Lecture #9 Advanced Data Structure (1)

Algorithm

JBNU Spring 2021

Jinhong Jung

In Previous Lecture

Linear selection algorithm on average

- quick-select() using Lomuto partition to select i-th smallest element in an array
- Has $\Theta(n)$ on average, and $\Theta(n^2)$ for a worst case

Linear selection algorithm in a worst case

- A fixed skewness in partitioning with a linear overhead leads to $\Theta(n)$ time for a worst case
- For that, mom-select() uses the median of medians as a pivot, and partitions the input array by MoM
- Has $\Theta(n)$ for a worst case

In This Lecture

Advanced data structure

- Red-black tree: self-balancing binary search tree

- Why do we need to consider the balance of BST?

- How to preserve the balance of BST?

Outline

Search problem and binary search tree

Red-black tree

- Self-balancing binary search tree
- Insertion
- Remove
- Analysis

Search

To find information that you want in massive data

- Examples
  - Search for a query in Google
  - Search for the digit of your friend in your cellphone
- Important to **efficiently** find the information

Problem Definition

Search

- **Input**: a key x and a data structure
- **Output**: the value (record) having the key x

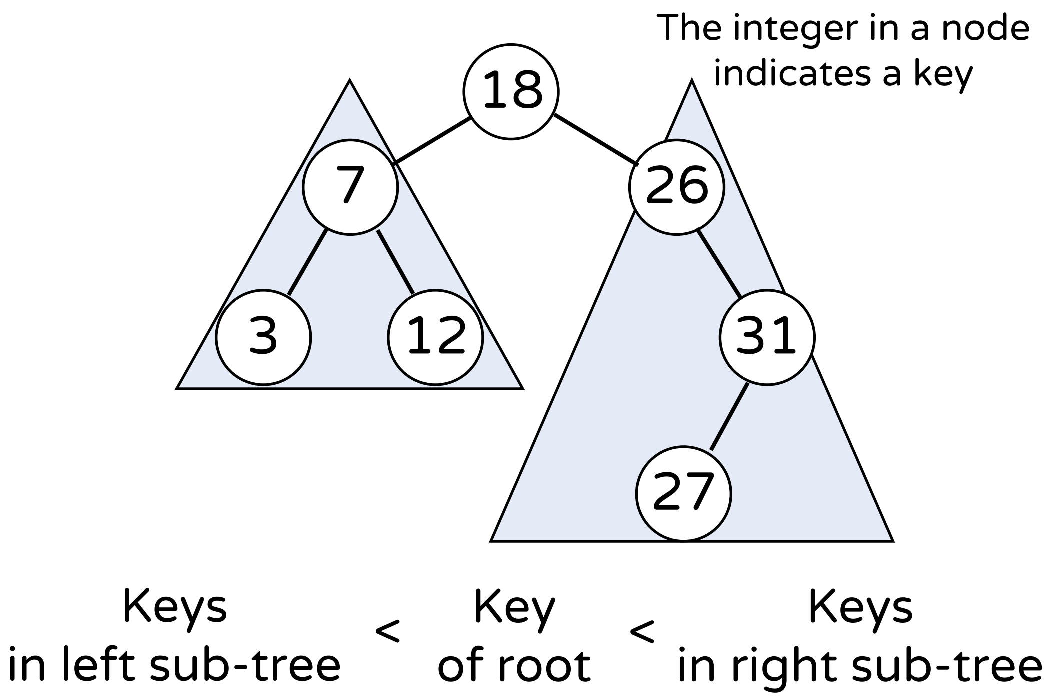
Data structure for search

- Should support efficient operations such as **search**, **insert**, and **remove**
- Data structures that we have learnt
  - List
  - Binary Search Tree << Today's topic
  - Hash

Binary Search Tree (1)

Binary tree for search satisfying BST properties

- Keys in a left sub-tree < the key of the root
- Keys in a right sub-tree > the key of the root
- Both left & right sub-tress are BST recursively



Binary Search Tree (2)

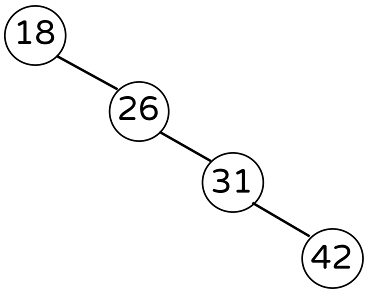
Main operations of BST

- search(key)
  - Search for a node having the querying key in the BST
- insert(key, value)
  - Insert a node having the key and value into the BST
  - After the operation, the tree should be BST
- remove(key)
  - Remove a node having the key from the BST
  - After this operation, the tree should be BST

Binary Search Tree (3)

Limitation of BST

- The main operations of BST having n nodes takes $O(\log n)$ time for the best and average case
  - In these cases, the tree's height is $O(\log n)$
- However, they takes $O(n)$ for a worst case
  - Consider sorted keys are inserted sequentially
  - Such trees are called degenerate tree



- Thus, operations of unbalanced BST are inefficient compared to those of balanced BST
- **Q. How can we make the BST balanced even for a worst case?**

Outline

Search problem and binary search tree

Red-black tree

- Self-balancing binary search tree
- Insertion
- Remove
- Analysis

Self-Balancing BST

The goal of self-balancing BST

- To preserve the balance of BST after insert and remove operations
  - So that the tree's height is $O(\log n)$ for a worst case
- The point is how to insert and remove a node while preserving the balance

Representative self-balancing BST

- AVL tree
- Red-black tree << in this lecture
  - It supports faster insertion and removal than AVL
  - It is more memory-efficient than AVL

