

SYMBOL TABLES

- ▶ API
- ▶ sequential search
- ▶ binary search
- ▶ ordered operations

► API

- sequential search
- binary search
- ordered operations

Symbol tables

Key-value pair abstraction.

- Insert a value with specified key.
- Given a key, search for the corresponding value.

Ex. DNS lookup.

- Insert URL with specified IP address.
- Given URL, find corresponding IP address.

URL	IP address
www.cs.princeton.edu	128.112.136.11
www.princeton.edu	128.112.128.15
www.yale.edu	130.132.143.21
www.harvard.edu	128.103.060.55
www.simpsons.com	209.052.165.60

key
↑

value
↑

Symbol table applications

application	purpose of search	key	value
dictionary	find definition	word	definition
book index	find relevant pages	term	list of page numbers
file share	find song to download	name of song	computer ID
financial account	process transactions	account number	transaction details
web search	find relevant web pages	keyword	list of page names
compiler	find properties of variables	variable name	type and value
routing table	route Internet packets	destination	best route
DNS	find IP address given URL	URL	IP address
reverse DNS	find URL given IP address	IP address	URL
genomics	find markers	DNA string	known positions
file system	find file on disk	filename	location on disk

Basic symbol table API

Associative array abstraction. Associate one value with each key.

```
public class ST<Key, Value>
```

```
    ST()
```

create a symbol table

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

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

← **a[key] = val;**

```
    Value get(Key key)
```

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

← **a[key]**

```
    void delete(Key key)
```

remove key (and its value) from table

```
    boolean contains(Key key)
```

is there a value paired with key?

```
    boolean isEmpty()
```

is the table empty?

```
    int size()
```

number of key-value pairs in the table

```
    Iterable<Key> keys()
```

all the keys in the table

Conventions

- Values are not `null`.
- Method `get()` returns `null` if key not present.
- Method `put()` overwrites old value with new value.

Intended consequences.

- Easy to implement `contains()`.

```
public boolean contains(Key key)
{   return get(key) != null; }
```

- Can implement lazy version of `delete()`.

```
public void delete(Key key)
{   put(key, null); }
```

Keys and values

Value type. Any generic type.

Key type: several natural assumptions.

- Assume keys are Comparable, use compareTo().
- Assume keys are any generic type, use equals() to test equality.
- Assume keys are any generic type, use equals() to test equality;

specify Comparable in API.



ST test client for traces

Build ST by associating value i with i^{th} string from standard input.

```
public static void main(String[] args)
{
    ST<String, Integer> st = new ST<String, Integer>();
    for (int i = 0; !StdIn.isEmpty(); i++)
    {
        String key = StdIn.readString();
        st.put(key, i);
    }
    for (String s : st.keys())
        StdOut.println(s + " " + st.get(s));
}
```

output

keys	S	E	A	R	C	H	E	X	A	M	P	L	E
values	0	1	2	3	4	5	6	7	8	9	10	11	12

A	8
C	4
E	12
H	5
L	11
M	9
P	10
R	3
S	0
X	7

ST test client for analysis

Frequency counter. Read a sequence of strings from standard input and print out one that occurs with highest frequency.

```
% more tinyTale.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
```

```
% java FrequencyCounter 1 < tinyTale.txt  
it 10
```

```
% java FrequencyCounter 8 < tale.txt  
business 122
```

```
% java FrequencyCounter 10 < leipzig1M.txt  
government 24763
```

tiny example
(60 words, 20 distinct)

real example
(135,635 words, 10,769 distinct)

real example
(21,191,455 words, 534,580 distinct)

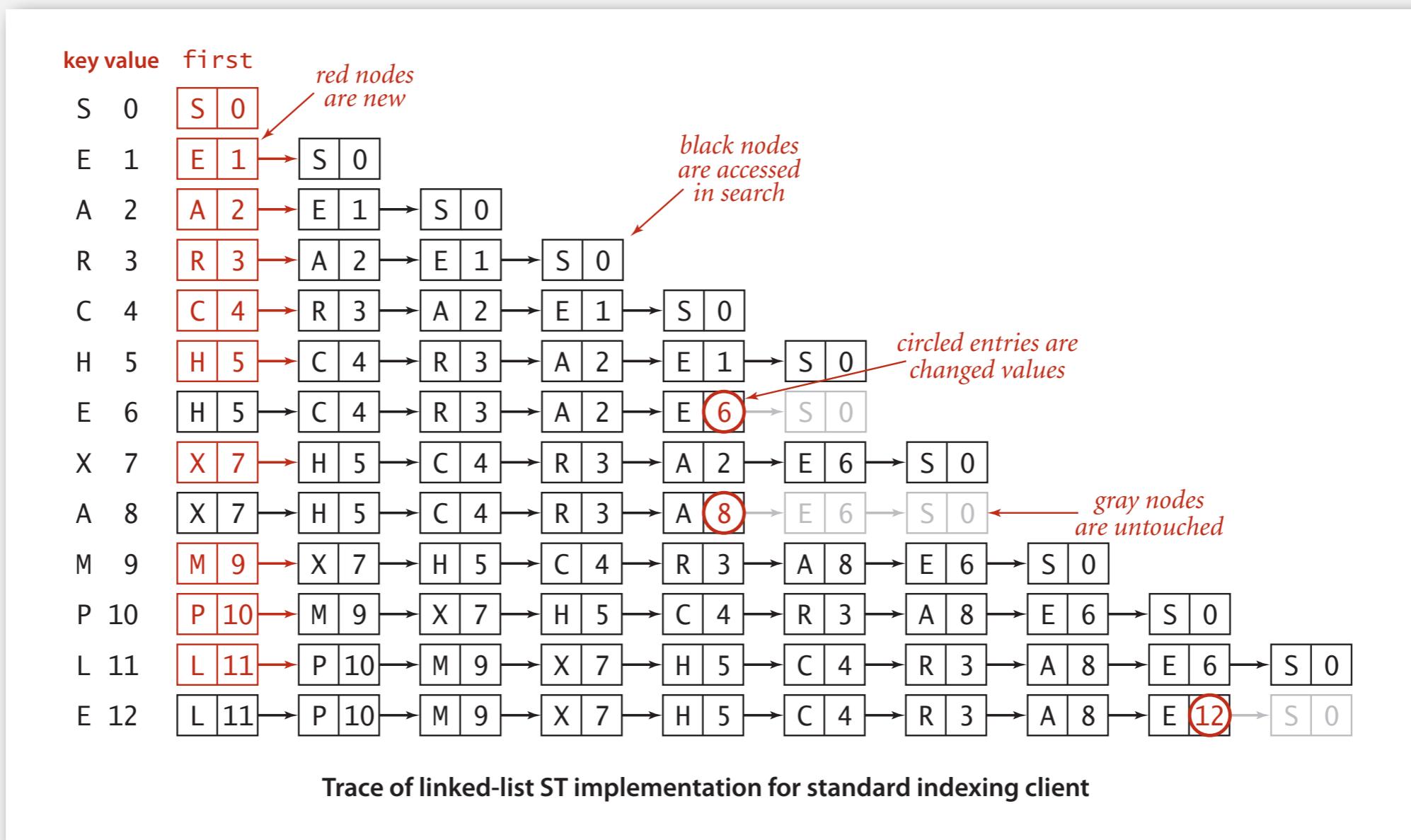
- ▶ API
- ▶ **sequential search**
- ▶ **binary search**
- ▶ **ordered operations**

Sequential search in a linked list

Data structure. Maintain an (unordered) linked list of key-value pairs.

Search. Scan through all keys until find a match.

Insert. Scan through all keys until find a match; if no match add to front.



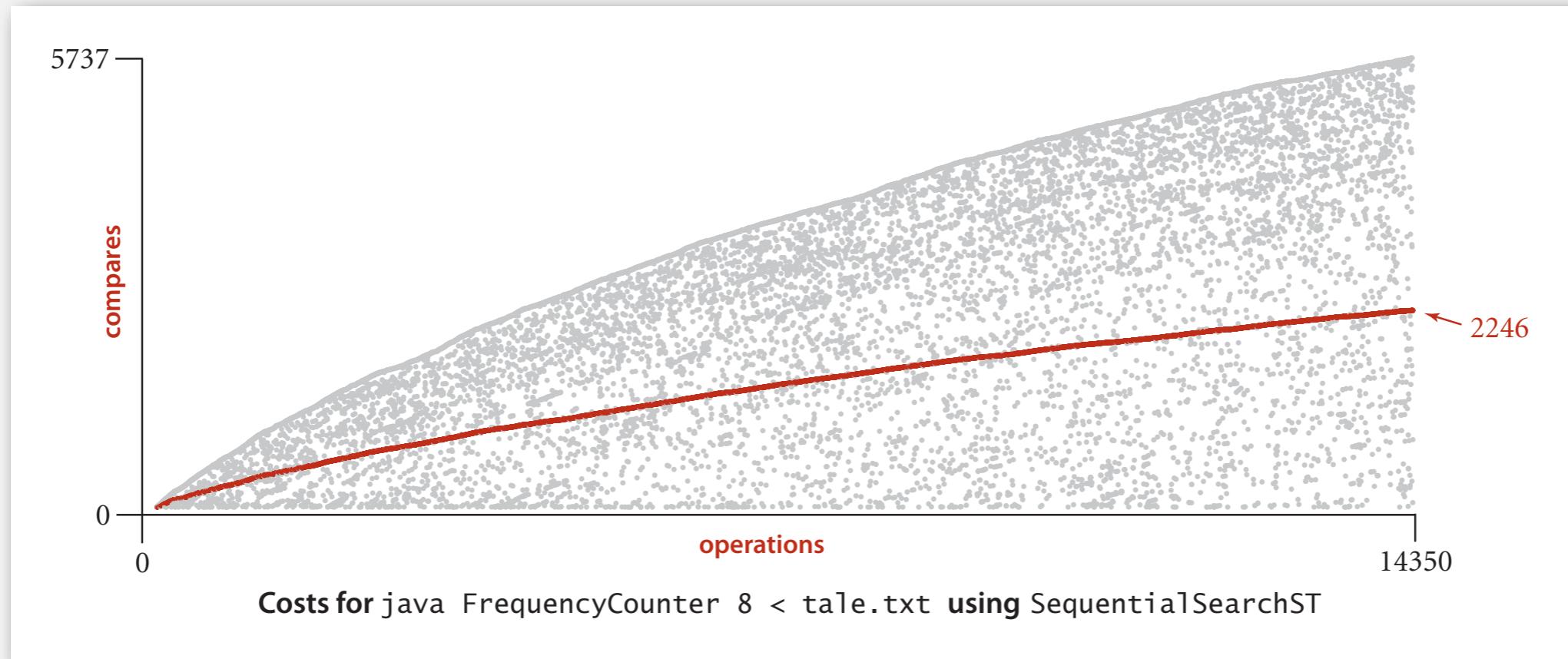
Elementary ST implementations: summary

ST implementation	worst-case cost (after N inserts)		average case (after N random inserts)		ordered iteration?	key interface
	search	insert	search hit	insert		
sequential search (unordered list)	N	N	N / 2	N	no	<code>equals()</code>

Challenge. Efficient implementations of both search and insert.

Elementary ST implementations: summary

ST implementation	worst case		average case		ordered iteration?	operations on keys
	search	insert	search hit	insert		
sequential search (unordered list)	N	N	N / 2	N	no	<code>equals()</code>



Challenge. Efficient implementations of both search and insert.

- ▶ API
- ▶ sequential search
- ▶ **binary search**
- ▶ ordered symbol table ops

Binary search

Data structure. Maintain an ordered array of key-value pairs.

Rank helper function. How many keys $< k$?

- Rank: Given a sorted array, determine the index associated with given key
 - If key is not in table, return number of keys that are smaller than key

keys[]										
	0	1	2	3	4	5	6	7	8	9
successful search for P	0	9	4	A	C	E	H	L	M	P
	5	9	7	A	C	E	H	L	M	P
	5	6	5	A	C	E	H	L	M	P
	6	6	6	A	C	E	H	L	M	P
unsuccessful search for Q	0	9	4	A	C	E	H	L	M	P
	5	9	7	A	C	E	H	L	M	P
	5	6	5	A	C	E	H	L	M	P
	7	6	6	A	C	E	H	L	M	P

entries in black are $a[lo..hi]$

entry in red is $a[m]$

loop exits with $keys[m] = P$: return 6

loop exits with $lo > hi$: return 7

Select: Find the key with a given rank

Binary search: mathematical analysis

Proposition. Binary search uses $\sim \lg N$ compares to search any array of size N .

Pf. $T(N) =$ number of compares to binary search in a sorted array of size N .

$$\leq T(\lfloor N/2 \rfloor) + 1$$



left or right half

Somewhat similar to mergesort

Binary search: trace of standard indexing client

Problem. To insert, need to shift all greater keys over.

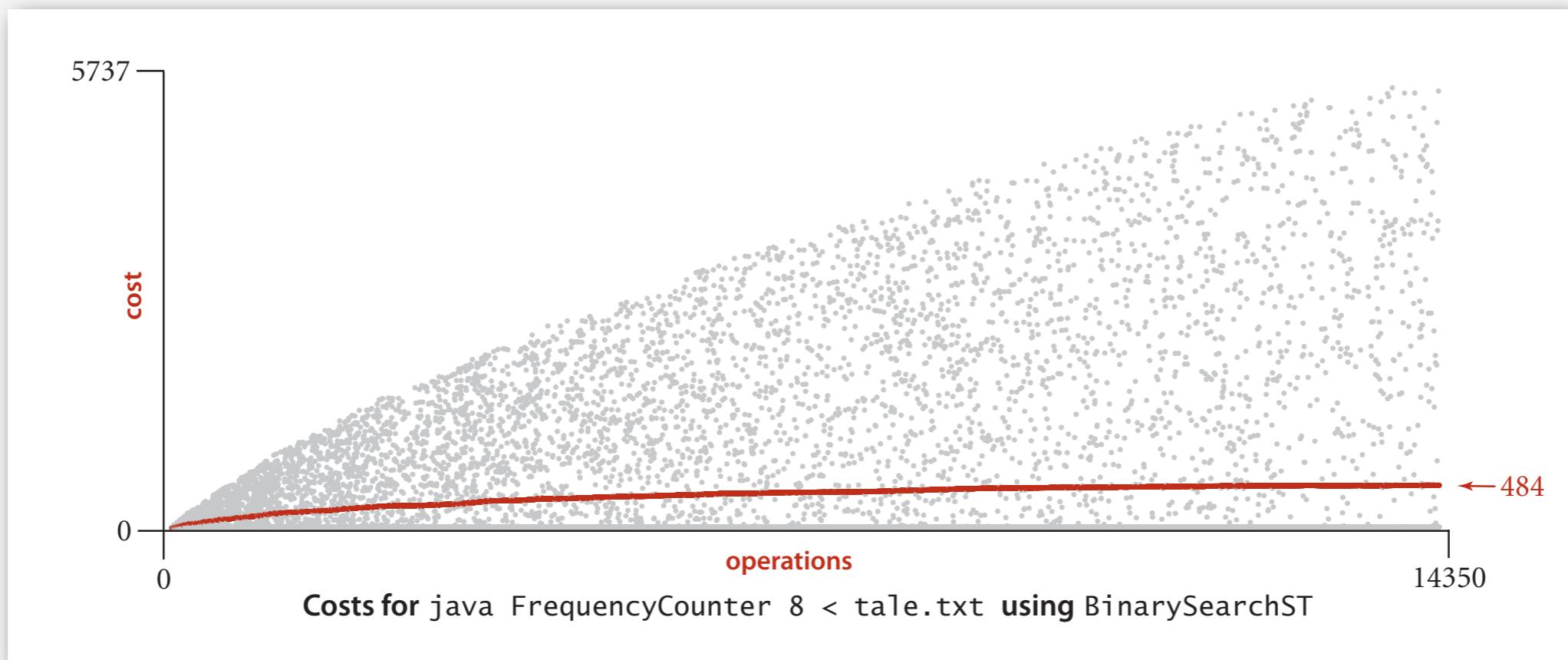
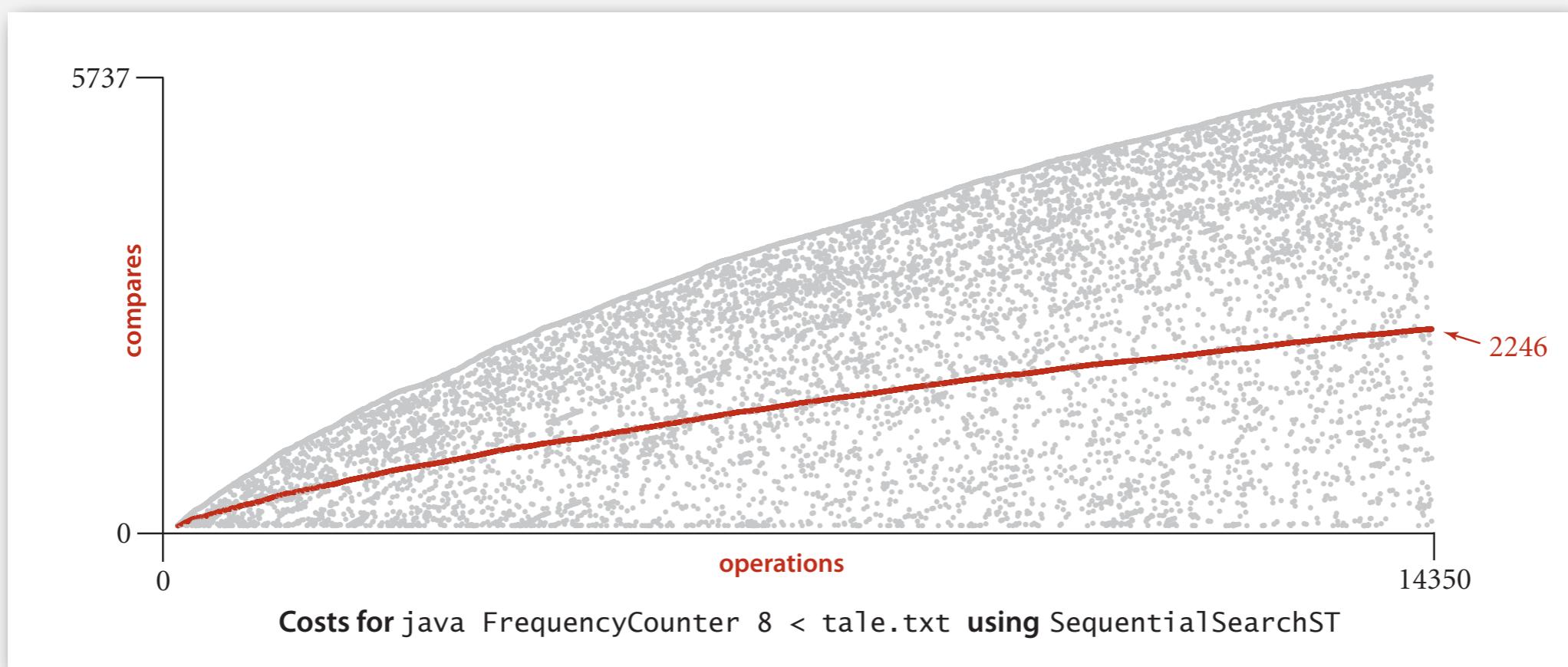
`put()` = rank tells us where to update value when key in table, and where to put the key when not in table

		keys []										vals []										
key	value	0	1	2	3	4	5	6	7	8	9	N	0	1	2	3	4	5	6	7	8	9
S	0	S										1	0									
E	1	E	S									2	1	0								
A	2	A	E	S								3	2	1	0							
R	3	A	E	R	S							4	2	1	3	0						
C	4	A	C	E	R	S						5	2	4	1	3	0					
H	5	A	C	E	H	R	S					6	2	4	1	5	3	0				
E	6	A	C	E	H	R	S					6	2	4	6	5	3	0				
X	7	A	C	E	H	R	S	X				7	2	4	6	5	3	0	7			
A	8	A	C	E	H	R	S	X				7	8	4	6	5	3	0	7			
M	9	A	C	E	H	M	R	S	X			8	8	4	6	5	9	3	0	7		
P	10	A	C	E	H	M	P	R	S	X		9	8	4	6	5	9	10	3	0	7	
L	11	A	C	E	H	L	M	P	R	S	X	10	8	4	6	5	11	9	10	3	0	7
E	12	A	C	E	H	L	M	P	R	S	X	10	8	4	12	5	11	9	10	3	0	7
		A	C	E	H	L	M	P	R	S	X		8	4	12	5	11	9	10	3	0	7

Annotations:

- Red text: "entries in red were inserted" points to the row for key 12.
- Gray text: "entries in gray did not move" points to the row for key 11.
- Red arrow: Points from the text "entries in black moved to the right" to the circled entry 6 in the val[6] column.
- Red circle: Circles the entry 12 in the val[12] column, with the text "circled entries are changed values" pointing to it.

Elementary ST implementations: frequency counter



Elementary ST implementations: summary

ST implementation	worst-case cost (after N inserts)		average case (after N random inserts)		ordered iteration?	key interface
	search	insert	search hit	insert		
sequential search (unordered list)	N	N	N / 2	N	no	<code>equals()</code>
binary search (ordered array)	$\log N$	N	$\log N$	N / 2	yes	<code>compareTo()</code>

Challenge. Efficient implementations of both search and insert.

- ▶ API
- ▶ sequential search
- ▶ binary search
- ▶ ordered operations

Ordered symbol table API

	<i>keys</i>	<i>values</i>
min()	→ 09:00:00	Chicago
	09:00:03	Phoenix
	09:00:13	→ Houston
get(09:00:13)	→ 09:00:59	Chicago
	09:01:10	Houston
floor(09:05:00)	→ 09:03:13	Chicago
	09:10:11	Seattle
select(7)	→ 09:10:25	Seattle
	09:14:25	Phoenix
	09:19:32	Chicago
	09:19:46	Chicago
keys(09:15:00, 09:25:00)	→ 09:21:05	Chicago
	09:22:43	Seattle
	09:22:54	Seattle
	09:25:52	Chicago
ceiling(09:30:00)	→ 09:35:21	Chicago
	09:36:14	Seattle
max()	→ 09:37:44	Phoenix
size(09:15:00, 09:25:00)	is 5	
rank(09:10:25)	is 7	

Examples of ordered symbol-table operations

Ordered symbol table API

public class ST<Key extends Comparable<Key>, Value>	
ST()	<i>create an ordered symbol table</i>
void put(Key key, Value val)	<i>put key-value pair into the table (remove key from table if value is null)</i>
Value get(Key key)	<i>value paired with key (null if key is absent)</i>
void delete(Key key)	<i>remove key (and its value) from table</i>
boolean contains(Key key)	<i>is there a value paired with key?</i>
boolean isEmpty()	<i>is the table empty?</i>
int size()	<i>number of key-value pairs</i>
Key min()	<i>smallest key</i>
Key max()	<i>largest key</i>
Key floor(Key key)	<i>largest key less than or equal to key</i>
Key ceiling(Key key)	<i>smallest key greater than or equal to key</i>
int rank(Key key)	<i>number of keys less than key</i>
Key select(int k)	<i>key of rank k</i>
void deleteMin()	<i>delete smallest key</i>
void deleteMax()	<i>delete largest key</i>
int size(Key lo, Key hi)	<i>number of keys in [lo..hi]</i>
Iterable<Key> keys(Key lo, Key hi)	<i>keys in [lo..hi], in sorted order</i>
Iterable<Key> keys()	<i>all keys in the table, in sorted order</i>

Binary search: ordered symbol table operations summary

	sequential search	binary search
search	N	$\lg N$
insert	1	N
min / max	N	1
floor / ceiling	N	$\lg N$
rank	N	$\lg N$
select	N	1
ordered iteration	$N \log N$	N

order of growth of the running time for ordered symbol table operations

BINARY SEARCH TREES

- ▶ BSTs
- ▶ ordered operations
- ▶ deletion

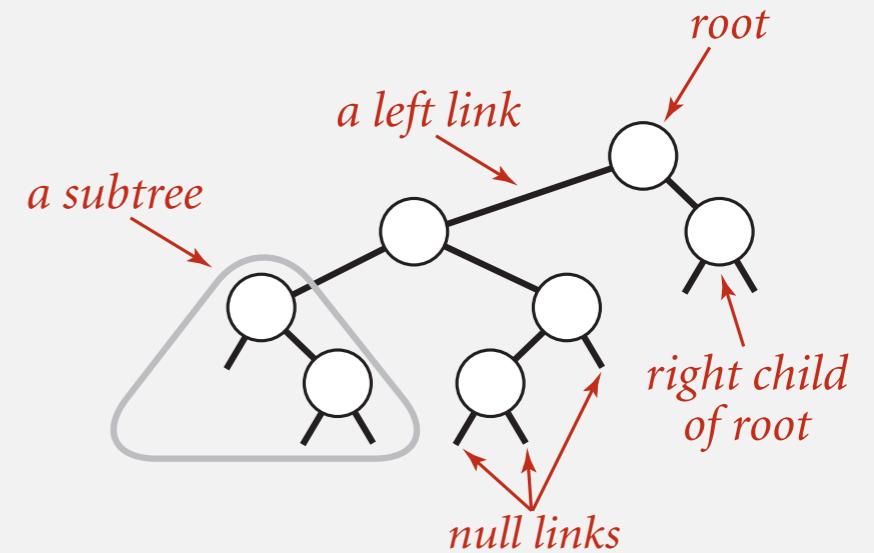
- ▶ BSTs
- ▶ ordered operations
- ▶ deletion

Binary search trees

Definition. A BST is a binary tree in **symmetric order**.

A binary tree is either:

- Empty.
- Two disjoint binary trees (left and right).

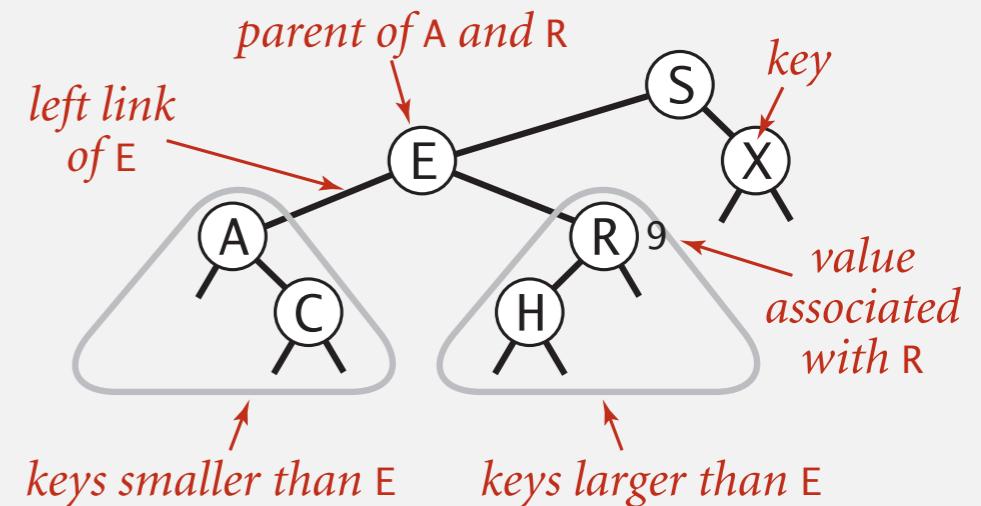


Anatomy of a binary tree

Symmetric order. Each node has a **key**,

and every node's key is:

- Larger than all keys in its left subtree.
- Smaller than all keys in its right subtree.

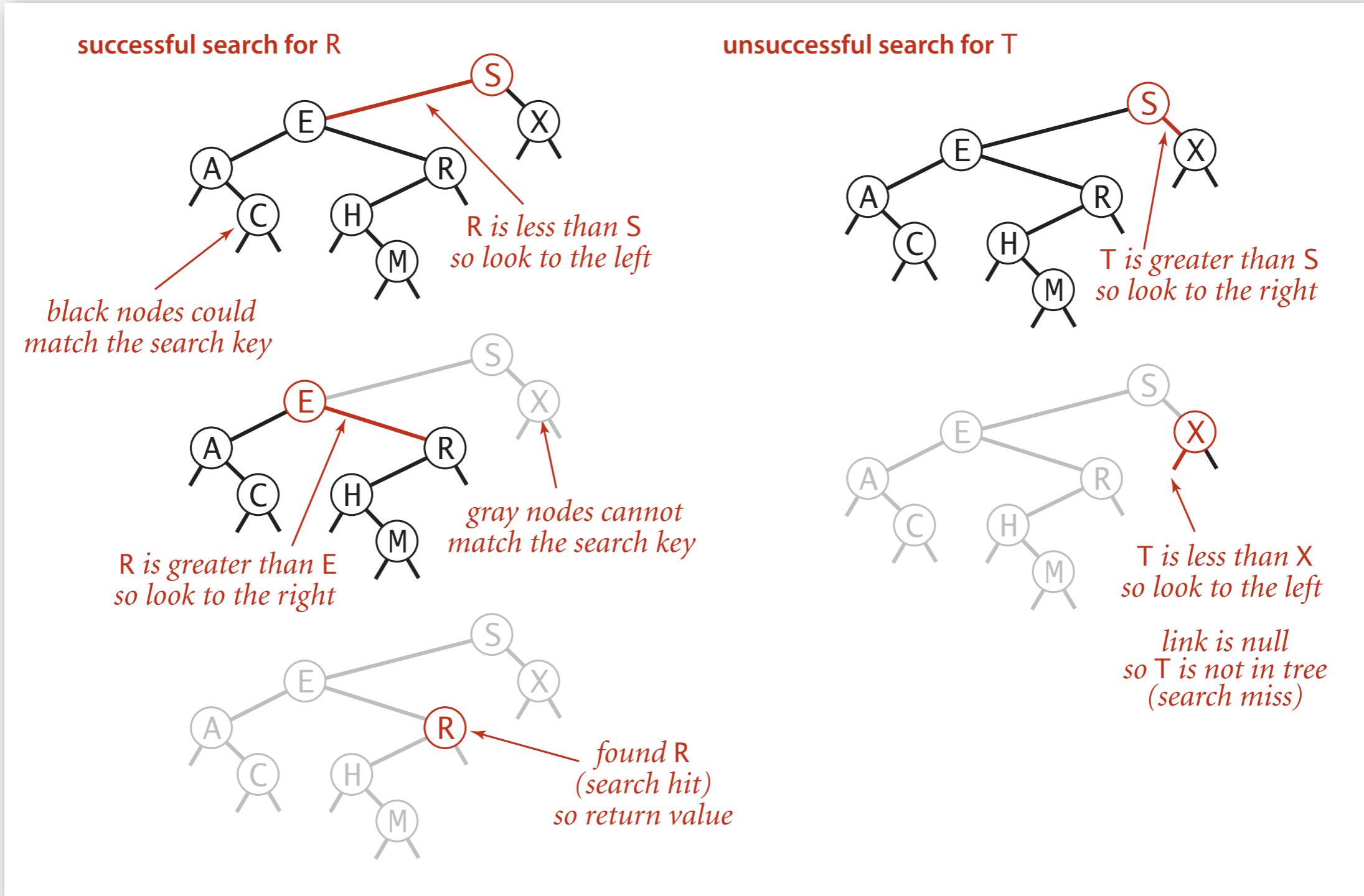


Anatomy of a binary search tree

BST search and insert demo

BST search

Get. Return value corresponding to given key, or `null` if no such key.



BST search

Get. Return value corresponding to given key, or `null` if no such key.

```
public Value get(Key key)
{
    Node x = root;
    while (x != null)
    {
        int cmp = key.compareTo(x.key);
        if      (cmp < 0) x = x.left;
        else if (cmp > 0) x = x.right;
        else if (cmp == 0) return x.val;
    }
    return null;
}
```

Cost. Number of compares is equal to $1 + \text{depth of node}$.

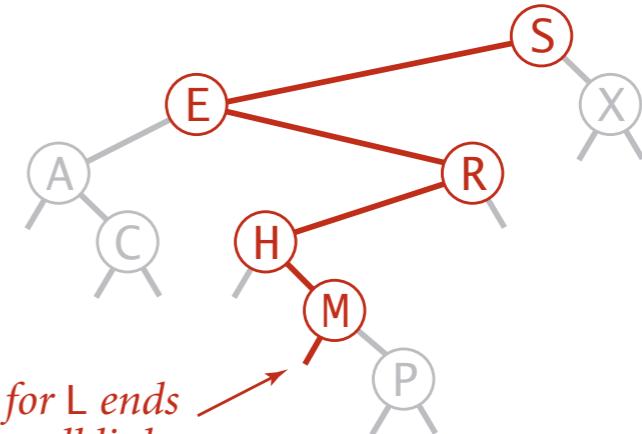
BST insert

Put. Associate value with key.

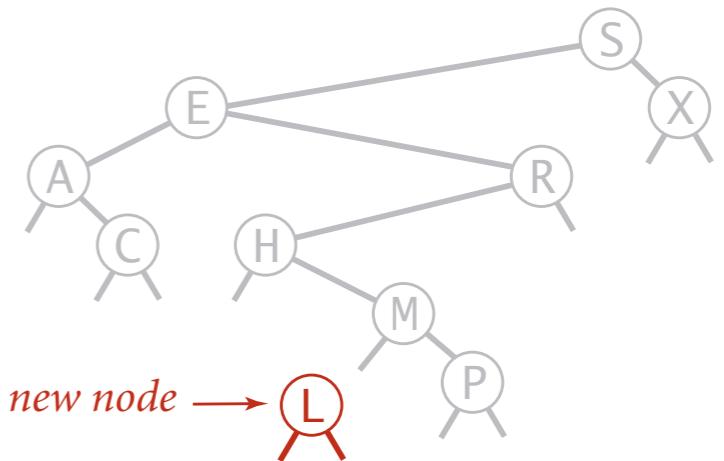
Search for key, then two cases:

- Key in tree \Rightarrow reset value.
- Key not in tree \Rightarrow add new node.

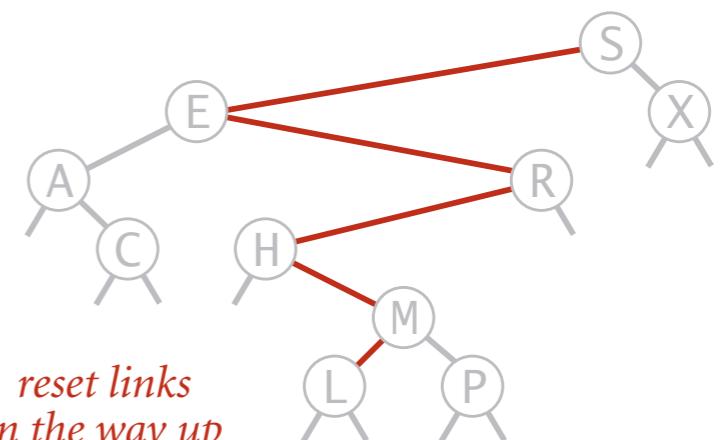
inserting L



search for L ends
at this null link



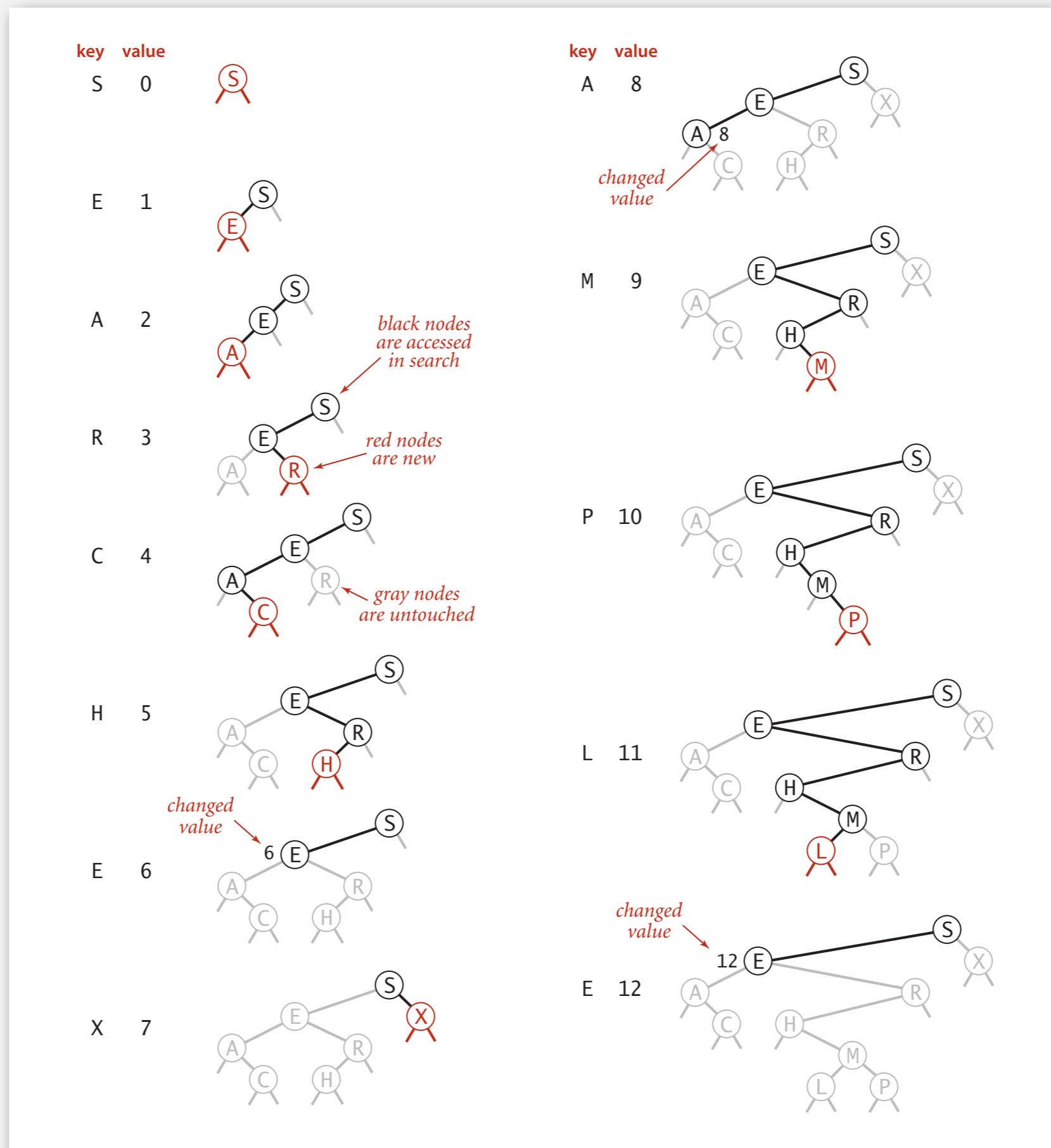
create new node



reset links
on the way up

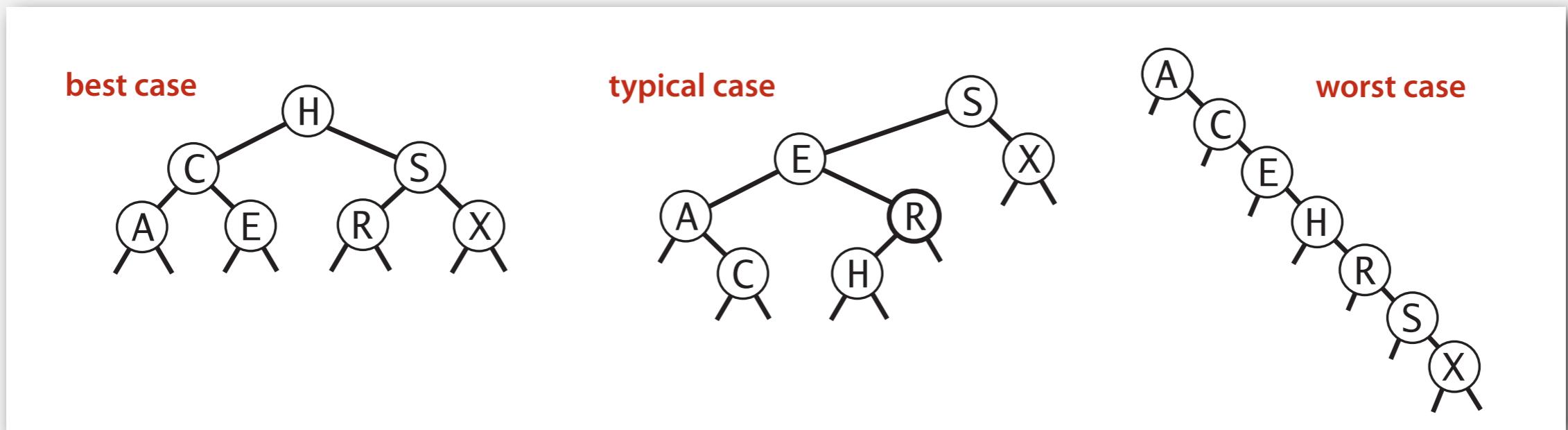
Insertion into a BST

BST trace: standard indexing client



Tree shape

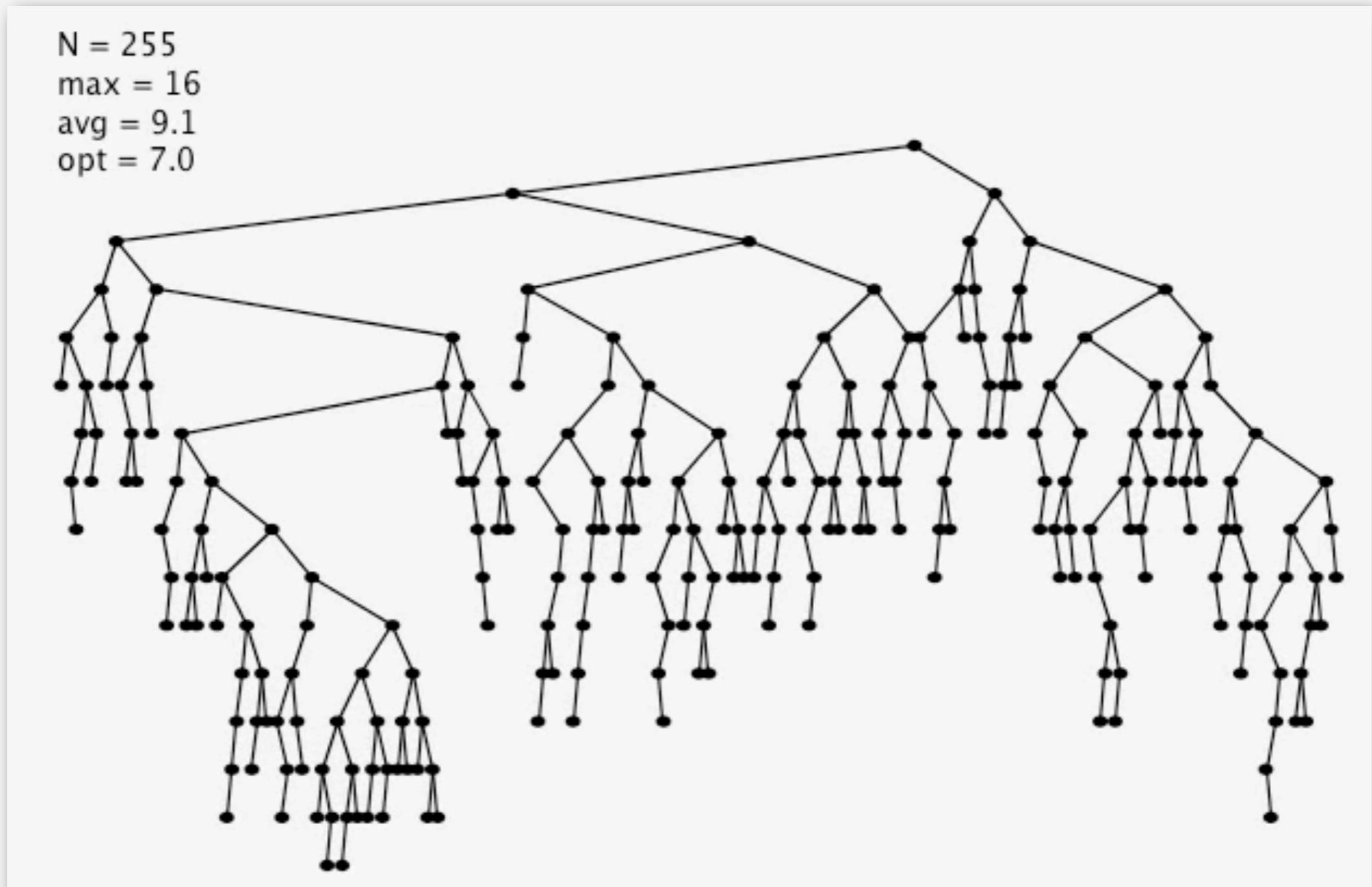
- Many BSTs correspond to same set of keys.
- Number of compares for search/insert is equal to $1 + \text{depth of node}$.



