

ADVANCED

overall tree
= binary tree of mini trees

mini trees

micro trees

actual nodes

$\frac{1}{4} \lg n$ nodes



DATA STRUCTURES

2

Adaptive Trees

Advanced Data Structures · Summer 2026

Prof. Dr. Sebastian Wild

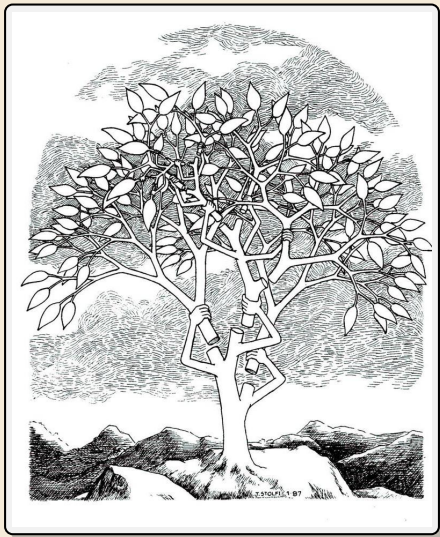
Outline

2 Adaptive Trees

- 2.1 Move-to-Root Heuristic
- 2.2 Move-to-Root Analysis
- 2.3 Splay Trees
- 2.4 Analysis of Splay Trees
- 2.5 Biased Search Trees
- 2.6 Excursion: Online Algorithms
- 2.7 Dynamic Optimality
- 2.8 The Geometric View of BSTs
- 2.9 Deferred Data Structures
- 2.10 Lazy Search Trees

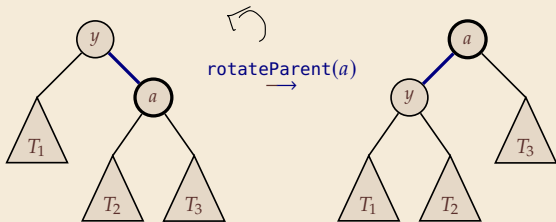
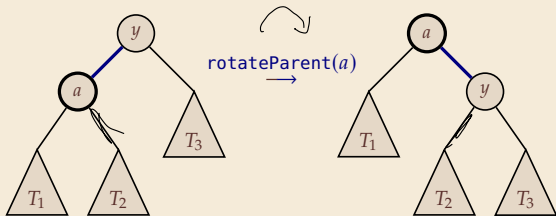
Self-Adjusting Data Structures

Idea: set of rules that makes data structure automatically adapt to use case



2.1 Move-to-Root Heuristic

Rotations



Move-to-root

Upon an access to an element, rotateParent until accessed element is at root!



Allen, Munro: *Self-Organizing Binary Search Trees*, JACM 1978

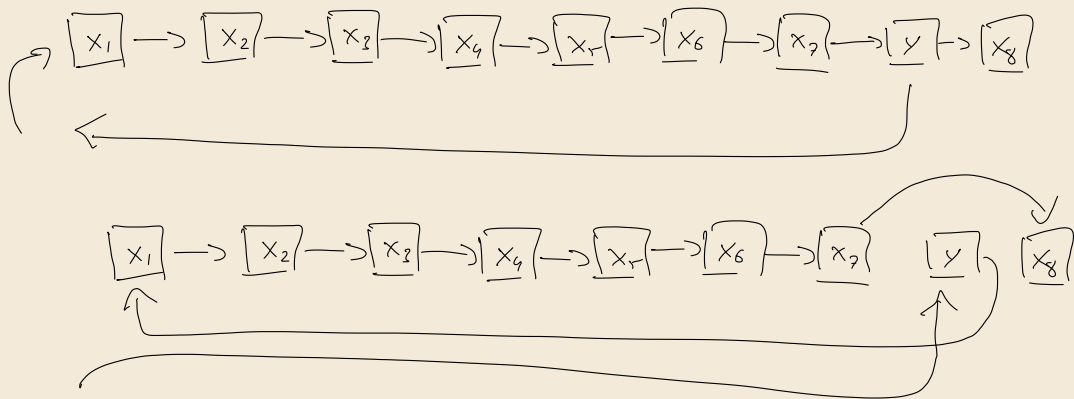
Inspired by **Move-To-Front heuristic** of self-adjusting linked lists

- ▶ cost via MTF is at most 2 times cost of optimal **static ordering**
- ▶ MTF is 2-competitive as an online algorithm
(at most 2 times the cost of *any* algorithm)
- ▶ (both true under “basic cost model” only)
- ▶ with resource augmentation, LRU in paging very successful

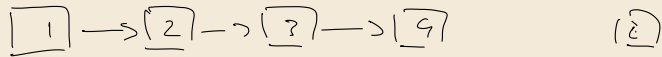
Move-to-front

unsorted linked list

search(y)



extreme case $p_i = 2^{-i}$ $p_n = 2^{-(n-1)}$



expected search cost $\sum_{i=1}^n p_i (i-1) \approx \sum_{i=1}^{\infty} 2^i (i-1) = O(1)$

$$\sum_{i \geq 0} 2^i = \frac{1}{1-2} \quad \left| \frac{d}{dz} \right.$$

$$\sum_{i \geq 0} i 2^{i-1} = +1 \cdot (1-2)^{-2} = \frac{1}{(1-2)^2}$$

Move-to-root

Upon an access to an element, rotateParent until accessed element is at root!



Allen, Munro: *Self-Organizing Binary Search Trees*, JACM 1978

Inspired by **Move-To-Front heuristic** of self-adjusting linked lists

- ▶ cost via MTF is at most 2 times cost of optimal **static ordering**
- ▶ MTF is 2-competitive as an online algorithm
(at most 2 times the cost of *any* algorithm)
- ▶ (both true under “basic cost model” only)
- ▶ with resource augmentation, LRU in paging very successful

Maybe, Move-to-root is similarly successful!

Iid Accesses

Iid Model: Suppose the accesses to keys $[1..n]$ in BST are *i.i.d. randomly* drawn with probabilities p_1, \dots, p_n .


↪ Expected search cost in a BST T given by $\sum_{i=1}^n p_i \cdot \text{depth}_T(\textcircled{i})$

▶ If p_i known up front, can compute **static optimal BST** via dynamic programming

↪ Cost of this optimal tree C_{OPT} can be bounded in terms of the entropy

Lemma 2.1 (Bayer 1975)

$$\mathcal{H} - \lg \mathcal{H} - \lg e + 1 \leq C_{OPT} \leq \mathcal{H} + 1 \quad \text{for} \quad \mathcal{H} = \sum_{i=1}^n p_i \lg(1/p_i)$$

 **Bayer:** *Improved Bounds on the Costs of Optimal and Balanced Binary Search Trees*, M.Sc. Thesis, MIT 1975

Note: C_{OPT} is for successful search / depth of internal nodes

The case of accesses to leaves only, with access to \boxed{j} w/p $q_j, j = 0, \dots, n$ even cleaner:

$$\mathcal{H} \leq \sum_{j=0}^n q_j \cdot (1 + \text{depth}_T(\boxed{j})) \leq \mathcal{H} + 2$$

2.2 Move-to-Root Analysis

Iid Accesses

Iid Model: Suppose the accesses to keys $[1..n]$ in BST are *i.i.d. randomly* drawn with probabilities p_1, \dots, p_n .

Iid Accesses

Iid Model: Suppose the accesses to keys $[1..n]$ in BST are *i.i.d. randomly* drawn with probabilities p_1, \dots, p_n .

↪ Expected search cost in a BST T given by $\sum_{i=1}^n p_i (\text{depth}_T(\textcircled{i}) + 1)$

—

$$(p_i = 2^{-i} \quad \mathcal{H} = \sum 2^i \cdot i)$$

$$0 \leq \mathcal{H} \leq \lg n$$

Iid Accesses

Iid Model: Suppose the accesses to keys $[1..n]$ in BST are *i.i.d. randomly* drawn with probabilities p_1, \dots, p_n .

↪ Expected search cost in a BST T given by $\sum_{i=1}^n p_i \cdot \text{depth}_T(\textcircled{i})$

▶ If p_i known up front, can compute **static optimal BST** via dynamic programming

↪ Cost of this optimal tree C_{OPT} can be bounded in terms of the entropy

Lemma 2.1 (Bayer 1975)

$$\mathcal{H} - \lg \mathcal{H} - \lg e + 1 \leq C_{OPT} \leq \mathcal{H} + 1 \quad \text{for} \quad \mathcal{H} = \sum_{i=1}^n p_i \lg(1/p_i)$$



Bayer: *Improved Bounds on the Costs of Optimal and Balanced Binary Search Trees*, M.Sc. Thesis, MIT 1975

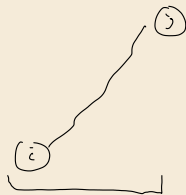
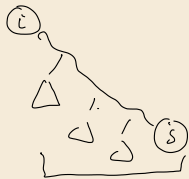
Move-to-Root on iid Accesses

Lemma 2.2 (MTR ancestors)

Let T be a tree maintained by the move-to-root heuristic and $i < j$ two keys.

Then (i) is an **ancestor** of (j) iff the most recent request for i came after the most recent requests of any of $i + 1, \dots, j$.

Similarly, (j) ancestor of (i) iff j requested more recently than any of $i, \dots, j - 1$. ◀



Proof: " \Rightarrow " (i) ancestor of (j) cannot have accessed (MTR) $i+1, \dots, j$ after (i)

" \Leftarrow " accesses to all of i, \dots, j but i last

Move-to-Root on iid Accesses

Lemma 2.2 (MTR ancestors)

Let T be a tree maintained by the move-to-root heuristic and $i < j$ two keys.

Then (i) is an **ancestor** of (j) iff the most recent request for i came after the most recent requests of any of $i + 1, \dots, j$.

Similarly, (j) ancestor of (i) iff j requested more recently than any of $i, \dots, j - 1$. ◀

Recall $A_j^i = [(i) \text{ ancestor of } (j)]$

Corollary 2.3

Under the iid accesses model and MTR heuristic, $\mathbb{P}[A_j^i] = \begin{cases} \frac{p_i}{p_i + \dots + p_j} & i < j \\ \frac{p_i}{p_j + \dots + p_i} & i > j \end{cases}$ ◀

Move-to-Root on iid Accesses – Analysis

Theorem 2.4 (MTR cost)

Always assume: every key accessed at least once

Let C_{MTR} be the expected search cost under the i.i.d. model in a tree T maintained by MTR.

$$C_{MTR} = 1 + 2 \sum_{1 \leq i < j \leq n} \frac{p_i p_j}{p_i + \dots + p_j} \leq 1 + 2 \underbrace{\ln 2}_{1.39} \cdot \mathcal{H}$$

$$\mathcal{H} = \sum_{i=1}^n p_i \lg\left(\frac{1}{p_i}\right) \quad \blacktriangleleft$$

Move-to-Root has close to optimal search cost in the i.i.d. model!

Move-to-Root on iid Accesses – Analysis

Theorem 2.4 (MTR cost)

Let C_{MTR} be the expected search cost under the i.i.d. model in a tree T maintained by MTR.

$$C_{MTR} = 1 + 2 \sum_{1 \leq i < j \leq n} \frac{p_i p_j}{p_i + \dots + p_j} \leq 1 + 2 \ln 2 \cdot \mathcal{H}$$

Move-to-Root has close to optimal search cost in the i.i.d. model!

Proof:

$$C_{MTR} = \sum_{i=1}^n p_i \cdot \left(\overset{\# \text{ comps}}{1 + \text{depth}(i)} \right)$$



Move-to-Root on iid Accesses – Analysis

Theorem 2.4 (MTR cost)

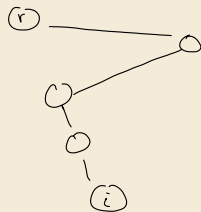
Let C_{MTR} be the expected search cost under the i.i.d. model in a tree T maintained by MTR.

$$C_{MTR} = 1 + 2 \sum_{1 \leq i < j \leq n} \frac{p_i p_j}{p_i + \dots + p_j} \leq 1 + 2 \ln 2 \cdot \mathcal{H}$$

Move-to-Root has close to optimal search cost in the i.i.d. model!

Proof:

$$C_{MTR} = \sum_{i=1}^n p_i \cdot (1 + \text{depth}(\textcircled{i})) = \sum_{i=1}^n p_i \cdot \left(1 + \sum_{j \neq i} \mathbb{E}[A_i^j] \right)$$



Move-to-Root on iid Accesses – Analysis

Theorem 2.4 (MTR cost)

Let C_{MTR} be the expected search cost under the i.i.d. model in a tree T maintained by MTR.

$$C_{MTR} = 1 + 2 \sum_{1 \leq i < j \leq n} \frac{p_i p_j}{p_i + \dots + p_j} \leq 1 + 2 \ln 2 \cdot \mathcal{H} \quad \blacktriangleleft$$

Move-to-Root has close to optimal search cost in the i.i.d. model!

