Efficient top-down updates in AVL trees

Vincent Jugé

Université Gustave Eiffel

08/05/2025

Contents

Maintaining ordered sets

Which data structure should I use to manipulate ordered sets?

Maintaining ordered sets

Which data structure should I use to manipulate ordered sets?

Query #1: Does key k belong to my set S?

Answer: Use a sorted array + dichotomy.

Maintaining ordered sets

Which data structure should I use to manipulate ordered sets?

Query #1: Does key k belong to my set S?

Answer: Use a sorted array + dichotomy.

$3 \in S$?

1	2	4	7	8	10	19

Maintaining ordered sets

Which data structure should I use to manipulate ordered sets?

Query #1: Does key k belong to my set S?

Answer: Use a sorted array + dichotomy.

$3 \in S?$

Maintaining ordered sets

Which data structure should I use to manipulate ordered sets?

Query #1: Does key k belong to my set S?

Answer: Use a sorted array + dichotomy.

$3 \in S$?

1	2	4	7	8	10	19

Maintaining ordered sets

Which data structure should I use to manipulate ordered sets?

Query #1: Does key k belong to my set S?

Answer: Use a sorted array + dichotomy.

$3 \in S$? **No!**

1	2	4	7	8	10	19

Maintaining ordered sets

Which data structure should I use to manipulate ordered sets?

Query #1: Does key k belong to my set S?
Query #2: Insert key k in my set S.

Answer: Use a sorted array + dichotomy.

Add 3!

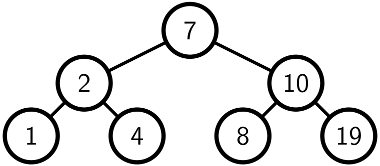
1	2	3	4	7	8	10	19

Maintaining ordered sets

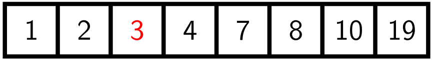
Which data structure should I use to manipulate ordered sets?

Query #1: Does key k belong to my set S?
Query #2: Insert key k in my set S.

Answer: Use a sorted array + dichotomy. **Oops!**
Use a low-height binary search tree instead.



Add 3!

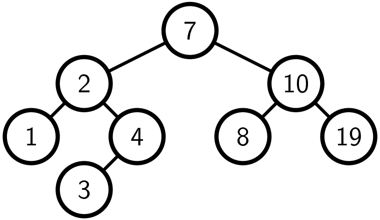


Maintaining ordered sets

Which data structure should I use to manipulate ordered sets?

Query #1: Does key k belong to my set S?
Query #2: Insert key k in my set S.

Answer: Use a sorted array + dichotomy. **Oops!**
Use a low-height binary search tree instead.



Add 3!

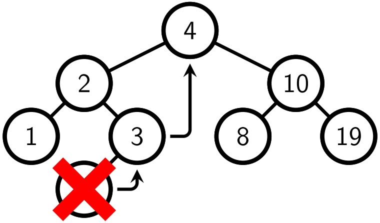
1	2	3	4	7	8	10	19

Maintaining ordered sets

Which data structure should I use to manipulate ordered sets?

Query #1: Does key k belong to my set S?
Query #2: Insert key k in my set S.
Query #3: Delete key k from my set S.

Answer: Use a sorted array + dichotomy. **Oops!**
Use a low-height binary search tree instead.



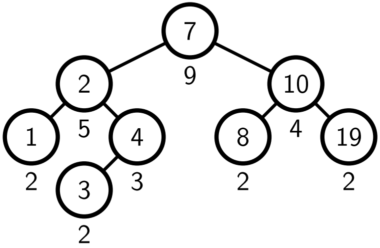
Remove 7!

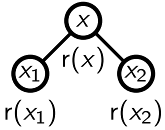
1	2	3	4	8	10	19

Balanced binary search trees

Maintaining search trees of height $\mathcal{O}(\log(n))$ often requires some kind of **rank** and **balance**.

1. Weight-bounded trees: $\mathrm{r}(x) = \#\mathcal{T}(x) + 1$ $\quad(\mathrm{r}(x) = \mathrm{r}(x_1) + \mathrm{r}(x_2)$ and $\mathrm{r}(\perp) = 1)$
   $\mathrm{r}(x_i) \leqslant \alpha \mathrm{r}(x)$ $\quad(1/\sqrt{2} \leqslant \alpha < 9/11)$

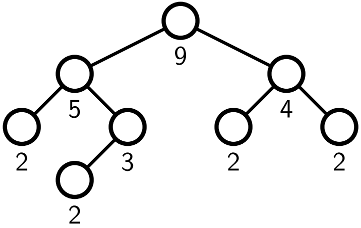


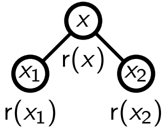


Balanced binary search trees

Maintaining search trees of height $\mathcal{O}(\log(n))$ often requires some kind of **rank** and **balance**.

1. Weight-bounded trees: $\mathsf{r}(x) = \#\mathcal{T}(x) + 1$ $\quad$ $(\mathsf{r}(x) = \mathsf{r}(x_1) + \mathsf{r}(x_2)$ and $\mathsf{r}(\perp) = 1)$
   $\mathsf{r}(x_i) \leqslant \alpha \mathsf{r}(x)$ $\quad$ $(1/\sqrt{2} \leqslant \alpha < 9/11)$

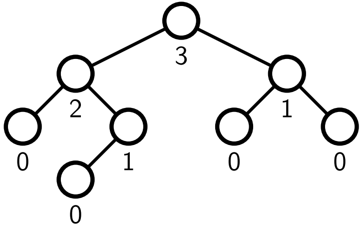


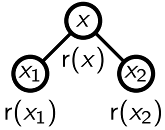


Balanced binary search trees

Maintaining search trees of height $\mathcal{O}(\log(n))$ often requires some kind of **rank** and **balance**.

1. Weight-bounded trees: $r(x) = \#\mathcal{T}(x) + 1$ $\quad (r(x) = r(x_1) + r(x_2)$ and $r(\perp) = 1)$
   $r(x_i) \leqslant \alpha r(x)$ $\quad (1/\sqrt{2} \leqslant \alpha < 9/11)$
2. AVL trees: $r(x) = h(x)$ $\quad (r(x) = \max\{r(x_1), r(x_2)\} + 1$ and $r(\perp) = -1)$
   $r(x) \leqslant r(x_i) + 2$

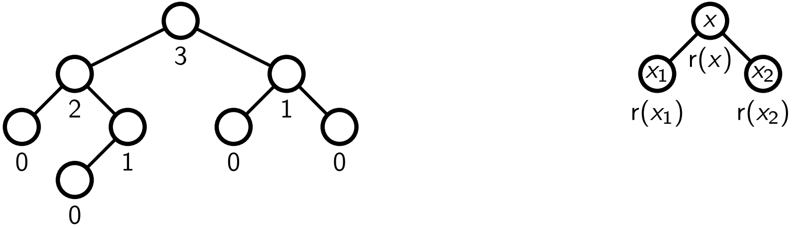




Balanced binary search trees

Maintaining search trees of height $\mathcal{O}(\log(n))$ often requires some kind of **rank** and **balance**.

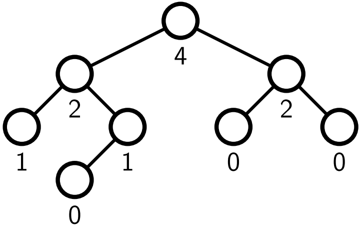
1. Weight-bounded trees: $\mathrm{r}(x) = \#\mathcal{T}(x) + 1$ $\quad (\mathrm{r}(x) = \mathrm{r}(x_1) + \mathrm{r}(x_2)$ and $\mathrm{r}(\bot) = 1)$
   $\mathrm{r}(x_i) \leqslant \alpha \mathrm{r}(x)$ $\quad (1/\sqrt{2} \leqslant \alpha < 9/11)$
2. AVL trees: $\mathrm{r}(x) = \mathrm{h}(x)$ $\quad (\mathrm{r}(x) = \max\{\mathrm{r}(x_1), \mathrm{r}(x_2)\} + 1$ and $\mathrm{r}(\bot) = -1)$
   $\mathrm{r}(x) \leqslant \mathrm{r}(x_i) + 2$
3. Weak AVL trees: $\mathrm{r}(x) = ?$
   $\mathrm{r}(x_i) + 1 \leqslant \mathrm{r}(x) \leqslant \mathrm{r}(x_i) + 2$ and $\mathrm{r}(\bot) = -1$

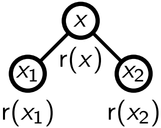


Balanced binary search trees

Maintaining search trees of height $\mathcal{O}(\log(n))$ often requires some kind of **rank** and **balance**.

