Augmenting Data Structures Jan 22

1. Definition:
   An augmented data structure is simply an existing data structure modified to store additional info and/or to perform additional operations.

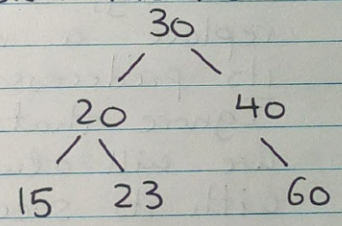
2. Adding rank and select:

   We want to add 2 operations, rank(k) and select(r).

   rank(k): Given a key k, this returns its rank (I.e. It's position among the elements in a data structure).

   Select(r): Given a rank r, this returns the key that has this rank.

   E.g. Consider the tree below.



15 has rank 1. → Select(1) = 15
20 has rank 2. → Select(2) = 20
23 has rank 3. → Select(3) = 23
30 has rank 4. → Select(4) = 30
40 has rank 5. → Select(5) = 40
60 has rank 6. → Select(6) = 60

There are 3 ways we can do this.

1. Use AVL Trees Without Modification:

   In this way, to find the rank of a node, do an in-order traversal of the tree, keeping track of the number of nodes visited, until the desired node is reached.

   Likewise, to find the select of a rank, do an in-order traversal of the tree, keeping track of the number of nodes visited, until the desired node is reached.

   Even though search, insert and delete won't be affected, the worst time-complexity for rank and search is $\Theta(n)$ because we may have to visit every node. This is very inefficient.

2. Augment Each Node With an Additional Field rank[x] That Stores x's Rank in the Tree:

   Now, both rank and search will take $\Theta(\log n)$. However, insert and delete will take $\Theta(n)$. This is because each time we add and delete a node, we have to update the rank field for all the nodes that come after it in

an in-order traversal. Now, rank and select are efficient but not insert or delete.

3. Augment Each Node With an Additional Field size[x] That Stores the Number of keys in the Subtree Rooted at x and Including x:

We know that Rank(x) = 1+ number of keys that come before x in the tree.

Let T_L be the left child of x.

The relative Rank(x) is equal to Size(T_L) +1.

This means that the rank of a node is related to the size of the subtrees rooted at neighbouring nodes.

Given a key k, to find the rank of k, we search for k, keeping track of the rank of the current node. Whenever we go down a right path, we add the size of the left subtree that we skipped and the key itself that we skipped.

When we find the key, to get its true rank, we add the size of its left child +1 to its current rank.

Finding the key with rank r:

- Let x be root(T). Start at the root and work down.
- Let S be the left child. Compare the given rank, r, to size[S]+1.
- If they are equal, return x.
- If (r < size[S]+1), we know that the element we are looking for is in S, so call the routine recursively on S.
- If (r > size[S]+1), then we know that the node we are looking for is in the right subtree. Therefore, the relative rank in the remaining elements (I.e. Ignoring S) is equal to r - (size[S]+1). We change r accordingly and go down the right subtree of S.

3. Complexity:

- The complexity for rank is the same as search. $O(\log n)$
- For insert and delete, there are 2 parts. First, there is the operation, and then there is the rebalancing.

Operations:

1. Insert(x): For each node visited when finding the position for x, increment its size.

2. Delete(x): If x is a leaf, then we traverse the path from x to the root and we decrement the size of each node on the path. If x is not a leaf, we replace it with its successor, y. Then, we traverse the path from y to the root, decrementing each node along the path.

Rotations:

- Each rotation, we only need to consider a constant number of nodes, so it takes $\Theta(1)$ time.

$\therefore$ Each operation takes $\Theta(\log n)$ time in the worst-case.

4. Interval Trees

1. Background and Operations:

- Closed Time Interval:
  $\{x \in R \mid \ell \leq x \leq h\} = [\ell, h]$

- Operations:

1. insert(ℓ, h): Store $[\ell, h]$ in the collection.

2. delete(ℓ, h): Deletes $[\ell, h]$.

3. search(ℓ, h): Return a stored interval that overlaps with $[\ell, h]$.

Note: Given the interval $[1, 5]$, $[-1, 1]$, $[0, 10]$, $[5, 8]$, $[-1, 5]$, $[0, 5]$ and $[1, 6]$ overlaps with it. However, $[-1, 0]$, $[6, 10]$ and $[8, 13]$ does not overlap with $[1, 5]$.

