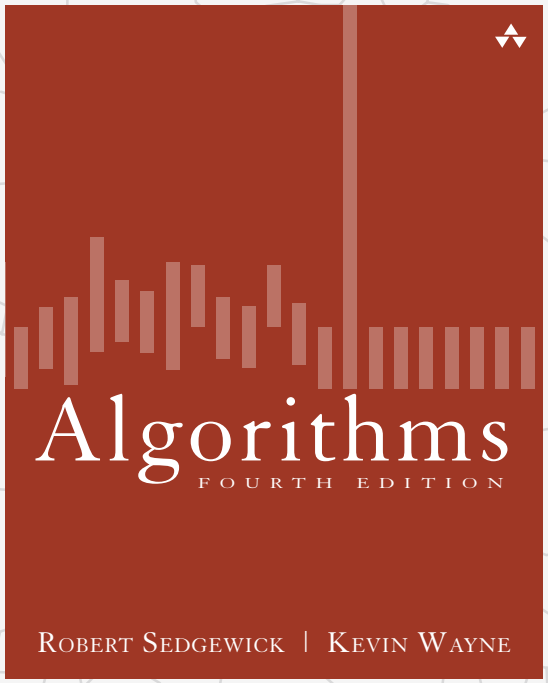


http://algs4.cs.princeton.edu

3.2 BINARY SEARCH TREES

- *BSTs*
- *ordered operations*
- *deletion*

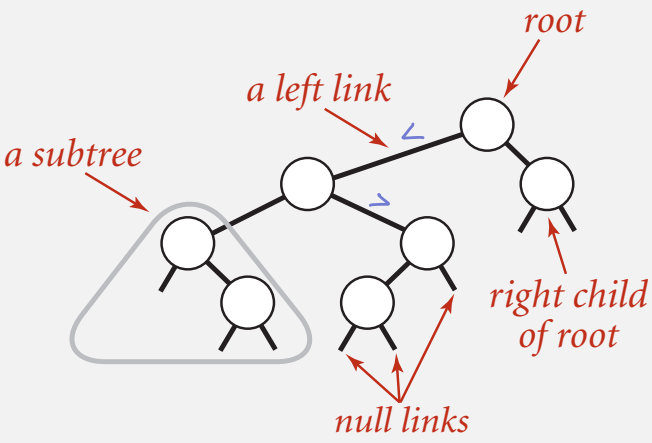
http://algs4.cs.princeton.edu

3.2 Binary Search Trees

- *BSTs*
- *ordered operations*
- *deletion*

Binary search trees

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

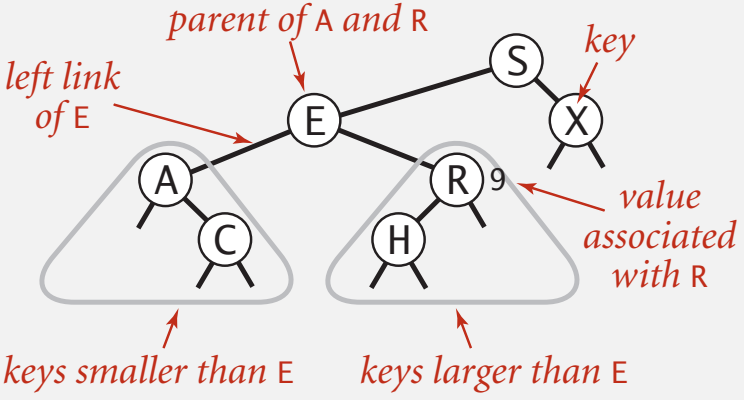


A binary tree is either:

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

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

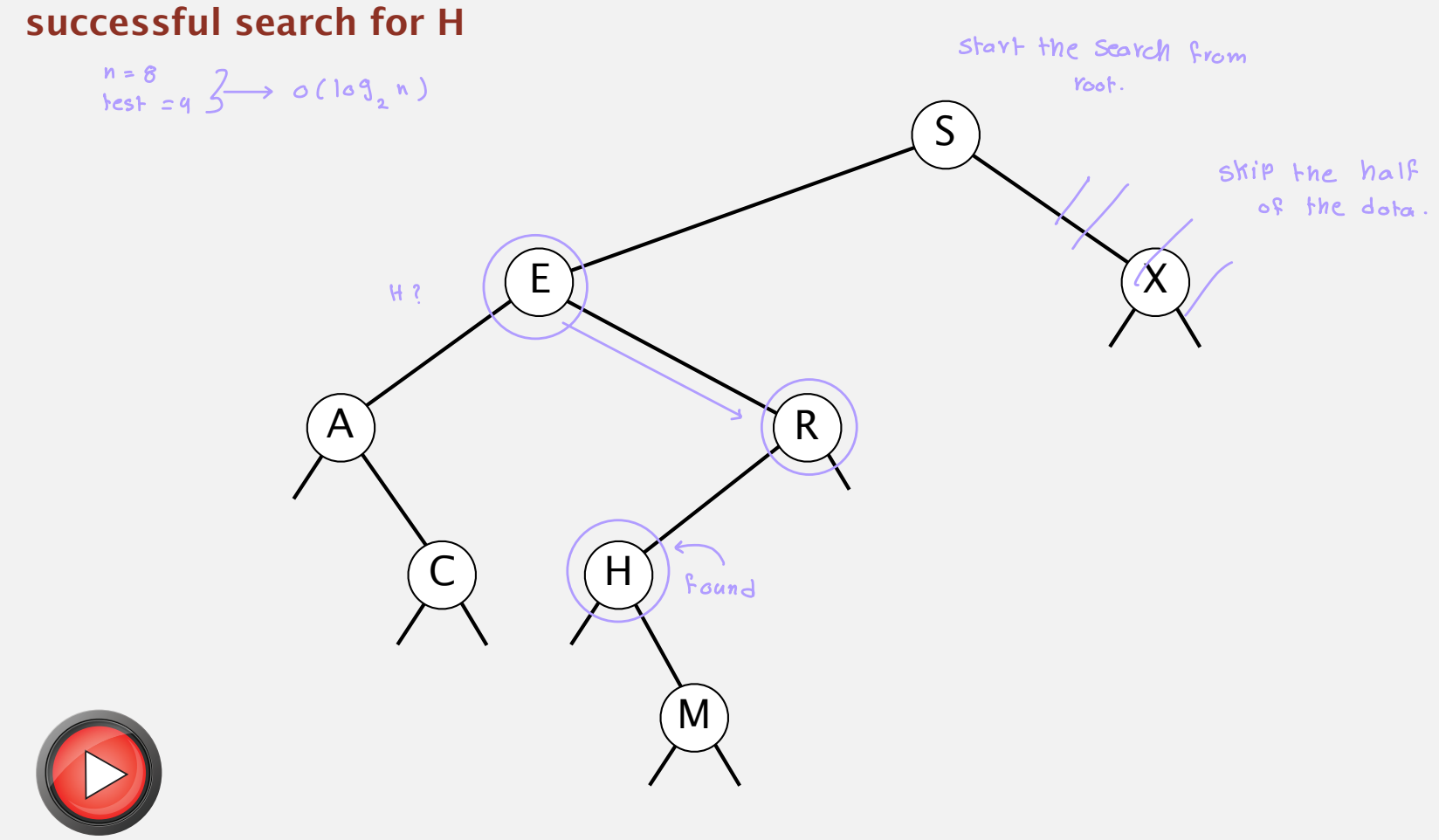
- Larger than all keys in its left subtree.
- Smaller than all keys in its right subtree.



