

# Midterm Exam Logistics

---

The midterm exam is on Thursday from 10:30-11:50 (strict end time). It will be held in 4 locations. Go to a location based on the first letter of your last name:

Chem 102: A-L

Fagin 116: M-Q

Fagin 118: R-T

Fagin 218: U-Z

There will be a review session TONIGHT from 8pm-10pm Berger Auditorium in Skirkanich Hall.

If you have 3 exams scheduled for Thursday, I will allow you to do a makeup exam on Friday from 11-12:30 in my office (Levine 506). No other makeup times will be offered. If you intend to take the makeup exam, email me TODAY to make the request, and list what other courses you're taking that have exams.



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

## 3.1 SYMBOL TABLES

---

- ▶ *API*
- ▶ *elementary implementations*
- ▶ *ordered operations*



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

## 3.1 SYMBOL TABLES

---

- ▶ *API*
- ▶ *elementary implementations*
- ▶ *ordered operations*

# Why are telephone books obsolete?

## Unsupported operations.

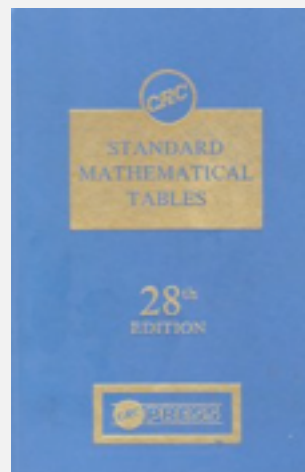
- Change the number associated with a given name.
- Add a new name, associated with a given number.
- Remove a given name and associated number.



key = term, value = article



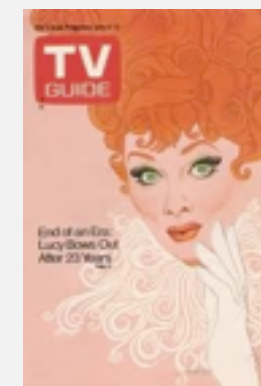
key = name  
value = phone number



key = function name and input  
value = function output



key = word, value = definition



key = time and channel  
value = TV show

# Symbol tables

---

## Key-value pair abstraction.

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

## Ex. DNS lookup.

- Insert domain name with specified IP address.
- Given domain name, find corresponding IP address.

domain name	IP address
<u>www.cis.upenn.edu</u>	158.130.69.163
<u>www.upenn.edu</u>	23.15.7.160
<u>www.princeton.edu</u>	140.180.223.42
www.harvard.edu	104.16.154.6
<u>www.fivethirtyeight.com</u>	192.0.79.32

↑  
key

↑  
value

# Symbol table applications

---

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

# Symbol tables: context

---

Also known as: maps, dictionaries, associative arrays.

Generalizes arrays. Keys need not be between 0 and  $N - 1$ .

Language support.

- External libraries: C, VisualBasic, Standard ML, bash, ...
- Built-in libraries: Java, C#, C++, Scala, ...
- Built-in to language: Awk, Perl, PHP, Tcl, JavaScript, Python, Ruby, Lua.

every array is an  
associative array

every object is an  
associative array

table is the only  
"primitive" data structure

```
has_nice_syntax_for_associative_arrays["Python"] = True
has_nice_syntax_for_associative_arrays["Java"]   = False
```

**legal Python code**



# Basic symbol table API

---

**Associative array abstraction.** Associate one value with each key.

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

```
    ST()
```

*create an empty symbol table*

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

*put key-value pair into the table* ← **a[key] = val;**

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

*value paired with key* ← **a[key]**

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

*is there a value paired with key?*

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

*all the keys in the table*

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

*remove key (and its value) from table*

```
    boolean isEmpty()
```

*is the table empty?*

```
    int size()
```

*number of key-value pairs in the table*



# Conventions

---

- Values are not null. ← `java.util.Map` allows null values
- Method `put()` overwrites old value with new value.
- Method `get()` returns null if key not present.

*“ Careless use of null can cause a staggering variety of bugs. Studying the Google code base, we found that something like 95% of collections weren't supposed to have any null values in them, and having those fail fast rather than silently accept null would have been helpful to developers. ”*



<https://code.google.com/p/guava-libraries/wiki/UsingAndAvoidingNullExplained>

# Conventions

---

- Values are not null. ← `java.util.Map` allows null values
- 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.


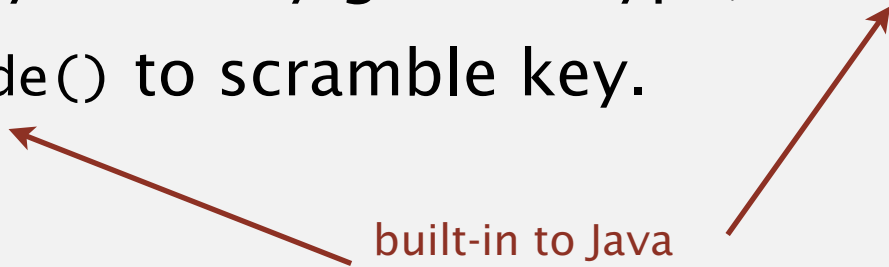
**Key type: several natural assumptions.**

- 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; use hashCode() to scramble key.

specify Comparable in API.



built-in to Java  
(stay tuned)



**Best practices.** Use immutable types for symbol table keys.

- Immutable in Java: Integer, Double, String, java.io.File, ...
- Mutable in Java: StringBuilder, java.net.URL, arrays, ...

# Equality test

---

All Java classes inherit a method `equals()`.

**Java requirements.** For any references `x`, `y` and `z`:

- Reflexive: `x.equals(x)` is true.
- Symmetric: `x.equals(y)` iff `y.equals(x)`.
- Transitive: if `x.equals(y)` and `y.equals(z)`, then `x.equals(z)`.
- Non-null: `x.equals(null)` is false.

} equivalence  
relation

do `x` and `y` refer to  
the same object?

**Default implementation.** `(x == y)`

**Customized implementations.** `Integer`, `Double`, `String`, `java.io.File`, ...

**User-defined implementations.** Some care needed.

# Implementing equals for user-defined types

---


Seems easy.

```
public      class Date implements Comparable<Date>
{
    private final int month;
    private final int day;
    private final int year;
    ...

    public boolean equals(Date that)
    {

        if (this.day    != that.day ) return false;
        if (this.month != that.month) return false;
        if (this.year   != that.year ) return false;
        return true;
    }
}
```

check that all significant  
fields are the same



# Implementing equals for user-defined types

Seems easy, but requires some care.

typically unsafe to use equals() with inheritance  
(would violate symmetry)

```
public final class Date implements Comparable<Date>
{
    private final int month;
    private final int day;
    private final int year;
    ...

    public boolean equals(Object y)
    {
        if (y == this) return true;

        if (y == null) return false;

        if (y.getClass() != this.getClass())
            return false;

        Date that = (Date) y;
        if (this.day != that.day ) return false;
        if (this.month != that.month) return false;
        if (this.year != that.year ) return false;
        return true;
    }
}
```

must be Object.  
Why? Experts still debate.

optimize for true object equality

check for null

objects must be in the same class  
(religion: getClass() vs. instanceof)


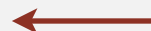

cast is guaranteed to succeed

check that all significant  
fields are the same

# Equals design

---

## "Standard" recipe for user-defined types.

- Optimization for reference equality.
- Check against `null`.
- Check that two objects are of the same type; cast.
- Compare each significant field:
  - if field is a primitive type, use `==`  but use `Double.compare()` with `double` (to deal with `-0.0` and `NaN`)
  - if field is an object, use `equals()`  apply rule recursively
  - if field is an array, apply to each entry  can use `Arrays.deepEquals(a, b)` but not `a.equals(b)`

## Best practices.

- No need to use calculated fields that depend on other fields.
- Compare fields mostly likely to differ first.
- Make `compareTo()` consistent with `equals()`.

 `x.equals(y)` if and only if `(x.compareTo(y) == 0)`



# 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 3 < tinyTale.txt
the 10
```

← tiny example  
(60 words, 20 distinct)

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

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

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

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

# Frequency counter implementation

---

```
public class FrequencyCounter
{
    public static void main(String[] args)
    {
        int minlen = Integer.parseInt(args[0]);

        ST<String, Integer> st = new ST<String, Integer>();
        while (!StdIn.isEmpty())
        {
            String word = StdIn.readString();
            if (word.length() < minlen) continue;
            if (!st.contains(word)) st.put(word, 1);
            else
                st.put(word, st.get(word) + 1);

            String max = "";
            st.put(max, 0);
            for (String word : st.keys())
                if (st.get(word) > st.get(max))
                    max = word;
            StdOut.println(max + " " + st.get(max));
        }
    }
}
```

← create ST

← ignore short strings

← read string and update frequency

print a string with max frequency



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

## 3.1 SYMBOL TABLES

---

▶ *API*

▶ *elementary implementations*

▶ *ordered operations*

# Sequential search in a linked list

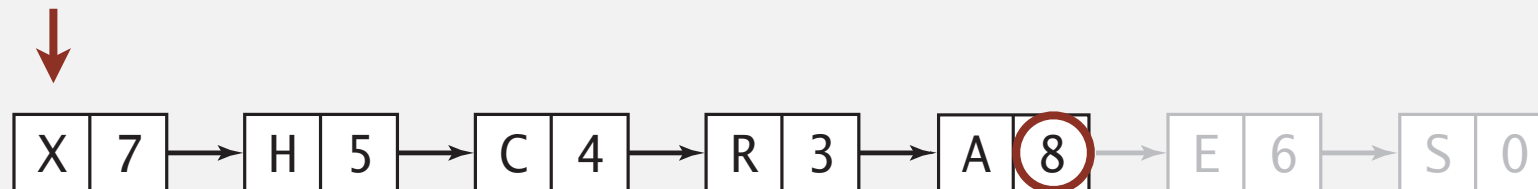
---

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

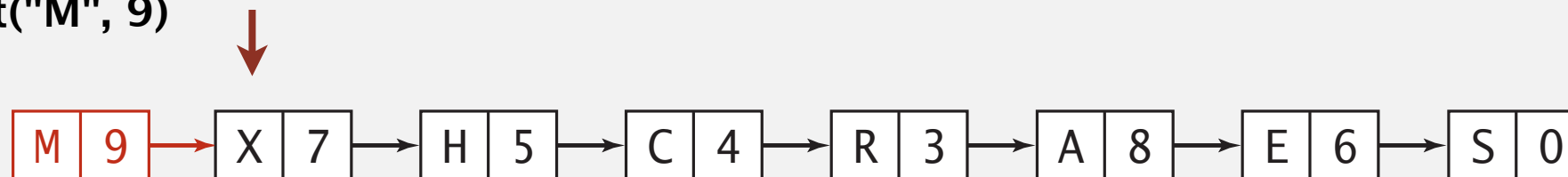
**Search.** Scan through all keys until find a match.

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

**get("A")**



**put("M", 9)**



# Elementary ST implementations: summary

---

implementation	guarantee		average case		operations on keys
	search	insert	search hit	insert	
<b>sequential search (unordered list)</b>	$N$	$N$	$N$	$N$	<code>equals()</code>

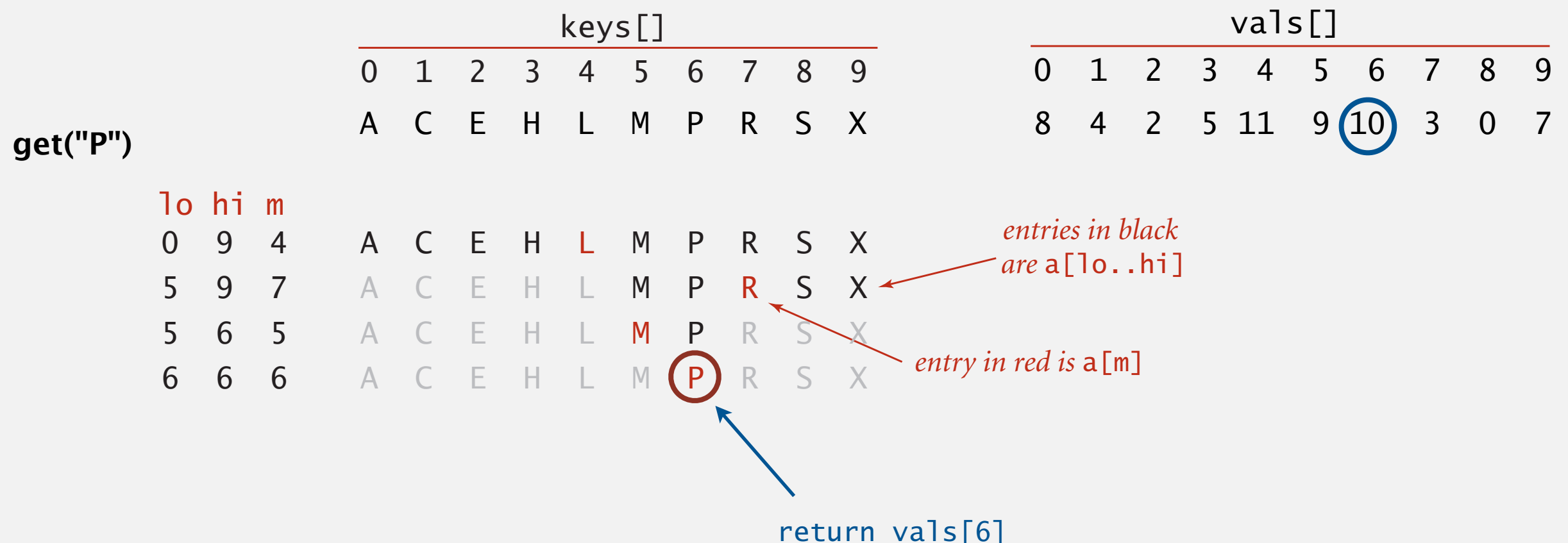
**Challenge.** Efficient implementations of both search and insert.

# Binary search in an ordered array

**Data structure.** Maintain parallel arrays for keys and values, sorted by keys.

**Search.** Use binary search to find key.

**Proposition.** At most  $\sim \lg N$  compares to search a sorted array of length  $N$ .



# Binary search in an ordered array

---

**Data structure.** Maintain parallel arrays for keys and values, sorted by keys.

**Search.** Use binary search to find key.

```
public Value get(Key key)
{
    int lo = 0, hi = N-1;
    while (lo <= hi)
    {
        int mid = lo + (hi - lo) / 2;
        int cmp = key.compareTo(keys[mid]);
        if (cmp < 0) hi = mid - 1;
        else if (cmp > 0) lo = mid + 1;
        else if (cmp == 0) return vals[mid];
    }
    return null; ← no matching key
}
```



# Binary search: insert

---

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

**Insert.** Use binary search to find place to insert; shift all larger keys over.

**Proposition.** Takes linear time in the worst case.

`put("P", 10)`

keys[]									
0	1	2	3	4	5	6	7	8	9
A	C	E	H	M	R	S	X	-	-

vals[]									
0	1	2	3	4	5	6	7	8	9
8	4	6	5	9	3	0	7	-	-

# Elementary ST implementations: summary

---

implementation	guarantee		average case		operations on keys
	search	insert	search hit	insert	
sequential search (unordered list)	$N$	$N$	$N$	$N$	<code>equals()</code>
binary search (ordered array)	$\log N$	$N^\dagger$	$\log N$	$N^\dagger$	<code>compareTo()</code>

$\dagger$  can do with  $\log N$  compares, but requires  $N$  array accesses

**Challenge.** Efficient implementations of both search and insert.



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

## 3.1 SYMBOL TABLES

---

- ▶ *API*
- ▶ *elementary implementations*
- ▶ *ordered operations*

# Examples of ordered symbol table API

---

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

`size(09:15:00, 09:25:00)` is 5  
`rank(09:10:25)` is 7

# Ordered symbol table API

---

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

```
    ⋮
```

```
    Key min()
```

*smallest key*

```
    Key max()
```

*largest key*

```
    Key floor(Key key)
```

*largest key less than or equal to key*

```
    Key ceiling(Key key)
```

*smallest key greater than or equal to key*

```
    int rank(Key key)
```

*number of keys less than key*

```
    Key select(int k)
```

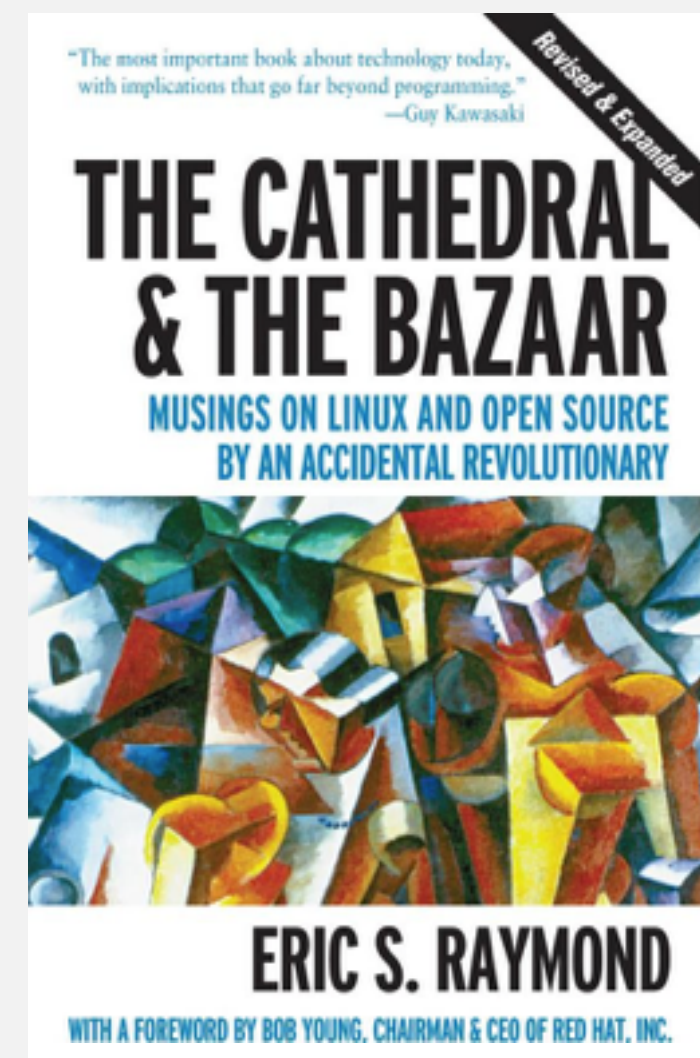
*key of rank k*