Proof:

$$\begin{aligned} C_{MTR} &= \sum_{i=1}^n p_i \cdot (1 + \text{depth}(\textcircled{i})) = \sum_{i=1}^n p_i \cdot \left(1 + \sum_{j \neq i} \mathbb{E}[A_i^j] \right) \\ &= 1 + \sum_{i=1}^n \sum_{j \neq i} p_i \mathbb{E}[A_i^j] \end{aligned}$$



Move-to-Root on iid Accesses – Analysis

Theorem 2.4 (MTR cost)

Let C_{MTR} be the expected search cost under the i.i.d. model in a tree T maintained by MTR.

$$C_{MTR} = 1 + 2 \sum_{1 \leq i < j \leq n} \frac{p_i p_j}{p_i + \dots + p_j} \leq 1 + 2 \ln 2 \cdot \mathcal{H} \quad \blacktriangleleft$$

Move-to-Root has close to optimal search cost in the i.i.d. model!

Proof:

$$\begin{aligned} C_{MTR} &= \sum_{i=1}^n p_i \cdot (1 + \text{depth}(\textcircled{i})) = \sum_{i=1}^n p_i \cdot \left(1 + \sum_{j \neq i} \mathbb{E}[A_i^j] \right) \\ &= 1 + \sum_{i=1}^n \sum_{j \neq i} p_i \mathbb{E}[A_i^j] \stackrel{\text{Lemma 2}}{=} \sum_{i=1}^n \sum_{j \neq i} \frac{p_j}{p_i + \dots + p_j} \\ &= 1 + 2 \sum_{1 \leq i < j \leq n} \frac{p_i p_j}{p_i + \dots + p_j} \end{aligned}$$

This proves the first part.

Move-to-Root on iid Accesses – Analysis

Proof (cont.):

Mehlhorn's trick: $\delta_{i,j} := \frac{p_i + \cdots + p_j}{p_i} \rightsquigarrow 1 = \delta_{i,i} \leq \delta_{i,i+1} \leq \cdots \leq \delta_{i,n} \leq \frac{1}{p_i}$



Move-to-Root on iid Accesses – Analysis

Proof (cont.):

Mehlhorn's trick: $\delta_{i,j} := \frac{p_i + \dots + p_j}{p_i} \rightsquigarrow 1 = \delta_{i,i} \leq \delta_{i,i+1} \leq \dots \leq \delta_{i,n} \leq \frac{1}{p_i}$

$$C_{MTR} = 1 + 2 \sum_{i=1}^{n-1} p_i \sum_{j=i+1}^n \frac{p_j}{p_i + \dots + p_j}$$



Move-to-Root on iid Accesses – Analysis

Proof (cont.):

$$\text{Mehlhorn's trick: } \delta_{i,j} := \frac{p_i + \dots + p_j}{p_i}$$

$$\rightsquigarrow 1 = \delta_{i,i} \leq \delta_{i,i+1} \leq \dots \leq \delta_{i,n} \leq \frac{1}{p_i}$$

$$\begin{aligned} C_{MTR} &= 1 + 2 \sum_{i=1}^{n-1} p_i \sum_{j=i+1}^n \frac{p_j}{p_i + \dots + p_j} \\ &= 1 + 2 \sum_{i=1}^{n-1} p_i \sum_{j=i+1}^n \frac{\delta_{i,j} - \delta_{i,j-1}}{\delta_{i,j}} \end{aligned}$$

$$\begin{aligned} & \frac{\delta_{i,j} - \delta_{i,j-1}}{\delta_{i,j}} \\ &= \frac{p_i + \dots + p_{j-1} + p_j - (p_i + \dots + p_{j-1})}{p_i + \dots + p_j} \\ &= \frac{p_j}{p_i + \dots + p_j} \end{aligned}$$

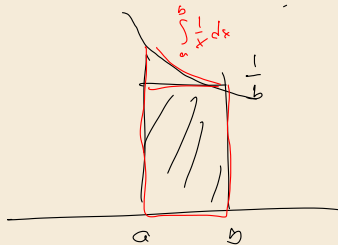
Move-to-Root on iid Accesses – Analysis

Proof (cont.):

Mehlhorn's trick: $\delta_{i,j} := \frac{p_i + \dots + p_j}{p_i} \rightsquigarrow 1 = \delta_{i,i} \leq \delta_{i,i+1} \leq \dots \leq \delta_{i,n} \leq \frac{1}{p_i}$

$$\begin{aligned} C_{MTR} &= 1 + 2 \sum_{i=1}^{n-1} p_i \sum_{j=i+1}^n \frac{p_j}{p_i + \dots + p_j} \\ &= 1 + 2 \sum_{i=1}^{n-1} p_i \sum_{j=i+1}^n \frac{\delta_{i,j} - \delta_{i,j-1}}{\delta_{i,j}} \end{aligned}$$

$$a \leq b \implies (b-a) \cdot \frac{1}{b} = \int_a^b \frac{1}{b} dx < \int_a^b \frac{1}{x} dx$$



Move-to-Root on iid Accesses – Analysis

Proof (cont.):

Mehlhorn's trick: $\delta_{i,j} := \frac{p_i + \dots + p_j}{p_i} \rightsquigarrow 1 = \delta_{i,i} \leq \delta_{i,i+1} \leq \dots \leq \delta_{i,n} \leq \frac{1}{p_i}$

$$\begin{aligned} C_{MTR} &= 1 + 2 \sum_{i=1}^{n-1} p_i \sum_{j=i+1}^n \frac{p_j}{p_i + \dots + p_j} \\ &= 1 + 2 \sum_{i=1}^{n-1} p_i \sum_{j=i+1}^n \frac{\delta_{i,j} - \delta_{i,j-1}}{\delta_{i,j}} \\ &< 1 + 2 \sum_{i=1}^{n-1} p_i \sum_{j=i+1}^n \int_{x=\delta_{i,j-1}}^{\delta_{i,j}} \frac{1}{x} dx \end{aligned}$$

$$a \leq b \implies (b - a) \cdot \frac{1}{b} = \int_a^b \frac{1}{b} dx < \int_a^b \frac{1}{x} dx$$

Move-to-Root on iid Accesses – Analysis

Proof (cont.):

Mehlhorn's trick: $\delta_{i,j} := \frac{p_i + \dots + p_j}{p_i} \rightsquigarrow 1 = \delta_{i,i} \leq \delta_{i,i+1} \leq \dots \leq \delta_{i,n} \leq \frac{1}{p_i}$

$$C_{MTR} = 1 + 2 \sum_{i=1}^{n-1} p_i \sum_{j=i+1}^n \frac{p_j}{p_i + \dots + p_j}$$

$$= 1 + 2 \sum_{i=1}^{n-1} p_i \sum_{j=i+1}^n \frac{\delta_{i,j} - \delta_{i,j-1}}{\delta_{i,j}}$$

$$a \leq b \implies (b-a) \cdot \frac{1}{b} = \int_a^b \frac{1}{b} dx < \int_a^b \frac{1}{x} dx$$

$$< 1 + 2 \sum_{i=1}^{n-1} p_i \sum_{j=i+1}^n \int_{x=\delta_{i,j-1}}^{\delta_{i,j}} \frac{1}{x} dx$$

$$= 1 + 2 \sum_{i=1}^{n-1} p_i \int_{x=\delta_{i,i}}^{\delta_{i,n}} \frac{1}{x} dx$$

$$\int_{x=a}^b \frac{1}{x} dx = \ln(b) - \ln(a) = \ln(b/a)$$

Move-to-Root on iid Accesses – Analysis

Proof (cont.):

Mehlhorn's trick: $\delta_{i,j} := \frac{p_i + \dots + p_j}{p_i}$

$$\rightsquigarrow 1 = \delta_{i,i} \leq \delta_{i,i+1} \leq \dots \leq \delta_{i,n} \leq \frac{1}{p_i}$$

$$C_{MTR} = 1 + 2 \sum_{i=1}^{n-1} p_i \sum_{j=i+1}^n \frac{p_j}{p_i + \dots + p_j}$$

$$= 1 + 2 \sum_{i=1}^{n-1} p_i \sum_{j=i+1}^n \frac{\delta_{i,j} - \delta_{i,j-1}}{\delta_{i,j}}$$

$$a \leq b \implies (b-a) \cdot \frac{1}{b} = \int_a^b \frac{1}{b} dx < \int_a^b \frac{1}{x} dx$$

$$< 1 + 2 \sum_{i=1}^{n-1} p_i \sum_{j=i+1}^n \int_{x=\delta_{i,j-1}}^{\delta_{i,j}} \frac{1}{x} dx$$

$$= 1 + 2 \sum_{i=1}^{n-1} p_i \int_{x=\delta_{i,i}}^{\delta_{i,n}} \frac{1}{x} dx$$

$$\int_{x=a}^b \frac{1}{x} dx = \ln(b) - \ln(a) = \ln(b/a)$$

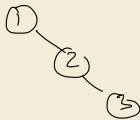
$$= 1 + 2 \sum_{i=1}^{n-1} p_i \ln(\delta_{i,n}/\delta_{i,i}) \leq 1 + 2 \ln 2 \sum_{i=1}^{n-1} p_i \lg(1/p_i) = 1 + 2 \ln 2 \cdot \mathcal{H}$$

Unbalanced BSTs under i.i.d. model

Interestingly, the same cost result from unbalanced BSTs!

- ▶ Suppose we repeatedly draw x i.i.d. from $[1..n]$ w/p p_1, \dots, p_n
- ▶ We insert x into initially empty unbalanced BST T
- ▶ **If x is already contained in T , the insertion has no effect**
- ▶ Repeat until saturation (i. e., until we have seen every value $x \in [1..n]$)

$n=3$ 1 2 2 1 3 2 1 2
 x x 3 2 1 2
 - -

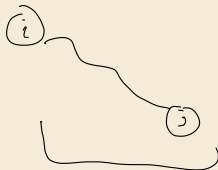


Unbalanced BSTs under i.i.d. model

Interestingly, the same cost result from unbalanced BSTs!

- ▶ Suppose we repeatedly draw x i.i.d. from $[1..n]$ w/p p_1, \dots, p_n
- ▶ We insert x into initially empty unbalanced BST T
- ▶ **If x is already contained in T , the insertion has no effect**
- ▶ Repeat until saturation (i. e., until we have seen every value $x \in [1..n]$)

Since insertions are i.i.d., probability that i is inserted first among $[i..j]$ is $\frac{p_i}{p_i + \dots + p_j}$



Unbalanced BSTs under i.i.d. model

Interestingly, the same cost result from unbalanced BSTs!

- ▶ Suppose we repeatedly draw x i.i.d. from $[1..n]$ w/p p_1, \dots, p_n
- ▶ We insert x into initially empty unbalanced BST T
- ▶ **If x is already contained in T , the insertion has no effect**
- ▶ Repeat until saturation (i. e., until we have seen every value $x \in [1..n]$)

Since insertions are i.i.d., probability that i is inserted first among $[i..j]$ is $\frac{p_i}{p_i + \dots + p_j}$

$$\rightsquigarrow \mathbb{P}[A_j^i] = \begin{cases} \frac{p_i}{p_i + \dots + p_j} & i < j \\ \frac{p_i}{p_j + \dots + p_i} & i > j \end{cases}$$

Unbalanced BSTs under i.i.d. model

Interestingly, the same cost result from unbalanced BSTs!

- ▶ Suppose we repeatedly draw x i.i.d. from $[1..n]$ w/p p_1, \dots, p_n
- ▶ We insert x into initially empty unbalanced BST T
- ▶ **If x is already contained in T , the insertion has no effect**
- ▶ Repeat until saturation (i. e., until we have seen every value $x \in [1..n]$)

Since insertions are i.i.d., probability that i is inserted first among $[i..j]$ is $\frac{p_i}{p_i + \dots + p_j}$

$$\rightsquigarrow \mathbb{P}[A_j^i] = \begin{cases} \frac{p_i}{p_i + \dots + p_j} & i < j \\ \frac{p_i}{p_j + \dots + p_i} & i > j \end{cases}$$

$$\rightsquigarrow C_T = 1 + 2 \sum_{1 \leq i < j \leq n} \frac{p_i p_j}{p_i + \dots + p_j} \leq 1 + 2 \ln 2 \cdot \mathcal{H}$$

Unbalanced BSTs under i.i.d. model

Interestingly, the same cost result from unbalanced BSTs!

- ▶ Suppose we repeatedly draw x i.i.d. from $[1..n]$ w/p p_1, \dots, p_n
- ▶ We insert x into initially empty unbalanced BST T
- ▶ **If x is already contained in T , the insertion has no effect**
- ▶ Repeat until saturation (i. e., until we have seen every value $x \in [1..n]$)

Since insertions are i.i.d., probability that i is inserted first among $[i..j]$ is $\frac{p_i}{p_i + \dots + p_j}$

$$\rightsquigarrow \mathbb{P}[A_j^i] = \begin{cases} \frac{p_i}{p_i + \dots + p_j} & i < j \\ \frac{p_i}{p_j + \dots + p_i} & i > j \end{cases}$$

$$\rightsquigarrow C_T = 1 + 2 \sum_{1 \leq i < j \leq n} \frac{p_i p_j}{p_i + \dots + p_j} \leq 1 + 2 \ln 2 \cdot \mathcal{H}$$

Both unbalanced BST and Move-to-Root behave well under i.i.d. model.

