Lecture 11.
Binary Search Tree.

Algorithms and Data Structures
Ivan Solomatin
MIPT 2020

Outline

- Unbalanced Binary Search Tree
  - BST invariant
  - BST find
  - BST insert/remove
  - Complexity
- Balanced Binary Search Trees
  - AVL tree
  - Red-Black tree
  - Treap (tree + heap)

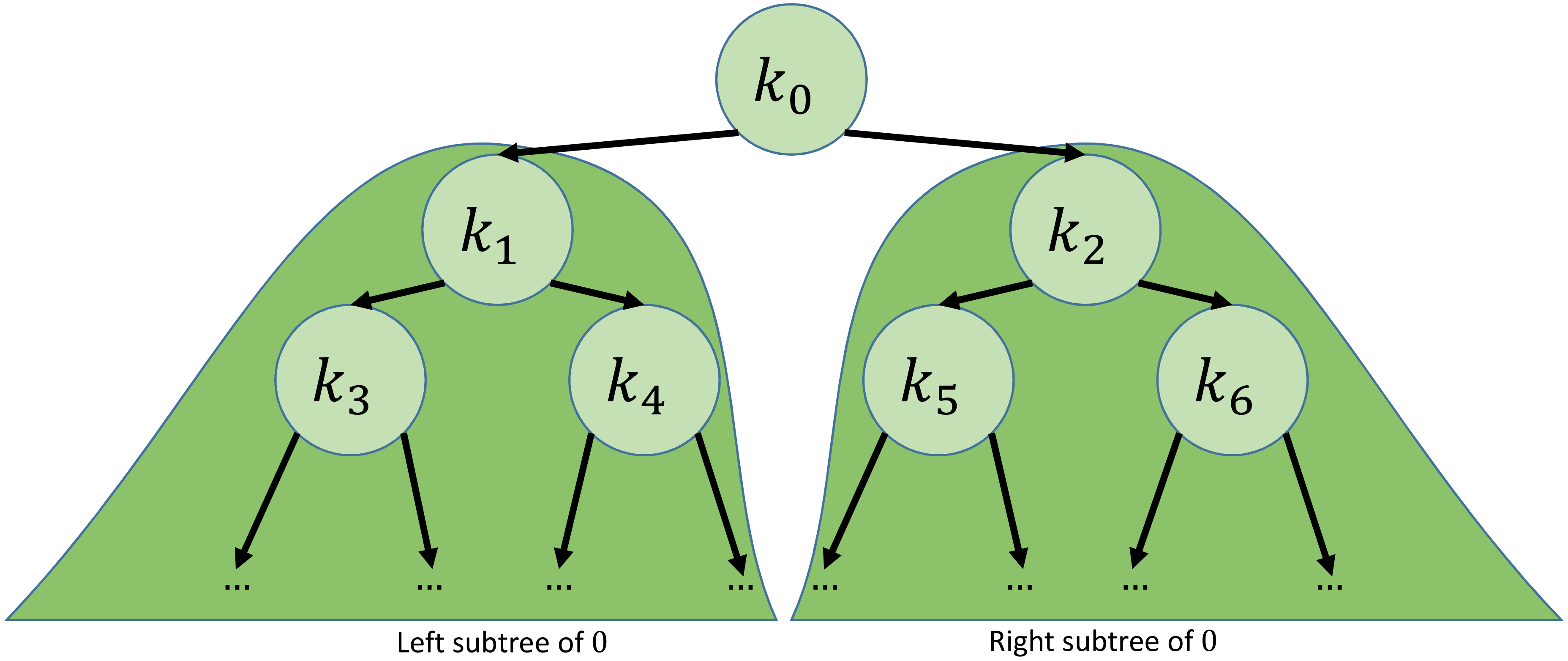
Unbalanced Binary Search Tree

- **BST invariant**
- BST find
- BST insert/remove
- Complexity

BST invariant

Invariant

BST is a binary tree: each node has 0-2 children.



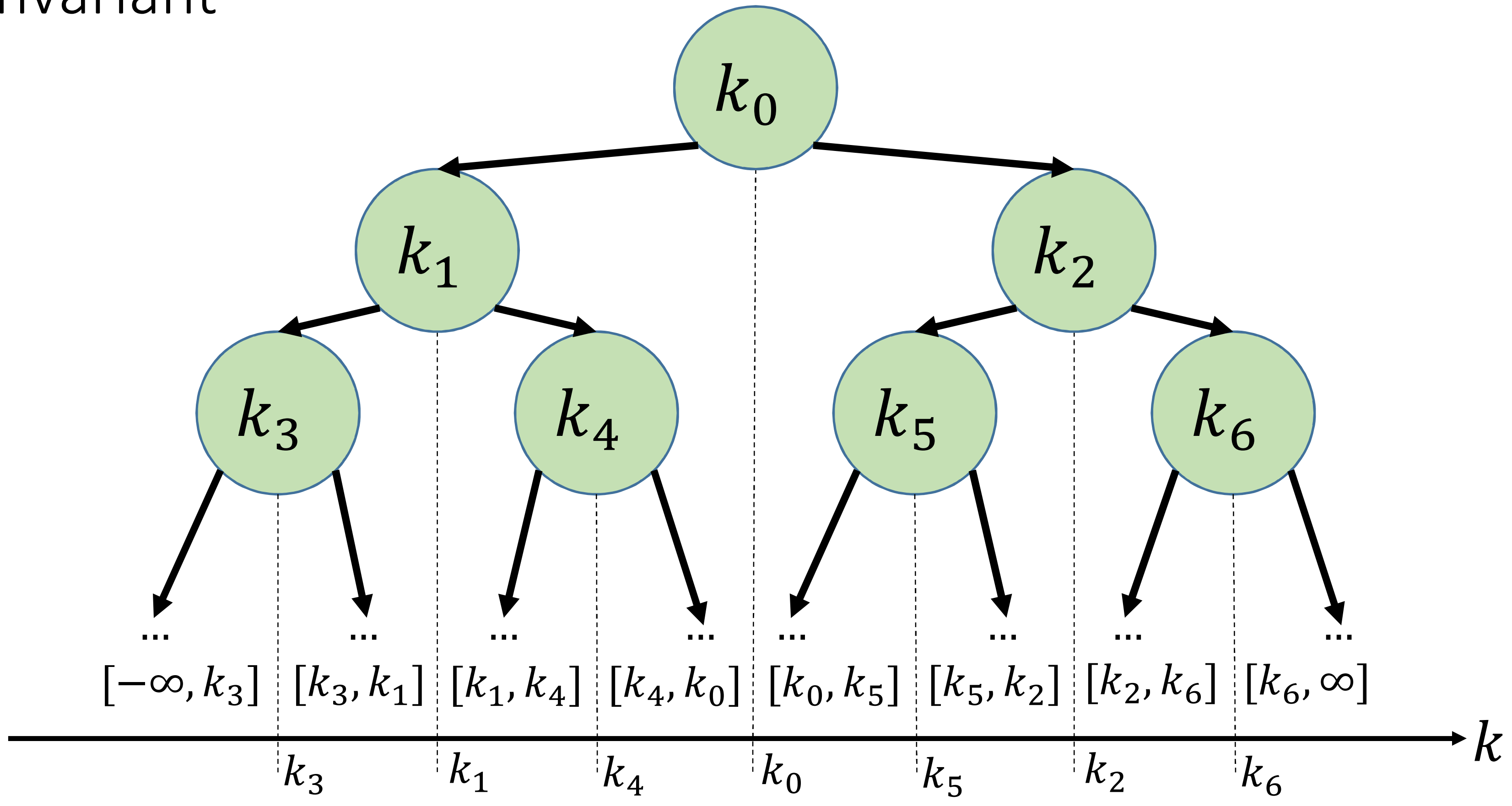
Invariant: if i and j are nodes of BST and j is in a subtree of i:

j is in left subtree of $i \Rightarrow k_j \leq k_i$

j is in right subtree of $i \Rightarrow k_i \leq k_j$

BST invariant

Invariant



Invariant: if i and j are nodes of BST and j is in a subtree of i:
j is in left subtree of $i \Rightarrow k_j \leq k_i$
j is in right subtree of $i \Rightarrow k_i \leq k_j$

