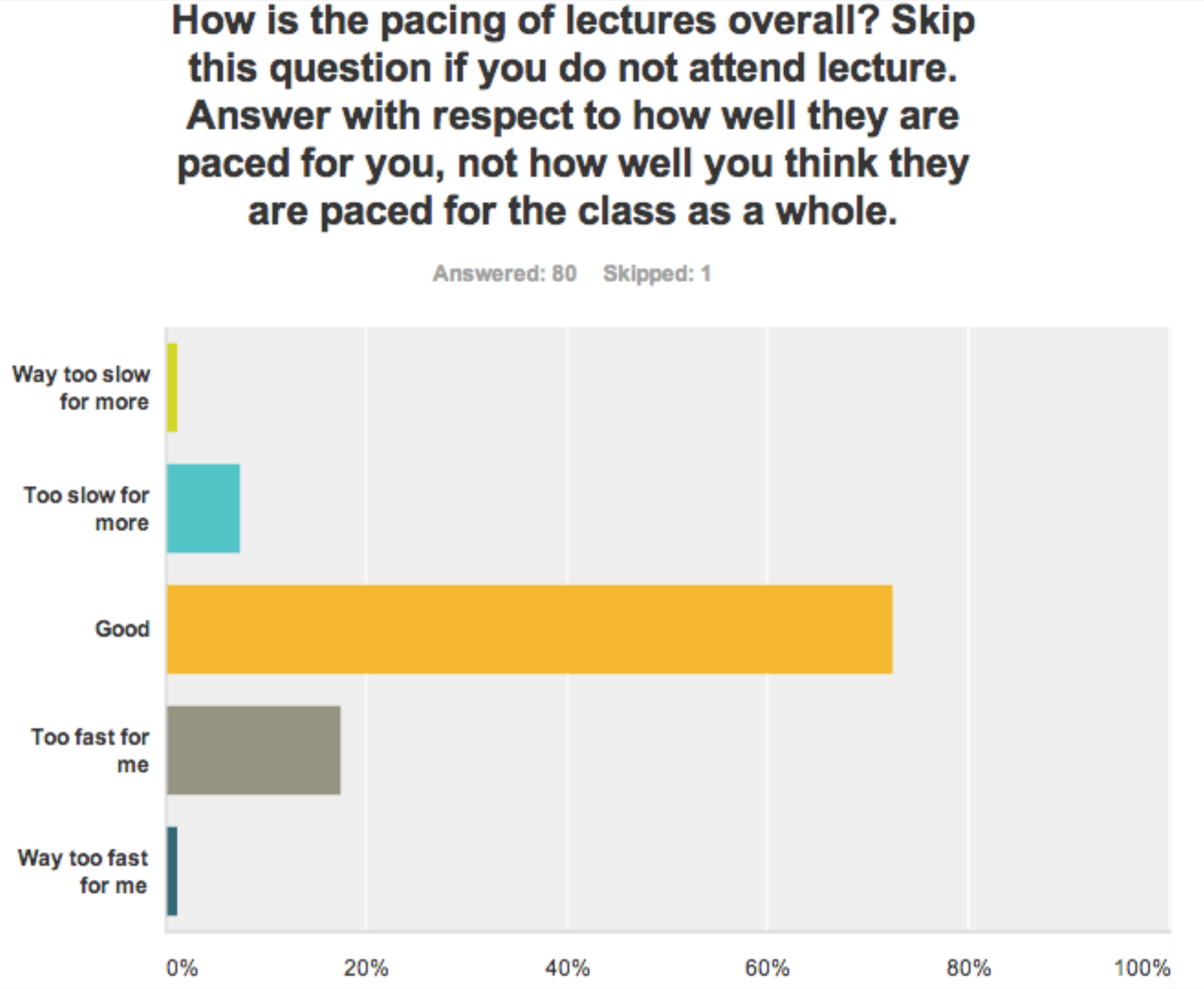


How is the pacing of lectures overall? Skip this question if you do not attend lecture. Answer with respect to how well they are paced for you, not how well you think they are paced for the class as a whole.

Answered: 80 Skipped: 1

http://algs4.cs.princeton.edu

3.1 SYMBOL TABLES

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

Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

3.1 SYMBOL TABLES

- *API*
- *elementary implementations*
- *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*	

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)

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.

specify `Comparable` in API.

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.

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()`.

- `x.equals(y)` works for any objects x and y, even if different class.

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

```
x = new Point(0, 0);
y = new Point(0, 0);
x.equals(y); //returns false
```

Default implementation. `(x == y)`

do x and y refer to the same object?

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 and cast.
- Compare each significant field:
  - if field is a primitive type, use ==
  - if field is an object, use `equals()` ← apply rule recursively
  - if field is an array, apply to each entry ← alternatively, use `Arrays.equals(a, b)` or `Arrays.deepEquals(a, b)`, but not `a.equals(b)`

Best practices.

e.g. cached `Manhattan() distance`

- 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)`

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

3.1 SYMBOL TABLES

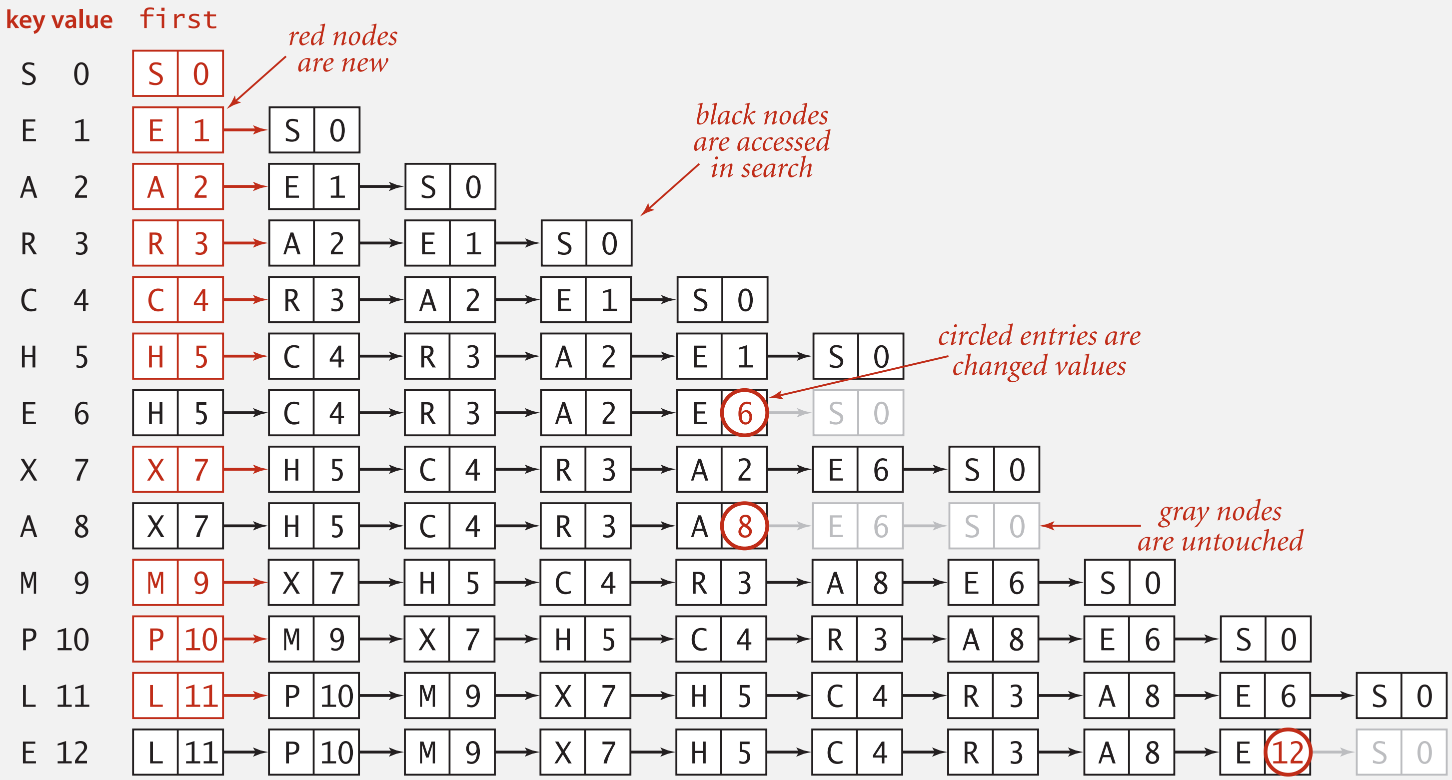
- *API*
- *elementary implementations*
- *ordered operations*

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.



Trace of linked-list ST implementation for standard indexing client

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	`equals()`

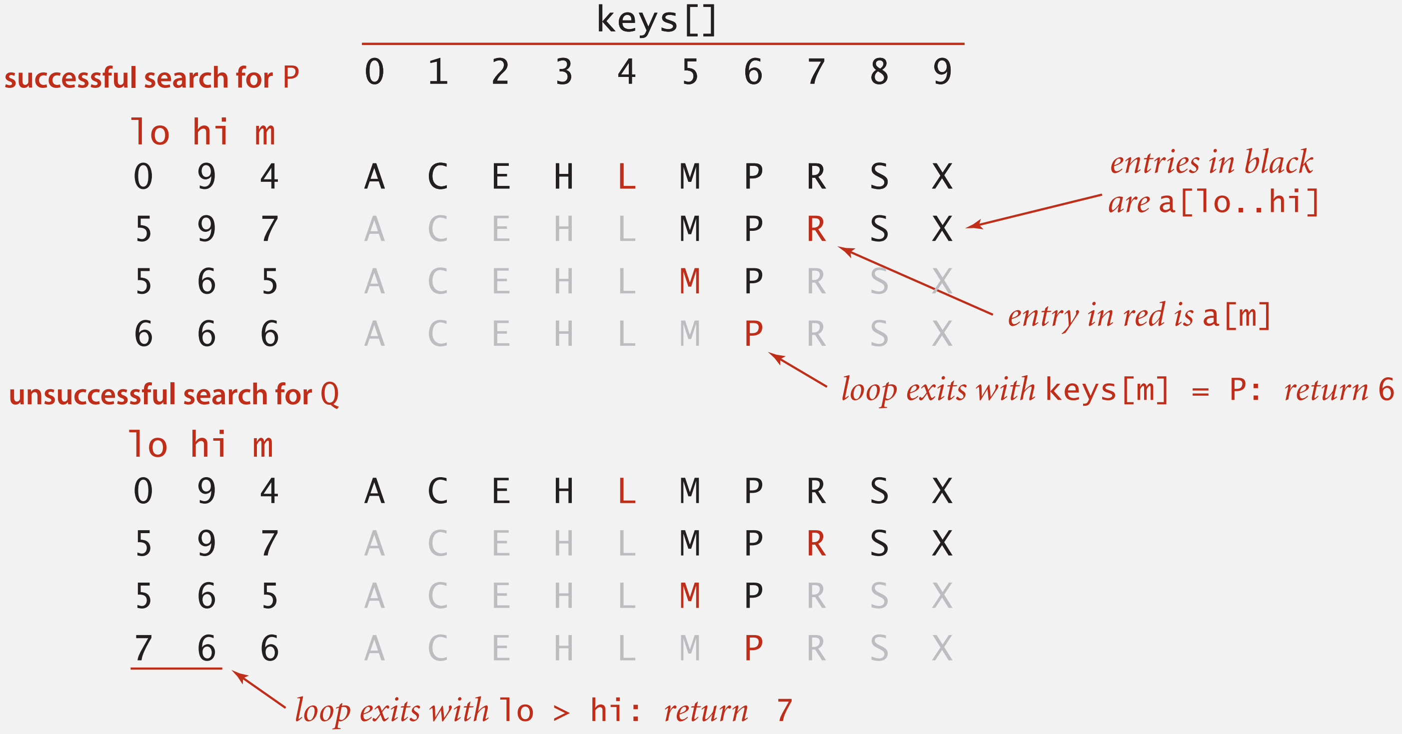
Challenge. Efficient implementations of both search and insert.

