Institute of
Pervasive Computing

TREES

Algorithms and Data Structures 1
Exercise – 2023S

Markus Jäger (Computer Science)
Florian Beck (Artificial Intelligence)
Bernhard Anzengruber (Artificial Intelligence)

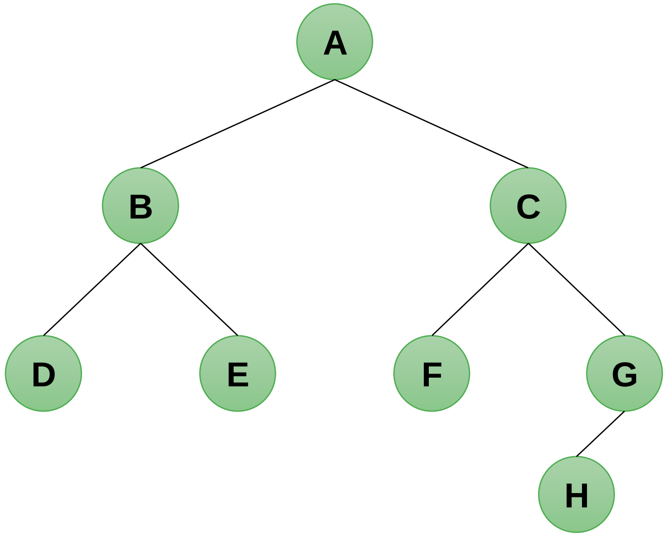
Institute of Pervasive Computing
Johannes Kepler University Linz

markus.jaeger@jku.at
florian.beck@jku.at
bernhard.anzengruber-tanase@jku.at

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
jku.at

BINARY TREE :: REVIEW

A	root
B	is parent of D and E
C	is sibling of B
D and E	are children of B
D, E, F, H	are external nodes or leaves
A, B, C, G	are internal nodes



The depth of E is 2.

The height of the tree is 3.

The order of B is 2, the order of G is 1.

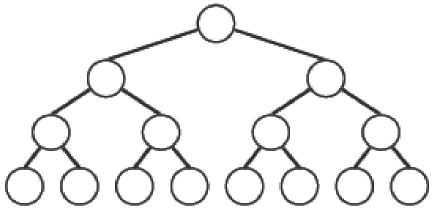
Number of edges is always equal to the number of nodes-1

BINARY TREE :: REVIEW

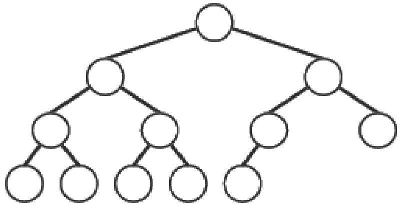
A tree in which each node has a **maximum of 2** subtrees

Each node has **0, 1 or 2 child nodes**

A binary tree of height *h* is **complete**, if it contains $2^{h+1} - 1$ nodes (all leaves have the same depth): e.g.: $h=3 \rightarrow 2^{3+1} - 1 = 16 - 1 = \underline{15}$



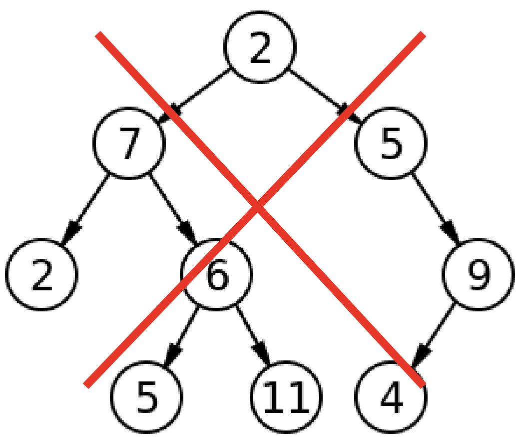
A binary tree of height *h* is **almost complete**, if it contains the maximum number of nodes on level 0 to *h*-1 and if all leaves at level *h* are placed in the leftmost position



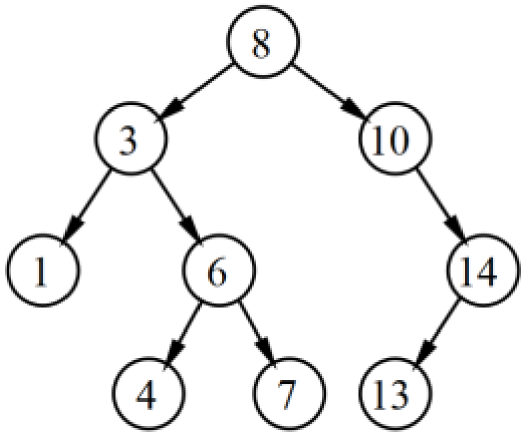
BINARY SEARCH TREE :: PROPERTIES

A **binary search tree** is a binary tree T in which

- each node contains a key
- keys in the left sub-tree of a node *n* are **smaller than** (or equal to) the key stored in *n*
- keys in the right sub-tree of a node *n* are **greater than** the key stored in *n*
- external nodes have no child branches



