INFS7901
Database Principles

Binary Search Trees

Rocky Chen

Notes

- Project:

  You can start working on Part 2 – due 26 May

- RiPPLE:

  3rd round due next week (28 Apr)

- Don't forget about the tutorials!

Dictionary ADT

Tree Terminology

Binary Search Trees (BSTs)

Insertion and Deletion in BSTs

Dictionary Abstract Data Type (ADT)

- Dictionary operations
  - create
  - destroy
  - insert
  - find
  - delete

insert

- *Indonesia*
  - *Jakarta*

- Australia
  - Canberra
- China
  - Beijing
- India
  - New Delhi

find(Australia)

- *Australia*
  - *Canberra*

- Stores *values* associated with user-specified *keys*
  - values may be any (homogenous) type
  - keys may be any (homogenous) comparable type

Search/Set ADT

- Dictionary operations
  - create
  - destroy
  - insert
  - find
  - delete
- Stores keys
  - keys may be any (homogenous) comparable type
  - quickly tests for membership

insert

- *Japan*

- Australia
- China
- India
- Indonesia

find(Malaysia)

NOT FOUND

Naïve Implementations

- Unsorted Array

2	4	11	98	37	44	3			

Operation	Implementation	Running Time
Insert	Add after the current last value, if there is space available.	O(1)
Find	Scan the entire array	O(n)
Delete (index known)	Delete item, fill empty space with the last element	O(1)
Delete (value known)	Scan the entire array to find the item then delete	O(n)

Naïve Implementations

- Sorted Array

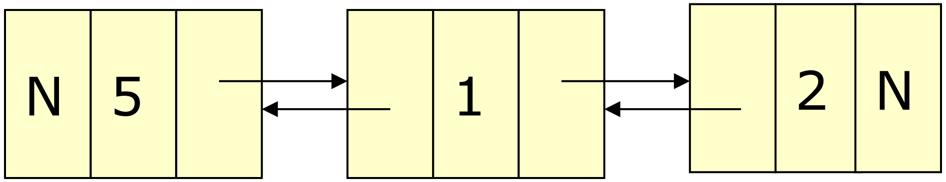
2	3	4	11	37	44	98			

Operation	Implementation	Running Time
Insert	Shift to make room for the insertion	O(n)
Find	Use binary search	O(log n)
Delete (index known)	Shift to get rid of empty space	O(n)
Delete (value known)	Use binary search + shift	O(n)

Linked Lists

- Consider the following <u>abstraction</u>, picturing a short linked list:

N represents NULL

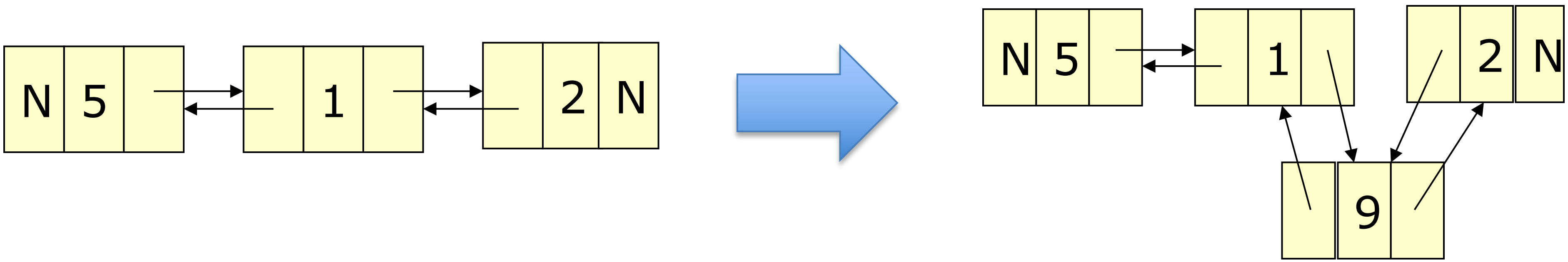


- What might it look like in memory?

	data	previous	next
1080	2	100	Null
…			
600	5	Null	100
…			
140			
120			
100	1	600	1080

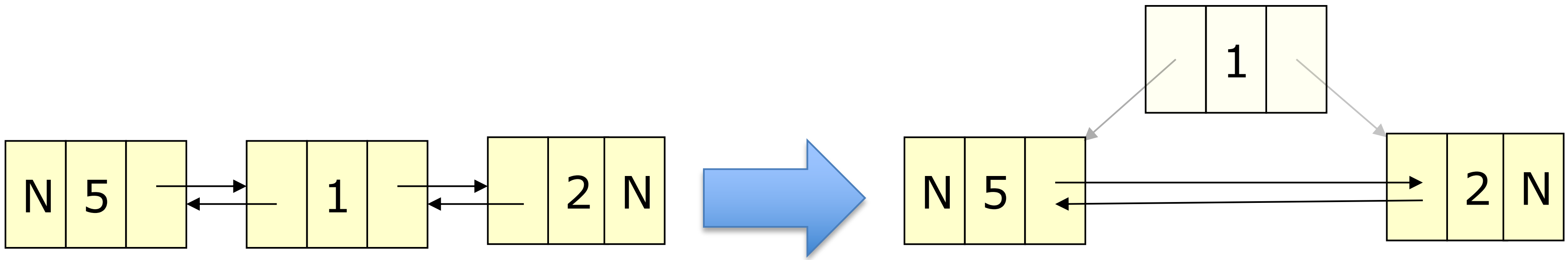
Node

Inserting an Element to a Linked List



	data	*previous*	*next*
1080	2	140	Null
…			
600	5	Null	100
…			
140	9	100	1080
120			
100	1	600	140

Removing an Element from a Linked List

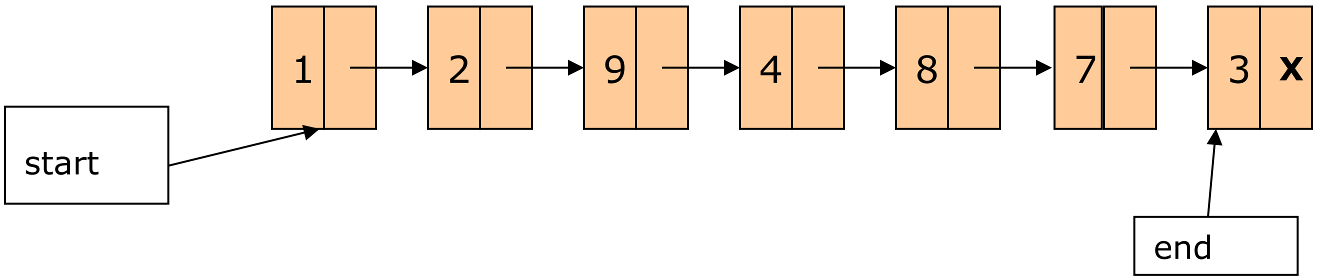


	data	previous	next	
1080	2	600	Null	
…				
600	5	Null	1080	
…				
140				
120				
100	1	600	1080	← *delete*

https://visualgo.net/en/list

Naïve Implementations

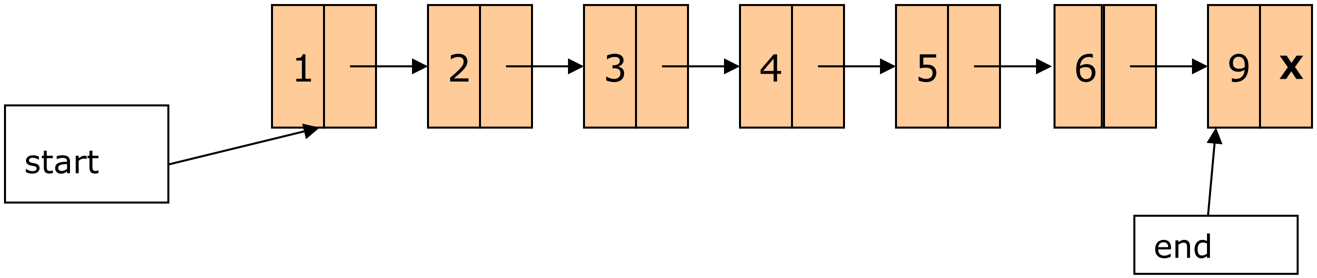
- Unsorted linked list



Operation	Implementation	Running Time
Insert	Add after end	O(1)
Find	Scan list	O(n)
Delete (Address known)	Remove and rechain	O(1)
Delete (value known)	Scan list + remove and rechain	O(n)

Naïve Implementations

- Sorted linked list

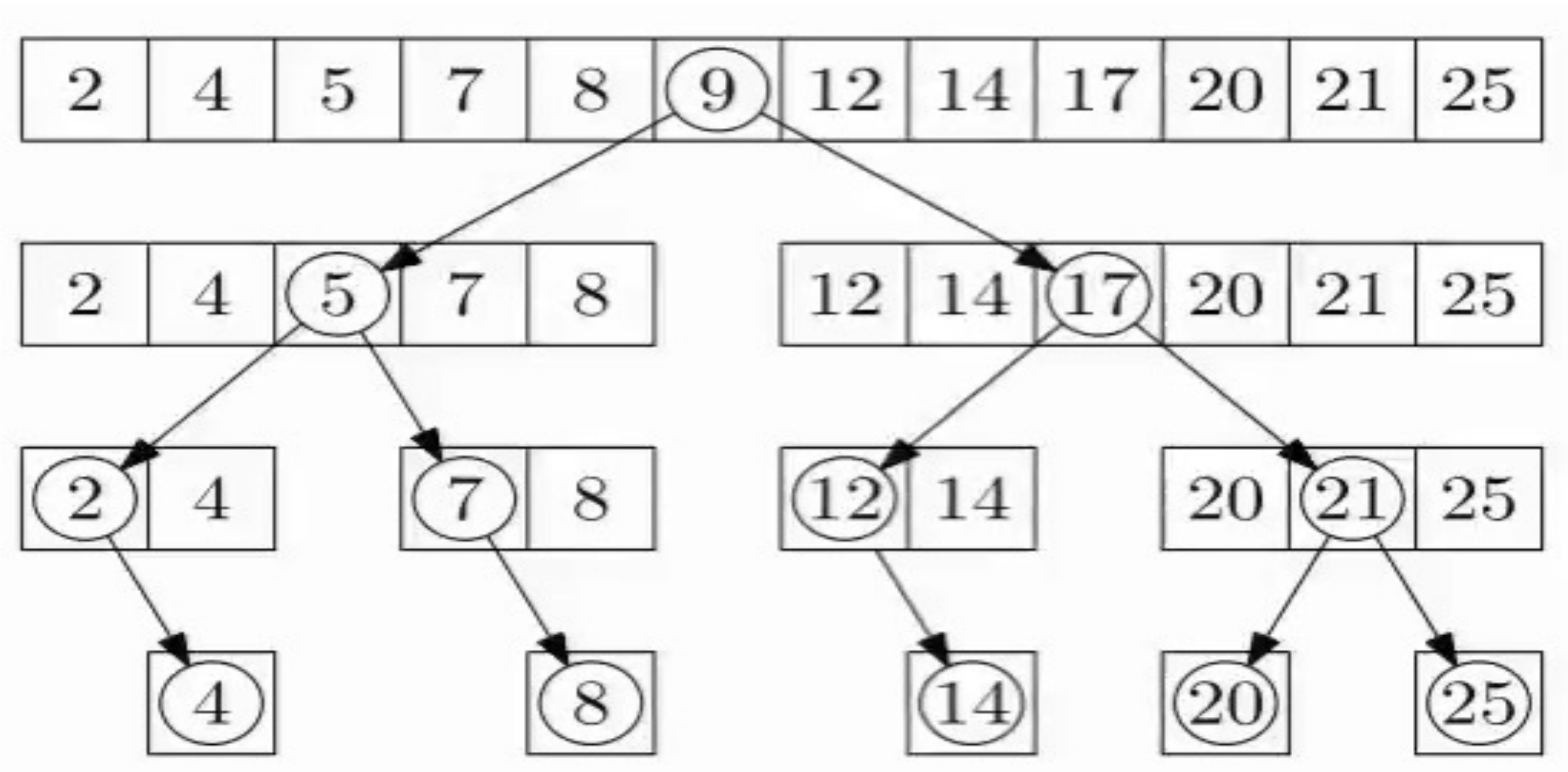


Operation	Implementation	Running Time
Insert	Scan list to find where to insert	O(n)
Find	Scan list	O(n)
Delete (Address known)	Remove and rechain	O(1)
Delete (value known)	Scan list + remove and rechain	O(n)

Naïve Implementations

	insert	*find*	*delete* *By value*	*delete* *By address*
• Linked list				
– Unsorted	$O(1)$	$O(n)$	$O(n)$	$O(1)$
– Sorted	$O(n)$	$O(n)$	$O(n)$	$O(1)$
• Array				
– Unsorted	$O(1)$	$O(n)$	$O(n)$	$O(1)$
– Sorted	$O(n)$	$O(\lg n)$	$O(n)$	$O(n)$
Can we do better?	O(lg n)	O(lg n)	O(lg n)	O(lg n)

From Binary Search to Binary Search Trees



General idea of BST (informally): You are allowed to shove new nodes into your list without having to shift things over.

