CS 240E – Data Structures and Data Management (Enriched)

Module 4: Dictionaries

Tom Iagovet

Based on lecture notes by many previous cs240 instructors

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

Spring 2026

version 2026-06-02 00:42

Outline

4. Dictionaries and Balanced Search Trees
   - ADT Dictionary
   - AVL Trees
   - Insertion in AVL Trees
   - Restructuring a BST: Rotations
   - AVL insertion revisited
   - Scapegoat Trees
   - Amortized analysis
   - Scapegoat trees: Amortized analysis

Review: ADT Dictionary

Dictionary: A collection of items, each of which contains

- a *key*
- some *data* (the "value")

and is called a *key-value pair* (KVP). Keys can be compared and are (typically) unique.

Operations:

- *search*(k) (also called *lookup*(k))
- *insert*(k, v)
- *delete*(k) (also called *remove*(k))
- optional: *successor*, *merge*, *is-empty*, *size*, *etc.*

Review: Elementary realizations

Common assumptions:

- Dictionary has n KVPs
- Each KVP uses constant space
- Keys can be compared in constant time

We commonly make one more assumption (to keep pseudo-code simple):

- Dictionary is non-empty both before and after operation.

  (In a real-life implementation you would have to treat these special cases.)

Easy realizations:

	search	*insert*	*delete*
unsorted list/array	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sorted array	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
binary search tree	$\Theta(height)$	$\Theta(height)$	$\Theta(height)$

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.)

AVL trees

Introduced by Adel'son-Vel'skiĭ and Landis in 1962, an **AVL tree** is a BST with an additional **height-balance property** at every node:

The heights of the left and right subtree differ by at most 1.

Rephrase: If node z has left subtree L and right subtree R, then

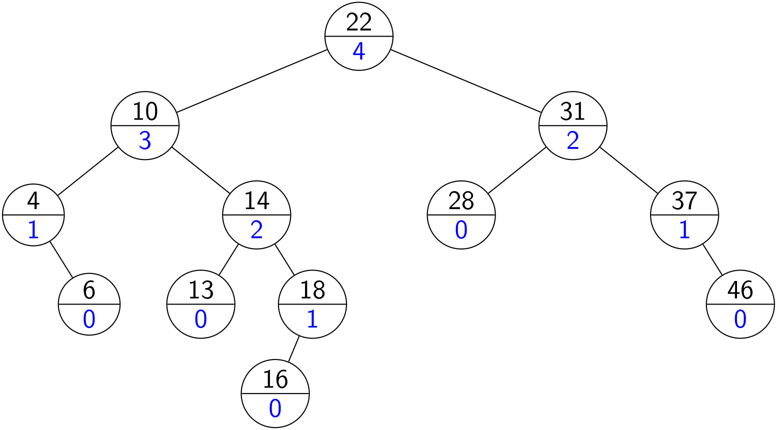
$$\mathit{height}(R) - \mathit{height}(L) \text{ must be in } \{-1, 0, 1\}$$

Things to show:

- This *structural condition* implies logarithmic height.
- After *insert* and *delete*, we can restore the structural condition within logarithmic time.
  - For this, we need to store at each node z the height of the subtree rooted at it.

AVL tree example

(The lower numbers indicate the height of the subtree.)



AVL trees: Height

Theorem: An AVL tree on n nodes has $\Theta(\log n)$ height.
$\Rightarrow$ *search*, *BST::insert*, *BST::delete* all cost $\Theta(\log n)$ in the *worst case!*

Proof:

- Define $N(h)$ to be the *least* number of nodes in a height-h AVL tree.
- What is a recurrence relation for $N(h)$?
- What does this recurrence relation resolve to?

AVL trees: Insertion

To perform *AVL::insert*(k, v):

- First, insert (k, v) with the usual BST insertion.
- We assume that this returns the new leaf z where the key was stored.
- Then, move up the tree from z.

  (We assume for this that we have parent-links. This can be avoided if *BST::insert* returns the full path to z.)

