Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — Sep. 24, 2019
Instructors: Jason Ku, Julian Shun, and Virginia Williams — Lecture 6: Binary Trees I

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:**

```
          ________<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>)
- The **depth** D(<X>) of node <X> in a tree rooted at <R> is length of path from <X> to <R>
- The **height** H(<X>) of node <X> is 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
- Next time, 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 before another in the traversal order (after is symmetric)
  - remove an item from the tree
- **Add** node <Y> before node <X> in the traversal order
  - If <X> has no left child, make <Y> the left child of <X>
  - Otherwise, make <Y> the right child of <X>'s predecessor (which cannot have a right child)
  - Running time is $O(h)$ where h is the height of the tree
  - **Example:** Add node <G> before <E> in traversal order

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

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

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

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

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!**

```
         ______<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
- **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. $\square$
- 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