Lecture 5: Self-Adjusting Networks

Prof. Chen Avin (BGU, Israel)
Prof. Stefan Schmid (TU Berlin, Germany)

Learning Goals

After this lecture, you will understand:

⇢ How binary search trees can become self-adjusting

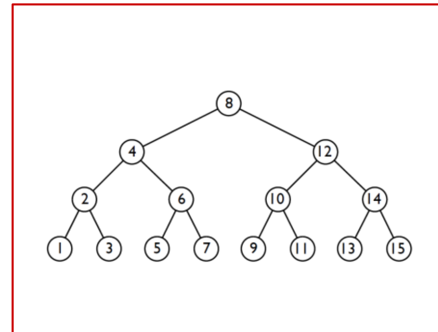
⇢ A model for self-adjusting networks and desirable properties

⇢ SplayNets, a self-adjusting networks

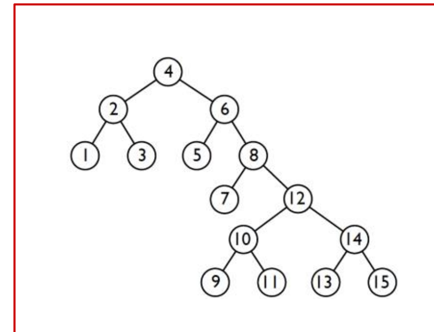
⇢ How ego-trees can be used to make networks self-adjusting

Recall

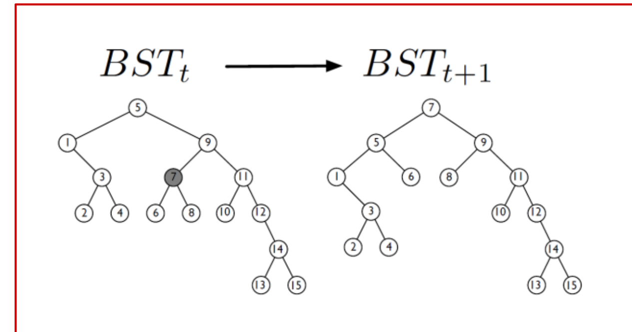
Traditional BST
(Worst-case coding)



Demand-aware BST
(Huffman coding)



Self-adjusting BST
(Dynamic Huffman coding)



Exploit more structure, from spatial to temporal: reduce **access cost**

⇢ Example of self-adjusting BST: splay tree

Self-Adjusting BSTs

Typical model:

⇢ Minimize service and adjustment cost: Cost=srv+adj

$$\text{Cost} = \sum_{i=1} srv(\sigma_i, \text{BST}_i) + \sum_{i=1} adj(\text{BST}_i, \text{BST}_{i+1})$$

⇢ srv: number of links traversed for lookup

⇢ adj: number of links adjusted

⇢ Requests arrive online, do not know future demand

In general, I need:

⇢ Operations to pull frequently accessed nodes closer to root

⇢ … while ensuring search property at any time

⇢ … and while keeping other frequent nodes close as well.

Self-Adjusting BSTs

Typical model:

⇢ Minimize service and adjustment cost: Cost=srv+adj

$$\text{Cost} = \sum_{i=1} srv(\sigma_i, \text{BST}_i) + \sum_{i=1} adj(\text{BST}_i, \text{BST}_{i+1})$$

⇢ srv: number of links traversed for lookup

⇢ adj: number of links adjusted

⇢ Requests arrive online, do not know future demand

In general, I need:

⇢ Operations to pull frequently accessed nodes closer to root

⇢ … while ensuring search property at any time

⇢ … and while keeping other frequent nodes close as well.

Formal?

Working Set Property

Working Set Property:

⇢ For any access j, let $t(j)$ be the number of different items accessed before access j since the last access of item i

⇢ Distance of j to root should be proportional to $t(j)$

Typically interested in ***amortized cost*** (or time):

⇢ Cost per operation averaged over a worst-case sequence: $\max_{\sigma} \sum_{i=1}^{n} \mathrm{srv}(\sigma_i)/|\sigma|$

⇢ Typically analyzed with potential functions

For example, for splay trees:

⇢ Amortized cost of access j is $O(\log(t(j) + 1))$

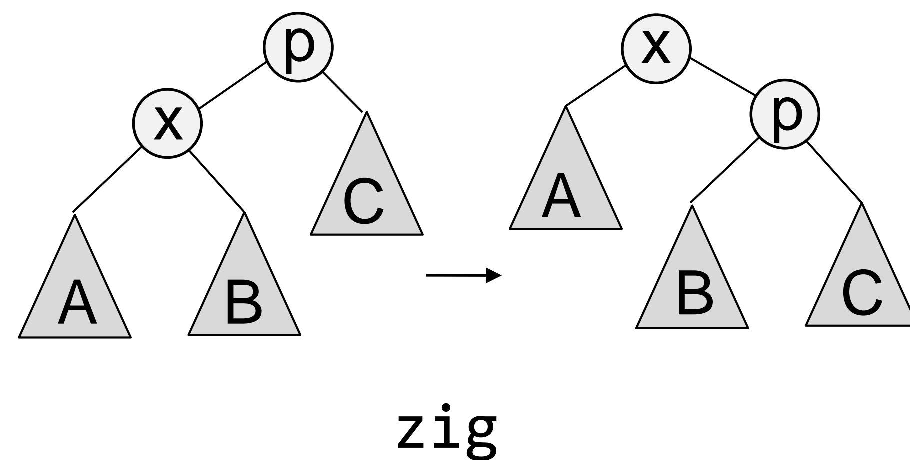
Splay Trees

Simple rule:

⇢ Upon access, promote node immediately to root

⇢ Similar to well-known “move-to-front” algorithm for lists

Based on iterative application of 3 operations:



zig

⇀ Rotate right

⇀ Moves x closer to root

⇀ Preserves search structure

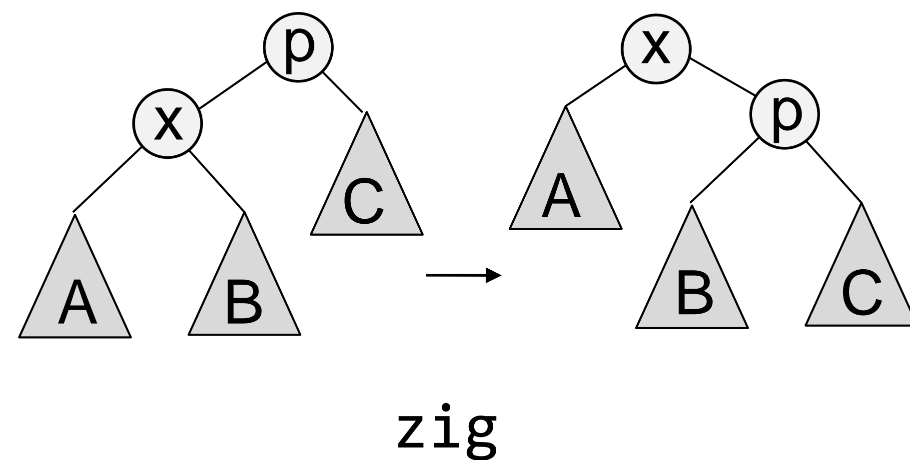
Splay Trees

Simple rule:

⇢ Upon access, promote node immediately to root

⇢ Similar to well-known “move-to-front” algorithm for lists

Based on iterative application of 3 operations:



zig

⇀ Rotate right

⇀ Moves x closer to root

⇀ Preserves search structure

Also in other direction: **zag**

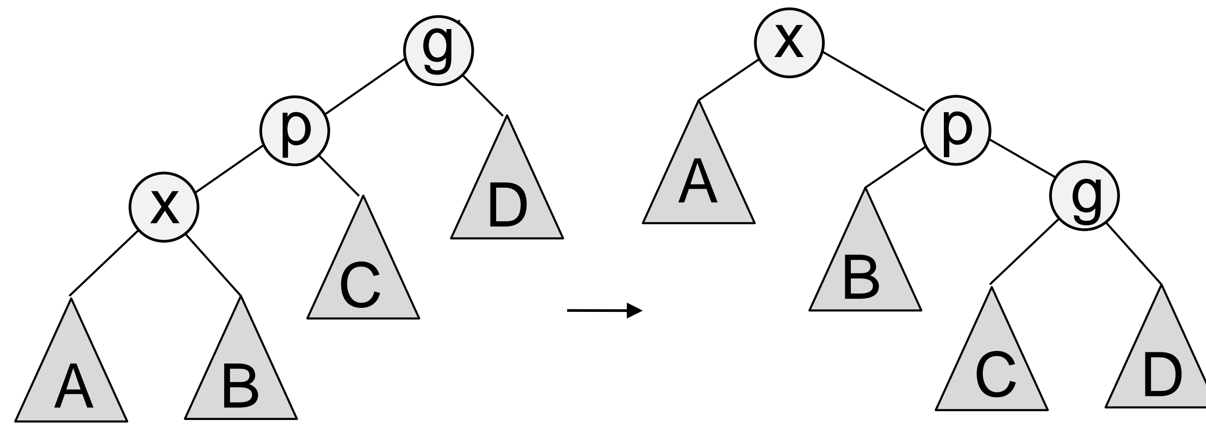
Splay Trees

Simple rule:

⇢ Upon access, promote node immediately to root

⇢ Similar to well-known “move-to-front” algorithm for lists

Based on iterative application of 3 operations:



zigzig

⇀ Rotate right twice

⇀ Moves x closer to root

⇀ Preserves search structure

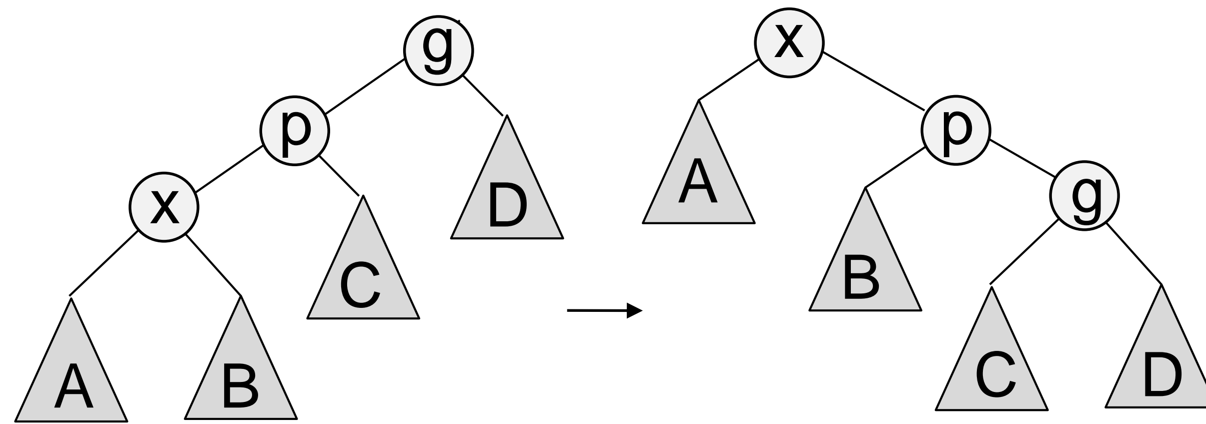
Splay Trees

Simple rule:

⇢ Upon access, promote node immediately to root

⇢ Similar to well-known “move-to-front” algorithm for lists

Based on iterative application of 3 operations:



zigzig

⇀ Rotate right twice
⇀ Moves x closer to root
⇀ Preserves search structure

Also in other direction: **zagzag**

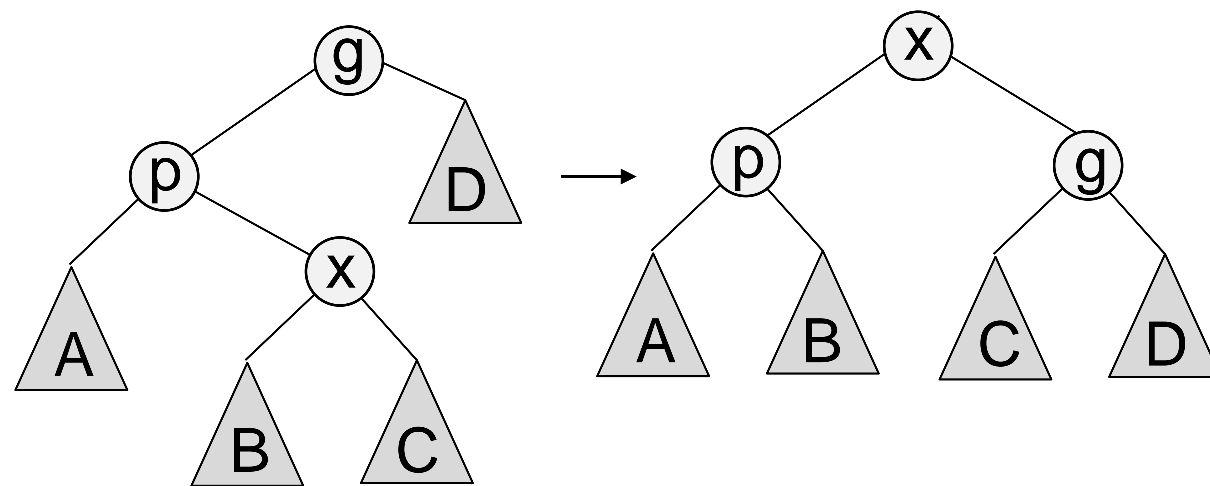
Splay Trees

Simple rule:

⇢ Upon access, promote node immediately to root

⇢ Similar to well-known “move-to-front” algorithm for lists

Based on iterative application of 3 operations:



zigzag

⇀ Rotate left-right

⇀ Moves x closer to root

⇀ Preserves search structure

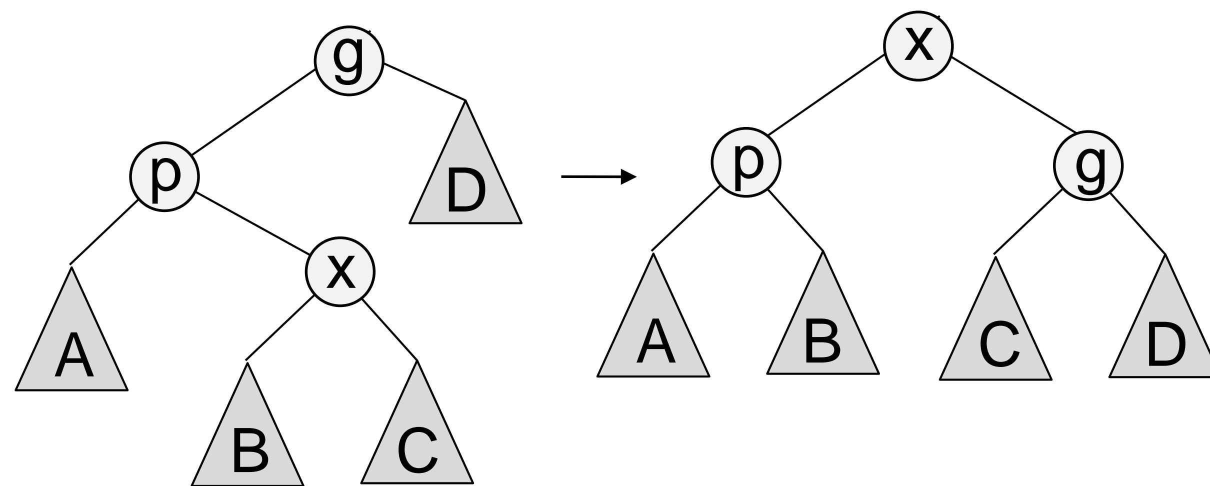
Splay Trees

Simple rule:

⇢ Upon access, promote node immediately to root

⇢ Similar to well-known “move-to-front” algorithm for lists

Based on iterative application of 3 operations:



zigzag

⇀ Rotate left-right

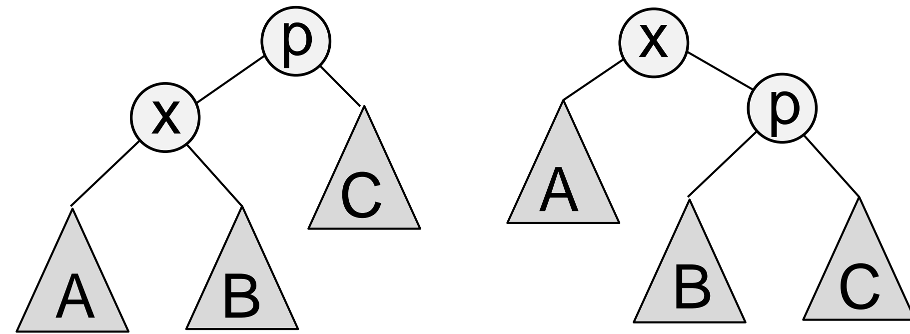
⇀ Moves x closer to root

⇀ Preserves search structure

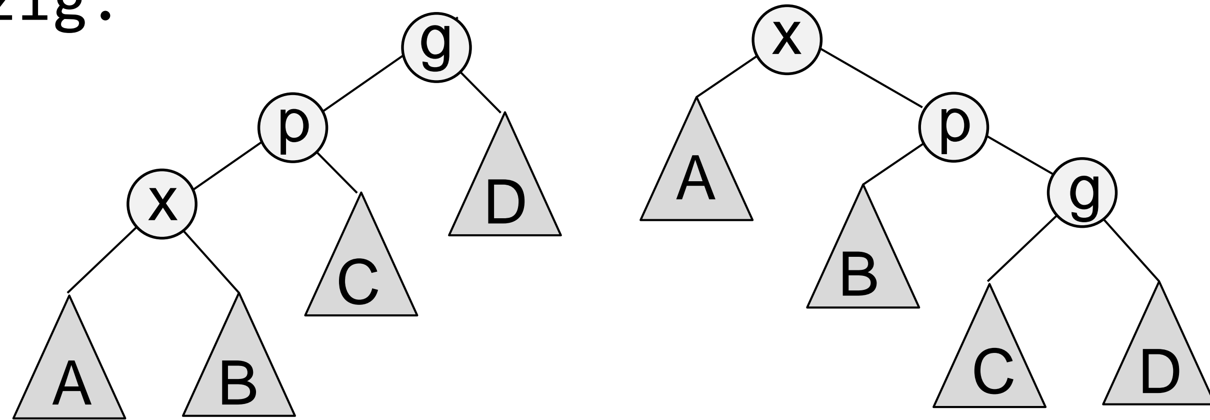
Also in other direction: **zagzig**

Splay Trees Update

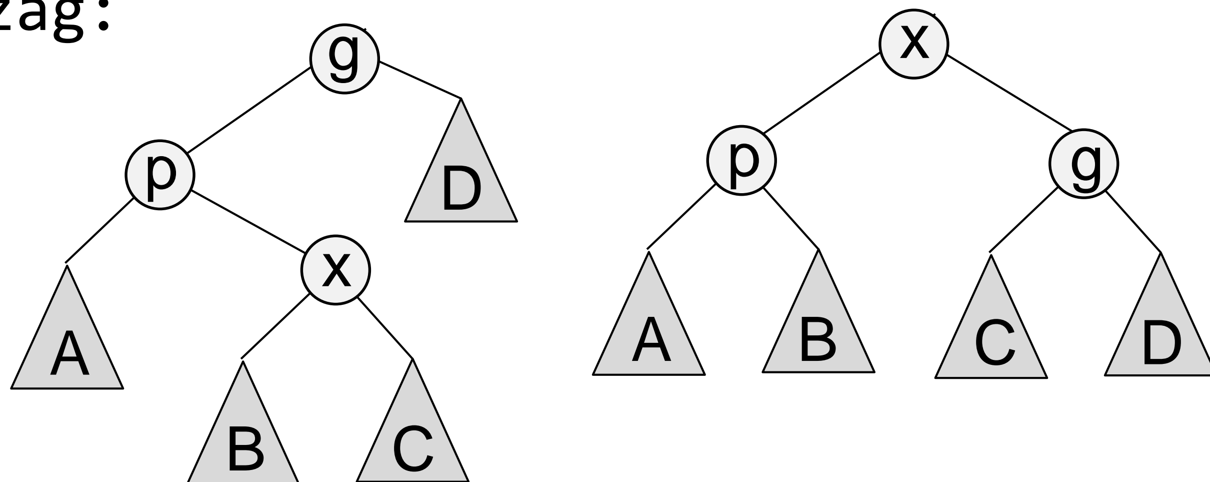
zig:



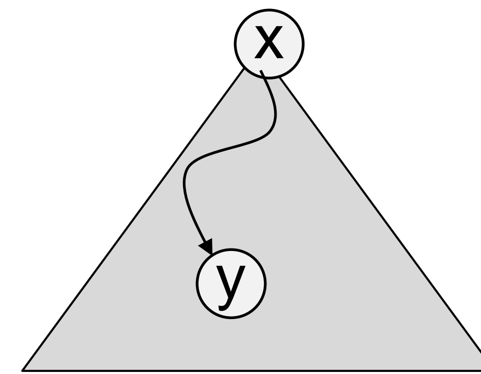
zigzig:

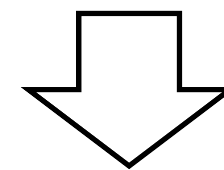


zigzag:

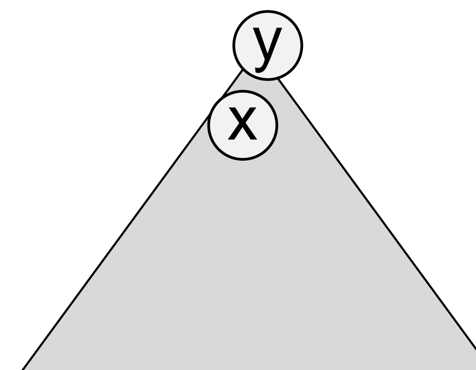


1. Access



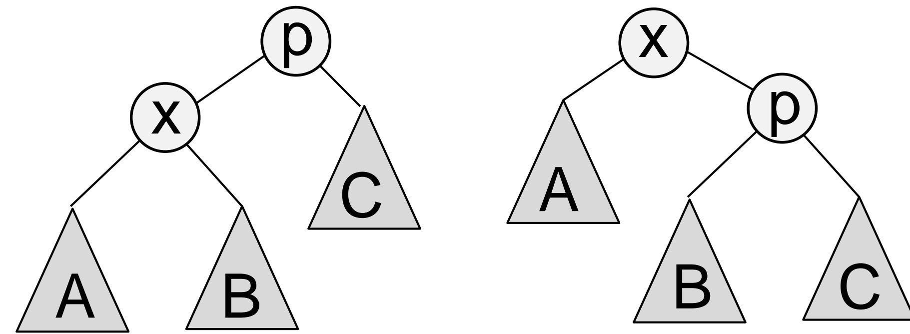


2. Move to root: splay

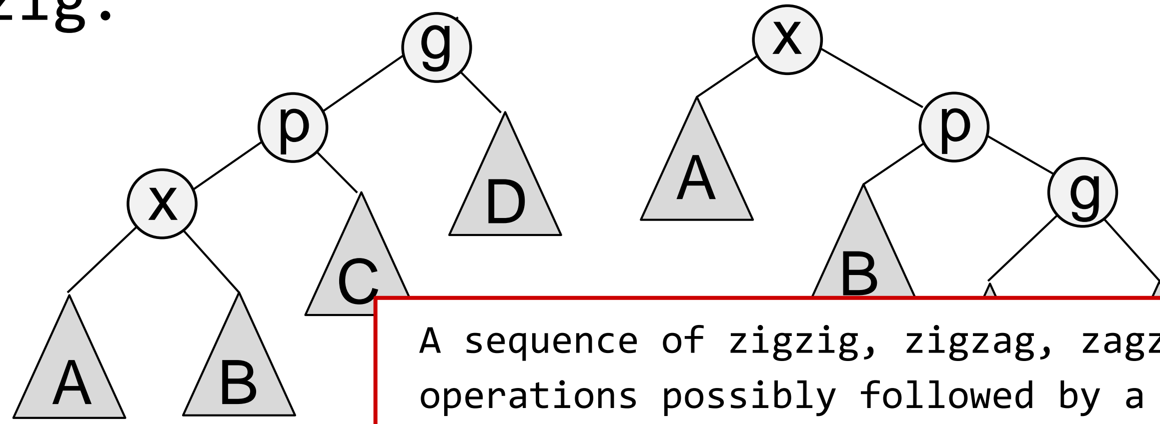


Splay Trees Update

zig:

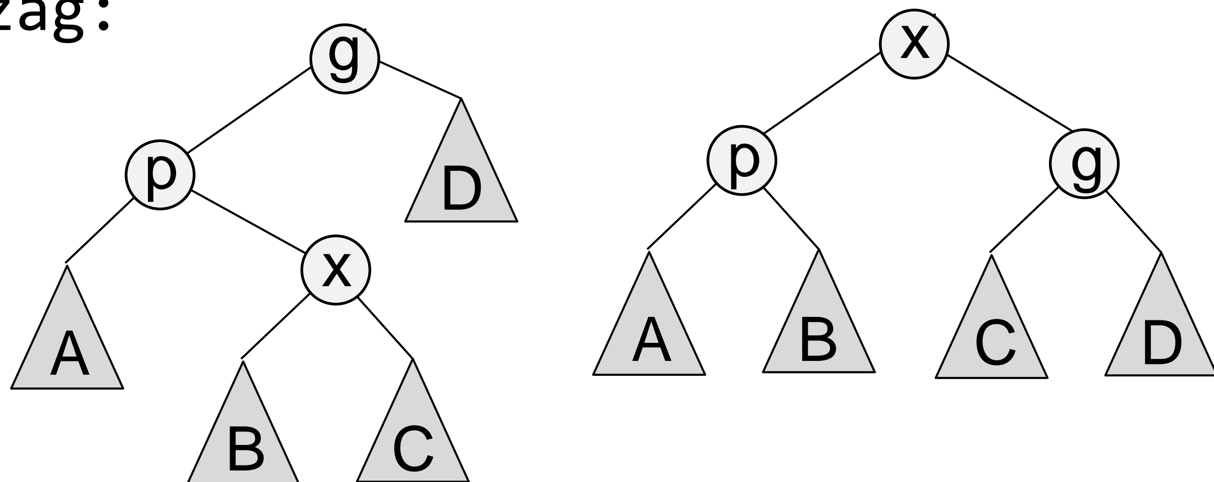


zigzig:

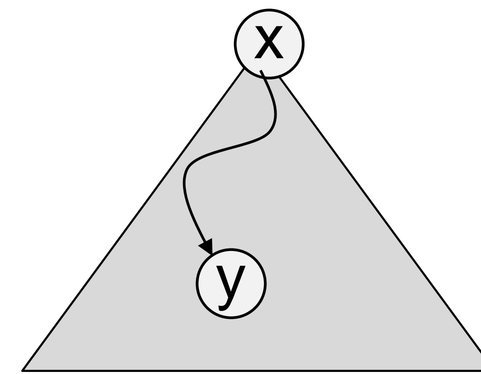


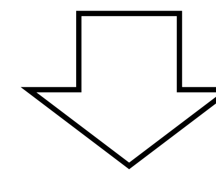
A sequence of zigzig, zigzag, zagzag, zagzig operations possibly followed by a zig or a zag, moving the accessed item to the root.

zigzag:

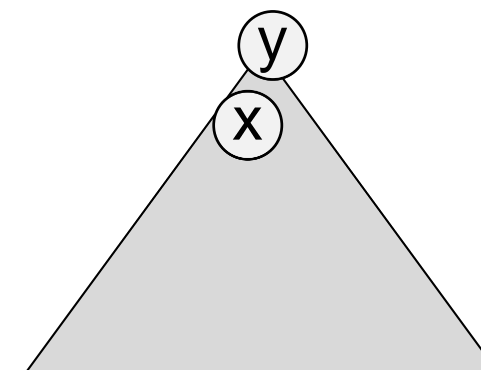


1. Access





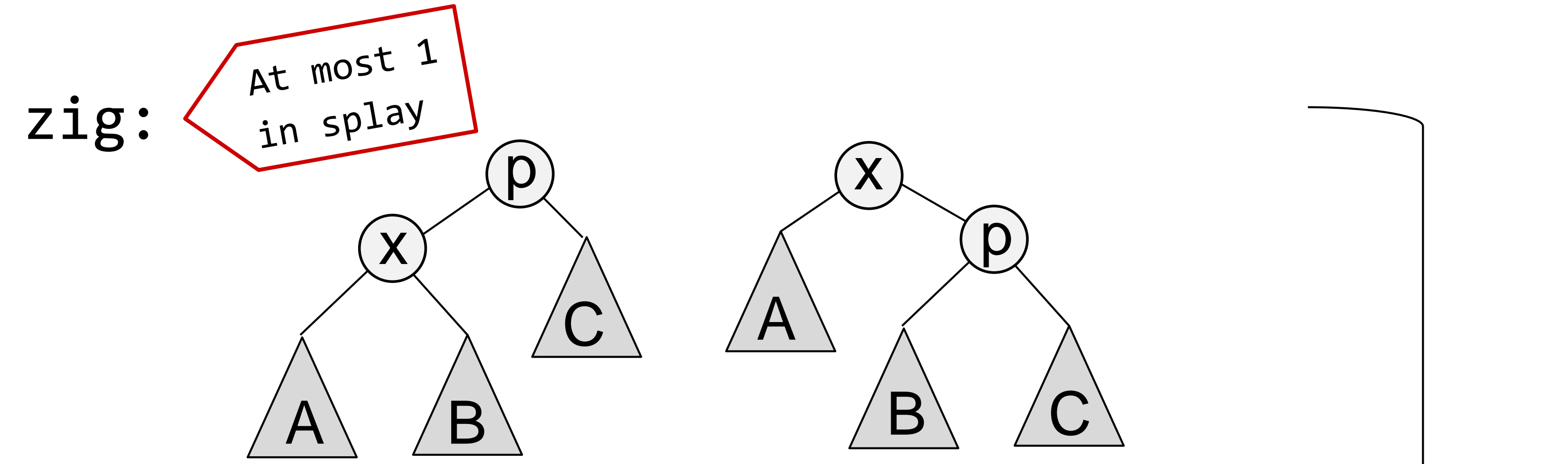
2. Move to root: splay



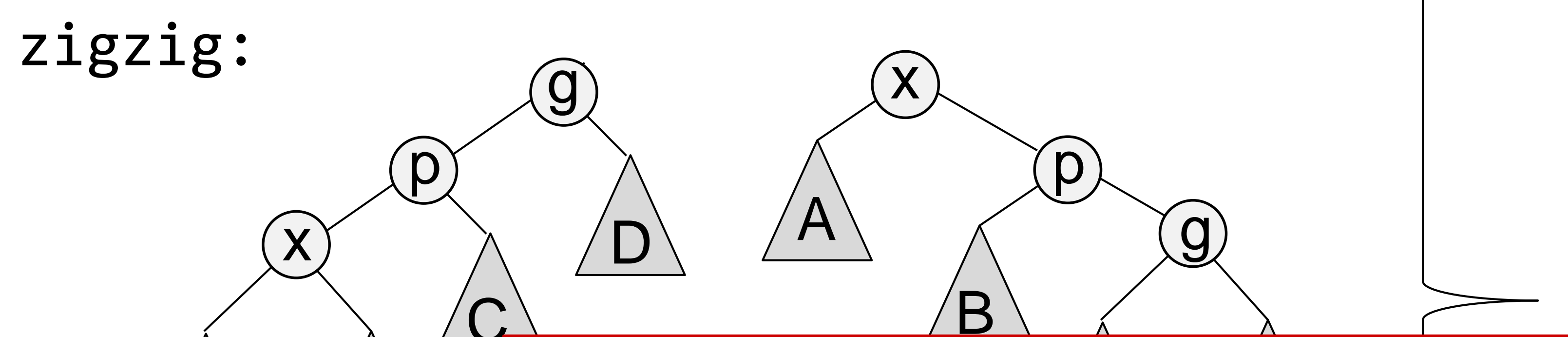
Splay Trees Update

zig:

At most 1 in splay

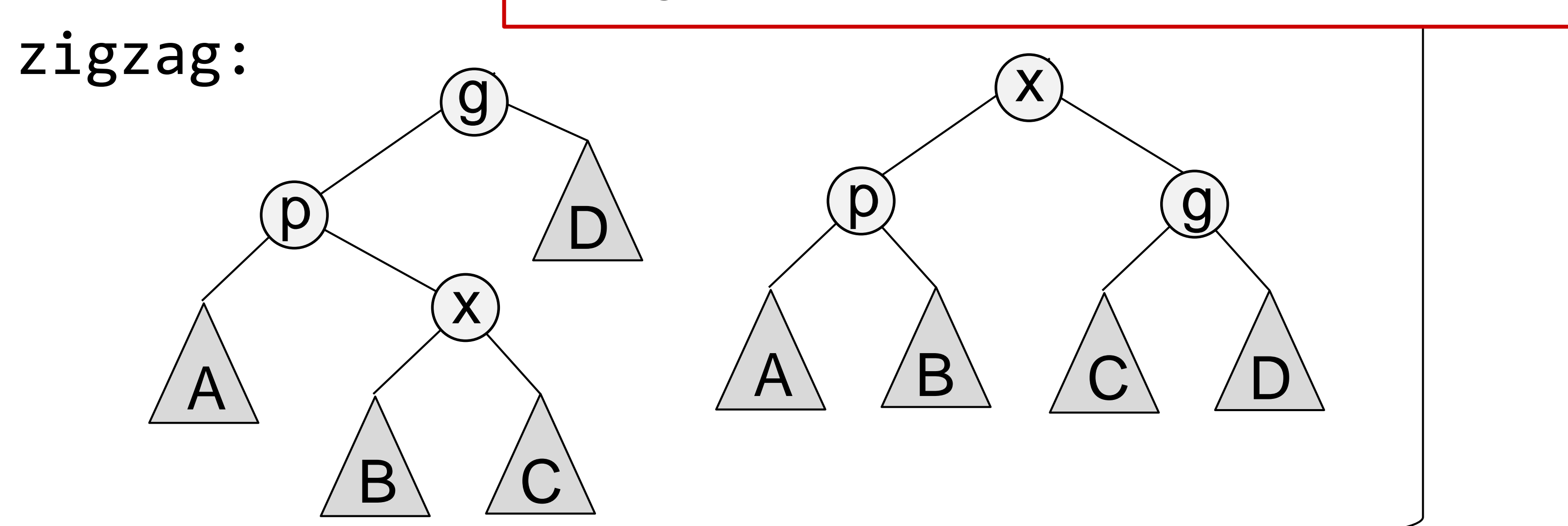


zigzig:

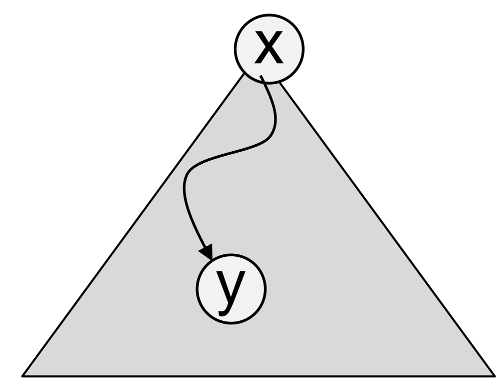


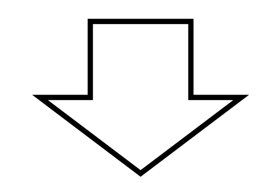
A sequence of zigzig, zigzag, zagzag, zagzig operations possibly followed by a zig or a zag, moving the accessed item to the root.

zigzag:

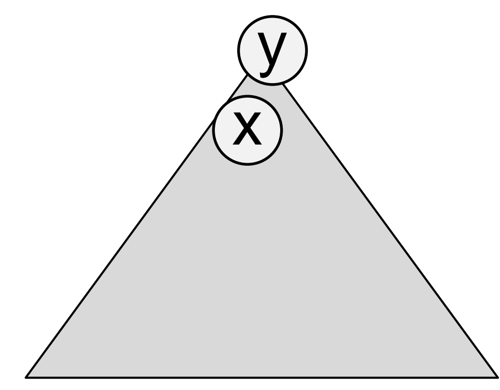


1. Access



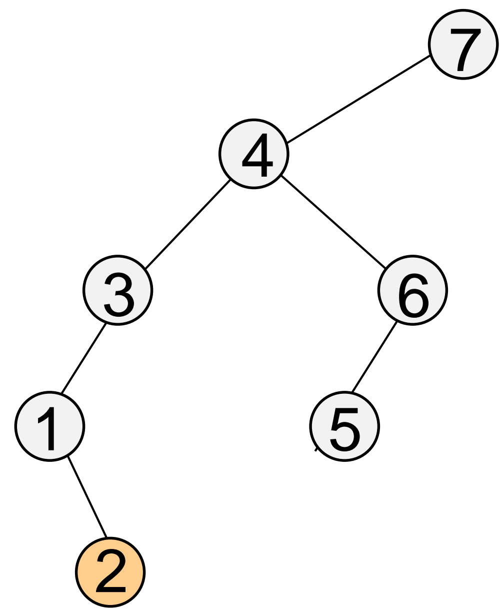


2. Move to root: splay



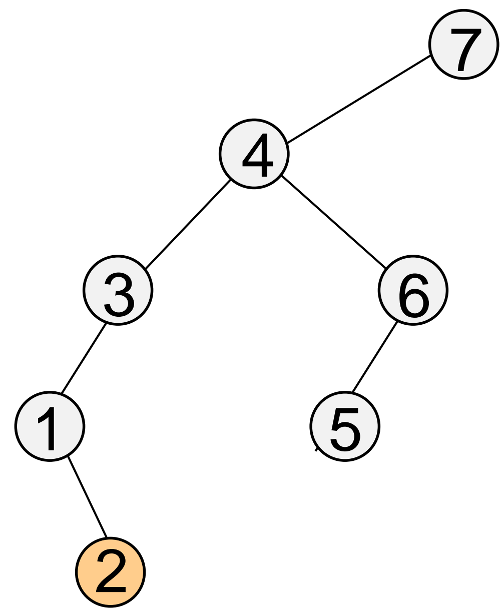
Working Set: Example

⇢ σ = 2



Working Set: Example

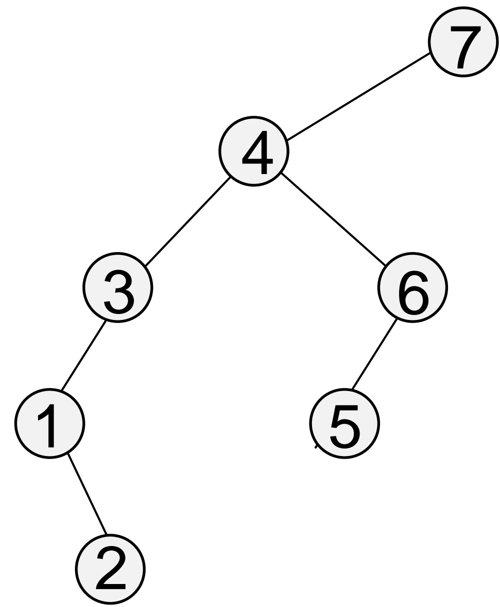
- $\sigma = 2$



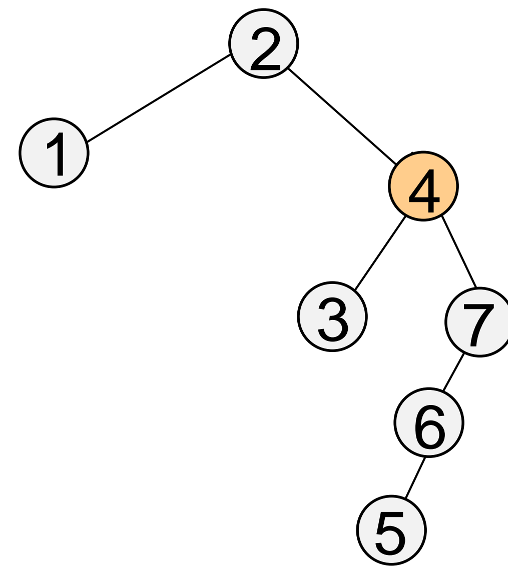
Cost 4

Working Set: Example

⇢ $\sigma = 2,4$



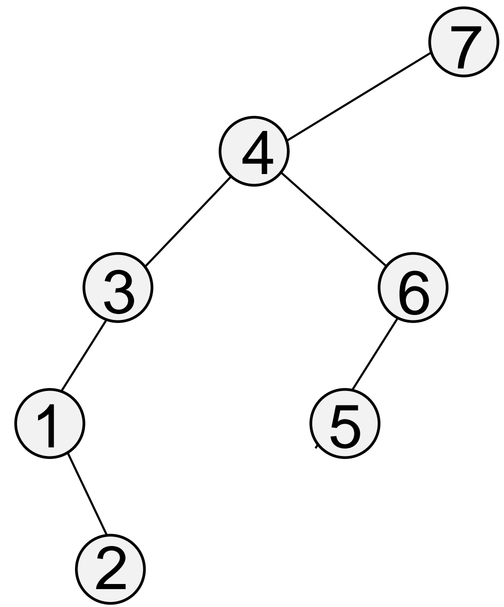
Cost 4



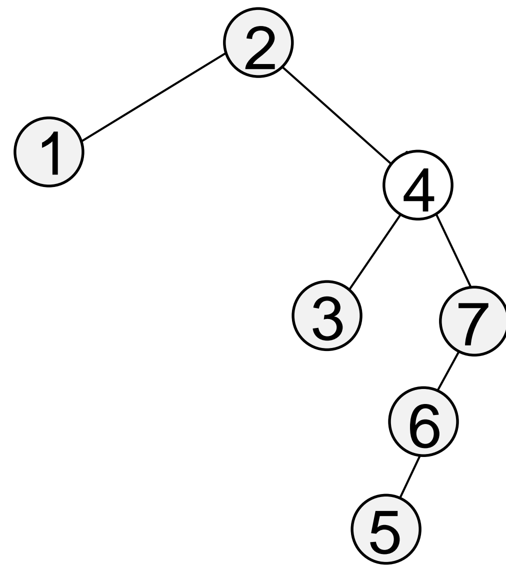
Cost 1

Working Set: Example

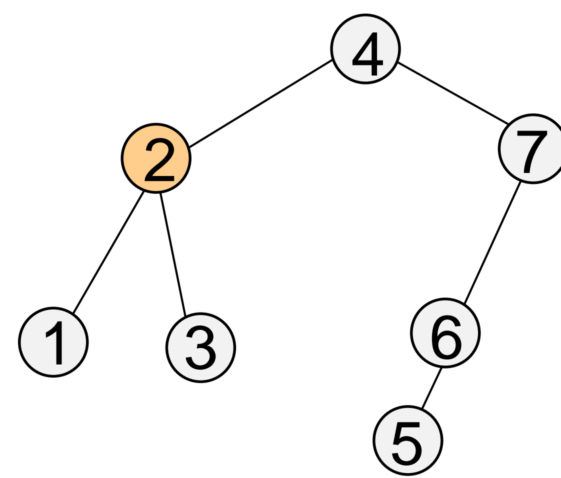
- σ = 2,4,2



Cost 4



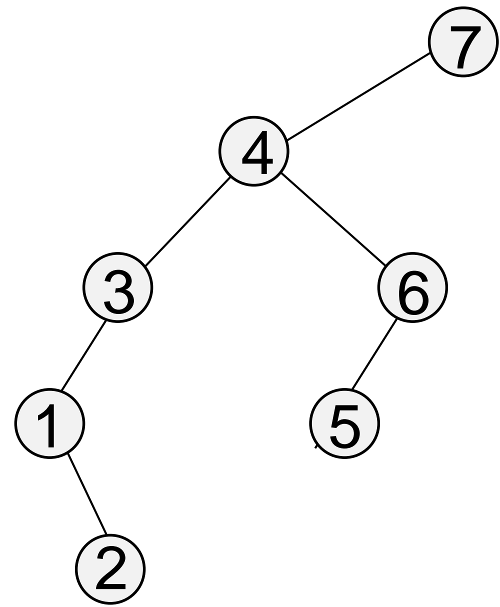
Cost 1

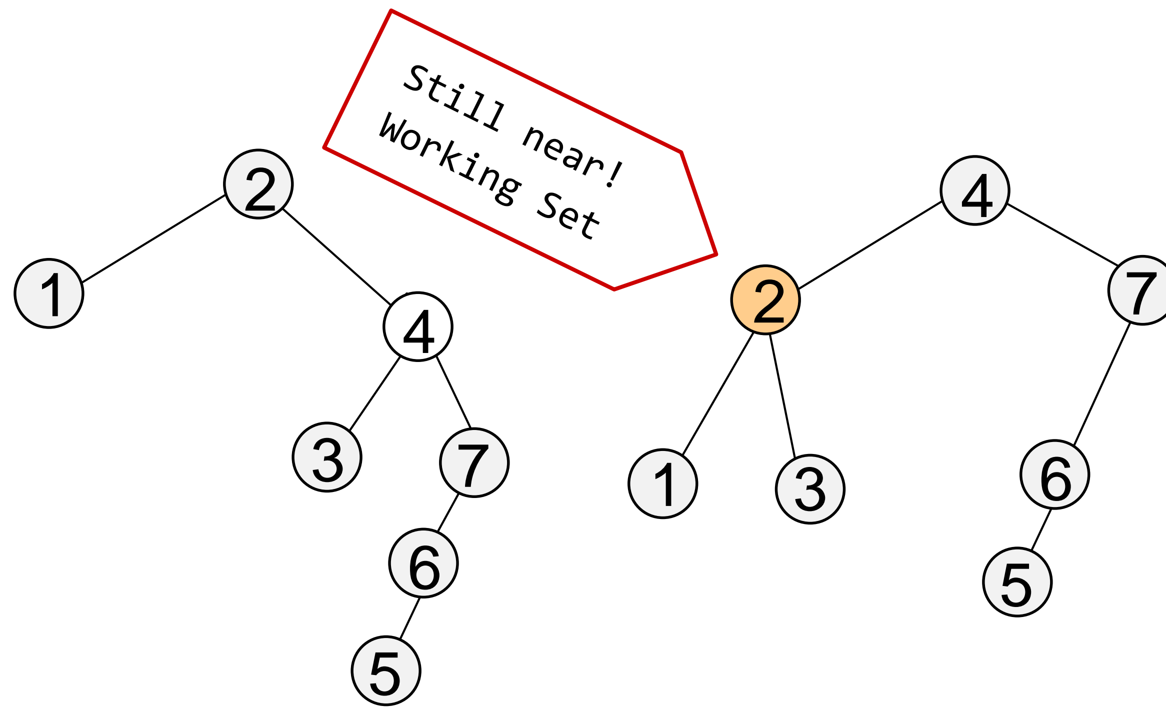


Cost 1

Working Set: Example

- σ = 2,4,2





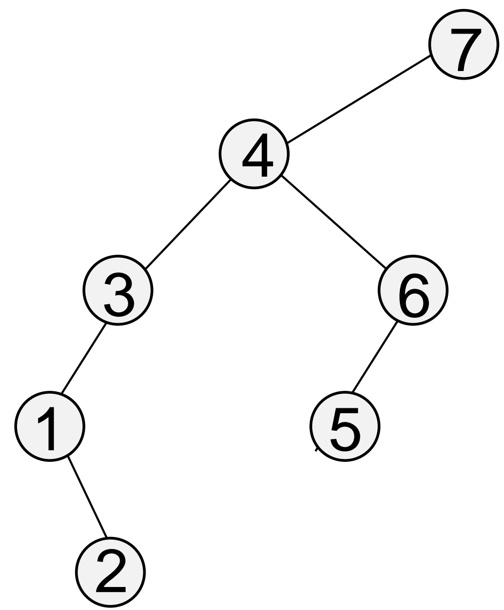
Cost 4

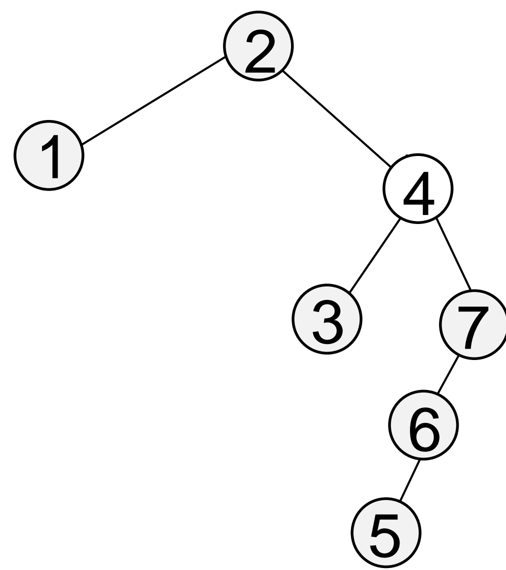
Cost 1

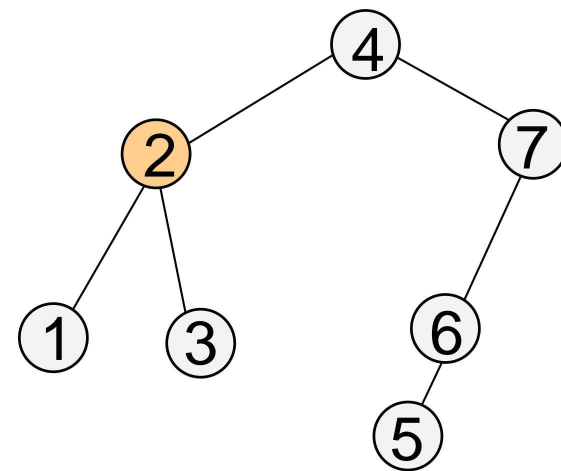
Cost 1

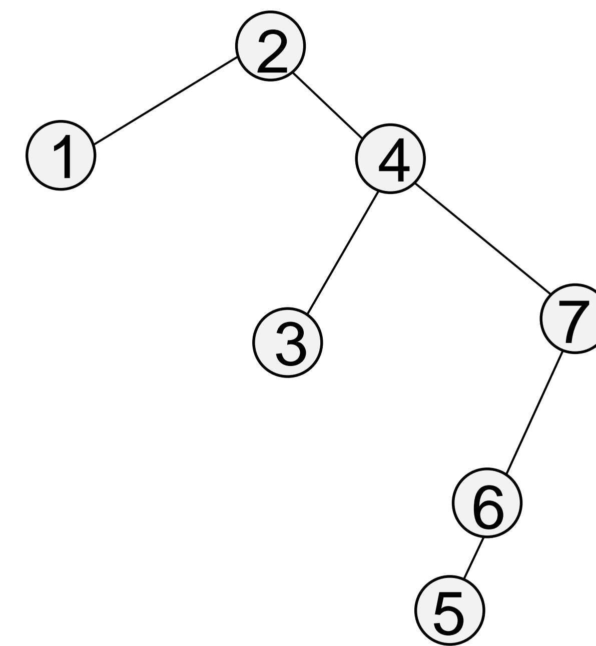
Working Set: Example

⇢ σ = 2,4,2









Cost 4

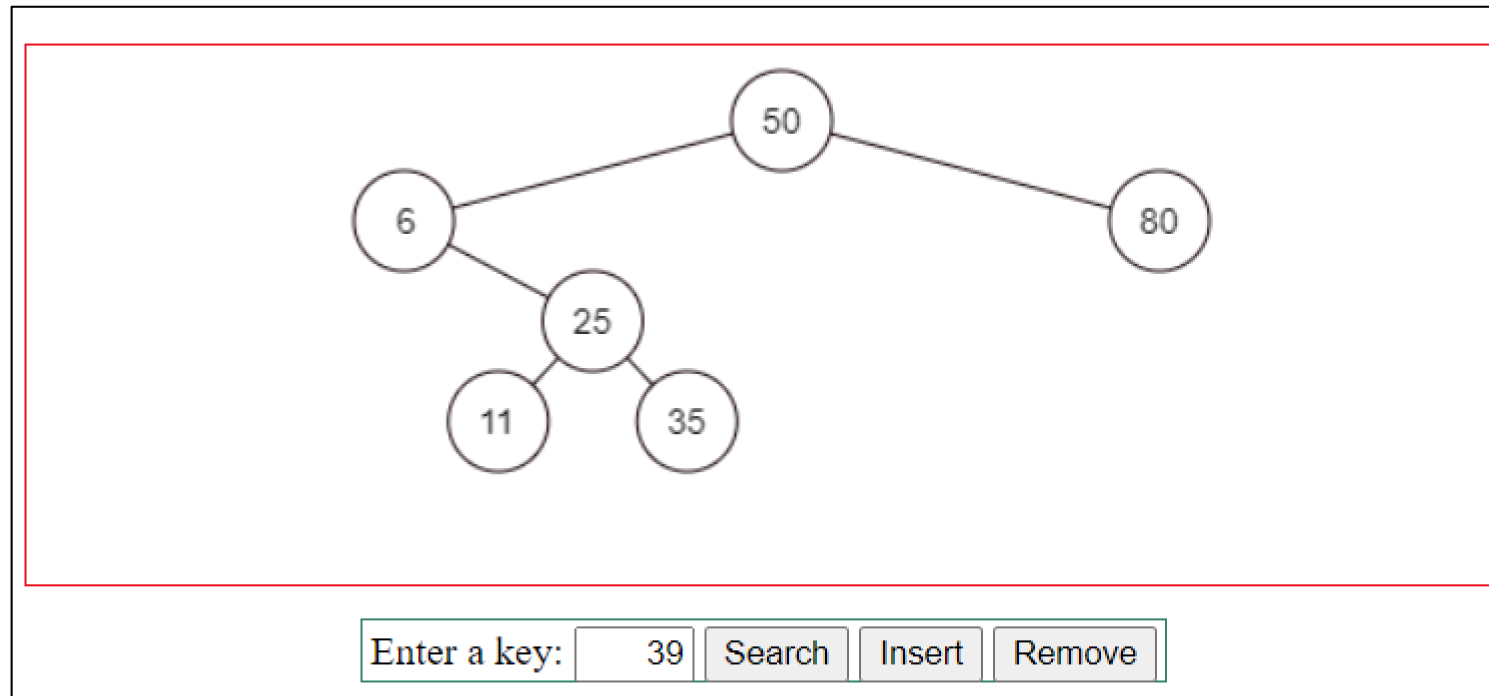
Cost 1

Cost 1

Amortized cost: (4+1+1)/3 = 2

Online Material

⇢ Splay tree animation by Y. Daniel Liang



https://yongdanielliang.github.io/animation/web/SplayTree.html

Access Lemma

Access Lemma enables much analysis of splay trees. We need:

⇢ Define for each node v arbitrary weight w(v)

⇢ Size of a node v, s(v), is sum of weights in v's subtree

⇢ Rank is logarithm of size: r(v)=log(s(v))

Access Lemma: The amortized time to splay a tree with root u at a node v is at most $3 \cdot (r(u) - r(v)) + 1 = O(\log(s(u)/s(v)))$.

Access Lemma

Access Lemma enables much analysis of splay trees. We need:

- ⇢ Define for each node v arbitrary weight w(v)
- ⇢ Size of a node v, s(v), is sum of weights in v's subtree
- ⇢ Rank is logarithm of size: r(v)=log(s(v))

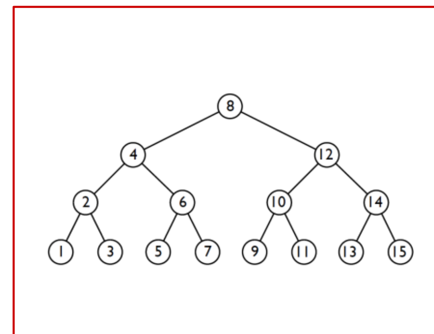
Access Lemma: The amortized time to splay a tree with root u at a node v is at most $3 \cdot (r(u) - r(v)) + 1 = O(\log(s(u)/s(v)))$.

Implications:

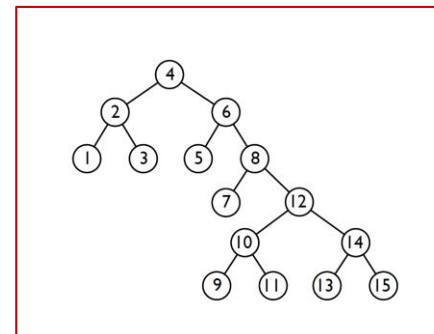
- ⇢ Working set property
- ⇢ Static optimality: splay tree is as efficient as any fixed search tree, including the optimum tree

What About Networks?

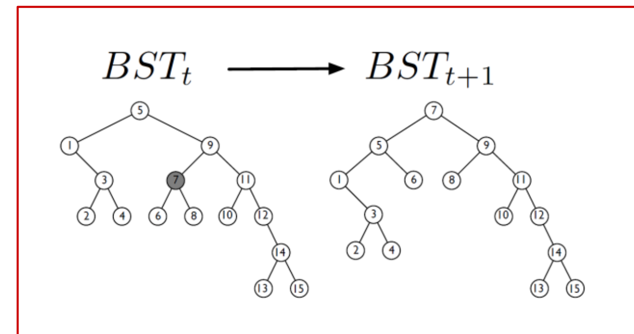
Traditional BST



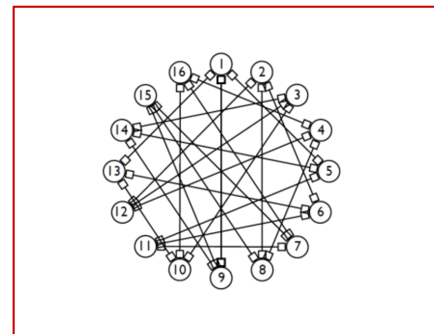
Demand-aware BST

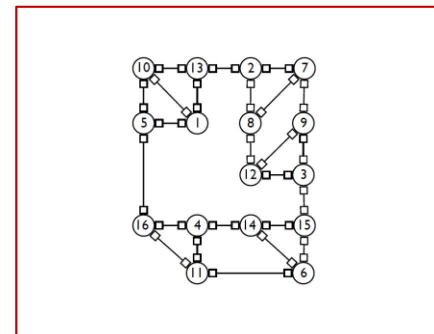


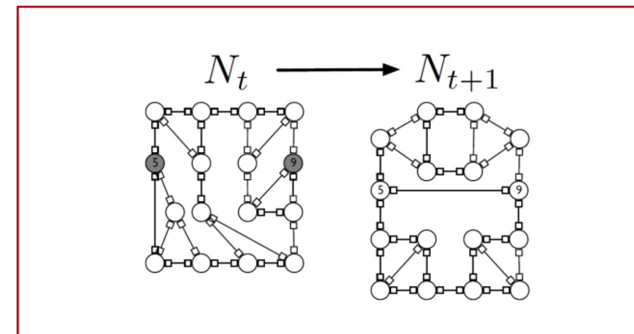
Self-adjusting BST



More structure: improved **access cost**





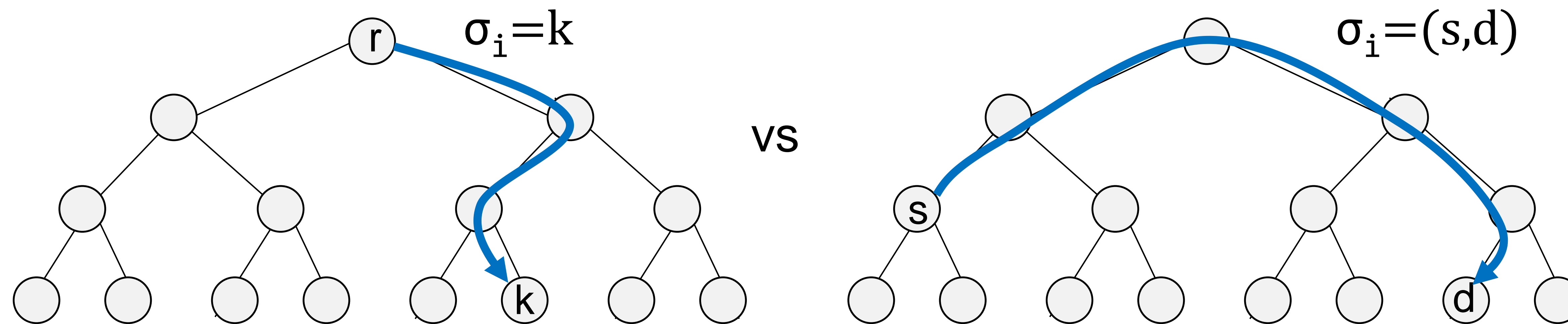


Reduced expected **route lengths**!

- Requests are not from root but between pairs of nodes
- Similar basic model: Cost=srv+adj, route length plus number of link changes

Splay Trees vs SplayNets

⇢ Requests from root vs between pairs of nodes



Request routing: simple local greedy rule

→ For node v:

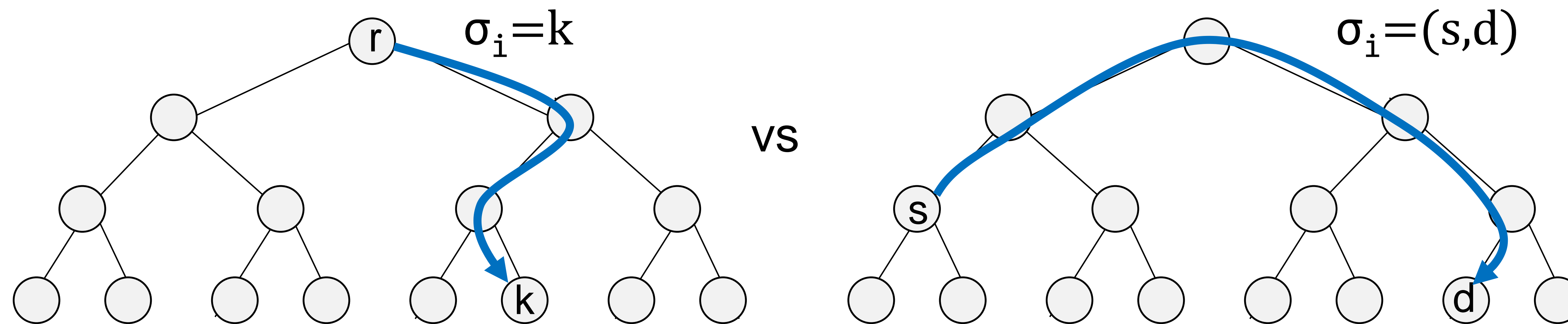
if key k < v: go left

if key k < v: go right

?

