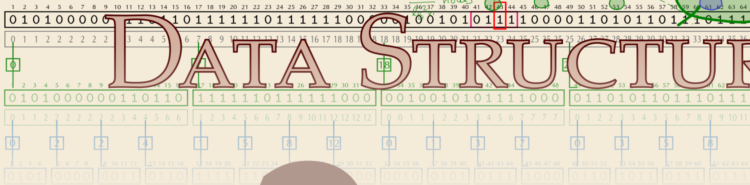
ADVANCED DATA STRUCTURES



2 Adaptive Trees

Advanced Data Structures · Summer 2026

Prof. Dr. Sebastian Wild

CS650 (Summer 2026)
Philipps-Universität Marburg

version `2026-05-19 10:12`

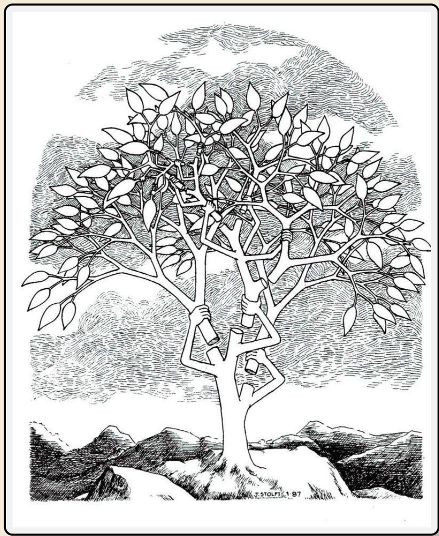
Outline

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
2.11 Lazy Search Tree Operations

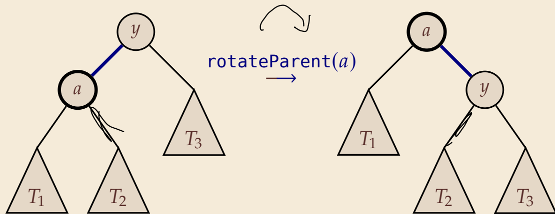
Self-Adjusting Data Structures

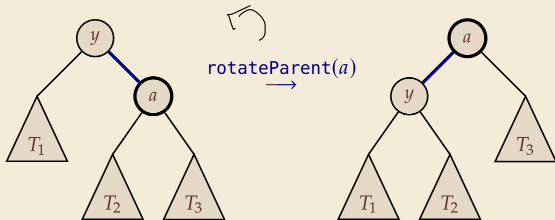
Idea: set of rules that makes data structure automatically adapt to use case



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-Organizating Binary Search Trees*, JACM **1978**

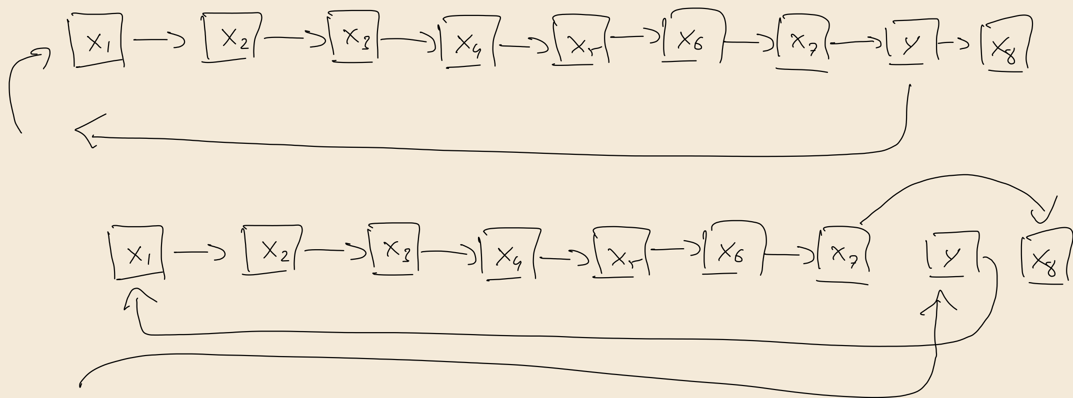
Inspired by **Move-To-Front heuristic** of self-adjusting linked **lists**

- 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

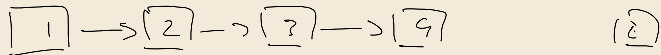
Move-to-front

unsorted linked list

search(y)



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



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

$$\sum_{i \geq 0} z^i = \frac{1}{1-z} \quad \Big| \ \frac{d}{dz}$$

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

Move-to-root

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

Allen, Munro: *Self-Organizating 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 $p_1, \ldots, p_n$.

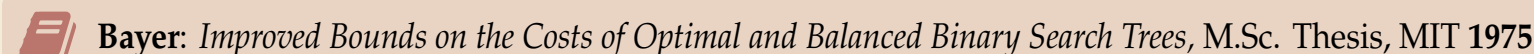
⇝ Expected search cost in a BST T given by $\sum_{i=1}^{n} p_i \cdot \mathrm{depth}_T(\textcircled{i})$

- 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)

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


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, $j = 0, \ldots, n$ even cleaner:

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

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 $p_1, \ldots, p_n$.

Iid Accesses

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

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

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

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

Iid Accesses

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

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

▶ 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)

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

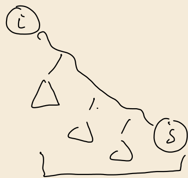
◀

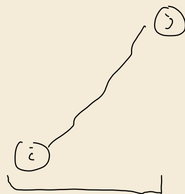
Bayer: *Improved Bounds on the Costs of Optimal and Balanced Binary Search Trees*, M.Sc. Thesis, MIT **1975**

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 $i+1,\ldots,j$.
Similarly, j ancestor of i iff j requested more recently than any of $i,\ldots,j-1$.





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

"⇐" accesses to all of $i,\ldots,j$ 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 $i+1, \ldots, j$.
Similarly, (j) ancestor of (i) iff j requested more recently than any of $i, \ldots, j-1$. ◀

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

Corollary 2.3

Under the iid accesses model and MTR heuristic, $\mathbb{P}[A^i_j] = \begin{cases} \dfrac{p_i}{p_i + \cdots + p_j} & i < j \\[2ex] \dfrac{p_i}{p_j + \cdots + p_i} & i > j \end{cases}$ ◀

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.

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

$2\ln 2 = 1.39$

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

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.

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

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

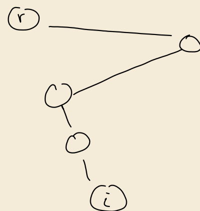
Proof:

$$C_{MTR} \;=\; \sum_{i=1}^{n} p_i \cdot (1 + \mathrm{depth}(\textcircled{i}))$$

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.

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



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

Proof:

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

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.

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

◀

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

Proof:

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

$$= \; 1 \;+\; \sum_{i=1}^{n} \sum_{j \ne i} p_i \mathbb{E}[A_i^j]$$

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.

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

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

Proof:

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

$$= 1 + \sum_{i=1}^{n} \sum_{j \ne i} p_i \mathbb{E}[A_i^j] \underset{\text{Lemma}}{=} \frac{p_j}{p_i + \cdots + p_j}$$

$$= 1 + 2 \sum_{1 \le i < j \le n} \frac{p_i p_j}{p_i + \cdots + p_j}$$

This proves the first part.

Move-to-Root on iid Accesses – Analysis

Proof (cont.):

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

Move-to-Root on iid Accesses – Analysis

Proof (cont.):

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

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

Move-to-Root on iid Accesses – Analysis

Proof (cont.):

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

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

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

$$\frac{\delta_{i,j} - \delta_{i,j-1}}{\delta_{i,j}} = \frac{\dfrac{p_i + \cdots + p_{j-1} + p_j - (p_i + \cdots + p_{j-1})}{p_i}}{\dfrac{p_i + \cdots + p_j}{p_i}}$$

Move-to-Root on iid Accesses – Analysis

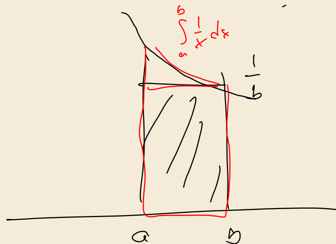
Proof (cont.):

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

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

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

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



Move-to-Root on iid Accesses – Analysis

Proof (cont.):

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

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

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

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

$$a \le b \implies (b-a)\cdot\tfrac{1}{b} = \int_a^b \tfrac{1}{b}\,dx < \int_a^b \tfrac{1}{x}\,dx$$

Move-to-Root on iid Accesses – Analysis

Proof (cont.):

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

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

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

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

Move-to-Root on iid Accesses – Analysis

Proof (cont.):

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

$\rightsquigarrow \quad 1 = \delta_{i,i} \le \delta_{i,i+1} \le \cdots \le \delta_{i,n} \le \dfrac{1}{p_i}$

$$
\begin{aligned}
C_{MTR} &= 1 + 2\sum_{i=1}^{n-1} p_i \sum_{j=i+1}^{n} \frac{p_j}{p_i + \cdots + p_j} \\
&= 1 + 2\sum_{i=1}^{n-1} p_i \sum_{j=i+1}^{n} \frac{\delta_{i,j} - \delta_{i,j-1}}{\delta_{i,j}} \\
&< 1 + 2\sum_{i=1}^{n-1} p_i \sum_{j=i+1}^{n} \int_{x=\delta_{i,j-1}}^{\delta_{i,j}} \frac{1}{x}\,dx \\
&= 1 + 2\sum_{i=1}^{n-1} p_i \int_{x=\delta_{i,i}}^{\delta_{i,n}} \frac{1}{x}\,dx \\
&= 1 + 2\sum_{i=1}^{n-1} p_i \ln(\delta_{i,n}/\delta_{i,i}) \;\le\; 1 + 2\ln 2 \sum_{i=1}^{n-1} p_i \lg(1/p_i) \;=\; 1 + 2\ln 2 \cdot \mathcal{H}
\end{aligned}
$$

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

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

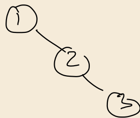
■

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 $p_1, \ldots, p_n$
- 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

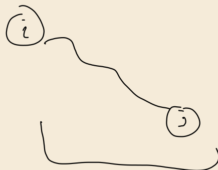


Unbalanced BSTs under i.i.d. model

Interestingly, the same cost result from unbalanced BSTs!

- Suppose we repeatedly draw x i.i.d. from $[1..n]$ w/p $p_1, \ldots, p_n$
- 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 $\dfrac{p_i}{p_i + \cdots + p_j}$



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 $p_1, \ldots, p_n$
- 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 $\dfrac{p_i}{p_i + \cdots + p_j}$

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

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 $p_1, \ldots, p_n$
- 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 $\dfrac{p_i}{p_i + \cdots + p_j}$

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

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

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 $p_1, \ldots, p_n$
- 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 $\dfrac{p_i}{p_i + \cdots + p_j}$

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

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

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)

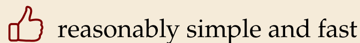
There exists a sequence of n requests, such that starting from any tree T, the amortized access cost over the course of the n requests is $\Omega(n)$.

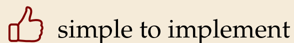
Proof:
see exercises

2.3 Splay Trees

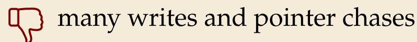
Splay Trees

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


reasonably simple and fast


simple to implement

remarkable theoretical properties (*access theorems*) ⇝ later

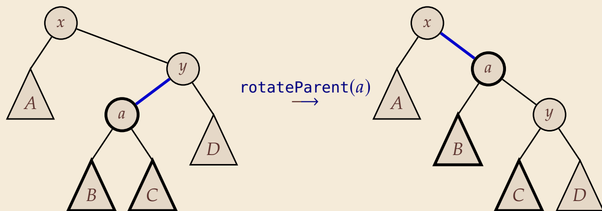

many writes and pointer chases

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

Move-to-root reloaded

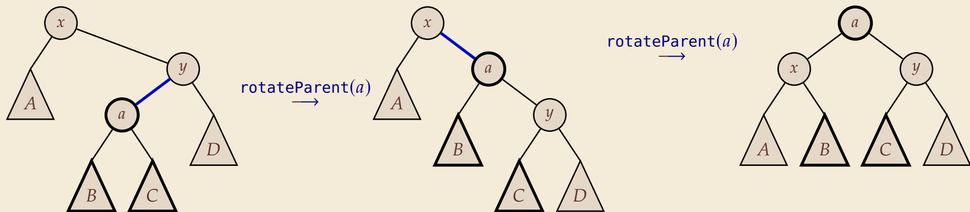
What's wrong in Move-to-front?

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



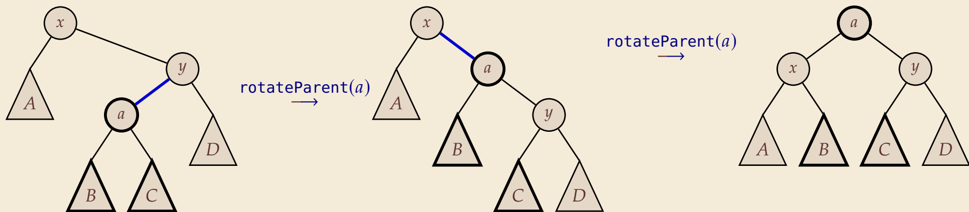
Move-to-root reloaded

What's wrong in Move-to-front? Let's look at the possible cases of parent **and grandparent**



Move-to-root reloaded

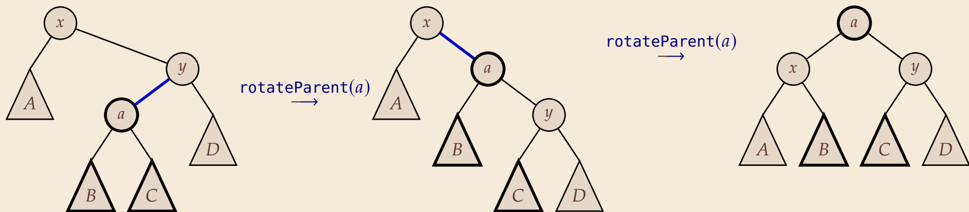
What's wrong in Move-to-front? Let's look at the possible cases of parent **and grandparent**



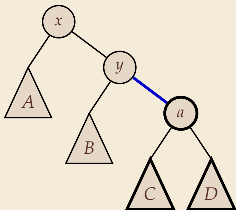
Using simple `rotateParent` calls, the subtree we came from (B or C) is lifted up!

Move-to-root reloaded

What's wrong in Move-to-front? Let's look at the possible cases of parent **and grandparent**

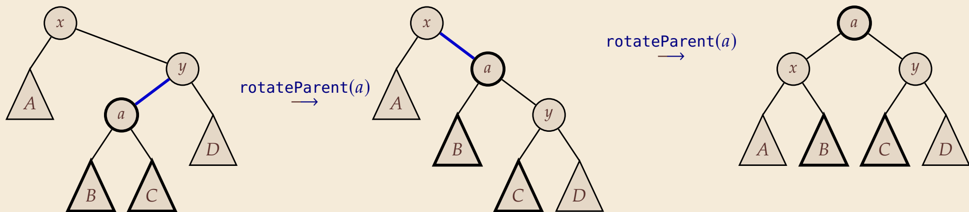


Using simple `rotateParent` calls, the subtree we came from (B or C) is lifted up!

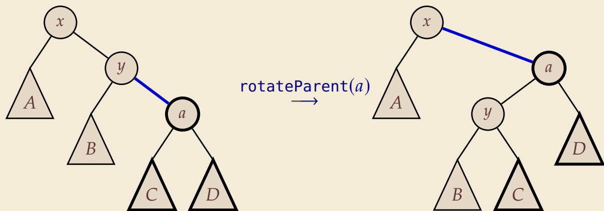


Move-to-root reloaded

What's wrong in Move-to-front? Let's look at the possible cases of parent **and grandparent**

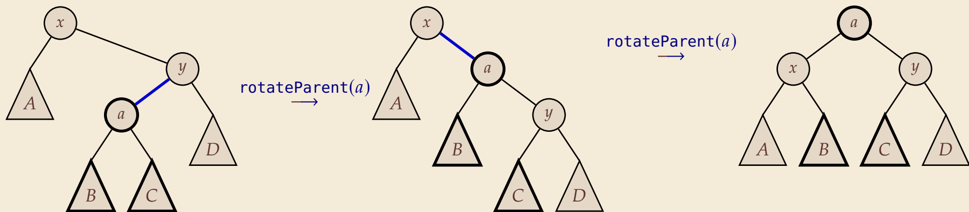


Using simple `rotateParent` calls, the subtree we came from (B or C) is lifted up!

