#include <stdio.h>

#include <stdlib.h>

#define MAX\_BLOCKS 100

typedef struct {

    int start;

    int end;

} MemoryBlock;

MemoryBlock freeBlocks[MAX\_BLOCKS];

int freeCount = 0;

// Function to print the largest free block

void printLargestFreeBlock() {

    int max\_size = 0;

    MemoryBlock largest = {-1, -1};

    for (int i = 0; i < freeCount; i++) {

        int size = freeBlocks[i].end - freeBlocks[i].start;

        if (size > max\_size) {

            max\_size = size;

            largest = freeBlocks[i];

        }

    }

    if (largest.start != -1) {

        printf("[%d %d]\n", largest.start, largest.end);

    }

}

// Function to merge adjacent free blocks

void mergeFreeBlocks() {

    for (int i = 0; i < freeCount - 1; i++) {

        if (freeBlocks[i].end == freeBlocks[i + 1].start) {

            freeBlocks[i].end = freeBlocks[i + 1].end;

            // Shift the remaining blocks

            for (int j = i + 1; j < freeCount - 1; j++) {

                freeBlocks[j] = freeBlocks[j + 1];

            }

            freeCount--;

            i--; // Recheck the merged block

        }

    }

}

// Function to allocate memory

void allocateMemory(int size) {

    for (int i = 0; i < freeCount; i++) {

        int blockSize = freeBlocks[i].end - freeBlocks[i].start;

        if (blockSize >= size) {

            printf("[%d %d]\n", freeBlocks[i].start, freeBlocks[i].start + size - 1);

            freeBlocks[i].start += size;

            if (freeBlocks[i].start == freeBlocks[i].end) {

                // If the block is fully allocated, remove it from the list

                for (int j = i; j < freeCount - 1; j++) {

                    freeBlocks[j] = freeBlocks[j + 1];

                }

                freeCount--;

            }

            mergeFreeBlocks(); // Merge if necessary

            printLargestFreeBlock();

            return;

        }

    }

}

// Function to add free memory block back and merge if necessary

void freeMemory(int start, int end) {

    freeBlocks[freeCount].start = start;

    freeBlocks[freeCount].end = end;

    freeCount++;

    mergeFreeBlocks(); // Merge adjacent blocks

    printLargestFreeBlock();

}

int main() {

    int n;

    scanf("%d", &n); // Input number of free blocks

    for (int i = 0; i < n; i++) {

        scanf("%d %d", &freeBlocks[i].start, &freeBlocks[i].end);

        freeCount++;

    }

    int size;

    scanf("%d", &size); // Input the size to allocate

    allocateMemory(size); // Allocate the requested memory size

    return 0;

}

#include <stdio.h>

#include <stdlib.h>

#define MAX\_MEETINGS 100

typedef struct {

    int start;

    int end;

} Meeting;

Meeting meetings[MAX\_MEETINGS];

int meetingCount = 0;

// Function to compare two meetings by their start time

int compare(const void\* a, const void\* b) {

    Meeting\* m1 = (Meeting\*)a;

    Meeting\* m2 = (Meeting\*)b;

    return m1->start - m2->start;

}

// Function to check if two meetings overlap

int isOverlapping(int start1, int end1, int start2, int end2) {

    return (start1 < end2 && start2 < end1);

}

// Function to book a meeting

void bookMeeting(int start, int end) {

    for (int i = 0; i < meetingCount; i++) {

        if (isOverlapping(start, end, meetings[i].start, meetings[i].end)) {

            printf("NA\n");

            return;

        }

    }

    meetings[meetingCount].start = start;

    meetings[meetingCount].end = end;

    meetingCount++;

    printf("Booked [%d %d]\n", start, end);

    // Sort meetings after booking

    qsort(meetings, meetingCount, sizeof(Meeting), compare);

}

// Function to cancel a meeting

void cancelMeeting(int start, int end) {

    int found = 0;

    for (int i = 0; i < meetingCount; i++) {

        if (meetings[i].start == start && meetings[i].end == end) {

            found = 1;

            for (int j = i; j < meetingCount - 1; j++) {

                meetings[j] = meetings[j + 1];

            }

            meetingCount--;

            printf("Canceled [%d %d]\n", start, end);

            break;

        }

    }

    if (!found) {

        printf("NA\n");

    }

}

// Function to list all currently booked meetings

void listMeetings() {

    if (meetingCount == 0) {

        printf("NA\n");

    } else {

        for (int i = 0; i < meetingCount; i++) {

            printf("[%d %d] ", meetings[i].start, meetings[i].end);

        }

        printf("\n");

    }

}

int main() {

    int n;

    scanf("%d", &n); // Number of operations

    for (int i = 0; i < n; i++) {

        char operation;

        int start, end;

        scanf(" %c", &operation); // Read operation (B, C, or L)

        if (operation == 'B') {

            scanf("%d %d", &start, &end);

            bookMeeting(start, end);

        } else if (operation == 'C') {

            scanf("%d %d", &start, &end);

            cancelMeeting(start, end);

        } else if (operation == 'L') {

            listMeetings();

        }

    }

    return 0;

