Amortized Analysis and Splay Trees

Inge Li Gørtz

CLRS Chapter 17
Jeff Erickson notes

Today

- Amortized analysis
    - Multipop-stack
    - Dynamic tables
    - Splay trees

Dynamic tables

- Problem. Have to assign size of table at initialization.
- Goal. Only use space Θ(n) for an array with n elements.
- Applications. Stacks, queues, hash tables,….

- Can insert and delete elements at the end.

Dynamic tables

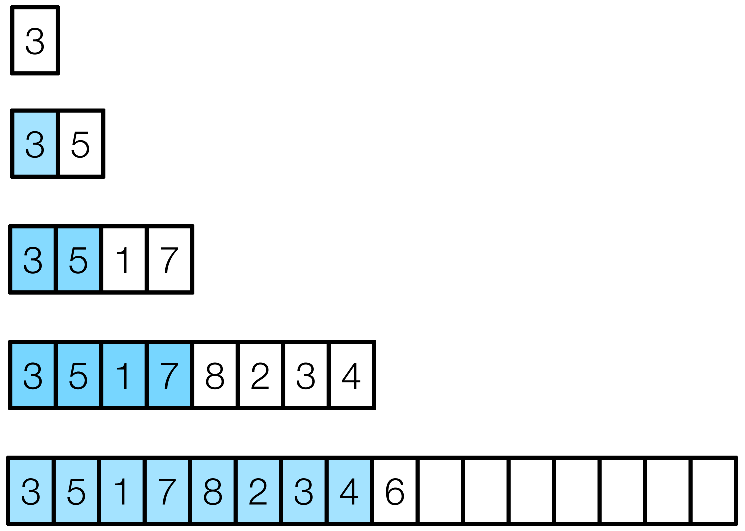
- First attempt.
    - Insert:
        - Create a new table of size n+1.
        - Move all elements to the new table.
        - Delete old table.
    - Size of table = number of elements

- Too expensive.
    - Have to copy all elements to a new array each time.
    - Insertion of N elements takes time proportional to: $1 + 2 + \cdots\cdots + n = \Theta(n^2)$.

- Goal. Ensure size of array does not change to often.

Dynamic tables

- Doubling. If the array is full (number of elements equal to size of array) copy the elements to a new array of double size.



- Consequence. Insertion of n elements take time:
    - n + number of reinsertions = n + 1 + 2 + 4 + 8 + ···· + $2^{\log n}$ < 3n.
    - Space: Θ(n).

Amortized Analysis

- Amortized analysis.
  - Average running time per operation over a *worst-case* sequence of operations.
- Methods.
  - Summation (aggregate) method
  - Accounting (tax) method
  - Potential method

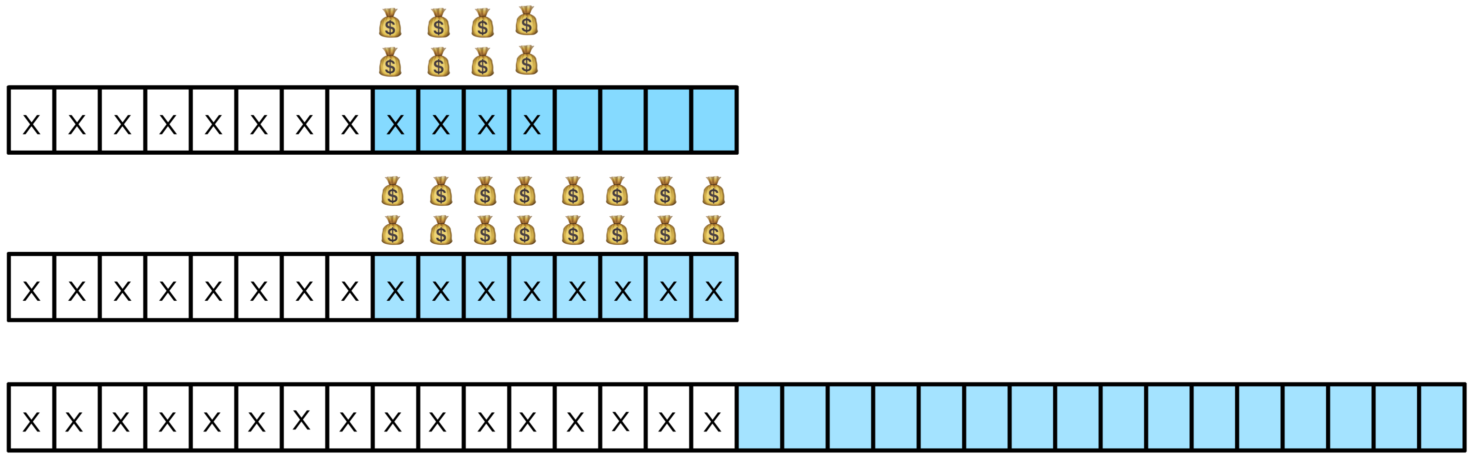
Summation (Aggregate) method

- Summation.
  - Determine total cost.
  - Amortized cost = total cost/#operations.

- Analysis of doubling strategy (without deletions):
  - Total cost: $n + 1 + 2 + 4 + \ldots + 2^{\log n} = \Theta(n)$.
  - Amortized cost per insert: $\Theta(1)$.

Dynamic Tables: Accounting Method

- Analysis: Allocate 2 credits to each element when inserted.
  - All elements in the array that is beyond the middle have 2 credits.
  - Table not full: insert costs 1, and we have 2 credits to save.
  - table full, i.e., doubling: half of the elements have 2 credits each. Use these to pay for reinsertion of all in the new array.
  - Amortized cost per operation: 3.

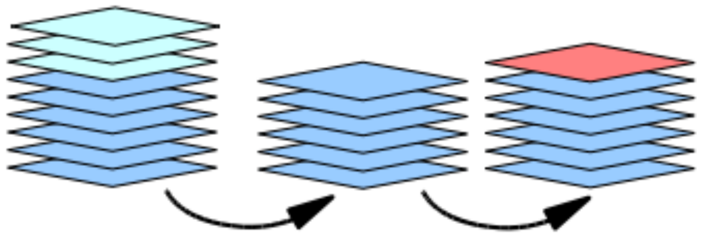

x x x x x x x x x x x x
x x x x x x x x x x x x x x x x
x x x x x x x x x x x x x x x x

Accounting method

- Accounting/taxation.
  - Some types of operations are overcharged (taxed).
  - Amortized cost of an operation is what we charge for an operation.
  - Credit allocated with elements in the data structure can be used to pay for later operations (only in analysis, not actually stored in data structure!).
  - Total credit must be non-negative at all times.
  - => Total amortized cost an upper bound on the actual cost.