Binary search tree demo

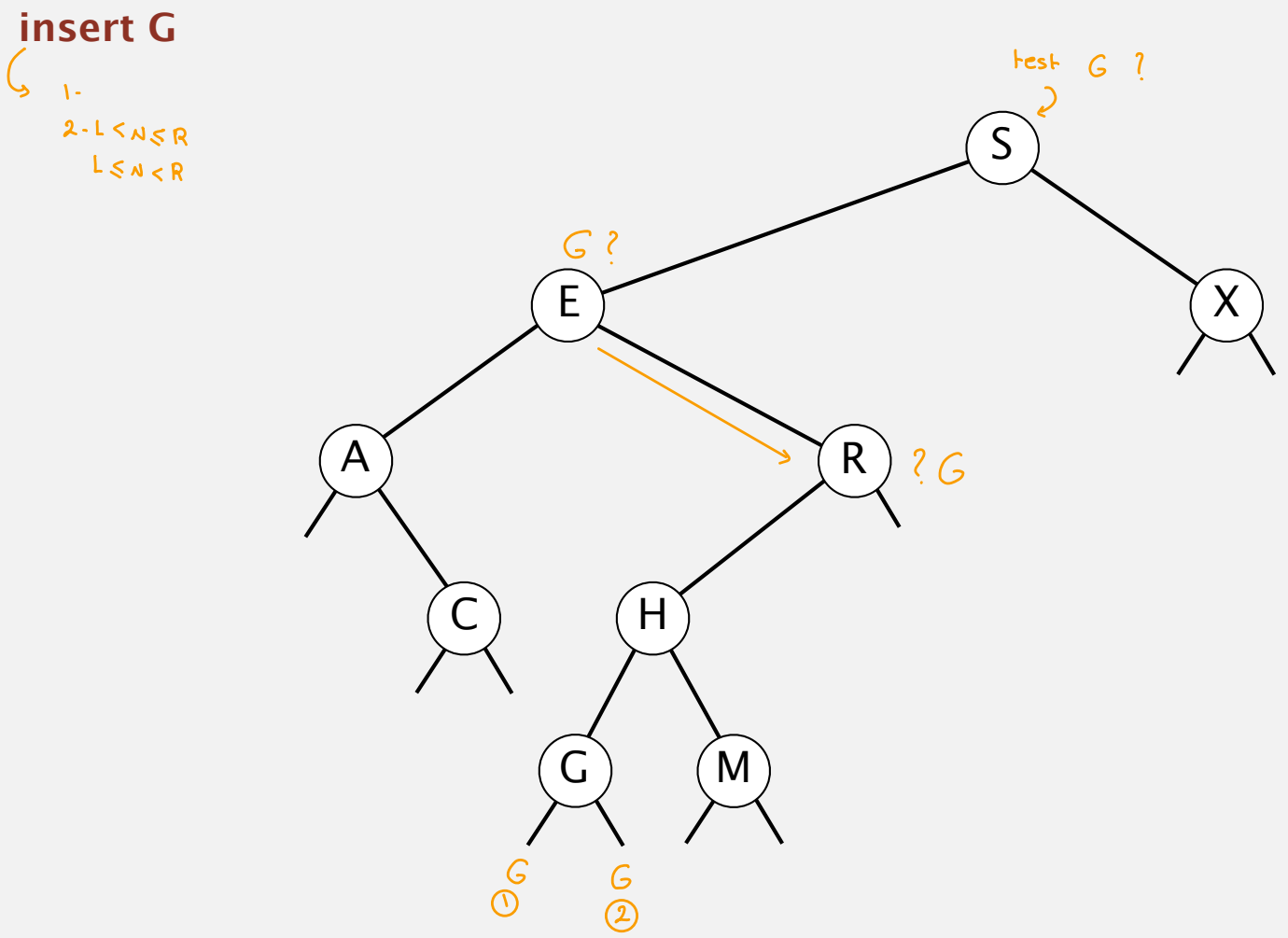
Search. If less, go left; if greater, go right; if equal, search hit.

successful search for H



Binary search tree demo

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



BST representation in Java

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

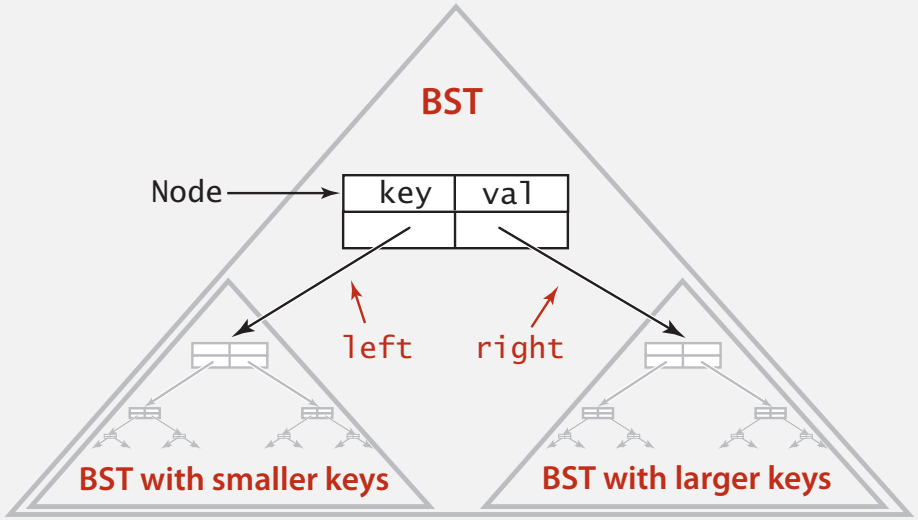
A Node is composed of four fields:

- 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;
   }
}
```

must be unique data.



Binary search tree

Key and Value are generic types; Key is Comparable

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

insert

search

BST search: Java implementation

must be orderd

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

Tree Search :

عندنا ثلاث احتمالات

1- يا اما متساويها .

2- يا اما اكبر منها .

3- يا اما انها null .

4- يا اما اصغر منها

اذا كانت (External) :

- معناتها : null مش موجودة وبتعمل return الـ Value .

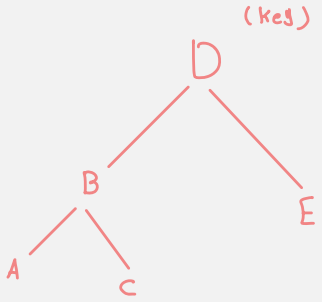
- واذا كانت اقل بتنادي الـ function من الجهة اليسار .

- اذا كانت تساوي معناته لقيها بيرجع الـ Value .

- اذا كانت اكبر بتنادي الـ function من الجهة اليمين .

