Data Structure (IV)
Theo

Agenda

Binary-indexed Tree (BIT)
Sparse Table

AVL tree
Treap

Recall

- Tree structures
  - Binary tree, complete binary tree
- Range query data structures
  - Prefix sum
  - Segment tree

Agenda

Binary-indexed Tree (BIT)

Sparse Table

AVL tree

Treap

Binary-indexed Tree

- Problem:
- Point update(pos, val)
- Range query(L, R)

Binary-indexed Tree

- Problem:
- Point update(pos, val)
- Range query(L, R)

Solution 1: Brute force

O(N^2)

Binary-indexed Tree

- Problem:
- Point update(pos, val)
- Range query(L, R)

Solution 2: Segment tree

O(N lg N)

Drawback?

Binary-indexed Tree

- Problem:
- Point update(pos, val)
- Range query(L, R)

Drawbacks of segment tree:

- Hidden constant (4x!)
- Large memory consumption (2x!)

  Getting serious if you consider 2-D version...

Binary-indexed Tree

Consider L = 1 for each Query(L, R)

That means we consider only Query(1, R)

Idea: break the segment [1..R] into subsegments, lengths having power of 2

Binary-indexed Tree

E.g.

R = 10

[1..R] = [1..8] + [9..10]

2^3 2^1

R = 14

[1..R] = [1..8] + [9..12] + [13..14]

2^3 2^2 2^1

Notice that each time we take the greatest power of 2 we can take

Binary-indexed Tree

Same idea:

For R = 10 = 1010(binary),

equals

[1001..1010] and [0001...1000]

For R = 14 = 1110(binary)

equals

[1101..1110] and [1001...1100] and [0001..1000]

Binary-indexed Tree

- For query [1..R],
- Let lowbit(R) be the lowest significant ‘1’ bit in R
- Then query[1..R] = a[R - lowbit(R) + 1..R] + query[1..R - lowbit(R)]
- a[R - lowbit(R) + 1..R] aka a[R]

Binary-indexed Tree

- Update (pos, val):
- while (pos <= n)
  - A[pos] += val
  - Pos += lowbit(pos)

Binary-indexed Tree

- How to implement lowbit(x)?
  - lowbit(x) = x & -x
- How to handle L != 1?
  - Query(L, R) = Query(1, R) - Query(1, L - 1)
- Complexity:
- O(N) build, O(lg N) per query, O(N) memory
  - Notice constant difference!

Binary-indexed Tree

BIT usage: when the queries are subtractive

i.e. F(L, R) = F(1, R) - F(1, L - 1)

E.g. addition, xor

Segment tree usage: when the queries are additive

i.e. F(L, R) = F(L, M) + F(M + 1, R)

E.g. max, min

Agenda

Binary-indexed Tree (BIT)

Sparse Table

AVL tree

Treap

Sparse Table

- Offline solution to range query problems
- Offers O(1) per query
- after O(N lg N) precompute

Sparse Table

- Idea similar to BIT
- What if we maintain, for each position i and each integer k, the answer for [i..i+2^k-1]?

- Notice: update now costs **O(N)**!!!

Sparse Table

- Consider range max query (L, R)
- If 2^k <= (R - L + 1), then
- Query(L, L + 2^k - 1) and
- Query(R - 2^k + 1, R) does not exceed the query range

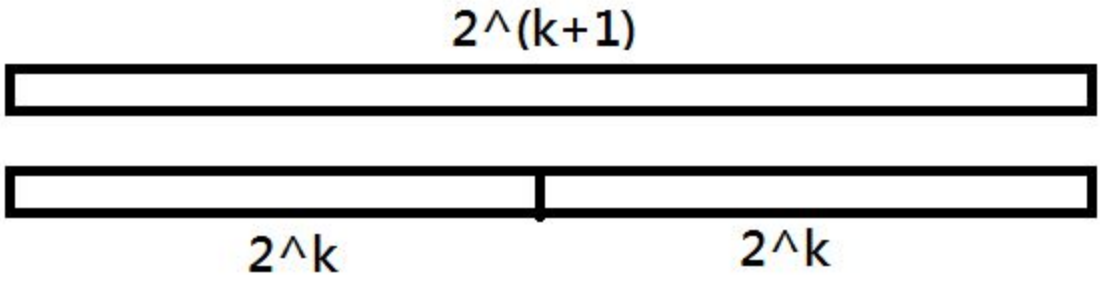
- But if k is the largest k, these two ranges cover the whole query (L,R)

