Algorithms and Data Structures

Lecture 6

Binary Search Trees I

Fabian Kuhn

Algorithms and Complexity

Abstract Data Types : Dictionary

Dictionary: (also: maps, associative arrays)

- Manages a set of elements, where each element is represented by a unique key

Operations

- *create* : creates a new empty dictionary
- *D.insert(key, value)* : inserts a new *(key,value)*-pair
  - If there already exists an entry for *key*, the old entry is replaced
- *D.find(key)* : returns entry for the given *key*
  - If such an entry exists (otherwise a default value is returned)
- *D.delete(key)* : deletes the entry for the given *key*

Can be implemented with hash tables in (amortized) constant time!

Abstract Data Types : Dictionary

Dictionary:

Additional possible operations:

- *D.minimum()* : returns smallest *key* in the data struture
- *D.maximum()* : returns largest *key* in the data structure
- *D.successor(key)* : returns next larger key
- *D.predecessor(key)* : returns next smaller key
- *D.getRange(k1, k2)* : returns all entries with keys in the interval *[k1,k2]*

These operations cannot be implemented efficiently with a hash table.

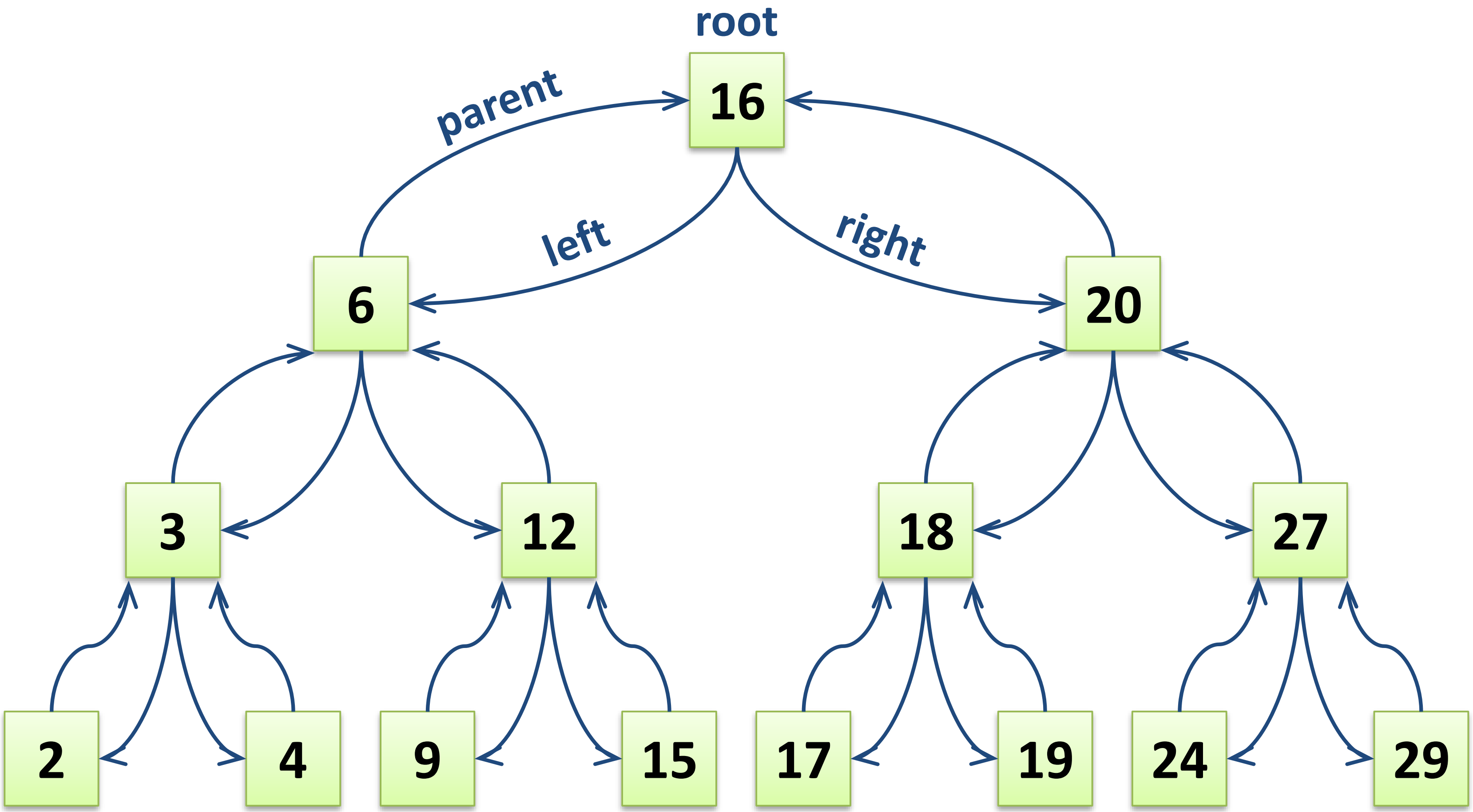
Binary Search Trees : Idea

Search for key 19:

