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

Module 4: Dictionaries

Therese Biedl

Based on lecture notes by many previous cs240 instructors

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

Winter 2025

version 2025−01−21 13:32

Outline

Outline

ADT Dictionary (review)

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.*

Examples: symbol table, license plate database

Elementary Realizations (review)

Common assumptions:

- Dictionary has n KVPs
- Each KVP uses constant space
  (if not, the "value" could be a pointer)
- Keys can be compared in constant time

Elementary Realizations (review)

Common assumptions:

- Dictionary has n KVPs
- Each KVP uses constant space
  (if not, the "value" could be a pointer)
- 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.)

Elementary Realizations (review)

Common assumptions:

- Dictionary has n KVPs
- Each KVP uses constant space
  (if not, the "value" could be a pointer)
- 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	$\Theta(\mathit{height})$	$\Theta(\mathit{height})$	$\Theta(\mathit{height})$

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.)

General strategy for balanced binary search trees

- 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

General strategy for balanced binary search trees

- 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

- *insert* and *delete* may destroy the structural condition
- If so: show how to *restore* structural condition in $O(\text{height})$ time
- With this, *insert* and *delete* takes $O(\log n)$ ... time

Lazy deletion

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

Lazy deletion

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

- *delete*(x) does not actually remove x from data structure.
  - Instead, have a flag at each item that is either “deleted” or “present”
  - *insert* sets the flag to “present”
  - *delete* calls *search*, then sets the flag to “deleted”
  - *search* ignores “deleted” items (but keeps searching)

Lazy deletion

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

- *delete*(x) does not actually remove x from data structure.
  - Instead, have a flag at each item that is either "deleted" or "present"
  - *insert* sets the flag to "present"
  - *delete* calls *search*, then sets the 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: $O(n * \textit{insert})$ (where $n = \#$ "present")
  - But: This only happens if we had n calls to *delete* since last rebuild. All other calls to *delete* take $O(\textit{search})$ time.

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

Lazy deletion

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

- *delete*(x) does not actually remove x from data structure.
  - Instead, have a flag at each item that is either "deleted" or "present"
  - *insert* sets the flag to "present"
  - *delete* calls *search*, then sets the 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: $O(n * \mathit{insert})$ (where $n = \#$ "present")
  - But: This only happens if we had n calls to *delete* since last rebuild. All other calls to *delete* take $O(\mathit{search})$ time.

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

- Lazy deletion wastes space; occasional operation is very slow.
- Most realizations actually can do *delete* directly.

Outline

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 v has left subtree L and right subtree R, then

$$\textbf{balance}(v) := \mathit{height}(R) - \mathit{height}(L) \text{ must be in } \{-1, 0, 1\}$$

$\mathit{balance}(v) = -1$ means v is *left-heavy*

$\mathit{balance}(v) = +1$ means v is *right-heavy*

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 v has left subtree L and right subtree R, then

$$\textbf{balance}(v) := height(R) - height(L) \text{ must be in } \{-1, 0, 1\}$$

$balance(v) = -1$ means v is *left-heavy*

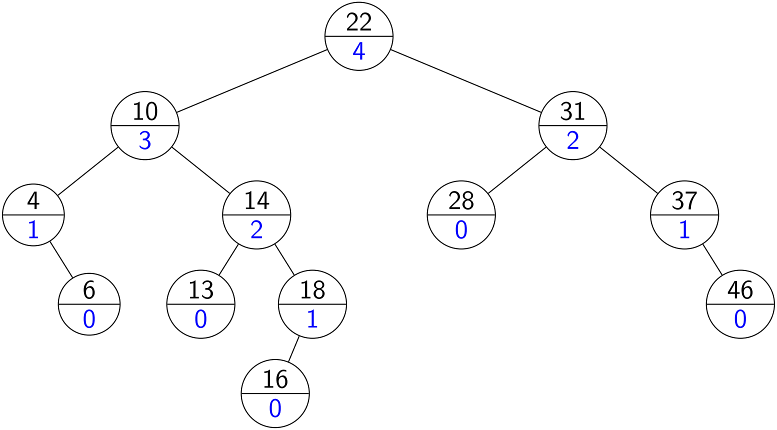
$balance(v) = +1$ means v is *right-heavy*

- Need to store at each node v the height of the subtree rooted at it

(There are ways to implement AVL-trees where we only store $balance(v)$, so fewer bits. But the code gets more complicated (no details).)

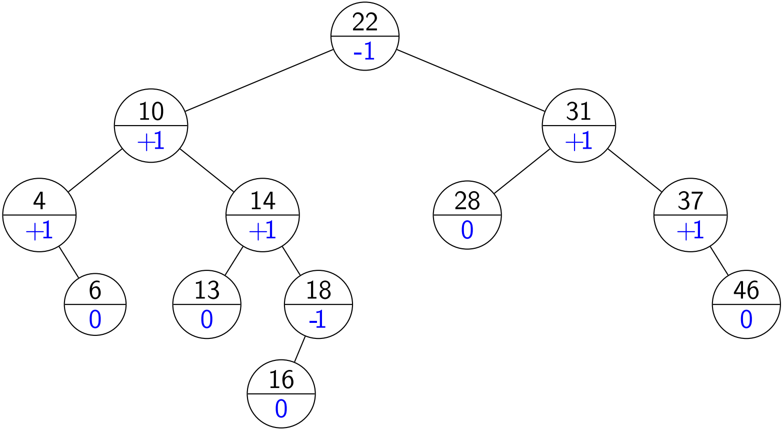
AVL tree example

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



AVL tree example

Alternative: store balance (instead of height) at each node.