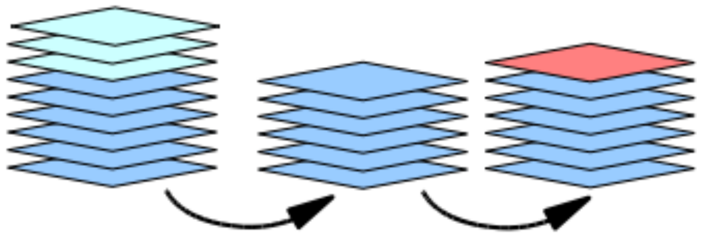
Example: Stack with MultiPop

- Stack with MultiPop.
  - Push(e): push element e onto stack.
  - MultiPop(k): pop top k elements from the stack
- Worst case: Implement via linked list or array.
  - Push: O(1).
  - MultiPop: O(k).
- Can prove amortized cost per operation: 2.



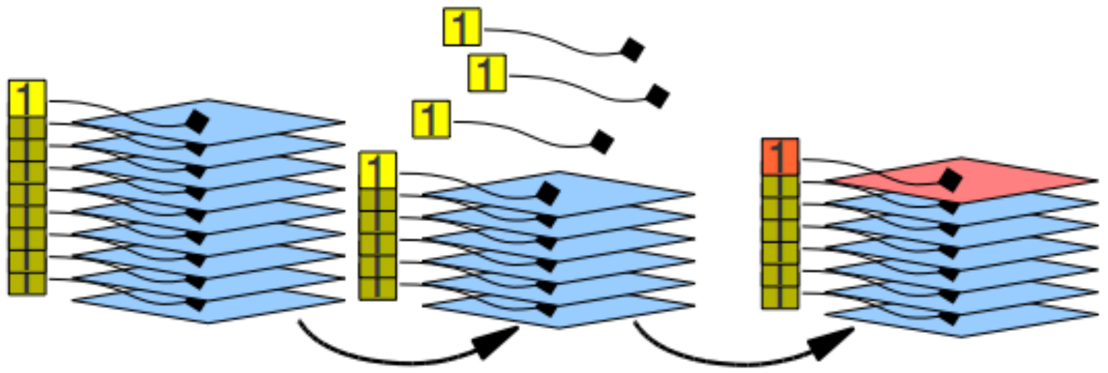
Stack: Aggregate Analysis

- Amortized analysis. Sequence of n Push and MultiPop operations.
  - Each object popped at most once for each time it is pushed.
  - #pops on non-empty stack ≤ #Push operations ≤ n.
  - Total time O(n).
- Amortized cost per operation: 2n/n = 2.



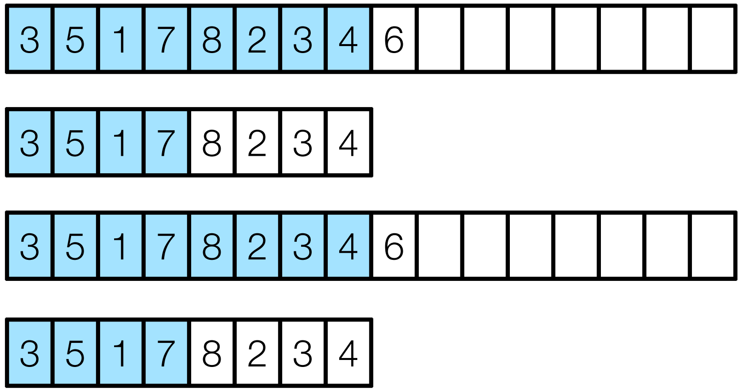
Stack: Accounting Method

- Amortized analysis. Sequence of n Push and MultiPop operations.
    - Pay 2 credits for each Push.
    - Keep 1 credit on each element on the stack.
- Amortized cost per operation:
    - Push: 2
    - MultiPop: 1 (to pay for pop on empty stack).



Dynamic tables with deletions

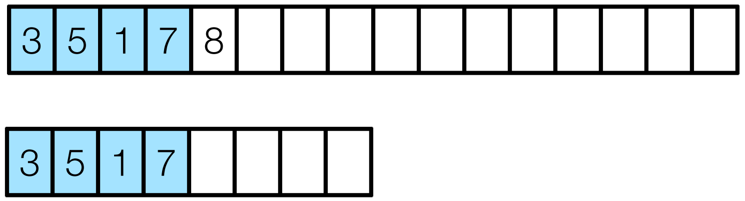
- Halving (first attempt). If the array is half full copy the elements to a new array of half the size.



- Consequence. The array is always between 50% and 100% full. **But** risk to use too much time (double or halve every time).

Dynamic tables

- Halving. If the array is a quarter full copy the elements to a new array of half the size.



- Consequence. The array is always between 25% and 100% full.

Potential method

- Potential method. Define a potential function for the data structure that is initially zero and always non-negative.
- Prepaid credit (potential) associated with the data structure (money in the bank).
- Ensure there is always enough "money in the bank" (non-negative potential).
- Amortized cost $\hat{c}_i$ of an operation: actual cost c_i plus change in potential.
    - $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
- Thus:

$$\sum_{i}^{m} \hat{c}_i = \sum_{i}^{m} \left(c_i + \Phi(D_i) - \Phi(D_{i-1})\right)$$

Dynamic tables

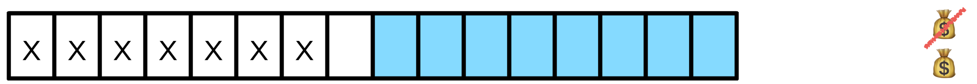
- Doubling. If the table is full (number of elements equal to size of array) copy the elements to a new array of double size.
- Halving. If the table is a quarter full copy the elements to a new array of half the size
- Potential function.
  - $\Phi(D_i) = \begin{cases} 2n - L & \text{if T at least half full} \\ L/2 - n & \text{if T less than half full} \end{cases}$
  - L = current array size, n = number of elements in array.

Dynamic tables

- Doubling. If the table is full (number of elements equal to size of array) copy the elements to a new array of double size.
- Halving. If the table is a quarter full copy the elements to a new array of half the size
- Potential function.
    - $\Phi(D_i) = \begin{cases} 2n - L & \text{if T at least half full} \\ L/2 - n & \text{if T less than half full} \end{cases}$
    - L = current array size, n = number of elements in array.

- Inserting when less than half full and still less than half full after insertion:

n = 7, L = 16



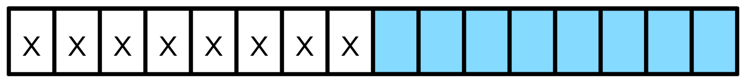
- amortized cost = 1 + - 💰 = 0

Dynamic tables

- Doubling. If the table is full (number of elements equal to size of array) copy the elements to a new array of double size.
- Halving. If the table is a quarter full copy the elements to a new array of half the size
- Potential function.
  - $\Phi(D_i) = \begin{cases} 2n - L & \text{if T at least half full} \\ L/2 - n & \text{if T less than half full} \end{cases}$
  - L = current array size, n = number of elements in array.

- Inserting when less than half full before and half full after:

n = 8, L = 16



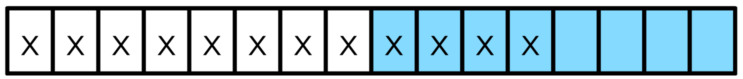
- amortized cost = 1 + - 💰 = 0

Dynamic tables

- Doubling. If the table is full (number of elements equal to size of array) copy the elements to a new array of double size.
- Halving. If the table is a quarter full copy the elements to a new array of half the size
- Potential function.
  - $\Phi(D_i) = \begin{cases} 2n - L & \text{if T at least half full} \\ L/2 - n & \text{if T less than half full} \end{cases}$
  - L = current array size, n = number of elements in array.