2	3	4	6	9	12	15	16	17	18	19	20	24	27	29

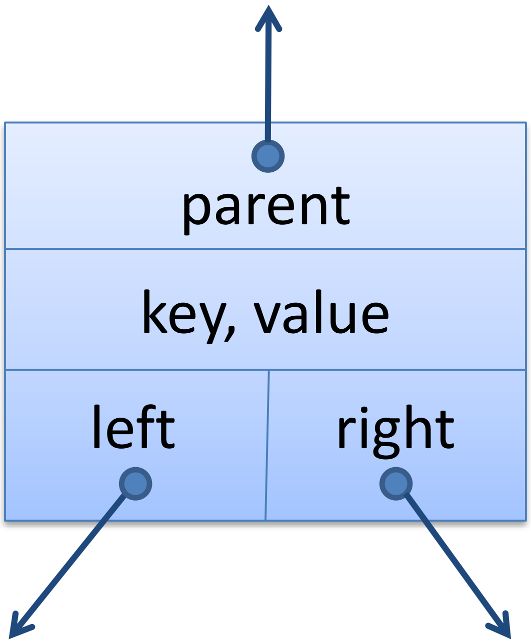
Binary Search Trees : Idea

- Use the search tree of the binary search as data structure



Binary Search Tree : Elements

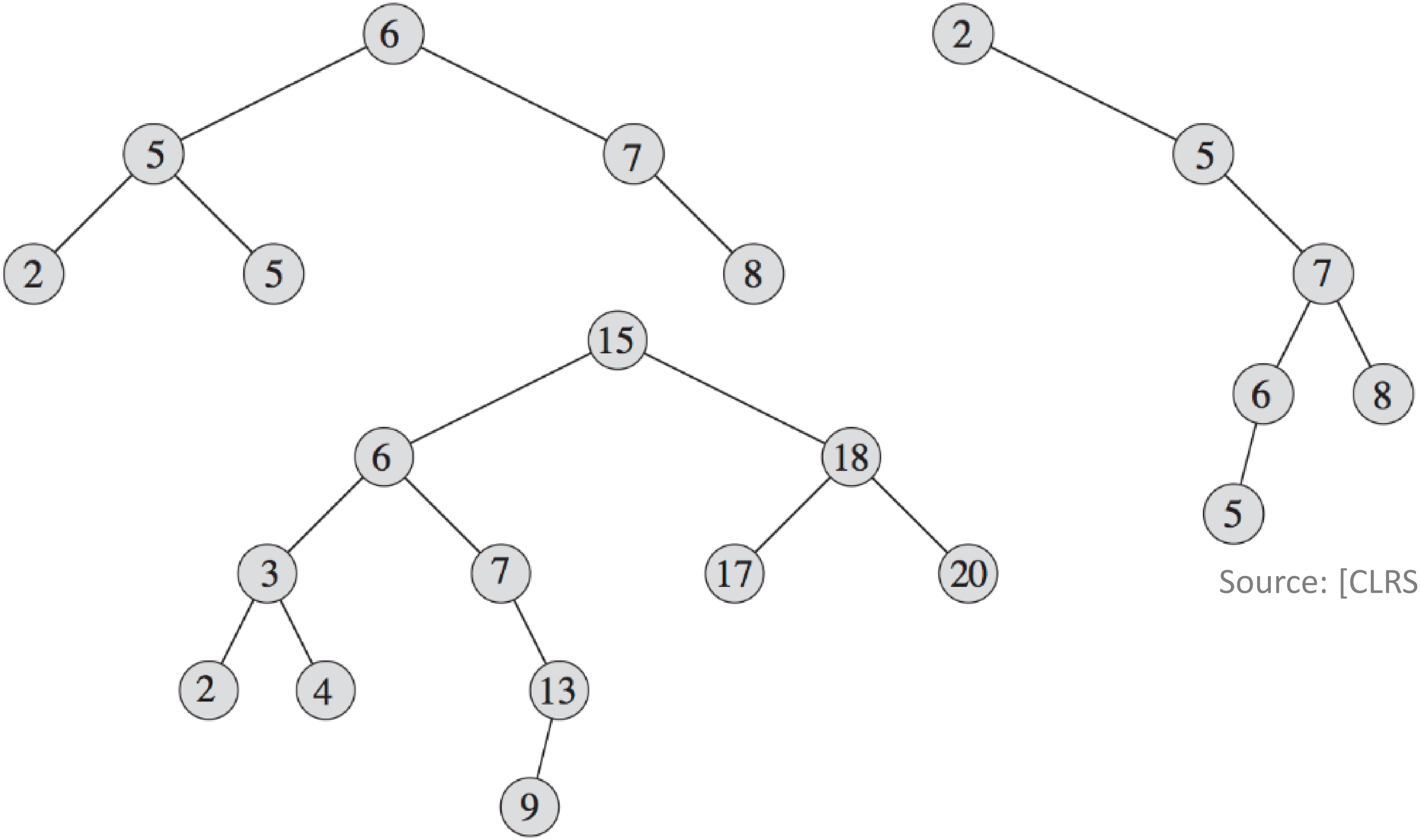
TreeElement:



Implementation: in the same way as for list elements

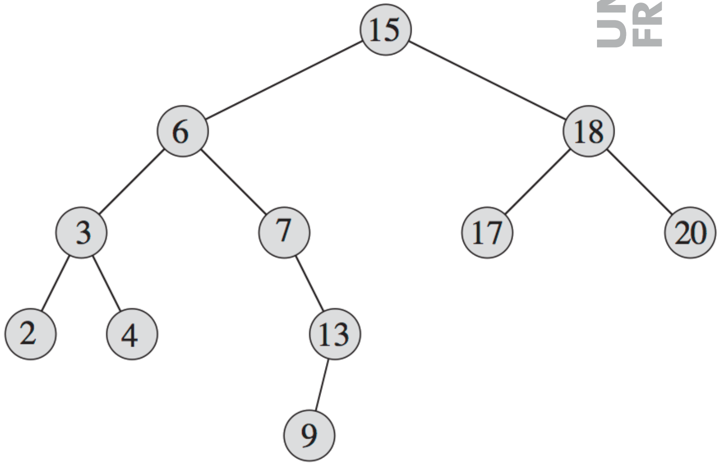
Binary Search Trees

- Binary search trees do not always need to be nice and symmetric…



Source: [CLRS]

Find in a Binary Search Tree



Search for key x

- Use binary search
  - That's way it's called a binary search tree …

Running time: O(depth of tree)

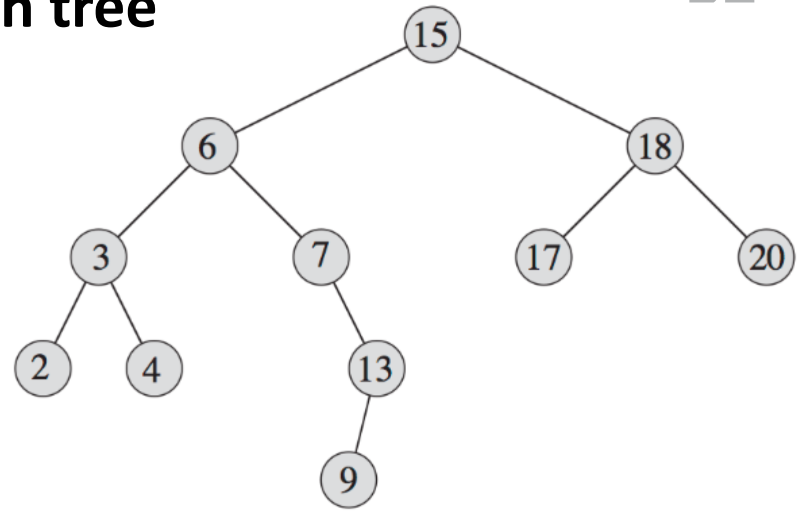
```
current = root
while current is not None and current.key != x:
    if current.key > x:
        current = current.left
    else:
        current = current.right
```

At the end:

- Key x not in the tree : `current == None`
- Key x found : `current.key == x`

Suche Minimum / Maximum

Find smallest element in a binary search tree



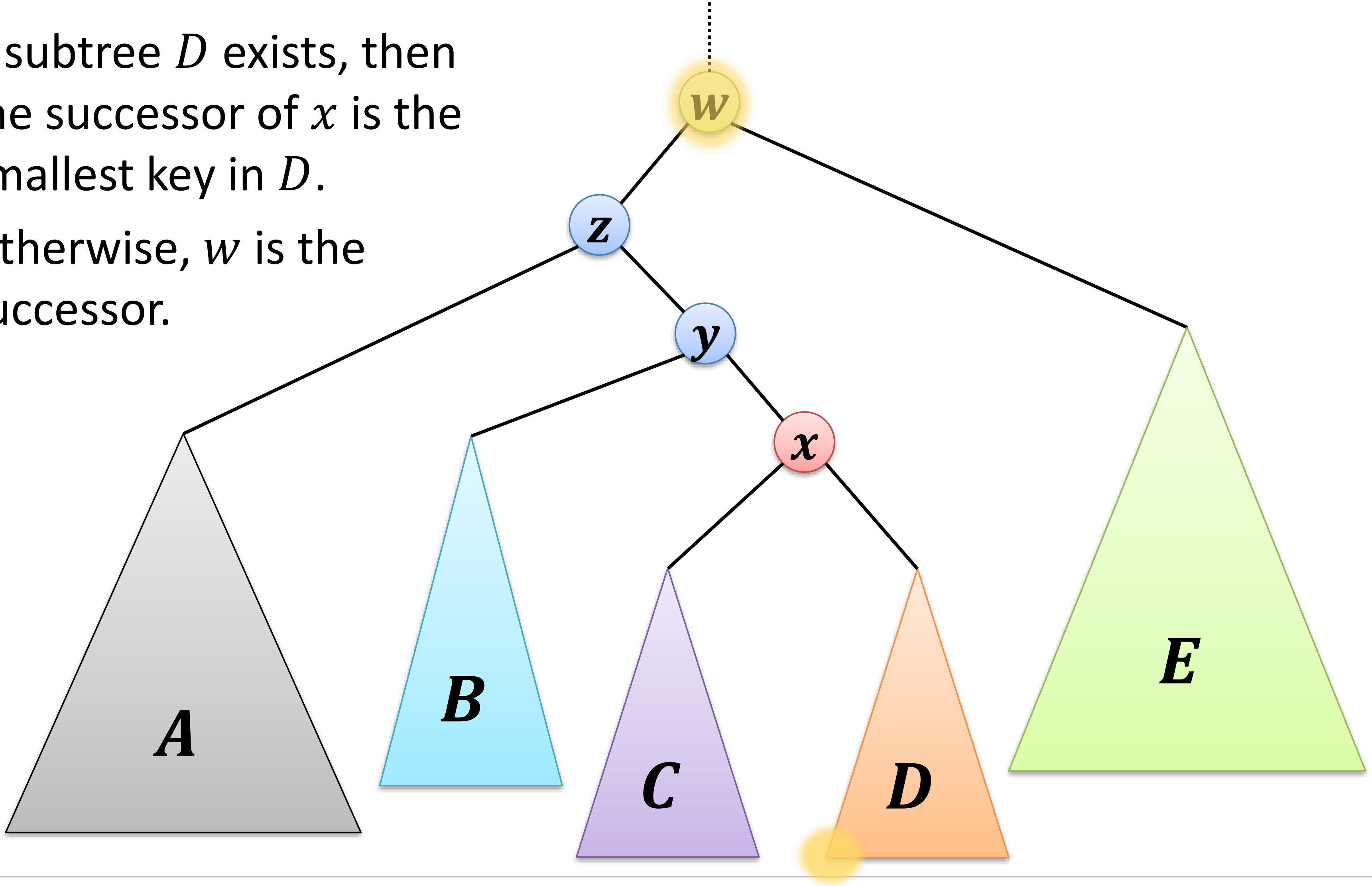
- All smaller elements are always in the left subtree.

```
current = root
while current.left is not None:
    current = current.left
```

