Introduction to Algorithms

Mong-Jen Kao (高孟駿)

Tuesday 10:10 – 12:00

Thursday 15:30 – 16:20

Data Structures

Particular ways of storing data *to support special operations*.

Search Trees

with Self-Balancing Guarantees

BSTs that have an $O(\log n)$ height guarantee.

BSTs with Self-Balancing Mechanisms

- In this lecture, we are going to see two types of BSTs with an $O(\log n)$-height guarantee.
  - ***Treap***

    – a data structure that has both *the BST property* and *the heap property* and has an **expected $\boldsymbol{O(\log n)}$ height**.
  - ***Red-Black Tree***

    – a data structure that has a *counting-based self-balancing mechanism* and a **worst-case $\boldsymbol{O(\log n)}$ height**.

Treap

Treap

- A treap is a binary tree T where
  - Each node $v \in T$ is associated
    with a key $\mathrm{val}(v)$ and a ***<u>randomly-assigned</u>*** priority $\mathrm{pri}(v)$.
  - For any $u, v \in T$,
    the probability that $\mathrm{pri}(u) = \mathrm{pri}(v)$ is small enough and ***<u>negligible</u>***.
  - It has the ***<u>BST property</u>*** with respect to $\mathbf{val(.)}$ and
    the ***<u>max-heap property</u>*** with respect to (random) $\mathbf{pri(.)}$.

Treap

- Let $A = \{ a_1, a_2, \ldots, a_n \}$ be a set of numbers and $p_1, \ldots p_n$ be ***randomly assigned priorities*** such that $\boldsymbol{p_i \neq p_j}$ **for** $\boldsymbol{i \neq j}$.
  - Then, the treap T_A for A w.r.t. $p_1, \ldots, p_n$ is uniquely defined.
  - Most importantly,
    we will (later) see that, ***the expected height of*** $\boldsymbol{T_A}$ is $\boldsymbol{O(\log n)}$.

Operations Supported by Treaps

- Treap supports all the standard operations for BSTs in **expected $O(\log n)$ time**.
  - Search, Predecessor, Successor, Minimum, Maximum, Insert, and Delete.

Unique Operations Supported by Treaps

- In addition, treap supports two unique operations in expected $O(\log n)$ time that other BSTs don't.
  - ***Merge***$(\boldsymbol{T_1}, \boldsymbol{T_2}, \boldsymbol{x})$ **–**

    Given T_1, T_2 with $u \le x$ for all $u \in T_1$ and $v \ge x$ for all $v \in T_2$, produce a treap $T = T_1 \cup T_2$.

  - ***Split***$(\boldsymbol{T}, \boldsymbol{x})$ **–**

    to split T into two treaps T_1 and T_2 such that $u \le x$ for all $u \in T_1$ and $v \ge x$ for all $v \in T_2$.

Unique Operations Supported by Treaps

- In addition, treap supports two unique operations in expected $O(\log n)$ time that other BSTs don't.
  - ***Merge***$(\boldsymbol{T_1}, \boldsymbol{T_2}, \boldsymbol{x})$
  - ***Split***$(\boldsymbol{T}, \boldsymbol{x})$

- In other words, treaps allow us to
  - ***Concatenate*** *two **ordered** sorted lists* or
  - ***Split*** *a sorted list into two ordered sorted lists*

  while *maintaining the searchable property* in ***expected*** $\mathbf{O(\log n)}$ ***time***.

Treap Operations

With existing operations for Max-Heap and BSTs,
the operations for treap can be *implemented easily*.

We describe the operations for treaps based on the operations
we have seen so far for Max-Heap and BSTs.

Insertion

- To insert a node v into a treap T,
  we proceed as follows.
  - Use Tree-Insert$(\mathrm{root}[T], v)$ to insert v **as a leaf of T**.
  - Use Increase-Key$\left(\mathrm{root}[T], v, \mathrm{pri}(v)\right)$ to restore the max-heap property for T.
    - However, we use **tree rotations** *instead of swap operation*.

This ensures the BST property.

- After this, both ***max-heap property*** and ***BST property*** are maintained.

Deletion

- To delete a given node v from a treap T, we proceed as follows.
  - Change $\mathrm{pri}(v)$ to be $-\infty$ and perform Max-Heapify(T, v) to sink the vertex v to the bottom of the treap T ***as a leaf***.
    - However, we use tree rotations instead of swap operation.
  - Use Tree-Delete$(\mathrm{root}[T], v)$ to delete v from T or **just delete v**.

This ensures the heap property.

- After this, both ***max-heap property*** and ***BST property*** are still maintained.

This does not alter the BST property.

Building a Treap Offline

- When the elements $a_1, \dots, a_n$ are given in sorted order, the treap can be built in $O(n)$ time.
  - First, we build a balanced BST T for $a_1, \dots, a_n$ in $O(n)$ time.
  - Then, we use Build-Max-Heap(T) to establish the max-heap property in $O(n)$ time.
    - Similarly, we use tree rotations instead of swap operation.

Merging Two Ordered Treaps

- Given two treaps T_1 and T_2 such that $u \leq x \leq v$ for all $u \in T_1$, $v \in T_2$ and some (unknown) x, we can merge T_1 and T_2 as follows.
  - Let $y \leftarrow$Tree-Max(T_1) and $z \leftarrow$Tree-Min(T_2).
    Report fail if $y > z$.
  - Create a new tree T with a new root node v, where T_1 and T_2 are the left- and the right- subtree of v.
  - Call Treap-Delete(T, v).

Splitting a Treap w.r.t. a Given Value

- Given a treap T and an element x,
  we can split T into T_1 and T_2 such that $u \leq x \leq v$ for all $u \in T_1$, $v \in T_2$.
  - Create a new node x with $\mathrm{pri}(x) \coloneqq \infty$.
  - Call Treap-Insert(T, x).
  - Let T_1 and T_2 be the left- and the right- subtrees of the node x.
  - Delete x and return T_1 and T_2.

Analysis of Treap Operations

It suffices to analyze the expected height of the treaps.

All the nontrivial treap operations take time $O(h)$.