- Inserting when at least half full, but not full:

n = 12, L = 16



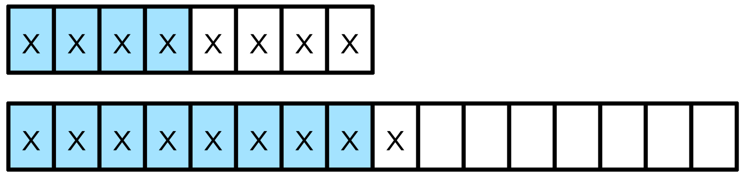
- amortized cost = 1 + 💰💰 = 3

Dynamic tables

- Doubling. If the table is full (number of elements equal to size of array) copy the elements to a new array of double size.
- Halving. If the table is a quarter full copy the elements to a new array of half the size
- Potential function.
  - $\Phi(D_i) = \begin{cases} 2n - L & \text{if T at least half full} \\ L/2 - n & \text{if T less than half full} \end{cases}$
  - L = current array size, n = number of elements in array.

- Inserting in full table and doubling

n = 9, L = 16



💰💰💰💰
💰💰💰💰

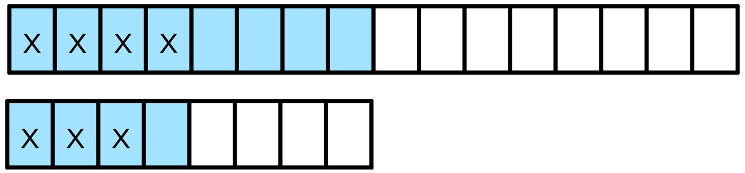
- amortized cost = 9 + - 💰💰💰💰💰💰 = 3

Dynamic tables

- Doubling. If the table is full (number of elements equal to size of array) copy the elements to a new array of double size.
- Halving. If the table is a quarter full copy the elements to a new array of half the size
- Potential function.
  - $\Phi(D_i) = \begin{cases} 2n - L & \text{if T at least half full} \\ L/2 - n & \text{if T less than half full} \end{cases}$
  - L = current array size, n = number of elements in array.

- Deleting in a quarter full table and halving

n = 3, L = 8



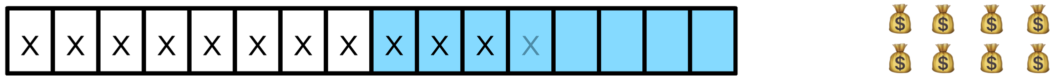
💰 💰
💰 💰

- amortized cost = 3 + - 💰 💰 💰 = 0

Dynamic tables

- Doubling. If the table is full (number of elements equal to size of array) copy the elements to a new array of double size.
- Halving. If the table is a quarter full copy the elements to a new array of half the size
- Potential function.
    - $\Phi(D_i) = \begin{cases} 2n - L & \text{if T at least half full} \\ L/2 - n & \text{if T less than half full} \end{cases}$
    - L = current array size, n = number of elements in array.

- Deleting when more than half full (still half full after): n = 11, L = 16



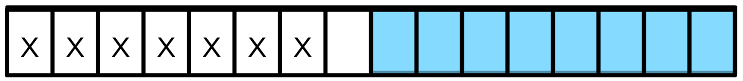
- amortized cost = 1 + - 💰💰 = -1

Dynamic tables

- Doubling. If the table is full (number of elements equal to size of array) copy the elements to a new array of double size.
- Halving. If the table is a quarter full copy the elements to a new array of half the size
- Potential function.
  - $\Phi(D_i) = \begin{cases} 2n - L & \text{if T at least half full} \\ L/2 - n & \text{if T less than half full} \end{cases}$
  - L = current array size, n = number of elements in array.

- Deleting when half full (not half full after):

n = 7, L = 16



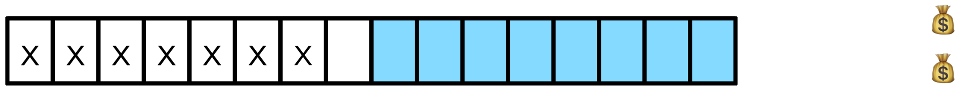
💰

- amortized cost = 1 + 💰 = 2

Dynamic tables

- Doubling. If the table is full (number of elements equal to size of array) copy the elements to a new array of double size.
- Halving. If the table is a quarter full copy the elements to a new array of half the size
- Potential function.
  - $\Phi(D_i) = \begin{cases} 2n - L & \text{if T at least half full} \\ L/2 - n & \text{if T less than half full} \end{cases}$
  - L = current array size, n = number of elements in array.
- Deleting in when less than half full (but still a quarter full after):

n = 7, L = 16



- amortized cost = 1 + 💰 = 2

Potential Method

- Summary:
  1. Pick a potential function, Φ, that will work (art).
  2. Use potential function to bound the amortized cost of the operations you're interested in.
  3. Bound $\Phi(D_0) - \Phi(D_{final})$
- Techniques to find potential functions: if the actual cost of an operation is high, then decrease in potential due to this operation must be large, to keep the amortized cost low.

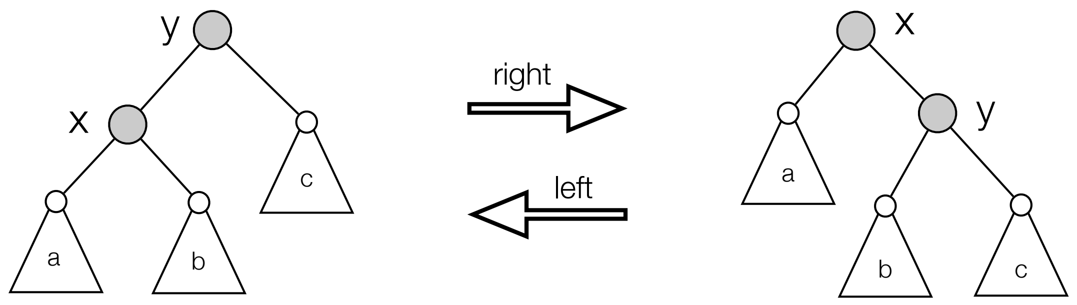
Splay Trees

Splay Trees

- Self-adjusting BST (Sleator-Tarjan 1983).
  - Most frequently accessed nodes are close to the root.
  - Tree reorganizes itself after each operation.
  - After access to a node it is moved to the root by splay operation.
  - Worst case time for insertion, deletion and search is O(n). Amortised time per operation O(log n).
- Operations. Search, predecessor, sucessor, max, min, insert, delete, join.