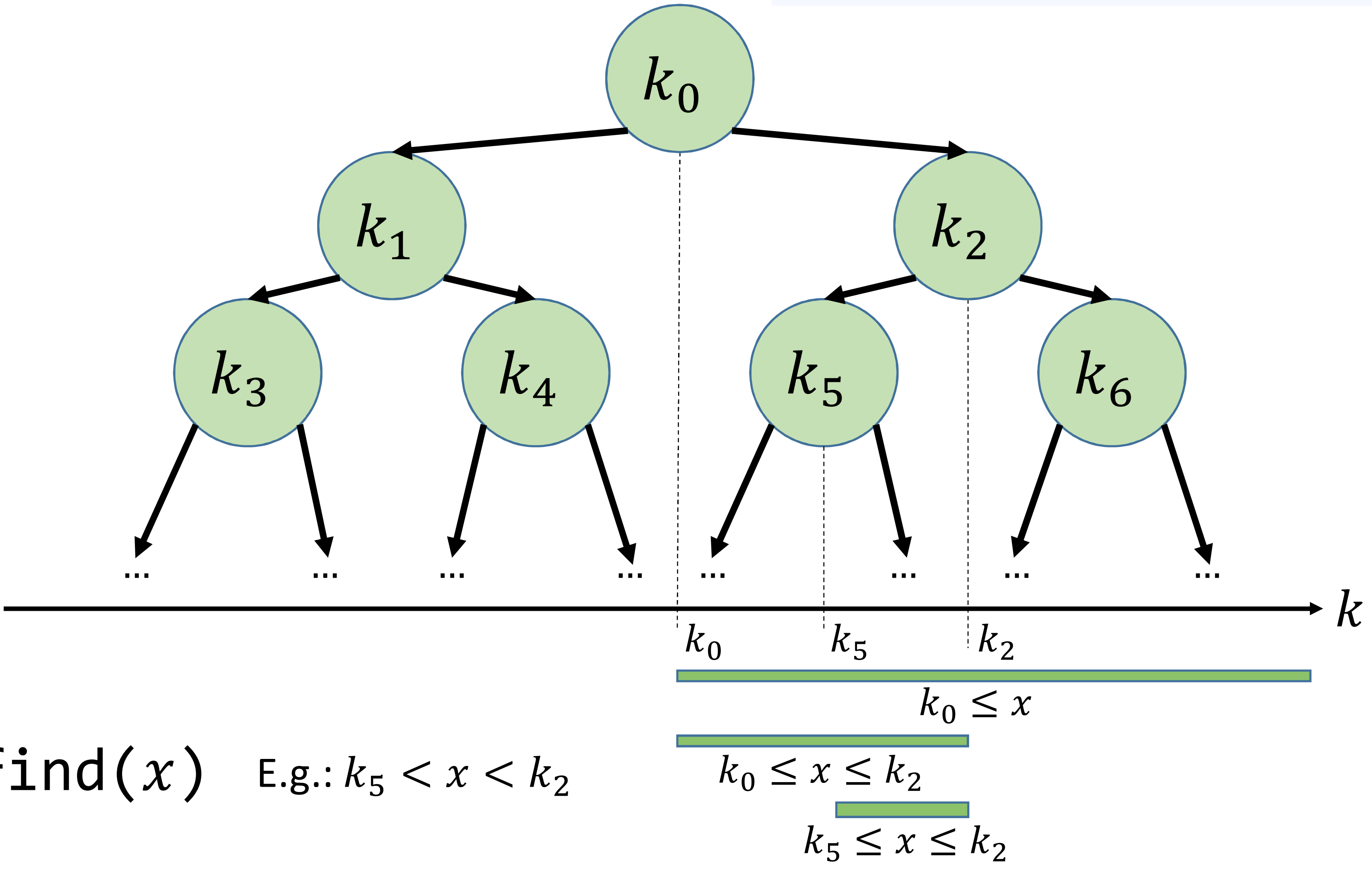
Unbalanced Binary Search Tree

- BST invariant
- **BST find**
- BST insert/remove
- Complexity

BST find

Find item with given k

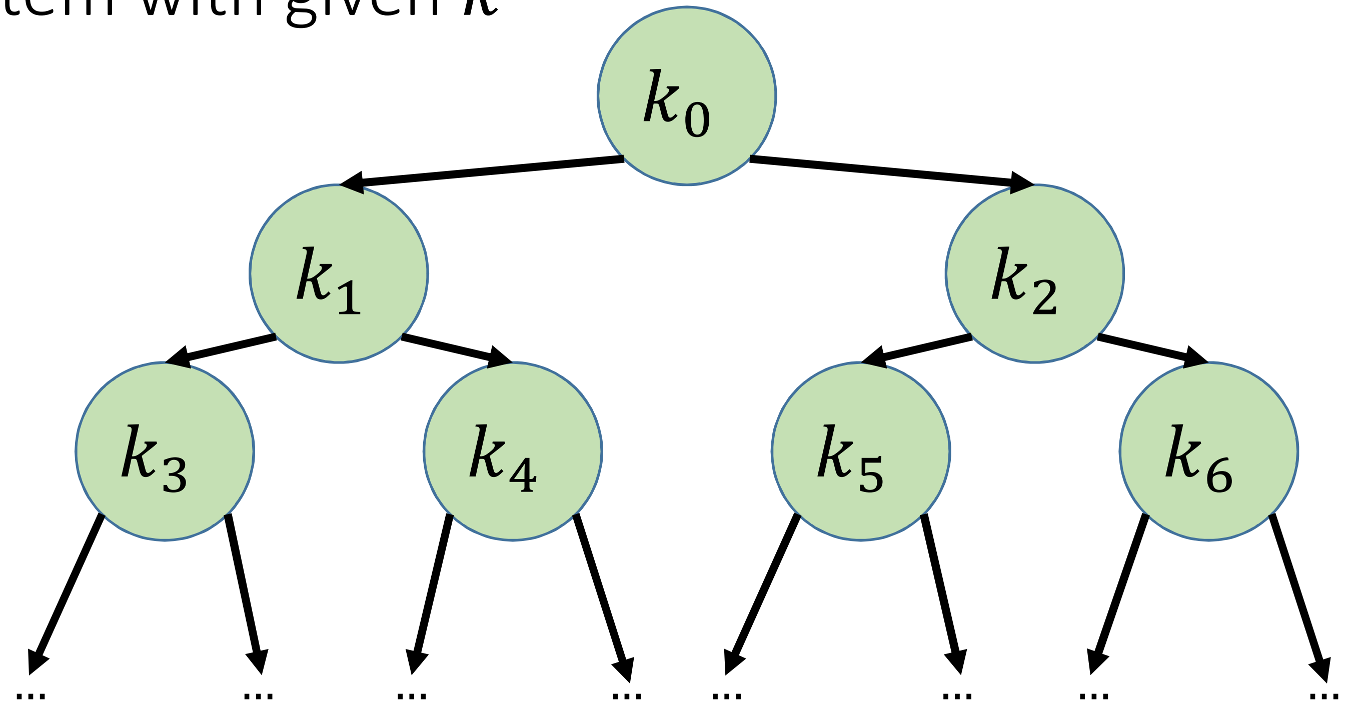
```
def find(x, i):
    if i is None or (k_i == x):
        return i
    if x < k_i:
        return find(x, left[i])
    else:
        return find(x, right[i])
```



find(x) E.g.: $k_5 < x < k_2$

BST invariant

Find item with given k



`find(x)` Let's suppose, x does not exist in the tree.

Search will stop in node i, such that one of the following is true:

$k_i < x$ and i doesn't have right child $\Rightarrow$ k_i is maximum number $\leq x$:

