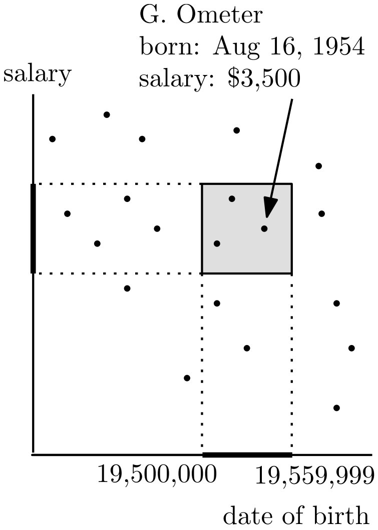
Massive Data Algorithmics

Lecture 4: External Search Trees

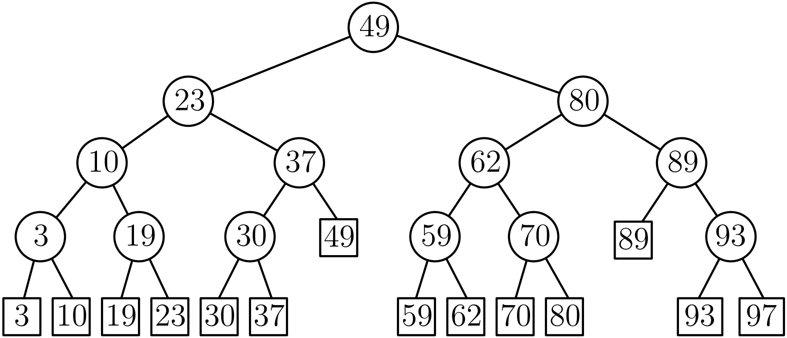
Database queries

A database query may ask for all employees with age between a_1 and a_2, and salary between s_1 and s_2