Expected Height of a Treap

- In the following we analyze the ***average-case performance*** / ***expected height*** of a treap.

- Let $a_1 < a_2 < \cdots < a_n$ be the elements in the treap.
  - We also assume that the $\mathbf{pri}(\boldsymbol{a_i}) \neq \mathbf{pri}(\boldsymbol{a_j})$ **for all** $\boldsymbol{i} \neq \boldsymbol{j}$.

- We will show that
  the expected height of any a_i in the treap T is $O(\log n)$.

Expected Height of a Treap

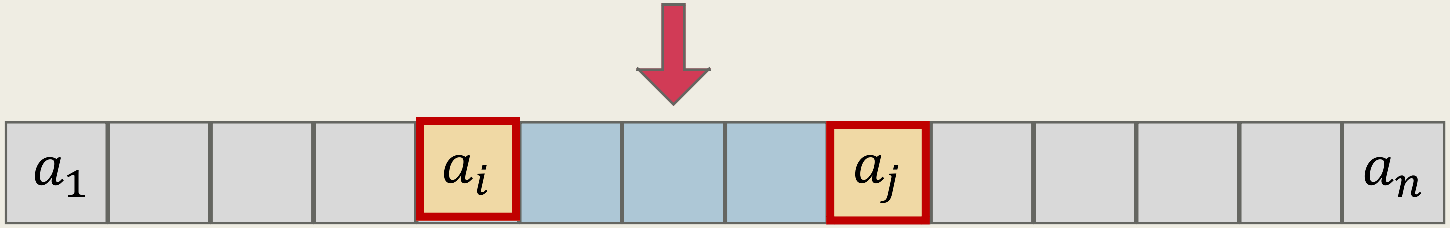
- Let $a_1 < a_2 < \cdots < a_n$ be the elements in the treap.
  - We also assume that the $\mathrm{pri}(a_i) \neq \mathrm{pri}(a_j)$ for all $i \neq j$.

- We will show that

  the expected height of any a_i in the treap T is $O(\log n)$.

  - The height of a node in the tree is equal to the ***number of ancestors*** of it.
  - Hence, we count the <u>***expected number of ancestors***</u> of a_i.

When can a_j become an ancestor of a_i ?



- Let $X_{i,j}$ be the indicator variable for the event that "a_j is an ancestor of a_i".
- $E_{i,j}$ is determined *completely* by the element a_k **between $a_i, \ldots, a_j$** with ***the highest priority***.
  - $X_{i,j} = 1$ if and only if a_k is equal to a_j.

When can a_j become an ancestor of a_i ?

- When the priorities of the elements are ***randomly drawn*** and ***distinct***, we have
$$\Pr[X_{i,j}] = \frac{1}{|j-i|+1} .$$

- The expected number of ancestors of a_i is
$$\sum_{j \neq i} \frac{1}{|j-i|+1} \leq 2 \cdot H_n = O(\log n) .$$

Red-Black Tree

A self-balancing BST with a worst-case $O(\log n)$ height guarantee.

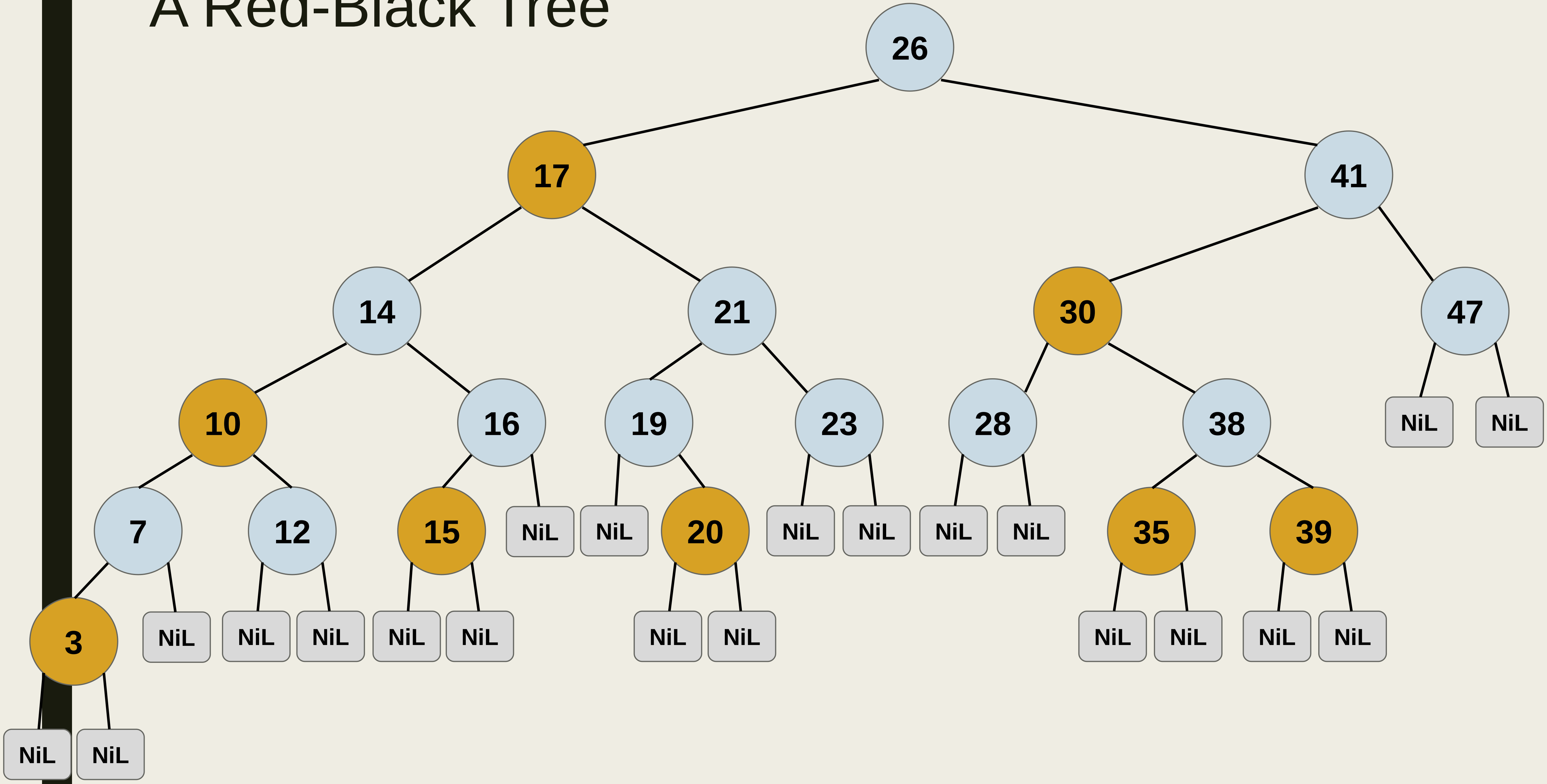
Red-Black Tree (RB-Tree)

