LC 1

Code

```
struct sNode {
   char data;
   struct sNode* next;
};

// Function to push an item to stack
void push(struct sNode** top_ref, int new_data);

// Function to pop an item from stack
int pop(struct sNode** top_ref);

// Returns 1 if character1 and character2 are matching left
// and right Brackets
bool isMatchingPair(char character1, char character2)
{
   if (character1 == '(' && character2 == ')')
      return 1;
   else if (character1 == '{' && character2 == '}')
      return 1;
   else if (character1 == '[' && character2 == ']')
      return 1;
   else
```

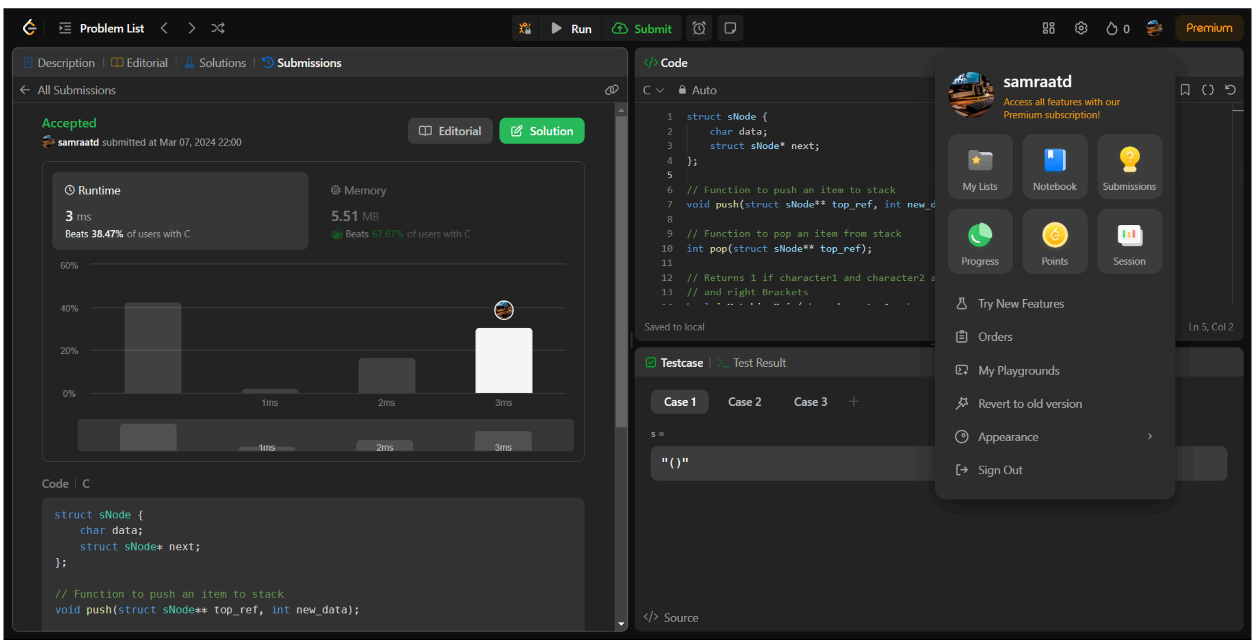
```
        return 0;
}

bool isValid(char* exp)
{
    int i = 0;

    struct sNode* stack = NULL;

    while (exp[i]) {

        if (exp[i] == '{' || exp[i] == '(' || exp[i] == '[')
            push(&stack, exp[i]);

        if (exp[i] == '}' || exp[i] == ')'
            || exp[i] == ']') {

            if (stack == NULL)
                return 0;

            // Pop the top element from stack, if it is not
            // a pair bracket of character then there is a
            // mismatch.
            // his happens for expressions like {(})
            else if (!isMatchingPair(pop(&stack), exp[i]))
```

```
            return 0;
        }
        i++;
    }

    // If there is something left in expression then there
    // is a starting bracket without a closing
    // bracket
    if (stack == NULL)
        return 1; // balanced
    else
        return 0; // not balanced
}

// Function to push an item to stack
void push(struct sNode** top_ref, int new_data)
{
    // allocate node
    struct sNode* new_node
        = (struct sNode*)malloc(sizeof(struct sNode));

    if (new_node == NULL) {
        printf("Stack overflow n");
        getchar();
        exit(0);
```

```
    }

    // put in the data
    new_node->data = new_data;

    // link the old list of the new node
    new_node->next = (*top_ref);

    // move the head to point to the new node
    (*top_ref) = new_node;
}

// Function to pop an item from stack
int pop(struct sNode** top_ref)
{
    char res;
    struct sNode* top;

    // If stack is empty then error
    if (*top_ref == NULL) {
        printf("Stack overflow n");
        getchar();
        exit(0);
    }
    else {
```

```
        top = *top_ref;
        res = top->data;
        *top_ref = top->next;
        free(top);
        return res;
    }
}
```