Remark. Tree shape depends on order of insertion.

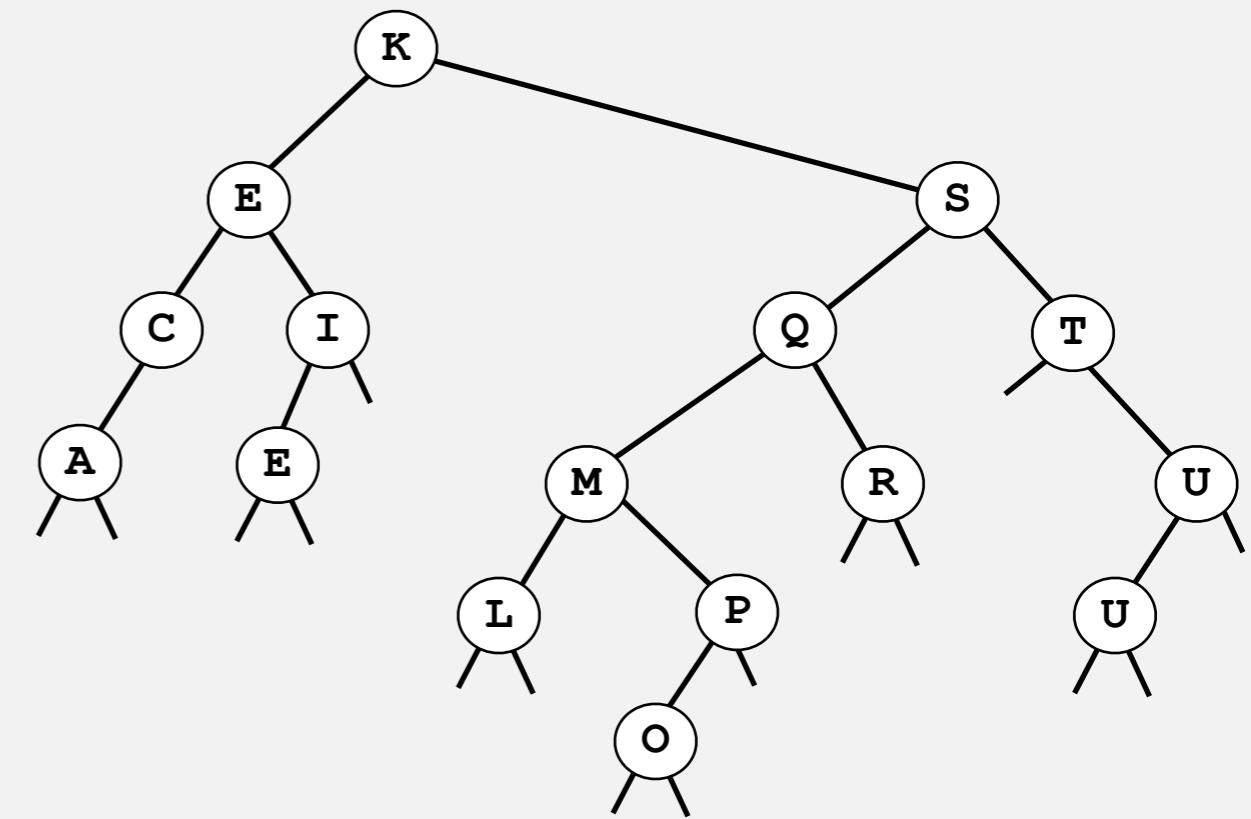
BST insertion: random order visualization

Ex. Insert keys in random order.



Correspondence between BSTs and quicksort partitioning

QUICKSORT EXAMPLE
E R A T E S L P U I M Q C X O K
E C A I E K L P U T M Q R X O S
A C E I E K L P U T M Q R X O S
A C E I E K L P U T M Q R X O S
A C E I E K L P U T M Q R X O S
A C E E I K L P O R M Q S X U T
A C E E I K L P O M Q R S X U T
A C E E I K L M O P Q R S X U T
A C E E I K L M O P Q R S X U T
A C E E I K L M O P Q R S X U T
A C E E I K L M O P Q R S X U T
A C E E I K L M O P Q R S X U T
A C E E I K L M O P Q R S X U T
A C E E I K L M O P Q R S X U T
A C E E I K L M O P Q R S X U T
A C E E I K L M O P Q R S X U T
A C E E I K L M O P Q R S T U X
A C E E I K L M O P Q R S T U X
A C E E I K L M O P Q R S T U X
A C E E I K L M O P Q R S T U X



Remark. Correspondence is 1-1 if array has no duplicate keys.

BSTs: mathematical analysis

Proposition. If keys are inserted in random order, the expected number of compares for a search/insert is $\sim 2 \ln N$.

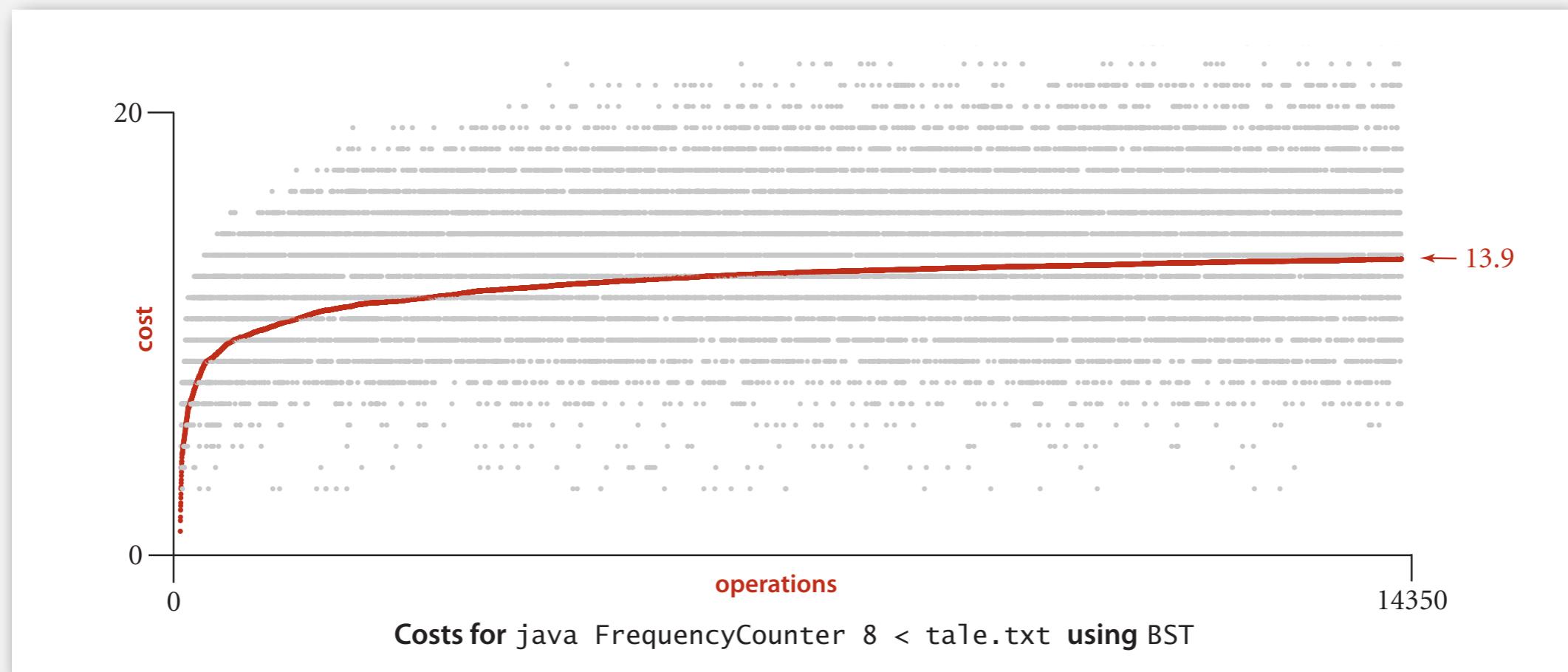
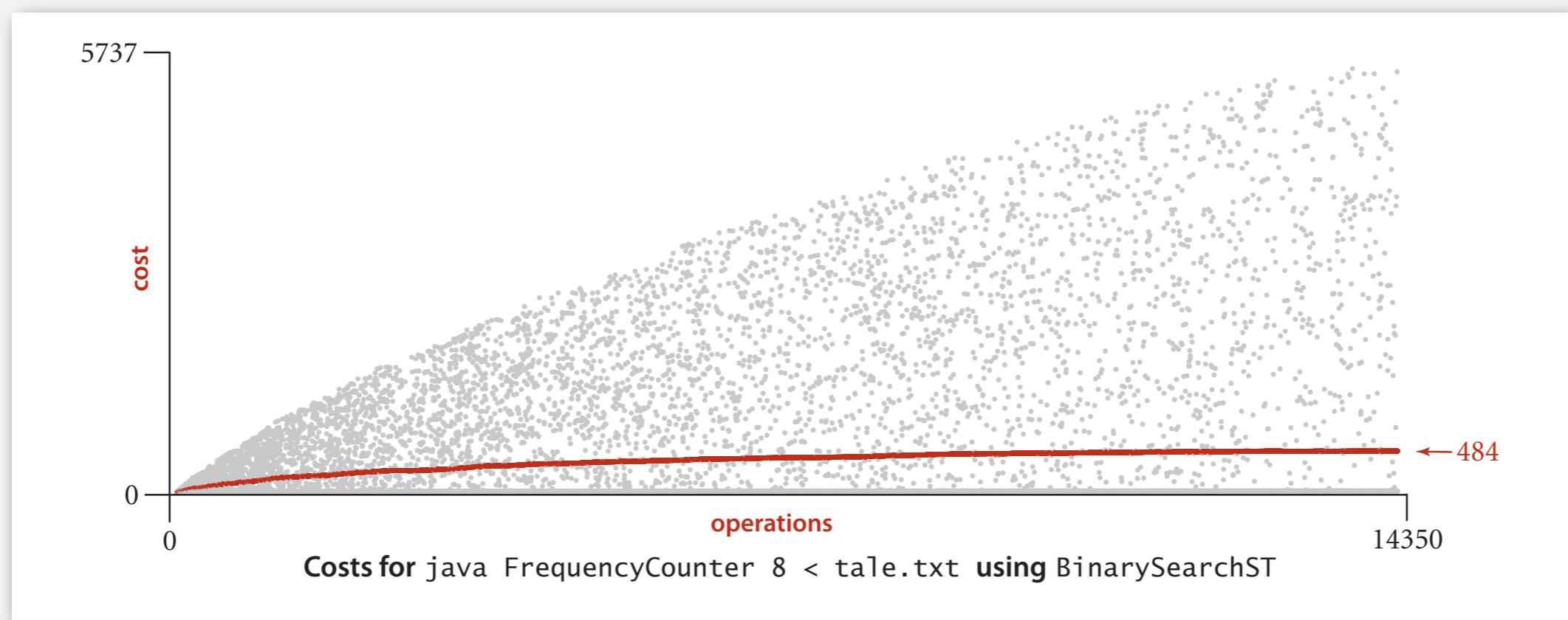
Pf. 1-1 correspondence with quicksort partitioning.

Proposition. [Reed, 2003] If keys are inserted in random order, expected height of tree is $\sim 4.311 \ln N$.

But... Worst-case height is N .

(exponentially small chance when keys are inserted in random order)

ST implementations: frequency counter



ST implementations: summary

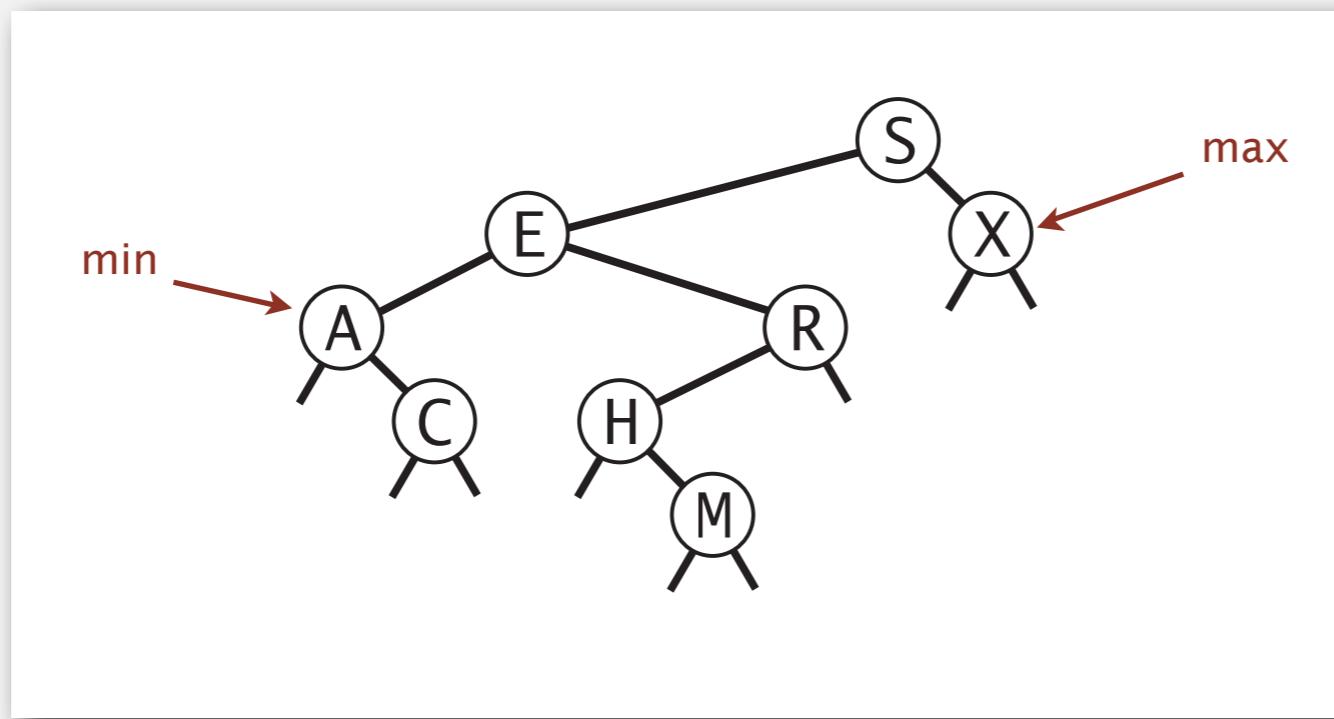
implementation	guarantee		average case		ordered ops?	operations on keys
	search	insert	search hit	insert		
sequential search (unordered list)	N	N	N/2	N	no	<code>equals()</code>
binary search (ordered array)	$\lg N$	N	$\lg N$	$N/2$	yes	<code>compareTo()</code>
BST	N	N	$1.39 \lg N$	$1.39 \lg N$?	<code>compareTo()</code>

- ▶ BSTs
- ▶ ordered operations
- ▶ deletion

Minimum and maximum

Minimum. Smallest key in table.

Maximum. Largest key in table.

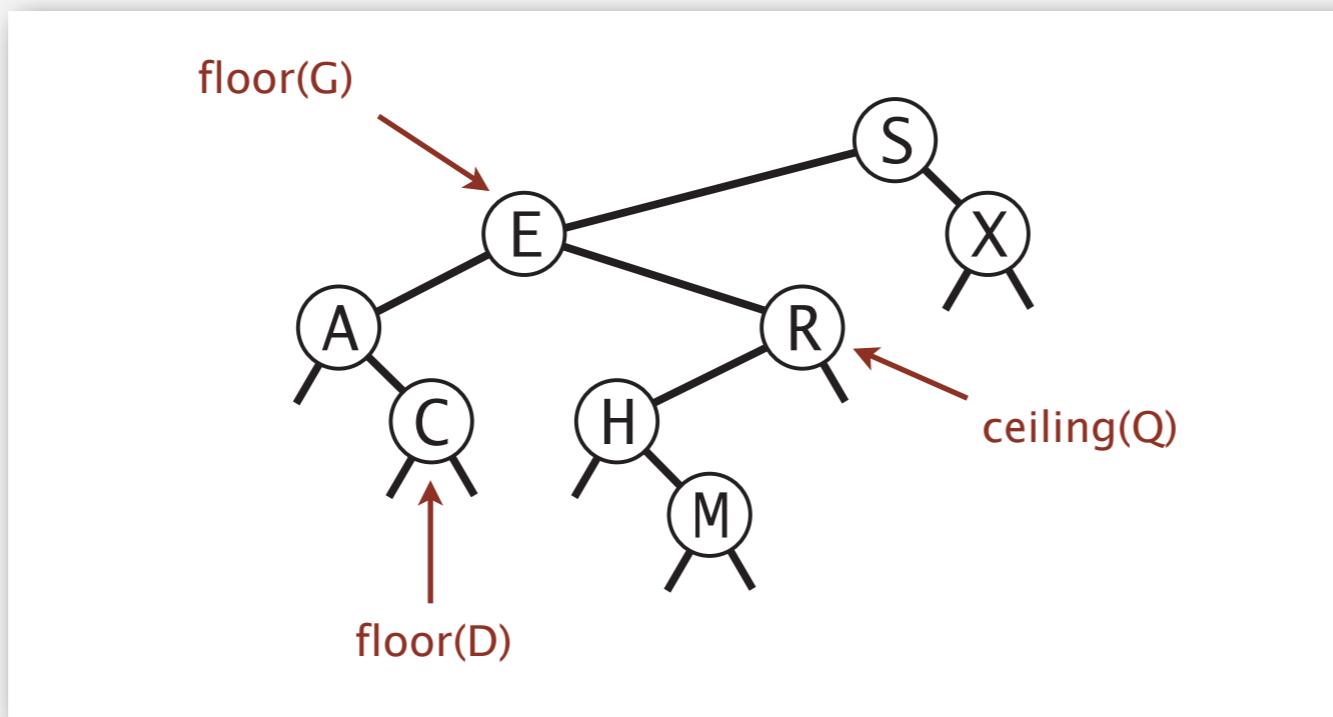


Q. How to find the min / max?

Floor and ceiling

Floor. Largest key \leq to a given key.

Ceiling. Smallest key \geq to a given key.



Q. How to find the floor /ceiling?

Computing the floor

Case 1. [k equals the key at root]

The floor of k is k .

Case 2. [k is less than the key at root]

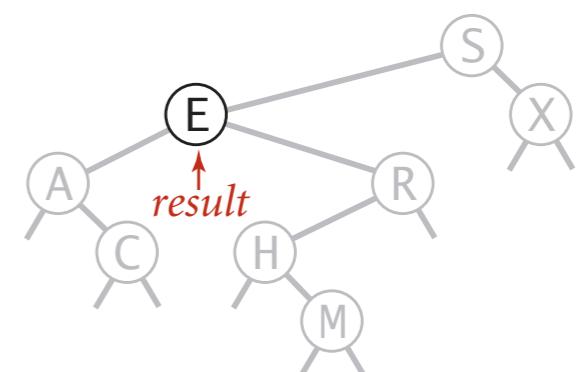
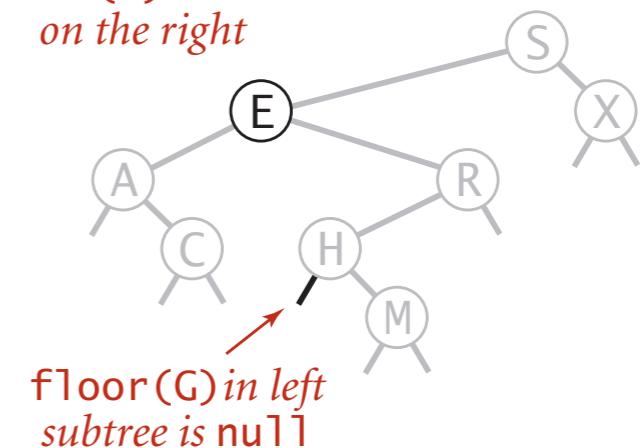
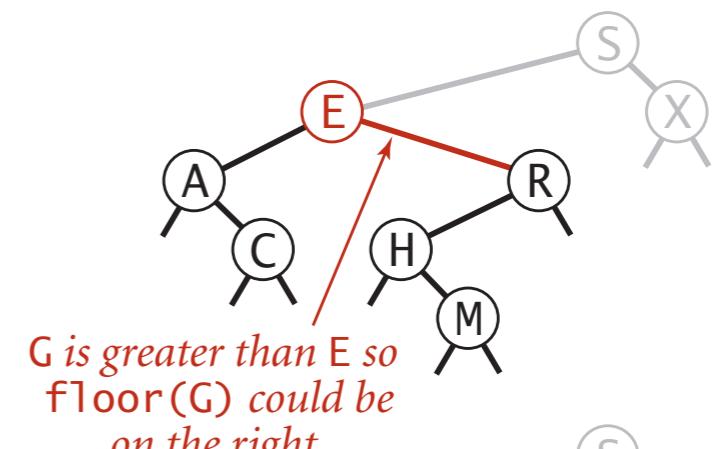
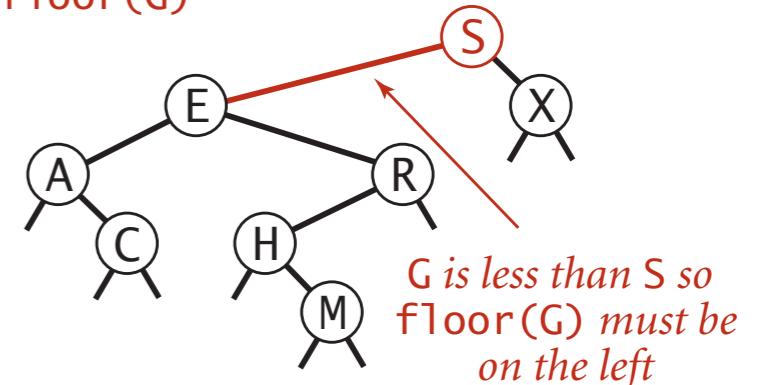
The floor of k is in the left subtree.

Case 3. [k is greater than the key at root]

The floor of k is in the right subtree

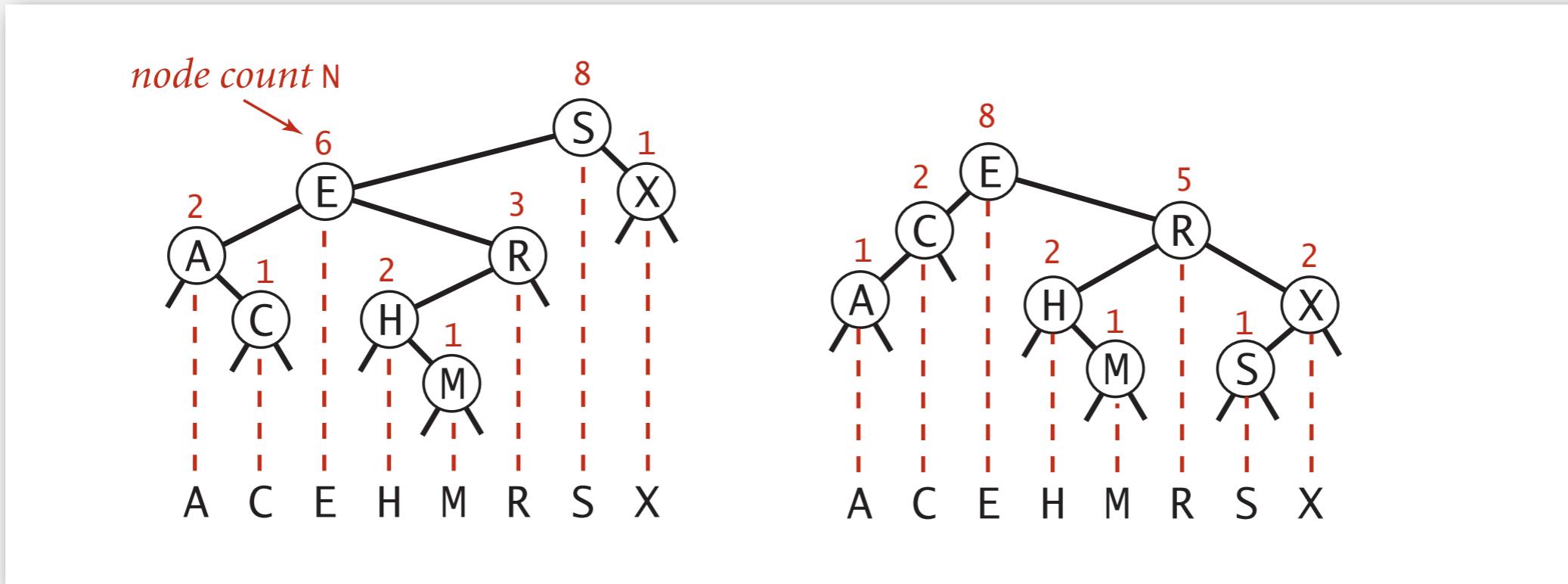
(if there is **any** key $\leq k$ in right subtree);
otherwise it is the key in the root.

finding $\text{floor}(G)$



Subtree counts

In each node, we store the number of nodes in the subtree rooted at that node.
To implement `size()`, return the count at the root.



Remark. This facilitates efficient implementation of `rank()` and `select()`.

BST implementation: subtree counts

```
private class Node  
{  
    private Key key;  
    private Value val;  
    private Node left;  
    private Node right;  
    private int N;  
}
```

number of nodes
in subtree

```
public int size()  
{    return size(root);    }  
  
private int size(Node x)  
{  
    if (x == null) return 0;  
    return x.N;  
}
```

ok to call when x is null

```
private Node put(Node x, Key key, Value val)  
{  
    if (x == null) return new Node(key, val);  
    int cmp = key.compareTo(x.key);  
    if (cmp < 0) x.left = put(x.left, key, val);  
    else if (cmp > 0) x.right = put(x.right, key, val);  
    else if (cmp == 0) x.val = val;  
    x.N = 1 + size(x.left) + size(x.right);  
    return x;  
}
```

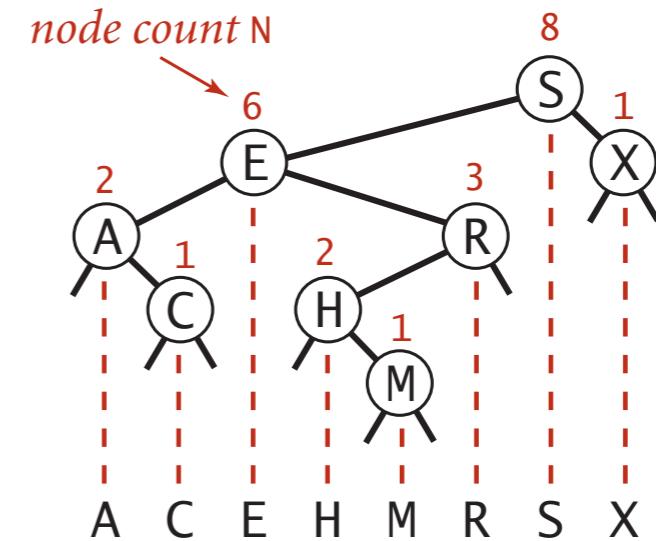
Rank

Rank. How many keys $< k$?

Return: rank of a given key

i. given key eq key at root:

return # of keys + in the left subtree



ii. given key is less than key at root:

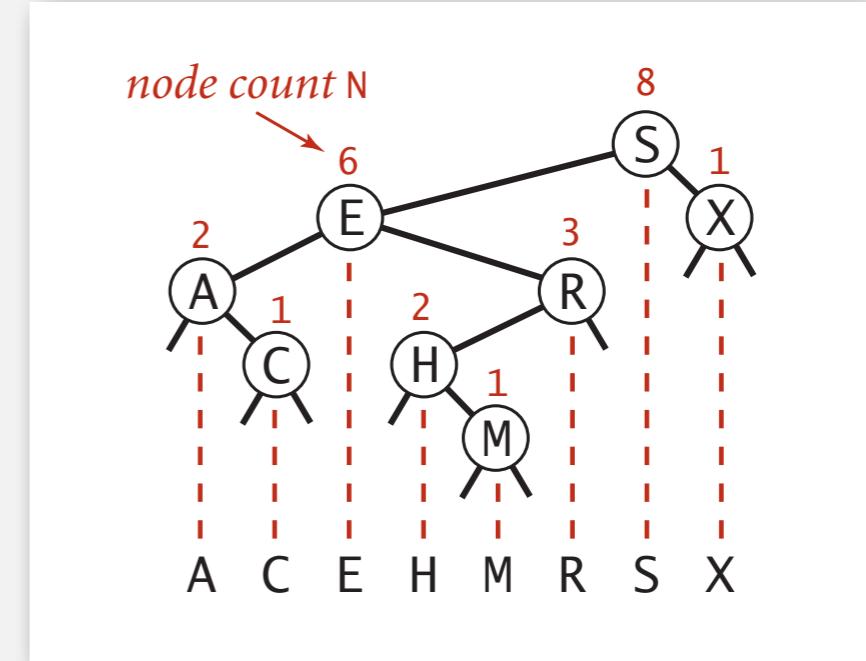
return rank() of key in the left subtree (recursively computed, that is)

iii. given key is larger than key at root:

return + plus one plus the rank of the key in the right tree (recursively computed, that is)

Rank

Rank. How many keys $< k$?



```
public int rank(Key key)
{   return rank(key, root); }

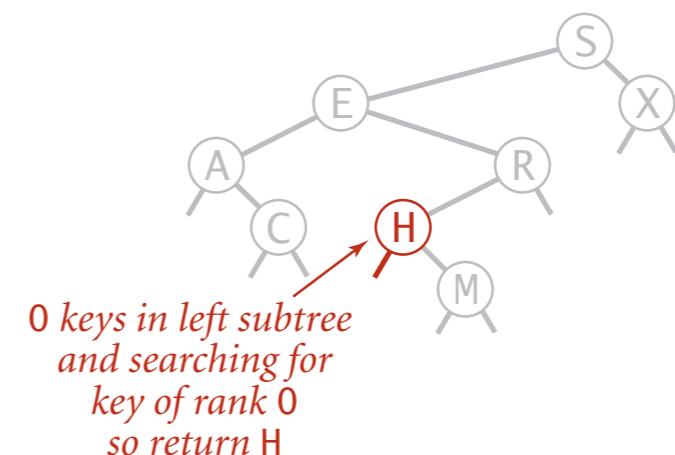
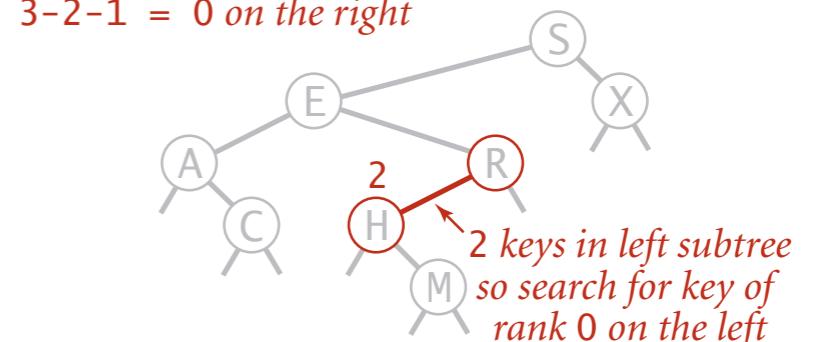
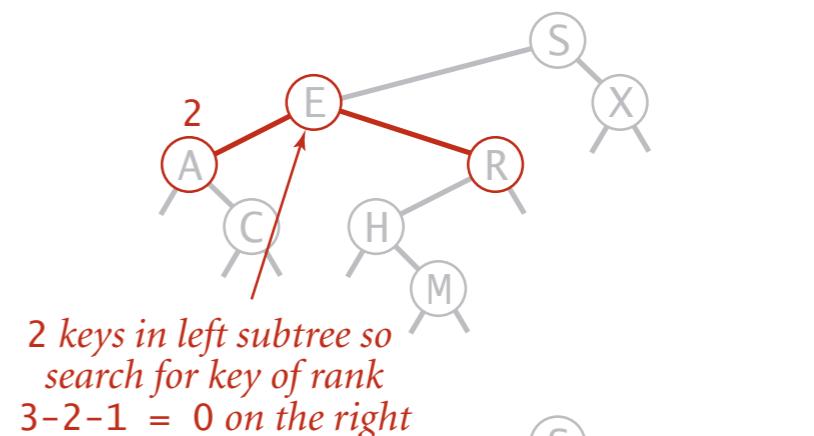
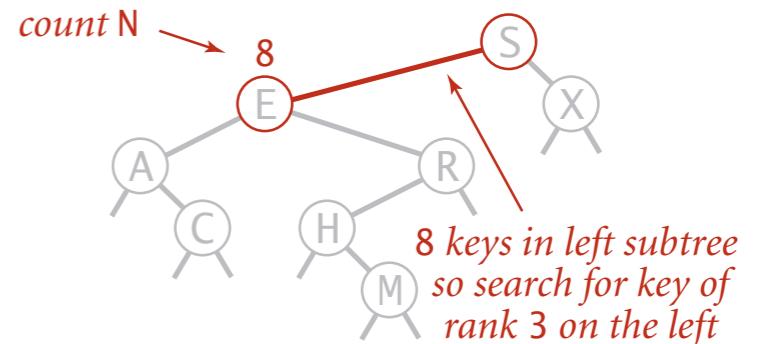
private int rank(Key key, Node x)
{
    if (x == null) return 0;
    int cmp = key.compareTo(x.key);
    if (cmp < 0) return rank(key, x.left);
    else if (cmp > 0) return 1 + size(x.left) + rank(key, x.right);
    else if (cmp == 0) return size(x.left);
}
```

Selection

Select. Key of given rank (the key such that precisely k other keys in the BST are smaller)

- i. if # of keys (t) in left subtree is $> k$
look recursively for key of rank k in left subt
- ii. if $t == k$
return the key of the root
- iii. if $t < k$
then look recursively for key of rank $(k - t - 1)$ in right sub-tree

finding select(3)
the key of rank 3

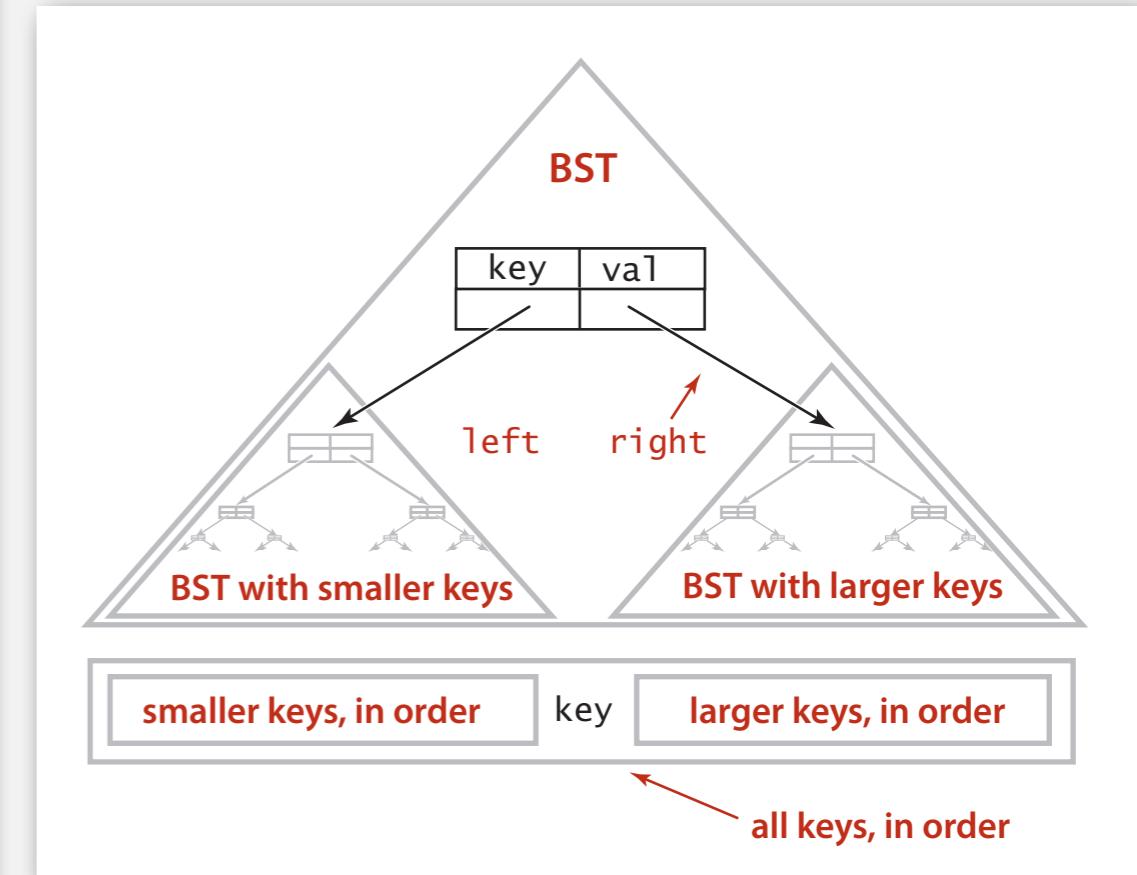


Inorder traversal

- Traverse left subtree.
- Enqueue key.
- Traverse right subtree.

```
public Iterable<Key> keys()
{
    Queue<Key> q = new Queue<Key>();
    inorder(root, q);
    return q;
}

private void inorder(Node x, Queue<Key> q)
{
    if (x == null) return;
    inorder(x.left, q);
    q.enqueue(x.key);
    inorder(x.right, q);
}
```



Property. Inorder traversal of a BST yields keys in ascending order.

