Online Algorithms
Tutorial 4 — Self-Organizing Datastructures

Self-Organizing Datastructures

Consider a set- or dictionary-datastructure:

- C++ map, set, etc.
- Java HashMap, TreeMap, etc.
- Python dict, set
- ...

Self-Organizing Datastructures

Consider a set- or dictionary-datastructure:

- C++ map, set, etc.
- Java HashMap, TreeMap, etc.
- Python dict, set
- ...

Operations:

- Find given key
- Add key
- Delete key

Classic vs. Self-organizing

Classic:

- Insert, Delete change the set and internal representation
- Find does not change anything

Classic vs. Self-organizing

Classic:

- Insert, Delete change the set and internal representation
- Find does not change anything

Self-organizing:

- Insert, Delete change the set and internal representation
- Find may change internal representation

Classic vs. Self-organizing

Classic:

- Insert, Delete change the set and internal representation
- Find does not change anything

Self-organizing:

- Insert, Delete change the set and internal representation
- Find may change internal representation

Pro's & Con's:

- Possibly faster if some keys are requested more often than others
- Can be used in data compression
- More complicated
- Bad for multithreading

Homework: Linked lists

Linked List:

- Not a very practical set implementation
- Can be used for compression
- Instead of encoding bytes or byte pairs, encode list positions
- Length depends on position in list
- Common byte pairs become shorter, uncommon ones longer

Results:

- MoveToFront 2-competitive
- FrequencyCount, Transpose not competitive

Arrays? Double-ended queues?

If sets are small or don't change much, (sorted) arrays are practical. Iteration is often more important than queries.

Arrays? Double-ended queues?

If sets are small or don't change much, (sorted) arrays are practical. Iteration is often more important than queries.

MoveToFront does not work well for arrays.

Arrays? Double-ended queues?

If sets are small or don't change much, (sorted) arrays are practical. Iteration is often more important than queries.

MoveToFront does not work well for arrays.

Double-ended queues or ring-buffers can work.

Hash tables

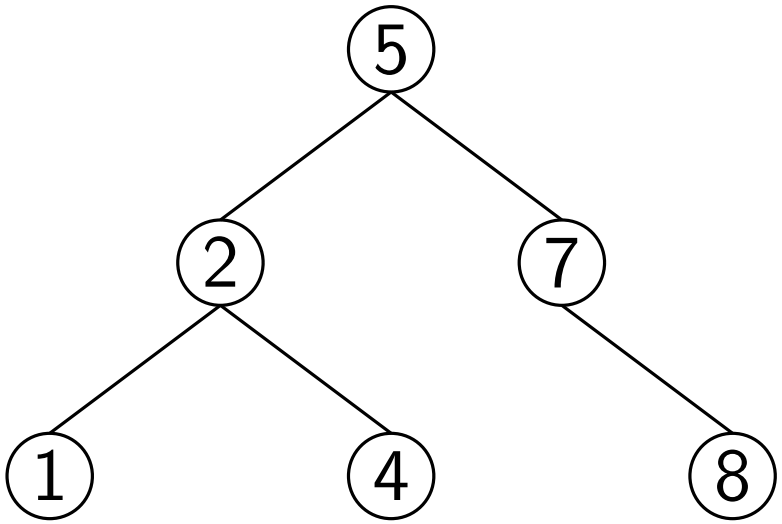
Have not been studied much; not much potential as self-organizing datastructures. Collisions should be rare.

Binary trees

A lot of potential for self-organization: There are very many different binary trees that encode the same set.

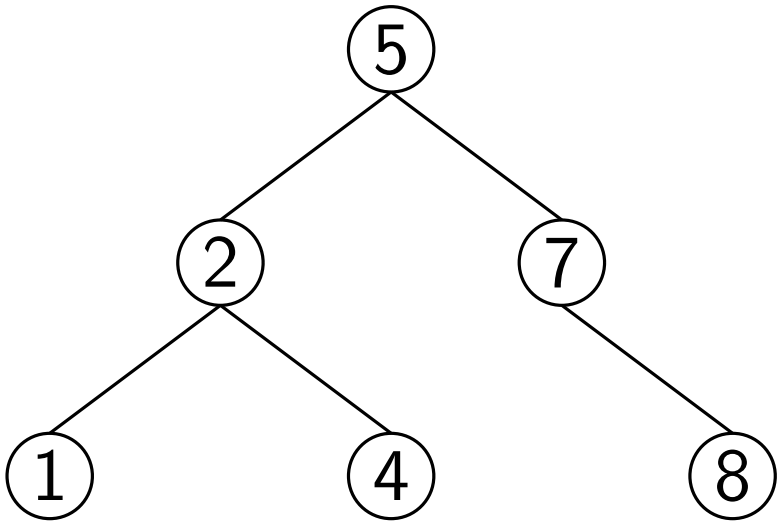
Binary trees

A lot of potential for self-organization: There are very many different binary trees that encode the same set.



