A Lock-free Priority Queue Design Based on Multi-dimensional Linked Lists

Deli Zhang Damian Dechev

University of Central Florida

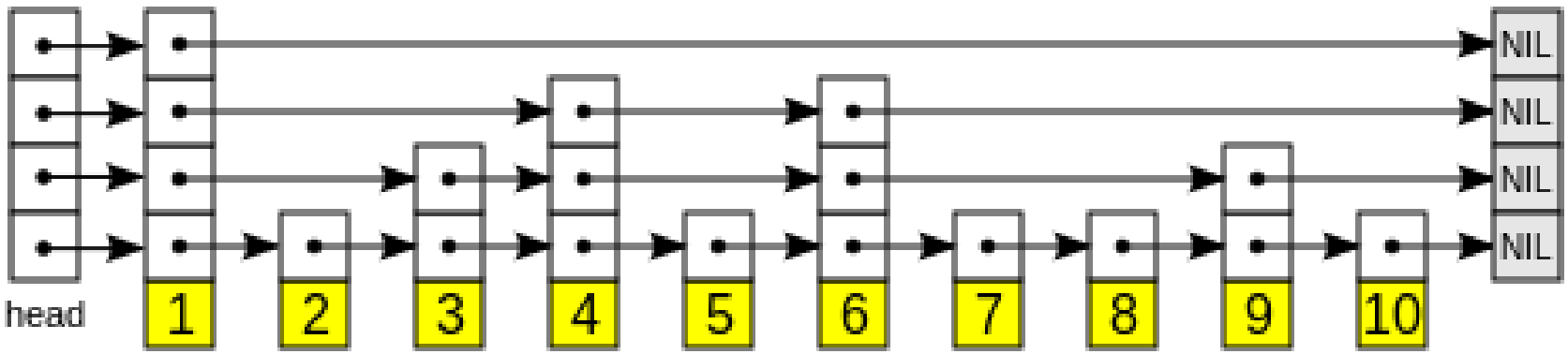
September 9, 2014

Priority Queues

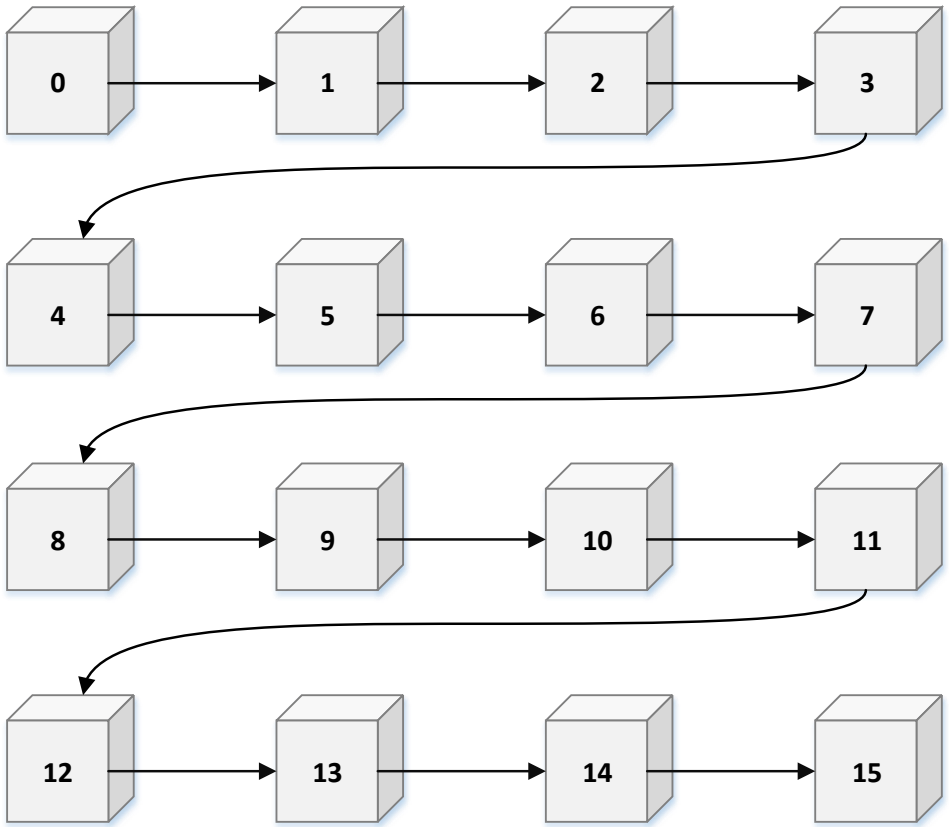
- Abstract Data Structure
  - `Insert` adds a key-value pair into the queue sorted by keys
  - `DeleteMin` returns and removes the first key-value pair
- Typical Sequential Implementations
  - Balanced Search Trees
  - Array-based Binary Heap

Concurrent Priority Queues

- Skip-list
  - Use randomization to avoid global balancing
  - Keep redundant short-cut lists for fast search
- Skip-list based approaches
  - Sundell et al. 2005: first lock-free linearizable implementation
  - Shavit et al. 2009: lock-free and quiescently consistent
  - Linden et al 2013: batch physical deletion to reduce contention

