Data structure [A11]

김종규, PhD

2017-05-15

Review: Dynamic set

- Now we could have duplicate values
- Node: A container holding a value
  - Manage extra information for the data structure
  - e.g., pointers (next in Linked List. left and right in Binary Tree)
- Operations
  - Test: Answer whether a value exists
  - Insert: Add a node
  - Delete: Delete the specified node

Review: Binary search tree

- Search time complexity $O(h)$
- Insert time complexity $O(h)$
- Delete time complexity $O(h)$
- Search and insert: recursive algorithm
  - Easily transformed to iterative algorithm

Review: Binary search tree

- Deleting a node in a BST
  - Generally, a two-step algorithm
    - find a successor node
    - transplant the subtree

$\longrightarrow$ How to keep h as low as possible?

Lowest h

- Suppose there are n nodes in a binary tree
- What is the minimum h?

$\longrightarrow$ $O(\lg n)$

Time complexity of binary search trees

- Querying: $O(h)$
- Modifying: $O(h)$
- Draw a binary search tree for 1, 2, 3, 4, 5

  $\longrightarrow$ skew
- What if we insert 3, 1, 2, 4, 5

Rotate concept

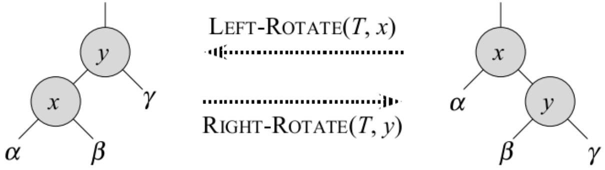

그림: Rotate concept

Balancing with rotate

- Insert 1, 2, 3
- Rotate left
- Insert 4
- Rotate left
- Insert 5

Rotate algorithm


```
LEFT-ROTATE(T, x)
y = x.right                   // set y
x.right = y.left              // turn y's left subtree into x's right subtree
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p                     // link x's parent to y
if x.p == T.nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
y.left = x                    // put x on y's left
x.p = y
```

그림: Left rotate

How to measure balancing

- h_l: Height of left subtree
- h_r: Height of right subtree
- BF = h_l - h_r
  - BF = 0 or $\pm 1 \longrightarrow$ balanced