Binary trees

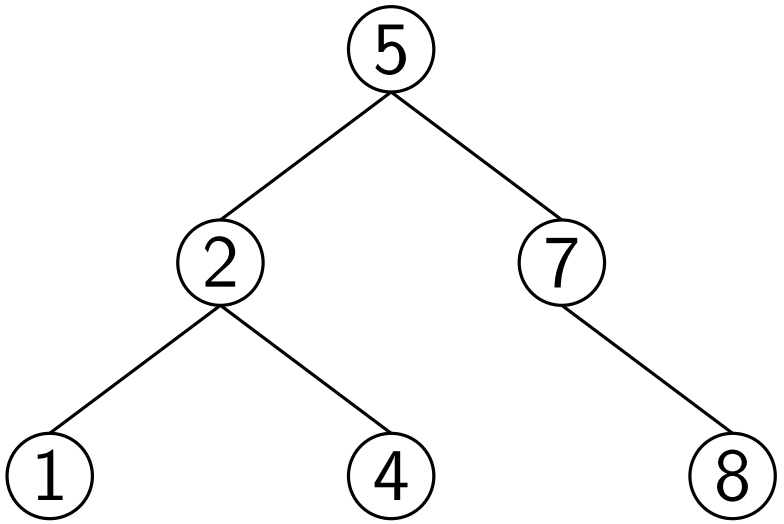
A lot of potential for self-organization: There are very many different binary trees that encode the same set.



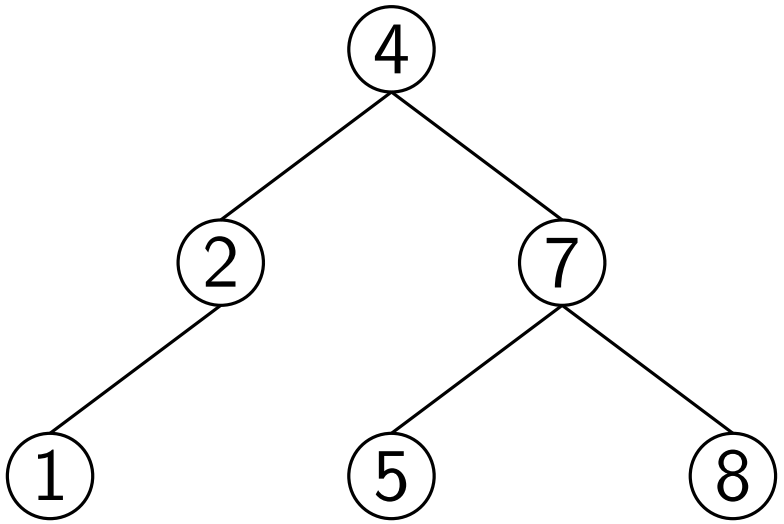
Search 5: Costs 1
Search 4: Costs 3
Search 7: Costs 2

Binary trees

A lot of potential for self-organization: There are very many different binary trees that encode the same set.



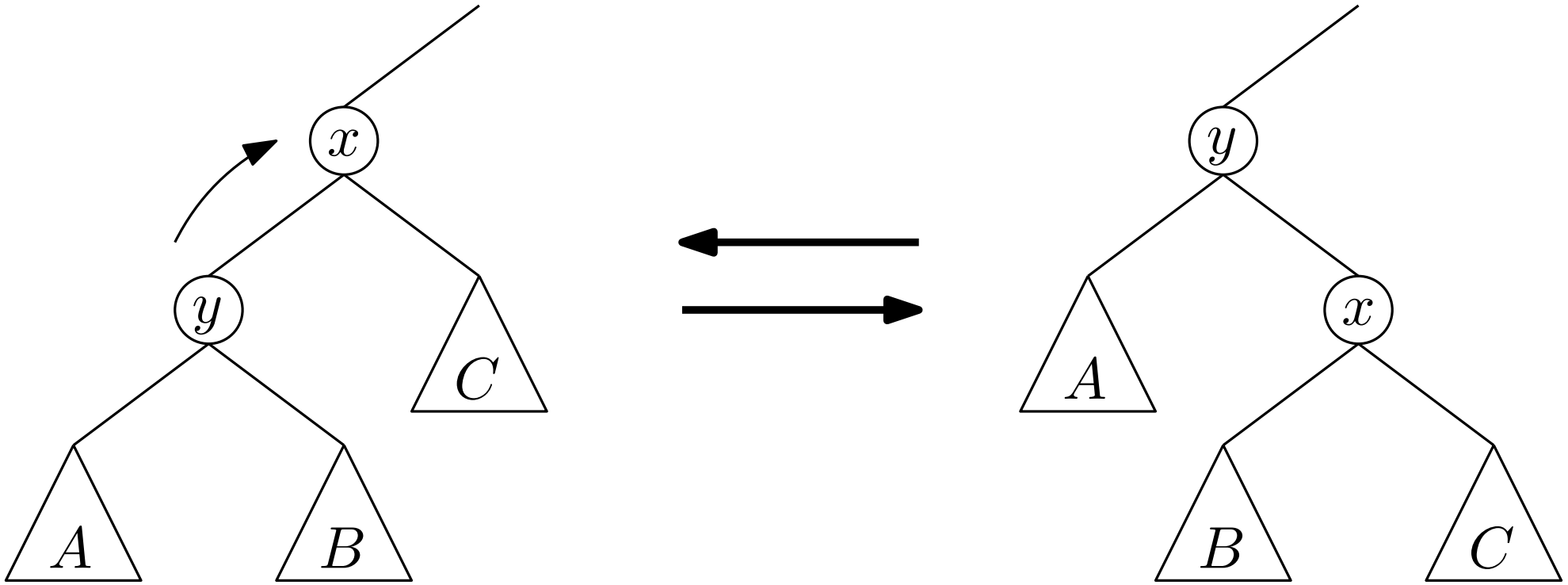
Search 5: Costs 1
Search 4: Costs 3
Search 7: Costs 2



