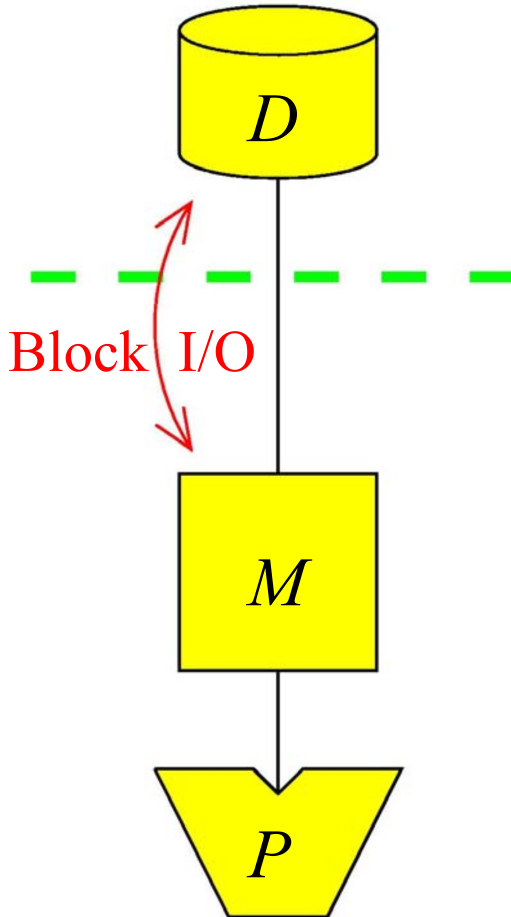
I/O-Algorithms

Lars Arge
Aarhus University

February 21, 2008

I/O-Model



- Parameters

  N = # elements in problem instance

  B = # elements that fits in disk block

  M = # elements that fits in main memory

  K = # output size in searching problem

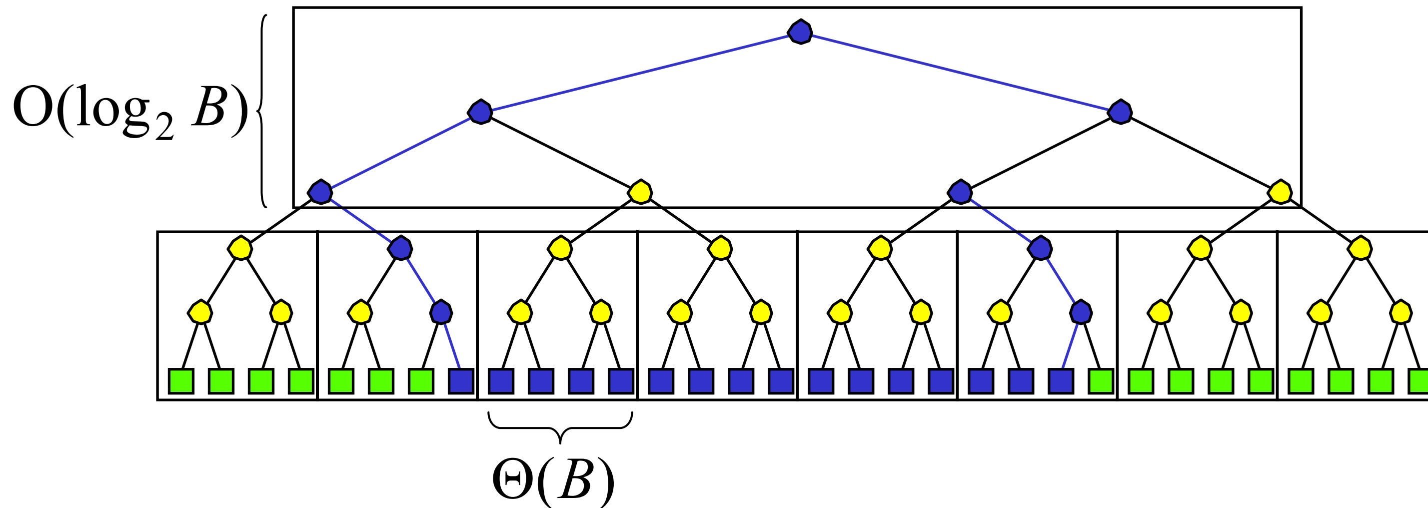
- We often assume that $M>B^2$

- I/O: Movement of block between memory and disk

Fundamental Bounds

	Internal	External
• Scanning:	N	$\frac{N}{B}$
• Sorting:	$N \log N$	$\frac{N}{B} \log_{M/B} \frac{N}{B}$
• Permuting	N	$\min\{N, \frac{N}{B} \log_{M/B} \frac{N}{B}\}$
• Searching:	$\log_2 N$	$\log_B N$

External Search Trees



- BFS blocking:
  - Block height $O(\log_2 N) / O(\log_2 B) = O(\log_B N)$
  - Output elements blocked

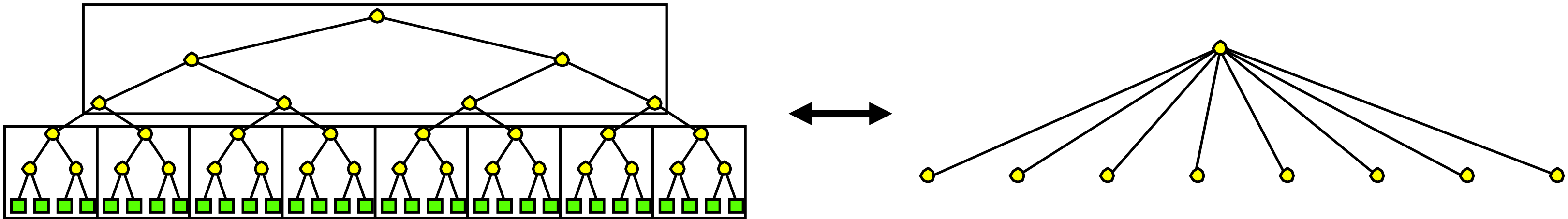
$\Downarrow$

Rangesearch in $O(\log_B N + T/B)$ I/Os

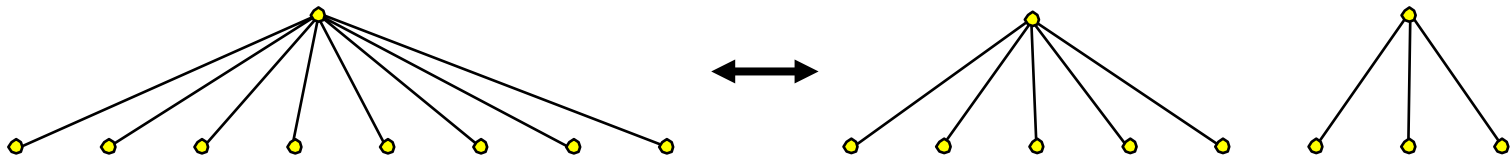
- Optimal: $O(N/B)$ space and $O(\log_B N + T/B)$ query