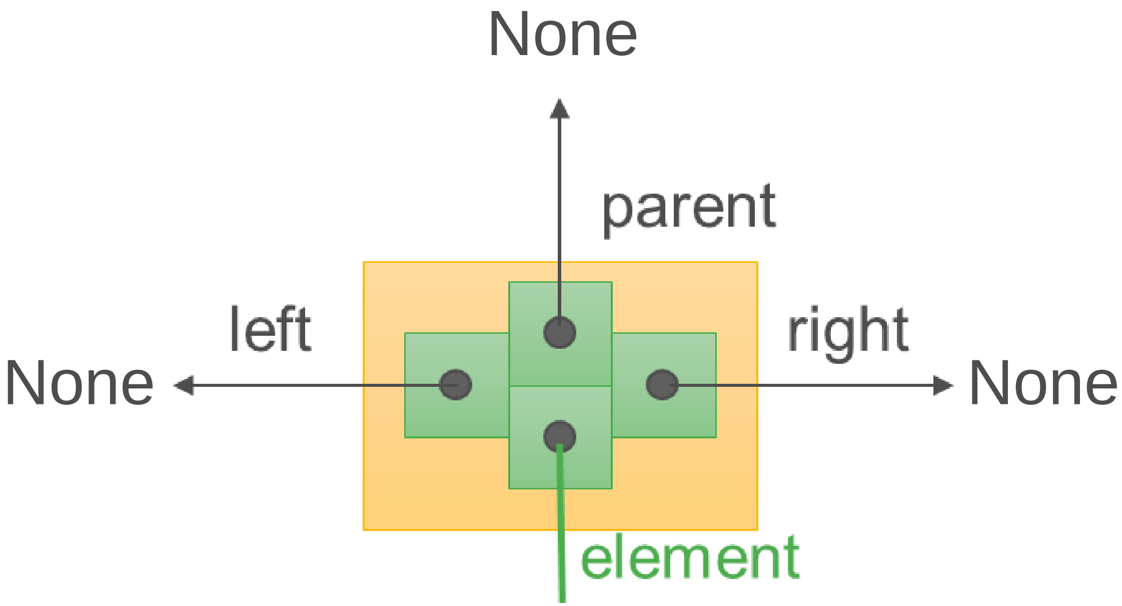
Binary Tree



Binary Search Tree

BINARY SEARCH TREE :: STRUCTURE

Link structure of a binary search tree

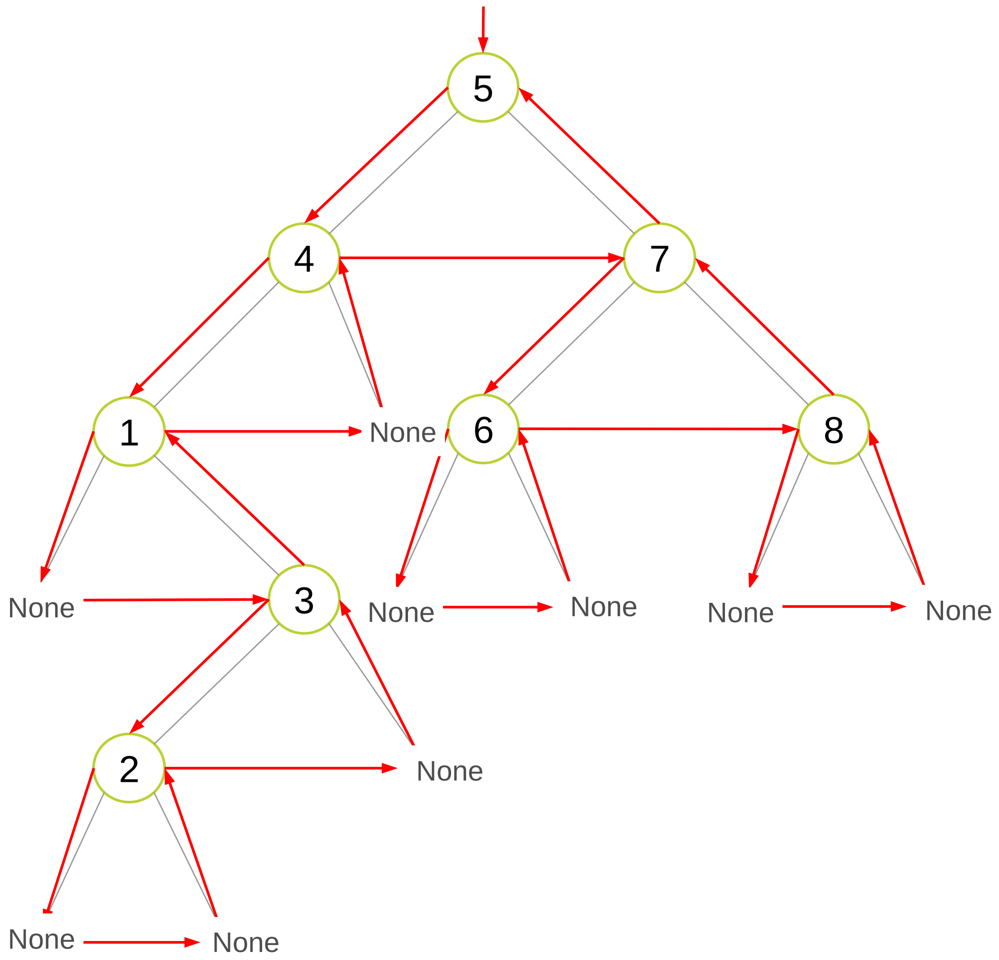



```
class TreeNode:
    def __init__(self, element=None, parent=None,
                 right=None, left=None):
        self.element = element
        self.parent_node = parent
        self.right_node = right
        self.left_node = left
```

