	52	41	43												

LZW compression for A B R A C A D A B R A B R A B R A

key	value	key	value
:	:	AB	81
A	41	BR	82
B	42	RA	83
C	43	AC	84
D	44	CA	85
:	:		

codeword table

LZW compression demo

input	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
matches	A	B	R	A	C	A											
value	41	42	52	41	43	41											

LZW compression for A B R A C A D A B R A B R A

key	value	key	value
:	:	AB	81
A	41	BR	82
B	42	RA	83
C	43	AC	84
D	44	CA	85
:	:		

codeword table

LZW compression demo

input	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
matches	A	B	R	A	C	A											
value	41	42	52	41	43	41											

LZW compression for A B R A C A D A B R A B R A

key	value	key	value
:	:	AB	81
A	41	BR	82
B	42	RA	83
C	43	AC	84
D	44	CA	85
:	:	AD	86

codeword table

LZW compression demo

input	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
matches	A	B	R	A	C	A	D										
value	41	42	52	41	43	41	44										

LZW compression for A B R A C A D A B R A B R A

key	value	key	value
:	:	AB	81
A	41	BR	82
B	42	RA	83
C	43	AC	84
D	44	CA	85
:	:	AD	86

codeword table

LZW compression demo

input	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
matches	A	B	R	A	C	A	D										
value	41	42	52	41	43	41	44										

LZW compression for A B R A C A D A B R A B R A

key	value	key	value	key	value
:	:	AB	81	DA	87
A	41	BR	82		
B	42	RA	83		
C	43	AC	84		
D	44	CA	85		
:	:	AD	86		

codeword table

LZW compression demo

input	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
matches	A	B	R	A	C	A	D	AB									
value	41	42	52	41	43	41	44	81									

LZW compression for A B R A C A D A B R A B R A

key	value	key	value	key	value
:	:	AB	81	DA	87
A	41	BR	82		
B	42	RA	83		
C	43	AC	84		
D	44	CA	85		
:	:	AD	86		

codeword table

LZW compression demo

input	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
matches	A	B	R	A	C	A	D	AB									
value	41	42	52	41	43	41	44	81									

LZW compression for A B R A C A D A B R A B R A B R A

key	value	key	value	key	value
:	:	AB	81	DA	87
A	41	BR	82	ABR	88
B	42	RA	83		
C	43	AC	84		
D	44	CA	85		
:	:	AD	86		

codeword table

LZW compression demo

input	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
matches	A	B	R	A	C	A	D	AB		RA							
value	41	42	52	41	43	41	44	81		83							

LZW compression for A B R A C A D A B R A B R A B R A

key	value	key	value	key	value
:	:	AB	81	DA	87
A	41	BR	82	ABR	88
B	42	RA	83		
C	43	AC	84		
D	44	CA	85		
:	:	AD	86		

codeword table

LZW compression demo

input	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
matches	A	B	R	A	C	A	D	AB		RA							
value	41	42	52	41	43	41	44	81		83							

LZW compression for A B R A C A D A B R A B R A B R A

key	value	key	value	key	value
:	:	AB	81	DA	87
A	41	BR	82	ABR	88
B	42	RA	83	RAB	89
C	43	AC	84		
D	44	CA	85		
:	:	AD	86		

codeword table

LZW compression demo

input	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
matches	A	B	R	A	C	A	D	AB	RA	BR							
value	41	42	52	41	43	41	44	81	83	82							

LZW compression for A B R A C A D A B R A B R A B R A

key	value	key	value	key	value
:	:	AB	81	DA	87
A	41	BR	82	ABR	88
B	42	RA	83	RAB	89
C	43	AC	84		
D	44	CA	85		
:	:	AD	86		

codeword table

LZW compression demo

input	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
matches	A	B	R	A	C	A	D	AB	RA	BR							
value	41	42	52	41	43	41	44	81	83	82							

LZW compression for A B R A C A D A B R A B R A B R A

key	value	key	value	key	value
:	:	AB	81	DA	87
A	41	BR	82	ABR	88
B	42	RA	83	RAB	89
C	43	AC	84	BRA	8A
D	44	CA	85		
:	:	AD	86		

codeword table

LZW compression demo

input	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
matches	A	B	R	A	C	A	D	AB	RA	BR	ABR						
value	41	42	52	41	43	41	44	81	83	82	88						

