DeltaNI: An Efficient Labeling Scheme for Versioned Hierarchical Data

Jan Finis   Robert Brunel   Alfons Kemper
Thomas Neumann   Franz Färber   Norman May

Technische Universität München

SAP AG

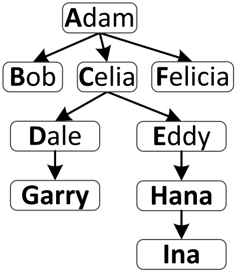
Motivation

Enterprise-Resource-Planning (ERP) systems
use a lot of dynamic hierarchical data!

Enterprise-Resource-Planning (ERP) systems use a lot of dynamic hierarchical data!

Examples:

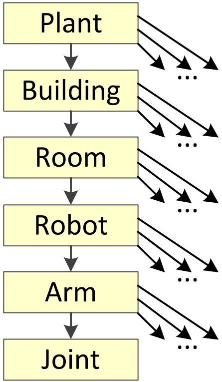
- Human Resources (HR) hierarchy
  - 1 million nodes
  - Some subtree moves (around 10-15%)



Enterprise-Resource-Planning (ERP) systems use a lot of dynamic hierarchical data!

Examples:

- Human Resources (HR) hierarchy
  - 1 million nodes
  - Some subtree moves (around 10-15%)
- Asset hierarchies
  - 10 — 100 million nodes
  - A lot of subtree moves (50% or more)

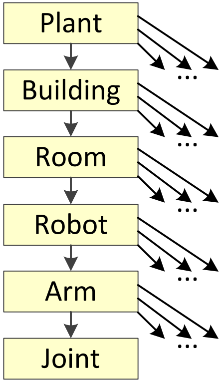


Enterprise-Resource-Planning (ERP) systems use a lot of dynamic hierarchical data!

Examples:

- Human Resources (HR) hierarchy
  - 1 million nodes
  - Some subtree moves (around 10-15%)
- Asset hierarchies
  - 10 — 100 million nodes
  - A lot of subtree moves (50% or more)

- Problem: Current indexing approaches do not support subtree moves!

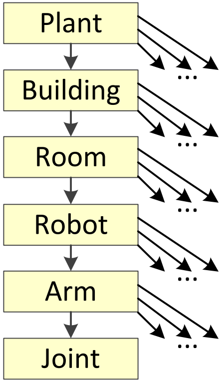


Enterprise-Resource-Planning (ERP) systems use a lot of dynamic hierarchical data!

Examples:

- Human Resources (HR) hierarchy
  - 1 million nodes
  - Some subtree moves (around 10-15%)
- Asset hierarchies
  - 10 — 100 million nodes
  - A lot of subtree moves (50% or more)

- Problem: Current indexing approaches do not support subtree moves!
- Challenge: Versioning required for accountability



Hierarchical Data

- Hierarchical Relationship over tuples of a table

Name	Boss	Salary	...
Adam	NULL	80,000	...
Bob	Adam	55,000	...
Celia	Adam	70,000	...
Dale	Celia	55,000	...
Eddy	Celia	45,000	...
Felicia	Adam	60,000	...
Gina	Felicia	75,000	...
Hana	Gina	45,000	...

Hierarchical Data

- Hierarchical Relationship over tuples of a table

Name	Boss	Salary	...
Adam	NULL	80,000	...
Bob	Adam	55,000	...
Celia	Adam	70,000	...
Dale	Celia	55,000	...
Eddy	Celia	45,000	...
Felicia	Adam	60,000	...
Gina	Felicia	75,000	...
Hana	Gina	45,000	...

- Queries over structural properties, e.g., subtree

```
SELECT name, salary FROM /Employee[name='‘Celia’']//*
```

Hierarchical Data

- Hierarchical Relationship over tuples of a table

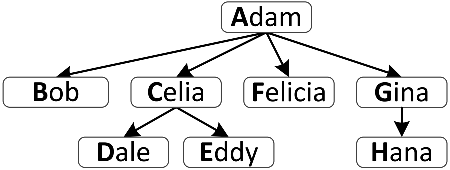
Name	Boss	Salary	...
Adam	NULL	80,000	...
Bob	Adam	55,000	...
Celia	Adam	70,000	...
Dale	Celia	55,000	...
Eddy	Celia	45,000	...
Felicia	Adam	60,000	...
Gina	Felicia	75,000	...
Hana	Gina	45,000	...

- Queries over structural properties, e.g., subtree

```
SELECT name, salary FROM /Employee[name=‘‘Celia’’]//*
```

- Scope: Index the hierarchy structure

- Hierarchical Relationship over tuples of a table



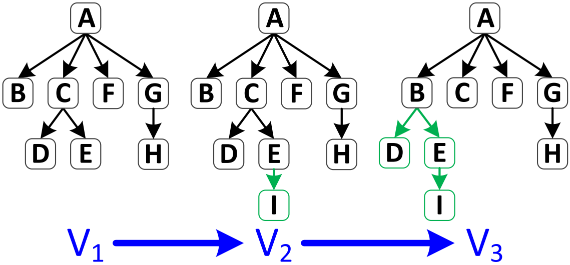
Name	Salary	...
Adam	80,000	...
Bob	55,000	...
Celia	70,000	...
Dale	55,000	...
Eddy	45,000	...
Felicia	60,000	...
Gina	75,000	...
Hana	45,000	...

- Queries over structural properties, e.g., subtree

```
SELECT name, salary FROM /Employee[name=‘‘Celia’’]//*
```

- Scope: Index the hierarchy structure

- Multiple versions of a hierarchy (1000+)
  - Updates at latest version create new version
  - Versioning of the table out of scope
  - Possibly branching history



- Versioned Queries

`SELECT name, salary FROM /Employee[name=‘‘Celia’’]//* IN` V_2

Goals

Goal: An efficient index for versioned hierarchies to speed up ERP systems (and other hierarchical databases).

Goals

Goal: An efficient index for versioned hierarchies to speed up ERP systems (and other hierarchical databases).

Desired properties:

- Efficient queries in all versions

Goals

Goal: An efficient index for versioned hierarchies to speed up ERP systems (and other hierarchical databases).

Desired properties:

- Efficient queries in all versions
- Low space consumption
  - Large hierarchies
  - Long histories
  - Main-memory database

Goals

Goal: An efficient index for versioned hierarchies to speed up ERP systems (and other hierarchical databases).

Desired properties:

- Efficient queries in all versions
- Low space consumption
  - Large hierarchies
  - Long histories
  - Main-memory database
- Efficient updates in latest version
  - Insert, delete, and subtree move

Goals

Goal: An efficient index for versioned hierarchies to speed up ERP systems (and other hierarchical databases).

Desired properties:

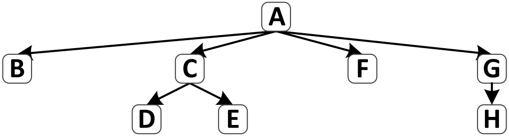
- Efficient queries in all versions
- Low space consumption
  - Large hierarchies
  - Long histories
  - Main-memory database
- Efficient updates in latest version
  - Insert, delete, and subtree move
- Allow branching histories

- Widely used hierarchy indexing: Labeling Schemes

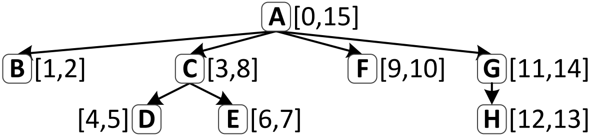
- Widely used hierarchy indexing: Labeling Schemes
  - Each node carries fixed set of labels
  - Queries can be answered by only considering labels

- Widely used hierarchy indexing: Labeling Schemes
  - Each node carries fixed set of labels
  - Queries can be answered by only considering labels
  - Widely applied in, e.g., XPath processing
  - Examples: pre/post, ORDPATH, nested intervals (NI)

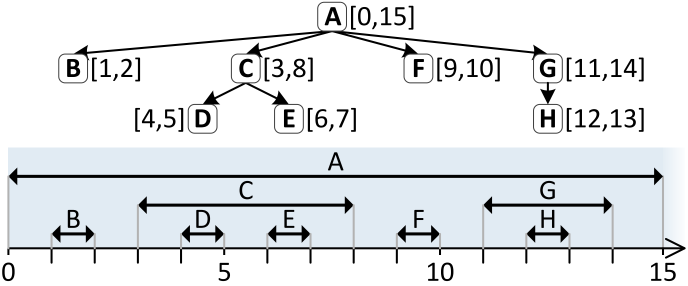
- Widely used hierarchy indexing: Labeling Schemes
  - Each node carries fixed set of labels
  - Queries can be answered by only considering labels
  - Widely applied in, e.g., XPath processing
  - Examples: pre/post, ORDPATH, nested intervals (NI)