Move-to-Root's Blind Spot

Problem: *Can't rely on accesses to be random.*

Theorem 2.5 (Move-to-Root Bad Case)

There exists a sequence of n requests, such that starting from any tree T , the amortized access cost over the course of the n requests is $\Omega(n)$.

Proof:


see exercises




2.3 Splay Trees


Splay Trees


Splay trees by now the most widely known self-adjusting data structure.

 reasonably simple and fast

 simple to implement

 remarkable theoretical properties (*access theorems*) \rightsquigarrow later

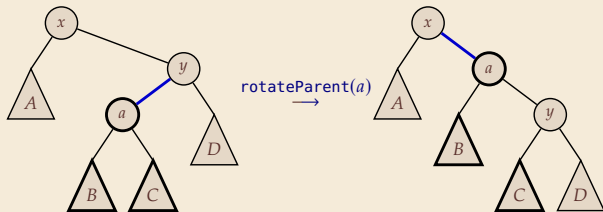
 many writes and pointer chases

 Sleator, Tarjan: *Self-Adjusting Binary Search Trees*, JACM 1985

Move-to-root reloaded

What's wrong in Move-to-front?

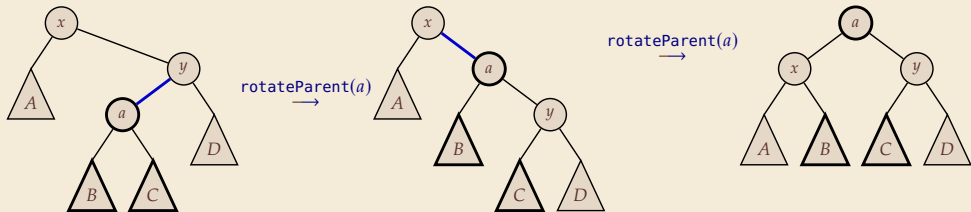
Let's look at the possible cases of parent and grandparent



Move-to-root reloaded

What's wrong in Move-to-front?

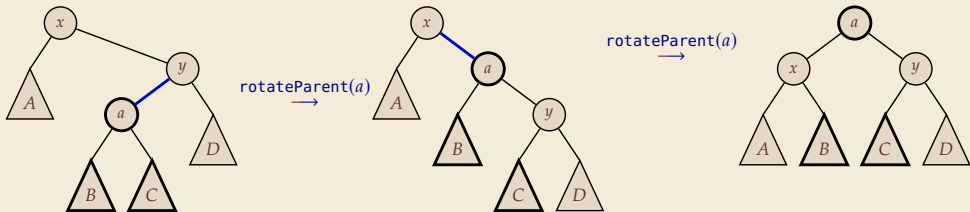
Let's look at the possible cases of parent and grandparent



Move-to-root reloaded

What's wrong in Move-to-front?

Let's look at the possible cases of parent and grandparent

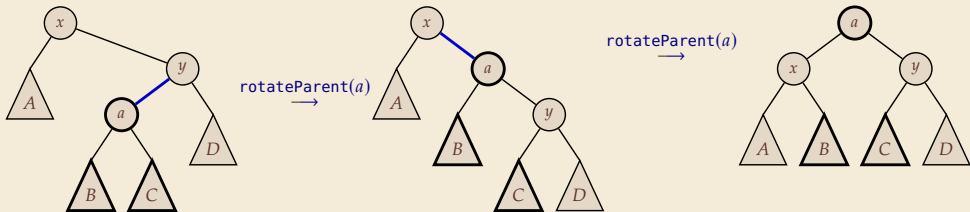


Using simple `rotateParent` calls, the subtree we came from (B or C) is lifted up!

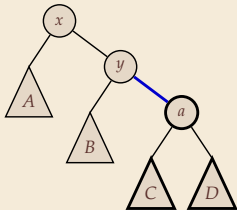
Move-to-root reloaded

What's wrong in Move-to-front?

Let's look at the possible cases of parent and grandparent



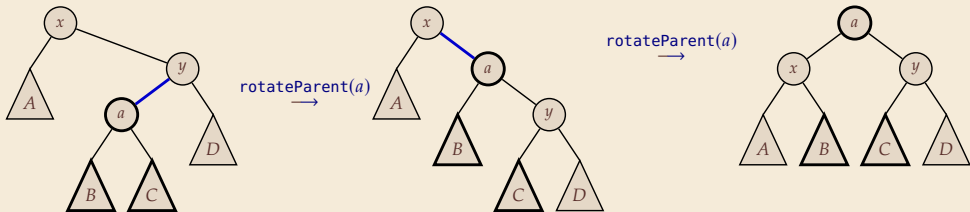
Using simple rotateParent calls, the subtree we came from (B or C) is lifted up!



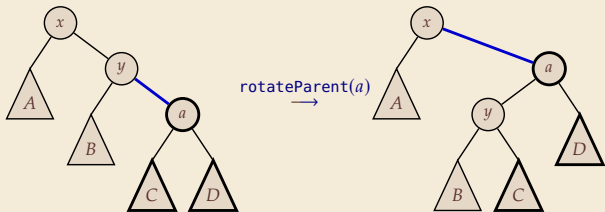
Move-to-root reloaded

What's wrong in Move-to-front?

Let's look at the possible cases of parent and grandparent



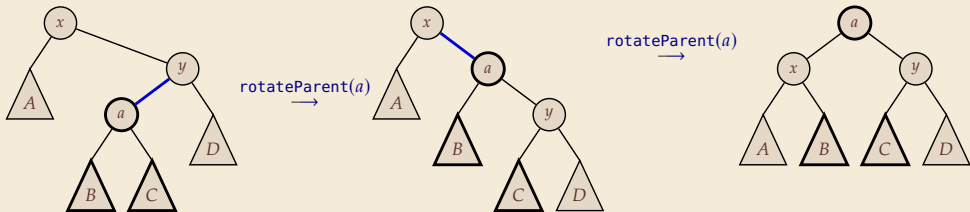
Using simple `rotateParent` calls, the subtree we came from (`B` or `C`) is lifted up!



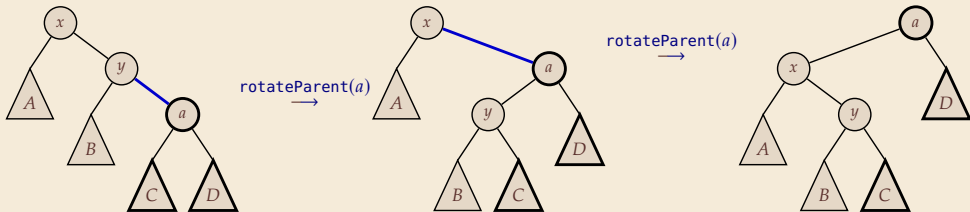
Move-to-root reloaded

What's wrong in Move-to-front?

Let's look at the possible cases of parent and grandparent



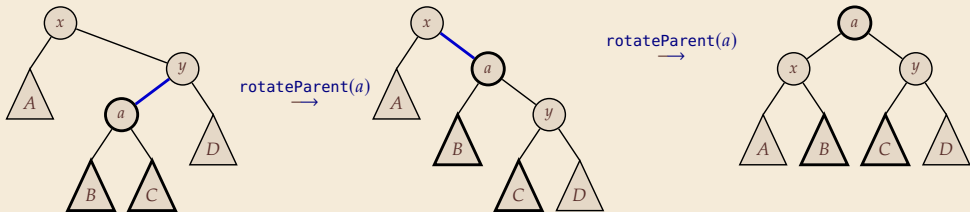
Using simple `rotateParent` calls, the subtree we came from (`B` or `C`) is lifted up!



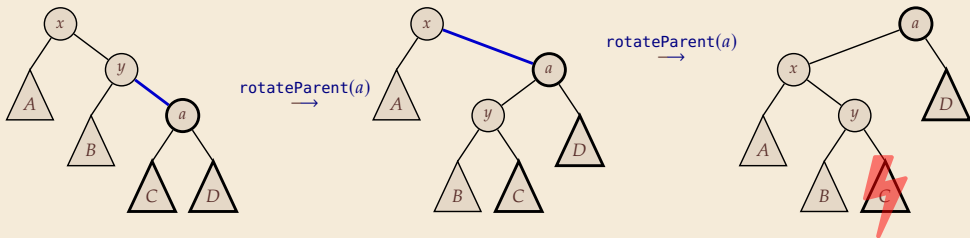
Move-to-root reloaded

What's wrong in Move-to-front?

Let's look at the possible cases of parent and grandparent



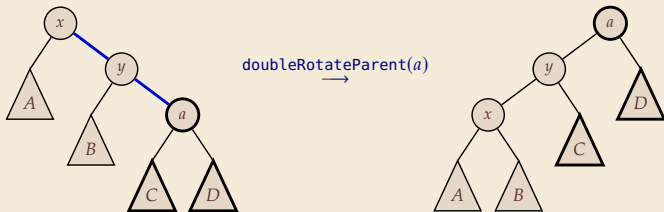
Using simple rotateParent calls, the subtree we came from (B or C) is lifted up!



Here, simple rotateParent calls do **not** lift C up!

The Fix

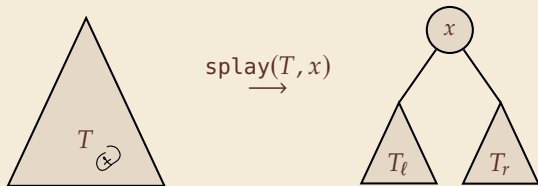
In the case that both parents are right children, do a *double rotation*



Now both C and D are lifted up!

Splay

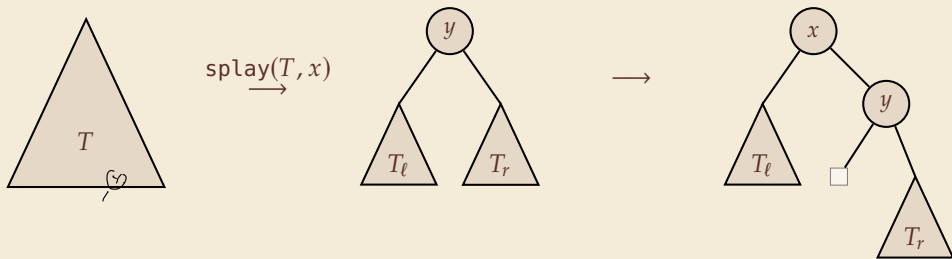
Aggressive restructuring allows to base all operations on single primitive: splay



```
1 def splay(T, x):
2   while x ≠ T.root
3     p = parent(x)
4     if p == T.root
5       rotateParent(x) # Zig Case
6   else:
7     g = parent(p)
8     if p == g.left ∧ x == p.left or p == g.right ∧ x == p.right:
9       doubleRotateParent(x) # Zig-Zig Case
10    else:
11      rotateParent(x) # Zig-Zag Case
12      rotateParent(x)
```

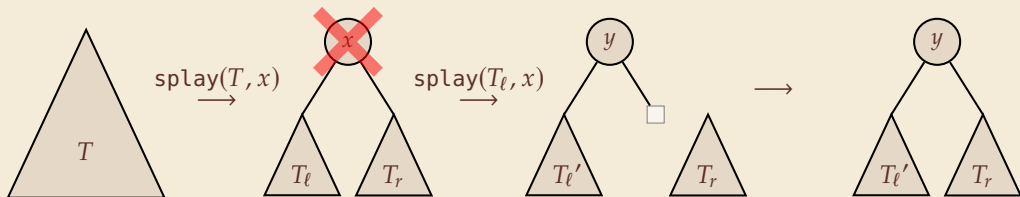
Splay – Insert

1. $\text{splay}(x, T) \rightsquigarrow$ root y and $T_\ell \leq y \leq T_r$ (key order)
2. If $x = y$ (key already stored) return.
3. Otherwise, w.l.o.g. $x < y$, can simply make x new root.



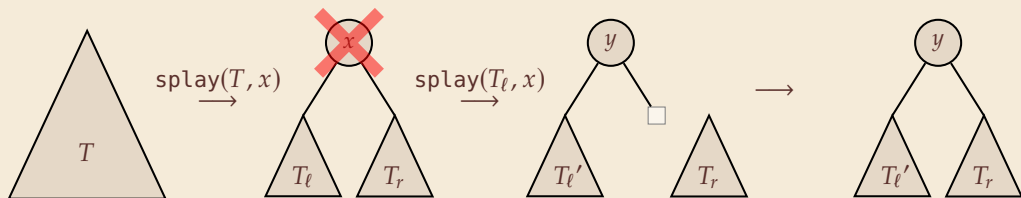
Splay – Delete

1. $\text{splay}(x, T) \rightsquigarrow$ root x (or not part of tree and we're done)
2. Remove x , left with T_ℓ and T_r
3. $\text{splay}(x, T_\ell) \rightsquigarrow$ root $y = \max(T_\ell)$ and $y.\text{right} = \text{null}$
4. $y.\text{right} := T_r$



Splay – Delete

1. $\text{splay}(x, T) \rightsquigarrow$ root x (or not part of tree and we're done)
2. Remove x , left with T_ℓ and T_r
3. $\text{splay}(x, T_\ell) \rightsquigarrow$ root $y = \max(T_\ell)$ and $y.\text{right} = \text{null}$
4. $y.\text{right} := T_r$



Note: We've effectively built **join** \rightsquigarrow naturally supported via splay.

split even more directly supported via splay: $\text{split}(T, x): \text{splay}(T, x), \text{return } (x.\text{left}, x)$

<https://www.cs.usfca.edu/~galles/visualization/SplayTree.html>

2.4 Analysis of Splay Trees

Rotation Cost

Note: In all operations, we follow a search path and splay.

