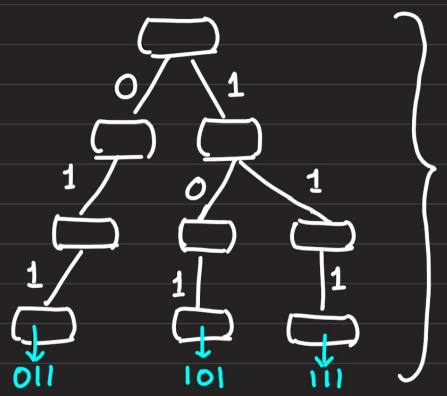
Bit Trie

3 → 011 , 5 → 101 , 7 → 111



mainly used for XOR related problem

Implementation

```
struct node{
    node* child[2];
    int cnt;
    node(){
        child[0] = NULL;
        child[1] = NULL;
        cnt = 0;
    }
};
```

```
struct trie
{
    node* root;
    trie(){
        root = new node();
    }
    void insert(int n)
    {
        node *cur = root;
        for(int i=30;i>=0;i--)
        {
            cur->cnt++;
            int x = (n&(1<<i)) ? 1: 0;
            if(cur->child[x]==NULL){
                cur->child[x] = new node;
            }
            cur = cur->child[x];
        }
        cur->cnt++;
    }
    void del(int n)
    {
        node *cur = root;
        for(int i=30;i>=0;i--)
        {
            cur->cnt--;
            int x = (n&(1<<i)) ? 1: 0;
            if(cur->child[x]==NULL){
                cur->child[x] = new node;
            }
            cur = cur->child[x];
        }
        cur->cnt--;
    }
};
```

Max XOR pair

```
#include<bits/stdc++.h>
using namespace std;

using lli = long long int;
using ii = pair<lli,lli>;

struct node{
   node *child[2];
   lli cnt[2];
   node(){
      child[0] = NULL;
      child[1] = NULL;
      cnt[0] = 0; cnt[1] = 0;
   }
};

struct trie{
   node *root;
   trie(){
      root = new node();
   }
   void insert(lli n){
      node *cur = root;
      for(int i=31;i>=0;i--)
      {
         int x = (n&(1ll<<i)) ? 1 : 0;
         cur->cnt[x]++;
         if(cur->child[x]==NULL){
            cur->child[x] = new node();
         }
         cur = cur->child[x];
      }
      cur->cnt[0]++;
   }
   void del(lli n){
      node *cur = root;
      for(int i=31;i>=0;i--)
      {
         int x = (n&(1ll<<i))?1:0;
         cur->cnt[x]--;
         cur = cur->child[x];
      }
      cur->cnt[0]--;
   }
   lli query(int n){
      node *cur = root;
      lli ans = 0;
      for(int i=31;i>=0;i--)
      {
         int x = (n&(1ll<<i)) ? 1 : 0;

         if(cur->child[(!x)]!=NULL && cur->cnt[(!x)]>0){
            ans += (1ll<<i);
            cur = cur->child[(!x)];
         }else{
            cur = cur->child[x];
         }
      }
      return ans;
   }
};
```

Max XOR subarray

```cpp
void solve()
{
    int n;cin>>n;
    int pre_xor = 0, ans = 0;
    int x;
    trie t;
    t.insert(0);
    for(int i=0;i<n;i++)
    {
        cin>>x;
        pre_xor ^= x;
        ans = max(ans,x);
        t.insert(pre_xor);
        ans = max(ans,t.query(pre_xor));
    }
    cout<<ans<<'\n';
}
```

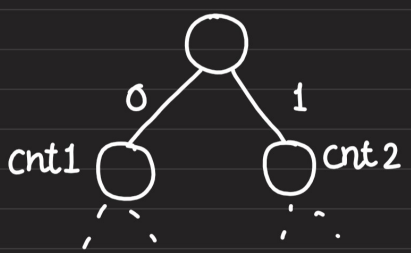
Insert and Query functions are same

for current pre-xor, we want the prev-pre-xor which gives the maxium value.

```cpp
//create a DS, that support
//+X, -X: insertion and deletion of X
//? Y K: find the number of X in the DS, s.t, (X^Y) <= K
```

We will maintain the count for insertion & deleting.

x = a
y = b
k = c



	k = 0	k = 1
b = 0	go to left / 0	go to right + cnt 1
b = 1	go to right / 1	go to left + cnt 2

$x \oplus y \leq k$

Subarrays with xor less than k

? Ask Doubt

Time-Limit: 5 sec Score: 0/100 Difficulty : ★★★★

Memory: 256 MB Accepted Submissions: 100 Relevant For: AZ-202 AZ-301

Description

Given an array of positive integers you have to print the number of subarrays whose XOR is less than K. Subarrays are defined as a sequence of continuous elements Ai, Ai+1, ..., Aj . XOR of a subarray is defined as Ai ^ Ai+1 ^ ... ^ Aj.