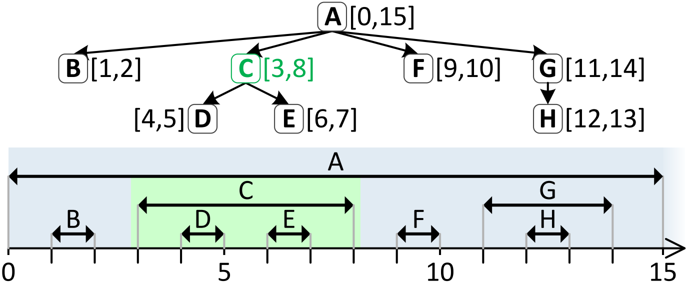
- Widely used hierarchy indexing: Labeling Schemes
  - Each node carries fixed set of labels
  - Queries can be answered by only considering labels
  - Widely applied in, e.g., XPath processing
  - Examples: pre/post, ORDPATH, nested intervals (NI)



- Widely used hierarchy indexing: Labeling Schemes
  - Each node carries fixed set of labels
  - Queries can be answered by only considering labels
  - Widely applied in, e.g., XPath processing
  - Examples: pre/post, ORDPATH, nested intervals (NI)



- Widely used hierarchy indexing: Labeling Schemes
  - Each node carries fixed set of labels
  - Queries can be answered by only considering labels
  - Widely applied in, e.g., XPath processing
  - Examples: pre/post, ORDPATH, nested intervals (NI)



`/Employee[name=‘‘Celia’’]//*` ⇒ “All nodes in [3,8]”

Challenges

- Challenge 1: Efficient Query Support

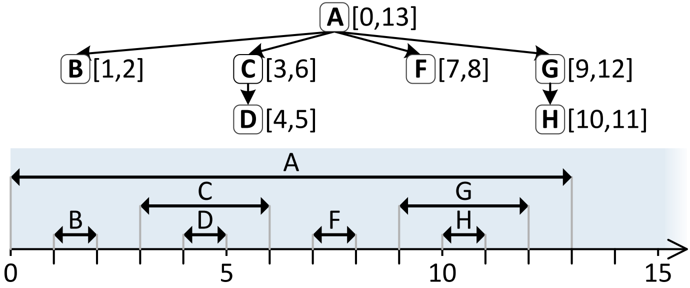
Challenges

- Challenge 1: Efficient Query Support
  - NI can be used to answer queries efficiently ✓

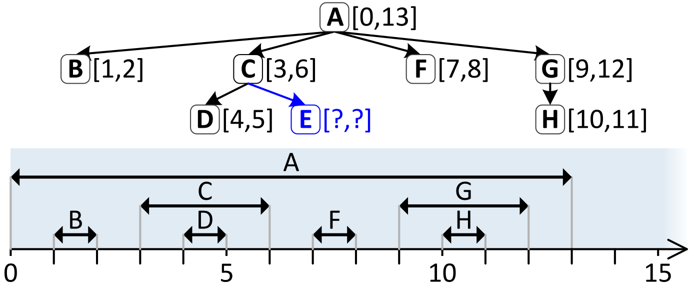
Challenges

- Challenge 1: Efficient Query Support
  - NI can be used to answer queries efficiently ✓
- Challenge 2: Efficient Update Support

- Challenge 1: Efficient Query Support
  - NI can be used to answer queries efficiently ✓
- Challenge 2: Efficient Update Support
  - NI not dynamic ($\mathcal{O}(n)$ bounds change per update) ☹

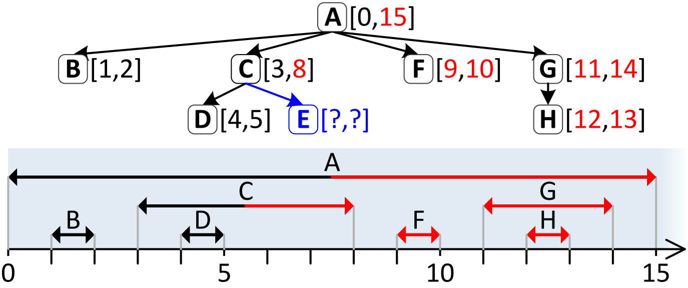


- Challenge 1: Efficient Query Support
  - NI can be used to answer queries efficiently ✓
- Challenge 2: Efficient Update Support
  - NI not dynamic ($\mathcal{O}(n)$ bounds change per update) ☹



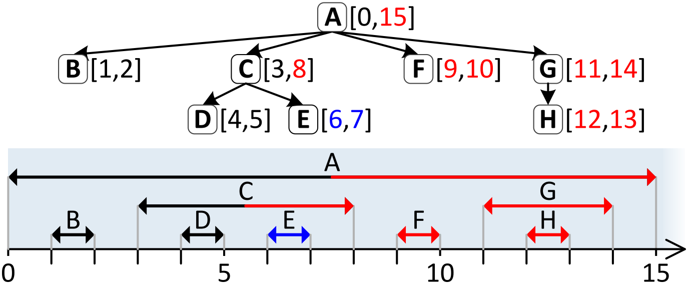
Challenges

- Challenge 1: Efficient Query Support
  - NI can be used to answer queries efficiently ✓
- Challenge 2: Efficient Update Support
  - NI not dynamic ($\mathcal{O}(n)$ bounds change per update) ☹



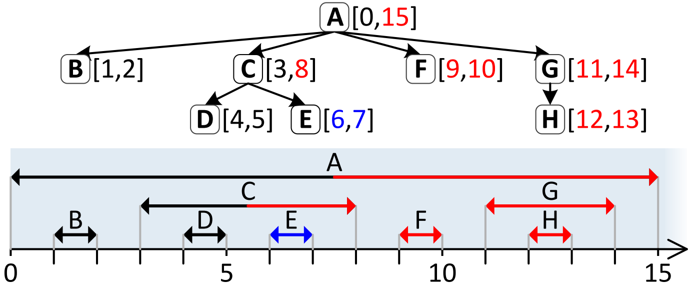
$\mathcal{O}(n)$ bound changes

- Challenge 1: Efficient Query Support
  - NI can be used to answer queries efficiently ✓
- Challenge 2: Efficient Update Support
  - NI not dynamic ($\mathcal{O}(n)$ bounds change per update) ☹



$\mathcal{O}(n)$ bound changes

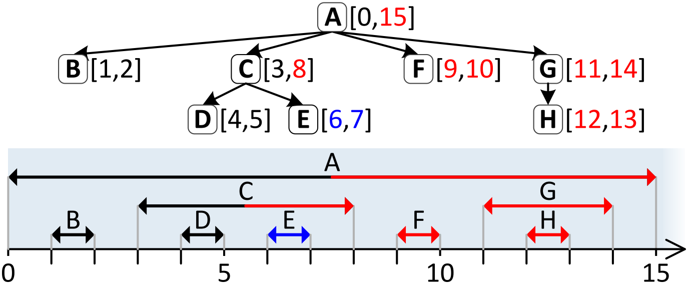
- Challenge 1: Efficient Query Support
  - NI can be used to answer queries efficiently ✓
- Challenge 2: Efficient Update Support
  - NI not dynamic ($\mathcal{O}(n)$ bounds change per update) ☹
- Challenge 3: Space Consumption of Histories



$\mathcal{O}(n)$ bound changes

Challenges

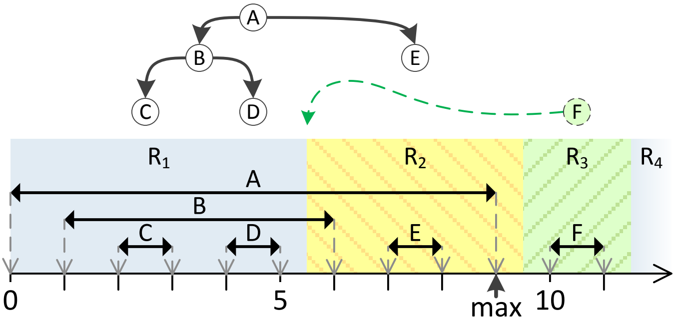
- Challenge 1: Efficient Query Support
  - NI can be used to answer queries efficiently ✓
- Challenge 2: Efficient Update Support
  - NI not dynamic ($\mathcal{O}(n)$ bounds change per update) ☹
- Challenge 3: Space Consumption of Histories
  - $\mathcal{O}(n)$ bounds change per update need to be stored ☹



$\mathcal{O}(n)$ bound changes

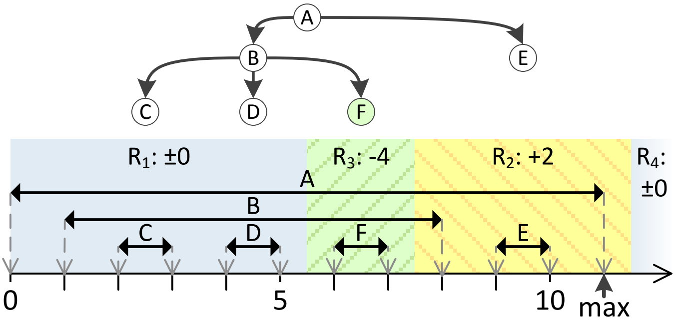
- Observation: Each update can be represented by a swap of two ranges of bounds

