1. Describe and analyze a data structure that maintains sequences of numbers, all initially equal to zero, subject to the following operations.

- \( S \leftarrow \text{Init}(n) \): Initialize a new sequence \( S[1..n] \) containing \( n \) zeros.
- \( \text{Shift}(S, i, j, \Delta) \): Add \( \Delta \) to every number in the interval \( S[i..j] \). The number \( \Delta \) is not necessarily an integer; moreover, \( \Delta \) could be positive, negative, or zero.
- \( \text{Scale}(S, i, j, \alpha) \): Multiply every number in the interval \( S[i..j] \) by \( \alpha \). The number \( \alpha \) is not necessarily an integer; moreover, \( \alpha \) could be positive, negative, or zero.
- \( x \leftarrow \text{Minimum}(S, i, j) \): Return the smallest number in the interval \( S[i..j] \).

**Solution:** I’ll first describe a static data structure that supports \text{Minimum} queries but no updates, then describe how to add the \text{Shift} and \text{Scale} updates, and finally sketch how to reduce the running time of \text{Init} to \( O(1) \). I will assume without loss of generality that \( n \) is a power of 2. I’ve made several arbitrary choices in my data structure design in the hope of simplifying the presentation; many other variants are also correct.

My data structure consists of a fixed and perfectly balanced binary tree, called a tournament tree, whose leaves store the values in the sequence in order from left to right. Each node \( v \) stores the following information:

- \( v.value \): the value of \( v \) (only if \( v \) is a leaf)
- \( v.left \): a pointer to \( v \)'s left child, if any
- \( v.right \): a pointer to \( v \)'s right child, if any
- \( v.first \): the minimum index among all leaf descendants of \( v \)
- \( v.last \): the maximum index among all leaf descendants of \( v \)
- \( v.min \): the minimum value among all leaf descendants of \( v \)

The \text{min}, \text{first}, and \text{last} fields are defined recursively as follows: If \( v \) is a leaf, we have

\[
\begin{align*}
    v.first &= v.last \\
    v.min &= v.value,
\end{align*}
\]

and otherwise,

\[
\begin{align*}
    v.first &= v.left.first, \\
    v.last &= v.right.last. \\
    v.min &= \min\{v.left.min, v.right.min\},
\end{align*}
\]

Initializng this data structure in \( O(n) \) time is straightforward.

To answer \text{Minimum}, we use an algorithm similar to the query algorithm for kd-trees from Homework 5. The first argument of \text{Minimum} is a node in the tree; specifically, in the top-level function call, \( v \) is the root.

Minimum\((v, i, j)\):
if \(i > v.\text{last}\) or \(j < v.\text{first}\)
   return \(\infty\)
else if \(i \leq v.\text{first}\) and \(j \geq v.\text{last}\)
   return \(v.\text{min}\)
else
   \(lmin \leftarrow \text{Minimum}(v.\text{left}, i, j)\)
   \(rmin \leftarrow \text{Minimum}(v.\text{right}, i, j)\)
   return \(\min\{lmin, rmin\}\)

Minimum\((v, i, j)\) calls itself recursively if and only if \(v.\text{first} < i \leq v.\text{last}\) or \(v.\text{first} \leq j < v.\text{last}\). At each level of the tree, there is at most one node \(v\) that meets each of these conditions. (These are the yellow nodes in the figure above.) It follows that the total number of recursive calls is at most \(4 \log_2 n\), which implies that Minimum runs in \(O(\log n)\) time.

Said differently, the tree partitions any index range \([i .. j]\) into \(O(\log n)\) canonical ranges, each associated with a node in the tree. The output of Minimum\((\cdot, i, j)\) is the smallest min value among these \(O(\log n)\) nodes. (These are the red nodes in the figure below.)