Output

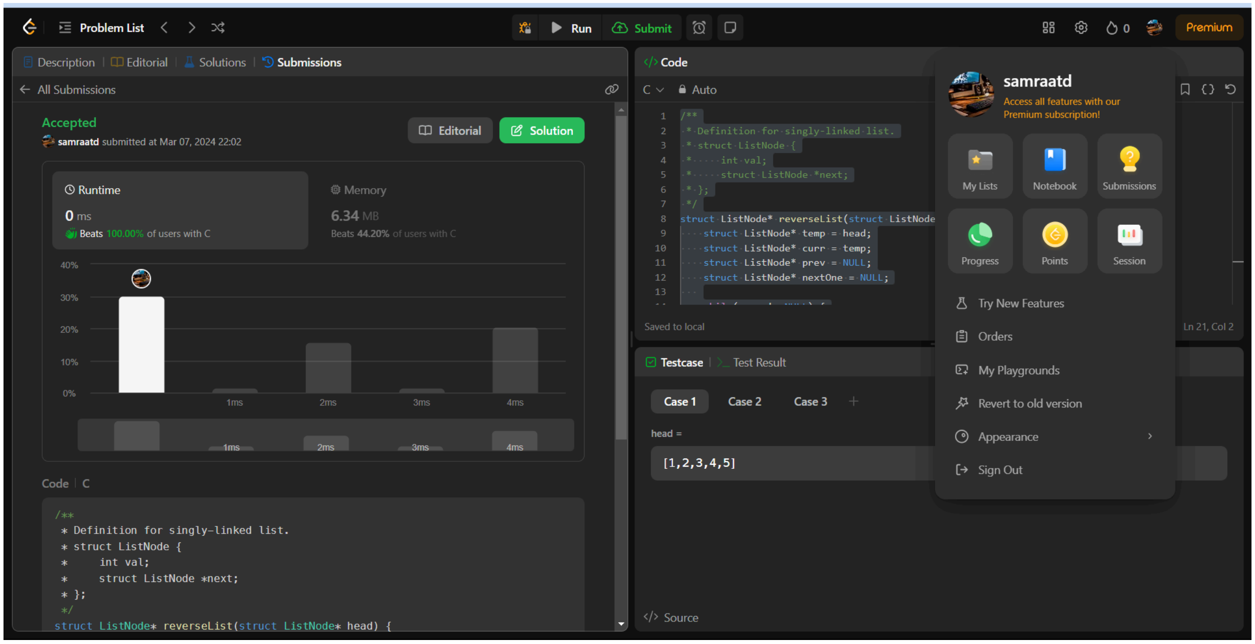


LC 2

Code

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
struct ListNode* reverseList(struct ListNode* head) {
    struct ListNode* temp = head;
    struct ListNode* curr = temp;
    struct ListNode* prev = NULL;
    struct ListNode* nextOne = NULL;

    while(curr != NULL) {
        nextOne = curr->next;
        curr->next = prev;
        prev = curr;
        curr = nextOne;
    }
    return prev;
}
```

Output



Problem List
Run
Submit
Premium
Description
Editorial
Solutions
Submissions
All Submissions
Accepted
samraatd submitted at Mar 07, 2024 22:02
Editorial
Solution
Runtime
0 ms
Beats 100.00% of users with C
Memory
6.34 MB
Beats 44.20% of users with C
Code | C
samraatd
Access all features with our Premium subscription!
My Lists
Notebook
Submissions
Progress
Points
Session
Try New Features
Orders
My Playgrounds
Revert to old version
Appearance
Sign Out
Testcase
Test Result
Case 1
Case 2
Case 3
head =
[1,2,3,4,5]
Saved to local
Ln 21, Col 2
Source

LC 3

Code

```
#include <assert.h>
#include <limits.h>
#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char* readline();

typedef struct SinglyLinkedListNode SinglyLinkedListNode;
typedef struct SinglyLinkedList SinglyLinkedList;

struct SinglyLinkedListNode {
   int data;
   SinglyLinkedListNode* next;
};

struct SinglyLinkedList {
   SinglyLinkedListNode* head;
   SinglyLinkedListNode* tail;
};
```