```
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;
}
```

Element does not exist.



$O(\log n + 1)$ → we add one to search

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

* كل ما كبرت الـ array
صار اسرع في الحسابات .

1	3	5	7	9
0	1	2	3	4

Search for (7)

first = 0
last = 4
mid = (0+4) / 2 = 2

a[2] = 7
5 < 7

first = Mid + 1 = 3
last = 4
mid = (3+4) / 2 = 3

a[3] = 7 Stop found
7 = 7

Search for (6) :

first = 0
last = 4
mid = (0+4) / 2 = 2

a[2] : 6
5 < 6

first = mid + 1 = 3
last = 4
mid = (3+4) / 2 = 3

a[3] ? 6
7 > 6

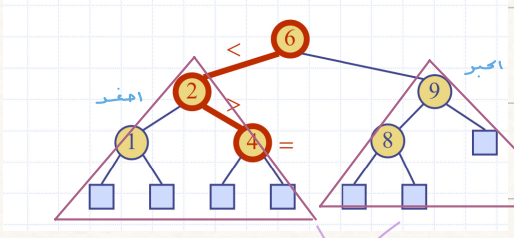
لمن يكون منتهي اهل
بيحرك الـ last .

first = 3
* last = mid - 1 = 2

لمن الـ last اصغر من الـ first حاتم الـ first اصغر من last
لان الـ first يجي من قدام ، والـ last من وراء ، فا الـ first قيمته كانديكس اصغر من الـ last .

فا لمن يصير كذا الـ first اكبر من الـ last معناته بعمل (Stop) و not found .

* key(w) ≤ key(v) ≤ key(w)
left < root ≤ right



* immediate successor : الي بعدة مباشرة ، بتحرك وحدة right وبعدين لاقصى left

* immediate Predecessor : الي قبلة مباشرة ، بتحرك وحده left وبعدين لاقصى right.

بيكون اكثر شي ، right . ← ايش اكبر قيمة فيها :

بيكون اكثر شيء left . ← ايش اصغر قيمة فيها :

من الي بعد ك ؟ : بتحرك وحده right .

بعدين left ، left ، left لاقصى left
ويكون هذا
الي بعد ك .

من قبل الـ ك ؟ : بتحرك وحدة left بعدين

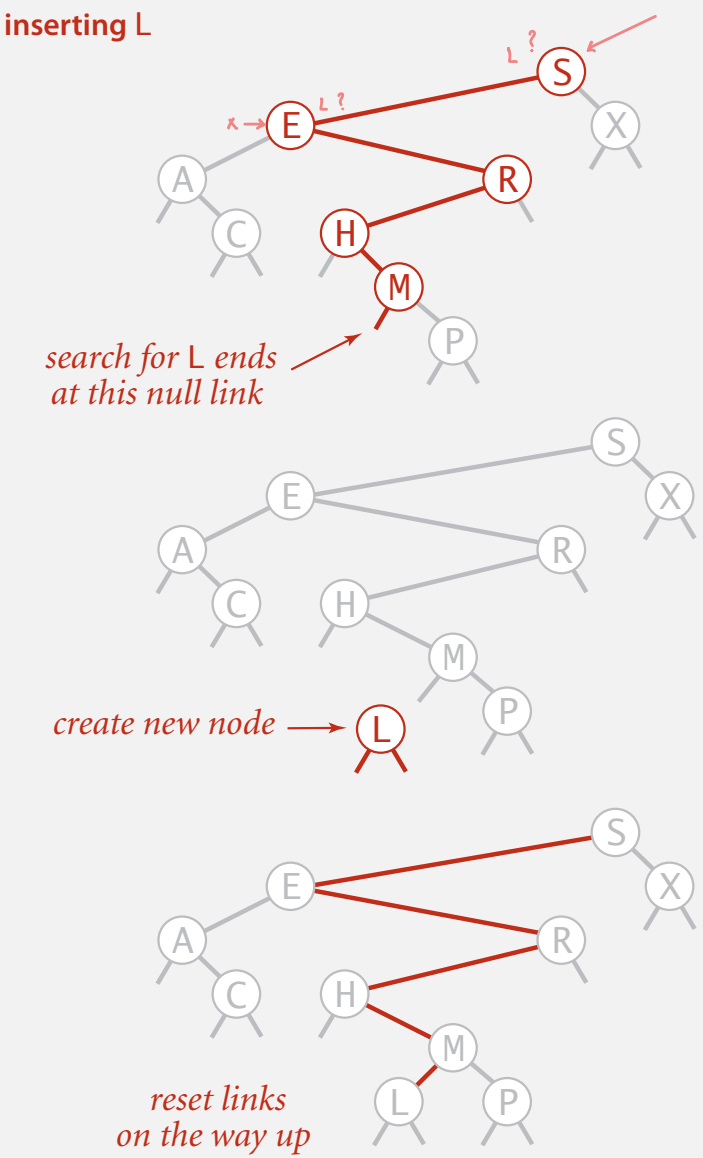
right ، right ، right لاقصى الـ right وبتكون هذي قبل الـ ك .

BST insert

Put. Associate value with key.

Search for key, then two cases:

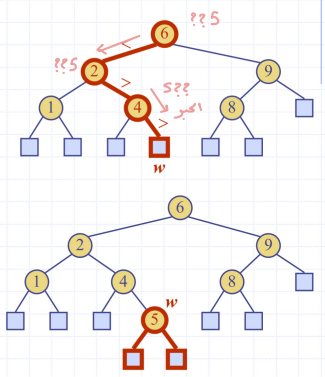
- Key in tree ⇒ reset value.
- Key not in tree ⇒ add new node.



Insertion into a BST

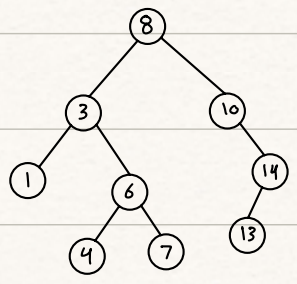
Insertion

- To perform operation put(k, o), we search for key k (using TreeSearch)
- Assume k is not already in the tree, and let w be the leaf reached by the search
- We insert k at node w and expand w into an internal node
- Example: insert 5



Insertion

- Insert the following elements in a BST



BST insert: Java implementation

Put. Associate value with key.

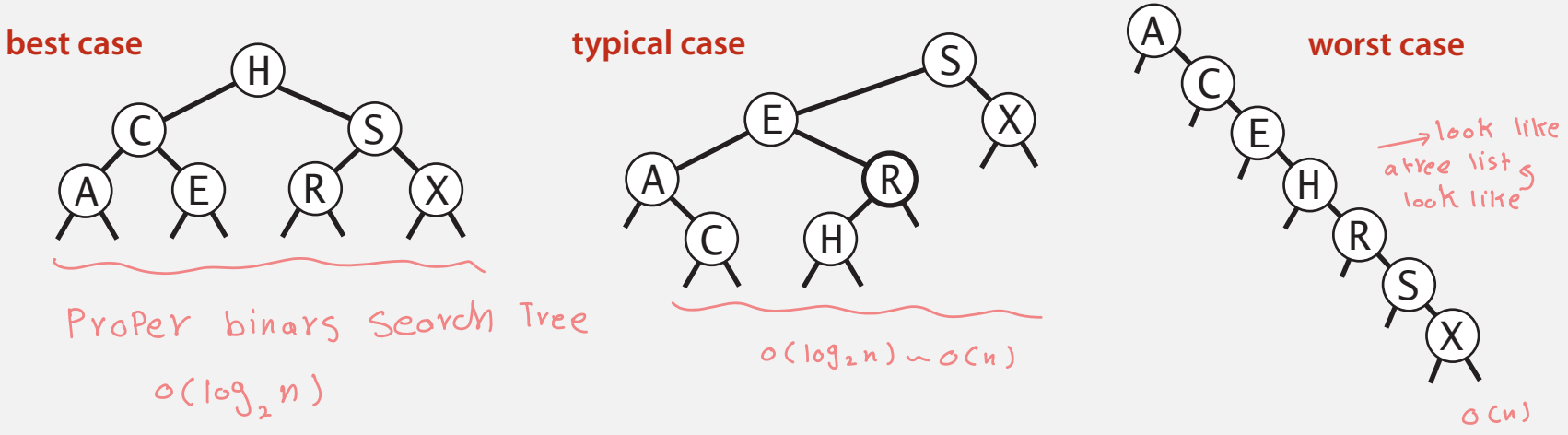
concise, but tricky, recursive code; read carefully!

