Lecture 6: Binary Trees I

Previously and New 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)`
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(X)`	`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            _________<A>_____          node   | <A> | <B> | <C> | <D> | <E> | <F> |
2        __<B>_____       <C>           item   |  A  |  B  |  C  |  D  |  E  |  F  |
3     __<D>      <E>                    parent |  -  | <A> | <A> | <B> | <B> | <D> |
4    <F>                                left   | <B> | <C> |  -  | <F> |  -  |  -  |
5                                       right  | <C> | <D> |  -  |  -  |  -  |  -  |
```

Terminology

- The **root** of a tree has no parent (**Ex:** <A>)
- A **leaf** of a tree has no children (**Ex:** <C>, <E>, and <F>)
- Define **depth**(<X>) of node <X> in a tree rooted at <R> to be length of path from <X> to <R>
- Define **height**(<X>) of node <X> to be max depth of any node in the **subtree** rooted at <X>
- **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 <X>'s left subtree is **before** <X>
  - every node in node <X>'s right subtree is **after** <X>
- 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 (<F>, <D>, <B>, <E>, <A>, <C>)
- Right now, traversal order has no meaning relative to the stored items
- Later, assign semantic meaning to traversal order to implement Sequence/Set interfaces

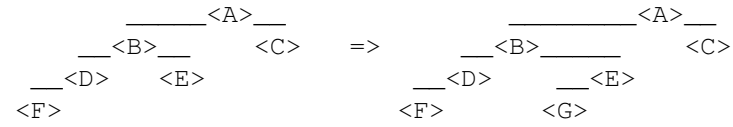
---

Tree Navigation

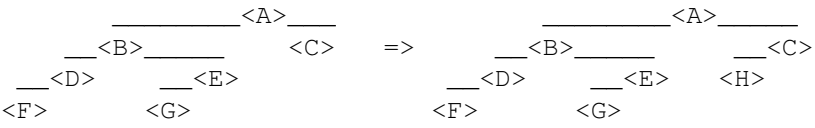
- **Find first** node in the traversal order of node <X>'s subtree (last is symmetric)
  - If <X> has left child, recursively return the first node in the left subtree
  - Otherwise, <X> is the first node, so return it
  - Running time is $O(h)$ where h is the height of the tree
  - **Example:** first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

Dynamic Operations

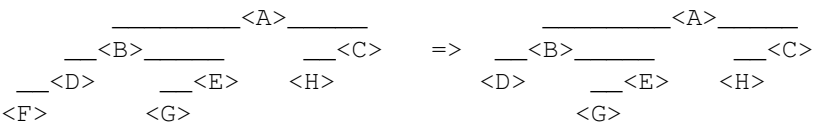
- Change the tree by a single item (only add or remove leaves):
  - add a node after another in the traversal order (before is symmetric)
  - remove an item from the tree
- **Insert** node <Y> after node <X> in the traversal order
  - If <X> has no right child, make <Y> the right child of <X>
  - Otherwise,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



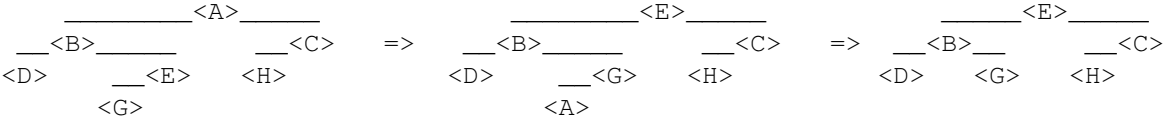
- **Example:** Insert node <H> after <A> in traversal order



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



  - **Example:** Remove <A> (not a leaf, so first swap down to a leaf)



Application: Set

- **Idea! Set Binary Tree** (a.k.a. **Binary Search Tree / BST**):
  Traversal order is sorted order increasing by key
  - Equivalent to **BST Property**: for every node, every key in left subtree $\leq$ node's key $\leq$ every key in right subtree
- Then can find the node with key k in node `<X>`'s subtree in $O(h)$ time like binary search:
  - If k is smaller than the key at `<X>`, recurse in left subtree (or return `None`)
  - If k is larger than the key at `<X>`, recurse in right subtree (or return `None`)
  - Otherwise, return the item stored at `<X>`
- Other Set operations follow a similar pattern; see recitation

---

Application: Sequence

- **Idea! Sequence Binary Tree**: Traversal order is sequence order
- How do we find i^{th} node in traversal order of a subtree? Call this operation `subtree_at(`i`)`
- Could just iterate through entire traversal order, but that's bad, $O(n)$
- However, if we could compute a subtree's **size** in $O(1)$, then can solve in $O(h)$ time
  - How? Check the size n_L of the left subtree and compare to i
  - If $i < n_L$, recurse on the left subtree
  - If $i > n_L$, recurse on the right subtree with $i' = i - n_L - 1$
  - Otherwise, $i = n_L$, and you've reached the desired node!