```
SinglyLinkedListNode* create_singly_linked_list_node(int node_data) {
    SinglyLinkedListNode* node = malloc(sizeof(SinglyLinkedListNode));

    node->data = node_data;
    node->next = NULL;

    return node;
}

void insert_node_into_singly_linked_list(SinglyLinkedList** singly_linked_list, int
node_data) {
    SinglyLinkedListNode* node = create_singly_linked_list_node(node_data);

    if (!(*singly_linked_list)->head) {
        (*singly_linked_list)->head = node;
    } else {
        (*singly_linked_list)->tail->next = node;
    }

    (*singly_linked_list)->tail = node;
}

void print_singly_linked_list(SinglyLinkedListNode* node, char* sep, FILE* fptr) {
    while (node) {
```

```
        fprintf(fptr, "%d", node->data);

        node = node->next;

        if (node) {
            fprintf(fptr, "%s", sep);
        }
    }
}

void free_singly_linked_list(SinglyLinkedListNode* node) {
    while (node) {
        SinglyLinkedListNode* temp = node;
        node = node->next;

        free(temp);
    }
}

// Complete the compare_lists function below.

/*
 * For your reference:
 *
 * SinglyLinkedListNode {
```

```
*    int data;
*    SinglyLinkedListNode* next;
* };
*
*/
bool compare_lists(SinglyLinkedListNode* head1, SinglyLinkedListNode* head2) {
    struct SinglyLinkedListNode *t1;
    struct SinglyLinkedListNode *t2;
    t1=head1;t2 = head2;
    if(t1==NULL && t2==NULL)
        return 1;
    if(t1 != NULL && t2 == NULL)
        return 0;
     if(t1 == NULL && t2 != NULL)
        return 0;
    else
    {
        while(t1->next != NULL && t2->next != NULL)
        {
            if(t1->data == t2->data)
            {
                t1 = t1->next;
                t2 = t2->next;
            }
            else return 0;
```

```
        }
        if(t1->next == NULL && t2->next == NULL)
            return 1;
        else return 0;
    }

}

int main()
{
    FILE* fptr = fopen(getenv("OUTPUT_PATH"), "w");

    char* tests_endptr;
    char* tests_str = readline();
    int tests = strtol(tests_str, &tests_endptr, 10);

    if (tests_endptr == tests_str || *tests_endptr != '\0') { exit(EXIT_FAILURE); }

    for (int tests_itr = 0; tests_itr < tests; tests_itr++) {
        SinglyLinkedList* llist1 = malloc(sizeof(SinglyLinkedList));
        llist1->head = NULL;
        llist1->tail = NULL;

        char* llist1_count_endptr;
        char* llist1_count_str = readline();
```

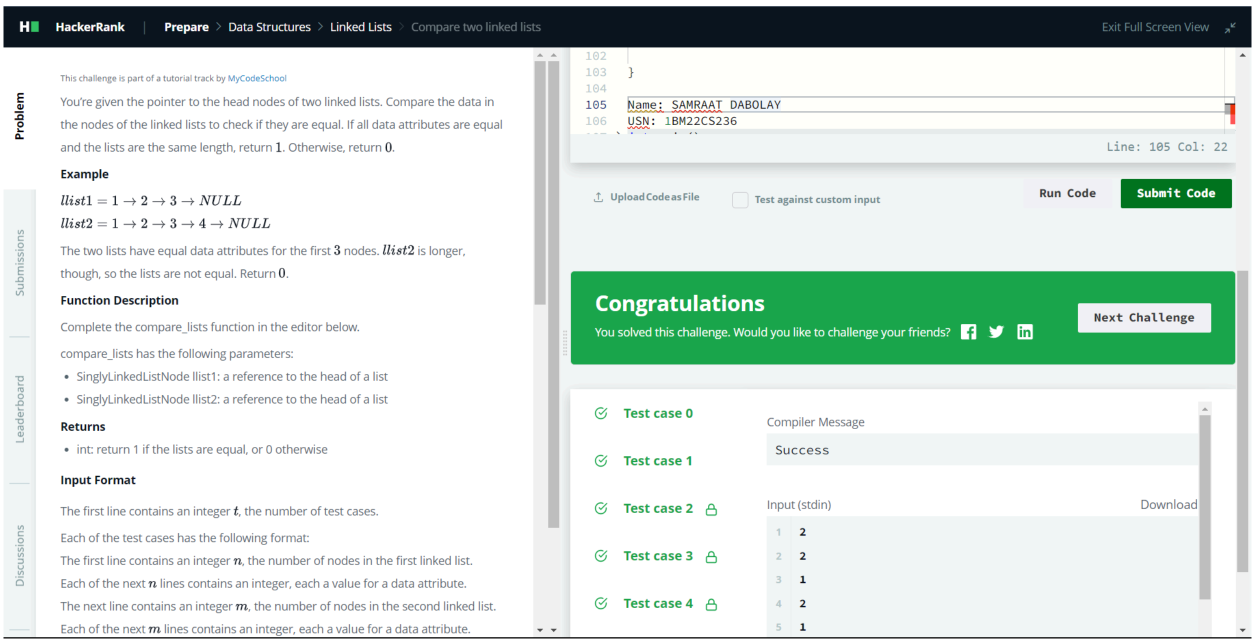
```
    int llist1_count = strtol(llist1_count_str, &llist1_count_endptr, 10);

