Computer Science I Program 5: Games Dictionary (Binary Search Tree)

Please Check Webcourses for the Due Date

Please read the whole assignment before you start coding

Objective

Give practice with implementing standard binary search tree functions.
Give practice with functions in C.
Give practice with creating a design for a program without a given list of functions or structs.

Games Dictionary

Each of the NY Times games uses strings of letters (some words, some not words) in many contexts. The creators of each of the games would like a common database where each of the possible strings they could use are stored. That's where you come in! Now that you've learned how to implement a binary search tree, you can, in the average case, efficiently manage the storage and retrieval of these strings. Your binary search tree will be sorted in alphabetical order by string.

Due to various reasons, on occasion, some strings will no longer be in use for some games, so you must have the ability to delete strings as well. Also, as time goes on, new strings may be allowed for some games.

Your binary search tree of nodes will utilize the following structs:

```
#define NUMGAMES 6

typedef struct NYT_String {
    char* str;
    int allowed[NUMGAMES];
} NYT_String;

typedef struct BST_Node {
    NYT_String* ptr;
    BST_Node* left;
    BST_Node* right;
} BST_Node;
```

The games in question are the same 6 games from assignment 4, with the same codes. When a string is added or deleted, it will be added or deleted for a single game. If mys is of type NYT_String*, then mys->allowed[i] will be set to 1 if mys->str is a valid string for game i. Otherwise, mys->allowed[i] will be set to 0.

Structurally, nodes should only present in the binary search tree you maintain if they store a string that is allowed in at least one of the 6 possible games. This means a new node is ONLY inserted

if the string in question is not currently being used an any of the 6 games, and a node is physically deleted only when a string is removed causing none of the 6 possible games to use it.
Thus, in many cases, adding or deleting a string for a single game will involve no structural changes to the binary search tree. Rather, this will often just involve searching for the node involving the string and changing the associated array index in the allowed component to 1 (if the string is being added to a game), or back to 0 (if a string is being removed from a game).

Your program will need to process the following operations and queries:

1. Add a string to a game.
2. Remove a string from a game.
3. Given a string, determine which game(s), if any, it's allowed in.
4. Return an alphabetically sorted list of all strings allowed for a particular game.
5. Return the number of strings of a particular length allowed in a particular game.
6. Given a string, return the next string alphabetically, that appears in any game.

Input

The first line of input contains a single positive integer, n ($1 \leq n \leq 10^6$), representing the number of operations for the input case. Note: Although there may be up to 1,000,000 operations, it is guaranteed that there will not be more than 200,000 nodes ever created in the binary search tree.

Each of the next n lines indicates information about a single operation/query. Here are the formats for each of those types of operations/queries:

The first integer on each of these n lines will be a single positive integer, t ($1 \leq t \leq 5$), representing the type of query, with $t = 1$ indicating adding a string for a game, $t = 2$, indicating deleting a string for a game, $t = 3$ indicating a query about a string, $t = 4$, indicating a query about all of the allowable strings for a particular game, and $t = 5$, indicating a query about the number of strings of a particular length allowed in a particular game.

If $t = 1$, the second token on the line will be a non-negative integer g ($0 \leq g \leq 5$), representing which game a string will be added to. Finally, the third token on the line will be the string in question. This string, and all strings in the input will be in between 1 and 19 lowercase letters long.

If $t = 2$, the second token on the line will be a non-negative integer g ($0 \leq g \leq 5$), representing which game a string will be deleted from. The third token on the line will be the string to be deleted. **It is guaranteed that this string is currently a valid string in the game at the point of this operation.**

If $t = 3$, the second token on the line will be a string for which the query is being made. It's possible that this string is not allowed in any of the games.

If $t = 4$, the second token on the line will be a non-negative integer g ($0 \leq g \leq 5$), representing the game for the query. **The total number of times this query will be made is no more than 10.**

If $t = 5$, the second token on the line will be a non-negative integer g ($0 \leq g \leq 5$), representing the game for the query, and the third token on the line will be a positive integer, L ($1 \leq L \leq 19$), representing the length of strings for the query. **The total number of times this query will be made is no more than 10.**

If $t = 6$, the second token on the line will be a string for which the query is being made. It's possible that this string is not allowed in any of the games.

Finally, the input will be such that the total height of the tree created will never exceed 100. In short, all test cases will be mimic randomly generated tree structures instead of forcing a worst case binary search tree.

Output

There is no output for operations of type #1 and #2.

For each query of type #3 (asking which games a particular string is used in), if there is no game in which the string is used, then output -1 on a line by itself. Otherwise, output, in numeric order, with a space following each integer, the games in which the queried string is allowed. For example, if the string "ranklist" is allowed in games 2, 3 and 5, then output the following on a line by itself:

```
2 3 5
```

Note that for ease of implementation, a space should be output after the last number outputted in the list in this case. (If -1 is the output, no space should be outputted after it on the line.)

For each query of type #4, output each string that satisfies the query, one string per line, in alphabetical order. **Note: There is an implementation requirement that goes along with this query type.**

For each query of type #5, just output the desired answer (an integer) on a line by itself.

For each query of type #6, if there exists a string that is different than the query string and comes after it alphabetically in the whole tree, print out the very next string alphabetically, if no such string exists, print out "NO SUCCESSOR" on a line by itself.

Sample Input	Sample Output
21 1 0 hello 1 3 hello 1 5 hello 3 hello 3 bye 1 3 bye 4 3 5 3 3 2 3 bye 2 0 hello 4 0 5 3 5 5 3 4 2 3 hello 2 5 hello 5 3 5 1 2 apple 1 4 strawberry 6 help 6 strawberry 6 zoo	0 3 5 -1 bye hello 1 1 0 0 strawberry NO SUCCESSOR NO SUCCESSOR

Implementation Requirements/Run Time Requirements

Required Constants, Struct and Constant Array
Please use the following constants, struct and constant array:

```
#define MAXSIZE 19
#define NUMGAMES 6

typedef struct NYT_String {
    char* str;
    int allowed[NUMGAMES];
} NYT_String;

typedef struct BST_Node {
    NYT_String* ptr;
    BST_Node* next;
} BST_Node;
```

You will be required to make a binary search tree where each node is of type BST_Node. You must only add a new node when the string being added is not valid for any other game, and you must only delete a node when deleting a string from the last game in which it's valid. In all other cases, you must just navigate to the appropriate node and change its appropriate allowed setting. **Note: We will check this by looking at your code not its execution. It's absolutely possible to get the correct output without following our implementation specifications, but a decent portion of the grade will come from following the implementation specifications.**

For queries of type 4, while you could just run an inorder traversal of the tree and print as necessary, **for full** credit, you will be required to write a function that takes in a pointer to the root of the binary search tree, an integer indicating the game number, and a pointer to an integer which will store the length of the array returned. The function should return a char**, an array of strings storing the answers for the query in alphabetical order. Here is the required function prototype:

```
char** allStringsInGame(BST_Node* root, int gameNo, int* arrSize);
```

To implement this function, please initially allocate a char** of size 200,000, or the size of the number of nodes in the tree, copying in each string into this char** (using a deep copy so malloc just the right amount of space for each string and then copy it in). Then, before returning, realloc the char** so that it's the right size.

Runtime Requirements

1. Options #1, #2, #3, and #6 should run in O(h) time, where h is the current height of the tree.

2. Options #4 and #5 should run in O(n) time, where n is the current number of nodes in the tree.

Deliverables

1. Please submit a source file `gamesdictionary.c`, via Webcourses, for your solution to the problem.