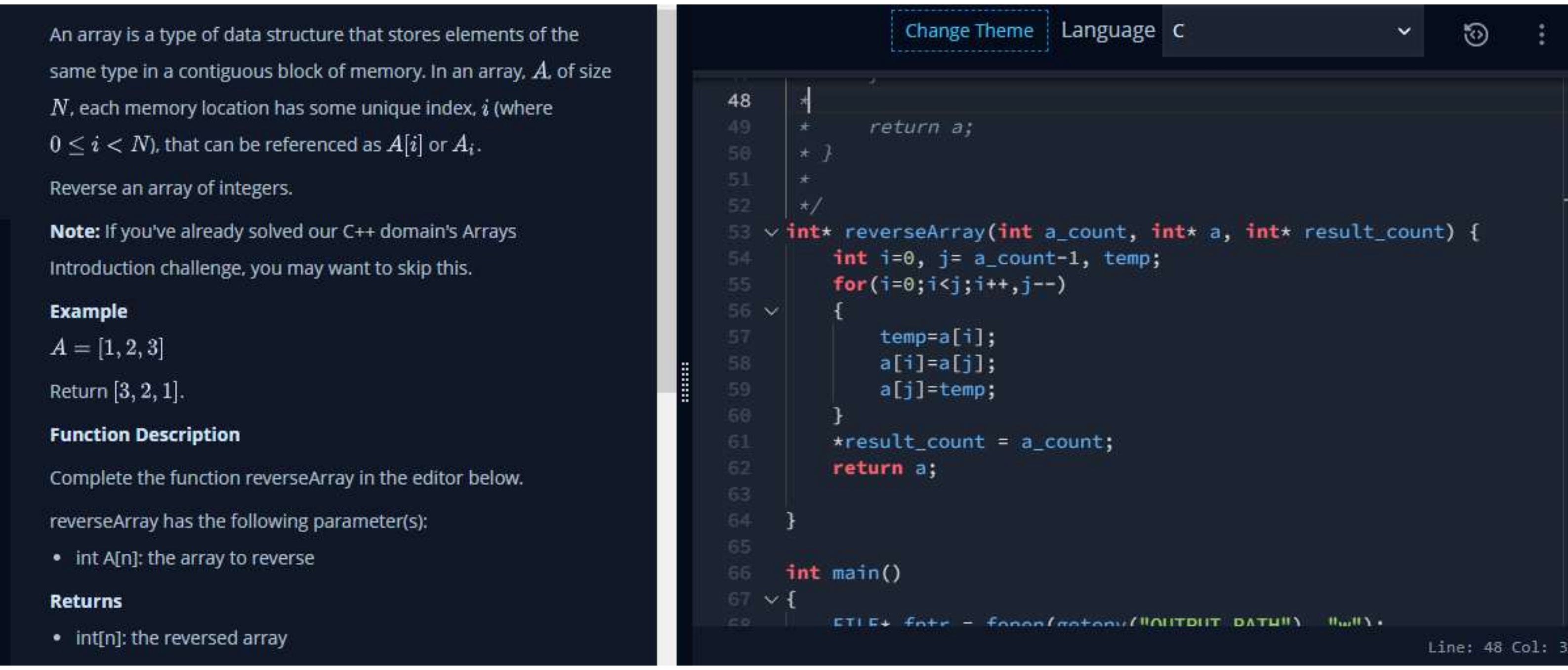


An array is a type of data structure that stores elements of the same type in a contiguous block of memory. In an array, A, of size N, each memory location has some unique index, i (where $0 \leq i < N$), that can be referenced as $A[i]$ or A_i.

Reverse an array of integers.

Note: If you've already solved our C++ domain's Arrays Introduction challenge, you may want to skip this.

Example

$A = [1, 2, 3]$

Return $[3, 2, 1]$.

Function Description

Complete the function reverseArray in the editor below.

reverseArray has the following parameter(s):

- int A[n]: the array to reverse

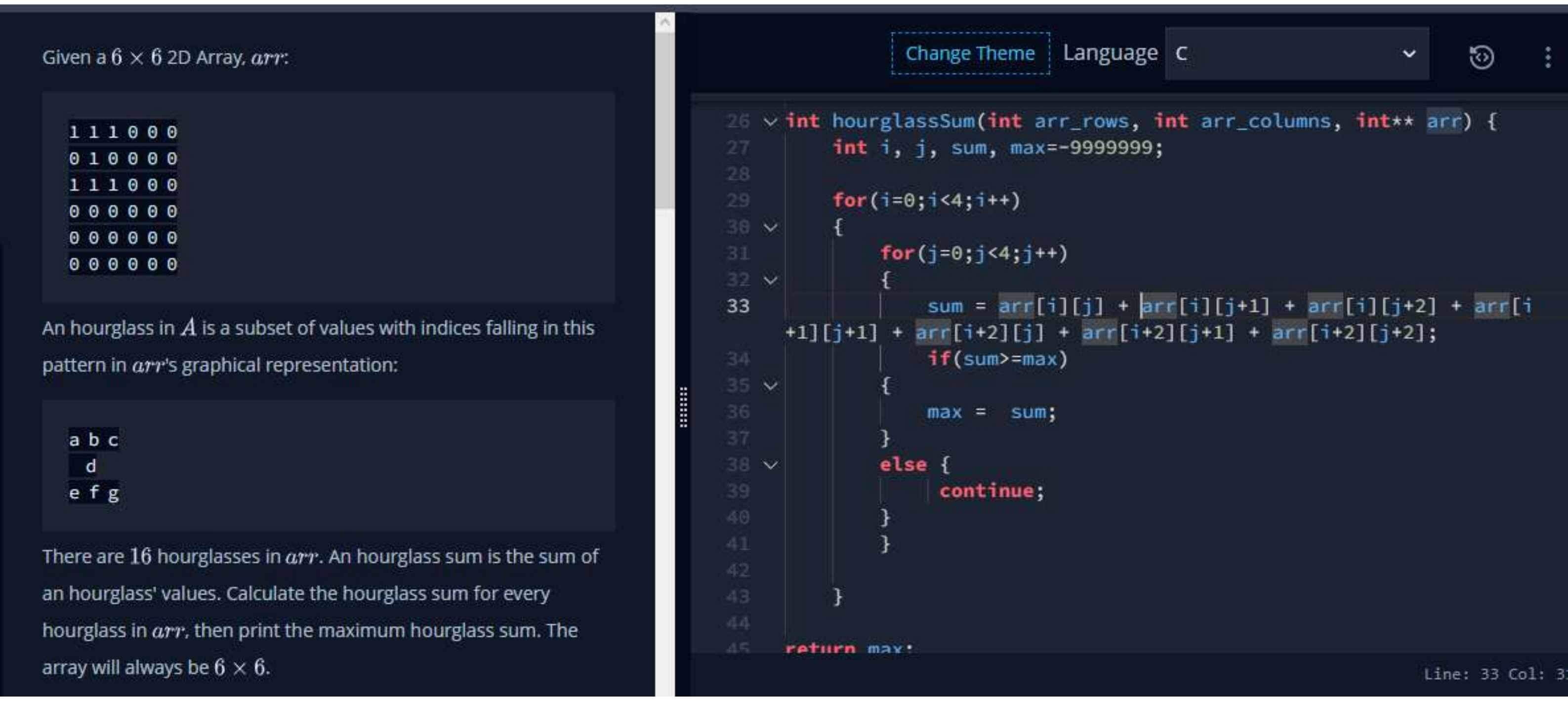
Returns

- int[n]: the reversed array

Language C

```
 *
 *     return a;
 * }
 *
 */
int* reverseArray(int a_count, int* a, int* result_count) {
    int i=0, j= a_count-1, temp;
    for(i=0;i<j;i++,j--)
    {
        temp=a[i];
        a[i]=a[j];
        a[j]=temp;
    }
    *result_count = a_count;
    return a;

}

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

Line: 48 Col: 3



Given a 6×6 2D Array, arr:

```
1 1 1 0 0 0
0 1 0 0 0 0
1 1 1 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
```

An hourglass in A is a subset of values with indices falling in this pattern in arr's graphical representation:

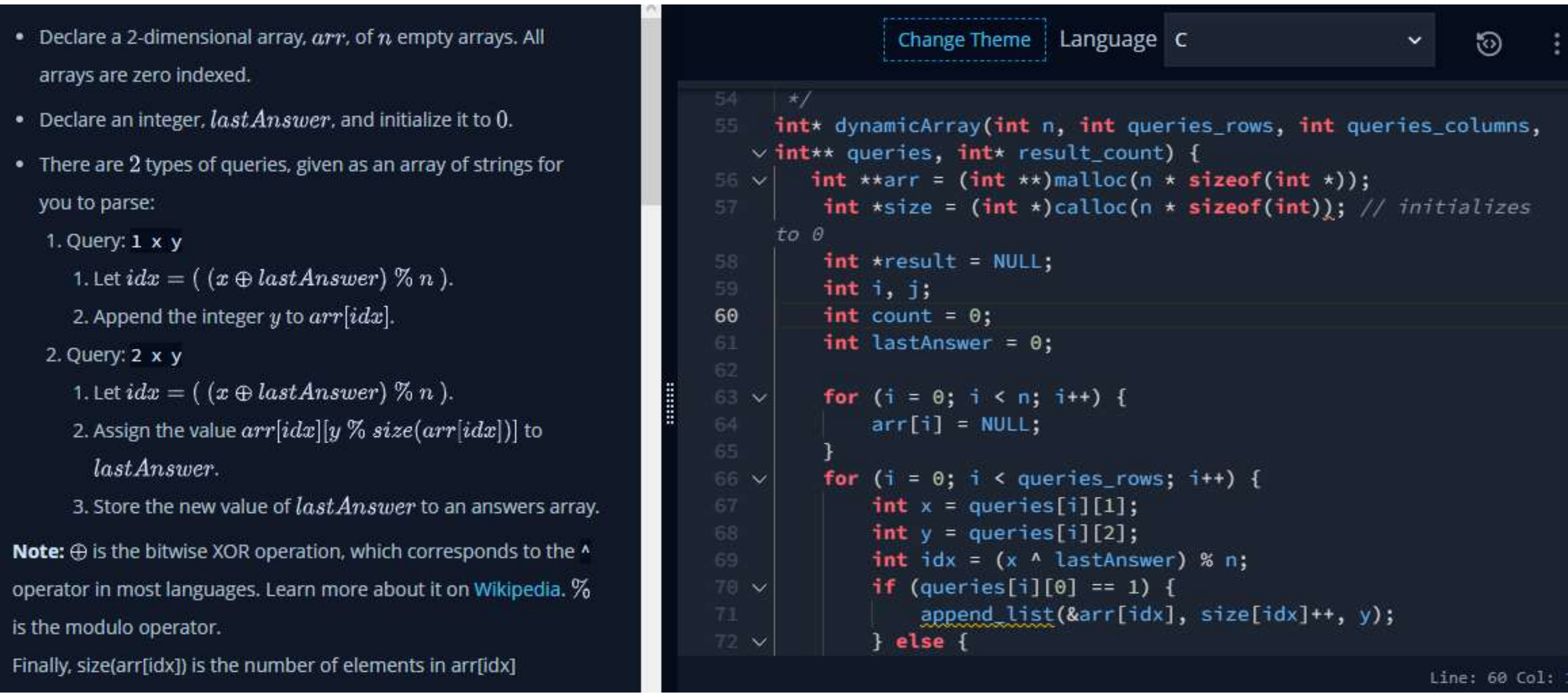
```
a b c
  d
e f g
```

There are 16 hourglasses in arr. An hourglass sum is the sum of an hourglass' values. Calculate the hourglass sum for every hourglass in arr, then print the maximum hourglass sum. The array will always be 6×6.

Language C

