TREAP

Huarui Liu, Jingzhou Qiu, Zicheng He

What is a Treap?

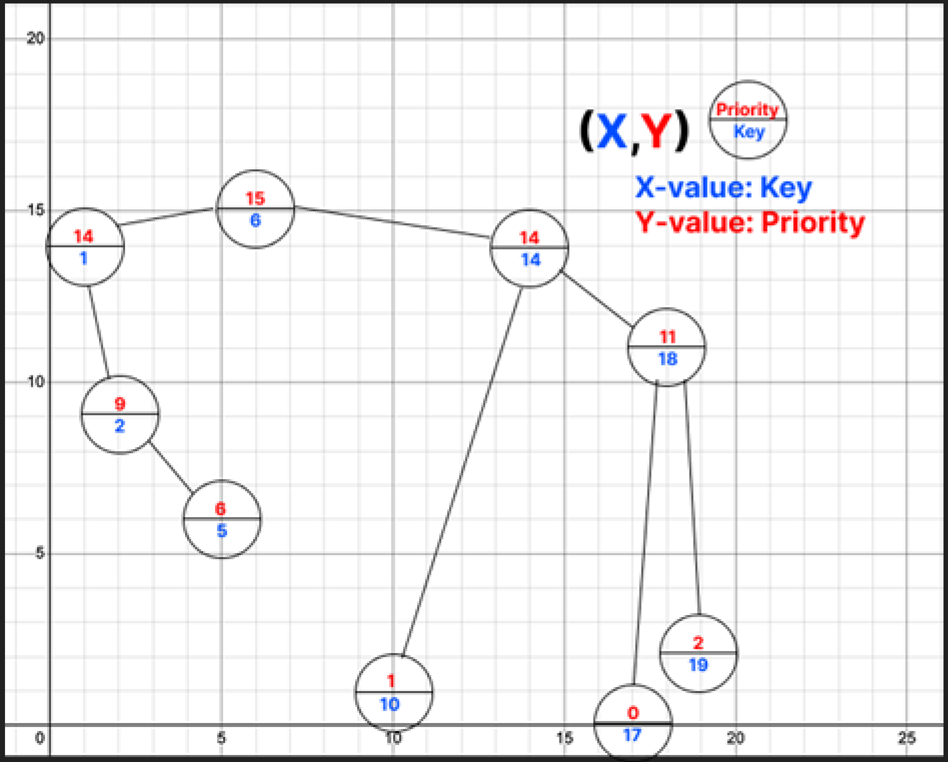
Keys in sorted order like a **BST**

Priorities follow the **heap property**

- Randomized BST Combines BST and heap properties
- A type of Cartesian Tree
- lookup, insertion, and removal in O(logN)

Additional Operations:

- Split: O(logN)
- Merge: O(logN)



Why Treap?

Self-Balancing via random priorities

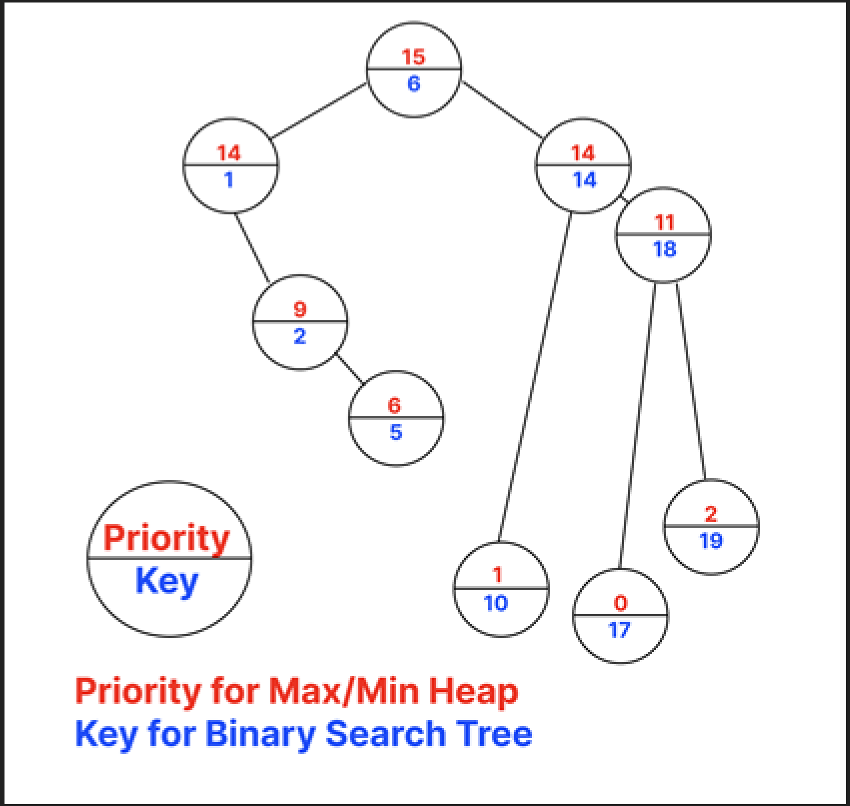
Simpler to implement than AVL or Red-Black trees

