CS 106B

Lecture 26: Esoteric Data Structures: Skip Lists and Bloom Filters

Friday, June 1, 2018
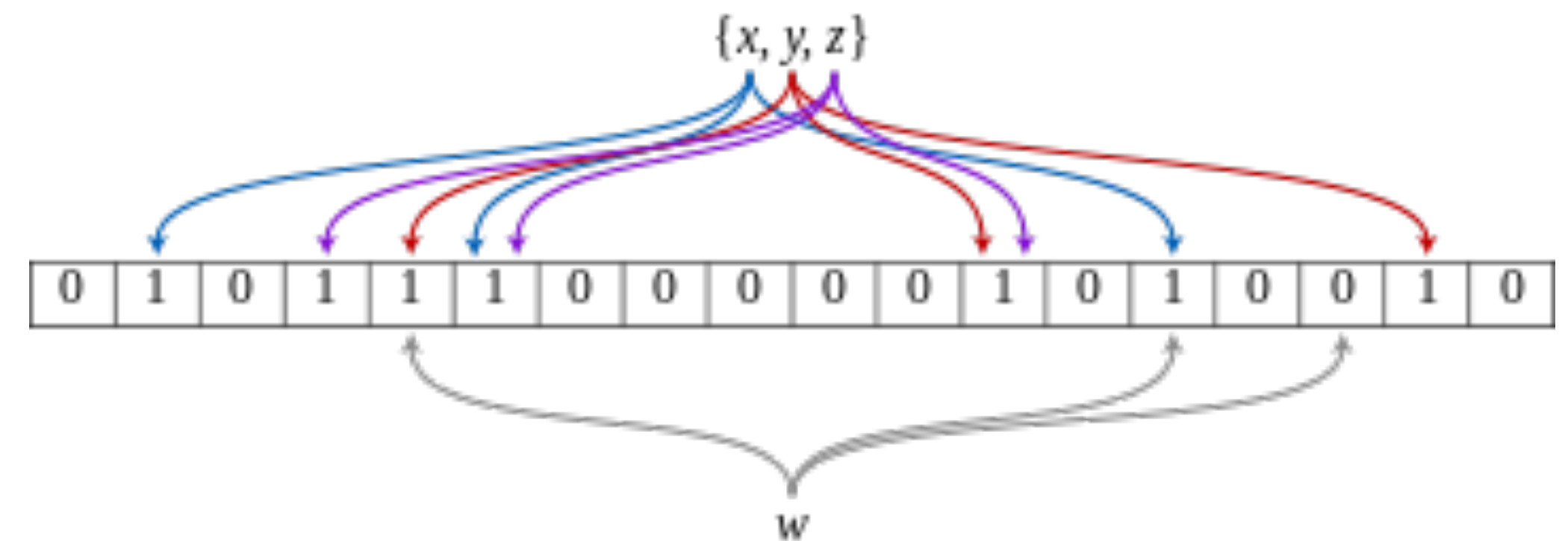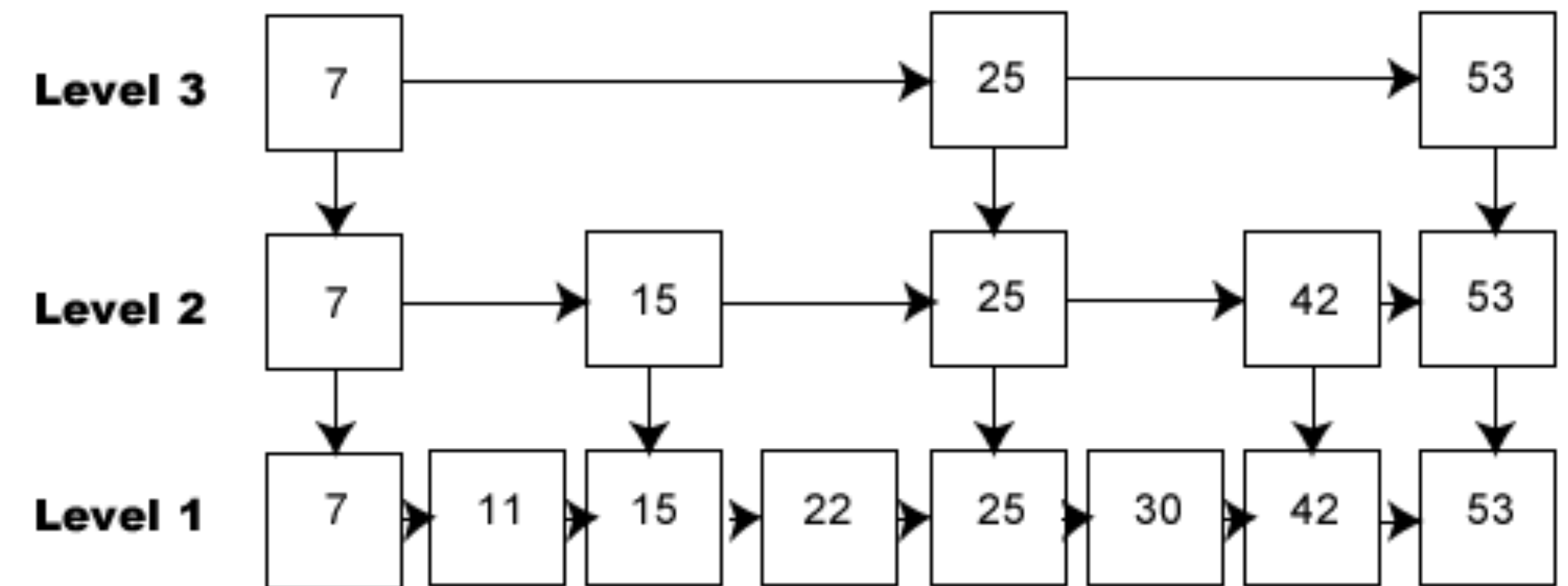
Today's Topics

- Logistics
  - Final Exam Review materials posted by 5pm today: http://web.stanford.edu/class/cs106b/handouts/final.html
  - We will have a review session Tuesday, location TBA.

- Esoteric Data Structures
  - Skip Lists
  - Bloom Filters

Esoteric Data Structures

- In CS 106B, we have talked about many standard, famous, and commonly used data structures: Vectors, Linked Lists, Trees, Hash Tables, Graphs

- However, we only scratched the surface of available data structures, and data structure research is alive and well to this day.

- Let's take a look at two interesting data structures that have interesting properties and you might not see covered in detail in a standard course: the **skip list** and the **bloom filter**.

difference-list
table
b-tree
2-3-heap
rolling-hash
heightmap
skip-list
kd-tree
splay-tree
lookup

Skip Lists

- A "skip list" is a balanced search structure that maintains an ordered, dynamic set for insertion, deletion and search

- What other efficient (log n or better) sorted search structures have we talked about?

  Hash Tables (nope, not sorted)

  Heaps (nope, not searchable)

  Sorted Array (kind of, but, insert/delete is O(n))

 Binary Trees (only if balanced, e.g., AVL or Red/Black)

Skip Lists

- A skip list is a simple, randomized search structure that will give us O(log N) in expectation for search, insert, and delete, but also with high probability.

- Invented by William Pugh in 1989 -- fairly recent!

- Let's see what we can do with a linked list to make it better.
- How long does it take to search a sorted, doubly-linked list for an element?

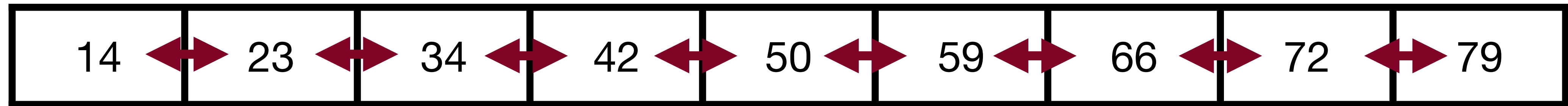head

| 14 | 23 | 34 | 42 | 50 | 59 | 66 | 72 | 79 |
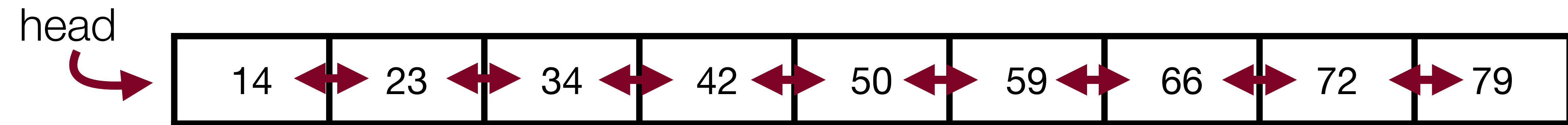
~~log(N)~~ nope!
it is O(n) … we *must* traverse the list!

• How might we help this situation?

| 14 | ⬌ | 23 | ⬌ | 34 | ⬌ | 42 | ⬌ | 50 | ⬌ | 59 | ⬌ | 66 | ⬌ | 72 | ⬌ | 79 |

• What if we put another link into the middle?

middle

head

| 14 | 23 | 34 | 42 | 50 | 59 | 66 | 72 | 79 |
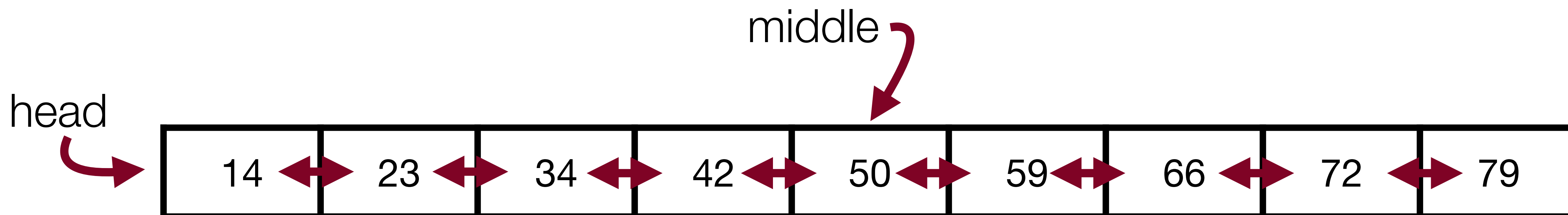
• This would help a little…we could start searching from the middle, but we would still have to traverse

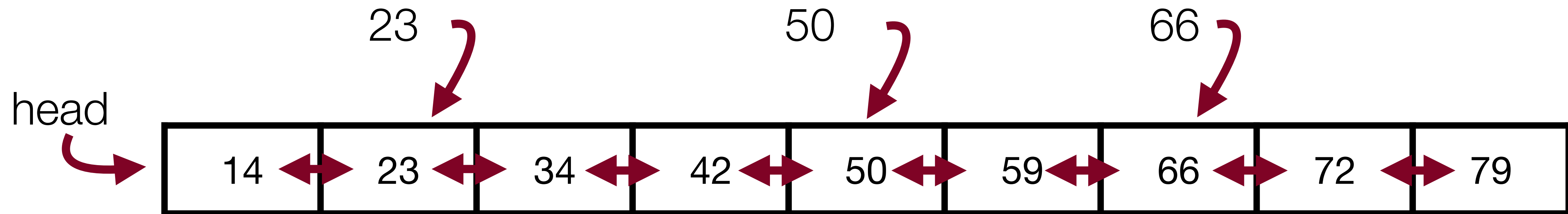• O(n) becomes … O((½)n) becomes … O(n)

Improving the Linked List

- Maybe we could add more pointers…
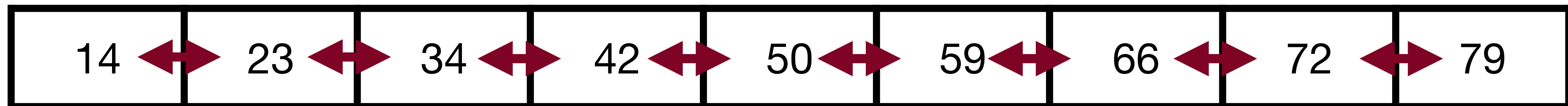

- This would help some more…but still doesn't solve the underlying problem.