Search 5: Costs 3
Search 4: Costs 1
Search 7: Costs 2

Operations on trees

Rotations: Transform one binary tree into the other.



Cost: 1 unit. Can also be used to realize AVL-trees. Can be applied anywhere in the tree.

Rotations

Can we move from any tree to any other by rotations?

Rotations

Can we move from any tree to any other by rotations?

Yes! Reaching a right path needs at most $n - 1$ rotations:

- We can always rotate a vertex into the right path
- Increases number of edges on the right path by 1
- In total, we have $n - 1$ edges

Rotations

Can we move from any tree to any other by rotations?

Yes! Reaching a right path needs at most $n - 1$ rotations:

- We can always rotate a vertex into the right path
- Increases number of edges on the right path by 1
- In total, we have $n - 1$ edges

To transform from one tree to another: Transform both to right path, apply rotations from target to right path in reverse.

$$\Rightarrow \text{at most } 2n - 2 \text{ rotations.}$$

Rotations

Can we move from any tree to any other by rotations?

Yes! Reaching a right path needs at most $n - 1$ rotations:

- We can always rotate a vertex into the right path
- Increases number of edges on the right path by 1
- In total, we have $n - 1$ edges

To transform from one tree to another: Transform both to right path, apply rotations from target to right path in reverse.

$$\Rightarrow \text{at most } 2n - 2 \text{ rotations.}$$

Actually, $2n - 6$ are sufficient.

Competitiveness

Relaxed competitive ratio $f(n)$ (similar to asymptotic):

$$A(\sigma, T_0) \leq f(n) \cdot \mathrm{OPT}(\sigma, T_0) + \mathcal{O}(n).$$

Competitiveness

Relaxed competitive ratio $f(n)$ (similar to asymptotic):

$$A(\sigma, T_0) \leq f(n) \cdot \mathrm{OPT}(\sigma, T_0) + \mathcal{O}(n).$$

Allows initial move to any initial tree!

Competitiveness

Relaxed competitive ratio $f(n)$ (similar to asymptotic):

$$A(\sigma, T_0) \leq f(n) \cdot \mathrm{OPT}(\sigma, T_0) + \mathcal{O}(n).$$

Allows initial move to any initial tree!

Related problem: Optimal static binary search tree.

- Given request probabilities for n keys
- Compute the tree minimizing the average depth
- NP-hard!

Competitiveness

Relaxed competitive ratio $f(n)$ (similar to asymptotic):

$$A(\sigma, T_0) \leq f(n) \cdot \mathrm{OPT}(\sigma, T_0) + \mathcal{O}(n).$$

Allows initial move to any initial tree!

Related problem: Optimal static binary search tree.

- Given request probabilities for n keys
- Compute the tree minimizing the average depth
- NP-hard!

$f(n)$-static-competitive:
Compare A not to OPT, but to optimal static tree.

Obvious bounds?

How bad can we be? What is the worst possible competitive ratio?

Obvious bounds?

How bad can we be? What is the worst possible competitive ratio?

$\mathcal{O}(n)$; corresponds to requesting last element of path.

Obvious bounds?

How bad can we be? What is the worst possible competitive ratio?

$\mathcal{O}(n)$; corresponds to requesting last element of path.

What is the best we can hope for?

Obvious bounds?

How bad can we be? What is the worst possible competitive ratio?

$\mathcal{O}(n)$; corresponds to requesting last element of path.

What is the best we can hope for?

$\mathcal{O}(1)$; constant competitive ratio.

Obvious bounds?

How bad can we be? What is the worst possible competitive ratio?

$\mathcal{O}(n)$; corresponds to requesting last element of path.

What is the best we can hope for?

$\mathcal{O}(1)$; constant competitive ratio.

Is there is an obvious $\mathcal{O}(\log n)$-competitive algorithm?

Obvious bounds?

How bad can we be? What is the worst possible competitive ratio?

$\mathcal{O}(n)$; corresponds to requesting last element of path.

What is the best we can hope for?

$\mathcal{O}(1)$; constant competitive ratio.

Is there is an obvious $\mathcal{O}(\log n)$-competitive algorithm?

Yes! Balanced static search trees (AVL, red-black, ...)!

