CS2040S

Tutorial 3: Lists, Stacks and Queues

Nicholas **Russell** Saerang (russellsaerang@u.nus.edu)

Let's start with...

NO FREE LUNCH

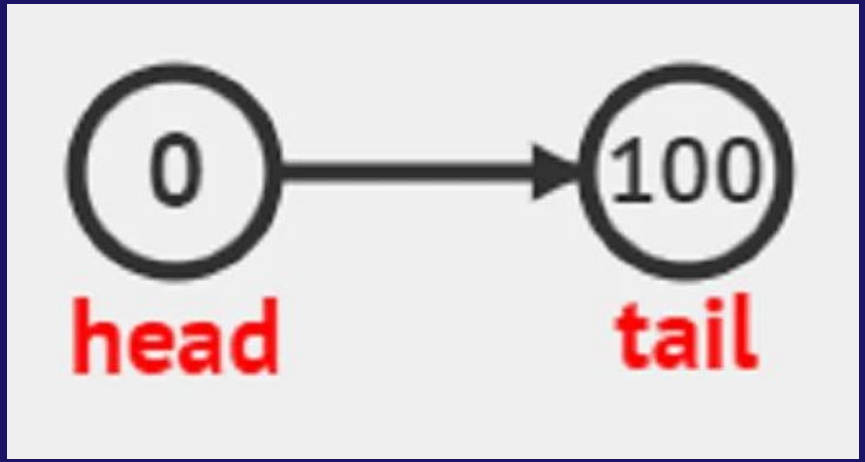
:D

LINEAR DATA STRUCTURES

Lists

There are two implementations of lists taught during lecture.

- Array implementation
- Linked list implementation



Lists: Array Implementation

Time complexity of the different list operations

- Retrieval: *getItemAtIndex(int i), getFirst(), getLast()*
  - O(1) – indexing into an array is constant time due to random access memory of the computer
- Insertion: *addItemAtIndex(int i, int item), addFront(), addBack()*
  - Best case = O(1) – if adding at the back and no need to enlarge array
  - Worst case = O(n) – if adding to the front due to shifting all item to the right or adding to the back but need to enlarge the array so have to perform copying of n items to new array (is this realistic?)
  - Average case = O(n) – on average need to shift ½(n) items to the right
- Deletion: *removeItemAtIndex(int i), removeFront(), removeBack()*
  - Best case = O(1) – if removing from the back
  - Worst case = O(n) – if removing from the front due to shifting all items to the left
  - Average case = O(n) – on average need to shift ½(n) items to the left

Lists: LL Implementation

Time complexity of the different list operations

- Retrieval: *getItemAtIndex(int i), getFirst(), getLast()*
  - Best case = O(1) – accessing the first node, return the head
  - Worst case = O(n) – accessing the last node, since you need to move all the way to the back from the head (n moves)
  - Average case = O(n) – need to move about half way through the list to access any node on average so ½(n) iterations of the for loop
- Insertion: *addItemAtIndex(int i, int item), addFront(), addBack()*
  - Best case = O(1) – if adding at the front (don't have to worry about enlarging the list unlike array)
  - Worst case = O(n) – if adding to the back due to having to move all the way to the back from the head (n moves)
  - Average case = O(n) – on average need to make ½(n) moves

Lists: LL Implementation

- Deletion: *removeItemAtIndex(int i), removeFront(), removeBack()*
  - Best case = O(1) – if removing from the front
  - Worst case = O(n) – if removing from the back, again due to moving all the way to the back from the head
  - Average case = O(n) – on average need to make ½(n) moves

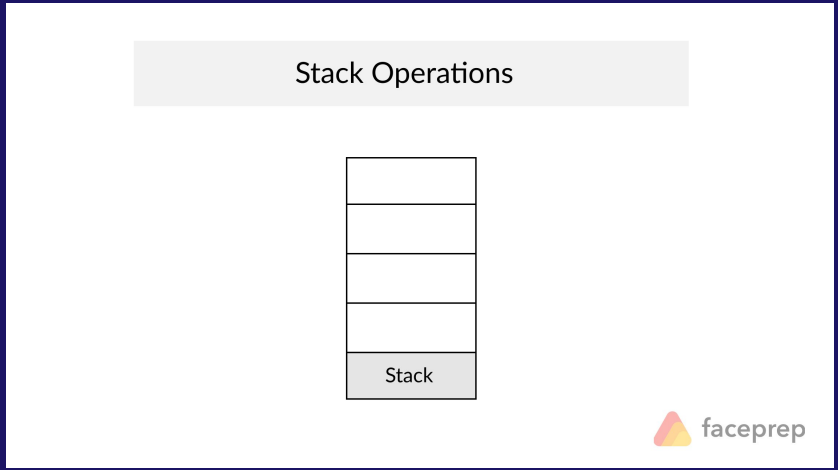
Take note that the complexities can be improved depending on the variant of the linked list, e.g. tailed linked list has O(1) runtime for getLast().

Stack and Queue

A stack is a data structure with the LIFO (Last In, First Out) property.

A queue is a data structure with the FIFO (First In, First Out) property.

Check animation here: https://atechdaily.com/posts/stack-explained-with-example-in-python



01

TRUE OR FALSE?

Question 1a

Deletion in any Linked List can always be done in O(1) time.

Answer:
False. Deletion only O(1) at head. O(n) otherwise.

Question 1b

A search operation in a Doubly Linked List will only take O(log n) time.

Answer:
False. Search is always O(n).

What if the elements in the doubly link list is sorted?

Still O(n)!

Even if it is sorted, we cannot search in O(log n) time using binary search since you cannot directly access a node at a particular index in O(1) time unlike the case of an array.

Question 1c

All operations in a stack are O(1) time when implemented using an array.

Answer:
False. On average, insertion is O(1) time, but in the worst case individual insertions can be O(n) time due to resizing of the array.

Note that if we consider amortization, we can prove that a queue implemented with an array has worst case amortized O(1) time complexity for insertion.

For this course we will use the amortized time complexity rather than the worst case time complexity for array-based implementation of list/stack/queue.

Question 1d

A stack can be implemented with a Singly Linked List with no tail reference with O(1) time for all operations.

Answer:
True. Insertion and deletion only required to be done at the head of the linked list.

Question 1e

All operations in a queue are O(1) time when implemented using a Doubly Linked List with no modification.

Answer:
True. Doubly linked list by default have tail reference.
Tail reference required to do insertion to the back in O(1) time.

Note: Some can argue based on other references or books that not having a tail is default, but we will go along with having a tail as default for this course.

In midterms and finals, you can make your own assumption (state it clearly) and answer based on your assumption. If your assumption is valid and not contradicting the question, your answer will be accepted.

Question 1f

Three items A, B, C are inserted (in this order) into an unknown data structure X.

If the first element removed from X is B, X can be a queue.

Answer:
False. First element removed should be A if X is a queue.
Queue is FIFO (First In First Out).

02

CIRCULAR LINKED LIST

Circular Linked List

Implement a method swap(int index) in the CircularLinkedList class below to **swap the node at the given index with the next node**. The ListNode class (as given in the lectures) contains an integer value.