Can be modified to support segment tree operations and even more– all in O(logN)

- Reverse on the interval.
- Addition / painting on the interval.

Applications

- Linux kernel page cache management
- General Purpose Allocator (GPA)



Insert (Min-heap)

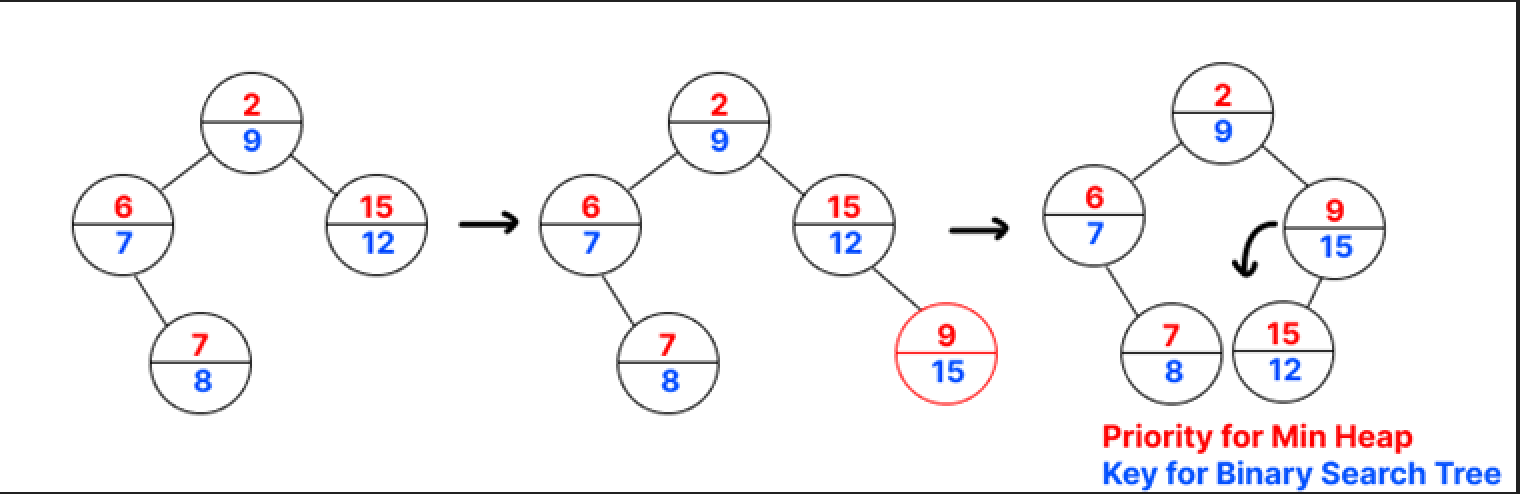
- Pick a random priority/specify a priority
- Insert as inserting in BST
- Rotate until the heap order is maintained

Runtime: O(logN)

```
function insert(node, key, priority):
    if node is empty:
        create and return a new node with key and priority

    if key is less than node's key:
        recursively insert into left subtree
        if left child has higher priority than current node:
            perform right rotation

    else if key is greater than node's key:
        recursively insert into right subtree
        if right child has higher priority than current node:
            perform left rotation

    else:
        // key is equal — duplicate, so do nothing

    return current node
```

Insert-Example

Insert(15) ->Random priority=9



Delete(Min-heap)