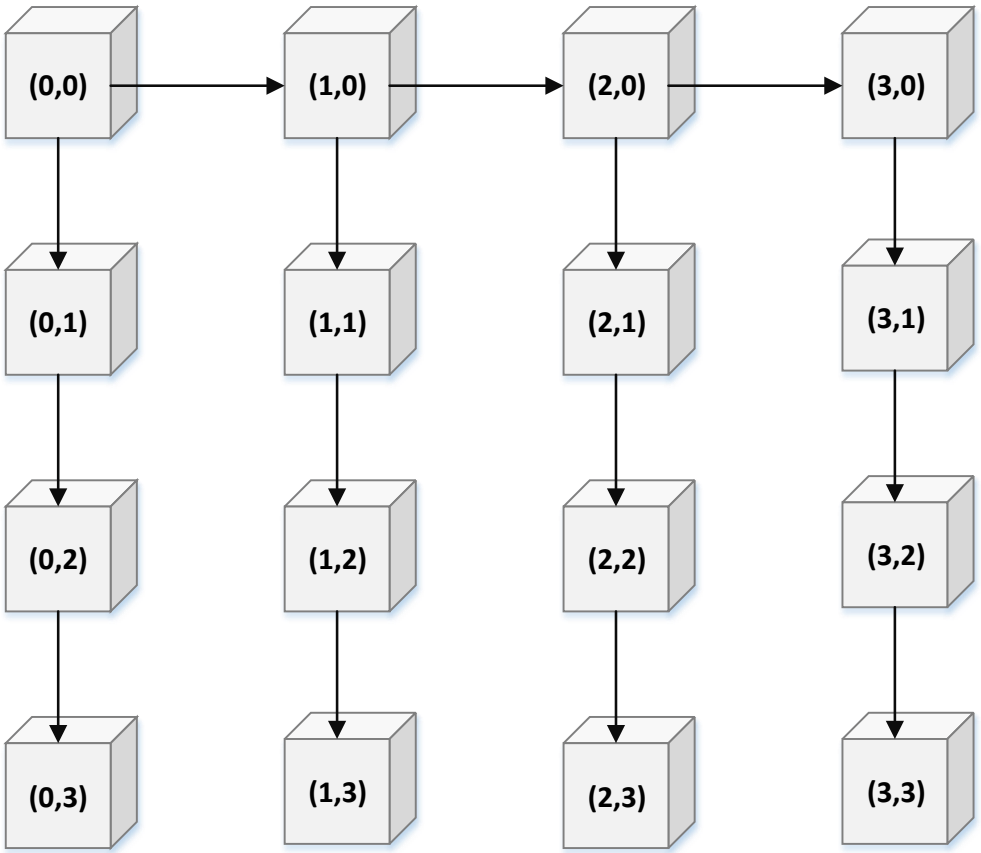


Ordered Linked List



- $\mathcal{O}(n)$ search/insertion
- $\mathcal{O}(1)$ deletion

Ordered 2-D List



- Vector (d_0, d_1) serves as insertion coordinates
- $\mathcal{O}(\sqrt{n})$ search/insertion
- $\mathcal{O}(1)$ deletion