Red-Black Tree

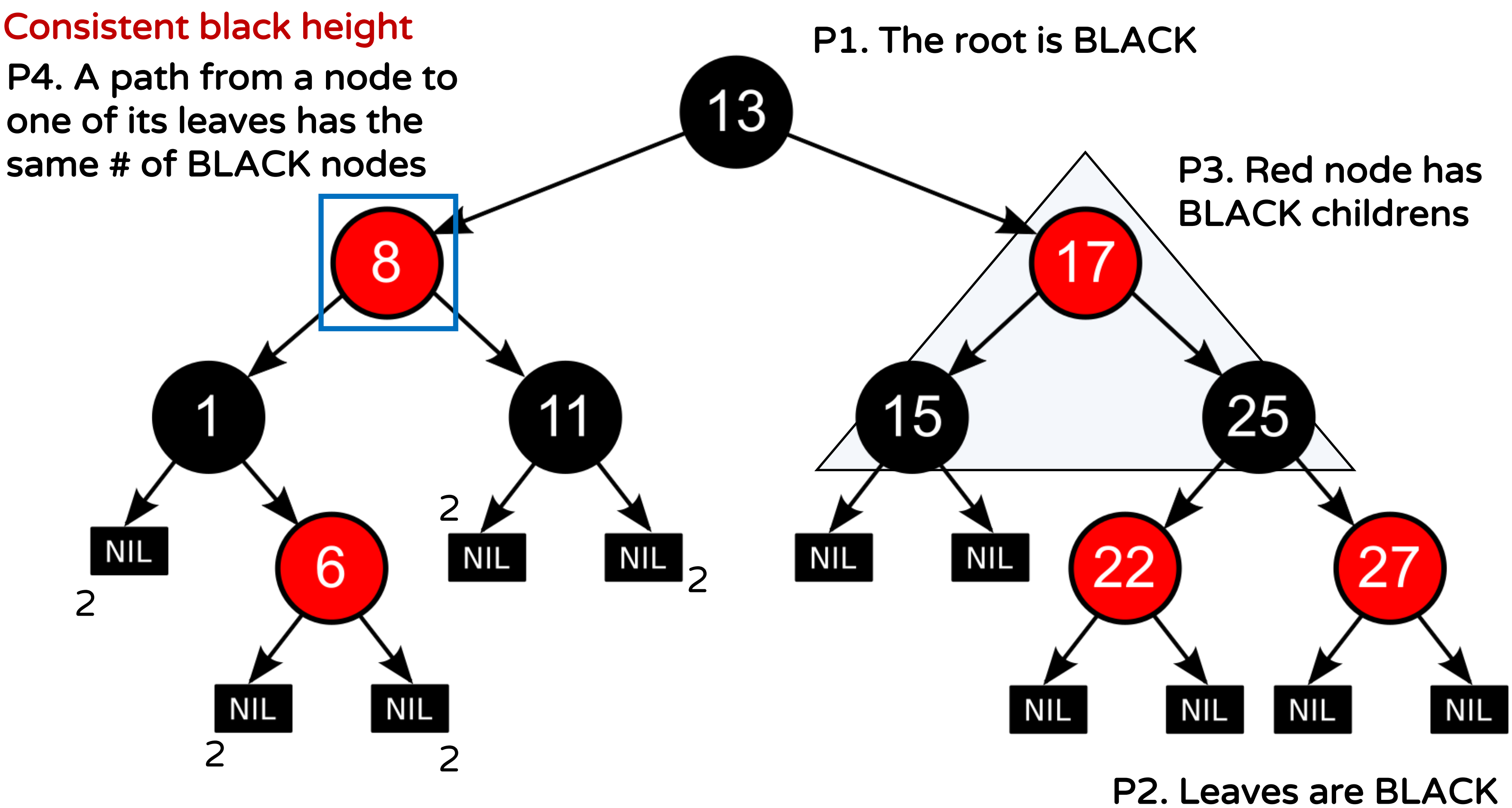
Definition of red-black tree

- Each node is colored by either "RED" or "**BLACK**"
  - Represented as one bit (e.g., 0 or 1)
  - Used to ensure that the tree remains balanced during operations

- **Red-black tree properties**
  - P1) The root node should be **BLACK**
  - P2) All leaf (NIL) nodes are considered as **BLACK**
  - P3) If a node is RED, then both its children must be **BLACK**
  - P4) Every path from a given node to any of its descendant NIL leaves goes through the same number of **BLACK** nodes

Red-Black Tree

Example of red-black tree



- NIL indicates an explicit leaf node
- When implementing this, use a single NIL node and point it

Red-Black Tree

Operations of red-black tree

- For search, red-black tree has the same operation of BST
  - Because this operation does not modify the tree

- For insert and remove operations,
  - RBT first performs the insert or remove operation of BST
  - Then, the modified tree’s balance can be broken
  - Thus, RBT needs to **properly re-arrange the modified tree** so that it satisfies the red-black tree properties
    - Using **re-coloring** and **rotations**

Outline

Search problem and binary search tree

Red-black tree

- Self-balancing binary search tree
- Insertion
- Remove
- Analysis

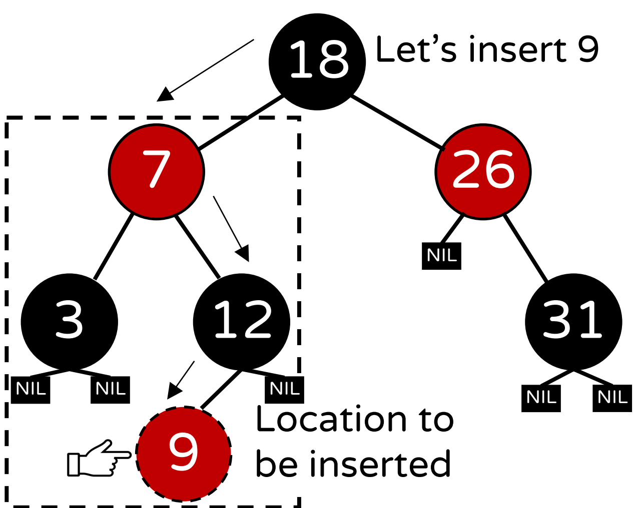
Insert of RBT (1)

Insert operation

- Step 1) insert a node into the tree using BST's insert operation
  - This new node is colored by "**RED**" at first
- Step 2) re-arrange the modified tree case-by-case

Note that

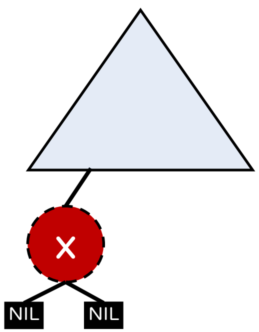
- The new node will be located at the bottom of the tree after BST's insertion
- Let's do case study around the inserted location



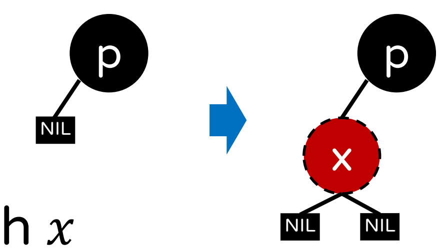
Insert of RBT (2)

Observation

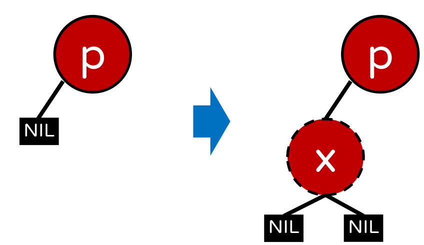
- When a node x is inserted, there is no violation below the inserted node
  - Because this node is red and had two NIL (**BLACK**) nodes



- This means we need to check some violations above the node
  - Let p denote the parent of the insert node
- If p is **BLACK**, then there is no violation
  - For P4, the previous NIL is properly replaced with x



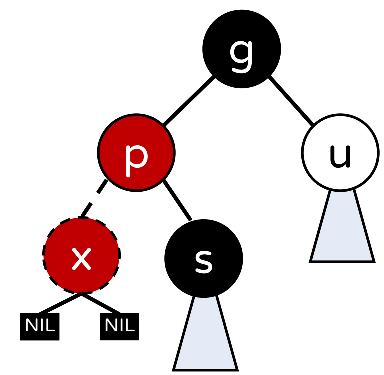
- If p is RED, then there is a violation!
  - P3 is violated (the RED node has a RED child)
  - How to handle this case?