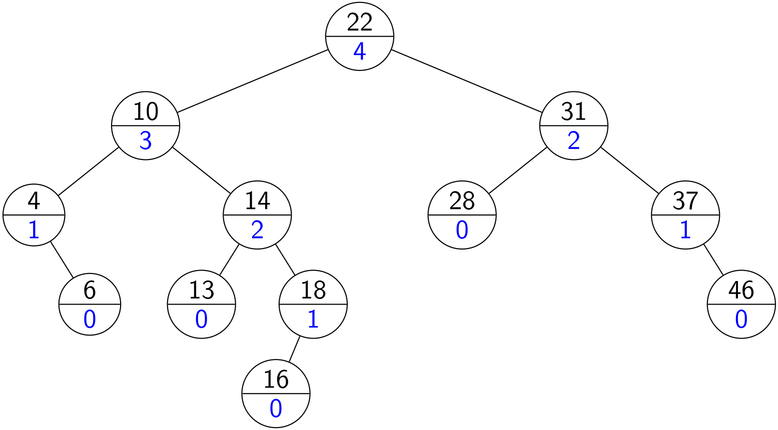
- Update height (easy to do in constant time):

  set-height-from-subtrees(u)
  1. $u.height \leftarrow 1 + \max\{u.left.height, u.right.height\}$

- If the height difference becomes ± 2 at node z, then z is **unbalanced**. Must re-structure the tree to rebalance.

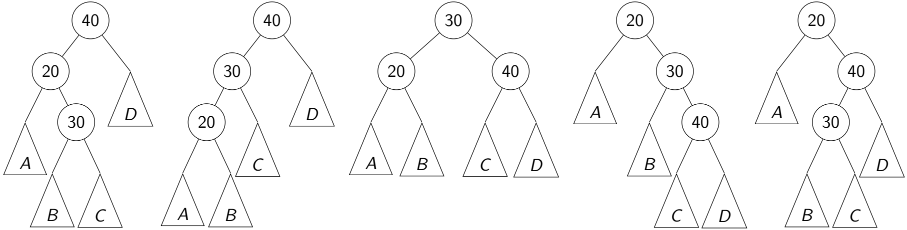
AVL tree insertion: Example

Example: *AVL::insert*(8)



Changing structure without changing order

Note: There are many different BSTs with the same keys.

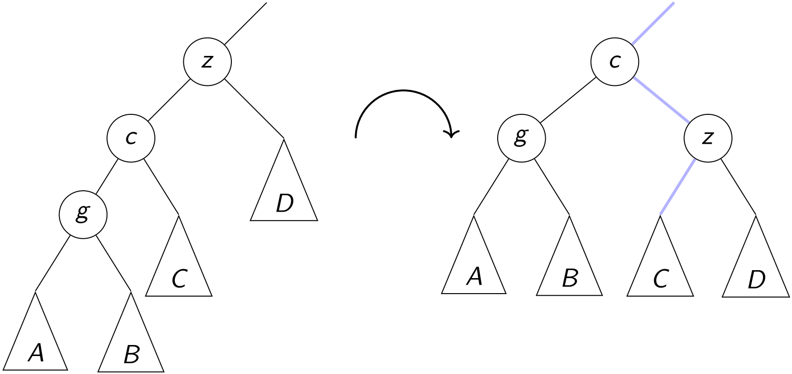


Goal: Change the *structure* locally without changing the *order*.

Longterm goal: Restructure such that the subtree becomes balanced.

Right Rotation

This is a **right rotation** on node z:



Note: Only $O(1)$ links are changed. Useful to fix left-left imbalance.

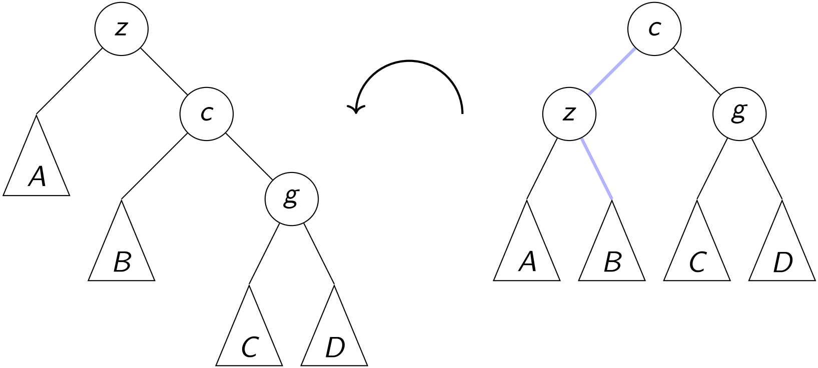
Right Rotation: Pseudocode

```
rotate-right(z)
c ← z.left

// fix links connecting to above
c.parent ← (p ← z.parent)
if p = NULL then root ← c else
      if p.left = z then p.left ← c else p.right ← c

// actual rotation
z.left ← c.right, c.right.parent ← z
c.right ← z, z.parent ← c

set-height-from-subtrees(z), set-height-from-subtrees(c)
return c      // returns new root of subtree
```

