Homework 7

Evan Lin, Roshan Hegde

April 5, 2024

Background

This data structure was made for Homework 7 of the CS 225 (Data Structures) Honors Supplement course at the University of Illinois Urbana Champaign in Spring 2024. We will provide an overview of the data structure, a proof of the runtime of each of the operations it supports, as well as a python implementation.

The objective is to create a data structure that stores a sequence of n numbers, where $n = 2^k - 1$, that are all initially zero that supports the following operations:

1. `Init`(n), to initialize a sequence of n zeros.
2. `Shift`(S, i, j, Δ) to add Δ to every number in the interval from i to j in S.
3. `Scale`(S, i, j, α) to multiply every number in the interval from i to j in S by α.
4. `Minimum`(S, i, j) to find the minimum number in the interval from i to j in S.

The problem was split into the following parts:

(a) Support `Minimum` on a static data structure in $O(\log(n))$ time.

(b) Support `Minimum` and `Shift` in $O(\log(n))$ time.

(c) Support `Minimum`, `Shift` and `Scale` in $O(\log(n))$ time.

(d) Support `Minimum`, `Shift` and `Scale` in $O(\log(n))$ time, and `Init` in $O(1)$ time.

Obviously, this data structure was made for fun, and we appreciate any advice you may have to improve the writeup.

If there are any mistakes, please contact me (found on my github profile, https://github.com/evanlin23)

Thank you!

- Evan Lin, Roshan Hedge

Lin-Hedge Matrix Tree Data Structure

Our data structure will be called the Lin-Hedge Matrix tree, or alternatively, LH Tree. We will start by describing the base data structure we will use for the following parts.

We are given that n, the number of nodes in our data structure is a power of 2 minus 1.

Special Case, $n = 1$

In the case that where our sequence is of size 1, we will store a simpler version of our LH Tree that only has its own shift factor Δ and scale factor stored α. Label this structure M.

Note that in this case, $i = j = 0$.

When we `Shift`(M, i, j, Δ), we set $M \to \Delta = (M \to \Delta) + \frac{\Delta}{M \to \alpha}$.

When we `Scale`(M, i, j, α) we set $M \to \alpha = (M \to \alpha) \cdot \alpha$.

When we call `Minimum`(M, i, j) we return the value of $(M \to \alpha) \cdot (M \to \Delta)$.

These 3 operations all take constant time.

General Case, $n \geq 3$

For values of $n \geq 3$, where n is the number of items in our sequence, we will do the following. We will represent our n values in a perfect binary tree of height $\log(n) - 1$. We will store a root pointer to the root of our tree. There will also be a left and right pointers to the left and right children of a node. At each of our nodes, we will store 9 values.

(1) Δ_L represents a shift applied to the entirety of the node's left subtree.

(2) Δ represents a shift applied to itself.

(3) Δ_R represents a shift applied to the entirety of the right subtree.

(4) α_L represents a scale factor applied to the entirety of the node's left subtree.

(5) α represents a scale factor applied to itself.

(6) α_R represents a scale factor applied to the entirety of the right subtree.

(7) $[min, max]$ will represent the minimum and maximum of the entire subtree rooted at the node.

(8) h represents the height of the subtree rooted at the node.

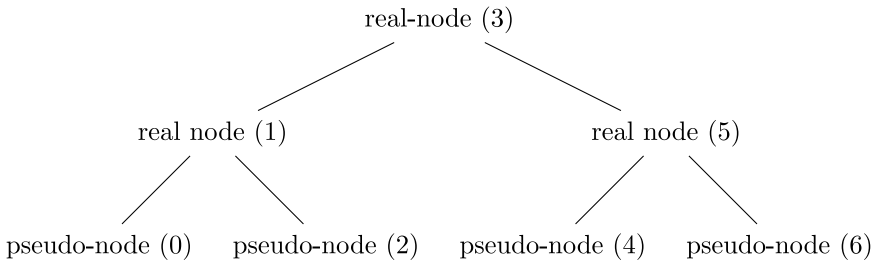
(9) idx represents the index of the sequence that the node corresponds to. Our items are indexed from 0.