$$k_i = \max_j \{k_j : k_j \leq x\}$$

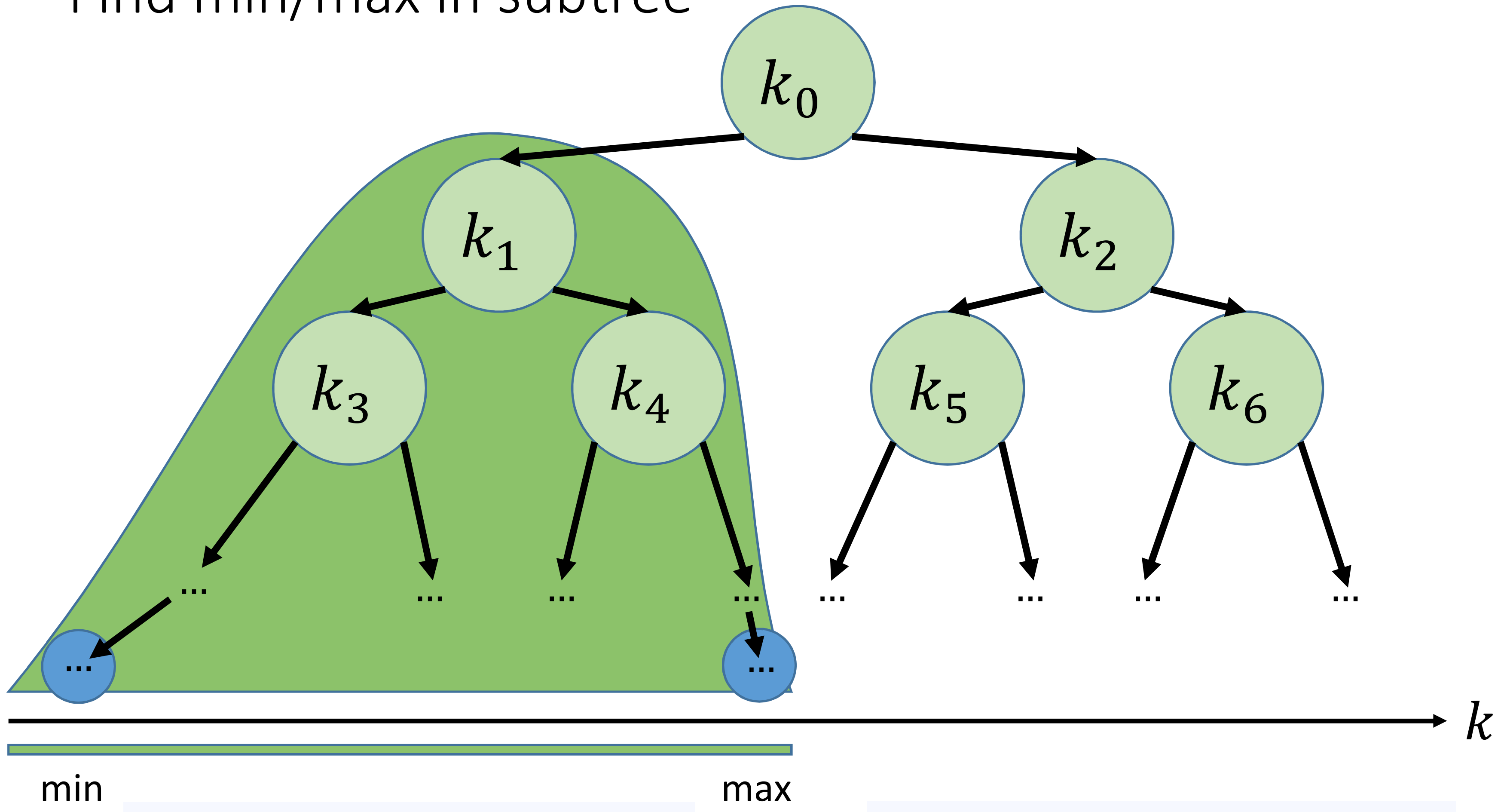
$k_i > x$ and i doesn't have left child $\Rightarrow$ k_i is minimum number $\geq x$:

$$k_i = \min_j \{k_j : k_j \geq x\}$$

It can be shown that both these nodes (max and min) are visited during search.

BST invariant

Find min/max in subtree



```python
def subtree_min(i):
    while left[i] is not None:
        i = left[i]
    return i
```

```python
def subtree_max(i):
    while right[i] is not None:
        i = right[i]
    return i
```

Unbalanced Binary Search Tree

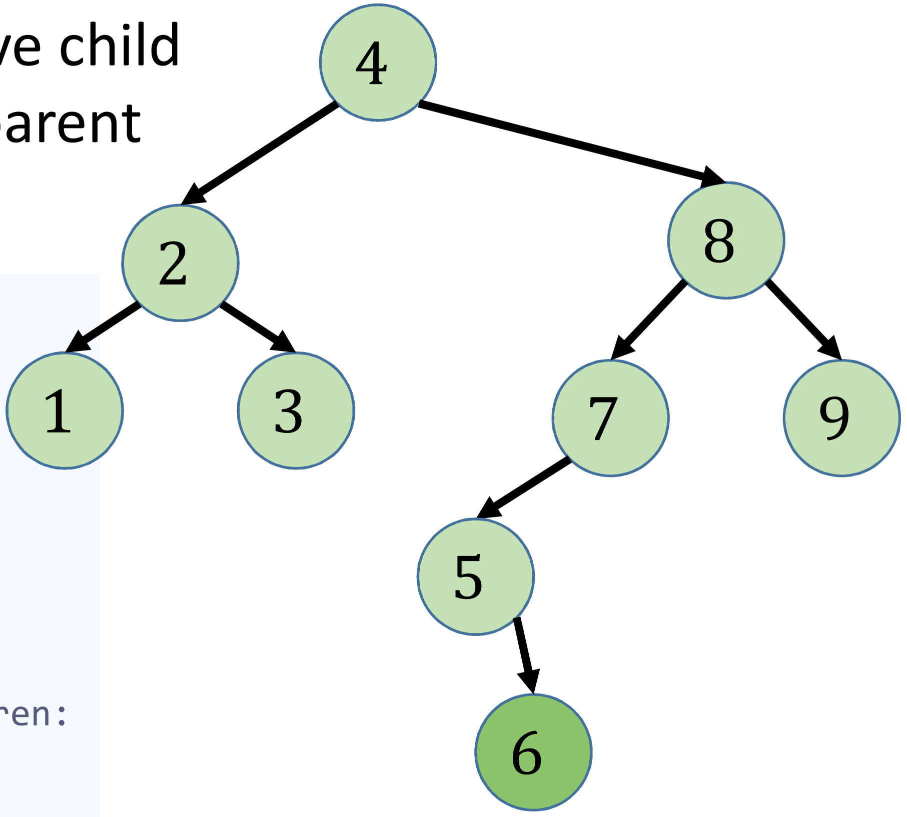
- BST invariant
- BST find
- **BST insert/remove**
- Complexity

BST invariant

Inserting item

Let’s just find a place to insert new node and reassign links:
Let’s repeat the same procedure as for find.
Final node which does not have child
in desired direction will be a parent
on new node.

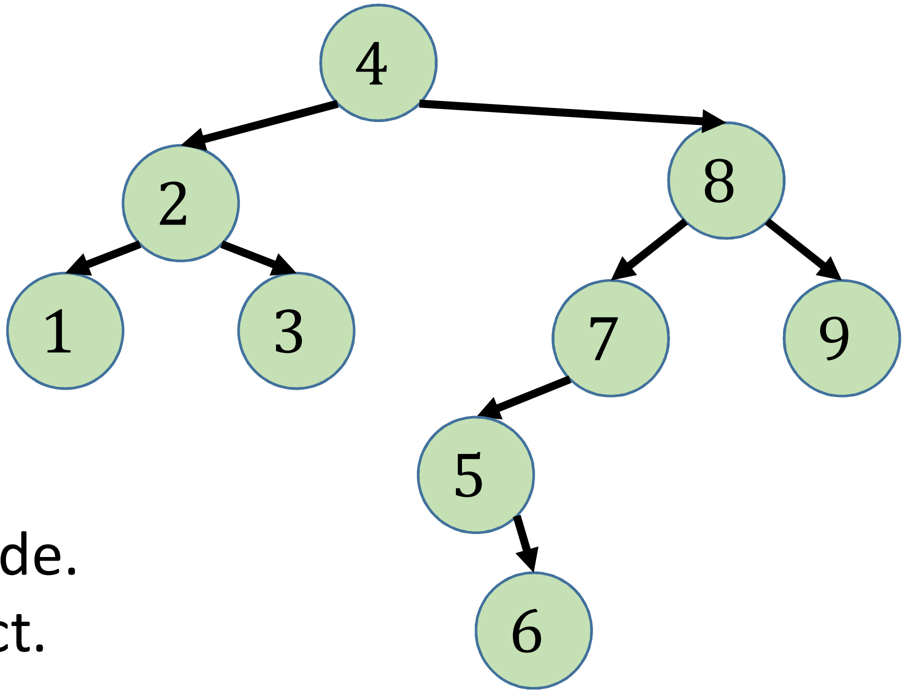
```
def insert(x, i):
    i_next = i
    while i_next is not None:
        i = i_next
    if x < $k_i$:
        i_next = left[i]
    else:
        i_next = right[i]

    # creating new node without children:
    if x < $k_i$:
        left[i] = new_node(x)
    else:
        right[i] = new_node(x)
```



BST invariant

Removing item

What if we need to remove some node from tree.
After removal we should restore invariant.
There are 3 possible cases to consider:
Case 1: Node to be removed has no children.