Search for Successor

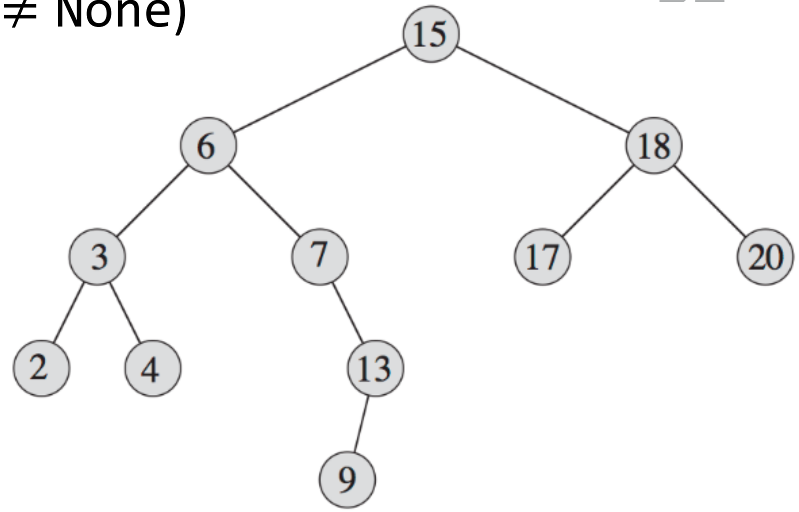
Ordering in tree: $A < z < B < y < C < x < D < w < E$

- If subtree D exists, then the successor of x is the smallest key in D.
- Otherwise, w is the successor.



Search for Successor

Find successor of a node u (assumption: $u \neq$ None)

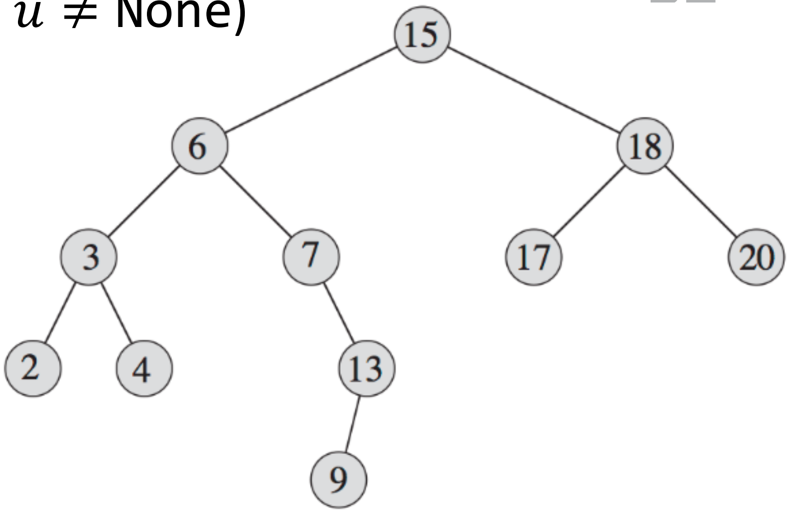


```
if u.right is not None:
    # min in right subtree
    current = u.right
    while current.left is not None:
        current = current.left
    return current
else
    # find first node towards root s.t. u is in left subtree
    current = u
    parent = current.parent
    while parent is not None and current == parent.right:
        current = parent
        parent = current.parent
    return parent
```

Running time: O(depth of tree)

Search for Predecessor

Find predecessor of a node u (assumption: $u \neq$ None)

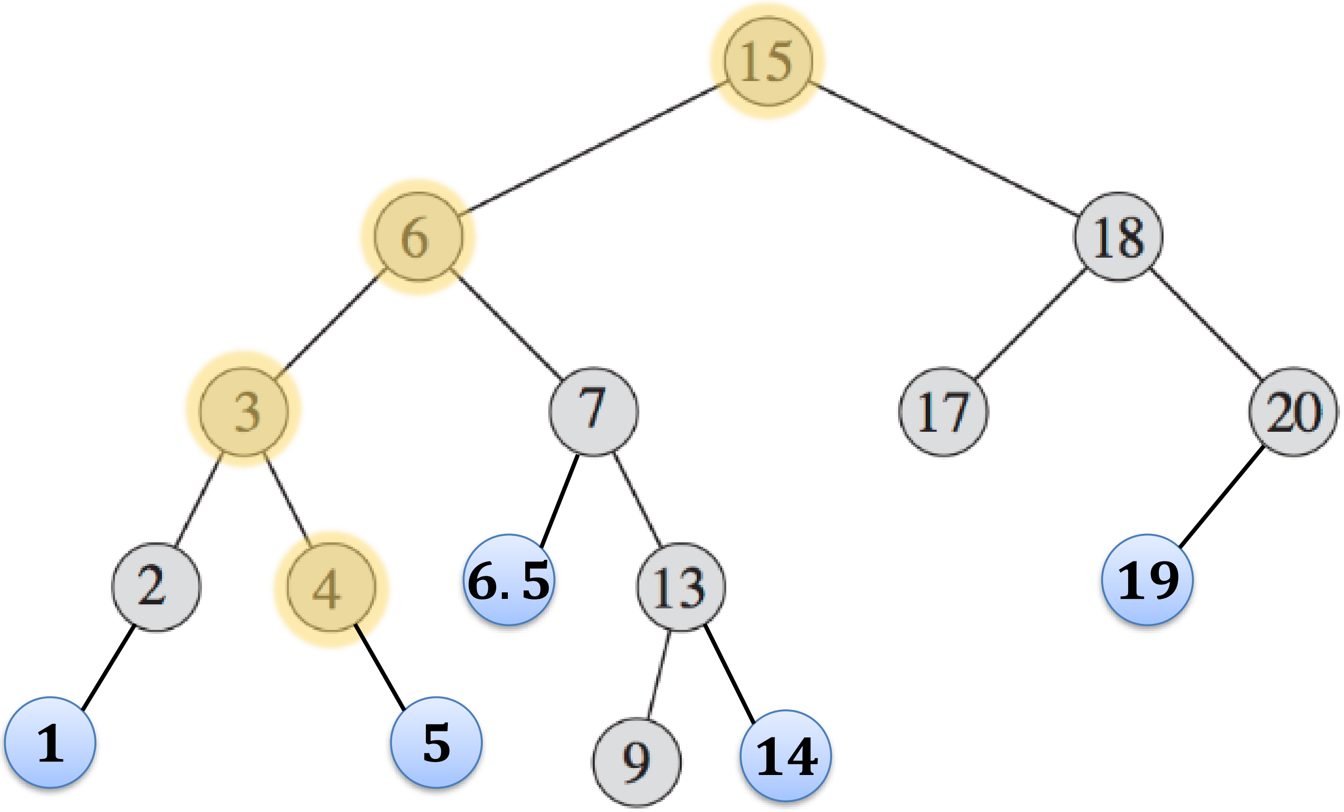


```
if u.left is not None:
    # max in left subtree
    current = u.left
    while current.right is not None:
        current = current.right
    return current
else
    # find first node towards root s.t. u is in right subtree
    current = u
    parent = current.parent
    while parent is not None and current == parent.left:
        current = parent
        parent = current.parent
    return parent
```