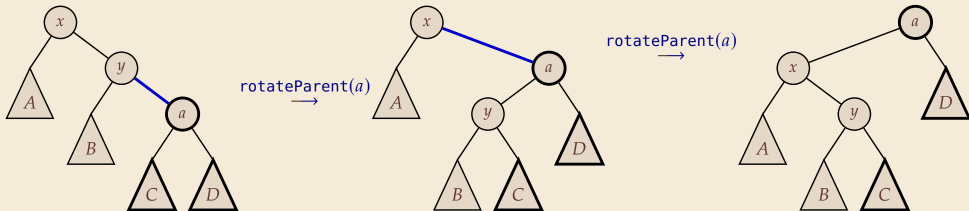


Move-to-root reloaded

What's wrong in Move-to-front? Let's look at the possible cases of parent **and grandparent**

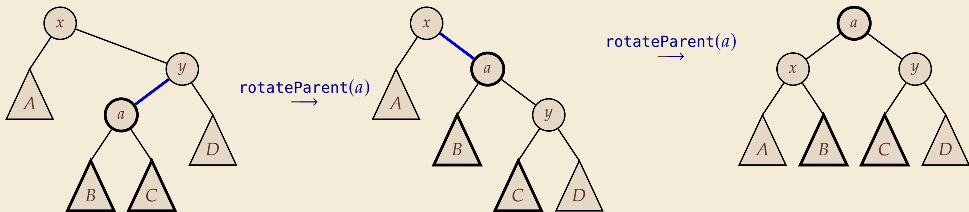


Using simple `rotateParent` calls, the subtree we came from (B or C) is lifted up!

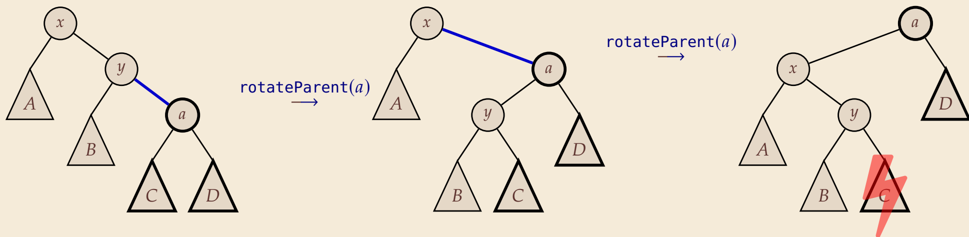


Move-to-root reloaded

What's wrong in Move-to-front? Let's look at the possible cases of parent **and grandparent**



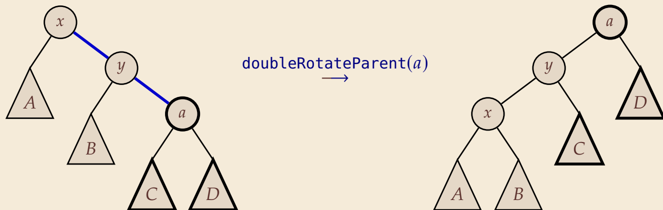
Using simple `rotateParent` calls, the subtree we came from (B or C) is lifted up!



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

The Fix

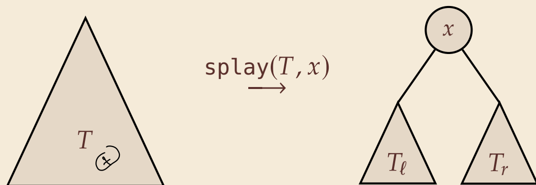
In the case that both parents are right children, do a double rotation



Now both C and D are lifted up!

Splay

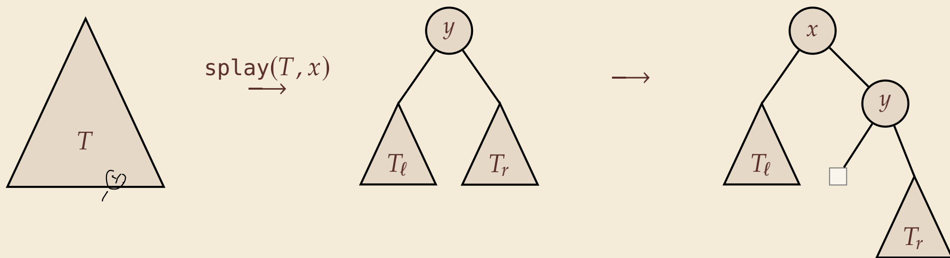
Aggressive restructuring allows to base all operations on single primitive: `splay`




```
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)
```

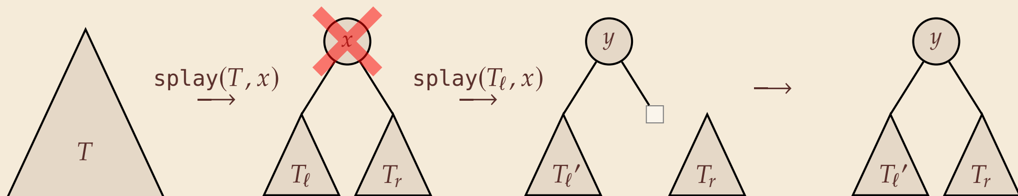
Splay – Insert

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



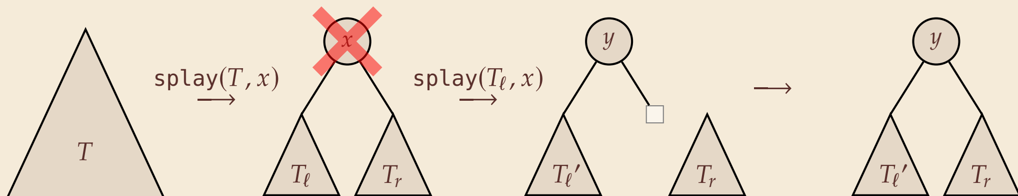
Splay – Delete

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



Splay – Delete

1. `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. `splay`(x, T_ℓ) $\rightsquigarrow$ root y = `max`(T_ℓ) and $y.right$ = `null`
4. $y.right := T_r$



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

`split` even more directly supported via `splay`: `split`(T, x): `splay`(T, x), return $(x.left, x)$

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 = **#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 **= #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 $op_1, op_2, op_3, \ldots, op_m$

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

a_i = amortized cost of op_i $:= c_i + \Delta\Phi_i \quad \leq B(n)$

$\Delta\Phi_i = \Phi_i - \Phi_{i-1}$

Φ = "potential" state of ds $\to \mathbb{R}$

Φ_i = potential after op$_i$ Φ_0 initial potential

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

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

Rotation Cost

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

splay walks back up search path

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

$\rightsquigarrow$ cost is dominated by **k = #rotations in `splay`** calls.

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

$\rightsquigarrow$ Can only hope for an ***amortized*** bound on k

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

$\Delta\Phi$ = change in potential

Goal: If `splay`(x) uses k rotations, then $k + \Phi' - \Phi \le 1 + 3 \cdot \big(r'(x) - r(x)\big)$

change in "rank" of x

$\rightsquigarrow$ Released potential can pay for actual cost k

Splay Tree Potential

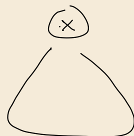
We choose a general method that will give several cool results.

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



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 $\boxed{s(x) \;:=\; \sum_{\substack{v \in T: \\ x \text{ ancestor of } v}} w(v)}$
- The ***rank*** of node x is $\boxed{r(x) \;:=\; \lg\big(s(x)\big)}$