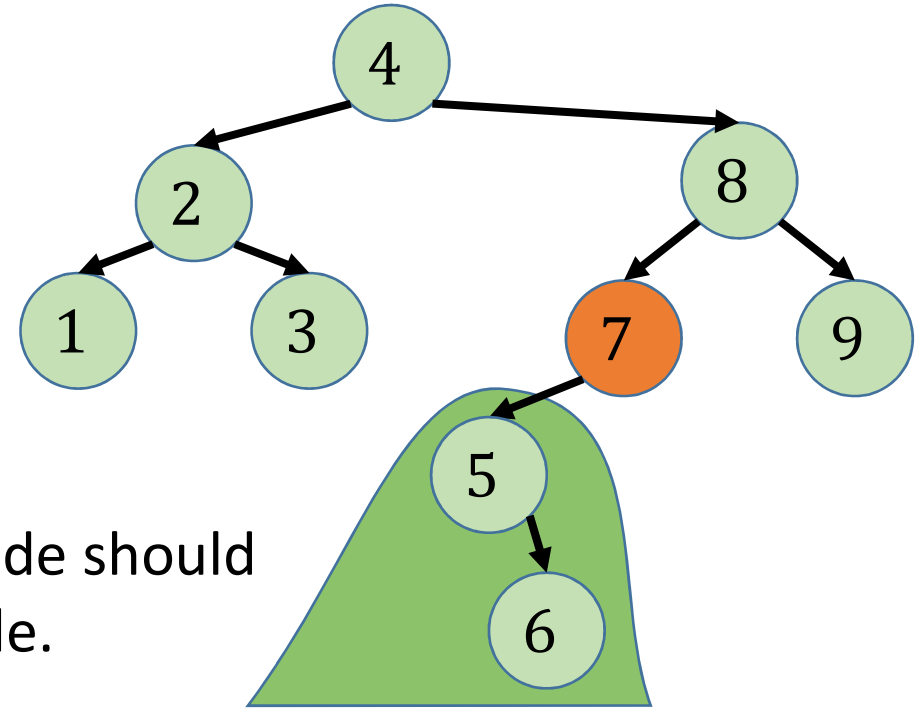


Actions:
We can just remove the node.
Invariant will remain correct.

BST invariant

Removing item

What if we need to remove some node from tree.
After removal we should restore invariant.
There are 3 possible cases to consider:
Case 2: Node to be removed has one child.

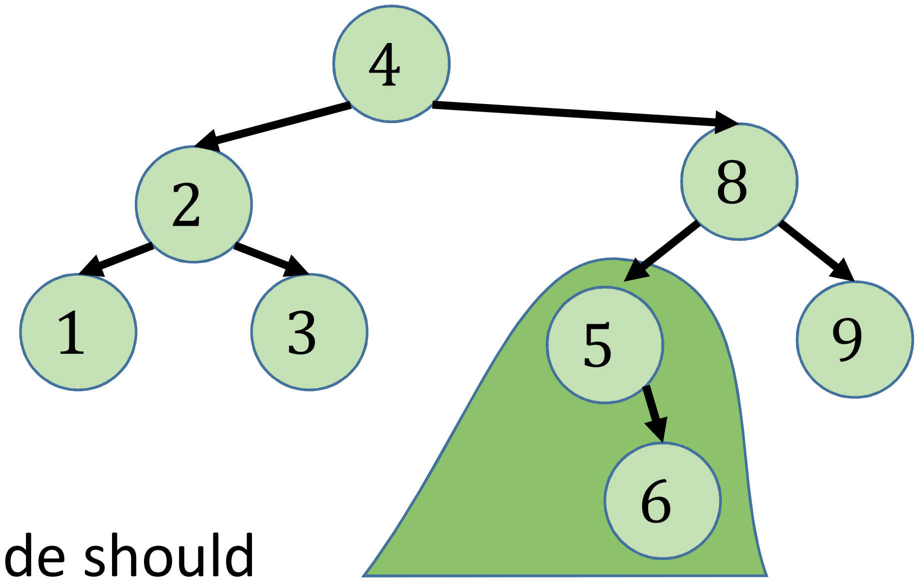


Actions:
Single child of removed node should take place of removed node.

BST invariant

Removing item

What if we need to remove some node from tree.
After removal we should restore invariant.
There are 3 possible cases to consider:
Case 2: Node to be removed has one child.



Actions:
Single child of removed node should take place of removed node.

BST invariant

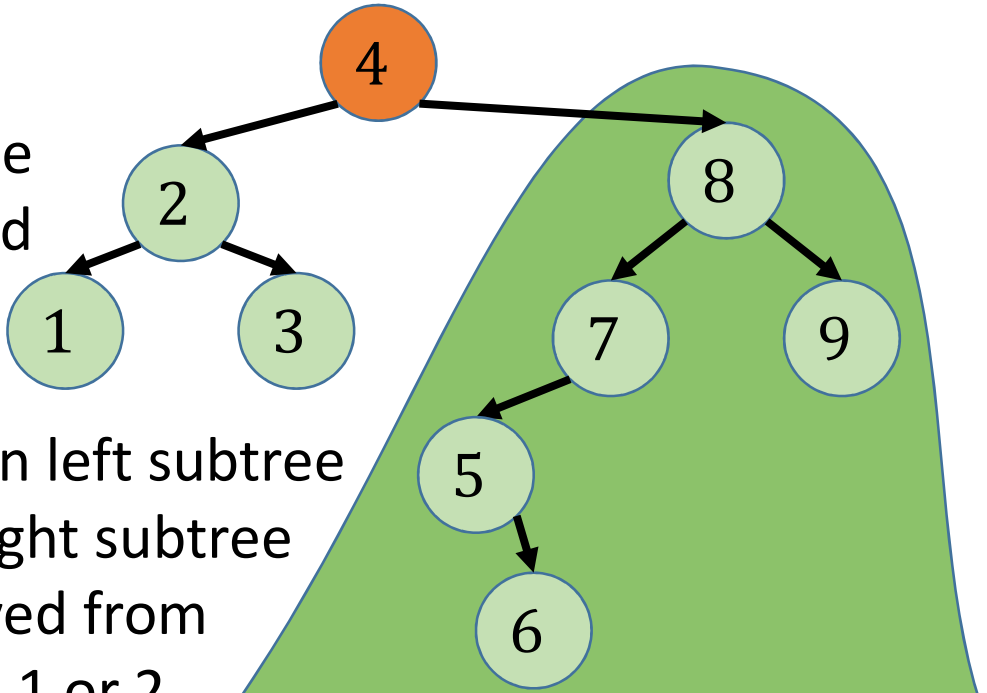
Removing item

What if we need to remove some node from tree.
After removal we should restore invariant.
There are 3 possible cases to consider:
Case 3: Node to be removed has two children.

Actions:
We should find a node to be inserted instead of removed one. We can find a node which will satisfy invariant:

- Node with maximum k in left subtree
- Or with minimum k in right subtree

This node should be removed from its place according to cases 1 or 2.



BST invariant

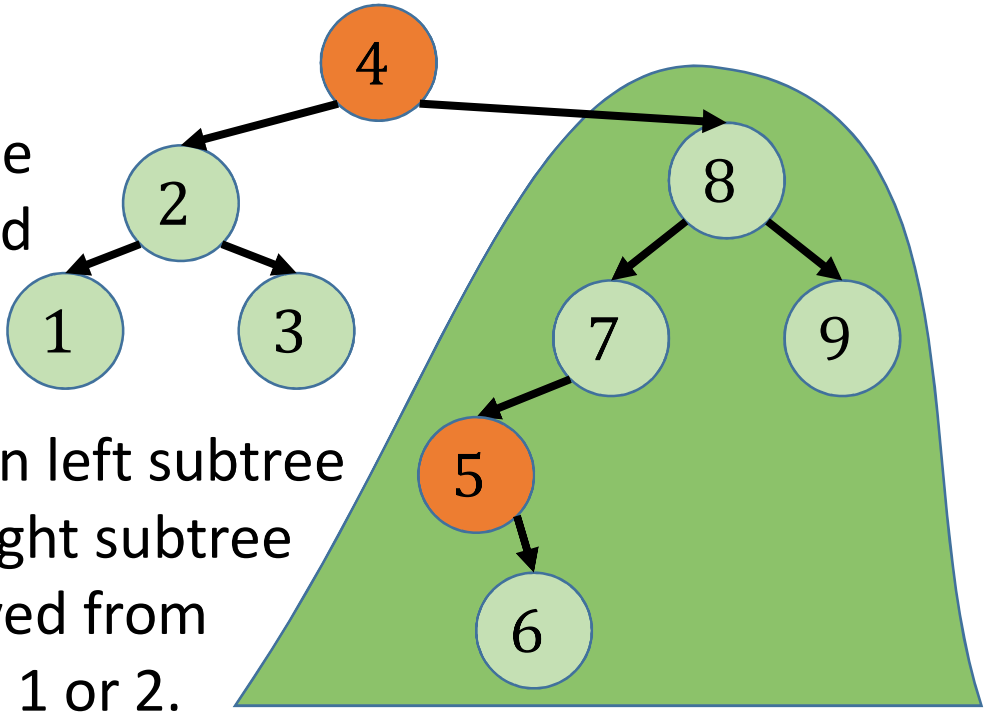
Removing item

What if we need to remove some node from tree.
After removal we should restore invariant.
There are 3 possible cases to consider:
Case 3: Node to be removed has two children.