```
int hourglassSum(int arr_rows, int arr_columns, int** arr) {
    int i, j, sum, max=-9999999;

    for(i=0;i<4;i++)
    {
        for(j=0;j<4;j++)
        {
            sum = arr[i][j] + arr[i][j+1] + arr[i][j+2] + arr[i+1][j+1] + arr[i+2][j] + arr[i+2][j+1] + arr[i+2][j+2];
            if(sum>=max)
            {
            max =  sum;
            }
            else {
                continue;
            }
            }

    }

return max;
```

Line: 33 Col: 3

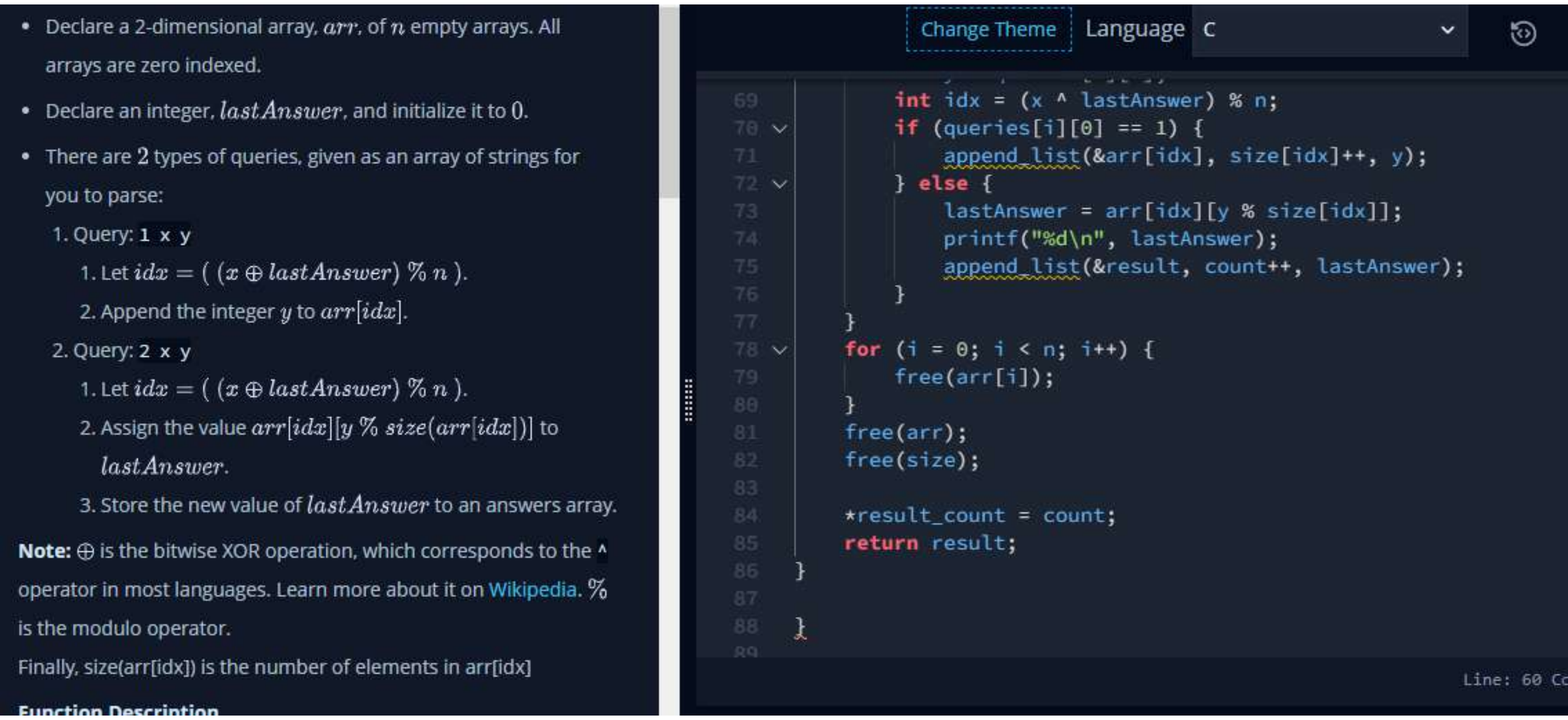


- Declare a 2-dimensional array, arr, of n empty arrays. All arrays are zero indexed.
- Declare an integer, $lastAnswer$, and initialize it to 0.
- There are 2 types of queries, given as an array of strings for you to parse:
  1. Query: `1 x y`
     1. Let $idx = (\,(x \oplus lastAnswer)\ \%\ n\,)$.
     2. Append the integer y to $arr[idx]$.
  2. Query: `2 x y`
     1. Let $idx = (\,(x \oplus lastAnswer)\ \%\ n\,)$.
     2. Assign the value $arr[idx][y\ \%\ size(arr[idx])]$ to $lastAnswer$.
     3. Store the new value of $lastAnswer$ to an answers array.

Note: $\oplus$ is the bitwise XOR operation, which corresponds to the `^` operator in most languages. Learn more about it on Wikipedia. $\%$ is the modulo operator.

Finally, size(arr[idx]) is the number of elements in arr[idx]

```c
 */
int* dynamicArray(int n, int queries_rows, int queries_columns, int** queries, int* result_count) {
    int **arr = (int **)malloc(n * sizeof(int *));
    int *size = (int *)calloc(n * sizeof(int)); // initializes to 0
    int *result = NULL;
    int i, j;
    int count = 0;
    int lastAnswer = 0;

    for (i = 0; i < n; i++) {
        arr[i] = NULL;
    }
    for (i = 0; i < queries_rows; i++) {
        int x = queries[i][1];
        int y = queries[i][2];
        int idx = (x ^ lastAnswer) % n;
        if (queries[i][0] == 1) {
            append_list(&arr[idx], size[idx]++, y);
        } else {
```



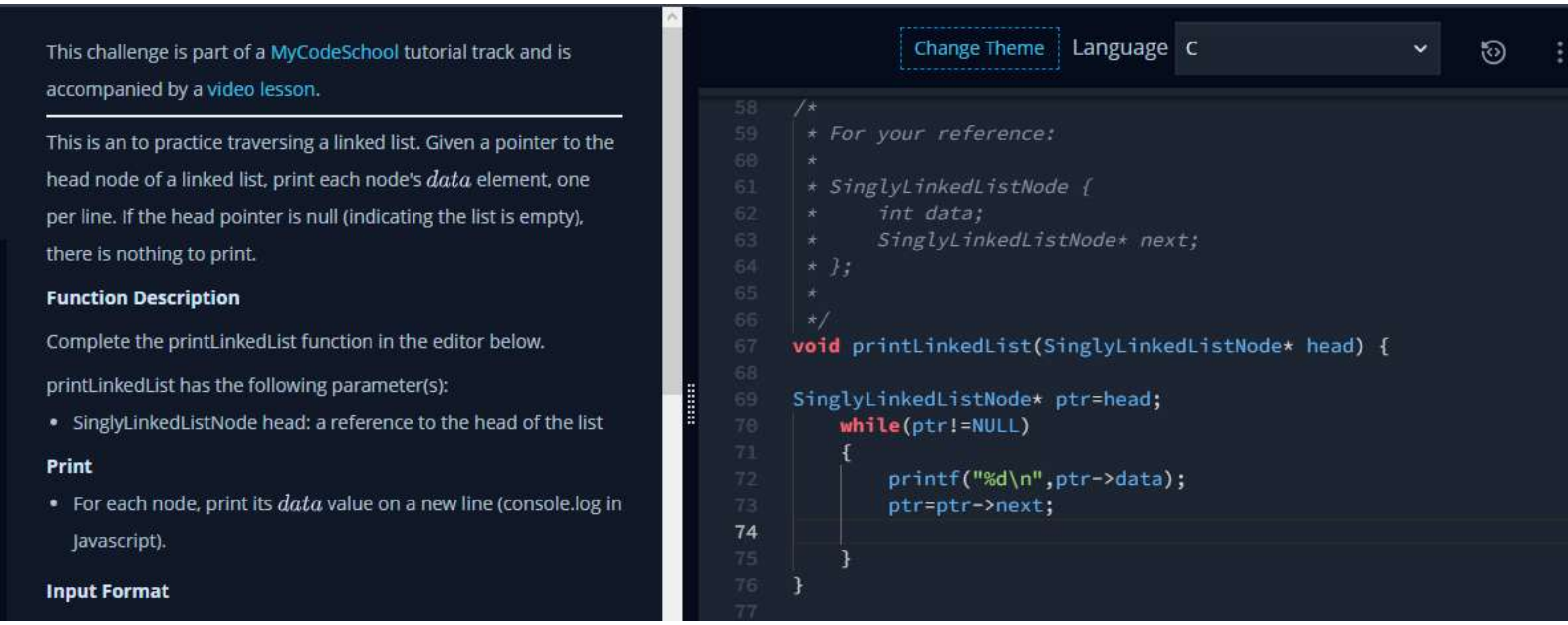
- Declare a 2-dimensional array, arr, of n empty arrays. All arrays are zero indexed.
- Declare an integer, $lastAnswer$, and initialize it to 0.
- There are 2 types of queries, given as an array of strings for you to parse:
  1. Query: `1 x y`
     1. Let $idx = (\ (x \oplus lastAnswer)\ \%\ n\ )$.
     2. Append the integer y to $arr[idx]$.
  2. Query: `2 x y`
     1. Let $idx = (\ (x \oplus lastAnswer)\ \%\ n\ )$.
     2. Assign the value $arr[idx][y\ \%\ size(arr[idx])]$ to $lastAnswer$.
     3. Store the new value of $lastAnswer$ to an answers array.

Note: $\oplus$ is the bitwise XOR operation, which corresponds to the ^ operator in most languages. Learn more about it on Wikipedia. $\%$ is the modulo operator.

Finally, size(arr[idx]) is the number of elements in arr[idx]

Function Description