B-trees

- Seems very difficult to maintain BFS blocking during rotation
- BFS-blocking naturally corresponds to tree with fan-out $\Theta(B)$

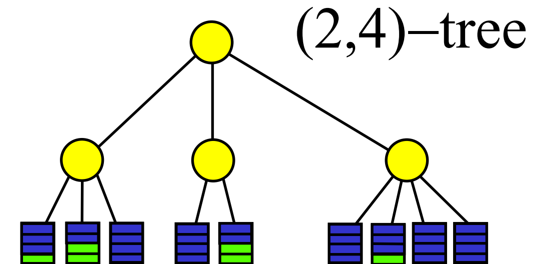


- B-trees balanced by allowing node degree to vary
  - Rebalancing performed by splitting and merging nodes



(a,b)-tree

- T is an (a,b)-tree ($a \geq 2$ and $b \geq 2a-1$)
  - All leaves on the same level (contain between a and b elements)
  - Except for the root, all nodes have degree between a and b
  - Root has degree between 2 and b

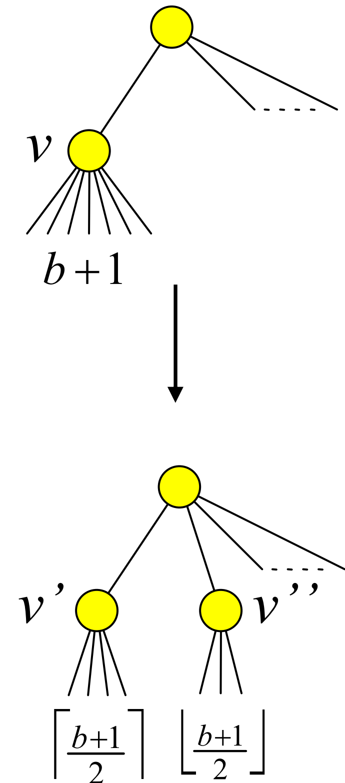


- (a,b)-tree uses linear space and has height $O(\log_a N)$

(*a*,*b*)-Tree Insert

- Insert:

> Search and insert element in leaf *v*
> DO *v* has *b+1* elements/children
>   Split *v*:
>   make nodes *v'* and *v''* with $\left\lceil \frac{b+1}{2} \right\rceil \leq b$ and $\left\lfloor \frac{b+1}{2} \right\rfloor \geq a$ elements
>   insert element (ref) in *parent(v)*
>   (make new root if necessary)
>   *v=parent(v)*

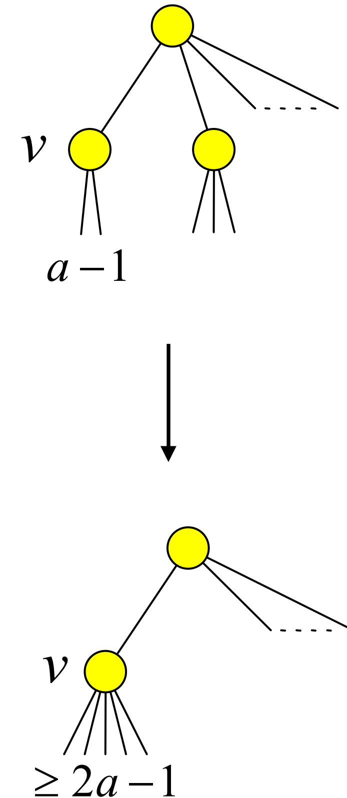


- Insert touch $O(\log_a N)$ nodes

(*a*,*b*)-Tree Delete

- Delete:

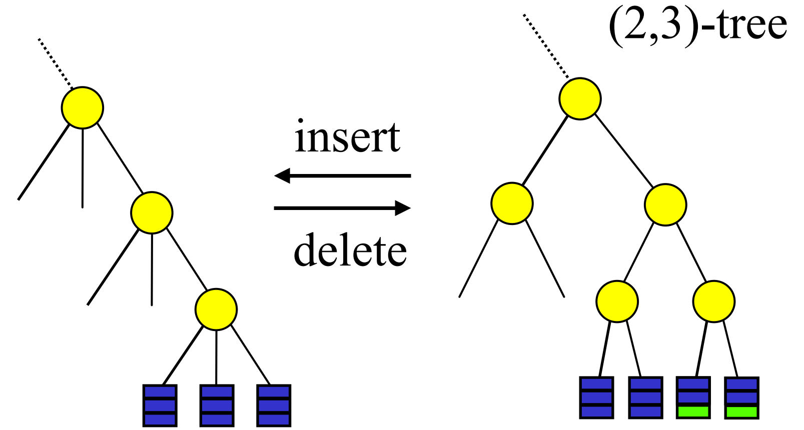
Search and delete element from leaf v
DO v has a-1 elements/children
Fuse v with sibling v':
move children of v' to v
delete element (ref) from *parent(v)*
(delete root if necessary)
If v has $>b$ (and $\leq a+b-1<2b$) children split v
v=*parent(v)*



- Delete touch $O(\log_a N)$ nodes

(*a*,*b*)-Tree

- (a,b)-tree properties:
  - If $b=2a-1$ every update can cause many rebalancing operations



