ID1020: Binary Search Trees

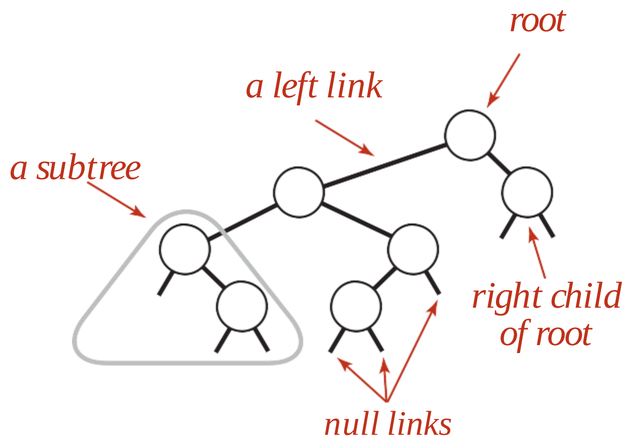
Dr. Per Brand
pbrand@kth.se

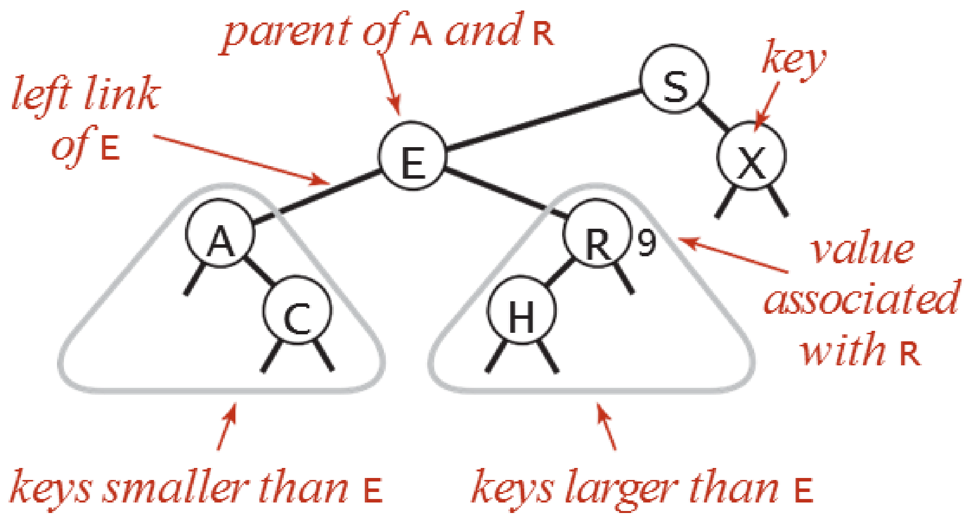
kap 3.2

Slides adapted from Algoritms 4th Edition, Sedgewick.

BSTs

- Binary Search Trees or BSTs
  - Is binary tree
  - In symmetric order
    - Each node has a key
    - Larger than all keys in its left subtree
    - Smaller than all keys in it right subtree





