Introduction to Algorithms

Meng-Tsung Tsai

10/17/2019

Announcements

Written Assignment 2 is due by Oct 31, 15:40. at https://e3.nctu.me

Programming Assignment 2 was extended, and is due by Nov 5, 23:59. at https://oj.nctu.me

Quiz 1 will be held in class on Oct 24.

Scope: slides 01 - 09, assignments, and their generalizations.

About Quiz 1

Asymptotic Bounds: was1-p1, was1-p2, was2-p1.

Basic DP: was2-p2, pas2-p1.

Reduction: was1-p5, was2-p6.

Don't forget the "I don't know" policy.

You may bring two cheating sheets in A4 size.

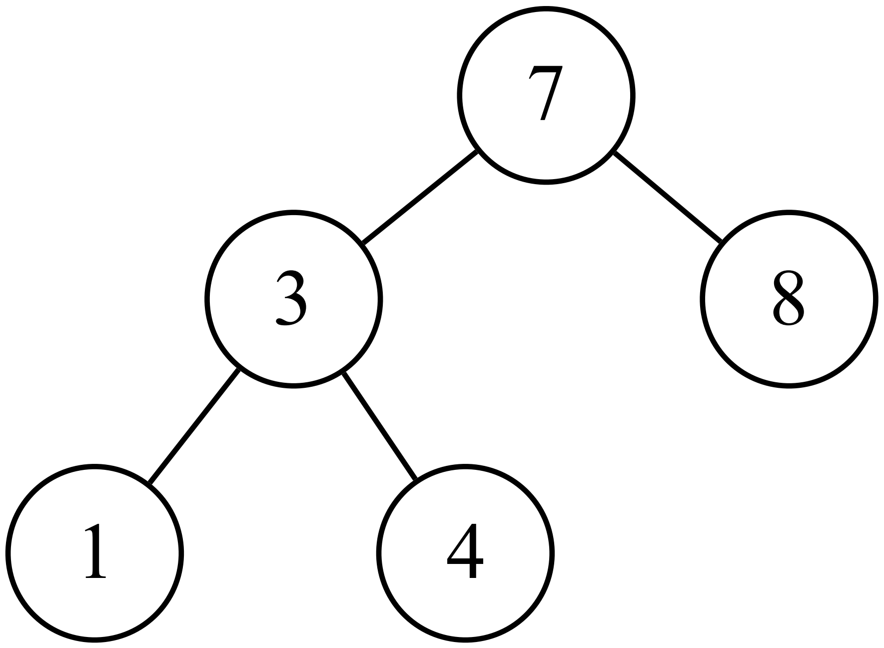
Binary Search Trees

What is a binary search tree?

A tree whose keys satisfy the *binary-search-tree property*:

For any node x, all nodes in the left subtree of x has key value at most x.key, and all nodes in the right subtree of x has key value at least x.key.

--- Example ---

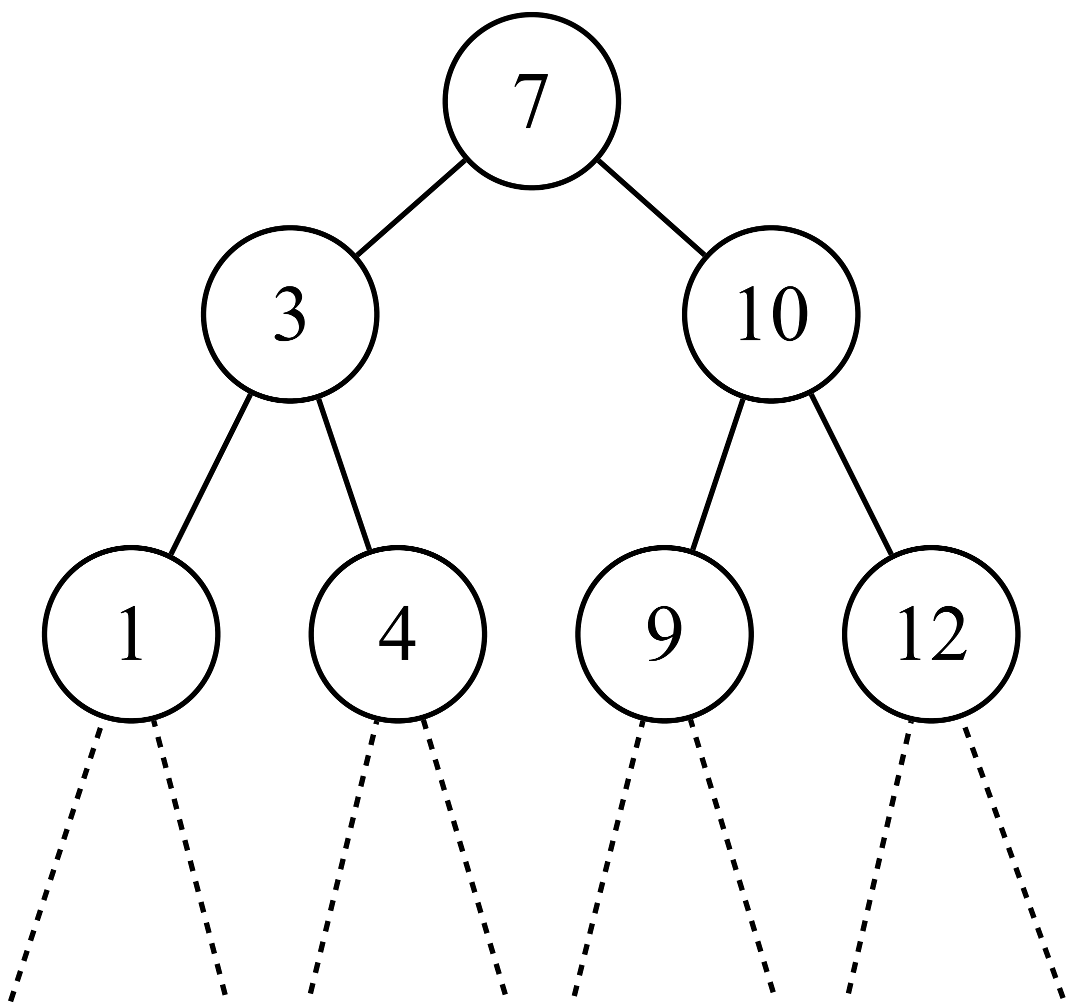


Every node x has 4 attributes:

x.key: the key value of x
x.left: a pointer to x's left subtree
x.right: a pointer to x's right subtree
x.p: a pointer to x's parent node

If a child or the parent is missing, the corresponding pointer is set as Null.

In-Order Tree Traversal



The in-order tree traversal visits the nodes in a tree in the order:

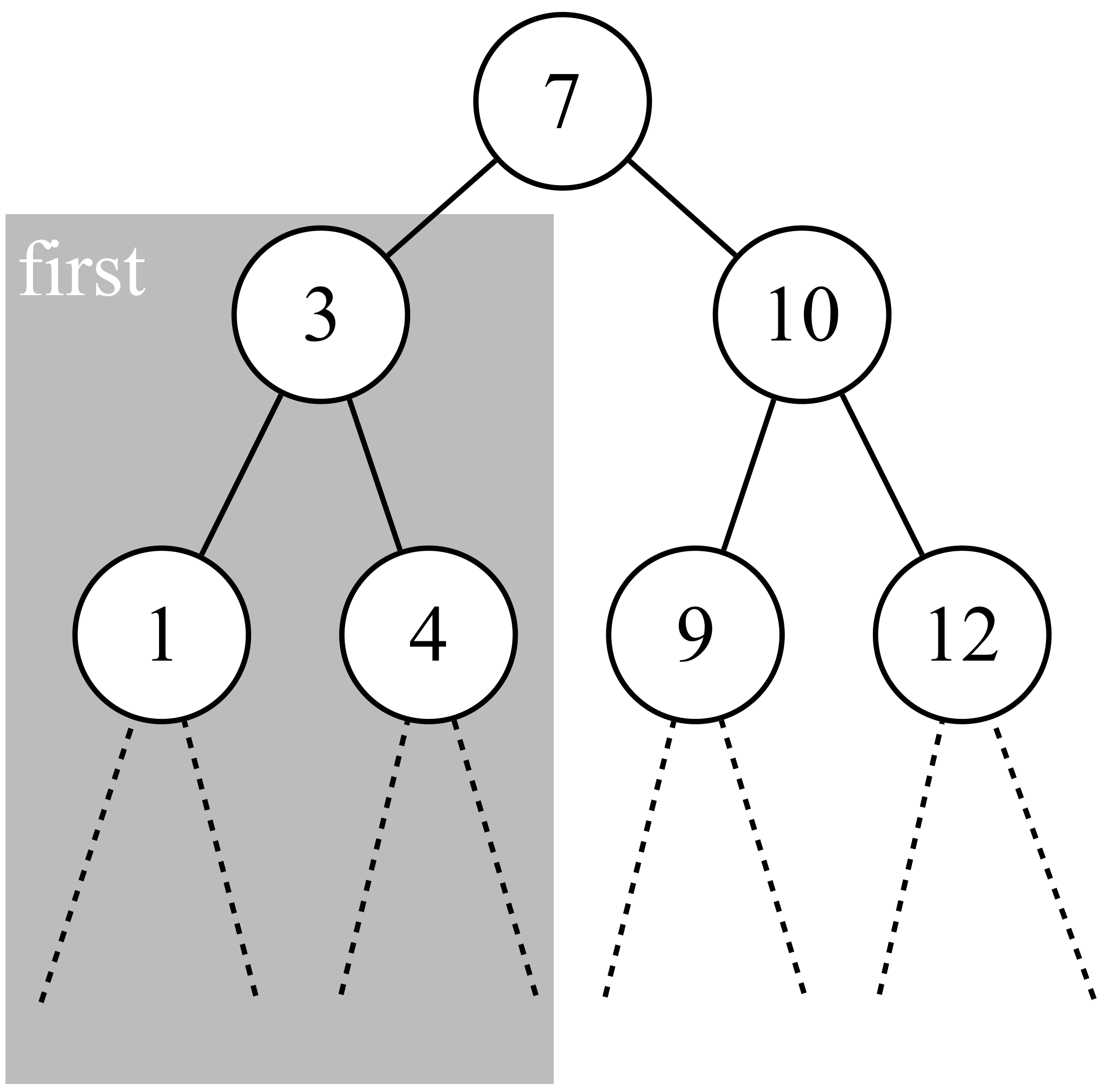
for any node x,

1. the nodes in x's left subtree are visited before x,

2. then x,

3. the nodes in x's right subtree are visited after x.

In-Order Tree Traversal



The in-order tree traversal visits the nodes in a tree in the order:

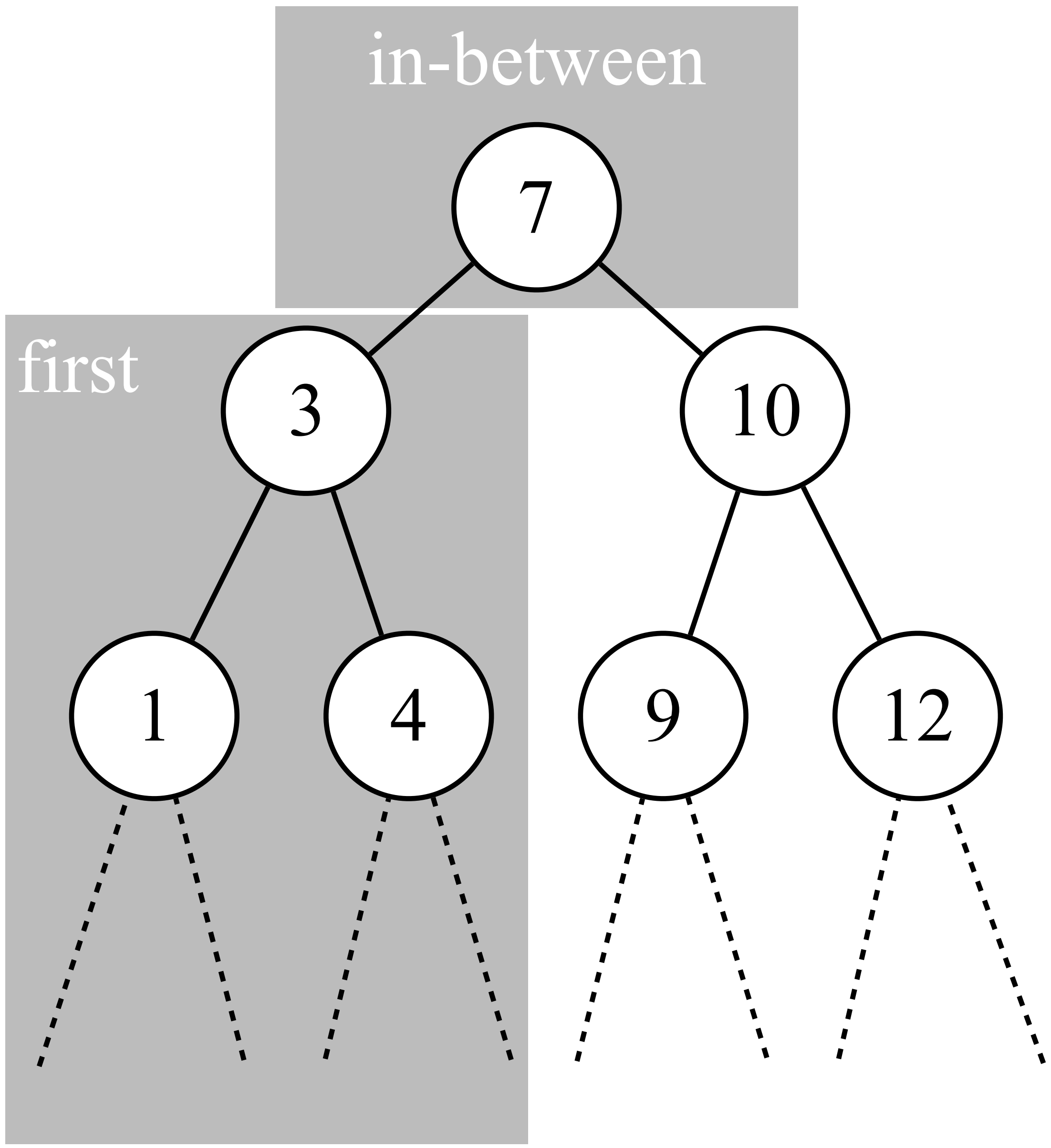
for any node x,

1. the nodes in x's left subtree are visited before x,

2. then x,

3. the nodes in x's right subtree are visited after x.