Inorder traversal

- Traverse left subtree.
- Enqueue key.
- Traverse right subtree.

```
inorder(S)
    inorder(E)
        inorder(A)
        enqueue A
    inorder(C)
        enqueue C
    enqueue E
inorder(R)
    inorder(H)
        enqueue H
    inorder(M)
        enqueue M
    enqueue R
enqueue S
inorder(X)
    enqueue X
```

A
C
E

H
M
R
S

X

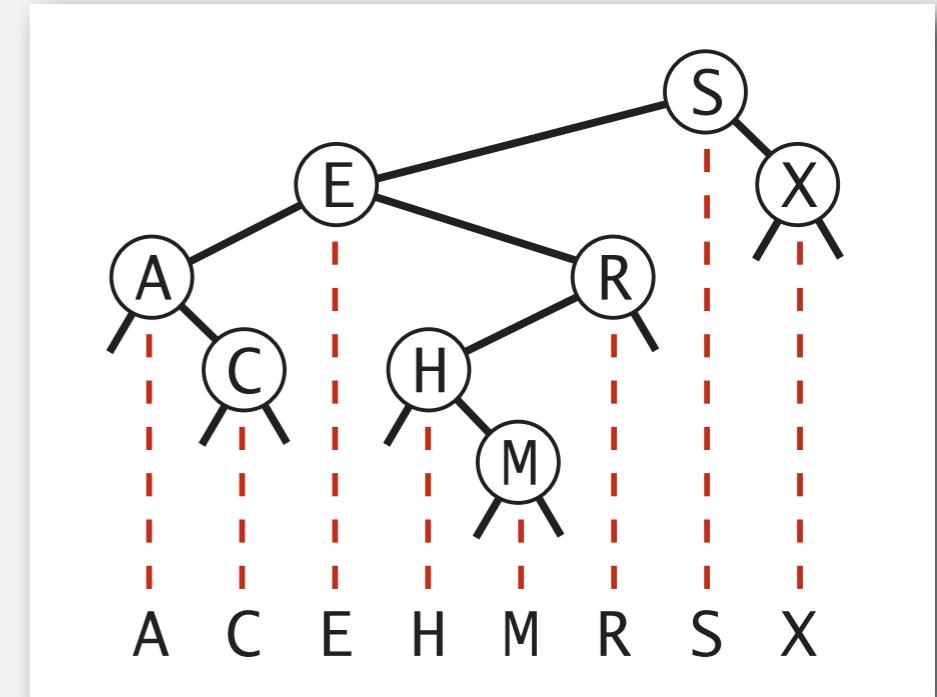
S
S E
S E A

S E A C

S E R
S E R H

S E R H M

S X



recursive calls

queue

function call stack

BST: ordered symbol table operations summary

	sequential search	binary search	BST
search	N	$\lg N$	h
insert	1	N	h
min / max	N	1	h
floor / ceiling	N	$\lg N$	h
rank	N	$\lg N$	h
select	N	1	h
ordered iteration	$N \log N$	N	N

h = height of BST
(proportional to $\log N$
if keys inserted in random order)

order of growth of running time of ordered symbol table operations

- ▶ BSTs
- ▶ ordered operations
- ▶ deletion

ST implementations: summary

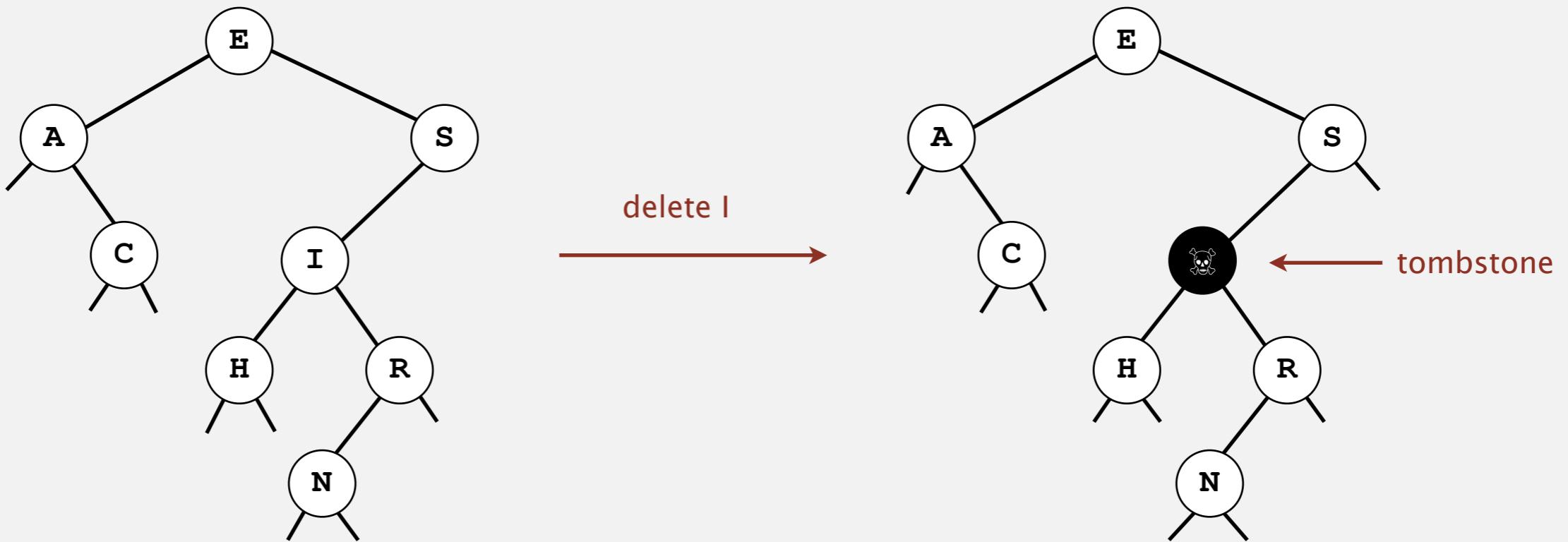
implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	N/2	N/2	yes	<code>compareTo()</code>
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$???	yes	<code>compareTo()</code>

Next. Deletion in BSTs.

BST deletion: lazy approach

To remove a node with a given key:

- Set its value to `null`.
- Leave key in tree to guide searches (but don't consider it equal to search key).



Cost. $\sim 2 \ln N'$ per insert, search, and delete (if keys in random order), where N' is the number of key-value pairs ever inserted in the BST.

Unsatisfactory solution. Tombstone overload.

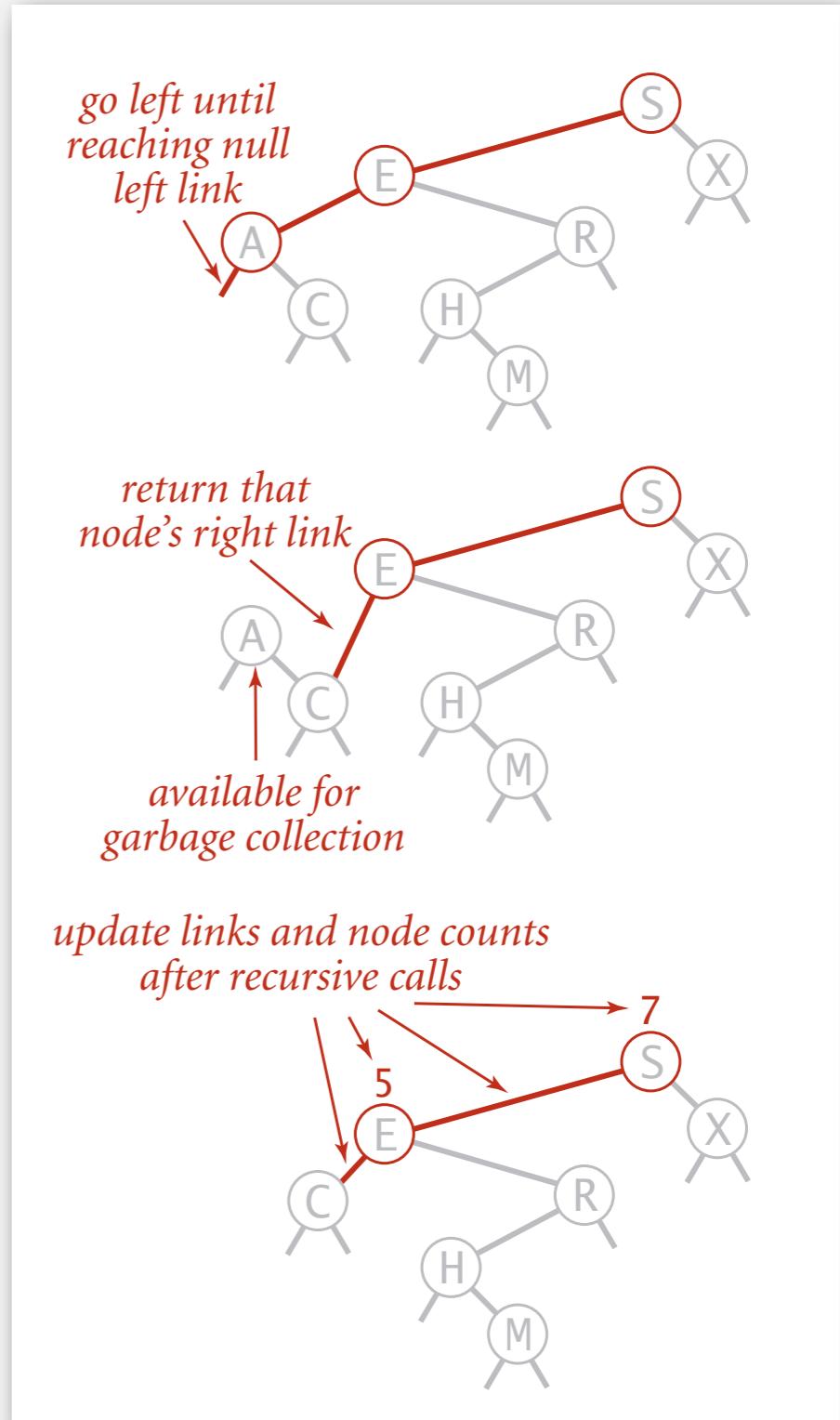
Deleting the minimum

To delete the minimum key:

- Go left until finding a node with a null left link.
- Replace that node by its right link.
- Update subtree counts.

```
public void deleteMin()
{   root = deleteMin(root);   }

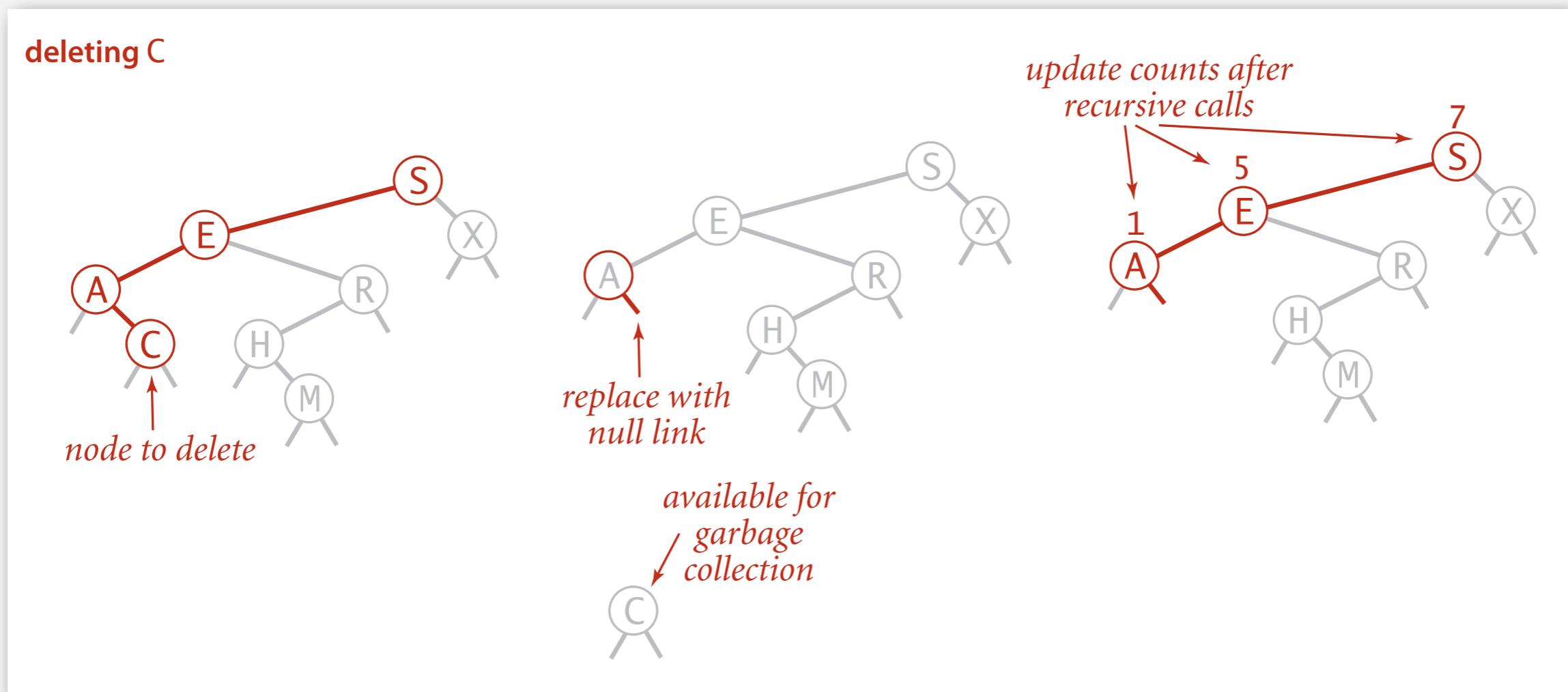
private Node deleteMin(Node x)
{
    if (x.left == null) return x.right;
    x.left = deleteMin(x.left);
    x.N = 1 + size(x.left) + size(x.right);
    return x;
}
```



Hibbard deletion

To delete a node with key k : search for node t containing key k .

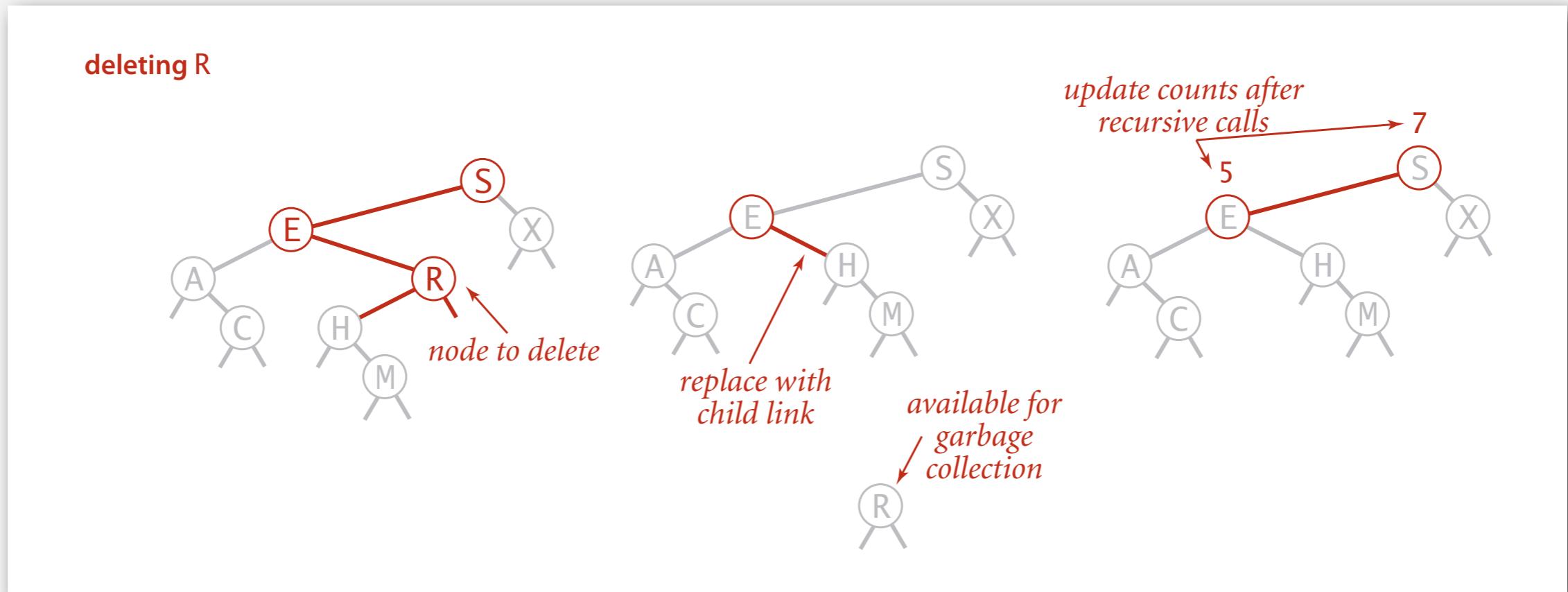
Case 0. [0 children] Delete t by setting parent link to null.



Hibbard deletion

To delete a node with key k : search for node t containing key k .

Case 1. [1 child] Delete t by replacing parent link.



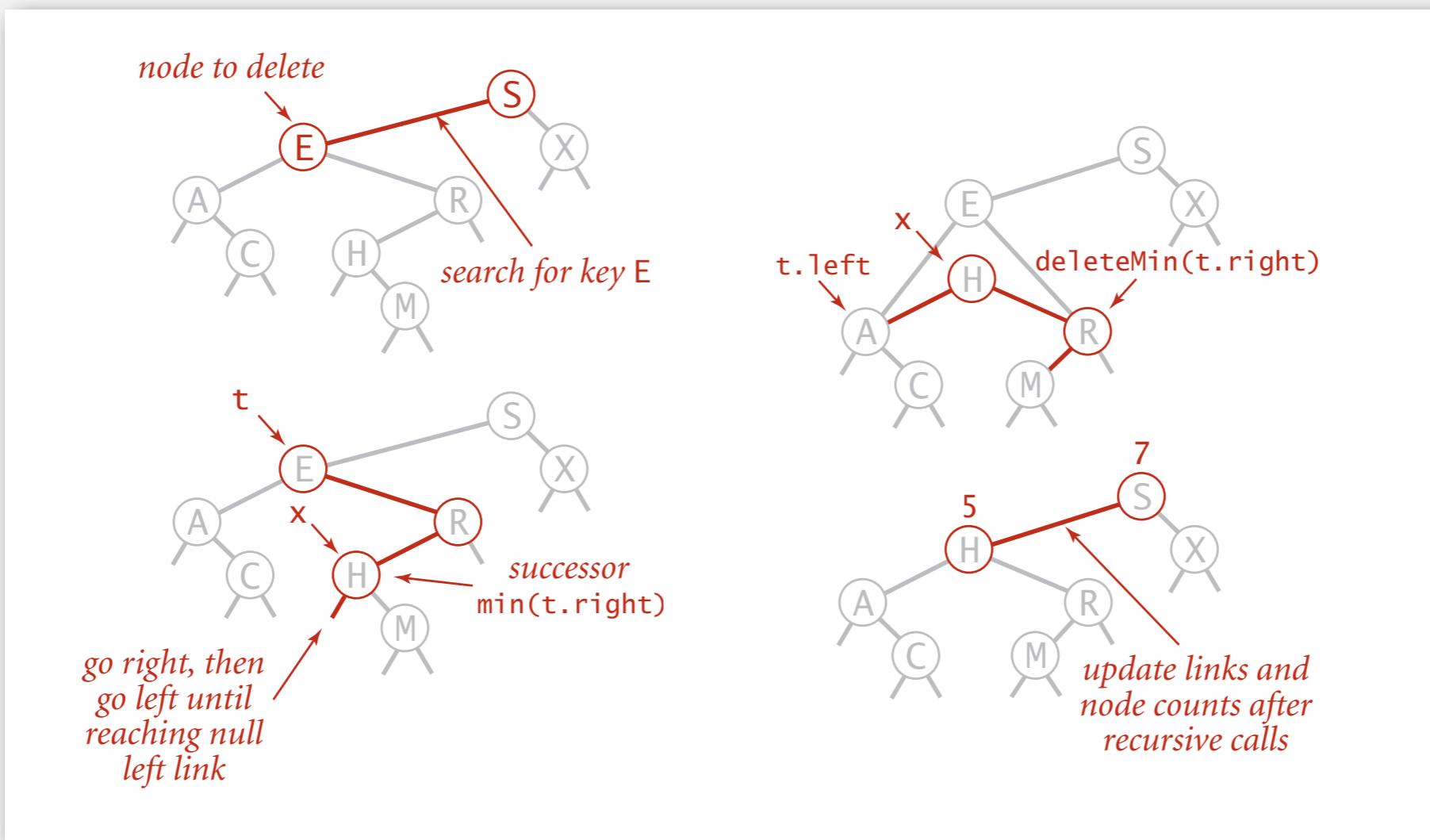
Hibbard deletion

To delete a node with key k : search for node t containing key k .

Case 2. [2 children]

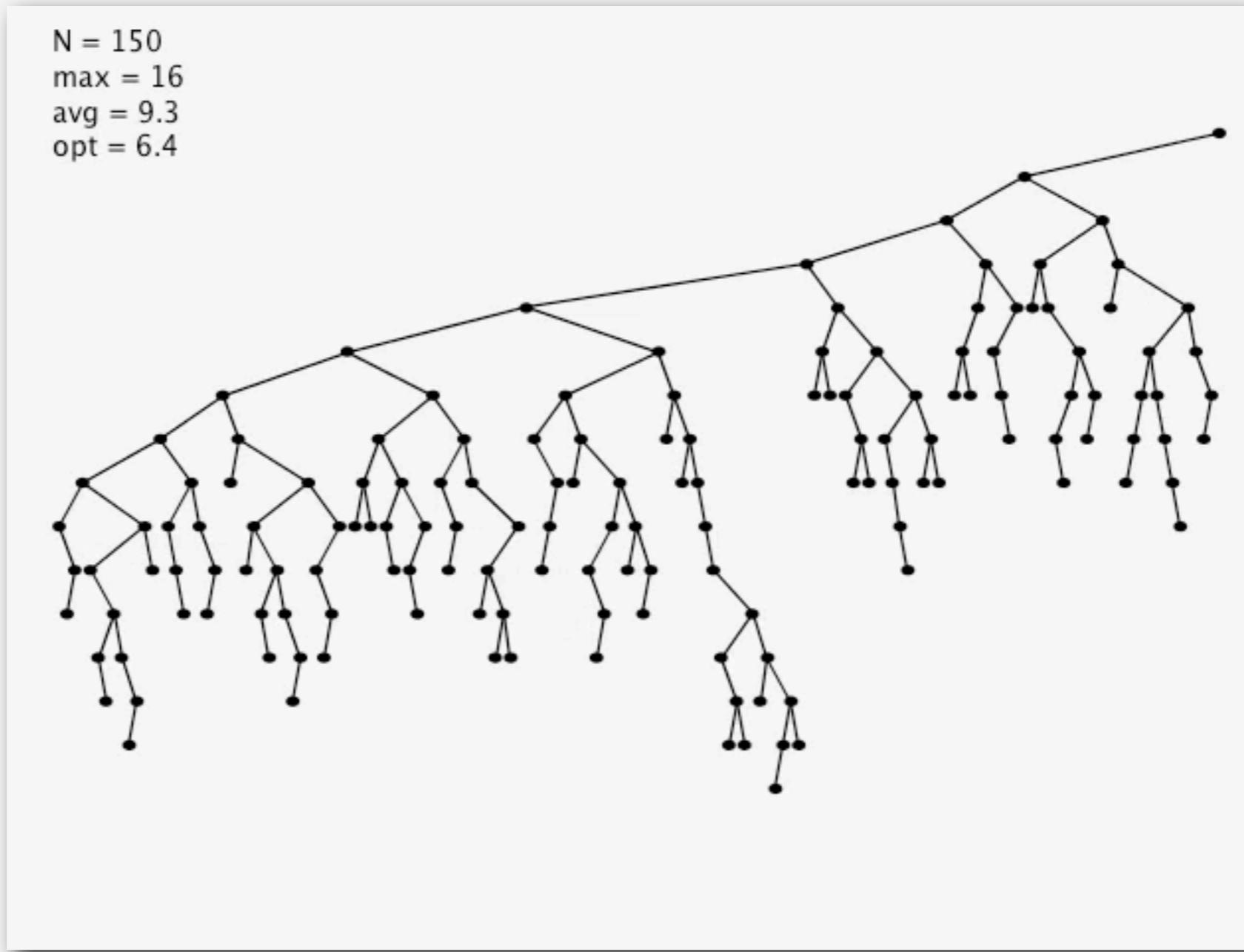
- Find successor x of t .
- Delete the minimum in t 's right subtree.
- Put x in t 's spot.

← x has no left child
← but don't garbage collect x
← still a BST



Hibbard deletion: analysis

Unsatisfactory solution. Not symmetric.



Surprising consequence. Trees not random (!) $\Rightarrow \sqrt{N}$ per op.

Longstanding open problem. Simple and efficient delete for BSTs.

ST implementations: summary

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	N/2	N/2	yes	<code>compareTo()</code>
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	\sqrt{N}	yes	<code>compareTo()</code>

other operations also become \sqrt{N}
if deletions allowed

Red-black BST. *Guarantee logarithmic performance for all operations.*

BALANCED SEARCH TREES

- ▶ 2-3 search trees
- ▶ red-black BSTs

Symbol table review

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	N/2	N/2	yes	<code>compareTo()</code>
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$?	yes	<code>compareTo()</code>
goal	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	yes	<code>compareTo()</code>

Challenge. Guarantee performance.

This lecture. 2-3 trees, left-leaning red-black BSTs, B-trees.



new concept (2007/8)

- 2-3 search trees
- red-black BSTs

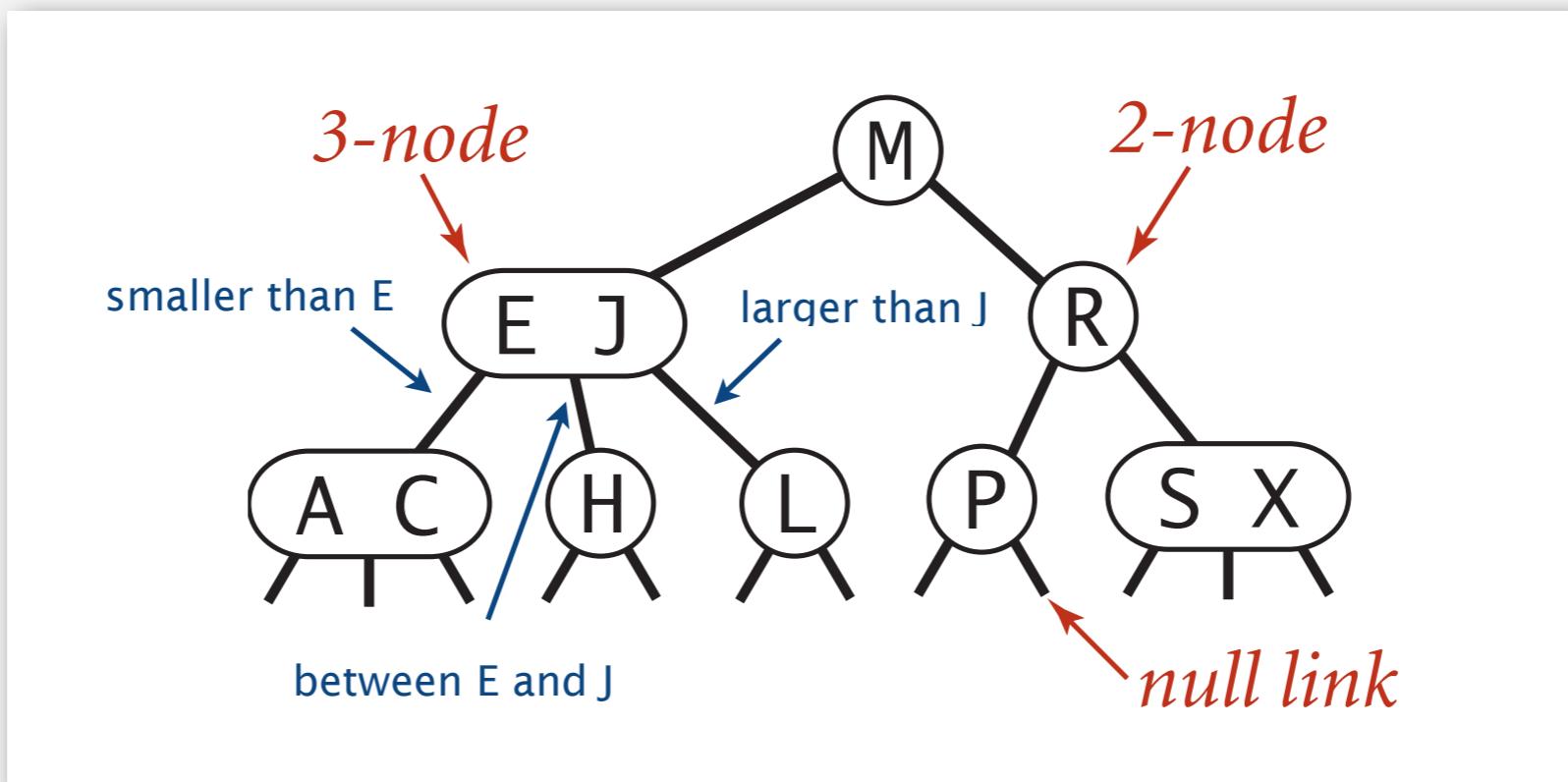
2-3 tree

Allow 1 or 2 keys per node.

- 2-node: one key, two children.
- 3-node: two keys, three children.

Symmetric order. Inorder traversal yields keys in ascending order.

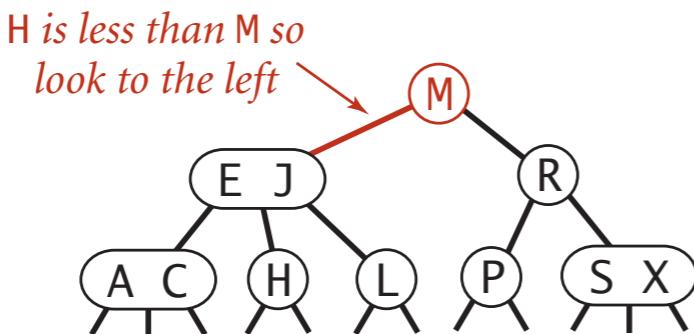
Perfect balance. Every path from root to null link has same length.



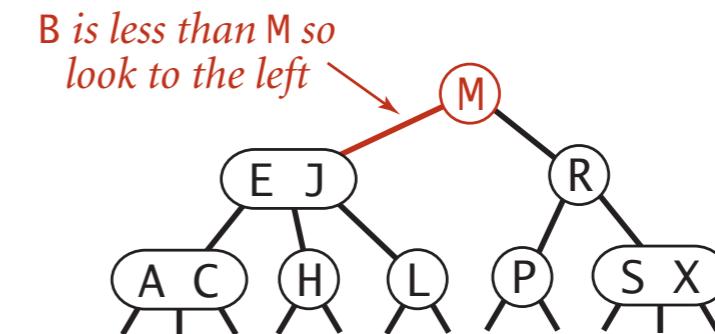
Search in a 2-3 tree

- Compare search key against keys in node.
- Find interval containing search key.
- Follow associated link (recursively).

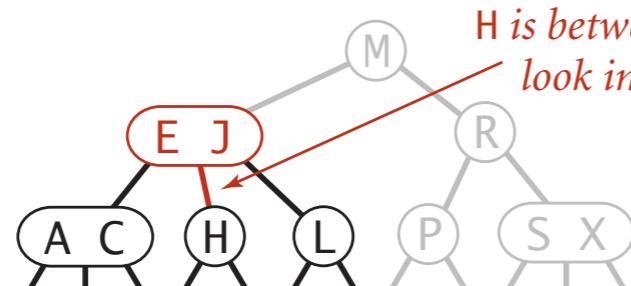
successful search for H



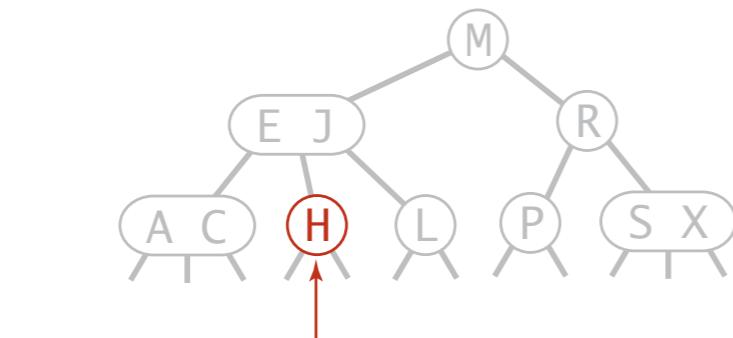
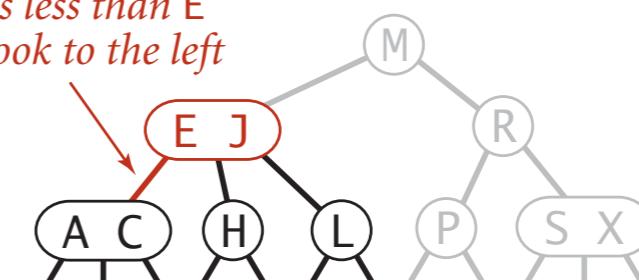
unsuccessful search for B



H is between E and L so
look in the middle



B is less than E
so look to the left

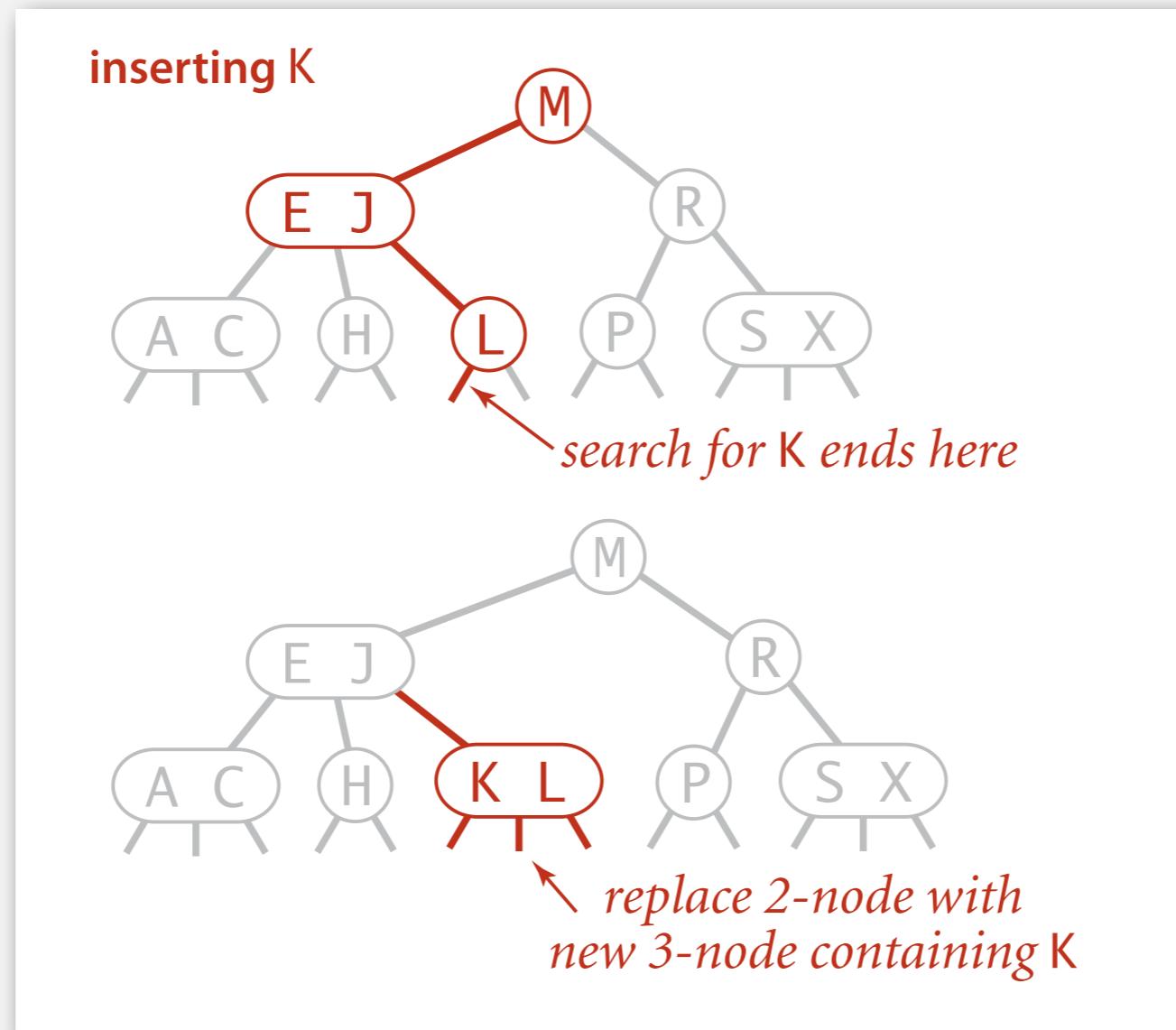


found H so return value (search hit)
B is between A and C so look in the middle
link is null so B is not in the tree (search miss)

Insertion in a 2-3 tree

Case 1. Insert into a 2-node at bottom.

- Search for key, as usual.
- Replace 2-node with 3-node.

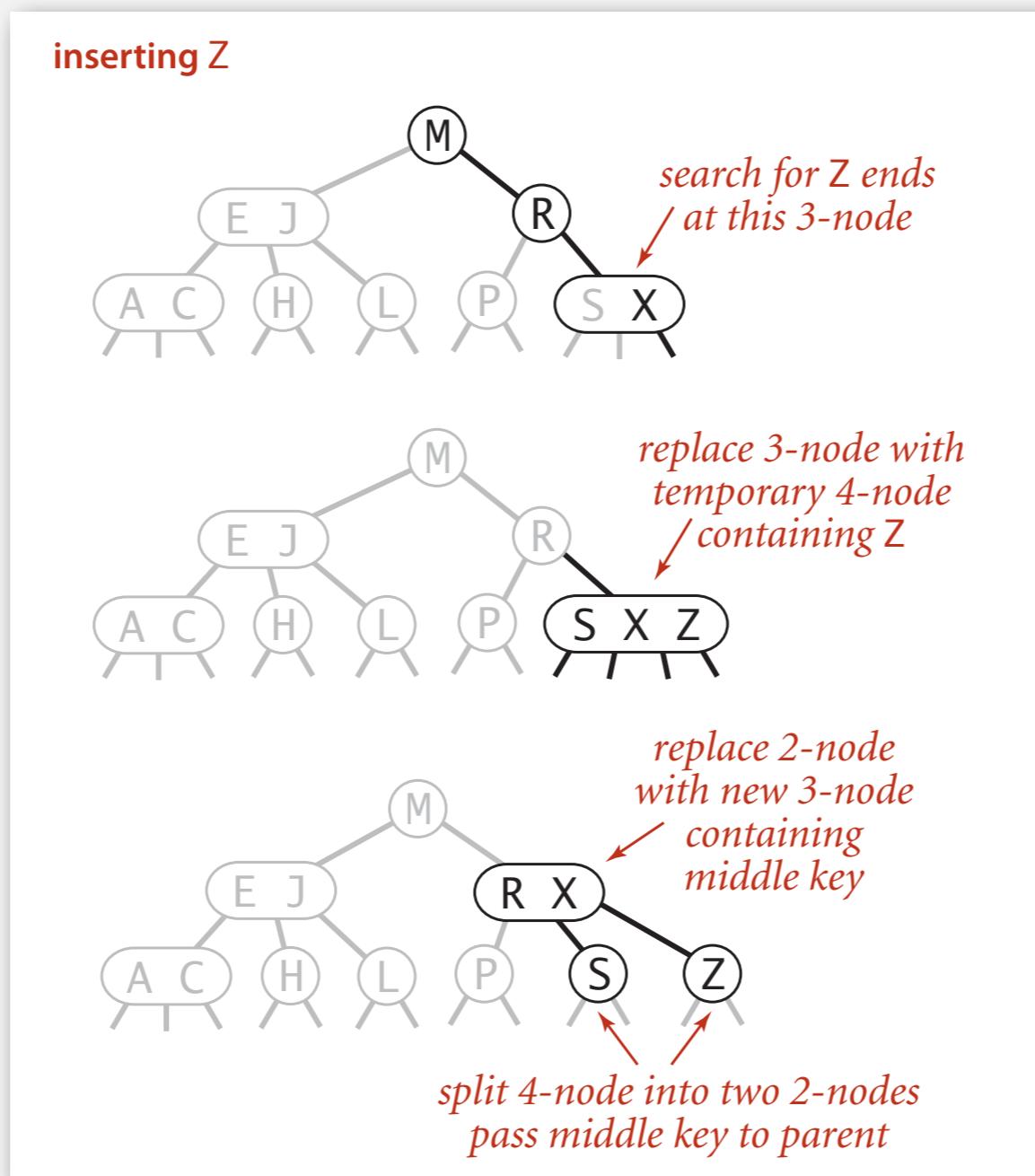


Insertion in a 2-3 tree

Case 2. Insert into a 3-node at bottom (or whose parent is a 2-node)

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.