Input Format

The first line contains an integer T (1 ≤ T ≤ 10), the number of test cases.

The first line of each test case contains 2 space-separated integers N,k 1 ≤ N ≤ 10^5, 1 ≤ k ≤ 10^6.

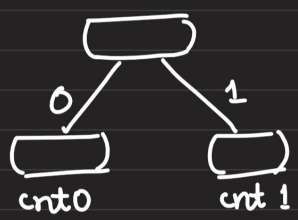
The second line of each test case contains N space-separated integers, 1≤ Ai ≤ 10^5

Soln: $a_1\ a_2\ a_3\ a_4 \mid a_5$

preXor = $a_1 \hat{} a_2 \hat{} a_3 \hat{} a_4 \hat{} a_5$

(a_5)

trie $a_1,\ a_1 \hat{} a_2,\ \ldots,\ a_1 \hat{} a_2 \hat{} a_3 \hat{} a_4$



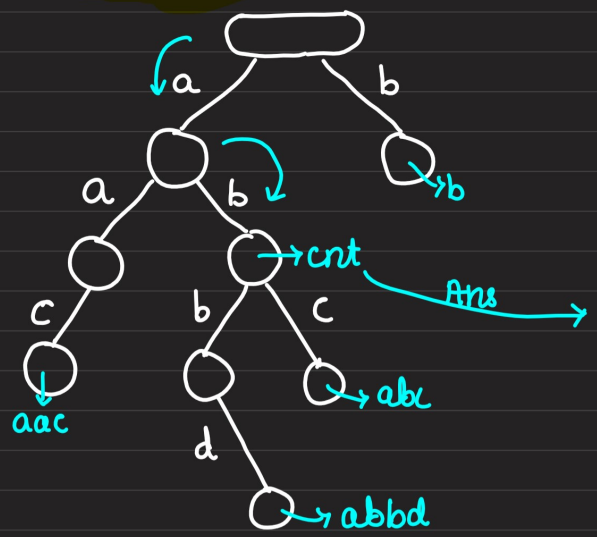
last bit	K = 0	K = 1
pXor = 0	move to 0	cnt 0 + move to 1
pXor = 1	move to 1	move to 0 + cnt 1

```
lli query(lli n, lli k)
{
    node *cur = root;
    lli ans = 0;
    for(int i=20;i>=0;i--)
    {
        int pl = (n&(1<<i)) ? 1 : 0;
        int kl = (k&(1<<i)) ? 1 : 0;

        if(cur==NULL) break;

        if(pl==0 && kl==0){
            cur = cur->child[0];
        }
        else if(pl==0 && kl==1){
            if(cur->child[0]) ans += cur->child[0]->cnt;
            cur = cur->child[1];
        }
        else if(pl==1 && kl==0){
            cur = cur->child[1];
        }
        else{
            if(cur->child[1]) ans += cur->child[1]->cnt;
            cur = cur->child[0];
        }
    }
    return ans;
}
```

Insert function is same

```
void solve()
{
    lli n,k;cin>>n>>k;
    vector<lli> a(n);
    for(int i=0;i<n;i++) cin>>a[i];

    trie t; t.insert(0);
    lli pre_xor = 0;
    lli ans = 0;
    for(int i=0;i<n;i++)
    {
        pre_xor^=a[i];
        ans += t.query(pre_xor,k);
        t.insert(pre_xor);
    }
    cout<<ans<<'\n';
}
```

