

Self-Adjusting BSTs

What Makes Them Tick?

Parinya Chalermsook¹

Mayank Goswami¹

László Kozma²

Kurt Mehlhorn¹

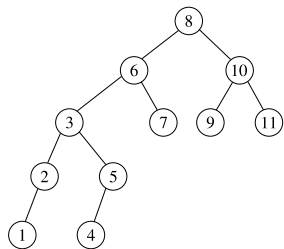
Thatchaphol Saranurak³

¹Max-Planck Institute for Informatics

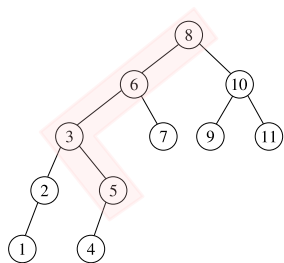
²Department of Computer Science, Saarland University

³KTH Royal Institute of Technology, Stockholm.

Introduction: Binary Search Trees

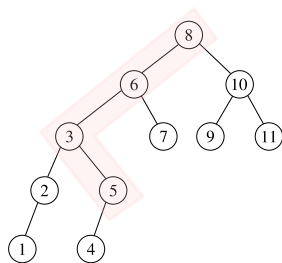


Introduction: Binary Search Trees



search(5)

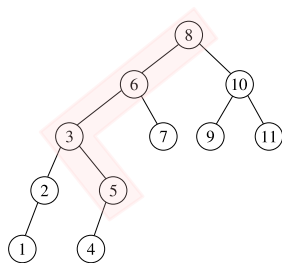
Introduction: Binary Search Trees



search(5)

cost \sim search path length

Introduction: Binary Search Trees

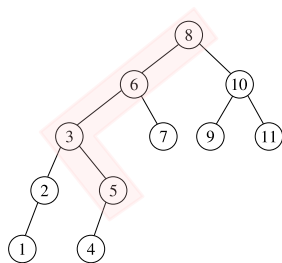


search(5)

cost \sim search path length

search(5), search(3), search(11),
search(3), search(1), ...

Introduction: Binary Search Trees



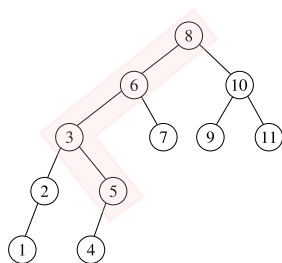
search(5)

cost \sim search path length

search(5), search(3), search(11),
search(3), search(1), ...

cost $\sim \sum$ of search path lengths

Introduction: Binary Search Trees



search(5)

cost \sim search path length

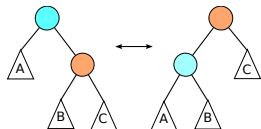
search(5), search(3), search(11),
search(3), search(1), ...

cost $\sim \sum$ of search path lengths

Optimal tree for a given sequence
(Knuth, 1971)

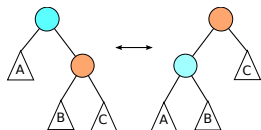
Introduction: Self-Adjusting BST

What if we can re-arrange the tree between accesses?



Introduction: Self-Adjusting BST

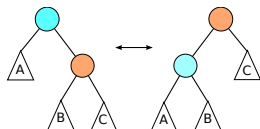
What if we can re-arrange the tree between accesses?



Example search sequence: $1, 2, 3, 4, \dots, n$.

Introduction: Self-Adjusting BST

What if we can re-arrange the tree between accesses?

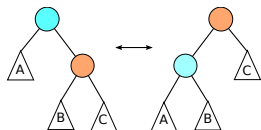


Example search sequence: $1, 2, 3, 4, \dots, n$.

Optimum tree cost: $\Omega(n \log n)$.

Introduction: Self-Adjusting BST

What if we can re-arrange the tree between accesses?



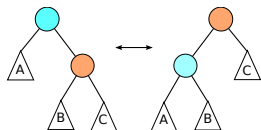
Example search sequence: $1, 2, 3, 4, \dots, n$.

Optimum tree cost: $\Omega(n \log n)$.

Optimum with re-arrangement: $O(n)$.

Introduction: Self-Adjusting BST

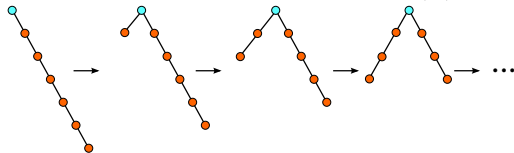
What if we can re-arrange the tree between accesses?



Example search sequence: $1, 2, 3, 4, \dots, n$.

Optimum tree cost: $\Omega(n \log n)$.

Optimum with re-arrangement: $O(n)$.



Splay Trees (Sleator and Tarjan, 1983)

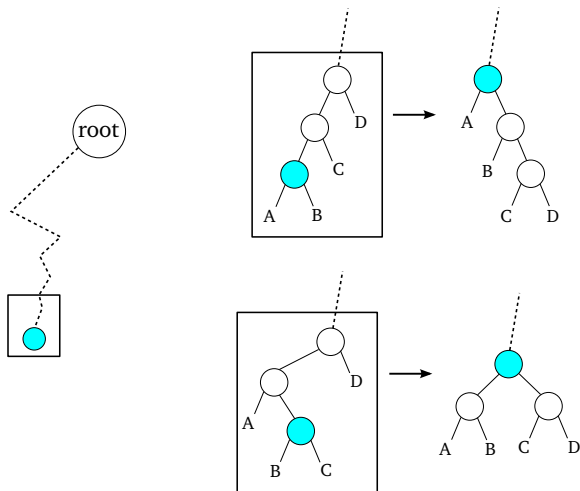
After each access:

Re-arrange search path bottom-up, until accessed element is root.

Splay Trees (Sleator and Tarjan, 1983)

After each access:

Re-arrange search path bottom-up, until accessed element is root.



Splay Trees (Sleator and Tarjan, 1983)

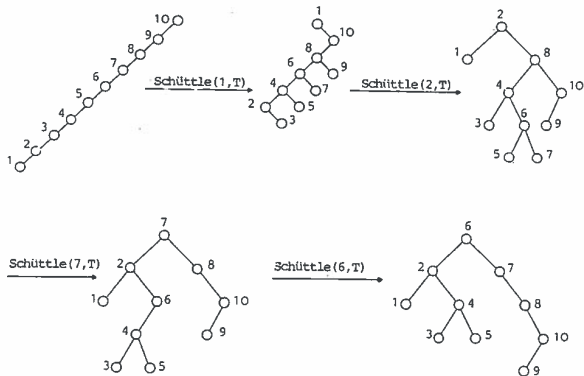


Abb. 95. 4 Schüttle-Operationen.