}

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_NAME\_LENGTH 100

typedef struct Contact {

    int roll\_number;

    char name[MAX\_NAME\_LENGTH];

    struct Contact \*left, \*right;

} Contact;

// Helper function to create a new contact

Contact\* createContact(int roll\_number, const char\* name) {

    Contact\* new\_contact = (Contact\*)malloc(sizeof(Contact));

    new\_contact->roll\_number = roll\_number;

    strcpy(new\_contact->name, name);

    new\_contact->left = new\_contact->right = NULL;

    return new\_contact;

}

// Helper function to insert or update a contact in the BST

Contact\* addContact(Contact\* root, int roll\_number, const char\* name) {

    if (root == NULL) {

        return createContact(roll\_number, name);

    }

    if (roll\_number < root->roll\_number) {

        root->left = addContact(root->left, roll\_number, name);

    } else if (roll\_number > root->roll\_number) {

        root->right = addContact(root->right, roll\_number, name);

    } else {

        // If the contact already exists, update the name

        strcpy(root->name, name);

    }

    return root;

}

// Helper function to find the minimum contact in a subtree

Contact\* findMin(Contact\* root) {

    while (root->left != NULL) {

        root = root->left;

    }

    return root;

}

// Helper function to delete a contact in the BST

Contact\* deleteContact(Contact\* root, int roll\_number) {

    if (root == NULL) {

        return NULL;

    }

    if (roll\_number < root->roll\_number) {

        root->left = deleteContact(root->left, roll\_number);

    } else if (roll\_number > root->roll\_number) {

        root->right = deleteContact(root->right, roll\_number);

    } else {

        // Node to be deleted found

        if (root->left == NULL) {

            Contact\* temp = root->right;

            free(root);

            return temp;

        } else if (root->right == NULL) {

            Contact\* temp = root->left;

            free(root);

            return temp;

        }

        // If the node has two children, replace it with the minimum node from the right subtree

        Contact\* temp = findMin(root->right);

        root->roll\_number = temp->roll\_number;

        strcpy(root->name, temp->name);

        root->right = deleteContact(root->right, temp->roll\_number);

    }

    return root;

}

// Helper function to find a contact in the BST

void findContact(Contact\* root, int roll\_number) {

    if (root == NULL) {

        printf("NA\n");

        return;

    }

    if (roll\_number < root->roll\_number) {

        findContact(root->left, roll\_number);

    } else if (roll\_number > root->roll\_number) {

        findContact(root->right, roll\_number);

    } else {

        printf("%s\n", root->name);

    }

}

int main() {

    int n;

    scanf("%d", &n); // Read the number of operations

    Contact\* root = NULL; // Initialize the root of the BST

    for (int i = 0; i < n; i++) {

        char operation;

        int roll\_number;

        char name[MAX\_NAME\_LENGTH];

        scanf(" %c", &operation); // Read the operation

        if (operation == 'A') {

            // Add contact operation

            scanf("%d %s", &roll\_number, name);

            root = addContact(root, roll\_number, name);

        } else if (operation == 'D') {

            // Delete contact operation

            scanf("%d", &roll\_number);

            root = deleteContact(root, roll\_number);

        } else if (operation == 'F') {

            // Find contact operation

            scanf("%d", &roll\_number);

            findContact(root, roll\_number);

        }

    }

    return 0;

}

#include <stdio.h>

#define MAX\_USERS 100001

int parent[MAX\_USERS];  // parent[i] points to the parent of node i

int size[MAX\_USERS];    // size[i] stores the size of the set whose root is i

// Find function with path compression

int find(int x) {

    if (parent[x] != x) {

        parent[x] = find(parent[x]);  // Path compression

    }

    return parent[x];

}

// Union function with size tracking

void unionSets(int x, int y) {

    int rootX = find(x);

    int rootY = find(y);

    if (rootX != rootY) {

        // Union by size: attach the smaller tree under the larger tree

        if (size[rootX] < size[rootY]) {

            parent[rootX] = rootY;

            size[rootY] += size[rootX];

        } else {

            parent[rootY] = rootX;

            size[rootX] += size[rootY];

        }

    }

}

// Initialize the Union-Find structure

void initialize(int n) {

    for (int i = 1; i <= n; i++) {

        parent[i] = i;  // Each node is initially its own parent (self-grouped)

        size[i] = 1;    // Each group starts with size 1

    }

}

int main() {

    int n, q;

    scanf("%d %d", &n, &q);

    // Initialize the Union-Find data structure

    initialize(n);

    for (int i = 0; i < q; i++) {

        char query\_type;

        int x, y;

        scanf(" %c", &query\_type);

        if (query\_type == 'F') {

            // Friendship Formation: Merge two users' groups

            scanf("%d %d", &x, &y);

            unionSets(x, y);

        } else if (query\_type == 'C') {

            // Friendship Check: Are users x and y in the same group?

            scanf("%d %d", &x, &y);

            if (find(x) == find(y)) {

                printf("Yes\n");

            } else {

                printf("No\n");

            }

        } else if (query\_type == 'S') {

            // Group Size Query: Print the size of the group containing user x

            scanf("%d", &x);

            printf("%d\n", size[find(x)]);

        }

    }

    return 0;

}