Binary search in an ordered array

- Data structure. Maintain two arrays: One for keys, one for values.
  - Keys are kept in order.
  - Values kept at same index as corresponding key.

equal to the index of k if it is in keys[]

Rank helper function. How many keys < k ?



keys[]

successful search for P

lo	hi	m	0	1	2	3	4	5	6	7	8	9
0	9	4	A	C	E	H	L	M	P	R	S	X
5	9	7	A	C	E	H	L	M	P	R	S	X
5	6	5	A	C	E	H	L	M	P	R	S	X
6	6	6	A	C	E	H	L	M	P	R	S	X

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

entry in red is `a[m]`

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

unsuccessful search for Q

lo	hi	m	0	1	2	3	4	5	6	7	8	9
0	9	4	A	C	E	H	L	M	P	R	S	X
5	9	7	A	C	E	H	L	M	P	R	S	X
5	6	5	A	C	E	H	L	M	P	R	S	X
7	6	6	A	C	E	H	L	M	P	R	S	X

loop exits with `lo > hi:` *return* 7

Trace of binary search for rank in an ordered array

Binary search: Java implementation

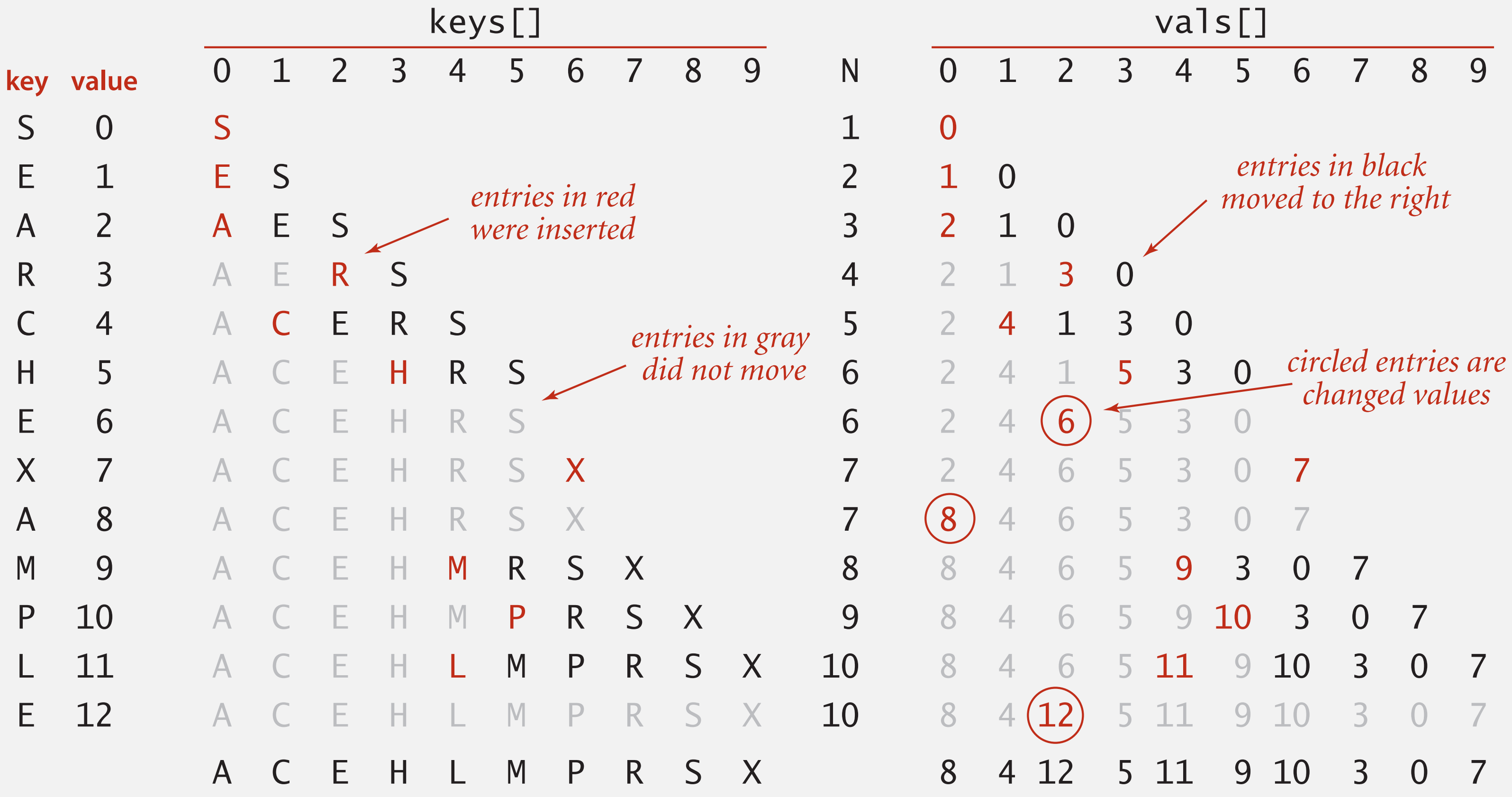
```
 public Value get(Key key)
 {
    if (isEmpty()) return null;
    int i = rank(key);
    if (i < N && keys[i].compareTo(key) == 0) return vals[i];
    else return null;
 }
```

number of keys < key

```
 private int rank(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 mid;
   }
   return lo;
}
```

Binary search: trace of standard indexing client

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



		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

entries in red were inserted

entries in gray did not move

entries in black moved to the right

circled entries are changed values

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	`equals()`
binary search (ordered array)	log N	N	log N	N / 2	yes	`compareTo()`

Challenge. Efficient implementations of both search and insert.

Algorithms

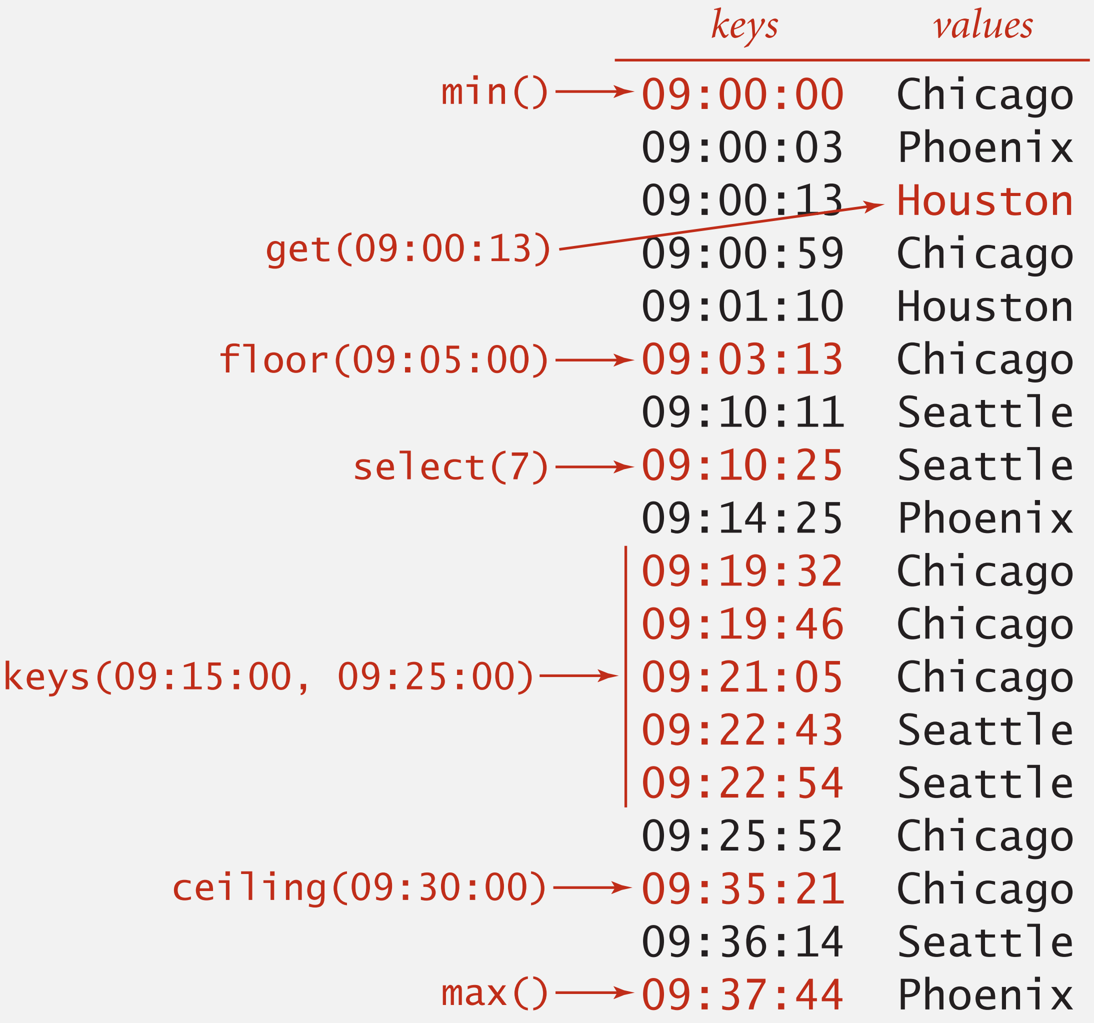
Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

3.1 SYMBOL TABLES

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

Examples of ordered symbol table API



	keys	*values*
min()	09:00:00	Chicago
	09:00:03	Phoenix
get(09:00:13)	09:00:13	Houston
	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
keys(09:15:00, 09:25:00)	09:19:32	Chicago
	09:19:46	Chicago
	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

Ordered symbol table API

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

	`ST()`	*create an ordered symbol table*
`void`	`put(Key key, Value val)`	*put key-value pair into the table* *(remove* `key` *from table if value is* `null`*)*
`Value`	`get(Key key)`	*value paired with* `key` *(*`null` *if* `key` *is absent)*
`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*
`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`
`void`	`deleteMin()`	*delete smallest key*
`void`	`deleteMax()`	*delete largest key*
`int`	`size(Key lo, Key hi)`	*number of keys in* `[lo..hi]`
`Iterable<Key>`	`keys(Key lo, Key hi)`	*keys in* `[lo..hi]`*, in sorted order*
`Iterable<Key>`	`keys()`	*all keys in the table, in sorted order*

Binary search: ordered symbol table operations summary

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

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

http://algs4.cs.princeton.edu

3.2 BINARY SEARCH TREES

- *BSTs*
- *deletion*
- *ordered operations (optional)*

Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

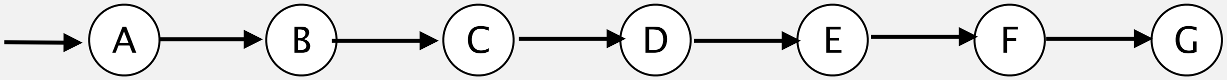
3.2 BINARY SEARCH TREES

- *BSTs*
- *deletion*
- *ordered operations (optional)*

Implementation of a symbol table (a.k.a. associative array)