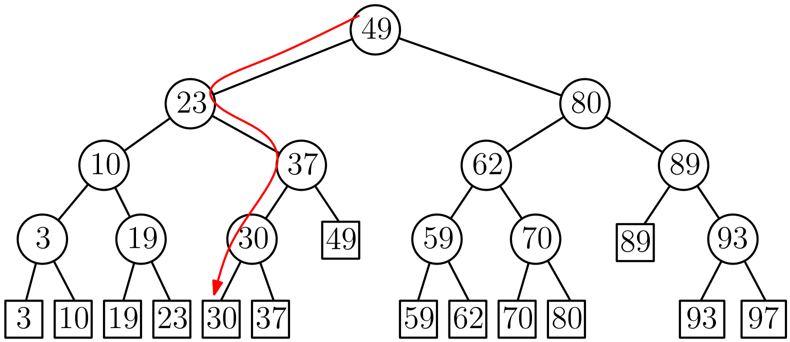
Balanced binary search trees

A balanced binary search tree with the points in the leaves



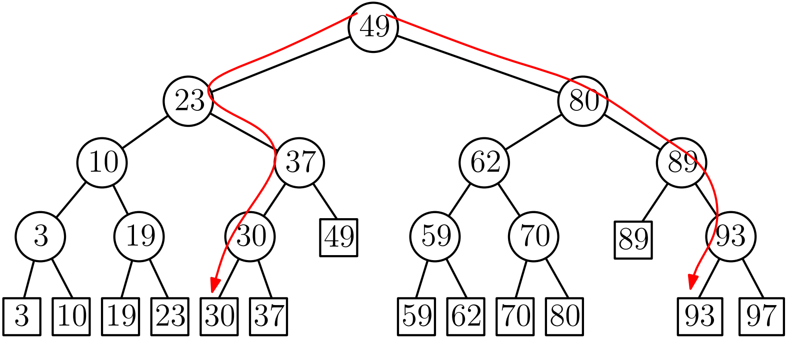
Balanced binary search trees

The search path for 25



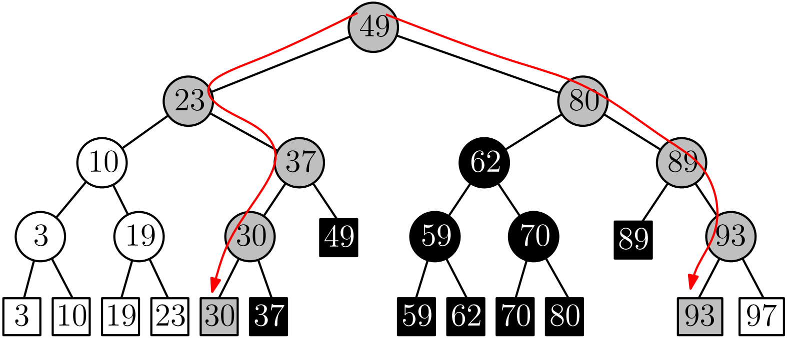
Balanced binary search trees

The search paths for 25 and for 90



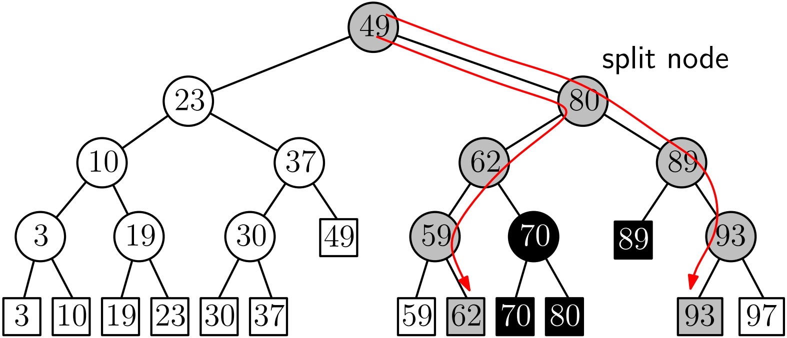
Example 1D range query

A 1-dimensional range query with $[25, 90]$



Example 1D range query

A 1-dimensional range query with $[61, 90]$



Node types for a query

Three types of nodes *for a given query*:

- **White nodes:** never visited by the query
- **Grey nodes:** visited by the query, unclear if they lead to output
- **Black nodes:** visited by the query, whole subtree is output

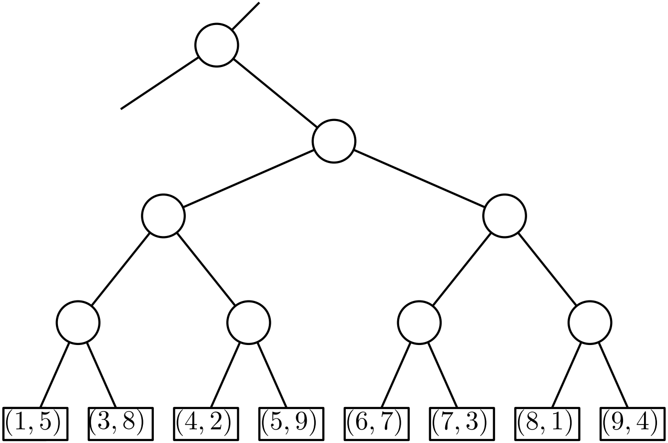
Examining 1D range queries

For any 1D range query, we can identify $O(\log n)$ nodes that together represent all answers to a 1D range query

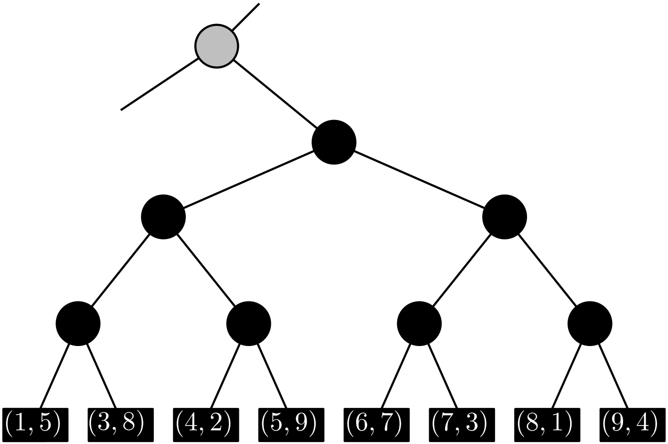
Toward 2D range queries

For any 2d range query, we can identify $O(\log n)$ nodes that together represent all points that have a correct first coordinate

