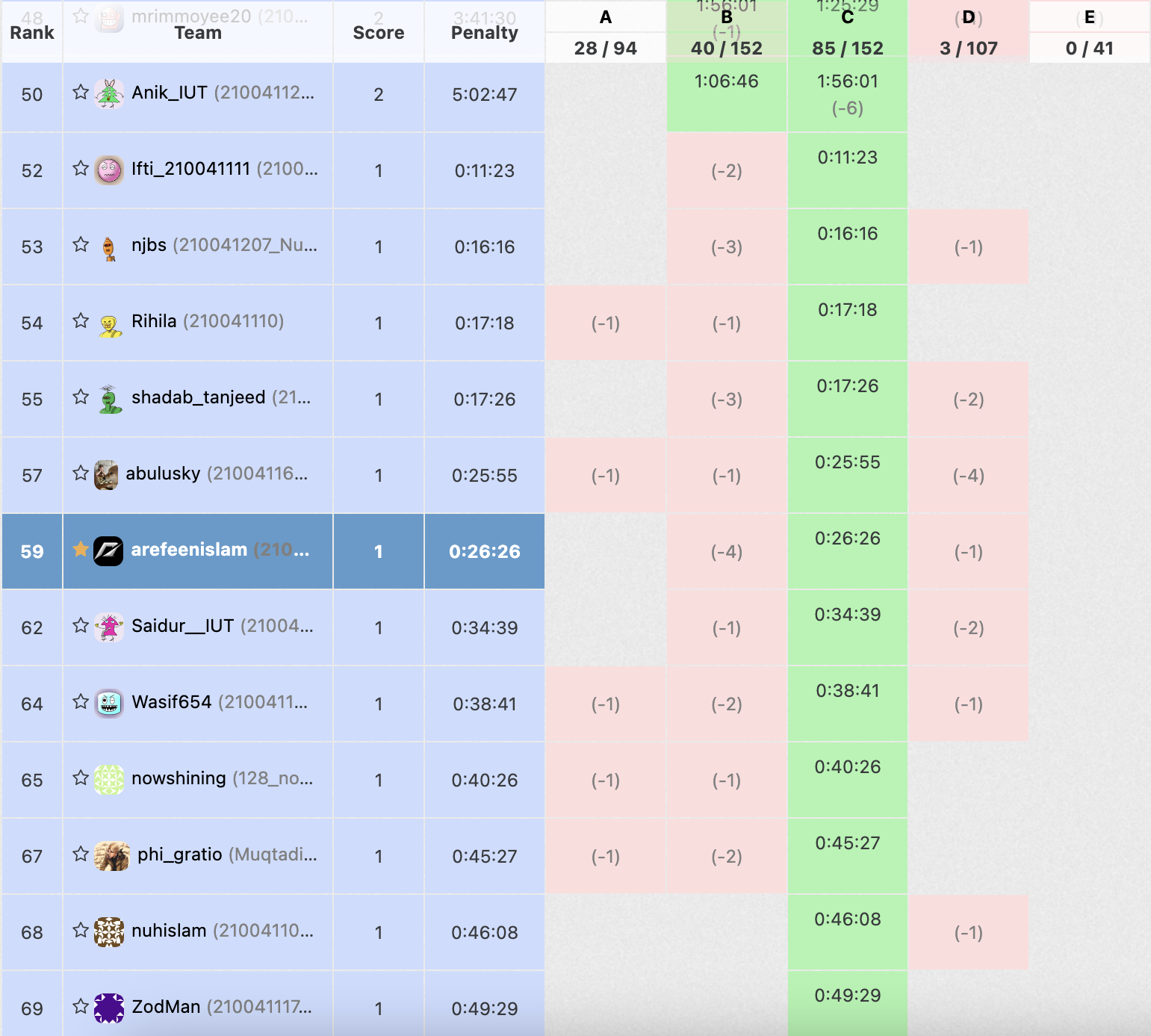
Rafiul Arefeen Islam

210041114 | CSE ’21 section 1 (1B)

