Lecture 2: Amortized Analysis & Splay Trees

Rafael Oliveira

University of Waterloo
Cheriton School of Computer Science

rafael.oliveira.teaching@gmail.com

September 14, 2020

Overview

- Introduction
  - Meet your TAs!
  - Types of amortized analyses
  - Splay Trees
- Implementing Splay-Trees
  - Setup
  - Splay Rotations
  - Analysis
- Acknowledgements

Meet your TAs

Thi Xuan Vu

Office hours: ~~[illegible] morning (put exact time here)~~
Tuesdays at 1pm (EDT)

Anubhav Srivastava

Office hours: Thursday afternoon ~~(put exact time here)~~
3 pm (EDT)

Admin notes

- **Late homework policy:** I updated the late homework policy to be more flexible. Now each student has *10 late days without penalty* for the entire term.

Admin notes

- **Late homework policy:** I updated the late homework policy to be more flexible. Now each student has *10 late days without penalty* for the entire term.
- **With regards to the final project:** I highly encourage you all to explore an open problem (but survey is also completely fine! :)). The reason I wanted to mention is that this may be a unique opportunity for many of you to explore! So be bold! :) If you solve an open problem, you also automatically get 100 in this course and get to publish a paper! (and also get to experience the exhilarating feeling of solving a cool problem!)

Admin notes

- **Late homework policy:** I updated the late homework policy to be more flexible. Now each student has *10 late days without penalty* for the entire term.
- **With regards to the final project:** I highly encourage you all to explore an open problem (but survey is also completely fine! :)). The reason I wanted to mention is that this may be a unique opportunity for many of you to explore! So be bold! :) If you solve an open problem, you also automatically get 100 in this course and get to publish a paper! (and also get to experience the exhilarating feeling of solving a cool problem!)
- 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

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 Θ(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.

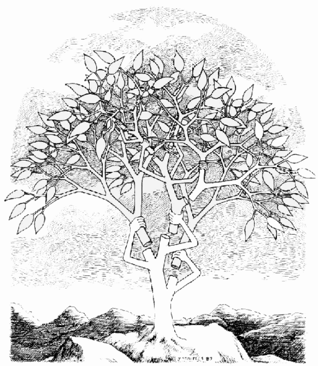
- We will not keep any balancing info
- Main idea: adjust the tree whenever a node is accessed (giving rise to name "self-adjusting trees")

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
  - Meet your TAs!
  - Types of amortized analyses
  - Splay Trees
- Implementing Splay-Trees
  - Setup
  - Splay Rotations
  - Analysis
- Acknowledgements

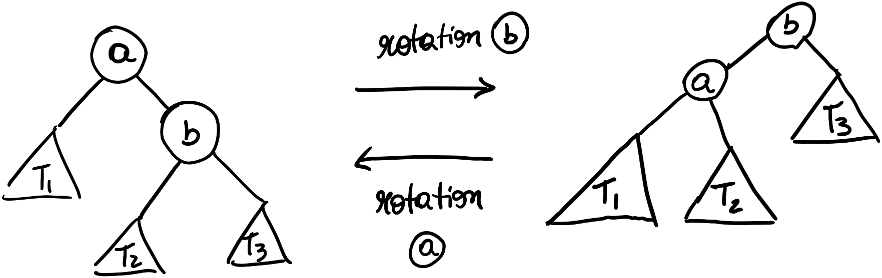
Naive approach

How to adjust tree to get good amortized bounds?

Naive approach

How to adjust tree to get good amortized bounds?

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



Naive approach

How to adjust tree to get good amortized bounds?

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?