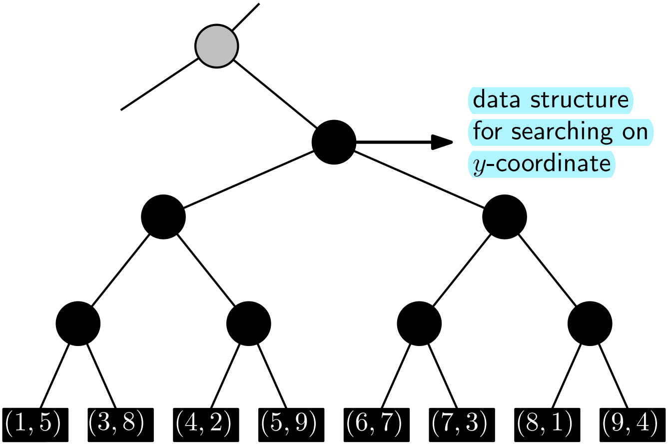
Toward 2D range queries



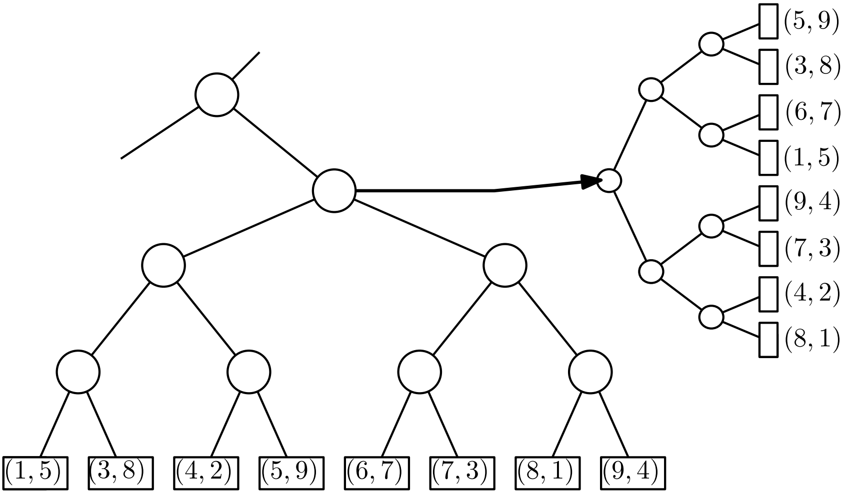
Toward 2D range queries



Toward 2D range queries

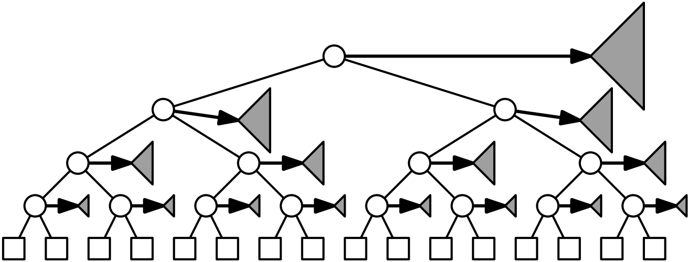


Toward 2D range queries



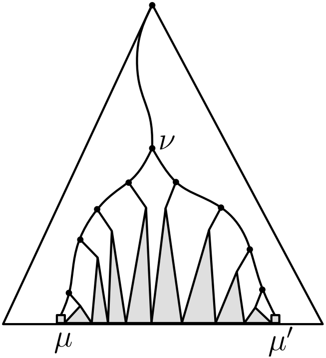
2D range trees

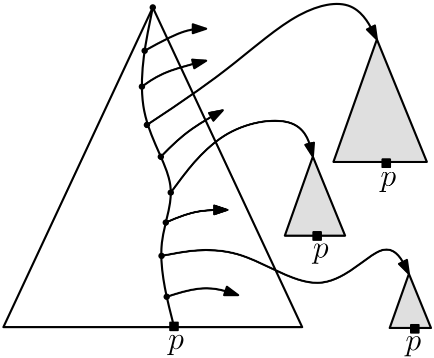
Every internal node stores a whole tree in an *associated structure*, on y-coordinate