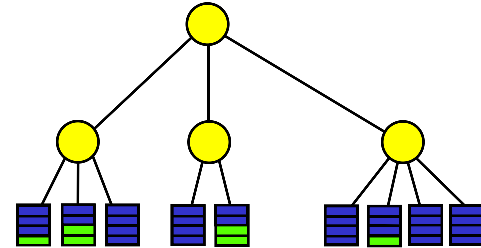
  - If $b \geq 2a$ update only cause $O(1)$ rebalancing operations amortized
  - If $b>2a$ $O(\frac{1}{b/2-a}) = O(1/a)$ rebalancing operations amortized
    * Both somewhat hard to show
  - If $b=4a$ easy to show that update causes $O(\frac{1}{a}\log_a N)$ rebalance operations amortized

Summary/Conclusion: B-tree

- B-trees: (a,b)-trees with $a,b = \Theta(B)$
  - $O(N/B)$ space
  - $O(\log_B N + T/B)$ query
  - $O(\log_B N)$ update
- B-trees with elements in the leaves sometimes called B^+-tree
- Construction in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os
  - Sort elements and construct leaves
  - Build tree level-by-level bottom-up

Summary/Conclusion: B-tree

- B-tree with branching parameter *b* and leaf parameter *k* ($b,k \geq 8$)
  - All leaves on same level and contain between *1/4k* and *k* elements
  - Except for the root, all nodes have degree between *1/4b* and *b*
  - Root has degree between 2 and *b*



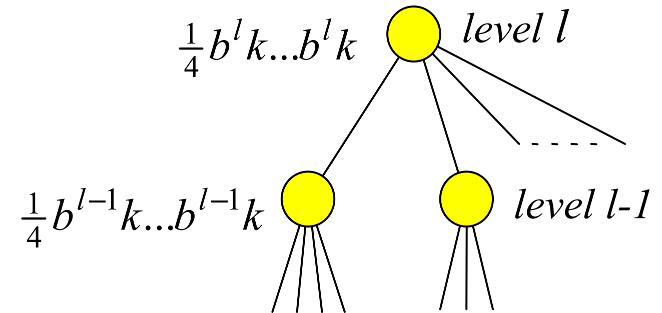
- B-tree with leaf parameter $k = \Omega(B)$
  - *O(N/B)* space
  - Height $O(\log_b \frac{N}{B})$
  - $O(1/k)$ amortized leaf rebalance operations
  - $O(\frac{1}{b \cdot k} \log_b \frac{N}{B})$ amortized internal node rebalance operations
- B-tree with branching parameter B^c, $0<c\leq 1$, and leaf parameter *B*
  - Space *O(N/B)*, updates $O(\log_B N)$, queries $O(\log_B N + T/B)$

Secondary Structures

- When secondary structures used, a rebalance on v often require $O(w(v))$ 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
- In internal memory BB[α]-trees have the desired property
  - But rebalanced using rotations

Weight-balanced B-tree

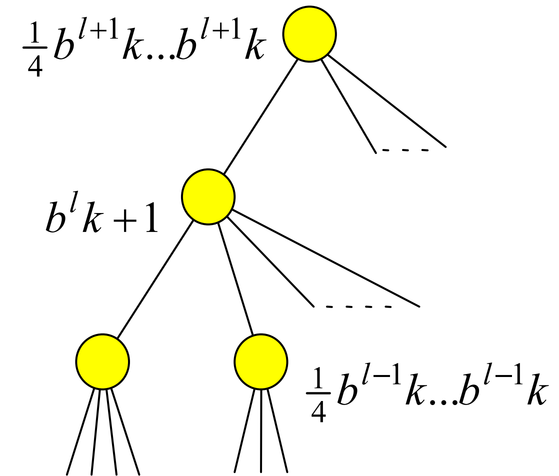
- Idea: Combination of B-tree and BB[α]-tree
  - Weight constraint on nodes instead of degree constraint
  - Rebalancing performed using split/fuse as in B-tree
- Weight-balanced B-tree with parameters b and k ($b>8$, $k\geq 8$)
  - All leaves on same level and contain between $k/4$ and k elements
 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



- Internal node degree between $\frac{1}{4} b^l k / b^{l-1} k = \frac{1}{4} b$ and $b^l k / \frac{1}{4} b^{l-1} k = 4b$

Weight-balanced B-tree Insert

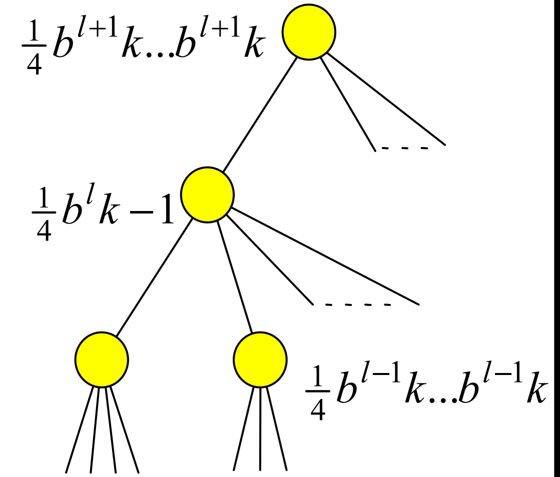
- Search for relevant leaf *u* and insert new element
- Traverse path from *u* to root:
  - If level *l* node *v* now has $w(v)=b^l k+1$ then split into nodes *v'* and *v''* with $w(v') \geq \lfloor \frac{1}{2}(b^l k+1) \rfloor - b^{l-1}k$ and $w(v'') \leq \lceil \frac{1}{2}(b^l k+1) \rceil + b^{l-1}k$



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

Weight-balanced B-tree Delete

- Search for relevant leaf u and delete element
- Traverse path from u to root:
  - If level l node v now has $w(v)=\frac{1}{4}b^l k-1$ then fuse with sibling into node v' with $\frac{2}{4}b^l k-1\leq w(v')\leq\frac{5}{4}b^l k-1$
  - If now $w(v')\geq\frac{7}{8}b^l k$ then split into nodes with weight $\geq\frac{7}{16}b^l k-1-b^{l-1}k\geq\frac{5}{16}b^l k-1$ and $\leq\frac{5}{8}b^l k+b^{l-1}k\leq\frac{6}{8}b^l k$