```
        int idx = (x ^ lastAnswer) % n;
        if (queries[i][0] == 1) {
            append_list(&arr[idx], size[idx]++, y);
        } else {
            lastAnswer = arr[idx][y % size[idx]];
            printf("%d\n", lastAnswer);
            append_list(&result, count++, lastAnswer);
        }
    }
    for (i = 0; i < n; i++) {
        free(arr[i]);
    }
    free(arr);
    free(size);

    *result_count = count;
    return result;
}

}
```



This challenge is part of a MyCodeSchool tutorial track and is accompanied by a video lesson.

This is an to practice traversing a linked list. Given a pointer to the head node of a linked list, print each node's $data$ element, one per line. If the head pointer is null (indicating the list is empty), there is nothing to print.

Function Description

Complete the printLinkedList function in the editor below.

printLinkedList has the following parameter(s):

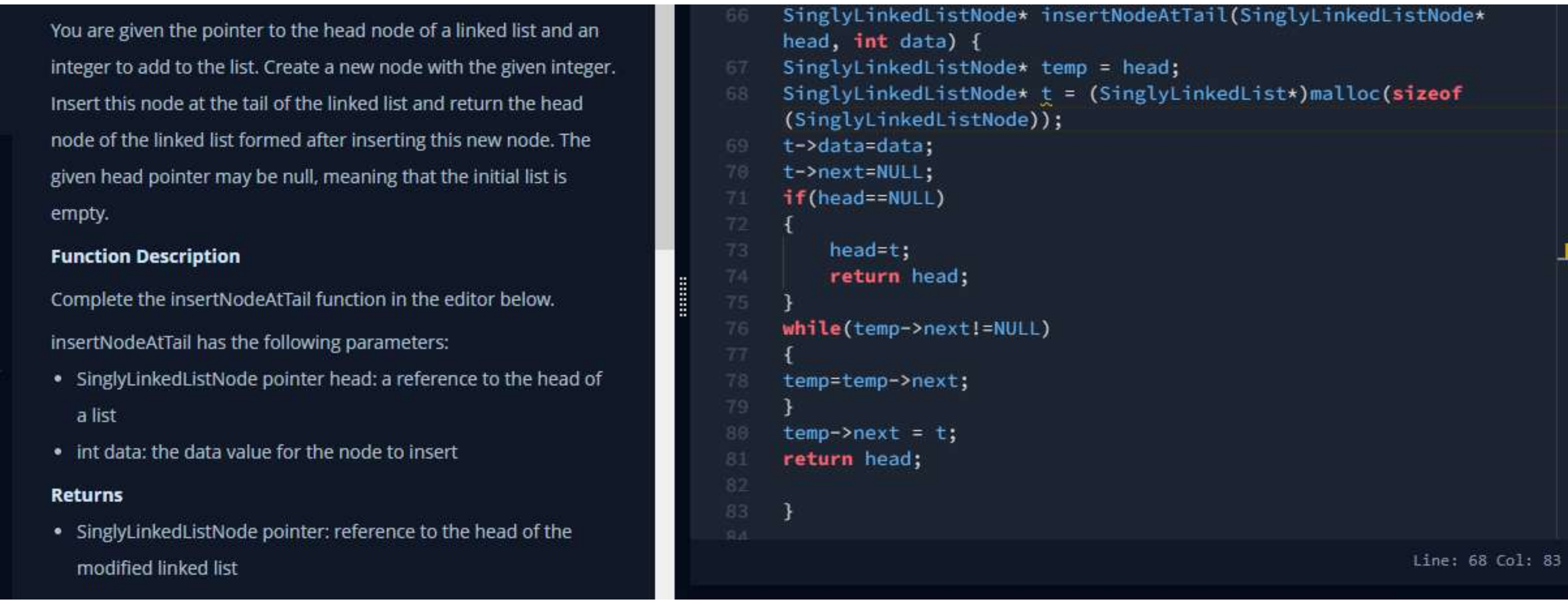
- SinglyLinkedListNode head: a reference to the head of the list

Print

- For each node, print its $data$ value on a new line (console.log in Javascript).

Input Format

```
/*
 * For your reference:
 *
 * SinglyLinkedListNode {
 *     int data;
 *     SinglyLinkedListNode* next;
 * };
 *
 */
void printLinkedList(SinglyLinkedListNode* head) {

SinglyLinkedListNode* ptr=head;
    while(ptr!=NULL)
    {
        printf("%d\n",ptr->data);
        ptr=ptr->next;

    }
}

```



You are given the pointer to the head node of a linked list and an integer to add to the list. Create a new node with the given integer. Insert this node at the tail of the linked list and return the head node of the linked list formed after inserting this new node. The given head pointer may be null, meaning that the initial list is empty.

Function Description

Complete the insertNodeAtTail function in the editor below.

insertNodeAtTail has the following parameters:

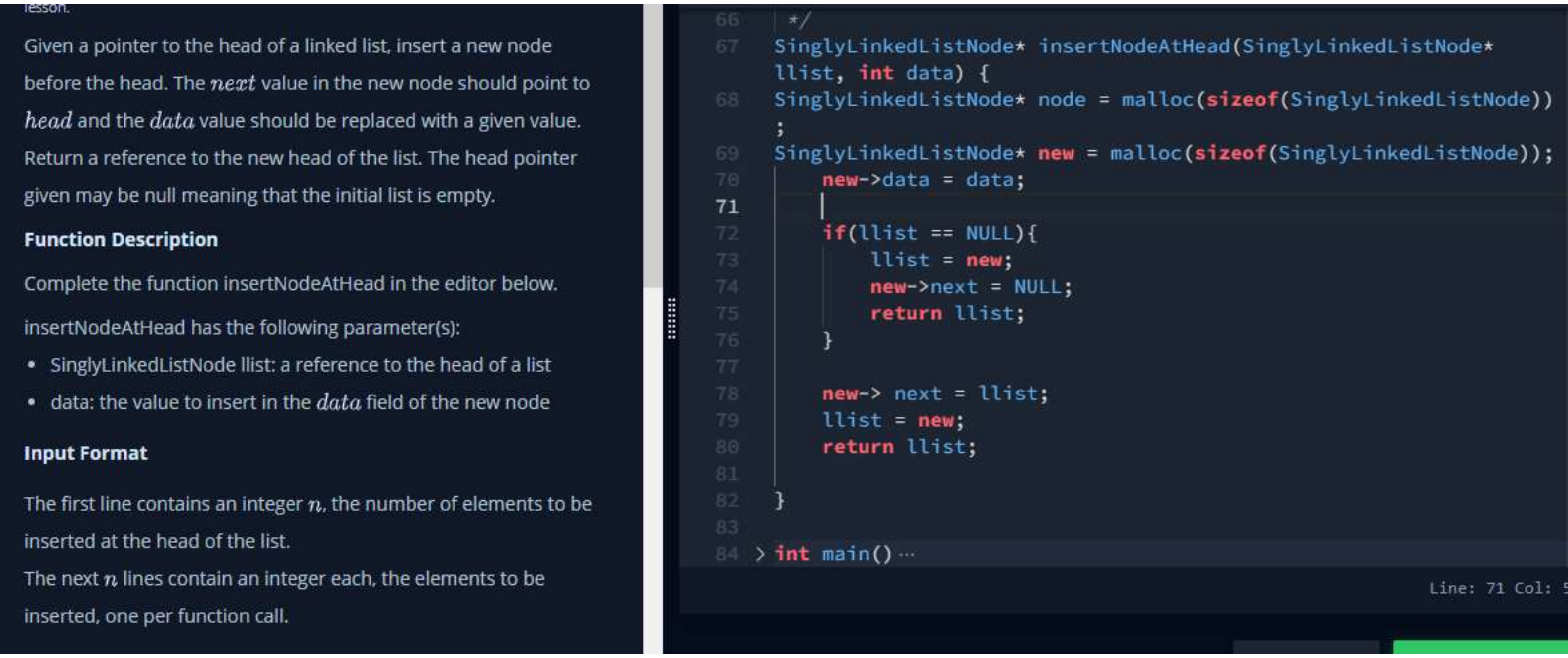
- SinglyLinkedListNode pointer head: a reference to the head of a list
- int data: the data value for the node to insert

Returns

- SinglyLinkedListNode pointer: reference to the head of the modified linked list