String Trie



b
a a c
a b c
a b b d

this is a prefix trie

ans → No. of strings with "ab" as prefix

```
struct trie{
    node *root;
    trie(){
        root = new node();
    }
    void insert(string s) //O(|S|) both time and memory
    {
        node *cur = root;
        for(int i=0;i<s.size();i++){
            cur->prefix++;
            int x = s[i] - 'a';
            if(cur->child[x]==NULL){
                cur->child[x] = new node();
            }
            cur = cur->child[x];
        }
        cur->wend.push_back(s);
    }
    void deletion(string s) //O(|S|) both time and memory
    {
        node *cur = root;
        for(int i=0;i<s.size();i++){
            cur->prefix--;
            int x = s[i] - 'a';
            cur = cur->child[x];
        }
        cur->wend.pop_back();
    }
};
```

Q1. Search – Recomendations

Given a vector of strings. Solve for the Queries of the form

? S K : Find the lexiographically smallest top K results with prefix match of S.

eg: a a
a a c
a c d
c d

S = "a"
K = 2

Ans: aa, aac

Design an efficient algorithm to handle Q queries.

Soln:

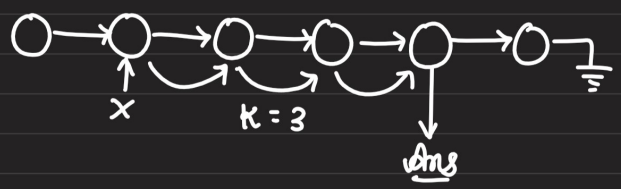
```
void dfs(node *cur,vector<string> &ans,int &k)
{
    if(ans.size()==k) return;
    if(cur->wend.size()!=0){
        ans.push_back(cur->wend[0]);
    }
    for(int i=0;i<26;i++){
        if(cur->child[i]) dfs(cur->child[i],ans,k);
    }
}
void query(string s, int k)
{
    node *cur = root;
    for(int i=0;i<s.size();i++){
        int x = s[i]-'a';
        cur = cur->child[x];
    }
    //only look for elements in the subtree of cur
    vector<string> ans;
    dfs(cur,ans,k);
    for(int i=0;i<ans.size();i++){
        cout<<ans[i]<<' ';
    }cout<<'\n';
}
```

INSERT FUNCTION REMAINS SAME

Binary Lifting

Given a sequence of structures with no random access, build a structure to efficiently answer queries of the form

? x K → Find the K^{th} element after the Xth one.



Soln: We can do the brute force but that will take O(N) per query. O(QN)

Also, one approach is to do pre-processing for each (X, K) which will take $O(N^2)$ and the we can answer queries in O(1)

TC : $O(N^2)$

Optimal Approach

Instead of saving for every i, we save for queries

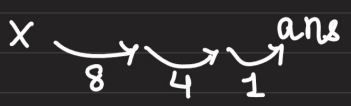
arr[x][i] = Q(x, 2^i)

For this pre-processing, it will only take O(NlogN)

MC & TC : O(NlogN)

NOW,

Breaking the K } say K = 13, $(\underbrace{1101}_{})_2$



O(logK) per query.

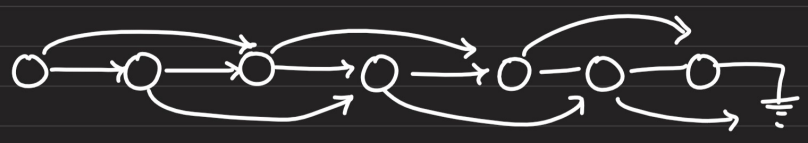
Ans = next (next (next (x, 3), 2) , 0)

2^3 2^2 2^0

Building next(x,i)

1) for every node calculate the imediate next element, i.e, next(x,0) $\forall$ x

2) next(x,i) = next(next(x,i-1), i-1) } similar to DP



```
for (i = lg, i>=0, i--)
    if (k & (1 << i))
        x = next(x,i)
```

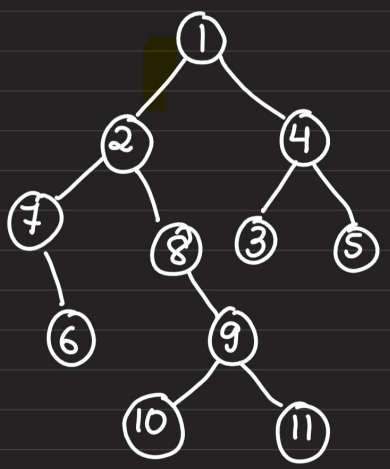
} Query processing

LCA (Classical LCA problem)

? u v

6 → {6, 7, 2, 1} ←
5 → {5, 4, 1} ←

} lca 1



Direct way - write ancestral chain for 6 & 5.