- Algorithm correct and touch $O(\log_b \frac{N}{k})$ 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 $O(\log_b \frac{N}{k})$
  - $O(\log_b N)$ 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 $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/O

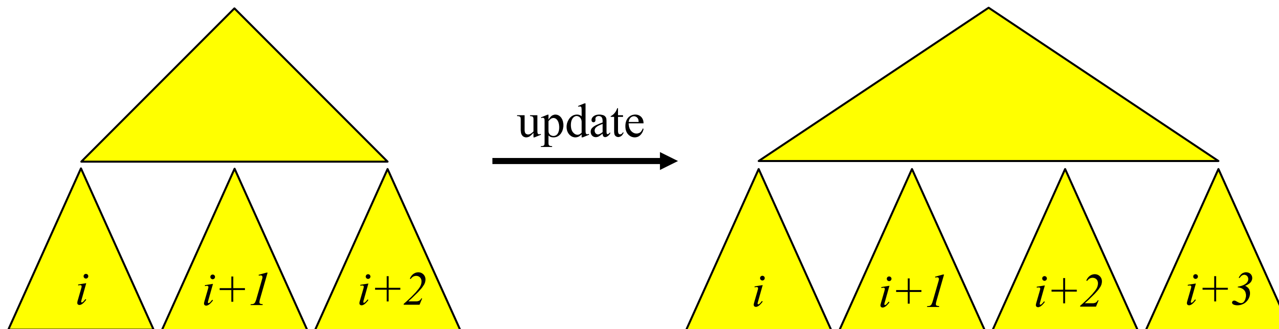
Persistent B-tree

- 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

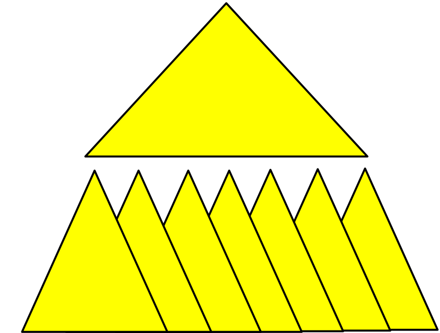
- Easy way to make B-tree partial persistent
  - Copy structure at each operation
  - Maintain "version-access" structure (B-tree)



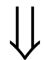
- Good $O(\log_B N + T/B)$ query in any version, but
  - $O(N/B)$ I/O update
  - $O(N^2/B)$ space

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

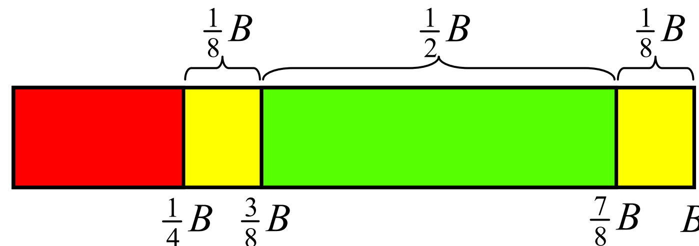


If $b=B$:

- Query at any time t in $O(\log_B N + T/B)$ I/Os

Persistent B-tree: Updates

- Updates performed as in B-tree
- To obtain linear space we maintain new-node invariant:
  - New node contains between $\frac{3}{8}B$ and $\frac{7}{8}B$ alive elements and no dead elements

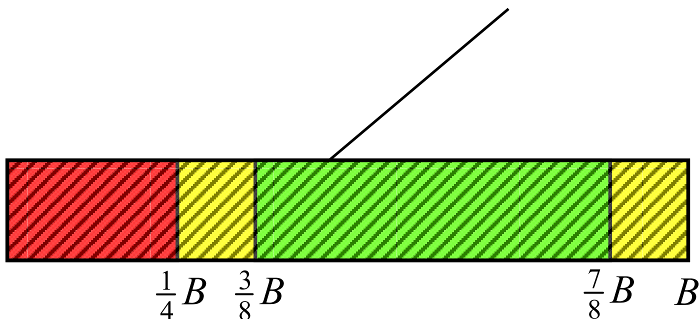


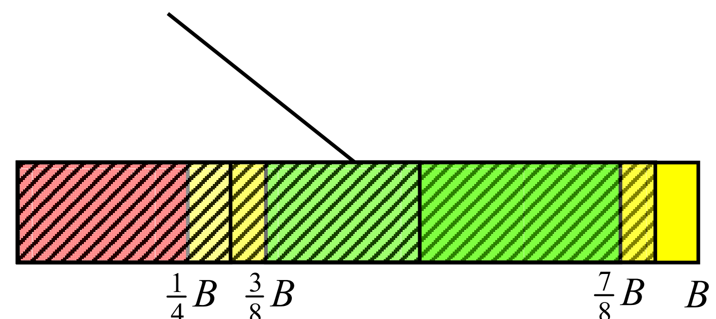
Persistent B-tree Insert

- 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 $x > \frac{7}{8}B$: Strong overflow
  - If $x < \frac{3}{8}B$: Strong underflow
  - If $\frac{3}{8}B \leq x \leq \frac{7}{8}B$ then recursively update $parent(l)$:

    Delete reference to u and insert reference to u'

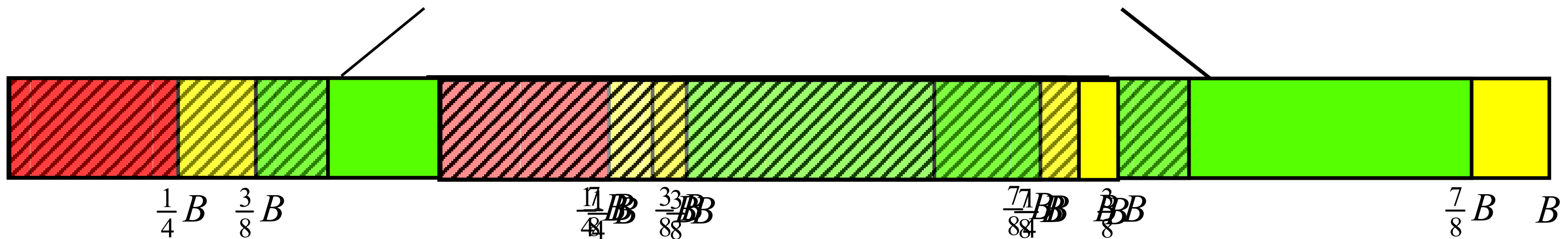




Persistent B-tree Insert

- Strong overflow ($x > \frac{7}{8}B$)
  - Split v into u' and u'' with $\frac{x}{2}$ elements each ($\frac{3}{8}B < \frac{x}{2} \leq \frac{1}{2}B$)
  - Recursively update *parent*(u):

