Binary Search Trees-I

ESO207

Indian Institute of Technology, Kanpur

Introduction

- We will now begin our study of the *Search trees* data structure that supports dynamic set operations such as SEARCH, INSERT, DELETE, MINIMUM, MAXIMUM, PREDECESSOR and SUCCESSOR.
- A search tree can be used as a dictionary structure or even as a priority-queue.
- The basic operations on binary search trees take time **proportional to the height of the tree.**
  1. For a complete binary tree with n nodes, these operations run in time $\Theta(\log n)$.
  2. However, if the tree is a linear chain of n nodes, the same operations take worst-case $\Theta(n)$ time.

Introduction

- We will first look at simple binary search trees.
- Next we will see a variation of binary search trees, called *red-black* trees, which guarantee that the height is $O(\log n)$, and hence have good $O(\log n)$ worst-case performance for search, insertion, deletions, etc.

Introduction: Operations on a binary search tree

Typical operations on a binary search tree are:

1. Search for a key
2. Insertion of a key
3. Deleting a node
4. Finding predecessor of a node, successor of a node.
5. Walking in a binary search tree to print its values in sorted order
6. Finding minimum and maximum element.

Binary Search Trees

- A binary search tree is a binary tree in which each node has a distinguished field *key*.
- Nodes may have other satellite data but they are not used directly in operations on a binary search tree.
- Each node is a structure containing the following fields (at a minimum).
  1. the key field, denoted *key*,
  2. a pointer to the left child denoted *left*, and,
  3. a pointer to the right child denoted *right*, and,
  4. a pointer to the parent node denoted *parent.*

Representing binary search trees

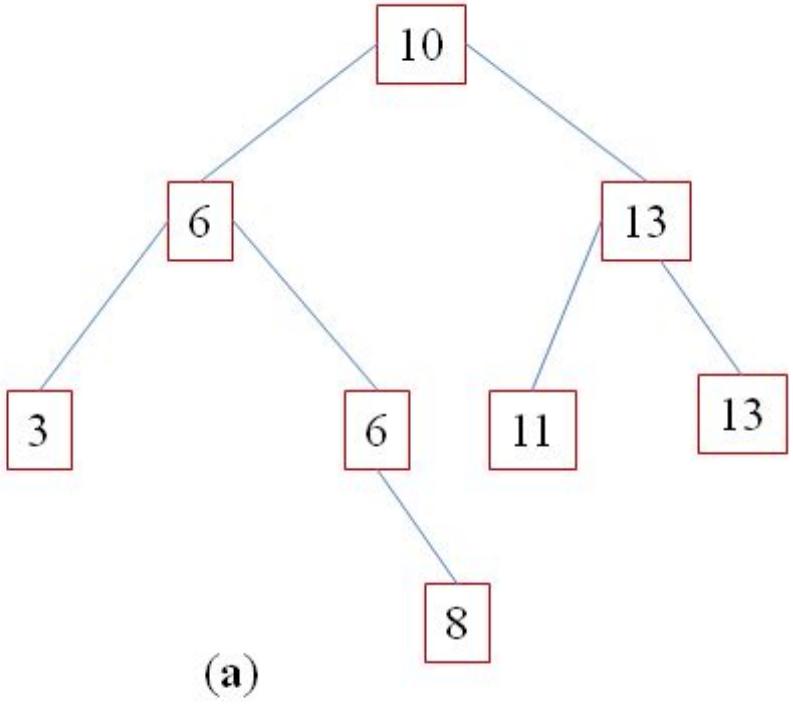
```
struct BSTNode{
    BSTNode *left
    BSTNode *right
    BSTNode *parent
    KEY key
}
```

Binary Search Tree Property

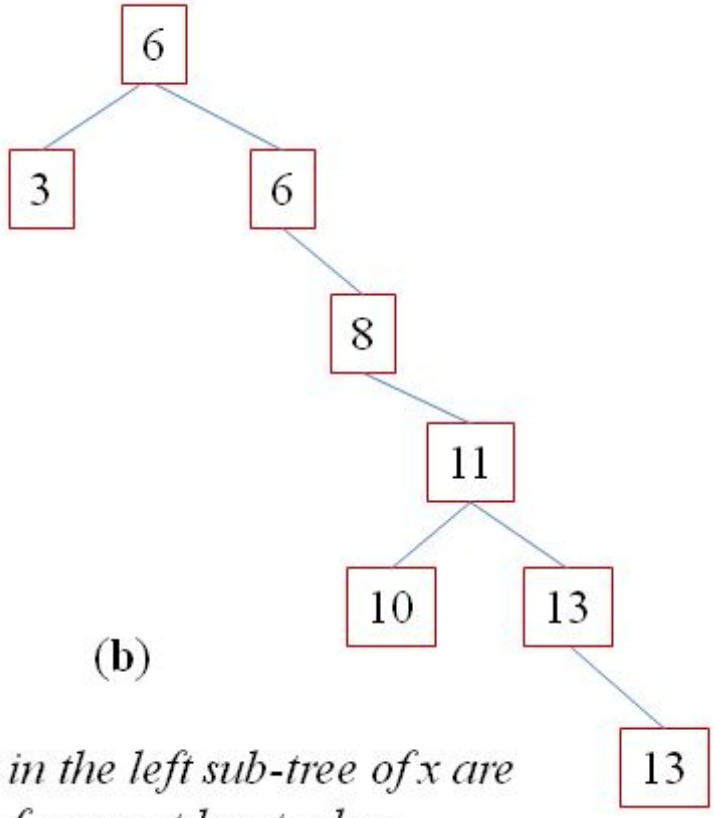
The salient property of a binary search tree is the following ***binary-search-tree property***.

1. For any node x of a binary search tree, all the keys in the left-subtree of x are at most $x.key$, and
2. All the keys in the right-subtree of x are at least $x.key$.

- If y is a node in the left subtree of x, then, $y.key \leq x.key$.
- If y is a node in the right subtree of x, then, $y.key \geq x.key$.