Splaying

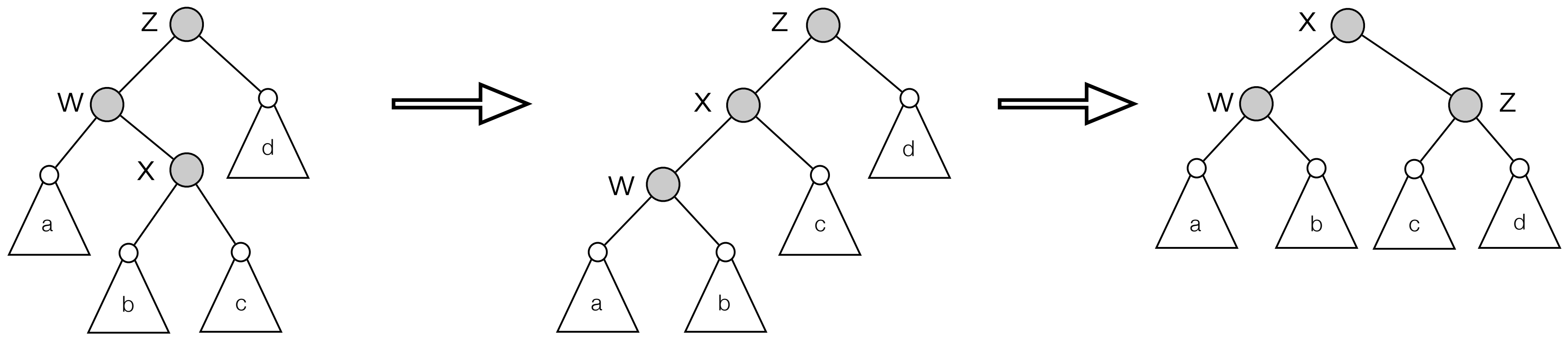
- Splay(x): do following rotations until x is the root. Let y be the parent of x.
  - right (or left): if x has no grandparent.



right rotation at x (and left rotation at y)

Splaying

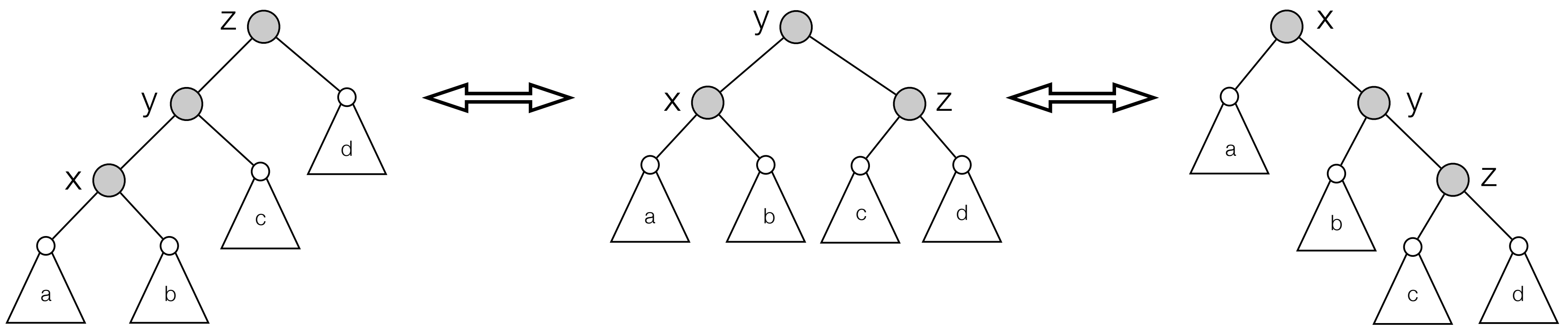
- Splay(x): do following rotations until x is the root. Let p(x) be the parent of x.
  - right (or left): if x has no grandparent.
  - zig-zag (or zag-zig): if one of x,p(x) is a left child and the other is a right child.



zig-zag at x

Splaying

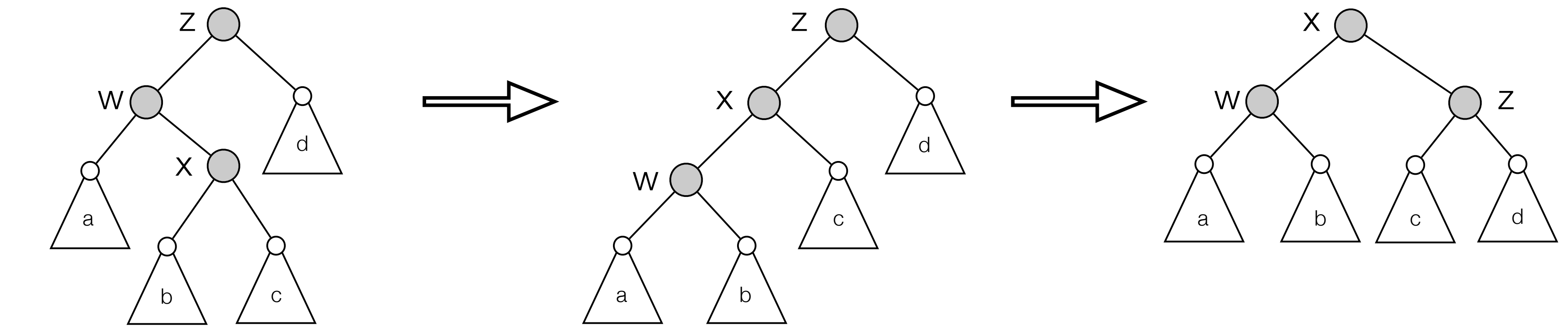
- Splay(x): do following rotations until x is the root. Let y be the parent of x.
  - right (or left): if x has no grandparent.
  - zig-zag (or zag-zig): if one of x,y is a left child and the other is a right child.
  - roller-coaster: if x and p(x) are either both left children or both right children.



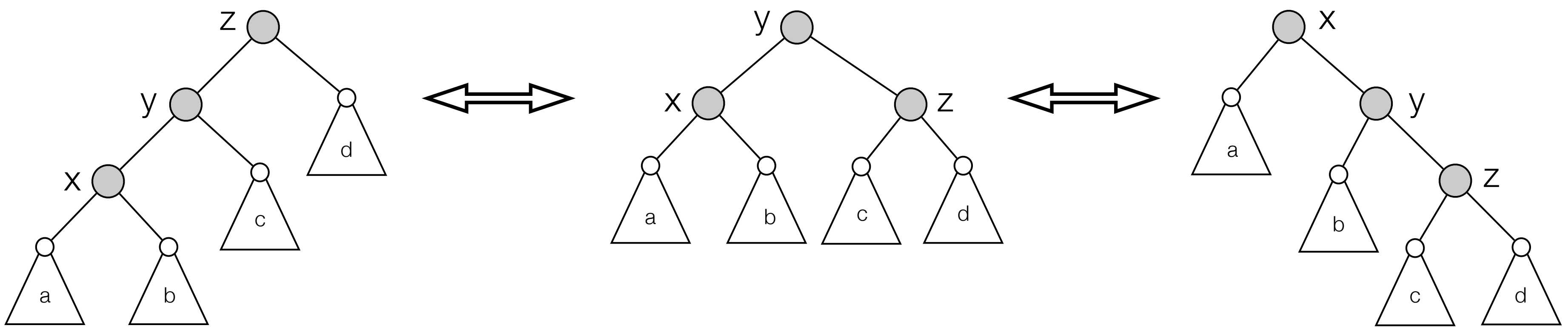
right roller-coaster at x (and left roller-coaster at z)

