What we are going to talk about

- Sparse Table
- AVL Tree
- Treap
- Split/Merge Treap
- Implicit Split/Merge Treap
Sparse Table

- Sparse Table is a data structure that supports query but not update.
- It requires $O(N \log N)$ to precompute.
- But can answer queries in $O(1)$!!!
- Very good when you need to perform many queries.
Sparse Table

• Assume we now want to solve the range minimum query problem

• Denote $f(k, i)$ as the minimum element from $i$ to $i+2^k - 1$

• $f(0, i)$ is obvious, as it ask for minimum in $[i, i]$ which equals to $a[i]$.

• For $f(1, i)$, $f(1, i) = \min(f(0, i), f(0, i+1))$

• Generally, we can combine information of $f(k, i)$ and $f(k, i+2^k)$ to $f(k + 1, i)$

• So we now can use $O(N \log N)$ to precompute $f(k, i)$
Sparse Table

- $f(0, i)$
- $f(0, i+1)$
- $f(0, i+2)$
- $f(0, i+3)$
- $f(1, i)$
- $f(1, i+2)$
- $f(2, i)$
Sparse Table

- Now for the query part.
- Suppose we query(l, r)
- Obviously if (r - l + 1) is $2^k$, $f(k, l)$ is what we need to find
- But what if (r - l + 1) can’t be represented by $2^k$?
- Let $k$ be the maximum integer such that $2^k \leq (r - l + 1)$
- $\min(f(k, l), f(k, r - 2^k + 1))$ is what we need to find
- Just combine information from two parts.
- Note that the two interval might overlap so it only work when function $f$ is associative.
Sparse Table for LCA

- For the lowest common ancestor problem, we could use sparse table to solve it.
- \( f(k, v) = u \) where \( \text{dist}(u, v) = 2^k \) and \( u \) is \( v \)'s ancestor.
- \( f(0, v) = \text{parent of } v \). \( f(i + 1, v) = f(i, f(i, v)) \). The \( 2^i \) parent of the \( 2^i \) parent for node \( v \) is its \( 2^{(i + 1)} \) parent.
Sparse Table for LCA

- For finding LCA of u and v. Assume depth(u) < depth(v)
- Keep lifting v using f(k, v) until depth(u) = depth(v)
- v = u then u is the answer
- Otherwise, iterate k from \( \log_2(n) \) to 0, if f(k, v) \( <> \) f(k, u), lift v and u to f(k, v) and f(k, u)
- f(0, v) would be the answer
Binary Search Tree

- Binary search tree is great for finding the k-th integer and also other information
- But normal bst might have high time complexity
- It degenerate to chain when the inserted data is sorted and query time become $O(N)$
- So people invented some self-balancing binary search tree
- (If there is no inserting operation, you can just construct a BST by choosing the median every time to achieve log N height)
Self-Balancing Binary search tree

- In c++, std::set and std::map is a balanced binary search tree (BBST). Don’t even try to code a BBST if contest if you could solve it using stl:: ... Just use set and map....

- If it is so unfortunate that you really need to code it, don’t spend all the time in contest to test and debug it...
Self-Balancing Binary search tree

- There are two kinds of BBST, one that would rotate and one that would not.

- **Self-Balancing by rotation:**
  - AVL tree
  - Treap
  - Splay tree
  - etc

- **Self-Balancing by without rotation:**
  - Split/merge treap
  - Scapegoat tree
  - etc
AVL Tree

- AVL Tree is a BBST that use rotation to balance the tree.
- Consider every node in AVL tree,
  - The left subtree and right subtree height would only differ at most by 1
  - Once the absolute difference > 1, we rotate the taller subtree as root
  - So the absolute different never exceed 1
AVL Tree

- Let’s define left rotate and right rotate in node v
- Let left child of v be l and right child be r
- Left rotate
  - v’s right child become r’s left child
  - r become the root, r’s left child become v
- Right rotate
  - v’s left child become l’s right child
  - l become the root, l’s right child become v
From wikipedia
From wikipedia
Insertion and Deletion in AVL Tree

- After inserting a node, trace back to the root, once a node is imbalanced, apply rotation (we will talk about what rotate to use later).
- For deletion, if the node is a leaf, just delete it and also trace back to root and maintain the balance.
- But if the node is not a leaf exchange it with it’s precedent (the largest node in the left subtree).
- Always fix the one farthest from the root first
Here is all the case

