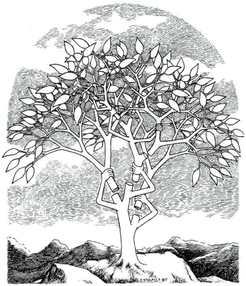
# Self Adjusting Data Structures

Pedro Ribeiro

DCC/FCUP

2020/2021

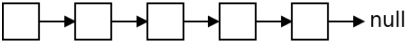


# What are self adjusting data structures?

- Data structures that can **rearrange** themselves **after operations** are committed to it.
- This is typically done in order to **improve efficiency** on future operations.
- The rearrangement can be **heuristic** in its nature and typically happens in **every operation** (even if it was only accessing an element).
- Some examples:
  - Self Organizing Lists
  - Self Adjusting Binary Search Trees

# Traversing Linked Lists

- Consider a classic linked list with $n$ elements:




- Consider a **cost model** in which accessing the element in position $i$ costs $i$ (*traversing the list*)
- What is the average cost for accessing an element using a **static list**?
  - Intuitively, if the element to be searched is a *"random"* element in the list, our average cost is "roughly" $n/2$

# Formalizing The Cost

- Let's formalize a little bit more:
  - Let $p(i)$ be the probability of searching for element in positions $i$
  - On average, our cost will be:

$$T_{avg} = 1 \times p(1) + 2 \times p(2) + 3 \times p(3) + \ldots + n \times p(n)$$

- Suppose that the the probability is the same for every element: $1/n$.
  - $T(n) = 1/n + 2/n + 3/n + \ldots + n/n = (1 + 2 + 3 + \ldots + n)/n = (n+1)/2$
- But what if the probability is not the same?
  - What if we typically access nodes at the front of the list?
  - What if we typically access nodes at the back of the list?

# Cost on non-uniform access

- Let's look at an example:

$$P(A) = 0.1 \quad P(B) = 0.1 \quad P(C) = 0.4 \quad P(D) = 0.1 \quad P(E) = 0.3$$

$$\boxed{A} \rightarrow \boxed{B} \rightarrow \boxed{C} \rightarrow \boxed{D} \rightarrow \boxed{E} \rightarrow \text{null}$$

$$T(n) = 1 \times 0.1 + 2 \times 0.1 + 3 \times 0.4 + 4 \times 0.1 + 5 \times 0.3 = 3.4$$

If we know in advance this access pattern can we do better?

$$\boxed{C} \rightarrow \boxed{E} \rightarrow \boxed{A} \rightarrow \boxed{B} \rightarrow \boxed{D} \rightarrow \text{null}$$

$$T(n) = 1 \times 0.4 + 2 \times 0.3 + 3 \times 0.1 + 4 \times 0.1 + 5 \times 0.1 = 2.2$$

And what if we know the exact (non-static) search pattern?

# Strategies for Improving

- Can you think of any strategies for improving if we do not know in advance what is the access pattern?
- **Intuition:** bring items frequently accessed closer to the front
- Three possible strategies (among others) after accessing an element:
  - **Move to Front (MTF)**: move element to the head of the list
  - **Transpose (TR)**: swap with previous element
  - **Frequency Count (FC)**: count and store the number of accesses to each element. Order by this count.

# Competitive Analysis

- **Idea:** look at the ratio of our algorithm *vs* best achievable

**$r$-competitiveness**

An algorithm has competitive ratio $r$ (or is $r$-competitive) if for some constant $b$, for any sequence of requests $s$, we have:

$Cost_{alg}(s) \leq r \times Cost_{OPT}(s) + b$

where $\mathtt{OPT}$ is the optimal algorithm (in hindsight)

- Consider the following cost model:
  - Accessing item at position $i$ costs $i$
  - After accessing it, we can bring it forwards as much as we want for free

# Competitive Analysis of Self Organizing Lists

**Claim - TR has as a bad competitive ratio:** $\Omega(n)$

Consider the following sequence of operations:

- Consider any list with $n$ elements
- Ask $n$ times for the last element in the sequence

  Example:
  $\mathtt{A} \rightarrow \mathtt{B} \rightarrow \mathtt{C} \rightarrow \mathtt{D} \rightarrow \mathtt{E}$
  find(E), find(D), find(E), find(D), ...

