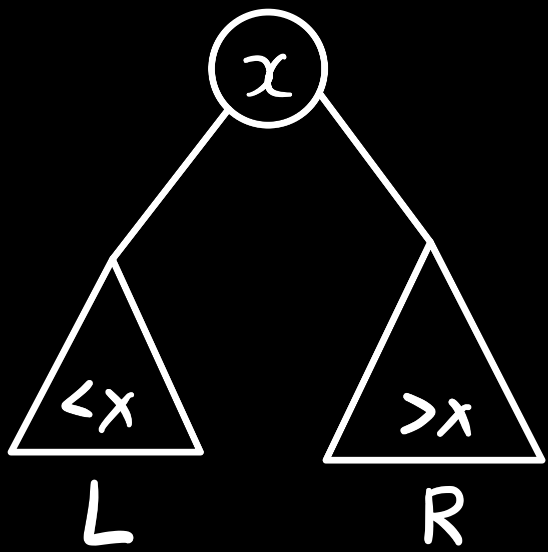
# Splay Trees

Roger Fu

# Binary Search Trees

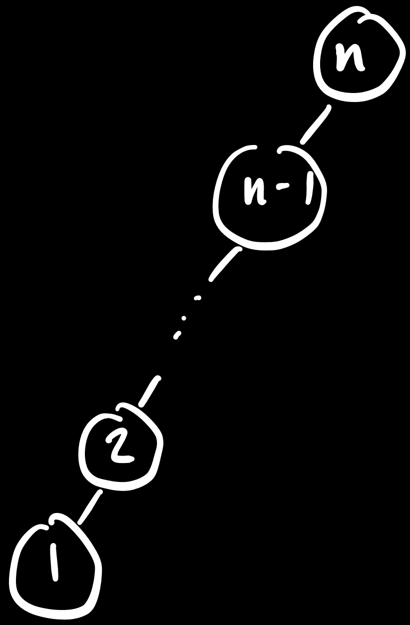
- Empty; or tuple $(x, L, R)$:




$x$ is the **key**, $L$ the **left subtree**, $R$ the **right subtree**

**Binary search tree property:** all keys in $L < x$, all keys in $R > x$.

# Binary Search Tree: A Weakness




Searching for $1, 2, \ldots, n-1, n$ takes

$$n + (n-1) + \cdots + 2 + 1 = \frac{n(n+1)}{2} \text{ operations}$$

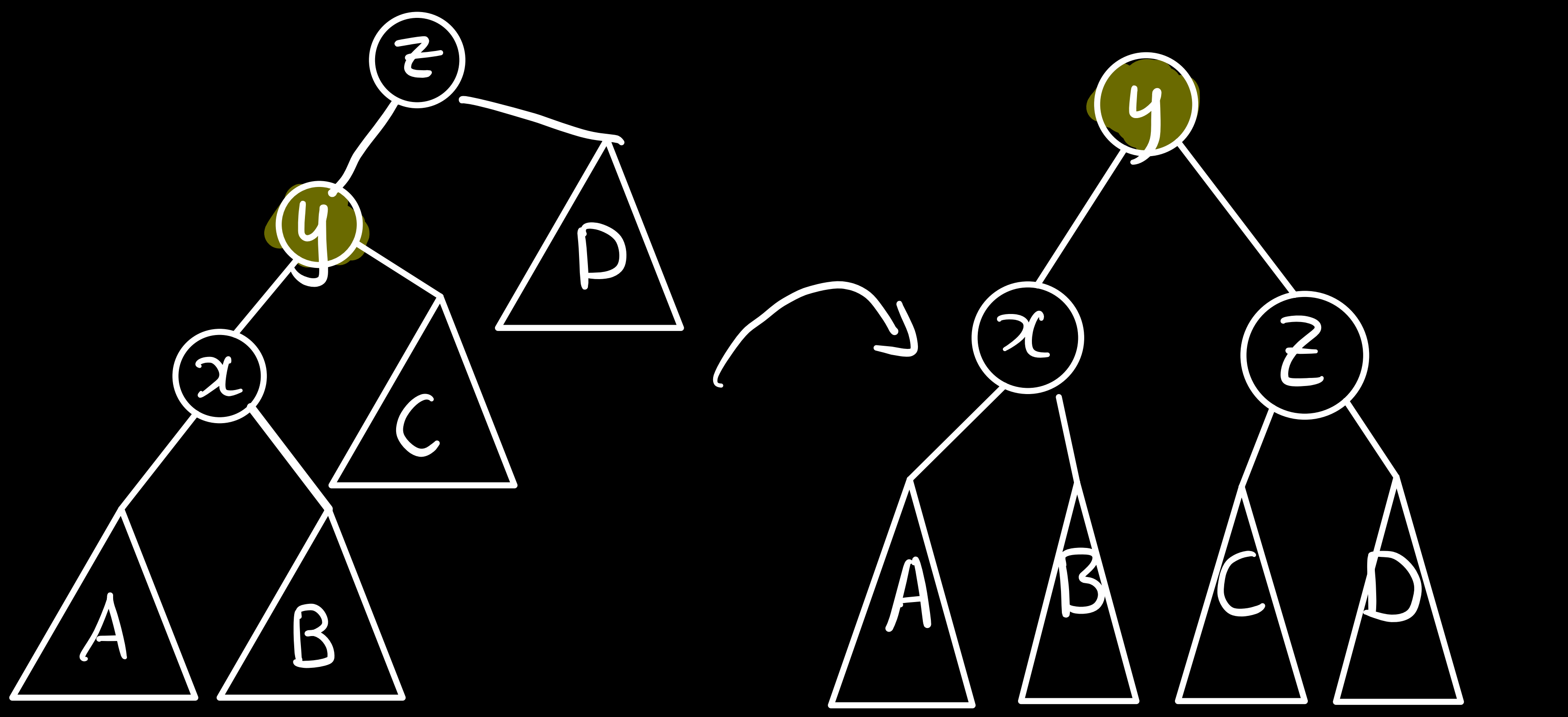
Idea: introduce splay trees to get subquadratic time

