CMSC 341

Amortized Analysis

What is amortized analysis?

- Consider a sequence of operations on a dynamic data structure
  - Insert or delete in any (valid) order
- Worst case analysis asks: What is the most expensive any single operation can be?
- Amortized analysis asks: What is an upper bound on the average per-operation cost over the entire sequence?

Nature of the amortized bound

- Amortized bounds are hard bounds
- They do not mean "on average (or most of the time) the bound holds"
- They do mean "for *any* sequence of n operations, the bound holds over that sequence"

Three methods

- Aggregate method
  - T(n) = upper bound on total cost of n operations
  - Amortized cost is T(n)/n
  - Some operations may cost more, a lot more, than T(n)/n
  - If so, some operations must cost less
  - But the average cost over the sequence will never exceed T(n)/n

Three methods

- Accounting method
  - Each operation pays a “fee” (cost of operation)
  - Overcharge some operations and store extra as pre-payment for later operations
  - Amortized cost is (total of fees paid)/n
  - Must ensure bank account never negative, otherwise fee was not high enough and bound does not hold
  - Overpayment stored with specific objects in data structure (e.g., nodes in a BST)

Three methods (cont.)

- Potential method
  - Like accounting method
  - Overpayment stored as “potential energy” of entire data structure (not specific objects)
  - Must ensure that potential energy never falls below zero

Increment a binary counter

A(k-1)	A(k-2)						A(2)	A(1)	A(0)
0	0	0	0	0	0	0	0	0	0

- K-bit value stored in array
- To increment value, flip bits right-to-left until you turn a 0 into a 1
- Each bit flip costs O(1)
- What is the amortized cost of counting from 0 to n?

Increment a binary counter

A(k-1)	A(k-2)						A(2)	A(1)	A(0)
0	0	0	0	0	0	0	0	0	0

- Worst case
  - Flip k bits per increment
  - Do that n times to count to n
  - O(kn)
- But, most of the time we don't flip many bits

Increment a binary counter

A(k-1)	A(k-2)						A(2)	A(1)	A(0)	Cost
0	0	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	0	0	1	0	2
0	0	0	0	0	0	0	0	1	1	1
0	0	0	0	0	0	0	1	0	0	3
0	0	0	0	0	0	0	1	0	1	1
0	0	0	0	0	0	0	1	1	0	2
0	0	0	0	0	0	0	1	1	1	1
0	0	0	0	0	0	1	0	0	0	4

Total: 15

Aggregate method

- A(0) flips every time, or n times
- A(1) flips every 2nd time, or n/2 times
- A(2) flips every 4th time, or n/4 times
- A(i) flips $n/2^i$ times

- Total cost is $\Sigma_{i=0,k-1} n/2^i \le \Sigma_{i=0,\infty} n/2^i = 2n = O(n)$
- So amortized cost is $O(n)/n = O(1)$

Accounting method

- Flipping a bit costs \$1 (one unit of computational work)
- Pay \$2 to change a 0 to a 1
  - Use \$1 to pay for flipping the bit to 1
  - Leave \$1 there to pay when/if the bit gets flipped back to 0
- Since only one bit gets flipped to 1 per increment, total cost is \$2n = O(n)

Accounting method

A(k-1)	A(k-2)						A(2)	A(1)	A(0)
0	0	0	0	0	0	0	0	0	0
\$0	\$0	\$0	\$0	\$0	\$0	\$0	\$0	\$0	\$0

- Flip A(0) to 1 and pay \$1 for flip and leave \$1 with that bit (total fee of \$2)

A(k-1)	A(k-2)						A(2)	A(1)	A(0)
0	0	0	0	0	0	0	0	0	1
\$0	\$0	\$0	\$0	\$0	\$0	\$0	\$0	\$0	\$1

Accounting method

A(k-1)	A(k-2)						A(2)	A(1)	A(0)
0	0	0	0	0	0	0	0	0	1
$0	$0	$0	$0	$0	$0	$0	$0	$0	$1

- Flip A(0) to 0 and pay with the $1 that was there already
- Flip A(1) to 1 and pay $1 for flip and leave $1 with that bit (total fee of $2)

A(k-1)	A(k-2)						A(2)	A(1)	A(0)
0	0	0	0	0	0	0	0	1	0
$0	$0	$0	$0	$0	$0	$0	$0	$1	$0

