Binary Search Trees

Dr. Chung-Wen Albert Tsao

Binary Search Trees

- Data structures that can support dynamic set operations.
  - Search, Minimum, **Maximum**, Predecessor, Successor, **Insert**, and **Delete**.
- Can be used to build
  - Dictionaries.
  - Priority Queues.
    - How about heap?
- Basic operations take time proportional to the height of the tree – $O(h)$.

Binary Search Trees

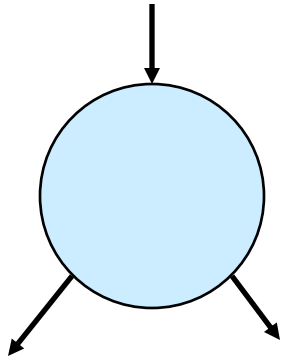
- Data structures that can support dynamic set operations.
  - Search, Minimum, **Maximum**, Predecessor, Successor, **Insert**, and **Delete**.
- Can be used to build
  - Dictionaries.
  - Priority Queues.
    - How about heap? (Cannot do Search efficiently)
- Basic operations take time proportional to the height of the tree – $O(h)$.

When to use BST or Heap?

- Heap is better at findMin/findMax (O(1)), while
- BST is good at searches (O(*lg* N)).
- Insert is O(*lg* N) for both structures.
- If you only care about findMin/findMax (e.g. priority-related), go with heap (simple implementation)
- If you want everything sorted, go with BST
  - more complicate implementation
  - tree height is NOT necessarily balanced O(lg N)

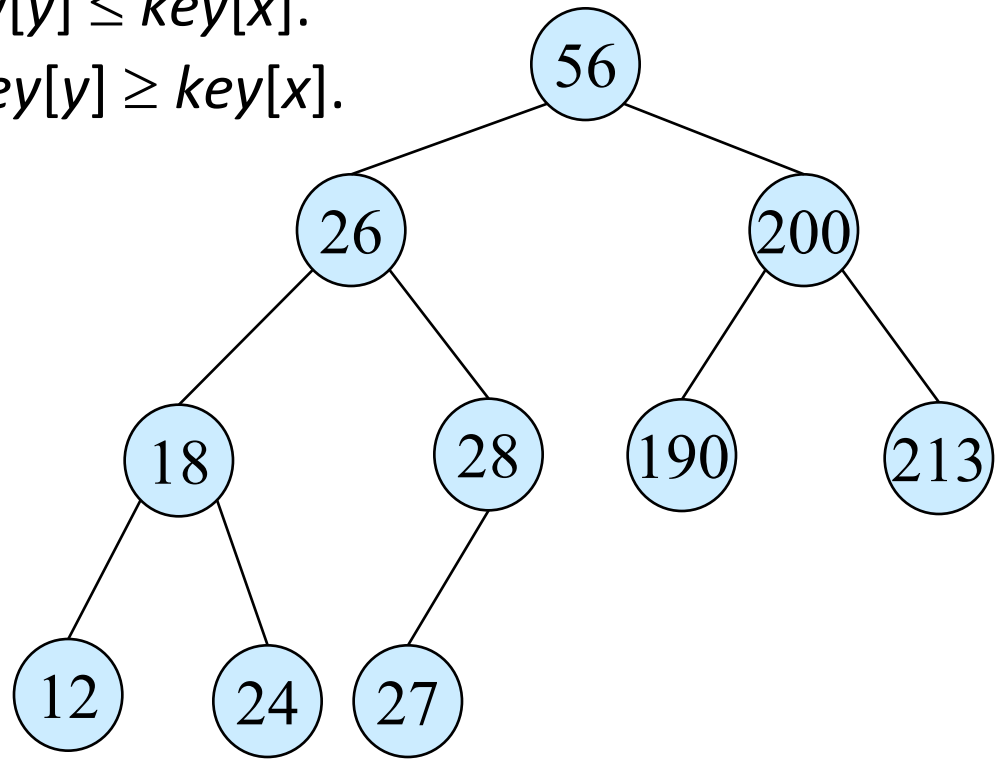
BST – Representation

- Represented by a linked data structure of nodes.
- *root*(*T*) points to the root of tree *T*.
- Each node contains fields:
  - *Key*
  - *left* – pointer to left child: root of left subtree (maybe nil)
  - *right* – pointer to right child : root of right subtree. (maybe nil)
  - *p* – pointer to parent. *p*[*root*[T]] = NIL
  - Satellite data



Binary **Search** Tree (BST) Property

- Stored keys must satisfy the *binary search tree* property.
  - $\forall$ *y* in left subtree of *x*, then *key*[*y*] $\leq$ *key*[*x*].
  - $\forall$ *y* in right subtree of *x*, then *key*[*y*] $\geq$ *key*[*x*].



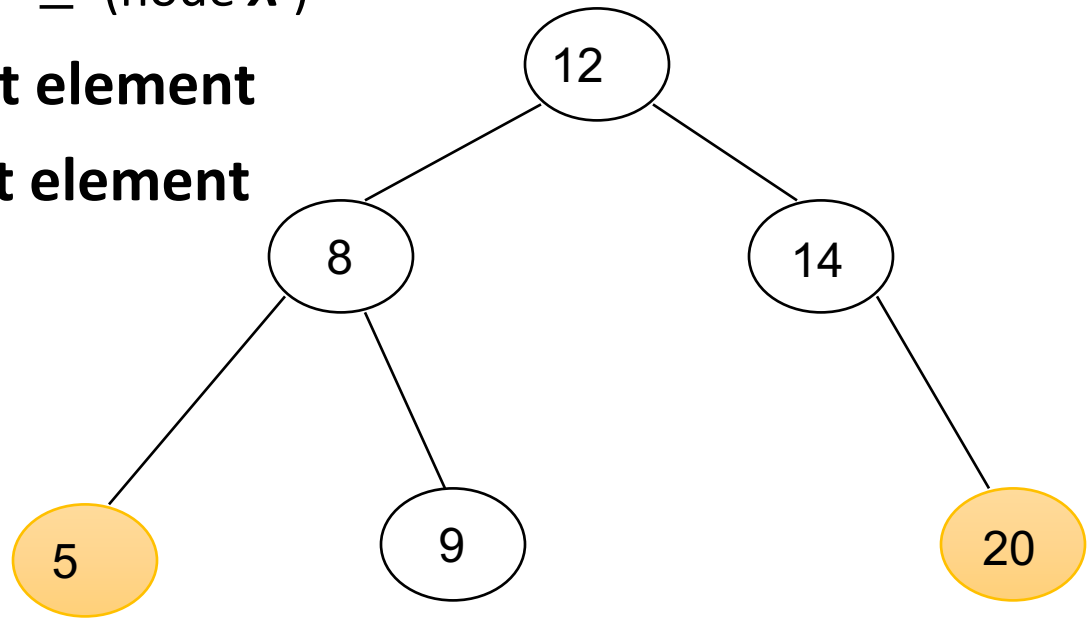
What else can we say?

$left(x) \leq x \leq right(x)$

- (All elements to the left of a node ***X)*** $\leq$ (node ***X***)
- (All elements to the right of a node ***X)*** $\geq$ (node ***X***)

- **The smallest element is the left-most element**
- **The largest element is the right-most element**

$left(x) \leq x \leq right(x)$



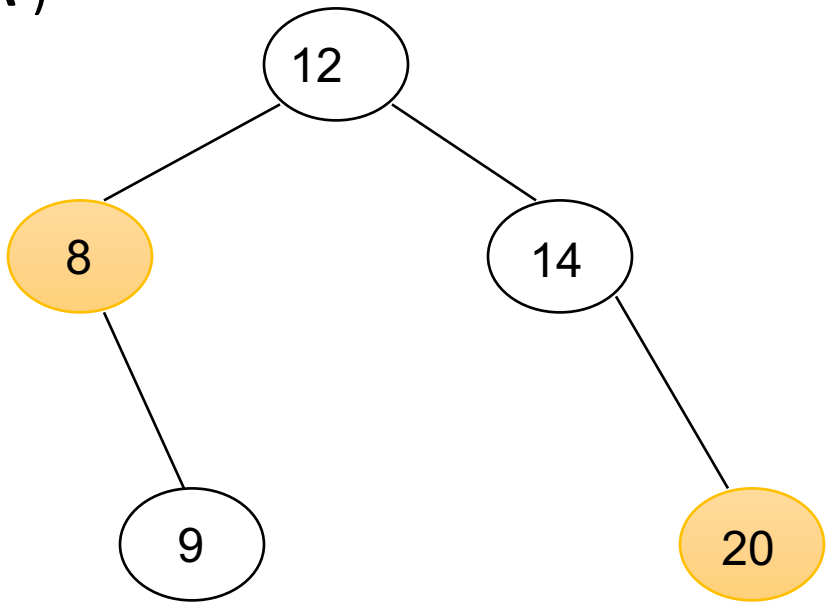
What else can we say?

$left(x) \leq x \leq right(x)$

- (All elements to the left of a node ***X)*** $\leqq$ (node ***X***)
- (All elements to the right of a node ***X)*** $\geqq$ (node ***X***)

- **The smallest element is the *left-most element***
- **The largest element is the *right-most element***

$left(x) \leq x \leq right(x)$



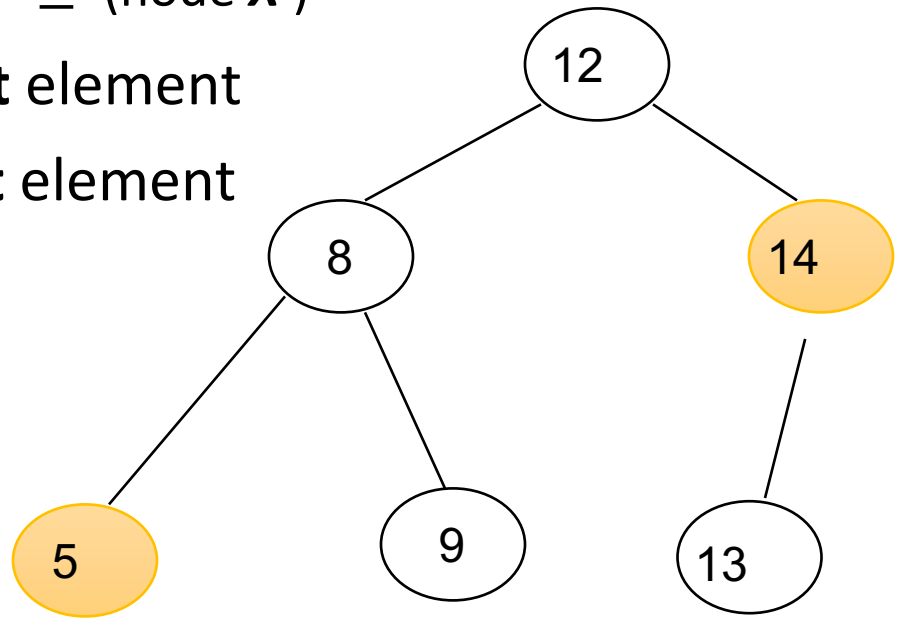
What else can we say?

$left(x) \leq x \leq right(x)$

- (All elements to the left of a node ***X)*** $\leqq$ (node ***X***)
- (All elements to the right of a node ***X)*** $\geqq$ (node ***X***)

- The **smallest** element is the **left-most** element
- The **largest** element is the **right-most** element

$left(x) \leq x \leq right(x)$



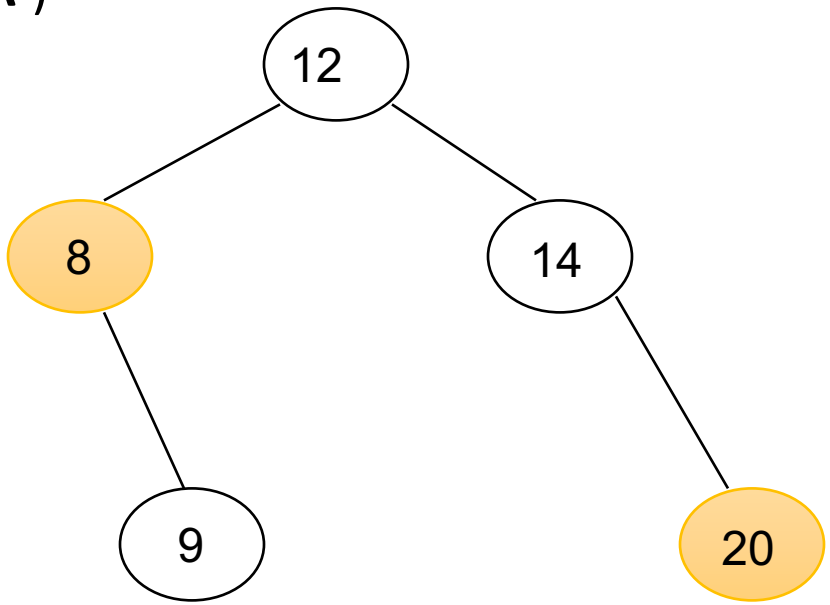
What else can we say?

$left(x) \leq x \leq right(x)$

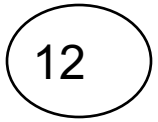
- (All elements to the left of a node ***X)*** $\leqq$ (node ***X***)
- (All elements to the right of a node ***X)*** $\geqq$ (node ***X***)

- The **smallest** element is the **left-most** element
- The **largest** element is the **right-most** element

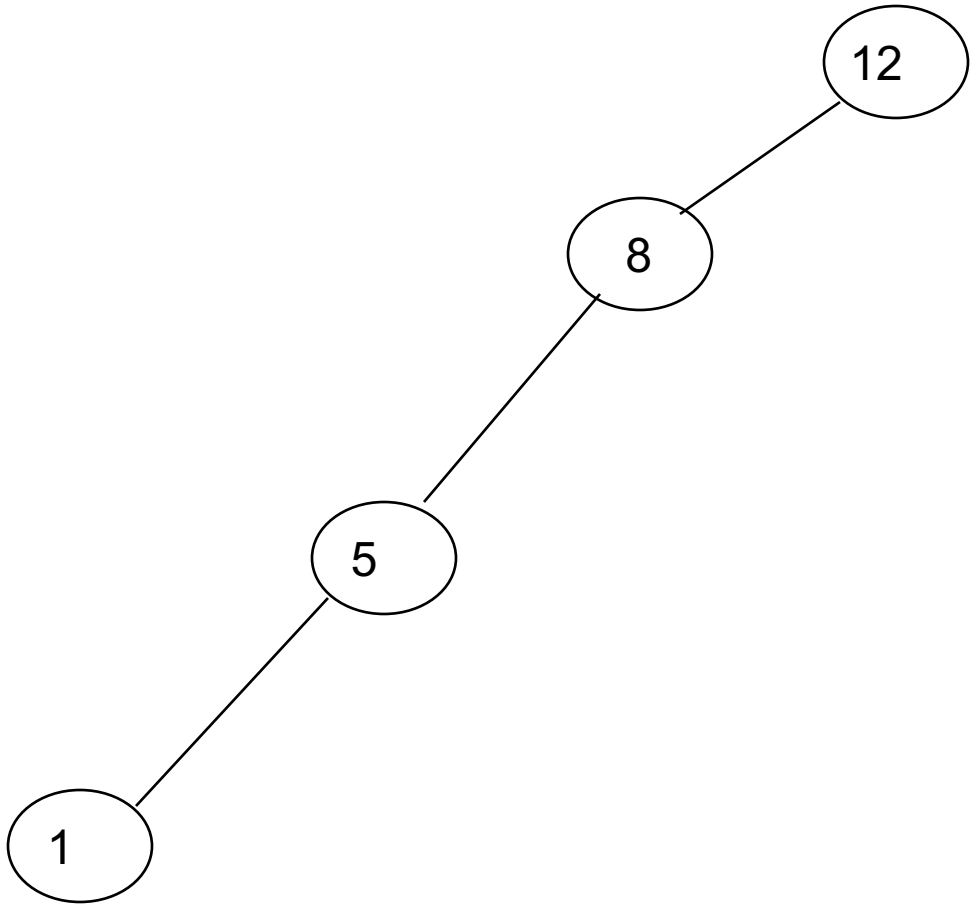
$left(x) \leq x \leq right(x)$



Another example: the loner



Another example: the twig



Operations

Search(T,k) – Does value k exist in tree T

Insert(T,k) – Insert value k into tree T

Delete(T,x) – Delete node x from tree T

Minimum(T) – What is the smallest value in the tree?

Maximum(T) – What is the largest value in the tree?

Successor(T,x) – What is the next element in sorted order after x

Predecessor(T,x) – What is the previous element in sorted order of x

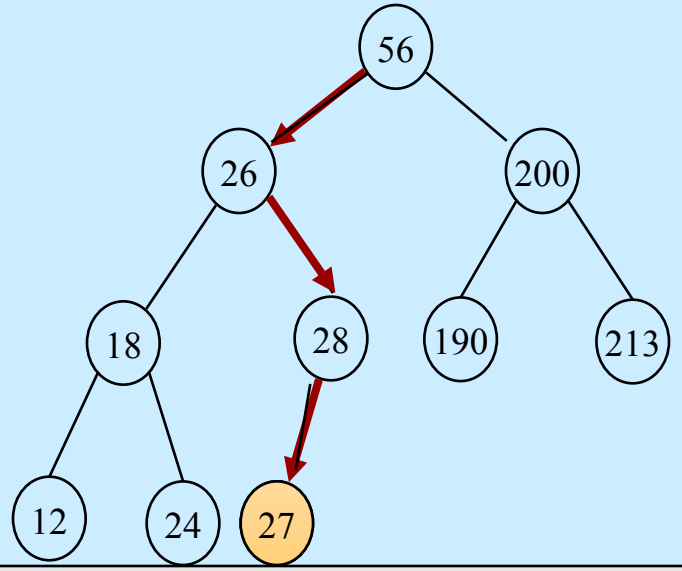
Tree Search

BST-Search(x, k)

1. **if** x = NIL *or* k = $key[x]$
2. **then** return x
3. **if** $k < key[x]$ return Tree-Search($left[x]$, k)
4. **else** return Tree-Search($right[x]$, k)

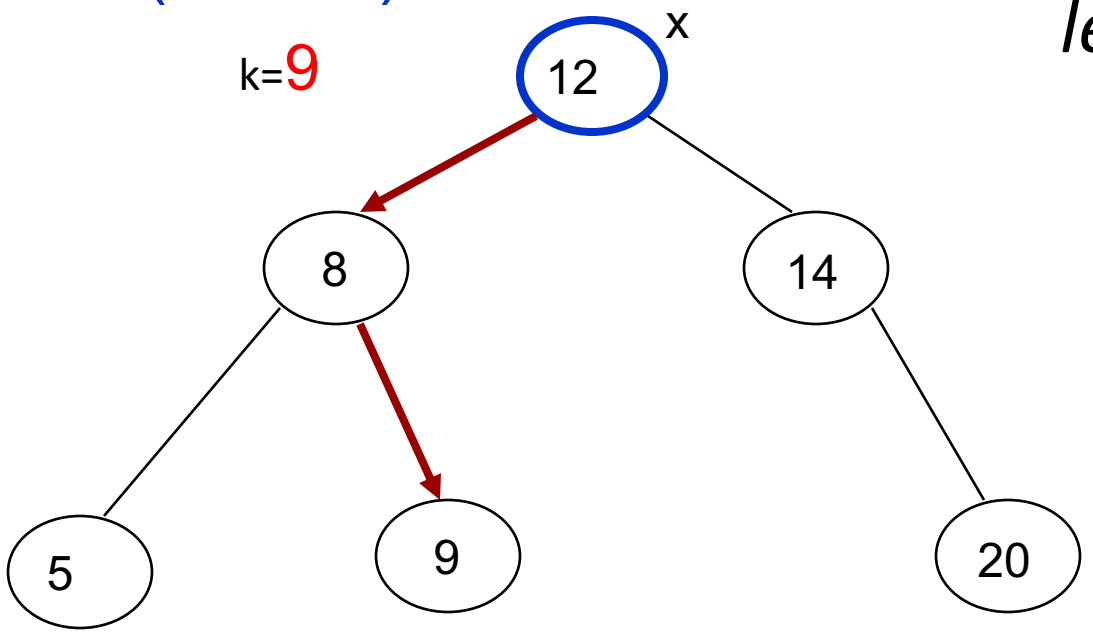
Running time: $O(h)$

Example: search for 27



Finding an element

Search(T, k=9)

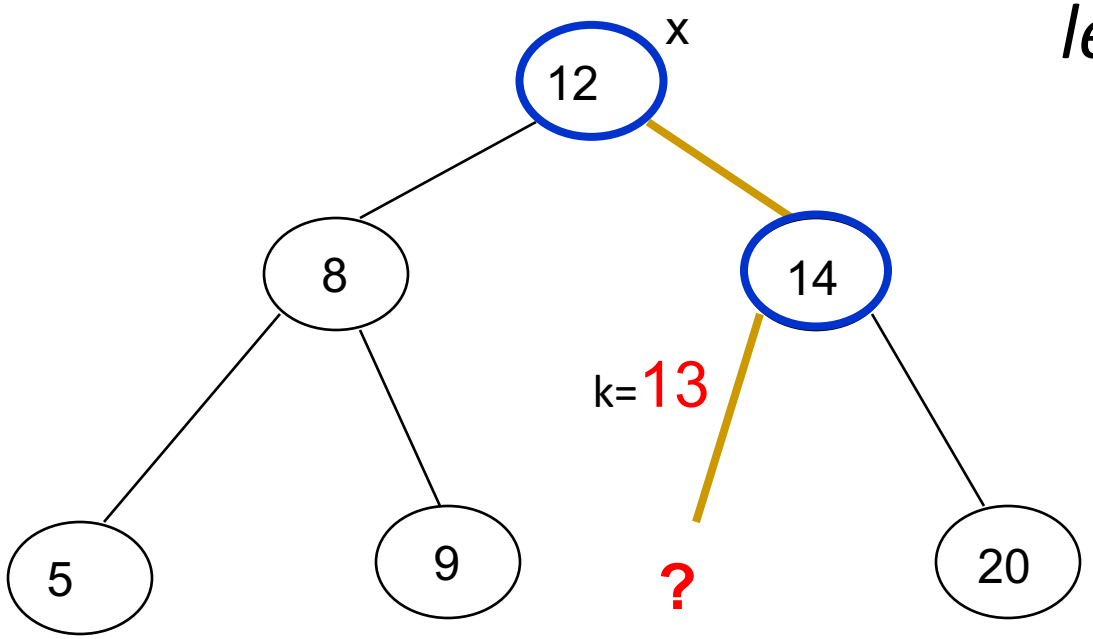


$left(x) \leqq x \leqq right(x)$

```
BSTSEARCH(x, k)
1  if x = null or k = x
2          return x
3  elseif k < x
4          return BSTSEARCH(LEFT(x), k)
5  else
6          return BSTSEARCH(RIGHT(x), k)
```