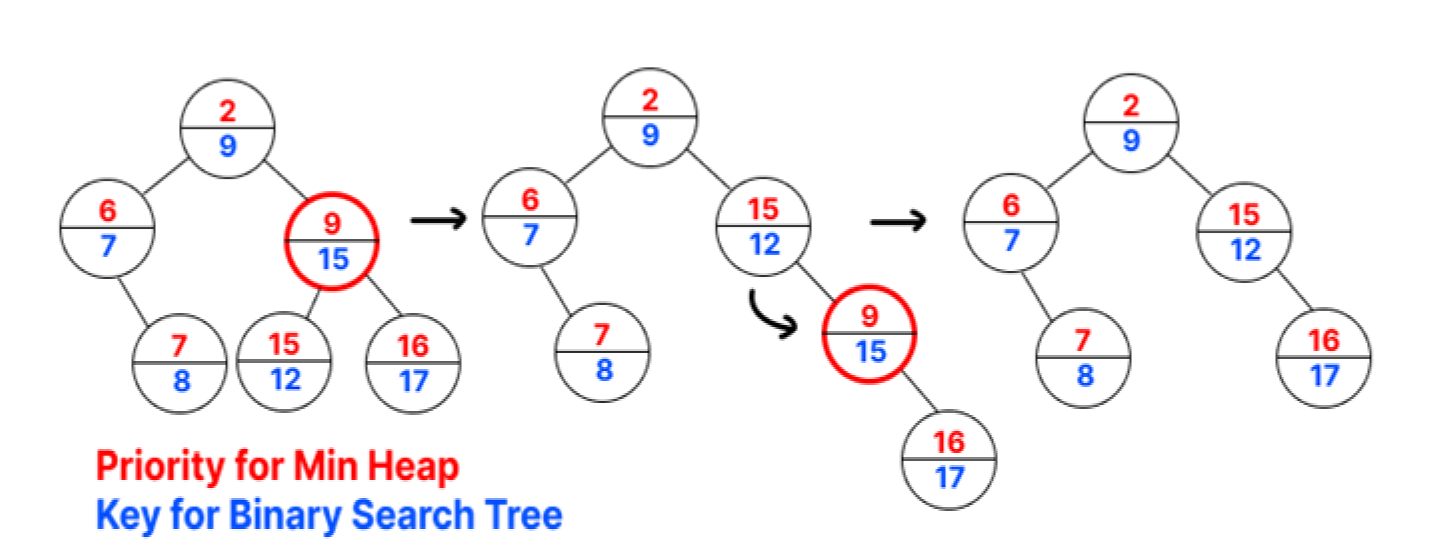
- Find the node by key (BST-style).
- If the node has 0 or 1 child:
  - return non-null child or null
- If the node has 2 children:
  - Rotate the child with the smaller priority up
  - Recurse on the same key to delete it

Runtime: O(logN)

```
function delete(node, key):
    if node is null:
        return null
    if key < node.key:
        node.left = delete(node.left, key)
    else if key > node.key:
        node.right = delete(node.right, key)
    else:
        if node has at most one child:
            return the non-null child (or null)
        if left.priority < right.priority:
            rotate right, then delete key from right child
        else:
            rotate left, then delete key from left child
    return node
```

Delete-Example

Delete (15)



Build

Heapify ensures the parent node has the highest/lowest priority by recursively swapping with the larger/smaller-priority child

Builds a tree from a list of values.

Case 1: Input Keys Are Sorted -> Build in O(N) time

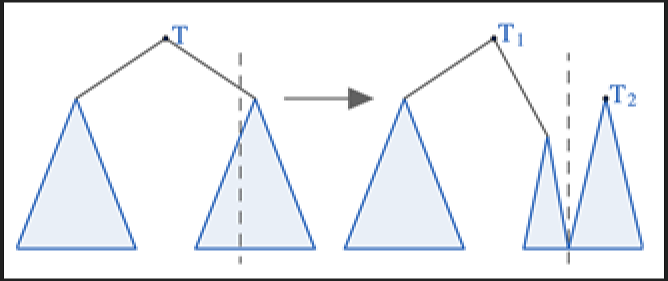
Select the middle element to construct BST

