Lecture 2: Amortized Analysis & Splay Trees

Rafael Oliveira

University of Waterloo
Cheriton School of Computer Science

rafael.oliveira.teaching@gmail.com

May 18, 2023

Overview

- Introduction
    - Types of amortized analyses
    - Splay Trees

- Implementing Splay-Trees
    - Setup
    - Splay Rotations
    - Analysis

- Conclusion & Open Problems

- Acknowledgements

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 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.

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.*

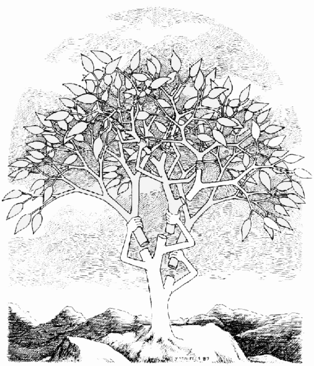
- We will not keep any balancing info
- Main idea: adjust the tree whenever a node is accessed (giving rise to name “self-adjusting trees”)

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)*

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)

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 $SPLAY(k)$. If k not in tree, do $SPLAY(k')$ where k' is the last node seen in the traversal
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 (and correct implementation)

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* γ_i of the i^{th} operation with respect to the potential function Φ is:

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

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* γ_i of the i^{th} operation with respect to the potential function Φ is:

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

The *amortized cost* of all operations is

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

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* γ_i of the i^{th} operation with respect to the potential function Φ is:

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

The *amortized cost* of all operations is

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

So long as $\Phi(D_m) \geq \Phi(D_0)$ 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)
- $\text{rank}(k) := \log(\delta(k))$
- $$\Phi(T) = \sum_{k \in T} \text{rank}(k)$$

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):

Example - min potential

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 rank(k) be the current rank of k and rank′(k) be the new rank of k after we perform a rotation on k.

Lemma (Amortized cost from SPLAY Subroutines)

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

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

Analysis - Splay operation

Let rank(k) be the current rank of k and rank′(k) be the new rank of k after we perform a rotation on k.

Lemma (Amortized cost from SPLAY Subroutines)

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

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

Lemma (Total Amortized 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

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

Proof of First Lemma (charge to zig)

Proof of First Lemma (charge to zig-zig)

Proof of First Lemma (charge to zig-zig)

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

Analysis - Amortized cost

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

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

---

[1] Charge of SPLAY already has the cost of traversing the tree and the cost of performing SPLAY and the change in potential coming from the SPLAY operation accounted for.

Analysis - Amortized cost

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

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

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

---

[1] Charge of SPLAY already has the cost of traversing the tree and the cost of performing SPLAY and the change in potential coming from the SPLAY operation accounted for.

Analysis - Amortized cost

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

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

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] Charge of SPLAY already has the cost of traversing the tree and the cost of performing SPLAY and the change in potential coming from the SPLAY operation accounted for.

Analysis - Amortized cost

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

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

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* → only splay changes the potential

---

[1] Charge of SPLAY already has the cost of traversing the tree and the cost of performing SPLAY and the change in potential coming from the SPLAY operation accounted for.

Analysis - Amortized cost

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

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

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* → only splay changes the potential
    2. *DELETE* → removing a node decreases potential

---

[1] Charge of SPLAY already has the cost of traversing the tree and the cost of performing SPLAY and the change in potential coming from the SPLAY operation accounted for.

Analysis - Amortized cost

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

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

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* → only splay changes the potential
    2. *DELETE* → removing a node decreases potential
    3. *INSERT* → adding new element k increases ranks of all ancestors of k post insertion (might be $O(n)$ of them)

---

[1] Charge of SPLAY already has the cost of traversing the tree and the cost of performing SPLAY and the change in potential coming from the SPLAY operation accounted for.

Handling INSERT potential

Let us check the potential change after an insert:

Final Analysis

- 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!

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

- 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 → child or child → 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