Problem 1. In addition to $Q\text{.min}$, $Q\text{.high}$, and $Q\text{.low[]}$, we augment the VeB structure with $Q\text{.cmax}$. Where unlike $Q\text{.min}$, $Q\text{.cmax}$ simply keeps a copy of the maximum element, but the maximum element itself is stored in the unaugmented part of the structure as usual.

Due to our design choice for $Q\text{.cmax}$, insert and delete-min are only trivially modified:

\textbf{insert}(x,Q):

1. If $Q$ is empty, set $Q\text{.cmax} := x$, proceed with usual insert,

2. Else if, $x > Q\text{.max}$, set $Q\text{.cmax} := x$, proceed with usual insert,

3. Else, proceed with usual insert.

This is adding 1 extra step per level, on the running time of insert which touches $\log b$ levels, therefore total running time remains $O(\log b)$.

\textbf{delete-min}(Q):

1. Call usual delete-min(Q) (which internally recurses with the augmented delete-min), it returns element $x$.

2. If $x$ was the last element, i.e. $x = Q\text{.cmax}$, reset $Q\text{.cmax} := \text{NULL}$,

3. Return $x$. 
Same running time argument as above.

\textbf{find}(x,Q):

\begin{verbatim}
if Q.min = x
    return x
else if Q.low[x_h] not empty
    return find(x_l,Q.low[x_h])
else
    return NULL
\end{verbatim}

Running time is $T(b) = 1 + T(b/2) = O(\log b)$.

\textbf{successor}(x,Q):

\begin{verbatim}
if Q.min = x
    y_h := Q.high.min
    y_l := Q.low[y_h].min
    return (y_h,y_l)
else if (y_l := successor(x_l, Q.low[x_h])) \neq NULL
    return (x_h,y_l)
else
    y_h = successor (x_h, Q.high)
    return (y_h, Q.low[y_h].min)
\end{verbatim}

Running time is $T(b) = 1 + T(b/2) = O(\log b)$.

\textbf{predecessor}(x,Q):

\begin{verbatim}
if x = Q.min
    return NULL
\end{verbatim}
else if (y_l := predecessor(x_l, Q.low[x_h])) != NULL
    return (x_h, y_l)
else
    y_h = predecessor(x_h, Q.high)
    return (y_h, Q.low[y_h].cmax)

Running time is $T(b) = 1 + T(b/2) = O(\log b)$.

Finally, we argue that the space of the augmented VeB is still ...
To **insert** an \( x \in \{1, \ldots, nC \} \), let \( x_0 \) be the integer formed by the high order \( \log n \) bits of \( x \), and let \( x_1 \) be the integer formed by the remaining low order \( \log C \) bits of \( x \). Then insert \( x_1 \) into the \( x_0 \)-th VeB.

To **decrease-key** \( x \) to \( y \), first, **delete** \( x_1 \) from the \( x_0 \)-th VeB and insert \( y_1 \) into the \( y_0 \)-th VeB.

To **delete-min**, run **delete-min** on the VeB at the current pointer location, and if it is empty increase the pointer to point to the next VeB, and try again.

Observe that due to Dijkstra’s monotone queue access, at most two consecutive VeBs in the array can ever be non-empty. Since we are keeping a pointer to the lower one at all times, the worst time **delete-min** could take is \( O(\log \log C) \).

All of the above queue operations now run in \( O(\log \log C) \) time. Finally, as a matter of space optimization, we can use a wrap-around array of only 2 VeBs, instead of \( n \).

---

**Problem 3a.** We begin be listing all real costs:

1. Insert:
   1A. Traverse down until hit blob or heap
   1B. Occasionally need to insert into heap

2. Decrease-key:
   2A. Travel some number of nodes down
   2B. May have to insert in heap, if hit a heap instead of blob
   2C. May have to do a heap decrease-key, if already in heap
3. Delete-min:
   3A. Do delete-min from heap that contains minimum
   3B. Scan up tree nodes to find next minimum
   3C. Occasionally, have to make heap from blob (time linear in size of blob)

Now we match up all real costs with their potential counterparts.

