Data Structures

10. Binary Search Trees

10. Binary Search Trees

1. Binary Search Trees
2. Balanced binary search trees : AVL Tree

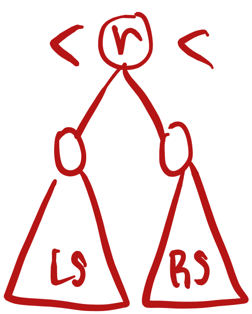
Binary Search Tree 이진탐색트리

- Heap
  - Provides a good performance of O(___$\log_2 n$___) only when handling the root
  - For searching other item than the root, it takes O(___n___) time
- Why binary search tree is necessary
  - requires O(   ) for any item, which is proportional to tree height (h)

Binary Search Tree

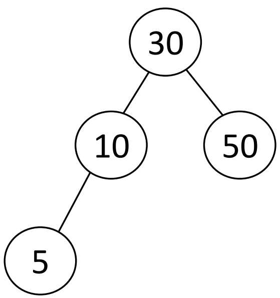
- Assume “no overlapping elements” in Tree

- The root > left sub-tree nodes
- The root < right sub-tree nodes
- Left & right sub-trees also must be binary search tree

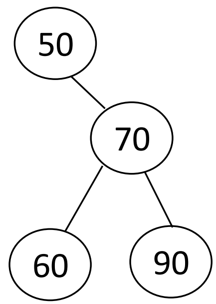


Binary Search Tree

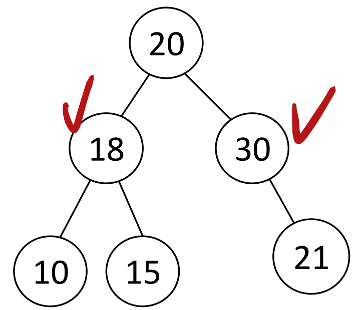
- left sub-tree < node < right sub-tree



(O)

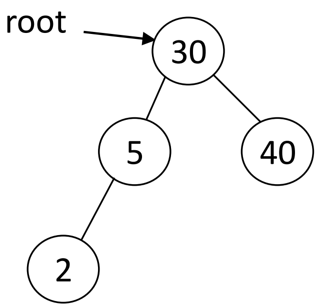


(O)



(X)

Binary Search Tree



- Search a key in BST
  - Recursive & iterative versions

```
tree_ptr rBST(tree_ptr root, int key)
{
  if (!root) return NULL;
  if (key == root->data) return root;
  if (key < root->data)
    return rBST(root->left, key);

  return rBST(root->right, key);
}
```

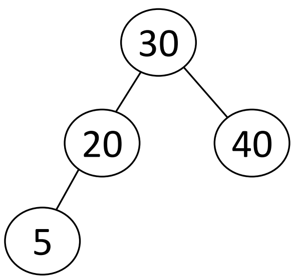
$O(\log_2 n)$ + recursion overhead

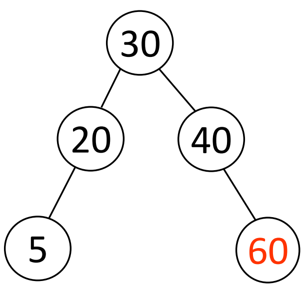
```
tree_ptr  iBST (tree_ptr tree, int key) {
  while (tree) {
    if (key == tree->data)
      return tree;
    if (key < tree->data)
      tree = tree->left;
    else
      tree = tree->right;
  }
  return  NULL;
}
```

$O(\log_2 n)$

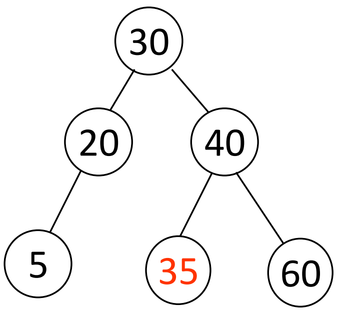
Binary Search Tree

- Insertion Algorithm
  - Before inserting a node, it should confirm whether there is no overlapping node. If overlapped, insertion fails
  - Otherwise, attach a new node (35) to the last item compared (40)





Insert 60



Insert 35

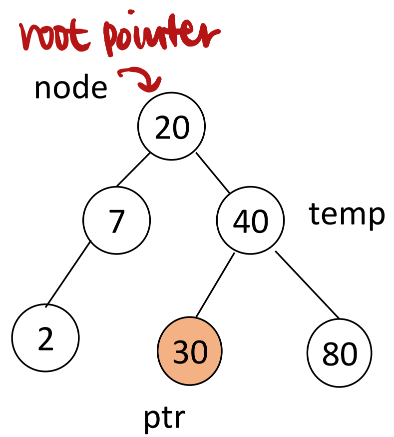
Binary Search Tree

- $\boxed{\text{Overlap\_check()}}$
  - If tree is empty, or any overlapped node exists, NULL is returned
  - Otherwise, a node searched lastly will be returned

- Time complexity : O(__log₂n__)
  - search : O(_log₂n_)
  - insert : O(__1__)

Binary Search Tree

```
void insert_node(tree_ptr *node, int num)
{
   tree_ptr ptr, temp = overlap_check(*node, num); //lastly found node
   if (temp || !(*node)) {                          // no overlap, or empty tree
      ptr = (tree_ptr) malloc (sizeof(node));
      if (IS_FULL(ptr)) {
         printf("The memory is full \n");    exit(1);
      }
      ptr->data = num;
      ptr->left = ptr->right = NULL;
      if (*node) {
         if(num < temp->data)
            temp->left = ptr;
         else
            temp->right = ptr;
      } else                          // empty tree
          *node = ptr;
   }
}
```

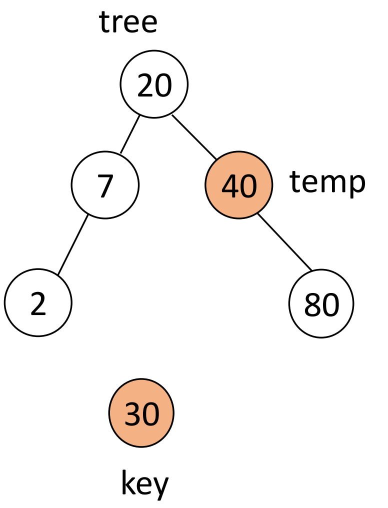


Binary Search Tree

```
tree_ptr overlap_check(tree_ptr tree, int key)
{
   tree_ptr temp = tree;

   while (tree)
   {
      temp = ___tree___;
      if (key == tree->data)    //overlapped
         return NULL;
      if (key < tree->data)
         tree = tree->left_child;
      else
         tree = tree->right_child;
   }

  return temp;
}
```

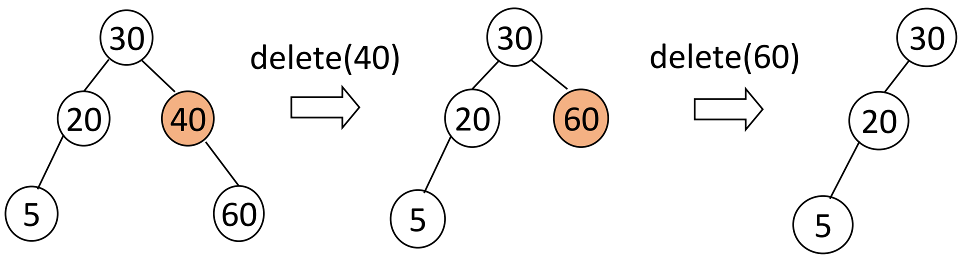


Binary Search Tree

- Deletion Algorithm
  1. - deleting a leaf node
       - Assign NULL to the parent's link
  2. - deleting a node with one child
       - Assign its child to the parent's link
  3. - deleting a non-leaf node with two children
       - choose either the _max_ node in left sub-tree or the _min_ node in right sub-tree which can reduce the tree height
       - substitutes chosen node for the node to be deleted
       - Rebuild the sub-tree for BST

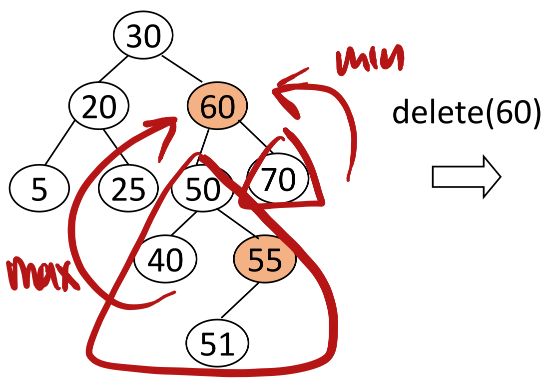
- Performance => $O(\log_2 n)$

Binary Search Tree

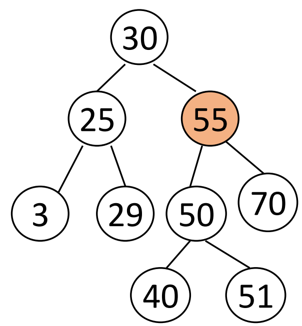


one child

leaf node



two children



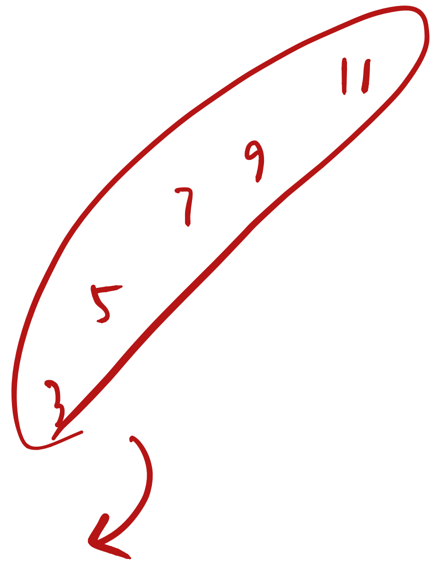
rebuilding

Binary Search Tree

- Time complexity

- Average case : $O(\log_2 n)$

- Worst case : skewed tree => O(_n_)



- Best case : balanced binary search tree => $O(\log_2 n)$

Balance Binary Search Tree 균형이진탐색트리

- Height of binary search tree : n
  - Insertion, deletion can be O(n) in the worst case
- Good to keep a tree height small
- Minimum height of a binary tree with n nodes : $O(\log_2 n)$

- Goal
  - keep the height of a binary search tree O($\underline{\log_2 n}$)

- Balanced binary search trees
  - AVL tree, 2-3-4 tree, red-black tree

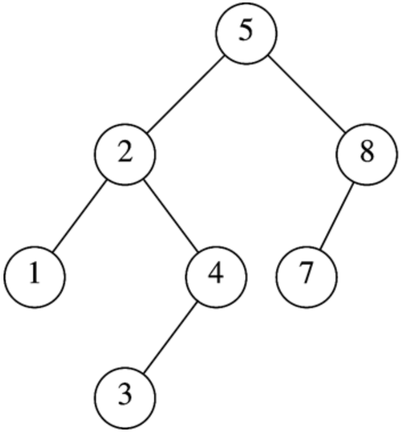
Balanced Tree?