1. Weight-bounded trees: $\mathrm{r}(x) = \#\mathcal{T}(x) + 1$ $\quad$ $(\mathrm{r}(x) = \mathrm{r}(x_1) + \mathrm{r}(x_2)$ and $\mathrm{r}(\perp) = 1)$
   $\mathrm{r}(x_i) \leqslant \alpha \mathrm{r}(x)$ $\quad$ $(1/\sqrt{2} \leqslant \alpha < 9/11)$
2. AVL trees: $\mathrm{r}(x) = \mathrm{h}(x)$ $\quad$ $(\mathrm{r}(x) = \max\{\mathrm{r}(x_1), \mathrm{r}(x_2)\} + 1$ and $\mathrm{r}(\perp) = -1)$
   $\mathrm{r}(x) \leqslant \mathrm{r}(x_i) + 2$
3. Weak AVL trees: $\mathrm{r}(x) = ?$
   $\mathrm{r}(x_i) + 1 \leqslant \mathrm{r}(x) \leqslant \mathrm{r}(x_i) + 2$ and $\mathrm{r}(\perp) = -1$

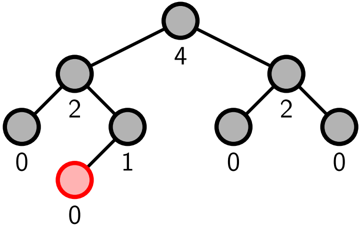


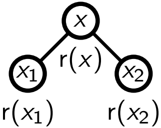


Balanced binary search trees

Maintaining search trees of height $\mathcal{O}(\log(n))$ often requires some kind of **rank** and **balance**.

1. Weight-bounded trees: $\mathrm{r}(x) = \#\mathcal{T}(x) + 1$ $\quad (\mathrm{r}(x) = \mathrm{r}(x_1) + \mathrm{r}(x_2)$ and $\mathrm{r}(\perp) = 1)$
   $\mathrm{r}(x_i) \leqslant \alpha \mathrm{r}(x)$ $\quad (1/\sqrt{2} \leqslant \alpha < 9/11)$
2. AVL trees: $\mathrm{r}(x) = \mathrm{h}(x)$ $\quad (\mathrm{r}(x) = \max\{\mathrm{r}(x_1), \mathrm{r}(x_2)\} + 1$ and $\mathrm{r}(\perp) = -1)$
   $\mathrm{r}(x) \leqslant \mathrm{r}(x_i) + 2$
3. Weak AVL trees: $\mathrm{r}(x) = \,?$
   $\mathrm{r}(x_i) + 1 \leqslant \mathrm{r}(x) \leqslant \mathrm{r}(x_i) + 2$ and $\mathrm{r}(\perp) = -1$
4. Red-black trees: *idem* $\quad (x_i$ red $\Leftrightarrow \lfloor \mathrm{r}(x_i)/2 \rfloor = \lfloor \mathrm{r}(x)/2 \rfloor)$

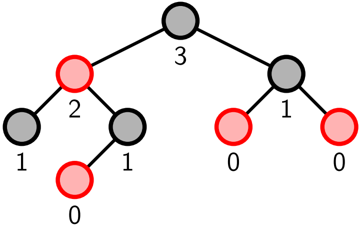


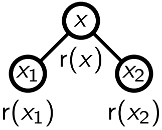


Balanced binary search trees

Maintaining search trees of height $\mathcal{O}(\log(n))$ often requires some kind of **rank** and **balance**.

1. Weight-bounded trees: $r(x) = \#\mathcal{T}(x) + 1$ $\quad (r(x) = r(x_1) + r(x_2)$ and $r(\bot) = 1)$
   $r(x_i) \leqslant \alpha r(x)$ $\quad (1/\sqrt{2} \leqslant \alpha < 9/11)$
2. AVL trees: $r(x) = h(x)$ $\quad (r(x) = \max\{r(x_1), r(x_2)\} + 1$ and $r(\bot) = -1)$
   $r(x) \leqslant r(x_i) + 2$
3. Weak AVL trees: $r(x) = ?$
   $r(x_i) + 1 \leqslant r(x) \leqslant r(x_i) + 2$ and $r(\bot) = -1$
4. Red-black trees: *idem* $\quad (x_i$ red $\Leftrightarrow \lfloor r(x_i)/2 \rfloor = \lfloor r(x)/2 \rfloor)$





Balanced binary search trees

Maintaining search trees of height $\mathcal{O}(\log(n))$ often requires some kind of **rank** and **balance**.

1. Weight-bounded trees: $\mathrm{r}(x) = \#\mathcal{T}(x) + 1$ $\quad(\mathrm{r}(x) = \mathrm{r}(x_1) + \mathrm{r}(x_2)$ and $\mathrm{r}(\bot) = 1)$
   $\mathrm{r}(x_i) \leqslant \alpha \mathrm{r}(x)$ $\quad(1/\sqrt{2} \leqslant \alpha < 9/11)$
2. AVL trees: $\mathrm{r}(x) = \mathrm{h}(x)$ $\quad(\mathrm{r}(x) = \max\{\mathrm{r}(x_1), \mathrm{r}(x_2)\} + 1$ and $\mathrm{r}(\bot) = -1)$
   $\mathrm{r}(x) \leqslant \mathrm{r}(x_i) + 2$
3. Weak AVL trees: $\mathrm{r}(x) = ?$
   $\mathrm{r}(x_i) + 1 \leqslant \mathrm{r}(x) \leqslant \mathrm{r}(x_i) + 2$ and $\mathrm{r}(\bot) = -1$
4. Red-black trees: *idem* $\quad(x_i$ red $\Leftrightarrow \lfloor \mathrm{r}(x_i)/2 \rfloor = \lfloor \mathrm{r}(x)/2 \rfloor)$

Tree	Invented	Height	Am. writes/update	Top-down
Weight-balanced	1972	$2\log_2(n)$	$\Theta(1)$	yes
AVL	1962	$1.44\log_2(n)$	$\Theta(\log(n))$	no
Weak AVL	2015	$2\log_2(n)$	$\Theta(1)$	yes
Red-black	1978	$2\log_2(n)$	$\Theta(1)$	yes

Amortised write operations and top-down updates

Write operations are quite more expensive than **read** operations.

Amortised write operations and top-down updates

Write operations are quite more expensive than **read** operations.

Amortised complexity: Worst-case **average** complexity over **arbitrary** sequences of operations

Amortised write operations and top-down updates

Write operations are quite more expensive than **read** operations.

Amortised complexity: Worst-case **average** complexity over **arbitrary** sequences of operations

Example:
In weak AVL trees, the first q queries trigger at most $13q$ rank updates and/or pointer rewrites:
Their amortised write complexity is 13 operations per update.

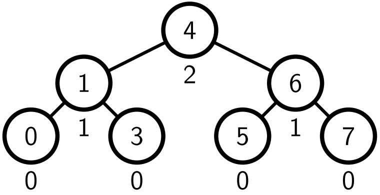
Amortised write operations and top-down updates

Write operations are quite more expensive than **read** operations.

Amortised complexity: Worst-case **average** complexity over **arbitrary** sequences of operations

Example:
In weak AVL trees, the first q queries trigger at most $13q$ rank updates and/or pointer rewrites: Their amortised write complexity is 13 operations per update.

Insert 2, then 8, **bottom-up**



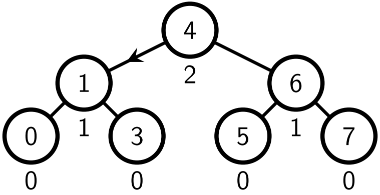
Amortised write operations and top-down updates

Write operations are quite more expensive than **read** operations.

Amortised complexity: Worst-case **average** complexity over **arbitrary** sequences of operations

Example:
In weak AVL trees, the first q queries trigger at most $13q$ rank updates and/or pointer rewrites: Their amortised write complexity is 13 operations per update.

Insert 2, then 8, **bottom-up**



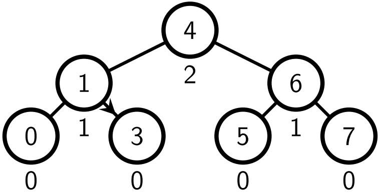
Amortised write operations and top-down updates

Write operations are quite more expensive than **read** operations.

Amortised complexity: Worst-case **average** complexity over **arbitrary** sequences of operations

Example:
In weak AVL trees, the first q queries trigger at most $13q$ rank updates and/or pointer rewrites:
Their amortised write complexity is 13 operations per update.

Insert 2, then 8, **bottom-up**



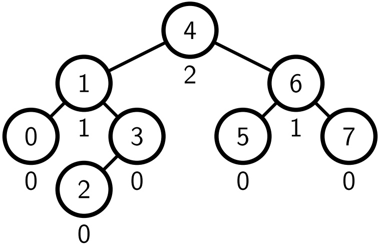
Amortised write operations and top-down updates

Write operations are quite more expensive than **read** operations.

Amortised complexity: Worst-case **average** complexity over **arbitrary** sequences of operations

Example:
In weak AVL trees, the first q queries trigger at most $13q$ rank updates and/or pointer rewrites:
Their amortised write complexity is 13 operations per update.