- Saves space (2 bits vs. 1 integer per node)
- Pseudo-code gets a lot more complicated $\rightsquigarrow$ not done here

Height of an AVL tree

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?

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
- Analysis of scapegoat trees

AVL 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.)

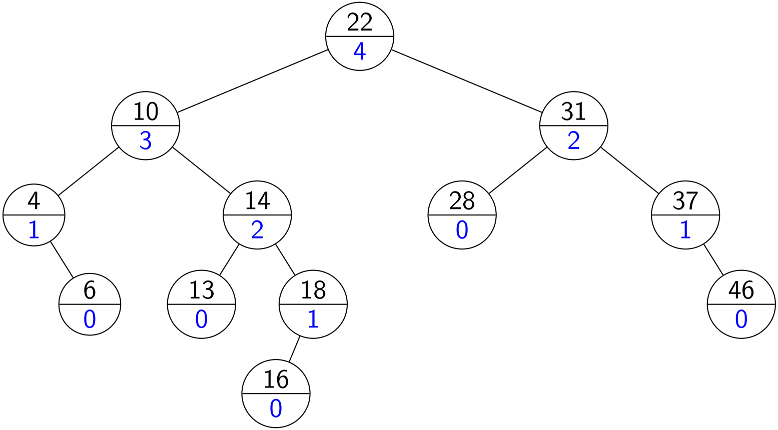
- Update height (easy to do in constant time):

  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.

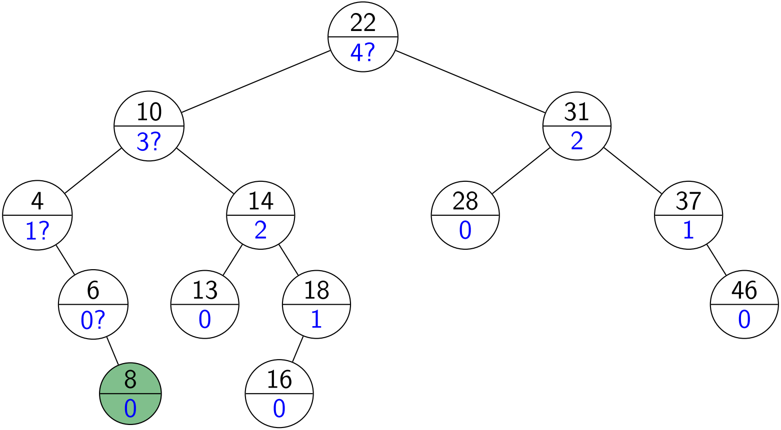
AVL Insertion Example

Example: *AVL::insert*(8)



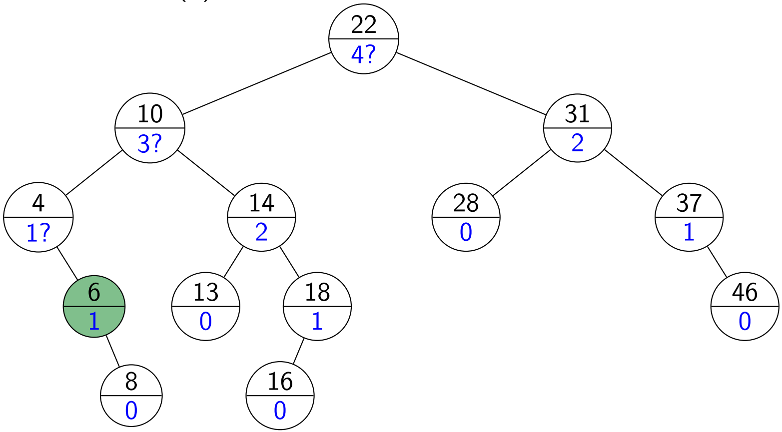
AVL Insertion Example

Example: *AVL::insert*(8)



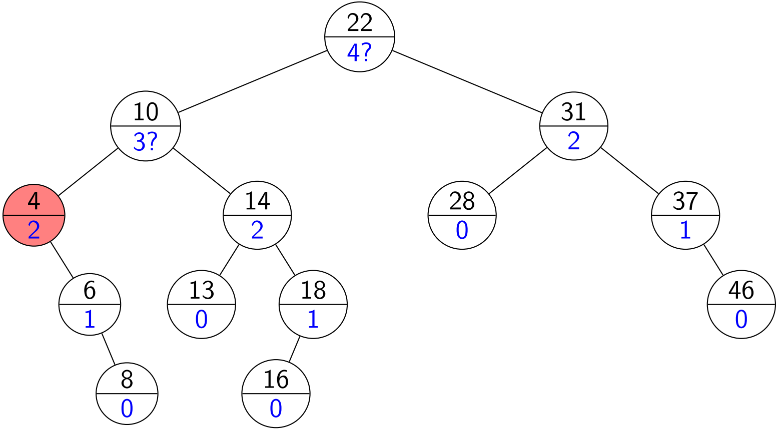
AVL Insertion Example

Example: *AVL::insert*(8)