Improvement Assignment

For CSE 4304 Mid exam

# 



# **A – Bangladesh Parliament**

## Solution Approach:

The problem statement essentially describes a Binary Search Tree. The given diagram illustrates this further. The entrance of each MP is like insertion in BST. Their speech order for even and odd sessions are different. Both are, however, variations of Post Order Traversal.   
For even session it is: Left wing -> Right wing -> Chairman  
For odd session it is: Right wing -> Left wing -> Chairman

For the input, the speech order is given for odd session. The output should be the speech order for the even session. Here the main challenge is to reconstruct the original BST. Once that is done, any kind of traversal is very easy. To reconstruct the BST we need to reverse what is done by the post order traversal. This can easily be done by inserting the input backwards. This works because when we insert backwards, we are inserting in this order: Parent, Right, Left.

This can be illustrated with a small example. Considering a small tree:

20

/ \

10 30

By the even session traversal (our input), we get:

10, 30, 20

Then if we insert in backwards order:

Step 1:

20

Step 2:

20

\

30

Step 3:

20

/ \

10 30

So, as we can see we get back our original tree. This will be true for bigger, complex trees as well. Then for our output we just traverse this tree in Right, Left, Parent order.

## Code:

#include <iostream>

#include <stack>

#include <queue>

class Node

{

public:

int data;

Node \*left;

Node \*right;

Node \*parent;

Node(int \_data) : data(\_data), left(nullptr), right(nullptr), parent(nullptr) {}

};

class BST

{

private:

Node \*root;

void \_evenTraversal(Node \*current)

{

if (current != nullptr)

{

\_evenTraversal(current->right);

\_evenTraversal(current->left);

std::cout << current->data << std::endl;

}

}

public:

BST() : root(nullptr) {}

void insert(int val)

{

Node \*newNode = new Node(val);

if (root == nullptr)

{

root = newNode;

return;

}

else

{

Node \*current = root;

Node \*up = nullptr;

while (current != nullptr)

{

up = current;

if (val < current->data)

current = current->left;

else

current = current->right;

}

if (val < up->data)

{

newNode->parent = up;

up->left = newNode;

}

else

{

newNode->parent = up;

up->right = newNode;

}

}

}

void evenTraversal()

{

\_evenTraversal(root);

}

};

int main()

{

int n, val;

std::vector<int> vec;

BST tree;

std::cin >> n;

while (n--)

{

std::cin >> val;

vec.push\_back(val);

}

for(int i = vec.size()-1; i>=0; i--)

{

tree.insert(vec[i]);

}

tree.evenTraversal();

return 0;

}

# **B – Berries**

## Solution Approach:

It is given that the ith group contains ai berries. So, for a given number of berries for each group it is easy to know up to how many berries the group contains.

For example, given 5 groups and 2, 7, 3, 4, 9 as number of berries in groups we can say by summing up that:

Group 1 contains berries up to number 2

Group 2 contains berries up to number 9

Group 3 contains berries up to number 12

Group 4 contains berries up to number 16

Group 5 contains berries up to number 25

Now our job is for a given berry number, to find in which group the berry resides in. For example, berry number 10 resides in group 3. This could be achieved by traversing an array that contains the sum up to each group and then stopping when the sum is equal to or greater than the berry number. This approach however takes a lot of time; thus, we have to use binary search instead.

In the solution the sum of berries up to a particular group is stored in the sumUp vector while the numbers of tasty berries are stored in the tasty vector. Then we take each item of the tasty vector and perform a binary search on the sumUp vector. The part to be noted here is that an ordinary binary search will not work here since it is not guaranteed that the sum values will match with the tasty berry numbers. Hence, we have to take the lower bound of the binary search. We achieve this by using the lower\_bound function. We take the lower bound and not the higher bound because the number of the tasty berry will always be equal to or LESS than the sum value of the group it resides in. It returns an iterator pointing to our desired group. We just subtract the beginning iterator to get the group number and then add 1 because vectors are zero indexed. This is our output.

