CS 240 – Data Structures and Data Management

Module 4E: Dictionaries - Enriched

A. Hunt A. Jamshidpey O. Veksler

Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Winter 2023

version *2023-01-31 14:59*

Outline

4. More Balanced Search Trees
   - Balanced binary search trees
   - Scapegoat Trees
   - Amortized analysis
   - Analysis of scapegoat trees

Outline

4 More Balanced Search Trees

- Balanced binary search trees
- Scapegoat Trees
- Amortized analysis
- Analysis of scapegoat trees

Overview of balanced binary search trees

We will see numerous variants of binary search trees.
The operations then have the following run-times:

- $\Theta(\log n)$ worst-case time (**AVL-trees**)
- $\Theta(\log n)$ amortized time (**Scapegoat trees**) and no rotations.
- $\Theta(\log n)$ expected time (**Treaps**)
- $\Theta(\log n)$ expected time (**Skip lists**) and space is smaller. (It's not even a tree.)
- $\Theta(\log n)$ amortized time (**Splay trees**) and space is smaller, and can handle biased requests.

(We will see "rotations", "amortized" and "biased requests" later.)

Overview of balanced binary search trees

General strategy for balanced binary search tree:

- Use a binary search tree, but impose *structural condition*
- Argue that structural condition implies $O(\log n)$ ... height (where ... might be worst-case / avg-case / expected)
- With this, search takes $O(\log n)$ ... time
- Explain how to do insert and delete so that structural condition continues to hold.
  - This must be done so that run-time stays $O(\text{height})$
  - With this, insert/delete takes $O(\log n)$ ... time

Outline

4. More Balanced Search Trees
   - Balanced binary search trees
   - Scapegoat Trees
   - Amortized analysis
   - Analysis of scapegoat trees

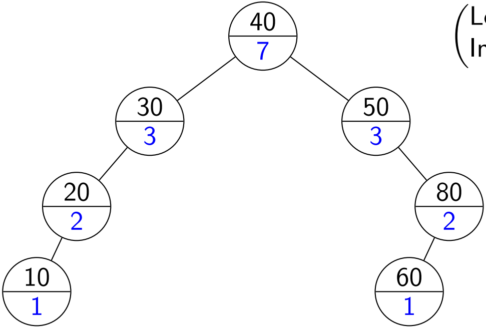
Scapegoat trees

- Can we have balanced binary search trees *without* rotations? (A later application will need such a tree.)
- This sounds impossible—we must sometimes restructure the tree.
- **Idea:** Rather than doing a small local change, occasionally do a large (near-global) rebuilt.
- With the right setup, this will lead to $O(\log n)$ height and $O(\log n)$ *amortized* time for all operations.

Scapegoat trees

Fix a constant α with $\frac{1}{2} < \alpha < 1$. A **scapegoat tree** is a binary search tree where any node v with a parent satisfies

$$v.size \leq \alpha \cdot v.parent.size.$$



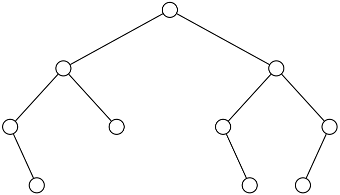
(Lower number = subtree-size. In our examples, $\alpha = \frac{2}{3}$.)

- $v.size$ needed during updates $\rightsquigarrow$ must be stored
- Any subtree is a constant fraction smaller $\rightsquigarrow$ height $O(\log n)$.

Scapegoat tree operations

- *search*: As for a binary search tree. $O(height) = O(\log n)$.
- For *insert* and *delete*, occasionally restructure a subtree into a **perfectly balanced tree**:

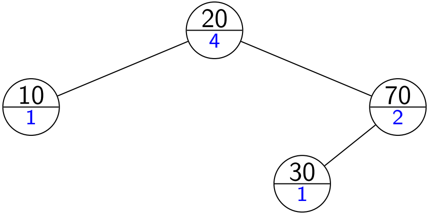
$$|size(z.left) - size(z.right)| \leq 1 \quad \text{for all nodes } z.$$