In-Order Tree Traversal



The in-order tree traversal visits the nodes in a tree in the order:

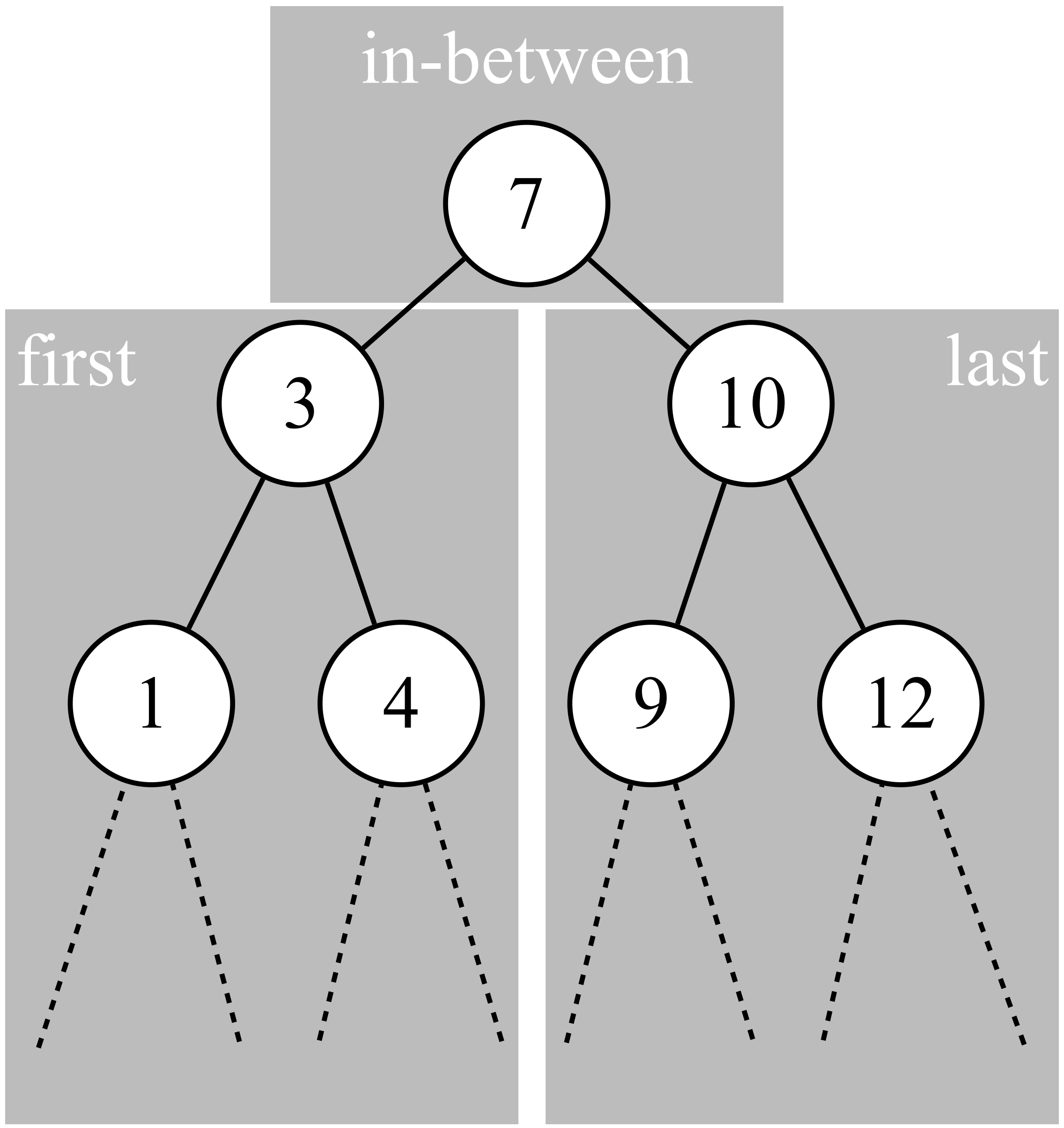
for any node x,

1. the nodes in x's left subtree are visited before x,

2. then x,

3. the nodes in x's right subtree are visited after x.

In-Order Tree Traversal



The in-order tree traversal visits the nodes in a tree in the order:

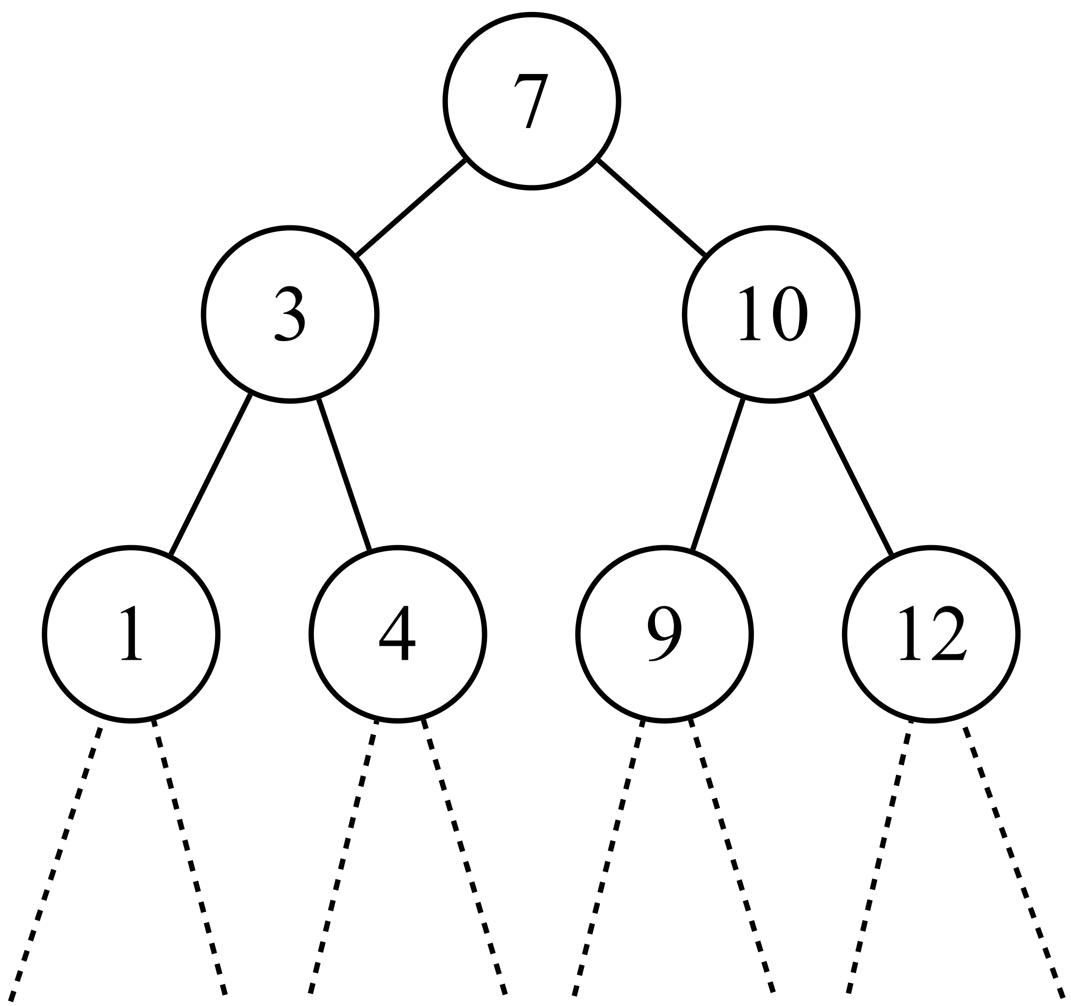
for any node x,

1. the nodes in x's left subtree are visited before x,

2. then x,

3. the nodes in x's right subtree are visited after x.

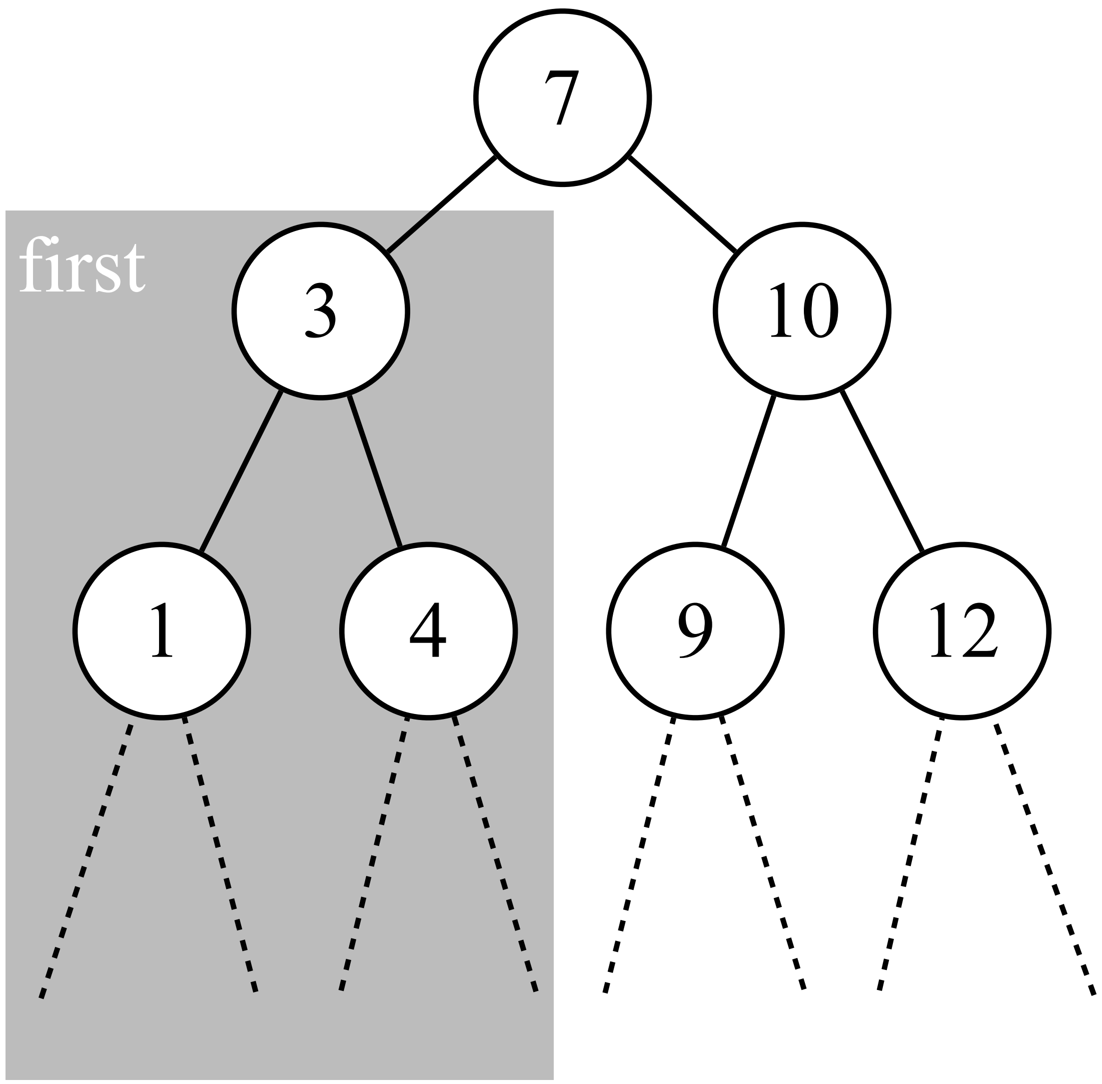
Output the keys on a BST in the sorted order




```
In-Order-Traversal(x){
    if(x ≠ NIL){
        In-Order-Traversal(x.left);
        print x.key;
        In-Order-Traversal(x.right);
    }
}
```

Q: Why does the above procedure output the keys in the sorted order?