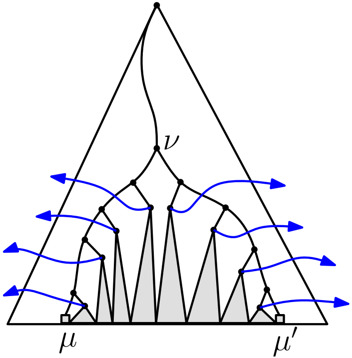
Question: How much storage does this take?

2D range queries



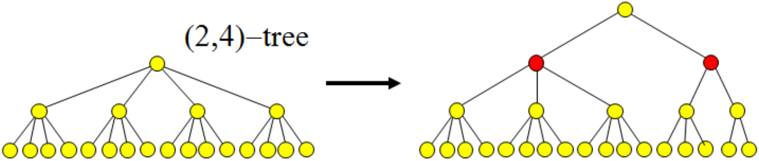


2D range queries



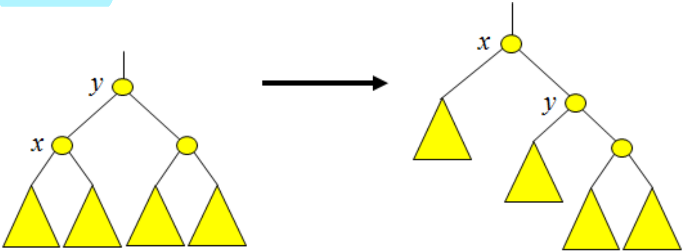
Secondary Structures

- When secondary structures used, a rebalance on v often requires $O(w(v))$ I/Os ($w(v)$ is weight of v)
  - If $\Omega(w(v))$ inserts have to be made below v between operations
  $\Rightarrow$ $O(1)$ amortized split bound
  $\Rightarrow$ $O(\log_B N)$ amortized insert bound
- Nodes in standard B-tree do not have this property



BB[α]-tree

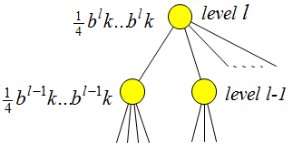
- In internal memory BB[α]-trees have the desired property
- Defined using weight-constraint
  - Ratio between weight of left child and weight of right child of a node v is between α and $1-\alpha$ ($\alpha < 1$) $\Rightarrow$ Height: $O(\log N)$
- If $2/11 < \alpha < 1 - 1/2\sqrt{2}$ rebalancing can be performed using rotations



- Seems hard to implement BB[α]-trees I/O-efficiently

Weight-balanced B-tree

- Idea: Combination of B-tree and BB[α]-tree
  - Weight constraint on nodes instead of degree constraint
  - Rebalancing performed using split/fuse as in B-tree
- Weight-balanced B-tree with parameters b and k ($b > 8$, $k \geq 8$)
  - All leaves on same level and contain between $k/4$ and k elements
  - Internal node v at level l has $w(v) < b^l k$
  - Except for the root, internal node v at level l has $w(v) > 1/4 b^l k$
  - The root has more than one child



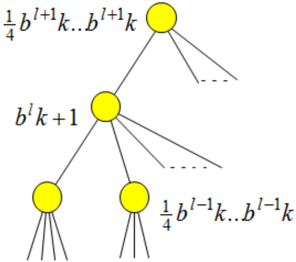
Weight-balanced B-tree

- Every internal node has degree between
  $1/4b^l k/b^{l-1}k = 1/4b$ and $b^l k/(1/4)b^{l-1}k = 4b$
  $\Rightarrow$ Height: $O(\log_b N/k)$