AVL Insertion Example

Example: *AVL::insert*(8)

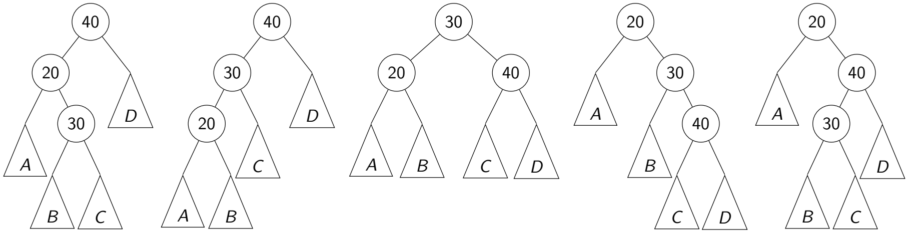


Outline

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
- Analysis of scapegoat trees

Changing structure without changing order

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

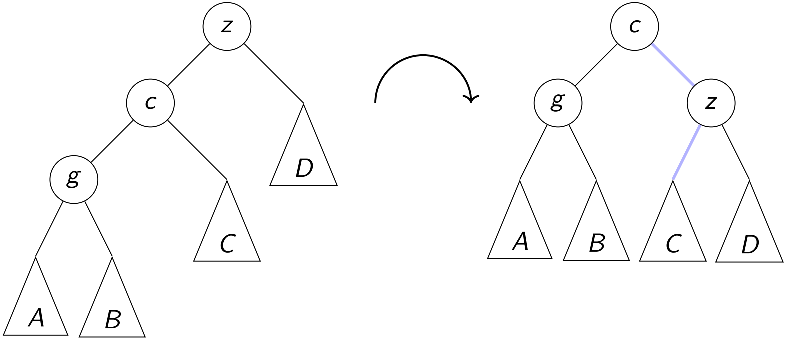


Goal: Change the *structure* locally nodes without changing the *order*.

Longterm goal: Restructure such the subtree becomes balanced.

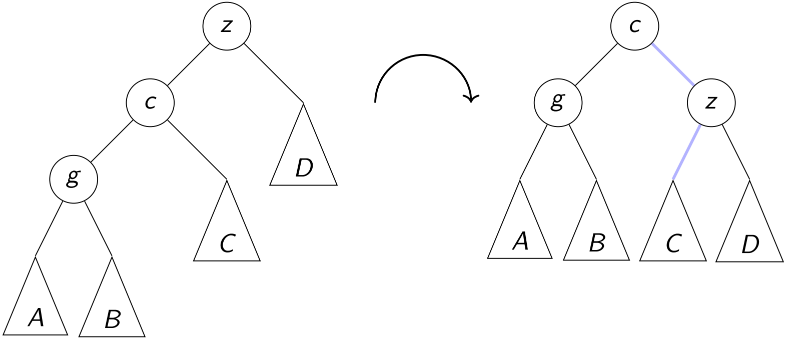
Right Rotation

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



Right Rotation

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



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

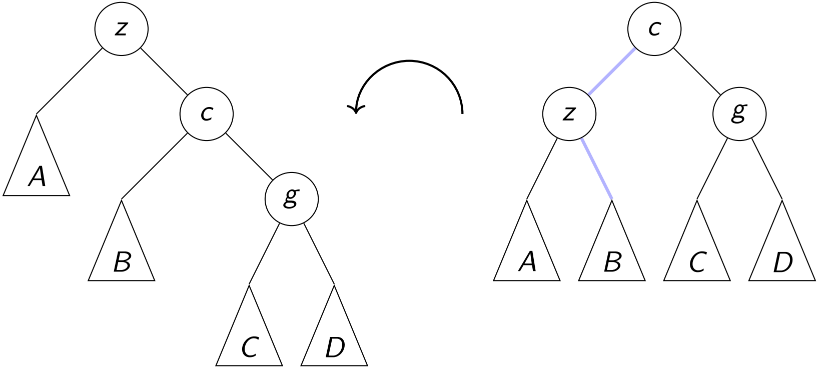
Right Rotation Pseudocode

rotate-right(z)
$c \leftarrow z.left$
`// fix links connecting to above`
$c.parent \leftarrow (p \leftarrow z.parent)$
if $p = NULL$ **then** $root \leftarrow c$ **else**
    if $p.left = z$ **then** $p.left \leftarrow c$ **else** $p.right \leftarrow c$
`// actual rotation`
$z.left \leftarrow c.right$, $c.right.parent \leftarrow z$
$c.right \leftarrow z$, $z.parent \leftarrow c$
set-height-from-subtrees(z), *set-height-from-subtrees*(c)
return c `// returns new root of subtree`