```
SinglyLinkedListNode* insertNodeAtTail(SinglyLinkedListNode* head, int data) {
SinglyLinkedListNode* temp = head;
SinglyLinkedListNode* t = (SinglyLinkedList*)malloc(sizeof(SinglyLinkedListNode));
t->data=data;
t->next=NULL;
if(head==NULL)
{
    head=t;
    return head;
}
while(temp->next!=NULL)
{
temp=temp->next;
}
temp->next = t;
return head;

}
```

Line: 68 Col: 83



lesson.

Given a pointer to the head of a linked list, insert a new node before the head. The *next* value in the new node should point to *head* and the *data* value should be replaced with a given value. Return a reference to the new head of the list. The head pointer given may be null meaning that the initial list is empty.

Function Description

Complete the function insertNodeAtHead in the editor below.

insertNodeAtHead has the following parameter(s):

- SinglyLinkedListNode llist: a reference to the head of a list
- data: the value to insert in the *data* field of the new node

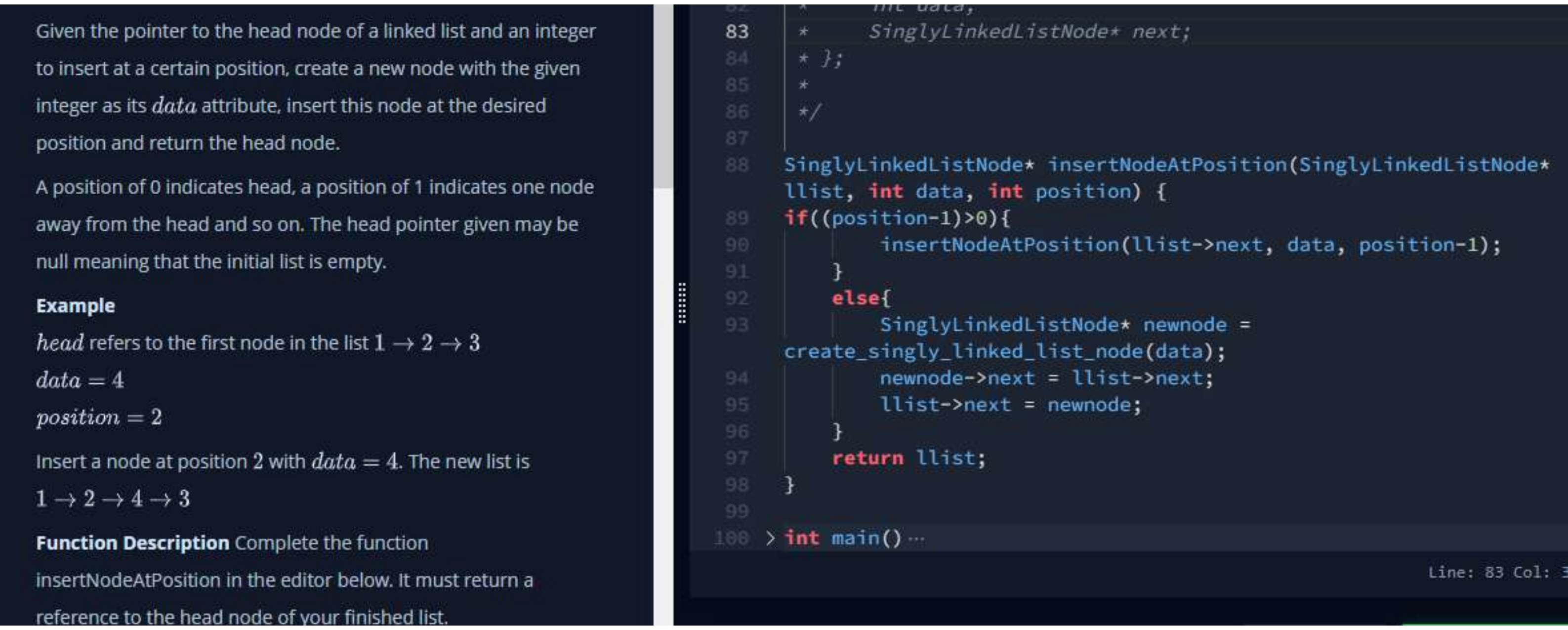
Input Format

The first line contains an integer n, the number of elements to be inserted at the head of the list.

The next n lines contain an integer each, the elements to be inserted, one per function call.

```
 */
SinglyLinkedListNode* insertNodeAtHead(SinglyLinkedListNode*
llist, int data) {
SinglyLinkedListNode* node = malloc(sizeof(SinglyLinkedListNode))
;
SinglyLinkedListNode* new = malloc(sizeof(SinglyLinkedListNode));
    new->data = data;
    
    if(llist == NULL){
        llist = new;
        new->next = NULL;
        return llist;
    }
    
    new-> next = llist;
    llist = new;
    return llist;
    
}

> int main() ···
```

Line: 71 Col: 5



Given the pointer to the head node of a linked list and an integer to insert at a certain position, create a new node with the given integer as its $data$ attribute, insert this node at the desired position and return the head node.

A position of 0 indicates head, a position of 1 indicates one node away from the head and so on. The head pointer given may be null meaning that the initial list is empty.

Example

$head$ refers to the first node in the list $1 \rightarrow 2 \rightarrow 3$

$data = 4$

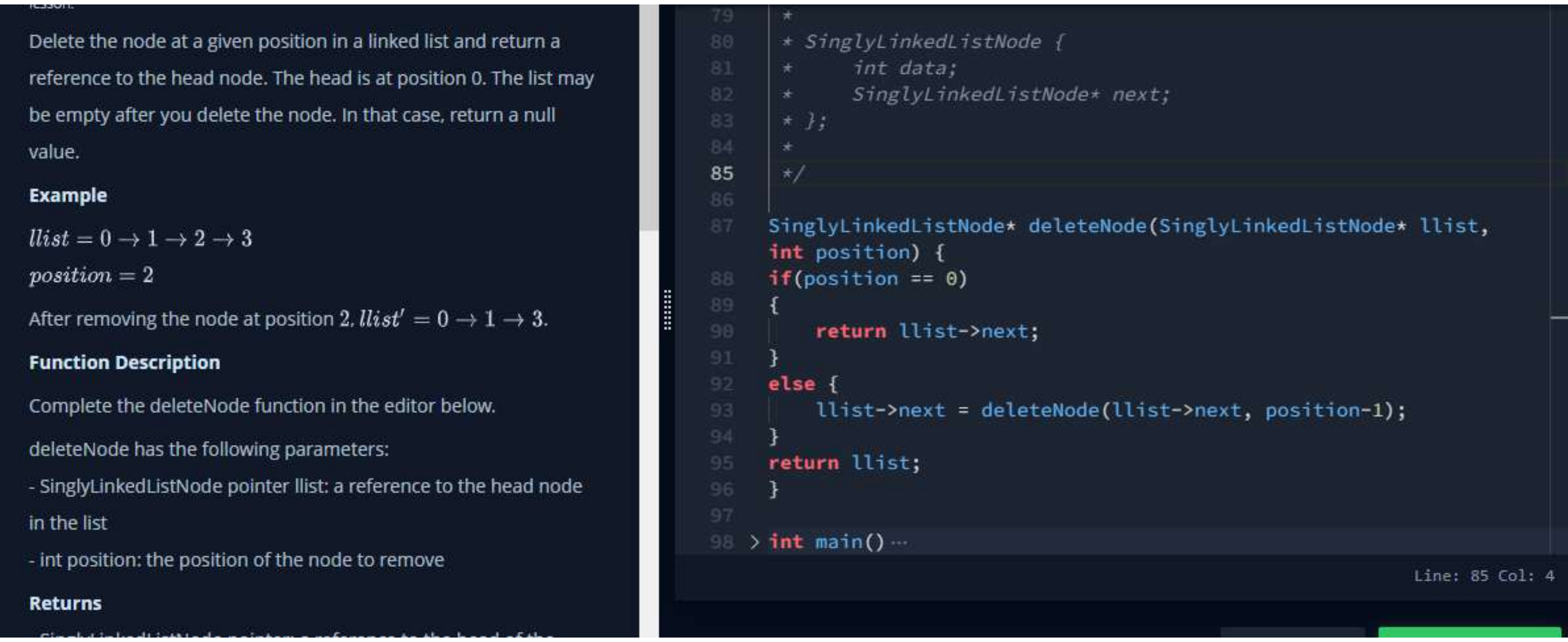
$position = 2$

Insert a node at position 2 with $data = 4$. The new list is $1 \rightarrow 2 \rightarrow 4 \rightarrow 3$

Function Description Complete the function insertNodeAtPosition in the editor below. It must return a reference to the head node of your finished list.