- External memory:
  - Choose $4b = B$ (or even B^c for $0 < c \leq 1$)
  - $k = B$
  $\Rightarrow$ $O(N/B)$ space, $O(\log_B N/B + T/B)$ query

Weight-balanced B-tree Insert

- Search for relevant leaf u and insert new element
- Traverse path from u to root:
  - If level l node v now has $w(v) = b^l k + 1$ then split into nodes v and v with $w(v') \geq \lfloor 1/2(b^l k + 1) \rfloor - b^{l-1} k$ and $w(v'') \leq \lceil 1/2(b^l k + 1) \rceil + b^{l-1} k$

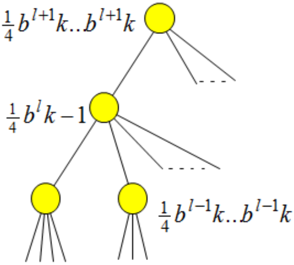


- Algorithm correct since $b^{l-1} k \leq 1/8 b^l k$ such that $w(v') \geq 3/8 b^l k$ and $w(v'') \leq 5/8 b^l k$
  - touch $O(\log_b N/k)$ nodes
- Weight-balance property:
  $\Omega(b^l k)$ updates below v and v before next rebalance operation

Weight-balanced B-tree Insert

- Search for relevant leaf u and insert new element
- Traverse path from u to root:

  - If level l node v now has $w(v) = 1/4b^l k - 1$ then fuse with sibling into nodes v with $2/4b^l k - 1 \leq w(v') \leq 5/4b^l k - 1$
  - If now $w(v') \geq 7/8b^l k$ then split into nodes with weight $\geq 7/16b^l k - 1 - b^{l-1}k \geq 5/16b^l k - 1$ and $\leq 5/8b^l k + b^{l-1}k \leq 6/8b^l k$



- Algorithm correct and touch $O(\log_b N/k)$ nodes
- Weight-balance property:
  $\Omega(b^l k)$ updates below v and v before next rebalance operation

Summary/Conclusion: Weight-balanced B-tree

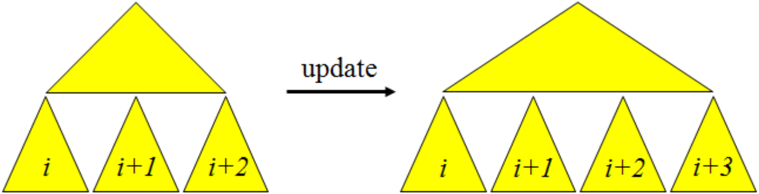
- Weight-balanced B-tree with branching parameter b and leaf parameter $k = \Omega(B)$
  - $O(N/B)$ space
  - Height $O(\log_b N/k)$
  - $O(\log_B N)$ rebalancing operations after update
  - $\Omega(w(v))$ updates below v between consecutive operations on v
- Weight-balanced B-tree with branching parameter B^c and leaf parameter B - Updates in $O(\log_B N)$ and queries in $O(\log_B N + T/B)$ I/Os
- Construction bottom-up in $O(N/B \log_{M/B} N/B)$ I/O

Persistent B-tree

- In some applications we are interested in being able to access previous versions of data structure
  - Databases
  - Geometric data structures (later)
- Partial persistence:
  - Update current version (getting new version)
  - Query all versions
- We would like to have partial persistent B-tree with
  - $O(N/B)$ space N is number of updates performed
  - $O(\log_B N)$ update
  - $O(\log_B N + T/B)$ query in any version

Persistent B-tree

- Easy way to make B-tree partial persistent
  - Copy structure at each operation
  - Maintain version-access structure (B-tree)



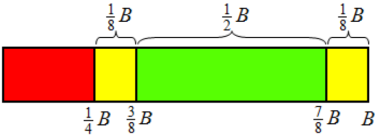
- Good $O(\log_B N)$ query in any version, but
  - $O(N/B)$ I/Os update
  - $O(N^2/B)$ space

