Lecture 2: Amortized Analysis & Splay Trees

Rafael Oliveira

University of Waterloo
Cheriton School of Computer Science

rafael.oliveira.teaching@gmail.com

May 13, 2021

Overview

Words of Wisdom

- Twenty years from now you will be more disappointed by the things you didn't do than by the ones you did do. So throw off the bowlines. Sail away from the safe harbor. Catch the trade winds in your sails. Explore. Dream. Discover.

  - Mark Twain

[1] He literally said: "Man lebt nur einmal!"

Words of Wisdom

- Twenty years from now you will be more disappointed by the things you didn't do than by the ones you did do. So throw off the bowlines. Sail away from the safe harbor. Catch the trade winds in your sails. Explore. Dream. Discover.

  - Mark Twain

- In short:

  YOLO

  - Johann Wolfgang von Goethe (1774)
  - Johann Strauss II (1855)[1]

[1] He literally said: "Man lebt nur einmal!"

Recap - Why Amortized Analysis?

In **amortized analysis**, one averages the *total time* required to perform a sequence of data-structure operations over *all operations performed*.

Upshot of amortized analysis: worst-case cost *per query* may be high for one particular query, so long as overall average cost per query is small in the end!

Remark

Amortized analysis is a *worst-case* analysis. That is, it measures the average performance of each operation in the worst case.

Remark

Data structures with great amortized running time are great for internal processes, such as *internal graph algorithms* (e.g. min spanning tree). It is bad when you have client-server model (i.e., internet-related things), as in this setting one wants to minimize worst-case *per query*.

Recap - Types of amortized analyses

Three common types of amortized analyses:

Recap - Types of amortized analyses

Three common types of amortized analyses:

1. **Aggregate Analysis:** determine upper bound $T(n)$ on total cost of sequence of n operations. So amortized complexity is $T(n)/n$.
2. **Accounting Method:** assign certain *charge* to each operation (independent of the actual cost of the operation). If operation is cheaper than the charge, then build up credit to use later.

Recap - Types of amortized analyses

Three common types of amortized analyses:

1. **Aggregate Analysis:** determine upper bound $T(n)$ on total cost of sequence of n operations. So amortized complexity is $T(n)/n$.
2. **Accounting Method:** assign certain *charge* to each operation (independent of the actual cost of the operation). If operation is cheaper than the charge, then build up credit to use later.
3. **Potential Method:** one comes up with *potential energy* of a data structure, which maps each state of entire data-structure to a real number (its "potential"). Differs from accounting method because we assign credit to the data structure as a whole, instead of assigning credit to each operation.

Why Splay Trees?

Binary search trees:

- extremely useful data structures (pervasive in computer science/industry)
- worst-case running time per operation $\Theta(\text{height})$
- Need technique to balance height.
- Different implementations: red-black trees [CLRS 2009, Chapter 13], AVL trees [CLRS 2009, Exercise 13-3] and many others (see [CLRS 2009, Chapter notes of ch. 13].
- All these implementations are quite involved, require extra information per node (i.e. more memory) and difficult to analyze.

Why Splay Trees?

Binary search trees:

- extremely useful data structures (pervasive in computer science/industry)
- worst-case running time per operation $\Theta(\text{height})$
- Need technique to balance height.
- Different implementations: red-black trees [CLRS 2009, Chapter 13], AVL trees [CLRS 2009, Exercise 13-3] and many others (see [CLRS 2009, Chapter notes of ch. 13].
- All these implementations are quite involved, require extra information per node (i.e. more memory) and difficult to analyze.

Splay trees are:

- Easier to implement
- don't keep any balance info!

Splay Trees (self-adjusting binary trees)

Theorem ([Sleator & Tarjan 1985])

Splay trees have $\Theta(\log n)$ *amortized cost per op.,* $\Theta(n)$ *worst-case time.*

Splay Trees (self-adjusting binary trees)

Theorem ([Sleator & Tarjan 1985])

Splay trees have $\Theta(\log n)$ amortized cost per op., $\Theta(n)$ worst-case time.

- We will not keep any balancing info

Splay Trees (self-adjusting binary trees)

Theorem ([Sleator & Tarjan 1985])

Splay trees have $\Theta(\log n)$ amortized cost per op., $\Theta(n)$ worst-case time.