- Do this at the *highest* node where the size-condition of scapegoat trees is violated

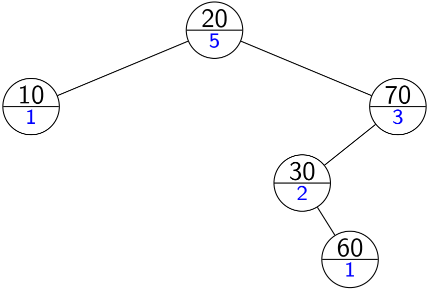
Scapegoat Tree Insertion Example

Example: *Scapegoat::insert*(60)



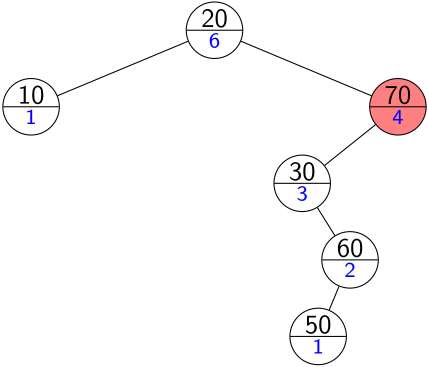
Scapegoat Tree Insertion Example

Example: *Scapegoat::insert*(60)



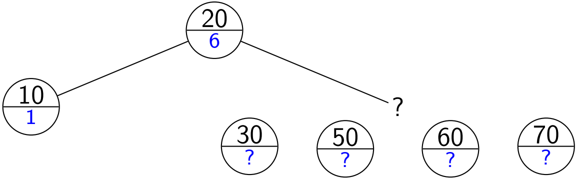
Scapegoat Tree Insertion Example

Example: *Scapegoat::insert*(50)



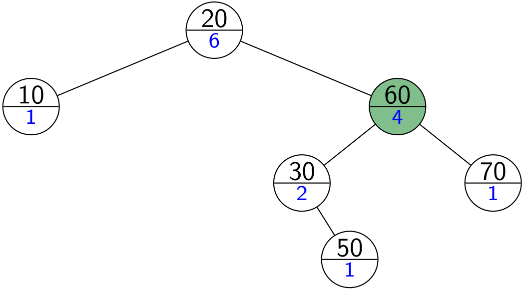
Scapegoat Tree Insertion Example

Example: *Scapegoat::insert*(50)



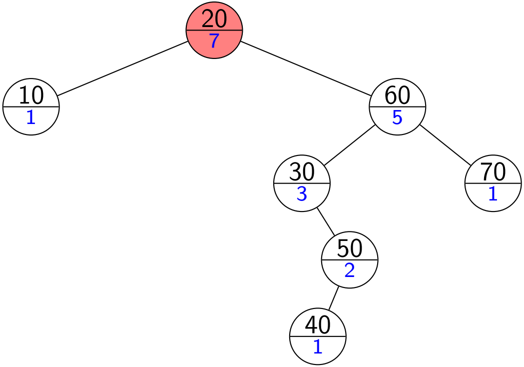
Scapegoat Tree Insertion Example

Example: *Scapegoat::insert*(50)



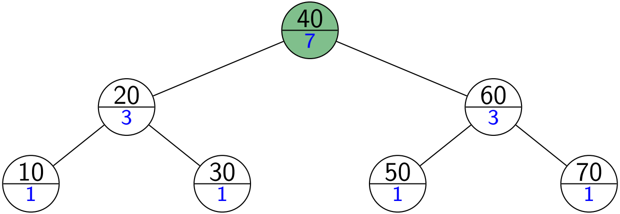
Scapegoat Tree Insertion Example

Example: *Scapegoat::insert*(40)



Scapegoat Tree Insertion Example

Example: *Scapegoat::insert*(40)



Scapegoat tree insertion

```
scapegoatTree::insert(k, v)
1.   z ← BST::insert(k, v)
2.   S ← stack initialized with z
3.   while (p ← z.parent ≠ NIL)        // update sizes, get path
4.       increase p.size
5.       S.push(p)
6.       z ← p
7.   while (S.size ≥ 2)                // size-condition violated?
8.       p ← P.pop()
9.       if (p.size < α · max{p.left.size, p.right.size})
10.          rebuild subtree at p into perfectly balanced tree
11.          return
```

- Rebuilding at p (line 10) can be done in $O(p.size)$ time (exercise).
- This restores scapegoat tree (we rebuild at the highest violation).

