CS2800 Design and Analysis of Algorithms
Problem Set #8 March 31, 2023

1. Show how to implement a queue using two stacks so that the amortized cost of enqueue and dequeue is $O(1)$. You can only perform push and pop on the stacks.

Solution:

Suppose there are two stacks S_1 and S_2. Enqueue is performed by pushing to S_1. To perform dequeue, we first check if S_2 is empty. If not, we will pop from S_2. If S_2 is empty, we will pop all elements from S_1 and push it into S_2 and then pop from S_2. This clearly performs enqueue and dequeue correctly.

To show the $O(1)$ amortized cost, consider the following accounting strategy: Each time an enqueue is performed, the element pushed to S_1 pays 4 units - one for the current push, one for a future pop from S_1, one for the subsequent push to S_2, and then one for the final pop from S_2. Since in a sequence of operations, there is exactly once that an element is pushed to S_1 and popped from S_1, and exactly once an element is pushed to S_2 and popped from S_2, this accounting covers all the possible costs for the operations for an elements. Thus the amortized cost is $O(1)$.

2. Let us consider a data structure to store a set of n elements that supports fast search and insertion. For a number n with binary representation $(b_k, b_{k-1}, \ldots, b_1, b_0)$, we have k arrays $A_0, A_1, \ldots, A_k$ such that if $b_i = 0$, then A_i is empty and otherwise A_i has 2^i elements. All the elements of the array A_i are sorted for each i, but there is no relation among the elements of different arrays.

   For instance, 11 elements will be stored using this data structure in sorted arrays A_3, A_1 and A_0 of lengths 8, 2 and 1 respectively since the binary representation of 11 is 1011.

   (a) Describe how you will perform a search operation in this data structure. What is the worst-case running time of your algorithm?

   Solution: If the number of elements in the data structure is n, then the number of arrays is $k = O(\log n)$. For each $i \leq k$, the array A_i has either 2^i elements or zero elements. In the worst case, the search-time for a key in A_i is $O(i)$. Thus, in the worst case all the arrays $A_1, A_2, \ldots, A_k$ are full and the running time is $\sum_{i=0}^{k} i = \Theta(\log^2 n)$.

   (b) Describe how you will insert a new element in this data structure. What is the worst-case and amortized running times of your algorithm?

   Solution: To insert an element e into the data structure, we first create an array with only e. If A_0 is empty, then we are done. Otherwise, we merge A_0 and $\{e\}$ to create

an array A'_1 with 2 elements. If A_1 is empty, we stop. Otherwise, we continue until we construct A'_i and A_i is empty. Merging of A_i with A'_i can be done in $O(|A_i|)$-time since the arrays are sorted. So the total running time is the time for merges, and in the worst case, if the data structure has n elements and all the arrays from A_0 to A_i consecutively contain all the n elements, then the worst-case time for an insert is $O(\sum_{j=1}^{i} |A_j|) = O(n)$.

For each element e in the data structure, when it is involved in a merge, then the size of the array it is contained in increases by a factor of 2. So, during the insertion of n items, the number of times an element is involved in a merge is at most $O(\log n)$. Hence, the total time for n insertions is at most $O(n \log n)$. Consequently, the amortized cost is $O(\log n)$.

(c) How will you implement a delete operation in this data structure? What is the worst-case running time of your algorithm?

Solution: Find the array A_i that contains the element e to be deleted ($O(\log^2 n)$-time). Now, find the smallest k such that A_k is non-empty ($O(\log n)$-time). Replace e in A_i with the first element in A_k and arrange accordingly ($O(n)$-time). Now, split the array A_k into arrays A_0, A_1, ..., A_{k-1} by copying elements from A_k. Worst-case running time is $O(n)$.

3. Recall the analysis of path compression that we did in class. We had divided the nodes into buckets based on the rank where all the nodes with rank between k and 2^k were put in a single bucket.

Suppose that we were not so clever, and we decided to create buckets where each bucket has nodes with ranks between $k+1$ and $2k$ starting from $k = 2$ together with nodes of rank 0, 1 and 2. If we did the analysis exactly like we did in class, what is the amortized running time for each Find operation that we will obtain?