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)
```

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

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

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

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 $\le 1 + 3\big(r'(x) - r(x)\big)$

(b) for a **Zig-Zig** is $\le 3\big(r'(x) - r(x)\big)$

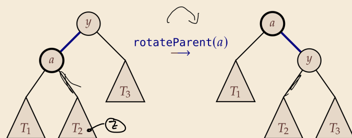
(c) for a **Zig-Zag** is $\le 3\big(r'(x) - r(x)\big)$

Access Lemma – Proof

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

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

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



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

$a = x$

$1 + \Phi' - \Phi$

real cost

only nodes outside T_1, T_2, T_3 change ranks

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

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

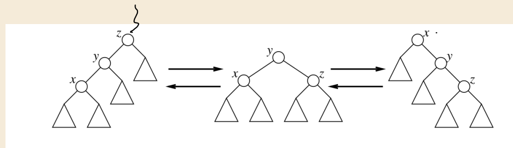
$\le 1 + 3(r'(a) - r(a))$

Access Lemma – Proof

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

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

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



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

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

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

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

$$\overset{!}{\leq} 3\big(r'(x) - r(x)\big)$$

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

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

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

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

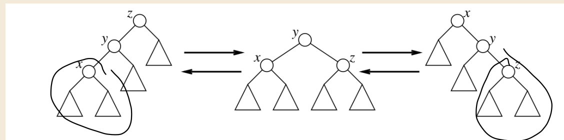
maximized $\quad x = y = \frac{1}{2}$

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

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

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

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



$s'(x)$ = entire subtree

subtrees of x before
z after

disjoint

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

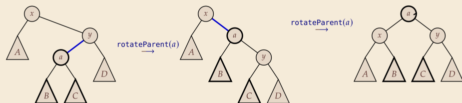
Access Lemma – Proof

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

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

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

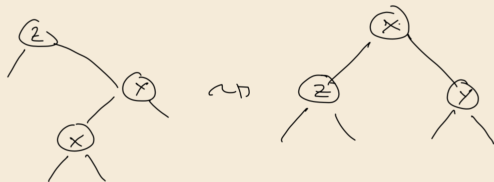
$s(x) := \sum_{\substack{v \in T: \\ x \text{ ancestor of } v}} w(v)$



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

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

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



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

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

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

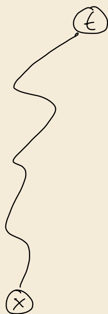
$s'(y) + s'(z) \leq s'(x)$

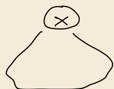
Splay Cost

Corollary 2.7

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

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





Lemma 2.6 (Access Lemma)

For **any** weakly positive node weights w, the amortized cost

(a) for a **Zig** is $\le 1 + 3\big(r'(x) - r(x)\big)$

(b) for a **Zig-Zig** is $\le 3\big(r'(x) - r(x)\big)$

(c) for a **Zig-Zag** is $\le 3\big(r'(x) - r(x)\big)$

Splay Cost

Corollary 2.7

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

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

Proof:

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

- $\rightsquigarrow$ The rank differences telescope.
- $\rightsquigarrow$ 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

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

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

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

Splay Cost

Corollary 2.7

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

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

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

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

Proof:

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

So $\Phi_m \;\le\; n \lg W$ and $\Phi_0 \;\ge\; \sum \lg w(x)$. ■

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 $= m \cdot O(\log n) + \Phi_m - \Phi_0$

$\Phi_m - \Phi_0 = O(n \log n)$

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 $X = x_1, \ldots, x_m$ 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 (i) $O\left(\log\left(\frac{1}{p_i}\right)\right)$

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

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

Corollary 2.7

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

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

$\Phi_m - \Phi_0 \le \sum_{i=1}^{n} \lg(1/p_i) \le m\mathcal{H}$

$m \ge n$

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

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

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

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

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

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

---

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

Splay Tree – Results [4]

For access sequence $X = x_1, \ldots, x_m$ 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 $\mathcal{O}\left(m + n \log n + \sum_{j=1}^{m} \log(1 + t(j))\right)$

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

$W = O(1)$

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

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

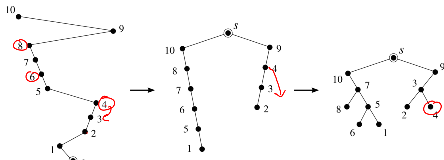


Fig. 2: A global view of splay trees. The transformation from the left to the middle illustrates rotate-to-root. The transformation from the left to the right illustrates splay trees.

Splay: Splay extends rotate-to-root: Let $s = v_0$, $v_1, \ldots v_k$ 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.*

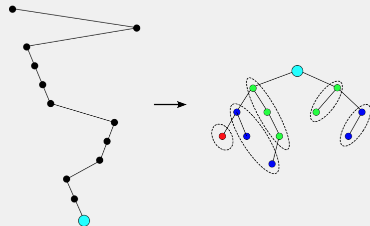
Intuition:

- 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: $\ell + z = \Omega(|P|)$

$\Longrightarrow$ Algorithm shares many known good properties of Splay.

László Kozma, *Binary search trees, rectangles and patterns*, slides, http://www.lkozma.net/research.html

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

$\rightsquigarrow$ 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

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

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

$\rightsquigarrow$ 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 $x_- = \texttt{predecessor}_T(x)$ and $x_+ = \texttt{successor}_T(x)$

Analysis of Updates

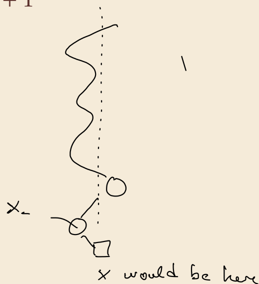
Theorem 2.13 (Splay Tree Update Lemma)

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

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

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

$3(r'(x) - r(x))$, $r'(x) = W$, $r(x) \geq \lg(w(x))$



Analysis of Updates

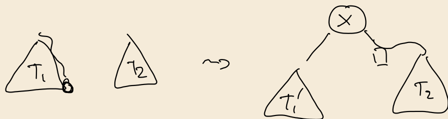
Theorem 2.13 (Splay Tree Update Lemma)

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

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

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

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



Analysis of Updates

Theorem 2.13 (Splay Tree Update Lemma)

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

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

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

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

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

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

Analysis of Updates

Theorem 2.13 (Splay Tree Update Lemma)

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

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

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

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

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

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

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

Analysis of Updates

Theorem 2.13 (Splay Tree Update Lemma)

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

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

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

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

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

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

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

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

Analysis of Updates – Proof

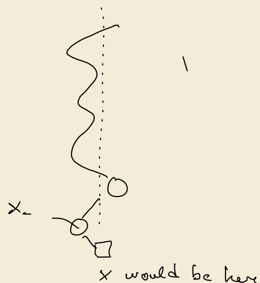
(a) $\text{access}_T(x)$ $x \in T$ from access lemma

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

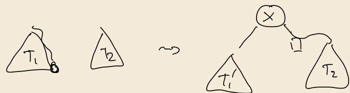
$\leq 3 \lg\left(\frac{W}{w(x)}\right) + 1$

(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

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

joining trees changes potential

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

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

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

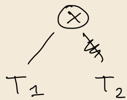
$\leq \lg(W) - \lg(T_1) \leq 3(\lg W - \lg(w(T_1))$

total amortized cost f

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

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

$x \in T$

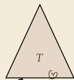


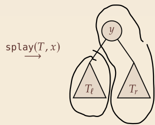
$\leq 3 \lg\left(\frac{W}{w(x)}\right) + O(1)$ am. cost

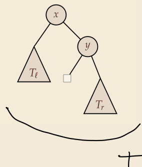
$\Delta\Phi \leq 0$

(e) $\text{split}_T(x)$ $x \notin T$ $\leadsto$ splay at x_- 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) $\text{insert}_T(x)$







T

$W = w(T)$

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

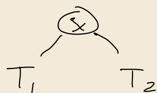
am. cost $\leq 3 \lg\left(\frac{W - w(x)}{w(y)}\right) + 1$

$\Delta\Phi$ y loses weight

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

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

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



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

□.

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** $\texttt{changeWeight}_T(x, \delta)$ **operation that sets** $w(x) := w(x) + \delta$

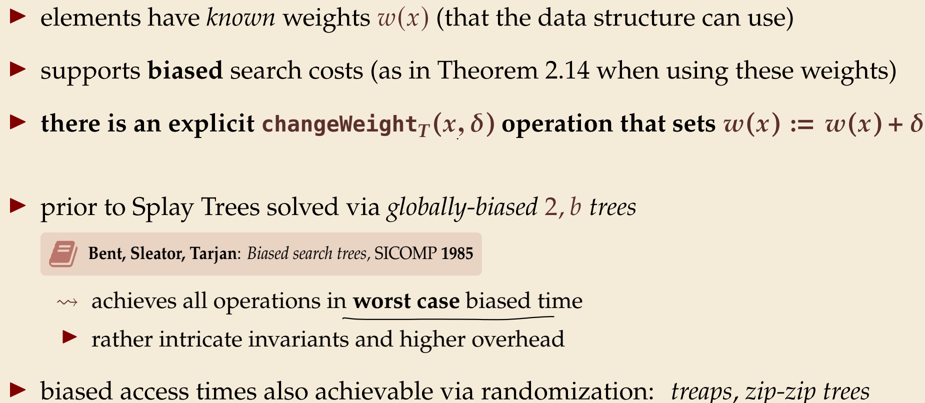
search costs $\lg\left(\frac{W}{w(x)} + 1\right)$

if $w(x) \ll \frac{1}{n}$ $\quad w(x) = 2^{-n}$ 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



Weight Changes

Lemma 2.14 (Weight Changes)

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

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

(b) $\texttt{changeWeight}_T(\texttt{root}(T), \delta)$ where $\delta < 0$: 0

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

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

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

$\leadsto$ 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 $x = x_1, \ldots, x_T$
- our algorithm $\mathcal{A}$ must produce output after each request
  At time t, output $y_t = \mathcal{A}(x_1, \ldots, x_t)$
- overall cost on instance depends on sequence of outputs
  $A(x) = \text{cost}\big(\mathcal{A}(x_1), \ldots, \mathcal{A}(x_1, \ldots, x_T)\big)$

Elewaiting as Sequence of Requests

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

$\rightsquigarrow$ **Elevator Problem**

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

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)

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 $w \le k - 1$ 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 $w \le k - 1$ and walk right away otherwise.

  $\rightsquigarrow \quad OPT(w) = \min\{w + 1, k\}$

Definition 2.15 (Competitive ratio)

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

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

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 $w \le k - 1$ 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

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

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

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

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 $w \le k-1$ 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

$$c = \max_x \frac{A(x)}{OPT(x)}$$ 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 $\max_{w \in [0..B)} \dfrac{k}{\min\{k, w+1\}} = k$
- "Always wait" has $\max_{w \in [0..B)} \dfrac{w+1}{\min\{k, w+1\}} = \dfrac{B}{k}$

Competitive Elewaiting

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

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

Competitive Elewaiting

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

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

⇝ Optimal choice for t

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

Competitive Elewaiting

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

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

⇝ Optimal choice for t

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

⇝ $\mathcal{A}_k$ 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

$\rightsquigarrow$ *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

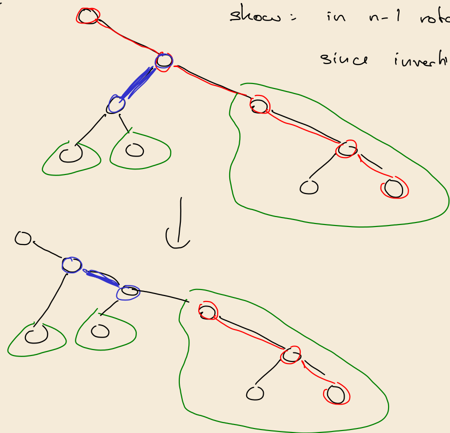
Rotation Distance & Initial Trees

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. ◀

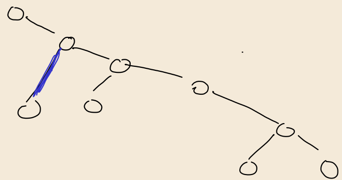
Note, actually $2n-6$ suffice.

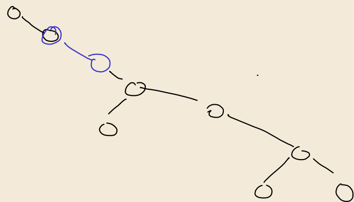
Proof:



show: in $n-1$ rotations can go to canonical tree

since invertible operation, this proves the stmt.





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 in 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

($\le 5^t$ 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 $n = 2^k$
  - $R_n = (\text{rev}_k(0), \text{rev}_k(1), \text{rev}_k(2), \ldots, \text{rev}_k(n-1))$, $\text{rev}_k(i) = i$ 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

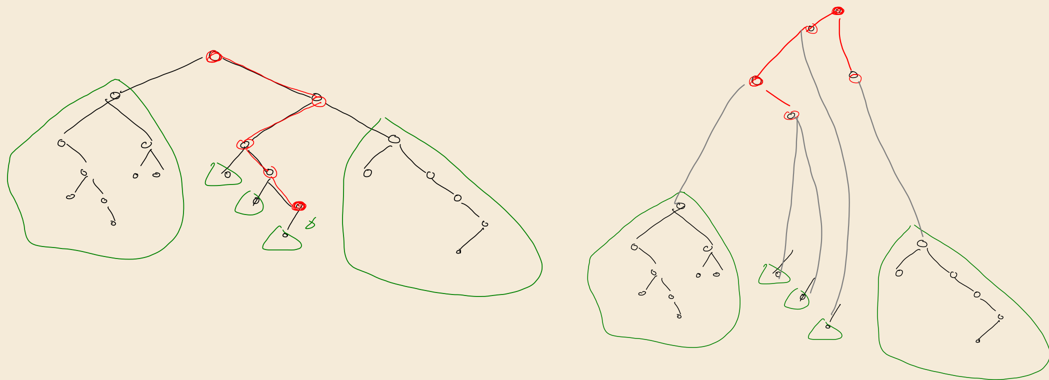
- Online BST algorithm specifically designed to stay close to one of Wilber's lower bounds
  details skipped
- achieves a competitive ratio of $O(\log \log n)$
- may use $\Omega(\log n \log \log n)$ amortized cost $\rightsquigarrow$ not $o(\log \log n)$-competitive

Demaine, Harmon, Iacono, Pătrașcu: *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

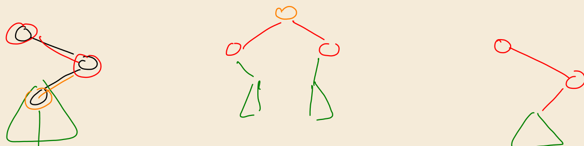
- Of course an **offline** method since it uses the future access times



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?*

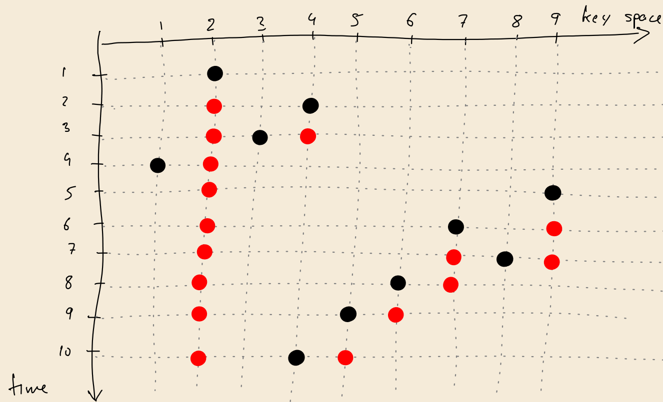
2.8 The Geometric View of BSTs

Different Way of Looking at BSTs

Beautiful reformulation of problem

Demaine, Harmon, Iacono, Kane, Pătrașcu: *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

$a_1, a_2, \dots, a_m$

⇝ 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, Pătrașcu: *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?)

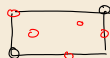
Arborally Satisfied Point Sets

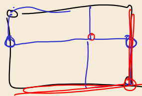
Each proper rectangle must contain another point.

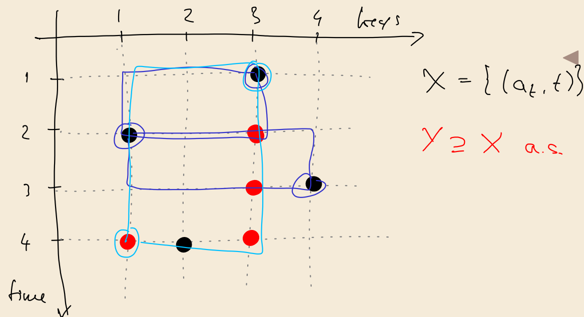
Definition 2.17 (Arborally Satisfied)

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







Arborally Satisfied Point Sets

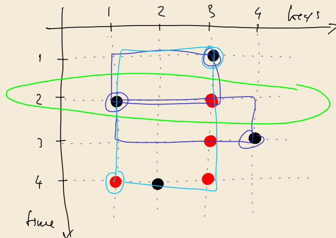
Each proper rectangle must contain another point.

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



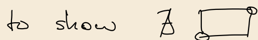
BST $\iff$ Arborally Satisfied

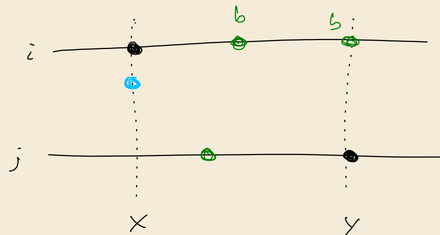
Theorem 2.18

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

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

Proof: "⇒" Let Y be the trace of a BST algorithm serving access $a_1, \ldots, a_m$





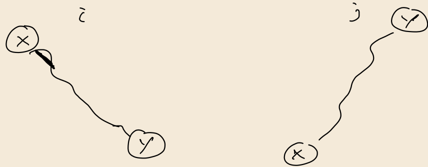
$b = lca(x, y)$ (changes over time)

$x \le b \le y$

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

⇒ 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

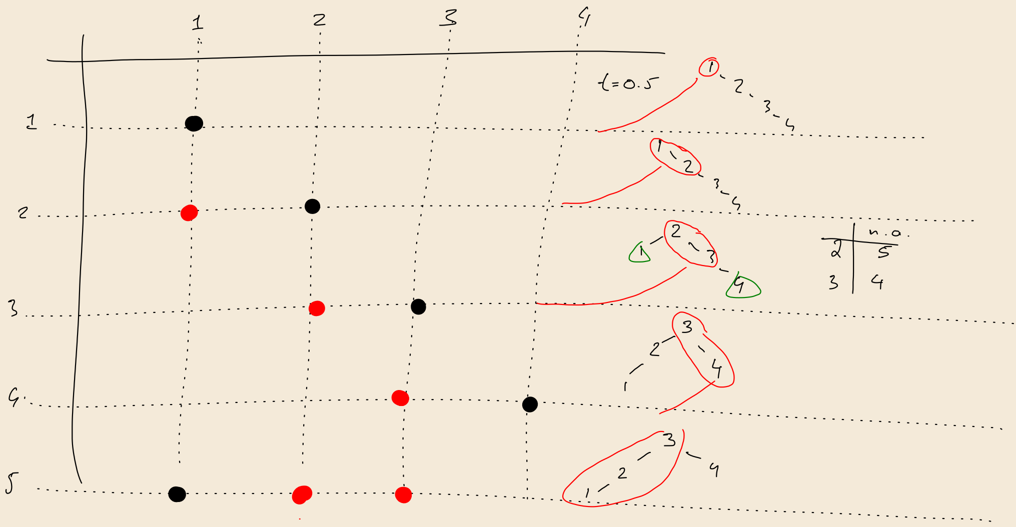
(must be possible to rearrange top part of tree)

at time t: touch top part $(Y \cap [n] \times \{t\})$

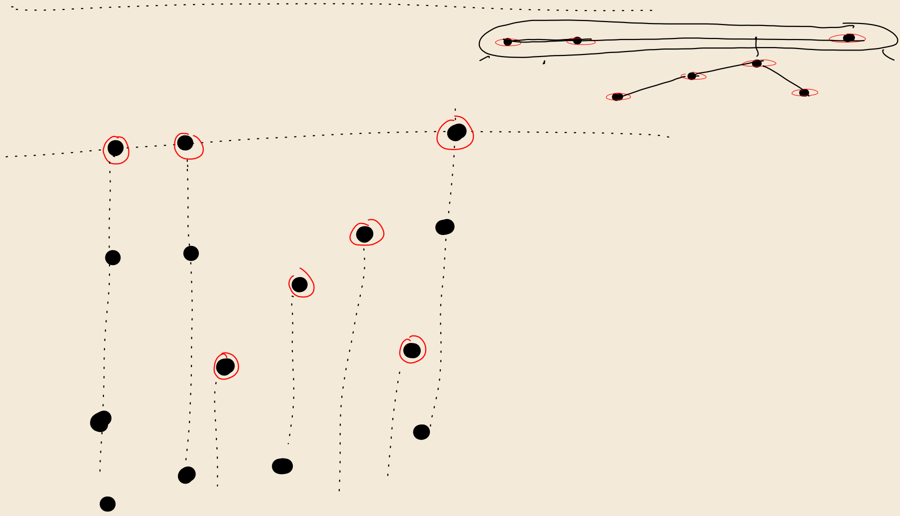
rearrange this as treap w/ priority = "next access time"

Invariant: between t and $t+1$ entire BST is a treap

w.r.t. next access time



1
2
3
4
1
2
3
4
5
t = 0.5
1 - 2 - 3 - 4
1 - 2 - 3 - 4
1 - 2 - 3 - 4
1 - 2 - 3 - 4
1 - 2 - 3 - 4
n.a.
2 5
3 4



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

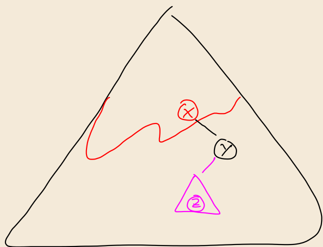
towards a contradiction, assume heap property violated (earliest such)

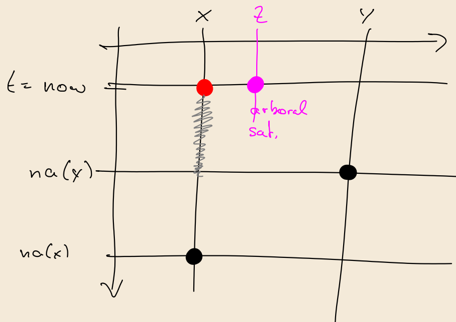
$\Rightarrow$ exists parent – child pair $x - y$ where na(y) earlier than na(x) ($<$)

case 1: both in top part ↯ we arrange top part as tree

case 2: both not in top part ↯ earlier

case 3: x in top part, y not





since Y arb. sat. must have point (z, t) touched now but y on path to z ↯

□.

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 $x_1, \ldots, x_m$
- cost of BST algorithm corresponding to Y: $|Y|$

Minimum Arborally Satisfied Superset

for general point set X
MinASS is NP-hard
(X has y-ties)

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 $x_1, \ldots, x_m$
- cost of BST algorithm corresponding to Y: $|Y|$
- If we're given all of X up front $\rightsquigarrow$ offline (BST) algorithm
- If X is revealed one time-slice X_t at a time $\rightsquigarrow$ online algorithm for geometric view

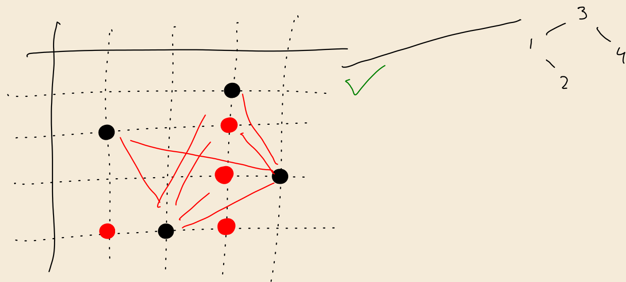
If (and only if) any BST algorithm is dynamically optimal ($O(1)$-competitive)
then can find $O(1)$-approximation for smallest arb. sat. superset of X online
(aka have a $O(1)$-competitive online ASS algorithm)

Greedy Arborally Satisfied Superset

Natural Greedy Algorithm: GreedyASS

- at time t, add the points now (current time row) that are necessary to satisfy $Y \cup X_t$

$\rightsquigarrow$ geometric sweep-line algorithm



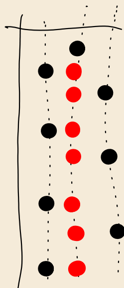
Greedy Arborally Satisfied Superset

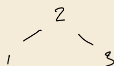
Natural Greedy Algorithm: GreedyASS

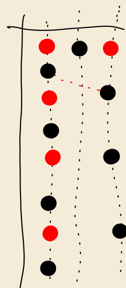
- at time t, add the points now (current time row) that are necessary to satisfy $Y \cup X_t$

$\rightsquigarrow$ geometric sweep-line algorithm

- Seems to work very well
- but easy to see that it has approximation ratio $\geq \frac{3}{2}$







Greedy Arborally Satisfied Superset

Natural Greedy Algorithm: GreedyASS

- at time t, add the points now (current time row) that are necessary to satisfy $Y \cup X_t$

$\rightsquigarrow$ geometric sweep-line algorithm

- Seems to work very well
- but easy to see that it has approximation ratio $\geq \frac{3}{2}$

Who controls the Future

Amazing fact: *GreedyASS translated to BSTs* ***is*** *GreedyFuture!* 🤯

- to make the treap-construction effective, we need the future accesses (!) . . .
- but can delay actual treapify until next access comes
- can simulate this in BST (*"split trees"*) in constant amortized time

Who controls the Future

Amazing fact: *GreedyASS translated to BSTs* ***is*** *GreedyFuture!* 🤯

- to make the treap-construction effective, we need the future accesses (!) ...
- but can delay actual treapify until next access comes
- can simulate this in BST (*"split trees"*) in constant amortized time
- GreedyASS is an **online** BST algorithm (can be implemented as)

⇝ New contender for a dynamically optimal algorithm!

Who controls the Future

Amazing fact: *GreedyASS translated to BSTs* ***is*** *GreedyFuture!* 🤯

- to make the treap-construction effective, we need the future accesses (!) ...
- but can delay actual treapify until next access comes
- can simulate this in BST (*"split trees"*) in constant amortized time
- GreedyASS is (can be implemented as) an **online** BST algorithm

⇝ New contender for a dynamically optimal algorithm!

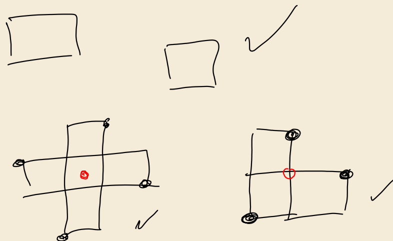
👍 Indeed, by now more positive results known about Greedy than Splay

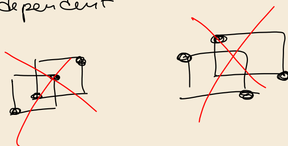
👎 probably mostly of theoretical interest as a BST algorithm itself

Lower Bounds

Geometric view also yields lower bounds on $OPT(X) = \min\{|Y| : Y \supset X, Y \text{ a.s.s.}\}$

- Independent rectangles (in X): no rectangle contains other rectangle's **corner** in its **inside**





Lower Bounds

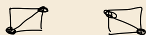
Geometric view also yields lower bounds on $OPT(X) = \min\{|Y| : Y \supset X, Y \text{ a.s.s.}\}$

- Independent rectangles (in X): no rectangle contains other rectangle's **corner** in its **inside**

Theorem 2.19

$OPT(X) \geq |X| + \frac{1}{2} \max |I(X)|$ with max over set of *independent rectangles* $I(X)$.

Proof: Consider separately



+ -rectangles − rectangle

Assumption in X, all (x and) y coordinates are distinct.

⧄-satisfied iff all ⧄ rectangles arb. sat. (ignore ⧅)

$OPT_{⧄}$ = smallest $Y \supseteq X$ that is ⧄-satisfied

Lemma: $OPT_{⧄} \geq |X| + \max |I_{⧄}|$ $I_{⧄}$ is set of independent ⧄ rectangles

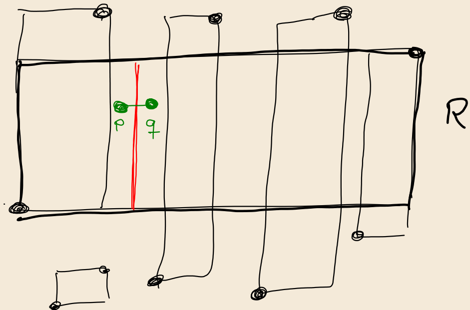
① Find rectangle $R \in I_{⧄}$ that is widest and unique v. line inside R

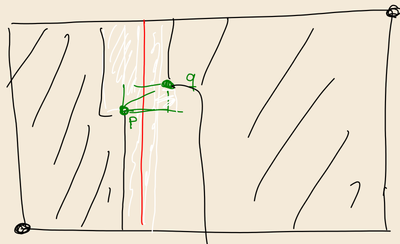
(2) Find horizontally adjacent points inside R

↳ charge R to there

p is topmost rightmost point inside in $OPT_{\boxslash}$ left of red line

q bottommost leftmost inside R in $OPT_{\boxslash}$ north of p



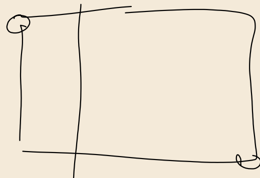


⇒ p and q must be on same y-line

⇒ charge R to (p, q), continue on remaining indep. rectangles

(p, q) cannot be charged ever again

because p, q are on same y-line $\{p, q\} \notin X$



$\square$.

Similarly, for ◩ - indep. rectangles.

Since $I = I_{◪} \dot{\cup} I_{◩}$ $\max\{|I_{◪}|, |I_{◩}|\} \geq \frac{1}{2} \cdot |I|$

$\square$.

Independent Rectangle Approximation

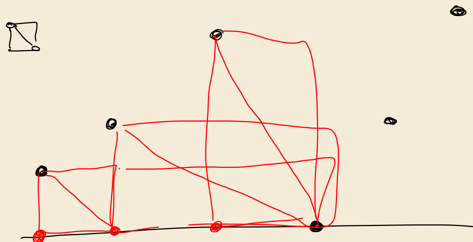
Approximate lower bound

- Computing $\max|I(X)|$ is not easy in general
- proof shows that signed independent rectangles suffice

⇝ Signed Greedy: satisfy only positive resp. only negative independent rectangles

- two valid lower bounds, use larger one!

same sweepline ↓ as in Greedy ASS
but only satisfy new (resp. new)
rectangles



Independent Rectangle Approximation

Approximate lower bound

- Computing $\max |I(X)|$ is not easy in general
- proof shows that signed independent rectangles suffice

$\leadsto$ Signed Greedy: satisfy only positive resp. only negative independent rectangles

- two valid lower bounds, use larger one!

- Can prove: Signed Greedy gives constant factor approximation to best independent-rectangle bound

$\neq$ constant factor within OPT (at least not known!)

OPEN: Is GreedyASS $O(1)$-competitive? at least $o(\log n)$ competitive?

2.9 Deferred Data Structures

Motivation

All our adaptive dictionaries so far were binary search trees.

Motivation

All our adaptive dictionaries so far were binary search trees.

$\rightsquigarrow$ *We may be able to structure them to have cheap access to "important" items . . .*

But we inevitably keep all elements of the dictionary perfectly sorted (BST!)

Motivation

All our adaptive dictionaries so far were binary search trees.

$\rightsquigarrow$ *We may be able to structure them to have cheap access to "important" items . . .*

But we inevitably keep all elements of the dictionary perfectly sorted (BST!)

$\rightsquigarrow$ Most insertions must cost $\Omega(\log n)$

- Reduction from sorting
- BST-Sort: Insert all n items & output in inorder traversal

Motivation

All our adaptive dictionaries so far were binary search trees.

$\rightsquigarrow$ *We may be able to structure them to have cheap access to "important" items . . .*

But we inevitably keep all elements of the dictionary perfectly sorted (BST!)

$\rightsquigarrow$ Most insertions must cost $\Omega(\log n)$

- Reduction from sorting
- BST-Sort: Insert all n items & output in inorder traversal

How can you possibly do better in a comparison-based model?

- Seems that we need sortedness for any fast queries, so sorting inevitable

Motivation

All our adaptive dictionaries so far were binary search trees.

$\rightsquigarrow$ *We may be able to structure them to have cheap access to "important" items . . .*

But we inevitably keep all elements of the dictionary perfectly sorted (BST!)

$\rightsquigarrow$ Most insertions must cost $\Omega(\log n)$

- Reduction from sorting
- BST-Sort: Insert all n items & output in inorder traversal

How can you possibly do better in a comparison-based model?

- Seems that we need sortedness for any fast queries, so sorting inevitable

$\rightsquigarrow$ True. What if we never query (certain elements), though?

- Insertion-heavy workloads rather common scenario in applications
- many insertions, few queries $\rightsquigarrow$ most elements never queried for! *So why sort them?*

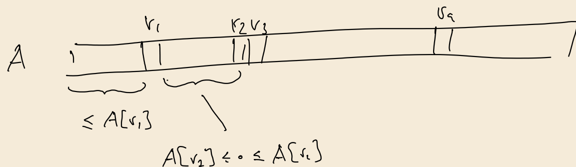
The Multiple Selection Problem

- **Selection by Rank:**
  - Given: n unsorted elements $x_1, \ldots, x_n$,
  - Goal: find the rth smallest $x_{(r)}$ (at (1-based) index r after sorting)

The Multiple Selection Problem

- **Selection by Rank:**
  - Given: n unsorted elements $x_1, \ldots, x_n$,
  - Goal: find the rth smallest $x_{(r)}$ (at (1-based) index r after sorting)

- **Multiple Selection**:
  - Given: n unsorted elements $x_1, \ldots, x_n$,
  - Goal: find q elements of ranks $r_1 < r_2 < \cdots < r_q$ from unsorted elements $x_1, \ldots, x_n$
    - $\rightsquigarrow$ Report sought elements $x_{(r_1)}, \ldots, x_{(r_q)}$ in sorted order



The Multiple Selection Problem

- **Selection by Rank:**
  - Given: n unsorted elements $x_1, \ldots, x_n$,
  - Goal: find the rth smallest $x_{(r)}$ (at (1-based) index r after sorting)

- **Multiple Selection**:
  - Given: n unsorted elements $x_1, \ldots, x_n$,
  - Goal: find q elements of ranks $r_1 < r_2 < \cdots < r_q$ from unsorted elements $x_1, \ldots, x_n$

    $\rightsquigarrow$ Report sought elements $x_{(r_1)}, \ldots, x_{(r_q)}$ in sorted order

- **Example:**

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}
67	30	45	33	15	99	26	90	55	9	96	45	95	31	3

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	9	15	26	30	31	33	45	45	55	67	90	95	96	99
↑ $r_1 = 1$	↑ $r_2 = 2$	↑ $r_3 = 3$					↑ $r_4 = 8$							

The Multiple Selection Problem

- **Selection by Rank:**
  - Given: n unsorted elements $x_1, \ldots, x_n$,
  - Goal: find the rth smallest $x_{(r)}$ (at (1-based) index r after sorting)

- **Multiple Selection**:
  - Given: n unsorted elements $x_1, \ldots, x_n$,
  - Goal: find q elements of ranks $r_1 < r_2 < \cdots < r_q$ from unsorted elements $x_1, \ldots, x_n$

    $\rightsquigarrow$ Report sought elements $x_{(r_1)}, \ldots, x_{(r_q)}$ in sorted order

- **Example:**

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}
67	30	45	33	15	99	26	90	55	9	96	45	95	31	3

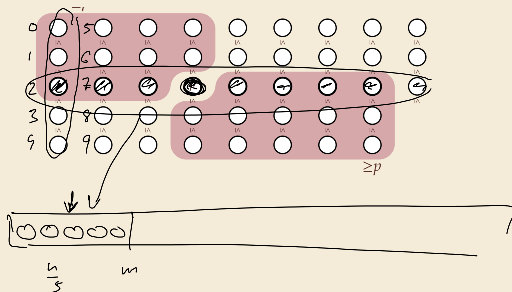
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	9	15	26	30	31	33	45	45	55	67	90	95	96	99
$\uparrow r_1 = 1$	$\uparrow r_2 = 2$	$\uparrow r_3 = 3$					$\uparrow r_4 = 8$							

Answer: 3, 9, 15, 45

Recall: The Median-of-Medians Algorithm