```
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;
}
```

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

Tree shape

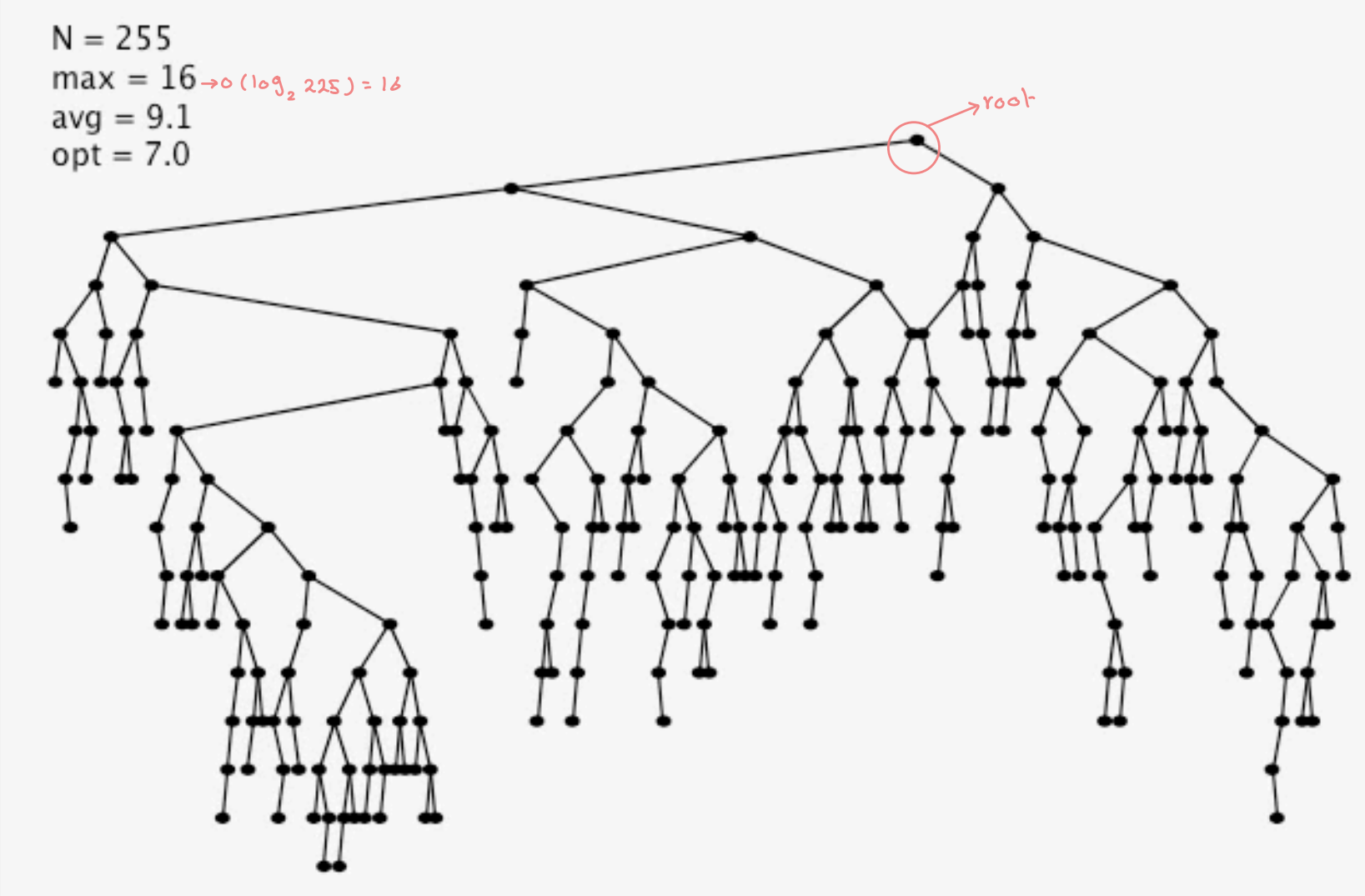
- Many BSTs correspond to same set of keys.
- Number of compares for search/insert is equal to 1 + depth of node.



Bottom line. Tree shape depends on order of insertion.

BST insertion: random order visualization

Ex. Insert keys in random order.



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 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 $\mathrm{Var}(H_n) = O(1)$.

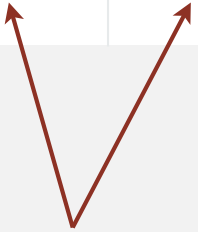
But... Worst-case height is $N - 1$.

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

ST implementations: summary

WCS

implementation	guarantee		average case		operations on keys
	search	insert	search hit	insert	
sequential search (unordered list)	N	N	$\frac{1}{2} N$	N	`equals()`
binary search (ordered array)	$\lg N$	N	$\lg N$	$\frac{1}{2} N$	`compareTo()`
BST	N	N	$1.39 \lg N$	$1.39 \lg N$	`compareTo()`



Why not shuffle to ensure a (probabilistic) guarantee of 4.311 ln N?

http://algs4.cs.princeton.edu