Constraints:

- Index ≥ 0
- If the index is larger than the size of the list, then the index wraps around.
- CANNOT create new nodes
- CANNOT modify element in any node

Objective: modify the pointers

```
class CircularLinkedList {

    public int size;
    public ListNode head;
    public ListNode tail;

    public void addFirst(int element) {
        size++;
        head = new ListNode(element, head);
        if (tail == null)
            tail = head;
        tail.setNext(head);
    }

    public void swap(int index) { ... }
}
```

Circular Linked List

- **Case 1: size < 2**
  Do nothing since there is nothing to swap!

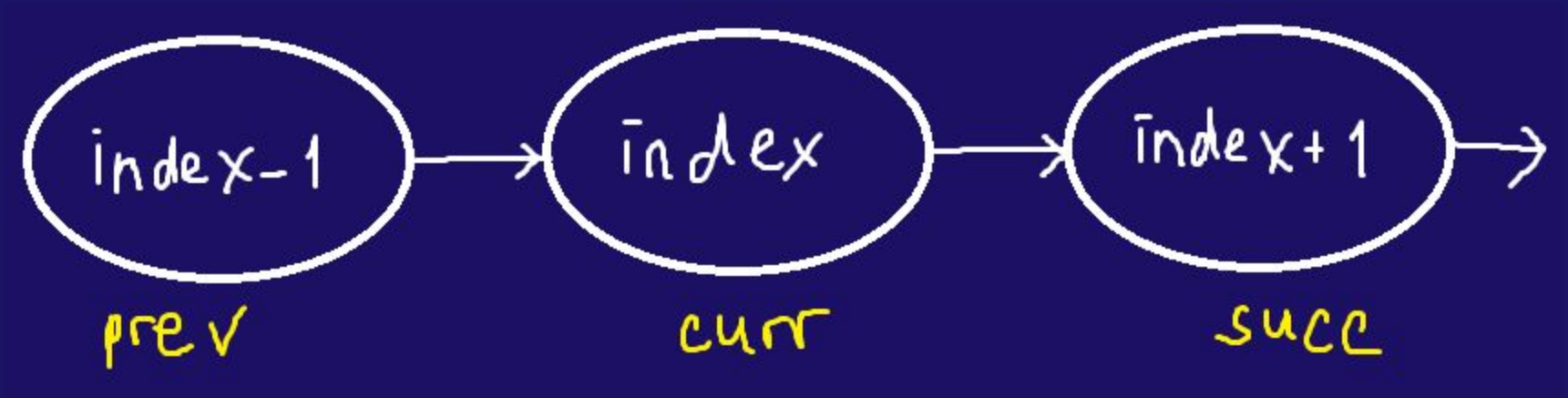
- **Case 2: size == 2**
  Simply swap head and tail.

- **Case 3: size > 2**
  Use modulo operation to reduce the index (we don't have to iterate through the entire linked list multiple times).

  Note that the order of changing the references matters as you may lose nodes if the references are not changed properly.

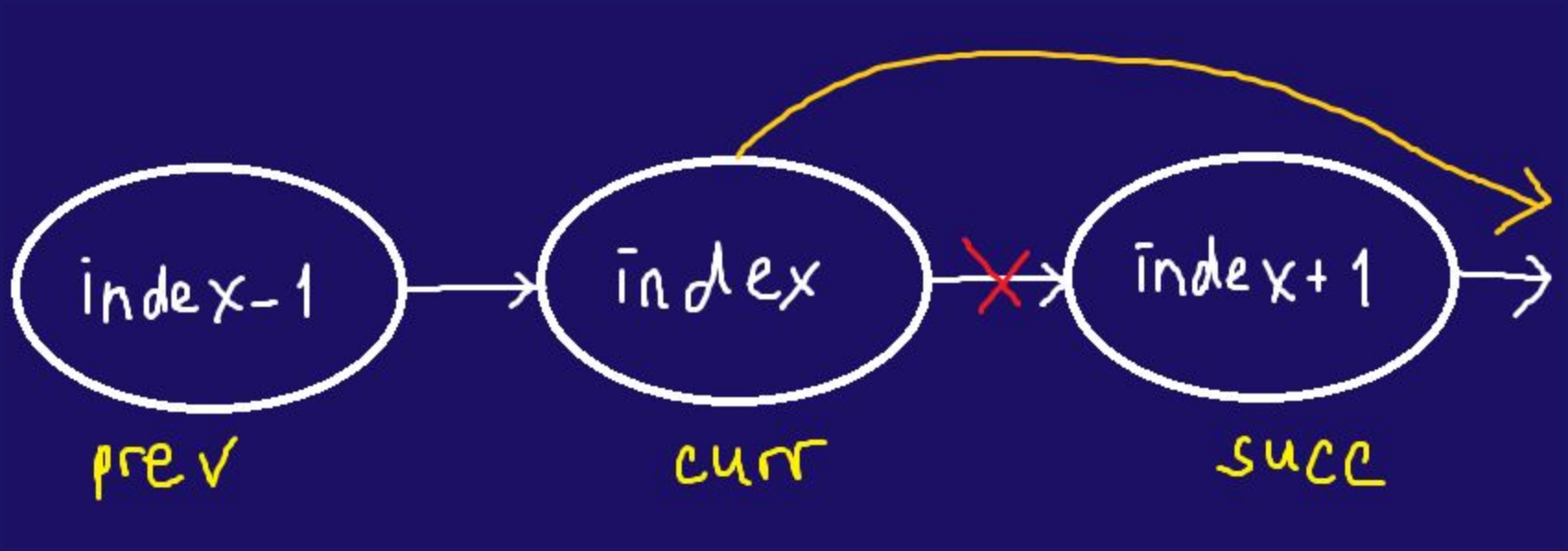
```
curr.setNext(succ.getNext());
succ.setNext(curr);
prev.setNext(succ);
```

Circular Linked List



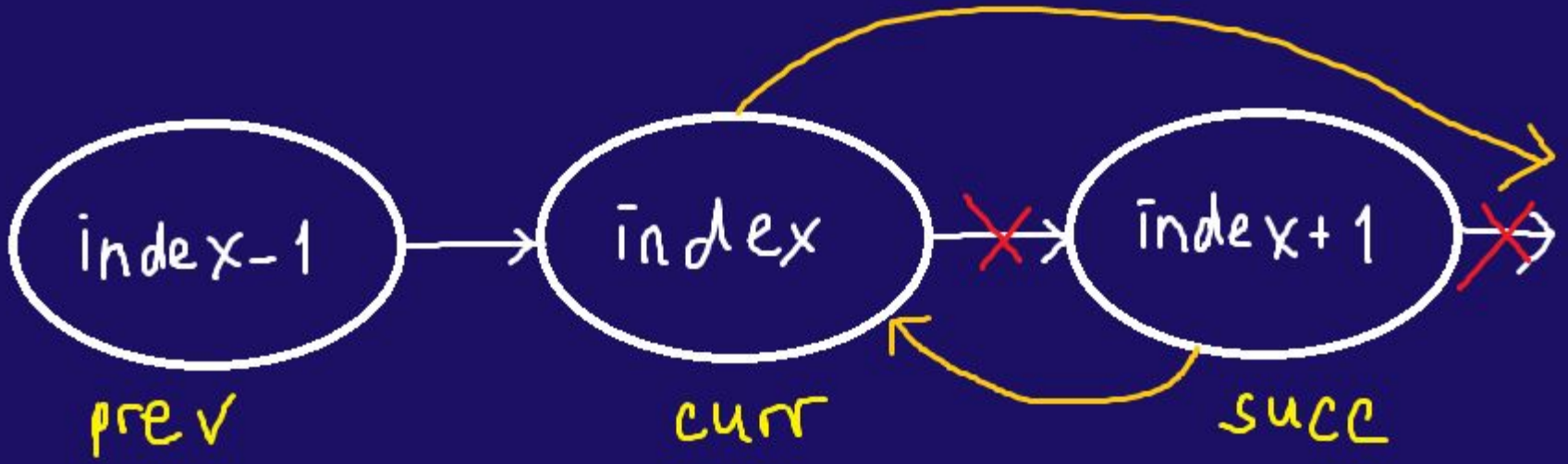

```
curr.setNext(succ.getNext());
succ.setNext(curr);
prev.setNext(succ);
```