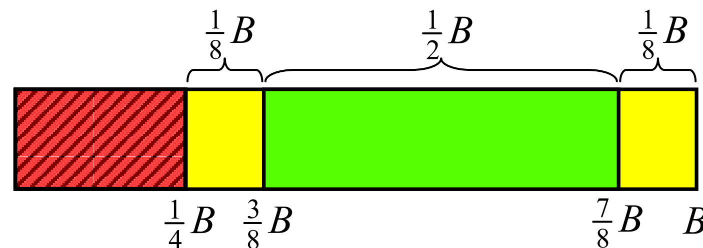
    Delete reference to l and insert reference to v' and v''



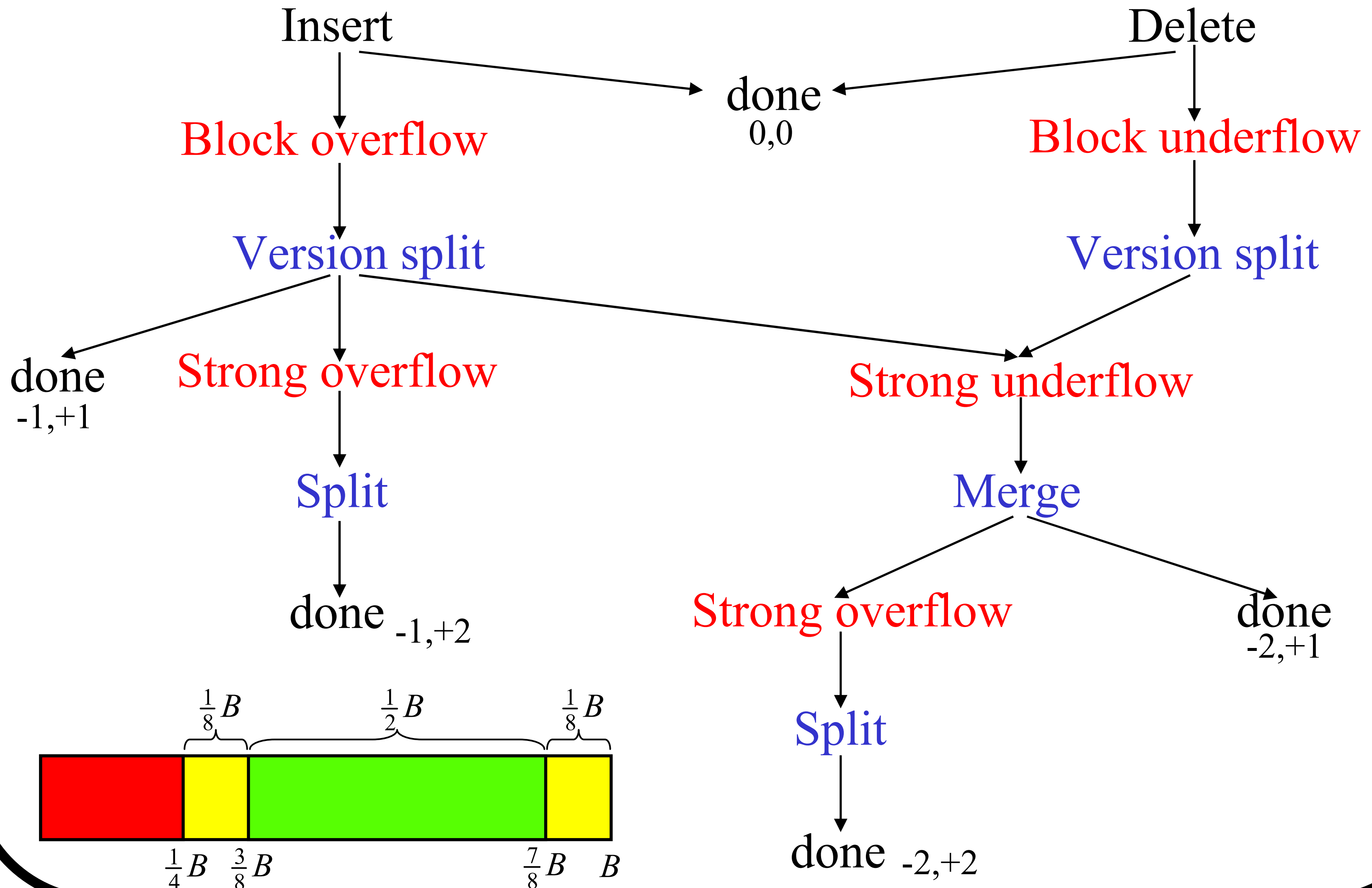
- Strong underflow ($x < \frac{3}{8}B$)
  - Merge x elements with y live elements obtained by version split on sibling ($\frac{1}{2}B \leq x + y \leq \frac{11}{8}B$)
  - If $x + y \geq \frac{7}{8}B$ then (strong overflow) perform split into nodes with $(x+y)/2$ elements each ($\frac{7}{16}B \leq (x+y)/2 \geq \frac{11}{16}B$)
  - 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 $x < \frac{1}{4}B$ alive elements: Block underflow
  - Version split:
    Mark u dead and create new node u' with x alive element
  - Strong underflow ($x < \frac{3}{8}B$):
    Merge (version split) and possibly split (strong overflow)
  - Recursively update *parent*(u):
    Delete two references insert one or two references

