

Lecture 6: Binary Trees I

Previously

Sequence Data Structure	Operations $O(\cdot)$				
	Container	Static	Dynamic		
	build(A)	get_at(i) set_at(i, x)	insert_first(x) delete_first()	insert_last(x) delete_last()	insert_at(i, x) delete_at(i)
Array	n	1	n	n	n
Linked List	n	n	1	n	n
Dynamic Array	n	1	n	$1_{(a)}$	n
Goal	n	$\log n$	$\log n$	$\log n$	$\log n$

Set Data Structure	Operations $O(\cdot)$				
	Container	Static	Dynamic	Order	
	build(A)	find(k)	insert(x) delete(k)	find_min() find_max()	find_prev(k) find_next(k)
Array	n	n	n	n	n
Sorted Array	$n \log n$	$\log n$	n	1	$\log n$
Direct Access Array	u	1	1	u	u
Hash Table	$n_{(e)}$	$1_{(e)}$	$1_{(a)(e)}$	n	n
Goal	$n \log n$	$\log n$	$\log n$	$\log n$	$\log n$

How? Binary Trees!

- Pointer-based data structures (like Linked List) can achieve **worst-case** performance
- Binary tree is pointer-based data structure with three pointers per node
- Node Representation: `node.{item, parent, left, right}`
- **Example:**

1									
2									
3									
4									
5									

node		<A>				<C>		<D>		<E>		<F>	
item		A		B		C		D		E		F	
parent		-		<A>		<A>						<D>	
left				<C>		-		<F>		-		-	
right		<C>		<D>		-		-		-		-	

Terminology

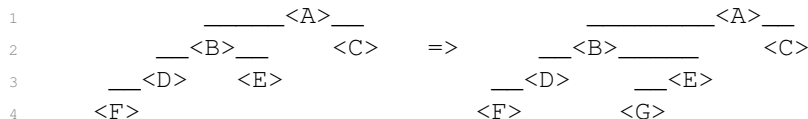
- The **root** of a tree has no parent (**Ex:** $\langle A \rangle$)
 - A **leaf** of a tree has no children (**Ex:** $\langle C \rangle$, $\langle E \rangle$, and $\langle F \rangle$)
 - The **depth** $D(\langle X \rangle)$ of node $\langle X \rangle$ in a tree rooted at $\langle R \rangle$ is length of path from $\langle X \rangle$ to $\langle R \rangle$
 - The **height** $H(\langle X \rangle)$ of node $\langle X \rangle$ is max depth of any node in the subtree rooted at $\langle X \rangle$
 - **Idea:** Design operations to run in $O(h)$ time for root height h , and maintain $h = O(\log n)$
 - A binary tree has an inherent order: its **traversal order**
 - every node in node $\langle X \rangle$'s left subtree is **before** $\langle X \rangle$
 - every node in node $\langle X \rangle$'s right subtree is **after** $\langle X \rangle$
 - List nodes in traversal order via a recursive algorithm starting at root:
 - Recursively list left subtree, list self, then recursively list right subtree
 - Runs in $O(n)$ time, since $O(1)$ work is done to list each node
 - **Example:** Traversal order is $(\langle F \rangle, \langle D \rangle, \langle B \rangle, \langle E \rangle, \langle A \rangle, \langle C \rangle)$
 - Right now, traversal order has no meaning relative to the stored items
 - Next time, assign semantic meaning to traversal order to implement Sequence/Set interfaces
-

Tree Navigation

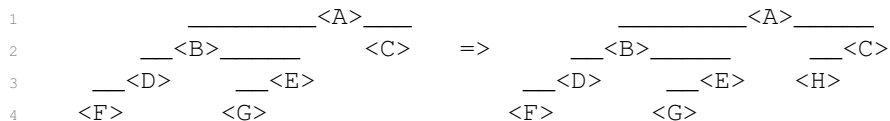
- **Find first** node in the traversal order of node $\langle X \rangle$'s subtree (last is symmetric)
 - If $\langle X \rangle$ has left child, recursively return the first node in the left subtree
 - Otherwise, $\langle X \rangle$ is the first node, so return it
 - Running time is $O(h)$ where h is the height of the tree
 - **Example:** first node in $\langle A \rangle$'s subtree is $\langle F \rangle$
- **Find successor** of node $\langle X \rangle$ in the traversal order (predecessor is symmetric)
 - If $\langle X \rangle$ has right child, return first of right subtree
 - Otherwise, return lowest ancestor of $\langle X \rangle$ for which $\langle X \rangle$ is in its left subtree
 - Running time is $O(h)$ where h is the height of the tree
 - **Example:** Successor of: $\langle B \rangle$ is $\langle E \rangle$, $\langle E \rangle$ is $\langle A \rangle$, and $\langle C \rangle$ is None