Actions:
We should find a node to be inserted instead of removed one. We can find a node which will satisfy invariant:

- Node with maximum k in left subtree
- Or with minimum k in right subtree

This node should be removed from its place according to cases 1 or 2.



BST invariant

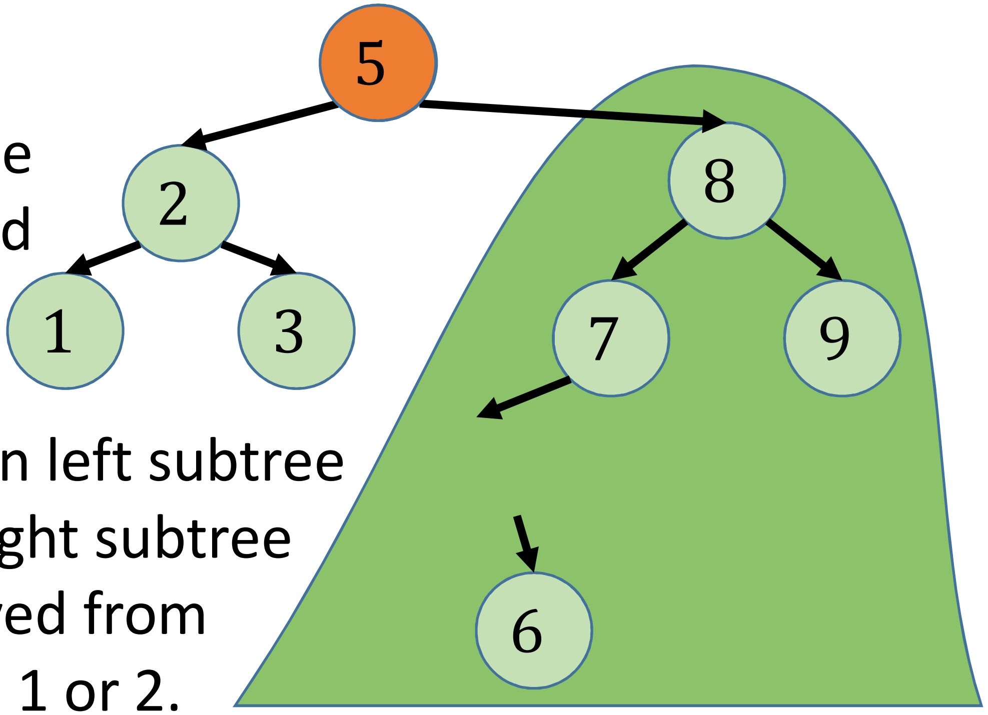
Removing item

What if we need to remove some node from tree.
After removal we should restore invariant.
There are 3 possible cases to consider:
Case 3: Node to be removed has two children.

Actions:
We should find a node to be inserted instead of removed one. We can find a node which will satisfy invariant:

- Node with maximum k in left subtree
- Or with minimum k in right subtree

This node should be removed from its place according to cases 1 or 2.



BST invariant

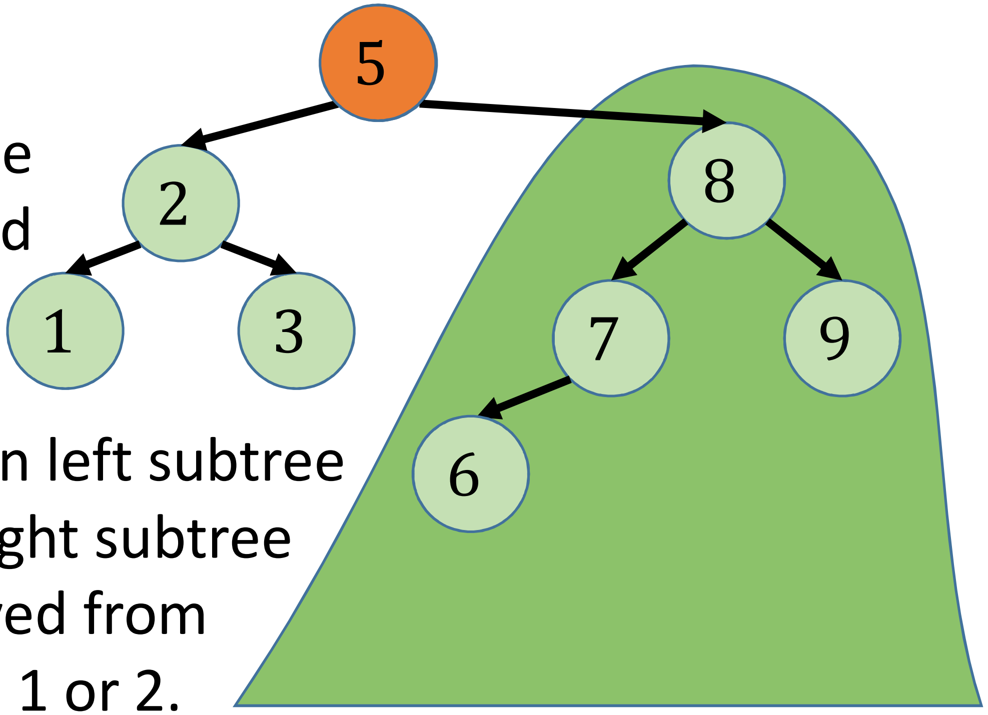
Removing item

What if we need to remove some node from tree.
After removal we should restore invariant.
There are 3 possible cases to consider:
Case 3: Node to be removed has two children.

Actions:
We should find a node to be inserted instead of removed one. We can find a node which will satisfy invariant:

- Node with maximum k in left subtree
- Or with minimum k in right subtree

This node should be removed from its place according to cases 1 or 2.



Unbalanced Binary Search Tree

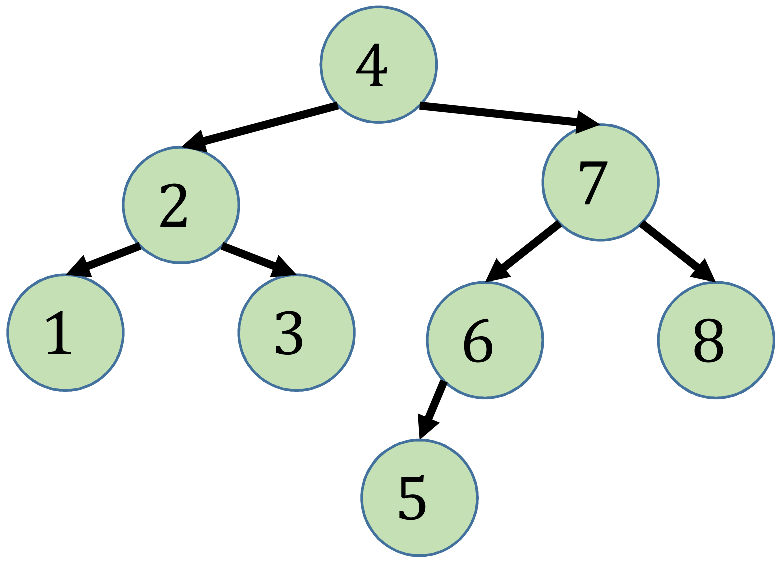
- BST invariant
- BST find
- BST insert/remove
- **Complexity**

BST invariant

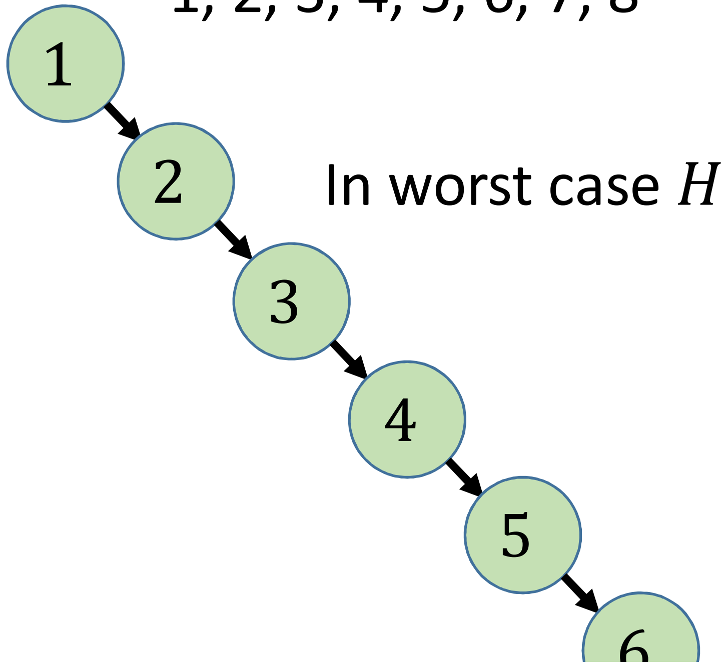
Complexity

All operations: find/insert/remove in worst case go from root of the tree to leaf.
This means, $O(H)$ operations.
H depends on the order in which elements are inserted.