(a)



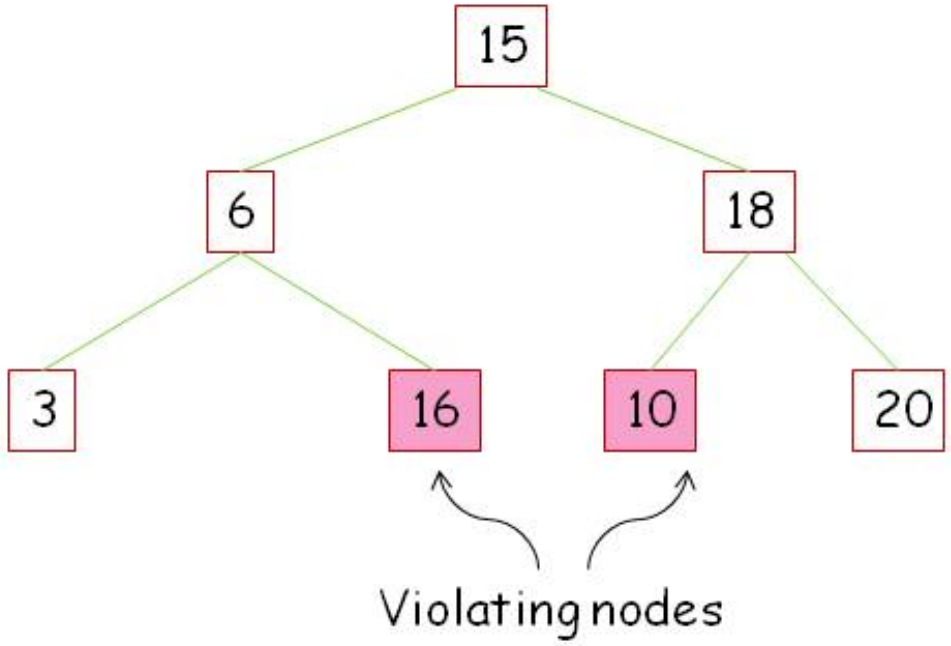
(b)

In a binary search tree, for any node x, all the keys in the left sub-tree of x are at most x.key and all the keys in the right sub-tree of x are at least x.key. Different binary search tree structures can represent the same set of values, for example, the two trees shown above have the same set of keys. However, the tree in (b) is less efficient for purposes of search, insertion etc.

Figure: Illustration of a binary search tree

Remark

- *Remark*. Note that the following condition does not necessarily satisfy the binary-search-tree property:
  1. for every node, the key at the node is no smaller than the key at its left-child, (if one exists) and,
  2. key at the node is no larger than the key at its right-child.



Digression: Tree traversals

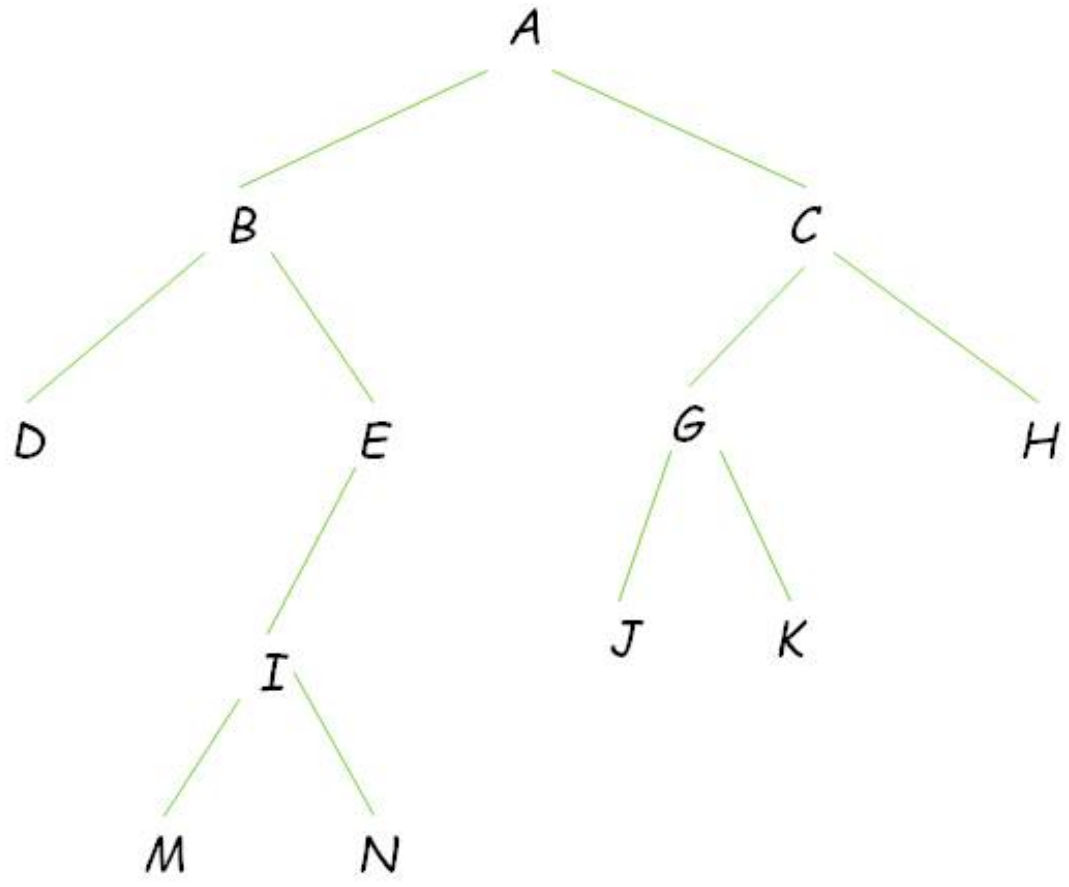
There are three natural and well-known *recursive* tree traversals.

- **inorder**
- **preorder**
- **postorder**

Inorder tree traversal

- If at leaf node, print the node.
- Otherwise,
  1. Recursively, perform inorder traversal of left subtree.
  2. Print the root node.
  3. Recursively perform inorder traversal of right subtree.