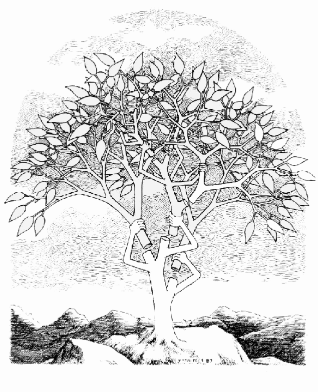
- We will not keep any balancing info
- Main idea: adjust the tree whenever a node is accessed (giving rise to name "self-adjusting trees")

Splay Trees (self-adjusting binary trees)

Theorem ([Sleator & Tarjan 1985])

Splay trees have $\Theta(\log n)$ amortized cost per op., $\Theta(n)$ worst-case time.

- We will not keep any balancing info
- Main idea: adjust the tree whenever a node is accessed (giving rise to name "self-adjusting trees")



A Self-Adjusting Search Tree

- Introduction
  - Types of amortized analyses
  - Splay Trees
- Implementing Splay-Trees
  - Setup
  - Splay Rotations
  - Analysis
- Conclusion & Open Problems
- Acknowledgements

Naive approach

How to adjust tree to get good amortized bounds?

Naive approach

How to adjust tree to get good amortized bounds?

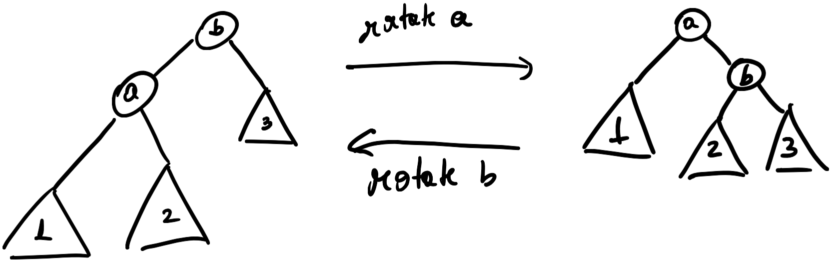
Idea (Splaying): every time we search some node, imagine this will be a "popular node" and move it up to the root. Moving a node to the root is called *splaying* the node.

Naive approach

How to adjust tree to get good amortized bounds?

Idea (Splaying): every time we search some node, imagine this will be a "popular node" and move it up to the root. Moving a node to the root is called *splaying* the node.

Naive Idea: perform [single] rotations to move the searched node to the root.



Naive approach

How to adjust tree to get good amortized bounds?

Idea (Splaying): every time we search some node, imagine this will be a "popular node" and move it up to the root. Moving a node to the root is called *splaying* the node.

Naive Idea: perform [single] rotations to move the searched node to the root.

This is not good. In exercises you will show that this gives amortized search cost of $\Omega(n)$.

Naive approach

How to adjust tree to get good amortized bounds?

Idea (Splaying): every time we search some node, imagine this will be a "popular node" and move it up to the root. Moving a node to the root is called *splaying* the node.

Naive Idea: perform [single] rotations to move the searched node to the root.

This is not good. In exercises you will show that this gives amortized search cost of $\Omega(n)$.

How do we fix this? By adding different kinds of rotations!

Setup

Notation:

- $n \leftarrow$ number of elements (we denote the elements by $1, 2, \ldots, n$)
- $m \leftarrow$ number of operations. That is

$$m = (\#\ \text{searches}) + (\#\ \text{insertions}) + (\#\ \text{deletions})$$

Setup

Notation:

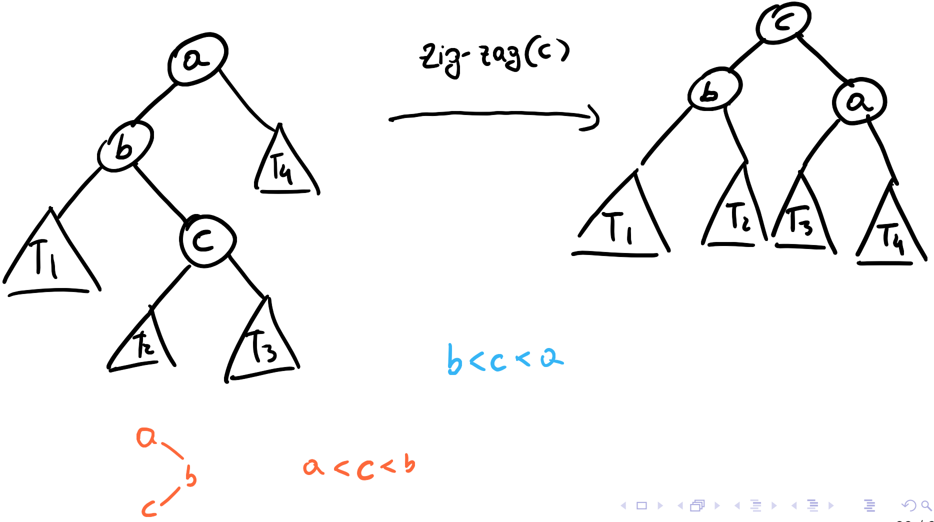
- $n \leftarrow$ number of elements (we denote the elements by $1, 2, \ldots, n$)
- $m \leftarrow$ number of operations. That is

$$m = (\# \text{ searches}) + (\# \text{ insertions}) + (\# \text{ deletions})$$

- $SEARCH(k) \leftarrow$ find whether element k is in tree
- $INSERT(k) \leftarrow$ insert element k in our tree
- $DELETE(k) \leftarrow$ delete element k from our tree