BINARY SEARCH TREE :: TRAVERSAL

Systematically processing all nodes in a tree

- **postOrder traversal**
  - left, right, root
  - 2 – 3 – 1 – 4 – 6 – 8 – 7 – 5
- **preOrder traversal**
  - root, left, right
  - 5 – 4 – 1 – 3 – 2 – 7 – 6 – 8
- **inOrder traversal**
  - left, root, right
  - 1 – 2 – 3 – 4 – 5 – 6 – 7 – 8



BINARY SEARCH TREE :: INSERT

Greater elements on the right side

Smaller (or equal) elements on the left side

tree.insert(5)

tree.insert(18)

tree.insert(1)

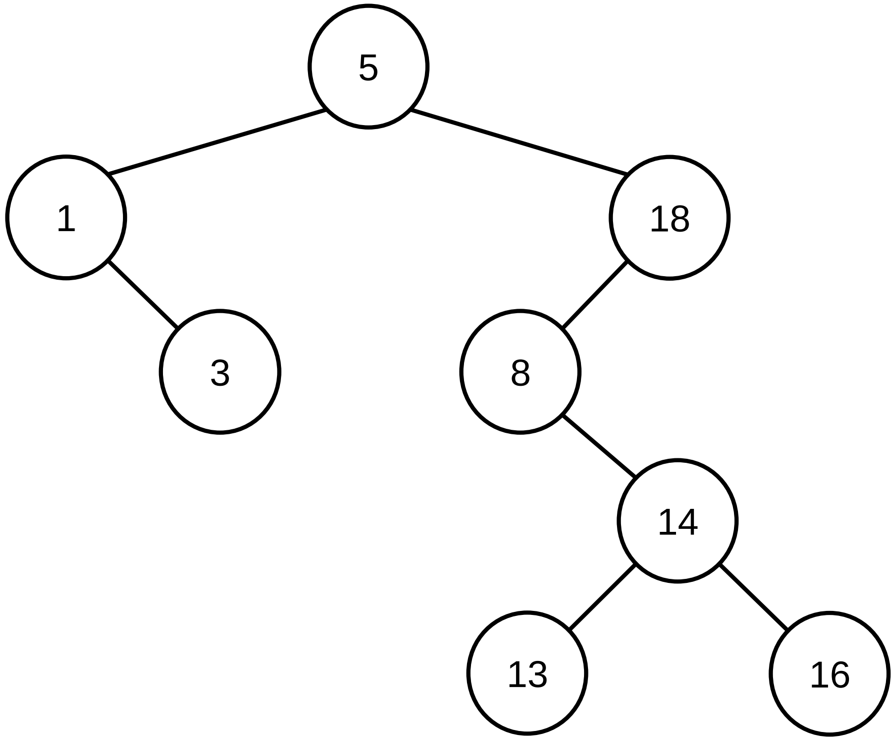
tree.insert(8)

tree.insert(14)

tree.insert(16)

tree.insert(13)

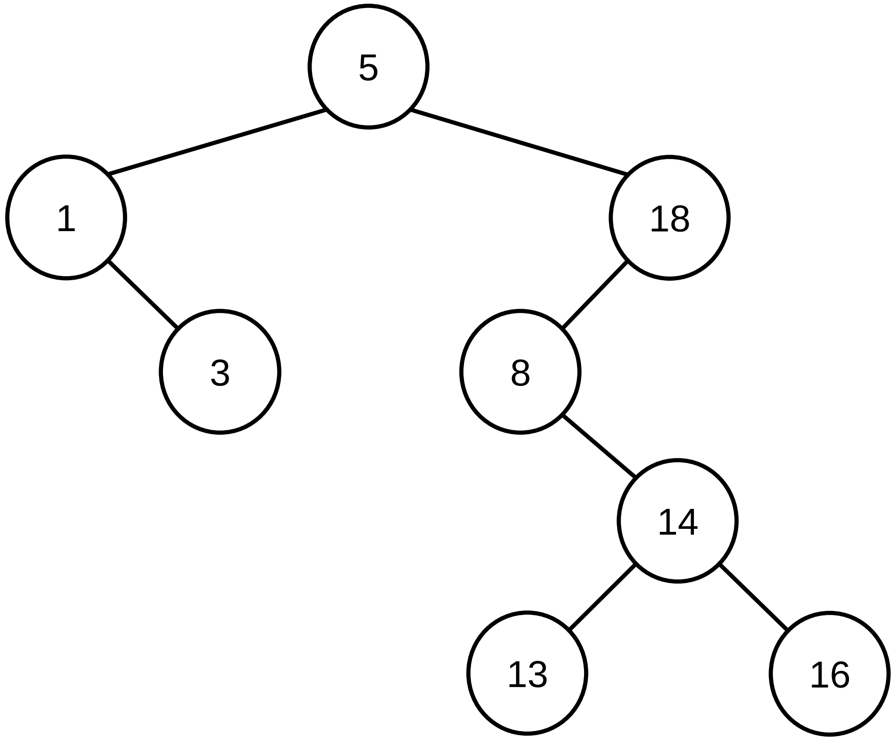
tree.insert(3)



