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

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

Solution: I'll first describe a static data structure that supports Minimum queries but no updates, then describe how to add the Shift and Scale updates, and finally sketch how to reduce the running time of 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 *min*, *first*, and *last* fields are defined recursively as follows: If v is a leaf, we have

$$\begin{aligned} v.first &= v.last \qquad \text{and} \\ v.min &= v.value, \end{aligned}$$

and otherwise,

$$\begin{aligned} v.first &= v.left.first, \\ v.last &= v.right.last. \\ v.min &= \min\{v.left.min,\ v.right.min\}, \end{aligned}$$

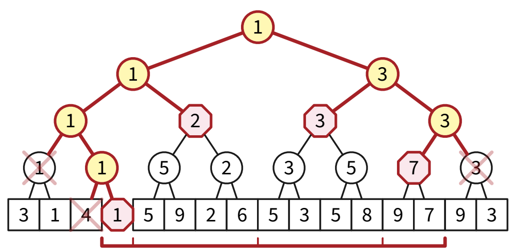
Initializing this data structure in $O(n)$ time is straightforward.

To answer Minimum, we use an algorithm similar to the query algorithm for kd-trees from Homework 5. The first argument of 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.last or j < v.first
      return ∞
  else if i ≤ v.first and j ≥ v.last
      return v.min
  else
      lmin ← MINIMUM(v.left, i, j)
      rmin ← MINIMUM(v.right, i, j)
      return min{lmin, rmin}
```

MINIMUM(v, i, j) calls itself recursively if and only if $v.first < i \le v.last$ or $v.first \le j < v.last$. 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.)

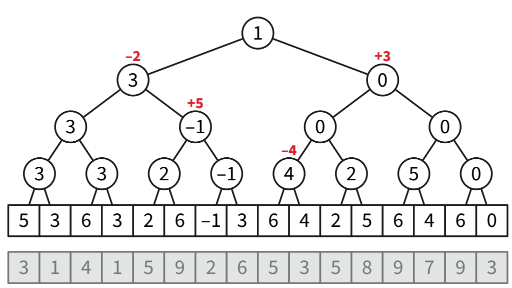


Answering a range-minimum query

Now let's consider the SHIFT function. To implement this function efficiently, we take a lazy approach. Instead of actually adding Δ to every number in the given index range, we record the a few *subtrees* should *eventually* be shifted upward by Δ. Specifically, we add a new field $v.shift$ to the record of each node v, which indicates that all values in the subtree rooted at v should *eventually* be shifted upward by $v.shift$. At all times, we maintain the invariant

$$v.min = \min \left\{ \begin{array}{c} v.left.min + v.left.shift, \\ v.right.min + v.right.shift \end{array} \right\}$$

for every internal node v. INITially, we set $v.shift \leftarrow 0$ 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.



Stored *value*s and *min*s are black; non-zero *shift* fields are red; represented leaf values are gray.

Before we examine any node v for any reason, we run the following algorithm, similar to OkayFine in the previous homework. By Cleaning nodes as we go, we can essentially pretend that these shifts do not exist, at only a small constant increase in running time.

```
⟨⟨Reset v.shift to 0 and propagate shift to children⟩⟩
CLEAN(v):
  if v is a leaf
      v.value ← v.value + v.shift
  else
      v.left.shift ← v.left.shift + v.shift
      v.right.shift ← v.right.shift + v.shift
  v.min ← v.min + v.shift
  v.shift ← 0
```

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

```
SHIFT(v, i, j, Δ):
  CLEAN(v)
  if i > v.last or j < v.first
      return
  else if i ≤ v.first and j ≥ v.last
      v.shift ← Δ
  else
      SHIFT(v.left, i, j, Δ)
      SHIFT(v.right, i, j, Δ)
```

```
MINIMUM(v, i, j):
  CLEAN(v)
  if i > v.last or j < v.first
      return ∞
  else if i ≤ v.first and j ≥ v.last
      return v.min
  else
      lmin ← MINIMUM(v.left, i, j)
      rmin ← MINIMUM(v.right, i, j)
      return min{lmin, rmin}
```

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

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

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

Initially, $v.scale = 1$ for every vertex v. Adapting Clean to handle *scale* is tedious but straightforward:

```
⟨⟨Reset v.shift and v.scale and propagate to children⟩⟩
CLEAN(v):
  if v is a leaf
      v.value ← v.value · v.scale + v.shift
  else
      v.left.scale ← v.left.scale · v.scale
      v.left.shift ← v.left.shift · v.scale + v.shift
      v.right.scale ← v.right.scale · v.scale
      v.right.shift ← v.right.shift · v.scale + v.shift
  v.min ← v.min · v.scale + v.shift
  v.max ← v.max · v.scale + v.shift
  if v.min > v.max
      swap v.min ↔ v.max
  v.scale ← 1
  v.shift ← 0
```

Finally, SCALE follows the same recursion pattern as MINIMUM and SHIFT, and thus also runs in $O(\log n)$ worst-case time.

```
SCALE(v, i, j, α):
  CLEAN(v)
  if i > v.last or j < v.first
      return
  else if i ≤ v.first and j ≥ v.last
      v.scale ← Δ
  else
      SCALE(v.left, i, j, α)
      SCALE(v.right, i, j, α)
```

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

```
CLEAN(v):
  if v.first = v.last
      v.value ← v.value · v.scale + v.shift
  else
      mid ← ⌊(v.first + v.last)/2⌋
      if v.left = NULL
          v.left ← NEWNODE(v.first, mid)
          v.right ← NEWNODE(mid + 1, v.last)
      v.left.scale ← v.left.scale · v.scale
      v.left.shift ← v.left.shift · v.scale + v.shift
      ⋮
  ⋮
```

```
NEWNODE(first, last):
  v ← new node
  v.left ← NULL
  v.right ← NULL
  v.first ← first
  v.last ← last
  v.value ← 0
  v.min ← 0
  v.max ← 0
  v.shift ← 0
  v.scale ← 1
  return v
```

■