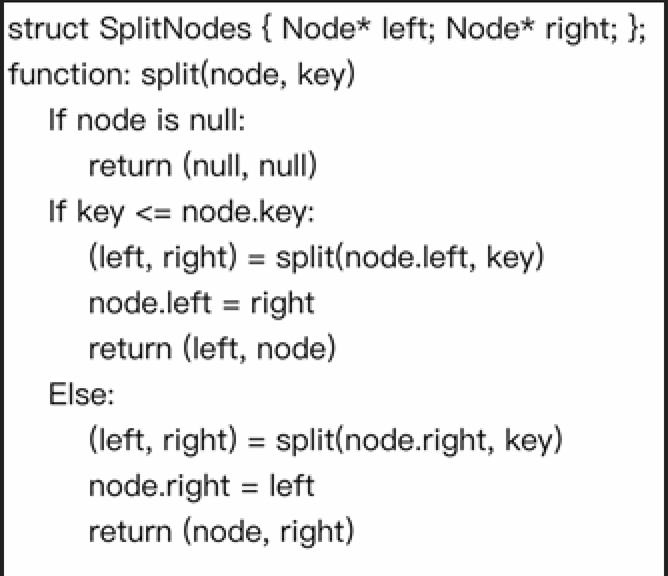
Use heapify to ensure the heap property based on priorities

Case 2: Perform N insertions -> O(Nlog N) time

Split

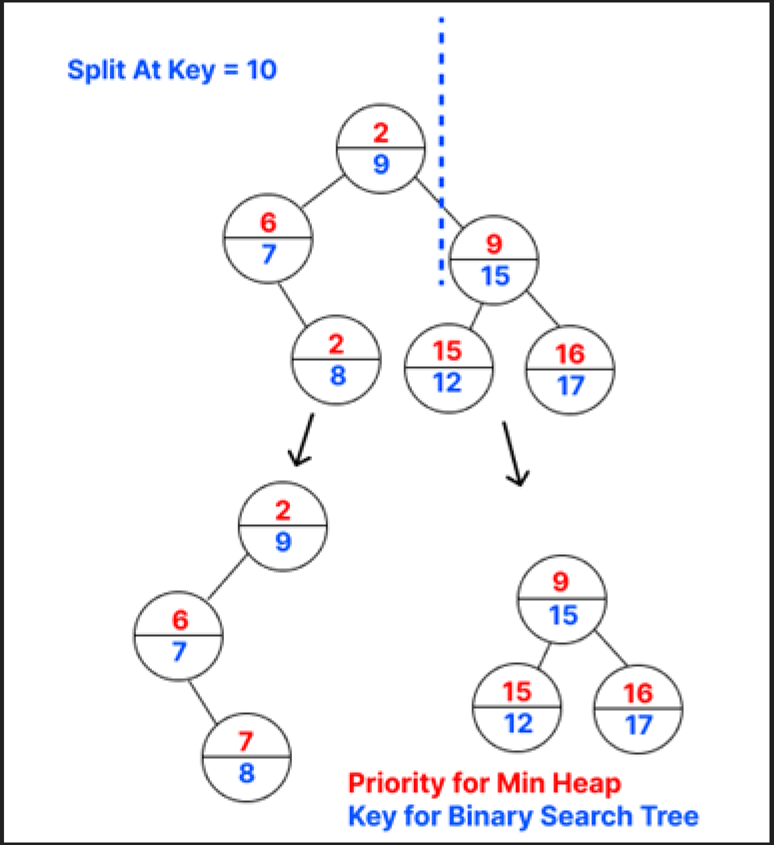
- Decide which subtree the root node would belong to (left or right)
- Recursively call split on one of its children
- Create the final result by reusing the recursive split call
- Runtime: O(logN)





```
struct SplitNodes { Node* left; Node* right; };
function: split(node, key)
    If node is null:
        return (null, null)
    If key <= node.key:
        (left, right) = split(node.left, key)
        node.left = right
        return (left, node)
    Else:
        (left, right) = split(node.right, key)
        node.right = left
        return (node, right)
```

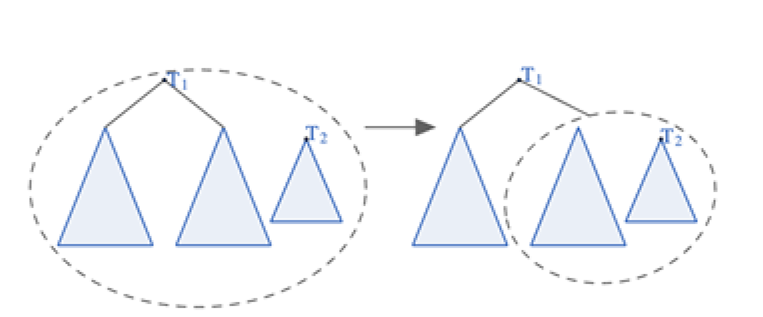
Split-Example



Merge(Min-heap)

