Lecture 7: Binary Trees II: AVL

Last Time and Today's Goal

Sequence Data Structure	Operations $O(\cdot)$				
	Container	Static	Dynamic		
	`build(X)`	`get_at(i)` `set_at(i,x)`	`insert_first(x)` `delete_first()`	`insert_last(x)` `delete_last()`	`insert_at(i, x)` `delete_at(i)`
Binary Tree	n	h	h	h	h
AVL Tree	n	$\log n$	$\log n$	$\log n$	$\log n$

Set Data Structure	Operations $O(\cdot)$				
	Container	Static	Dynamic	Order	
	`build(X)`	`find(k)`	`insert(x)` `delete(k)`	`find_min()` `find_max()`	`find_prev(k)` `find_next(k)`
Binary Tree	$n \log n$	h	h	h	h
AVL Tree	$n \log n$	$\log n$	$\log n$	$\log n$	$\log n$

Height Balance

- How to maintain height $h = O(\log n)$ where n is number of nodes in tree?
- A binary tree that maintains $O(\log n)$ height under dynamic operations is called **balanced**
  - There are many balancing schemes (Red-Black Trees, Splay Trees, 2-3 Trees, ...)
  - First proposed balancing scheme was the **AVL Tree** (Adelson-Velsky and Landis, 1962)

Rotations

- Need to reduce height of tree without changing its traversal order, so that we represent the same sequence of items
- How to change the structure of a tree, while preserving traversal order? **Rotations!**

```
         ______<D>__       rotate_right(<D>)      __<B>______
      __<B>___     <E>            =>             <A>      __<D>__
     <A>    <C>    / \                           / \     <C>    <E>
     / \    / \   /___\           <=            /___\    / \    / \
    /___\  /___\               rotate_left(<B>)          /___\  /___\
```

- A rotation relinks $O(1)$ pointers to modify tree structure and maintains traversal order

Rotations Suffice

- **Claim:** $O(n)$ rotations can transform a binary tree to any other with same traversal order.
- **Proof:** Repeatedly perform last possible right rotation in traversal order; resulting tree is a canonical chain. Each rotation increases depth of the last node by 1. Depth of last node in final chain is $n-1$, so at most $n-1$ rotations are performed. Reverse canonical rotations to reach target tree. $\square$
- Can maintain height-balance by using $O(n)$ rotations to fully balance the tree, but slow :(
- We will keep the tree balanced in $O(\log n)$ time per operation!

AVL Trees: Height Balance

- AVL trees maintain **height-balance** (also called the **AVL Property**)
  - A node is **height-balanced** if heights of its left and right subtrees differ by at most 1
  - Let **skew** of a node be the height of its right subtree minus that of its left subtree
  - Then a node is height-balanced if its skew is $-1, 0$, or 1

- **Claim:** A binary tree with height-balanced nodes has height $h = O(\log n)$ (i.e., $n = 2^{\Omega(h)}$)
- **Proof:** Suffices to show fewest nodes $F(h)$ in any height h tree is $F(h) = 2^{\Omega(h)}$

  $$F(0) = 1,\ F(1) = 2,\ F(h) = 1+F(h-1)+F(h-2) \geq 2F(h-2) \implies F(h) \geq 2^{h/2} \quad \square$$

- Suppose adding or removing leaf from a height-balanced tree results in imbalance
  - Only subtrees of the leaf's ancestors have changed in height or skew
  - Heights changed by only ± 1, so skews still have magnitude ≤ 2
  - **Idea:** Fix height-balance of ancestors starting from leaf up to the root
  - Repeatedly rebalance lowest ancestor that is not height-balanced, wlog assume skew 2

- **Local Rebalance:** Given binary tree node <B>:
  - whose skew 2 and
  - every other node in <B>'s subtree is height-balanced,
  - then <B>'s subtree can be made height-balanced via one or two rotations
  - (after which <B>'s height is the same or one less than before)

- **Proof:**
  - Since skew of <B> is 2, <B>'s right child <F> exists
  - **Case 1:** skew of <F> is 0 or **Case 2:** skew of <F> is 1
    * Perform a left rotation on <B>

```
     __<B>______                  ______<F>____
    <A>     ___<F>___           __<B>___       <G>
    / \   <D>      <G>    =>   <A>    <D>      / \
   /___\  / \      / \         / \    / \     /   \
         /___\    /   \       /___\  /___\   /_____\
        /_____\  /_____\            /_____\
```