```
 1 procedure choosePivotMoM(A[l..r)):
 2     m := ⌊n/5⌋
 3     for i := 0, ..., m − 1
 4         sort(A[5i..5i + 4])   // O(n) time overall
 5         // collect median of 5
 6         Swap A[i] and A[5i + 2]
 7     return quickselectMoM(A[0..m), ⌊(m−1)/2⌋)
 8
 9 procedure quickselectMoM(A[l..r), k):
10     if r − l ≤ 1 then return A[l]
11     b := choosePivotMoM(A[l..r))   // use random element here → quickselect
12     j := partition(A[l..r), b)
13     if j == k
14         return A[j]
15     else if j < k
16         quickselectMoM(A[j + 1..r), k)
17     else // j > k
18         quickselectMoM(A[l..j), k)
```

≤ A[j]   j   ≥ A[j]



Recall: The Median-of-Medians Algorithm

```
1  procedure choosePivotMoM(A[l..r)):
2      m := ⌊n/5⌋
3      for i := 0, ..., m − 1
4          sort(A[5i..5i + 4])
5          // collect median of 5
6          Swap A[i] and A[5i + 2]
7      return quickselectMoM(A[0..m), ⌊(m−1)/2⌋)
8
9  procedure quickselectMoM(A[l..r), k):
10     if r − l ≤ 1 then return A[l]
11     b := choosePivotMoM(A[l..r))
12     j := partition(A[l..r), b)
13     if j == k
14         return A[j]
15     else if j < k
16         quickselectMoM(A[j + 1..r), k)
17     else // j > k
18         quickselectMoM(A[l..j), k)
```

Analysis:

- Note: 2 mutually recursive procedures
  $\rightsquigarrow$ effectively 2 recursive calls!

1. recursive call inside choosePivotMoM on $m \le \frac{n}{5}$ elements

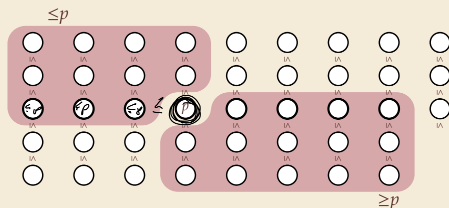
Recall: The Median-of-Medians Algorithm

```
1  procedure choosePivotMoM(A[l..r]):
2      m := ⌊n/5⌋
3      for i := 0, ..., m − 1
4          sort(A[5i..5i + 4])
5          // collect median of 5
6          Swap A[i] and A[5i + 2]
7      return quickselectMoM(A[0..m), ⌊(m−1)/2⌋)
8
9  procedure quickselectMoM(A[l..r), k):
10     if r − l ≤ 1 then return A[l]
11     b := choosePivotMoM(A[l..r))
12     j := partition(A[l..r), b)
13     if j == k
14         return A[j]
15     else if j < k
16         quickselectMoM(A[j + 1..r), k)
17     else // j > k
18         quickselectMoM(A[l..j), k)
```

Analysis:

- Note: 2 mutually recursive procedures
  ⇝ effectively 2 recursive calls!

1. recursive call inside choosePivotMoM on $m \le \frac{n}{5}$ elements
2. recursive call inside quickselectMoM



⇝ partition excludes $\sim 3 \cdot \frac{m}{2} \sim \frac{3}{10}n$ elem.

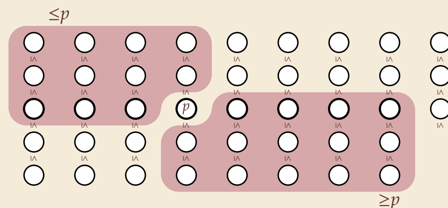
Recall: The Median-of-Medians Algorithm

```
1  procedure choosePivotMoM(A[l..r)):
2      m := ⌊n/5⌋
3      for i := 0, ..., m − 1
4          sort(A[5i..5i + 4])
5          // collect median of 5
6          Swap A[i] and A[5i + 2]
7      return quickselectMoM(A[0..m), ⌊(m−1)/2⌋)
8
9  procedure quickselectMoM(A[l..r), k):
10     if r − l ≤ 1 then return A[l]
11     b := choosePivotMoM(A[l..r))
12     j := partition(A[l..r), b)
13     if j == k
14         return A[j]
15     else if j < k
16         quickselectMoM(A[j + 1..r), k)
17     else // j > k
18         quickselectMoM(A[l..j), k)
```

Analysis:

- Note: 2 mutually recursive procedures
  $\rightsquigarrow$ effectively 2 recursive calls!

1. recursive call inside choosePivotMoM
   on $m \le \frac{n}{5}$ elements
2. recursive call inside quickselectMoM



$\rightsquigarrow$ partition excludes $\sim 3 \cdot \frac{m}{2} \sim \frac{3}{10}n$ elem.

$\rightsquigarrow C(n) \;\le\; \Theta(n) + C(\frac{1}{5}n) + C(\frac{7}{10}n)$

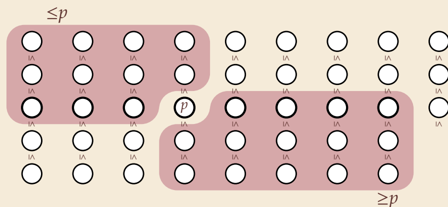
Recall: The Median-of-Medians Algorithm