Sparse Table

- Since we have, for all index and all k, the answer for [x..2^k+1]
- We can just lookup the value of two ranges and max them
  - O(1)

Sparse Table

- Construction:



- Combine two 2^k segments to get one 2^(k+1) segments!

Agenda

Binary-indexed Tree (BIT)

Sparse Table

AVL tree

Treap

AVL Tree

- Self-balancing binary search tree
- But wait why do we need one?

Bad thing about BSTs

- Let the height of the tree be h
- For naive implementation of BSTs
  - h is in O(n)
  - Which means every search is O(n)
  - Not much better than brute force scan

- Target: make BST balanced, by
  - Making h in O(lg N) OR
  - Make the operation fast in amortized time

Standard tools

- Normally you want std::set<int> to do that
  - Don’t even think about coding balanced tree if you can use them

- Problem: C++ (pre 11) set (and maps) does not support rank (given an element find out how many elements are larger than itself)

AVL

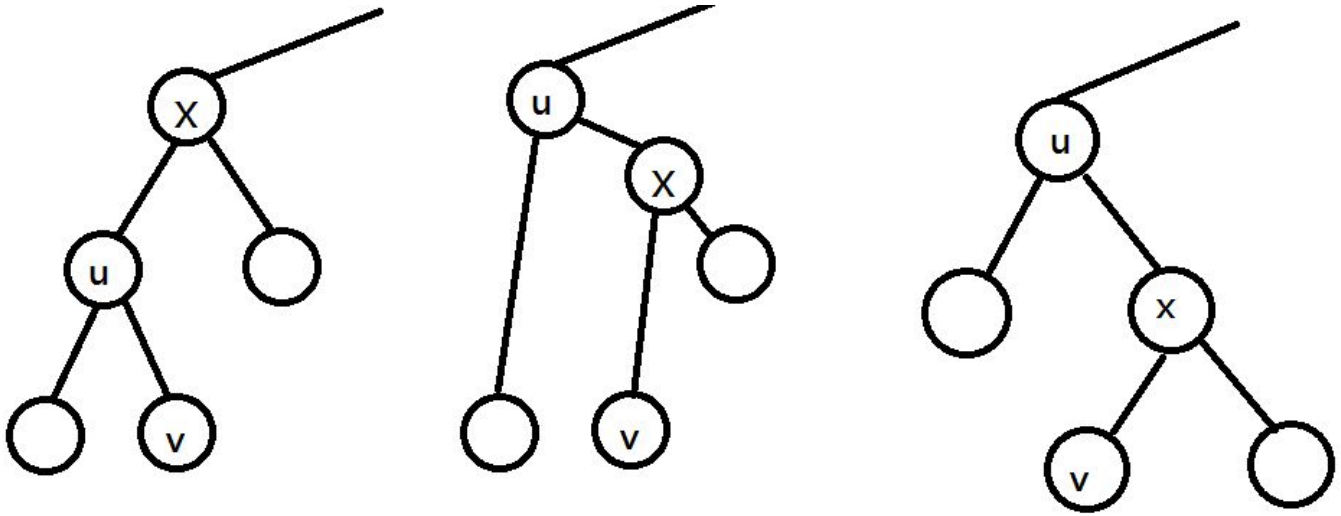
- Former approach
  - Make the tree very balanced when
    - Insert OR
    - Delete
  - And make queries very fast

AVL

- Define balance value of a node to be the difference between height of left child and height of right child
  - NULL child is defined to have height 0!
- The tree is considered balance if every balance value is between -1 and 1
- Achieved by rotating trees