Finding an element

Search(T, k=13)



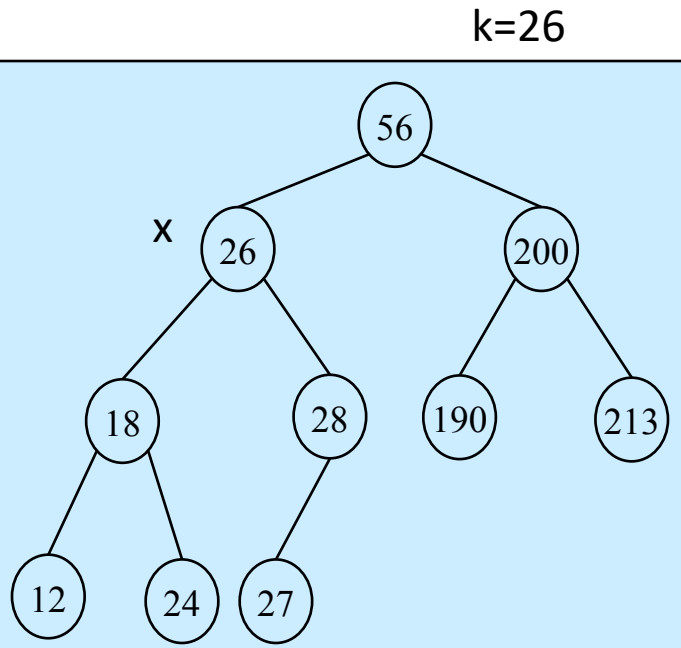
$left(x) \leqq x \leqq right(x)$

```
BSTSEARCH(x, k)
1  if x = null or k = x
2          return x
3  elseif k < x
4          return BSTSEARCH(LEFT(x), k)
5  else
6          return BSTSEARCH(RIGHT(x), k)
```

Iterative Tree Search

Iterative-Tree-Search(x, k)

1. **while** $x \neq NIL$ **and** $k \neq key[x]$
2. **if** $k < key[x]$
3. $x \leftarrow left[x]$
4. **else** $x \leftarrow right[x]$
5. **return** x



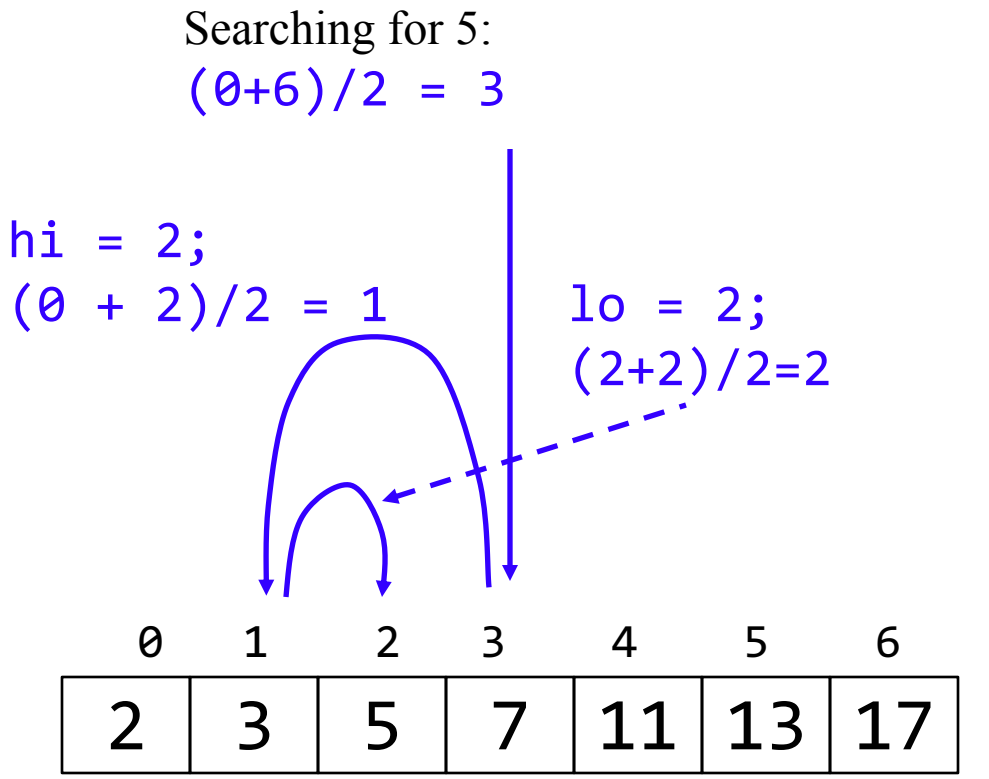
- The iterative tree search is more efficient on most computers.
- The recursive tree search is more straightforward.

Querying (Searching) a Binary Search Tree

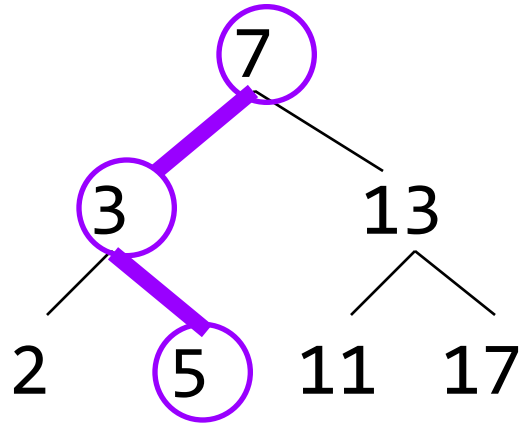
- All dynamic-set search operations can be supported in $O(h)$ time.
- $h = \Theta(lg\ n)$ for a balanced binary tree (and for an average tree built by adding nodes in random order.)
- $h = \Theta(n)$ for an unbalanced tree that resembles a linear chain of n nodes in the worst case.

Binary search in a sorted array

- Look at array location `(lo + hi)/2`



Using a binary search tree



Height of the tree

Worst case height?

- n-1
- “the twig”

Best case height?

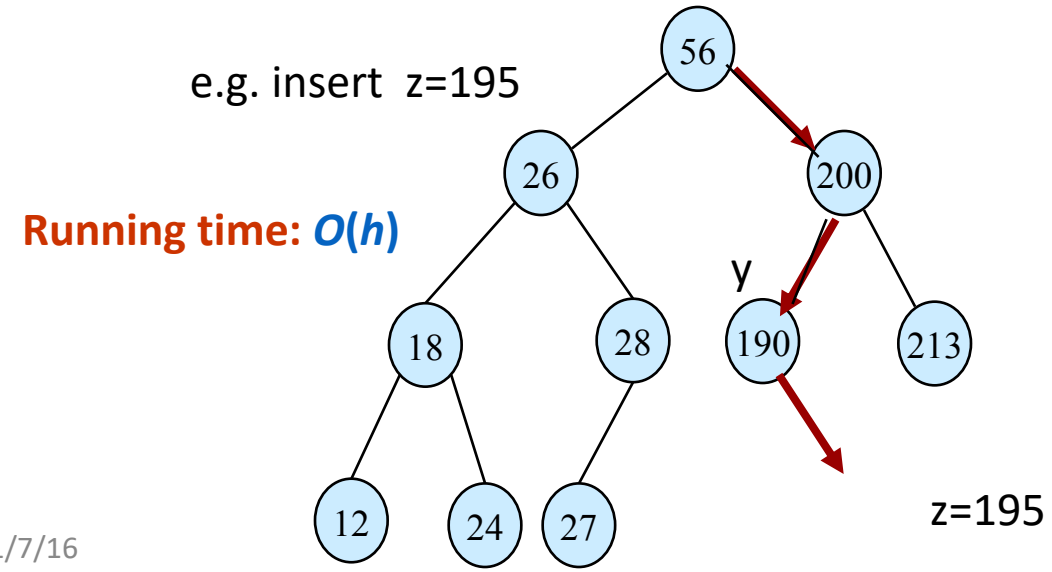
- floor($\log_2 n$)
- complete (or near complete) binary tree

Average case height?

- Depends on two things:
  - the data
  - how we build the tree!

BST Insertion – Pseudocode

- Change the dynamic set represented by a BST.
- Ensure the binary-search-tree property holds after change.
- Similar to Tree-Search
- Insert z in place of NIL




```
BST-Insert(T, z)
1.   y ← NIL
2.   x ← root[T]
3.   while x ≠ NIL
4.       do y ← x
5.          if key[z] < key[x]
6.                 x ← left[x]
7.          else x ← right[x]
8.   p[z] ← y
9.   if y = NIL
10.         root[t] ← z
11.  else if key[z] < key[y]
12.         left[y] ← z
13.  else right[y] ← z
```

Similar to search

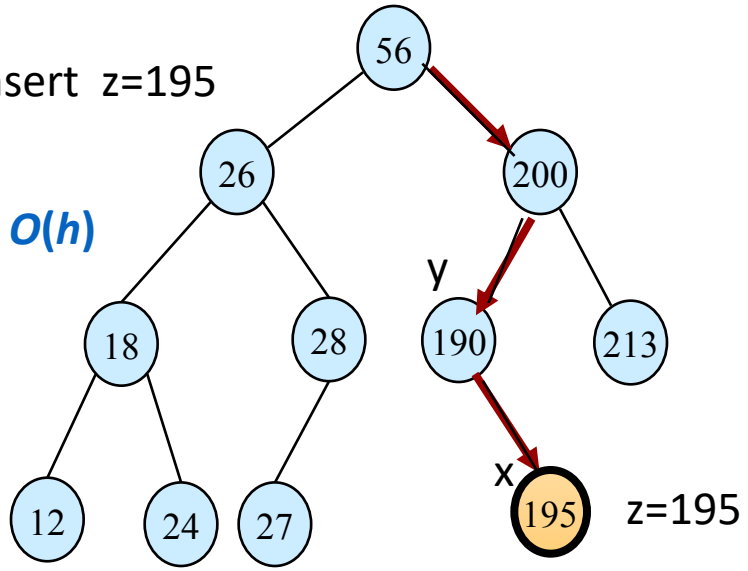
Find the correct location in the tree

keeps track of the previously visited node (so we know when we fall off the tree).

add node onto the bottom of the tree

BST Insertion – Pseudocode

- Change the dynamic set represented by a BST.
- Ensure the binary-search-tree property holds after change.
- Similar to Tree-Search
- Insert z in place of NIL




```
BST-Insert(T, z)
1.   y ← NIL
2.   x ← root[T]
3.   while x ≠ NIL
4.       do y ← x
5.          if key[z] < key[x]
6.                x ← left[x]
7.          else x ← right[x]
8.   p[z] ← y
9.   if y = NIL
10.         root[t] ← z
11.  else if key[z] < key[y]
12.         left[y] ← z
13.  else right[y] ← z
```

Similar to search

Find the correct location in the tree

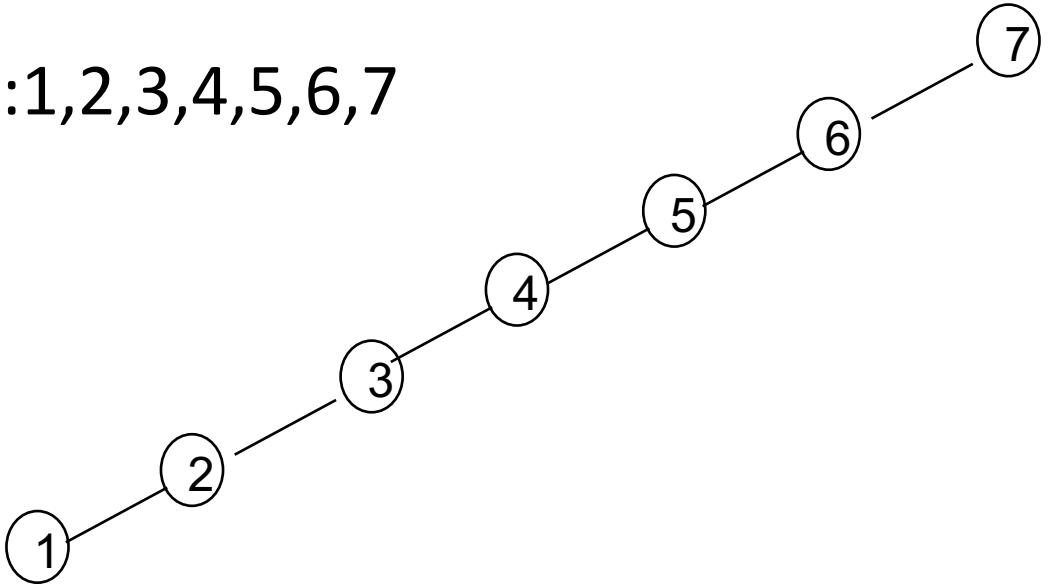
keeps track of the previously visited node (so we know when we fall off the tree).

add node onto the bottom of the tree

Build a Binary Search Tree

- Worst Case Run Time: $T(n) = O(n^2)$
  - When inputs are all sorted or reversely sorted
- Example:7,6,5,4,3,2,1

- Example:1,2,3,4,5,6,7

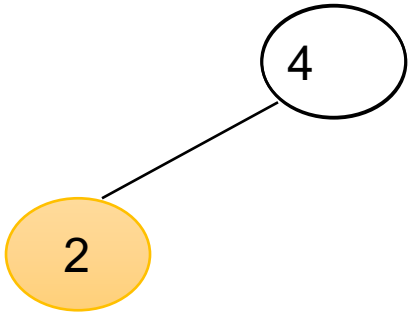


Exercise

- Best Case Run Time: T(n) = O(n log n)
- Example: **4**, 2, 1, 3, 6, 5, 7

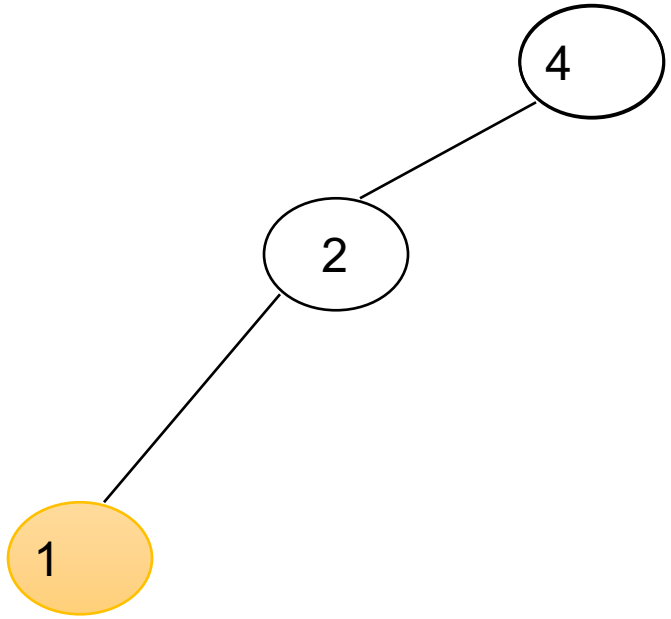
Exercise

- Best Case Run Time: T(n) = O(n log n)
- Example: 4, **2**, 1, 3, 6, 5, 7



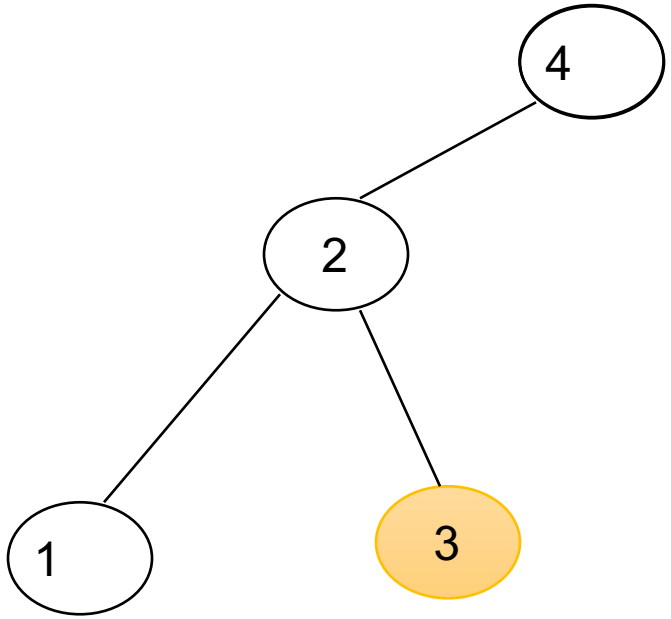
Exercise

- Best Case Run Time: T(n) = O(n log n)
- Example: 4, 2, **1**, 3, 6, 5, 7



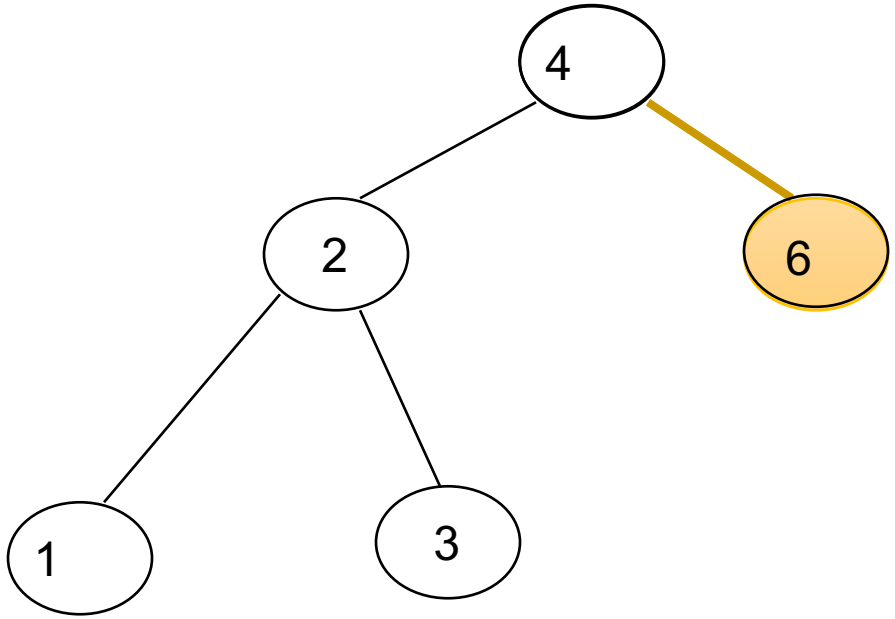
Exercise

- Best Case Run Time: T(n) = O(n log n)
- Example: $4,\ 2,\ 1,\ \mathbf{3},\ 6,\ 5,\ 7$



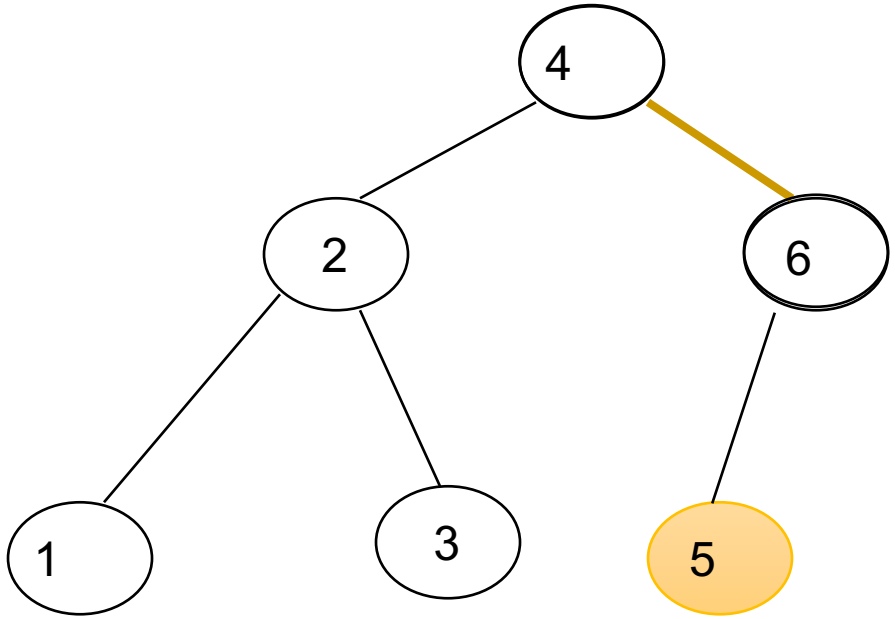
Exercise

- Best Case Run Time: T(n) = O(n log n)
- Example: 4, 2, 1, 3, **6**, 5, 7



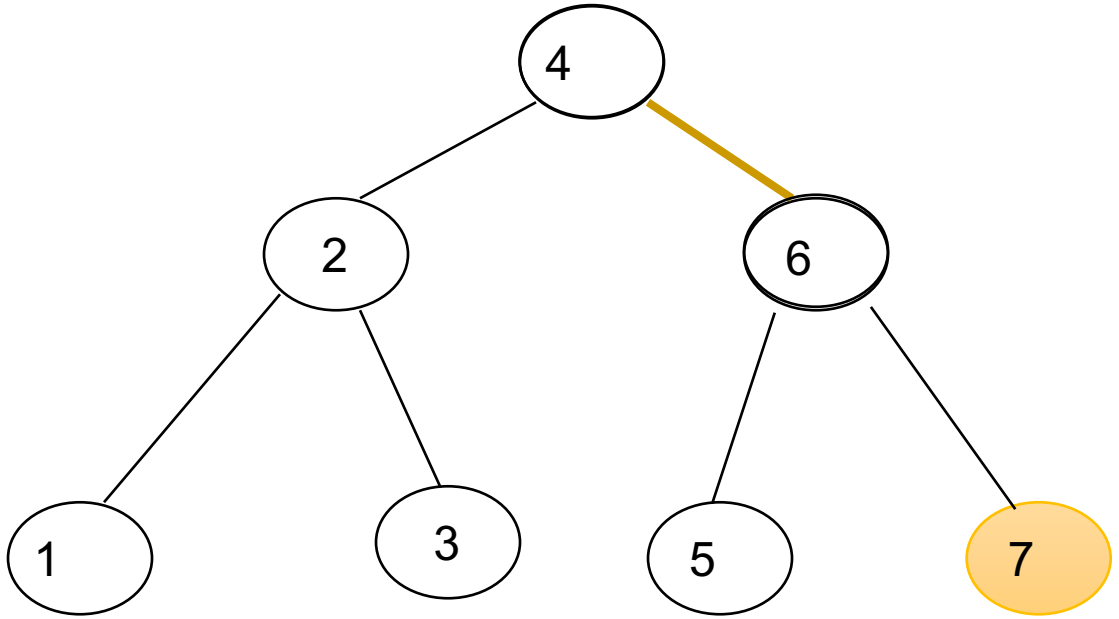
Exercise

- Best Case Run Time: T(n) = O(n log n)
- Example: 4, 2, 1, 3, 6, **5**, 7



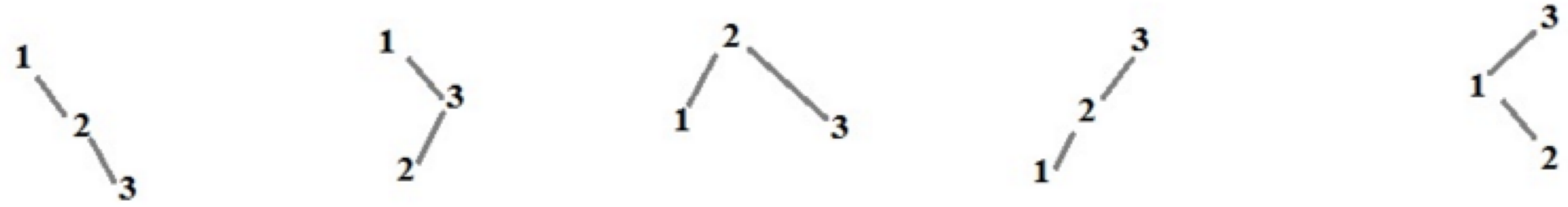
Exercise

- Best Case Run Time: T(n) = O(n log n)
- Example: 4, 2, 1, 3, 6, **5**, 7



Number of BST's For n Distinct Elements

- Enumerate all BST's out of n elements
  - Order the elements as 1, 2, 3, ..., n.
  - Pick one of those elements to use as the pivot.
    - This splits the remaining elements into two groups of elements before and after the pivot
  - Recursively build structures out of those two groups.
  - Combine those two structures

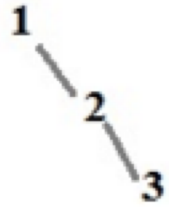


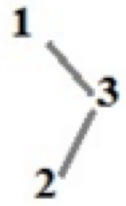
Number of BST's For n Distinct Elements

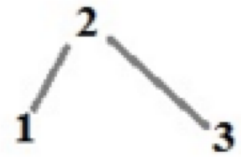
- How many binary search trees can be constructed from n distinct elements?

