Stacks & Queues

Q1:

Write a function that takes in two stacks of varied length and returns a linked list containing the sum of the two stacks' numbers. If one stack is longer than the other, the remaining digits should be added to the linked list at the end.

Example:

- stack A: (top) 9 → 6 → 4 → 4 → 2 → 1
- stack B: (top) 10 → 9 → 8 → 3 → 0 → 6 → 7

Return:

- **head** of linked list C: 19 → 15 → 12 → 7 → 2 → 7 → 7

```
Node* addTwoStacks(stack st1, stack st2){









}
```

Q2:

Leetcode 20. Valid Parentheses

Given a string s containing just the characters:
`'(', ')', '{', '}', '[' and ']'`
determine if the input string is valid.

An input string is valid if:

- Open brackets must be closed by the same type of brackets.
- Open brackets must be closed in the correct order.
- Every close bracket has a corresponding open bracket of the same type.

Note: assume string s consists of parentheses only `'()[]{}'`.

Example 1:

- `Input: s = "()"`
- `Output: true`

Example 2:

- `Input: s = "()[]{}"`
- `Output: true`

Example 3:

- `Input: s = "(]"`
- `Output: false`

```
class Solution {
public:
    bool isValid(string s) {









    }
};
```

https://leetcode.com/problems/valid-parentheses/

Q3:

Leetcode 225. Implement Stack using Queues

Implement a last-in-first-out (LIFO) stack using only two queues. The implemented stack should support all the functions of a normal stack (push, top, pop, and empty).

Implement the MyStack class:

`void push(int x)` Pushes element x to the top of the stack.

`int pop()` Removes the element on the top of the stack and returns it.
`int top()` Returns the element on the top of the stack.
`boolean empty()` Returns true if the stack is empty, false otherwise.
Example:

- `Input:`
  ```
  ["MyStack", "push", "push", "top", "pop", "empty"]
  [[], [1], [2], [], [], []]
  ```
- `Output:`
  ```
  [null, null, null, 2, 2, false]
  ```

Explanation:

```
MyStack myStack = new MyStack();
myStack.push(1);
myStack.push(2);
myStack.top(); // return 2
myStack.pop(); // return 2
myStack.empty(); // return False

class MyStack {
public:

    MyStack() {

    }

    void push(int x) {





    }

    int pop() {
```

```
    }

    int top() {

    }

    bool empty() {

    }
};

/**
 * Your MyStack object will be called as such:
 * MyStack* obj = new MyStack();
 * obj->push(x);
 * int param_2 = obj->pop();
 * int param_3 = obj->top();
 * bool param_4 = obj->empty();
 */
```

https://leetcode.com/problems/implement-stack-using-queues/description/

Q4:

Leetcode 387. First Unique Character in a String

```
class Solution {
public:
    int firstUniqChar(string s) {
        }
};
```
https://leetcode.com/problems/first-unique-character-in-a-string/

Q5:

Write a program that can convert a postfix expression into an infix expression using a stack.

- A **postfix expression** is an expression where the operators are written after the operands such as: abc+++.
- An **infix expression** is where the operators are between operands. It's what we're already used to, so for example like: a + b + c.

Linked Lists

Q1:

You are given a doubly linked list. Write a C++ function to print the middle element of the list. You can assume that the list of numbers will always be odd.

```
void printMiddleElement(node* head) { . . . }
```

Example:

1 ←› 2 ←› 3 ←› 4 ←› 5

Calling printMiddleElement(head) should print:

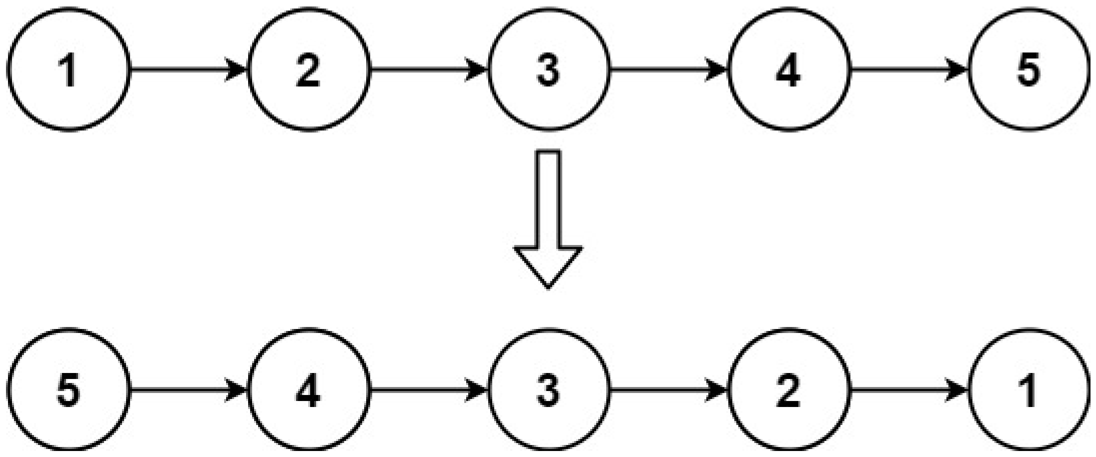
3

Q2:

Leetcode 206. Reverse Linked List

Given the head of a singly linked list, reverse the list, and return the reversed list.

Example:



Input: head = [1,2,3,4,5]

Output: [5,4,3,2,1]

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* reverseList(ListNode* head) {

    }
};
```

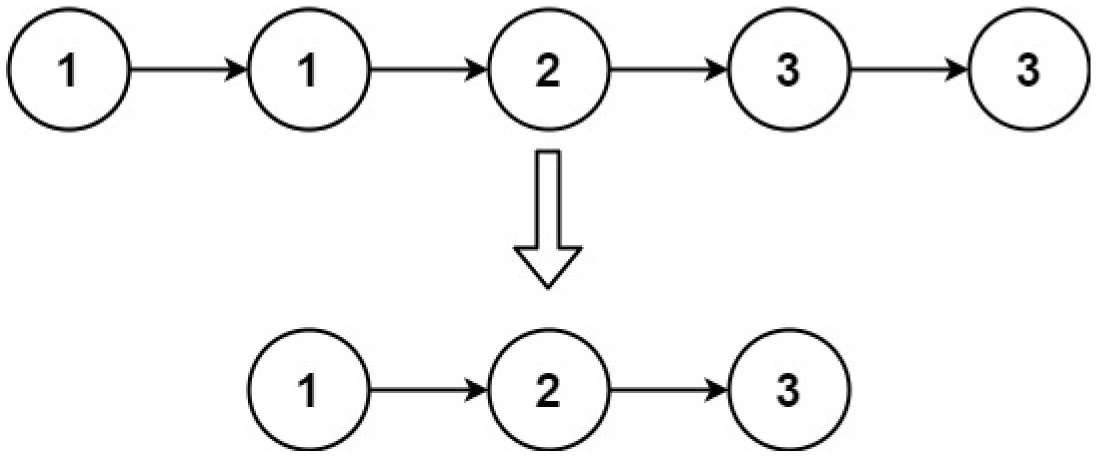
https://leetcode.com/problems/reverse-linked-list/

Q3:

Leetcode 83. Remove Duplicates from Sorted List

Given the head of a sorted linked list, delete all duplicates such that each element appears only once. Return the linked list sorted as well.

Example:



Input: head = [1,1,2,3,3]

Output: [1,2,3]

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {

    }
};
```

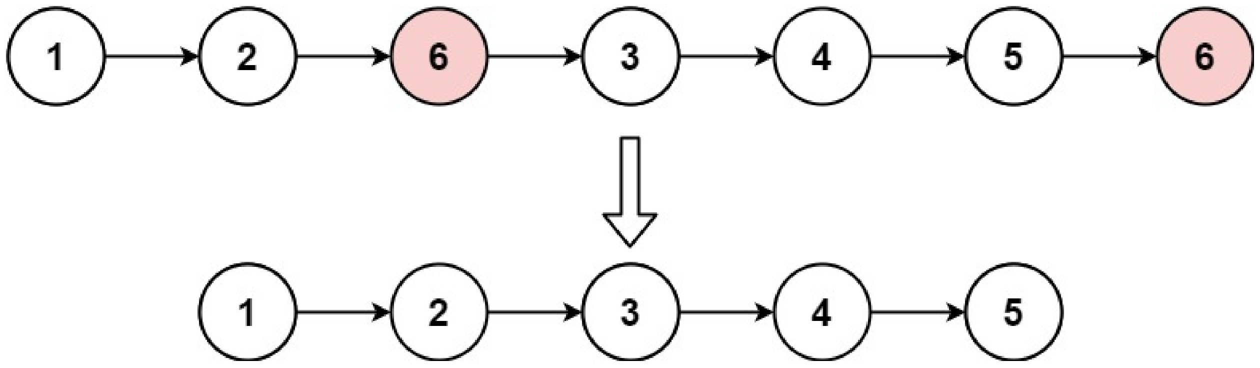
https://leetcode.com/problems/remove-duplicates-from-sorted-list/

Q4:

Leetcode 203. Remove Linked List Elements

Given the head of a linked list and an integer val, remove all the nodes of the linked list that has `Node.val == val`, and return the new head.

Example 1:



Input: head = [1,2,6,3,4,5,6], val = 6

Output: [1,2,3,4,5]

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* removeElements(ListNode* head, int val) {

    }
};
```

https://leetcode.com/problems/remove-linked-list-elements/description/

Q5:

a. List and explain the types of Linked Lists you learned about in class.
b. What does "traversal" of a Linked List mean? Explain the process.
c. What are some advantages and disadvantages of Linked Lists compared to arrays?

Heaps

Q1:

Build the min/max heap for the following array A = {8, 15, 9, 3, 12, 5}. Element in the array will be added to the heap one by one starting from index 0.

Q2:

Consider the following binary max heap: [25, 14, 16, 13, 10, 8, 12]. What is the content of the array after two delete operations? You may draw the tree or give the array representation if you'd like.

Sorts

Q1:

Show this array: A = {38, 27, 43, 3, 9, 82, 10}
After each pass of these sorts: Insertion, Selection, Mergesort, Quicksort