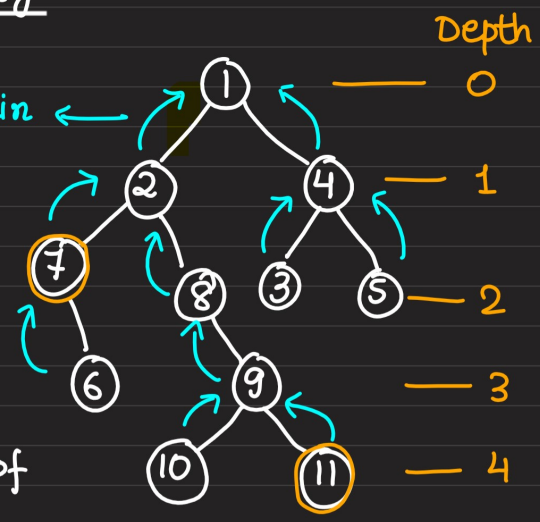
{ go from the end & the last node which is same is the LCA

10 → {10, 9, 8, 2, 1} ←
11 → {11, 9, 8, 2, 1} ←

} lca 9

But this is O(N) per query.

Recognising Chains for Lifting



Step 1

Bringing to same level:

eg: ? 7 11

Bring 11 to depth of 7

move up in the ancestor of 11 by (4-2), i.e, 2 steps

Here, we will get 8

Step 2: Jump simultaneously till not same.

Jumping will occur in powers of 2. We will jump only if the ancestors of u & v (bought to same level) are not same and it does not goes beyond the tree root.

Then we can recursively calculated the LCA of new ancestors.

```
//Best implementation
void dfs(int nn,int pp, int dd)
{
    depth[nn] = dd;
    par[nn][0] = pp;

    //note if it goes beyond the tree we are making the ancestor or 2^i th parent as 0
    for(int i=1;i<20;i++){
        par[nn][i] = par[par[nn][i-1]][i-1];
    }

    for(auto v: g[nn]){
        if(v!=pp){
            dfs(v,nn,dd+1);
        }
    }
}
```

```cpp
//O(20) per query
int lca(int u, int v){
    if(depth[u]<depth[v]){
        swap(u,v);
    }
    //uplifting
    int diff = depth[u] - depth[v];
    for(int i=19;i>=0;i--)
    {
        if((diff&(1<<i))){
            u = par[u][i];
        }
    }
    if(u==v) return u;
    for(int i=19;i>=0;i--)
    {
        if(par[u][i]!=par[v][i])
        {
            v = par[v][i];
            u = par[u][i];
        }
    }
    return par[u][0];
}
```

```cpp
void solve()
{
    int n;cin>>n;
    g.resize(n+1); par.assign(n+1,vector<int>(20,0));
    depth.assign(n+1,0);
    for(int i=0;i<n-1;i++)
    {
        int u, v; cin>>u>>v;
        g[u].push_back(v);
        g[v].push_back(u);
    }
    dfs(1,0,0);
    int q; cin>>q;
    while(q--){
        int u,v; cin>>u>>v;
        cout<<lca(u,v)<<'\n';
    }
}
```

LCA Twist

Difficulty	Time Limit	Memory	Score
[illegible]	5 sec	256000KB	100

80/80 XP 30/30

Description

Given a tree of N nodes and Q queries -

. In each query, three nodes are given x, y, z → find the lowest common ancestor of node x and y if z is a root of the tree.

Suppose we want to find the LCA of two nodes u and v in a tree T, with root x. Now, suppose we want to find the LCA of u and v with respect to a new root node z.

To do this, we can take advantage of the fact that we already know the LCA of u and v with respect to the original root node x. Let's call this node l.

The first step is to find the LCA of u and z, and the LCA of v and z, using the standard LCA algorithm. Let's call these nodes a and b, respectively.