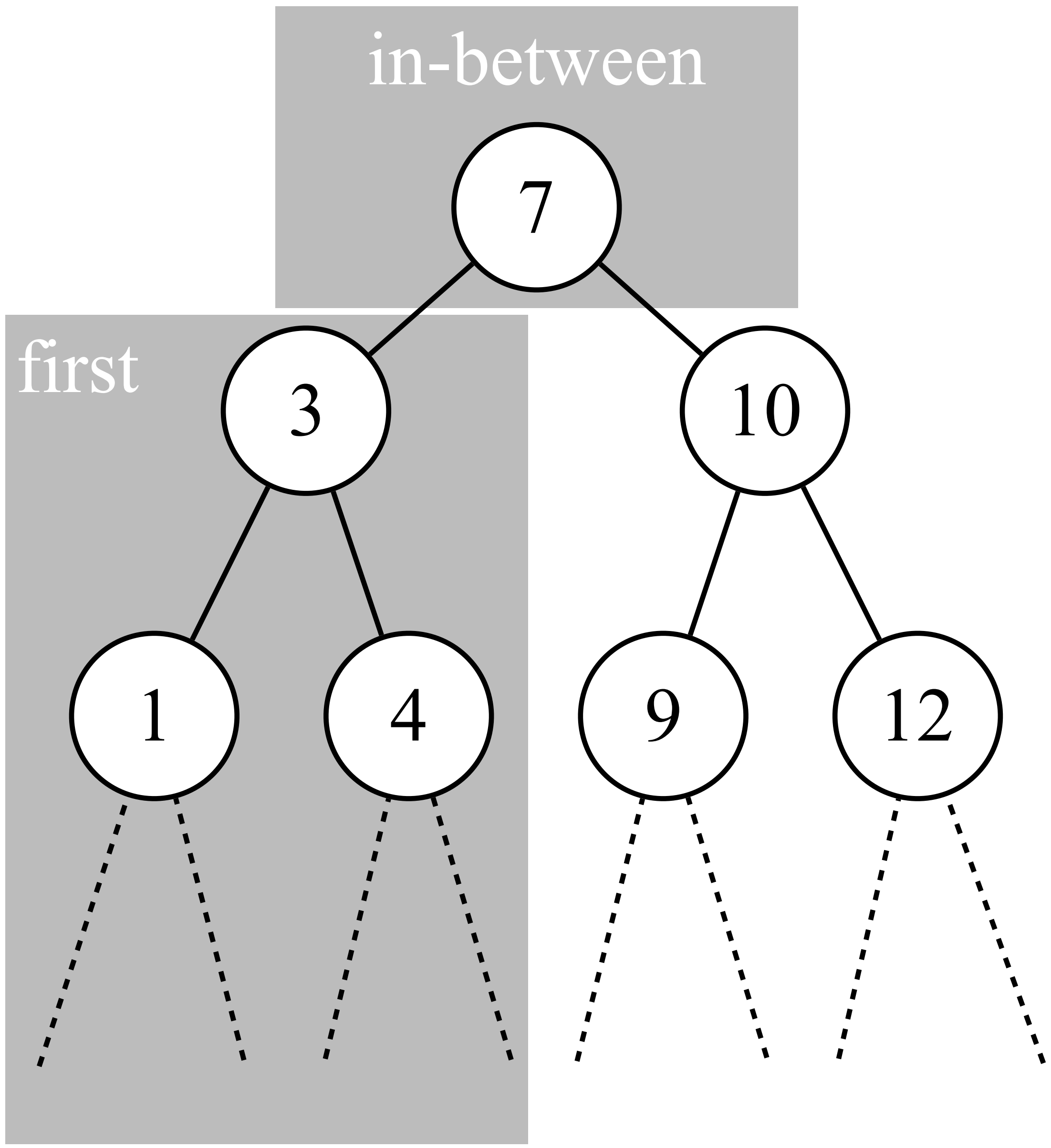
Output the keys on a BST in the sorted order




```
In-Order-Traversal(x){
    if(x ≠ NIL){
        In-Order-Traversal(x.left);
        print x.key;
        In-Order-Traversal(x.right);
    }
}
```

Q: Why does the above procedure output the keys in the sorted order?

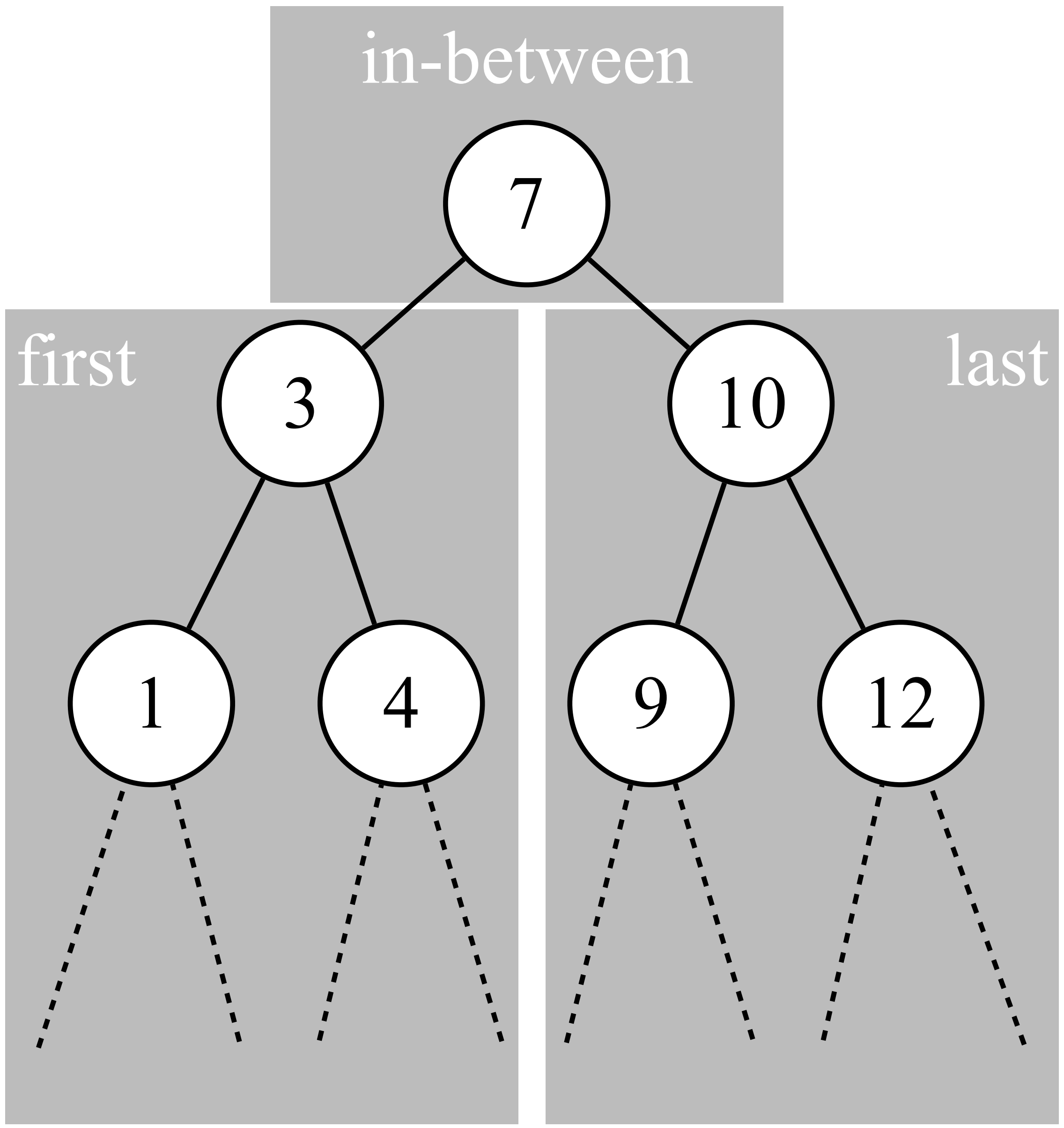
Output the keys on a BST in the sorted order




```
In-Order-Traversal(x){
    if(x ≠ NIL){
        In-Order-Traversal(x.left);
        print x.key;
        In-Order-Traversal(x.right);
    }
}
```

Q: Why does the above procedure output the keys in the sorted order?

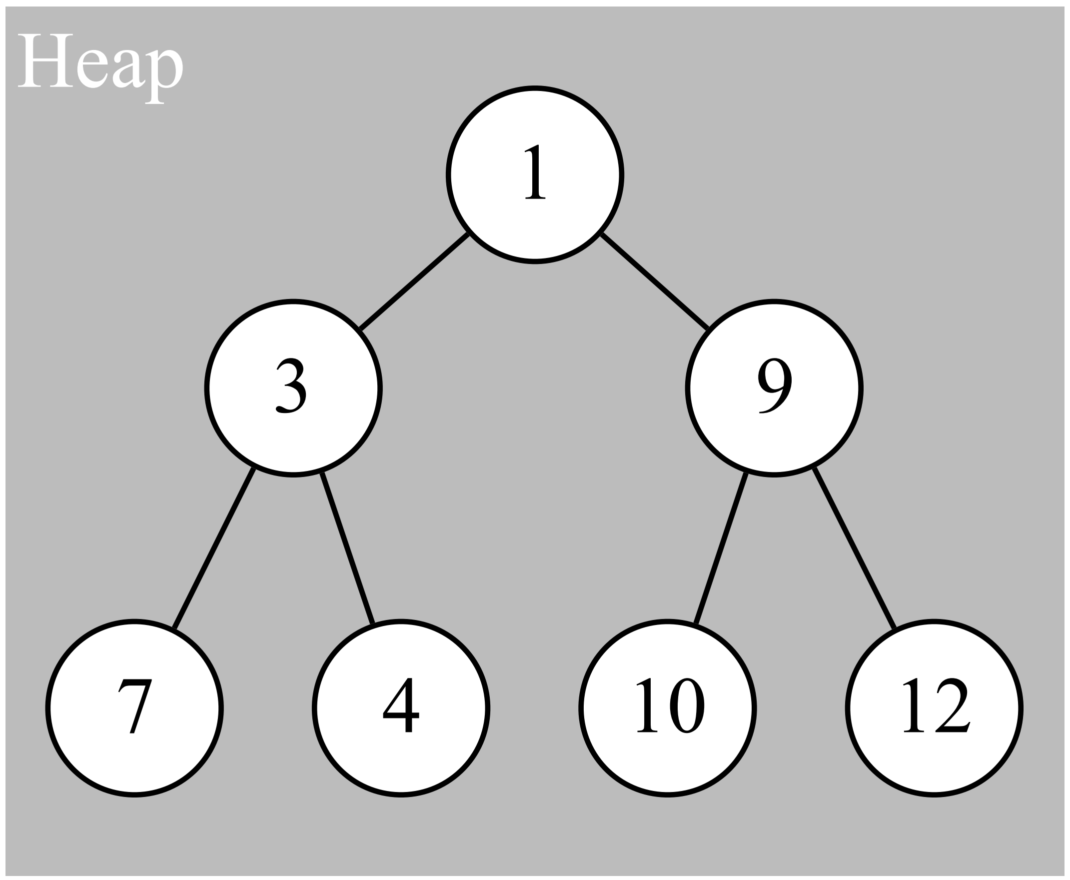
Output the keys on a BST in the sorted order