```
1  procedure choosePivotMoM(A[l..r]):
2      m := ⌊n/5⌋
3      for i := 0, ..., m − 1
4          sort(A[5i..5i + 4])
5          // collect median of 5
6          Swap A[i] and A[5i + 2]
7      return quickselectMoM(A[0..m), ⌊(m−1)/2⌋)
8
9  procedure quickselectMoM(A[l..r], k):
10     if r − l ≤ 1 then return A[l]
11     b := choosePivotMoM(A[l..r))
12     j := partition(A[l..r), b)
13     if j == k
14         return A[j]
15     else if j < k
16         quickselectMoM(A[j + 1..r), k)
17     else // j > k
18         quickselectMoM(A[l..j), k)
```

Analysis:

- Note: 2 mutually recursive procedures
  $\rightsquigarrow$ effectively 2 recursive calls!

1. recursive call inside choosePivotMoM on $m \le \frac{n}{5}$ elements
2. recursive call inside quickselectMoM



$\rightsquigarrow$ partition excludes $\sim 3 \cdot \frac{m}{2} \sim \frac{3}{10}n$ elem.

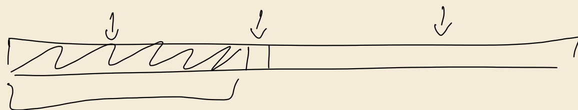
$$\rightsquigarrow \quad C(n) \le \Theta(n) + C(\tfrac{1}{5}n) + C(\tfrac{7}{10}n)$$
$$\le \Theta(n) + C(\tfrac{1}{5}n + \tfrac{7}{10}n)$$
$$= \Theta(n) + C(\tfrac{9}{10}n) \quad \rightsquigarrow \quad C(n) = \Theta(n)$$

ansatz: overall cost linear

Multiple Selection – Multiple Solutions

Several algorithmic ideas possible:

1. q **calls** to selection algorithm (quickselect, median of medians) $\rightsquigarrow$ $O(qn)$
   clearly wasteful

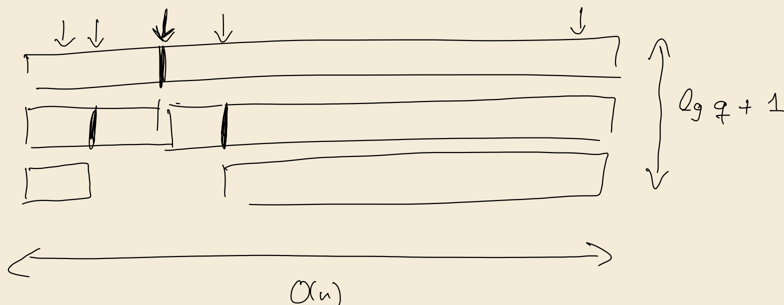


Multiple Selection – Multiple Solutions

Several algorithmic ideas possible:

1. q **calls** to selection algorithm (quickselect, median of medians) $\rightsquigarrow$ $O(qn)$
   clearly wasteful

2. **Divide & Conquer over Ranks**:
   (single) **select $r_{\lceil q/2 \rceil}$-th** smallest and partition $x_1, \ldots, x_n$ around it.
   Recursively select $r_1, \ldots, r_{\lceil q/2 \rceil - 1}$ and $r_{\lceil q/2 \rceil + 1}, \ldots, r_q$



Multiple Selection – Multiple Solutions

Several algorithmic ideas possible:

1. q **calls** to selection algorithm (quickselect, median of medians) $\rightsquigarrow$ $O(qn)$
   clearly wasteful

2. **Divide & Conquer over Ranks**:
   (single) **select** $r_{\lceil q/2 \rceil}$**-th** smallest and partition $x_1, \ldots, x_n$ around it.
   Recursively select $r_1, \ldots, r_{\lceil q/2 \rceil - 1}$ and $r_{\lceil q/2 \rceil + 1}, \ldots, r_q$
   $\rightsquigarrow$ $O(n \lg q)$

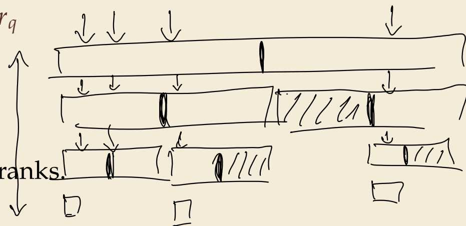
Multiple Selection – Multiple Solutions

Several algorithmic ideas possible:

1. q **calls** to selection algorithm (quickselect, median of medians) $\rightsquigarrow$ $O(qn)$
   clearly wasteful

2. **Divide & Conquer over Ranks**:
   (single) **select** $r_{\lceil q/2 \rceil}$**-th** smallest and partition $x_1, \ldots, x_n$ around it.
   Recursively select $r_1, \ldots, r_{\lceil q/2 \rceil - 1}$ and $r_{\lceil q/2 \rceil + 1}, \ldots, r_q$
   $\rightsquigarrow$ $O(n \lg q)$

3. **Divide & Conquer over Elements:** $\le \lg n$
   Find **median of $x_1, \ldots, x_n$** and split query ranks.
   Recurse on subproblem only of it contains query ranks.



after $\lceil \lg q \rceil + 1$ median rounds

$\Rightarrow$ $2q$ subproblems, all of size $\frac{n}{2q}$

$\Rightarrow$ $\ge q$ subproblems w/o query ranks $\Rightarrow$ $\ge \frac{n}{2}$ elements in subproblems w/o ranks

total cost $O(n \cdot \lg q)$

$\Rightarrow$ don't recurse on those

$$T(n) \leq O(n \lg q) + T\left(\frac{n}{2}\right)$$

$$T(n) = O(n \lg q)$$

Multiple Selection – Multiple Solutions

Several algorithmic ideas possible:

1. q **calls** to selection algorithm (quickselect, median of medians) $\rightsquigarrow$ $O(qn)$
   clearly wasteful

2. **Divide & Conquer over Ranks**:
   (single) **select** $\boldsymbol{r_{\lceil q/2 \rceil}}$**-th** smallest and partition $x_1, \ldots, x_n$ around it.
   Recursively select $r_1, \ldots, r_{\lceil q/2 \rceil - 1}$ and $r_{\lceil q/2 \rceil + 1}, \ldots, r_q$
   $\rightsquigarrow$ $O(n \lg q)$

3. **Divide & Conquer over Elements:**
   Find **median of** $\boldsymbol{x_1, \ldots, x_n}$ and split query ranks.
   Recurse on subproblem only of it contains query ranks.
   $\rightsquigarrow$ $O(n \lg q)$

Multiple Selection – Multiple Solutions

Several algorithmic ideas possible:

1. q **calls** to selection algorithm (quickselect, median of medians) $\rightsquigarrow$ $O(qn)$
   clearly wasteful

2. **Divide & Conquer over Ranks**:
   (single) **select** $r_{\lceil q/2 \rceil}$**-th** smallest and partition $x_1, \ldots, x_n$ around it.
   Recursively select $r_1, \ldots, r_{\lceil q/2 \rceil - 1}$ and $r_{\lceil q/2 \rceil + 1}, \ldots, r_q$
   $\rightsquigarrow$ $O(n \lg q)$

3. **Divide & Conquer over Elements:**
   Find **median of** $x_1, \ldots, x_n$ and split query ranks.
   Recurse on subproblem only of it contains query ranks.
   $\rightsquigarrow$ $O(n \lg q)$

Can also show: $\Omega(n \lg q)$ comparisons needed in worst case. $\rightsquigarrow$ *Case closed?*

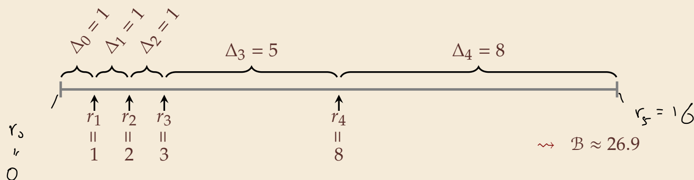
Multiple Selection – Lower Bound

... not if we put our adaptive-analysis glasses on!

"Gap Entropy":

$$\mathcal{B} = \sum_{i=1}^{q+1} \Delta_i \log_2 \frac{n}{\Delta_i}$$

with $\Delta_i = r_i - r_{i-1}$ $\quad (1 \le i \le q+1,\ r_0 = 0$ and $r_{q+1} = n)$



Theorem 2.20 (Multiple Selection Lower Bound)

Multiple selection requires $\ge \mathcal{B} - O(n)$ comparisons and $\Omega(\mathcal{B} + n)$ time in the worst and average case over inputs of n elements and query ranks $r_1 < \cdots < r_q$.

$$\mathcal{B} \geq q \cdot 1 \cdot \lg\left(\frac{n}{1}\right) + (n-q) \cdot \lg\left(\frac{n}{n-q}\right)$$

$$= q \cdot \lg n + (n-q) \cdot O\left(\frac{q}{n-q}\right) \qquad \text{“} \lg\left(1 + \frac{q}{n-q}\right)$$

$$\approx q \ln n + O(q) \qquad q \ll n$$

Proof.

$$\mathcal{B} = \sum_{i=1}^{q+1} \Delta_i \log_2 \frac{n}{\Delta_i}$$

Multiple Selection – Lower Bound Proof

Proof:

Suppose we have multiple-selection algorithm $\mathcal{A}$.

If we run $\mathcal{A}$ using $r_1 < \cdots < r_q$, we get output $x_{(r_1)}, \ldots, x_{(r_q)}$.

For $\mathcal{A}$ to be correct, it needs to learn how every other element compares to $x_{(r_1)}, \ldots, x_{(r_q)}$.

Multiple Selection – Lower Bound Proof

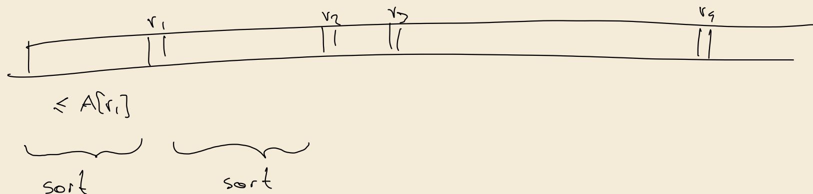
Proof:

Suppose we have multiple-selection algorithm $\mathcal{A}$.

If we run $\mathcal{A}$ using $r_1 < \cdots < r_q$, we get output $x_{(r_1)}, \ldots, x_{(r_q)}$.

For $\mathcal{A}$ to be correct, it needs to learn how every other element compares to $x_{(r_1)}, \ldots, x_{(r_q)}$.

We can thus build a *sorting algorithm* for $x_1, \ldots, x_n$ by sorting the $\Delta_i - 1$ elements between $x_{(r_{i-1})}$ and $x_{(r_i)}$, for $i = 1, \ldots, q$.



Multiple Selection – Lower Bound Proof

Proof:

Suppose we have multiple-selection algorithm $\mathcal{A}$.

If we run $\mathcal{A}$ using $r_1 < \cdots < r_q$, we get output $x_{(r_1)}, \ldots, x_{(r_q)}$.

For $\mathcal{A}$ to be correct, it needs to learn how every other element compares to $x_{(r_1)}, \ldots, x_{(r_q)}$.

We can thus build a *sorting algorithm* for $x_1, \ldots, x_n$ by sorting the $\Delta_i - 1$ elements between $x_{(r_{i-1})}$ and $x_{(r_i)}$, for $i = 1, \ldots, q$.

Since sorting is subject to the $\lg(n!)$ lower bound, we must use $\geq \lg(n!)$ in total.

For $\mathcal{A}$'s comparisons, this means

#comparisons

Multiple Selection – Lower Bound Proof

Proof:

Suppose we have multiple-selection algorithm $\mathcal{A}$.
If we run $\mathcal{A}$ using $r_1 < \cdots < r_q$, we get output $x_{(r_1)}, \ldots, x_{(r_q)}$.
For $\mathcal{A}$ to be correct, it needs to learn how every other element compares to $x_{(r_1)}, \ldots, x_{(r_q)}$.

We can thus build a *sorting algorithm* for $x_1, \ldots, x_n$ by sorting the $\Delta_i - 1$ elements between $x_{(r_{i-1})}$ and $x_{(r_i)}$, for $i = 1, \ldots, q$.

Since sorting is subject to the $\lg(n!)$ lower bound, we must use $\geq \lg(n!)$ in total.
For $\mathcal{A}$'s comparisons, this means

$$\#\text{comparisons} \geq \lg(n!) - \sum_{i=1}^{q+1} \lg((\Delta_i - 1)!)$$

Multiple Selection – Lower Bound Proof

Proof:
Suppose we have multiple-selection algorithm $\mathcal{A}$.
If we run $\mathcal{A}$ using $r_1 < \cdots < r_q$, we get output $x_{(r_1)}, \ldots, x_{(r_q)}$.
For $\mathcal{A}$ to be correct, it needs to learn how every other element compares to $x_{(r_1)}, \ldots, x_{(r_q)}$.

We can thus build a *sorting algorithm* for $x_1, \ldots, x_n$ by sorting the $\Delta_i - 1$ elements between $x_{(r_{i-1})}$ and $x_{(r_i)}$, for $i = 1, \ldots, q$.

Since sorting is subject to the $\lg(n!)$ lower bound, we must use $\geq \lg(n!)$ in total.
For $\mathcal{A}$'s comparisons, this means

$$\begin{aligned}
\#\text{comparisons} \;&\geq\; \lg(n!) - \sum_{i=1}^{q+1} \lg((\Delta_i - 1)!) \\
&=\; n \lg n \pm O(n) - \sum_{i=1}^{q+1} (\Delta_i - 1)\lg(\Delta_i - 1) \pm O(\Delta_i)
\end{aligned}$$

Multiple Selection – Lower Bound Proof [2]

Proof (cont.):

using $(x-1)\lg(x-1) = x\lg x \pm O(\lg x)$ as $x \to \infty$, we get

$$\lg(1+x) = O(x)$$

Multiple Selection – Lower Bound Proof [2]

Proof (cont.):

using $(x-1)\lg(x-1) = x\lg x \pm O(\lg x)$ as $x \to \infty$, we get

$$\begin{aligned}\#\text{comparisons} &\ge n\lg n - \sum_{i=1}^{q+1} \Delta_i \lg(\Delta_i) \pm O(\Delta_i + \lg \Delta_i) \pm O(n) \\ &= \sum_{i=1}^{q+1} \Delta_i \lg(n/\Delta_i) \pm O(n).\end{aligned}$$

■

An Adaptive Multiple-Selection Algorithm

Our last approach, **Divide & Conquer over Elements**, is indeed **optimal** (up to constant factors)

