CSCB63 – Design and Analysis of Data Structures

Anya Tafliovich[1]

[1] based on notes by Anna Bretscher and Albert Lai

AVL tree

- stores key/value pairs in all nodes (both leaf and internal)
- has a property relating the keys stored in a subtree to the key stored in the parent node (ordering)
- maintains the height (number of edges on a root-to-leaf path) of $\mathcal{O}(\log n)$
  - balance factor = height(left subtree) − height(right subtree)
  - maintain balance factor of ± 1 or 0 for all nodes

Operations are $\mathcal{O}(\log n)$:

- `search(k, T)`: return the value corresponding to key k in the tree T
- `insert(k, v, T)`: insert the new key/value pair k/v into the tree T
- `delete(k, T)`: delete the key/value pair with key k from the tree T

more AVL operations

Given two AVL trees, T_1 and T_2, create the

- union of T_1 and T_2
  - an AVL tree T that contains key/value pairs from T_1 as well as from T_2
  - if $(k, v_1) \in T_1$ and $(k, v_2) \in T_2$, then decide whether $(k, v_1) \in T$ or $(k, v_2) \in T$
- intersection of T_1 and T_2
  - an AVL tree T that contains key/value pairs that are in both T_1 and T_2
  - if $(k, v_1) \in T_1$ and $(k, v_2) \in T_2$, then decide whether $(k, v_1) \in T$ or $(k, v_2) \in T$
- difference of T_1 and T_2
  - an AVL tree T that contains key/value pairs that are in T_1 but not in T_2

AVL union

Given two AVL trees, T_1 and T_2, create the union of T_1 and T_2:

- an AVL tree T that contains key/value pairs from T_1 as well as from T_2
- if $(k, v_1) \in T_1$ and $(k, v_2) \in T_2$, then we will have $(k, v_2) \in T$ (update)

Simple way to construct the union:

- wlog, $numnodes(T_1) = n <= m = numnodes(T_2)$
- insert all nodes from T_1 into T_2
- complexity? $\underbrace{n}_{n \text{ nodes}} \cdot \underbrace{\log_2(m+n)}_{\text{insert 1 node}}$
- can we do better?

divide and conquer algorithms

Idea:

- split the input into smaller pieces (divide)
  - obtain smaller problems of the same kind
- apply the algorithm to the smaller pieces (conquer)
  - obtain solutions to the smaller problems
- build the answer from the answers to the smaller problems

Some example you have seen before?

binary search, quicksort, mergesort

AVL union

Given two AVL trees, T_1 and T_2, create the union of T_1 and T_2.

Divide and conquer approach:

- split T_1 into smaller trees
- split T_2 into smaller trees
- build unions of smaller trees
- merge results into union of T_1 and T_2

AVL union: split

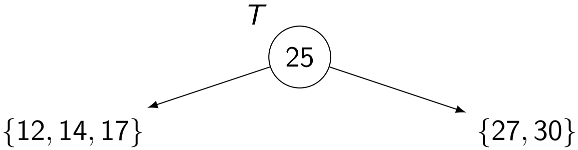
- suppose tree T_2 has key k at root node
- split T_1 into $T_{<k}$ and $T_{>k}$, both balanced
  - $T_{<k}$ contains keys from T_1 that are less than k
  - $T_{>k}$ contains keys from T_1 that are bigger than k

- need algorithm `split(T, k)` that returns `(`$T_{<k}$`, `$T_{>k}$`)` such that both $T_{<k}$ and $T_{>k}$ are AVL trees

3 join(2 union($T_{<k}$, L), k, 2 union($T_{>k}$, R))

AVL union: split

`split(T, k)` idea



- how to split at key 16?