# Splay Trees

Intuition: make recently accessed elements easy to acess again

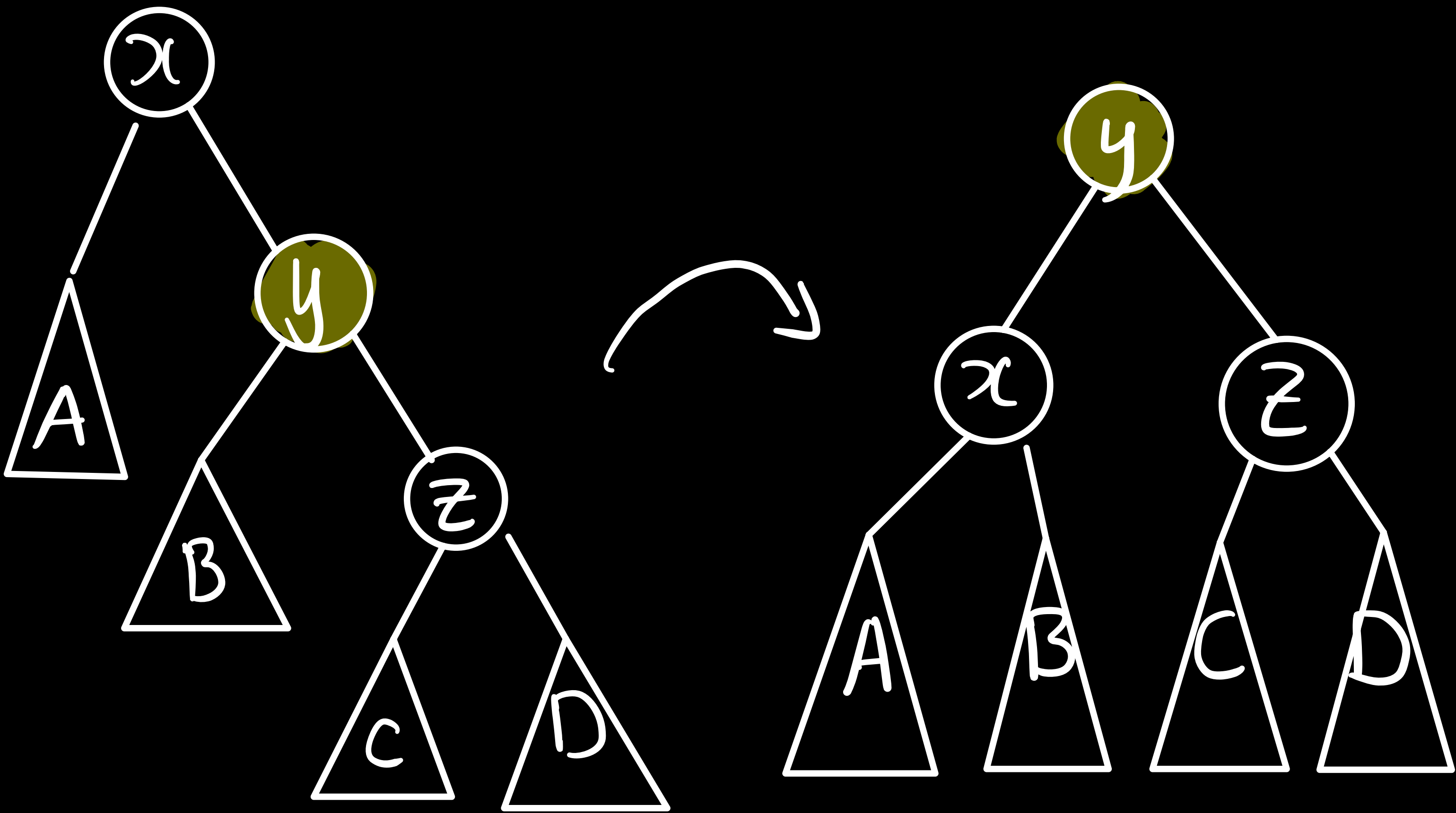
We will splay the last accessed node to the root after each operation

# Rotation




Right rotation

# Rotation




Left rotation

# Rotating Towards Root

If $x$ is left (right) child of $p$, we say a right (left) rotation rotates $x$ towards the root.