Inorder tree traversal: Example



In order traversal sequence

D B M I N E A J G K C H

Inorder traversal sequence

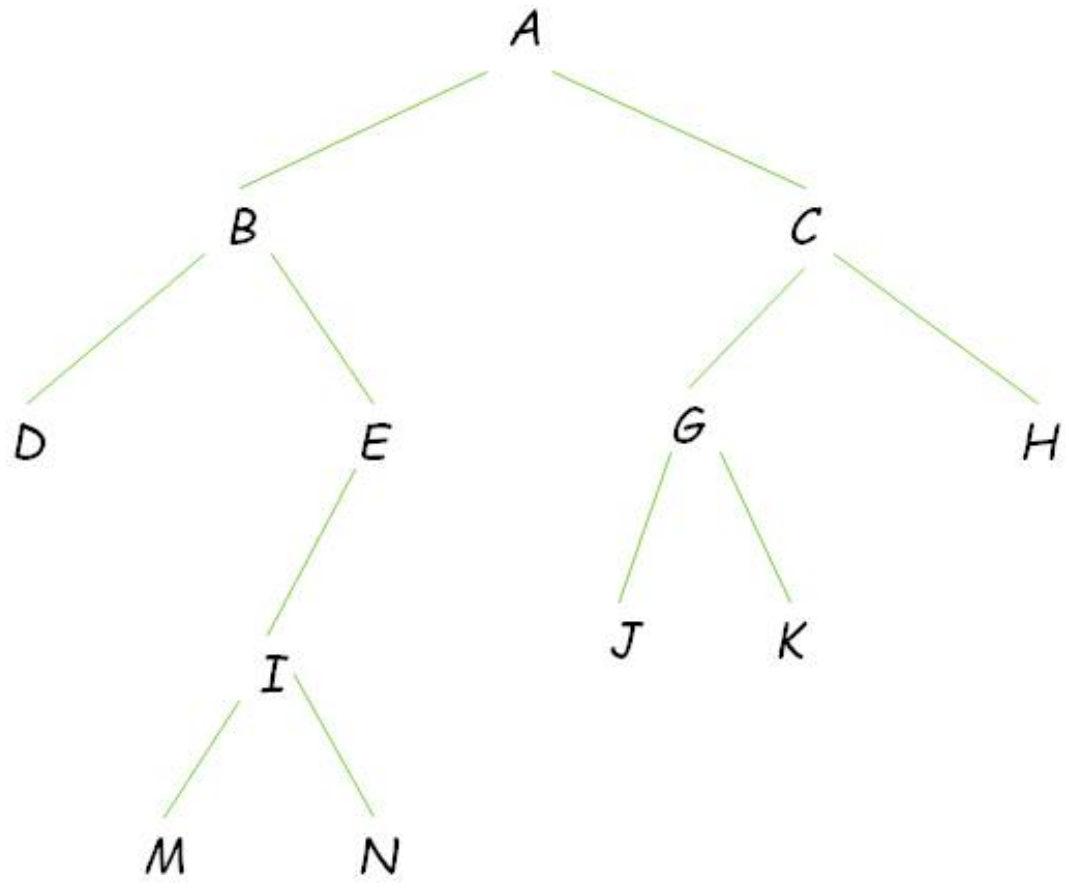
- Summary: Left-subtree, then root, then Right-subtree (recursively).

Preorder traversal sequence

Summary: Root, then left-subtree, then right-subtree.

- Print root node.
- If left child exists, then, recursively call preorder traversal on left subtree.
- If right child exists, then, recursively call preorder traversal on right subtree.

Preorder tree traversal: Example



Preorder traversal

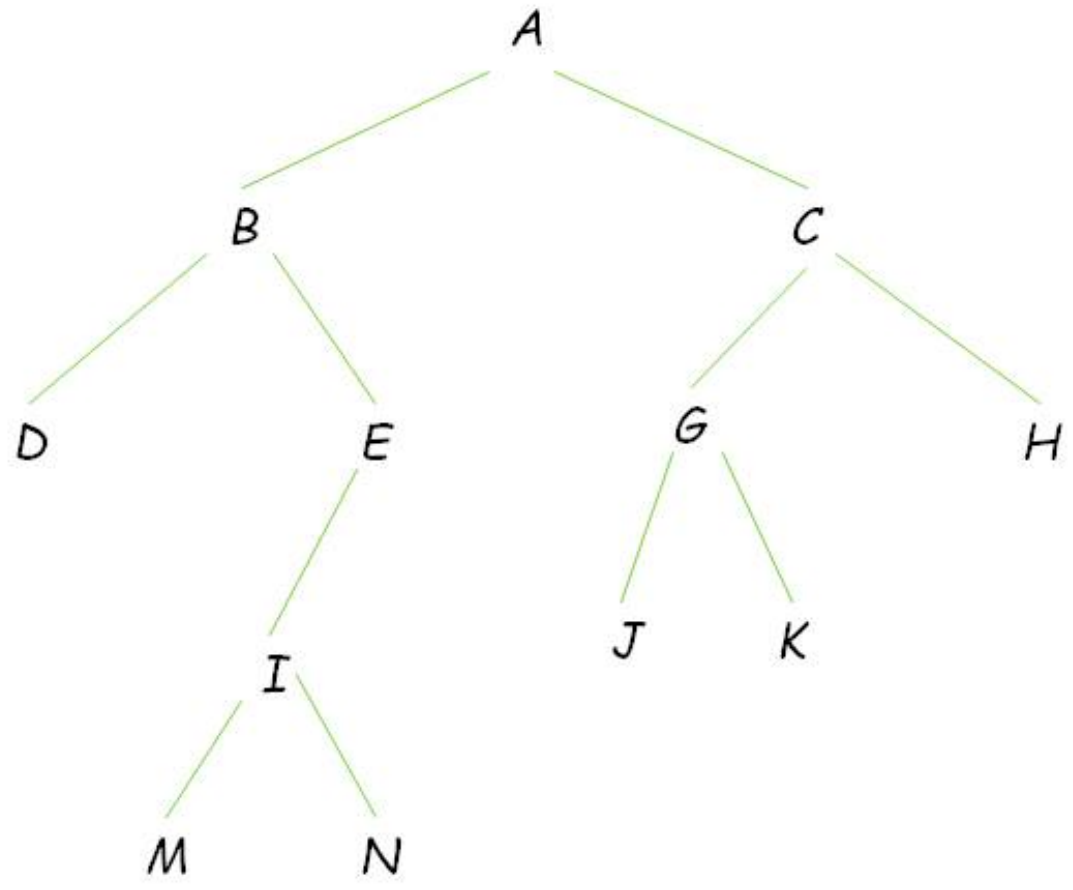
A B D E I M N C G J K H

Postorder traversal sequence

Summary: left-subtree, then right-subtree, then root.

- If left child exists, then, recursively call postorder traversal on left subtree.
- If right child exists, then, recursively call postorder traversal on right subtree.
- Print root node.