The shift and scale factors, as well as the rest of a node's information will be stored as following, hence the namesake of the data structure:

$$\texttt{Data} = \begin{bmatrix} \Delta_L & \Delta & \Delta_R \\ \alpha_L & \alpha & \alpha_R \\ [min, max] & h & idx \end{bmatrix}$$

The height of our LH Tree is $\log(n) - 1$, meaning we do not have a node for every element in our structure. This is because we can store the children's data in the parent node. To figure out the index of the sequence that each node corresponds to, we will first place left and right pseudo-node children stemming all of the nodes at the bottom level of our LH tree. Then, including these pseudo-node children, the order of the nodes when performing an inorder traversal will correspond to their placement in the sequence the tree represents.

For example, take a look at the following LH Tree of height 1. Only the index the nodes or pseudo-nodes correspond to are shown for clarity. Only the root, and its left and right children are part of the LH Tree of height 1; the 4 pseudo-nodes at the bottom of the tree are not stored, and are only shown for clarity to be used to figure out the index of the sequence each node corresponds to. Notice that all the information needed about these pseudo-nodes are stored in their parents.

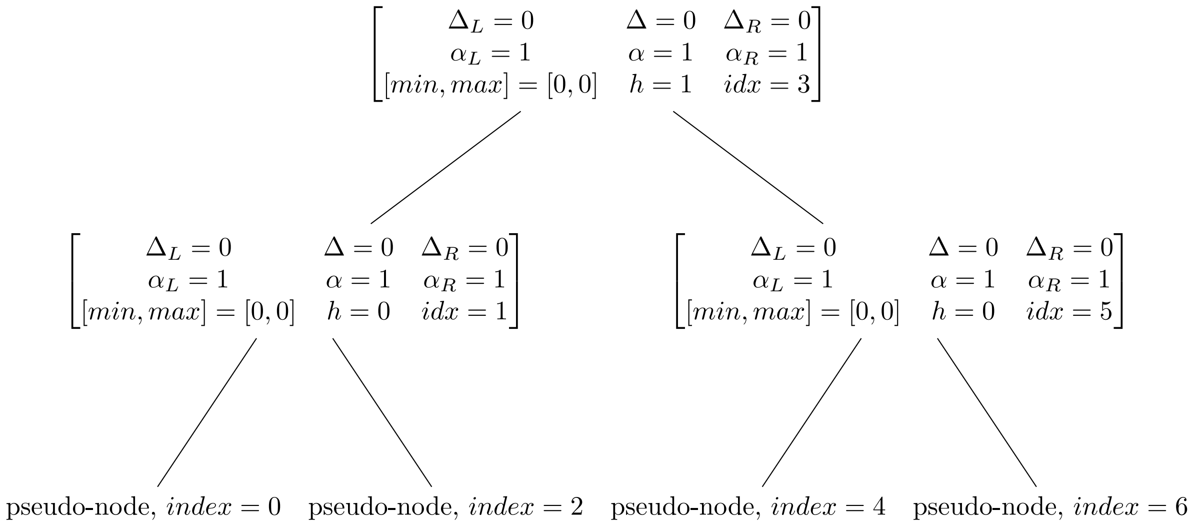


When we initialize a new sequence, each node's Δ_L, Δ, and Δ_R start at 0, and each node's α_L, α and α_R start at 1.

Upon initialization, we instantiate all the nodes for a full LH tree; Later, to have a constant `Init`, we will be lazily creating our nodes. This lazy creation means we only create a given node as we need to visit and descend down to them; as creating a node takes constant time, this is a constant amount of work on top of a constant amount of work for whatever given operation we were already going to perform.

As an example, let's initialize a sequence of length 7, meaning a LH tree of height 1.

We have the following:


$\begin{bmatrix} \Delta_L = 0 & \Delta = 0 & \Delta_R = 0 \\ \alpha_L = 1 & \alpha = 1 & \alpha_R = 1 \\ [min, max] = [0,0] & h = 1 & idx = 3 \end{bmatrix}$
$\begin{bmatrix} \Delta_L = 0 & \Delta = 0 & \Delta_R = 0 \\ \alpha_L = 1 & \alpha = 1 & \alpha_R = 1 \\ [min, max] = [0,0] & h = 0 & idx = 1 \end{bmatrix}$
$\begin{bmatrix} \Delta_L = 0 & \Delta = 0 & \Delta_R = 0 \\ \alpha_L = 1 & \alpha = 1 & \alpha_R = 1 \\ [min, max] = [0,0] & h = 0 & idx = 5 \end{bmatrix}$
pseudo-node, $index = 0$
pseudo-node, $index = 2$
pseudo-node, $index = 4$
pseudo-node, $index = 6$