3.2 Binary Search Trees

- *BSTs*
- *ordered operations*
- *deletion*

ST implementations: summary

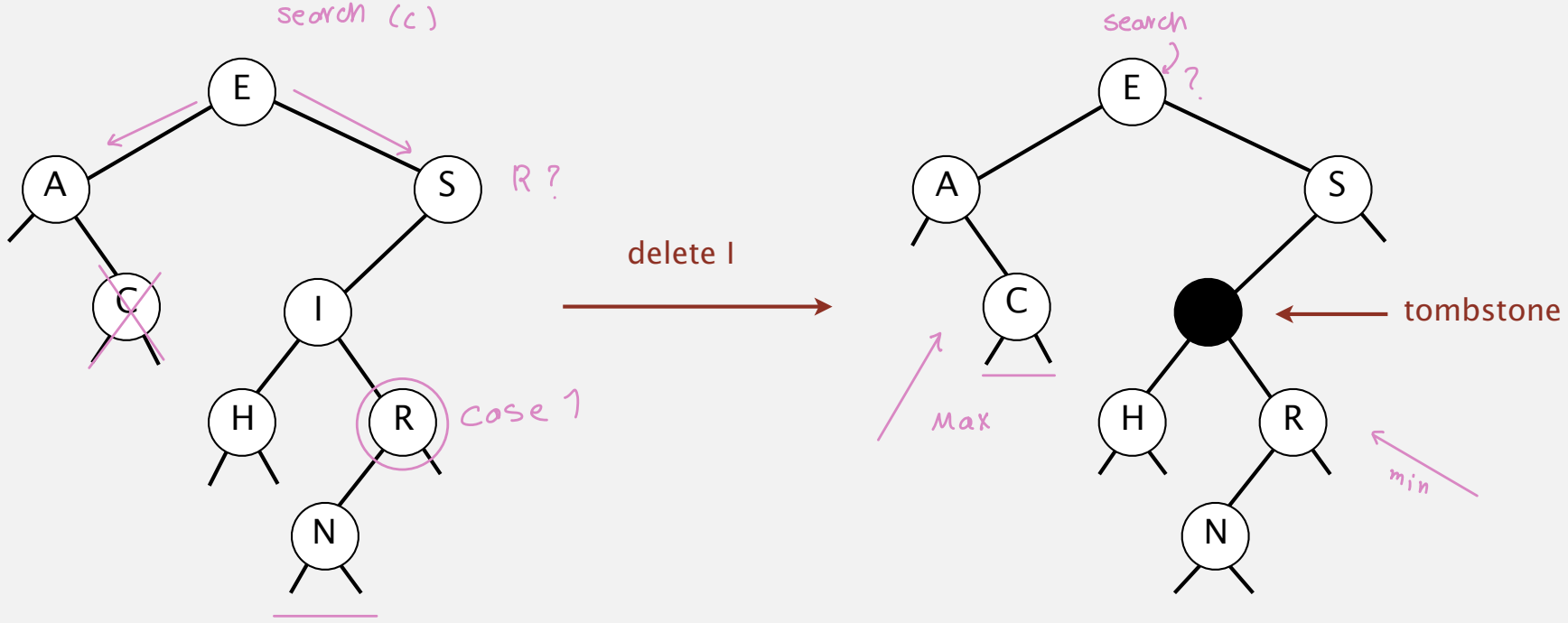
implementation	guarantee			average case			ordered ops?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	$\frac{1}{2} N$	N	$\frac{1}{2} N$		`equals()`
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✔	`compareTo()`
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	? ? ?	✔	`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.

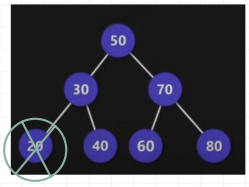
Deletion

- Deletion in BST has 3 cases:
  - Node to be deleted is a leaf node (has no child)
  - Node to be deleted has only one child
  - Node to be deleted has two children

Deletion

- Case 1:
  - Delete node with value '20'

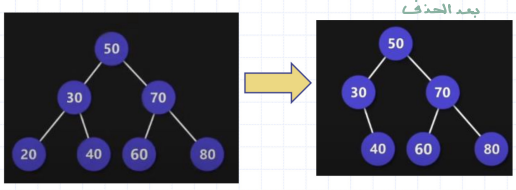
اول شي نبحث من الـ 20 .
ثاني شي امد اولادها.
اذا ما عندها جلغي الـ node .



Deletion

- Case 1:
  - Delete node with value '20'

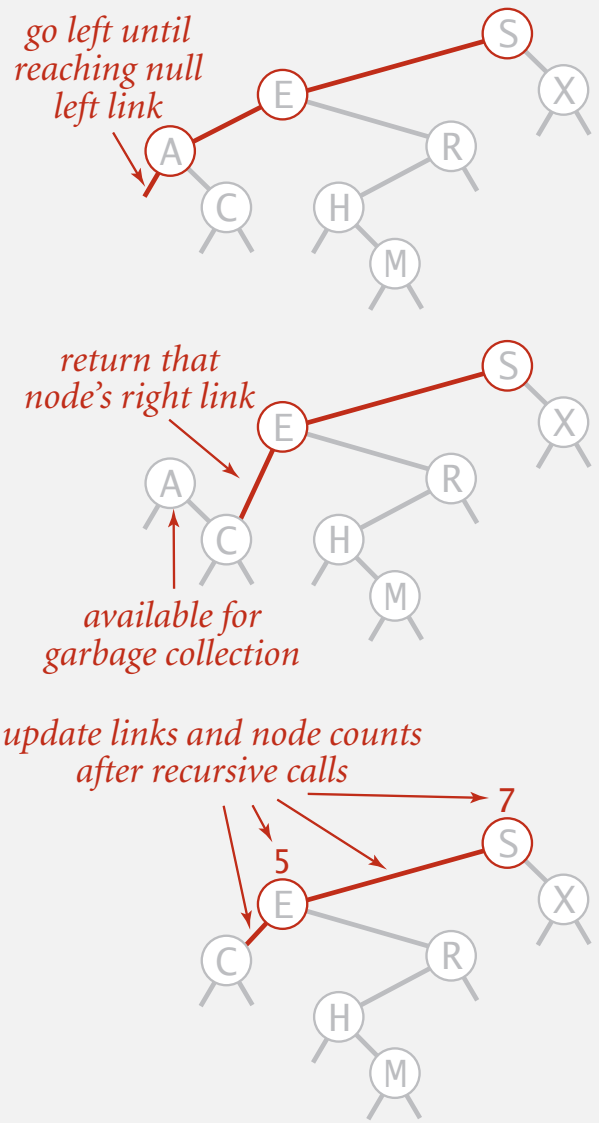
بعد الحذف



Deleting the minimum

To delete the minimum key:

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



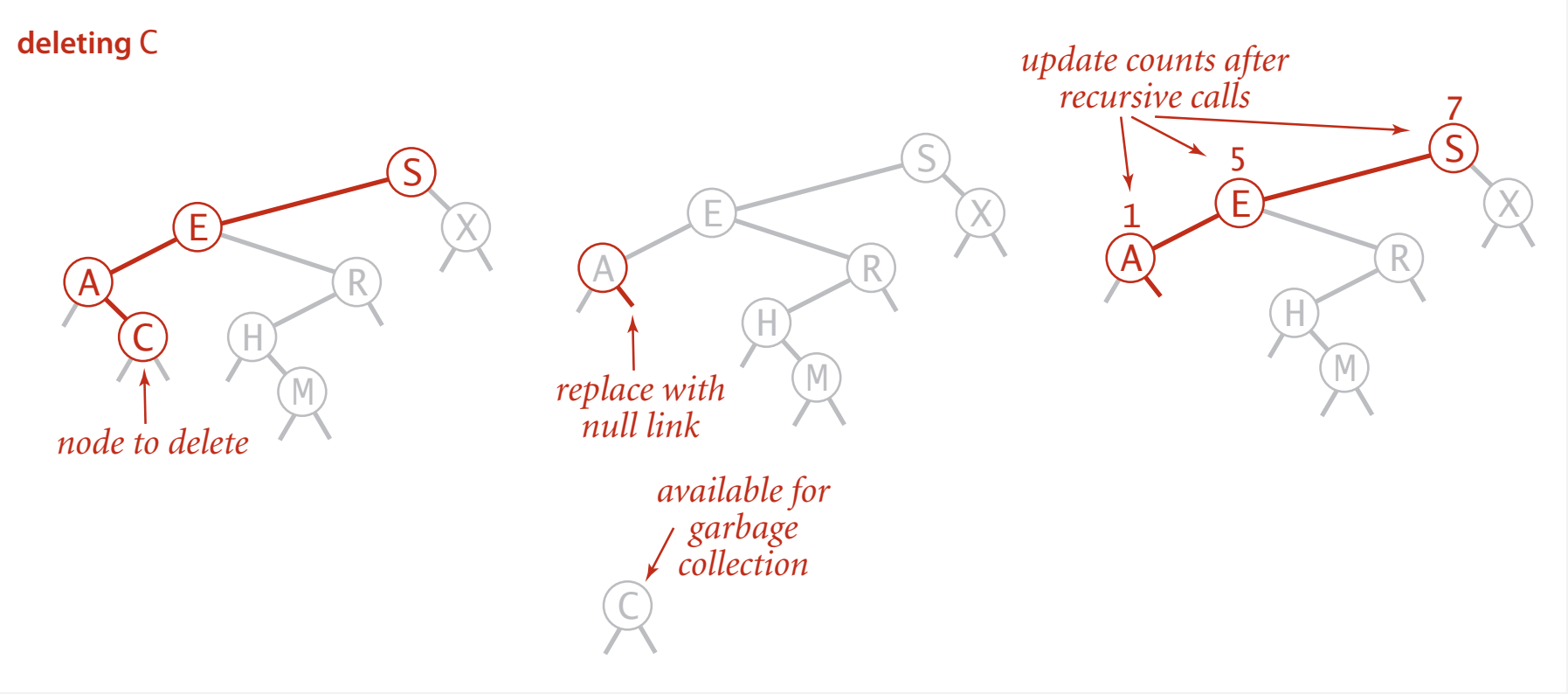