- Red-Black Tree is a ***binary search tree*** imposed *with* ***extra constraints on its structure*** to achieve a worst-case $O(\log n)$ height guarantee.

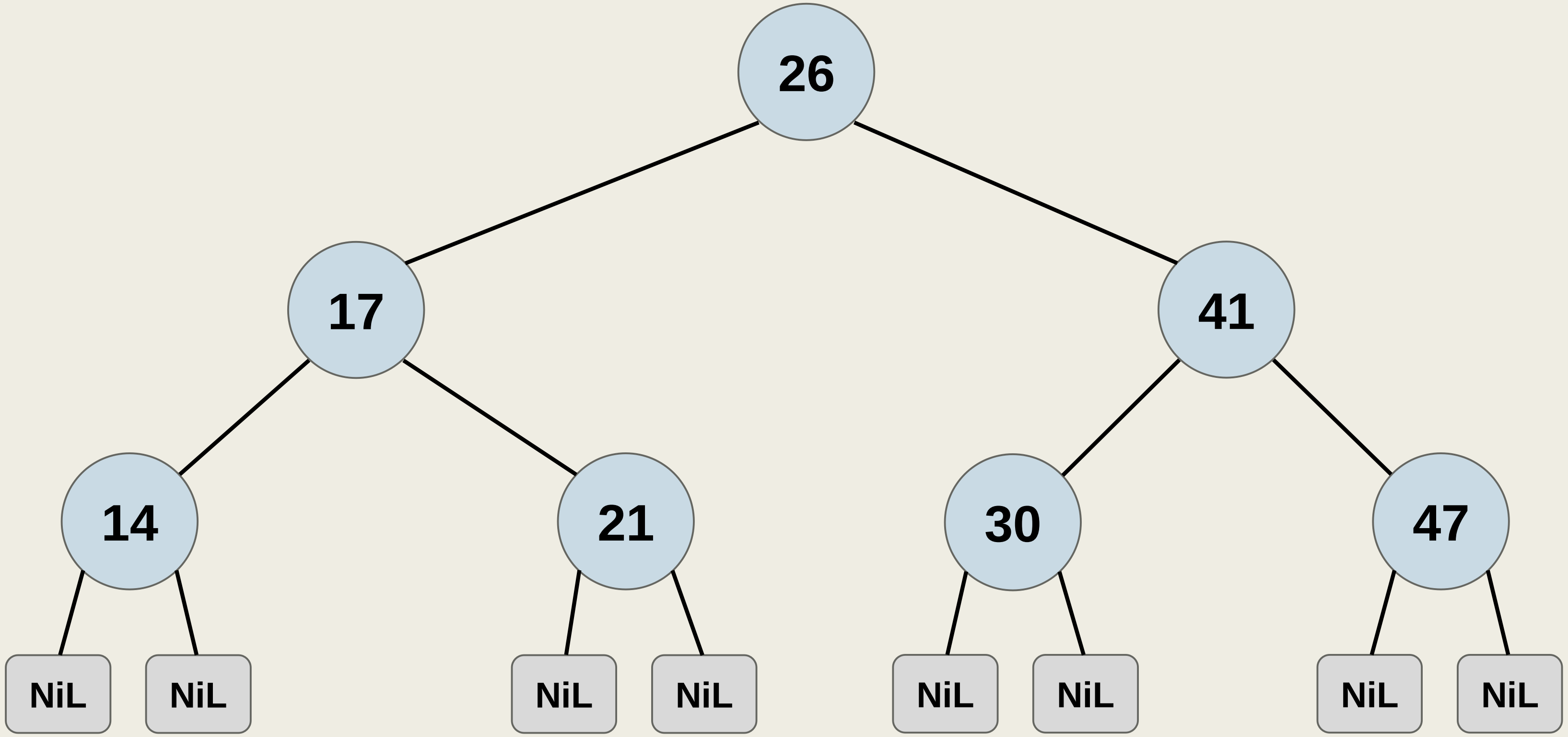
1. Each node in the RB-tree is *either* ***red*** or ***black***.
2. The ***NiL pointer*** is *considered as a* ***black node*** (with no children).
3. ***Every red node*** has *exactly two black children* nodes.
4. For each node, ***any simple path*** from that node ***to descendent leaves*** contains the *same number of black nodes*.

The *key constraint* to guarantee.