Python Implementation

We understand that in depth description of our data structure can be quite verbose, so we have implemented our data structure in python.

```
from math import log2

class Transform:
    alpha = 1
    delta = 0
    def val(self):
        return self.alpha * self.delta
    def __init__(self, alpha=1, delta=0):
        self.alpha = alpha
        self.delta = delta
    def __add__(self, o):
        return Transform(self.alpha, self.delta + o / self.alpha)
    def __mul__(self, o):
        return Transform(self.alpha * o, self.delta)
    def apply(self, o):
        return Transform(self.alpha * o.alpha, self.delta / o.alpha + o.delta)
    def scale(self, factor, t):
        return t.apply(self) * factor + -t.val()

class Node:
    height = 0
    index = 0

    transform_left = Transform()
    transform_right = Transform()
    transform = Transform()
    min = 0
    max = 0

    left = None
    right = None

    def __init__(self, height, index):
        self.height = height
        self.index = index

class Op:
    here  = lambda tree,      transform: None # behavior if interval includes node index
    left  = lambda tree,      transform: None # behavior if interval includes entire left
    ↪  subtree
    right = lambda tree,      transform: None # behavior if interval includes entire
    ↪  right subtree
    after = lambda tree, arr, transform: None # behavior after operations have been done

def build_tree(start, end):
    height = int(log2(end - start)) - 1
    index = (end - 1 + start) // 2
    root = Node(height, index)
    if height > 0:
        root.left = build_tree(start, index)
        root.right = build_tree(index + 1, end)
    return root

def init(N):
    return build_tree(0, N)


def apply(tree, start, end, op, transform):
    arr = []
    if start <= tree.index and end > tree.index: # if the interval includes current index
        arr.append(op.here(tree, transform))

    if tree.height == 0: # if we are a root node
        if start == tree.index - 1:
            arr.append(op.left(tree, transform))
        if end - 1 == tree.index + 1:
            arr.append(op.right(tree, transform))
        return op.after(tree, arr, transform)

    # if the interval doesnt cover the current node, we descend in to one of the subtrees
    if end - 1 < tree.index:
        return apply(tree.left, start, end, op, transform.apply(tree.transform_left))
    if start > tree.index:
        return apply(tree.right, start, end, op, transform.apply(tree.transform_right))

    if start < tree.index:
        interval_size = tree.index - start
        tree_size = 2 ** (tree.left.height + 2) - 1
        if (interval_size == tree_size): # no need to descend if interval covers whole
        ↪  subtree
            arr.append(op.left(tree, transform))
        else: # descend into subtree
            arr.append(apply(tree.left, start, tree.index, op,
            ↪  transform.apply(tree.transform_left)))

    if end - 1 > tree.index:
        interval_size = end - (tree.index + 1)
        tree_size = 2 ** (tree.right.height + 2) - 1
        if (interval_size == tree_size):
            arr.append(op.right(tree, transform))
        else:
            arr.append(apply(tree.right, tree.index + 1, end, op,
            ↪  transform.apply(tree.transform_right)))

    return op.after(tree, arr, transform)
```

Here, we have a generic way to query our tree. Using the `Op` class, we can abstract the behavior of what we want to do on each subtree, while keeping the interval query standardized. To implement any operation, we simply need to define the four behaviors our operation has:

Analysis of Number of Visits to Nodes

We have described the structure of each node within the LH trees; now we will provide a basis to prove the runtime of the rest of the operations needed is $O(\log(n))$.

For further use when proving the runtimes of `Minimum`, `Shift` and `Scale`, we will prove that for any range-based operation on our LH Tree will take at most $O(log(n))$ visits to nodes.

Let M denote a particular LH tree. Let `Operation`(M, i, j, o) denote any operation on the subsequence from i to j, and o is a optional additional parameter, depending on the `Operation` used. Let $\texttt{Data}_L$ denote the information stored in a node about its left subtree, $\texttt{Data}_S$ denote the information stored in a node about itself and $\texttt{Data}_R$ denote the information stored in a node about its right subtree. $\texttt{Data}_L$, $\texttt{Data}_S$ and $\texttt{Data}_R$ are all generally extracted from a node's `Data`.

