# Problem Set 3 Solutions

**Problem 1.** We augment the vEB queue to also hold a maximum element. We implement the desired operations as follows:

- **find(x,Q)**: We check if $x$ is either the minimum or maximum of the current queue. If so, we return it. Otherwise, make a recursive call and find $\mathsf{low(x)}$ in the subqueue $Q[\mathsf{high(x)}]$.
- **predecessor(x,Q)**: If $x$ is less than the minimum of $Q$, return null. If $x$ is greater than the maximum of $Q$, return the maximum of $Q$. Otherwise, we make a recursive call to find the predecessor of $\mathsf{low(x)}$ in the subqueue $Q[\mathsf{high(x)}]$. If the result of this recursive call is non-null, then we return the result. Otherwise, we make a call to find the predecessor of $\mathsf{high(x)}$ in $Q.\mathsf{summary}$. The result of this call tells us the subqueue that is non-empty among the subqueues. In particular, if it is non-null, then we return the maximum element from that subqueue. However, if the result of the call was null, then we can return the minimum of $Q$.
- **successor(x,Q)**: The algorithm is very similar to predecessor.

**Problem 2.** Let us first recall Dijkstra's algorithm. Given a weighted graph $G = (V, E)$, and a node $s \in V$, it finds the length of the shortest path from $s$ to any node $u \in V$. The algorithm works by maintaining a set of nodes $S$ for which the shortest path from $s$ to $v \in S$ is already known. During each iteration, the algorithm chooses the node in $u \in V \setminus S$ whose estimated distance from $s$ is minimum. Then, for each node $w$ that is adjacent to $u$, the algorithm sets the estimated distance from $s$ to $w$ to be the the minimum of the current estimated distance and the distance from $s$ to $u$, and then using the edge $(u, w)$.

The key property to note in the above algorithm is that the maximum difference between the estimated distances from $s$ to any node in the priority queue holding the nodes in $V \setminus S$ is $C$. This can be proven by induction on the iteration on the number of nodes in $S$.

The solution then is to use two vEB queues, where the range of values in each vEB queue is $[1, C]$. The "left" queue will store the smaller values, and the "right" queue will store the larger values. When we want to insert an estimated distance into the queue, we only insert the distance mod $C$. Along with each queue, we will store the "base" value of that queue, which will be a multiple of $C$. By the key property above, we will never have to worry about filling a third queue, since the difference between any values in the two queues is at most $C$.

**Problem 3.**

**(a)** We will present how to perform the operations, and how to amortize the cost.

**Insert** We associate with each inserted item a potential of $k+\frac{k\Delta}{t}$. The additive factor of $k$ will be spent on constant time operations that will be performed each time an item goes one level down, and the factor of $\frac{k\Delta}{t}$ will reimburse the cost of scanning nodes. A new node is created when the number of elements in a heap exceeds $t$. Each of these elements donates $\frac{\Delta}{t}$ of energy, and goes one level down, so in total we collect $\Delta$ units of energy, and assign it to the newly created node as a reimbursement for scanning this node. Eventually, while performing insert of $x$, we descent down the trie, charging $x$'s energy, until we meet either a blob, into which we insert $x$ in constant time, or a heap, into which we insert $x$ in $O(I(t))$ time, or which is expanded down, and the heap's elements pay for it. Finally, the amortized cost of the operation is $O(k + \frac{k\Delta}{t} + I(t))$.

**Decrease-key** We remove an element $x$ from a structure in which it is. It can be either a heap or a blob, and the operation costs us at most $O(D(t))$ time. Next, in constant time, we determine a new bucket in which it should be, and follow down if the bucket has been already expanded (this is paid by the potential of $x$). Eventually, we determine a bucket or a blob into which $x$ has to be inserted, and we can do it in $O(I(t))$ time. Possible expansions are paid by the potential of elements of a heap. The amortized cost of decrease-key operation is $O(D(t) + I(t))$ time.

**Extract-min** Starting from the last entry from which we have extracted a minimum, we scan for the first non-empty entry, possibly going down, if the starting entry has been meanwhile expanded. In total, we scan each entry once, and the scanning is paid by the potential associated with nodes. We find a heap or a blob. If the latter, we convert it into a heap, and if an expansion occurs, we continue scanning. After all we find a non-empty heap, from which we remove the minimum in $O(X(t))$ time. For everything except the last removal the potential is charged. The amortized time cost is only $O(X(t))$.