Running time: O(depth of tree)

Inserting a Key

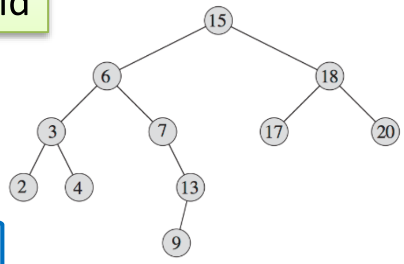
Insert keys 5, 1, 14, 6.5, 19 …



Running time: O(depth of tree)

Inserting a key x with value a

key value parnet left child right child



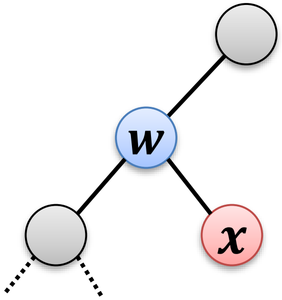
```
if root is None:
    root = new TreeElement(x, a, None, None, None)
else:
    current = root; parent = None
    while current is not None and current.key != x:
        parent = current
        if x < current.key:
            current = current.left
        else:
            current = current.right
    if current is None:   (key x is not contained in tree)
        if x < parent.key:
            parent.left = new TreeElement(x, a, parent, None, None)
        else:
            parent.right = new TreeElement(x, a, parent, None, None)
    else:
        current.value = a  (key x is already contained, replace value)
```

binary search

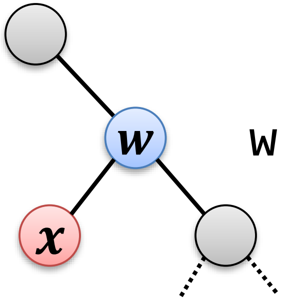
Deleting a Key I

Delete key x, simple cases:

- Key x is in a leaf node u of the tree
  - leaf = node has no children

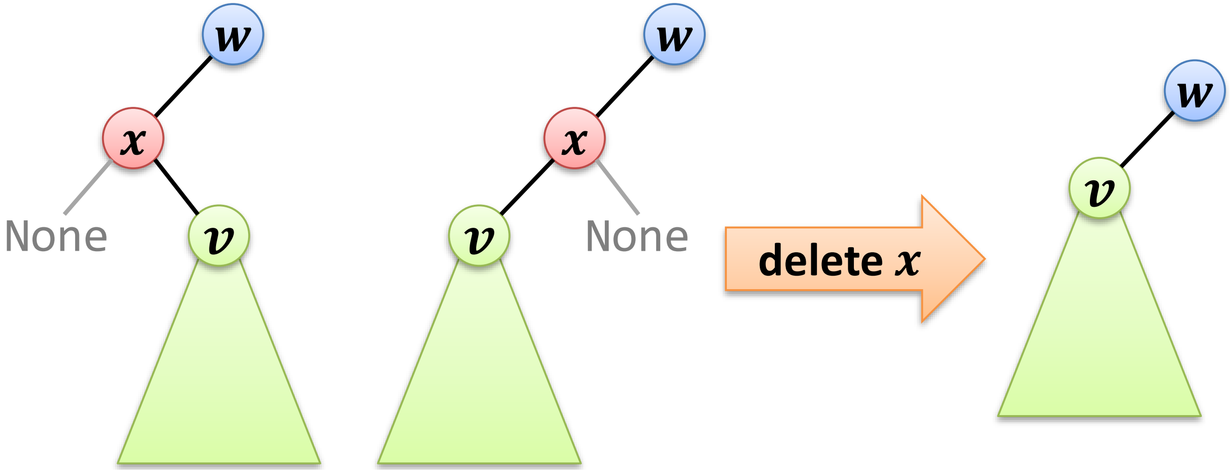


`w.right = None`



`w.left = None`

- Node with key x has only 1 child



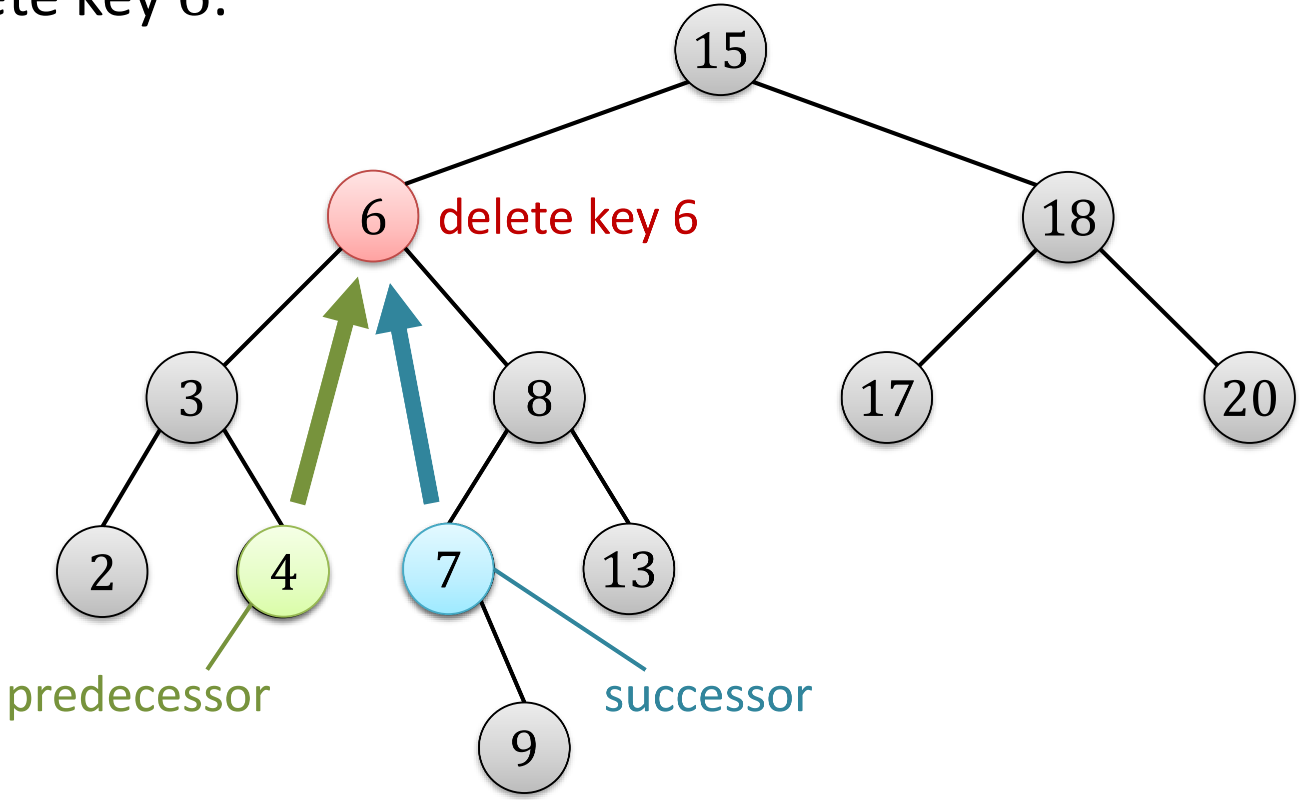
`w.left = v`

Case, where x is the right child of w is symmetric.