Persistent B-tree

- Idea: Elements augmented with existence interval and stored in one structure
- Persistent B-tree with parameter b (> 16):
  - Directed graph
    * Nodes contain elements augmented with existence interval
    * At any time t, nodes with elements alive at time t form B-tree with leaf and branching parameter b
  - B-tree with leaf and branching parameter b on indegree 0 node (roots)
- If $b = B$: Query at any time t in $O(\log_B N + T/B)$ I/Os

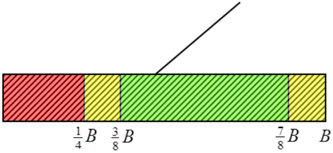
Persistent B-tree:Updates

- Updates performed as in B-tree
  - alive block: containing at least one alive element at current version
  - each alive block must contain at least $1/4B$ alive elements
- To obtain linear space we maintain new-node invariant:
  - New node contains between $3/8B$ and $7/8B$ alive elements and no dead elements



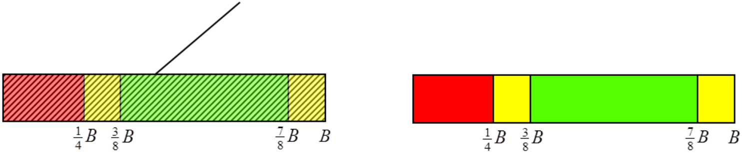
Persistent B-tree Insert

- Search for relevant leaf u and insert new element
- If u contains $B+1$ elements: Block overflow
  - Version split:
    Mark u dead and create new node u' with x alive element
  - If $x > 7/8B$: Strong overflow
  - If $x < 3/8B$: Strong underflow
  - If $3/8B \leq x \leq 7/8B$ then recursively persistently update parent(u):
    Delete reference to u (dead ref.) and insert reference to u' (alive ref.)



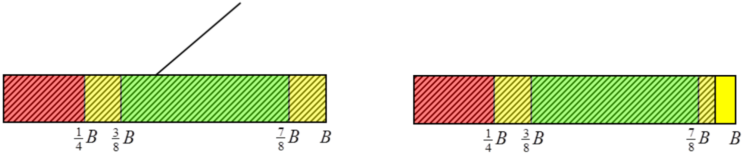
Persistent B-tree Insert

- Search for relevant leaf u and insert new element
- If u contains $B+1$ elements: Block overflow
  - Version split:
    Mark u dead and create new node u' with x alive element
  - If $x > 7/8B$: Strong overflow
  - If $x < 3/8B$: Strong underflow
  - If $3/8B \leq x \leq 7/8B$ then recursively persistently update parent(u):
    Delete reference to u (dead ref.) and insert reference to u' (alive ref.)



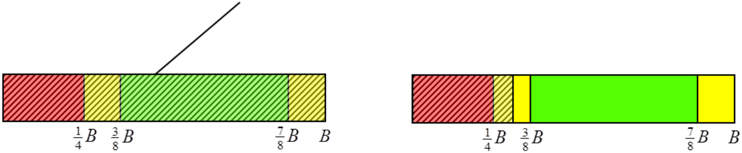
Persistent B-tree Insert

- Search for relevant leaf u and insert new element
- If u contains $B+1$ elements: Block overflow
  - Version split:
    Mark u dead and create new node u' with x alive element
  - If $x > 7/8B$: Strong overflow
  - If $x < 3/8B$: Strong underflow
  - If $3/8B \leq x \leq 7/8B$ then recursively persistently update parent(u):
    Delete reference to u (dead ref.) and insert reference to u' (alive ref.)



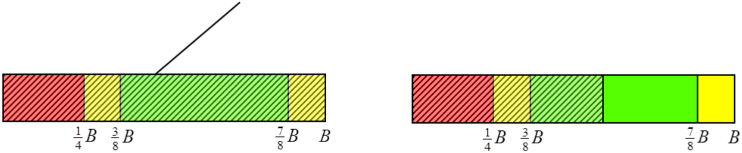
Persistent B-tree Insert

- Search for relevant leaf u and insert new element
- If u contains $B+1$ elements: Block overflow
  - Version split:
    Mark u dead and create new node u' with x alive element
  - If $x > 7/8B$: Strong overflow
  - If $x < 3/8B$: Strong underflow
  - If $3/8B \leq x \leq 7/8B$ then recursively persistently update parent(u):
    Delete reference to u (dead ref.) and insert reference to u' (alive ref.)