Dictionary ADT

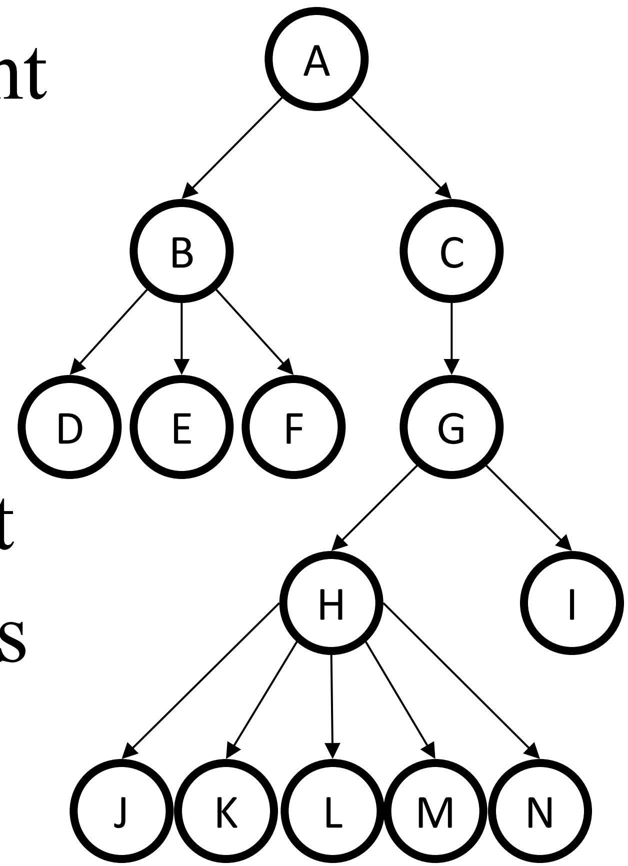
Tree Terminology

Binary Search Trees (BSTs)

Insertion and Deletion in BSTs

Tree Terminology

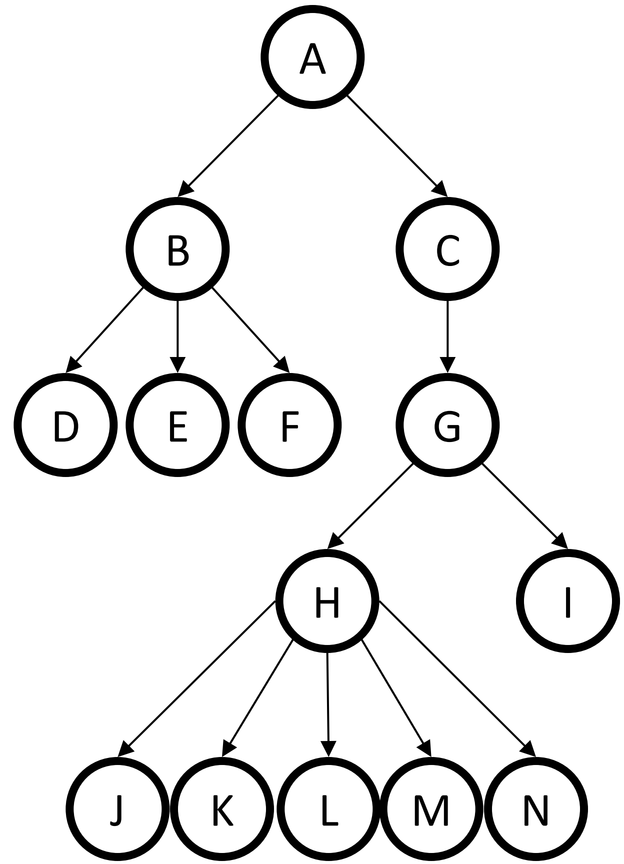
- *root:* the single node with no parent
- *leaf:* a node with no children
- *child:* a node pointed to by me
- *parent:* the node that points to me
- *sibling:* another child of my parent
- *ancestor:* my parent or my parent's ancestor



- *descendent:* my child or my child's descendent
- *subtree:* a node and its descendants

Tree Terminology

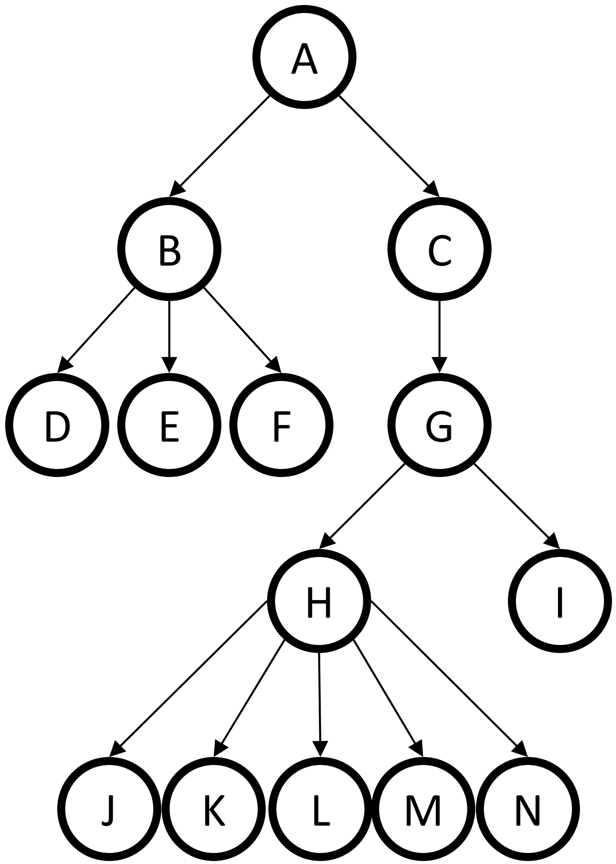
- *depth:* # of edges along path from root to node
  - *depth of H?*
    - *3*



- *height*: # of edges along longest path from node to leaf or, for whole tree, from root to leaf
  - Height of this tree?
    - 4

Tree Terminology

- *degree:* # of children of a node
  - *degree of B?*
    - *3*

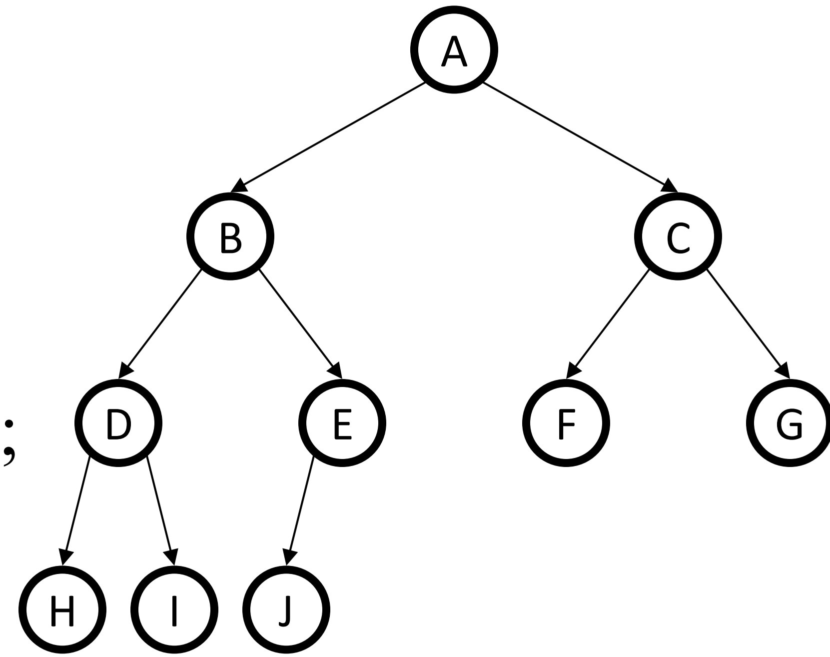


- *branching factor:* maximum degree of any node in the tree

2 for binary trees,
5 for this weird tree

One More Tree Terminology Slide

- *binary*: branching factor of 2 (each child has at most 2 children)
- *n-ary*: branching factor of n
- *complete*: "packed" binary tree; as many nodes as possible for its height



- *nearly complete*: complete plus some nodes on the left at the bottom (last level filled from left to right)

Trees and (Structural) Recursion

A tree is either:

- the empty tree
- a root node and an ordered list of subtrees

Trees are a recursively defined structure, so it makes sense to operate on them recursively.

Dictionary ADT

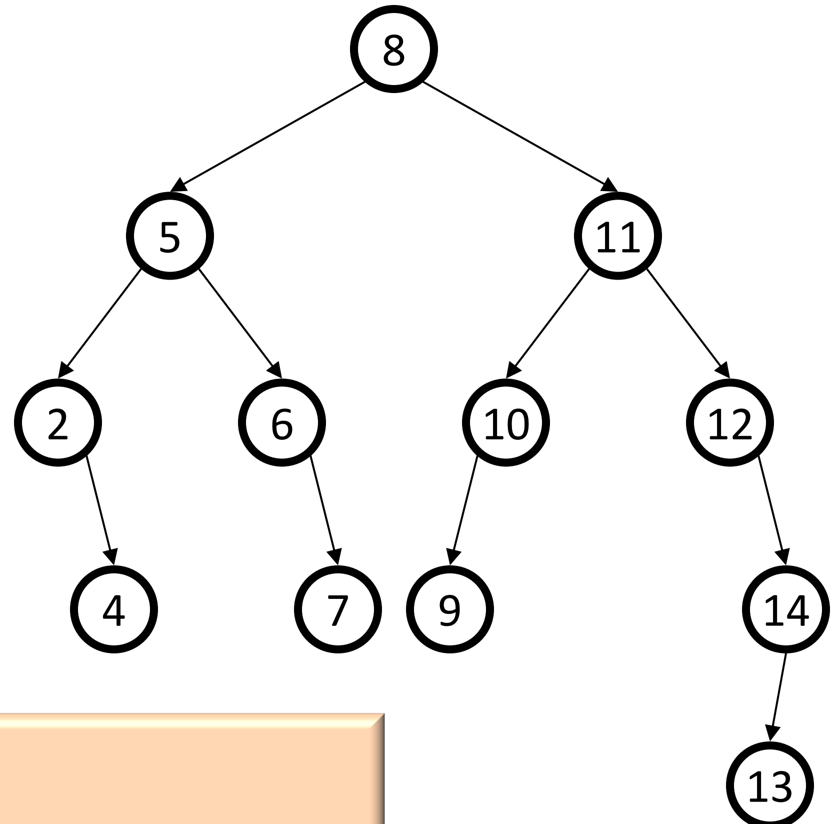
Tree Terminology

Binary Search Trees (BSTs)