- Let's play a game. I've chosen the numbers for this list in a particular way. Does anyone recognize the sequence?

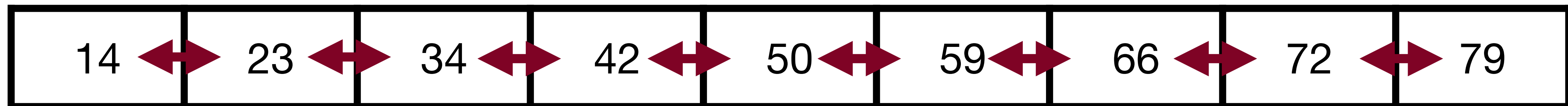| 14 | ↔ | 23 | ↔ | 34 | ↔ | 42 | ↔ | 50 | ↔ | 59 | ↔ | 66 | ↔ | 72 | ↔ | 79 |

• Let's play a game. I've chosen the numbers for this list in a particular way. Does anyone recognize the sequence?

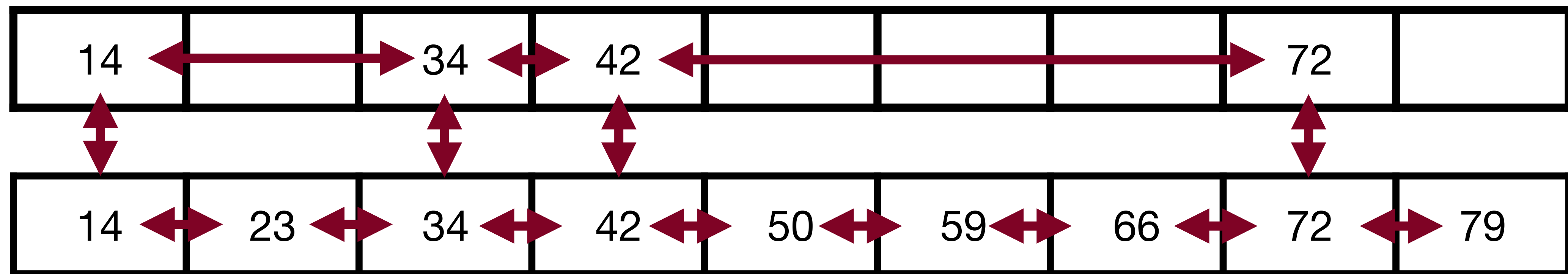| 14 | ↔ | 23 | ↔ | 34 | ↔ | 42 | ↔ | 50 | ↔ | 59 | ↔ | 66 | ↔ | 72 | ↔ | 79 |

• These are subway stops on the NYC 7th Avenue line :)

- A somewhat unique feature in the New York City subway system is that it has *express lines*:

- This models a skip list almost perfectly!

- To search the list (or ride the subway): Walk right in the top list (L1) and when you've gone too far, go back and then down to the bottom list (L2) (e.g., search for 59)
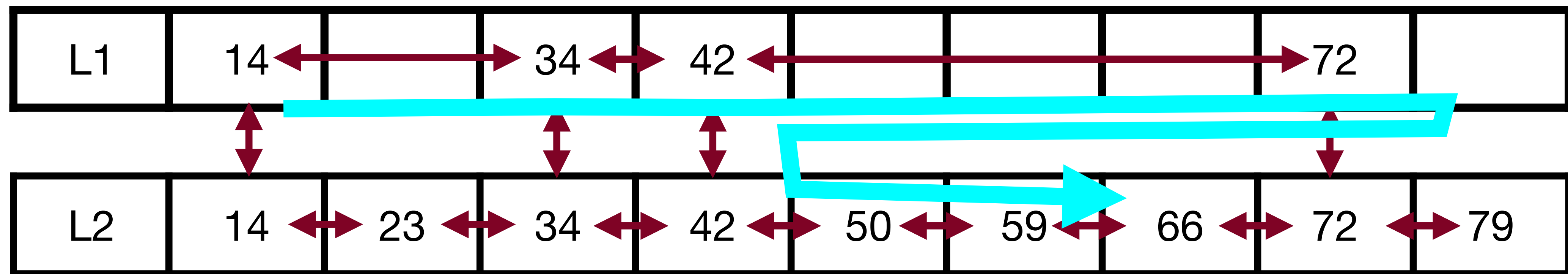
- What is the best placement for the nodes in L1?
- This placement might be good for subways, but we care about worst-case performance, which we want to minimize. How about equally spaced nodes?

| L1 | 14 | | 34 | 42 | | | | 72 | |

| L2 | 14 | 23 | 34 | 42 | 50 | 59 | 66 | 72 | 79 |
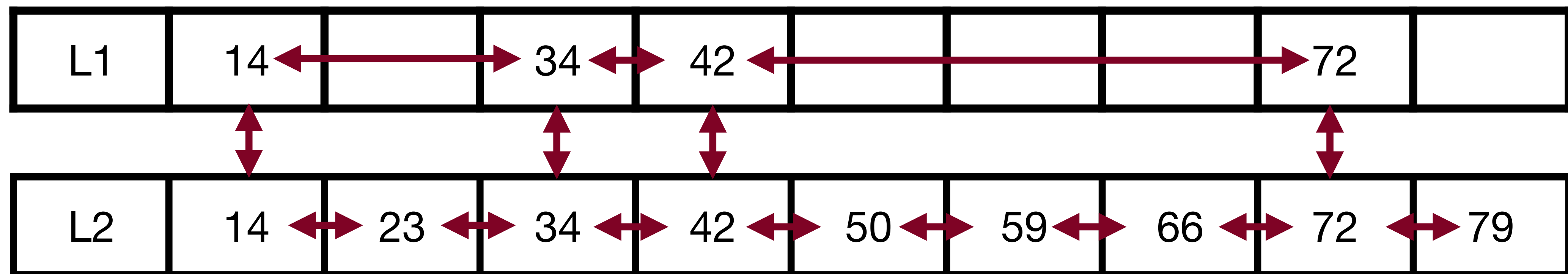
- The "search cost" can be represented by |L1| + (|L2| / |L1|), or |L1| + (n / |L1|), where n is the number of nodes in L2 (L2 must have all stops)
- Let's do some calculus to minimize this amount…
- The minimum will be when |L1| is equal to (n/|L1|), or when |L1|=$\sqrt{n}$
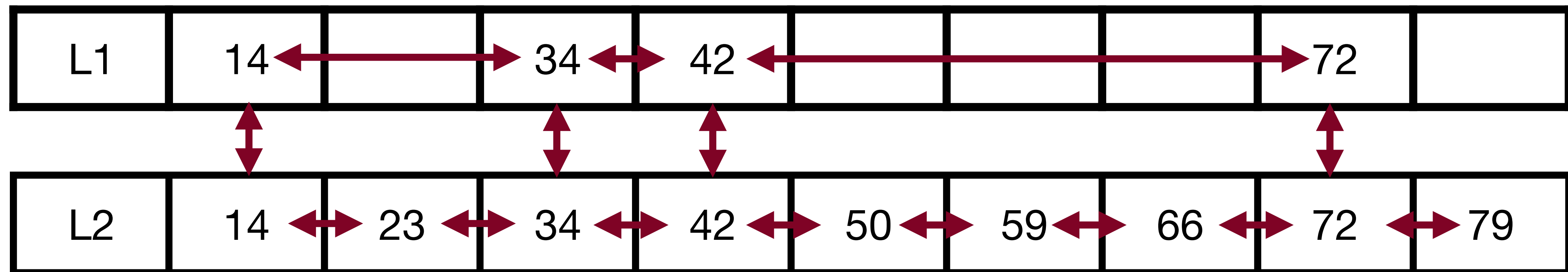
| L1 | 14 | | 34 | 42 | | | | 72 | |

| L2 | 14 | 23 | 34 | 42 | 50 | 59 | 66 | 72 | 79 |

- The minimum will be when |L1| is equal to (n/|L1|), or when $|L1|=\sqrt{n}$
- So, the search cost with a minimum second list is $\sqrt{n} + n/\sqrt{n} = 2\sqrt{n}$
- We want them equally spaced.
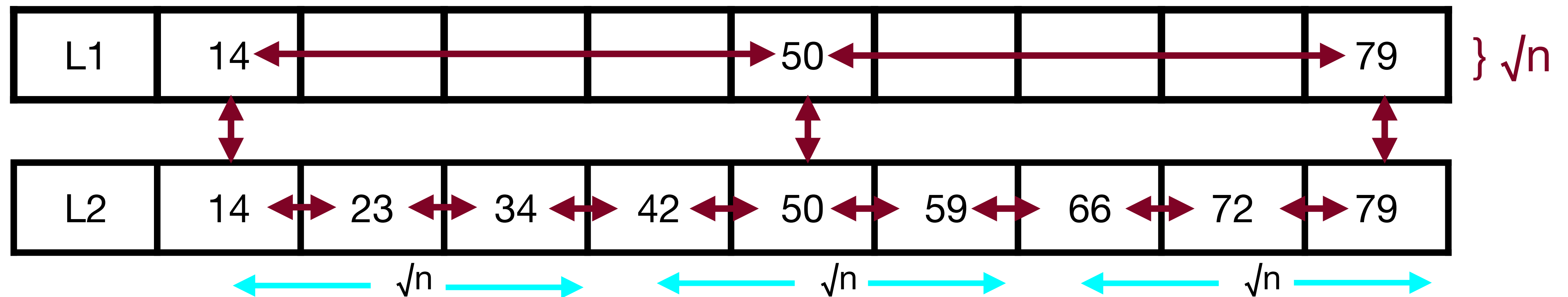
- The minimum will be when |L1| is equal to (n/|L1|), or when |L1|=$\sqrt{n}$
- So, the search cost with a minimum second list is $\sqrt{n} + n/\sqrt{n} = 2\sqrt{n}$
- We want them equally spaced. Big O?

$O(2\sqrt{n}) = O(\sqrt{n})$

Good? Let's compare to O(log n)

What if we had more linked lists??