$\{12, 14\}$, join($\{17\}$, 25, $\{27, 30\}$)

AVL union: split

`split(T, k)` algorithm

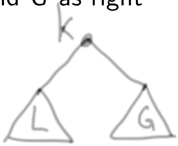
```
if T == nil:
    return (nil, nil)
if k == T.key:
    return (T.left, T.right)
if k < T.key:
    (L, R) = split(T.left, k)
    R' = join(R, T.key, T.right)
    return (L, R')
if k > T.key:
    (L, R) = split(T.right, k)
    L' = join(T.left, T.key, L)
    return (L', R)
```

Need algorithm for join!

AVL union: join

`join(L, k, G)` idea

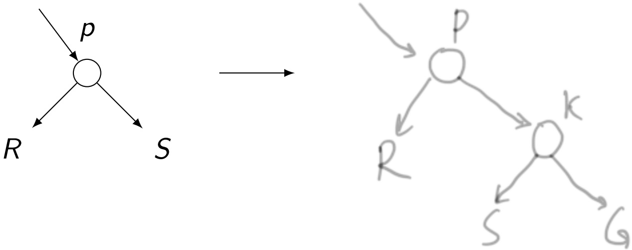
- L already contains keys $< k$, G already contains keys $> k$
- if L much taller than G $(height(L) - height(G) > 1)$
  - insert k and G as subtree into L
- if G much taller than L $(height(G) - height(L) > 1)$
  - insert k and L as subtree into G
- if L and G differ by ≤ 1 $(abs(height(L) - height(G)) \leq 1)$
  - make a tree with k in root, L as left subtree, and G as right subtree



AVL union: join

if $height(L) - height(G) > 1$, insert G as subtree into L:

1. in L, keep going to the right to find the node p such that
   - p is still too tall: $height(p) - height(G) > 1$, but
   - but $p.right$ is just right: $height(p.right) - height(G) \leq 1$
2. create new node q with key k, left child $p.right$, and right child G, this node becomes p's new right child
3. rebalance from p upwards, as needed



AVL union: join

if $height(L) - height(G) > 1$, insert G as subtree into L.

How do we know the result is an AVL?

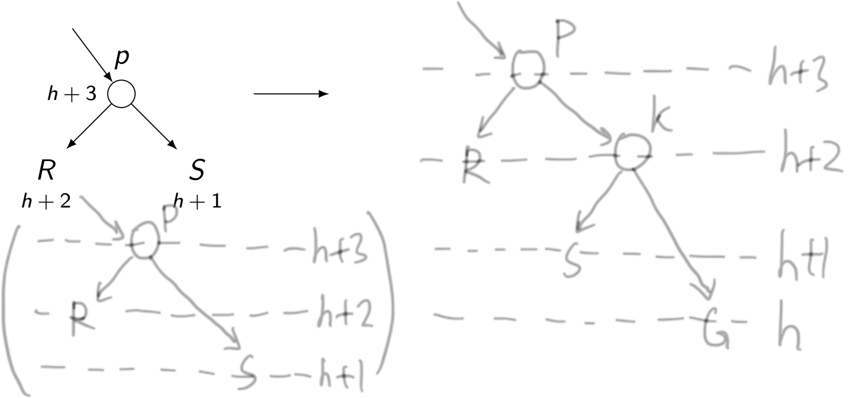
AVL union: join

- $height(p) - height(G) > 1$, but
- $height(p.right) - height(G) \leq 1$

no rotation required

Let $h = height(G)$.

Case 1:



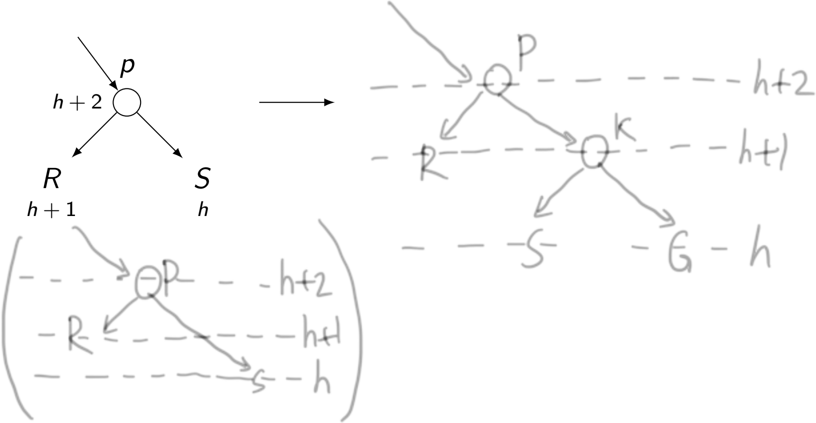
AVL union: join

- $height(p) - height(G) > 1$, but
- $height(p.right) - height(G) \leq 1$

no rotation required (sorted & balanced)

Let $h = height(G)$.

Case 2:

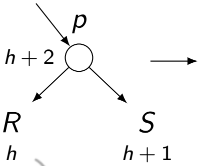


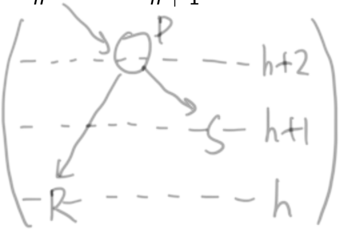
AVL union: join

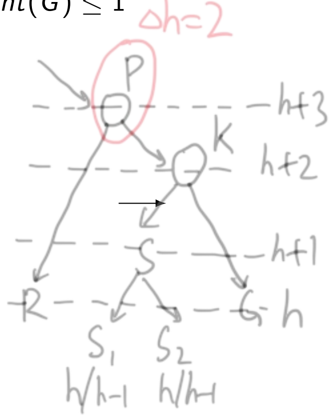
- *height*(p) − *height*$(G) > 1$, but
- *height*$(p.right)$ − *height*$(G) \leq 1$

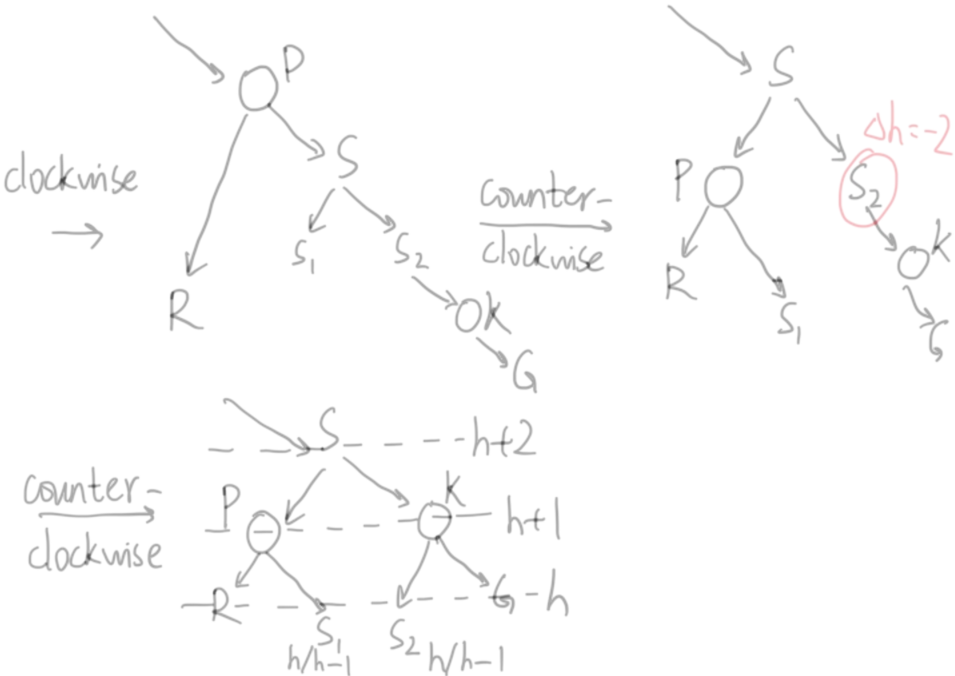
Let $h =$ *height*(G).