BINARY SEARCH TREE :: SEARCH

Termination criteria

- key is found
- search terminates in a leaf



BINARY SEARCH TREE :: SEARCH

Termination criteria

- key is found
- search terminates in a leaf

Iterative

```
def find(key, n):
  while n!=None and n.key!=key:
    if key < n.key:
      n = n.left
    else:
      n = n.right

  return n
```

Recursive

```
def find(key, n):
  if n==None:
    return None
  if n.key==key:
    return n
  if key < n.key:
    return find(key, n.left)
  return find(key, n.right)
```

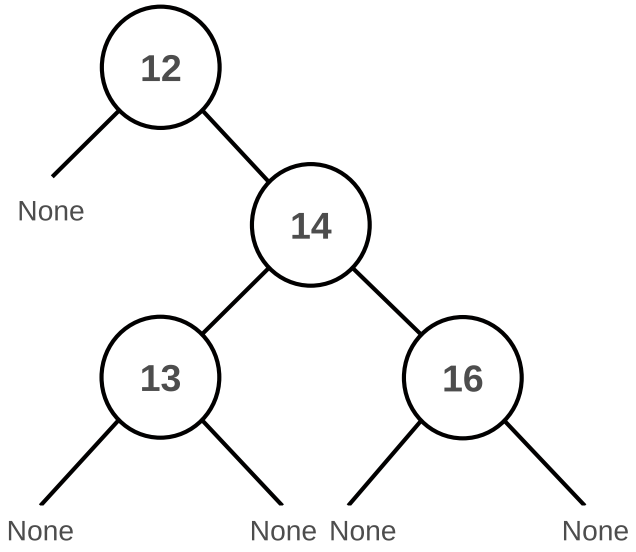
- When searching for a non-existent element, we discover exactly where the element should be inserted → **Reuse code!**

BINARY SEARCH TREE :: REMOVE

Different cases with different solutions

Case 1: If node has no child node, it can simply be removed

- Search node n
- Store parent node p_n of n
- If $n < p_n \rightarrow (p_n).left = \text{None}$
- If $n > p_n \rightarrow (p_n).right = \text{None}$



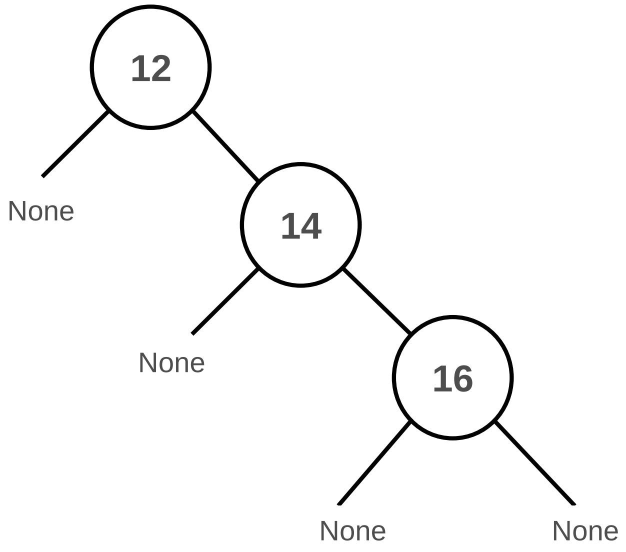
tree.remove(13)

BINARY SEARCH TREE :: REMOVE

Different cases with different solutions

Case 1: If node has no child node, it can simply be removed

- Search node n
- Store parent node p_n of n
- If $n < p_n \rightarrow (p_n).left\ = \text{None}$
- If $n > p_n \rightarrow (p_n).right\ = \text{None}$



tree.remove(13)