- 2 sorted lists: $2\sqrt{n}$

| L1 | 14 | | | 42 | | | 66 | | 79 | $\}\ \sqrt{n}$ |

| L2 | 14 | 23 | 34 | 42 | 50 | 59 | 66 | 72 | 79 |

$\sqrt{n}$ $\sqrt{n}$ $\sqrt{n}$
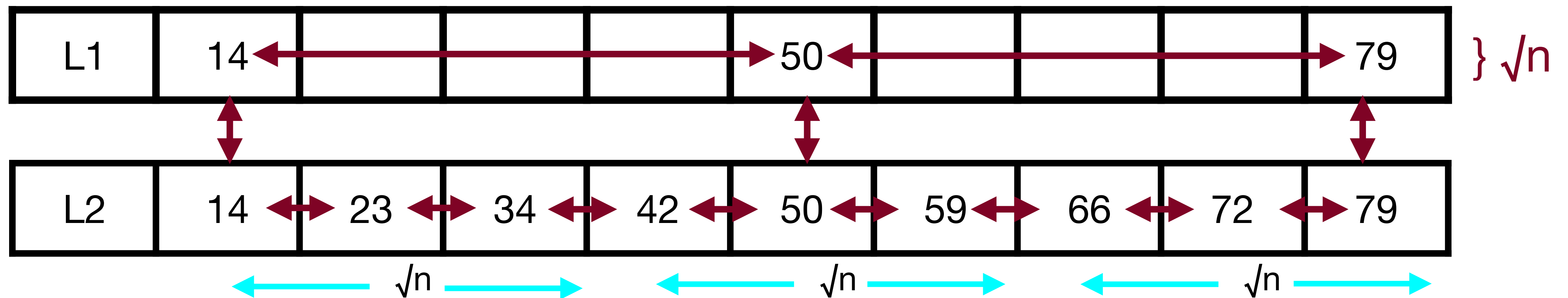
Improving the Linked List

What if we had more linked lists??
- 2 sorted lists: $2\sqrt{n}$
- 3 sorted lists: $3\sqrt[3]{n}$

| L1 | 14 | | | | 50 | | | | 79 | } 3 |

| L2 | 14 ⟷ | 23 ⟷ | 34 ⟷ | 42 ⟷ | 50 ⟷ | 59 ⟷ | 66 ⟷ | 72 ⟷ | 79 | } 9 |

$\sqrt[3]{n}$ $\sqrt[3]{n}$ $\sqrt[3]{n}$

| L3 | ⟷ | ⟷ | ⟷ | ⟷ | ⟷ | ⟷ | ⟷ | ⟷ | ⟷ | } 27 |

What if we had more linked lists??
- 2 sorted lists: $2\sqrt{n}$
- 3 sorted lists: $3\sqrt[3]{n}$
- k sorted lists: $k\sqrt[k]{n}$

L1	14				50				79

L1	14			42			66	$\}\ \sqrt{n}$	

L2	14	23	34	42	50	59	66	72	79

$\sqrt{n}$ $\sqrt{n}$ $\sqrt{n}$

What if we had more linked lists??
- 2 sorted lists: $2\sqrt{n}$
- 3 sorted lists: $3\sqrt[3]{n}$
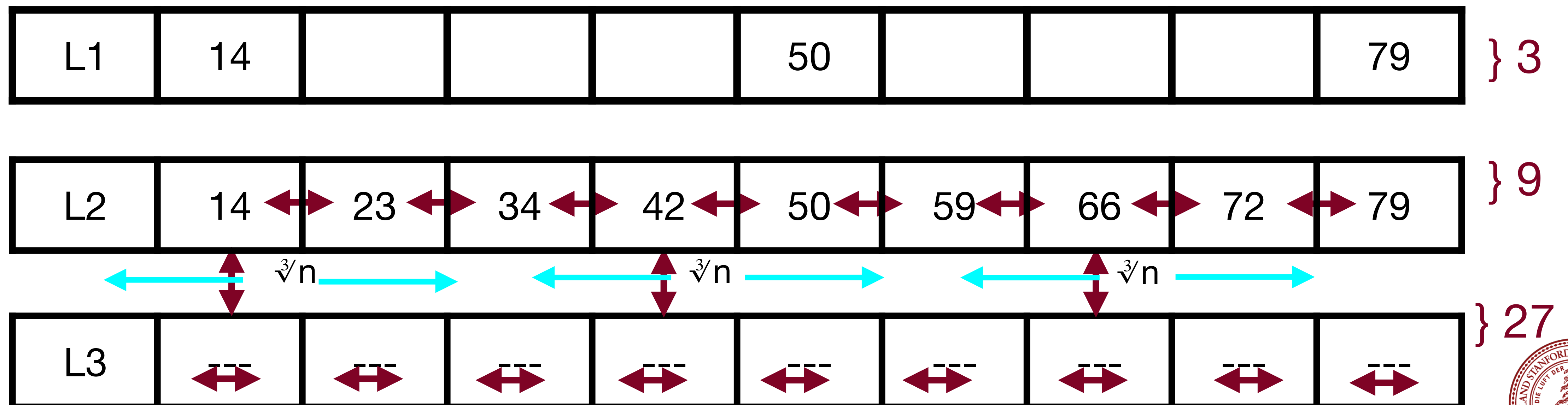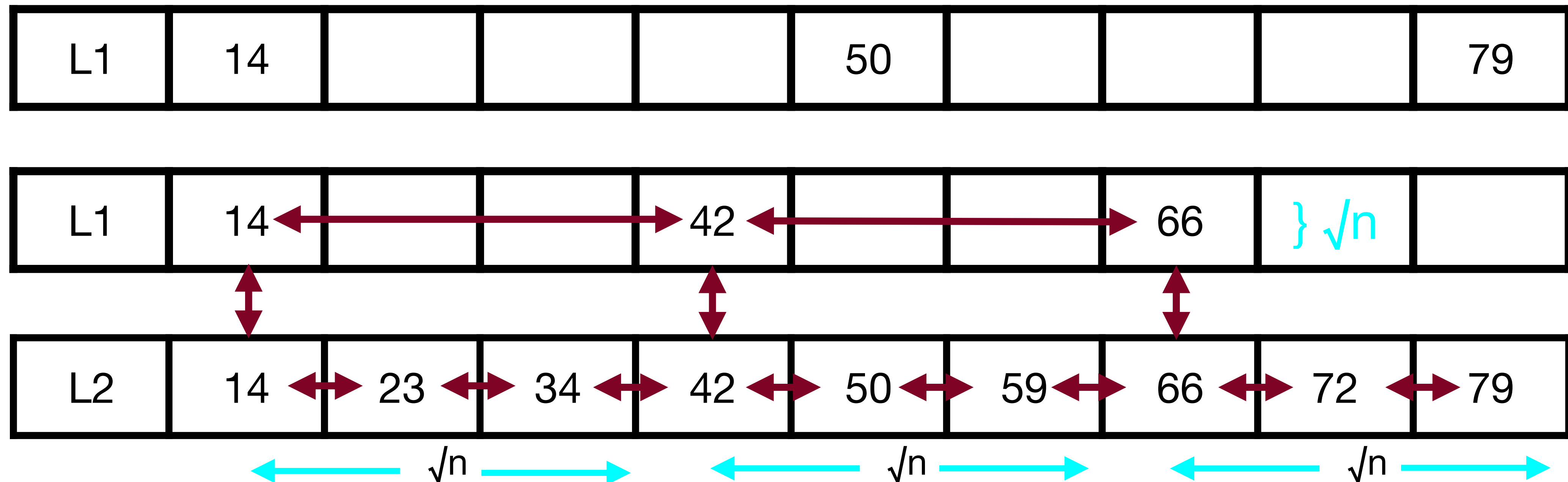- k sorted lists: $k\sqrt[k]{n}$

- log n sorted lists:

What if we had more linked lists??
- 2 sorted lists: $2\sqrt{n}$
- 3 sorted lists: $3\sqrt[3]{n}$
- k sorted lists: $k\sqrt[k]{n}$
- log n sorted lists: $\log n \sqrt[\log n]{n}$

What is $\sqrt[\log n]{n}$ equal to?

What if we had more linked lists??
- 2 sorted lists: $2\sqrt{n}$
- 3 sorted lists: $3\sqrt[3]{n}$
- k sorted lists: $k\sqrt[k]{n}$
- log n sorted lists: $\log n \sqrt[\log n]{n}$

What is $\sqrt[\log n]{n}$ equal to?

$$\sqrt[\log n]{n} = 2$$

L1	14				50				79

L1	14			42			66	$\}\sqrt{n}$	

L2	14	23	34	42	50	59	66	72	79

$\sqrt{n}$ $\sqrt{n}$ $\sqrt{n}$

What if we had more linked lists??
- 2 sorted lists: $2\sqrt{n}$
- 3 sorted lists: $3\sqrt[3]{n}$
- k sorted lists: $k\sqrt[k]{n}$
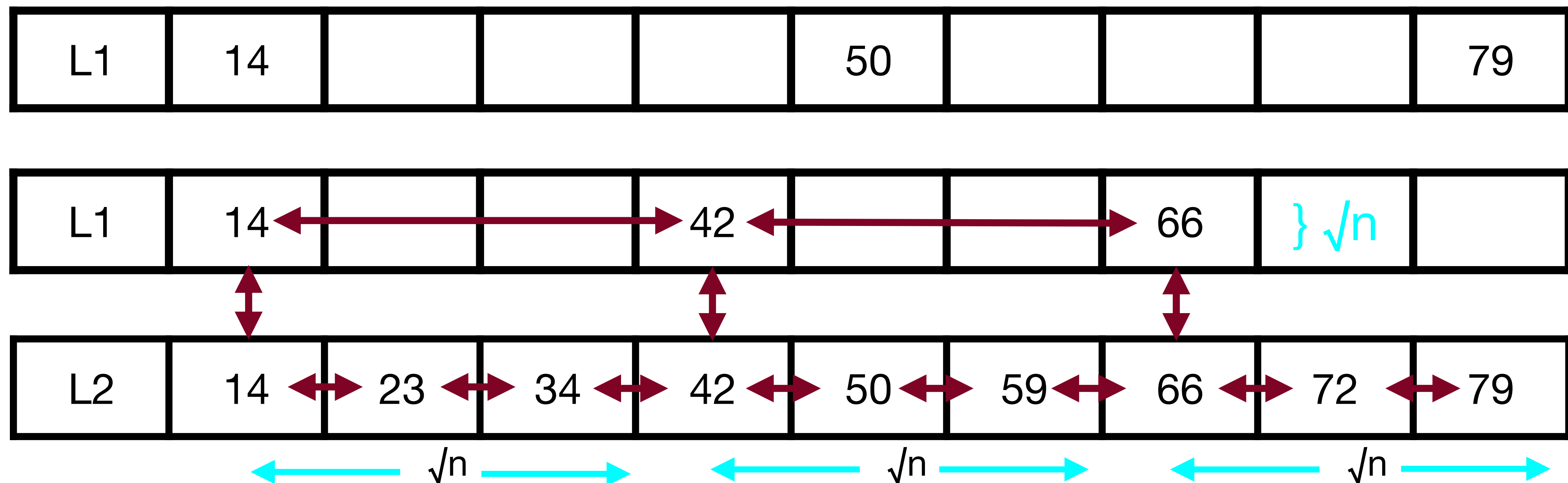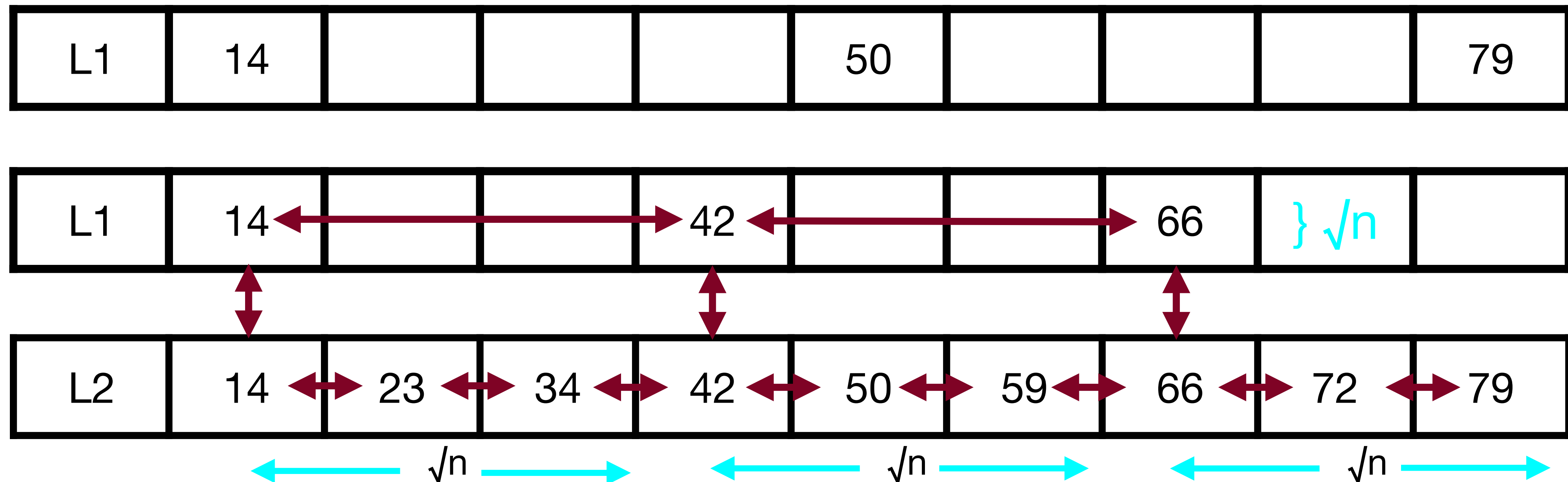- log n sorted lists: $\log n \sqrt[\log n]{n} = 2 \log n$: logarithmic behavior!