```
1 procedure MultiSelect(X[1..n], R[1..q]):
2     if R == ∅ return
3     p := median(X[1..n]) // linear-time selection
4     X_1, X_2 := partition(X[1..n], p)
5     r := |X_1| + 1 // rank of p in X
6     R_1, R_2 := partition(R[1..q], r)
7     MultiSelect(X_1, R_1)
8     if r ∈ R then report p end if
9     MultiSelect(X_2, R_2)
```

Theorem 2.21 (MultiSelect Analysis)

Divide & Conquer over Elements using linear-time median selection finds ranks $r_1 < \cdots < r_q$ among n unsorted elements in $O(\mathcal{B} + n)$ time. ◀

Proof:

We charge to Δ_i

on level $d = 0, 1, \ldots,$

$\min\{\Delta_i, 2 \cdot (\frac{1}{2})^d \cdot n\}$

Subproblems on level d

have $(\frac{1}{2})^d \cdot n$

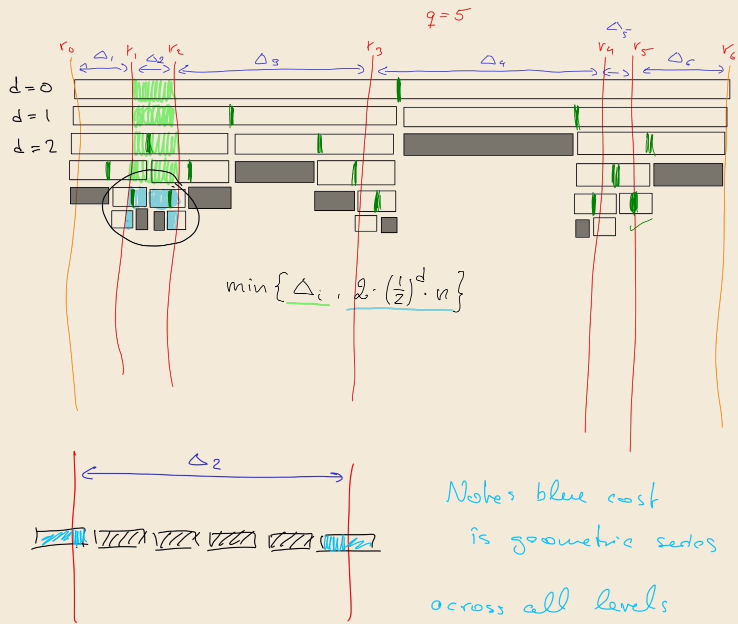
$\Rightarrow$ all partitioning paid for

Note: only charge

green term if $\Delta_i \leq 2(\frac{1}{2})^d n$

$\Leftrightarrow d \leq \lg\left(\frac{2n}{\Delta_i}\right)$

$= \lg\left(\frac{n}{\Delta_i}\right) + 1$



$$C \le \sum_{i=1}^{q+1} \sum_{d=0}^{\infty} \min\left\{\Delta_i, 2\cdot\left(\tfrac{1}{2}\right)^d \cdot n\right\}$$

$$\le \sum_{i=1}^{q+1} \left[\lg\left(\frac{n}{\Delta_i}\right) + 1\right]\Delta_i + O(\Delta_i) \qquad = \mathcal{B} + O(n)$$

$$(\#\text{comparisons} = \Theta(C))$$

Multiple Selection Algorithms

One can indeed get very close to lower bound

- $\mathcal{B} + o(\mathcal{B}) + O(n)$ *comparisons* suffice

Kaligosi, Mehlhorn, Munro & Sanders: *Towards Optimal Multiple Selection*, ICALP **2005**

  - Using **Divide & Conquer over Elements** with pivot chosen as median of random $n^{3/4}$ elements works (analysis by induction)
  - A more complicated algorithm also achieves the same deterministically

Going online

Note that MultiSelect's recursion and partitioning does ***not*** *depend on sought ranks*

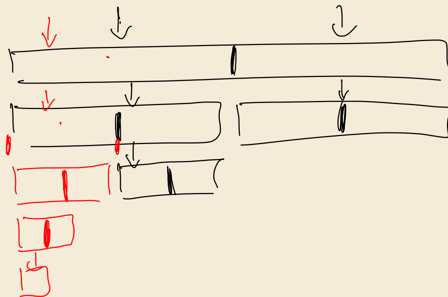
- We stop when a subproblem has no rank
- other than that, just "median your way through"

Going online

*Note that MultiSelect's recursion and partitioning does **not** depend on sought ranks*

- We stop when a subproblem has no rank
- other than that, just "median your way through"

⇝ Can easily **add** further ranks *later*!



Going online

*Note that MultiSelect's recursion and partitioning does **not** depend on sought ranks*

- We stop when a subproblem has no rank
- other than that, just "median your way through"

$\rightsquigarrow$ Can easily **add** further ranks *later*!

Quicksortus interruptus

- as MultiSelect, but remember pivots used in the past
- start between closest past pivots, repeatedly partition around median

Going online

*Note that MultiSelect's recursion and partitioning does **not** depend on sought ranks*

- We stop when a subproblem has no rank
- other than that, just "median your way through"

⇝ Can easily **add** further ranks *later*!

Quicksortus interruptus

- as MultiSelect, but remember pivots used in the past
- start between closest past pivots, repeatedly partition around median

⇝ Same partitioning steps as in offline MultiSelect, just "on a funny schedule"

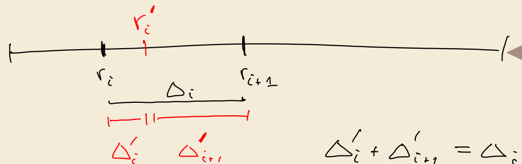
⇝ analysis of Theorem 2.21 remains valid
(with $\mathcal{B}$ for the final set of queried ranks)

Online Multiple Selection

Lemma 2.22

The cost of one new query r'_i in (r_i, r_{i+1}), splitting gap i of size Δ_i into $\Delta'_i = r'_i - r_i$ and $\Delta'_{i+1} = r_{i+1} - r'_i$ costs

$$\Delta'_i \lg\left(\frac{\Delta_i}{\Delta'_i}\right) + \Delta'_{i+1} \lg\left(\frac{\Delta_i}{\Delta'_{i+1}}\right) \quad \in \quad [\lg \Delta_i, \Delta_i]$$



cost of extension of MultiSelect

$$B = \sum_{i=1}^{q+1} \Delta_i \lg\left(\frac{n}{\Delta_i}\right)$$

$$= B' - B$$

$$= \Delta'_i \lg\left(\frac{n}{\Delta'_i}\right) + \Delta'_{i+1} \lg\left(\frac{n}{\Delta'_{i+1}}\right) - \Delta_i \lg\left(\frac{n}{\Delta_i}\right)$$

$$= \Delta'_i \lg\left(\frac{\Delta_i}{\Delta'_i}\right) + \Delta'_{i+1} \lg\left(\frac{\Delta_i}{\Delta'_{i+1}}\right)$$

2.10 Lazy Search Trees

Going dynamic

So we know how to add (and answer) more *queried ranks* in an online fashion

How about maintaining a dynamic set subject to insertions and deletions?

- Maintaining data in single array inconvenient

Going dynamic

So we know how to add (and answer) more *queried ranks* in an online fashion

How about maintaining a dynamic set subject to insertions and deletions?

- Maintaining data in single array inconvenient

$\rightsquigarrow$ Linked lists work fine!

  - key property: access is *scanning* based
  - partitioning can directly work with linked lists (even reusing nodes)
  - median-of-medians pivot selection can also directly work with scans

Going dynamic

So we know how to add (and answer) more *queried ranks* in an online fashion

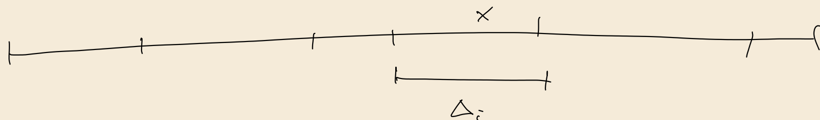
How about maintaining a dynamic set subject to insertions and deletions?

- Maintaining data in single array inconvenient

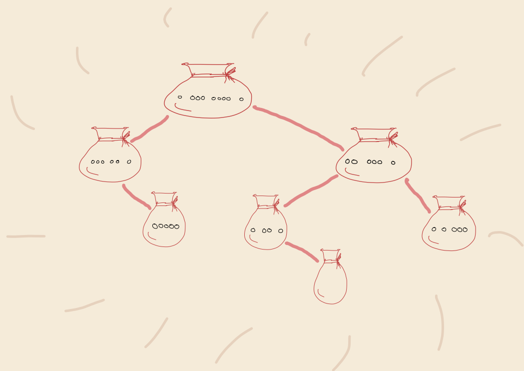
⇝ Linked lists work fine!

  - key property: access is *scanning* based
  - partitioning can directly work with linked lists (even reusing nodes)
  - median-of-medians pivot selection can also directly work with scans

- Insertion can use pivots to find correct gap, add element to list



Wishful-Thinking Solution



- BST of unsorted "bags"
- Insert simply finds right bag, adds element
- To answer query must look inside bag

⇝ partition one bag around query

Is this good? What running times should we aim for? Can we do better?

Notation

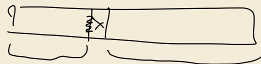
We mostly use the notation from multiple selection, with a few convenience changes

- m = number of gaps $q+1$
- Gaps $\Delta_1, \ldots, \Delta_m$ partition elements into bags (previously Δ_i denoted only the size)

  $$\sum_{i=1}^{m} |\Delta_i| = n$$

  $|\Delta_i|$

- queried elements are smallest or largest in gap



- $r_0 = 0, r_{q+1} = n \quad \rightsquigarrow \quad |\Delta_i| = r_i - r_{i-1}$

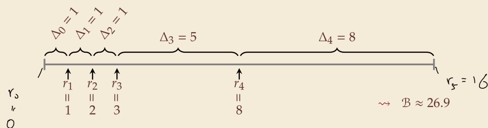
Working against $\mathcal{B}$

Idea: *Any snapshot still a multiple-select problem, subject to lower bound* $\Omega(\mathcal{B} + n)$

⇝ Trace how $\mathcal{B} + n$ changes, try to match that as amortized running time.

"Gap Entropy":

$$\mathcal{B} = \sum_{i=1}^{q+1} \Delta_i \log_2 \frac{n}{\Delta_i} \qquad \text{with } \Delta_i = r_i - r_{i-1} \qquad (1 \le i \le q+1,\ r_0 = 0 \text{ and } r_{q+1} = n)$$



Theorem 2.20 (Multiple Selection Lower Bound)

Multiple selection requires $\ge \mathcal{B} - O(n)$ comparisons and $\Omega(\mathcal{B} + n)$ time in the worst and average case over inputs of n elements and query ranks $r_1 < \cdots < r_q$. ◀

Working against $\mathcal{B}$

Idea: *Any snapshot still a multiple-select problem, subject to lower bound* $\Omega(\mathcal{B}+n)$

⇝ Trace how $\mathcal{B}+n$ changes, try to match that as amortized running time.

Lemma 2.23 (Query lower bound)

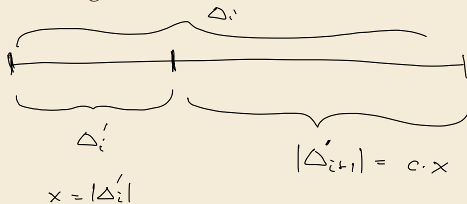
Suppose a query splits a gap Δ_i into two gaps of size x and $c \cdot x$ for $c \geq 1$.
Then the bound $\mathcal{B}+n$ increases by $\Omega(x \log c)$. $lg(x) = \max\{\log(x), 1\}$
Moreover, $\mathcal{B} = \Omega(m \log n)$ for $m \geq 2$, so a query has $\Omega(\log n)$ amortized costs. ◀

$\mathcal{B}' - \mathcal{B}$

$$= |\Delta_i'| \cdot lg\left(\frac{|\Delta_i|}{|\Delta_i'|}\right) + |\Delta_{i+1}'|\, lg\left(\frac{|\Delta_i|}{|\Delta_{i+1}'|}\right)$$

$$= x\, lg\left(\frac{(c+1)x}{x}\right) + cx\, lg\left(\frac{(c+1)x}{cx}\right)$$

$$= x\left((c+1)\, lg(c+1) - \underbrace{c\, lg\, c}_{\geq 0}\right) \geq x\, lg(c+1)$$



$$B = \sum_{i=1}^{m} |\Delta_i| \lg\left(\frac{n}{|\Delta_i|}\right) \overset{?}{=} \Omega(m \lg n) \qquad m \geq 2 \qquad (\Delta = |\Delta_i|)$$

- If $\lg n \leq |\Delta_i| \leq \frac{n}{2}$ $\qquad \Delta \lg\left(\underbrace{\frac{n}{\Delta}}_{\geq 2}\right) \geq \Delta \geq \lg n$
- at most one Δ_i has $\Delta > \frac{n}{2}$, ignore Δ_i
- If $1 \leq |\Delta_i| < \lg n$ $\qquad \Delta \lg\left(\underbrace{\frac{n}{\Delta}}_{> \frac{n}{\lg n}}\right) > \lg(n) - \lg \lg n$

Working against $\mathcal{B}$

Idea: *Any snapshot still a multiple-select problem, subject to lower bound* $\Omega(\mathcal{B} + n)$

$\rightsquigarrow$ Trace how $\mathcal{B} + n$ changes, try to match that as amortized running time.

Lemma 2.23 (Query lower bound)

Suppose a query splits a gap Δ_i into two gaps of size x and $c \cdot x$ for $c \geq 1$.
Then the bound $\mathcal{B} + n$ increases by $\Omega(x \log c)$.

Moreover, $\mathcal{B} = \Omega(m \log n)$ for $m \geq 2$, so a query has $\Omega(\log n)$ amortized costs.

Lemma 2.24 (Insert lower bound)

Suppose we insert an element into gap Δ_i.

Then the bound $\mathcal{B} + n$ increases by $\Omega\left(\log\left(\frac{n}{|\Delta_i|}\right)\right)$.

Proof: • First consider n unchanged $\quad \Delta := |\Delta_i|$

Increasing $|\Delta_i|$ turns $\Delta \lg\left(\frac{n}{\Delta}\right)$ into $(\Delta+1)\lg\left(\frac{n}{\Delta+1}\right)$

change $(\Delta+1)\lg\left(\frac{n}{\Delta+1}\right) - \Delta\lg\left(\frac{n}{\Delta}\right) \quad (*)$

$f(x) = x \lg\left(\frac{n}{x}\right)$

$$f(x+1) - f(x) \geq 1 \cdot \min_{y \in [x, x+1]} f'(y) \qquad f'(x) = \lg\left(\frac{n}{x}\right) - \frac{1}{\ln 2}$$

$$(*) \geq \lg\left(\frac{n}{\Delta+1}\right) - 1.443$$

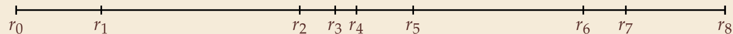
- Increasing n never makes B smaller n goes up by 1

$\Rightarrow$ total increase $\Omega\left(\lg\left(\frac{n}{\Delta}\right)\right)$

Step 1: Biased search tree over gaps

Step 1: Biased search tree over gaps

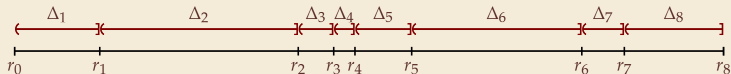
$r_1 < r_2 < \cdots < r_q$ queries ranks



Step 1: Biased search tree over gaps

$r_1 < r_2 < \cdots < r_q$ queries ranks

$|\Delta_i| = r_i - r_{i-1}$
$(r_0 = 0,\ r_{q+1} = n)$



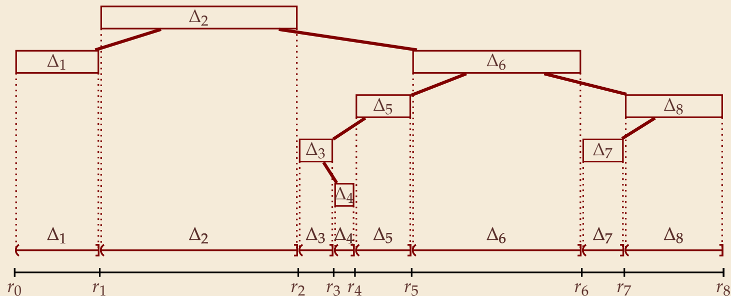
Step 1: Biased search tree over gaps