```
 *     int data;
 *     SinglyLinkedListNode* next;
 * };
 *
 */

SinglyLinkedListNode* insertNodeAtPosition(SinglyLinkedListNode* llist, int data, int position) {
if((position-1)>0){
        insertNodeAtPosition(llist->next, data, position-1);
    }
    else{
        SinglyLinkedListNode* newnode = create_singly_linked_list_node(data);
        newnode->next = llist->next;
        llist->next = newnode;
    }
    return llist;
}

> int main() ···
```

Line: 83 Col: 3



lesson.

Delete the node at a given position in a linked list and return a reference to the head node. The head is at position 0. The list may be empty after you delete the node. In that case, return a null value.

Example

$llist = 0 \to 1 \to 2 \to 3$

$position = 2$

After removing the node at position 2, $llist' = 0 \to 1 \to 3$.

Function Description

Complete the deleteNode function in the editor below.

deleteNode has the following parameters:

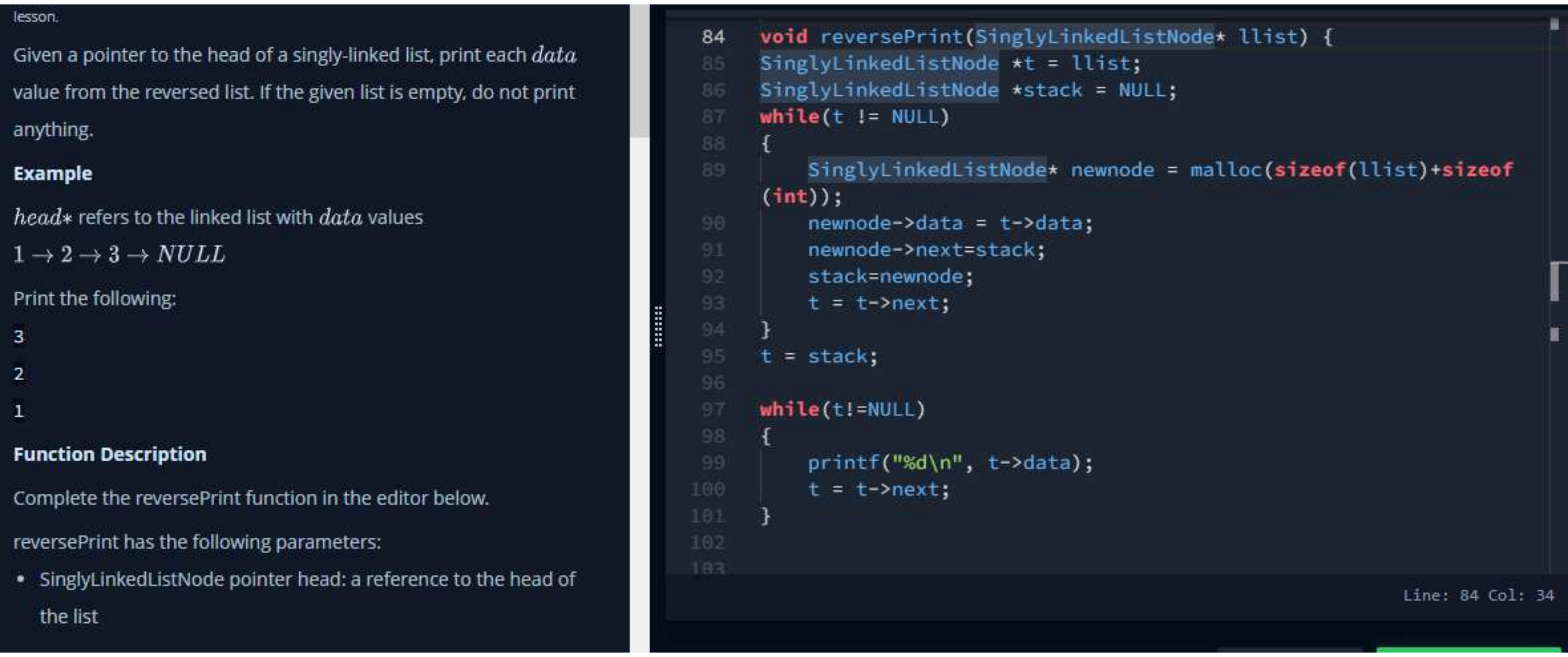
- SinglyLinkedListNode pointer llist: a reference to the head node in the list

- int position: the position of the node to remove

Returns

- SinglyLinkedListNode pointer: a reference to the head of the

```
 *
 * SinglyLinkedListNode {
 *     int data;
 *     SinglyLinkedListNode* next;
 * };
 *
 */

SinglyLinkedListNode* deleteNode(SinglyLinkedListNode* llist, int position) {
if(position == 0)
{
    return llist->next;
}
else {
    llist->next = deleteNode(llist->next, position-1);
}
return llist;
}

> int main() ···
```



lesson.

Given a pointer to the head of a singly-linked list, print each $data$ value from the reversed list. If the given list is empty, do not print anything.

Example

$head*$ refers to the linked list with $data$ values

$1 \rightarrow 2 \rightarrow 3 \rightarrow NULL$

Print the following:

```
3
2
1
```

Function Description

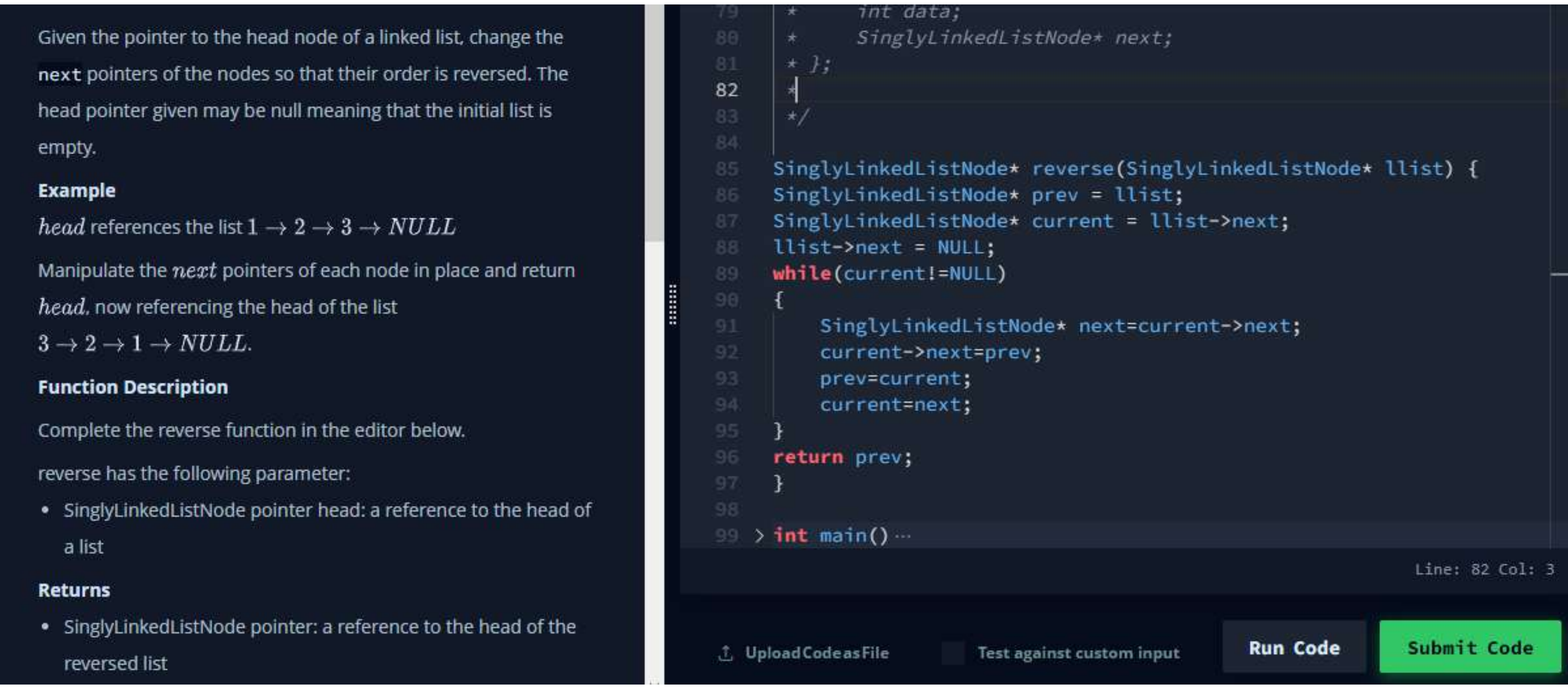
Complete the reversePrint function in the editor below.

reversePrint has the following parameters:

- SinglyLinkedListNode pointer head: a reference to the head of the list

```c
void reversePrint(SinglyLinkedListNode* llist) {
SinglyLinkedListNode *t = llist;
SinglyLinkedListNode *stack = NULL;
while(t != NULL)
{
    SinglyLinkedListNode* newnode = malloc(sizeof(llist)+sizeof(int));
    newnode->data = t->data;
    newnode->next=stack;
    stack=newnode;
    t = t->next;
}
t = stack;

while(t!=NULL)
{
    printf("%d\n", t->data);
    t = t->next;
}
```

Line: 84 Col: 34



Given the pointer to the head node of a linked list, change the `next` pointers of the nodes so that their order is reversed. The head pointer given may be null meaning that the initial list is empty.

Example

$head$ references the list $1 \rightarrow 2 \rightarrow 3 \rightarrow NULL$

Manipulate the $next$ pointers of each node in place and return $head$, now referencing the head of the list $3 \rightarrow 2 \rightarrow 1 \rightarrow NULL$.

Function Description

Complete the reverse function in the editor below.

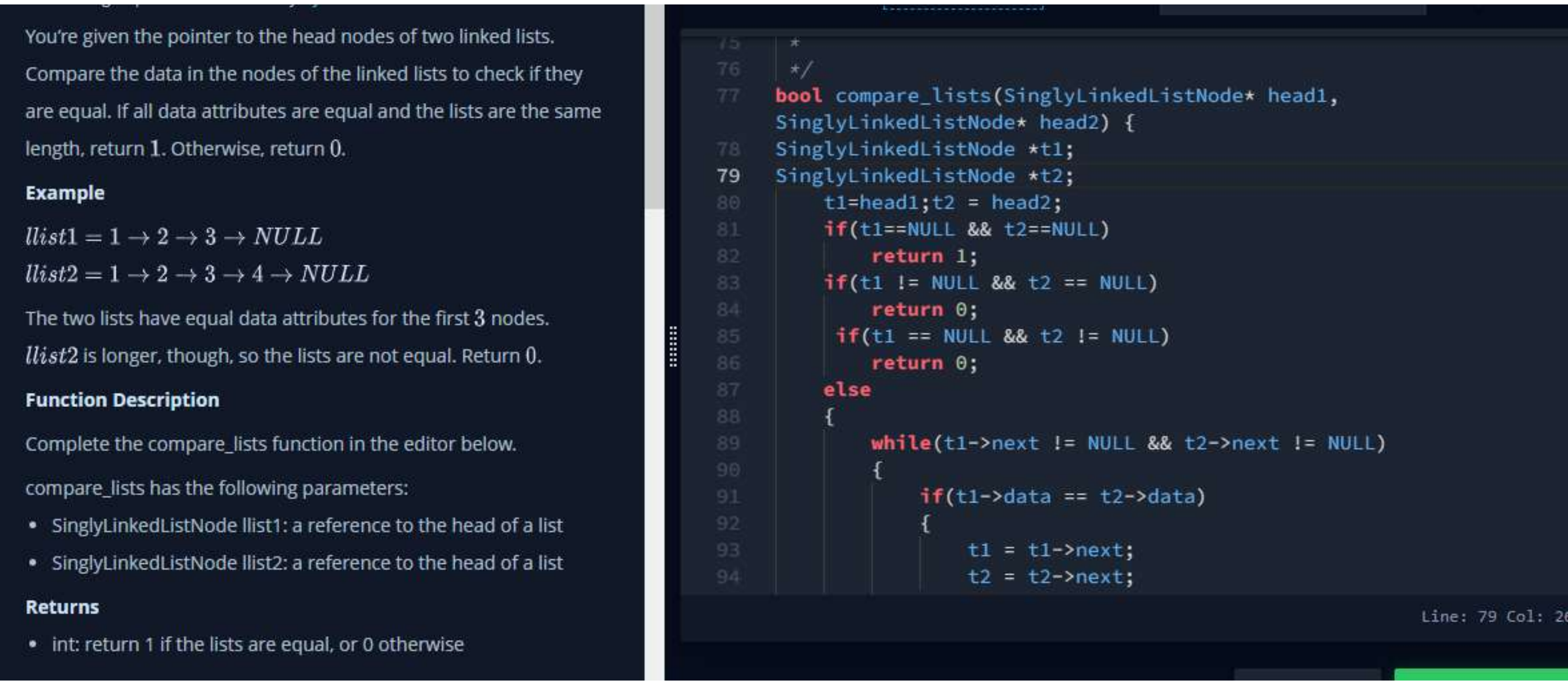
reverse has the following parameter:

- SinglyLinkedListNode pointer head: a reference to the head of a list

Returns

- SinglyLinkedListNode pointer: a reference to the head of the reversed list