L1	14				50				79

L1	14			42			66	$\}\sqrt{n}$	

L2	14	23	34	42	50	59	66	72	79

$\sqrt{n}$ $\sqrt{n}$ $\sqrt{n}$
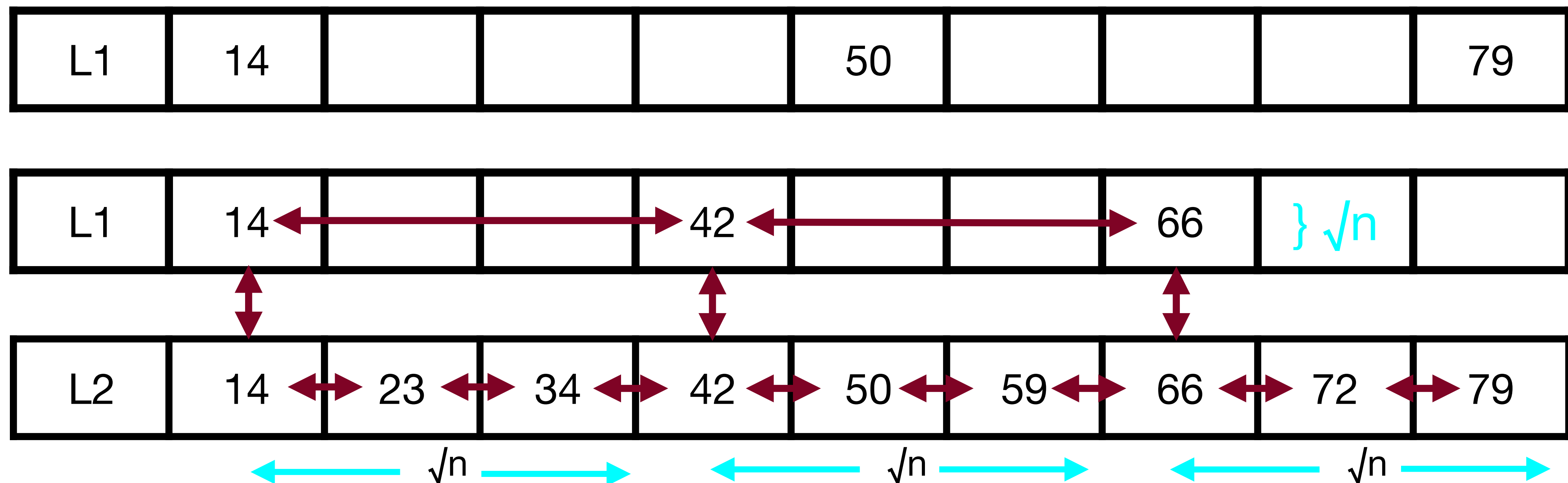
Skip Lists

log n linked lists look like a binary tree (and act like one!)

14								79

14				50				79

14		34		50		66		79

14	23	34	42	50	59	66	72	79
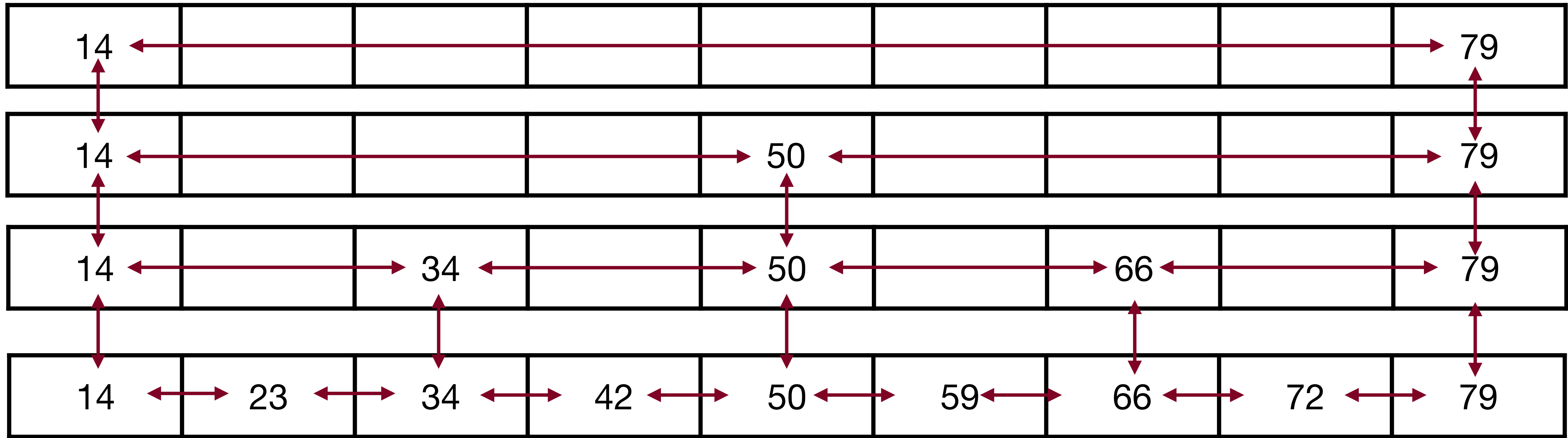
We just determined that the best option if we have n elements is to have log_2n lists.

To build a skip list, we could try to keep all the elements perfectly aligned — in the lowest list, we have n elements, and in the next list up we have n/2 elements, etc.

To build a skip list, we could try to keep all the elements perfectly aligned — in the lowest list, we have n elements, and in the next list up we have n/2 elements, etc.

This is not efficient…we would have to be moving links all over the place!

So…what we do instead is implement a *probabilistic strategy* —

So…what we do instead is implement a *probabilistic strategy* — we flip a coin!

1. All elements must go into the bottom list (search to find the spot)
2. After inserting into the bottom list, flip a fair, two sided coin. If the coin comes up heads, add the element to the next list up, and flip again, repeating step 2.
3. If the coin comes up tails, stop.
   (example on board - you do have to have -∞ on each level)

Let's build one!

To search a skip list, we "traverse" the list level-by-level, and there is a high probability that there are log n levels. Each level up has a good probability to have approximately half the number of elements. There is a high probability that searching is O(log n).

To insert, we first search O(log n), and then we must flip the coin to keep adding. Worst case? O(∞). But, there is a *very good probability* that we will have to do a small number of inserts up the list. So, this has a high probability of also being simply O(log n)

To delete? Find the first instance of your value, then delete from all the lists — also O(log n).

Our second esoteric data structure is called a *bloom filter*, named for its creator, Burton Howard Bloom, who invented the data structure in 1970.

A bloom filter is a space efficient, probabilistic data structure that is used to tell whether a member is in a set.

Bloom filters are a bit odd because they can *definitely* tell you whether an element is *not* in the set, but can only say whether the element is *possibly* in the set.

Bloom Filters

In other words: "false positives" are possible, but "false negatives" are not.
(A *false positive* would say that the element is in the set when it isn't, and a *false negative* would say that the element is not in the set when it is.

Bloom Filters

The idea is that we have a "bit array." We will model a bit array with a regular array, but you can compress a bit array by up to 32x because there are 8 bits in a byte, and there are 4 bytes to a 32-bit number (thus, 32x!) (although Bloom Filters themselves need more space per element than 1 bit).

a bit array:

1	0	1	1	0	1	1	1

Bloom Filters: start with an empty bit array (all zeros), and *k* hash functions.

k1 = (13 - (x % 13))% 7, k2 = (3 + 5x) % 7, etc.

0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0

Bloom Filters: start with an empty bit array (all zeros), and *k* hash functions.

The hash functions should be independent, and the optimal amount is calculable based on the number of items you are hashing, and the length of your table (see Wikipedia for details).

0	1	2	3	4	5	6	7
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Values then get hashed by all k hashes, and the bit in the hashed position is set to 1 in each case.

0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0

Insert 129: x=129, k1=1, k2=4

k1 = (13 - (x % 13))% 7, k2 = (3 + 5x) % 7, etc.

0	1	2	3	4	5	6	7
0	**1**	**0**	**0**	**1**	**0**	**0**	**0**

k1 == 1, so we change bit 1 to a 1
k2 == 4, so we change bit 4 to a 1

Bloom Filters

Insert 479: x=479, k1=2, k2=4

k1 = (13 - (x % 13))% 7, k2 = (3 + 5x) % 7, etc.

0	1	2	3	4	5	6	7
0	**1**	**1**	**0**	**1**	**0**	**0**	**0**

k1 == 2, so we change bit 2 to a 1
k2 == 4, so we would change bit
4 to a 1, but it is already a 1.

To check if 129 is in the table, just hash again and check the bits.

k1=1, k2=4: probably in the table!

0	1	2	3	4	5	6	7
0	1	1	0	1	0	0	0

k1 = (13 - (x % 13))% 7, k2 = (3 + 5x) % 7, etc.

To check if 123 is in the table, hash and check the bits. k1=0, k2=2: *cannot* be in table because the 0 bit is still 0.

0	1	2	3	4	5	6	7
0	1	1	0	1	0	0	0

k1 = (13 - (x % 13))% 7, k2 = (3 + 5x) % 7, etc.

To check if 402 is in the table, hash and check the bits. k1=1, k2=4:

Probably in the table (but isn't! False positive!).

0	1	2	3	4	5	6	7
0	**1**	**1**	**0**	**1**	**0**	**0**	**0**

Online example: http://billmill.org/bloomfilter-tutorial/

k1 = (13 - (x % 13))% 7, k2 = (3 + 5x) % 7, etc.

What is the probability that we have a false positive?

If m is the number of bits in the array, then the probability that a bit is not set to 1 is

$$1 - \frac{1}{m}$$

If k is the number of hash functions, the probability that the bit is not set to 1 by any hash function is

$$\left(1 - \frac{1}{m}\right)^k$$

If we have inserted n elements, the probability that a certain bit is still 0 is

$$\left(1 - \frac{1}{m}\right)^{kn}$$

To get the probability that a bit is 1 is just 1- the answer on the previous slide:

$$1 - \left(1 - \frac{1}{m}\right)^{kn}$$

Now test membership of an element that is not in the set. Each of the k array positions computed by the hash functions is 1 with a probability as above. The probability of all of them being 1, (false positive):

$$\left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^{k} \approx \left(1 - e^{-kn/m}\right)^{k}$$

For our previous example, m=8, n=2, k=2, so:

$$\left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^{k} = 0.17, \text{ or } 17\% \text{ of the time we will get}$$
a false positive.