LZW compression for A B R A C A D A B R A B R A B R A

key	value	key	value	key	value
:	:	AB	81	DA	87
A	41	BR	82	ABR	88
B	42	RA	83	RAB	89
C	43	AC	84	BRA	8A
D	44	CA	85		
:	:	AD	86		

codeword table

LZW compression demo

input	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
matches	A	B	R	A	C	A	D	AB	RA	BR	ABR						
value	41	42	52	41	43	41	44	81	83	82	88						

LZW compression for A B R A C A D A B R A B R A B R A

key	value	key	value	key	value
:	:	AB	81	DA	87
A	41	BR	82	ABR	88
B	42	RA	83	RAB	89
C	43	AC	84	BRA	8A
D	44	CA	85	ABRA	8B
:	:	AD	86		

codeword table

LZW compression demo

input	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
matches	A	B	R	A	C	A	D	AB	RA	BR	ABR						A
value	41	42	52	41	43	41	44	81	83	82	88						41

LZW compression for A B R A C A D A B R A B R A B R A

key	value	key	value	key	value
:	:	AB	81	DA	87
A	41	BR	82	ABR	88
B	42	RA	83	RAB	89
C	43	AC	84	BRA	8A
D	44	CA	85	ABRA	8B
:	:	AD	86		

codeword table

LZW compression demo

input	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
matches	A	B	R	A	C	A	D	AB	RA	BR	ABR						A
value	41	42	52	41	43	41	44	81	83	82	88				41	80	

LZW compression for A B R A C A D A B R A B R A B R A

key	value	key	value	key	value
:	:	AB	81	DA	87
A	41	BR	82	ABR	88
B	42	RA	83	RAB	89
C	43	AC	84	BRA	8A
D	44	CA	85	ABRA	8B
:	:	AD	86		

codeword table

Lempel-Ziv-Welch compression

LZW compression.

- Create ST associating W -bit codewords with string keys.
- Initialize ST with codewords for single-char keys.
- Find longest string s in ST that is a prefix of unscanned part of input.
- Write the W -bit codeword associated with s .
- Add $s + c$ to ST, where c is next char in the input.

Lempel-Ziv-Welch compression

LZW compression.

- Create ST associating W -bit codewords with string keys.
- Initialize ST with codewords for single-char keys.
- Find longest string s in ST that is a prefix of unscanned part of input.
- Write the W -bit codeword associated with s .
- Add $s + c$ to ST, where c is next char in the input.