Deleting a Key II

Delete key x, node has two children:

- Delete key 6:



Deleting a Key III

Delete key x, node has two children:

- Predecessor is largest key in left subtree.
  - Predecessor has no right child.
- Successor is smallest key in right subtree.
  - Successor has no left child.

- Write key and data of precedessor (or alternatively successor) to the node of key x
- Delete predecessor / successor node
  - Predecessor / successor is either a leaf or it has only one child.

Deleting a Key IV

Delete key x:

1. Find node u with $u.\mathit{key} = x$
   - as usual, by using binary search
2. If u does not have 2 children, delete node u
   - Assumption: v is parent of u, u is left child of v (other case is symmetric)
   - If u is a leaf, we do $v.\mathit{left} = \text{None}$
   - If u has one child w, we do $v.\mathit{left} = w$
3. If u has two children, determine predecessor node v
   - also works with successor node
4. Set $u.\mathit{key} = v.\mathit{key}$ and $u.\mathit{data} = v.\mathit{data}$
5. Delete node v (in the same way as deleting u above)
   - Node v has at most 1 child!

Running time: O(depth of tree)

Running Times Binary Search Tree

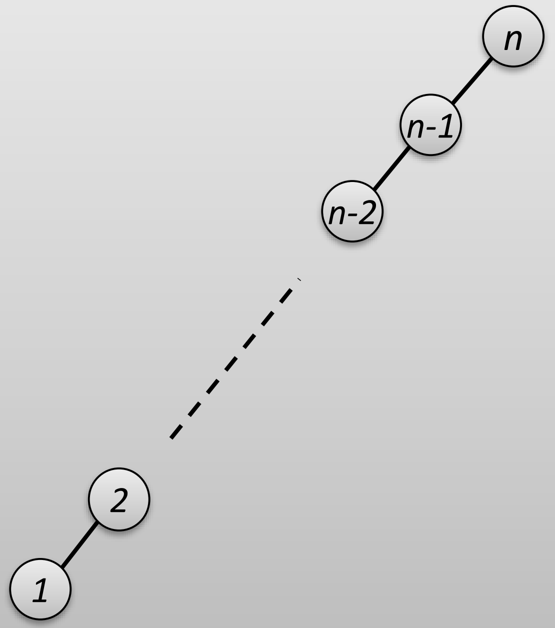
The operations

find, min, max, predecessor, successor, insert, delete

all have **running time O(depth of tree)**.

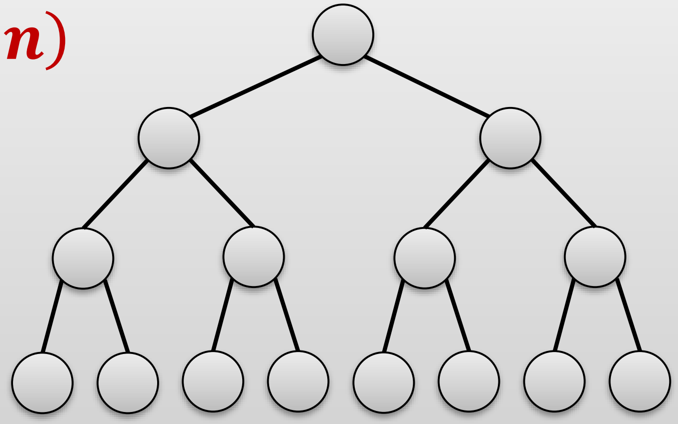
What is the depth of a binary search tree?

Worst Case: $\Theta(n)$



Best Case: $\Theta(\log n)$

- max. #nodes in depth k is 2^k
- depth $\geq \lfloor \log_2 n \rfloor$



Average Case: $\Theta(\log n)$

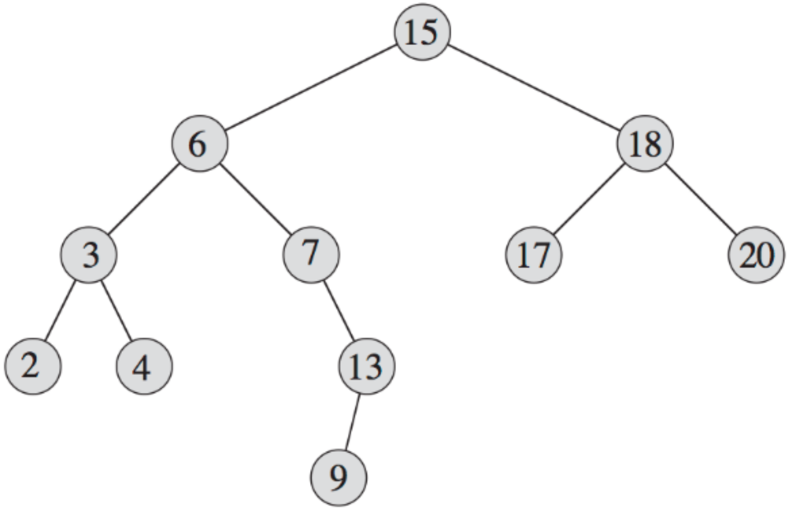
- If the keys are inserted in random order, the depth is $O(\log n)$
- typical case?

Sorting with a Binary Search Tree

1. Insert all element into a binary search tree
2. Read out the elements in sorted order
    - Simplest solution: always find and delete minimum
    - Or better: find minimum and afterwards $n-1$ times *successor*

Better solution: reading out all elements:

- Recursively:
    1. Read out left subtree (recursively)
    2. Read out root
    3. Read out right subtree (recursively)



Running time for depth $\boldsymbol{O(\log n)}$:

- Insert: $O(n \cdot \log n)$
- Read out: $O(n)$

Reading Out a Part of the Elements

Given: keys $x_{\min}$ and $x_{\max}$ $(x_{\min} \leq x_{\max})$

Goal: Output **all keys x** with $x_{\min} \leq x \leq x_{\max}$.

