

Chen Jun Hong
chen.junhong@u.nus.edu

CS2040 Lab 3

AY22/23 Sem 1, Week 5



- Lab 2 is due on 11 September 23:59 PM
- No visualgo quiz in lab sessions
- First visualgo quiz next week: two sessions
 - 14 September 11 AM
 - 15 September 5:30 PM

Lab 3 - Stacks/Queues

Special kinds of lists

- Stacks only allow inserting, accessing and deleting from the top of the stack in $O(1)$ time (First-In Last-Out/Last-In First-Out)
- Queues only allow accessing and deleting from the front of the queue, and inserting from the back of the queue in $O(1)$ time (First-In First-Out/ Last-In Last-Out)

Lab 3 - Stack Example

```
import java.util.Stack; //imports stack (Can use java.util.*)

Stack<Integer> stack = new Stack<>(); //Can replace Integer with another class
of your choice

stack.push(1); //stack = [1] from bottom to top

stack.push(2); //stack = [1,2] from bottom to top

stack.push(3); //stack = [1,2,3] from bottom to top

int number = stack.pop(); //3 is popped from the stack and assigned to number

System.out.println(stack.peek()); // 3 is popped from stack

System.out.println(stack.size()); // Prints 2

System.out.println(stack.isEmpty()); // Prints false
```

Lab 3 - Queue Example

```
import java.util.LinkedList; //imports LinkedList (Can use java.util.*)

Queue<Integer> queue = new LinkedList<>(); //Can replace Integer with another
class of your choice

queue.offer(1); //queue = [1] from front to back

queue.offer(2); //queue = [1,2] from front to back

queue.offer(3); //queue = [1,2,3] from front to back

int number = queue.poll(); //1 is polled from the stack and assigned to number

System.out.println(queue.peek()); // 3 is popped from stack

System.out.println(queue.size()); // Prints 2

System.out.println(queue.isEmpty()); // Prints false
```

Lab 3A

Fishing



Problem statement summary

- Legend says that there are n fishes at the sea. The quality of the fishes are represented by an array of integer values. Note that fishes can have negative quality, as they may be rotten or dead. Fisherman John is trying to catch some fish from this array with his fishing net of size k . His fishing net allows him to catch a continuous subarray of fish from the array. The subarray is always of size k . The total value he gets from fishing is the sum of the qualities of fishes inside the subarray. Find out what is the maximum value that he can get from casting his net at most once.
- n fishes
- Fishing net of size k used to capture fishes; modelled as subarray of size k
- Quality of fish given as integer value
- Aim: maximise value in fishing net
- In other words: maximise sum of subarray of fixed size k

Input Output

- The first line are integers n and k . ($1 \leq n \leq 10^6$), ($k \leq n$) The next line will consist of n space separated 32-bit integers, which denotes the quality of each fish.
- $1 \leq k \leq n \leq 10^6$
- We may need to add up to 10^6 32-bit integers when computing the sum of values
- There is possibility of integer overflow; should use long data type to store their sum
- Output: an *integer* on a single line, the maximum value he can get, terminated by a newline.
- Note: *integer* here does not refer to the int data type

A procedure we need for all approaches

- we output 0 if our max sum is negative.
- we also need to constantly check our current sum of k numbers against our max sum so far;
- a possible way to update without if-else:

// initialise:

```
long maxSoFar = 0
```

// to compare and update:

```
maxSoFar = Math.max(maxSoFar, curSum)
```

First attempt

brute force:

- populate array with inputs; $O(n)$
- initialise maxSoFar = 0 to track max sum
- iterating through the array,

- retrieve and add up first k numbers starting from index 0;
- store as curSum;
- check curSum against maxSoFar; update maxSoFar if required;

- retrieve and add up next set of k numbers starting from index 1;
- store as curSum;
- check..

- .
- .
- .

- retrieve and add up final set of k numbers, starting from index $n-k$, ending at index $n-1$;
- check..