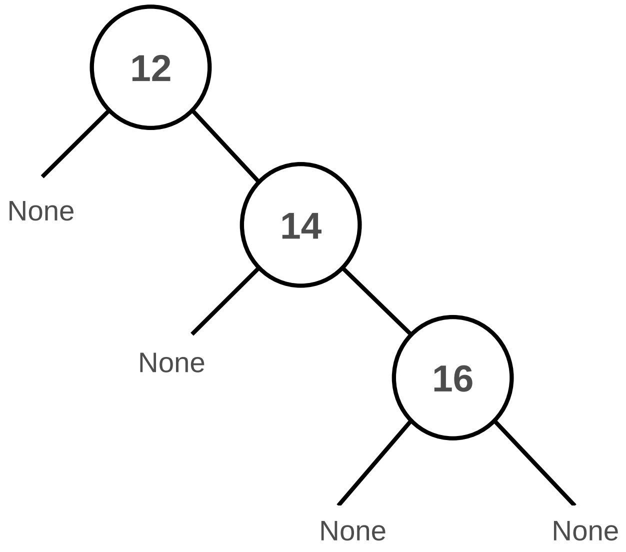
BINARY SEARCH TREE :: REMOVE

Different cases with different solutions

Case 1: If node has no child node, it can simply be removed

Case 2: If node has only one child node, replace it by child node

- Search node n
- Store parent node p_n of n
- Store child node c_n of n
- If $n < p_n \rightarrow (p_n).left\ = c_n$
- If $n > p_n \rightarrow (p_n).right\ = c_n$



tree.remove(14)

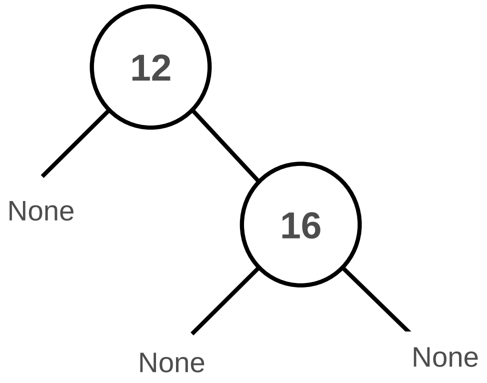
BINARY SEARCH TREE :: REMOVE

Different cases with different solutions

Case 1: If node has no child node, it can simply be removed

Case 2: If node has only one child node, replace it by child node

- Search node n
- Store parent node p_n of n
- Store child node c_n of n
- If $n < p_n \rightarrow (p_n).left \ = c_n$
- If $n > p_n \rightarrow (p_n).right \ = c_n$



tree.remove(14)

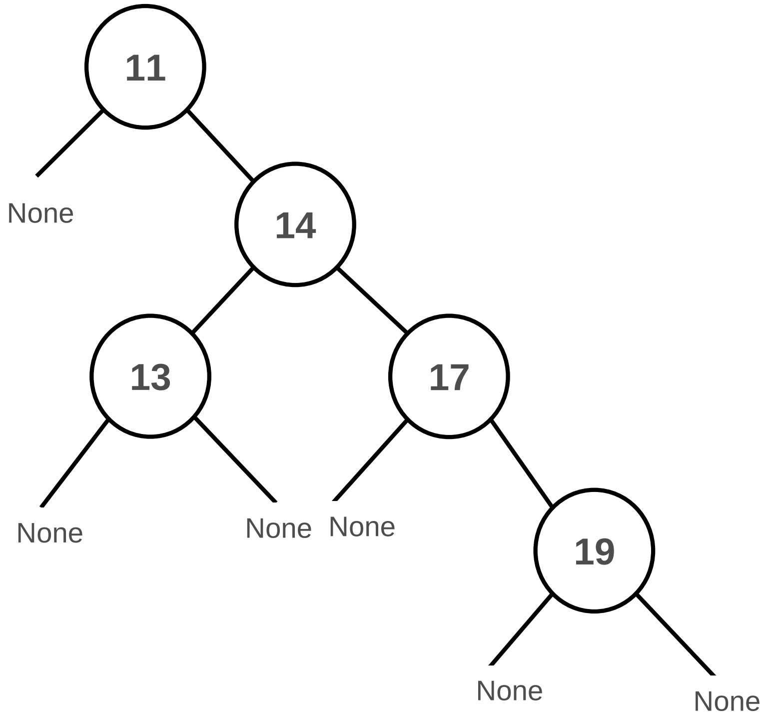
BINARY SEARCH TREE :: REMOVE

Different cases with different solutions

Case 1: If node has no child node, it can simply be removed

Case 2: If node has only one child node, replace it by child node

Case 3: If node has two child nodes, we have to distinguish further cases



tree.remove(14)