(For delete twice)

splay walks back up search path

↪ use $k = \Theta(d)$ rotations for splaying node at depth d .

↪ cost is dominated by $k = \underline{\text{\#rotations in splay}}$ calls.

Rotation Cost

Note: In all operations, we follow a search path and splay.

(For delete twice)

splay walks back up search path

↪ use $k = \Theta(d)$ rotations for splaying node at depth d .

↪ cost is dominated by $k = \text{\#rotations in splay}$ calls.

Clearly, individual operations could be expensive (use $k = \Omega(n)$ rotations).

↪ Can only hope for an *amortized* bound on k

Idea of amortized analysis: sequence of operations $op_1, op_2, op_3, \dots, op_m$

$c_i =$ actual cost of op_i (chaotic) $\sum_{i=1}^m c_i$

$a_i =$ amortized cost of op_i $= c_i + \Delta \Phi_i \leq \underline{B(n)}$

$\Phi =$ 'potential' state of ds $\rightarrow \mathbb{R}$ $\Phi_i - \Phi_{i-1}$

Φ_i = potential after op_i Φ_0 initial potential

$$\sum_{i=1}^m a_i = \sum_{i=1}^m (c_i + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^m c_i + \Phi_m - \Phi_0$$

$$\sum_{i=1}^m c_i = \sum_{i=1}^m \underbrace{a_i}_{\leq B(u)} + \Phi_0 - \Phi_m \leq m \cdot B(u) + \Phi_0 - \Phi_m$$

Rotation Cost

Note: In all operations, we follow a search path and splay.
(For delete twice)

splay walks back up search path

↪ use $k = \Theta(d)$ rotations for splaying node at depth d .

↪ cost is dominated by $k = \text{\#rotations in splay}$ calls.

Clearly, individual operations could be expensive (use $k = \Omega(n)$ rotations).

↪ Can only hope for an *amortized* bound on k

We define a potential function $\Phi = \Phi(T)$ such that

Goal: If $\text{splay}(x)$ uses k rotations, then

$$k + \Phi' - \Phi \leq 1 + 3 \cdot (r'(x) - r(x))$$

$\Delta\Phi = \text{change in potential}$

change in "rank" of x

↪ Released potential can pay for actual cost k

Splay Tree Potential

We choose a general method that will give several cool results.

- ▶ For that, we allow each node x to have a *weight* $w(x)$

Splay Tree Potential

We choose a general method that will give several cool results.

► For that, we allow each node x to have a *weight* $w(x)$

► The *size* of a node x is defined as $s(x) := \sum_{\substack{v \in T: \\ x \text{ ancestor of } v}} w(v) \geq w(x)$



Splay Tree Potential

We choose a general method that will give several cool results.

► For that, we allow each node x to have a *weight* $w(x)$

► The *size* of a node x is defined as $s(x) := \sum_{\substack{v \in T: \\ x \text{ ancestor of } v}} w(v)$

► The *rank* of node x is $r(x) := \lg(s(x))$

Splay Tree Potential

We choose a general method that will give several cool results.

► For that, we allow each node x to have a *weight* $w(x)$

► The *size* of a node x is defined as $s(x) := \sum_{\substack{v \in T: \\ x \text{ ancestor of } v}} w(v)$

► The *rank* of node x is $r(x) := \lg(s(x))$

► The *potential* of a tree is $\Phi = \sum_{v \in T} r(v)$

► We always denote with Φ, r, s the quantities **before** an operation and with Φ', r', s' the quantities **after** an operation

The Access Lemma

```
1 def splay(T, x):
2   while x ≠ T.root
3     p = parent(x)
4     if p == T.root
5       rotateParent(x) # Zig Case
6     else:
7       g = parent(p)
8       if p == g.left ∧ x == p.left or p == g.right ∧ x == p.right:
9         doubleRotateParent(x) # Zig-Zig Case
10      else:
11        rotateParent(x) # Zig-Zag Case
12        rotateParent(x)
```

$$\Phi = \sum_{v \in T} r(v)$$

$$r(x) := \lg(s(x))$$

$$s(x) := \sum_{\substack{v \in T: \\ x \text{ ancestor of } v}} w(v)$$

Lemma 2.6 (Access Lemma)

For any weakly positive node weights w , the amortized cost of one splay step at x

- (a) for a **Zig** is $\leq 1 + 3(r'(x) - r(x))$
- (b) for a **Zig-Zig** is $\leq 3(r'(x) - r(x))$
- (c) for a **Zig-Zag** is $\leq 3(r'(x) - r(x))$

Access Lemma – Proof

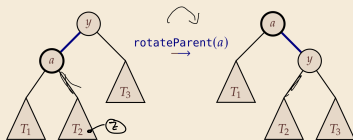
(a) for a Zig is $\leq 1 + 3(r'(x) - r(x))$

$$a = x$$

$$\Phi = \sum_{v \in T} r(v)$$

$$r(x) := \lg(s(x))$$

$$s(x) := \sum_{\substack{v \in T: \\ x \text{ ancestor of } v}} w(v)$$



only nodes outside T_1, T_2, T_3 change rank

$$1 + \Phi' - \Phi$$

↑
real cost

$$= 1 + r'(a) + r'(y) - r(a) - r(y)$$

$$\underbrace{\hspace{15em}}$$

$$\leq 1 + r'(a) - r(a)$$

$$\leq 1 + 3(r'(a) - r(a))$$

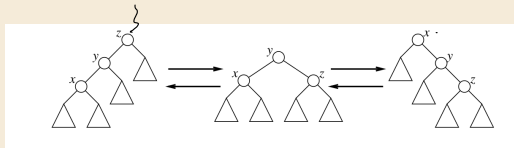
Access Lemma – Proof

(b) for a Zig-Zig is $\leq 3(r'(x) - r(x))$

$$\Phi = \sum_{v \in T} r(v)$$

$$r(x) := \lg(s(x))$$

$$s(x) := \sum_{\substack{v \in T: \\ x \text{ ancestor of } v}} w(v)$$



$$2 + \Phi' - \Phi$$

$$= 2 + \underbrace{r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)}_{=}$$

$$\leq 2 + r'(x) + r'(z) - 2r(x)$$

$$r'(x) \geq r'(y)$$

$$r(y) \geq r(x)$$

$$\leq 3(r'(x) - r(x))$$

to show $2r'(x) - r(x) - r'(z) \geq 2$

$$\lg x + \lg y \quad x + y \leq 1$$

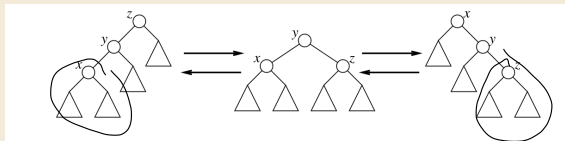
maximized $x = y = \frac{1}{2}$

$$\lg x + \lg y \leq -2$$

$$r(x) + r'(z) - 2r'(x)$$

$$= \lg\left(\frac{s(x)}{s'(x)}\right) + \lg\left(\frac{s'(z)}{s'(x)}\right) \leq -2$$

$$\Rightarrow 2 + \underline{\Phi} - \underline{\Phi} \leq 3(r'(x) - r(x))$$



$s'(x)$ = entire subtree

subtrees of x before

z after

disjoint

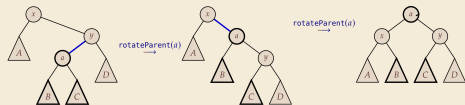
$$s(x) + s'(z) \leq s'(x)$$

Access Lemma - Proof

$$\Phi = \sum_{v \in T} r(v)$$

$$r(x) := \lg(s(x))$$

$$s(x) := \sum_{\substack{v \in T: \\ x \text{ ancestor of } v}} w(v)$$



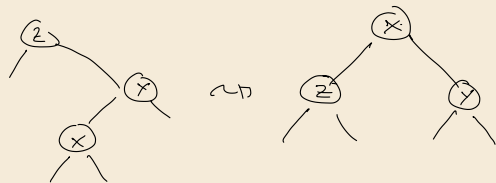
(c) for a Zig-Zag is $\leq 3(r'(x) - r(x))$

$$2 + \Phi' - \Phi = 2 + \underbrace{r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)}$$

$$\leq 2 + r'(y) + r'(z) - 2r(x)$$

$$\stackrel{!}{\leq} 2(r'(x) - r(x))$$

to show $2r'(x) - r'(y) - r'(z) \geq 2$



$$r(x) \leq r(y)$$

$$s'(y) + s'(z) \leq s'(x)$$

$$\lg\left(\frac{s'(x)}{s'(y)}\right) + \lg\left(\frac{s'(z)}{s'(x)}\right) \leq -2$$

Splay Cost

Corollary 2.7

The amortized cost to splay a tree with root t at node x is at most

$$3(r(t) - r(x)) + 1 = O\left(\log\left(\frac{s(t)}{s(x)}\right)\right)$$

$r(t)$
 $r'(x)$



Lemma 2.6 (Access Lemma)

For any weakly positive node weights w , the amortized cost of one splay step at x

- (a) for a **Zig** is $\leq 1 + 3(r'(x) - r(x))$
- (b) for a **Zig-Zig** is $\leq 3(r'(x) - r(x))$
- (c) for a **Zig-Zag** is $\leq 3(r'(x) - r(x))$

Splay Cost

Corollary 2.7

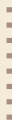
The amortized cost to splay a tree with root t at node x is at most

$$3(r(t) - r(x)) + 1 = O\left(\log\left(\frac{s(t)}{s(x)}\right)\right)$$

Proof:

Splay does rotations involving the same node x until it is the root.

- ↪ The rank differences telescope.
- ↪ Result follow from Lemma 2.6.



Splay Cost

Corollary 2.7

The amortized cost to splay a tree with root t at node x is at most

$$3(r(t) - r(x)) + 1 = O\left(\log\left(\frac{s(t)}{s(x)}\right)\right)$$

Proof:

Splay does rotations involving the same node x until it is the root.

- ↪ The rank differences telescope.
- ↪ Result follow from Lemma 2.6.

Lemma 2.8 (Potential drop)

If weights $w(x)$ are fixed, the potential drop over a sequence of m splay operations is

$$\Phi_0 - \Phi_m \leq \sum_{x \in T} \lg\left(\frac{W}{w(x)}\right) \quad \text{where} \quad W = \sum_{x \in T} w(x)$$

Splay Cost

Corollary 2.7

The amortized cost to splay a tree with root t at node x is at most

$$3(r(t) - r(x)) + 1 = O\left(\log\left(\frac{s(t)}{s(x)}\right)\right)$$

Proof:

Splay does rotations involving the same node x until it is the root.

- ↪ The rank differences telescope.
- ↪ Result follow from Lemma 2.6.

Lemma 2.8 (Potential drop)

If weights $w(x)$ are fixed, the potential drop over a sequence of m splay operations is

$$\Phi_0 - \Phi_m \leq \sum_{x \in T} \lg\left(\frac{W}{w(x)}\right) \quad \text{where} \quad W = \underbrace{\sum_{x \in T} w(x)}$$

Proof:

Always have $\Phi \leq n \lg W$ and $\Phi \geq \sum_{x \in T} \lg w(x)$ (by definition)

So $\Phi_m \leq n \lg W$ and $\Phi_0 \geq \sum \lg w(x)$.

Splay Tree – Results

Theorem 2.9 (Balance Theorem)

Consider a splay tree containing n keys and an arbitrary sequence of m accesses to these keys. The total access time is $O(n \log n + m \log n)$. ◀

Proof: choose $w(x) = \frac{1}{n}$ $W = 1$

\Rightarrow

any operation has amortized cost

$$O\left(\log\left(\frac{1}{1/n}\right)\right) = O(\log n)$$

total cost of m operations = m + $\Phi_m - \Phi_0$

$\underbrace{\hspace{2cm}}_{O(n \log n)}$

Corollary 2.7

The amortized cost to splay a tree with root t at node x is at most

$$3(r(t) - r(x)) + 1 = O\left(\log\left(\frac{s(t)}{s(x)}\right)\right)$$

Splay Tree – Results [2]

Fix an access sequence $X = x_1, \dots, x_m$ on the n nodes in splay tree T , and denote by $p_i = |X|_i/m$ the relative frequency of i in X .

Theorem 2.10 (Static Optimality)

Assume access sequence X contains all n nodes (i. e., $p_i > 0$ for all i).