```
public void deleteMin()
{  root = deleteMin(root);  }

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

Hibbard deletion

Case 2 :

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

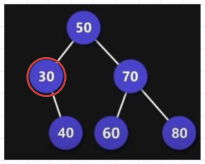
Case 0. [0 children] Delete `t` by setting parent link to null.



Deletion

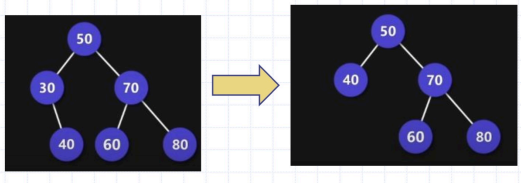
- Case 2
  - Delete node with value '30'

اول شيء ابحث عن الـ 30
مثلاً شي ٤ اعمد اولادة.
عنده ابن و احد.
سواء يسار او يمين، بشبك الأب بالإبن دائماً.



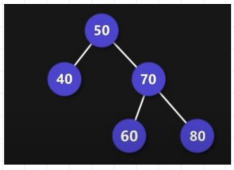
Deletion

- Case 2
  - Delete node with value '30'



Deletion

- Case 3
  - Delete node with value '50'

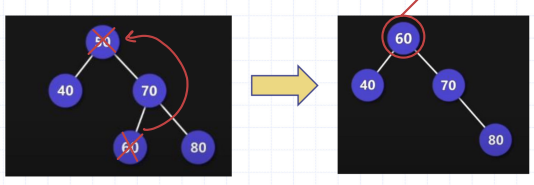


اول شي نبحث من الـ 50.
ثاني شي كم عدد الاولاد
عنده TwoChild.
مابيصير الخاء الـ node . موقوفة.
لازم يجي واحد يصير مكانه لان اذا فكيت الـ node راح
تضيع الـ tree.
راح نجيب واحد مناسب يجي مكانة.
one left, right . left ـ يا باخذ اكبر node من جهة
one right, left. right ـ او باخذ اصغر node من جهة

Deletion

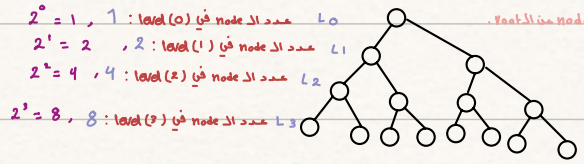
- Case 3
  - Delete node with value '50'

هو اخذ الـ 60 الي هو اصغر node
من جهة right .