Accounting method

A(k-1)	A(k-2)						A(2)	A(1)	A(0)
0	0	0	0	0	0	0	0	1	0
\$0	\$0	\$0	\$0	\$0	\$0	\$0	\$0	\$1	\$0

- Flip A(0) to 1 and pay \$1 for flip and leave \$1 with that bit (total fee of \$2)

A(k-1)	A(k-2)						A(2)	A(1)	A(0)
0	0	0	0	0	0	0	0	1	1
\$0	\$0	\$0	\$0	\$0	\$0	\$0	\$0	\$1	\$1

Accounting method

A(k-1)	A(k-2)						A(2)	A(1)	A(0)
0	0	0	0	0	0	0	0	1	1
\$0	\$0	\$0	\$0	\$0	\$0	\$0	\$0	\$1	\$1

- Flip A(0) and A(1) to 0 and pay with the \$\$ that were there already
- Flip A(2) to 1 and pay \$1 for flip and leave \$1 with that bit (total fee of \$2)

A(k-1)	A(k-2)						A(2)	A(1)	A(0)
0	0	0	0	0	0	0	1	0	0
\$0	\$0	\$0	\$0	\$0	\$0	\$0	\$1	\$0	\$0

Accounting method

- … and so on
- We “overpay” by $1 for flipping each 0 to 1
- Use the extra $1 to pay for the cost of flipping it back to a zero
- Because a $2 fee for each increment ensures that we have enough money stored to complete that increment, amortized cost is $2 = O(1) per operation

Potential method

- Record overpayments as "potential energy" (or just "potential") of entire data structure
- Contrast with accounting method where overpayments stored with specific parts of data structure (e.g., array cells)

Potential method

- Initial data structure is D_0
- Perform operations i = 1, 2, 3, …, n
- The actual cost of operation i is c_i
- The i^{th} operation yields data structure D_i

- $\Phi(D_i)$ = potential of D_i, or stored overpayment
- Amortized cost of the i^{th} operation is
  - $x_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

Potential method

- If $\Phi(D_i) - \Phi(D_{i-1}) > 0$ then x_i is an overcharge to i^{th} operation
  - We paid more than the actual cost of the operation
- If $\Phi(D_i) - \Phi(D_{i-1}) < 0$ then x_i is an undercharge to the i^{th} operation
  - We paid less than the actual cost of the operation, but covered the difference by spending potential

Potential method

- Total amortized cost:
  - $\Sigma x_i = \Sigma(c_i + \Phi(D_i) - \Phi(D_{i-1})) = \Sigma c_i + \Phi(D_n) - \Phi(D_0)$
  - Sum of actual costs plus whatever potential we added but didn't use

- Require that $\Phi(D_i) \geq 0$ so we always "pay in advance"

Potential method: Binary counter

- Need to choose potential function $\Phi(D_i)$
- Want to make $x_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$ small
- Usually have to be lucky or clever!
- Let $\Phi(D_i) = b_i$, the number of ones in the counter after the i^{th} operation
  - Note that $\Phi(D_i) \geq 0$ so we're OK
  - Recall $1 stored with each 1 in the array when using the accounting method

Potential method: Binary counter

- Operation i resets (zeroes) t_i bits
- True cost of operation i is $t_i + 1$
  - The +1 is for setting a single bit to 1
- Number of ones in counter after i^{th} operation is therefore $b_i = b_{i-1} – t_i + 1$

Potential method: Binary counter

- Number of ones in counter after i^{th} operation is $b_i = b_{i-1} - t_i + 1$
- Potential difference is
  - $\Phi(D_i) - \Phi(D_{i-1}) = b_i - b_{i-1} = (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i$
- Amortized cost is
  - $x_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = (t_i + 1) + (1 - t_i) = 2$
  - If we pay just $2 per operation, we always have enough potential to cover our actual costs per operation

Amortized analysis of splay trees

- Use the accounting method
  - Store $$ with nodes in tree
- First, some definitions
  - Let n_x be the number of nodes in the subtree rooted by node x
  - Let $r_x = \text{floor}(\log(n_x))$
    - Called the rank of x

