Design and Analysis of Algorithm

Advanced Data Structure
(Binary Search Tree)

Lecture -24-25

Overview

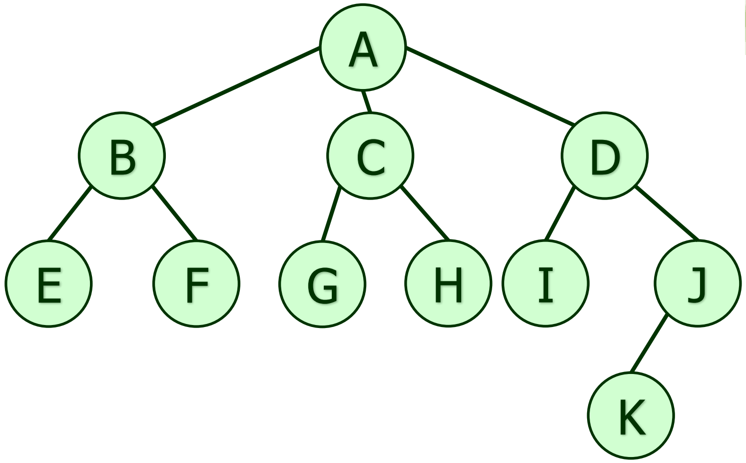
- Data structures that support many dynamic-set operations.
- Can be used as both a dictionary and as a priority queue.
- Basic operations take time proportional to the height of the tree.
    - For complete binary tree with n nodes: worst case $\Theta(\log n)$.
    - For linear chain of n nodes: worst case $\Theta(n)$.

Trees Terminology

- A tree is a data structure that represents data in a hierarchical manner.
- It associates every object to a node in the tree and maintains the parent/child relationships between those nodes.
- Each tree must have exactly one node, called the root, from which all nodes of the tree extend (and which has no parent of its own).
- The other end of the tree – the last level down — contains the leaf nodes of the tree.

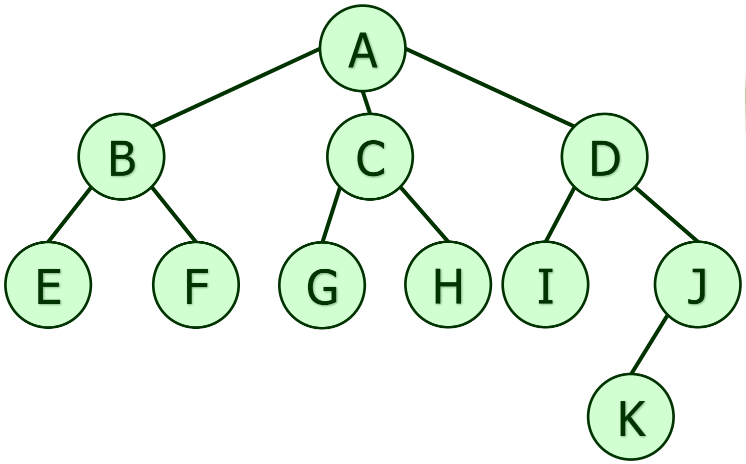
Trees Terminology

- The number of lines you pass through when you travel from the root until you reach a particular node is the depth of that node in the tree (node G in the figure above has a depth of 2).
- The height of the tree is the maximum depth of any node in the tree (the tree in given figure has a height of 3).



Trees Terminology

- The number of children emanating from a given node is referred to as its degree — for example, node A above has a degree of 3 and node J has a degree of 1.



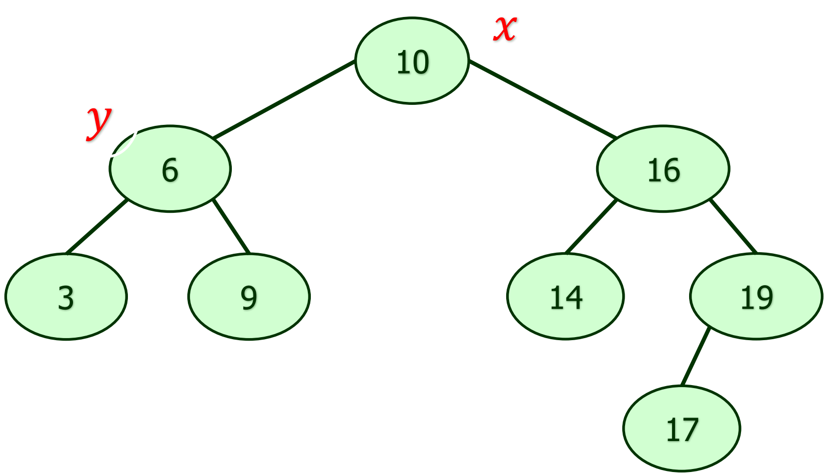
Binary Search Tree (BST)

Binary search trees are an important data structure for dynamic sets.

- Accomplish many dynamic-set operations in $O(h)$ time, where h = height of tree.
- A binary tree by a linked data structure in which each node is an object.
- *root*[T] points to the root of tree T.
- Each node contains the fields
  - *key* (and possibly other satellite data).
  - *left*: points to left child.
  - *right*: points to right child.
  - *p*: points to parent. $p[root[T]]$ = NIL.

Binary Search Tree (BST)

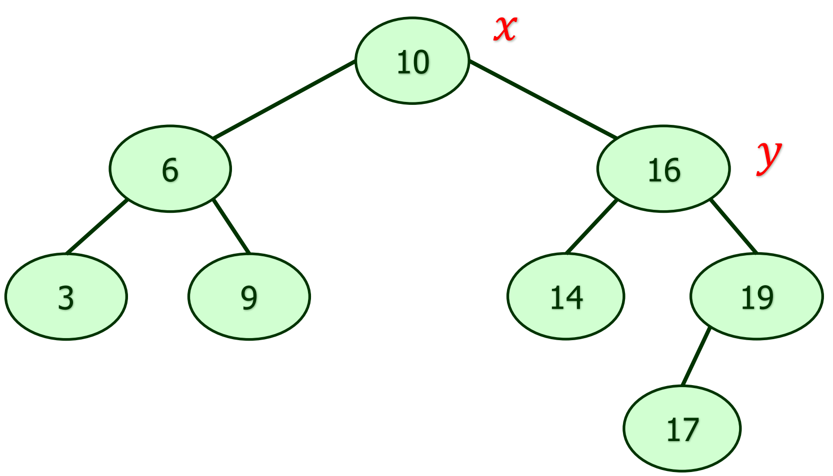
- Stored keys must satisfy the ***binary-search-tree property***.
  - If y is in left subtree of x, then key[y] ≤ key[x].
  - If y is in right subtree of x, then key[y] ≥ key[x].



(Figure: A BST on 8 nodes with height 3)

Binary Search Tree (BST)

- Stored keys must satisfy the ***binary-search-tree property***.
  - If y is in left subtree of x, then key[y] ≤ key[x].
  - If y is in right subtree of x, then key[y] ≥ key[x].



(Figure: A BST on 8 nodes with height 3)

Binary Search Tree (BST)

- Possible operations on BST
  - Traversing
  - Searching
  - Inserting
  - Deleting

Binary Search Tree (BST)

- Possible operations on BST
  - Traversing
  - Searching
  - Inserting
  - Deleting

Binary Search Tree (BST)

- Traversing

  The binary-search-tree property allows us to print keys in a binary search tree in order, recursively, using an algorithm called an in-order tree walk. Elements are printed in monotonically increasing order.

  How INORDER-TREE-TRAVERSAL works:

  - Check to make sure that x is not NIL.
  - Recursively, print the keys of the nodes in $x's\ left$ subtree.
  - Print $x's\ key$.
  - Recursively, print the keys of the nodes in $x's\ right$ subtree.