- Suggestion
  - every node must have left and right subtrees of the same height
  - Hard to satisfy this except for complete full binary trees

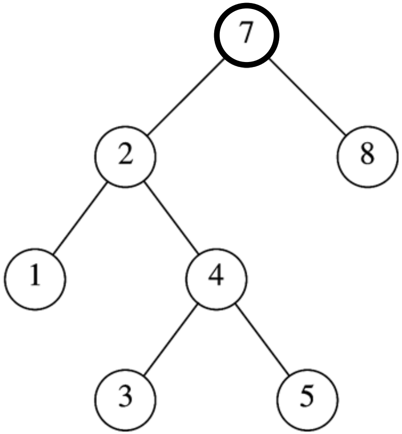
- Our choice
  - for each node, the height of the left and right subtrees can differ at most 1

AVL Tree

- Adelson-Velskii and Landis,1962
- AVL tree is a binary search tree in which
  - for every node in the tree, the height of the left and right sub-trees differ by at most 1



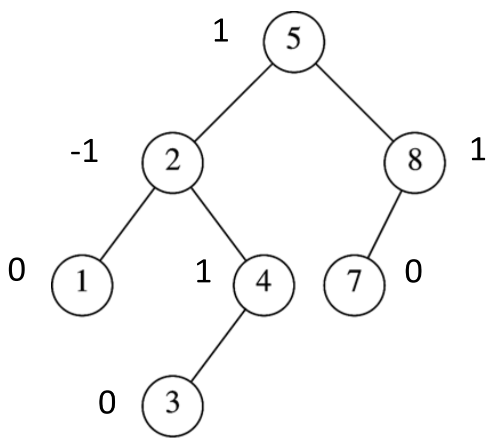
AVL tree



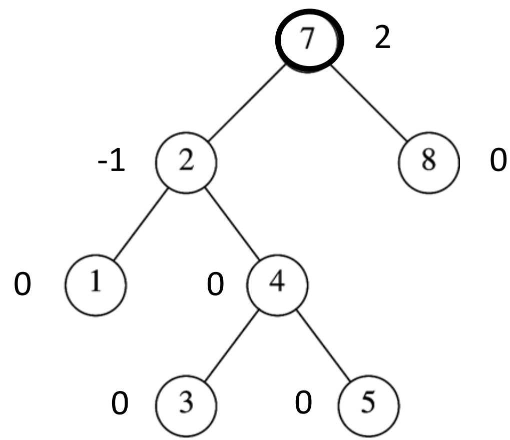
AVL property violated at ___

Balance factor

- Balance factor (BF) of a node 균형인수
  = Height (left subtree of the node) – Height (right subtree)
- AVL tree : BF of all node should be 1, 0 아 -1 ______________



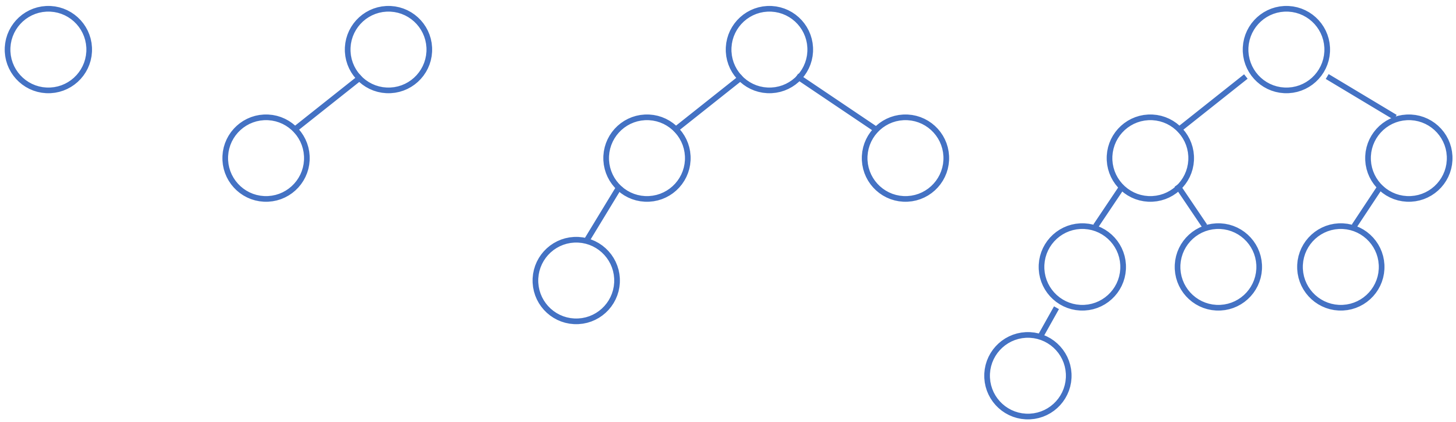
AVL tree



AVL property
violated here

AVL Tree with Min Number of Nodes

N_h = minimum # of nodes in a AVL tree of height h

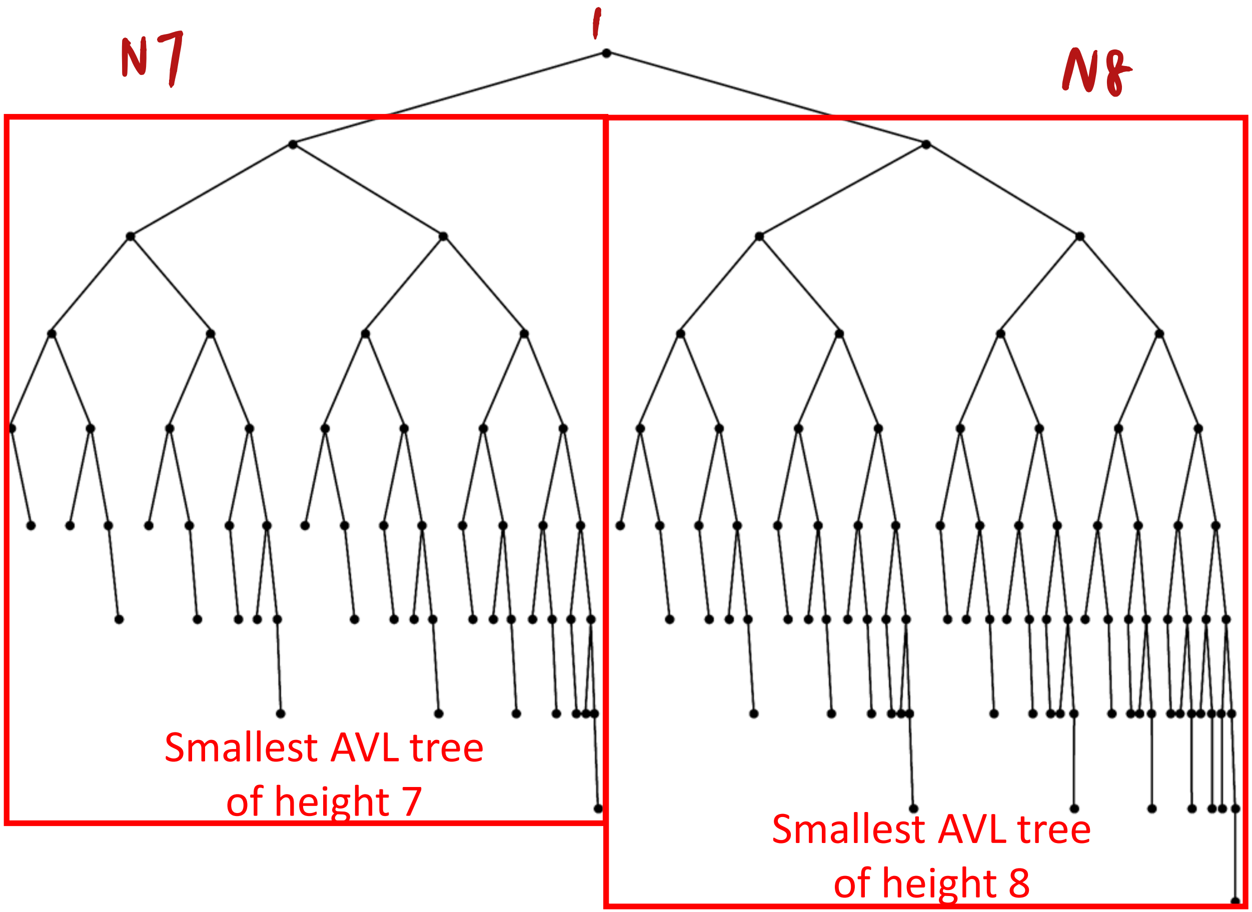


$N_1 = 1$ $N_2 = 2$ $N_3 = 4$ $N_4 = N_3 + N_2 + 1 = 7$

$N_0 = 0$

$$N_h = N_{h-1} + N_{h-2} + 1$$

Thus, searching on an AVL tree will take O($\log_2 n$) time

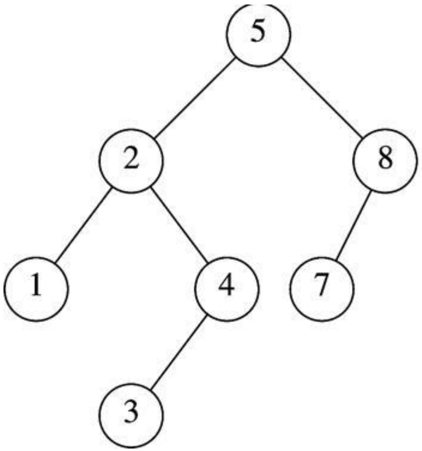


Smallest AVL tree of height 9 N9

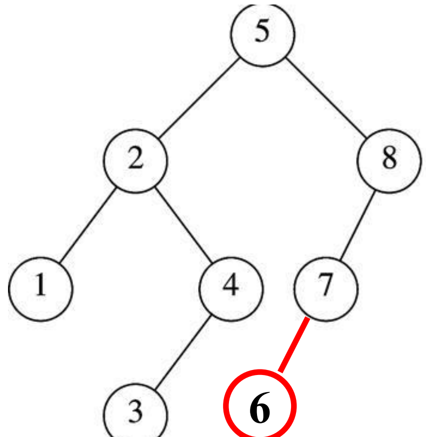
$$N_h = N_{h-1} + N_{h-2} + 1$$

Insertion in AVL Tree

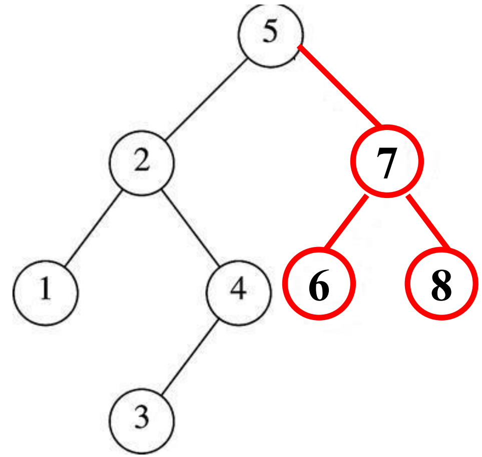
- Basically follows insertion strategy of binary search tree
  - But may cause violation of AVL tree property
- Restore the destroyed balance condition if needed