- Observation: Each update can be represented by a swap of two ranges of bounds
- Insert Node: Before

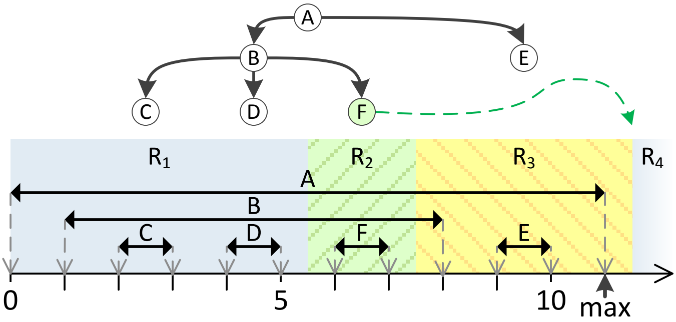


Swaps

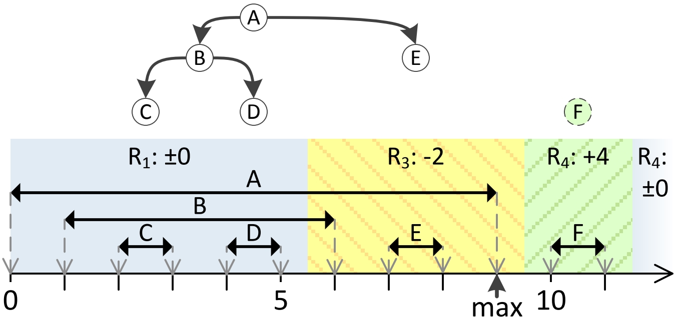
- Observation: Each update can be represented by a swap of two ranges of bounds
- Insert Node: After



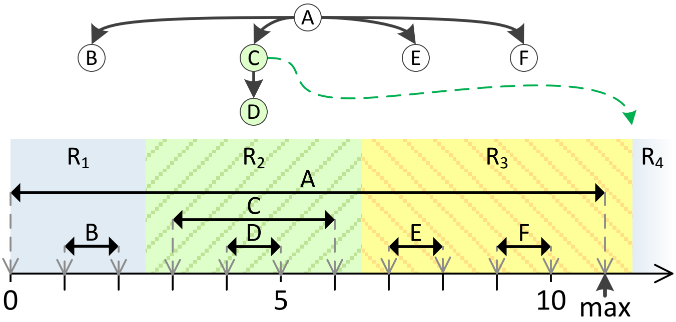
- Observation: Each update can be represented by a swap of two ranges of bounds
- Delete Node: Before



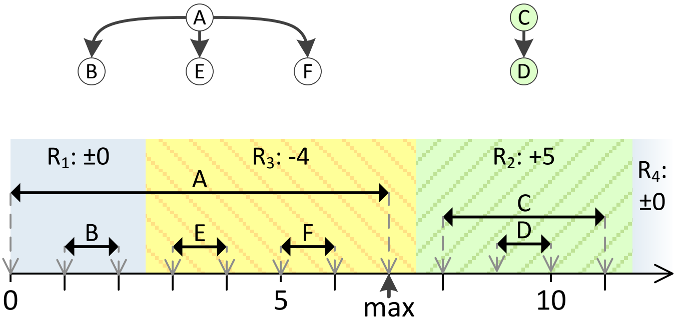
- Observation: Each update can be represented by a swap of two ranges of bounds
- Delete Node: After



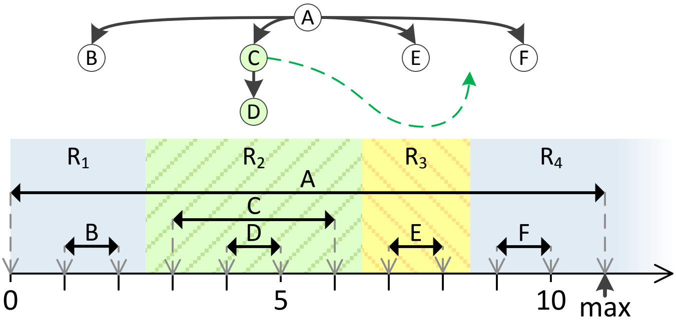
- Observation: Each update can be represented by a swap of two ranges of bounds
- Delete Subtree: Before



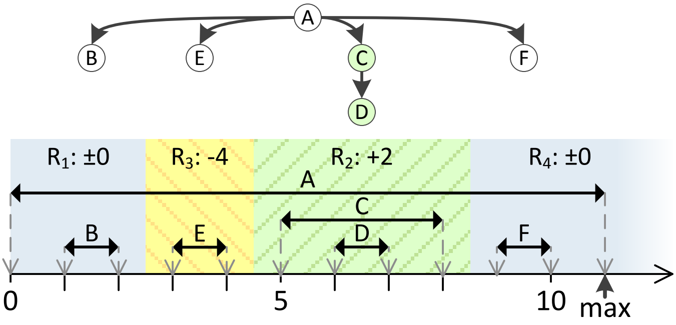
- Observation: Each update can be represented by a swap of two ranges of bounds
- Delete Subtree: After



- Observation: Each update can be represented by a swap of two ranges of bounds
- Move Subtree: Before



- Observation: Each update can be represented by a swap of two ranges of bounds
- Move Subtree: After



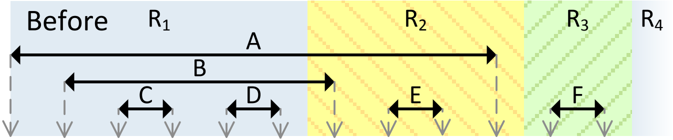
- Observation: Each update can be represented by a swap of two ranges of bounds

Storing Swaps

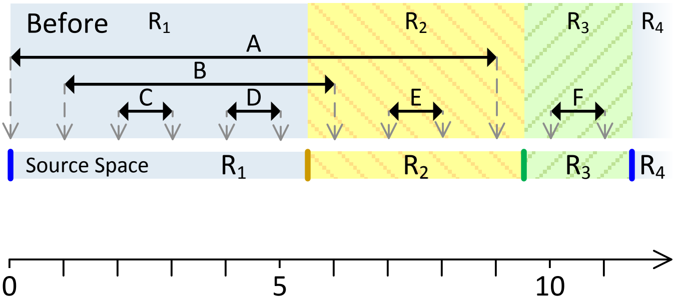
- Observation: Each update can be represented by a swap of two ranges of bounds
- Idea: Simply store that swap instead of the changed bounds

Storing Swaps

- Observation: Each update can be represented by a swap of two ranges of bounds
- Idea: Simply store that swap instead of the changed bounds

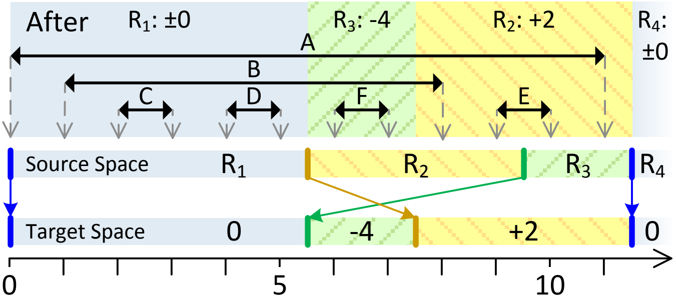


- Observation: Each update can be represented by a swap of two ranges of bounds
- Idea: Simply store that swap instead of the changed bounds

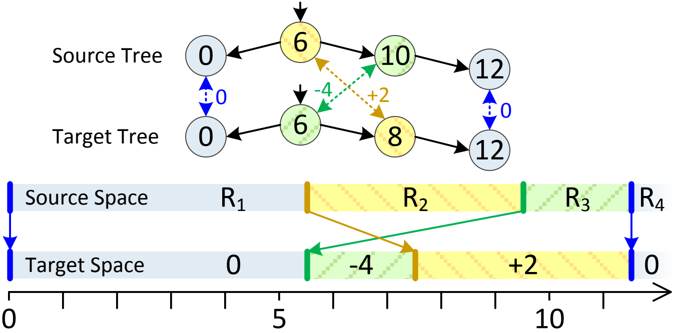


Storing Swaps

- Observation: Each update can be represented by a swap of two ranges of bounds
- Idea: Simply store that swap instead of the changed bounds

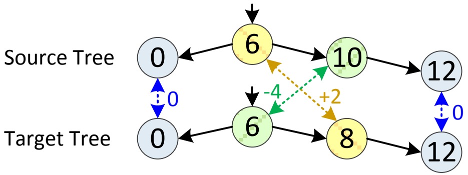


- Representation: Two balanced (binary) search trees (*"double tree"*)
- Node content: Lower border and link to other tree