Splaying

zig-zag at x

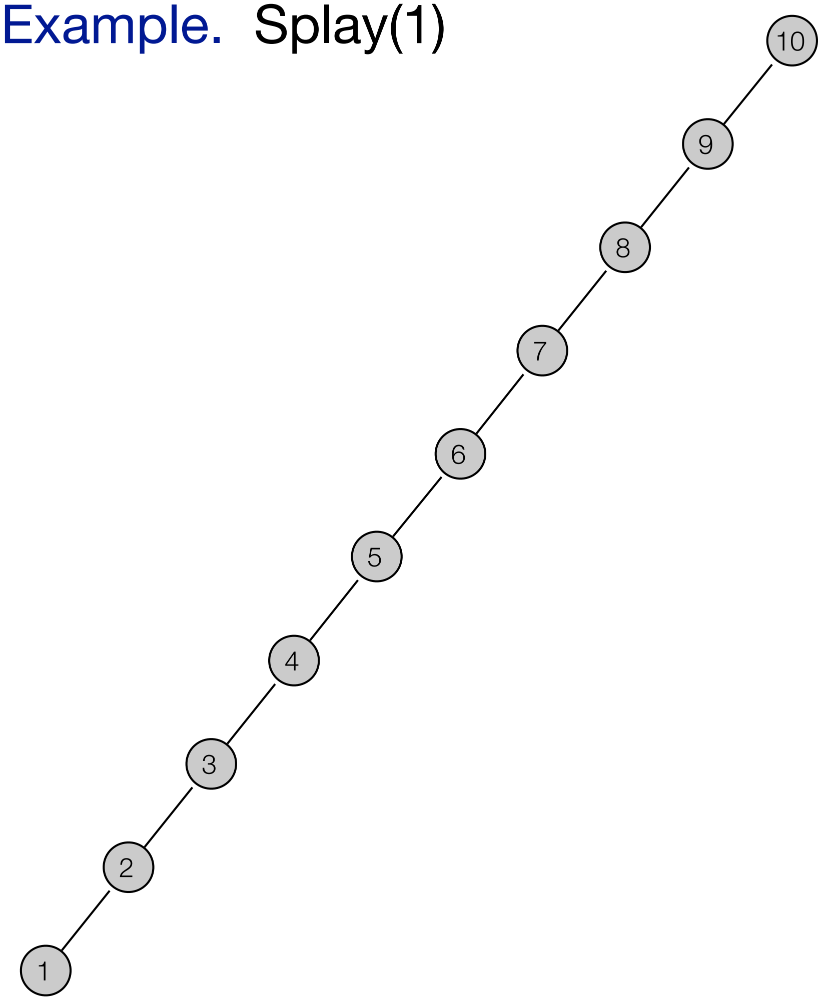


right roller-coaster at x (and left roller-coaster at z)



Splay

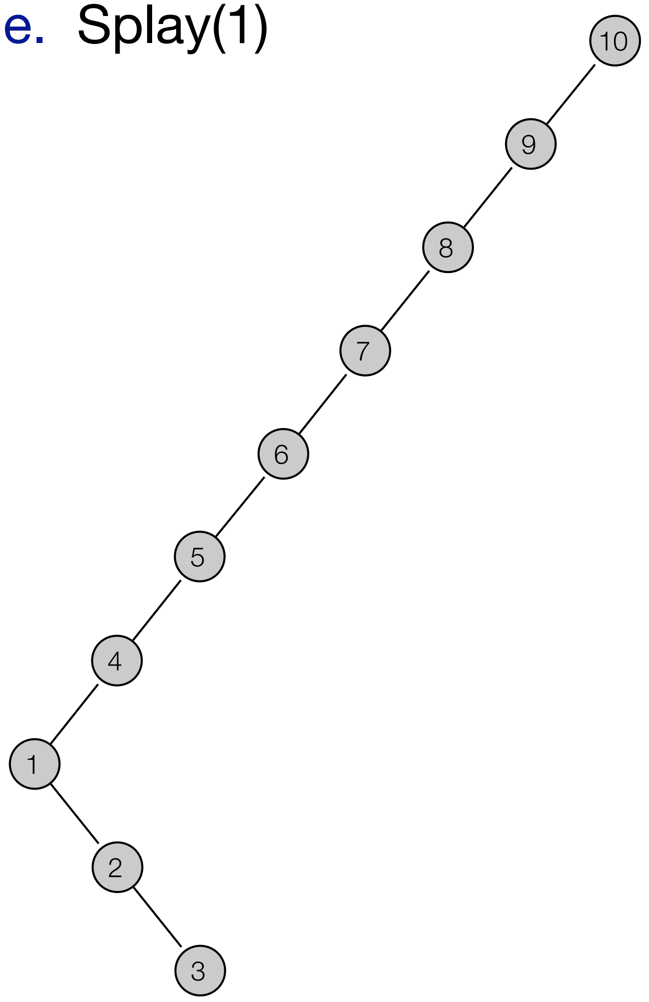
- **Example.** Splay(1)



right roller-coaster at 1

Splay

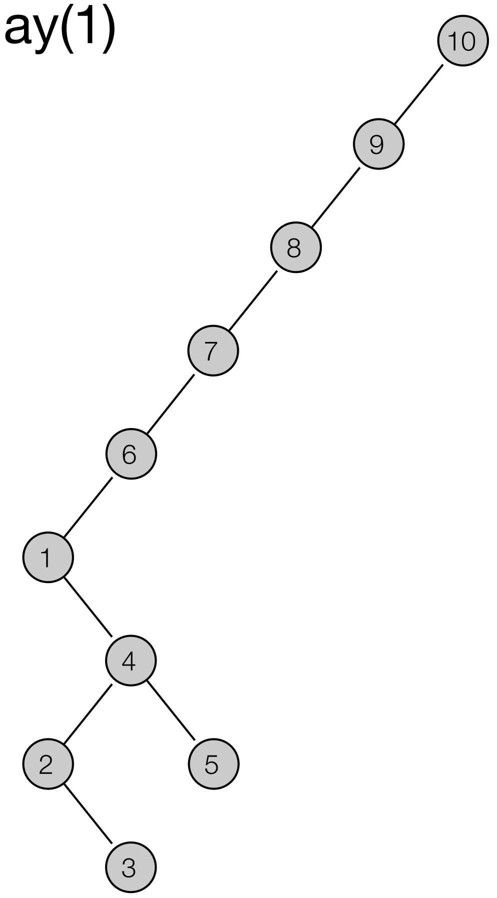
- Example. Splay(1)