Original AVL tree



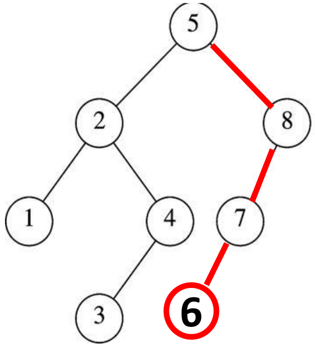
Insert 6
Property violated



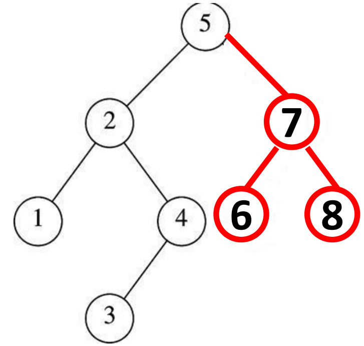
Restore AVL property

Insertion in AVL Tree

- After an insertion, only nodes that are on the path from the insertion point to the root might have their balance altered
  - Because only those nodes have their subtrees altered
- Rebalance the tree at the deepest, such node guarantees that the entire tree satisfies the AVL property



Node 5, 8, 7 might
have balance altered



Rebalance node 7
guarantees the whole tree be AVL

Cases for Rebalance

- Denote the node that must be rebalanced α
  1. ______ Case : an insertion into the left subtree of the left child of α
  2. ______ Case : an insertion into the right subtree of the left child of α
  3. ______ Case : an insertion into the left subtree of the right child of α
  4. ______ Case : an insertion into the right subtree of the right child of α

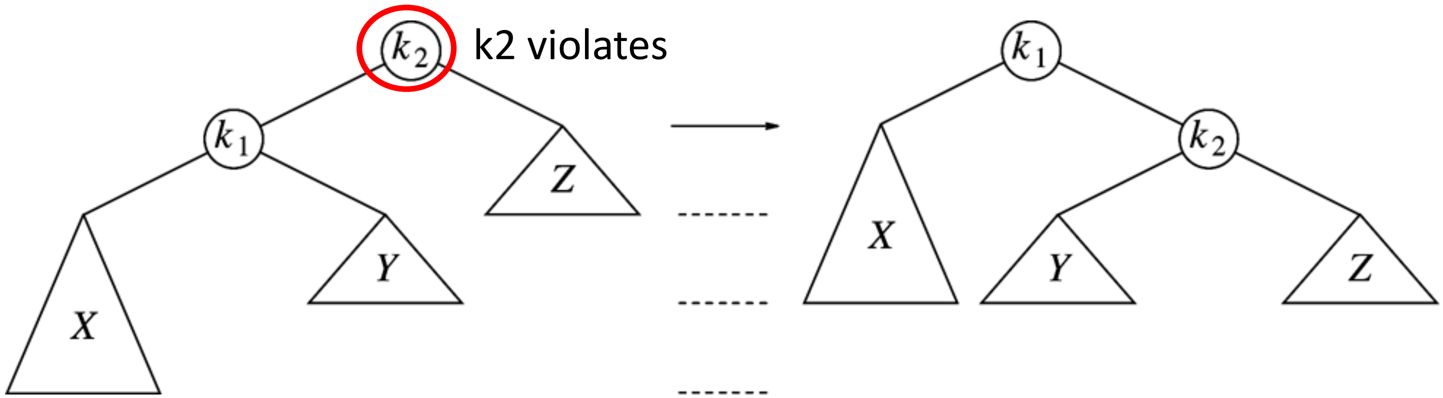
Rebalance of AVL Tree

- Rebalance of AVL tree are done with simple modification to tree, known as rotation
- Insertion occurs on the "outside"
  - left-left or right-right cases
  - Is fixed by single rotation of the tree
- Insertion occurs on the "inside"
  - left-right or right-left cases
  - is fixed by double rotation of the tree

Insertion Algorithm

- First, insert a new key as a new leaf just as in ordinary binary search tree
- Check BF of each node in the path between a new node (N) and   the root.
  - If BF is OK(0, -1, +1), proceed to parent(x)
  - If not, restructure it by doing either a single rotation or a double rotat ion
- Note
  - once we perform a rotation at a node x, we won’t need to perform an y rotation at any ancestor of x.

Single Rotation (Left-Left Insertion)

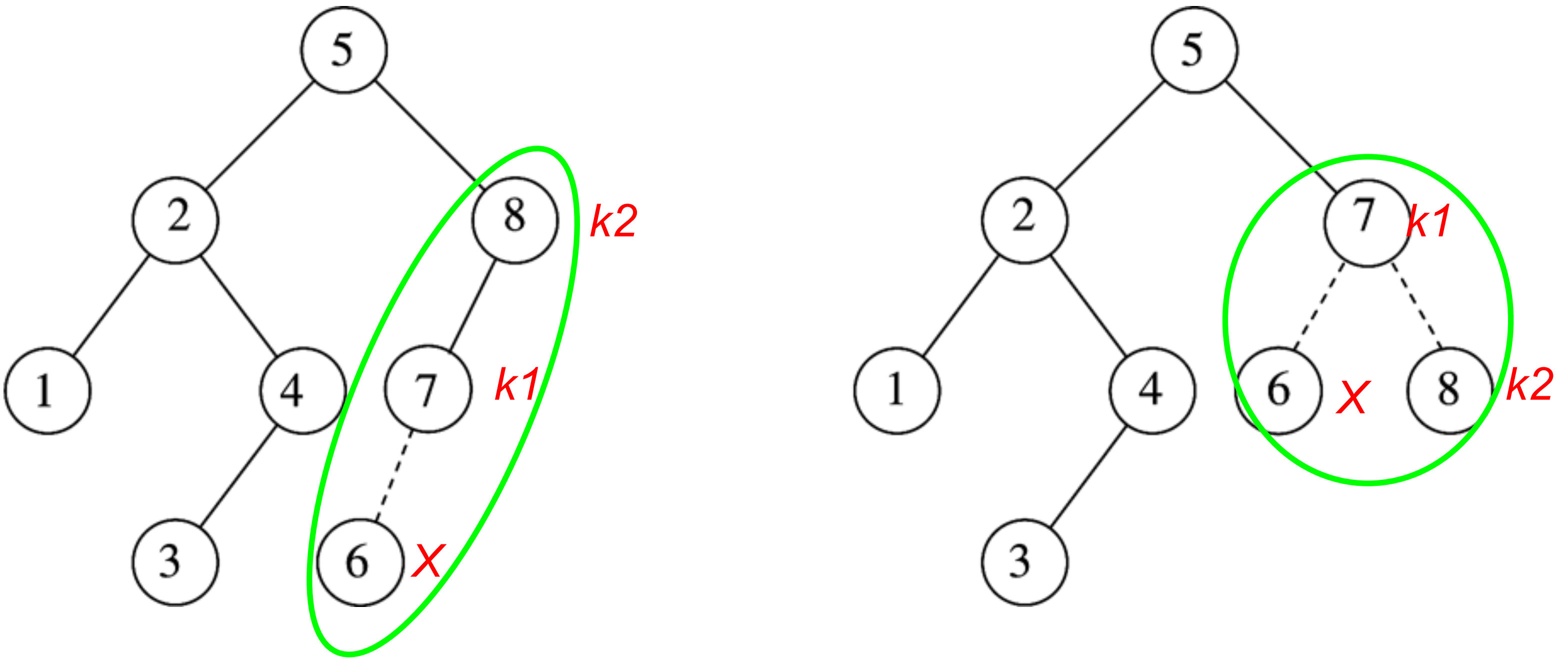


An insertion in subtree X,

AVL property violated at node k2

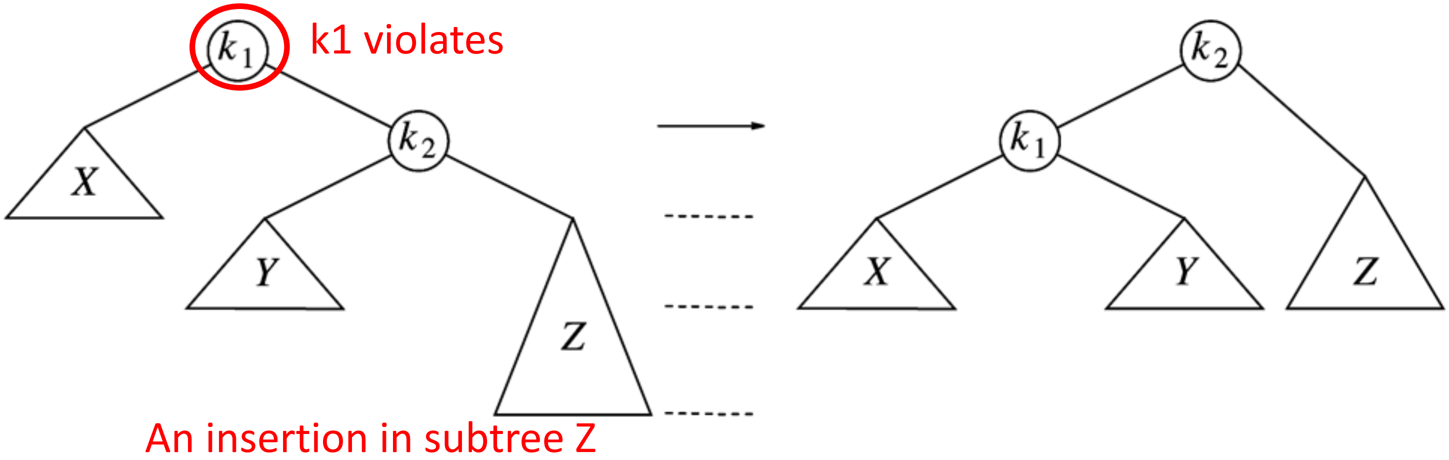
Solution: single rotation

Single Rotation (Left-Left insertion)