Insert of RBT (3)

Case study when p is RED

- Before the insertion, the parent of p should be **BLACK**
  - Let g denote the grandparent x
- If p has a child, it's the sibling of x and should be **BLACK**
- The sibling of p (uncle) is either RED or **BLACK**
  - Let u denote the uncle of x (sibling of p)
- Cases for re-arrangement
  - Case 1: u is RED
  - Case 2: u is **BLACK**
    - [LR] Case 2-1: p is the left of g & x is the right of p
    - [LL] Case 2-2: p is the left of g & x is the left of p
    - [RL] Case 2-3: p is the right of g & x is the left of p (mirror of case 2-1)
    - [RR] Case 2-4: p is the right of g & x is the right of p (mirror of case 2-2)

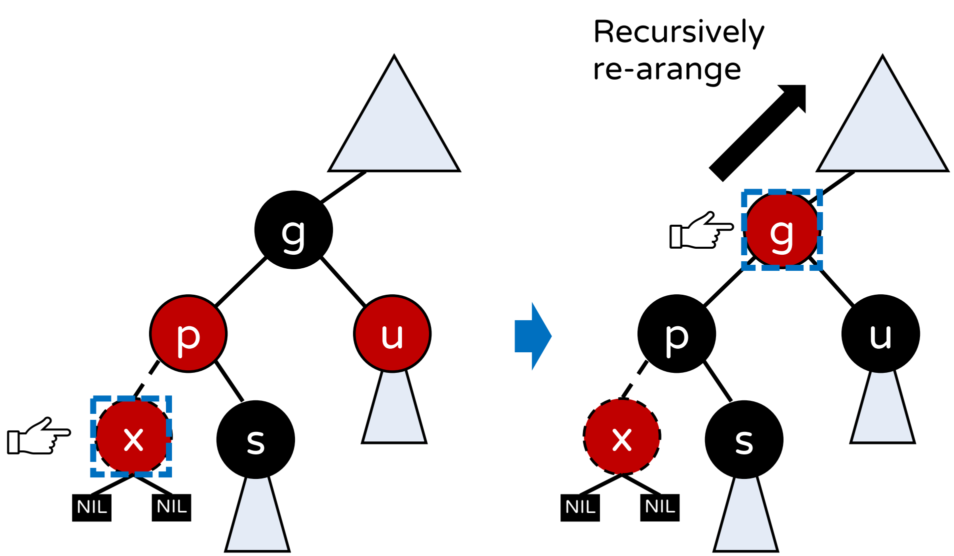


Insert of RBT (4)

EOP: end-of-procedure

Case 1: u is RED

- 1) Change the color of p and u to **BLACK** & g to RED
  - If p^2 is the root, then change g to black; [EOP]
- 2) Recursively handle a potential violation on g
  - i.e., when the parent of g is **RED**



Handle the violation on x

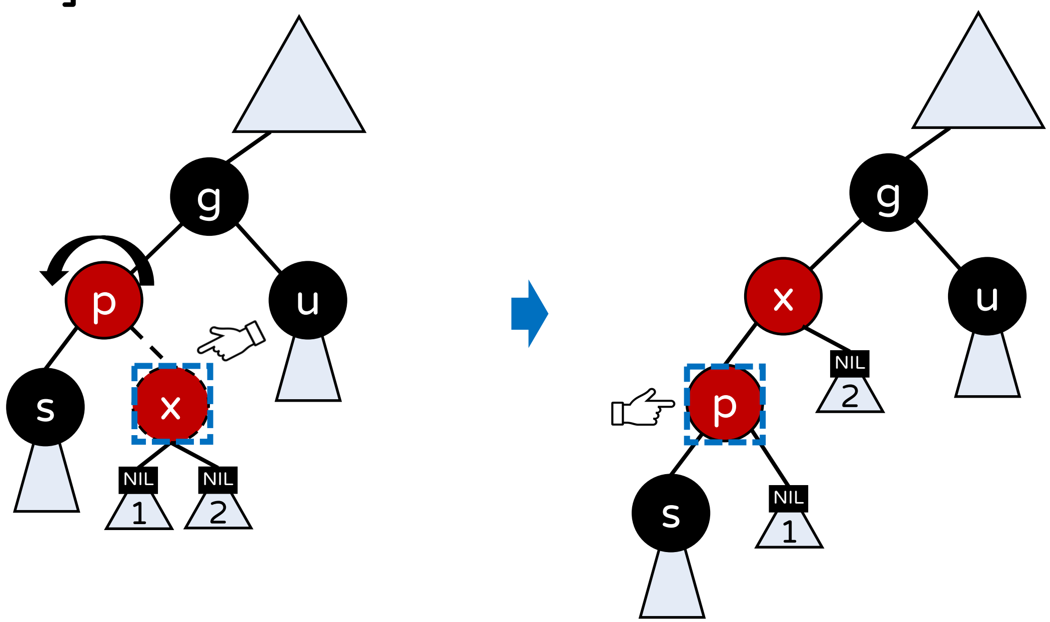
Handle the violation on g

Insert of RBT (5)

[LR] Case 2-1: p is the left of g & x is the right of p

- 1) Rotate the tree of p left
  - Note that $x > p$
    - p becomes the left child of x
    - The left sub-tree of x becomes the right child of p
- 2) Go to [LL] Case 2-2

LR rotation on p



Case 2-1 on x

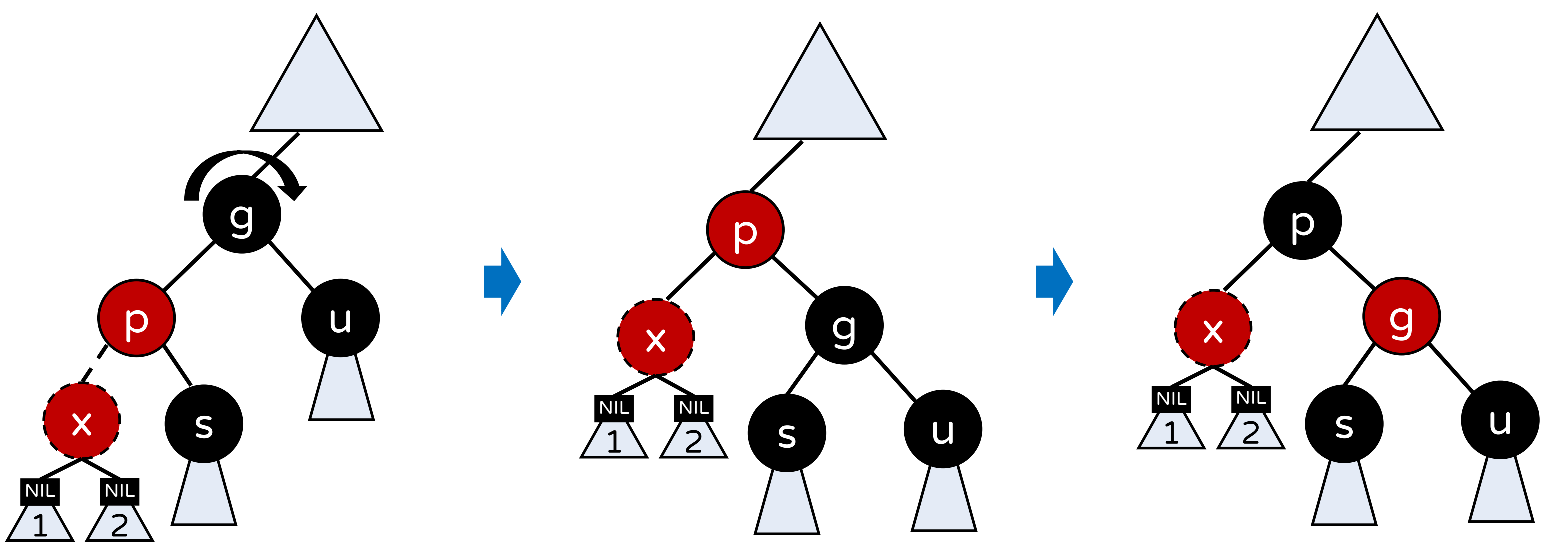
Case 2-2 on p

Insert of RBT (6)