Insertion and Deletion in BSTs

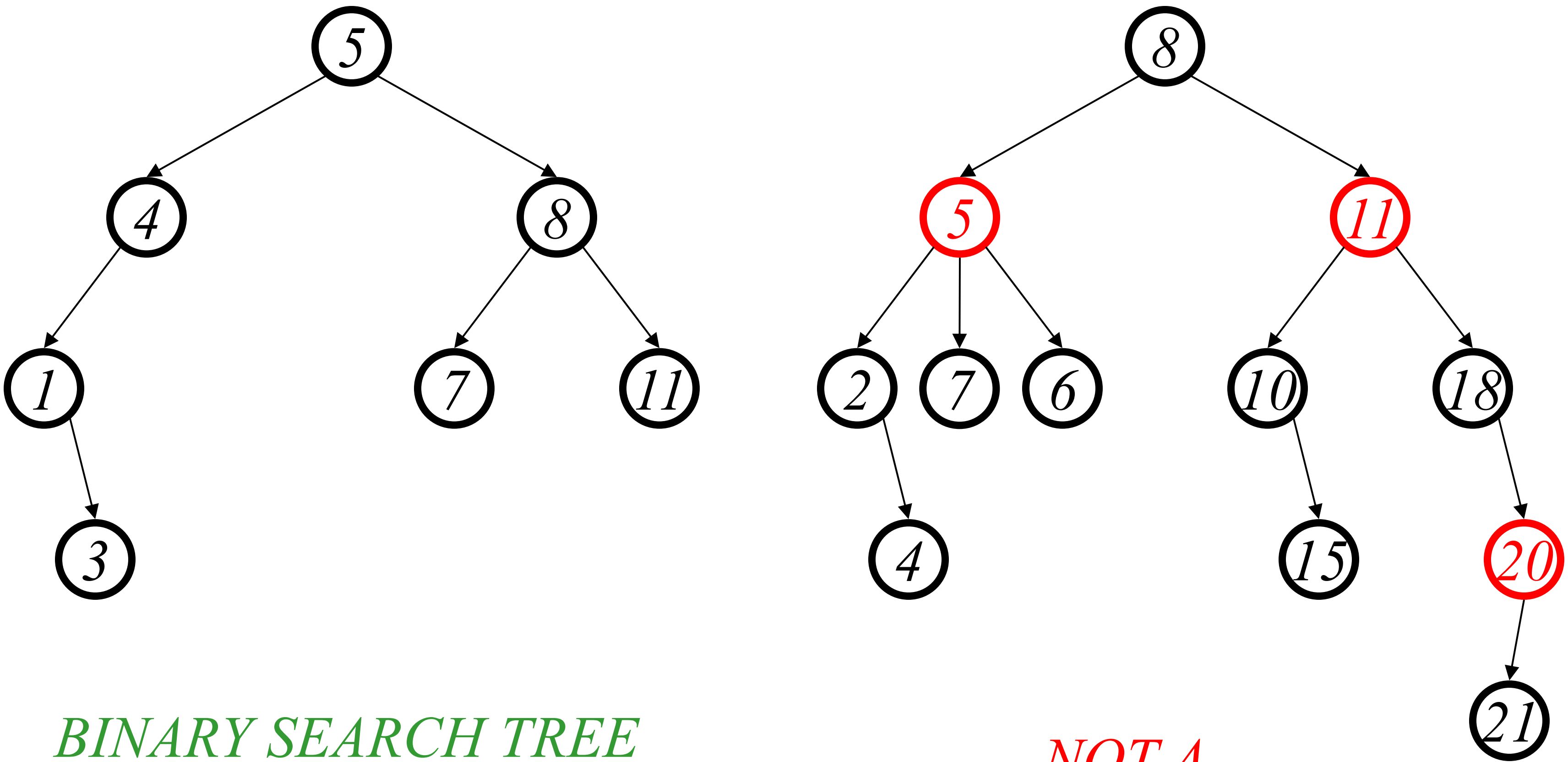
Binary Search Tree

- Binary tree property
  - each node has ≤ 2 children
  - result:
    - operations are simple



- Search tree property
  - **all** keys in left subtree smaller than root's key
  - **all** keys in right subtree larger than root's key
  - result:
    - easy to find any given key

Example and Counter-Example

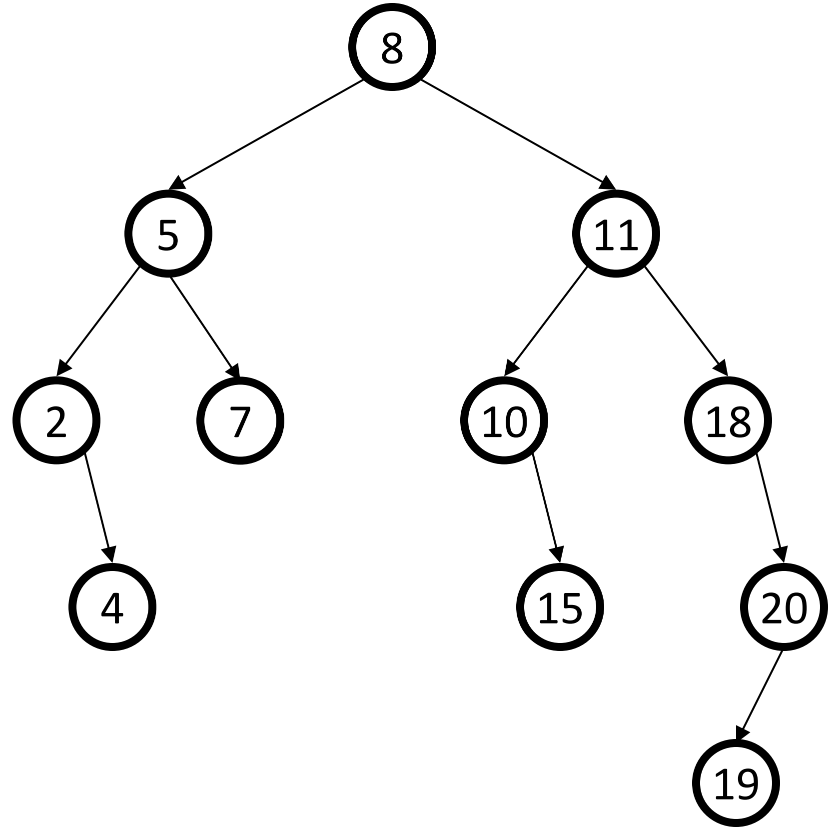


BINARY SEARCH TREE

NOT A
BINARY SEARCH TREE

Clicker Question

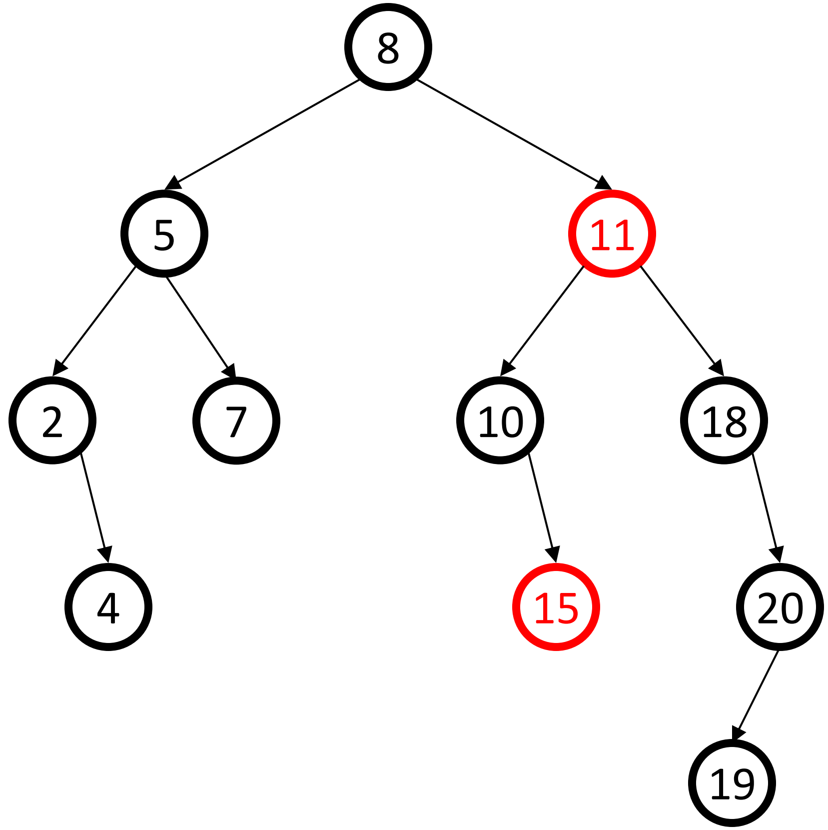
- The tree on the right is…



A. a Binary Search Tree.
B. not a Binary Search Tree.
C. hmm... I don't know.

Clicker Question

- The tree on the right is…

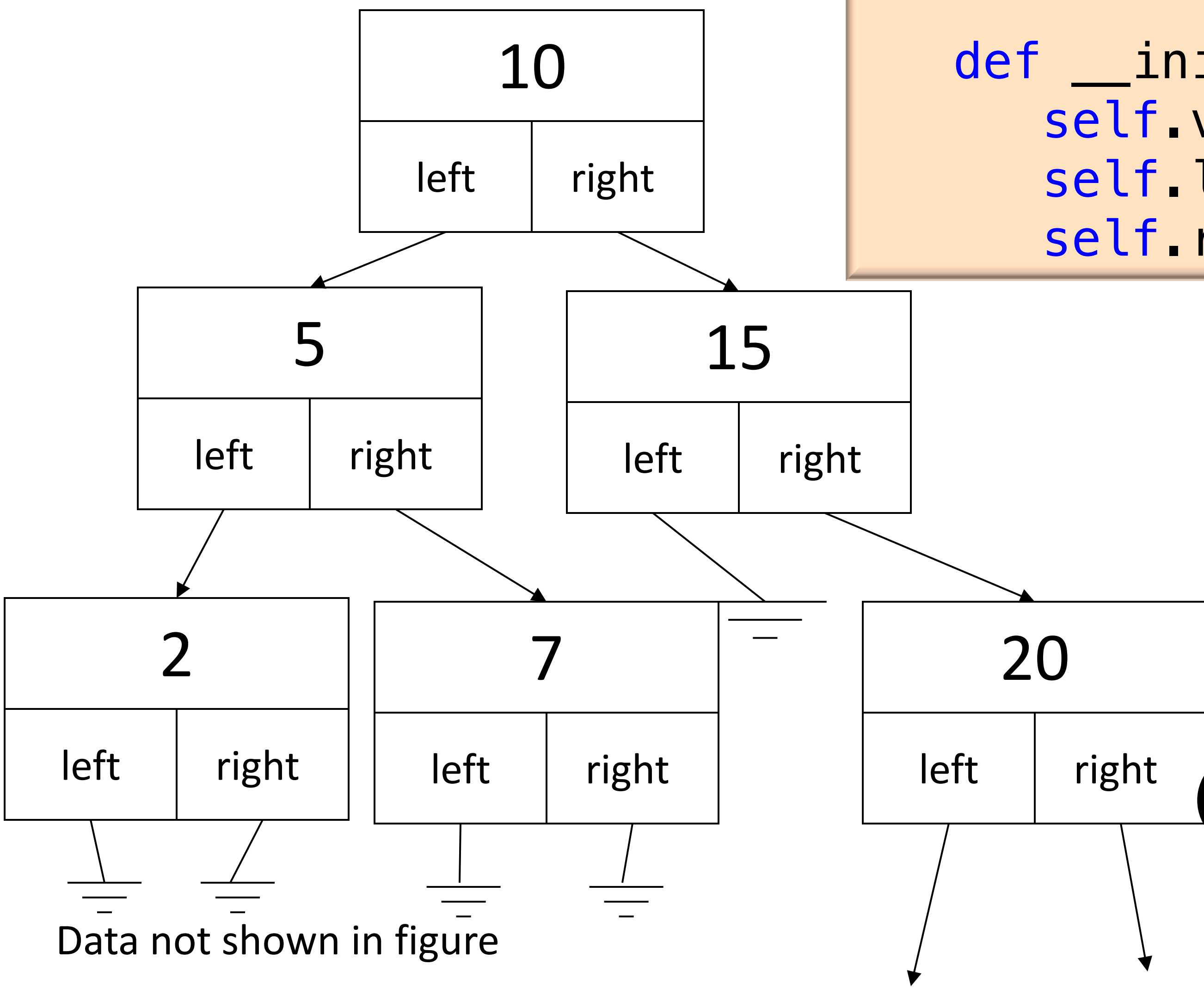


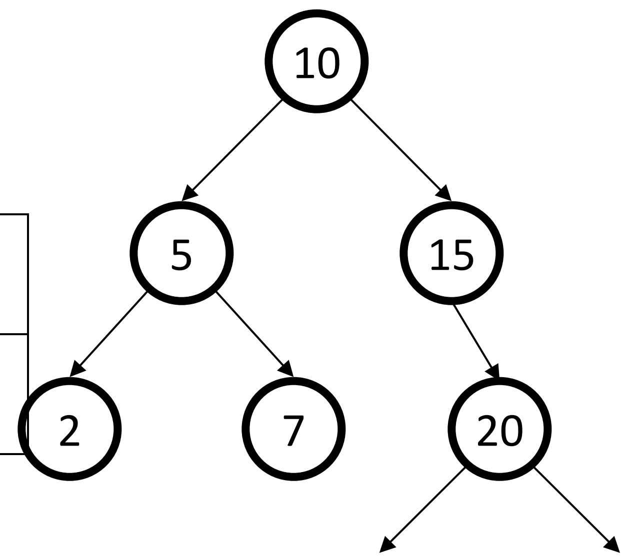
A. a Binary Search Tree.
B. not a Binary Search Tree.
C. hmm… I don’t know.

