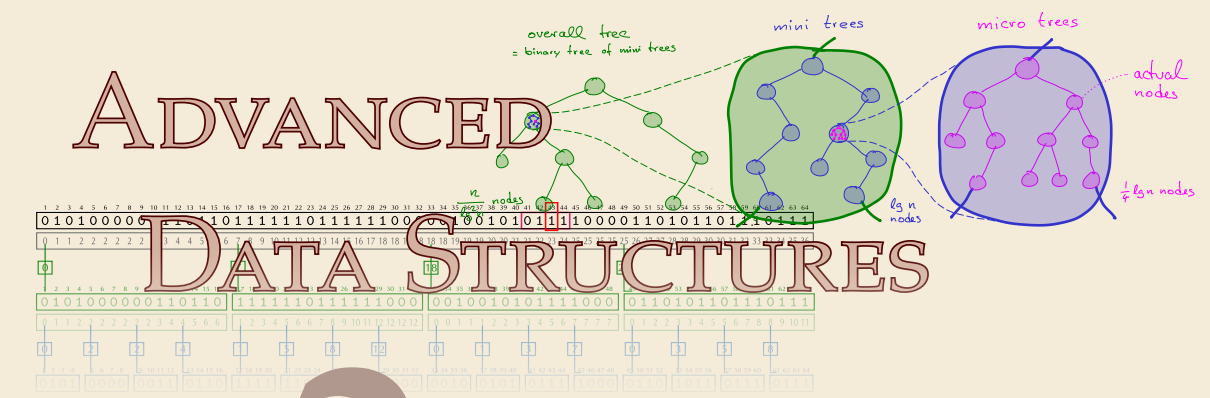


Advanced Data Structures

2 Adaptive Trees

Advanced Data Structures · Summer 2026

Prof. Dr. Sebastian Wild

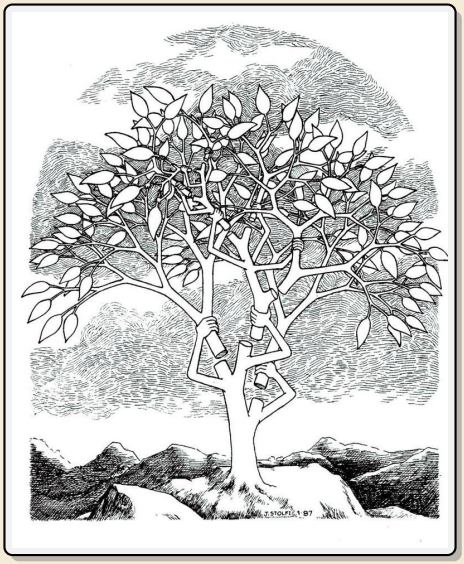
CS650 (Summer 2026)
Philipps-Universität Marburg
version 2026-05-12 00:58 H

Outline

2 Adaptive Trees

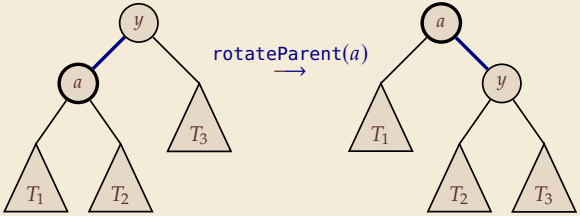
Self-Adjusting Data Structures

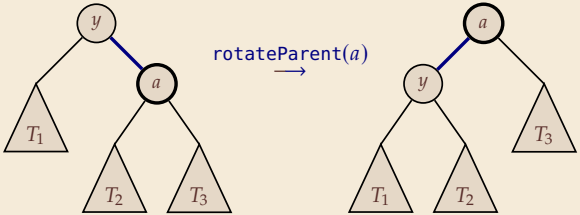
Idea: set of rules that makes data structure automatically adapt to use case



2.1 Move-to-Root Heuristic

Rotations


y
a
T1
T2
T3
rotateParent(a)
a
y
T1
T2
T3


y
a
T1
T2
T3
rotateParent(a)
a
y
T1
T2
T3

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!

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 $p_1, \ldots, p_n$.

$\rightsquigarrow$ 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

$\rightsquigarrow$ 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, $j = 0, \ldots, n$ 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 $\textcircled{i}$ is an **ancestor** of $\textcircled{j}$ iff the most recent request for i came after the most recent requests of any of $i+1, \ldots, j$.
Similarly, $\textcircled{j}$ ancestor of $\textcircled{i}$ iff j requested more recently than any of $i, \ldots, j-1$. ◀

Recall $A^i_j = [\textcircled{i}$ ancestor of $\textcircled{j}]$

Corollary 2.3

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

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.