Ordered Linked List

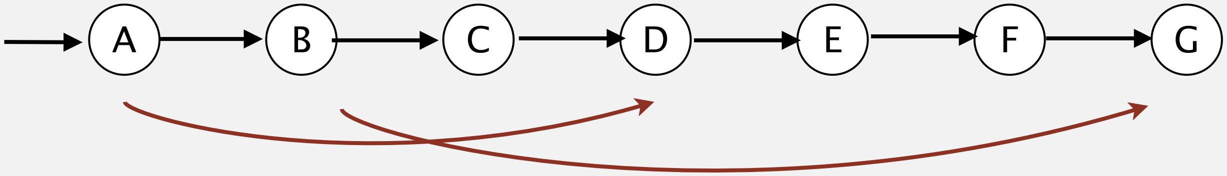
- Slow to find items we want (even though we're in order)



Implementation of a symbol table (a.k.a. associative array)

Ordered Linked List

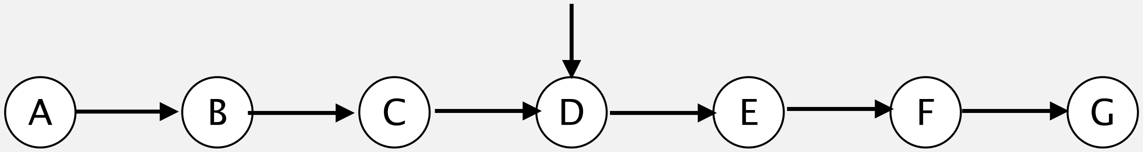
- Slow to find items we want (even though we're in order)
- Adding (random) express lanes: Skip list (won't discuss in 226)



Implementation of a symbol table (a.k.a. associative array)

Ordered Linked List to BST

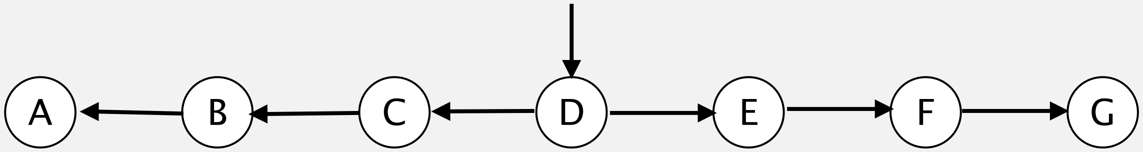
- Slow to find items we want (even though we're in order)
- Move pointer to middle: Can't see earlier elements



Implementation of a symbol table (a.k.a. associative array)

Ordered Linked List to BST

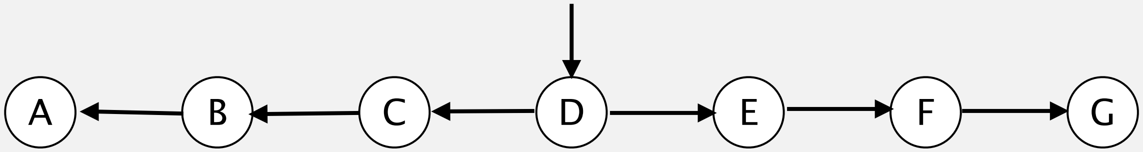
- Slow to find items we want (even though we’re in order).
- Pointer in middle, flip left links: Search time is halved.



Implementation of a symbol table (a.k.a. associative array)

Ordered Linked List to BST

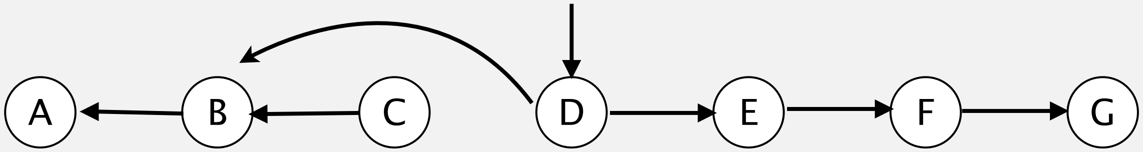
- Slow to find items we want (even though we're in order).
- Pointer in middle, flip left links: Search time is halved.
- Can do better: Dream big!



Implementation of a symbol table (a.k.a. associative array)

Ordered Linked List to BST

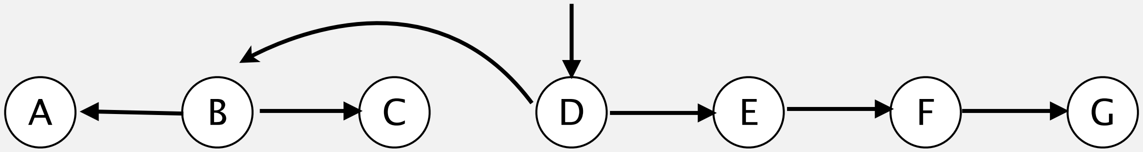
- Slow to find items we want (even though we're in order).
- Pointer in middle, flip left links: Search time is halved.
- Allow every node to make big jumps.



Implementation of a symbol table (a.k.a. associative array)

Ordered Linked List to BST

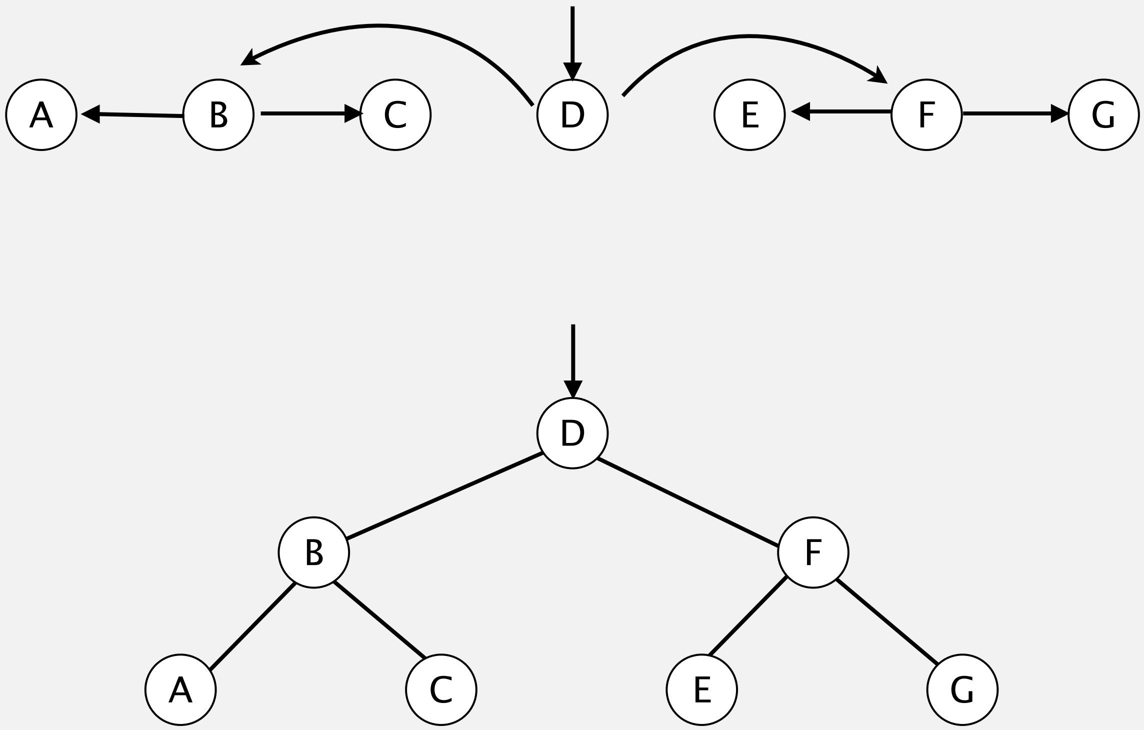
- Slow to find items we want (even though we're in order).
- Pointer in middle, flip left links: Search time is halved.
- Allow every node to make big jumps.



Implementation of a symbol table (a.k.a. associative array)

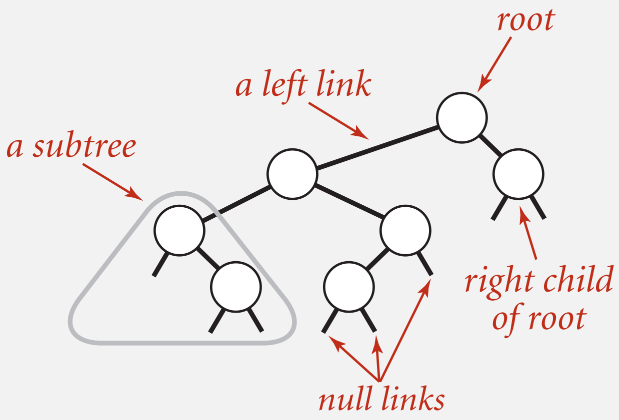
Ordered Linked List to BST

- Slow to find items we want (even though we’re in order).
- Pointer in middle, flip left links: Search time is halved.
- Allow every node to make big jumps.



Binary search trees

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

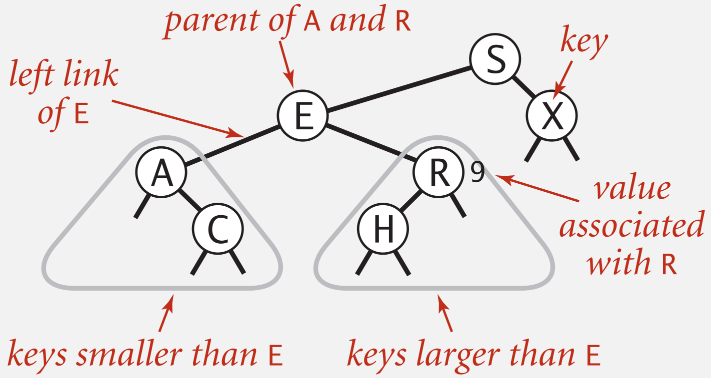


A binary tree is either:

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

Symmetric order. Each node has a key, and every node's key is:

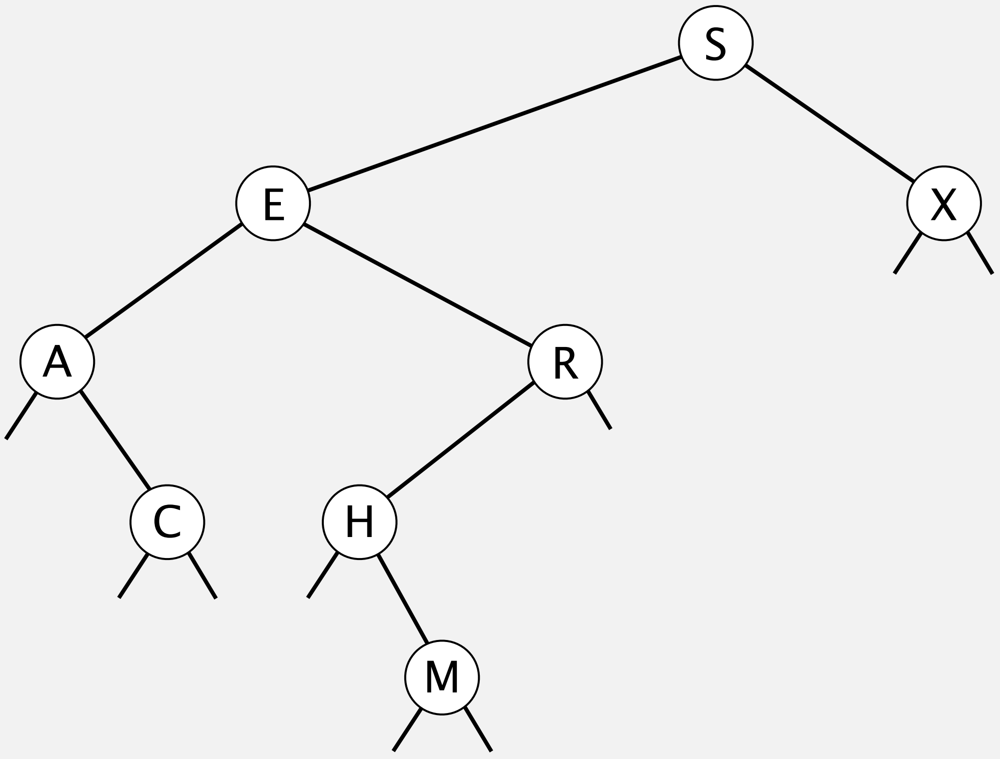
- Larger than all keys in its left subtree.
- Smaller than all keys in its right subtree.