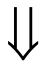


Persistent B-tree



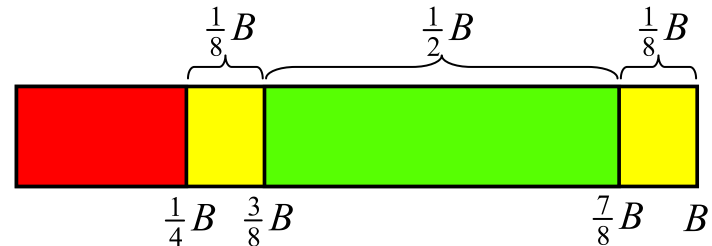
Persistent B-tree Analysis

- Update: $O(\log_B N)$
  - Search and "rebalance" on one root-leaf path
- Space: $O(N/B)$
  - At least $\frac{1}{8}B$ 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*(l))



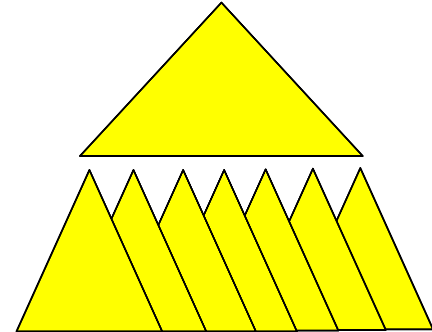
  - During N updates we create:
    - $O(N/B)$ leaves
    - $O(N/B^i)$ nodes i levels up

$\Rightarrow \sum_i O(N/B^i) = O(N/B)$ blocks

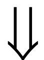


Summary/Conclusion: Persistent B-tree

- Persistent B-tree
  - Update current version
  - Query all versions



- Efficient implementation obtained using existence intervals
  - Standard technique



- During N operations
  - $O(N/B)$ space
  - $O(\log_B N)$ update
  - $O(\log_B N + T/B)$ query

Other B-tree Variants

- Level-balanced B-trees
  - Global instead of local balancing strategy
  - Whole subtrees rebuilt when too many nodes on a level
  - Used when parent pointers and divide/merge operations needed
- String B-trees
  - Used to maintain and search (variable length) strings

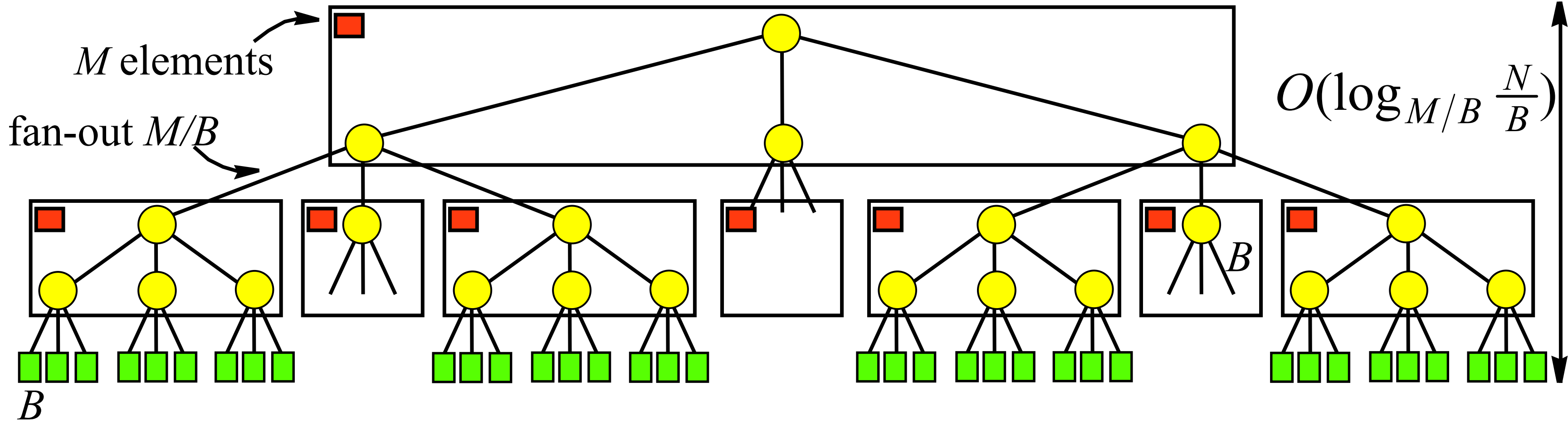
B-tree Construction

- In internal memory we can sort N elements in $O(N \log N)$ time using a balanced search tree:
  - Insert all elements one-by-one (construct tree)
  - Output in sorted order using in-order traversal

- Same algorithm using B-tree use $O(N \log_B N)$ I/Os
  - A factor of $O(B \frac{\log \frac{M}{B}}{\log B})$ non-optimal

- As discussed we could build B-tree bottom-up in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os
  - But what about persistent B-tree?
  - In general we would like to have dynamic data structure to use in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ algorithms $\Rightarrow O(\frac{1}{B} \log_{M/B} \frac{N}{B})$ I/O operations

Buffer-tree Technique



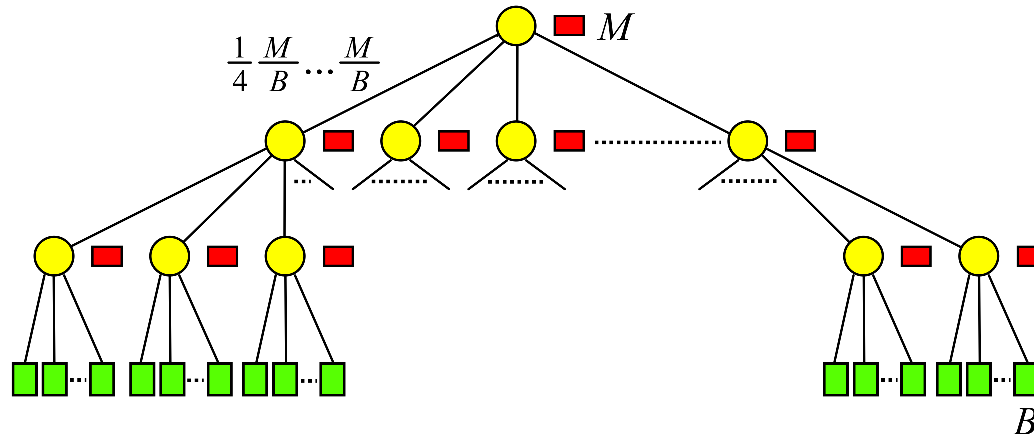
- Main idea: Logically group nodes together and add buffers
  - Insertions done in a "lazy" way – elements inserted in buffers.
  - When a buffer runs full elements are pushed one level down.
  - Buffer-emptying in $O(M/B)$ I/Os

  $\Rightarrow$ every *block* touched constant number of times on each level

  $\Rightarrow$ inserting N elements (N/B blocks) costs $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os.