LCA(u,v) = l
LCA(u,z) = a
LCA(v,z) = b

Now, there are two possible cases:

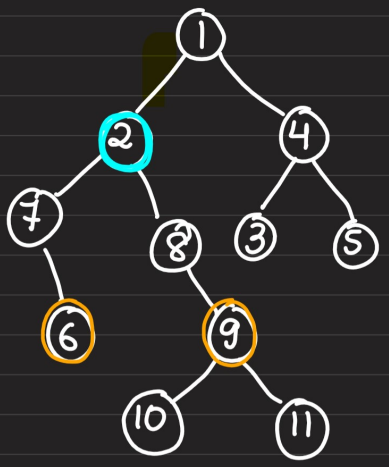
Case 1

If either a or b is equal to l, then z must lie in the branch of the other node. Specifically, if a == l, then z lies in the branch of v, so we return b as the new LCA of u and v with respect to z. Similarly, if b == l, then z lies in the branch of u, so we return a as the new LCA of u and v with respect to z.

Case 2

If neither a nor b is equal to l, then z lies outside the branches of both u and v. In this case, the LCA of u and v with respect to z is simply l, since l is the lowest common ancestor of u and v with respect to the original root node x.

This approach allows us to efficiently calculate the LCA of u and v with respect to any new root node z, without having to rebuild the entire LCA data structure from scratch.

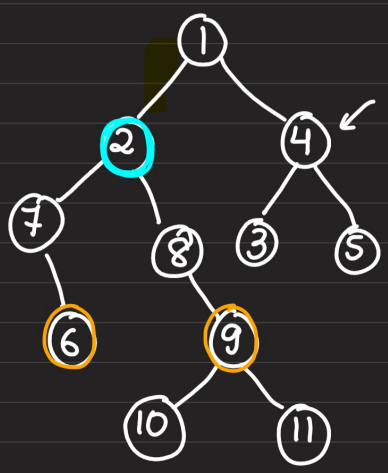


lca (6,9) = 2

new root = 8

lca (6,8) = 2
lca (9,8) = 8

lca (6,9, root = 8) = lca(9,8)

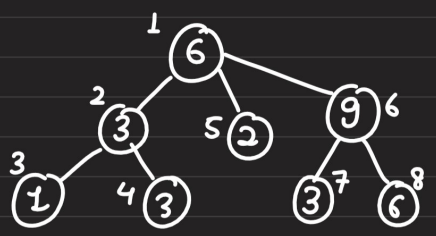


new root } LCA won't change.

Path GCD Query:

Find GCD of paths in a tree.

Q → ? u v } find gcd of path u to v



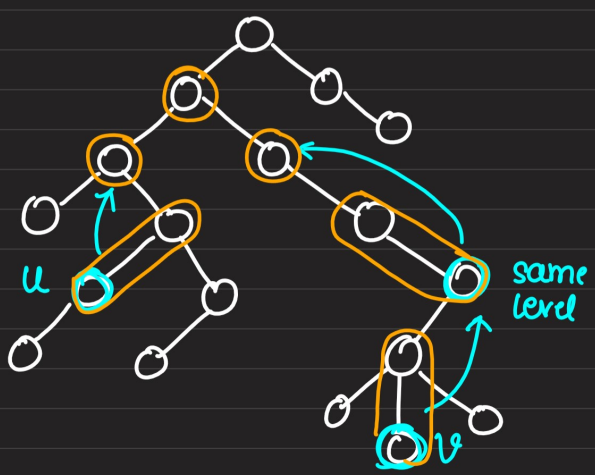
dp(x, i) = gcd of path from  } node

dp(x, 2) = gcd([1,4]) } note: that we do-not include 5 (because we want to calculate only for 2^i nodes)

$$dp(x, i) = \gcd\left[dp(x, i-1),\ dp(next(x, i-1),\ i-1) \right]$$

This, we can do for any other type of Query like sum, Xor, OR, AND, max, min, etc.

While making jumps, we will be aggregating the property together



Problem

Given a tree with **N** nodes (numbered **1** to **N**). For each valid **i**, the **i-th** node has a value of **A[i]** associated with it.

Your target is to handle the queries of the following **4** types -

- "1 X Y": Sum of values of all the nodes in the **path** from **X** to **Y**.
- "2 X Y": Sum of **pairwise OR** of every possible pair in the path from **X** to **Y** i.e. $\sum(a[i] \mid a[j])$ **where** $i < j$.
- "3 X Y": Sum of **OR** of all **triplet** in the path from **X** to **Y** i.e. $\sum(a[i] \mid a[j] \mid a[k])$ **where** $i < j < k$.
- "4 X Y": Sum of **OR** of all groups of type **(a,b,c,d)** in the path from **X** to **Y** i.e. $\sum(a[i] \mid a[j] \mid a[k] \mid a[l])$ **where** $i < j < k < l$.

