Solution:

1. We store $S$ in a vEB tree, and store the labels in a new array (named Labels) of length $u$. So Labels[$s$] = $v(s)$ for each $s$ in $S$. We change the vEB tree a little bit so that $min$ stores the index $s$ such that $v(s)$ is the minimum value of all the labels and $max$ store the index $s$ such that $v(s)$ is the maximum value. The algorithms and time complexity of the three operations are as follows.

   **Initialize**($S$). We initialize our vEB tree by inserting each $s \in S$ into the vEB tree and updating the $min$ and $max$ fields accordingly. We need to call the insertion function $|S|$ times, so the initialization takes $O(|S| \log \log u)$ time.

   **DecreaseKey**($s, x$). We need $O(\log \log u)$ to retrieve $v(s)$ and then check if $v(s) \leq x$. If so we are done. Otherwise, we delete $s$ and insert $s$ with the new value $x$, i.e. $v(s) = x$, during which min and max need to be updated accordingly. Both deletion and insertion take $O(\log \log u)$. Therefore, the overall time complexity is $O(\log \log u)$.

   **Minimum**($x$). If $V.min \leq x$, then the minimum label is Labels[V.min]. Otherwise we compare Minimum(V.cluster[high($x$)], low($x$)) with the minimums of the clusters on the left side of V.cluster[high($x$)]. Therefore,

   \[ T(u) \leq T(\sqrt{u}) + O(\sqrt{u}) \]

   applies and the overall time complexity is $O(\sqrt{u})$.

   **Space complexity.** On top of the original vEB tree, we only add an array of labels of length $u$, so space complexity is still $O(u)$.

2. (1) We store a cluster in a bit array instead of a subtree when its size is close to the computer word size. When $u$ is very large, only the clusters far away from the root have a word size, so the height of the vEB tree reduces by only a small constant. For example, if the word size is 16, the height decreases by 3. The recurrence relation is still the same $T(u) \leq T(\sqrt{u}) + O(1)$, and the base case is $T(16)$. This change does not influence the asymptotic performance $T(u) = O(\log \log u)$.

   (2) The pointer storage on the clusters near the end of the branches is avoided if these clusters are stored in bit arrays instead of trees, saving a lot of space. For example, if the word size is 16, then the cluster pointers in the nodes on the level where $u = 16$ and the level where $u = 4$ are all eliminated. Let $w$ be the word size. Before switching to a bit array, a cluster of size $w$ has $\sqrt{w}$ pointers pointing to its children that could be avoided if we switch this cluster to a bit array. It has $\sqrt{w}$ children and each child node has $\sqrt{\sqrt{w}}$ pointers. So on and so forth. The total number of pointers that can be avoided is

   \[ w^{\frac{1}{2}} + w^{\frac{1}{4}} + w^{\frac{1}{8}} + \ldots = \sum_{i=1}^{\log \log w} w^{\frac{1}{2^i}} = \frac{w}{2} \sum_{i=1}^{\log \log w} \frac{1}{2^i} = w \sum_{i=1}^{\log \log w} \left( \frac{1}{2^i} \right). \]

   Meanwhile, any operation can be directly done on the bit array without pointer dereferences, i.e., without recursively deciding which tree branch to visit, saving a lot of time as well.
• vEB-Tree-Minimum(V) or vEB-Tree-Maximum(V). No differences. They can be accessed at the root of the tree in constant time.

• vEB-Tree-Member(V,x). When we trace from the root to the node whose size is $w$, we can stop the recursion and directly check if the target bit in the bit array is 1, which is constant time. The amount of savings is at most $\lg \lg w$ steps of recursion.

• vEB-Tree-Successor(V,x) or vEB-Tree-Predecessor(V,x). Similarly, when we reach the node whose size is $w$, we can stop the recursion and directly find the successor/predecessor in the bit array, which is constant time. The amount of savings is at most $\lg \lg w$ steps of recursion.

• vEB-Tree-Insert(V,x) or vEB-Tree-Delete(V,x). Similarly, when we reach the node whose size is $w$, we can stop the recursion and directly set or reset the target bit in the bit array and update max and min, which is constant time. The amount of savings is at most $\lg \lg w$ steps of recursion.