Binary Search Tree (BST)

- Traversing
- A common BST traversing algorithm (i.e. In-order tree traversal) is given below for easy understanding:

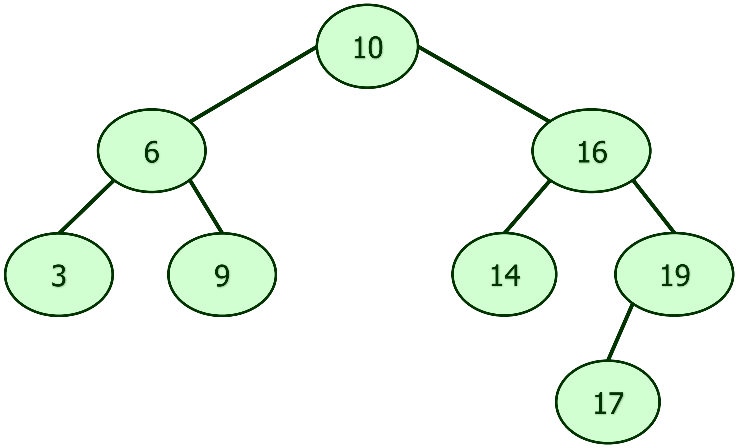
$Inorder - Tree(x)$

$if\ x \neq\ NIL$

$then\ Inorder - Tree(left[x])$

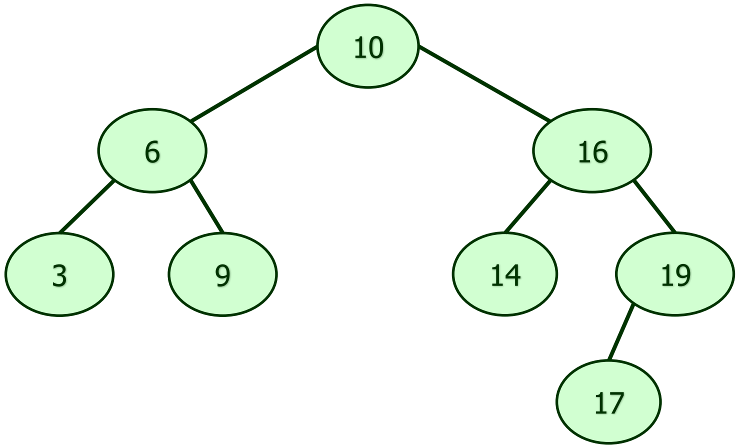
$print\ key[x]$

$Inorder - Tree(right[x])$



Binary Search Tree (BST)

- Traversing
  - Example: The in-order tree traversal on the example below, getting the output $3 \rightarrow 6 \rightarrow 9 \rightarrow 10 \rightarrow 14 \rightarrow 16 \rightarrow 17 \rightarrow 19$
  - Correctness: Follows by induction directly from the binary-search-tree property.
  - Time: Intuitively, the walk takes $\Theta(n)$ time for a tree with n nodes, because we visit and print each node once.



Binary Search Tree (BST)

- Possible operations on BST
  - Traversing
  - Searching
  - Inserting
  - Deleting

Binary Search Tree (BST)

- Searching
  - The search procedure returns a pointer to the node with key ‘k’ if the key exists, otherwise return ‘NULL’ .

$\boldsymbol{Tree - Search(x, k)}$

$if\ x\ =\ NIL\ or\ k\ =\ key[x]$

$\quad then\ return\ x$

$if\ k\ <\ key[x]$

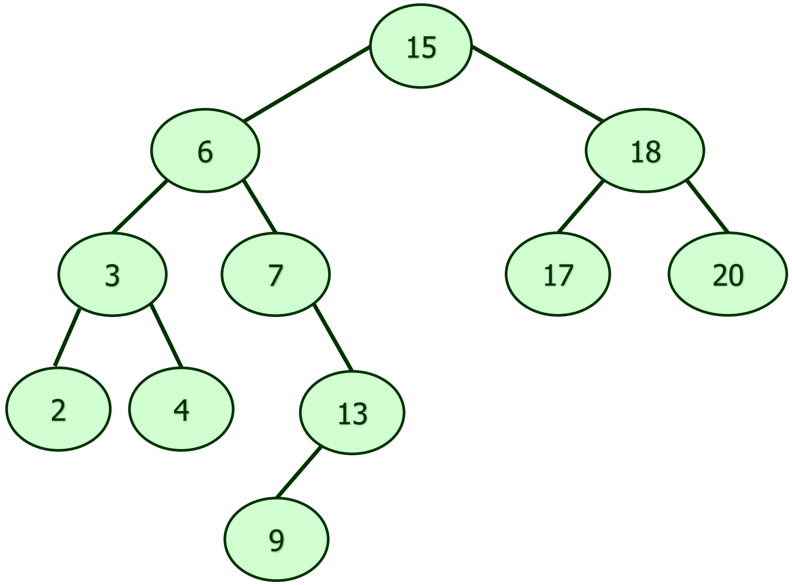
$\quad then\ return\ Tree - Search(left[x], k)$

$\quad else\ return\ Tree - Search(right[x], k)$

$Initial\ call\ is\ Tree - Search(root[T\ ], k).$

Binary Search Tree (BST)

- Searching
  - How to search key 13 on the following Tree.



Binary Search Tree (BST)

- Searching
  - How to search key 13 on the following Tree.

$\boldsymbol{Tree - Search(x, k)}$

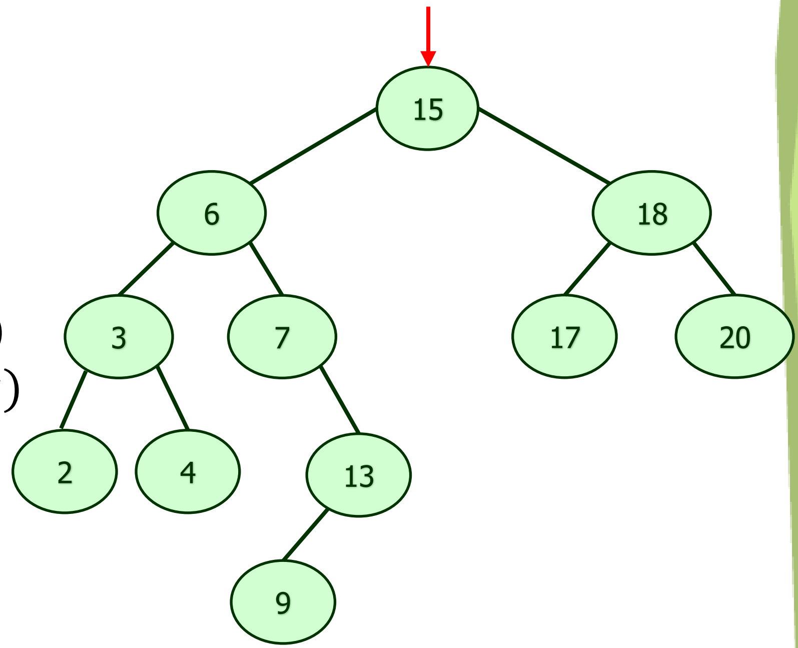
$if\ x\ =\ NIL\ or\ k\ =\ key[x]$

$\quad then\ return\ x$

$if\ k\ <\ key[x]$

$\quad then\ return\ Tree - Search(left[x], k)$

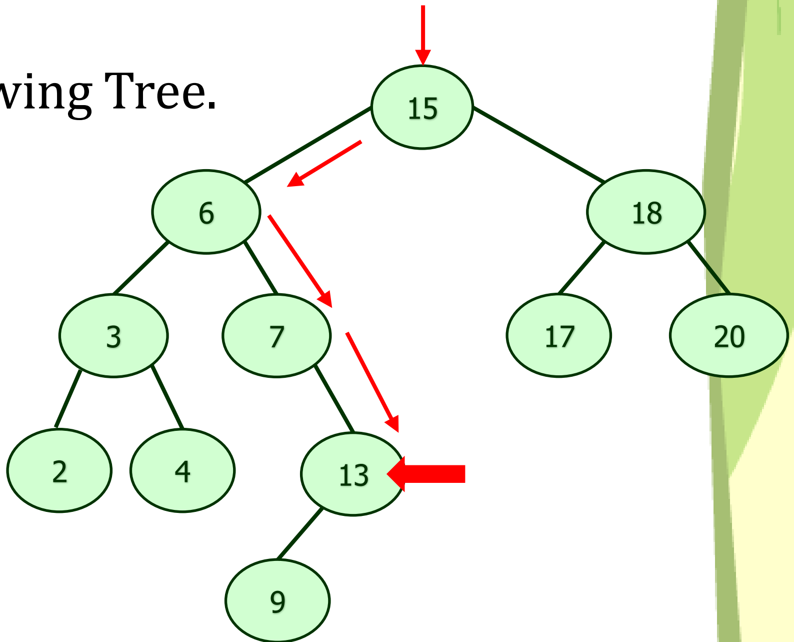
$\quad else\ return\ Tree - Search(right[x], k)$



$Initial\ call\ is\ Tree - Search(root[T\ ], k).$

Binary Search Tree (BST)

- Searching
  - How to search key 13 on the following Tree.