Outline

4. More Balanced Search Trees
   - Balanced binary search trees
   - Scapegoat Trees
   - Amortized analysis
   - Analysis of scapegoat trees

Detour: Amortized analysis

As for dynamic arrays and lazy deletion, we have the following pattern:

- usually the operation is fast,
- the occasional operation is quite slow.

The worst-case run-time bound here would not reflect that overall this works quite well.

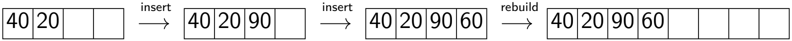
Instead, try to find an **amortized run-time bound**: A bound that holds if we add the bounds up over all operations.

$$\sum_{i=1}^{k} T^{\mathrm{actual}}(\mathcal{O}_i) \leq \sum_{i=1}^{k} T^{\mathrm{amort}}(\mathcal{O}_i).$$

(where $\mathcal{O}_1, \ldots, \mathcal{O}_k$ is any feasible sequence of operations, $T^{\mathrm{actual}}(\cdot)$ is the actual run-time, and $T^{\mathrm{amort}}(\cdot)$ is the amortized run-time (or an upper bound for it).

Detour: Amortized analysis

For dynamic arrays, some ad-hoc methods work.

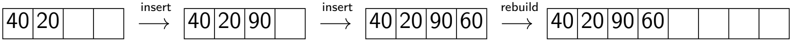


- Direct argument:
  - $n/2$ fast inserts takes $\Theta(1)$ time each.
  - Then one slow insert takes $\Theta(n)$.
  - Averaging out therefore $\Theta(1)$ per operation.

  - This is doing math with asymptotic notation - dangerous.

Detour: Amortized analysis

For dynamic arrays, some ad-hoc methods work.



- Direct argument:
  - $n/2$ fast inserts takes $\Theta(1)$ time each.
  - Then one slow insert takes $\Theta(n)$.
  - Averaging out therefore $\Theta(1)$ per operation.

  - This is doing math with asymptotic notation - dangerous.
- Explicitly define $T^{\mathrm{amort}}(\cdot)$ and verify.
  - Set time units such that $T^{\mathrm{actual}}(\textit{insert}) \leq 1$ and $T^{\mathrm{actual}}(\textit{resize}) \leq n$.
  - Define $T^{\mathrm{amort}}(\textit{insert}) = 3$ and $T^{\mathrm{amort}}(\textit{resize}) = 0$.

Verify $\sum_{i=1}^{k} T^{\mathrm{actual}}(\mathcal{O}_i) \leq$

$$\leq \sum_{i=1}^{k} T^{\mathrm{amort}}(\mathcal{O}_i).$$

Potential function method

Usually we need more systematic methods.

- **Potential function:** A function $\Phi(\cdot)$ that depends on the current status of the data structure.
  - E.g.: $\Phi(i) = \max\{0, 2 \cdot \mathit{size} - \mathit{capacity}\}$ for dynamic arrays.
  - "i" = operations $\mathcal{O}_1, \ldots, \mathcal{O}_i$ have been executed.
- Potential function must satisfy: $\Phi(0) = 0$, $\Phi(i) \geq 0$ for all i.
  - Can verify this for dynamic-array function above.

Potential function method

Usually we need more systematic methods.