Splay Trees vs SplayNets

⇢ Requests from root vs between pairs of nodes

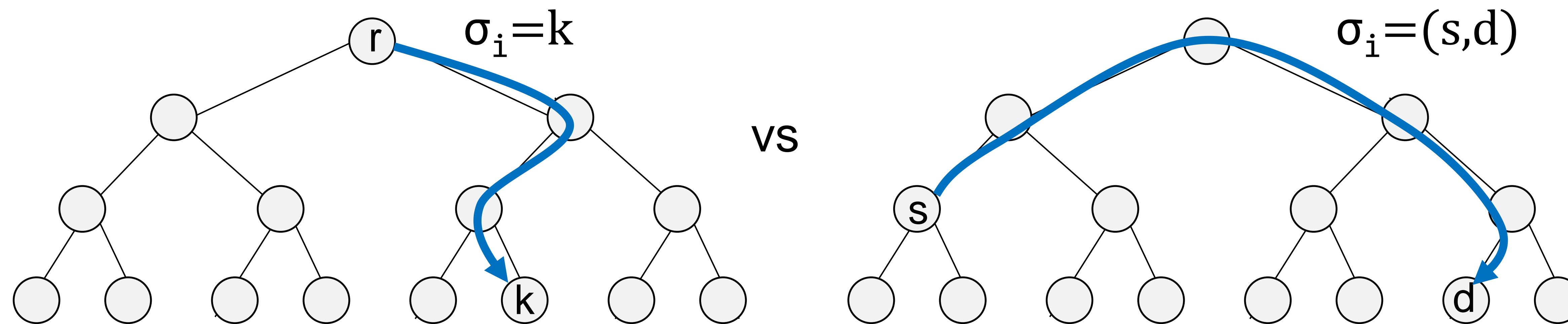


Request routing: simple local greedy rule
⇀ For node v:
   if key k < v: go left
   if key k < v: go right

Request routing: simple local greedy rule!
⇀ For node u: consider subtree T(u)
   u',u'': smallest and largest value in T(u)
   if destination u'≤d≤u: forward left
   if destination u≤d≤u'': forward right
   otherwise forward up

Splay Trees vs SplayNets

⇢ Requests from root vs between pairs of nodes



Request routing: simple local greedy rule
⇀ For node v:
  if key k < v: go left
  if key k < v: go right

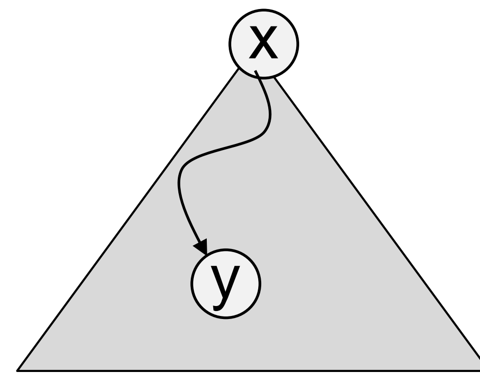
Request routing: simple local greedy rule!
⇀ For node u: consider subtree T(u)
  u’,u’’: smallest and largest value in T(u)
  if destination u’≤d≤u: forward left
  if destination u≤d≤u’’: forward right
  otherwise forward up

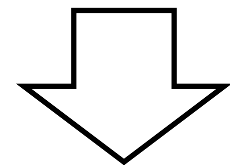
⇢ Greedy is attractive in dynamic networks: no route recomputation!

SplayNets Adjustments

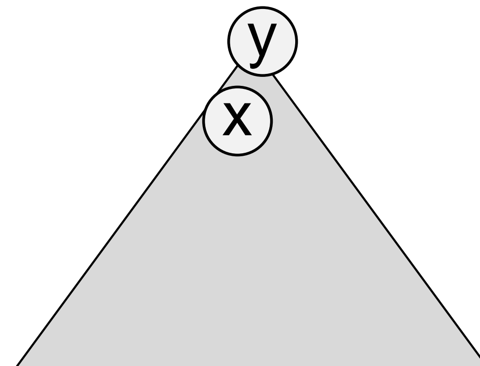
Splay Tree

1. Access



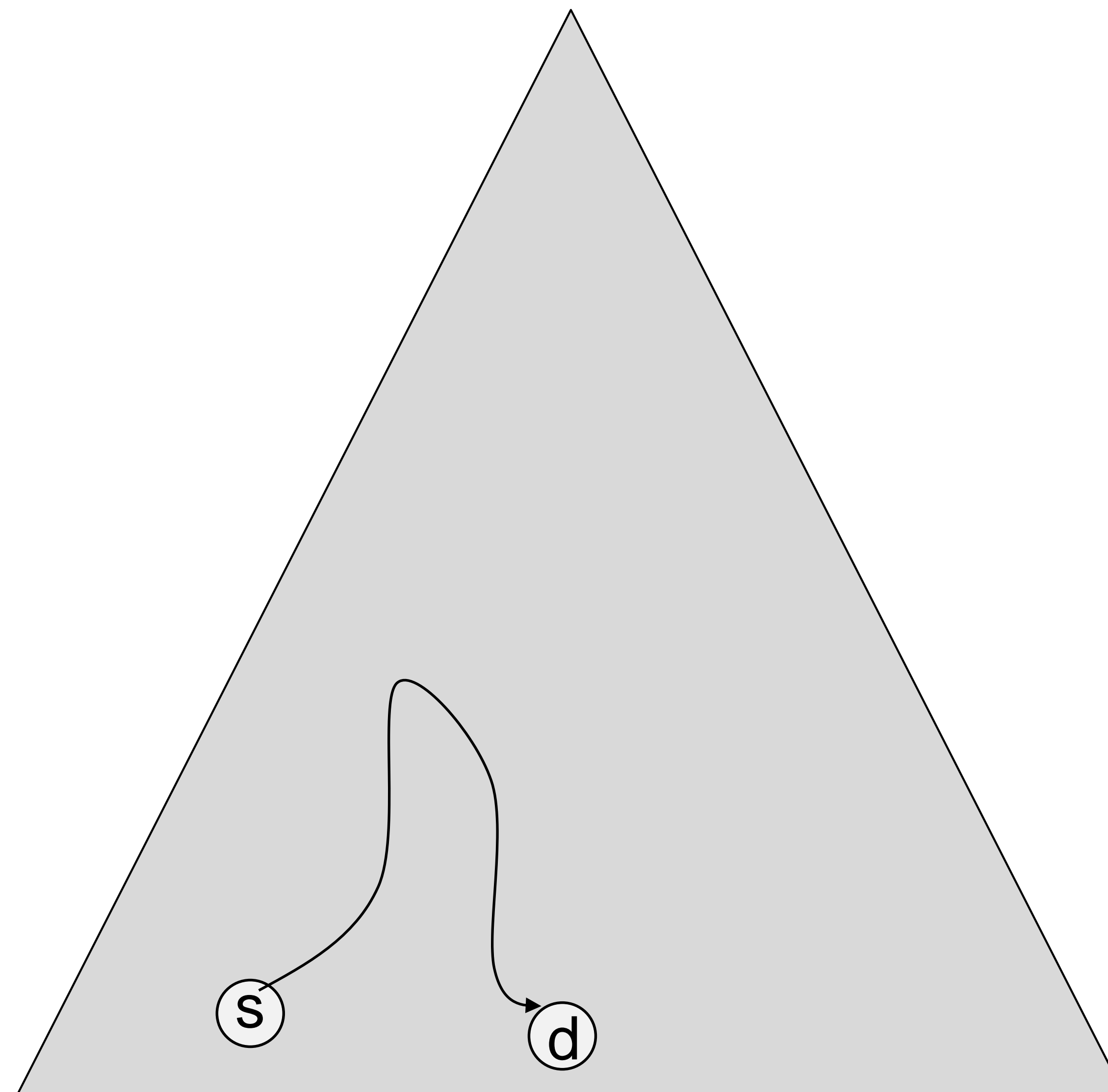


2. Splay to root



SplayNet

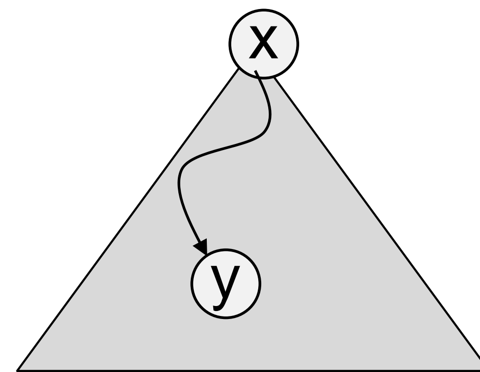
1. Route

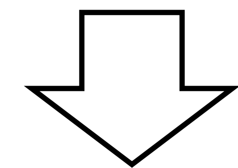


SplayNets Adjustments

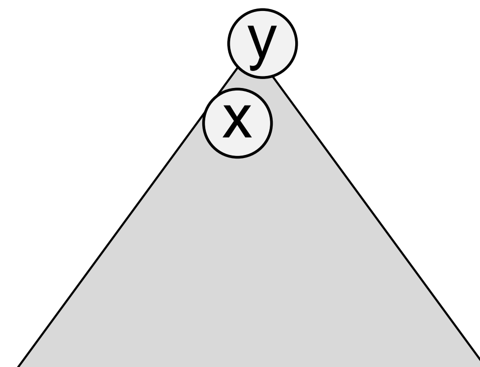
Splay Tree

1. Access



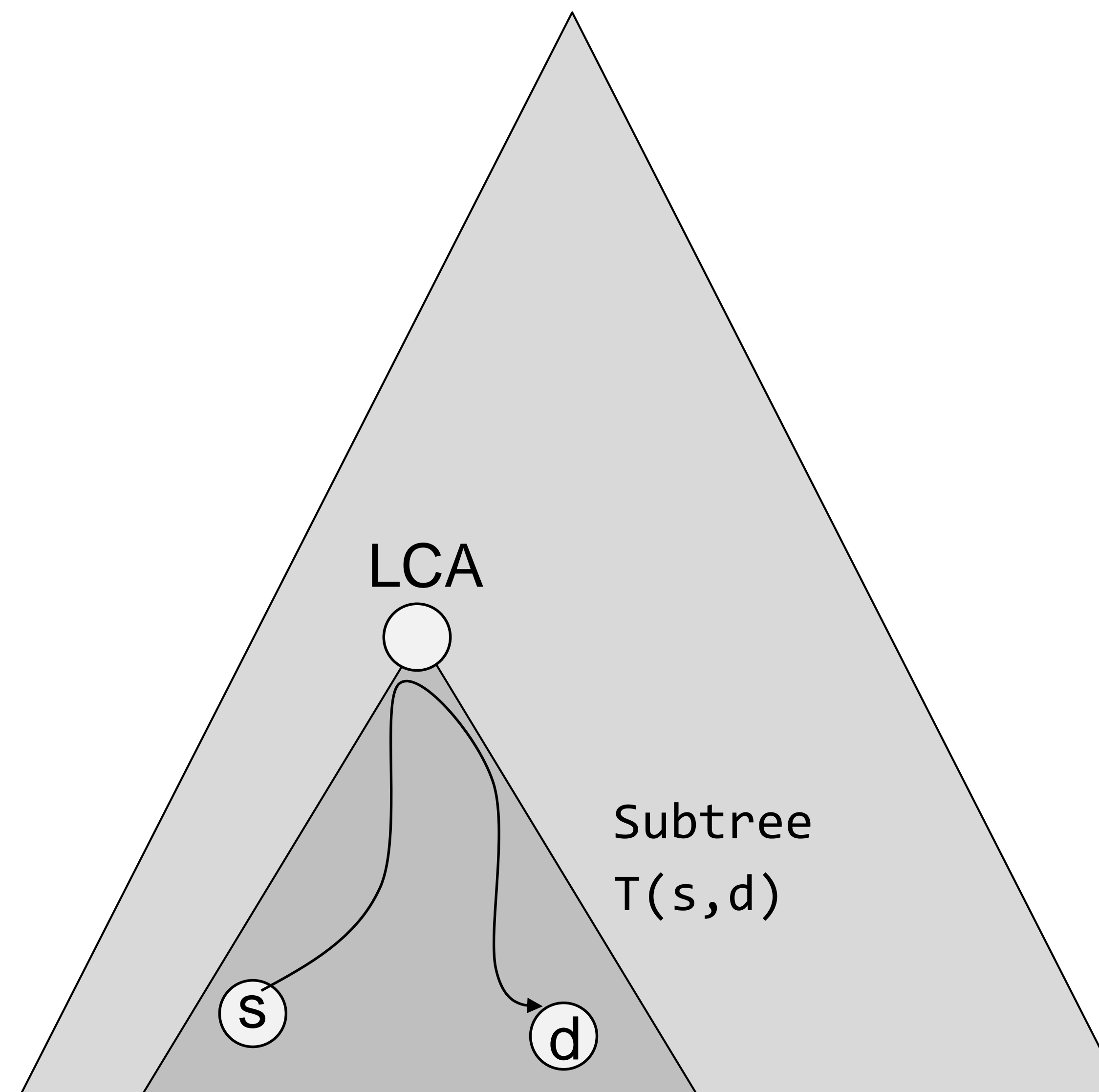


2. Splay to root



SplayNet

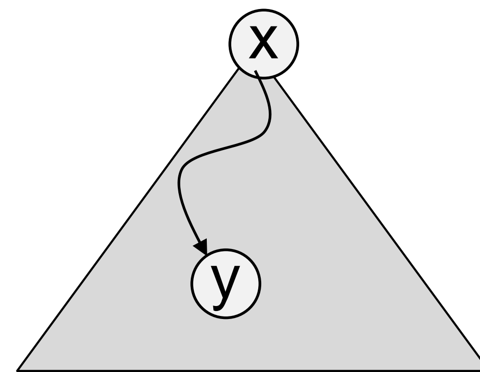
1. Route

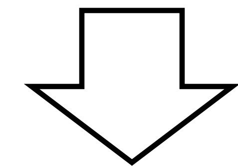


SplayNets Adjustments

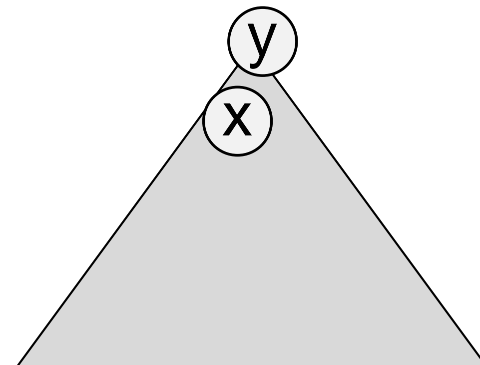
Splay Tree

1. Access



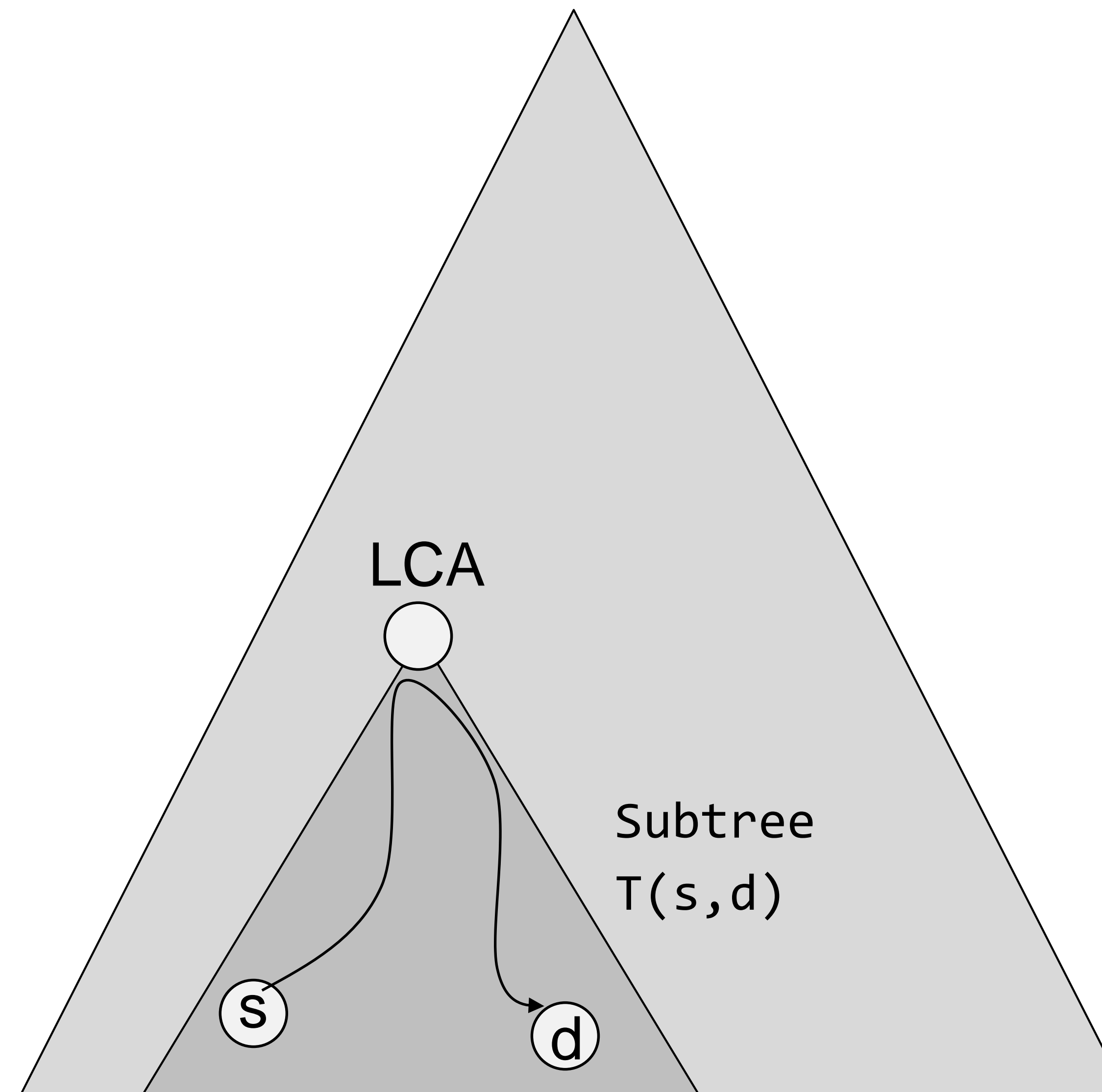


2. Splay to root



SplayNet

1. Route

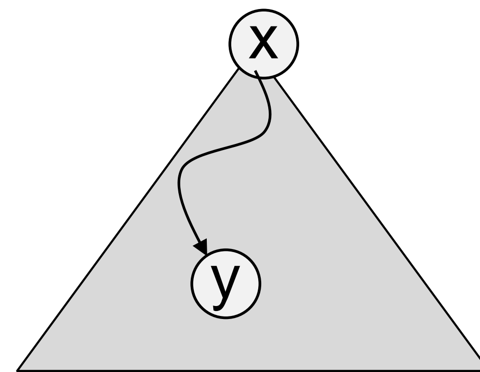


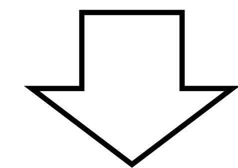
2. Splay s to LCA -> T(s,d)
Splay d to child of s