deal with subtree of u

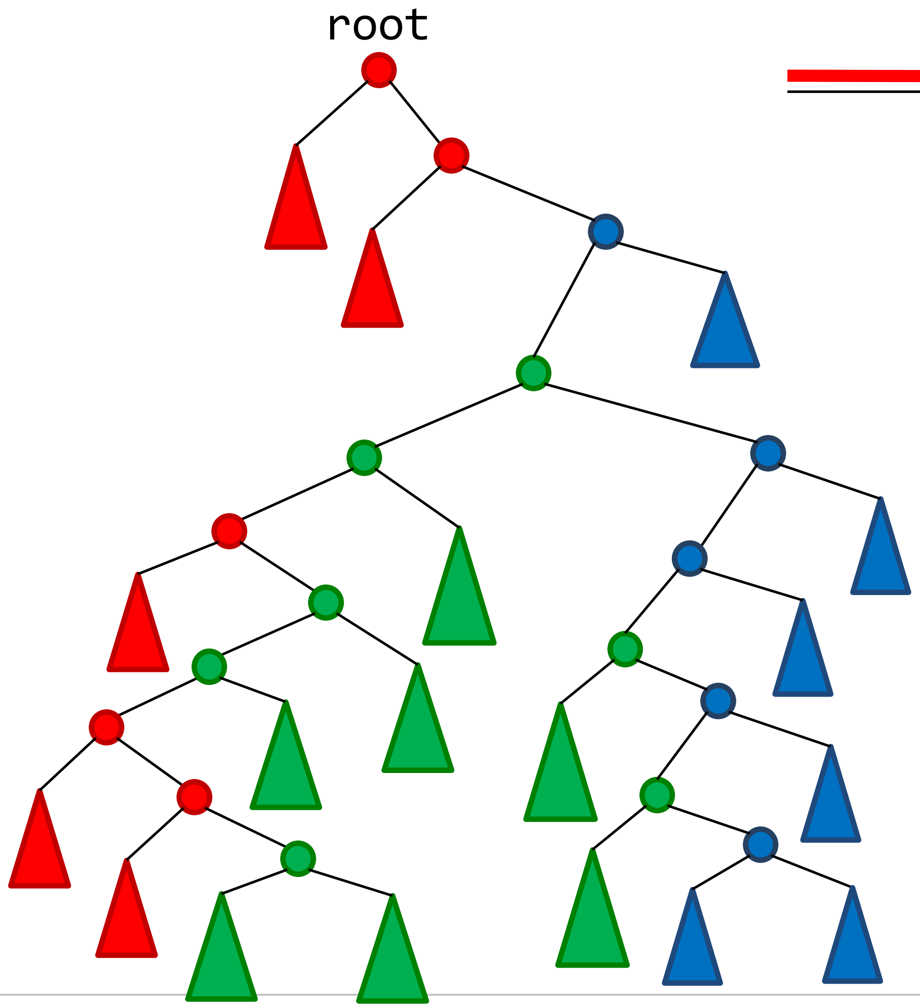
```
getrange(u, xmin, xmax):
    if u is not None:
        if u.key > xmin:
            getrange(u.left, xmin, xmax)
        if (xmin <= u.key) and (u.key <= xmax):
            add u to output
        if u.key < xmax:
            getrange(u.right, xmin, xmax)
```

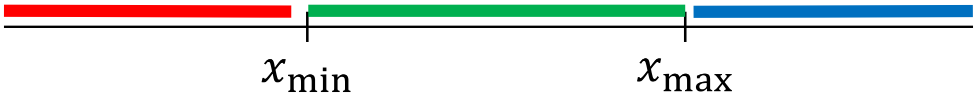
- **Assumption:** #keys in range $[x_{\min}, x_{\max}]$ is equal to k
- **Running time:** certainly $O(n)$ and certainly also $\Omega(k + \text{depth})$

Reading Out a Part of the Elements

Given: keys $x_{\min}$ and $x_{\max}$ $(x_{\min} \leq x_{\max})$

Goal: Output **all keys x** with $x_{\min} \leq x \leq x_{\max}$.





Number of visited nodes:

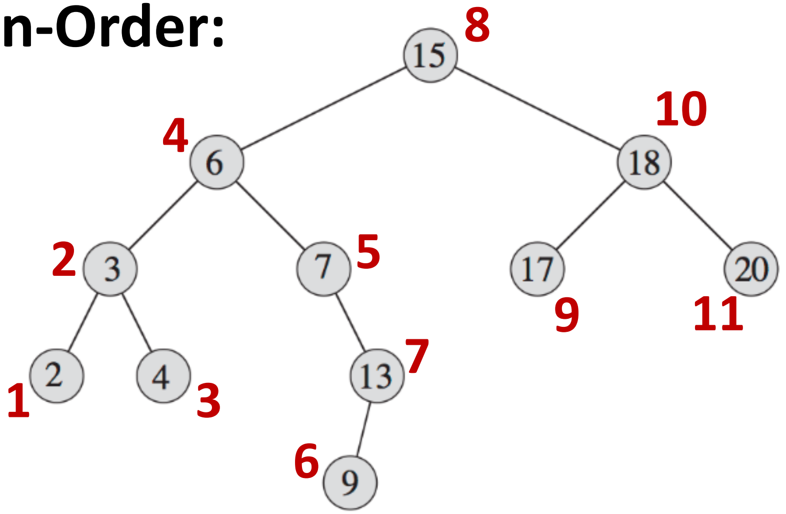
- **#grün $= k$**
- **#rot $\leq$ Tiefe**
- **#blau $\leq$ Tiefe**

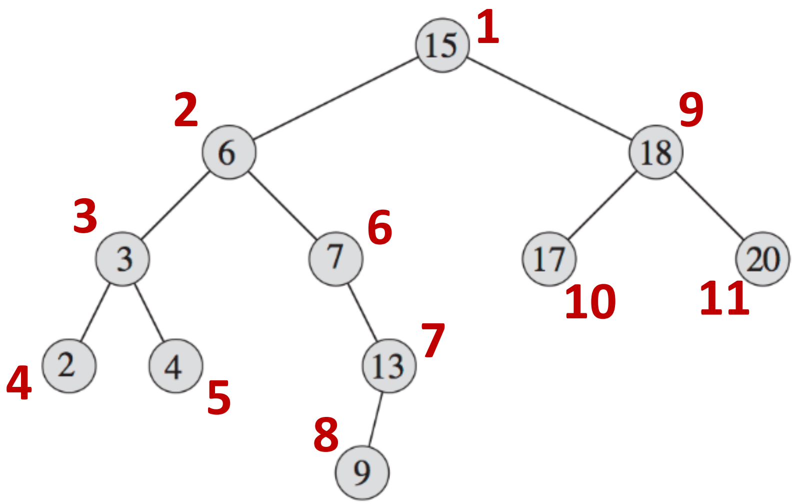
Running time: $O(k + \text{depth})$

Traversal of a Binary Search Tree

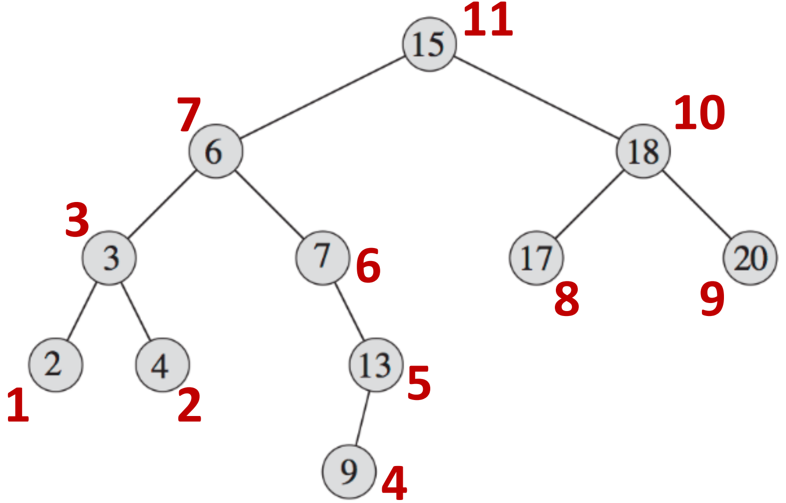
Goal: visit all nodes of a binary search tree once.