Insert 2, then 8, **bottom-up**



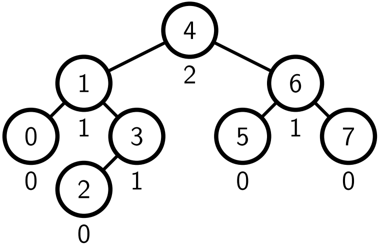
Amortised write operations and top-down updates

Write operations are quite more expensive than **read** operations.

Amortised complexity: Worst-case **average** complexity over **arbitrary** sequences of operations

Example:
In weak AVL trees, the first q queries trigger at most $13q$ rank updates and/or pointer rewrites: Their amortised write complexity is 13 operations per update.

Insert 2, then 8, **bottom-up**



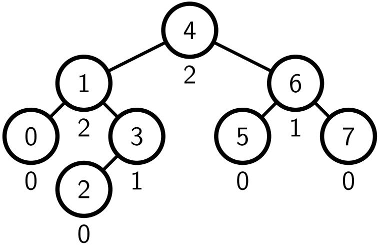
Amortised write operations and top-down updates

Write operations are quite more expensive than **read** operations.

Amortised complexity: Worst-case **average** complexity over **arbitrary** sequences of operations

Example:
In weak AVL trees, the first q queries trigger at most $13q$ rank updates and/or pointer rewrites: Their amortised write complexity is 13 operations per update.

Insert 2, then 8, **bottom-up**



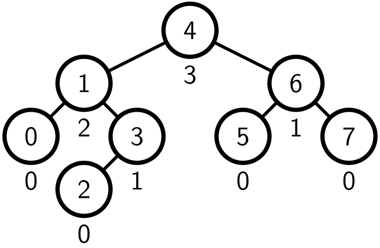
Amortised write operations and top-down updates

Write operations are quite more expensive than **read** operations.

Amortised complexity: Worst-case **average** complexity over **arbitrary** sequences of operations

Example:
In weak AVL trees, the first q queries trigger at most $13q$ rank updates and/or pointer rewrites:
Their amortised write complexity is 13 operations per update.

Insert 2, then 8, **bottom-up**



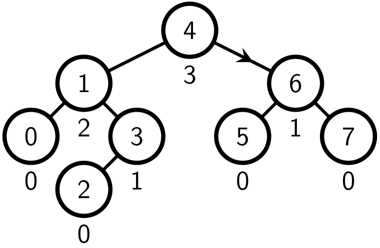
Amortised write operations and top-down updates

Write operations are quite more expensive than **read** operations.

Amortised complexity: Worst-case **average** complexity over **arbitrary** sequences of operations

Example:
In weak AVL trees, the first q queries trigger at most $13q$ rank updates and/or pointer rewrites: Their amortised write complexity is 13 operations per update.

Insert 2, then 8, **bottom-up**



Amortised write operations and top-down updates

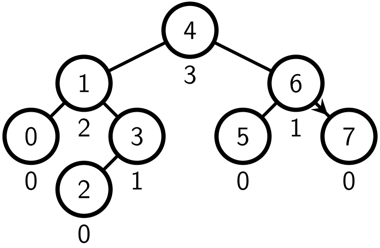
Write operations are quite more expensive than **read** operations.

Amortised complexity: Worst-case **average** complexity over **arbitrary** sequences of operations

Example:

In weak AVL trees, the first q queries trigger at most $13q$ rank updates and/or pointer rewrites: Their amortised write complexity is 13 operations per update.

Insert 2, then 8, **bottom-up**



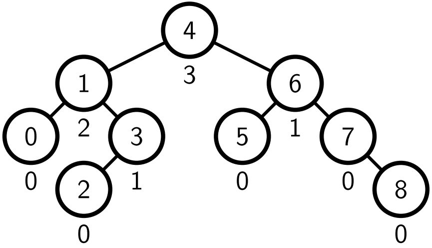
Amortised write operations and top-down updates

Write operations are quite more expensive than **read** operations.

Amortised complexity: Worst-case **average** complexity over **arbitrary** sequences of operations

Example:
In weak AVL trees, the first q queries trigger at most $13q$ rank updates and/or pointer rewrites:
Their amortised write complexity is 13 operations per update.

Insert 2, then 8, **bottom-up**



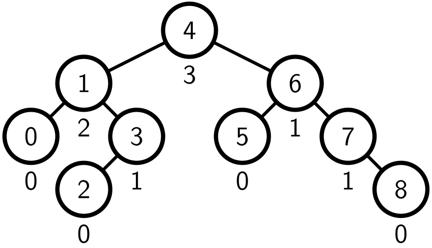
Amortised write operations and top-down updates

Write operations are quite more expensive than **read** operations.

Amortised complexity: Worst-case **average** complexity over **arbitrary** sequences of operations

Example:
In weak AVL trees, the first q queries trigger at most $13q$ rank updates and/or pointer rewrites: Their amortised write complexity is 13 operations per update.

Insert 2, then 8, **bottom-up**



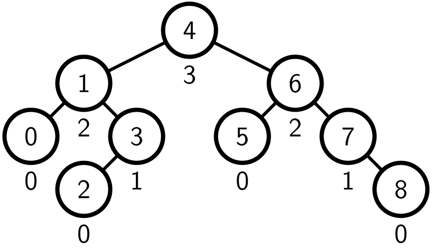
Amortised write operations and top-down updates

Write operations are quite more expensive than **read** operations.

Amortised complexity: Worst-case **average** complexity over **arbitrary** sequences of operations

Example:
In weak AVL trees, the first q queries trigger at most $13q$ rank updates and/or pointer rewrites:
Their amortised write complexity is 13 operations per update.

Insert 2, then 8, **bottom-up**



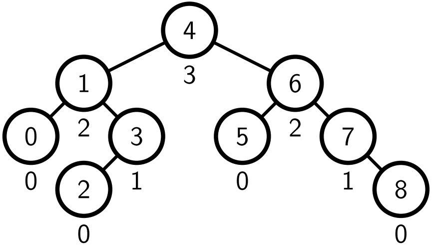
Amortised write operations and top-down updates

Write operations are quite more expensive than **read** operations.

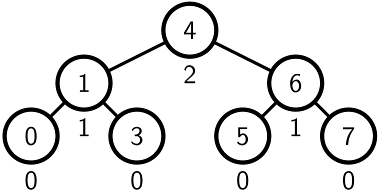
Amortised complexity: Worst-case **average** complexity over **arbitrary** sequences of operations

Example:
In weak AVL trees, the first q queries trigger at most $13q$ rank updates and/or pointer rewrites: Their amortised write complexity is 13 operations per update.

Insert 2, then 8, **bottom-up**



Insert 2, then 8, **top-down**



Amortised write operations and top-down updates

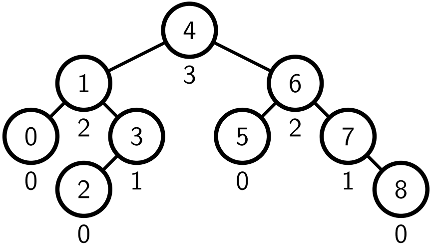
Write operations are quite more expensive than **read** operations.

Amortised complexity: Worst-case **average** complexity over **arbitrary** sequences of operations

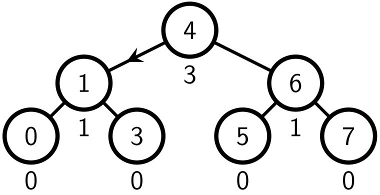
Example:

In weak AVL trees, the first q queries trigger at most $13q$ rank updates and/or pointer rewrites: Their amortised write complexity is 13 operations per update.

Insert 2, then 8, **bottom-up**



Insert 2, then 8, **top-down**



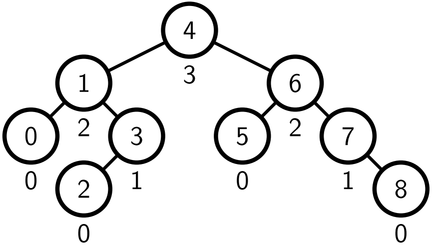
Amortised write operations and top-down updates

Write operations are quite more expensive than **read** operations.

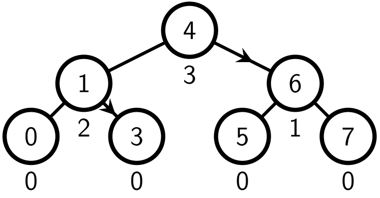
Amortised complexity: Worst-case **average** complexity over **arbitrary** sequences of operations

Example:
In weak AVL trees, the first q queries trigger at most $13q$ rank updates and/or pointer rewrites: Their amortised write complexity is 13 operations per update.

Insert 2, then 8, **bottom-up**



Insert 2, then 8, **top-down**



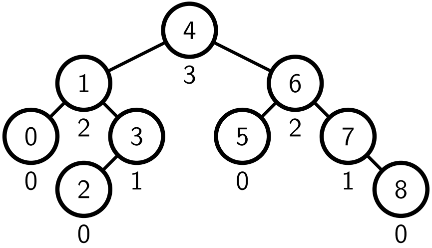
Amortised write operations and top-down updates

Write operations are quite more expensive than **read** operations.

