

# ADVANCED

overall tree  
= binary tree of mini trees

mini trees

micro trees

actual nodes

$\lg n$  nodes

$\frac{1}{7} \lg n$  nodes



# DATA STRUCTURES

# 2

## Adaptive Trees

*Advanced Data Structures · Summer 2026*

Prof. Dr. Sebastian Wild

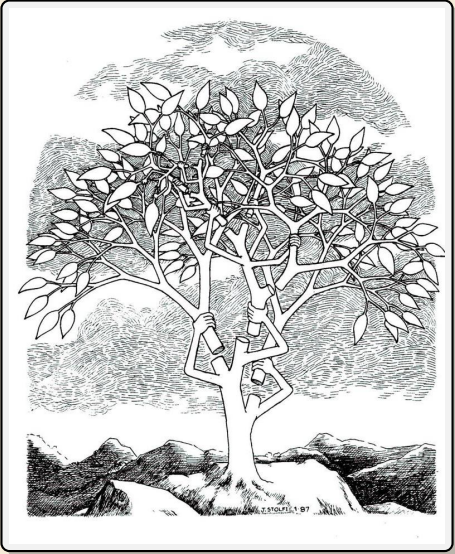
# Outline

## 2 Adaptive Trees

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

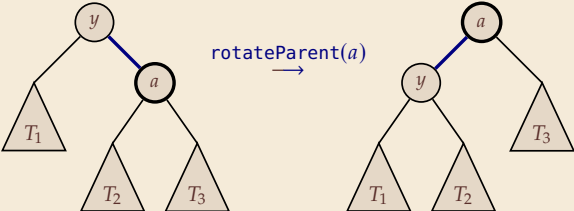
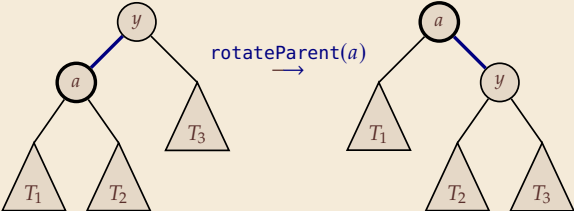
# Self-Adjusting Data Structures

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-Organizing Binary Search Trees*, JACM 1978

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

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

*Maybe, Move-to-root is similarly successful!*

## **2.2 Move-to-Root Analysis**

# Iid Accesses

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


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

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

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

## Lemma 2.1 (Bayer 1975)

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

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

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

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

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

# Move-to-Root on iid Accesses

## Lemma 2.2 (MTR ancestors)

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

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

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

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

## Corollary 2.3

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

# Move-to-Root on iid Accesses – Analysis

## Theorem 2.4 (MTR cost)

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

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

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

**Proof:**

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

This proves the first part.

# Move-to-Root on iid Accesses – Analysis

Proof (cont.):

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

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

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

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

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

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

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

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

## Unbalanced BSTs under i.i.d. model

Interestingly, the same cost result from unbalanced BSTs!

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

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

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

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

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

# Move-to-Root's Blind Spot

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

## Theorem 2.5 (Move-to-Root Bad Case)

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

**Proof:**

see exercises

## 2.3 Splay Trees

# Splay Trees

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

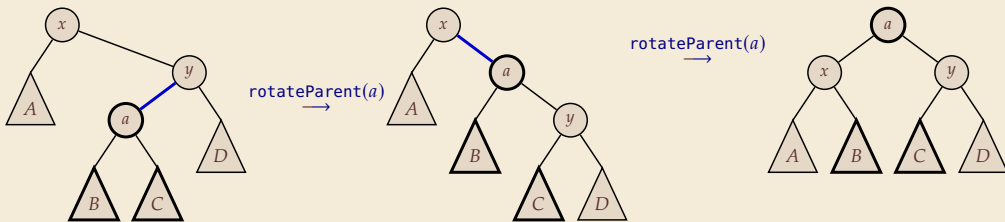
- 👍 reasonably simple and fast
- 👍 simple to implement
- 👍 remarkable theoretical properties (*access theorems*)  $\rightsquigarrow$  later
- 👎 many writes and pointer chases