Q. How to represent LZW compression code table?

Lempel-Ziv-Welch compression

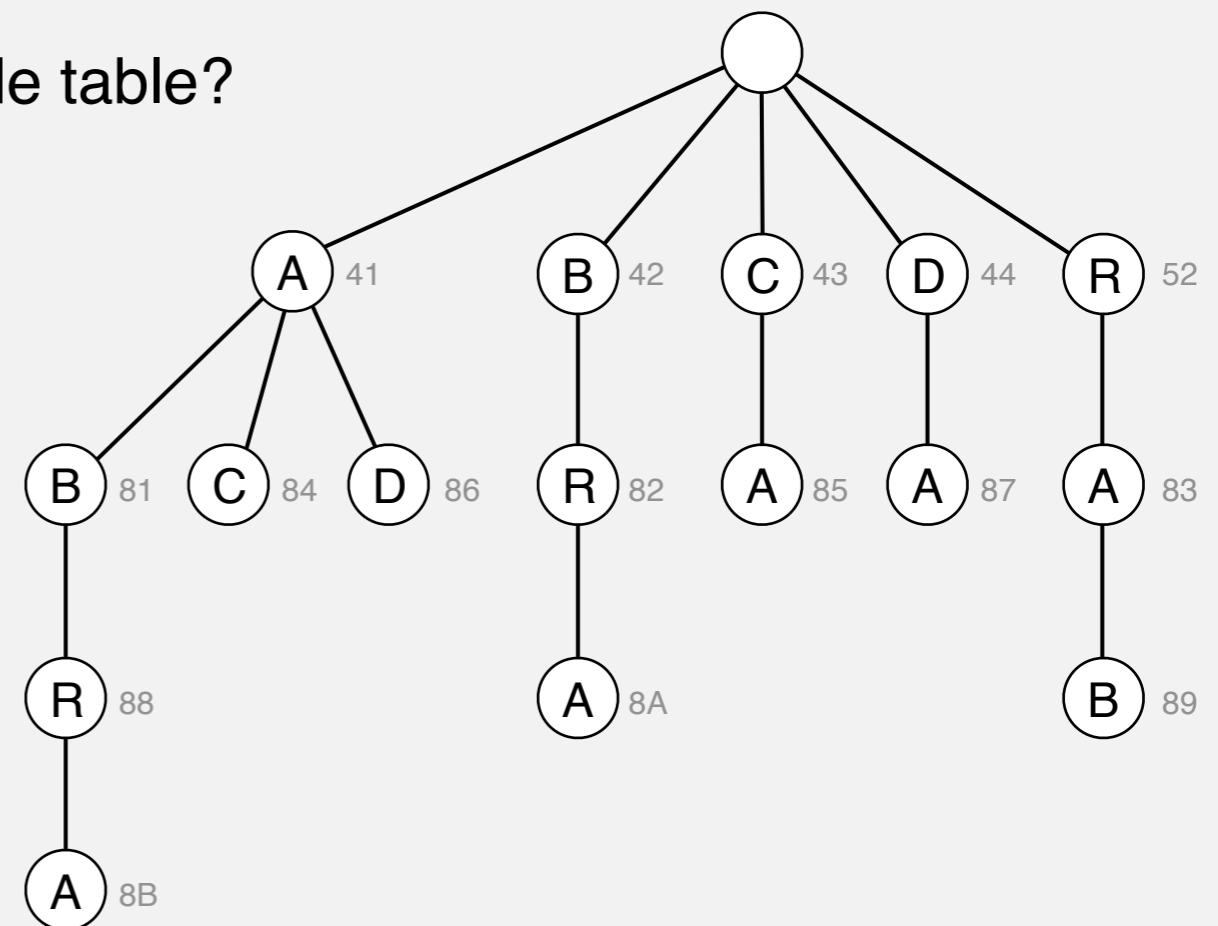
LZW compression.

- Create ST associating W -bit codewords with string keys.
- Initialize ST with codewords for single-char keys.
- Find longest string s in ST that is a prefix of unscanned part of input.
- Write the W -bit codeword associated with s .
- Add $s + c$ to ST, where c is next char in the input.

longest prefix match

Q. How to represent LZW compression code table?

A. A trie to support longest prefix match.



LZW expansion demo

value 41 42 52 41 43 41 44 81 83 82 88 41 80

output

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

key	value
:	:
41	A
42	B
43	C
44	D
:	:

codeword table

LZW expansion demo

value 41 42 52 41 43 41 44 81 83 82 88 41 80

output A

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

key	value
:	:
41	A
42	B
43	C
44	D
:	:

codeword table

LZW expansion demo

value	41	42	52	41	43	41	44	81	83	82	88	41	80
output	A	B											

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

key	value
:	:
41	A
42	B
43	C
44	D
:	:

codeword table

LZW expansion demo

value	41	42	52	41	43	41	44	81	83	82	88	41	80
output	A	B											

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

key	value	key	value
:	:	81	AB
41	A		
42	B		
43	C		
44	D		
:	:		

codeword table

LZW expansion demo

value	41	42	52	41	43	41	44	81	83	82	88	41	80
output	A	B	R										

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

key	value	key	value
:	:	81	AB
41	A		
42	B		
43	C		
44	D		
:	:		

codeword table

LZW expansion demo

value	41	42	52	41	43	41	44	81	83	82	88	41	80
output	A	B	R										

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

key	value	key	value
:	:	81	AB
41	A	82	BR
42	B		
43	C		
44	D		
:	:		

codeword table

LZW expansion demo

value	41	42	52	41	43	41	44	81	83	82	88	41	80
output	A	B	R	A									

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

key	value	key	value
:	:	81	AB
41	A	82	BR
42	B		
43	C		
44	D		
:	:		