Obvious bounds?

How bad can we be? What is the worst possible competitive ratio?

$\mathcal{O}(n)$; corresponds to requesting last element of path.

What is the best we can hope for?

$\mathcal{O}(1)$; constant competitive ratio.

Is there is an obvious $\mathcal{O}(\log n)$-competitive algorithm?

Yes! Balanced static search trees (AVL, red-black, ...)!

Simple strategies (as for linked lists)?

Obvious bounds?

How bad can we be? What is the worst possible competitive ratio?

$\mathcal{O}(n)$; corresponds to requesting last element of path.

What is the best we can hope for?

$\mathcal{O}(1)$; constant competitive ratio.

Is there is an obvious $\mathcal{O}(\log n)$-competitive algorithm?

Yes! Balanced static search trees (AVL, red-black, ...)!

Simple strategies (as for linked lists)?

RotateOnce, RotateToTop

How good are simple strategies?

How good are simple strategies?

RotateOnce is bad (like Transpose): Start with path, request last two elements alternatingly.

How good are simple strategies?

RotateOnce is bad (like Transpose): Start with path, request last two elements alternatingly.

RotateToTop:

- Any initial tree
- Sequence $(1, \ldots, n)$
- How does the tree look now?

How good are simple strategies?

RotateOnce is bad (like Transpose): Start with path, request last two elements alternatingly.

RotateToTop:

- Any initial tree
- Sequence $(1, \ldots, n)$
- How does the tree look now?

After 1: 1 is at the root! After 2: Left path is 2-1!

How good are simple strategies?

RotateOnce is bad (like Transpose): Start with path, request last two elements alternatingly.

RotateToTop:

- Any initial tree
- Sequence $(1, \ldots, n)$
- How does the tree look now?

After 1: 1 is at the root! After 2: Left path is 2-1!
After n: Left path n-...-1.

How good are simple strategies?

RotateOnce is bad (like Transpose): Start with path, request last two elements alternatingly.

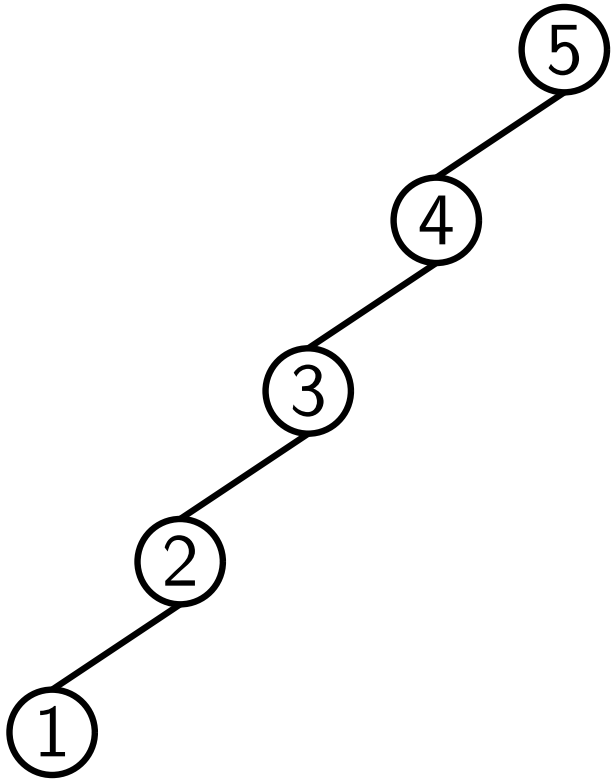
RotateToTop:

- Any initial tree
- Sequence $(1, \ldots, n)$
- How does the tree look now?

After 1: 1 is at the root! After 2: Left path is 2-1!
After n: Left path n-...-1.

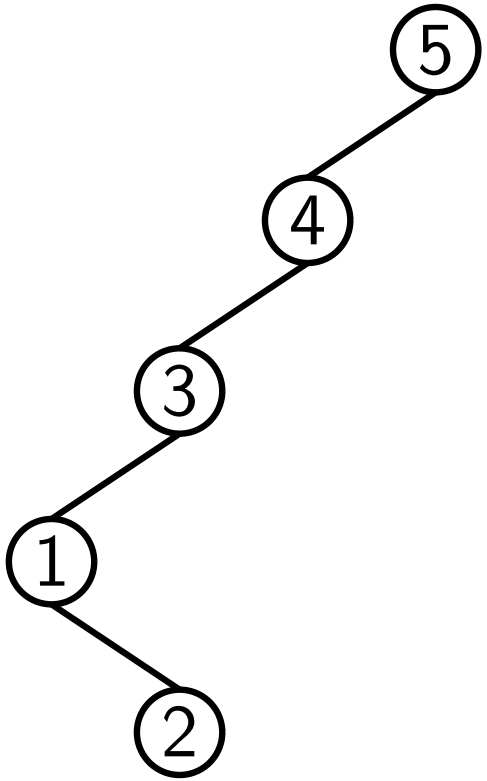
RotateToTop

Request for 1:



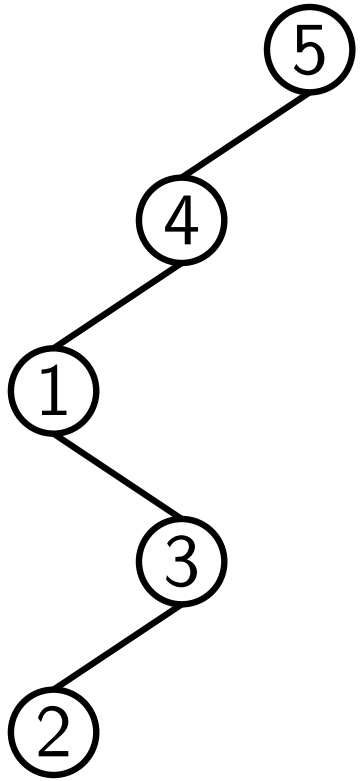
RotateToTop

Request for 1:



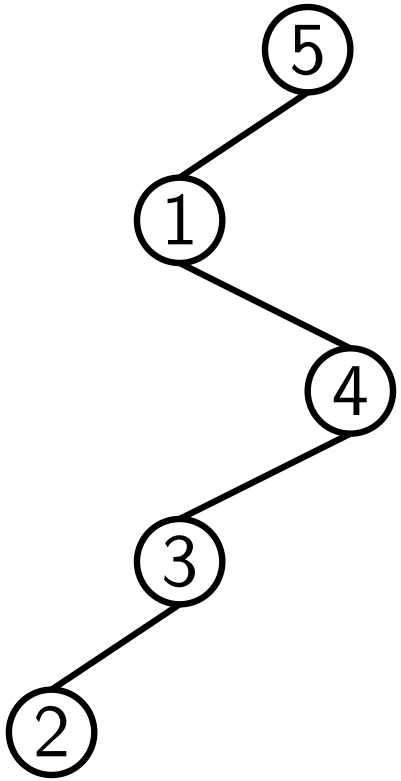
RotateToTop

Request for 1:



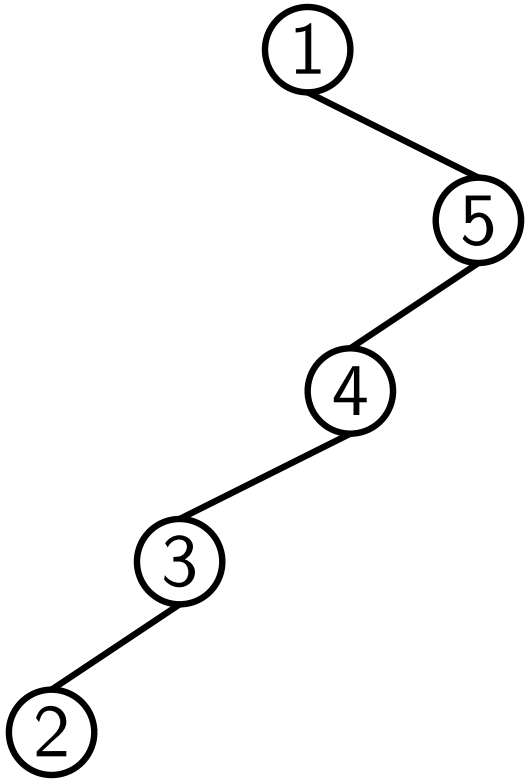
RotateToTop

Request for 1:



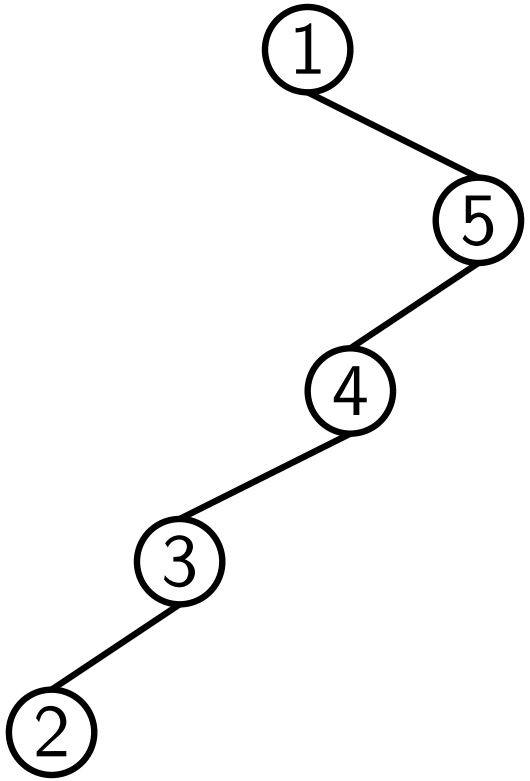
RotateToTop

Request for 1:



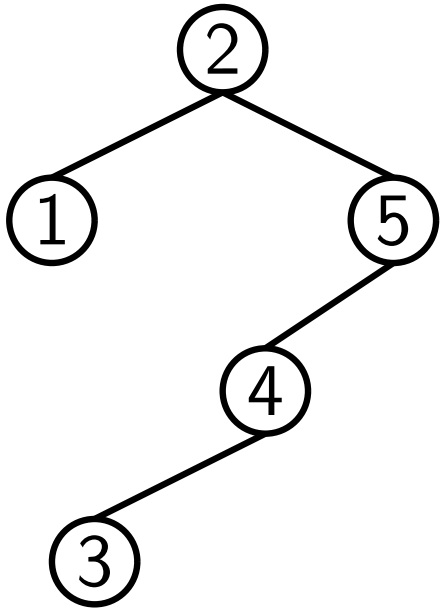
RotateToTop

Request for 1:



RotateToTop

After request for 2:



RotateToTop

Generally, the tree stays a path.

$\Rightarrow \Omega(n^2)$ cost per repetition of $(1, \ldots, n)$.

RotateToTop

Generally, the tree stays a path.
$\Rightarrow \Omega(n^2)$ cost per repetition of $(1, \ldots, n)$.

We can achieve $\mathcal{O}(n)$ per repetition:
First rotate into right path; costs $\mathcal{O}(n)$.
Then $\mathcal{O}(1)$ per request.
$\Rightarrow$ RotateToTop is $\Omega(n)$-competitive.

Better idea?

Can we do better?

- No better (proven) algorithm than $\mathcal{O}(\log n)$
- But a (conjectured) candidate for $\mathcal{O}(1)$: Splay trees

Better idea?

Can we do better?

- No better (proven) algorithm than $\mathcal{O}(\log n)$
- But a (conjectured) candidate for $\mathcal{O}(1)$: Splay trees

Basic idea: RotateToTop with changed rotations

- Consider up to two rotations at once
- If they are in the same direction, rotate upper edge first
- Operations: Zig (last rotation), Zig-Zag, Zig-Zig
- Zig and Zig-Zag are normal bottom-up rotations
- Zig-Zig: Does not destroy balance

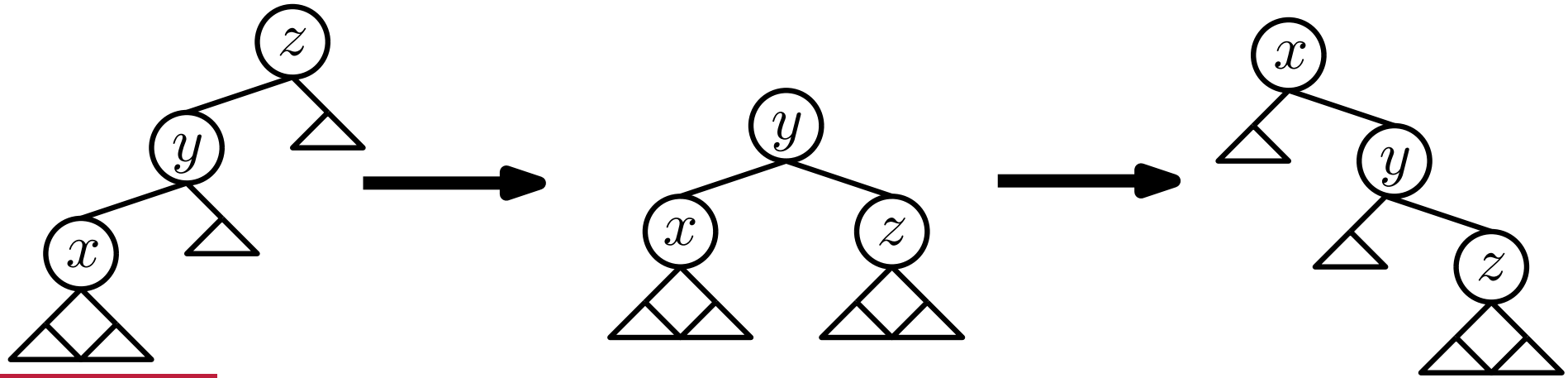
Better idea?

Can we do better?

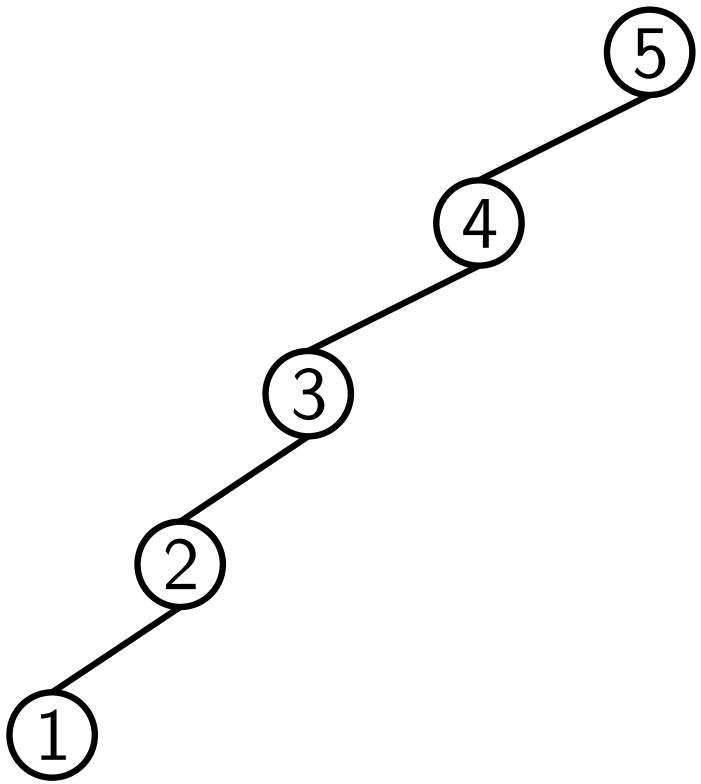
- No better (proven) algorithm than $\mathcal{O}(\log n)$
- But a (conjectured) candidate for $\mathcal{O}(1)$: Splay trees