Now let’s consider the \textit{Shift} function. To implement this function efficiently, we take a lazy approach. Instead of actually adding \(\Delta\) to every number in the given index range, we record the a few subtrees should eventually be shifted upward by \(\Delta\). Specifically, we add a new field \(v.\text{shift}\) to the record of each node \(v\), which indicates that all values in the subtree rooted at \(v\) should eventually be shifted upward by \(v.\text{shift}\). At all times, we maintain the invariant
\[
v.\text{min} = \min \left\{ v.\text{left.min} + v.\text{left.shift}, \quad v.\text{right.min} + v.\text{right.shift} \right\}
\]
for every internal node \(v\). Initially, we set \(v.\text{shift} \leftarrow 0\) at every node \(v\). The following figure shows an example of a tournament tree with non-zero shift values that represents the same sequence of numbers as the figure above.
Before we examine any node \( v \) for any reason, we run the following algorithm, similar to \texttt{OKAYFINE} in the previous homework. By \texttt{Cleaning} nodes as we go, we can essentially pretend that these shifts do not exist, at only a small constant increase in running time.

\[
\text{Clean}(v): \\
\begin{align*}
\text{if } v \text{ is a leaf} & \quad v.\text{value} \leftarrow v.\text{value} + v.\text{shift} \\
\text{else} & \quad v.\text{left} \leftarrow v.\text{left} + v.\text{shift} \\
& \quad v.\text{right} \leftarrow v.\text{right} + v.\text{shift} \\
& \quad v.\text{min} \leftarrow v.\text{min} + v.\text{shift} \\
& \quad v.\text{shift} \leftarrow 0
\end{align*}
\]

The actual \texttt{Shift} algorithm closely resembles \texttt{Minimum}. If the query range \([i..j]\) contains every leaf below \( v \), we adjust \( v.\text{shift} \). Otherwise, if the query range \([i..j]\) contains at least one leaf below \( v \), we recursively \texttt{Shift} both children of \( v \). The \texttt{Minimum} algorithm itself needs only one minor change. Both algorithms run in \( O(\log n) \) time in the worst case.

\[
\text{Shift}(v, i, j, \Delta): \\
\begin{align*}
\text{Clean}(v) & \quad \text{if } i > v.\text{last} \text{ or } j < v.\text{first} \\
& \quad \text{return} \\
\text{else if } i \leq v.\text{first} \text{ and } j \geq v.\text{last} & \quad v.\text{shift} \leftarrow \Delta \\
\text{else} & \quad \text{Shift}(v.\text{left}, i, j, \Delta) \\
& \quad \text{Shift}(v.\text{right}, i, j, \Delta)
\end{align*}
\]

\[
\text{Minimum}(v, i, j): \\
\begin{align*}
\text{Clean}(v) & \quad \text{if } i > v.\text{last} \text{ or } j < v.\text{first} \\
& \quad \text{return } \infty \\
\text{else if } i \leq v.\text{first} \text{ and } j \geq v.\text{last} & \quad \text{return } v.\text{min} \\
\text{else} & \quad l\text{min} \leftarrow \text{Minimum}(v.\text{left}, i, j) \\
& \quad r\text{min} \leftarrow \text{Minimum}(v.\text{right}, i, j) \\
& \quad \text{return } \min\{l\text{min}, r\text{min}\}
\end{align*}
\]

To add support for \texttt{Scale}, we introduce two more lazily-updated fields \( v.\text{scale} \) and \( v.\text{max} \) at every vertex, and we maintain the following invariants. (We need the \texttt{max} field and the more complicated invariants because \( v.\text{scale} \) could be negative!)

\[
v.\text{min} = \begin{cases} 
\min \left\{ v.\text{left} . \text{min} \cdot v.\text{scale} + v.\text{left} . \text{shift}, \right. \\
\left. v.\text{right} . \text{min} \cdot v.\text{scale} + v.\text{right} . \text{shift} \right\} & \text{if } v.\text{scale} \geq 0 \\
\min \left\{ v.\text{left} . \text{max} \cdot v.\text{scale} + v.\text{left} . \text{shift}, \right. \\
\left. v.\text{right} . \text{max} \cdot v.\text{scale} + v.\text{right} . \text{shift} \right\} & \text{otherwise}
\end{cases}
\]

\[
v.\text{max} = \begin{cases} 
\max \left\{ v.\text{left} . \text{max} \cdot v.\text{scale} + v.\text{left} . \text{shift}, \right. \\
\left. v.\text{right} . \text{max} \cdot v.\text{scale} + v.\text{right} . \text{shift} \right\} & \text{if } v.\text{scale} \geq 0 \\
\max \left\{ v.\text{left} . \text{min} \cdot v.\text{scale} + v.\text{left} . \text{shift}, \right. \\
\left. v.\text{right} . \text{min} \cdot v.\text{scale} + v.\text{right} . \text{shift} \right\} & \text{otherwise}
\end{cases}
\]