Circular Linked List



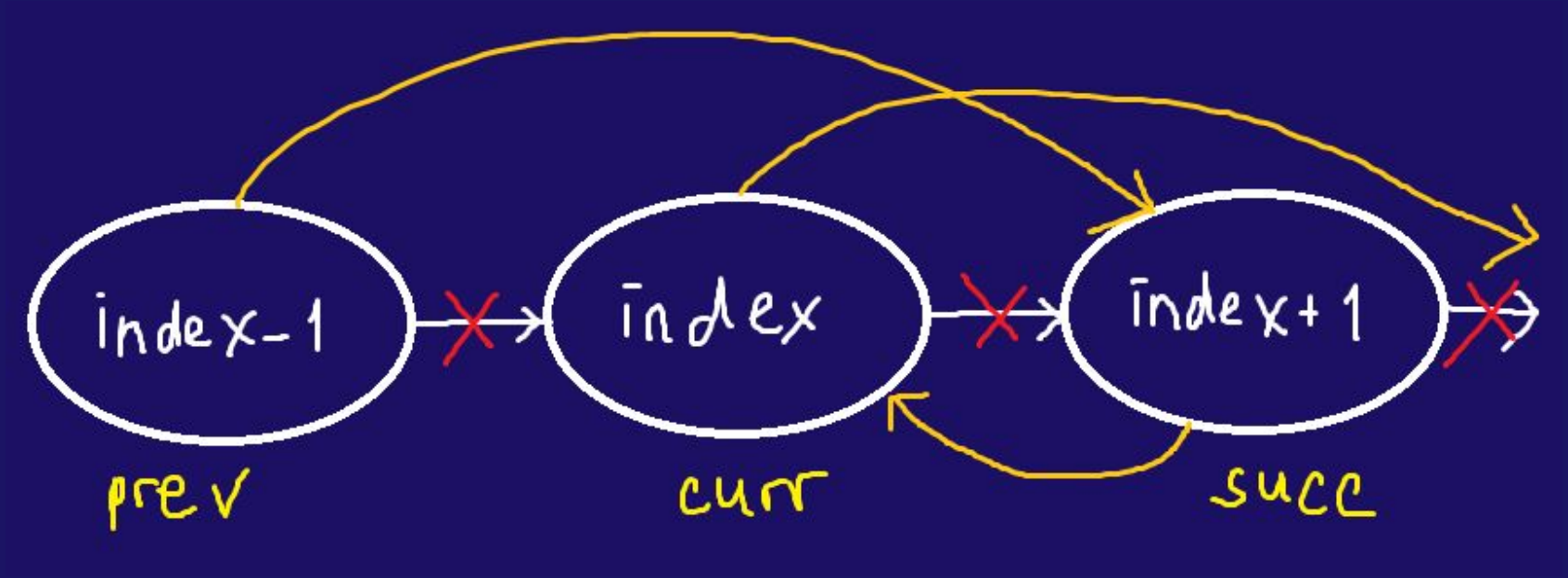

```
curr.setNext(succ.getNext());
succ.setNext(curr);
prev.setNext(succ);
```

Circular Linked List



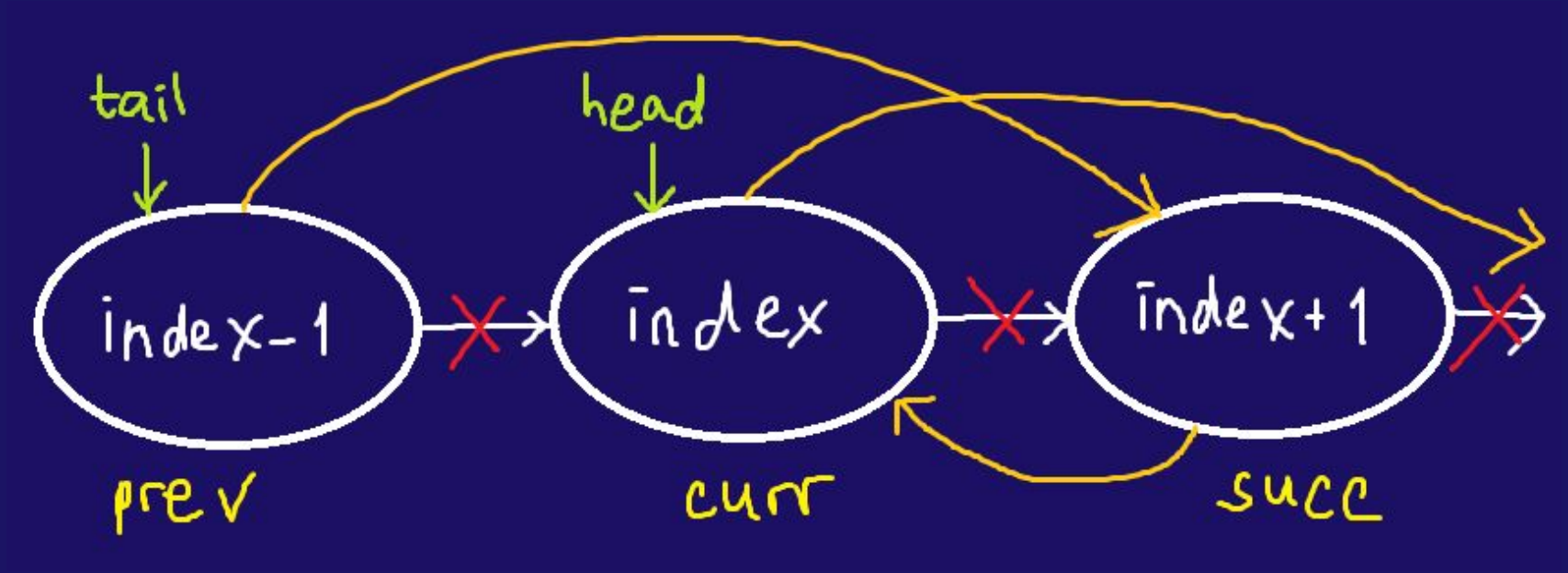

```
curr.setNext(succ.getNext());
succ.setNext(curr);
prev.setNext(succ);
```