**(b)** We perform at most $n$ inserts, $n$ extract-min, and $m$ decrease-key operations. This means that running time $R$ can be bounded by

$$O\left(n\left(k + \frac{k\Delta}{t} + I(t)\right) + nX(t) + m\left(D(t) + I(t)\right)\right).$$

Since in the case of Fibonacci heaps $I(t) = O(1)$, $D(t) = O(1)$, $X(t) = O(\log t)$, we can simplify our upper bound to

$$O\left(m + n\left(k + \frac{k\Delta}{t} + \log t\right)\right).$$

We substitute first $\Delta = C^{1/k}$, and $t = 2^k$, achieving

$$R = O\left(m + n\left(k + \frac{kC^{1/k}}{2^k}\right)\right).$$

We transform the last addend, knowing that $k = \sqrt{\log C}$:

$$\frac{kC^{1/k}}{2^k} = \frac{\sqrt{\log C}2^{\log C/\sqrt{\log C}}}{2^{\sqrt{\log C}}} = \sqrt{\log C}.$$

Eventually, we achieve

$$R = O(m + n\sqrt{\log C}).$$

**Problem 4.** **(a)** Consider the $(k+1)^{st}$ item inserted. Since only $k$ buckets (at worst) are occupied, the probability that *both* candidate locations are occupied is only $(k/n^{1/5})^2$. Thus, the expected number of times an item is actually inserted into an already-occupied bucket is at most

$$\begin{aligned}\sum_{k=0}^{n-1}(k/n^{1.5})^2 &= \frac{(n-1)(n)(2n-1)}{6n^3} \\ &\leq 1/3\end{aligned}$$

Now let's consider pairwise collisions. Item $k$ collides with item $j < k$ only if (i) one of the candidate locations of item $k$ is the location as item $j$ (this has probability at most $2/n^{1.5}$) and (ii) the other candidate location for item $k$ contains at least one element (probability $k/n^{1.5}$). Thus, the probability $k$ collides with $j$ is at most $k/n^3$. Summing over the $k$ possible values of $j < k$, we find the expected number of collisions for item $k$ is at most $k^2/n^3$. Summing over all $k$, we get the same result as above: $O(1)$ expected collisions.

**(b)** Start with a 2-universal family of hash functions mapping $n$ items to $2n^{1.5}$ locations. Consider any particular set of $n$ items. Consider choosing a random function from the hash family family. The probability that item $k$ collides with item $j$ is $1/2n^{1.5}$ by pairwise independence, implying by the union bound that the probability $k$ collides with *any* item is at most $1/2\sqrt{n}$.

Now suppose that we allocate *two* arrays of size $2n^{1.5}$ and choose a random 2-universal hash function from the family independently for each array. If an item has no collision in *either* array, then it will be placed in an empty bucket by the bash function. We need merely analyze the probability that this happens for every item (this would make the bash function perfect).

The probability that item $k$ has a collision in *both* arrays is at most $(1/2\sqrt{n})^2 = 1/4n$. It follows that the expected number of items colliding with some other item is at most $1/4$. This implies in turn that with probability $3/4$, every item is placed in an empty bucket by the (perfect) bash function. This in turn implies that *some* pair of 2-universal hash functions defines a perfect bash for our set of $n$ items.

Since every set of items gets a perfect bash from this scheme, it follows that the family of pairs of 2-universal functions above is a perfect bash family. Since the

2-universal family has size polynomial in the universe, so does the family of pairs of 2-universal functions.

**(c)** If we map our $n$ items to $k$ candidate locations in an array of size $n^{1+1/k}$, our collision odds work out as above and we get a constant number of collisions. Similarly, $k$ random 2-universal hash families, each mapping to a set of size $n^{1+1/k}$, has a constant probability of being perfect for any particular set of items, so the set of all such functions provides a perfect family (of polynomial size for any constant $k$). This gives a tradeoff of $k$ probes for perfect hashing in space $O(n^{1+1/k})$.

Note that while we can achieve perfect hashing to $O(n)$ space, the resulting family does *not* have polynomial size (since a different, subsidiary hash function must be chosen for each sub-hash-table).