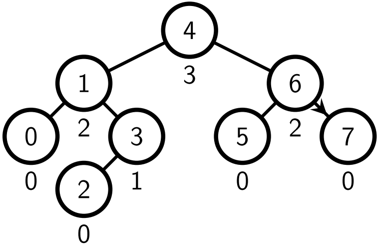
Amortised complexity: Worst-case **average** complexity over **arbitrary** sequences of operations

Example:
In weak AVL trees, the first q queries trigger at most $13q$ rank updates and/or pointer rewrites:
Their amortised write complexity is 13 operations per update.

Insert 2, then 8, **bottom-up**



Insert 2, then 8, **top-down**



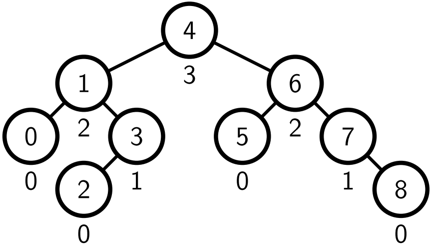
Amortised write operations and top-down updates

Write operations are quite more expensive than **read** operations.

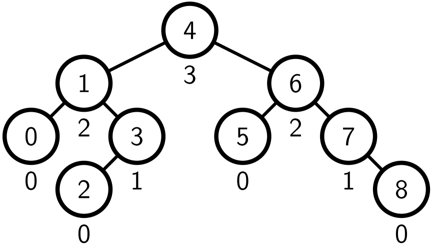
Amortised complexity: Worst-case **average** complexity over **arbitrary** sequences of operations

Example:
In weak AVL trees, the first q queries trigger at most $13q$ rank updates and/or pointer rewrites:
Their amortised write complexity is 13 operations per update.

Insert 2, then 8, **bottom-up**

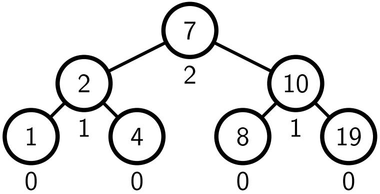


Insert 2, then 8, **top-down**



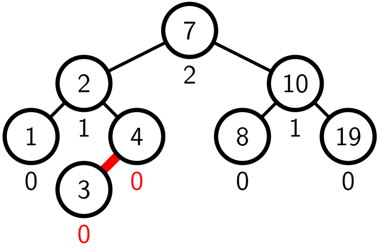
Contents

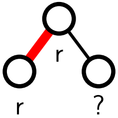
Updating AVL trees: Eliminating anomalies in node ranks



Updating AVL trees: Eliminating anomalies in node ranks

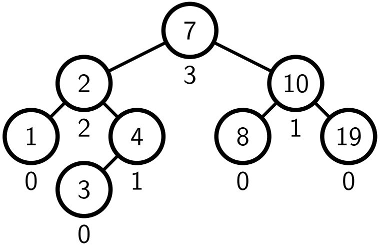
Goals: Avoiding **zero-edges**.

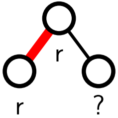




Updating AVL trees: Eliminating anomalies in node ranks

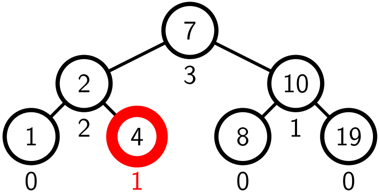
Goals: Avoiding zero-edges.

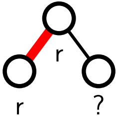




Updating AVL trees: Eliminating anomalies in node ranks

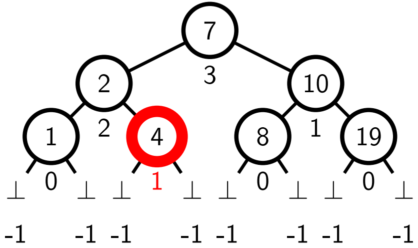
Goals: Avoiding **zero-edges**.

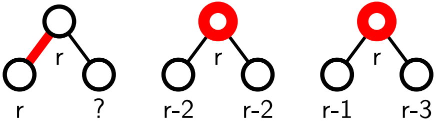




Updating AVL trees: Eliminating anomalies in node ranks

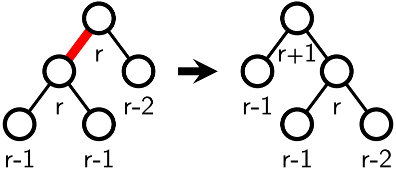
Goals: Avoiding **zero-edges** and **four-nodes**.

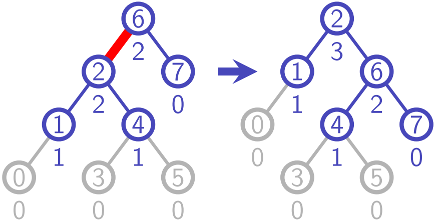




Updating AVL trees: Eliminating anomalies in node ranks

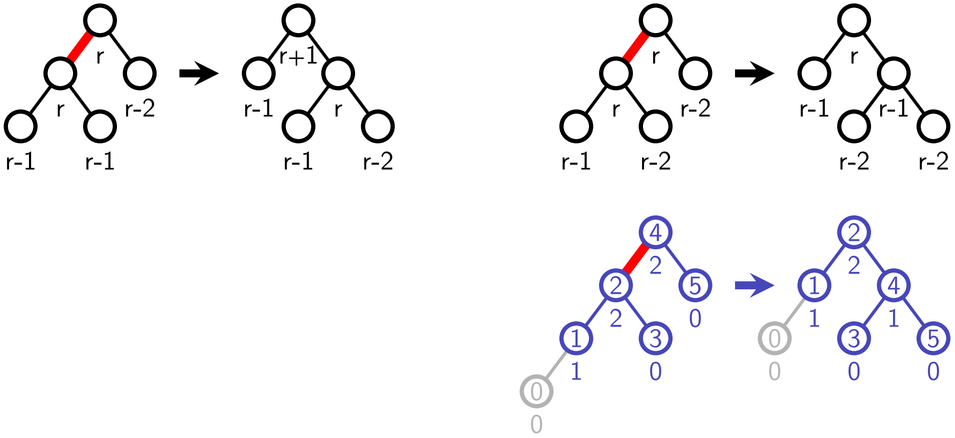
Goals: Avoiding zero-edges and **four-nodes**.





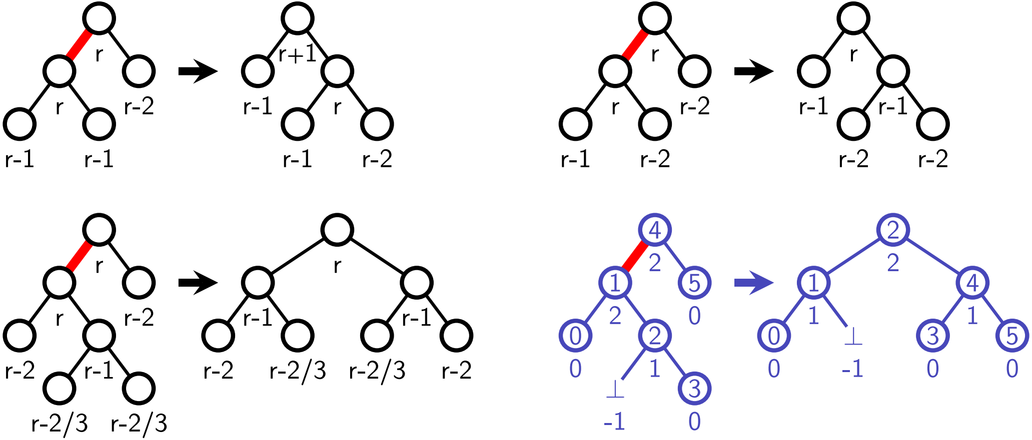
Updating AVL trees: Eliminating anomalies in node ranks

Goals: Avoiding **zero-edges** and **four-nodes**.



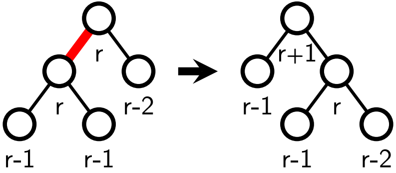
Updating AVL trees: Eliminating anomalies in node ranks

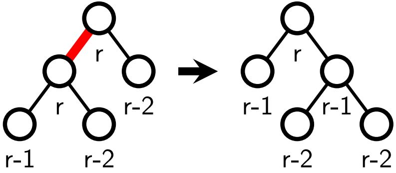
Goals: Avoiding zero-edges and **four-nodes**.

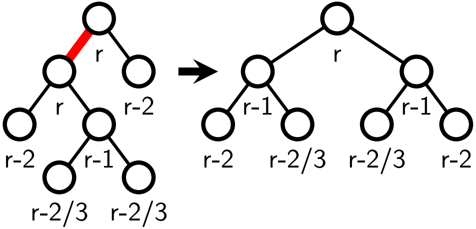


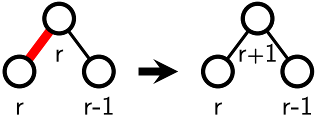
Updating AVL trees: Eliminating anomalies in node ranks

Goals: Avoiding zero-edges and **four-nodes**.