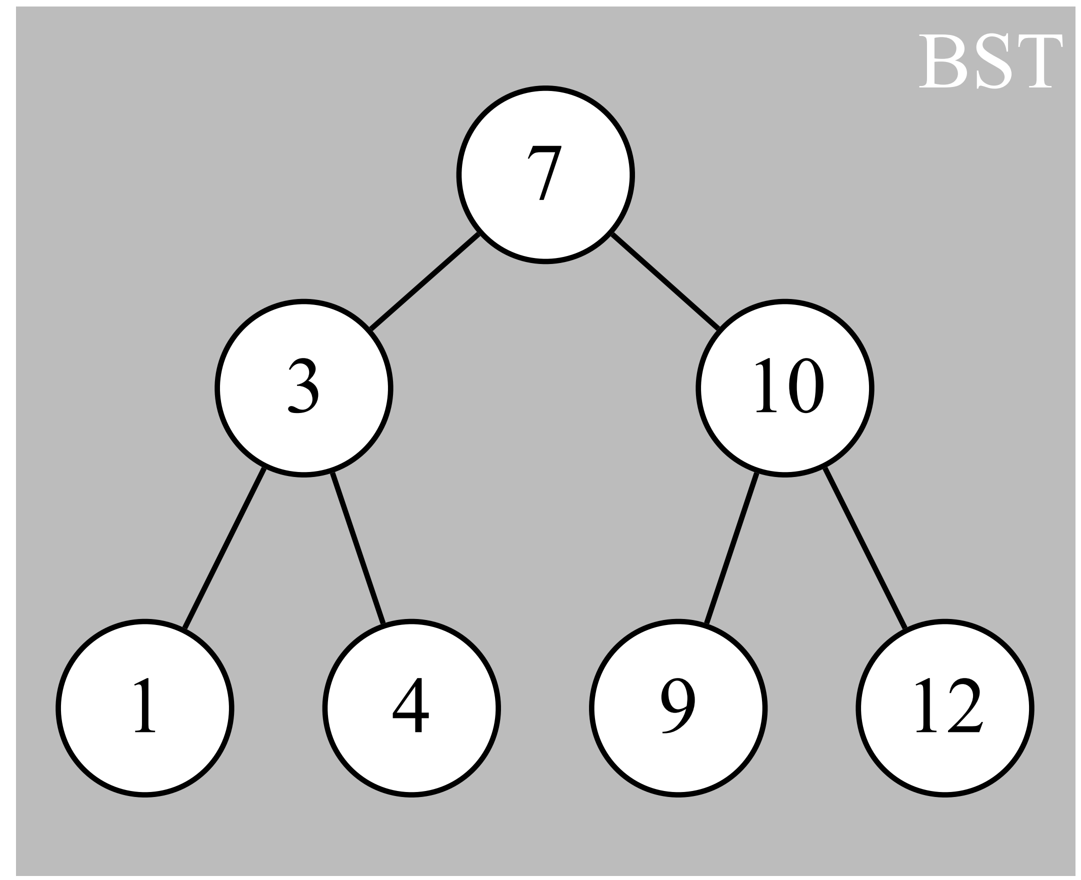



```
In-Order-Traversal(x){
    if(x ≠ NIL){
        In-Order-Traversal(x.left);
        print x.key;
        In-Order-Traversal(x.right);
    }
}
```

Q: Why does the above procedure output the keys in the sorted order?

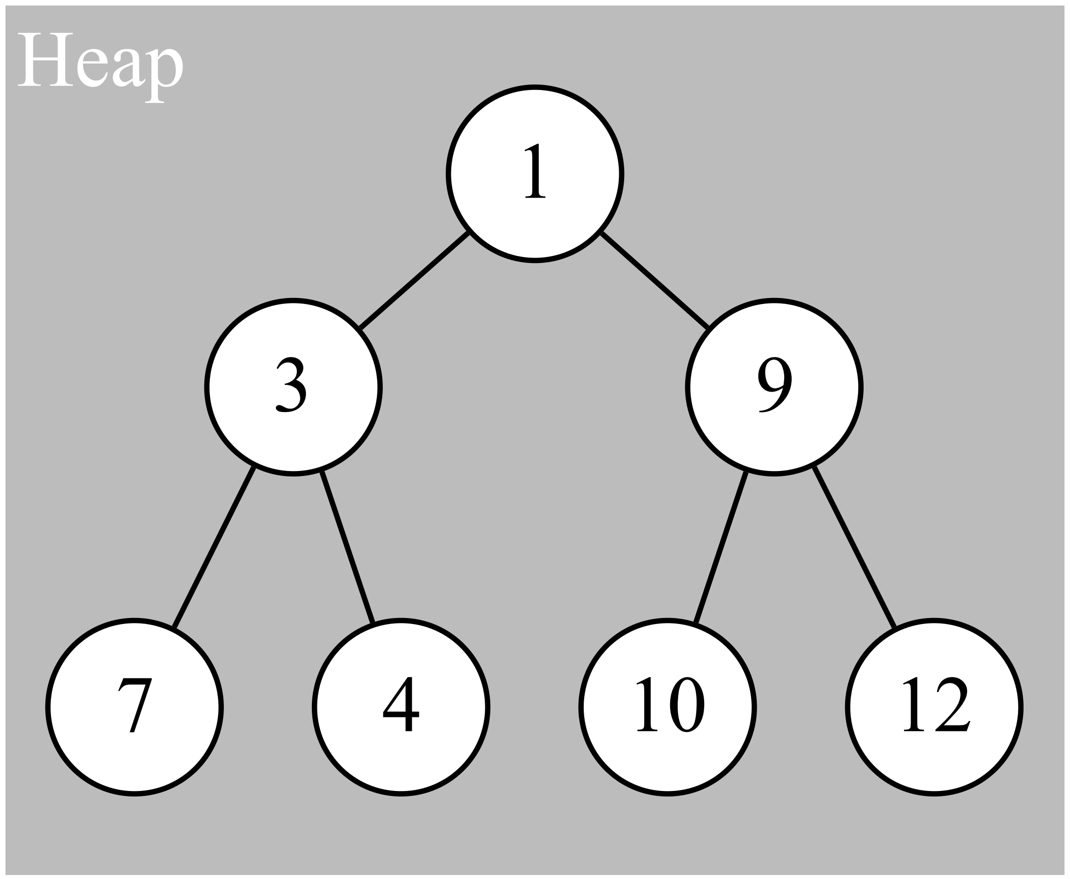
Differences between Heaps and BSTs

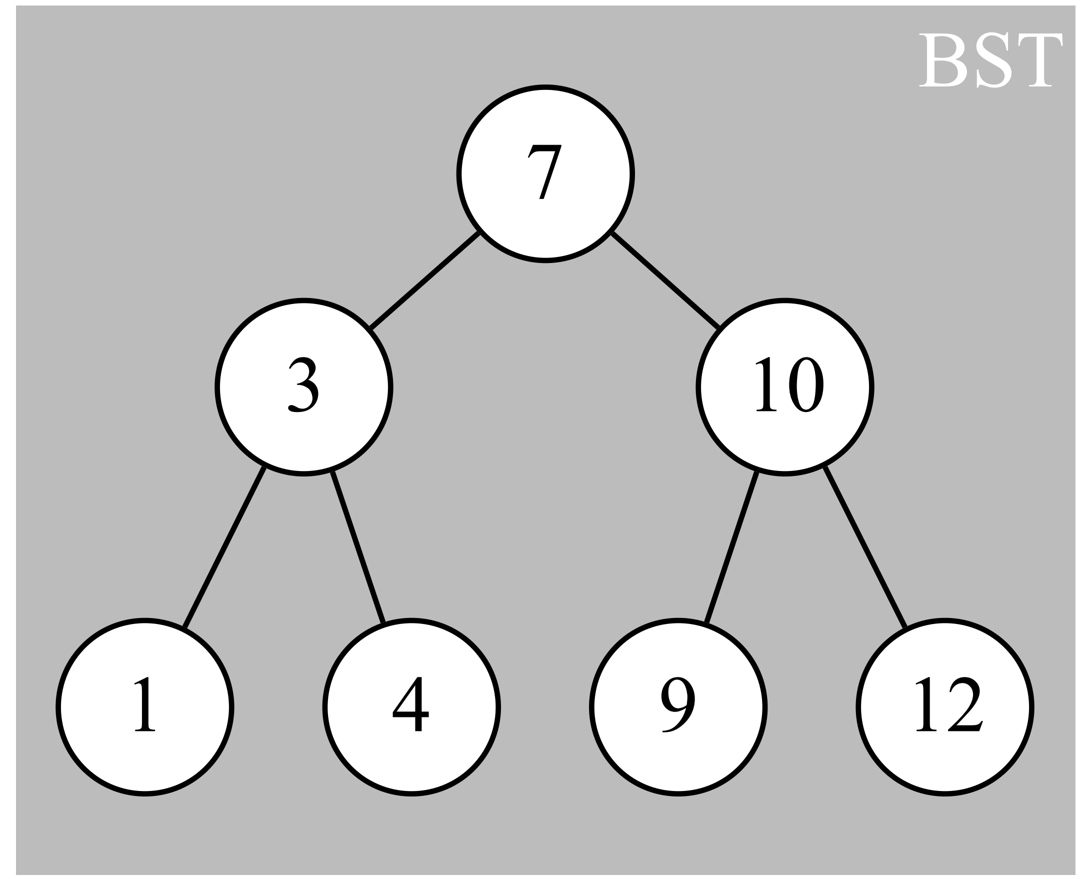




Q: Is it possible to output the keys in a heap in the sorted order by an O(n)-time traversal in the comparison-based model?