Run-time: $O(1)$

Left Rotation

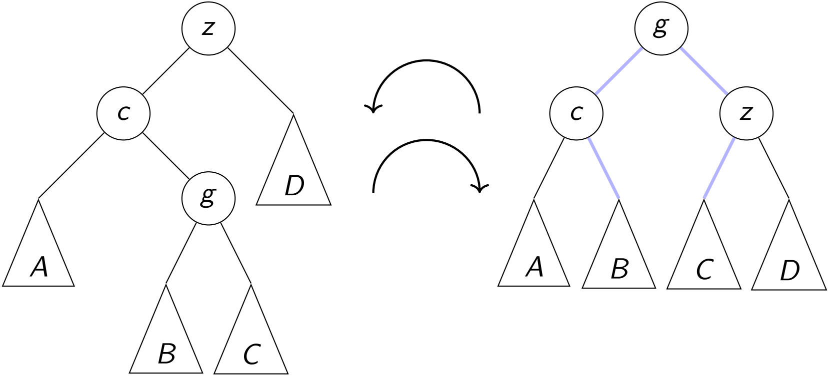
Symmetrically, this is a **left rotation** on node z:



Again, only $O(1)$ links need to be changed. Useful to fix right-right imbalance.

Double Right Rotation

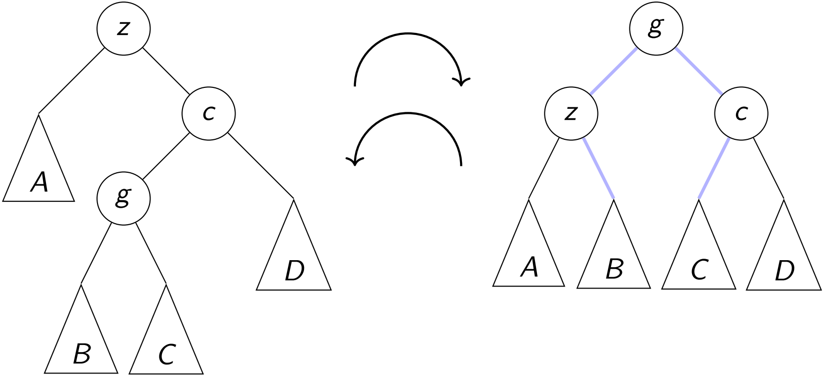
This is a **double right rotation** on node z:



First, a left rotation at c.
Second, a right rotation at z.

Double Left Rotation

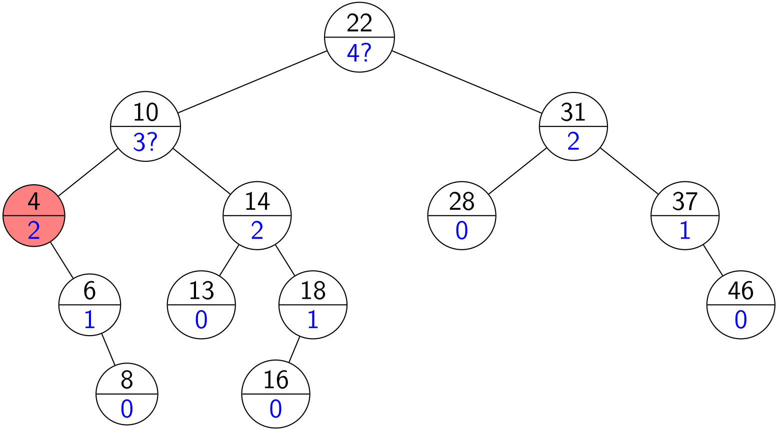
Symmetrically, there is a **double left rotation** on node z:



First, a right rotation at c.
Second, a left rotation at z.