```
    ⋮
```

# Smart data structures

---

*“ Smart data structures and dumb code works a lot better than the other way around. ” — Eric S. Raymond*



# Binary search: ordered symbol table operations summary

---

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

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





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

## 3.2 BINARY SEARCH TREES

---

- ▶ *BSTs*
- ▶ *ordered operations*
- ▶ *iteration*
- ▶ *deletion (see book)*



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

## 3.2 BINARY SEARCH TREES

---

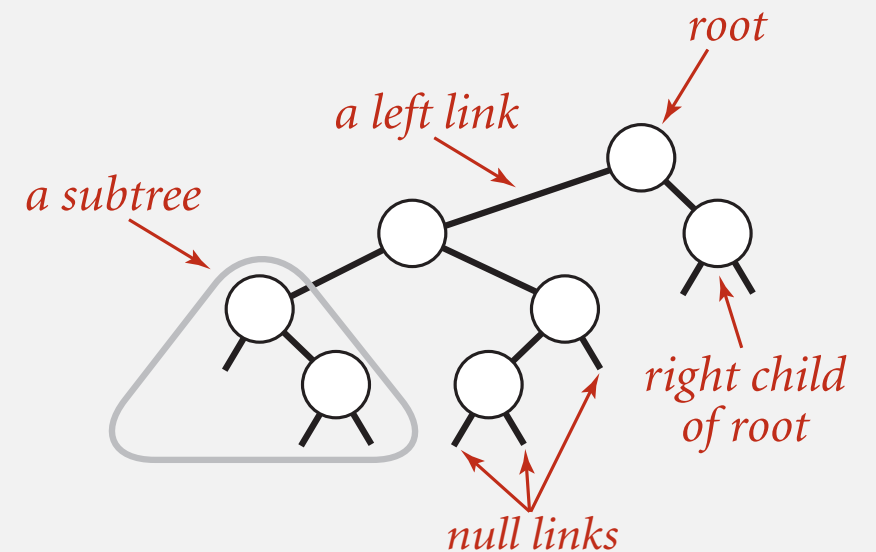
- ▶ *BSTs*
- ▶ *ordered operations*
- ▶ *iteration*
- ▶ *deletion*

# Binary search trees

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

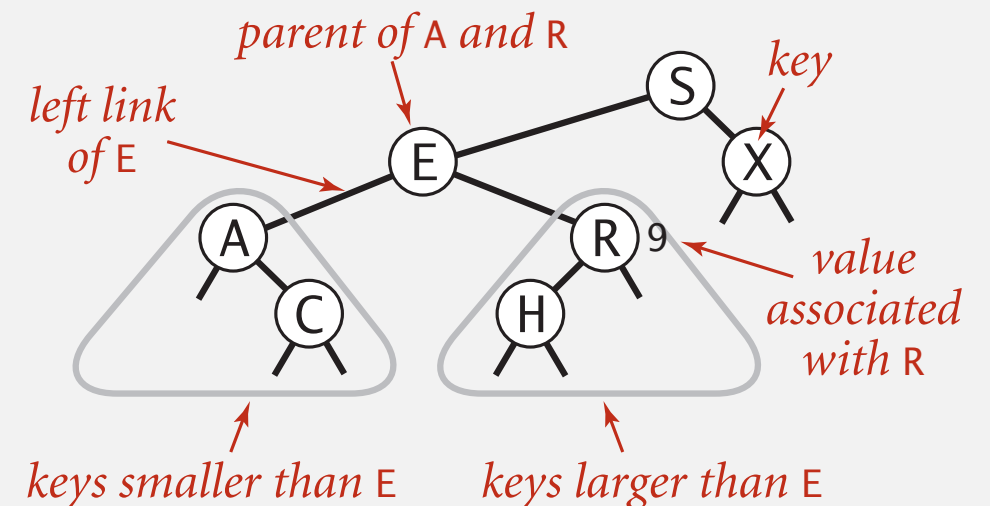
A binary tree is either:

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



**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.

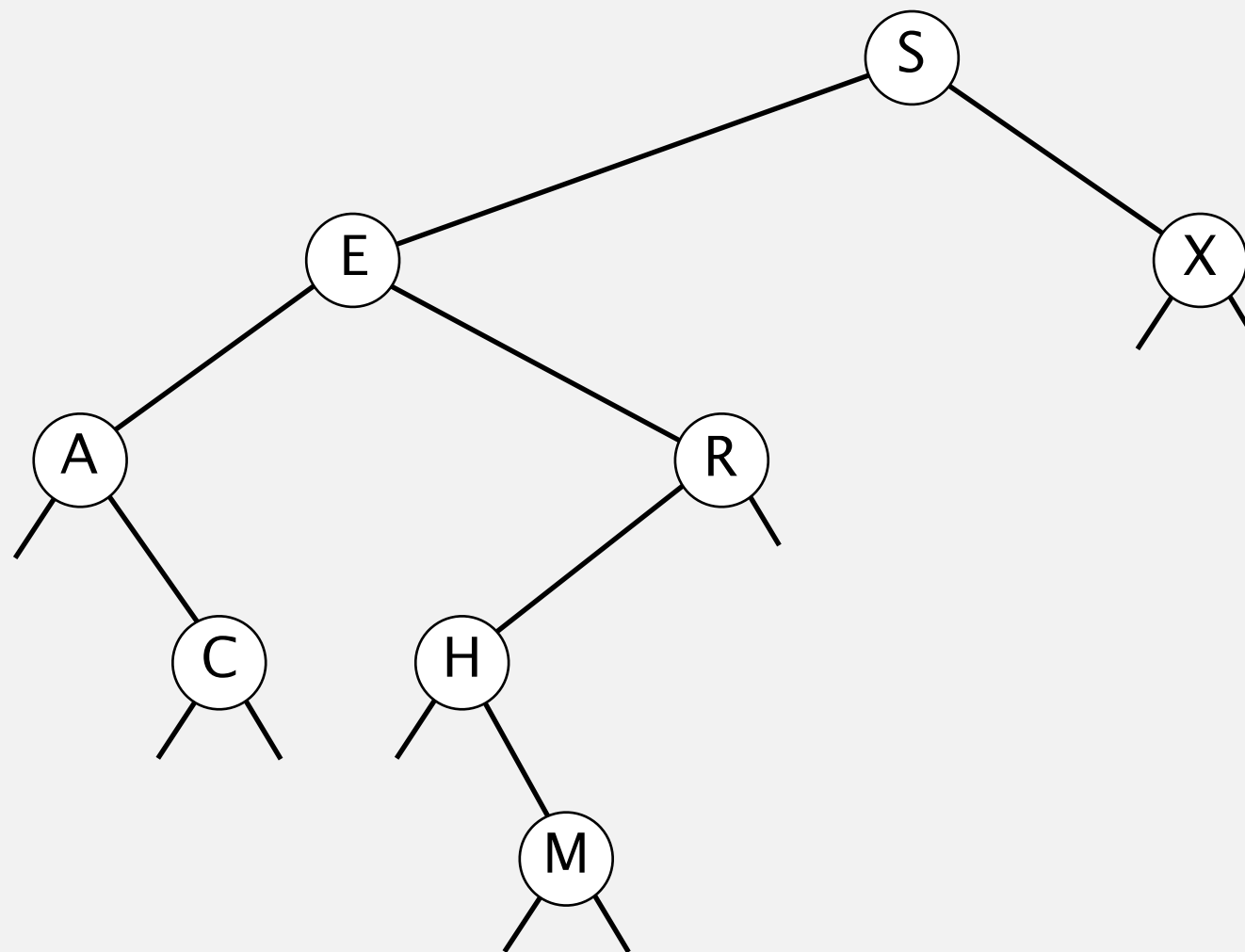


# Binary search tree demo

---

**Search.** If less, go left; if greater, go right; if equal, search hit.

successful search for H

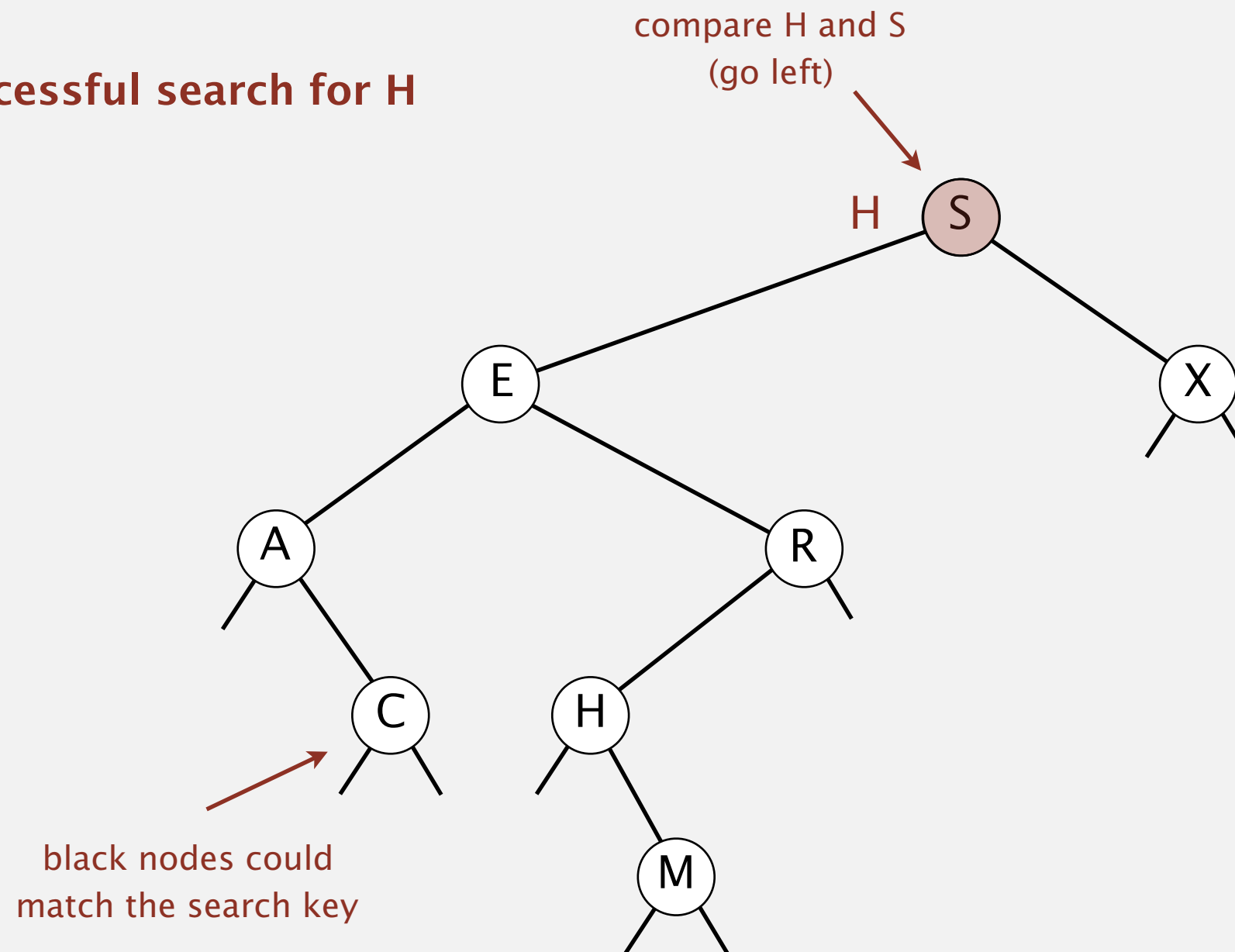


# Binary search tree demo

---

**Search.** If less, go left; if greater, go right; if equal, search hit.

**successful search for H**

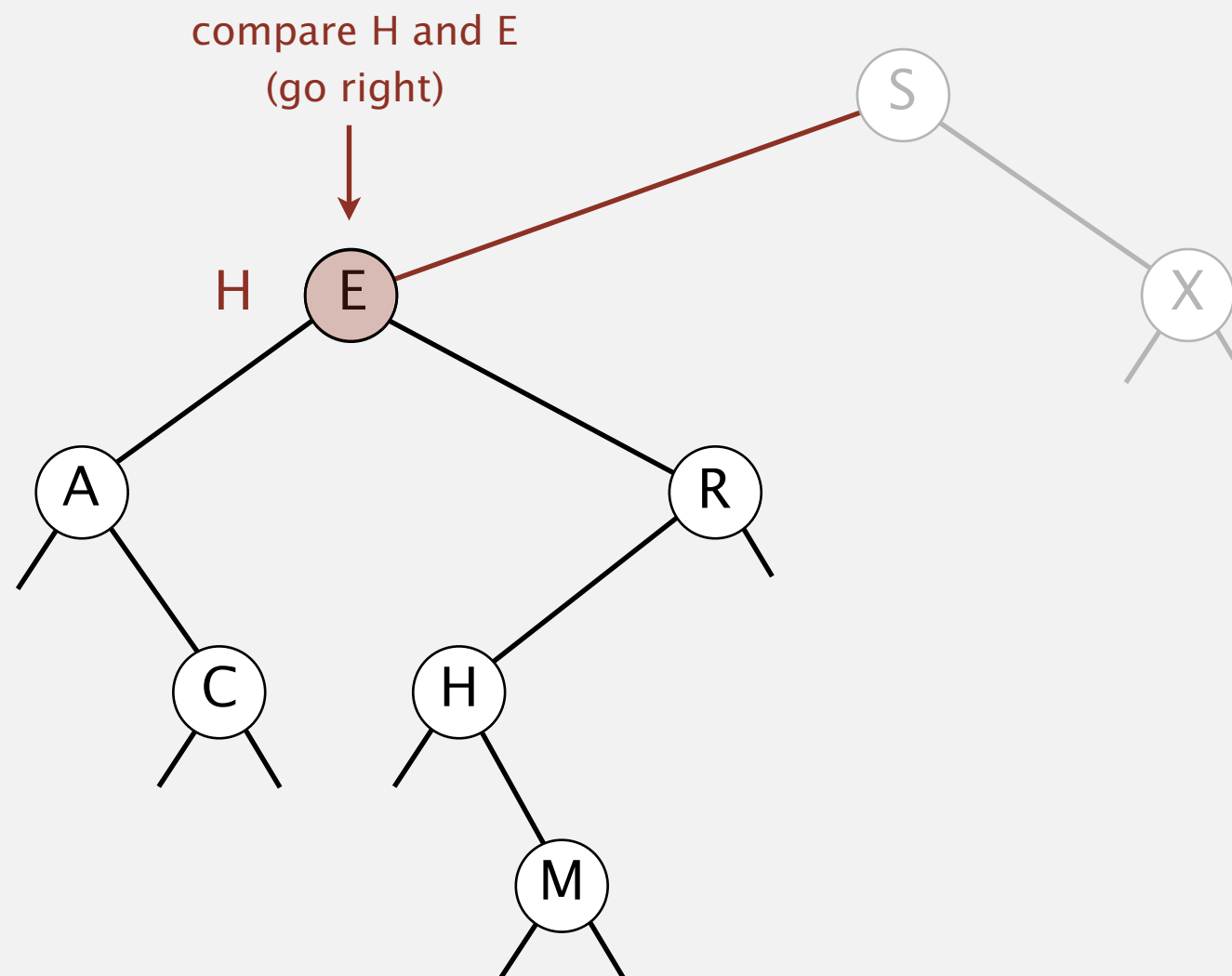


# Binary search tree demo

---

**Search.** If less, go left; if greater, go right; if equal, search hit.

**successful search for H**

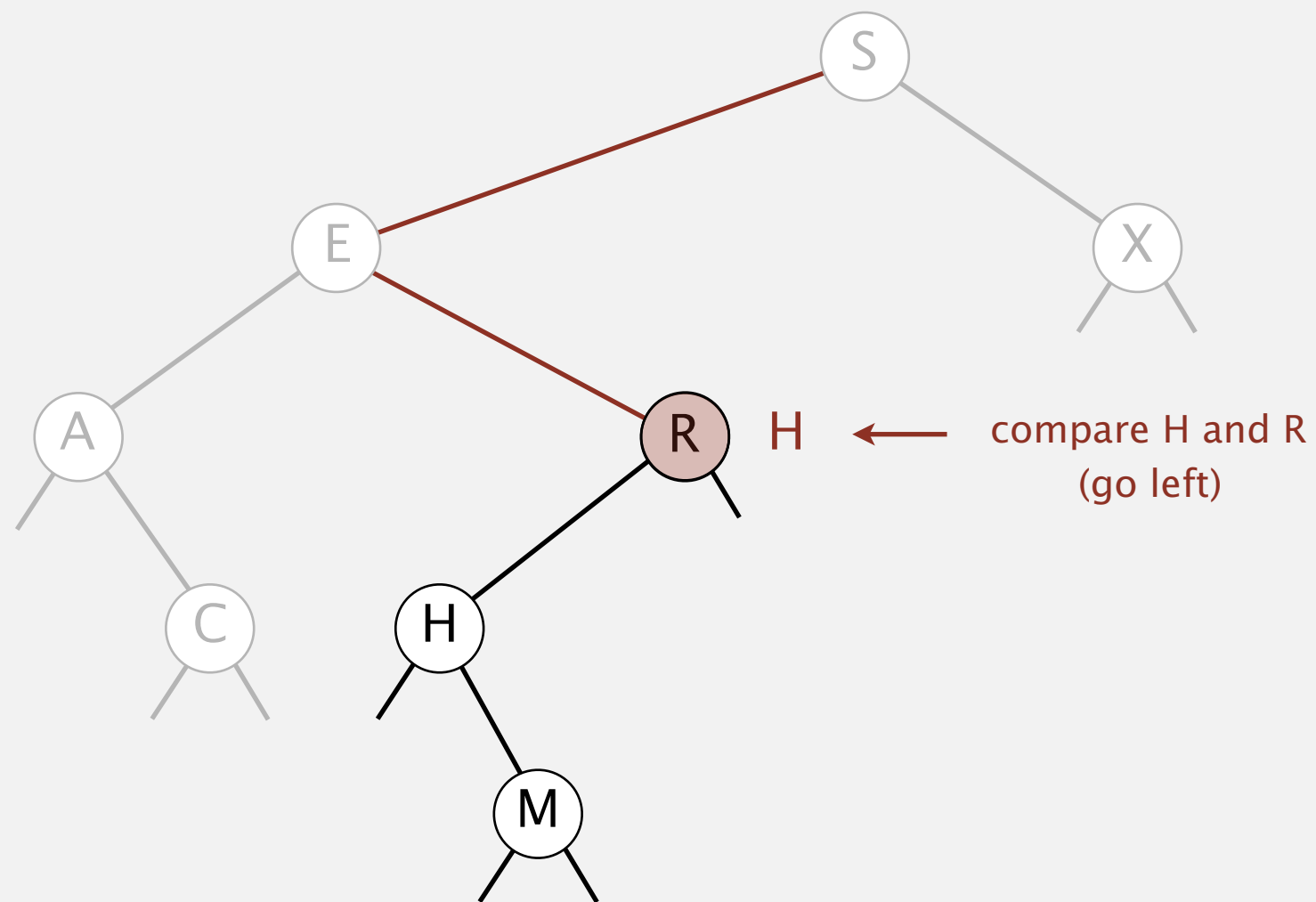


# Binary search tree demo

---

**Search.** If less, go left; if greater, go right; if equal, search hit.

successful search for H



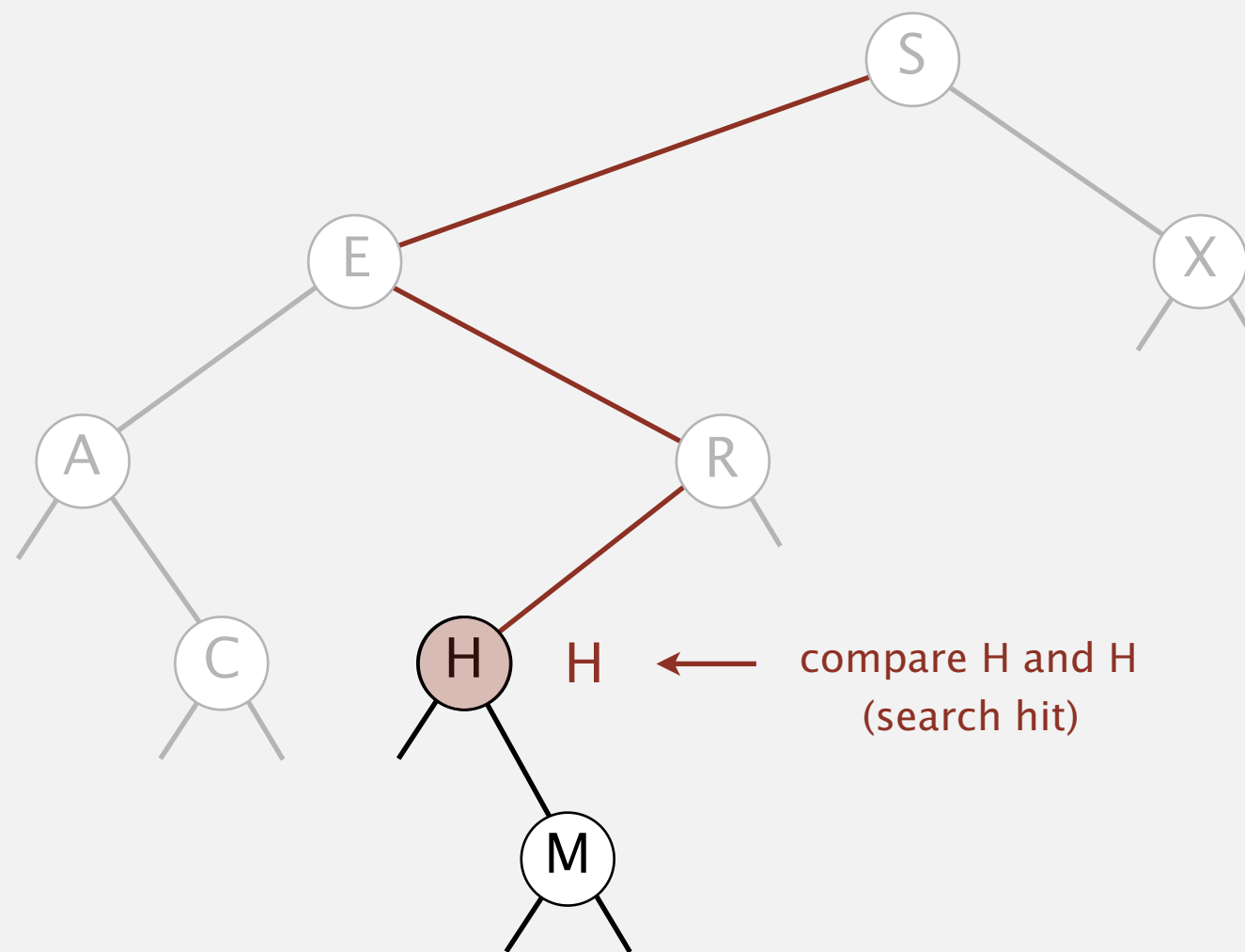


# Binary search tree demo

---

**Search.** If less, go left; if greater, go right; if equal, search hit.

successful search for H

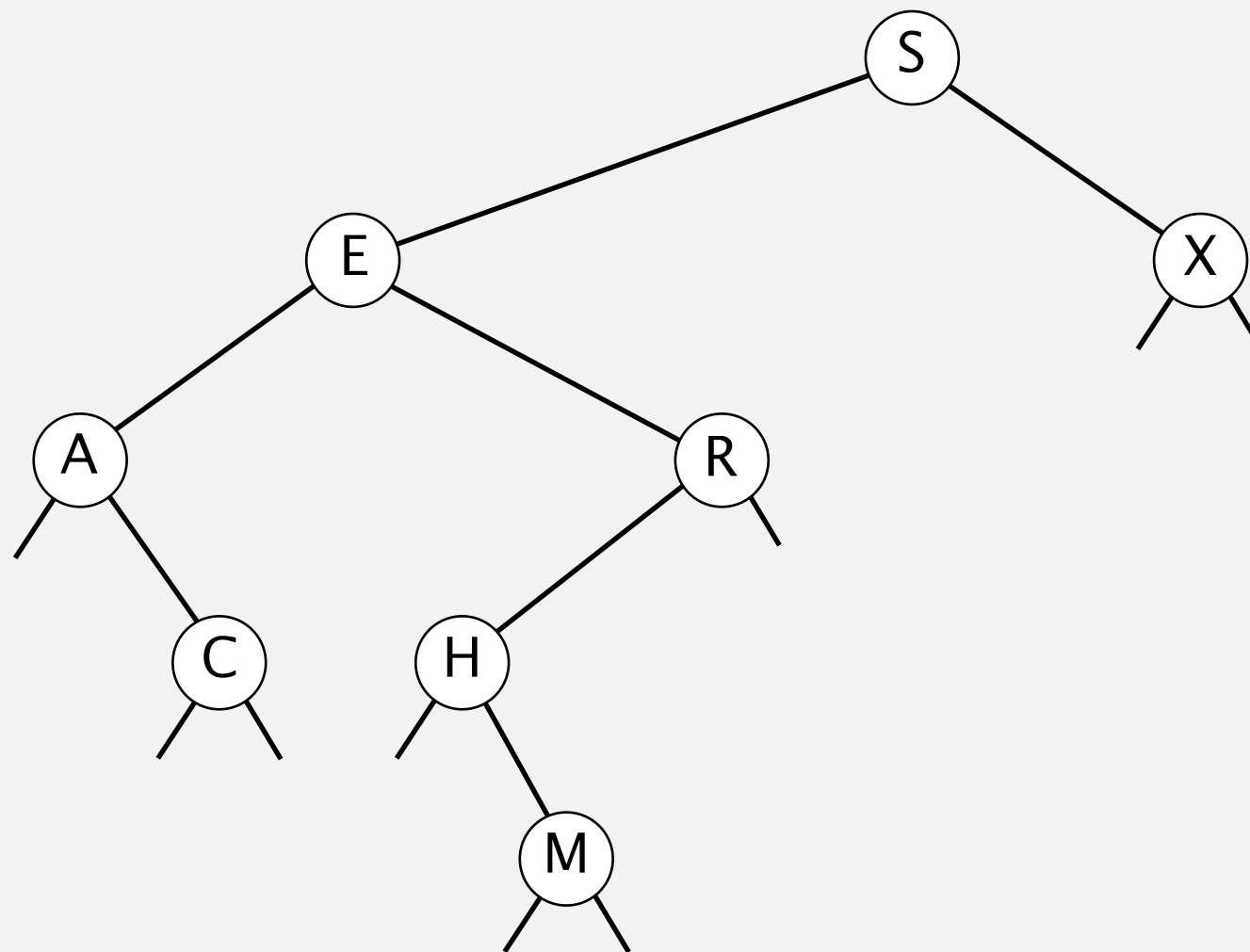


# Binary search tree demo

---

**Search.** If less, go left; if greater, go right; if equal, search hit.

**unsuccessful search for G**

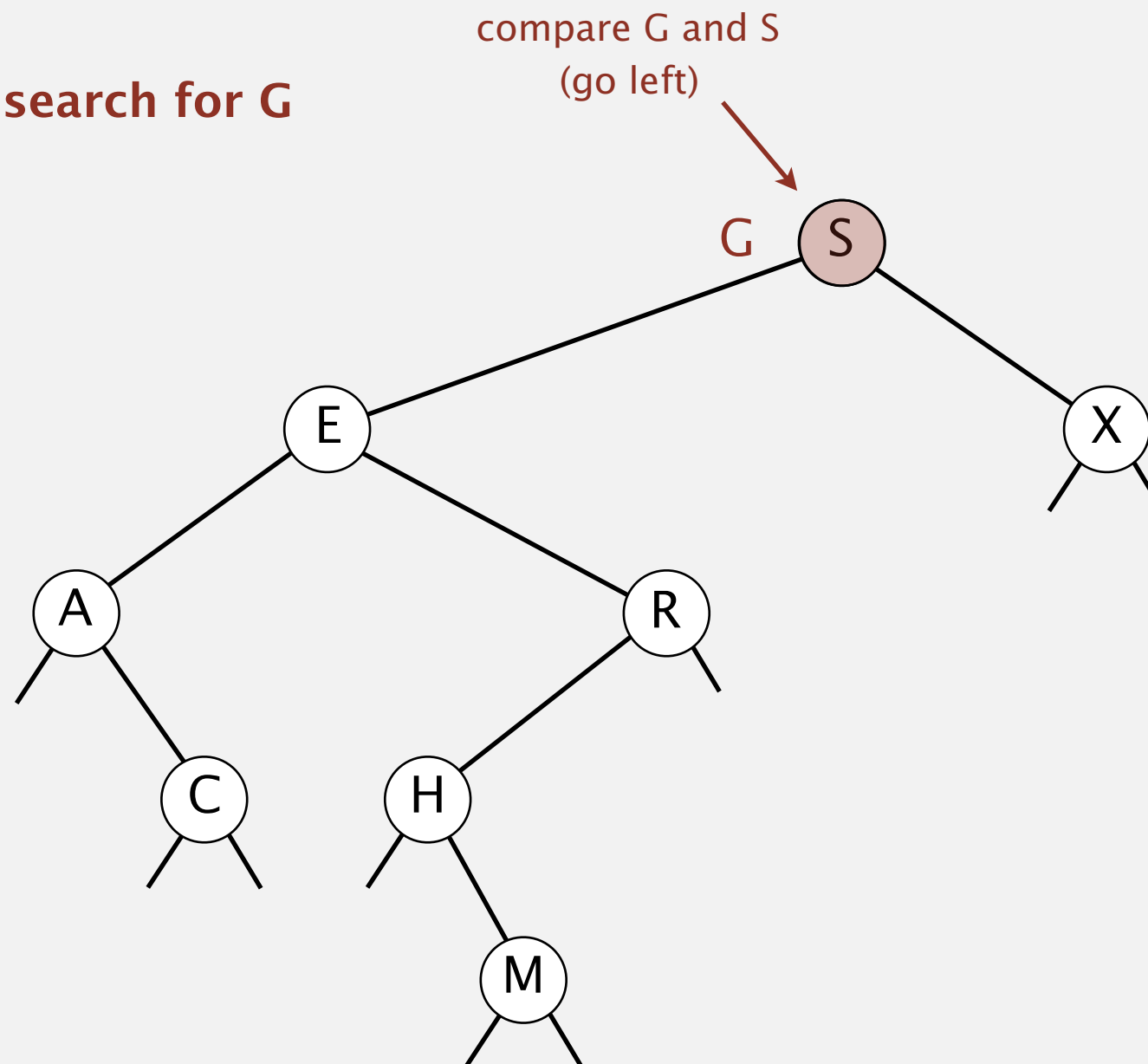


# Binary search tree demo

---

**Search.** If less, go left; if greater, go right; if equal, search hit.

unsuccessful search for G

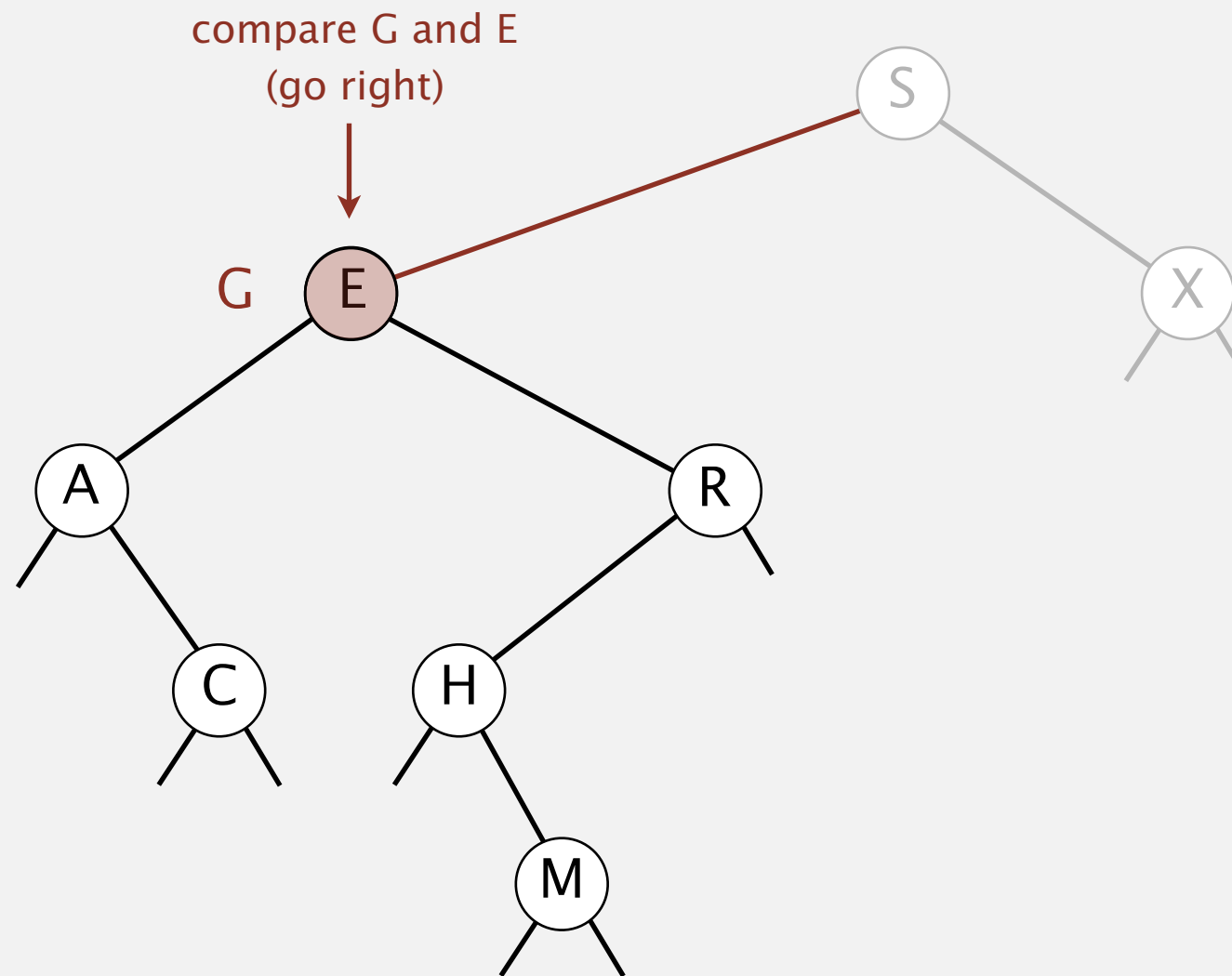


# Binary search tree demo

---

**Search.** If less, go left; if greater, go right; if equal, search hit.

**unsuccessful search for G**

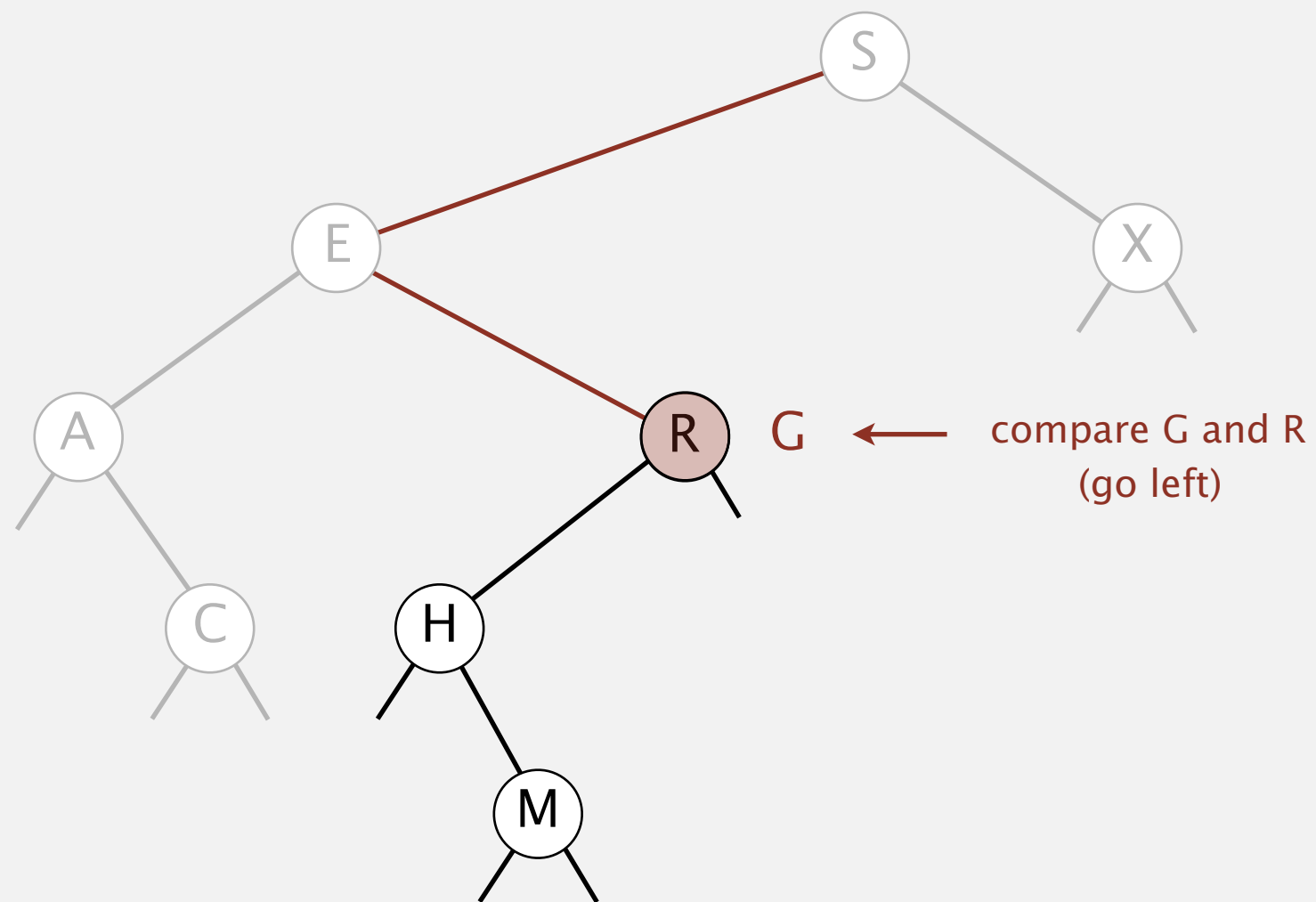


# Binary search tree demo

---

**Search.** If less, go left; if greater, go right; if equal, search hit.

**unsuccessful search for G**

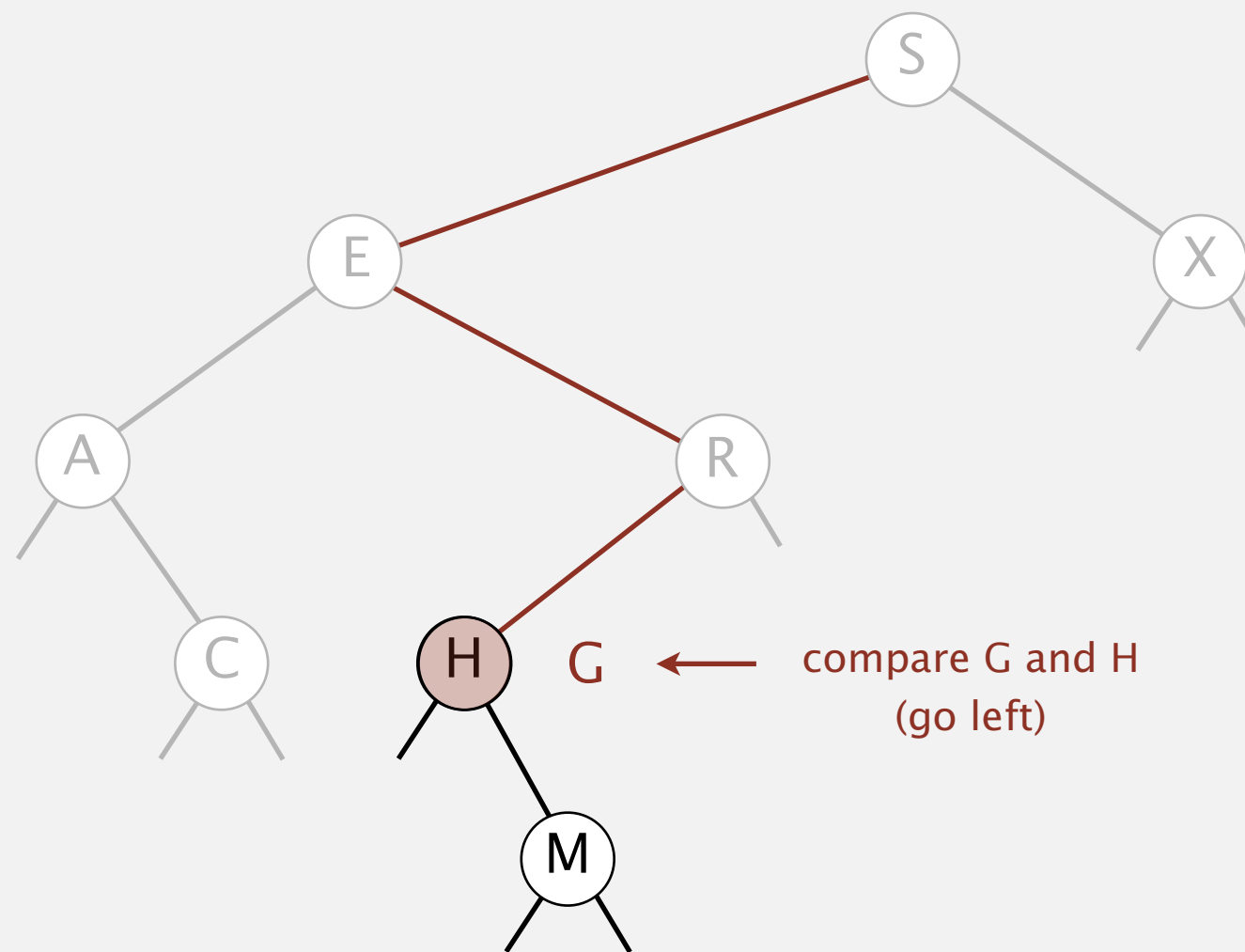


# Binary search tree demo

---

**Search.** If less, go left; if greater, go right; if equal, search hit.

**unsuccessful search for G**

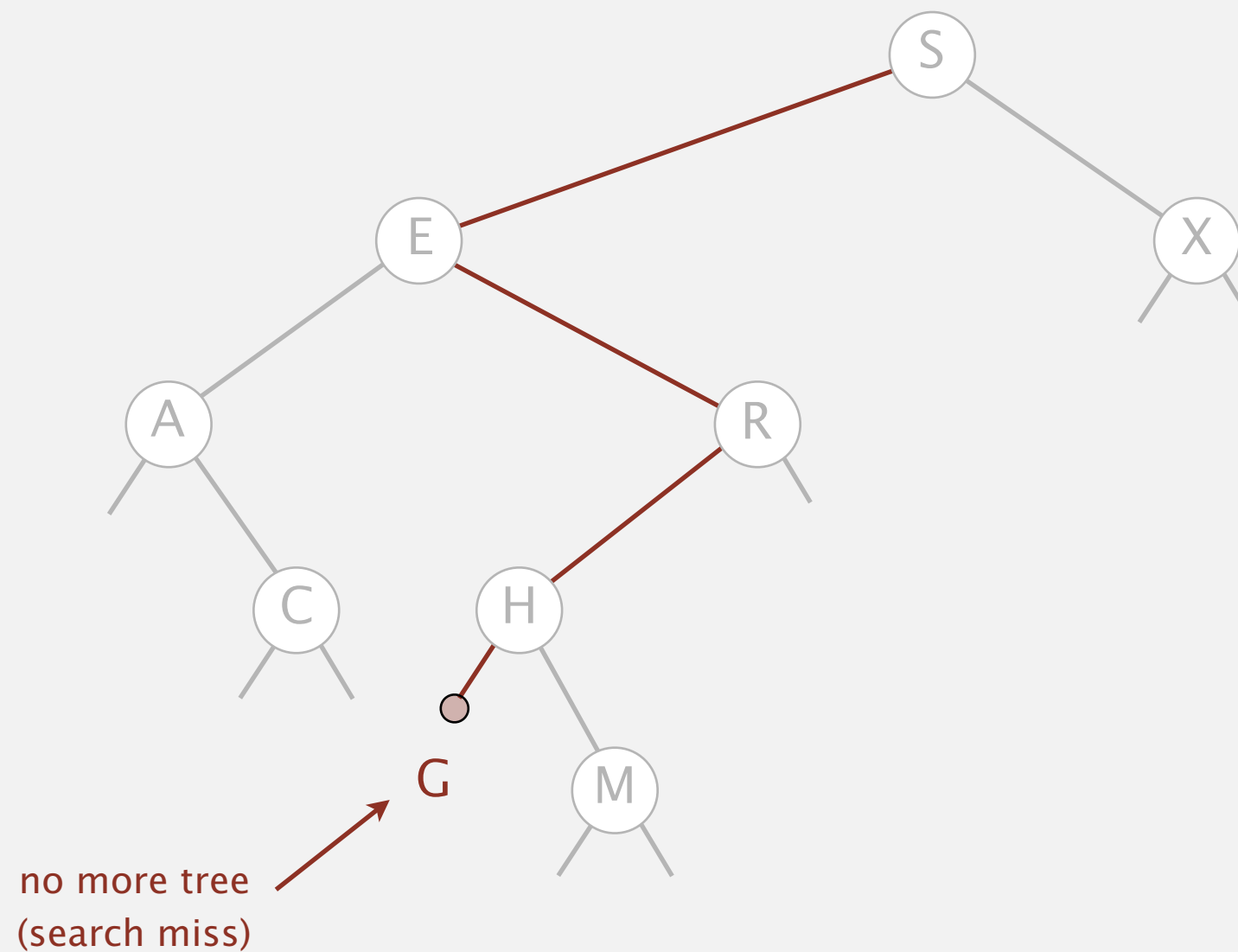


# Binary search tree demo

---

**Search.** If less, go left; if greater, go right; if equal, search hit.

unsuccessful search for G

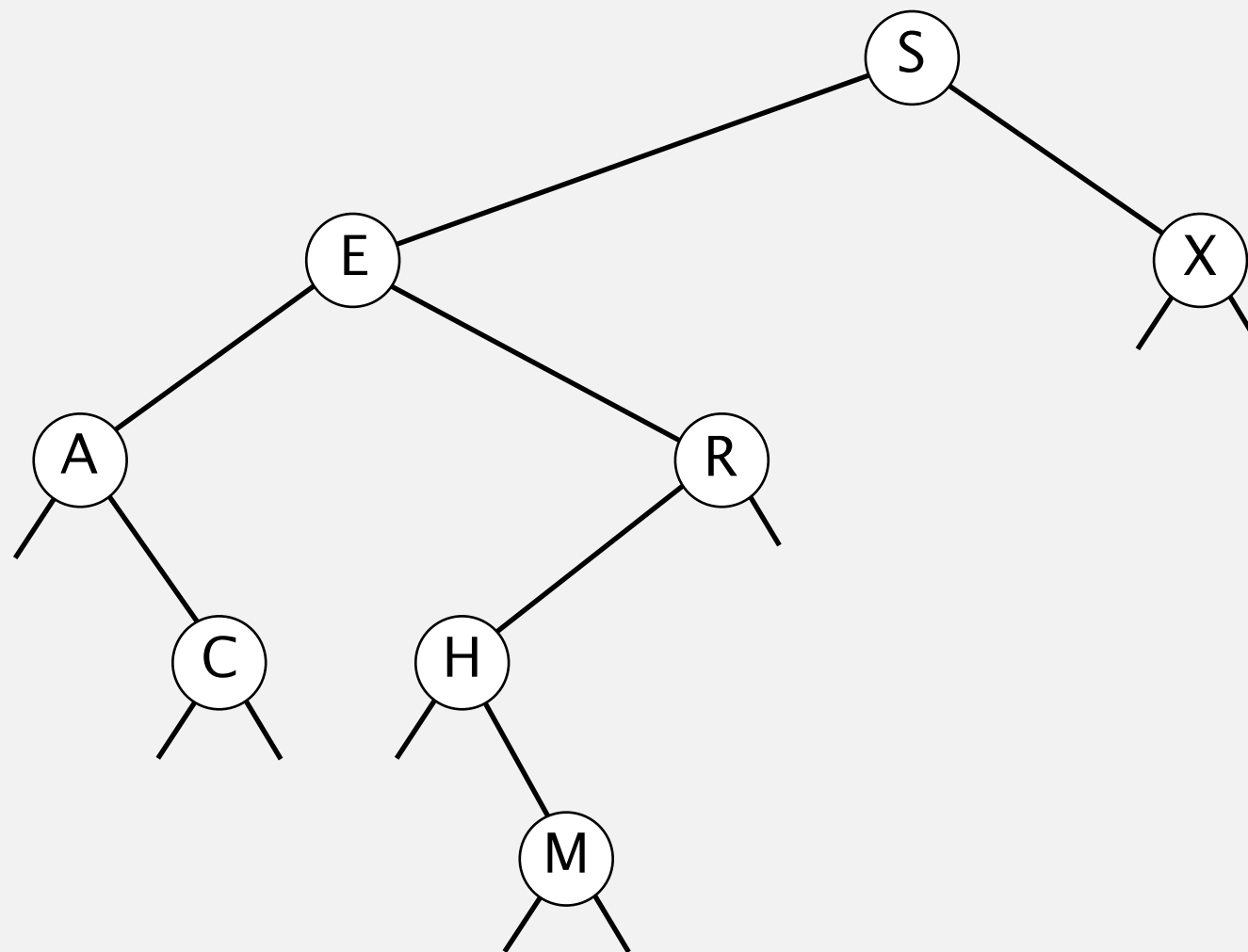


# Binary search tree demo

---

**Insert.** If less, go left; if greater, go right; if null, insert.

insert G



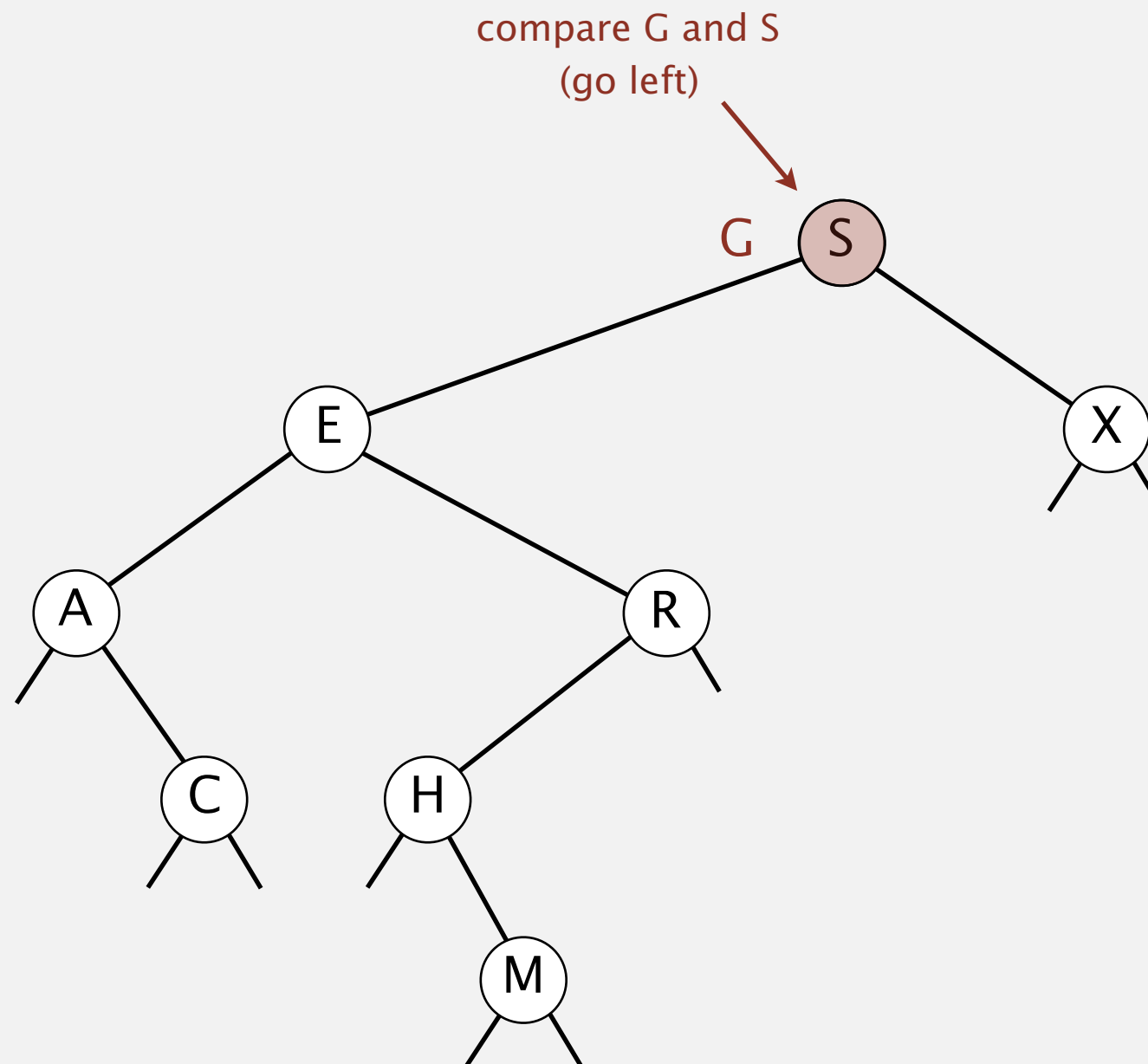


# Binary search tree demo

---

**Insert.** If less, go left; if greater, go right; if null, insert.

insert G

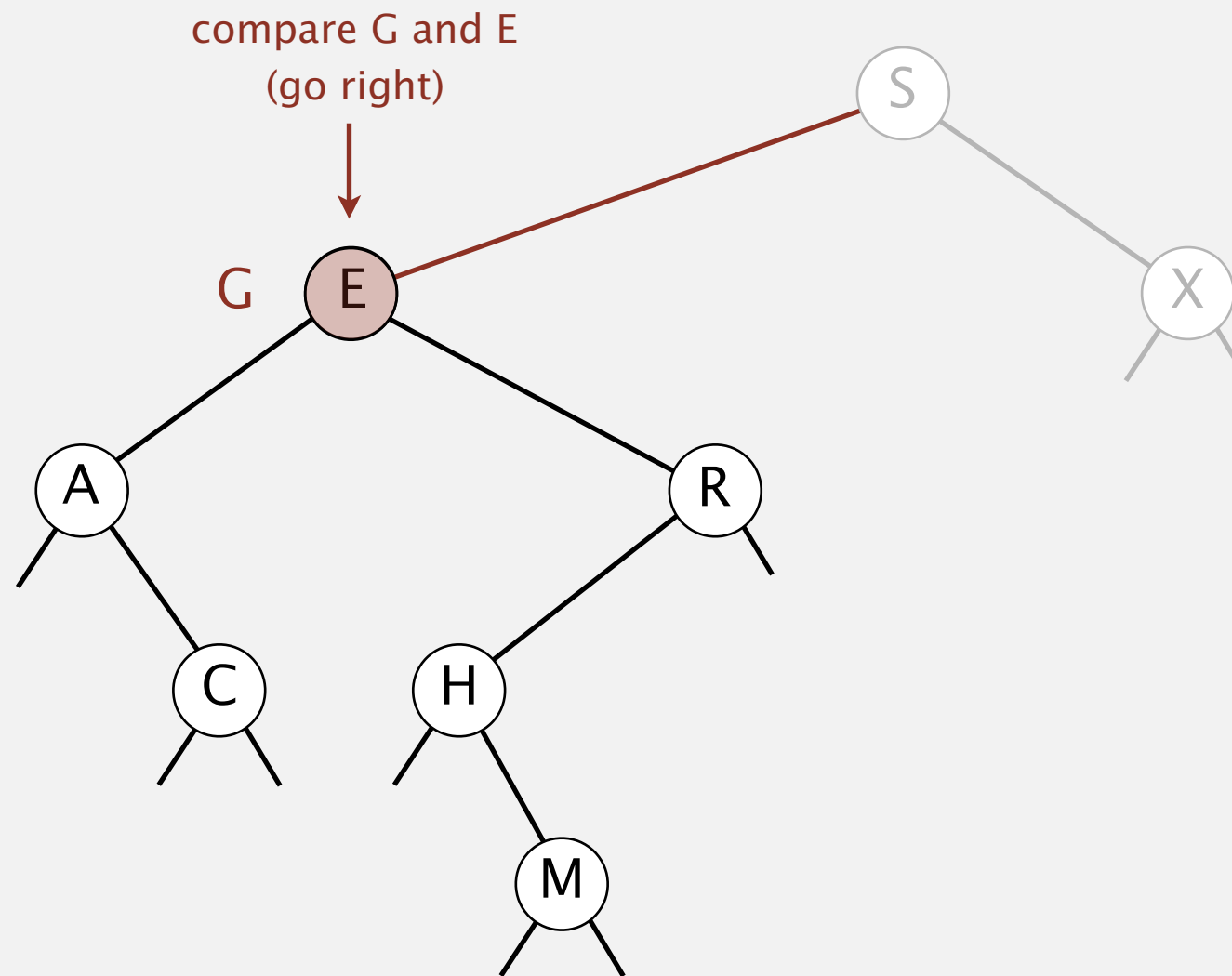


# Binary search tree demo

---

**Insert.** If less, go left; if greater, go right; if null, insert.

**insert G**

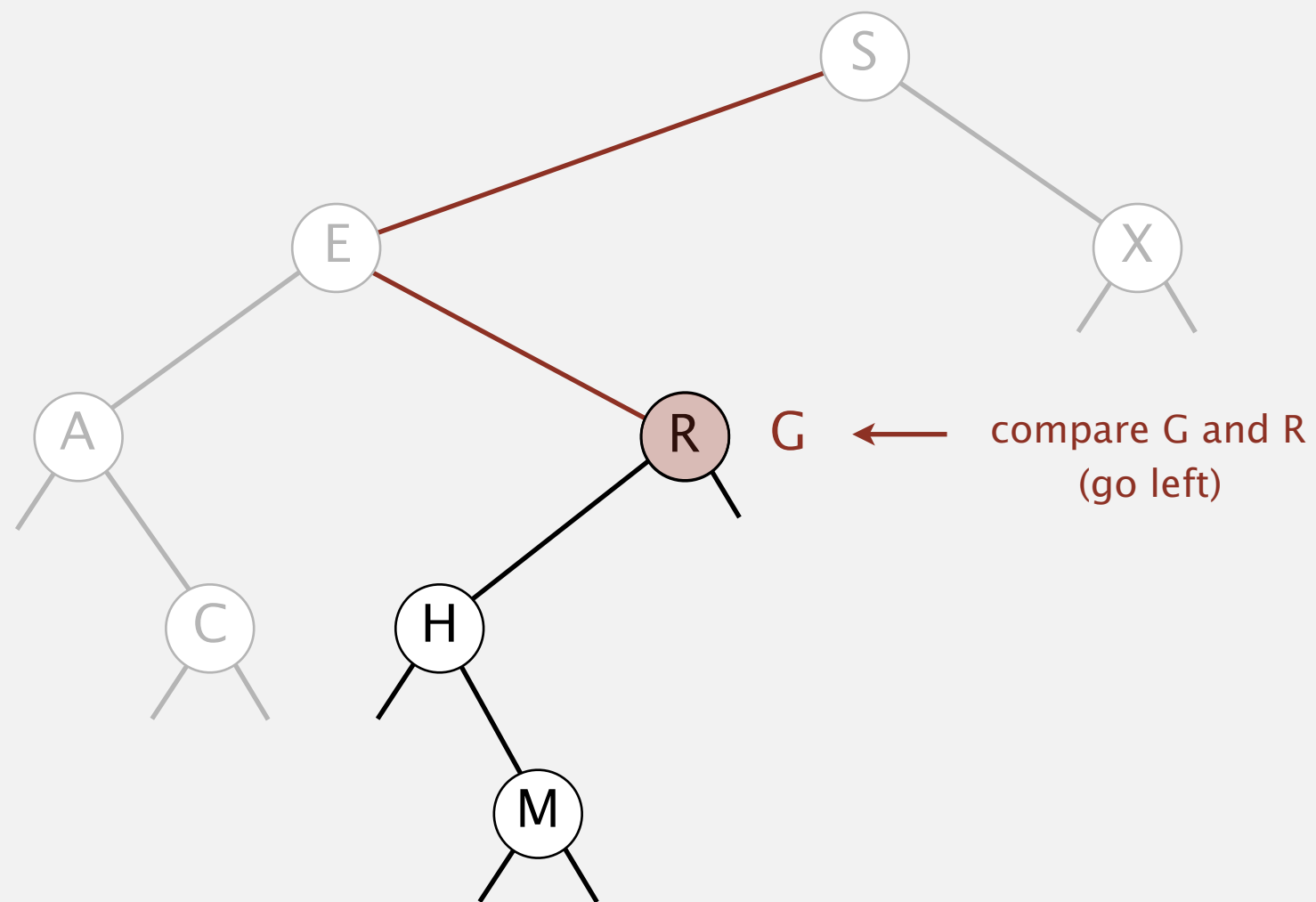


# Binary search tree demo

---

**Insert.** If less, go left; if greater, go right; if null, insert.

**insert G**

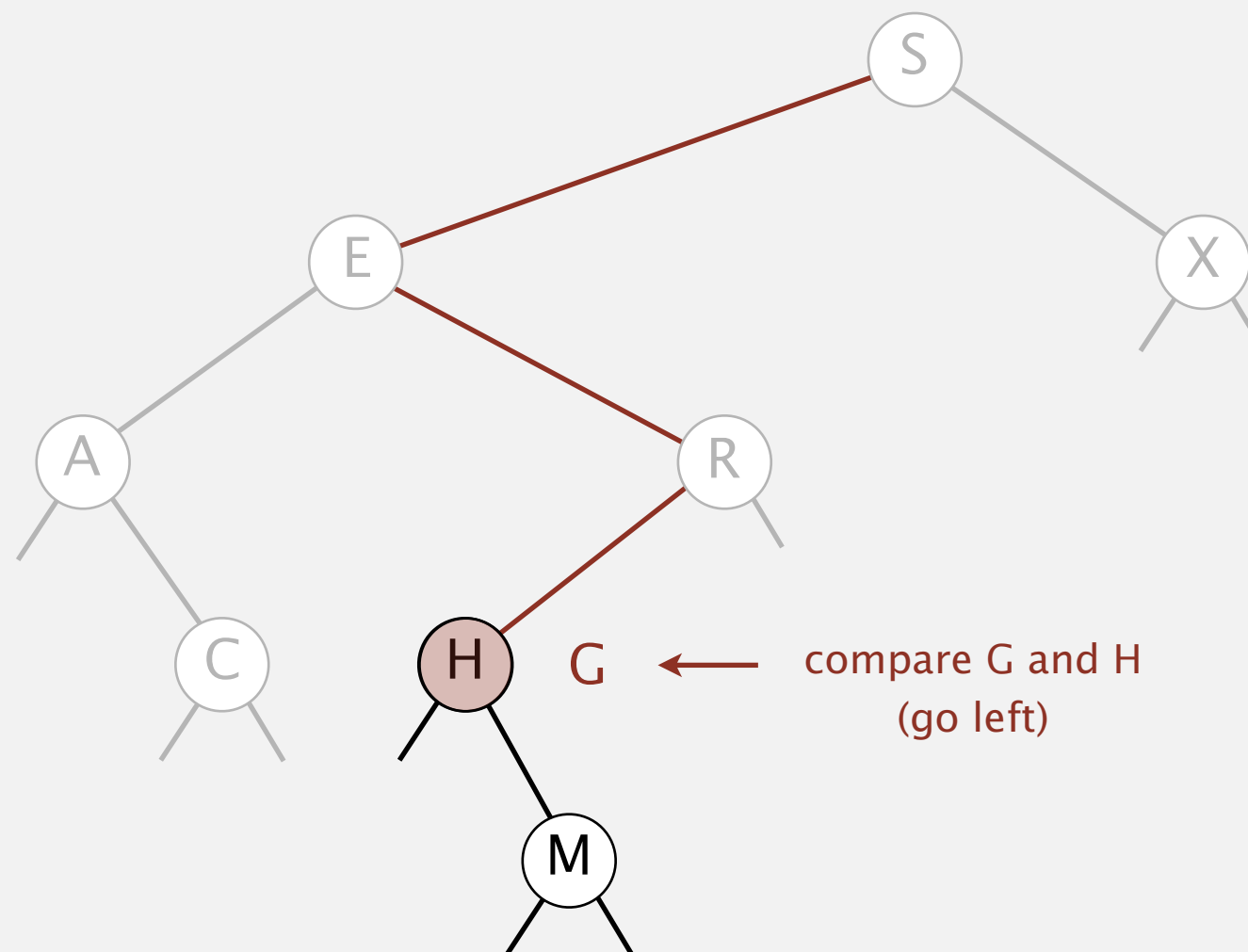


# Binary search tree demo

---

**Insert.** If less, go left; if greater, go right; if null, insert.

**insert G**

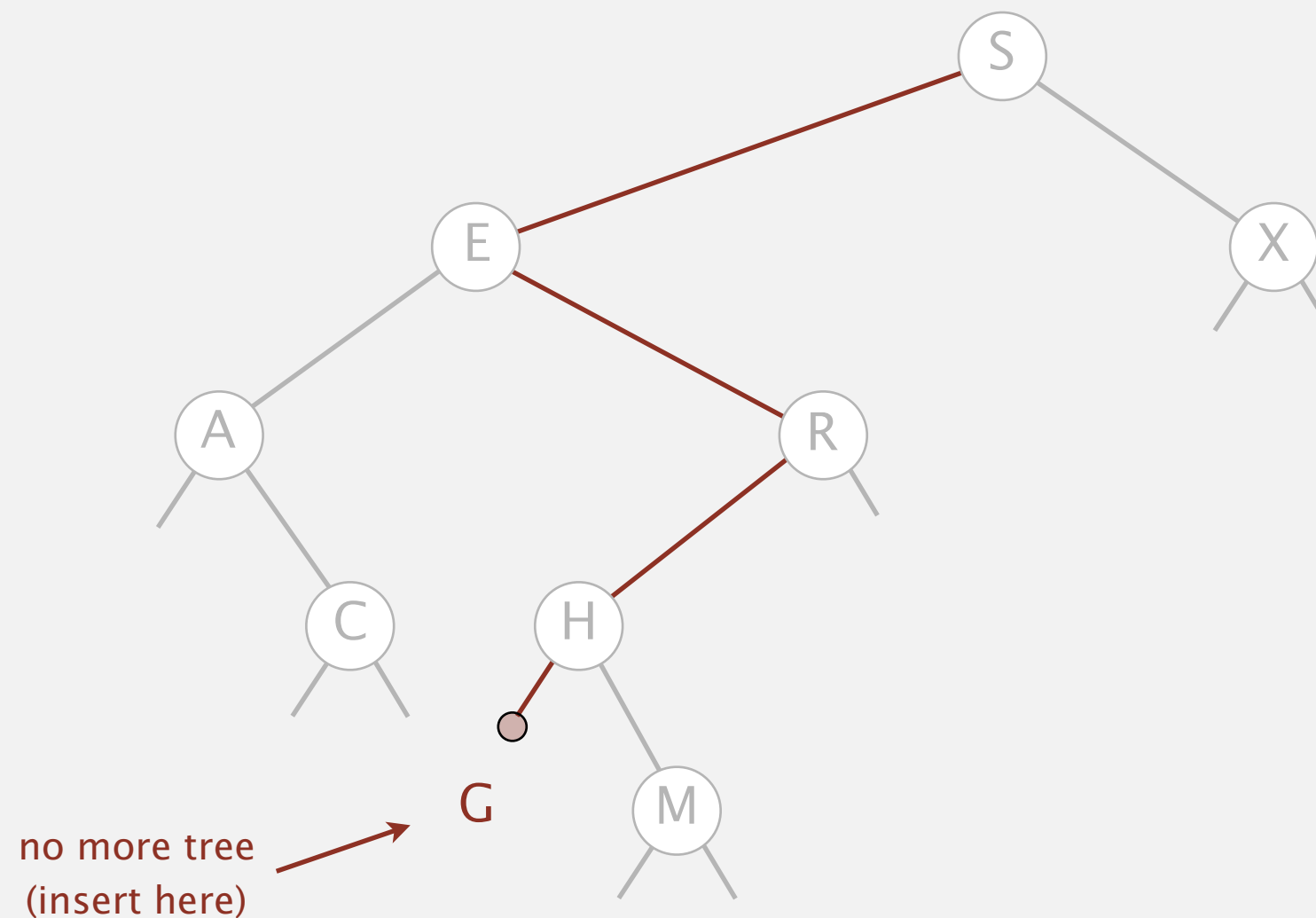


# Binary search tree demo

---

**Insert.** If less, go left; if greater, go right; if null, insert.

**insert G**

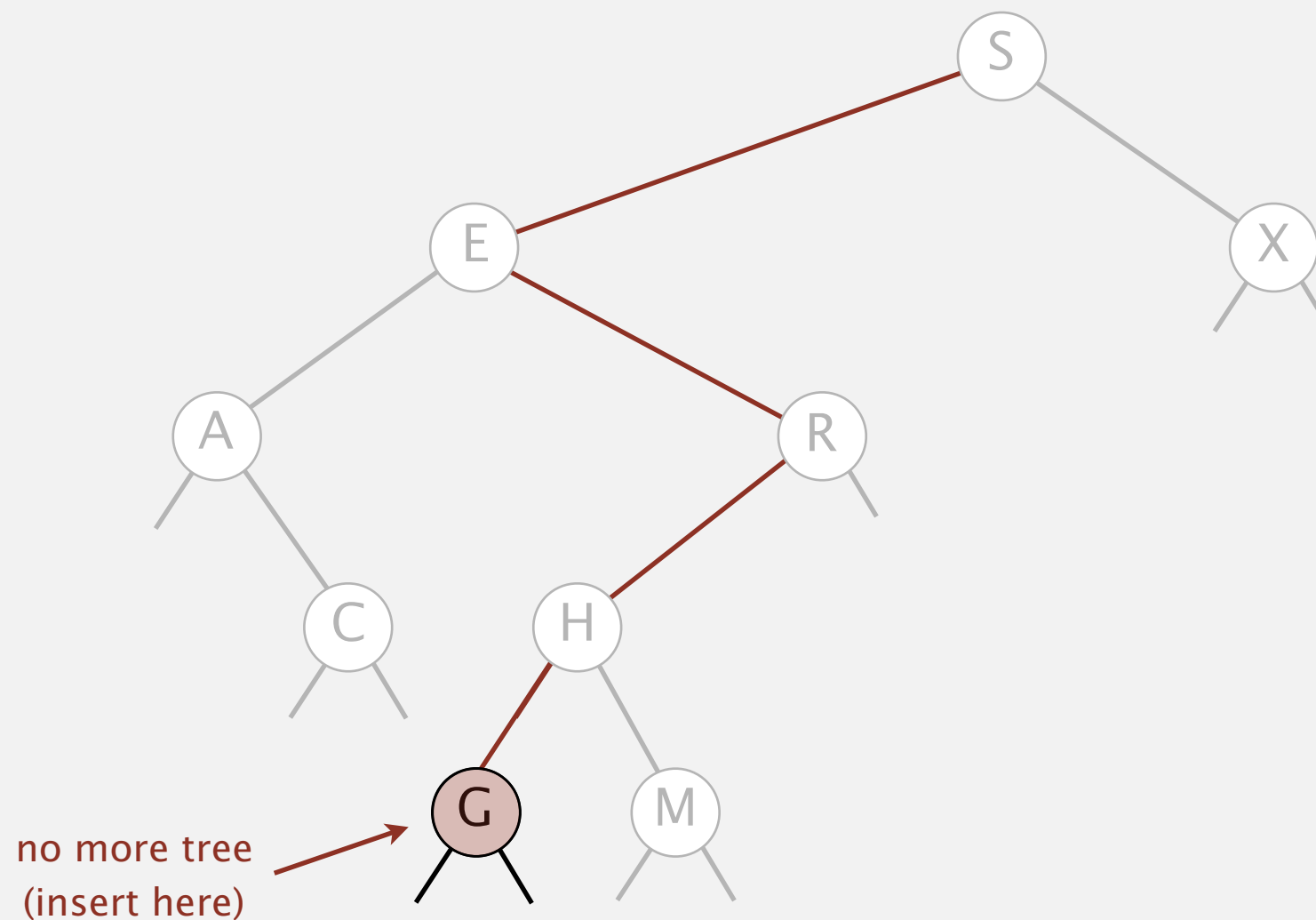


# Binary search tree demo

---

**Insert.** If less, go left; if greater, go right; if null, insert.

**insert G**

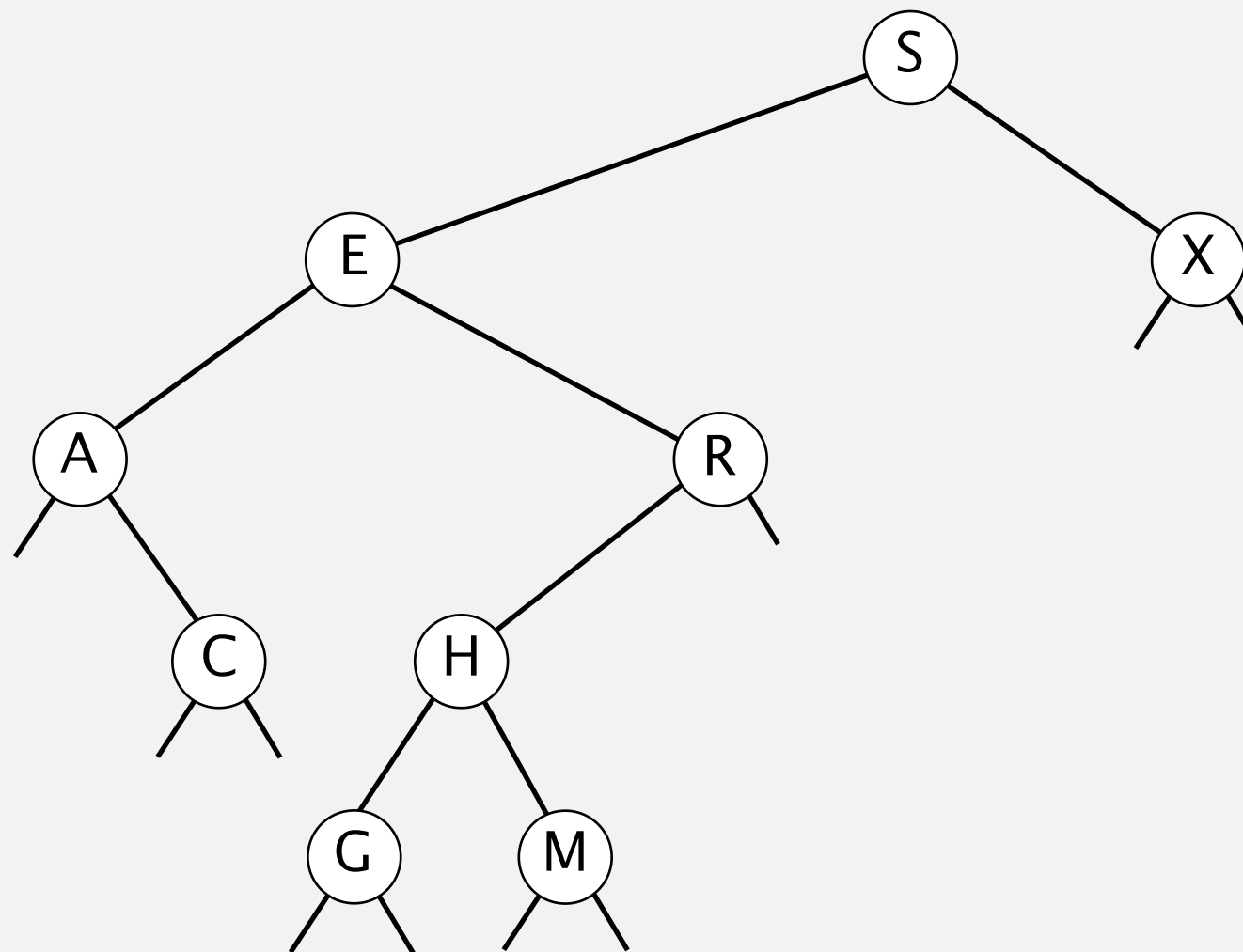


# Binary search tree demo

---

**Insert.** If less, go left; if greater, go right; if null, insert.

**insert G**



# BST representation in Java

**Java definition.** A BST is a reference to a root Node.

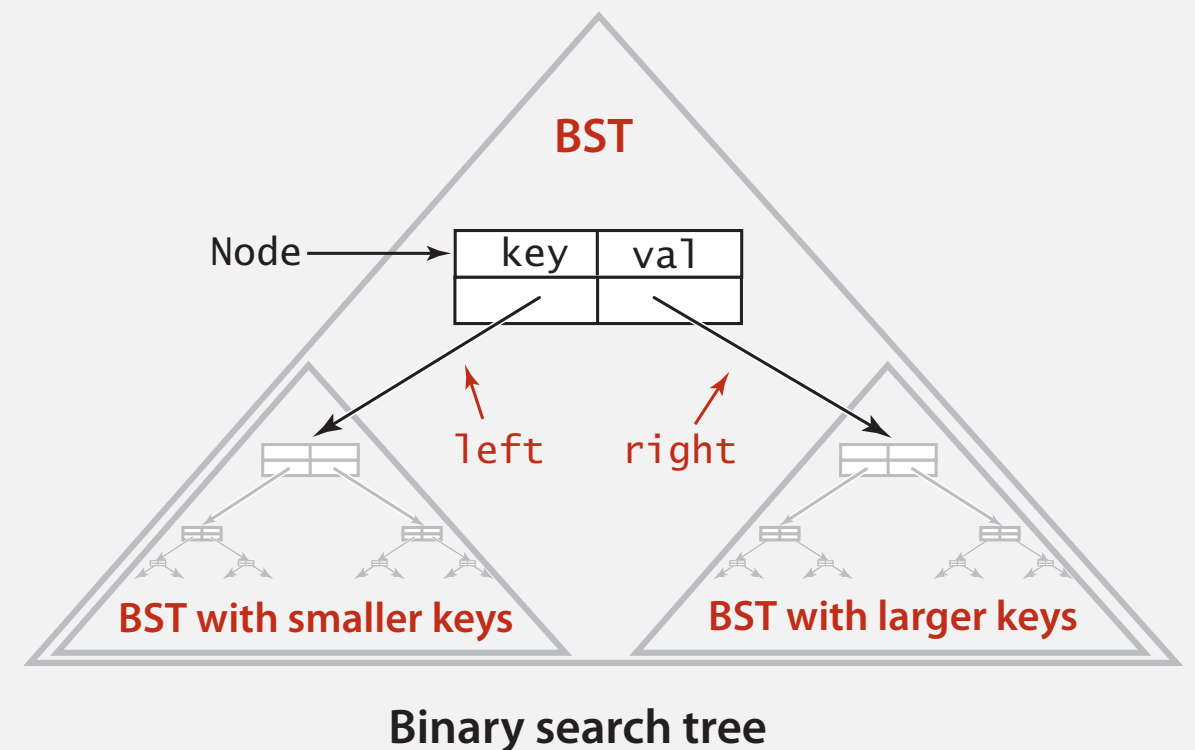
A Node is composed of four fields:

- A Key and a Value.
- A reference to the left and right subtree.

↑ smaller keys      ↑ larger keys