Splay Trees (Sleator and Tarjan, 1983)

Splay trees have many nice properties:

- amortized logarithmic cost

Splay Trees (Sleator and Tarjan, 1983)

Splay trees have many nice properties:

- amortized logarithmic cost
- static optimality

Splay Trees (Sleator and Tarjan, 1983)

Splay trees have many nice properties:

- amortized logarithmic cost
- static optimality
- working set property
- static finger property

Splay Trees (Sleator and Tarjan, 1983)

Splay trees have many nice properties:

- amortized logarithmic cost
- static optimality
- working set property
- static finger property

- sequential access

Splay Trees (Sleator and Tarjan, 1983)

Splay trees have many nice properties:

- amortized logarithmic cost
- static optimality
- working set property
- static finger property

- sequential access
- ...

- conjectured: dynamic optimality

Splay Trees (Sleator and Tarjan, 1983)

Splay trees have many nice properties:

- amortized logarithmic cost
- static optimality
- working set property
- static finger property

- sequential access
- ...

- conjectured: dynamic optimality

First 4 properties captured by **access lemma**.

Splay Trees (Sleator and Tarjan, 1983)

Access Lemma for Splay Trees (simplified, unweighted)

Let P be the path to the accessed element s in T , and let T' be the tree after splaying s . Then

$$|P| \leq O(1) \cdot \log n + \Phi(T) - \Phi(T').$$

Splay Trees (Sleator and Tarjan, 1983)

Access Lemma for Splay Trees (simplified, unweighted)

Let P be the path to the accessed element s in T , and let T' be the tree after splaying s . Then

$$|P| \leq O(1) \cdot \log n + \Phi(T) - \Phi(T').$$

Sum-of-Logs Potential (simplified, unweighted)

For any $a \in T$, let T_a be the subtree rooted at a . Then

$$\Phi(T) = \sum_{a \in T} \log |T_a|$$

Splay Trees (Sleator and Tarjan, 1983)

Access Lemma for Splay Trees (simplified, unweighted)

Let P be the path to the accessed element s in T , and let T' be the tree after splaying s . Then

$$|P| \leq O(1) \cdot \log n + \Phi(T) - \Phi(T').$$

Sum-of-Logs Potential (simplified, unweighted)

For any $a \in T$, let T_a be the subtree rooted at a . Then

$$\Phi(T) = \sum_{a \in T} \log |T_a|$$

Implies logarithmic amortized cost. (A weighted version implies the other properties.)

Splay Trees (Sleator and Tarjan, 1983)

Access Lemma for Splay Trees (simplified, unweighted)

Let P be the path to the accessed element s in T , and let T' be the tree after splaying s . Then

$$|P| \leq O(1) \cdot \log n + \Phi(T) - \Phi(T').$$

Sum-of-Logs Potential (**weighted**)

Let $w : T \rightarrow \mathbb{R}_{>0}$, and $w(S) = \sum_{a \in S} w(a)$. Then

$$\Phi(T) = \sum_{a \in T} \log w(T_a)$$

Implies logarithmic amortized cost. (A weighted version implies the other properties.)

Splay Trees (Sleator and Tarjan, 1983)

Access Lemma for Splay Trees (**weighted**)

Let P be the path to the accessed element s in T , and let T' be the tree after splaying s . Then

$$|P| \leq O(1) \cdot \log \frac{w(T)}{w(s)} + \Phi(T) - \Phi(T').$$

Sum-of-Logs Potential (**weighted**)

Let $w : T \rightarrow \mathbb{R}_{>0}$, and $w(S) = \sum_{a \in S} w(a)$. Then

$$\Phi(T) = \sum_{a \in T} \log w(T_a)$$

Implies logarithmic amortized cost. (A weighted version implies the other properties.)

Splay Trees (Sleator and Tarjan, 1983)

Access Lemma for Splay Trees (**weighted**)

Let P be the path to the accessed element s in T , and let T' be the tree after splaying s . Then

$$|P| \leq O(1) \cdot \log \frac{w(T)}{w(s)} + \Phi(T) - \Phi(T').$$

Sum-of-Logs Potential (**simplified, unweighted**)

For any $a \in T$, let T_a be the subtree rooted at a . Then

$$\Phi(T) = \sum_{a \in T} \log |T_a|$$

Implies logarithmic amortized cost. (A weighted version implies the other properties.)

Splay Trees (Sleator and Tarjan, 1983)

Access Lemma for Splay Trees (simplified, unweighted)

Let P be the path to the accessed element s in T , and let T' be the tree after splaying s . Then

$$|P| \leq O(1) \cdot \log n + \Phi(T) - \Phi(T').$$

Sum-of-Logs Potential (simplified, unweighted)

For any $a \in T$, let T_a be the subtree rooted at a . Then

$$\Phi(T) = \sum_{a \in T} \log |T_a|$$

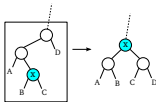
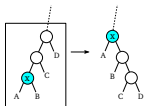
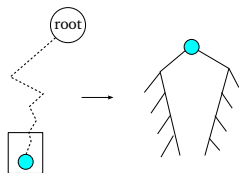
Implies logarithmic amortized cost. (A weighted version implies the other properties.)

Splay Trees (Sleator and Tarjan, 1983)

Splay trees satisfy access lemma (proof sketch):

Splay Trees (Sleator and Tarjan, 1983)

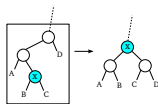
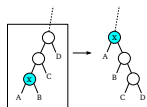
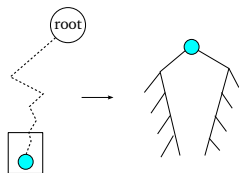
Splay trees satisfy access lemma (proof sketch):



Splay Trees (Sleator and Tarjan, 1983)

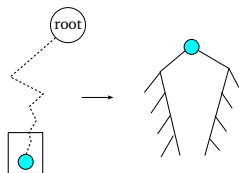
Splay trees satisfy access lemma (proof sketch):

- Reminder: $\Phi(T) = \sum_{a \in T} \log |T_a|$

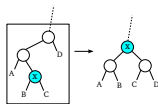
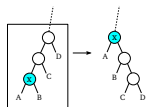


Splay Trees (Sleator and Tarjan, 1983)

Splay trees satisfy access lemma (proof sketch):

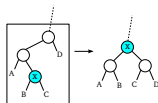
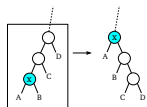
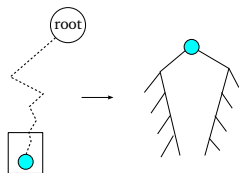


- Reminder: $\Phi(T) = \sum_{a \in T} \log |T_a|$
- only nodes on the search path change potential



Splay Trees (Sleator and Tarjan, 1983)