Node structure in Java

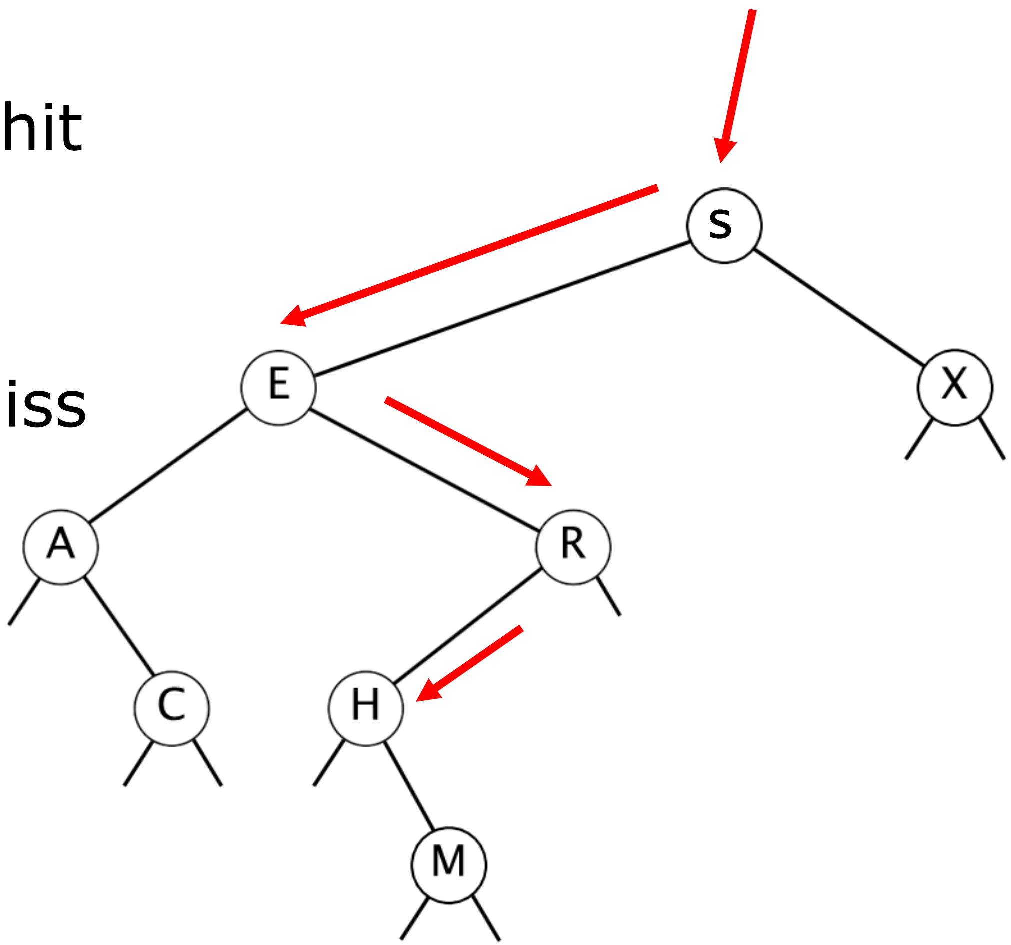
- Node has four fields
  - Key, value, reference to left subtree, reference to right subtree
  - A BST is then a reference to root node

```
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;
    }
```

Searching a BST

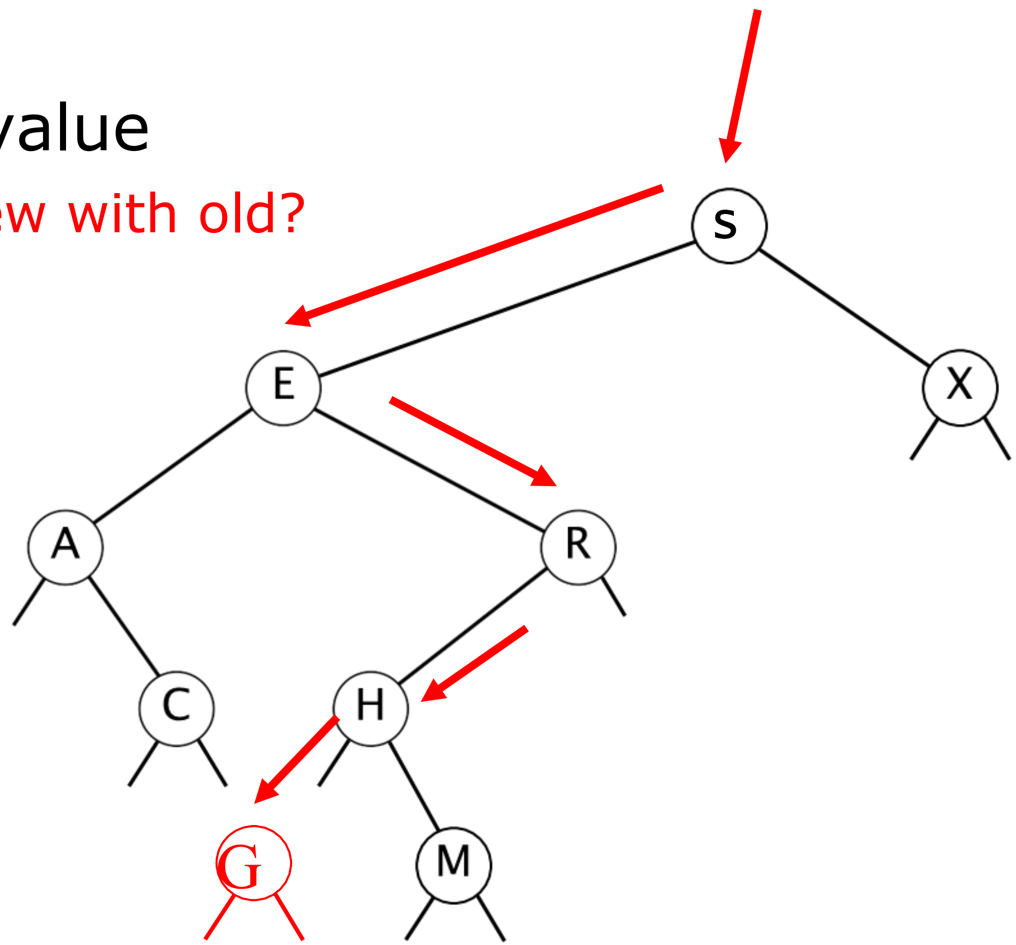
- Method
  - If equal we have a hit
  - If less go left
  - If greater go right
  - If null we have a miss