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:

Setup

Notation:

- $n \leftarrow$ number of elements (we denote the elements by $1, 2, \ldots, n$)

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:

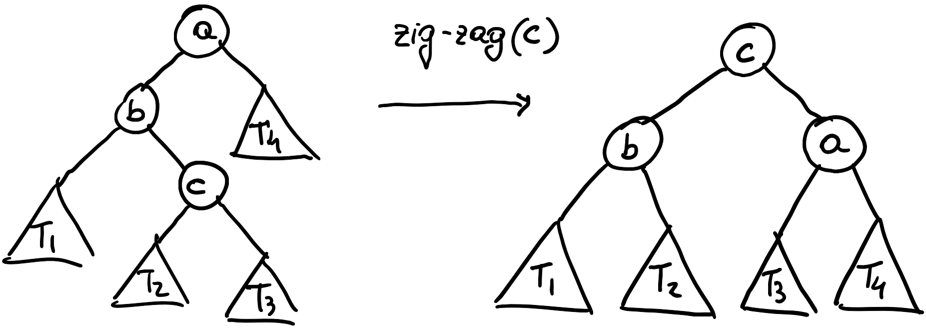
- $n \leftarrow$ number of elements (we denote the elements by $1, 2, \ldots, n$)
- $m \leftarrow$ number of operations. That is

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

- $SEARCH(k) \leftarrow$ 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

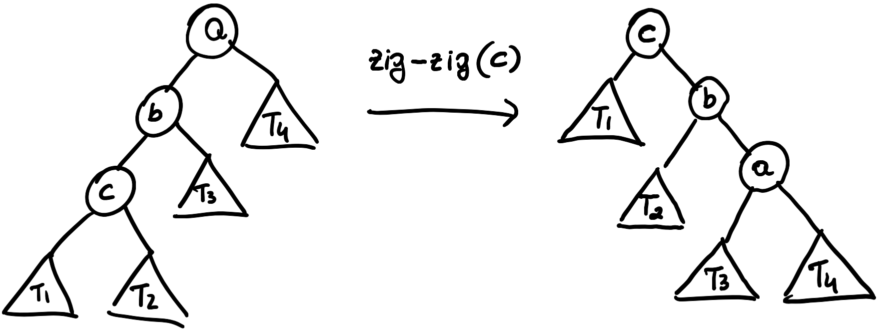
Splay Operation

Rotation type 1: *zig-zag rotations*



Splay Operation (continued)

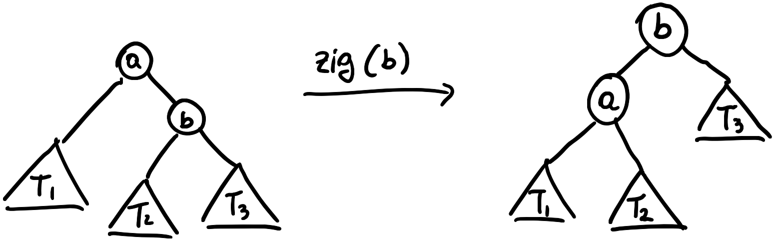
Rotation type 2: *zig-zig rotations*



Splay Operation (continued)

Rotation type 3: *normal rotations (zigs)*

(this is whenever our node is child of the root)



Splay Operation (continued)

Definition (SPLAY operation)

$SPLAY(k)$

Splay Operation (continued)

Definition (SPLAY operation)

$SPLAY(k)$

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

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)