$\boldsymbol{Tree - Search(x, k)}$

$if\ x = NIL\ or\ k = key[x]$

$\quad then\ return\ x$

$if\ k < key[x]$

$\quad then\ return\ Tree - Search(left[x], k)$

$\quad else\ return\ Tree - Search(right[x], k)$

$Initial\ call\ is\ Tree - Search(root[T\ ], k).$

$15 \rightarrow 6 \rightarrow 7 \rightarrow 13$

Time: The algorithm is recursive and visit nodes on a downward path from the root. Thus, running time is $O(h)$, where h is the height of the tree.

Binary Search Tree (BST)

- Searching (**Minimum and Maximum**)

  The binary-search-tree property guarantees that

  - the minimum key of a binary search tree is located at the leftmost node, and
  - the maximum key of a binary search tree is located at the rightmost node.

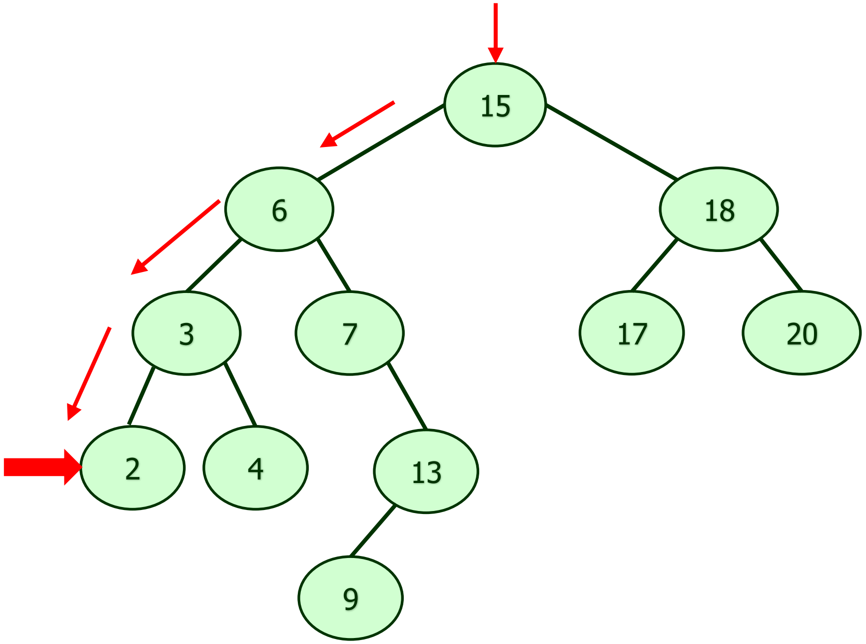
  Traverse the appropriate pointers (left or right) until NIL is reached.

Binary Search Tree (BST)

- Searching
  - Find minimum and maximum node in BST.

The following procedure returns a pointer to the minimum element in the subtree rooted at a given node x, which we assume to be not NIL.

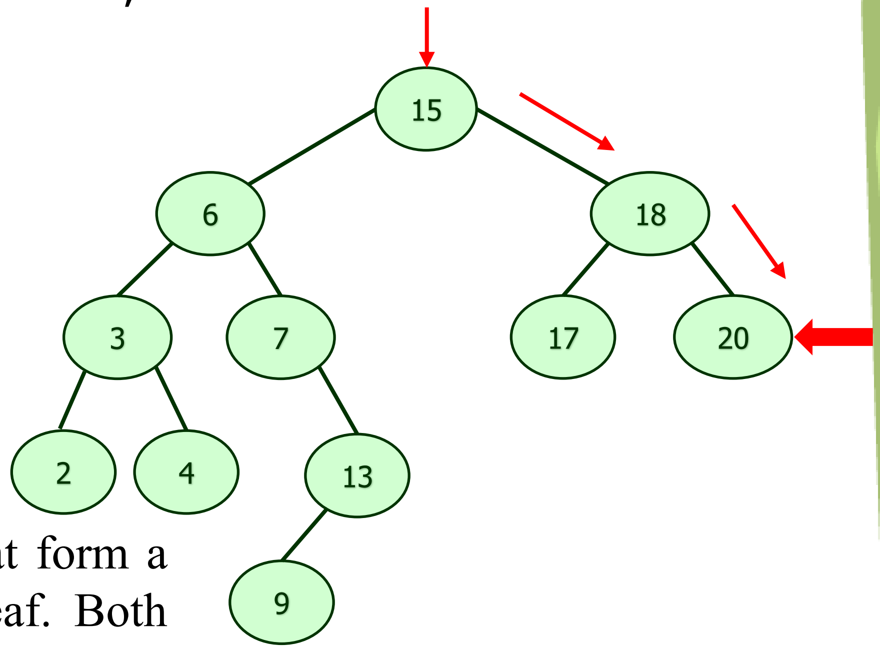
$Tree - Minimum(x)$
$while\ left[x] \neq\ NIL$
$\quad do\ x\ \leftarrow\ left[x]$
$return\ x$



Binary Search Tree (BST)

- Searching
  - Find minimum and maximum node in BST.

The following procedure returns a pointer to the maximum element in the subtree rooted at a given node x, which we assume to be not NULL



$$\begin{aligned}
&Tree - Maximum(x)\\
&while\ right[x] \neq\ NIL\\
&\quad do\ x \leftarrow\ right[x]\\
&return\ x
\end{aligned}$$

Time: Both procedures visit nodes that form a downward path from the root to a leaf. Both procedures run in *O(h)* time, where *h* is the height of the tree.

Binary Search Tree (BST)

- Searching (***Successor and predecessor)***
  - Assuming that all keys are distinct, the successor of a $node\ x$ is the $node\ y$ such that $key[y]$ is the $smallest\ key\ >\ key[x]$.
  - If x has the largest key in the binary search tree, then we say that $x's$ successor is NIL.

There are two cases:

1. If $node\ x$ has a non-empty right subtree, then $x's$ successor is the minimum in $x's$ right subtree.

2. If $node\ x$ has an empty right subtree, notice that:

   - As long as we move to the left up the tree (move up through right children), we are visiting smaller keys.
   - $x's$ successor y is the node that x is the predecessor of (x is the maximum in $y's$ left subtree).

Binary Search Tree (BST)

- Searching (***Successor and predecessor***)
  - Find successor node in BST.

Let us find the successor() with the help of a tree example.

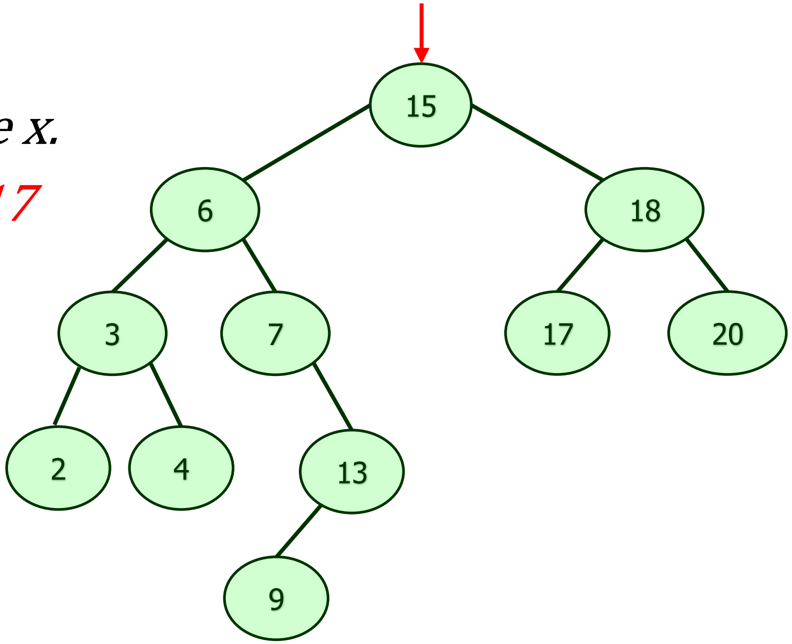
Successor:

The next increased value of node x.

i.e. The successor of node 15 is 17

and

The successor of node 13 is 15



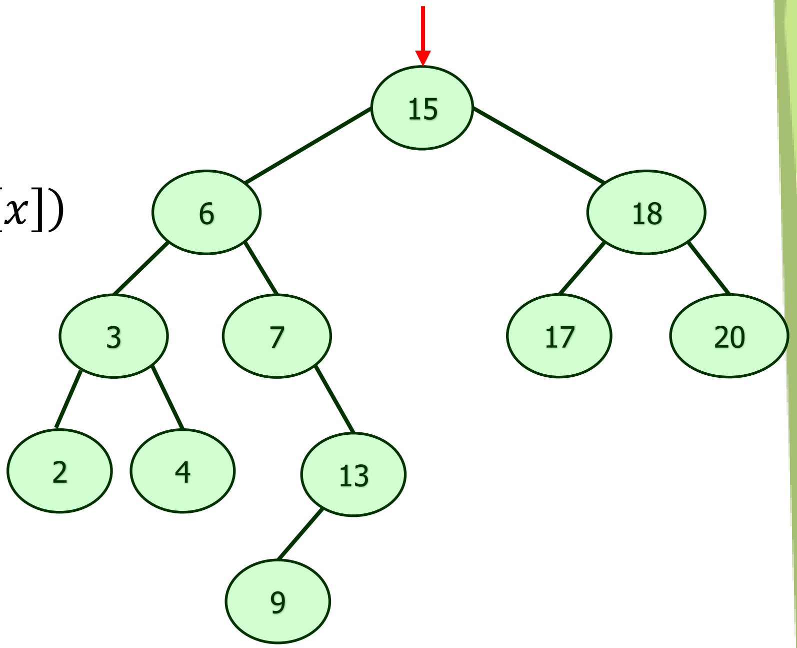
Binary Search Tree (BST)

- Searching (*Successor and predecessor)*
  - Find successor node in BST.

In BST, if all the keys are distinct, then the successor of a node x is the node with the smallest key greater than key[x].

The following procedure returns the successor of a node x in BST.

$\boldsymbol{Tree\_successor(x)}$
$if\ right[x] \neq NIL$
    $then\ return\ Tree - Minimum(right[x])$
$y \leftarrow p[x]$
$while\ y \neq NIL\ and\ x = right[y]$
    $do\ x \leftarrow y$
    $y \leftarrow p[y]$
$return\ y$

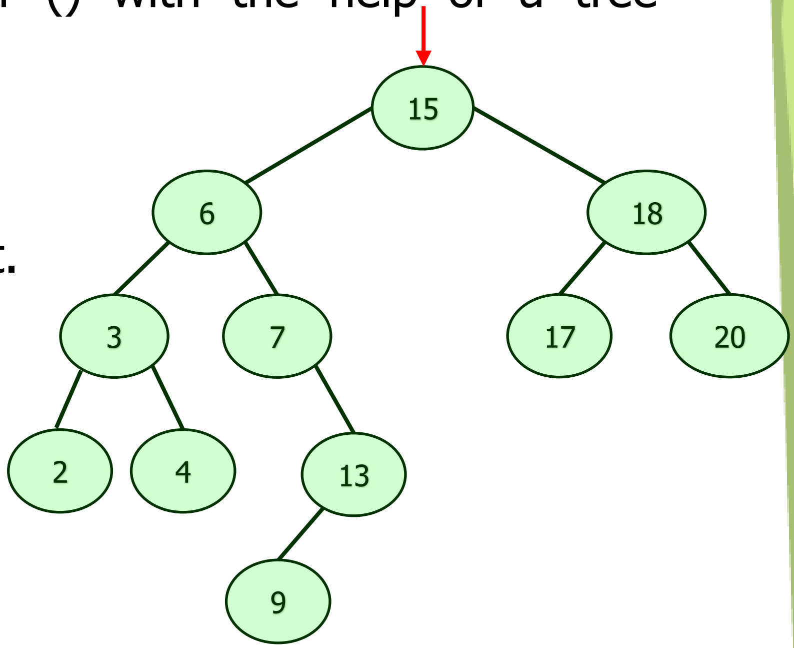


Binary Search Tree (BST)

- Searching (*Successor and predecessor)*
  - Find predecessor node in BST.

In BST, if all the keys are distinct, then the in-order predecessor of a node x is the previous node in in-order traversal of it.

Let us find the in-order predecessor () with the help of a tree example.



For Example :

In-order predecessor of 2 do not exist.

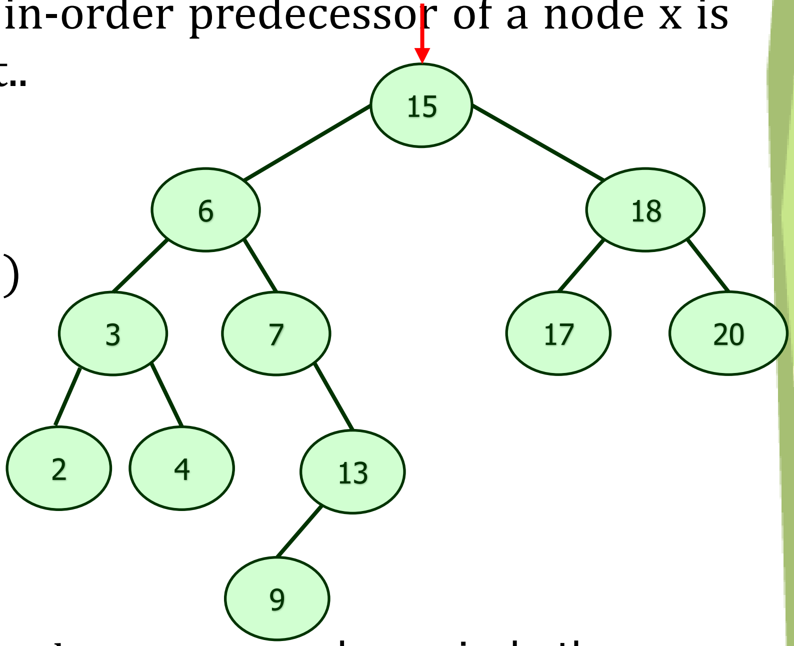
In-order predecessor of 15 is 13

In-order predecessor of 18 is 17

Binary Search Tree (BST)

- Searching (*Successor and predecessor)*
  - Find predecessor node in BST.

In BST, if all the keys are distinct, then the in-order predecessor of a node x is the previous node in in-order traversal of it..



$\boldsymbol{Tree\_predecessor}(\boldsymbol{x})$
$if\ left[x] \neq\ NIL$
$\quad then\ return\ Tree - Maximum(left[x])$
$y\ \leftarrow\ p[x]$
$while\ y \neq NIL\ and\ x\ = left[y]$
$\quad do\ x\ \leftarrow\ y$
$\quad y\ \leftarrow\ p[y]$
$return\ y$

Time: For both the Tree_successor and Tree_predecessor procedures, in both cases, we visit nodes on a path down the tree or up the tree. Thus, running time is O(h), where h is the height of the tree.