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

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

Proof:

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

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

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

This proves the first part.

Move-to-Root on iid Accesses – Analysis

Proof (cont.):

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

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

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

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

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

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

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

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

■

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

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

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

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.

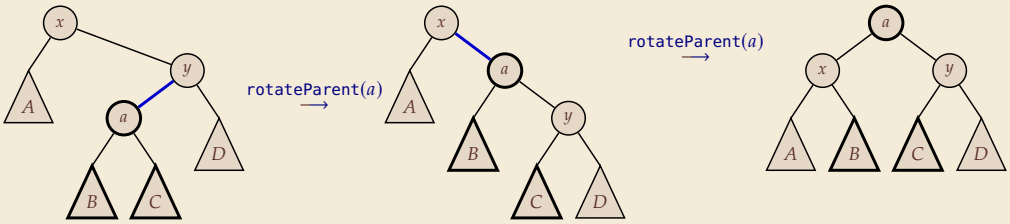
- reasonably simple and fast
- simple to implement
- remarkable theoretical properties (*access theorems*) $\rightsquigarrow$ later
- many writes and pointer chases

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

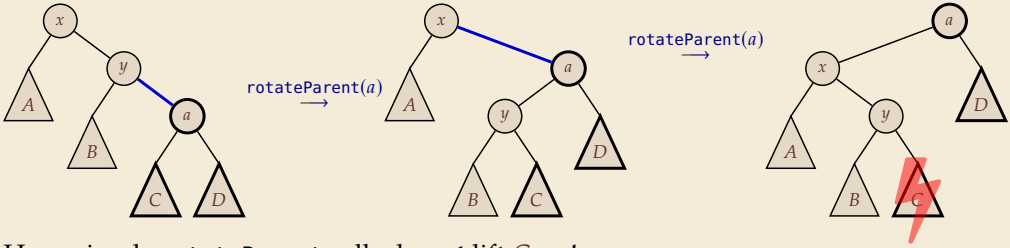
Move-to-root reloaded

What's wrong in Move-to-front?

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



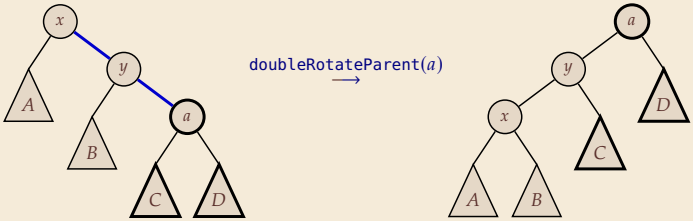
Using simple `rotateParent` calls, the subtree we came from (B or C) is lifted up!



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

The Fix

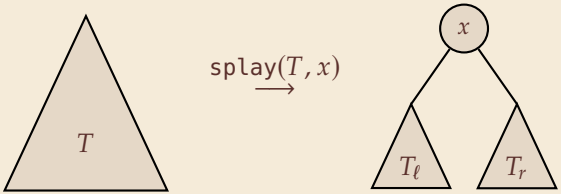
In the case that both parents are right children, do a double rotation



Now both C and D are lifted up!

Splay

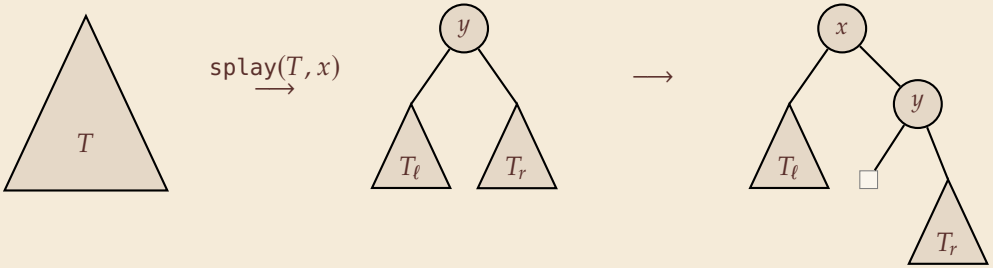
Aggressive restructuring allows to base all operations on single primitive: `splay`




```
def splay(T, x):
    while x ≠ T.root
        p = parent(x)
        if p == T.root
            rotateParent(x)  # Zig Case
        else:
            g = parent(p)
            if p == g.left ∧ x == p.left or p == g.right ∧ x == p.right:
                doubleRotateParent(x)  # Zig-Zig Case
            else:
                rotateParent(x)  # Zig-Zag Case
                rotateParent(x)
```