Case 3:









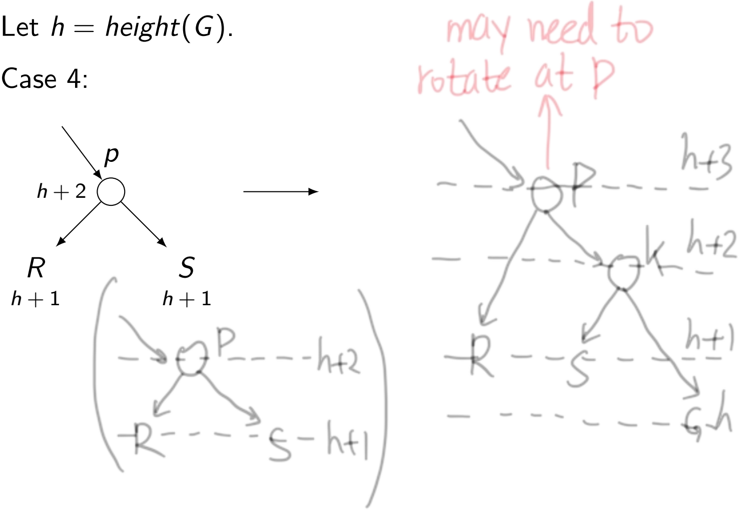
clockwise
P
S
R
S_1
S_2
K
G
counter-clockwise
S
P
S_2
$\Delta h = -2$
K
R
S_1
G
counter-clockwise
S
h+2
P
K
h+1
R
S_1
S_2
G
h
h/h-1
h/h-1

AVL union: join

- *height*(p) − *height*$(G) > 1$, but
- *height*$(p.right)$ − *height*$(G) \leq 1$

Let $h =$ *height*(G).

Case 4:



AVL union: join

`join(L, k, G)` pseudocode

```
if height(L) - height(G) > 1:
  p = L (G)             (p.left)
  while height(p.right) - height(G) > 1:
                                   (L)
    p = p.right (p.left)
  q = new node(key=k, left=p.right, right=G)
(p.left) p.right = q             (L)         (P.left)
  rebalance and update heights at p up to the root
  return L (G)
elif height(G) - height(L) > 1:
  ... symmetrical ...
else:
  return new node(key=k, left=L, right=G)
```

AVL union

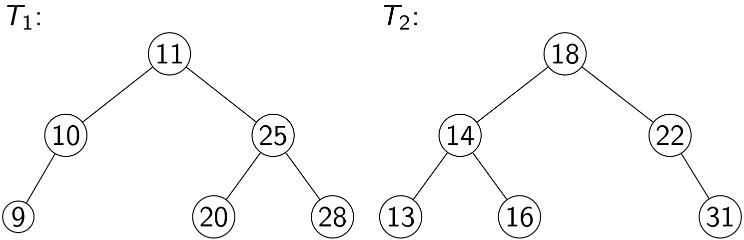
Finally, `union(`T_1`, `T_2`)` algorithm:

```
if T_1 == nil:
  return T_2
if T_2 == nil:
  return T_1

k = T_2.key
(L, R) = split(T_1, k)
L' = union(L, T_2.left)
R' = union(R, T_2.right)
return join(L', k, R')
```

AVL union: example

Follow all the steps of the algorithm above to construct the union of:



Complete example in tutorial.

AVL union: complexity

- So, did we do better than our first try?

- Best union / intersection / difference algorithm for balanced trees (including AVL and red-black trees) is $\Theta(m \log(\frac{n}{m} + 1))$

- Can find proof of complexity in Guy Blelloch, Daniel Ferizovic, and Yihan Sun, *Parallel ordered sets using join*. ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), 2016. `https://arxiv.org/abs/1602.02120`