↳ $x$ replaces $p$ as root

# Naive Attempt: Rotate-to Root

Natural 1st idea: repeatedly rotate node $x$ to root

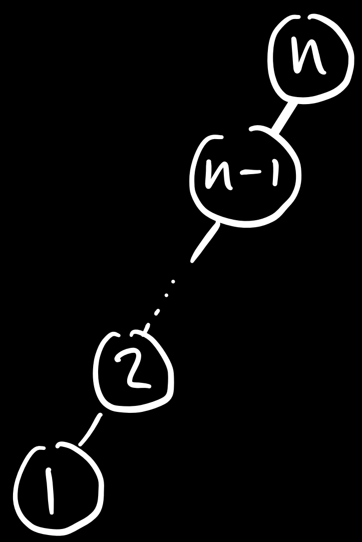
Denote as $rtr(x)$

# Naive Attempt: Rotate-to Root

Natural 1st idea: repeatedly rotate node $x$ to root

Denote as $rtr(x)$

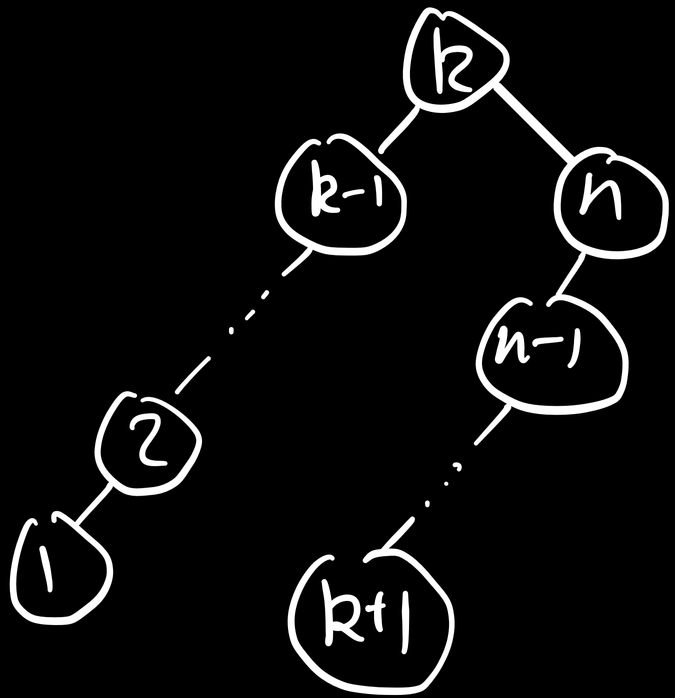
Problem:




Consider $rtr(1), \ldots rtr(n)$

# Naive Attempt: Rotate-to Root

Claim: After rtr(1), ..., rtr(k) we get



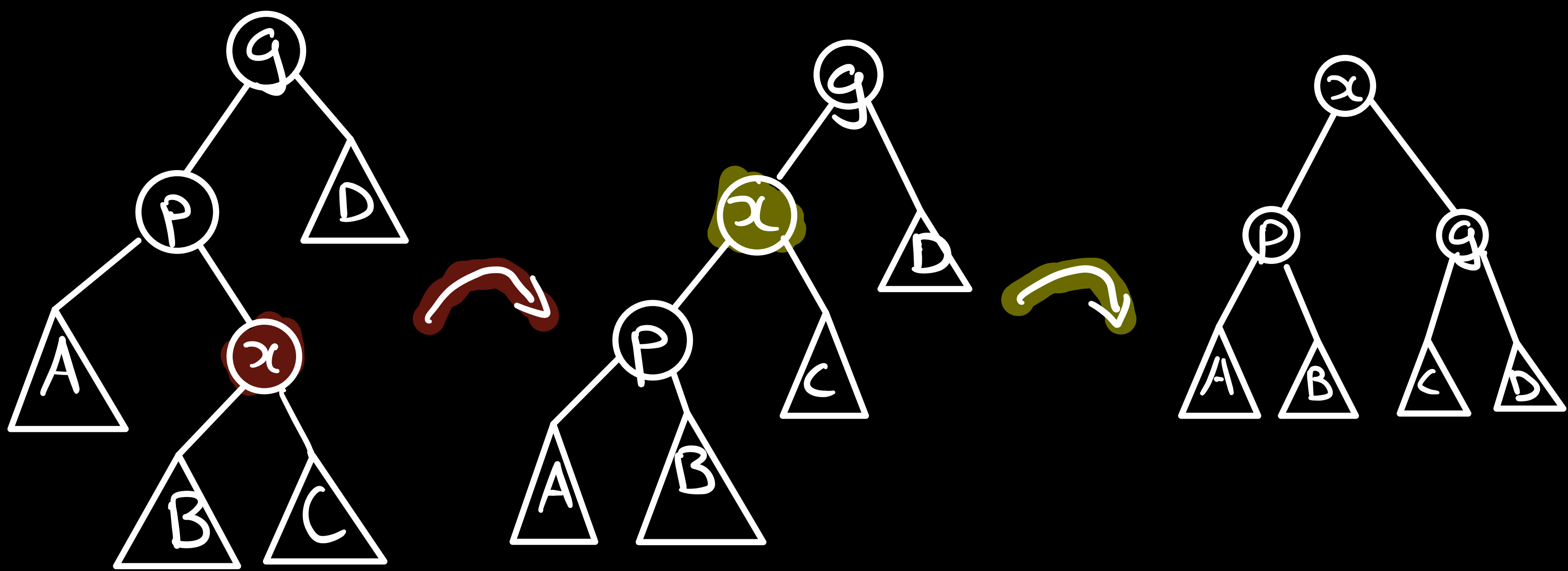

Pf: Induct.

Note that rtr(k+1) takes $\Theta(n-k)$ rotations : total: $\Theta(n^2)$

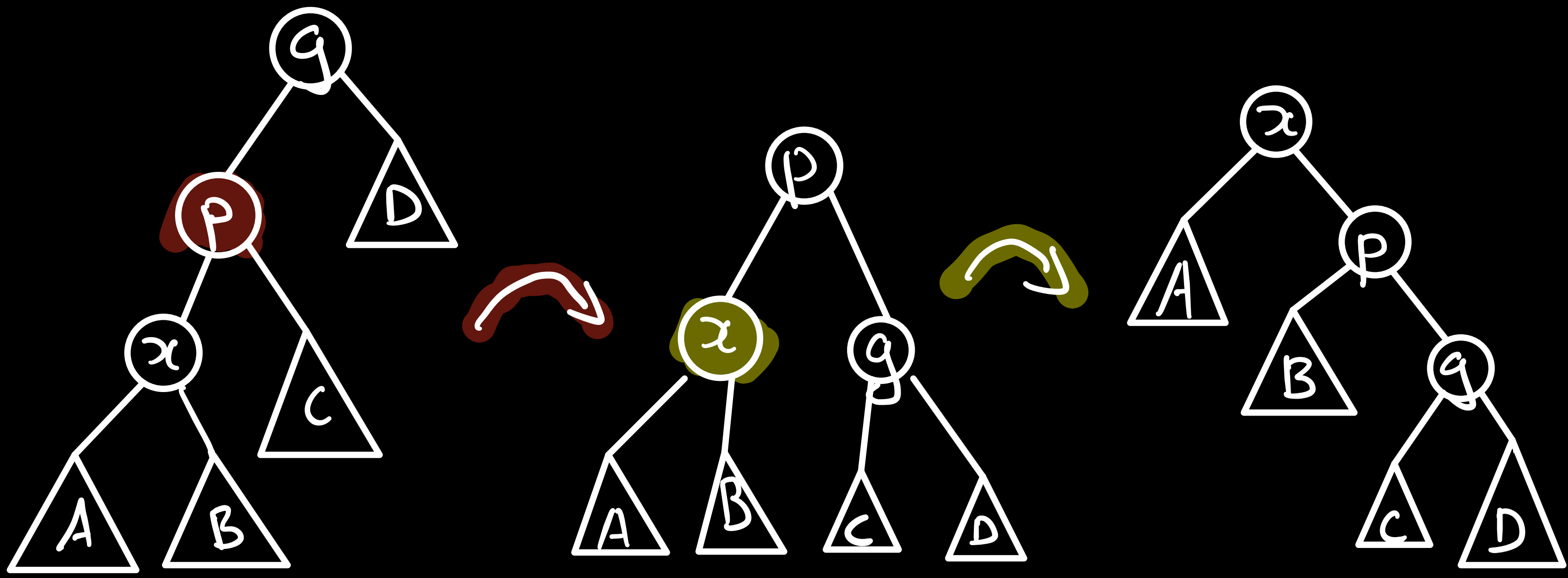
# The Splay Operation:

```
splay(x):
    while x is not the root:
        p ← x.parent
        if p is the root:
            rotate x towards root
        else:
            if p and x both left/right children:
                zig-zig(x)
            else:
                zig-zag(x)
```