4, 2, 3, 7, 1, 8, 6, 5



1, 2, 3, 4, 5, 6, 7, 8



In worst case $H = N$

BST invariant

Complexity

If order in which items are added is not important, we can add them in random order.
That will give good expectation of height:

$$\mathrm{E}(H) = O(\log N)$$

And as a result, expectation of operations will be $O(\log N)$.

When order may be not important? E.g. in static problems, when set of items is not changed.
In most cases BST is supposed to be dynamic and order in which items are added is supposed to be important.

To provide $O(\log \mathrm{N})$ complexity for find/insert/remove operations in dynamic case, tree balancing is used.

Balanced Binary Search Tree

- **AVL tree**
- Red-Black tree
- Treap (tree + heap)

AVL tree

Invariant

One of the easiest ways to balance BSTs is AVL tree approach.
AVL: Adelson-Velsky, Landis

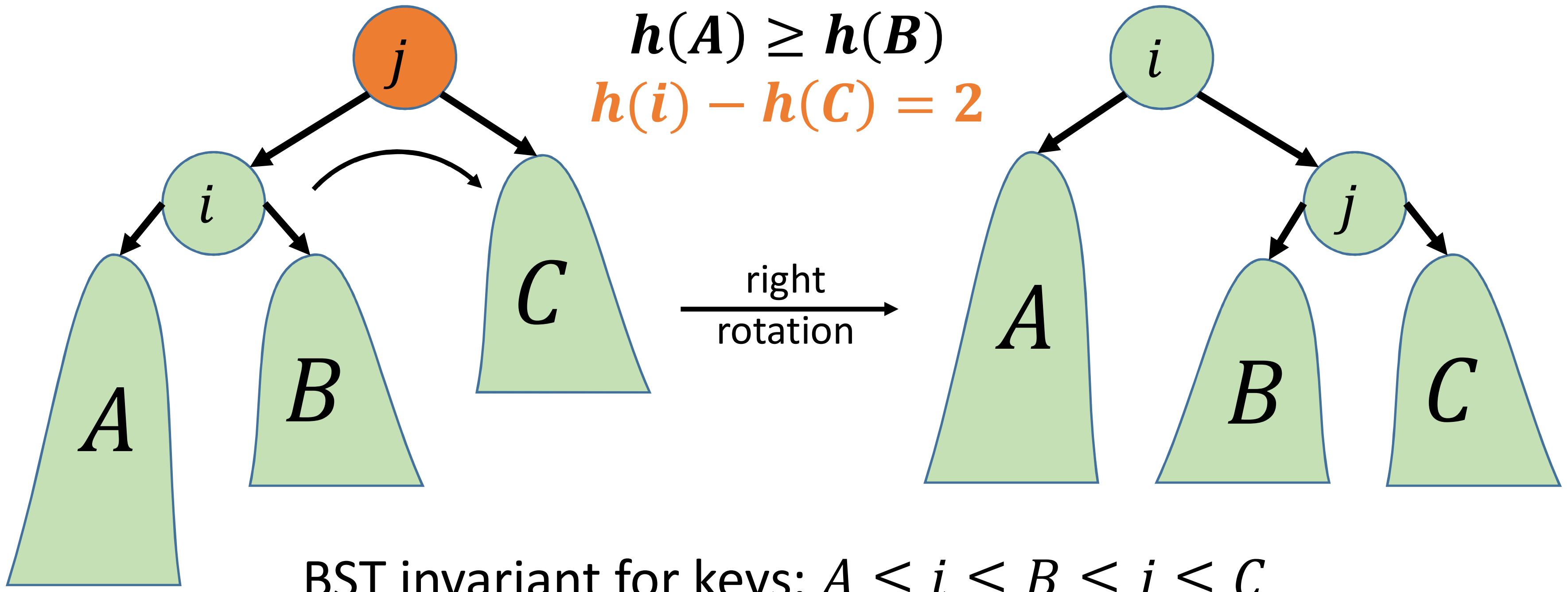
Additional balancing invariant:
$h[i]$ – height of subtree of node i

$$\forall i\colon \left|h\left[left[i]\right] - h\left[right[i]\right]\right| \leq 1$$

AVL tree

Simple rotation

Rotations allow to manage height of tree by reordering nodes in the way which will satisfy invariant.



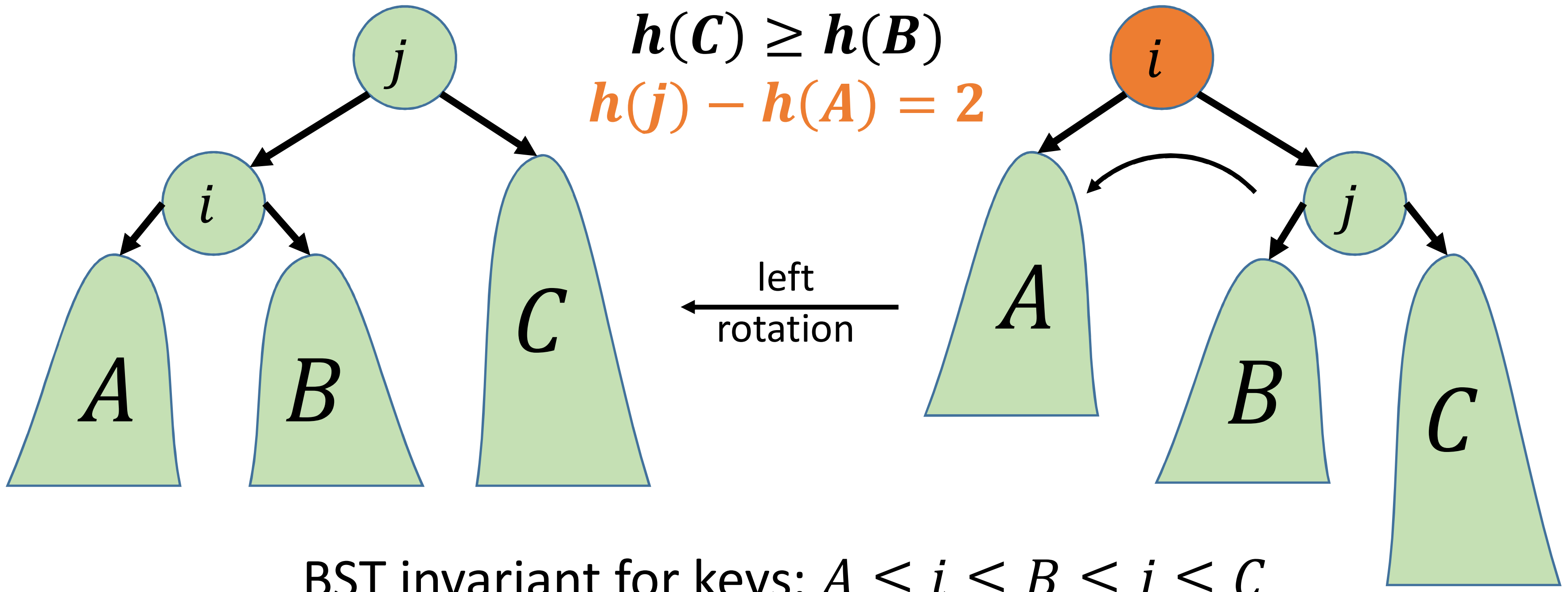
BST invariant for keys: $A \leq i \leq B \leq j \leq C$

$$\begin{aligned} h(i) &= \max(h(A), h(B)) \\ h(j) &= \max(h(i), h(C)) \end{aligned} \longrightarrow \begin{aligned} h(j) &= \max(h(B), h(C)) \\ h(i) &= \max(h(A), h(j)) \end{aligned}$$

AVL tree

Simple rotation

Rotations allow to manage height of tree by reordering nodes in the way which will satisfy invariant.