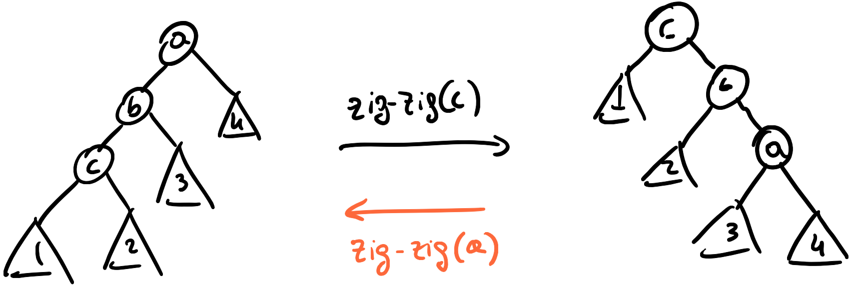
Splay Operation

Rotation type 1: *zig-zag rotations*



Splay Operation (continued)

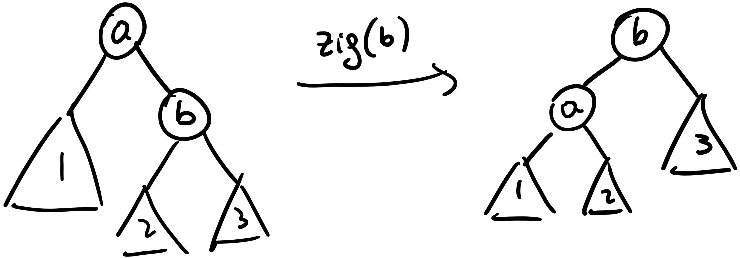
Rotation type 2: *zig-zig rotations*



Splay Operation (continued)

Rotation type 3: *normal rotations (zigs)*

(this will only be used if node is child of root)
(having no grandparent)



Splay Operation (continued)

Definition (SPLAY operation)

SPLAY(k)

- **Input:** element k
- **Output:** "rebalancing of the binary search tree"

a new binary search tree with k as the root.

Splay Operation (continued)

Definition (SPLAY operation)

$SPLAY(k)$

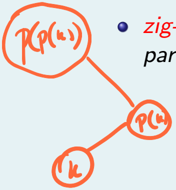
- **Input:** element k
- **Output:** "rebalancing of the binary search tree"
- Repeat until k is the root of the tree:

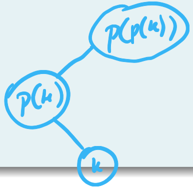
Splay Operation (continued)

Definition (SPLAY operation)

$SPLAY(k)$

- **Input:** element k
- **Output:** "rebalancing of the binary search tree"
- Repeat until k is the root of the tree:
  - If node of k in tree satisfies the zig-zag condition, perform zig-zag rotation.
    - *zig-zag condition:* $parent(k)$ has k as left-child (right child) and $parent(parent(k))$ has $parent(k)$ as right-child (left child)



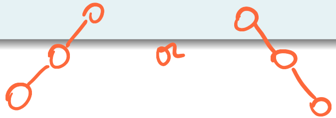


Splay Operation (continued)

Definition (SPLAY operation)

$SPLAY(k)$

- **Input:** element k
- **Output:** "rebalancing of the binary search tree"
- Repeat until k is the root of the tree:
  - If node of k in tree satisfies the zig-zag condition, perform zig-zag rotation.
    - *zig-zag condition:* $parent(k)$ has k as left-child (right child) and $parent(parent(k))$ has $parent(k)$ as right-child (left child)
  - If node of k in tree satisfies the zig-zig condition, perform zig-zig rotation.
    - *zig-zig condition:* $parent(k)$ has k as left-child (right child) and $parent(parent(k))$ has $parent(k)$ as left-child (right child)