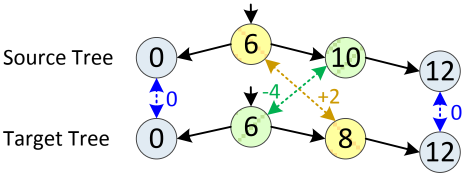


NI Deltas

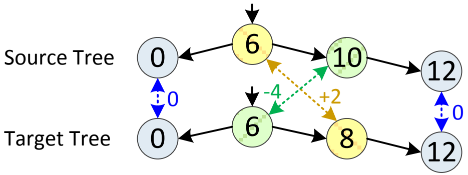
- The double tree represents a function $\delta : \mathbb{N} \mapsto \mathbb{N}$



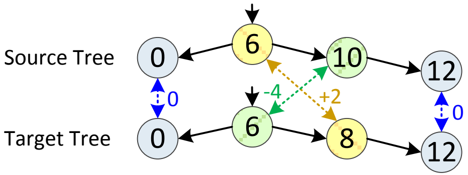
- The double tree represents a function $\delta : \mathbb{N} \mapsto \mathbb{N}$
- δ maps interval bounds from source space to target space



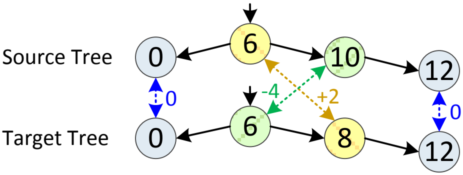
- The double tree represents a function $\delta : \mathbb{N} \mapsto \mathbb{N}$
- δ maps interval bounds from source space to target space
- Let b be a bound in source space, then $\delta(b)$ is equivalent bound in target space



- The double tree represents a function $\delta : \mathbb{N} \mapsto \mathbb{N}$
- δ maps interval bounds from source space to target space
- Let b be a bound in source space, then $\delta(b)$ is equivalent bound in target space
- Given an NI encoding in version V_i and a delta $\delta_{V_i \rightarrow V_j}$ from version V_i to another version V_j, we can answer queries in V_j



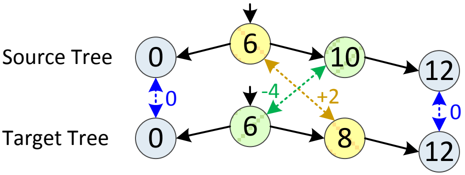
- Computing $\delta(b)$ is easy:



Computing δ on the Double Tree

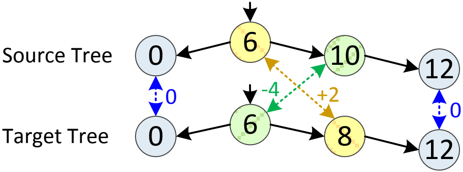
- Computing $\delta(b)$ is easy:
  - Find greatest node in source tree less than b

    $\Rightarrow$ Usual search-tree lookup



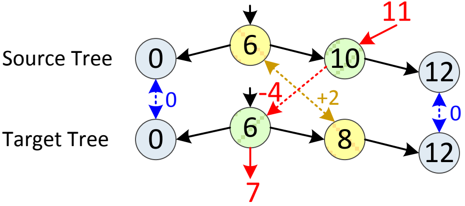
Computing δ on the Double Tree

- Computing $\delta(b)$ is easy:
  - Find greatest node in source tree less than b
    - $\Rightarrow$ Usual search-tree lookup
  - Apply translation of that node



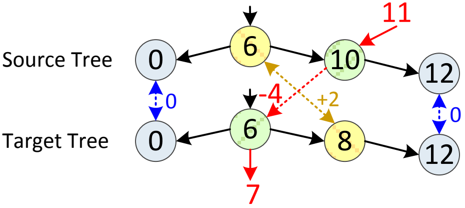
- Computing $\delta(b)$ is easy:
  - Find greatest node in source tree less than b
    - $\Rightarrow$ Usual search-tree lookup
  - Apply translation of that node

$\delta(11) = 7$:



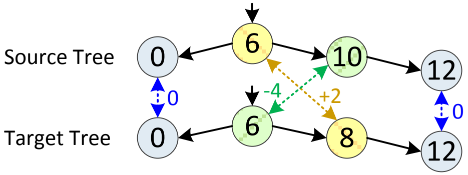
- Computing $\delta(b)$ is easy:
  - Find greatest node in source tree less than b
    - $\Rightarrow$ Usual search-tree lookup
  - Apply translation of that node

$\delta(11) = 7$:



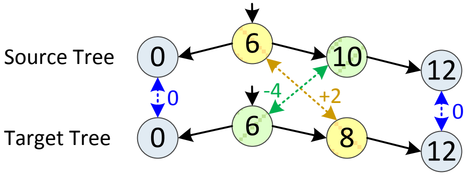
- Computation of $\delta^{-1}(b)$ similar

Does the double tree delta solve the problems?



Does the double tree delta solve the problems?

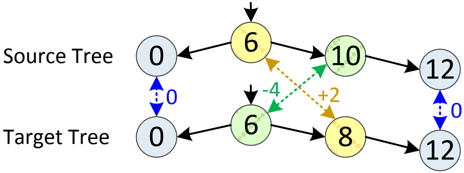
- Challenge 1: Efficient Query Support



Double tree feasible?

Does the double tree delta solve the problems?

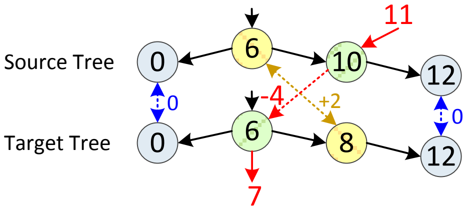
- Challenge 1: Efficient Query Support
  - NI can be used to answer queries efficiently ✓



Double tree feasible?

Does the double tree delta solve the problems?

- Challenge 1: Efficient Query Support
  - NI can be used to answer queries efficiently ✓
  - Calculating $\delta(b)$ (search tree lookup) is in $\mathcal{O}(\log c)$ ✓

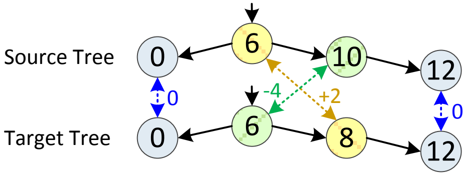


n = number of nodes, c = number of changes in delta

Double tree feasible?

Does the double tree delta solve the problems?

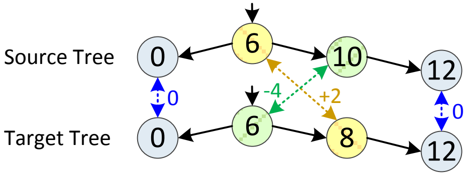
- Challenge 1: Efficient Query Support ✓
- Challenge 2: Space Consumption



n = number of nodes, c = number of changes in delta

Does the double tree delta solve the problems?

- Challenge 1: Efficient Query Support ✓
- Challenge 2: Space Consumption
  - Storing all changed bounds: $\mathcal{O}(n)$ space ☹

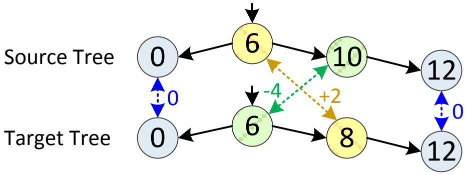


n = number of nodes, c = number of changes in delta

Double tree feasible?

Does the double tree delta solve the problems?

- Challenge 1: Efficient Query Support ✓
- Challenge 2: Space Consumption
  - Storing all changed bounds: $\mathcal{O}(n)$ space ☹
  - Storing only range borders: $\mathcal{O}(c)$ space ☺

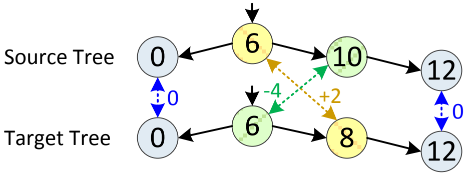


n = number of nodes, c = number of changes in delta

Double tree feasible?

Does the double tree delta solve the problems?

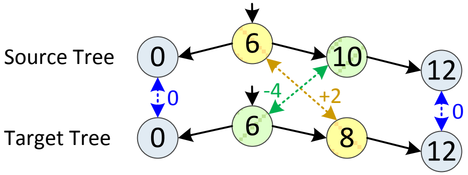
- Challenge 1: Efficient Query Support ✓
- Challenge 2: Space Consumption ✓
- Challenge 3: Efficient Update Support



n = number of nodes, c = number of changes in delta

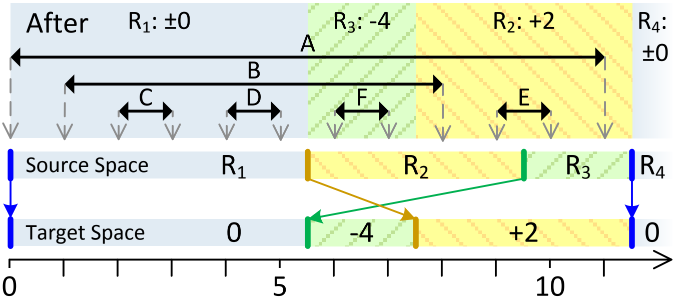
Does the double tree delta solve the problems?