Why would we want a structure that can produce false positives?

Example: Google Chrome uses a local Bloom Filter to check for malicious URLs — if there is a hit, a stronger check is performed.

There is one more negative issue with a Bloom Filter: you can't delete! If you delete, you might delete another inserted value, as well! You could keep a second bloom filter of removals, but then you could get false positives in that filter…

You have to perform *k* hashing functions for an element, and then either flip bits, or read bits. Therefore, they perform in O(k) time, which is independent of the number of elements in the structure. Additionally, because the hashes are independent, they can be parallelized, which gives drastically better performance with multiple processors.

- **References:**
  - MIT Skip Lists lecture: http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-12-skip-lists/
  - Online Bloom Filter example: http://billmill.org/bloomfilter-tutorial/
  - Wikipedia Bloom Filters: https://en.wikipedia.org/wiki/Bloom_filter

Esoteric Data Structure: Ropes

Normally, strings are kept in memory in contiguous chunks:

"The_quick_fox_jumps_over_the_dog"

| 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
T	h	e	_	q	u	i	c	k	_	f	o	x	_	j	u	m	p	s	_	o	v	e	r	_	t	h	e	_	d	o	g

Ropes

However, this doesn't make it easy to insert into a string: you have to break the whole string up each time, and re-create a new string.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
T	h	e	_	q	u	i	c	k	_	f	o	x	_	j	u	m	p	s	_	o	v	e	r	_	t	h	e	_	d	o	g

Ropes

A "rope" is a tree of smaller strings (eventually—it can start as a long string) that makes it efficient to store and manipulate the entire string.

16

10

24

10

6

19

3

The_quick_

brown_

19

5

dog

fox_jumps_over_the_

lazy_
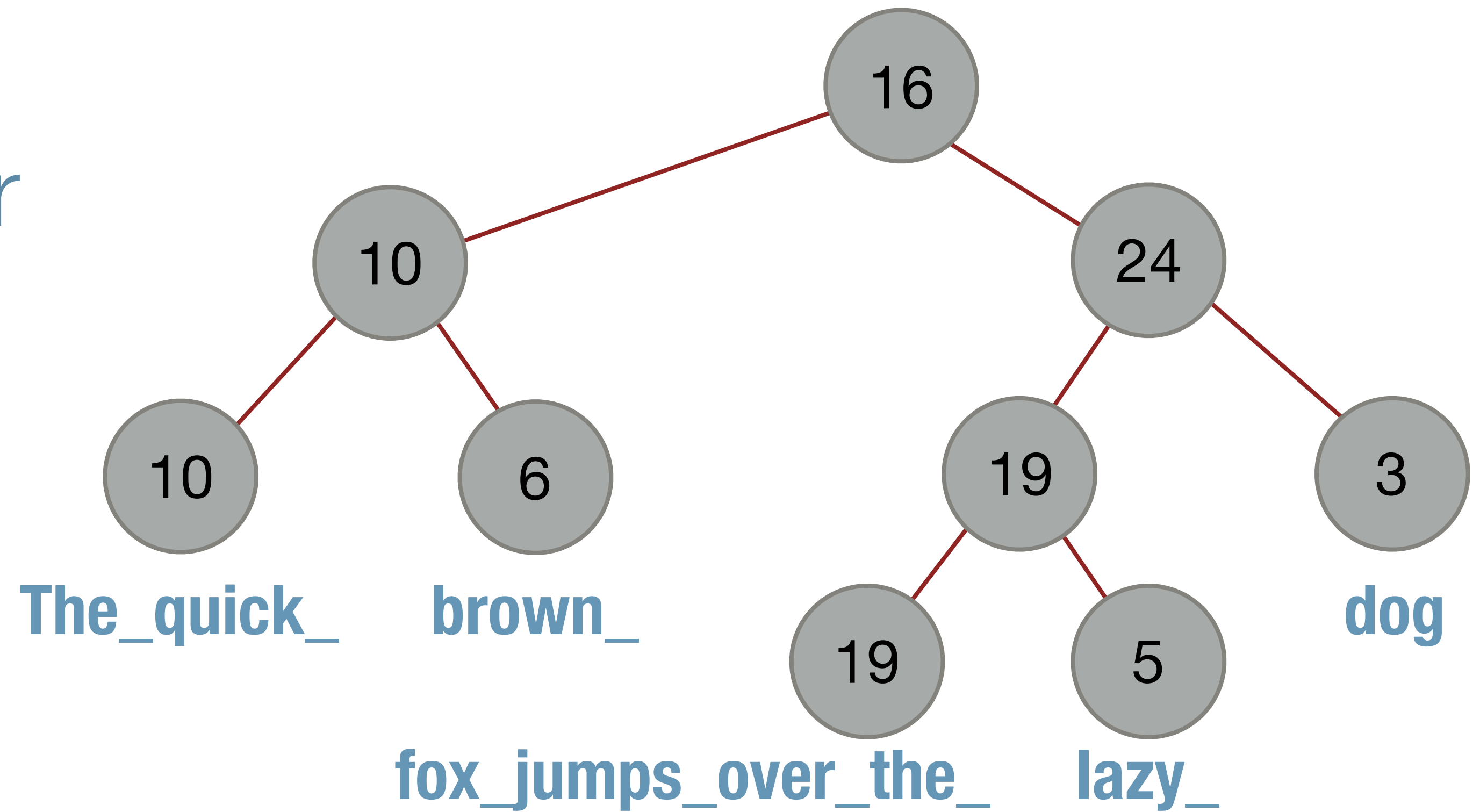
Ropes

Strings are only kept at leaves, and the weight of a node is the length of the string plus the sum of all of the weights in its left subtree.
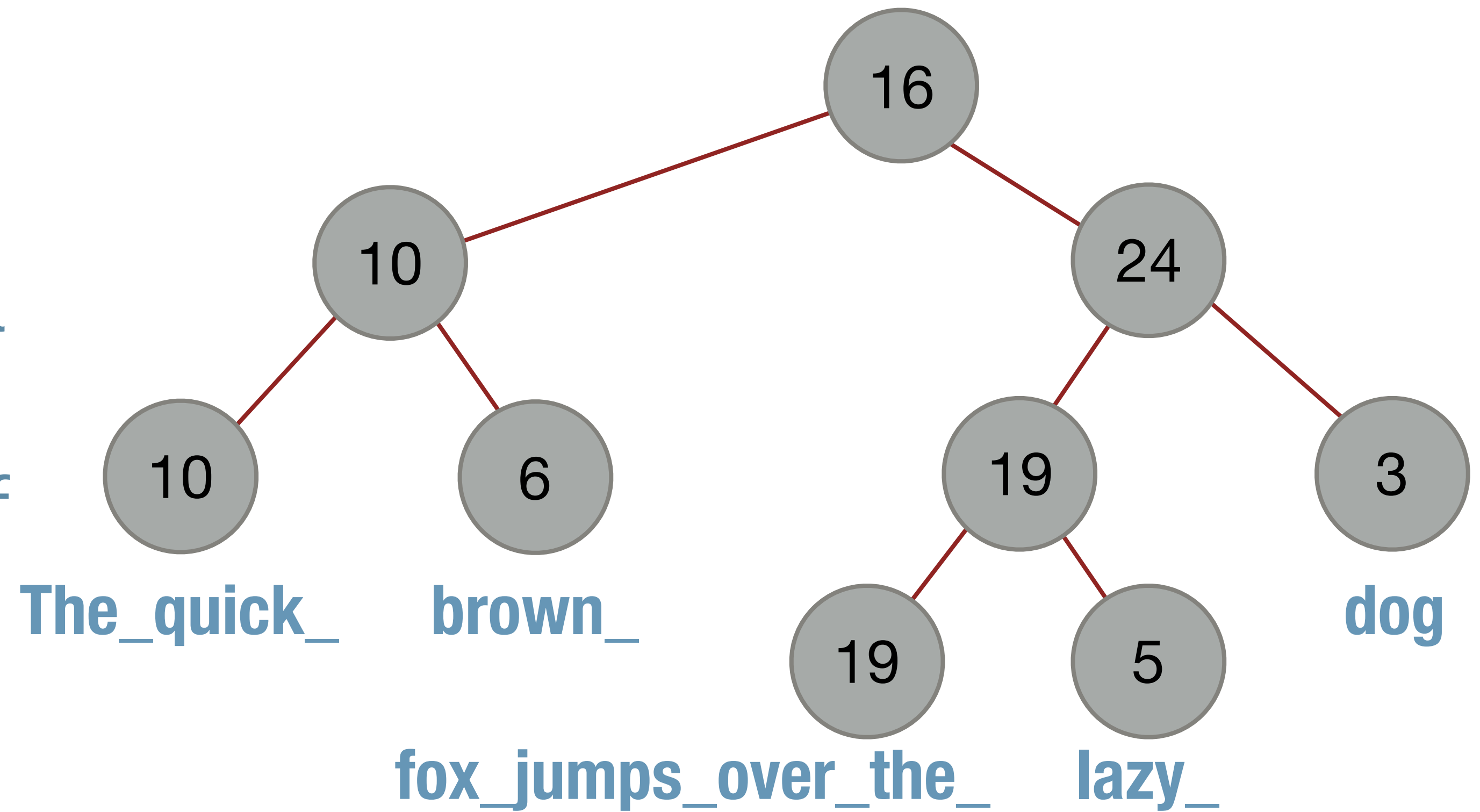
Ropes

Searching for a character at a position, do a recursive search from the root: to search for the "j" at character position 21:

- The root is 16, which is less than 21. We subtract 21-16==5, and we go right.
- 24 > 5, no subtraction (only on right), go left
- 19 > 5, go left.
- 19 > 5, but no more left! The character at the index of the string at that node is "j"
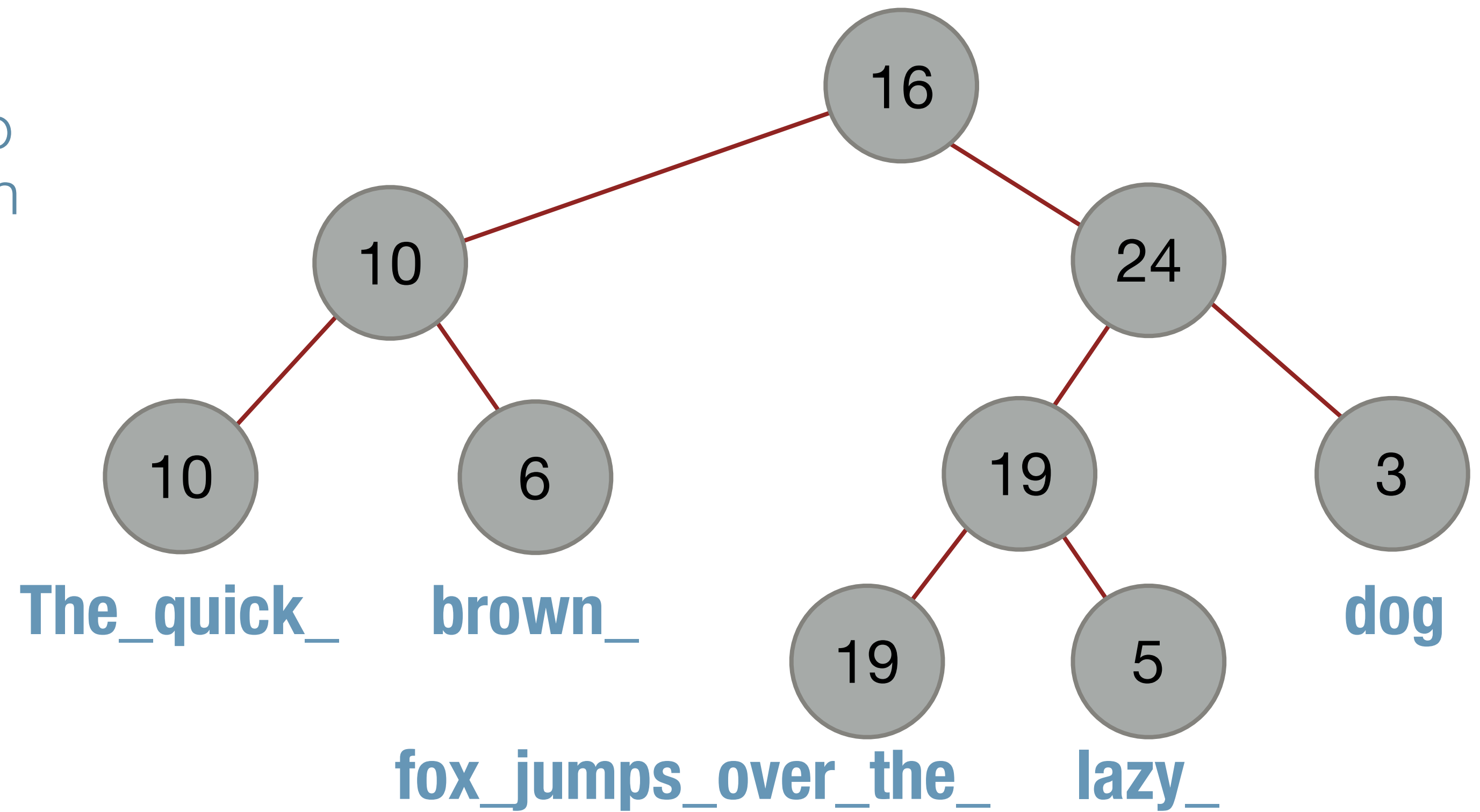
Ropes: Full search algorithm: O(log n)

```
// Note: Assumes 1-based indexing.
function index(RopeNode node, integer i)
    if node.weight < i then
        return index(node.right, i - node.weight)
    else
        if exists(node.left) then
            return index(node.left, i)
        else
            return node.string[i]
        endif
    endif
endif
```
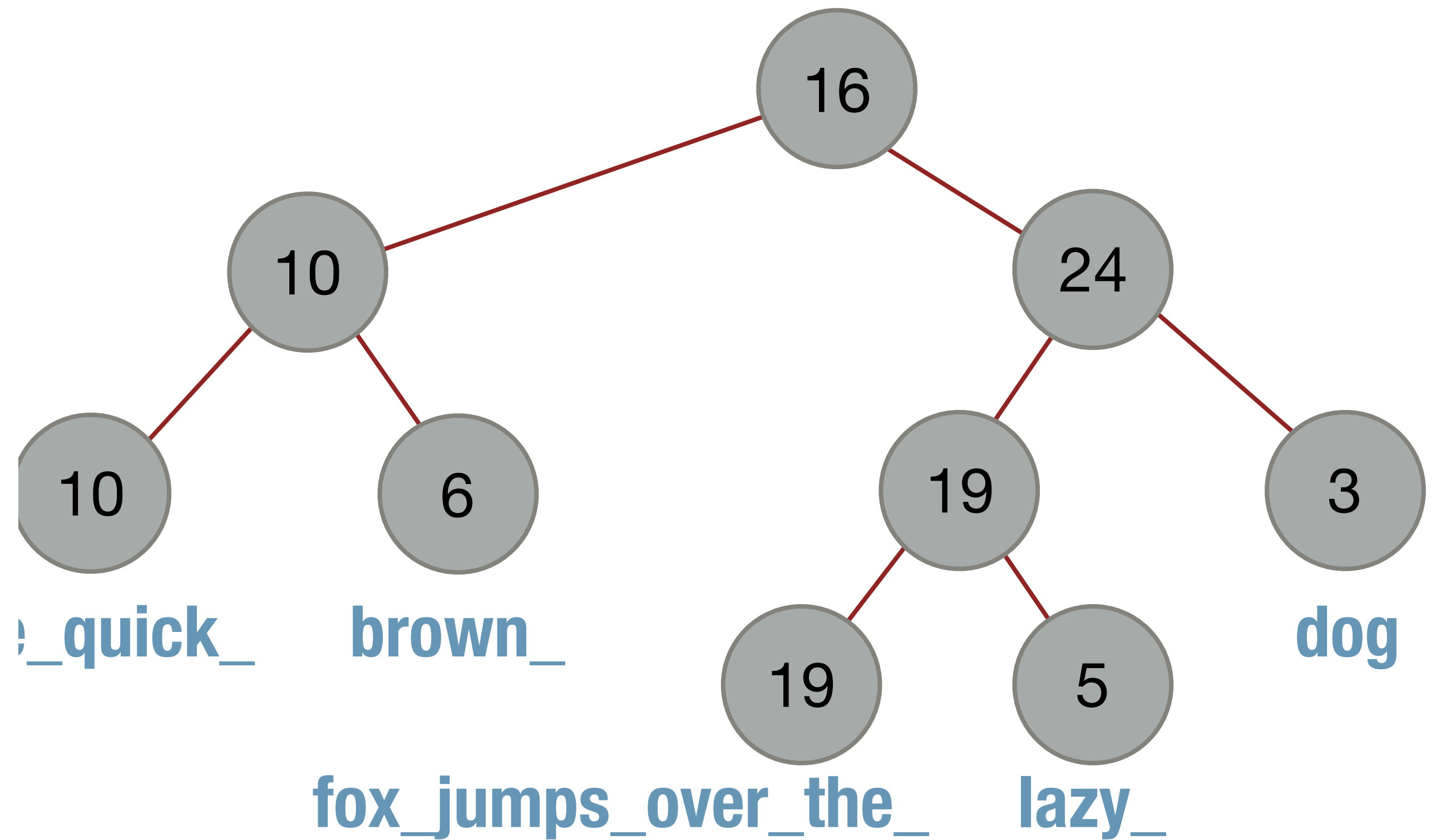
Ropes: Concatenate(S1,S2)
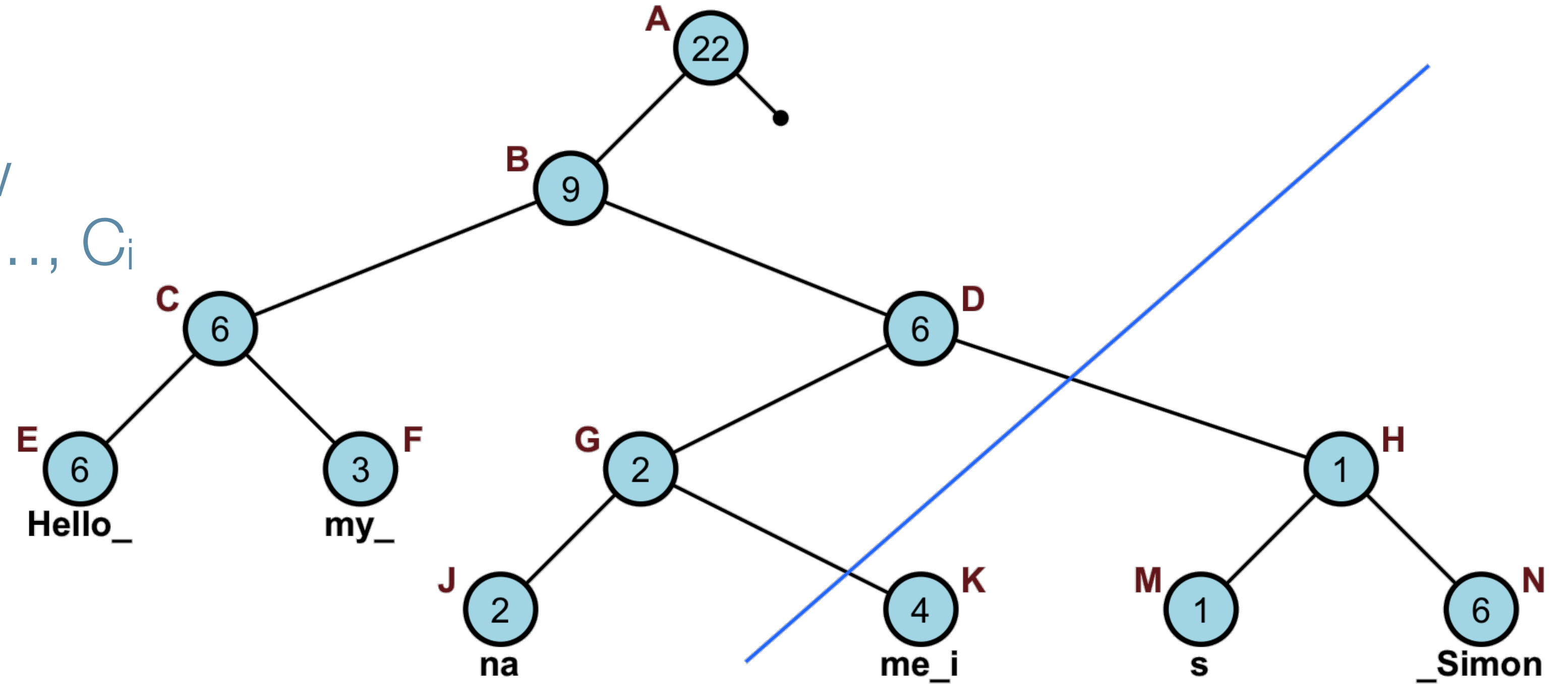Time: O(1) (or O(log N) time to compute the root weight)