BINARY SEARCH TREE :: REMOVE

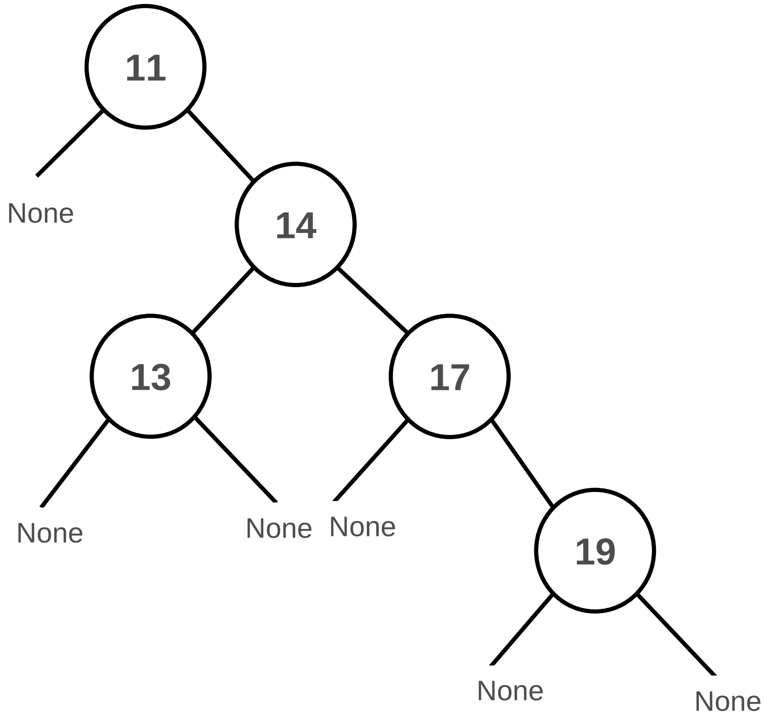
Different cases with different solutions

Case 3a: Node has two child nodes, optimizable

If

- rc_n or lc_n has no child,
- lc_n has no right child,
- rc_n has no left child,

n can be efficiently replaced by this child without violating the order relation



tree.remove(14)

BINARY SEARCH TREE :: REMOVE

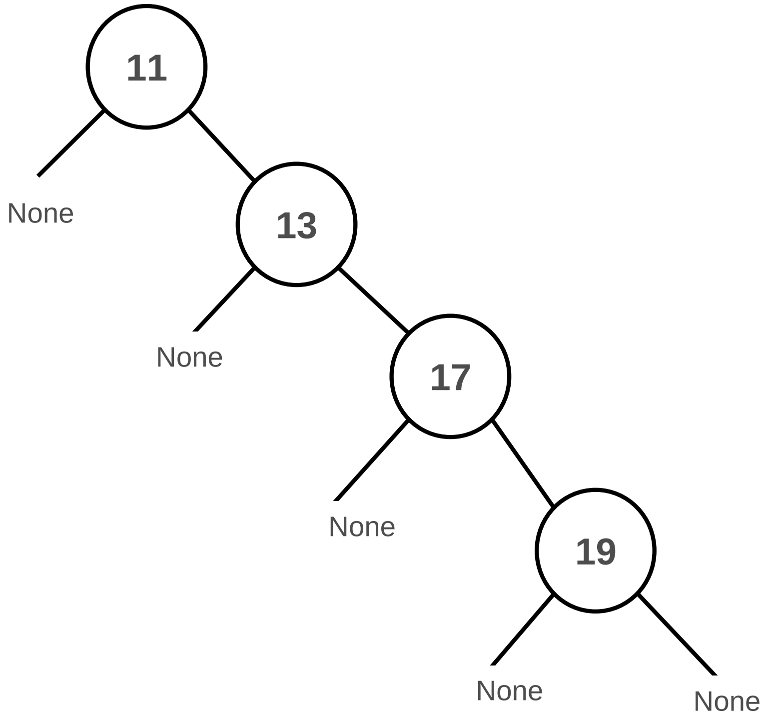
Different cases with different solutions

Case 3a: Node has two child nodes, optimizable

If

- rc_n or lc_n has no child,
- lc_n has no right child,
- rc_n has no left child,

n can be efficiently replaced by this child without violating the order relation



tree.remove(14)

BINARY SEARCH TREE :: REMOVE

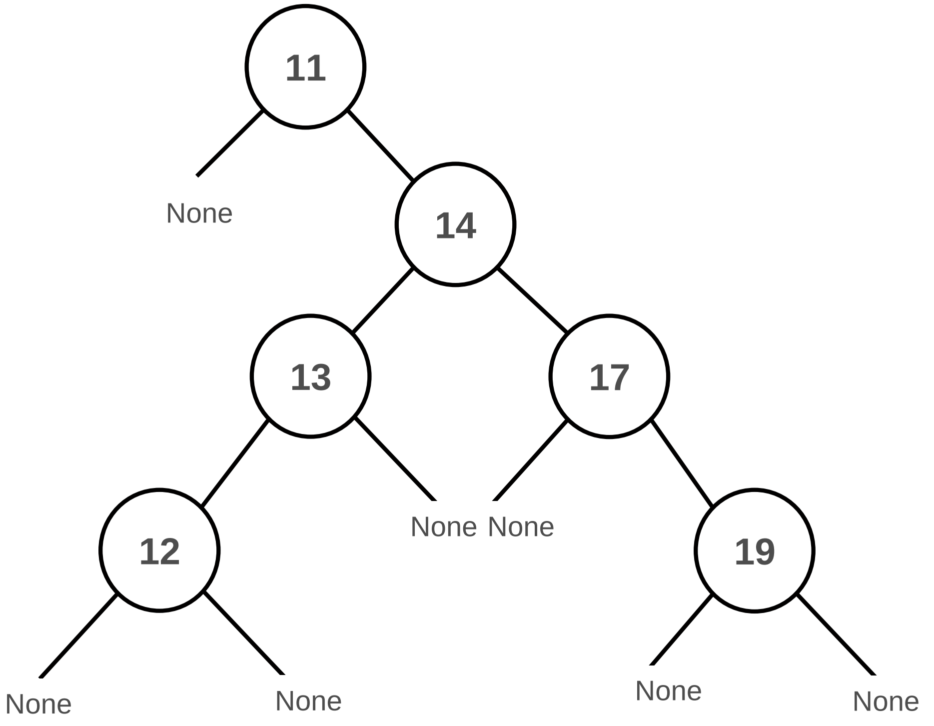
Different cases with different solutions