Representing Binary Search Trees

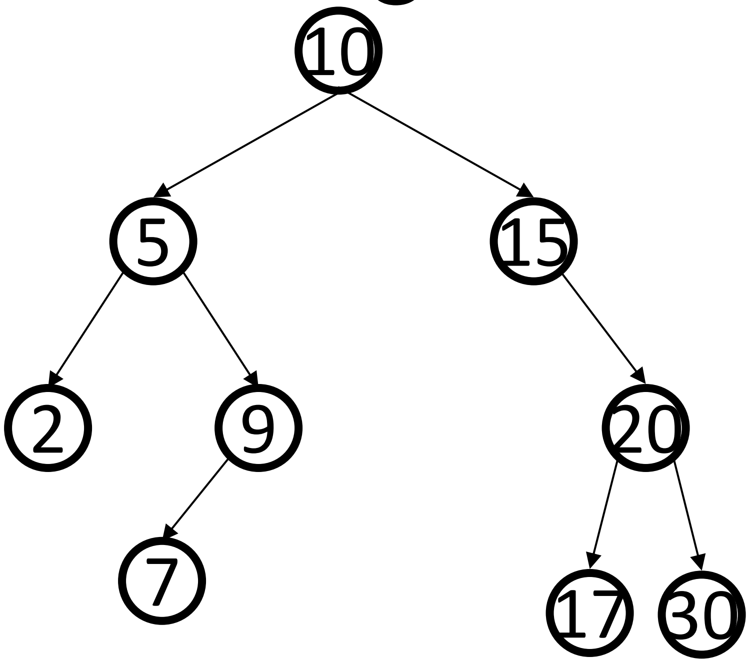
```python
class Node2(object):

    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
```



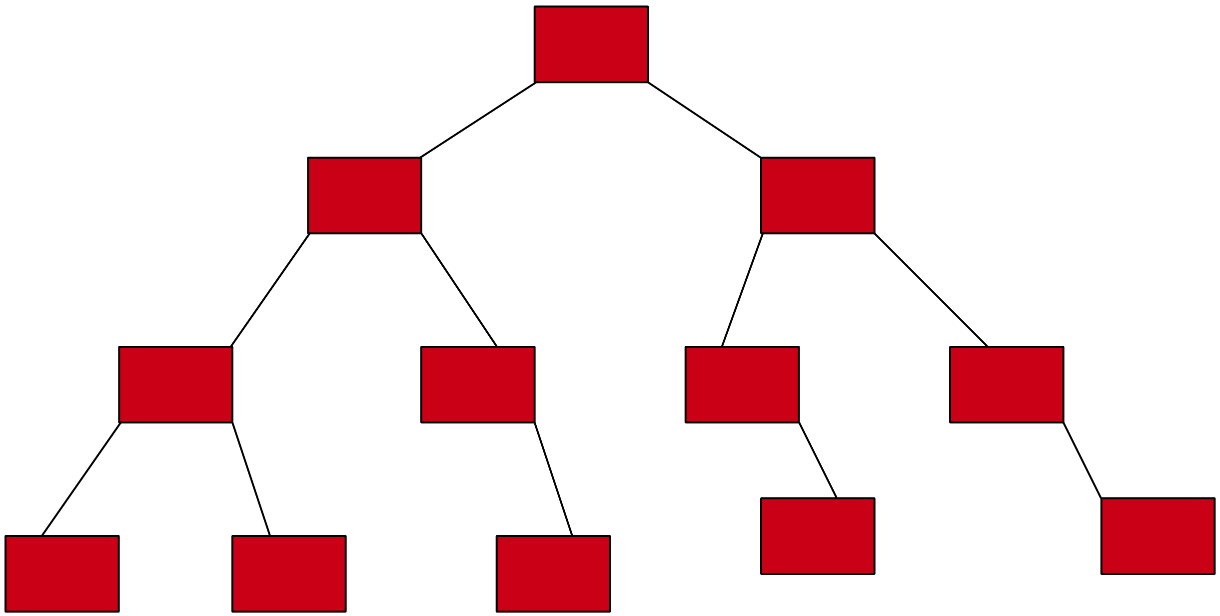


Finding a Node



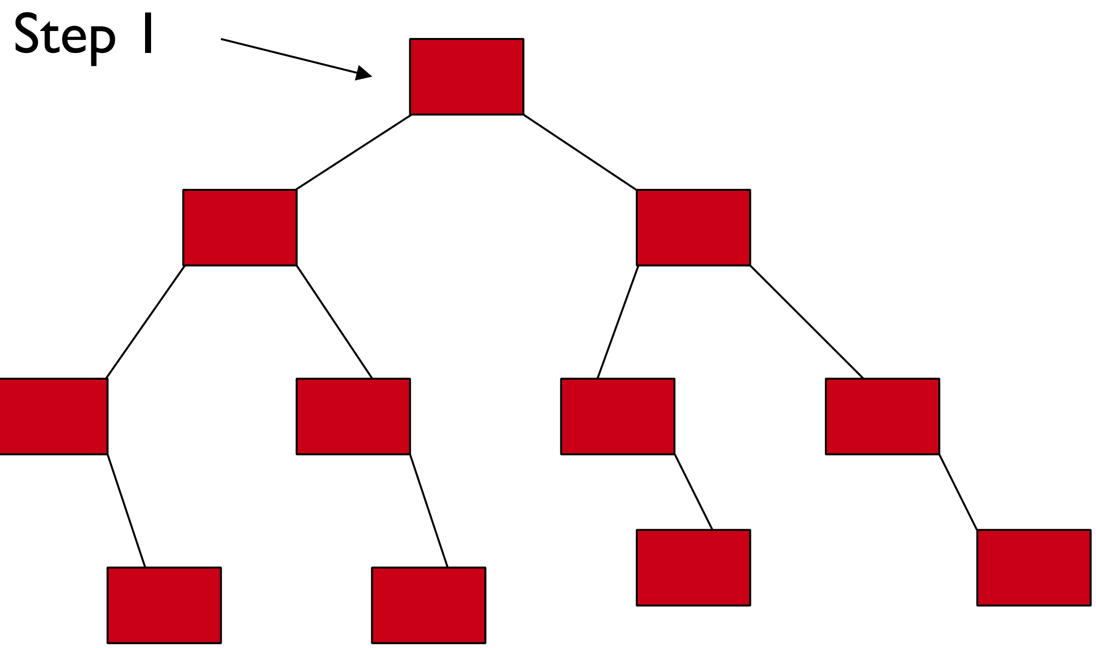

```
def search(root,key):
    # Base Cases: root is null or key is present at root
    if root is None or root.value == key:
        return root
    # Key is greater than root's key
    if root.value < key:
        return search(root.right,key)
    # Key is smaller than root's key
    return search(root.left,key)
```

What Makes a Balanced BST Efficient for Searching?



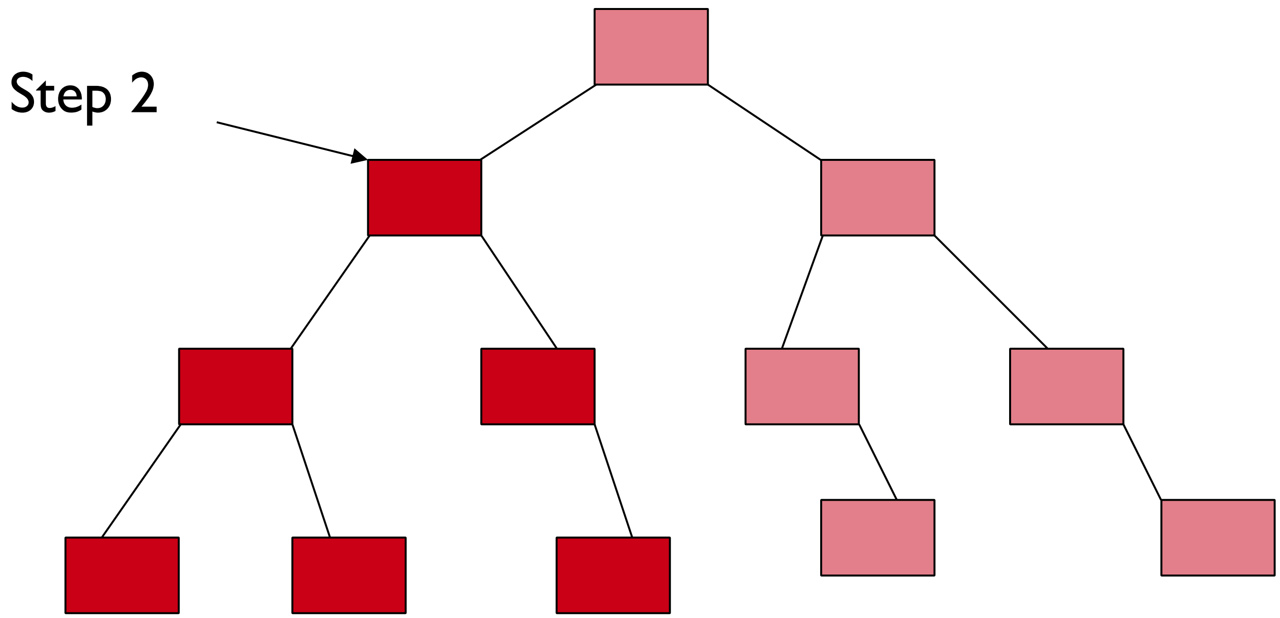
As we search the tree, each step we take reduces the remaining search space by half.

Sample Search



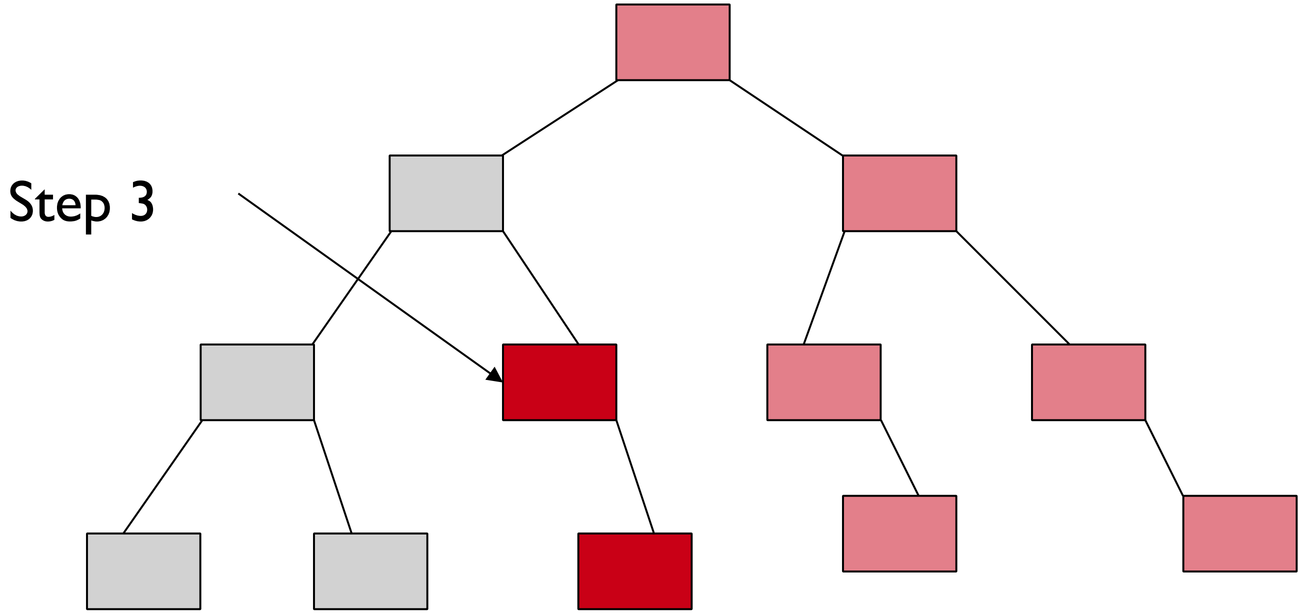
Each step we take as we search the tree reduces the remaining search space by half.