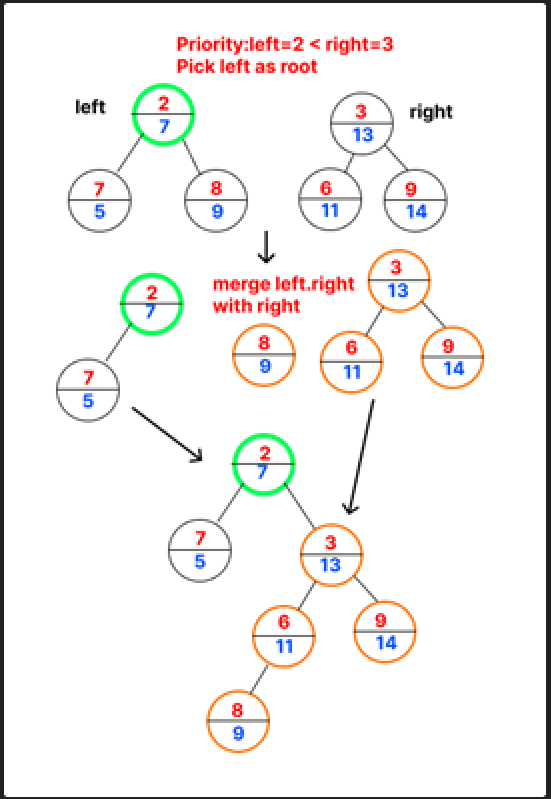
- Merges two treaps (left and right) assuming all keys in left are less than those in right.
- Chooses the root with larger/smaller priority to maintain the heap property

Runtime: O(logN)




```
function: merge(left, right)
    If left is null or right is null:
        return left if left exists, otherwise right
    If left.priority < right.priority:
        left.right = merge(left.right, right)
        return left
    Else:
        right.left = merge(left, right.left)
        return right
```

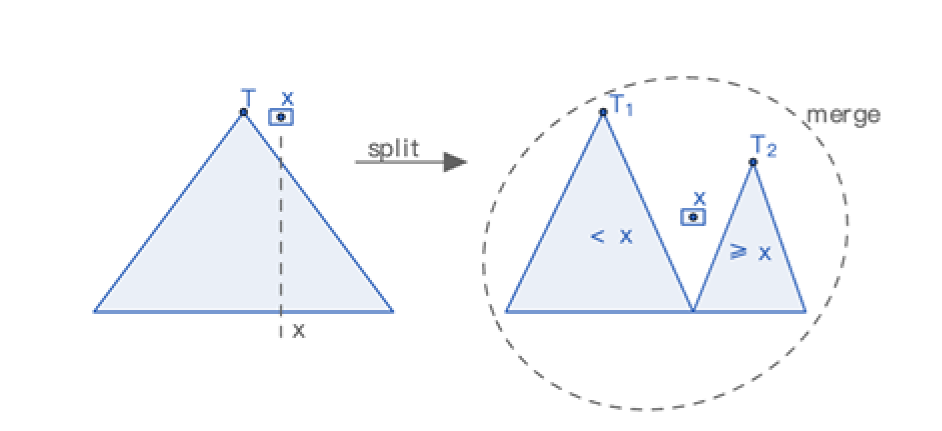
Merge-Example



Insert-Using Split

Runtime: O(logN)

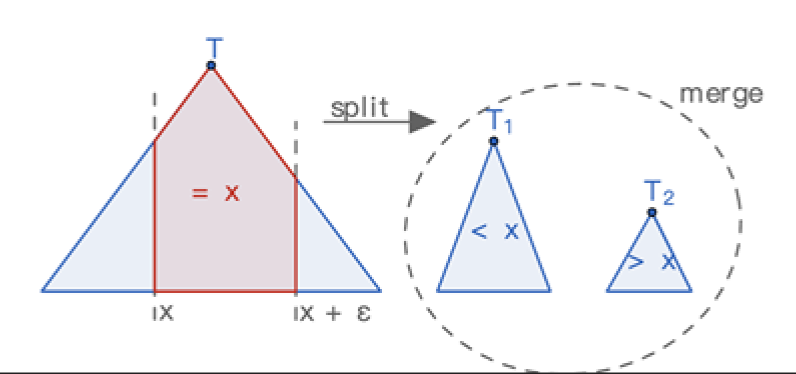
- Pick a random priority/specify a priority
- Insert as inserting in BST
- Rotate until the heap order is maintained




```
function insertSplit(key, optional_priority):
    if optional_priority is not provided:
        priority = generate_random_priority()
    else:
        priority = optional_priority
    new_node = new Node(key, priority)

    (L, R) = split(root, key)
    root = merge(merge(L, new_node), R)

    update_subtree_size(root)
```