Case 3a: Node has two child nodes, optimizable

If

- rc_n or lc_n has no child,
- lc_n has no right child,
- rc_n has no left child,

n can be efficiently replaced by this child without violating the order relation



tree.remove(14)

BINARY SEARCH TREE :: REMOVE

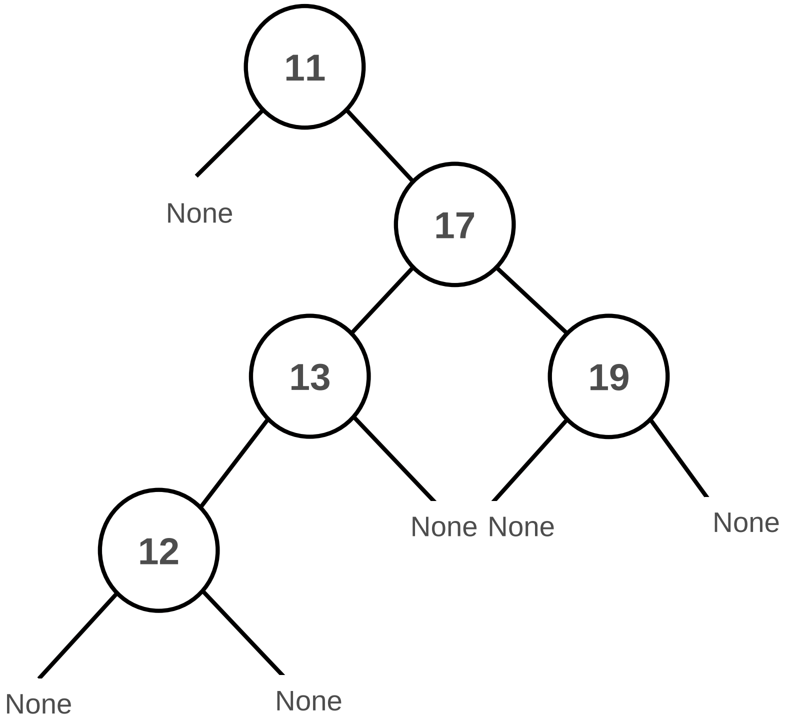
Different cases with different solutions

Case 3a: Node has two child nodes, optimizable

If

- rc_n or lc_n has no child,
- lc_n has no right child,
- rc_n has no left child,

n can be efficiently replaced by this child without violating the order relation



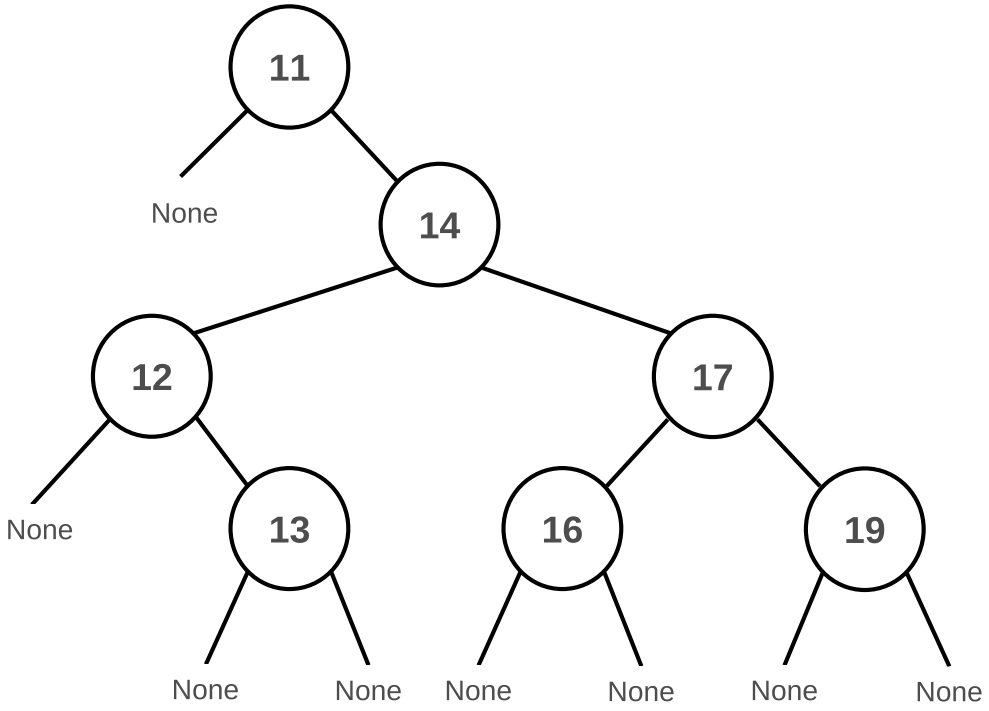
tree.remove(14)