Binary search tree demo

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

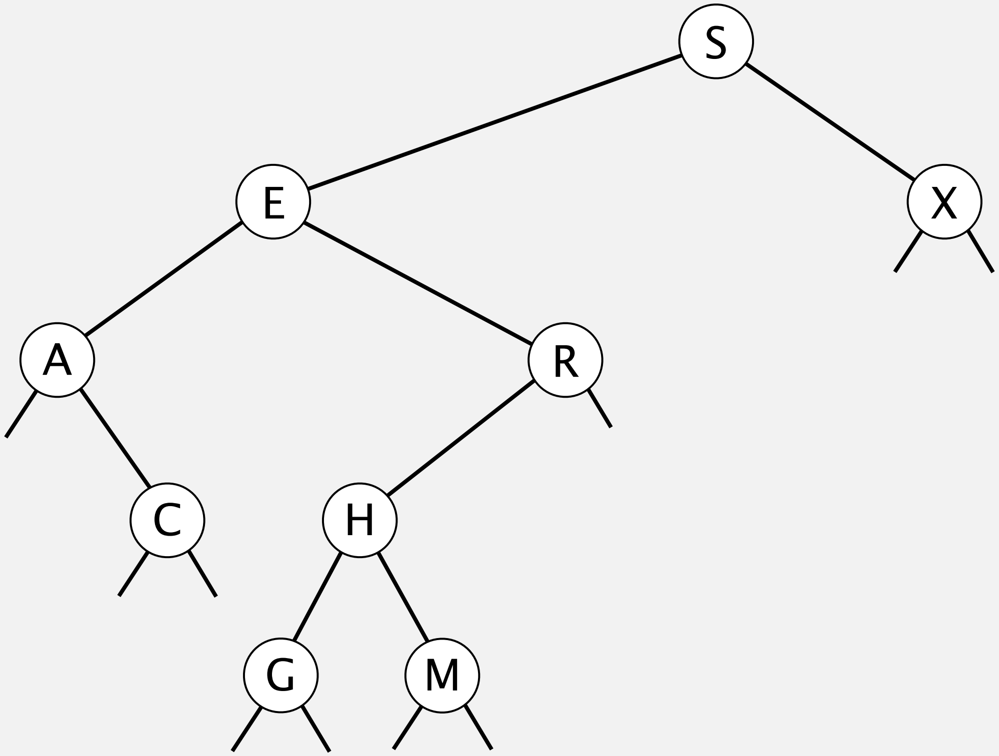
successful search for H



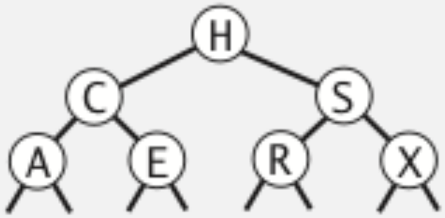
Binary search tree demo

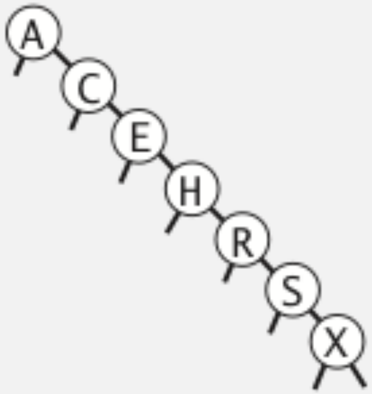
Insert. If less, go left; if greater, go right; if null, insert.

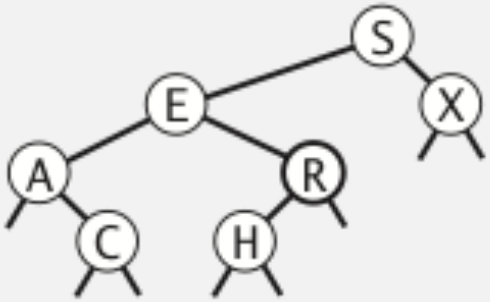
insert G

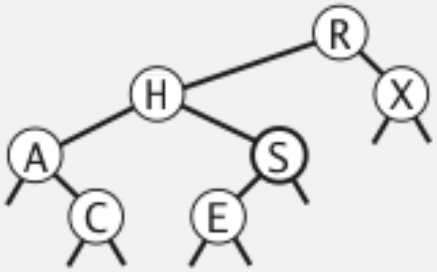


How many BSTs?









pollEv.com/jhug text to **37607**

How many of the figures above are BSTs?

A. 1 [907808]

B. 2 [907809]

C. 3 [907810]

D. 4 [907811]

BST representation in Java

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

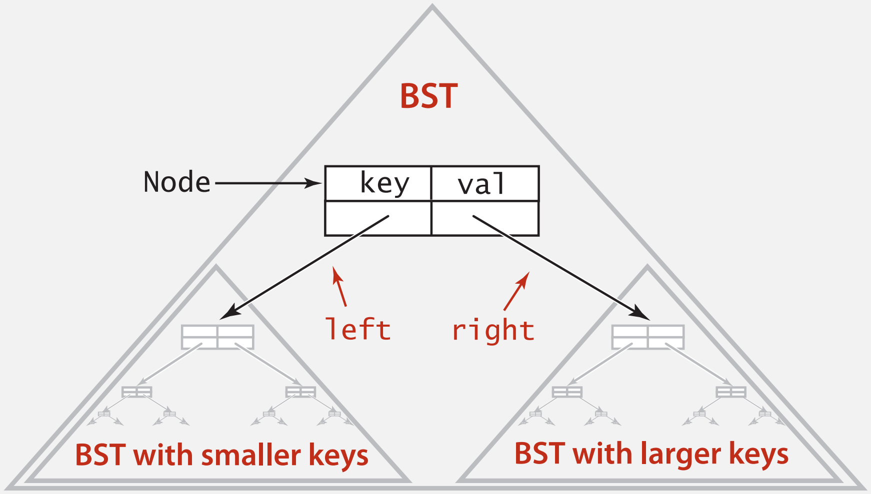
A `Node` is comprised of four fields:

- 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



Binary search tree

BST implementation (skeleton)

```
public class BST<Key extends Comparable<Key>, Value>
{
    private Node root;

   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 void delete(Key key)
   {  /* see next slides */  }

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

}
```

root of BST

BST search: Java implementation

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

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

```
public Value get(Node x, Key key) {
  if (x == null)  return null;
  int cmp = key.compareTo(x.key);
  if (cmp < 0)  return get(x.left, key);
  if (cmp > 0)  return get(x.right, key);
  if (cmp == 0)  return x.value;
}
```

don't write if statements like this! Use else instead!

This code is like this to match the pseudocode on the board.

Cost. Number of compares is equal to 1 + depth of node.

BST search: Java implementation

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

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

```
public Value get(Node x, Key key) {
   if (x == null) return null;

   int cmp = key.compareTo(x.key);
   if (cmp < 0)            return get(key, x.left);
   else if (cmp > 0)       return get(key, x.right);
   else                    return x.value;
}
```

Cost. Number of compares is equal to 1 + depth of node.

BST search: Java implementation

Style warning. Don’t be afraid to rely on your base cases!

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

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

don’t do this!

KdTree. This will be very important for assignment 5 (due after break)!

BST search: Java implementation

Iterative version.

- More intuitive for novices.
- Slightly better performance.
- Harder to prove correctness for experts.
- Much more complex code for fancier trees (stay tuned).

```
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;
}
```

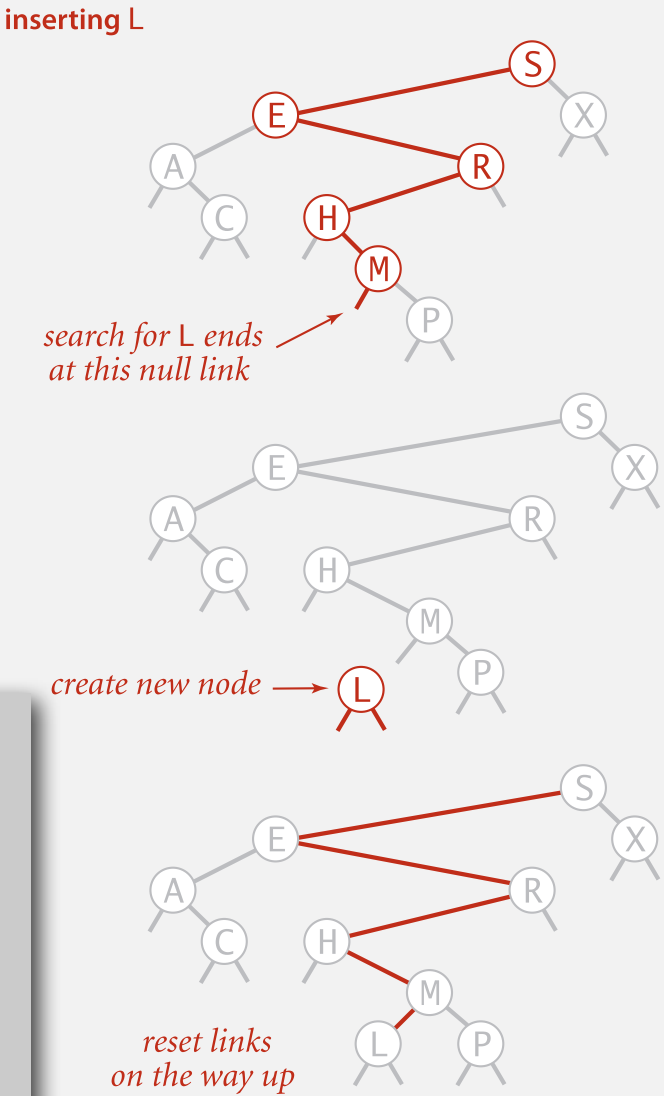
BST insert

Put. Associate value with key.

Search for key, then two cases:

- Key in tree ⇒ reset value.
- Key not in tree ⇒ add new node.