codeword table

LZW expansion demo

value	41	42	52	41	43	41	44	81	83	82	88	41	80
output	A	B	R	A									

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

key	value	key	value
:	:	81	AB
41	A	82	BR
42	B	83	RA
43	C		
44	D		
:	:		

codeword table

LZW expansion demo

value	41	42	52	41	43	41	44	81	83	82	88	41	80
output	A	B	R	A	C								

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

key	value	key	value
:	:	81	AB
41	A	82	BR
42	B	83	RA
43	C		
44	D		
:	:		

codeword table

LZW expansion demo

value	41	42	52	41	43	41	44	81	83	82	88	41	80
output	A	B	R	A	C								

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

key	value	key	value
:	:	81	AB
41	A	82	BR
42	B	83	RA
43	C	84	AC
44	D		
:	:		

codeword table

LZW expansion demo

value	41	42	52	41	43	41	44	81	83	82	88	41	80
output	A	B	R	A	C	A							

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

key	value	key	value
:	:	81	AB
41	A	82	BR
42	B	83	RA
43	C	84	AC
44	D		
:	:		

codeword table

LZW expansion demo

value	41	42	52	41	43	41	44	81	83	82	88	41	80
output	A	B	R	A	C	A							

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

key	value	key	value
:	:	81	AB
41	A	82	BR
42	B	83	RA
43	C	84	AC
44	D	85	CA
:	:		

codeword table

LZW expansion demo

value	41	42	52	41	43	41	44	81	83	82	88	41	80
output	A	B	R	A	C	A	D						

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

key	value	key	value
:	:	81	AB
41	A	82	BR
42	B	83	RA
43	C	84	AC
44	D	85	CA
:	:		

codeword table

LZW expansion demo

value	41	42	52	41	43	41	44	81	83	82	88	41	80
output	A	B	R	A	C	A	D						

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

key	value	key	value
:	:	81	AB
41	A	82	BR
42	B	83	RA
43	C	84	AC
44	D	85	CA
:	:	86	AD

codeword table

LZW expansion demo

value	41	42	52	41	43	41	44	81	83	82	88	41	80
output	A	B	R	A	C	A	D	AB					

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

key	value	key	value
:	:	81	AB
41	A	82	BR
42	B	83	RA
43	C	84	AC
44	D	85	CA
:	:	86	AD

codeword table

LZW expansion demo

value	41	42	52	41	43	41	44	81	83	82	88	41	80
output	A	B	R	A	C	A	D	AB					

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

key	value	key	value	key	value
:	:	81	AB	87	DA
41	A	82	BR		
42	B	83	RA		
43	C	84	AC		
44	D	85	CA		
:	:	86	AD		

codeword table

LZW expansion demo

value	41	42	52	41	43	41	44	81	83	82	88	41	80
output	A	B	R	A	C	A	D	AB	RA				

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

key	value	key	value	key	value
:	:	81	AB	87	DA
41	A	82	BR		
42	B	83	RA		
43	C	84	AC		
44	D	85	CA		
:	:	86	AD		

codeword table

LZW expansion demo

value	41	42	52	41	43	41	44	81	83	82	88	41	80
output	A	B	R	A	C	A	D	AB	RA				

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

key	value	key	value	key	value
:	:	81	AB	87	DA
41	A	82	BR	88	ABR
42	B	83	RA		
43	C	84	AC		
44	D	85	CA		
:	:	86	AD		

codeword table

LZW expansion demo

value	41	42	52	41	43	41	44	81	83	82	88	41	80
output	A	B	R	A	C	A	D	AB	RA	BR			

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

key	value	key	value	key	value
:	:	81	AB	87	DA
41	A	82	BR	88	ABR
42	B	83	RA		
43	C	84	AC		
44	D	85	CA		
:	:	86	AD		

codeword table

LZW expansion demo

value	41	42	52	41	43	41	44	81	83	82	88	41	80
output	A	B	R	A	C	A	D	AB	RA	BR			

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

key	value	key	value	key	value
:	:	81	AB	87	DA
41	A	82	BR	88	ABR
42	B	83	RA	89	RAB
43	C	84	AC		
44	D	85	CA		
:	:	86	AD		

