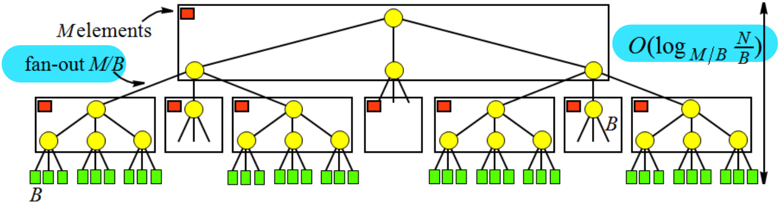
Massive Data Algorithmics

Lecture 5: External Search Trees

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

Buffer-tree Technique



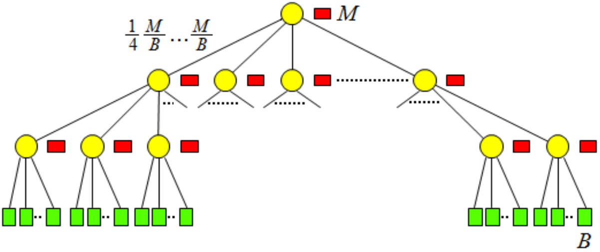
- Main idea: Logically group nodes together and add buffers
  - Insertions done in a lazy way elements inserted in buffers
  - When a buffer runs full elements are pushed one level down
  - Buffer-emptying in $O(M/B)$ I/Os

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

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

Buffer-tree Technique

- Definition:
  - B-tree with branching parameter M/B and leaf parameter B
  - Size M buffer in each internal node



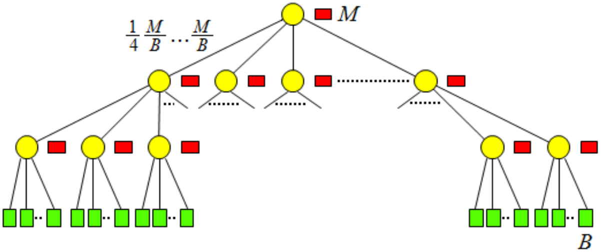
- Updates:
  - Add time-stamp to insert/delete element
  - Collect B elements in memory before inserting in root buffer
  - Perform buffer-emptying when buffer runs full

Buffer-tree Technique

- 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

Buffer-tree Technique

- Note:
  - Buffer can be larger than M during recursive buffer-emptying
    - Buffer can be larger than M during recursive buffer-emptying $\Rightarrow$ 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

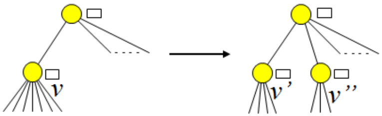


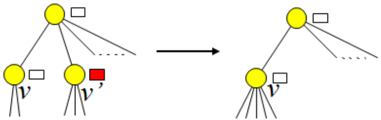
Buffer-tree Technique

- 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

Buffer-tree Technique

- 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



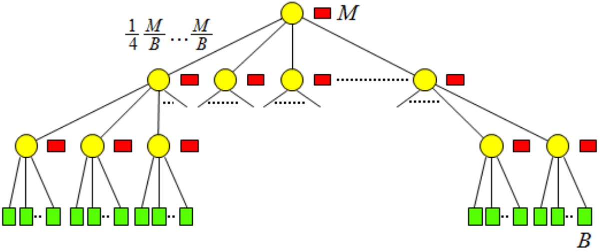


Buffer-tree Technique

- 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 $O(N/B \log_{M/B} N/B)$ 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
    * $I(\frac{N/B}{M/B} \log_{M/B} N/B)$ internal node split/fuse $\Rightarrow$ Total cost of N operations: $O(N/B \log_{M/B} N/B)$ I/Os

Buffer-tree Technique

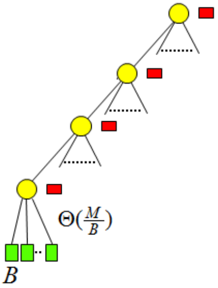
- Emptying all buffers after N insertions:
  - Perform buffer-emptying on all nodes in BFS-order
    $\Rightarrow$ resulting full-buffer emptyings cost $O(N/B \log_{M/B} N/B)$ I/Os
    empty $O(\frac{N/B}{M/B})$ non-full buffers using O(M/B) I/Os $\Rightarrow$ O(N/B) I/Os



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

Buffered Priority Queue

- Basic buffer tree can be used in external priority queue
- To delete minimal element:
  - $O(1/B \log_{M/B} N/B)$ I/O updates amortized
  - All buffers emptied in $O(N/B \log_{M/B} N/B)$ I/Os



- $O(M/B \log_{M/B} N/B)$ I/Os every $O(M)$ delete $\Rightarrow$ $O(1/B \log_{M/B} N/B)$ 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
  - $O(1/B \log_2 N/B)$ on tournament tree
  - Major open problem to do it in $O(1/B \log_{M/B} N/B)$ I/Os
- Worst case efficient priority queue has also been developed
  - B operations require $O(\log_{M/B} N/B)$ I/Os

Summary/Conclusion: Buffer-tree

- Batching of operations on B-tree using M-sized buffers
  - $O(1/B \log_{M/B} N/B)$ I/O updates amortized
  - All buffers emptied in $O(N/B \log_{M/B} N/B)$ I/Os
- Using buffer technique persistent B-tree built in $O(N/B \log_{M/B} N/B)$ I/Os
- Priority Queue with $O(1/B \log_{M/B} N/B)$ I/Os amortized update