AVL tree insertion: Example revisited

Example:



AVL tree insertion revisited

- Imbalance at z: do (single or double) rotation
- Choose c as child where subtree has bigger height.

```
AVL::insert(k, v)
z ← BST::insert(k, v)              // new leaf with k
while (z is not NULL)
    if (|z.left.height − z.right.height| > 1) then
        Let c be taller child of z
        Let g be taller child of c (so grandchild of z)
        z ← restructure(g, c, z) // see later
        break          // can argue that we are done
    set-height-from-subtrees(z)
    z ← z.parent
```

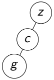
Can argue: For insertion *one* rotation restores all heights of subtrees.

$\Rightarrow$ No further imbalances, can stop checking.

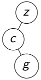
Fixing a slightly-unbalanced AVL tree

restructure(g, c, z)
node g is child of c which is child of z

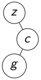
1. **case**



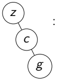
:
`// Right rotation`
$u \leftarrow$ *rotate-right*(z)



: `// Double-right rotation`
rotate-left(c)
$u \leftarrow$ *rotate-right*(z)



: `// Double-left rotation`
rotate-right(c)
$u \leftarrow$ *rotate-left*(z)



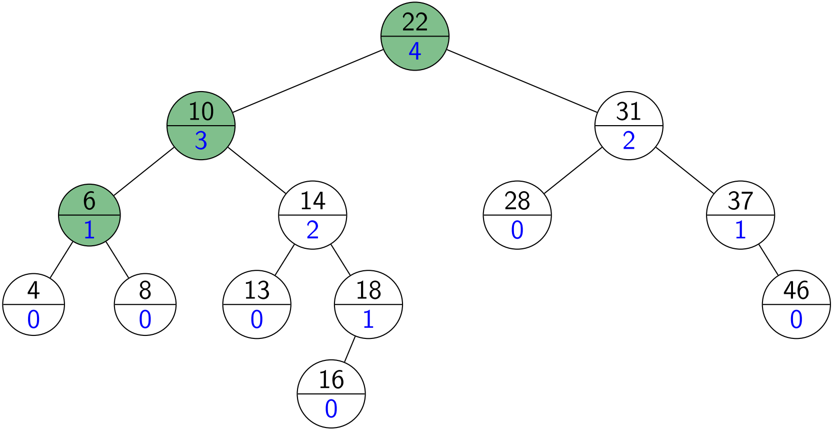
: `// Left rotation`
$u \leftarrow$ *rotate-left*(z)

2. **return** u

Rule: The middle key of g, c, z becomes the new root.

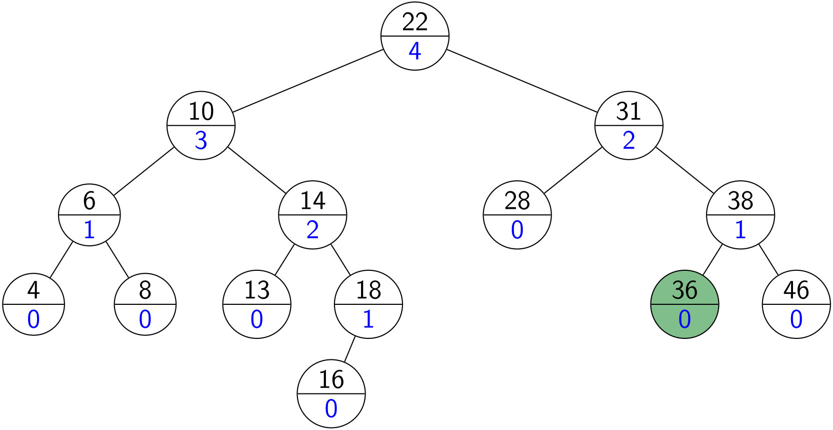
AVL tree insertion: Example revisited

Example:



AVL tree insertion: Second example

Example: *AVL::insert*(36), *AVL::insert*(35)