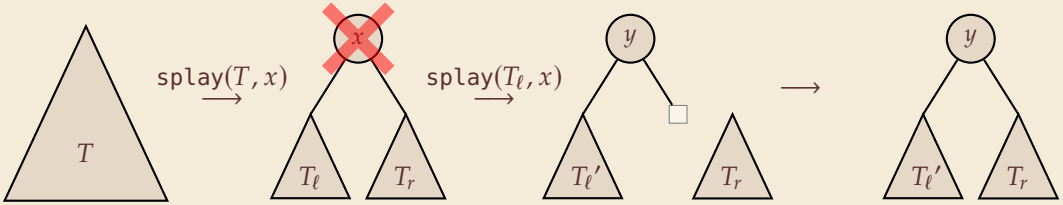
Splay – Insert

1. `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. `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 $y = \texttt{max}(T_\ell)$ and $y.right = \texttt{null}$
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 $(x.left, x)$

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.

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 `splay`(x) uses k rotations, then

$\Delta\Phi$ = change in potential

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

change in "rank" of x

- ⇝ Released potential can pay for actual cost k

Splay Tree Potential

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

- For that, we allow each node x to have a ***weight*** $w(x)$
- The ***size*** of a node x is defined as $\boxed{s(x) \coloneqq \sum_{\substack{v \in T: \\ x \text{ ancestor of } v}} w(v)}$
- The ***rank*** of node x is $\boxed{r(x) \coloneqq \lg(s(x))}$
- The ***potential*** of a tree is $\boxed{\Phi = \sum_{v \in T} r(v)}$
- 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

```
def splay(T, x):
    while x ≠ T.root
        p = parent(x)
        if p == T.root
            rotateParent(x)  # Zig Case
        else:
            g = parent(p)
            if p == g.left ∧ x == p.left or p == g.right ∧ x == p.right:
                doubleRotateParent(x)  # Zig-Zig Case
            else:
                rotateParent(x)  # Zig-Zag Case
                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

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

Access Lemma – Proof

Splay Cost

Corollary 2.7

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

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

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

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

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 $X = x_1, \ldots, x_m$ 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 $\mathcal{H} = \sum_{i=1}^{n} p_i \lg(1/p_i)$. ◀

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

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 $O\left(m + n \log n + \sum_{j=1}^{m} \log(1 + t(j))\right)$ ◀

What makes Splay tick?

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

A global view of Splay

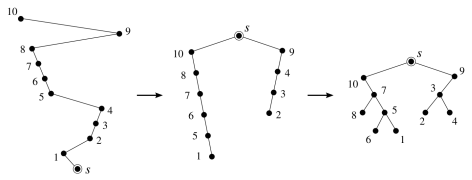


Fig. 2: A global view of splay trees. The transformation from the left to the middle illustrates rotate-to-root. The transformation from the left to the right illustrates splay trees.

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

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

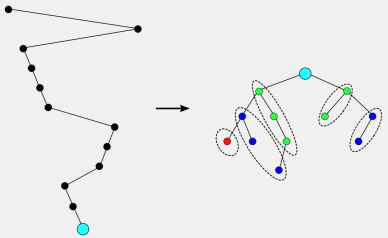
Intuition:

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

Anything magic about Splay?

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

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



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

Theorem. If, for every search:

(1) max left/right depth: $d = O(1)$,

(2) number of leaves+zigzags: $\ell + z = \Omega(|P|)$

$\Longrightarrow$ 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

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

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

◀

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 `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**

  - $\rightsquigarrow$ 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 $w : U \to \mathbb{R}_{>0}$, 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 $3 \lg\left(\frac{W}{w(x)}\right) + 1$.

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 ... *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
  $A(x) = \text{cost}\big(\mathcal{A}(x_1), \ldots, \mathcal{A}(x_1, \ldots, x_T)\big)$

$\rightsquigarrow$ **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)
- For the elevator problem, OPT would wait if $w \leq k-1$ 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