Circular Linked List



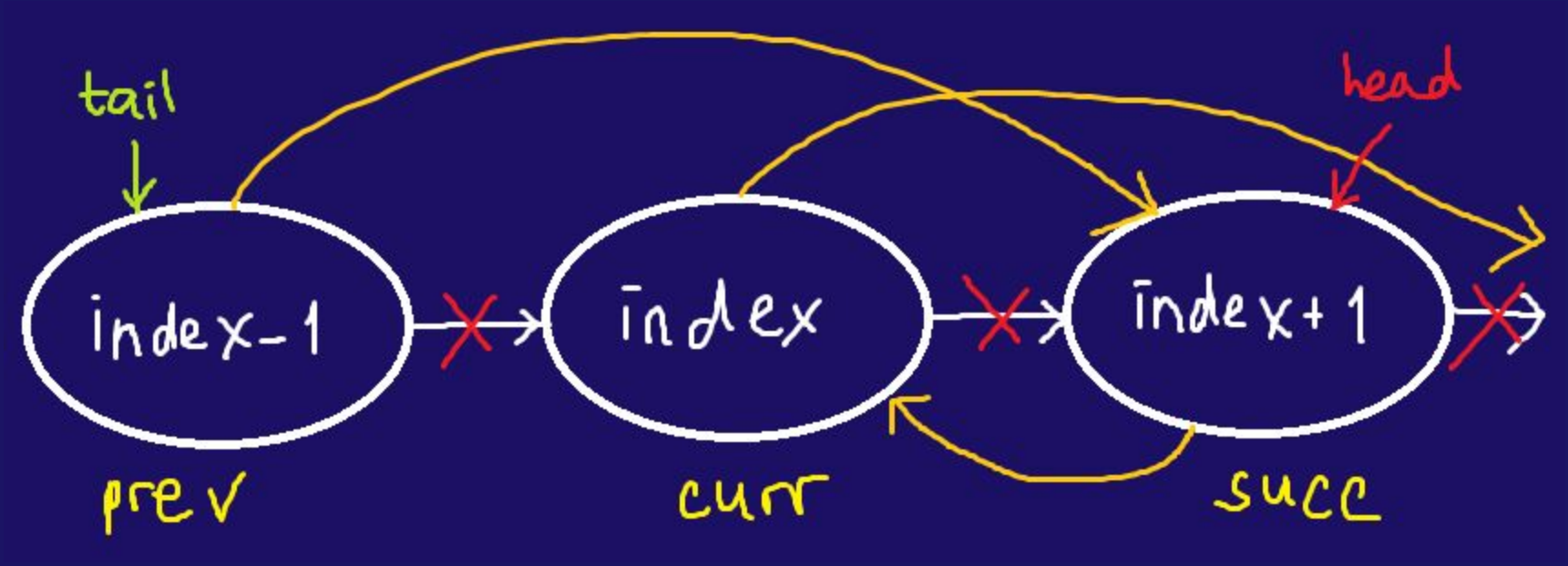

```
curr.setNext(succ.getNext());
succ.setNext(curr);
prev.setNext(succ);
```

Edge Cases



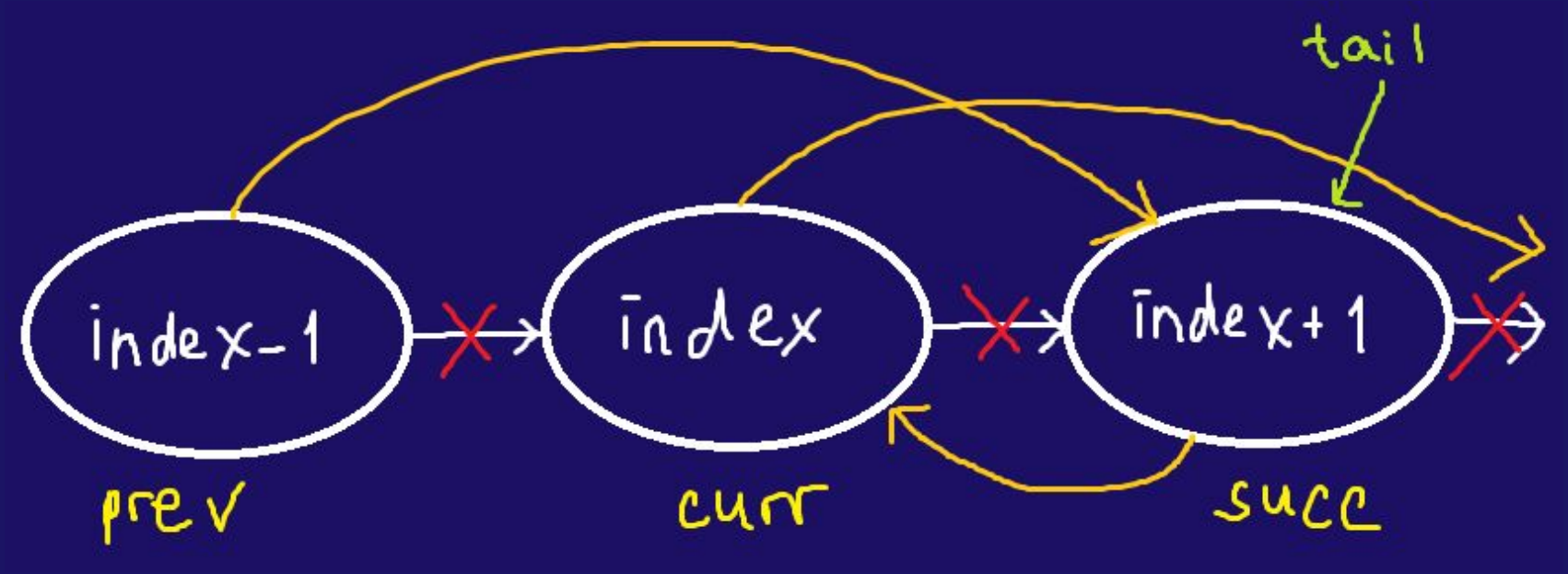
What if index == 0?
What if index == size - 2?
What if index == size - 1?