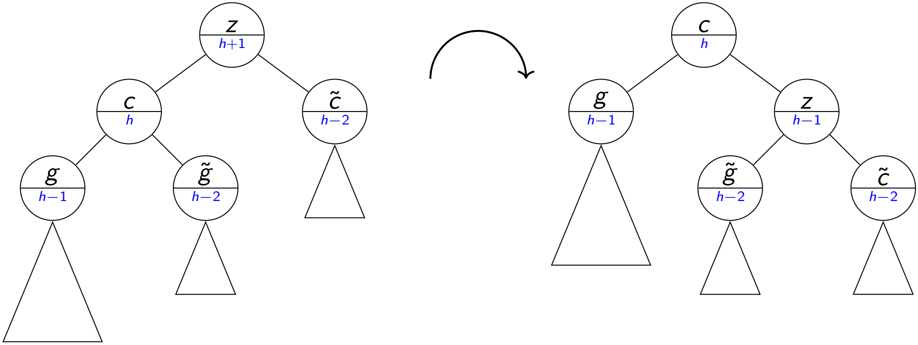


Correctness of rotations

Claim: If we perform *restructure*(g, c, z) during *AVL::insert*, then the returned subtree is balanced, and its height is restored.

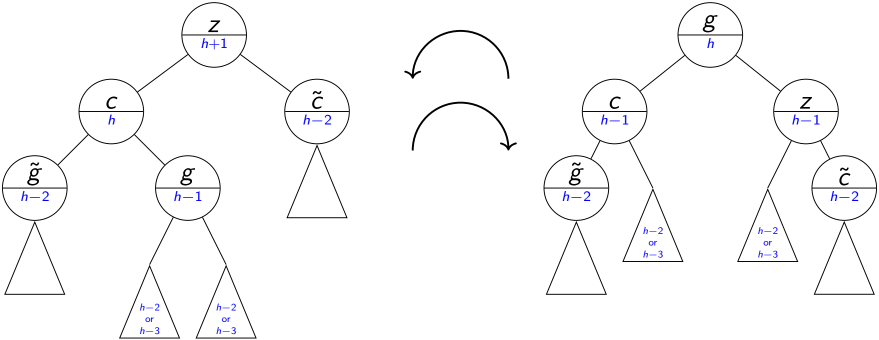
Proof: By symmetry, assume that c is the left child of z.

Case 1: g was the left child of c



Correctness of rotations

Case 2: g is the right child of c.



Lazy deletion

General trick for ADT Dictionary: Reduce *delete* to *search* and *insert*.

- *delete*(x) does not actually remove x from data structure.
  - Instead, *delete* calls *search*, then sets a flag to "deleted"
  - *search* ignores "deleted" items (but keeps searching)
- Keep track of how many items are "deleted".
- If at least half are "deleted": *completely* rebuild
  - Run-time: $O(n * insert)$ (where $n = \#$ "non-deleted")
  - But: This only happens if we had n calls to *delete* since last rebuild. All other calls to *delete* take $O(search)$ time.

$\Rightarrow$ *delete* then takes $O(search + insert)$ time in average over operations.

- Lazy deletion wastes space; occasional operation is very slow.
- Most realizations actually can do *delete* directly; the course notes have details for AVL-trees.

AVL trees: Summary

search: Just like in BSTs, costs $\Theta(height)$

insert: *BST::insert*, then check & update along path to new leaf

- total cost $\Theta(height)$
- *restructure* will be called *at most once*.

delete: *BST::delete*, then check & update along path to deleted node

(we did not see details of how to do this)

- total cost $\Theta(height)$
- *restructure* may be called $\Theta(height)$ times.

Worst-case cost for all operations is $\Theta(height) = \Theta(\log n)$.

- In practice, the constant is quite large.
- Other realizations of ADT Dictionary are better in practice ($\rightarrow$ later)

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 rebuild an entire (large) subtree.
  - This feels a lot like lazy deletion and will be analyzed the same way.
- With the right setup, this will lead to $O(\log n)$ height and $O(\log n)$ **amortized** time for all operations.

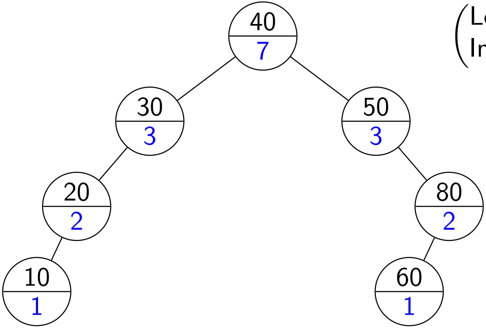
Scapegoat trees

Fix a constant α with $\frac{1}{2} < \alpha < 1$. A **scapegoat tree** is a binary search tree with a **size-balance condition**:

> *The size at any non-root z is an α-fraction smaller than the size at its parent.*

Rephrase: At any non-root node z we have

$$z.\mathit{size} \leq \alpha \cdot z.\mathit{parent}.\mathit{size}.$$



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

Scapegoat trees: Operations

Claim: Any scapegoat tree has height $O(\log n)$.