Run-time: $O(1)$

Left Rotation

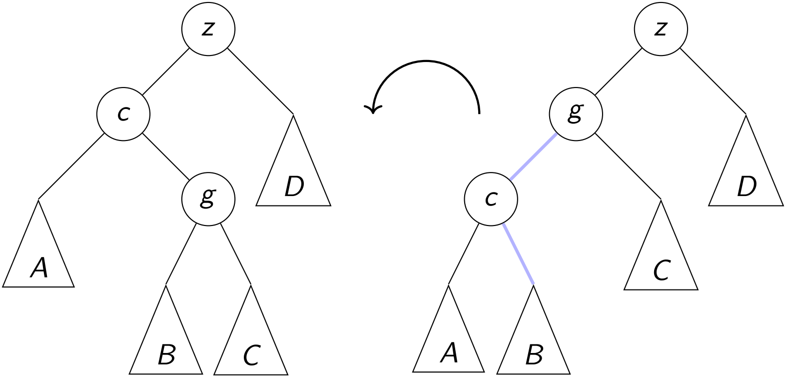
Symmetrically, this is a **left rotation** on node z:



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

Double Right Rotation

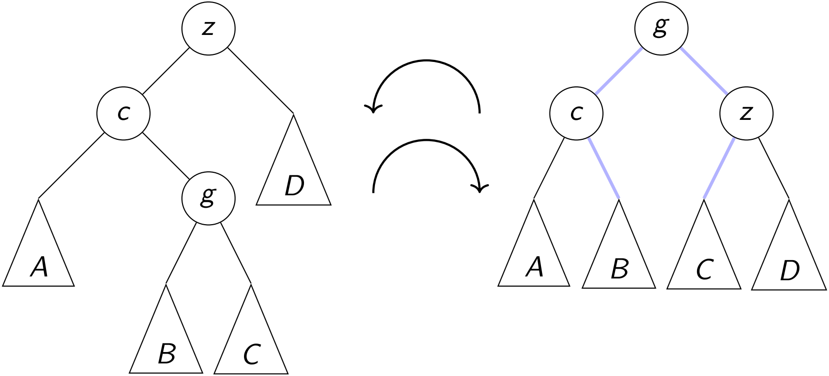
This is a **double right rotation** on node z:



First, a left rotation at c.

Double Right Rotation

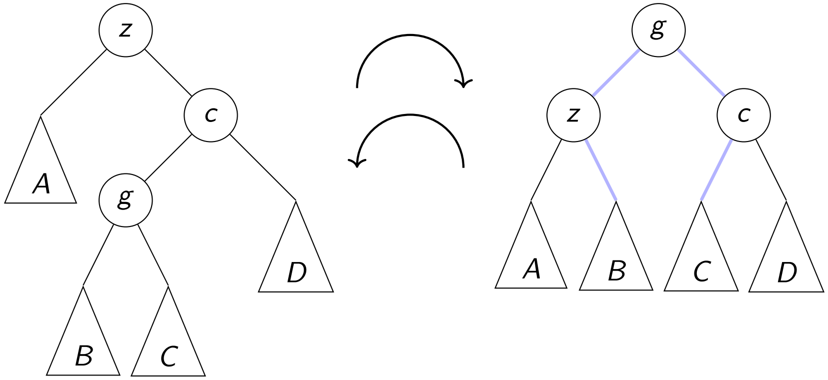
This is a **double right rotation** on node z:



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:



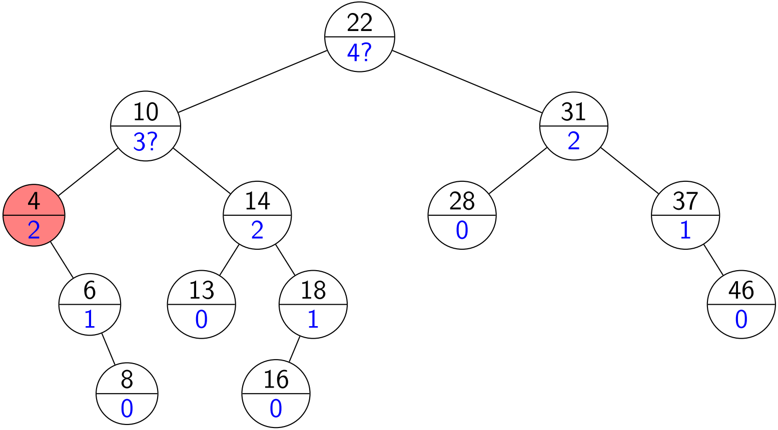
First, a right rotation at c.
Second, a left rotation at z.