$r_1 < r_2 < \cdots < r_q$ queries ranks

$|\Delta_i| = r_i - r_{i-1}$

$(r_0 = 0,\ r_{q+1} = n)$

Step 1: Biased search tree over gaps

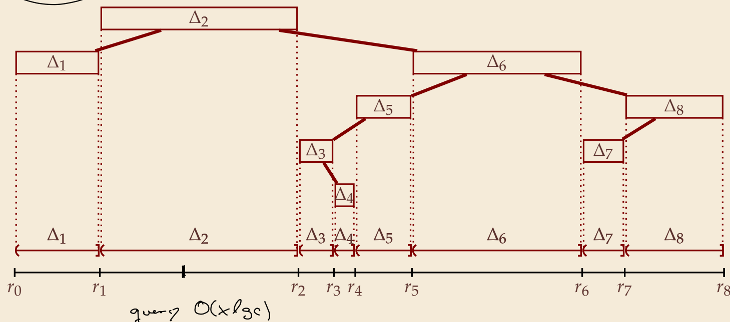


$r_1 < r_2 < \cdots < r_q$ queries ranks

$|\Delta_i| = r_i - r_{i-1}$

$(r_0 = 0,\ r_{q+1} = n)$

Step 1: Biased search tree over gaps



$r_1 < r_2 < \cdots < r_q$ queries ranks

$|\Delta_i| = r_i - r_{i-1}$

$(r_0 = 0,\ r_{q+1} = n)$

Gap Data Structure

- store in ***biased search tree*** (by size $|\Delta_i|$) $\rightsquigarrow$ access Δ_i in $O\left(\log \frac{W}{w_i}\right) = O\left(\log \frac{n}{|\Delta_i|}\right)$

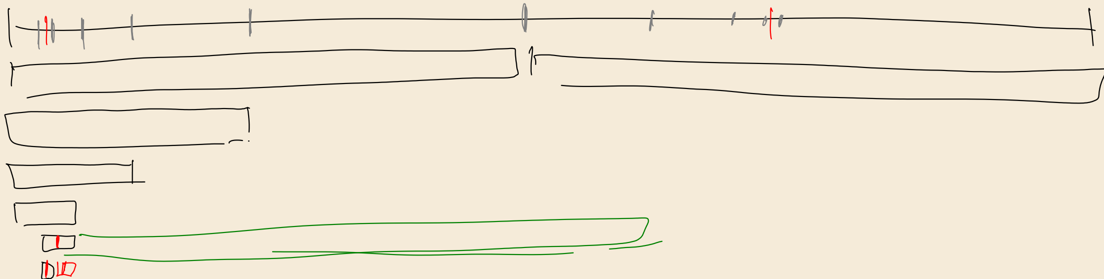
- ☑ Insert does find and `changeWeight`

- ☑ Query finds gap and replace it by two new ones, but $\Omega(\log n)$ increase in lower bound

The Query Problem

Recall that recursive median splitting was good enough for query in online multiple selection when set of elements static.

However, with (adversarial) insertions, medians don't stay medians . . .

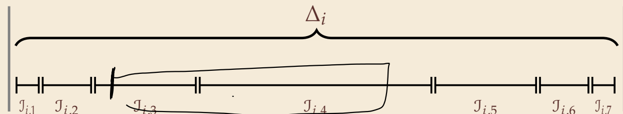


The Query Problem

Recall that recursive median splitting was good enough for query in online multiple selection when set of elements static.

However, with (adversarial) insertions, medians don't stay medians . . .

Must maintain further structure inside gaps: ***intervals***



Smaller intervals $\rightsquigarrow$ faster queries
More intervals $\rightsquigarrow$ slower insertions

Non-Solutions

- Collect insertions in buffers?
  - ϟ $O(x \log c)$ amortized cost for queries?
  - ϟ unclear how to balance gap data structure without flushing buffers

Non-Solutions

- Collect insertions in buffers?
  - ϟ $O(x \log c)$ amortized cost for queries?
  - ϟ unclear how to balance gap data structure without flushing buffers

- Interval sizes roughly doubling as we move away from boundary.
  - ϟ unclear if maintainable upon query in allowed time

Our Solution: Lazy Search Trees

Sandlund, Wild: *Lazy Search Trees*, FOCS **2020**

Theorem 2.25 (Lazy search tree runtimes)

For n the current number of elements, $\{\Delta_i\}$ as above and q the total number of queries, we do

- `Insert(`x`)` in $O(\min(\log(n/|\Delta_i|) + \log\log|\Delta_i|,\ \log q))$ worst-case time, where $x \in \Delta_i$.
- `RankBasedQuery(`r`)` in $O(x\log c + \log n)$ amortized time, resulting gaps have size cx and $x, c \ge 1$.
- **Sequence of insert and query w/o duplicate queries** in total $O(\mathcal{B} + \min(n\log\log n, n\log q))$.

Our Solution: Lazy Search Trees

Theorem 2.25 (Lazy search tree runtimes)

For n the current number of elements, $\{\Delta_i\}$ as above and q the total number of queries, we do

- `Insert(`x`)` in $O(\min(\log(n/|\Delta_i|) + \log\log|\Delta_i|, \log q))$ worst-case time, where $x \in \Delta_i$.
- `RankBasedQuery(`r`)` in $O(x \log c + \log n)$ amortized time, resulting gaps have size cx and $x, c \ge 1$.
- **Sequence of insert and query w/o duplicate queries** in total $O(\mathcal{B} + \min(n \log\log n, n \log q))$.
- `Construction(`S`)` in $O(n)$ worst-case time, where $|S| = n$.
- `Delete(ptr)` in $O(\log n)$ worst-case time.
- `ChangeKey(ptr, `x'`)` in $O(\min(\log q, \log\log|\Delta_i|))$ worst-case time, where the element at `ptr`, is $x \in \Delta_i$ and x' lies to its closest query rank in Δ_i; otherwise, takes $O(\log n)$ worst-case time.
- `Split(`r`)` in time according to `RankBasedQuery(`r`)`.
- `Merge(`T_1`, `T_2`)` (where $T_1 \le T_2$) in $O(\log n)$ worst-case time. ◀

In short: all operations amortized to lower bound, except $+O(\log\log n)$ per insert

Our Solution: Lazy Search Trees

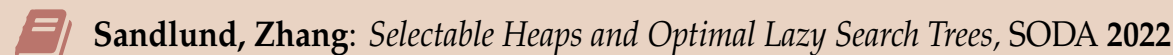
Theorem 2.25 (Lazy search tree runtimes)

For n the current number of elements, $\{\Delta_i\}$ as above and q the total number of queries, we do

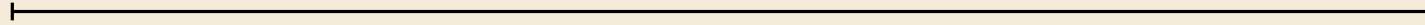
- `Insert(`x`)` in $O(\min(\log(n/|\Delta_i|) + \log\log|\Delta_i|, \log q))$ worst-case time, where $x \in \Delta_i$.
- `RankBasedQuery(`r`)` in $O(x \log c + \log n)$ amortized time, resulting gaps have size cx and $x, c \geq 1$.
- **Sequence of insert and query w/o duplicate queries** in total $O(\mathcal{B} + \min(n \log\log n, n \log q))$.
- `Construction(`S`)` in $O(n)$ worst-case time, where $|S| = n$.
- `Delete(ptr)` in $O(\log n)$ worst-case time.
- `ChangeKey(ptr, `x'`)` in $O(\min(\log q, \log\log|\Delta_i|))$ worst-case time, where the element at `ptr`, is $x \in \Delta_i$ and x' lies to its closest query rank in Δ_i; otherwise, takes $O(\log n)$ worst-case time.
- `Split(`r`)` in time according to `RankBasedQuery(`r`)`.
- `Merge(`T_1`, `T_2`)` (where $T_1 \leq T_2$) in $O(\log n)$ worst-case time. ◀

In short: all operations amortized to lower bound, except $+O(\log\log n)$ per insert

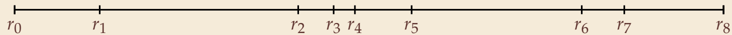
$O(\log\log n)$ later removed via selectable heaps



Lazy Search Trees – Overview



Lazy Search Trees – Overview

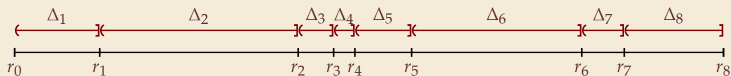


$r_1 < r_2 < \cdots < r_q$ queries ranks

Lazy Search Trees – Overview

Three levels:

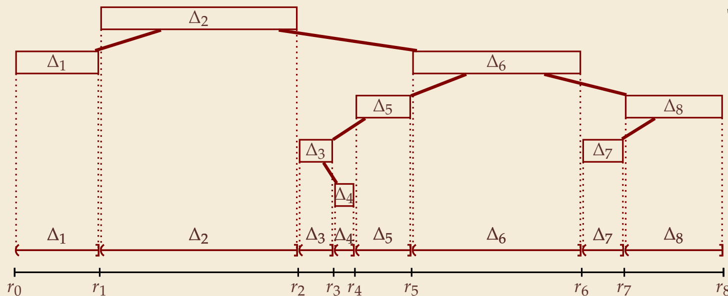
1. *Gaps* Δ_i (= bags)



$r_1 < r_2 < \cdots < r_q$ queries ranks

$|\Delta_i| = r_i - r_{i-1}$
$(r_0 = 0,\ r_{q+1} = N)$

Lazy Search Trees – Overview



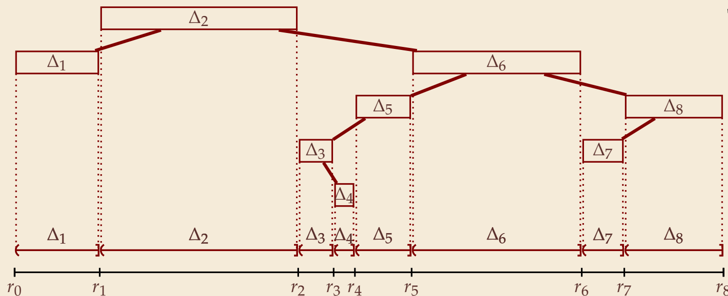
Three levels:

1. *Gaps* Δ_i (= bags)
   - maintained in a ***biased search tree***

$r_1 < r_2 < \cdots < r_q$ queries ranks

$|\Delta_i| = r_i - r_{i-1}$
$(r_0 = 0,\ r_{q+1} = N)$

Lazy Search Trees – Overview



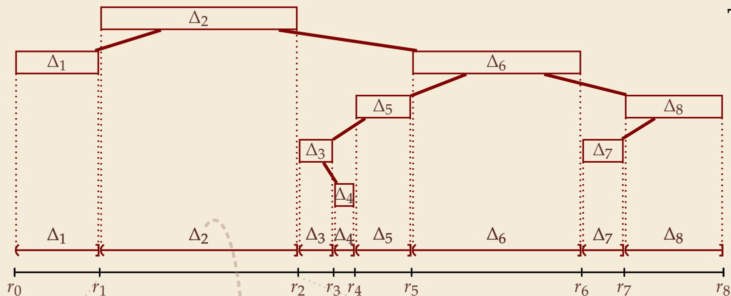
Three levels:

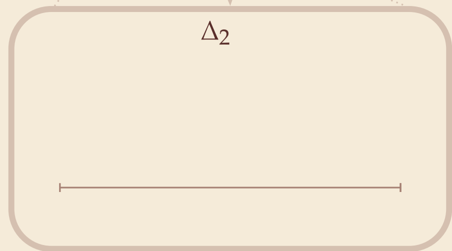
1. *Gaps* Δ_i (= bags)
   - maintained in a ***biased search tree***
   - access Δ_i in $O(\log \frac{W}{w_i}) = O(\log \frac{n}{|\Delta_i|})$

$r_1 < r_2 < \cdots < r_q$ queries ranks

$|\Delta_i| = r_i - r_{i-1}$
$(r_0 = 0, \ r_{q+1} = N)$

Lazy Search Trees – Overview





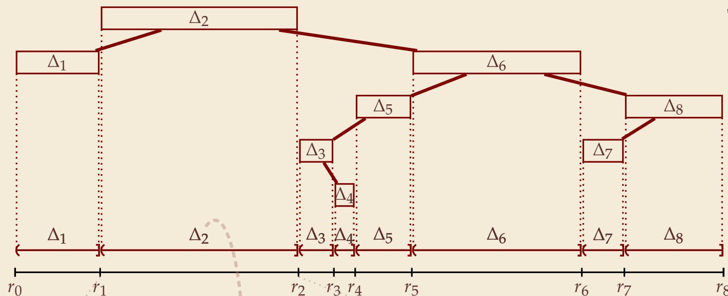
$r_1 < r_2 < \cdots < r_q$ queries ranks

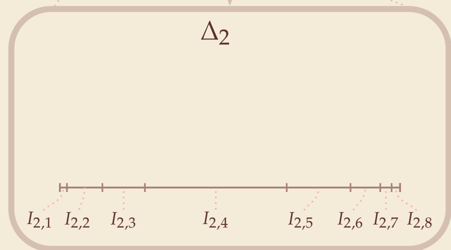
$|\Delta_i| = r_i - r_{i-1}$
$(r_0 = 0,\ r_{q+1} = N)$

Three levels:

1. *Gaps* Δ_i (= bags)
   - maintained in a ***biased search tree***
   - ☑ access Δ_i in $O(\log \frac{W}{w_i}) = O(\log \frac{n}{|\Delta_i|})$
2. Within Δ_i, maintain set of *intervals* for fast queries

Lazy Search Trees – Overview





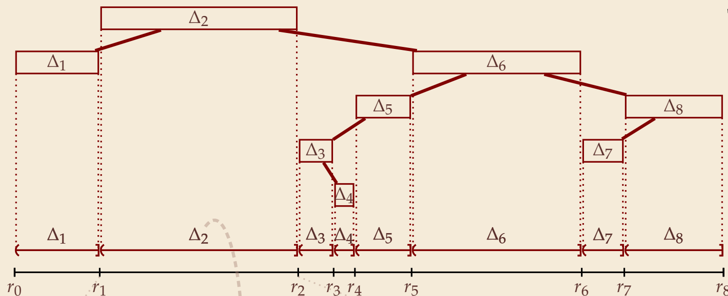
$r_1 < r_2 < \cdots < r_q$ queries ranks

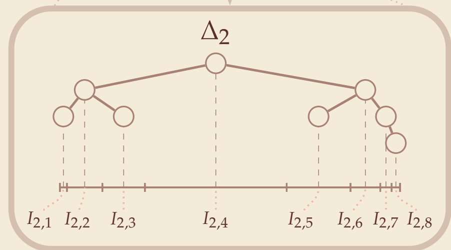
$|\Delta_i| = r_i - r_{i-1}$
$(r_0 = 0,\ r_{q+1} = N)$

Three levels:

1. *Gaps* Δ_i (= bags)
   - maintained in a ***biased search tree***
   - ☑ access Δ_i in $O(\log \frac{W}{w_i}) = O(\log \frac{n}{|\Delta_i|})$
2. Within Δ_i, maintain set of *intervals* for fast queries
   - $O(\log |\Delta_i|)$ intervals
3. Each interval is stored as *unsorted list*.

Lazy Search Trees – Overview





$r_1 < r_2 < \cdots < r_q$ queries ranks

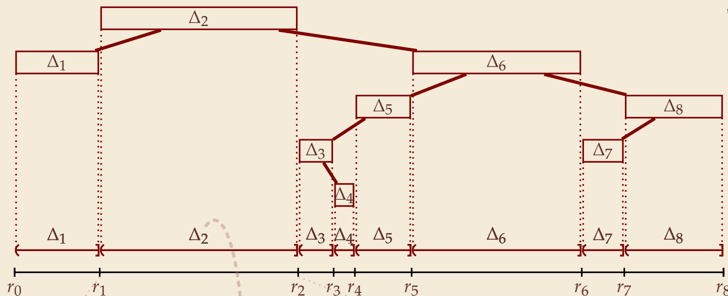
$|\Delta_i| = r_i - r_{i-1}$

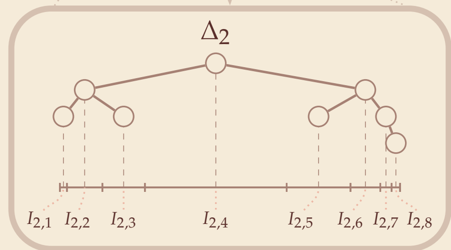
$(r_0 = 0,\ r_{q+1} = N)$

Three levels:

1. *Gaps* Δ_i (= bags)
   - maintained in a ***biased search tree***
   - access Δ_i in $O(\log \frac{W}{w_i}) = O(\log \frac{n}{|\Delta_i|})$
2. Within Δ_i, maintain set of *intervals* for fast queries
   - $O(\log |\Delta_i|)$ intervals
   - stored in BBST
3. Each interval is stored as *unsorted list*.

Lazy Search Trees – Overview





$r_1 < r_2 < \cdots < r_q$ queries ranks

$|\Delta_i| = r_i - r_{i-1}$

$(r_0 = 0,\ r_{q+1} = N)$

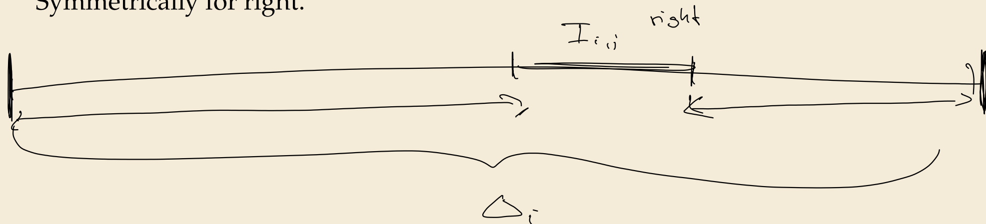
Three levels:

1. *Gaps* Δ_i (= bags)
   - maintained in a ***biased search tree***
   - access Δ_i in $O(\log \frac{W}{w_i}) = O(\log \frac{n}{|\Delta_i|})$
2. Within Δ_i, maintain set of *intervals* for fast queries
   - $O(\log |\Delta_i|)$ intervals
   - stored in BBST
   - $O(\log \log n)$ time to access interval
3. Each interval is stored as *unsorted list*.

Intervals

Interval $\mathcal{I}_{i,j}$: Unsorted list with values between two pivots

- **Left/Right Intervals:**
  interval $\mathcal{I}_{i,j}$ in gap Δ_i is on the *left side* if the closest query rank is to the left
  Symmetrically for right.

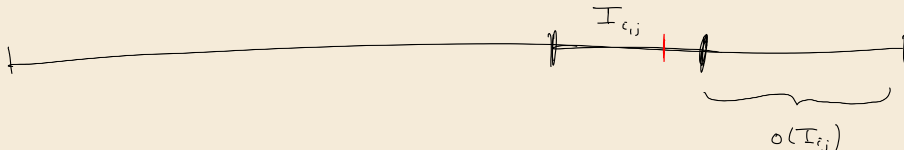


Intervals

Interval $\mathcal{I}_{i,j}$: Unsorted list with values between two pivots

- **Left/Right Intervals:**
  interval $\mathcal{I}_{i,j}$ in gap Δ_i is on the *left side* if the closest query rank is to the left
  Symmetrically for right.

- **Outside of an interval:**
  $\mathbf{o}(\mathcal{I}_{i,j})$ = #elements ***outside*** of $\mathcal{I}_{i,j} = \min\{|\mathcal{I}_{i,1}| + \cdots + |\mathcal{I}_{i,\mathbf{j-1}}|, \ |\mathcal{I}_{i,\mathbf{j+1}}| + \cdots + |\mathcal{I}_{i,\ell_i}|\}$



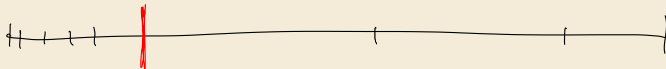
Intervals

Interval $\mathcal{I}_{i,j}$: Unsorted list with values between two pivots

- **Left/Right Intervals:**
  interval $\mathcal{I}_{i,j}$ in gap Δ_i is on the *left side* if the closest query rank is to the left
  Symmetrically for right.

- **Outside of an interval:**
  $\mathbf{o}(\mathcal{I}_{i,j})$ = #elements ***outside*** of $\mathcal{I}_{i,j} = \min\{|\mathcal{I}_{i,1}| + \cdots + |\mathcal{I}_{i,\boldsymbol{j-1}}|,\ |\mathcal{I}_{i,\boldsymbol{j+1}}| + \cdots + |\mathcal{I}_{i,\ell_i}|\}$

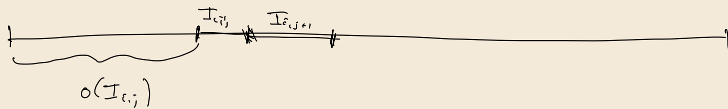
- **Invariant (#Int):** In Δ_i, the number ℓ_i of intervals $I_{i,j}$ is at most $\ell_i \le 4\log_2|\Delta_i|$
  - when at risk (upon query), restored via interval-merging rule (below)



Intervals

Interval $\mathcal{I}_{i,j}$: Unsorted list with values between two pivots

- **Left/Right Intervals:**
  interval $\mathcal{I}_{i,j}$ in gap Δ_i is on the *left side* if the closest query rank is to the left
  Symmetrically for right.
- **Outside of an interval:**