Analysis of brute force

- we will be retrieving and adding k numbers a total of $n-k+1$ times
 - time complexity: $O(k(n-k+1))$
- how to do better? identify unnecessary/repeated work
- observation: we are doing retrieving and adding k numbers each iteration, but the middle few numbers are repeated; only the 2 extreme values are changing each time
- attempts at improvement will be motivated by this observation; can we avoid recomputing sum of the middle few numbers?

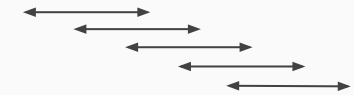
Approach 2: Sliding Window (Queue)

- much like a sliding window with a fixed window size;
 - view the first k numbers [1, -2, 5] in one window
 - 'slide' the window to the right to view [-2, 5, 7]
 - 'slide' the window to the right to view [5, 7, -1]
 - 'slide' the window to the right to view [7, -1, 8]
 - and so on..
-
- notice that the first number 1 is first to be removed from the window, while 5 will only be removed a few iterations later; FIFO !!
 - we can use a queue (fixed size k) to insert (from right) and remove values (from left)

Example input:

$n = 7, k = 3$

1 -2 5 7 -1 8 9



Example input:

$n = 7, k = 3$

1 -2 5 7 -1 8 9

Window:

[1, -2, 5]

[-2, 5, 7]

[5, 7, -1]

[7, -1, 8]

curSum = 4

curSum = $4 + 7 - 1 = 10$

curSum = $10 + (-1) - (-2) = 11$

curSum = $11 + 8 - 5 = 14$

- in each iteration/window:
 - to maintain current sum, simply add new value and subtract oldest value;
 - to maintain queue, add new value to queue and poll oldest value;
 - constant no. of operations compared to previous k additions !!
- work done in each iteration now independent of k
- total time $O(n-k)$
- $O(k)$ space for queue

Sliding Window in array

- sliding window is a popular technique for array/string optimisation problems;
 - string can be treated as an array of characters!
- can be done by working with array indices without the queue to achieve $O(1)$ space (other than the array)
- general strategy:
 - keep 2 variables for array indices, tracking the left(start) and right(end) of the window;
 - expand the window towards the right till some constraint is violated; then shrink the window from the left till window is valid again; then expand rightwards again.
 - in this case, the constraint is window size $\leq k$

Sample visualisation



initialisation: left = 0, right = 1
while (left < right && right < n) ...

first valid window: left = 0, right = 2
// do some work

invalid window: left = 0, right = 3

next valid window: left = 1, right = 3
// do some work

AND SO ON ...

final valid window: left = 4, right = 6
// do some work

termination: right = 7

*it is also possible to
view this as one step
ie. shift both pointers
together*

Analysis of sliding window (array)

- Implementation details:
 - Be careful not to let the 2 pointers go out of bounds
 - Window size = right - left + 1
 - After shifting right pointer, add value to curSum; before shifting left pointer, subtract value from curSum
- As long as constant work is done within each window, this algorithm is guaranteed to be $O(n)$ time.
- This is because in the procedure, the left and right pointer will never move left
- Every element in the array is then only ever visited at most twice (once by each pointer)

Approach 3: Prefix Sum

Example input:

$n = 7, k = 3$

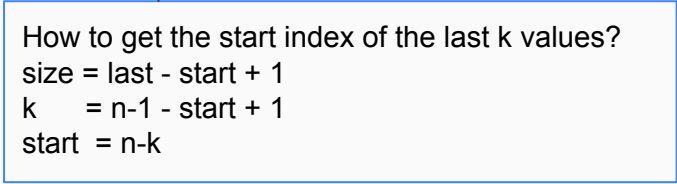
1 -2 5 7 -1 8 9

Index: 0 1 2 3 4 5 6

- let us define the prefix sum of index i , $pSum(i)$ = sum of array from index 0 to i inclusive;
- $pSum(0) = 1$; $pSum(1) = -1$; $pSum(2) = 4$; $pSum(3) = 11$; ...
- define $kSum(i)$ as sum of k continuous values starting from index i ;
 - aka $curSum$ starting from index i
- observe $curSum$ from index 1 to 3, $kSum(1) = 10 = pSum(3) - pSum(0)$
- generalising, we have $kSum(i) = pSum(i+k-1) - pSum(i-1)$
- but this relationship is not yet well-defined:
 - notice to compute $kSum(0)$, we would need $pSum(-1)$
 - to overcome this, we can
 - define $pSum(-1) = 0$, or
 - define $kSum(0) = pSum(i+k-1)$