Binary Search Tree (BST)

- Possible operations on BST
  - Traversing
  - Searching
  - Inserting
  - Deleting

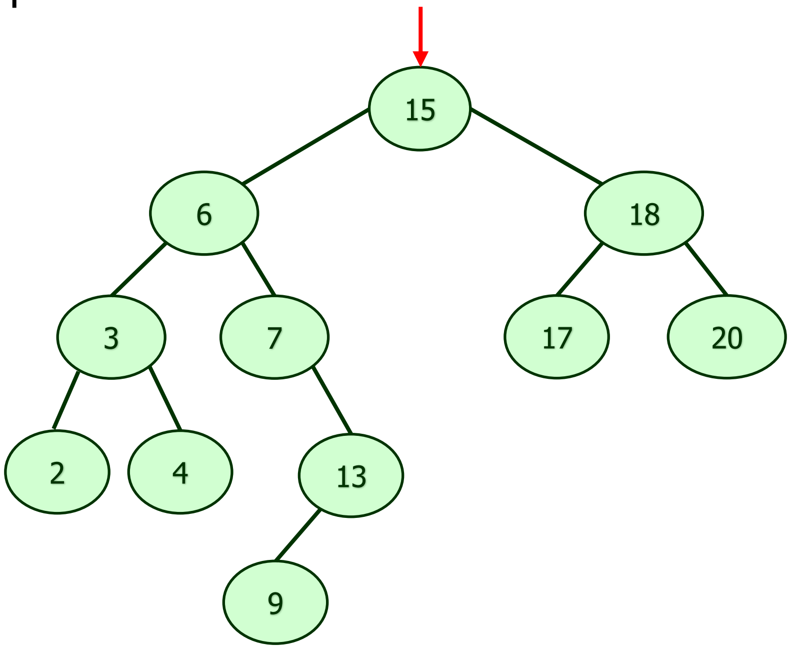
Binary Search Tree (BST)

- Insertion
  - To insert value v into the binary search tree, the procedure is given node z, with $key[z] = v,\ left[z] = NIL, and\ right[z] = NIL$.
  - Beginning at root of the tree, trace a downward path, maintaining two pointers.
    - Pointer x: traces the downward path.
    - Pointer y: "trailing pointer" to keep track of parent of x.
  - Traverse the tree downward by comparing the value of node at x with v, and move to the left or right child accordingly.
  - When x is NIL, it is at the correct position for node z.
  - Compare $z's$ value with $y's$ value, and insert z at either $y's$ left or right, appropriately.

Binary Search Tree (BST)

- Insertion

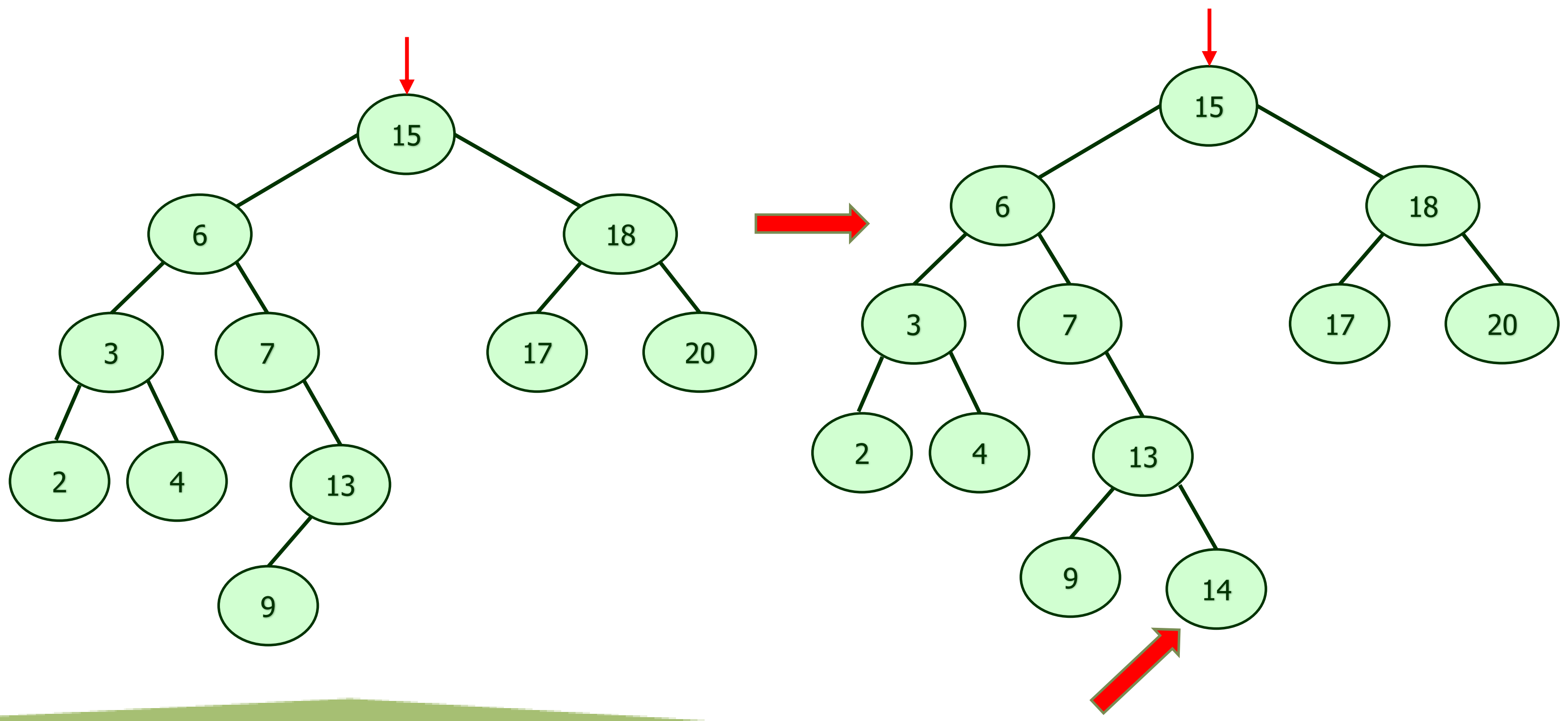
Example: insert 14



Binary Search Tree (BST)

- Insertion

Example: insert 14



Binary Search Tree (BST)

- Insertion

$\boldsymbol{Tree - Insert}(\boldsymbol{T}, \boldsymbol{z})$

$y \leftarrow NIL$

$x \leftarrow root[T]$

$while\ x \neq NIL$

$\quad do\ y \leftarrow x$

$\quad if\ key[z] < key[x]$

$\qquad then\ x \leftarrow left[x]$

$\quad else$

$\qquad x \leftarrow right[x]$

$p[z] \leftarrow y$

$if\ y = NIL$

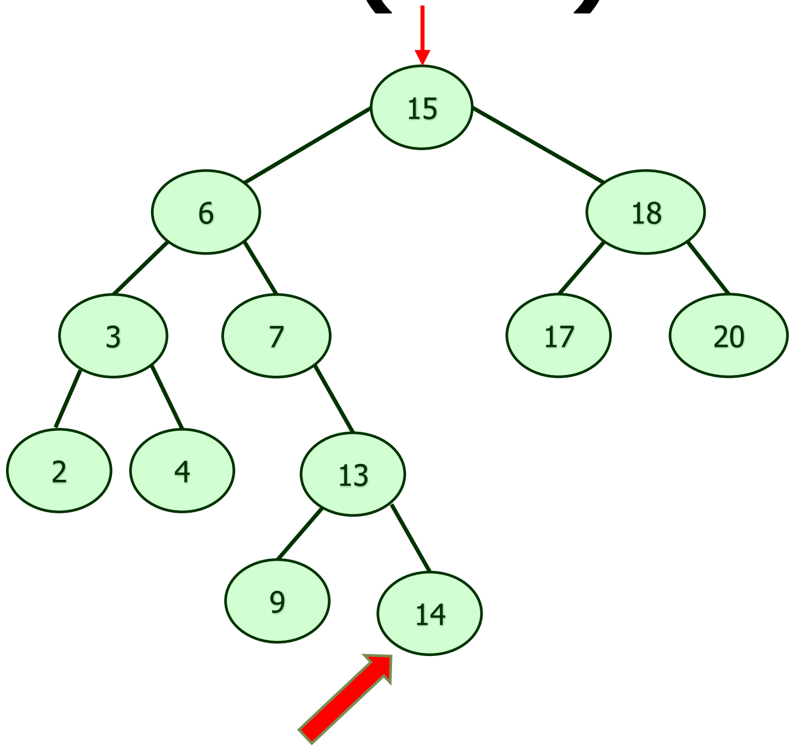
$\quad then\ root[T] \leftarrow z \rhd \quad Tree\ T\ was\ empty$

