**1. Problem Discussion :**

1• You are given a partially written LinkedList class. 2• You are required to complete the body of the IsPalindrome function. The function is expected to check if the linked list is a palindrome or not and return true or false accordingly. 3• Input and Output are managed for you.

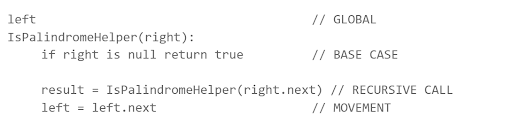
If you face any difficulty understanding the problem, watch the Question video.

**2. Approach :**

In a normal linked list, we cannot access the first and the last, the second and the second last element at the same time. If we use a function like getNodeAt() it will take O(n) and thereby make the entire algorithm quadratic. So, what should we rather do?

We can keep two variables, left and right. Left will be in the heap, while right will be passed through the stack. Now, how do we pass elements through the stack? Yes, by using a recursive function.

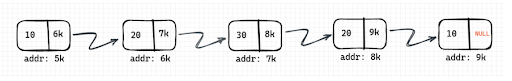
**3. Pseudocode**

****

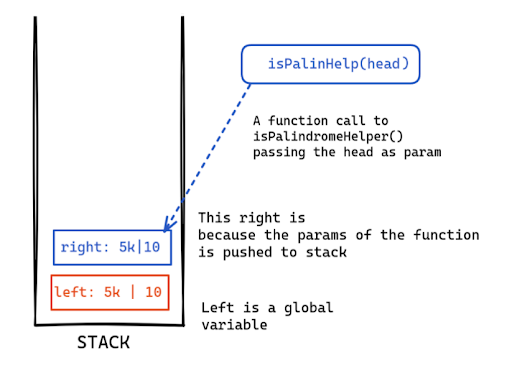
we will call this function as IsPalindromeHelper(head) i.e pass head to right initially. Let's look at the stack for this function. Also remember we have a global variable called left.

**4. Analysis:**

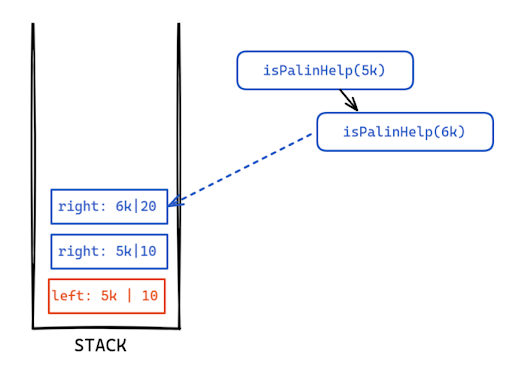
Let's assume the linked list is like this:

****

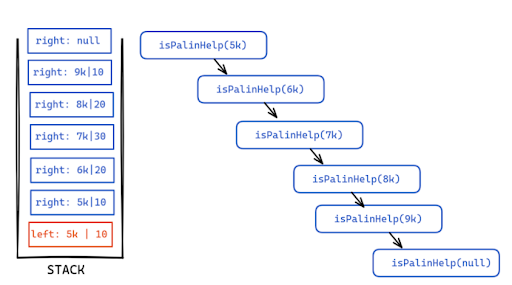
Now, let's say we initialize left with head. So there will be the corresponding data in the heap. Also when we call the function isPalindromeHelper(head) the param i.e right will be stored on the heap. So it will look like this:

****

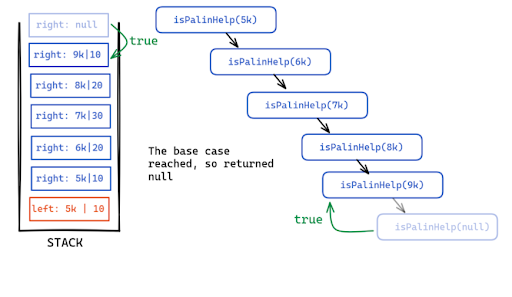
Now this function will return again and call the isPalindromeHelper(right.next) i.e 6k will be passed.

****

Now this function will be called isPalindromeHelper(7k) and so on. We will keep doing this until we reach the base case. So it will look like this:

****

Now that we reach the base case it will return true.

****

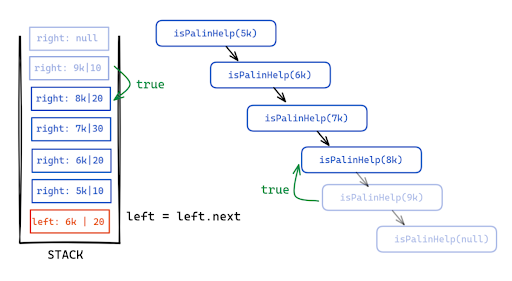
At this point if you look left is pointing to the first node and the right is pointing to the last node. Now after this function returns the top of the stack will again be popped and our right will become 8k. So, to maintain congruence we have to increment left to point to its next i.e 6k. Now, the left will point to the second node, and the right will point to the second from the last node.

Clearly, we are seeing a pattern. This way we can have the pointer left to move from left to right and the pointer right-pointing from right to left. Isn't it an amazing trick?

Now to check palindromes, all we need to do is see if the left and right values are the same or not. If they are not the same we will return false. Else we will return true.

Also, if the result from the recursive call is false, which means somewhere the data didn't match then we will directly return false.

Let's come back to our example. Now the value at right i.e 9k is 10 and that of left i.e 5k is also 10, so we will return true.

****

**5. CODE**

ConsoleCpp

#include <bits/stdc++.h>

using namespace std;

class node

{

public :

int data;

node \*next;

};

class linked\_list

{

public:

node \*head,\*tail;

int size=0;

public:

linked\_list()

{

head = NULL;

tail = NULL;

}

void add\_node(int n)

{

node \*tmp = new node;

tmp->data = n;

tmp->next = NULL;

if(head == NULL)

{

head = tmp;

tail = tmp;

}

else

{

tail->next = tmp;

tail = tail->next;

}

size++;

}

void display(){

for(node\* tmp = head; tmp != NULL; tmp = tmp->next){

cout<<tmp->data<<" ";

}

}

node\* pleft;

int IsPalindromeHelper(node\* right){

if(right == NULL){

return 1;

}

int rres = IsPalindromeHelper(right->next);

if(rres == 0){

return 0;

} else if(pleft->data != right->data){

return 0;

} else {

pleft = pleft->next;

return 1;

}

}

int isPalindrome(){

pleft = head;

return IsPalindromeHelper(head);

}

}

;

int main()

{

int b ;

cin>>b;

linked\_list a;

vector<int> arr(b,0);

for(int i=0;i<b;i++)

{

cin>>arr[i];

a.add\_node(arr[i]);

}

int res = a.isPalindrome();

if(res==1)

{

cout<<"true";

}

else

{

cout<<"false";

}

return 0;

}

**6. Analysis:**

Time Complexity:

This is an O(n) solution because all you need to do is look at the recursive stack. How many items/nodes do you see? Yes. n items where n is the size of the linked list.

And in each recursive call, we are doing constant-time operations. So it will be O(n).

Space Complexity:

Even though we are not using any auxiliary space, we are using the run-time stack n times to store the params of the recursive function. So it will be O(n) as well.