Multi-dimensional List

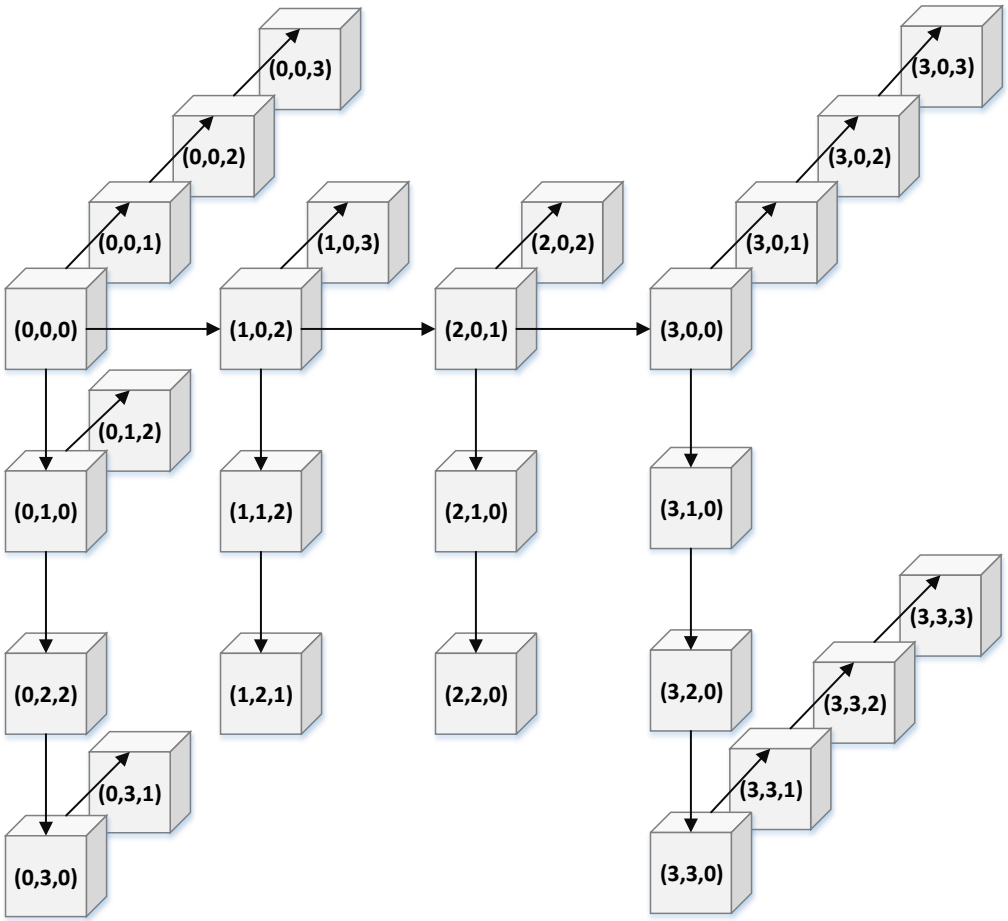
Definition

A D-dimensional list is a rooted tree in which each node is implicitly assigned a dimension of $d \in [0, D)$. The root node's dimension is 0. A node of dimension d has no more than $D - d$ children, where the mth child is assigned a dimension of $d' = d + m - 1$.

Definition

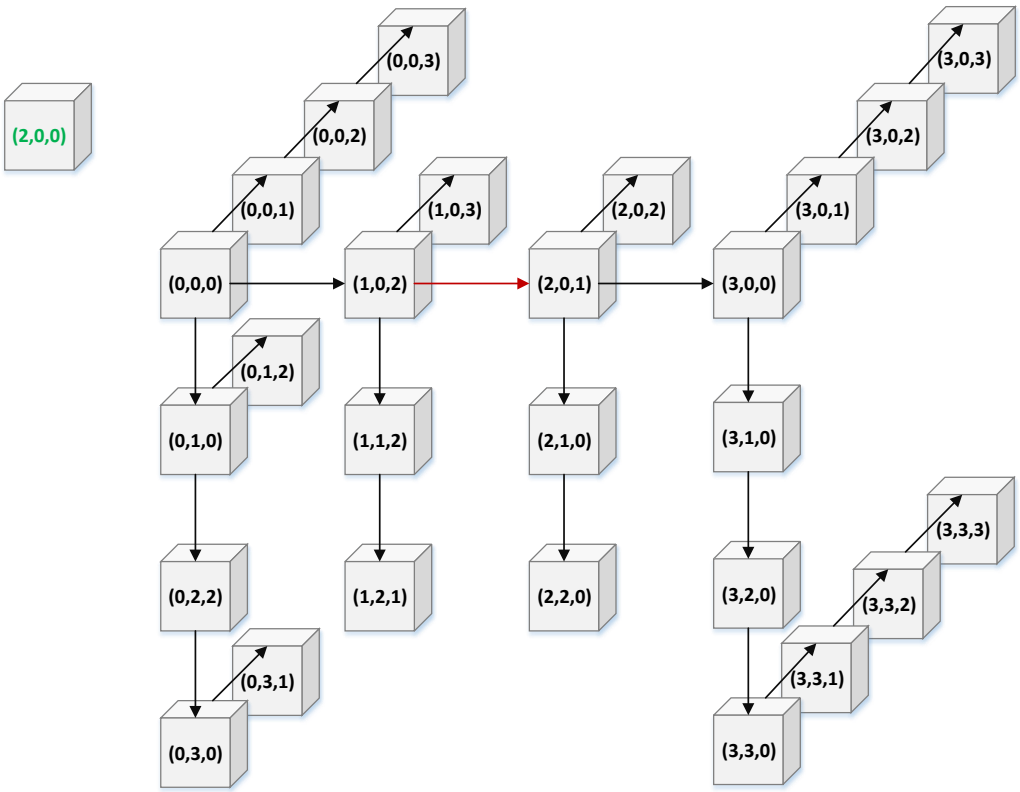
Given a non-root node of dimension d with coordinate $\mathbf{k} = (k_0, ..., k_{D-1})$ and its parent with coordinate $\mathbf{k'} = (k'_0, ..., k'_{D-1})$ in an ordered D-dimensional list: $k_i = k'_i, \ \forall \ i \in [0, d) \wedge k_d > k'_d$.