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

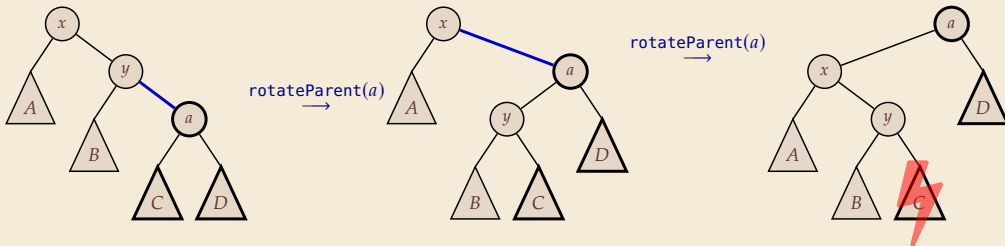
# Move-to-root reloaded

What's wrong in Move-to-front?

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



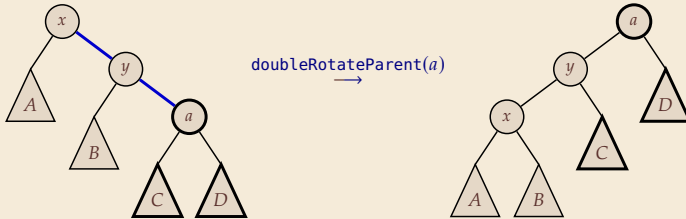
Using simple `rotateParent` calls, the subtree we came from (`B` or `C`) is lifted up!



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

# The Fix

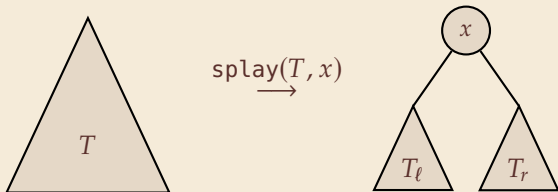
In the case that both parents are right children, do a *double rotation*



Now both  $C$  and  $D$  are lifted up!

# Splay

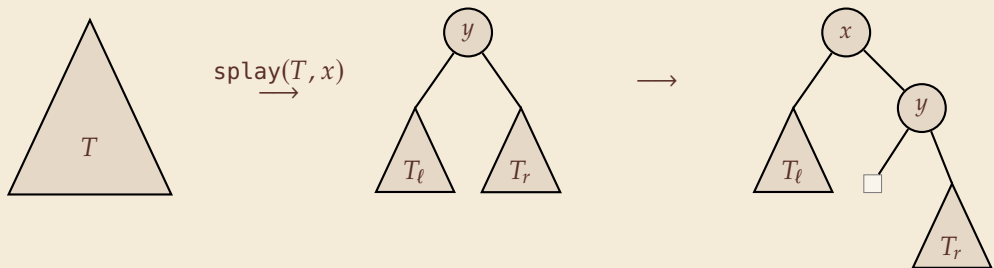
Aggressive restructuring allows to base all operations on single primitive: splay



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

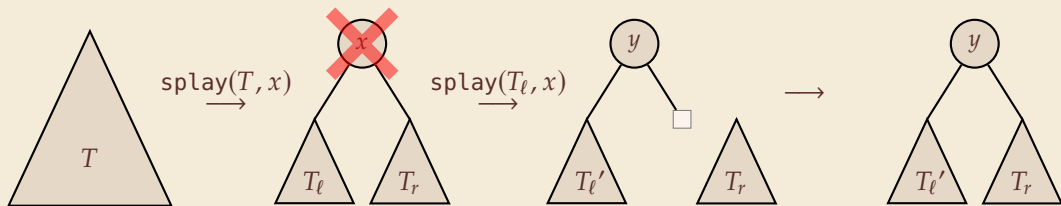
## Splay – Insert

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



# Splay – Delete

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



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

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

## 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  $k = \text{\#rotations in splay}$  calls.

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

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

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

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

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

↪ Released potential can pay for actual cost  $k$

↑  
change in "rank" of  $x$

# Splay Tree Potential

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

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

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

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

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

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

# The Access Lemma

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

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

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

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

## Lemma 2.6 (Access Lemma)

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

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



## Access Lemma – Proof

# Splay Cost

## Corollary 2.7

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

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

**Proof:**

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

↪ The rank differences telescope.