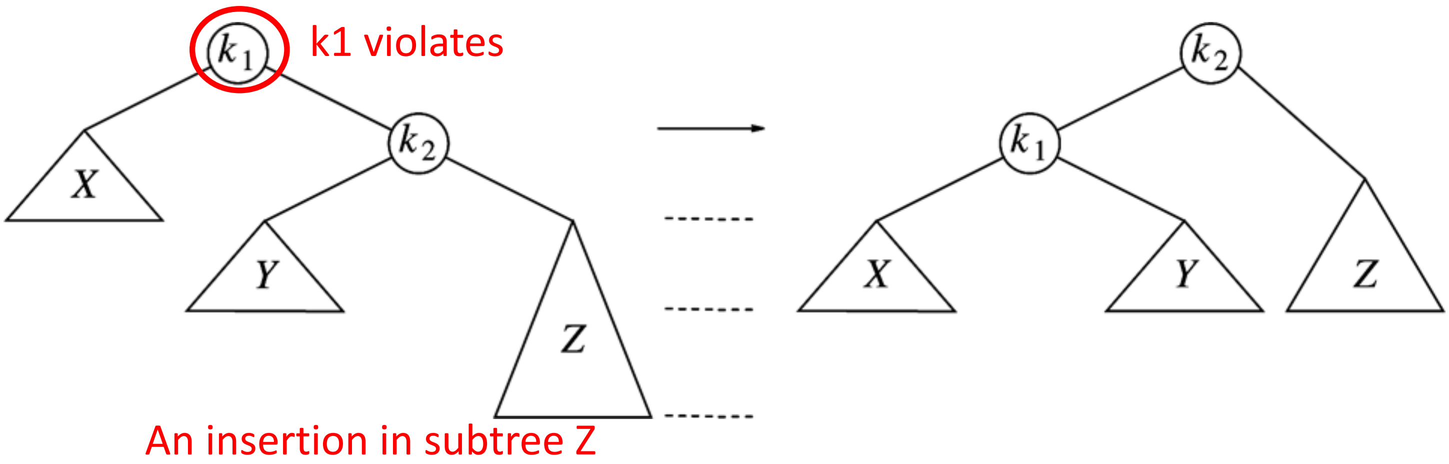
Single Rotation (Right-Right insertion)

- Case 4 is a symmetric case to case 1
- Insertion takes O(_______________) time, Single rotation takes O(1) time



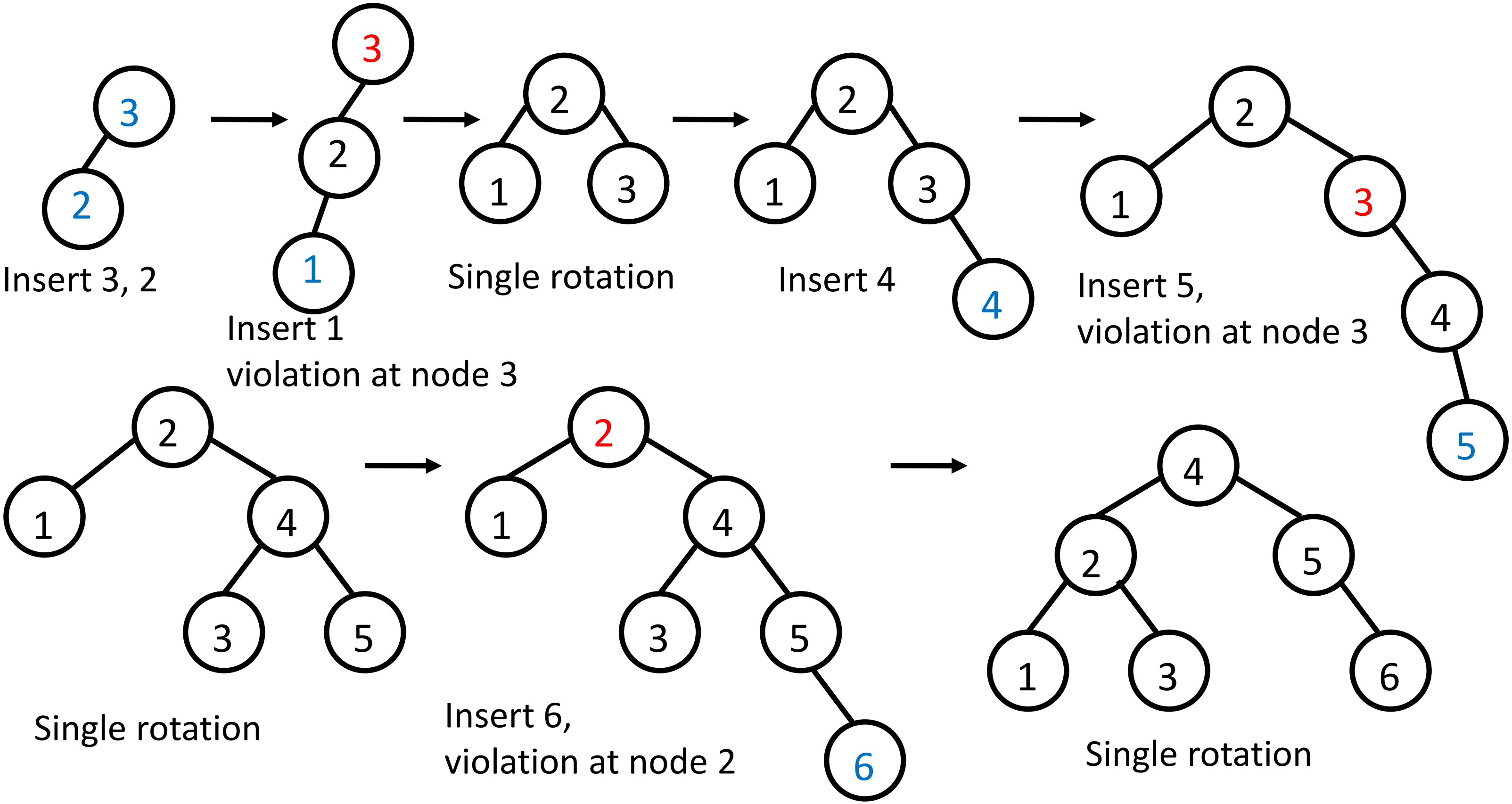
Single Rotation (Right-Right insertion)

- right-right insertion

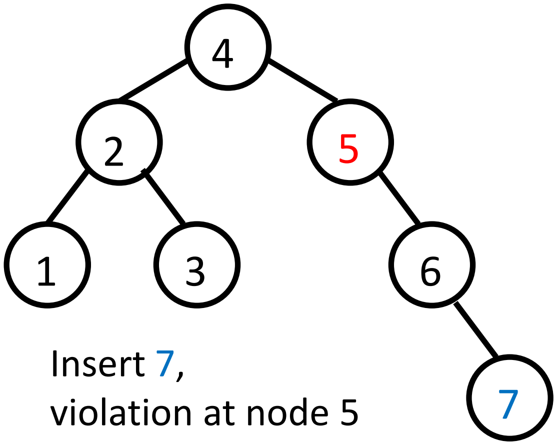


AVL Tree Construction

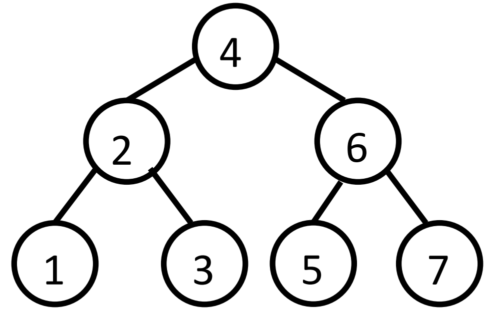
- Sequentially insert 3, 2, 1, 4, 5, 6 to an AVL Tree



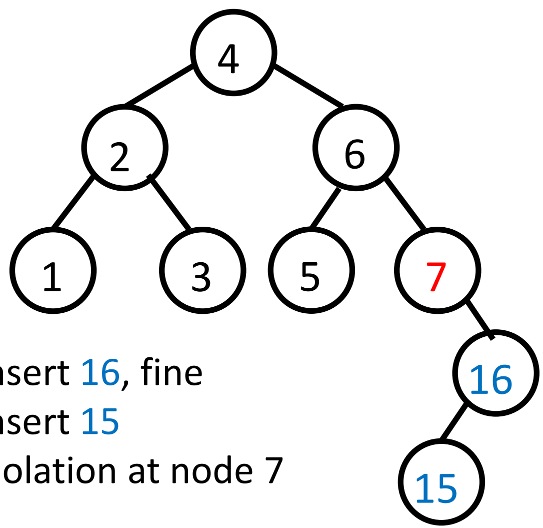
AVL Tree Construction



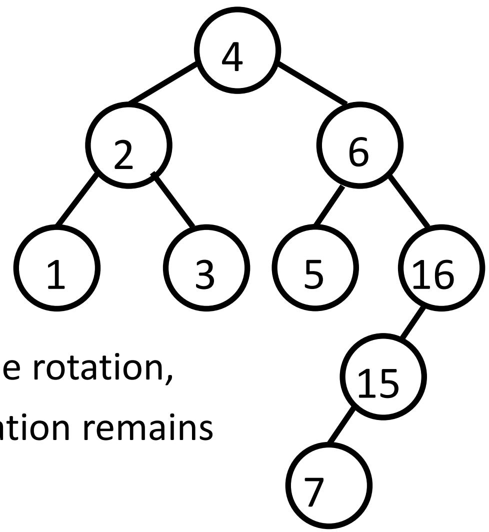
Insert 7,
violation at node 5



Single rotation



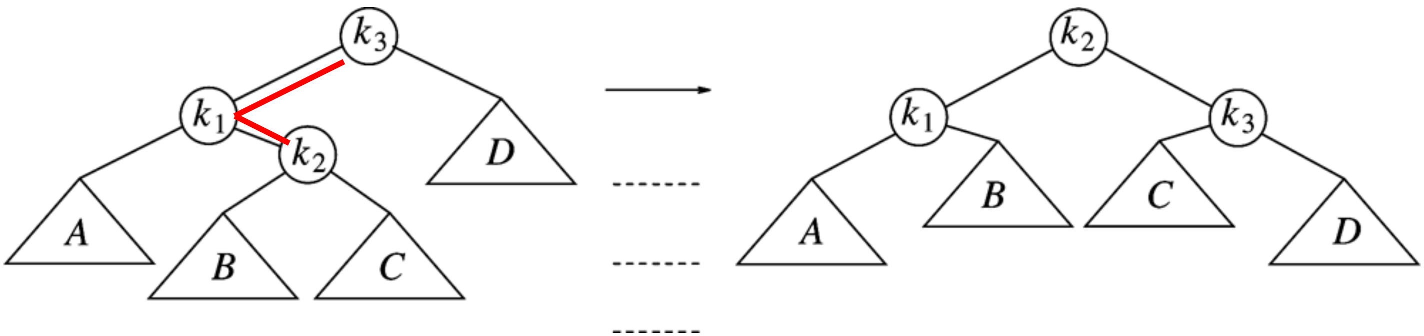
Insert 16, fine
Insert 15
violation at node 7