```
private class Node
{
    private Key key;
    private Value val;
    private Node left, right;
    public Node(Key key, Value val)
    {
        this.key = key;
        this.val = val;
    }
}
```

Key and Value are generic types; Key is Comparable





# BST implementation (skeleton)

---

```
public class BST<Key extends Comparable<Key>, Value>  
{
```

```
    private Node root;
```

← root of BST

```
    private class Node  
    { /* see previous slide */ }
```

```
    public void put(Key key, Value val)  
    { /* see next slides */ }
```

```
    public Value get(Key key)  
    { /* see next slides */ }
```

```
    public Iterable<Key> iterator()  
    { /* see slides in next section */ }
```

```
    public void delete(Key key)  
    { /* see textbook */ }
```

```
}
```

# BST search: Java implementation

---

**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 = 1 + depth of node.

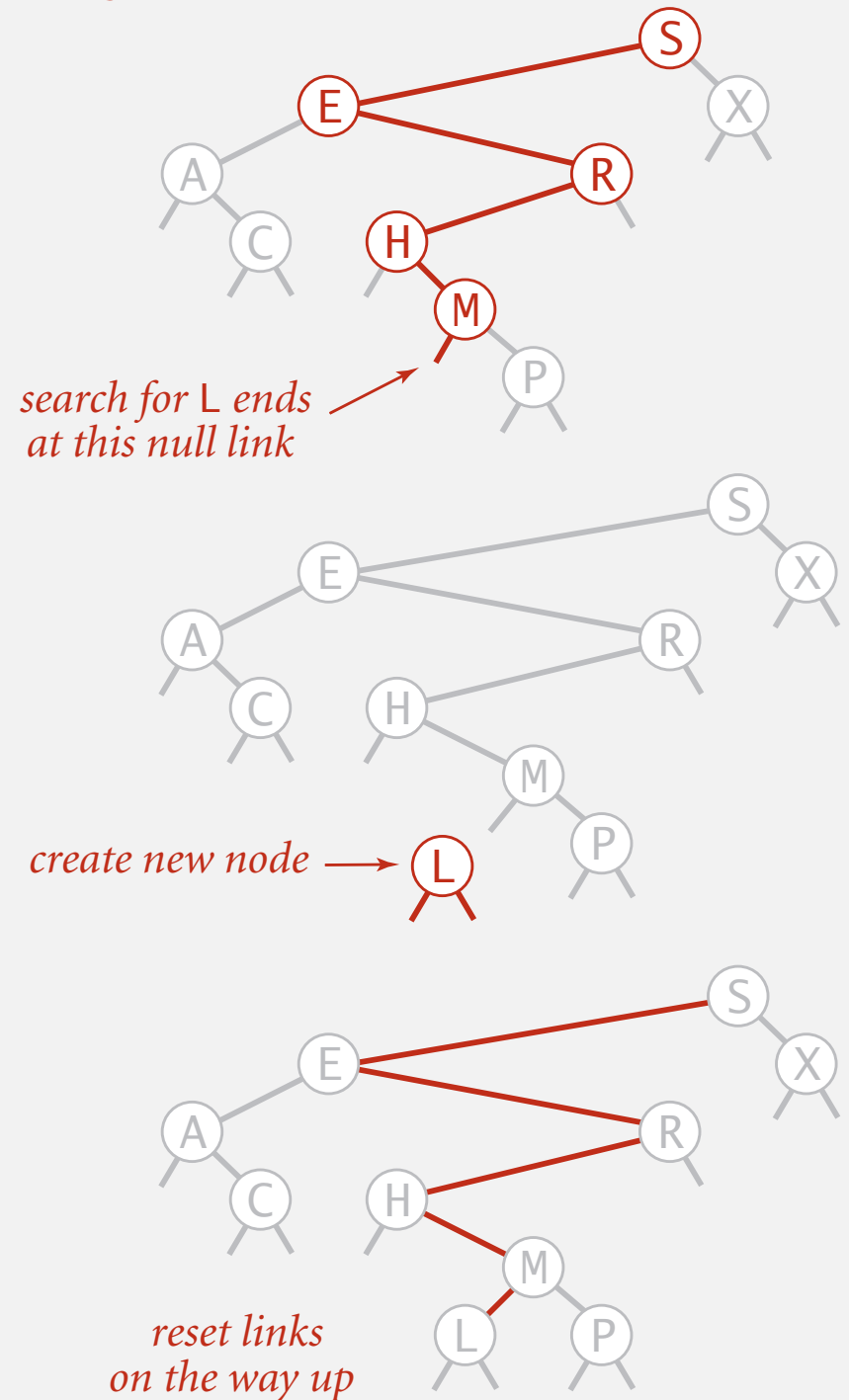
# BST insert

**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



Insertion into a BST

# BST insert: Java implementation

---

**Put.** Associate value with key.