↪ Result follow from Lemma 2.6.

## Lemma 2.8 (Potential drop)

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

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

**Proof:**

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

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

# Splay Tree – Results

## Theorem 2.9 (Balance Theorem)

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

## Splay Tree – Results [2]

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

### Theorem 2.10 (Static Optimality)

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

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

## 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  $x \leq z < y$  (or  $x \leq z < x$ )

## Splay Tree – Results [4]

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

### Theorem 2.12 (Working Set Theorem)

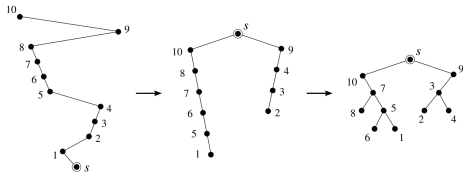
The total cost of access sequence  $X$  is  $O\left(m + n \log n + \sum_{j=1}^m \log(1 + t(j))\right)$



# What makes Splay tick?

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

## A global view of Splay



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

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

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

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

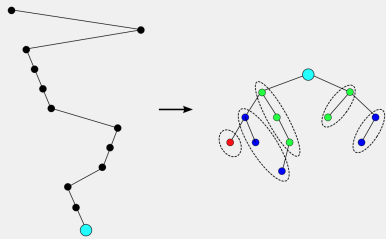
## Intuition:

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

# Anything magic about Splay?

Same paper shows: whole **class** of rearrangements satisfy access lemma

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



- length of search path:  $|P|$  12
- number of zigzags:  $z$  4
- number of leaves:  $\ell$  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:  $\ell + z = \Omega(|P|)$

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

## 2.5 Biased Search Trees

# Updates in Splay Trees

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

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

## Dynamic Splay Tree Potential

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

↪ universe  $U$  of items is fixed

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

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

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

# Analysis of Updates

## Theorem 2.13 (Splay Tree Update Lemma)

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

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

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

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

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

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

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

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

# Analysis of Updates – Proof

# Biased Search Trees

*Biased search trees* are a refinement of sorted dictionaries where

- ▶ elements have *known* weights  $w(x)$  (that the data structure can use)
- ▶ supports **biased** search costs (as in Theorem 2.14 when using these weights)
- ▶ **there is an explicit  $\text{changeWeight}_T(x, \delta)$  operation that sets  $w(x) := w(x) + \delta$**
- ▶ prior to Splay Trees solved via *globally-biased 2, b trees*



Bent, Sleator, Tarjan: *Biased search trees*, SICOMP 1985

- ↪ achieves all operations in **worst case** biased time
  - ▶ rather intricate invariants and higher overhead
- ▶ biased access times also achievable via randomization: *treaps, zip-zip trees*

# Weight Changes

## Lemma 2.14 (Weight Changes)

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

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

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

## 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
- ▶ default mode of thinking for **data structures!**
  - ▶ usually must answer one query before we receive the next

## Opposite: Offline Problems

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

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

## The Elevator Problem\*

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

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

*What should you do (to minimize time)?*

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

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

# Elewaiting as Sequence of Requests

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

## ↪ Elevator Problem

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

# Competitive Analysis

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

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

## Definition 2.15 (Competitive ratio)

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

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

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

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

# Competitive Elewaiting

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

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

$\rightsquigarrow$  Optimal choice for  $t$

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

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

## Further Famous Online-Algorithms

- ▶ Move-to-Front is 4-competitive for the list-update problem assuming only local swaps are allowed
- ▶ 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*

## Compare to what

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

## The BST Model of Computation

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

↪ *OPT finds shortest valid sequence of operations serving accesses*

# The Splay Tree Conjecture

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

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

(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.