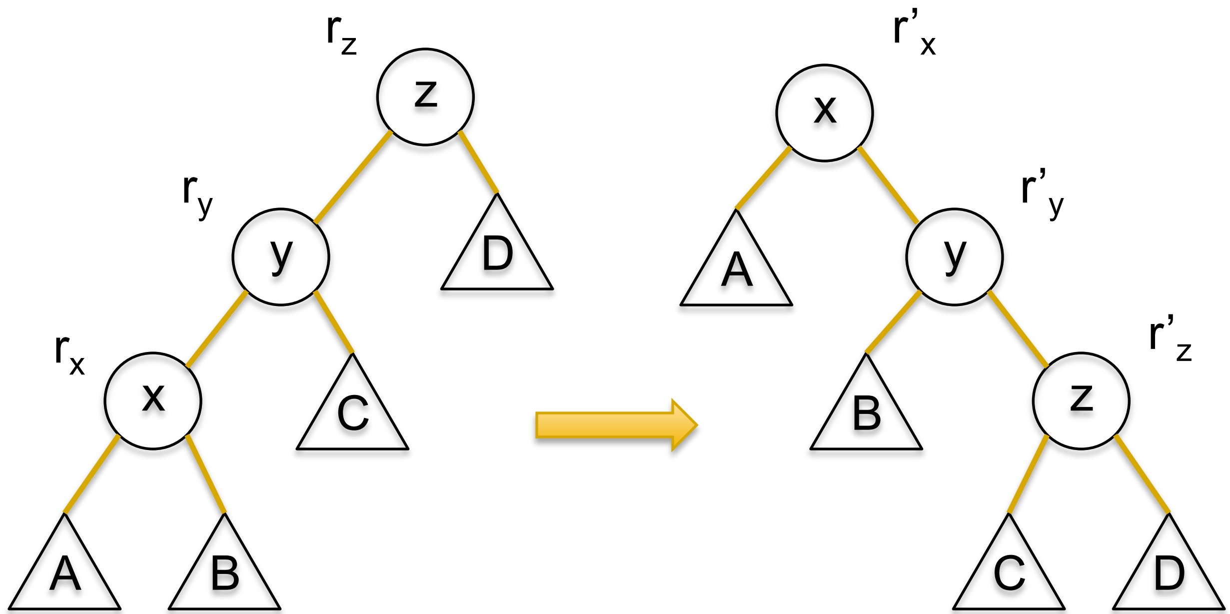
What we'll show

- If every node x always has r_x credits then splay operations require amortized O(lgn) time
- This is called the "credit invariant"
- For each operation (find, insert, delete) we'll have to show that we can maintain the credit invariant and pay for the true cost of the operation with O(lgn) $$ per operation

First things first

- Consider a single splay step
  - Single rotation (no grandparent), zig-zig, or zig-zag
  - Nodes x, y = parent(x), z = parent(y)
  - r_x, r_y, and r_z are ranks before splay step
  - r'_x, r'_y, and r'_z are ranks after splay step

For example (zig-zig case)



What does a single splay step cost?

- To pay for rotations (true cost of step) and maintain credit invariant
  - $3(r'_x - r_x) + 1$ credits suffice for single rotation
  - $3(r'_x - r_x)$ credits suffice for zig-zig and zig-zag

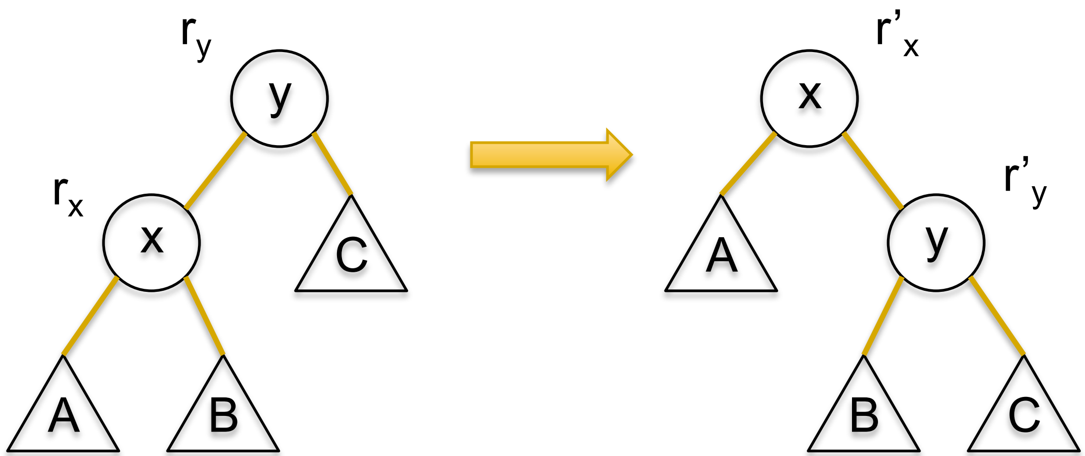
What does a sequence of splay steps cost?