Postorder tree traversal: Example

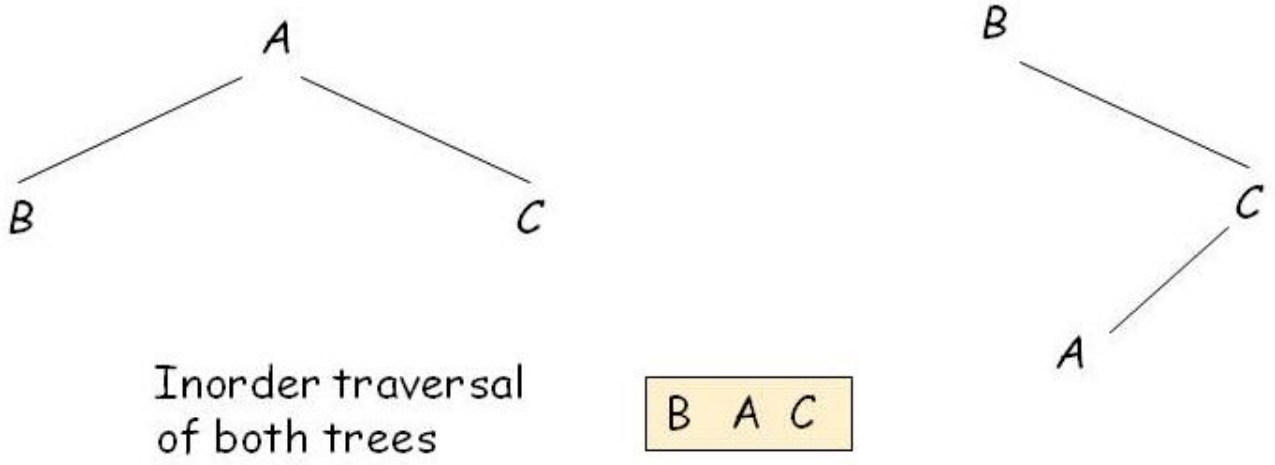


Postorder traversal sequence

D M N I E B J K G H C A

Remark on traversals

- A single traversal sequence does not uniquely determine a rooted, ordered tree, that is, multiple rooted ordered trees may have the same traversal sequence.
- Example for inorder traversal:

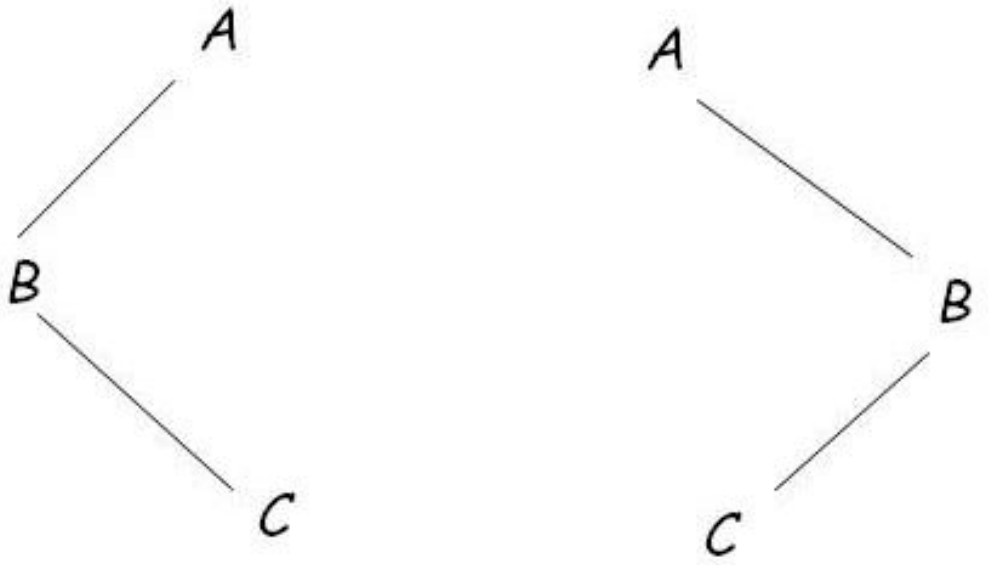


- Similar examples can be designed for preorder and postorder traversals.

Remark on traversals

- Inorder traversal sequence together with any one of preorder traversal or postorder traversal sequence uniquely determines a rooted, ordered tree.
- **Problem:** Give an efficient algorithm to deduce the rooted, ordered tree given the inorder traversal sequence and preorder (or, postorder) traversal sequence.
- However, given preorder and postorder traversal sequence, it does not uniquely determine a rooted, ordered tree.