$else\ if\ key[z] < key[y]$

$\qquad then\ left[y] \leftarrow z$

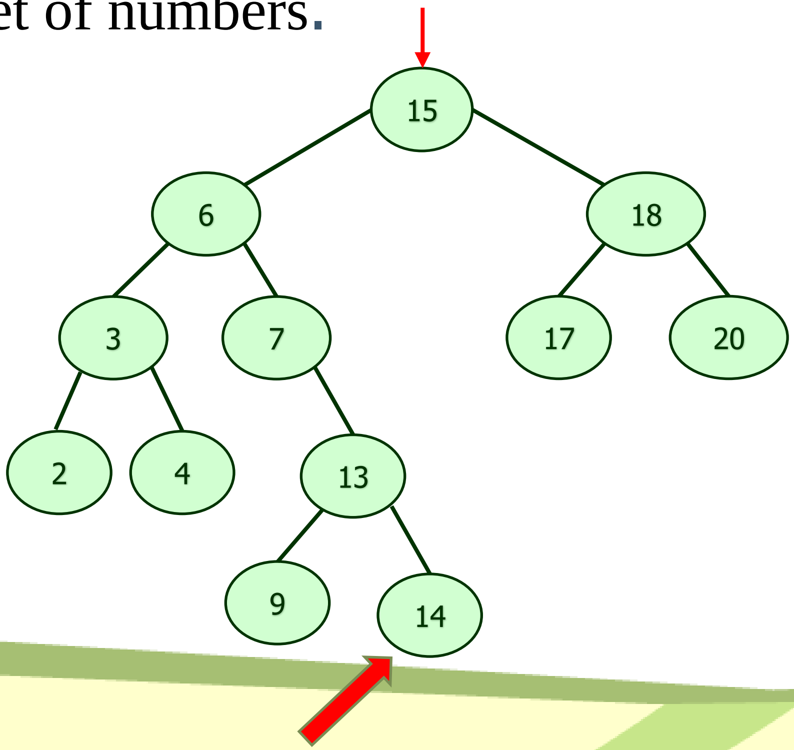
$\quad else$

$\qquad right[y] \leftarrow z$



Binary Search Tree (BST)

- Insertion (Time complexity)
  - Same as Tree-Search() . On a tree of height h, procedure takes O(h) time.
  - Tree-Insert() can be used with Inorder-Tree() to sort a given set of numbers.



Binary Search Tree (BST)

- Possible operations on BST
  - Traversing
  - Searching
  - Inserting
  - Deleting

Binary Search Tree (BST)

Deletion

TREE-DELETE is broken into three cases.

Case 1: node *z* has no children.

- Delete node *z* by making the parent of *z* point to NULL, instead of to *z*.

Case 2: node *z* has one child.

- Delete *z* by making the parent of *z* point to *z*.s child, instead of to *z*.

Case 3: *z* has two children.

- node *z*'s successor *y* has either no children or one child. (*y* is the minimum node.with no left child.in *z*.s right subtree.)
- Delete *y* from the tree (via Case 1 or 2).
- Replace *z*.s key and satellite data with *y*.s.

Binary Search Tree (BST)

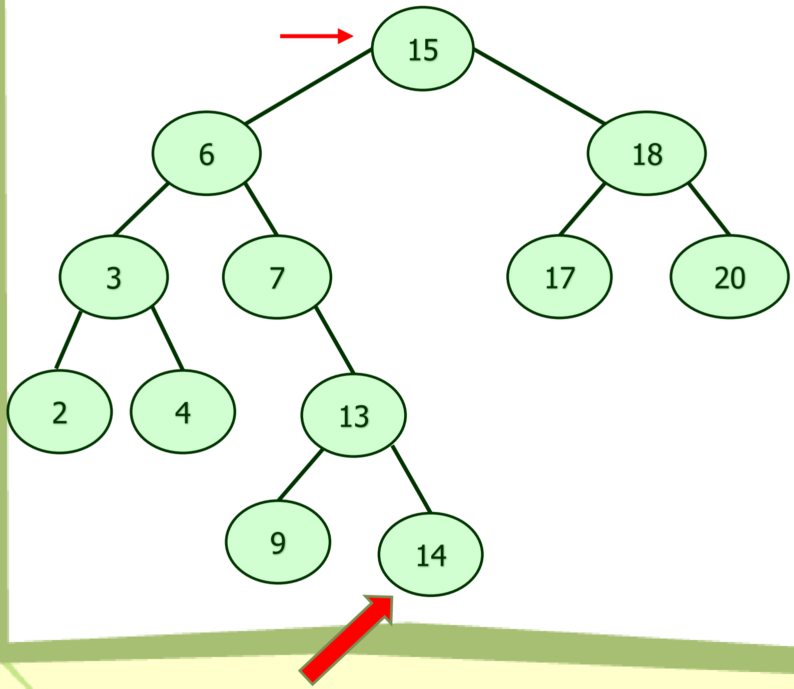
Deletion

TREE-DELETE is broken into three cases.

Case 1: node *z* has no children.

– Delete node *z* by making the parent of *z* point to NULL, instead of to *z*.

Example: Delete 14



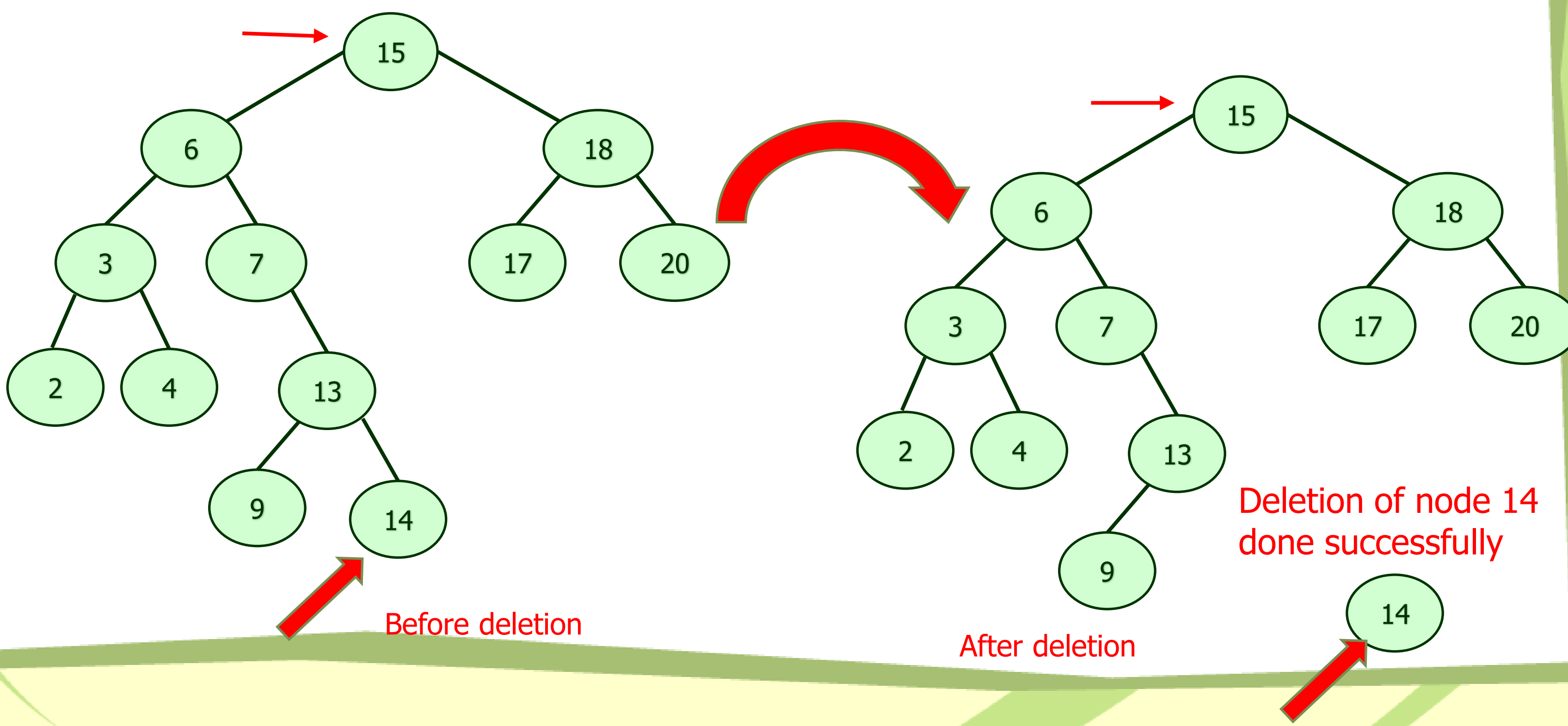
Binary Search Tree (BST)

Deletion

Case 1: node *z* has no children.