- Maintain the size of each node's subtree at the node via **augmentation**
  - Add `node.size` field to each `node`
  - When adding new leaf, add $+1$ to `a.size` for all ancestors a in $O(h)$ time
  - When deleting a leaf, add -1 to `a.size` for all ancestors a in $O(h)$ time
- Sequence operations follow directly from a fast `subtree_at(i)` operation
- Naively, `build(X)` takes $O(nh)$ time, but can be done in $O(n)$ time; see recitation

So Far

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
Goal	$n \log n$	$\log n$	$\log n$	$\log n$	$\log n$

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
Goal	n	$\log n$	$\log n$	$\log n$	$\log n$

Next Time

- Keep a binary tree **balanced** after insertion or deletion
- Reduce $O(h)$ running times to $O(\log n)$ by keeping $h = O(\log n)$

6.006	Lecture 6	Feb. 25, 2020

TODAY: Binary Trees I

- binary tree & traversal order
- subtree ops: first, successor, insert, delete
- set binary tree (Binary Search Tree / BST)
- sequence bin. tree via subtree-size augmentation

Goal: one roughly best data structure

↳ up to log factors

O(1) or O(lg n) ←

- linked list good only for insert/delete_first/(last)()
- (dynamic) array good only for
  - seq.: get/set_at() & insert/delete_(first)/last() (with 2 ↑, doubly ↓)
  - set: find(), find_next/prev/min/max() [if sorted]
- hash table good only for insert/delete/find()
- no seq. DS good for insert/delete_at()
- no set DS good for insert/delete() + find_next/prev()

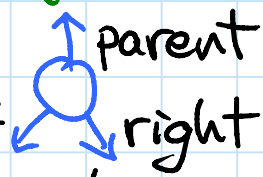
O(lg n) ←

- AVL trees [L7] are good for all these ops.
- binary trees [TODAY] are first step, achieve O(h)

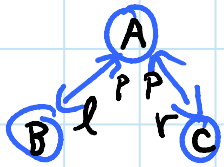
How? we'll build a pointer-based data structure

- problem with linked lists is that most nodes have depth Θ(n)
  ↳ distance from root node (head + tail)
- saw how depth Θ(lg n) is possible: decision tree [L4 & L5]

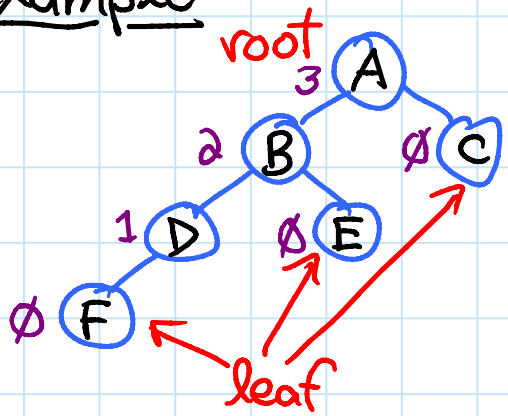
(rooted)
Binary tree: pointer data structure (like linked list)
- each node has 3 pointers: parent, left, right & 1 item



- invariant: node = node.left.parent & node = node.right.parent