[LL] Case 2-2: p is the left of g & x is the left of p

- 1) Rotate the tree of g right
  - Note that $g > s$
    - s becomes the left child of g
- 2) Swap the color of p and g [EOP]

LL rotation on g



Insert of RBT (7)

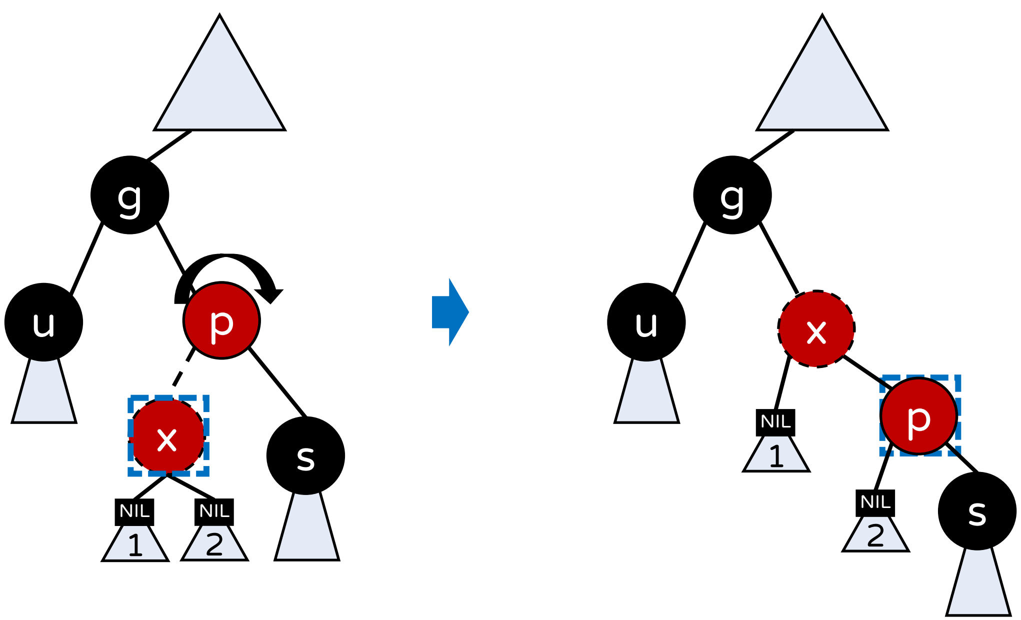
[RL] Case 2-3: p is the right of g & x is the left of p

- 1) Rotate the tree of p right
  - Note that $x < p$
    - p becomes the right child of x

      The right sub-tree of x becomes the left child of p

RL rotation on p

- 2) Go to [RR] Case 2-4

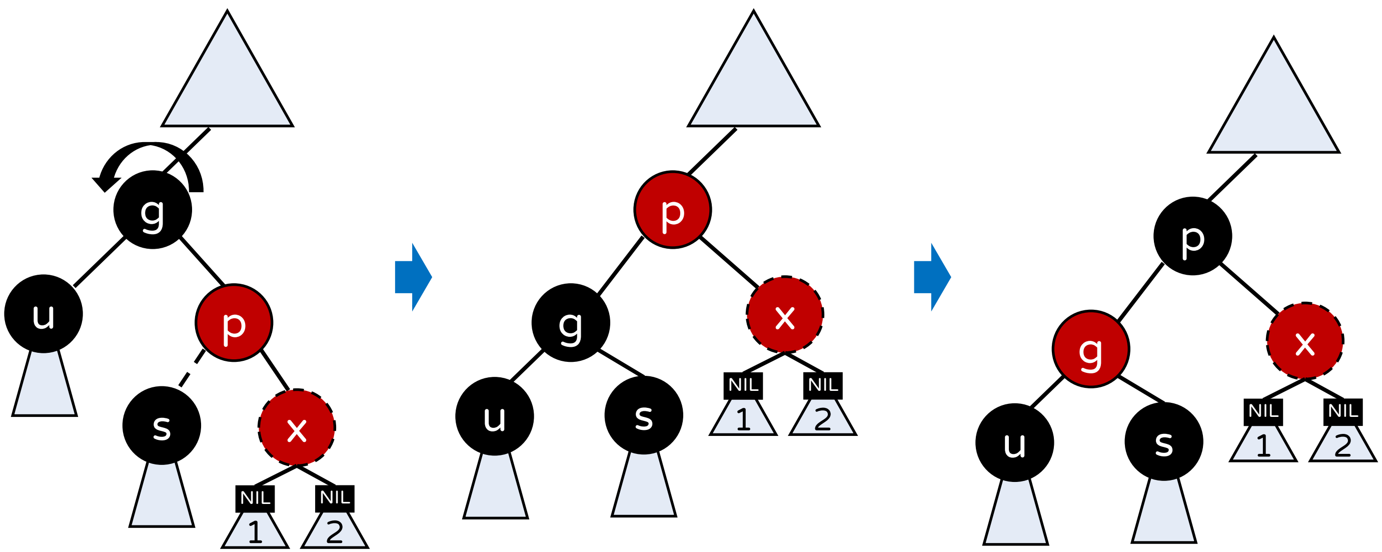


Insert of RBT (8)

[RR] Case 2-4: p is the right of g & x is the right of p

- 1) Rotate the tree of g left
  - Note that $g < s$
    - s becomes the right child of g
- 2) Swap the color of p and g [EOP]

RR rotation on g



Insert of RBT (9)

Pseudocode of insert (x is the newly inserted node)

- Step 1) Do BST's insert on x and make the color of x RED
- Step 2) If x is root, change the color of x as **BLACK** [EOP]
- Step 3) Else if x isn't root & x's parent p is RED
  - **If case 1 – x's uncle u is RED**
    - Change the color of p and u to black & g to RED
    - Recursively handle a potential violation on g ($x \leftarrow g$ & repeat steps 2 & 3)
  - **Else if [LR] case 2-1**
    - Do LR rotation on p & go to case 2-2 ($x \leftarrow p$ & repeat step 3)
  - **Else if [LL] case 2-2**
    - Do LL rotation on g & swap the color of p and g [EOP]
  - **Else if [RL] case 2-3 (mirror of case 2-1)**
    - Do RL rotation on p and go to case 2-4 ($x \leftarrow p$ & repeat step 3)
  - **Else if [RR] case 2-4 (mirror of case 2-2)**
    - Do RR rotation on g & swap the color of p and g [EOP]

Outline

Search problem and binary search tree

Red-black tree

- Self-balancing binary search tree
- Insertion
- Remove
- Analysis

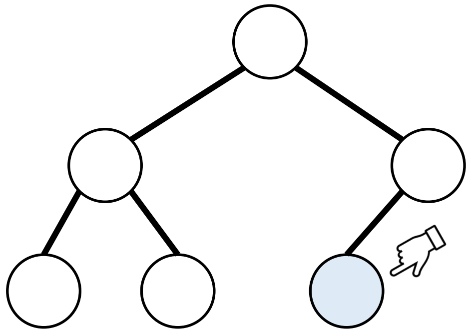
Remove of RBT (1)

