

Lecture 6: Binary Search Trees

Dynamic (Multi)set of Ordered Data

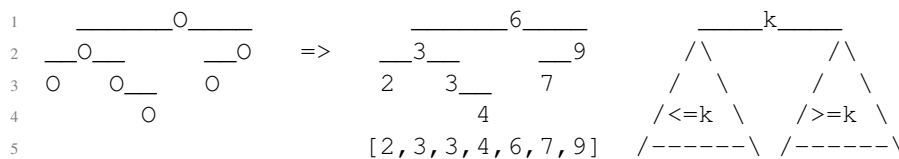
- Want to maintain an **order** on a **dynamic** set or multiset of elements.
 - Many benefits of a sorted array, but with dynamic data
 - When priority queues are too specialized
- Some applications:
 - Resource scheduling, make reservation at requested time
 - Online game ranking, who is ranked above and below me?
- Operations:
 - **Dynamic:** `insert(k)`, `delete(k)`
 - **Search/Order:** `find(k)`, `find_next(k)`, `find_prev(k)`, `find_min()`, `find_max()`, `delete_min()`, `delete_max()`
- API Variations
 - Simplified API in lecture: items **are** keys, e.g., `insert(7)`
 - More general: items **have** keys (maybe with other info), and the data structure knows to look only at `item.key`
 - * `insert(item)`, where `item` is an **object** that has `item.key == 7`, but also (for example) `item.chi == "who?"` and `item.quay == "wharf"`

Implementations

Data Structure	<code>insert(k)</code>	<code>delete(k)</code>	<code>find(k)</code>	<code>find_next(k)</code>
Array	1	n	n	n
Sorted Array	n	n	$\log n$	$\log n$
Goal (Thursday)	$\log n$	$\log n$	$\log n$	$\log n$

Linked Binary Tree

- Array OK for complete, left-aligned binary tree, but want **any** binary tree.
- `node.{key, parent, left, right}`
- Word-RAM: parent, left, right are **pointers** to other nodes
 - w -bit words indicating memory addresses
- Always keep pointer to the root node
- **Example:**



- **Binary Search Tree:** binary tree satisfying BST Property.
- **BST Property:** `node.key` is (\geq, \leq) keys in (left, right) sub-tree
- After every operation, must restore BST Property

Find

- How to find key in rooted sub-tree?
- Either key is same, or in left or right sub-tree: recursive call
- If reach bottom and no key, key not in tree!
- Find 4 and 8 in example.

```

1 def find(node, k):
2     if node.key == k:
3         return node
4     elif k < node.key and node.left:
5         return find(node.left, k)
6     elif k > node.key and node.right:
7         return find(node.right, k)
8     return None

```

Minimum

- How to find minimum of rooted sub-tree?
- By BST Property, will be in left sub-tree. Walk left as far as possible.

```

1 def find_min(node):
2     if node.left: return find_min(node.left)
3     return node
4 # OR
5 def find_min_iterative(node):
6     while node.left: node = node.left
7     return node

```

Sorting/Traversal

- Keys basically in sorted order!
- Can use BST Property to list nodes in guaranteed sorted order:

```

1 def traversal(node):
2     if node.left: yield from traversal(node.left)
3     yield node
4     if node.right: yield from traversal(node.right)

```

- Runtime? $O(n)$, since constant work at every node (not counting work in recursive calls)
- If can make a BST on elements, can return sorted list in linear time
 - Our goal of $O(\log n)$ insert would give an $O(n \log n)$ sorting algorithm

Find Next

- What does in-order traversal look like? (Draw it!)
- How to step from a node to the next node in order?

```

1 def is_right_child(node):
2     return node.parent and (node.parent.right is node)
3
4 def successor(node):
5     if node.right:
6         return find_min(node.right)
7     while is_right_child(node):
8         node = node.parent
9     return node.parent

```

- Note: `successor(node)` is a BST-specific function, not part of the outward-facing API. The API call `find_next(k)` takes in a key instead of a node, and can be implemented with `find` followed by `successor`.
 - The `find_next` API call is ambiguous with multisets
 - `successor(node)` in a BST is unambiguously defined, but depends on the internal structure of the tree
- Runtime? $O(h)$
- Alternative traversal:


```

1 def traversal_iterative(root):
2     node = find_min(root)
3     while node:
4         yield node
5         node = successor(node)
      
```
- Looks like $O(nh)$, but look again!
 - Visits each node at most 3 times, so $O(n)$ overall

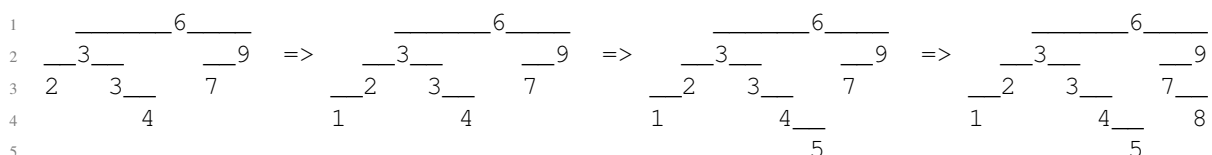
Insert

- Given node, how to insert new key?
- **Idea:** Add as a leaf!
- Search for position similarly to `find`

```

1 def insert(node, k):
2     if k <= node.key:
3         if node.left:
4             insert(node.left, k)
5         else:
6             node.left = new_node(key = k, parent = node,
7                                   left = None, right = None)
8     else:
9         <same on right>
      
```

- On example, insert 1, 5, 8



Delete

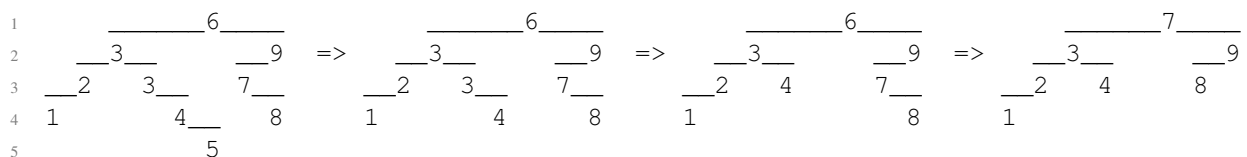
- Given node, how to delete key?
- If missing a child, trim or shortcut like a linked list!
- Otherwise, swap key with successor and delete

```

1 def delete(node):
2     if node.left and node.right:
3         succ = find_min(node.right)
4         node.key = succ.key
5         delete(succ)
6     elif node.left: <replace node with node.left>
7     elif node.right: <replace node with node.right>
8     else: <unlink node from parent>

```

- On example, delete 5 (trim leaf), lower 3 (shortcut), and 6 (swap with successor).



Analysis

- How long do these operations take? Order of height of tree, $O(h)$
- But h can be big (linear)!
 - E.g. inserted in sorted order (chain) or alternating lowest highest (zigzag)
- Next lecture we will show how to ensure $O(\log n)$ during dynamic operations