Name: Andrii Bezrukov **UČO:** 570943

In the *Deleted sequence* problem, you maintain a dynamic set M of elements from the domain $\{1, 2, \ldots, n\}$ under the operations INSERT and DELETEMIN. The input is a sequence S of n INSERT and m DELETEMIN calls (possibly interleaved), where each key in $\{1, 2, \ldots, n\}$ is inserted exactly once. Your goal is to determine which key is returned by each DELETEMIN call. The expected output is an array $DeletedKeys[1 : m]$, where $DeletedKeys[i]$ is the key returned by the ith DELETEMIN call (for $i = 1, 2, \ldots, m$). Note that you are allowed to see the entire sequence S before determining any of the returned keys. To develop an algorithm for this problem, break the sequence S into homogenous subsequences $I_1, D, I_2, D, I_3, \ldots, I_m, D, I_{m+1}$, where each D represents a single DELETEMIN call and each I_j represents a (possibly empty) sequence of INSERT calls. For each subsequence I_j, initially place the keys inserted by these operations into a set K_j, which is empty if I_j is empty. Then execute the DELETEDSEQUENCE function.

```
function DeletedSequence(m, n)
    for i = 1 to m do
        determine j such that i ∈ K_j
        if j ≠ m + 1 then
            DeletedKeys[j] = i
            let l be the smallest value greater than j for which set K_l exists
            K_l = K_j ∪ K_l, destroying K_j
    return DeletedKeys
```

Choose an appropriate data structure and describe how to efficiently implement DELETEDSEQUENCE. Give as tight a bound as you can on the worst case running time of your implementation. *For full credit, you should choose the best data structure.*

1 Main Idea

The key insight is to use a Union-Find data structure with path compression to efficiently manage the sets K_j and implement the merging operations. Each element is represented as a node in the Union-Find structure, with the root of each set serving as its representative.

2 Data Structure

We will use Union-Find with:

- **Path compression** for `Find` operations.
- **Union-by-rank** for merging sets.
- An additional array to track which set (K_j) each element belongs to.

3 Implementation

Below is the pseudocode for the algorithm:

Name: Andrii Bezrukov **UČO:** 570943

0007 *list* 2 *učo* 570943 *body*

0123456789

```
function DeletedSequence(m, n, S)
    // Initialize data structures
    Parent[1...n] ← (1, 2, ..., n)
    Rank[1...n] ← (0, 0, ..., 0)
    SetID[1...n]
    DeletedKeys[1...m] ← (0, 0, ..., 0)
    j ← 1
    foreach operation op in S do
        if op is Insert(x) then
            SetID[x] ← j
        else if op is DeleteMin then
            j ← j + 1
    // Define Find with path compression
    function Find(x)
        if Parent[x] ≠ x then
            Parent[x] ← Find((Parent[x]))
        else
            return Parent[x]
        return Parent[x]
    // Define Union by rank
    function Union(x, y)
        rootX ← Find(x)
        rootY ← Find(y)
        if rootX = rootY then
            return
        if Rank[rootX] < Rank[rootY] then
            Parent[rootX] ← rootY
        else if Rank[rootX] > Rank[rootY] then
            Parent[rootY] ← rootX
        else
            Parent[rootY] ← rootX
            Rank[rootX] ← Rank[rootX] + 1
    for i = 1 to n do
        determine j such that i ∈ K_j                         // from SetID[i]
        if j ≠ m + 1 then
            DeletedKeys[j] ← i
            let l be the smallest value greater than j for which set K_l exists
            foreach x with SetID[x] = j do
                SetID[x] ← l
                // Assume y is a representative element in K_l
                if Find(x) ≠ Find(y) then
                    Union(x, y)
    return DeletedKeys
```

Algorithm 1: Pseudocode for DeletedSequence

Name: Andrii Bezrukov **UČO:** 570943

0007 *list* 3 *učo* 570943 *body*

0123456789

4 Time Complexity Analysis

- **Initialization:** $O(n)$ time to set up the data structures and process the input sequence.
- **Find operations with path compression:** $O(n \cdot \alpha(n))$ total time for all `Find` operations, where $\alpha(n)$ is the inverse Ackermann function.
- **Union operations:** $O(n \cdot \alpha(n))$ total time for all `Union` operations.
- **Overall:** $O(n \cdot \alpha(n))$, nearly linear since $\alpha(n)$ grows extremely slowly.

5 Worst-Case Time Complexity

In practice, the overall time complexity is $O(n \cdot \alpha(n))$, which for all practical purposes can be considered almost linear—effectively $O(n)$ because $\alpha(n)$ is a very slowly growing function.