- This strategy will pay $n^2$
- A better option would be bringing both elements to front paying $n+n+2+2+2+2+2+2+\ldots = n+n+2(n-2) = 2n+2n-4 = 4n-4$
- The ratio for $m$ operations like these is $n^2/(4n-4)$ which is $\Theta(n)$

# Competitive Analysis of Self Organizing Lists

**Claim - FC has as a bad competitive ratio:** $\Omega(n)$

Consider the following sequence of operations:

- Consider an initial request sequence that sets up counts: $n-1, n-2, \ldots, 2, 1, 0$
- Repeat indefinitely: ask $n$ times for the element that was last

  Example:
  $\texttt{A} \rightarrow \texttt{B} \rightarrow \texttt{C} \rightarrow \texttt{D} \rightarrow \texttt{E}$
  find(E), find(E), find(E), find(E), ...

- Each of these iterations will pay $n + (n-1) + (n-2) + \ldots + 2 + 1 = n(n+1)/2 = (n^2+n)/2$
- Optimal in this case would bring the element to the front on the first request, paying $n + 1 + 1 + 1 + 1 + 1 + \ldots = 2n - 1$
- The ratio for $m$ operations like these is $\Theta(n)$

# Competitive Analysis of Self Organizing Lists

What about MTF? Can you find any "bad" sequence of operations?

**Claim - MTF is** $2$**-competitive**

For this we can use **amortized analysis**

# Remembering Amortized Analysis

- In **amortized analysis** we are concerned about the the average over a **sequence of operations**
  - Some operations may be costly, but others may be quicker, and in the end they even out
- One possible method is using **potential functions**
  - A potential function $\Phi$ maps the state of a data structure to non-negative numbers
  - You can think of it as "potential energy" that you can use later (like a guarantee of the "money we have in the bank")
  - If the potential is non-negative and starts at 0, and at each step the actual cost of our algorithm plus the change in potential is at most $c$, then after $n$ steps our total cost is at most $cn$.

# Remembering Amortized Analysis

- Relationship between potential and actual cost
  - State of data structure at time $x$: $S_x$
  - Sequence of $n$ operations: $O = o_1, o_2, \ldots, o_n$
  - Amortized cost per operation $o$:
    $T_{am}(o) = T_{real}(o) + (\Phi(S_{after}) - \Phi(S_{before}))$
  - Total amortized cost: $T_{am}(O) = \sum_i T_{am}(o_i)$
  - Total actual (real) cost: $T_{real}(O) = \sum_i T_{real}(o_i)$
  - $T_{am}(O) = \sum_{i=1}^{n} T_{real}(o) + (\Phi(S_{i+1}) - \Phi(S_i)) =$
    $T_{real}(O) + (\Phi(S_{end}) - \Phi(S_{start}))$
  - $T_{real}(O) = T_{am}(O) + (\Phi(S_{start}) - \Phi(S_{end}))$

    If $\Phi(S_{start}) = 0$ and $\Phi(S_{end}) \geq 0$, then $T_{real}(O) \leq T_{am}(O)$ and our amortized cost can be used to accurately predict the actual cost!

# Competitive Analysis of Self Organizing Lists

**Claim - MF is 2-competitive**

- The key is defining the right potential function $\Phi$
- Let $\Phi$ be the number of **inversions** between MTF and OPT lists, i.e., $\#pairs(x, y)$ such that $x$ is before $y$ in MTF and after $y$ in OPT list.
- Initially our $\Phi$ is zero and it will never be negative.
- We are going to show that amortized cost of MTF is smaller or equal than twice the real cost of OPT:

  $Cost_{MTF} + (\text{change in potential}) \leq 2 \times Cost_{OPT}$

  This means that after any sequence of requests:

  $cost_{MTF} + \Phi_{final} \leq 2 \times cost_{OPT}$

  Hence, it would mean that MTF is 2-competitive.

# Competitive Analysis of Self Organizing Lists

**Claim - MF is 2-competitive**

- $\Phi$ is the number of **inversions** between MTF and OPT lists,
- Consider request to $x$ at position $p$ in MTF list.
- Of the $p-1$ items in front of $x$, say that $k$ are also in front of $x$ in the OPT list. The remaining $p-1-k$ are behind $x$
- $Cost_{MTF} = p$ and $Cost_{OPT} \geq k+1$
- What happens to the potential?
  - When MTF moves $x$ forward, $x$ cuts in front of $k$ elements (increase $\Phi$ by $k$)
  - At the same time, the $p-1-k$ there were in front of $x$ aren't any more (decrease $\Phi$ by $p-1-k$)
  - When OPT moves $x$ forward it can only reduce $\Phi$.
  - In the end, change in potential is $\leq 2k-p+1$
  - This means that:
    $Cost_{MTF} +$ (change in potential) $\leq p+2k-p+1 \leq 2 \times Cost_{OPT}$