Note: To compare 2 intervals, $[a, b]$ and $[\ell, h]$:

1. If $a < \ell$, then $[a, b] < [\ell, h]$.
2. If $a = \ell$ and $b < h$, then $[a, b] < [\ell, h]$.

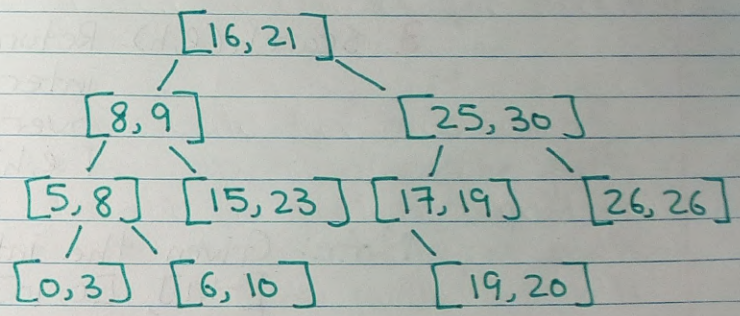
Hilroy

E.g. $[1,3] < [2,5]$
E.g. $[2,3] < [2,5]$

Although this is enough for insert and delete, we need more info for search to work.

First, each node x_i stores l_i and h_i, the interval's two ends, as the key. However, this is not enough, still.

Consider the interval tree below.



Suppose we want to do search(10,12). Consider [8,9]. [10,12] doesn't overlap with [8,9] and lies to the right of [8,9]. If [10,12] were to overlap with an interval in the left subtree of [8,9], there must be some $h_i \geq 10$. If there is an $h_i \geq 10$ in the left subtree, then it is guaranteed that it overlaps with [10,12]. We see that [6,10] overlaps with [10,12] in the left subtree.

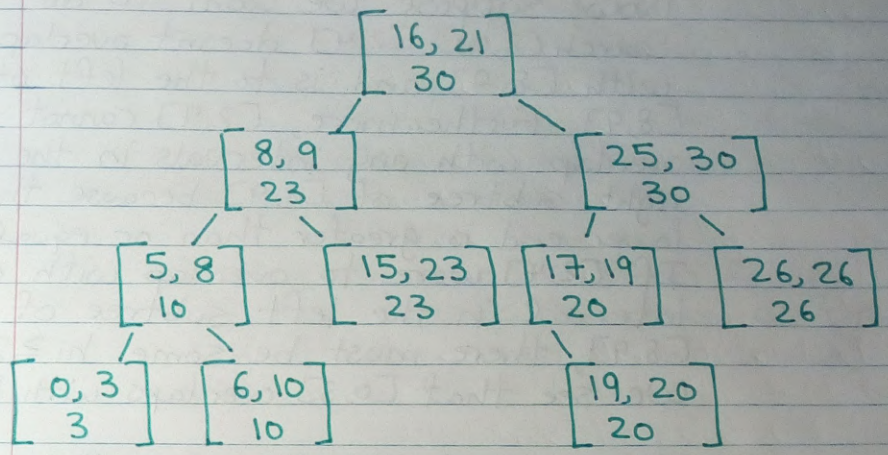
Now, suppose we want to do search $(2,4)$. $[2,4]$ doesn't overlap with $[8,9]$ and is to the left of $[8,9]$. Furthermore, $[2,4]$ cannot overlap with any intervals in the right subtree of $[8,9]$ because their lower end is greater than or equal to 8. If $[2,4]$ were to overlap with an interval in the left subtree of $[8,9]$, there must be some $hi \geq 2$. We see that $[0,3]$ overlaps with $[2,4]$.

Notice that in both cases, there needed to be an $hi \geq \ell$ in order for there to exist an overlapping interval in the left subtree.

2. Augmenting the Tree:

We can augment the tree by adding the max hi in the subtree rooted at x to each node x.

I.e. For each node x, we add the max hi of the subtree rooted at x.



The augmented version of the previous tree.

3. Pseudo-Code For Search:

Let x be the root of the tree.

```
Search(x, ℓ, h):
    if x=null:          There is no overlapping
        return null     interval.

    Check if x's interval overlaps
    with [ℓ, h].
    if x.ℓ ≤ h and ℓ ≤ x.h:
        return [x.ℓ, x.h]

    if h < x.ℓ or x.h < ℓ:
        if x.left == null:
            return search(x.r, ℓ, h)
```

```
elif ℓ > x.left.max:
    return search(x.right, ℓ, h)
else:
    return search(x.left, ℓ, h)
```

4. Explaining the Pseudo-Code:

First, we check if the root is null. If it is, then there can be no overlap.

Next, if the root isn't null, we compare the root's interval with the inputted interval. We need $x.\ell \leq h$ and $\ell \leq x.h$. If only one of the two requirements is met, it wouldn't work.

Finally, if $h < x.\ell$ or $x.h < \ell$, then we compare $[\ell, h]$ to x's children. If the left subtree is empty or if ℓ is greater than the max h of the left subtree, we go to the right subtree. Otherwise, we go to the left subtree.

I.e. Consider the 2 cases below.

Case 1: We go down the right subtree. Then, one of the following must be true.

1. There is an overlap in the right subtree.
2. There is no overlap in either subtree. We go to the right subtree only when the left is null or $\ell > x.left.max$.

Case 2: We go down the left subtree. Then, one of the following must be true.