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
   - Analysis of scapegoat trees

AVL Insertion Example revisited

Example: *AVL::insert*(8)



AVL 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)   // leaf where k is now stored
    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
```

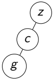
Can argue: For insertion *one* rotation restores all heights of subtrees.

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

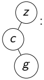
Fixing a slightly-unbalanced AVL tree

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

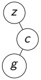
1. **case**



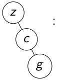
: // Right rotation
$u \leftarrow$ *rotate-right*(z)



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



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



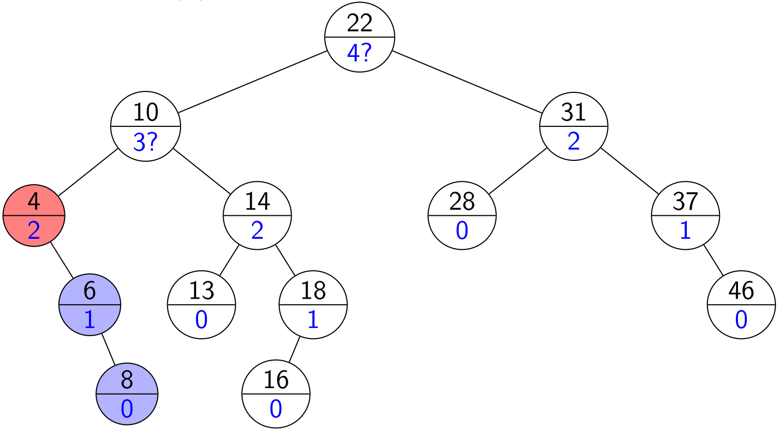
: // Left rotation
$u \leftarrow$ *rotate-left*(z)

2. **return** u

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

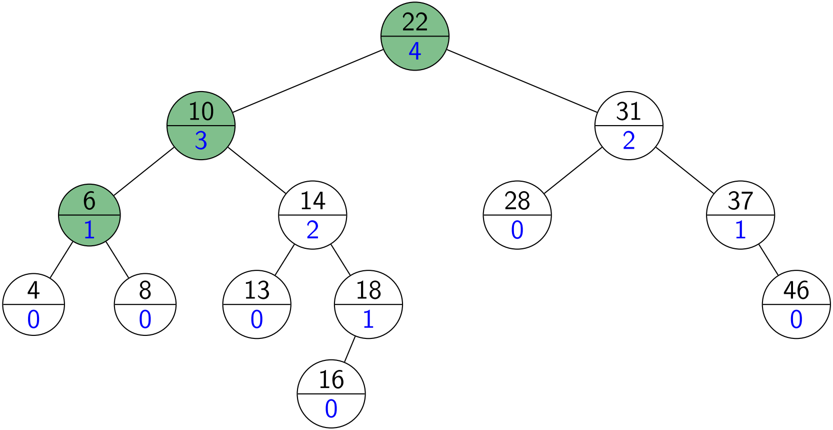
AVL Insertion Example revisited

Example: *AVL::insert*(8)



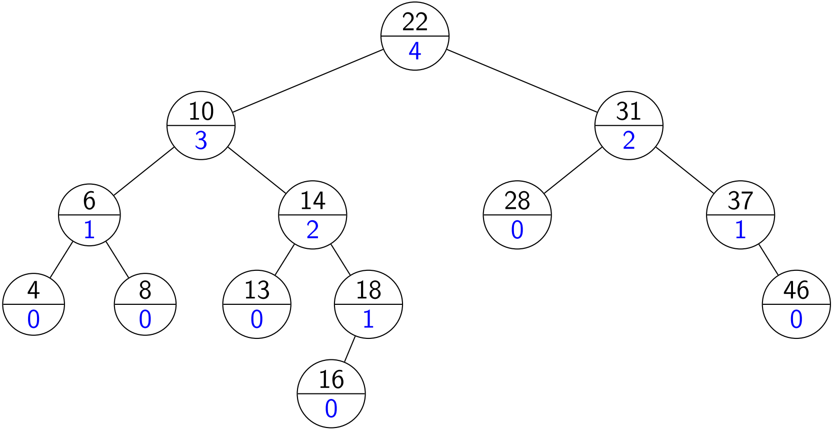
AVL Insertion Example revisited

Example: *AVL::insert*(8)



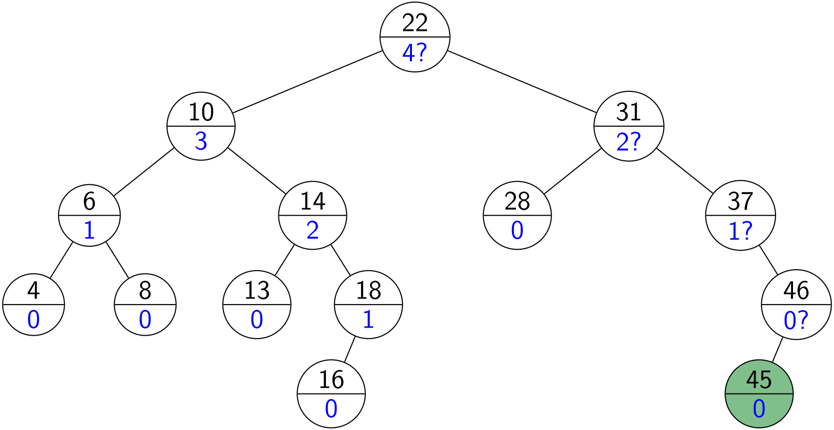
AVL Insertion: Second example

Example: *AVL::insert*(45)



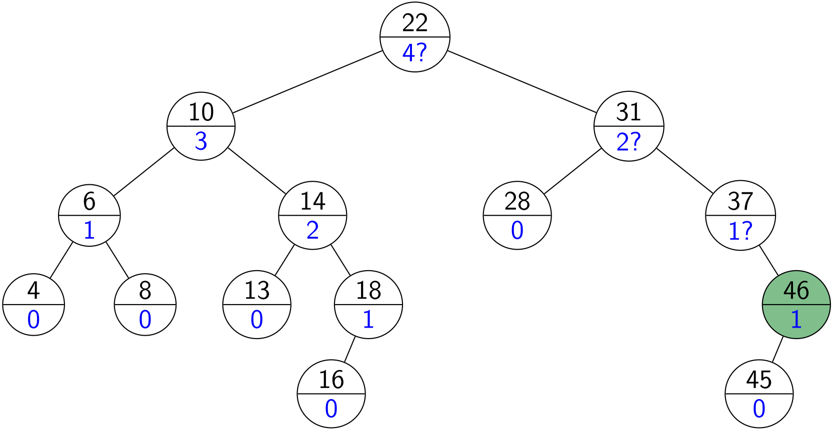
AVL Insertion: Second example

Example: *AVL::insert*(45)



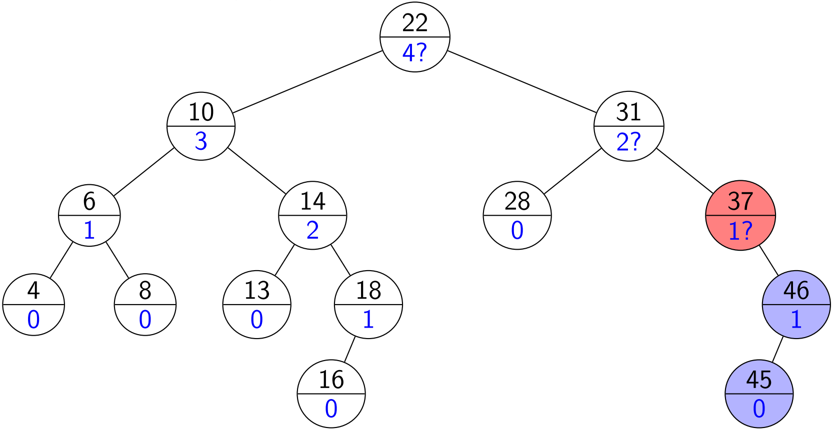
AVL Insertion: Second example

Example: *AVL::insert*(45)



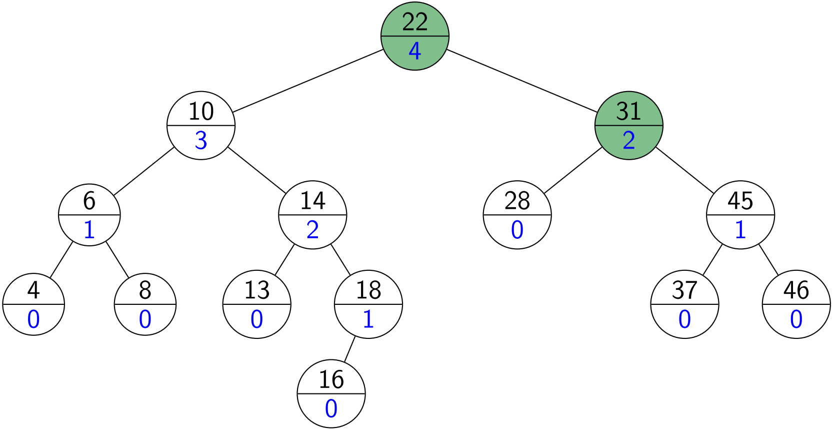
AVL Insertion: Second example

Example: *AVL::insert*(45)



AVL Insertion: Second example

Example: *AVL::insert*(45)

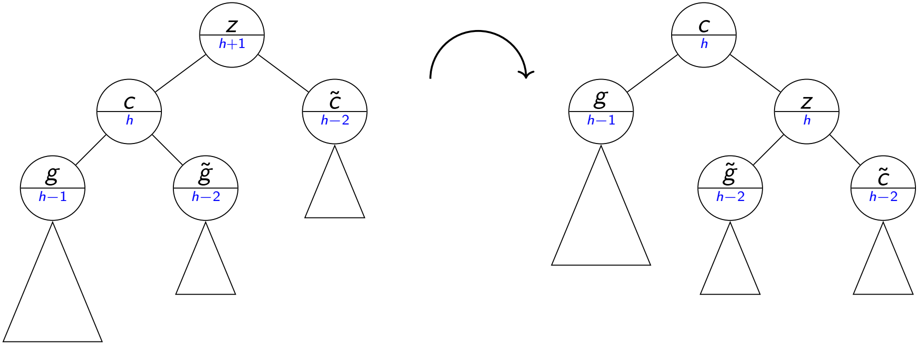


Correctness of rotations

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

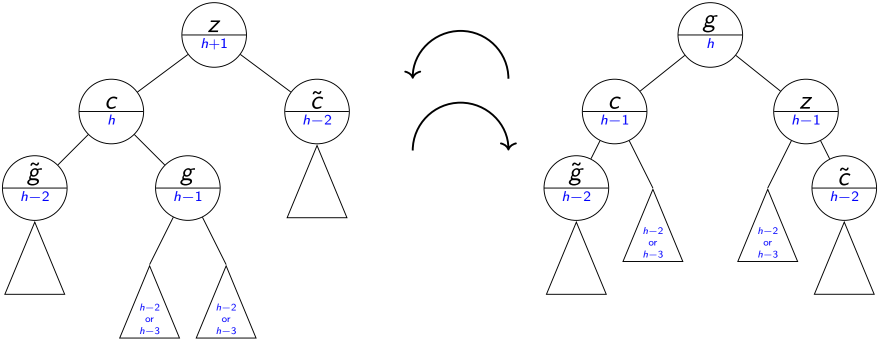
Proof: By symmetry, assume that c is the left child of z.

Case 1: g was the left child of c



Correctness of rotations

Case 2: g is the right child of c.