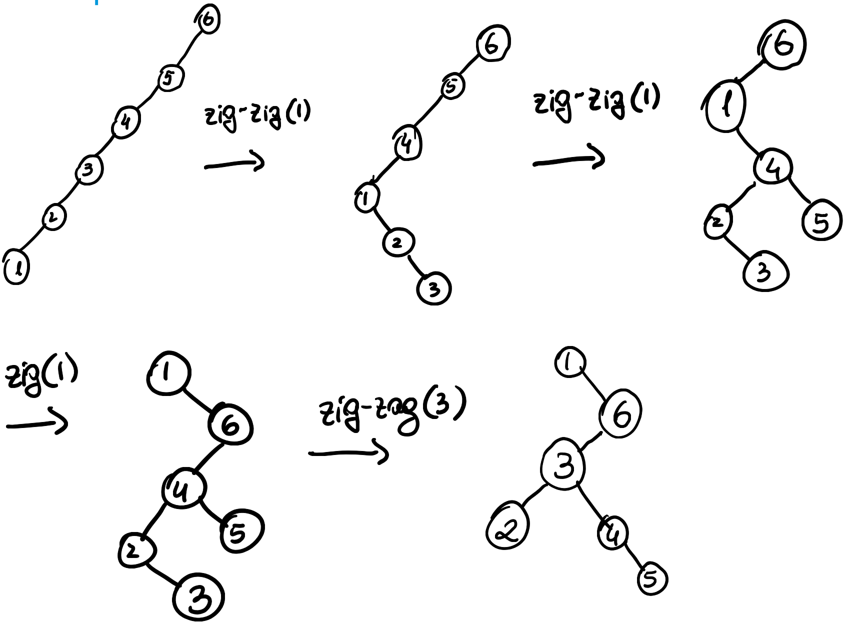
Splay Operation (continued)

Definition (SPLAY operation)

$SPLAY(k)$

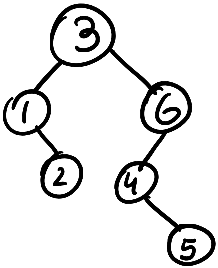
- **Input:** element k
- **Output:** "rebalancing of the binary search tree"
- Repeat until k is the root of the tree:
  - If node of k in tree satisfies the zig-zag condition, perform zig-zag rotation.
    - *zig-zag condition:* $parent(k)$ has k as left-child (right child) and $parent(parent(k))$ has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

zig-zag(3)



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

Splay Tree Algorithm

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 - 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

$$\begin{aligned}\sum_{i=1}^{m} \hat{c}_i &= \sum_{i=1}^{m} c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= \underbrace{\Phi(D_m)}_{\text{final potential}} - \underbrace{\Phi(D_0)}_{\text{initial potential}} + \overbrace{\sum_{i=1}^{m} c_i}^{\text{actual cost}}\end{aligned}$$

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

$$\begin{aligned}\sum_{i=1}^{m} \hat{c}_i &= \sum_{i=1}^{m} c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= \Phi(D_m) - \Phi(D_0) + \sum_{i=1}^{m} c_i\end{aligned}$$

So long as final potential ($\Phi(D_m)$) greater than or equal to initial potential ($\Phi(D_0)$) 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)
- $\text{rank}(k) := \log(\delta(k))$
- $$\Phi(T) = \sum_{k \in T} \text{rank}(k)$$

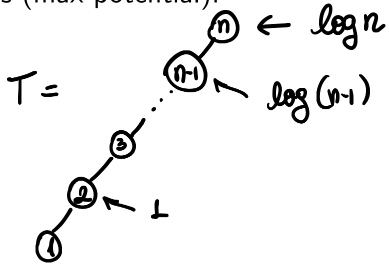
If node is far from root, splay is expensive but potential will pay for it (potential account for how balanced a tree is).

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):



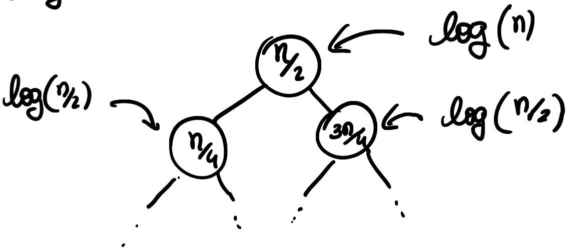
unbalanced tree

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

$$= O(n \log n)$$

Example - min potential

perfectly balanced tree



$$\Phi(T) = \sum_{h=0}^{logn} 2^h \log\left(\frac{n}{2^h}\right) = O(n)$$

Assuming j=logn – h, the sum reduces to $n\sum_{j=0}^{logn} j2^{-j}$

But $\sum_{j=0}^{\infty} j2^{-j}$ = 2

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 (Potential Change from SPLAY Subroutines)