Basic idea: RotateToTop with changed rotations

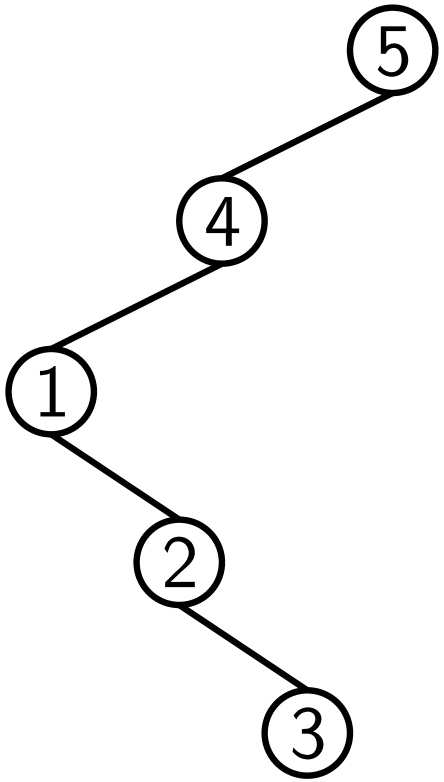
- Consider up to two rotations at once
- If they are in the same direction, rotate upper edge first
- Operations: Zig (last rotation), Zig-Zag, Zig-Zig
- Zig and Zig-Zag are normal bottom-up rotations
- Zig-Zig: Does not destroy balance



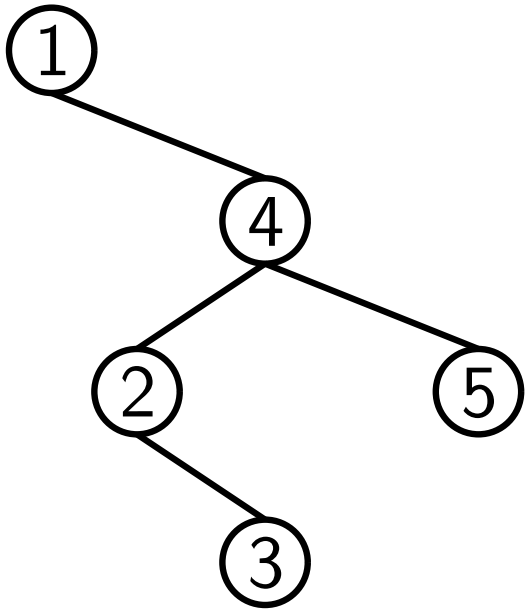
Splaying a path



Splaying a path



Splaying a path



Splay trees

Static optimality theorem:

For any access probabilities, splay trees are within a constant factor of optimal static binary trees.

Splay trees

Static optimality theorem:
For any access probabilities, splay trees are within a constant factor of optimal static binary trees.

Proof is technical, would take too long, but understandable.

`https://www.cs.cmu.edu/~sleator/papers/self-adjusting.pdf`

Splay trees

Static optimality theorem:
For any access probabilities, splay trees are within a constant factor of optimal static binary trees.

Proof is technical, would take too long, but understandable.

`https://www.cs.cmu.edu/~sleator/papers/self-adjusting.pdf`

Key lemma: **Splay Access Lemma**

Splay trees

Static optimality theorem:
For any access probabilities, splay trees are within a constant factor of optimal static binary trees.

Proof is technical, would take too long, but understandable.

`https://www.cs.cmu.edu/~sleator/papers/self-adjusting.pdf`

Key lemma: **Splay Access Lemma**

Number of nodes below node: $n(v)$
Rank of v: $r(v) = \log_2(n(v))$
Tree potential $\Phi(T) = \sum_v r(v)$

Splay trees

Static optimality theorem:
For any access probabilities, splay trees are within a constant factor of optimal static binary trees.

Proof is technical, would take too long, but understandable.