```
public void put(Key key, Value val)
{
    root = put(root, key, val);
}

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;
    return x;
}
```



**Warning: concise but tricky code; read carefully!**

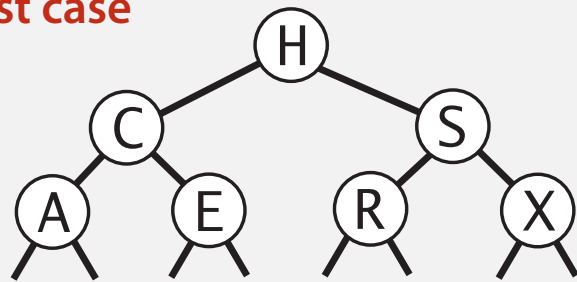
**Cost.** Number of compares = 1 + depth of node.

# Tree shape

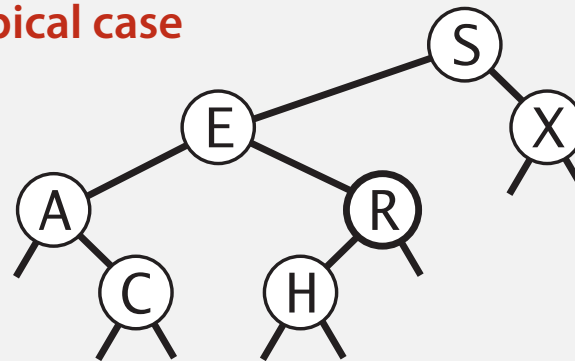
---

- Many BSTs correspond to same set of keys.
- Number of compares for search/insert = 1 + depth of node.

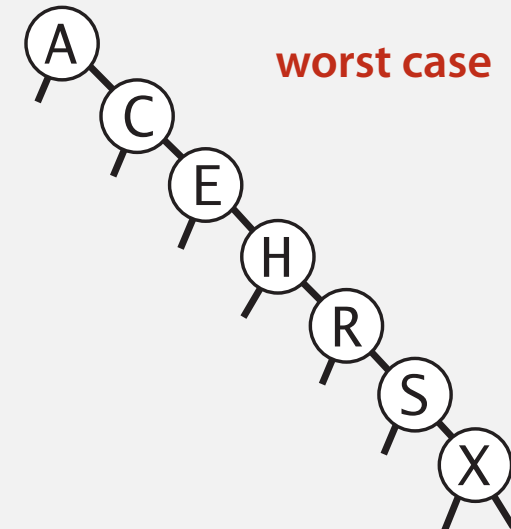
best case



typical case



worst case

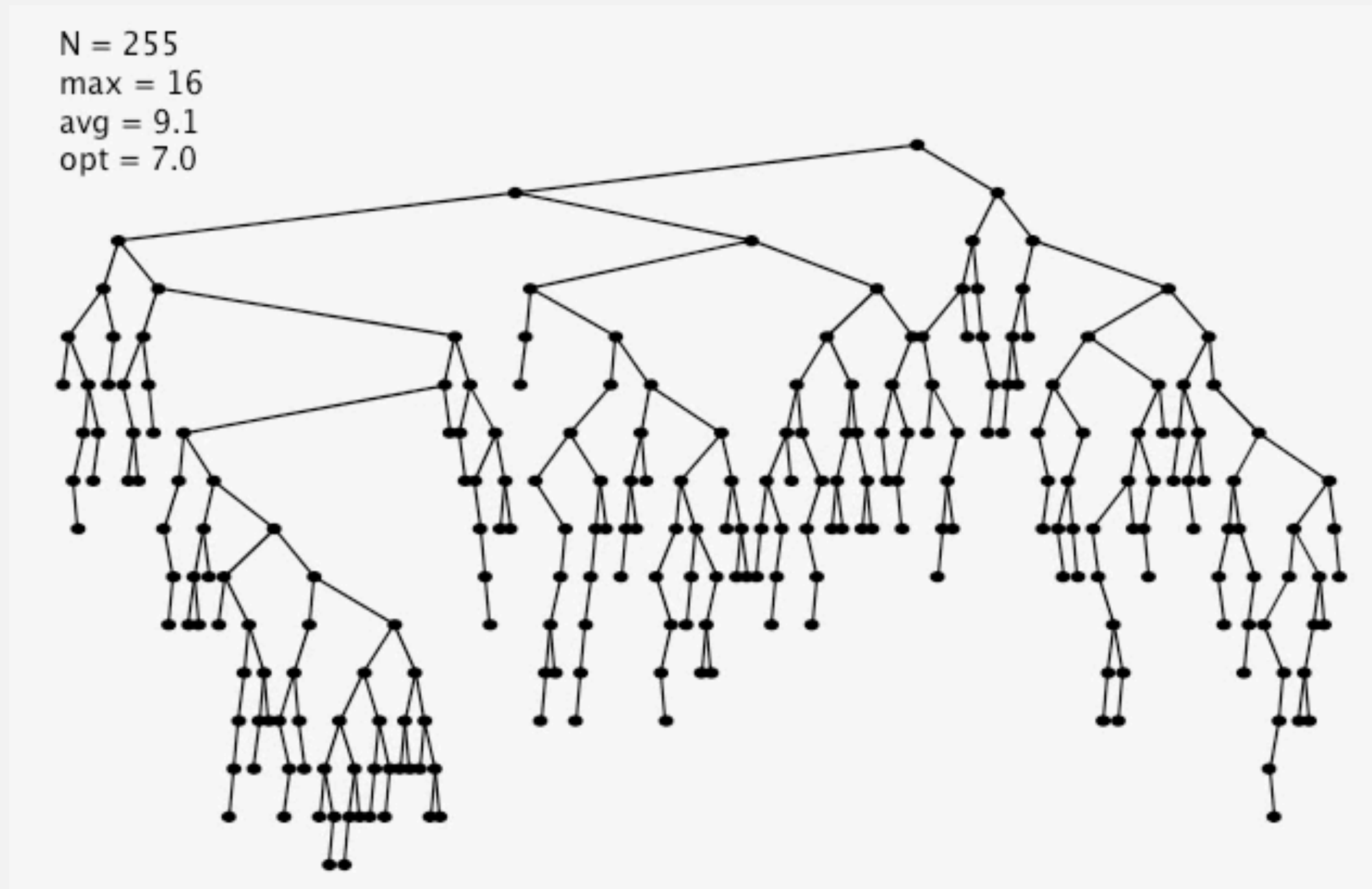


**Bottom line.** Tree shape depends on order of insertion.

# BST insertion: random order visualization

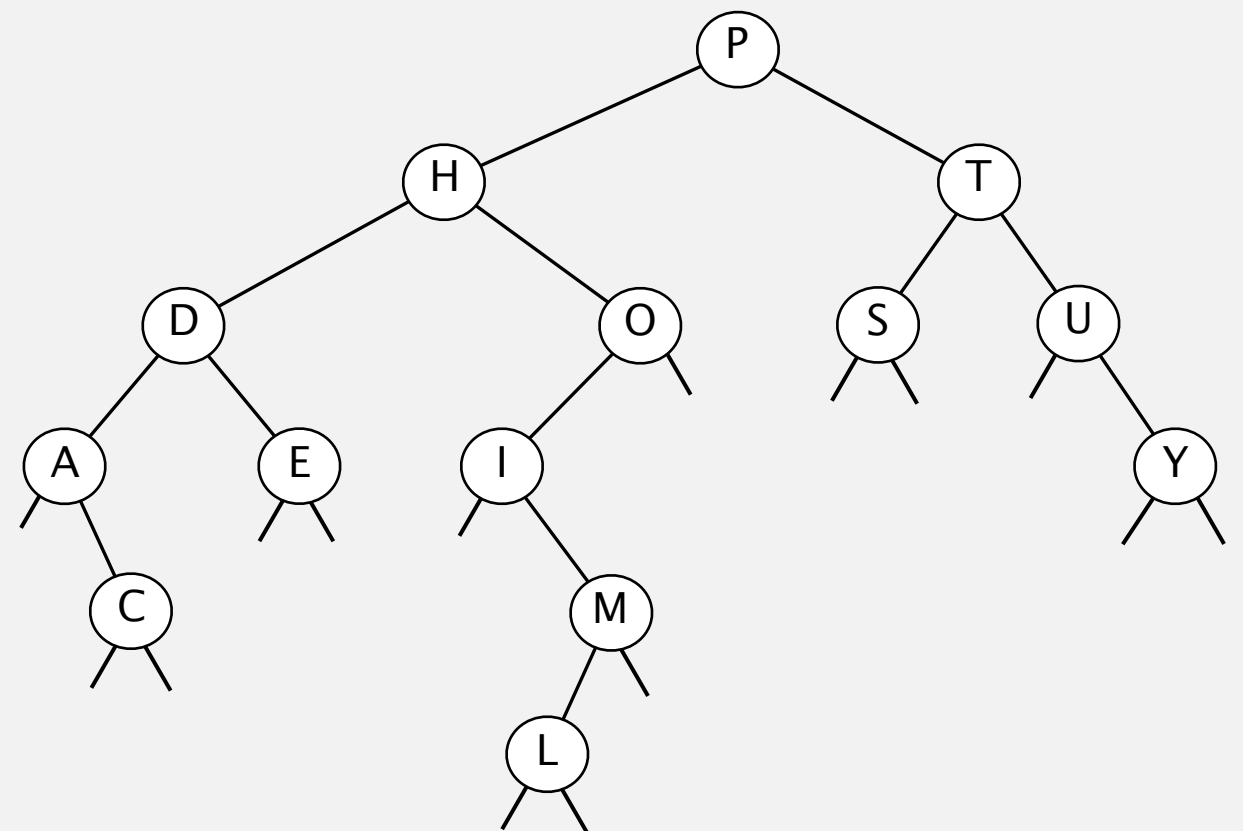
---

Ex. Insert keys in random order.



# Correspondence between BSTs and quicksort partitioning

0	1	2	3	4	5	6	7	8	9	10	11	12	13
P	S	E	U	D	O	M	Y	T	H	I	C	A	L
P	S	E	U	D	O	M	Y	T	H	I	C	A	L
H	L	E	A	D	O	M	C	I	P	T	Y	U	S
D	C	E	A	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	I	M	L	O	P	T	Y	U	S
A	C	D	E	H	I	M	L	O	P	T	Y	U	S
A	C	D	E	H	I	L	M	O	P	T	Y	U	S
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y



**Remark.** Correspondence is 1–1 if array has no duplicate keys.

# BSTs: mathematical analysis


---

**Proposition.** If  $N$  distinct keys are inserted into a BST 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  $N$  distinct keys are inserted into a BST in random order, the expected height is  $\sim 4.311 \ln N$ .

expected depth of  
function-call stack in quicksort



## How Tall is a Tree?

Bruce Reed  
CNRS, Paris, France  
reed@moka.ccr.jussieu.fr

### ABSTRACT

Let  $H_n$  be the height of a random binary search tree on  $n$  nodes. We show that there exists constants  $\alpha = 4.31107\dots$  and  $\beta = 1.95\dots$  such that  $\mathbf{E}(H_n) = \alpha \log n - \beta \log \log n + O(1)$ . We also show that  $\text{Var}(H_n) = O(1)$ .

**But...** Worst-case height is  $N - 1$ .

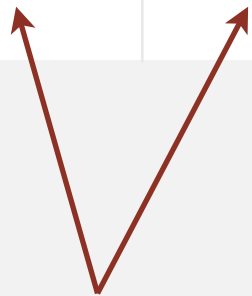
[ exponentially small chance when keys are inserted in random order ]



# ST implementations: summary

---

implementation	guarantee		average case		operations on keys
	search	insert	search hit	insert	
<b>sequential search</b> (unordered list)	$N$	$N$	$N$	$N$	<code>equals()</code>
<b>binary search</b> (ordered array)	$\log N$	$N$	$\log N$	$N$	<code>compareTo()</code>
<b>BST</b>	$N$	$N$	$\log N$	$\log N$	<code>compareTo()</code>



Why not shuffle to ensure a (probabilistic) guarantee of  $\log N$ ?



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

## 3.2 BINARY SEARCH TREES

---

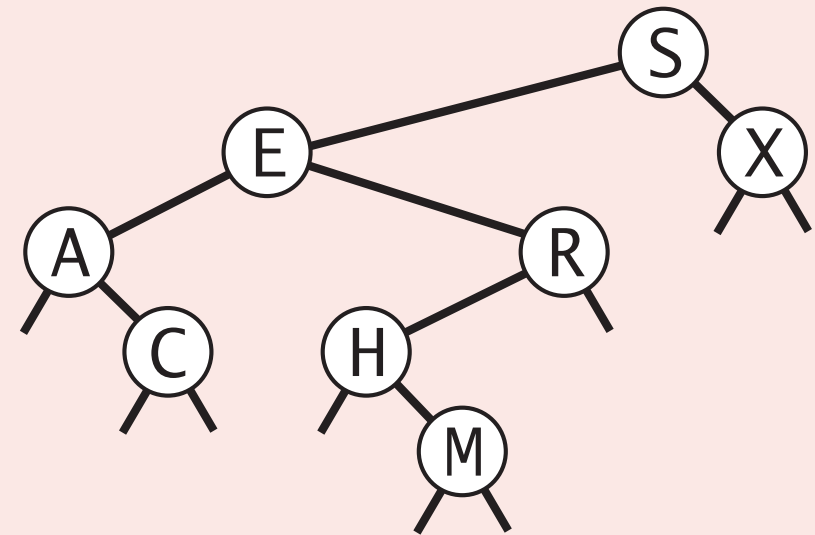
- ▶ *BSTs*
- ▶ *iteration*
- ▶ *ordered operations*
- ▶ *deletion*

## Binary search trees: quiz 2

---

In what order does the `traverse(root)` code print out the keys in the BST?

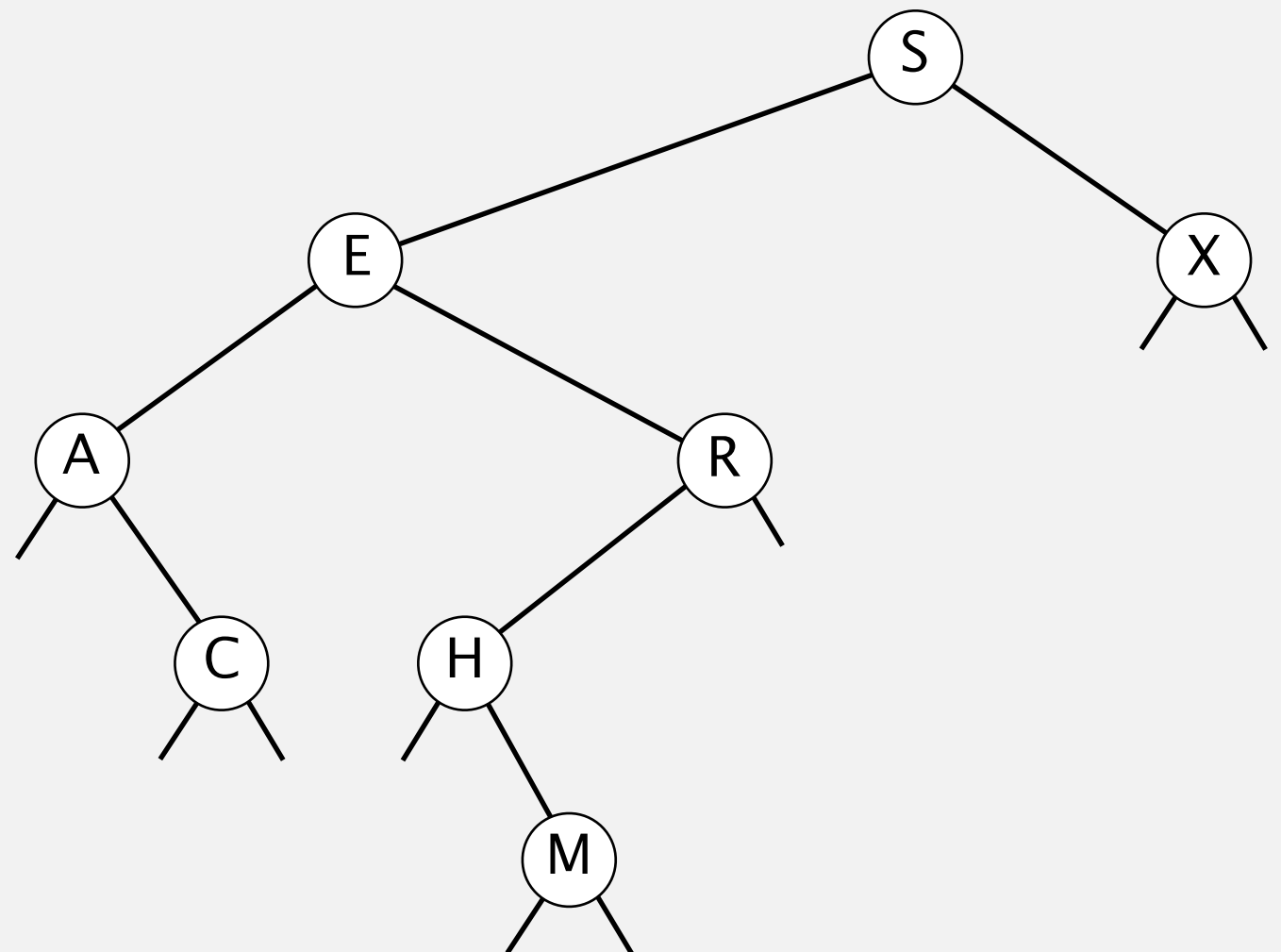
```
private void traverse(Node x)
{
    if (x == null) return;
    traverse(x.left);
    StdOut.println(x.key);
    traverse(x.right);
}
```



- A. A C E H M R S X
- B. A C E R H M X S
- C. S E A C R H M X
- D. C A M H R E X S
- E. *I don't know.*

# Inorder traversal demo

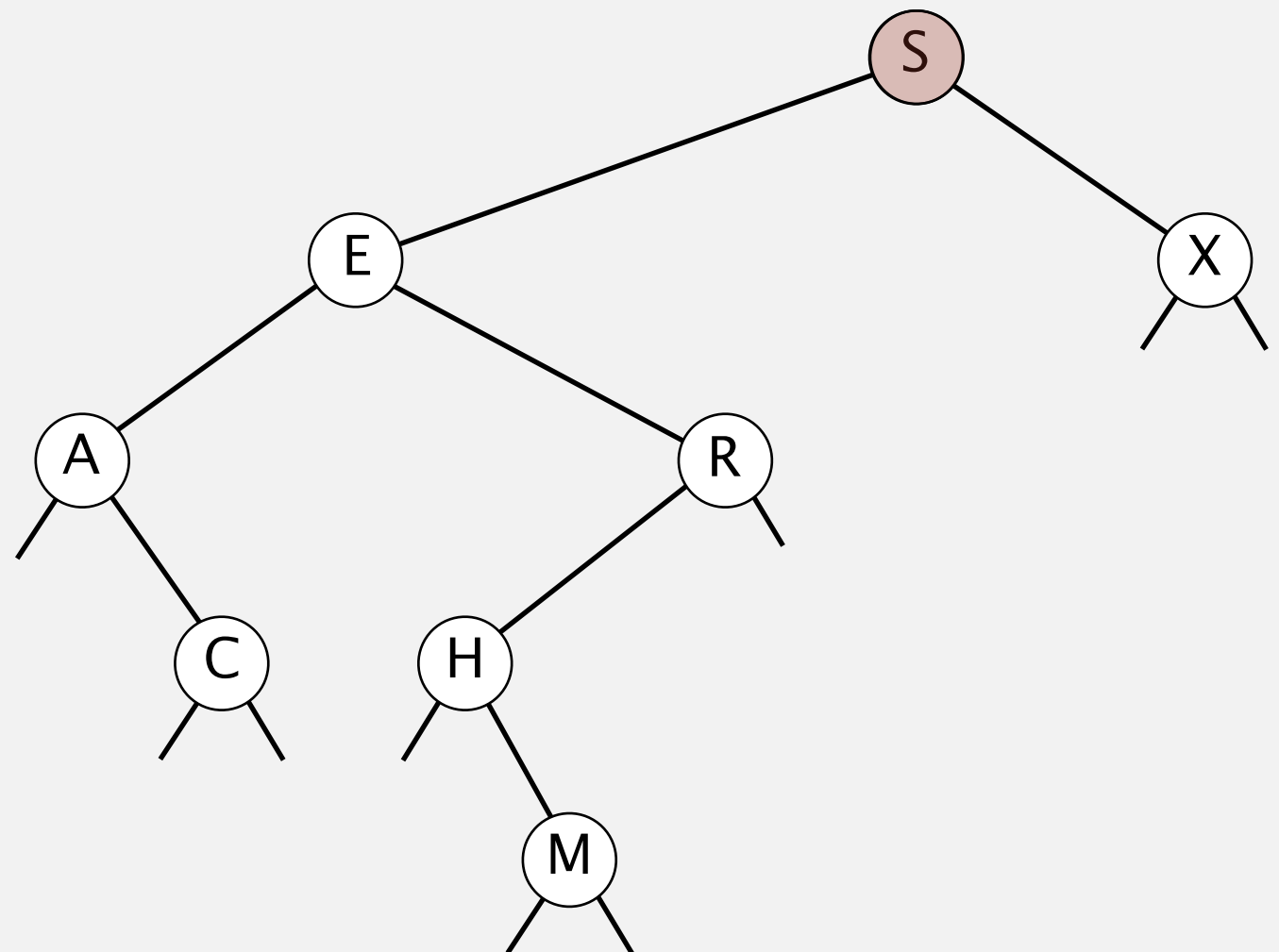
```
private void inorder(Node x)
{
    if (x == null) return;
    inorder(x.left);
    StdOut.println(x.key);
    inorder(x.right);
}
```



# Inorder traversal demo

---

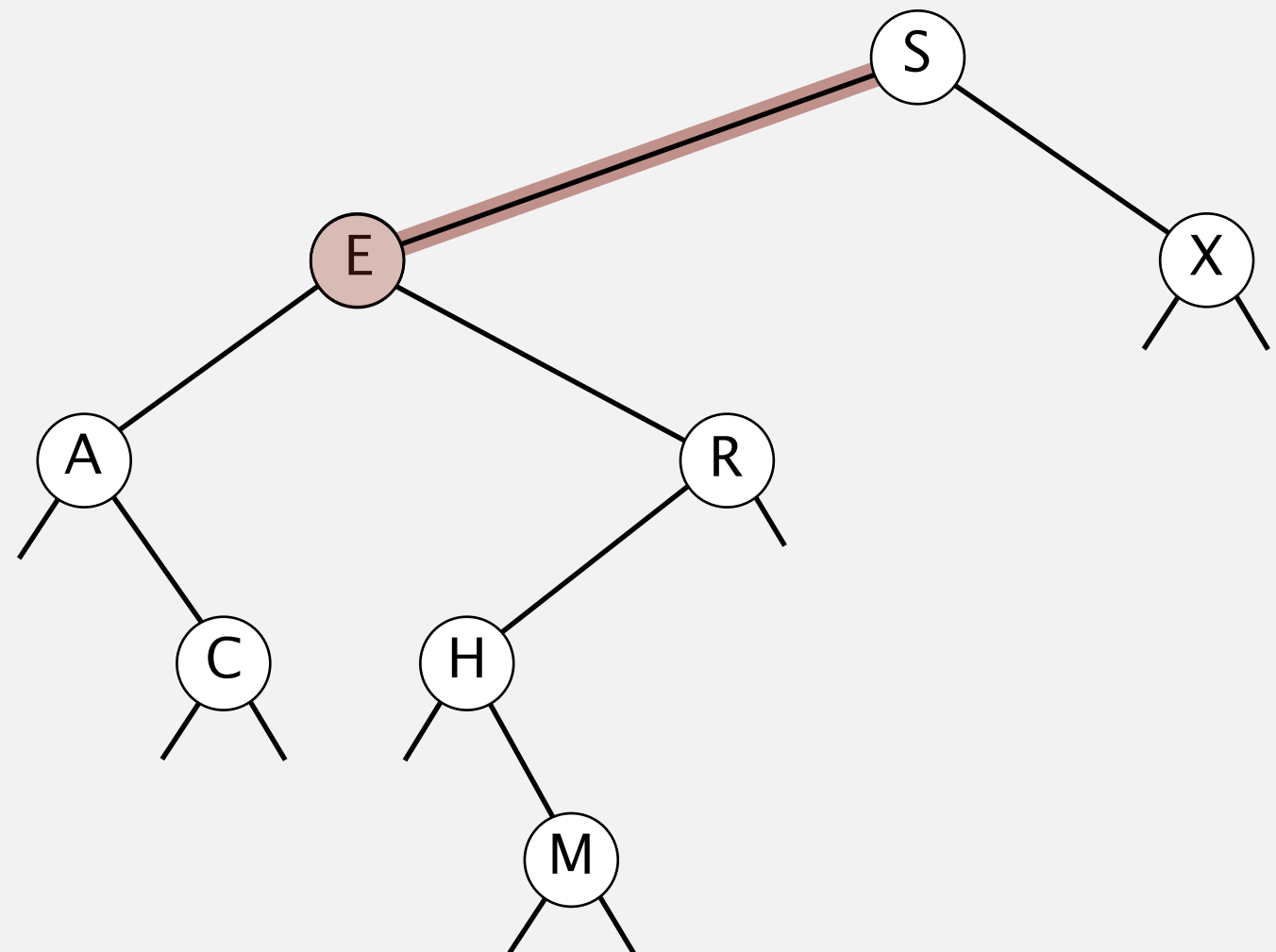
`inorder(S)`



# Inorder traversal demo

---

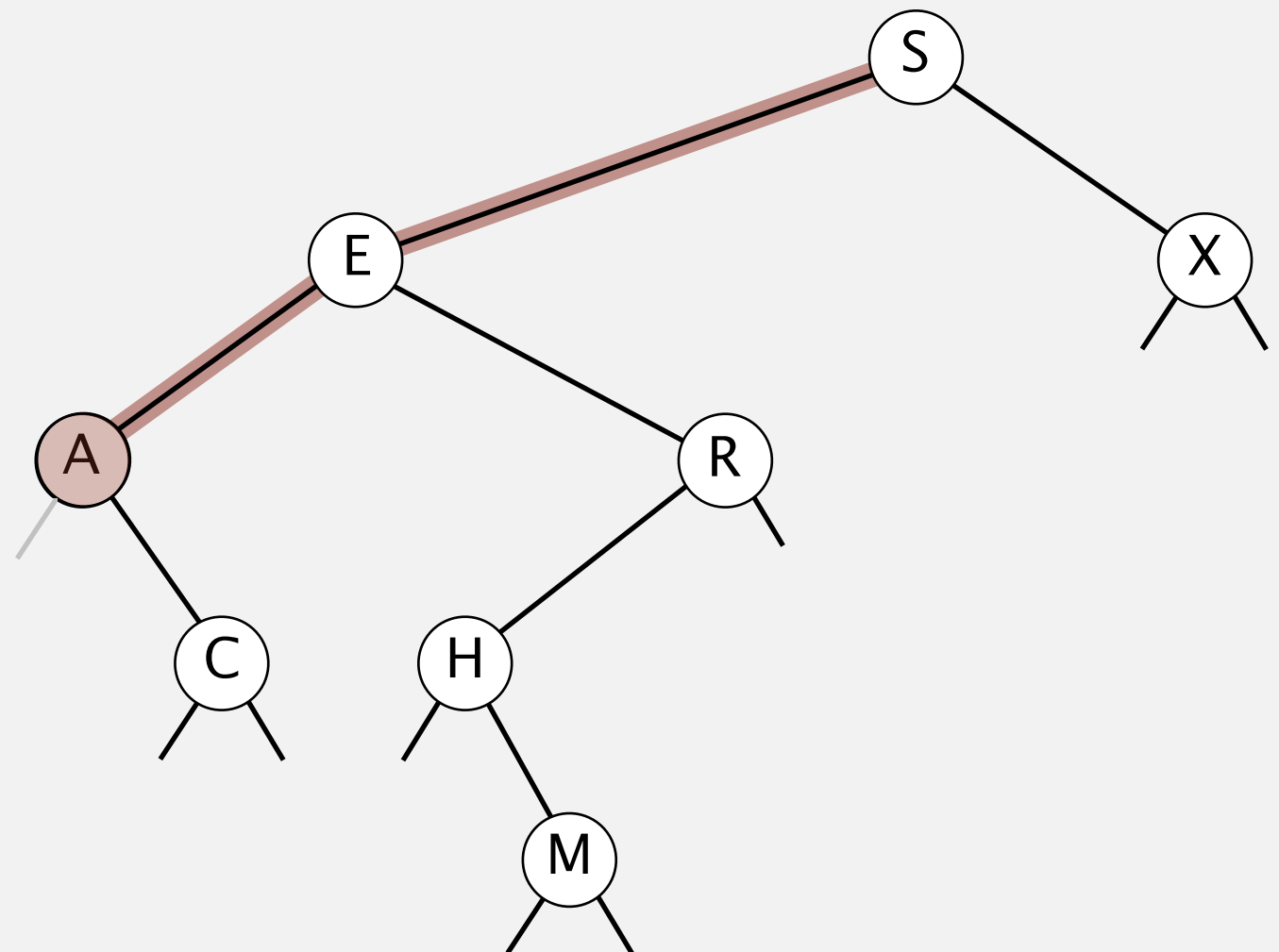
inorder(S)  
  inorder(E)



# Inorder traversal demo

---

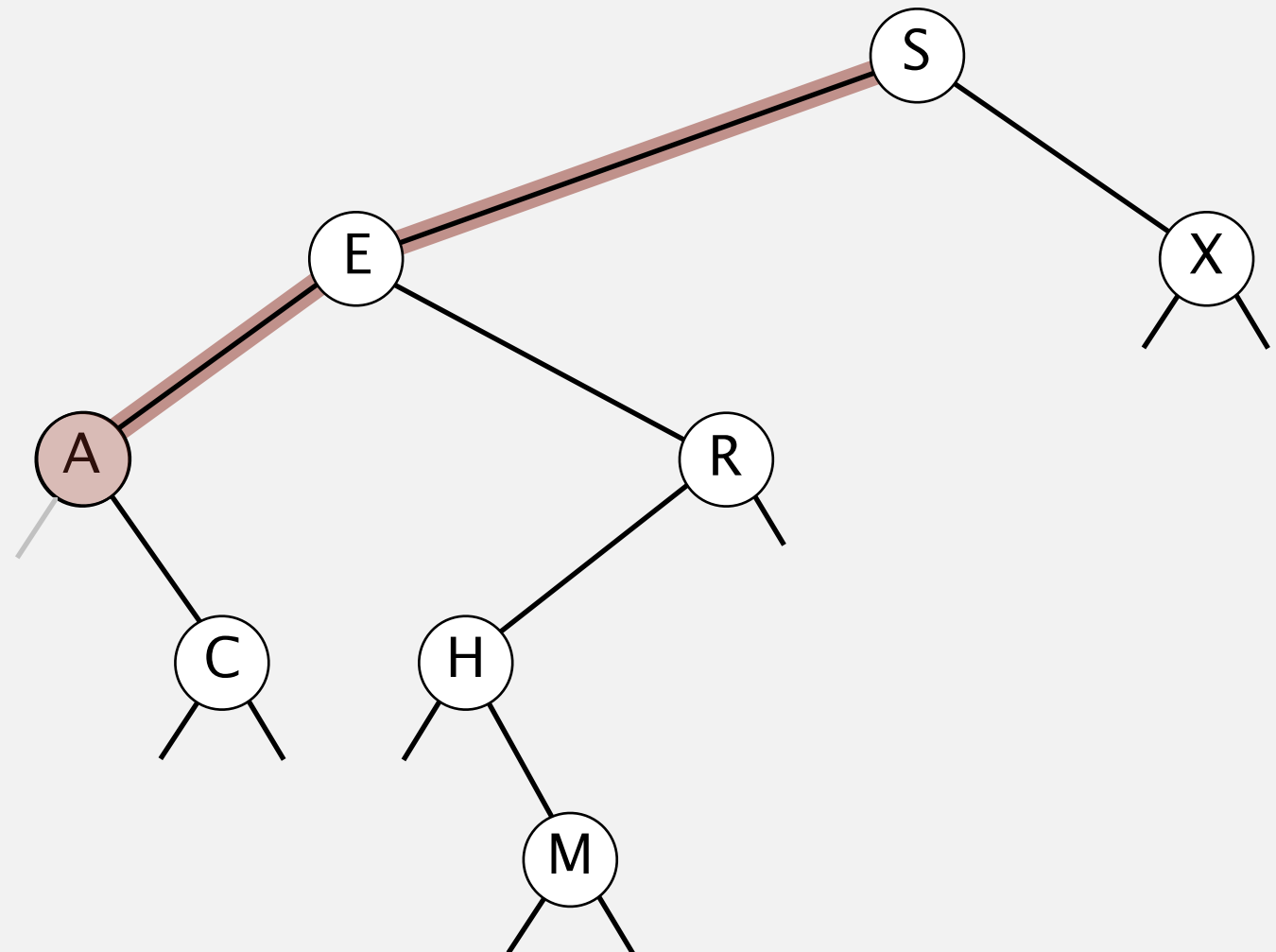
inorder(S)  
  inorder(E)  
    inorder(A)



# Inorder traversal demo

---

```
inorder(S)
  inorder(E)
    inorder(A)
      print A
```



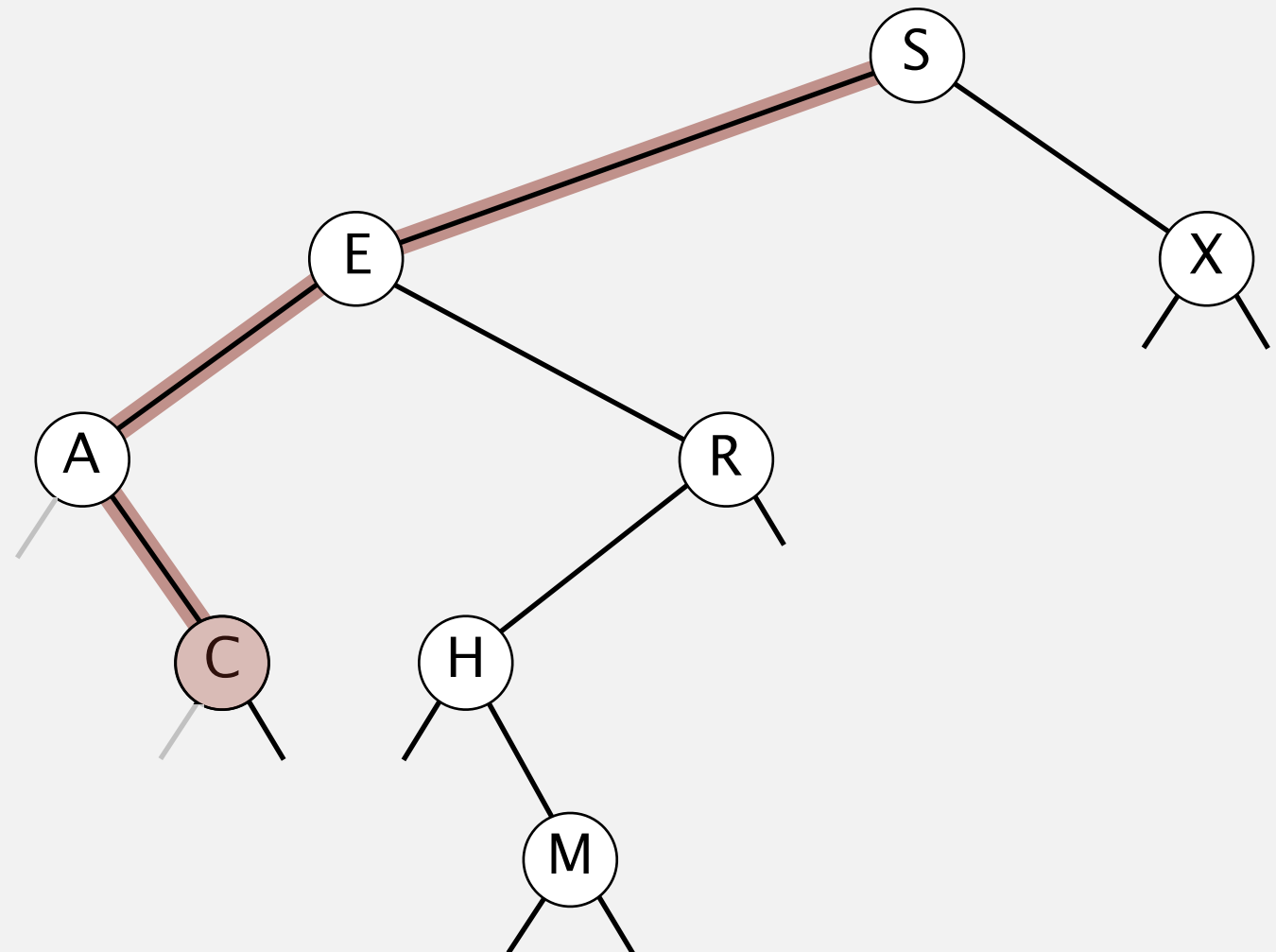
**output: A**



# Inorder traversal demo

---

```
inorder(S)
  inorder(E)
    inorder(A)
      print A
    inorder(C)
```

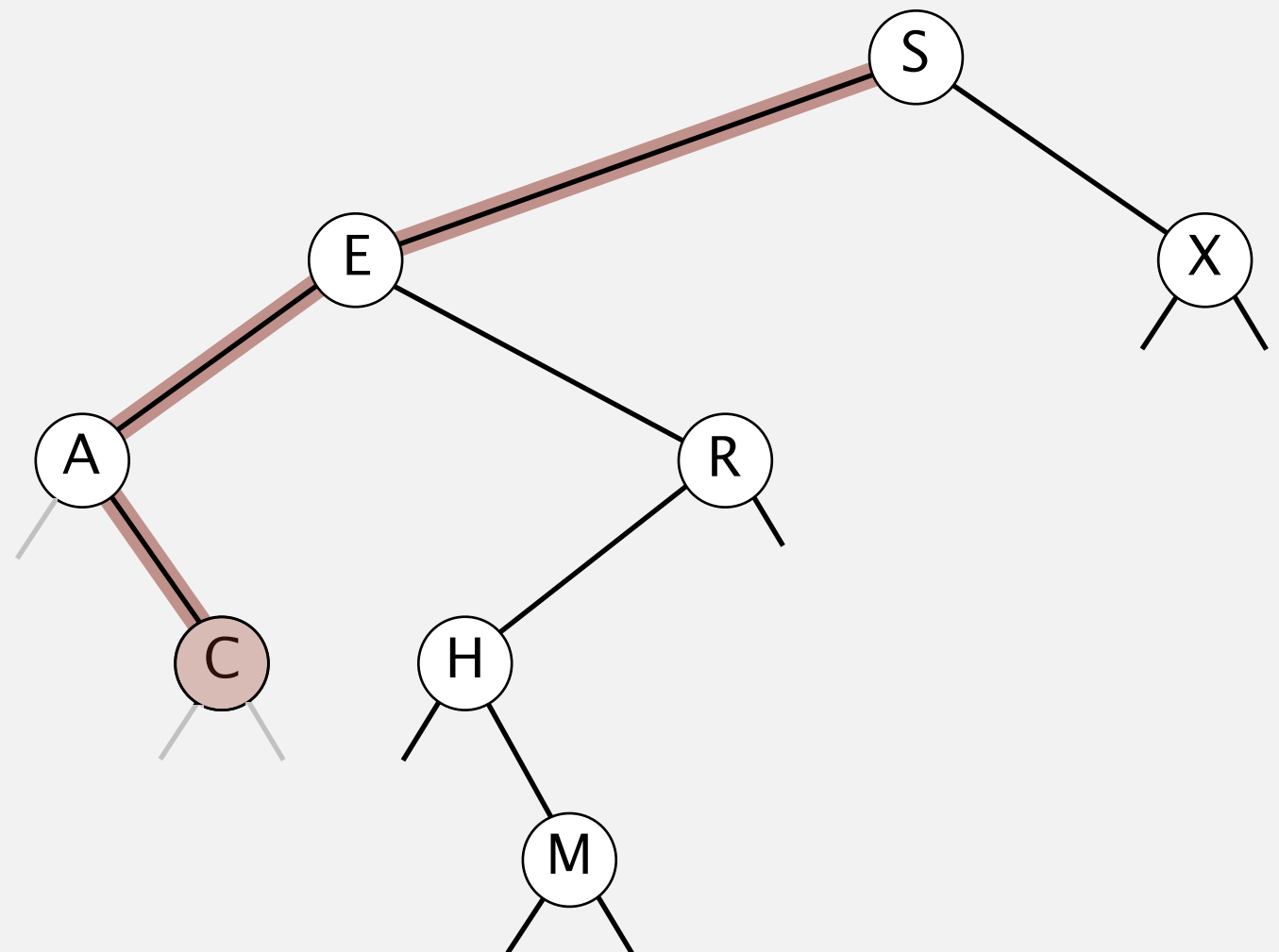


**output: A**

# Inorder traversal demo

---