# Simplified BST Model

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

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

## Facts on OPT

- ▶ can be computed in exponential time (already not trivial)

- ▶ exact *OPT* likely NP-hard

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

# Lower Bounds for OPT

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



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

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

# Tango Trees

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



Demaine, Harmon, Iacono, 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
- ▶ 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*
- ▶ but has not fueled the resolution of the Splay tree conjecture (yet?)

# Arborally Satisfied Point Sets

*Each proper rectangle must contain another point.*

## Definition 2.17 (Arborally Satisfied)

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

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

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

**BST-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.  $\blacktriangleleft$

# Minimum Arborally Satisfied Superset

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

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

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

# Greedy Arborally Satisfied Superset

## Natural Greedy Algorithm: GreedyASS

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

↪ 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

can be implemented as

- ▶ GreedyASS is 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**

### Theorem 2.19

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

# 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!
  
  - ▶ Can prove: Signed Greedy gives constant factor approximation to best independent-rectangle bound
- ≠ constant factor within  $OPT$  (at least not known!)

## **2.9 Deferred Data Structures**

# Motivation

All our adaptive dictionaries so far were binary search trees.

↪ *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!)*

↪ 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

↪ True. What if we never query (certain elements), though?

▶ Insertion-heavy workloads rather common scenario in applications

▶ many insertions, few queries ↪ most elements never queried for! *So why sort them?*

# The Multiple Selection Problem

## ▶ Selection by Rank:

- ▶ Given:  $n$  unsorted elements  $x_1, \dots, x_n$ ,
- ▶ Goal: find the  $r$ th smallest  $x_{(r)}$  (at (1-based) index  $r$  after sorting)

## ▶ Multiple Selection:

- ▶ Given:  $n$  unsorted elements  $x_1, \dots, x_n$ ,
- ▶ Goal: find  $q$  elements of ranks  $r_1 < r_2 < \dots < r_q$  from unsorted elements  $x_1, \dots, x_n$   
↪ Report sought elements  $x_{(r_1)}, \dots, x_{(r_q)}$  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

Answer: 3, 9, 15, 45

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
↑	↑	↑					↑							
$r_1$	$r_2$	$r_3$					$r_4$							
1	2	3					8							