To make this more concrete, remember that above we defined

$$node \rightarrow \texttt{Data} = \begin{bmatrix} \Delta_L & \Delta & \Delta_R \\ \alpha_L & \alpha & \alpha_R \\ [min, max] & h & idx \end{bmatrix}$$

So, we could define

$$\texttt{Data}_L = \begin{bmatrix} \Delta_L \\ \alpha_L \end{bmatrix}$$
$$\texttt{Data}_S = \begin{bmatrix} \Delta \\ \alpha \end{bmatrix} \text{ and } \begin{bmatrix} [min, max] & h & idx \end{bmatrix}$$
$$\texttt{Data}_R = \begin{bmatrix} \Delta_R \\ \alpha_R \end{bmatrix}$$

Next, we can define the act of operating, Op, at any node to mean performing an list of actions at that node that takes constant time. For example, Op can be modifying any of the items stored within a node's `Data`, or it could be a constant time comparison between items, or anything else that takes constant time.

Inductive proof

(i) Inductive hypothesis: Suppose that for LH trees of height k, where $k \geq 1$, and $k = log(n) - 1$, performing any constant time operation within any given subsequence will take at most $O(log(n))$ visits, with the following 2 cases:

   (1) If have not split (where a split denotes the need to descend down both children), $2k - 1$ visits will be needed in the worst case. Additionally, we know that there can only be 1 such splits where both the left and right children must be visited, as the sequence we are operating on must be contiguous.

   (2) If we have split earlier, within an ancestor of a node, not including itself, k visits will be needed in the worst case. This is because we know that if we have split earlier on, the entirety of either our current node's left or right subtree will be included within the sequence. So, we need to descend down to only one of a given node's children in the worst case. Each time we descend down to one of the children, we divide the number of potential nodes we can visit by 2, so at worst we can only visit k nodes.

   Both cases take a number of visits proportional to k, which is defined to be $log(n) - 1 = O(log(n))$.

(ii) Base case, $k = 1$. Remember, this means that there are 3 items in our sequence. Therefore, there are the following 6 possibilities.

   (1) When we query `Operation`$(M, 0, 0, o)$, we visit our root node, notice that the ending index requested is less than the index of our root, so we can operate on $\texttt{Data}_L$. This takes $1 \leq 1$ visits.

(2) When we query `Operation`$(M, 1, 1, o)$, we visit our root node, notice that the bounding indices cover only the root node, so we can operate on $\texttt{Data}_S$. This takes $1 \leq 1$ visits.

(3) When we query `Operation`$(M, 2, 2, o)$, we visit our root node, notice that the starting index is greater than the index of our root, so we return we can operate on $\texttt{Data}_R$. This takes $1 \leq 1$ visits.

(4) When we query `Operation`$(M, 0, 1, o)$ we visit our root node, notice that our starting index is less than the index of our root, but the ending index is equal to our root. So, we don't need to look at the right subtree, and only need to operate on $\texttt{Data}_L$ and $\texttt{Data}_S$. This takes $1 \leq 1$ visits.

(5) When we query `Operation`$(M, 1, 2, o)$, we visit our root node, notice that our starting index is equal to the index of the root node, but the ending index is greater than the index of the root. So, we don't need to look at the left subtree, and only need to operate on $\texttt{Data}_S$ and $\texttt{Data}_R$. This takes $1 \leq 1$ visits.

(6) When we query `Operation`$(M, 0, 2, o)$ we notice that this encompasses our entire LH tree, as the length of this sequence is equal to $size = 2^{h+2} - 1 = 2^2 - 1 = 3$. So, we can operate on $\texttt{Data}_L$, $\texttt{Data}_S$ and $\texttt{Data}_R$. This takes $1 \leq 1$ visits.

(iii) Inductive reasoning: Suppose we have a LH tree of height $k + 1$. We can split our LH tree into three parts: $\{M_L, r, M_R\}$. As above, we let i denote the starting index and j denote the ending index. Define $size = 2^{(k+1)+1} - 1 = 2^{k+2} - 1$. We have the following cases:

(1) Case 1: $i = 0$ and $j = size$. We operate on $\texttt{Data}_L$, `Data` and $\texttt{Data}_R$. This takes 1 visit.

(2) Case 2: We split at the root, and descend down both subtrees: $0 < i < root \to idx$ and $root \to idx < j < size$ We visit our root node, and also descend down our left and right subtrees. We know we can only split once, so we cannot split in either of M_L and M_R. Since M_L and M_R are both LH trees of height k, and we cannot split, at worst $k = log(n) - 1$ visits will be needed for each of these subtrees, from our inductive hypothesis. So, this will take overall $2k+1 \leq 2(k+1)-1$ visits total at worst.