Ordered 3-D List



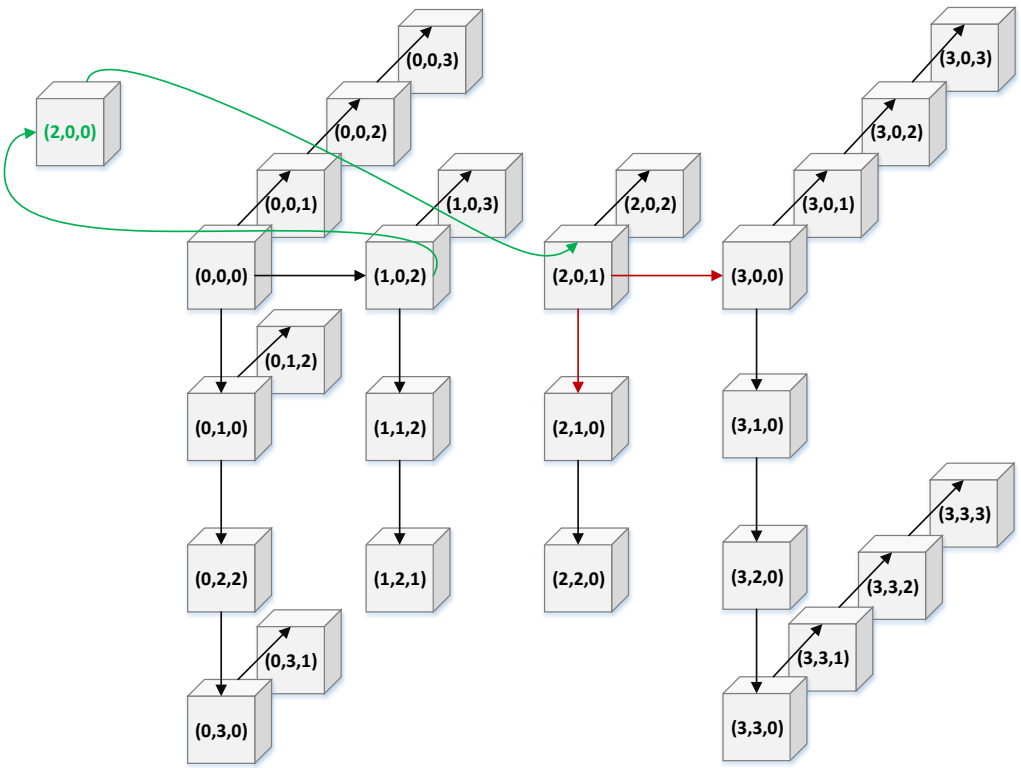
- Worst-case $\mathcal{O}(D \sqrt[D]{n})$ search/insertion
- Choose $D = \log n$ then $\log n \sqrt[\log n]{n} = \mathcal{O}(\log n)$
- No need for global re-balancing/randomization

Locate Unique Inserting Position



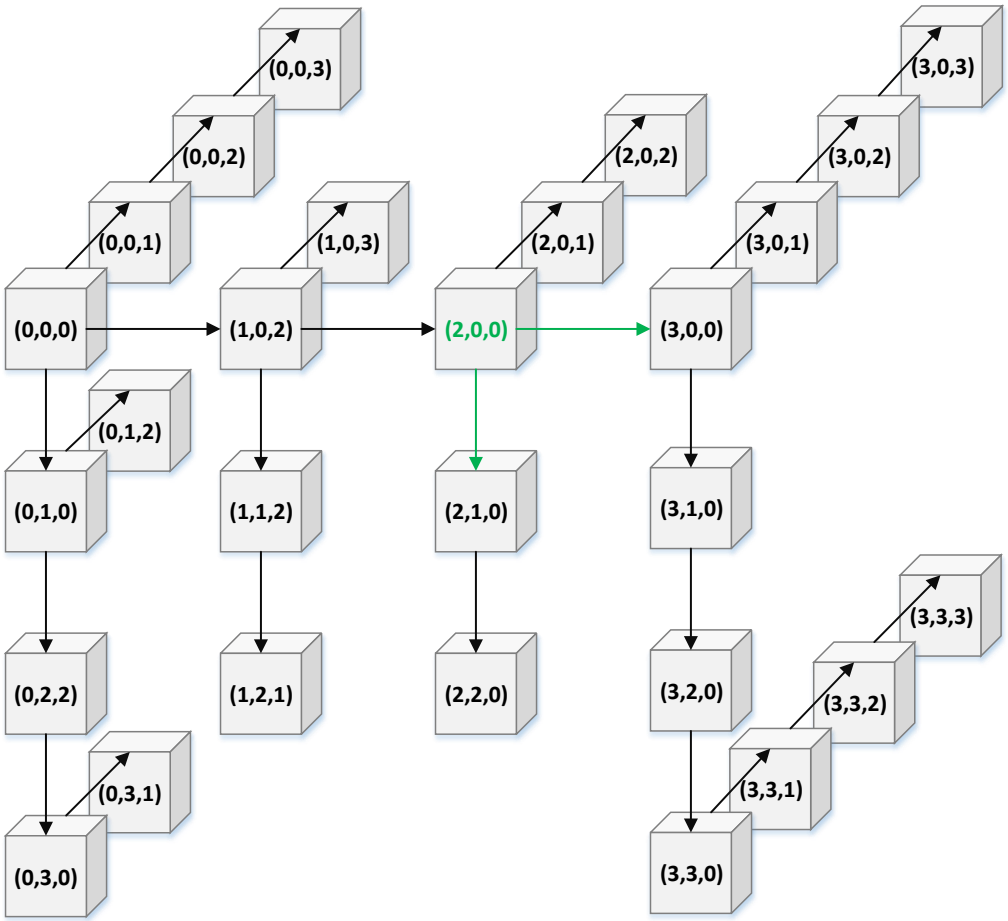
- Compare coordinate vector from $d = 0$
- Increase d if equal
- Go to dth child if greater
- Stop if smaller