Solution: First recall that the number of nodes with rank k in the disjoint set data structure that uses union-by-rank and path compression is at most $n/2^k$, where n is the total number of elements. Also recall that once a node becomes a non-root due to a union-by-rank, its rank remains unchanged throughout the rest of the unions and finds. Furthermore, for any node x, $\text{rank}(x) < \text{rank}(\text{parent}(x))$.

If we divide the ranks into buckets such that ranks $k+1$ to $2k$ form a bucket, then since the highest rank that is possible is $\log n$, we will have $\log \log n$ buckets. So, all the rank 1 nodes will be in a bucket, all the rank 2 nodes will be in a different bucket, nodes of rank 3 and 4 will be in a bucket, followed by nodes of rank between 5 and 8 and so on. In fact, using induction, you can show that the largest rank of a node in the i^{th} bucket is 2^{i-1}.

Suppose that when a node x becomes a non-root node its rank is between $k+1$ to $2k$, then x pays $2k$ as tax towards future path compression. The total tax payed by all nodes in this bucket is given by the formula

$$2k \sum_{i=1}^{k} \frac{n}{2^{k+i}} < 2k \frac{n}{2^k} < n \,,$$

since the number of nodes of rank $k+i$ is at most $n/2^{k+i}$ and each of them pay a tax of $2k$. Therefore the total tax payed by all the nodes is $n \cdot (\#\text{number of buckets}) \leq n \log\log n$. As we saw in class, this is sufficient to pay for all the links that connect nodes within the same bucket. The total cost payed for links between different buckets is again at most $n \log\log n$ since within a path from a node to the root, there are at most $\log\log n$ links between different buckets.

Therefore, the total cost of the n operations is at most $O(n \log\log n)$, and hence the amortized cost of Find operation is $O(\log\log n)$.

4. Recall the interval scheduling problem that we saw when we discussed greedy algorithms. We had briefly mentioned a recursive way of thinking about finding the optimal set of non-overlapping intervals. Design a dynamic programming based algorithm for the problem.

Solution: Sort the intervals in the decreasing order of end-times. Let schedule(i) denote the maximum number of overlapping intervals among the intervals from 1 to i. For an interval i, let conflict(i) denote the interval with smallest id less than i that conflicts with i. Then we can write the following recurrence.

$$\text{schedule}(i) = \min\{1 + \text{schedule}(\text{conflict}(i) - 1), \text{schedule}(i-1)\}$$

Now, we have schedule$(0) = 0$ and schedule$(1) = 1$ as base cases, and we have to compute schedule(n).

5. In the text segmentation problem that we saw in class, you are given a sequence of letters in an array $A[1, 2, \ldots, n]$, and you want to divide them into valid words. You have access to a subroutine IsWord that takes two indices i, j and returns **true** iff $A[i, i+1, \ldots, j]$ is a valid word.

   (a) Given an array $A[1, 2, \ldots, n]$ of characters, compute the number of partitions of A into valid words. For instance ARTISTOIL can be split in two ways: ARTIST-OIL or ART-IS-TOIL.

Solution: Let $\text{num}(i)$ denote the number of ways to divide the array $A[1, 2, \ldots, i]$ into valid words. Assume that $\text{IsWord}(i, j)$ returns 1 if $A[i, i+1, \ldots, j]$ is a valid word, and 0 otherwise. Then we can write,

$$\text{num}(i) = \sum_{k=1}^{i} \text{num}(k-1) \cdot \text{IsWord}(k, i)$$

As a base case, we have $\text{num}(k-1) = 1$, and we want to calculate $\text{num}(n)$.

(b) Given two arrays $A[1, 2, \ldots, n]$ and $B[1, 2, \ldots, n]$ of characters, decide if A and B can be partitioned into valid words at the same indices. For instance, BOTHEARTHANDSAT and PINSTARTRAPSAND can be partitioned at the same indices as BOT-HEART-HAND-SAT and PIN-START-RAPS-AND.

Solution: Let $\text{split}(i)$ denote the value 1 if both $A[1, 2, \ldots, i]$ and $B[1, 2, \ldots, i]$ can be split into valid words at the same indices. Thus, we can write

$$\text{split}(i) = \bigvee_{k=1}^{i} \text{split}(k-1) \wedge \text{IsWord}_A(k, i) \wedge \text{IsWord}_B(k, i).$$