- **Potential function:** A function $\Phi(\cdot)$ that depends on the current status of the data structure.
  - E.g.: $\Phi(i) = \max\{0, 2 \cdot \mathit{size} - \mathit{capacity}\}$ for dynamic arrays.
  - "i" = operations $\mathcal{O}_1, \ldots, \mathcal{O}_i$ have been executed.
- Potential function must satisfy: $\Phi(0) = 0$, $\Phi(i) \geq 0$ for all i.
  - Can verify this for dynamic-array function above.
- Define $\boxed{T^{\text{amort}}(\mathcal{O}_i) = T^{\text{actual}}(\mathcal{O}_i) + \Phi(i) - \Phi(i-1)}$
  - Often we just write $T^{\text{amort}}(\mathcal{O}) = T^{\text{actual}}(\mathcal{O}) + \Phi^{\text{after}} - \Phi^{\text{before}}$

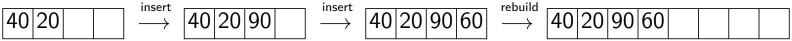
Potential function method

Usually we need more systematic methods.

- **Potential function:** A function $\Phi(\cdot)$ that depends on the current status of the data structure.
  - E.g.: $\Phi(i) = \max\{0, 2 \cdot \textit{size} - \textit{capacity}\}$ for dynamic arrays.
  - "i" = operations $\mathcal{O}_1, \ldots, \mathcal{O}_i$ have been executed.
- Potential function must satisfy: $\Phi(0) = 0$, $\Phi(i) \geq 0$ for all i.
  - Can verify this for dynamic-array function above.
- Define $\boxed{T^{\text{amort}}(\mathcal{O}_i) = T^{\text{actual}}(\mathcal{O}_i) + \Phi(i) - \Phi(i-1)}$
  - Often we just write $T^{\text{amort}}(\mathcal{O}) = T^{\text{actual}}(\mathcal{O}) + \Phi^{\text{after}} - \Phi^{\text{before}}$

Lemma: This satisfies $\sum_i T^{\text{actual}}(\mathcal{O}_i) \leq \sum_i T^{amort}(\mathcal{O}_i)$.

Example: Dynamic arrays



- Potential function $\Phi(i) = \max\{0, 2 \cdot \textit{size} - \textit{capacity}\}$
- As before set time units such that

$$T^{\text{actual}}(\textit{insert}) \leq 1 \text{ and } T^{\text{actual}}(\textit{resize}) \leq n.$$

Example: Dynamic arrays

$$\boxed{40\,|\,20\,|\;\;|\;\;} \xrightarrow{\text{insert}} \boxed{40\,|\,20\,|\,90\,|\;\;} \xrightarrow{\text{insert}} \boxed{40\,|\,20\,|\,90\,|\,60} \xrightarrow{\text{rebuild}} \boxed{40\,|\,20\,|\,90\,|\,60\,|\;\;|\;\;|\;\;|\;\;}$$

- Potential function $\Phi(i) = \max\{0, 2 \cdot \textit{size} - \textit{capacity}\}$
- As before set time units such that

$$T^{\text{actual}}(\textit{insert}) \le 1 \text{ and } T^{\text{actual}}(\textit{resize}) \le n.$$

- *insert* increases *size*, does not change *capacity*

$$\Rightarrow \;\; \Delta\Phi = \Phi^{\text{after}} - \Phi^{\text{before}} \le 2 - 0 = 2$$

$$T^{amort}(\textit{insert}) \le 1 + 2 - 0 = 3 \in O(1)$$

Example: Dynamic arrays

40	20			$\xrightarrow{\text{insert}}$	40	20	90		$\xrightarrow{\text{insert}}$	40	20	90	60	$\xrightarrow{\text{rebuild}}$	40	20	90	60				

- Potential function $\Phi(i) = \max\{0, 2 \cdot \mathit{size} - \mathit{capacity}\}$
- As before set time units such that

$$T^{\text{actual}}(\mathit{insert}) \leq 1 \text{ and } T^{\text{actual}}(\mathit{resize}) \leq n.$$

