Lecture 6: Binary Trees I

Previously and New Goal

<table>
<thead>
<tr>
<th>Sequence Data Structure</th>
<th>Container</th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>$n$</td>
<td>$1$</td>
<td>$n$</td>
</tr>
<tr>
<td>Linked List</td>
<td>$n$</td>
<td>$n$</td>
<td>$1$</td>
</tr>
<tr>
<td>Dynamic Array</td>
<td>$n$</td>
<td>$1$</td>
<td>$1(a)$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Set Data Structure</th>
<th>Container</th>
<th>Static</th>
<th>Dynamic</th>
<th>Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>Sorted Array</td>
<td>$n \log n$</td>
<td>$\log n$</td>
<td>$n$</td>
<td>$1$</td>
</tr>
<tr>
<td>Direct Access Array</td>
<td>$u$</td>
<td>$1$</td>
<td>$1(a)$</td>
<td>$u$</td>
</tr>
<tr>
<td>Hash Table</td>
<td>$n(e)$</td>
<td>$1(e)$</td>
<td>$1(a(e))$</td>
<td>$n$</td>
</tr>
</tbody>
</table>

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

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:

```
node | <A> | <B> | <C> | <D> | <E> | <F> |
item | A   | B   | C   | D   | E   | F   |
parent | -   | <A> | <A> | <B> | <B> | <D> |
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

```
1 ______<A>__  __________<A>__
2 ___<B>___ <C> => __<B>______ <C>
3 __<D>__ <E>   __<D>__ <E>
4  <F> ______<F>__ <G>
```

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

```
1 ________<A>___  __________<A>___
2 ___<B>______ <C> => __<B>______ __<C>
3 __<D>__ __<E>   __<D>__ __<E> <H>
4  <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)

```
1 __________<A>___  __________<A>___
2 ___<B>______ <C> => __<B>______ __<C>
3 __<D>__ __<E>   __<D>__ __<E> <H>
4  <F> __<G>  <F> <G>
```

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

```
1 ________<A>___  __________<E>___
2 ___<B>______ __<C> => __<B>______ __<C> => __<B>______ __<C>
3 __<D>__ __<E> <H>   __<D>__ __<G> <H>   __<D>__ __<G> <H>
4  <G> __<A>  <G> <A>
```
Lecture 6: Binary Trees I

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 \( \text{subtree\_at}(i) \)
- 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 \( \text{node\.size} \) 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 \( \text{subtree\_at}(i) \) operation
- Naively, \( \text{build}(X) \) takes \( O(nh) \) time, but can be done in \( O(n) \) time; see recitation
# Lecture 6: Binary Trees I

## So Far

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Operations $O(\cdot)$</th>
<th>Container</th>
<th>Static</th>
<th>Dynamic</th>
<th>Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set</td>
<td></td>
<td>build(X)</td>
<td>find(k)</td>
<td>insert(x)</td>
<td>find_min()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>delete(k)</td>
<td>find_max()</td>
</tr>
<tr>
<td>Binary Tree</td>
<td>$n \log n$</td>
<td>$h$</td>
<td>$h$</td>
<td>$h$</td>
<td>$h$</td>
</tr>
<tr>
<td>Goal</td>
<td>$n \log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Operations $O(\cdot)$</th>
<th>Container</th>
<th>Static</th>
<th>Dynamic</th>
<th>Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td></td>
<td>build(X)</td>
<td>get_at(i)</td>
<td>insert_first(x)</td>
<td>insert_last(x)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>set_at(i, x)</td>
<td>delete_first()</td>
<td>delete_last()</td>
</tr>
<tr>
<td>Binary Tree</td>
<td>$n$</td>
<td>$h$</td>
<td>$h$</td>
<td>$h$</td>
<td>$h$</td>
</tr>
<tr>
<td>Goal</td>
<td>$n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
</tr>
</tbody>
</table>

## Next Time

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