In consequence, *search* has run-time $O(\log n)$.
For *insert* and *delete*:

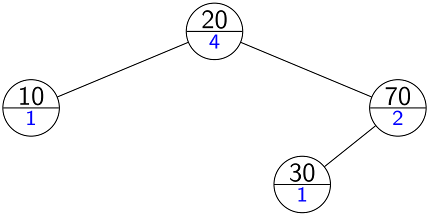
- Perform *insert* and *delete* as in the BST
- Go back up from node z where structural change happened
- Verify that size-condition holds.

  (This is easy if we store *z.size* with every node z and have parent-references. With some modification it can be done without these.)
- If the condition holds everywhere: $O(\log n)$ time.
- If condition does not hold: *rebuild entire subtree*
  - Let p be the *highest* node where condition is violated.
  - Rebuild entire subtree of p to be **perfectly size-balanced**:

$$|z.left.size - z.right.size| \leq 1 \text{ at all nodes } z$$

Scapegoat tree insertion: Example

Example:
insert 30, 60, 50, 40 (in this order)



Scapegoat tree insertion

scapegoatTree::insert(k, v)

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

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**. Informally, this is a bound that holds if we add the run-times over operations $\mathcal{O}_1, \ldots, \mathcal{O}_k$:

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

(and try to keep $T^{\text{amort}}(\cdot)$ small within that constraint)

Detour: Amortized analysis

For dynamic arrays and lazy deletion, a direct argument works.

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

- $n/2$ 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.

More systematic method: Use a **potential function**

- A function $\Phi(\cdot)$ that depends on the current status.
  - E.g.: $\Phi(t) = \max\{0, 2 \cdot \textit{size} - \textit{capacity}\}$ for dynamic arrays.
  - t ($\approx$ time) means "after executing t operations"
- *Requirement*: $\Phi(0) = 0$, $\Phi(i) \geq 0$ for all i.

  (Nothing else required, but some functions give better amortized bounds than others.)

Steps of analysis via potential function

- Define **time units**: how much can be done in one unit of time?
  - Tool to avoid doing math with asymptotic notation.
  - One time unit should correspond to $\Theta(1)$ run-time.
  - Dynamic arrays: "Set time units such that
    $T^{\text{actual}}(\textit{insert}) \leq 1$ and $T^{\text{actual}}(\textit{resize}) \leq n$."
- Define a potential function Φ and verify that $\Phi(0) = 0, \Phi(i) \geq 0$.
  - Finding Φ is non-trivial ($\rightsquigarrow$ later)
- Define $\boxed{T^{\text{amort}}(\mathcal{O}_t) = T^{\text{actual}}(\mathcal{O}_t) + \Phi(t) - \Phi(t-1)}$
  - Often we just write $T^{\text{amort}}(\mathcal{O}) = T^{\text{actual}}(\mathcal{O}) + \Phi^{\text{new}} - \Phi^{\text{old}}$
  - Easy to show: $\sum_i T^{\text{actual}}(\mathcal{O}_i) \leq \sum_i T^{\text{amort}}(\mathcal{O}_i)$ holds.
- Find asymptotic upper bounds for $T^{\text{amort}}(\mathcal{O})$ for each operation.

Example: Dynamic arrays

40	20		

$\xrightarrow{\text{insert}}$

40	20	90	

$\xrightarrow{\text{insert}}$

40	20	90	60

$\xrightarrow{\text{resize}}$

40	20	90	60				

- Set time units such that $T^{\text{actual}}(\textit{insert}) \le 1$ and $T^{\text{actual}}(\textit{resize}) \le n$.
- Potential function $\Phi(t) = \max\{0, 2 \cdot \textit{size} - \textit{capacity}\}$
- *insert* increases *size* by 1, does not change *capacity*
  $\Rightarrow$ $\Phi^{\text{new}} - \Phi^{\text{old}} \le 2$ and $T^{\text{amort}}(\textit{insert}) \le 1 + 2 = 3 \in O(1)$
- *resize* happens only if $\textit{size} = \textit{capacity} = n$
  $\Rightarrow$ $\Phi^{\text{old}} = 2n - n = n$.
  $\Rightarrow$ $\Phi^{\text{new}} = 2n - 2n = 0$ since the new capacity is $2n$.

$$T^{\text{amort}}(\textit{resize}) \le n + 0 - n = 0 \in O(1)$$

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 changes a lot?