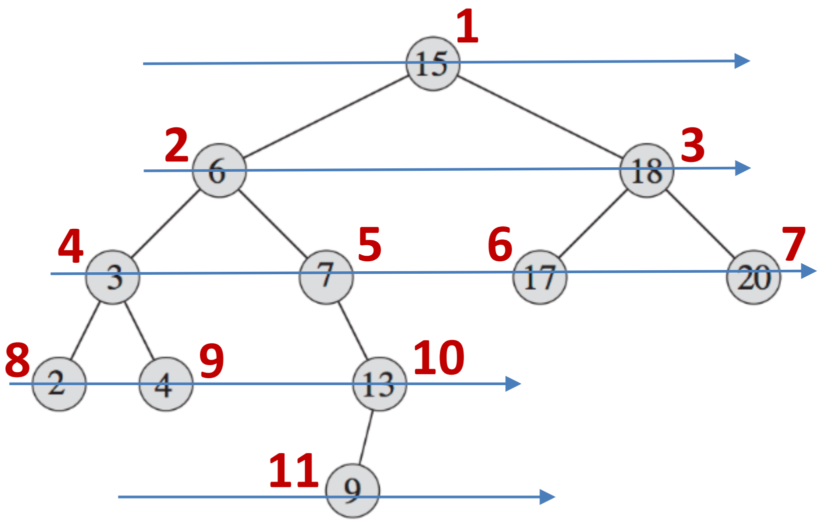
In-Order:





Post-Order:





Traversal of a Binary Search Tree

Depth First Search / DFS Traversal

Pre-Order: 15, 6, 3, 2, 4, 7, 13, 9, 18, 17, 20

In-Order: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20

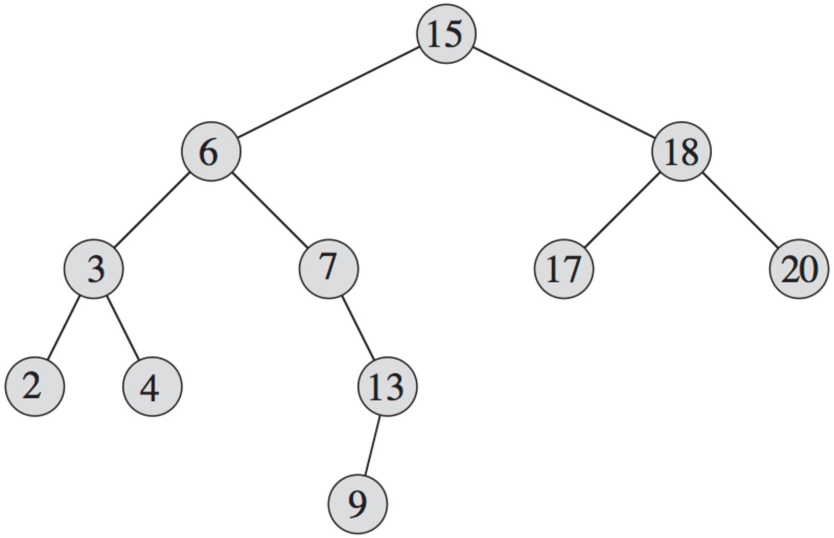
Post-Order: 2, 4, 3, 9, 13, 7, 6, 17, 20, 18, 15

recursively

Breadth First Search / BFS Traversal

Level-Order: 15, 6, 18, 3, 7, 17, 20, 2, 4, 13, 9

- Does not work in the same way
  ⟹ we will afterwards look at this



DFS Traversal

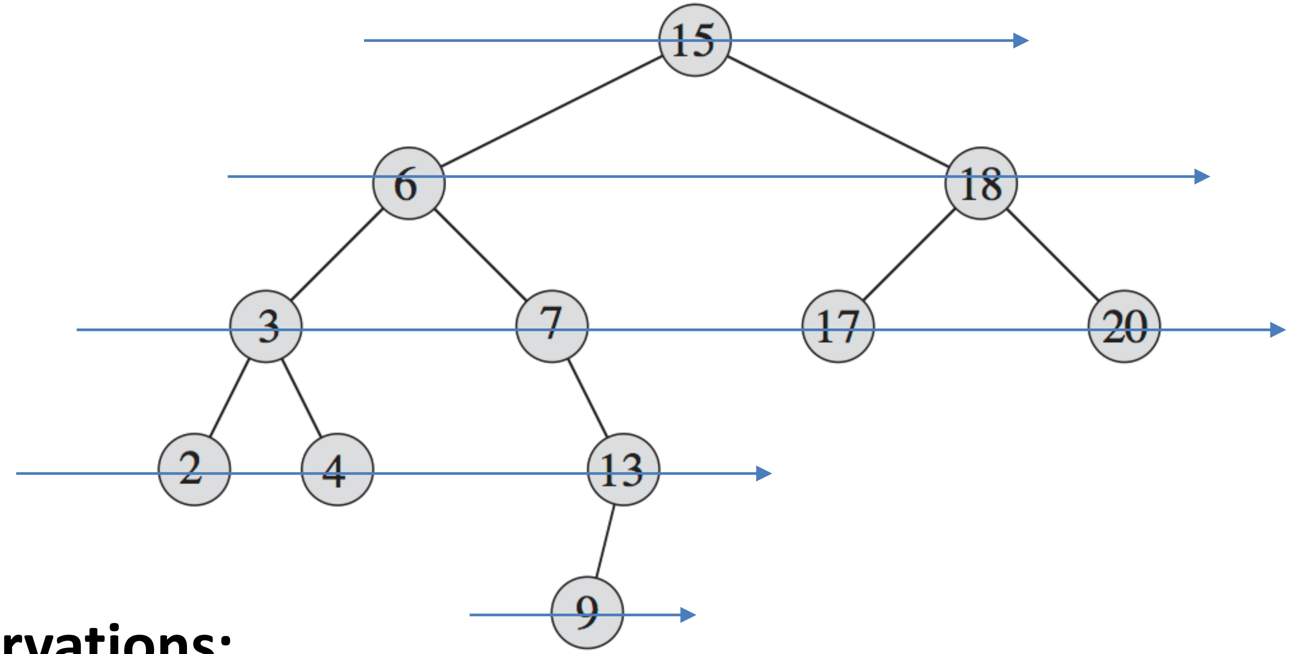
```
preorder(node):
    if node is not None:
        visit(node)
        preorder(node.left)
        preorder(node.right)

inorder(node):
    if node is not None:
        inorder(node.left)
        visit(node)
        inorder(node.right)

postorder(node):
    if node is not None:
        postorder(node.left)
        postorder(node.right)
        visit(node)
```