A splay tree serving access X incurs total cost $O(m(\mathcal{H} + 1))$ for $\mathcal{H} = \sum_{i=1}^n p_i \lg(1/p_i)$. ◀

(essentially optimal (constant factors)
& what MTR achieved)

Proof: $w(i) = p_i \Rightarrow W = 1$

amortized cost $\odot O(\lg(\frac{1}{p_i}))$

$$\sum \text{cost}(x_i) \leq 3 \sum_{i=1}^n |X|_i \lg\left(\frac{1}{p_i}\right)$$

$$= 3m \cdot \sum_{i=1}^n p_i \lg\left(\frac{1}{p_i}\right) = 3m \mathcal{H}$$

Corollary 2.7

The amortized cost to splay a tree with root t at node x is at most

$$3(r(t) - r(x)) + 1 = O\left(\log\left(\frac{s(t)}{s(x)}\right)\right)$$

$$\Phi_m - \Phi_0 \leq \sum_{i=1}^n \lg\left(\frac{1}{p_i}\right) \leq m \mathcal{H}$$

$$m \geq n$$

Splay Tree – Results [3]

Theorem 2.11 (Static Finger Theorem)

For any fixed node f (the “finger”), the total cost of access sequence X is

$$O\left(m + n \log n + \sum_{j=1}^m \log(1 + |x_j - f|)\right)$$

Here $|x - y|$ is the *rank distance*, i. e., the number of nodes $z \in T$ with $x \leq z < y$ (or $x \leq z < y$)

Proof: “first attempt” set $w(x) := \frac{1}{|x-f|+1}$. $W = ?$

$$w(x) = \frac{1}{(1+|x-f|)^2} \quad W \leq C$$

amortized cost $O\left(\lg\left(\frac{w}{w(x)}\right)\right) = O\left(\lg(1+|x-f|)\right)$

dynamic finger : $f = x_{i-1}$

Splay Tree – Results [4]

For access sequence $X = x_1, \dots, x_m$ define $t(j)$ as the **number of different** items accessed since the **previous** access to the same node x_j (or since beginning)

Theorem 2.12 (Working Set Theorem)

The total cost of access sequence X is $O\left(m + n \log n + \sum_{j=1}^m \log(1 + t(j))\right)$ ◀

Proof: Assign weights $1, \frac{1}{4}, \frac{1}{9}, \dots$ to items in order of first access.

$$W = O(1)$$

first access has am. cost $O\left(\lg\left(\frac{W}{w(x_j)}\right)\right) = O\left(\lg\left(\frac{1}{1/t(j)^2}\right)\right) = O(\lg t(j))$

now: need to change weights!

after each access, permute weights to maintain in sorted order by $t(j)$

the accessed item gets 1

every item with previous weight $\frac{1}{k^2}$ if $\frac{1}{k^2} \geq w(x_j)$

$$\rightarrow \frac{1}{(k+1)^2}$$

What makes Splay tick?

When you rotate constantly, it is easy to lose orientation in the forest ...

A global view of Splay

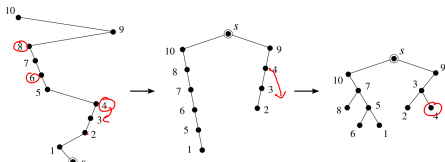


Fig. 2: A global view of splay trees. The transformation from the left to the middle illustrates rotate-to-root. The transformation from the left to the right illustrates splay trees.

Splay: Splay extends rotate-to-root: Let $s = v_0, v_1, \dots, v_k$ be the reversed search path. We view splaying as a two step process, see Figure 2. We first make s the root and split the search path into two paths, the path of elements smaller than s and the path of elements larger than s . If v_{2i+1} and v_{2i+2} are on the same side of s , we rotate them, i.e., we remove v_{2i+2} from the path and make it a child of v_{2i+1} .

Proposition 17. *The above description of splay is equivalent to the Sleator-Tarjan description.*

 Chalermsook, Goswami, Kozma, Mehlhorn, Saranurak: *Self-Adjusting Binary Search Trees: What Makes Them Tick?*, ESA 2015

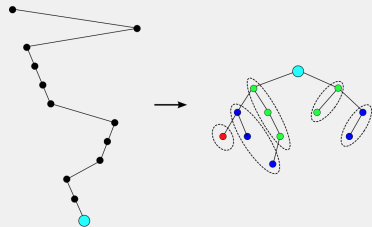
Intuition:

- ▶ Move-to-root = treap with newly accessed element gets *new maximal priority*
- ▶ Splay = that, plus *demotion of even index if previous on same side*

Anything magic about Splay?

Same paper shows: whole class of rearrangements satisfy access lemma

Characteristic quantities of the **search path** and the **after-tree**:



- length of search path: $|P|$ 12
- number of zigzags: z 4
- number of leaves: ℓ 5
- max left-depth or right-depth: d 3

Theorem. If, for every search:

- (1) max left/right depth: $d = O(1)$,
- (2) number of leaves+zigzags: $\ell + z = \Omega(|P|)$

\Rightarrow Algorithm shares many known good properties of Splay.

2.5 Biased Search Trees

Updates in Splay Trees

We've already seen: insert, delete, split, join all implemented on top of splay

- ▶ suffices to analyze these splay calls.
- ▶ but: potential so far only defined for fixed tree!

Updates in Splay Trees

We've already seen: insert, delete, split, join all implemented on top of splay

- ▶ suffices to analyze these splay calls.
- ▶ but: potential so far only defined for fixed tree!

Dynamic Splay Tree Potential

- ▶ **general state:** collection \mathcal{T} of splay trees (initially empty)

Updates in Splay Trees

We've already seen: insert, delete, split, join all implemented on top of splay

- ▶ suffices to analyze these splay calls.
- ▶ but: potential so far only defined for fixed tree!

Dynamic Splay Tree Potential

- ▶ **general state:** collection \mathcal{T} of splay trees (initially empty)
- ▶ all items ever inserted exist from the beginning in the "item ether" \mathcal{E} (outside any tree)
deleted items go back to the ether

↪ universe U of items is fixed

Updates in Splay Trees

We've already seen: insert, delete, split, join all implemented on top of splay

- ▶ suffices to analyze these splay calls.
- ▶ but: potential so far only defined for fixed tree!

Dynamic Splay Tree Potential

- ▶ **general state:** collection \mathcal{T} of splay trees (initially empty)
- ▶ all items ever inserted exist from the beginning in the "item ether" \mathcal{E} (outside any tree)
deleted items go back to the ether

↪ universe U of items is fixed

- ▶ assume every item in at most one tree at any time

$$\rightsquigarrow \Phi = \sum_{T \in \mathcal{T}} \sum_{x \in T} \underbrace{\lg(s(x))}_{r(x)} + \sum_{x \in \mathcal{E}} \underbrace{\lg(w(x))}_{r(x)}$$

Updates in Splay Trees

We've already seen: insert, delete, split, join all implemented on top of splay

- ▶ suffices to analyze these splay calls.
- ▶ but: potential so far only defined for fixed tree!

Dynamic Splay Tree Potential

- ▶ **general state:** collection \mathcal{T} of splay trees (initially empty)
- ▶ all items ever inserted exist from the beginning in the "item ether" \mathcal{E} (outside any tree)
deleted items go back to the ether

↪ universe U of items is fixed

- ▶ assume every item in at most one tree at any time

$$\rightsquigarrow \Phi = \sum_{T \in \mathcal{T}} \sum_{x \in T} \lg(s(x)) + \sum_{x \in \mathcal{E}} \lg(w(x))$$

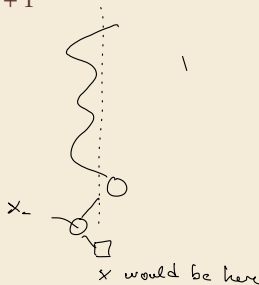
Notation: For $x \in U$ and T clear from context, abbreviate
 $x_- = \text{predecessor}_T(x)$ and $x_+ = \text{successor}_T(x)$

Analysis of Updates

Theorem 2.13 (Splay Tree Update Lemma)

For any assignment of weights $w : U \rightarrow \mathbb{R}_{>0}$, the amortized operation costs are upper bounded by

- (a) $\text{access}_T(x)$ where $x \in T$: $3 \lg \left(\frac{W}{w(x)} \right) + 1$ for $W = \sum_{x \in T} w(x)$ $\underbrace{3(r'(x) - r(x))}_{= W} \geq \lg(w(x))$
- (b) $\text{access}_T(x)$ where $x \notin T$: $3 \lg \left(\max \left\{ \frac{W}{w(x_-)}, \frac{W}{w(x_+)} \right\} \right) + 1$



Analysis of Updates

Theorem 2.13 (Splay Tree Update Lemma)

For any assignment of weights $w : U \rightarrow \mathbb{R}_{>0}$, the amortized operation costs are upper bounded by

(a) $\text{access}_T(x)$ where $x \in T$: $3 \lg \left(\frac{W}{w(x)} \right) + 1$ for $W = \sum_{x \in T} w(x)$

(b) $\text{access}_T(x)$ where $x \notin T$: $3 \lg \left(\max \left\{ \frac{W}{w(x_-)}, \frac{W}{w(x_+)} \right\} \right) + 1$

(c) $\text{join}(T_1, T_2)$: $3 \lg \left(\frac{W}{w(x)} \right) + O(1)$ for $x = \max(T_1)$, $W = \sum_{x \in T_1} w(x) + \sum_{x \in T_2} w(x)$



Analysis of Updates

Theorem 2.13 (Splay Tree Update Lemma)

For any assignment of weights $w : U \rightarrow \mathbb{R}_{>0}$, the amortized operation costs are upper bounded by

(a) $\text{access}_T(x)$ where $x \in T$: $3 \lg \left(\frac{W}{w(x)} \right) + 1$ for $W = \sum_{x \in T} w(x)$

(b) $\text{access}_T(x)$ where $x \notin T$: $3 \lg \left(\max \left\{ \frac{W}{w(x_-)}, \frac{W}{w(x_+)} \right\} \right) + 1$

(c) $\text{join}(T_1, T_2)$: $3 \lg \left(\frac{W}{w(x)} \right) + O(1)$ for $x = \max(T_1)$, $W = \sum_{x \in T_1} w(x) + \sum_{x \in T_2} w(x)$

(d) $\text{split}_T(x)$ where $x \in T$: $3 \lg \left(\frac{W}{w(x)} \right) + O(1)$ for $W = \sum_{x \in T} w(x)$

(e) $\text{split}_T(x)$ where $x \notin T$: $3 \lg \left(\max \left\{ \frac{W}{w(x_-)}, \frac{W}{w(x_+)} \right\} \right) + O(1)$

Analysis of Updates

Theorem 2.13 (Splay Tree Update Lemma)

For any assignment of weights $w : U \rightarrow \mathbb{R}_{>0}$, the amortized operation costs are upper bounded by

(a) $\text{access}_T(x)$ where $x \in T$: $3 \lg \left(\frac{W}{w(x)} \right) + 1$ for $W = \sum_{x \in T} w(x)$

(b) $\text{access}_T(x)$ where $x \notin T$: $3 \lg \left(\max \left\{ \frac{W}{w(x_-)}, \frac{W}{w(x_+)} \right\} \right) + 1$

(c) $\text{join}(T_1, T_2)$: $3 \lg \left(\frac{W}{w(x)} \right) + O(1)$ for $x = \max(T_1)$, $W = \sum_{x \in T_1} w(x) + \sum_{x \in T_2} w(x)$

(d) $\text{split}_T(x)$ where $x \in T$: $3 \lg \left(\frac{W}{w(x)} \right) + O(1)$ for $W = \sum_{x \in T} w(x)$

(e) $\text{split}_T(x)$ where $x \notin T$: $3 \lg \left(\max \left\{ \frac{W}{w(x_-)}, \frac{W}{w(x_+)} \right\} \right) + O(1)$

(f) $\text{insert}_T(x)$: $3 \lg \left(\max \left\{ \frac{W - w(x)}{w(x_-)}, \frac{W - w(x)}{w(x_+)} \right\} \right) + \lg \left(\frac{W}{w(x)} \right) + O(1)$

Analysis of Updates

Theorem 2.13 (Splay Tree Update Lemma)

For any assignment of weights $w : U \rightarrow \mathbb{R}_{>0}$, the amortized operation costs are upper bounded by

(a) $\text{access}_T(x)$ where $x \in T$: $3 \lg \left(\frac{W}{w(x)} \right) + 1$ for $W = \sum_{x \in T} w(x)$

(b) $\text{access}_T(x)$ where $x \notin T$: $3 \lg \left(\max \left\{ \frac{W}{w(x_-)}, \frac{W}{w(x_+)} \right\} \right) + 1$

(c) $\text{join}(T_1, T_2)$: $3 \lg \left(\frac{W}{w(x)} \right) + O(1)$ for $x = \max(T_1)$, $W = \sum_{x \in T_1} w(x) + \sum_{x \in T_2} w(x)$

(d) $\text{split}_T(x)$ where $x \in T$: $3 \lg \left(\frac{W}{w(x)} \right) + O(1)$ for $W = \sum_{x \in T} w(x)$

(e) $\text{split}_T(x)$ where $x \notin T$: $3 \lg \left(\max \left\{ \frac{W}{w(x_-)}, \frac{W}{w(x_+)} \right\} \right) + O(1)$