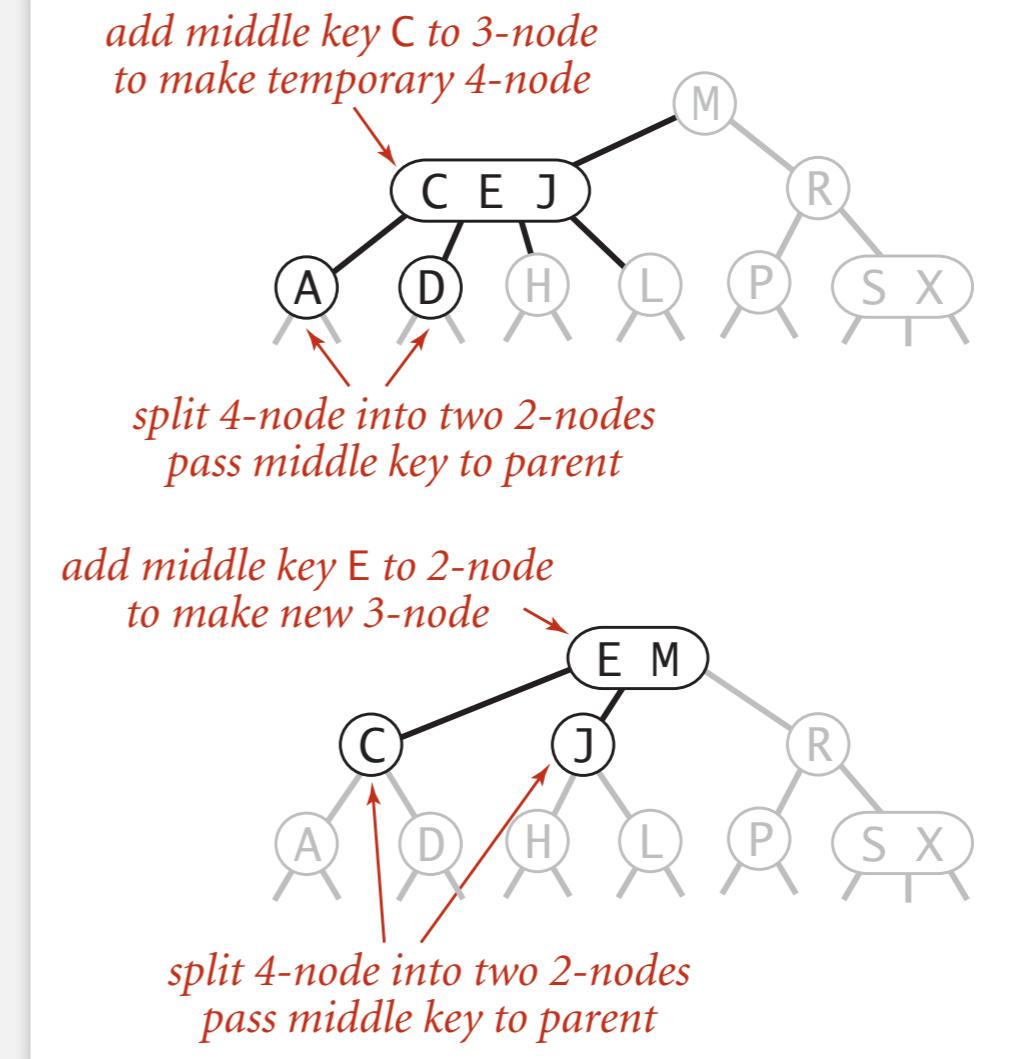
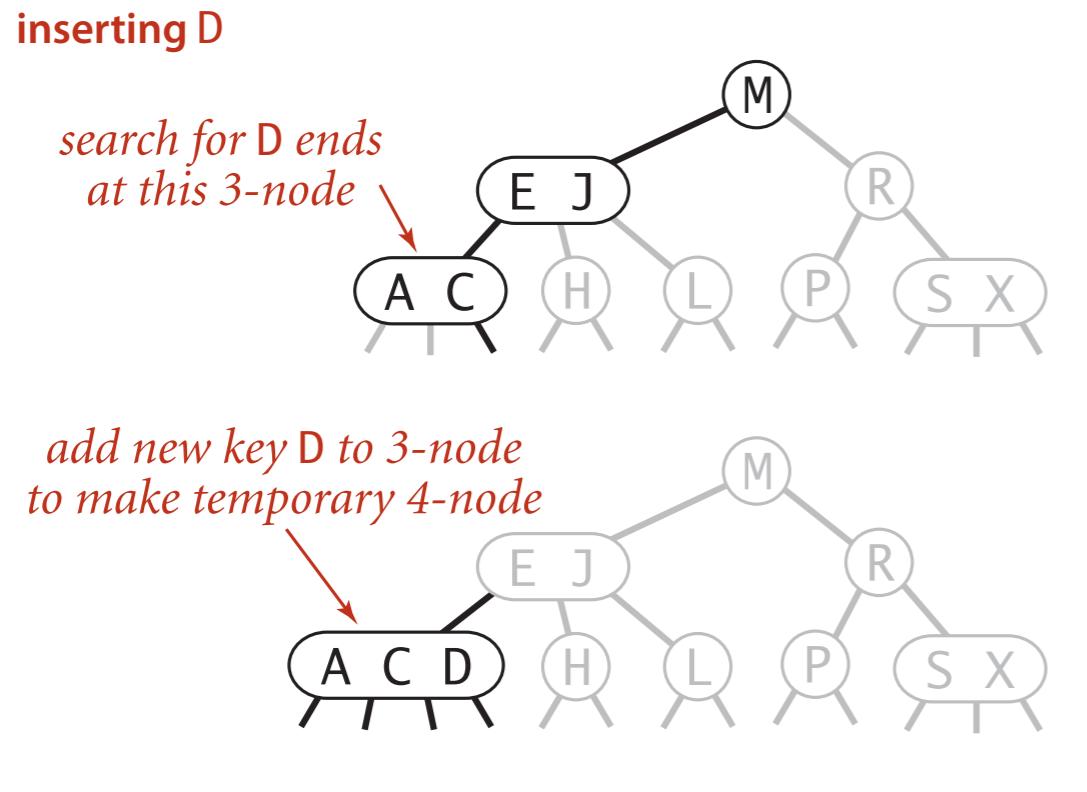
why middle key?



Insertion in a 2-3 tree

Case 2. Insert into a 3-node at bottom (or whose parent is 3-node)

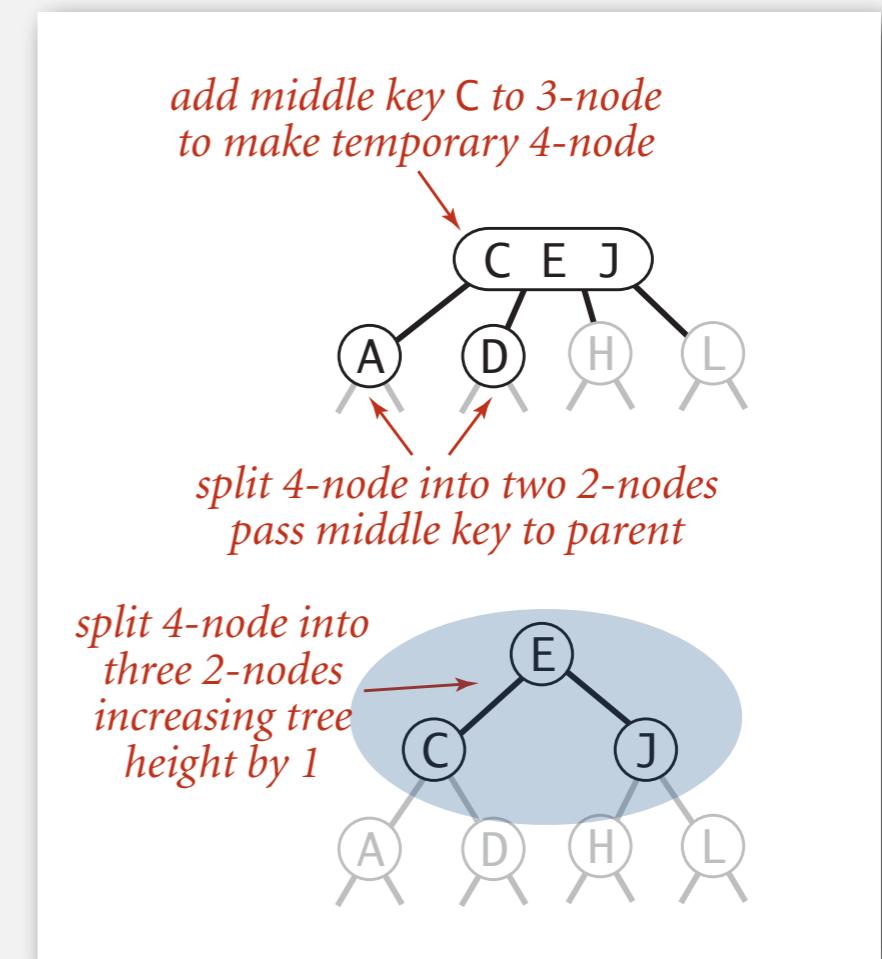
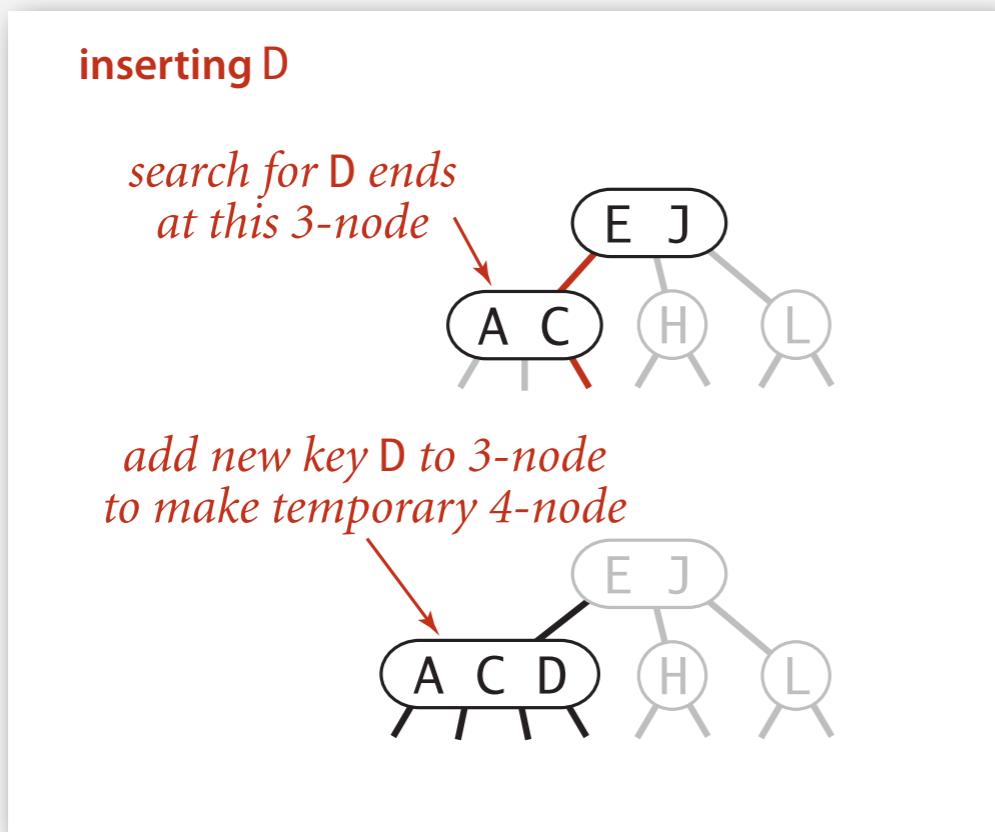
- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
- Repeat up the tree, as necessary.



Insertion in a 2-3 tree

Case 2. Insert into a 3-node at bottom.

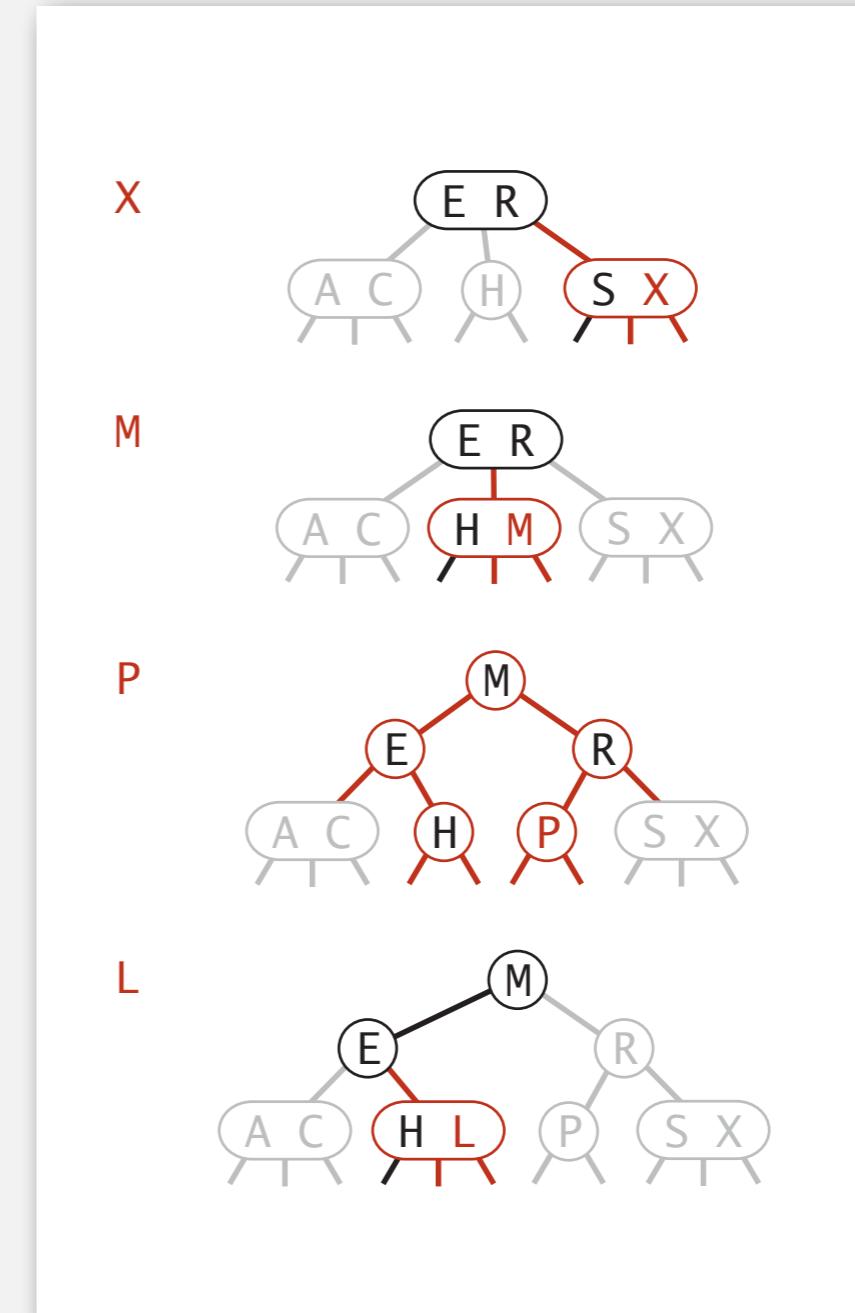
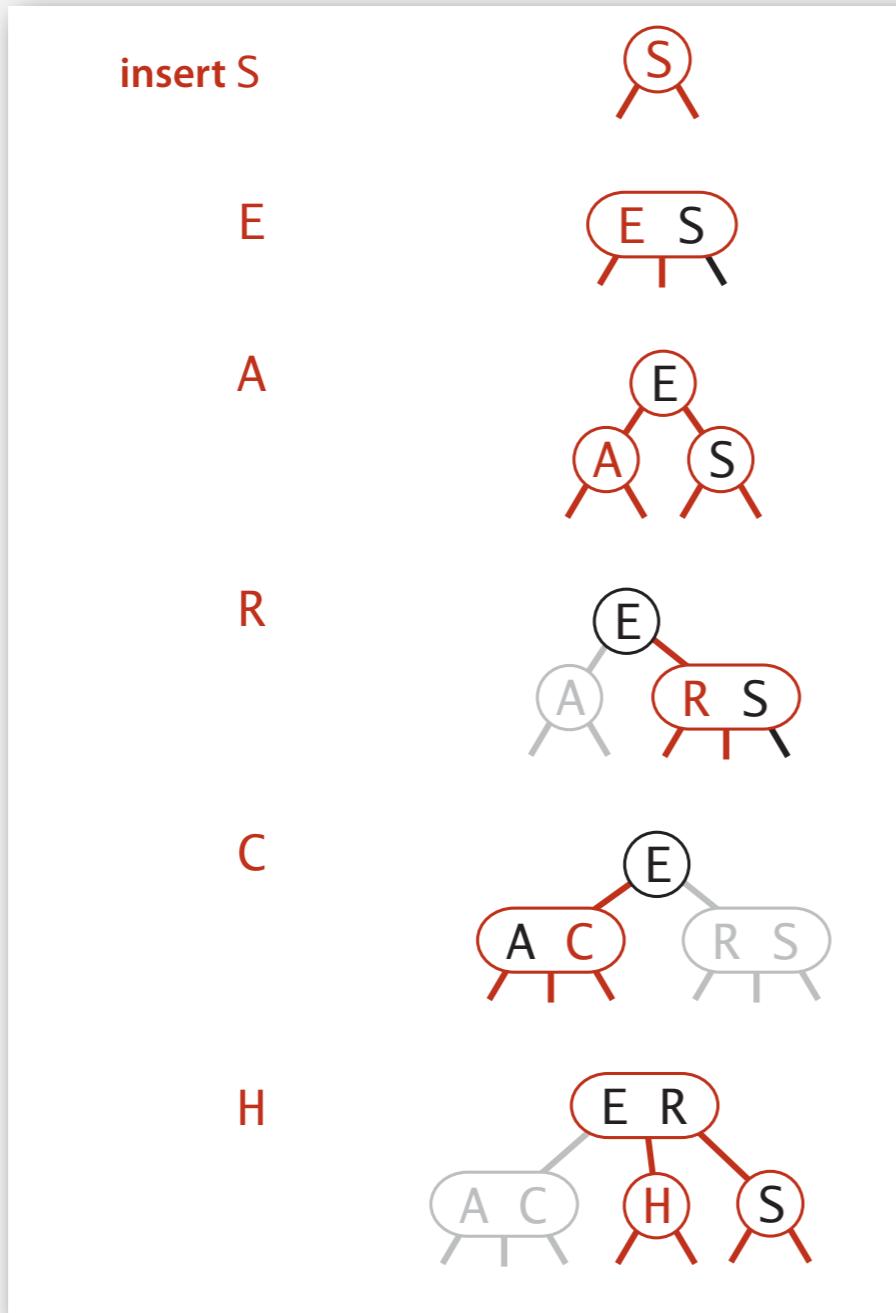
- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
- Repeat up the tree, as necessary.
- If you reach the root and it's a 4-node, split it into three 2-nodes.



Remark. Splitting the root increases height by 1.

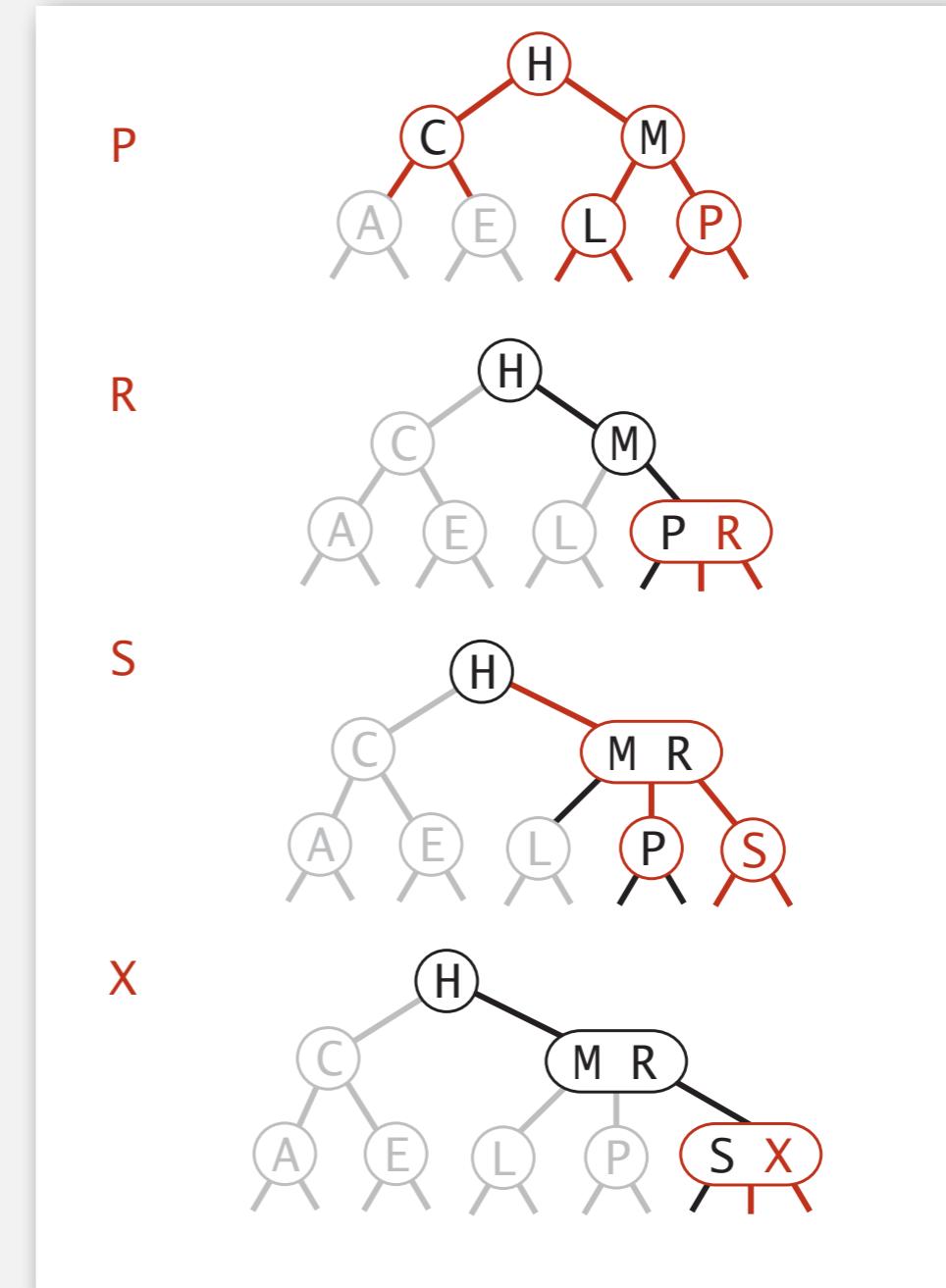
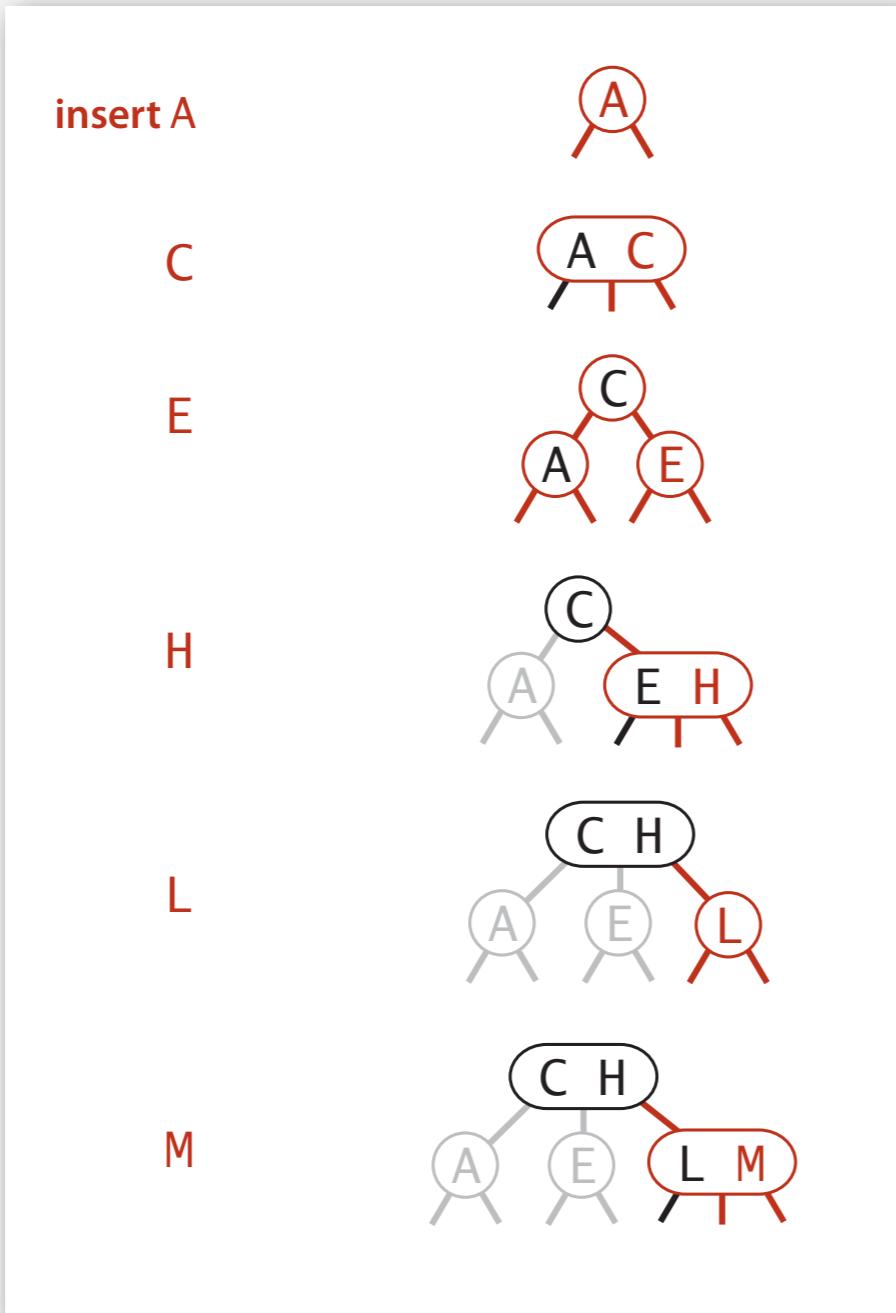
2-3 tree construction trace

Standard indexing client.



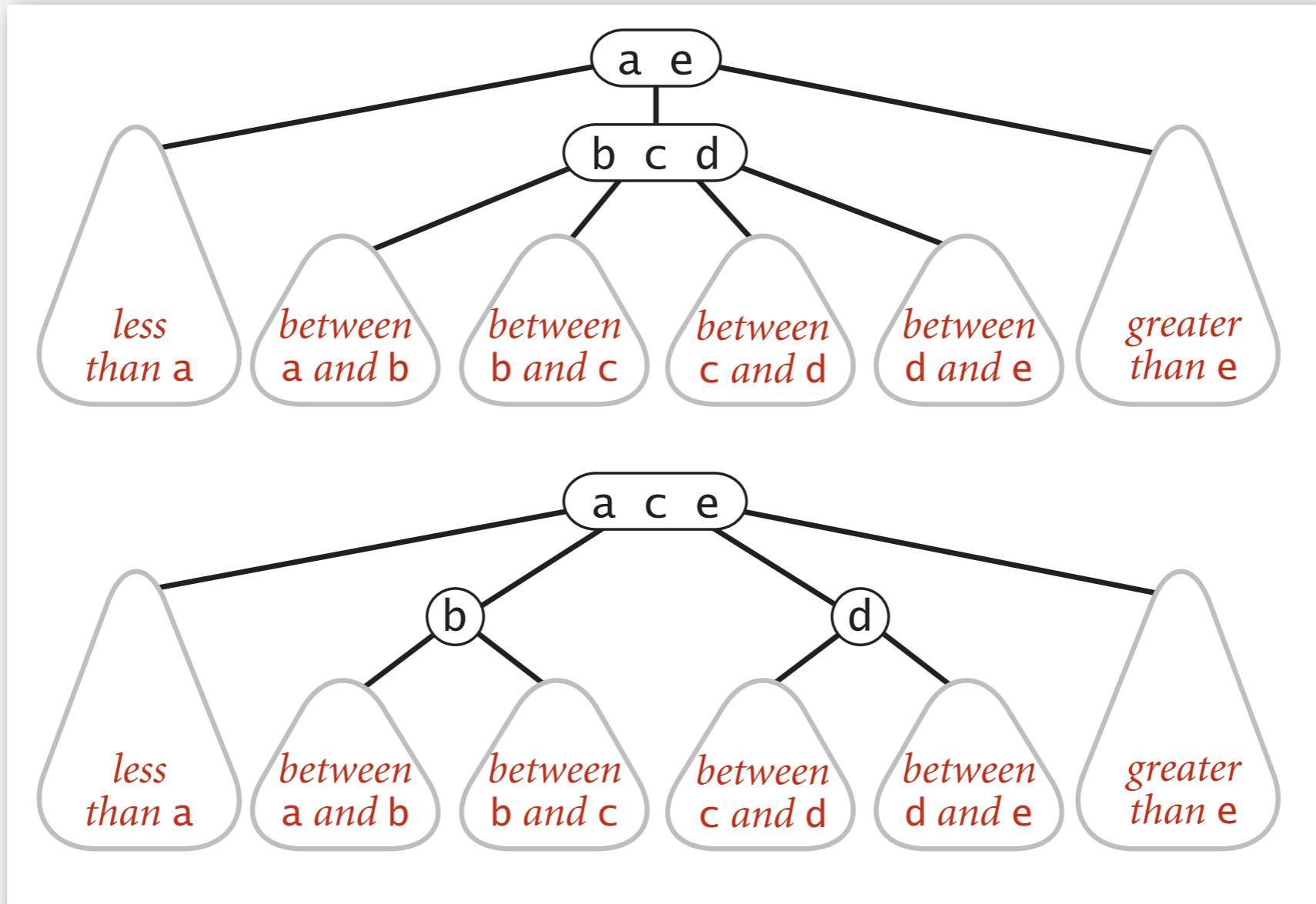
2-3 tree construction trace

The same keys inserted in ascending order.



Local transformations in a 2-3 tree

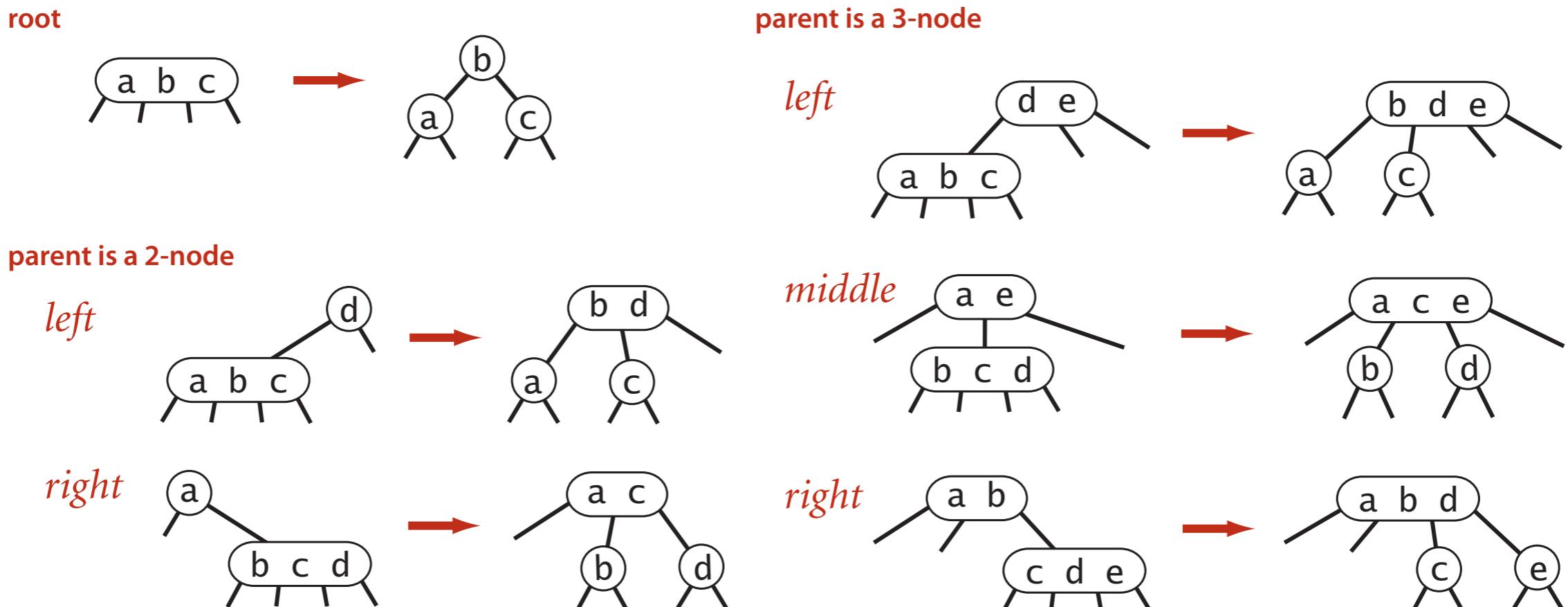
Splitting a 4-node is a **local transformation**: constant number of operations.



Global properties in a 2-3 tree

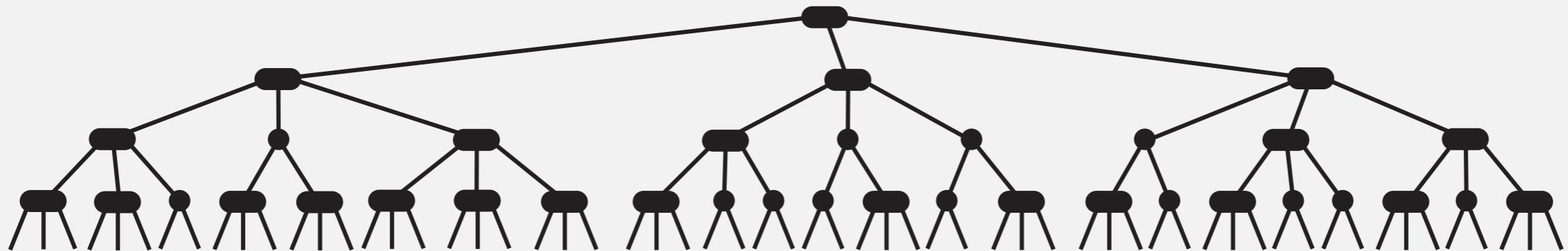
Invariants. Maintains symmetric order and perfect balance.

Pf. Each transformation maintains symmetric order and perfect balance.



2-3 tree: performance

Perfect balance. Every path from root to null link has same length.

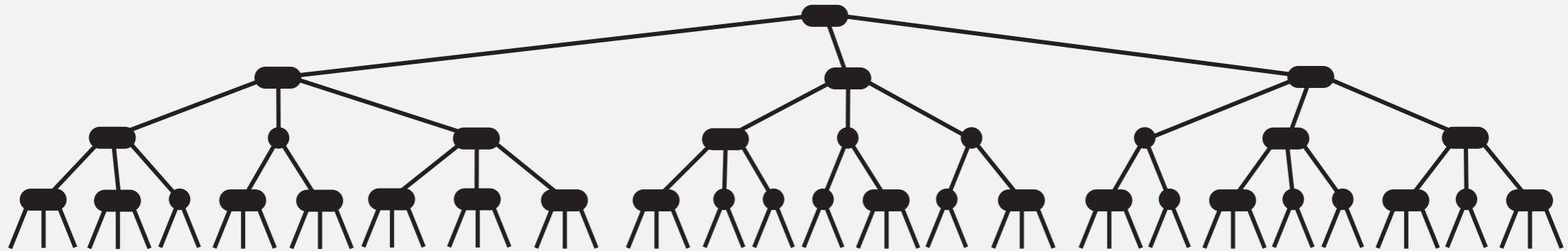


Tree height.

- Worst case:
- Best case:

2-3 tree: performance

Perfect balance. Every path from root to null link has same length.



Tree height.

- Worst case: $\lg N$. [all 2-nodes]
- Best case: $\log_3 N \approx .631 \lg N$. [all 3-nodes]
- Between 12 and 20 for a million nodes.
- Between 18 and 30 for a billion nodes.

Guaranteed logarithmic performance for search and insert.

ST implementations: summary

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	N/2	N/2	yes	<code>compareTo()</code>
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$?	yes	<code>compareTo()</code>
2-3 tree	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	yes	<code>compareTo()</code>



 constants depend upon
 implementation

2-3 tree: implementation?