- As node x moves up the tree, sum costs of individual steps
- r'_x for one step becomes r_x for next step
- Summing over all steps to the root telescopes to become $3(r_v - r_x) + 1$ where v is the root node
  - $3(r'_x - r_x) + 3(r''_x - r'_x) + 3(r'''_x - r''_x) \ldots + 1$
  - Note +1 only required (sometimes) for last step

The punch line!

- $3(r_v - r_x) + 1 = O(\log n)$
  - v is root node of tree with n nodes
  - $r_v = \text{floor}(\log n)$
- We can splay any node to the root in O(logn) time

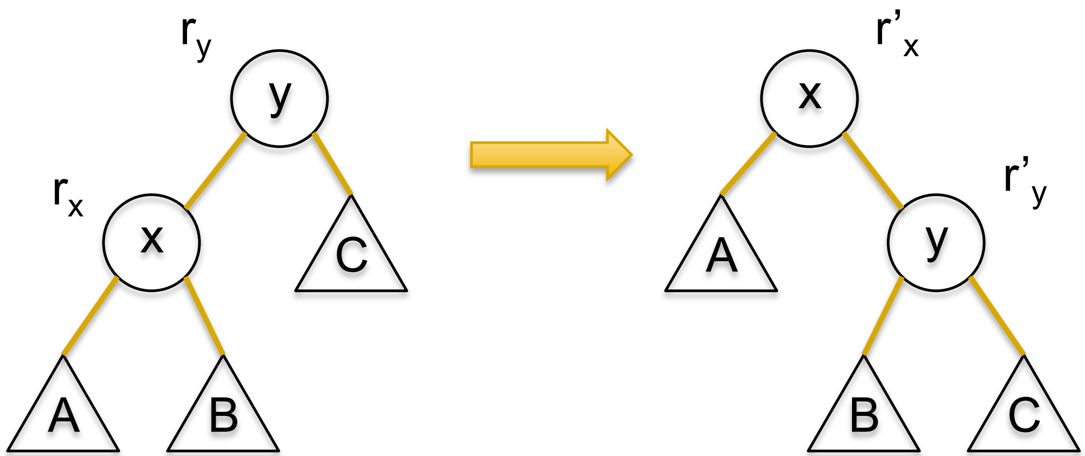
Single rotation



To maintain credit invariant at all nodes need to add $\Delta

- Only r_x and r_y can change
- $\Delta = (r'_x - r_x) + (r'_y - r_y)$
- Note that $r'_x = r_y$ so …
- $\Delta = r'_y - r_x$
- Note that $r'_x \geq r'_y$ so …
- $\Delta = r'_y - r_x \leq r'_x - r_x$