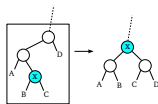
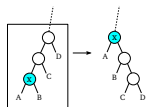
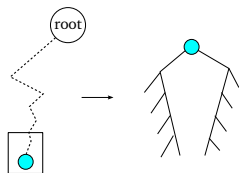
Splay trees satisfy access lemma (proof sketch):



- Reminder: $\Phi(T) = \sum_{a \in T} \log |T_a|$
- only nodes on the search path change potential
- for the nodes in a single transformation:
 $\Phi' - \Phi \leq \Phi'(X) - \Phi(X) - C.$
(follows from subtree-inclusions and properties of \log)

Splay Trees (Sleator and Tarjan, 1983)

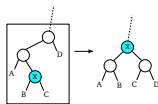
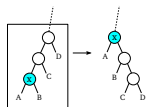
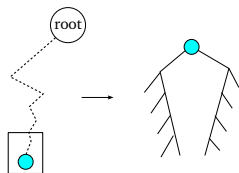
Splay trees satisfy access lemma (proof sketch):



- Reminder: $\Phi(T) = \sum_{a \in T} \log |T_a|$
- only nodes on the search path change potential
- for the nodes in a single transformation:
$$\Phi' - \Phi \leq \Phi'(X) - \Phi(X) - C.$$
(follows from subtree-inclusions and properties of \log)
- for the entire re-arrangement this telescopes into:
$$\Phi(T') - \Phi(T) \leq \underbrace{\Phi'(root)}_{\leq \log(n)} - c \cdot |P|.$$

Splay Trees (Sleator and Tarjan, 1983)

Splay trees satisfy access lemma (proof sketch):



- Reminder: $\Phi(T) = \sum_{a \in T} \log |T_a|$
- only nodes on the search path change potential
- for the nodes in a single transformation:
$$\Phi' - \Phi \leq \Phi'(X) - \Phi(X) - C.$$
(follows from subtree-inclusions and properties of \log)
- for the entire re-arrangement this telescopes into:
$$\Phi(T') - \Phi(T) \leq \underbrace{\Phi'(root)}_{\leq \log(n)} - c \cdot |P|.$$
- similarly for weighted case

Questions:

- Why are splay trees efficient?
- Are there other efficient re-arrangements?

Questions:

- Why are splay trees efficient?
- Are there other efficient re-arrangements?

[Subramanian, 1996; Georgakopoulos, McClurkin, 2004]:

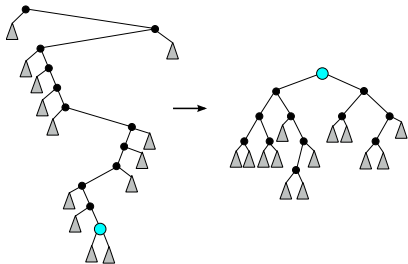
A family of local transformations (templates):

- of constant size
- applied on the search path bottom-up
- fulfilling certain natural conditions

guarantee the access lemma (proof similar to splay proof).

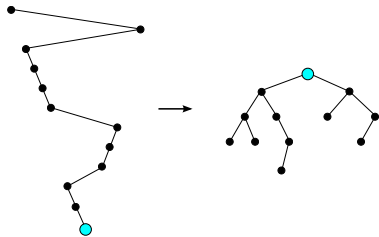
A more general view:

A more general view:



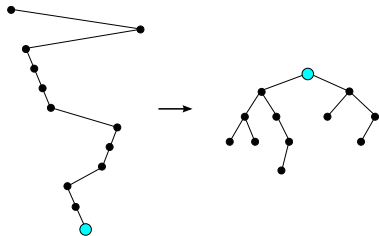
- (search path \rightarrow after-tree)
re-arrangement

A more general view:



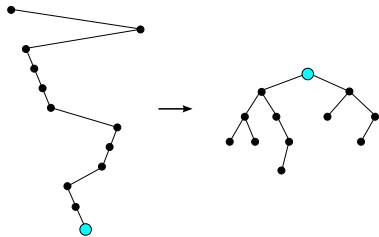
- (search path \rightarrow after-tree)
re-arrangement
- cost of re-arrangement is
absorbed in access cost.

A more general view:



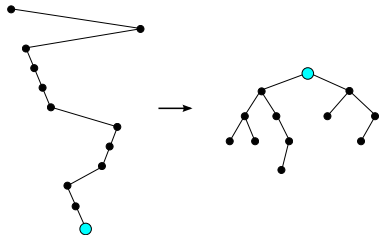
- (search path \rightarrow after-tree) re-arrangement
- cost of re-arrangement is absorbed in access cost.
- **which re-arrangements lead to an efficient algorithm?**

A more general view:



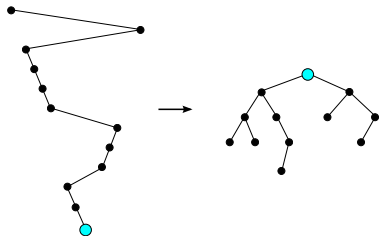
- (search path \rightarrow after-tree) re-arrangement
- cost of re-arrangement is absorbed in access cost.
- **which re-arrangements lead to an efficient algorithm?**
- ideas from the literature: **path balance** (Sleator),
depth-halving (Sleator, Tarjan, Subramanian, Georgakopoulos),
...

Properties of (search path \rightarrow after-tree):

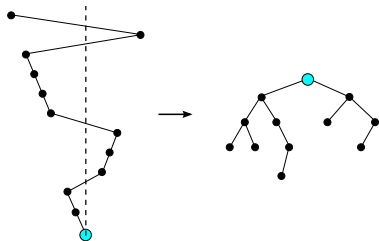


Properties of (search path \rightarrow after-tree):

- length of the search path: $|P|$

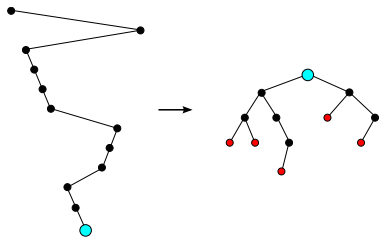


Properties of (search path \rightarrow after-tree):



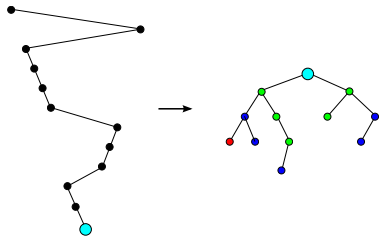
- length of the search path: $|P|$
- number of zigzags: z

Properties of (search path \rightarrow after-tree):



- length of the search path: $|P|$
- number of zigzags: z
- number of leaves: ℓ

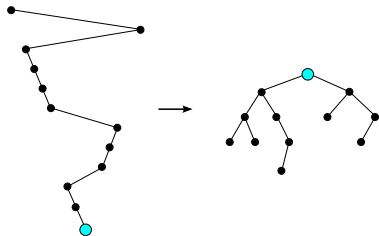
Properties of (search path \rightarrow after-tree):