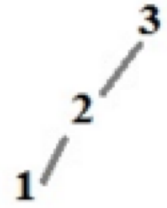
$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!} = \prod_{k=2}^{n}\frac{n+k}{k} \qquad \text{for } n \geq 0.$$

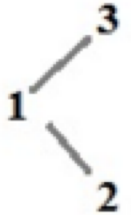
- Catalan number https://en.wikipedia.org/wiki/Catalan_number







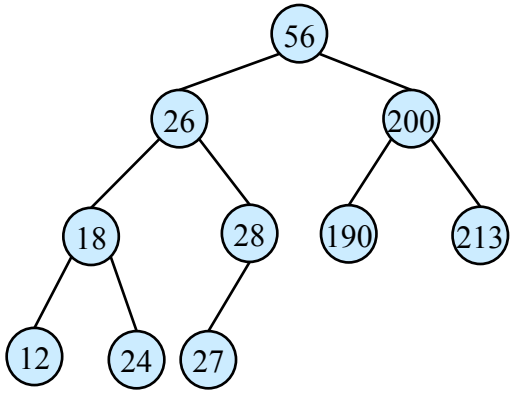




Inorder Traversal

The binary-search-tree property allows the keys of a binary search tree to be printed, in (monotonically increasing) order, recursively.

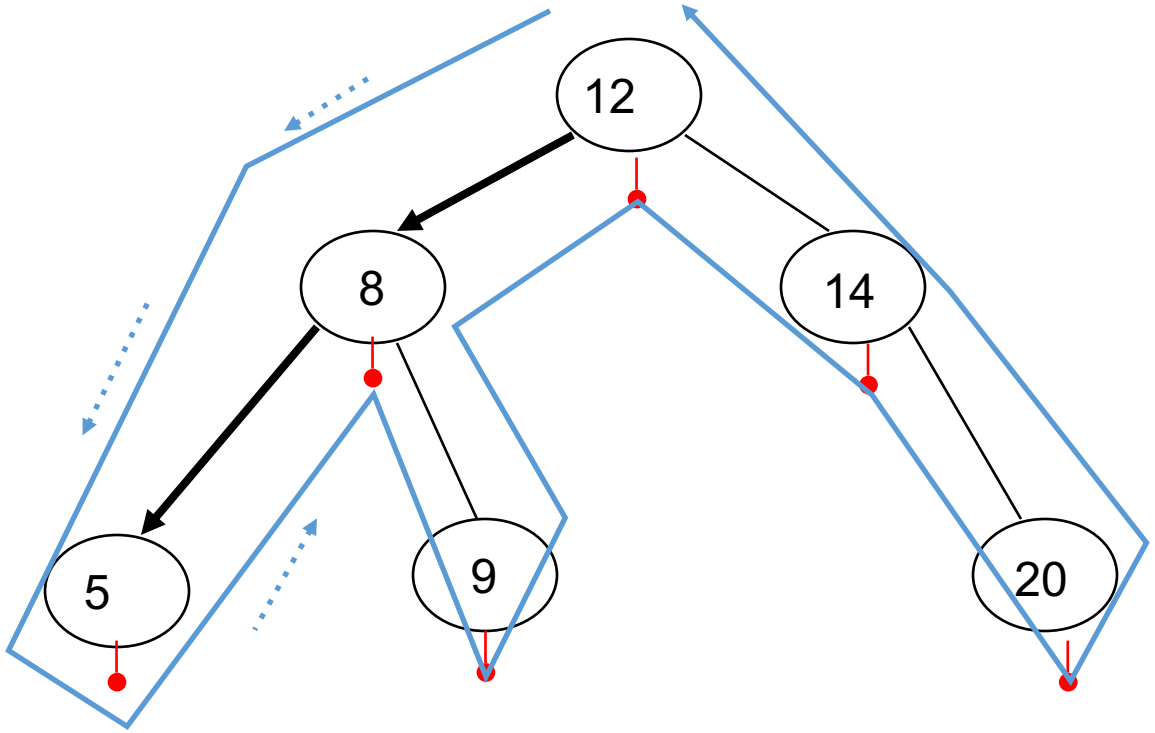
```
INORDERTREEWALK(x)
1  if x ≠ null
2        INORDERTREEWALK(LEFT(x))
3        print x
4        INORDERTREEWALK(RIGHT(x))
```



- How long does the walk take? $\Theta(n)$

Visiting all nodes

In sorted order



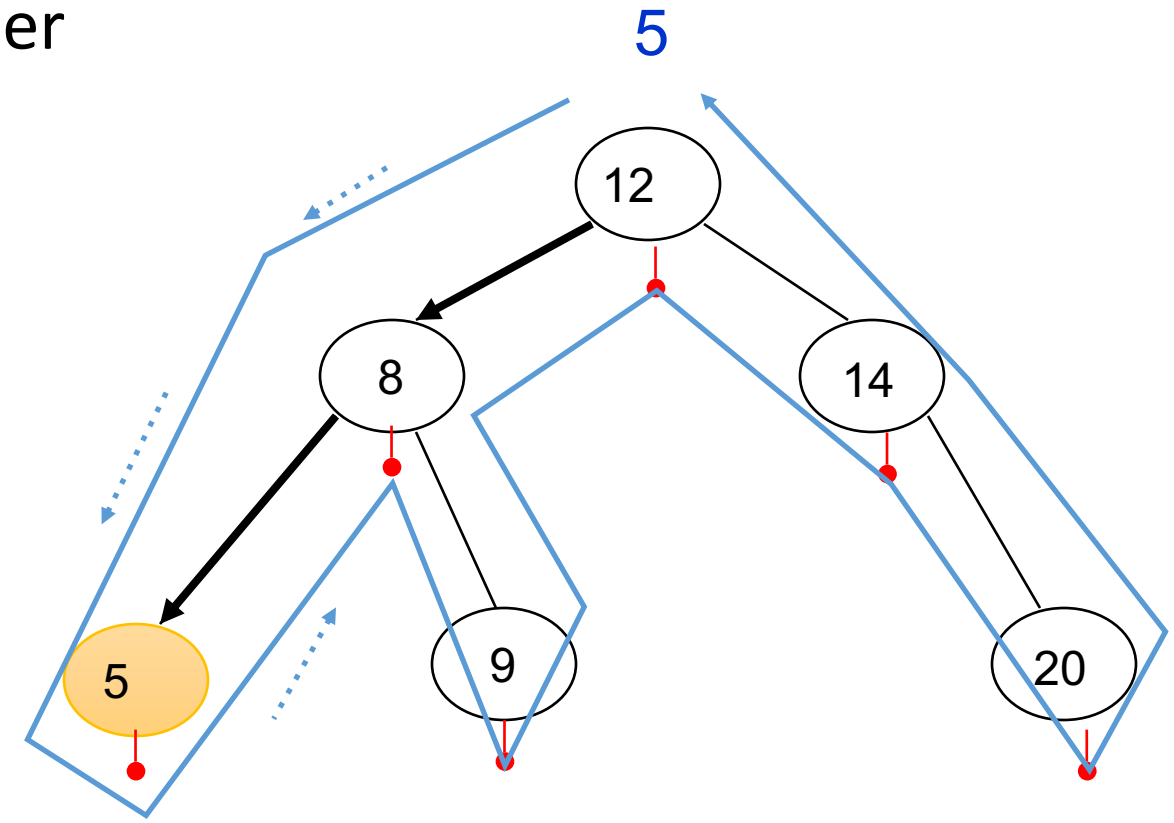
Inorder-Tree-Walk(*left*[*x*])

print *key*[*x*]

Inorder-Tree-Walk(*right*[*x*])

Visiting all nodes

In sorted order



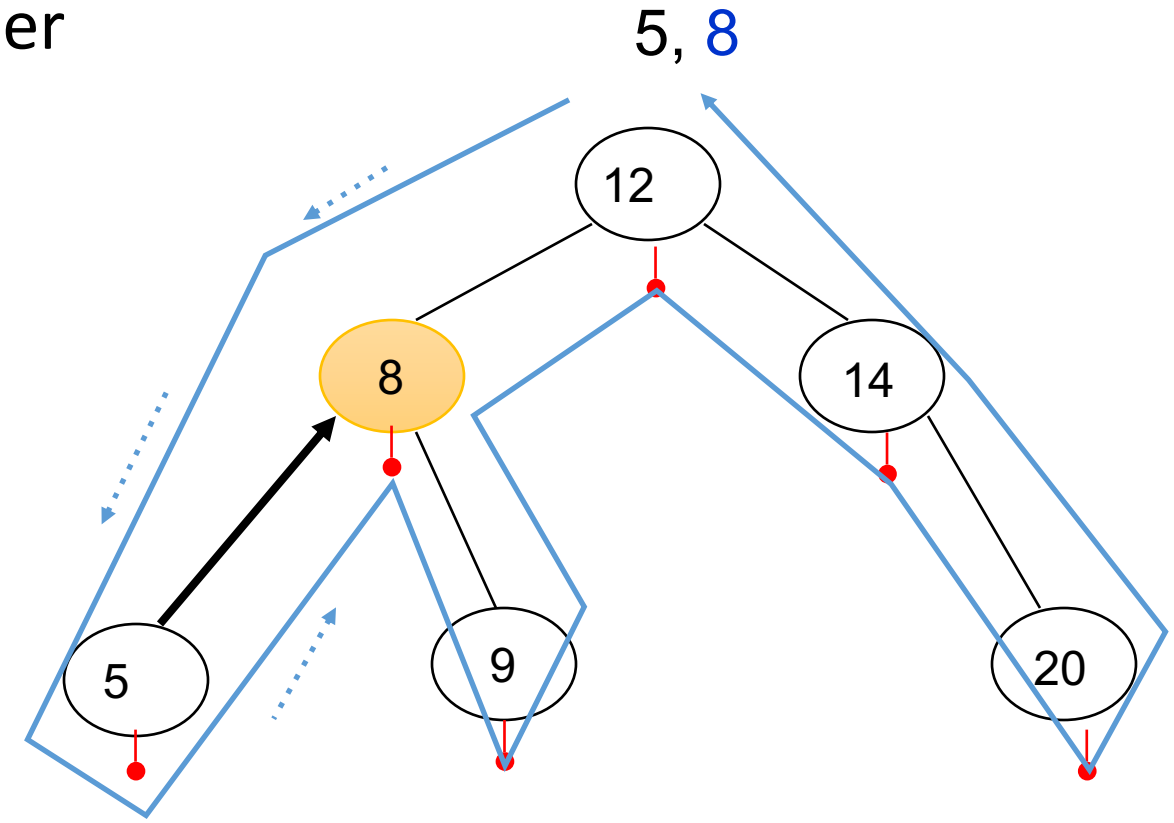
Inorder-Tree-Walk(*left*[*x*])

print *key*[*x*]

Inorder-Tree-Walk(*right*[*x*])

Visiting all nodes

In sorted order



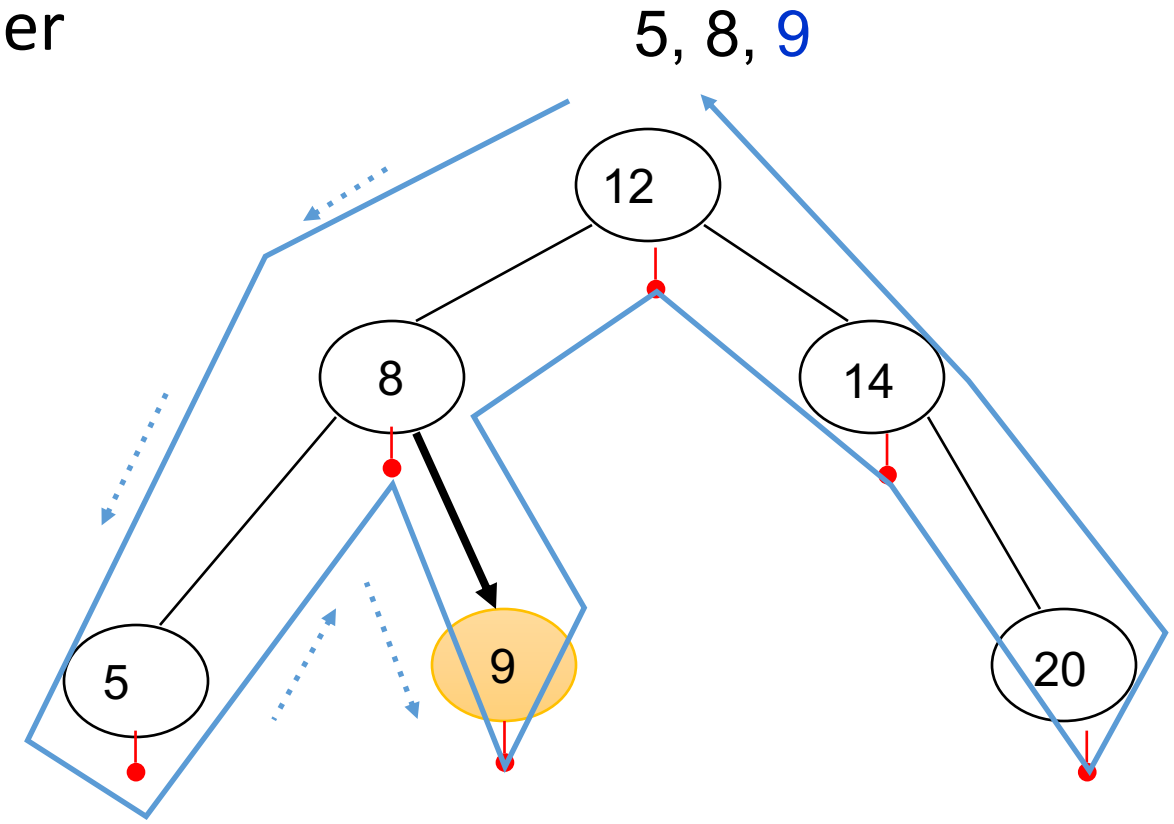
Inorder-Tree-Walk(*left*[*x*])

print *key*[*x*]

Inorder-Tree-Walk(*right*[*x*])

Visiting all nodes

In sorted order



Inorder-Tree-Walk(*left*[*x*])

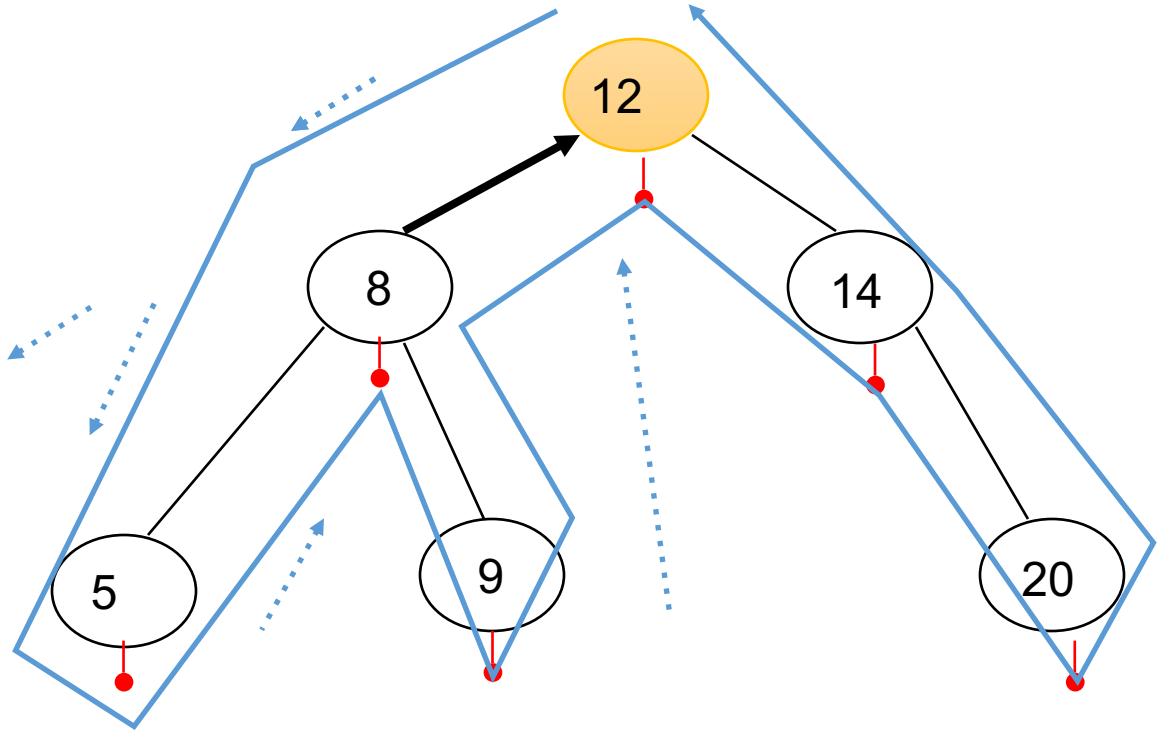
print *key*[*x*]

Inorder-Tree-Walk(*right*[*x*])

Visiting all nodes

In sorted order

5, 8, 9, 12



Inorder-Tree-Walk(*left*[*x*])

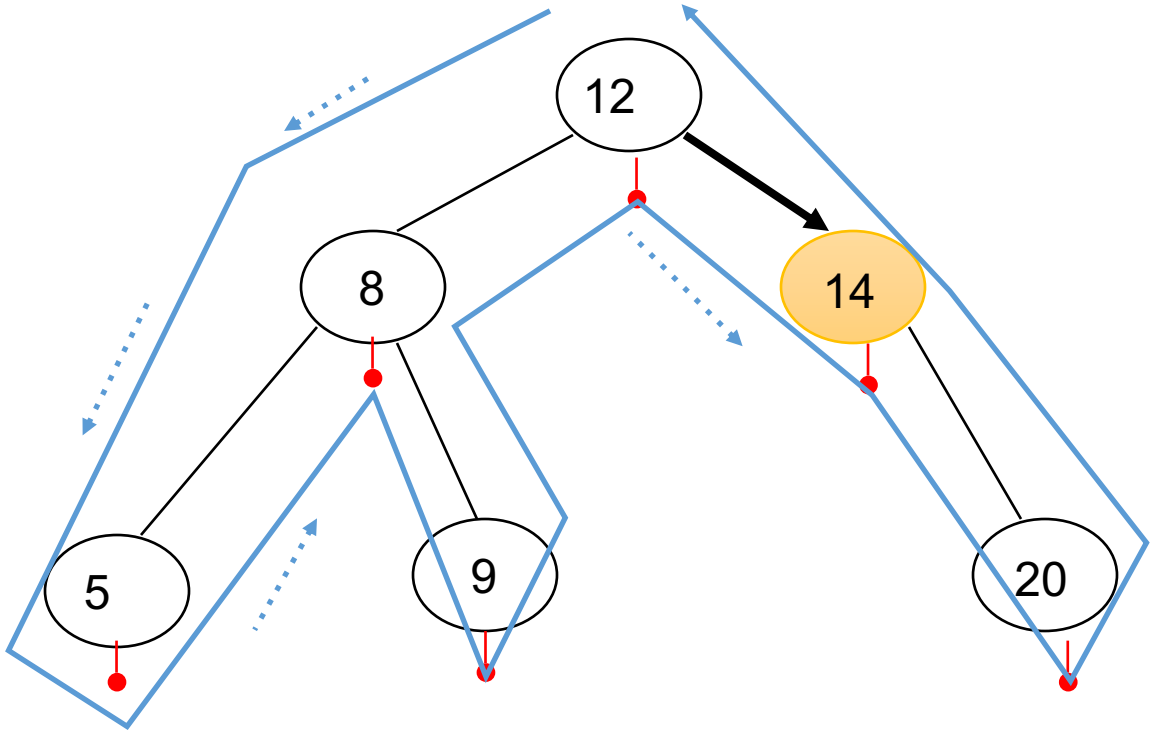
print *key*[*x*]

Inorder-Tree-Walk(*right*[*x*])

Visiting all nodes

In sorted order

5, 8, 9, 12, 14



Inorder-Tree-Walk(*left*[*x*])