Highlighted node is the subtree that is taller

Image from wikipedia
Treap

- Treap is another BBST
- It uses random solution to problem
- We randomly assign a priority value for each node
- Then we have to maintain the treap so that priority of every node is larger than its child
- I don’t know how to prove but the expected time complexity is $O(\log N)$
Insertion/Deletion in Treap

- After insert and delete, we have to maintain the property that priority of every node is larger than their child
- Similar technique to AVL tree is used
- If left/right child have larger priority, right/left rotation is applied
- For deletion, if the deleted item isn’t leaf, exchange with it’s precedent. Keep rotating the exchanged node to maintain the tree property
Split/Merge Treap

- Split/merge treap has same property with treap, the only different is that it does not use rotation but through split and merge to maintain the property
- Split/merge treap is great that it just need a few lines of code
- It has two major operation, split and merge of course
- Define split(T, key) as we split T into two tree L and R such that L contains all element < key
- Merge(T₁, T₂) as merging two tree. All element in T₁ < element in T₂
- Also the resulted tree would not violate the treap property
Insertion/Deletion in Split/Merge Tree

- Insertion is easy when we have the split and merge operation. We go down in the tree like normal insertion.
- If the current subtree is empty, assign the inserted node to it.
- If the priority of the current node is smaller than the one we insert, split the current subtree $T$ by the inserted node’s key, i.e. $\text{split}(T, \text{key})$, $T_1$ is the left subtree for the inserted node and $T_2$ is the right subtree. And the inserted node become the root.
Insertion/Deletion in Split/Merge Tree

- Deletion is even more easy
- Once we found the node $v$, replace $v$ with merging its left and right child, i.e. $\text{merge}(v\text{'s left, } v\text{'s right)}$
Split/Merge Treap

- Now we just need to learn how to implement split and merge
- Both of them could be done with easy recursion
Split/Merge Treap

- For split T into L, R by Key
  - Let’s say current node is v
  - If v is empty then L, R is empty also
  - If v’s key is less than Key
    - split v’s right subtree. We get L’ and R’ from it
    - Assign the split tree L’ as v’s new right subtree.
    - Assign R’ to R
    - Assign T to L
  - Otherwise
    - split v’s left subtree. We get L’ and R’ from it
    - Assign the split tree R’ as v’s new left subtree
    - Assign L’ to L
    - Assign T to R
Split/Merge Treap

- For merge L, R to T
  - If L and R is empty, T is empty
  - If L or R is empty, assign the one isn’t empty to T
  - If L’s root have higher priority
    - Merge L’s right subtree with R
    - Assign L to T
  - Otherwise
    - Merge R’s left subtree with L
    - Assign R to T
Split/Merge Treap

- Clearly all operation works in $O(\log N)$
- So we got great time complexity!
- You can also maintain the subtree size in every node. So one can easily find the rank of any element.
- Remember to update in split and merge
Implicit BBST

• We could use a BBST to represent an array and answer some query.
• We would use the index as key and stores other data.
• The key is implicit because when we insert a node it might take $O(N)$ to update other node’s key.
• We can quickly find a node’s key by adding the node’s left subtree size and maybe some of its ancestor’s left subtree size.
• So we should maintain the subtree size for every node.
• Update the size in some operation.
Implicit Split/Merge Treap

- Suppose we have to solve RMQ, again
- We can store the information in node like segment tree where node v maintain the largest value in it’s subtree.
- We maintain this value along with subtree size
- We can now answer some range query \([l, r]\)
- Split the tree into three part, \(T_1\), \(T_2\) and \(T_3\) where \(T_1\) represent \([1, l - 1]\), \(T_2\) represent \([l, r]\) and \(T_3\) represent \([r + 1, n]\)
- \(T_2\)’s root information would be answer
Implicit Split/Merge Treap

- For implicit split/merge treap, we can even support some operation like insert element in array position $x$. We just have to insert a node where it rank would be $x$.

- Also, implicit BBST also support like reversing a segment of the array. Just tag a lazy flag in the node. Reverse the left and right subtree and push the flag down.
Data Structure

- You should know the pros and cons of every data structures so you could choose the best one for solving particular problems.
- Ex. many query and function is associative → sparse table
- Normal query → Segment tree/BIT
- Kth integer → Treap
- Interval Kth integer with flipping interval → Implicit split/merge treap
- This is only for general and problems might have different properties and other details so think carefully before choosing which data structures
Reference

Feel free to ask me for any question