(f) $\text{insert}_T(x)$: $3 \lg \left(\max \left\{ \frac{W - w(x)}{w(x_-)}, \frac{W - w(x)}{w(x_+)} \right\} \right) + \lg \left(\frac{W}{w(x)} \right) + O(1)$

(g) $\text{delete}_T(x)$: $3 \lg \left(\frac{W}{w(x)} \right) + 3 \lg \left(\frac{W - w(x)}{w(x_-)} \right) + O(1)$



Analysis of Updates – Proof

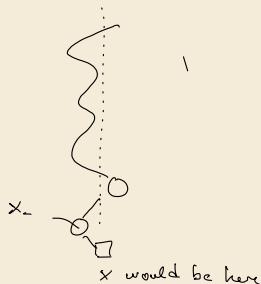
(a) $\text{access}_+(x)$ $x \in T$ from access lemma

$$W = w(T) \quad \text{directly get} \leq 3(r'(x) - r(x)) + 1$$

$$\leq 3 \lg\left(\frac{W}{w(x)}\right) + 1$$

(b) $x \notin T$

splay at x_- or x_+
at max for cost



(c) join (T_1, T_2)

splay at $x = \max T_1$



by access lemma

$$\leq 3(\lg(w_1(T_1)) - \lg(w(x))) + 1$$



joining trees changes potential

$$W = w(T_1) + w(T_2)$$

$$\Delta \Phi = \Phi(T) - (\Phi(\Delta) + \Phi(T_2))$$

$$= r'(x) - r(x)$$

$$\leq \lg(W) - \lg(w(T_1)) \leq 3(\lg W - \lg(w(T_1)))$$

total amortized cost $\leq 3 \lg\left(\frac{W}{w(x)}\right) + 1$

(d) $\text{splay}_T(x)$ $\text{splay at } x$

$x \in T$



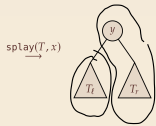
$\leq 3 \lg\left(\frac{W}{w(x)}\right) + O(1)$ am. cost

$\Delta \Phi \leq 0$

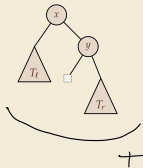
(e) $\text{splay}_T(x)$ $x \notin T \rightarrow \text{splay at } x_- \text{ or } x_+$

am. cost $\leq 3 \max\left\{\lg\left(\frac{W}{w(x_-)}\right), \lg\left(\frac{W}{w(x_+)}\right)\right\} + 1$

(f) $\text{insert}_T(x)$



\rightarrow



$W = w(T)$

$\text{splay}_T(y)$ $y \in \{x_-, x_+\}$

am. cost $\leq 3 \lg\left(\frac{W - w(x)}{w(y)}\right) + 1$

$\Delta \Phi$ y loses weight

x gains weight, new $s'(x) = W$ $s(x) = w(x)$

$\Delta \Phi \leq \lg\left(\frac{W}{w(x)}\right)$

(y) $\text{delete}_T(x)$

$\text{split}_{T_1}(x)$



$\text{join}(T_1, T_2)$

□.

Biased Search Trees

Biased search trees are a refinement of sorted dictionaries where

- ▶ elements have *known* weights $w(x)$ (that the data structure can use)
- ▶ supports **biased** search costs (as in Theorem 2.14 when using these weights)
- ▶ there is an explicit $\text{changeWeight}_T(x, \delta)$ operation that sets $w(x) := w(x) + \delta$

$$\text{Search costs } \lg \left(\frac{w}{w(x)} + 1 \right)$$


if $w(x) \ll \frac{1}{n}$ $w(x) = 2^{-n}$ could be at depth n

can keep height of tree $O(\log n)$

$$\bar{w}(x) := \frac{w(x) + \frac{1}{n}}{2}$$

Biased Search Trees

Biased search trees are a refinement of sorted dictionaries where

- ▶ elements have *known* weights $w(x)$ (that the data structure can use)
- ▶ supports **biased** search costs (as in Theorem 2.14 when using these weights)
- ▶ **there is an explicit $\text{changeWeight}_T(x, \delta)$ operation that sets $w(x) := w(x) + \delta$**
- ▶ prior to Splay Trees solved via *globally-biased 2, b trees*
 -  Bent, Sleator, Tarjan: *Biased search trees*, SICOMP 1985
 - ↪ achieves all operations in worst case biased time
 - ▶ rather intricate invariants and higher overhead
- ▶ biased access times also achievable via randomization: *treaps, zip-zip trees*

Weight Changes

Lemma 2.14 (Weight Changes)

For any assignment of weights $w : U \rightarrow \mathbb{R}_{>0}$, the amortized operation costs are upper bounded by

- (a) $\text{changeWeight}_T(\text{root}(T), \delta)$ where $\delta > 0$: $\lg\left(1 + \frac{\delta}{W}\right)$ for $W = \sum_{x \in T} w(x)$
- (b) $\text{changeWeight}_T(\text{root}(T), \delta)$ where $\delta < 0$: 0

Note: For changeWeight on arbitrary item x , first $\text{splay}(x)$ at cost $3 \lg\left(\frac{W}{w(x)}\right) + 1$.

Proof: b) $\Delta \Phi < 0$

$$\text{a) } \Delta \Phi = \lg(W + \delta) - \lg(W) = \lg\left(\frac{W + \delta}{W}\right) = \lg\left(1 + \frac{\delta}{W}\right)$$

2.6 Excursion: Online Algorithms

Online Problems

Online Problem

- ▶ algorithmic problem where input is only revealed over time
- ▶ and decisions/outputs need to be made along the way
- ▶ Typical example: stock trading, many scheduling problems

Online Problems

Online Problem

- ▶ algorithmic problem where input is only revealed over time
- ▶ and decisions/outputs need to be made along the way
- ▶ Typical example: stock trading, many scheduling problems
- ▶ default mode of thinking for **data structures!**
 - ▶ usually must answer one query before we receive the next

Online Problems

Online Problem

- ▶ algorithmic problem where input is only revealed over time
- ▶ and decisions/outputs need to be made along the way
- ▶ Typical example: stock trading, many scheduling problems
- ▶ default mode of thinking for **data structures!**
 - ▶ usually must answer one query before we receive the next

Opposite: Offline Problems

- ▶ get entire sequence of requests up front
- ↪ can work out solution with foresight

Elevator (a.k.a. Ski Rental) Problem

The Elevator Problem*

*Rent skis or buy when not known how many days of snow left.

- ▶ Suppose you arrive in a building and need to go k floors up.
- ▶ You can either take the stairs, which takes you 1 min per floor $\rightsquigarrow k$ min.
- ▶ Or take the elevator, which takes only 1 min for the k floors ... *once it has arrived!*

Elevator (a.k.a. Ski Rental) Problem

The Elevator Problem*

*Rent skis or buy when not known how many days of snow left.

- ▶ Suppose you arrive in a building and need to go k floors up.
- ▶ You can either take the stairs, which takes you 1 min per floor $\rightsquigarrow k$ min.
- ▶ Or take the elevator, which takes only 1 min for the k floors ... *once it has arrived!*
- ▶ The elevator arrives after w min, where you know $w \in [0..B)$ $\rightsquigarrow w + 1$ min
- ▶ Assume B is a finite bound, but $B \gg k$.

What should you do (to minimize time)?

Elevator (a.k.a. Ski Rental) Problem

The Elevator Problem*

*Rent skis or buy when not known how many days of snow left.

- ▶ Suppose you arrive in a building and need to go k floors up.
- ▶ You can either take the stairs, which takes you 1 min per floor $\rightsquigarrow k$ min.
- ▶ Or take the elevator, which takes only 1 min for the k floors ... *once it has arrived!*
- ▶ The elevator arrives after w min, where you know $w \in [0..B)$ $\rightsquigarrow w + 1$ min
- ▶ Assume B is a finite bound, but $B \gg k$.

What should you do (to minimize time)?

- ▶ Just walk?
- ▶ Wait for 2 min, then walk?
- ▶ Wait as long as it takes?

Elevator (a.k.a. Ski Rental) Problem

The Elevator Problem*

*Rent skis or buy when not known how many days of snow left.

- ▶ Suppose you arrive in a building and need to go k floors up.
- ▶ You can either take the stairs, which takes you 1 min per floor $\rightsquigarrow k$ min.
- ▶ Or take the elevator, which takes only 1 min for the k floors ... *once it has arrived!*
- ▶ The elevator arrives after w min, where you know $w \in [0..B)$ $\rightsquigarrow w + 1$ min
- ▶ Assume B is a finite bound, but $B \gg k$.

What should you do (to minimize time)?

- ▶ Just walk?
- ▶ Wait for 2 min, then walk?
- ▶ Wait as long as it takes?

Impossible to tell what is better without some assumption on w , e. g., distribution assumption.

Elewaiting as Sequence of Requests

- ▶ formally, instance of online problem is sequence of requests $x = x_1, \dots, x_T$
- ▶ our algorithm \mathcal{A} must produce output after each request
At time t , output $y_t = \mathcal{A}(x_1, \dots, x_t)$
- ▶ overall cost on instance depends on sequence of outputs
 $A(x) = \text{cost}(\mathcal{A}(x_1), \dots, \mathcal{A}(x_1, \dots, x_T))$

Elewaiting as Sequence of Requests

- ▶ formally, instance of online problem is sequence of requests $x = x_1, \dots, x_T$
- ▶ our algorithm \mathcal{A} must produce output after each request
At time t , output $y_t = \mathcal{A}(x_1, \dots, x_t)$
- ▶ overall cost on instance depends on sequence of outputs
 $A(x) = \text{cost}(\mathcal{A}(x_1), \dots, \mathcal{A}(x_1, \dots, x_T))$

↪ Elevator Problem

- ▶ $x_t = [\text{elevator arrives at time } t], \quad T = B$
- ▶ $y_t = [\text{take stairs at time } t]$

Competitive Analysis

Elegant alternative to (potentially unrealistic) random models: *competitive analysis*

- ▶ Suppose $\underline{OPT}(x)$ is the (cost of the) **optimal offline solution** for x
Note: \underline{OPT} knows the future (entire access sequence x)

Competitive Analysis

Elegant alternative to (potentially unrealistic) random models: *competitive analysis*

- ▶ Suppose $OPT(x)$ is the (cost of the) **optimal offline solution** for x
Note: OPT knows the future (entire access sequence x)
- ▶ For the elevator problem, OPT would wait if $w \leq k - 1$ and walk right away otherwise.
 $\rightsquigarrow OPT(w) = \min\{w + 1, k\}$

Competitive Analysis

Elegant alternative to (potentially unrealistic) random models: *competitive analysis*

- ▶ Suppose $OPT(x)$ is the (cost of the) **optimal offline solution** for x
Note: OPT knows the future (entire access sequence x)
- ▶ For the elevator problem, OPT would wait if $w \leq k - 1$ and walk right away otherwise.
 $\rightsquigarrow OPT(w) = \min\{w + 1, k\}$

Definition 2.15 (Competitive ratio)

The *competitive ratio* of an online algorithm \mathcal{A} (for size n) is

$$c = \max_x \frac{A(x)}{OPT(x)} \quad \text{where the maximum is taken over all instances of size } n.$$

We say that \mathcal{A} is *c-competitive*. ◀

Competitive Analysis

Elegant alternative to (potentially unrealistic) random models: *competitive analysis*

- ▶ Suppose $OPT(x)$ is the (cost of the) **optimal offline solution** for x
Note: OPT knows the future (entire access sequence x)
- ▶ For the elevator problem, OPT would wait if $w \leq k - 1$ and walk right away otherwise.
 $\rightsquigarrow OPT(w) = \min\{w + 1, k\}$

Definition 2.15 (Competitive ratio)

The *competitive ratio* of an online algorithm \mathcal{A} (for size n) is

$$c = \max_x \frac{A(x)}{OPT(x)} \quad \text{where the maximum is taken over all instances of size } n.$$

We say that \mathcal{A} is *c-competitive*. ◀

- ▶ competitive ratio of “Just walk” is $\max_{w \in [0..B)} \frac{k}{\min\{k, w + 1\}} = k$.