Simply create a new root node, with left=S1 and right=S2.
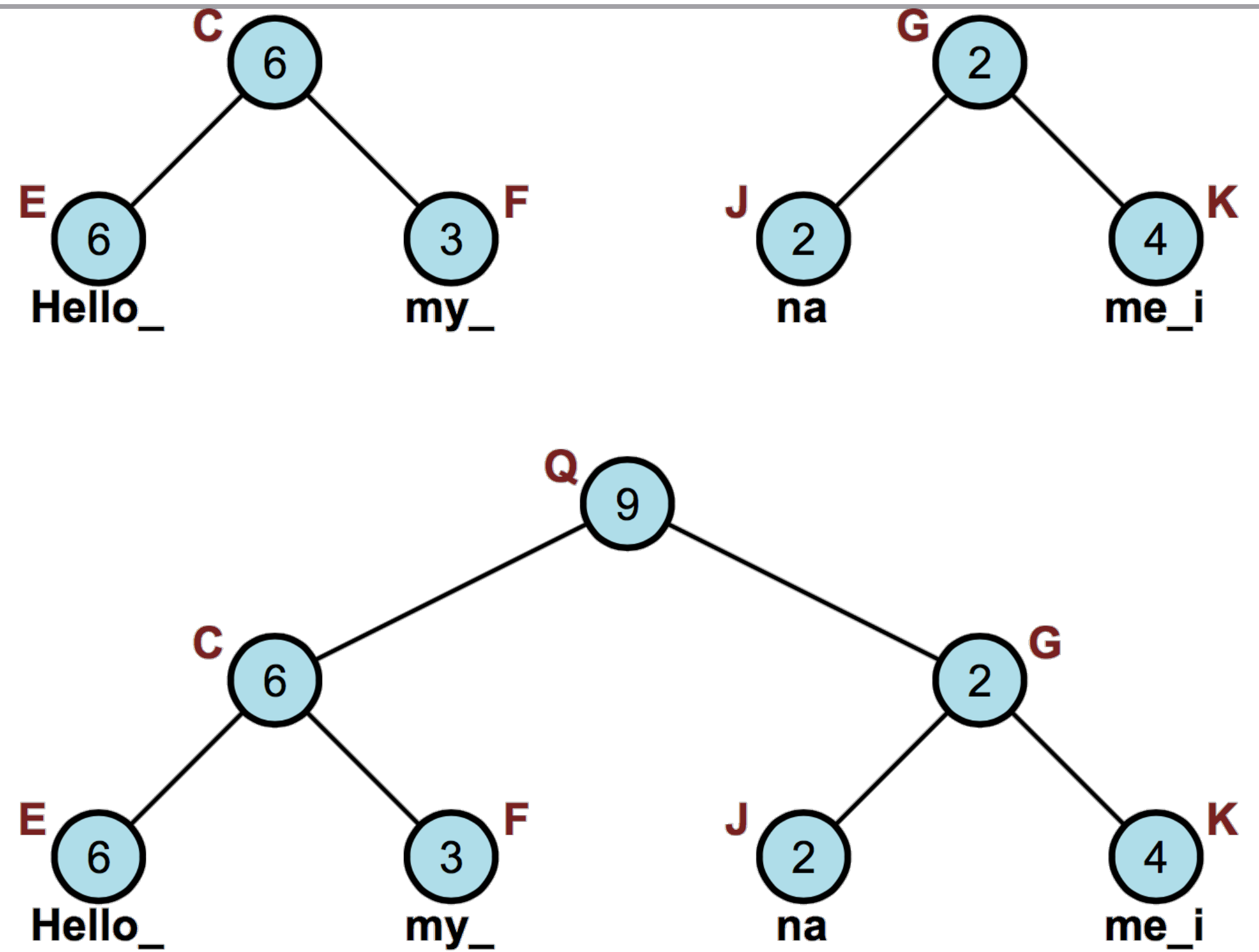
Ropes: Split(i,S)

split the string S into two new
strings S1 and S2, S1 = $C_1, \ldots, C_i$
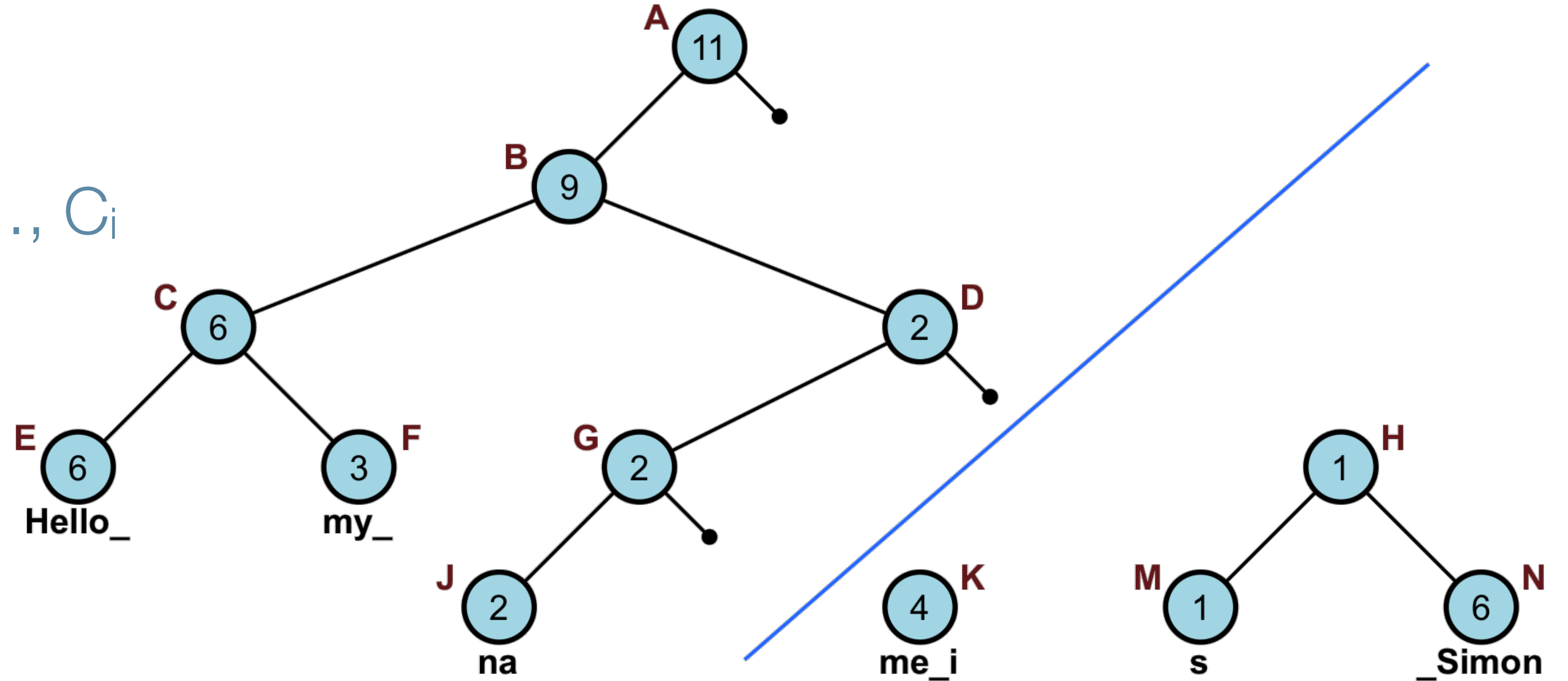and S2 = $C_{i+1}, \ldots, C_m$.
Time complexity: O(log N)

(step 1: split)

Ropes: Split(i,S)

split the string S into two new
strings S1 and S2, S1 = C_1, …, C_i
and S2 = C_{i+1}, …, C_m.
Time complexity: O(log N)

(step 2: update left (node D),
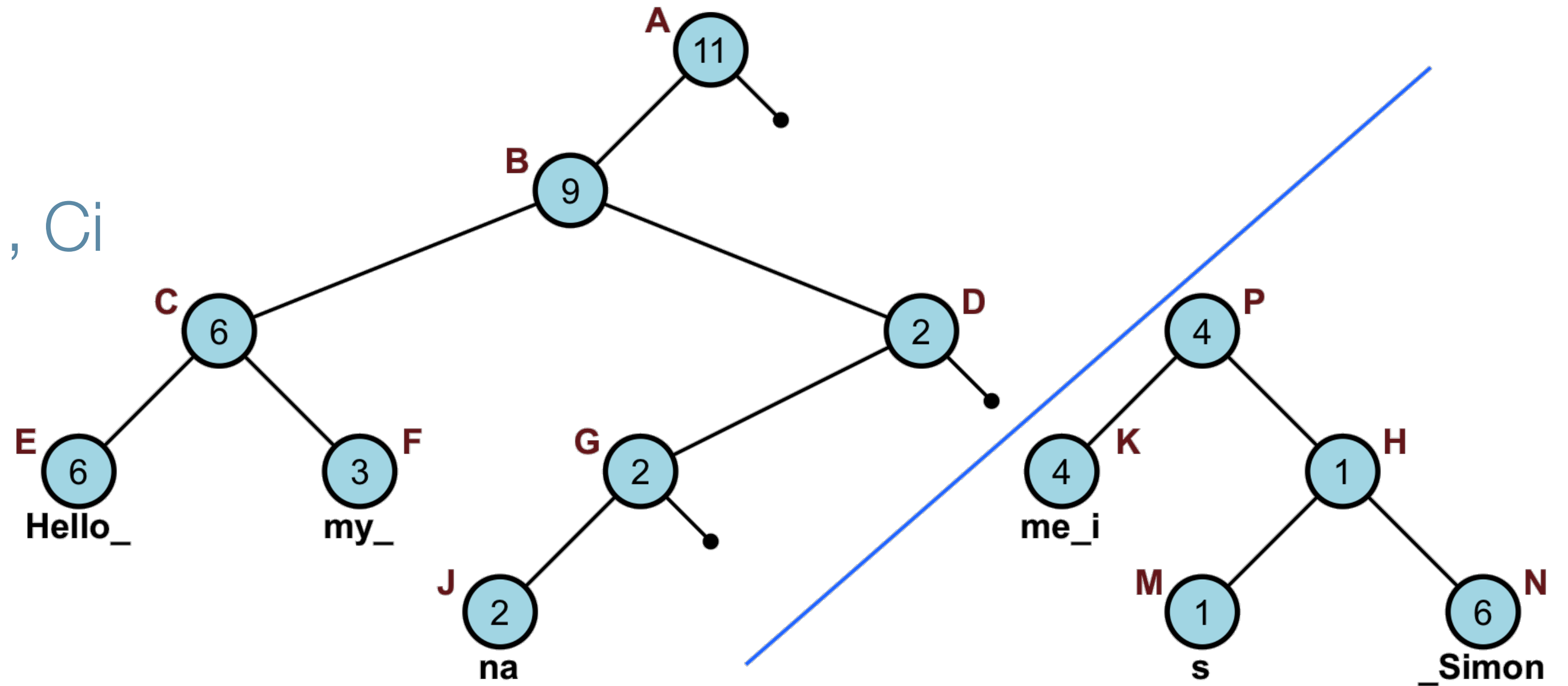and elements on right still need
to be combined)

Ropes: Split(i,S)

split the string S into two new
strings S1 and S2, S1 = C1, ..., Ci
and S2 = C_{i+1}, ..., C_m.
Time complexity: O(log N)