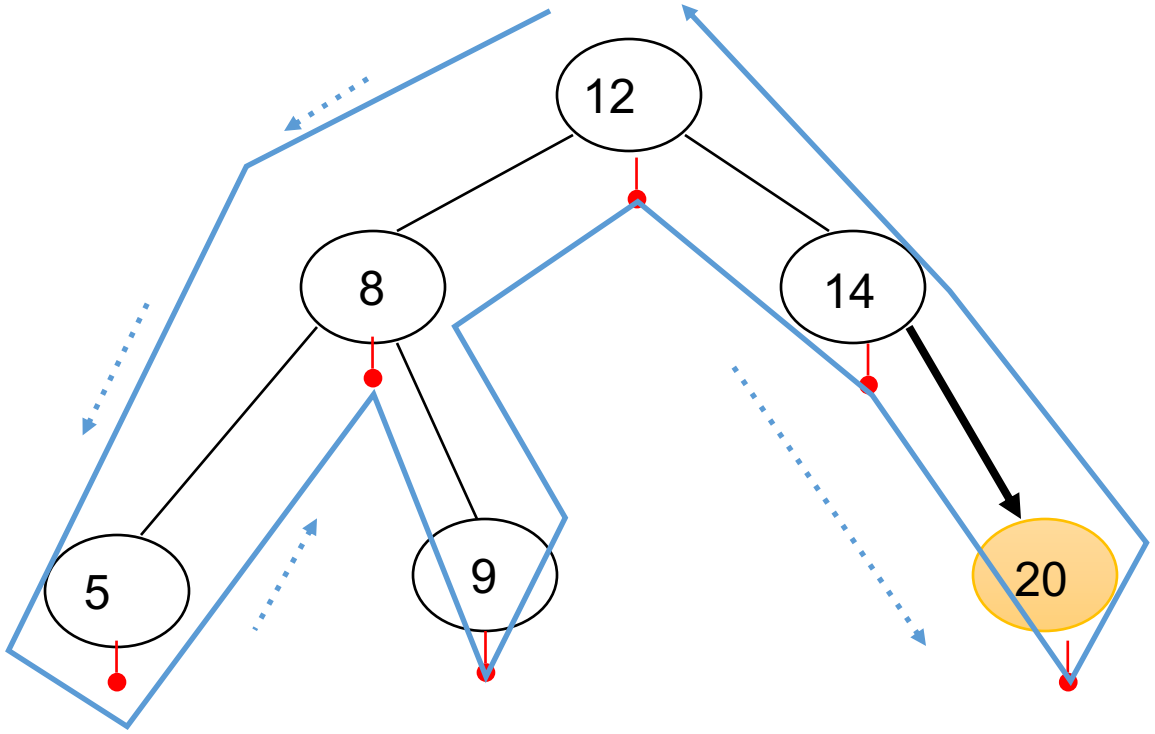
print *key*[*x*]

Inorder-Tree-Walk(*right*[*x*])

Visiting all nodes

In sorted order

5, 8, 9, 12, 14, 20



Inorder-Tree-Walk(*left*[*x*])

print *key*[*x*]

Inorder-Tree-Walk(*right*[*x*])

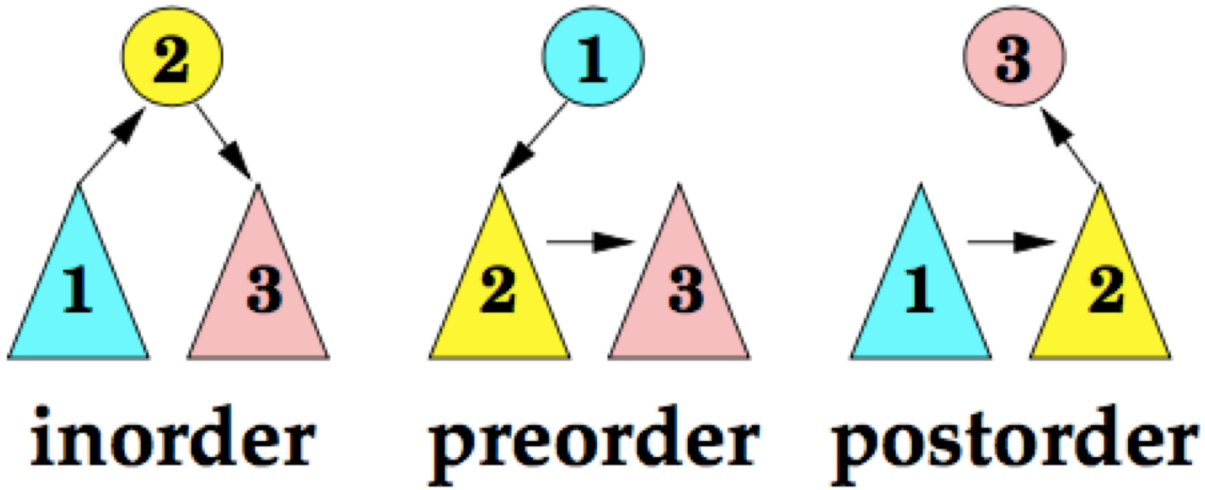
Is it correct?

Does it print out all of the nodes in sorted order?

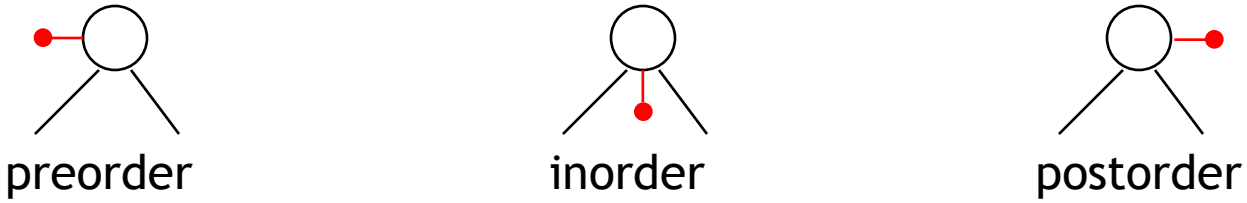
```
INORDERTREEWALK(x)
1  if x ≠ null
2        INORDERTREEWALK(LEFT(x))
3        print x
4        INORDERTREEWALK(RIGHT(x))
```

$$left(i) \leq i \leq right(i)$$

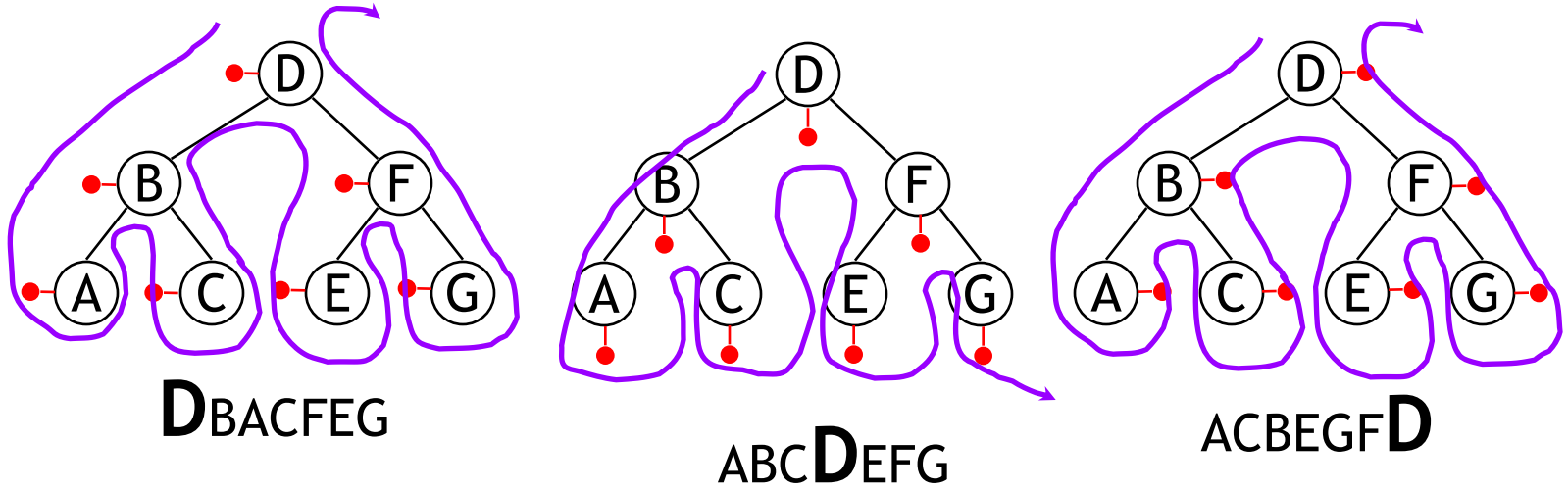
What about?



- The order in which the nodes are visited during a tree traversal can be easily determined by imagining there is a "flag" attached to each node, as follows:

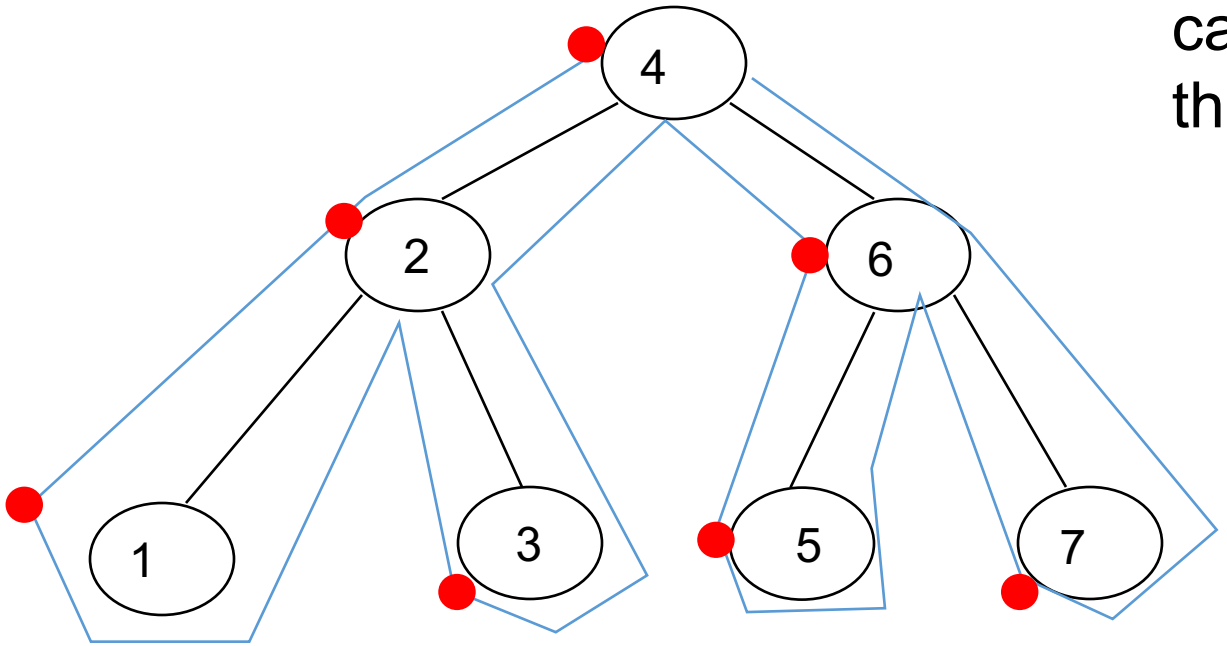


- To traverse the tree, collect the flags:



BST **Pre**Order Traversal (Depth-First Search)

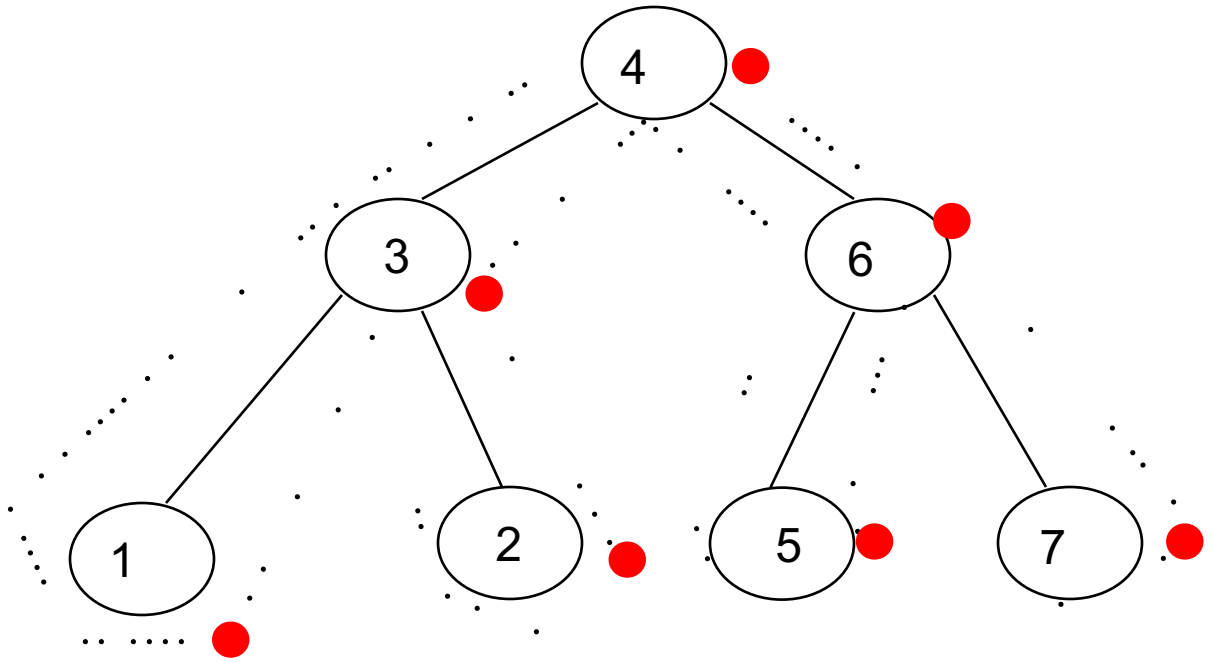
```
TreeWalk(x)
1   if x ≠ null
2         print x
3         TreeWalk(Left(x))
4         TreeWalk(Right(x))
```



Pre-Order Traversal, 4,2,1,3,6,5,7 , can be saved and used to re-build the tree.

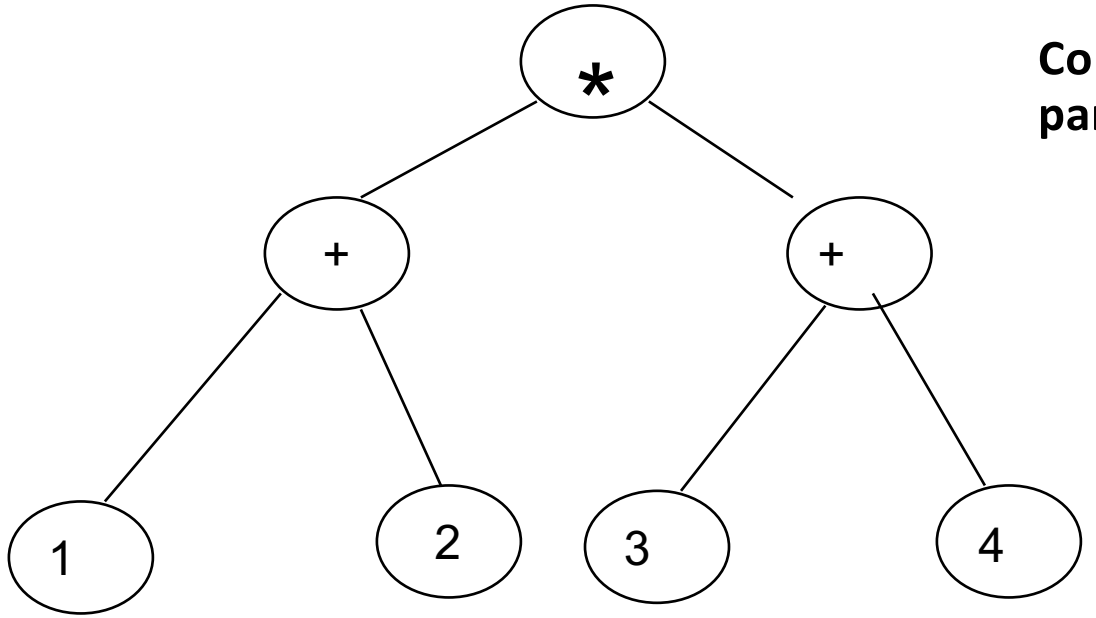
BST **Post**Order Traversal

```
TreeWalk(x)
1  if x ≠ null
2        TreeWalk(Left(x))
3        TreeWalk(Right(x))
4        print x
```



1,2,3,5,7,6,4

Why Preorder or Postorder Traversal ?



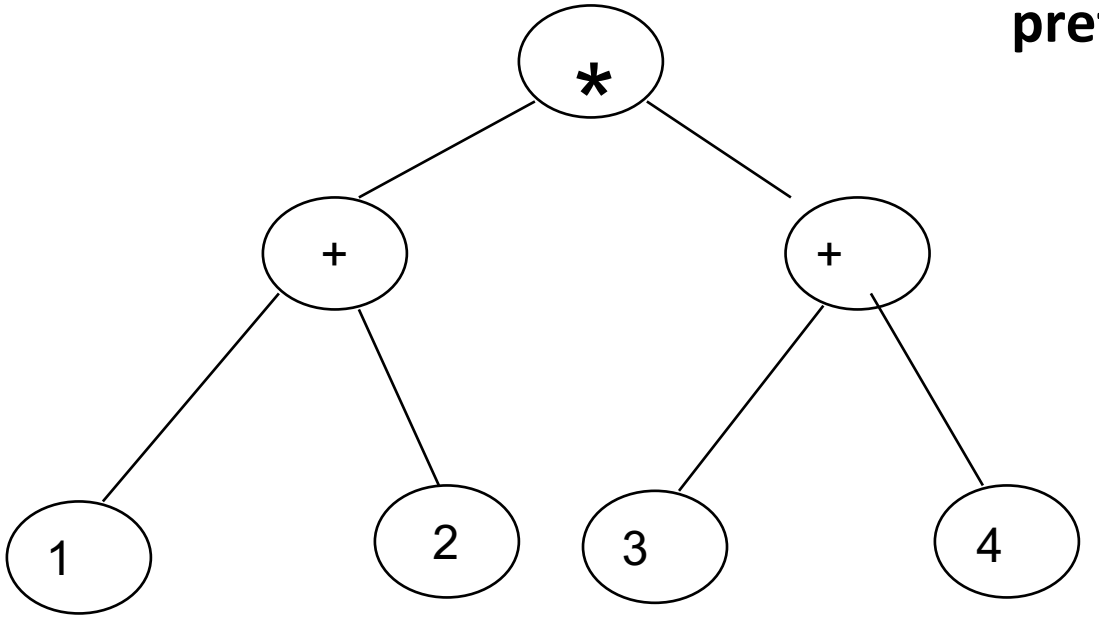
Conventional notation can be confusing withut parenthesis

(1+2)*(3+4) ⇛ (InOrder Traveral)
1+2*3+4

Parse Tree in Compiler

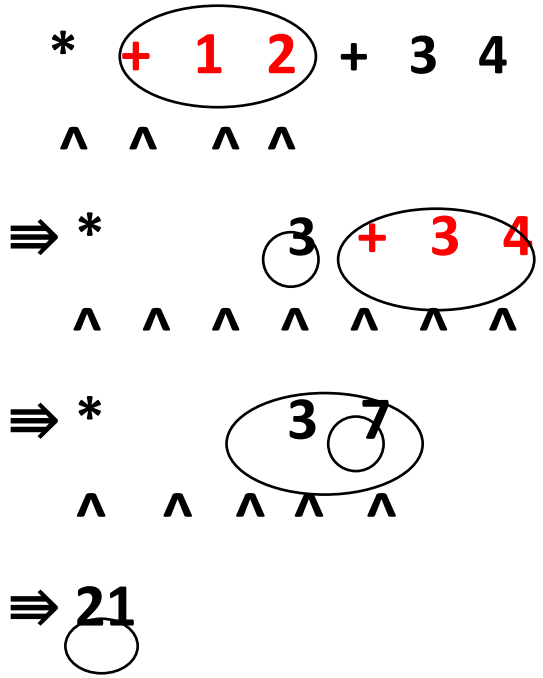
https://en.wikipedia.org/wiki/Polish_notation

Preorder traversal



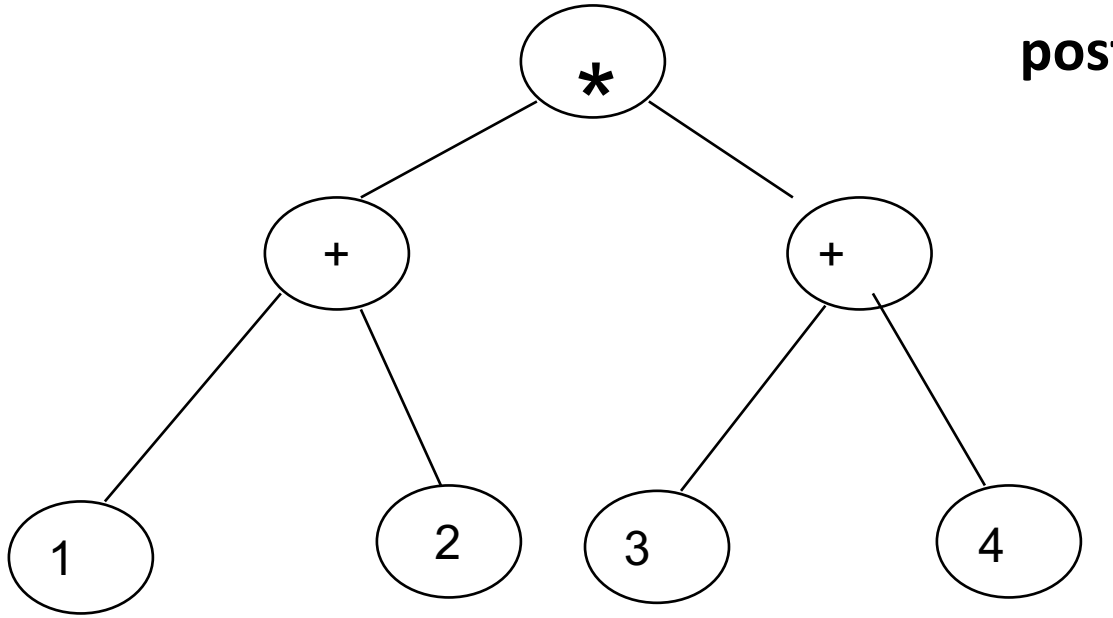
Parse Tree

prefix notation: (1+2)*(3+4) ⇛ *+12+34



No ambiguity with parenthesis since all operation needs two operands

Postorder traversal



Parse Tree

postfix notation: (1+2)*(3+4) ⇛ 12+34+*

1 2 + 3 4 + *
^ ^ ^

⇛ 3 3 4 + *
^ ^ ^ ^ ^ ^

⇛ 3 7 *
^ ^ ^ ^ ^ ^^ ^

⇛ 21

No ambiguity with parenthesis since all operation needs two operands

Sorting Using BST

- Build BST Using Insertion Recursively
  - Best Case: O (n log n)
  - Worst Case: $O(n^2)$
- BST InOrder Traversal
  - O(n)

Finding Min & Max

- The binary-search-tree property guarantees that:
  - » The minimum is located at the left-most node.
  - » The maximum is located at the right-most node.