```
inorder(S)
  inorder(E)
    inorder(A)
      print A
    inorder(C)
      print C
```

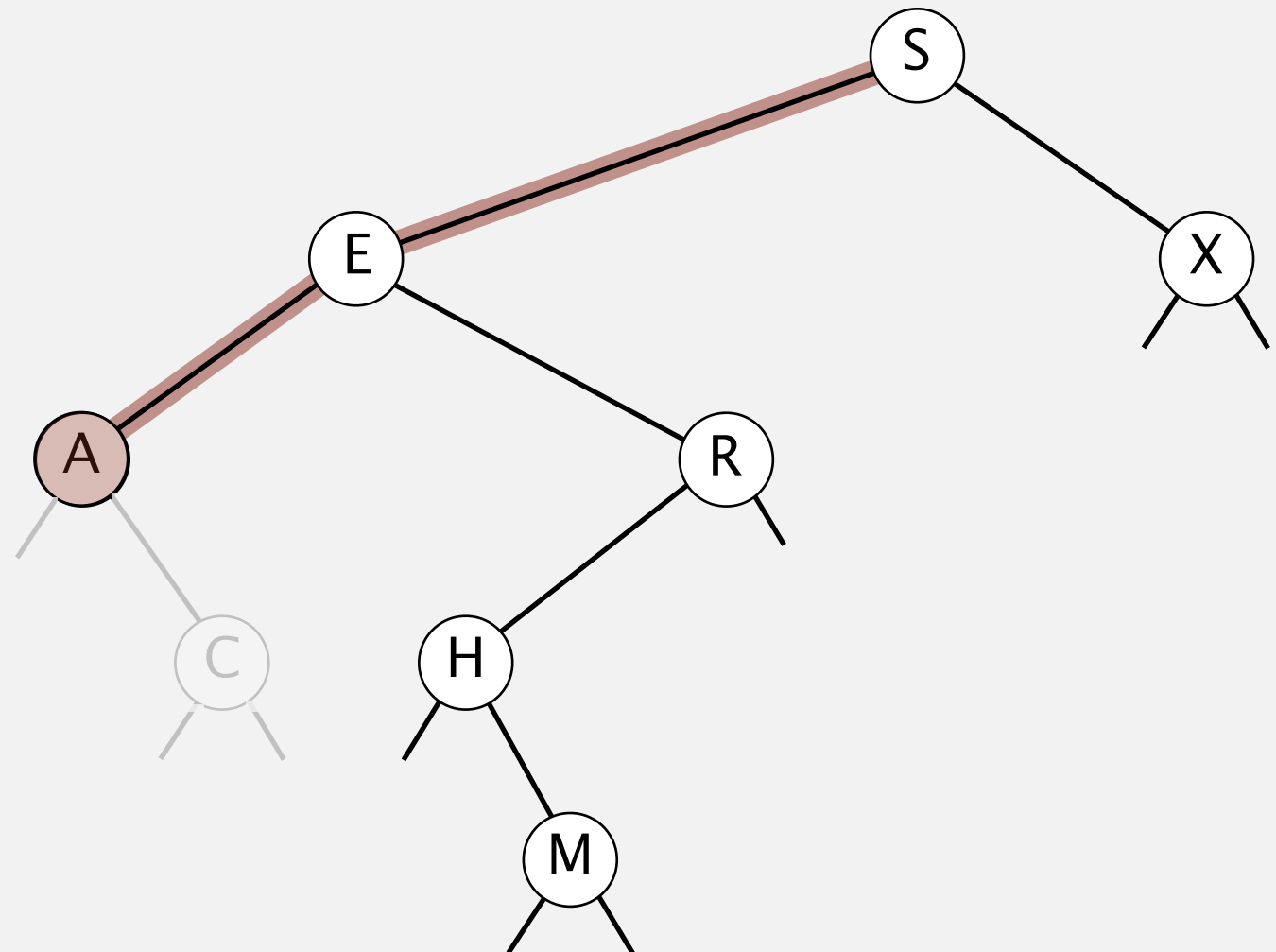


**output: A C**

# Inorder traversal demo

---

```
inorder(S)
  inorder(E)
    inorder(A)
      print A
    inorder(C)
      print C
    done C
```

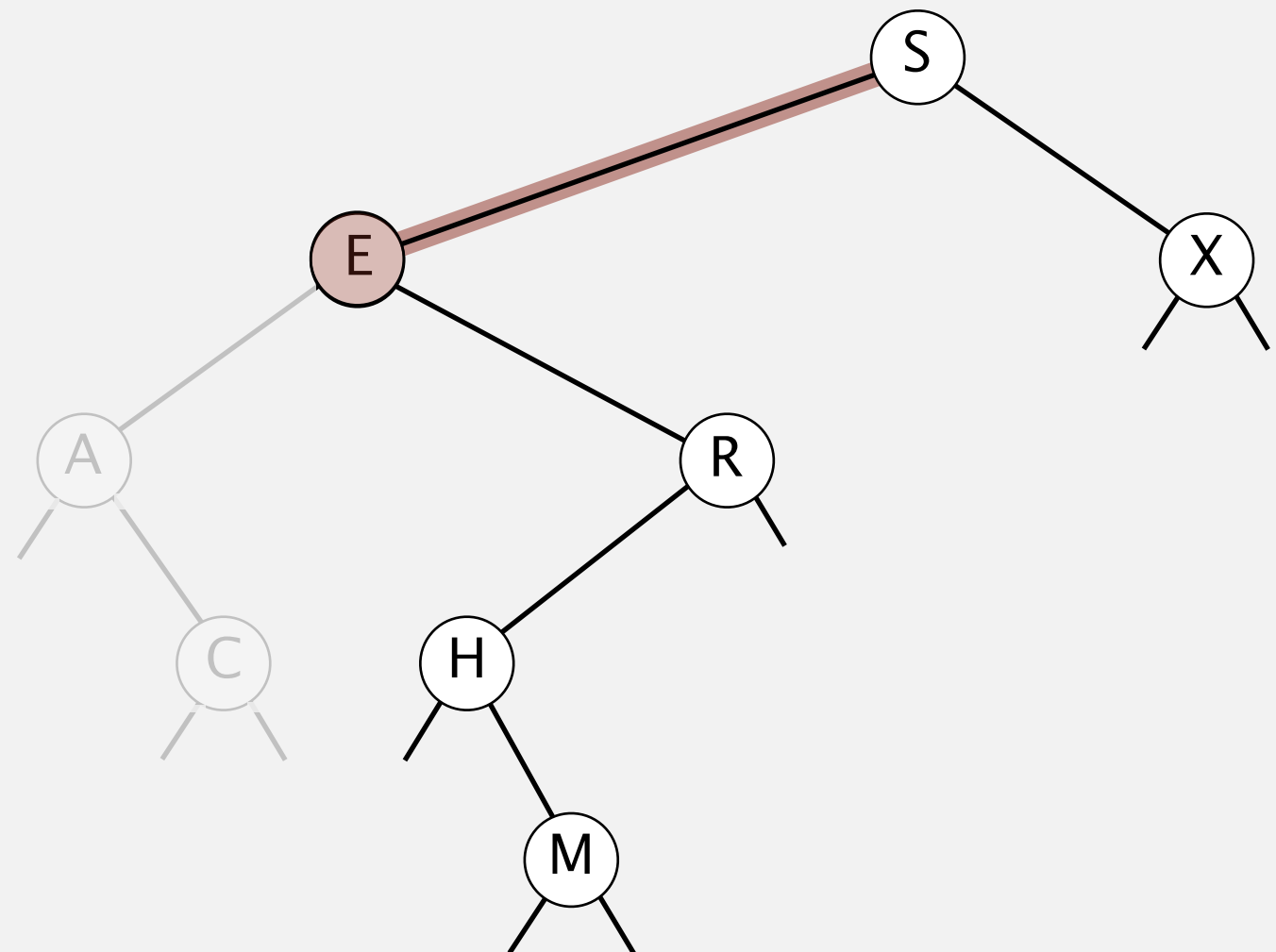


**output: A C**

# Inorder traversal demo

---

```
inorder(S)
  inorder(E)
    inorder(A)
      print A
    inorder(C)
      print C
    done C
  done A
```

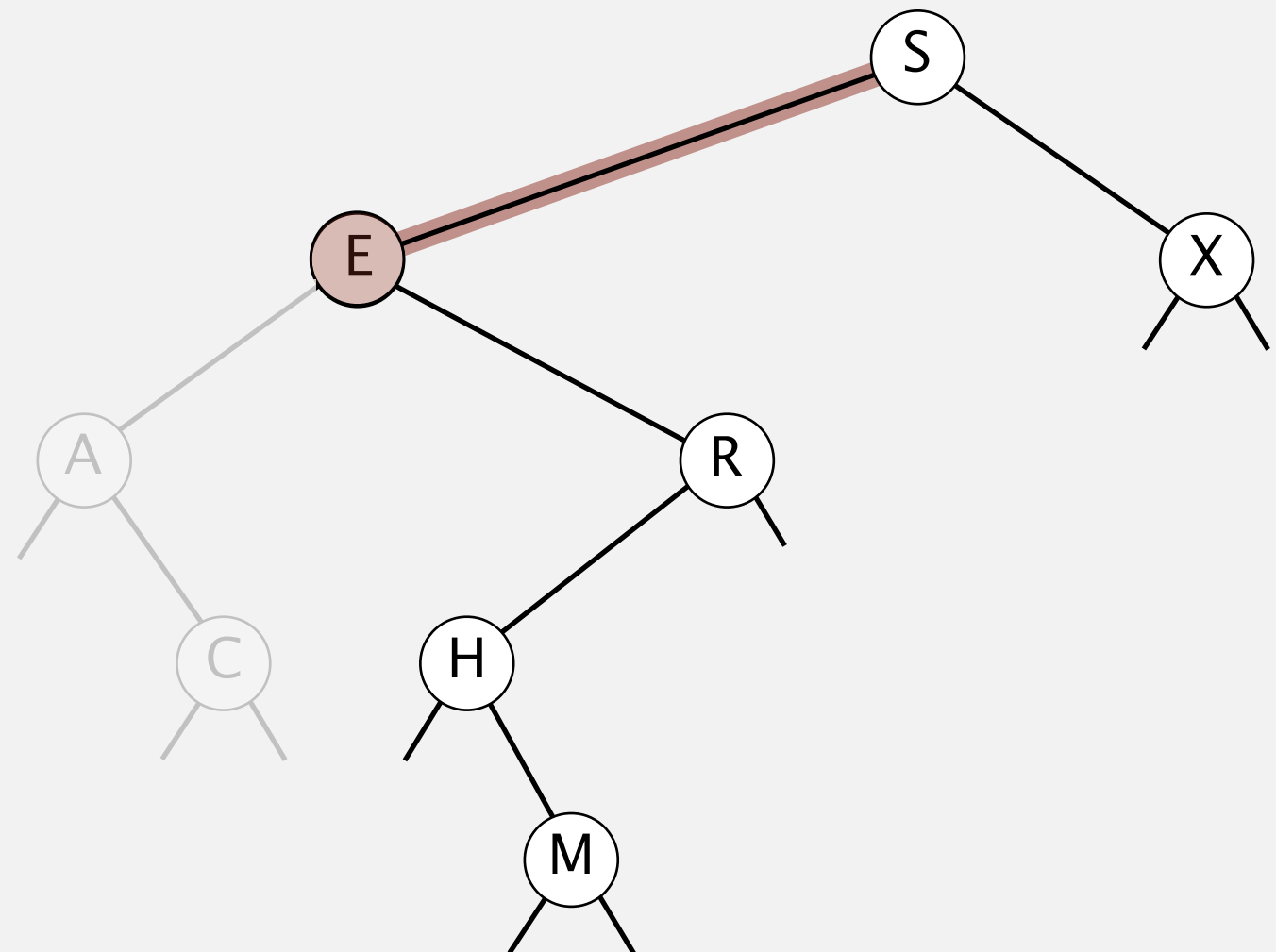


**output: A C**

# Inorder traversal demo

---

```
inorder(S)
  inorder(E)
    inorder(A)
      print A
    inorder(C)
      print C
    done C
  done A
print E
```

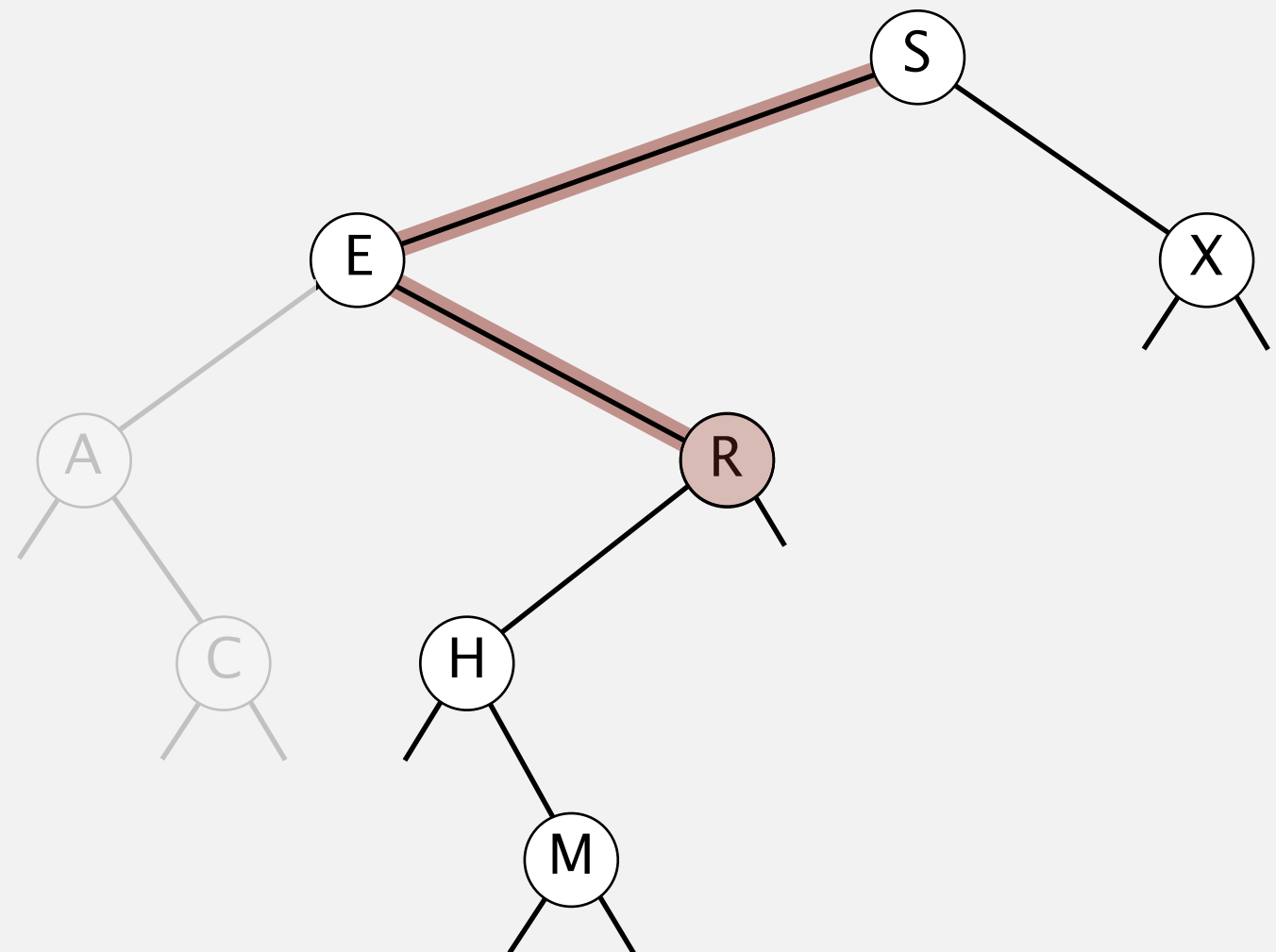


**output: A C E**

# Inorder traversal demo

---

```
inorder(S)
  inorder(E)
    inorder(A)
      print A
    inorder(C)
      print C
    done C
  done A
print E
inorder(R)
```

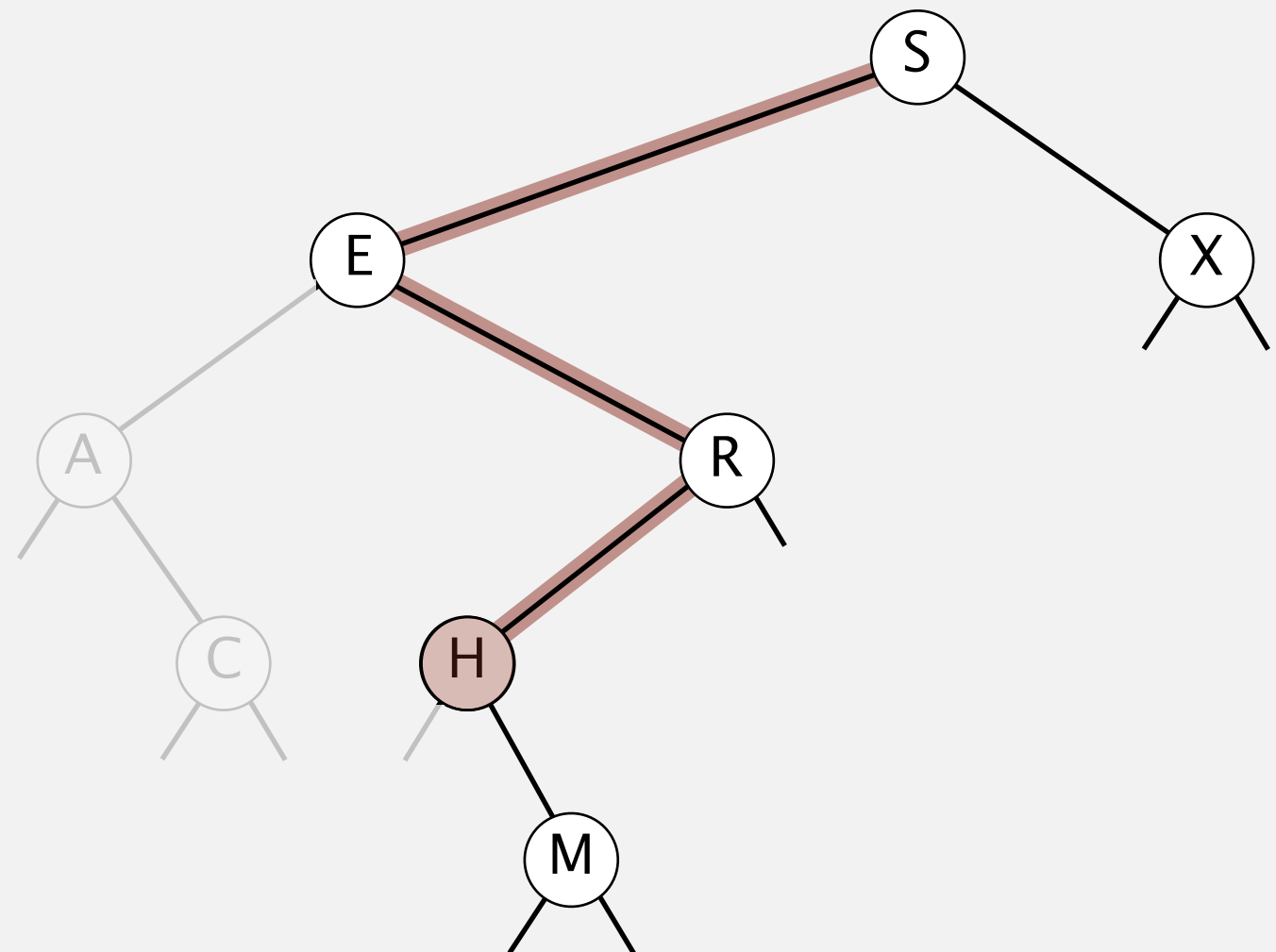


**output: A C E**

# Inorder traversal demo

---

```
inorder(S)
  inorder(E)
    inorder(A)
      print A
      inorder(C)
        print C
        done C
      done A
    print E
    inorder(R)
      inorder(H)
```

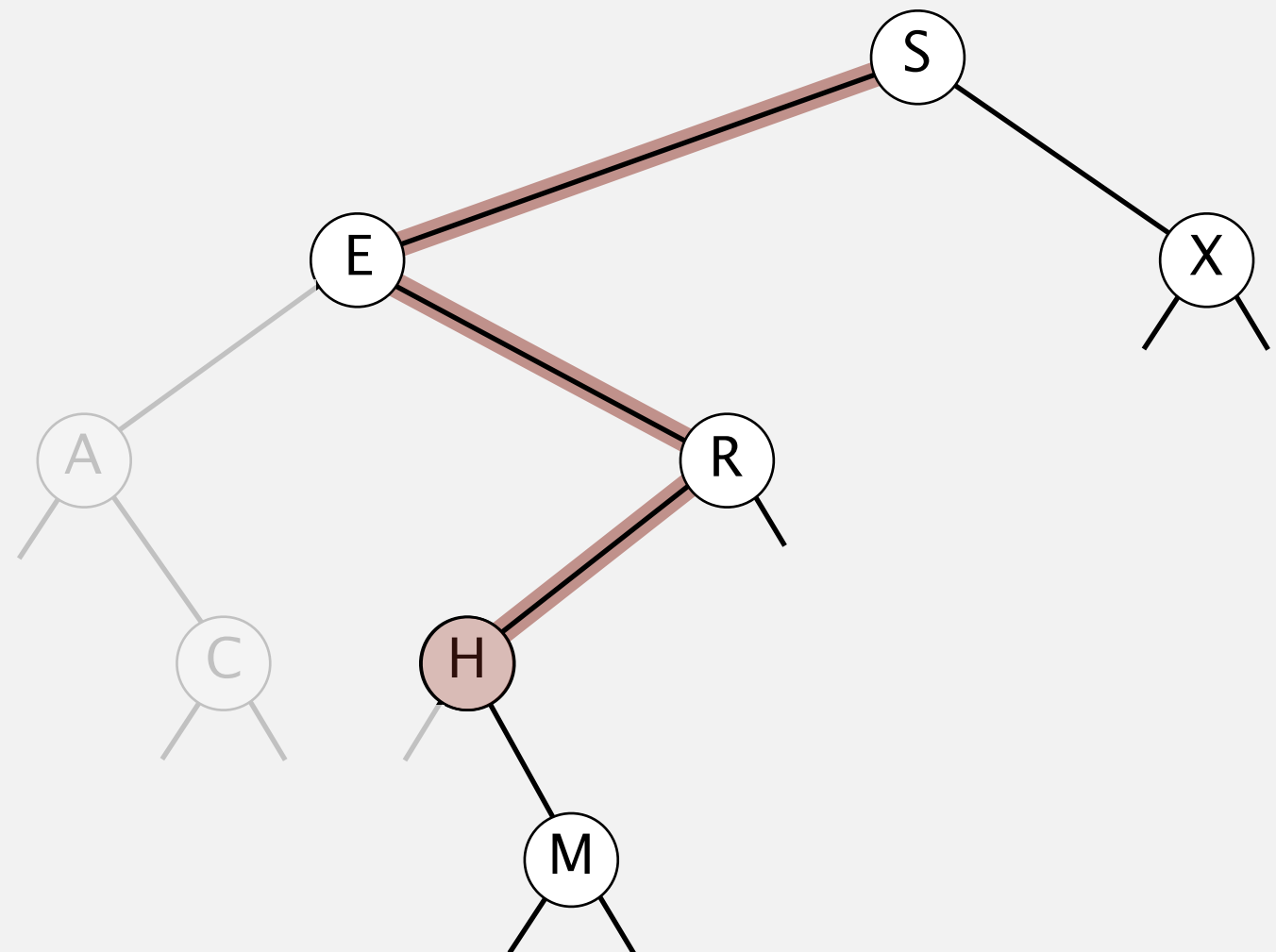


**output: A C E**

# Inorder traversal demo

---

```
inorder(S)
  inorder(E)
    inorder(A)
      print A
    inorder(C)
      print C
    done C
  done A
print E
inorder(R)
  inorder(H)
    print H
```



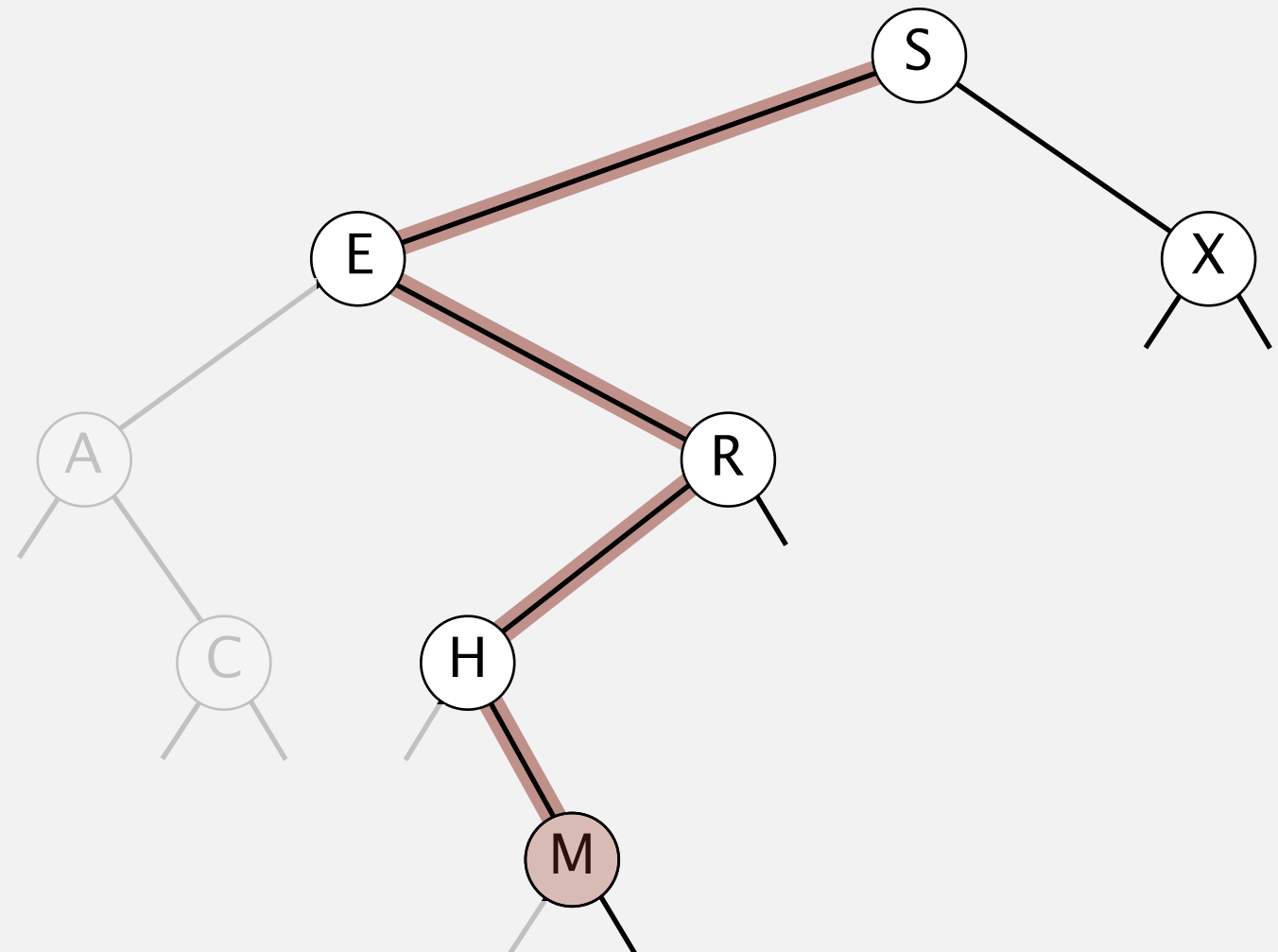
**output: A C E H**



# Inorder traversal demo

---

```
inorder(S)
  inorder(E)
    inorder(A)
      print A
      inorder(C)
        print C
        done C
      done A
    print E
    inorder(R)
      inorder(H)
        print H
        inorder(M)
```

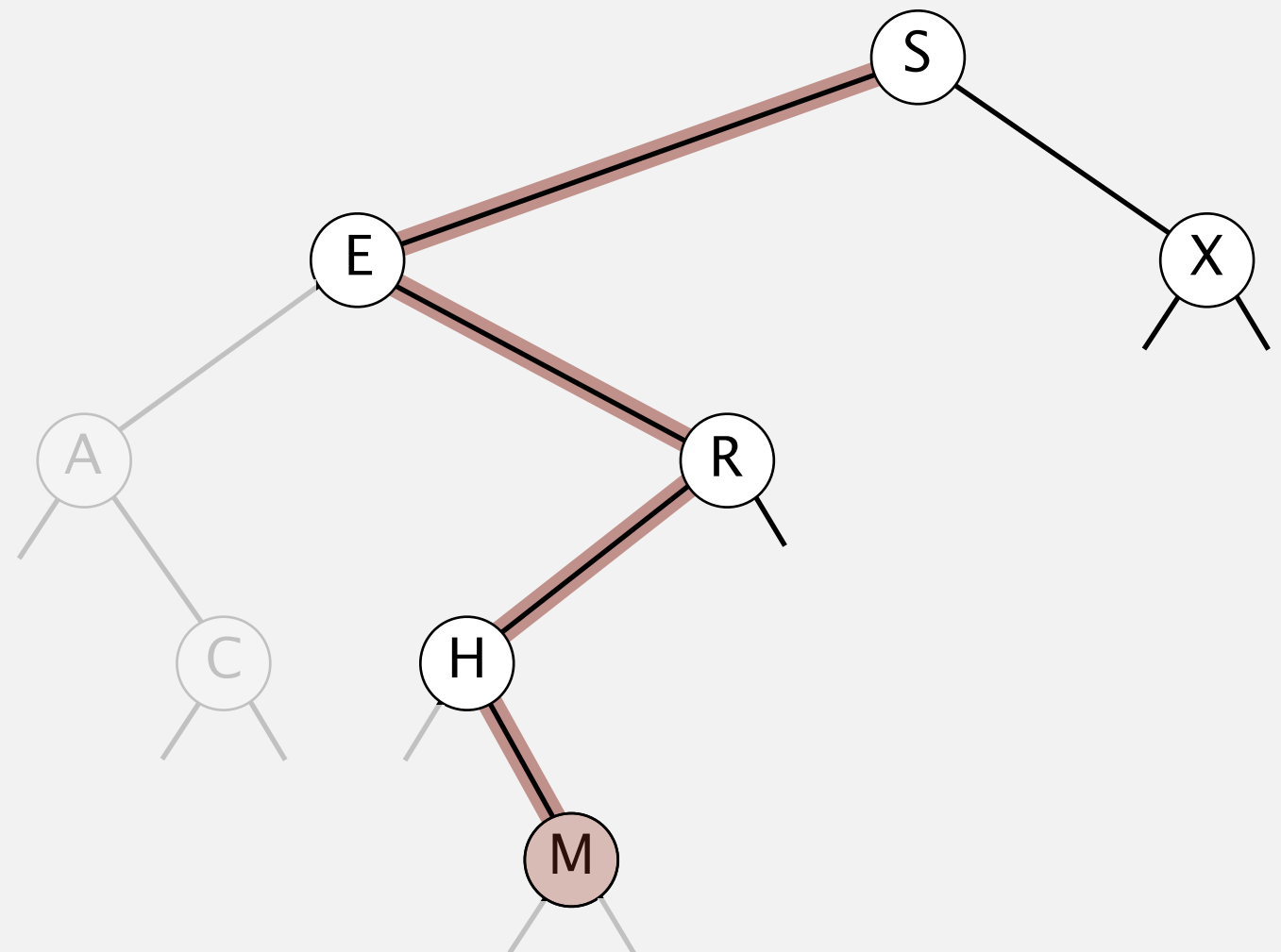


output: **A C E H**

# Inorder traversal demo

---

```
inorder(S)
  inorder(E)
    inorder(A)
      print A
      inorder(C)
        print C
        done C
      done A
    print E
    inorder(R)
      inorder(H)
        print H
        inorder(M)
          print M
```

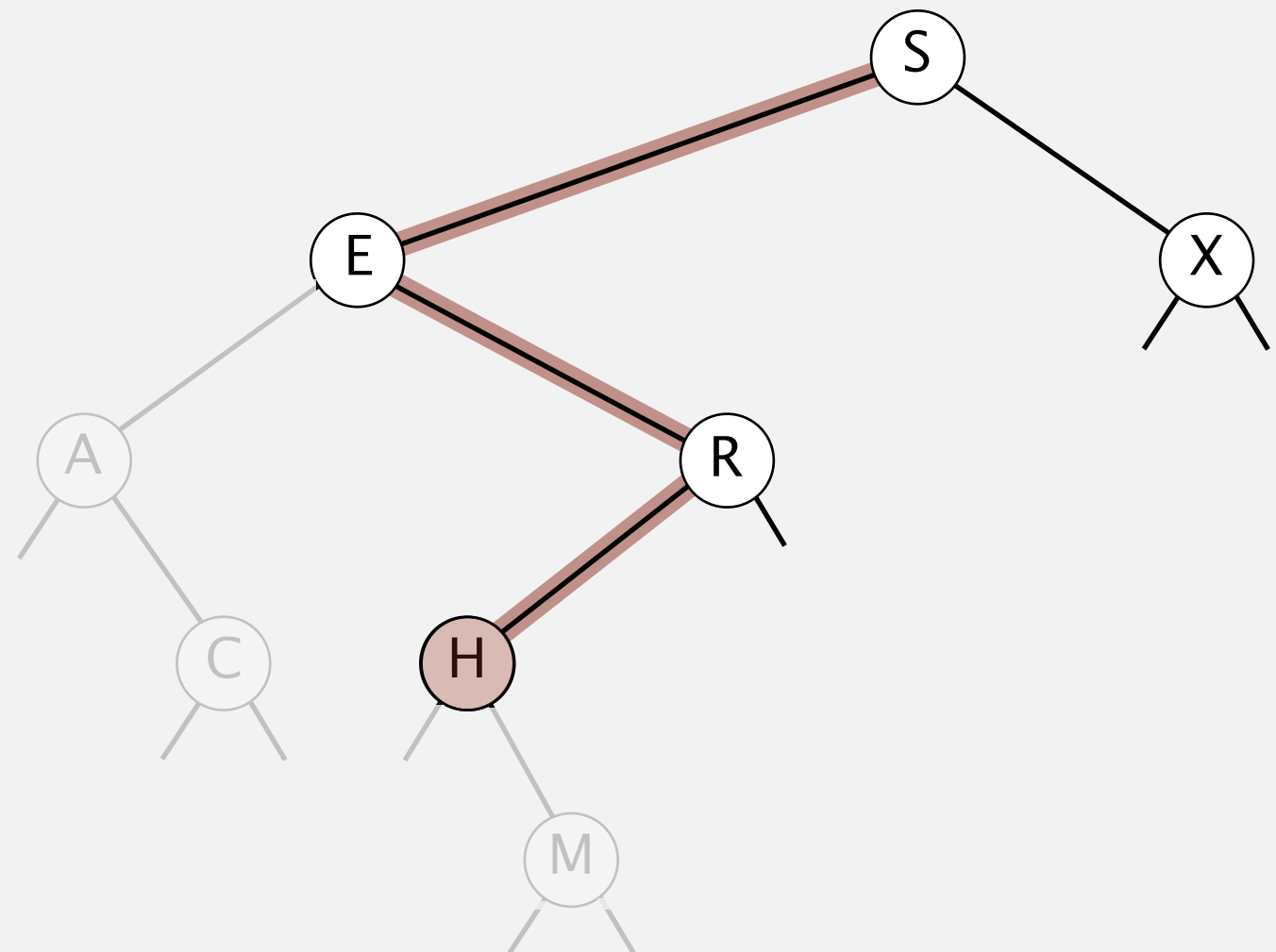


output: **A C E H M**

# Inorder traversal demo

---