Remove operation

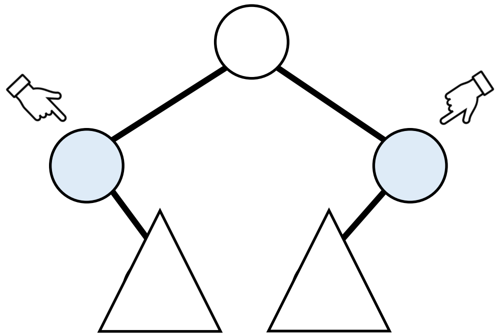
- Step 1) remove a node using BST's remove operation
- Step 2) re-arrange the tree case-by-case

Cases for remove in BST

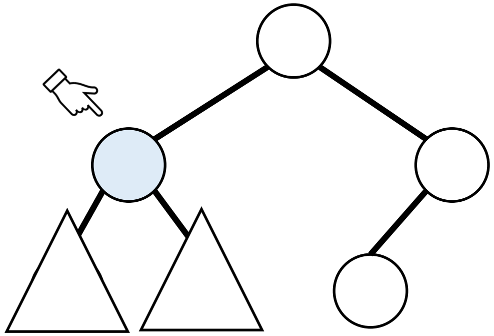
- **NC**) the target node has **Not a Child**
- **OC**) the target node has **One** (left | right) **Child** (sub-tree)
- **TC**) the target node has **Two Children** (sub-trees)



NC case



OC case

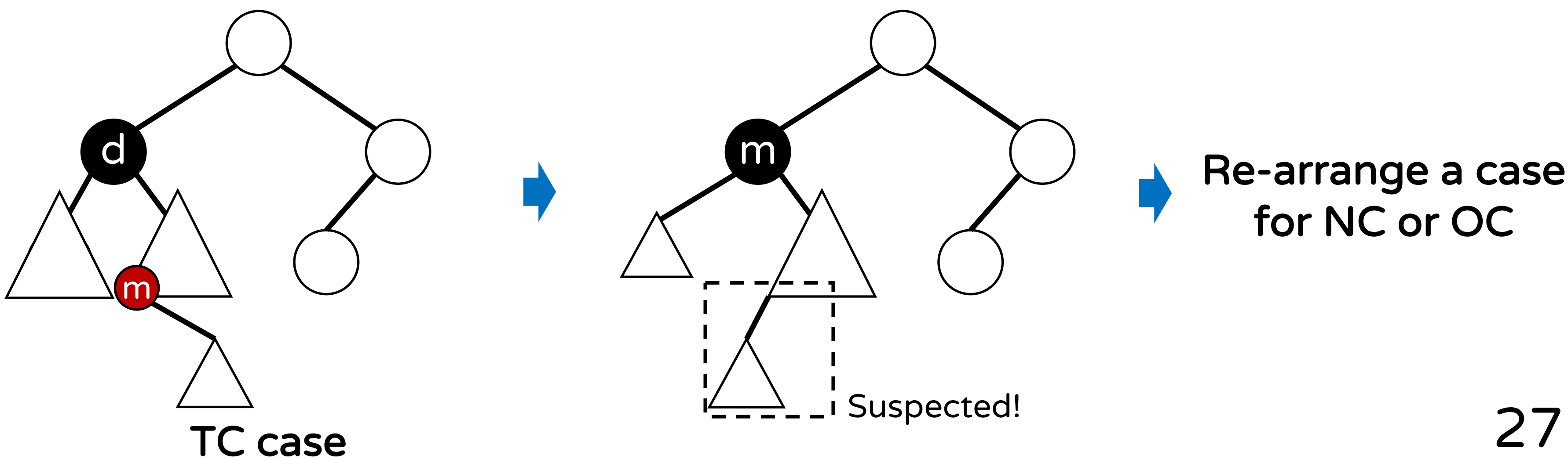


TC case

Remove of RBT (2)

For TC case, BST's remove

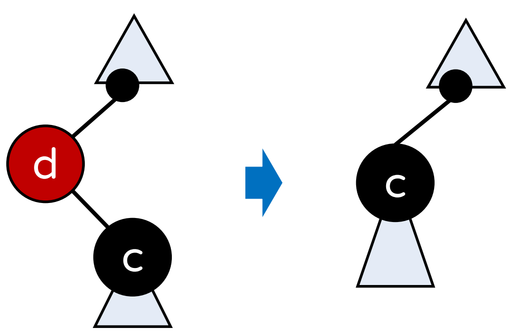
- Replaces the target node d's key with successor's key
  - The successor m is the minimum node of the right sub-tree
  - Only keys are swapped; thus, there is no violation in this step
- Remove the successor
  - Note that the successor has at most one child
  - i.e., TC case is reduced to NC or OC case



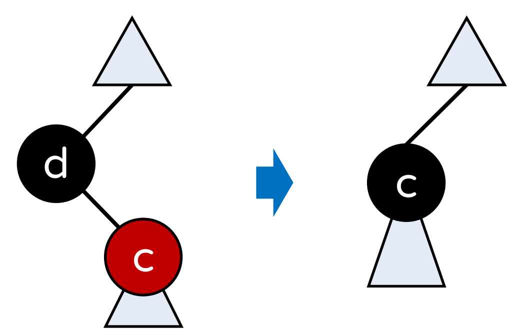
Remove of RBT (3)

Remove operation

- After BST's remove, re-arrange the tree case-by-case
  - Handle cases where a node hasn't a child or had one child
  - Let d be the node to be deleted and c be its child
- Simple cases
  - If d is RED, then there is no violation
    - Meaning we only need to check when d is **BLACK**
  - If d is **BLACK** and c is RED, then there is no violation
    - Change the color of c to **BLACK** after removing d