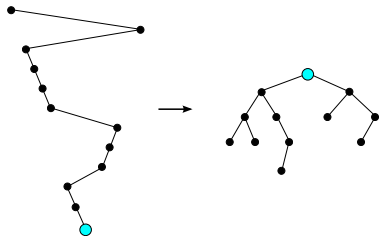
- length of the search path: $|P|$
- number of zigzags: z
- number of leaves: ℓ
- max right- or left-depth: d

Main result (simplified):

If



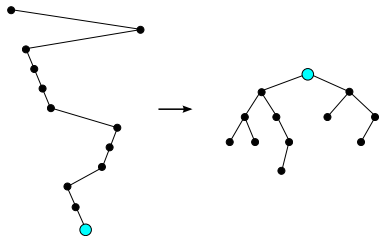
Main result (simplified):



If

① accessed element goes to root,

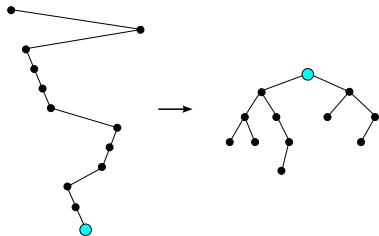
Main result (simplified):



If

- 1 accessed element goes to root,
- 2 the max right- or left-depth of the after-tree is $d = O(1)$,

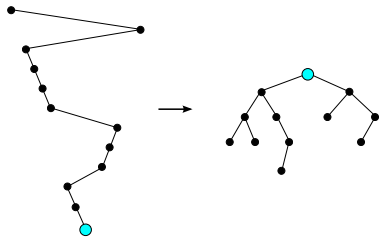
Main result (simplified):



If

- 1 accessed element goes to root,
- 2 the max right- or left-depth of the after-tree is $d = O(1)$,
- 3 the number of leaves in the after-tree is $\ell = \Omega(|P| - z)$,

Main result (simplified):



If

- 1 accessed element goes to root,
- 2 the max right- or left-depth of the after-tree is $d = O(1)$,
- 3 the number of leaves in the after-tree is $\ell = \Omega(|P| - z)$,

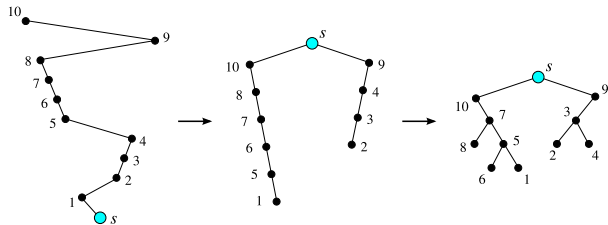
\implies access lemma (weighted version).

Application 1: Splay Trees

Splay trees - global view:

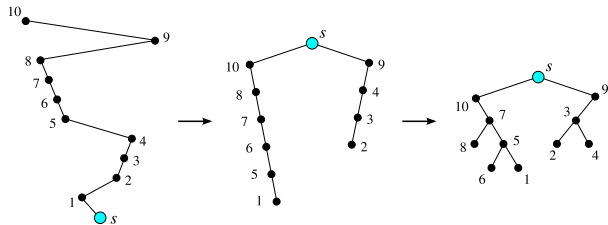
Application 1: Splay Trees

Splay trees - global view:



Application 1: Splay Trees

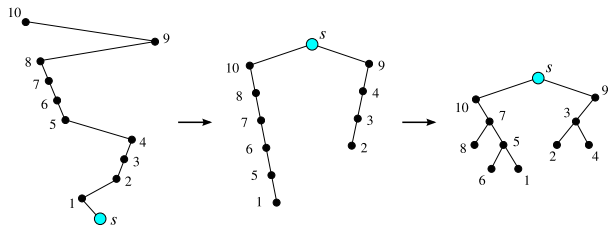
Splay trees - global view:



① accessed element to root ✓

Application 1: Splay Trees

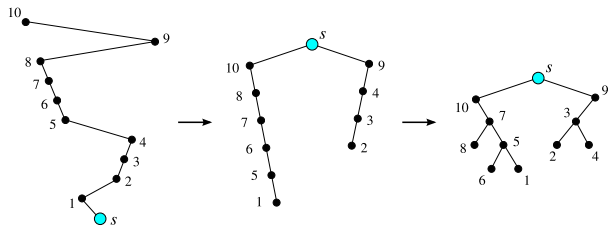
Splay trees - global view:



- 1 accessed element to root ✓
- 2 max right-(left) depth is $d = 2$ ✓

Application 1: Splay Trees

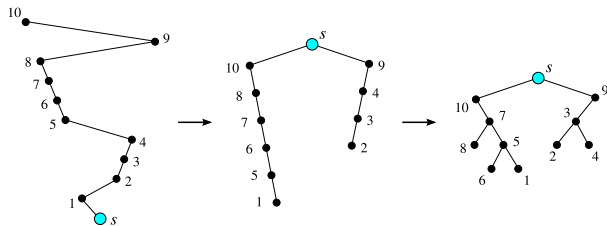
Splay trees - global view:



- 1 accessed element to root ✓
- 2 max right-(left) depth is $d = 2$ ✓
- 3 $z + \ell = |P|/2$ ✓

Application 1: Splay Trees

Splay trees - global view:



- 1 accessed element to root ✓
- 2 max right-(left) depth is $d = 2$ ✓
- 3 $z + \ell = |P|/2$ ✓

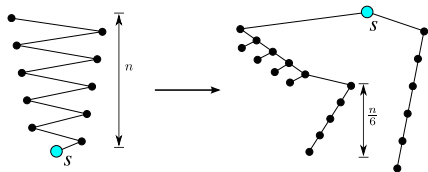
Similarly for Subramanian's and Georgakopoulos and McClurkin's generalizations.

Application 2: Depth halving

In splay every node on the search path roughly halves its depth.
Is this property sufficient?

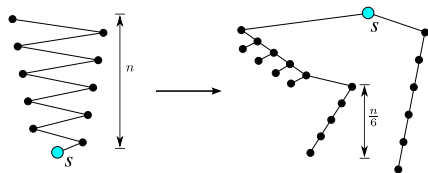
Application 2: Depth halving

In splay every node on the search path roughly halves its depth. Is this property sufficient?



Application 2: Depth halving

In splay every node on the search path roughly halves its depth. Is this property sufficient?



① max left-depth is linear! \times

Application 2: Depth halving

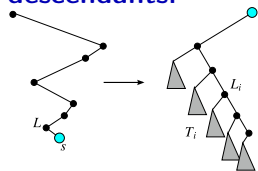
A stricter depth-halving is sufficient.

Every node x on the search path loses at least $(1/2 + \epsilon)d(x) - O(1)$ ancestors and gains at most $O(1)$ new descendants.

Application 2: Depth halving

A stricter depth-halving is sufficient.

