DOMAIN WINTER CAMP

(Department of Computer Science and Engineering)

Name: Aman Bansal UID: 22BCS13365 Section: KPIT 901-B

DAY-4

(Easy)

Q1 Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Implement the MinStack class:

- **MinStack() initializes the stack object.**
- **void push(int val) pushes the element val onto the stack.**
- **void pop() removes the element on the top of the stack.**
- **int top() gets the top element of the stack.**
- **int getMin() retrieves the minimum element in the stack.**

You must implement a solution with O(1) time complexity for each function.

Example 1:

Input

["MinStack","push","push","push","getMin","pop","top","getMin"]

[[],[-2],[0],[-3],[],[],[],[]]

Program code:

```
#include <iostream>
#include <stack>
#include <vector>
```

```
#include <string>
using namespace std;

class CustomStack {
private:
    stack<int> primaryStack;   // Stack to hold all elements
    stack<int> minTrackerStack;    // Stack to keep track of
minimum elements

public:
    CustomStack() {}

    void add(int value) {
        primaryStack.push(value);
        if (minTrackerStack.empty() || value <=
minTrackerStack.top()) {
            minTrackerStack.push(value);
        }
    }

    void remove() {
        if (!primaryStack.empty() && primaryStack.top() ==
minTrackerStack.top()) {
            minTrackerStack.pop();
        }
        primaryStack.pop();
    }

    int peek() {
        return primaryStack.top();
    }

    int getMinimum() {
        return minTrackerStack.top();
    }
};

int main() {
    vector<string> commands = {"CustomStack", "add",
"add", "add", "getMinimum", "remove", "peek",
"getMinimum"};
```

```
    vector<vector<int>> parameters = {{}, {-2}, {0}, {-3},
{}, {}, {}, {}};
    vector<string> results;

    CustomStack* customStack = nullptr;

    for (size_t i = 0; i < commands.size(); ++i) {
        if (commands[i] == "CustomStack") {
            customStack = new CustomStack();
            results.push_back("null");
        } else if (commands[i] == "add") {
            customStack->add(parameters[i][0]);
            results.push_back("null");
        } else if (commands[i] == "remove") {
            customStack->remove();
            results.push_back("null");
        } else if (commands[i] == "peek") {
            results.push_back(to_string(customStack->peek()));
        } else if (commands[i] == "getMinimum") {
            results.push_back(to_string(customStack-
>getMinimum()));
        }
    }

    // Print the results
    cout << "[";
    for (size_t i = 0; i < results.size(); ++i) {
        cout << results[i];
        if (i < results.size() - 1) cout << ",";
    }
    cout << "]" << endl;

    return 0;
}
```

Output:

Q 2 (Medium) Given a circular integer array nums (i.e., the next element of nums[nums.length - 1] is nums[0]), return the next greater number for every element in nums.

The next greater number of a number x is the first greater number to its traversing-order next in the array, which means you could search circularly to find its next greater number. If it doesn't exist, return -1 for this number.

Example 1:

Input: nums = [1,2,1]

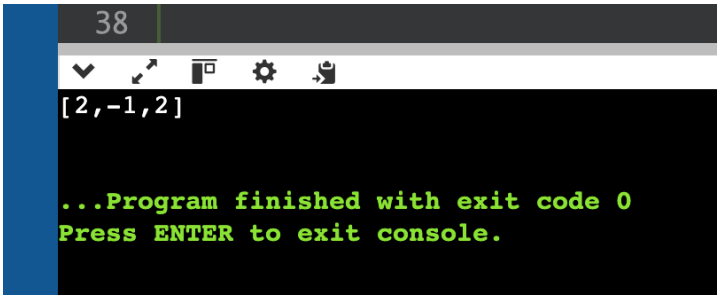
Output: [2,-1,2]

Program code:

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;

vector<int> findNextGreaterElements(vector<int>&
elements) {
   int size = elements.size();
   vector<int> nextGreater(size, -1); // Initialize the result
array with -1
   stack<int> indexStack;           // Monotonic stack to store
indices