`https://www.cs.cmu.edu/~sleator/papers/self-adjusting.pdf`

Key lemma: **Splay Access Lemma**

Number of nodes below node: $n(v)$
Rank of v: $r(v) = \log_2(n(v))$
Tree potential $\Phi(T) = \sum_v r(v)$

Cost of access/splaying with k rotations:

$$k + \Phi(U) - \Phi(T) \in \mathcal{O}(\log n).$$

Splay trees

Static optimality theorem:
For any access probabilities, splay trees are within a constant factor of optimal static binary trees.

Proof is technical, would take too long, but understandable.

`https://www.cs.cmu.edu/~sleator/papers/self-adjusting.pdf`

Key lemma: **Splay Access Lemma**

Number of nodes below node: $n(v)$
Rank of v: $r(v) = \log_2(n(v))$
Tree potential $\Phi(T) = \sum_v r(v)$ **Careful**: Initial potential > 0!

Cost of access/splaying with k rotations:

$$k + \Phi(U) - \Phi(T) \in \mathcal{O}(\log n).$$

Dynamic Optimality Conjecture

Splay trees are $\mathcal{O}(1)$-competitive in our relaxed sense:
A sequence σ of accesses costs $\mathcal{O}(n + \mathrm{OPT}(\sigma))$.

Dynamic Optimality Conjecture

Splay trees are $\mathcal{O}(1)$-competitive in our relaxed sense:
A sequence σ of accesses costs $\mathcal{O}(n + \mathrm{OPT}(\sigma))$.

Status: open. Works for all special sequence families studied so far. Works experimentally in practice.

Dynamic Optimality Conjecture

Dynamic Optimality Conjecture

Splay trees are $\mathcal{O}(1)$-competitive in our relaxed sense:
A sequence σ of accesses costs $\mathcal{O}(n + \mathrm{OPT}(\sigma))$.

Status: open. Works for all special sequence families studied so far. Works experimentally in practice.

Scanning Theorem

Access sequences like $1, \ldots, n$ or $n, \ldots, 1$ cost $\mathcal{O}(n)$.

Dynamic Optimality Conjecture

Dynamic Optimality Conjecture

Splay trees are $\mathcal{O}(1)$-competitive in our relaxed sense:
A sequence σ of accesses costs $\mathcal{O}(n + \mathrm{OPT}(\sigma))$.

Status: open. Works for all special sequence families studied so far. Works experimentally in practice.

Scanning Theorem

Access sequences like $1, \ldots, n$ or $n, \ldots, 1$ cost $\mathcal{O}(n)$.

Static Finger Theorem

In a sequence σ with many accesses close to a *finger* f:

$$\mathcal{O}\left(n \log n + \sum_{x \in \sigma} \log(|x - f| + 2)\right).$$

Dynamic Finger Theorem

Let x, y be consecutive elements in the sequence σ. Total cost:

$$\mathcal{O}\left(n + \sum_{x,y} \log(|x - y| + 2)\right).$$

Further Properties

Dynamic Finger Theorem

Let x, y be consecutive elements in the sequence σ. Total cost:

$$\mathcal{O}\left(n + \sum_{x,y} \log(|x - y| + 2)\right).$$

Working Set Theorem

Let $t(x)$ be the number of distinct elements accessed since last access to x. Total cost:

$$\mathcal{O}\left(n \log n + \sum_{x} \log(t(x) + 2)\right).$$

Possible application: Sorting

Splay Sort

Possible application: Sorting

Insert:

As for binary trees, but then splay the inserted element.

Splay Sort

Possible application: Sorting

Insert:
As for binary trees, but then splay the inserted element.

Splay Sort:
Insert n elements into splay tree, then walk the tree in-order.

Splay Sort

Possible application: Sorting

Insert:
As for binary trees, but then splay the inserted element.

Splay Sort:
Insert n elements into splay tree, then walk the tree in-order.

In-order traversal: $\mathcal{O}(n)$, **insertion:** $\mathcal{O}(n \log n)$ worst-case

Splay Sort

Possible application: Sorting

Insert:
As for binary trees, but then splay the inserted element.

Splay Sort:
Insert n elements into splay tree, then walk the tree in-order.

In-order traversal: $\mathcal{O}(n)$, **insertion:** $\mathcal{O}(n \log n)$ worst-case

Practice: Slower than QuickSort, MergeSort, etc. for random sequences

Splay Sort

Possible application: Sorting

Insert:
As for binary trees, but then splay the inserted element.

SPLAY SORT:
Insert n elements into splay tree, then walk the tree in-order.

In-order traversal: $\mathcal{O}(n)$, **insertion:** $\mathcal{O}(n \log n)$ worst-case

Practice: Slower than QUICKSORT, MERGESORT, etc. for random sequences

But: Adaptive! Works better with partially-sorted inputs!

Splay Sort

Possible application: Sorting

Insert:
As for binary trees, but then splay the inserted element.

Splay Sort:
Insert n elements into splay tree, then walk the tree in-order.

In-order traversal: $\mathcal{O}(n)$, **insertion:** $\mathcal{O}(n \log n)$ worst-case

Practice: Slower than QuickSort, MergeSort, etc. for random sequences

But: Adaptive! Works better with partially-sorted inputs!

There are specialized, better algorithms for that case though.