Single rotation,
Violation remains

Double Rotation (Left-Right insertion)

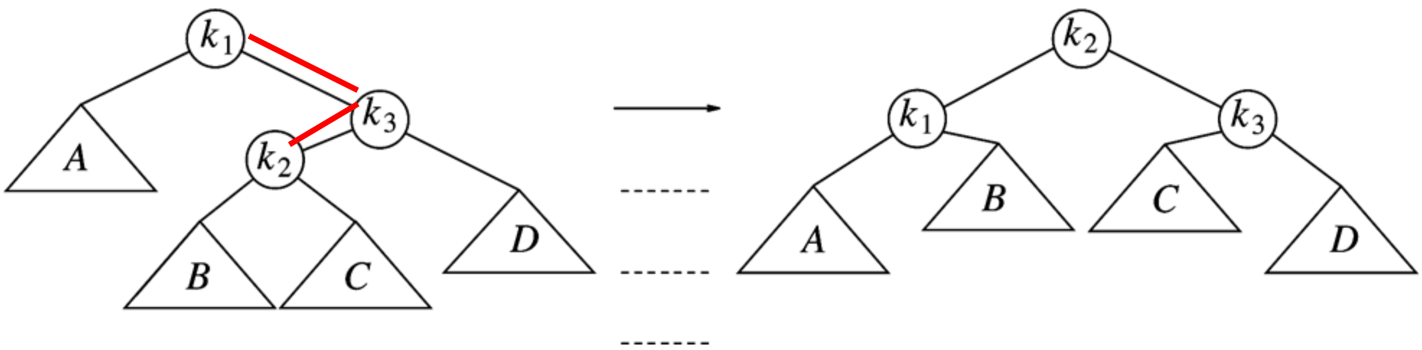
- Facts
  - A new key is inserted in the subtree B or C
  - The AVL-property is violated at ___
  - k_3-k_1-k_2 forms a zig-zag shape
- Solution
  - The only alternative is to place ____ as the new root



Double rotation to fix case 2

Double Rotation (Right-Left insertion)

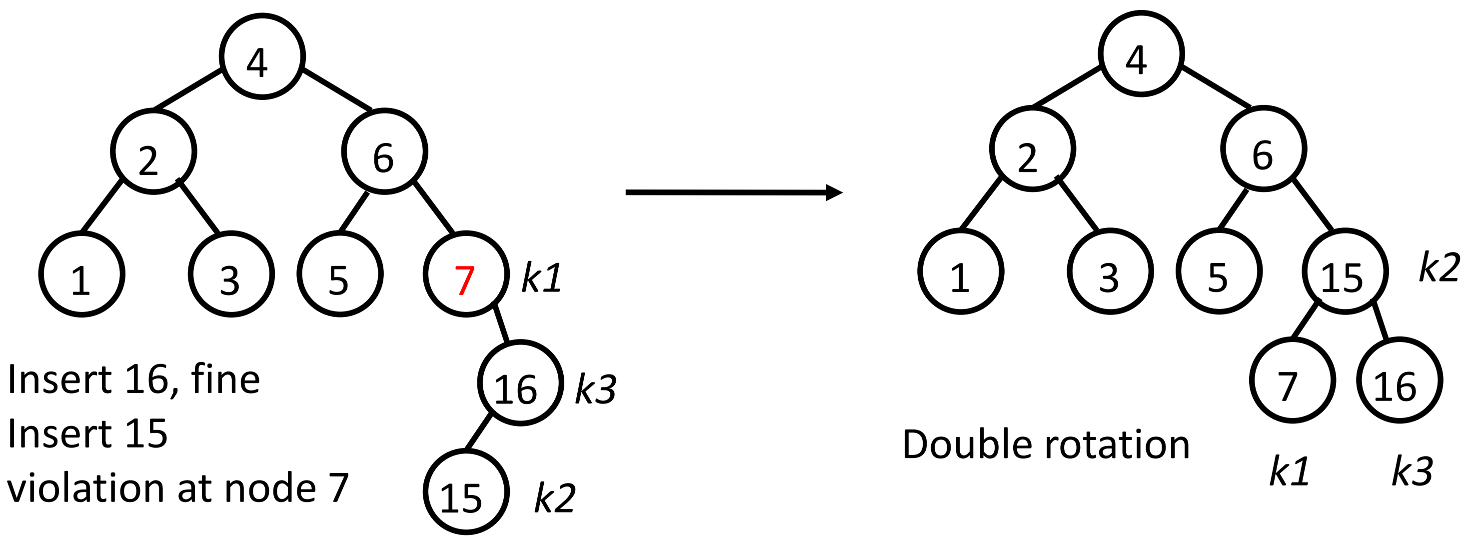
- Facts
  - The new key is inserted in the subtree B or C
  - The AVL-property is violated at ____
  - k_1-k_3-k_2 forms a zig-zag shape
- Case 3 is a symmetric case to case 2

