Facilitating Large-Scale Graph Searches with Lock-Free Pairing Heaps

Jeremy Mayers, Charles Newton, Peter Tonner

12 March 2012

Project Overview

- We are constructing a lock-free version of a Pairing heap (a self-balancing heap.)
- Pairing heaps have an efficient `decreaseKey` implementation (near-constant performance) that allow you to decrease the value in a heap without reinserting it.
- This improves the asymptotic performance of certain algorithms (e.g., Dijkstra's algorithm.)
- We are comparing our heap against Skipqueues (a priority queue backed with a Skiplist.)

Dijkstra's Algorithm

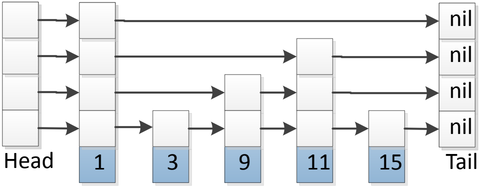
Single-Source Shortest Path Problem

In a weighted graph $G = (V, E)$, find the shortest path from a target vertex $v \in V$ to all other vertices in the graph.

- Push every node in a graph into a priority queue (PQ.) In the PQ, each node is weighted by current information we have about their distance.
- Initially, all nodes have a distance of ∞, except the target node, which has a distance of 0.
- Dynamic programming approach: Inductively build up our shortest routes.
  - Pop off the node on the PQ with the smallest distance. (The shortest path from the source to this node is finished.)
  - Update the weights in the PQ with the distances emanating from the popped node.

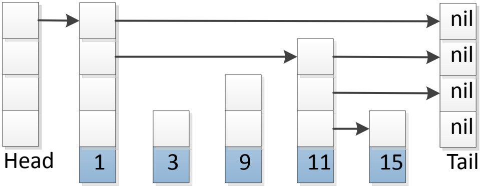
Skiplists

- Skiplists are constructed from a hierarchy of linked lists with the *Skiplist property*: the set of elements contained in level i is a subset of all the levels below it.
- Links allow you to do a binary search and "jump" around a list.
- The height of an element is randomly sampled from a power-law distribution: the probability of a node having a height of $i \geq 0$ is 2^{-i}.
- Time complexity of operations are probabilistically the same as for a binary search tree, but $\mathcal{O}(n)$ in the worst case.



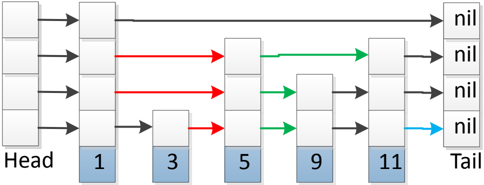
Skiplists (Search)

- For each level, move right until you run into a node greater than your target. Then, from this point, move right on the next lowest level, and repeat.
- Likely runs in $\mathcal{O}(\log n)$ time.
- Head / tail nodes have values of $-\infty$ and ∞, respectively.



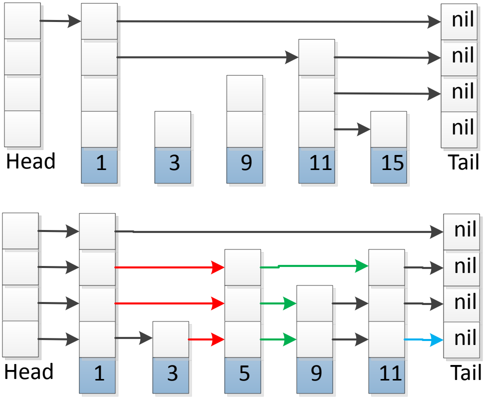
Next and Previous Windows

- A useful abstraction to make is the set of all pointers related to a given node. (Aggregate all the pointers related to one node.)
- `prev`$[i] \rightarrow$ the node in level i pointing to the target node.
- `next`$[i] \rightarrow$ the node the target node points to at level i.
- Use the search process to construct these sets.