A Red-Black Tree



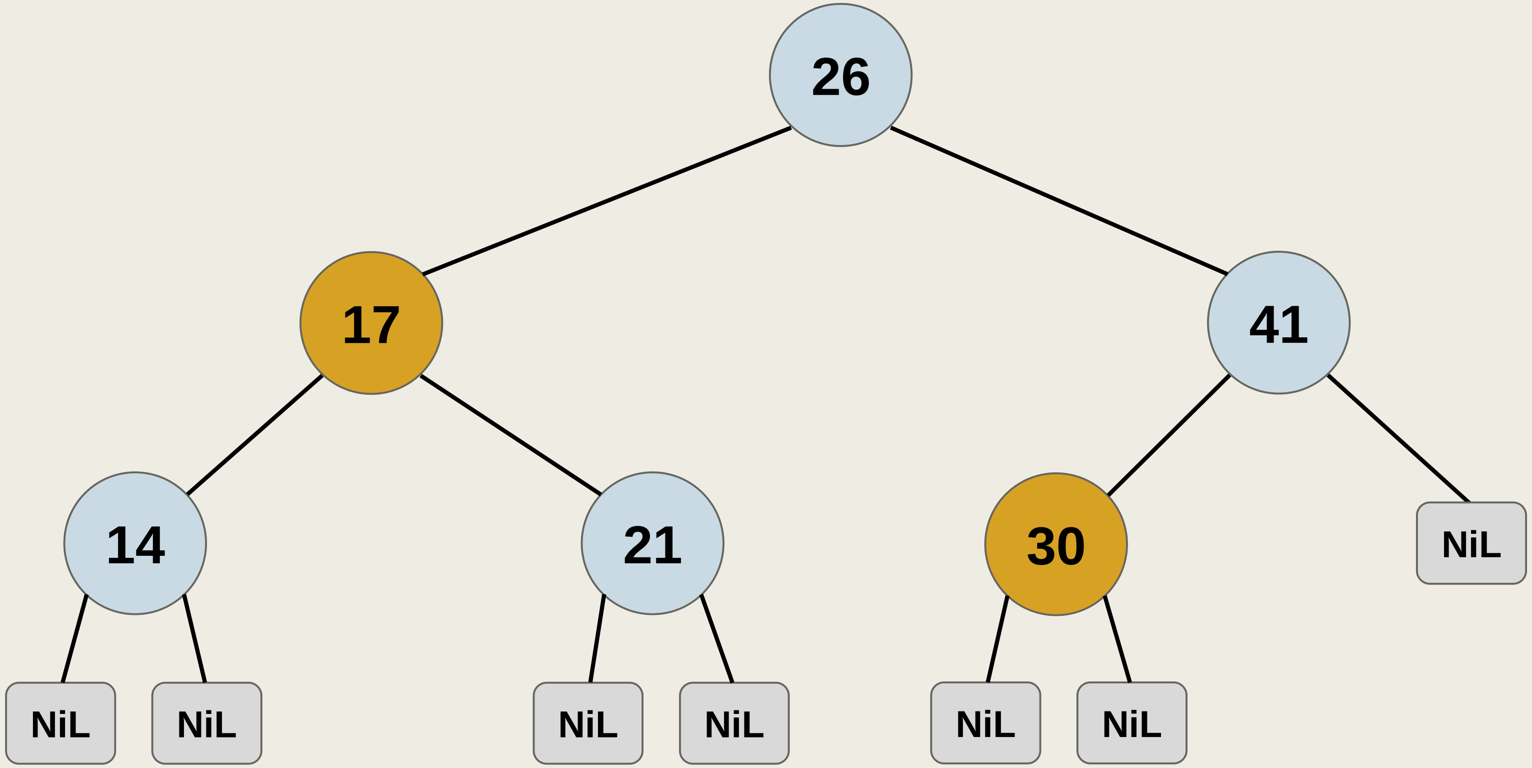
- Why Red & Black?
- Can’t we simply color all the nodes black?
  - Yes, but only when the tree is ***complete***.



- Can't we simply color all the noc
  - Yes, but only when the tree

Some nodes have to turn ***Red*** in order to maintain the ***RB-tree property***.