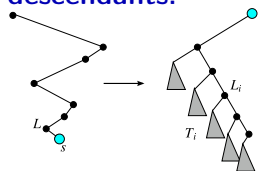
Every node x on the search path loses at least $(1/2 + \epsilon)d(x) - O(1)$ ancestors and gains at most $O(1)$ new descendants.



Application 2: Depth halving

A stricter depth-halving is sufficient.

Every node x on the search path loses at least $(1/2 + \epsilon)d(x) - O(1)$ ancestors and gains at most $O(1)$ new descendants.

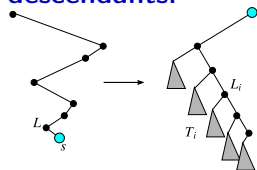


Nodes in T_i are new descendants of L_i . Thus $|T_i| = O(1)$ and hence left-depth of left side = $O(1)$. \checkmark

Application 2: Depth halving

A stricter depth-halving is sufficient.

Every node x on the search path loses at least $(1/2 + \epsilon)d(x) - O(1)$ ancestors and gains at most $O(1)$ new descendants.



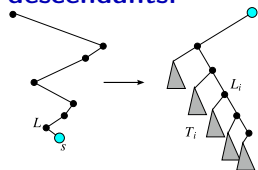
Nodes in T_i are new descendants of L_i . Thus $|T_i| = O(1)$ and hence left-depth of left side = $O(1)$. ✓

Assume $d_\ell(s) \geq d_r(s)$: thus $d_r(L) \leq d(s)/2$.

Application 2: Depth halving

A stricter depth-halving is sufficient.

Every node x on the search path loses at least $(1/2 + \epsilon)d(x) - O(1)$ ancestors and gains at most $O(1)$ new descendants.



Nodes in T_i are new descendants of L_i . Thus $|T_i| = O(1)$ and hence left-depth of left side = $O(1)$. \checkmark

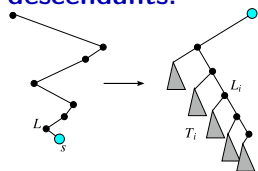
Assume $d_\ell(s) \geq d_r(s)$: thus $d_r(L) \leq d(s)/2$.

L loses $(1/2 + \epsilon)d(L) - O(1)$ ancestors, hence $(1/2 + \epsilon)d(L) - O(1) - d_r(L) \geq \epsilon d(s) - O(1)$ left ancestors.

Application 2: Depth halving

A stricter depth-halving is sufficient.

Every node x on the search path loses at least $(1/2 + \epsilon)d(x) - O(1)$ ancestors and gains at most $O(1)$ new descendants.



Nodes in T_i are new descendants of L_i . Thus $|T_i| = O(1)$ and hence left-depth of left side = $O(1)$. ✓

Assume $d_\ell(s) \geq d_r(s)$: thus $d_r(L) \leq d(s)/2$.

L loses $(1/2 + \epsilon)d(L) - O(1)$ ancestors, hence $(1/2 + \epsilon)d(L) - O(1) - d_r(L) \geq \epsilon d(s) - O(1)$ left ancestors.

Thus $(\epsilon d(s) - O(1))/O(1) = \Omega(d(s))$ trees T_i , each one having at least one leaf. ✓

Application 2: Depth halving (continued)

Every node x on the search path loses at least $(1/2 + \epsilon)d(x) - O(1)$ ancestors and gains at most $O(1)$ new descendants.

Application 2: Depth halving (continued)

Every node x on the search path loses at least $(1/2 + \epsilon)d(x) - O(1)$ ancestors and gains at most $O(1)$ new descendants.

The parameters appear to be tight.

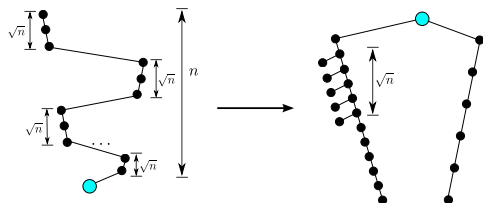
Example: without the $+\epsilon$ the conditions do not hold.

Application 2: Depth halving (continued)

Every node x on the search path loses at least $(1/2 + \epsilon)d(x) - O(1)$ ancestors and gains at most $O(1)$ new descendants.

The parameters appear to be tight.

Example: without the $+\epsilon$ the conditions do not hold.

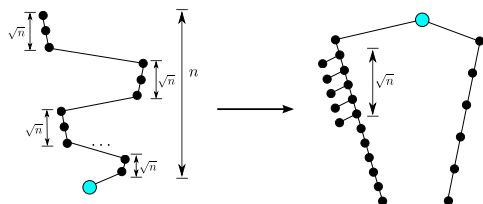


Application 2: Depth halving (continued)

Every node x on the search path loses at least $(1/2 + \epsilon)d(x) - O(1)$ ancestors and gains at most $O(1)$ new descendants.

The parameters appear to be tight.

Example: without the $+\epsilon$ the conditions do not hold.



Open Question 1: Other natural heuristics for which the access lemma holds?

Application 3: Path Balance

Path Balance heuristic [Sleator]: transform search path into a balanced BST.

Application 3: Path Balance

Path Balance heuristic [Sleator]: transform search path into a balanced BST.

Best bound on amortized cost: $O(\log n \log \log n / \log \log \log n)$.
[Balasubramanian, Raman, 1995]

Application 3: Path Balance

Path Balance heuristic [Sleator]: transform search path into a balanced BST.

Best bound on amortized cost: $O(\log n \log \log n / \log \log \log n)$.
[Balasubramanian, Raman, 1995]

Our conditions:

Application 3: Path Balance

Path Balance heuristic [Sleator]: transform search path into a balanced BST.

Best bound on amortized cost: $O(\log n \log \log n / \log \log \log n)$.
[Balasubramanian, Raman, 1995]

Our conditions:

- ① accessed element to root ✓

Application 3: Path Balance

Path Balance heuristic [Sleator]: transform search path into a balanced BST.

Best bound on amortized cost: $O(\log n \log \log n / \log \log \log n)$.
[Balasubramanian, Raman, 1995]

Our conditions:

- 1 accessed element to root ✓
- 2 max right-(left) depth is $d = \log |P| \times$

Application 3: Path Balance

Path Balance heuristic [Sleator]: transform search path into a balanced BST.

Best bound on amortized cost: $O(\log n \log \log n / \log \log \log n)$.
[Balasubramanian, Raman, 1995]

Our conditions:

- 1 accessed element to root ✓
- 2 max right-(left) depth is $d = \log |P| \times$
- 3 $z + \ell \geq |P|/2$ ✓

Application 3: Path Balance

Path Balance heuristic [Sleator]: transform search path into a balanced BST.