The charge c of an operation (zig, zig-zig, zig-zag) is bounded by:

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

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 (Potential Change from SPLAY Subroutines)

The charge c of an operation (zig, zig-zig, zig-zag) is bounded by:

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

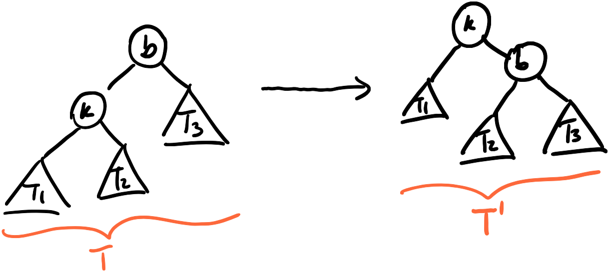
Lemma (Total Cost of $SPLAY(k)$)

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 (potential change from SPLAY subroutine)

Regular rotation (zig):



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

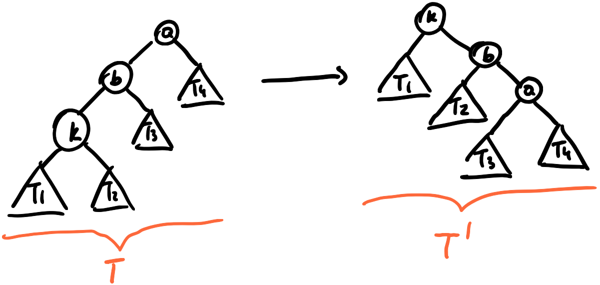
$$\text{charge} = \text{cost} + \phi(T') - \phi(T)$$

$$= \underbrace{1}_{\text{rotation}} + \underbrace{\cancel{\text{rank}'(k)} + \text{rank}'(b) - \text{rank}(k) - \cancel{\text{rank}(b)}}_{\text{change in potential}}$$

$$= 1 + \text{rank}'(b) - \text{rank}(k) \underset{k \text{ is parent of } b \text{ in } T'}{\leq} 1 + \text{rank}'(k) - \text{rank}(k) \leq 1 + 3(\text{rank}'(k) - \text{rank}(k))$$

Proof of First Lemma (potential change from SPLAY subroutine)

Zig-zig:



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

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

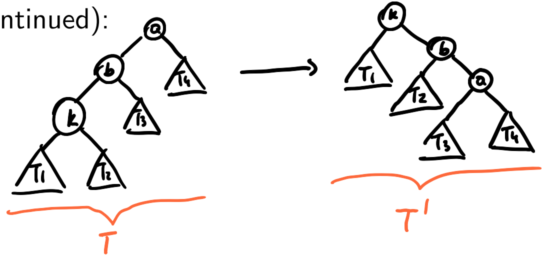
$$\text{charge} = (\text{cost of rotations}) + (\text{change in potential})$$

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

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

Proof of First Lemma (potential change from SPLAY subroutine)

Zig-zig (continued):



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

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

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

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

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

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

concavity of log.

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

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

$\Rightarrow \text{charge} \leq 3(\text{rank}'(k) - \text{rank}(k))$.

(Same proof for zig-zag)

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

T is our tree, t its root, k the element we want to SPLAY.

Let's add up all charges from all SPLAY operations (zig/ zig-zag/ zig-zig)

$\gamma_i \leftarrow$ charge from i^{th} SPLAY op.

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

OBS: $rank^{(0)}(k) = rank(k)$, $rank^{(\ell)}(k) = rank(t)$ (final rank of k).

