Buffer Heap: Basic Data Structure

The Buffer heap is essentially a set of Element and Update Buffers

These buffers are organized in levels of increasing size where each level holds 2^i element and 2^i Updates. Leading to a total of $r = \log_2 N$ levels

It is a lazy update structure where operations are accumulated and batch processed. This is done by inserting operations in the top buffer and pushing them down when appropriate.

Updates are applied when processing a delete-min, the reconstruction of the data structure, or when the update buffer overflows

---

These are the invariants that need to be maintained in order to ensure correctness of the priority queue.

Each Buffer in B and U contain at most $2^{depth\_of\_buffer}$ elements and 2^i operations

This limit on the update buffer helps ensure the I/O bound

The priority of all elements in a level's buffer is less than all the elements in buffers bellow

For any element all the operations to be applied to it lie in levels above or at in its level.

Elements in each buffer are kept sorted in ascending order by element ID

Updates are divided into a constant number of segments (3). Each of these segments have the operations sorted by element ID to which they apply

---

The delete and decrease key operations are inserted into the top buffer

Doing so may overflow the buffer. A fixU is called which in turn processes updates of an update buffer that overflowed.

Updates may turn into inserts and cause the element buffer to overflow

When that happens the 2^i smallest elements are kept and the rest are pushed down the structure

this is done recursively until the invariants are satisfied

Delete-Mins are processed immediately

They process the updates level by level until they are able to return an element

The data structure is also reconstructed from time to time, specifically when the updates are atleast twice the number of elements in the structure.

Reconstruction calls apply updates on all levels

The basic priority queue operations cost
- 1/b(lg N – lg B) when the first lg B levels can be placed in memory
- consequently when the first lg M levels are kept in internal memory the I/O bound drops to 1/b lg N/M
- without a limit on the update buffer and assuming a tall cache assumption in the cache oblivious model chowdhry and ramadachandran claim that it still falls in the bound

Chowdhry and ramadachandran claim the reconstruction is free when amortized due to charging of the updates to element movements through the levels

---

As for the implementation, it is done in C++ mainly for the low overhead costs and memory allocation capabilities

In my implementation I used the following structure to represent the elements of the priority queue

The reason for the extra fields is to use the same structure to host both operations and elements

the timestamps is only meaningful for operations and is basically a counter of the order in which the operations where inserted into the priority queue.

The operation code represents the type of operation, when set to null it signifies an element

---

The main data structure that makes up the priority queue is a vector, that would allow for storage of all the levels

Since Both elements and operations are of the same data type they can be placed into this vector

Originally, I had each level sorted by ID, mixing operations and elements. At first I thought this would reduce the number of comparisons, since we do a single scan of the buffer. However, to maintain these new invariants would require constant sorting of the buffers. So I decided to stick to the original proposed invariants that I mentioned earlier.

The Buffers are laid out in order of decreasing size leaving the top buffer at the end of the vector to allow for efficient additions to that buffer.

A main bottleneck in the efficiency of the program is handling the overflow of the buffers. To do this I currently use the random selection algorithm.

The algorithm keeps partitioning until 2^i elements have been gathered and the rest can be sunk to lower levels

To maintain the sorting of the elements by ID wither a sort is called after selection, or maintaining the partitions through merging while processing the selection. Sorting proves to not be as efficient

Also the randomized selection seems to be outperformed by the linear selection algorithm
The modified sampling referenced here, is the linear time algo creating three partitions into three rather than two.

There are some issues that slow down the priority Queue.

The first being that we start off with level that are quite small which overflow very quickly. A fix for this is to start at a level other than 0 to get initial buffers of bugger capacity

Another solution is to increase the base size of the buffers. Instead of going in powers of two, it could be increased. However, the trade off here is that each level is much smaller than the one after it and would not fill it fast enough. Also having large buffers would require more I/Os to scan at each level, since we have less levels we have a bigger chance of hitting a large buffer and incur a larger cost.
