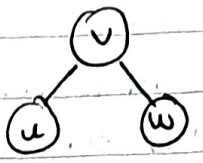
Week 4-1 Binary Search Tree

- Has key / key-value pairs
  ↑ must be able to be ordered
- For any node v,
  if u is in left subtree of v
  and w is in right subtree of v.

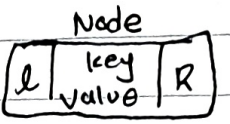


key(u) < key(v) < key(w)

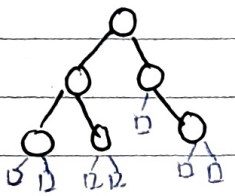
for unique keys.

↳ for multiple values with same key, store values in list *

* Inorder traversal visits keys of BST in increasing order!



BST Implementation

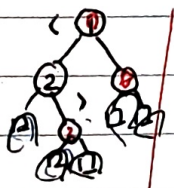


- only store keys in internal nodes
- external nodes will be represented by null.

Searching is more efficient! (than in list)

- Use BST property to search for elements in more efficient manner.

O(h)




```
def search(key, node):          ← normally root (called with)
    if node == null:            ← external
        return null
    if key(node) == key:        ← found
        return node
    elif key(node) > key:
        return search(key, node.left)
    else: (i.e. key(node) < key
        return search(key, node.right)
```

Node: key

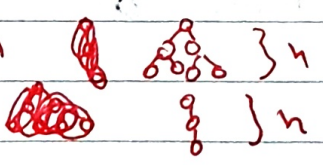
Node: key

Time Complexity

O(h)
- best case: $\log_2(n)$
- worst case: $n-1$

$h = \log_2(n+1) - 1$

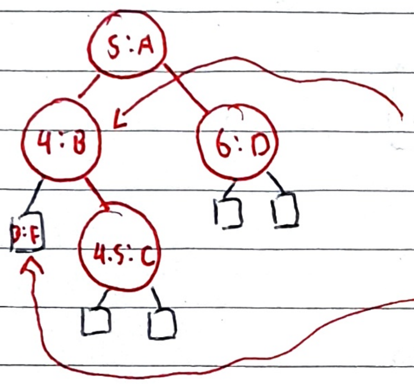
$h = n - 1$



O(h)

Insertion put(k, o) ← key ← value * For unique key BST

- First, search for k in tree.
  - if node with key k is not found:
    external_node = Node with key = k, value = o
  - if node is found:
    replace node's value with o.



put(4, E) = replace B with E

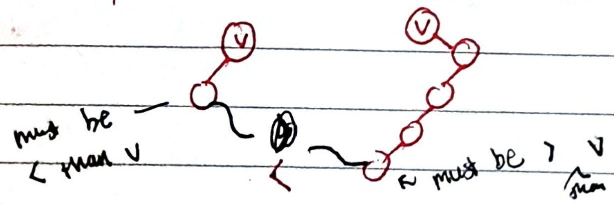
put(3, F) = insert Node(3:F)

deletion O(h) remove(k)

First, search for node with key k.
let us call it v.

- if v is null, node to delete is not in the tree
  → throw exception(?)
- If v has 2 external children:
  → simply remove v.
- elif v has 1 external child: and 1 internal child
  → link v.parent to v.internal_child. ← be it left or right
  (promote its internal child to its own place)
- else: 2 internal children
  → swap v and minimum (leftmost) of v's right subtree.
  → remove v

By definition, 왼쪽으로만 가야 minimum 찾음



4-1 cont

Range Queries

- find all keys between k_1, k_2. (inclusive)
- restricted version of inorder traversal

Recursive Search

- if $k < k_1$ → recursively search right subtree
- if $k_1 < k < k_2$ → recur-search left subtree + recur-search right subtree
- if $k > k_2$ → recursively search left subtree

```
output = []
def range(v, k1, k2):
    if v == null:
        return
    if v.key < k1:
        range(v.right, k1, k2)
    elif v.key > k2:
        range(v.left, k1, k2)
    else    (k1 <= v.key <= k2)
        range(v.left, k1, k2)
        output.append(v.key)
        range(v.right, k1, k2)
```

O(‖output‖ + height)

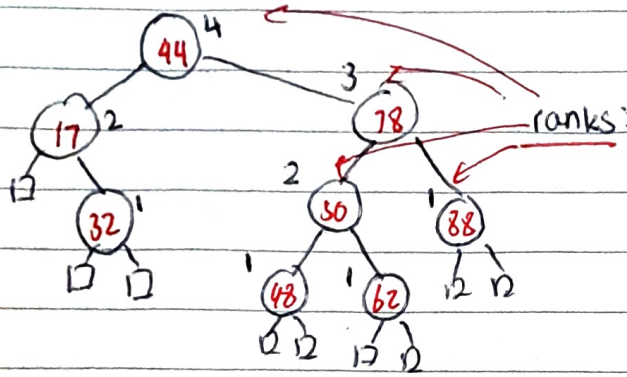
Lecture 4-2 AVL Trees

Rank-balanced Trees

↳ left/right subtree has approximately the same size

ex) AVL tree (lecture)