# Recall: The Median-of-Medians Algorithm

```

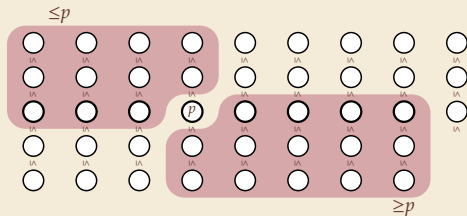
1 procedure choosePivotMoM(A[l..r]):
2   m := ⌊n/5⌋
3   for i := 0, ..., m - 1
4     sort(A[5i..5i + 4])
5     // collect median of 5
6     Swap A[i] and A[5i + 2]
7   return quickselectMoM(A[0..m], ⌊ $\frac{m-1}{2}$ ⌋)
8
9 procedure quickselectMoM(A[l..r], k):
10  if r - l ≤ 1 then return A[l]
11  b := 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  $m \leq \frac{n}{5}$  elements
2. recursive call inside quickselectMoM



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

$$\rightsquigarrow C(n) \leq \Theta(n) + C\left(\frac{1}{5}n\right) + C\left(\frac{7}{10}n\right)$$

$$\leq \Theta(n) + C\left(\frac{1}{5}n + \frac{7}{10}n\right)$$

ansatz: overall cost linear  $\rightarrow$

$$= \Theta(n) + C\left(\frac{9}{10}n\right) \rightsquigarrow C(n) = \Theta(n)$$

# 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  $x_1, \dots, x_n$  around it.

Recursively select  $r_1, \dots, r_{\lceil q/2 \rceil - 1}$  and  $r_{\lceil q/2 \rceil + 1}, \dots, r_q$

$\rightsquigarrow O(n \lg q)$

3. **Divide & Conquer over Elements:**

Find **median of**  $x_1, \dots, x_n$  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?*

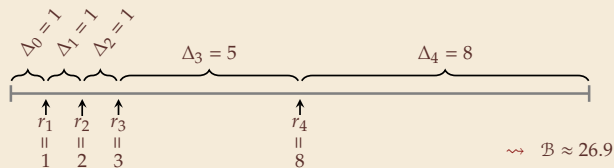
# Multiple Selection – Lower Bound

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



## Theorem 2.20 (Multiple Selection Lower Bound)

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

# Multiple Selection – Lower Bound Proof

**Proof:**

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

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

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

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

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  $x \rightarrow \infty$ , we get

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

# 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  $r == \emptyset$  return
3    $p := \text{median}(X[1..n])$  // linear-time selection
4    $X_1, X_2 := \text{partition}(X[1..n], p)$ 
5    $r := |X_1| + 1$  // rank of  $p$  in  $X$ 
6    $R_1, R_2 := \text{partition}(R[1..q], r)$ 
7   MultiSelect( $X_1, R_1$ )
8   if  $r \in R$  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  $r_1 < \dots < r_q$  among  $n$  unsorted elements in  $O(\mathcal{B} + n)$  time. ◀

# 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”

↪ 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  $\Delta_i$  into  $\Delta'_i = r'_i - r_i$  and  $\Delta'_{i+1} = r_{i+1} - r'_i$  costs

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



## 2.10 Lazy Search Trees

# Going dynamic

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

