CS 240 – Data Structures and Data Management

Module 4: Dictionaries - Enriched

T. Biedl   É. Schost   O. Veksler

Based on lecture notes by many previous cs240 instructors

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

Winter 2021

version *2021–02–10 12:28*

Outline

1. Dictionaries and Balanced Search Trees
   - Scapegoat Trees

Outline

1. Dictionaries and Balanced Search Trees
   - Scapegoat Trees

Scapegoat trees

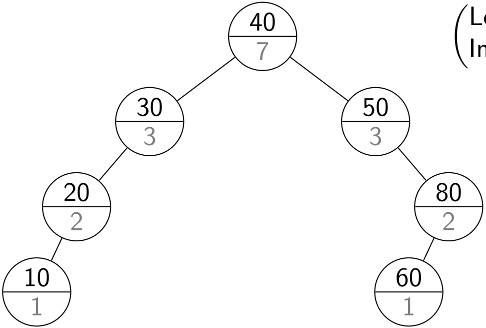
- Can we have balanced binary search trees *without* rotations?
  (A later application will need such a tree.)
- This sounds impossible—we must sometimes restructure the tree.
- **Idea:** Rather than doing a small local change, occasionally do a large (near-global) rebuilt.

- 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

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



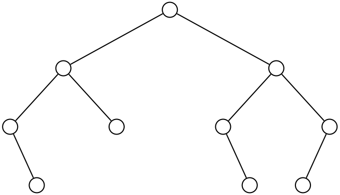
$\left(\begin{array}{l}\text{Lower number} = \text{subtree-size.}\\ \text{In our examples, } \alpha = \frac{2}{3}.\end{array}\right)$

- $v.size$ 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**:

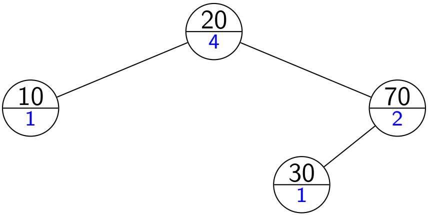
$$|size(z.left) - size(z.right)| \leq 1 \quad \text{for all nodes } z.$$



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

Scapegoat Tree Insertion Example

Example:



Scapegoat tree insertion

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

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

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.

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

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

Detour: Amortized analysis

For dynamic arrays, some ad-hoc methods work.

$$\begin{array}{|c|c|c|c|}\hline 40 & 20 & & \\ \hline\end{array} \xrightarrow{\text{insert}} \begin{array}{|c|c|c|c|}\hline 40 & 20 & 90 & \\ \hline\end{array} \xrightarrow{\text{insert}} \begin{array}{|c|c|c|c|}\hline 40 & 20 & 90 & 60 \\ \hline\end{array} \xrightarrow{\text{rebuild}} \begin{array}{|c|c|c|c|c|c|c|c|}\hline 40 & 20 & 90 & 60 & & & & \\ \hline\end{array}$$

- 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 $T^{\mathrm{actual}}(\mathit{insert}) \leq 1$ and $T^{\mathrm{actual}}(\mathit{resize}) \leq n$.
  - Define $T^{\mathrm{amort}}(\mathit{insert}) = 3$ and $T^{\mathrm{amort}}(\mathit{resize}) = 0$.

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.: $\Phi(i) = \max\{0, 2 \cdot \textit{size} - \textit{capacity}\}$ 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 $\boxed{T^{\mathrm{amort}}(\mathcal{O}_i) = T^{\mathrm{actual}}(\mathcal{O}_i) + \Phi(i) - \Phi(i-1)}$
  - Often we just write $T^{\mathrm{amort}}(\mathcal{O}) = T^{\mathrm{actual}}(\mathcal{O}) + \Phi^{\mathrm{after}} - \Phi^{\mathrm{before}}$

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

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 $\Phi(i) = \max\{0, 2 \cdot \mathit{size} - \mathit{capacity}\}$
- 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 $\mathit{size} = \mathit{capacity} = n$

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

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

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

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

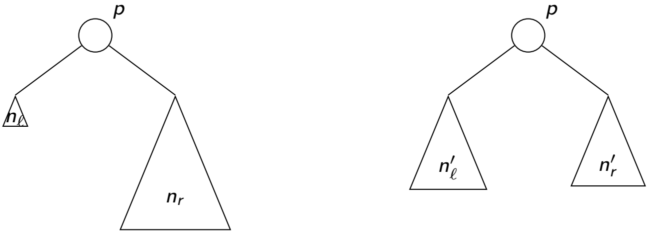
$\xrightarrow{\text{rebuild}}$

40	20	90	60				

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

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 $\sum_v |v.\mathit{left}.\mathit{size} - v.\mathit{right}.\mathit{size}|$.

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.
- *insert* and *delete* increases contribution at ancestors by at most 1 and does not increase other contributions.

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

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

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

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

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