For each query answer can be very large so print it MODULO **1000000007 (1e9 + 7).**

- Its solution depends on bits of every node value i.e. A[i].
- First we will write a standard LCA (LINK) code using binary lifting for Path queries.
- Use Dp to calculate number of set bits at each position in binary form from root to every other node i.e. Dp[bit][node] gives us number of set bits at position bit in binary form of A[i]'s in a path from root to node.
- Now answer answer can be calculated using Combinatorics
- for each bit position calculate number of set bits for all the nodes present in the path, similary we can count number of unset bits for every node in the path. Let number of set bits be n and unset bits be m.
- z X Y

1. for z = 1 => we can calculate answer as count of set bits mutiply by (1<<bit)
2. for z = 2 => answer for every bit can be calculated as (count of set bits in path = n,count of unset bits = m) ans = (1<<bit)*(nC2 + nC1*mC1)
3. for z = 3 => (1<<bit) (nC3 + nC2*mC1 + nC1*mC2)
4. similary for z =4

Prefix Sum on Tree

```cpp
//prefix[node] = distance to get to that node from root, i.e, 1
//note prefix will work only if the the operation is sum or xor
//for other types of queries use the previous approach only

void dfs(int nn, int pp, int dd)
{
    par[nn][0] = pp;
    depth[nn] = dd;
    prefix[nn] = prefix[pp] + weight[{nn,pp}];

    for(int i=1;i<20;i++){
        par[nn][i] = par[par[nn][i-1]][i-1];
    }

    for(auto v: g[nn]){
        if(v.F!=pp){
            dfs(v.F,nn,dd+1);
        }
    }
}
```

```cpp
#include<bits/stdc++.h>
using namespace std;
using ii = pair<int,int>;
#define F first
#define S second
vector<vector<ii>> g;
vector<vector<int>> par;
vector<long long int> prefix;
vector<int> depth;
map<ii,int> weight;
```

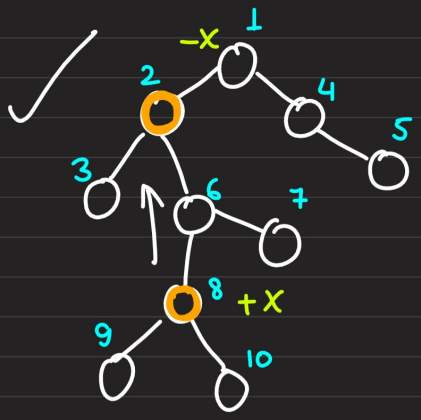
```cpp
long long int path_sum(int u, int v)
{
    int temp1 = u, temp2 = v;
    if(depth[u]<depth[v]){
        swap(u,v);
    }
    int diff = depth[u] - depth[v];
    for(int i=19;i>=0;i--)
    {
        if((diff&(1<<i)))
        {
            u = par[u][i];
        }
    }
    if(u==v){ //u is the lca
        return prefix[temp1] - prefix[u];
    }

    for(int i=19;i>=0;i--)
    {
        if(par[u][i]!=par[v][i])
        {
            u = par[u][i];
            v = par[v][i];
        }
    }
    int lca = par[u][0];
    return prefix[temp1] + prefix[temp2] - 2*prefix[lca];
}
```

```cpp
void solve()
{
    int n;cin>>n;
    g.resize(n+1); depth.assign(n+1,0);
    prefix.assign(n+1,0);
    par.assign(n+1,vector<int>(20,0));

    for(int i=0;i<n-1;i++)
    {
        int u,v,wt; cin>>u>>v>>wt;
        weight[{u,v}] = wt;
        weight[{v,u}] = wt;

        g[u].push_back({v,wt});
        g[v].push_back({u,wt});
    }
    weight[{1,0}] = 0;
    dfs(1,0,0);

    int q; cin>>q;
    while(q--){
        int u,v;
        cin>>u>>v;
        cout<<path_sum(u,v)<<'\n';
    }
}
```

Partial Sum on Trees

val[node] += val[par[node]]

will not work as +X will also get shifted to nod 3



push up → it's fixed that value is shifted only to parents.