Best bound on amortized cost: $O(\log n \log \log n / \log \log \log n)$.
[Balasubramanian, Raman, 1995]

Our conditions:

- 1 accessed element to root ✓
- 2 max right-(left) depth is $d = \log |P| \times$
- 3 $z + \ell \geq |P|/2$ ✓

A parameterized version of our theorem (simplified):

$$|P| \leq \Phi(T) - \Phi(T') + O(d) \cdot \log n.$$

Application 3: Path Balance

Path Balance heuristic [Sleator]: transform search path into a balanced BST.

Best bound on amortized cost: $O(\log n \log \log n / \log \log \log n)$.
[Balasubramanian, Raman, 1995]

Our conditions:

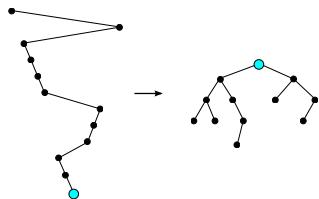
- 1 accessed element to root ✓
- 2 max right-(left) depth is $d = \log |P| \times$
- 3 $z + \ell \geq |P|/2$ ✓

A parameterized version of our theorem (simplified):

$$|P| \leq \Phi(T) - \Phi(T') + O(d) \cdot \log n.$$

It follows that the total cost of m accesses is $O((m + n) \log n \log \log n)$.

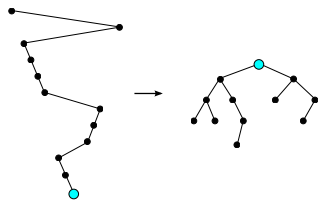
Proof sketch of main theorem



Conditions:

- 1 the max right- or left-depth of the after-tree is $d = O(1)$,
- 2 the number of leaves in the after-tree is $\ell = \Omega(|P| - z)$.

Proof sketch of main theorem

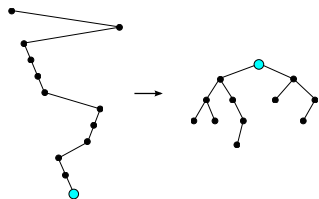


Conditions:

- 1 the max right- or left-depth of the after-tree is $d = O(1)$,
- 2 the number of leaves in the after-tree is $\ell = \Omega(|P| - z)$.

Reminder: $\Phi(T) = \sum_{a \in T} \log |T_a|$.

Proof sketch of main theorem



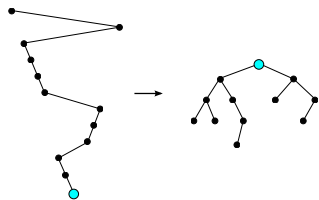
Conditions:

- 1 the max right- or left-depth of the after-tree is $d = O(1)$,
- 2 the number of leaves in the after-tree is $\ell = \Omega(|P| - z)$.

Reminder: $\Phi(T) = \sum_{a \in T} \log |T_a|$.

Want: $\Phi(T') - \Phi(T) \leq O(1) \cdot \log n - |P|$.

Proof sketch of main theorem



Conditions:

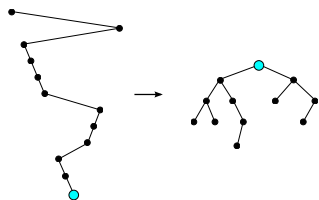
- 1 the max right- or left-depth of the after-tree is $d = O(1)$,
- 2 the number of leaves in the after-tree is $\ell = \Omega(|P| - z)$.

Reminder: $\Phi(T) = \sum_{a \in T} \log |T_a|$.

Want: $\Phi(T') - \Phi(T) \leq O(1) \cdot \log n - |P|$.

Contribution to $\Phi(T') - \Phi(T)$ of:

Proof sketch of main theorem



Conditions:

- 1 the max right- or left-depth of the after-tree is $d = O(1)$,
- 2 the number of leaves in the after-tree is $\ell = \Omega(|P| - z)$.

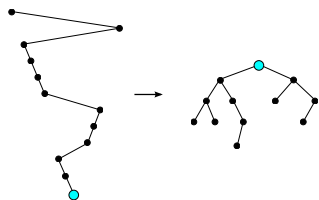
Reminder: $\Phi(T) = \sum_{a \in T} \log |T_a|$.

Want: $\Phi(T') - \Phi(T) \leq O(1) \cdot \log n - |P|$.

Contribution to $\Phi(T') - \Phi(T)$ of:

- a neighborhood-disjoint set (set of nodes with disjoint subtrees)

Proof sketch of main theorem



Conditions:

- 1 the max right- or left-depth of the after-tree is $d = O(1)$,
- 2 the number of leaves in the after-tree is $\ell = \Omega(|P| - z)$.

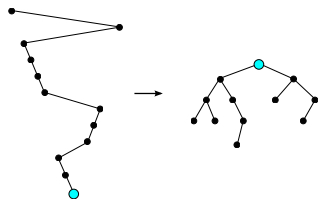
Reminder: $\Phi(T) = \sum_{a \in T} \log |T_a|$.

Want: $\Phi(T') - \Phi(T) \leq O(1) \cdot \log n - |P|$.

Contribution to $\Phi(T') - \Phi(T)$ of:

- a neighborhood-disjoint set (set of nodes with disjoint subtrees)
- a monotone set (set of nodes with same left-(right) depth)

Proof sketch of main theorem



Conditions:

- 1 the max right- or left-depth of the after-tree is $d = O(1)$,
- 2 the number of leaves in the after-tree is $\ell = \Omega(|P| - z)$.

Reminder: $\Phi(T) = \sum_{a \in T} \log |T_a|$.

Want: $\Phi(T') - \Phi(T) \leq O(1) \cdot \log n - |P|$.

Contribution to $\Phi(T') - \Phi(T)$ of:

- a neighborhood-disjoint set (set of nodes with disjoint subtrees)
- a monotone set (set of nodes with same left-(right) depth)
- a zig-zag set (set of alternating pairs in search path)

Proof sketch of main theorem (continued)

Contribution to $\Phi(T') - \Phi(T)$ of:

Proof sketch of main theorem (continued)

Contribution to $\Phi(T') - \Phi(T)$ of:

- a neighborhood-disjoint set X
 - nodes with disjoint subtrees in the after-tree
 - example: leaves of the after-tree
 - we prove: $\Phi_{T'}(X) - \Phi_T(X) \leq O(1) \cdot \log n - |X|$

Proof sketch of main theorem (continued)

Contribution to $\Phi(T') - \Phi(T)$ of:

- a neighborhood-disjoint set X
 - nodes with disjoint subtrees in the after-tree
 - example: leaves of the after-tree
 - we prove: $\Phi_{T'}(X) - \Phi_T(X) \leq O(1) \cdot \log n - |X|$
- a monotone set X
 - nodes with same left-(right) depth
 - we prove: $\Phi_{T'}(X) - \Phi_T(X) \leq O(1) \cdot \log n$