Insert



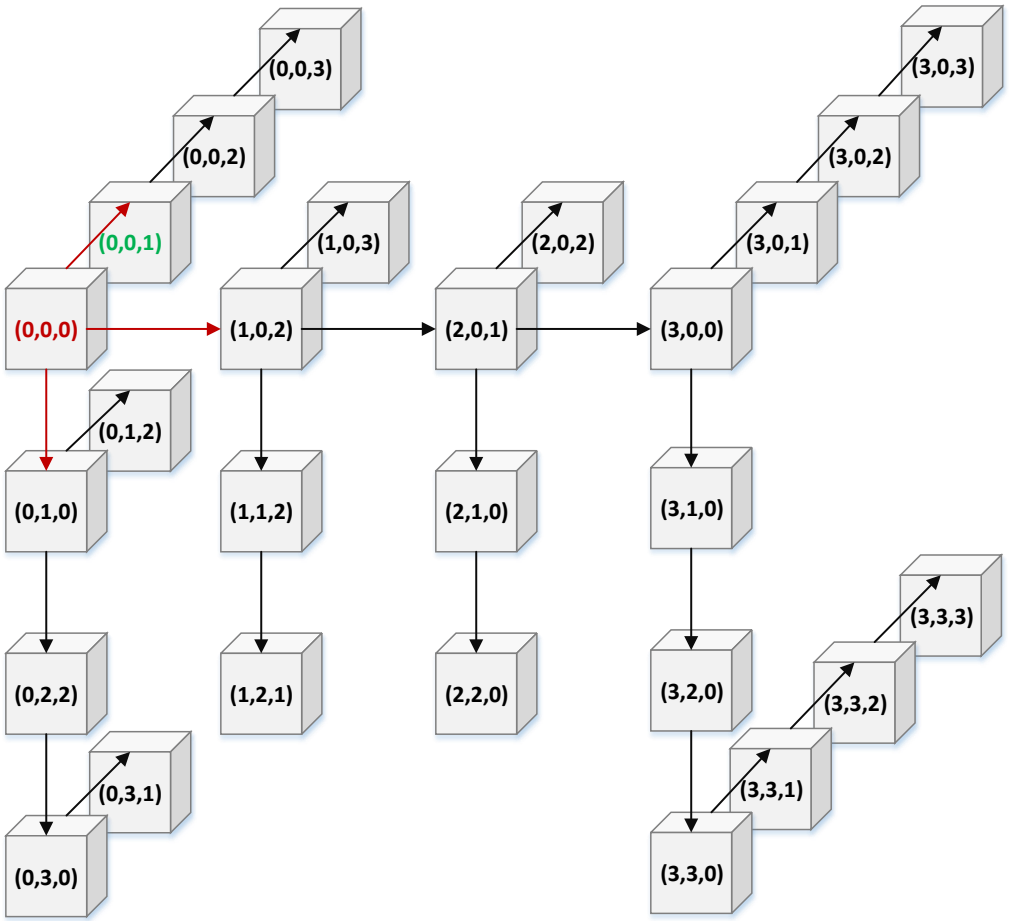
- Point to parent's old child
- Update parent's child pointer
- Adopt old child's children if old child's dimension is changed