Analysis and remarks

- we can precompute all prefix sums up to $pSum(n-1)$, storing them in an array
 - can be done in one for-loop $\Rightarrow O(n)$ time
- Subsequently, compute $kSum$ starting from $kSum(0)$ to $kSum(n-k)$
 - retrieve the 2 relevant $pSum$ and subtract them
 - $kSum(i) = pSum(i+k-1) - pSum(i-1)$
 - $O(1)$ in each iteration
- total time complexity: $O(n)$
- remark:
 - we have seen in previous labs the use of deferring operations (lazy increment, lazy reverse)
 - sometimes, pre-processing/precomputing certain values allow for more efficient solutions, though such insights may not be easy to spot



How to get the start index of the last k values?
 $size = last - start + 1$
 $k = n-1 - start + 1$
 $start = n-k$

Lab #3B

Entrepreneurship - Graded

Problem Statement - Summary

- Pizzas are sold in batches
- Orders come in batches, addition and deletion of orders occur in batches
- Deletion of batches of pizzas occur in LIFO order
- Orders are processed in the order they are read in (FIFO)
- Orders read in can be read from either left or right
- If there are insufficient pizzas left to fulfill an order, the entire order will not be processed

Inputs and Outputs

- N (number of commands), $1 \leq N \leq 10000$
- M (maximum of pizzas that can be sold), $0 \leq M \leq 2^{31}-1$
- ADD <Q> <D> (Q is the number of orders in the batch, D is the direction to read the orders in)
 - $1 \leq Q \leq 100$
 - Followed by $2Q$ space separated numbers of <P> <A>
 - P (number of pizzas), $1 \leq P \leq 2^{31}-1$
 - A (average pizza price), $0.0 < A < 100.0$
 - A has a maximum of 1 decimal place
 - P and A are always read in from left to right regardless of D
- CANCEL (B is the number of batches to cancel)
- Output: Total Revenue to 1 decimal place

Extraction of Keywords

- All integers can be stored using int type
 - No integer overflow will occur for the given ranges
- Parse commands to ADD or CANCEL **batches** of pizza
- When processing output, process by **individual orders**, only the individual order is not processed if there are insufficient pizzas, not the entire batch
- Output formatted correct to 1 decimal place
 - ```
pw.printf("%s\n",String.format("%.01f", revenue));
```

# Things to consider

- What data structure(s) can we use to keep track of both the batches of pizzas as well as individual orders?
  - Cancellation of pizzas are in LIFO order
  - Processing of orders are in FIFO order

Think: Stacks/Queues!

# Things to consider (cont. )

- We want to store individual orders within batches, and be able to delete batches in LIFO order in  $O(1)$  time
  - Think: Stack
- However, we also want to be able to access the batches in FIFO order, as we want to access the individual orders in FIFO order to calculate the revenue
  - Think: Queue
- Hence, what data structure(s) can we make use of to solve the problem?



# Approach

- A batch of pizzas is represented as a list of pairs (P, A)
  - You may create your own custom class to store each pair
- Store all batches of pizzas within an ArrayList
  - Supports fast deletion at the back of list, similar to Stack.pop()
  - Supports iteration from front of list to back of list to calculate output, similar to FIFO

# Alternative Approaches

- Other possible approaches?
  - LinkedList
  - Stack/Queue?
- Your approach?

