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	`insert(k)`	`delete(k)`	`find(k)`	`find_next(k)`
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:**

```
  ______O____          ______6____       ____k____
__O__     __O   =>   __3__     __9      /\       /\
O   O__   O          2   3__   7       /  \     /  \
      O                    4          /<=k \   />=k \
                    [2,3,3,4,6,7,9]  /------\ /------\
```

- **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.

```python
def find(node, k):
    if node.key == k:
        return node
    elif k < node.key and node.left:
        return find(node.left, k)
    elif k > node.key and node.right:
        return find(node.right, k)
    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.

```python
def find_min(node):
    if node.left:  return find_min(node.left)
    return node
# OR
def find_min_iterative(node):
    while node.left:  node = node.left
    return node
```

Sorting/Traversal

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

```python
def traversal(node):
    if node.left:  yield from traversal(node.left)
    yield node
    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?

```python
def is_right_child(node):
    return node.parent and (node.parent.right is node)

def successor(node):
    if node.right:
        return find_min(node.right)
    while is_right_child(node):
        node = node.parent
    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:

```
def traversal_iterative(root):
    node = find_min(root)
    while node:
        yield node
        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`

```
def insert(node, k):
    if k <= node.key:
        if node.left:
            insert(node.left, k)
        else:
            node.left = new_node(key = k, parent = node,
                                 left = None, right = None)
    else:
        <same on right>
```

- On example, insert 1, 5, 8

```
  ______6____         ______6____         ______6____         ______6____
__3__     __9  =>   __3__     __9  =>   __3__     __9  =>   __3__     __9
2   3__   7       __2   3__   7       __2   3__   7       __2   3__   7__
      4           1       4           1       4__         1       4__   8
                                                5                   5
```

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

```
def delete(node):
    if node.left and node.right:
        succ = find_min(node.right)
        node.key = succ.key
        delete(succ)
    elif node.left:  <replace node with node.left>
    elif node.right: <replace node with node.right>
    else:            <unlink node from parent>
```

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

```
    ______6_____        ______6_____        ______6_____        ______7_____
  __3__     __9   =>  __3__     __9   =>  __3__     __9   =>  __3__     __9
__2   3__   7__     __2   3__   7__     __2   4     7__     __2   4     8
1       4__   8     1       4     8     1             8     1
          5
```

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