1. There is an overlap in the left subtree.
2. There is no overlap in either tree.

Now, consider these facts:

1. We went to the left subtree because $\ell \leq x.left.max$.
2. $x.left.max$ is an hi in one of the intervals in the left subtree. Let that interval be $[a, x.left.max]$.
3. Since $[\ell, h]$ doesn't overlap with any node in the left subtree, h must be smaller than a.
4. All nodes are ordered by the low value, ℓi. This means that all nodes in x's right subtree must be greater than a.

From these facts, we can deduce that $[\ell, h]$ cannot overlap with any interval in x's right subtree.

5. Weight-Balanced BST

1. Background:

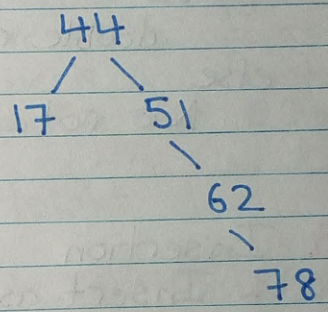
- A weight-balanced BST is another way to achieve $O(\log n)$ tree height.
- All weight-balanced BSTs have this property:

  At every node v,
  1. $\text{weight}(v.\text{left}) \leq \text{weight}(v.\text{right}) \times 3$
  2. $\text{weight}(v.\text{right}) \leq \text{weight}(v.\text{left}) \times 3$

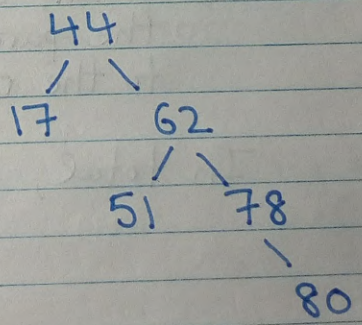
  where $\text{weight}(v) = \text{size}(v) + 1$

- Examples:

1.



2.



2. Balancing and Rotation

For each node v on the path from the new/deleted node back to the root:

```
if   w(v.e)x3 < w(v.r):
    let x = v.right
    if w(x.e) < w(x.r)x2:
        single rotation ccw
    else
        double rotation cw then ccw
else if w(v.e) > w(v.r)x3:
    let x = v.left
    if w(x.e)x2 > w(x.r):
        single rotation cw
    else
        double rotation ccw then cw
else
    No rotation
```

3. Insertion

1. Insert as an AVL Tree.
2. Check and fix balance. $\Theta(\lg n)$ nodes
   Then, update the size from the parent of the new node to the root. $\Theta(1)$ time per node

In total, it takes $\Theta(\lg n)$ time.

4. Deletion

1. Find the node that has the key
   $\Theta(\lg n)$ time
2. If the node is a leaf, remove and update the balance starting at the node's parent and up to the root.
   $\Theta(1)$ time
3. Else:
   a) Replace the key of the node with its successor key.
   $\Theta(\lg n)$ time
   b) Successor's parent adopts successor's right child. $\Theta(1)$ time
   c) From the adopter to the root, check and fix balance and update size. $\Theta(\lg n)$ time

5. WBT Height

Claim: $h(T) \leq c \times \lg(\text{size}(T)+1)$ where $c = \frac{1}{\lg(4/3)}$ and $h(T)$ is the height of T.

Proof:
We can do a proof by induction.

Base Case:

T is empty.

$h(T) = h(\text{empty})$

$= 0$

$c \times \lg(\text{size}(T)+1)$

$= c \times \lg(\text{size}(\text{empty})+1)$

$= c \times \lg(0+1)$

$= c \times \lg(1)$ Note: $\log(1) = 0$

$= 0$

$\therefore h(T) \leq c \times \lg(\text{size}(T)+1)$, as wanted

Induction Hypothesis:

Suppose for all $k \in \mathbb{N}$ where $0 \leq k < n$, the height of every weight-balanced BST of size k is at most $c \times \lg(k+1)$.

Induction Step:

Let $n_l = \text{size}(T.\text{left})$

Let $n_r = \text{size}(T.\text{right})$

$n = n_l + n_r + 1$

$n+1 = n_l + n_r + 1 + 1$

$\geq \frac{(n_r+1)}{3} + n_r + 1$ T is balanced

$= \frac{4n_r+4}{3}$

$= \frac{4}{3}(n_r+1)$

$n_r + 1 \leq \frac{3}{4}(n+1)$

Without loss of generality, assume $h(T.\text{left}) \leq h(T.\text{right})$.

$$
\begin{aligned}
h(T) &= 1 + h(T.\text{right}) \\
&\leq 1 + c \times \lg(n_r + 1) \quad \text{By I.H.} \\
&\leq 1 + c \times \lg\left(\tfrac{3}{4}(n+1)\right) \\
&= 1 + c\left(\lg\left(\tfrac{3}{4}\right) + \lg(n+1)\right) \\
&= 1 + (-1) + c \times \lg(n+1) \\
&= c \times \lg(\text{size}(T)+1), \text{ as wanted}
\end{aligned}
$$