(3) Case 3: We descend down the left subtree: $i \leq root \to idx$ and $j \leq root \to idx$, meaning we first descend down the left subtree. Since our left child is a LH tree of height k, and we haven't split yet, this will take at worst $2k - 1$ visits within the left subtree based on our inductive hypothesis. So, this will take at worst $1 + 2k - 1 = 2k < 2(k + 1) - 1$ visits.

(4) Case 4: We descend down the right subtree: $i \geq root \to idx$ and $j \geq root \to idx$, meaning we first descend down the right subtree. Since our right child is a LH tree of height k, and we haven't split yet, this will take at worst $2k - 1$ visits within the right subtree based on our inductive hypothesis. So, this will take at worst $1 + 2k - 1 = 2k < 2(k + 1) - 1$ visits.

Therefore, we have shown that any given `Operation`(M, i, j) will take at worst $2k - 1$ visits for any given LH Tree of height k. From above we defined $k = \log(n) - 1$, where n is the number of nodes in our sequence, so at worst `Operation` would need to visit $O(\log(n))$ nodes, therefore taking $O(\log(n))$ runtime as each visit and Op takes $O(1)$ or constant time.

From the above proof, we can define the following.

Lemma 1
For LH trees of height k, performing a constant time operation, Op, on any given subsequence will take $O(log(n))$ time.

Part a

We initialize our LH tree of height $\log(n) - 1$. To intialize from a sequence, our LH tree nodes will store the given data from the initial sequence, and the min values at each node will be updated as needed. At the moment, we don't care about how long this initialization process takes.

As we do not need to support `Shift` nor `Scale`, all the values in the sequence will be the static.

Operation Definitions and Proof

Again, we can call our data structure M. We will utilize the logic for visits that we defined above to prove that `Minimum`(M, i, j) on any subsequence will take $O(\log(n))$ time.

We can define a base operation Op_0, which is used to figure out what parts of a subtree is covered by an interval to be:

1. A calculation of $size = 2^{k+2} - 1$.
2. A comparison between any of the following: $node \to idx$ or i or j or $size$ to learn which way to descend down. These are used for the cases defined when proving lemma 1.

We can define our operation Op_{min} for minimum specifically to be as following:

First, if needed, a comparison between any of the following: min of the node, the min of either child, and the calulated value at a node, which is $(node \to \Delta) \cdot (node \to \alpha)$, plus the accumulated values from the ancestors of a node.

Then, to deal with min values within the children and their subtrees, we have the following:

1. If the entire subtree is encompassed, we can just return $node \to min$.
2. If the sequence includes only the left or only the right subtree, we descend down to that child and it is no longer this node's problem.
3. If the sequence excludes only the left or right subtree, meaning it includes all of either child and the node, we compare the node's value to the min of the given subtree, and decide if we return or if we continue descending.
4. If the interval only partially includes one or more of the subtrees, we descend down to the respective subtrees needed to find their minimum values within modified intervals (for the left, the end will be $node \to idx - 1$, and for the right, the start will be $node \to idx + 1$, instead of what they were before), and compare those returned values in the end to find the true minimum.

These actions all take constant time.

Now we can use lemma 1 as proven above with the combination of $Op_{min^{\Sigma}} = [Op_0, Op_{min}]$ as defined above. Using Lemma 1, we have shown that `Minimum`(M, i, j) takes $O(\log(n))$ time.

Python Implementation

```python
def min_in(tree, start, end):
    def h(curr, t):
        return t.apply(curr.transform).val()
    def l(curr, t):
        if curr.left is not None:
            return curr.left.min
        return t.apply(curr.transform_left).val()
    def r(curr, t):
        if curr.right is not None:
            return curr.right.min
        return t.apply(curr.transform_right).val()
    def a(curr, arr, t):
        return min(arr)
    op = Op()
    op.here, op.left, op.right, op.after = h, l, r, a
    # now we simply call apply to apply our operation over the correct interval
    return apply(tree, start, end, op, Transform())
```

Part b

Our data structure stays the same, and the definition for `Min` stays the same; However, we will define additional functionality on how to utilize the $node \rightarrow \Delta_L, node \rightarrow \Delta$ and $node \rightarrow \Delta_R$ values to perform `Shift` on arbitrary subsequences.