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

private Node put(Node x, Key key, Value val)
{
}
```



Insertion into a BST

BST insert: Java implementation

Put. Associate value with key.

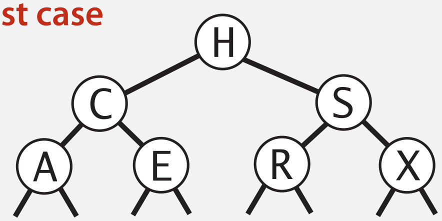
```
 public void put(Key key, Value val)
 {  root = put(root, key, val);  }

 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                  x.value = val;
    return x;
}
```

Cost. Number of compares is equal to 1 + depth of node.

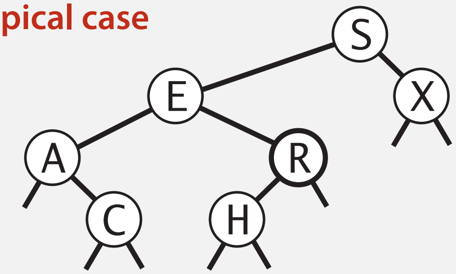
Tree shape

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

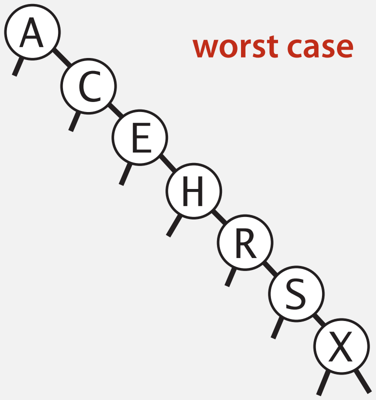
best case



typical case



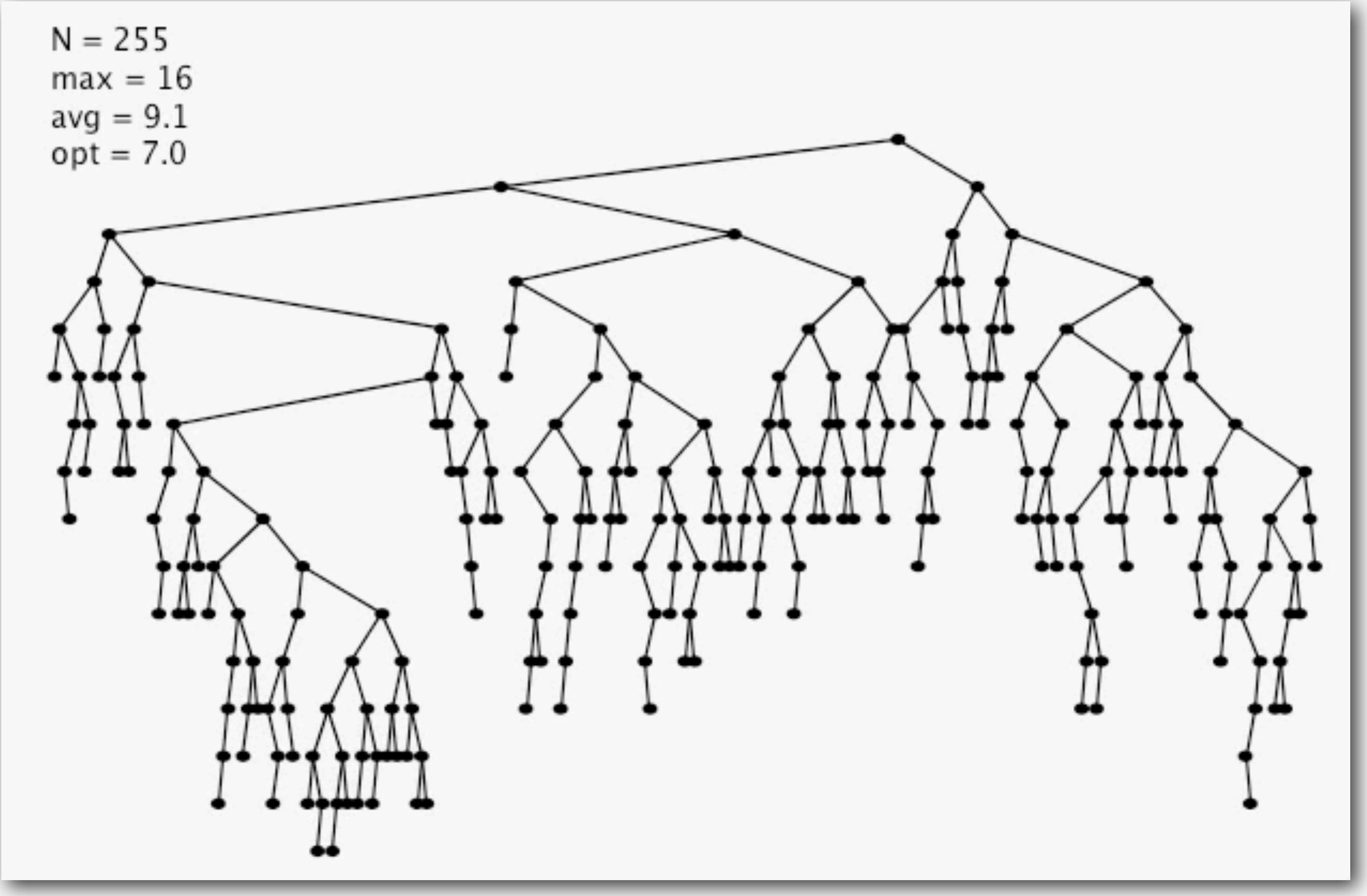
worst case



Remark. Tree shape depends on order of insertion.

BST insertion: random order visualization

Ex. Insert keys in random order.



Sorting using a BST

Proposed sort for arbitrary data.

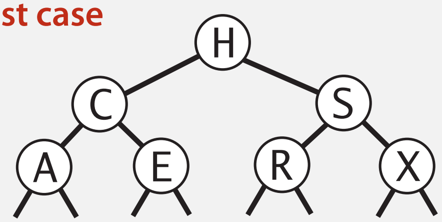
- Insert all items into a binary search tree.
- Print out the tree in order (takes N time, algorithm in a few slides).

pollEv.com/jhug text to **37607**

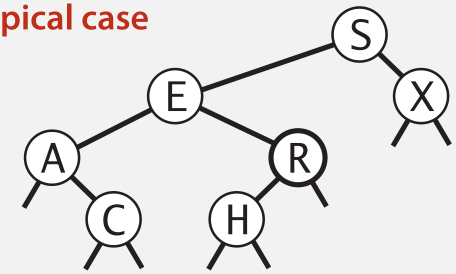
What is the runtime of this sort? (May be more than one right answer)

O(N log N):	Always runs in N log N time or less.	907734
$O(N^2)$:	Always runs in N^2 time or less.	907735
Ω(N log N):	Always runs in N log N time or more.	907736
Θ(N log N):	Always runs in exactly N log N time.	907737

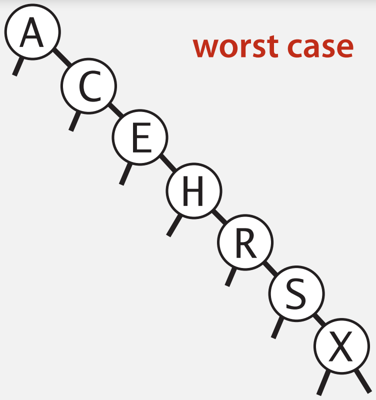
best case



typical case



worst case

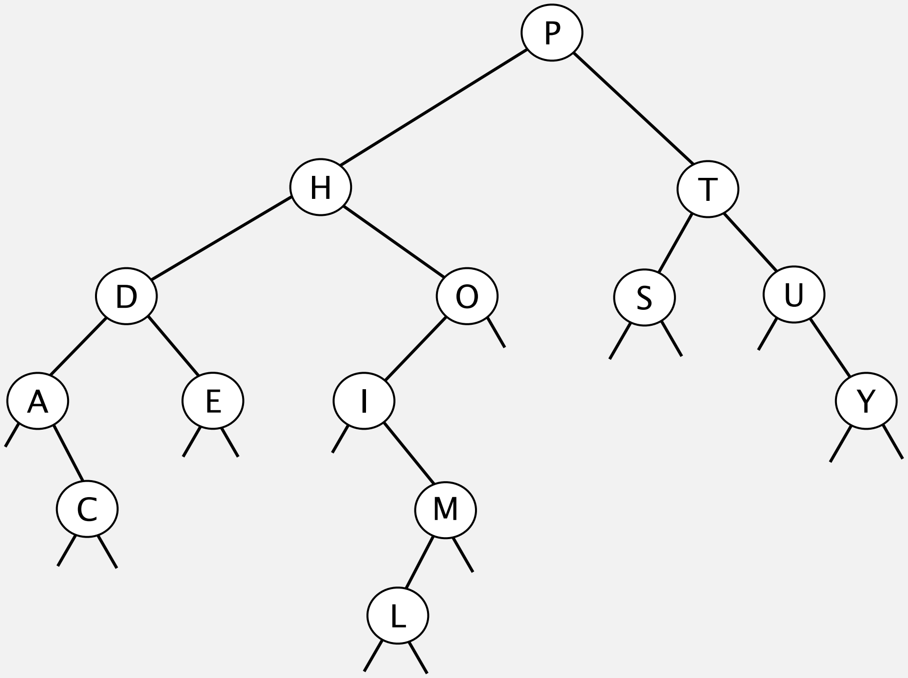


Name for this sort?

Quicksort!

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	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
A	C	D	E	H	I	L	M	O	P	S	T	U	Y



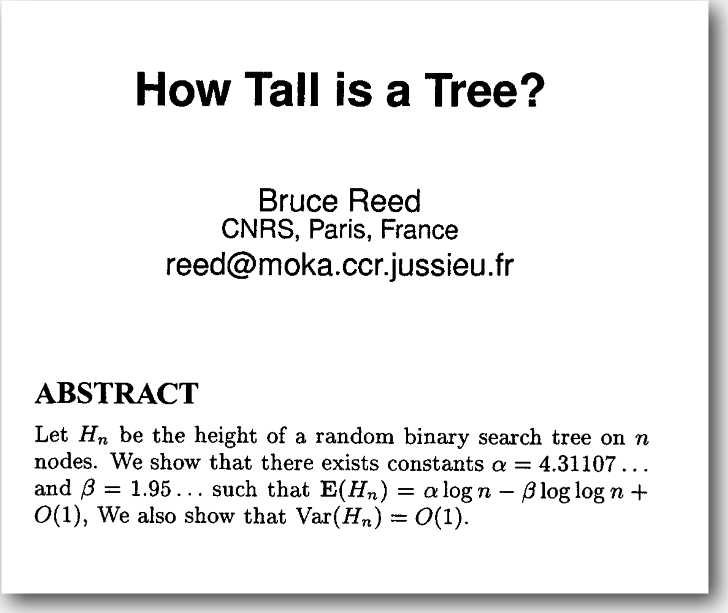
Remark. Correspondence is 1-1 if array has no duplicate keys.

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 (optional: see recurrence relation in book for full proof).

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



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 $\alpha = 4.31107\ldots$ and $\beta = 1.95\ldots$ such that $\mathbf{E}(H_n) = \alpha \log n - \beta \log\log n + O(1)$, We also show that $\mathrm{Var}(H_n) = O(1)$.

But… Worst-case height is N.
(exponentially small chance when keys are inserted in random order)

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	`equals()`
binary search (ordered array)	lg N	N	lg N	N/2	yes	`compareTo()`