Adopting Children



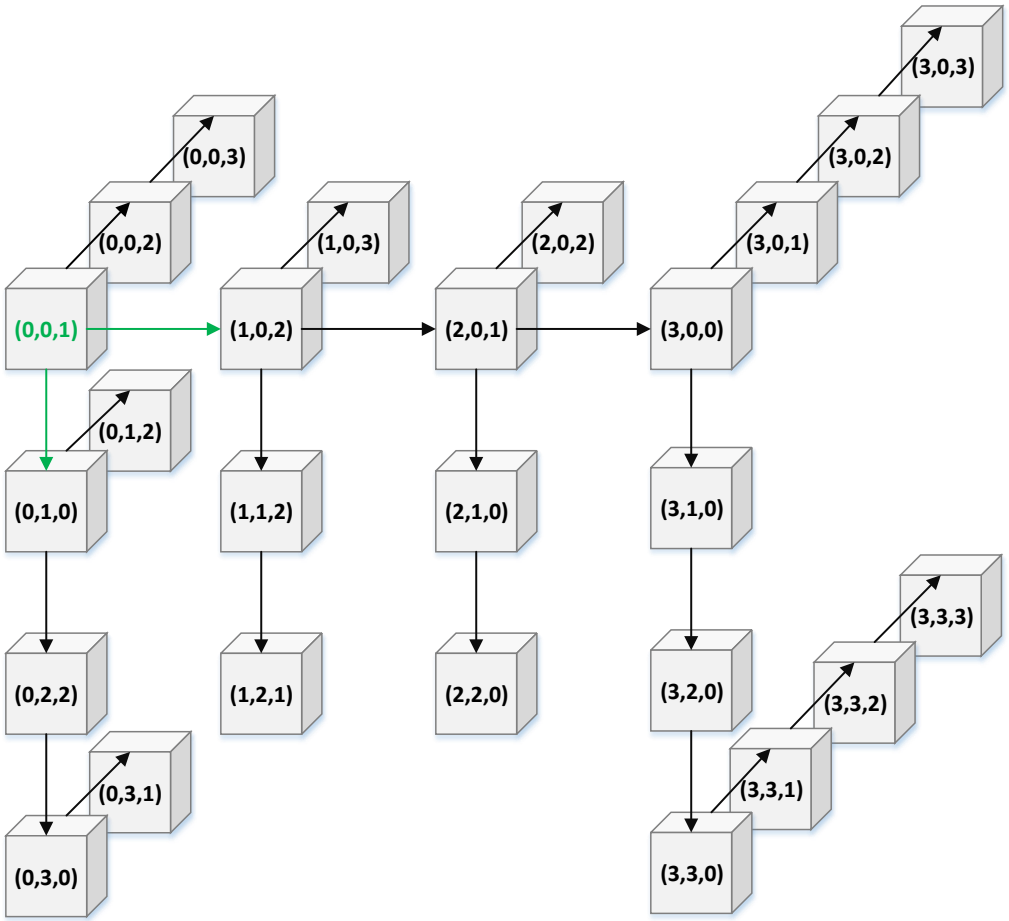
- New node $d = 0$
- Old child $d = 2$
- Old child's children on dimension 0, 1 are transfer to new node

Normal Deletion



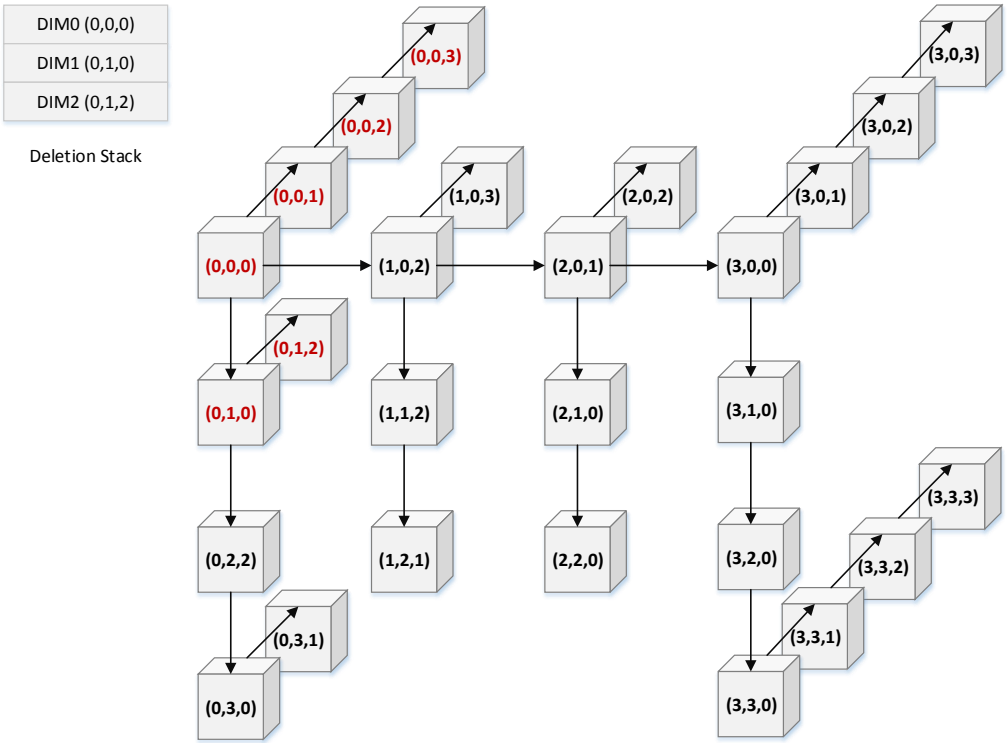
- Prompt the next minimal node
- Preserve two children