- Example
  - Search for H



Inserting into a BST

- Method
  - If equal overwrite value
    - How do we merge new with old?
  - If less go left
  - If greater go right
  - If null insert



- Example
  - Insert G

Java Implementation (1)

```java
public class BST<Key extends Comparable<Key>, Value>
{                                       5
    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

Implementation of get

- Return value associated with key

- Complexity
  - 1+depth(node)

```
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;
}
```

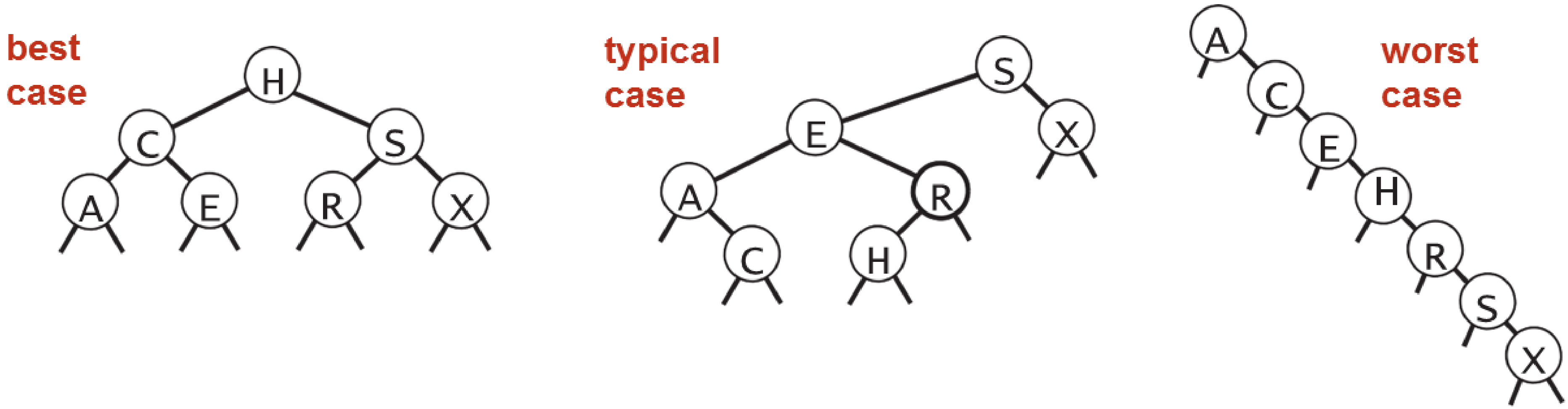
Implementation of put