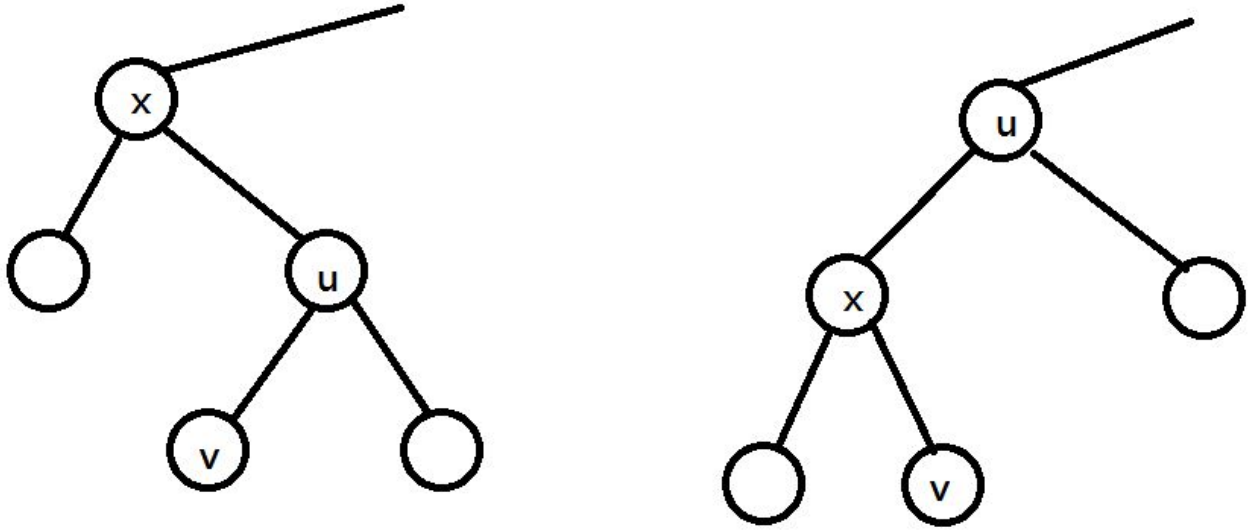
Rotating

- A process to change the structure of tree without affecting the ordering
- Consider the right rotation on x
-



Rotating

- Similarly we can define left rotation
-



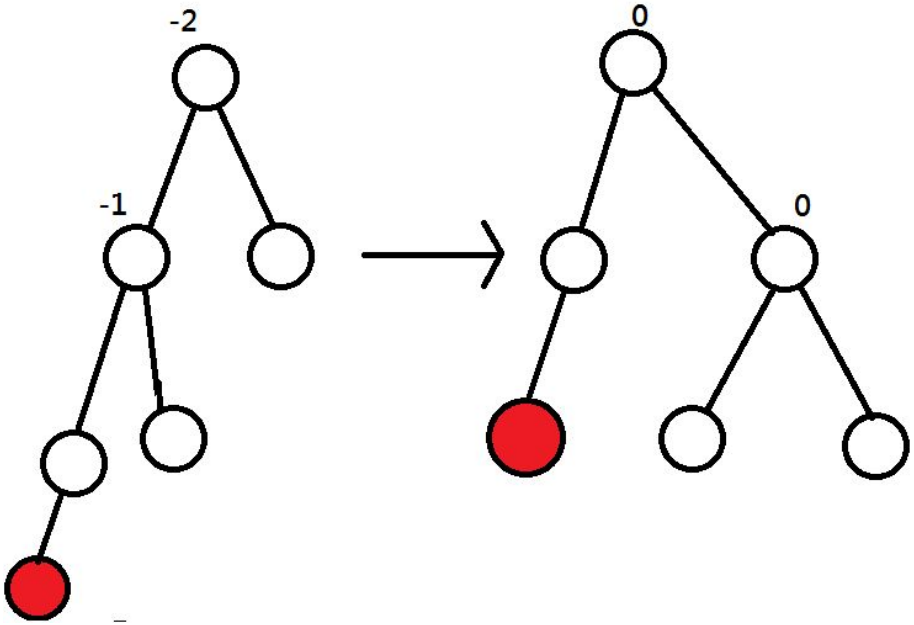
AVL

- Consider inserting a node to a tree
- It will affect all the balance values on the path to the root
- Fix the one farthest to the root first.