كم عدد الـ node في level معين ؟

level : راح نحسب الـ level : اس هو بعد الـ node من الـ root .



- ايش العلاقة بين الـ level و عدد الـ nodes :

2^{level} = numbers of node
عدد النودات في level معين .

BT : O(n)

- ايش هو اقل عدد في النودات : node واحد

- ايش هو الـ hight : ٤ ← اخر level + ١ = hight

- ايش هو الـ maximum : راح اجمعهم

$2^4 - 1 = 15$ ← 1+2+4+8=15

---

hight : بعدد او ارتفاعه من الارض :

deepth : من فوق لتحت .

balance : فرق العدل من الجهة اليمين ومن الجهة اليسار , لازم يكون (٠ ، ١- ، ١) .

left-right
or
right-left

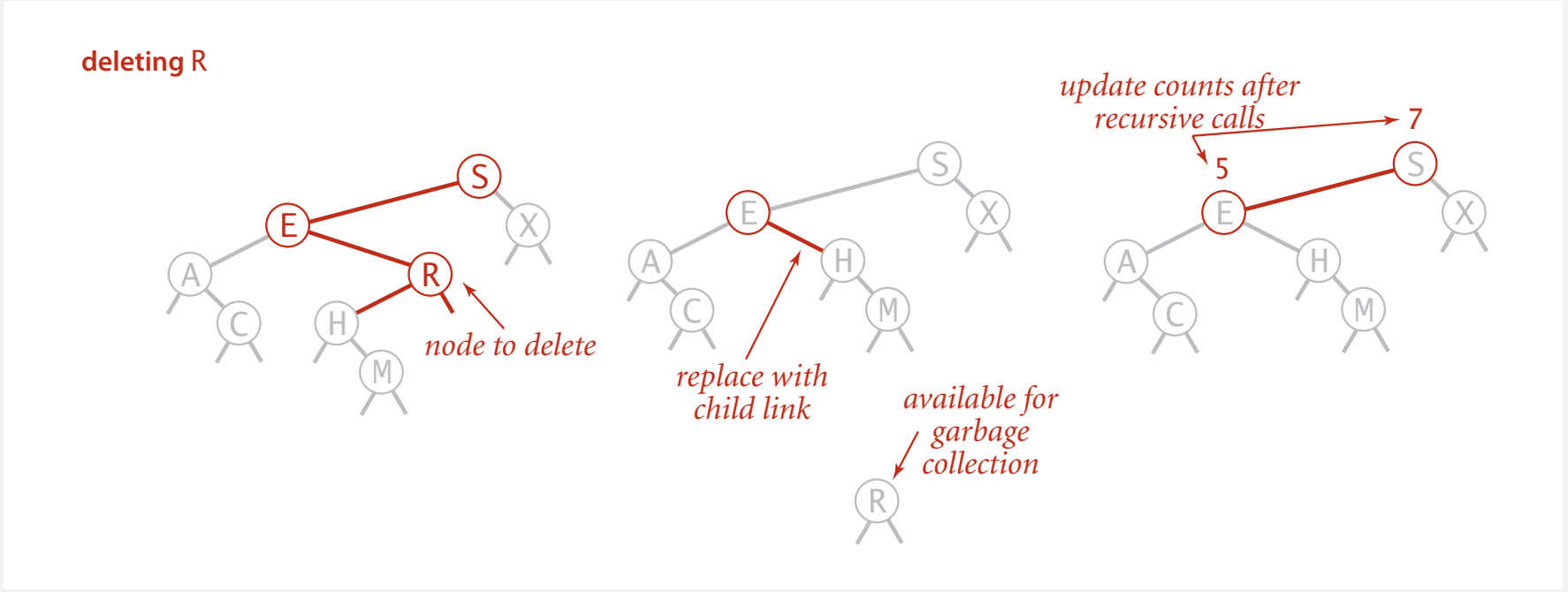
h(F) = 0
h(d) = 0
h(c) = 0
h(E) = 1
h(b) = 2
h(a) = 3

d tree = 3 ← h
عدد الـ edges بعد النود من الروت .