Initially, \( v.\text{scale} = 1 \) for every vertex \( v \). Adapting \texttt{Clean} to handle \texttt{scale} is tedious but straightforward:
((Reset v.shift and v.scale and propagate to children))

\[
\text{\textbf{Clean(v):}} \\
\begin{align*}
\text{if } v \text{ is a leaf:} & \\
\quad & v.\text{value} \leftarrow v.\text{value} \cdot v.\text{scale} + v.\text{shift} \\
\text{else:} & \\
\quad & v.\text{left.scale} \leftarrow v.\text{left.scale} \cdot v.\text{scale} \\
\quad & v.\text{left.shift} \leftarrow v.\text{left.shift} \cdot v.\text{scale} + v.\text{shift} \\
\quad & v.\text{right.scale} \leftarrow v.\text{right.scale} \cdot v.\text{scale} \\
\quad & v.\text{right.shift} \leftarrow v.\text{right.shift} \cdot v.\text{scale} + v.\text{shift} \\
\quad & v.\text{min} \leftarrow v.\text{min} \cdot v.\text{scale} + v.\text{shift} \\
\quad & v.\text{max} \leftarrow v.\text{max} \cdot v.\text{scale} + v.\text{shift} \\
\quad & \text{if } v.\text{min} > v.\text{max:} \\
\quad & \quad \text{swap } v.\text{min} \leftrightarrow v.\text{max} \\
\quad & v.\text{scale} \leftarrow 1 \\
\quad & v.\text{shift} \leftarrow 0
\end{align*}
\]

Finally, \textbf{Scale} follows the same recursion pattern as \textbf{Minimum} and \textbf{Shift}, and thus also runs in \(O(\log n)\) worst-case time.

\[
\text{\textbf{Scale}(v, i, j, \alpha):} \\
\begin{align*}
\text{\textbf{Clean}(v)} \\
\text{if } i > v.\text{last} \text{ or } j < v.\text{first:} & \\
\quad & \text{return} \\
\text{else if } i \leq v.\text{first} \text{ and } j \geq v.\text{last:} & \\
\quad & v.\text{scale} \leftarrow \Delta \\
\text{else:} & \\
\quad & \text{\textbf{Scale}(v.\text{left}, i, j, \alpha)} \\
\quad & \text{\textbf{Scale}(v.\text{right}, i, j, \alpha)}
\end{align*}
\]

Finally, to implement \textbf{Init} in \(O(1)\) time, we use a simple idea: \textbf{Don’t allocate nodes until we actually need them.} The modified \textbf{Init} only allocates the root node \(r\), by calling \textbf{NewNode}(1, n). To lazily create the other nodes, it suffices to add a few more lines to \textbf{Clean} that allocate the children of the node being cleaned if they do not already exist.

\[
\text{\textbf{Clean}(v):} \\
\begin{align*}
\text{if } v.\text{first} = v.\text{last:} & \\
\quad & v.\text{value} \leftarrow v.\text{value} \cdot v.\text{scale} + v.\text{shift} \\
\text{else:} & \\
\quad & \text{mid} \leftarrow \lfloor (v.\text{first} + v.\text{last}) \div 2 \rfloor \\
\quad & \text{if } v.\text{left} = \text{NULL:} \\
\quad & \quad v.\text{left} \leftarrow \text{\textbf{NewNode}(v.\text{first, mid})} \\
\quad & \quad v.\text{right} \leftarrow \text{\textbf{NewNode}(mid + 1, v.\text{last})} \\
\quad & v.\text{left.scale} \leftarrow v.\text{left.scale} \cdot v.\text{scale} \\
\quad & v.\text{left.shift} \leftarrow v.\text{left.shift} \cdot v.\text{scale} + v.\text{shift} \\
\quad & \vdots
\end{align*}
\]

\[
\text{\textbf{NewNode}(first, last):} \\
\begin{align*}
& v \leftarrow \text{new node} \\
& v.\text{left} \leftarrow \text{NULL} \\
& v.\text{right} \leftarrow \text{NULL} \\
& v.\text{first} \leftarrow \text{first} \\
& v.\text{last} \leftarrow \text{last} \\
& v.\text{value} \leftarrow 0 \\
& v.\text{min} \leftarrow 0 \\
& v.\text{max} \leftarrow 0 \\
& v.\text{shift} \leftarrow 0 \\
& v.\text{scale} \leftarrow 1 \\
& \text{return } v
\end{align*}
\]