Sample Search



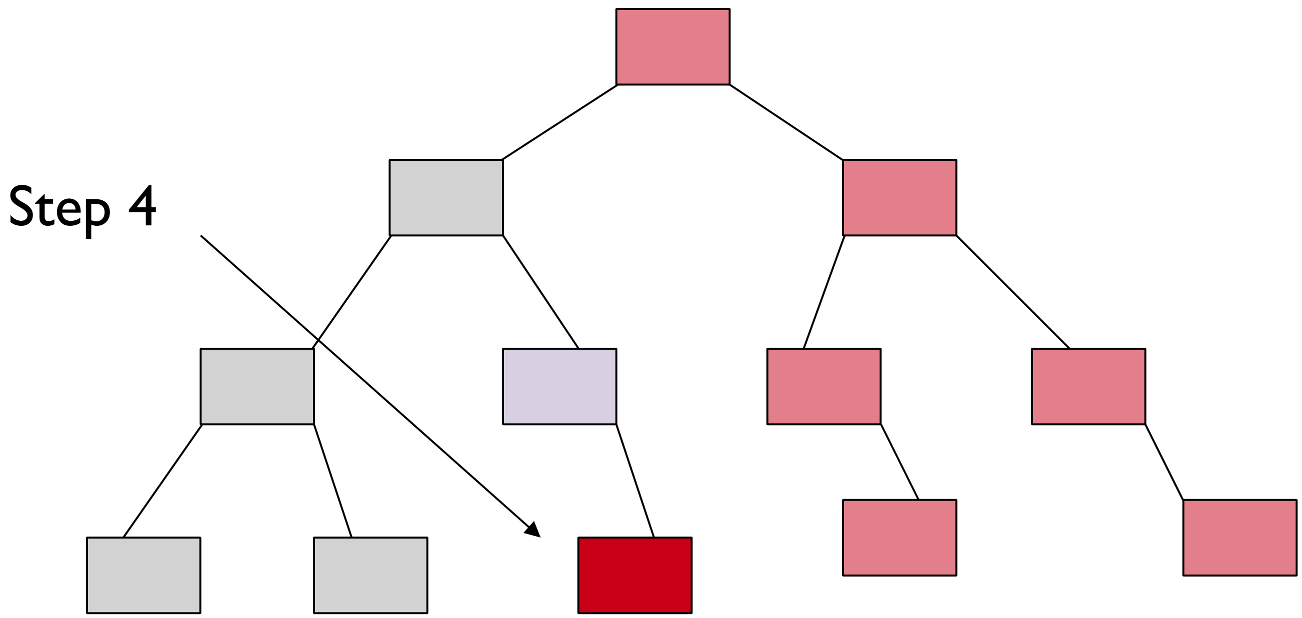
Each step we take as we search the tree reduces the remaining search space by half.

Sample Search



Each step we take as we search the tree reduces the remaining search space by half.

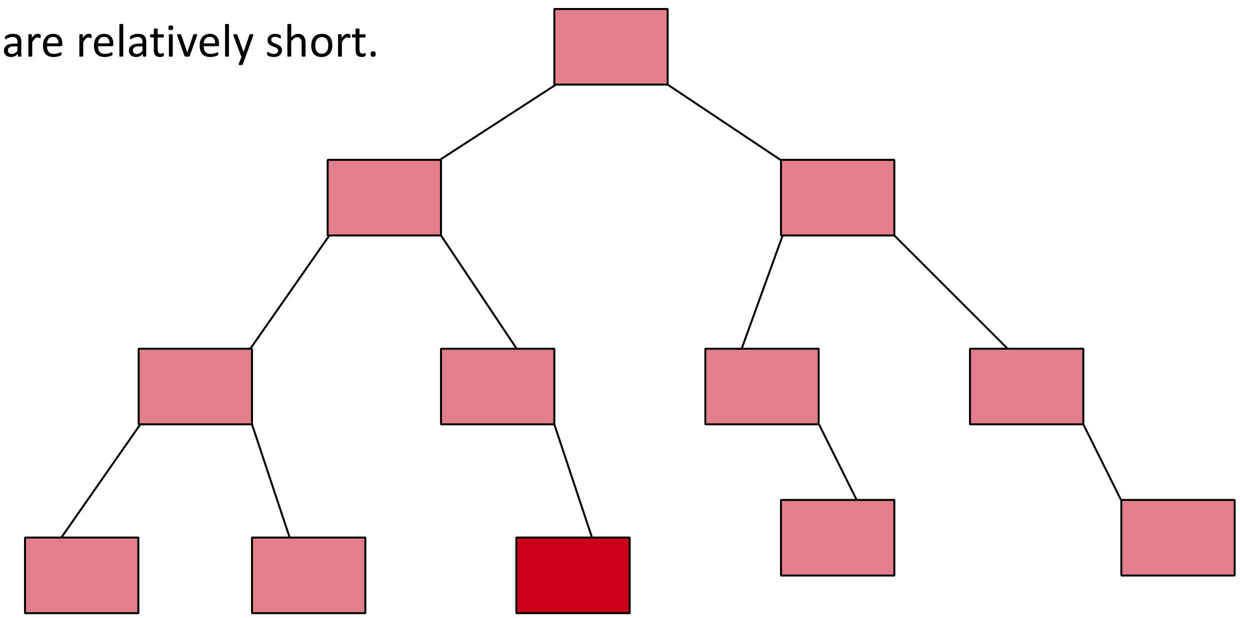
Sample Search



Each step we take as we search the tree reduces the remaining search space by half.

Sample Search

The tree has low height and all paths from the root node to other nodes are relatively short.



Each step we take as we search the tree reduces the remaining search space by half.

Clicker Question

- What is the running time of the find function in BSTs? (height represents the height of the tree)

A. $\Theta(n)$

B. $\Theta(lg\ n)$

C. $\Theta(height)$

D. $\Theta(n^2)$

E. *None of the above*

Clicker Question

- What is the running time of the find function in BSTs? (height represents the height of the tree)

A. $\Theta(n)$

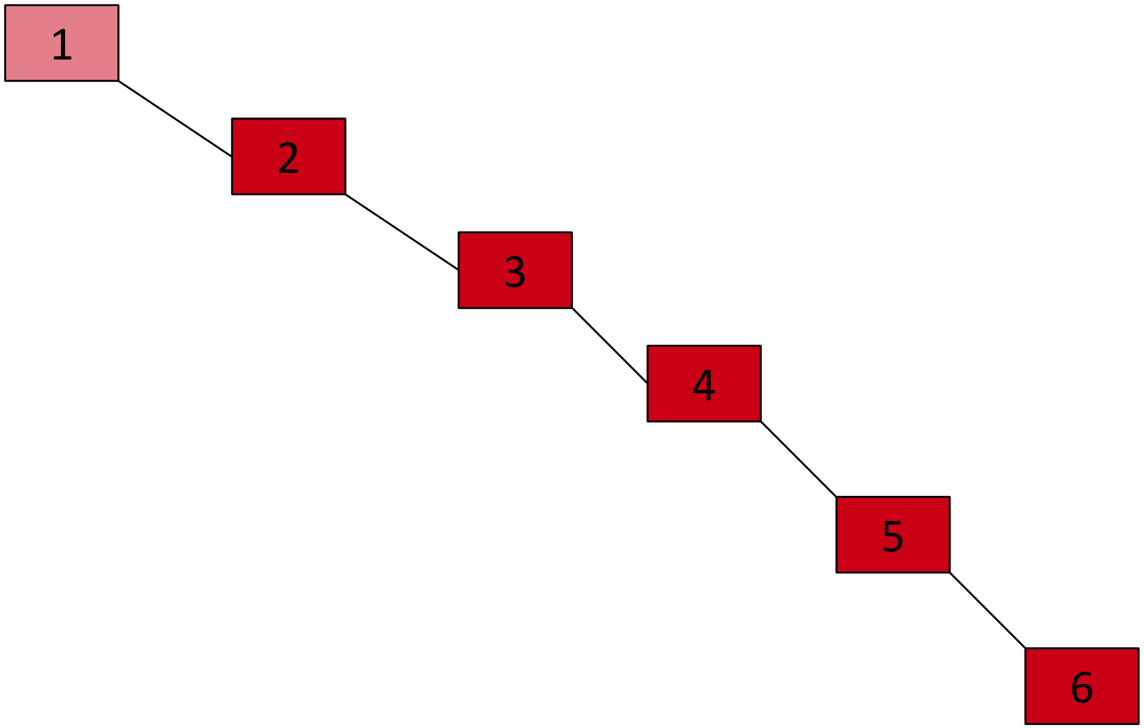
B. $\Theta(lg\ n)$

C. $\Theta(height)$

D. $\Theta(n^2)$

E. None of the above

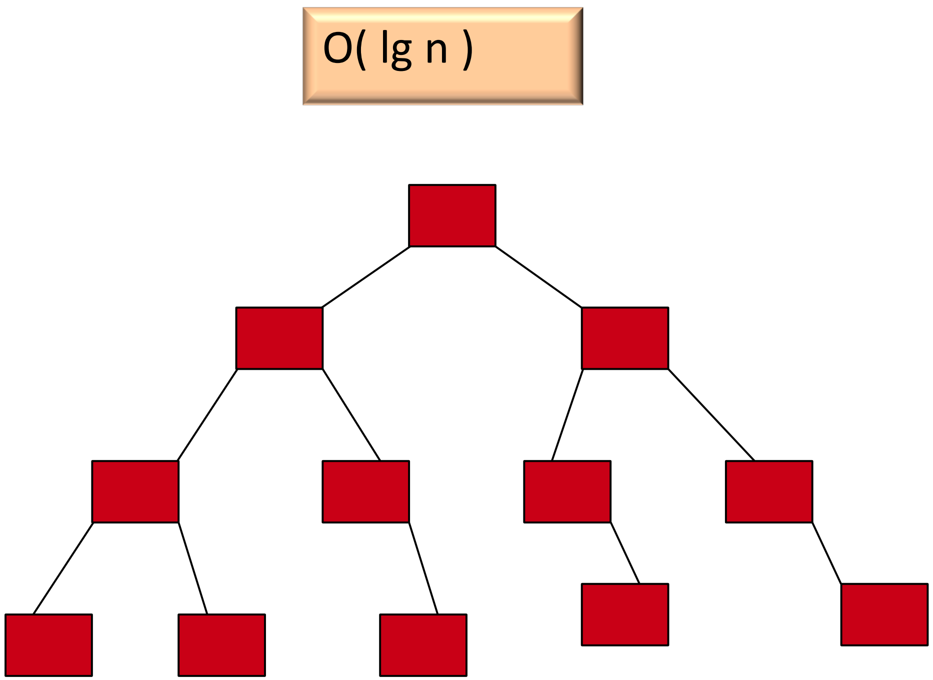
Unbalanced Trees

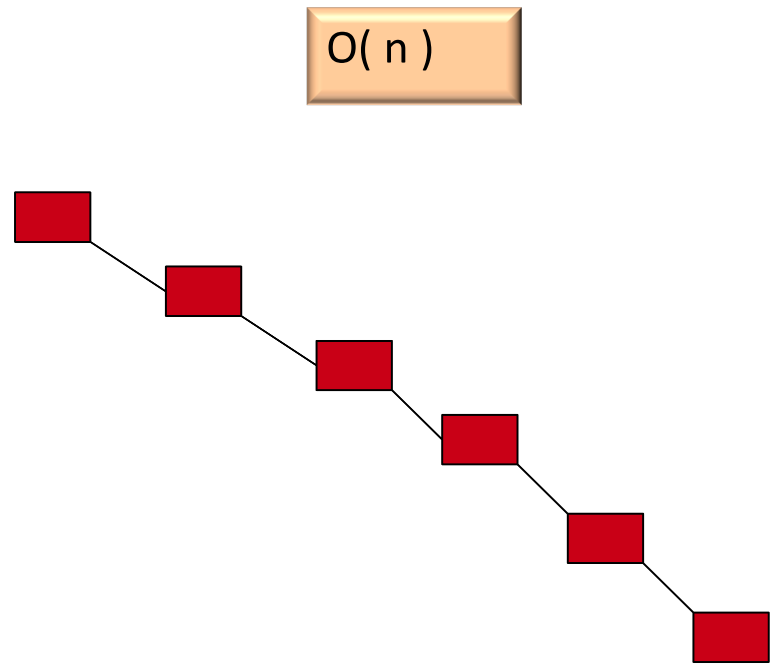


In contrast, this unbalanced tree has long paths from the root to other nodes. It essentially has degenerated to a linked list, which is very slow to search through. Now, with each step we take, we have only reduced the search space by one node.

Time of Search

- Time of search is proportional to the height of the tree





Time of Search

- What does this tell you about strengths and weaknesses of BSTs?