- example:



height

node	A	B	C	D	E	F
item	A	B	C	D	E	F
parent	/	A	A	B	B	C
left	B	D	/	F	/	/
right	C	E	/	/	/	/

- root has no parent (unique)
- leaf has no children
- depth(X) = length of path from X up to root (ancestors)
  - length → # edges
  - path → unique
  - depth Ø ⇔ root

- subtree(X) = tree of X & its descendants
  if we cut X from its parent ⇒ root is X
  - subtree(root) = entire tree
  - subtree(leaf) = just itself
- height(X) = max. depth of node within subtree(X) = length of longest downward path
  - height Ø ⇔ leaf
- h = height(root) = max. depth of node (leaf)
  "height of tree"

Traversal order of binary tree: for every node (X)
- every node in (X)'s left subtree is before (X)
- every node in (X)'s right subtree is after (X)

iter_traversal(tree): subtree_iter(tree.root)
subtree_iter(node): - subtree_iter(node.left)
- output node
- subtree_iter(node.right)

O(n) time

↓ traversal order of subtree(node)

- example: (F), (D), (B), (E), (A), (C)
- we'll use traversal order to represent either sequence order or set sorted order

Traversal ops: all O(h) time

subtree_first(node): first node in subtree_iter order
- go left (node = node.left) until this would "fall off tree" (hit None)
- return last node

subtree_last(node): ditto, right ⇒ last in order

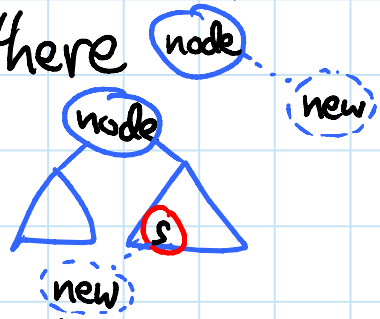
successor(node): next node in traversal order (↗ not necc. within subtree)
- if node.right: subtree_first(node)
- else: - go up (node = node.parent) until node is node.parent.left
- return node

predecessor(node): ditto, left, last, right ⇒ previous

Dynamic ops:

subtree_insert_after(node, new): add new node in traversal order immediately after node

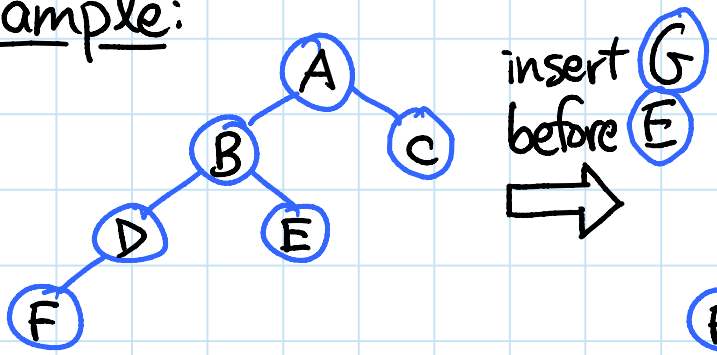
- if node.right empty: put new there
- else: - s = successor(node)
  - put new in s.left
    which is empty because first in subtree(node.right)



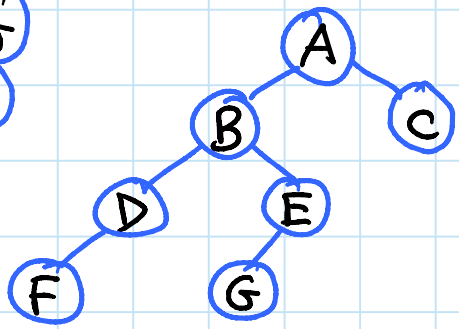
⇒ always adding new as leaf

subtree_insert_before(node, new): symmetric

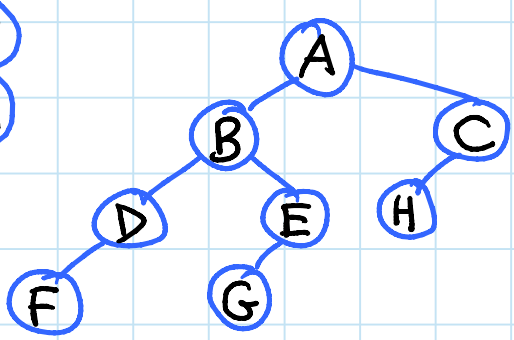
Example:



insert G before E ⇨



insert H after A ⇨