    if (llist1_count_endptr == llist1_count_str || *llist1_count_endptr != '\0') {
exit(EXIT_FAILURE); }

    for (int i = 0; i < llist1_count; i++) {
        char* llist1_item_endptr;
        char* llist1_item_str = readline();
        int llist1_item = strtol(llist1_item_str, &llist1_item_endptr, 10);

        if (llist1_item_endptr == llist1_item_str || *llist1_item_endptr != '\0') {
exit(EXIT_FAILURE); }

        insert_node_into_singly_linked_list(&llist1, llist1_item);
    }

      SinglyLinkedList* llist2 = malloc(sizeof(SinglyLinkedList));
    llist2->head = NULL;
    llist2->tail = NULL;

    char* llist2_count_endptr;
    char* llist2_count_str = readline();
    int llist2_count = strtol(llist2_count_str, &llist2_count_endptr, 10);
```

```
        if (llist2_count_endptr == llist2_count_str || *llist2_count_endptr != '\0') {
exit(EXIT_FAILURE); }

        for (int i = 0; i < llist2_count; i++) {
            char* llist2_item_endptr;
            char* llist2_item_str = readline();
            int llist2_item = strtol(llist2_item_str, &llist2_item_endptr, 10);

            if (llist2_item_endptr == llist2_item_str || *llist2_item_endptr != '\0') {
exit(EXIT_FAILURE); }

            insert_node_into_singly_linked_list(&llist2, llist2_item);
        }

        bool result = compare_lists(llist1->head, llist2->head);

        fprintf(fptr, "%d\n", result);
    }

    fclose(fptr);

    return 0;
}

char* readline() {
```

```
size_t alloc_length = 1024;
size_t data_length = 0;
char* data = malloc(alloc_length);

while (true) {
   char* cursor = data + data_length;
   char* line = fgets(cursor, alloc_length - data_length, stdin);

   if (!line) { break; }

   data_length += strlen(cursor);

   if (data_length < alloc_length - 1 || data[data_length - 1] == '\n') { break; }

   size_t new_length = alloc_length << 1;
   data = realloc(data, new_length);

   if (!data) { break; }

   alloc_length = new_length;
}

if (data[data_length - 1] == '\n') {
   data[data_length - 1] = '\0';
}
```

```
    data = realloc(data, data_length);


    return data;
}
```

Output



LC 4

Code

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */

void findLeaves(struct TreeNode* node, int** leafValues, int* size, int* capacity) {
    if (node == NULL) {
        return;
    }

    if (node->left == NULL && node->right == NULL) {
        if (*size >= *capacity) {
            *capacity *= 2;
            *leafValues = (int*) realloc(*leafValues, *capacity * sizeof(int));
        }
        (*leafValues)[(*size)++] = node->val;
    }
```

```
    findLeaves(node->left, leafValues, size, capacity);
    findLeaves(node->right, leafValues, size, capacity);
}

bool leafSimilar(struct TreeNode* root1, struct TreeNode* root2) {
    int *leaves1 = (int*) malloc(sizeof(int) * 10);
    int size1 = 0, capacity1 = 10;

    int *leaves2 = (int*) malloc(sizeof(int) * 10);
    int size2 = 0, capacity2 = 10;

    findLeaves(root1, &leaves1, &size1, &capacity1);
    findLeaves(root2, &leaves2, &size2, &capacity2);

    if (size1 != size2) {
        free(leaves1);
        free(leaves2);
        return false;
    }

    for (int i = 0; i < size1; i++) {
        if (leaves1[i] != leaves2[i]) {
            free(leaves1);
            free(leaves2);
            return false;
```

```
        }
    }

    free(leaves1);
    free(leaves2);
    return true;
}
```

Output