CS381 Lecture Notes on Searching in k Arrays

We are given k sorted arrays $L_1, \ldots, L_k$ of n elements each, $k \leq n$. We want to build a data structure for processing $Locate(x)$ queries: Such a query is supposed to return the position (i.e., rank) of x in each of $L_1, \ldots, L_k$ (even if x is not in an L_i we still want its position in L_i – just as if we had done binary search for x in L_i).

The following is a sketch of solution that uses an $O(kn)$ space data structure, that can be built in $O(kn)$ time, and that can process a $Locate(x)$ query in $O(k + \log n)$ time (which is better than doing k separate binary searches for x, one in each array).

Here is a preliminary ("bad") solution that achieves the desired query time but uses $O(kn \log k)$ time and space to build the data structure: Build a binary tree T of height $\log k$ "on top" of the k arrays, with L_i being the ith leftmost leaf of T. For each node v of T, let $L(v)$ be defined as follows: if v is the ith leftmost leaf of T then $L(v) = L_i$, if v is an internal node then $L(v)$ is the merge of the lists of its two children. This takes $O(kn \log k)$ space because each of the kn items appears $\log k$ times in T. It also takes $O(kn \log k)$ time to build because, if u and w are children of v then $L(v)$ can be obtained in $O(|L(u)| + |L(w)|)$ time by merging $L(u)$ and $L(w)$; during this merge we also compute the cross-rank information between $L(v)$ and each of $L(u)$ and $L(w)$ (i.e., the rank of every item of $L(v)$ in each of $L(u)$ and $L(w)$). To locate x in the k arrays at the leaves, we first do a binary search for x in the array at the root, after which we go down one level at a time: we locate x in the children of the root (in constant time for each child because we just follow a cross-rank link to that child's list), then in the grandchildren of the root (again in constant time for each grandchild), etc (until we reach the leaves). Therefore, after we have done the $O(\log n)$ time binary search at the root, we can complete the job in an additional $O(k)$ time.

To improve the above to $O(kn)$ space and $O(kn)$ construction time for the data structure, we modify the way $L(v)$ is obtained from the children of v: If v has children u and w then we define $L(v)$ as the merge of $L'(u)$ and $L'(w)$, where $L'(u)$ is obtained from $L(u)$ by choosing every other element of $L(u)$ ($L'(w)$ is similarly obtained from $L(w)$). Therefore for any v, the size of $L(v)$ is n (whereas in the previous "bad" solution it was $2^h n$ if v was at a height of h). The total size of the $k/2$ lists at parents of leaves is therefore $kn/2$, the total size of the $k/4$ lists at grandparents of leaves is $kn/4$, etc. The overall space (and hence the time to build the structure) is therefore $(nk) \sum_i 2^{-i} = O(nk)$.
Of course in this "slimmed down" version of T, by following a cross-rank link we locate x in an $L'(v)$ rather than in an $L(v)$, but obtaining the position of x in $L(v)$ when we already know its positon in $L'(v)$ takes constant time (1 extra comparison).