Dynamic Operations

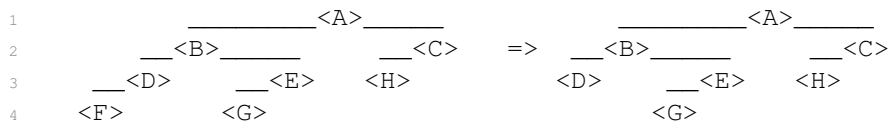
- Change the tree by a single item (only add or remove leaves):
 - add a node before another in the traversal order (after is symmetric)
 - remove an item from the tree
- **Add** node $\langle Y \rangle$ before node $\langle X \rangle$ in the traversal order
 - If $\langle X \rangle$ has no left child, make $\langle Y \rangle$ the left child of $\langle X \rangle$
 - Otherwise, make $\langle Y \rangle$ the right child of $\langle X \rangle$'s predecessor (which cannot have a right child)
 - Running time is $O(h)$ where h is the height of the tree
 - **Example:** Add node $\langle G \rangle$ before $\langle E \rangle$ in traversal order



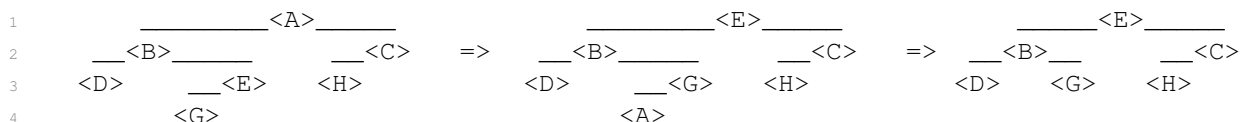
- **Example:** Add node $\langle H \rangle$ after $\langle A \rangle$ in traversal order



-
- **Remove** the item in node $\langle X \rangle$ from $\langle X \rangle$'s subtree
 - If $\langle X \rangle$ is a leaf, detach from parent and return
 - Otherwise, $\langle X \rangle$ has a child
 - * If $\langle X \rangle$ has a left child, swap items with the predecessor of $\langle X \rangle$ and recurse
 - * Otherwise $\langle X \rangle$ has a right child, swap items with the successor of $\langle X \rangle$ and recurse
 - Running time is $O(h)$ where h is the height of the tree
 - **Example:** Remove $\langle F \rangle$ (a leaf)

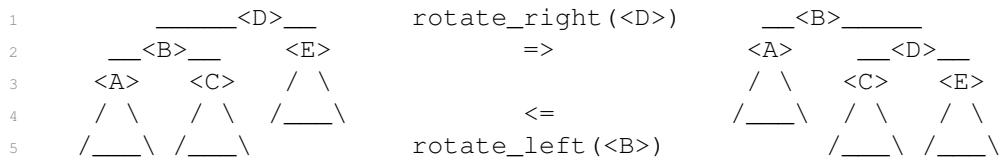


- **Example:** Remove $\langle A \rangle$ (not a leaf, so first swap down to a leaf)



Rotations

- Want trees with small height, i.e., $h = O(\log n)$
- If height grows, need to change tree structure without changing traversal order
- How to change the structure of a tree, while preserving traversal order? **Rotations!**



- A rotation relinks $O(1)$ pointers to modify tree structure and maintains traversal order
- **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 one. 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. □
- Can maintain height-balance by using $O(n)$ rotations to fully balance the tree, but slow :(
- But we want to keep the tree balanced in $O(\log n)$ time!

Next Time

- Keep a binary tree balanced after insertion or deletion
- Implement efficient Set and Sequence Interfaces using a Binary Tree