BST-Minimum(x)	BST-Maximum(x)
1. **while** $left[x] \neq NIL$	1. **while** $right[x] \neq NIL$
2. $x \leftarrow left[x]$	2. $x \leftarrow right[x]$
3. **return** x	3. **return** x

Q: How long do they take? O(h)

Q: How about Heap?

Successor and Predecessor

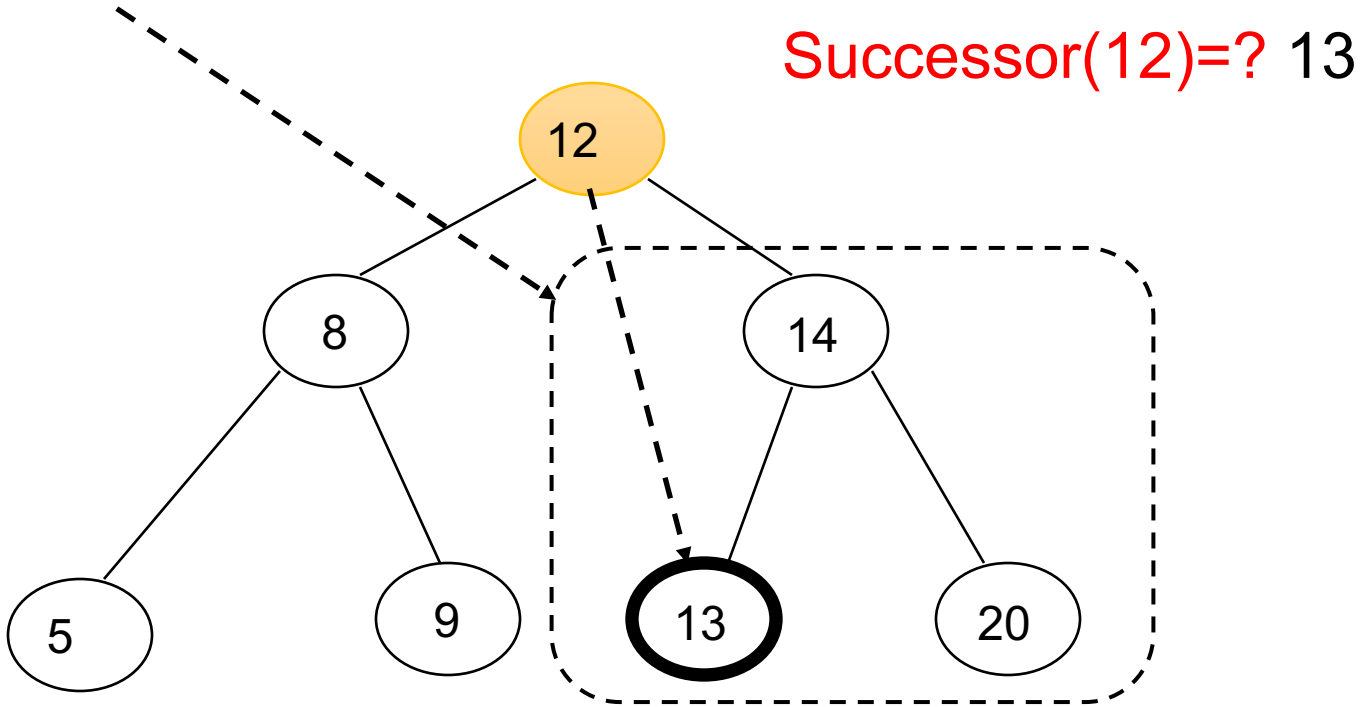
Given sorted list: a, b, c, d, e, x, y, f, g, h, I, …

- Successor of node *x* is node *y*
- Predecessor of node *y* is node *x*
  - The successor of the last node (with the largest key) is NIL.
  - The predecessor of the first node (with the smallest key) is NIL.

Successor

Smallest node of all those larger than this node

→ Leftmost element of the right subtree

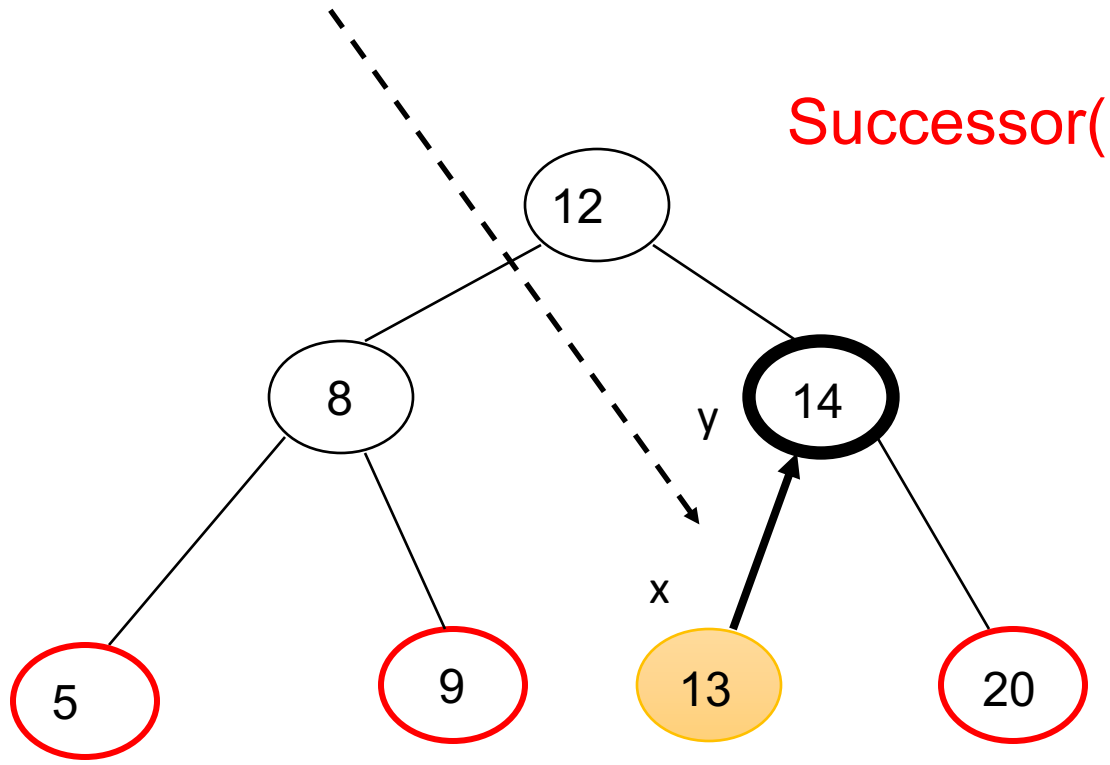


Successor

What if the node doesn't have a right subtree?

Keep going up until x is no longer a right child of y

Successor(13)=? 14

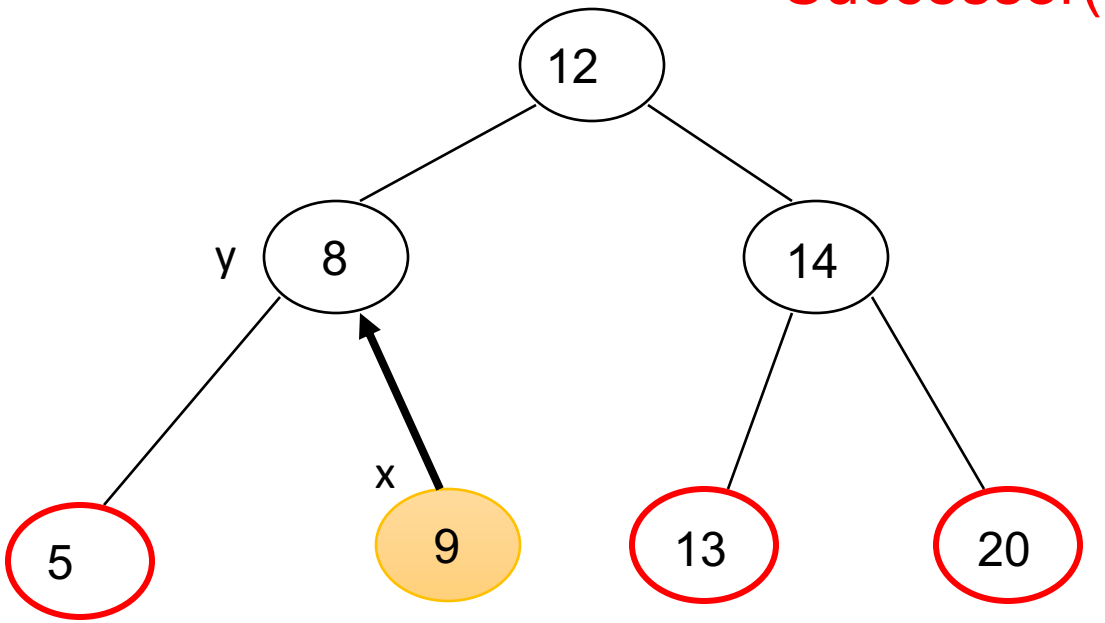


Successor

What if the node doesn't have a right subtree?

Keep going up until x is no longer a right child of y

Successor(9)=?

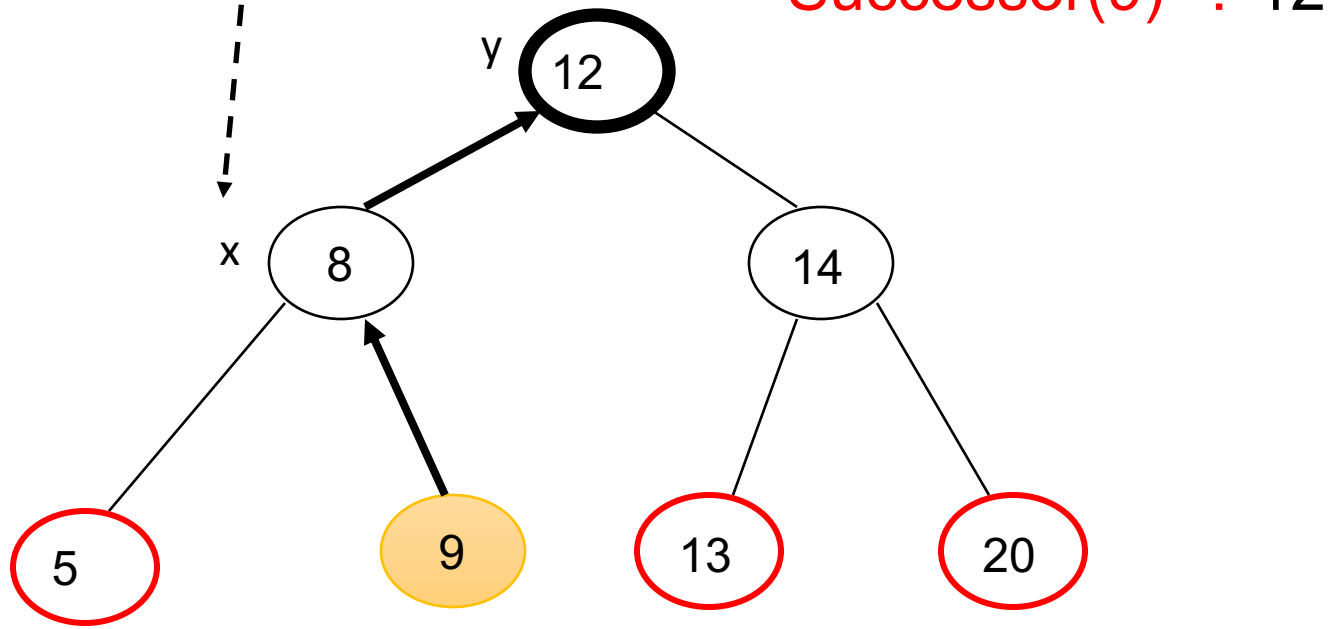


Successor

What if the node doesn't have a right subtree?

Keep going up until x is no longer a right child of y

Successor(9)=? 12



Pseudo-code for Successor

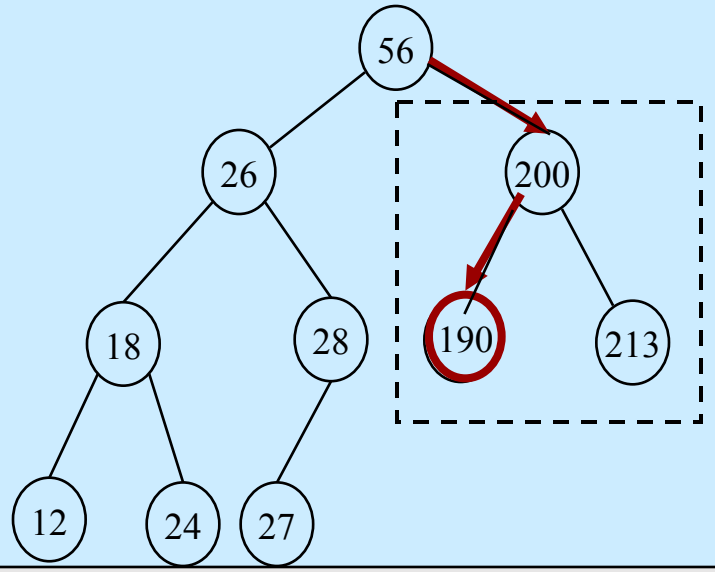
Tree-Successor(*x*)

1. **if** *right*[*x*] ≠ *NIL*
2. return Tree-Minimum(*right*[*x*])
3. y ← *p*[*x*]
4. **while** *y* ≠ *NIL* **and** *x* = *right*[*y*]
5. *x* ← *y*
6. *y* ← *p*[*y*]
7. **return** *y*

Code for ***predecessor*** is symmetric.

Running time: $O(h)$

Example: successor of 56



Pseudo-code for Successor

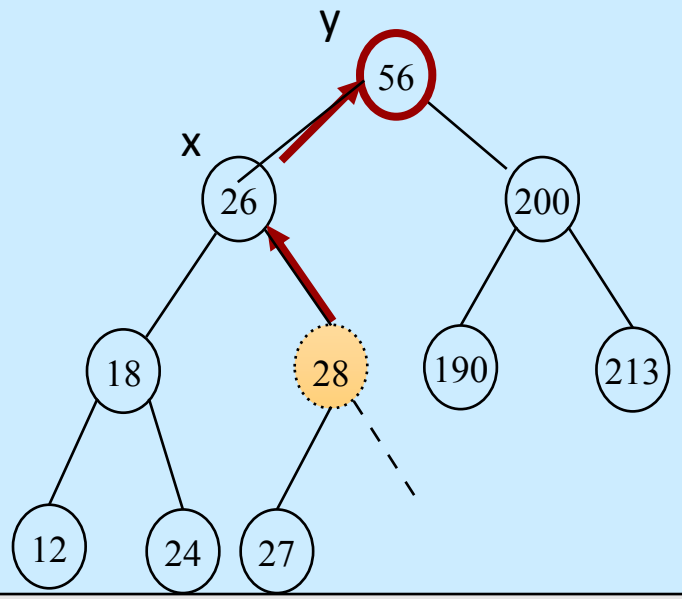
Tree-Successor(*x*)

```
1.  if right[x] ≠ NIL
2.      then return Tree-Minimum(right[x])
3.  y ← p[x]
4.  while y ≠ NIL and x = right[y]
5.      do x ← y
6.          y ← p[y]
7.  return y
```

Code for ***predecessor*** is symmetric.

Running time: $O(h)$

Example: successor of 28



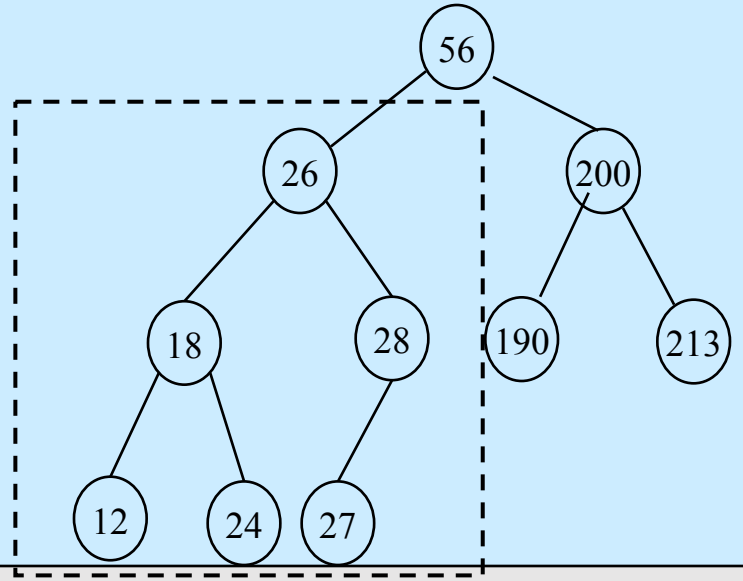
Lowest node whose left child is an ancestor of x.

Pseudo-code for Predecessor

Tree-Successor(*x*)

1. **if** *left*[*x*] ≠ *NIL*
2. return Tree-Maximum(*left*[*x*])
3. y ← *p*[*x*]
4. **while** *y* ≠ *NIL* **and** *x* = *left*[*y*]
5. *x* ← *y*
6. *y* ← *p*[*y*]
7. **return** *y*

Example: predecessor of 56

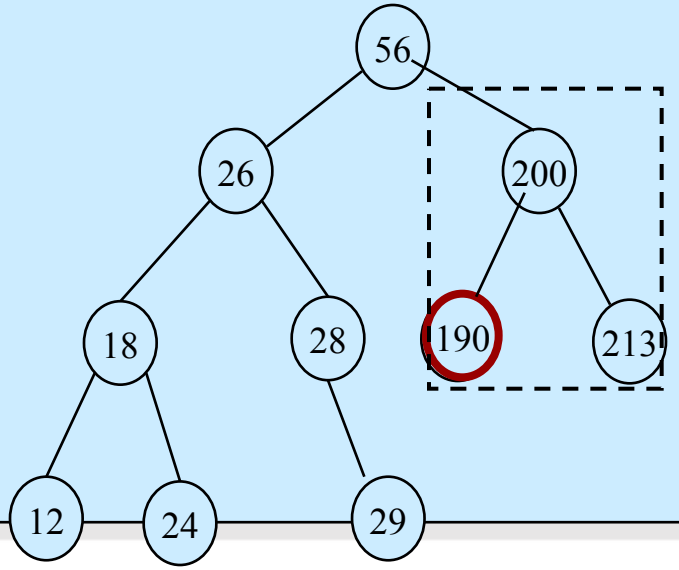


Pseudo-code for Predecessor

Tree-Successor(*x*)

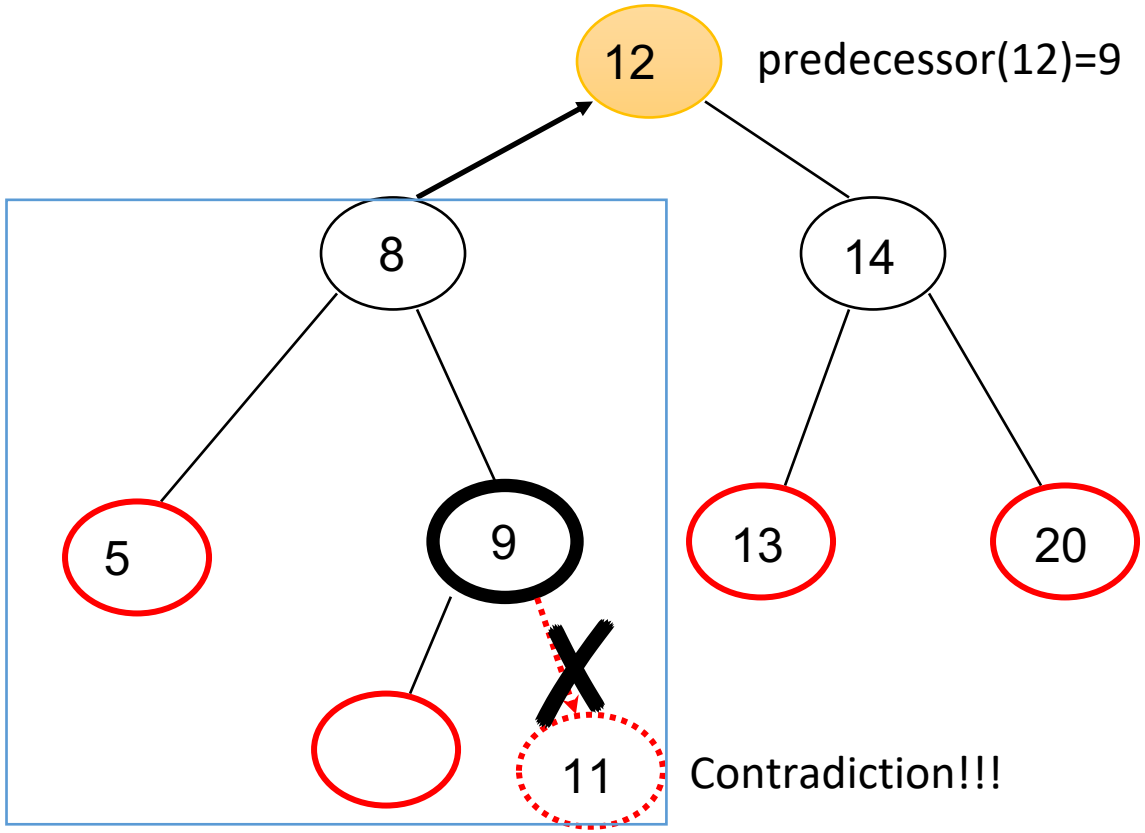
```
1.   if left[x] ≠ NIL
2.         return Tree-Maximum(left[x])
3.   y ← p[x]
4.   while y ≠ NIL and x = left[y]
5.         x ← y
6.         y ← p[y]
7.   return y
```