- When a node is missing…

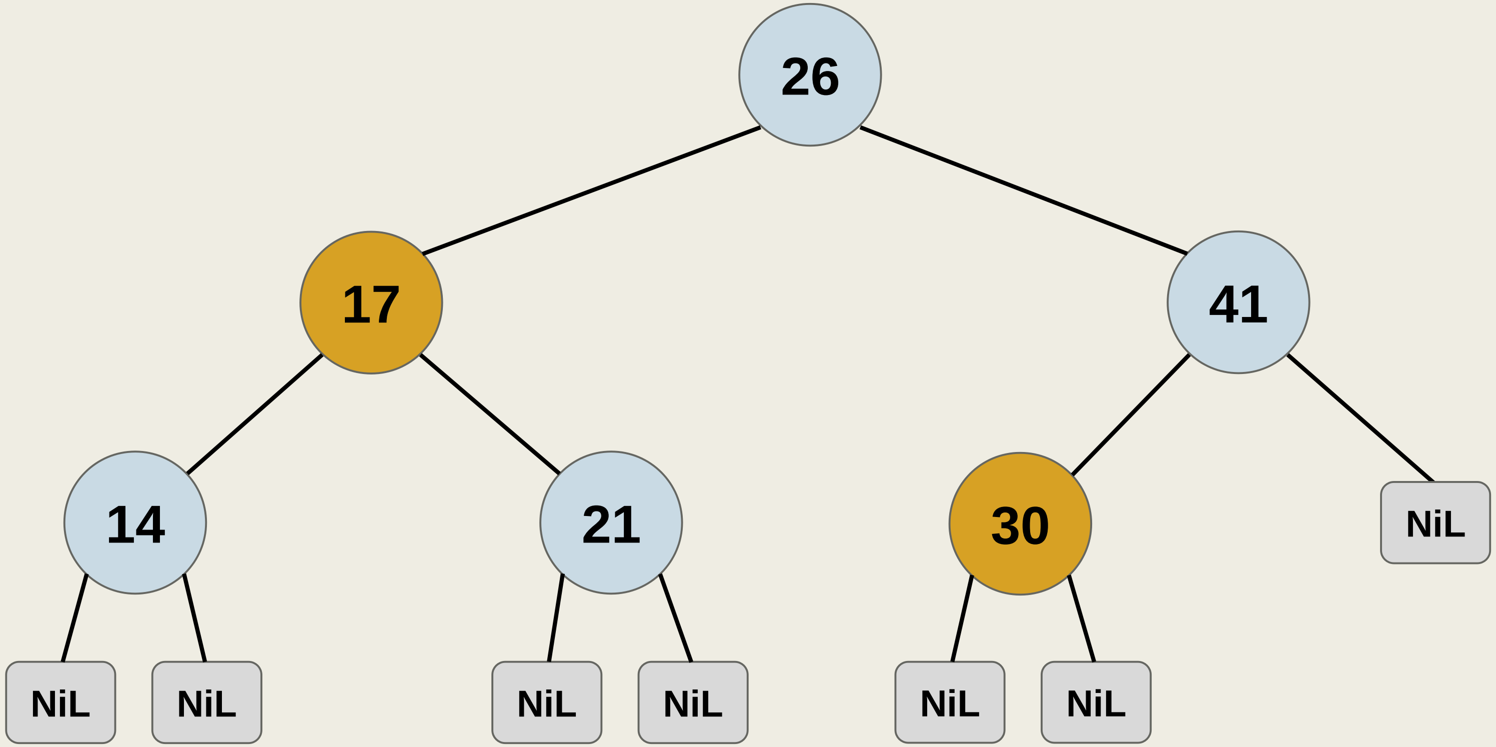


- Can't we simply color all the noc
  - Yes, but only when the tree
- When a node is missing…

Some nodes have to turn ***Red*** in order to maintain the ***RB-tree property***.

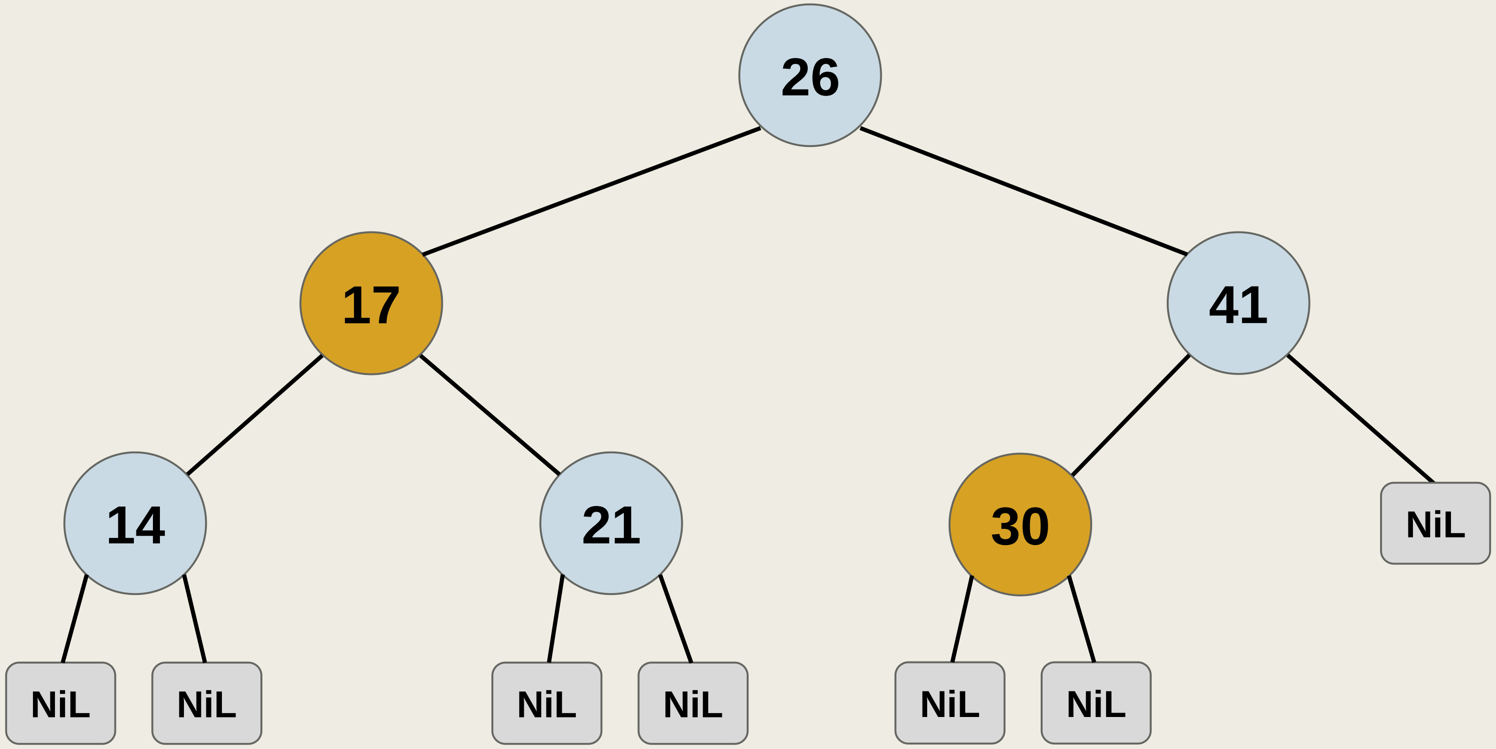
At most $O(\log n)$ nodes **need to turn red.**

Have you seen why?



Exercise

- Try to compose an efficient procedure that fixes the RB-tree property when a new node (assumed black) needs to be inserted to the tree.



Notes

- Red-Black Tree is a binary search tree imposed with extra constraints on its structure to achieve a worst-case $O(\log n)$ height guarantee.

  In the textbook, the following constraint is listed.

  5. The ***root*** is a ***black node***.

- However, this constraint is *not necessary* in obtaining the $O(\log n)$ guarantee.

Justify this.