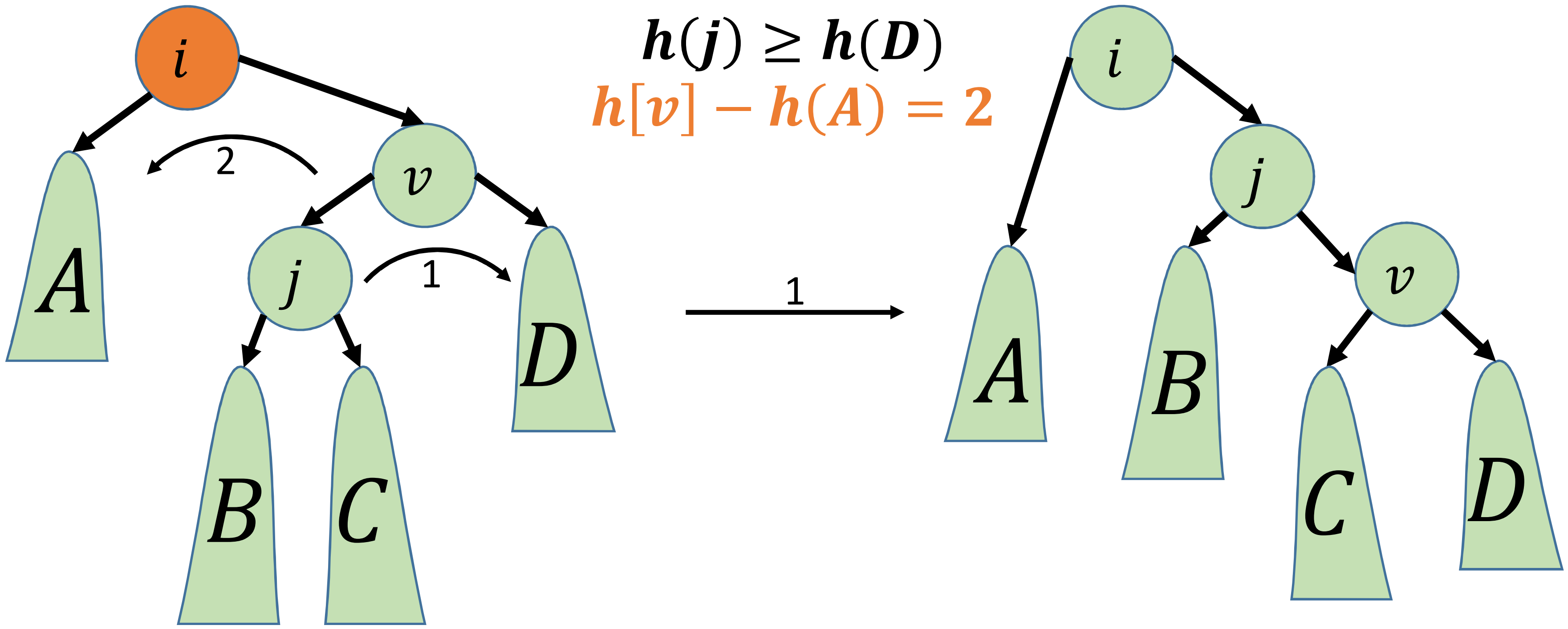


BST invariant for keys: $A \leq i \leq B \leq j \leq C$

$$h(i) = \max(h(A), h(B)) \qquad\qquad h(j) = \max(h(B), h(C))$$
$$h(j) = \max(h(i), h(C)) \longleftarrow h(i) = \max(h(A), h(j))$$

AVL tree

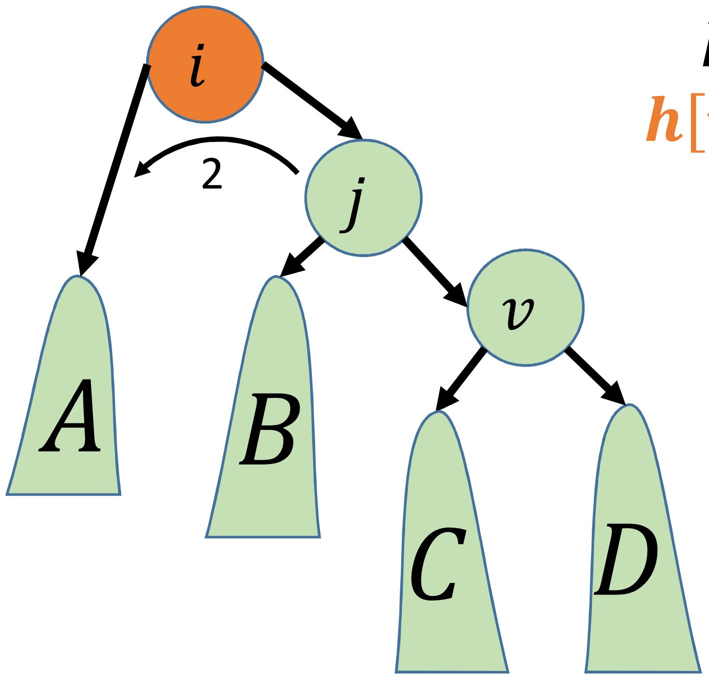
Double rotation



BST invariant for keys: $A \leq i \leq B \leq j \leq C \leq v \leq D$

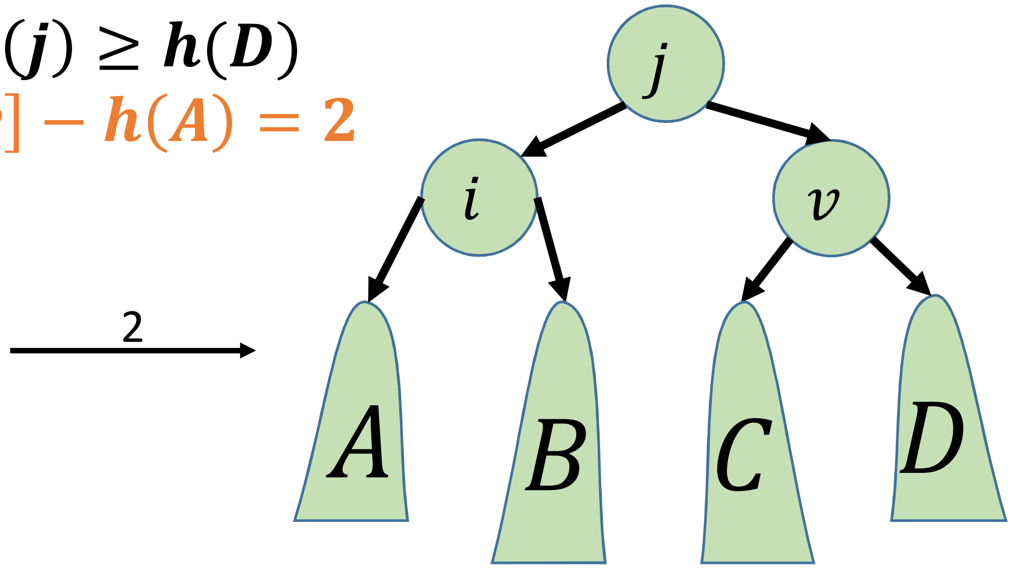
AVL tree

Double rotation



$\boldsymbol{h(j) \geq h(D)}$

$\boldsymbol{h[v] - h(A) = 2}$



BST invariant for keys: $A \leq i \leq B \leq j \leq C \leq v \leq D$

AVL tree

Complexity

AVL tree invariant guarantees estimation on height of tree:

$$h \leq \lfloor 1.45 \log_2(N+2) \rfloor = O(\log N)$$

All rotations are made in $O(1)$, so final complexity of BST tree with AVL balancing:

$O(\log N)$ for find/insert/remove.