```
 *     int data;
 *     SinglyLinkedListNode* next;
 * };
 *
 */

SinglyLinkedListNode* reverse(SinglyLinkedListNode* llist) {
SinglyLinkedListNode* prev = llist;
SinglyLinkedListNode* current = llist->next;
llist->next = NULL;
while(current!=NULL)
{
    SinglyLinkedListNode* next=current->next;
    current->next=prev;
    prev=current;
    current=next;
}
return prev;
}

> int main() ···
```



You're given the pointer to the head nodes of two linked lists. Compare the data in the nodes of the linked lists to check if they are equal. If all data attributes are equal and the lists are the same length, return 1. Otherwise, return 0.

Example

$llist1 = 1 \rightarrow 2 \rightarrow 3 \rightarrow NULL$

$llist2 = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow NULL$

The two lists have equal data attributes for the first 3 nodes. $llist2$ is longer, though, so the lists are not equal. Return 0.

Function Description

Complete the compare_lists function in the editor below.

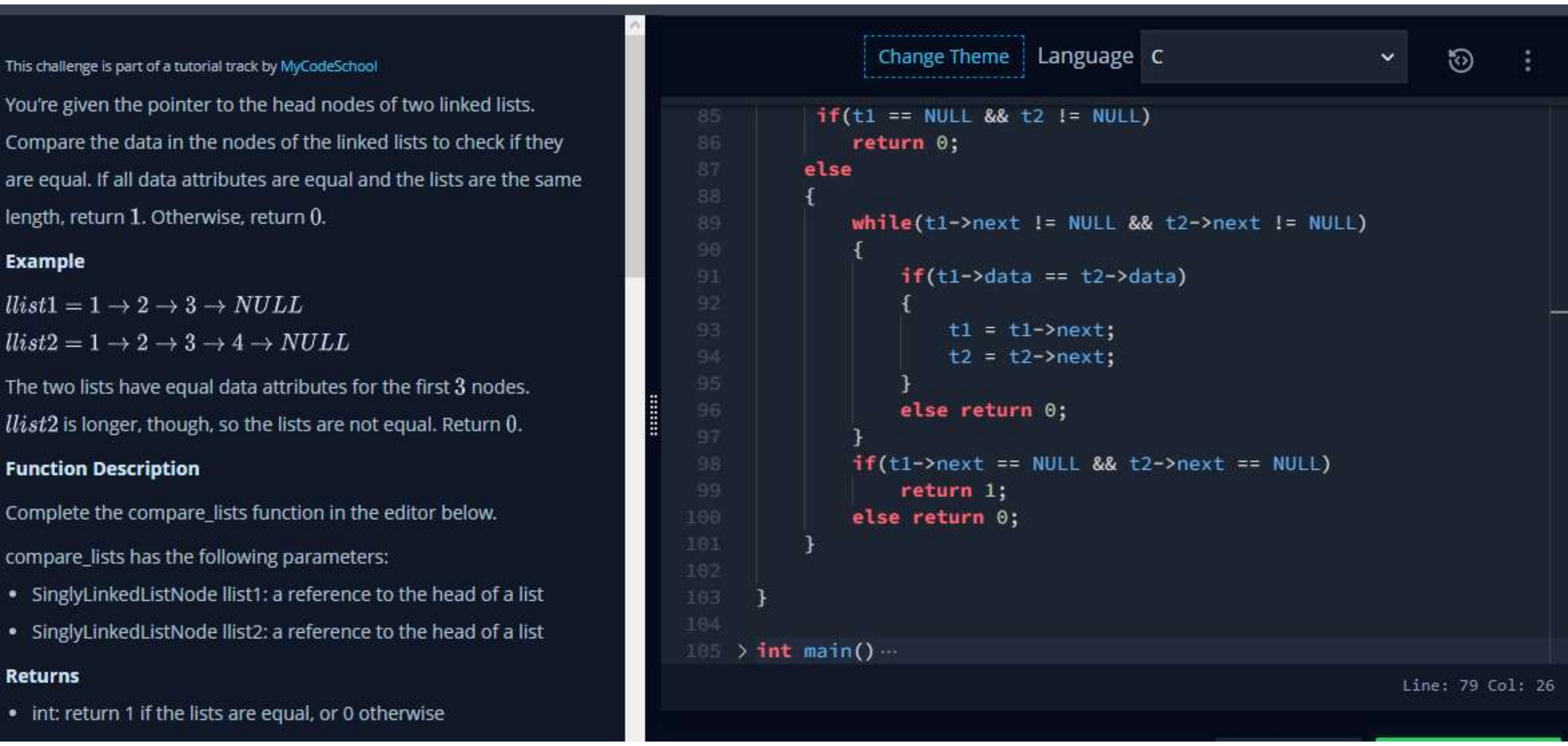
compare_lists has the following parameters:

- SinglyLinkedListNode llist1: a reference to the head of a list
- SinglyLinkedListNode llist2: a reference to the head of a list

Returns

- int: return 1 if the lists are equal, or 0 otherwise

```
 *
 */
bool compare_lists(SinglyLinkedListNode* head1,
SinglyLinkedListNode* head2) {
SinglyLinkedListNode *t1;
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;
```



This challenge is part of a tutorial track by MyCodeSchool

You're given the pointer to the head nodes of two linked lists. Compare the data in the nodes of the linked lists to check if they are equal. If all data attributes are equal and the lists are the same length, return 1. Otherwise, return 0.

Example

$llist1 = 1 \rightarrow 2 \rightarrow 3 \rightarrow NULL$

$llist2 = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow NULL$

The two lists have equal data attributes for the first 3 nodes. $llist2$ is longer, though, so the lists are not equal. Return 0.

Function Description

Complete the compare_lists function in the editor below.

compare_lists has the following parameters:

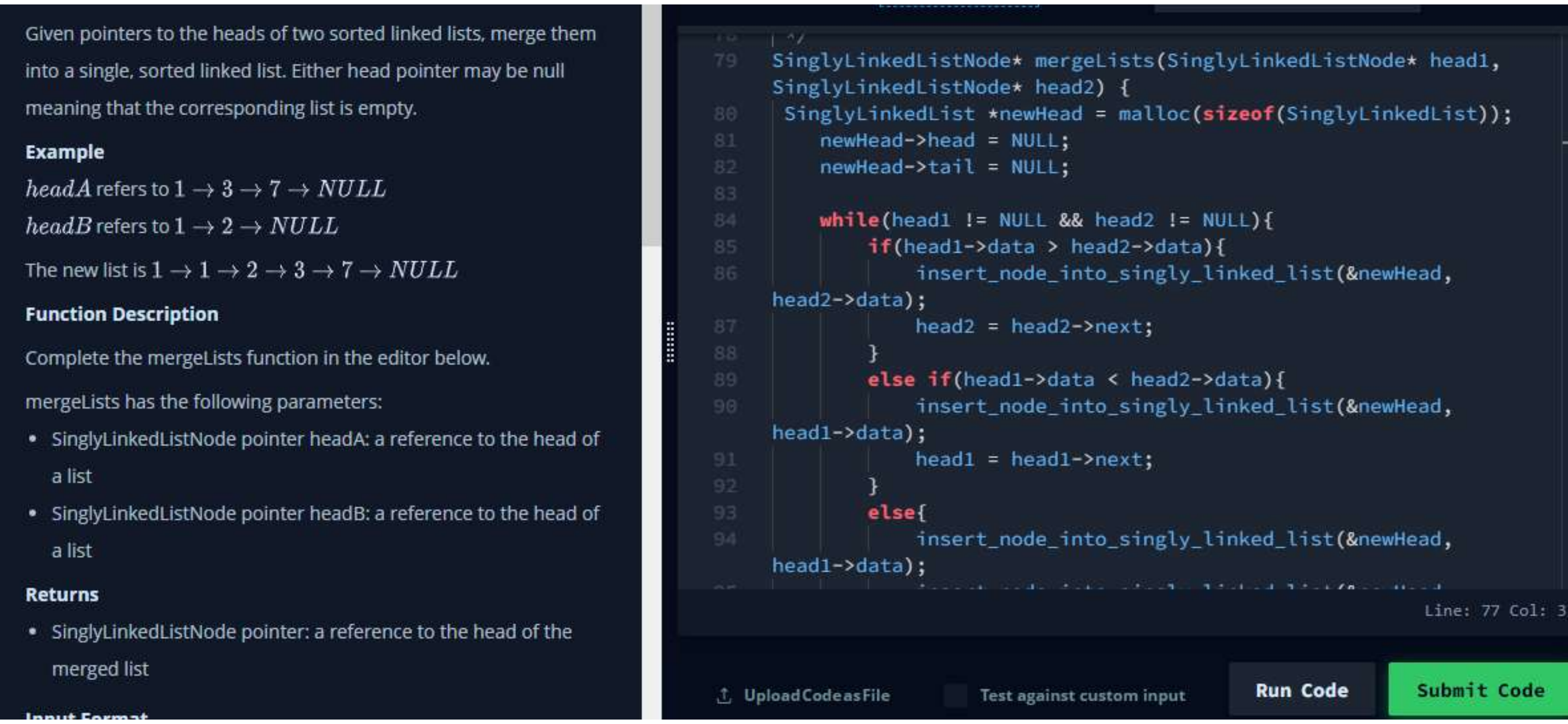
- SinglyLinkedListNode llist1: a reference to the head of a list
- SinglyLinkedListNode llist2: a reference to the head of a list

Returns

- int: return 1 if the lists are equal, or 0 otherwise