Normal Deletion - Promotion



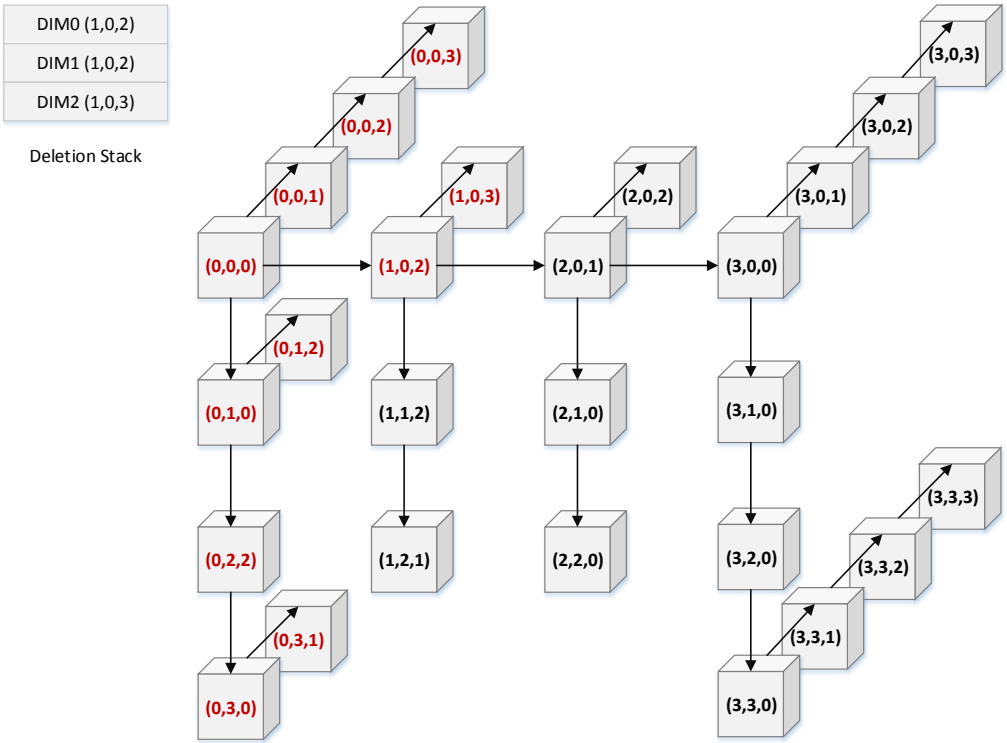
- Transfer them to the new head
- Heavy contention on the head nodes

Stack-based Deletion



- Mark for logical deletion
- DPS to find next node
- Use stack as cache to DPS path

Stack-based Deletion

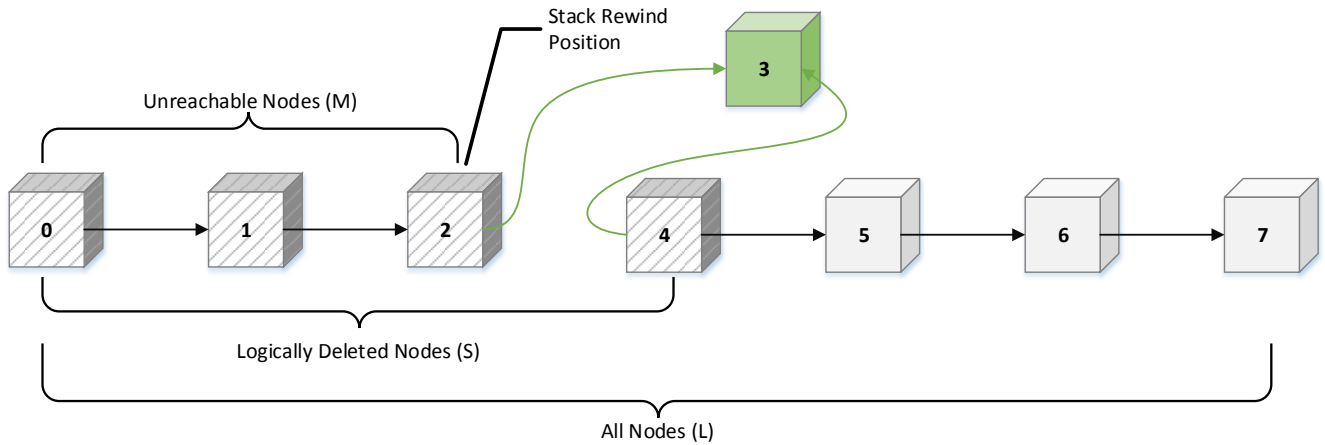


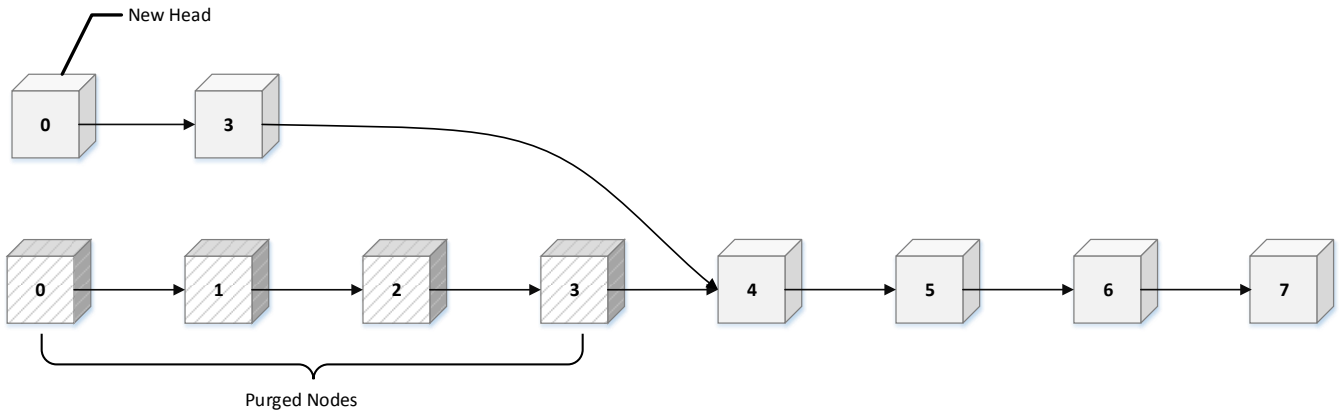
- Mark for logical deletion
- DPS to find next node
- Use stack as cache to DPS path