# Lab #3C

Tasks



# Problem

To-do list that keeps track of a list of tasks for the day

The to-do list must supports these commands:

- Add Front **X**
- Add Back **X**
- Add Middle **X**
- Get **I**
- Remove Front
- Remove Back
- Remove Middle

There will be a total of  $n$  commands, where  $1 \leq n \leq 10^6$

# Example

## Commands

**ADD\_BACK A**

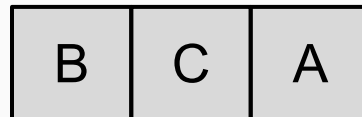
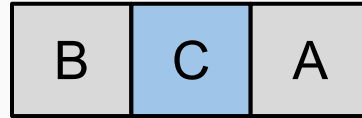
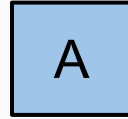
**ADD\_FRONT B**

**ADD\_MIDDLE C**

**ADD\_MIDDLE D**

**REMOVE\_MIDDLE**

## To-do list



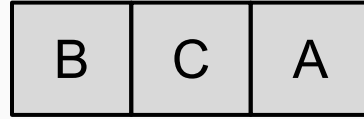
# Example

## Commands

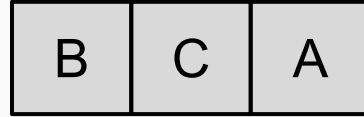
**GET 0**

**GET 1**

## To-do list



Output: **B**



Output: **C**

# Approach 1: Single Deque

## Approach 1: Using Java's Linked List to simulate a deque

- We can use Java's LinkedList API to satisfy all the commands

| LinkedList Method               | Operation satisfied  | Time Complexity |
|---------------------------------|----------------------|-----------------|
| <i>addLast(String X)</i>        | <b>ADD_BACK X</b>    | O(1)            |
| <i>addFirst(String X)</i>       | <b>ADD_FRONT X</b>   | O(1)            |
| <i>removeFirst()</i>            | <b>REMOVE_FRONT</b>  | O(1)            |
| <i>removeLast()</i>             | <b>REMOVE_BACK</b>   | O(1)            |
| <i>add(int index, String X)</i> | <b>ADD_MIDDLE X</b>  | O(N)            |
| <i>remove(int index)</i>        | <b>REMOVE_MIDDLE</b> | O(N)            |
| <i>get(int index)</i>           | <b>GET</b>           | O(N)            |

# Approach 1: Single Deque

**Approach 1:** Using Java's Linked List to simulate a deque

## Analysis

- The **FRONT** and **BACK** operations can run in  $O(1)$  time, but the **MIDDLE** operations and **GET** operation take  $O(N)$  time
- Our worst case time complexity will therefore be  $O(N^2)$
- Since we can have up to  $10^6$  commands, this will mean we will have up to  $10^{12}$  operations
- Recall: A computer can do  $\sim 10^8$  operations per second
  - This approach will definitely **exceed the time limit**



## Approach 2: 2 Deques

**Approach 2:** Using 2 deques to store the tasks

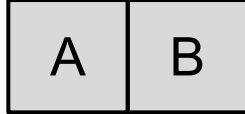
- To support **ADD/REMOVE\_MIDDLE** operations in  $O(1)$  time, we can use 2 LinkedLists to simulate 2 deques
- The 1st deque will store the first half of the to-do list, the 2nd deque will store the second half of the list
- After each command, the 2 deques will have to balance out to maintain appropriate sizes
  - i.e. Both deques must always be approximately the same size

## Approach 2: 2 Deques

**Approach 2:** Using 2 deques to store the tasks

To-do list: **[ A, B, C, D ]**

Deque 1



Deque 2



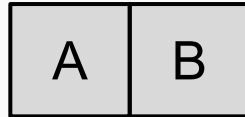
Command: **ADD\_BACK E**

## Approach 2: 2 Deques

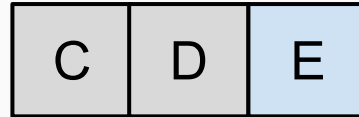
**Approach 2:** Using 2 deques to store the tasks

To-do list: [A, B, C, D, **E**]