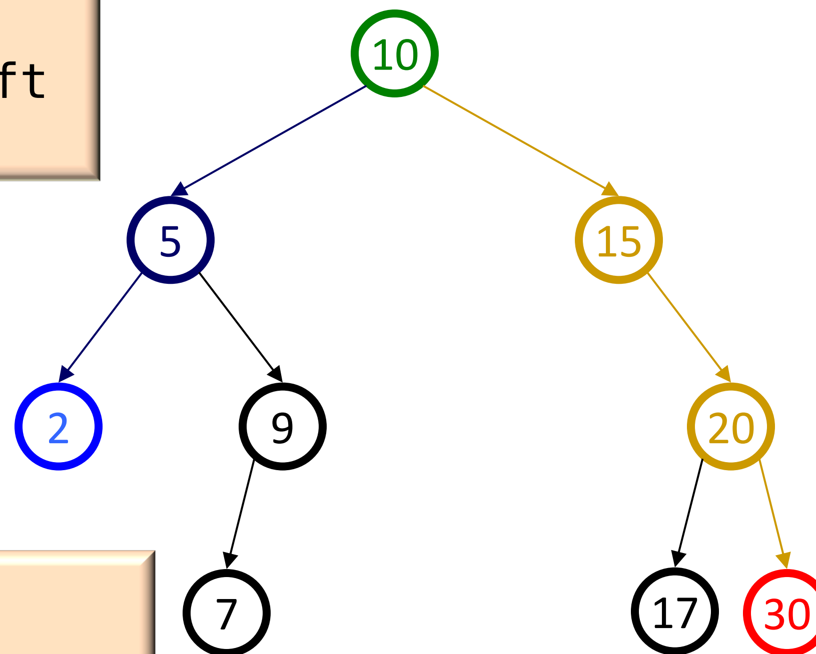
- Using BSTs is only efficient if they are fairly balanced. Whether BSTs are balanced is highly dependent on the order of the values being added.

- There are extensions and improvements to Binary Search Trees such as **AVL trees, B+ trees** or **red-black trees**. You are encouraged to read about them, but they are outside of the scope of this course.

Bonus: FindMin/FindMax

- Find minimum

```
def find_min(root):
    current_node = root
    while current_node.left:
        current_node =current_node.left
    return current_node
```



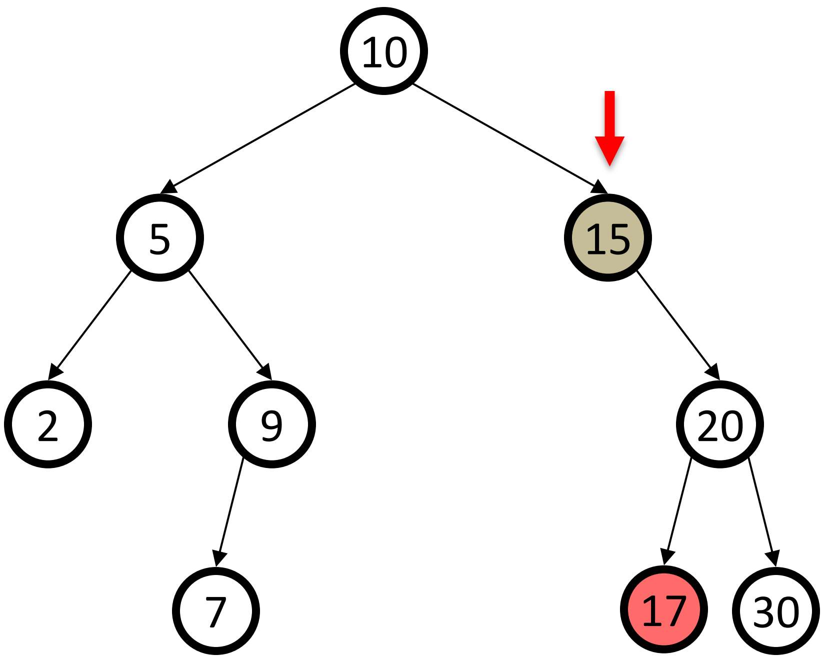
- Find maximum

```
def find_max(root):
    current_node = root
    while current_node.right:
        current_node = current_node.right
    return current_node
```

Double Bonus: Successor

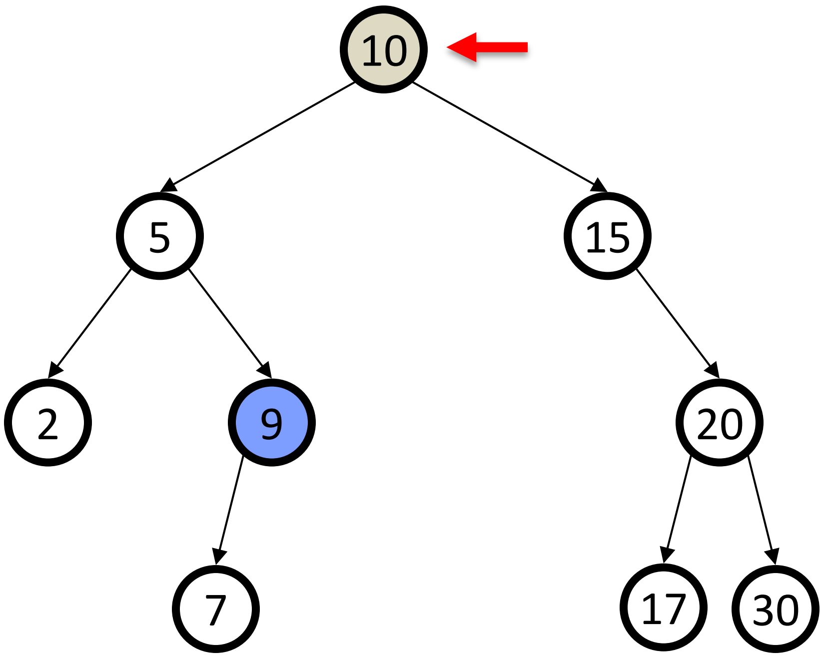
Find the next larger node in a node's subtree.

```
def succ (x):
    return find_min(x.right)
```



More Double Bonus: Predecessor

Find the next smaller node in a node's subtree.




```
def Pred (x):
    return find_max(x.left)
```

Dictionary ADT

Tree Terminology

Binary Search Trees (BSTs)

Insertion and Deletion in BSTs

Insertion Algorithm

1. perform search for value X
2. if X is in tree

    return

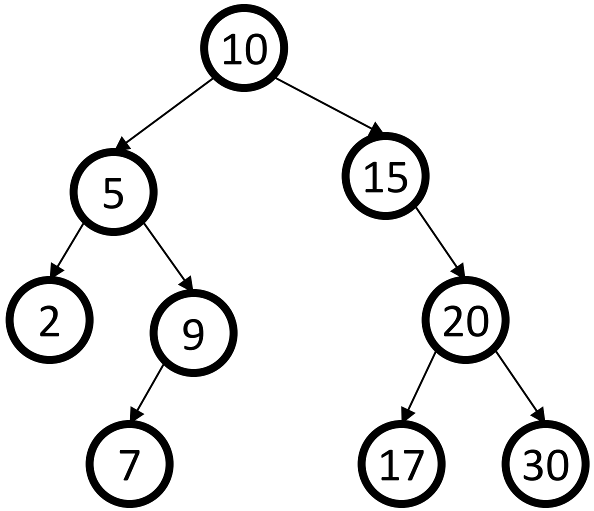
    // else search will end at a node. Call it Y

3. if X < Y

    insert X as new left subtree for Y

4. if X > Y

    insert X as new right subtree for Y



Insertion Algorithm

1. perform search for value X
2. if X is in tree

    return

    // else search will end at a node. Call it Y

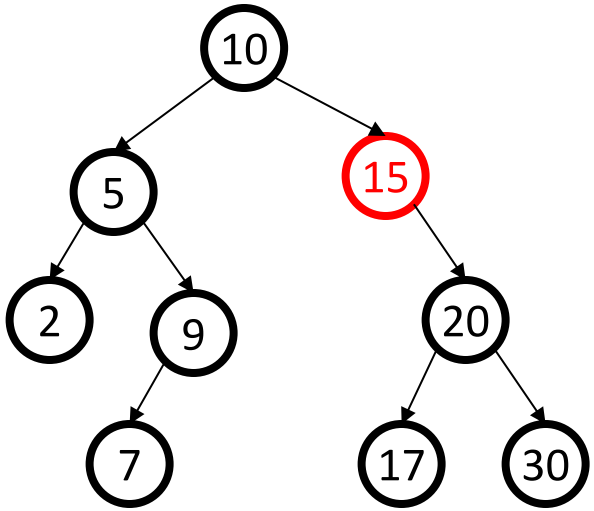
3. if X < Y

    insert X as new left subtree for Y

4. if X > Y

    insert X as new right subtree for Y

insert(15)



Insertion Algorithm

1. perform search for value X
2. if X is in tree

   return

   // else search will end at a node. Call it Y

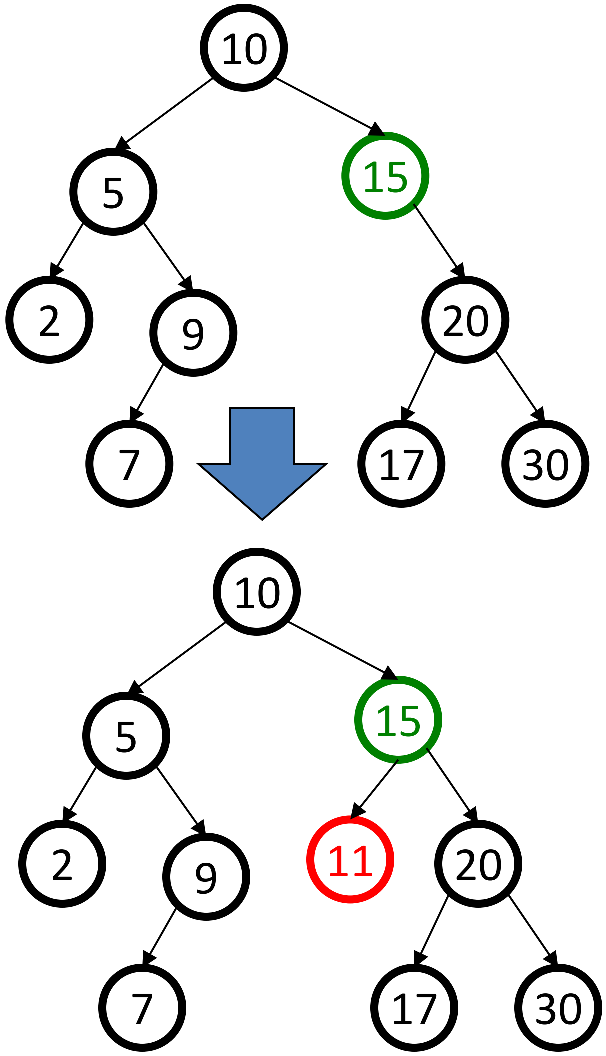
3. if X < Y

   insert X as new left subtree for Y

4. if X > Y

   insert X as new right subtree for Y

insert(11)



Insertion Algorithm

1. perform search for value X
2. if X is in tree

   return

   // else search will end at a node. Call it Y

3. if X < Y

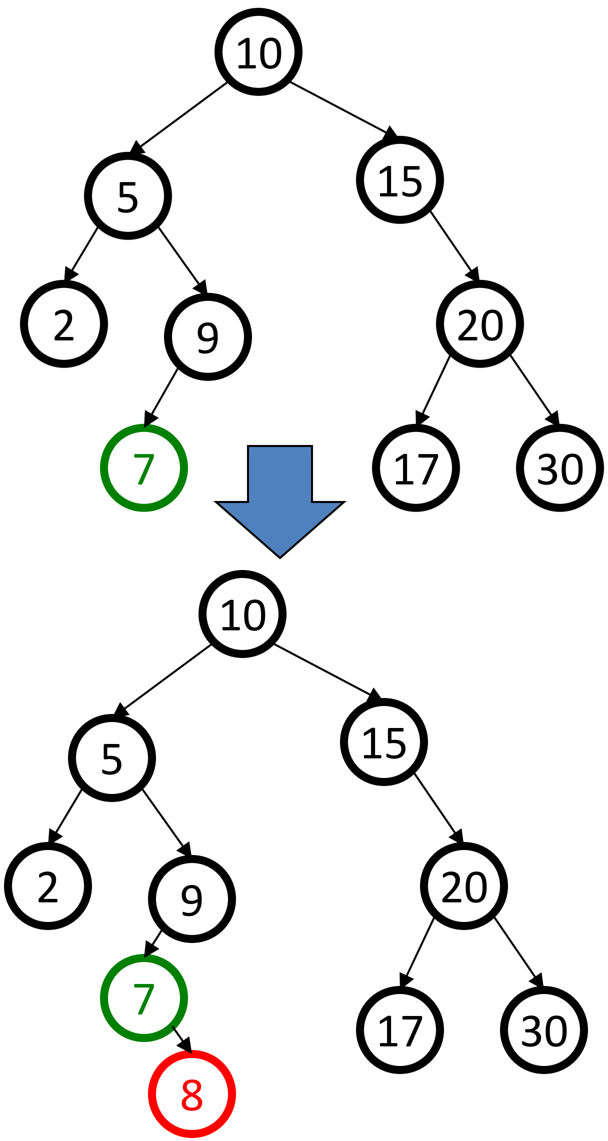
   insert X as new left subtree for Y

4. if X > Y

   insert X as new right subtree for Y

Runtime?