Proof sketch of main theorem (continued)

Contribution to $\Phi(T') - \Phi(T)$ of:

- a neighborhood-disjoint set X
 - nodes with disjoint subtrees in the after-tree
 - example: leaves of the after-tree
 - we prove: $\Phi_{T'}(X) - \Phi_T(X) \leq O(1) \cdot \log n - |X|$
- a monotone set X
 - nodes with same left-(right) depth
 - we prove: $\Phi_{T'}(X) - \Phi_T(X) \leq O(1) \cdot \log n$
- a zig-zag set Z
 - set of alternating pairs in search path
 - we prove: $\Phi_{T'}(X) - \Phi_T(X) \leq O(1) \cdot \log n - |X|$

Proof sketch of main theorem (continued)

Contribution to $\Phi(T') - \Phi(T)$ of:

- a neighborhood-disjoint set X
 - nodes with disjoint subtrees in the after-tree
 - example: leaves of the after-tree
 - we prove: $\Phi_{T'}(X) - \Phi_T(X) \leq O(1) \cdot \log n - |X|$
- a monotone set X
 - nodes with same left-(right) depth
 - we prove: $\Phi_{T'}(X) - \Phi_T(X) \leq O(1) \cdot \log n$
- a zig-zag set Z
 - set of alternating pairs in search path
 - we prove: $\Phi_{T'}(X) - \Phi_T(X) \leq O(1) \cdot \log n - |X|$

If conditions hold: we can partition P into $O(1)$
neighborhood-disjoint and monotone sets \implies access lemma.

Necessity of our conditions

- 1 accessed element goes to root (\checkmark for convenience),
- 2 the max right- or left-depth of the after-tree is $d = O(1)$,
- 3 the number of leaves in the after-tree is $\ell = \Omega(|P| - z)$.

Necessity of our conditions

- 1 accessed element goes to root (\checkmark for convenience),
- 2 the max right- or left-depth of the after-tree is $d = O(1)$,
- 3 the number of leaves in the after-tree is $\ell = \Omega(|P| - z)$.

(2) is equivalent with **locality**

Local = (search path \rightarrow after-tree) can be done bottom-up, with constant-sized buffer (i.e. as in splay or in Subramanian's algo).

Necessity of our conditions

- 1 accessed element goes to root (\checkmark for convenience),
- 2 the max right- or left-depth of the after-tree is $d = O(1)$,
- 3 the number of leaves in the after-tree is $\ell = \Omega(|P| - z)$.

(2) is equivalent with **locality**

Local = (search path \rightarrow after-tree) can be done bottom-up, with constant-sized buffer (i.e. as in splay or in Subramanian's algo).

Proof sketch: $d = \omega(1) \implies$ nonlocal: easy.

$d = O(1) \implies$ local: proof is an algorithm.

Necessity of our conditions

- 1 accessed element goes to root (\checkmark for convenience),
- 2 the max right- or left-depth of the after-tree is $d = O(1)$,
- 3 the number of leaves in the after-tree is $\ell = \Omega(|P| - z)$.

(2) is equivalent with **locality**

Local = (search path \rightarrow after-tree) can be done bottom-up, with constant-sized buffer (i.e. as in splay or in Subramanian's algo).

Proof sketch: $d = \omega(1) \implies$ nonlocal: easy.

$d = O(1) \implies$ local: proof is an algorithm.

Necessity of (2)

If self-adjusting BST algorithm \mathcal{A} satisfies the (weighted) access lemma by the sum-of-logs potential, then \mathcal{A} is local.

Necessity of our conditions

- 1 accessed element goes to root (\checkmark for convenience),
- 2 the max right- or left-depth of the after-tree is $d = O(1)$,
- 3 the number of leaves in the after-tree is $\ell = \Omega(|P| - z)$.

(2) is equivalent with **locality**

Local = (search path \rightarrow after-tree) can be done bottom-up, with constant-sized buffer (i.e. as in splay or in Subramanian's algo).

Proof sketch: $d = \omega(1) \implies$ nonlocal: easy.

$d = O(1) \implies$ local: proof is an algorithm.

Necessity of (2)

If self-adjusting BST algorithm \mathcal{A} satisfies the (weighted) access lemma by the sum-of-logs potential, then \mathcal{A} is local.

Proof idea: adversarial weight assignment.

Open questions

Open question 2: Are non-local algorithms bad or does it need a different potential function to prove that they are good?

Open questions

Open question 2: Are non-local algorithms bad or does it need a different potential function to prove that they are good?

Open question 3: Path balance, depth halving, and other natural heuristics are non-local. Are they good?

Open questions

Open question 2: Are non-local algorithms bad or does it need a different potential function to prove that they are good?

Open question 3: Path balance, depth halving, and other natural heuristics are non-local. Are they good?

Open question 4: Can these heuristics have logarithmic amortized cost (without access lemma in its full generality)?

Necessity of our conditions (continued)

- 1 accessed element goes to root,
- 2 the max right- or left-depth of the after-tree is $d = O(1)$,
- 3 the number of leaves in the after-tree is $\ell = \Omega(|P| - z)$.

Necessity of our conditions (continued)

- 1 accessed element goes to root,
- 2 the max right- or left-depth of the after-tree is $d = O(1)$,
- 3 the number of leaves in the after-tree is $\ell = \Omega(|P| - z)$.

Necessity of (3)

If algorithm \mathcal{A} satisfies (1) and (2), but only creates $n^{o(1)}$ leaves, then \mathcal{A} needs $n \cdot \omega(1)$ time to access $1, 2, \dots, n$.

Necessity of our conditions (continued)

- 1 accessed element goes to root,
- 2 the max right- or left-depth of the after-tree is $d = O(1)$,
- 3 the number of leaves in the after-tree is $\ell = \Omega(|P| - z)$.

Necessity of (3)

If algorithm \mathcal{A} satisfies (1) and (2), but only creates $n^{o(1)}$ leaves, then \mathcal{A} needs $n \cdot \omega(1)$ time to access $1, 2, \dots, n$.

Open question 5: Strengthen $n^{o(1)}$ to $o(n)$.

Necessity of our conditions (continued)

- 1 accessed element goes to root,
- 2 the max right- or left-depth of the after-tree is $d = O(1)$,
- 3 the number of leaves in the after-tree is $\ell = \Omega(|P| - z)$.

Necessity of (3)

If algorithm \mathcal{A} satisfies (1) and (2), but only creates $n^{o(1)}$ leaves, then \mathcal{A} needs $n \cdot \omega(1)$ time to access $1, 2, \dots, n$.

Open question 5: Strengthen $n^{o(1)}$ to $o(n)$.

Open question 6: Heuristic: create $\Omega(|P|)$ leaves at every step. Is this sufficient? This would also imply Path Balance conjecture. Try to disprove it!