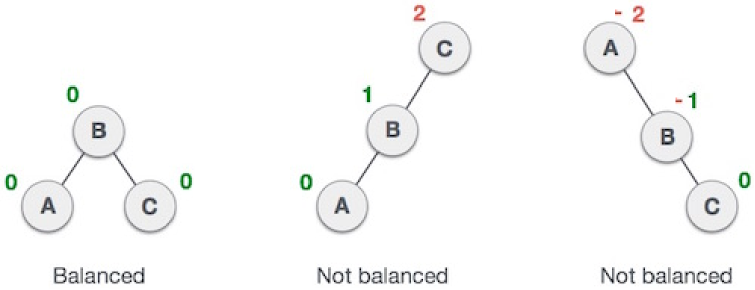

그림: Measuring balance

Balancing operation

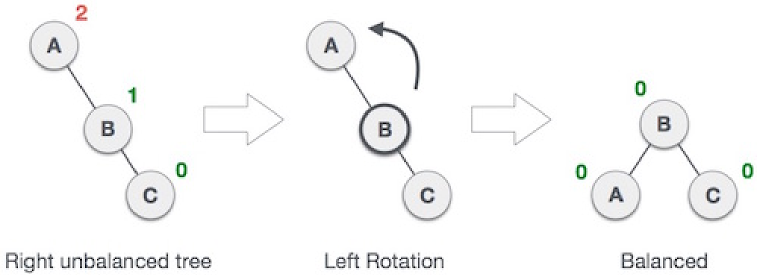

그림: Right skew

Balancing operation

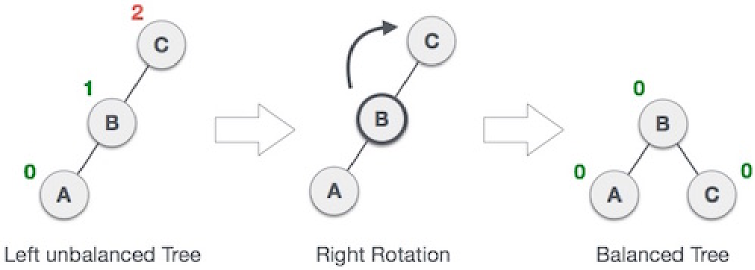

그림: Left skew

Balancing operation

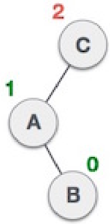

그림: Left and right

Balancing operation

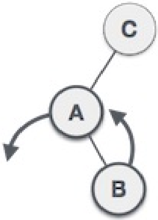

그림: Right rotation

Balancing operation

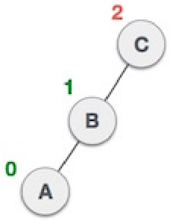

그림: Straighten – Left skew

Balancing operation

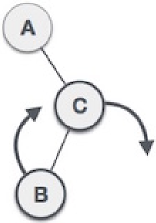

그림: Right and left

Balancing operation

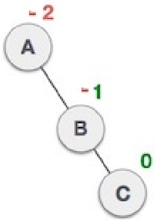

그림: Left rotation

Balancing operation

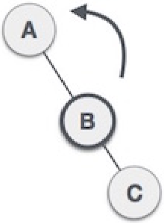

그림: Straighten – Right skew

Balanced tree

- BF is at most 1
- Search $O(h) = O(\lg n)$
- Insert $O(h) = O(\lg n)$
- Delete $O(h) = O(\lg n)$
- But
  - Every insert and delete results in rotation leading to root

$\longrightarrow$ Red-black tree

Concept: sentinel

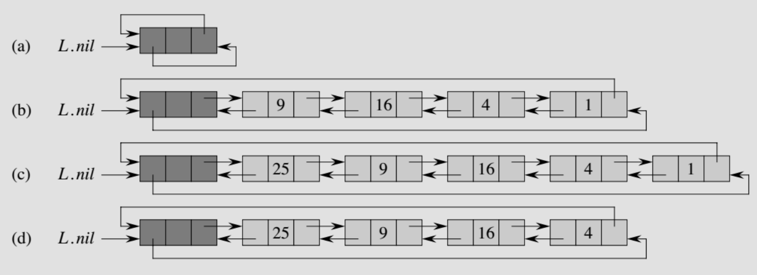

그림: Linked list with a sentinel

Balanced tree

- Every tree has a simple path from the root and all leaves
- The length is the height (h) (by definition)
- Suppose $h_{\text{max}} \leq 2h_{\text{min}}$

  $\longrightarrow$ Balanced tree
- Possible? yes
- How? $\longrightarrow$ Abstract concept: red-black property

Red-black tree in practice

- Linux `https://github.com/torvalds/linux/blob/master/lib/rbtree.c`
- Invented in 1972 and popularized circa 1992

Red-black properties

- Every node is either red or black
- The root is black
- Every leaf (including Nil) is black
- If a node is red then both its children are black
- For each node, all simple paths from the node to descendent leaves contain the same number of balck nodes

An example

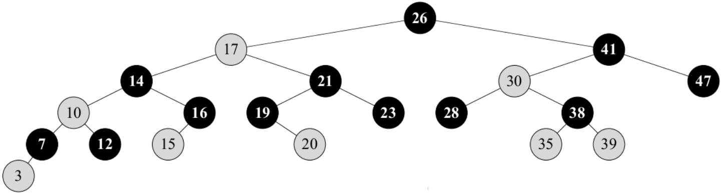

그림: An example red-black tree

김종규, PhD

Red-black tree insert

- Insert as a binary search tree
- The inserted node is red
- If red-black property is not satisfied

$\longrightarrow$ Fix it by rotating and recoloring

Property violations after insert

- Every node is either red or black $\longrightarrow$ satisfied
- The root is black $\longrightarrow$ satisfied
- Every leaf (including Nil) is black $\longrightarrow$ satisfied
- If a node is red then both its children are black $\longrightarrow$ maybe not
- For each node, all simple paths from the node to descendent leaves contain the same number of balck nodes $\longrightarrow$ satisfied

RB insert

```
RB-INSERT(T, z)
y = T.nil
x = T.root
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-INSERT-FIXUP(T, z)
```

그림: RB-Insert

RB insert fixup

```
RB-INSERT-FIXUP(T, z)
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK                    // case 1
            y.color = BLACK                      // case 1
            z.p.p.color = RED                    // case 1
            z = z.p.p                            // case 1
        else if z == z.p.right
                z = z.p                          // case 2
                LEFT-ROTATE(T, z)                // case 2
            z.p.color = BLACK                    // case 3
            z.p.p.color = RED                    // case 3
            RIGHT-ROTATE(T, z.p.p)               // case 3
    else (same as then clause
            with "right" and "left" exchanged)
T.root.color = BLACK
```

그림: RB-Insert-Fixup

How to resolve red-red violations?