Worst-Case Guarantee of

Red-Black Trees

The Black-Height of a Node

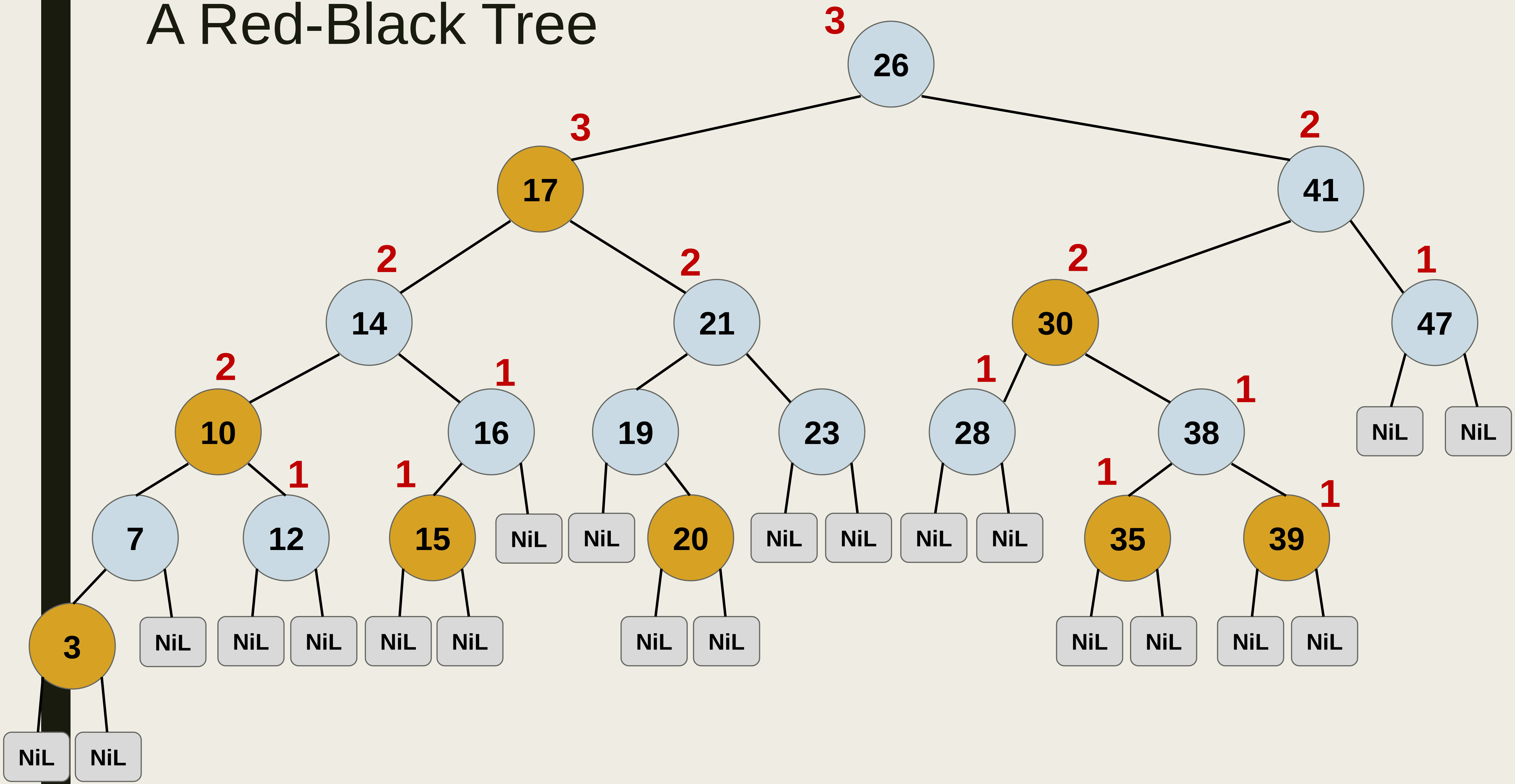
- Let T be a RB-tree.
- For any node $v \in T$,

  we define the "***<u>black-height</u>***" of the node v to be

> The number of black nodes in any path from v (but not including) to any descending (NiL) leaf node.

The black-height of any node is *<u>well-defined</u>* by the RB-tree property.

A Red-Black Tree



Claim 1.

Let $v \in T$ be a node with black-height $\mathrm{bh}(v)$.

Then the subtree rooted at v has at least $2^{bh(v)} - 1$ internal nodes.

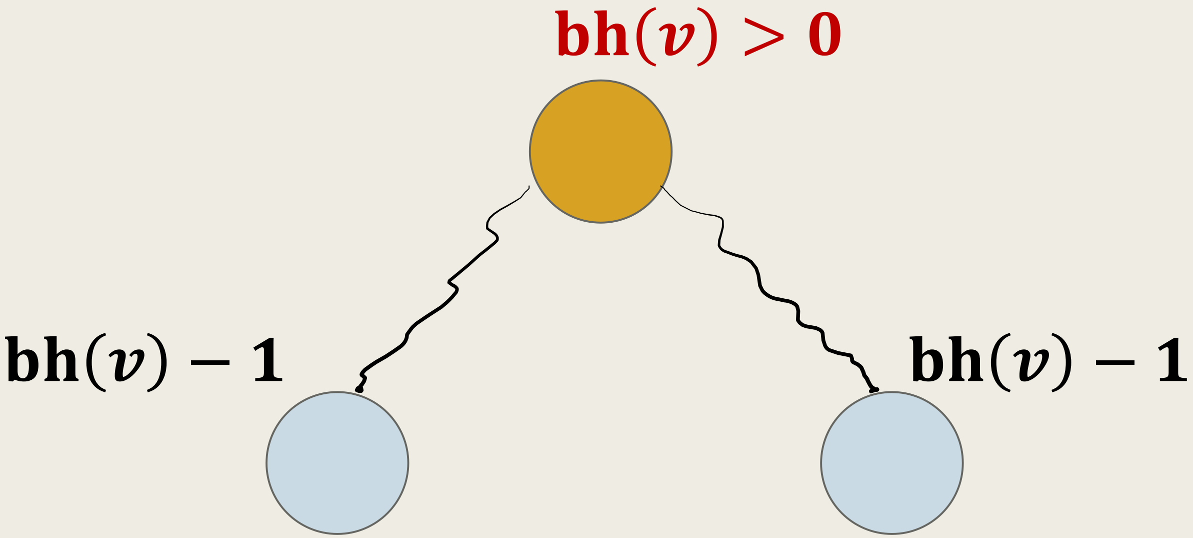
- We prove this claim by induction on $\mathrm{bh}(v)$.
  - If $\mathrm{bh}(v) = 0$,

    then $2^{bh(v)} - 1 = 0$, and the statement holds trivially.

    v is a leaf (NiL) node.
  - If $\mathrm{bh}(v) > 0$,

    then v is an *internal node* of T *with two children nodes*.