# Zig-zag



# Zig-zig

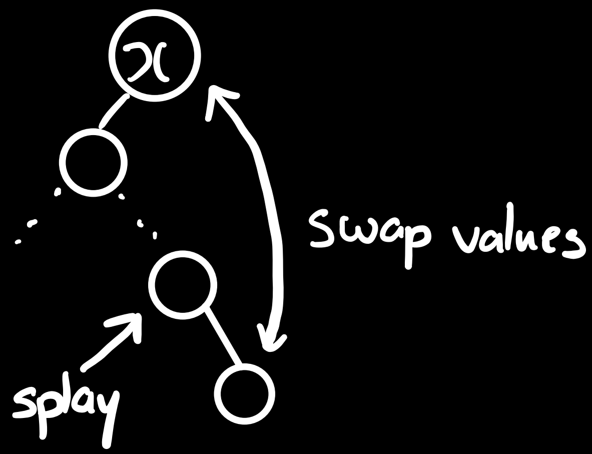



Remark: This is the difference between rotate-to-root & splay

# Binary Search Tree Operations

Usual binary search tree operations augmented by splaying last accessed node (insertion, searching)

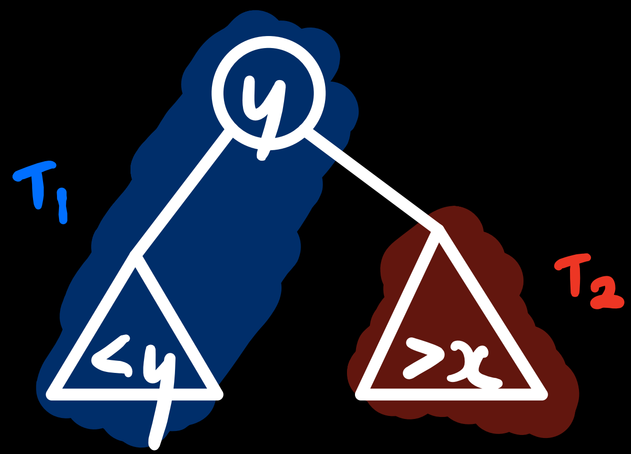
Deletion: if node is deleted, splay the parent:



# Splitting

Split: given splay tree $T$, partition into $T = T_1 \cup T_2$ where all keys in $T_1 \leq x$, all keys in $T_2 > x$

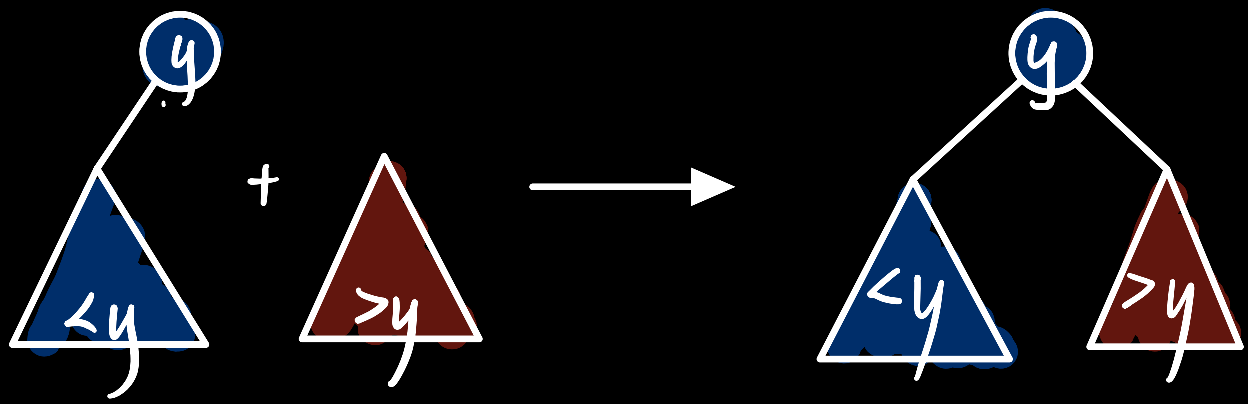
1. Find largest $y \leq x$ s.t. $y \in T$
2. Splay $y$ to root
3. Partition right subtree into $T_2$:



# Merging

**Merge:** given $T_1$, $T_2$ where $\max(T_1) < \min(T_2)$, combine to form $T = T_1 \cup T_2$.

1. Splay largest element of $T_1$ to root
2. Attach $T_2$ as right child of $T_1$



# Time Complexity

- Time complexity dominated by cost of searching for & splaying node
- Problem: runtime of splaying depends on structure of tree, which depends on previous operations done

  ↳ Can establish amortized bound of $O(\log n)$

# Detour: Amortized Analysis

Let $T^{actual}(\mathcal{O})$ runtime of operation $\mathcal{O}$ & $T^{amort}(\cdot)$ function on operations. $T^{amort}(\cdot)$ upper bounds the amortized run-time if for any sequence of operations $\mathcal{O}_1, \dots, \mathcal{O}_k$ we have

$$\sum_{i=1}^{k} T^{actual}(\mathcal{O}_i) \leq \sum_{i=1}^{k} T^{amort}(\mathcal{O}_i).$$

Intuition: Amortized time complexity is like average upper bound

# Splay Trees & Range Queries I

**Range query problem:** given list $x_1, \dots, x_n$ support:

1. Calculating $f(x_i, x_{i+1}, \dots, x_j)$
2. Changing $x_i$.

Will focus on case where there exists $g$ such that

$$g(g(f(x_i, \dots, x_{k-1}), x_k), f(x_{k+1}, \dots, x_j))$$

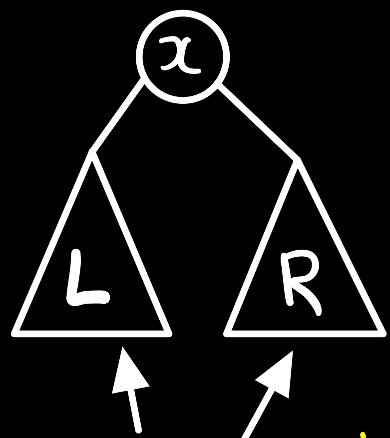
**Example:**

- $f$ is the sum
- $f$ is the max
- $f$ is max subarray sum

# Splay Trees & Range Queries II

Augment splay tree T by storing $f(T)$ at root node

Combining: $g(g(f(L), x), f(R))$




assume, recursively,
$f(L)$ & $f(R)$ known

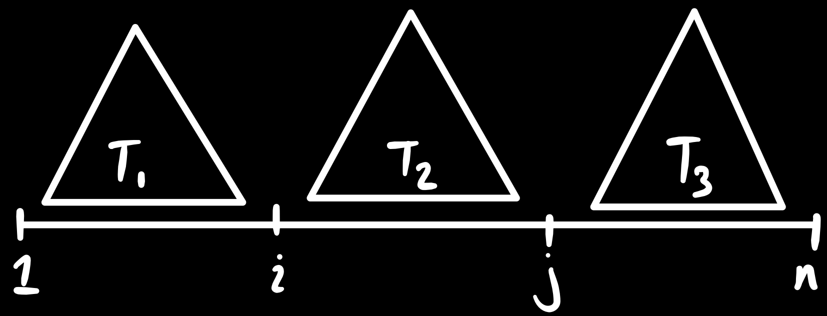
Note: when rotating, $f$ needs to be recomputed

# Splay Tree & Range Queries III

1. Construct splay tree w/ keys $1, 2, \ldots, n$

2. At node $i$, store $x_i$ and for each subtree $T$, compute $f(T)$

3. To query $f(x_i, \ldots, x_j)$, split on $i-1$ and $j$:




Return $f(T_2)$ (and merge the trees back together)

# Splay Tree & Range Queries IV

4. To update $x_i$, splay $i^{th}$ node to root & update $x_i$.
   ↳ Only need to recompute $f(T)$.

## Time Complexity:

Operations dominated by cost of splaying.