Edge Cases



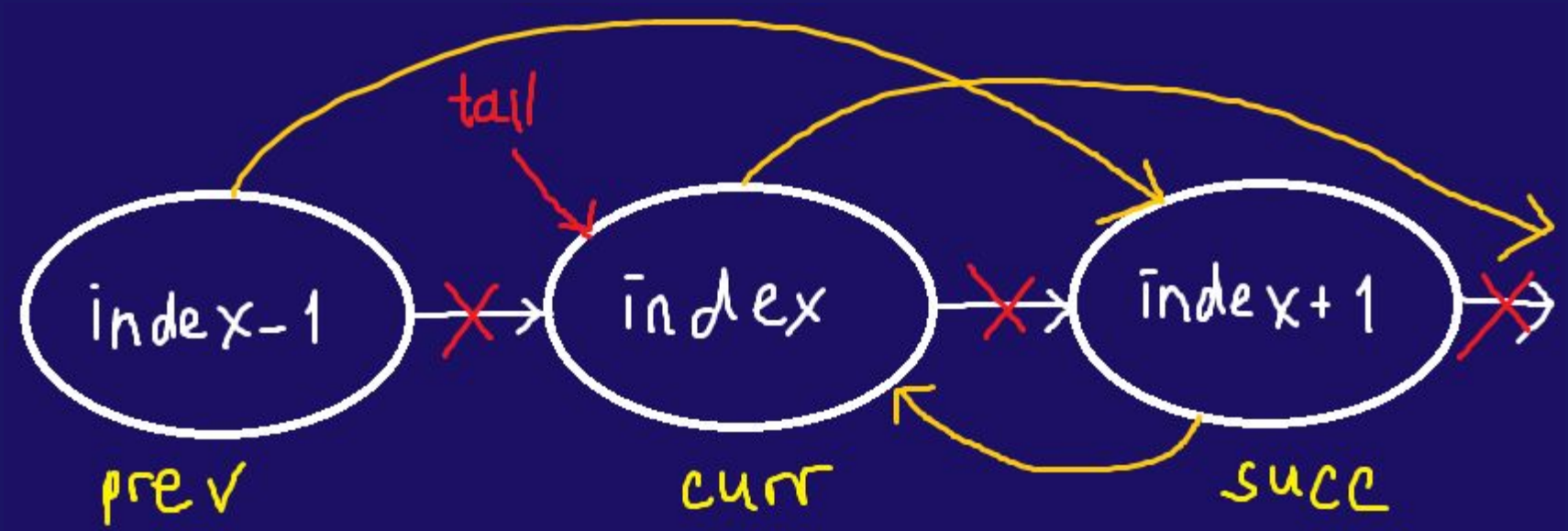
What if index == 0? succ is the new head!
What if index == size - 2?
What if index == size - 1?

Edge Cases



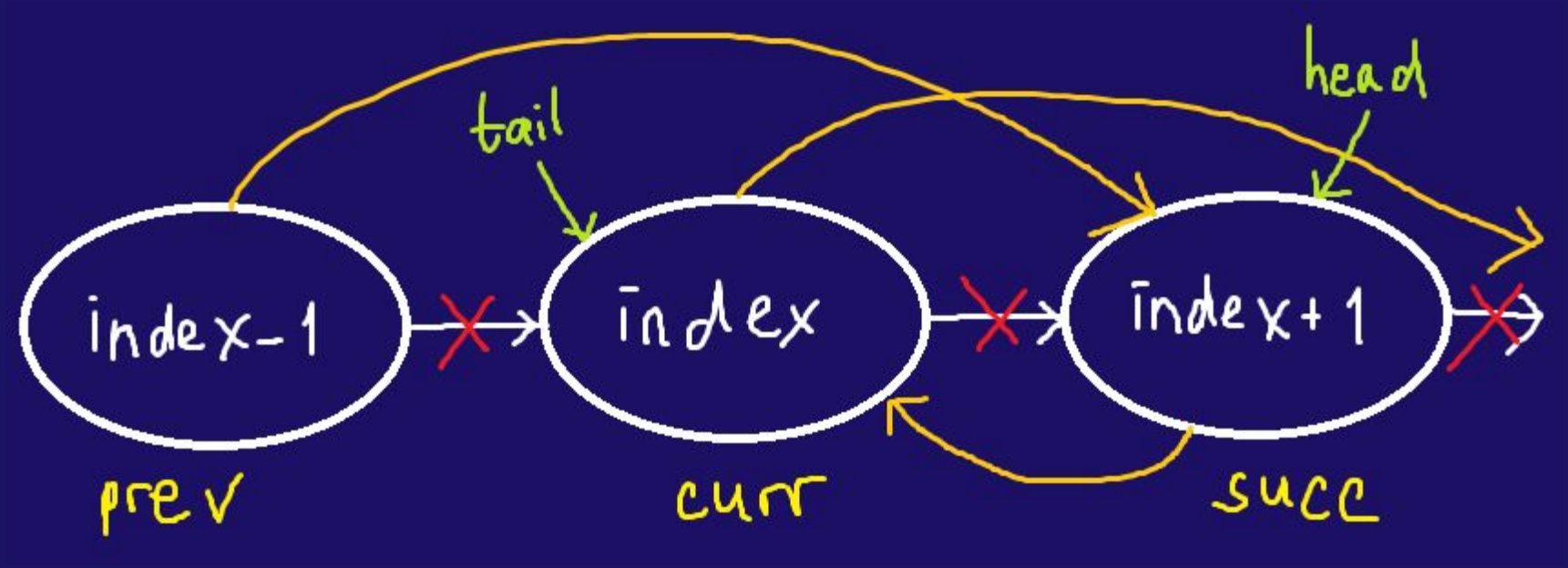
What if index == 0?
What if index == size - 2?
What if index == size - 1?

Edge Cases



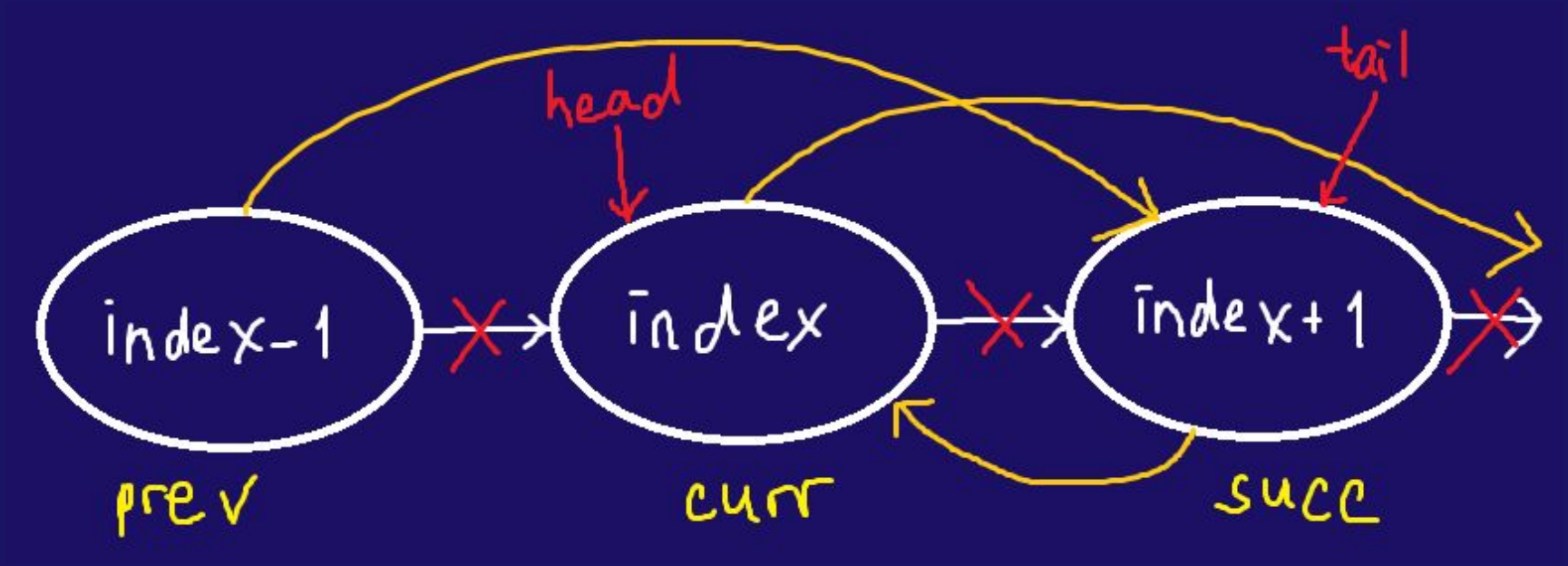
What if index == 0?
What if index == size - 2? curr is the new tail!
What if index == size - 1?