- Challenge 1: Efficient Query Support ✓
- Challenge 2: Space Consumption ✓
- Challenge 3: Efficient Update Support ?



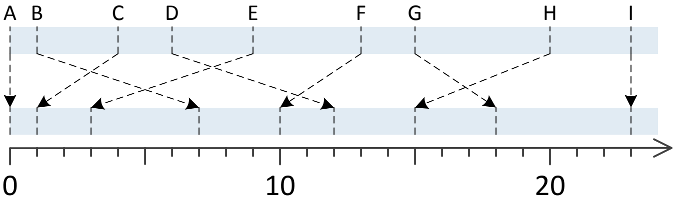
n = number of nodes, c = number of changes in delta

- Until now, we only considered deltas with one change

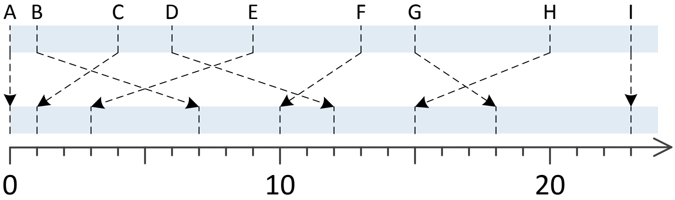


Updating Deltas

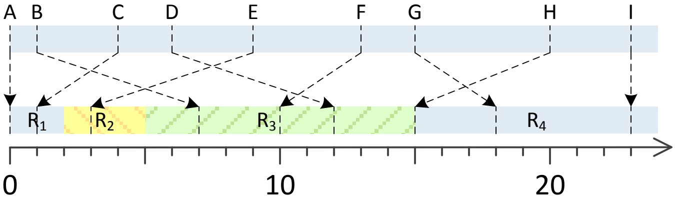
- Until now, we only considered deltas with one change
- How to build deltas with more changes?



Updating Deltas

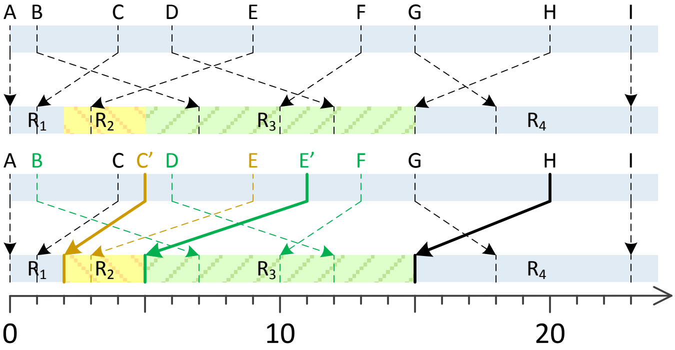


- Task: Swap range R_2 with R_3



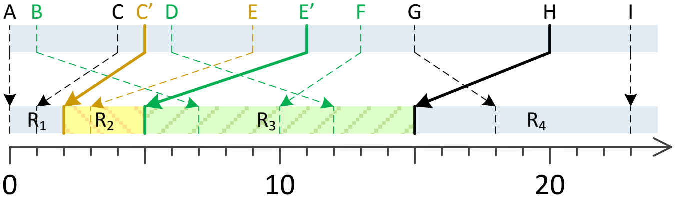
Updating Deltas

- Step 1: Insert range borders

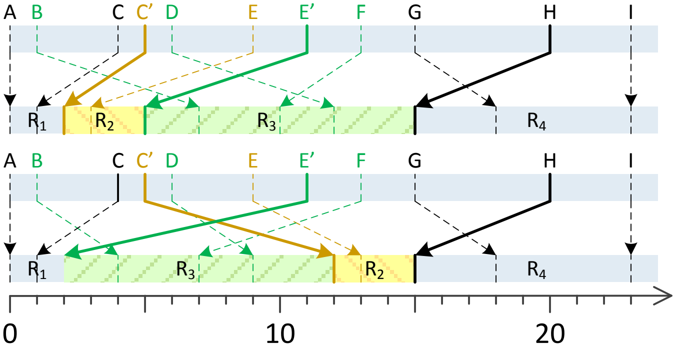


- Search tree insert: $\mathcal{O}(\log c)$ ✓

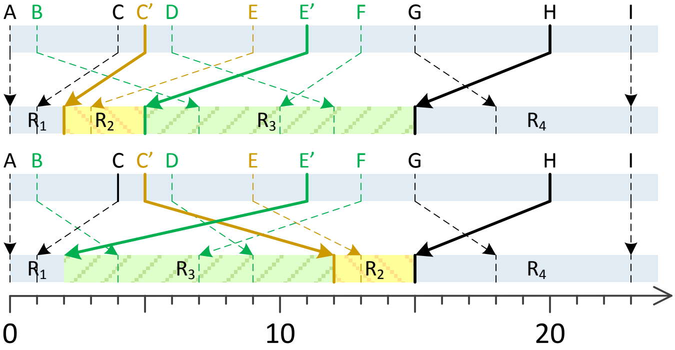
Updating Deltas



- Step 2: Swap borders in R_2 and R_3

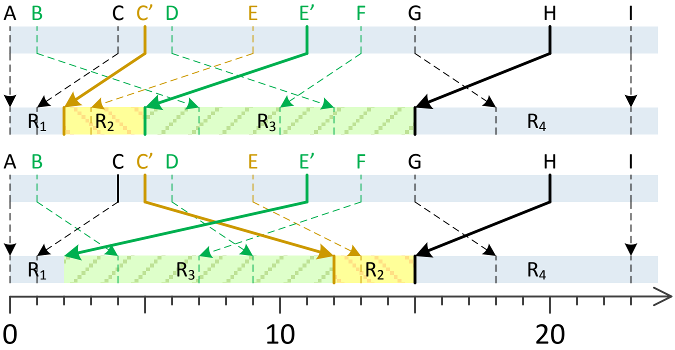


- Step 2: Swap borders in R_2 and R_3

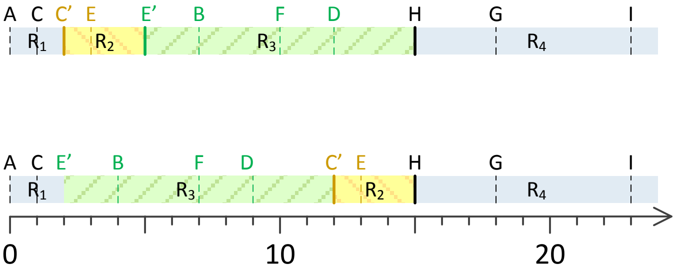


- Naive: Delete and reinsert all changed borders: $\mathcal{O}(c \log c)$ ☹
- $\Rightarrow$ Better approach required

- Observation: Only target space changes

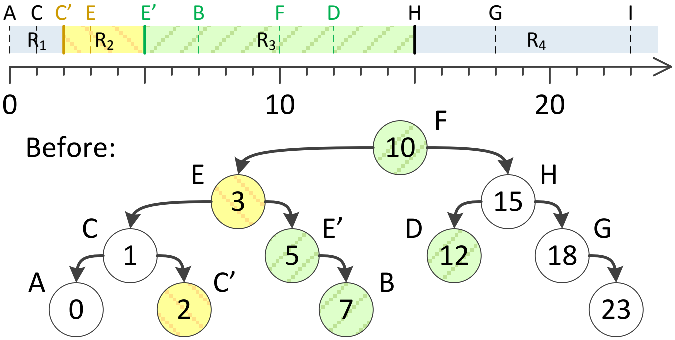


- Observation: Only target space changes

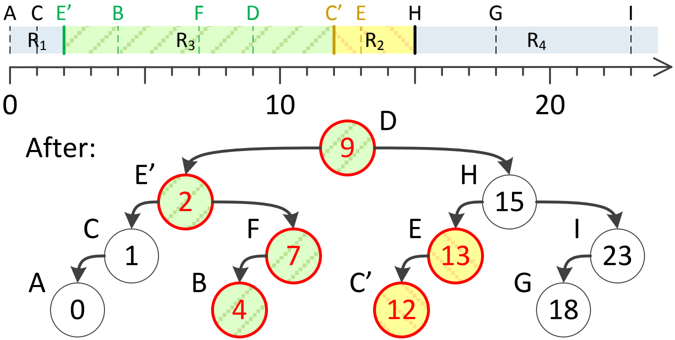


Efficient Border Swap

- Observation: Only target space changes
- Steps: Adjust keys $\mathcal{O}(c)$ keys, swap $\mathcal{O}(c)$ nodes