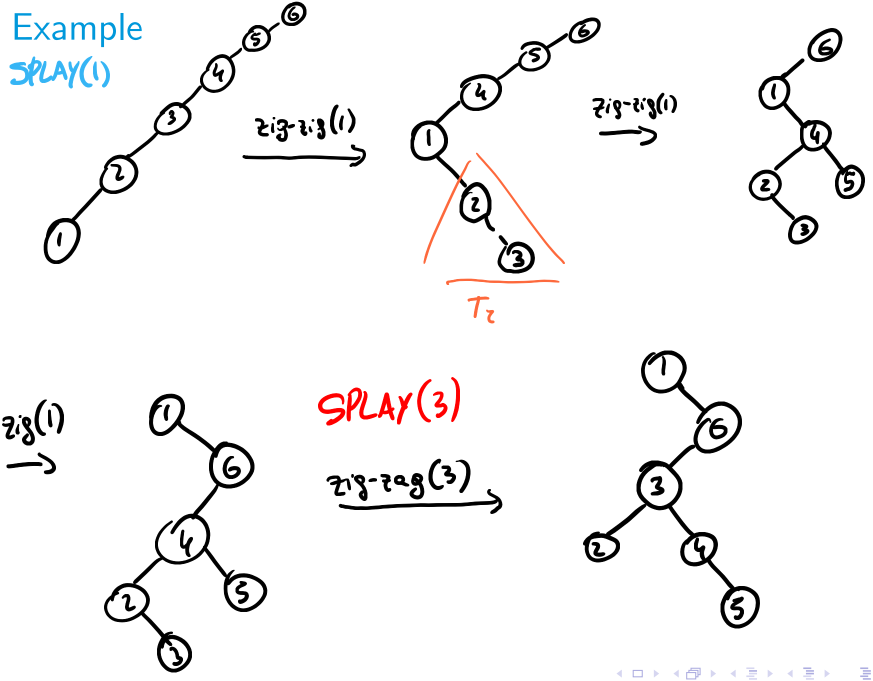
Splay Operation (continued)

Definition (SPLAY operation)

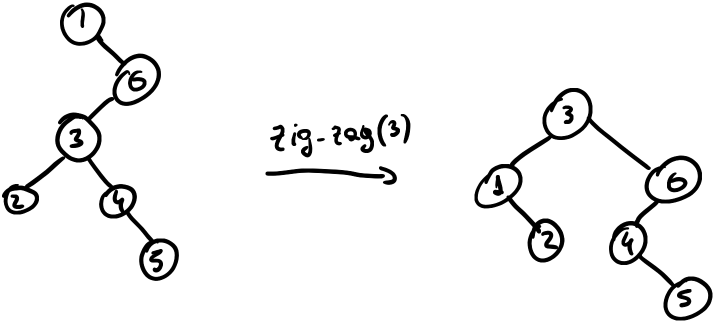
$SPLAY(k)$

- **Input:** element k
- **Output:** "rebalancing of the binary search tree"
- Repeat until k is the root of the tree:
  - If node of k in tree satisfies the zig-zag condition, perform zig-zag rotation.
    - *zig-zag condition:* $parent(k)$ has k as left-child (right child) and $parent(parent(k))$ has $parent(k)$ as right-child (left child)
  - If node of k in tree satisfies the zig-zig condition, perform zig-zig rotation.
    - *zig-zig condition:* $parent(k)$ has k as left-child (right child) and $parent(parent(k))$ has $parent(k)$ as left-child (right child)
  - If node of k in tree is a child of the root, perform normal rotation (zig).

Example



Example (continued)



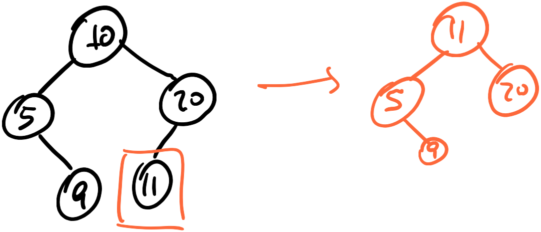
Intuition: zig-zig and zig-zag make a lot of progress in balanced trees

Splay Tree Algorithm

Input: set of elements $\{1, 2, \dots, n\}$

Output: at each step, a binary-search tree data structure and the answer to the query being asked.

1. $SEARCH(k) \rightarrow$ after searching for k, if k in the tree, do $SPLAY(k)$
2. $INSERT(k) \rightarrow$ standard insert operation, then do $SPLAY(k)$
3. $DELETE(k) \rightarrow$ standard delete operation, then $SPLAY(parent(k))$
   - delete first "moves k to the bottom of tree (by finding successor)
   - then delete k as in the cases where k has at most one child
   - then we splay the parent of k (after we place k at the bottom)
   - see [CLRS 2009, Chapter 12] for a recap


10
5
20
9
11
11
5
20
9

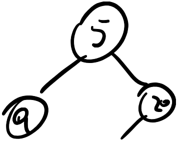

5
9
20

Figure: Is that it?