Red-black tree (textbook)



balance constraints: Ranks of the two children of every internal node can differ at most by 1.

ex) ~~Node(50)~~ & Node(88) has 1 rank difference (ok) :)

Node(48) & Node(62) has 0 " (even happier) :))

∴ **height of AVL tree**: $O(\log n)$

Proof:

Let $N(h)$ be minimum number of keys tree with height h must contain.

base case:

$N(1) = 1$

$N(2) = 2$

$N(3) = 4$

$N(4) = 8$ ~~8~~

$N(5) = 16$ ~~16~~

$N(6) = 32$ ~~32~~

Invariants: ① $N(h) > N(h-1)$

for $n > 2$,

$N(h) \geq 1 + N(h-1) + N(h-2) > 2N(h-2)$

$\therefore N(h) > 2N(h-2) > 4N(h-4)$ ~~$(N(h-4))$~~ $> 8(N-3) > 16(N-8)$.

$\therefore 2^{h/2} N(0)$

$\log N(h) \geq \log 2^{h/2}$

↳ $\log(N(h)) > \frac{h}{2}$ $= \boxed{h < 2\log_2(N(h))}$ $\therefore O(\log n)$

height $\leq$ 2 log(min # of nodes in tree with height h)

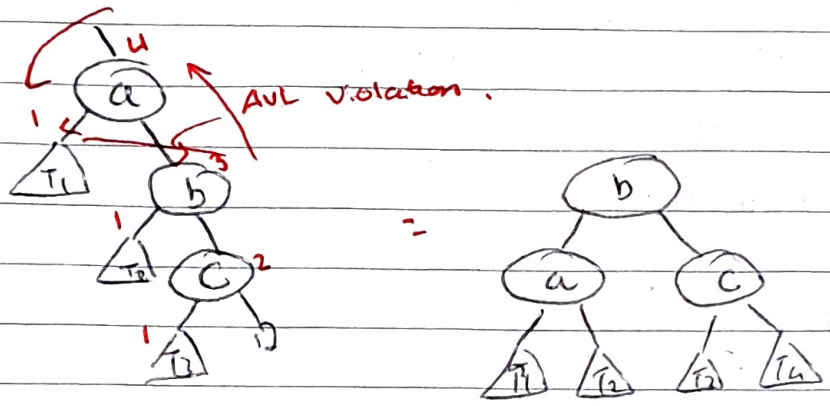
* Details of AVL tree not required to know.

Insertion

1. Insert as you would in normal BST.
2. ROTATE to re-arrange tree, so that AVL property is restored.

augment BST with h (height) attribute.

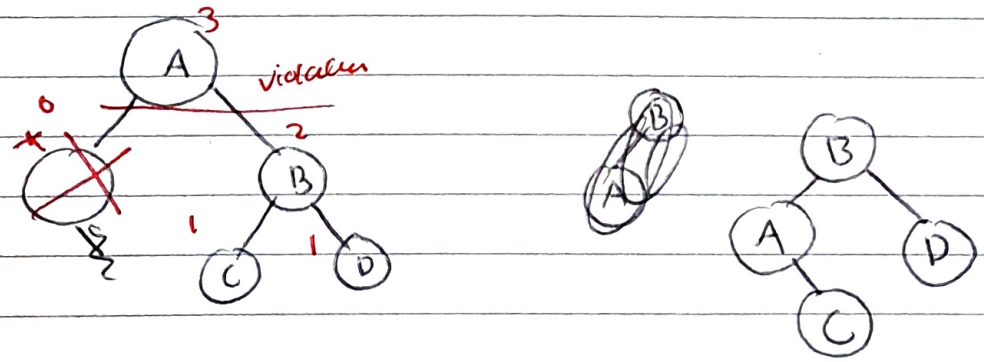
often, trinode restructure



Performance:

single, double rotation takes O(1) time...
imple-

Removal ex



height = O(log n) *guaranteed

{ searching, insertion, removal } O(log n)

4-3 Lecture : Map Intro

Map ADT

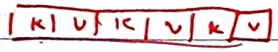
- get(k) - return value with key k.
- put(k, v) - add/update key k with value v.
- remove(k) - remove key k.

DS

What can we use for map?

- BST
- (doubly) linked list

- array



- dictionary 아직은 어떻게 생겼는지 모릅니다.

List-Based (unordered)

get: O(n)

put: ~~O(1)~~ key may already exist ∴ search needed
O(n)

BST-based (balanced e.g. AVL)

get: O(log n)

put: O(log n)