```
inorder(S)
  inorder(E)
    inorder(A)
      print A
      inorder(C)
        print C
        done C
      done A
    print E
    inorder(R)
      inorder(H)
        print H
        inorder(M)
          print M
          done M
        done H
      done R
    done E
  done S
```

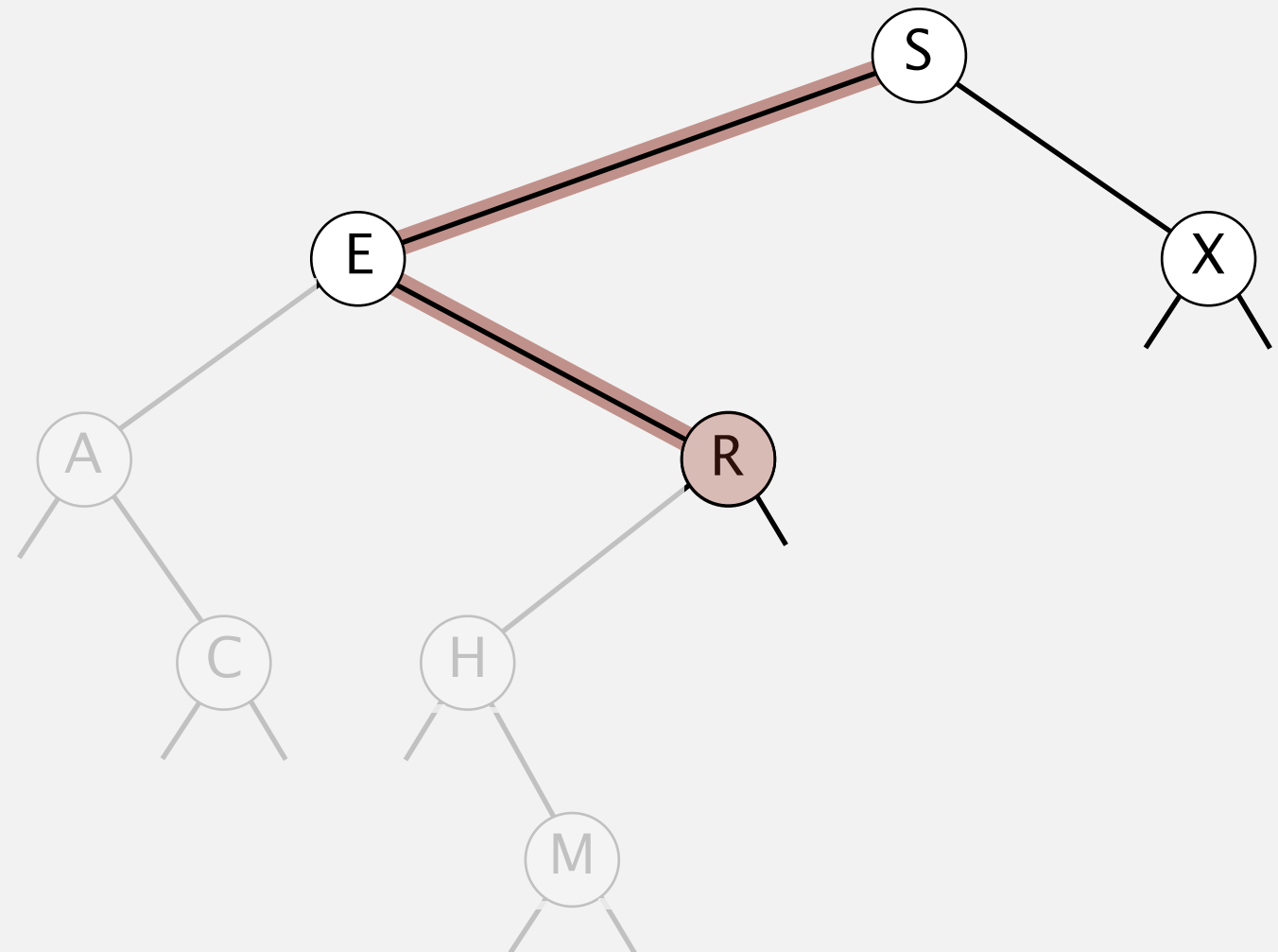


output: **A C E H M**

# Inorder traversal demo

---

```
inorder(S)
  inorder(E)
    inorder(A)
      print A
      inorder(C)
        print C
        done C
      done A
    print E
    inorder(R)
      inorder(H)
        print H
        inorder(M)
          print M
          done M
        done H
      done R
    done E
  done S
```

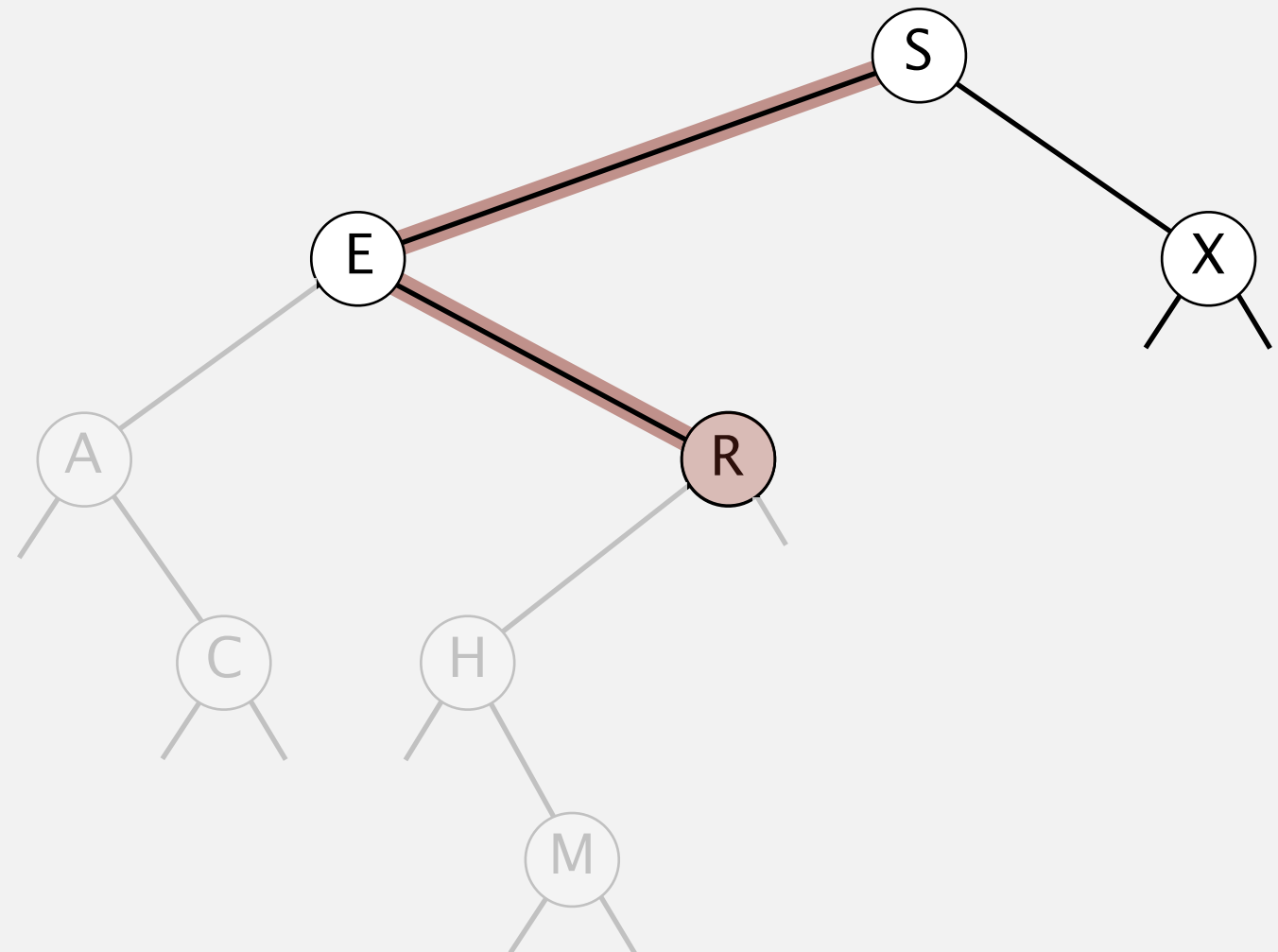


**output: A C E H M**

# Inorder traversal demo

---

```
inorder(S)
  inorder(E)
    inorder(A)
      print A
      inorder(C)
        print C
        done C
      done A
    print E
    inorder(R)
      inorder(H)
        print H
        inorder(M)
          print M
          done M
        done H
      print R
```

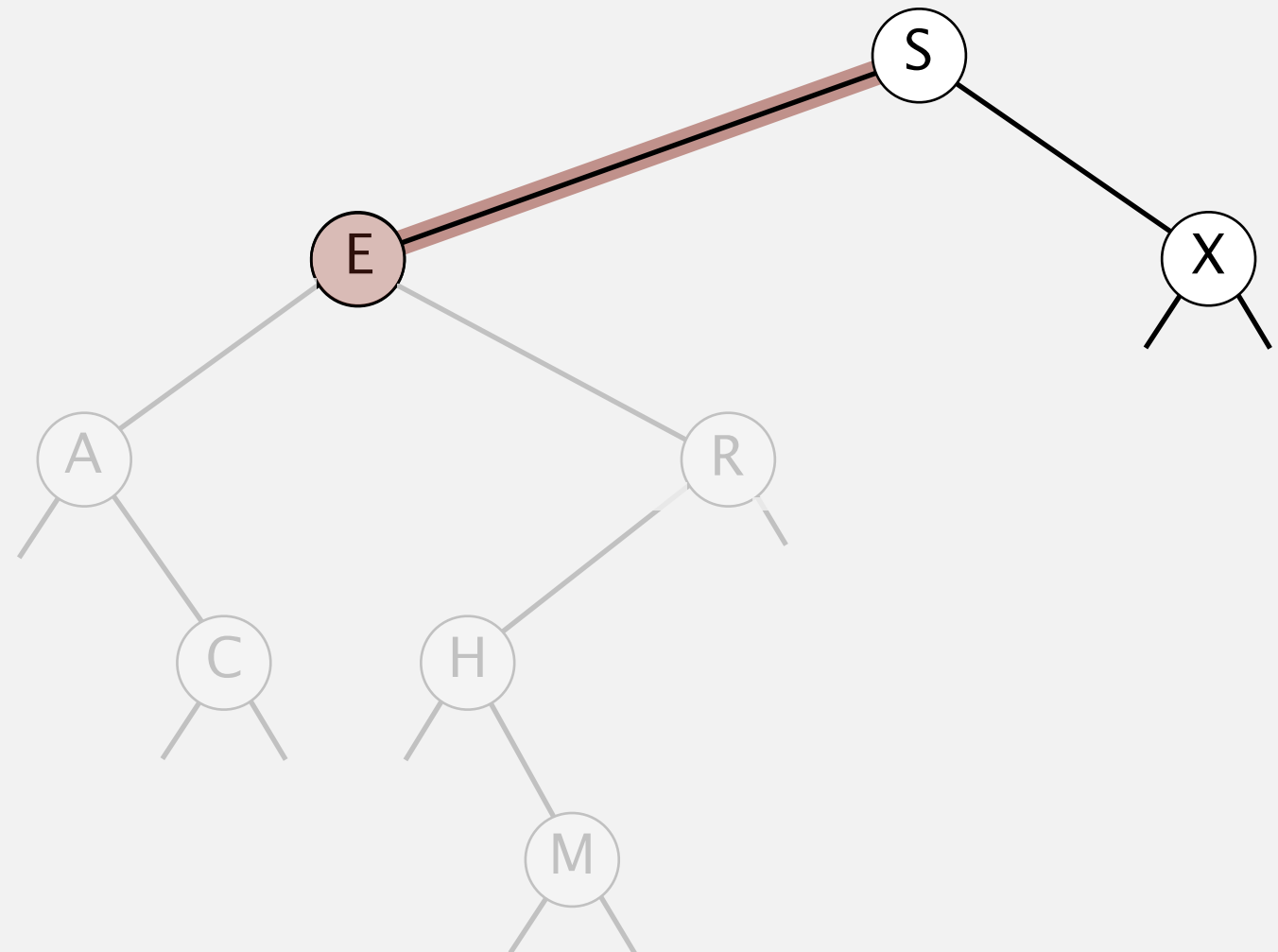


**output: A C E H M R**

# Inorder traversal demo

---

```
inorder(S)
  inorder(E)
    inorder(A)
      print A
      inorder(C)
        print C
        done C
      done A
    print E
    inorder(R)
      inorder(H)
        print H
        inorder(M)
          print M
          done M
        done H
      print R
      done R
```

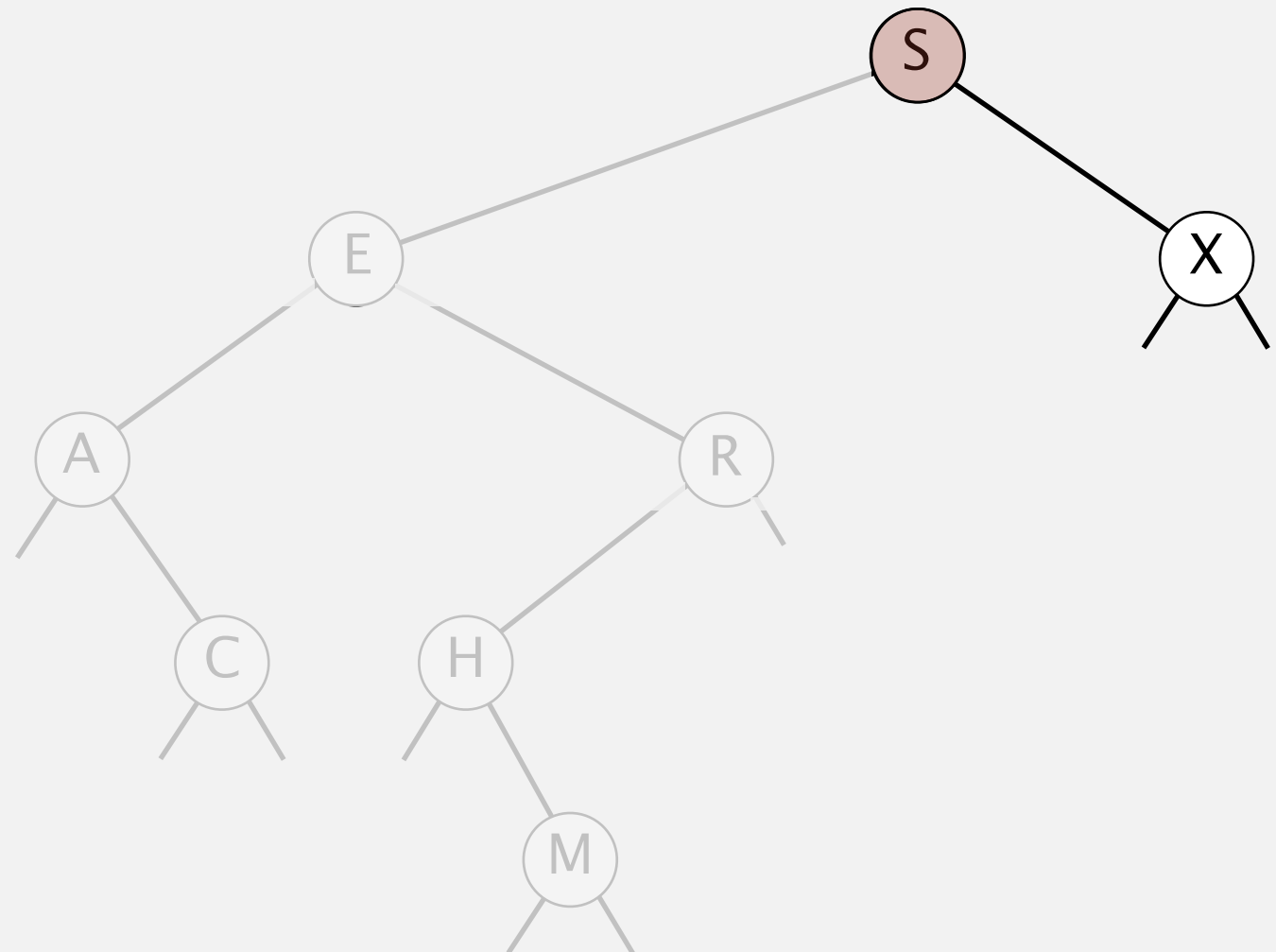


**output: A C E H M R**

# Inorder traversal demo

---

```
inorder(S)
  inorder(E)
    inorder(A)
      print A
      inorder(C)
        print C
        done C
      done A
    print E
    inorder(R)
      inorder(H)
        print H
        inorder(M)
          print M
          done M
        done H
      print R
      done R
    done E
```

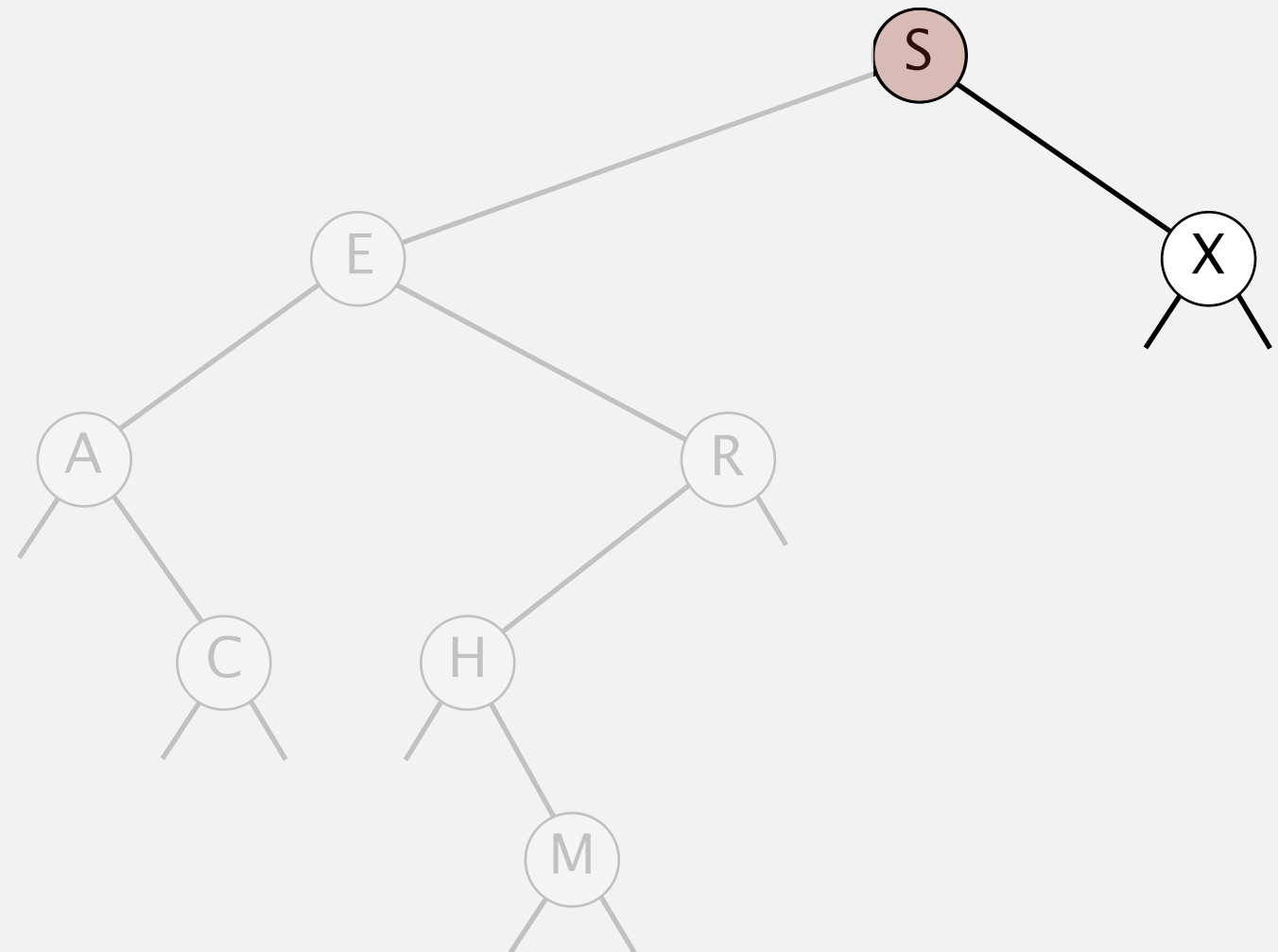


**output: A C E H M R**

# Inorder traversal demo

---