When d is RED

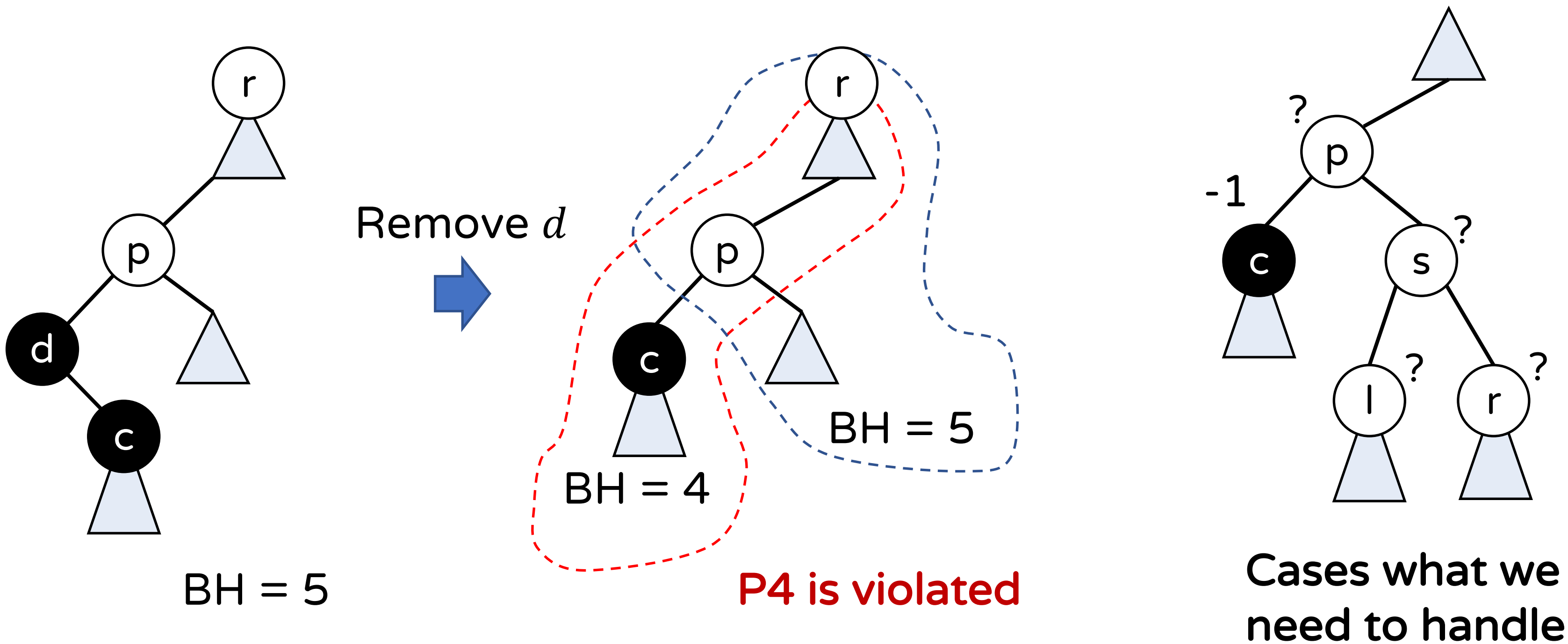


When d is **BLACK**

Remove of RBT (4)

Double-black case

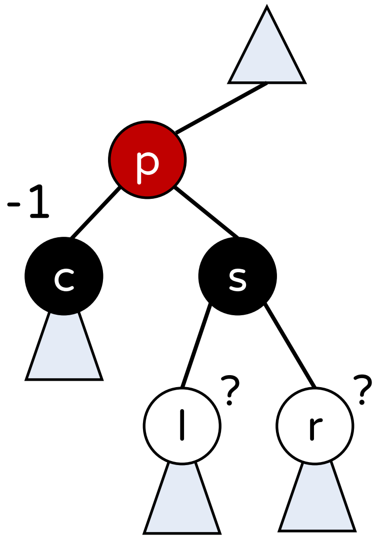
- Called when d is **BLACK** and c is **BLACK**
- After removing d, P4 is violated (i.e., black height is not consistent)



Remove of RBT (5)

Case study

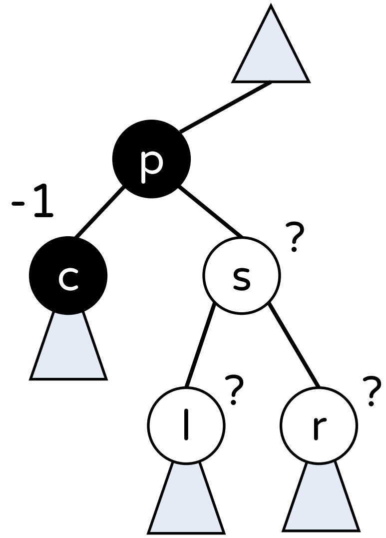
- Case 1: p is **RED** $\Rightarrow s$ must be **BLACK**
  - Case 1-1: $< l, r >$=<B, B>
  - Case 1-2: $< l, r >$=<R, R> or <B, R> $\Rightarrow$ <$*$, R>
  - Case 1-3: $< l, r >$=<R, B>



Remove of RBT (6)

Case study

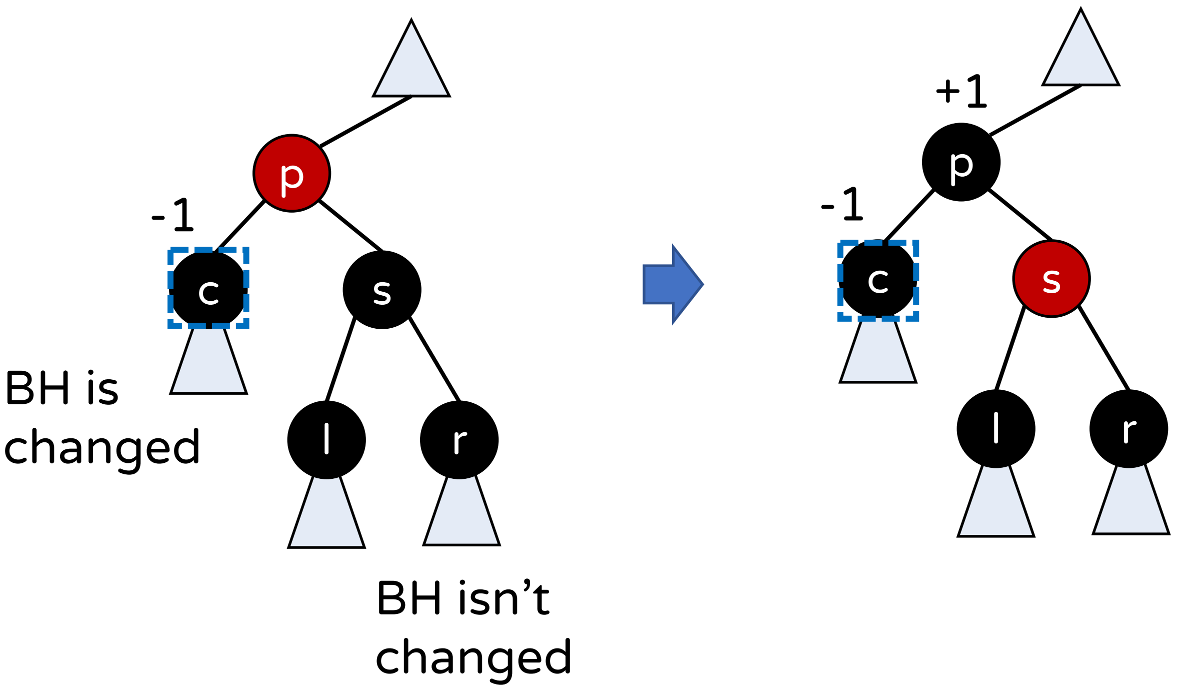
- Case 2: p is **BLACK**
  - Case 2-1: $< s, l, r >$=**<B, B, B>**
  - Case 2-2: $< s, l, r >$=**<B, R, R>** or **<B, B, R>** ⇒ <B, $*$, **R**>
  - Case 2-3: $< s, l, r >$=**<B, R, B>**
  - Case 2-4: $< s, l, r >$=**<R, B, B>**
    - If s is red, both l and r are black



Remove of RBT (7)