```
    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() ···
```

Line: 79 Col: 26



Given pointers to the heads of two sorted linked lists, merge them into a single, sorted linked list. Either head pointer may be null meaning that the corresponding list is empty.

Example

$headA$ refers to $1 \rightarrow 3 \rightarrow 7 \rightarrow NULL$

$headB$ refers to $1 \rightarrow 2 \rightarrow NULL$

The new list is $1 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 7 \rightarrow NULL$

Function Description

Complete the mergeLists function in the editor below.

mergeLists has the following parameters:

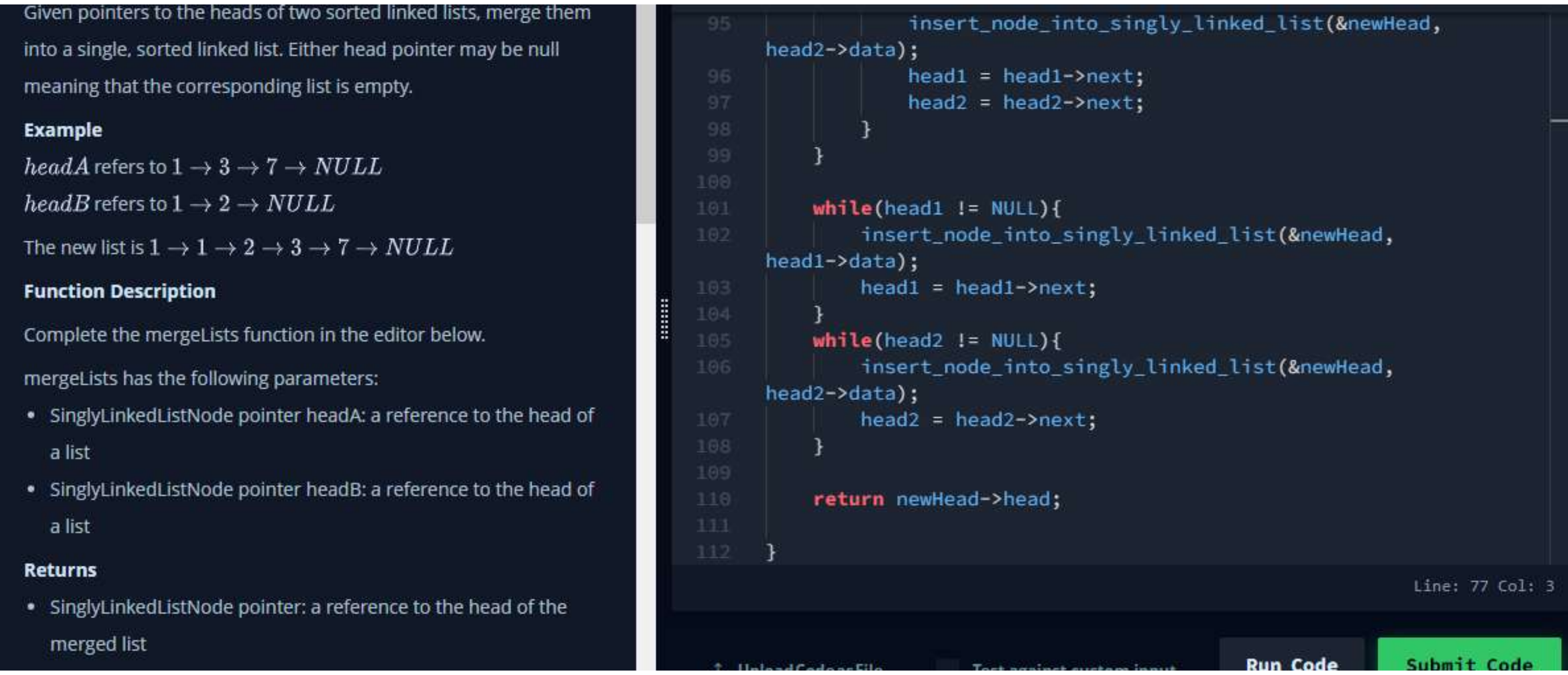
- SinglyLinkedListNode pointer headA: a reference to the head of a list
- SinglyLinkedListNode pointer headB: a reference to the head of a list

Returns

- SinglyLinkedListNode pointer: a reference to the head of the merged list

Input Format

```
 */
SinglyLinkedListNode* mergeLists(SinglyLinkedListNode* head1,
SinglyLinkedListNode* head2) {
 SinglyLinkedList *newHead = malloc(sizeof(SinglyLinkedList));
    newHead->head = NULL;
    newHead->tail = NULL;

    while(head1 != NULL && head2 != NULL){
        if(head1->data > head2->data){
            insert_node_into_singly_linked_list(&newHead,
head2->data);
            head2 = head2->next;
        }
        else if(head1->data < head2->data){
            insert_node_into_singly_linked_list(&newHead,
head1->data);
            head1 = head1->next;
        }
        else{
            insert_node_into_singly_linked_list(&newHead,
head1->data);
            insert_node_into_singly_linked_list(&newHead,
```



Given pointers to the heads of two sorted linked lists, merge them into a single, sorted linked list. Either head pointer may be null meaning that the corresponding list is empty.

Example

$headA$ refers to $1 \to 3 \to 7 \to NULL$

$headB$ refers to $1 \to 2 \to NULL$

The new list is $1 \to 1 \to 2 \to 3 \to 7 \to NULL$

Function Description

Complete the mergeLists function in the editor below.

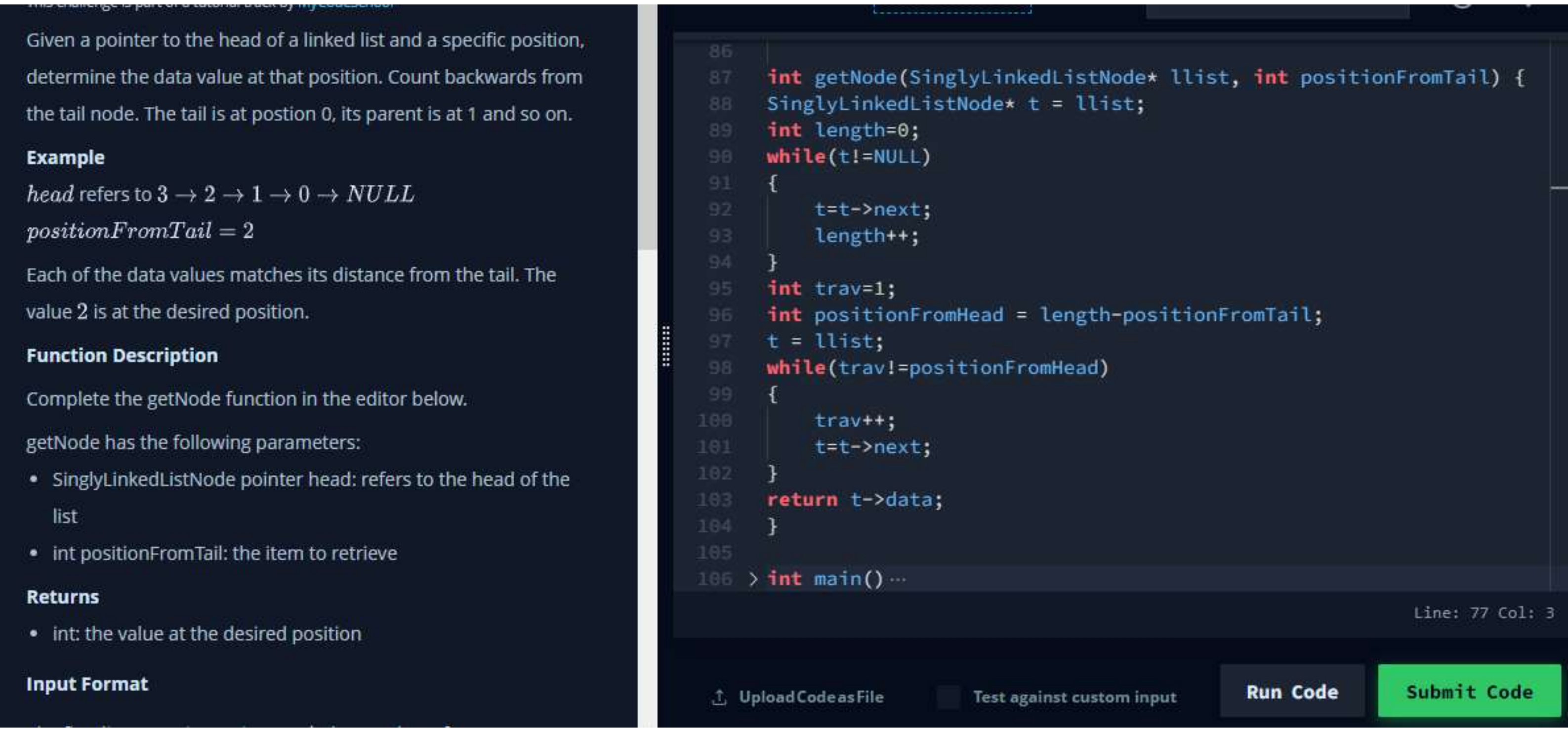
mergeLists has the following parameters:

- SinglyLinkedListNode pointer headA: a reference to the head of a list
- SinglyLinkedListNode pointer headB: a reference to the head of a list

Returns

- SinglyLinkedListNode pointer: a reference to the head of the merged list

