Splay Trees

Disadvantage of balanced search trees:
- worst case; no advantage for easy inputs
- additional memory required
- complicated implementation

Splay Trees:
+ after access, an element is moved to the root; splay(\(x\)) repeated accesses are faster
- only amortized guarantee
- read-operations change the tree

find(\(x\))
- search for \(x\) according to a search tree
- let \(\bar{x}\) be last element on search-path
- splay(\(\bar{x}\))

insert(\(x\))
- search for \(x\); \(\bar{x}\) is last visited element during search (successor or predecessor of \(x\))
- splay(\(\bar{x}\)) moves \(\bar{x}\) to the root
- insert \(x\) as new root

The illustration shows the case when \(\bar{x}\) is the predecessor of \(x\).

delete(\(x\))
- search for \(x\); splay(\(x\)); remove \(x\)
- search largest element \(\bar{x}\) in \(A\)
- splay(\(\bar{x}\)) (on subtree \(A\))
- connect root of \(B\) as right child of \(\bar{x}\)
Move to Root

How to bring element to root?
- one (bad) option: \( \text{moveToRoot}(x) \)
- iteratively do rotation around parent of \( x \) until \( x \) is root
- if \( x \) is left child do right rotation otw. left rotation

Splay: Zig Case

better option \( \text{splay}(x) \):
- zig case: if \( x \) is child of root do left rotation or right rotation around parent

Splay: Zigzag Case

better option \( \text{splay}(x) \):
- zigzag case: if \( x \) is right child and parent of \( x \) is left child (or \( x \) left child parent of \( x \) right child)
- do double right rotation around grand-parent (resp. double left rotation)

Double Rotations
**Splay: Zigzag Case**

![Diagram showing splay: Zigzag Case](image)

**Better option splay(x):**

- **Zigzag case:** if \( x \) is left child and parent of \( x \) is left child (or \( x \) right child, parent of \( x \) right child)
- Do right rotation around grand-parent followed by right rotation around parent (resp. left rotations)

---

**Splay vs. Move to Root**

Input tree on which splay(\( x \)) and moveToRoot(\( x \)) is executed.

---

**Result after moveToRoot(\( x \)).**

---

**Result after splay(\( x \)).**

---

**Splay vs. Move to Root**

Input tree on which splay(\( x \)) and moveToRoot(\( x \)) is executed.

---

**Result after moveToRoot(\( x \)).**

---

**Result after splay(\( x \)).**
Static Optimality

Suppose we have a sequence of \( m \) find-operations. \( \text{find}(x) \) appears \( h_x \) times in this sequence.

The cost of a static search tree \( T \) is:

\[
\text{cost}(T) = m + \sum_x h_x \text{depth}_T(x)
\]

The total cost for processing the sequence on a splay-tree is \( \mathcal{O}(\text{cost}(T_{\text{min}})) \), where \( T_{\text{min}} \) is an optimal static search tree.

Dynamic Optimality

Let \( S \) be a sequence with \( m \) find-operations.

Let \( A \) be a data-structure based on a search tree:

- the cost for accessing element \( x \) is \( 1 + \text{depth}(x) \);
- after accessing \( x \) the tree may be re-arranged through rotations;

**Conjecture:**
A splay tree that only contains elements from \( S \) has cost \( \mathcal{O}(\text{cost}(A,S)) \), for processing \( S \).

Amortized Analysis

**Lemma 1**
Splay Trees have an amortized running time of \( \mathcal{O}(\log n) \) for all operations.

**Definition 2**
A data structure with operations \( \text{op}_1(), \ldots, \text{op}_k() \) has amortized running times \( t_1, \ldots, t_k \) for these operations if the following holds.

Suppose you are given a sequence of operations (starting with an empty data-structure) that operate on at most \( n \) elements, and let \( k_i \) denote the number of occurrences of \( \text{op}_i() \) within this sequence. Then the actual running time must be at most \( \sum_i k_i \cdot t_i(n) \).
Potential Method

Introduce a potential for the data structure.
- $\Phi(D_i)$ is the potential after the $i$-th operation.
- Amortized cost of the $i$-th operation is
  \[ \hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) . \]
- Show that $\Phi(D_i) \geq \Phi(D_0)$.

Then
\[
\sum_{i=1}^{k} c_i \leq \sum_{i=1}^{k} c_i + \Phi(D_k) - \Phi(D_0) = \sum_{i=1}^{k} \hat{c}_i
\]
This means the amortized costs can be used to derive a bound on the total cost.

Example: Stack

Use potential function $\Phi(S) =$ number of elements on the stack.