Analysis - Potential Method

We will use for the analysis the *potential method*.

Analysis - Potential Method

We will use for the analysis the *potential method*.

In the potential method, we assign a *potential function* Φ which maps each *data structure* D to a *real number* $\Phi(D)$, which is potential associated with data structure D.

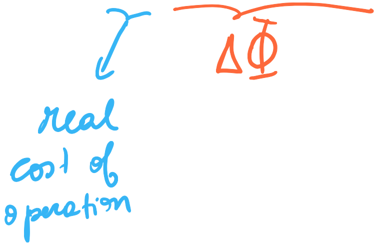
Analysis - Potential Method

We will use for the analysis the *potential method*.

In the potential method, we assign a *potential function* Φ which maps each *data structure* D to a *real number* $\Phi(D)$, which is potential associated with data structure D.

The *charge* $\hat{c}_i$ of the i^{th} operation with respect to the potential function Φ is:

$$\hat{c}_i := c_i + \Phi(D_i) - \Phi(D_{i-1})$$



Analysis - Potential Method

We will use for the analysis the *potential method*.

In the potential method, we assign a *potential function* Φ which maps each *data structure* D to a *real number* $\Phi(D)$, which is potential associated with data structure D.

The *charge* $\hat{c}_i$ of the i^{th} operation with respect to the potential function Φ is:

$$\hat{c}_i := c_i + \Phi(D_i) - \Phi(D_{i-1})$$

The *amortized cost* of all operations is

$$\sum_{i=1}^{m} \hat{c}_i = \sum_{i=1}^{m} c_i + \Phi(D_i) - \Phi(D_{i-1})$$

$$= \Phi(D_m) - \Phi(D_0) + \sum_{i=1}^{m} c_i$$

telescopes

charges

$\Delta\Phi_{final}$

total actual cost

Analysis - Potential Method

We will use for the analysis the *potential method*.

In the potential method, we assign a *potential function* Φ which maps each *data structure* D to a *real number* $\Phi(D)$, which is potential associated with data structure D.

The *charge* $\hat{c}_i$ of the i^{th} operation with respect to the potential function Φ is:

$$\hat{c}_i := c_i + \Phi(D_i) - \Phi(D_{i-1})$$

The *amortized cost* of all operations is

$$\sum_{i=1}^{m} \hat{c}_i = \sum_{i=1}^{m} c_i + \Phi(D_i) - \Phi(D_{i-1})$$

$$= \Phi(D_m) - \Phi(D_0) + \sum_{i=1}^{m} c_i$$

So long as $\Phi(D_m) \geq \Phi(D_0)$ then amortized charge is an upper bound on amortized cost.

Potential Function

Definition (Potential Function)

- $\delta(k) :=$ number of descendants of k (including k)

Potential Function

Definition (Potential Function)

- $\delta(k) :=$ number of descendants of k (including k)
- $\text{rank}(k) := \log(\delta(k))$

Potential Function

Definition (Potential Function)

- $\delta(k) :=$ number of descendants of k (including k)
- $\text{rank}(k) := \log(\delta(k))$
- $$\Phi(T) = \sum_{k \in T} \text{rank}(k)$$

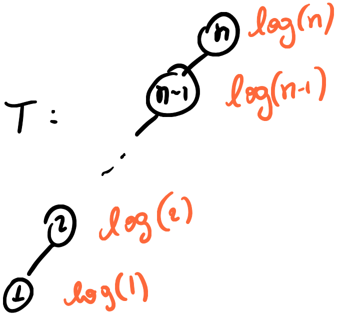
If a node is far from the root, splay is expensive but potential will pay for it (potential accounts for how balanced a tree is)

Potential Function

Definition (Potential Function)

- $\delta(k)$:= number of descendants of k (including k)
- $\text{rank}(k) := \log(\delta(k))$
- $$\Phi(T) = \sum_{k \in T} \text{rank}(k)$$

Examples (max potential):



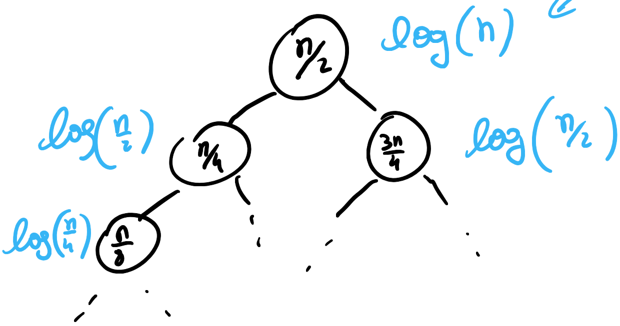
super unbalanced tree

$$\Phi(T) = \sum_{i=1}^{n} \log(i) = O(n \log n)$$

Example - min potential

perfectly balanced tree

height is potential of node



$=: T$

$$\Phi(T) = \sum_{h=1}^{\log(n)} h \cdot (\#\text{nodes of height } h) = \sum_{h=1}^{\log n} h \cdot \frac{n}{2^h} = O(n)$$

Splay Tree Algorithm - Recap

Input: set of elements $\{1, 2, \ldots, n\}$

Output: at each step, a binary-search tree data structure and the answer to the query being asked.