BFS Traversal

- Does not work recursively as for DFS traversal

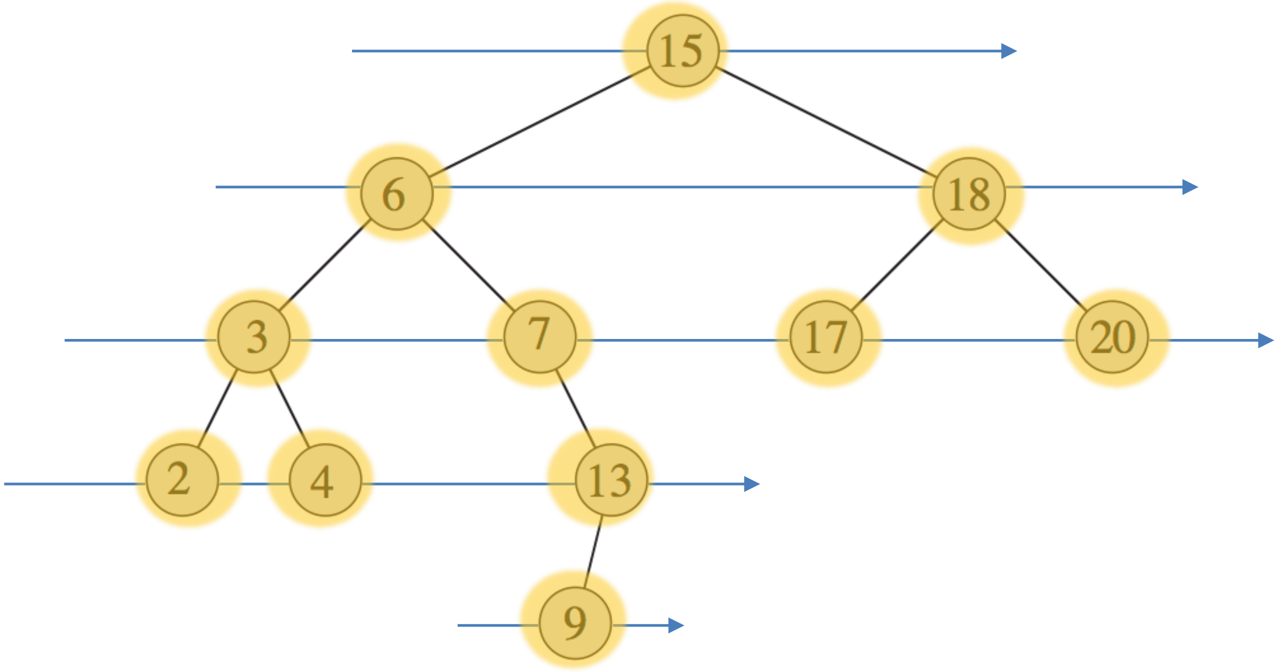


- **Observations:**
  - The root of a subtree is always visited before its children
  - If a node u is visited before node v, then also the children of node u are visited before the children of node v.

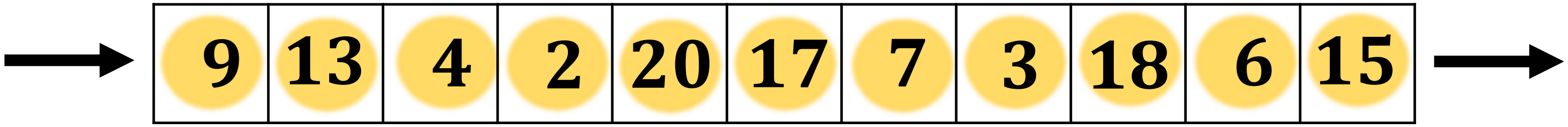
  - **Idea:** Use a FIFO queue: when visiting u, then the children of u are inserted into the FIFO queue.

BFS Traversal

- Does not work recursively as for DFS traversal



FIFO Queue:



BFS Traversal

- Does not work recursively as for DFS traversal
- Solution with a FIFO queue:
  - When visiting a node, insert its children into a FIFO queue

```
BFS-Traversal:
    Q = new Queue()
    Q.enqueue(root)
    while not Q.empty():
        node = Q.dequeue()
        visit(node)
        if node.left is not None:
            Q.enqueue(node.left)
        if node.right is not None:
            Q.enqueue(node.right)
```

Analysis Tree Traversal

DFS Traversal:

- Each node is visited exactly once
- Time cost per node: $O(1)$
- **Overall time** for DFS traversal: $\boldsymbol{O(n)}$

BFS Traversal:

- Each node is visited exactly once
  - Cost per node is linear in the number of children, i.e., $O(1)$ for binary trees
  - Each node is inserted into the FIFO queue exactly once
- Cost per node: $O(1)$
- **Overall time** for BFS traversal: $\boldsymbol{O(n)}$

Application DFS Traversal I

In-order traversal:

- Visits all elements of a binary search tree in sorted order
- Sorting:
    1. Insert all elements
    2. In-order traversal

- Observation: Order only depends on the set of elements (keys) and not on the structure of the tree.

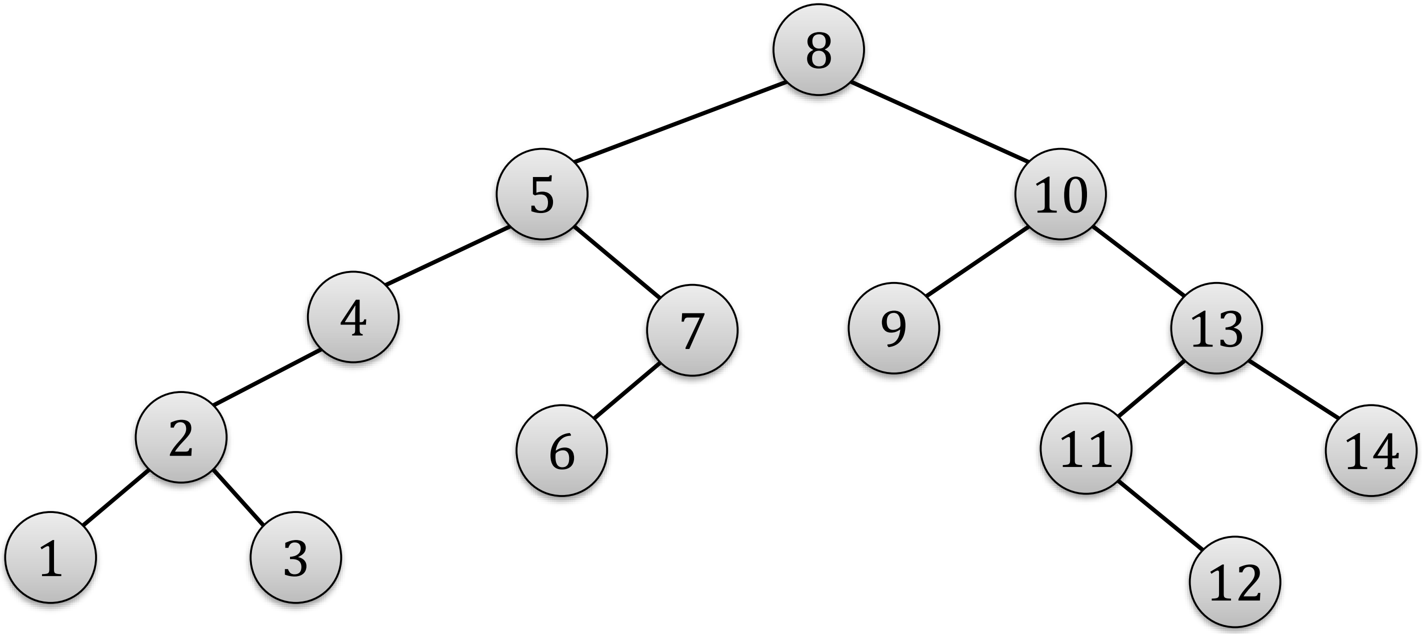
Application DFS Traversal II

Pre-order traversal:

- From a pre-order traversal sequence, the tree can be reconstructed uniquely (and efficiently).
- Useful to store a tree, e.g., in a file

Example: 8, 5, 4, 2, 1, 3, 7, 6, 10, 9, 13, 11, 12, 14

left subtree right subtree



Application DFS Traversal III

Post-order traversal:

- Deleting a whole binary search tree
- First, one has to free the memory of the substrees before freeing the memory of the root node.

```
delete-tree(node)
    if (node != None)
        delete-tree(node.left)
        delete-tree(node.right)
        delete node
```

Efficiency of a Binary Search Tree

Worst case running time of the operations
find, min, max, predecessor, successor, insert, delete:

$$\boldsymbol{O}(\textbf{depth of tree})$$

- In the **best case**, the depth is $\boldsymbol{\log_2 n}$
  - Definition depth: length of longest path from the root to a leaf
- In the **worst case**, the depth is $\boldsymbol{n-1}$
- In the **average case**, the depth is $\boldsymbol{O(\log n)}$
  - Average case here means a random insertion order

Next lecture: How can we guarantee that the depth of a binary search tree is always $O(\log n)$?