```
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;
}
```

Often this assignment is a noop
When ?
Why necessary?

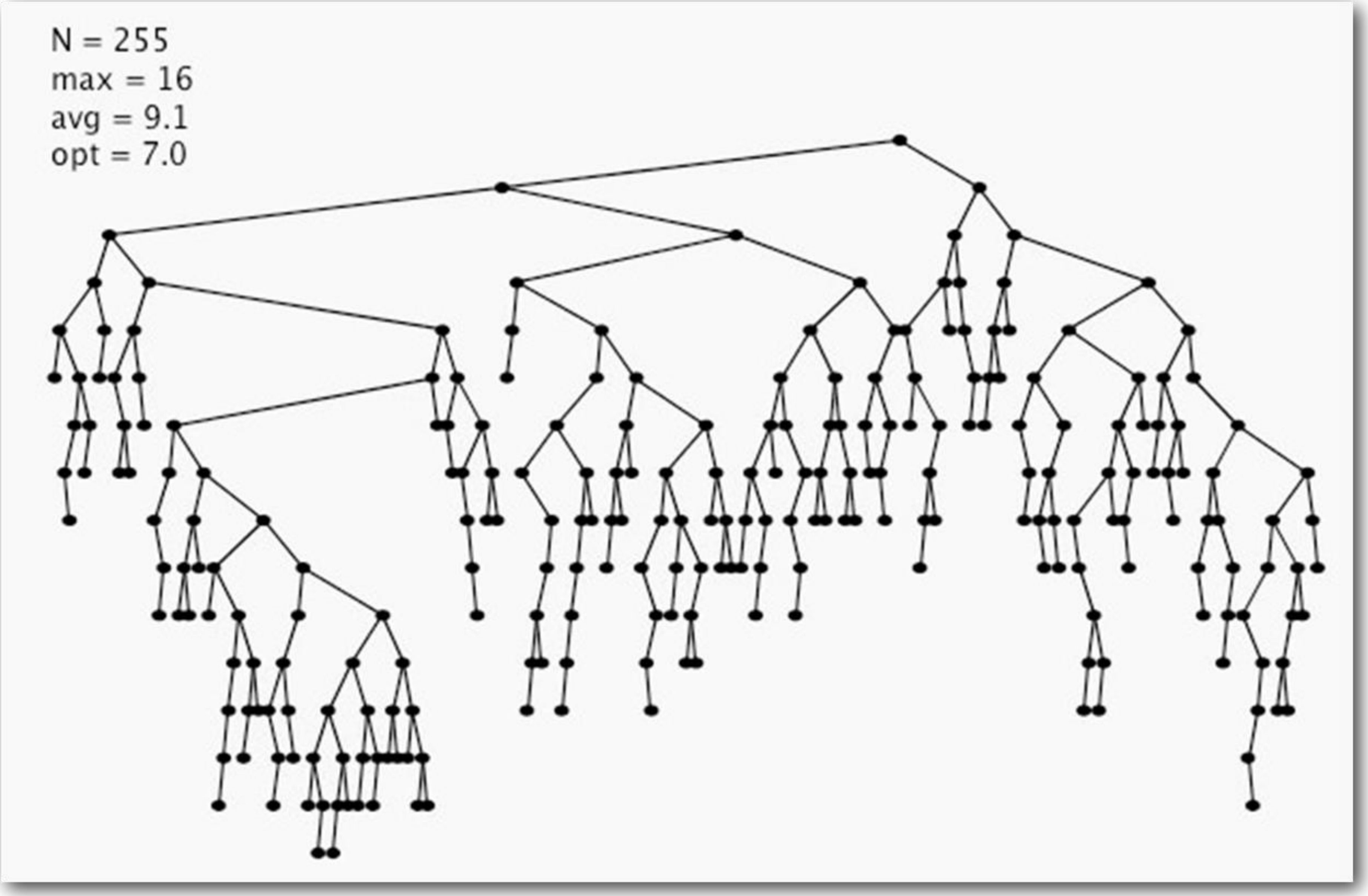
Tree shape

- Many different possible shapes for same set of keys
- Shape depends on order of insertion



- Remember complexity of get/put is 1+depth of node