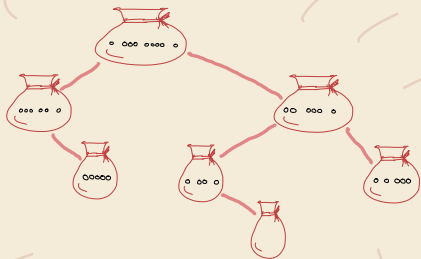
*How about maintaining a dynamic set subject to insertions and deletions?*

- ▶ Maintaining data in single array inconvenient

↪ 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
- ▶ Gaps  $\Delta_1, \dots, \Delta_m$  partition elements into bags (previously  $\Delta_i$  denoted only the size)

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

- ▶ queried elements are smallest or largest in gap
- ▶  $r_0 = 0, r_{q+1} = n \rightsquigarrow |\Delta_i| = r_i - r_{i-1}$

## 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.

### Lemma 2.23 (Query lower bound)

Suppose a query splits a gap  $\Delta_i$  into two gaps of size  $x$  and  $c \cdot x$  for  $c \geq 1$ .

Then the bound  $\mathcal{B} + n$  increases by  $\Omega(x \log c)$ .

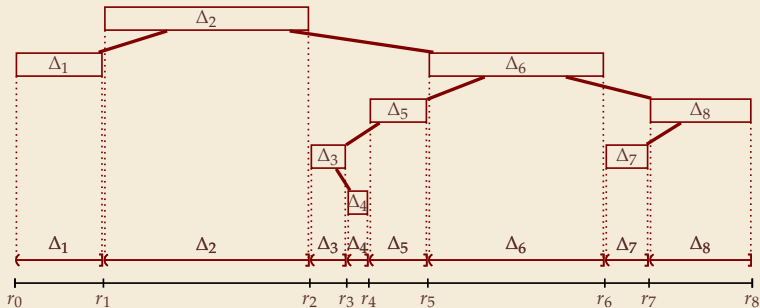
Moreover,  $\mathcal{B} = \Omega(m \log n)$  for  $m \geq 2$ , so a query has  $\Omega(\log n)$  amortized costs. ◀

### Lemma 2.24 (Insert lower bound)

Suppose we insert an element into gap  $\Delta_i$ .

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

# Step 1: Biased search tree over gaps



$r_1 < r_2 < \dots < r_q$  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  $\Delta_i$  in  $O\left(\log \frac{W}{w_i}\right) = O\left(\log \frac{n}{|\Delta_i|}\right)$

✓ Insert does find and changeWeight

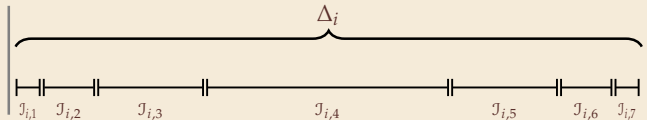
✓ Query finds gap and replace it by two new ones, but  $\Omega(\log n)$  increase in lower bound

# 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 . . .*

Must maintain further structure inside gaps: *intervals*



Smaller intervals  $\rightsquigarrow$  faster queries

More intervals  $\rightsquigarrow$  slower insertions

# Non-Solutions

- ▶ Collect insertions in buffers?
  - ⚡  $O(x \log c)$  amortized cost for queries?
  - ⚡ unclear how to balance gap data structure without flushing buffers
  
- ▶ Interval sizes roughly doubling as we move away from boundary.
  - ⚡ 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

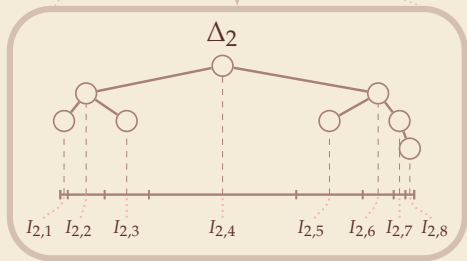
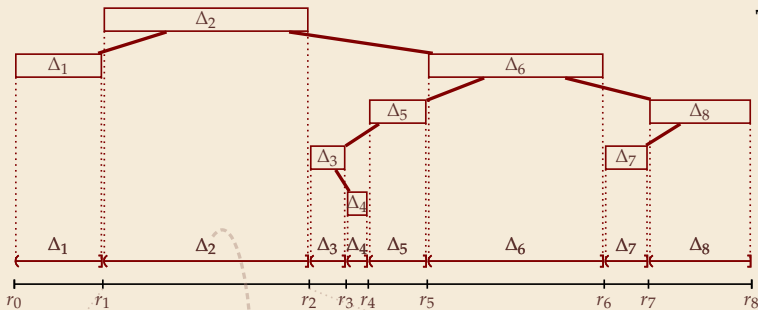
- ▶ Insert( $x$ ) in  $O(\min(\log(n/|\Delta_i|) + \log \log |\Delta_i|, \log q))$  worst-case time, where  $x \in \Delta_i$ .
- ▶ RankBasedQuery( $r$ ) in  $O(x \log c + \log n)$  amortized time, resulting gaps have size  $cx$  and  $x, c \geq 1$ .
- ▶ **Sequence of insert and query w/o duplicate queries**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

 Sandlund, Zhang: *Selectable Heaps and Optimal Lazy Search Trees*, SODA 2022

# Lazy Search Trees – Overview



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

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