1. $SEARCH(k) \rightarrow$ after searching for k, if k in the tree, do $SPLAY(k)$
2. $INSERT(k) \rightarrow$ standard insert operation, then do $SPLAY(k)$
3. $DELETE(k) \rightarrow$ standard delete operation, then $SPLAY(parent(k))$

Analysis - Splay operation

Let $\text{rank}(k)$ be the current rank of k and $\text{rank}'(k)$ be the new rank of k after we perform a rotation on k.

Analysis - Splay operation

Let $\text{rank}(k)$ be the current rank of k and $\text{rank}'(k)$ be the new rank of k after we perform a rotation on k.

Lemma (Charge from SPLAY Subroutines)

The charge $\hat{c}$ of an operation (zig, zig-zig, zig-zag) is bounded by:

$$\hat{c} \leq \begin{cases} 3 \cdot (\text{rank}'(k) - \text{rank}(k)) & \textit{for zig-zig, zig-zag} \\ 3 \cdot (\text{rank}'(k) - \text{rank}(k)) + 1 & \textit{for zig} \end{cases}$$

Analysis - Splay operation

Let $\text{rank}(k)$ be the current rank of k and $\text{rank}'(k)$ be the new rank of k after we perform a rotation on k.

Lemma (Charge from SPLAY Subroutines)

The charge $\hat{c}$ of an operation (zig, zig-zig, zig-zag) is bounded by:

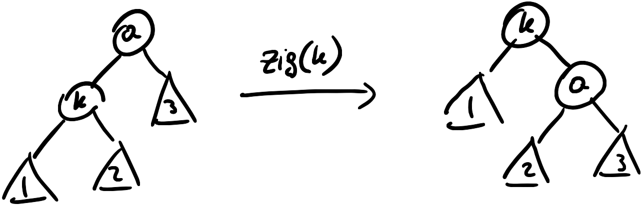
$$\hat{c} \leq \begin{cases} 3 \cdot (\text{rank}'(k) - \text{rank}(k)) & \textit{for zig-zig, zig-zag} \\ 3 \cdot (\text{rank}'(k) - \text{rank}(k)) + 1 & \textit{for zig} \end{cases}$$

Lemma (Total Cost of $SPLAY(k)$)

Let T be our current tree, with root t and k be a node in this tree. The charge of $SPLAY(k)$ is

$$\leq 3 \cdot (\text{rank}(t) - \text{rank}(k)) + 1 \leq 3 \cdot \text{rank}(t) + 1 = O(\log n)$$

Proof of First Lemma (charge to zig)



T $\qquad$ T'

$$rank'(k) = rank(a)$$

$$rank'(a) \le rank'(k)$$

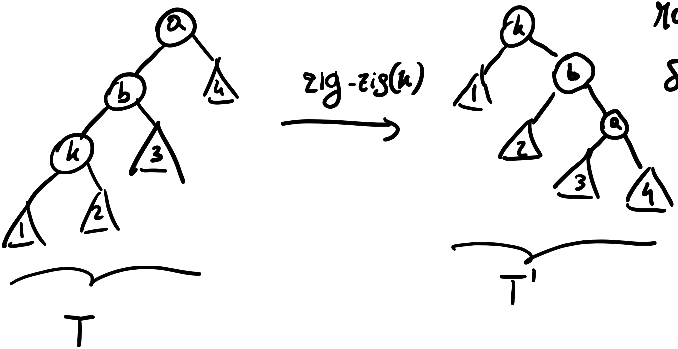
charge: $(\text{cost of op.}) + \Phi(T') - \Phi(T)$

$$= 1 + \cancel{rank'(k)} + rank'(a) - rank(k) - \cancel{rank(a)}$$

$$= 1 + rank'(a) - rank(k) \le 1 + rank'(k) - rank(k)$$

$$\le 1 + 3(rank'(k) - rank(k))$$

Proof of First Lemma (charge to zig-zig)