Visualization of random ordered tree



Complexity

- N distinct keys inserted in random order
  - Expected number of compares for get/put ~2 ln N
  - Or ~1.39 lg N
  - Proof analagous to qsort

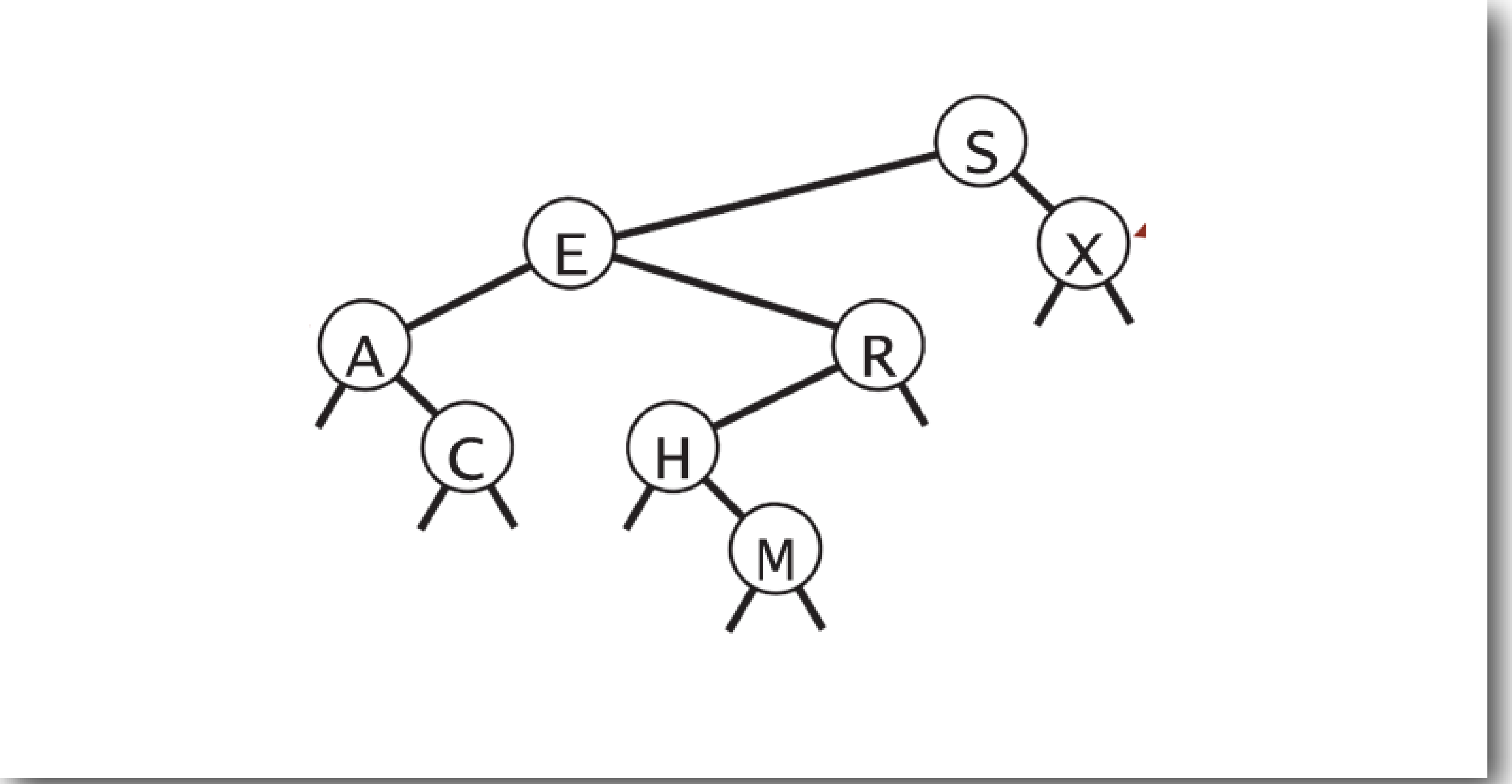
- Observe
  - If perfectly balanced would expect lg N
  - Worst case is ~N
  - Expected case close to best case