Edge Cases



What if index == 0?
What if index == size - 2?
What if index == size - 1?

Edge Cases



What if index == 0?
What if index == size - 2?
What if index == size - 1? Swap head and tail!

03

WAITING QUEUE

Waiting Queue

Abridged problem description:
We have a queue that is implemented using array. Implement a function `leave(String personName)` that allows a person with name personName to leave the queue at any time.

Give at least 2 ways and state the time complexity.

Waiting Queue

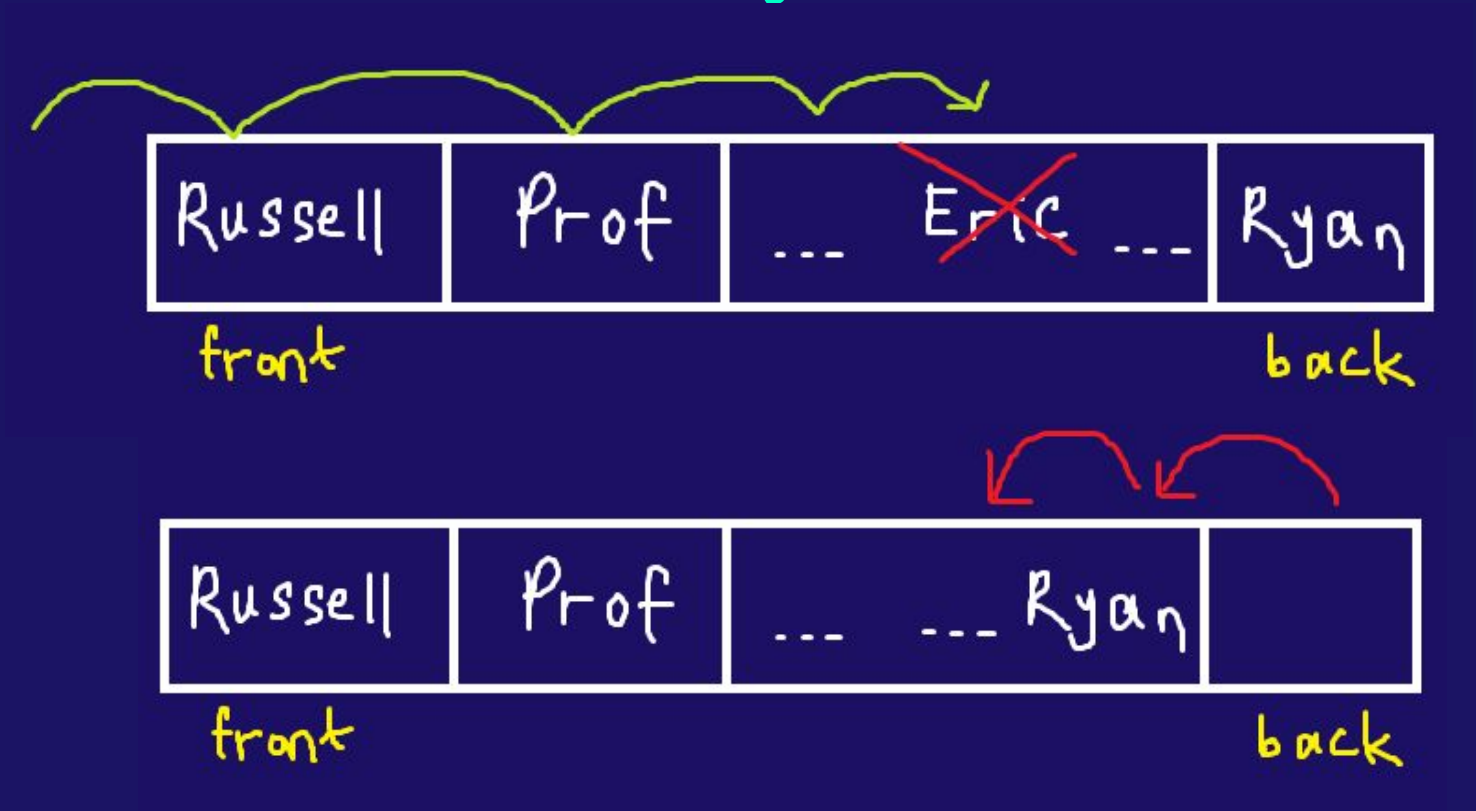
For all solutions, we need to maintain 2 variables front and back that denotes the start and end of the queue.

Solution 1:
Iterate from front to back.

If personName is found at index i, shift all elements after index i to the left by 1.

Time complexity: O(n)