$\text{rank}'(k) = \text{rank}(a)$

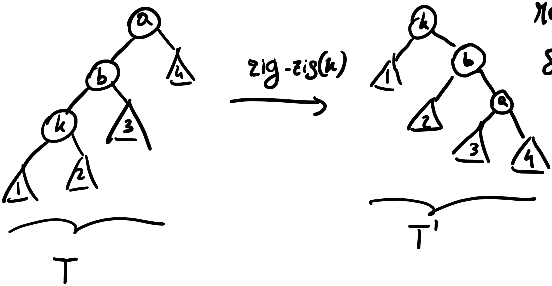
$\delta'(k) \geq \delta'(a) + \delta(k)$

charge: $(\text{cost of operation}) + \Phi(T') - \Phi(T)$

$= 2 + \text{rank}'(a) + \text{rank}'(b) + \cancel{\text{rank}'(k)} - \cancel{\text{rank}(a)} - \text{rank}(b) - \text{rank}(k)$

$= 2 + \text{rank}'(a) + \text{rank}'(b) - \text{rank}(b) - \text{rank}(k)$

Proof of First Lemma (charge to zig-zig)



$\text{rank}'(k) = \text{rank}(a)$

$\delta'(k) \geq \delta'(a) + \delta(k)$ $(*)$

$\text{rank}'(b) \leq \text{rank}'(k)$

$\text{rank}(b) \geq \text{rank}(k)$

$$\text{charge} = 2 + \text{rank}'(a) + \text{rank}'(b) - \text{rank}(b) - \text{rank}(k)$$

$$\leq 2 + \text{rank}'(a) + \text{rank}'(k) - 2\,\text{rank}(k)$$

$$\delta'(a) + \delta(k) \leq \delta'(k) \Rightarrow \log\left(\frac{\delta'(a)}{\delta'(k)}\right) + \log\left(\frac{\delta(k)}{\delta'(k)}\right) \leq -2$$

concavity of log

$$\Rightarrow \text{rank}'(a) + \text{rank}(k) \leq 2\,\text{rank}'(k) - 2 \Rightarrow \text{charge} \leq 3(\text{rank}'(k) - \text{rank}(k))$$

Proof of Second Lemma (total charge of *SPLAY*(*k*))

T is our tree, t root, k element we are splaying

the entire change of potential = sum of all splay rotations

$\gamma_i \leftarrow$ charge of i^{th} SPLAY rotation

$rank^{(i)}(k) \leftarrow$ rank of k after the i^{th} SPLAY rotation

OBS: $rank^{(0)}(k) = rank(k)$ and $rank^{(f)}(k) = rank(t)$

charge of SPLAY(k) : $\sum_{i=1}^{f} \gamma_i \leq 1 + \sum_{i=1}^{f} 3\left(rank^{(i)}(k) - rank^{(i-1)}(k)\right)$

$\leq 1 + 3\left(rank^{(f)}(k) - rank^{(0)}(k)\right) = 1 + 3\left(rank(t) - rank(k)\right)$ ☒

$\leq 1 + 3\log(n)$

Analysis - Amortized cost

1. For each operation (INSERT, SEARCH, DELETE) we have:

$$\text{(charge per operation)} = \text{(charge of SPLAY)} + \text{(potential change \textit{not} from SPLAY)}$$

Analysis - Amortized cost

1. For each operation (INSERT, SEARCH, DELETE) we have:

$$\begin{aligned}(\text{charge per operation}) = {} & (\text{charge of SPLAY}) \\ & + (\text{potential change } \textcolor{red}{\textit{not}} \text{ from SPLAY})\end{aligned}$$

2. (charge of SPLAY) $= O(\log n)$ (by second lemma)
3. charge of SPLAY already includes the cost of the operation

cost to traverse the tree

Analysis - Amortized cost

1. For each operation (INSERT, SEARCH, DELETE) we have:

$$\text{(charge per operation)} = \text{(charge of SPLAY)} + \text{(potential change } \textit{not} \text{ from SPLAY)}$$

2. (charge of SPLAY) $= O(\log n)$ (by second lemma)
3. charge of SPLAY already includes the cost of the operation
4. Tracking potential change outside splay:

Analysis - Amortized cost

1. For each operation (INSERT, SEARCH, DELETE) we have:

$$\text{(charge per operation)} = \text{(charge of SPLAY)} + \text{(potential change } \textit{not} \text{ from SPLAY)}$$

2. (charge of SPLAY) $= O(\log n)$ (by second lemma)
3. charge of SPLAY already includes the cost of the operation
4. Tracking potential change outside splay:
    1. *SEARCH* $\rightarrow$ only splay changes the potential

Analysis - Amortized cost

1. For each operation (INSERT, SEARCH, DELETE) we have:

$$\begin{aligned}\text{(charge per operation)} &= \text{(charge of SPLAY)} \\ &\quad + \text{(potential change } \textit{not} \text{ from SPLAY)}\end{aligned}$$