BST	N	N	1.39 lg N	1.39 lg N	next	`compareTo()`

Why don't we just shuffle to ensure probabilistic guarantee of height 4.311 ln N?

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

3.2 BINARY SEARCH TREES

- *BSTs*
- *deletion*
- *ordered operations (optional)*

ST implementations: summary

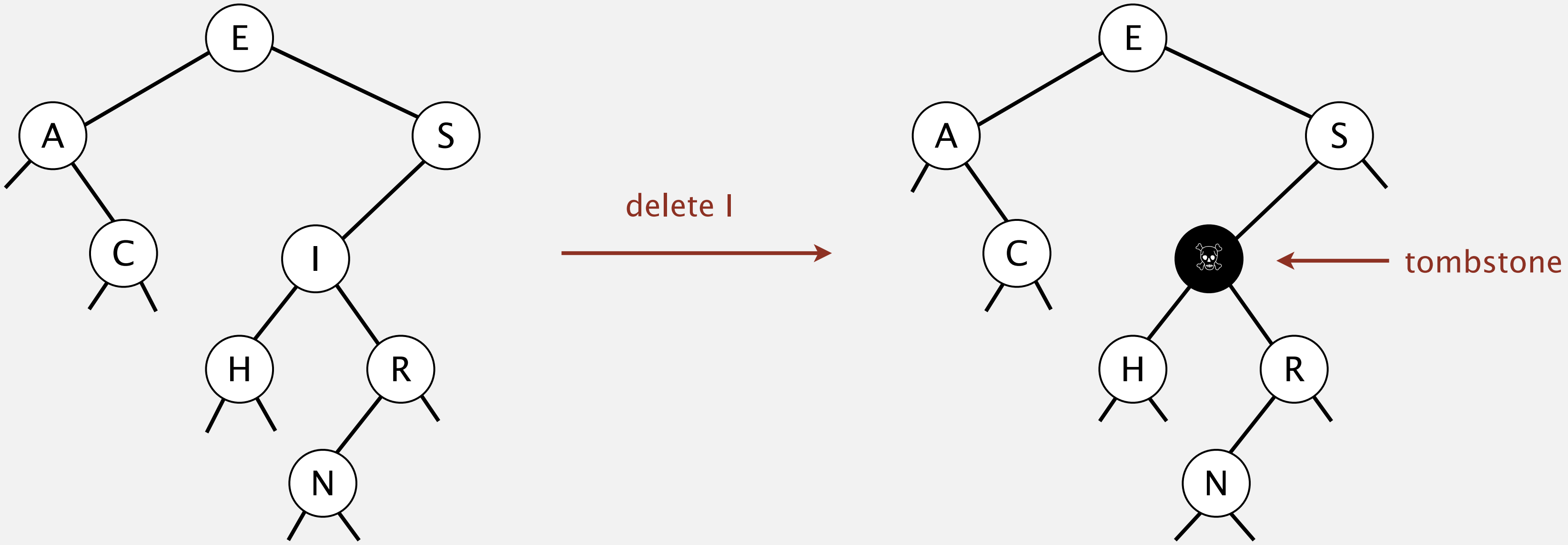
implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.39 lg N	1.39 lg N	???	yes	`compareTo()`

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 search (but don't consider it equal in search).



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.

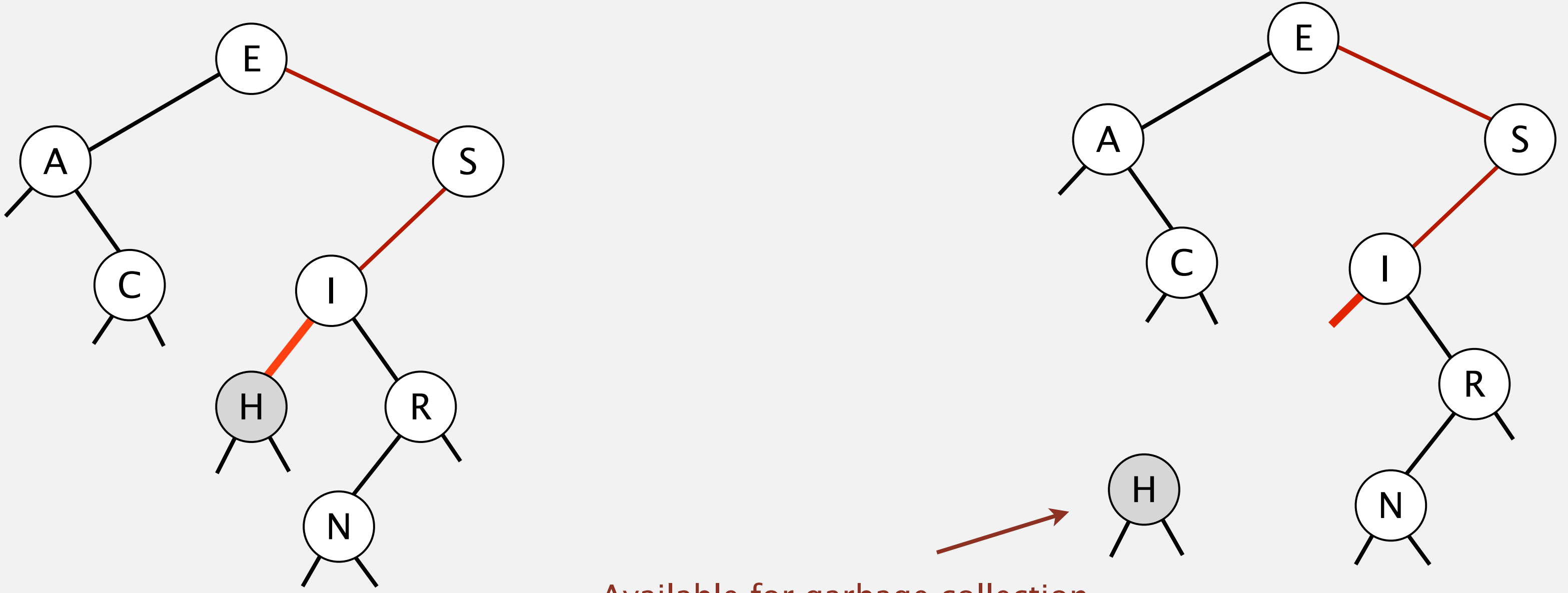
Unsatisfactory solution. Tombstone (memory) overload.

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.

Example. `delete(H)`



Available for garbage collection

Recursive Call. Much like `put()`, visited nodes return a new pointer used by parent. Example: When `x = I: x.left = delete(x.left, H);`

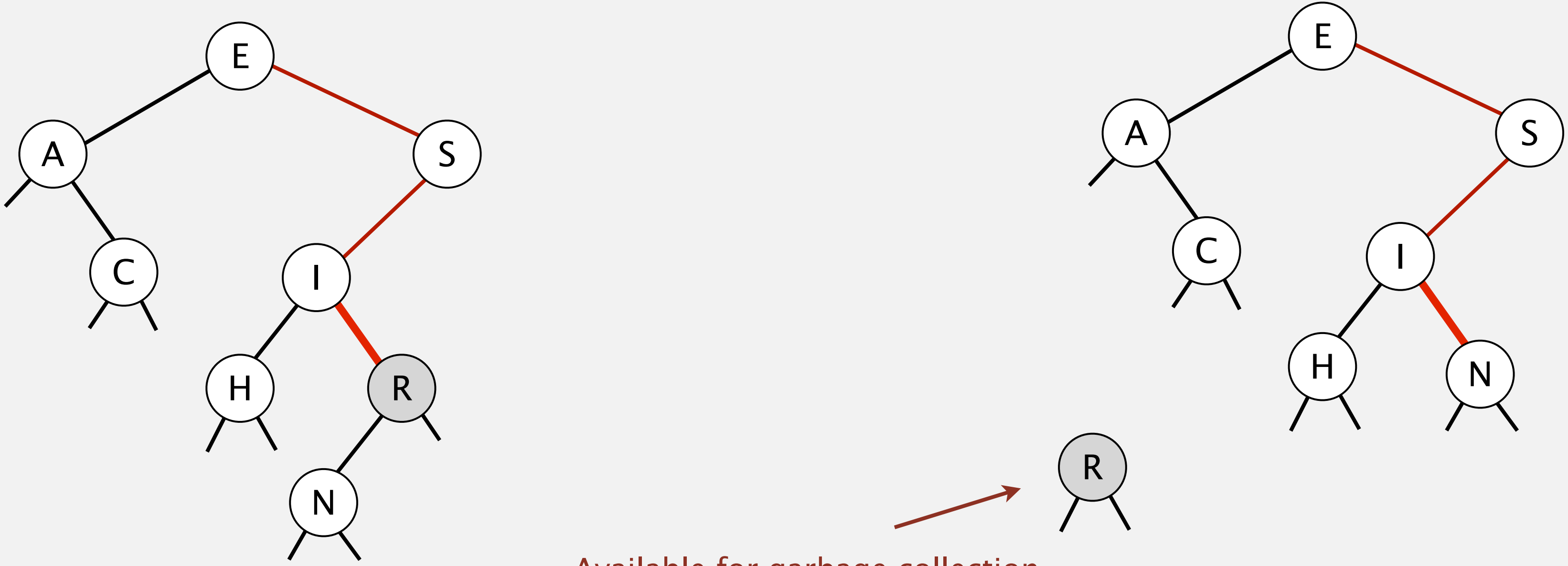
When `x = H: return null;`

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.

Example. `delete(R)`



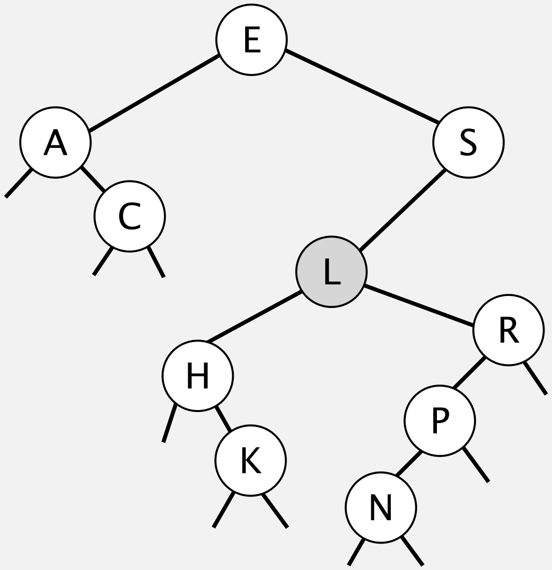
Available for garbage collection

Hibbard deletion

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

Case 2. [2 children] Delete `t` by replacing parent link.

Example. `delete(L)`



pollEv.com/jhug text to **37607**

Which key could we move into L’s place and still have a BST?

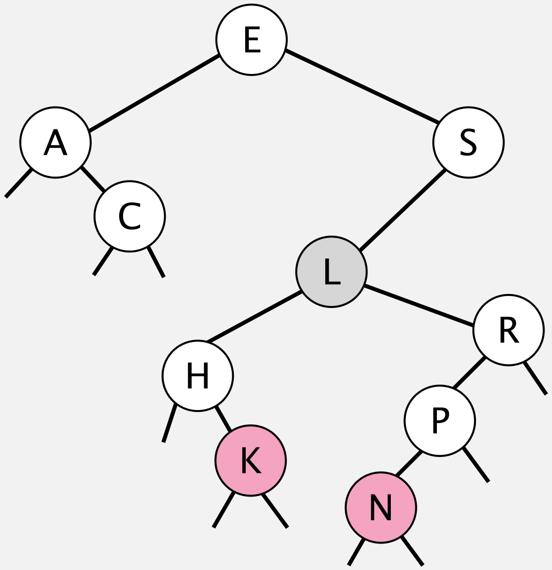
A	[907394]
H	[907395]
K	[907396]
R	[907397]
P	[907398]
N	[907399]

Hibbard deletion

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

Case 2. [2 children] Delete `t` by replacing parent link.

Example. `delete(L)`



Choosing a replacement.

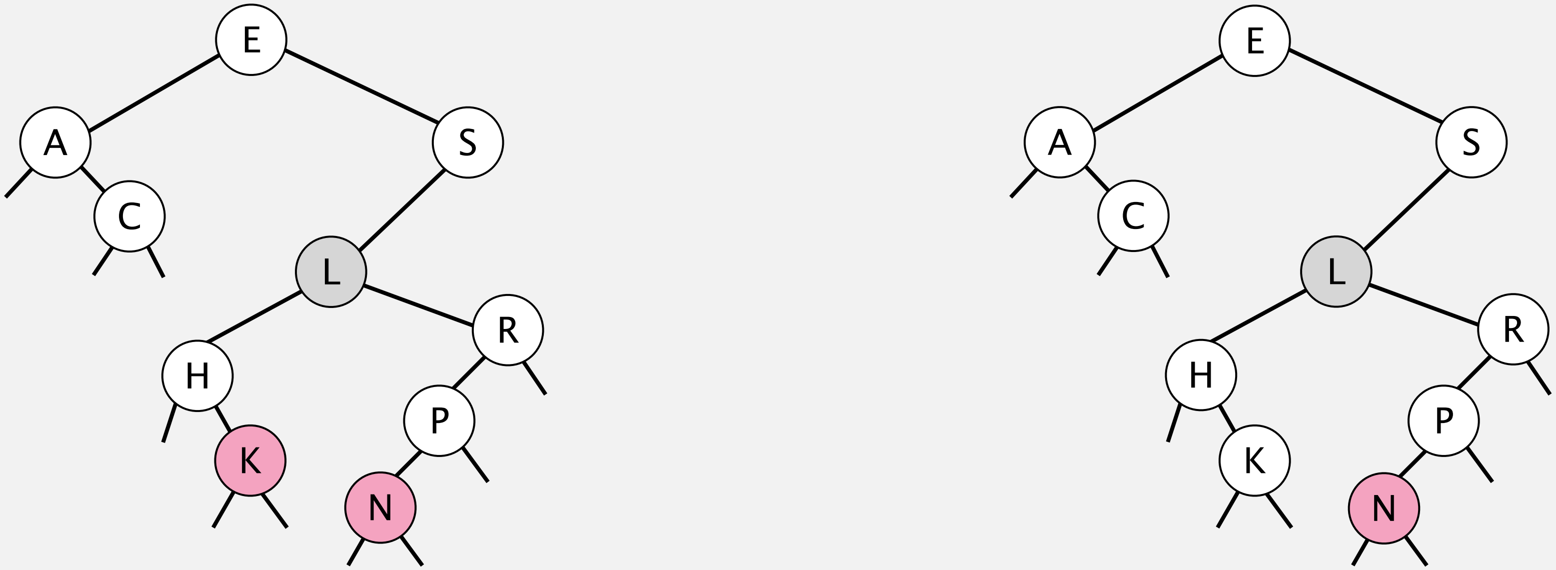
- Successor: N
- Predecessor: K

Hibbard deletion

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

Case 2. [2 children] Delete `t` by replacing parent link.

Example. `delete(L)`



Choosing a replacement.

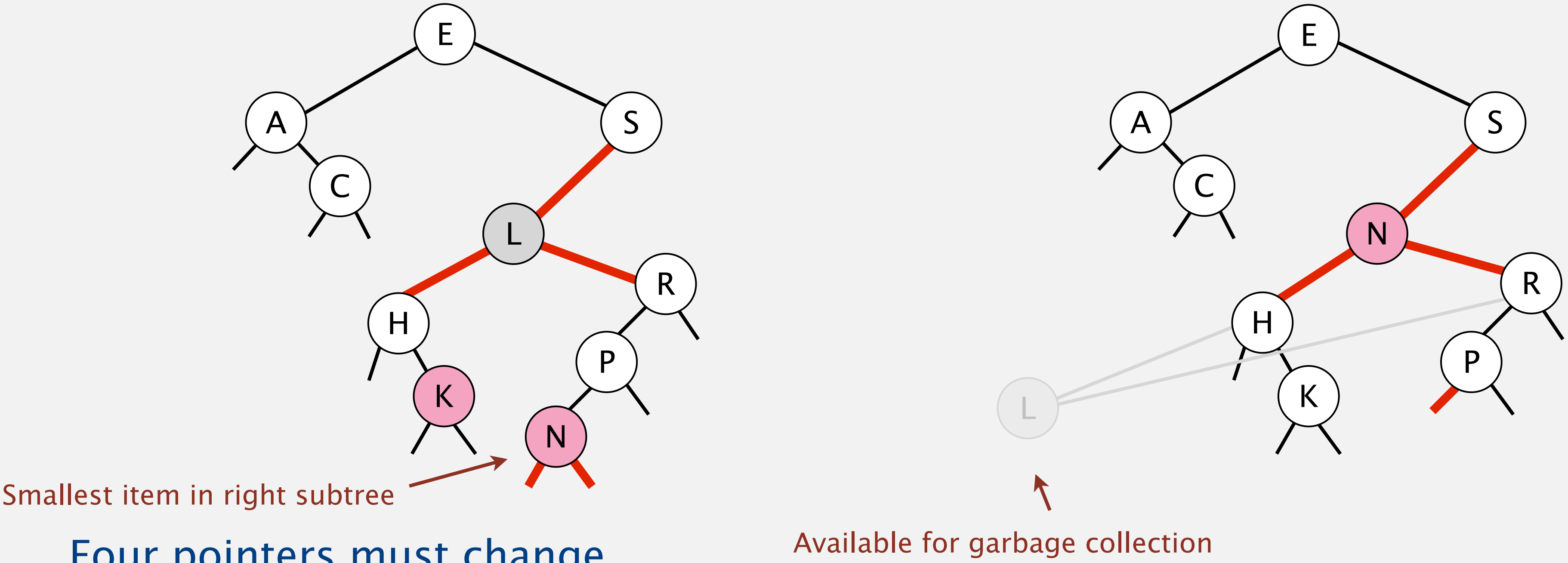
- Successor: N [by convention]
- ~~Predecessor: K~~

Hibbard deletion

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

Case 2. [2 children] Delete `t` by replacing parent link.

Example. `delete(L)`



Four pointers must change.

- Parent of deleted node
- Parent of successor
- Left child of successor
- Right child of successor

Hibbard deletion: Java implementation

Note: this code is way too much to digest in lecture! Look again later.