Waiting Queue



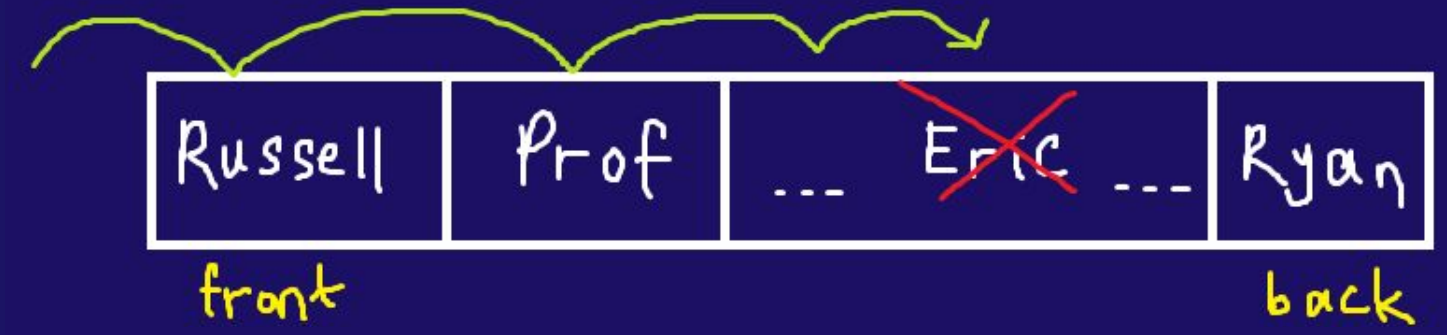
Waiting Queue

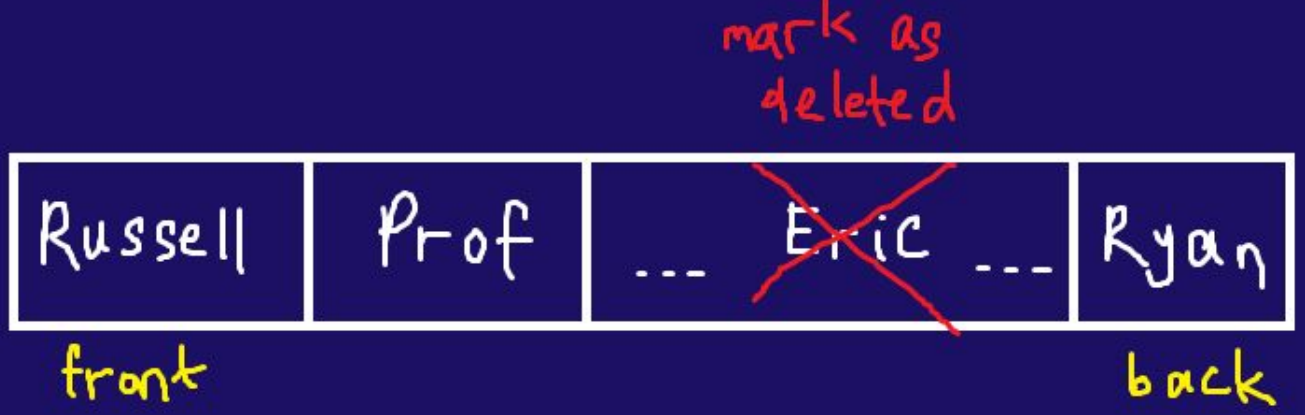
Solution 2: Lazy deletion
Each element in the queue has a Boolean flag to indicate whether a person has left the queue or not. (we could also make a Person class with flag as one of its attributes)

The time complexity of the leave operation is still O(n), since we need to iterate from front to back to find the person and flag it.

However, the dequeue operation now could run in O(n) when everyone left the queue already but we need to dequeue the entire queue.

Waiting Queue





Waiting Queue

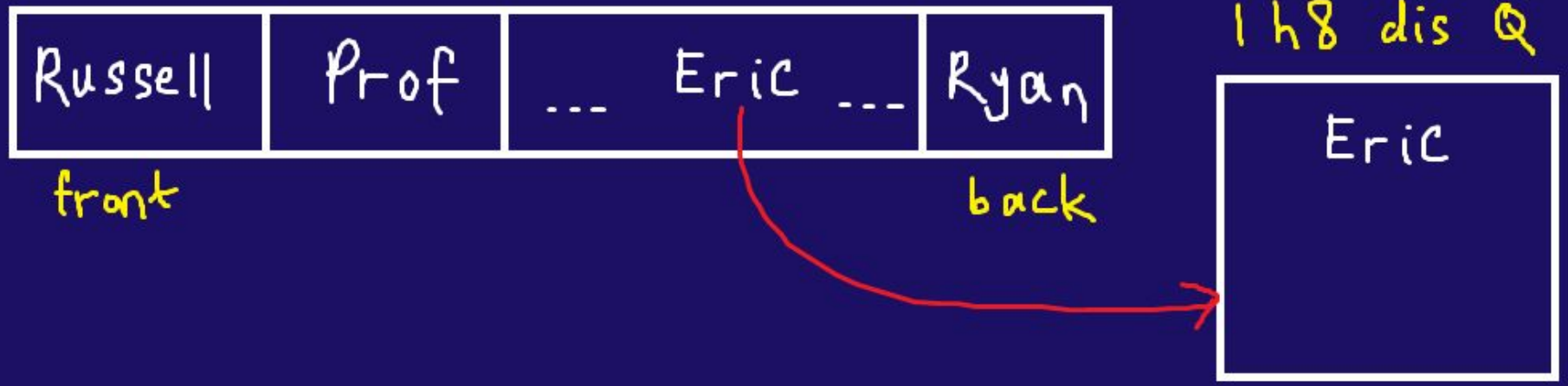
Solution 3:
Store the names of the people who want to leave the queue in a separate data structure (hash table).

When a person is served, the collection is searched to find a matching person.

The efficiency of leave is improved to O(1), but the method requires more space (extra hash table!).

dequeue also deteriorates to O(n), as we may have to remove multiple elements until we find someone who has not already left the queue.

Waiting Queue



04

EXPRESSION EVALUATION

Expression Evaluation

Abridged problem description:
There are only 4 operators +, −, *, / and the following rules apply:

- (+ a b c) returns the sum of all the operands, and (+) returns 0.
- (− a b c) returns a − b − c − ... and (− a) returns 0 − a. The minus operator must have at least one operand.
- (* a b c) returns the product of all the operands, and (*) returns 1.
- (/ a b c) returns a / b / c / ... and (/ a) returns 1 / a, using double division. The divide operator must have at least one operand.

These are called Lisp expressions.

Expression Evaluation

Design and implement an algorithm that uses stacks to evaluate a legal Lisp expression with n tokens, each token separated by a space, all inputs are valid, no division by 0. Output the result, which will be one double value.

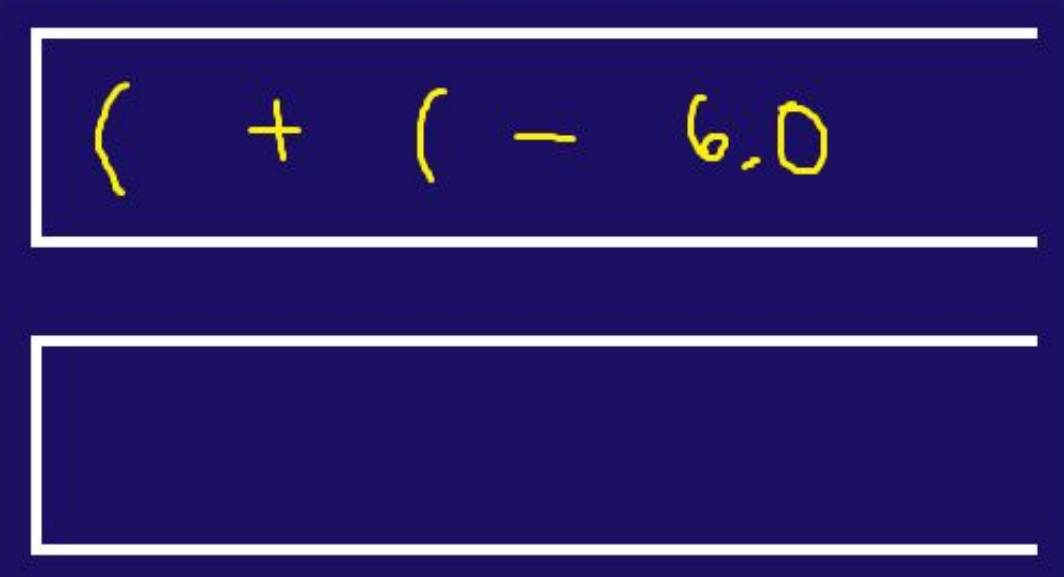
Expression Evaluation

The algorithm requires 2 stacks.