AVL

Case 1: added node at left subtree of left child

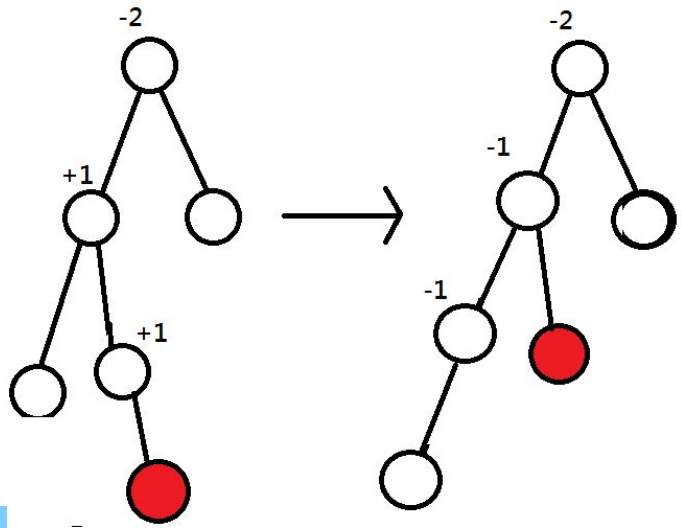
Do a right rotation and we can now fix the ancestors



AVL

Case 2: added node at right subtree of left child

Do a left rotation **on left child**, and then a right rotation on itself and we can now fix the ancestors

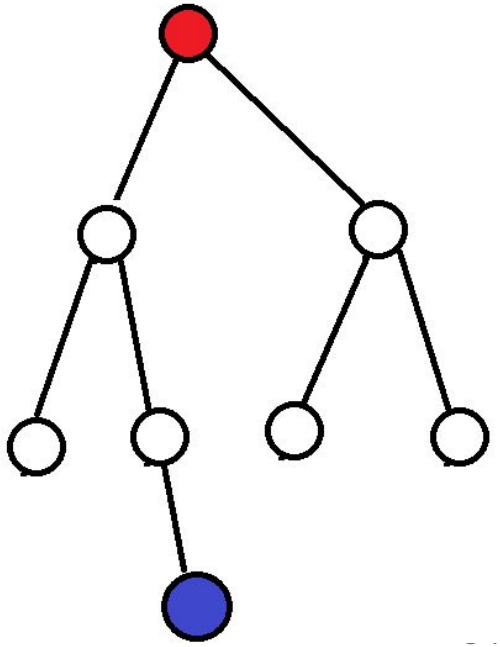


AVL

- Case 3 and 4 (if we insert) the right child)can be solved by considering the symmetry
- What if deletion?

AVL

- Idea: if the deleted node is not a leaf, exchange it with the immediate previous (or last) element
- Then we can treat as deleting leaves, which can be done similarly as insertion.



AVL

- Height of tree: ~O(lg N)
- Therefore each operation, which depends on height of tree, is in O(lg N)
- Now the tree is balanced!

Misc

- How to recalculate balance factor
- How to get the rank of the node
- How to find immediately previous element

Agenda

Binary-indexed Tree (BIT)

Sparse Table

AVL tree

Treap

Treap

- Random solution to the problem
- Use the property that random binary trees are expected to be almost balanced

Treap

- Consider giving each node a priority
- In addition to the ordered data property of the tree,
  - Priority data of the tree need to be a heap, too!

Treap

- Consider insertion
  - Property may be broken
  - Solved by rotation!

Treap

- Idea is easy:
  - Say the right child's priority is smaller than itself
  - We do right rotation and we check if the ancestors are good to go or not
  - Done. (already!)

Treap

- Consider deletion
  - Swap with immediate previous element to make it a leaf
  - Now the swapped element violates the heap rule
  - Swap it down when approaite