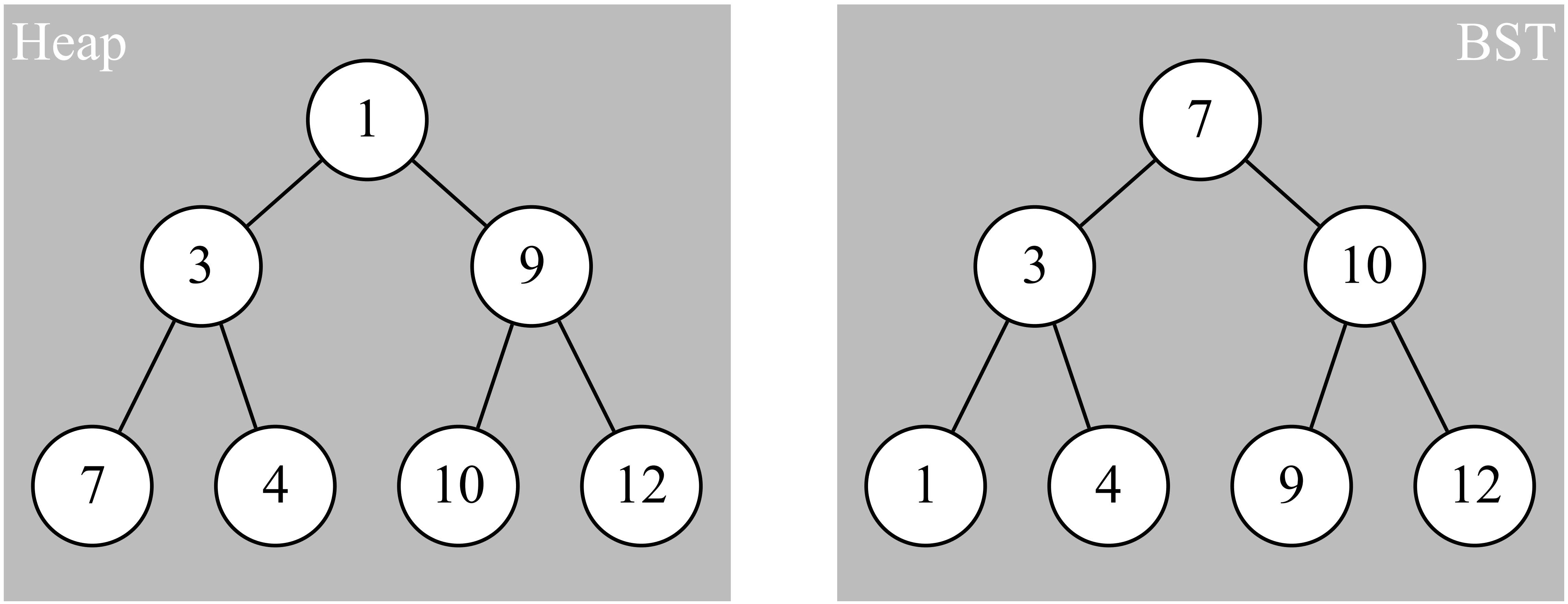
Differences between Heaps and BSTs





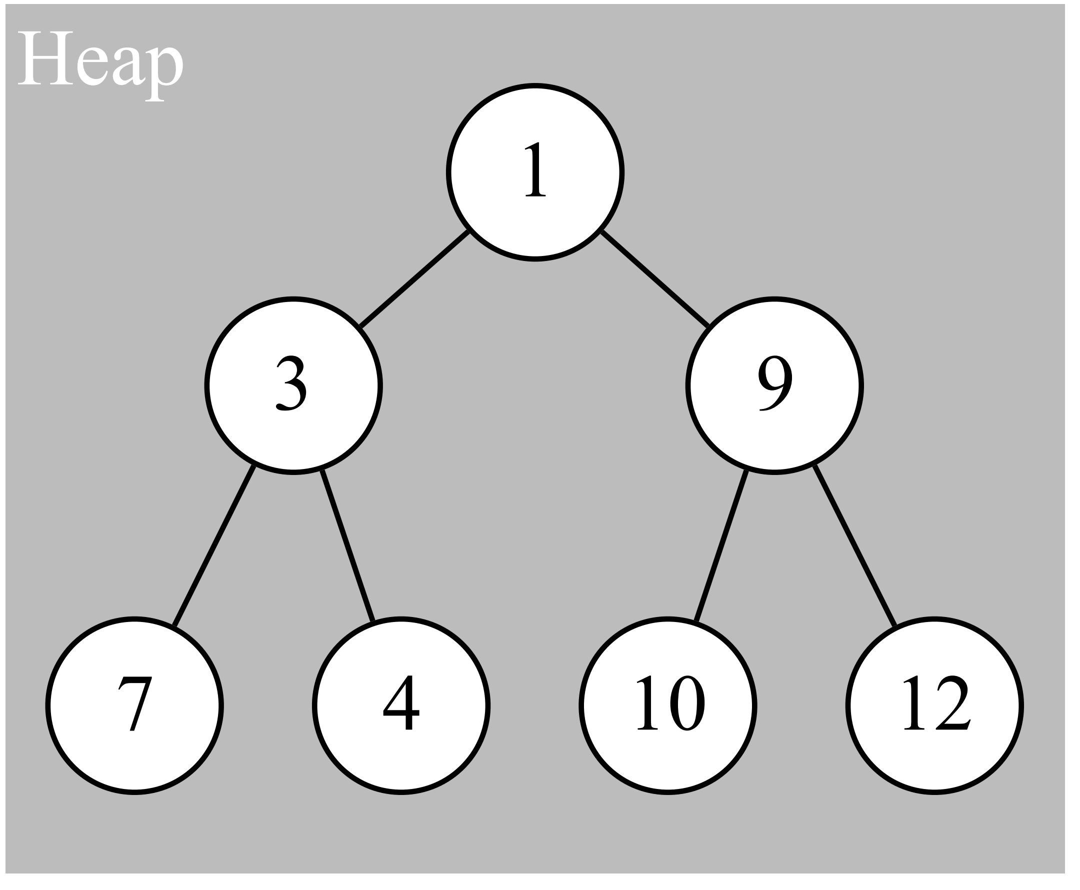
A: No. Otherwise build a heap (O(n) time) followed by the traversal (O(n) time) implements a sorting algorithm. →←

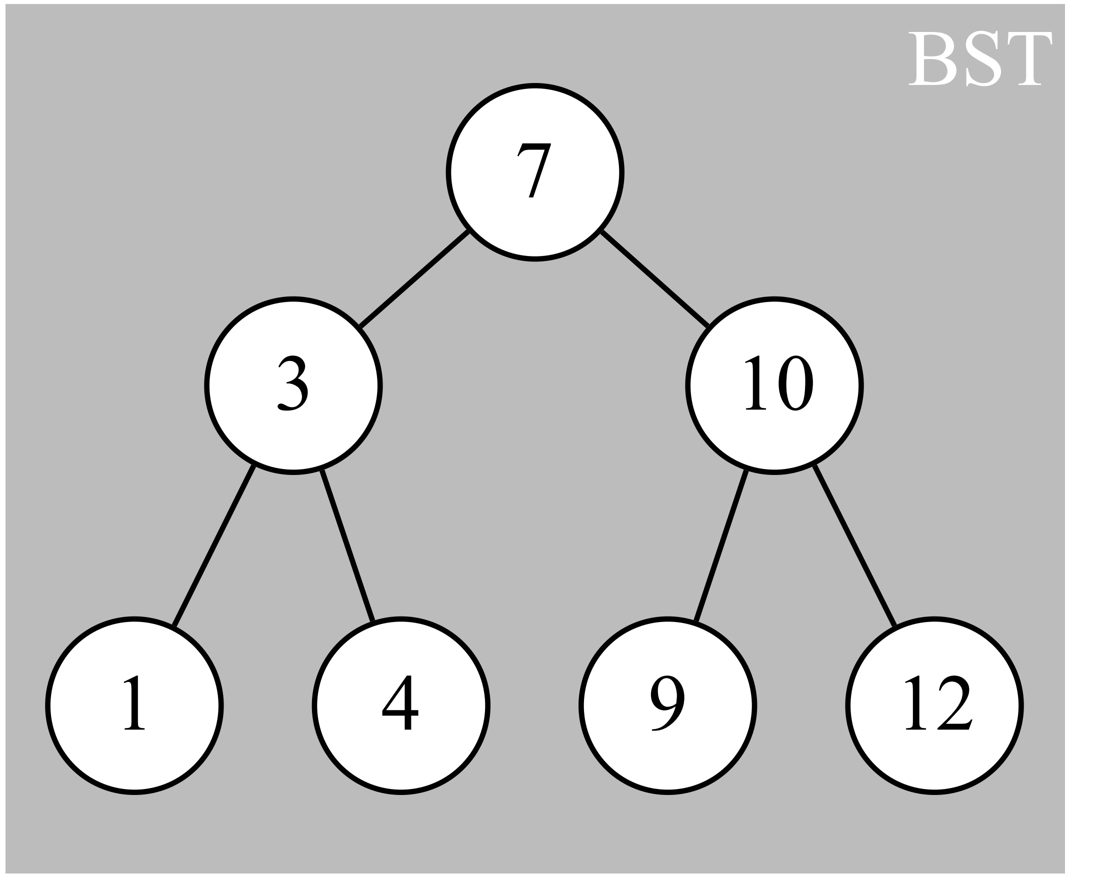
Differences between Heaps and BSTs



Q: Is it possible to build an n-node BST in $o(n \log n)$ time in the comparison-based model?

Differences between Heaps and BSTs





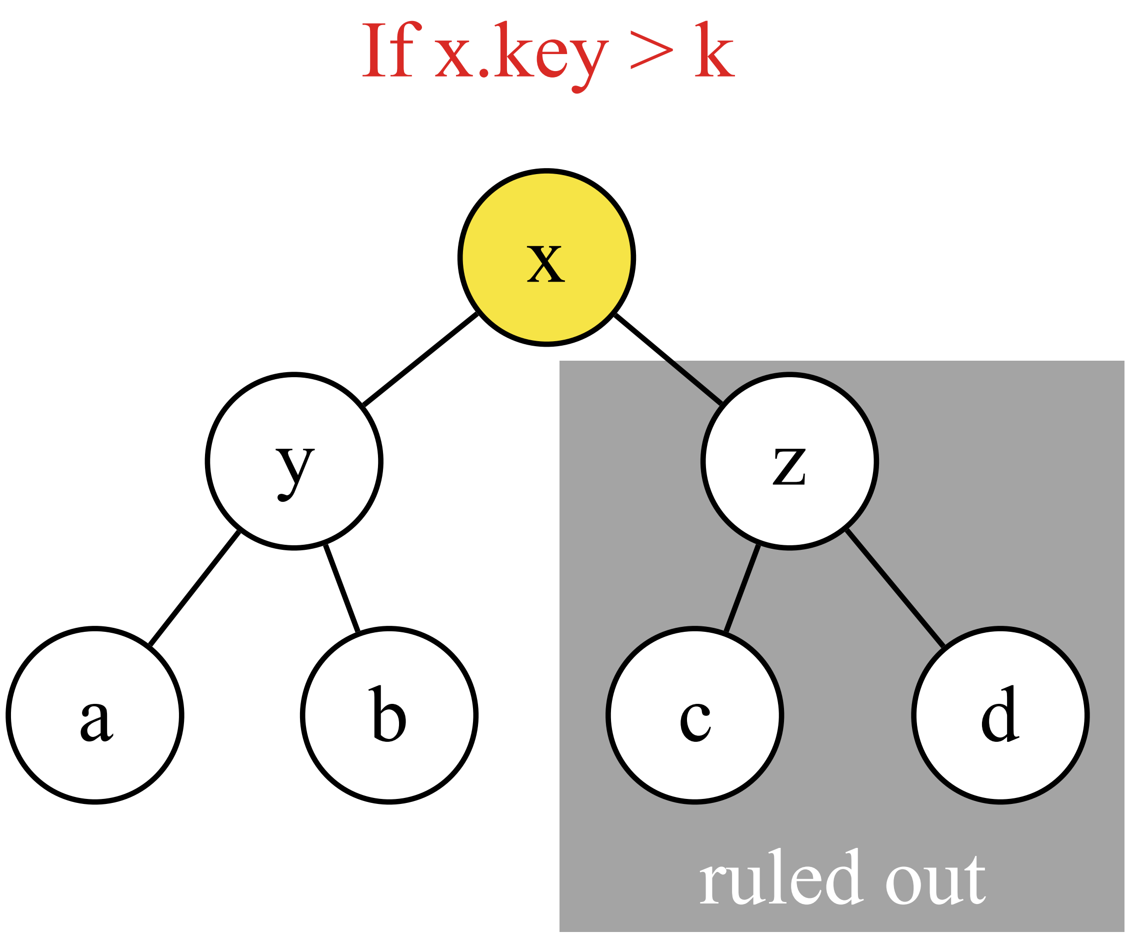
A: No. Otherwise build a BST ($o(n \log n)$ time) followed by the traversal ($O(n)$ time) implements a sorting algorithm. $\rightarrow\leftarrow$

Query Operations

Tree-Search(x, k)

Output "Yes" if some node y in the tree rooted at x has value k, or otherwise "No."

--- Example ---



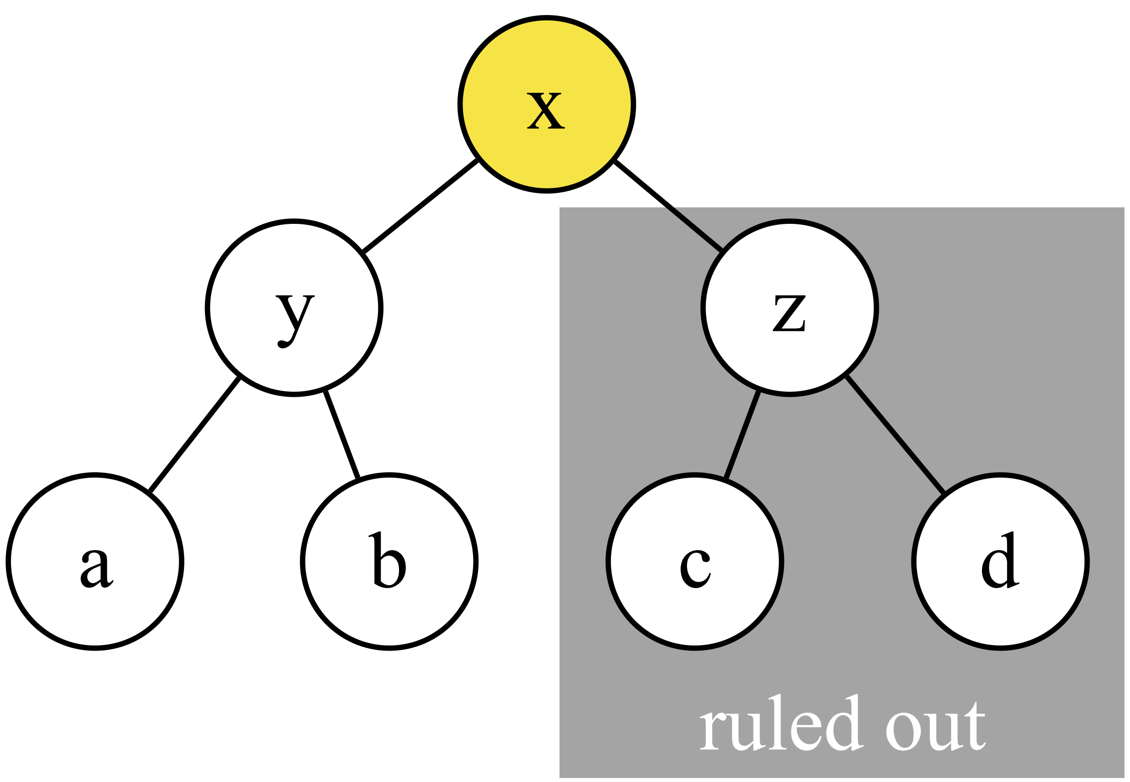

```
bool Tree-Search(x, k){

  if(x == Null) return No;

  if(x.key == k) return Yes;

  if(x.key < k)
    return Tree-Search(x.right, k);
  else
    return Tree-Search(x.left, k);
}
```

Tree-Search(x, k)

Output "Yes" if some node y in the tree rooted at x has value k, or otherwise "No."