- Delete node *z* by making the parent of *z* point to NULL, instead of to *z*.

Example: Delete 14



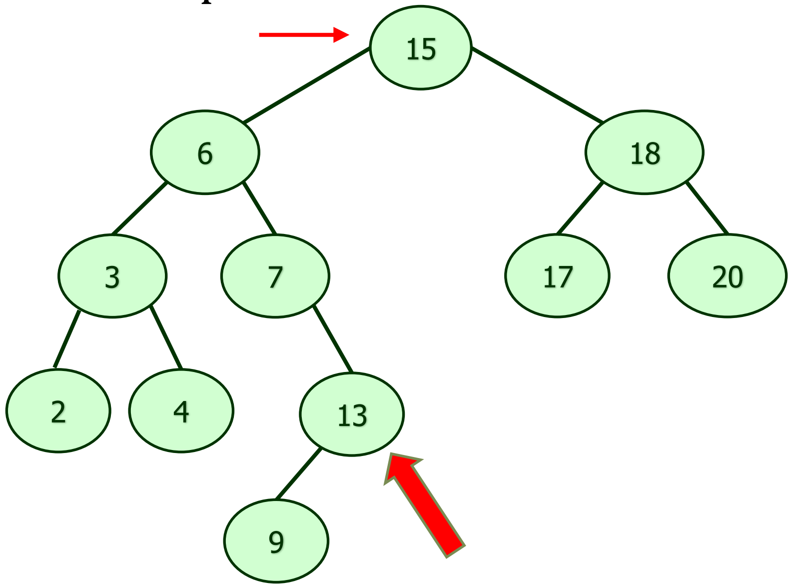
Binary Search Tree (BST)

Deletion

Case 2: node *z* has one child.

- Delete *z* by making the parent of *z* point to *z*.s child, instead of to *z*.

Example: Delete node 13



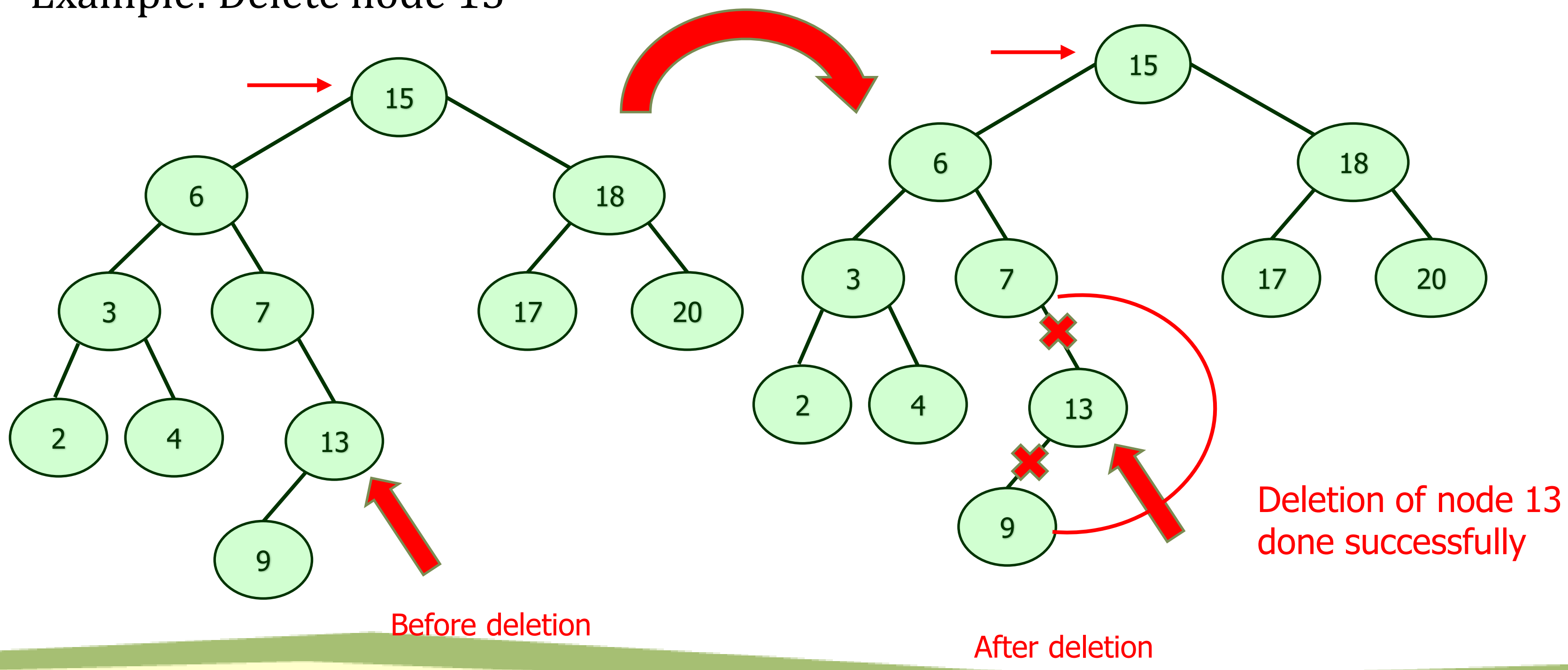
Binary Search Tree (BST)

Deletion

Case 2: node *z* has one child.

- Delete *z* by making the parent of *z* point to *z*.s child, instead of to *z*.

Example: Delete node 13



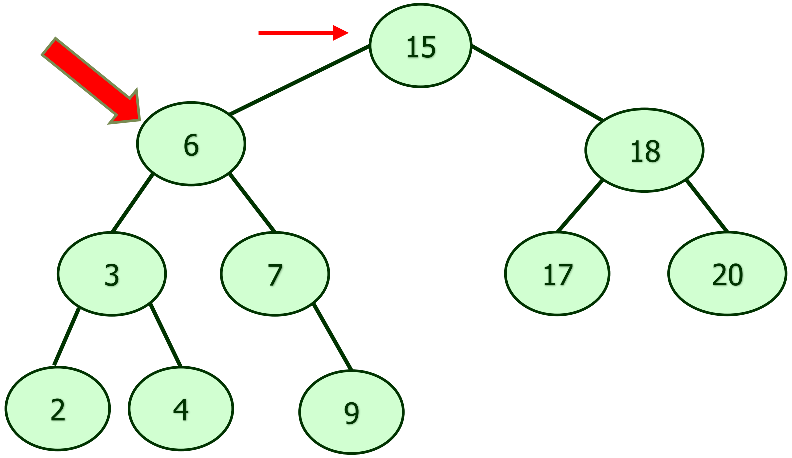
Binary Search Tree (BST)

Deletion

Case 3: node z has two children.

- Find node z's successor y . y has either no children or one child. (y is the minimum node with no left child.in $z's$ right subtree.)
- Delete y from the tree (via Case 1 or 2).
- Replace $z \leftarrow key$ and satellite data with $y \leftarrow key$.