Deque 1



Deque 2



*deque2.addLast(E)*

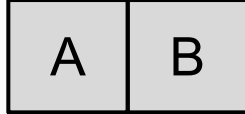
Command: **ADD\_BACK E**

## Approach 2: 2 Deques

**Approach 2:** Using 2 deques to store the tasks

To-do list: **[ A, B, C, D, E ]**

Deque 1



Deque 2



Command: **REMOVE\_MIDDLE**

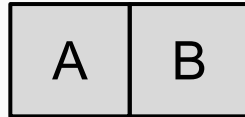
*Note: We will always remove from the 2nd deque*

## Approach 2: 2 Deques

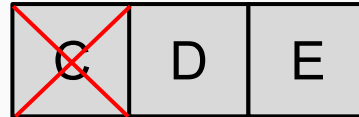
**Approach 2:** Using 2 deques to store the tasks

To-do list: [A, B, ~~C~~, D, E]

Deque 1



Deque 2



`deque2.removeFirst()`

Command: **REMOVE\_MIDDLE**

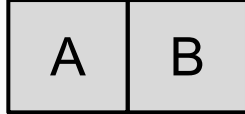
*Note: We will always remove from the 2nd deque*

## Approach 2: 2 Deques

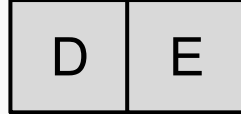
**Approach 2:** Using 2 deques to store the tasks

To-do list: **[ A, B, D, E ]**

Deque 1



Deque 2



Command: **REMOVE\_MIDDLE**

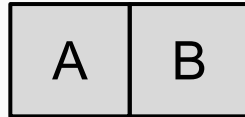
*Note: We will always remove from the 2nd deque*

## Approach 2: 2 Deques

**Approach 2:** Using 2 deques to store the tasks

To-do list: [ A, B, ~~D~~, E ]

Deque 1



Deque 2



`deque2.removeFirst()`

Command: **REMOVE\_MIDDLE**

*Note: We will always remove from the 2nd deque*

## Approach 2: 2 Deques

**Approach 2:** Using 2 deques to store the tasks

To-do list: **[ A, B, E ]**



Command: **REMOVE\_MIDDLE**

- We want to maintain `deque2.size() >= deque1.size()`
- We need to transfer 1 task from deque1 to deque2

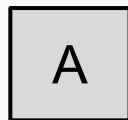


## Approach 2: 2 Deques

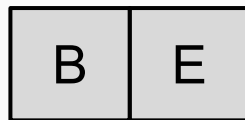
**Approach 2:** Using 2 deques to store the tasks

To-do list: **[ A, B, E ]**

Deque 1



Deque 2



Command: **REMOVE\_MIDDLE**

- We want to maintain `deque2.size() >= deque1.size()`
- We need to transfer 1 task from deque1 to deque2

## Approach 2: 2 Deques

### **Approach 2:** Using 2 deques to store the tasks

- We always need to ensure that the 2nd deque's size is always equal to the 1st deque or larger by 1
  - This means that we'll need to balance the deques accordingly for each ADD/REMOVE operation as well
- **However**, the **GET** operation still runs in  $O(N)$  time if we use Java's LinkedList :(

## Approach 2: 2 Deques

### **Approach 2:** Using 2 deques to store the tasks

- To achieve  $O(1)$  time for the **GET** command, we need to consider the underlying data structure used to implement the deques
  - What data structure allows random access in  $O(1)$  time?

### **Array**

- Java has an ArrayDeque class that uses arrays as the underlying data structure to implement the Deque interface
  - However, this class does not allow random access

## Approach 2: 2 Deques

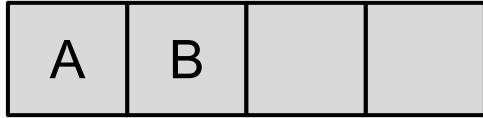
### **Approach 2:** Using 2 deques to store the tasks

- We therefore have to implement our own Deque class that allows the following operations in  $O(1)$  time:
  - Adding/removing from the front
  - Adding/removing from the back
  - Random access by index
- We use arrays as the underlying data structure