   // Traverse the array twice to handle the circular nature
   for (int index = 0; index < 2 * size; index++) {
      while (!indexStack.empty() &&
elements[indexStack.top()] < elements[index % size]) {
         nextGreater[indexStack.top()] = elements[index %
size];
         indexStack.pop();
      }
      if (index < size) {
         indexStack.push(index);
      }
   }

   return nextGreater;
}
```

```
int main() {
    vector<int> elements = {1, 2, 1};
    vector<int> nextGreater =
findNextGreaterElements(elements);

    cout << "[";
    for (size_t i = 0; i < nextGreater.size(); i++) {
        cout << nextGreater[i];
        if (i < nextGreater.size() - 1) cout << ",";
    }
    cout << "]" << endl;

    return 0;
}
```

Output:



Ques 3. Given a queue, write a recursive function to reverse it.

Standard operations allowed :

enqueue(x) : Add an item x to rear of queue.
dequeue() : Remove an item from front of queue.
empty() : Checks if a queue is empty or not.

Examples 1:

Input : Q = [5, 24, 9, 6, 8, 4, 1, 8, 3, 6]

Output : Q = [6, 3, 8, 1, 4, 8, 6, 9, 24, 5]

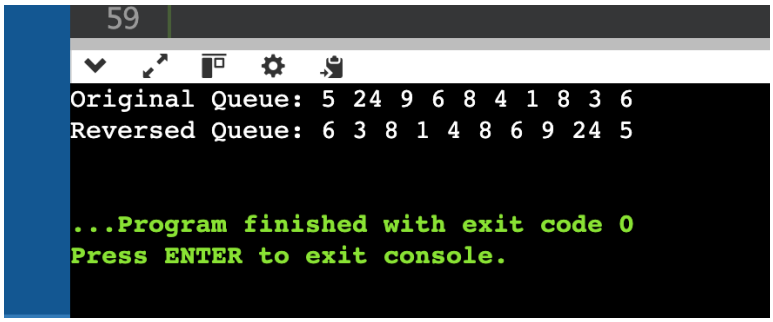
Program Code:

```
#include <iostream>
#include <queue>
using namespace std;
```

```
// Function to reverse the queue recursively
void reverseQueue(queue<int>& q) {
    // Base case: if the queue is empty, return
    if (q.empty()) {
        return;
    }

    // Step 1: Remove the front element of the queue
    int front = q.front();
    q.pop();

    // Step 2: Recursively reverse the remaining queue
    reverseQueue(q);

    // Step 3: Add the removed element to the back of the queue
    q.push(front);
}

int main() {
    // Input queue
    queue<int> Q;
    Q.push(5);
    Q.push(24);
    Q.push(9);
    Q.push(6);
    Q.push(8);
    Q.push(4);
    Q.push(1);
    Q.push(8);
    Q.push(3);
    Q.push(6);

    // Print original queue
    cout << "Original Queue: ";
    queue<int> tempQ = Q; // Temporary queue to preserve original for printing
    while (!tempQ.empty()) {
        cout << tempQ.front() << " ";
        tempQ.pop();
    }
    cout << endl;
```

```
    // Reverse the queue
    reverseQueue(Q);

    // Print reversed queue
    cout << "Reversed Queue: ";
    while (!Q.empty()) {
        cout << Q.front() << " ";
        Q.pop();
    }
    cout << endl;

    return 0;
}
```

Output:



Ques 4. You are given an array of integers nums, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position.

Return the max sliding window.

Example 1:

Input: nums = [1,3,-1,-3,5,3,6,7], k = 3

Output: [3,3,5,5,6,7]

Explanation:

Window position	Max
[1 3 -1] -3 5 3 6 7	3
1 [3 -1 -3] 5 3 6 7	3
1 3 [-1 -3 5] 3 6 7	5

1 3 -1 [-3 5 3] 6 7 5

1 3 -1 -3 [5 3 6] 7 6

1 3 -1 -3 5 [3 6 7] 7

Program Code:

```
#include <iostream>
#include <vector>
#include <deque>
using namespace std;

vector<int> slidingWindowMaximum(vector<int>& elements, int windowSize) {
    deque<int> indexDeque; // Stores indices of array elements
    vector<int> maxValues;

    for (int idx = 0; idx < elements.size(); ++idx) {
        // Remove elements from deque that are out of the current window
        if (!indexDeque.empty() && indexDeque.front() == idx - windowSize) {
            indexDeque.pop_front();
        }