```
public void delete(Key key)
{  root = delete(root, key);  }

private Node delete(Node x, Key key) {
   if (x == null) return null;
   int cmp = key.compareTo(x.key);
   if      (cmp < 0) x.left  = delete(x.left,  key);
   else if (cmp > 0) x.right = delete(x.right, key);
   else {
      if (x.right == null) return x.left;
      if (x.left == null) return x.right;

      Node t = x;
      x = min(t.right);
      x.right = deleteMin(t.right);
      x.left = t.left;
   }
   x.count = size(x.left) + size(x.right) + 1;
   return x;
}
```

search for key

no right child

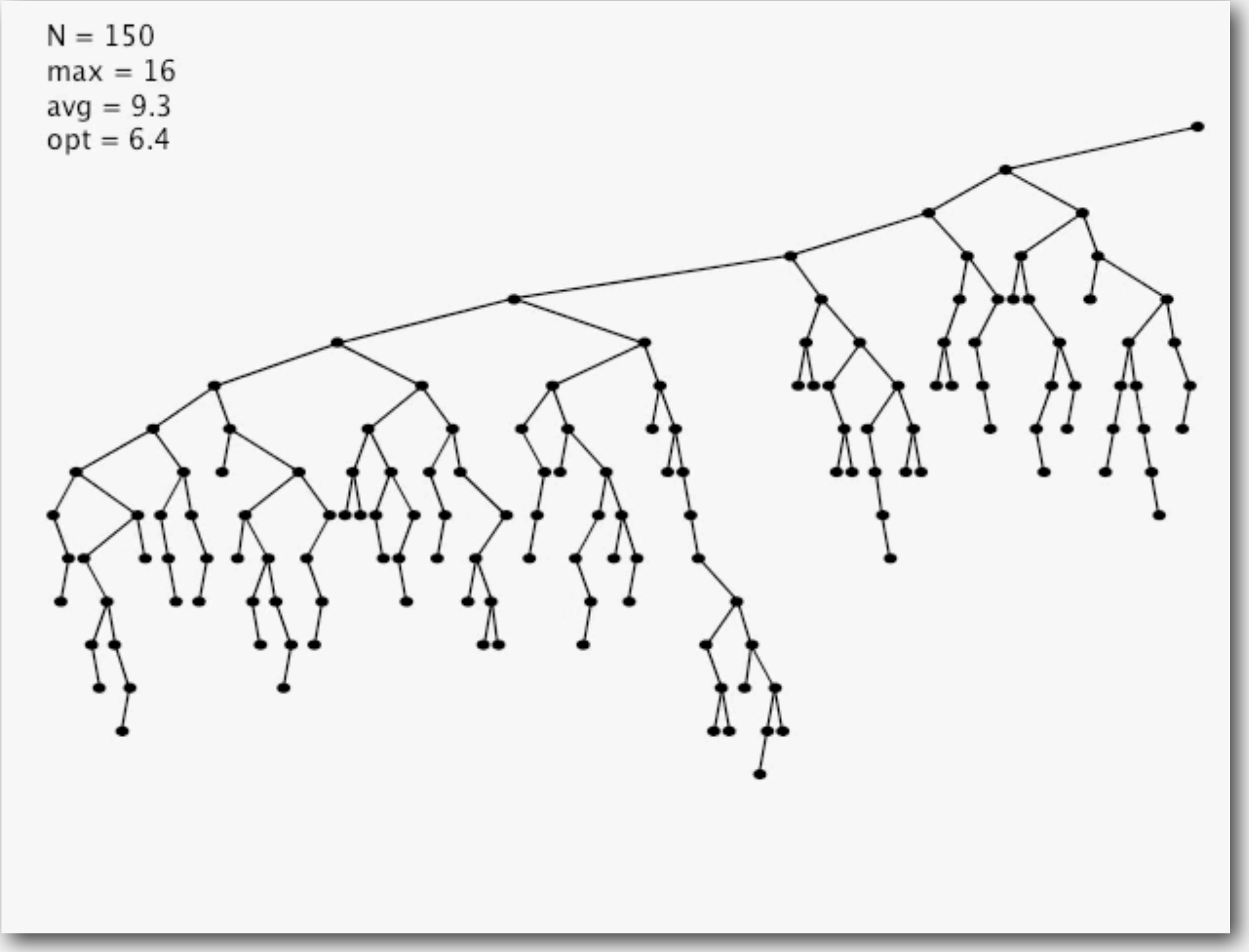
no left child

replace with successor

update subtree counts

Hibbard deletion: analysis

Unsatisfactory solution. Not symmetric.



Surprising consequence. Trees not random (!) ⇒ 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	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.39 lg N	1.39 lg N	√N	yes	`compareTo()`

other operations also become √N
if deletions allowed

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

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

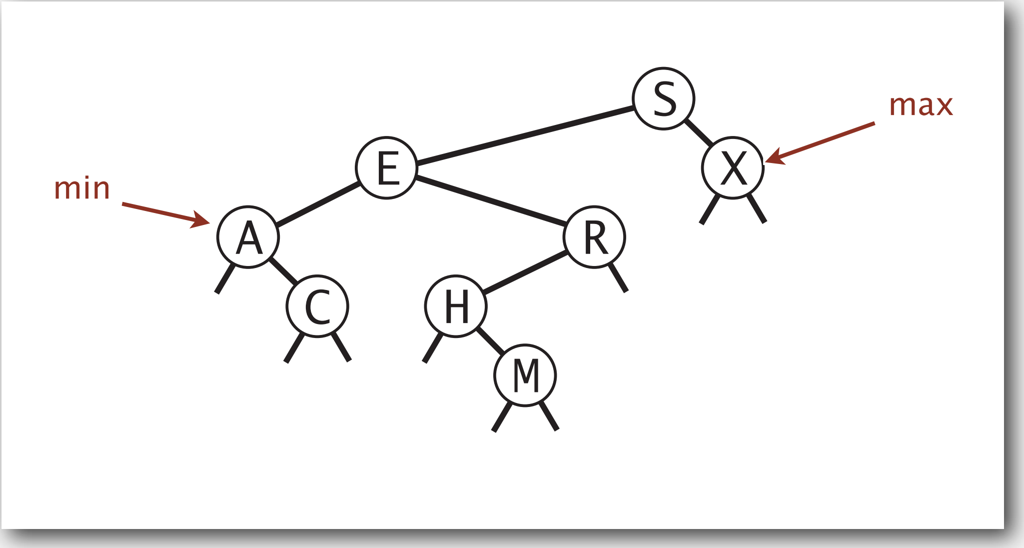
3.2 BINARY SEARCH TREES

- *BSTs*
- *deletion*
- *ordered operations (optional)*

Minimum and maximum

Minimum. Smallest key in table.

Maximum. Largest key in table.

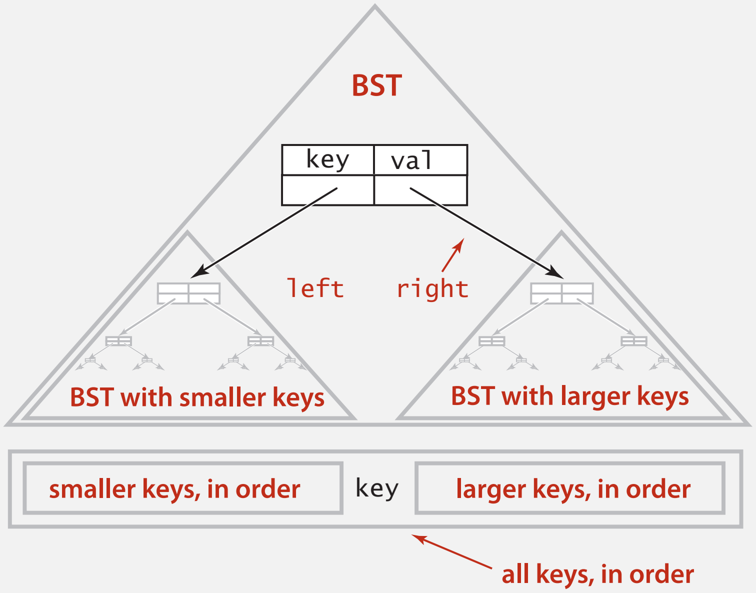


Q. How to find the min / max?

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);
}
```