  $\mathbf{o}(\mathcal{I}_{i,j})$ = #elements *outside* of $\mathcal{I}_{i,j} = \min\{|\mathcal{I}_{i,1}| + \cdots + |\mathcal{I}_{i,j-1}|,\ |\mathcal{I}_{i,j+1}| + \cdots + |\mathcal{I}_{i,\ell_i}|\}$
- **Invariant (#Int):** In Δ_i, the number ℓ_i of intervals $I_{i,j}$ is at most $\ell_i \le 4\log_2|\Delta_i|$
  - when at risk (upon query), restored via interval-merging rule (below)
- sizes of intervals not restricted $\rightsquigarrow$ insert does no maintenance
- **Merge Rule (M):** If $|\mathcal{I}_{i,j}| + |\mathcal{I}_{i,j+1}| \le \mathbf{o}(\mathcal{I}_{i,j})$, merge $\mathcal{I}_{i,j}$ and $\mathcal{I}_{i,j+1}$.
  - Only triggered upon query; not an invariant
  - This rule only applies between intervals of the same type;
    a left interval is never merged with a right interval.


$I_{i-1,j}$
$I_{i,j+1}$
$o(I_{i,j})$

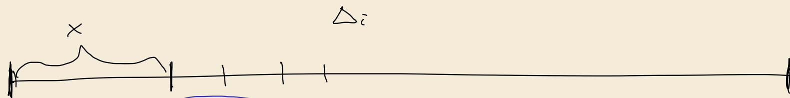
(M) implies (#Int)

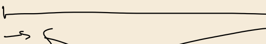
Merge Rule (M): If $|\mathcal{I}_{i,j}| + |\mathcal{I}_{i,j+1}| \le \mathbf{o}(\mathcal{I}_{i,j})$, merge $\mathcal{I}_{i,j}$ and $\mathcal{I}_{i,j+1}$.

Lemma 2.26

Right after applying rule (M), Invariant (#Int) holds.

no more merges possible





#intervals right of x elements $O(\lg(\frac{\Delta}{x}))$

x, x, 2x

1, 1, 1, 2, 2, 4, 4, 8, 8, 16, 16

first k intervals contain $\ge 2 \cdot \sqrt{2}^k = \sqrt{2}^{k+2}$

$\Rightarrow k \le 2 \lg \Delta - 2$

two sides $k \le 2 \cdot 2 \lg(\frac{\Delta}{2}) \le 4 \lg \Delta$

2.11 Lazy Search Tree Operations

Insert

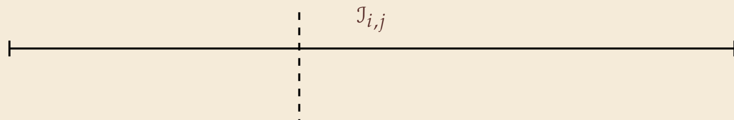
(1) Find correct gap Δ_i (in biased search tree)
(2) Find correct interval $\mathcal{I}_{i,j}$ (in balanced search tree)
(3) Insert new element and increment weight of Δ_i

— no maintenance —

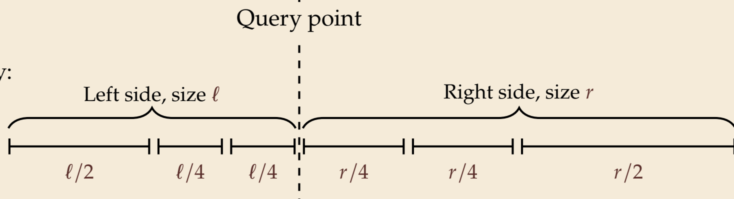
Rank-Based Query

(1) Find correct gap Δ_i (in biased search tree)
(2) Invoke Rule (M) on Δ_i
(3) Find correct interval $\mathcal{I}_{i,j}$ (in balanced search tree)
(4) Split interval $\mathcal{I}_{i,j}$ 5 times (see below)
(5) Split gap and distribute intervals
(6) Invoke Rule (M) on Δ'_i and Δ'_{i+1}

Interval before query:



Intervals after query:



Rank-Based Queries

Rank-Based Queries are all of the following

- Find the rank of element x (rank is the result)
- Select the element of rank r (rank r)
- Get element for key x (rank of x)
- Successor / Predecessor for key x (rank of returned element)
- min / max (rank 1 resp. n)

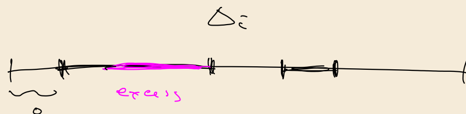
Potential

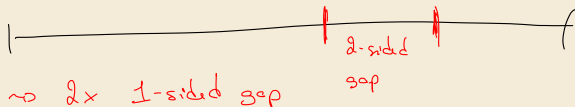
The amortized analysis uses a potential function

$$\Phi = N_{01} + \sum_{i,j} \Phi_{i,j}$$

$$\Phi_{i,j} = 1 + \max\{|\mathcal{I}_{i,j}| - \mathbf{o}(\mathcal{I}_{i,j}), 0\}$$

$$N_{01} = \text{\#elements in 0-sided and 1-sided gaps}$$





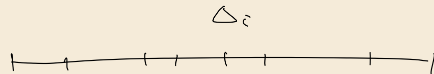
Potential

The amortized analysis uses a potential function

$$\Phi = N_{01} + \sum_{i,j} \Phi_{i,j}$$

$$\Phi_{i,j} = 1 + \max\left\{|\mathcal{I}_{i,j}| - \mathbf{o}(\mathcal{I}_{i,j}),\ 0\right\}$$

$$N_{01} = \text{\#elements in 0-sided and 1-sided gaps}$$



Lemma 2.27

Applying Rule (M) to Δ_i has amortized cost $O(\ell_i)$.

$O(\log n)$

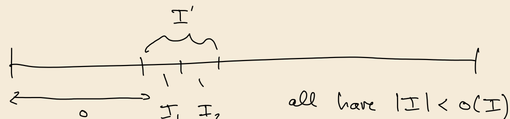
- actual cost
- $\Delta\Phi$ single merge $\Delta\Phi = -1$ (lose one interval)

compute $o(I)$ $O(1)$ per interval

apply merges needed $O(1)$

$O(\ell_i)$ time ℓ_i = # int in Δ_i $= O(\lg|\Delta_i|)$

rebuild BBST after merges $O(\ell_i)$



Amortized Costs

full details spelled out in

Lemma 2.28

The amortized cost of `Insert(x)` into Δ_i is $O\left(\log \frac{n}{|\Delta_i|} + \log\log n\right)$ ◀

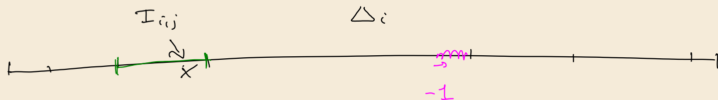
Proof

actual cost:

(1) (2) (3) $O(1)$

(1) Find correct gap Δ_i (in biased search tree)
(2) Find correct interval $\mathcal{I}_{i,j}$ (in balanced search tree)
(3) Insert new element and increment weight of Δ_i

$\Delta\Phi$



$|I_{i,j}| \mathrel{+}= 1$

+1

might also have N_{o1} +1

$\Delta\Phi \le O(1)$

$$\Phi_{i,j} = 1 + \max\{|\mathcal{I}_{i,j}| - \mathbf{o}(\mathcal{I}_{i,j}),\ 0\}$$

Amortized Costs

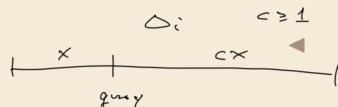
full details spelled out in

Lemma 2.28

The amortized cost of `Insert(x)` into Δ_i is $O\left(\log \frac{n}{|\Delta_i|} + \log\log n\right)$ ◀

Lemma 2.29

The amortized cost of `RankBasedQuery(r)` is is $O(x \log c + \log n)$ ◀



(2), (6) amortized cost $O(\log n)$

actual cost (1) $O(\log \frac{n}{|\Delta_i|}) = O(\log n)$

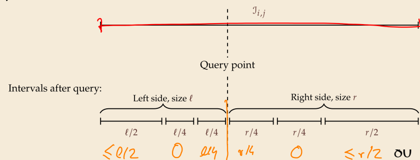
(3) $O(\log \log |\Delta_i|) = O(\log n)$

(5) rebuilding BSTs $O(\log n)$

insert new Δ'_{i+1} into biased tree $O(\log n)$

(4) $O(|I_{i,j}|)$ $I := |I_{i,j}|$

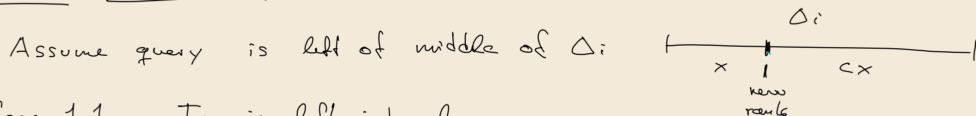
(1) Find correct gap Δ_i (in biased search tree)
(2) Invoke Rule (M) on Δ_i
(3) Find correct interval $\mathcal{I}_{i,j}$ (in balanced search tree)
(4) Split interval $\mathcal{I}_{i,j}$ 5 times (see below)
(5) Split gap and distribute intervals
(6) Invoke Rule (M) on Δ'_i and Δ'_{i+1}



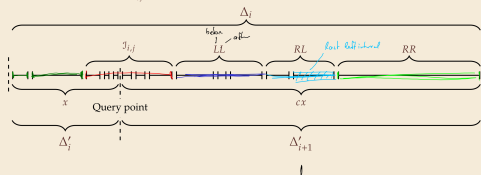
$\Delta\Phi$ (Want $\Delta\Phi \le -I + O(x \log c + \log n)$) $O(x \log c) = \Omega(x)$

Case 1: Δ_i 2-sided $\Rightarrow$ Δ'_i and Δ'_{i+1} also 2-sided

Assume query is left of middle of Δ_i



Case 1.1 $I_{i,j}$ is left interval



N_{01} doesn't change

$\Delta\Phi_I$ change in potential from splitting

$= -\Phi_{i,j} + \frac{3}{4} I + 5$

$\Phi_{i,j} = \max\{ I - o(I_{i,j}), 0\}$

(a) $I \le x$

$\Rightarrow \Delta\Phi \le \frac{3}{4} x + 5 = O(x)$

actual cost also $O(x)$ ✓

} amortized cost $O(x \log c)$

(b) $I > x$

$$\Rightarrow \Delta\Phi \le 5 + \frac{3}{4} I - \max\{\underset{>x}{I} - \underset{<x}{o(I)}, 0\}$$

$$\le 5 + \frac{3}{4} I - I + x$$

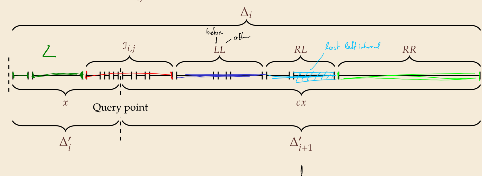
$$= 5 - \frac{1}{4} I + x$$

actual splitting cost $O(I)$ $\Rightarrow$ amortized cost $O(x)$

$\Delta\Phi_L$ intervals unchanged

total size $\le x$

$\Phi_L = O(x)$



$\Delta\Phi_{RR} = 0$

$\Delta\Phi_{LL}$ all lose x in outside

$|\Delta'_{i+1}| = c \cdot x$

Rule (M) holds here

\# intervals in LL by Lemma 2.26

$$\le 2\lg\left(\frac{cx}{x}\right) = O(\lg c)$$

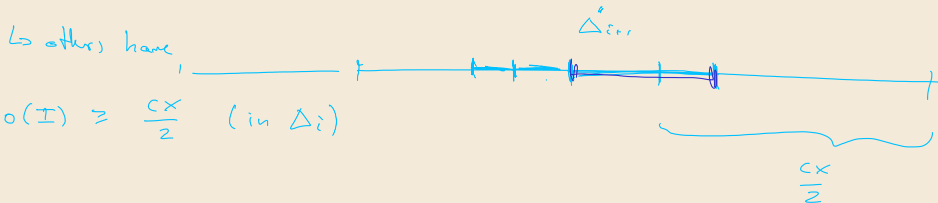
$$\Rightarrow \Delta\Phi_{LL} \leq O(\ell_{gc}) \cdot O(x) = O(x \ell_{gc})$$

$\Delta\Phi_{RL}$ intervals were right before, now shift to left by removing x elements

ignore rightmost left interval in Δ'_{i+1} (single middle interval)

↳ others have

$o(I) \geq \frac{cx}{2}$ (in Δ_i)



Rule (M) $\leq 2 \cdot \lg\left(\frac{cx}{cx/2}\right) = O(1)$ RL intervals

lose x on left $\Rightarrow$ all gain at most x potential

$$\Delta\Phi_{RL} = O(1) \cdot O(x) = O(x \ell_{gc})$$

Case 1.2 $I_{i,j}$ right

$\Rightarrow$ total # elements right of $I_{i,j} = O(x)$

$$\Delta\Phi_R = O(x)$$

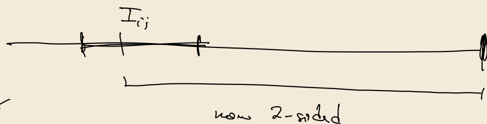
Case 2 Δ_i 0-sided

single interval ⇝ like $\Delta\Phi_I$ above

Case 3 Δ_i 1-sided

3.1 Query on left similar to Case 1

3.2 Query on right



$o(I_{ij})$ can be $c \cdot x$

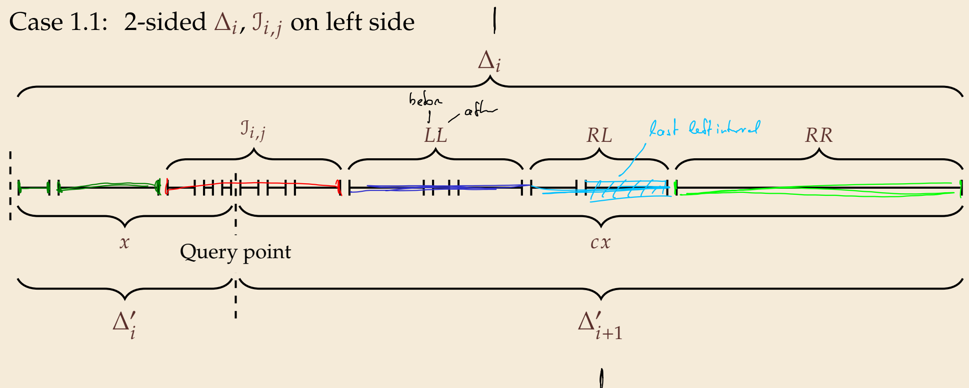
$\Rightarrow N_{01}$ drops by cx

rest similar

□

Amortized Costs – Query Analysis

Case 1.1: 2-sided Δ_i, $\mathcal{J}_{i,j}$ on left side



Lazy Search Trees – Discussion

- 👍 Lazy search trees (nearly) optimally solve insertions and queries compared to multiple-selection lower bound $\Omega(\mathcal{B} + n)$
- 👍 Never perform worse than standard BST: $O(\log n)$ amortized per operation

Lazy Search Trees – Discussion

- 👍 Lazy search trees (nearly) optimally solve insertions and queries compared to multiple-selection lower bound $\Omega(\mathcal{B} + n)$
- 👍 Never perform worse than standard BST: $O(\log n)$ amortized per operation
- 👍 When using splay tree as gap data structure and interval data structures inherit the **same access lemma guarantees as splay trees**

Lazy Search Trees – Discussion

- Lazy search trees (nearly) optimally solve insertions and queries compared to multiple-selection lower bound $\Omega(\mathcal{B} + n)$
- Never perform worse than standard BST: $O(\log n)$ amortized per operation
- When using splay tree as gap data structure and interval data structures inherit the **same access lemma guarantees as splay trees**
- When used like a **priority queue** (only query for min)
  Lazy Search Trees behave like an efficient priority queue!
  - `Insert`, `DecreaseKey` in $O(\log \log n)$ time
  - `DeleteMin` in $O(\log n)$ time

Lazy Search Trees – Discussion

- 👍 Lazy search trees (nearly) optimally solve insertions and queries compared to multiple-selection lower bound $\Omega(\mathcal{B} + n)$
- 👍 Never perform worse than standard BST: $O(\log n)$ amortized per operation
- 👍 When using splay tree as gap data structure and interval data structures inherit the **same access lemma guarantees as splay trees**
- 👍 When used like a **priority queue** (only query for min)
  Lazy Search Trees behave like an efficient priority queue!
  - `Insert`, `DecreaseKey` in $O(\log \log n)$ time
  - `DeleteMin` in $O(\log n)$ time

⇝ *Lazy Search Trees smoothly interpolate between priority queue and BST but do so automatically, according to actual usage*

Lazy Search Trees – Discussion

- 👍 Lazy search trees (nearly) optimally solve insertions and queries compared to multiple-selection lower bound $\Omega(\mathcal{B} + n)$
- 👍 Never perform worse than standard BST: $O(\log n)$ amortized per operation
- 👍 When using splay tree as gap data structure and interval data structures inherit the **same access lemma guarantees as splay trees**
- 👍 When used like a **priority queue** (only query for min)
  Lazy Search Trees behave like an efficient priority queue!
  - `Insert`, `DecreaseKey` in $O(\log \log n)$ time
  - `DeleteMin` in $O(\log n)$ time

⇝ *Lazy Search Trees smoothly interpolate between priority queue and BST but do so automatically, according to actual usage*

- 👎 Optimality corresponds to worst case of inserting all elements first, then doing all queries