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

## Three levels:

### 1. Gaps $\Delta_i$ (= bags)

► maintained in a *biased search tree*

☑ access  $\Delta_i$  in  
 $O(\log \frac{W}{w_i}) = O(\log \frac{n}{|\Delta_i|})$

### 2. Within $\Delta_i$ , 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  $\mathcal{J}_{i,j}$ : Unsorted list with values between two pivots

▶ **Left/Right Intervals:**

interval  $\mathcal{J}_{i,j}$  in gap  $\Delta_i$  is on the *left side* if the closest query rank is to the left  
Symmetrically for right.

▶ **Outside of an interval:**

$o(\mathcal{J}_{i,j}) = \# \text{elements } \underline{\text{outside}} \text{ of } \mathcal{J}_{i,j} = \min\{|\mathcal{J}_{i,1}| + \dots + |\mathcal{J}_{i,j-1}|, |\mathcal{J}_{i,j+1}| + \dots + |\mathcal{J}_{i,\ell_i}|\}$

▶ **Invariant (#Int):** In  $\Delta_i$ , the number  $\ell_i$  of intervals  $\mathcal{J}_{i,j}$  is at most  $\ell_i \leq 4 \log_2 |\Delta_i|$

▶ 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  $|\mathcal{J}_{i,j}| + |\mathcal{J}_{i,j+1}| \leq o(\mathcal{J}_{i,j})$ , merge  $\mathcal{J}_{i,j}$  and  $\mathcal{J}_{i,j+1}$ .

▶ 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.

## **(M) implies (#Int)**

### **Lemma 2.26**

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



## 2.11 Lazy Search Tree Operations

## Insert

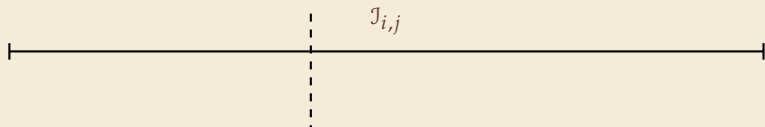
- (1) Find correct gap  $\Delta_i$  (in biased search tree)
- (2) Find correct interval  $\mathcal{J}_{i,j}$  (in balanced search tree)
- (3) Insert new element and increment weight of  $\Delta_i$

— no maintenance —

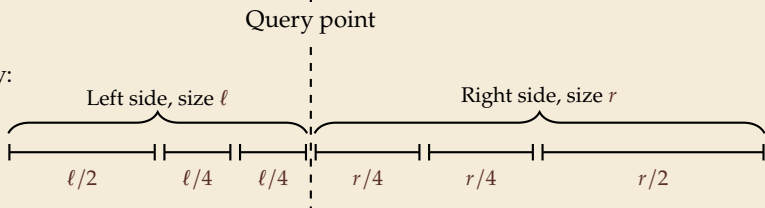
# Rank-Based Query

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

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

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

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

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

## Lemma 2.27

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



# Amortized Costs

full details spelled out in



Rysgaard, Wild: *Towards Lazy B-Trees*, MFCS 2025

## Lemma 2.28

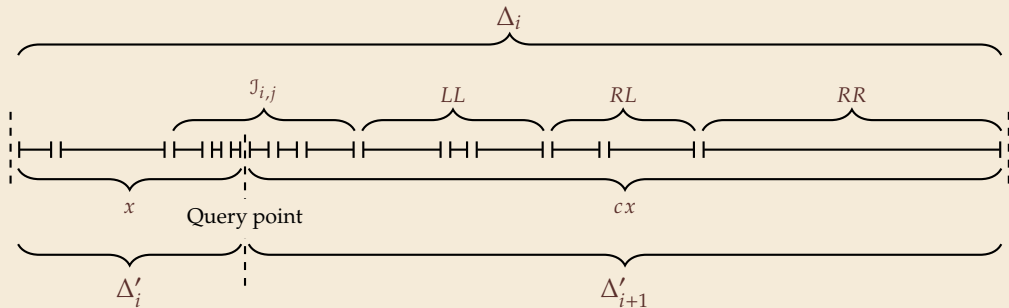
The amortized cost of  $\text{Insert}(x)$  into  $\Delta_i$  is  $O\left(\log \frac{n}{|\Delta_i|} + \log \log n\right)$  ◀

## Lemma 2.29

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

# Amortized Costs – Query Analysis

Case 1.1: 2-sided  $\Delta_i, \mathcal{J}_{i,j}$  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

👍 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 w. r. t. worst case of inserting all elements first, then doing all queries  
**OPEN:** *Can we build dynamic ordered dictionaries adaptive to insertion and query order?*