```
            insert_node_into_singly_linked_list(&newHead,
head2->data);
            head1 = head1->next;
            head2 = head2->next;
        }
    }

    while(head1 != NULL){
        insert_node_into_singly_linked_list(&newHead,
head1->data);
        head1 = head1->next;
    }
    while(head2 != NULL){
        insert_node_into_singly_linked_list(&newHead,
head2->data);
        head2 = head2->next;
    }

    return newHead->head;

}
```



This challenge is part of a tutorial track by MyCodeSchool

Given a pointer to the head of a linked list and a specific position, determine the data value at that position. Count backwards from the tail node. The tail is at postion 0, its parent is at 1 and so on.

Example

$head$ refers to $3 \rightarrow 2 \rightarrow 1 \rightarrow 0 \rightarrow NULL$

$positionFromTail = 2$

Each of the data values matches its distance from the tail. The value 2 is at the desired position.

Function Description

Complete the getNode function in the editor below.

getNode has the following parameters:

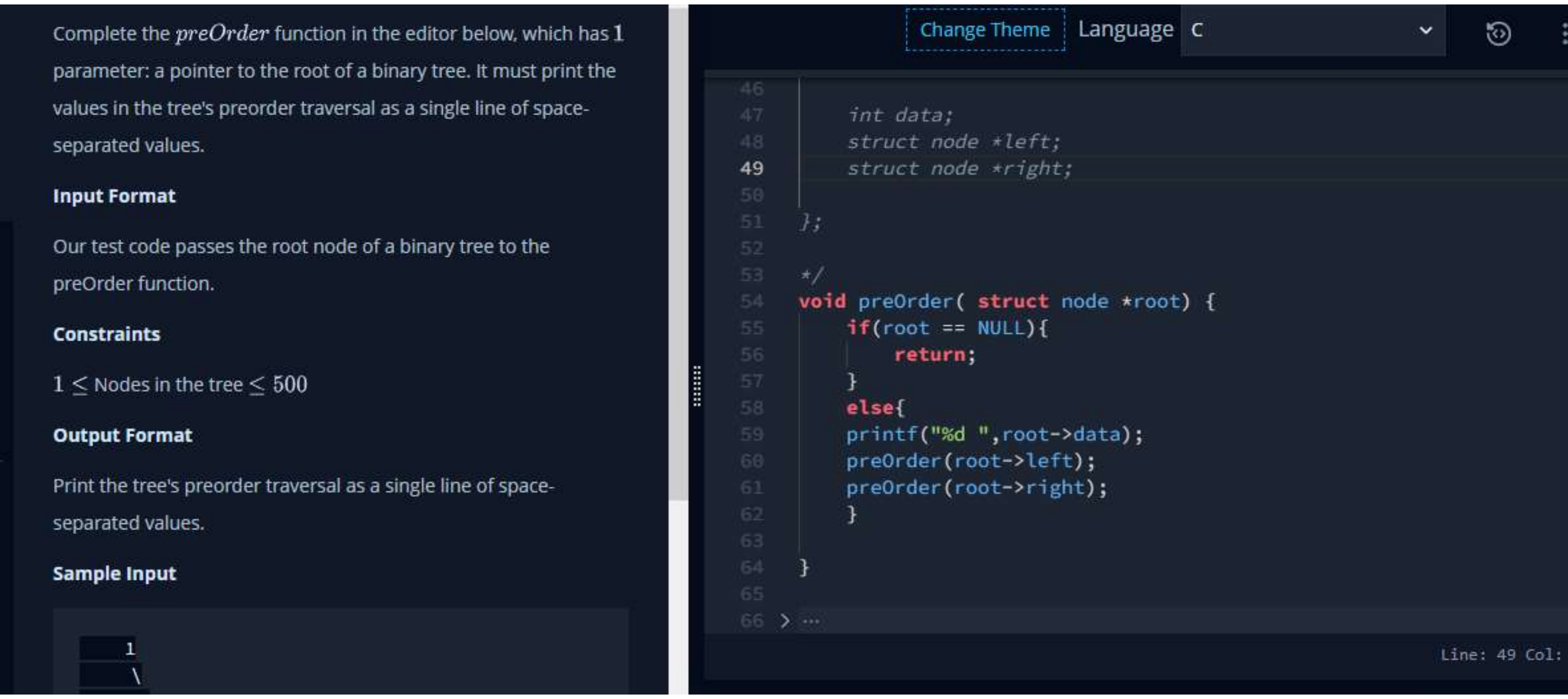
- SinglyLinkedListNode pointer head: refers to the head of the list
- int positionFromTail: the item to retrieve

Returns

- int: the value at the desired position

Input Format

```
int getNode(SinglyLinkedListNode* llist, int positionFromTail) {
SinglyLinkedListNode* t = llist;
int length=0;
while(t!=NULL)
{
    t=t->next;
    length++;
}
int trav=1;
int positionFromHead = length-positionFromTail;
t = llist;
while(trav!=positionFromHead)
{
    trav++;
    t=t->next;
}
return t->data;
}

> int main() ···
```



Complete the $preOrder$ function in the editor below, which has 1 parameter: a pointer to the root of a binary tree. It must print the values in the tree's preorder traversal as a single line of space-separated values.

Input Format

Our test code passes the root node of a binary tree to the preOrder function.

Constraints

$1 \leq$ Nodes in the tree ≤ 500

Output Format

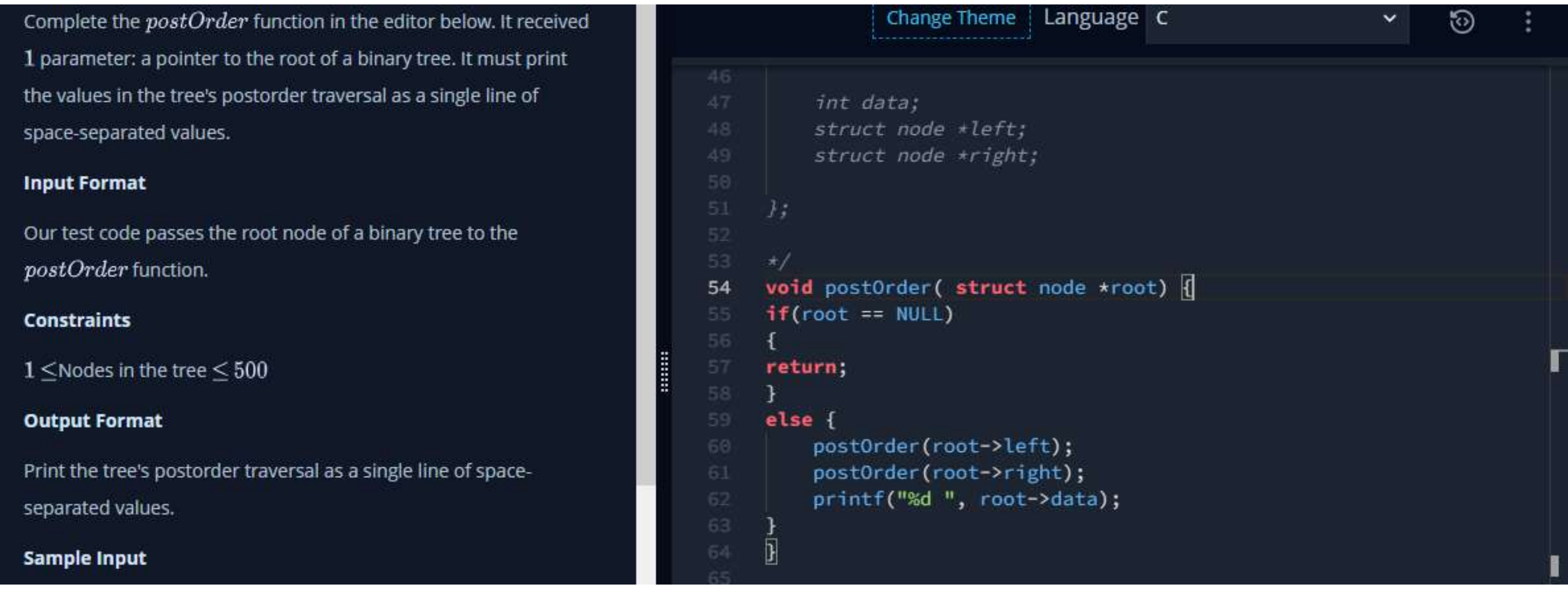
Print the tree's preorder traversal as a single line of space-separated values.

Sample Input

```
     1
      \
```

Language C

```

    int data;
    struct node *left;
    struct node *right;

};

*/
void preOrder( struct node *root) {
    if(root == NULL){
        return;
    }
    else{
    printf("%d ",root->data);
    preOrder(root->left);
    preOrder(root->right);
    }

}

> ...
```



Complete the $postOrder$ function in the editor below. It received 1 parameter: a pointer to the root of a binary tree. It must print the values in the tree's postorder traversal as a single line of space-separated values.

Input Format

Our test code passes the root node of a binary tree to the $postOrder$ function.

Constraints

$1 \leq$Nodes in the tree ≤ 500

Output Format

Print the tree's postorder traversal as a single line of space-separated values.

Sample Input

Language C

```

    int data;
    struct node *left;
    struct node *right;

};

*/
void postOrder( struct node *root) {
if(root == NULL)
{
return;
}
else {
    postOrder(root->left);
    postOrder(root->right);
    printf("%d ", root->data);
}
}
```