Case 1-1

- Change the color of p and s

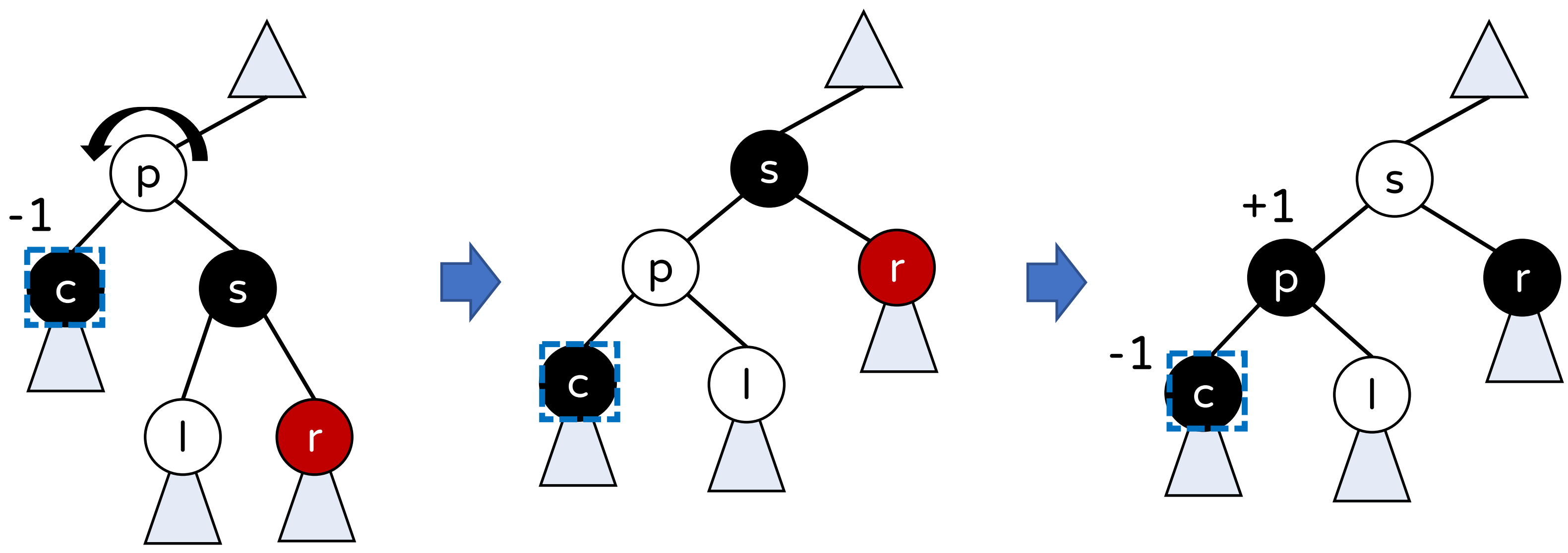


Remove of RBT (8)

Case $*$-2 (Cases 1-2 and 2-2)

- Case 1-2: $< l, r >=$ <$*$, R>
- Case 2-2: $< s, l, r >=$ <B, $*$, R>

- Rotate the tree of p left
- Swap the color of p and s & change r to **BLACK**

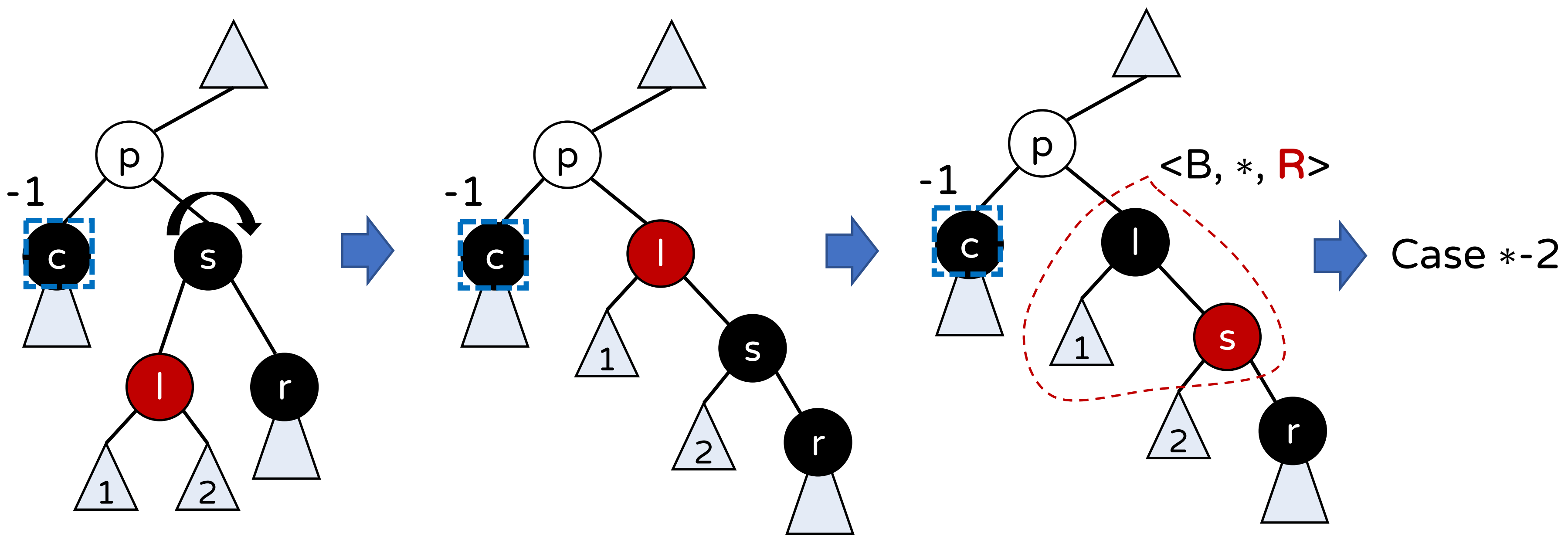


Remove of RBT (9)

Case $*$-3 (Cases 1-3 and 2-3)

- Case 1-3: $< l, r >$=<R, B>
- Case 2-3: $< s, l, r >$=<B, R, B>

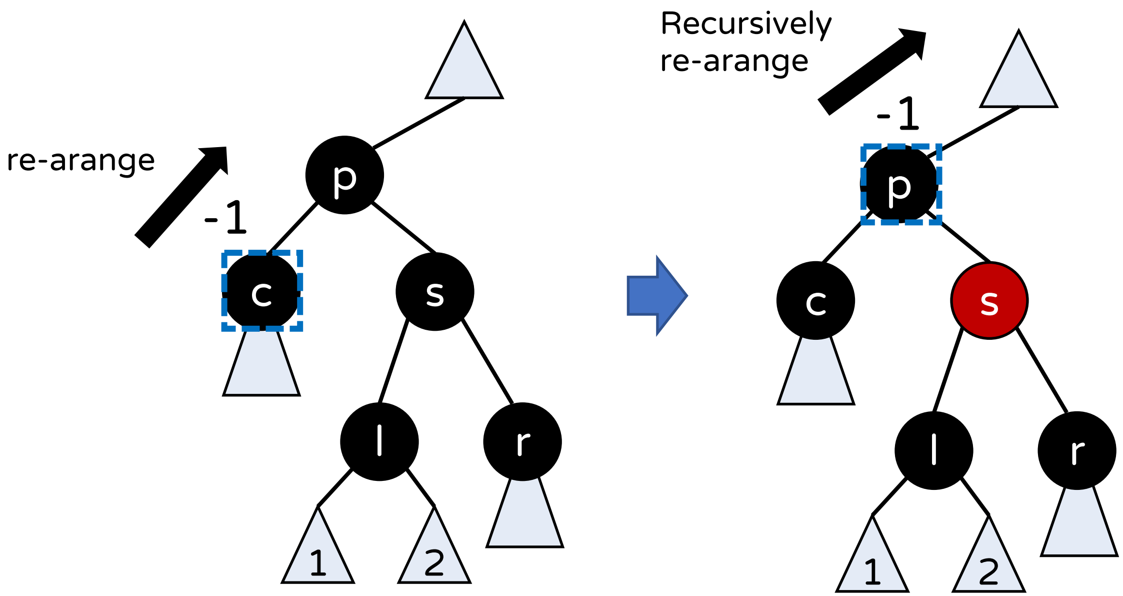
- Rotate the tree of s right
- Swap the color of l and s & go to Case $*$-2