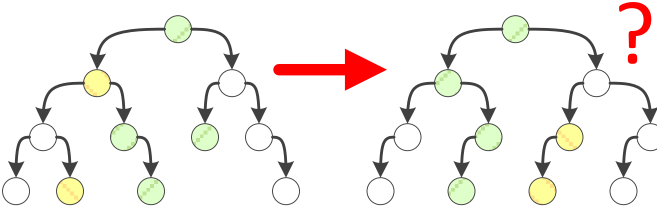


- Observation: Only target space changes

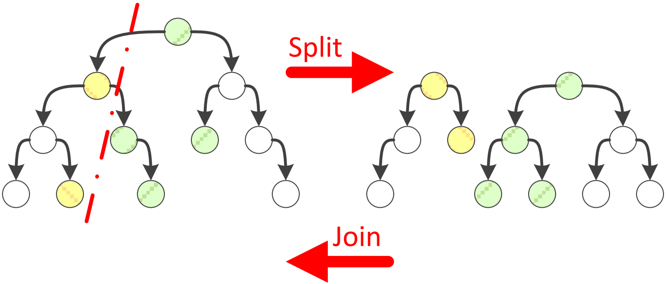


Efficient Node Rearrangement

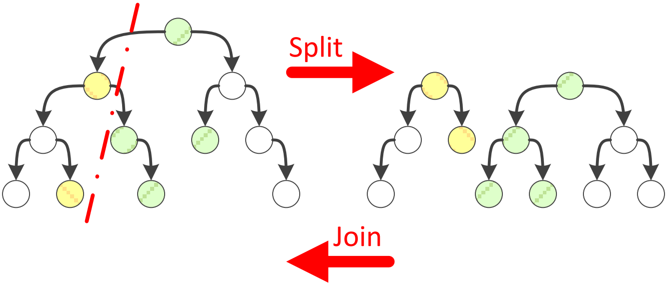
- How to swap $\mathcal{O}(c)$ nodes in a search tree in $\mathcal{O}(\log c)$?



- How to swap $\mathcal{O}(c)$ nodes in a search tree in $\mathcal{O}(\log c)$?
- Solution: Split and join



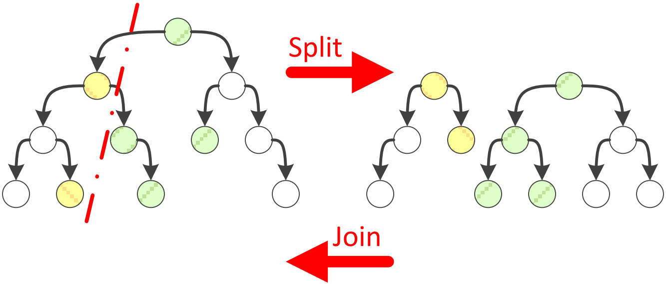
- How to swap $\mathcal{O}(c)$ nodes in a search tree in $\mathcal{O}(\log c)$?



- Split: Split a tree into two new balanced trees

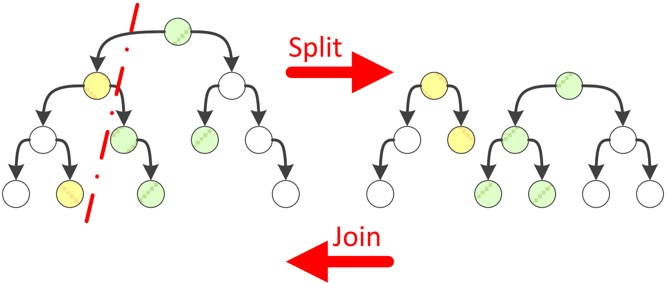
Efficient Node Rearrangement

- How to swap $\mathcal{O}(c)$ nodes in a search tree in $\mathcal{O}(\log c)$?



- Split: Split a tree into two new balanced trees
- Join: Concatenate two trees to one balanced one

- How to swap $\mathcal{O}(c)$ nodes in a search tree in $\mathcal{O}(\log c)$?



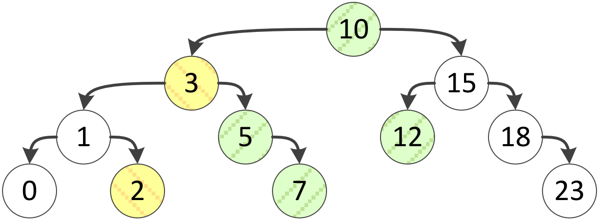
- Split: Split a tree into two new balanced trees
- Join: Concatenate two trees to one balanced one
- Both operations run in $\mathcal{O}(\log c)$ ✓

Updating Keys Efficiently

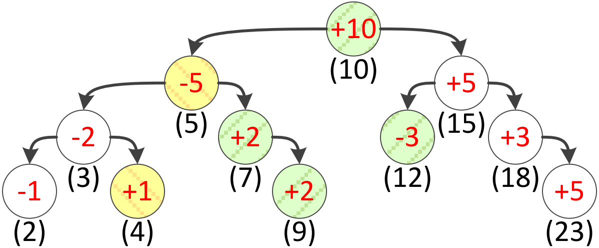
- Split and join can rearrange nodes efficiently
- But: Keys are not updated $\Rightarrow$ search tree condition violated!
- Updating one by one would require $\mathcal{O}(c)$

Updating Keys Efficiently

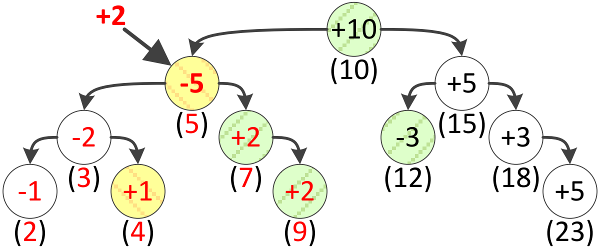
- Split and join can rearrange nodes efficiently
- But: Keys are not updated $\Rightarrow$ search tree condition violated!
- Updating one by one would require $\mathcal{O}(c)$
- Solution: Accumulation tree
  - $\Rightarrow$ Node key: Sum of all keys on path to root



- Split and join can rearrange nodes efficiently
- But: Keys are not updated ⇒ search tree condition violated!
- Updating one by one would require $\mathcal{O}(c)$
- Solution: Accumulation tree
  - ⇒ Node key: Sum of all keys on path to root

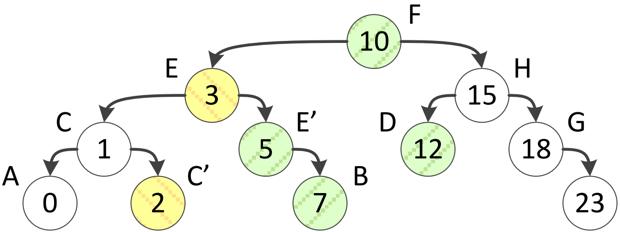


- Split and join can rearrange nodes efficiently
- But: Keys are not updated ⇒ search tree condition violated!
- Updating one by one would require $\mathcal{O}(c)$
- Solution: Accumulation tree
  - ⇒ Node key: Sum of all keys on path to root

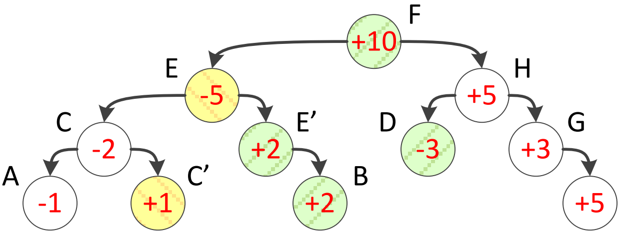


- Changing all keys in a subtree: $\mathcal{O}(1)$

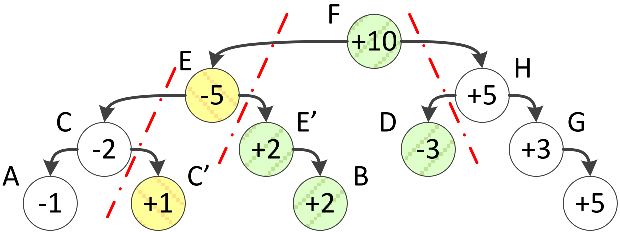
- Using split/join and the accumulation tree, updating in $\mathcal{O}(\log c)$ is possible
- Target tree before update:



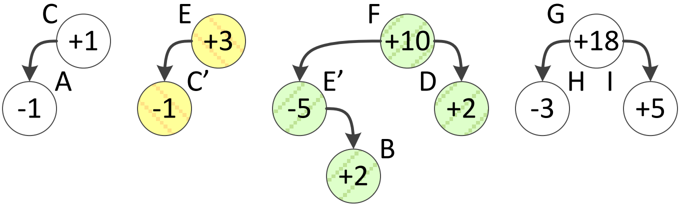
- Using split/join and the accumulation tree, updating in $\mathcal{O}(\log c)$ is possible
- Target tree with accumulation before update:



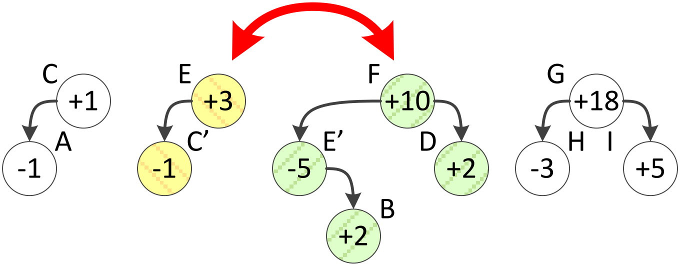
- Using split/join and the accumulation tree, updating in $\mathcal{O}(\log c)$ is possible
- Step 1: Split tree ($\mathcal{O}(\log c)$)



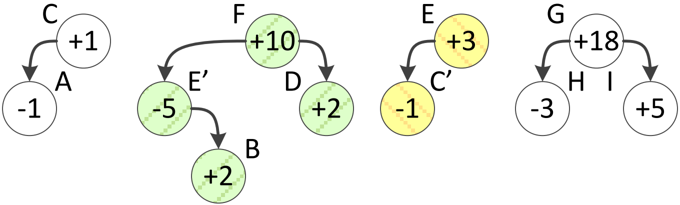
- Using split/join and the accumulation tree, updating in $\mathcal{O}(\log c)$ is possible
- Step 1: Split tree ($\mathcal{O}(\log c)$)



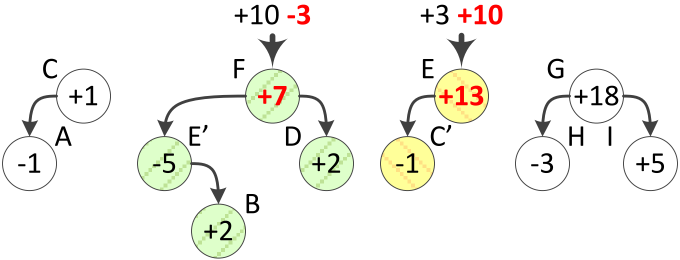
- Using split/join and the accumulation tree, updating in $\mathcal{O}(\log c)$ is possible
- Rearrange trees (no-op)



- Using split/join and the accumulation tree, updating in $\mathcal{O}(\log c)$ is possible
- Rearrange trees (no-op)

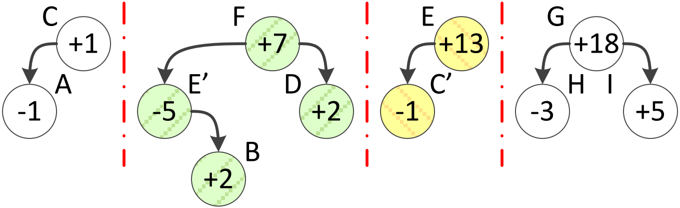


- Using split/join and the accumulation tree, updating in $\mathcal{O}(\log c)$ is possible
- Step 2: Translate keys ($\mathcal{O}(1)$)

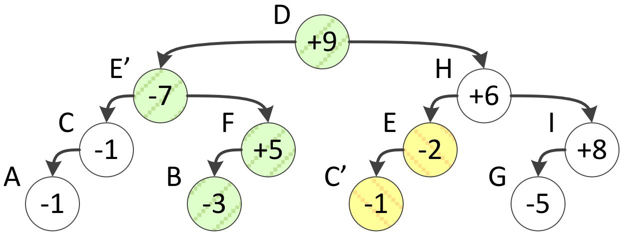


The Swap Algorithm

- Using split/join and the accumulation tree, updating in $\mathcal{O}(\log c)$ is possible
- Step 3: Join trees ($\mathcal{O}(\log c)$)



- Using split/join and the accumulation tree, updating in $\mathcal{O}(\log c)$ is possible
- Final result:



- What we have shown:

- What we have shown:
  - Double tree delta efficiently represents the changes in a version
    - Efficient Queries (NI Encoding)
    - Efficient Updates (Swap Algorithm)
    - Low Space Consumption ($\mathcal{O}(c)$)

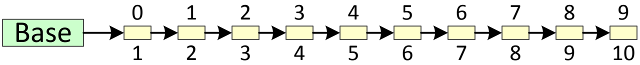
- What we have shown:
  - Double tree delta efficiently represents the changes in a version
    - Efficient Queries (NI Encoding)
    - Efficient Updates (Swap Algorithm)
    - Low Space Consumption ($\mathcal{O}(c)$)
- What is missing:

Overview

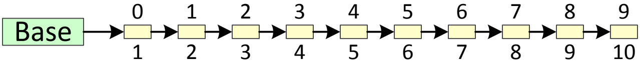
- What we have shown:
  - Double tree delta efficiently represents the changes in a version
    - Efficient Queries (NI Encoding)
    - Efficient Updates (Swap Algorithm)
    - Low Space Consumption ($\mathcal{O}(c)$)
- What is missing:
  - How to represent whole version histories efficiently?

- Assume:
  - Linear history of n versions $V_0, \ldots, V_{n-1}$
  - Constantly bounded number of changes c per version
- What we need:
  - V_0 has a fully materialized NI encoding
  - We need deltas that lead to each other version (transitively)
  - E.g., $\delta_{0\rightarrow 3}$ and $\delta_{3\rightarrow 5}$ lead to V_5 by applying $\delta_{3\rightarrow 5}(\delta_{0\rightarrow 3}(b))$
- Which deltas to store in order to...
  - minimize space consumption?
  - minimize query runtime?

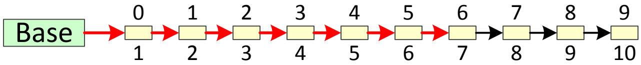
- Minimize space consumption: linear topology



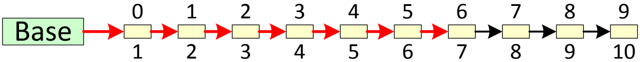
- Minimize space consumption: linear topology
  - $\Rightarrow$ $O(n)$ space consumption ✓



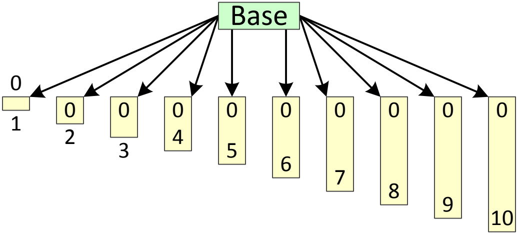
- Minimize space consumption: linear topology
  - $\Rightarrow$ $O(n)$ space consumption ✓
  - $\Rightarrow$ $O(n)$ query time ☹



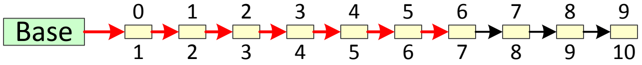
- Minimize space consumption: linear topology
  - $\Rightarrow O(n)$ space consumption ✓
  - $\Rightarrow O(n)$ query time ☹



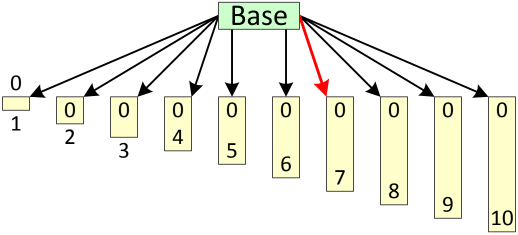
- Minimize query time: star topology



- Minimize space consumption: linear topology
  - $\Rightarrow$ $O(n)$ space consumption ✓
  - $\Rightarrow$ $O(n)$ query time ☹

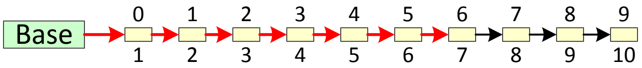


- Minimize query time: star topology
  - $\Rightarrow$ $O(\log n)$ query time ✓

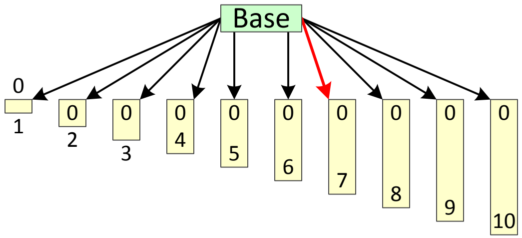


Simple Schemes

- Minimize space consumption: linear topology
  - ⇒ $O(n)$ space consumption ✓
  - ⇒ $O(n)$ query time ☹



- Minimize query time: star topology
  - ⇒ $O(\log n)$ query time ✓
  - ⇒ $O(n^2)$ space consumption ☹

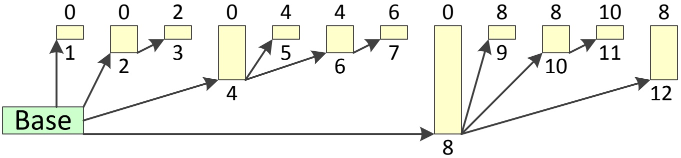


Exponential Scheme

- We need a better space/time tradeoff!