Amortized $O(c \log n)$ where $c$ is cost of $g(\cdot)$

Warning! Splay tree is generally slower than segment tree!

*JUST BECAUSE BOTH ARE $O(\log n)$ DOES NOT MAKE THEM INTERCHANGABLE!

# Implicitly Keyed Splay Trees

Idea: Instead of explicitly using $1, \ldots, n$ as keys, use order in in-order traversal as key

Advantages: Supports modifying underlying list:

- Insertion at arbitray indices
- Deletion of arbitrary indices
- Moving subarrays around
- Reversing subarrays

# Implicitly Keyed Splay Trees

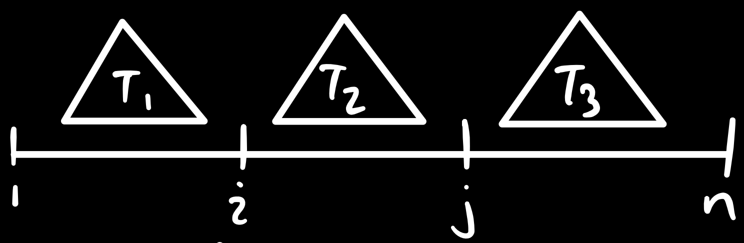
```
def get_val(x: node, pos: int):
    if sz(x.L)+1 = pos:
        splay(x)
        return x
    else if sz(x.L)+1 < pos:
        return get_val(x.R, pos-(sz(x.L)+1))
    else:
        return get_val(x.L, pos)
```

Remark: You can similarly modify split. Join becomes equivalent to concatination.

# Lazy Propogation

**Idea:** Instead of applying update to range, update the stored aggregate value and set flag to propogate changes to children

**Example:** Range sum update & query
- For each subtree $T$ store sum $f(T)$
- To query sum, split:




and return $f(T_2)$

- To update, split and update lazy propogation flag on $T_2$ (and update $f(T_2)$)

# Reversing Ranges

- Lazy propogation to swap left & right child
- Be careful if your aggregate function f is not commutative!
  - ↳ Example: largest prefix sum

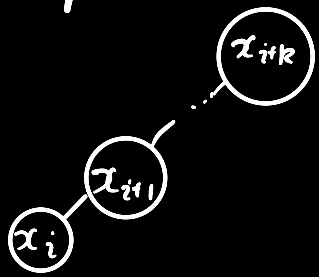
# Practice Problems

- spoj.com/problems/SEQ2
- dmoj.ca/problem/cco16p6
- dmoj.ca/problem/ds4

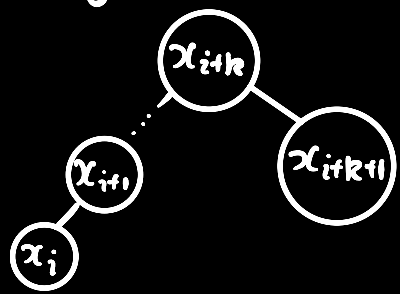
# Hints

- SEQ2: you should aim to insert $k$ consecutive numbers in $O(k + \log n)$

  Verify inductively the inserted elements form a tree like:




and inserting $x_{i+k+1}$ gives



# Bonus: Size-Balanced Tree

- Balanced binary search tree that balances itself by checking invariant on subtree sizes.

- Advantages over splay tree:
  - ↳ Doesn't store extra data for rebalancing (splay tree needs parent pointers)
  - ↳ "Tendancy of perfect BST in practice"

- Disadvantages compared to splay tree:
  - ↳ Not clear how to implement split/merge