Complexity 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()`

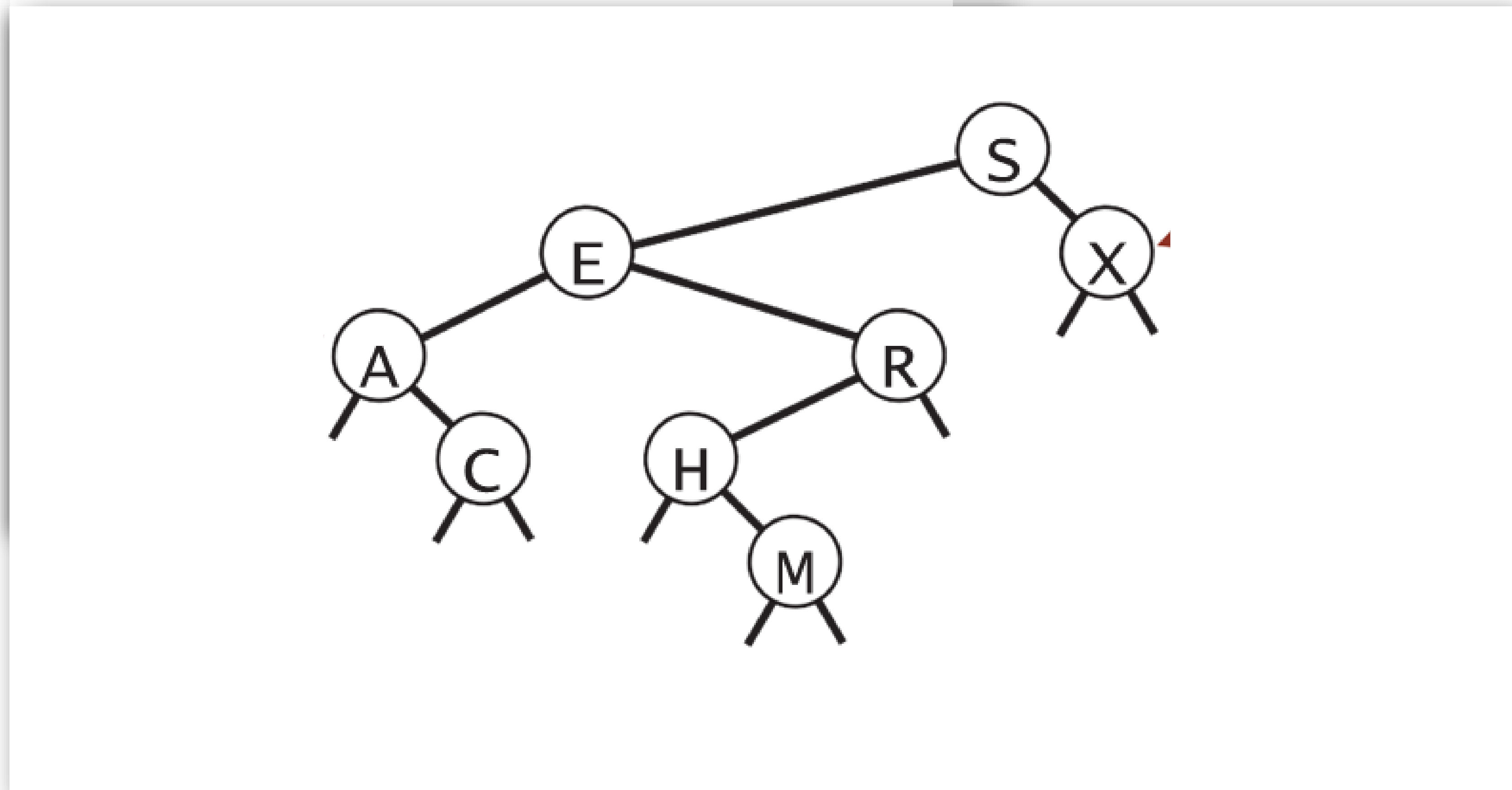
More operations on BSTs

- Find minimum/maximum
  - Value associated with minimum key
  - Value associated with maximum key
  - How ?



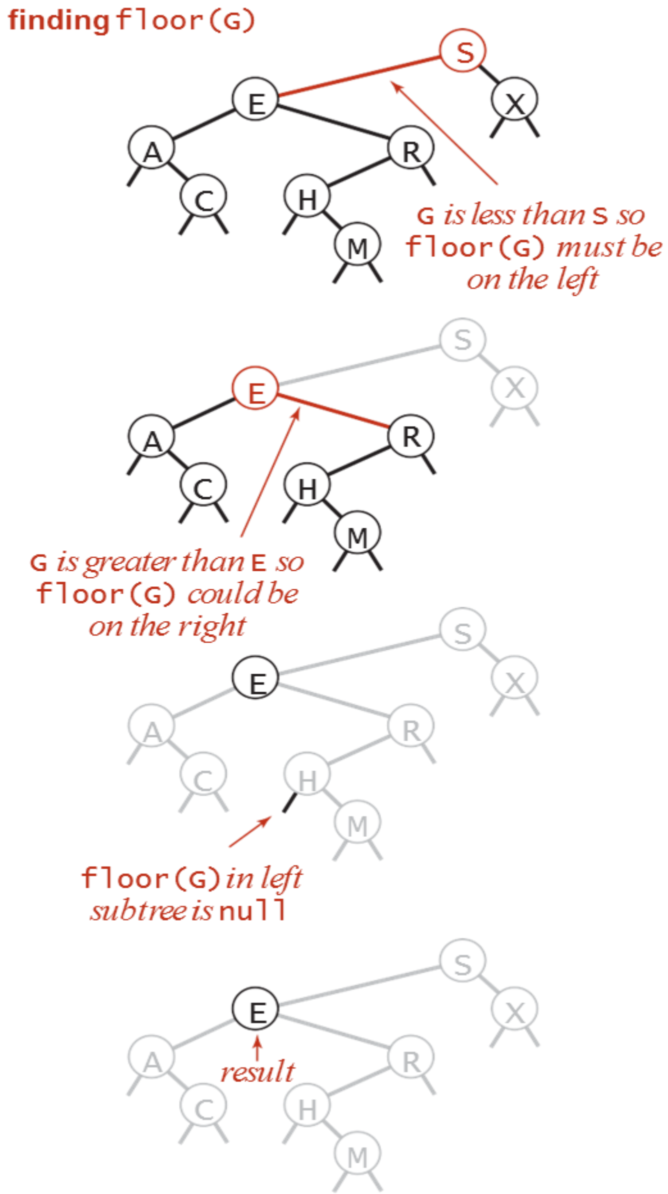
More operations on BSTs (2)

- Find floor/ceiling of k