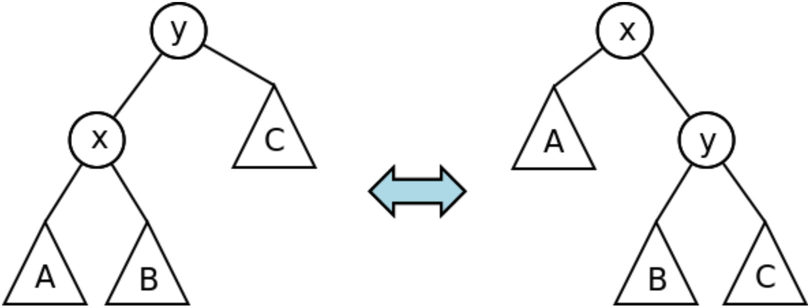
□

# Splay Trees

- A **self-adjusting binary search tree**
- They were invented by **D. Sleator** and **R. Tarjan** in **1985**
- The **key ideas** are similar to self-organizing linked lists:
  - accessed items are moved to the root
  - recently accessed elements are quick to access again
- It provides guarantees of logarithmic access time in **amortized sense**

- Consider the following "rotations" designed to move a node to the root of a (sub)tree:

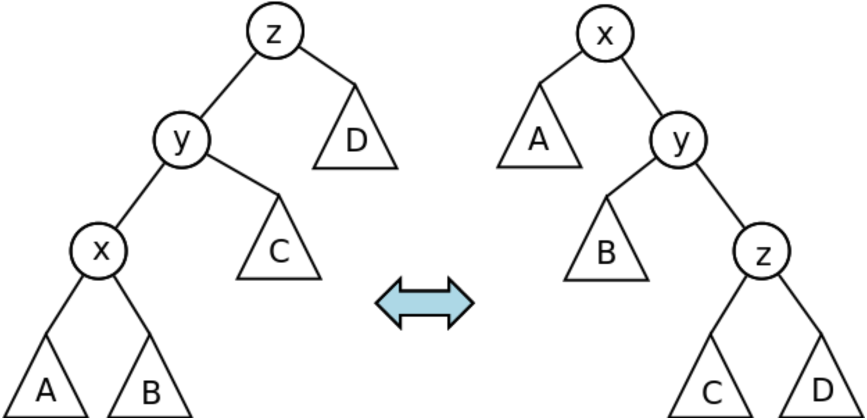
**Zig** (or **Zag**) - Simple Rotation
*(also used in AVL and red-black trees)*



# Trees and Rotations

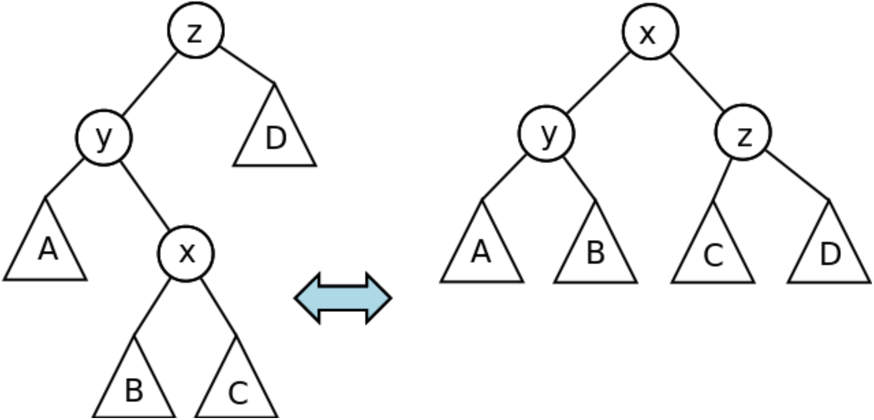
- Consider the following "rotations" designed to move a node to the root of a (sub)tree:

**Zig-Zig** (or **Zag-Zag**)



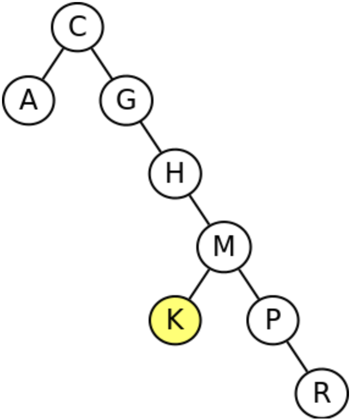
- Consider the following "rotations" designed to move a node to the root of a (sub)tree:

**Zig-Zag** (or **Zag-Zig**)



# Splay Operation

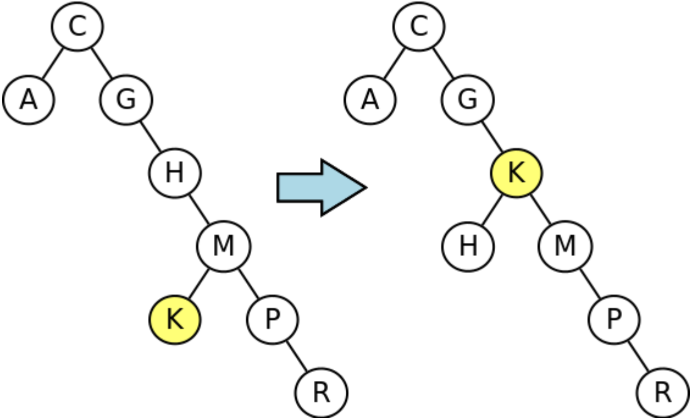
- Splaying a node means moving it to the root of a tree using the operations given before:




**Original tree**

# Splay Operation

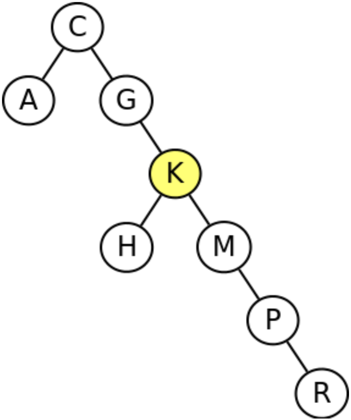
- Splaying a node means moving it to the root of a tree using the operations given before:




**Zig-Zag Left (or Zag-Zig)**

# Splay Operation

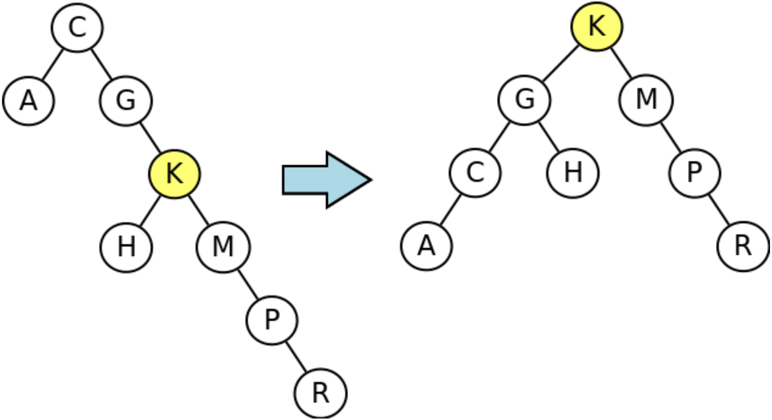
- Splaying a node means moving it to the root of a tree using the operations given before:




Now the tree is like this

# Splay Operation

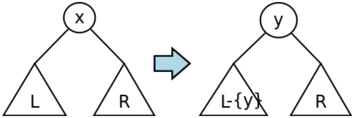
- Splaying a node means moving it to the root of a tree using the operations given before:




**Zig-Zig Left (or Zag-Zag)**

# Operations on a Splay Tree

- **Idea:** do as in a normal BST but in the end splay the node
  - **find($x$):** do as in BST and then splay $x$
    (if $x$ is not present splay the last node accessed)
  - **insert($x$):** do as in BST and then splay $x$
  - **remove(x):** find $x$, splay $x$, delete $x$ (leaves its subtress $R$ and $L$ "detached"), find largest element $y$ in $L$ and make it the new root:




- Running time is **dominated** by the splay operation.

# Why do splay trees work in practice?

**Efficiency of splay trees**

For any sequence of $m$ operations on a splay tree, the running time is $\mathcal{O}(m \log n)$, where $n$ is the max number of nodes in the tree at any time.

- **Intuition:** any operation on a deeper side of the tree will "bring" nodes from that side closer to the root
  - It is possible to make a splay tree have $\Theta(n)$ height, and hence a splay applied to the lowest leaf will take $\Theta(n)$ time. However, the resulting splayed tree will have an average node depth roughly decreased by half!
- **Two quantities: real cost** and **increase in balance**
  - If we spend much, then we will also be balancing a lot
  - If don't balance a lot, than we also did not spend much

# Amortized Analysis of Splay Trees

- The key is defining the right potential function $\Phi$
- Consider the following:
  - $size(x)$ = number of nodes below $x$ (including $x$)
  - $rank(x) = \log_2(size(x))$
  - $\Phi(S) = \sum_x rank(x)$
- Our **potential function** is the sum of the ranks of all tree nodes
- Let the **cost** be the number of rotations