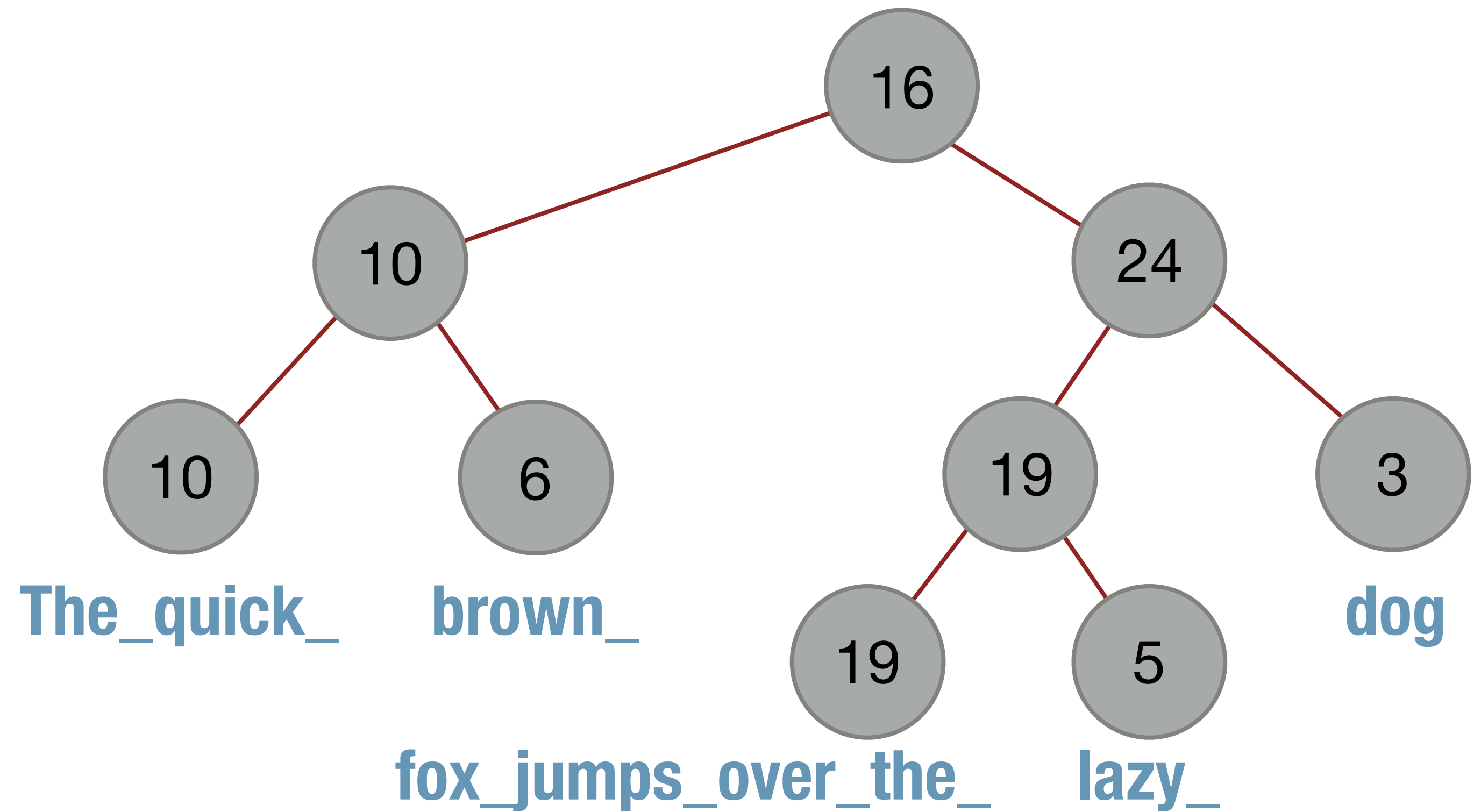(step 3: combine with new root
P for right side)
(may need to balance)

Ropes: Insert(i,S')

insert the string S' beginning at position i in the string s, to form a new string $C_1, \ldots, C_i, S', C_{i+1}, \ldots, C_m$.

Time complexity: $O(\log N)$.

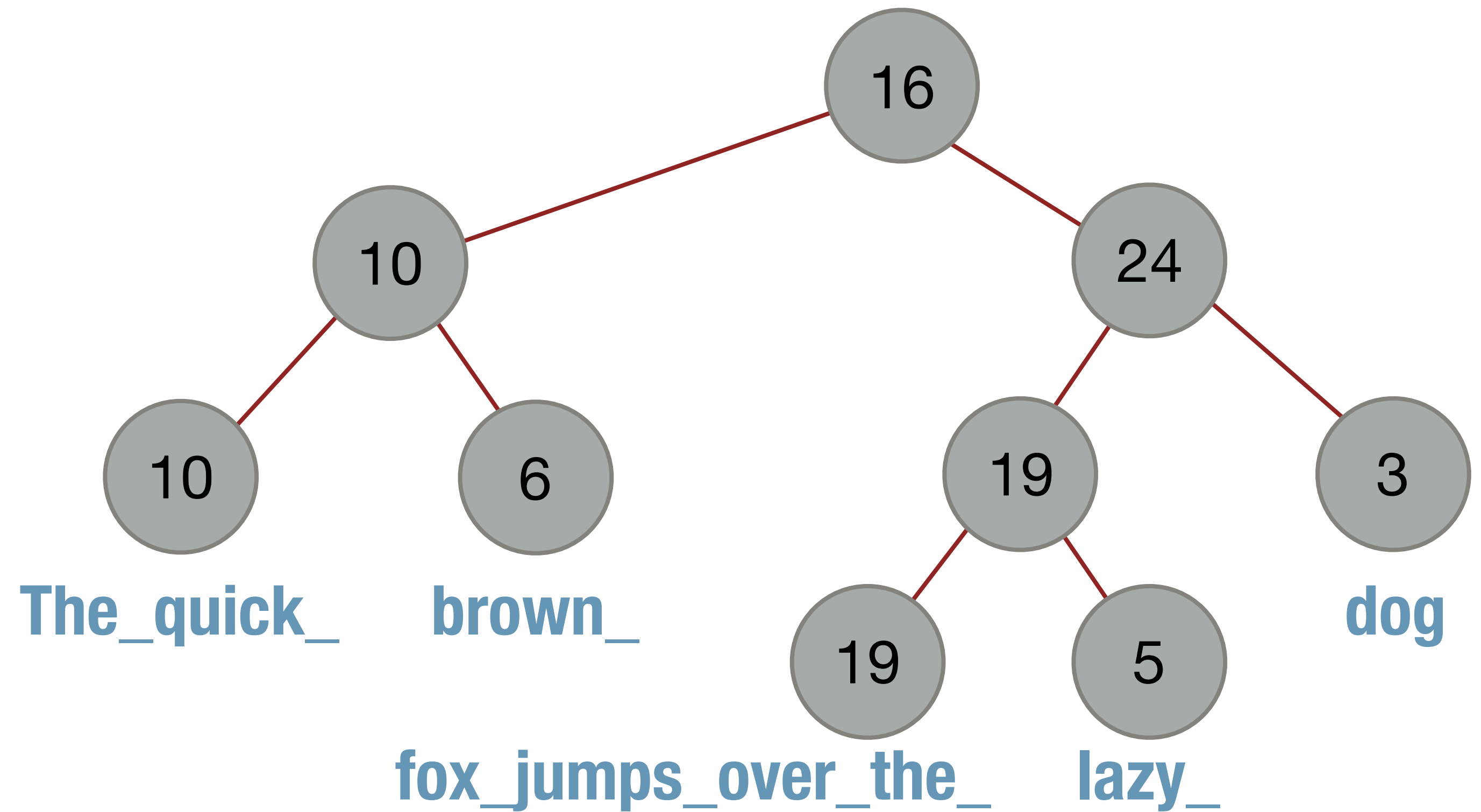Can be done by a Split() and two Concat() operations

Ropes: Delete(i,j)

delete the substring $C_i, \ldots, C_{i+j-1}$, from s to form a new string $C_1, \ldots, C_{i-1}, C_{i+j}, \ldots, C_m$.
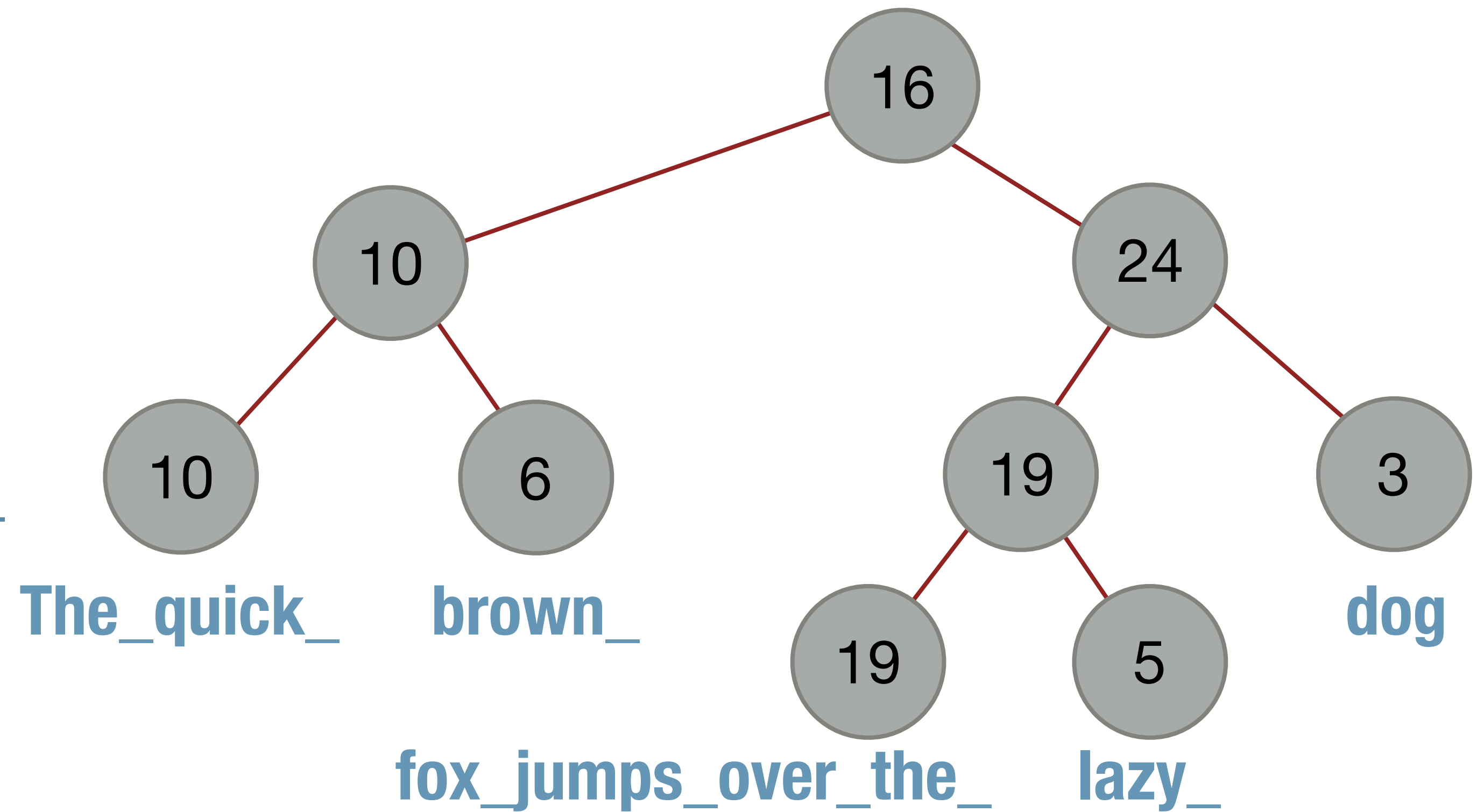Time complexity: $O(\log N)$.

Can be done by two Split() operations and one Concat() operation

16
10
24
10
6
19
3
19
5

The_quick_   brown_   dog

fox_jumps_over_the_   lazy_

Ropes: Report(i,j)

output the string $C_i, \ldots, C_{i+j-1}$.
Time complexity: $O(j + \log N)$

To report the string $C_i, \ldots, C_{i+j-1}$, output $C_i, \ldots, C_{i+j-1}$ by doing an in-order traversal of T starting at the node that has the ith element.

Comparison: Ropes -vs- Strings (from Wikipedia)

Rope Advantages:
- Ropes enable much faster insertion and deletion of text than monolithic string arrays, on which operations have time complexity O(n).
- Ropes don't require O(n) extra memory when operated upon (arrays need that for copying operations)
- Ropes don't require large contiguous memory spaces.
- If only nondestructive versions of operations are used, rope is a persistent data structure. For the text editing program example, this leads to an easy support for multiple undo levels.

Comparison: Ropes -vs- Strings (from Wikipedia)

Rope Disadvantages:
- Greater overall space usage when not being operated on, mainly to store parent nodes. There is a trade-off between how much of the total memory is such overhead and how long pieces of data are being processed as strings; note that the strings in example figures above are unrealistically short for modern architectures. The overhead is always O(n), but the constant can be made arbitrarily small.
- Increase in time to manage the extra storage
- Increased complexity of source code; greater risk for bugs