        // Remove elements from deque that are smaller than the current element
        while (!indexDeque.empty() && elements[indexDeque.back()] <
elements[idx]) {
            indexDeque.pop_back();
        }

        // Add the current element's index to the deque
        indexDeque.push_back(idx);

        // Add the maximum element of the current window to the result
        if (idx >= windowSize - 1) {
            maxValues.push_back(elements[indexDeque.front()]);
        }
    }

    return maxValues;
}

int main() {
```

```
    vector<int> elements = {1, 3, -1, -3, 5, 3, 6, 7};
    int windowSize = 3;
    vector<int> maxValues = slidingWindowMaximum(elements, windowSize);

    // Print the result
    cout << "[";
    for (size_t i = 0; i < maxValues.size(); i++) {
        cout << maxValues[i];
        if (i < maxValues.size() - 1) cout << ",";
    }
    cout << "]" << endl;

    return 0;
}
```

Output:

Ques 5 You have an infinite number of stacks arranged in a row and numbered (left to right) from 0, each of the stacks has the same maximum capacity.

Implement the DinnerPlates class:

DinnerPlates(int capacity) Initializes the object with the maximum capacity of the stacks capacity.

void push(int val) Pushes the given integer val into the leftmost stack with a size less than capacity.

int pop() Returns the value at the top of the rightmost non-empty stack and removes it from that stack, and returns -1 if all the stacks are empty.

int popAtStack(int index) Returns the value at the top of the stack with the given index index and removes it from that stack or returns -1 if the stack with that given index is empty.

Program Code:

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;

class PlateStacks {
private:
    int stackCapacity;
    vector<stack<int>> plateStacks;
    int nextAvailableStack;

public:
    // Constructor to initialize the PlateStacks object
    PlateStacks(int capacity) {
        this->stackCapacity = capacity;
        this->nextAvailableStack = 0;
        cout << "null" << endl; // Output null for constructor
    }

    // Pushes a value into the leftmost stack with available
space
    void addPlate(int value) {
        while (nextAvailableStack < plateStacks.size() &&
plateStacks[nextAvailableStack].size() == stackCapacity) {
            nextAvailableStack++;
        }

        if (nextAvailableStack == plateStacks.size()) {
```

```
            plateStacks.push_back(stack<int>());
        }

        plateStacks[nextAvailableStack].push(value);
        cout << "null" << endl; // Output null for addPlate
    }

    // Removes the top plate from the rightmost non-empty
stack
    int removePlate() {
        if (plateStacks.empty()) {
            return -1;
        }

        while (!plateStacks.empty() &&
plateStacks.back().empty()) {
            plateStacks.pop_back();
        }

        if (plateStacks.empty()) {
            return -1;
        }

        int topPlate = plateStacks.back().top();
        plateStacks.back().pop();
        return topPlate;
    }

    // Removes the top plate from a specific stack
```

```
    int removePlateAt(int index) {
        if (index >= plateStacks.size() ||
plateStacks[index].empty()) {
            return -1;
        }

        int topPlate = plateStacks[index].top();
        plateStacks[index].pop();
        return topPlate;
    }
};

int main() {
    PlateStacks ps(2);  // Constructor call, should output 'null'

    ps.addPlate(1);  // Should output 'null'
    ps.addPlate(2);  // Should output 'null'
    ps.addPlate(3);  // Should output 'null'
    ps.addPlate(4);  // Should output 'null'
    ps.addPlate(5);  // Should output 'null'

    cout << ps.removePlateAt(0) << endl; // 2
    ps.addPlate(20);  // Should output 'null'
    ps.addPlate(21);  // Should output 'null'

    cout << ps.removePlateAt(0) << endl; // 20
    cout << ps.removePlateAt(2) << endl; // 21
    cout << ps.removePlate() << endl;    // 5
    cout << ps.removePlate() << endl;    // 4
```

```
    cout << ps.removePlate() << endl;    // 3
    cout << ps.removePlate() << endl;    // 1
    cout << ps.removePlate() << endl;    // -1


    return 0;
}
```

Output: