Sign In

COUGARCS

OCTOBER 9TH, 2023

DATA STRUCTURES
MIDTERM REVIEW

COUGARCS

Agenda

1. Warmup with Linked Lists
2. Introduce Stacks and Queues
3. Merge Sort, Quick Sort, & Heap Sort
4. Breakout!

Warm up: Linked Lists

Given the head of a sorted Linked List, **delete all duplicates** such that the element only appears once.

Example:

Warm up: Solution

To solve this problem:

1. Edge case: check if there's only one node. Or if the Linked List is empty
2. Keep track of the previous node as you traverse the Linked List.
3. If the previous node is a duplicate.
    - If there's a duplicate delete it.
    - Otherwise continue traversing

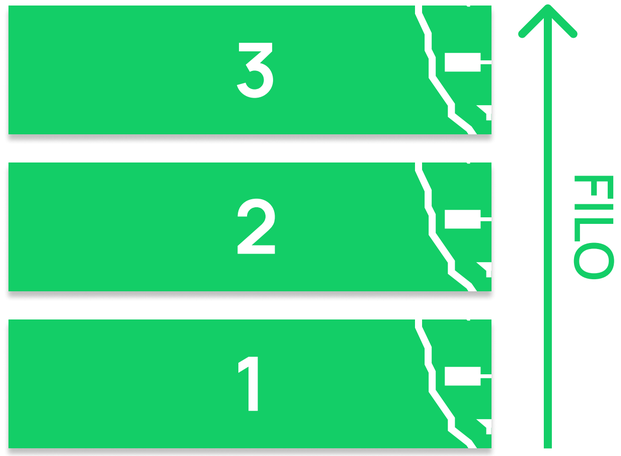
```
9   class Solution {
10  public:
11      ListNode* deleteDuplicates(ListNode* head) {
12          if(head == nullptr  ||  head->next == nullptr)
13              return head;
14
15          ListNode*prev = nullptr;
16          ListNode*temp = nullptr;
17          ListNode*curr = head;
18
19          while(curr != nullptr){
20              if(prev == nullptr){
21                  prev = curr;
22                  temp = prev;
23                  curr = curr->next;
24                  continue;
25              }
26              if(curr->val == prev->val){
27                  curr=curr->next;
28                  if(curr == nullptr) prev->next=nullptr;
29              }
30              else{
31                  prev->next = curr;
32                  prev = curr;
33                  curr = curr->next;
34              }
35          }
36          return temp;
37      }
38  };
```

What are Stacks?

Stacks are a linear data structure that insert and remove items according to the **First In Last Out (FILO)**.

Abstract Data Type (ADT) of Stacks:

- push(e): Inserts element e to the top.
- pop(): Removes the top element.
- top(): Returns the top element.
- size(): Returns the number of elements.
- empty(): Returns true if empty.

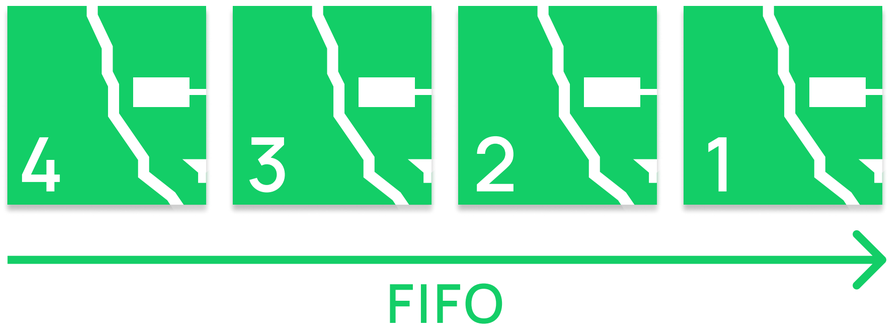


What are Queues?

Queues are a linear data structure that insert and remove items according to the **First In First Out (FIFO)**.

Abstract Data Type (ADT) of Queues:

- **push(e)**: Inserts element e to the back.
- **pop()**: Removes the front element.
- **front()**: Returns the front element.
- **size()**: Returns the number of elements.
- **empty()**: Returns true if empty.



Practice: Find a Palindrome Using a Stack and a Queue

A **palindrome** is a word that reads the same forwards and backwards. For example the word ‘Kayak’ or ‘Noon’.

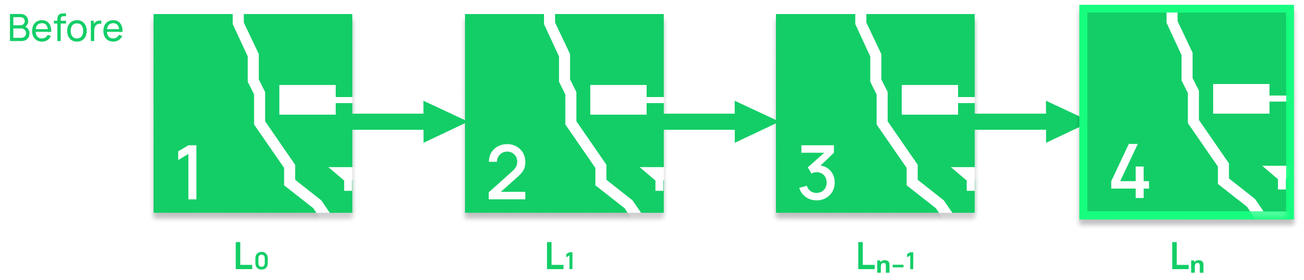
Given a string, determine whether or not it is a palindrome by using a stack and queue.

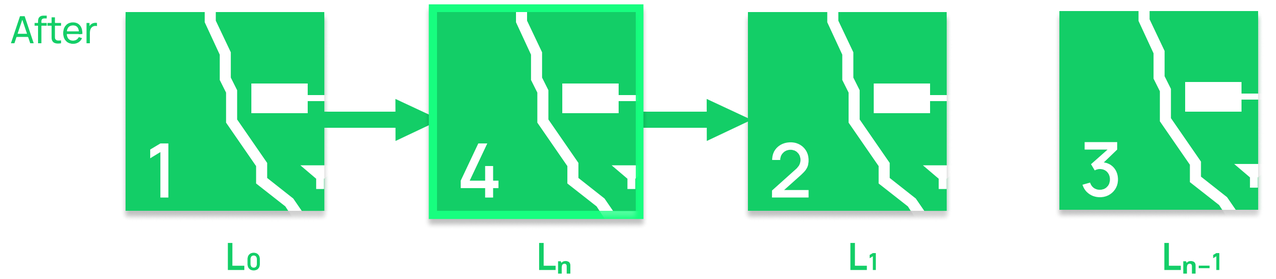
Solution: Find a Palindrome Using a Stack and a Queue

```
6   bool ispalindrome(Stack<char> stack , Queue<char> queue){
7     if(queue.isempty() && stack.isempty())
8       return true;
9
10    if(queue.getSize() != stack.getSize())
11      return false;
12
13    if(stack.top() == queue.front()->data) {
14        stack.pop();
15        queue.pop();
16        return ispalindrome(stack, queue);
17    } else
18      return false;
19  }
```

Practice: Reorder a Linked List Using a Stack

Given a head of a singly Linked List, reorder the list to be of the order shown below:





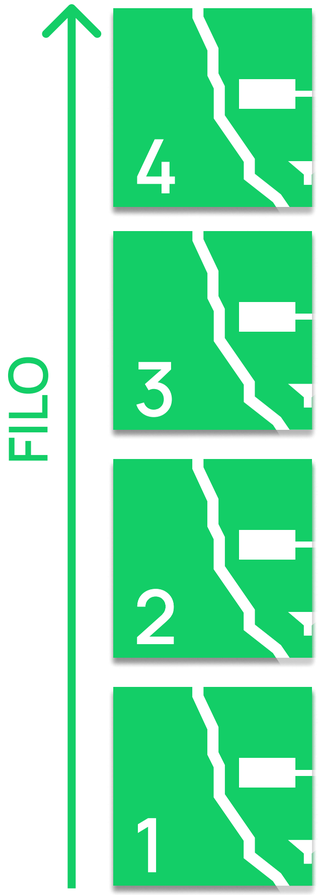
Practice: Reorder a Linked List Using a Stack

To solve this problem:

1. Edge case: check if there's only one node or if the list is empty.
2. Iterate through the list, pushing each node into our stack.
    - Count the number of nodes!
3. Iterate for only half of the list.
    - Pop from the stack and connect the current node to the popped node.
        - This should be the last element we have added to our stack.

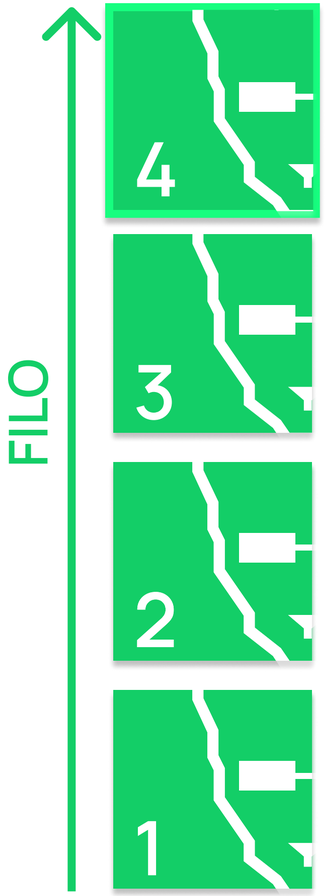
```
11  class Solution {
12  public:
13      void reorderList(ListNode* head) {
14          if ((!head) || (!head->next) || (!head->next->next)) return;
15
16          stack<ListNode*> my_stack;
17          ListNode* ptr = head;
18          int size = 0;
19          while (ptr != NULL)
20          {
21              my_stack.push(ptr);
22              size++;
23              ptr = ptr->next;
24          }
25
26          ListNode* pptr = head;
27          for (int j=0; j<size/2; j++)
28          {
29              ListNode *element = my_stack.top();
30              my_stack.pop();
31              element->next = pptr->next;
32              pptr->next = element;
33              pptr = pptr->next->next;
34          }
35          pptr->next = NULL;
36      }
37  };
```

Practice: Reorder a Linked List Using a Stack



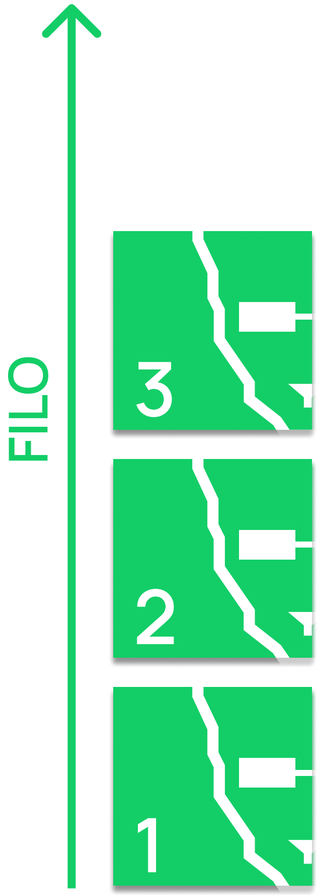
We first check the top of the stack and assign that node to a pointer.

Practice: Reorder a Linked List Using a Stack



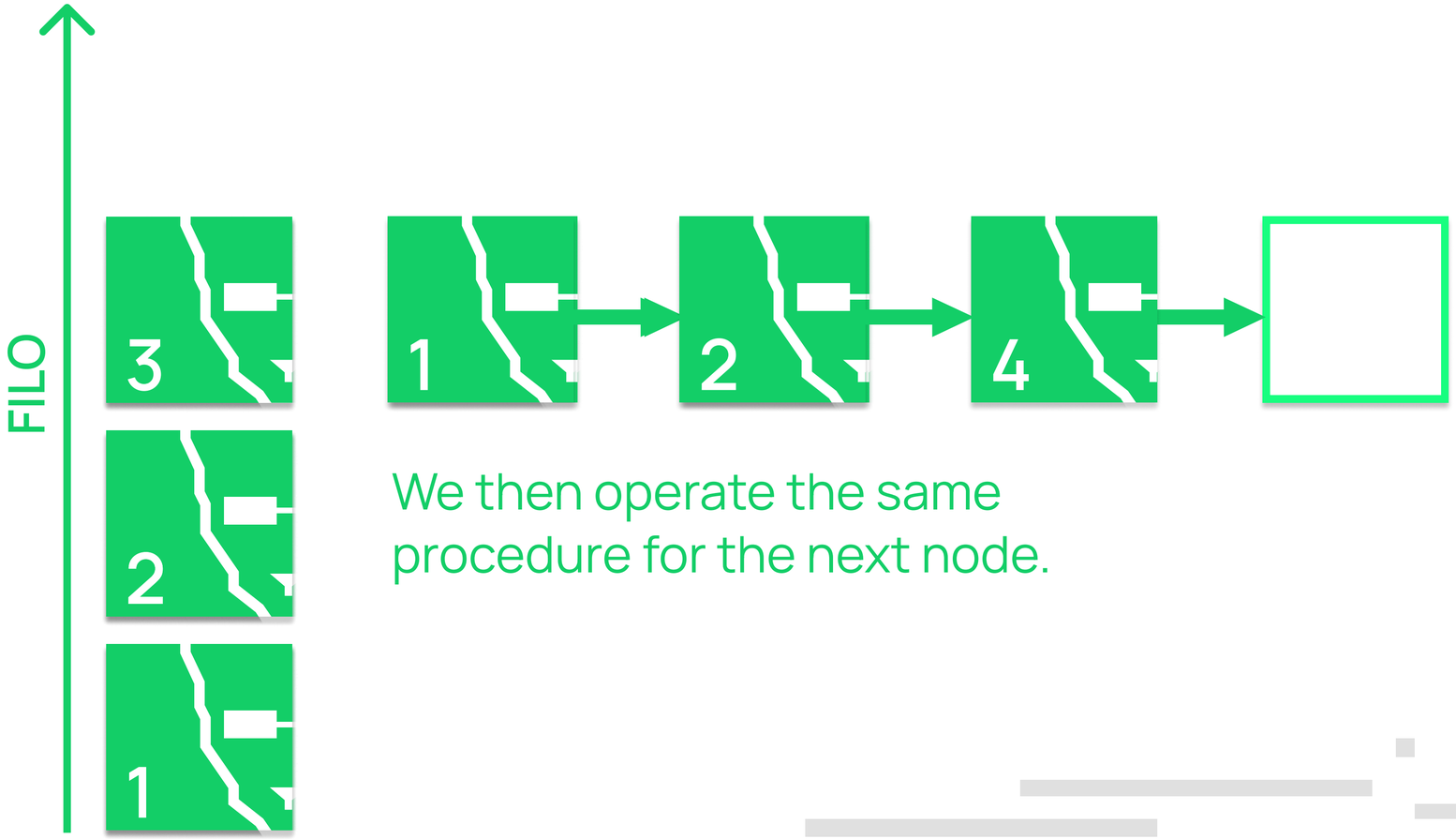
The next thing we do is pop this node from the stack.

Practice: Reorder a Linked List Using a Stack



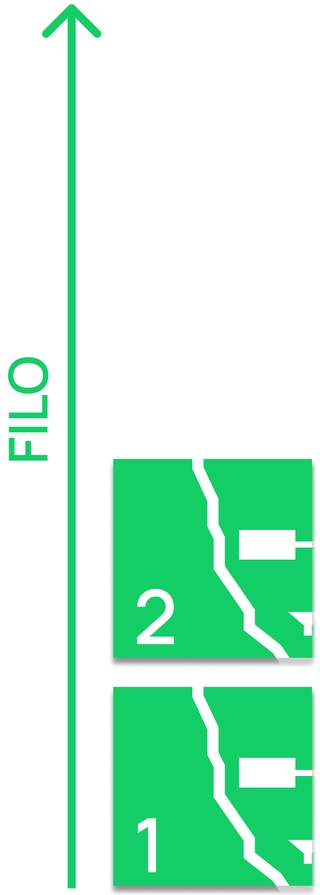
Once that's done, we are now able to use that temporary node from earlier and insert it into our list.

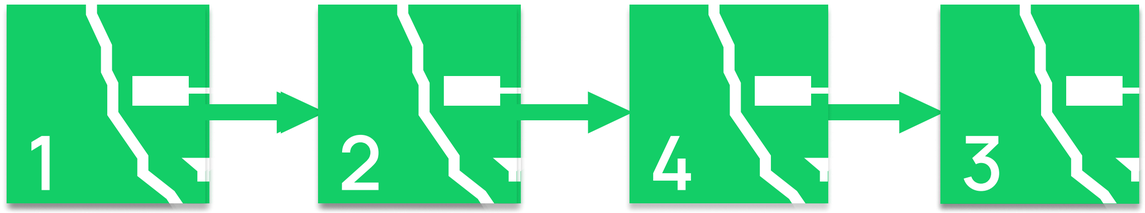
Practice: Reorder a Linked List Using a Stack



We then operate the same procedure for the next node.

Practice: Reorder a Linked List Using a Stack





This process is repeated until we've reached the midpoint of the stack. (n/2).

Sorts for Exam 2

O(n log n)

- Merge Sort
- Quick Sort
- Heap Sort

O (n^2) (Dan only)

- Insertion
- Selection
- Bubble

Merge Sort Overview

Given an array divide into smaller sub arrays, sort each sub array then merge back together to form sorted arrays.

Merge Sort's best, average and worst time complexity are O(n log n) space complexity is O(n)

Merge Sort

Each successive split halves the array.

Repeat until you have one node left in the subarray

Heap Sort Overview

Given an array perform a **heapify** operation (either min or max depending on implementation), then pop the root and place it in the next available space, perform heapify continuously of n-1 sizes until the list is sorted.

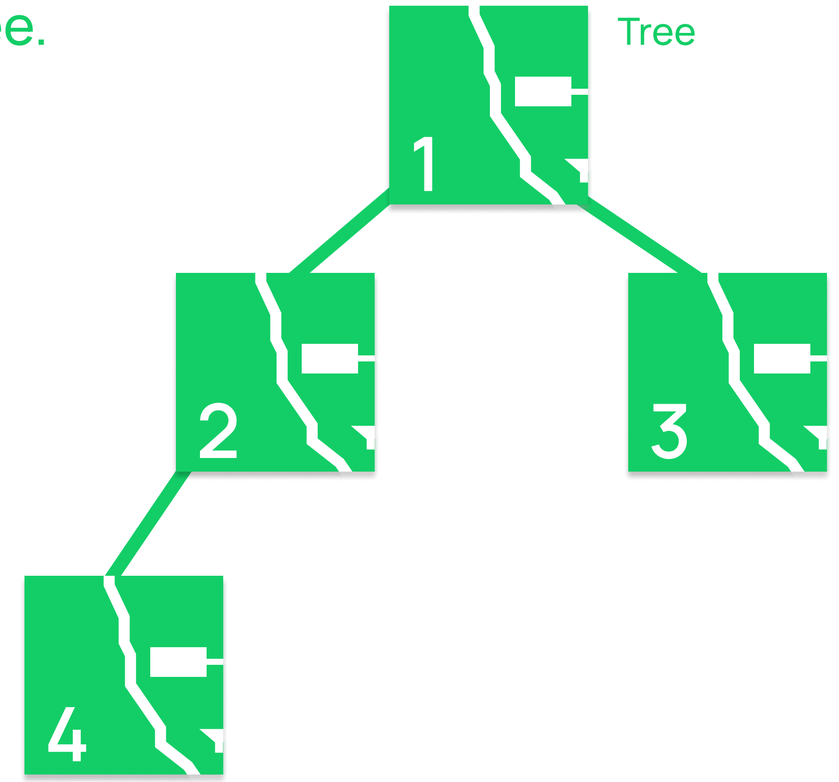
Best, worst and average time complexity are O(n log n), space complexity is O(n)

Relationship Between an Array and a Heap

A heap data structure is best represented by a binary tree.

- For each index **i** in the array:
  - It's left child is at **2i+1**
  - It's right child is at **2i+2**

Array



Tree

Pseudocode for Heap Sort

```
function heapsort(array):
    for i >= 0, decrement i:
        swap(array[0], arr[i])
        heapify(array, i, 0)
```

Heapsort function

```
function heapify(array, n, i):
    int largest = i
    int left = 2 * i + 1
    int right = 2 * i + 2

    if left < n and array[left] > array[largest]
        largest = left

    if right < n and array[right] > array[largest]
        largest = right

    if largest != i
        swap array[i] and array[largest]
        heapify(array, n, largest)
```

Heapif helper function

Quick Sort Overview

Given an array choose any pivot (usually 0 or n-1) and partition the array into two sub arrays where one side is less than the pivot and the other side greater and recursively sort the sub arrays using the same method.

Worst case is O(n^2) best and average case are n log n, time complexity is determined by how good the pivot choice is. Space complexity is O(n)

Quick Sort

We choose a pivot at array[0]

The array is divided into subarrays.

1. One subarray is made up of values less than the pivot.
2. Another is made for values greater than the pivot.

Theory question (Mostly for Dan students)

Given knowledge of quick sort and quick select explain how quick select has a better time complexity for finding certain elements (max/min/median/3 largest/etc) and what is that time complexity?

BREAKOUT

Feel free to ask for help!

COUGARCS