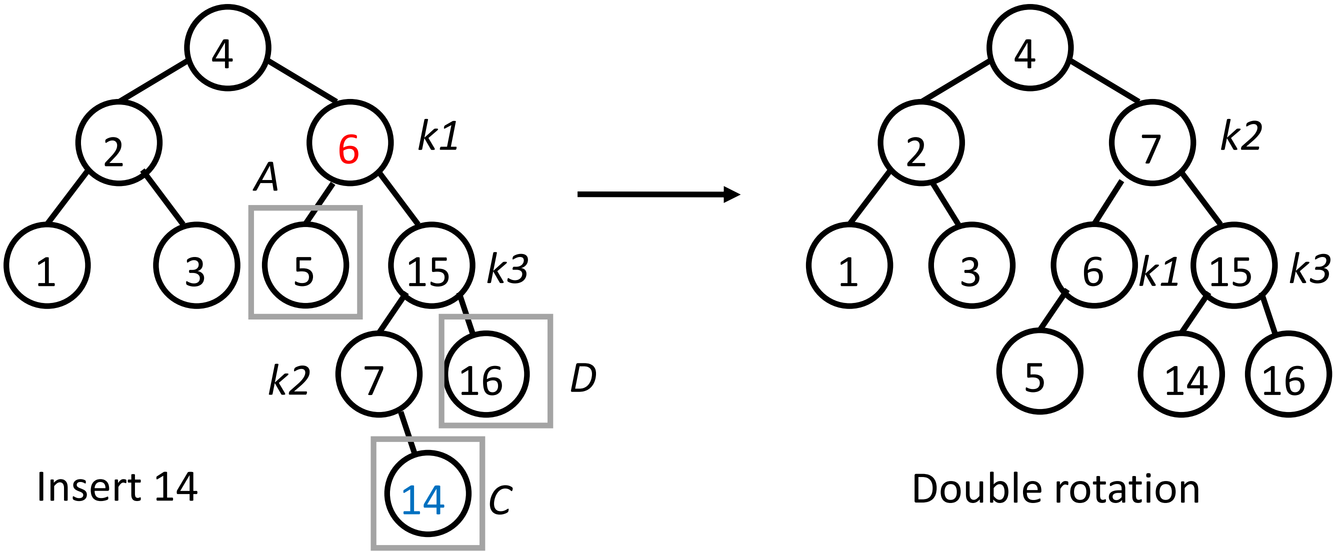


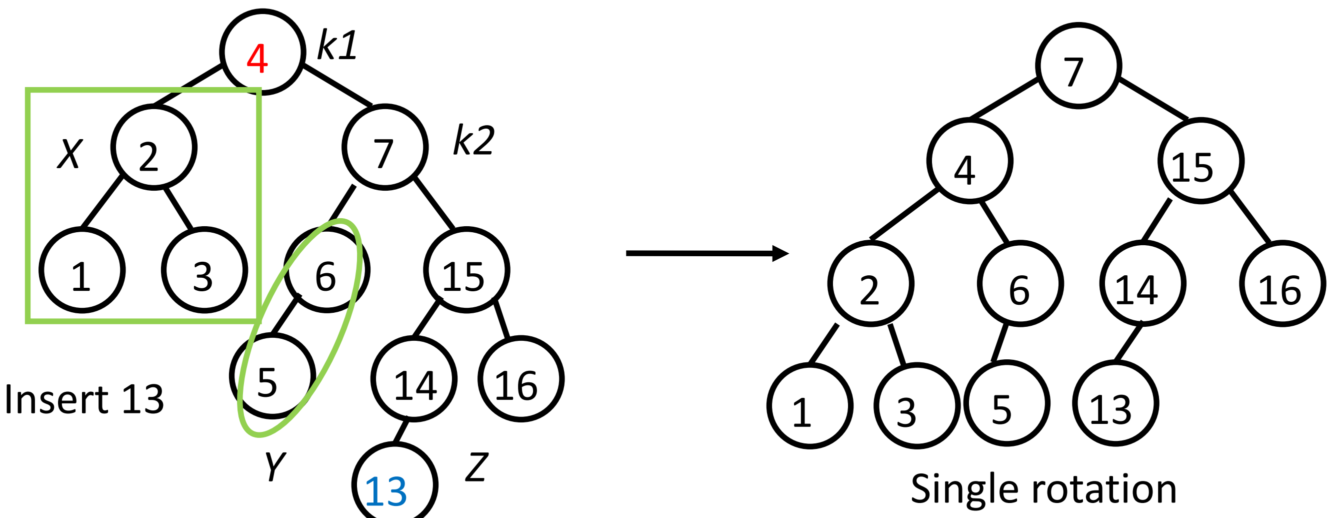
Double rotation to fix case 3

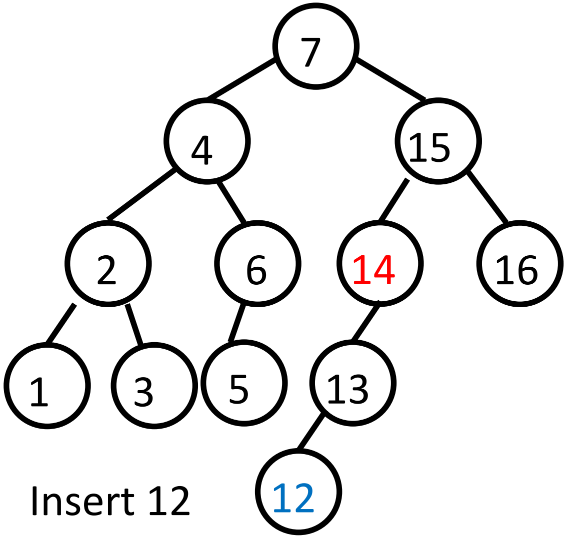
AVL Tree Construction

- continue to insert 15, 14, 13, 12, 11, 10, 8, 9

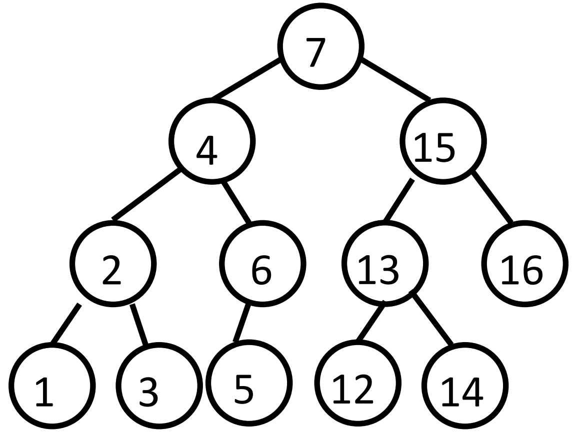



4
2
6
k1
A
1
3
5
15
k3
k2
7
16
D
14
C
Insert 14
4
2
7
k2
1
3
6
k1
15
k3
5
14
16
Double rotation

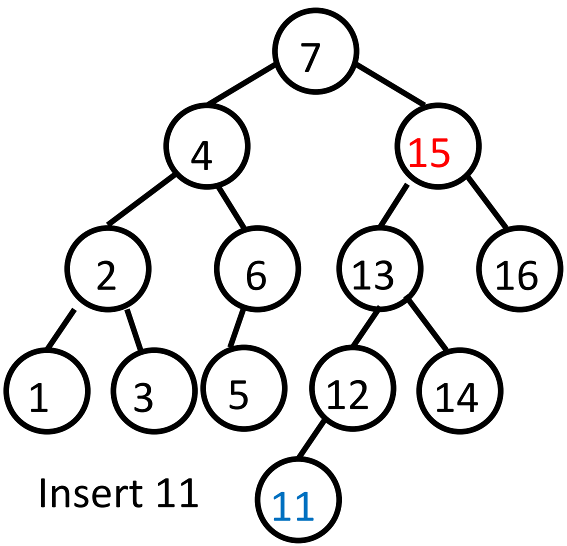

4
k1
X
2
7
k2
1
3
6
15
5
14
16
Insert 13
Y
13
Z
7
4
15
2
6
14
16
1
3
5
13
Single rotation



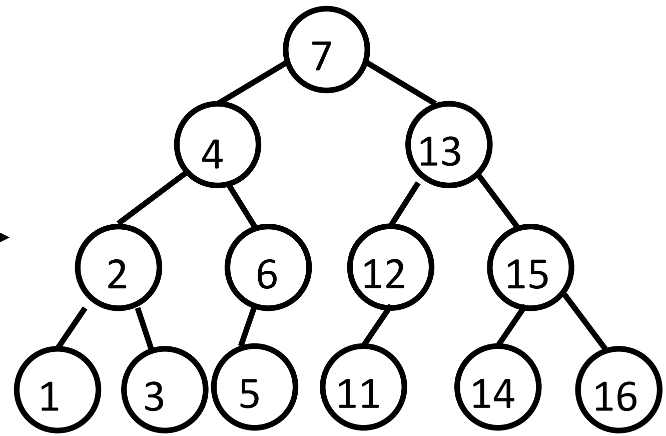
Insert 12



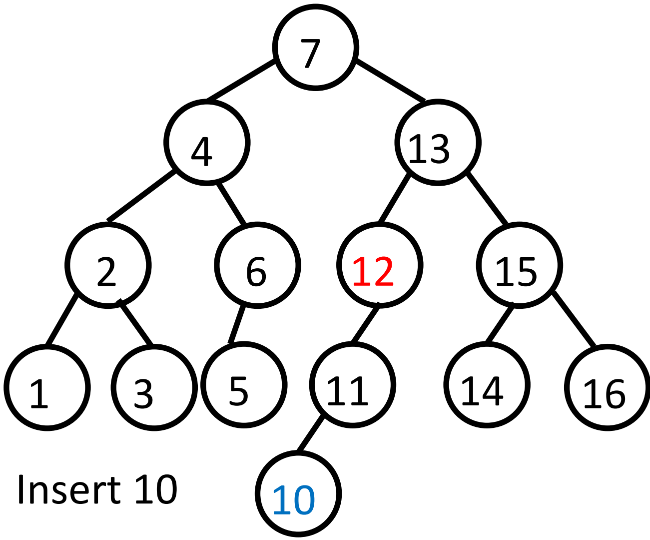
Single rotation



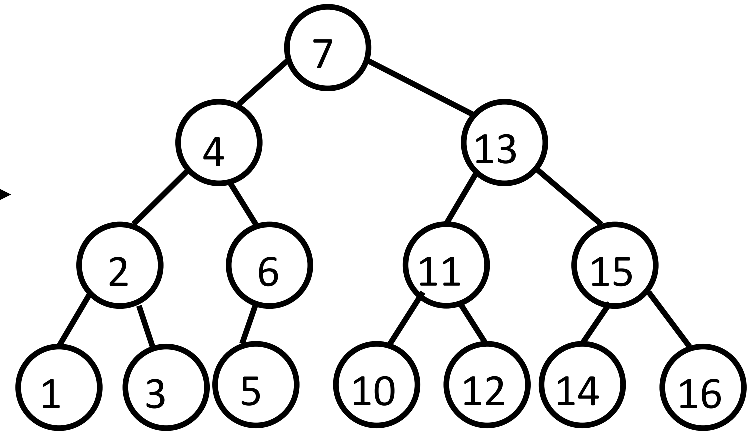
Insert 11



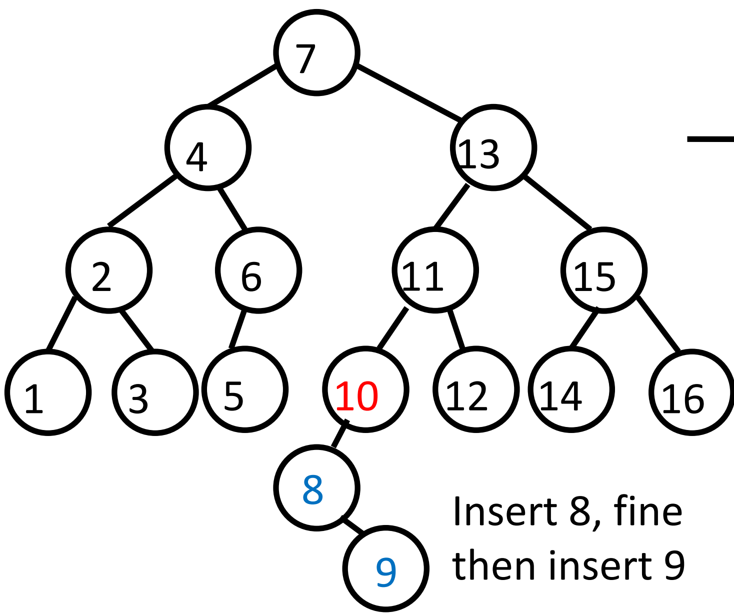
Single rotation



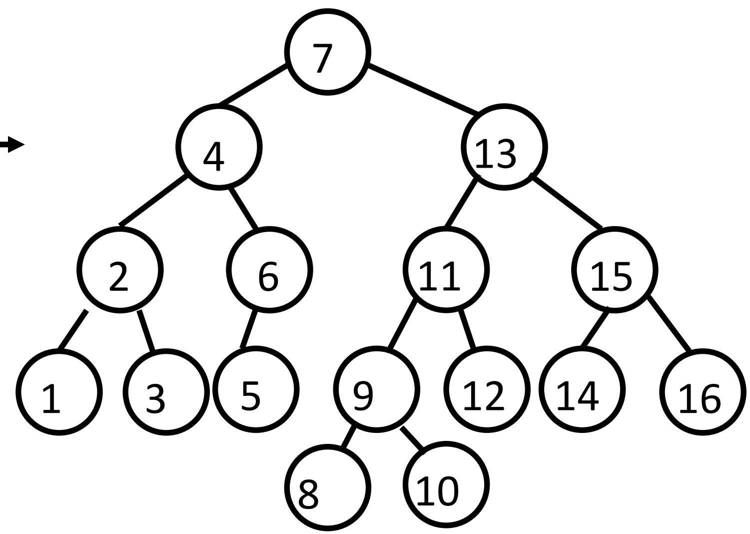
Insert 10



Single rotation



Insert 8, fine
then insert 9



Double rotation

5/19(수)

B-Trees

Motivation for B-Trees

- Index structures for large datasets cannot be stored in main memory
- Storing it on disk requires different approach to efficiency
- Assuming that a disk spins at 7200 RPM, one revolution occurs in 1/120 of a second, or 8.3 ms
- Crudely speaking, one disk access takes about the same time as 200,000 instructions

Motivation (cont.)