--- Running Time ---

If x.key > k



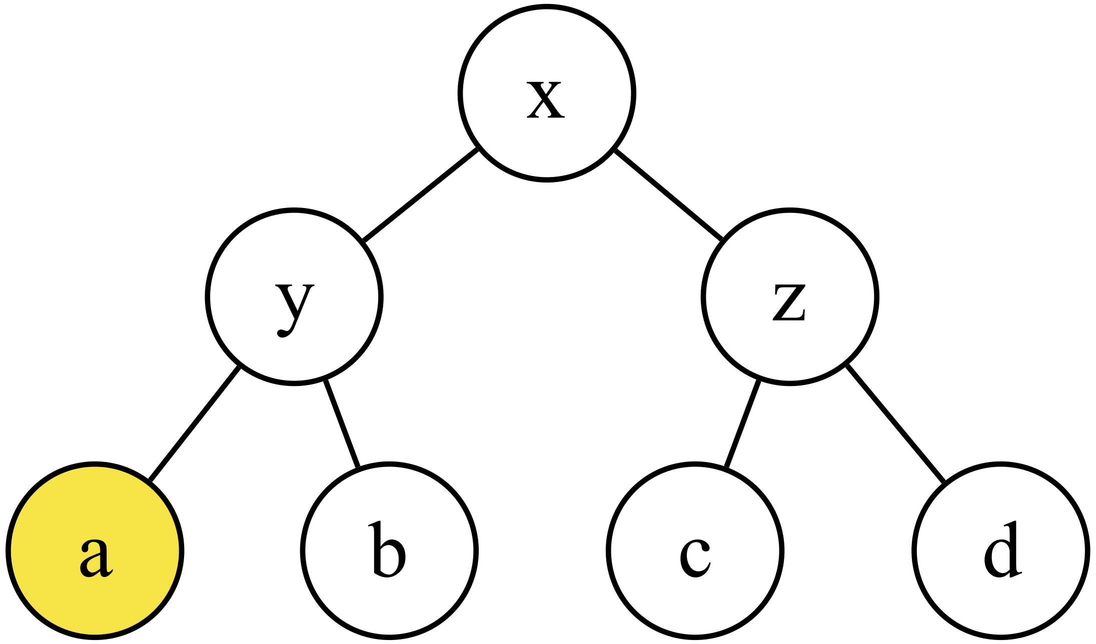
The runtime is O(height).

What is largest possible height of an n-node tree?

Tree-Min(x)

Output the minimum key value in the tree rooted at x has value k, and if no such a value exist, output "-∞."

--- Example ---




```
int Tree-Min(x){

  if(x == Null) return -∞;

  if(x.left != Null){
      return Tree-Min(x.left);
  }else{
      return x.key;
  }
}
```

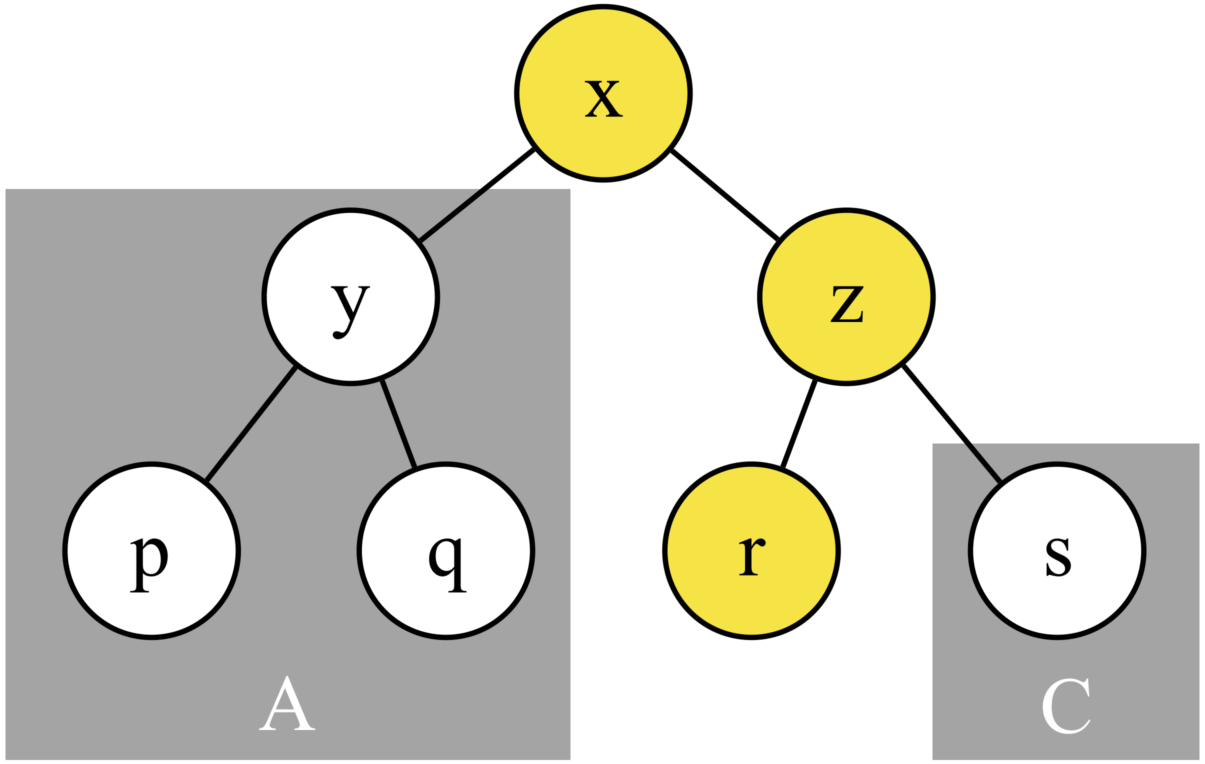
Exercise

Why is the implementation of Tree-Min correct?

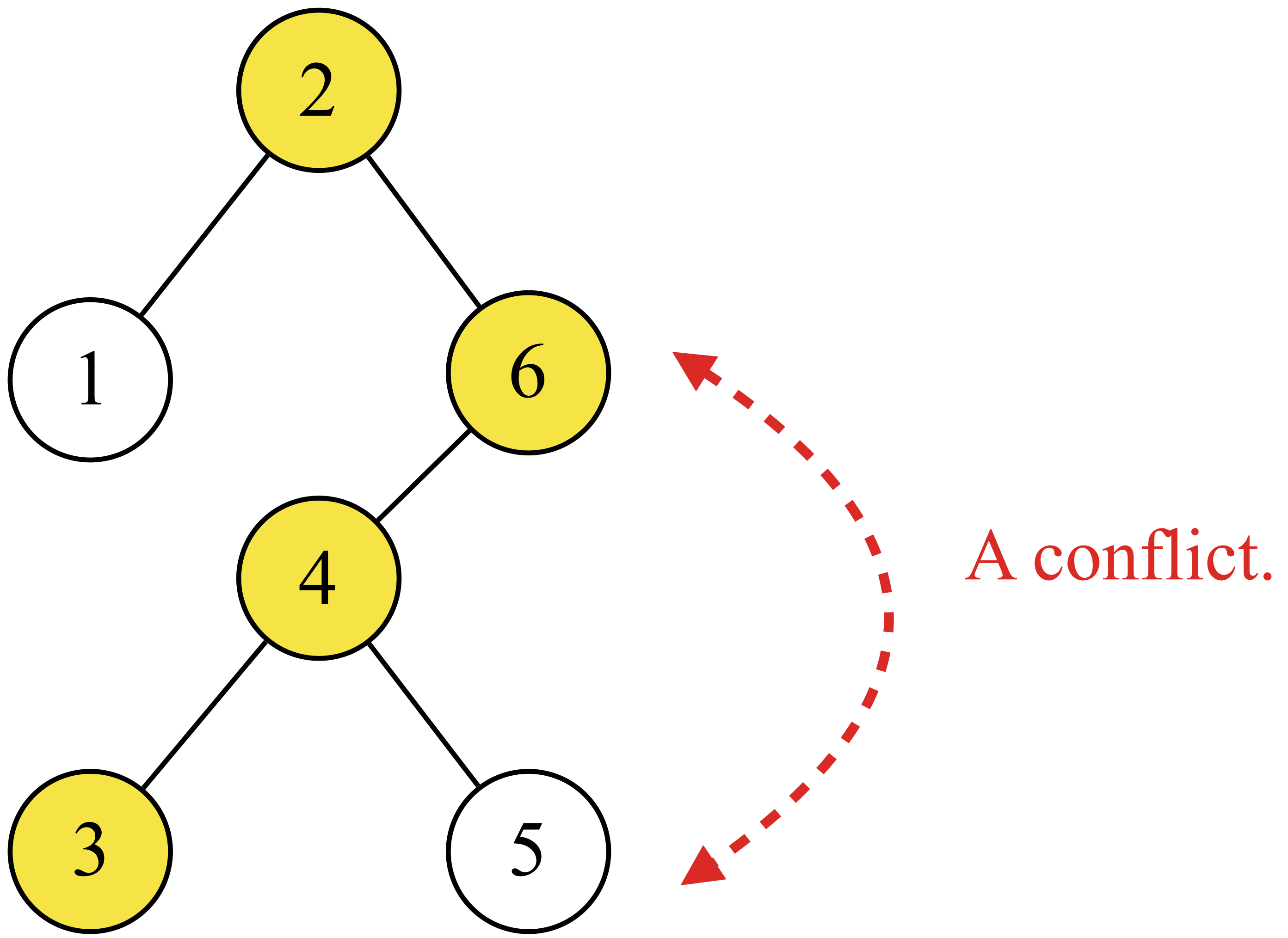
How to implement Tree-Max?

Exercise

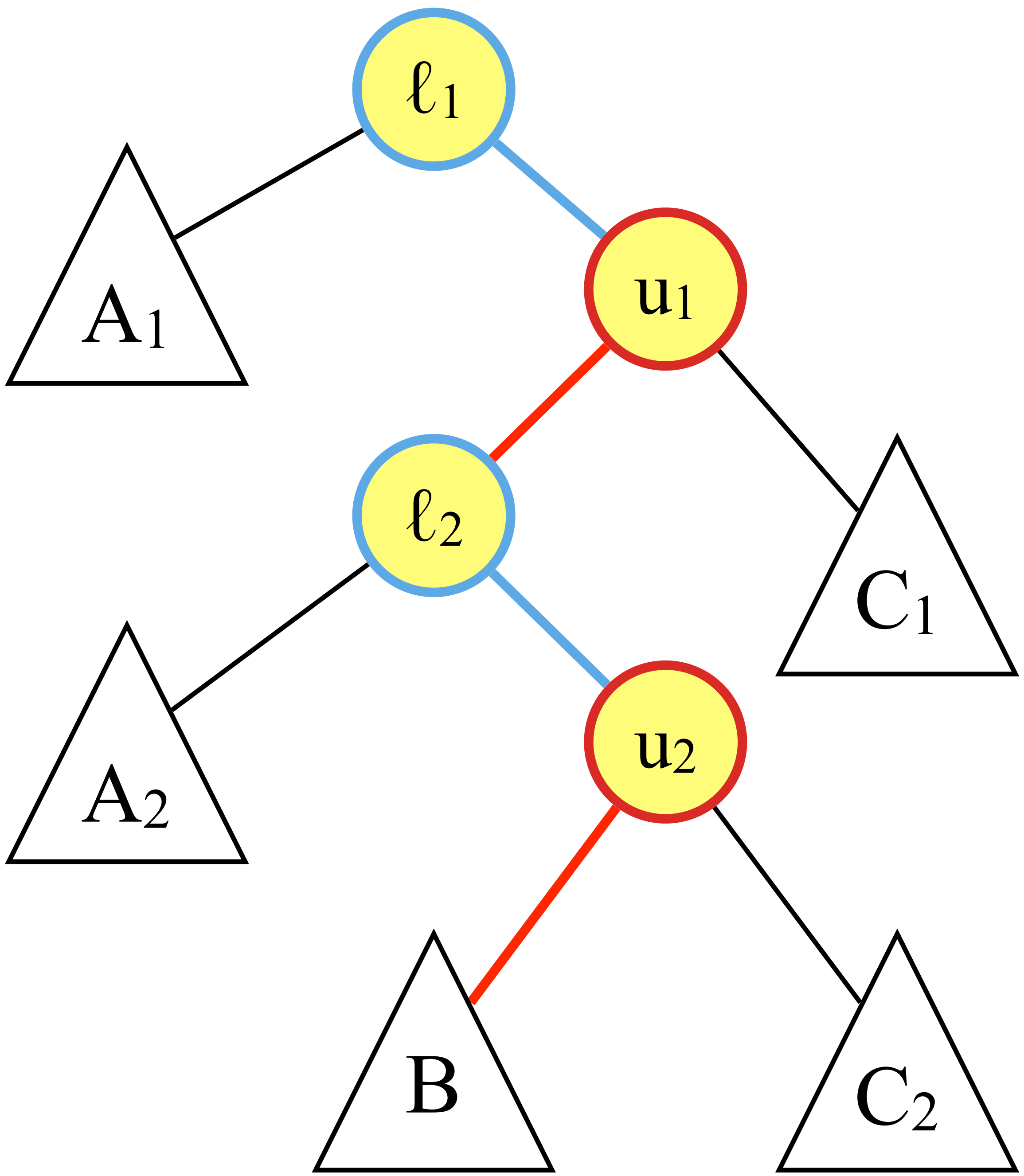
Let P be a root-to-leaf path of a binary search tree. Let A be the set of nodes to the left of P. Let B be the set of nodes on P. Let C be the set of nodes to the right of P. Prove or disprove that for any $a \in A$, $b \in B$, $c \in C$, we have $a.key \leq b.key \leq c.key$.