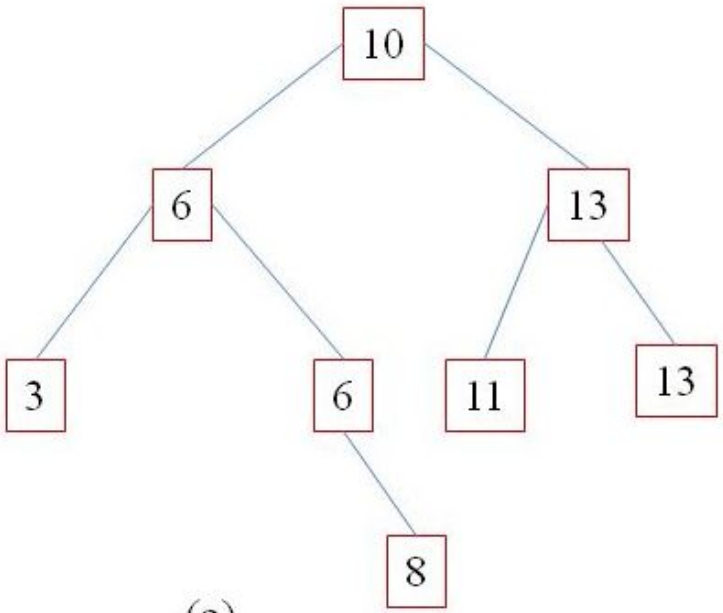
Example

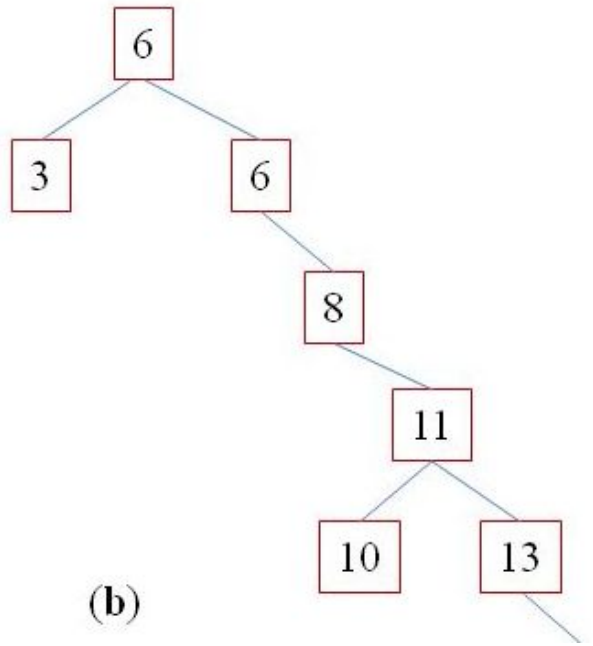


Preorder traversal	A B C
Postorder traversal	C B A

Back to Binary Search trees

- Binary search tree property: For any node x, $x.key$ is no smaller than any key in the left subtree of x and is no larger than any key in the right subtree of x.

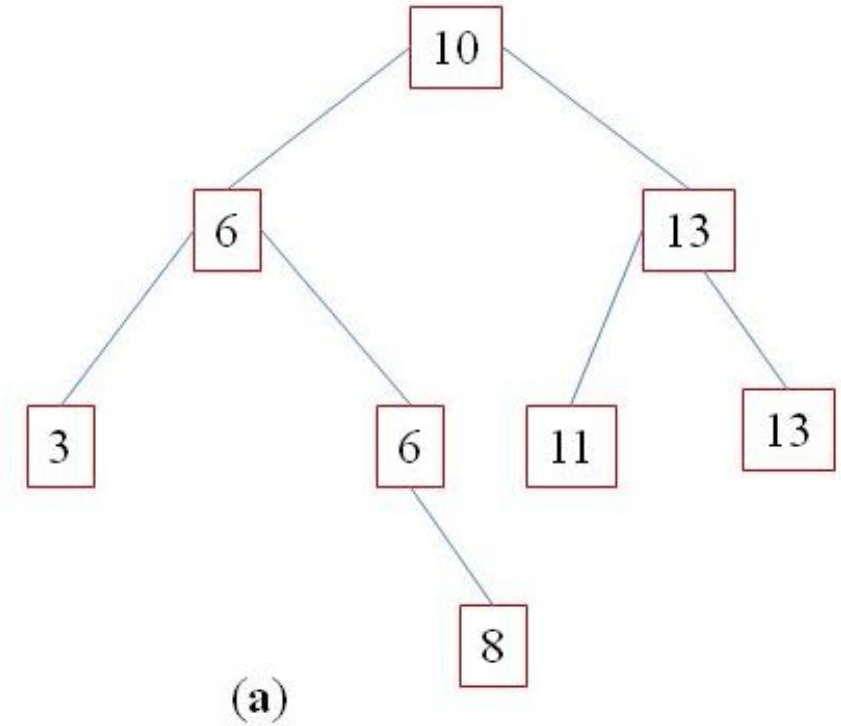




Printing keys of Binary search tree in sorted order

- The binary-search-tree property ensures that an *inorder* traversal prints the keys in sorted order.
- Inorder $\equiv$ Left subtree, root, Right subtree.
- All keys in left subtree $\leq$ key at root $\leq$ all keys in right subtree.
- This holds for each sub-tree.

Printing keys in sorted order



(a)

- Inorder sequence is 3 6 6 8 10 11 13 13

Searching in binary search tree

- The key $T.key$ at the root node partitions the set of keys into two parts,
  1. the keys less than or equal to $T.key$ are in the left-subtree, and,
  2. the keys greater than or equal to $T.key$ are in the right sub-tree.
- If the given key k is less than the key at the root node, we search further in the left subtree.
- If k is larger than the key at the root node, then we search in the right subtree.

Search: pseudo-code

A recursive search procedure

SEARCH(BST T, *Key* k)
// Search for a node with key k in binary search tree T

```
1.   if T == NIL  or  T.key == k
2.         return T
3.   if k < T.key
4.         SEARCH(T.left, k)
5.   else
6.         SEARCH(T.right, k)
```

Search: iterative version

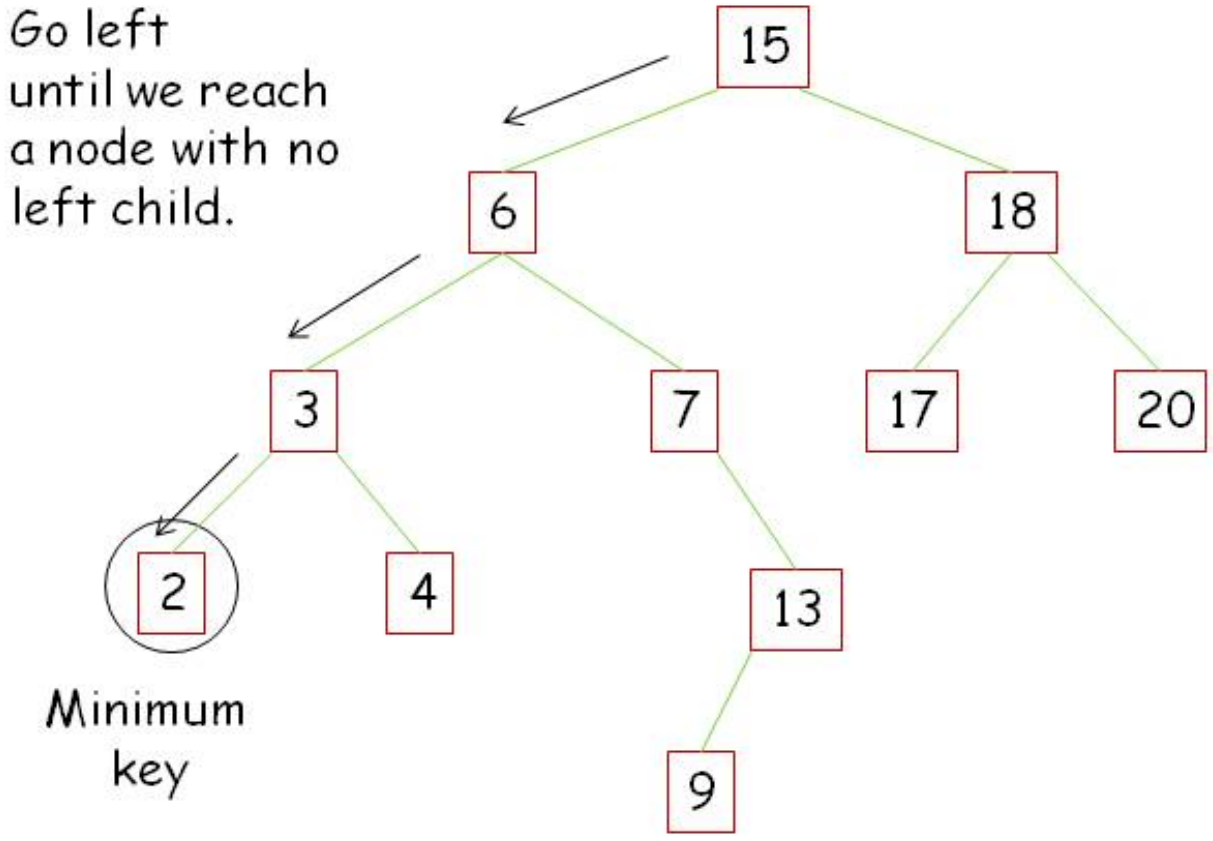
An iterative equivalent of the search procedure.