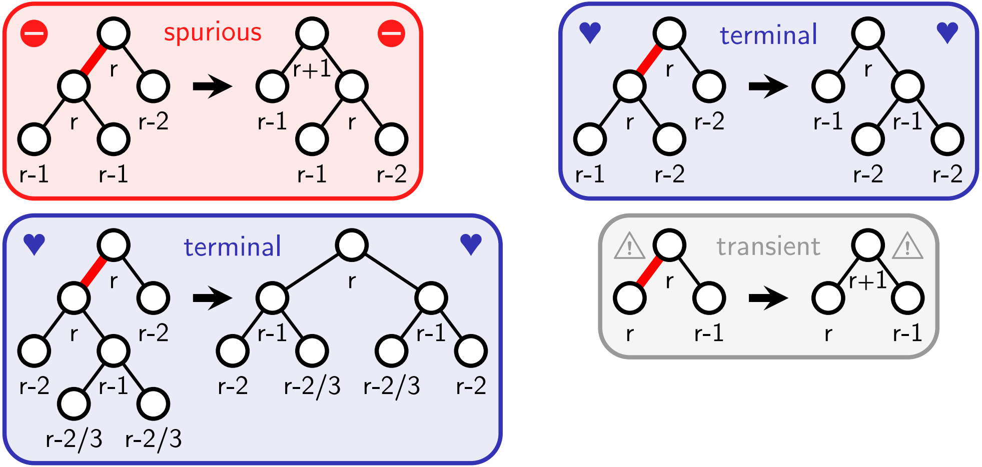






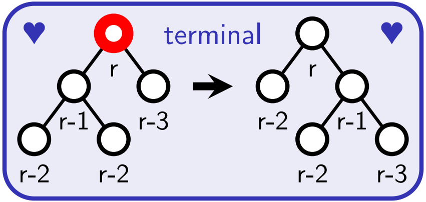
Updating AVL trees: Eliminating anomalies in node ranks

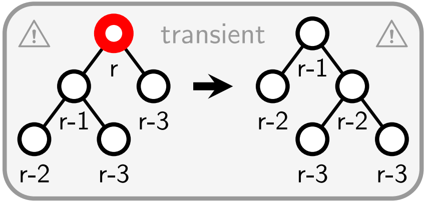
Goals: Avoiding **zero-edges** and **four-nodes**.

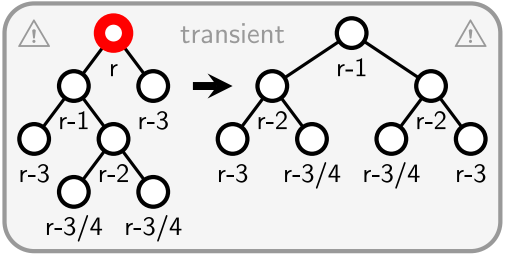


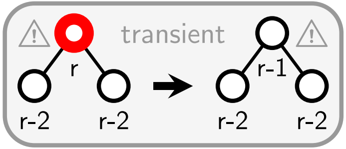
Updating AVL trees: Eliminating anomalies in node ranks

Goals: Avoiding **zero-edges** and **four-nodes**.

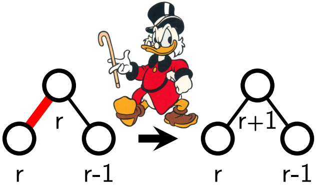


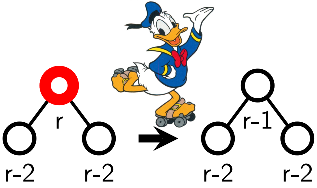




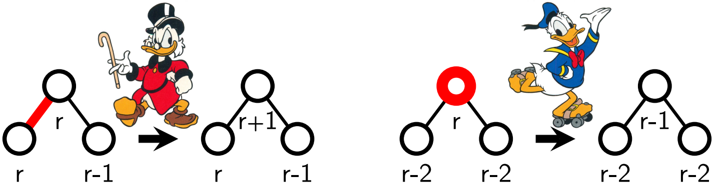


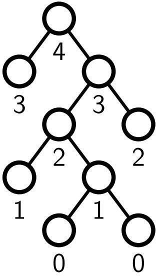
Updating AVL trees: The evil pair



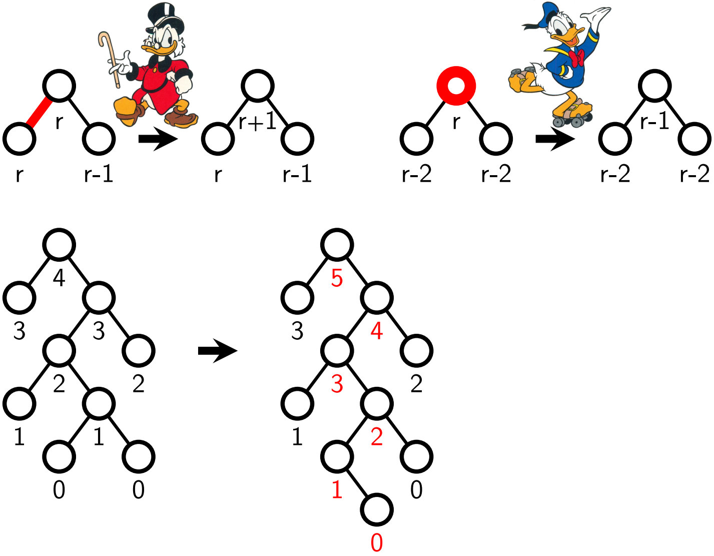


Updating AVL trees: The evil pair

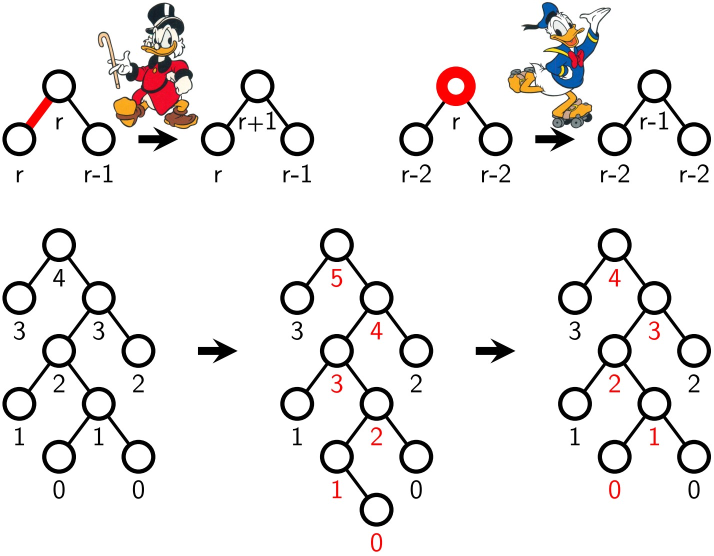




Updating AVL trees: The evil pair

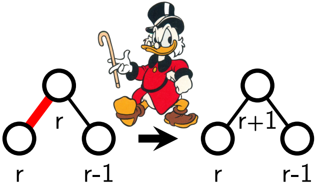


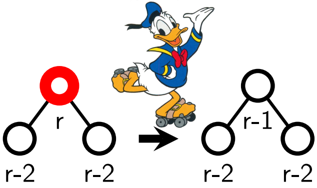
Updating AVL trees: The evil pair

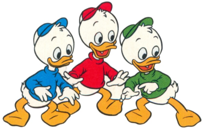


Updating AVL trees: The evil pair. . . and how to defeat it!

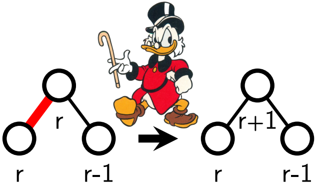
Updating AVL trees: The evil pair. . . and how to defeat it!

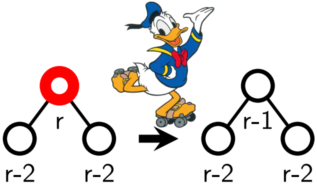


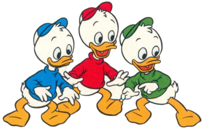


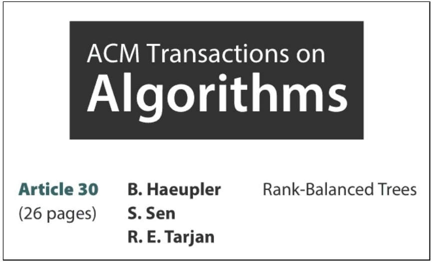


Updating AVL trees: The evil pair. . . and how to defeat it!