AVL Tree 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)

Outline

Scapegoat trees

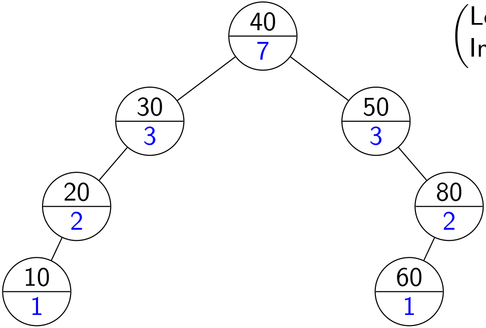
- Can we have balanced binary search trees *without* rotations?
  (A later application will need such a tree.)
- This sounds impossible—we must sometimes restructure the tree.
- **Idea:** Rather than doing a small local change, occasionally rebuild an entire (large) subtree.

- 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

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



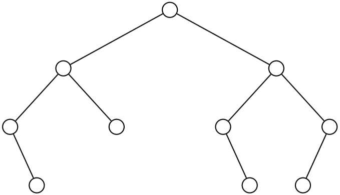
$\left(\begin{array}{l}\text{Lower number} = \text{subtree-size.} \\ \text{In our examples, } \alpha = \frac{2}{3}.\end{array}\right)$

- $v.\mathit{size}$ 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 (size-)balanced tree**:

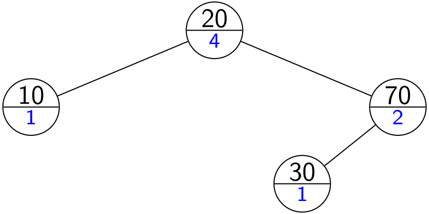
$$|size(z.left) - size(z.right)| \leq 1 \quad \text{for all nodes } z.$$



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

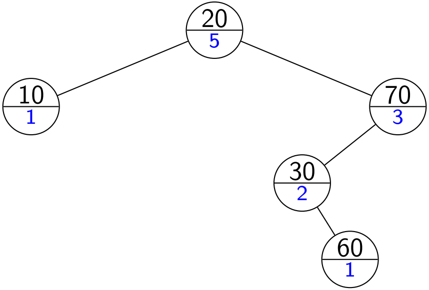
Scapegoat Tree Insertion Example

Example: *Scapegoat::insert*(60)



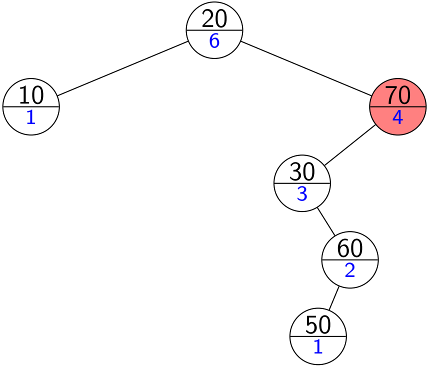
Scapegoat Tree Insertion Example

Example: *Scapegoat::insert*(60)



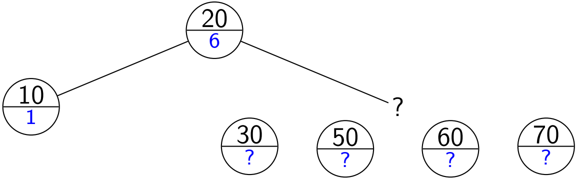
Scapegoat Tree Insertion Example

Example: *Scapegoat::insert*(50)



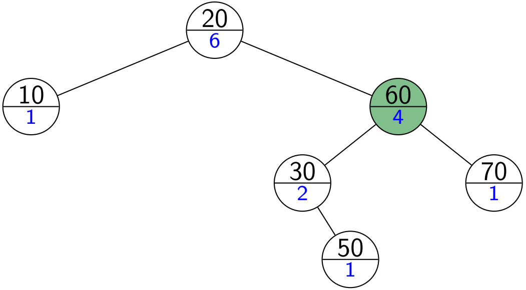
Scapegoat Tree Insertion Example

Example: *Scapegoat::insert*(50)



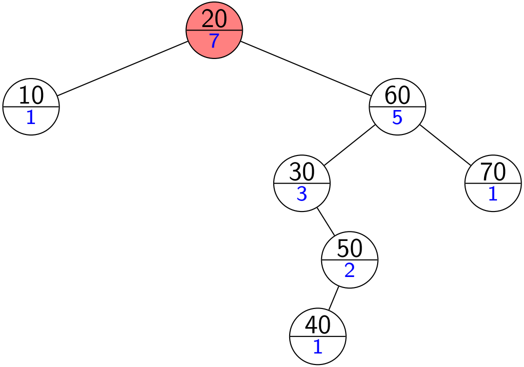
Scapegoat Tree Insertion Example

Example: *Scapegoat::insert*(50)



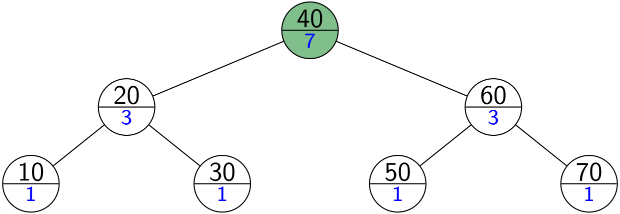
Scapegoat Tree Insertion Example

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 ≠ NULL)    // 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

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 bounds up over all operations.

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

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.

Detour: Amortized analysis

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