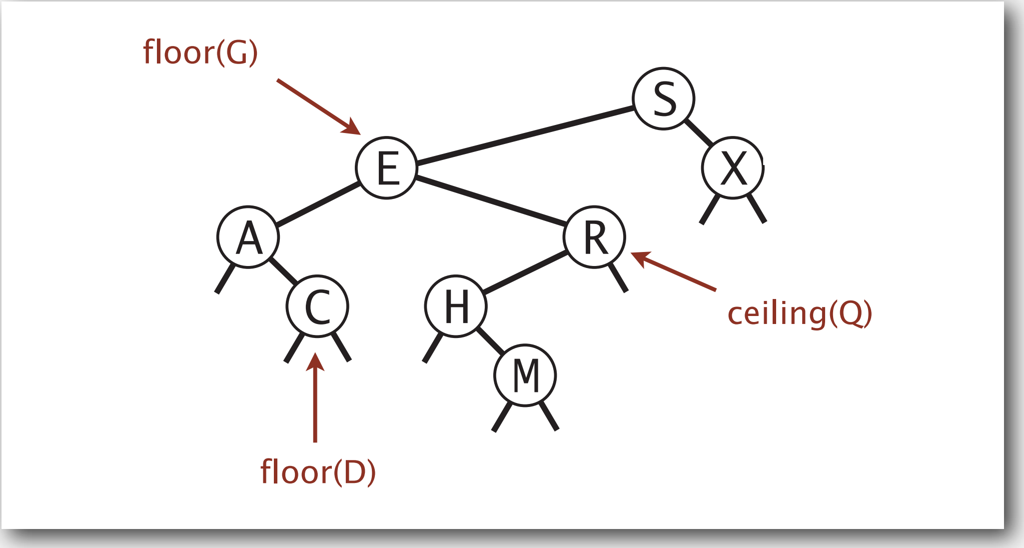


Property. Inorder traversal of a BST yields keys in ascending order.

Floor and ceiling

Floor. Largest key ≤ a given key.

Ceiling. Smallest key ≥ a given key.



Q. How to find the floor / ceiling?

Computing the floor of k

Case 1. [k equals the key at root]

The floor of k is k.

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

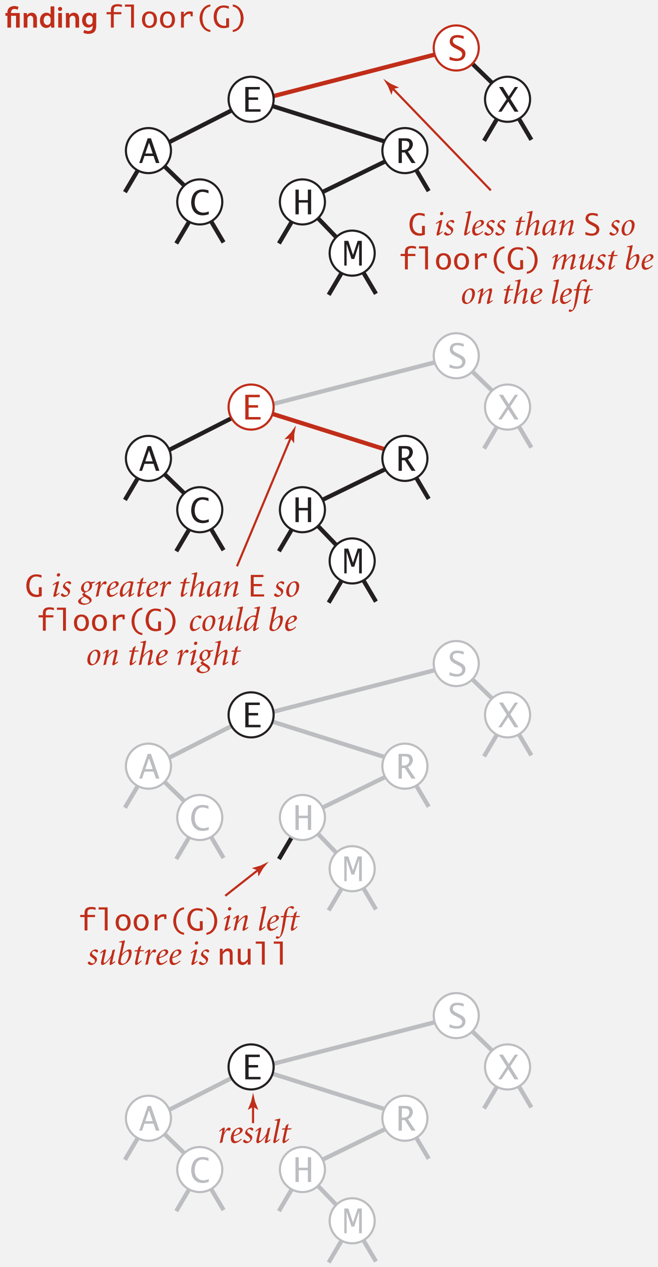
The floor of k is in the left subtree.

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

The floor of k is in the right subtree

(if there is **any** key ≤ k in right subtree);

otherwise it is the key in the root.



finding `floor(G)`

`G` *is less than* `S` *so* `floor(G)` *must be on the left*

`G` *is greater than* `E` *so* `floor(G)` *could be on the right*

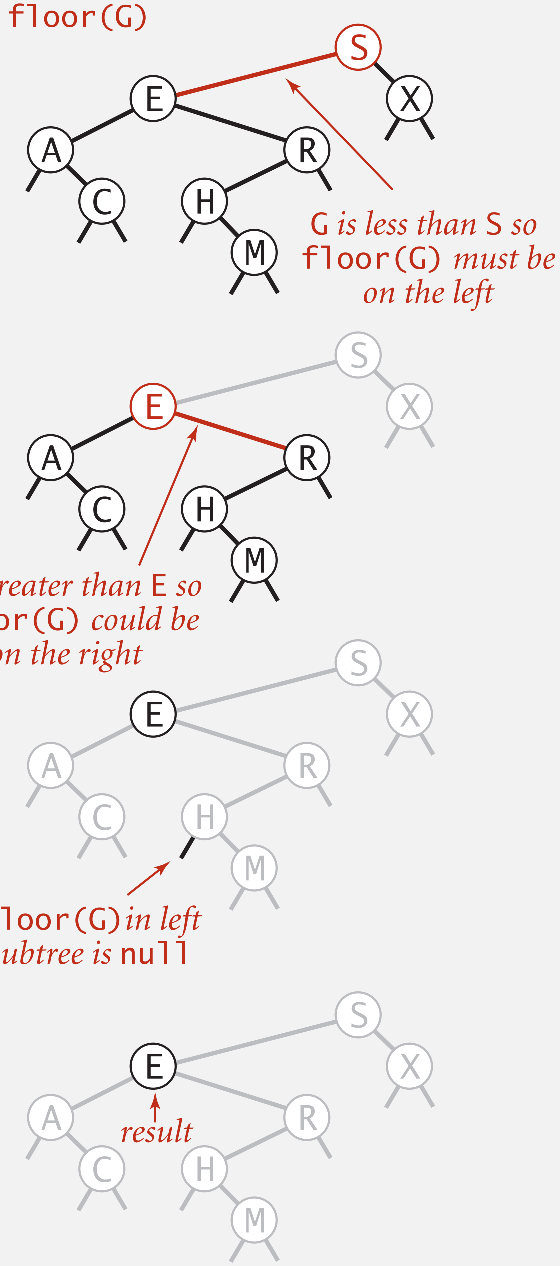
`floor(G)` *in left subtree is* `null`

result

Computing the floor

```
public Key floor(Key key)
{
   Node x = floor(root, key);
   if (x == null) return null;
   return x.key;
}
private Node 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);

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

}
```

finding `floor(G)`



G is less than S so `floor(G)` *must be on the left*

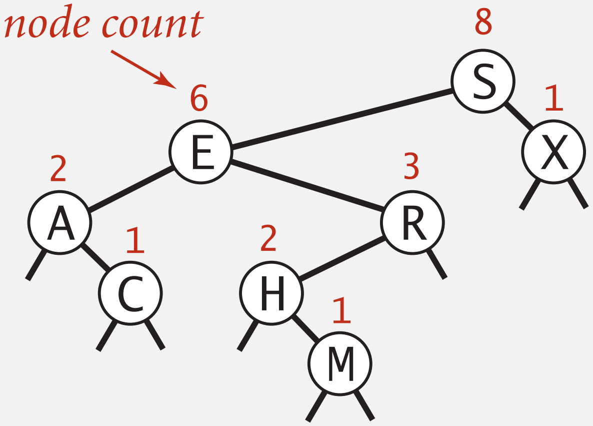
G is greater than E so `floor(G)` *could be on the right*

`floor(G)` *in left subtree is* `null`

result

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 count;
}
```

number of nodes in subtree

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

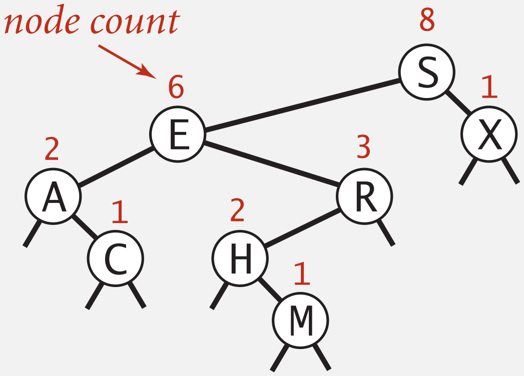
```
private int size(Node x)
{
   if (x == null) return 0;
   return x.count;
}
```

ok to call when x is null

```
private Node put(Node x, Key key, Value val)
{
   if (x == null) return new Node(key, val);
   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;
}
```

Rank

Rank. How many keys < 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);
}
```

BST: ordered symbol table operations summary

	sequential search	binary search	BST
search	N	lg N	h
insert	N	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

order of growth of running time of ordered symbol table operations

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