Remove of RBT (10)

Case 2-1 ($< s, l, r >$=<B, B, B>)

- Change s to RED
  - The paths via node p lack one black node now
- Recursively handle the problem on p

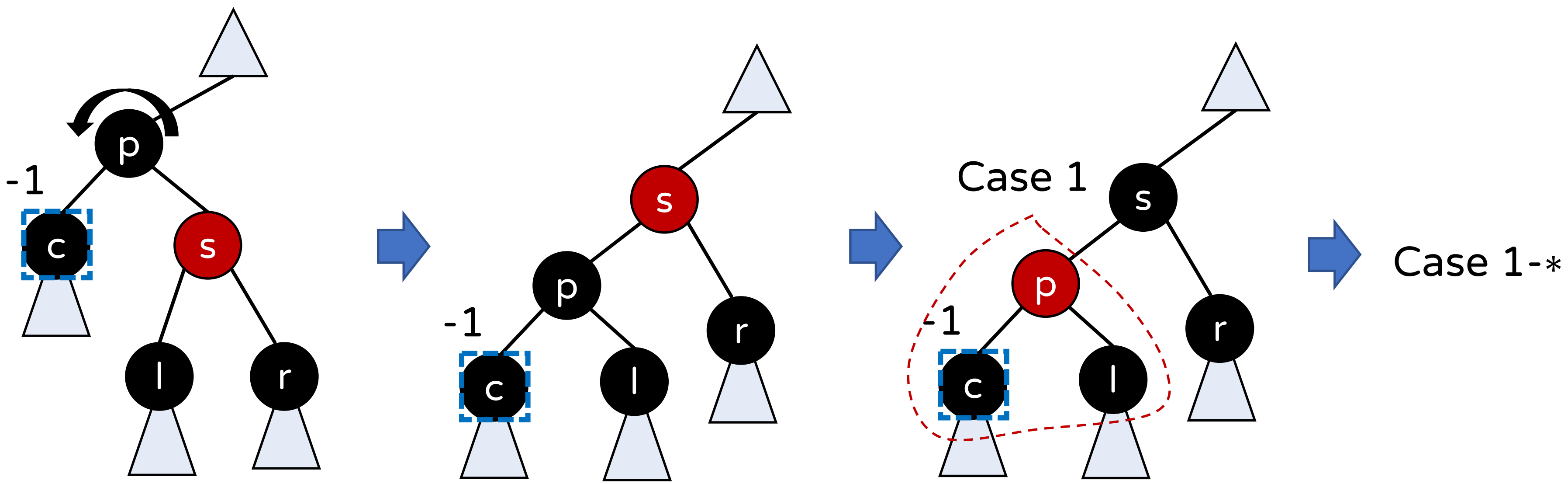


Problem initiated by c Problem initiated by p

Remove of RBT (11)

Case 2-4 ($< s, l, r >$=<R, B, B>)

- Rotating the tree of p left
- Swap the color of p and s & go to case 1



Remove of RBT (12)

Pseudocode of remove (d is the node to be delete)

- Step 1) Do BST's remove on d
- Step 2) If d is root, mark root NIL [EOP]
- Step 3) Else if d isn't root & d is **BLACK** & c is **BLACK**
  - **If Case 1-1,** then change the color of p and s [EOP]
  - **Else if Case $*$-2**
    - Rotate the tree of p left & swap the color of p and s & change r to **BLACK [EOP]**
  - **Else if Case $*$-3**
    - Rotate the tree of s right & swap the color of l and s & go to Case $*$-2
  - **Else if Case 2-1**
    - Change s to **RED** & recursively handle the problem on p ($c \leftarrow p$ and go to Step 3)
  - **Else if Case 2-4**
    - Rotating the tree of p left & swap the color of p and s & go to case 1-$*$

Outline

Search problem and binary search tree

Red-black tree

- Self-balancing binary search tree
- Insertion
- Remove
- **Analysis**

Analysis of RBT (1)

Maximum height of RBT

- RBT with n internal nodes has height at most $O(\log n)$
  - Using the fact: a sub-tree of black height $\mathrm{bh}(v)$ in an RBT contains at least $2^{\mathrm{bh}(v)} - 1$ internal nodes
    - $\mathrm{bh}(v)$: # of black nodes on a path from v to a leaf node (not counting v)
    - Proved by induction (see Appendix)
  - Let h be the height of the RBT, and let P denote a path from the root to the furthest leaf
    - At least half of nodes on P must be **BLACK** (no two consecutive red nodes)
  - Hence, the black height of the root is at least $h/2$
  - By the fact, the RBT has $2^{h/2} - 1$ internal nodes at least

$$2^{h/2} - 1 \leq n \Leftrightarrow h \leq 2\log(n+1) = \mathrm{O}(\log n)$$

Analysis of RBT (2)

Time complexity of RBT's operations

- Search: $O(\log n)$ since RBT has $O(\log n)$ height at most
  - RBT's search is equal to BST's search
- Insert and remove: $O(\log n)$
  - Re-coloring and rotation take constant time
  - For some cases, we need to do something recursively toward the root $\Rightarrow O(\log n)$

Space complexity of RBT

- $O(n)$ space
  - To store n nodes (key, value, pointers) + 1 bit for each node

What You Need To Know

Red-black tree (used in std::map)

- Why do we need red-black tree?
  - $\Rightarrow$ For a worst case, BST has $O(n)$ height, but RBT guarantees $O(\log n)$ height
- Definition and properties
  - BST where each node is colored by either **RED** or **BLACK**
    - P1) **BLACK** root node
    - P2) All **BLACK** leaf (or NIL) nodes
    - P3) No two consecutive RED nodes
    - P4) Consistent black height
- Insert and remove
  - Do BST's corresponding operation and re-arrange violated area using re-colorring and rotation case by case

In Next Lecture

Advanced data structure

- Disjoint set

Thank You

Appendix

A sub-tree of black height $\mathrm{bh}(v)$ in an RBT contains at least $2^{\mathrm{bh}(v)} - 1$ internal nodes

- Base cases ($\mathrm{height}(v) = 0$)
  - If $\mathrm{height}(v)$ is 0, then v is a leaf
  - The black height of v is 0
  - The sub-tree T_v rooted at v contains $0 = 2^{\mathrm{bh}(v)} - 1$ inner nodes
- **Inductive step**
  - Suppose v is a node with $\mathrm{height}(v) > 0$
  - v has two children with strictly smaller height
  - These children (c_1, c_2) either have $\mathrm{bh}(c_i) = \mathrm{bh}(v)$ or $\mathrm{bh}(c_i) = \mathrm{bh}(v) - 1$
  - By hypothesis both sub-trees contains at least $2^{\mathrm{bh}(v)-1} - 1$
  - Then, T_v contains at least $2\left(2^{\mathrm{bh}(v)-1} - 1\right) + 1 \geq 2^{\mathrm{bh}(v)} - 1$ [Q.E.D.]