Competitive Analysis

Elegant alternative to (potentially unrealistic) random models: *competitive analysis*

- ▶ Suppose $OPT(x)$ is the (cost of the) **optimal offline solution** for x
Note: OPT knows the future (entire access sequence x)
- ▶ For the elevator problem, OPT would wait if $w \leq k - 1$ and walk right away otherwise.
 $\rightsquigarrow OPT(w) = \min\{w + 1, k\}$

Definition 2.15 (Competitive ratio)

The *competitive ratio* of an online algorithm \mathcal{A} (for size n) is

$$c = \max_x \frac{A(x)}{OPT(x)} \quad \text{where the maximum is taken over all instances of size } n.$$

We say that \mathcal{A} is *c-competitive*. ◀

- ▶ competitive ratio of “Just walk” is $\max_{w \in [0..B)} \frac{k}{\min\{k, w + 1\}} = k$
- ▶ “Always wait” has $\max_{w \in [0..B)} \frac{w + 1}{\min\{k, w + 1\}} = \frac{B}{k}$

Competitive Elewaiting

Can do much better by hedging out bets a bit: \mathcal{A}_t waits t min then walks

$$\blacktriangleright A_t(w) = \begin{cases} w + 1 & w \leq t \\ t + k & w > t \end{cases}$$

Competitive Elewaiting

Can do much better by hedging out bets a bit: \mathcal{A}_t waits t min then walks

$$\blacktriangleright A_t(w) = \begin{cases} w + 1 & w \leq t \\ t + k & w > t \end{cases}$$

\rightsquigarrow Optimal choice for t

$$\arg \min_t \max_{w \in [0..B]} \frac{A_t(w)}{OPT(w)} =$$

Competitive Elewaiting

Can do much better by hedging out bets a bit: \mathcal{A}_t waits t min then walks

$$\blacktriangleright A_t(w) = \begin{cases} w + 1 & w \leq t \\ t + k & w > t \end{cases}$$

\rightsquigarrow Optimal choice for t

$$\arg \min_t \max_{w \in [0..B]} \frac{A_t(w)}{OPT(w)} = k$$

$\rightsquigarrow \mathcal{A}_k$ is 2-competitive

Further Famous Online-Algorithms

- ▶ Move-to-Front is 4-competitive for the list-update problem
assuming only local swaps are allowed *∴ sensitive to cost model*
- ▶ LRU is k -competitive in online paging (maintaining a cache of size k)
- ▶ Online bin packing (which box to put new item into, repacking not allowed)

2.7 Dynamic Optimality

Adaptive Trees as Online Algorithm

- ▶ Serving unknown access sequence to nodes of a BST is an online problem *par excellence*
Note: Focus here on access-only (no insertions & deletions).
- ▶ Splay trees are an online algorithm for this problem!
explicitly designed to do well in spite of not knowing the future

Adaptive Trees as Online Algorithm

- ▶ Serving unknown access sequence to nodes of a BST is an online problem *par excellence*
Note: Focus here on access-only (no insertions & deletions).
- ▶ Splay trees are an online algorithm for this problem!
explicitly designed to do well in spite of not knowing the future

Compare to what

- ▶ *OPT* gets entire access sequence in advance
- ▶ but need to fix what it is allowed to do! when is an access counted as served?

Adaptive Trees as Online Algorithm

- ▶ Serving unknown access sequence to nodes of a BST is an online problem *par excellence*
Note: Focus here on access-only (no insertions & deletions).
- ▶ Splay trees are an online algorithm for this problem!
explicitly designed to do well in spite of not knowing the future

Compare to what

- ▶ *OPT* gets entire access sequence in advance
- ▶ but need to fix what it is allowed to do! when is an access counted as served?

The BST Model of Computation

- ▶ At any point in time, maintain a BST over $[1..n]$, and a *finger* (pointer) at one node.
Initial tree can be chosen by algorithm. (doesn't matter if access sequence long)

Adaptive Trees as Online Algorithm

- ▶ Serving unknown access sequence to nodes of a BST is an online problem *par excellence*
Note: Focus here on access-only (no insertions & deletions).
- ▶ Splay trees are an online algorithm for this problem!
explicitly designed to do well in spite of not knowing the future

Compare to what

- ▶ *OPT* gets entire access sequence in advance
- ▶ but need to fix what it is allowed to do! when is an access counted as served?

The BST Model of Computation

- ▶ At any point in time, maintain a BST over $[1..n]$, and a *finger* (pointer) at one node.
Initial tree can be chosen by algorithm. (doesn't matter if access sequence long)
- ▶ Each step of the “computation” is one of the following 5 operations:
 - ▶ **move** finger from the current node to **left child**, **right child**, or **parent** (assuming they exist)

Adaptive Trees as Online Algorithm

- ▶ Serving unknown access sequence to nodes of a BST is an online problem *par excellence*
Note: Focus here on access-only (no insertions & deletions).
- ▶ Splay trees are an online algorithm for this problem!
explicitly designed to do well in spite of not knowing the future

Compare to what

- ▶ *OPT* gets entire access sequence in advance
- ▶ but need to fix what it is allowed to do! when is an access counted as served?

The BST Model of Computation

- ▶ At any point in time, maintain a BST over $[1..n]$, and a *finger* (pointer) at one node.
Initial tree can be chosen by algorithm. (doesn't matter if access sequence long)
- ▶ Each step of the "computation" is one of the following 5 operations:
 - ▶ **move** finger from the current node to **left child**, **right child**, or **parent** (assuming they exist)
 - ▶ **rotate** the parent edge of the current node (assuming not at root)

Adaptive Trees as Online Algorithm

- ▶ Serving unknown access sequence to nodes of a BST is an online problem *par excellence*
Note: Focus here on access-only (no insertions & deletions).
- ▶ Splay trees are an online algorithm for this problem!
explicitly designed to do well in spite of not knowing the future

Compare to what

- ▶ OPT gets entire access sequence in advance
- ▶ but need to fix what it is allowed to do! when is an access counted as served?

The BST Model of Computation

- ▶ At any point in time, maintain a BST over $[1..n]$, and a *finger* (pointer) at one node.
Initial tree can be chosen by algorithm. (doesn't matter if access sequence long)
- ▶ Each step of the "computation" is one of the following 5 operations:
 - ▶ **move** finger from the current node to **left child**, **right child**, or **parent** (assuming they exist)
 - ▶ **rotate** the parent edge of the current node (assuming not at root)
 - ▶ **find**: report the current node as the sought value, reset finger to root of tree

↪ OPT finds shortest valid sequence of operations serving accesses

The Splay Tree Conjecture

Since Splay trees have all the strong adaptive properties, Sleator and Tarjan immediately conjectured:

CONJECTURE 1 (DYNAMIC OPTIMALITY CONJECTURE). *Consider any sequence of successful accesses on an n -node search tree. Let A be any algorithm that carries out each access by traversing the path from the root to the node containing the accessed item, at a cost of one plus the depth of the node containing the item, and that between accesses performs an arbitrary number of rotations anywhere in the tree, at a cost of one per rotation. Then the total time to perform all the accesses by splaying is no more than $O(n)$ plus a constant times the time required by algorithm A .*

The Splay Tree Conjecture

Since Splay trees have all the strong adaptive properties, Sleator and Tarjan immediately conjectured:

CONJECTURE 1 (DYNAMIC OPTIMALITY CONJECTURE). *Consider any sequence of successful accesses on an n -node search tree. Let A be any algorithm that carries out each access by traversing the path from the root to the node containing the accessed item, at a cost of one plus the depth of the node containing the item, and that between accesses performs an arbitrary number of rotations anywhere in the tree, at a cost of one per rotation. Then the total time to perform all the accesses by splaying is no more than $O(n)$ plus a constant times the time required by algorithm A .*

(paper predates the term competitive ratio ...)

In today's words: **CONJECTURE 1:** Splay is $O(1)$ -competitive.

The Splay Tree Conjecture

Since Splay trees have all the strong adaptive properties, Sleator and Tarjan immediately conjectured:


CONJECTURE 1 (DYNAMIC OPTIMALITY CONJECTURE). Consider any sequence of successful accesses on an n -node search tree. Let A be any algorithm that carries out each access by traversing the path from the root to the node containing the accessed item, at a cost of one plus the depth of the node containing the item, and that between accesses performs an arbitrary number of rotations anywhere in the tree, at a cost of one per rotation. Then the total time to perform all the accesses by splaying is no more than $O(n)$ plus a constant times the time required by algorithm A .

(paper predates the term competitive ratio ...)

In today's words: **CONJECTURE 1:** Splay is $O(1)$ -competitive.

We're far from settling that conjecture ...

OPEN: Is any online BST $o(\log \log n)$ -competitive?

OPEN: Are Splay trees $o(\log n)$ -competitive? 

► Clearly, $OPT(x) = \Omega(m)$ for $m = |x|$ \rightsquigarrow any static balanced tree is $O(\log n)$ -competitive

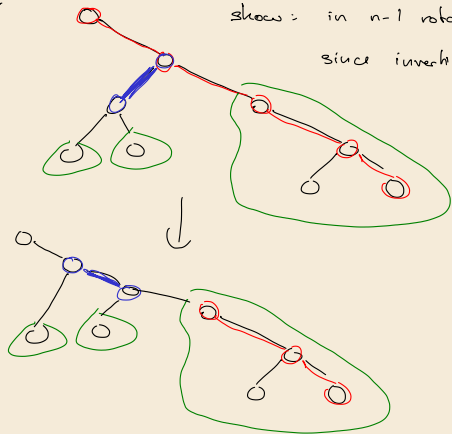
Rotation Distance & Initial Trees

Lemma 2.16

Using at most $2n - 2$ rotations, we can transform any BST on $[1..n]$ into any other BST on the same keys. ◀

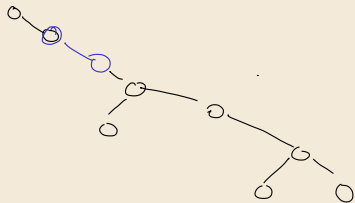
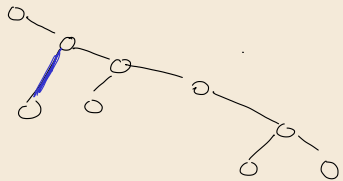
Note, actually $2n - 6$ suffice.

Proof.



shows: in $n-1$ rotations can go to canonical tree
since invertible operation, this proves the sketch.

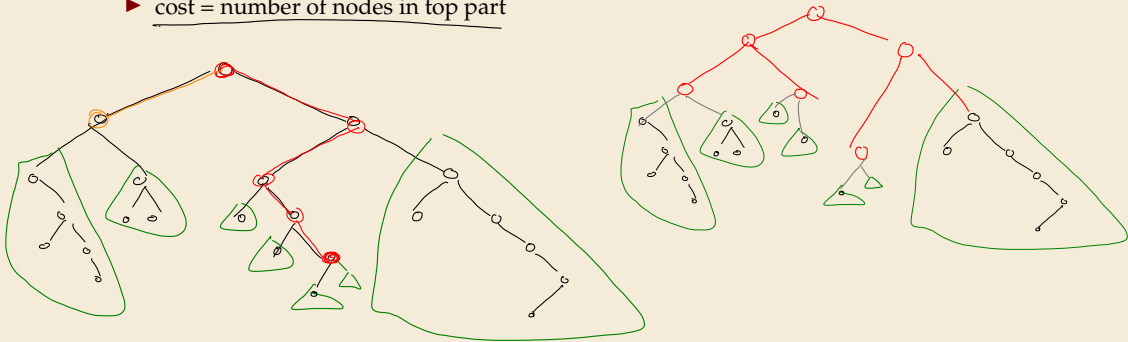
o-o-o-o-o



Simplified BST Model

equivalent to original model up to constant factors, but easier to design algorithms

- ▶ rearrange “top part” (touched part) of tree (connected subgraph of tree containing root)
- ▶ cost = number of nodes in top part



Simplified BST Model

equivalent to original model up to constant factors, but easier to design algorithms

- ▶ rearrange “top part” (touched part) of tree (connected subgraph of tree containing root)
 - ▶ cost = number of nodes in top part
 - ▶ (isolated rotations in tree can be delayed)
 - ▶ **OPEN:** *Is restricting changes to **search path** w.l.o.g. in BST algorithms?*
 (“strict model”)
 - ▶ all sensible online algorithms work that way ...

Simplified BST Model

equivalent to original model up to constant factors, but easier to design algorithms

- ▶ rearrange “top part” (touched part) of tree (connected subgraph of tree containing root)
 - ▶ cost = number of nodes in top part
 - ▶ (isolated rotations in tree can be delayed)
 - ▶ **OPEN:** *Is restricting changes to **search path** w.l.o.g. in BST algorithms?*
 (“strict model”)
 - ▶ all sensible online algorithms work that way ...
- ▶ access only at root
 - ▶ at constant-factor overhead, can rotate up and back down again.

Facts on *OPT*

- ▶ can be computed in exponential time (already not trivial)
- ▶ exact *OPT* likely NP-hard

If each access is a set of items that the algorithm may order as convenient, the problem is known to be NP-complete

Lower Bounds for OPT

- ▶ Most access sequences must be “hard”, i. e., require $\Omega(m \log n)$ costs even for *OPT*
 - ▶ execution of BST algorithm **encodes** access sequence (must serve each differently!)
 - ▶ number of access sequences is n^m
 - ▶ each step of a BST algorithm is one of $O(1)$ choices

$\lg(n^m) = m \lg n$ bits to encode most sequences

get $O(1)$ bits per step of BST algorithm

($\leq 5^t$ different access sequences with t steps)

Lower Bounds for OPT

- ▶ Most access sequences must be “hard”, i. e., require $\Omega(m \log n)$ costs even for *OPT*
 - ▶ execution of BST algorithm **encodes** access sequence (must serve each differently!)
 - ▶ number of access sequences is n^m
 - ▶ each step of a BST algorithm is one of $O(1)$ choices
- ▶ The bitwise reversal sequence R_n needs $OPT(R_n) = \Omega(n \log n)$ cost.