**Lemma**

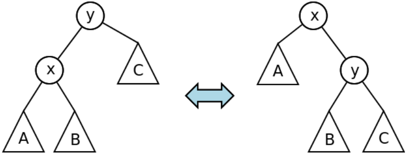
The amortized time of splaying node $x$ in a tree with root $r$ is at most $3(rank(r) - rank(x)) + 1$

- The rank of a single node is at most $\log n$ and therefore the above means the amortized time per operation is $\mathcal{O}(\log n)$

# Amortized Analysis of Splay Trees

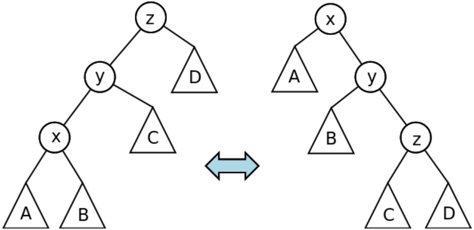
- If $x$ is at the root, the bound is trivially achieved
- If not, we will have a sequence of zig-zig and zig-zag rotations, followed by at most one simple rotation at the top
- Let $r(x)$ be the the rank of $x$ before the rotation and $r'(x)$ be its rank afterwards.
- We will show that a simple rotation takes time at most $3(r'(x) - r(x)) + 1$ and that the other operations take $3(r'(x) - r(x))$
- If you think about the sequence of rotations, than successive $r(x)$ and $r'(x)$ will cancel out and we are left at the end with $3(r(root) - r(x)) + 1$
- The worst case is $r(x) = 0$ and in that case we have $3 \times \log_2 n + 1$

**Case 1: Simple Rotation**




- Only $x$ and $y$ change rank
  - $x$ increases rank
  - $y$ decreases rank
- Cost is $1 + r'(x) + r'(y) - r(x) - r(y)$
- This is $\leq 1 + r'(x) - r(x)$ since $r(y) \geq r'(y)$
- This is $\leq 1 + 3(r'(x) - r(x))$ since $r'(x) \geq r(x)$

**Case 2: Zig-Zig Operation**



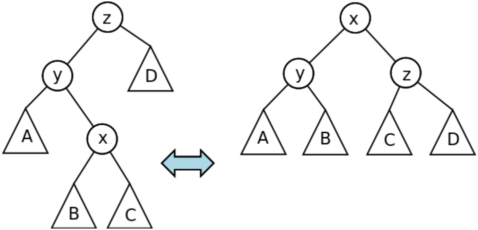

- Only $x$, $y$ and $z$ change rank
- Cost is $2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)$
- This is $= 2 + r'(y) + r'(z) - r(x) - r(y)$ since $r'(x) = r(z)$
- This is $\leq 2 + r'(x) + r'(z) - 2r(x)$ since $r'(x) \geq r'(y)$ and $r(y) \geq r(x)$

# Amortized Analysis of Splay Trees

**Case 2: Zig-Zig Operation**

- $2 + r'(x) + r'(z) - 2r(x)$ is at most $3(r'(x) - r(x))$
- This is equivalent to say that $2r'(x) - r(x) - r'(z) \geq 2$
- $2r'(x) - r(x) - r'(z) = \log_2(s'(x)/s(x)) + \log_2(s'(x)/s'(z))$
- Notice that $s'(x) \geq s(x) + s'(z)$
- Given that log is convex, the way to make the two logarithms as small as possible is to choose $s(x) = s(z) = s'(x)/2$. In that case $\log_2 2 + \log_2 2 = 1 + 1 = 2$ and we have proved what we wanted!

**Case 3: Zig-Zag Operation**




- Only $x$, $y$ and $z$ change rank
- Cost is $2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)$
- This is $= 2 + r'(y) + r'(z) - r(x) - r(y)$ since $r'(x) = r(z)$
- This is $\leq 2 + r'(y) + r'(z) - 2r(x)$ since $r(y) \geq r(x)$

**Case 3: Zig-Zag Operation**

- $2 + r'(y) + r'(z) - 2r(x)$ is at most $2(r'(x) - r(x))$
- This is equivalent to say that $2r'(x) - r'(y) - r'(z) \geq 2$
- $2r'(x) - r'(y) - r'(z) = \log_2(s'(x)/s'(y)) + \log_2(s'(x)/s'(z))$
- Notice that $s'(x) \geq s'(y) + s'(z)$
- By the same argument a before, the way to minimize is to choose $s'(y) = s'(z) = s'(x)/2$. In that case $\log_2 2 + \log_2 2 = 1 + 1 = 2$
- Obviously, $2(r'(x) - r(x)) < 3(r'(x) - r(x))$

$\square$