SEARCH(BST T, *Key* k)

```
1.  while T ≠ NIL and T.key ≠ k
2.        if k < T.key
3.              T = T.left
4.        else
5.              T = T.right
6.  return T
```

Minimum key in binary search tree

- How can we compute the minimum key in the binary search tree?
- All the keys in the left-subtree are less than or equal to the key at the root.
- So, the minimum of the keys in the left-subtree is the minimum key in the tree.
- An algorithm would be to keep following the left node, until we reach a node that has no left-child.
- The key at this node is the minimum in the binary search tree.



Finding Minimum in a binary search tree

Minimum: Pseudo code

FIND-MINIMUM(T)
// finds the minimum key in the binary search tree
1. $x = T$
1. **while** $x.left \neq$ NIL
2. $\quad\quad x = x.left$
3. **return** $x.key$

Maximum

- Analogously, the ***maximum*** key in the tree is the obtained by following the right-child pointer, until we arrive at a node that has no right-child.

  FIND-MAXIMUM(T) // finds the minimum key in the binary search
  1. $x = T$
  2. **while** $x.right \neq$ NIL
  3. $\quad\quad x = x.right$
  4. **return** x

Successor and Predecessor node

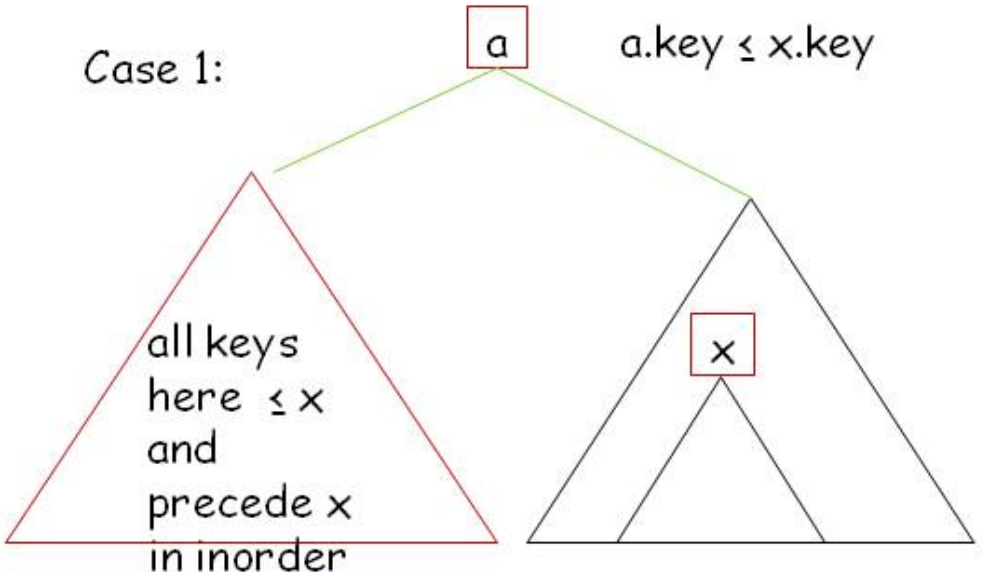
- Given a node in a binary search tree, its ***successor node*** is the node that follows it in the sorted order determined by an inorder traversal.
- Suppose all keys are *distinct*.
- Let x be a node in a binary search tree with key value say $x.key$.
- Then, its successor node is a node y with key $y.key > x.key$ and such that $y.key$ is the smallest key value in the tree that is larger than $x.key$.
- (Note that this holds only for binary search tree s with distinct key values).

Algorithm for Successor

- How can we find the successor? There are two cases to consider.
- The first is when x *has a right child*.
- In this case, the successor node is in *right subtree* of x. Therefore, it must be the node with the minimum key in the *right subtree* of x. Let us see *why*?
- Consider the smallest key k' in the right subtree of x.
- Consider any ancestor node a of x.

Successor: Case 1a

- Consider any ancestor node a of x.
- *Case 1a:* If x is in the right sub-tree of a, then $a.key \leq x.key$ and so a cannot be a successor and the keys in the left-subtree of a are at most $a.key$ and therefore cannot be successor of x.
- Another reason a cannot be a successor is that even if $a.key = x.key$, a appears before x in the inorder traversal of the tree.