Direct implementation is complicated, because:

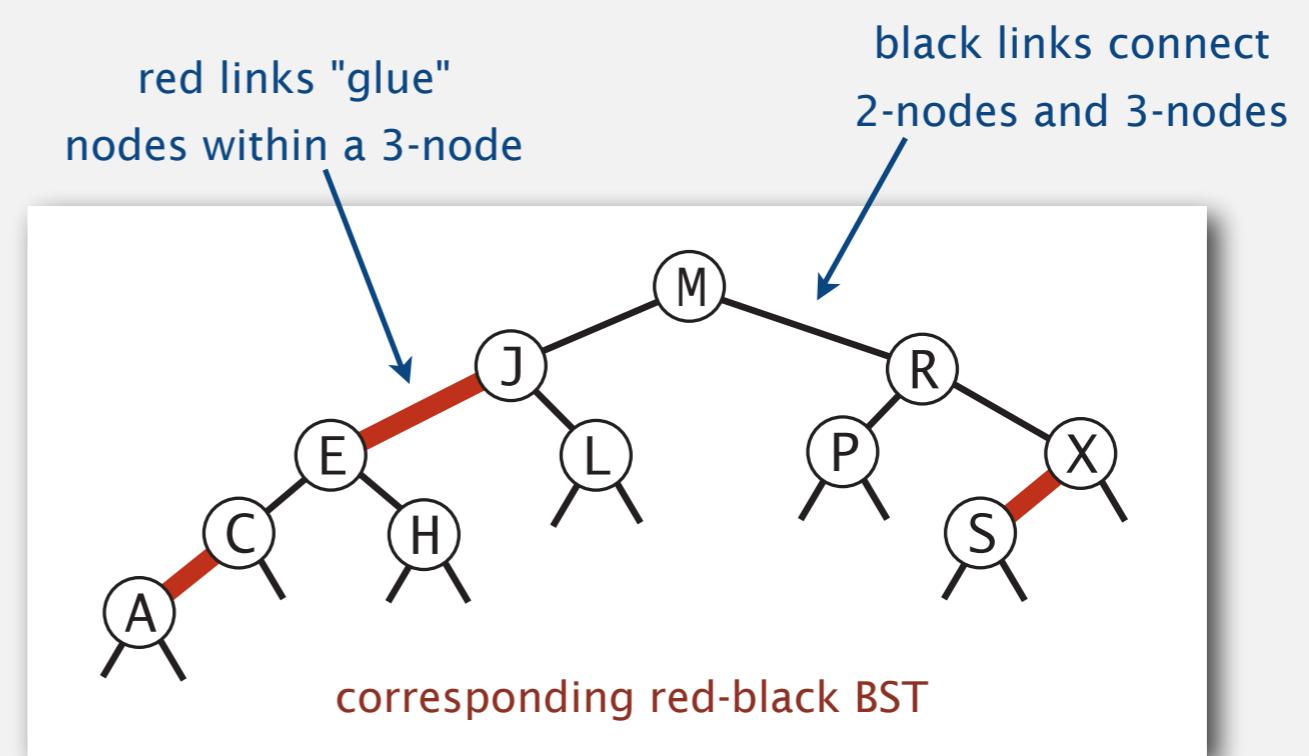
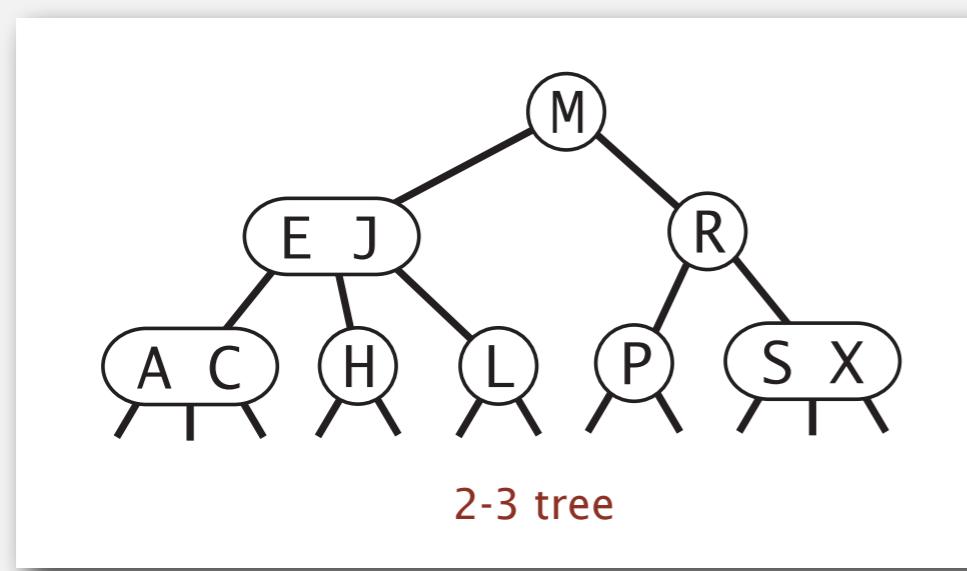
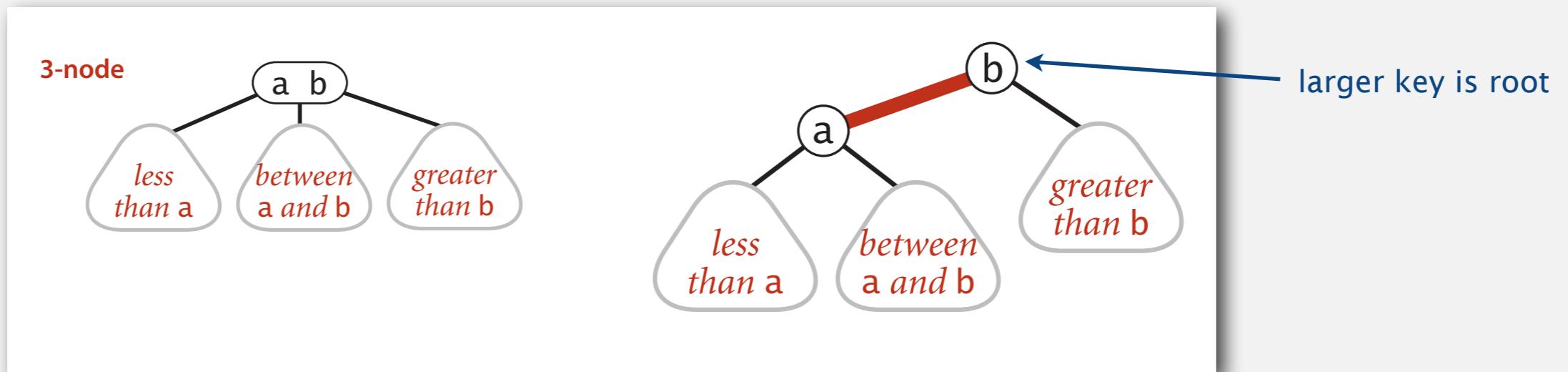
- Maintaining multiple node types is cumbersome.
- Need multiple compares to move down tree.
- Need to move back up the tree to split 4-nodes.
- Large number of cases for splitting.

Bottom line. Could do it, but there's a better way.

- 2-3 search trees
- red-black BSTs

Left-leaning red-black BSTs (Guibas-Sedgewick 1979 and Sedgewick 2007)

1. Represent 2-3 tree as a BST.
2. Use "internal" left-leaning links as "glue" for 3-nodes.

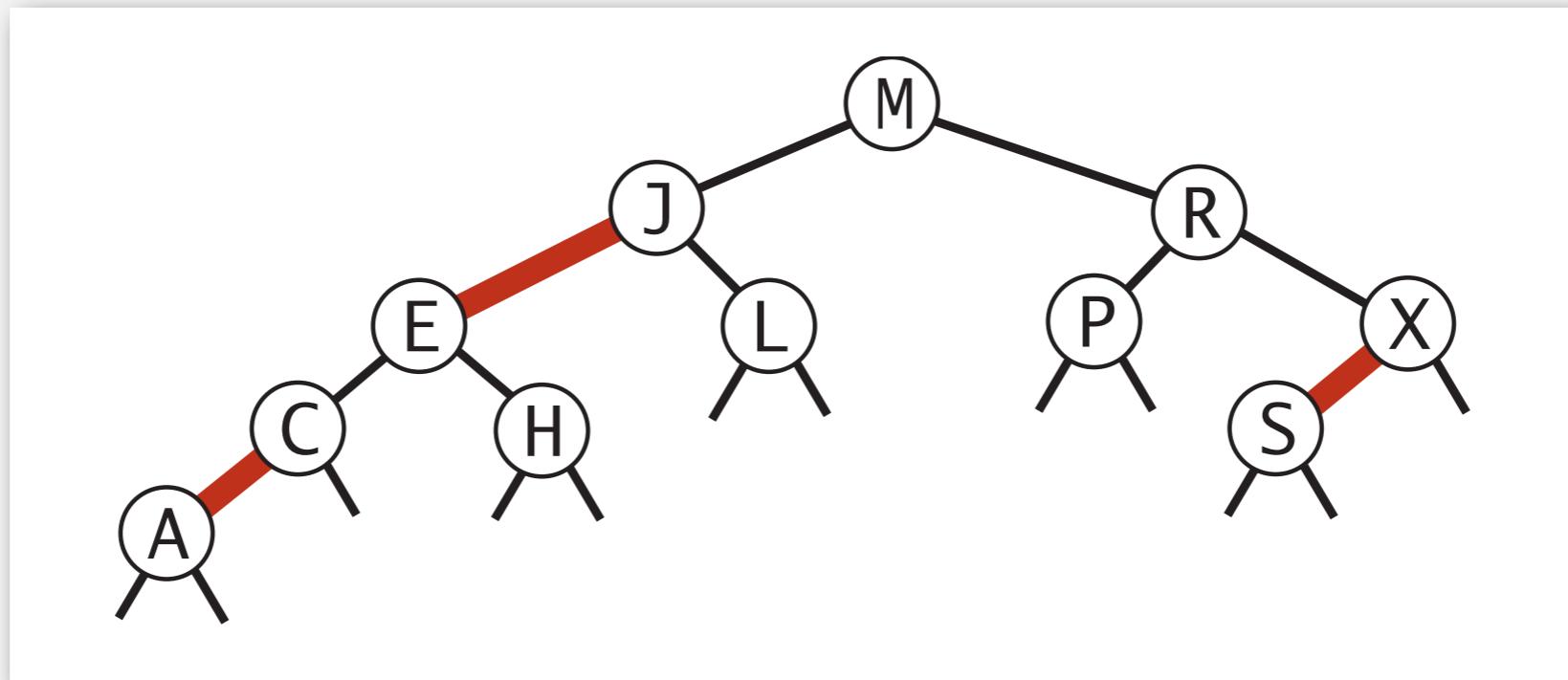


An equivalent definition

A BST such that:

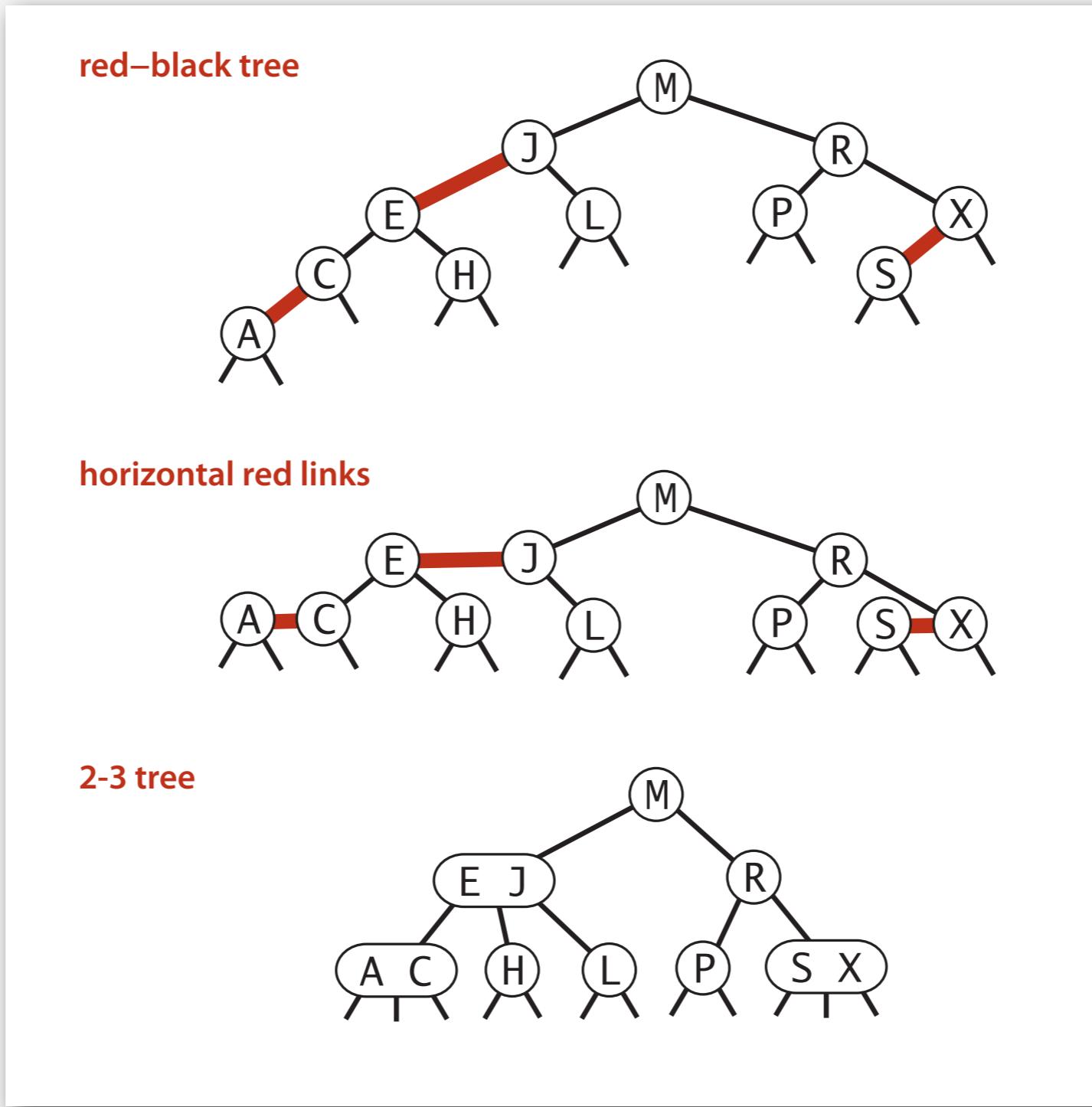
- No node has two red links connected to it.
- Every path from root to null link has the same number of black links.
- Red links lean left.

"perfect black balance"



Left-leaning red-black BSTs: 1-1 correspondence with 2-3 trees

Key property. 1-1 correspondence between 2-3 and LLRB.



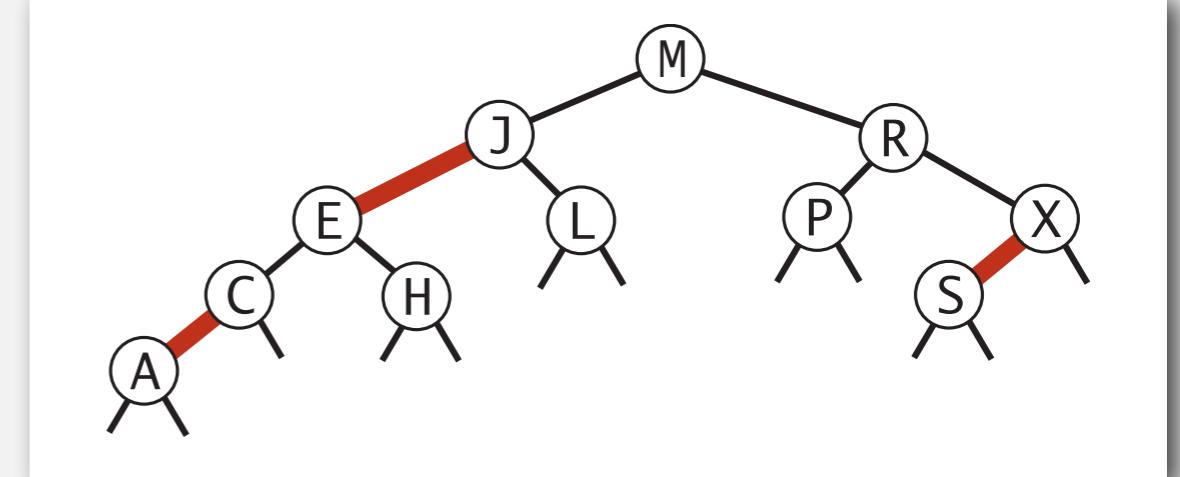
Search implementation for red-black BSTs

Observation. Search is the same as for elementary BST (ignore color).



but runs faster because of better balance

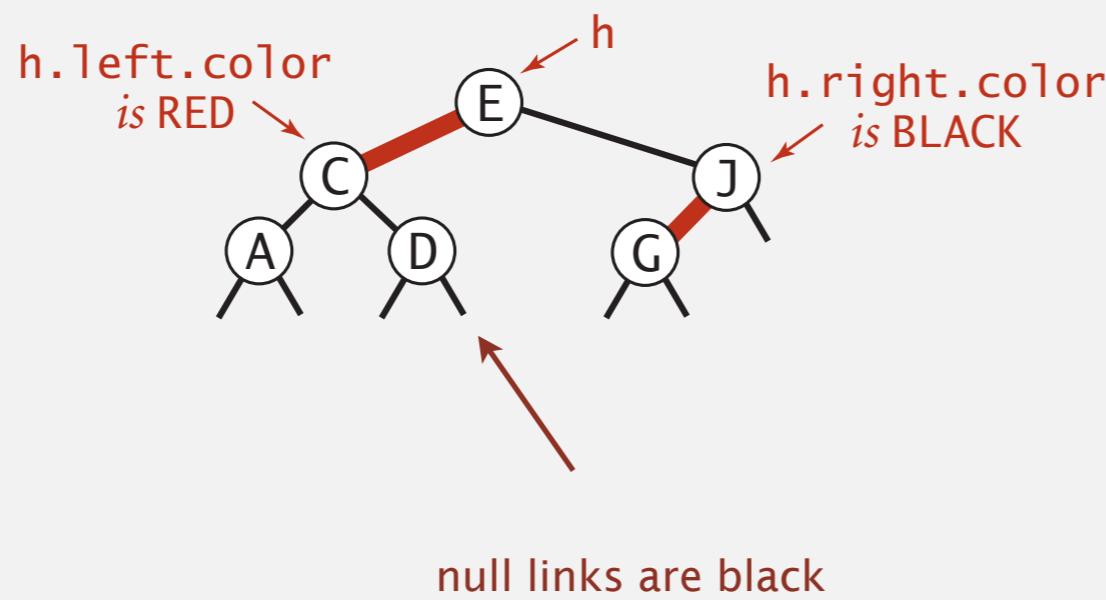
```
public Val get(Key key)
{
    Node x = root;
    while (x != null)
    {
        int cmp = key.compareTo(x.key);
        if (cmp < 0) x = x.left;
        else if (cmp > 0) x = x.right;
        else if (cmp == 0) return x.val;
    }
    return null;
}
```



Remark. Most other ops (e.g., ceiling, selection, iteration) are also identical.

Red-black BST representation

Each node is pointed to by precisely one link (from its parent) \Rightarrow
can encode color of links in nodes.

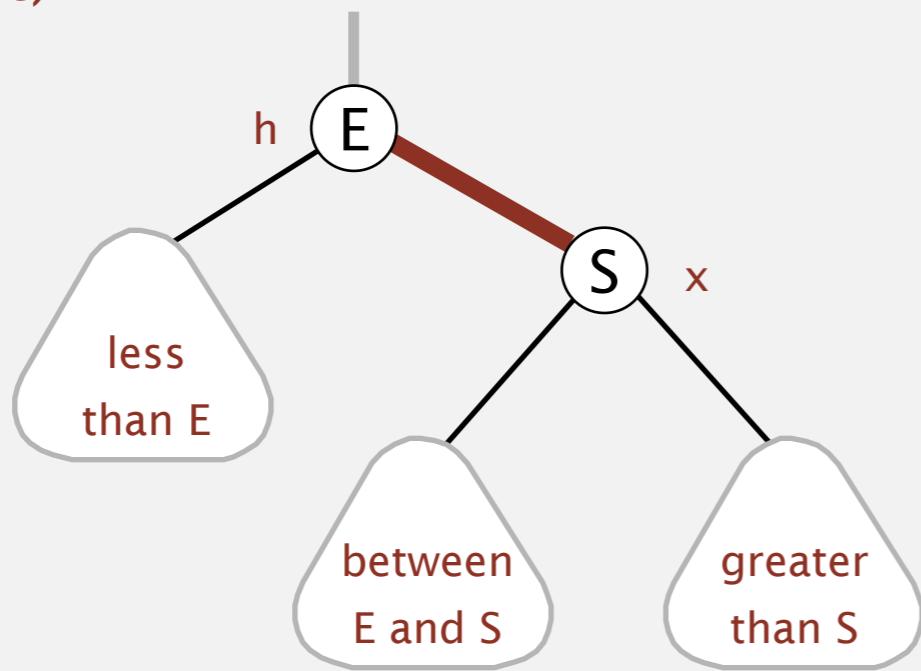


Elementary red-black BST operations

Left rotation. Orient a (temporarily) right-leaning red link to lean left.

rotate E left

(before)



```
private Node rotateLeft(Node h)
{
    assert isRed(h.right);
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```

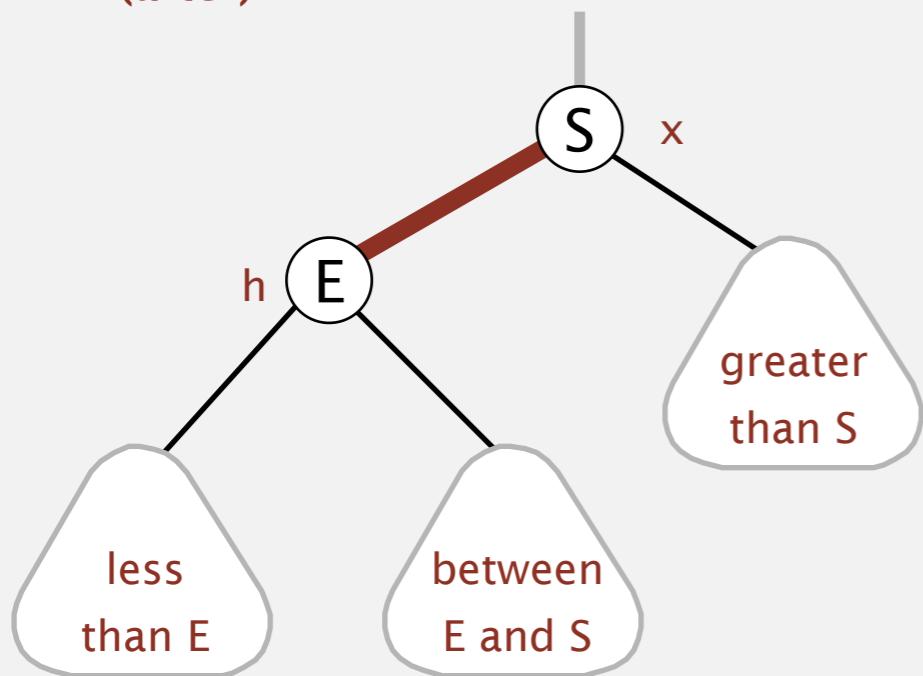
Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Left rotation. Orient a (temporarily) right-leaning red link to lean left.

rotate E left

(after)



```
private Node rotateLeft(Node h)
{
    assert isRed(h.right);
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```

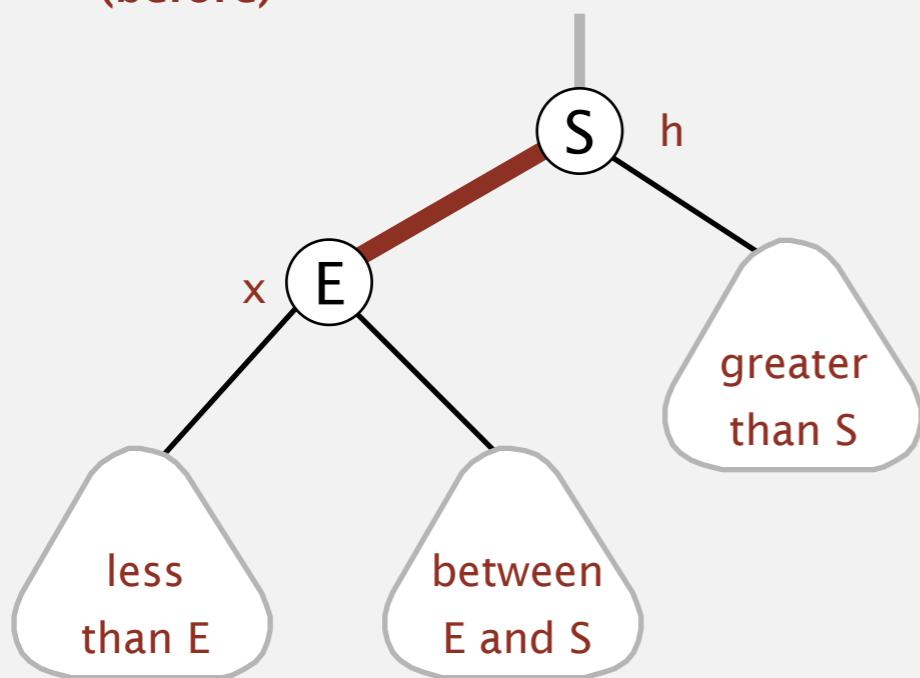
Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Right rotation. Orient a left-leaning red link to (temporarily) lean right.

rotate S right

(before)



```
private Node rotateRight(Node h)
{
    assert isRed(h.left);
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```

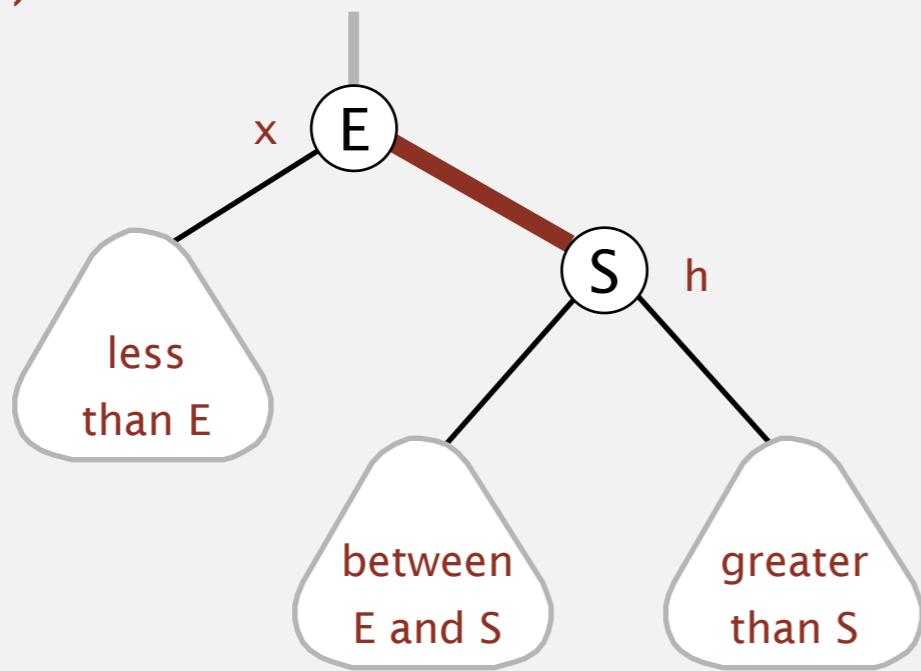
Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Right rotation. Orient a left-leaning red link to (temporarily) lean right.

rotate S right

(after)



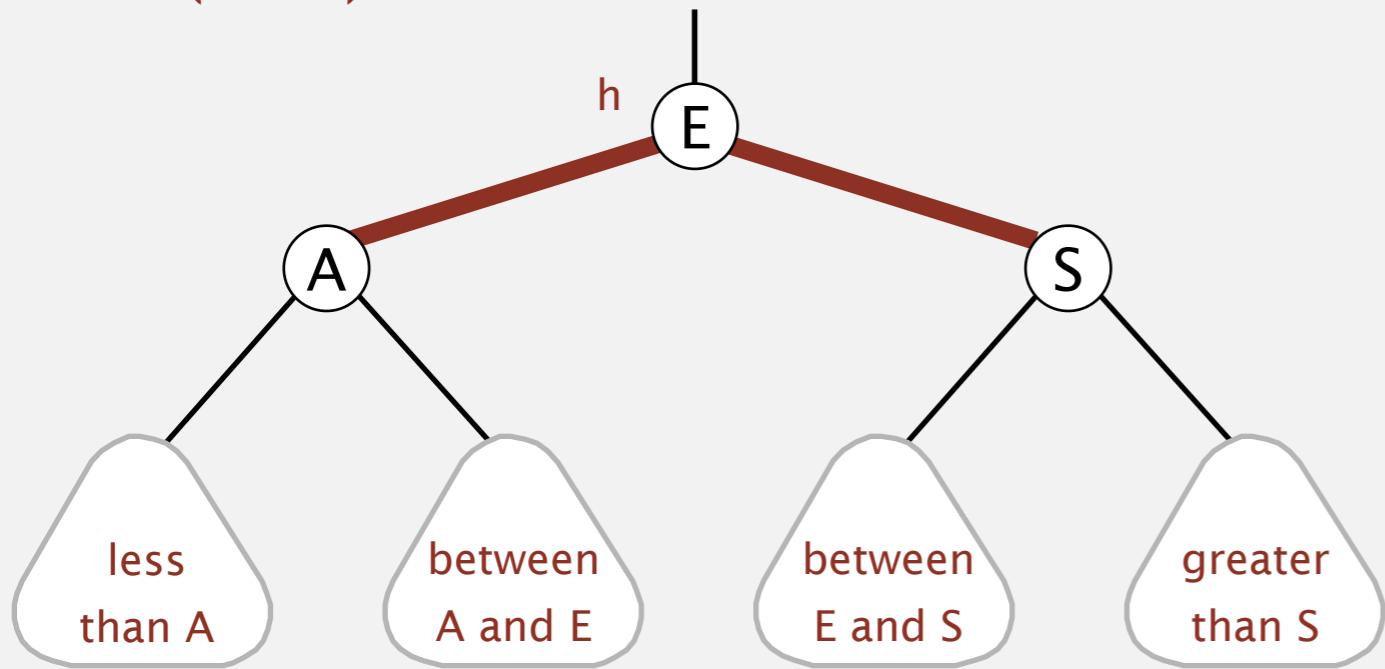
```
private Node rotateRight(Node h)
{
    assert isRed(h.left);
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Color flip. Recolor to split a (temporary) 4-node.

flip colors
(before)



```
private void flipColors(Node h)
{
    assert !isRed(h);
    assert isRed(h.left);
    assert isRed(h.right);
    h.color = RED;
    h.left.color = BLACK;
    h.right.color = BLACK;
}
```

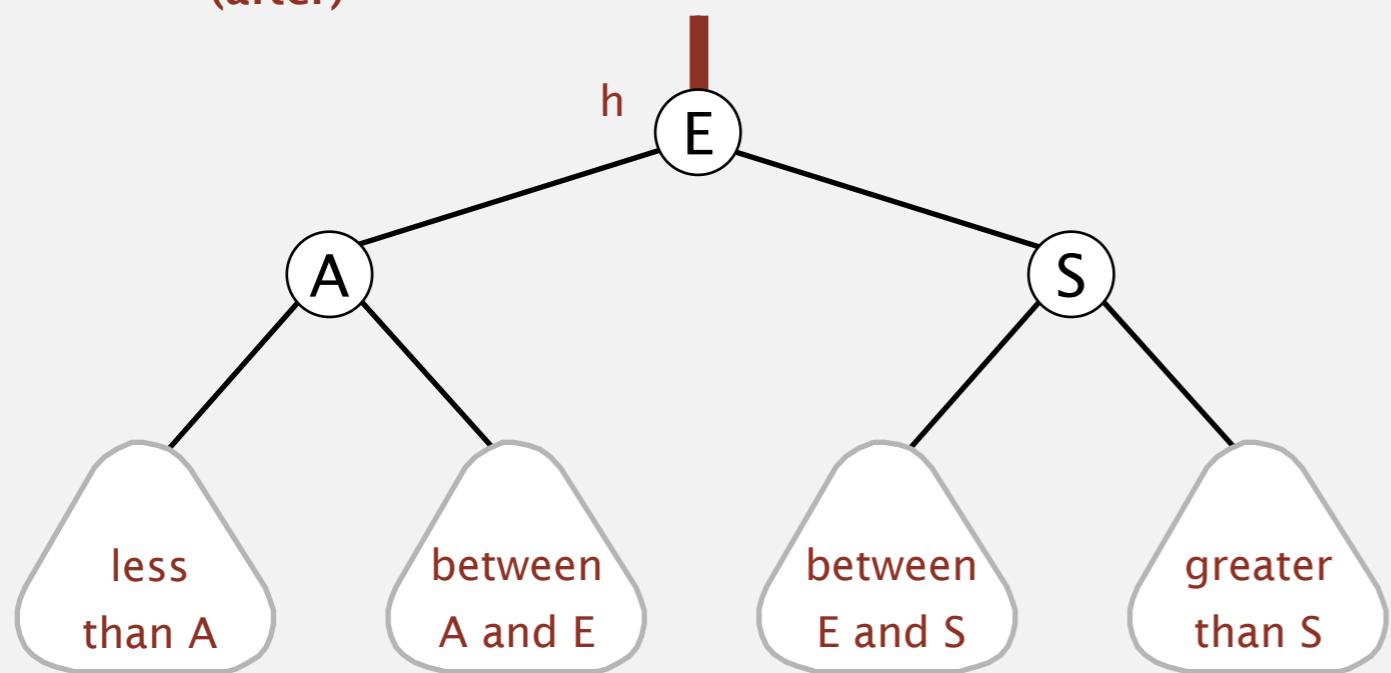
Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Color flip. Recolor to split a (temporary) 4-node.

flip colors

(after)

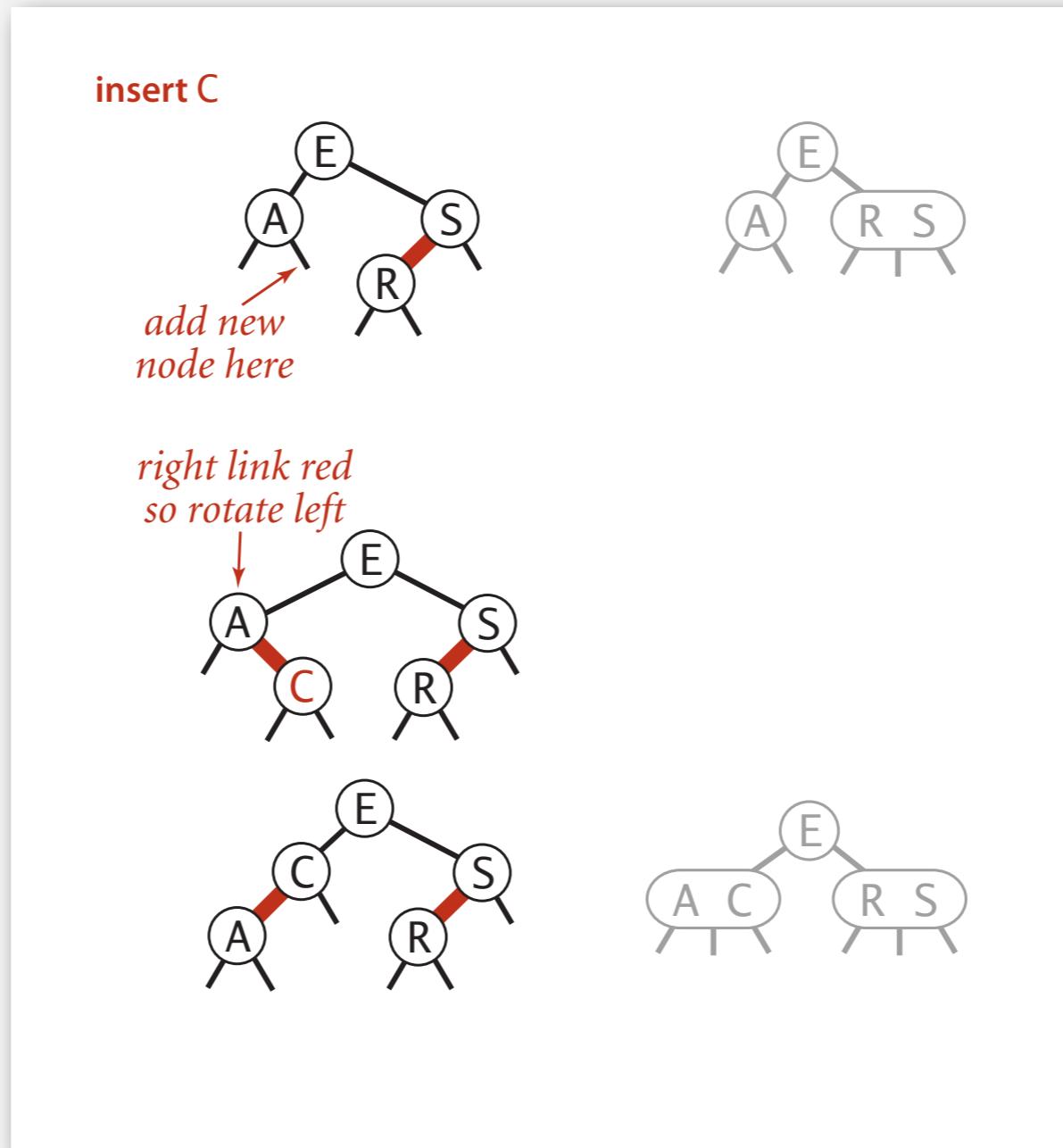


```
private void flipColors(Node h)
{
    assert !isRed(h);
    assert isRed(h.left);
    assert isRed(h.right);
    h.color = RED;
    h.left.color = BLACK;
    h.right.color = BLACK;
}
```

Invariants. Maintains symmetric order and perfect black balance.

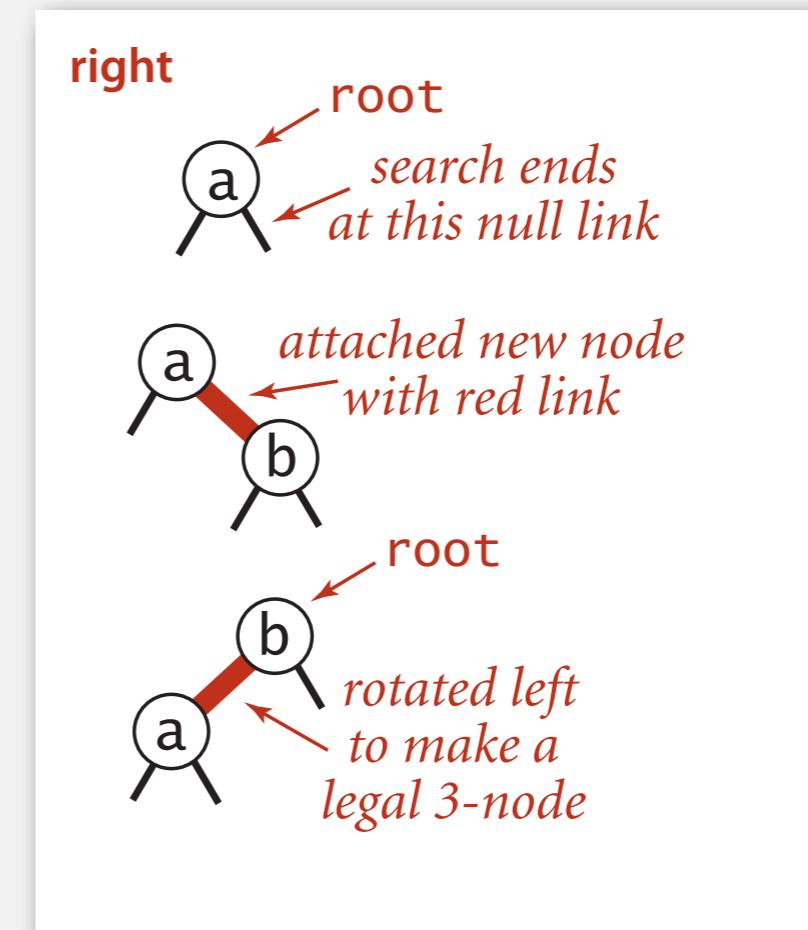
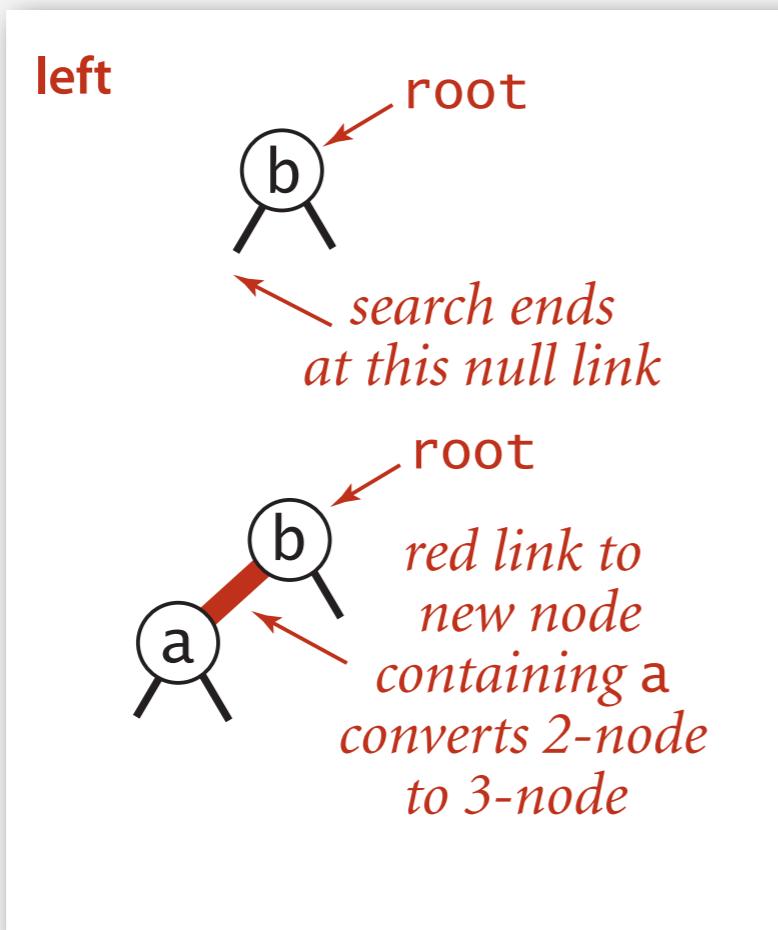
Insertion in a LLRB tree: overview

Basic strategy. Maintain 1-1 correspondence with 2-3 trees by applying elementary red-black BST operations.