A Counter-Example



Property



$\max(\ell_1, \ell_2, \ldots) \leq B \leq \min(u_1, u_2, \ldots)$

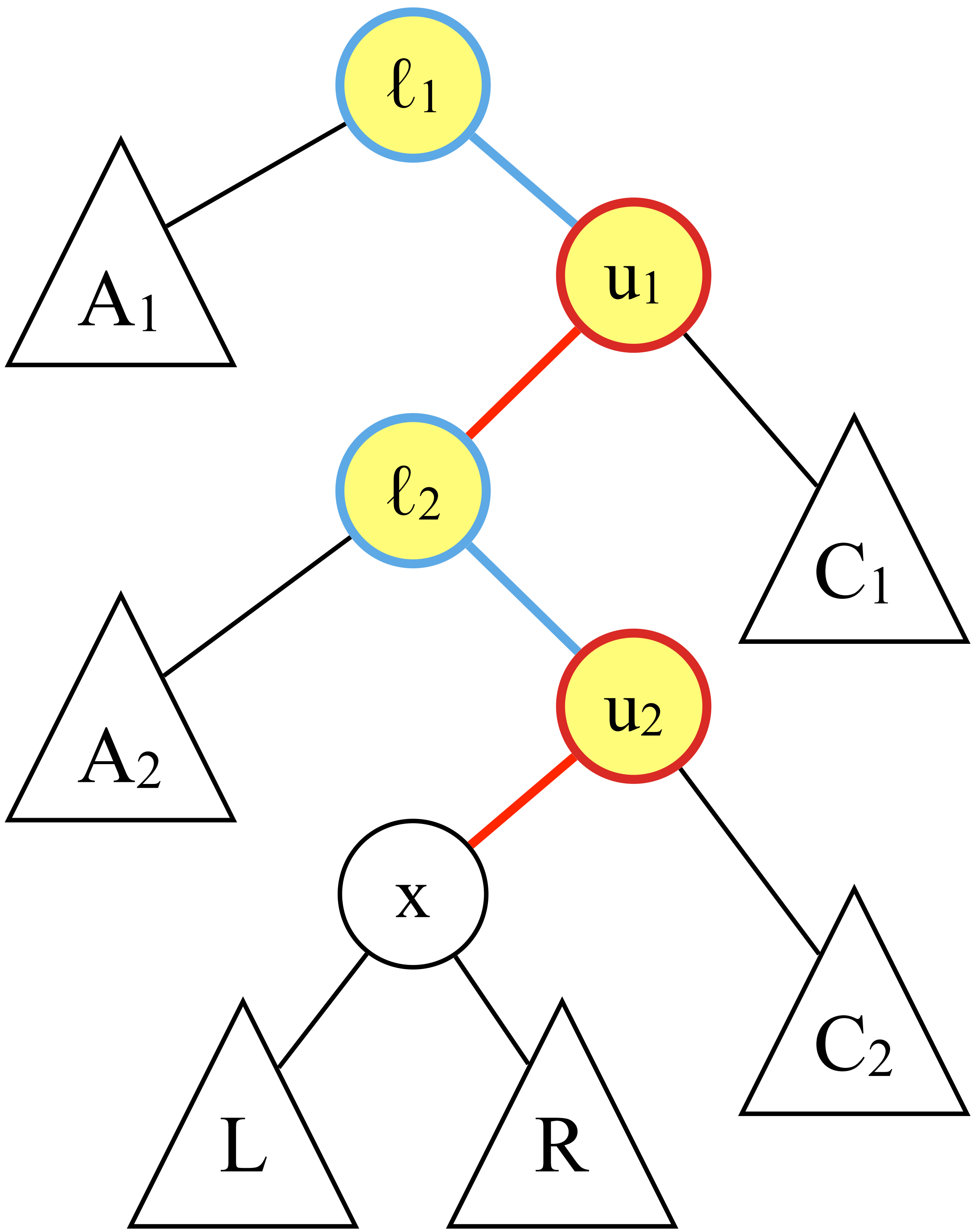
$\ell_1 \leq \ell_2 \leq \ldots \leq u_2 \leq u_1$

$A \leq \max(\ell_1, \ell_2, \ldots)$

$\min(u_1, u_2, \ldots) \leq C$

$A \leq B \leq C$

Tree-Successor(x)

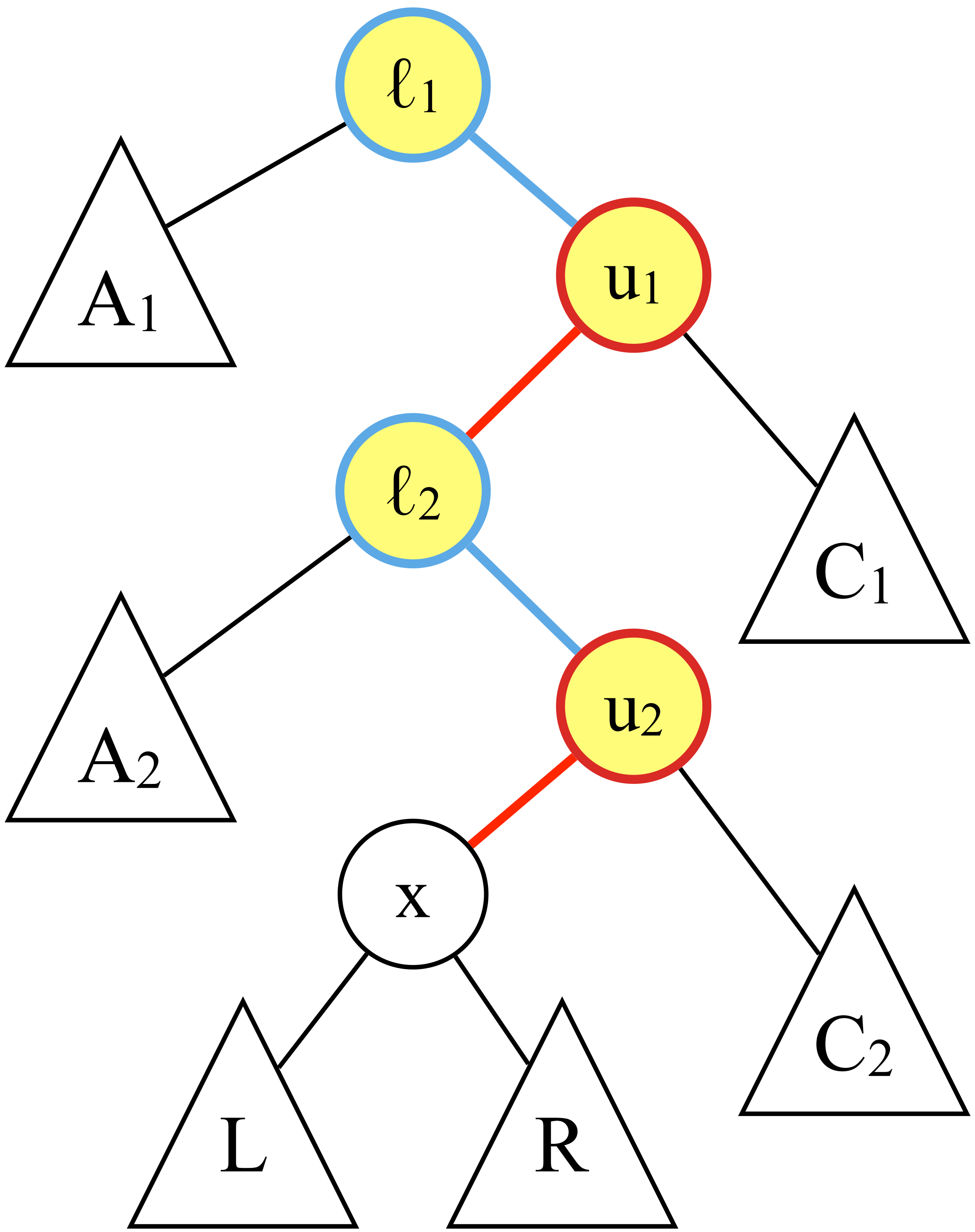


Output the smallest key ≥ x.key.

(Assume all keys are distinct, or tie-break arbitrarily.)

Tree-Successor(x) is in the tree rooted at x or some u_i. (Why?)

Tree-Predecessor(x)



Output the largest key ≤ x.key.

(Assume all keys are distinct, or tie-break arbitrarily.)

Tree-Predecessor(x) is in the tree rooted at x or some ℓ_i. (Why?)

Exercise

Give an implementation of Tree-successor(x) so that invoking the function k times can be done in O(k + height) time.

Optimal BST

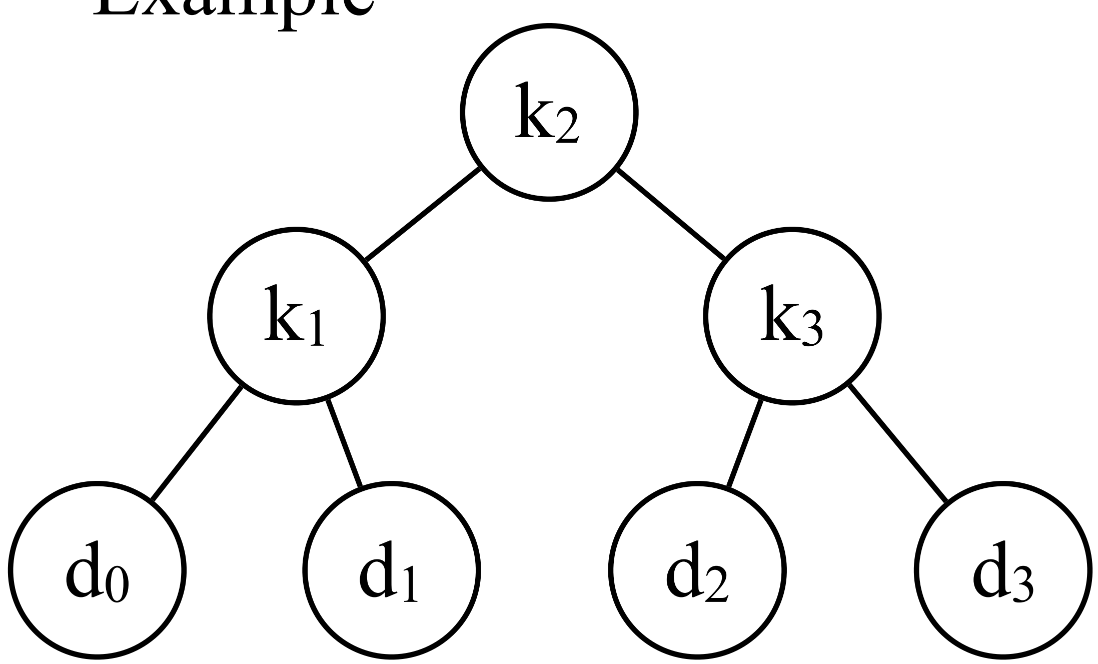
Optimal BST

Input: n keys $k_1 < k_2 < ... < k_n$ and a distribution of queried values:

- The queried value is k_i with probability p_i for each i in [1, n].
- The queried value is in-between (k_i, k_{i+1}) (assume it hits a dummy node) with probability q_i for each i in [1, n] where $k_0 = -\infty$ and $k_{n+1} = \infty$.

Output a BST so that the expected search cost is minimized.