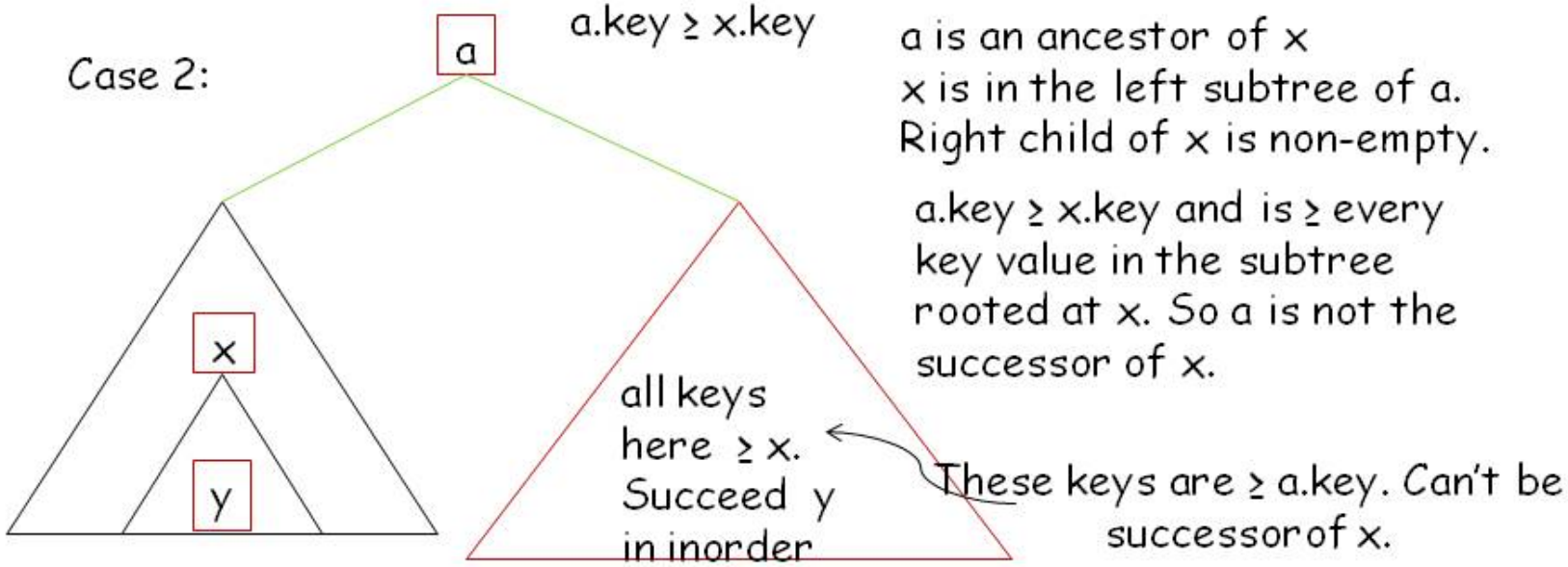


a is an ancestor of x
x is in the right subtree of a.
Right child of x is non-empty.

a appears before x in
inorder sequence.
So a cannot be a successor
of x.

Successor: Case 1b

- *Case 1b:* If x is in the left sub-tree of a, then all nodes in the subtree rooted at x are $\leq a.key$. Hence, a cannot be a successor of x, and neither can any node in the right subtree of a.
- Even if $a.key = x.key$, the nodes in the sub-tree of x will appear before a in the inorder traversal.

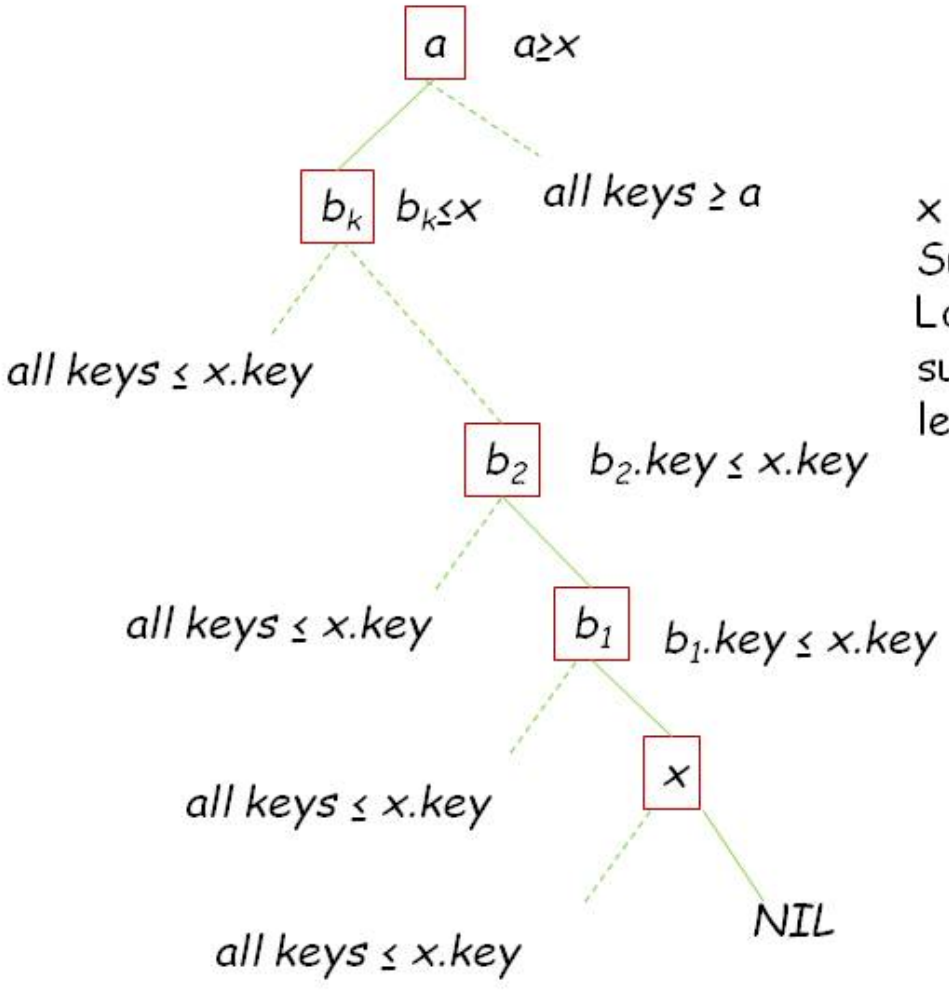

Case 2:
a
a.key ≥ x.key
x
y
all keys
here ≥ x.
Succeed y
in inorder
a is an ancestor of x
x is in the left subtree of a.
Right child of x is non-empty.
a.key ≥ x.key and is ≥ every
key value in the subtree
rooted at x. So a is not the
successor of x.
These keys are ≥ a.key. Can't be
successor of x.

Successor: Case 1

- Case 1: x has a right child.
- Rule 1: *if x has a right child, then the successor of x is the node with the minimum key in the right subtree of x*.

Successor: Case 2

- x does not have a right child.
- Consider the lowest ancestor a of x such that x lies in the left subtree of a.
- This node is the successor of x.
- Note: lowest ancestor satisfying a property means the ancestor of x that is closest to x and satisfies that property.



x has no right child.
Successor of x is the
Lowest ancestor a of x
such that x lies in the
left subtree of a.