Amortized cost:
- $S$. push(): cost
  \[ \hat{C}_{\text{push}} = C_{\text{push}} + \Delta \Phi = 1 + 1 \leq 2 . \]
- $S$. pop(): cost
  \[ \hat{C}_{\text{pop}} = C_{\text{pop}} + \Delta \Phi = 1 - 1 \leq 0 . \]
- $S$. multipop($k$): cost
  \[ \hat{C}_{\text{mp}} = C_{\text{mp}} + \Delta \Phi = \min\{\text{size}, k\} - \min\{\text{size}, k\} \leq 0 . \]

Note that the analysis becomes wrong if pop() or multipop() are called on an empty stack.

Example: Binary Counter

Incrementing a binary counter:
Consider a computational model where each bit-operation costs one time-unit.

Incrementing an $n$-bit binary counter may require to examine $n$-bits, and maybe change them.

Actual cost:
- Changing bit from 0 to 1: cost 1.
- Changing bit from 1 to 0: cost 1.
- Increment: cost is $k + 1$, where $k$ is the number of consecutive ones in the least significant bit-positions (e.g, 001101 has $k = 1$).
Example: Binary Counter

Choose potential function $\Phi(x) = k$, where $k$ denotes the number of ones in the binary representation of $x$.

**Amortized cost:**

- Changing bit from 0 to 1:
  \[
  \hat{C}_{0\rightarrow1} = C_{0\rightarrow1} + \Delta\Phi = 1 + 1 \leq 2.
  \]

- Changing bit from 1 to 0:
  \[
  \hat{C}_{1\rightarrow0} = C_{1\rightarrow0} + \Delta\Phi = 1 - 1 \leq 0.
  \]

- Increment: Let $k$ denotes the number of consecutive ones in the least significant bit-positions. An increment involves $k$ ($1 \rightarrow 0$)-operations, and one ($0 \rightarrow 1$)-operation.

Hence, the amortized cost is $k\hat{C}_{1\rightarrow0} + \hat{C}_{0\rightarrow1} \leq 2$.

---

Splay Trees

**Potential function for splay trees:**

- $s(x) = |T_x|$ (size $s$)
- $r(x) = \log_2(s(x))$ (rank $r$)
- $\Phi(T) = \sum_{v \in T} r(v)$ (potential change $\Phi$)

**Amortized cost** = real cost + potential change

The cost is essentially the cost of the splay-operation, which is 1 plus the number of rotations.

---

Splay: Zig Case

\[\Delta\Phi = r'(x) + r'(p) - r(x) - r(p)\]
\[\leq r'(x) - r(x)\]

\[\text{cost}_{\text{zig}} \leq 1 + 3(r'(x) - r(x))\]

Splay: Zigzig Case

\[\Delta\Phi = r'(x) + r'(p) + r'(g) - r(x) - r(p) - r(g)\]
\[\leq r'(x) + r'(g) - r(x) - r(x)\]
\[\leq r'(x) + r'(g) + r(x) - 3r'(x) + 3r'(x) - r(x) - 2r(x)\]
\[\leq -2 + 3(r'(x) - r(x))\]

\[\Rightarrow \text{cost}_{\text{zigzig}} \leq 3(r'(x) - r(x))\]
Splay: Zigzig Case

\[
\frac{1}{2}(r(x) + r'(g) - 2r'(x)) \\
= \frac{1}{2}\left( \log(s(x)) + \log(s'(g)) - 2 \log(s'(x)) \right) \\
= \frac{1}{2} \log\left( \frac{s(x)}{s'(x)} \right) + \frac{1}{2} \log\left( \frac{s'(g)}{s'(x)} \right) \\
\leq \log\left( \frac{1}{2} \frac{s(x)}{s'(x)} + \frac{1}{2} \frac{s'(g)}{s'(x)} \right) \leq \log \left( \frac{1}{2} \right) = -1
\]

The last inequality holds because \( \log \) is a concave function.

Splay: Zigzag Case

\[
\Delta \Phi = r'(p) + r'(g) - r(x) - r(p) - r(g) \\
= r'(p) + r'(g) - r(x) - r(p) \\
\leq r'(p) + r'(g) - r(x) - r(x) \\
= r'(p) + r'(g) - 2r'(x) + 2r'(x) - 2r(x) \\
\leq -2 + 2(r'(x) - r(x)) \quad \Rightarrow \text{cost}_{\text{zigzag}} \leq 3(r'(x) - r(x))
\]

Amortized cost of the whole splay operation:

\[
\leq 1 + 1 + \sum_{\text{steps } t} 3(r_t(x) - r_{t-1}(x)) \\
= 2 + r(\text{root}) - r_0(x) \\
\leq O(\log n)
\]

7.3 Splay Trees

Ernst Mayr, Harald Räcke

The first one is added due to the fact that so far for each step of a splay-operation we have only counted the number of rotations, but the cost is \(1 + \#\text{rotations}\).

The second one comes from the zig-operation. Note that we have at most one zig-operation during a splay.
Splay Trees

Bibliography

7.3 Splay Trees