Insertion in a LLRB tree

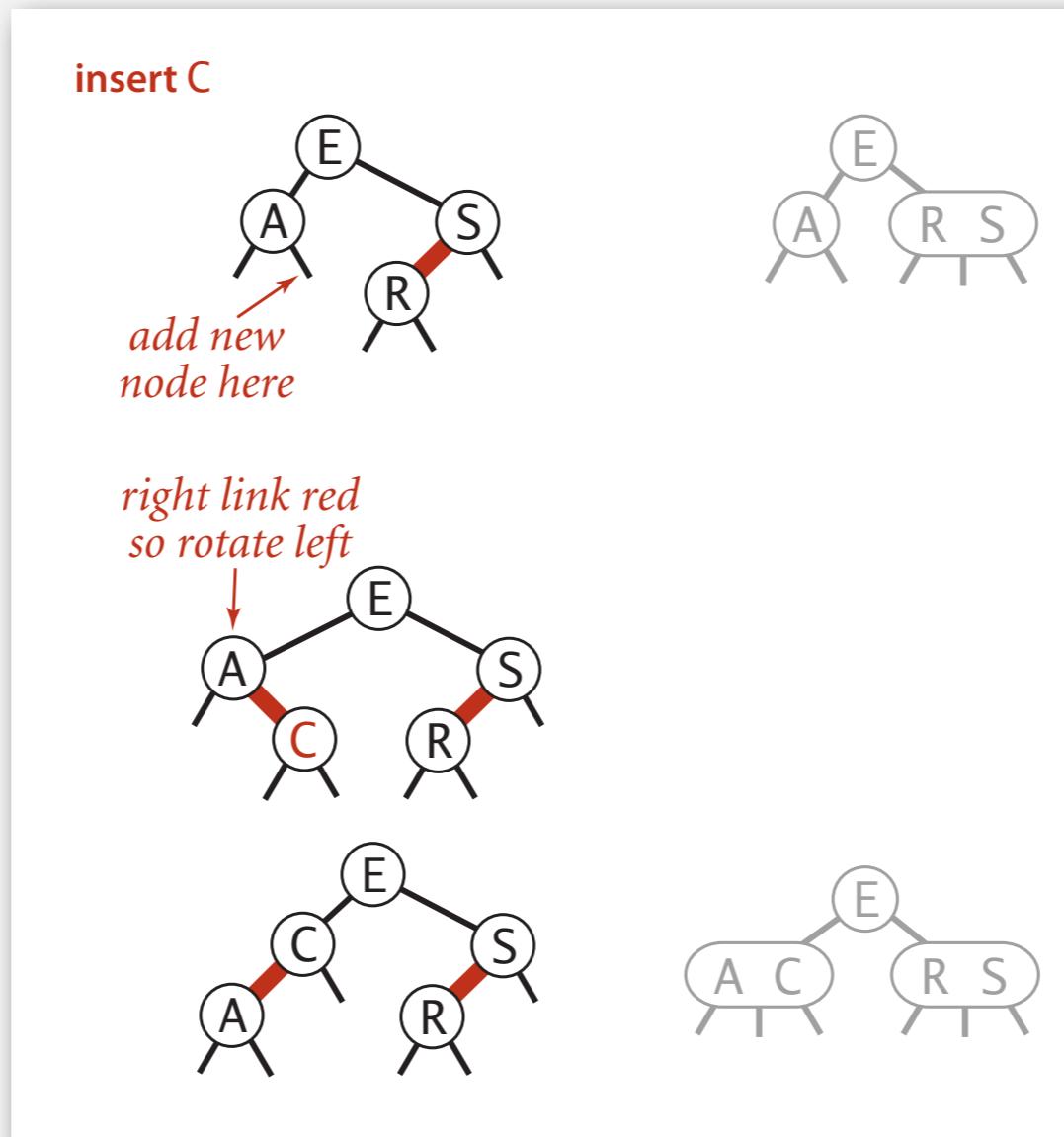
Warmup 1. Insert into a tree with exactly 1 node.



Insertion in a LLRB tree

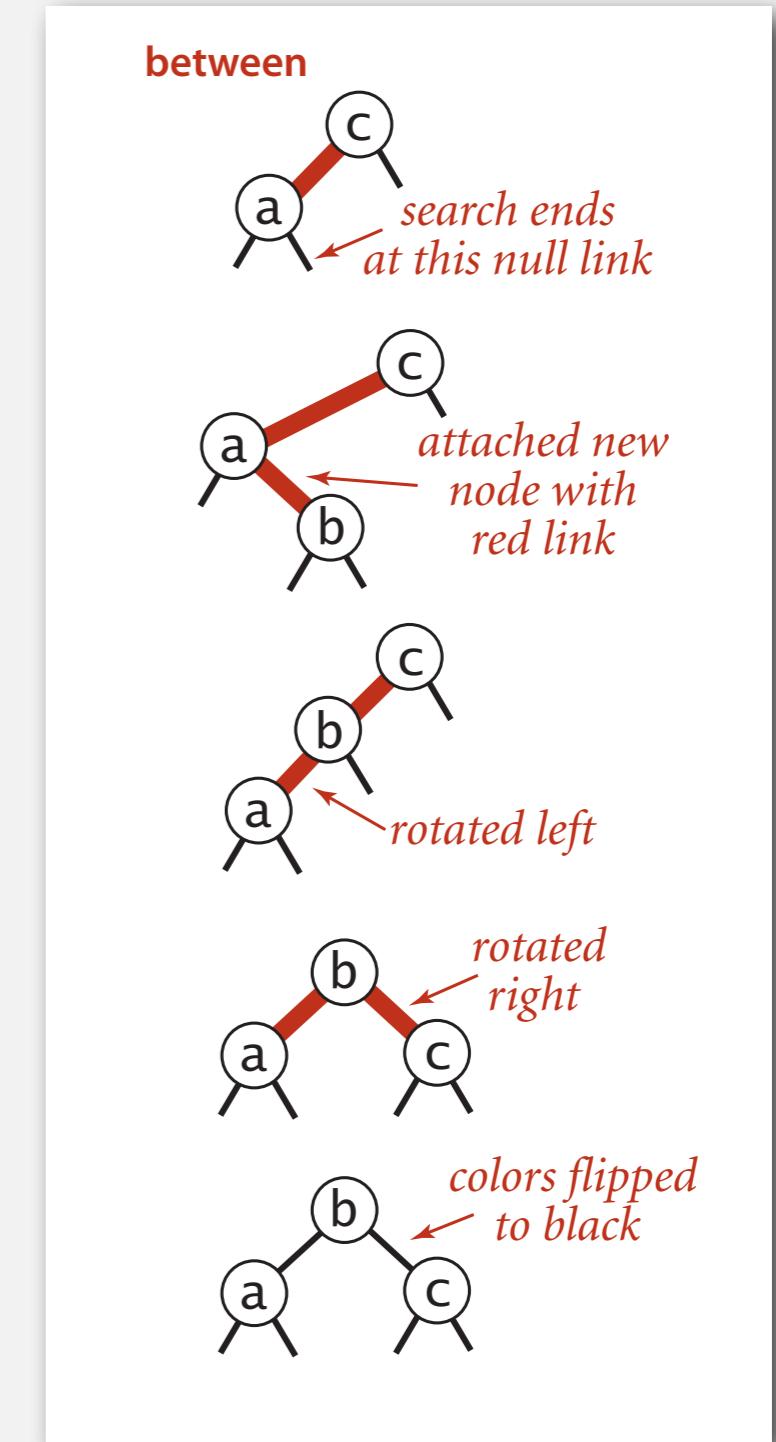
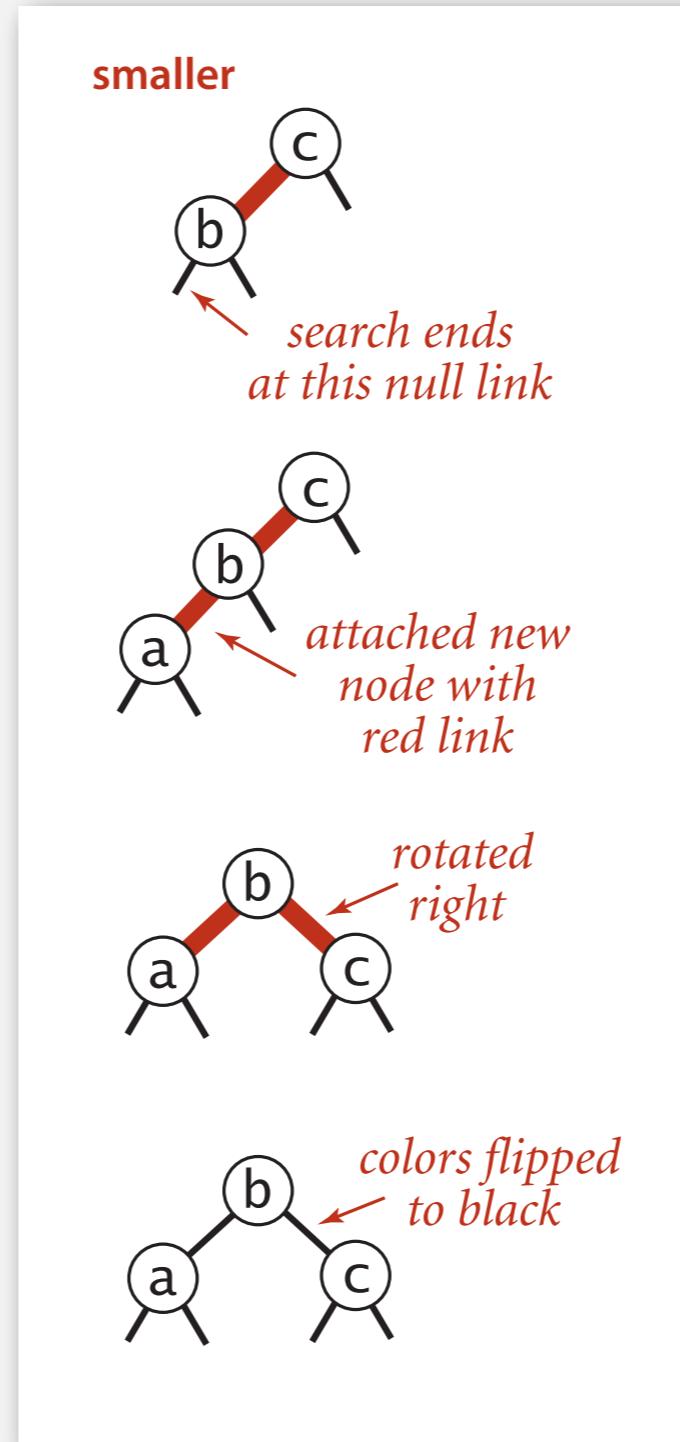
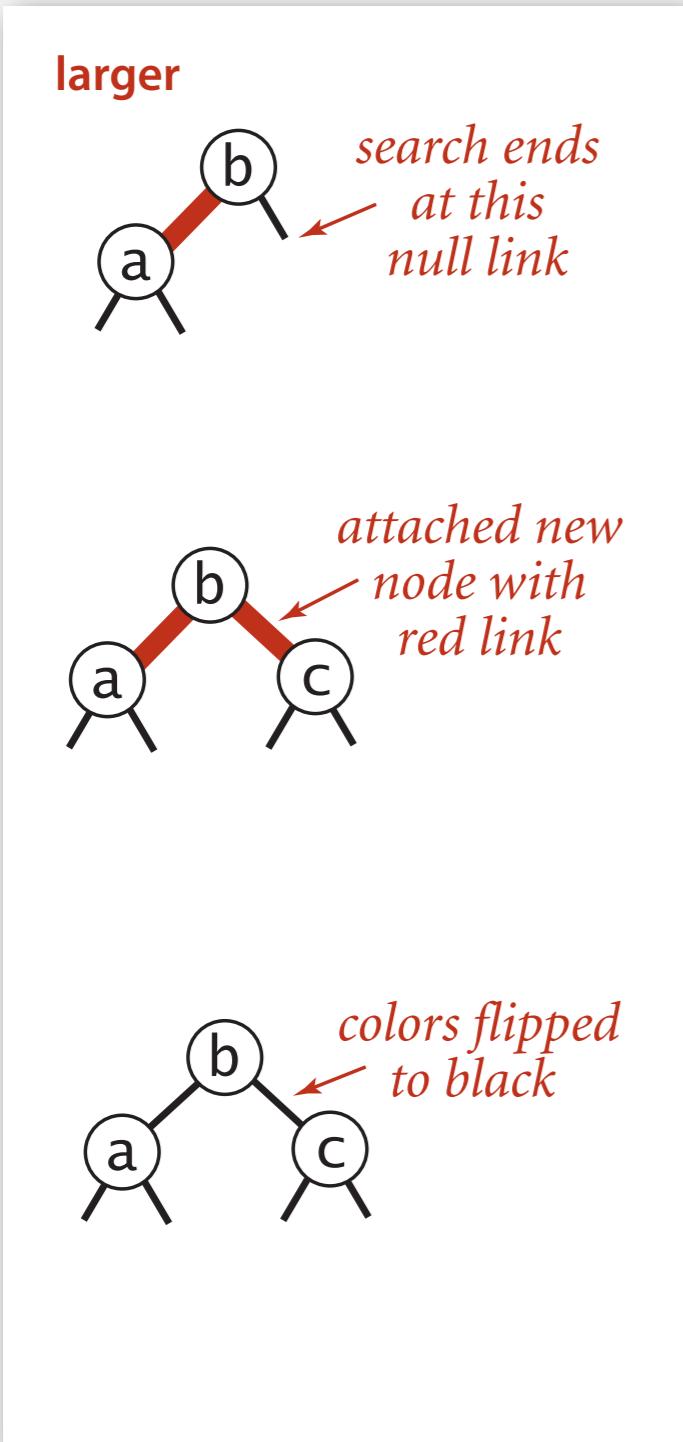
Case 1. Insert into a 2-node at the bottom.

- Do standard BST insert; color new link red.
- If new red link is a right link, rotate left.



Insertion in a LLRB tree

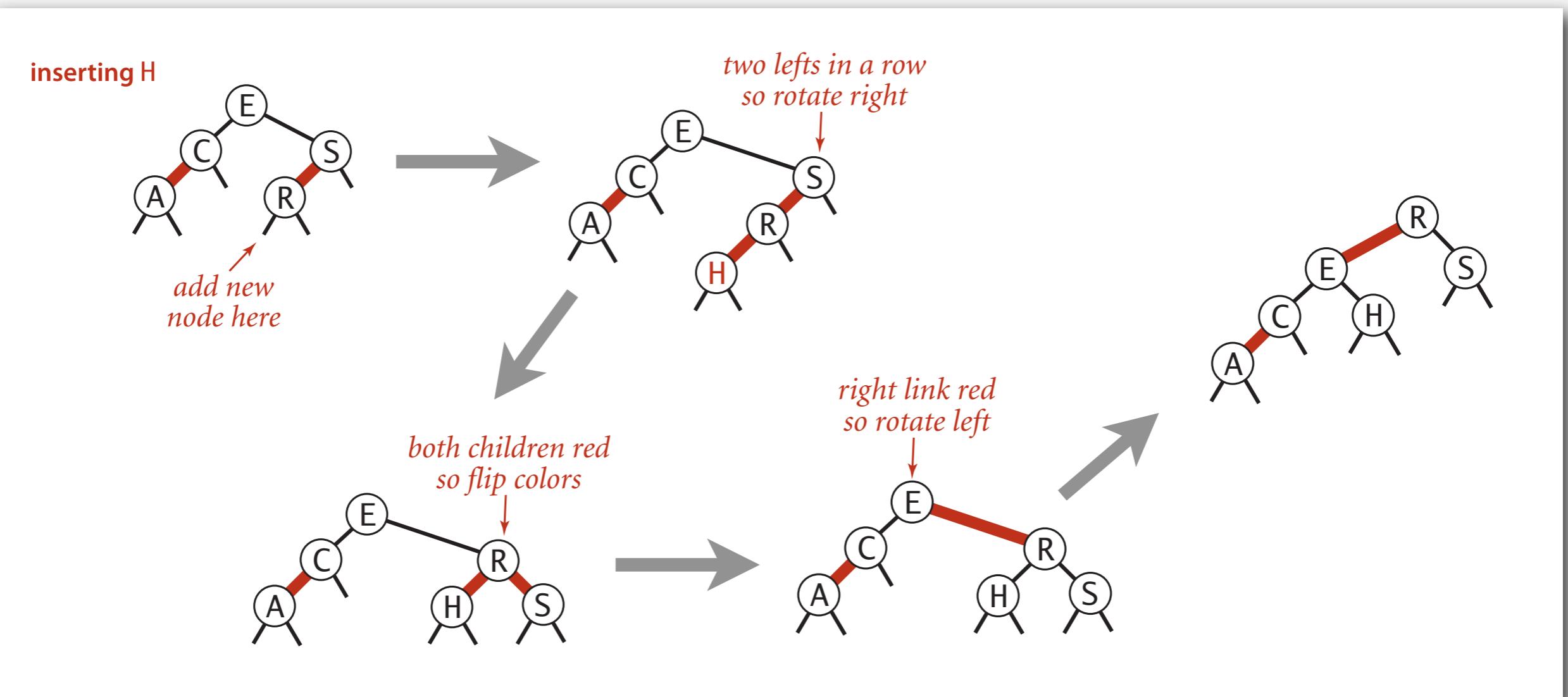
Warmup 2. Insert into a tree with exactly 2 nodes (or in a 3 node)



Insertion in a LLRB tree

Case 2. Insert into a 3-node at the bottom.

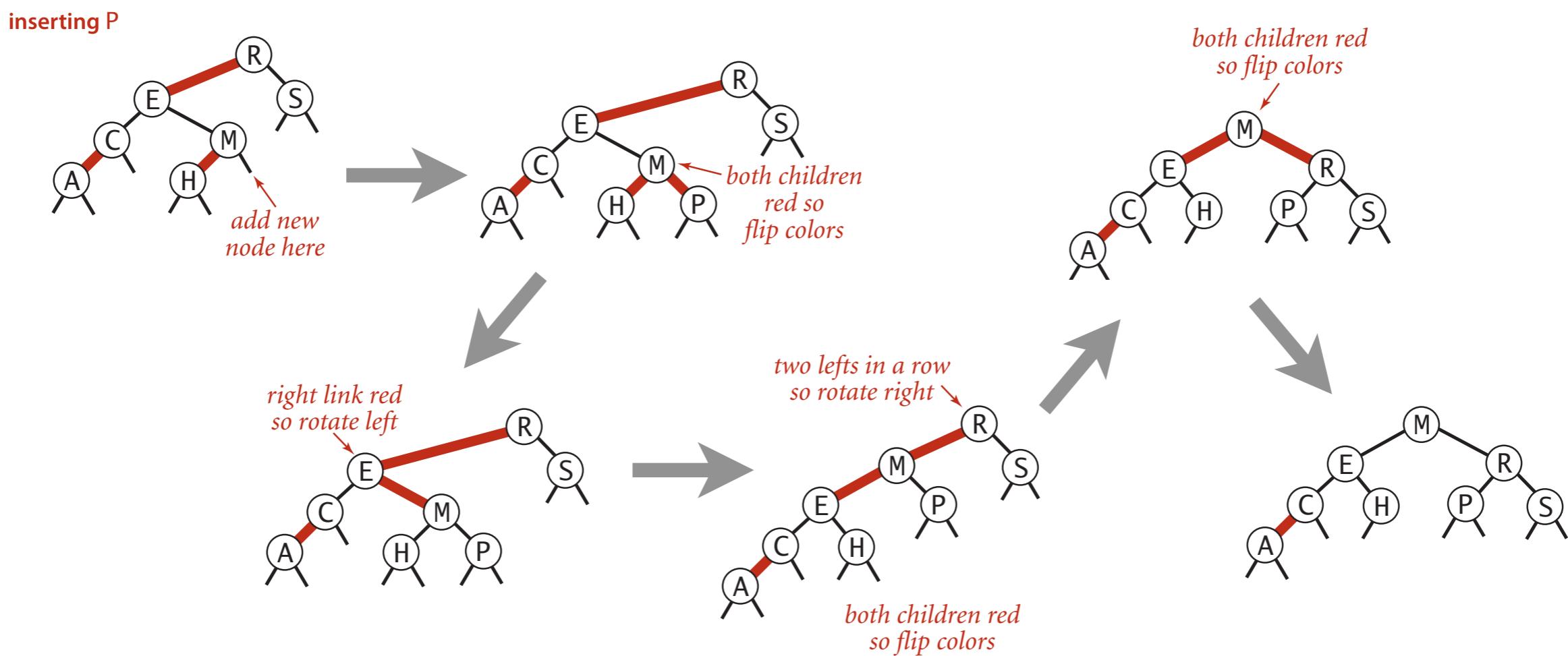
- Do standard BST insert; color new link red.
- Rotate to balance the 4-node (if needed).
- Flip colors to pass red link up one level.
- Rotate to make lean left (if needed).



Insertion in a LLRB tree: passing red links up the tree

Case 2. Insert into a 3-node at the bottom.

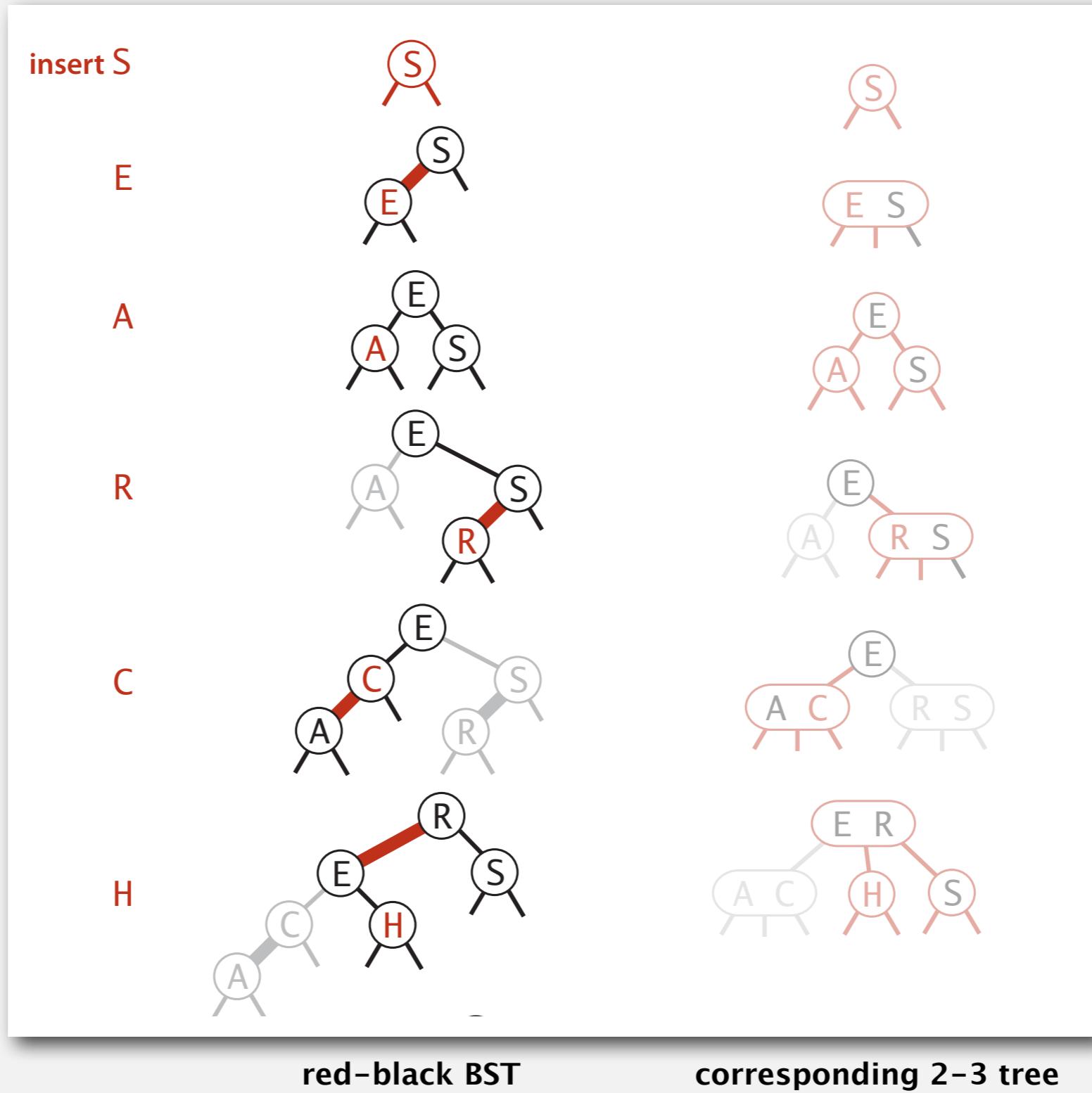
- Do standard BST insert; color new link red.
- Rotate to balance the 4-node (if needed).
- Flip colors to pass red link up one level.
- Rotate to make lean left (if needed).
- Repeat case 1 or case 2 up the tree (if needed).



LLRB tree insertion demo

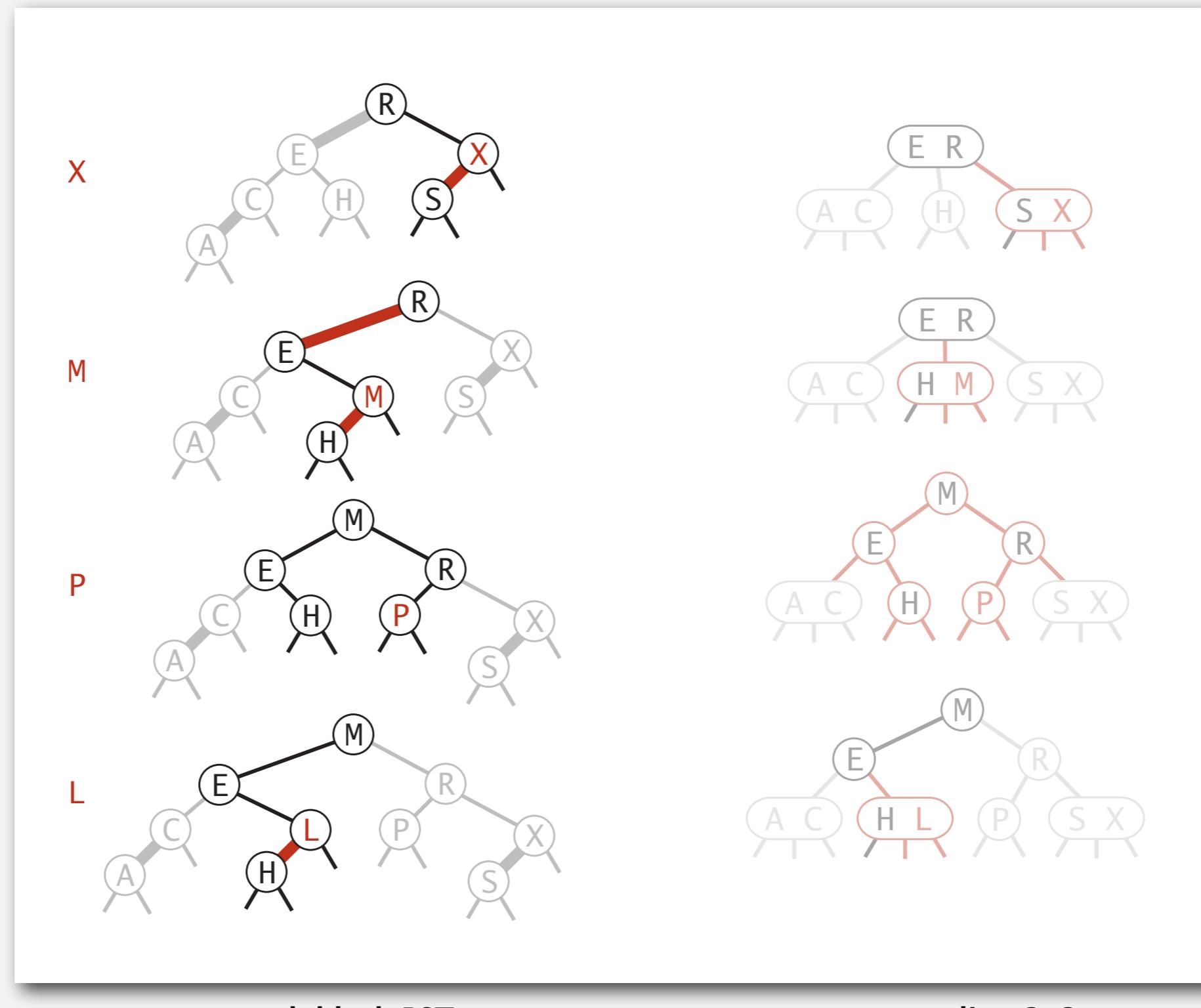
LLRB tree insertion trace

Standard indexing client.



LLRB tree insertion trace

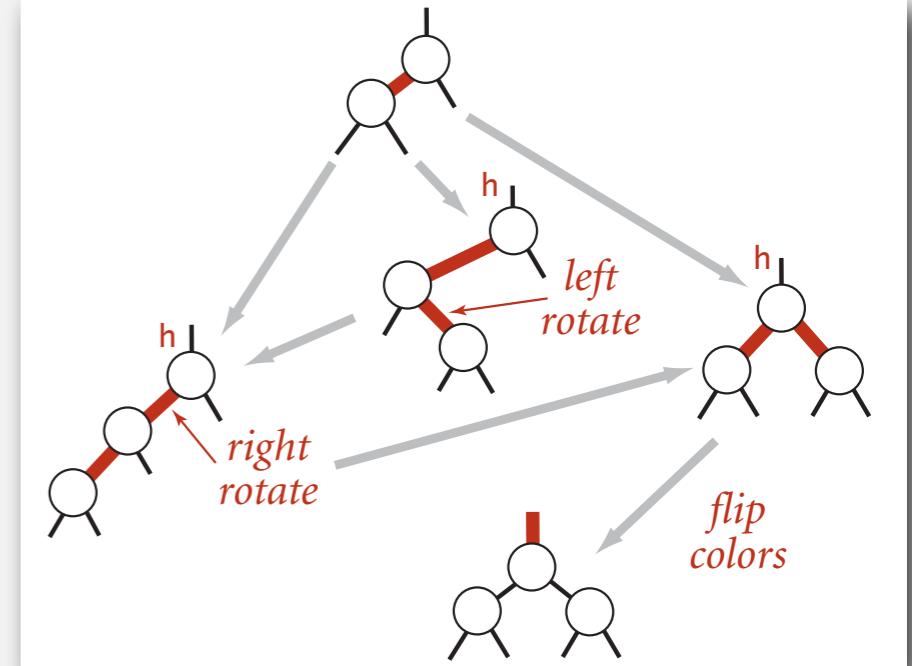
Standard indexing client (continued).



Insertion in a LLRB tree: Maintaining 1-1 correspondence with 2-3 Trees

Same code for both cases.

- Right child red, left child black: **rotate left**.
- Left child, left-left grandchild red: **rotate right**.
- Both children red: **flip colors**.



```
private Node put(Node h, Key key, Value val)
{
    if (h == null) return new Node(key, val, RED);
    int cmp = key.compareTo(h.key);
    if (cmp < 0) h.left = put(h.left, key, val);
    else if (cmp > 0) h.right = put(h.right, key, val);
    else if (cmp == 0) h.val = val;

    if (isRed(h.right) && !isRed(h.left))      h = rotateLeft(h);
    if (isRed(h.left) && isRed(h.left.left))   h = rotateRight(h);
    if (isRed(h.left) && isRed(h.right))       flipColors(h);

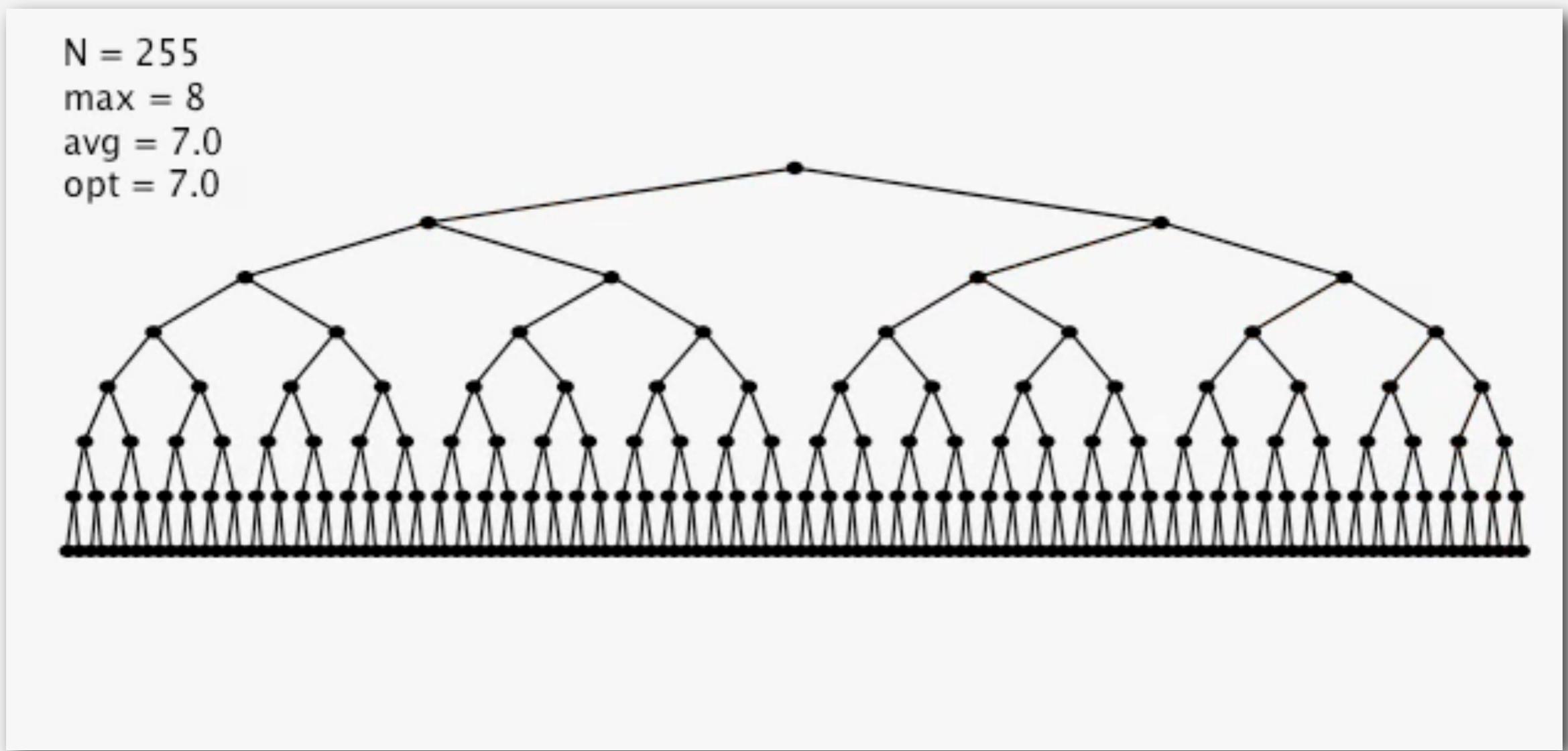
    return h;
}
```

only a few extra lines of code
to provide near-perfect balance

insert at bottom
(and color red)

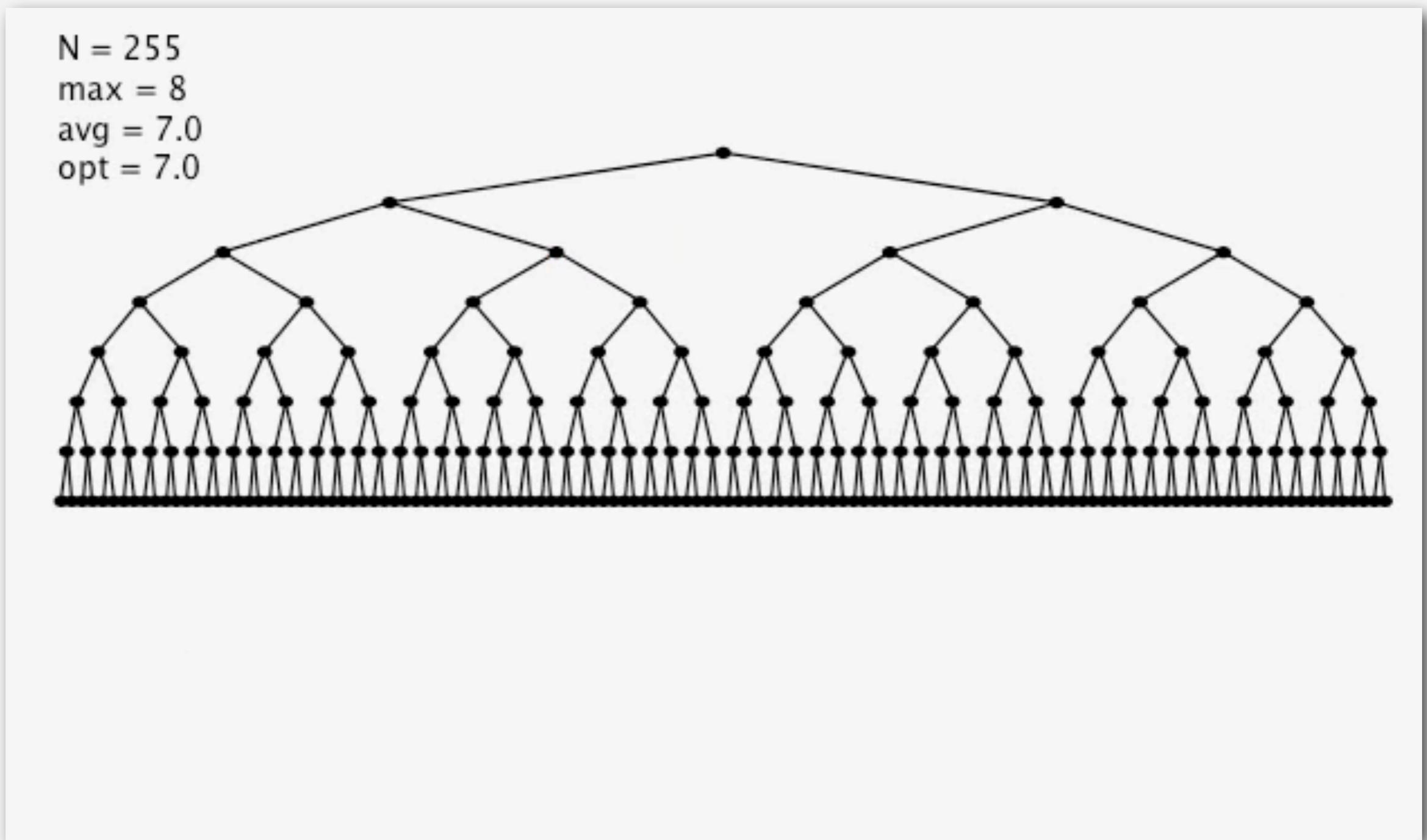
lean left
balance 4-node
split 4-node

Insertion in a LLRB tree: visualization



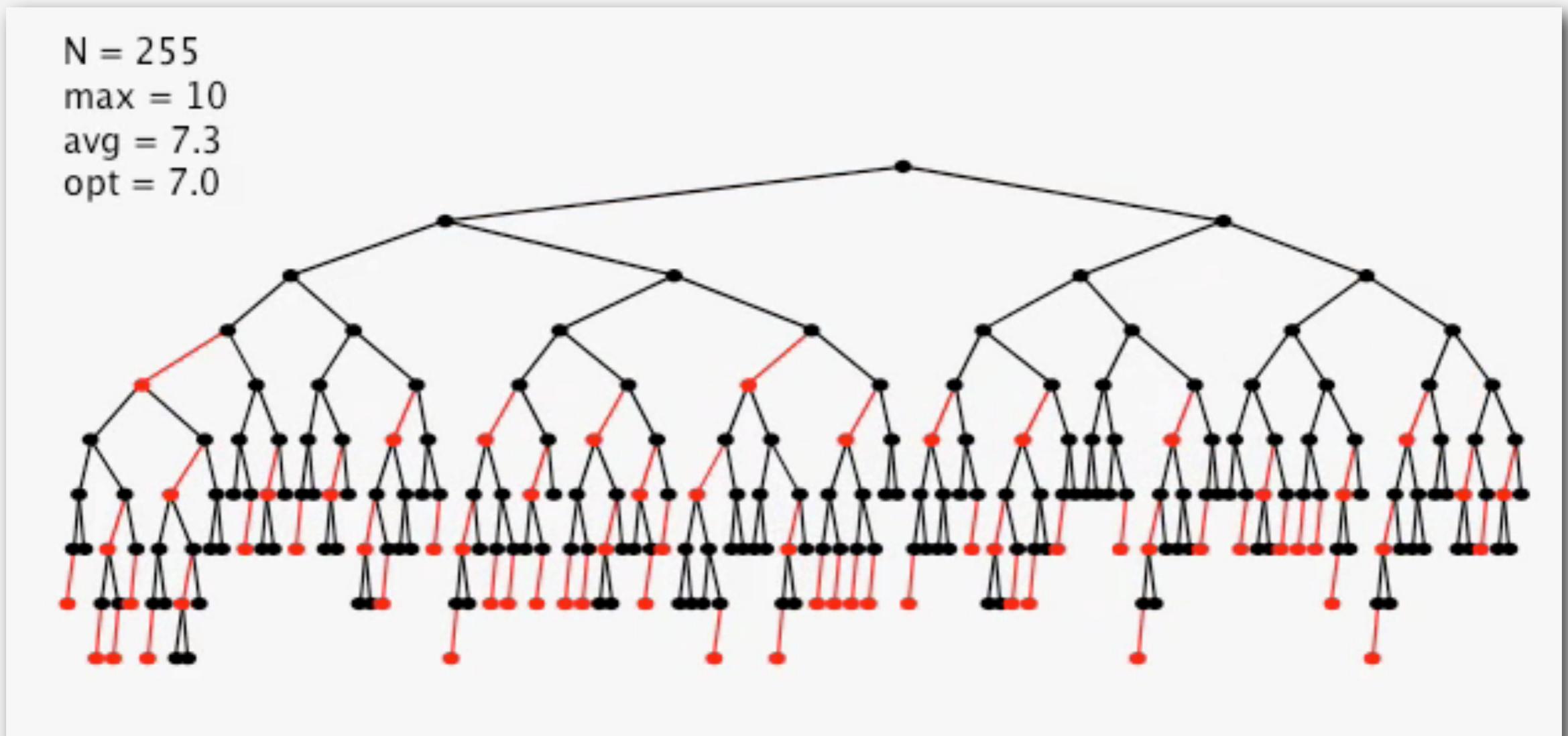
255 insertions in ascending order

Insertion in a LLRB tree: visualization



255 insertions in descending order

Insertion in a LLRB tree: visualization

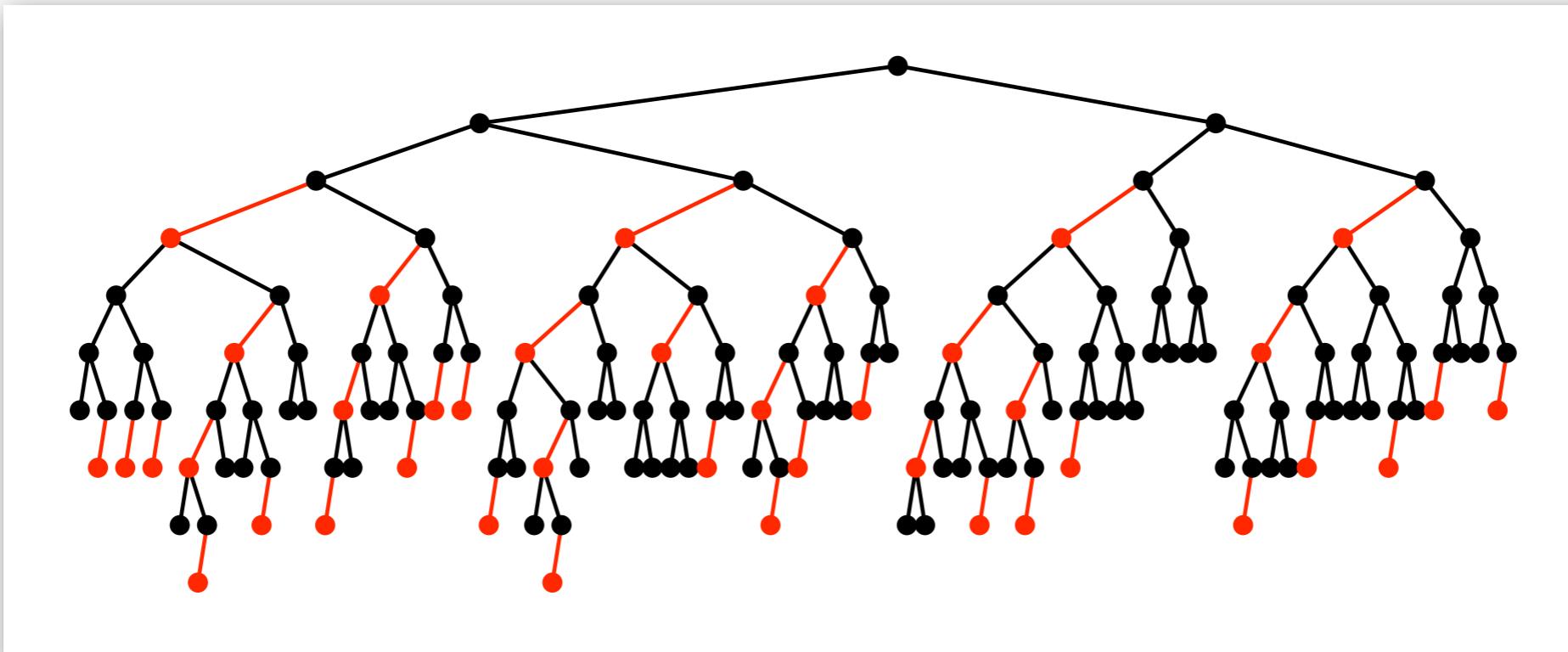


Balance in LLRB trees

Proposition. Height of tree is $\leq 2 \lg N$ in the worst case.