Example: successor of 190



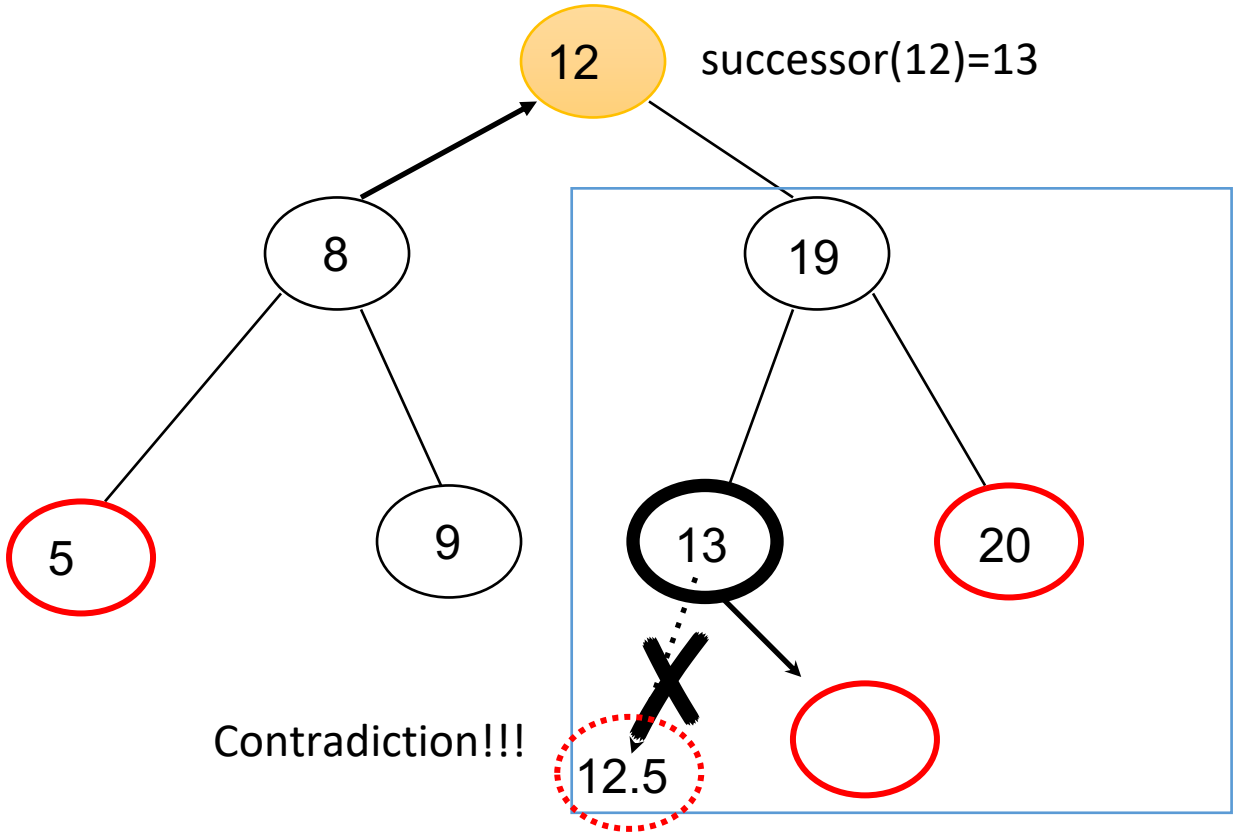
Predecessor Has At Most 1 Child

Predecessor of any node must have no right child. Why?



Successor Has At Most 1 Child

The Successor of any node must have no left child. Why?



Tree-Delete (T, z)

if *z* has no children ♦ case 0

remove *z*

if *z* has one child ♦ case 1

make *p*[*z*] point to child

if *z* has two children (subtrees) ♦ case 2

swap *z* with its successor

perform case 0 or case 1 to delete it (How about case 2?)

⇒ TOTAL: $O(h)$ time to delete a node

Deletion – Pseudocode

```
Tree-Delete(T, z)
/* Determine which node to splice out: either z or z's successor y. */
1.    if left[z] = NIL or right[z] = NIL
2.       then y ← z                          // case 0 or 1
3.    else y ← Tree-Successor[z]   // case 2
/* Set x to a non-NIL child of y, or to NIL if y has no children. */
4.   if  left[y] ≠ NIL
5.       then x ← left[y]
6.   else x ← right[y]
/* y is removed from the tree by manipulating pointers of  p[y] and x */
7.   if x ≠ NIL
8.      then p[x] ← p[y]                  // re-connect y's child x to y's parent
/* Continued on next slide */
```

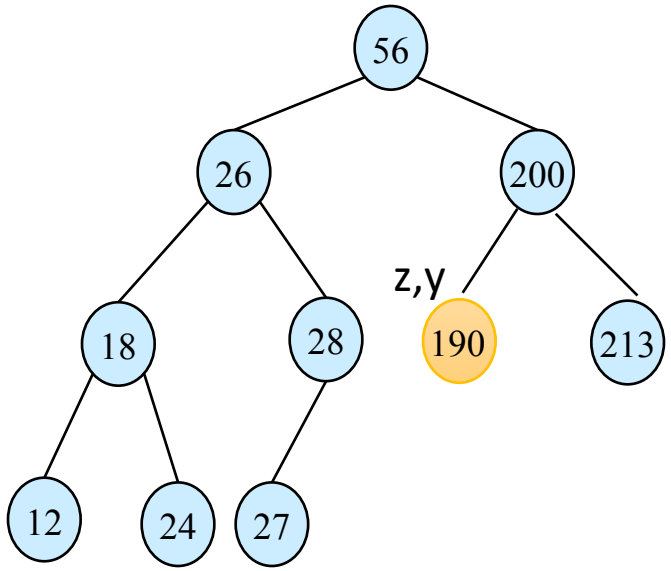
Deletion – Pseudocode

Tree-Delete(*T*, *z*) (Contd. from previous slide)

```
9.    if p[y] = NIL
10.          then root[T] ← x
11.   else if  left[p[y]] = y  // re-connect the child link of deleted node y's parent
12.          then left[p[y]] ← x
13.   else       right[p[y]] ← x
/* If z's successor y was spliced out, copy its data into z */
14.  if y ≠ z
15.      then  key[z] ← key[y]      /* copy y's satellite data into z. */
16.  return y
```