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

  - Dynamic arrays: *resize* *increases* capacity.
  - Potential function should decrease (to "pay" for the operation).
  - So potential function should have term "$-$*capacity*".

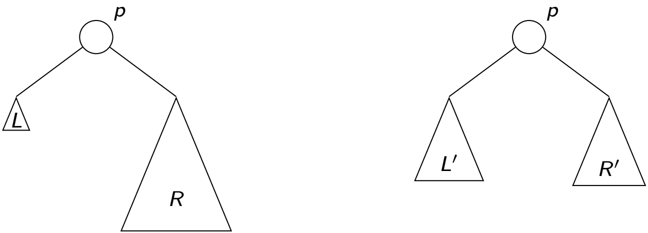
- Study condition $\Phi(\cdot) \geq 0$ and $\Phi(0) = 0$.
  - Dynamic arrays: Usually have $\mathit{capacity} \leq 2 \cdot \mathit{size}$.
    So usually $2 \cdot \mathit{size} - \mathit{capactiy} \geq 0$.
  - We added a $\max\{0, \dots\}$ term so that also $\Phi(0) = 0$.

- Compute the amortized time and see whether you get good bounds.

- Lather, rinse, repeat.

Scapegoat trees: Amortized analysis

- Expensive operation: Rebuild subtree at p. Time $\Theta(p.size)$.



- **Claim:** If we rebuild at p, then $|L.size - R.size| \geq (2\alpha - 1)p.size + 1$
  Proof:

- **Observe:** After rebuild, we have $|L'.size - R'.size| \leq 1$
- **Idea:** Potential function should involve $\sum_z |z.left.size - z.right.size|$.

Scapegoat trees: Amortized analysis

- Define time units such that *insert* (without re-building) takes $\leq \log n$ time units, and *rebuild* at p takes $\leq p.size$ time units.
- Use $\Phi(t) = c \cdot \sum_z \underbrace{\max\{|z.\mathit{left}.\mathit{size} - z.\mathit{right}.\mathit{size}| - 1, 0\}}_{\Phi_z}$

  (constant c will be chosen suitably below)

- *insert*: Let ℓ be the new leaf
  - *size* increases by 1 at all ancestors of ℓ
  - Φ_z changes by at most 1 at all ancestors of ℓ and does not change at any other nodes
  - So $\Phi^{\mathrm{new}} - \Phi^{\mathrm{old}} \leq c\#\text{ancestors}$
  - Recall: $\#\text{ancestors} \leq \mathit{height} \in O(\log n)$

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

Scapegoat trees: Amortized analysis

- Use $\Phi(t) = c \cdot \sum_z \underbrace{\max\{|z.\mathit{left.size} - z.\mathit{right.size}| - 1, 0\}}_{\Phi_z}$

- *rebuild*: Let p be the node whose subtree got rebuilt
  - Φ_p decreases by $(2\alpha - 1)p.\mathit{size}$
  - For all nodes z in subtree of p, $\Phi_z^{\text{new}} = 0$ by size-balance
    $\Rightarrow \Phi_z$ did not increase
  - For all other nodes z, Φ_z did not change
  - So $\Phi^{\text{new}} - \Phi^{\text{old}} \leq c(-2\alpha - 1)p.\mathit{size}$

$$\begin{aligned} T^{\text{amort}}(\mathit{rebuild}) &= T^{\mathit{actual}}(\mathit{rebuild}) + \Phi^{\text{new}} - \Phi^{\text{old}} \\ &\leq p.\mathit{size} + c(-(2\alpha - 1)p.\mathit{size}) \end{aligned}$$

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

Summary: Scapegoat trees realize ADT Dictionary with $O(\log n)$ amortized time for all operations and *no rotations*.