Successor of x is a

Successor node: Pseudo-code

```
SUCCESSOR(T, x)
1.   if x.right ≠ NIL
2.         return FIND-MINIMUM(x)
3.   else
4.         y = x
5.         z = y.parent
6.         while z ≠ NIL  and  z.right == y
7.               y = z
8.               z = z.parent
9.         return z
```

Proof of Case 2

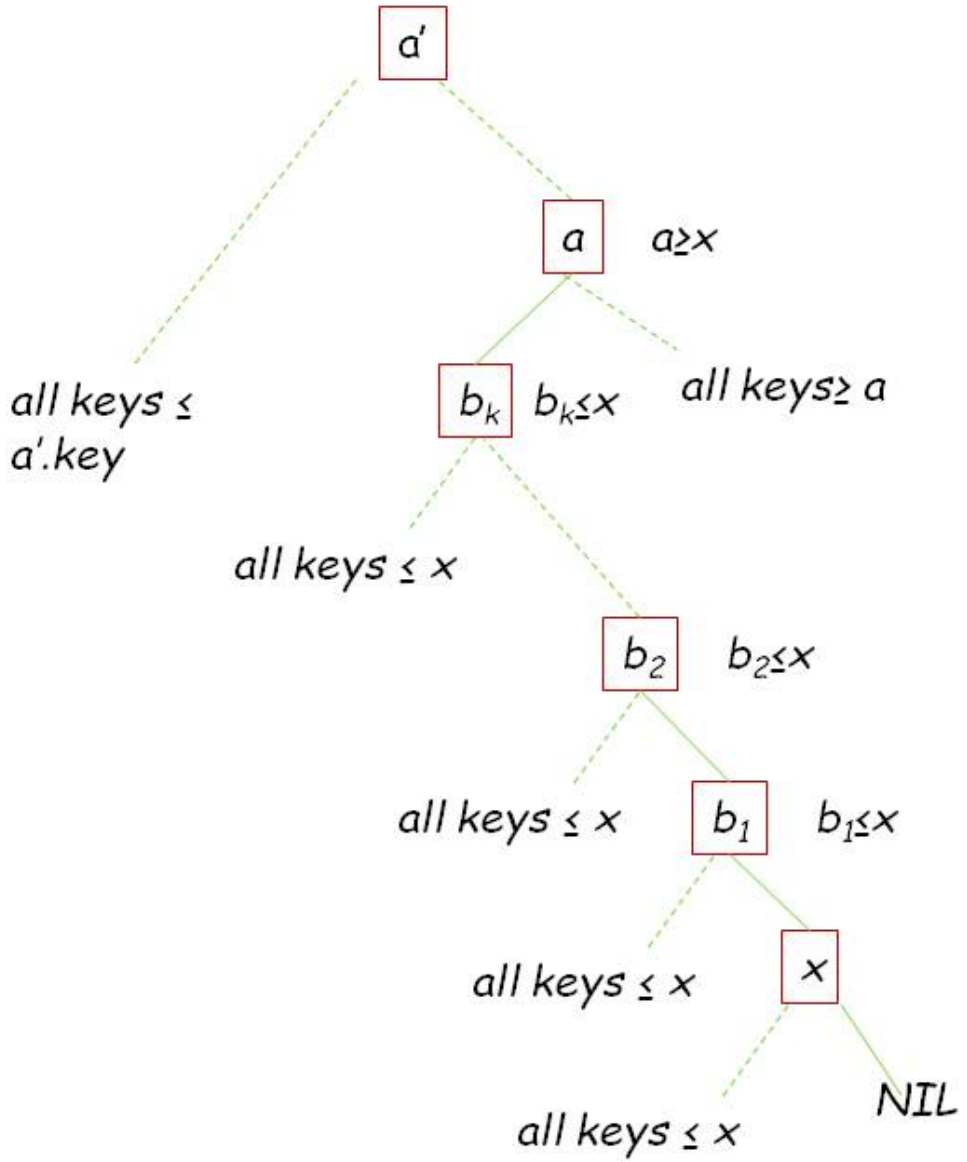
Let x be a node with no right child and let a be the lowest ancestor a of x such that x lies in the left subtree of a. Then, a is the successor node of x.

Proof:

- Let a be the lowest ancestor of x such that x lies in the left-subtree of a.
- Since x lies in the left-subtree of x, $x.key \leq a.key$.
- Further, in the inorder traversal of the tree, a appears after x.
- Any node b *strictly* between a and x has key **at most** $x.key$ and its left subtree has keys at most $b.key \leq x.key$.
- Hence none of these nodes or nodes in their left-subtree can be a successor of x.
- Also none of the nodes in the right subtree of a can be a successor of x since they have keys at least $a.key \geq x.key$.

Proof of Case 2

- Consider any ancestor a' of a.
- If a lies in the left sub-tree of a', then, $a'.key \geq a.key$.
- So neither a' nor any of the nodes in the right subtree of a' can be the successor node of x.
- If a lies in the right sub-tree of a', then x also lies in the right subtree of a' and so $x.key \geq a'.key$.
- Also, $x.key$ is no smaller than any element in the left-subtree of a'.
- Thus, a' or any of the nodes in the left-subtree of a' can be the successor node of x.



x has no right child. Successor of x is the Lowest ancestor a of x such that x lies in the left subtree of a.

Successor of x is a

Proof of Case 2: Concluded

- So for any ancestor a' of a such that a is in the left (respectively, right) subtree of a', then, a' or any node in the right (respectively, left) subtree of a' cannot be the successor x.
- Hence, the successor of x is a.

Predecessor

- **Predecessor**. The rule for finding predecessor is analogous to finding successor and can be reasoned similarly.
- Given a node x, its predecessor is found as follows.
  1. If x has a left-child, then, its predecessor is the node with the maximum key in its left-subtree.
  2. If x does not have a left-child, then its predecessor is the lowest ancestor a such that x is in the right subtree of a.

Predecessor: Pseudo-code

```
PREDECESSOR(T, x)
1.   if x.left ≠ NIL
2.         return FIND-MAXIMUM(x)
3.   else
4.         y = x
5.         z = y.parent
6.         while z ≠ NIL  and  z.left == y
7.               y = z
8.               z = z.parent
9.         return z
```