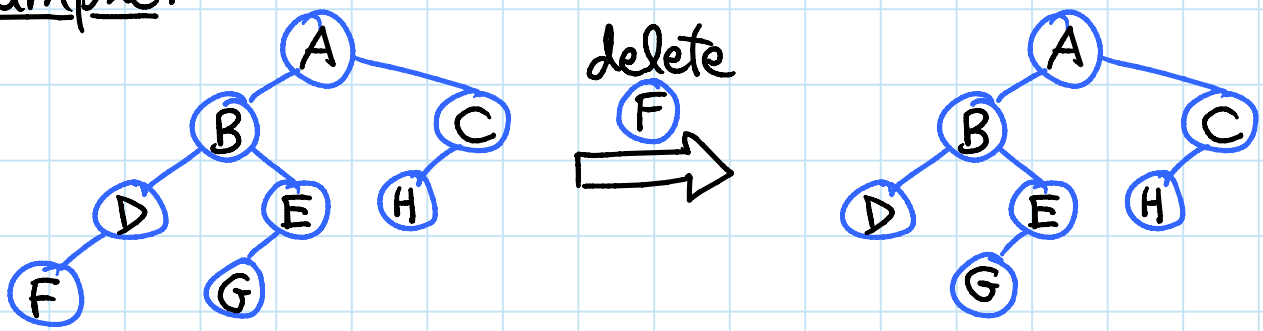
Subtree_delete(node):

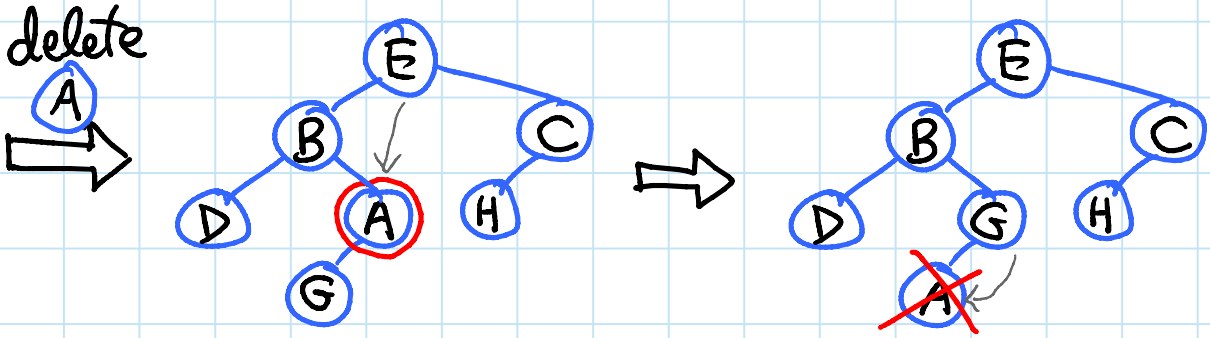
- if node is leaf: detach from parent
- else: node has ≥ 1 child

  - if node.left:
    - swap node.item ↔ predecessor(node).item
      → preserves traversal order of all but the to-be-deleted item
    - subtree_delete(predecessor(node))
      ⇒ continually go down in tree ⇒ O(h)
  - if node.right: ditto with successor

⇒ always detaching a leaf

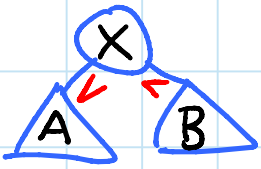
Example:





Application: set binary tree
("Binary Search Tree" / BST)

- traversal order = increasing key
- BST property: for every node:
  keys in subtree(node.left) → A
  < node.item.key → X.key
  < keys in subtree(node.right) → B



find(k): subtree_find(root, k)
subtree_find(node, k): like binary search O(h)
- if node = None: return None (no match)
- if k < node.item.key: recurse on node.left
- if k = node.item.key: return node
- if k > node.item.key: recurse on node.right

find_next(k): find(k) until would fall off tree
⇒ node is adjacent (next or prev) to where k fits
- if k < node.item.key: return node.item
- else: return successor(node) O(h)

find_prev(k): ditto, predecessor & >

find_min(): root.subtree_first()
find_max(): root.subtree_last()

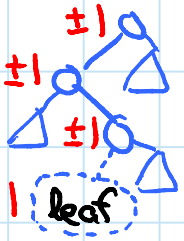
Application: sequence binary tree
- suppose we knew size of every subtree
  - ↳ # nodes

subtree_at(node, i): ith node in traversal order of subtree(node)
- n_L = size(node.left)
- if $i < n_L$: subtree_at(node.left, i)
- if $i = n_L$: return node
- if $i > n_L$: subtree_at(node.right, $i - n_L - 1$)

get_at(i): subtree_at(root, i).item
set_at(i, x): subtree_at(root, i).item = x

↳ for node

Augment binary tree to store subtree sizes
- each node stores node.size
- whenever insert a leaf, set its size = 1 & increment size of its ancestors
- whenever delete a leaf, decrement size of its ancestors
- additive O(h) overhead



Augmentation is a more general, powerful idea that we'll see more of in L7

Next: keep tree **balanced** i.e. $h = O(\lg n)$
⇒ all ops. in $O(\lg n)$ time