--- Example ---



key	search cost
k_1	$2*p_1$
k_2	$1*p_2$
k_3	$2*p_3$
d_0	$3*q_0$
d_1	$3*q_1$
d_2	$3*q_2$
d_3	$3*q_3$

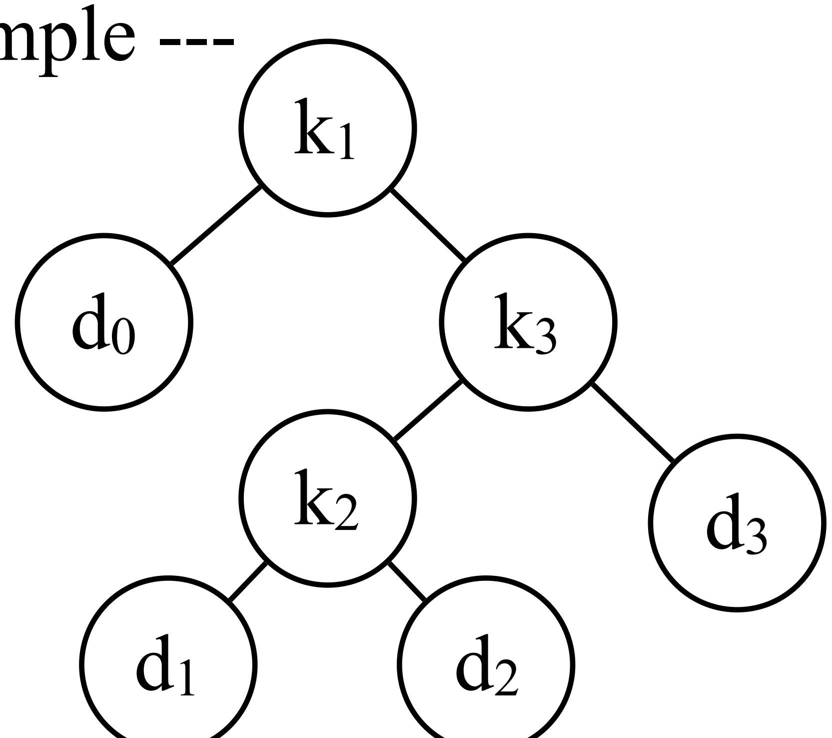
Optimal BST

Input: n keys $k_1 < k_2 < \ldots < k_n$ and a distribution of queried values:

- The queried value is k_i with probability p_i for each i in [1, n].
- The queried value is in-between (k_i, k_{i+1}) (assume it hits a dummy node) with probability q_i for each i in [1, n] where $k_0 = -\infty$ and $k_{n+1} = \infty$.

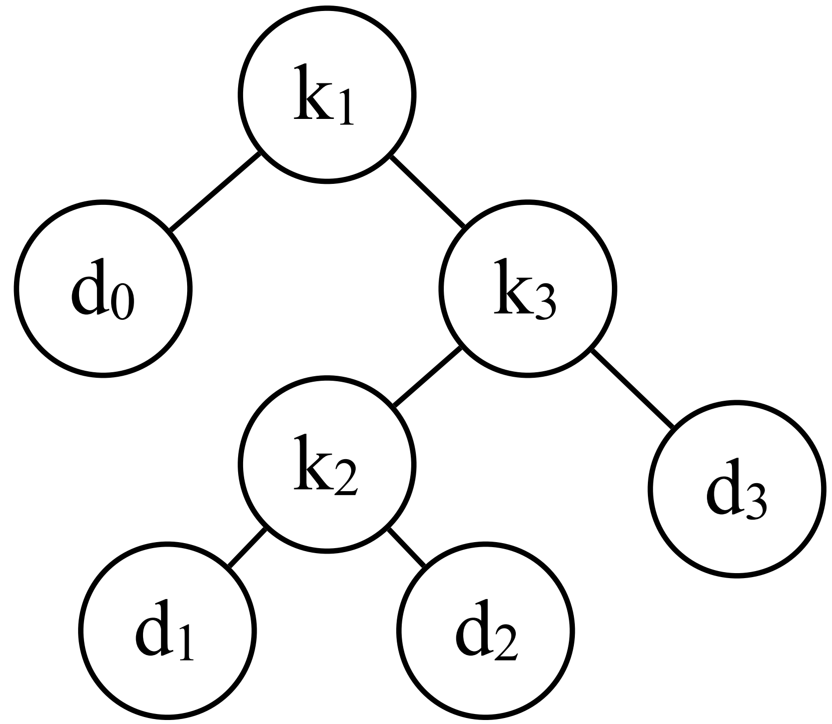
Output a BST so that the expected search cost is minimized.

--- Example ---



key	search cost
k_1	$1*p_1$
k_2	$3*p_2$
k_3	$2*p_3$
d_0	$2*q_0$
d_1	$4*q_1$
d_2	$4*q_2$
d_3	$3*q_3$

Optimal BST

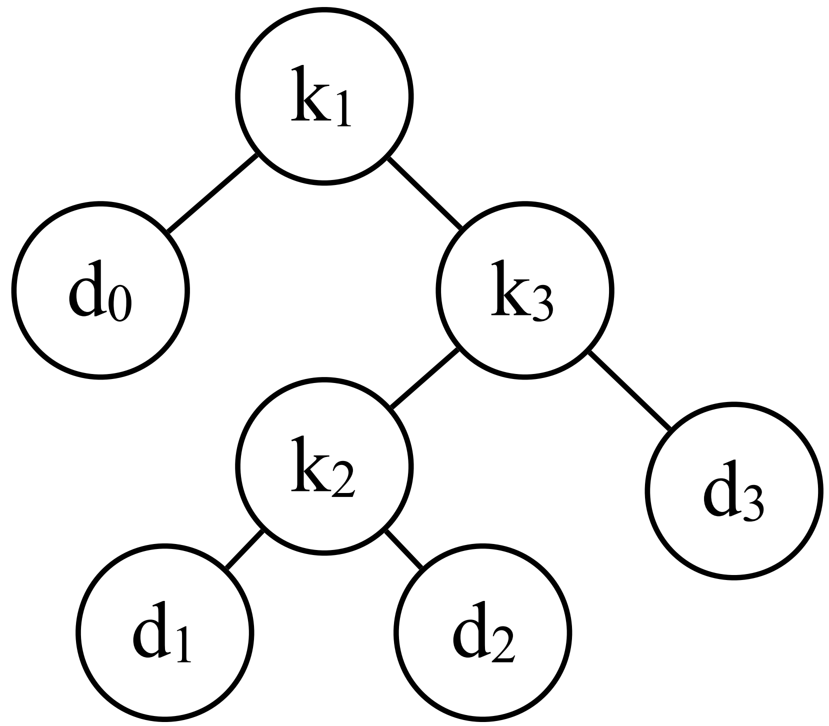


Claim. Every t-key subtree has t+1 dummy nodes.

count	a key node	a dummy node
parent	1 (except the root)	1
children	2	0

Let X be # dummy node.
By the double counting method on # (parent,child) pairs, we get:
$t - 1 + X = 2t + 0X$,
so $X = t+1$.

Optimal BST

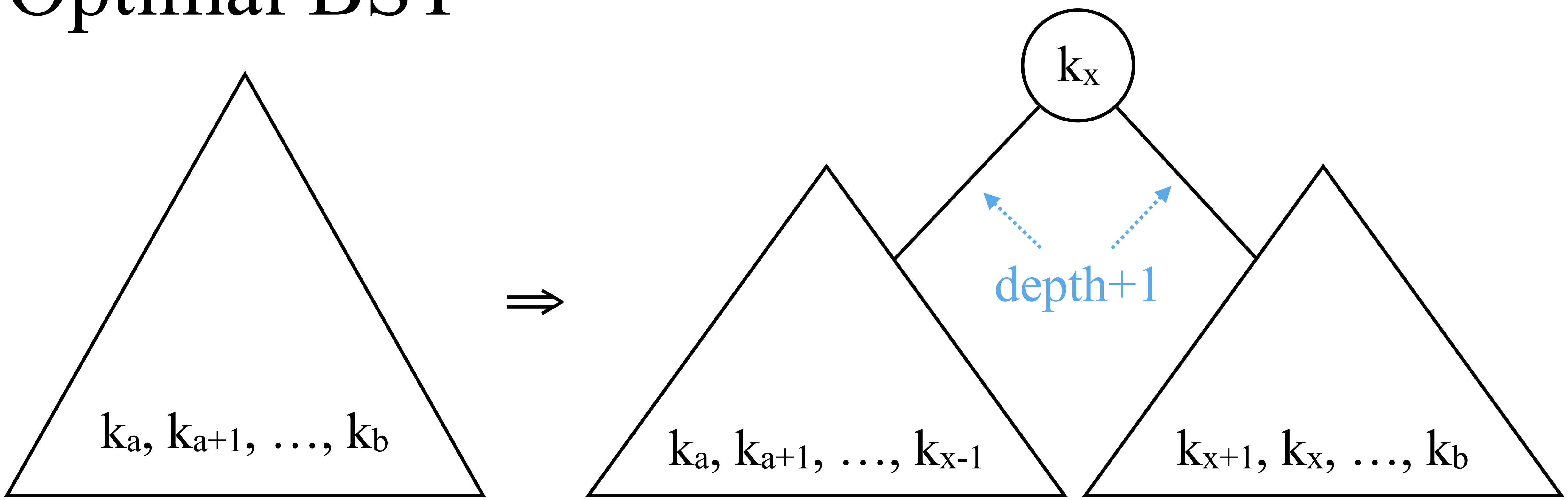


Claim. Every subtree that contains keys k_a, k_{a+1}, ..., k_b also contains dummy nodes d_{a-1}, d_a, ..., d_b.

Recall the property of BST: $A \leq \max(\ell_1, \ell_2, \ldots) \leq B \leq \min(u_1, u_2, \ldots) \leq C$,

so the keys and dummy nodes form a continous interval.

Optimal BST



Let $W(a, b)$ be the sum of probabilities associated with $k_a, k_{a+1}, \dots, k_b$ and $d_{a-1}, d_a, \dots, d_b$, and $Opt(a, b)$ be the smallest expected cost of an optimal BST that comprises the same keys and dummy nodes.

<u>Claim</u>. $Opt(a, b)$
$= \min_x Opt(a, x-1) + W(a, x-1) + p_x + Opt(x+1, b) + W(x+1, b)$
$= \min_x Opt(a, x-1) + Opt(x+1, b) + W(a, b)$.

Divide and Conquer

```
OBST(a, b){ // return the smallest expected cost of an optimal BST that comprises keys k_a, k_{a+1}, ..., k_b and dummy nodes d_{a-1}, d_a, ..., d_b

    if(a-1 == b) return q_b; // an empty subtree

    opt = ∞;

    for(x = a; x ≤ b; ++x){ // pick a root at k_x
        if(OBST(a, x-1)+OBST(x+1, b)+W(a, b) < opt){
            opt = OBST(a, x-1)+OBST(x+1, b)+W(a, b);
        }
    }
    return opt;
}
------
The initial call is OBST(1, n);
```

Dynamic Programming

```
OBST(a, b, sol[][]){ // return the smallest expected cost of an optimal
BST that comprises keys k_a, k_{a+1}, ..., k_b and dummy nodes d_{a-1}, d_a, ..., d_b

    if(a-1 == b) return q_b; // an empty subtree
    if(sol[a][b] < ∞) return sol[a][b];
    opt = ∞;

    for(x = a; x ≤ b; ++x){ // pick a root at k_x
        if(OBST(a, x-1, sol)+OBST(x+1, b, sol)+W(a, b) < opt){
            opt = OBST(a, x-1, sol)+OBST(x+1, b, sol)+W(a, b);
        }
    }
    return sol[a][b] = opt;
}
```

The initial call is OBST(1, n, sol = {∞});

Output the OBST

```
OBST(a, b, sol[][]){ // return the smallest expected cost of an optimal BST that comprises keys k_a, k_{a+1}, ..., k_b and dummy nodes d_{a-1}, d_a, ..., d_b

    if(a-1 == b) return q_b; // an empty subtree
    if(sol[a][b] < ∞) return sol[a][b];
    opt = ∞;

    for(x = a; x ≤ b; ++x){ // pick a root at k_x
        if(OBST(a, x-1, sol)+OBST(x+1, b, sol)+W(a, b) < opt){
            opt = OBST(a, x-1, sol)+OBST(x+1, b, sol)+W(a, b);
            root[a][b] = x; }
    }
    return sol[a][b] = opt;
}
------
The initial call is OBST(1, n, sol = {∞});
```

Running Time of OBST: $O(n^3)$.

Output the OBST

OutputOBST(a, b){ // output an OBST that comprises keys k_a, k_{a+1}, ..., k_b and dummy nodes d_{a-1}, d_a, ..., d_b (in in-order)

```
    if(a-1 == b){ output d_b; return; }

    OutputOBST(a, root[a][b]-1);

    output k_root[a][b];

    OutputOBST(root[a][b]+1, b);
}
```

Exercise

Knuth proves that there always exists an OBST so that

$$\text{root}[i][j-1] \le \text{root}[i][j] \le \text{root}[i+1][j] \text{ for every } i, j \text{ in } [1, n].$$

Use this fact to compute OBST in $O(n^2)$ time.

Running Time of OBST:
$O(n^2)$.