ACM Transactions on
Algorithms

Article 30 (26 pages) — B. Haeupler, S. Sen, R. E. Tarjan — Rank-Balanced Trees

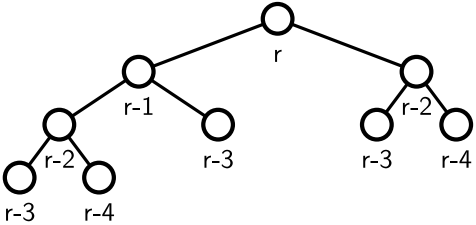
Updating AVL trees: Stopping anomalies faster

Stop propagating **0-edges** and **4-nodes** faster by **demoting** and **promoting** nodes.

Updating AVL trees: Stopping anomalies faster

Stop propagating **0-edges** and **4-nodes** faster by **demoting** and **promoting** nodes.

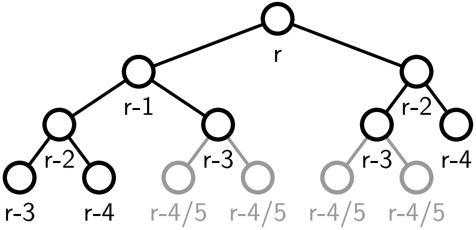
- **Hollow** nodes can be **demoted**. (descendants at rank $\geqslant r-2$ have a 2-child)



Updating AVL trees: Stopping anomalies faster

Stop propagating **0-edges** and **4-nodes** faster by **demoting** and **promoting** nodes.

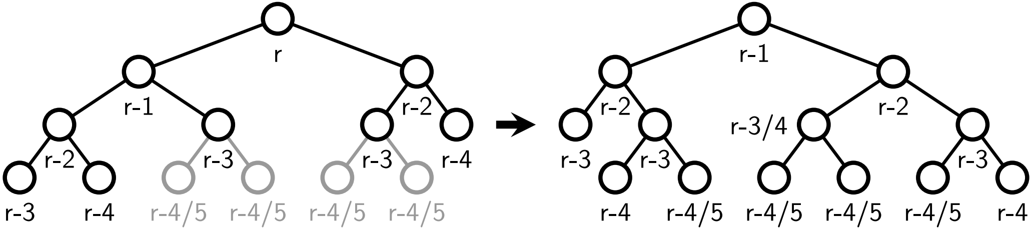
- **Hollow** nodes can be **demoted**. (descendants at rank $\geqslant r-2$ have a 2-child)



Updating AVL trees: Stopping anomalies faster

Stop propagating **0-edges** and **4-nodes** faster by **demoting** and **promoting** nodes.

- **Hollow** nodes can be **demoted**. (descendants at rank $\geqslant r-2$ have a 2-child)

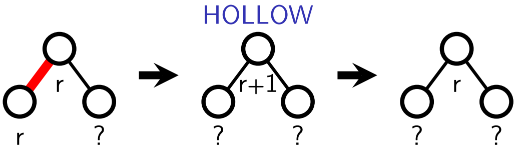


Updating AVL trees: Stopping anomalies faster

Stop propagating **0-edges** and **4-nodes** faster by **demoting** and **promoting** nodes.

- **Hollow** nodes can be **demoted**: (descendants at rank $\geqslant r-2$ have a 2-child)

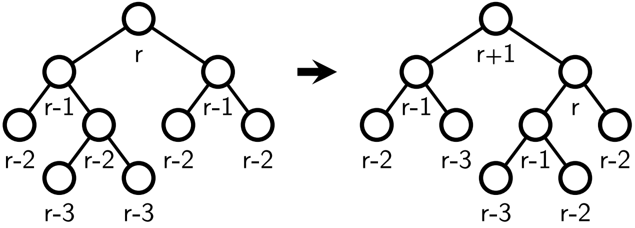
  when propagating a 0-edge creates an empty node, just demote it!



Updating AVL trees: Stopping anomalies faster

Stop propagating **0-edges** and **4-nodes** faster by **demoting** and **promoting** nodes.

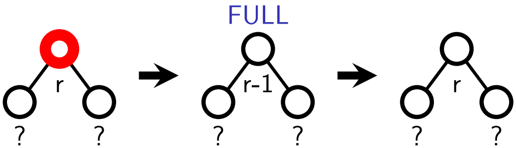
- **Hollow** nodes can be **demoted**: (descendants at rank $\geqslant r - 2$ have a 2-child)
  when propagating a 0-edge creates an empty node, just demote it!
- **Full** nodes can be **promoted**. (node, children & central grand-child have 1-children only)



Updating AVL trees: Stopping anomalies faster

Stop propagating **0-edges** and **4-nodes** faster by **demoting** and **promoting** nodes.

- **Hollow** nodes can be **demoted**: (descendants at rank $\geqslant r - 2$ have a 2-child)
  when propagating a 0-edge creates an empty node, just demote it!
- **Full** nodes can be **promoted**: (node, children & central grand-child have 1-children only)
  when propagating a 4-node creates a full node, just promote it!



Updating AVL trees efficiently

Using our fast stopping procedure,

Transient **deletion** operations create no full nodes and destroy one 2-child.
insertion operations destroy one full node and create one 2-child.

Updating AVL trees efficiently

Using our fast stopping procedure,

Transient **deletion** operations create no full nodes and destroy one 2-child.
insertion operations destroy one full node and create one 2-child.

Theorem

Starting from the empty AVL tree, q queries trigger $\mathcal{O}(q)$ write operations.

Updating AVL trees efficiently

Using our fast stopping procedure,

Transient **deletion** operations create no full nodes and destroy one 2-child.
insertion operations destroy one full node and create one 2-child.

Theorem

Starting from the empty AVL tree, q queries trigger $\mathcal{O}(q)$ write operations.

Proof:

Tree potential: $2 \times \#(\text{full nodes}) + \#(\text{2-children})$

decreases with each transient operation!

Updating AVL trees efficiently

Using our fast stopping procedure,

Transient **deletion** operations create no full nodes and destroy one 2-child.
insertion operations destroy one full node and create one 2-child.

Theorem

Starting from the empty AVL tree, q queries trigger $\mathcal{O}(q)$ write operations.

Proof:

Tree potential: $2 \times \#(\text{full nodes}) + \#(\text{2-children})$
decreases with each transient operation!

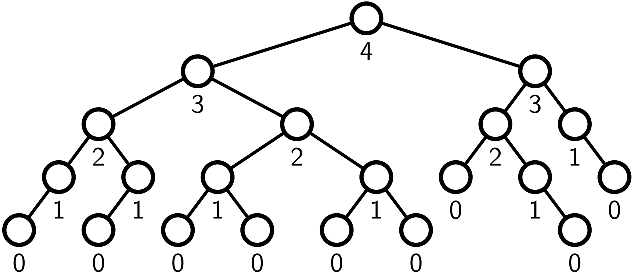
With q queries + $\mathbf{T}$ transient operations:

- tree potential increases by $\mathcal{O}(q) - \mathbf{T}$ or less, and
- tree potential remains non-negative, hence
- $\mathbf{T} = \mathcal{O}(q)$.

Contents

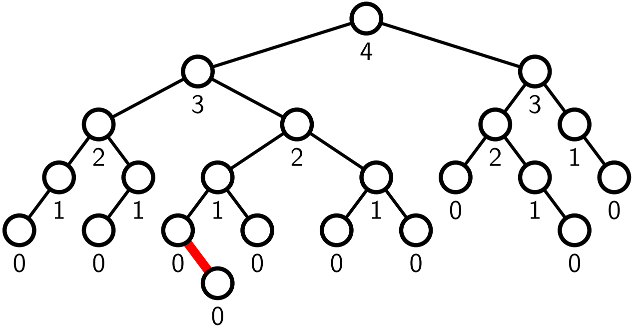
Safe nodes: When do we stop anomalies?

A node is insertion-**safe** if its stops propagating **zero-edges**.



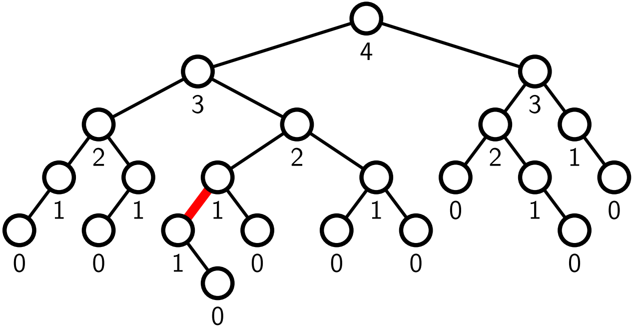
Safe nodes: When do we stop anomalies?

A node is insertion-**safe** if its stops propagating **zero-edges**.



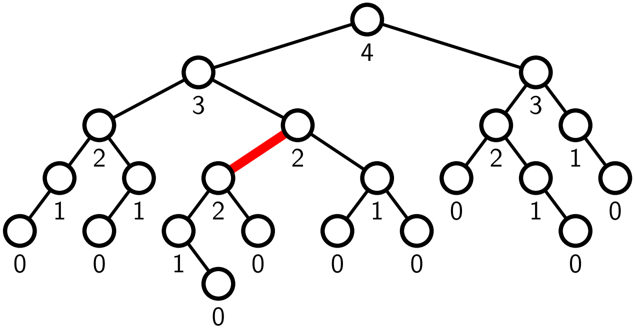
Safe nodes: When do we stop anomalies?