BINARY SEARCH TREE :: REMOVE

Different cases with different solutions

Case 3b: Node has two child nodes,
not optimizable

- Search n, p_n, rc_n, lc_n
- Search InOrder-successor n_{+1}
  - > right-left-left-left-...-None
  - > The left child node of n_{+1} is None
- Store parent node $p_{n_{+1}}$
- Store right child node $rc_{n_{+1}}$ of n_{+1}
- $(p_{n_{+1}}).left = rc_{n_{+1}}$
- Swap n, n_{+1} and correct Parent-Child-Relation



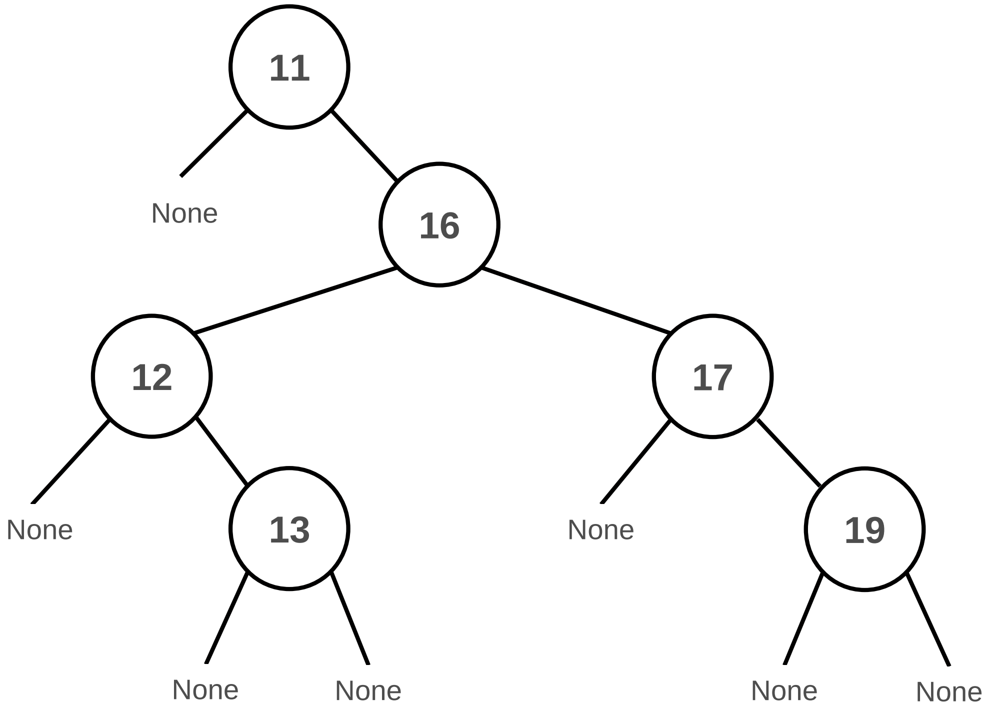
tree.remove(14)

BINARY SEARCH TREE :: REMOVE

Different cases with different solutions

Case 3b: Node has two child nodes,
not optimizable

- Search n, p_n, rc_n, lc_n
- Search InOrder-successor n_{+1}
  - > right-left-left-left-...-None
  - > The left child node of n_{+1} is None
- Store parent node $p_{n_{+1}}$
- Store right child node $rc_{n_{+1}}$ of n_{+1}
- $(p_{n_{+1}}).left = rc_{n_{+1}}$
- Swap n, n_{+1} and correct Parent-Child-Relation



tree.remove(14)

BINARY SEARCH TREE :: REMOVE

Different cases with different solutions

- **Case 1:** node has no child node
  → node can simply be removed

- **Case 2:** node has only one child node
  → replace node by child node

- **Case 3a:** node has two child nodes, optimizable
- **Case 3b:** node has two child nodes, not optimizable

- **Special case**
  - What happens if the element to be removed is the root?
  - What happens if the tree is empty?

TREES

Algorithms and Data Structures 1
Exercise – 2023S

Markus Jäger (Computer Science)
Florian Beck (Artificial Intelligence)
Bernhard Anzengruber (Artificial Intelligence)

Institute of Pervasive Computing
Johannes Kepler University Linz

markus.jaeger@jku.at
florian.beck@jku.at
bernhard.anzengruber-tanase@jku.at

JOHANNES KEPLER
UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
jku.at