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

by lemma 1, each $\gamma_i \leq 3\left(rank^{(i)}(k) - rank^{(i-1)}(k)\right)$ (for zig-zig, zig-zag) and at most one operation is zig (hence the +1 out side the summand)

$$\leq 1 + 3\left(rank^{(\ell)}(k) - rank^{(0)}(k)\right) = 1 + 3\left(rank(t) - rank(k)\right) \qquad \blacksquare$$

Analysis - Amortized cost

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

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

Analysis - Amortized cost

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

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

2. (charge of SPLAY) $= O(\log n)$ (by second lemma)

Analysis - Amortized cost

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

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

2. (charge of SPLAY) $= O(\log n)$ (by second lemma)
3. (cost of operation) $\leq$ (charge of SPLAY) (walking down tree)

Analysis - Amortized cost

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

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

2. (charge of SPLAY) $= O(\log n)$ (by second lemma)
3. (cost of operation) $\leq$ (charge of SPLAY) (walking down tree)
4. Tracking potential change outside splay:

Analysis - Amortized cost

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

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

2. (charge of SPLAY) $= O(\log n)$ (by second lemma)
3. (cost of operation) $\leq$ (charge of SPLAY) (walking down tree)
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:

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

2. (charge of SPLAY) $= O(\log n)$ (by second lemma)
3. (cost of operation) $\leq$ (charge of SPLAY) (walking down tree)
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:

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

2. (charge of SPLAY) $= O(\log n)$ (by second lemma)
3. (cost of operation) $\leq$ (charge of SPLAY) (walking down tree)
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) need to handle this

Handling INSERT potential

Let us check the potential change after an insert:

Adding element increases potential of all ancestors.

Let $k = k_0 \to k_1 \to k_2 \to \cdots \to k_\ell = \text{root}$ be the path from k to root after INSERT(k), $\begin{cases} \delta'(a) = \text{new \# descendants} \\ \delta(a) = \text{old \# descendants} \end{cases}$

Reminder: when we insert a node in our tree, the node becomes a leaf of the new tree.

Thus we have:

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

$\therefore$ change in potential:

$$\sum_{i=0}^{\ell} \text{rank}'(k_i) - \sum_{i=1}^{\ell} \text{rank}(k_i)$$

Handling INSERT potential change (contd.)

$\sum_{i=0}^{l}$ rank'(k_i)- $\sum_{i=1}^{l}$ rank(k_i)

= rank'(k_0) + $\sum_{i=1}^{l}$(rank'(k_i) - rank(k_i)) = $\sum_{i=1}^{l}$(rank'(k_i) - rank(k_i))

As rank'(k_0) = log 1 = 0

$\delta(k_i) + 1 \leq \delta(k_{i+1})$

$\delta'(k_i) = \delta(k_i) + 1 \leq \delta(k_{i+1})$

$\Rightarrow$ log(δ'(k_i)) ≤ log(δ(k_{i+1})) $\Rightarrow$ rank'(k_i) ≤ rank(k_{i+1})

So, $\sum_{i=1}^{l}$(rank'(k_i) - rank(k_i)) ≤ rank'(k_l) + $\sum_{i=1}^{l-1}$(rank(k_{i+1}) - rank(k_i))

[ignoring rank(k_l)]

= rank'(k_l) + rank(k_l) – rank(k_1) $\leq$ 2rank'(k_l) – rank(k_1)

$\leq$ 2rank'(k_l) – rank'(k_0) $\leq$ 2rank'(k_l) $= 2\log(n+1) = O(logn)$

Final Analysis:

Q: why is this a valid potential scheme?

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

$$\therefore \sum \hat{c}_i = \sum c_i + \underbrace{\Phi_{final}}_{\geq 0} - \underbrace{\Phi_0}_{= 0} \quad ✓$$

$$\text{charge per operation} = \underbrace{\text{charge of SPLAY}}_{O(\log n)} + \underbrace{\text{cost of operation}}_{\substack{\leq \text{ cost of SPLAY} \\ \text{(walking down tree)}}} + \underbrace{\text{potential change not from SPLAY}}_{\substack{\text{only insert} \\ \leq \log n}}$$

$$= O(\log n)$$

$$\therefore \text{ total charge} = (\#\text{ operations}) \cdot (\text{charge per operation})$$

$$= O(m \cdot \log n)$$

$\Rightarrow$ amortized cost $O(\log n)$. $\boxtimes$

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