Wilber: *Lower Bounds for Accessing Binary Search Trees with Rotations*, SICOMP 1989

- ▶ defined for $n = 2^k$
- ▶ $R_n = (\text{rev}_k(0), \text{rev}_k(1), \text{rev}_k(2), \dots, \text{rev}_k(n-1))$, $\text{rev}_k(i) = i$ in binary written in reverse.
 $R_{16} = (0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15)$
- ▶ Wilber proved further lower bounds on *OPT*

Tango Trees

- ▶ Online BST algorithm specifically designed to stay close to one of Wilber's lower bounds
details skipped
- ▶ achieves a competitive ratio of $O(\log \log n)$
- ▶ may use $\Omega(\log n \log \log n)$ amortized cost \rightsquigarrow not $o(\log \log n)$ -competitive

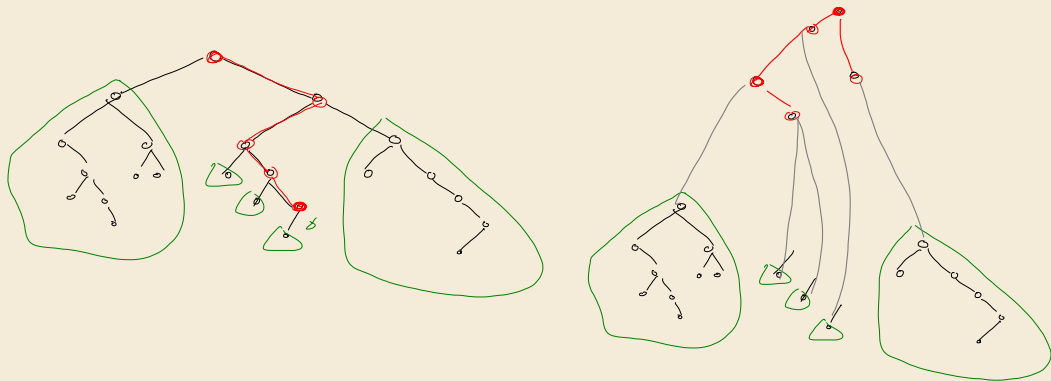


Demaine, Harmon, Iacono, Pătraşcu: *Dynamic Optimality – Almost*, SICOMP 2007

Greedy Future

Natural **offline** idea: find element, then rearrange search path as treap with next-future-access time (to element or subtrees) as priority

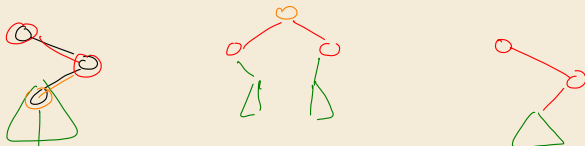
- Of course an **offline** method since it uses the future access times



Greedy Future

Natural **offline** idea: find element, then rearrange search path as treap with next-future-access time (to element or subtrees) as priority

- ▶ Of course an **offline** method since it uses the future access times
- ▶ Long conjectured to be close to optimal ($+m$ over OPT ?)



Greedy Future

Natural **offline** idea: find element, then rearrange search path as treap with next-future-access time (to element or subtrees) as priority

- ▶ Of course an **offline** method since it uses the future access times
- ▶ Long conjectured to be close to optimal ($+m$ over OPT ?)
- ▶ Now known to be at best a 2-approximation, and has cost $OPT + \Omega(m \log \log n)$ on some access sequences



Sadeh, Kaplan: *Dynamic Binary Search Trees: Improved Lower Bounds for the Greedy-Future Algorithm*, STACS 2023

Greedy Future

Natural **offline** idea: find element, then rearrange search path as treap with next-future-access time (to element or subtrees) as priority

- ▶ Of course an **offline** method since it uses the future access times
- ▶ Long conjectured to be close to optimal ($+m$ over OPT ?)
- ▶ Now known to be at best a 2-approximation, and has cost $OPT + \Omega(m \log \log n)$ on some access sequences



Sadeh, Kaplan: *Dynamic Binary Search Trees: Improved Lower Bounds for the Greedy-Future Algorithm*, STACS 2023

- ▶ *May still be an offline $O(1)$ -approximation to OPT*
OPEN: Is GreedyFuture a $o(\log n)$ -approximation?

2.8 The Geometric View of BSTs

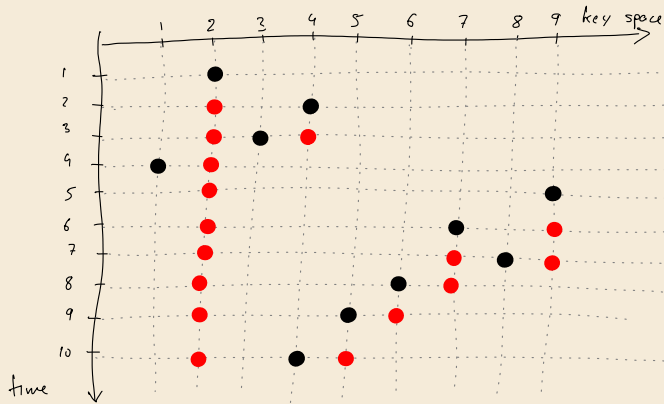
Different Way of Looking at BSTs

Beautiful reformulation of problem



Demaine, Harmon, Iacono, Kane, Pătraşcu: *The Geometry of Binary Search Trees*, SODA 2009

- ▶ inspired a new contender for a dynamically optimal BST algorithm (beyond Splay)
- ▶ made several existing results more intuitive, in particular lower bounds for *OPT*



access sequence

a_1, a_2, \dots, a_m

\leadsto point set

$P = \{(a_t, t), t \in [m]\}$

add points to satisfy
rectangles

Different Way of Looking at BSTs

Beautiful reformulation of problem



Demaine, Harmon, Iacono, Kane, Pătraşcu: *The Geometry of Binary Search Trees*, SODA 2009

- ▶ inspired a new contender for a dynamically optimal BST algorithm (beyond Splay)
- ▶ made several existing results more intuitive, in particular lower bounds for *OPT*
- ▶ but has not fueled the resolution of the Splay tree conjecture (yet?)

Arborally Satisfied Point Sets

Each proper rectangle must contain another point.

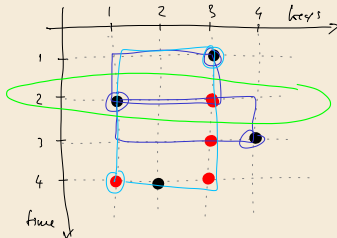
Definition 2.17 (Arborally Satisfied)

A 2D point set X is arborally satisfied if it does not contain any unsatisfied rectangles.

An *unsatisfied rectangle* is formed by $p_1, p_2 \in X, p_1 = (x_1, y_1), p_2 = (x_2, y_2)$ with

- ▶ with $x_1 \neq x_2$ and $y_1 \neq y_2$ and
- ▶ $X \cap [x_1, x_2] \times [y_1, y_2] = \{p_1, p_2\}$

BST tract interpretation: $X_t := X \cap \mathbb{R} \times \{t\}$ corresponds to BST nodes touched at time t




BST \iff Arborally Satisfied

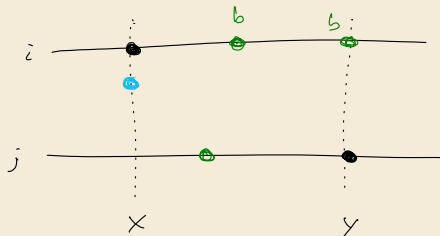
Theorem 2.18

A point set $Y \subset [n] \times [m]$ corresponds to valid BST algorithm iff Y is arborally satisfied. \blacktriangleleft

time slice $Y \cap [n] \times \{t\} =$ to path of BST touched at time t

Proof: " \implies " Let γ be the trace of a BST algorithm serving access a_1, \dots, a_m

to show \exists 

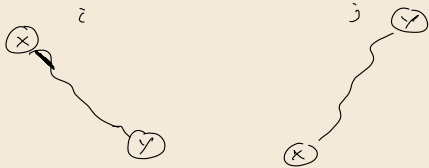


$b = lca(x, y)$ (changes over time)
 $x \leq b \leq y$

- right before time i
 if $b \neq x \rightarrow$ b on path from root to x ✓
- right before time j
 if $b \neq y \rightarrow$ visit b now ✓

\implies only case left:

at time i $b = x$ and at j $b = y$



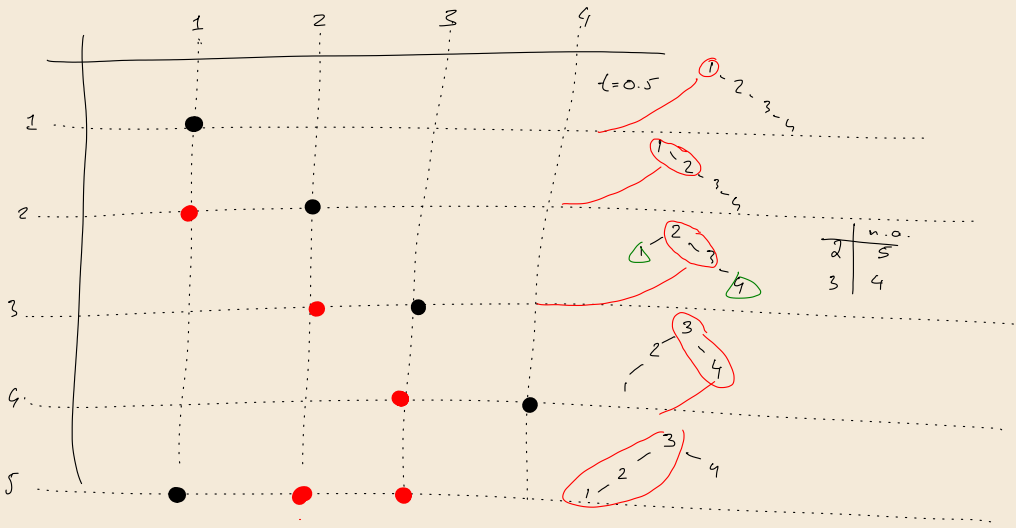
at some time $k \in (i..j)$

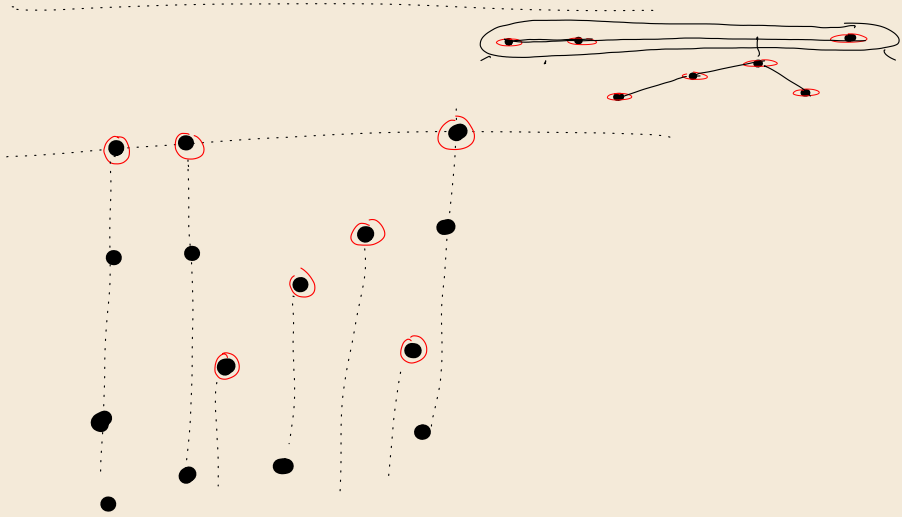
x is rotated

" \Leftarrow " given Y a.s. point set to show: \exists BST algorithm with trace Y
 (must be possible to rearrange top part of tree)

at time t : touch top part ($Y \cap [u] \times \{t\}$)
 rearrange this as treap w/ priority = "next access time"

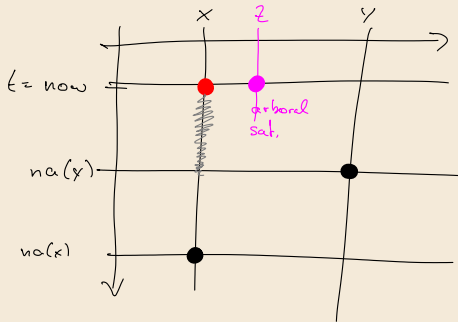
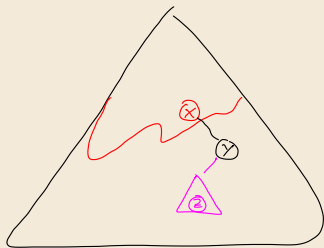
Invariant: between t and $t+1$ entire BST is a treap
 w.r.b. next access time





to show: invariant holds (which implies next access is to top part of BJT)

$na(y)$ earlier than $na(x)$



since Y as b. sat.
 must have point
 (z, t) touched now
 but y on path
 to z \downarrow

□

Minimum Arborally Satisfied Superset

Given: Set of points $X \subset [n] \times \mathbb{N}$

Goal: smallest **arborally satisfied** superset $Y \supseteq X$

- ▶ care particularly for $X = \{(x_t, t) : t \in [m]\}$ for accesses x_1, \dots, x_m
- ▶ cost of BST algorithm corresponding to Y : $|Y|$