codeword table

LZW expansion demo

value	41	42	52	41	43	41	44	81	83	82	88	41	80
output	A	B	R	A	C	A	D	AB	RA	BR	ABR		

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

key	value	key	value	key	value
:	:	81	AB	87	DA
41	A	82	BR	88	ABR
42	B	83	RA	89	RAB
43	C	84	AC		
44	D	85	CA		
:	:	86	AD		

codeword table

LZW expansion demo

value	41	42	52	41	43	41	44	81	83	82	88	41	80
output	A	B	R	A	C	A	D	AB	RA	BR	ABR		

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

key	value	key	value	key	value
:	:	81	AB	87	DA
41	A	82	BR	88	ABR
42	B	83	RA	89	RAB
43	C	84	AC	8A	BRA
44	D	85	CA		
:	:	86	AD		

codeword table

LZW expansion demo

value	41	42	52	41	43	41	44	81	83	82	88	41	80
output	A	B	R	A	C	A	D	AB	RA	BR	ABR	A	

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

key	value	key	value	key	value
:	:	81	AB	87	DA
41	A	82	BR	88	ABR
42	B	83	RA	89	RAB
43	C	84	AC	8A	BRA
44	D	85	CA		
:	:	86	AD		

codeword table

LZW expansion demo

value	41	42	52	41	43	41	44	81	83	82	88	41	80
output	A	B	R	A	C	A	D	AB	RA	BR	ABR	A	

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

key	value	key	value	key	value
:	:	81	AB	87	DA
41	A	82	BR	88	ABR
42	B	83	RA	89	RAB
43	C	84	AC	8A	BRA
44	D	85	CA	8B	ABRA
:	:	86	AD		

codeword table

LZW expansion

LZW expansion.

- Create ST associating string values with W -bit keys.
- Initialize ST to contain single-char values.
- Read a W -bit key.
- Find associated string value in ST and write it out.
- Update ST.

LZW expansion

LZW expansion.

- Create ST associating string values with W -bit keys.
- Initialize ST to contain single-char values.
- Read a W -bit key.
- Find associated string value in ST and write it out.
- Update ST.

Q. How to represent LZW expansion code table?

LZW expansion

LZW expansion.

- Create ST associating string values with W -bit keys.
- Initialize ST to contain single-char values.
- Read a W -bit key.
- Find associated string value in ST and write it out.
- Update ST.

Q. How to represent LZW expansion code table?

A. An array of size 2^W .

key	value
:	:
65	A
66	B
67	C
68	D
:	:
129	AB
130	BR
131	RA
132	AC
133	CA
134	AD
135	DA
136	ABR
137	RAB
138	BRA
139	ABRA
:	:

LZW tricky case: compression

input	A	B	A	B	A	B	A
matches							
value							

LZW compression for ABABABA

key	value
:	:
A	41
B	42
C	43
D	44
:	:

codeword table

LZW tricky case: compression

input	A	B	A	B	A	B	A
matches	A						
value	41						

LZW compression for ABABABA

key	value
:	:
A	41
B	42
C	43
D	44
:	:

codeword table

LZW tricky case: compression

input	A	B	A	B	A	B	A
matches	A						
value	41						

LZW compression for ABABABA

key	value
:	:
A	41
B	42
C	43
D	44
:	:

key	value
AB	81

codeword table

LZW tricky case: compression

input	A	B	A	B	A	B	A
matches	A	B					
value	41	42					

LZW compression for ABABABA

key	value	key	value
:	:		
A	41	AB	81
B	42		
C	43		
D	44		
:	:		

codeword table

LZW tricky case: compression

input	A	B	A	B	A	B	A
matches	A	B					
value	41	42					

LZW compression for ABABABA

key	value	key	value
:	:	AB	81
A	41	BA	82
B	42		
C	43		
D	44		
:	:		

codeword table

LZW tricky case: compression

input	A	B	A	B	A	B	A
matches	A	B	AB				
value	41	42	81				

LZW compression for ABABABA

key	value
:	:
A	41
B	42
C	43
D	44
:	:

key	value
AB	81
BA	82

codeword table

LZW tricky case: compression

input	A	B	A	B	A	B	A
matches	A	B	AB				
value	41	42	81				

LZW compression for ABABABA

key	value
:	:
A	41
B	42
C	43
D	44
:	:
key	value
AB	81
BA	82
ABA	83

codeword table

LZW tricky case: compression

input	A	B	A	B	A	B	A
matches	A	B	AB		ABA		
value	41	42	81		83		

LZW compression for ABABABA

key	value
:	:
A	41
B	42
C	43
D	44
:	:
key	value
AB	81
BA	82
ABA	83

codeword table

LZW tricky case: compression

input	A	B	A	B	A	B	A
matches	A	B	AB		ABA		
value	41	42	81		83		80

LZW compression for ABABABA

key	value
:	:
A	41
B	42
C	43
D	44
:	:
AB	81
BA	82
ABA	83

codeword table

LZW tricky case: expansion

value	41	42	81	83	80
output					

LZW expansion for 41 42 81 83 80

key	value
:	:
41	A
42	B
43	C
44	D
:	:

codeword table

LZW tricky case: expansion

value	41	42	81	83	80
output	A				

LZW expansion for 41 42 81 83 80

key	value
:	:
41	A
42	B
43	C
44	D
:	:

codeword table

LZW tricky case: expansion

value	41	42	81	83	80
output	A	B			

LZW expansion for 41 42 81 83 80

key	value
:	:
41	A
42	B
43	C
44	D
:	:

codeword table

LZW tricky case: expansion

value	41	42	81	83	80
output	A	B			

LZW expansion for 41 42 81 83 80

key	value
:	:
41	A
42	B
43	C
44	D
:	:

key	value
81	AB

codeword table

LZW tricky case: expansion

value	41	42	81	83	80
output	A	B	AB		

LZW expansion for 41 42 81 83 80

key	value
:	:
41	A
42	B
43	C
44	D
:	:

key	value
81	AB

codeword table

LZW tricky case: expansion

value	41	42	81	83	80
output	A	B	AB		

LZW expansion for 41 42 81 83 80

key	value	key	value
:	:	81	AB
41	A	82	BA
42	B		
43	C		
44	D		
:	:		

codeword table

LZW tricky case: expansion

value	41	42	81	83	80
output	A	B	AB		

need to know which
key has value 83
before it is in ST!

LZW expansion for 41 42 81 83 80

key	value
:	:
41	A
42	B
43	C
44	D
:	:

key	value
81	AB
82	BA

codeword table

LZW tricky case: expansion

value	41	42	81	83	80
output	A	B	AB	ABA	←

need to know which
key has value 83
before it is in ST!

LZW expansion for 41 42 81 83 80

key	value
:	:
41	A
42	B
43	C
44	D
:	:

key	value
81	AB
82	BA
83	ABA

codeword table

LZW implementation details

How big to make ST?

- How long is message?
- Whole message similar model?
- [many other variations]

LZW implementation details

How big to make ST?

- How long is message?
- Whole message similar model?
- [many other variations]

What to do when ST fills up?

- Throw away and start over. [GIF]
- Throw away when not effective. [Unix compress]
- [many other variations]

LZW implementation details

How big to make ST?

- How long is message?
- Whole message similar model?
- [many other variations]

What to do when ST fills up?

- Throw away and start over. [GIF]
- Throw away when not effective. [Unix compress]
- [many other variations]

Why not put longer substrings in ST?

- [many variations have been developed]

LZW in the real world

Lempel-Ziv and friends.

- LZ77.
LZ77 not patented ⇒ widely used in open source
- LZ78.
LZW patent #4,558,302 expired in U.S. on June 20, 2003
- LZW.
- Deflate / zlib = LZ77 variant + Huffman.