  - Floor: Largest key ≤ k
  - Ceiling: Smallest key ≥ k
  - How ?



Computing floor

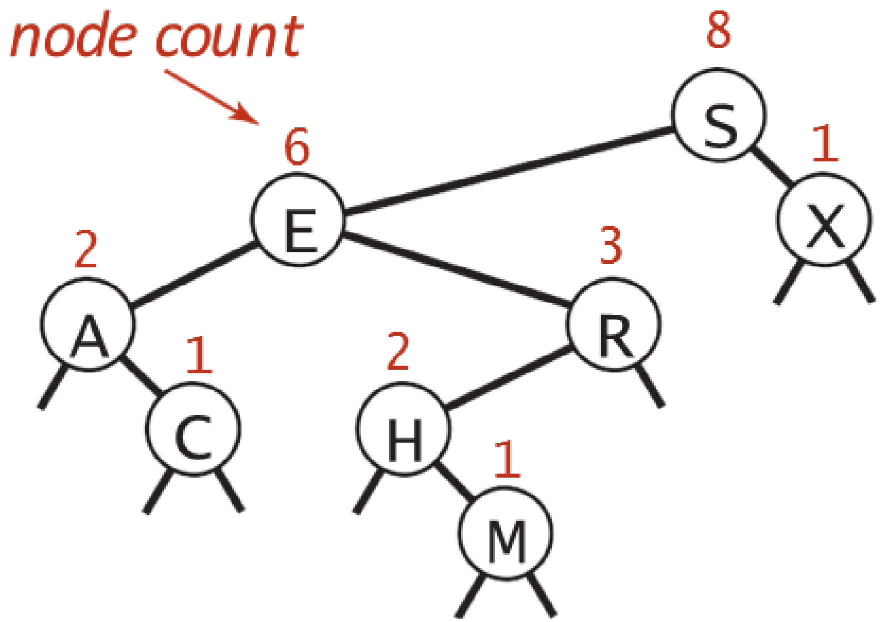
```
public Key floor(Key key)
{
   Node x = floor(root, key);
   if (x == null) return null;
   return x.key;
}
private Node floor(Node x, Key key)
{
   if (x == null) return null;
   int cmp = key.compareTo(x.key);