Lock-free Priority Queue

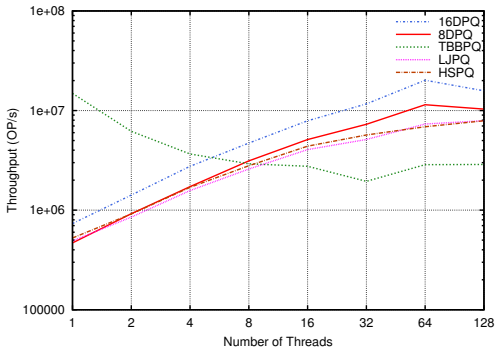
- Concurrent Insertion
  - Use CAS to swing predecessor's pointer
  - Use descriptor that resides in the node to finish the children adoption process
- Concurrent DeleteMin
  - Deletion threads use *deletion stack* to find the next node
  - Deletion threads move stack forward, and purge deleted nodes
  - Insertion threads rewind stack if new node cannot be reached

Abstract State Mapping

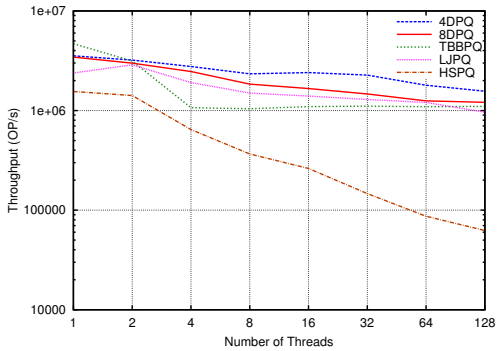




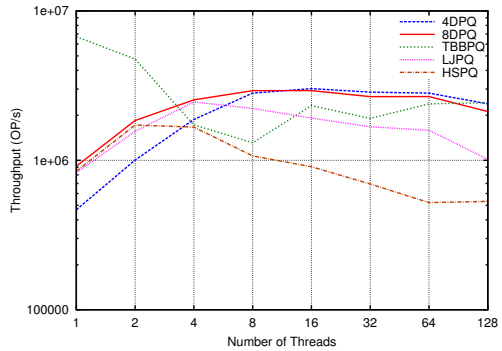
Throughput



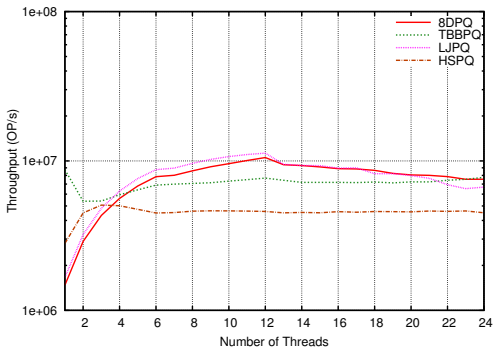
(a) 100% INSERT on the NUMA System



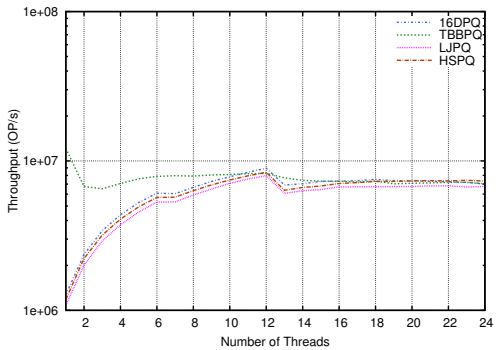
(b) 100% DELETEMIN on the NUMA System



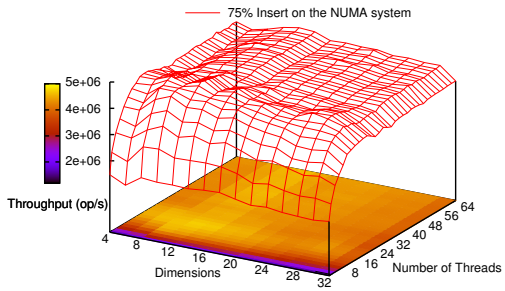
(c) 50% INSERT on the NUMA System



(d) 50% INSERT on the SMP System



(e) 80% INSERT on the SMP System



(f) Dimensionality and Performance