$$\boxed{40\,|\,20\,|\quad|\quad} \xrightarrow{\text{insert}} \boxed{40\,|\,20\,|\,90\,|\quad} \xrightarrow{\text{insert}} \boxed{40\,|\,20\,|\,90\,|\,60} \xrightarrow{\text{resize}} \boxed{40\,|\,20\,|\,90\,|\,60\,|\quad|\quad|\quad|\quad}$$

- $n/2$ 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.

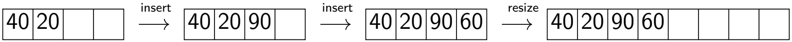
More systematic method: Use a **potential function**

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

Potential function method

- Define **time units**: how much can be done in one unit of time?
  - Needed so that we do not do math with asymptotic notation.
  - Dynamic arrays: "Set time units such that
    $$T^{\text{actual}}(\textit{insert}) \leq 1 \text{ and } T^{\text{actual}}(\textit{resize}) \leq n."$$
- Define a potential function Φ and verify $\Phi(0) = 0, \Phi(i) \geq 0$.
  - 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



- Set time units such that $T^{\text{actual}}(\textit{insert}) \leq 1$ and $T^{\text{actual}}(\textit{resize}) \leq n$.

Example: Dynamic arrays

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

- Set time units such that $T^{\mathrm{actual}}(\mathit{insert}) \le 1$ and $T^{\mathrm{actual}}(\mathit{resize}) \le n$.
- Potential function $\Phi(t) = \max\{0, 2 \cdot \mathit{size} - \mathit{capacity}\}$

Example: Dynamic arrays

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

- Set time units such that $T^{\text{actual}}(\mathit{insert}) \leq 1$ and $T^{\text{actual}}(\mathit{resize}) \leq n$.
- Potential function $\Phi(t) = \max\{0, 2 \cdot \mathit{size} - \mathit{capacity}\}$
- *insert* increases *size*, does not change *capacity*

  $\Rightarrow$ $\Phi^{\text{new}} - \Phi^{\text{old}} \leq 2$ and $T^{\text{amort}}(\mathit{insert}) \leq 1 + 2 = 3 \in O(1)$

Example: Dynamic arrays

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

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

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

Example: Dynamic arrays

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

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

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?