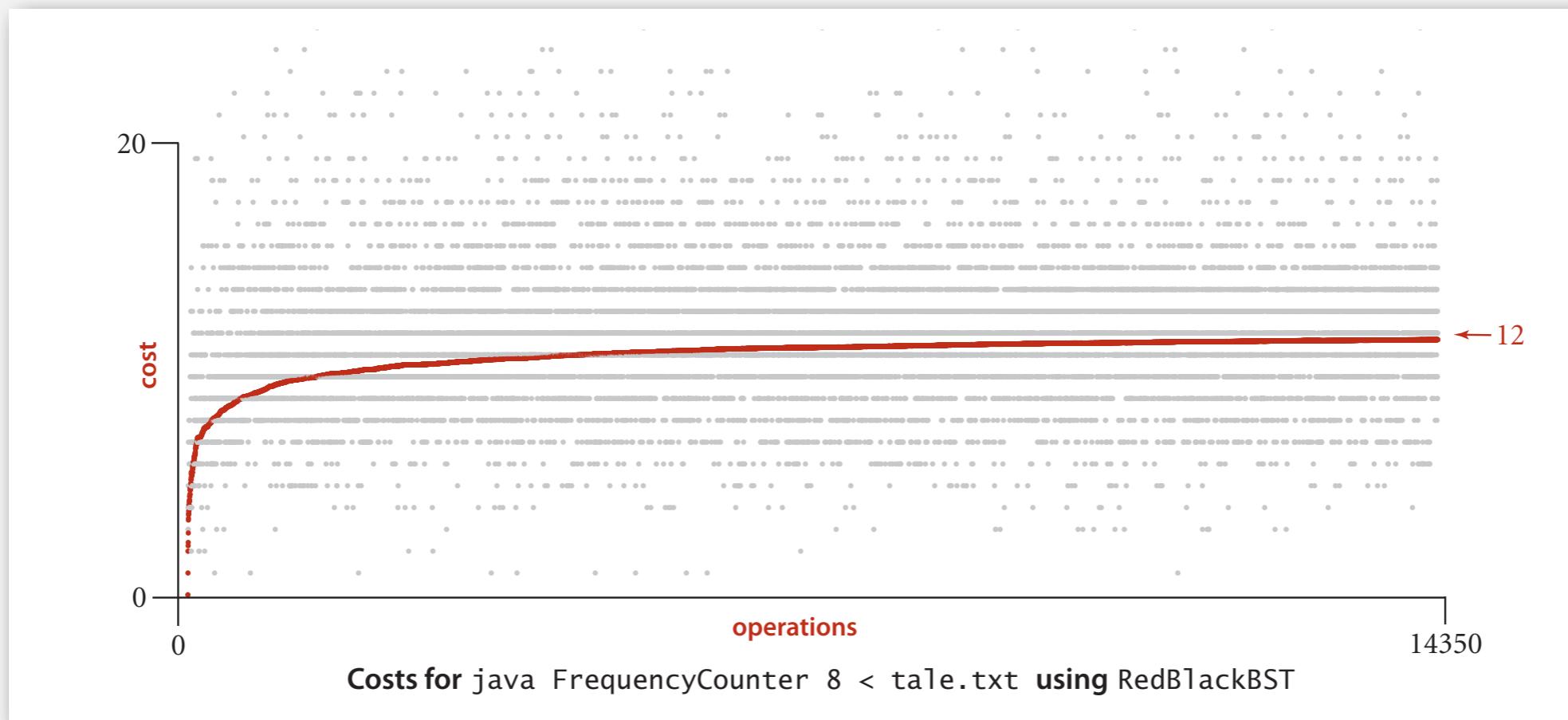
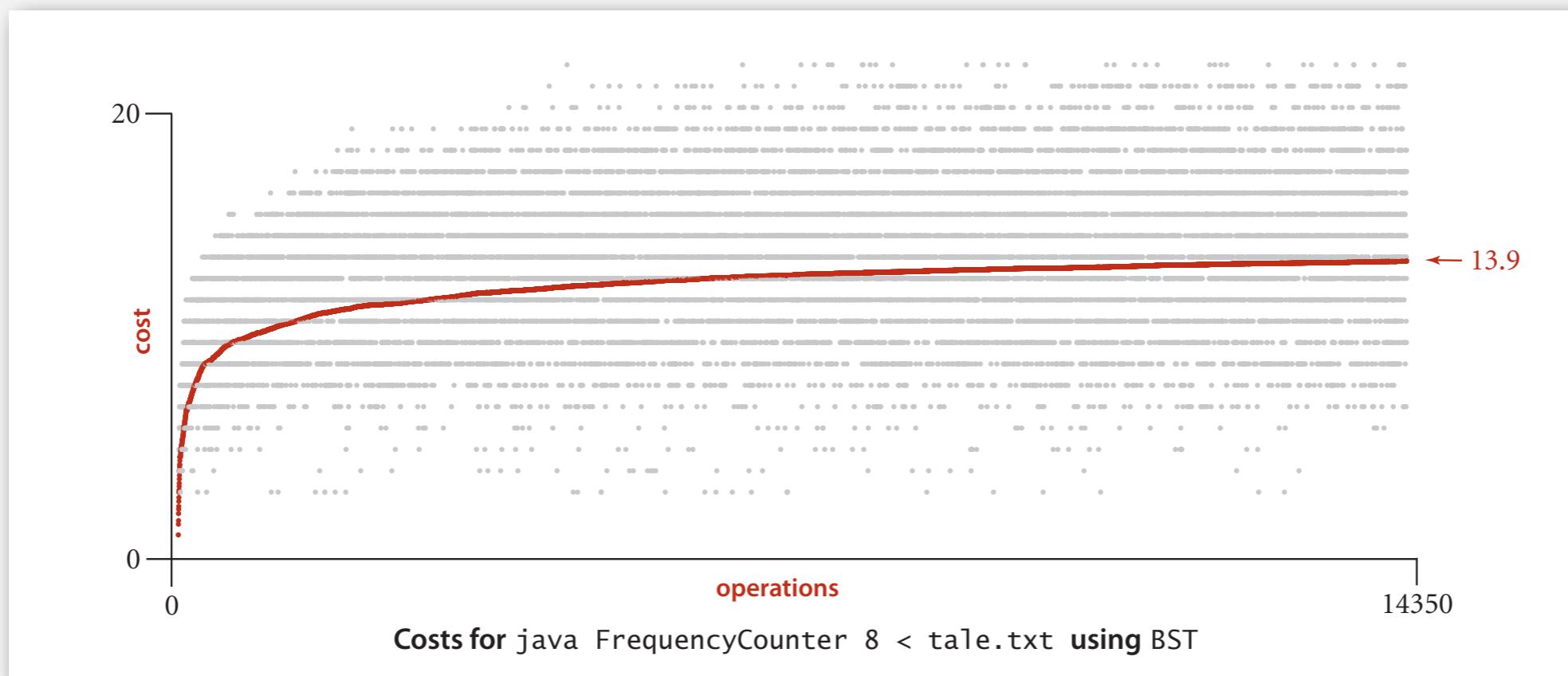
Pf.

- Every path from root to null link has same number of black links.
- Never two red links in-a-row.



Property. Height of tree is $\sim 1.00 \lg N$ in typical applications.

ST implementations: frequency counter



ST implementations: summary

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	N/2	N/2	yes	<code>compareTo()</code>
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$?	yes	<code>compareTo()</code>
2-3 tree	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	yes	<code>compareTo()</code>
red-black BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.00 \lg N$	$1.00 \lg N$ *	$1.00 \lg N$ *	yes	<code>compareTo()</code>

* exact value of coefficient unknown but extremely close to 1

Why left-leaning red-black BSTs?

Simplified code.

- Left-leaning restriction reduces number of cases.
- Short inner loop.

Same ideas simplify implementation of other operations.

- Delete min/max.
- Arbitrary delete.

2008
1978

Improves widely-used balanced search trees.

- AVL trees, splay trees, randomized BSTs, ...
- 2-3 trees, 2-3-4 trees.
- Red-black BSTs.

1972

Bottom line. Left-leaning red-black BSTs are among the simplest balanced BSTs to implement and among the fastest in practice.

- ▶ 2-3 search trees
- ▶ red-black BSTs
- ▶ B-trees

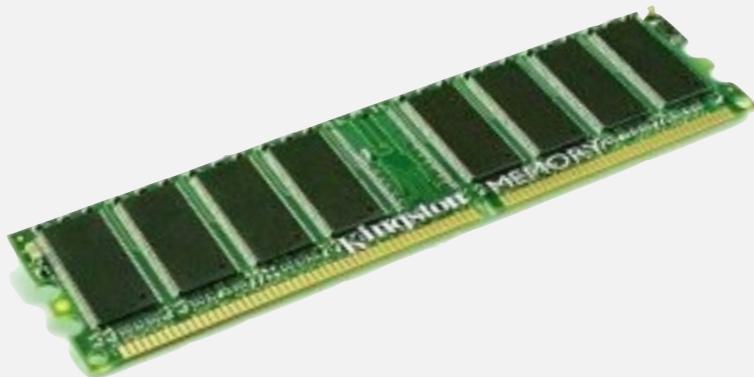
File system model

Page. Contiguous block of data (e.g., a file or 4,096-byte chunk).

Probe. First access to a page (e.g., from disk to memory).



slow



fast

Property. Time required for a probe is much larger than time to access data within a page.

Cost model. Number of probes.

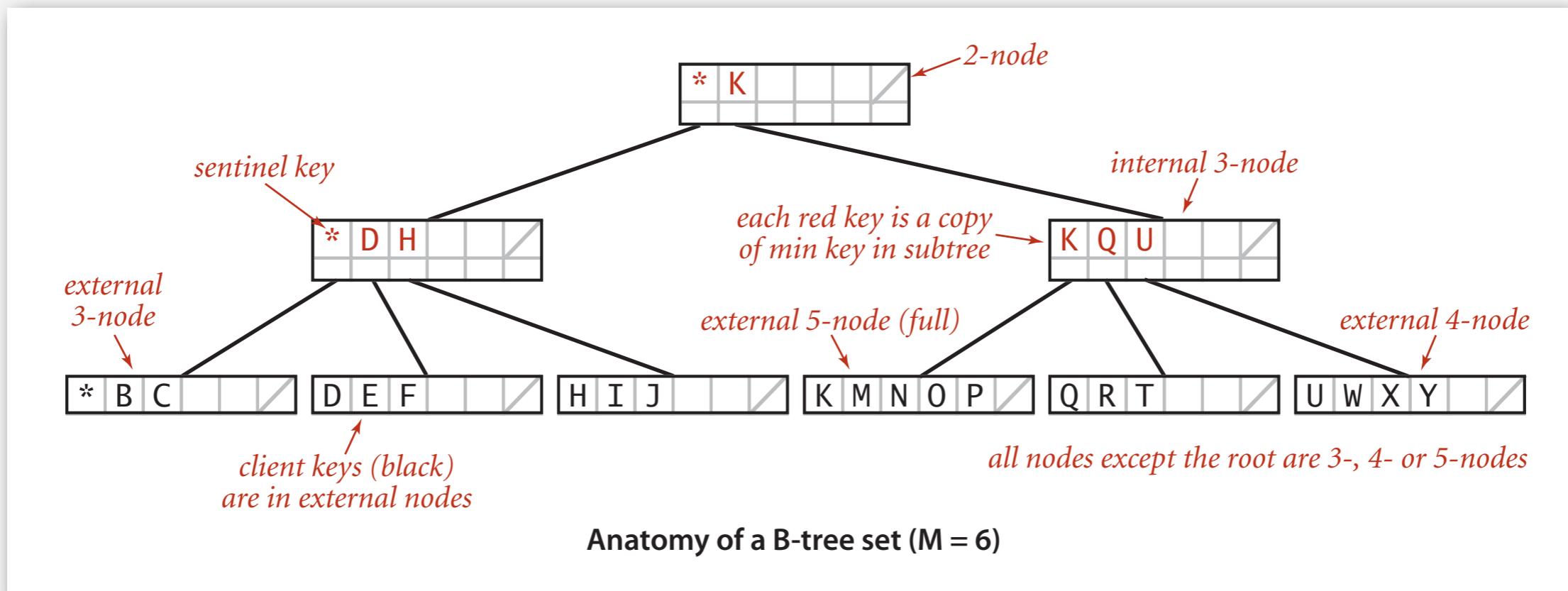
Goal. Access data using minimum number of probes.

B-trees (Bayer-McCreight, 1972)

B-tree. Generalize 2-3 trees by allowing up to $M - 1$ key-link pairs per node.

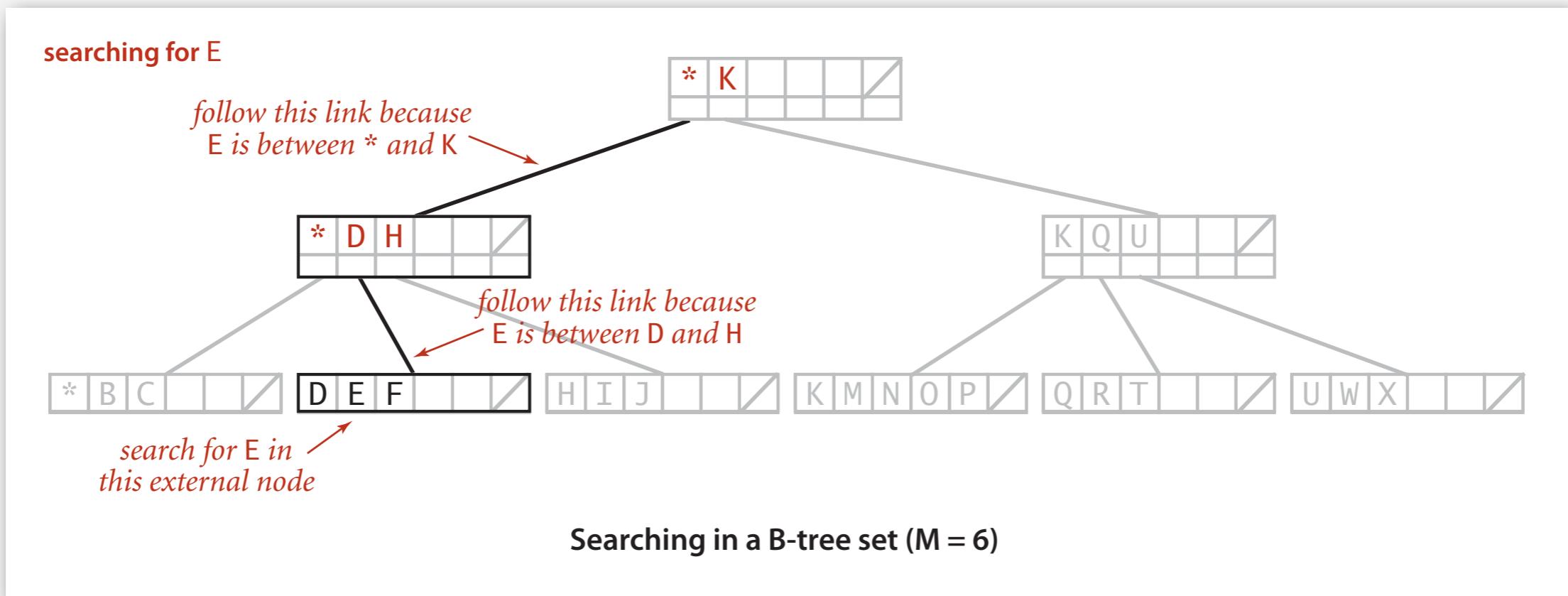
- At least 2 key-link pairs at root.
- At least $M / 2$ key-link pairs in other nodes.
- External nodes contain client keys.
- Internal nodes contain copies of keys to guide search.

choose M as large as possible so that M links fit in a page, e.g., $M = 1024$



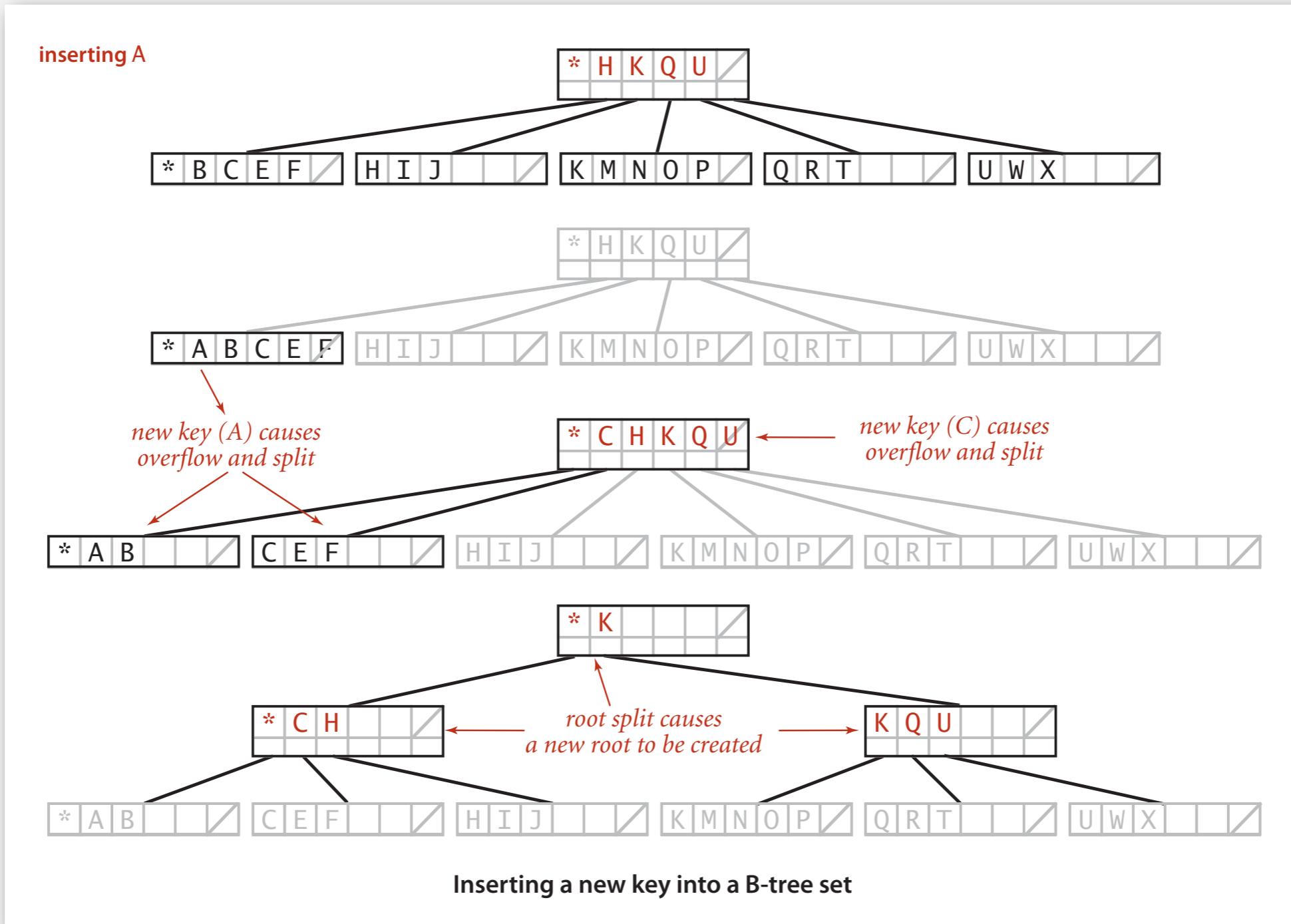
Searching in a B-tree

- Start at root.
- Find interval for search key and take corresponding link.
- Search terminates in external node.



Insertion in a B-tree

- Search for new key.
- Insert at bottom.
- Split nodes with M key-link pairs on the way up the tree.



Balance in B-tree

Proposition. A search or an insertion in a B-tree of order M with N keys requires between $\log_{M-1} N$ and $\log_{M/2} N$ probes.

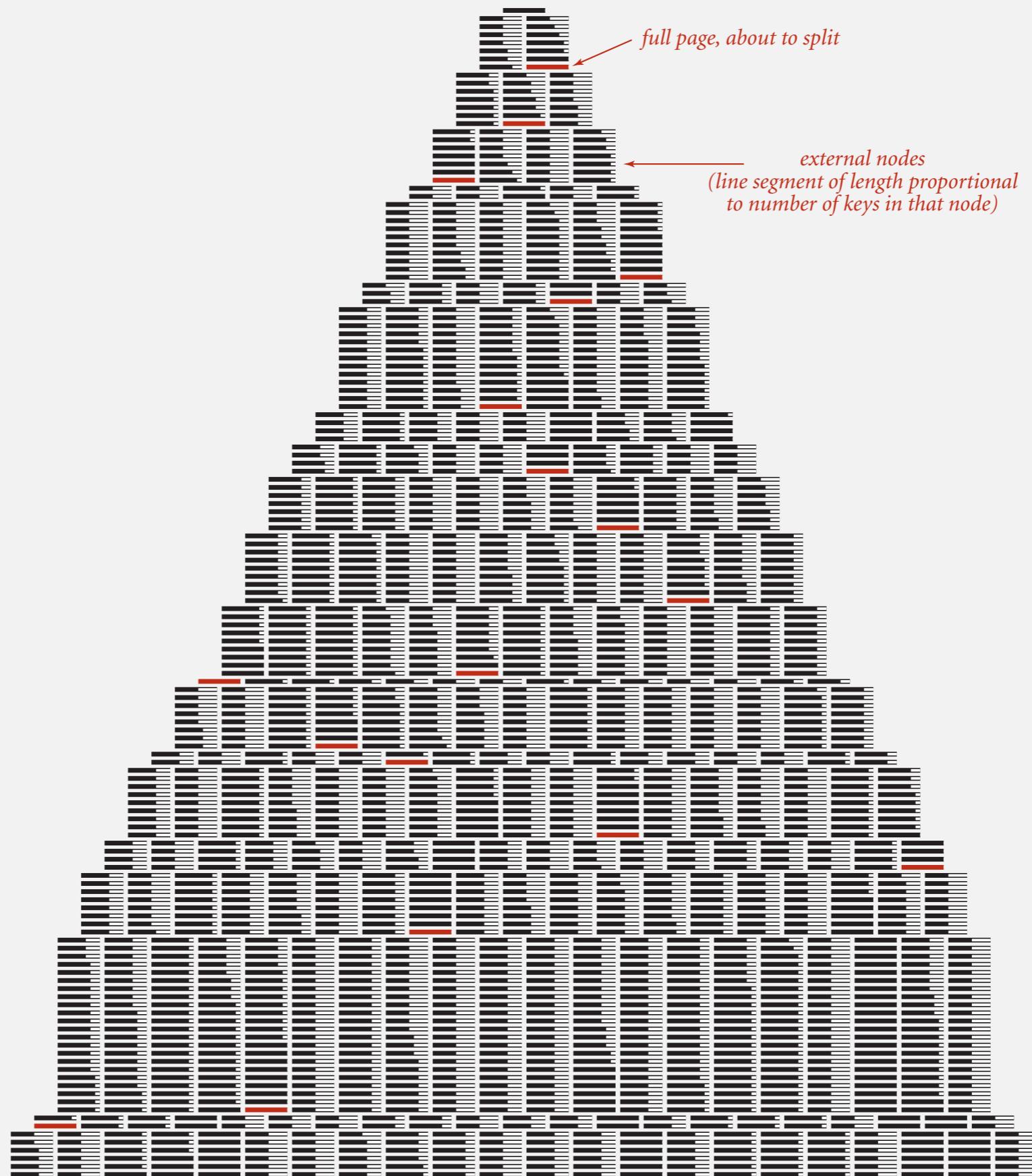
Pf. All internal nodes (besides root) have between $M/2$ and $M - 1$ links.

In practice. Number of probes is at most 4. \leftarrow

$$M = 1024; N = 62 \text{ billion}$$
$$\log_{M/2} N \leq 4$$

Optimization. Always keep root page in memory.

Building a large B tree



HASH TABLES

- ▶ hash functions
- ▶ separate chaining
- ▶ linear probing
- ▶ applications

ST implementations: summary

implementation	worst-case cost (after N inserts)			average-case cost (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	N/2	N/2	yes	<code>compareTo()</code>
BST	N	N	N	$1.38 \lg N$	$1.38 \lg N$?	yes	<code>compareTo()</code>
red-black BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.00 \lg N$	$1.00 \lg N$	$1.00 \lg N$	yes	<code>compareTo()</code>

Q. Can we do better?

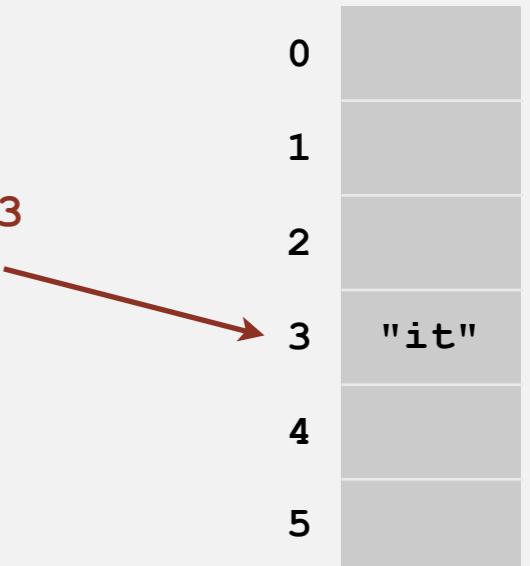
A. Yes, but with different access to the data.

Hashing: basic plan

Save items in a **key-indexed table** (index is a function of the key).

Hash function. Method for computing array index from key.

`hash("it") = 3`



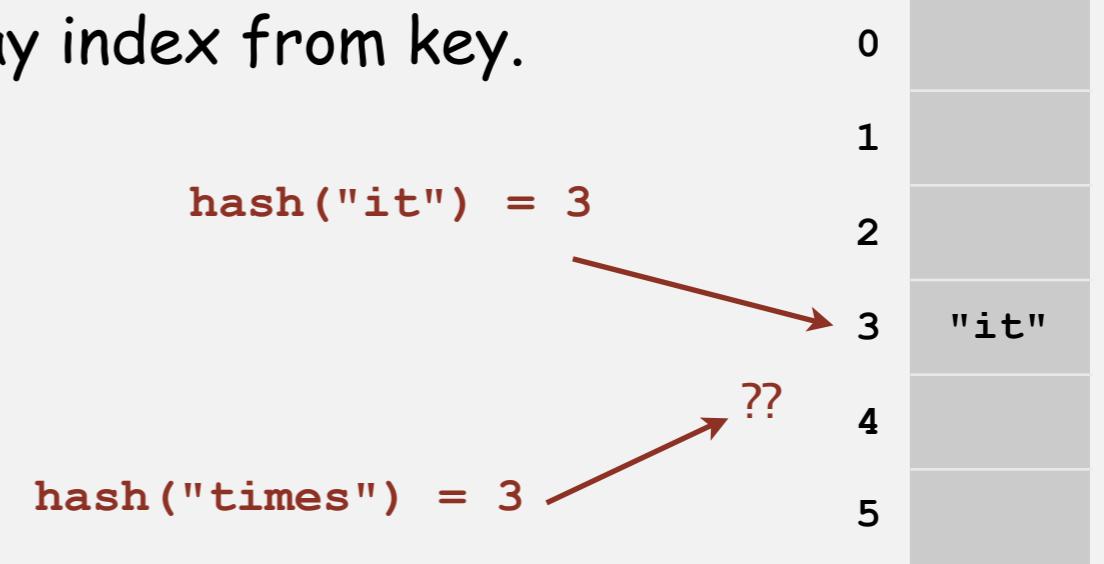
Issues.

- Computing the hash function.
- Equality test: Method for checking whether two keys are equal.

Hashing: basic plan

Save items in a **key-indexed table** (index is a function of the key).

Hash function. Method for computing array index from key.



Issues.

- Computing the hash function.
- Equality test: Method for checking whether two keys are equal.
- Collision resolution: Algorithm and data structure to handle two keys that hash to the same array index.

Classic space-time tradeoff.

- No space limitation: trivial hash function with key as index.
- No time limitation: trivial collision resolution with sequential search.
- Space and time limitations: hashing (the real world).

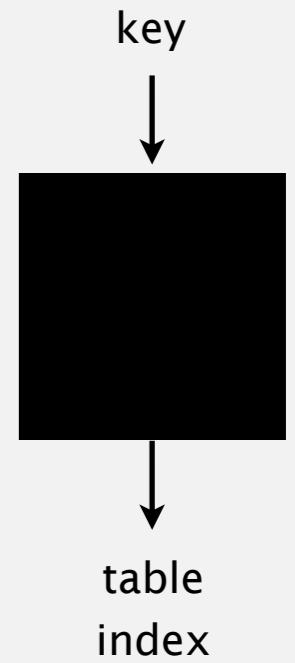
- ▶ hash functions
- ▶ separate chaining
- ▶ linear probing
- ▶ applications

Computing the hash function

Idealistic goal. Scramble the keys uniformly to produce a table index.

- Each table index equally likely for each key.
- Efficiently computable.
- Consistent --- equal keys must produce same hash

thoroughly researched problem,
still problematic in practical applications



Ex 1. Phone numbers.

- Bad: first three digits.
- Better: last three digits.

573 = California, 574 = Alaska
(assigned in chronological order within geographic region)

Ex 2. Social Security numbers.

- Bad: first three digits.
- Better: last three digits.

Practical challenge. Need different approach for each key type.

Hash code. An int between -2^{31} and $2^{31}-1$.

Hash function. An int between 0 and $M-1$ (for use as array index).

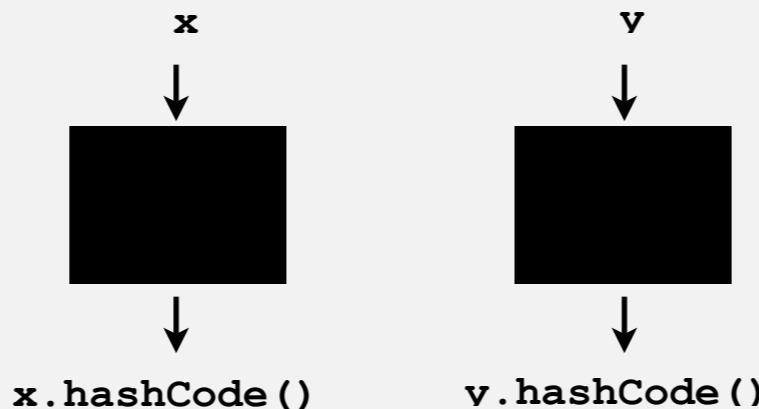
Basic rule. Need to use the whole key to compute hash code;

Java's hash code conventions

All Java classes inherit a method `hashCode()`, which returns a 32-bit `int`.

Requirement. If `x.equals(y)`, then `(x.hashCode() == y.hashCode())`.

Highly desirable. If `!x.equals(y)`, then `(x.hashCode() != y.hashCode())`.



Default implementation. Memory address of `x`.

Legal (but poor) implementation. Always return 17.

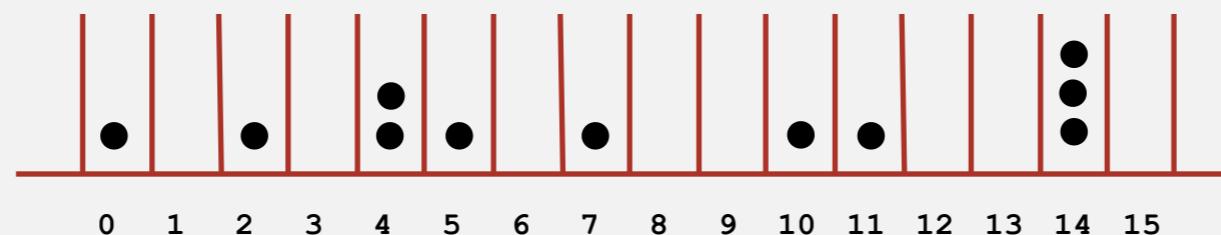
Customized implementations. `Integer`, `Double`, `String`, `File`, `URL`, `Date`, ...

User-defined types. Users are on their own.

Uniform hashing assumption

Uniform hashing assumption. Each key is equally likely to hash to an integer between 0 and $M - 1$. [Efficient to compute and consistent --- equal keys must produce same hash]

Bins and balls. Throw balls uniformly at random into M bins.



Birthday problem. Expect two balls in the same bin after $\sim \sqrt{\pi M / 2}$ tosses.

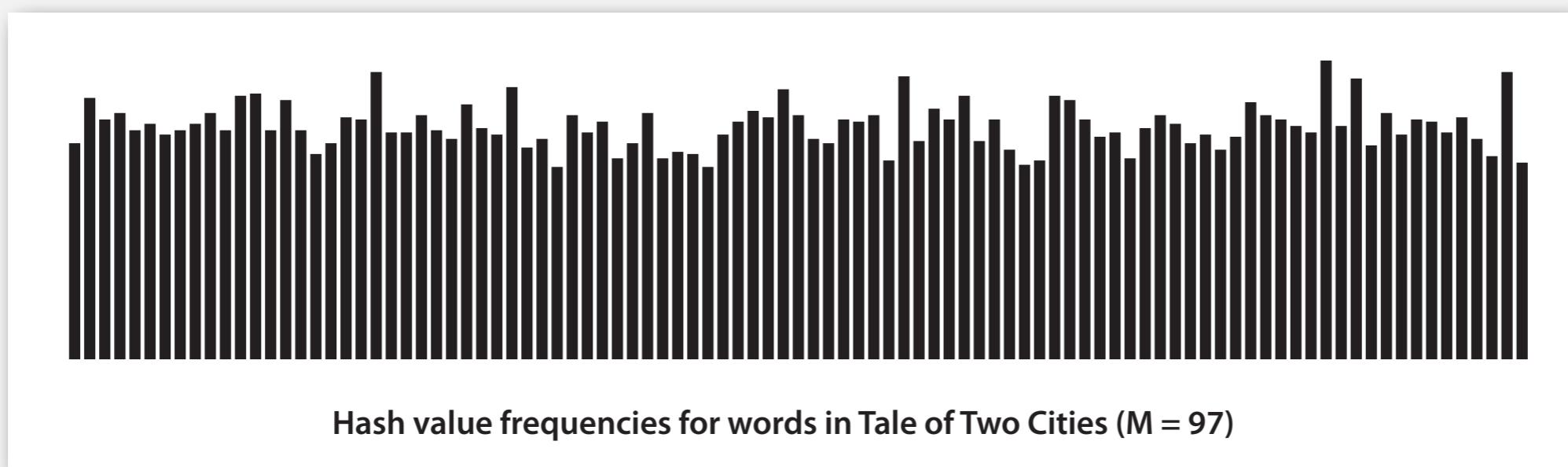
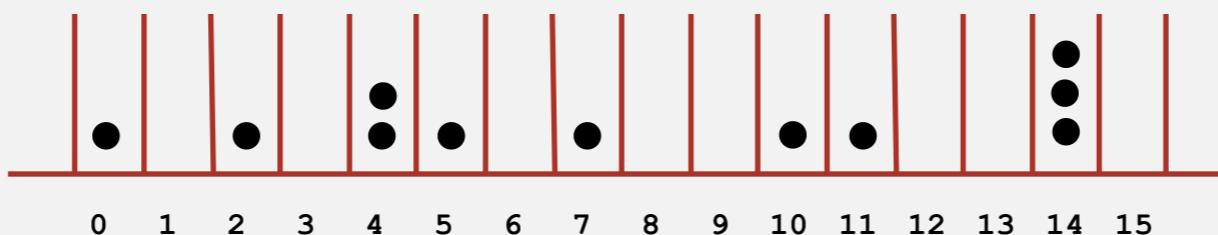
Coupon collector. Expect every bin has ≥ 1 ball after $\sim M \ln M$ tosses.