United States Patent [19]
Welch

[11] Patent Number: **4,558,302**
[45] Date of Patent: **Dec. 10, 1985**

[54] HIGH SPEED DATA COMPRESSION AND DECOMPRESSION APPARATUS AND METHOD
[75] Inventor: Terry A. Welch, Concord, Mass.
[73] Assignee: Sperry Corporation, New York, N.Y.
[21] Appl. No.: 505,638
[22] Filed: Jun. 20, 1983
[51] Int. Cl.⁴ G06F 5/00
[52] U.S. Cl. 340/347 DD; 235/310
[58] Field of Search 340/347 DD; 235/310, 235/311; 364/200, 900

[56] References Cited
U.S. PATENT DOCUMENTS
4,464,650 8/1984 Eastman 340/347 DD

OTHER PUBLICATIONS
Ziv, "IEEE Transactions on Information Theory", IT-24-5, Sep. 1977, pp. 530-537.
Ziv, "IEEE Transactions on Information Theory", IT-23-3, May 1977, pp. 337-343.

Primary Examiner—Charles D. Miller
Attorney, Agent, or Firm—Howard P. Terry; Albert B. Cooper

[57] ABSTRACT
A data compressor compresses an input stream of data character signals by storing in a string table strings of data character signals encountered in the input stream. The compressor searches the input stream to determine the longest match to a stored string. Each stored string comprises a prefix string and an extension character where the extension character is the last character in the string and the prefix string comprises all but the extension character. Each string has a code signal associated therewith and a string is stored in the string table by, at least implicitly, storing the code signal for the string, the code signal for the string prefix and the extension character. When the longest match between the input data character stream and the stored strings is determined, the code signal for the longest match is transmitted as the compressed code signal for the encountered string of characters and an extension string is stored in the string table. The prefix of the extended string is the longest match and the extension character of the extended string is the next input data character signal following the longest match. Searching through the string table and entering extended strings therein is effected by a limited search hashing procedure. Decompression is effected by a decompressor that receives the compressed code signals and generates a string table similar to that constructed by the compressor to effect lookup of received code signals so as to recover the data character signals comprising a stored string. The decompressor string table is updated by storing a string having a prefix in accordance with a prior received code signal and an extension character in accordance with the first character of the currently recovered string.

181 Claims, 9 Drawing Figures



LZW in the real world

Lempel-Ziv and friends.

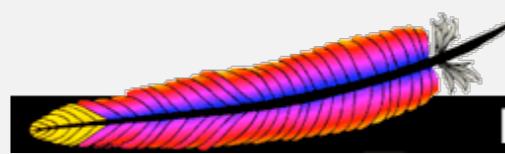
- LZ77.
- LZ78.
- LZW.
- Deflate / zlib = LZ77 variant + Huffman.



Unix compress, GIF, TIFF, V.42bis modem: LZW.

zip, 7zip, gzip, jar, png, pdf: deflate / zlib.

iPhone, Sony Playstation 3, Apache HTTP server: deflate / zlib.



Apache
HTTP SERVER PROJECT

Data compression summary

Lossless compression.

- Represent fixed-length symbols with variable-length codes. [Huffman]
- Represent variable-length symbols with fixed-length codes. [LZW]

Data compression summary

Lossless compression.

- Represent fixed-length symbols with variable-length codes. [Huffman]
- Represent variable-length symbols with fixed-length codes. [LZW]

Lossy compression. [not covered in this course]

- JPEG, MPEG, MP3, ...
- FFT, wavelets, fractals, ...

Data compression summary

Lossless compression.

- Represent fixed-length symbols with variable-length codes. [Huffman]
- Represent variable-length symbols with fixed-length codes. [LZW]

Lossy compression. [not covered in this course]

- JPEG, MPEG, MP3, ...
- FFT, wavelets, fractals, ...

Theoretical limits on compression. Shannon entropy:

$$H(X) = - \sum_i^n p(x_i) \lg p(x_i)$$

Data compression summary

Lossless compression.

- Represent fixed-length symbols with variable-length codes. [Huffman]
- Represent variable-length symbols with fixed-length codes. [LZW]

Lossy compression. [not covered in this course]

- JPEG, MPEG, MP3, ...
- FFT, wavelets, fractals, ...

Theoretical limits on compression. Shannon entropy:

$$H(X) = - \sum_i^n p(x_i) \lg p(x_i)$$

Practical compression. Use extra knowledge whenever possible.