- *insert* increases *size*, does not change *capacity*

$$\Rightarrow \Delta\Phi = \Phi^{\text{after}} - \Phi^{\text{before}} \leq 2 - 0 = 2$$

$$T^{amort}(\mathit{insert}) \leq 1 + 2 - 0 = 3 \in O(1)$$

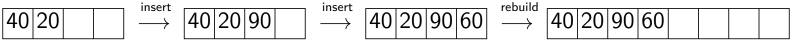
- *rebuild* happens only if $\mathit{size} = \mathit{capacity} = n$

$$\Rightarrow \Phi^{\text{before}} = 2n - n = n.$$

$$\Rightarrow \Phi^{\text{after}} = 2n - 2n = 0 \text{ since new capacity is } 2n.$$

$$T^{amort}(\mathit{insert}) \leq n + 0 - n = 0 \in O(1)$$

Example: Dynamic arrays



- Potential function $\Phi(i) = \max\{0, 2 \cdot \textit{size} - \textit{capacity}\}$
- As before set time units such that

$$T^{\text{actual}}(\textit{insert}) \leq 1 \text{ and } T^{\text{actual}}(\textit{resize}) \leq n.$$

- *insert* increases *size*, does not change *capacity*

  $\Rightarrow \ \Delta\Phi = \Phi^{\text{after}} - \Phi^{\text{before}} \leq 2 - 0 = 2$

$$T^{amort}(\textit{insert}) \leq 1 + 2 - 0 = 3 \in O(1)$$

- *rebuild* happens only if $\textit{size} = \textit{capacity} = n$

  $\Rightarrow \ \Phi^{\text{before}} = 2n - n = n.$

  $\Rightarrow \ \Phi^{\text{after}} = 2n - 2n = 0$ since new capacity is $2n$.

$$T^{amort}(\textit{insert}) \leq n + 0 - n = 0 \in O(1)$$

Result: The amortized run-time of dynamic arrays is $O(1)$.

Potential function method

How to find a suitable potential function?
(No recipe, but some guidelines.)

- Study the expensive operation: What gets smaller?

40	20	90	60

$\xrightarrow{\text{rebuild}}$

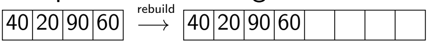
40	20	90	60				

  - Dynamic arrays: *rebuild* *increases* capacity.
    We want the potential function to get smaller.
    So potential function should have term "$-$*capacity*.

Potential function method

How to find a suitable potential function?
(No recipe, but some guidelines.)

- Study the expensive operation: What gets smaller?



  - Dynamic arrays: *rebuild increases* capacity.
    We want the potential function to get smaller.
    So potential function should have term "−*capacity*.
- Study condition $\Phi(\cdot) \geq 0$ and $\Phi(0) = 0$.
  - Dynamic arrays: Usually have $\mathit{capacity} \leq 2 \cdot \mathit{size}$.
    So usually $2 \cdot \mathit{size} - \mathit{capactiy} \geq 0$,
  - We added a $\max\{0, \dots\}$ term so that also $\Phi(0) = 0$.

Potential function method

How to find a suitable potential function?
(No recipe, but some guidelines.)

- Study the expensive operation: What gets smaller?

40	20	90	60

  $\xrightarrow{\text{rebuild}}$

40	20	90	60				

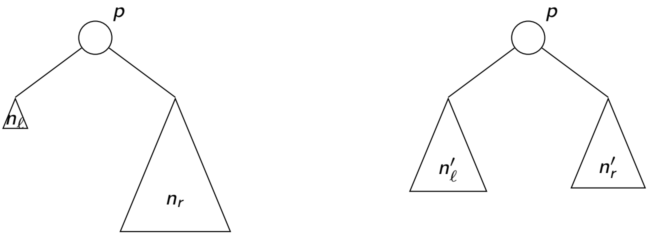
  - Dynamic arrays: *rebuild* *increases* capacity.
    We want the potential function to get smaller.
    So potential function should have term "$-$*capacity*.
- Study condition $\Phi(\cdot) \geq 0$ and $\Phi(0) = 0$.
  - Dynamic arrays: Usually have *capacity* $\leq 2 \cdot$ *size*.
    So usually $2 \cdot$ *size* $-$ *capactiy* ≥ 0,
  - We added a $\max\{0, \dots\}$ term so that also $\Phi(0) = 0$.
- Compute the amortized time and see whether you get good bounds.
- Rinse, lather, repeat.