insert(8)



Insertion Algorithm

```python
def insert(root,node):
    if root is None:
        root = node
    else:
        if root.value == node.value: # already exists
            return
        # To be inserted on the right side of current root
        elif root.value < node.value:
            if root.right is None:
                root.right = node
            else:
                insert(root.right, node)
        # To be inserted on the left side of current root
        else:
            if root.left is None:
                root.left = node
            else:
                insert(root.left, node)
```

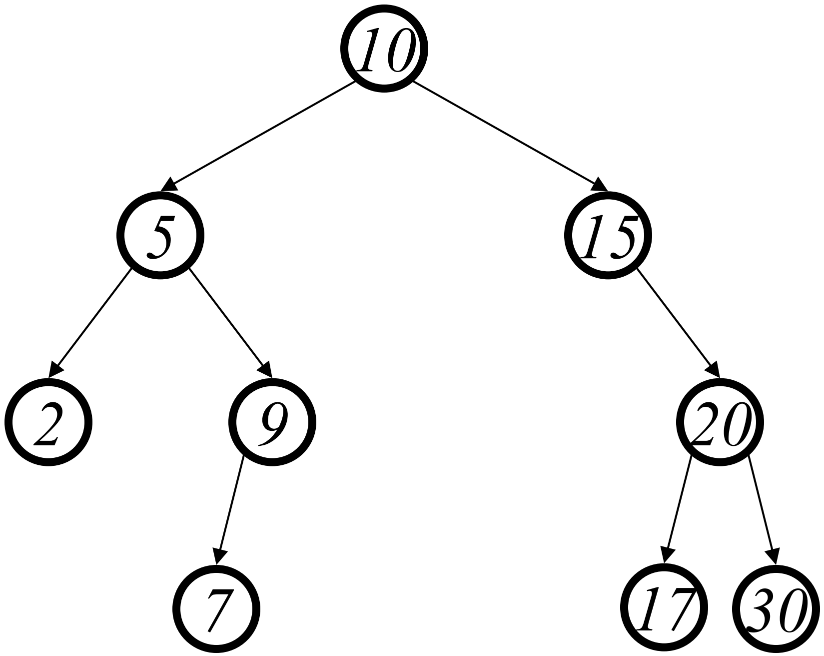
BuildTree for BSTs

- Suppose the data 1, 2, 3, 4, 5, 6, 7, 8, 9 is inserted into an initially empty BST:
  - in order (1, 2, 3, 4, 5, 6, 7, 8, 9)
  - in reverse order (9, 8, 7, 6, 5, 4, 3, 2, 1)
  - median first, then left median, right median, etc. so: 5, 3, 8, 2, 4, 7, 9, 1, 6

Analysis of BuildTree

- Worst case: $O(n^2)$ as we've seen
- Average case assuming all orderings equally likely turns out to be $O(n \lg n)$.

Deletion



Why might deletion be harder than insertion?

Deletion Algorithm

1. perform search for value X
2. if X is not in tree

   return

   //else X is in tree

3. if X has no children,

   delete X

4. else if X has only one child

   promote the unique child to X's place.

   delete X

5. else // X must have two children

   identify X's successor. Call it Y

   replace X with Y

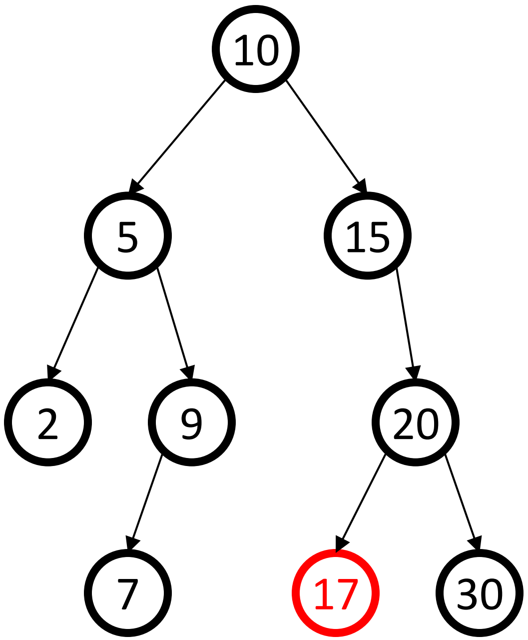
   delete Y // Y $\leq$ 1 children

Deletion Algorithm

1. perform search for value X
2. if X is not in tree
   return

   //else X is in tree
3. if X has no children,
   delete X
4. else if X has only one child
   promote the unique child to X's place.
   delete X
5. else // X must have two children
   identify X's successor. Call it Y
   replace X with Y
   delete Y // $Y \leq 1$ children

Delete(17)



Deletion Algorithm

Delete(15)

1. perform search for value X
2. if X is not in tree

   return

   //else X is in tree

3. if X has no children,

   delete X

4. else if X has only one child

   promote the unique child to X’s place.

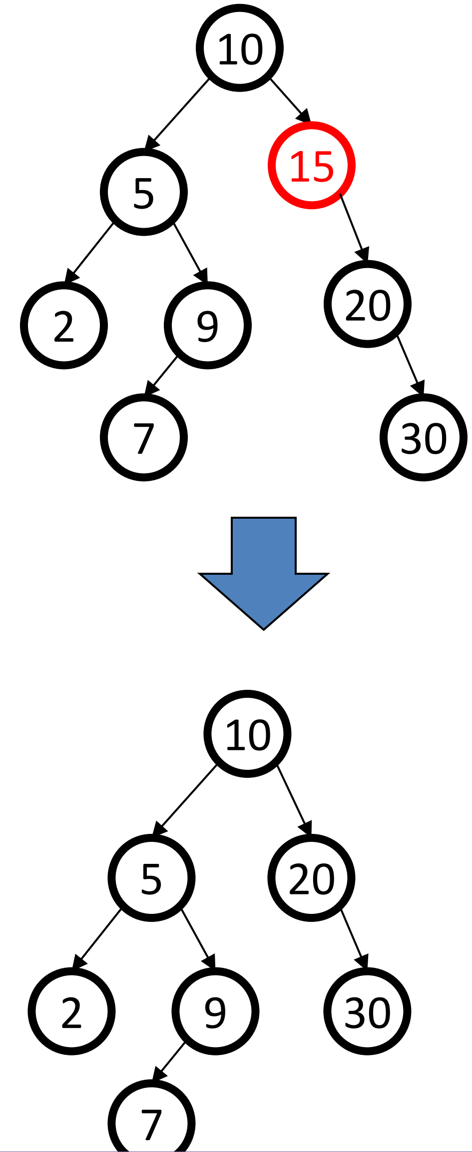
   delete X

5. else // X must have two children

   identify X’s successor. Call it Y

   replace X with Y

   delete Y // Y $\leq$ 1 children



Deletion Algorithm

1. perform search for value X
2. if X is not in tree

   return

   //else X is in tree

3. if X has no children,

   delete X

4. else if X has only one child

   promote the unique child to X's place.

   delete X

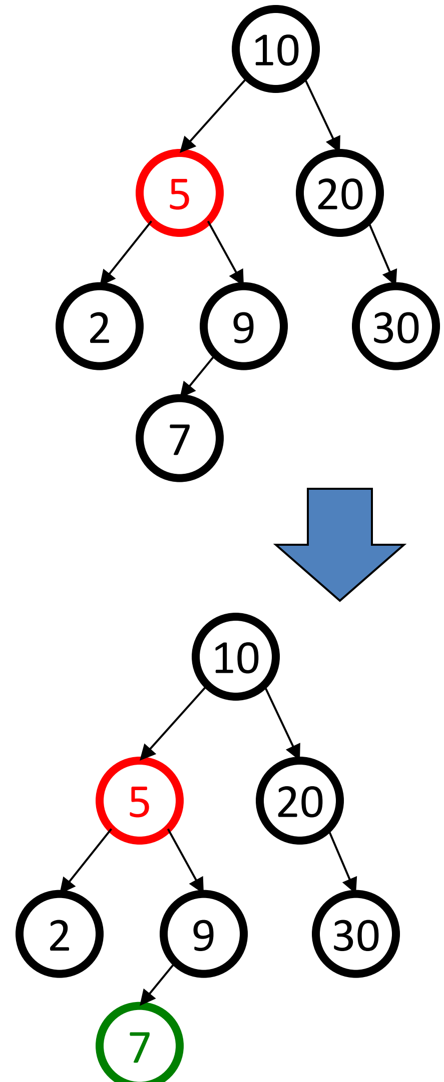
5. else // X must have two children

   identify X's successor. Call it Y

   replace X with Y

   delete Y // Y ≤ 1 children

Delete(5)



Deletion Algorithm

1. perform search for value X
2. if X is not in tree

   return

   //else X is in tree

3. if X has no children,

   delete X

4. else if X has only one child

   promote the unique child to X's place.

   delete X

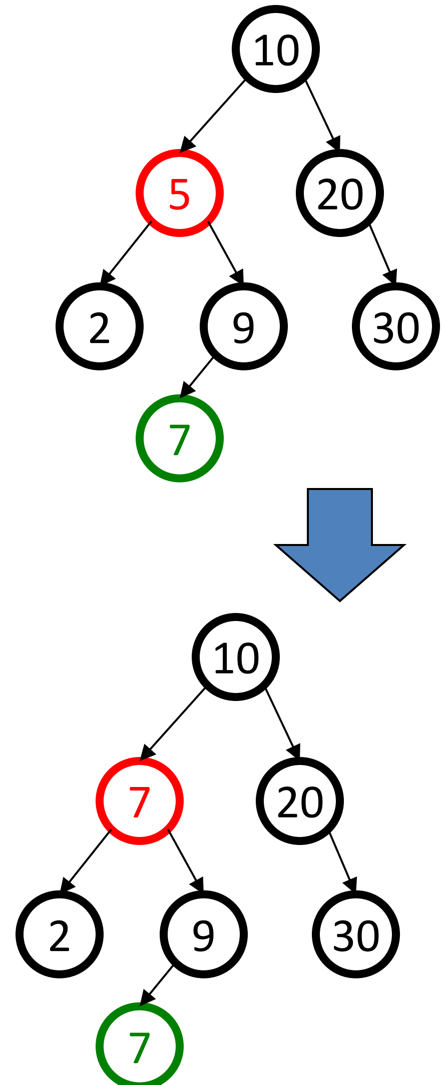
5. else // X must have two children

   identify X's successor. Call it Y

   replace X with Y

   delete Y // Y $\leq$ 1 children

Delete(5)



Deletion Algorithm

1. perform search for value X
2. if X is not in tree

   return

   //else X is in tree
3. if X has no children,

   delete X
4. else if X has only one child

   promote the unique child to X's place.