Basic Buffer-tree

- Definition:
  - B-tree with branching parameter $\frac{M}{B}$ and leaf parameter B
  - Size M buffer in each internal node

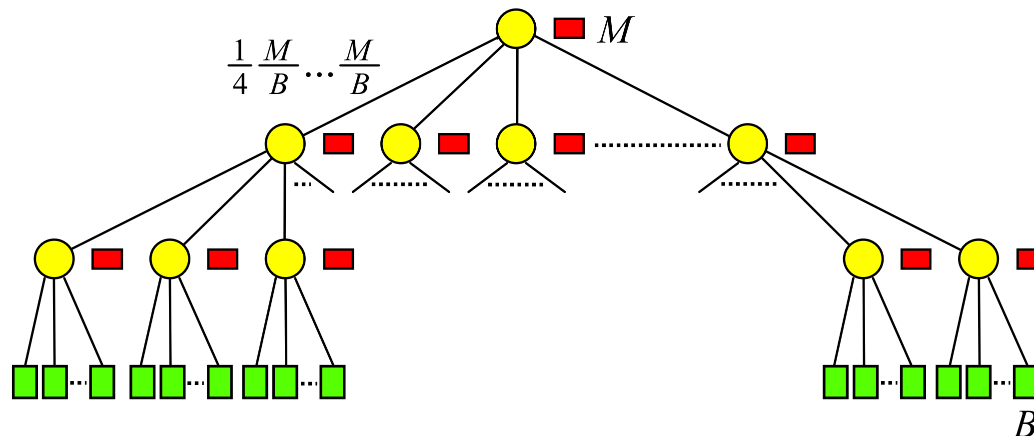


- Updates:
  - Add time-stamp to insert/delete element
  - Collect B elements in memory before inserting in root buffer
  - Perform buffer-emptying when buffer runs full

Basic Buffer-tree

- Note:
  - Buffer can be larger than M during recursive buffer-emptying
    - Elements distributed in sorted order

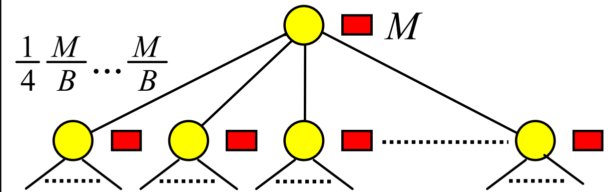
      $\Rightarrow$ at most M elements in buffer unsorted
  - Rebalancing needed when "leaf-node" buffer emptied
    - Leaf-node buffer-emptying only performed after all full internal node buffers are emptied



Basic Buffer-tree

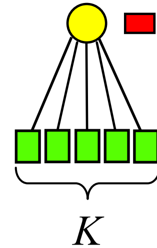
- Internal node buffer-empty:

  - Load first M (unsorted) elements into memory and sort them
  - Merge elements in memory with rest of (already sorted) elements
  - Scan through sorted list while
    - Removing "matching" insert/deletes
    - Distribute elements to child buffers
  - Recursively empty full child buffers



- Emptying buffer of size X takes $O(X/B+M/B)$=O(X/B) I/Os

Basic Buffer-tree

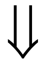


- Buffer-empty of leaf node with K elements in leaves
  - Sort buffer as previously
  - Merge buffer elements with elements in leaves
  - Remove "matching" insert/deletes obtaining K' elements
  - If $K' < K$ then
    * Add $K-K'$ "dummy" elements and insert in "dummy" leaves

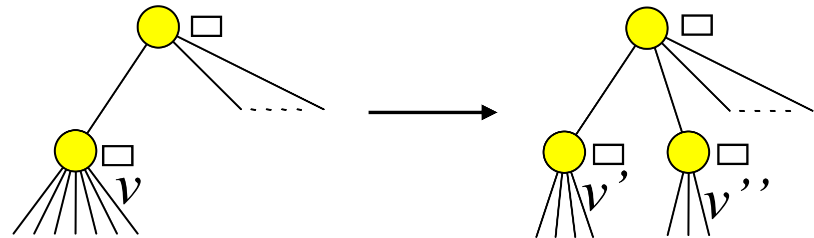
    Otherwise
    * Place K elements in leaves
    * Repeatedly insert block of elements in leaves and rebalance
- Delete dummy leaves and rebalance when all full buffers emptied

Basic Buffer-tree

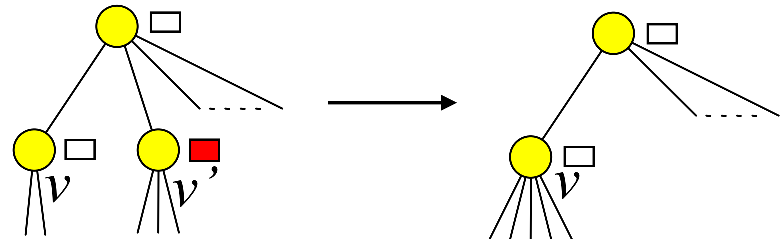
- Invariant:
  Buffers of nodes on path from root to emptied leaf-node are empty



- Insert rebalancing (splits) performed as in normal B-tree



- Delete rebalancing: v' buffer emptied before fuse of v
  - Necessary buffer emptyings performed before next dummy-block delete
  - Invariant maintained



Basic Buffer-tree

- Analysis:
  - Not counting rebalancing, a buffer-emptying of node with $X \geq M$ elements (full) takes $O(X/B)$ I/Os