    * Let $h = \text{height}(\texttt{<A>})$. Then $\text{height}(\texttt{<G>}) = h + 1$ and $\text{height}(\texttt{<D>})$ is $h + 1$ in Case 1, h in Case 2
    * After rotation:
      - the skew of <B> is either 1 in Case 1 or 0 in Case 2, so <B> is height balanced
      - the skew of <F> is -1, so <F> is height balanced
      - the height of <B> before is $h + 3$, then after is $h + 3$ in Case 1, $h + 2$ in Case 2

  - **Case 3:** skew of <F> is -1, so the left child <D> of <F> exists
    * Perform a right rotation on <F>, then a left rotation on <B>

```
     __<B>___________                    _____<D>______
    <A>          _____<F>__           __<B>__      __<F>__
    / \      __<D>__     <G>     =>  <A>   <C>    <E>   <G>
   /___\   <C>    <E>    / \         / \   /_\    /_\   / \
           /_\    /_\   /___\       /___\ /___\  /___\ /___\
          /___\  /___\
```

    * Let $h = \text{height}(\texttt{<A>})$. Then $\text{height}(\texttt{<G>}) = h$ while $\text{height}(\texttt{<C>})$ and $\text{height}(\texttt{<E>})$ are each either h or $h - 1$
    * After rotation:
      - the skew of <B> is either 0 or -1, so <B> is height balanced
      - the skew of <F> is either 0 or 1, so <F> is height balanced
      - the skew of <D> is 0, so D is height balanced
      - the height of <B> is $h + 3$ before, then after is $h + 2$

- **Global Rebalance:** Add or remove a leaf from height-balanced tree T to produce tree T'. Then T' can be transformed into a height-balanced tree T'' using at most $O(\log n)$ rotations.
- **Proof:**
  - Only ancestors of the affected leaf have different height in T' than in T
  - Affected leaf has at most $h = O(\log n)$ ancestors whose subtrees may have changed
  - Let <X> be lowest ancestor that is not height-balanced (with skew magnitude 2)
  - If a leaf was added into T:
    * Insertion increases height of <X>, so in Case 2 or 3 of Local Rebalancing
    * Rotation decreases subtree height: balanced after one rotation
  - If a leaf was removed from T:
    * Deletion decreased height of one child of <X>, not <X>, so only imbalance
    * Could decrease height of <X> by 1; parent of <X> may now be imbalanced
    * So may have to rebalance every ancestor of <X>, but at most $h = O(\log n)$ of them
- So can maintain height-balance using only $O(\log n)$ rotations after insertion/deletion!
- But requires us to evaluate whether possibly $O(\log n)$ nodes were height-balanced

---

Computing Height

- How to tell whether node <X> is height-balanced? Compute heights of subtrees!
- How to compute the height of node <X>? Naive algorithm:
  - Recursively compute height of the left and right subtrees of <X>
  - Add 1 to the max of the two heights
  - Runs in $\Omega(n)$ time, since we recurse on every node :(
- **Idea:** Augment each node with the height of its subtree! (Save for later!)
- Height of <X> can be computed in $O(1)$ time from the heights of its children:
  - Look up the stored heights of left and right subtrees in $O(1)$ time
  - Add 1 to the max of the two heights
- During dynamic operations, we must **maintain** our augmentation as the tree changes shape
- Recompute subtree augmentations at every node whose subtree changes:
  - Update relinked nodes in a rotation operation in $O(1)$ time (ancestors don't change)
  - Update all ancestors of an inserted or deleted node in $O(h)$ time by walking up the tree

Steps to Augment a Binary Tree

- In general, to augment a binary tree with a **subtree property** `P`, you must:
  - State the subtree property `P(<X>)` you want to store at each node `<X>`
  - Show how to compute `P(<X>)` from the augmentations of `<X>`'s children in $O(1)$ time
- Then stored property `P(<X>)` can be maintained without changing dynamic operation costs

---

Application: Sequence

- For sequence binary tree, we needed to know subtree **sizes**
- For just inserting/deleting a leaf, this was easy, but now need to handle rotations
- Subtree size is a subtree property, so can maintain via augmentation
  - Can compute size from sizes of children by summing them and adding 1

Conclusion

- Set AVL trees achieve $O(\lg n)$ time for all set operations,
  except $O(n \log n)$ time for build and $O(n)$ time for iter
- Sequence AVL trees achieve $O(\lg n)$ time for all sequence operations,
  except $O(n)$ time for build and iter

Application: Sorting

- Any Set data structure defines a sorting algorithm: build (or repeatedly insert) then iter
- For example, Direct Access Array Sort from Lecture 5
- AVL Sort is a new $O(n \lg n)$-time sorting algorithm

MIT OpenCourseWare
https://ocw.mit.edu

6.006 Introduction to Algorithms
Spring 2020

For information about citing these materials or our Terms of Use, visit: https://ocw.mit.edu/terms