Balanced Binary Search Tree

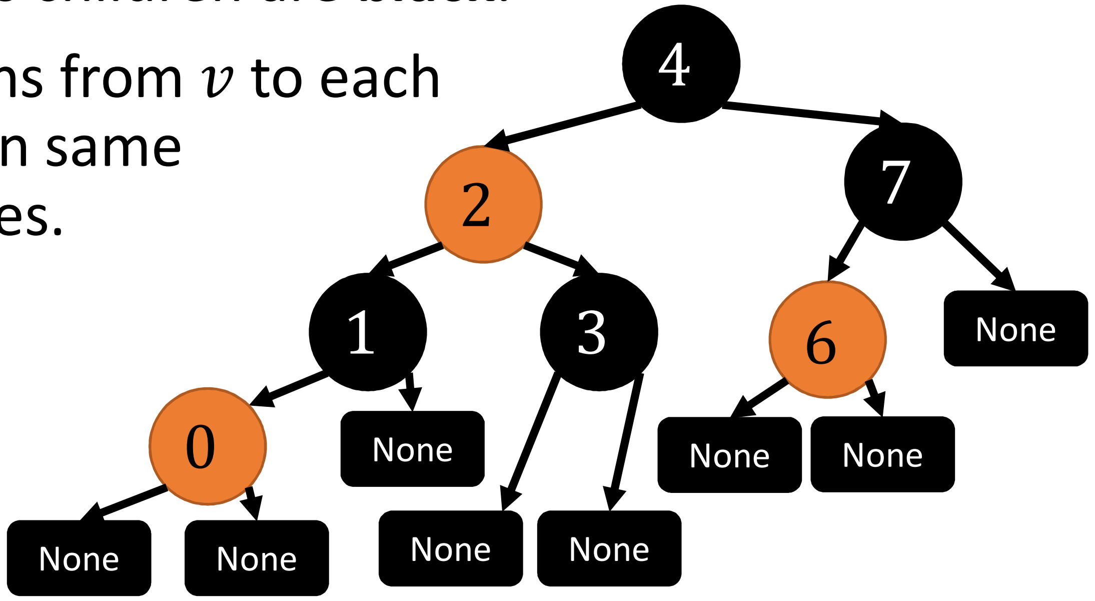
- AVL tree
- **Red-Black tree**
- Treap (tree + heap)

Red-black tree

Invariant

Balancing invariant of red-black tree:

1. Each node is either **black** or **red**.
2. Root of the tree is **black**
3. Each leaf of the tree (None) is **black**
4. If node is **red**, both its children are **black**.
5. For each node v, paths from v to each leaf in subtree contain same number of **black** nodes.



Red-black tree

Height estimation

Let's denote number of black nodes from v to leafs in its subtree (v is not counted) as $bh(v)$. $n(v)$ – number of non-fictive nodes in subtree of v.

Theorem 1: Subtree of each node v in red-black tree has $\geq 2^{bh(v)} - 1$ nodes. $\forall v: n(v) \geq 2^{bh(v)} - 1$.

Proof: induction by $bh(v)$.

Basis: $bh(v) = 0 \Rightarrow v$ is fictive node (None) $\Rightarrow n(v) = 0$.

$$n(v) = 0 \qquad \geq \qquad 2^0 - 1 = 0$$

Step: For children of v: v_1, v_2 there are two cases:

$$bh(v_1), bh(v_2) = \left[\begin{array}{c} bh(v) \\ bh(v) - 1 \end{array}\right. \geq bh(v) - 1$$

According to induction hypothesis, this means that:

$$n(v_1), n(v_2) \geq 2^{bh(v)-1} - 1.$$

$$n(v) = n(v_1) + n(v_2) + 1 \geq 2\left(2^{bh(v)-1} - 1\right) + 1 = 2^{bh(v)} - 1 \ \blacksquare$$

Red-black tree

Height estimation

Let’s denote root of tree as r.

Theorem 2: Height of red-black tree with $n(r) = N$ non-fictive nodes is:

$$h \leq 2\log_2(N+1)$$

Proof:

According to invariant rule 4, at least half of nodes on path from r to leafs are black. So, $bh(r) \geq h/2$.

According to theorem 1: $n(r) \geq 2^{\frac{h}{2}} - 1 \Rightarrow h \leq 2\log_2(n(r)+1)$ ■

This means,

$$h \leq 2\log_2(N+1) = O(\log N)$$

Balanced Binary Search Tree

- AVL tree
- Red-Black tree
- **Treap (tree + heap)**

Treap

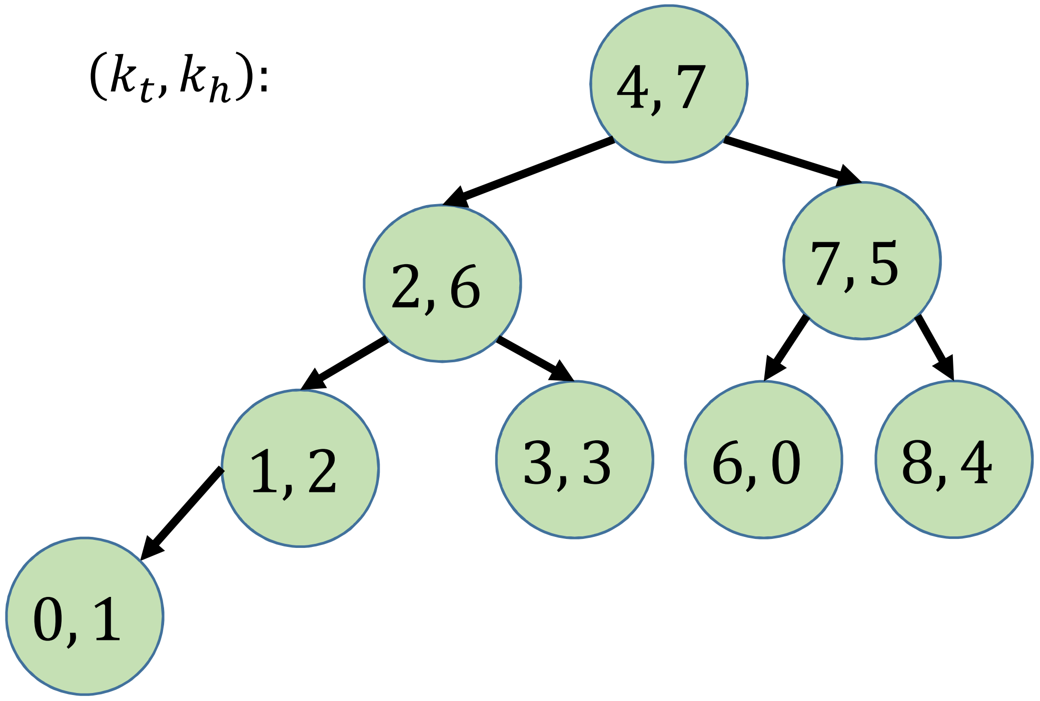
Idea

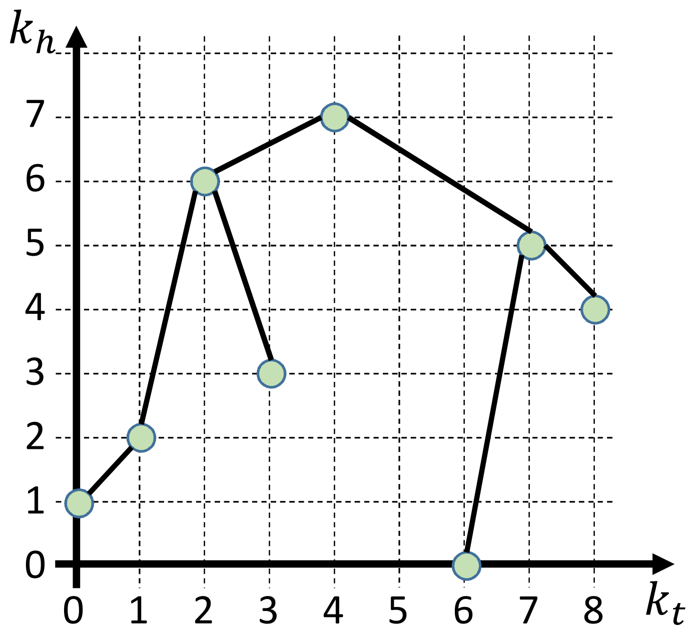
Treap is a combination of BST and binary heap (tree+heap=treap).

Balancing invariant:

Each node has two key values: k_t, k_h and it should satisfy BST invariant for k_t and binary heap invariant for k_h.

(k_t, k_h):





Treap

Idea

There are different ways to use and assign k_h value.

One of the easiest application is BST balancing by assigning random k_h.

If k_h is random, expectation of height is logarithmic:

$$\mathrm{E}(H) = O(\log N)$$

And that, in turn, makes expectation of complexity of find/insert/remove operations on the tree equal to $O(\log N)$.

Conclusion

Thank you for watching!