## Code:

#include <bits/stdc++.h>

struct data

{

int group;

int upTo;

};

int main()

{

int n, m, x;

int sum;

std::vector<int> tasty;

std::vector<int> sumUp;

std::cin >> n;

std::cin >> x;

sumUp.push\_back(x);

n--;

while (n--)

{

std::cin >> x;

sumUp.push\_back(x + sumUp.back());

}

std::cin >> m;

while (m--)

{

std::cin >> x;

tasty.push\_back(x);

}

for (int i = 0; i < tasty.size(); i++)

{

int res = (std::lower\_bound(sumUp.begin(), sumUp.end(), tasty[i]) - sumUp.begin());

std::cout << res + 1 << std::endl;

}

return 0;

}

# **D – Three Buttons**

## Solution Approach:

In this problem we have to simulate keyboard input and display the final result. The keys are the lowercase alphabet and ‘[‘, ‘]’, ‘<’ keys. ‘[‘ acts as home, ‘]’ as end and ‘<’ as backspace.

For our input we have a string called input. For our output we have a deque called output. We also have a char stack called temp. This temp stack will be used for insertion at the front since anything inserted at the front needs to be pushed front in reverse order in the output deque.

We also have two Boolean variables; isAtEnd and isAtStart. These two help us to identify three cases that can happen:

1. isAtEnd == true and isAtStart == false: Cursor is at end
2. isAtEnd == false and isAtStart == true: Cursor is at start
3. Both are false: Cursor is somewhere in the middle

Based on these three conditions we perform insertion and deletion. For ‘[‘ and ‘]’ we just set the Boolean variables to the correct values. For ‘<’ we perform deletion:

1. If cursor is at end then pop\_back() from the output deque
2. If cursor is in the middle then we pop() from the temp stack
3. If cursor is at the start we do nothing

Otherwise for lowercase alphabet we perform insertion. We get three cases here as well:

1. If cursor is at end then first we add the contents of the stack to the front of the output deque in reverse order then push\_back() the new char in the deque
2. If cursor is in the middle then we push the char in the temp stack
3. If cursor is at the start we do nothing we first add the contents of the stack to the front of the output deque in reverse order then push the new char to the temp stack. Finally we set isAtStart to false since the cursor is in the middle now

It is to be noted that for every pop operation empty() is checked. Then we add any remaining items in the stack to the front of the deque in reverse. Finally, we print the contents of the deque.

## Code:

#include <iostream>

#include <deque>

#include <stack>

int main()

{

int t;

char ch;

std::string input;

std::deque<char> output;

std::cin >> t;

std::cin.ignore();

while (t--)

{

bool isAtEnd = true;

bool isAtStart = false;

input.erase();

output.clear();

std::stack<char> temp;

std::getline(std::cin, input);

for (int i = 0; i < input.length(); i++)

{

if (input[i] == ']')

{

isAtEnd = true;

isAtStart = false;

}

else if (input[i] == '[')

{

isAtEnd = false;

isAtStart = true;

}

else if (input[i] == '<')

{

if (isAtEnd && !isAtStart)

{

if (!output.empty())

output.pop\_back();

}

else if (!isAtEnd && !isAtStart)

{

if (!temp.empty())

temp.pop();

}

else

continue;

}

else

{

if (isAtEnd && !isAtStart)

{

while (!temp.empty())

{

output.push\_front(temp.top());

temp.pop();

}

output.push\_back(input[i]);

}

else if (isAtStart && !isAtEnd)

{

while (!temp.empty())

{

output.push\_front(temp.top());

temp.pop();

}

temp.push(input[i]);

isAtStart = false;

}

else

temp.push(input[i]);

}

}

while (!temp.empty())

{

output.push\_front(temp.top());

temp.pop();

}

for (int i = 0; i < output.size(); i++)

{

std::cout << output[i];

}

std::cout << std::endl;

}

return 0;

}