## Approach 2: 2 Deques

**Approach 2:** Using 2 deques to store the tasks

- Self-implemented Array Deque



Front: 0

Back: 2

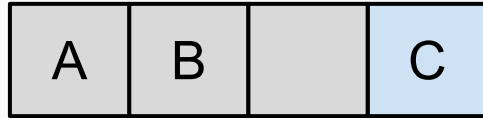
Capacity: 4

***addFirst('C')***

## Approach 2: 2 Deques

### Approach 2: Using 2 deques to store the tasks

- Self-implemented Array Deque



Front: -1

Back: 2

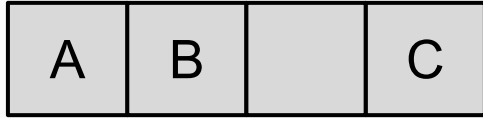
Capacity: 4

***addFirst('C')***

## Approach 2: 2 Deques

**Approach 2:** Using 2 deques to store the tasks

- Self-implemented Array Deque



Front: -1

Back: 2

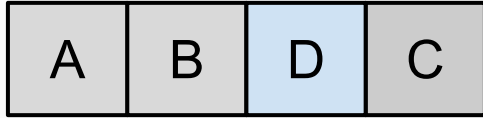
Capacity: 4

***addBack('D')***

## Approach 2: 2 Deques

### Approach 2: Using 2 deques to store the tasks

- Self-implemented Array Deque



Front: -1

Back: 3

Capacity: 4

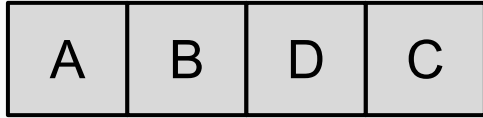
***addBack('D')***



## Approach 2: 2 Deques

### Approach 2: Using 2 deques to store the tasks

- Self-implemented Array Deque



Front: -1

Back: 3

Capacity: 4

***addFront('E')***

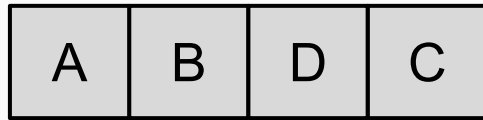
We need to enlarge the array first

## Approach 2: 2 Deques

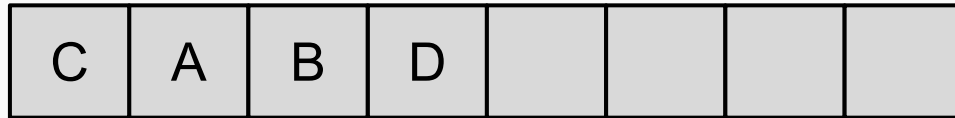
### Approach 2: Using 2 deques to store the tasks

- Self-implemented Array Deque

***enlargeArr()***



Front: -1  
Back: 3  
Capacity: 4

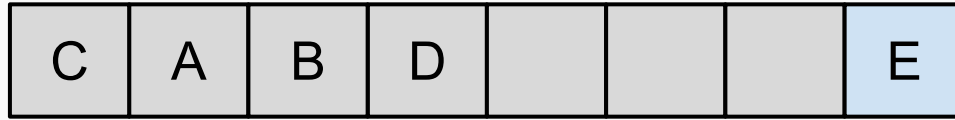


Front: 0  
Back: 4  
Capacity: 8

## Approach 2: 2 Deques

### Approach 2: Using 2 deques to store the tasks

- Self-implemented Array Deque



Front: -1

Back: 4

Capacity: 8

***addFront('E')***

## Approach 2: 2 Deques

### **Approach 2:** Using 2 deques to store the tasks

- Using 2 custom Array Deques, we can therefore achieve  $O(1)$  time complexity for all commands
- Question: Enlarging the array seems to take  $O(N)$  time, why do we still consider it to be  $O(1)$  time here?
  - Constant amortized time
  - The operation only happens “once in awhile” and averages out to be  $O(1)$  time

# Lab #3C Demo