2. (charge of SPLAY) $= O(\log n)$ (by second lemma)
3. charge of SPLAY already includes the cost of the operation
4. Tracking potential change outside splay:
    1. *SEARCH* $\rightarrow$ only splay changes the potential
    2. *DELETE* $\rightarrow$ removing a node decreases potential

Analysis - Amortized cost

1. For each operation (INSERT, SEARCH, DELETE) we have:

$$\text{(charge per operation)} = \text{(charge of SPLAY)} + \text{(potential change } \textit{not} \text{ from SPLAY)}$$

2. (charge of SPLAY) $= O(\log n)$ (by second lemma)
3. charge of SPLAY already includes the cost of the operation
4. Tracking potential change outside splay:
    1. *SEARCH* $\rightarrow$ only splay changes the potential
    2. *DELETE* $\rightarrow$ removing a node decreases potential
    3. *INSERT* $\rightarrow$ adding new element k increases ranks of all ancestors of k post insertion (might be $O(n)$ of them)

Handling INSERT potential

Let us check the potential change after an insert:

Let $k = k_0 \to k_1 \to k_2 \to \cdots \to k_\ell = \text{root}$

path from k to root after INSERT(k).

$\delta'(a)$ = new # descendants

$\delta(a)$ = old # descendants

Reminder: in BST whenever we insert, we insert in a leaf of the new tree.

$\delta'(k_i) = \delta(k_i) + 1 \qquad 0 \le i \le \ell$

$\therefore$ change in potential $\sum_{i=1}^{\ell} (\text{rank}'(k_i) - \text{rank}(k_i)) = \sum_{i=1}^{\ell} \log\left(\frac{\delta(k_i)+1}{\delta(k_i)}\right)$

$$\le \sum_{i=1}^{n} \log\left(\frac{i+1}{i}\right) = O(\log(n))$$

Final Analysis:

Q: why is this a valid potential scheme?

A: potential is always ≥ 0, initial potential $= 0$ (empty tree)

$$\therefore \underbrace{\sum_{i=1}^{m} \hat{c}_i} = \sum_{i=1}^{m} c_i + \underbrace{\Phi(T_m)}_{\geq 0} - \underbrace{\Phi(\text{empty tree})}_{= 0}$$

charge per operation: $\underbrace{(\text{charge of SPLAY (includes the cost of op)})}_{O(\log(n))} + \underbrace{(\text{potential change not from SPLAY})}_{\text{insertion } \leq O(\log n)}$

$\therefore$ total charge: $\leq O(m \cdot \log(n))$ $\therefore$ amortized time $O(\log n)$.

- Introduction
  - Types of amortized analyses
  - Splay Trees
- Implementing Splay-Trees
  - Setup
  - Splay Rotations
  - Analysis
- Conclusion & Open Problems
- Acknowledgements

After Learning Splay Trees

Figure: You to whoever taught you red-black trees

Conclusion

- Splay trees gives us a fairly *simple algorithm* to balance a tree
- Great amortized cost!

$$O(\log n) \quad \text{per operation}$$

- Analysis is very clever (yet principled!)
- Remember: this only works in the amortized setting (may be very bad for client-server model for instance)

Dynamic Optimality Conjecture

Open Question ([Sleator & Tarjan 1985])

Splay Trees are optimal (within a constant) in a very strong sense:

Given a sequence of items to search for $a_1, \ldots, a_m$*, let* OPT *be the minimum cost of doing these searches + any rotations you like on the binary search tree.*
You can charge 1 for following tree pointer (parent $\rightarrow$ *child or child* $\rightarrow$ *parent), charge 1 per rotation.*

Conjecture: *Cost of splay tree is* $O(OPT)$*.*

Note that for OPT, you get to look at the sequence of searches first and plan ahead. (we will cover this in more detail in the online algorithms part of the course)
Also, OPT can adjust the tree so it's even better than the static optimal binary search trees you may have seen in CS 341.

Acknowledgement

- Lecture based largely on Anna Lubiw's notes. See her notes at https://www.student.cs.uwaterloo.ca/~cs466/Lectures/Lecture4.pdf
- Picutre of self-adjusting tree taken from Robert Tarjan's website

References I

Sleator, Daniel and Tarjan, Robert (1985)
Self-adjusting binary search trees.
J. Assoc. Comput. Mach. 32(3), 652 – 686

Cormen, Thomas and Leiserson, Charles and Rivest, Ronald and Stein, Clifford. (2009)
Introduction to Algorithms, third edition.
MIT Press