SplayNets Adjustments

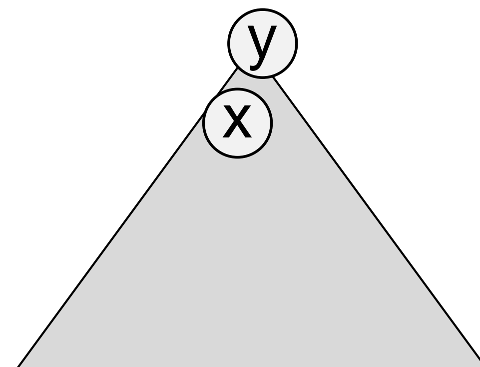
Splay Tree

1. Access



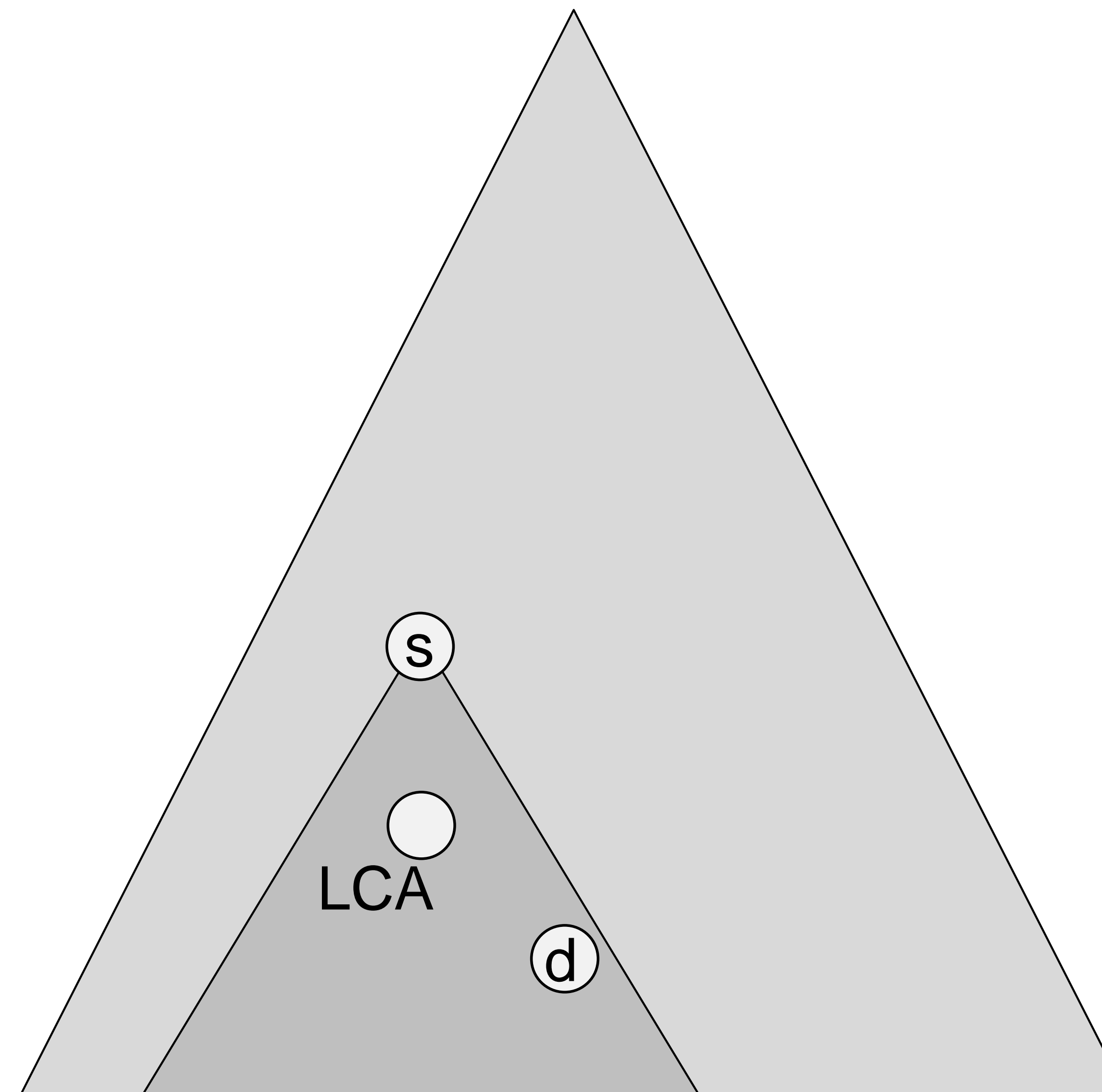


2. Splay to root



SplayNet

1. Route

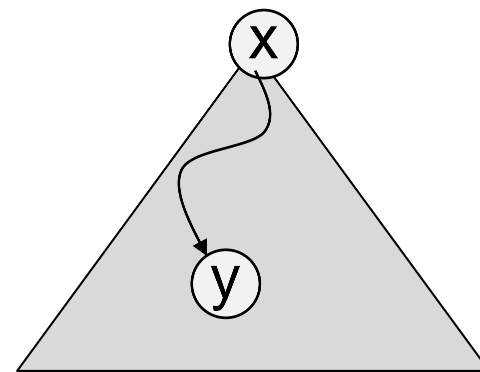


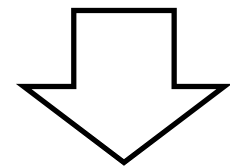
2. Splay s to LCA -> T(s,d)
   Splay d to child of s

SplayNets Adjustments

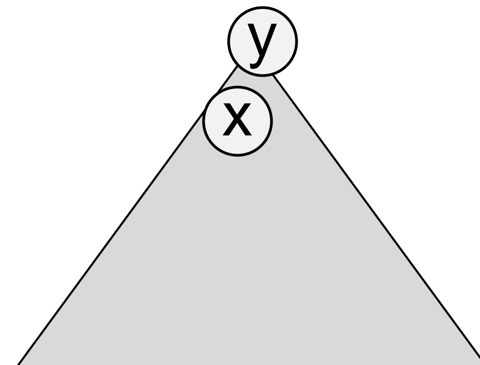
Splay Tree

1. Access



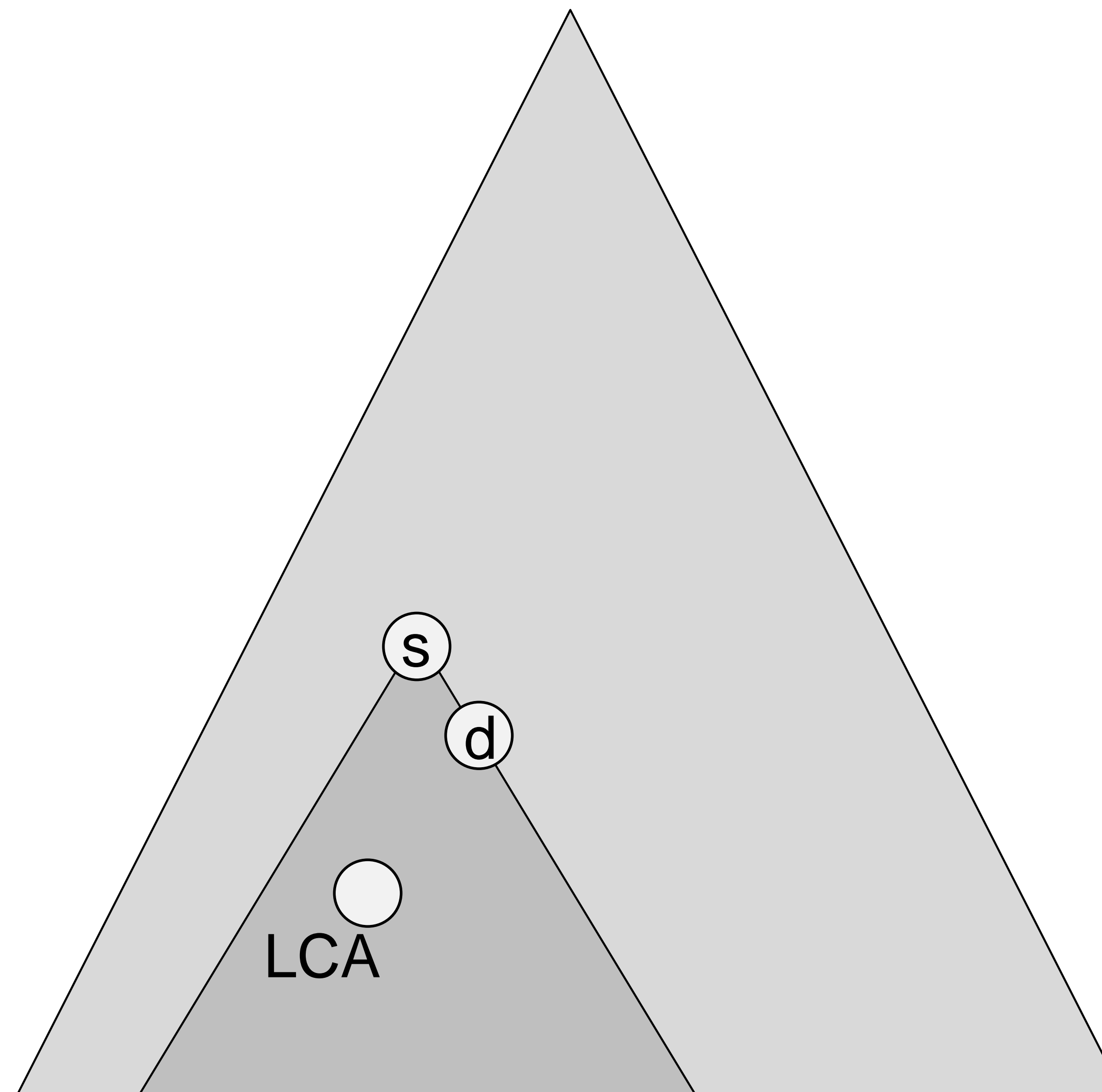


2. Splay to root



SplayNet

1. Route

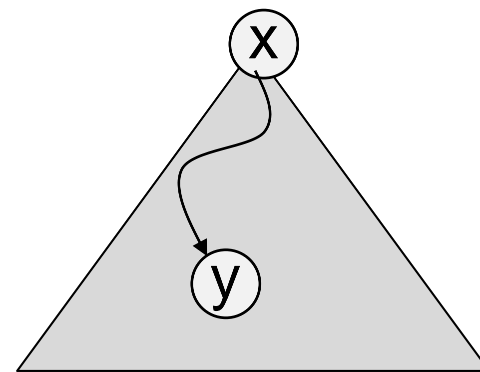


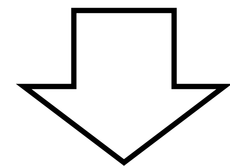
2. Splay s to LCA -> T(s,d)
   Splay d to child of s