Hibbard deletion

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

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

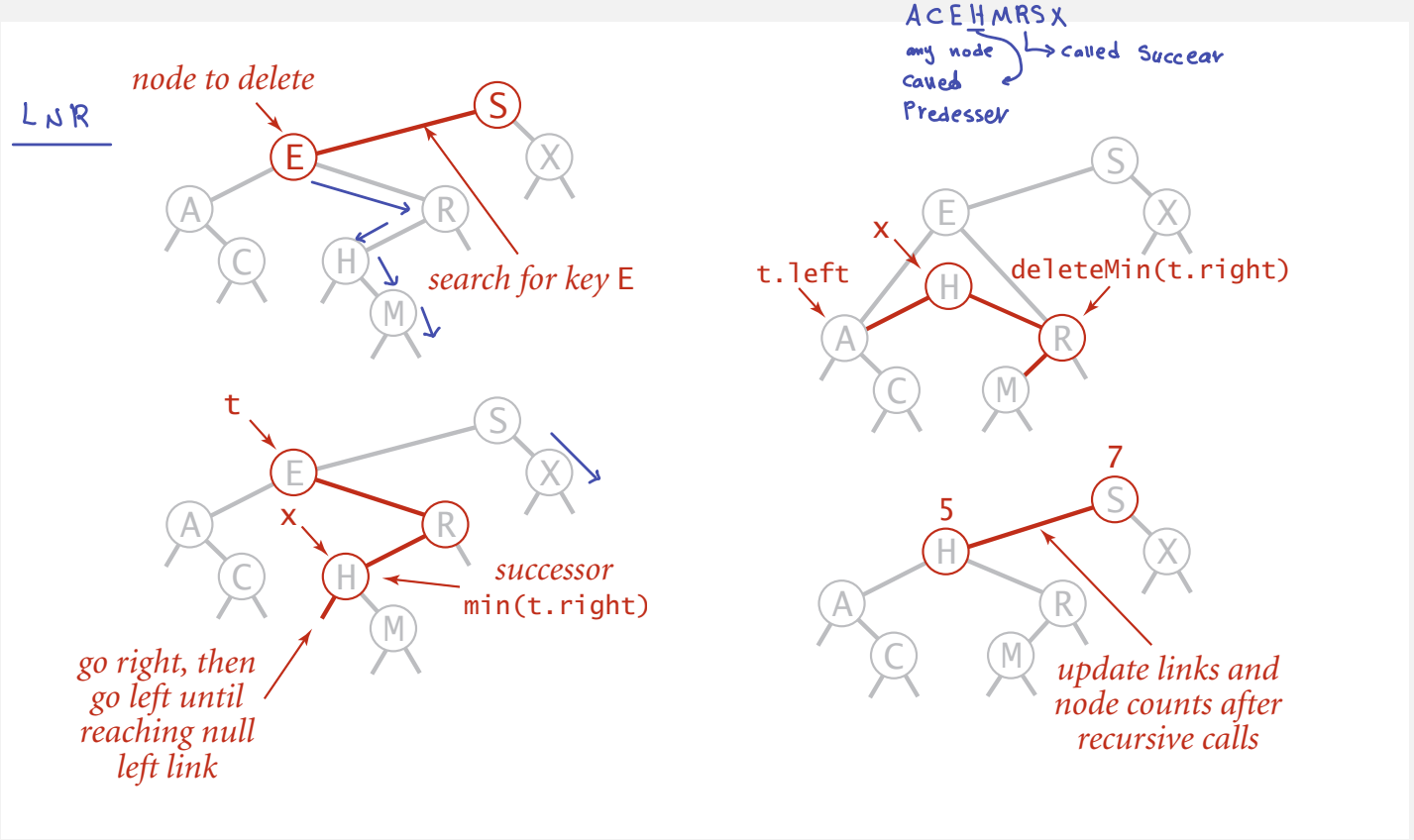


Hibbard deletion

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

Case 2. [2 children]

- Find successor `x` of `t`. ← x has no left child
- Delete the minimum in `t`'s right subtree. ← but don't garbage collect x
- Put `x` in `t`'s spot. ← still a BST



Hibbard deletion: Java implementation

```
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

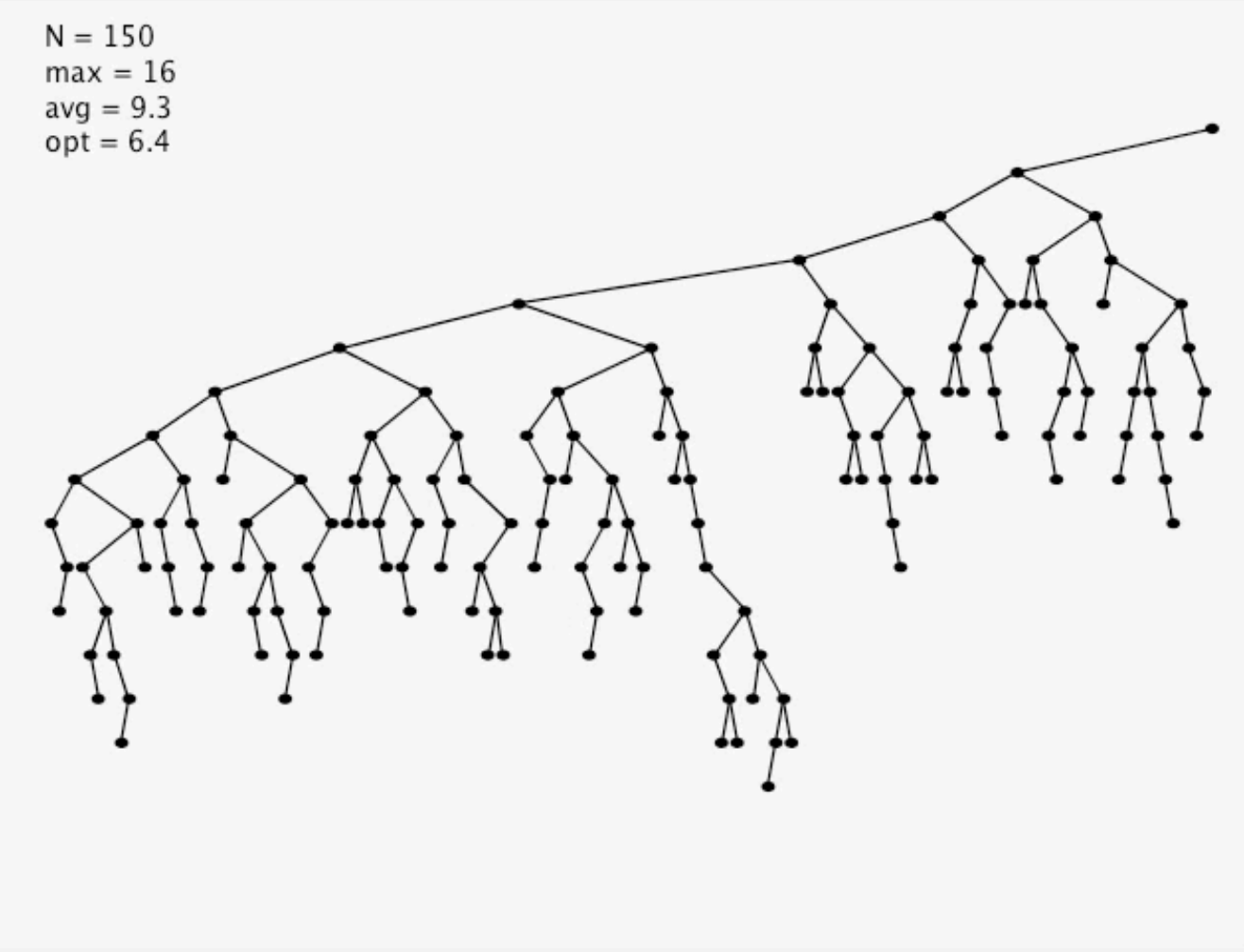
no left child

replace with successor

update subtree counts

Hibbard deletion: analysis

Unsatisfactory solution. Not symmetric.



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

Longstanding open problem. Simple and efficient delete for BSTs.

ST implementations: summary

w.c.s (بتكون على جهة وحدة) O(n)

b.c.s (مرتبه) O(log₂n)

implementation	guarantee			average case			ordered ops?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	$\frac{1}{2} N$	N	$\frac{1}{2} N$		`equals()`
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✔	`compareTo()`
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✔	`compareTo()`

search best case : log n

(Preorder, inorder, Postorder) Travers : n

other operations also become √N if deletions allowed

Next lecture. Guarantee logarithmic performance for all operations.

Method	Running Time
size, isEmpty	$O(1)$
root, parent, left, right, sibling, children, numChildren	$O(1)$
isInternal, isExternal, isRoot	$O(1)$
addRoot, addLeft, addRight, set, attach, remove	$O(1)$
depth(p)	$O(d_p + 1)$
height	$O(n)$

Table 8.1: Running times for the methods of an n-node binary tree implemented with a linked structure. The space usage is $O(n)$.