Persistent B-tree Insert

- Search for relevant leaf u and insert new element
- If u contains $B+1$ elements: Block overflow
  - Version split:
    Mark u dead and create new node u' with x alive element
  - If $x > 7/8B$: Strong overflow
  - If $x < 3/8B$: Strong underflow
  - If $3/8B \leq x \leq 7/8B$ then recursively persistently update parent(u):
    Delete reference to u (dead ref.) and insert reference to u' (alive ref.)



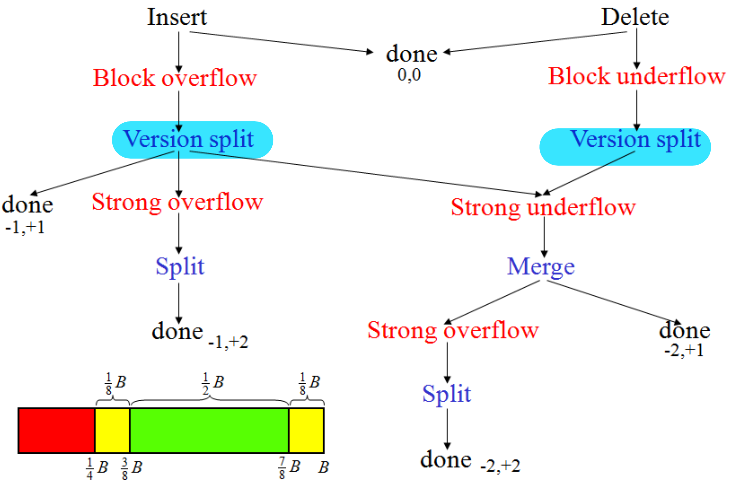
Persistent B-tree Insert

- Strong overflow $(x > 7/8B)$
  - Split u into u' and u'' with $x/2$ elements each $(3/8B \leq x \leq 1/2B)$
  - Recursively update parent(u):
    Delete reference to u and insert reference to u' and u''
- Strong underflow $(x < 3/8B)$:
  - Merge x elements with y live elements obtained by version split on sibling $(1/2B \leq x+y \leq 11/8B)$
  - If $x+y > 7/8B$ then (strong overflow) perform split into nodes with $(x+y)/2$ elements each $(7/16B \leq (x+y)/2 \leq 11/16B)$
  - Recursively update parent(u): Delete two insert one/two references

Persistent B-tree Delete

- Search for relevant leaf u and mark element dead
- If u contains $x < 1/4B$ alive elements: Block underflow
  - Version split
    Mark u dead and create new node u' with x alive element
  - Strong underflow $(x < 3/8B)$:
    Merge (version split) and possibly split (strong overflow)
  - Recursively update parent(u):
    Delete two references and insert one or two references

Persistent B-tree Updates



Persistent B-tree Analysis

- Update: $O(\log_B N)$
  - Search and rebalance on one root-leaf path
- Space: $O(N/B)$
  - At least $1/8B$ updates in leaf in existence interval
  - When leaf u dies
    - At most two other nodes are created
    - At most one block over/underflow one level up (in parent(u))
  - During N updates we create:
    - $O(N/B)$ leaves
    - $O(N/B^i)$ nodes i levels up
  - $\Rightarrow \sum O(N/B^i) = O(N/B)$ blocks

Summary/Conclusion: Persistent B-tree

- Persistent B-tree
  - Update current version
  - Query all versions
- Efficient implementation obtained using existence intervals
  - Standard technique
- During N operations
  - O(N/B) space
  - $O(\log_B N)$ updates
  - $O(\log_B N + T/B)$ query