SplayNets Adjustments

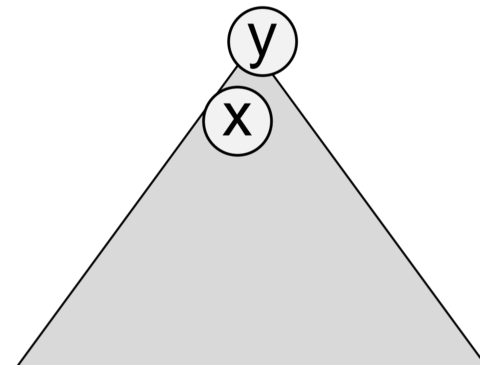
Splay Tree

1. Access



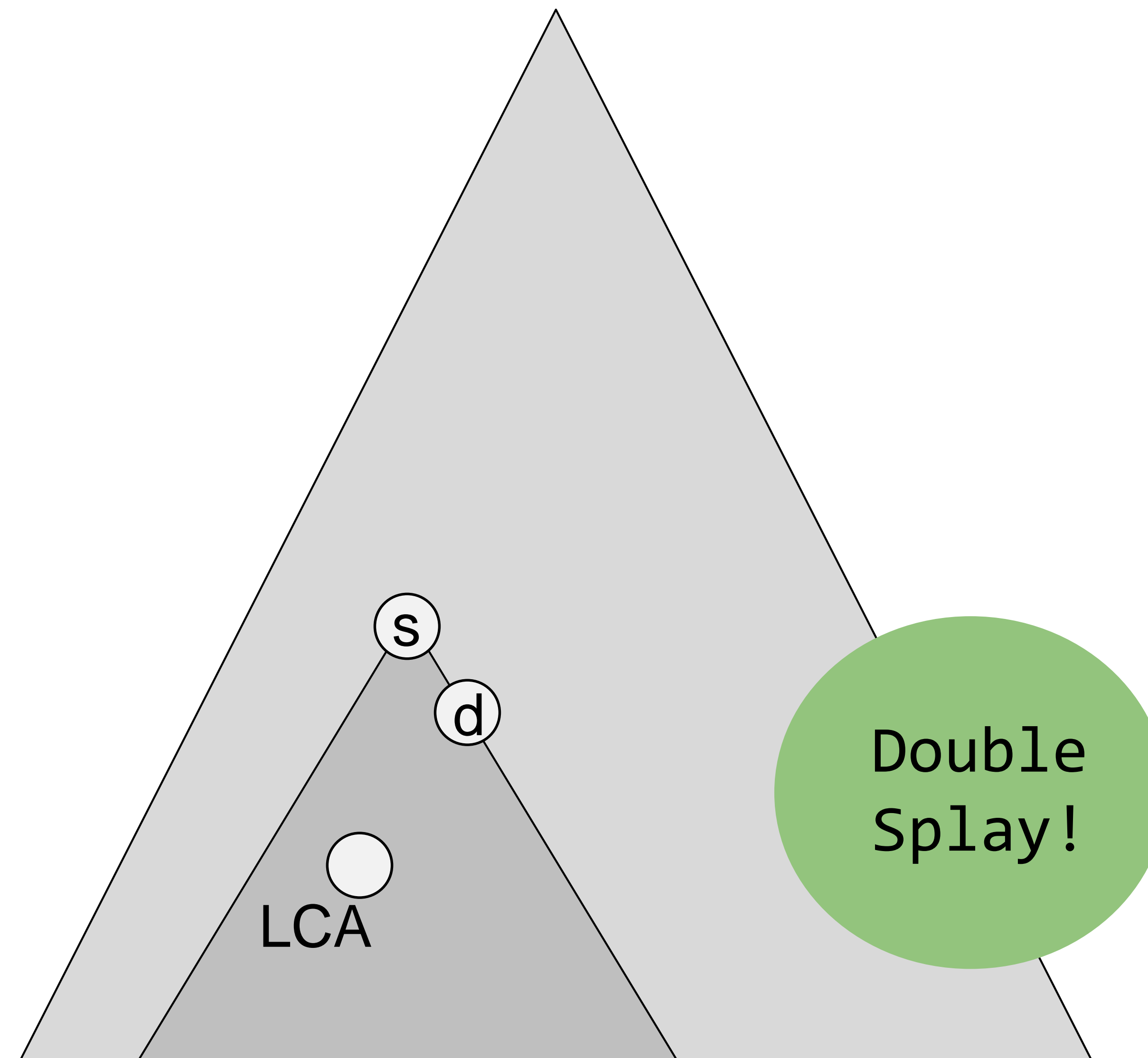


2. Splay to root



SplayNet

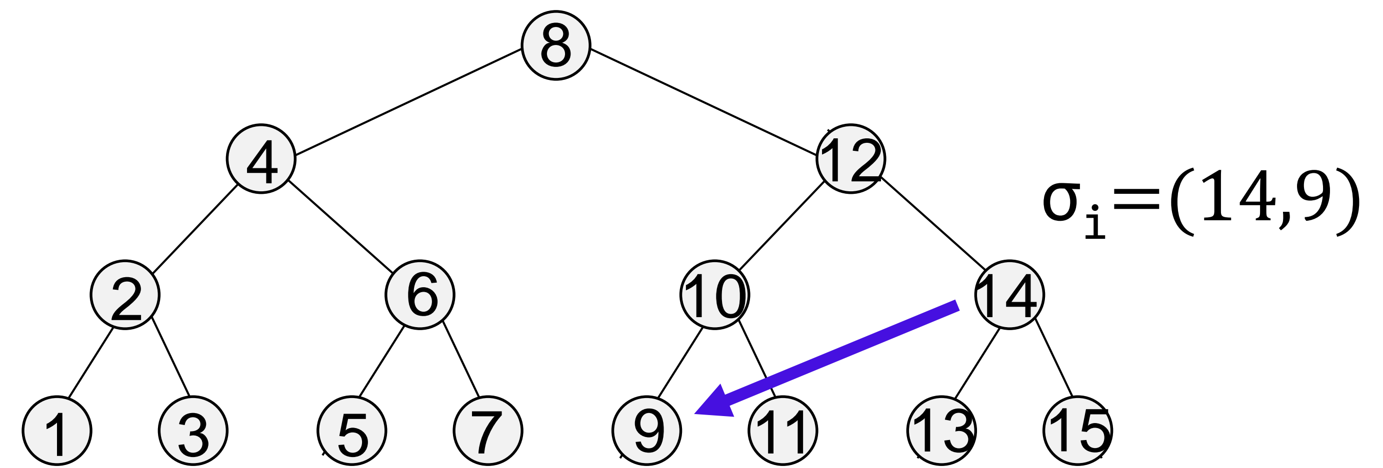
1. Route



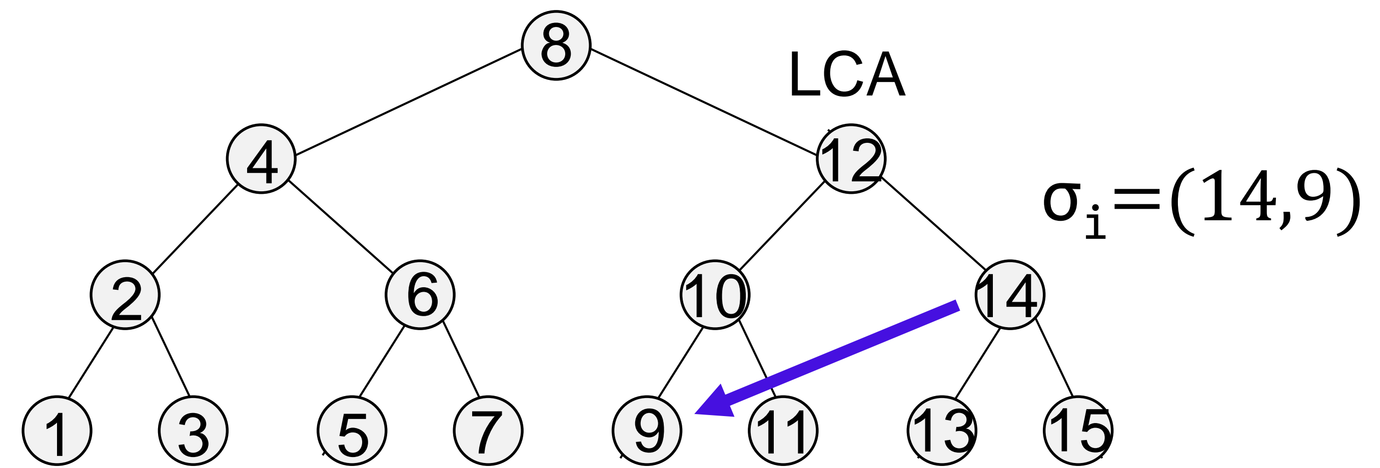
Double Splay!

2. Splay s to LCA -> T(s,d)
Splay d to child of s

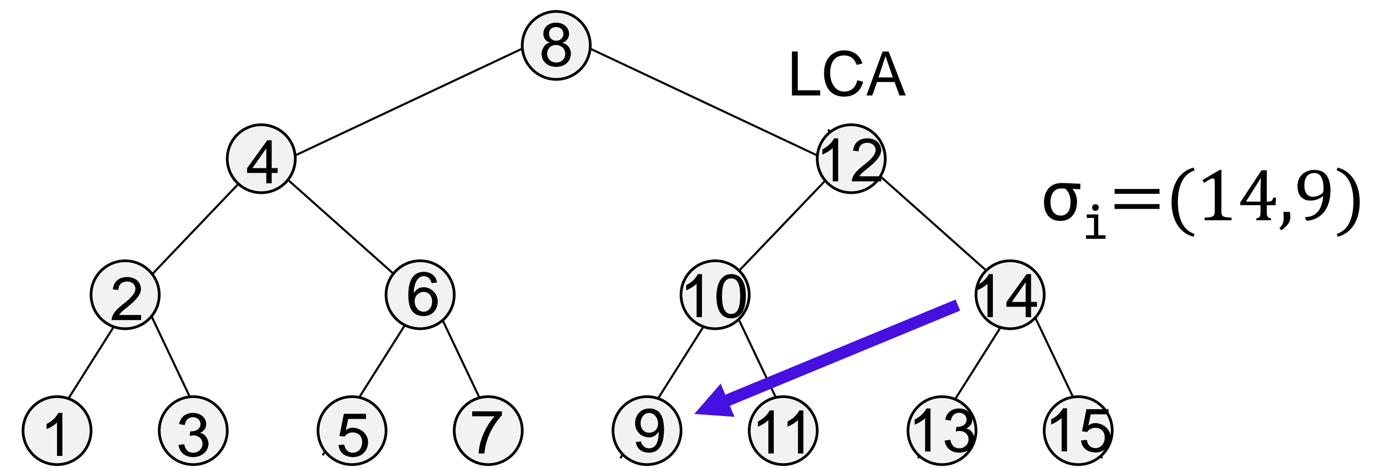
Example

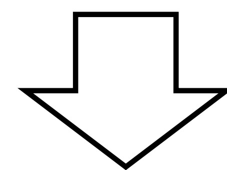


Example

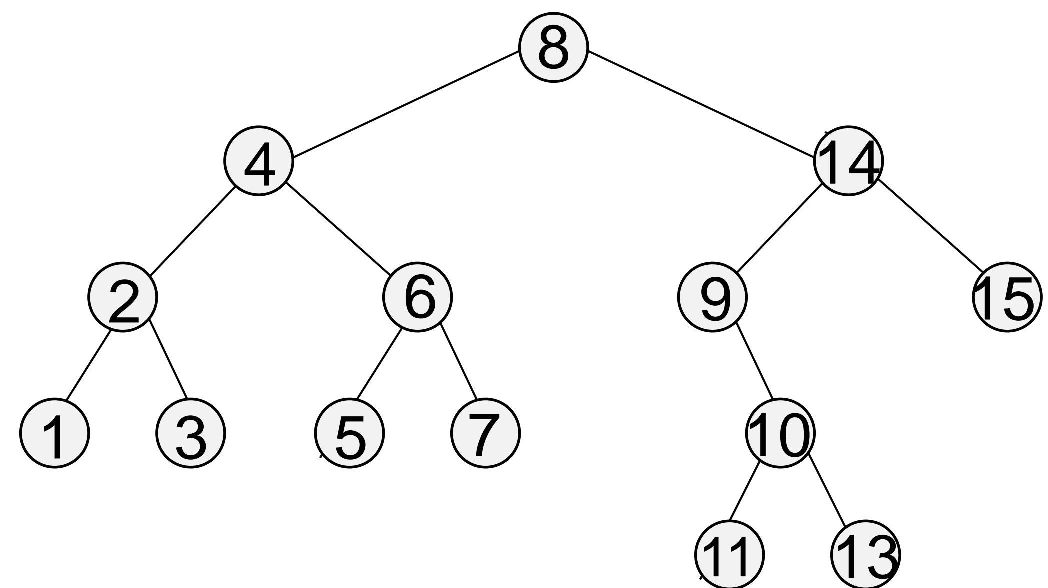


Example

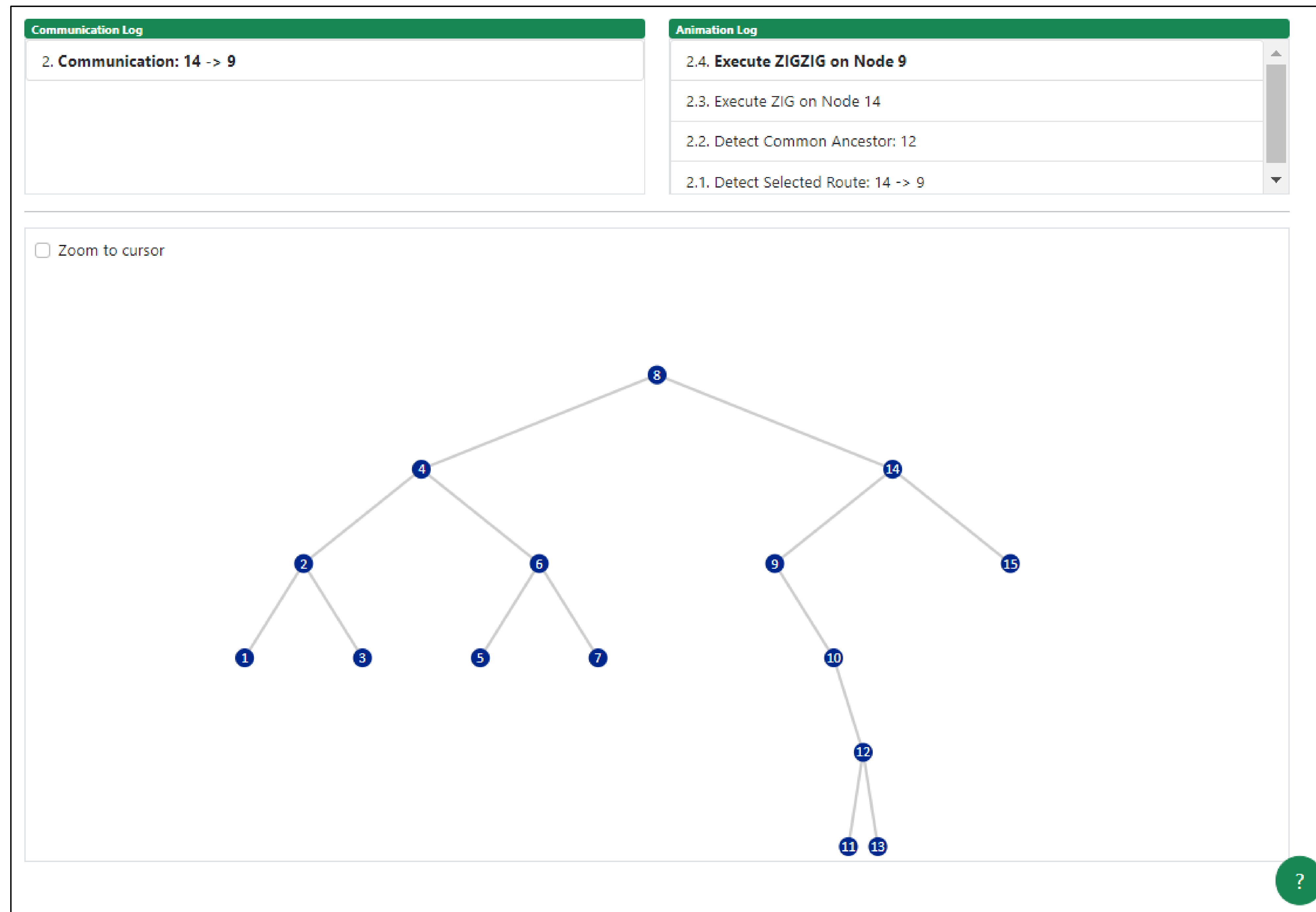




Splay 14 to LCA
Splay 9 to child of 14



Online Material



https://self-adjusting.net/slaynet-visualization/

Analysis: Upper Bound

Lemma 1:
Cost(SplayNet, σ) = O(H(X)+H(Y))

Proof: For any node v: let s(v) = # v appears as a source in σ, and similarly d(v) as a destination. Assign each node v two weights s(v)/m and d(v)/m and analyze the two basic operations of SplayNet separately: first splaying the source to the common ancestor and second splaying the destination to the new common ancestor.

So: Cost(SplayNet, σ) = 1/m ∑ splay src(σ_t) + splay dst(σ_t)

Since the size of a source v is at least s(v)/m and the size of any node is at most 1 (analogously for the case of destinations: just replace s(v) by d(v)), the claim follows from Access Lemma.

Analysis: Upper Bound

Lemma 1:
Cost(SplayNet, σ) = O(H(X)+H(Y))

Proof: For any node v: let s(v) = # v appears as a source in σ, and similarly d(v) as a destination. Assign each node v two weights s(v)/m and d(v)/m and analyze the two basic operations of SplayNet separately: first splaying the source to the common ancestor and second splaying the destination to the new common ancestor.

$$\text{So: Cost(SplayNet}, \sigma) = 1/m \sum \underbrace{\text{splay src}(\sigma_t)}_{H(X)} + \underbrace{\text{splay dst}(\sigma_t)}_{H(Y)}$$

Since the size of a source v is at least s(v)/m and the size of any node is at most 1 (analogously for the case of destinations: just replace s(v) by d(v)), the claim follows from Access Lemma.

Analysis: Lower Bound

Lemma 2:
Cost(SplayNet, σ) = $\Omega(H(Y|X)+H(X|Y))$

Proof: For any node x, let Y_x denote the frequency distribution of the destinations given that the source is x.

⇢ Consider an optimal tree T with root x. We know from the biased BST that the average path length of requests is $\Omega(H(Y_x))$.

⇢ Considering the optimal tree for each source, we have a cost of at least $\sum f(x_i)H(Y_{xi}) = \Omega(H(Y|X))$.

⇢ Similar argument holds for destinations.

Optimality

Product distribution: special but important distribution! The probability $p(u,v)$ of a communication request (u,v) can be described by the product of the activity levels $p(u)$ and $p(v)$ of the nodes, i.e., $p(u,v) = p(u) \cdot p(v)$.

Theorem 1:
SplayNets are optimal if σ follows product distribution.