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

  - Dynamic arrays: *resize* *increases* capacity.
    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?

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

  - Dynamic arrays: *resize* *increases* capacity.
    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 $\Phi(0) = 0$.

Potential function method

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

- Study the expensive operation: What gets smaller?

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

  - Dynamic arrays: *resize* *increases* capacity.
    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.
- Lather, rinse, repeat.

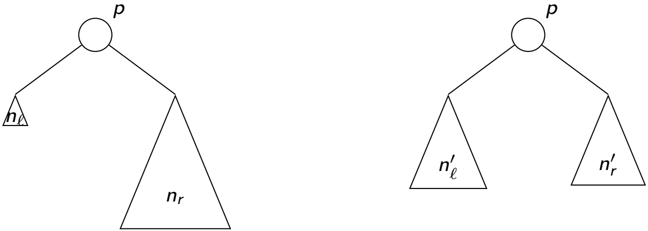
Outline

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
- Analysis of scapegoat trees

Amortized analysis of scapegoat trees

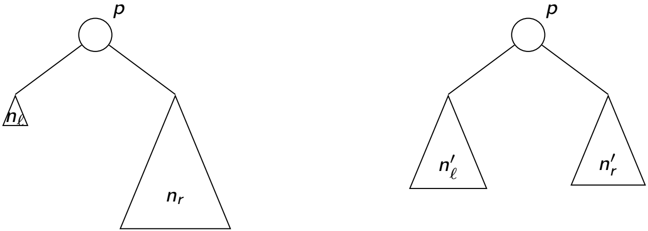
- Expensive operation: Rebuild subtree at p.



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

Amortized analysis of scapegoat trees

- Expensive operation: Rebuild subtree at p.



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

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

Amortized analysis of scapegoat trees

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

Amortized analysis of scapegoat trees

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

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

Amortized analysis of scapegoat trees

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

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

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

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

Amortized analysis of scapegoat trees

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

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

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

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

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

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