$c = \max_x \frac{A(x)}{OPT(x)}$ 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

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

$\rightsquigarrow$ 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

$\rightsquigarrow$ *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.*

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

Tango Trees

- Online BST algorithm specifically designed to stay close to one of Wilber's lower bounds

  details skipped

- achieves a competitive ratio of $O(\log \log n)$
- may use $\Omega(\log n \log \log n)$ amortized cost $\rightsquigarrow$ not $o(\log \log n)$-competitive

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

Greedy Future

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

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

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

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

2.8 The Geometric View of BSTs

Different Way of Looking at BSTs

Beautiful reformulation of problem

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

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

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: $|Y|$

- 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 $Y \cup X_t$

$\rightsquigarrow$ 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

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

$\rightsquigarrow$ 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!)

2.9 Deferred Data Structures

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?

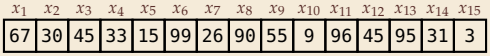
- Insertion-heavy workloads rather common scenario in applications
- many insertions, few queries $\rightsquigarrow$ most elements never queried for! *So why sort them?*

The Multiple Selection Problem

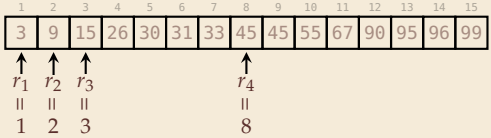
- **Selection by Rank:**
  - Given: n unsorted elements $x_1, \ldots, x_n$,
  - Goal: find the rth smallest $x_{(r)}$ (at (1-based) index r after sorting)

- **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:**



Answer: 3, 9, 15, 45



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?*

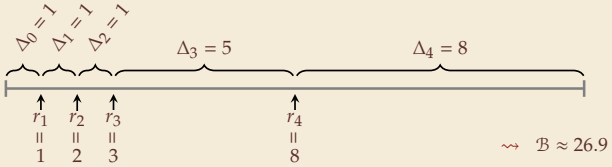
Multiple Selection – Lower Bound

. . . 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 $r_{q+1} = n+1)$



Theorem 2.20 (Multiple Selection Lower Bound)

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

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 $x_{(1)}, \ldots, x_{(r)}|$.
For $\mathcal{A}$ to be correct, it needs to learn how every other element compares to $x_{(1)}, \ldots, x_{(r)}|$.

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

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

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 == ∅ 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. ◀

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 **add** further ranks after the fact!

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

Multiple Selection Algorithms

One can indeed get very close to lower bound

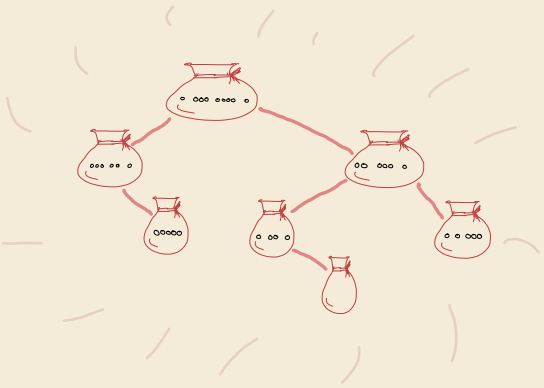
- $\mathcal{B} + o(\mathcal{B}) + O(n)$ *comparisons* suffice

  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

2.10 Lazy Search Trees

Wishful-Thinking Solution

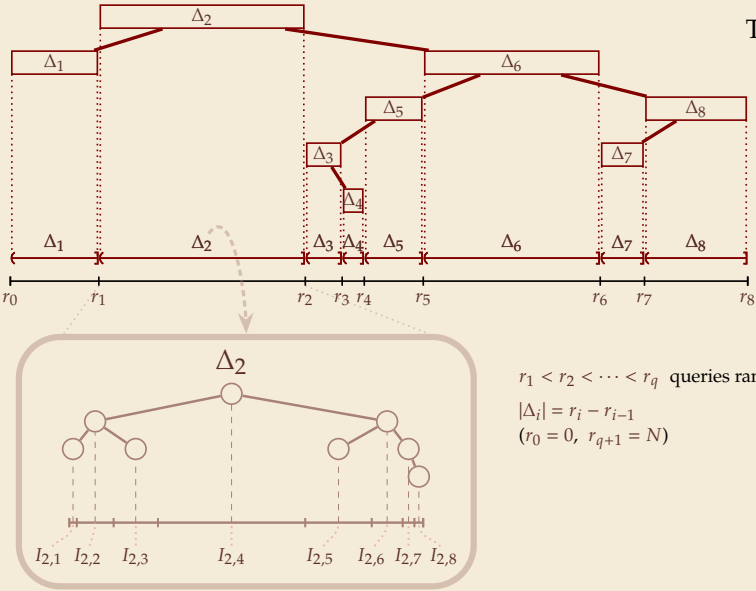


- BST of unsorted "bags"
- To answer query must look inside bag
- ⇝ partition one bag around query
- Collect insertions in buffer?

general rank-aware lower bound (see paper)

- ↯ $O(x \log c)$ amortized cost for queries?
- ↯ tricky to balance "tree of bags" without flushing buffers

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*** (by size)
   - ☑ access Δ_i in $O(\log \frac{W}{w_i}) = O(\log \frac{N}{|\Delta_i|})$
2. Within Δ_i, maintain set of *intervals* to fast queries
   - $O(\log |\Delta_i|)$ intervals
   - stored in BBST
   - ☑ split in $O(x \log c)$ (amortized)
3. Each interval is stored as *unsorted array*.