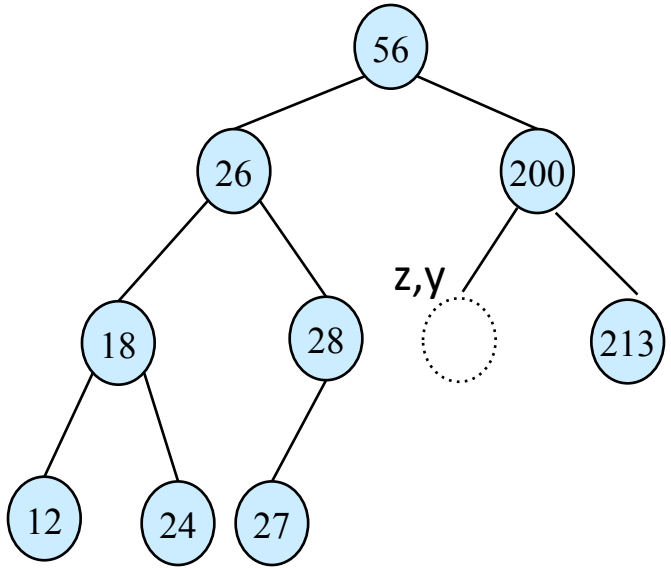
Case 0

- z has no children
- e.g. delete 190



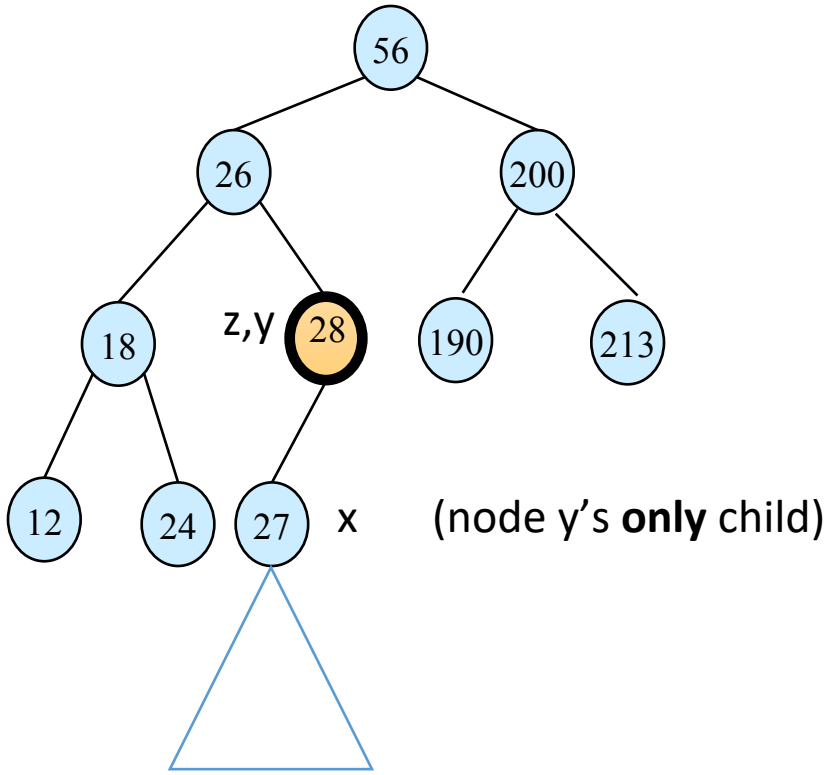
Case 0

- z has no children
- e.g. delete 190



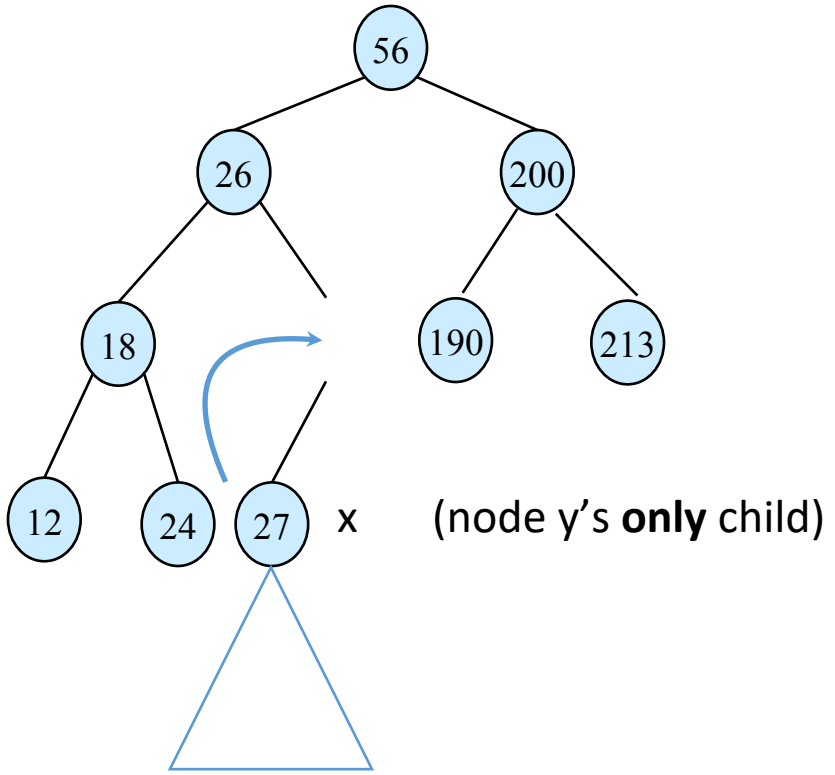
Case 1

- z has only one child
- e.g. delete 28



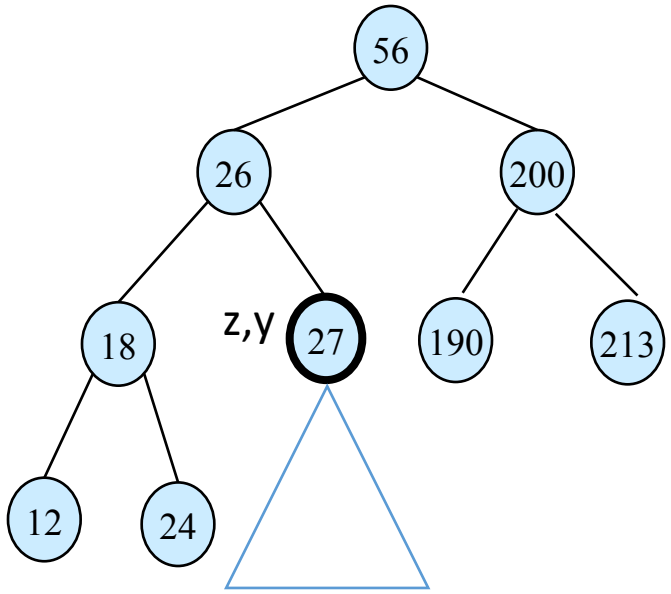
Case 1

- z has only one child
- e.g. delete 28



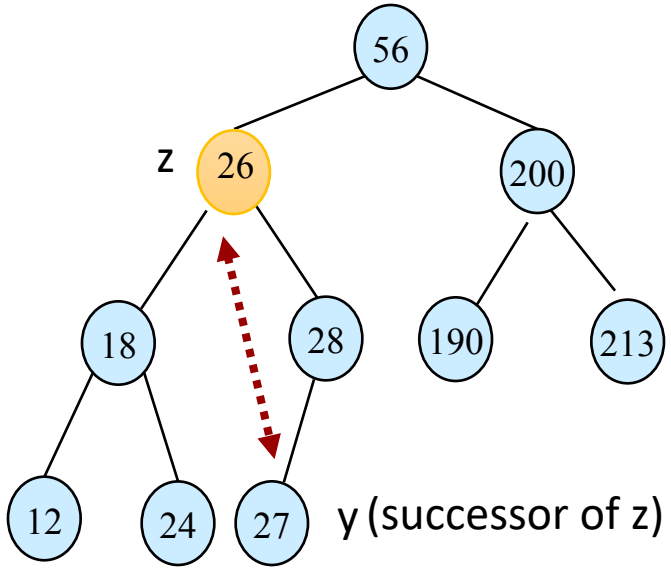
Case 1

- z has only one child
- e.g. delete 28



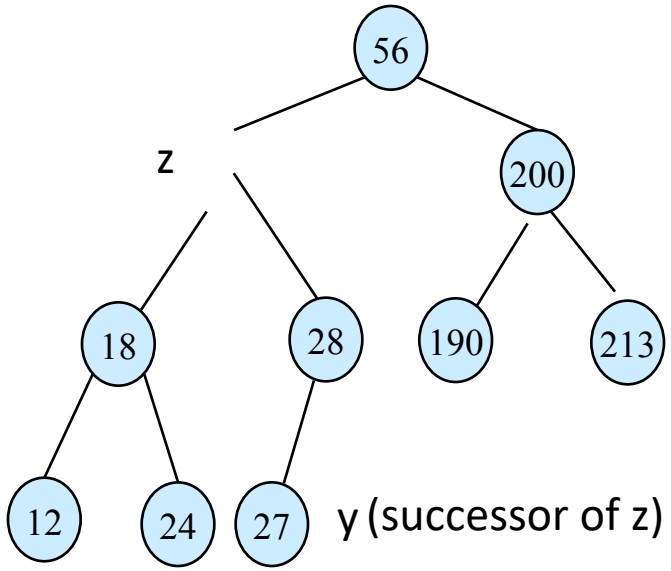
Case 2-1

- z has two children, y has no children
- e.g. delete z=26, where y=27



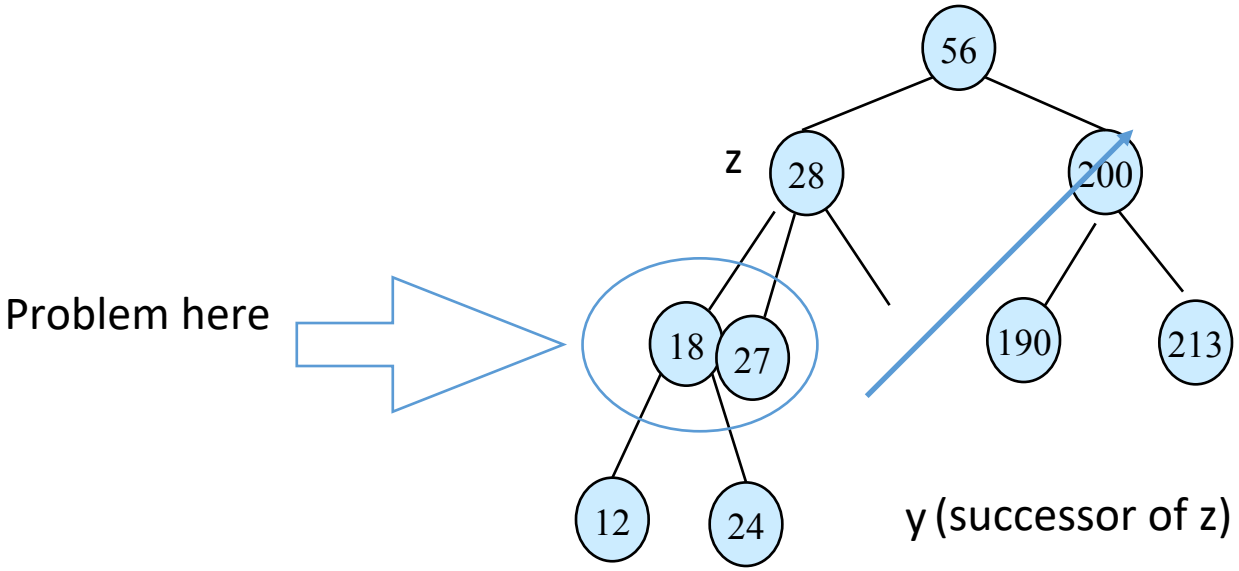
Case 2-1

- z has two children, y has no children
- e.g. delete z=26, where y=27



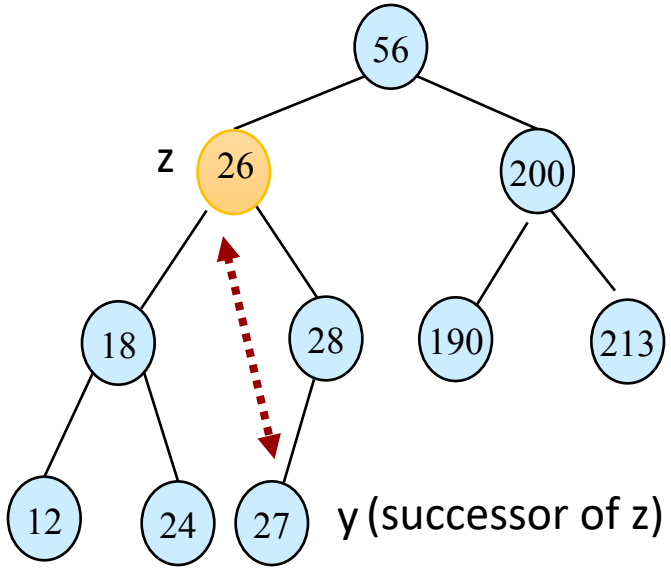
Case 2-1

- z has two children, y has no children
- e.g. delete z=26, where y=27



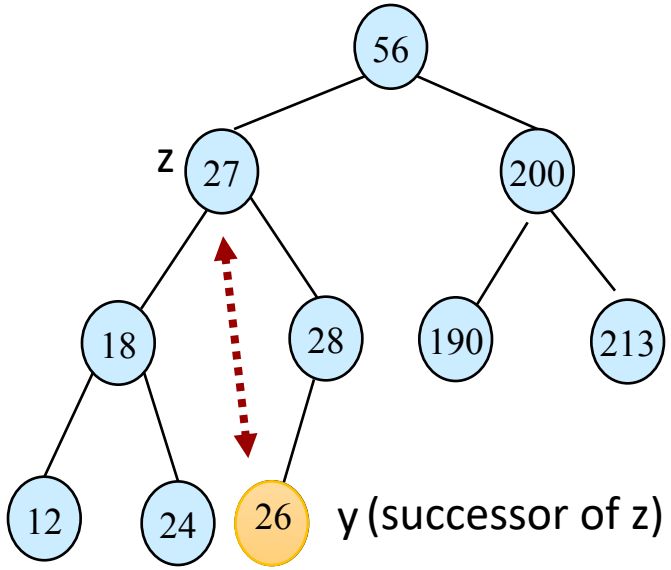
Case 2-1

- z has two children, y has no children
- e.g. delete z=26, where y=27



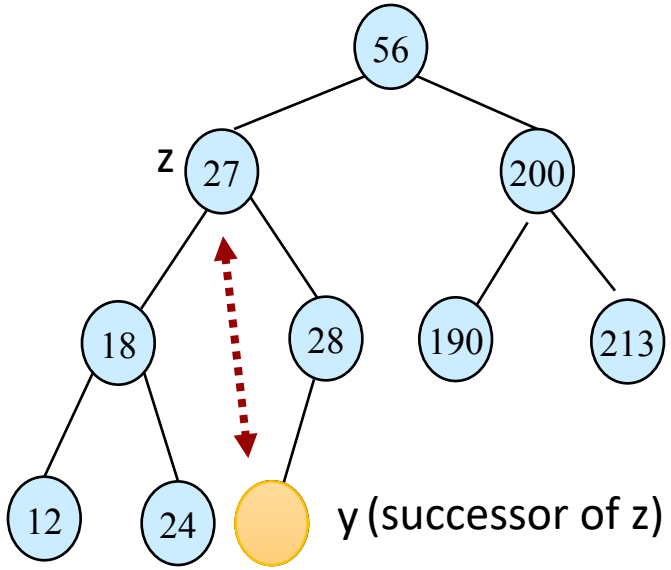
Case 2-1

- z has two children, y has no children
- e.g. delete z=26, where y=27



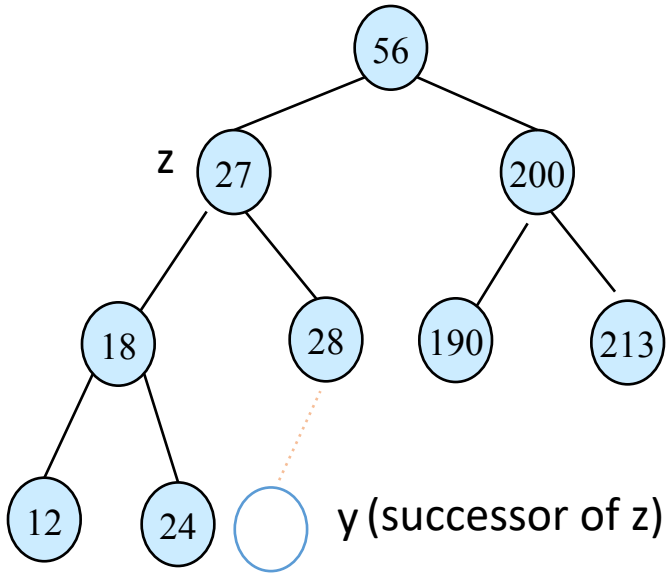
Case 2-1

- z has two children, y has no children
- e.g. delete z=26, where y=27



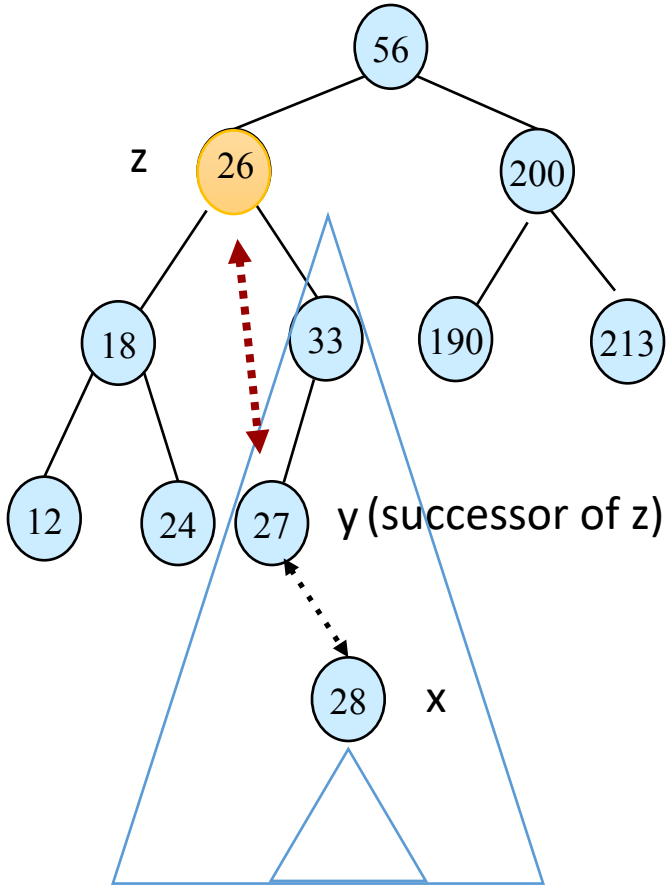
Case 2-1

- z has two children, y has no children
- e.g. delete z=26, where y=27



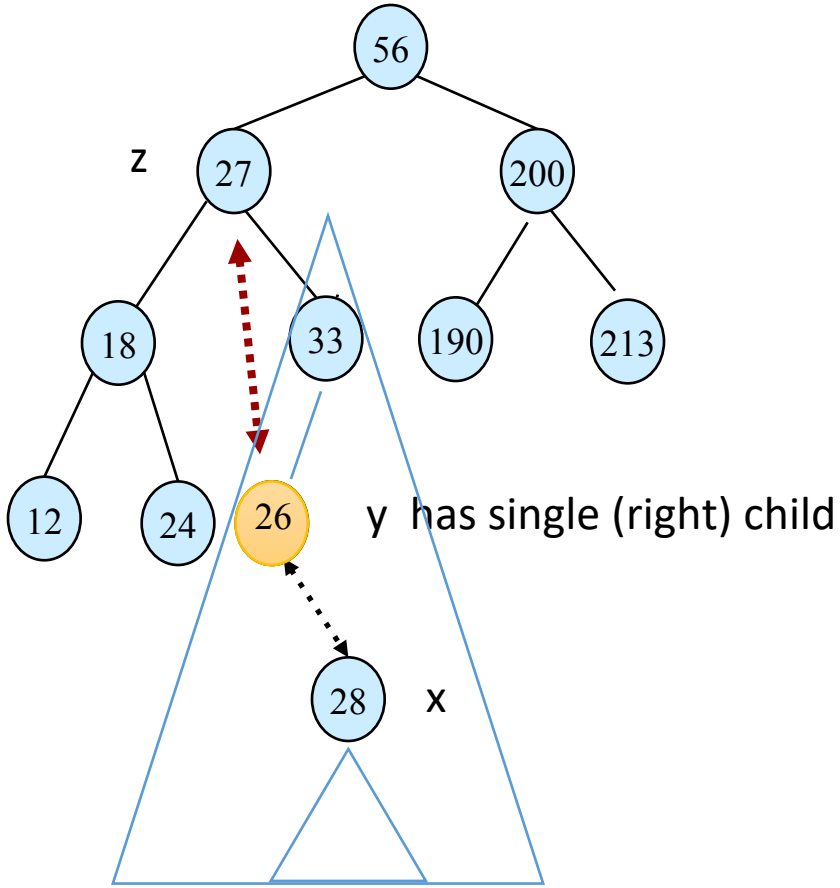
Case 2-2

- z has two children, z's successor y has single (right) child
- e.g. delete z=26, where y=27



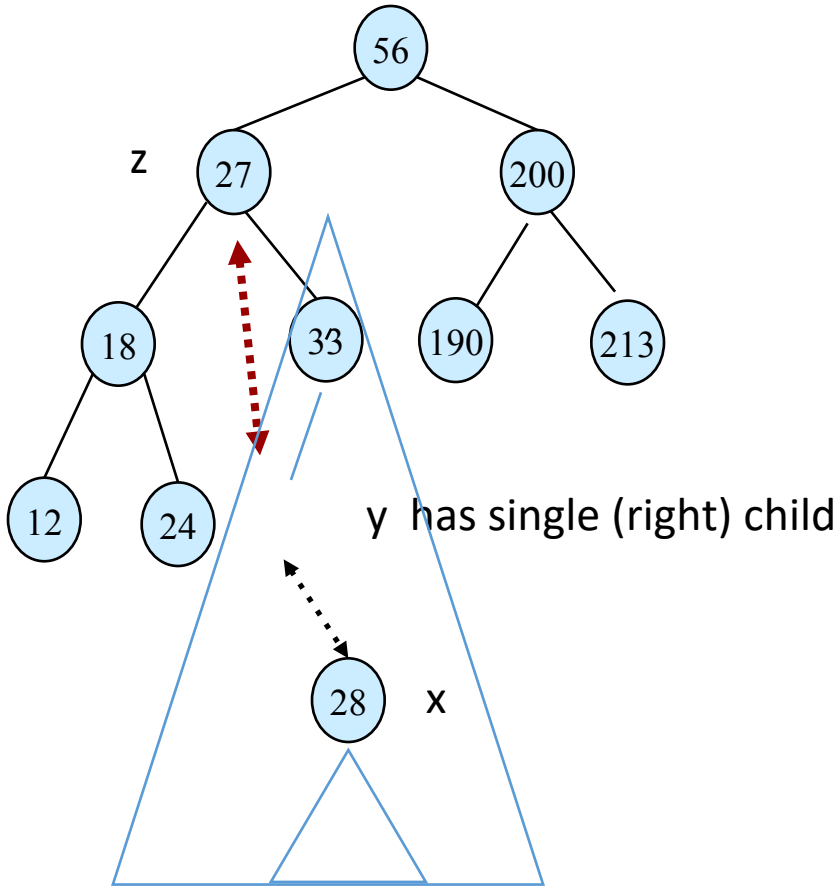
Case 2-2

- z has two children, z's successor y has single (right) child
- e.g. delete z=26, where y=27



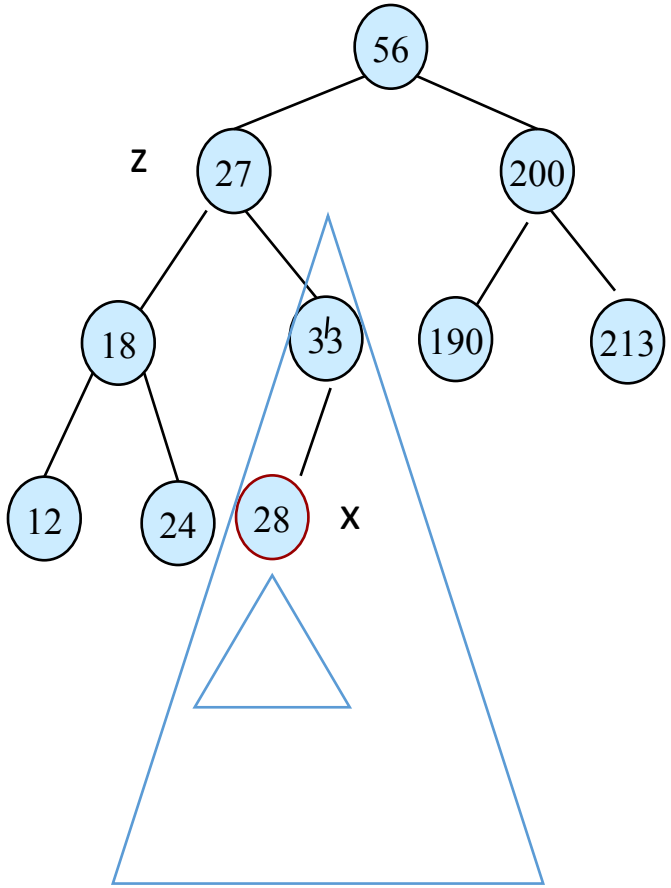
Case 2-2

- z has two children, z's successor y has single (right) child
- e.g. delete z=26, where y=27



Case 2-2

- z has two children, z's successor y has single (right) child
- e.g. delete z=26, where y=27

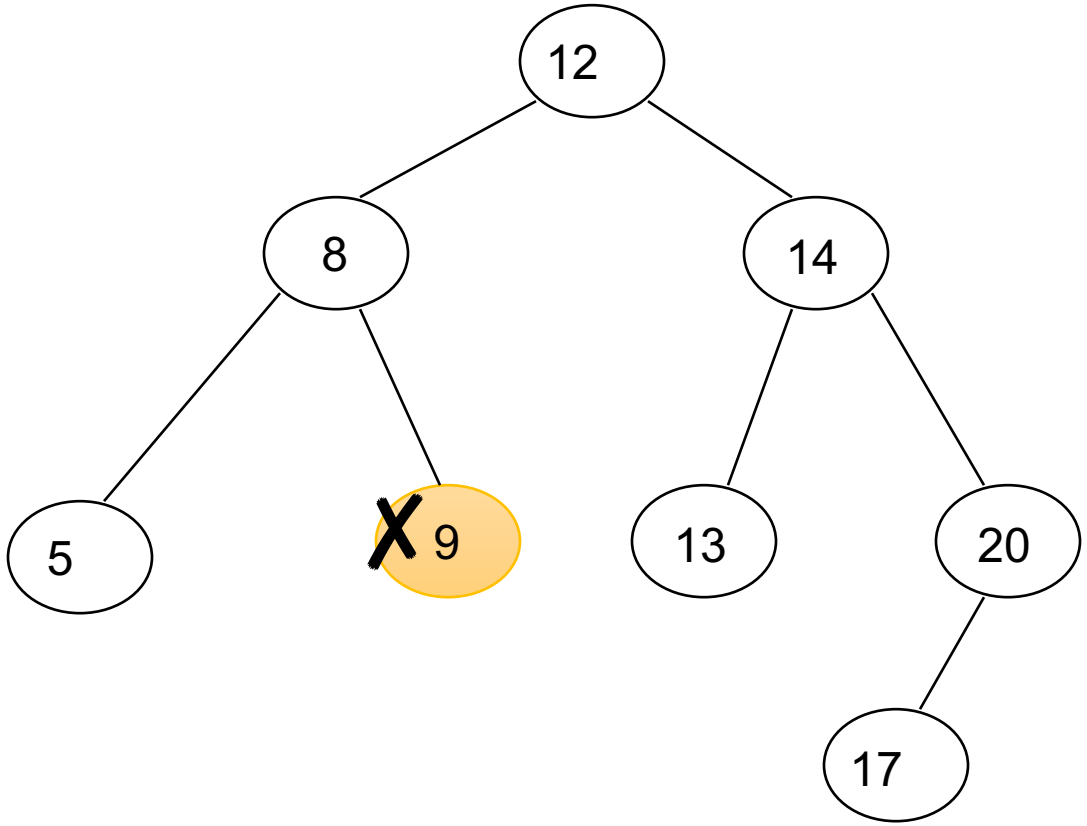


Correctness of Tree-Delete

- How do we know case 2 should go to case 0 or case 1 instead of back to case 2?
  - Because when *z* has 2 children, its successor has at most 1 child. (See Proof Earlier.)
- Equivalently, we could swap with predecessor instead of successor. It might be good to alternate to avoid creating lopsided tree.