right roller-coaster at 1

Splay

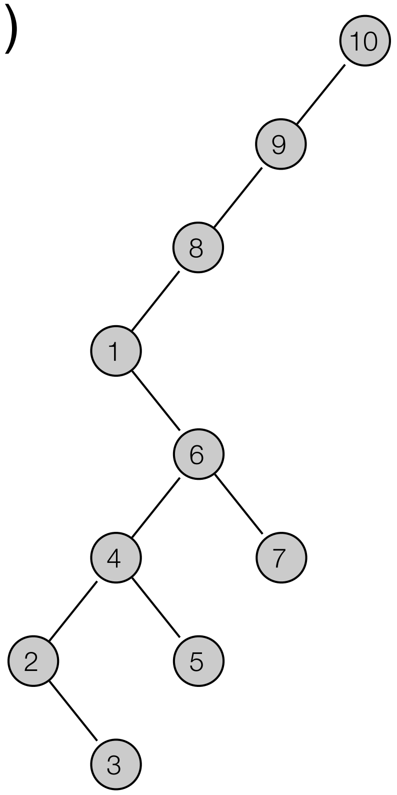
- Example. Splay(1)



right roller-coaster at 1

Splay

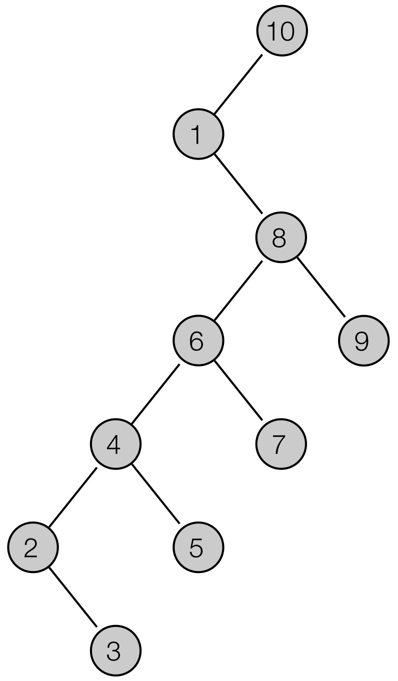
- Example. Splay(1)



right roller-coaster at 1

Splay

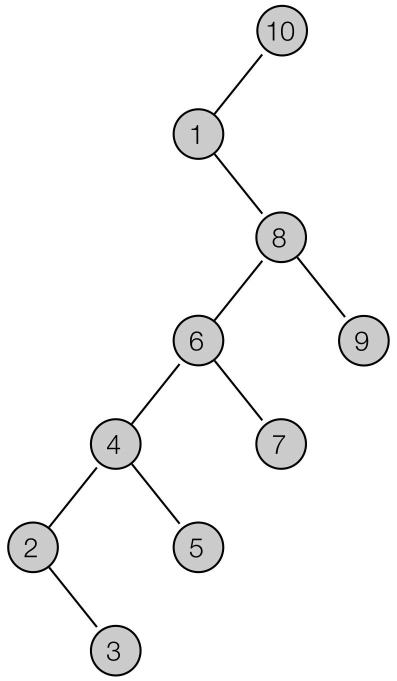
- **Example.** Splay(1)



right roller-coaster at 1

Splay

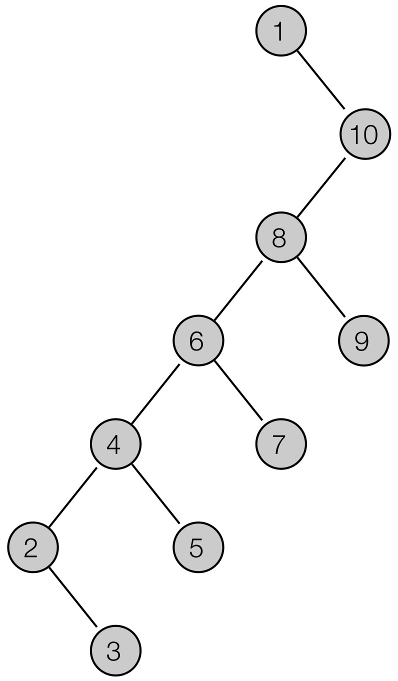
- Example. Splay(1)



right rotation at 1

Splay

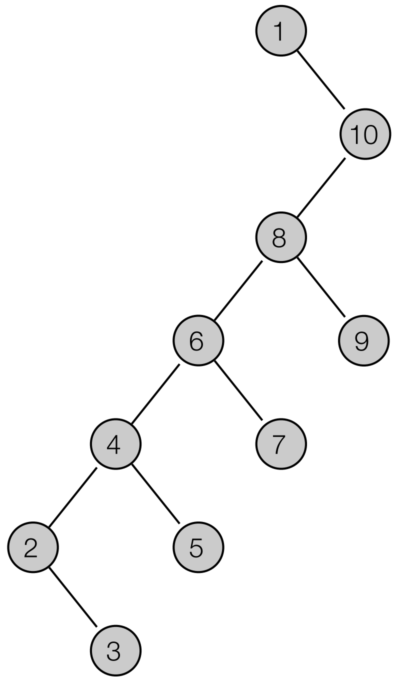
- Example. Splay(1)



right rotation at 1

Splay

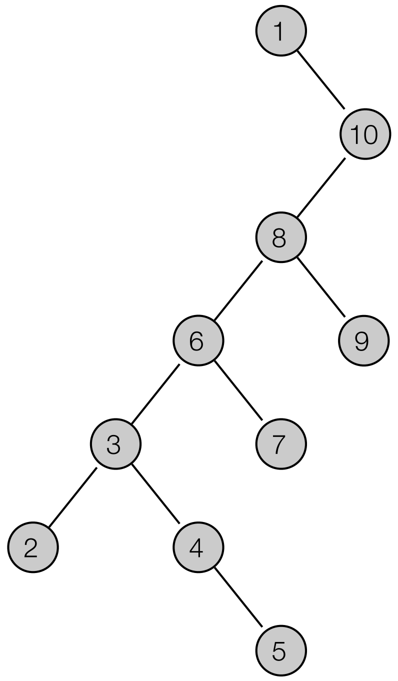
- Example. Splay(3)



zig-zag at 3

Splay

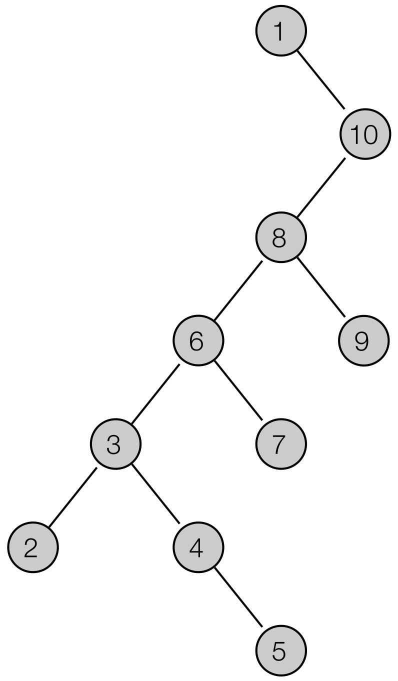
- **Example.** Splay(3)