- We prove this claim by induction on $\mathrm{bh}(v)$.
  - If $\mathrm{bh}(v) > 0$,
    then v is an *internal node* of T *with two children nodes*.
  - We show that, there exists *at least one node* with ***black-height*** $\mathbf{bh}(\boldsymbol{v}) - \mathbf{1}$ both in the ***left-*** and the ***right- subtrees*** rooted at v.



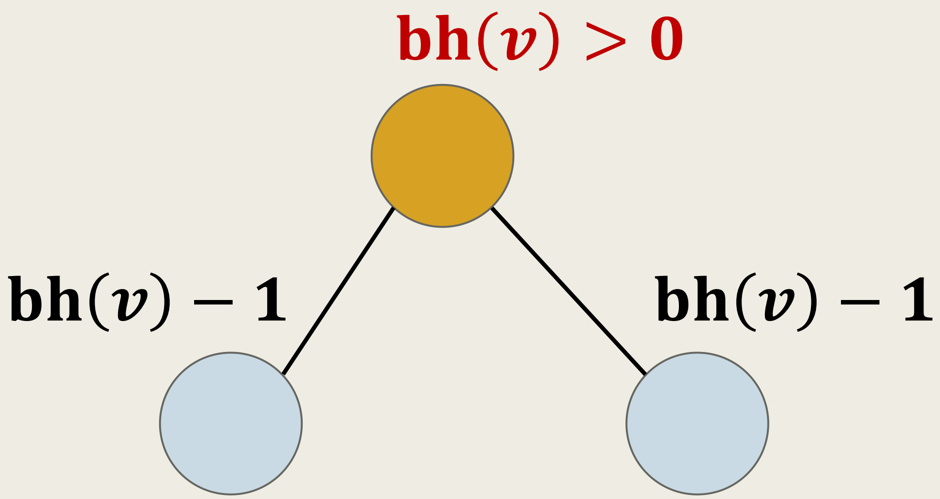
- We prove this claim by induction on $\mathrm{bh}(v)$.
  - If $\mathrm{bh}(v) > 0$,
    then v is an *internal node* of T *with two children nodes*.
  - We claim that, there exists *at least one node* with ***black-height*** $\mathbf{bh}(\boldsymbol{v}) - \mathbf{1}$ both in the ***left-*** and the ***right- subtrees*** rooted at v.
  - Then, *by the induction hypothesis*,
    the number of internal nodes at the subtree rooted at v is ***at least***

$$2 \cdot \left(2^{\mathrm{bh}(v)-1} - 1\right) + 1 \;=\; 2^{\mathrm{bh}(v)} - 1\,.$$

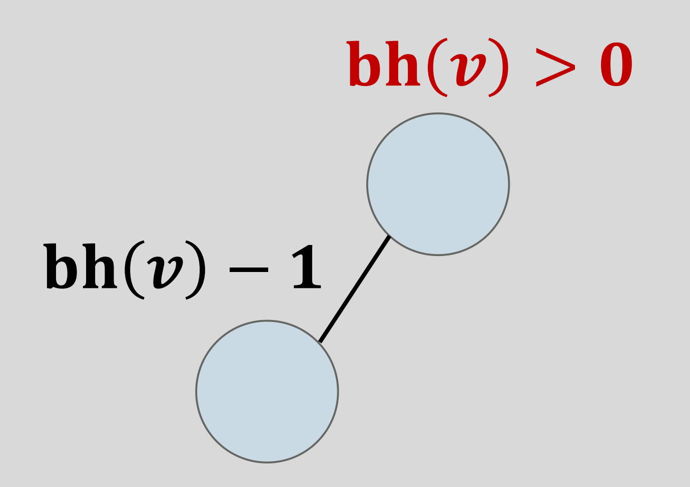
It suffices to prove the claim.

- If $\mathrm{bh}(v) > 0$, then there exists *<u>at least one node</u>* with ***<u>black-height</u>*** $\mathbf{bh}(\boldsymbol{v}) - \mathbf{1}$ both in the ***left-*** and the ***right- subtrees*** rooted at v.

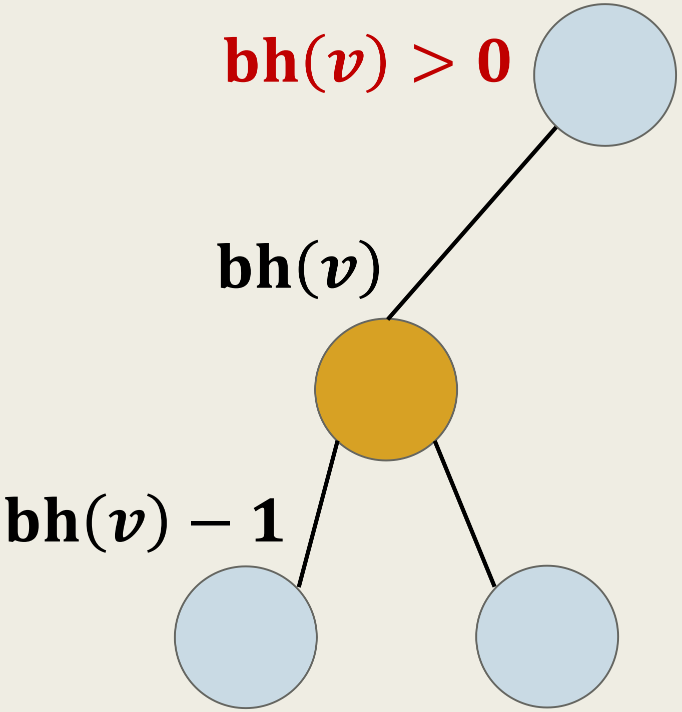
- Let us consider the color of v.
  - If $\boldsymbol{v}$ ***is <u>red</u>***,
    then it has two black children.
  - Each of them has black-height $\mathrm{bh}(v) - 1$ *<u>by definition</u>*.