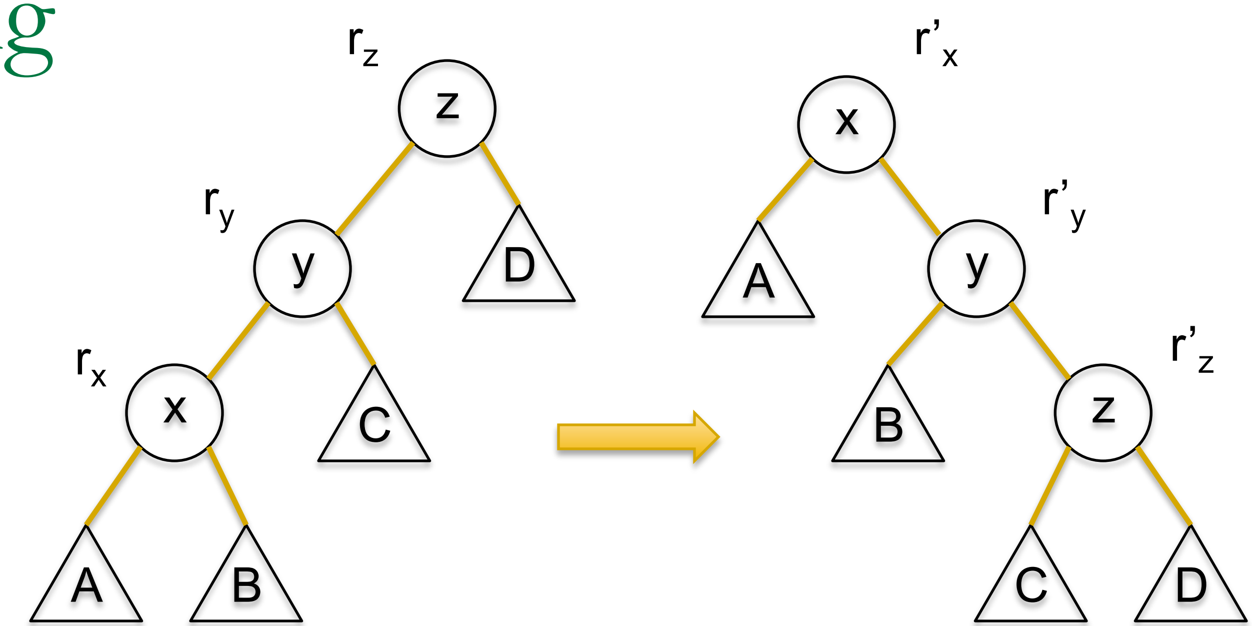
Single rotation



To maintain credit invariant at all nodes it suffices to pay \$($r'_x - r_x$)

- Still need to pay O(1) for the rotation
- We allocated $3(r'_x - r_x) + 1$ credits
- If $r'_x > r_x$ we've still got $2(r'_x - r_x) > 1$ credits to pay for the rotation
- The +1 is there in case $r'_x = r_x$
  - When can that happen?