- Assume that we use an AVL tree to store about 20 million records
- We end up with a very deep binary tree with lots of different disk accesses; $\log_2 20{,}000{,}000$ is about 24, so this takes about __0.2__ seconds
- We know we can't improve on the log *n* lower bound on search for a binary tree
- But, the solution is to use more branches and thus reduce the __height__ of the tree! (자식이 최대 2개로 고정돼있으므로)
  - As __branching__ increases, __depth__ decreases

→ 여러개의 자식을 가질수 있게하여 높이를 줄여야한다.

Comparing Trees

- Binary trees
  - Can become *unbalanced* and *lose* their good time complexity (O(__$\log_2 n$__))
  - AVL trees are strict binary trees that *overcome the balance problem*
  - Heaps remain balanced but only good to get the root(max/min)

- Multi-way trees 다원트리
  - B-Trees can be *m*-way, they can have any number of children

m원 B-tree

Definition : m-way B-Tree

- DEF: A B-Tree of order m is an m-way search tree that either is empty or satisfies the following properties

(1) The root node has at least two children or a leaf

(2) All nodes other than the root node and external nodes have at most m and at least ceil(m/2) children

내부

(3) All external nodes are at the same level

(4) The number of keys is one less than the number of children for non-leaf nodes and at most m-1 and at least ceil(m/2)-1 for leaf nodes

M-way B tree

- m = 3 (__2-3__ Tree) , min children = ceil(3/2)
  - Degree(internal) = 2 or 3, Degree(root) = 0, 2, 3
  - # keys in a leaf node = 1, 2
- m = 4 (_2-3-4__ tree) , min children = ceil(4/2)
  - Degree(internal) = 2, 3, 4, Degree(root) = 0, 2, 3, 4
  - # keys in a leaf node = 1, 2, 3
- m = 5, min children = ceil (5/2)
  - Degree(internal) = 3, 4, 5, Degree(root) = 0, 2, 3, 4, 5
  - # keys in a node = 2, 3, 4

Entries in B-trees of Various orders

Order	Number of Subtrees		Number of Entries	
	Min	Max	Min	Max
3	2	3	1	2
4	2	4	1	3
5	3	5	2	4
6	3	6	2	5
…	…	…	…	…
m	ceil(m/2)	m	ceil(m/2) -1	m-1

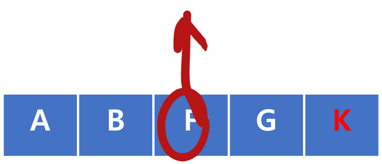
Creating a B-tree of order 5

- A G F B K D H M J E S I R X C L N T U P

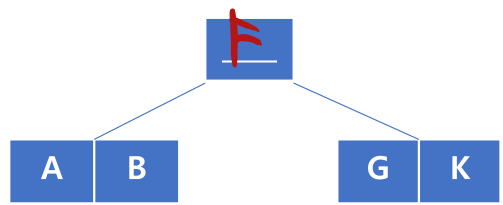
Insert into 5-way B-tree

- A G F B K D H M J E S I R X C L N T U P

- A G F B **K** D H M J E S I R X C L N T U P



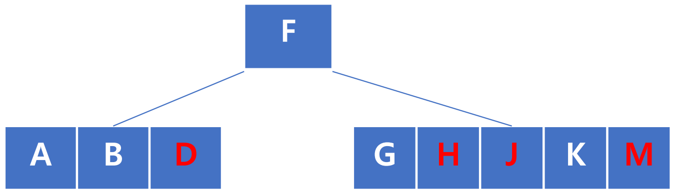
overflow

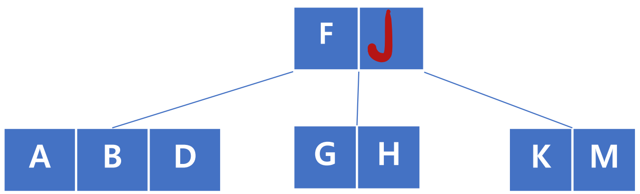


Split

Insert into 5-way B-tree

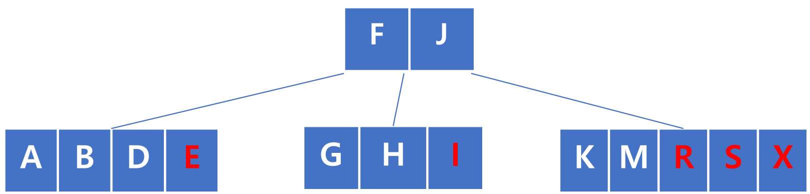
- A G F B K **D H M J** E S I R X C L N T U P

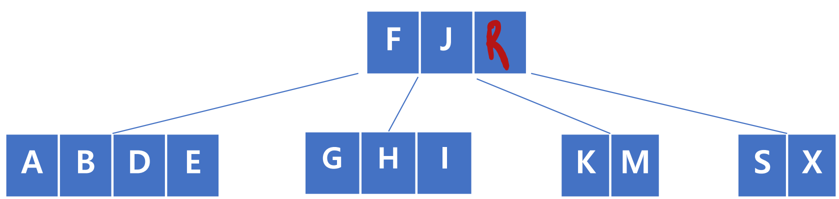




Insert into 5-way B-tree

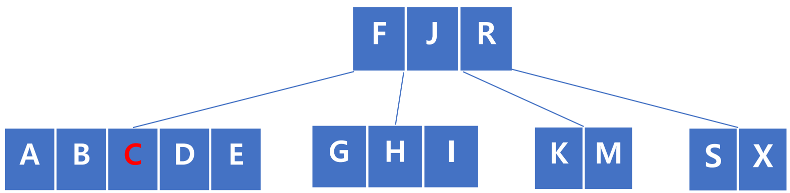
- A G F B K D H M J **E S I R X** C L N T U P

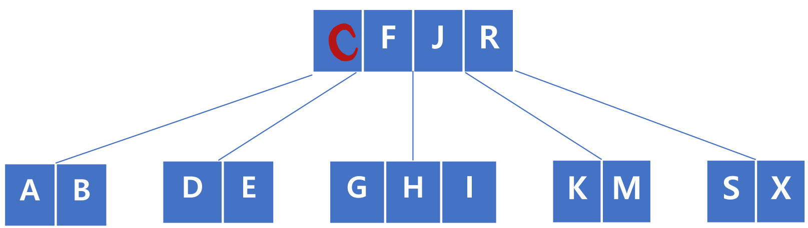




Insert into 5-way B-tree

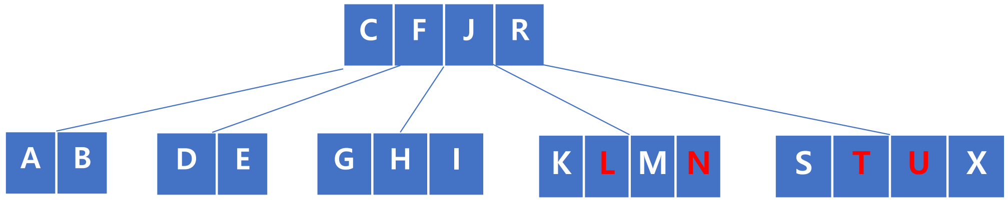
- A G F B K D H M J E S I R X **C** L N T U P