    $\Rightarrow$ total full node emptying cost $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os
  - Delete rebalancing buffer-emptying (non-full) takes $O(M/B)$ I/Os

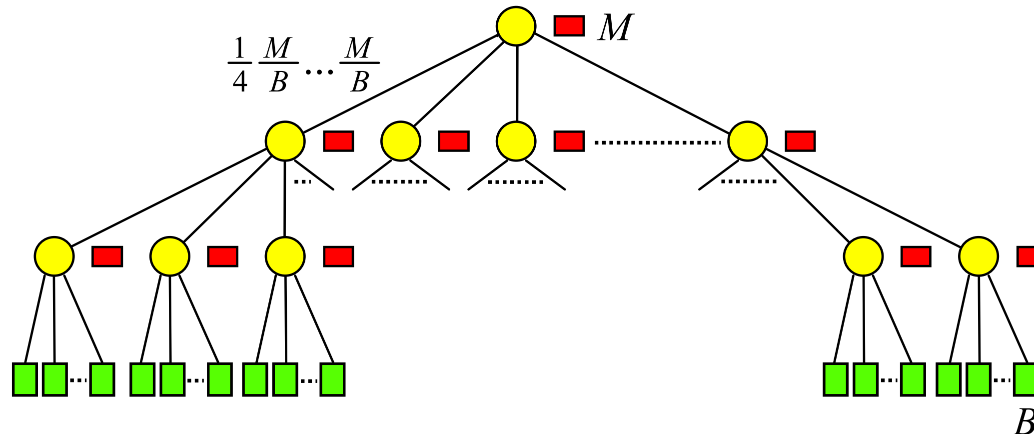
    $\Rightarrow$ cost of one split/fuse $O(M/B)$ I/Os
  - During N updates
    * $O(N/B)$ leaf split/fuse
    * $O(\frac{N/B}{M/B}\log_{M/B}\frac{N}{B})$ internal node split/fuse

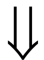
  $\Downarrow$

  Total cost of N operations: $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os

Basic Buffer-tree

- Emptying all buffers after N insertions:
  Perform buffer-emptying on all nodes in BFS-order
  $\Rightarrow$ resulting full-buffer emptyings cost $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os
  empty $O(\frac{N/B}{M/B})$ non-full buffers using $O(M/B) \Rightarrow O(N/B)$ I/Os

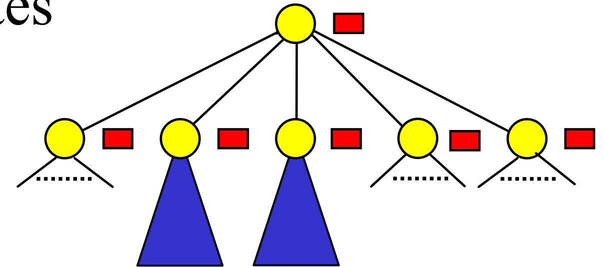




- N elements can be sorted using buffer tree in $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os

Summary/Conclusion: Buffer-tree

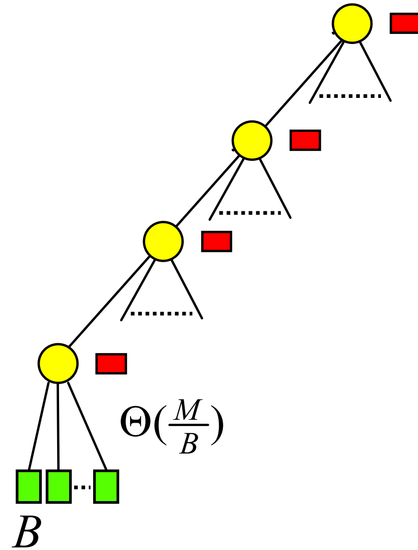
- Batching of operations on B-tree using M-sized buffers
  - $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ I/O updates amortized
  - All buffers emptied in $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os
- One-dim. rangesearch operations can also be supported in $O(\frac{1}{B}\log_{M/B}\frac{N}{B}+\frac{T}{B})$ I/Os amortized
  - Search elements handle lazily like updates
  - All elements in relevant sub-trees reported during buffer-emptying
  - Buffer-emptying in $O(X/B+T'/B)$, where T' is reported elements



- Using buffer technique persistent B-tree built in $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/O

Buffered Priority Queue

- Basic buffer tree can be used in external priority queue
- To delete minimal element:
  - Empty all buffers on leftmost path
  - Delete $\frac{1}{4}M$ elements in leftmost leaf and keep in memory
  - Deletion of next M minimal elements free
  - Inserted elements checked against minimal elements in memory



- $O(\frac{M}{B}\log_{M/B}\frac{N}{B})$ I/Os every $O(M)$ delete $\Rightarrow O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ amortized

Other External Priority Queues

- Buffer technique can be used on other priority queue structures
  - Heap
  - Tournament tree
- Priority queue supporting update often used in graph algorithms
  - $O(\frac{1}{B}\log_2 \frac{N}{B})$ on tournament tree
  - Major open problem to do it in $O(\frac{1}{B}\log_{M/B} \frac{N}{B})$ I/Os
- Worst case efficient priority queue has also been developed
  - B operations require $O(\log_{M/B} \frac{N}{B})$ I/Os

Other Buffer-tree Technique Results

- Attaching $\Theta(B)$ size buffers to normal B-tree can also be use to improve update bound
- Buffered segment tree
  - Has been used in batched range searching and rectangle intersection algorithm
- Has been used on String B-tree to obtain I/O-efficient string sorting algorithms

Summary/Conclusions: Fund. Data Structures

- B-tree
  - $O(N/B)$ space, $O(\log_B N)$ update, $O(\log_B N+T/B)$ query
- Weight-balanced B-tree
  - $\Omega(w(v))$ updates below v between consecutive operations on v
- Persistent B-tree
  - Query in any previous version
- Buffer tree
  - Batching of operations to obtain $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ bounds

References

- **External Memory Geometric Data Structures**
  Lecture notes by Lars Arge.
  - Section 4-5