CSCI2100: Regular Exercise Set 6

Prepared by Yufei Tao

Problems marked with an asterisk may be difficult.

Problem 1 (Counting Sort on a Multi-Set—A Linked List Version). Let S be a set of n key-value pairs (k, v). We know that each key k comes from the domain $[1, U]$ where U is an integer. The keys are not necessarily distinct. Design an algorithm to arrange these pairs in non-descending order of key. For example, if $S = \{(35, a), (12, b), (28, c), (12, d), (35, e), (7, f), (63, g), (35, h)\}$, a possible output of your algorithm is the array $(7, f, 12, b, 12, d, 28, c35, a, 35, b, 35, c, 63, g)$. Your algorithm should terminate in $O(n + U)$ time (note: U may be smaller than n). Recall that we have seen an algorithm for this purpose in the tutorial. See if you can come up with another (simpler) algorithm that combines arrays and linked lists.

Solution. Create an array A of length U, where each cell stores a pair of pointers referencing the head node of a linked list. In other words, there are U linked lists in total. At the beginning, all these linked lists are empty. This step takes $O(U)$ time.

For each $i \in [1, n]$, obtain the i-th key-value pair (k, v). Insert the pair into the linked list that $A[k]$ references. This takes $O(1)$ time per value of i, and hence, $O(n)$ time for all $i \in [1, n]$.

Scan the U linked lists from left to right, collect the key-value pairs stored there, and append them to the output array. The ordering of the elements in the same linked list does not matter. This step can be completed in $O(U)$ time.

Problem 2. Describe an algorithm to achieve the following purposes. Let S be a dynamic set of integers, which is empty at the beginning. Then, you will receive a number of insertions, each of which adds a new integer into S. Eventually, you will receive a signal "end", which indicates that there will be no more insertions. You will need to report the integers in the reverse chronological order, namely, if an integer is received earlier, it should be reported later. If there are n insertions in total, then your algorithm should have total running time $O(n)$. Note that you do not know the value of n until you receive the signal.

For example, if the integers inserted are in this order: 35, 12, 28, 12, 33), then you should produce an array $(33, 12, 28, 12, 35)$.

Solution. Push all the incoming integers into a stack. After receiving the end signal, produce the output array by popping all the elements from the stack.

Problem 3. Describe an algorithm to calculate $x + y$, where x and y are positive integers each of which has n decimal integers. Specifically, the formula is given in an array where the decimal digits of x are followed by $+$, and then followed by the decimal digits of y. For example, the expression $123 + 456$ is given in an array of length 7: $(1, 2, 3, +, 4, 5, 6)$. You should produce the answer in an array that stores the decimal digits of $x + y$. For instance, in the previous example, your output should be $(5, 7, 9)$. Your algorithm must finish in $O(n)$ time.

Solution. There are many ways to solve the problem. Next, we describe a way based on stacks.

Scan A from left to right. Keep pushing the digits into a stack A, until encountering the symbol

+. Then, push the remaining digits into another stack B. Next, we will pop the digits from the two stacks synchronously, and in the mean time perform the addition. Specifically, initialize an empty stack R, and set c to 0 (which is the carry). Repeat the following until both A and B are empty:

- Pop a number a from A, and a number b from B.
- Calculate $d = a + b + c$.
- If $d > 10$, then set c to 1, and d to $d - 10$; otherwise, set c to 0.
- Push d into R.

Finally, if $c = 1$, then push c into stack R.

To produce the output, simply keep popping from R, and append the obtained number into the output array.

Problem 4.** Let A be an array of n integers, some of which may be identical. Design a data structure that consumes $O(n)$ space, and supports the following *replacement operation* efficiently:

- `Replacement`(u, v), where u and v are both values in A, The operation replaces all the occurrences of u in A with v.

Your structure must support each such operation in $O(\log n + k)$ time, where k is the number of occurrences of u. Also, you need to design an algorithm to prepare the structure from A in $O(n \log n)$ time.

For example, suppose that $A = (35, 12, 28, 12, 35, 7, 63, 35)$. You can now run your algorithm to create the structure—your algorithm must finish in $O(n \log n)$ time. After this is done, you will be given replacement operations to process. For example, given `replacement`$(12, 28)$, you must update A to $(35, 28, 28, 28, 35, 7, 63, 35)$; for this operation, $k = 2$. Now, given `replacement`$(28, 35)$, you must further update A to $(35, 35, 35, 35, 35, 7, 63, 35)$; for this operation, $k = 3$.

Solution. First, collect all the distinct integers into B, and sort them in ascending order. This can be done in $O(n \log n)$ time. For every integer $x \in B$, create a linked list $L(x)$ storing all the positions of x in A. For example, $L(12)$ stores the set $\{2, 4\}$. Associate the head and tail pointers of $L(x)$ with the entry of x in B. This is our data structure. It can be constructed in $O(n \log n)$ time.

To perform a `Replacement`(u, v), first find the entry of u and v in B with binary search. This takes $O(\log n)$ time. Then, scan all the elements in $L(u)$; for every x stored therein, change $A[x]$ to v. This takes $O(k)$ time. Finally, append the entire $L(u)$ to $L(v)$. This can be done in $O(1)$ time by setting the next pointer of the tail of $L(v)$ to the address of the head of $L(u)$.

In our example, before the first replacement, $L(12)$ stores $\{2, 4\}$ and $L(28)$ stores $\{3\}$. After the first replacement, $L(28)$ stores $\{3, 2, 4\}$.

Problem 5* (Textbook Exercise 17.3-7). Suppose that we want to implement the following two operations on a set S of integers (S is empty at the beginning):

- Insert(e): Add a new integer e into S (you are assured that e is not already in S).
- Delete-Half: Delete the $\lceil |S|/2 \rceil$ smallest elements from S.

Describe a data structure that consumes $O(|S|)$ space, and supports each operation in $O(\log |S|)$ time amortized.

Solution. Simply store all the elements of S in a linked list. Every insertion takes $O(1)$ time. To perform a delete-half operation, move all the elements in S to an array of length $|S|$, and sort them in $O(|S| \log |S|)$ time. Then, remove the smallest $\lceil |S|/2 \rceil$ elements in $O(|S|)$ time. Charge the $O(|S| \log |S|)$ time on the insertions that added those elements. Each insertion bears only $O(\log |S|)$ extra cost. Every insertion can be charged only once.