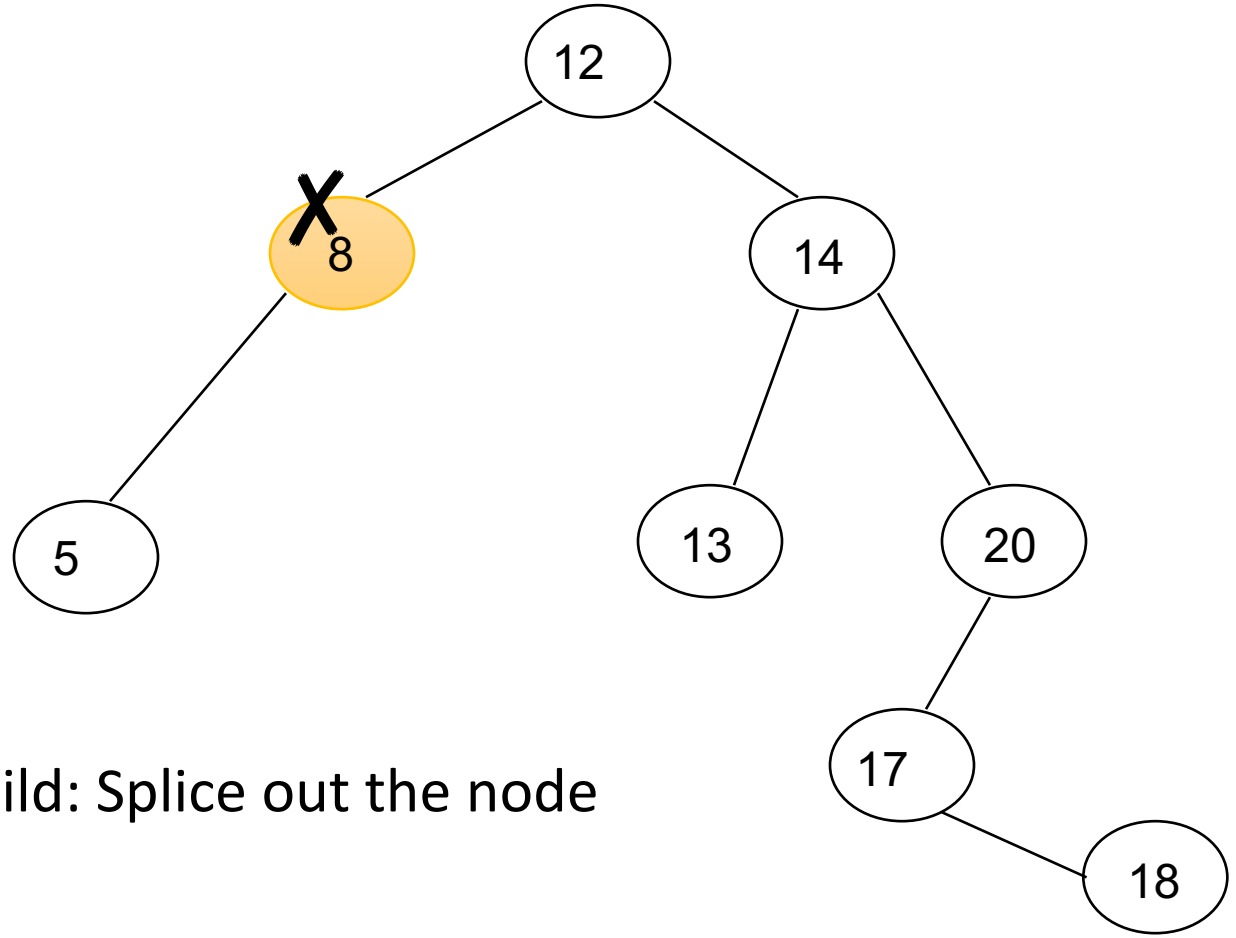
Another Deletion Example

Case 0



Another Deletion Example

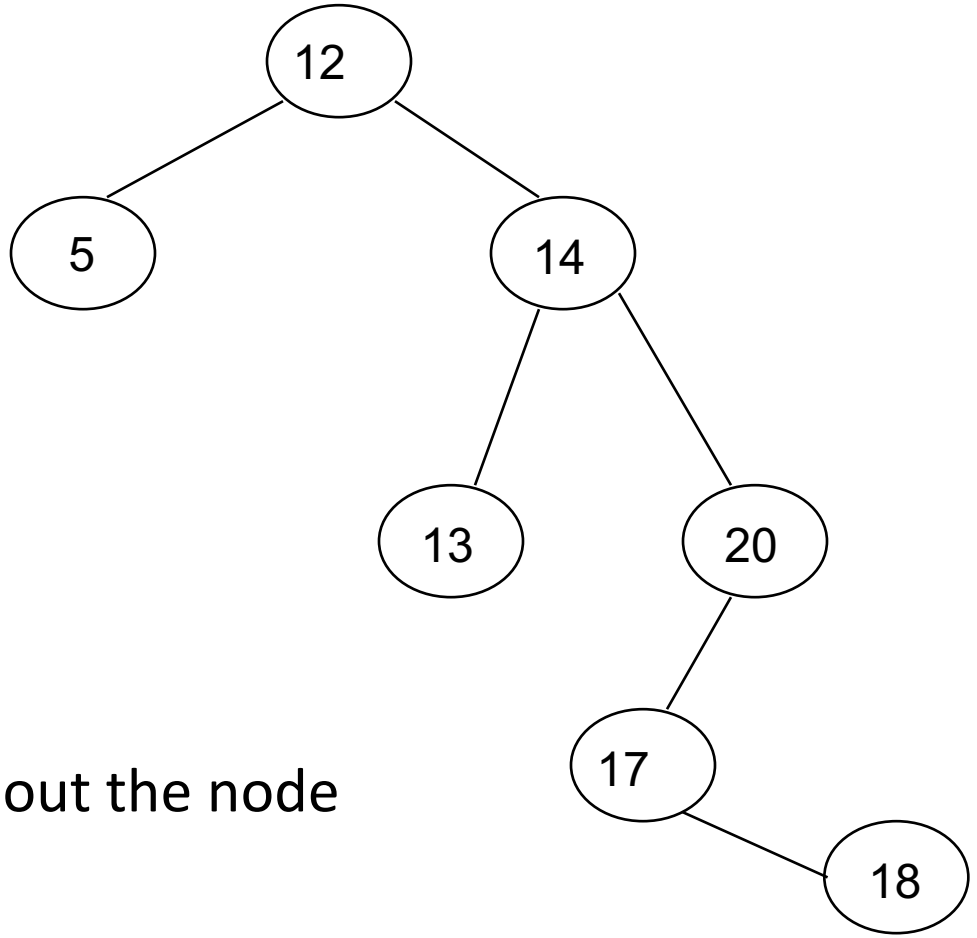
Case 1



One child: Splice out the node

Another Deletion Example

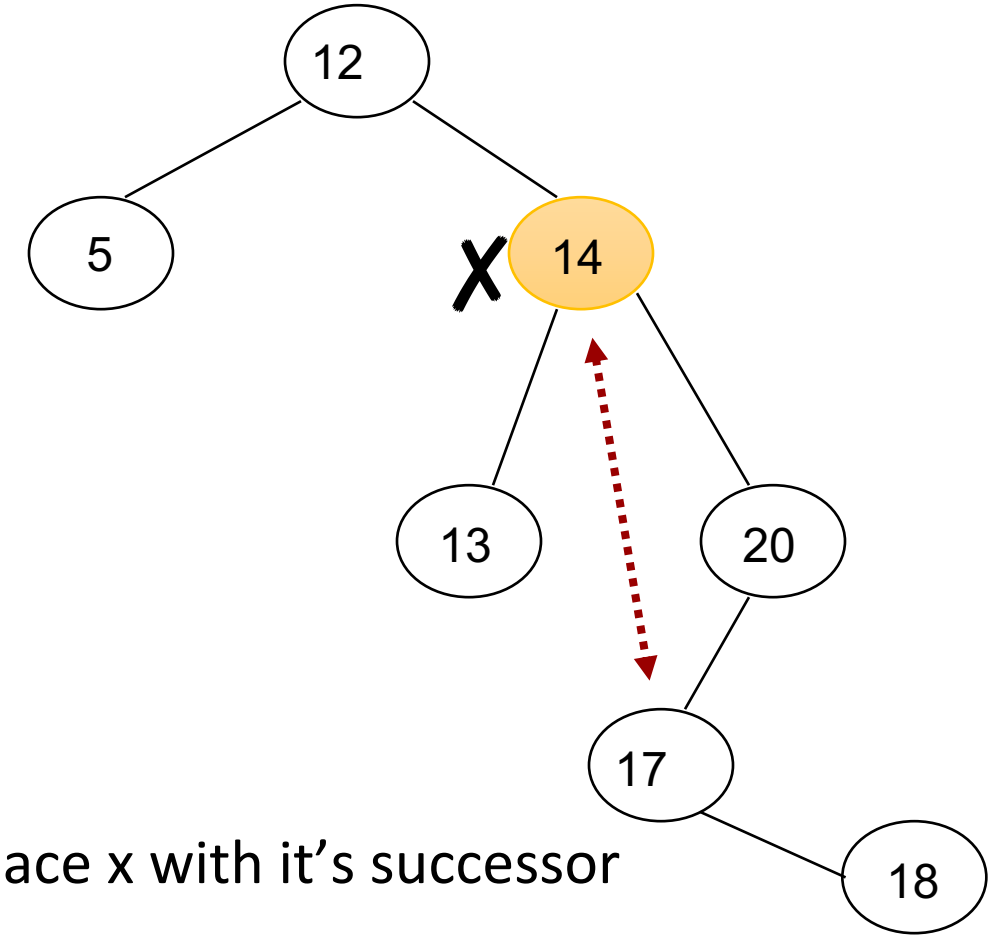
Case 1



One child: Splice out the node

Another Deletion Example

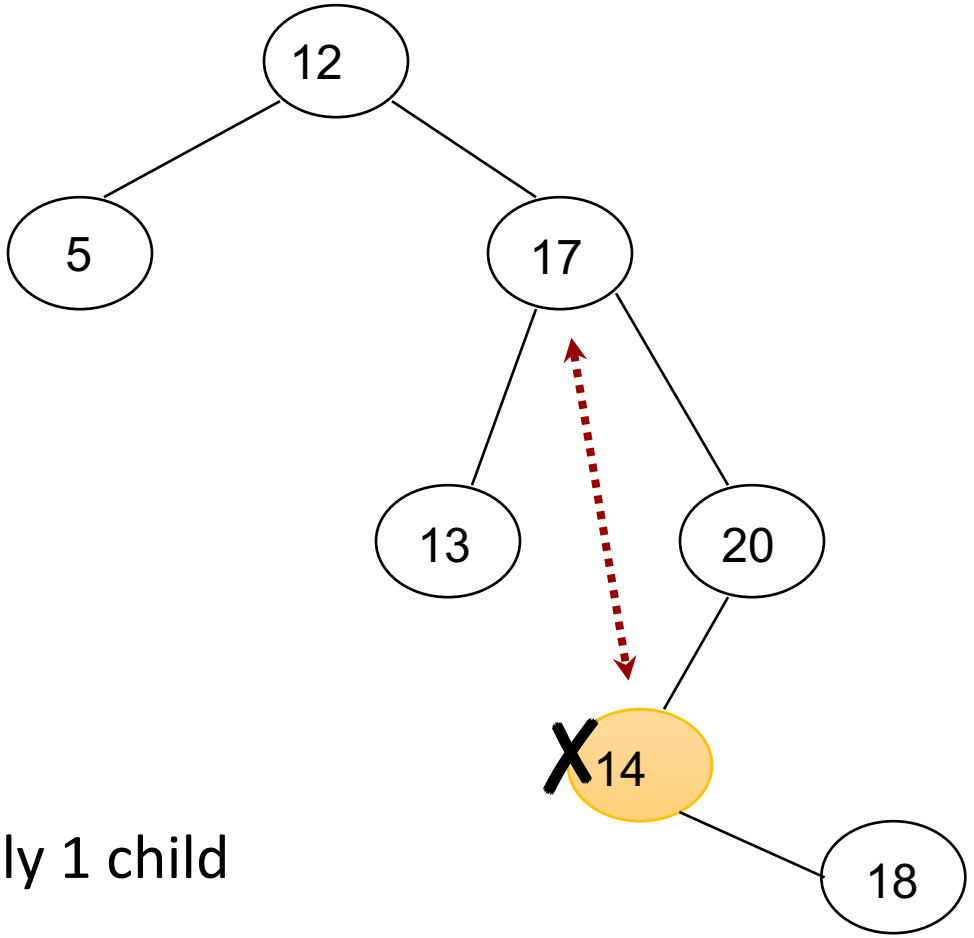
Case 2-2



Two children: Replace x with it's successor

Another Deletion Example

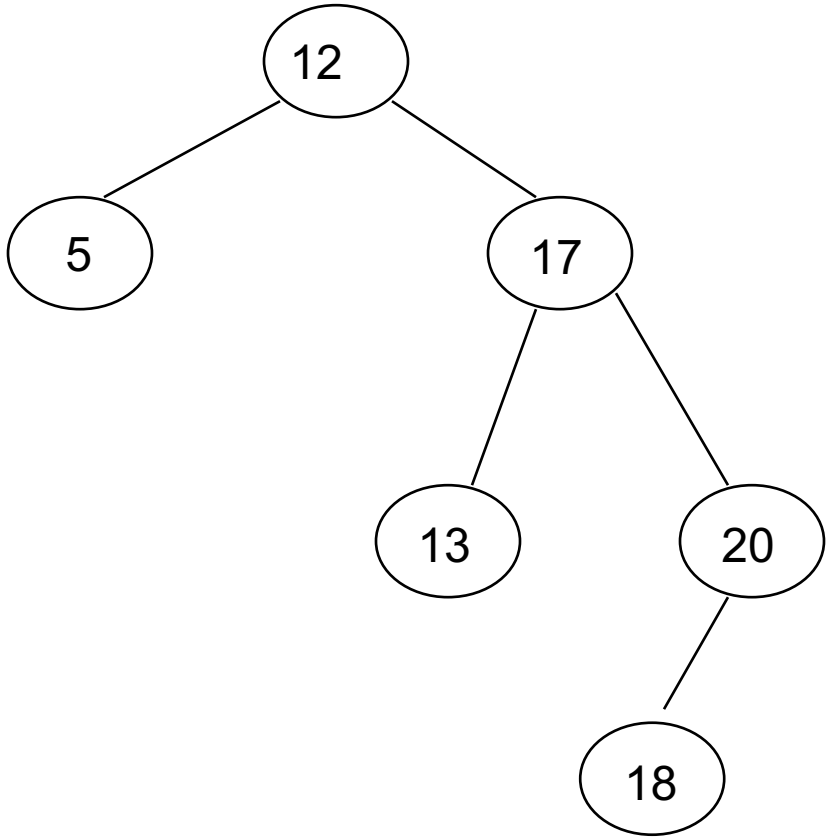
Case 2-2



Now node 14 has only 1 child

Another Deletion Example

Case 2-2



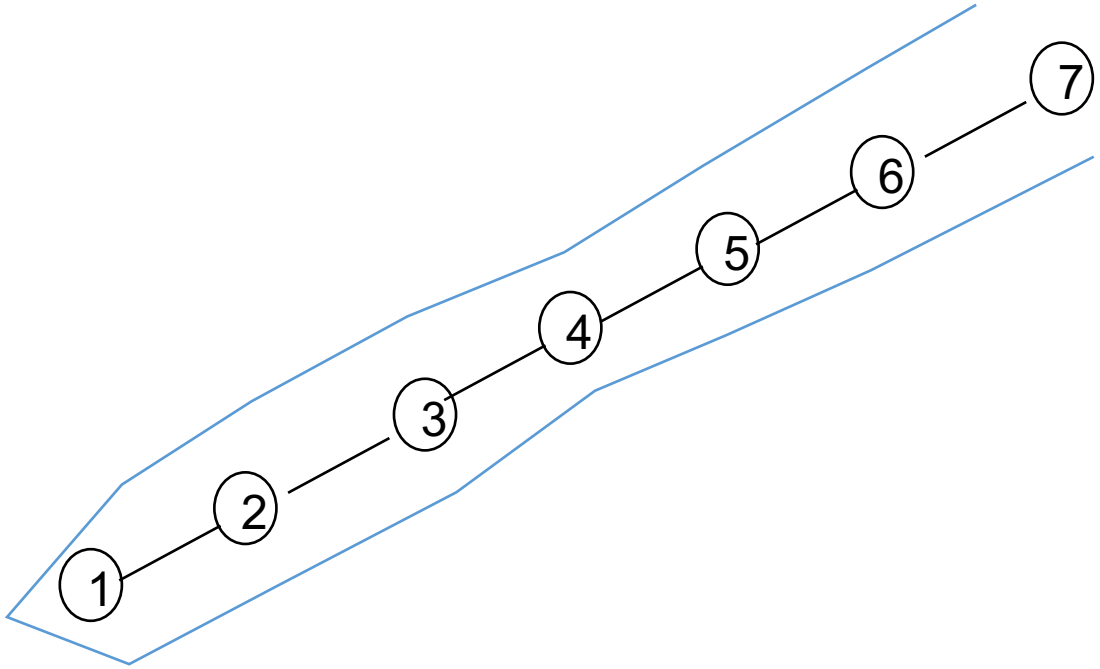
Node 14 is gone!

When to use BST or Heap?

- Heap is better at findMin/findMax (O(1)), while
- BST is good at searches (O(lg N)).
- Insert is O(lg N) for both structures.
- If you only care about findMin/findMax (e.g. priority-related), go with heap (simple implementation)
- If you want everything sorted, go with BST
  - more complicate implementation
  - tree height is NOT necessarily balanced O(lg N)

Worst case of BST construction

Worst case: 7,6,5,4,3,2,1 ==> The highest binary search tree

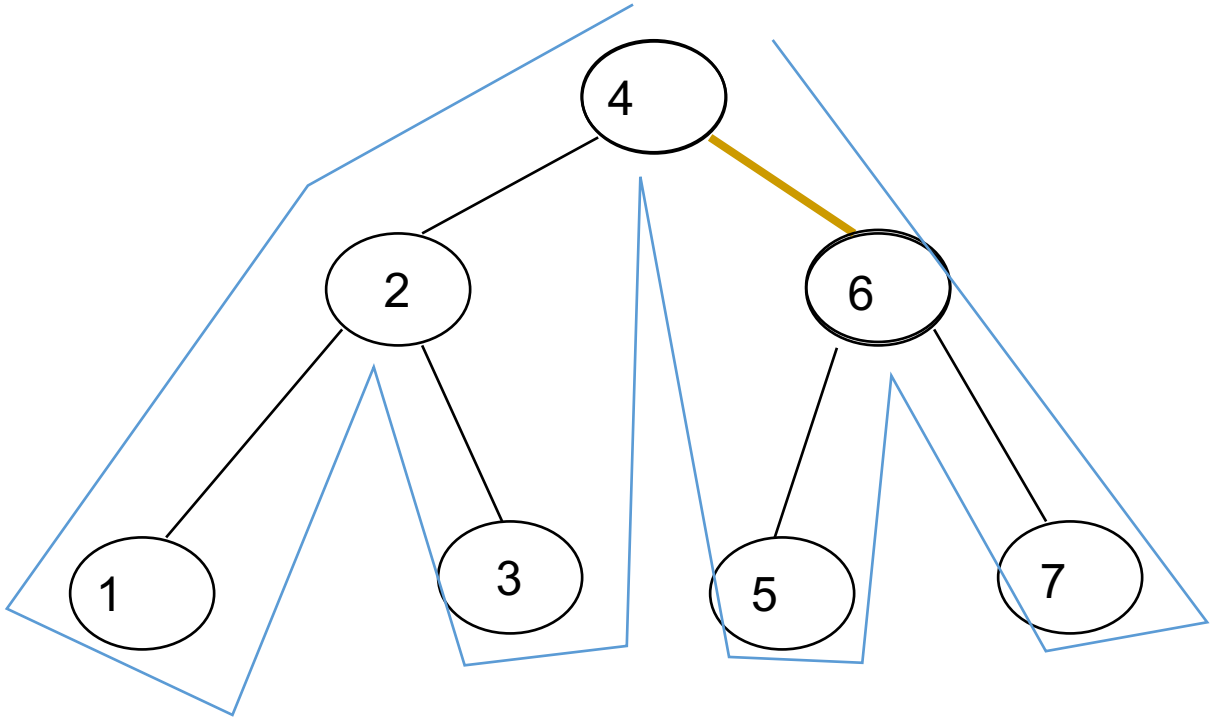


Another Worst case of BST construction: 1,2,3,4,5,6,7 ==> The highest binary search tree

Best case of BST construction

4, 2,1,3,6,5,7 ==> the shortest binary search tree

4,2,6,1,3,5,7 ==> the shortest binary search tree



Exercise

Suppose that we have numbers between 1 and 1000 in a binary search tree, and we want to search for the number 363. Which of the following sequences could *not* be the sequence of nodes examined?

a) 2, 252, 401, 398, 330, 344, 397, 363.

b) 924, 220, 911, 244, 898, 258, 362, 363.

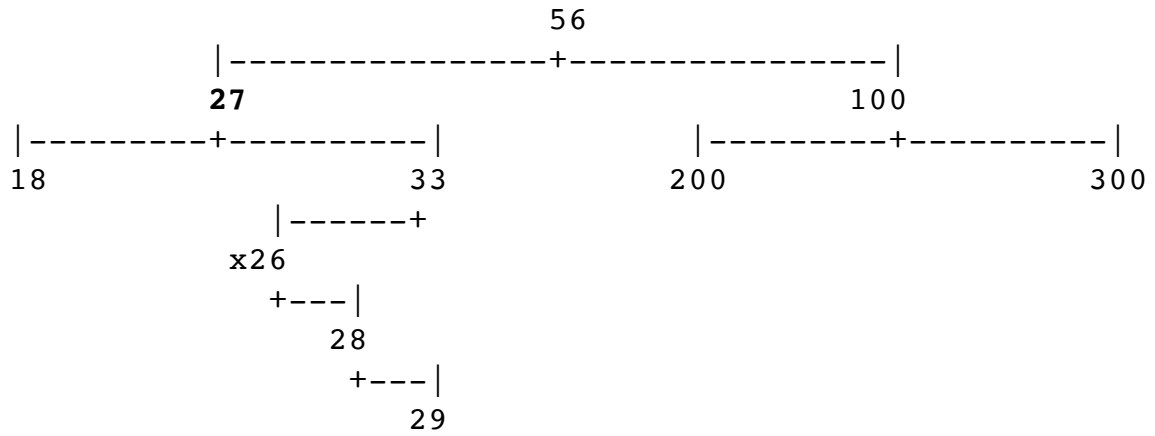
c) 925, 202, 911, 240, 912, 245, 363.

d) 2, 399, 387, 219, 266, 382, 381, 278, 363.

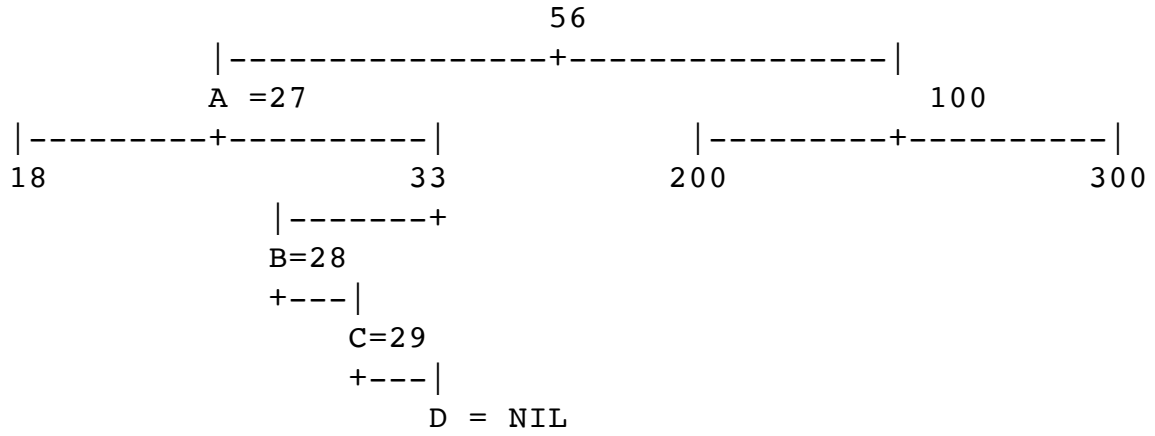
e) 935, 278, 347, 621, 299, 392, 358, 363.

Quiz Preview

Q#27: Consider the following Binary Search Tree (BST):

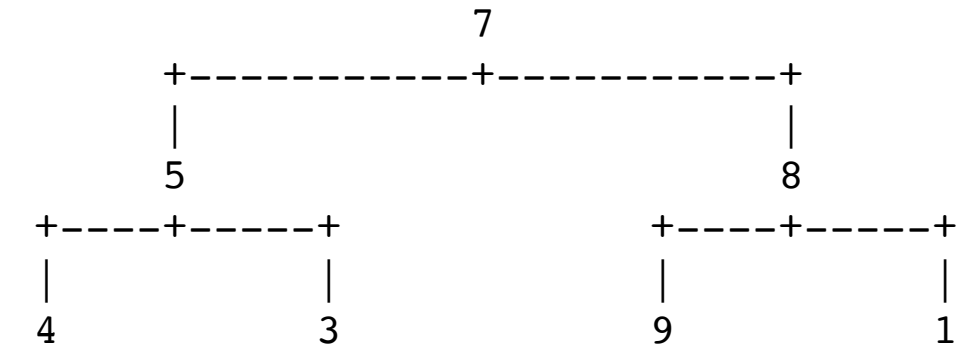


After the deletion of node Z=26, what will the new BST look like? In particular, what are the values of nodes A,B, and D.



Quiz Preview

Q#1: Using the following binary tree (not a binary SEARCH tree)



i. 4,5,3,7,9,8,1
ii. 7,5,4,3,8,9,1
iii. 4,3,5,9,1,8,7
Which of the above sequence would be the outputs for Pre-order, In-order, and Post-order traversal

Answer: _____?_____(1pt)

Quiz Preview

Q#2. A tree sort is a sort algorithm that builds a binary search tree from the elements to be sorted (using TREE-INSERT repeatedly to insert the numbers one by one), and then traverses the tree (in-order) so that the elements come out in sorted order.

What are the worst-case and best-case running times for this sorting algorithm?

a) worst-case: $O(n^2)$, best-case: O(n log n)
b) worst-case: O(n), best-case: O(log n)
c) worst-case: O(n), best-case: O(n)
d) worst-case: O(n log n), best-case: O(n log n) Answer:_____?_____ (1pt)

Q#3. Which of the following statements are correct about the height of binary search tree for the given input sequence of numbers? (Choose two).

a) The tree height for 7,6,5,4,3,2,1 will be the largest 7.
b) The tree height for 4, 2,1,3,6,5,7 will be the smallest, 3;
c) The tree height for 7,6,5,4,3,2,1 is between 3 and 7, excluding.
d) The tree height for 7,6,5,4,3,2,1 will be the smallest 3.
e) The tree height for 4, 2,1,3,6,5,7 will be the largest 7.

Answer:_____?_____ (1pt)

Quiz Preview

Q#4. Which of the following input sequence of 15 numbers will produce the shortest and highest binary search tree. (Choose two.)

a) 8 4,2,1,3,6,5,7, 12,10,9,11,14,13,15 will produce the shortest.
b) 1,2,3,4,5,6,7,8, 9,10,11,12,13,14,15 will produce the shortest.
c) 8 4,2,1,3,6,5,7, 12,10,9,11,14,13,15 will produce the highest.
d) 1,2,3,4,5,6,7,8, 9,10,11,12,13,14,15 will produce the highest.

Answer: ? (1pt)

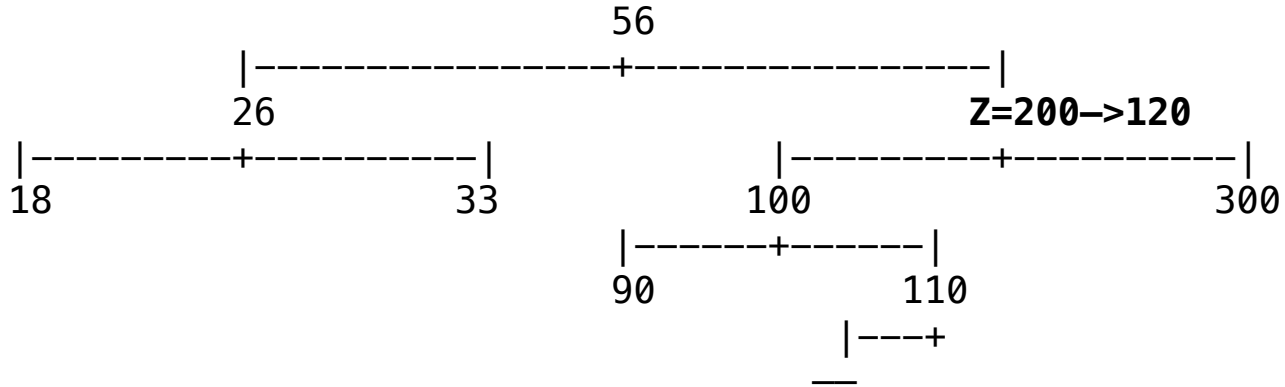
Q#5. Which of the following sequence of numbers will result in a different in-order traversal of BST than the others?

a) 1,2,3,4,5,6,7
b) 7,6,5,4,3,2,1
c) 4,2,1,3,6,5,7
d) 2,1,4,3,5,6,7
e) All of the above will have the same in-order traversal of BST.

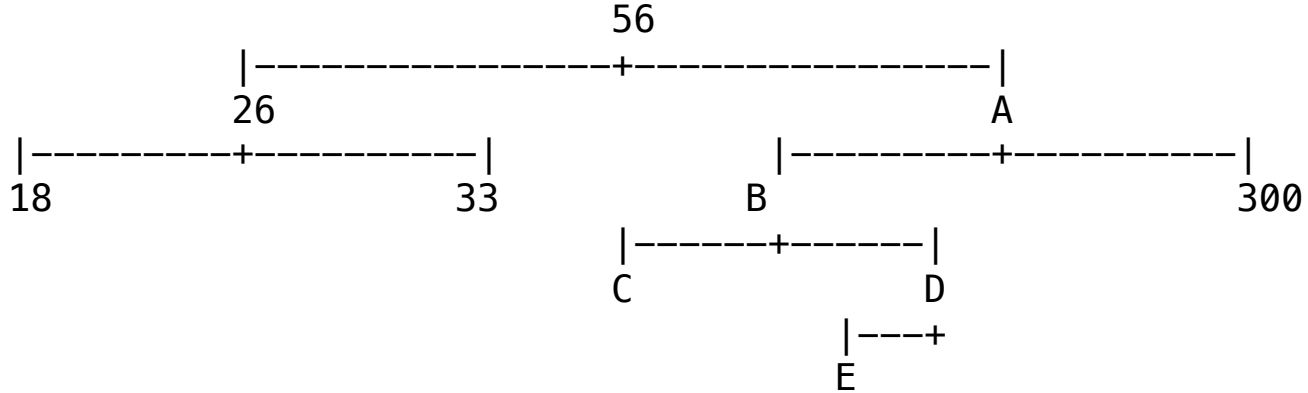
Answer: ? (1pt)

Quiz Preview

Q#6: Consider the following Binary Search Tree (BST):



After the deletion of node Z=200, what will the new BST look like? In particular, what are the values of nodes A,B, D, and E?



Which of the following is the correct about the values of nodes A,B,C,D and E ?

Answer: ? (1pt)