CSE 30331
Treaps
Red-Black Tree: Review

**Good**

- **Efficient** insert, erase, and search operations

**Bad**

- **Complex** balancing mechanisms
Treaps
Treaps: Overview

Treap = Binary Tree + Max Binary Heap

- Each Node has a **value** and a randomly assigned **priority**
- Use the **value** to form **Binary Search Tree**
- Use **priority** to form **Max Binary Heap**
- With uniform distribution, the resulting tree should be **relatively balanced**
Treaps: Invariants

1. Each node’s value maintains the Binary Search Tree invariants
   a. Each node's value is never less than any value in the left subtree
   b. Each node's value is always less than any value in the right subtree

2. Each node's priority maintains the Max Binary Heap invariants
   a. Each node's value is always greater than the values in its children
Treap: **Search**

Same as BST!
Treap: Insert

1. Perform standard Binary Search Tree insertion

2. When location is found, create new Node with given value and random priority

3. Use rotations to ensure inserted node's priority follows Max Binary Heap property
Treap: Insert (Example)

Insert: D (60), F (75), H (14), C (70), A (55), I (65), J (45)

Insert: D (60), F (75), H (14), C (70), A (55), I (65), J (45)

Rotate Right

Rotate Left
Trep: **Insert (Example)**

Insert: D (60), F (75), H (14), C (70), A (55), I (65), J (45)

```
I
```

```
F
   C
      A
      D
```

```
F
   I
      H
```

```
F
   C
      A
      D
      H
```

```
F
   I
```

```
F
   C
      A
      D
      H
      J
```
Treap: **Insert** (Pseudo-Code)

```plaintext
Insert(node, value):
    if node is None:  # Base Case: Allocate and return new Node
        return Node(value)

    if value <= node.value:
        # Recursively insert into left sub-tree based on BST invariant
        node.left = Insert(node.left, value)
        # Check Max Binary Heap invariant, Rotate if Necessary
        if node.left.priority > node.priority:
            node = RotateRight(node)
    else:
        # Recursively insert into right sub-tree based on BST invariant
        node.right = Insert(node.right, value)
        # Check Max Binary Heap invariant, rotate if Necessary
        if node.right.priority > node.priority:
            node = RotateLeft(node)

    # Return update node
    return node
```
**Treap: Rotate Right (Pseudo-Code)**

*RotateRight*(p):

```plaintext
c, T2 = p.left, c.right
c.right = p
p.left = T2
return c
```
Treap: Rotate Left (Pseudo-Code)

\[
\text{RotateLeft}(p):
\]
\[
c, T2 \ = \ p.\text{right}, \ c.\text{left} \\
c.\text{left} \ = \ p \\
p.\text{right} \ = \ T2 \\
\text{return} \ c
\]
Treas: Erase

1. Find the node to be deleted as in Binary Search Tree

2. If node is a leaf, just delete it

3. Otherwise, replace node's priority with minus infinite (-INF), and do appropriate rotations to bring the node down to a leaf and then delete it.
Treap: Properties

- **Best case**: $O(\log(n))$
- **Average case**: $O(\log(n))$
- **Worst case**: $O(n)$

**Treaps** probabilistically balance **Binary Search Trees** by overlaying a **Max Binary Heap**