Changes and Definitions from Above

We will take our definition of Op_0 as defined above, in part a.

Operation Definitions and Proof

We can call our data structure M. We will utilize the logic for visits that we defined above to prove that `Shift`(M, i, j, Δ) on any subsequence will take $O(\log(n))$ time.

We define our operation Op_{shift} to be adding Δ to any of $node \rightarrow \Delta_L, node \rightarrow \Delta$, and $node \rightarrow \Delta_R$ at a certain node.

To determine which operations on $node \rightarrow \Delta_L, node \rightarrow \Delta$, and $node \rightarrow \Delta_R$ we should perform, we can see as follows:

1. If the entire subtree is encompassed, we can add Δ to $node \rightarrow \Delta_L, node \rightarrow \Delta$, and $node \rightarrow \Delta_R$
2. If the sequence includes only the left or only the right subtree, we add Δ to the either Δ_L or Δ_R, whichever corresponds to the subtree the sequence covers.
3. If the sequence excludes only the left or right subtree, meaning it includes all of either child and the node, we add Δ to $node \rightarrow \Delta$ and to either Δ_L or Δ_R, whichever corresponds to the subtree the sequence covers.
4. If the interval only partially includes one or more of the subtrees, we descend down to the respective subtrees needed to modify the shift factors of nodes within modified intervals (for the left, the end will be $node \rightarrow idx - 1$, and for the right, the start will be $node \rightarrow idx + 1$, instead of what they were before).

As we follow the path of nodes taken back up the tree, we modify the $node \rightarrow min$ and $node \rightarrow max$ values to ensure they are still accurate after the `Shift`. This can be thought of as doing a new constant time operation, Op_{fix} on a new iteration of traversing down the tree, and modifying the $node \rightarrow min$ as needed, saved from the earlier descent from a `Shift`.

These actions all take constant time.

Now we can use lemma 1 as proven above with the combination of $Op_{shift^{\Sigma}} = [Op_0,\ Op_{shift}, Op_{fix}]$ as defined above. Using Lemma 1, we have shown that `Shift`(M, i, j, Δ) takes $O(\log(n))$ time.

So, our LH tree now supports both `Minimum` and `Shift` in $O(\log(n))$ time.

Python Implementation

```
def shift(tree, start, end, delta):
    def h(curr, t):
        curr.transform += delta # shift the current node
    def l(curr, t):
        curr.transform_left += delta # shift all of the left subtree
        if curr.left is not None: # update extrema accordingly
            curr.left.min += delta
            curr.left.max += delta
    def r(curr, t):
        curr.transform_right += delta
        if curr.right is not None:
            curr.right.min += delta
            curr.right.max += delta
    def a(curr, _, t): # at the end we want the extrema to reflect that of the entire
    ↪   subtree
        if curr.height == 0:
            vals = [ t.apply(curr.transform).val(), t.apply(curr.transform_left).val(),
            ↪   t.apply(curr.transform_right).val() ]
            curr.min = min(vals)
            curr.max = max(vals)
        else:
            curr.min = min([ t.apply(curr.transform).val(), curr.right.min, curr.left.min
            ↪   ])
            curr.max = max([ t.apply(curr.transform).val(), curr.right.max, curr.left.max
            ↪   ])

    op = Op()
    op.here, op.left, op.right, op.after = h, l, r, a
    apply(tree, start, end, op, Transform())
```

Part c

Our data structure stays the same, and the definition for `Min` and `Shift` stays the same; However, we will define additional functionality on how to utilize the $node \to \alpha_L, node \to \alpha$ and $node \to \alpha_R$ values to perform `Scale` on arbitrary subsequences.

Changes and Definitions from Above

First, we modify our definition for Op_{shift} to instead be adding $\frac{\Delta}{node \to \alpha}$ to any of $node \to \Delta_L, node \to \Delta$, and $node \to \Delta_R$ at a certain node. This did not matter before, as $node \to \alpha_L, node \to \alpha$, and $node \to \alpha_R$ were all constant at 1, but does matter now.

We will take our definition of Op_0 as defined above, in part a. We will take our definition of Op_{fix} as defined above, in part b.

Operation Definitions and Proof

We can call our data structure M. We will utilize the logic for visits that we defined above to prove that `Scale`(M, i, j, α) on any subsequence will take $O(\log(n))$ time.