Necessity of our conditions (continued)

- 1 accessed element goes to root,
- 2 the max right- or left-depth of the after-tree is $d = O(1)$,
- 3 the number of leaves in the after-tree is $\ell = \Omega(|P| - z)$.

Necessity of (3)

If algorithm \mathcal{A} satisfies (1) and (2), but only creates $n^{o(1)}$ leaves, then \mathcal{A} needs $n \cdot \omega(1)$ time to access $1, 2, \dots, n$.

Open question 5: Strengthen $n^{o(1)}$ to $o(n)$.

Open question 6: Heuristic: create $\Omega(|P|)$ leaves at every step. Is this sufficient? This would also imply Path Balance conjecture. Try to disprove it!

Open question 7: Which algorithms satisfy sequential access?

Dynamic optimality

Access sequence $X = (x_1, \dots, x_m) \in [n]^m$.

Dynamic optimality

Access sequence $X = (x_1, \dots, x_m) \in [n]^m$.

- $OPT_{static}(X)$ is the access cost by the best static BST.
(very well understood quantity)

Dynamic optimality

Access sequence $X = (x_1, \dots, x_m) \in [n]^m$.

- $OPT_{static}(X)$ is the access cost by the best static BST.
(very well understood quantity)
- $OPT(X)$ is the access cost by the best offline BST algorithm.
(very poorly understood)

Dynamic optimality

Access sequence $X = (x_1, \dots, x_m) \in [n]^m$.

- $OPT_{static}(X)$ is the access cost by the best static BST.
(very well understood quantity)
- $OPT(X)$ is the access cost by the best offline BST algorithm.
(very poorly understood)
- $OPT_{online}(X)$ is the access cost by the best online BST algorithm. (very poorly understood)

Dynamic optimality

Access sequence $X = (x_1, \dots, x_m) \in [n]^m$.

- $OPT_{static}(X)$ is the access cost by the best static BST.
(very well understood quantity)
- $OPT(X)$ is the access cost by the best offline BST algorithm.
(very poorly understood)
- $OPT_{online}(X)$ is the access cost by the best online BST algorithm. (very poorly understood)

Clearly: $cost_{splay}(X) \geq OPT_{online}(X) \geq OPT(X)$.

Dynamic optimality

Access sequence $X = (x_1, \dots, x_m) \in [n]^m$.

- $OPT_{static}(X)$ is the access cost by the best static BST.
(very well understood quantity)
- $OPT(X)$ is the access cost by the best offline BST algorithm.
(very poorly understood)
- $OPT_{online}(X)$ is the access cost by the best online BST algorithm.
(very poorly understood)

Clearly: $cost_{splay}(X) \geq OPT_{online}(X) \geq OPT(X)$.

OQ 8: Dynamic Optimality Conjecture (Sleator, Tarjan, 1983)

$cost_{splay}(X) \approx OPT_{online}(X) \approx OPT(X)$.

Dynamic optimality

Access sequence $X = (x_1, \dots, x_m) \in [n]^m$.

- $OPT_{static}(X)$ is the access cost by the best static BST.
(very well understood quantity)
- $OPT(X)$ is the access cost by the best offline BST algorithm.
(very poorly understood)
- $OPT_{online}(X)$ is the access cost by the best online BST algorithm.
(very poorly understood)

Clearly: $cost_{splay}(X) \geq OPT_{online}(X) \geq OPT(X)$.

OQ 8: Dynamic Optimality Conjecture (Sleator, Tarjan, 1983)

$cost_{splay}(X) \approx OPT_{online}(X) \approx OPT(X)$.

Only the trivial $\log n$ approximation factor is known. Best factor for any algorithm: $\log \log n$ (Tango trees, Multi-splay trees, etc.)

Dynamic optimality (continued)

Open Question 9: Can we compute or approximate $OPT(X)$ in poly-time?

Dynamic optimality (continued)

Open Question 9: Can we compute or approximate $OPT(X)$ in poly-time?

For most $X \in [m]^n$, we have $OPT(X) = \Omega(m \log n)$.

(Wilber, 1989)

Even a static (balanced) tree can match this.

Dynamic optimality (continued)

Open Question 9: Can we compute or approximate $OPT(X)$ in poly-time?

For most $X \in [m]^n$, we have $OPT(X) = \Omega(m \log n)$.

(Wilber, 1989)

Even a static (balanced) tree can match this.

\implies dynamic optimality is interesting only for “easy” sequences.

Dynamic optimality (continued)

Open Question 9: Can we compute or approximate $OPT(X)$ in poly-time?

For most $X \in [m]^n$, we have $OPT(X) = \Omega(m \log n)$.

(Wilber, 1989)

Even a static (balanced) tree can match this.

\implies dynamic optimality is interesting only for “easy” sequences.

Open question 10: For which X is $OPT(X) = O(m)$?

Dynamic optimality (continued)

Open Question 9: Can we compute or approximate $OPT(X)$ in poly-time?

For most $X \in [m]^n$, we have $OPT(X) = \Omega(m \log n)$.

(Wilber, 1989)

Even a static (balanced) tree can match this.

\implies dynamic optimality is interesting only for “easy” sequences.

Open question 10: For which X is $OPT(X) = O(m)$?

Many known results of this type: $\mathcal{X} \subseteq [n]^m$ is easy, for $X \in \mathcal{X}$ algorithm \mathcal{A} matches optimum (asymptotically, in amortized sense).

Dynamic optimality (continued)

Open Question 9: Can we compute or approximate $OPT(X)$ in poly-time?

For most $X \in [m]^n$, we have $OPT(X) = \Omega(m \log n)$.

(Wilber, 1989)

Even a static (balanced) tree can match this.

\implies dynamic optimality is interesting only for “easy” sequences.

Open question 10: For which X is $OPT(X) = O(m)$?

Many known results of this type: $\mathcal{X} \subseteq [n]^m$ is easy, for $X \in \mathcal{X}$ algorithm \mathcal{A} matches optimum (asymptotically, in amortized sense).

Examples: low entropy (static optimality), spatial/temporal clustering (dynamic finger, working set), sequential access, etc.

Dynamic optimality (continued)

Open Question 9: Can we compute or approximate $OPT(X)$ in poly-time?

For most $X \in [m]^n$, we have $OPT(X) = \Omega(m \log n)$.

(Wilber, 1989)

Even a static (balanced) tree can match this.

\implies dynamic optimality is interesting only for “easy” sequences.

Open question 10: For which X is $OPT(X) = O(m)$?

Many known results of this type: $\mathcal{X} \subseteq [n]^m$ is easy, for $X \in \mathcal{X}$ algorithm \mathcal{A} matches optimum (asymptotically, in amortized sense).

Examples: low entropy (static optimality), spatial/temporal clustering (dynamic finger, working set), sequential access, etc.