- If $\mathrm{bh}(v) > 0$, then there exists <u>*at least one node*</u> wit $\mathbf{bh}(\boldsymbol{v}) - \mathbf{1}$ both in the ***left-*** and the ***right- subtrees***



- Let us consider the color of v.
  - If $\boldsymbol{v}$ ***is <u>black</u>***, then further consider the color of each of its children nodes, say, u.
    - If u is black, then it has black-height $\mathrm{bh}(v) - 1$.
    - If u is red, then it has two black children, both has black-height $\mathrm{bh}(v) - 1$.



Claim 1.

Let $v \in T$ be a node with black-height $\mathrm{bh}(v)$.

Then the subtree rooted at v has at least $2^{bh(v)} - 1$ internal nodes.

- We prove this claim by induction on $\mathrm{bh}(v)$.
  - If $\mathrm{bh}(v) = 0$, then $2^{bh(v)} - 1 = 0$, and the statement is true.
  - If $\mathrm{bh}(v) > 0$, then there exists *at least one node* with ***black-height*** $\mathbf{bh}(\boldsymbol{v}) - \mathbf{1}$ both in the ***left-*** and the ***right- subtrees*** rooted at v.

    Hence, *by the induction hypothesis*, the number of internal nodes at v is ***at least*** $2 \cdot \left(2^{\mathrm{bh}(v)-1} - 1\right) + 1 \; = \; 2^{\mathrm{bh}(v)} - 1$.

The Height Guarantee of an RB-Tree

Lemma. (Height of the RB-Tree)

An RB-tree with n internal nodes has height at most $2\log(n+1)$.

- Let T be an RB-tree with n nodes, root r, and height h.
  - By the RB-tree property,
    the root node has ***black-height at least*** $h/2$.
    - ***At most $h/2$ red node*** can exist in any root-to-leaf path.

The Height Guarantee of an RB-Tree

Lemma. (Height of the RB-Tree)

An RB-tree with n internal nodes has height at most $2\log(n+1)$.

- Let T be an RB-tree with n nodes, root r, and height h.
  - By the RB-tree property, $\mathrm{bh}(r) \geq h/2$.
  - By Claim 1,

$$n \;\geq\; 2^{\mathrm{bh}(r)} - 1 \;\geq\; 2^{h/2} - 1,$$

and hence $h \leq 2\log(n+1)$.

Operations in Red-Black Trees

Insertion / Deletion

- It remains to show that, the insertion and deletion operations for the Red-Black Trees can also be done in $O(\log n)$ time.
  - After an insertion or a deletion,
    the RB-tree property will be violated (slightly).
  - We can use ***rotations*** and ***recolor*** *some of the nodes* properly ***to adjust black-heights*** and reestablish the RB-tree property.
- The details of the two operations, however, are less interesting under the aim of this course.

Refer to the textbook for the details.

Common Self-Balancing BSTs

-- A Note

Treap

- Treap is a BST the supports the common ***insertion*** / ***deletion*** / ***look-up (search)*** operations and also two unique ***merge*** / ***split*** operations with an average-case (expected) $O(\log n)$-time guarantee.
  - Its performance guarantee is based on the assumption that each element is provided with a ***unique*** ***randomly assigned*** priority.
  - This data structure is ***very easy to implement***.
- Nevertheless, it does not provide a worst-case guarantee and may not be preferred in performance-critical applications.

The Red-Black Tree

- We have seen that the RB-trees provides insertion / deletion / look-up (search) in worst-case $O(\log n)$ time.
  - For each node, one extra bit (color) is required for storage.
  - The balancing guarantee is not strict.
    - For a node, the heights of its left- and its right- subtrees can differ ***by a factor up to*** $\mathbf{2}$.
  - The insertion / deletion operations are *intuitive* and ***relatively easy*** to implement.

The AVL Tree

- AVL tree is another self-balancing BST that provides a worst-case $O(\log n)$-time guarantee for insertion / deletion / look-up (search).
  - For each node, _one extra integer_ (balance factor) is stored.
  - It has a ***strict balancing guarantee***.
    - For each node, the heights of its left- and its right- subtrees must *differ by* ***at most 1***.
  - Hence, the look-up / search operation in AVL trees is generally faster than RB-trees.
    - Preferred by look-up intensive applications such as ***databases***.

The AVL Tree

- AVL tree is another self-balancing BST that provides a worst-case $O(\log n)$-time guarantee for insertion / deletion / look-up (search).
  - For each node, <u>*one extra integer*</u> (balance factor) is stored.
  - It has a ***strict balancing guarantee***.
    - For each node, the heights of its left- and its right- subtrees must *differ by* ***at most 1***.
  - Due to the same reason, the insertion / deletion operations are <u>***more complicated***</u> and <u>*generally slower*</u> than the RB-trees.
    - For update-intensive applications, RB-trees are preferred.

Self-Balancing BSTs

	Treap	Red-Black Tree	AVL Tree
Guarantee	Average-case	Worst-case	Worst-case
Extra Storage	$O(n)$	n bits	$O(n)$
Advantage	Simple	Faster ins / del than AVL tree	Faster look-up than RB-tree
Disadvantage		Slower look-up than AVL tree	Slower ins / del than RB-tree
Preferred by	??	Update-intensive	Look-up intensive

B-Trees

- B-Tree is a self-balancing search tree that is designed to ***work well*** on ***<u>disk drivers</u>*** or other ***<u>direct-access</u>*** ***<u>secondary storage devices</u>***.
  - Each leaf has the same height.
  - For each node v,
    - v may store multiple keys that are sorted in order.
    - v has $k + 1$ children nodes if k keys are stored.
    - The ***<u>stored keys</u>*** divides the range of values that can be stored in the subtrees.
  - The leaf nodes have no children nodes.

B-Trees

- B-Tree is a self-balancing search tree that is designed to ***work well*** on ***<u>disk drivers</u>*** or other ***<u>direct-access</u>*** ***<u>secondary storage devices</u>***.
  - Each leaf has the same height.
  - For a given value $t \geq 2$,
    - Each node can store up to $2t - 1$ keys.
    - Each non-root node must store at least $t - 1$ keys.

Refer to the textbook for the details.