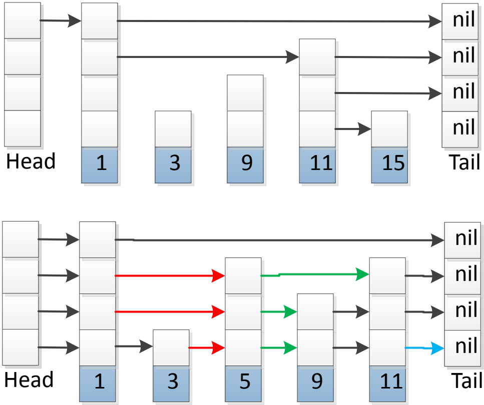
Skiplists (Insertion)

- Insertion: Find the location the target should be at. Set all of the node's next pointers to `next`$[i]$ and set the next pointers at each `prev`$[i]$ to the new node.



Skiplists (Deletion)

- Deletion: Set all of the next pointers in $\texttt{prev}[i]$ equal to $\texttt{next}[i]$.



Lock-free Skiplists

- Instead of linked lists at each level, we use lock-free linked lists.
- We lose the Skiplist property. (Levels aren't necessarily subsets of each other.) In particular, this means we need to always verify a node is in the lowest level.

Lock-free Skiplists

- Instead of linked lists at each level, we use lock-free linked lists.
- Linearization point: membership is defined at the lowest level of the Skiplist.
- We lose the Skiplist property. (Levels aren't necessarily subsets of each other.) In particular, this means we need to always verify a node is in the lowest level.

Lock-Free Skiplists (Insertion)

- Construct the `prev`$[i]$ and `next`$[i]$ sets.
- Set all of the node's next pointers to `next`$[i]$.
- If we can CAS the node into the bottom level, continue. Otherwise, something changed, and restart (reconstruct `prev`$[i]$ and `next`$[i]$).
- Next, CAS all `prev`$[i]$ to the new node. If a CAS fails, reconstruct `prev`$[i]$.

Lock-Free Skiplists (Deletion)

- Pointers are atomically *markable*.
  - In C / C++, steal a bit from the pointer.
  - In Java, use `AtomicMarkableReference`.
- Construct the `prev`$[i]$ and `next`$[i]$ sets.
- For each pointer in our target node, mark them as deleted.

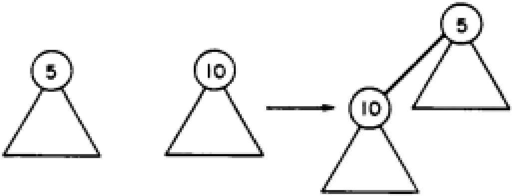
Optimized `decreaseKey` for Lock-Free Skiplists

- Currently, to implement a `decreaseKey` for Skiplists, we need to do two $\mathcal{O}(\log n)$ operations (one insert and one delete.)
- We can optimize this in some cases by removing redundant work: reuse the `prev`$[i]$ set as a starting point for the next insertion.

Pairing Heaps

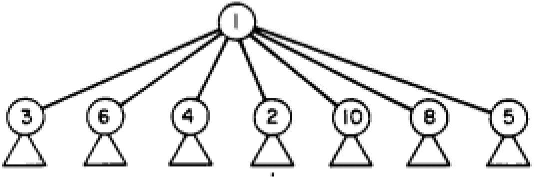
- Pairing Heaps were introduced by Fredman and Tarjan.
- `decreaseKey` runs in $2^{\mathcal{O}(\sqrt{\log\log n})}$ time
  - In practice, constant time. E.g., for a graph with a billion nodes, $2^{\sqrt{\log\log 10^9}=3.34}$.

Pairing Heaps: Melding Two Heaps



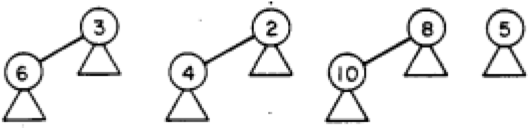
Pairing Heaps: `deleteMin`

- Pairing heaps have a two-step deleteMin operator.



Pairing Heaps: `deleteMin`

- After removing the root, `meld` each pair of heaps.



Pairing Heaps: `deleteMin`

- `meld` each resultant heap from right-to-left.