- Case 1: z's uncle is red $\longrightarrow$ color change
- Case 2: z's uncle is black z is a right child $\longrightarrow$ rotate left
- Case 3: z's uncle is black z is a left child $\longrightarrow$ color change and rotate right

An example

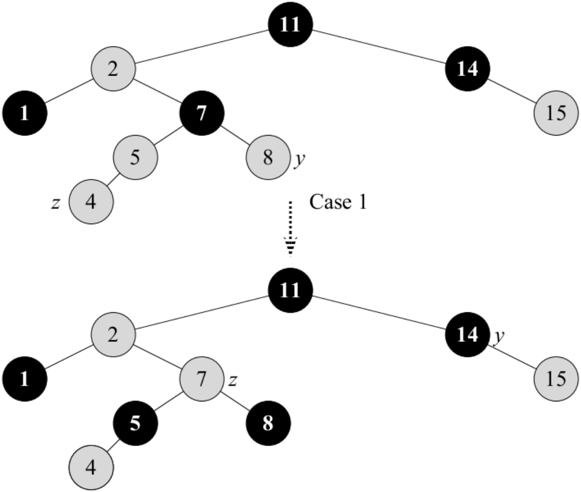

그림: Insert 4

An example

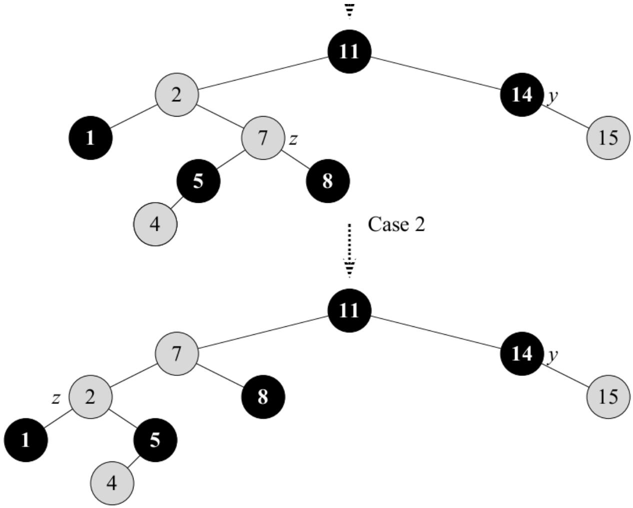

그림: Fixup 7

An example

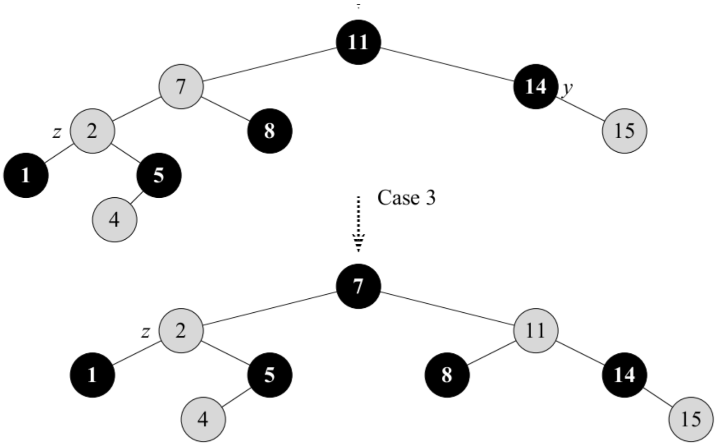

그림: Fixup 2

Case 1: Color change

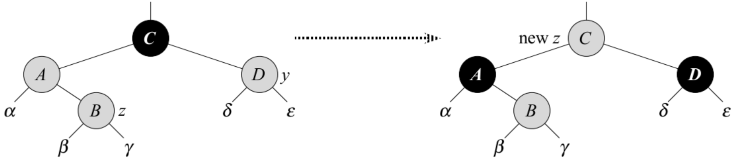

그림: Case 1

Case 1: Color change

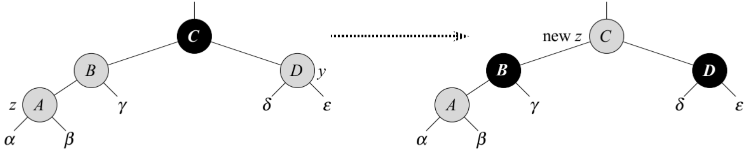

그림: Case 1

Case 2, 3: Color change and rotate right

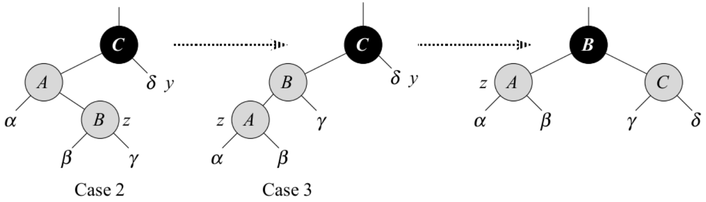

그림: Case 2, 3

Wrap-up

- Binary search tree is one of the most widely used data structure for dynamic sets
- Modifying operations of binary search tree relies on querying operations such as minimum or successor
- By maintaining the red-black tree property, we could keep a binary search tree blanced
- We will learn how to delete red-black tree