A node is insertion-**safe** if its stops propagating **zero-edges**.



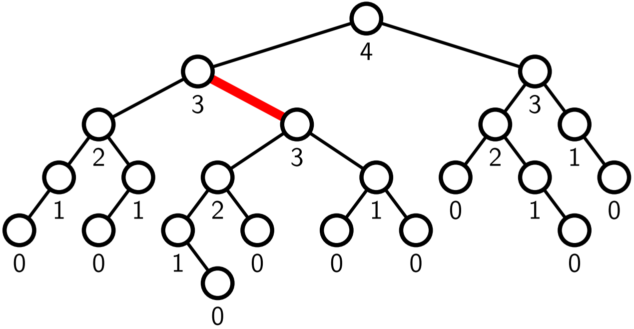
Safe nodes: When do we stop anomalies?

A node is insertion-**safe** if its stops propagating **zero-edges**.



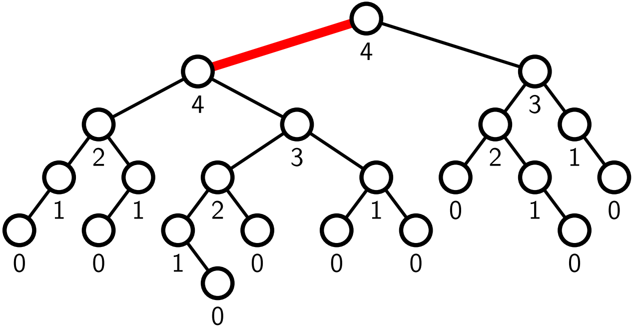
Safe nodes: When do we stop anomalies?

A node is insertion-**safe** if its stops propagating **zero-edges**.



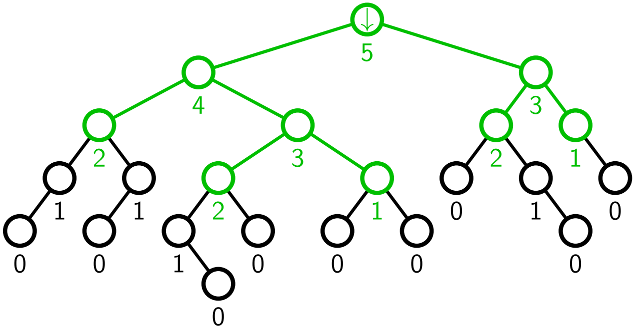
Safe nodes: When do we stop anomalies?

A node is insertion-**safe** if its stops propagating **zero-edges**.



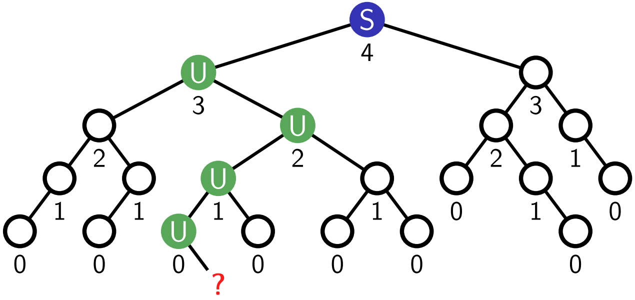
Safe nodes: When do we stop anomalies?

A node is insertion-**safe** if its stops propagating **zero-edges**.



Safe nodes: When do we stop anomalies?

A node is insertion-**safe** if its stops propagating **zero-edges**.

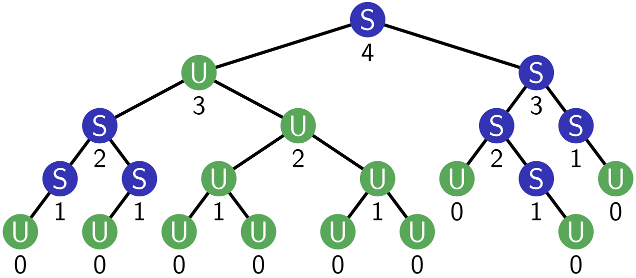


Safe nodes: When do we stop anomalies?

A node is insertion-**safe** if its stops propagating **zero-edges**.

Nodes of rank $r \geqslant 2$ are insertion-**safe** when they are not **full**.

(Other insertion-safe nodes are those with rank $r = 1$ and one 2-child.)



Safe nodes: When do we stop anomalies?

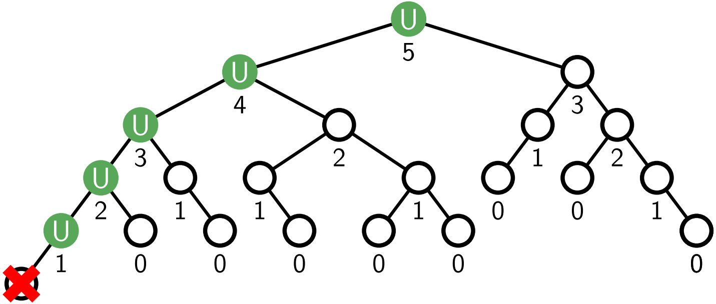
A node is insertion-**safe** if its stops propagating **zero-edges**.
Nodes of rank $r \geqslant 2$ are insertion-**safe** when they are not **full**.
(Other insertion-safe nodes are those with rank $r = 1$ and one 2-child.)

A node is deletion-**safe** if it stops propagating **four-nodes**.
Whether a node is deletion-**safe** depends on a finite neighbourhood **and**
on which branch contains the leaf to delete.

Safe nodes: When do we stop anomalies?

A node is insertion-**safe** if its stops propagating **zero-edges**.
Nodes of rank $r \geqslant 2$ are insertion-**safe** when they are not **full**.
(Other insertion-safe nodes are those with rank $r = 1$ and one 2-child.)

A node is deletion-**safe** if it stops propagating **four-nodes**.
Whether a node is deletion-**safe** depends on a finite neighbourhood **and**
on which branch contains the leaf to delete.



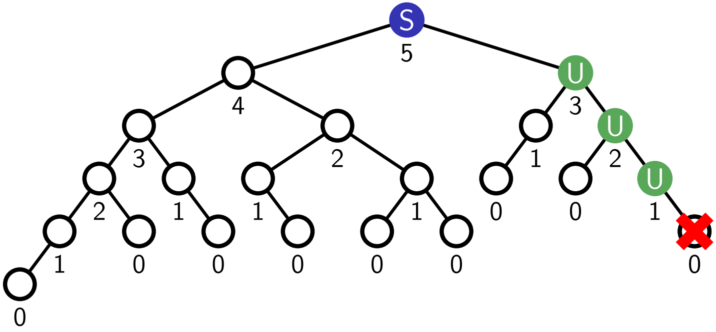
Safe nodes: When do we stop anomalies?

A node is insertion-**safe** if its stops propagating **zero-edges**.
Nodes of rank $r \geqslant 2$ are insertion-**safe** when they are not **full**.

(Other insertion-safe nodes are those with rank $r = 1$ and one 2-child.)

A node is deletion-**safe** if it stops propagating **four-nodes**.
Whether a node is deletion-**safe** depends on a finite neighbourhood **and**

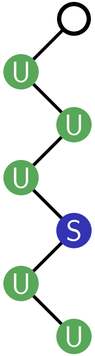
on which branch contains the leaf to delete.



Finding or creating safe nodes (1/2)

Top-down insertion algorithm:

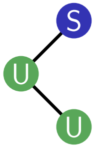
1. Look for a **safe** node on your insertion branch.



Finding or creating safe nodes (1/2)

Top-down insertion algorithm:

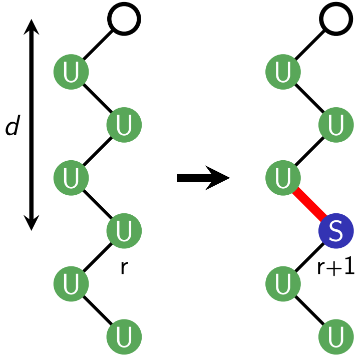
1. Look for a **safe** node on your insertion branch.
2. If you succeed quickly, restart from that node. (write cost = 0)



Finding or creating safe nodes (1/2)

Top-down insertion algorithm:

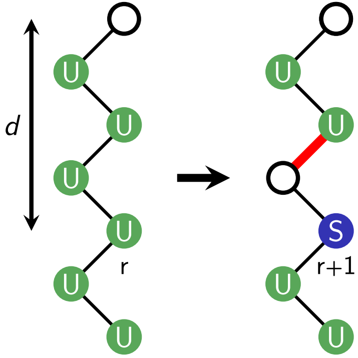
1. Look for a **safe** node on your insertion branch.
2. If you succeed quickly, restart from that node. (write cost = 0)
3. If you fail,
   - **promote** an **unsafe** node at depth d & make it **safe**;



Finding or creating safe nodes (1/2)

Top-down insertion algorithm:

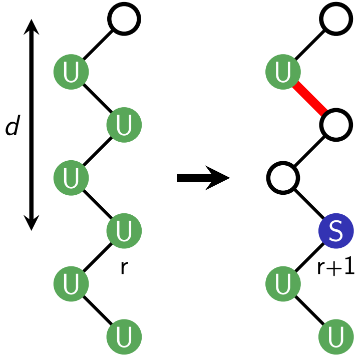
1. Look for a **safe** node on your insertion branch.
2. If you succeed quickly, restart from that node. (write cost = 0)
3. If you fail,
   - **promote** an **unsafe** node at depth d & make it **safe**;
   - **propagate** to the top the **zero-edge** you just created.