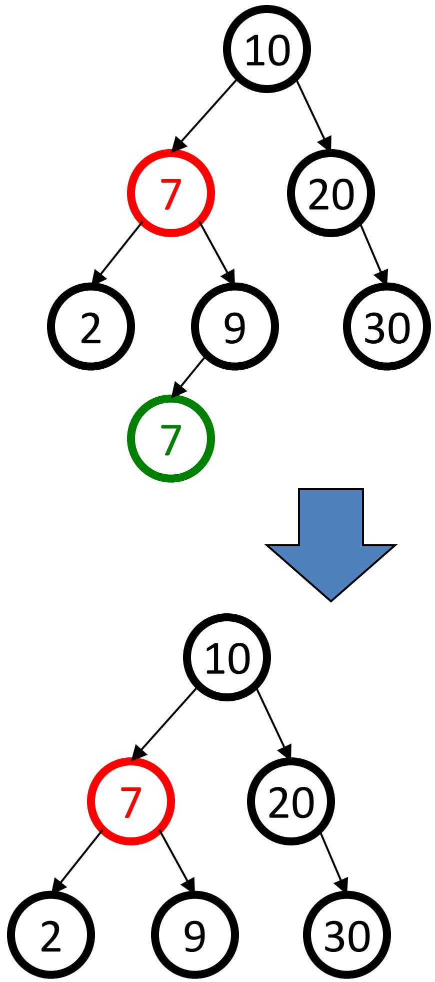
   delete X
5. else // X must have two children

   identify X's successor. Call it Y

   replace X with Y

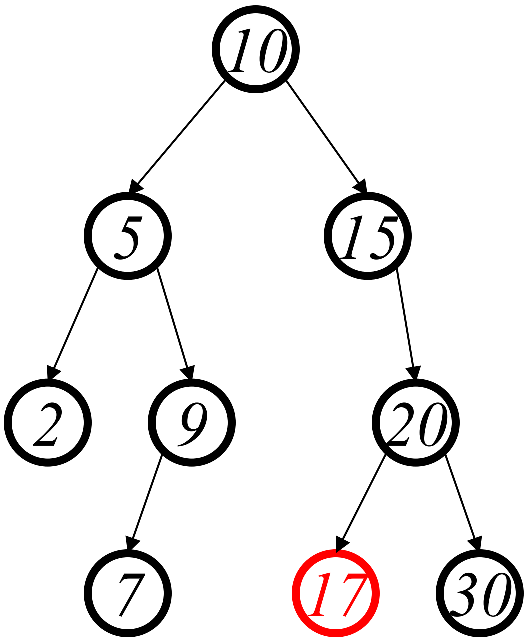
   delete Y // Y $\leq$ 1 children

Delete(5)



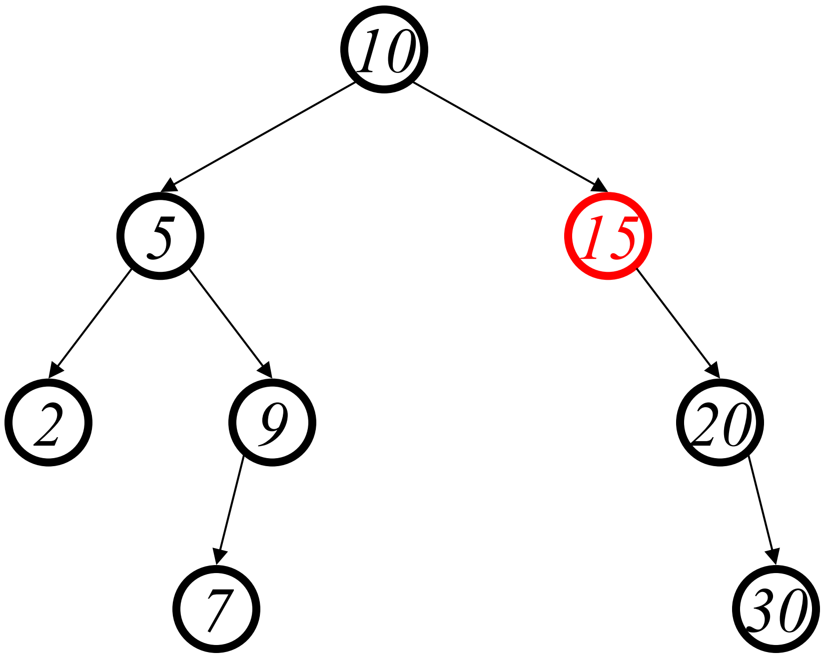
Deletion - Leaf Case

Delete(17)



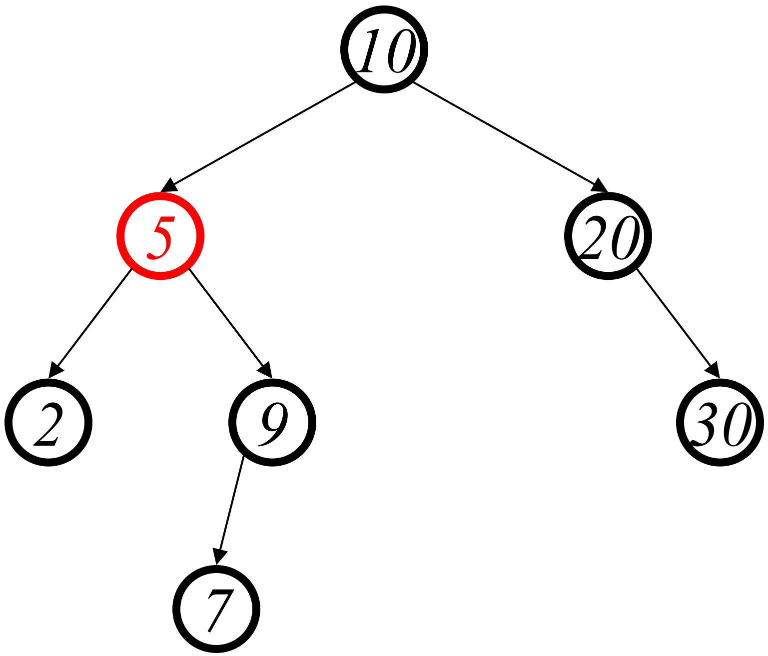
Deletion - One Child Case

Delete(15)



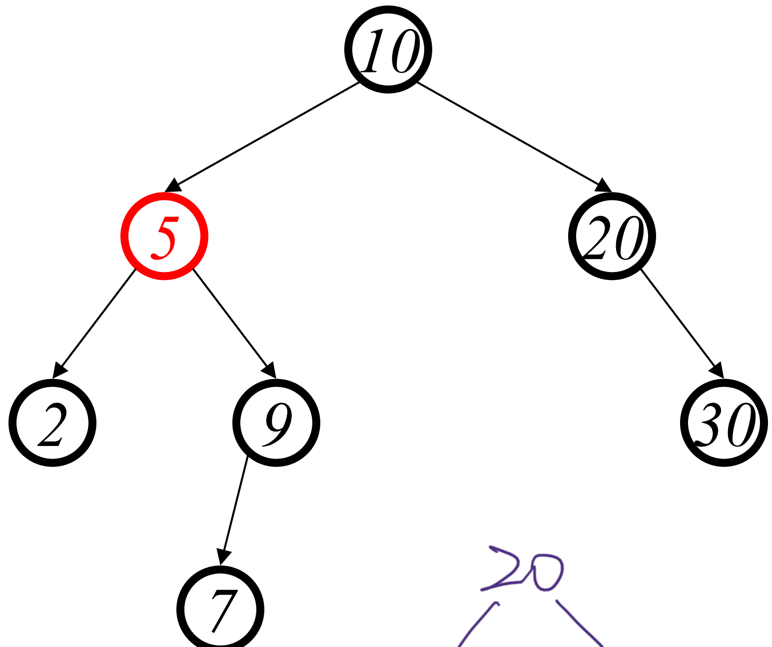
Deletion - Two Child Case

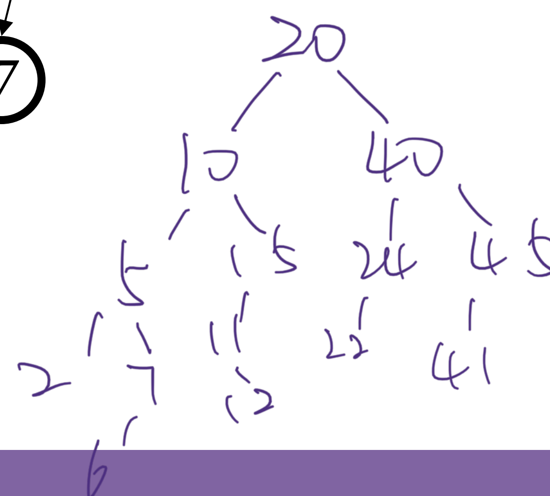
Delete(5)



Deletion - Two Child Case

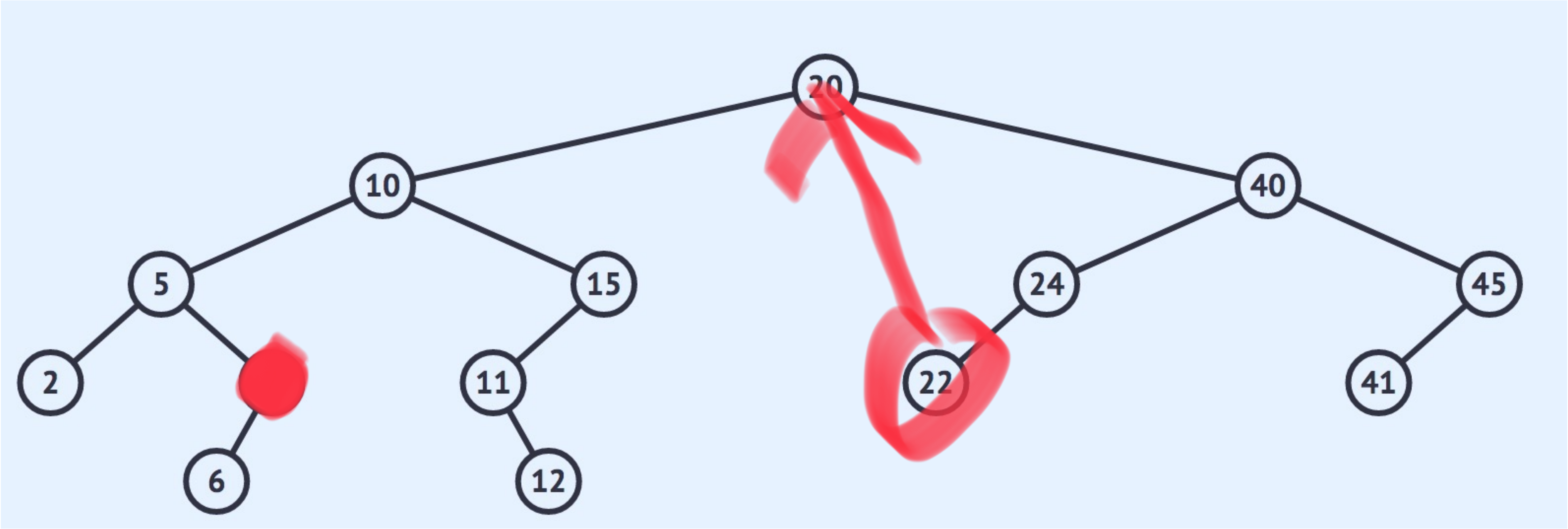
Delete(5)





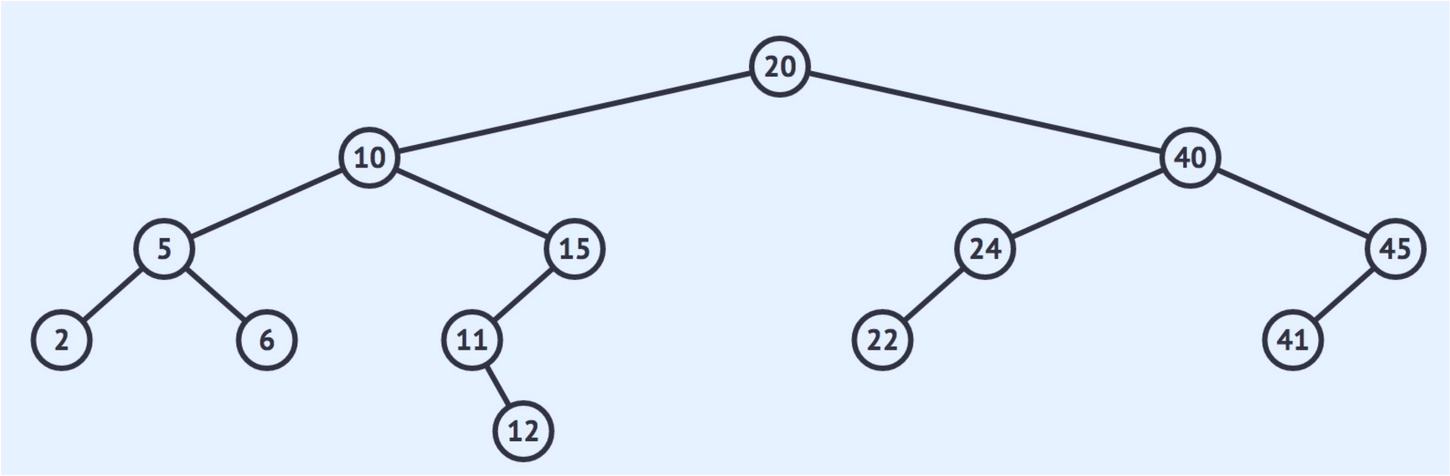
In-Class Exercise

- Draw the binary search tree, which results from adding the following keys in the given order:
  - 20, 10, 40, 5, 7, 2, 15, 11, 12, 6, 24, 22, 45, 41



In-Class Exercise

- From your tree remove key with value 7



- From your tree remove key with value 20 (using successor)