zig-zag at 3

Splay

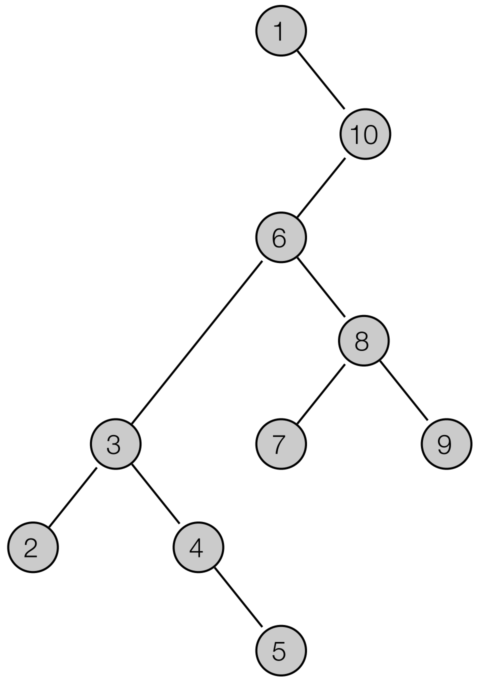
- Example. Splay(3)



roller-coaster at 3

Splay

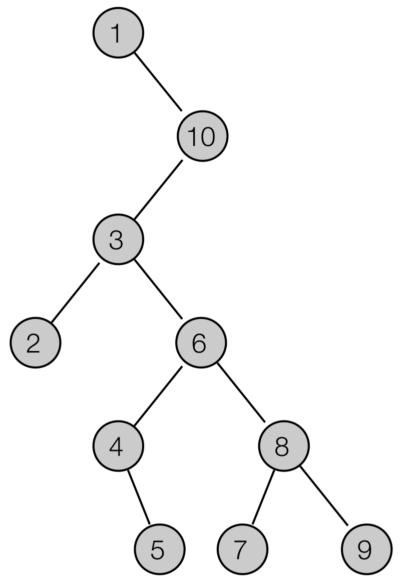
- Example. Splay(3)



roller-coaster at 3

Splay

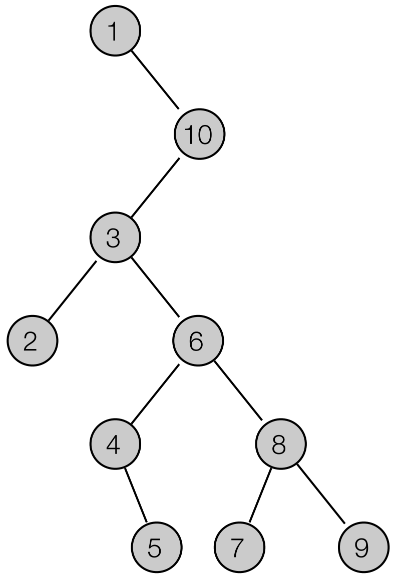
- Example. Splay(3)



roller-coaster at 3

Splay

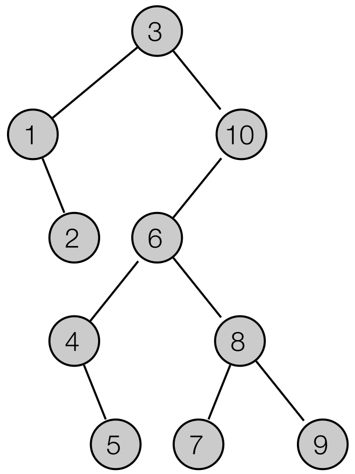
- Example. Splay(3)



zag-zig at 3

Splay

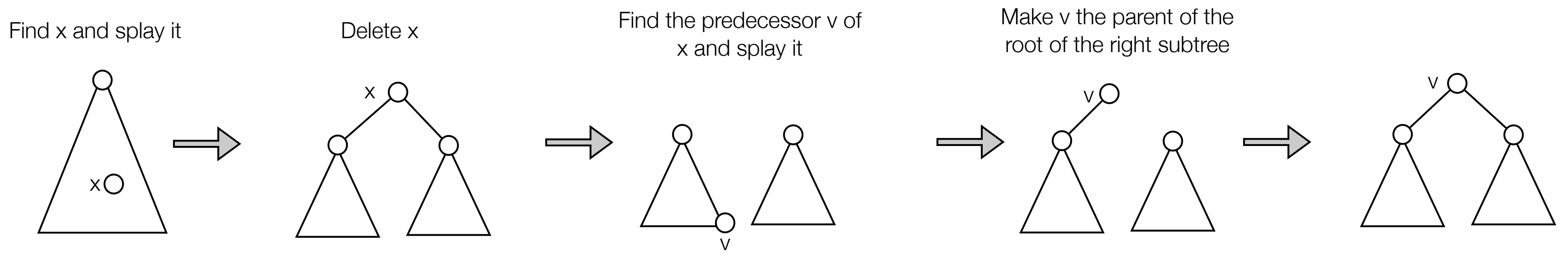
- Example. Splay(3)



zag-zig at 3

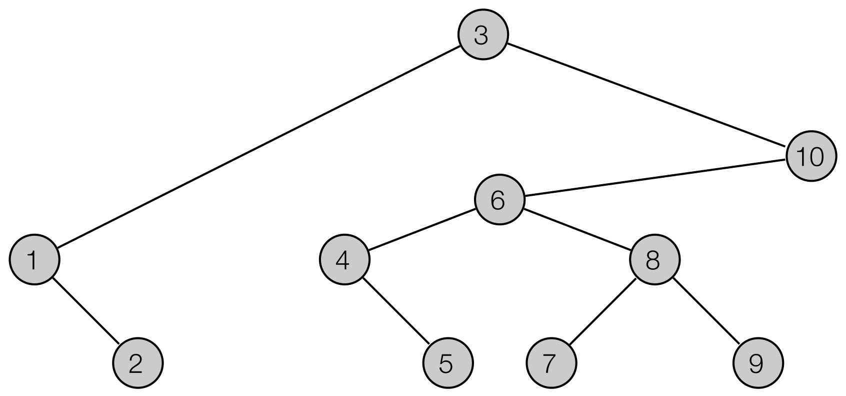
Splay Trees

- Search(x). Find node containing key x (or predecessor/successor) using usual search algorithm. Splay found node.
- Insert(x). Insert node containing key x using algorithm for binary search trees. Splay inserted node.
- Delete(x). Find node x, splay it and delete it. Tree now divided in two subtrees. Find node with largest key in left subtree, splay it and join it to the right subtree by making it the new root.


