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:**

```
             __________<A>______       node   | <A> | <B> | <C> | <D> | <E> | <F> |
         __<B>______          <C>      item   |  A  |  B  |  C  |  D  |  E  |  F  |
     __<D>        <E>                  parent |  -  | <A> | <A> | <B> | <B> | <D> |
    <F>                                left   | <B> | <C> |  -  | <F> |  -  |  -  |
                                       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

```
          _____<A>__                ________<A>__
       __<B>__    <C>   =>       __<B>_____    <C>
    __<D>   <E>               __<D>     __<E>
   <F>                       <F>       <G>
```

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

```
         ________<A>___             ________<A>_____
      __<B>_____    <C>   =>     __<B>_____      __<C>
   __<D>     __<E>            __<D>     __<E>   <H>
  <F>       <G>              <F>       <G>
```

- **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)

```
         ________<A>_____            ________<A>_____
      __<B>_____      __<C>   =>  __<B>_____      __<C>
   __<D>     __<E>   <H>         <D>     __<E>   <H>
  <F>       <G>                         <G>
```

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

```
      ________<A>_____          ________<E>_____          _____<E>_____
   __<B>_____      __<C>  =>  __<B>_____      __<C>  =>  __<B>__     __<C>
  <D>     __<E>   <H>        <D>     __<G>   <H>        <D>   <G>   <H>
         <G>                        <A>
```

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 $h = O(\log n)$