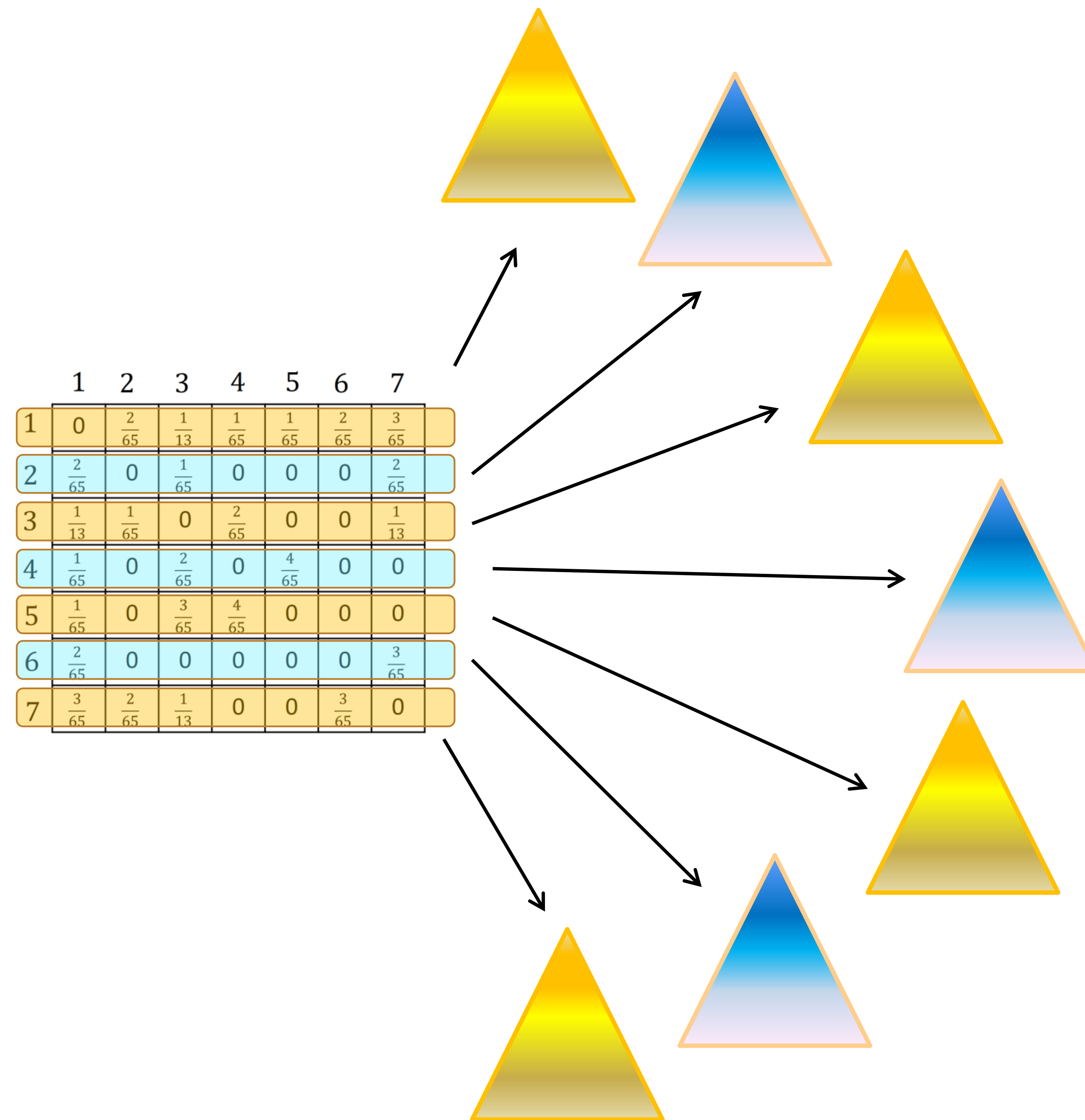
Proof: Follows from Lemma 1 and Lemma 2: from the independence it follows that the entropy of the communication sources given the destinations equals the entropy of the communication sources only (i.e., $H(X|Y) = H(X)$), and vice versa for the destinations.

Alternative Approach to Design S-DANs

Revisiting Ego-Trees

Recall Algorithm for

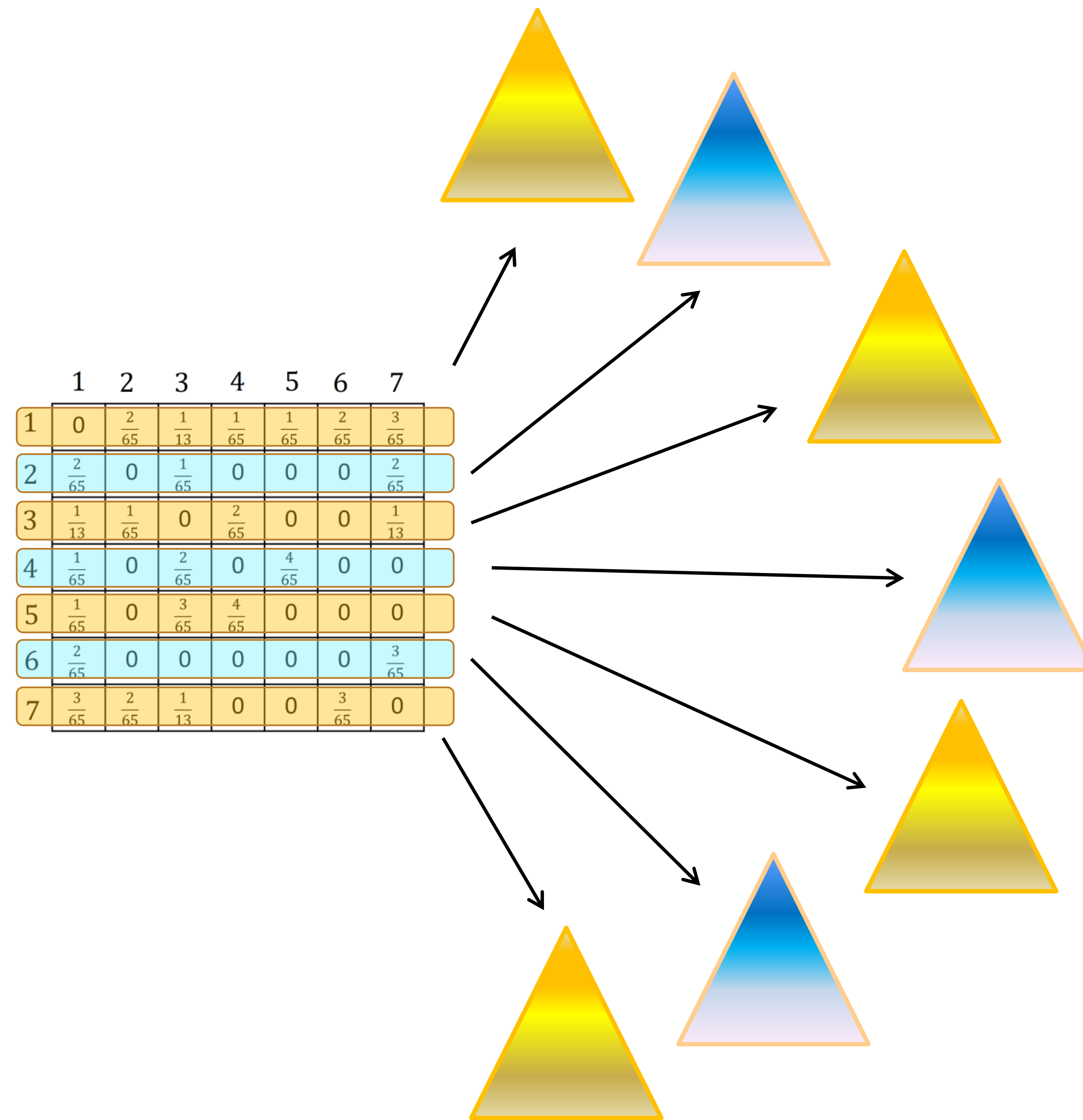
S-DANs



- ⇢ Idea: union of „ego-tree“
- ⇢ Reduce degree

An Idea for an Algorithm for

D-DANs



	1	2	3	4	5	6	7
1	0	$\frac{2}{65}$	$\frac{1}{13}$	$\frac{1}{65}$	$\frac{1}{65}$	$\frac{2}{65}$	$\frac{3}{65}$
2	$\frac{2}{65}$	0	$\frac{1}{65}$	0	0	0	$\frac{2}{65}$
3	$\frac{1}{13}$	$\frac{1}{65}$	0	$\frac{2}{65}$	0	0	$\frac{1}{13}$
4	$\frac{1}{65}$	0	$\frac{2}{65}$	0	$\frac{4}{65}$	0	0
5	$\frac{1}{65}$	0	$\frac{3}{65}$	$\frac{4}{65}$	0	0	0
6	$\frac{2}{65}$	0	0	0	0	0	$\frac{3}{65}$
7	$\frac{3}{65}$	$\frac{2}{65}$	$\frac{1}{13}$	0	0	$\frac{3}{65}$	0

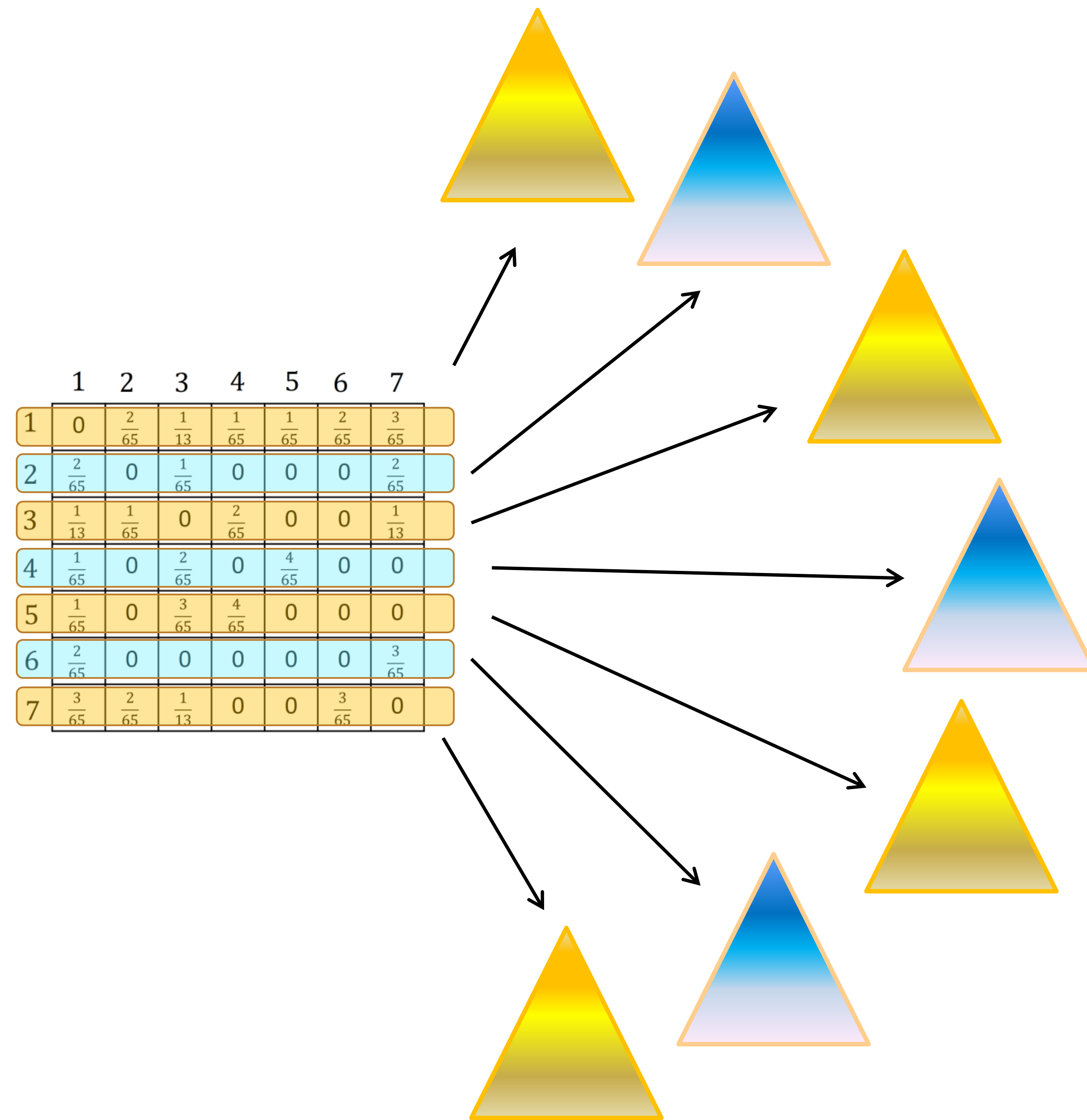
⇢ Idea: union of „ego-tree“

⇢ Reduce degree

Make Ego-tree Dynamic

An Idea for an Algorithm for

D-DANs



⇢ Idea: union of „ego-tree“

⇢ Reduce degree

Make Ego-tree Dynamic

E.g., splay tree, push-down trees, …

Learning Goals

After this lecture, you will understand:

⇢ How binary search trees can become self-adjusting

⇢ A model for self-adjusting networks and desirable properties

⇢ SplayNets, a self-adjusting networks

⇢ How ego-trees can be used to make networks self-adjusting

Further Reading

SplayNet: Towards Locally Self-Adjusting Networks

Stefan Schmid*, Chen Avin*, Christian Scheideler, Michael Borokhovich, Bernhard Haeupler, Zvi Lotker

***Abstract*—This paper initiates the study of locally self-adjusting networks: networks whose topology adapts dynamically and in a decentralized manner, to the communication pattern σ. Our vision can be seen as a distributed generalization of the self-adjusting datastructures introduced by Sleator and Tarjan [22]: In contrast to their splay trees which dynamically optimize the lookup costs from a *single node* (namely the tree root), we seek to minimize the routing cost between arbitrary *communication pairs* in the network.**

As a first step, we study distributed binary search trees (BSTs), which are attractive for their support of greedy routing. We introduce a simple model which captures the fundamental tradeoff between the benefits and costs of self-adjusting networks. We present the *SplayNet* algorithm and formally analyze its performance, and prove its optimality in specific case studies. We also introduce lower bound techniques based on interval cuts and edge expansion, to study the limitations of any demand-optimized network. Finally, we extend our study to multi-tree networks, and highlight an intriguing difference between classic and distributed splay trees.

I. Introduction

In the 1980s, Sleator and Tarjan [22] proposed an appealing new paradigm to design efficient Binary Search Tree (BST) datastructures: rather than optimizing traditional metrics such as the search tree depth in the *worst-case*, their *splay* datastruc- toward static metrics, such as the diameter or the length of the longest route: the self-adjusting paradigm has not spilled over to distributed networks yet.

We, in this paper, initiate the study of a distributed generalization of self-optimizing datastructures. This is a non-trivial generalization of the classic splay tree concept: While in classic BSTs, a *lookup request* always originates from the same node, the tree root, distributed datastructures and networks such as skip graphs [2], [13] have to support *routing requests* between arbitrary pairs (or *peers*) of communicating nodes; in other words, both the source as well as the destination of the requests become variable. Figure 1 illustrates the difference between classic and distributed binary search trees.

In this paper, we ask: Can we reap similar benefits from self-adjusting *entire networks*, by adaptively reducing the distance between frequently communicating nodes?

As a first step, we explore fully decentralized and self-adjusting Binary Search Tree networks: in these networks, nodes are arranged in a binary tree which respects node identifiers. A BST topology is attractive as it supports greedy routing: a node can decide locally to which port to forward a request given its destination address.

References

Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)* 32.3 (1985): 652-686.

Stefan Schmid, Chen Avin, Christian Scheideler, Michael Borokhovich, Bernhard Haeupler, and Zvi Lotker. SplayNet: Towards Locally Self-Adjusting Networks. IEEE/ACM Transactions on Networking (TON), Volume 24, Issue 3, 2016.

Chen Avin and Stefan Schmid. ReNets: Statically-Optimal Demand-Aware Networks. SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS), Alexandria, Virginia, USA, January 2021.