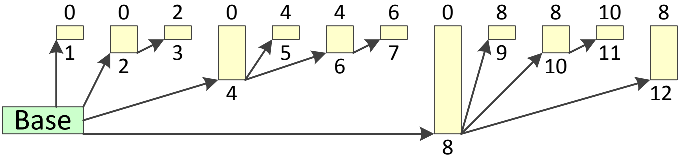
Exponential Scheme

- We need a better space/time tradeoff!
- Solution: Exponential scheme



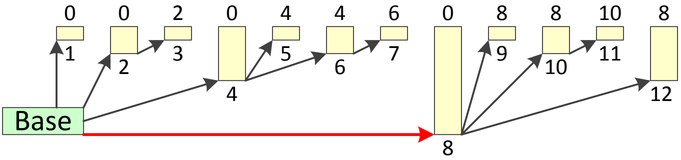
Exponential Scheme

- We need a better space/time tradeoff!
- Solution: Exponential scheme
  - $\Rightarrow$ $O(n \log n)$ space consumption ✓

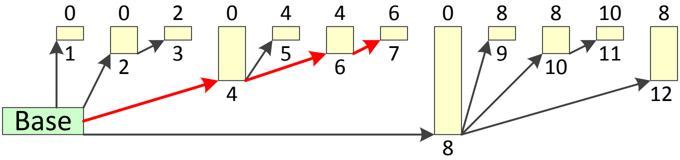


Exponential Scheme

- We need a better space/time tradeoff!
- Solution: Exponential scheme
  - $\Rightarrow$ $O(n \log n)$ space consumption ✓
  - $\Rightarrow$ $O(\log n)$ best case query time ✓



- We need a better space/time tradeoff!
- Solution: Exponential scheme
  - $\Rightarrow$ $O(n \log n)$ space consumption ✓
  - $\Rightarrow$ $O(\log n)$ best case query time ✓
  - $\Rightarrow$ $O(\log^2 n)$ worst case query time ✓



- Baseline: Currently strongest algorithms

Evaluation Baseline

- Baseline: Currently strongest algorithms
  - Labeling with ORDPATH
    - No relabeling $\Rightarrow$ efficient updates
    - Efficient queries

Evaluation Baseline

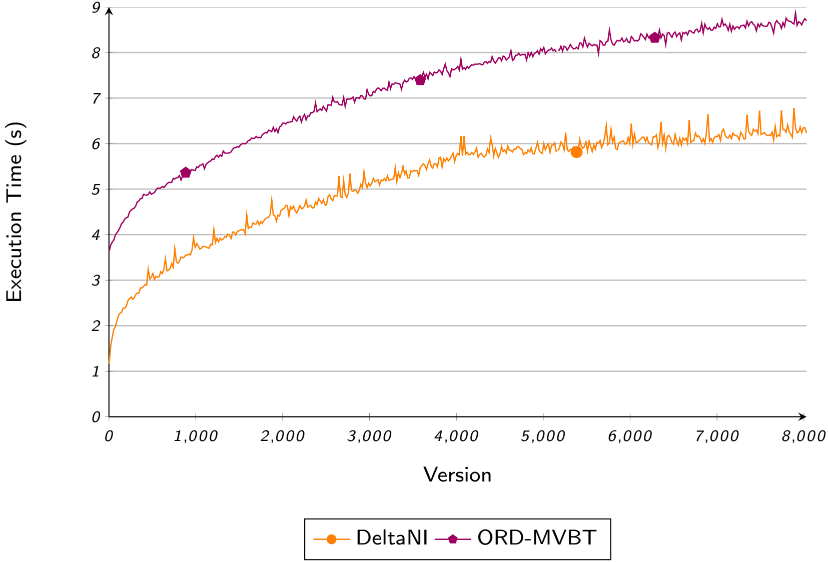
- Baseline: Currently strongest algorithms
  - Labeling with ORDPATH
    - No relabeling $\Rightarrow$ efficient updates
    - Efficient queries
  - Versioning with Multiversion B-Tree (MVBT)
    - Asymptotically optimal query time and space consumption

- Baseline: Currently strongest algorithms ORD-MVBT
  - Labeling with ORDPATH
    - No relabeling $\Rightarrow$ efficient updates
    - Efficient queries
  - Versioning with Multiversion B-Tree (MVBT)
    - Asymptotically optimal query time and space consumption

Evaluation Baseline

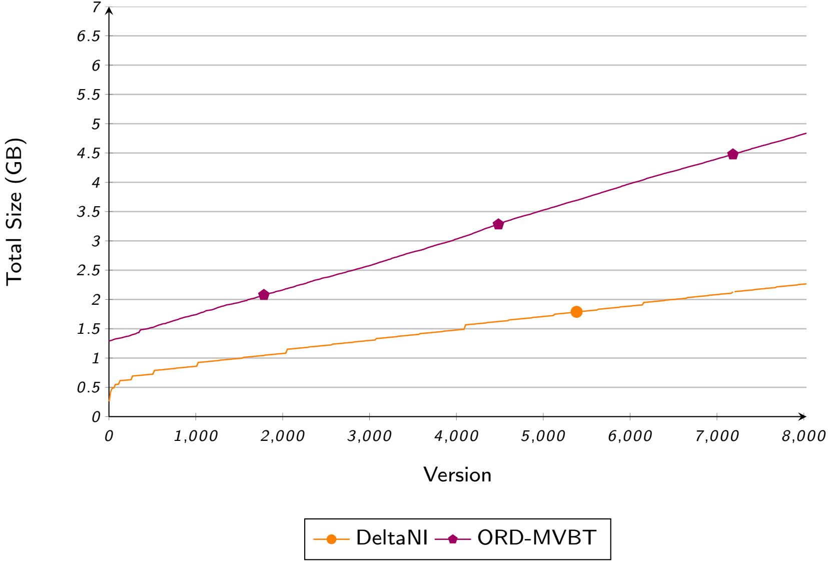
- Baseline: Currently strongest algorithms ORD-MVBT
  - Labeling with ORDPATH
    - No relabeling $\Rightarrow$ efficient updates
    - Efficient queries
  - Versioning with Multiversion B-Tree (MVBT)
    - Asymptotically optimal query time and space consumption
- Improvements with DeltaNI
  - Support of subtree relocation and deletion
  - Branching histories
  - Simple integer comparisons instead of bytewise comparisons

Time for one million queries



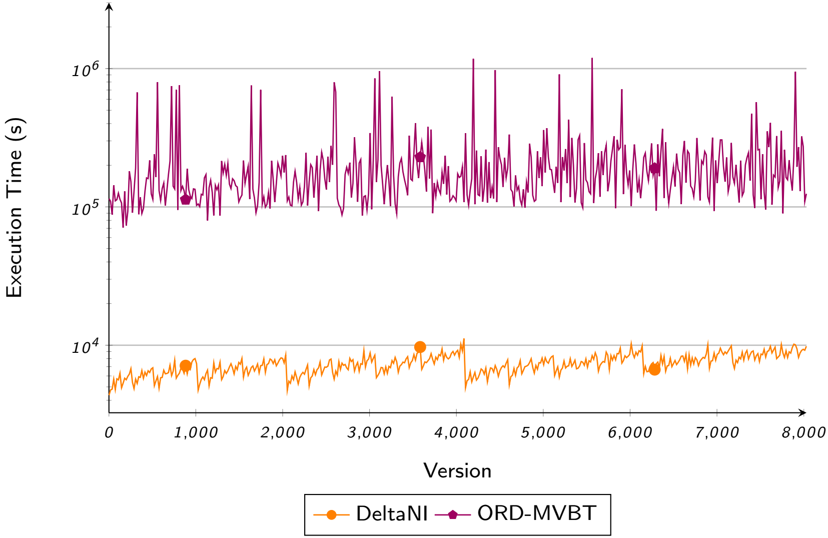
Evaluation: Space Consumption

Space consumption



Evaluation: Update Performance

Time for one million updates



Conclusion

- Core observation: All updates reducable to range swap in the NI encoding

Conclusion

- Core observation: All updates reducable to range swap in the NI encoding
- Double tree interval deltas make NI encoding dynamic
  - $\mathcal{O}(c)$ space consumption
  - $\mathcal{O}(\log c)$ update complexity
  - Even complex updates supported (subtree relocation)

- Core observation: All updates reducable to range swap in the NI encoding
- Double tree interval deltas make NI encoding dynamic
  - $\mathcal{O}(c)$ space consumption
  - $\mathcal{O}(\log c)$ update complexity
  - Even complex updates supported (subtree relocation)
- Versioning via exponential delta-packing scheme
  - Yields reasonable space/time tradeoff

Thank you for your attention!

Any questions?