   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;

}
```



Subtree counts

- Add new field to each node, *size*
  - Number of nodes in subtree rooted at node

- At root implements size()



Rank

- Rank – number of keys < given key
- Three cases
  - two of which lead to recursive calls in left or right subtree

```
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);
}
```

Two more operations

- Select
  - Return node with given rank

- Inorder traversal
  - Use a queue during traversal of the tree

Summary

- Summary of symbol table implementations covered so far

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()`

- How do we delete from a BST ?

Deleting the minimum

- As deleting is difficult
  - Begin by slightly easier problems
  - Deleting with zero children (leaf) trivial
  - Deleting the minimum node (Analgous to deleting any node with only one child

```
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;
}
```

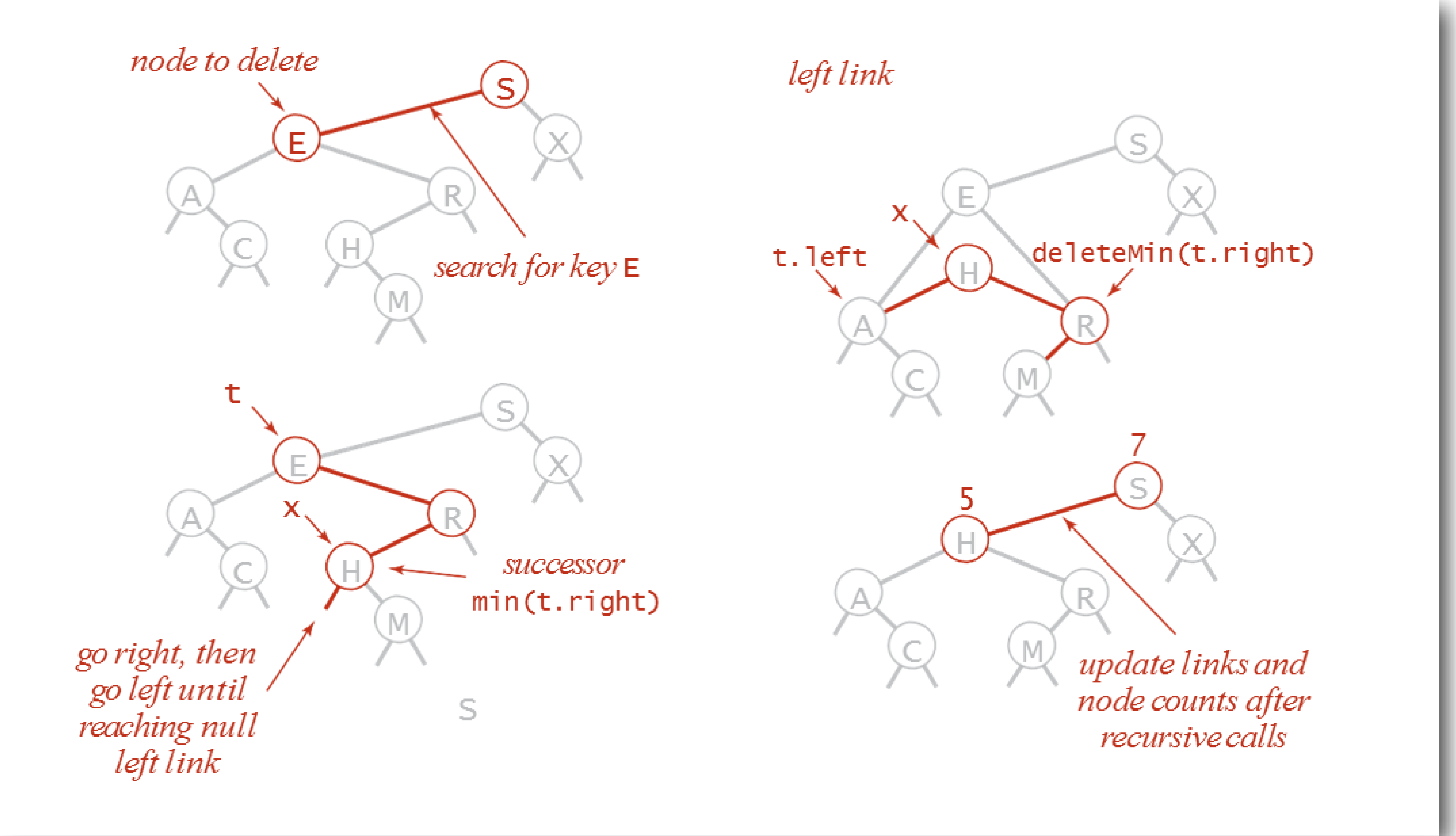
- But what about nodes with 2 children ?

Hibbard deletion

- Three cases
  - 0 children just delete the node
  - 1 child analgous to deleting min
  - 2 children use successor

- Successor of a node is smallest key in right subtree
  - We can replace a node to be deleted with its successor

2 children case illustrated



Hibbard deletion - 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

However

- Hibbard deletion turns out to be unsatisfactory
- Details are beyond the scope of this course
- However note that repetitive Hibbard deletions distorts the tree (making it even less balanced)

- Operations tend to have $\sqrt{N}$ complexity

- Next lecture – how to achieved logaritmic guarantee