Find x and splay it
x
Delete x
x
Find the predecessor v of x and splay it
v
Make v the parent of the root of the right subtree
v
v

Deletion in Splay Trees

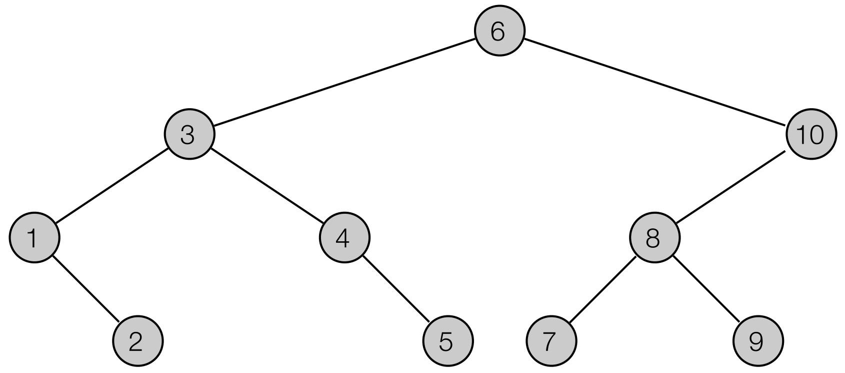
- Delete 6.



splay 6: zag-zig at 6

Deletion in Splay Trees

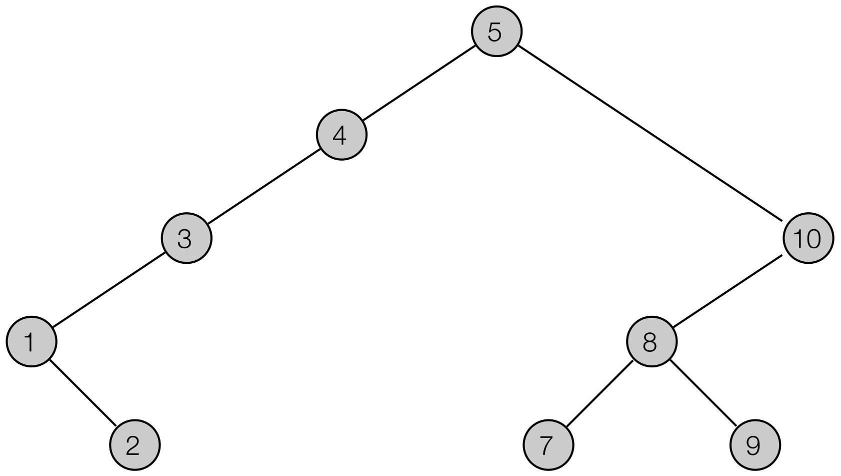
- Delete 6.



delete 6
splay 5

Deletion in Splay Trees

- Delete 6.



connect

Analysis of splay trees

- Amortized cost of a search, insert, or delete operation is O(log n).
- All costs bounded by splay.

Analysis of splay trees

- Rank of a node.
  - size(v) = #nodes in subtree of v
  - $\mathrm{rank}(v) = \lfloor \lg \mathrm{size}(v) \rfloor$
- Potential function.

$$\Phi = \sum_v \mathrm{rank}(v) = \sum_v \lfloor \lg \mathrm{size}(v) \rfloor$$

- Rotation Lemma. The amortized cost of a single rotation at any node v is at most 1 + 3 rank’(v) - 3 rank(v), and the amortized cost of a double rotation at any node v is at most 3 rank’(v) - 3 rank(v).
- Splay Lemma. The amortized cost of a splay(v) is at most 1 + 3rank’(v) - 3 rank(v).

Splay Lemma Proof

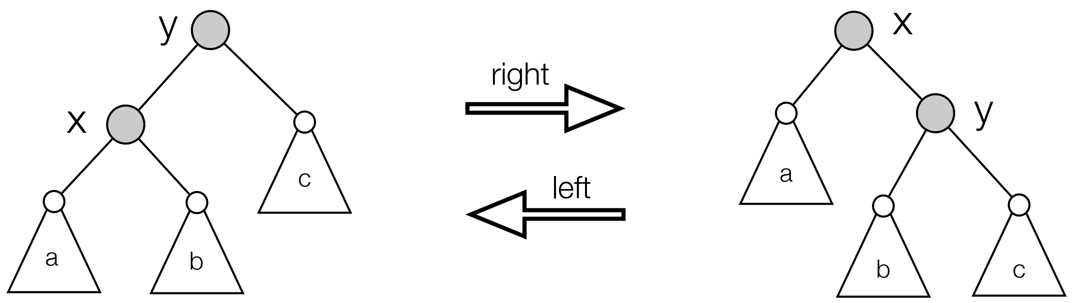
- Rotation Lemma. The amortized cost of a single rotation at any node v is at most 1 + 3 rank’(v) - 3 rank(v), and the amortized cost of a double rotation at any node v is at most 3 rank’(v) - 3 rank(v).
- Splay Lemma. The amortized cost of a splay(v) is at most 1 + 3rank’(v) - 3 rank(v).
- Proof.
  - Assume we have k rotations.
  - Only last one can be a single rotation.

$$\sum_{i=0}^{k} \hat{c}_i \leq \sum_{i=1}^{k-1} \left(r_i(v) - r_{i-1}(v)\right) + (1 + r_k(v) - r_{k-1}(v)) = 1 + r_k(v) - r_0(v)m = O(\lg n)$$

where $r_i(v)$ is the rank of v after the ith rotation.

Rotation Lemma

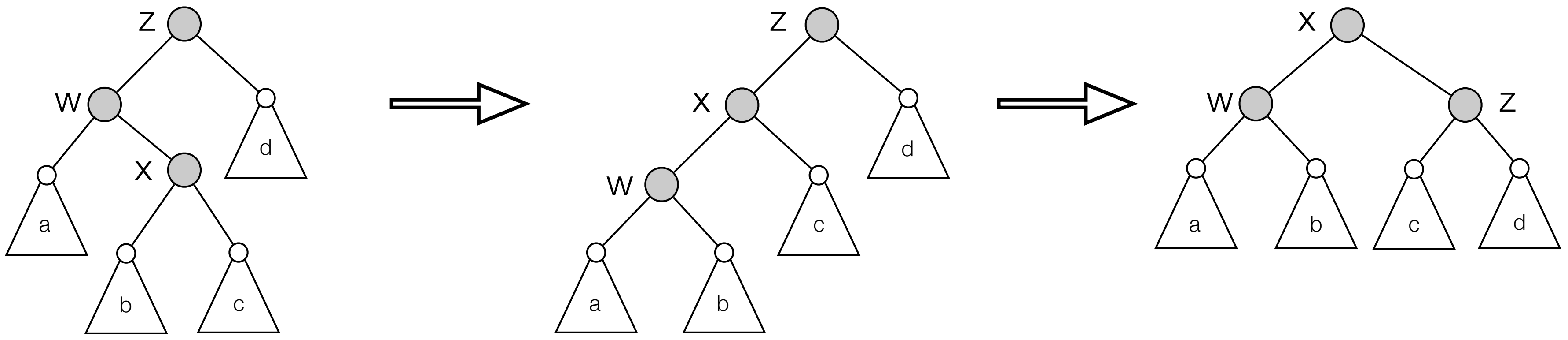
- Proof of rotation lemma: Single rotation.
  - Actual cost: 1
  - Change in potential:
    - Only x and y can change rank.
    - Change in potential at most r'(x) - r(x).
  - Amortized cost ≤ 1 + r'(x) - r(x) ≤ 1 + 3r'(x) - 3r(x).



right rotation at x (and left rotation at y)

Rotation Lemma

- Proof of rotation lemma: zig-zag.
  - Actual cost: 2
  - Change in potential:
    - Only x, w and z can change rank.
    - Change in potential at most 2r'(x) - 2r(x) - 2.
  - Amortized cost: ≤ 2 + 2r'(x) - 2r(x) - 2 ≤ 2r'(x) - 2r(x) ≤ 3r'(x) - 3r(x).



zig-zag at x