Example : Delete node 6



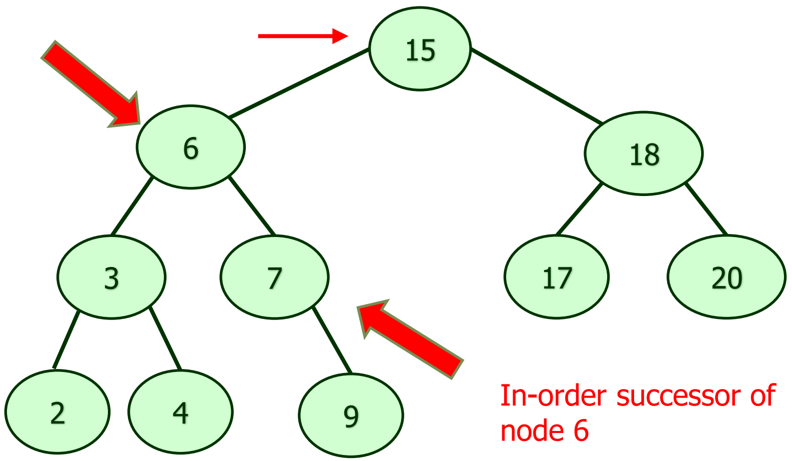
Binary Search Tree (BST)

Deletion

Case 3: node z has two children.

- Find node z's successor y . y has either no children or one child. (y is the minimum node with no left child.in $z's$ right subtree.)
- Delete y from the tree (via Case 1 or 2).
- Replace $z \leftarrow key$ and satellite data with $y \leftarrow key$.

Example : Delete node 6



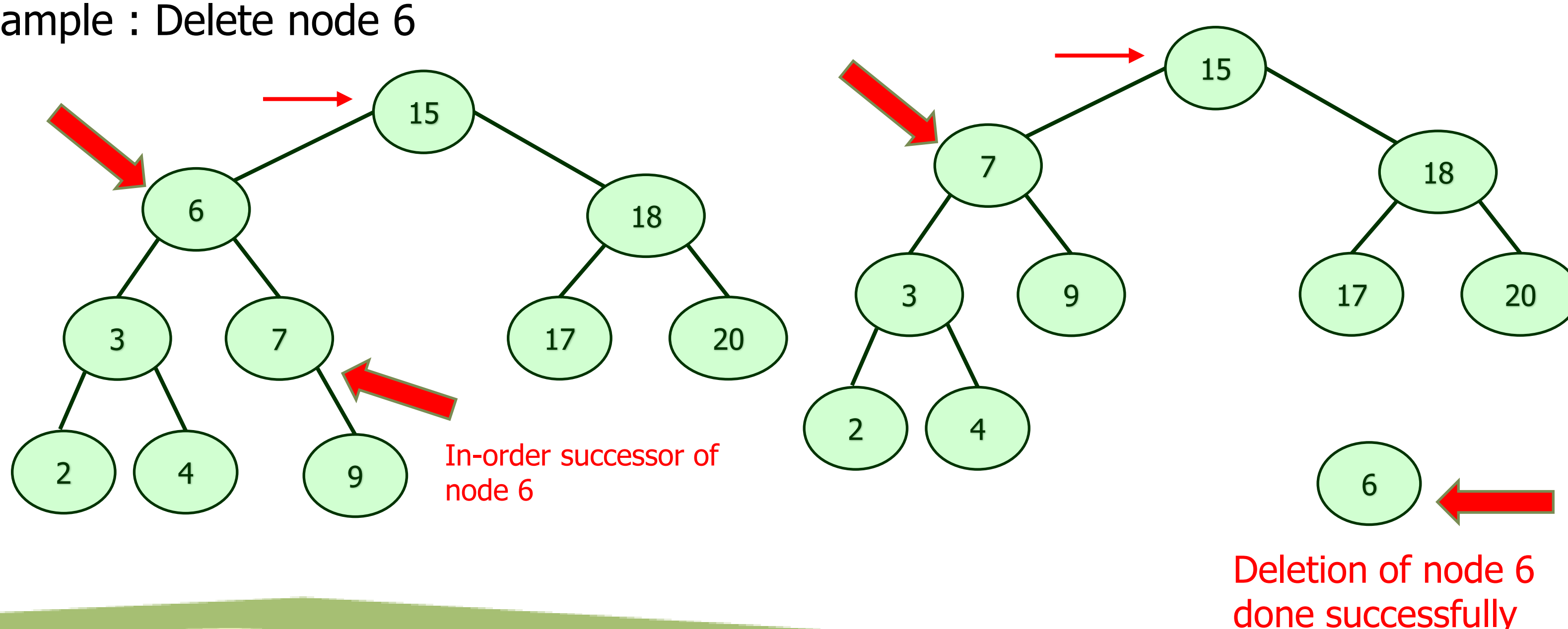
Binary Search Tree (BST)

Deletion

Case 3: node z has two children.

- Find node z's successor y . y has either no children or one child. (y is the minimum node with no left child.in $z's$ right subtree.)
- Delete y from the tree (via Case 1 or 2).
- Replace $z \leftarrow key$ and satellite data with $y \leftarrow key$.

Example : Delete node 6



Binary Search Tree (BST)

Deletion:

$\boldsymbol{Tree - Delete}(\boldsymbol{T}, \boldsymbol{z})$

// $Determine\ which\ node\ y\ to\ splice\ out:\ either\ z\ or\ z's\ successor.$

$\mathbf{if}\ left[z]\ =\ NIL\ or\ right[z]\ =\ NIL$

$\quad \mathbf{then}\ y \leftarrow z$

$\quad \mathbf{else}\ y \leftarrow Tree - Successor(z)$

// $x\ is\ set\ to\ a\ non-NIL\ child\ of\ y, or\ to\ NIL\ if\ y\ has\ no\ children.$

$\mathbf{if}\ left[y] \neq NIL$

$\quad \mathbf{then}\ x \leftarrow left[y]$

$\quad \mathbf{else}\ x \leftarrow right[y]$

// $y\ is\ removed\ from\ the\ tree\ by\ manipulating\ pointers\ of\ p[y]\ and\ x.$

$\mathbf{if}\ x \neq NIL$

$\quad \mathbf{then}\ p[x] \leftarrow p[y]$

$\mathbf{if}\ p[y]\ =\ NIL$

$\quad \mathbf{then}\ root[T\ ] \leftarrow x$

$\quad \mathbf{else\ if}\ y\ =\ left[p[y]]$

$\qquad\quad \mathbf{then}\ left[p[y]] \leftarrow x$

$\qquad \mathbf{else}\ right[p[y]] \leftarrow x$

// $If\ it\ was\ z's\ successor\ that\ was\ spliced\ out, copy\ its\ data\ into\ z.$

$\mathbf{if}\ y\ =\ z$

$\quad \mathbf{then}\ key[z] \leftarrow key[y]$

$\qquad copy\ y.s\ satellite\ data\ into\ z$

$\mathbf{return}\ y$

Binary Search Tree (BST)

Deletion:

$\boldsymbol{Tree - Delete(T, z)}$

$//\ Determine\ which\ node\ y\ to\ splice\ out:\ either\ z\ or\ z's\ successor.$

$\mathbf{if}\ left[z] = NIL\ or\ right[z] = NIL$

$\quad \mathbf{then}\ y \leftarrow z$

$\quad \mathbf{else}\ y \leftarrow Tree - Successor(z)$

$//\ x\ is\ set\ to\ a\ non - NIL\ child\ of\ y, or\ to\ NIL\ if\ y\ has\ no\ children.$

$\mathbf{if}\ left[y] \neq NIL$

$\quad \mathbf{then}\ x \leftarrow left[y]$

$\quad \mathbf{else}\ x \leftarrow right[y]$

$//\ y\ is\ removed\ from\ the\ tree\ by\ manipulating\ pointers\ of\ p[y]\ and\ x.$

$\mathbf{if}\ x \neq NIL$

$\quad \mathbf{then}\ p[x] \leftarrow p[y]$

$\mathbf{if}\ p[y] = NIL$

$\quad \mathbf{then}\ root[T] \leftarrow x$

$\quad \mathbf{else\ if}\ y = left[p[y]]$

$\qquad \mathbf{then}\ left[p[y]] \leftarrow x$

$\qquad \mathbf{else}\ right[p[y]] \leftarrow x$

$//\ If\ it\ was\ z's\ successor\ that\ was\ spliced\ out, copy\ its\ data\ into\ z.$

$\mathbf{if}\ y = z$

$\quad \mathbf{then}\ key[z] \leftarrow key[y]$

$\qquad copy\ y.s\ satellite\ data\ into\ z$

$\mathbf{return}\ y$

Time: *O(h)*, on a tree of height *h*.

Thank u