Load balancing. After M tosses, expect most loaded bin has $\Theta(\log M / \log \log M)$ balls.

Uniform hashing assumption

Uniform hashing assumption. Each key is equally likely to hash to an integer between 0 and $M - 1$.

Bins and balls. Throw balls uniformly at random into M bins.



Java's `String` data uniformly distribute the keys of Tale of Two Cities

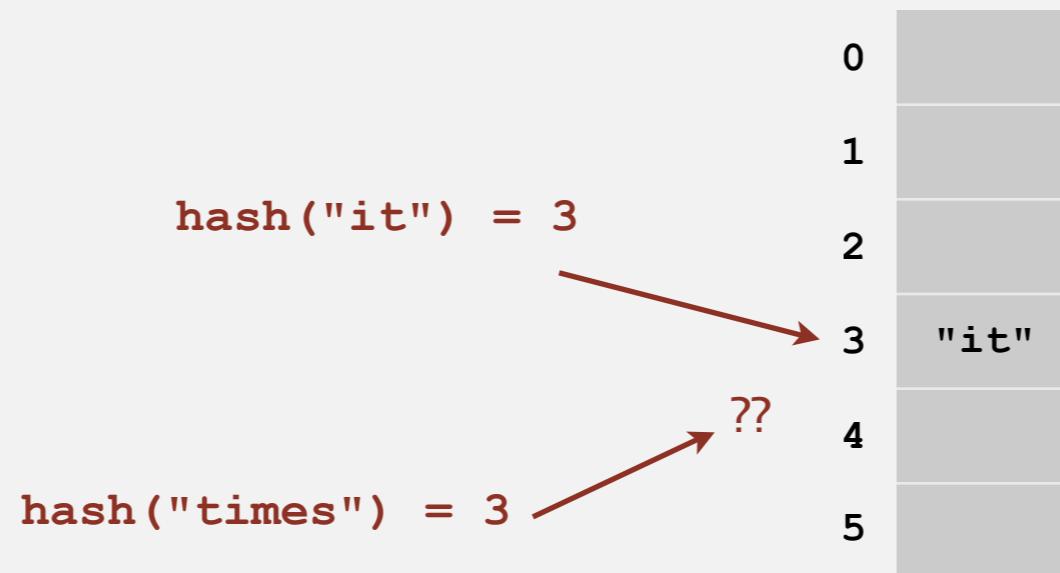
- ▶ hash functions
- ▶ **separate chaining**
- ▶ linear probing
- ▶ applications

Collisions

Collision. Two distinct keys hashing to same index.

- Birthday problem \Rightarrow can't avoid collisions unless you have a ridiculous (quadratic) amount of memory.
- Coupon collector + load balancing \Rightarrow collisions will be evenly distributed.

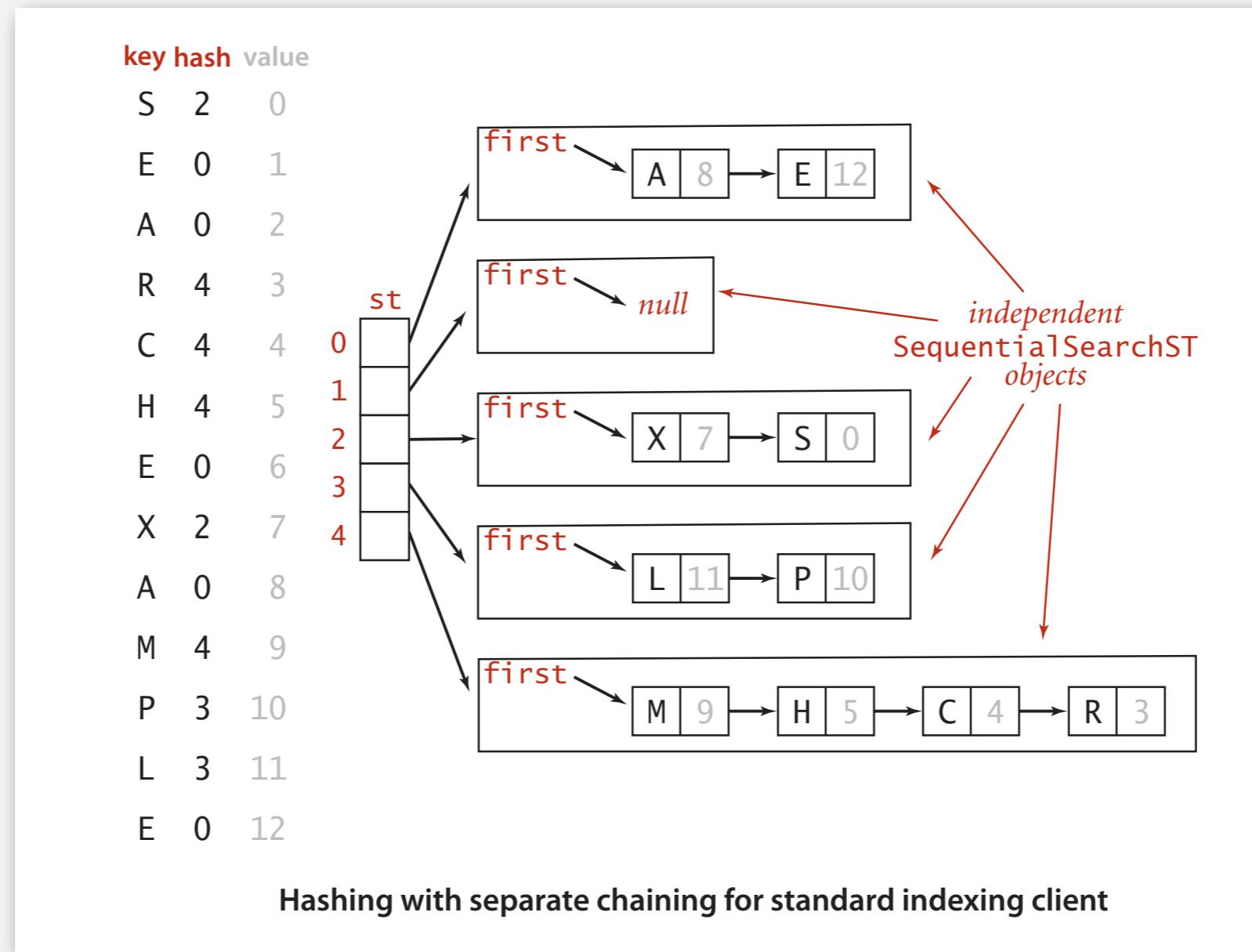
Challenge. Deal with collisions efficiently.



Separate chaining ST

Use an array of $M < N$ linked lists. [H. P. Luhn, IBM 1953]

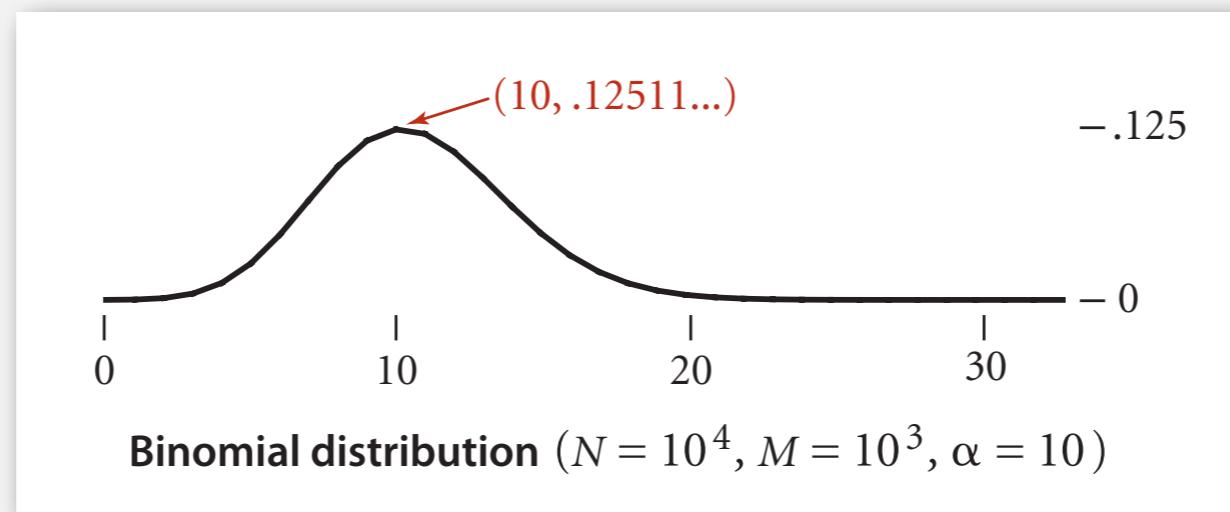
- Hash: map key to integer i between 0 and $M - 1$.
- Insert: put at front of i^{th} chain (if not already there).
- Search: only need to search i^{th} chain.



Analysis of separate chaining

Proposition. Under uniform hashing assumption, probability that the number of keys in a list is within a constant factor of N / M is extremely close to 1.

Pf sketch. Distribution of list size obeys a binomial distribution.



- Consequence.** Number of probes for search/insert is proportional to N / M .
- M too large \Rightarrow too many empty chains.
 - M too small \Rightarrow chains too long.
 - Typical choice: $M \sim N / 5 \Rightarrow$ constant-time ops.
- \uparrow
M times faster than
sequential search

equals() and hashCode()

ST implementations: summary

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	N/2	N/2	yes	<code>compareTo()</code>
BST	N	N	N	$1.38 \lg N$	$1.38 \lg N$?	yes	<code>compareTo()</code>
red-black tree	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.00 \lg N$	$1.00 \lg N$	$1.00 \lg N$	yes	<code>compareTo()</code>
separate chaining	$\lg N^*$	$\lg N^*$	$\lg N^*$	O(1)*	O(1)*	O(1)*	no	<code>equals()</code>

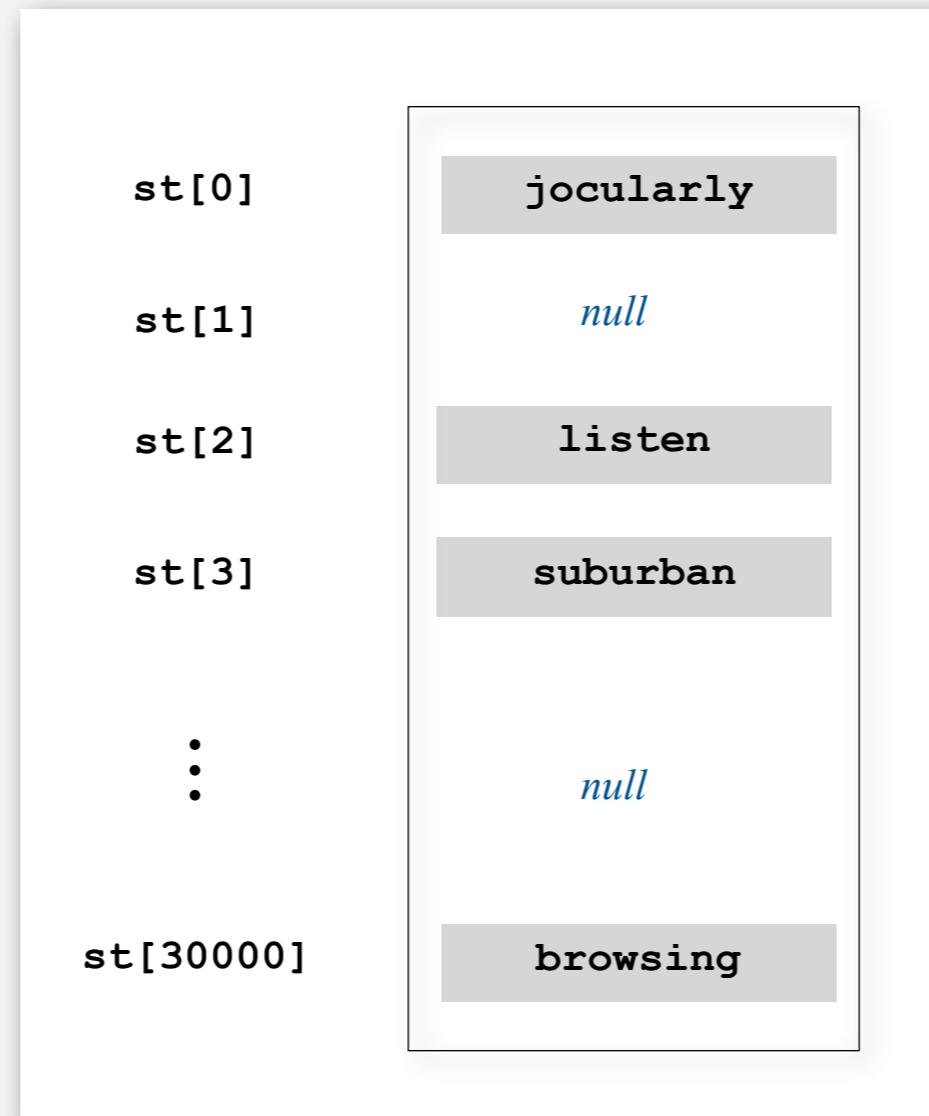
* under uniform hashing assumption

- ▶ hash functions
- ▶ separate chaining
- ▶ linear probing
- ▶ applications

Collision resolution: open addressing

Open addressing. [Amdahl-Boehme-Rochester-Samuel, IBM 1953]

When a new key collides, find next empty slot, and put it there.



linear probing ($M = 30001$, $N = 15000$)

Linear probing

Use an array of size $M > N$.

- Hash: map key to integer i between 0 and $M - 1$.
- Insert: put at table index i if free; if not try $i + 1, i + 2$, etc.
- Search: search table index i ; if occupied but no match, try $i + 1, i + 2$, etc.

-	-	-	S	H	-	-	A	C	E	R	-	-
0	1	2	3	4	5	6	7	8	9	10	11	12

-	-	-	S	H	-	-	A	C	E	R	I	-
0	1	2	3	4	5	6	7	8	9	10	11	12

insert I
hash(I) = 11

-	-	-	S	H	-	-	A	C	E	R	I	N
0	1	2	3	4	5	6	7	8	9	10	11	12

insert N
hash(N) = 8

Linear probing: trace of standard indexing client

key	hash	value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
S	6	0							S										
E	10	1							S				E						
A	4	2							0				E						
R	14	3					A	S				E			R				
C	5	4					A	C	S			E			R				
H	4	5					2	4	0	H			E		R				
E	10	6					A	C	S	H			E		R				
X	15	7					2	4	0	5			(6)		3				
A	4	8					A	C	S	H		E		R	X				
M	1	9					8	4	0	5		6			R	X			
P	14	10	P	M			A	C	S	H		E		R	X				
L	6	11	10	9			8	4	0	5		6			3	7			
E	10	12	P	M			A	C	S	H	L	E		R	X				
			10	9			8	4	0	5	11		(12)		3	7			

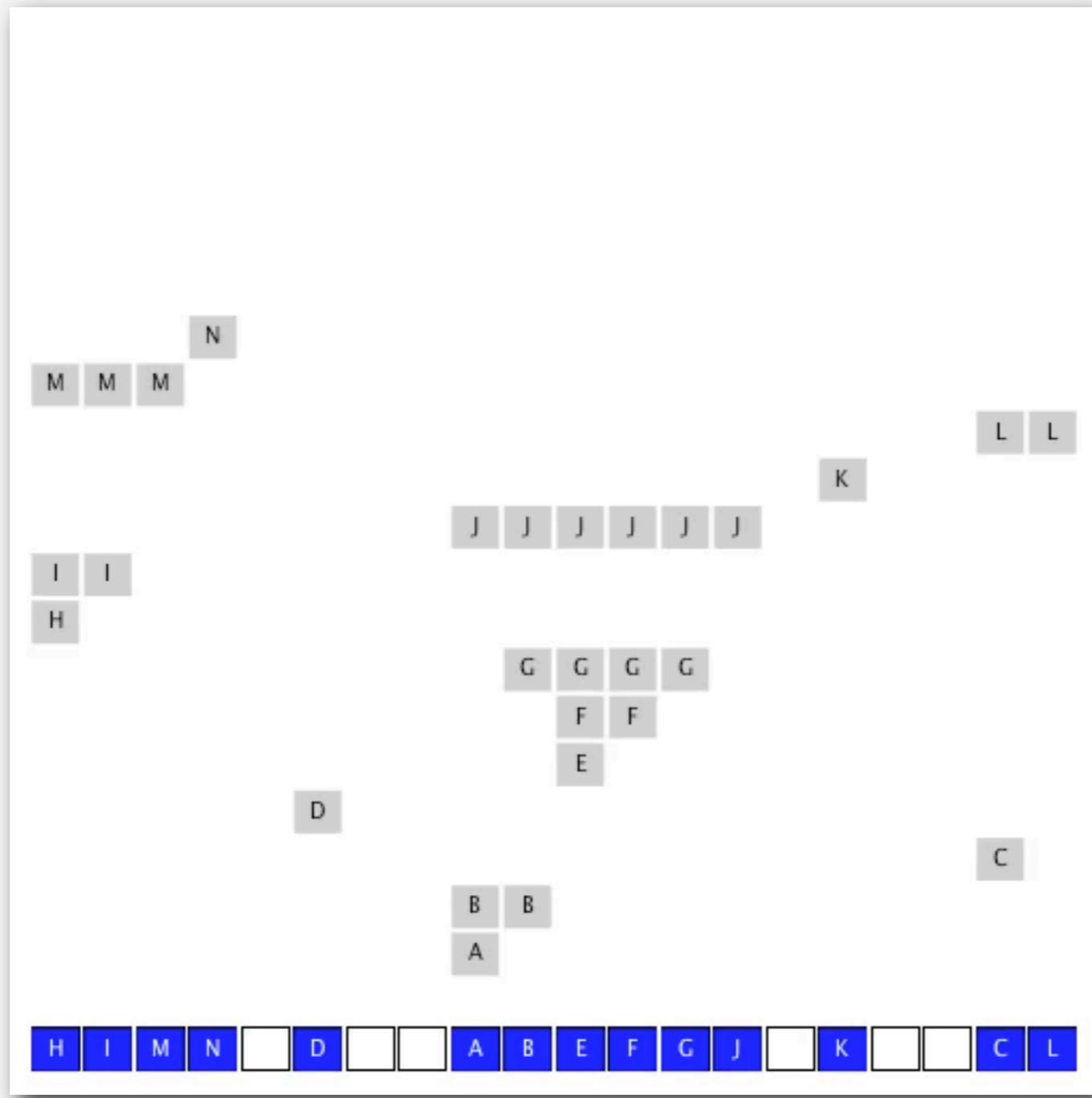
Annotations:

- entries in red are new* (points to A at index 4)
- entries in gray are untouched* (points to E at index 10)
- keys in black are probes* (points to A at index 4)
- probe sequence wraps to 0* (points to P at index 10)
- keys[]* (points to row 12)
- vals[]* (points to row 12)

Clustering

Cluster. A contiguous block of items.

Observation. New keys likely to hash into middle of big clusters.

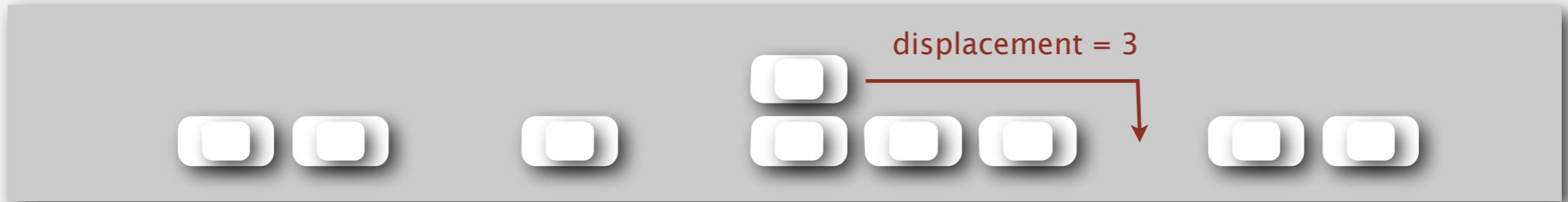


Knuth's parking problem

Model. Cars arrive at one-way street with M parking spaces.

Each desires a random space i : if space i is taken, try $i + 1, i + 2$, etc.

Q. What is mean displacement of a car?



Half-full. With $M/2$ cars, mean displacement is $\sim 3/2$.

Full. With M cars, mean displacement is $\sim \sqrt{\pi M / 8}$

Analysis of linear probing

Proposition. Under uniform hashing assumption, the average number of probes in a hash table of size M that contains $N = \alpha M$ keys is:

$$\sim \frac{1}{2} \left(1 + \frac{1}{1 - \alpha} \right)$$

search hit

$$\sim \frac{1}{2} \left(1 + \frac{1}{(1 - \alpha)^2} \right)$$

search miss / insert

Pf. [Knuth 1962] A landmark in analysis of algorithms.

Parameters.

- M too large \Rightarrow too many empty array entries.
- M too small \Rightarrow search time blows up.
- Typical choice: $\alpha = N/M \sim 1/2$.

probes for search hit is about 3/2
probes for search miss is about 5/2

ST implementations: summary

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	N/2	N/2	yes	<code>compareTo()</code>
BST	N	N	N	$1.38 \lg N$	$1.38 \lg N$?	yes	<code>compareTo()</code>
red-black tree	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.00 \lg N$	$1.00 \lg N$	$1.00 \lg N$	yes	<code>compareTo()</code>
separate chaining	$\lg N^*$	$\lg N^*$	$\lg N^*$	O(1)*	O(1)*	O(1)*	no	<code>equals()</code>
linear probing	$\lg N^*$	$\lg N^*$	$\lg N^*$	O(1)*	O(1)*	O(1)*	no	<code>equals()</code>

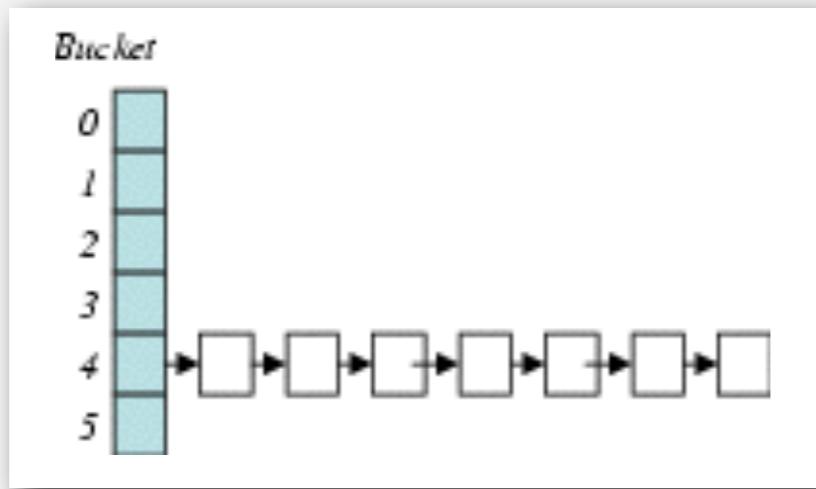
* under uniform hashing assumption

War story: algorithmic complexity attacks

Q. Is the uniform hashing assumption important in practice?

A. Obvious situations: aircraft control, nuclear reactor, pacemaker.

A. Surprising situations: denial-of-service attacks.



malicious adversary learns your hash function
(e.g., by reading Java API) and causes a big pile-up
in single slot that grinds performance to a halt

Real-world exploits. [Crosby-Wallach 2003]

- Bro server: send carefully chosen packets to DOS the server, using less bandwidth than a dial-up modem.
- Perl 5.8.0: insert carefully chosen strings into associative array.
- Linux 2.4.20 kernel: save files with carefully chosen names.

Separate chaining vs. linear probing

Separate chaining.

- Easier to implement delete.
- Performance degrades gracefully.
- Clustering less sensitive to poorly-designed hash function.

Linear probing.

- Less wasted space.
- Better cache performance.

Hashing: variations on the theme

Many improved versions have been studied.

Two-probe hashing. (separate-chaining variant)

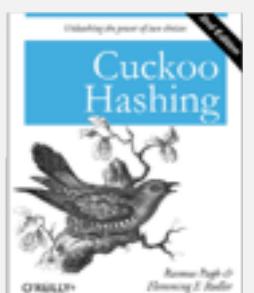
- Hash to two positions, insert key in shorter of the two chains.
- Reduces expected length of the longest chain to $\log \log N$.

Double hashing. (linear-probing variant)

- Use linear probing, but skip a variable amount, not just 1 each time.
- Effectively eliminates clustering.
- Can allow table to become nearly full.
- Difficult to implement delete.

Cuckoo hashing. (linear-probing variant)

- Hash key to two positions; insert key into either position; if occupied, reinsert displaced key into its alternative position (and recur).
- Constant worst case time for get.



Hashing vs. balanced search trees

Hashing.

- Simpler to code.
- No effective alternative for unordered keys.
- Faster for simple keys (a few arithmetic ops versus $\log N$ compares).
- Better system support in Java for strings (e.g., cached hash code).

Balanced search trees.

- Stronger performance guarantee.
- Support for ordered ST operations.
- Easier to implement `compareTo()` correctly than `equals()` and `hashCode()`.

- ▶ hash functions
- ▶ separate chaining
- ▶ linear probing
- ▶ applications

Mathematical set. A collection of distinct keys.

```
public class SET<Key extends Comparable<Key>>
```

SET()	<i>create an empty set</i>
void add(Key key)	<i>add the key to the set</i>
boolean contains(Key key)	<i>is the key in the set?</i>
void remove(Key key)	<i>remove the key from the set</i>
int size()	<i>return the number of keys in the set</i>
Iterator<Key> iterator()	<i>iterator through keys in the set</i>

Q. How to implement?

Exception filter

- Read in a list of words from one file.
- Print out all words from standard input that are { in, not in } the list.

```
% more list.txt  
was it the of  
  
% java WhiteList list.txt < tinyTale.txt  
it was the of it was the of  
  
% java BlackList list.txt < tinyTale.txt  
best times worst times  
age wisdom age foolishness  
epoch belief epoch incredulity  
season light season darkness  
spring hope winter despair
```



list of exceptional words

Exception filter applications

- Read in a list of words from one file.
- Print out all words from standard input that are { in, not in } the list.

application	purpose	key	in list
spell checker	identify misspelled words	word	dictionary words
browser	mark visited pages	URL	visited pages
parental controls	block sites	URL	bad sites
chess	detect draw	board	positions
spam filter	eliminate spam	IP address	spam addresses
credit cards	check for stolen cards	number	stolen cards

Dictionary lookup

Command-line arguments.

- A comma-separated value (CSV) file.
- Key field.
- Value field.

Ex 1. DNS lookup.

```
% java LookupCSV ip.csv 0 1  
adobe.com  
192.150.18.60  
www.princeton.edu  
128.112.128.15  
ebay.edu  
Not found  
  
% java LookupCSV ip.csv 1 0  
128.112.128.15  
www.princeton.edu  
999.999.999.99  
Not found
```

URL is key IP is value

IP is key URL is value

```
% more ip.csv  
www.princeton.edu,128.112.128.15  
www.cs.princeton.edu,128.112.136.35  
www.math.princeton.edu,128.112.18.11  
www.cs.harvard.edu,140.247.50.127  
www.harvard.edu,128.103.60.24  
www.yale.edu,130.132.51.8  
www.econ.yale.edu,128.36.236.74  
www.cs.yale.edu,128.36.229.30  
espn.com,199.181.135.201  
yahoo.com,66.94.234.13  
msn.com,207.68.172.246  
google.com,64.233.167.99  
baidu.com,202.108.22.33  
yahoo.co.jp,202.93.91.141  
sina.com.cn,202.108.33.32  
ebay.com,66.135.192.87  
adobe.com,192.150.18.60  
163.com,220.181.29.154  
passport.net,65.54.179.226  
tom.com,61.135.158.237  
nate.com,203.226.253.11  
cnn.com,64.236.16.20  
daum.net,211.115.77.211  
blogger.com,66.102.15.100  
fastclick.com,205.180.86.4  
wikipedia.org,66.230.200.100  
rakuten.co.jp,202.72.51.22  
...
```

Dictionary lookup

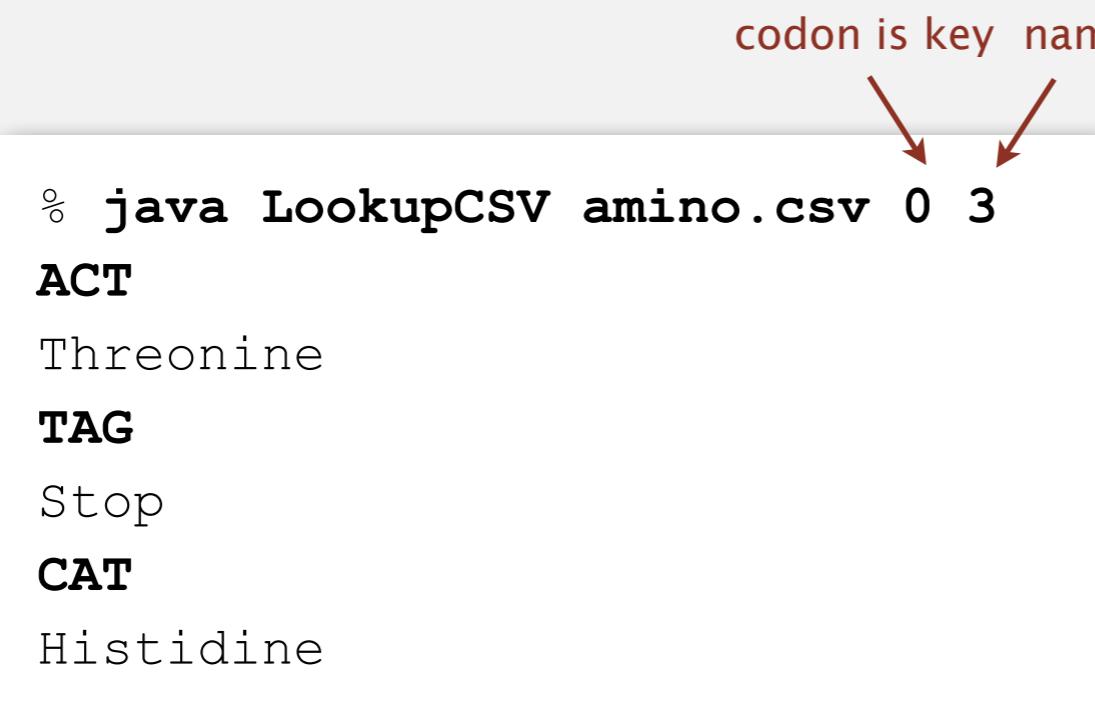
Command-line arguments.

- A comma-separated value (CSV) file.
- Key field.
- Value field.

Ex 2. Amino acids.

```
% java LookupCSV amino.csv 0 3
ACT
Threonine
TAG
Stop
CAT
Histidine
```

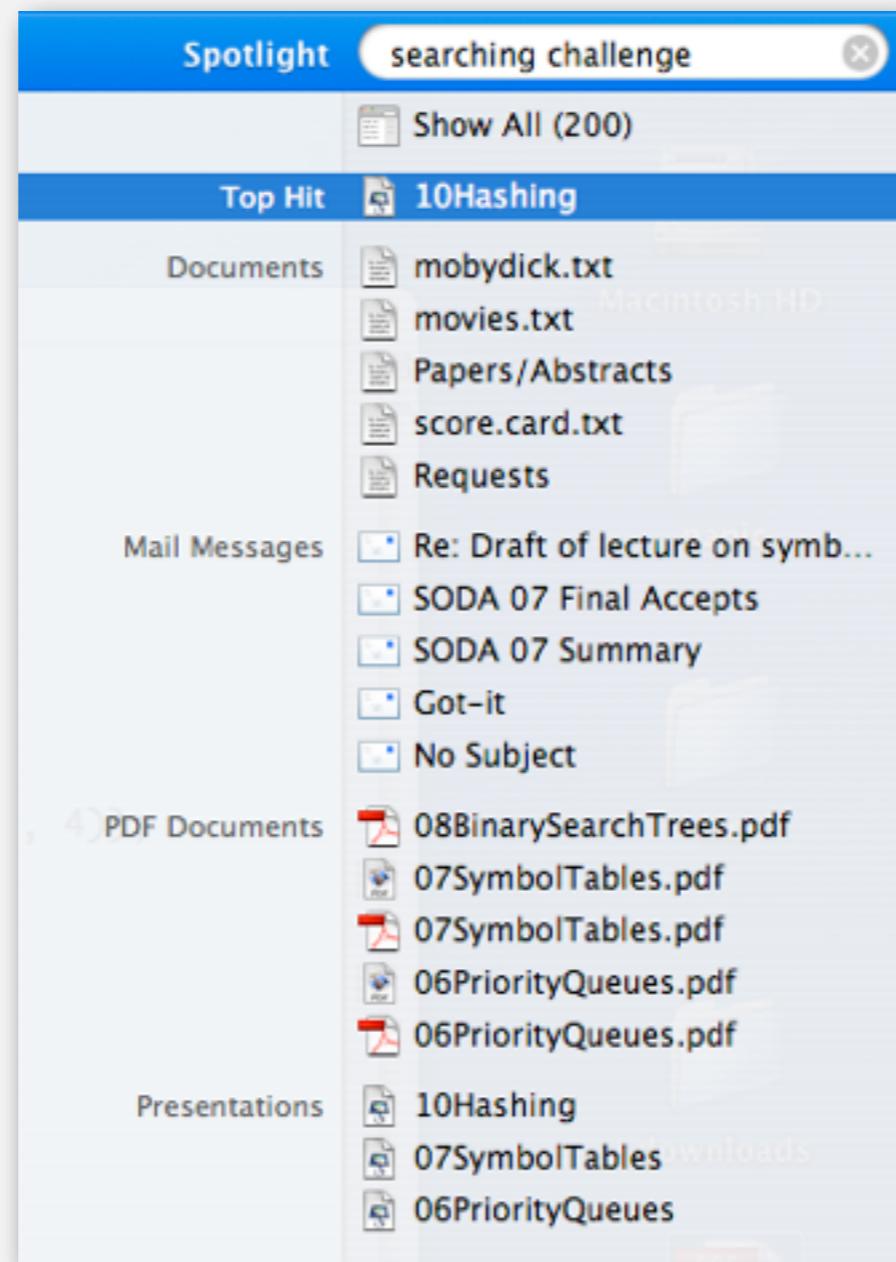
codon is key name is value



```
% more amino.csv
TTT,Phe,F,Phenylalanine
TTC,Phe,F,Phenylalanine
TTA,Leu,L,Leucine
TTG,Leu,L,Leucine
TCT,Ser,S,Serine
TCC,Ser,S,Serine
TCA,Ser,S,Serine
TCG,Ser,S,Serine
TAT,Tyr,Y,Tyrosine
TAC,Tyr,Y,Tyrosine
TAA,Stop,Stop,Stop
TAG,Stop,Stop,Stop
TGT,Cys,C,Cysteine
TGC,Cys,C,Cysteine
TGA,Stop,Stop,Stop
TGG,Trp,W,Tryptophan
CTT,Leu,L,Leucine
CTC,Leu,L,Leucine
CTA,Leu,L,Leucine
CTG,Leu,L,Leucine
CCT,Pro,P,Proline
CCC,Pro,P,Proline
CCA,Pro,P,Proline
CCG,Pro,P,Proline
CAT,His,H,Histidine
CAC,His,H,Histidine
CAA,Gln,Q,Glutamine
CAG,Gln,Q,Glutamine
CGT,Arg,R,Arginine
CGC,Arg,R,Arginine
...
```

File indexing

Goal. Index a PC (or the web).



File indexing

Goal. Given a list of files specified as command-line arguments, create an index so that can efficiently find all files containing a given query string.

```
% ls *.txt  
aesop.txt magna.txt moby.txt  
sawyer.txt tale.txt  
  
% java FileIndex *.txt  
freedom  
magna.txt moby.txt tale.txt  
  
whale  
moby.txt  
  
lamb  
sawyer.txt aesop.txt
```

```
% ls *.java  
  
% java FileIndex *.java  
BlackList.java Concordance.java  
DeDup.java FileIndex.java ST.java  
SET.java WhiteList.java  
  
import  
FileIndex.java SET.java ST.java  
  
Comparator  
null
```

Solution. Key = query string; value = set of files containing that string.

File indexing

```
public class FileIndex
{
    public static void main(String[] args)
    {
        ST<String, SET<File>> st = new ST<String, SET<File>>(); ← symbol table

        for (String filename : args) {
            File file = new File(filename);
            In in = new In(file);
            while (!in.isEmpty())
            {
                String word = in.readString();
                if (!st.contains(word))
                    st.put(s, new SET<File>());
                SET<File> set = st.get(key);
                set.add(file);
            }
        }

        while (!StdIn.isEmpty())
        {
            String query = StdIn.readString();
            StdOut.println(st.get(query));
        }
    }
}
```

list of file names from command line

for each word in file, add file to corresponding set

process queries

Goal. Index for an e-book.

Index

- Abstract data type (ADT), 127-195
 - abstract classes, 163
 - classes, 129-136
 - collections of items, 137-139
 - creating, 157-164
 - defined, 128
 - duplicate items, 173-176
 - equivalence-relations, 159-162
 - FIFO queues, 165-171
 - first-class, 177-186
 - generic operations, 273
 - index items, 177
 - insert/remove* operations, 138-139
 - modular programming, 135
 - polynomial, 188-192
 - priority queues, 375-376
 - pushdown stack, 138-156
 - stubs, 135
 - symbol table, 497-506
- ADT interfaces
 - array (`myArray`), 274
 - complex number (`Complex`), 181
 - existence table (ET), 663
 - full priority queue (`PQfull`), 397
 - indirect priority queue (`PQi`), 403
 - item (`myItem`), 273, 498
 - key (`myKey`), 498
 - polynomial (`Poly`), 189
 - point (`Point`), 134
 - priority queue (`PQ`), 375
 - queue of int (`intQueue`), 166
- stack of int (`intStack`), 140
- symbol table (ST), 503
 - text index (TI), 525
 - union-find (UF), 159
- Abstract in-place merging, 351-353
- Abstract operation, 10
- Access control state, 131
- Actual data, 31
- Adapter class, 155-157
- Adaptive sort, 268
- Address, 84-85
- Adjacency list, 120-123
 - depth-first search, 251-256
- Adjacency matrix, 120-122
- Ajtai, M., 464
- Algorithm, 4-6, 27-64
 - abstract operations, 10, 31, 34-35
 - analysis of, 6
 - average-/worst-case performance, 35, 60-62
 - big-Oh notation, 44-47
 - binary search, 56-59
 - computational complexity, 62-64
 - efficiency, 6, 30, 32
 - empirical analysis, 30-32, 58
 - exponential-time, 219
 - implementation, 28-30
 - logarithm function, 40-43
 - mathematical analysis, 33-36, 58
 - primary parameter, 36
 - probabilistic, 331
 - recurrences, 49-52, 57
 - recursive, 198
 - running time, 34-40
 - search, 53-56, 498
 - steps in, 22-23
- See also* Randomized algorithm
- Amortization approach, 557, 627
- Arithmetic operator, 177-179, 188, 191
- Array, 12, 83
 - binary search, 57
 - dynamic allocation, 87
- and linked lists, 92, 94-95
- merging, 349-350
- multidimensional, 117-118
- references, 86-87, 89
- sorting, 265-267, 273-276
- and strings, 119
- two-dimensional, 117-118, 120-124
- vectors, 87
- visualizations, 295
- See also* Index, array
- Array representation
 - binary tree, 381
 - FIFO queue, 168-169
 - linked lists, 110
 - polynomial ADT, 191-192
 - priority queue, 377-378, 403, 406
 - pushdown stack, 148-150
 - random queue, 170
 - symbol table, 508, 511-512, 521
- Asymptotic expression, 45-46
- Average deviation, 80-81
- Average-case performance, 35, 60-61
- AVL tree, 583
- B tree, 584, 692-704
 - external/internal pages, 695
 - 4-5-6-7-8 tree, 693-704
- Markov chain, 701
 - remove, 701-703
 - search/insert, 697-701
 - select/sort, 701
- Balanced tree, 238, 555-598
 - B tree, 584
 - bottom-up, 576, 584-585
 - height-balanced, 583
 - indexed sequential access, 690-692
 - performance, 575-576, 581-582, 595-598
 - randomized, 559-564
 - red-black, 577-585
 - skip lists, 587-594
 - splay, 566-571