The **insert** operation takes amortized $O(k + k\Delta/t + I(t))$: $k$ accounts for 1A, 2A, and 3C; $k\Delta/t$ accounts for 3B, because when each of $t$ nodes deposits $\Delta/t$ potential at a node, the node has $\Delta$ potential for a scan during 3B; and $I(t)$ accounts for 1B.

The **decrease-key** operation takes amortized $O(D(t) + I(t))$: $D(t)$ accounts for 2C; $I(t)$ accounts for 2B.

The **delete-min** operation takes amortized $O(X(t))$, which accounts for 3A.

The so described matching between charged (amortized) costs for operations and real costs fully covers the algorithm, which proves the claim.

**Problem 3b.** Substituting $k = \sqrt{\log C}$, $t = 2^{\sqrt{\log C}}$, $\Delta = C^{1/\sqrt{\log C}}$, $I(t) = O(1)$, $D(t) = O(1)$, and $X(t) = O(\log t)$ we get:

- **Delete-min**: $O(X(t)) = O(\log 2^{\sqrt{\log C}}) = O(\sqrt{\log C})$
- **Decrease-key**: $O(D(t) + I(t)) = O(1)$
- **Increase**: $O(k + k\Delta/t + I(t)) = O\left(\sqrt{\log C} + \sqrt{\log C} C^{1/2\sqrt{\log C}} + 1\right) = O(\sqrt{\log C})$

Therefore, we get Dijkstra's running time to be $O(m + n\sqrt{\log C})$. 

\[\sim\]
Problem 4a. Let $X_{ij}$ be the indicator random variable that the $i$-th inserted item collides with the $j$-th inserted item ($j < i$), where:

$$\Pr[X_{ij} = 1] \leq \frac{1}{n^{1.5}} \frac{l}{n^{1.5}} + \frac{1}{n^{1.5}} \frac{i - l}{n^{1.5}} = \frac{i}{n^3}$$

This is the union bound of: (a) $h_1(x_i) = h_1(x_j)$ and $h_2(x_i)$ hitting an allocated spot, and (b) $h_2(x_i) = h_2(x_j)$ and $h_1(x_i)$ hitting an allocated spot.

Expected total number of collisions is now:

$$E\left[\sum_{i=1}^{n} \sum_{j=1}^{i-1} X_{i,j}\right] = \sum_{i=1}^{n} \sum_{j=1}^{i-1} E[X_{i,j}]$$

$$= \sum_{i=1}^{n} \sum_{j=1}^{i-1} \frac{i}{n^3}$$

$$\leq \sum_{i=1}^{n} \frac{i^2}{n^3} \rightarrow \frac{1}{3}$$

Problem 4b. Let’s look at the terms comprising $\Pr[X_{ij} = 1]$ above: (a) $1/n^{1.5}$ uses that $i$ and $j$ are pairwise independent, and (b) $l/n^{1.5}$ is a union bound of $l$ copies of $1/n^{1.5}$ which themselves assume only pairwise independence.

Note that it is also required that the two hash functions used are independent from each other, otherwise we wouldn’t be able to multiply $1/n^{1.5}$ by $l/n^{1.5}$ e.g. above.

Therefore all we need is two independent hash functions that are each pairwise independant.

Problem 4c. Using Markov’s inequality (or just the definition of expectation), with $1/2$ probability, we get a pair of randomly selected pairwise
independant hash functions to produce no collisions. Therefore, we have to re-attempt to generate new hash functions expected 2 times until we get a perfect pair. Since the time for each attempt is $O(n)$, the expected running total running time is still $O(n)$.

Each hash function is described by 2 2-word seeds, therefore we need 8 words in total to describe both hash functions.