```
inorder(S)
  inorder(E)
    inorder(A)
      print A
      inorder(C)
        print C
        done C
      done A
    print E
    inorder(R)
      inorder(H)
        print H
        inorder(M)
          print M
          done M
        done H
      print R
      done R
    done E
  print S
```



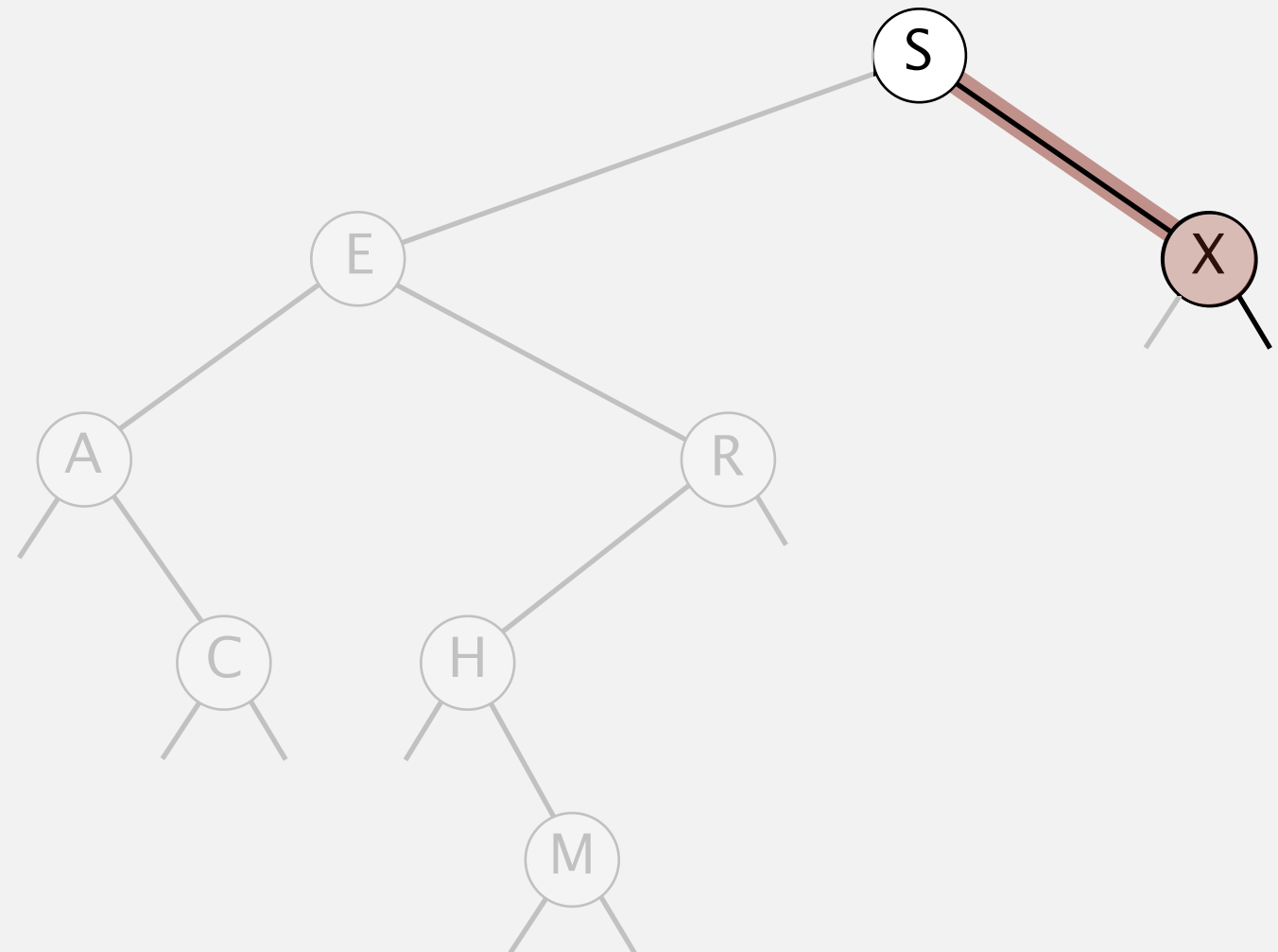
**output: A C E H M R S**



# Inorder traversal demo

---

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

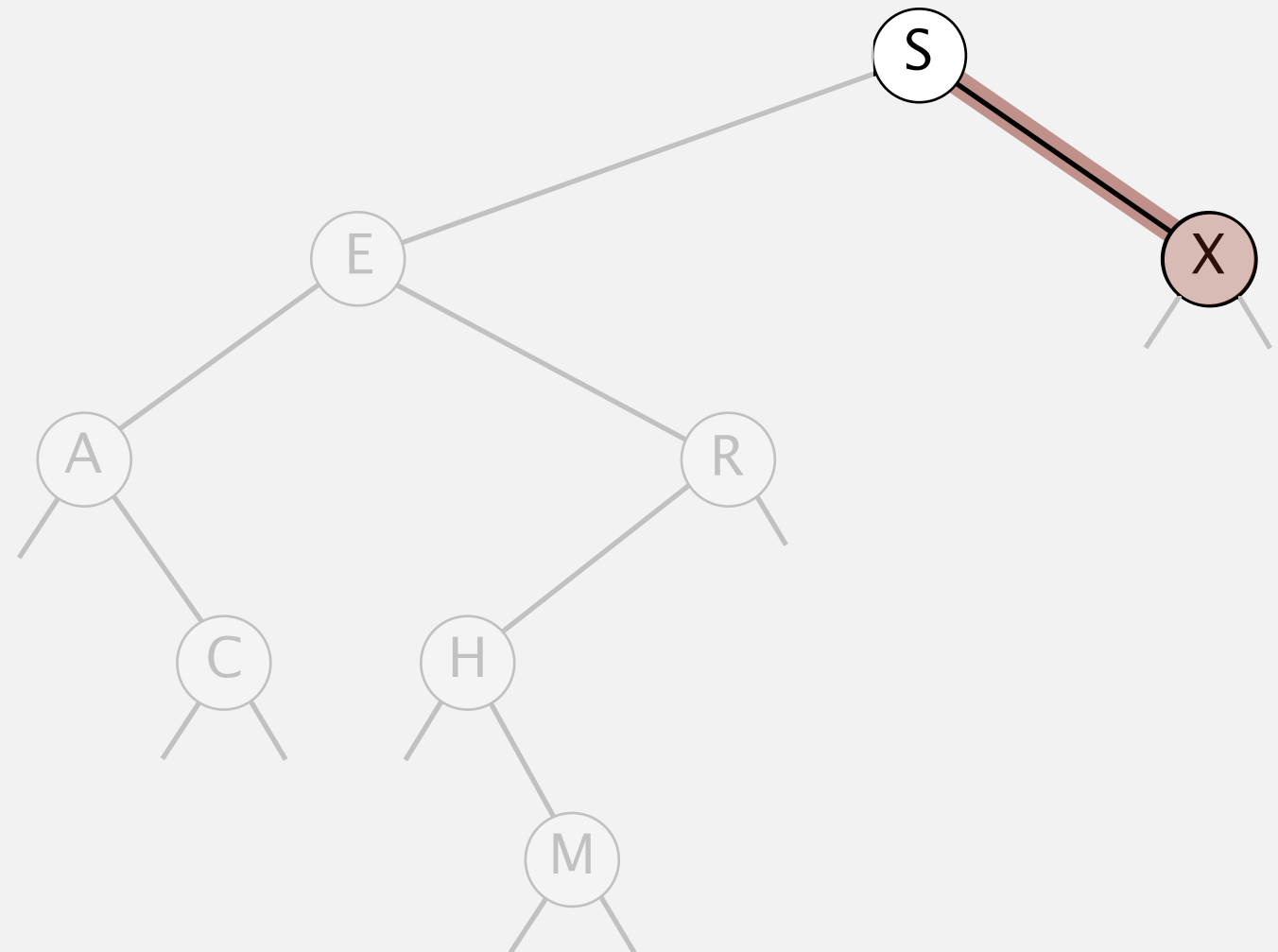


output: **A C E H M R S**

# Inorder traversal demo

---

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

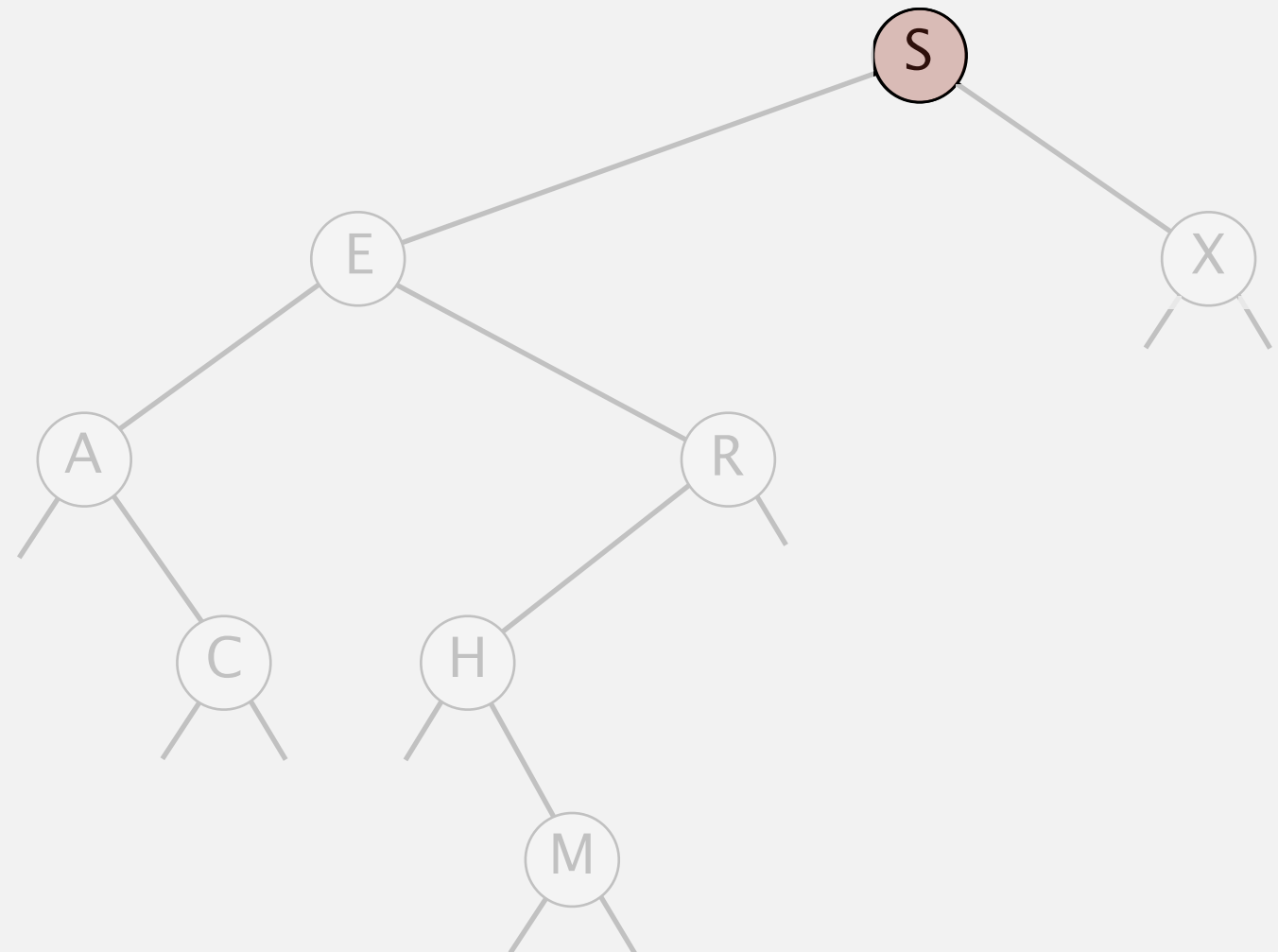


**output: A C E H M R S X**

# Inorder traversal demo

---

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

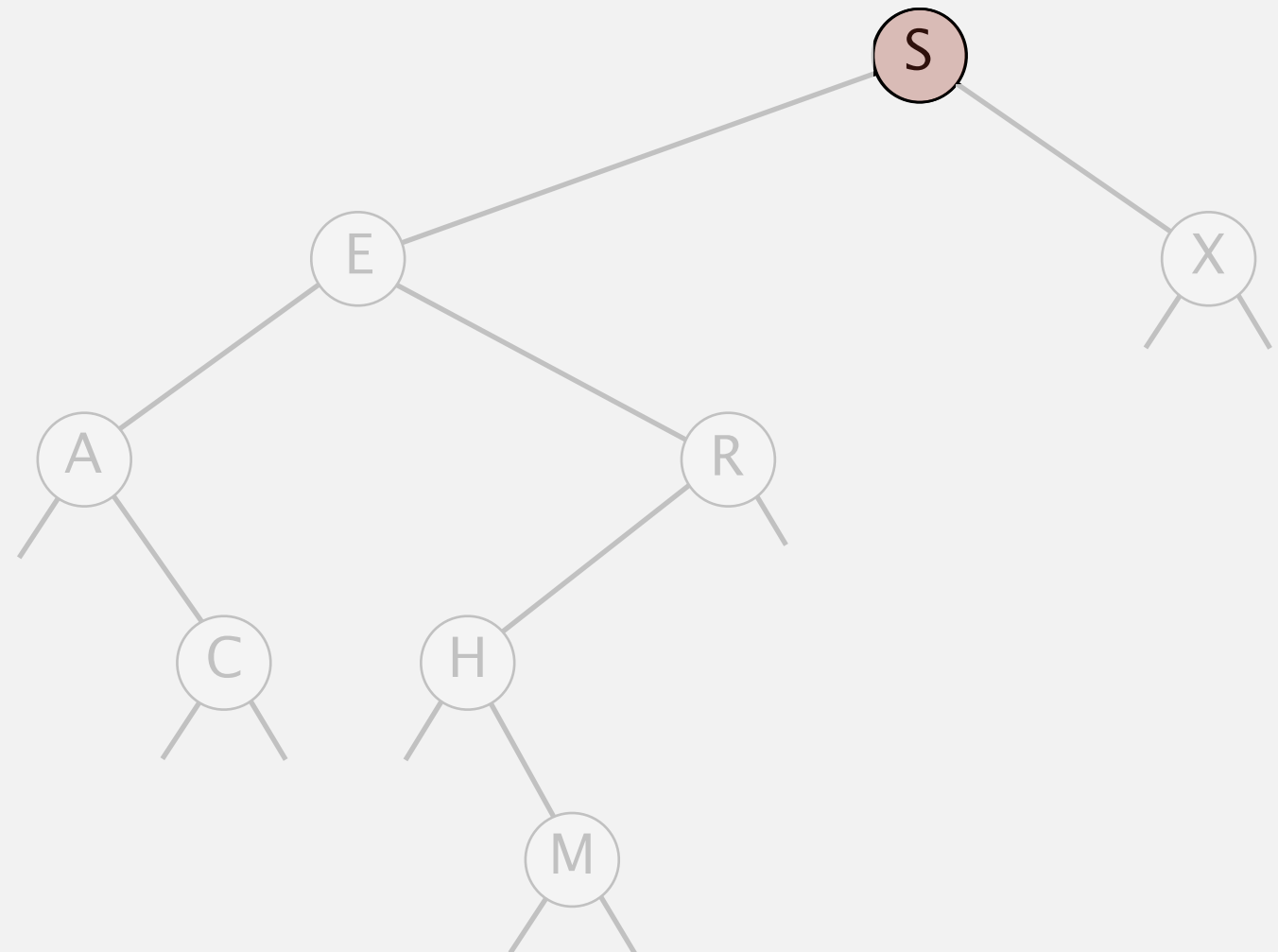


**output: A C E H M R S X**

# Inorder traversal demo

---

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



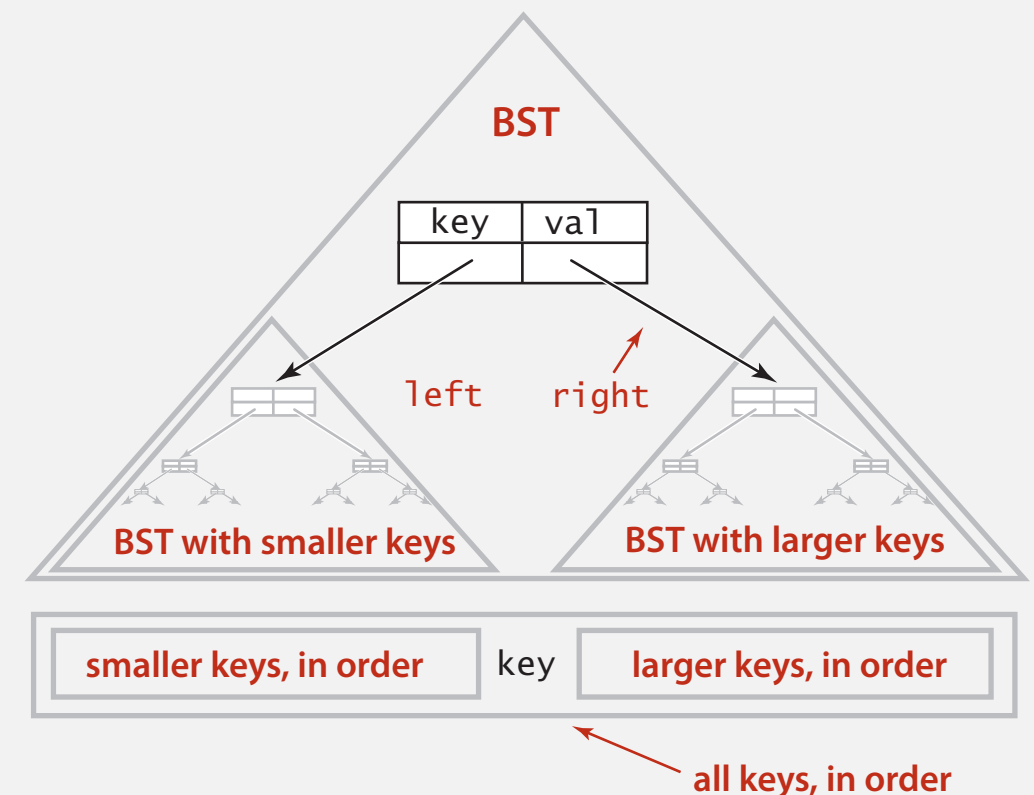
**output: A C E H M R S X**

# 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.



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

## 3.2 BINARY SEARCH TREES

---

- ▶ *BSTs*
- ▶ *iteration*
- ▶ *ordered operations*
- ▶ *deletion*

# BST: ordered symbol table operations summary

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

# ST implementations: summary

---

implementation	guarantee		average case		ordered ops?	key interface
	search	insert	search hit	insert		
sequential search (unordered list)	$N$	$N$	$N$	$N$		<code>equals()</code>
binary search (ordered array)	$\log N$	$N$	$\log N$	$N$	✓	<code>compareTo()</code>
BST	$N$	$N$	$\log N$	$\log N$	✓	<code>compareTo()</code>
red-black BST	$\log N$	$\log N$	$\log N$	$\log N$	✓	<code>compareTo()</code>

Next lecture. **Guarantee** logarithmic performance for all operations.

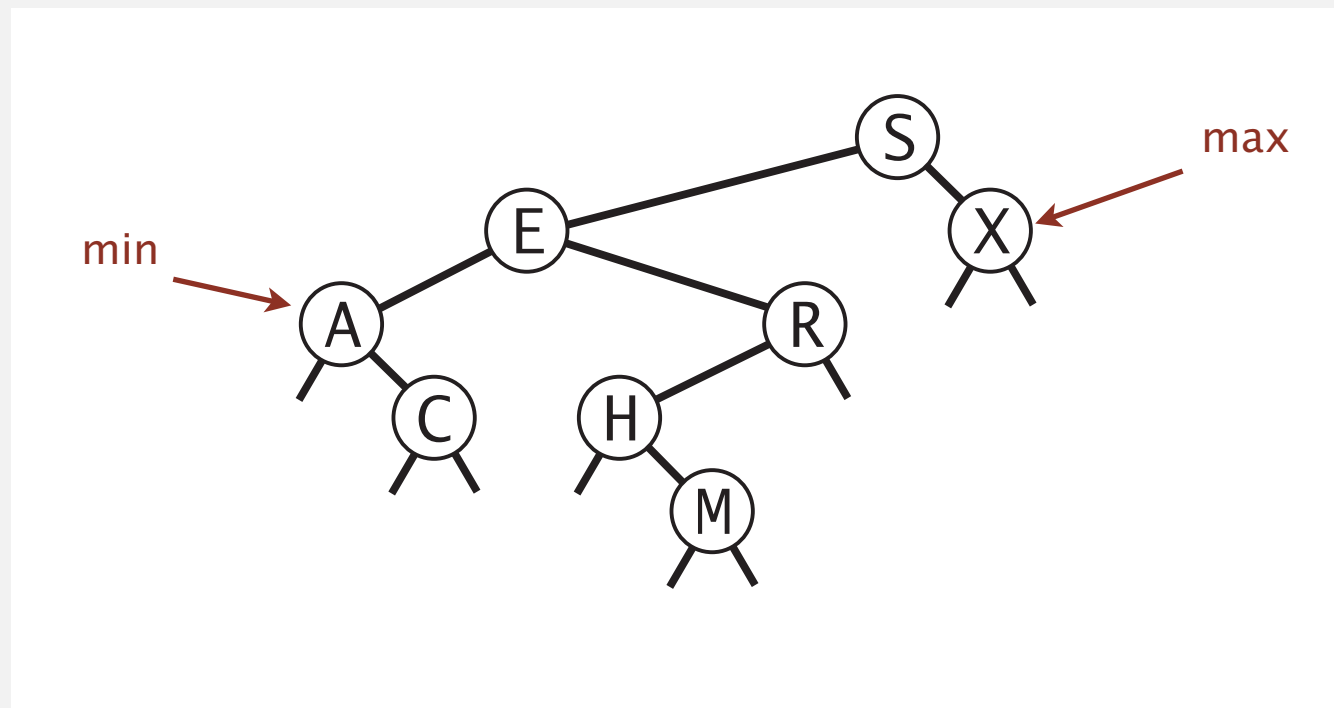


# Minimum and maximum

---

**Minimum.** Smallest key in BST.

**Maximum.** Largest key in BST.



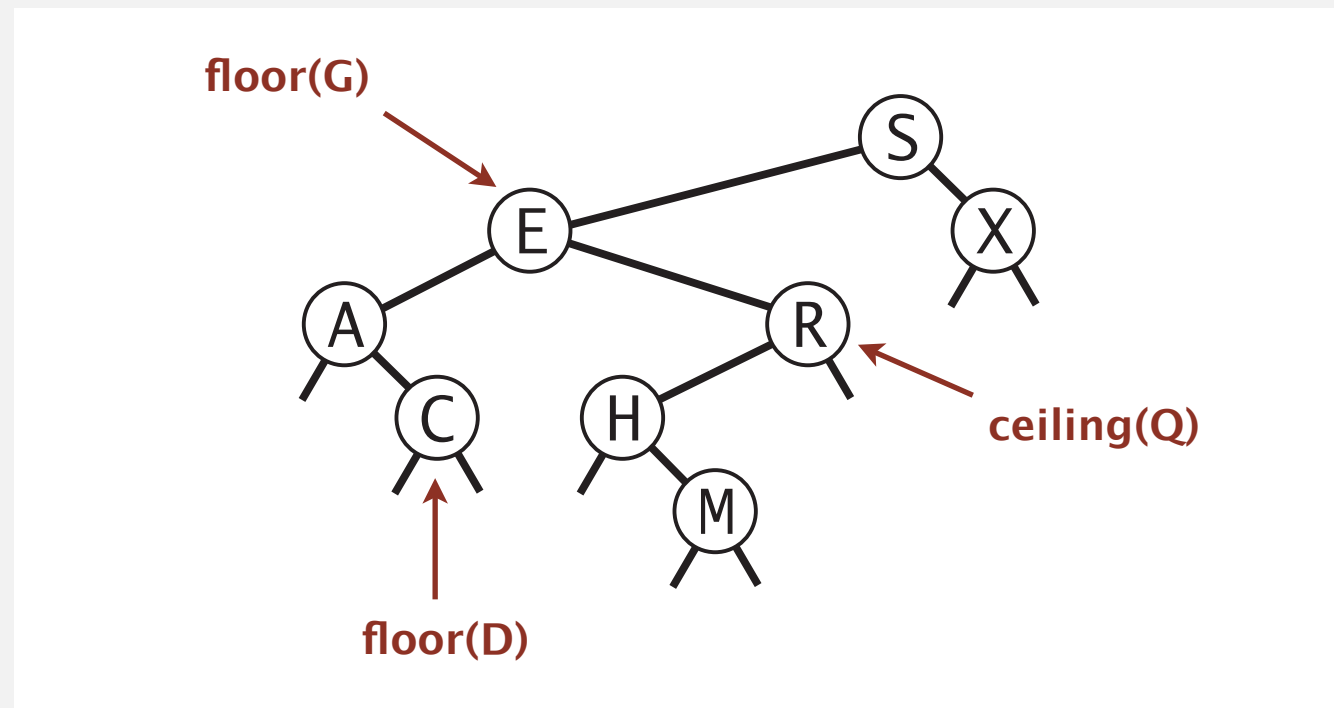
**Q.** How to find the min / max?

# Floor and ceiling

---

**Floor.** Largest key in BST  $\leq$  query key.

**Ceiling.** Smallest key in BST  $\geq$  query key.



Q. How to find the floor / ceiling?

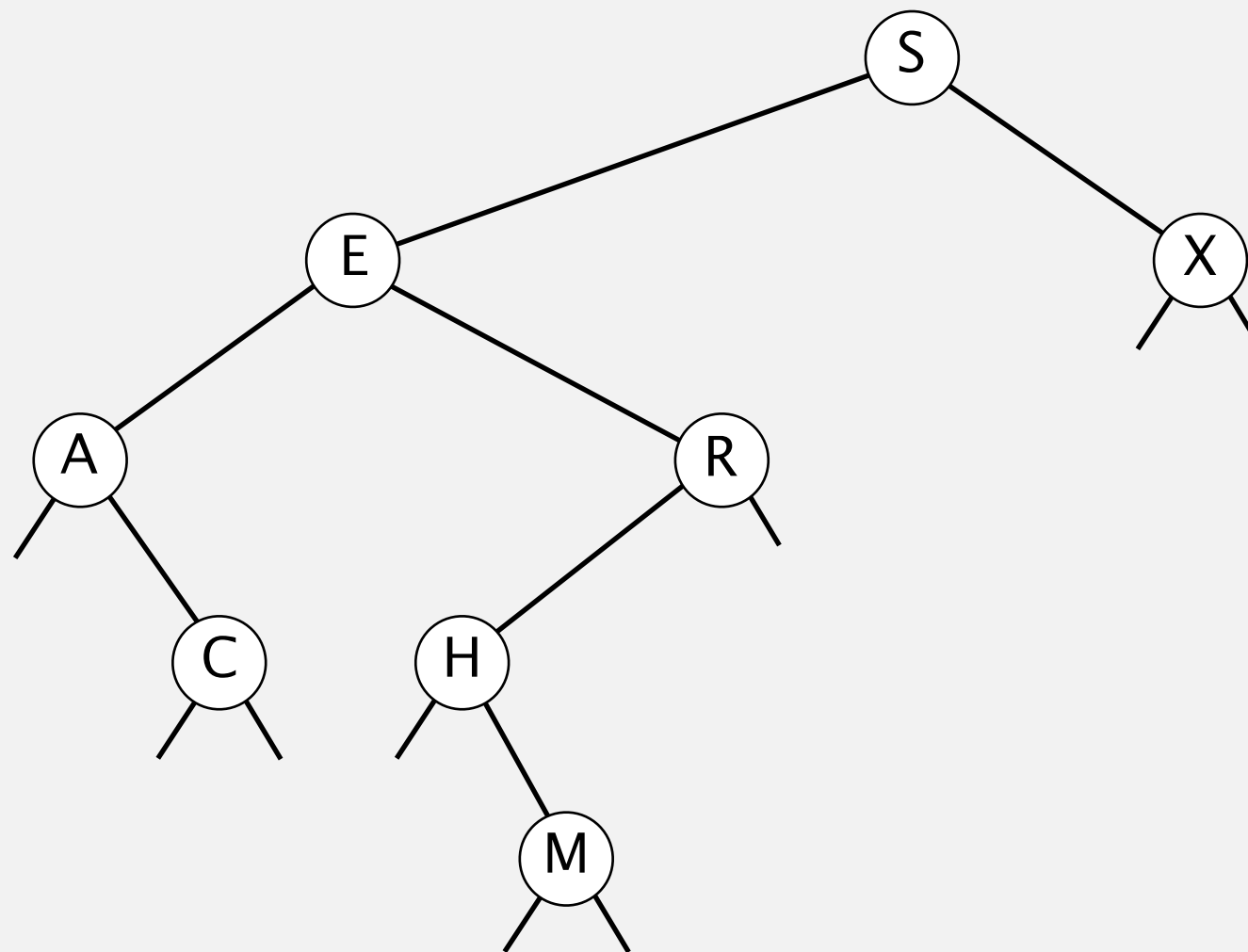


# Floor in a BST demo

---

**Floor.** Find the largest key in a BST that is  $\leq k$ ?

**floor of G**

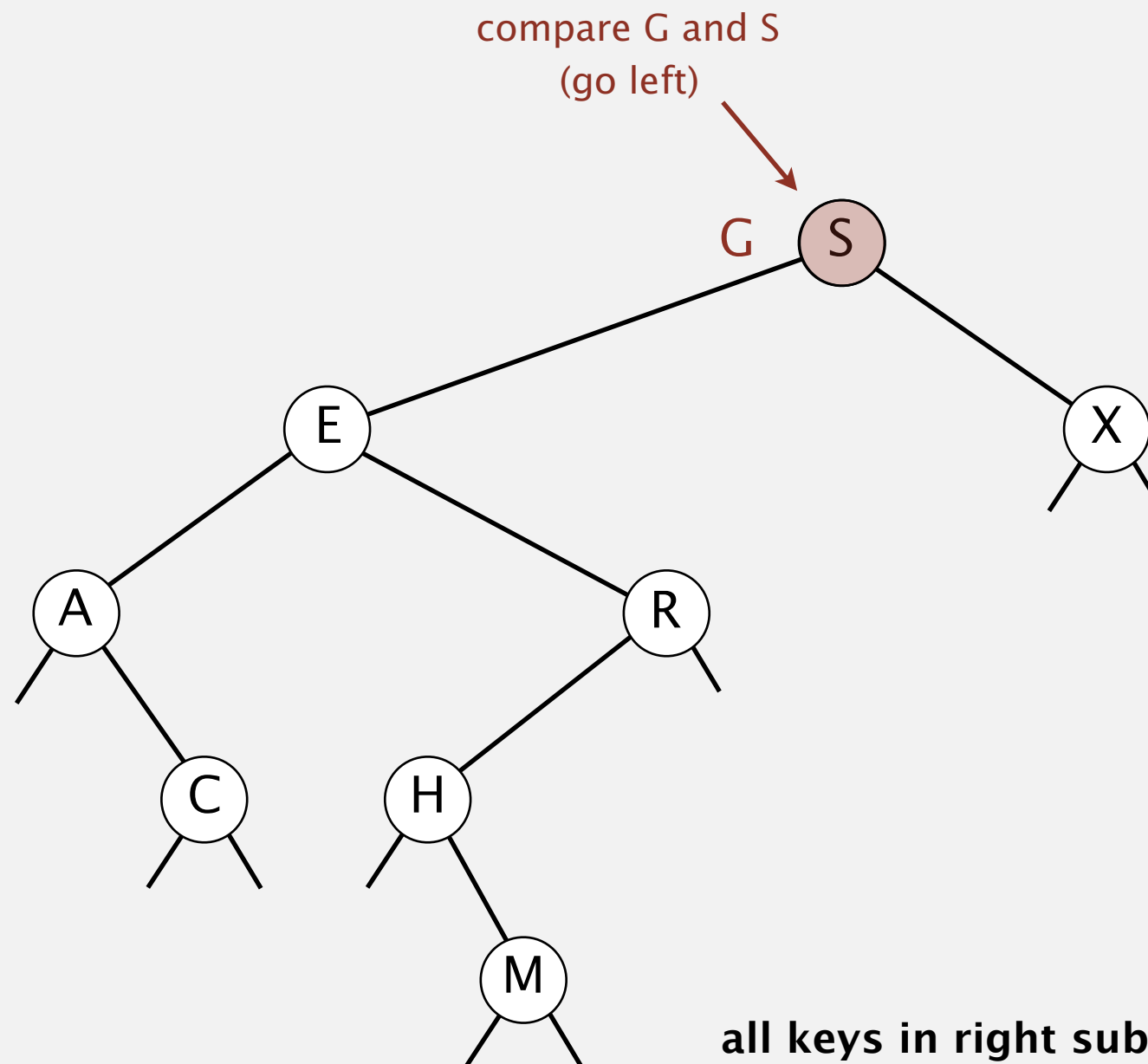


# Floor in a BST demo

---

**Floor.** Find the largest key in a BST that is  $\leq k$ ?

**floor of G**



all keys in right subtree of S are greater than G  
 $\Rightarrow$  compute floor of G in left subtree

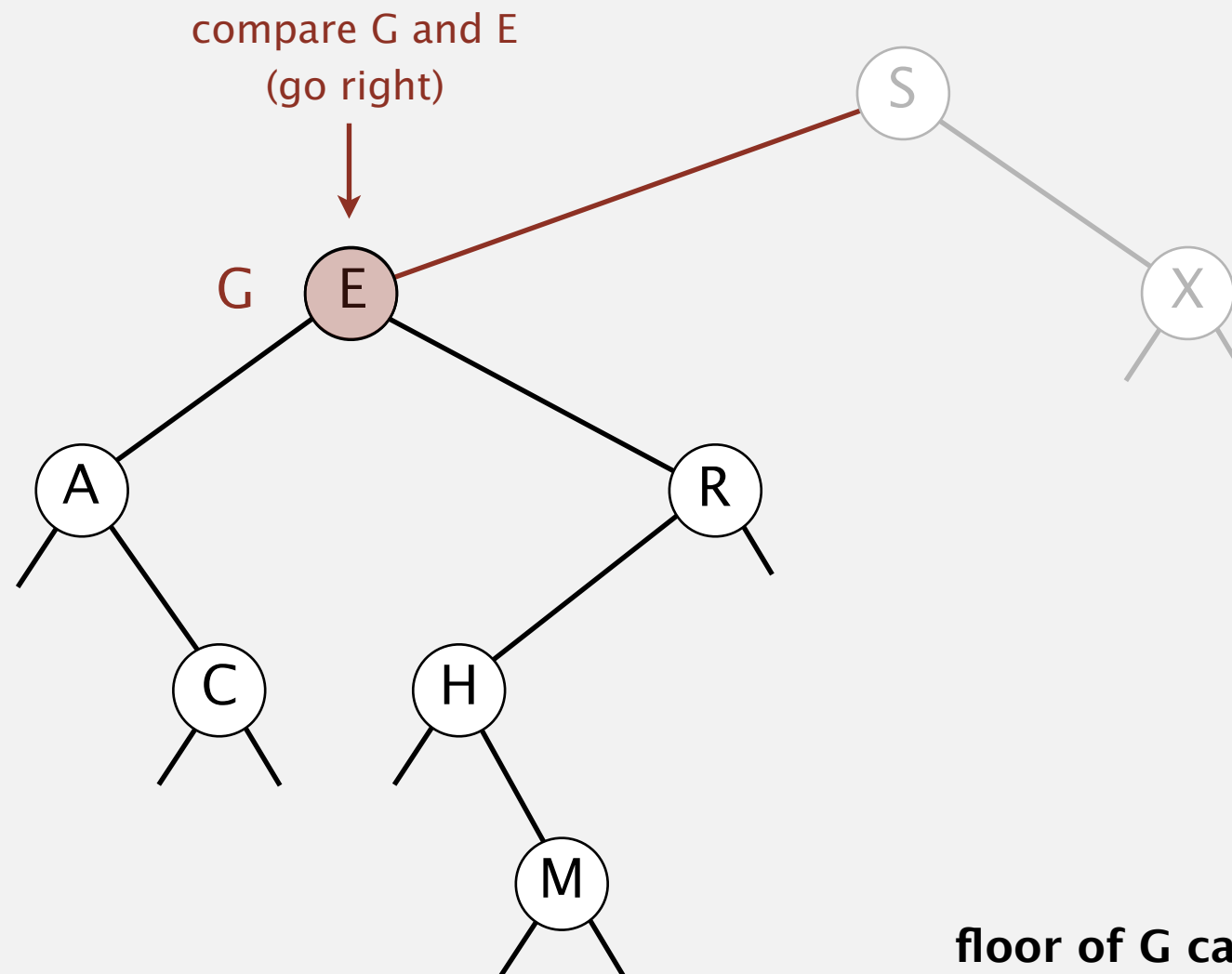
# Floor in a BST demo

---

**Floor.** Find the largest key in a BST that is  $\leq k$ ?

**floor of G**

E



**floor of G can't be in left subtree;  
floor is either E or floor of G in right subtree**

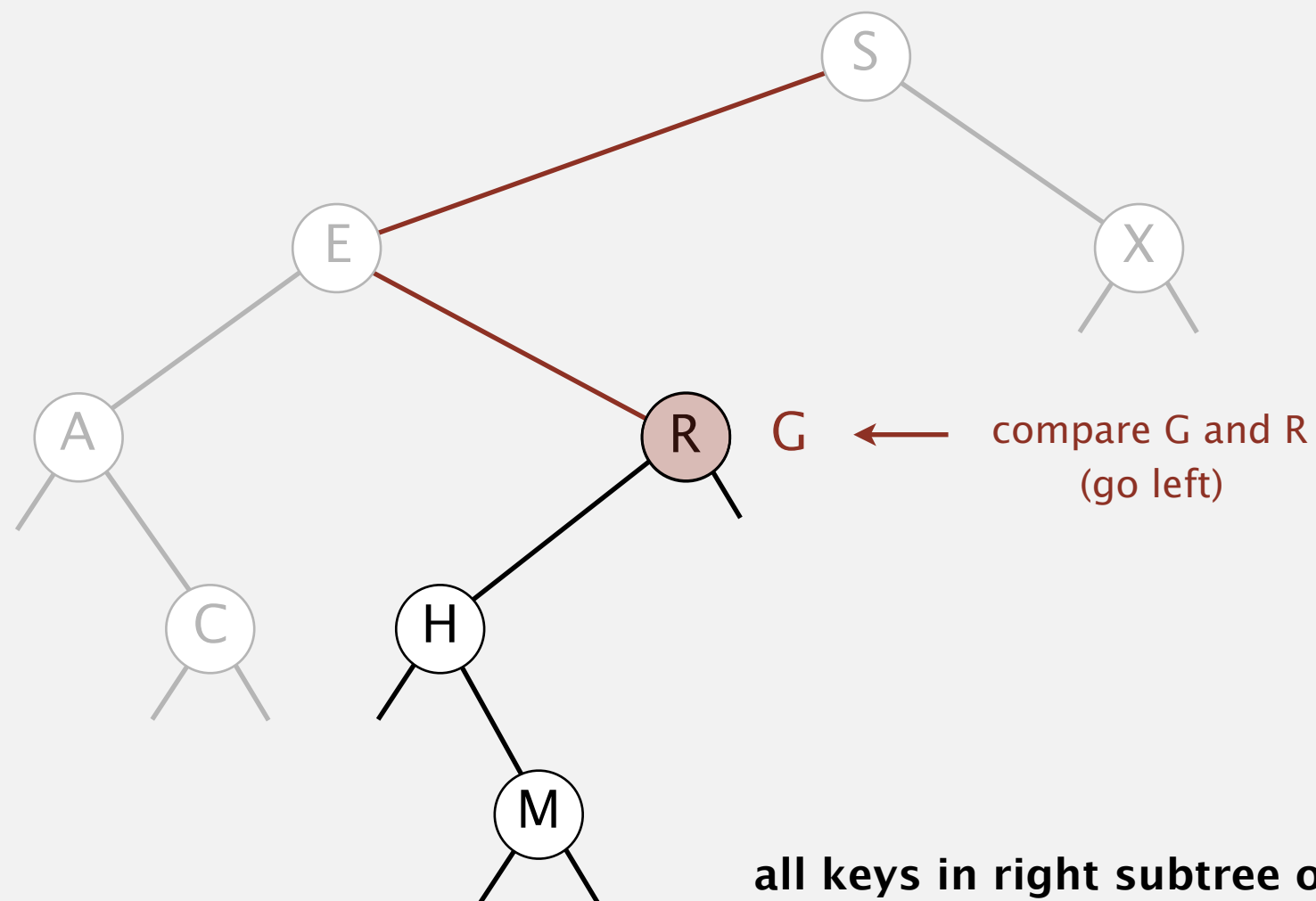
# Floor in a BST demo

---

**Floor.** Find the largest key in a BST that is  $\leq k$ ?

**floor of G**

E



all keys in right subtree of R are greater than G  
 $\Rightarrow$  compute floor of G in left subtree

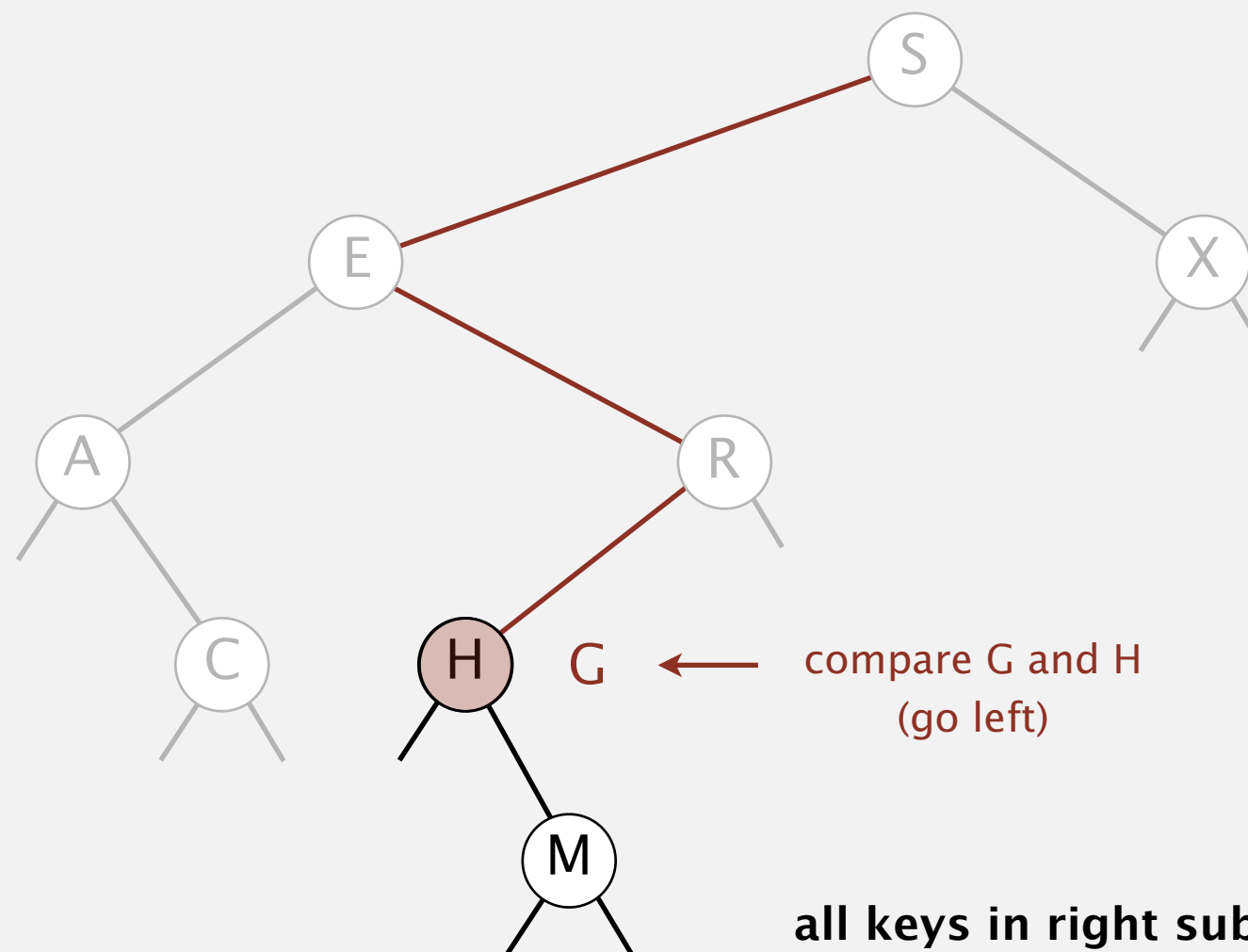
# Floor in a BST demo

---

**Floor.** Find the largest key in a BST that is  $\leq k$ ?

**floor of G**

E



all keys in right subtree of H are greater than G  
 $\Rightarrow$  compute floor of G in left subtree

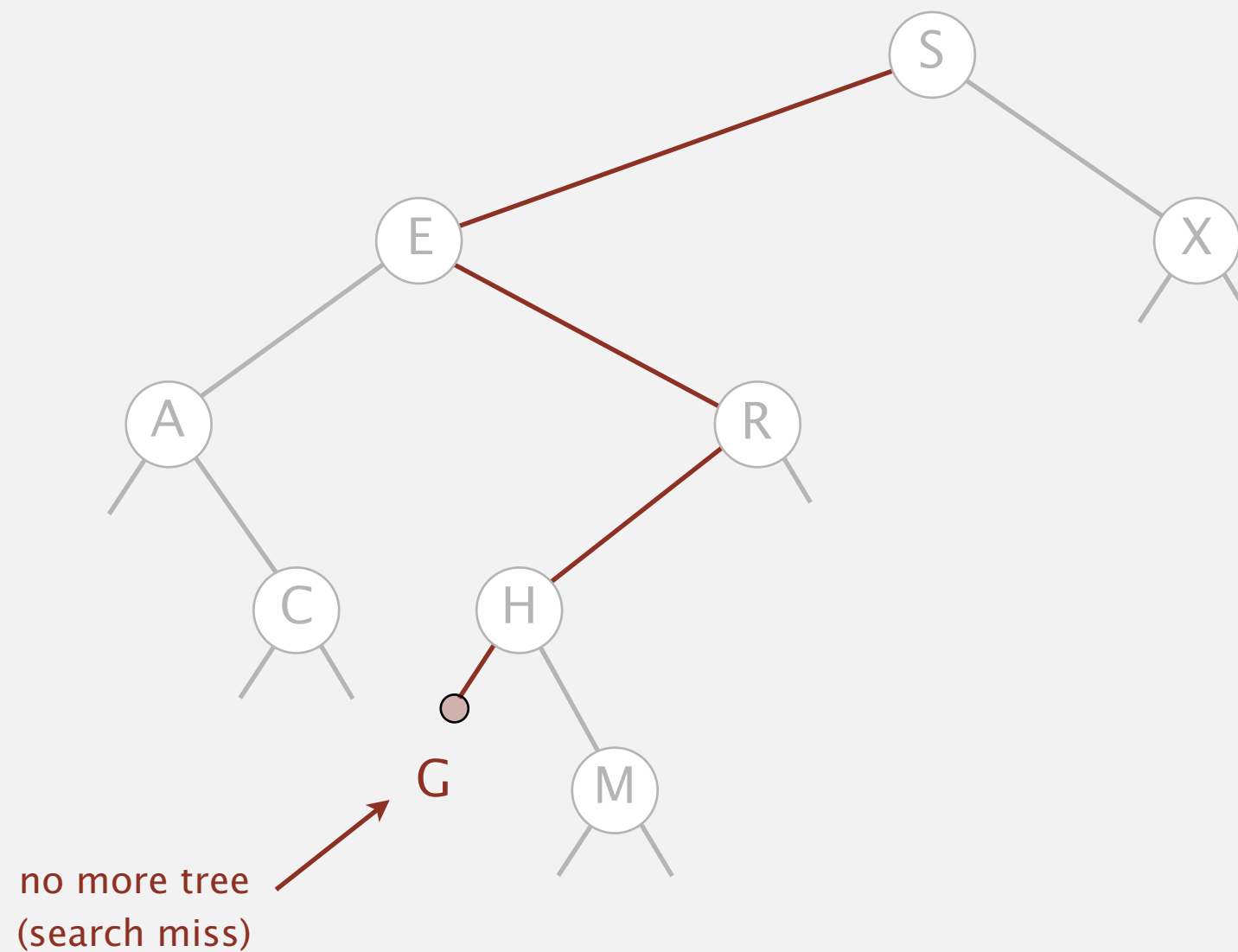
# Floor in a BST demo

---

**Floor.** Find the largest key in a BST that is  $\leq k$ ?

**floor of G**

**E**





# Computing the floor

**Floor.** Largest key in  $\text{BST} \leq k$ ?

**Case 1.** [ key in node  $x = k$  ]

The floor of  $k$  is  $k$ .

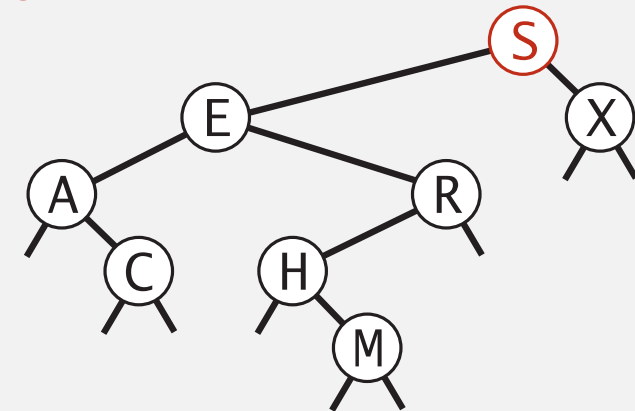
**Case 2.** [ key in node  $x > k$  ]

The floor of  $k$  is in the left subtree of  $x$ .

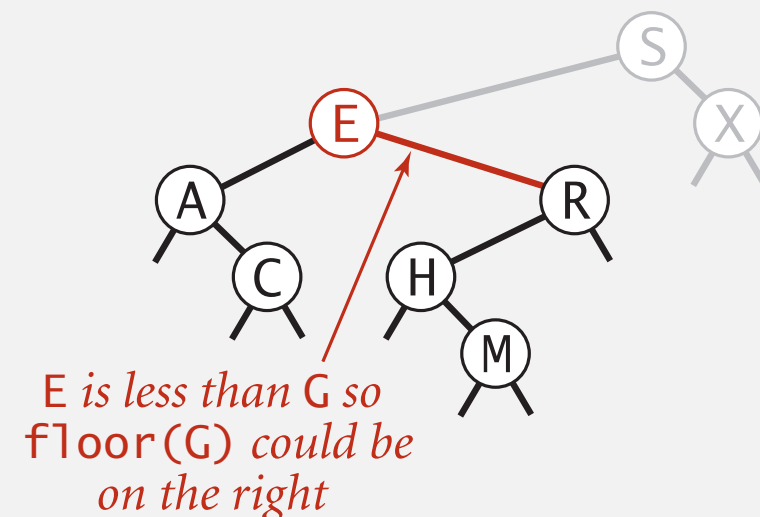
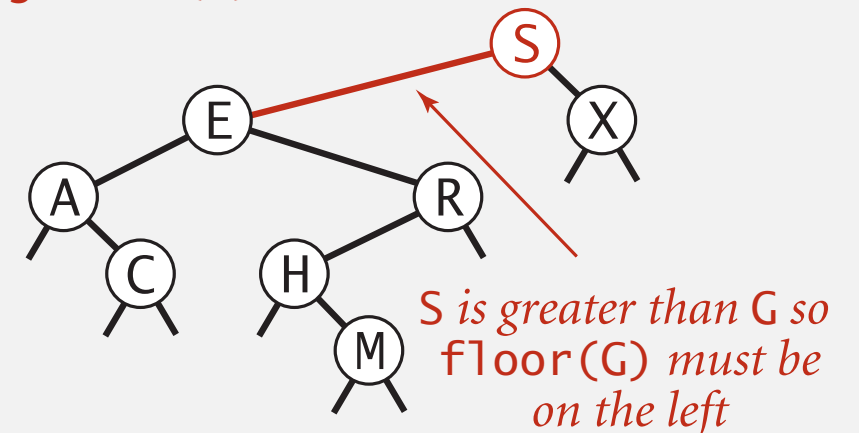
**Case 3.** [ key in node  $x < k$  ]

The floor of  $k$  can't be in left subtree of  $x$ :  
it is either in the right subtree of  $x$  or  
it is the key in node  $x$ .

finding floor(S)



finding floor(G)



# Computing the floor

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

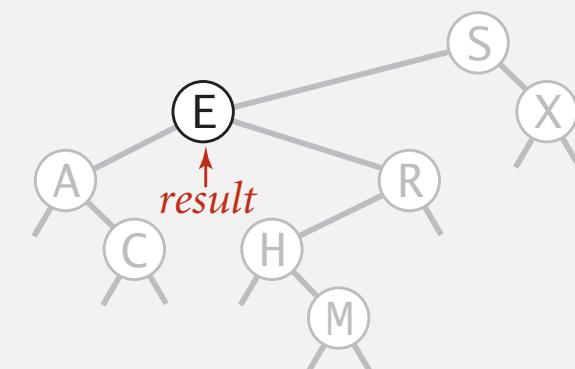
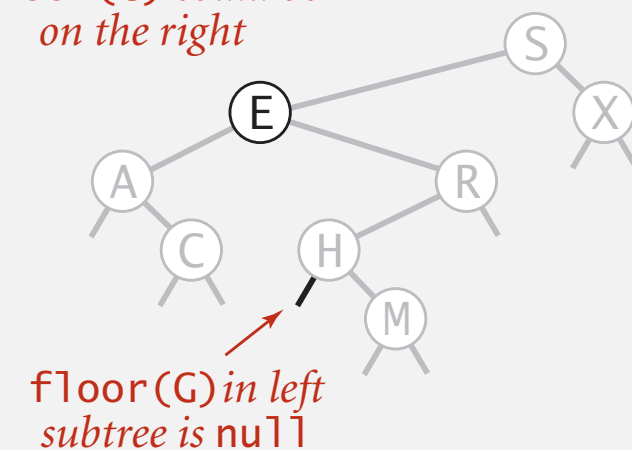
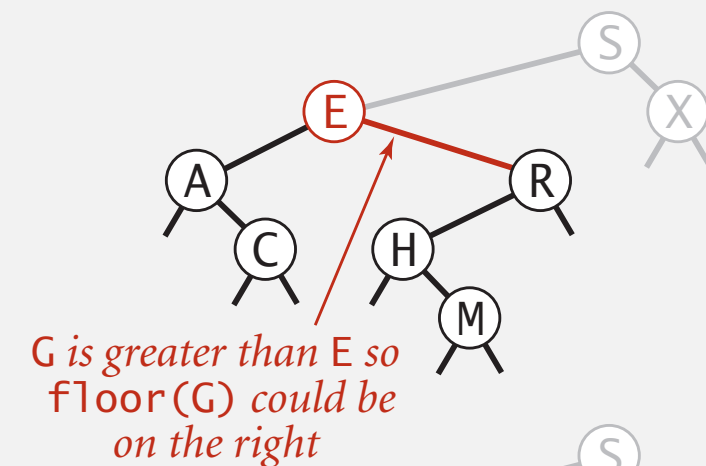
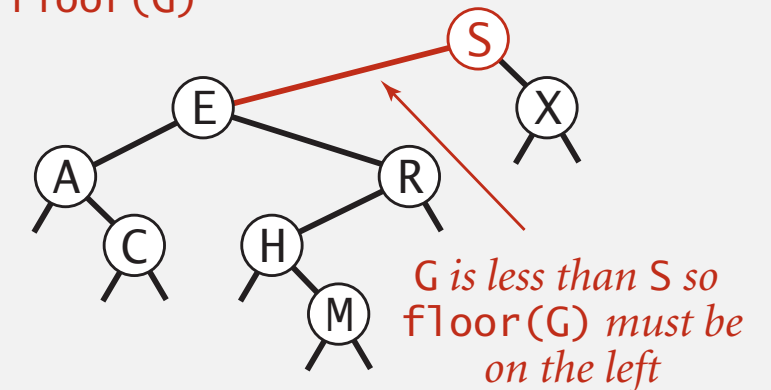
private Key floor(Node x, Key key)
{
    if (x == null) return null;
    int cmp = key.compareTo(x.key);

    if (cmp == 0) return x;

    if (cmp < 0) return floor(x.left, key);

    Key t = floor(x.right, key);
    if (t != null) return t;
    else return x.key;
}
```