Delete-Using Split

- The tree is split into three parts:
  - L: nodes with keys < key
  - target: node with key == key
  - R: nodes with keys > key
- Delete the target node
- Merge L and R back together



Runtime: O(logN)

```
function deleteSplit(key):
    (L, mid) = split(root, key)
    (target, R) = split(mid, key + 1)

    delete target
    root = merge(L, R)

    update_subtree_size(root)
```

Extension on Treaps-Implicit Treap

Implicit Keys:

- The position of each node is its key
- Index is calculated using subtree sizes

Nodes store more info to support extra features

Allow fast range queries and updates

Flexible Operations: Allows insertions, deletions, and reversals

```
struct TreapNode {
    int value;
    int priority;
    int subtreeSize;
    bool reversed;
    int addValue;
    bool needsUpdate;

    TreapNode* leftChild;
    TreapNode* rightChild;
}
```

Utility functions

```
function count(node):
    if node is null:
        return 0
    return node.subtreeSize
// subtreeSize: subtree node count including current node
```

```
function updateCount(node):
    if node is not null:
        node.subtreeSize = 1 + count(node.left) + count(node.right)
```

SubtreeSize serves as the node's position like an implicit index

updateCount function maintains accurate subtree sizes

We will implement an Implicit Treap with the reverse function

Updated Split

Runtime: O(logN)

```
function split(node, index):
    if node is null:
        return (null, null)

    push(currentNode)
    compute currentNodeIndex based on size of left subtree

    if index <= currentNodeIndex:
        split left subtree
        attach result to node's left
        return (left part, node)
    else:
        split right subtree
        attach result to node's right
        return (node, right part)

    update node's metadata (e.g., subtree size)
```

- calculates each node's position
- split the nodes with into left and right subtree based on the given index
- uses push()to apply any pending lazy updates (in our example, the reversal flag only)
- use updateCount() to keep track of subtree sizes accurate

*push() function will be introduced later

Updated Merge

```
function merge(left, right):
    push(leftTree)
    push(rightTree)

    if either tree is null:
        return the non-null tree

    if left.priority < right.priority:
        left.right = merge(left.right, right)
        update left's metadata
        return left
    else:
        right.left = merge(left, right.left)
        update right's metadata
        return right
```

- uses push() to apply any pending lazy updates
- same merging logic, except we need to keep track of subtree sizes and other metadata

Runtime: O(logN)

Lazy Propagation for Reverse

```
function push(node):
    if node is not null and node.reversed is true:
        swap(node.left, node.right)

        if node.left is not null:
            node.left.reversed = not node.left.reversed

        if node.right is not null:
            node.right.reversed = not node.right.reversed

        node.reversed = false
```

- Instead of immediately swapping all left/right children(which would be O(N)), flip the reversed flag
- Call push() when the Node is accessed

Runtime: O(logN)

Reverse Function

Runtime: O(logN)

```
function reverseSegment(root, leftIndex, rightIndex):
    // Split the tree into two parts:
    (leftSubtree, remainingTree) = split(root, leftIndex)

    // Split remainingTree again:
    (middleSubtree, rightSubtree) = split(remainingTree, rightIndex – leftIndex + 1)

    // Mark the middleSubtree to be reversed (lazy propagation)
    if middleSubtree is not null:
        middleSubtree.reverseFlag = not middleSubtree.reverseFlag

    // Merge all three parts back together in order
    root = merge(leftSubtree, middleSubtree)
    root = merge(root, rightSubtree)
```

Conclusion

- Treap combines the balanced BSTs with array-style indexing with most operations in O(log N) time
- more powerful than segment trees

However:

- They're mostly only used in competitive programming and academia:
- developers favor other more commonly used structures like segment trees or balanced BST libraries

Reference

https://cp-algorithms.com/data_structures/treap.html

https://www.youtube.com/watch?v=6x0UIIBLRsc

https://courses.cs.washington.edu/courses/cse326/00wi/handouts/lecture19/sld017.htm