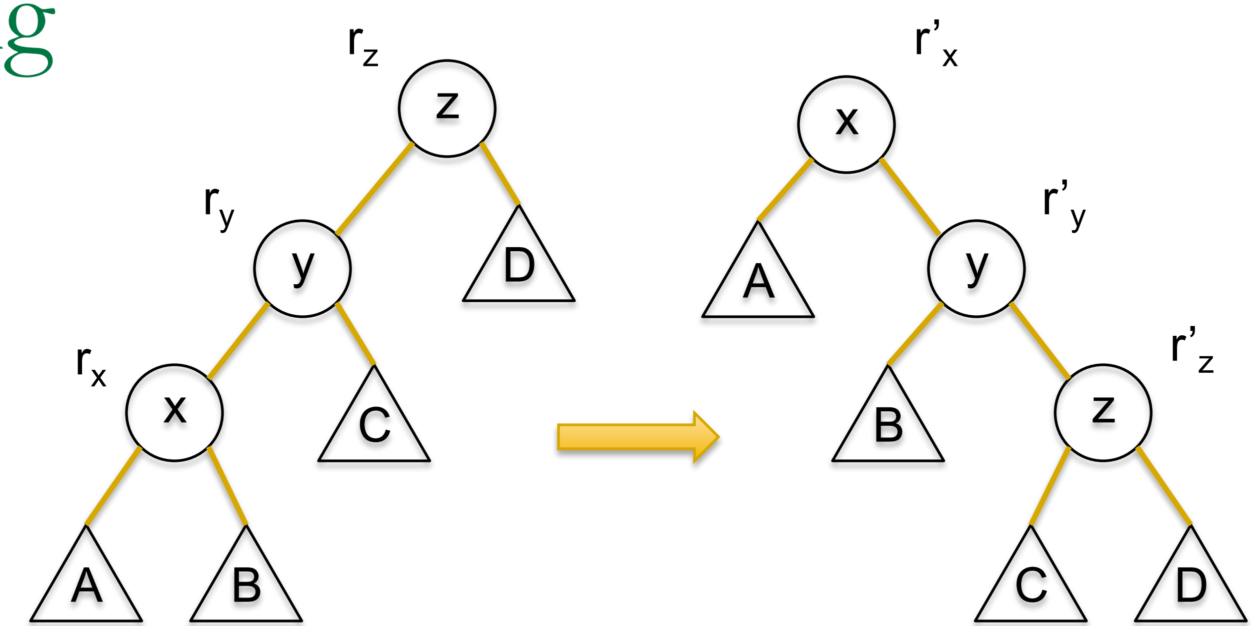
Zig-zig



To maintain credit invariant at all nodes need to add $\Delta

- $\Delta = (r'_x – r_x) + (r'_y – r_y) + (r'_z – r_z)$
- Note that $r'_x = r_z$ so …
- $\Delta = r'_y + r'_z – r_x – r_y$
- Note that $r'_x \geq r'_y$ and $r'_x \geq r'_z$ and $r_x \leq r_y$ so …
- $\Delta = r'_y + r'_z – r_x – r_y \leq r'_x + r'_x – r_x – r_x = 2(r'_x – r_x)$

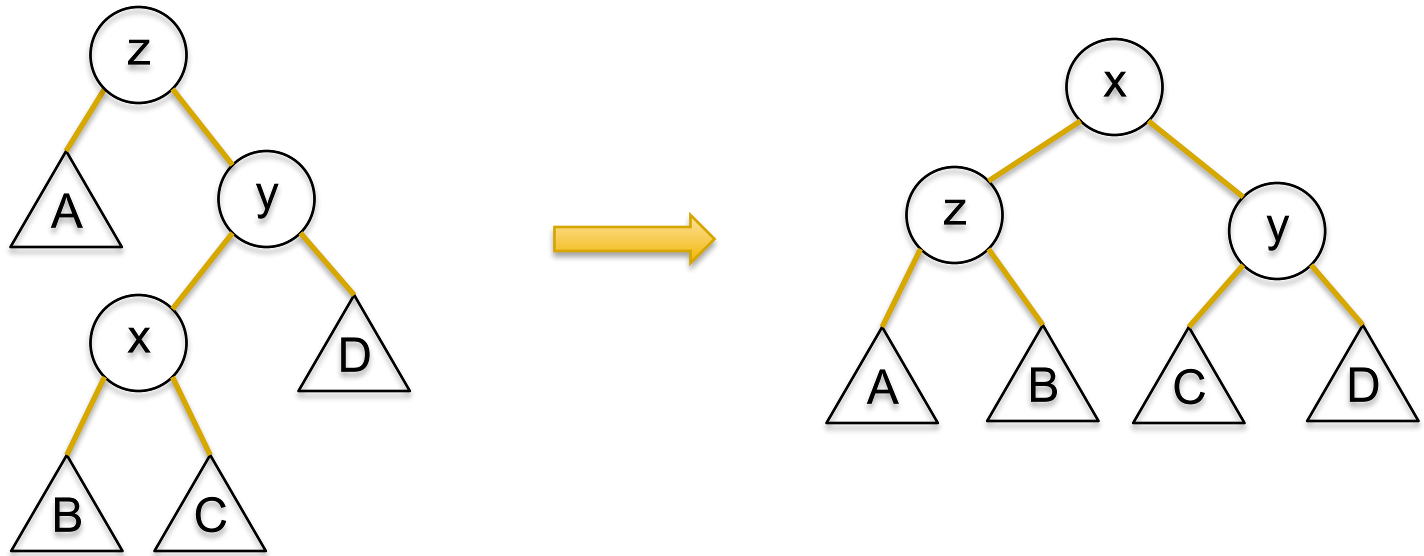
Zig-zig



To maintain credit invariant at all nodes it suffices to pay \$2($r'_x - r_x$)

- Still need to pay O(1) for the rotations
- If $r'_x > r_x$ we can use $r'_x - r_x \geq 1$ credits to pay for the two rotations for a total of \$3($r'_x - r_x$)
- Otherwise, $r'_x = r_x$ so $r'_x = r_x = r_y = r_z$
    - Why?
- In this case, we can show that maintaining the invariant frees one or more credits that can be used to pay for the rotations

Zig-zag



- Analysis analogous to zig-zig step
- At most \$3($r'_x - r_x$) required to maintain invariant and pay for rotations