We define our operation Op_{scale} to be multiplying any of $node \to \alpha_L, node \to \alpha$, and $node \to \alpha_R$ at a certain node by α.

To determine which operations on $node \to \alpha_L, node \to \alpha$, and $node \to \alpha_R$ we should perform, we can see as follows:

1. If the entire subtree is encompassed, we can multiply $node \to \alpha_L, node \to \alpha$, and $node \to \alpha_R$ by α.
2. If the sequence includes only the left or only the right subtree, multiply either Δ_L or Δ_R, whichever corresponds to the subtree the sequence covers, by α.
3. If the sequence excludes only the left or right subtree, meaning it includes all of either child and the node, we multiply $node \to \alpha$ and either α_L or α_R, whichever corresponds to the subtree the sequence covers, by α.
4. If the interval only partially includes one or more of the subtrees, we descend down to the respective subtrees needed to modify the scale factors of nodes within modified intervals (for the left, the end will be $node \to idx - 1$, and for the right, the start will be $node \to idx + 1$, instead of what they were before), and it is no longer this node's issue.
5. If α is negative, swap min and max.

These actions all take constant time.

Now we can use lemma 1 as proven above with the combination of $Op_{scale^{\Sigma}} = [Op_0, Op_{scale}, Op_{fix}]$ as defined above. Using Lemma 1, we have shown that `Scale`(M, i, j, α) takes $O(\log(n))$ time.

So, our LH tree now supports `Minimum`, `Shift` and `Scale` in $O(\log(n))$ time.

Python Implementation

```
def scale(tree, start, end, alpha):
    def h(curr, t):
        curr.transform = curr.transform.scale(alpha, t)
    def l(curr, t):
        curr.transform_left = curr.transform_left.scale(alpha, t)
        if curr.left is not None:
            curr.left.min *= alpha
            curr.left.max *= alpha
            if alpha < 0: # if we are scaling by negative, we need to swap extrema
                curr.left.min, curr.left.max = curr.left.max, curr.left.min
    def r(curr, t):
        curr.transform_right = curr.transform_right.scale(alpha, t)
        if curr.right is not None:
            curr.right.min *= alpha
            curr.right.max *= alpha
            if alpha < 0:
                curr.right.min, curr.right.max = curr.right.max, curr.right.min
    def a(curr, _, t):
        if curr.height == 0:
            vals = [ t.apply(curr.transform).val(), t.apply(curr.transform_left).val(),
            ↪  t.apply(curr.transform_right).val() ]
            curr.min = min(vals)
            curr.max = max(vals)
        else:
            curr.min = min([ t.apply(curr.transform).val(), curr.right.min, curr.left.min
            ↪  ])
            curr.max = max([ t.apply(curr.transform).val(), curr.right.max, curr.left.max
            ↪  ])

    op = Op()
    op.here, op.left, op.right, op.after = h, l, r, a
    apply(tree, start, end, op, Transform())
```

Part d

Our data structure stays the same. However, as mentioned before, we will no longer be creating all of the nodes when we initialize the tree. Instead, we will only make the root node. This takes constant time.

We define a new operation Op_{create} to be an operation that creates a node, if it does not exist already. This takes constant time to check if a pointer is null, and constant time to initialize each of the values needed to define a node.

Then, we modify Op_{min^Σ}, Op_{shift^Σ}, and Op_{scale^Σ} as follows, to see if we need to initialize a node that we will be descending to.

$$\begin{aligned} Op_{min^\Sigma} &= [Op_0, Op_{create}, Op_{min}] \\ Op_{shift^\Sigma} &= [Op_0, Op_{create}, Op_{shift}, Op_{fix}] \\ Op_{scale^\Sigma} &= [Op_0, Op_{create}, Op_{scale}, Op_{fix}] \end{aligned}$$

Now we can use lemma 1 as proven above with each of $Op_{min^\Sigma}, Op_{shift^\Sigma}$ and Op_{scale^Σ}. Using Lemma 1, we have shown that each of these take $O(\log(n))$ time in our LH Tree.

So, our LH tree now supports `Minimum`, `Shift` and `Scale` in $O(\log(n))$ time, and `Init` in $O(1)$ time.

Sources

1. Ben Fauman (BugsJelly), for sanity checks during CS 225 lab.
2. Naveed, for the discarded idea of ”Mad Rizz” Trees.