finding floor(G)

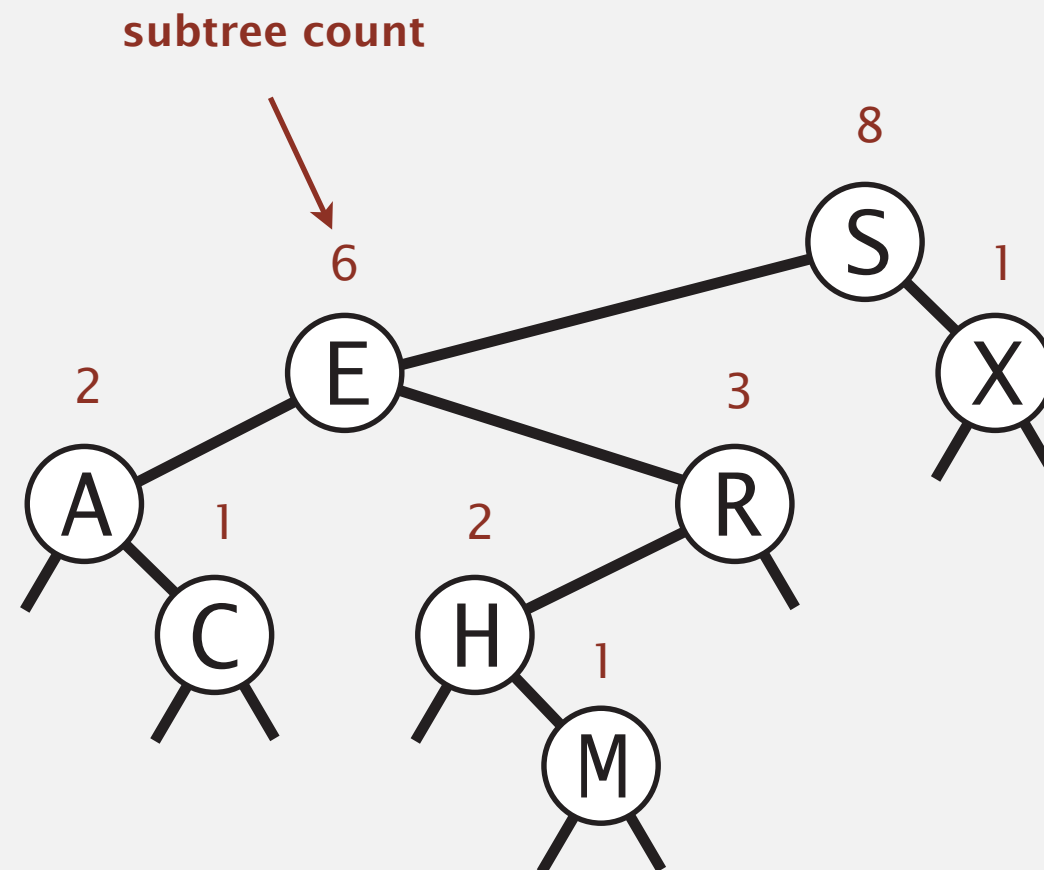


# Rank and select

---

Q. How to implement `rank()` and `select()` efficiently for BSTs?

A. In each node, store the number of nodes in its subtree.



# BST implementation: subtree counts

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

number of nodes in subtree

```
public int size()
{ return size(root); }
```

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

ok to call  
when x is null

```
private Node put(Node x, Key key, Value val)
{
    if (x == null) return new Node(key, val, 1);
    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.count = 1 + size(x.left) + size(x.right);
    return x;
}
```

initialize subtree  
count to 1

# Computing the rank

**Rank.** How many keys in  $\text{BST} < k$ ?

**Case 1.** [  $k < \text{key in node}$  ]

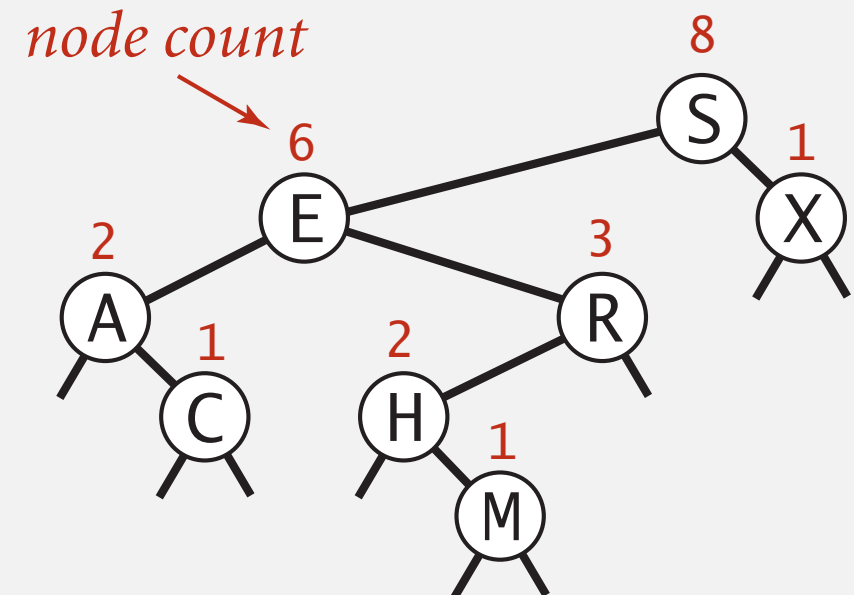
- Keys in left subtree? *count*
- Key in node? 0
- Keys in right subtree? 0

**Case 2.** [  $k > \text{key in node}$  ]

- Keys in left subtree? *all*
- Key in node. 1
- Keys in right subtree? *count*

**Case 3.** [  $k = \text{key in node}$  ]

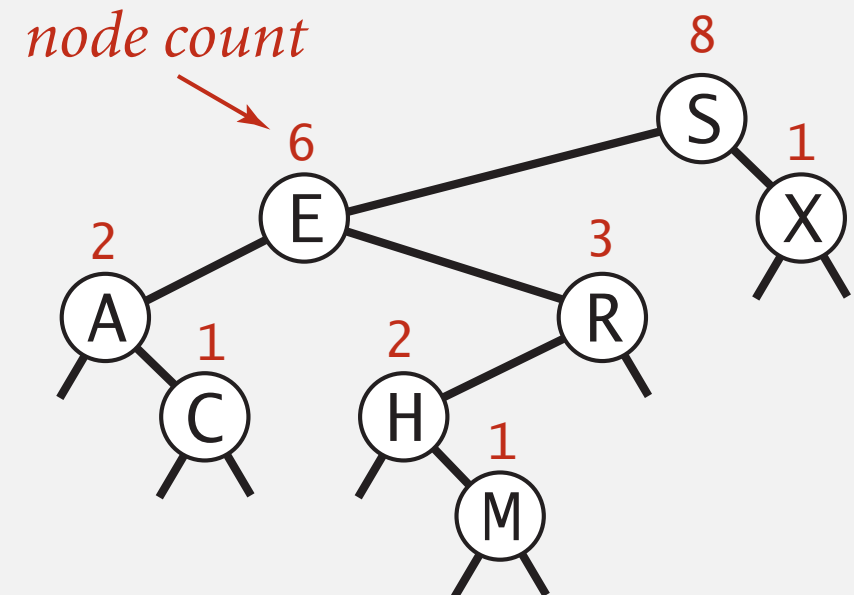
- Keys in left subtree? *count*
- Key in node. 0
- Keys in right subtree? 0



# Rank

**Rank.** How many keys in BST  $< k$ ?

Easy recursive algorithm (3 cases!)



```
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);
```

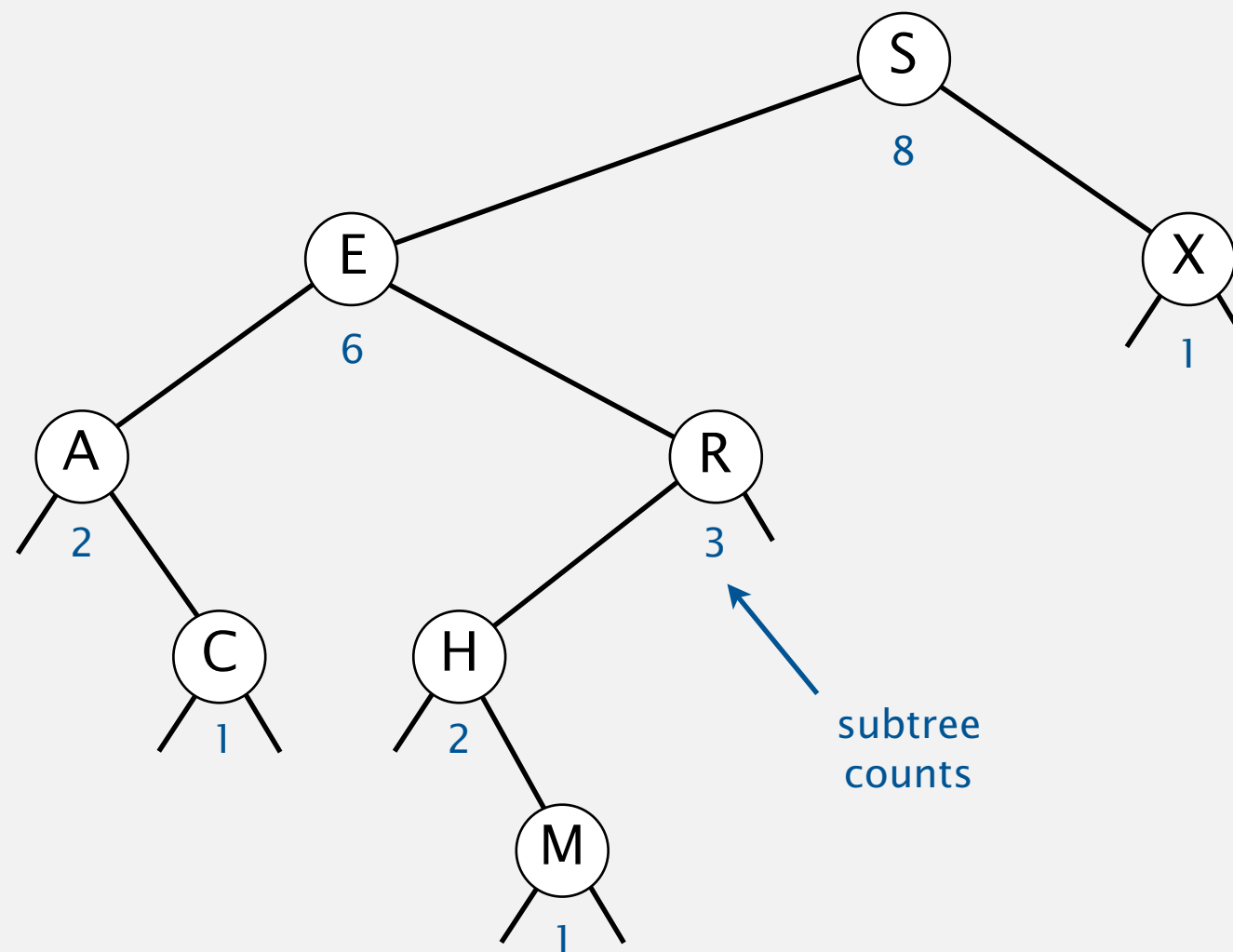
```
}
```

# Selection in a BST demo

---

**Select.** Find the key in a BST of rank  $k$ .

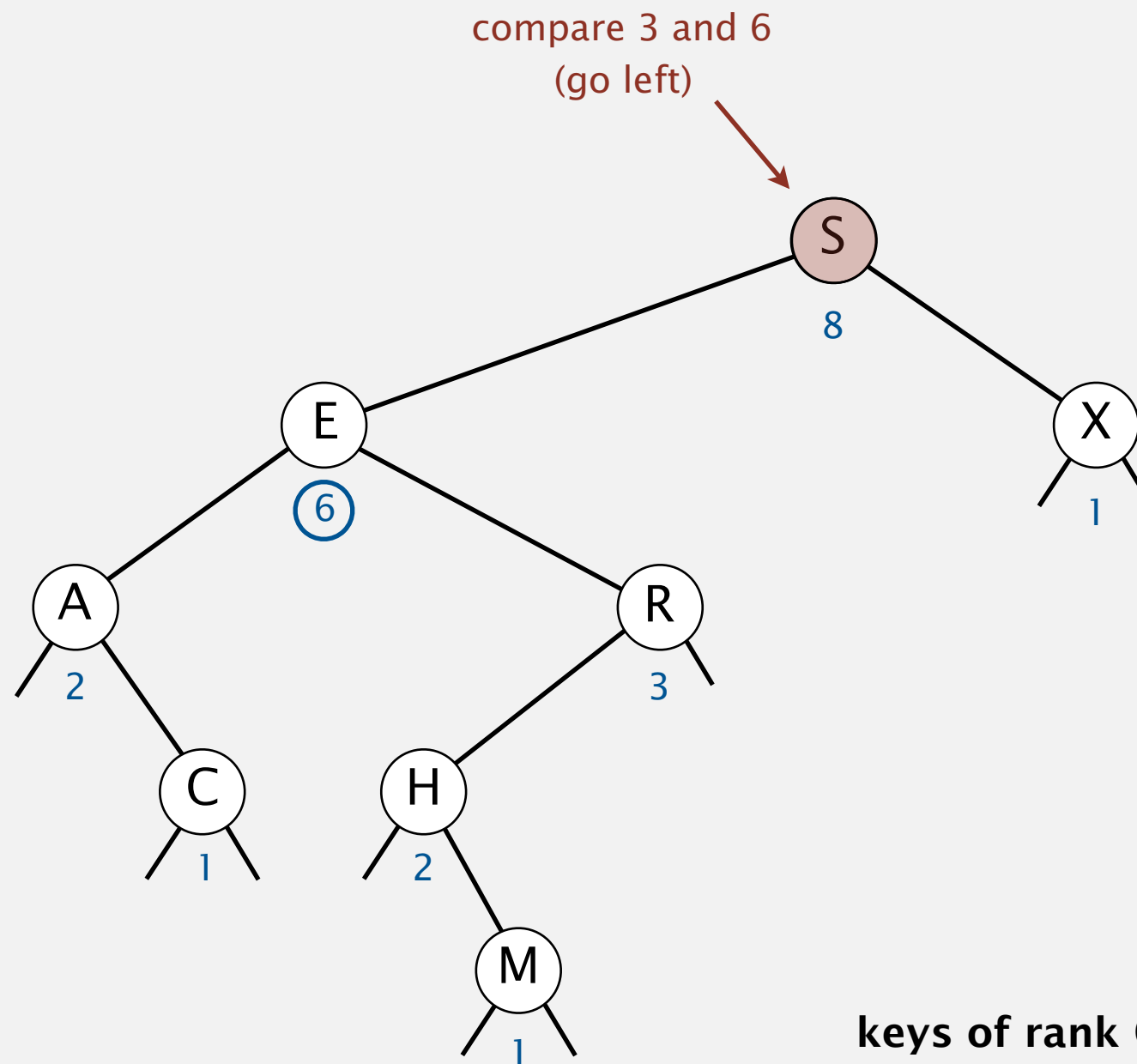
**rank(S, 3)**



# Selection in a BST demo

**Select.** Find the key in a BST of rank  $k$ .

**rank(S, 3)**



keys of rank 0–5 are in left subtree  $\Rightarrow$   
find key of rank 3 in subtree rooted at E



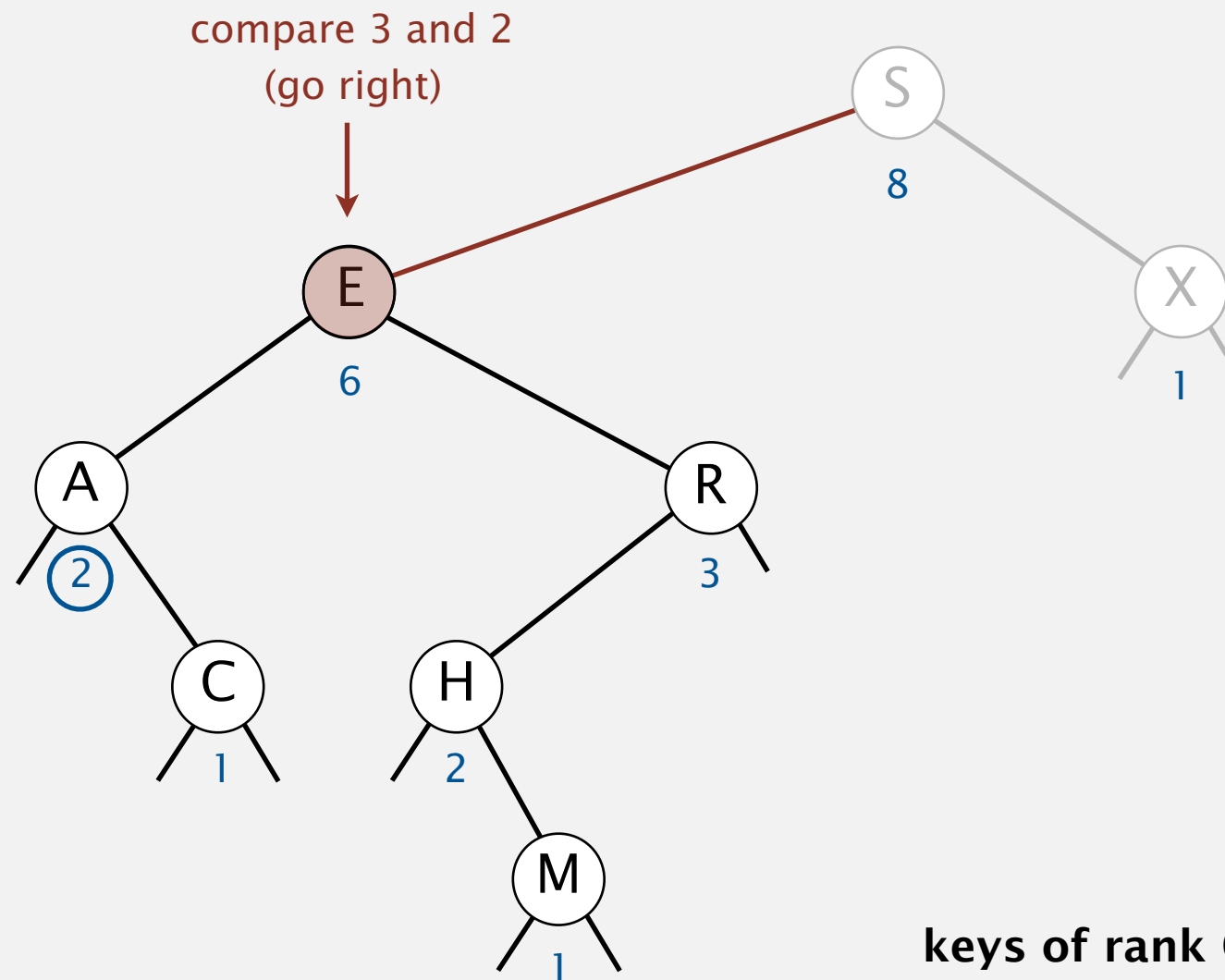
# Selection in a BST demo

---

**Select.** Find the key in a BST of rank  $k$ .

$\text{rank}(S, 3)$

$\text{rank}(E, 3)$



keys of rank 0–1 are in left subtree  $\Rightarrow$   
find key of rank 0 in subtree rooted at R

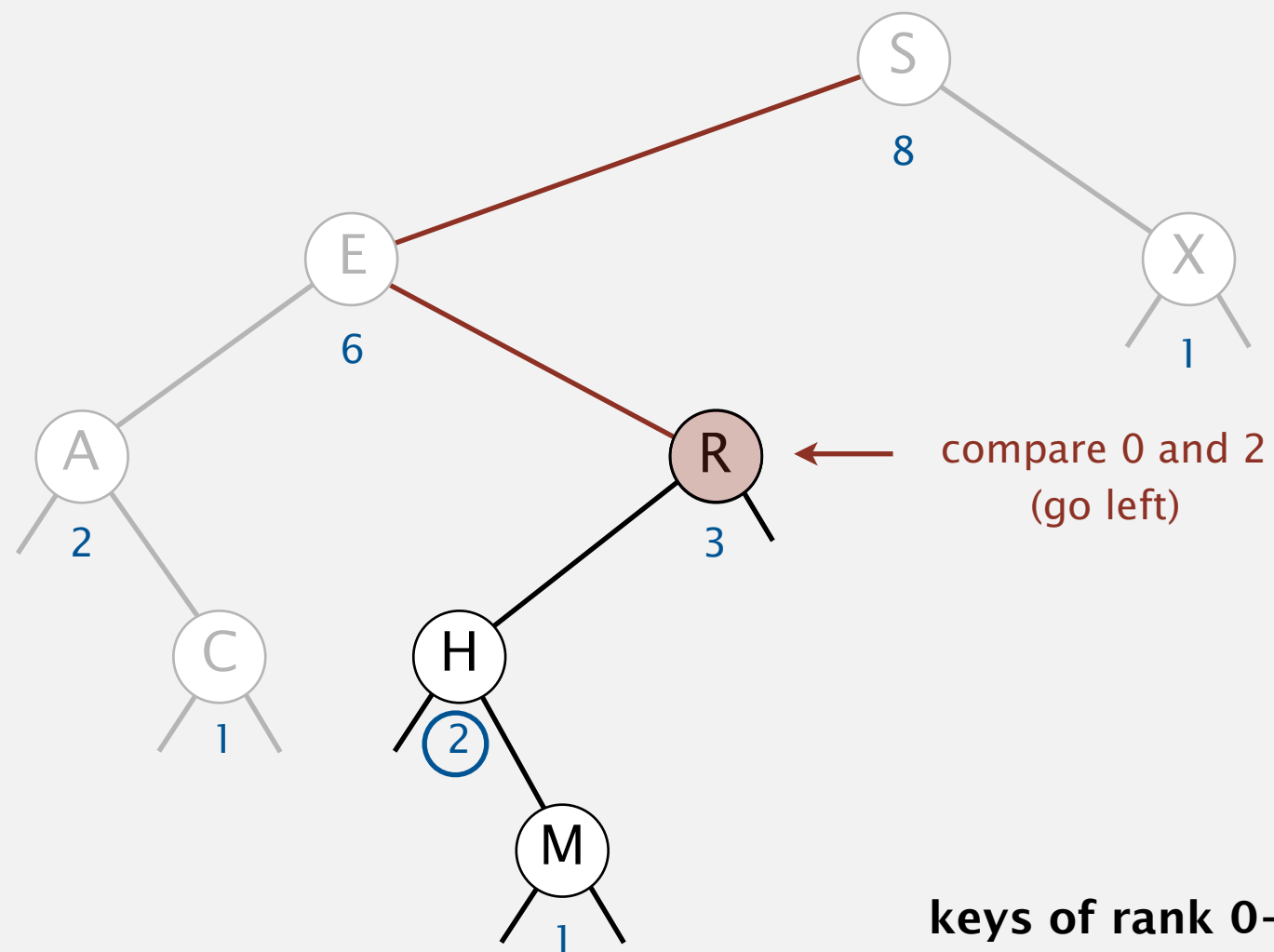
# Selection in a BST demo

**Select.** Find the key in a BST of rank  $k$ .

rank(S, 3)

rank(E, 3)

rank(R, 0)



keys of rank 0–1 are in left subtree  $\Rightarrow$   
find key of rank 0 in subtree rooted at H

# Selection in a BST demo

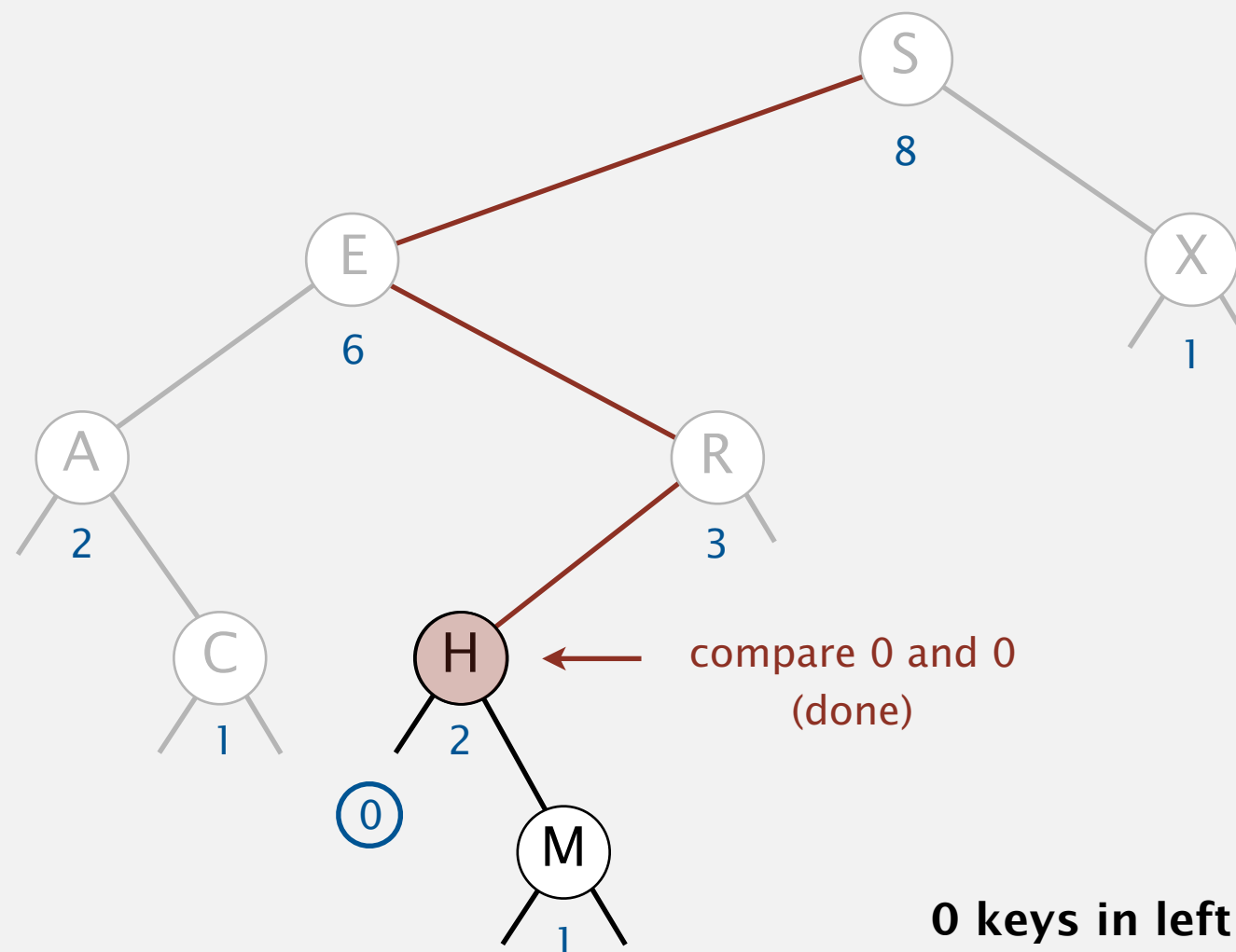
**Select.** Find the key in a BST of rank  $k$ .

rank(S, 3)

rank(E, 3)

rank(R, 0)

rank(H, 0)



0 keys in left subtree  $\Rightarrow$   
key of rank 0 in subtree rooted at H is H

# BST: ordered symbol table operations summary

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

# ST implementations: summary

---

implementation	guarantee		average case		ordered ops?	key interface
	search	insert	search hit	insert		
sequential search (unordered list)	$N$	$N$	$N$	$N$		equals()
binary search (ordered array)	$\log N$	$N$	$\log N$	$N$	✓	compareTo()
BST	$N$	$N$	$\log N$	$\log N$	✓	compareTo()
red-black BST	$\log N$	$\log N$	$\log N$	$\log N$	✓	compareTo()

Next lecture. **Guarantee** logarithmic performance for all operations.