Finding or creating safe nodes (1/2)

Top-down insertion algorithm:

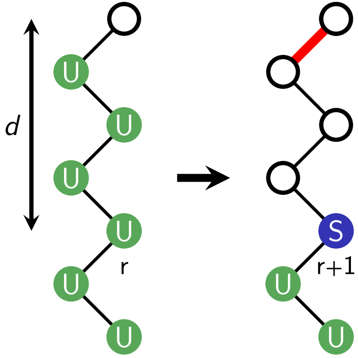
1. Look for a **safe** node on your insertion branch.
2. If you succeed quickly, restart from that node. (write cost = 0)
3. If you fail,
   - **promote** an **unsafe** node at depth d & make it **safe**;
   - **propagate** to the top the **zero-edge** you just created.



Finding or creating safe nodes (1/2)

Top-down insertion algorithm:

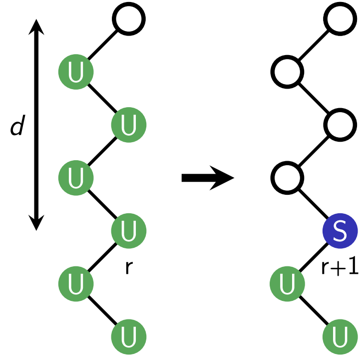
1. Look for a **safe** node on your insertion branch.
2. If you succeed quickly, restart from that node. (write cost = 0)
3. If you fail,
   - **promote** an **unsafe** node at depth d & make it **safe**;
   - **propagate** to the top the **zero-edge** you just created.



Finding or creating safe nodes (1/2)

Top-down insertion algorithm:

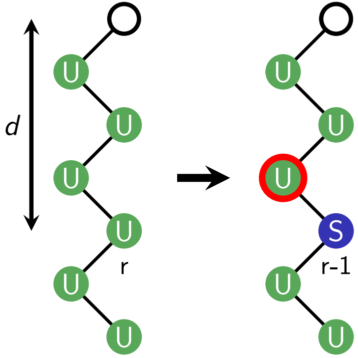
1. Look for a **safe** node on your insertion branch.
2. If you succeed quickly, restart from that node. (write cost $= 0$)
3. If you fail, (write cost $= \mathcal{O}(d+1)$ & $\Delta\mathrm{Pot} \leqslant \mathcal{O}(1) - d$)
   - **promote** an **unsafe** node at depth d & make it **safe**;
   - **propagate** to the top the **zero-edge** you just created.



Finding or creating safe nodes (2/2)

Top-down deletion algorithm:

1. Look for a **safe** node on your deletion branch.
2. If you succeed quickly, restart from that node. (write cost $= 0$)
3. If you fail, (write cost $= \mathcal{O}(d+1)$ & $\Delta\text{Pot} \leqslant \mathcal{O}(1) - d$)
   - **demote** an **unsafe** node at depth d & make it **safe**;
   - **propagate** to the top the **four-node** you just created.

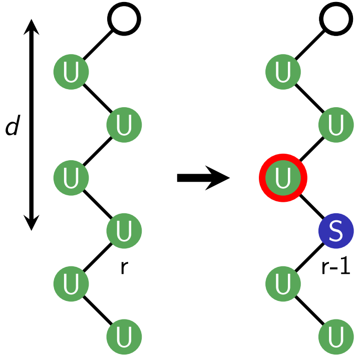


Finding or creating safe nodes (2/2)

Top-down deletion algorithm:

1. Look for a **safe** node on your deletion branch.
2. If you succeed quickly, restart from that node. (write cost $= 0$)
3. If you fail, (write cost $= \mathcal{O}(d+1)$ & $\Delta\text{Pot} \leqslant \mathcal{O}(1) - d$)
   - **demote** an **unsafe** node at depth d & make it **safe**;
   - **propagate** to the top the **four-node** you just created.

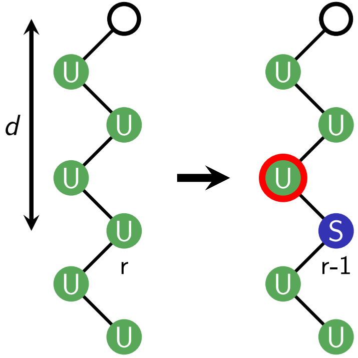
Challenge: Demote deletion-unsafe nodes & make them safe!



Finding or creating safe nodes (2/2)

Top-down deletion algorithm:

1. Look for a **safe** node on your deletion branch.
2. If you succeed quickly, restart from that node. (write cost $= 0$)
3. If you fail, (write cost $= \mathcal{O}(d+1)$ & $\Delta\text{Pot} \leqslant \mathcal{O}(1) - d$)
   - **demote** an **unsafe** node at depth d & make it **safe**;
   - **propagate** to the top the **four-node** you just created.



Challenge: Demote deletion-unsafe nodes & make them safe!
Answer: When facing 4 unsafe nodes in a row, you can do it!

Updating AVL trees efficiently top-down

Theorem

Starting from the empty AVL tree, q top-down queries trigger $\mathcal{O}(q)$ write operations.

Proof:

Tree potential decreases with each batch of d transient operations once d is large enough.

With q queries + B batches:

- tree potential increases by $\mathcal{O}(q) - \mathsf{B}$ or less, and
- tree potential remains non-negative, hence
- $\mathsf{B} = \mathcal{O}(q)$.

Contents

Like all over balanced tree structures. . .

AVL trees enjoy:

1. Efficient bottom-up updating algorithms.

Like all over balanced tree structures...

AVL trees enjoy:

1. Efficient bottom-up updating algorithms;
2. Efficient top-down updating algorithms.

Like all over balanced tree structures. . .

AVL trees enjoy:

1. Efficient bottom-up updating algorithms;
2. Efficient top-down updating algorithms.

Ongoing tasks:

3. Improving d; (Currently, $d = 20$)
4. Deleting **internal nodes** in an (efficient) top-down manner.

Like all over balanced tree structures. . .

AVL trees enjoy:

1. Efficient bottom-up updating algorithms;
2. Efficient top-down updating algorithms.

Ongoing tasks:

3. Improving d; (Currently, $d = 20$)
4. Deleting **internal nodes** in an (efficient) top-down manner;
5. Adapting **rank-based** analysis from weak AVL trees.

Like all over balanced tree structures. . .

AVL trees enjoy:

1. Efficient bottom-up updating algorithms;
2. Efficient top-down updating algorithms.

Ongoing tasks:

3. Improving d; (Currently, $d = 20$)
4. Deleting **internal nodes** in an (efficient) top-down manner;
5. Adapting **rank-based** analysis from weak AVL trees;
6. Adapting these algorithms to **stratified** AVL trees.

Like all over balanced tree structures...

AVL trees enjoy:

1. Efficient bottom-up updating algorithms;
2. Efficient top-down updating algorithms.

Ongoing tasks:

3. Improving d; (Currently, $d = 20$)
4. Deleting **internal nodes** in an (efficient) top-down manner;
5. Adapting **rank-based** analysis from weak AVL trees;
6. Adapting these algorithms to **stratified** AVL trees.

Main take-away:

7. Analyse algorithms and data structures that you love and adapt them!

Bibliography

G. Adel'son-Velskii & E. Landis, *An algorithm for organization of information* 1962

J. Nievergelt & E. Reingold, *Binary search trees of bounded balance* 1972

L. Guibas & R. Sedgewick, *A dichromatic framework for balanced trees* 1978

N. Blum and K. Mehlhorn, *Average number of rebalancing operations in weight-balanced trees* 1980

J. van Leeuwen & M. Overmars, *Stratified balanced search trees* 1983

K. Mehlhorn & A. Tsakalidis, *An amortized analysis of insertions into AVL-trees* 1986

T. Lai & D. Wood, *A top-down updating algorithm for weight-balanced trees* 1993

B. Haeupler, S. Sen & R. Tarjan, *Rank-balanced trees* 2015

M. Amani, K. Lai & R. Tarjan, *Amortized rotation cost in AVL trees* 2016

V. Jugé, *Efficient top-down updates in AVL trees* 2025^{+}


MERCI POUR VOTRE ATTENTION !
NE POSEZ PAS DE QUESTIONS TROP DIFFICILES S'IL VOUS PLAÎT !