Outline

4. More Balanced Search Trees
   - Balanced binary search trees
   - Scapegoat Trees
   - Amortized analysis
   - Analysis of scapegoat trees

Amortized analysis of scapegoat trees

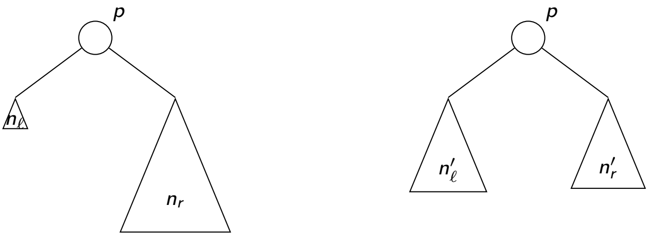
- Expensive operation: Rebuild subtree at p.



- **Claim:** If we rebuild at p, then $|n_r - n_\ell| \geq (2\alpha - 1)n_p$.
  Proof:

Amortized analysis of scapegoat trees

- Expensive operation: Rebuild subtree at p.



- **Claim:** If we rebuild at p, then $|n_r - n_\ell| \geq (2\alpha - 1)n_p$.
  Proof:

- **Idea:** Potential function should involve $\sum_v |v.\mathit{left.size} - v.\mathit{right.size}|$.

Amortized analysis of scapegoat trees

- Use $\Phi(i) = c \cdot \sum_v \max\{|v.\mathit{left} - v.\mathit{right}| - 1, 0\}$ for some constant c.

Amortized analysis of scapegoat trees

- Use $\Phi(i) = c \cdot \sum_v \max\{|v.\mathit{left} - v.\mathit{right}| - 1, 0\}$ for some constant c.
- *insert* and *delete* increases contribution at ancestors by at most 1 and does not increase other contributions.

$$
\begin{aligned}
T^{amort}(\mathit{insert}) &= T^{actual}(\mathit{insert}) + \Phi_{\text{after}} - \Phi_{\text{before}} \\
&\leq \log n + c\#\{\text{ancestors}\} \in O(\log n)
\end{aligned}
$$

Amortized analysis of scapegoat trees

- Use $\Phi(i) = c \cdot \sum_v \max\{|v.left - v.right| - 1, 0\}$ for some constant c.
- *insert* and *delete* increases contribution at ancestors by at most 1 and does not increase other contributions.

$$\begin{aligned} T^{amort}(insert) &= T^{actual}(insert) + \Phi_{\text{after}} - \Phi_{\text{before}} \\ &\leq \log n + c\#\{\text{ancestors}\} \in O(\log n) \end{aligned}$$

- *rebuild* decreases contribution at p by $(2\alpha - 1)n_p$ and does not increase other contributions.

$$\begin{aligned} T^{amort}(rebuild) &= T^{actual}(rebuild) + \Phi_{\text{after}} - \Phi_{\text{before}} \\ &\leq n_p + c(-(2\alpha - 1)n_p) \end{aligned}$$

Amortized analysis of scapegoat trees

- Use $\Phi(i) = c \cdot \sum_v \max\{|v.\mathit{left} - v.\mathit{right}| - 1, 0\}$ for some constant c.
- *insert* and *delete* increases contribution at ancestors by at most 1 and does not increase other contributions.

$$\begin{aligned} T^{amort}(\mathit{insert}) &= T^{actual}(\mathit{insert}) + \Phi_{\text{after}} - \Phi_{\text{before}} \\ &\leq \log n + c\#\{\text{ancestors}\} \in O(\log n) \end{aligned}$$

- *rebuild* decreases contribution at p by $(2\alpha - 1)n_p$ and does not increase other contributions.

$$\begin{aligned} T^{amort}(\mathit{rebuild}) &= T^{actual}(\mathit{rebuild}) + \Phi_{\text{after}} - \Phi_{\text{before}} \\ &\leq n_p + c(-(2\alpha - 1)n_p) \end{aligned}$$

With $c = 1/(2\alpha - 1)$, this is at most 0 and *rebuild* is free.

Result: All operations have amortized run-time in $O(\log n)$.