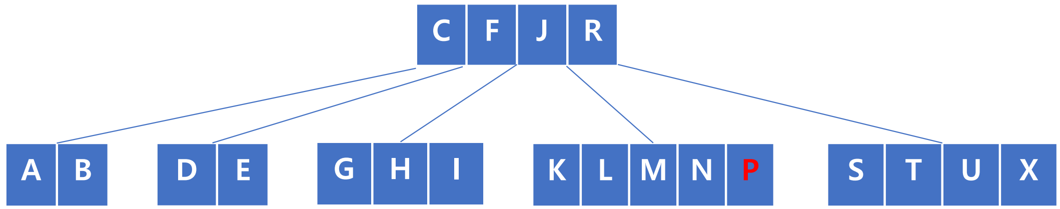


Insert into 5-way B-tree

- A G F B K D H M J E S I R X C **L N T U** P

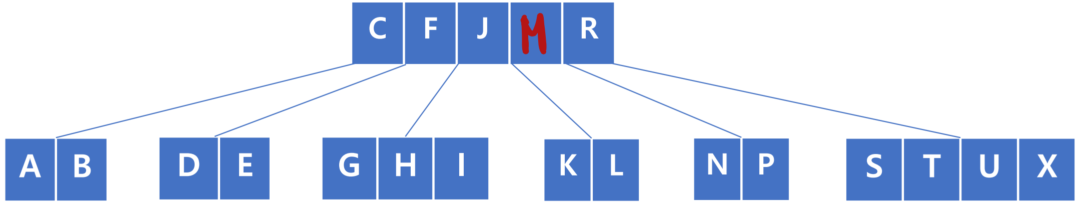


- A G F B K D H M J E S I R X C L N T U **P**

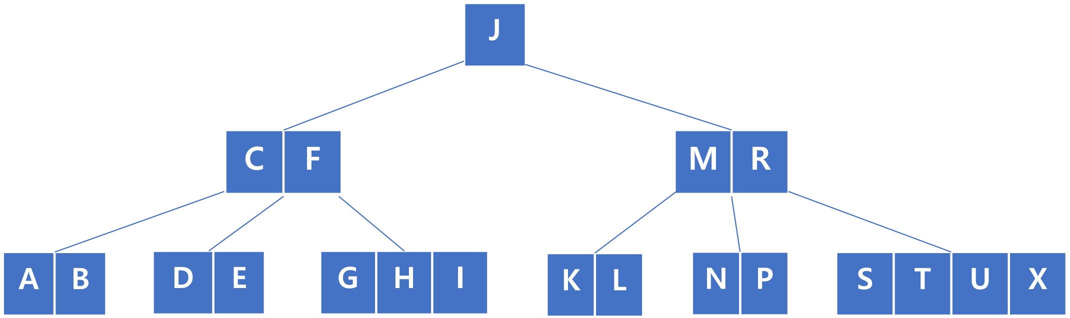


Final

- A G F B K D H M J E S I R X C L N T U P

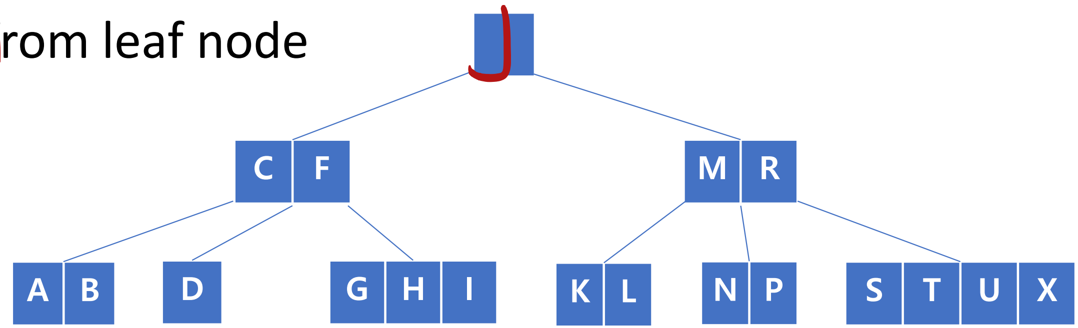


- Final



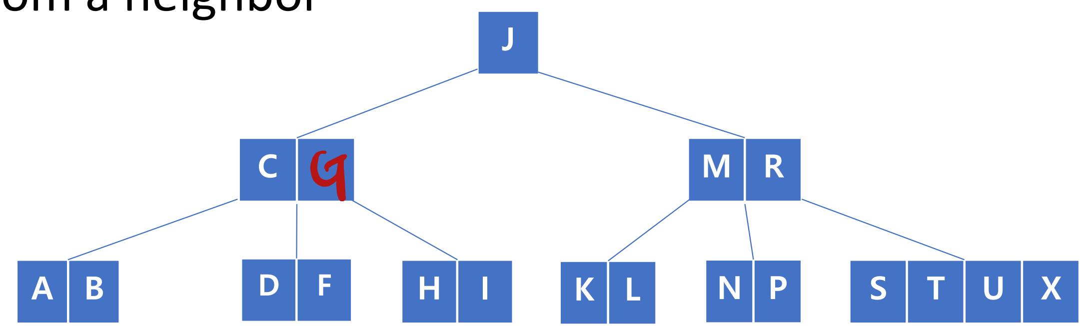
Delete

- Delete E from leaf node



↳ underflow

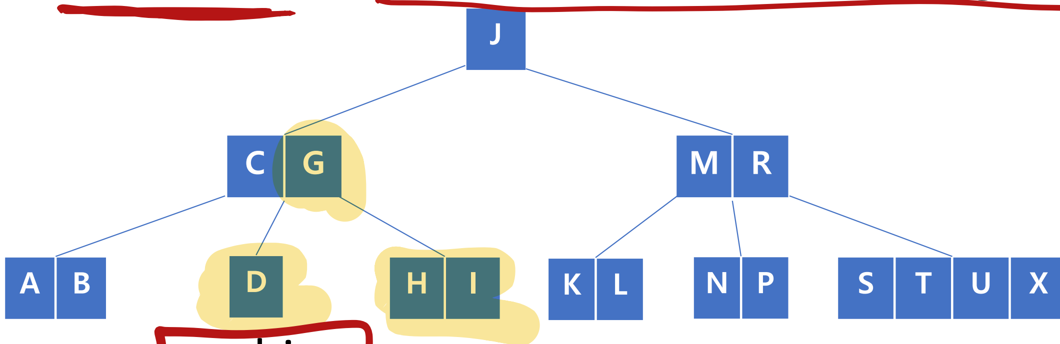
- Borrow from a neighbor



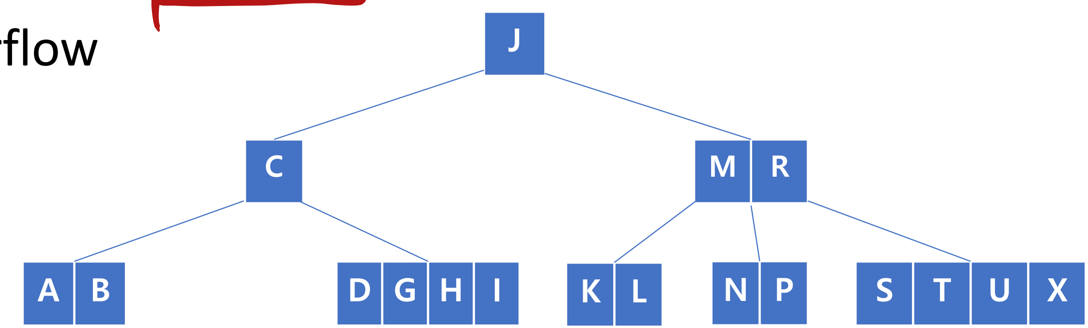
F가 내려오고 G가 올라감

Delete

- Delete F => underflow => but can't borrow from a neighbor

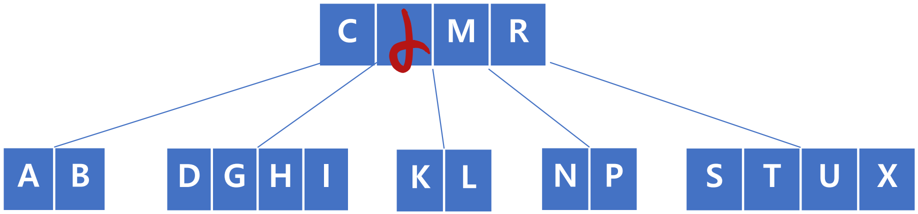


- Can't borrow so combine
- c is underflow

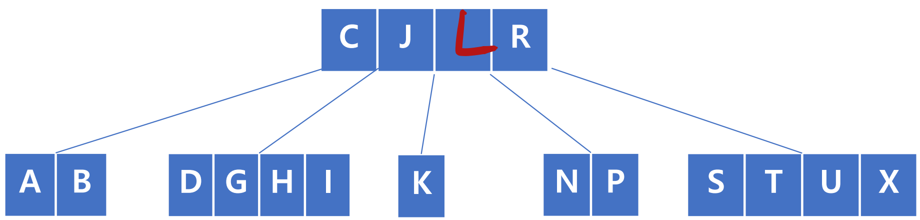


Delete

- so combine

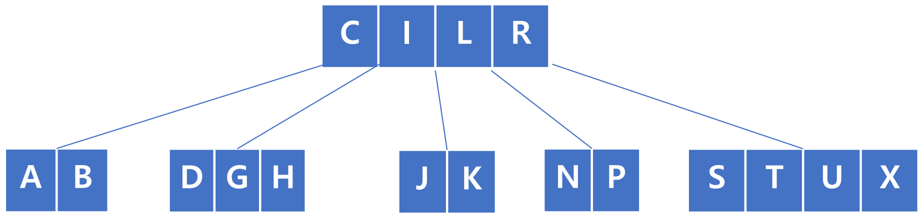


- Delete M from non-leaf node
- Note: immediate predecessor in non-leaf Is always in a leaf.
- Overwrite M with immediate predecessor (L) => underflow

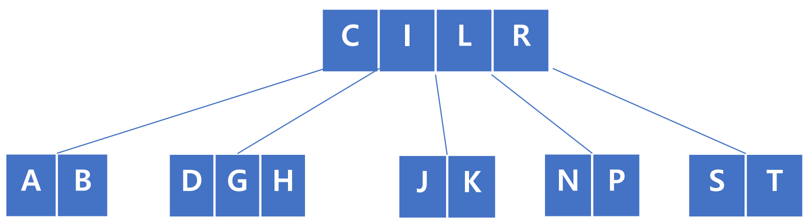


Delete

- Borrow from a neighbor

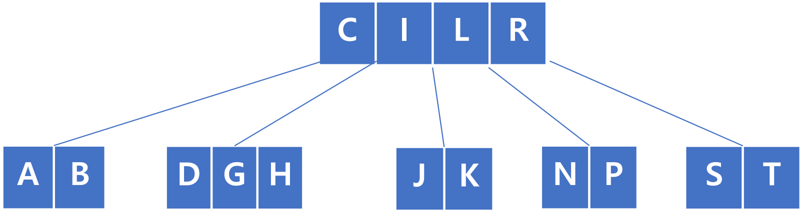


- Delete U, X

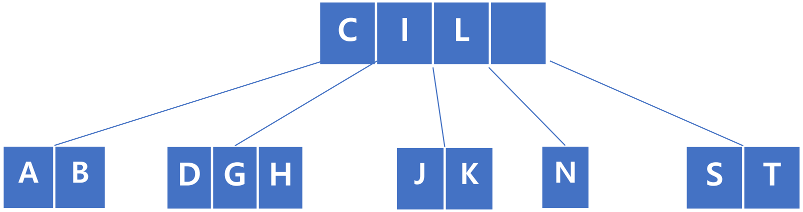


Delete

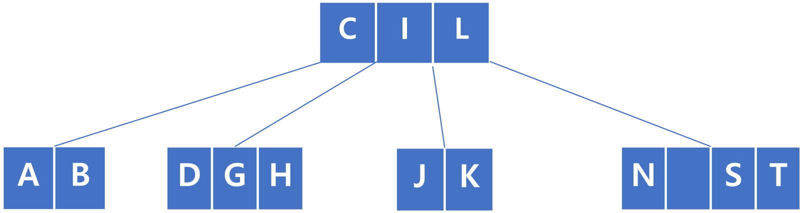
- Delete R



- Underflow, can't borrow => combine



Final



Analysis of B-Trees

- The maximum number of items in a B-tree of order m and height h:

  root $\quad m - 1$

  height 1 $\quad m(m - 1)$

  height 2 $\quad m^2(m - 1)$

  . . .

  height h $\quad m^h(m - 1)$

- So, the total number of items is

$$(1 + m + m^2 + m^3 + \ldots + m^h)(m - 1) =$$

$$[(m^{h+1} - 1)/\ (m - 1)]\ (m - 1) = \underline{m^h - 1}$$

- When $m = 5$ and $h = 2$, this gives $5^3 - 1 = \underline{124}$

Reasons for using B-Trees

- When searching tables held on disc, the cost of each disc transfer is high but doesn't depend much on the amount of data transferred, especially if consecutive items are transferred
  - If we use a B-tree of order 101, say, we can transfer each node in one disc read operation
  - A B-tree of order 101 and height 3 can hold $101^4 - 1$ items (approximately 100 million) and any item can be accessed with 3 disc reads (assuming we hold the root in memory)
- If we take *m* = 3, we get a **2-3 tree**, in which non-leaf nodes have two or three children (i.e., one or two keys)
  - B-Trees are always balanced (since the leaves are all at the same level), so 2-3 trees make a good type of balanced tree