1. We start by pushing the tokens one by one into the first stack until we see the first “)”.
2. We then pop tokens in the first stack and push them into the second stack one by one until we pop the element “(”.
3. Now in the second stack, the operator is the first tokens to be removed followed by the tokens to be operated on, and we can remove the tokens inside one by one and evaluate the expression in the same order as they were given in the input. The result of the expression is pushed back into the first stack.
4. We repeat the above steps until all tokens are processed, and the final answer will be the one remaining value inside the first stack.
5. The time complexity of the algorithm is O(n), because each token will only be added or removed from a stack not more than 4 times.

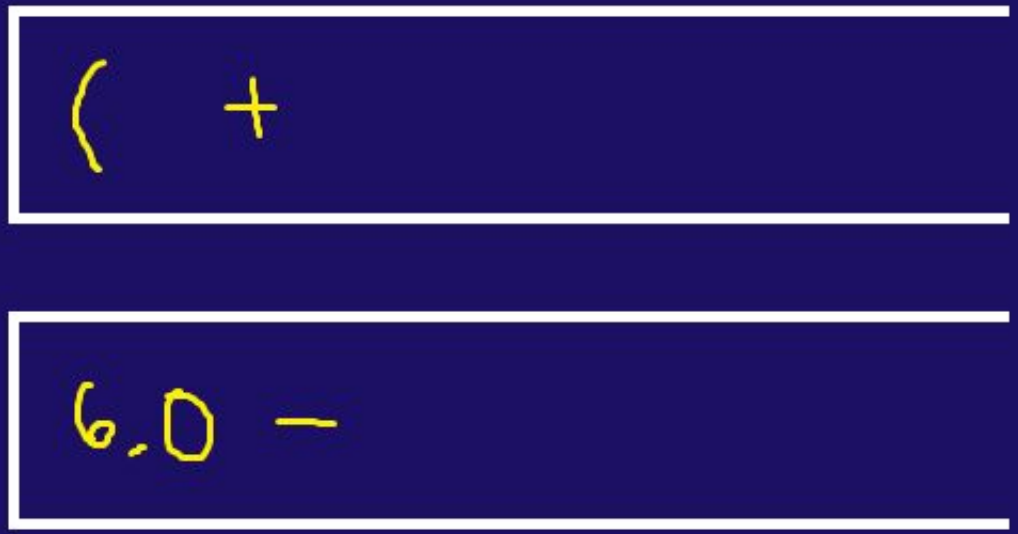
(+ (- 6) (* 2 3 4))

The main stack pushes the tokens one by one until it reads “)”.



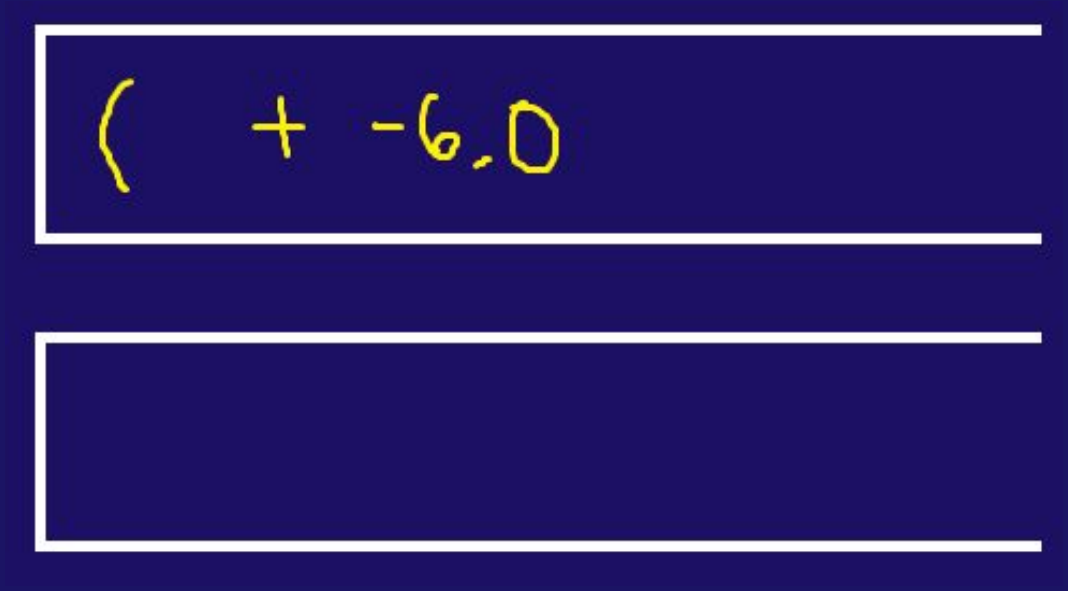
(+ (- 6) (* 2 3 4))

The main stack transfers its tokens to the temporary stack for evaluation.



(+ (- 6) (* 2 3 4))

The temporary stack pushes back the result after performing subtraction.



(+ (- 6) (* 2 3 4))

Main stack continues to push tokens until it reads “)”.

(+ (- 6) (* 2 3 4))

Main stack transfers tokens to temporary stack one by one.

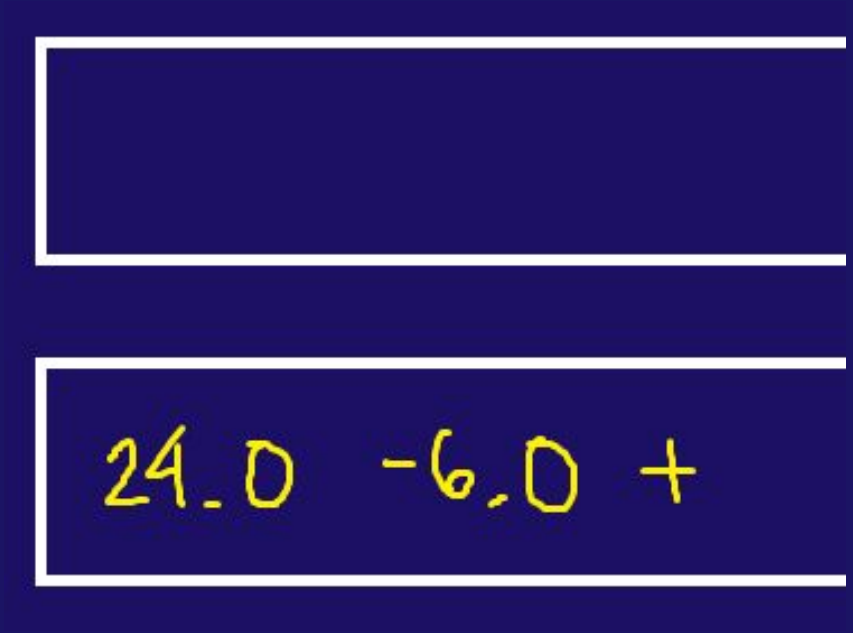
(+ (- 6) (* 2 3 4))

Temporary stack pushes back the result after calculation.

(+ (- 6) (* 2 3 4))

Main stack pushes until “)”. No change in diagram.

Main stack pushes until “)”.

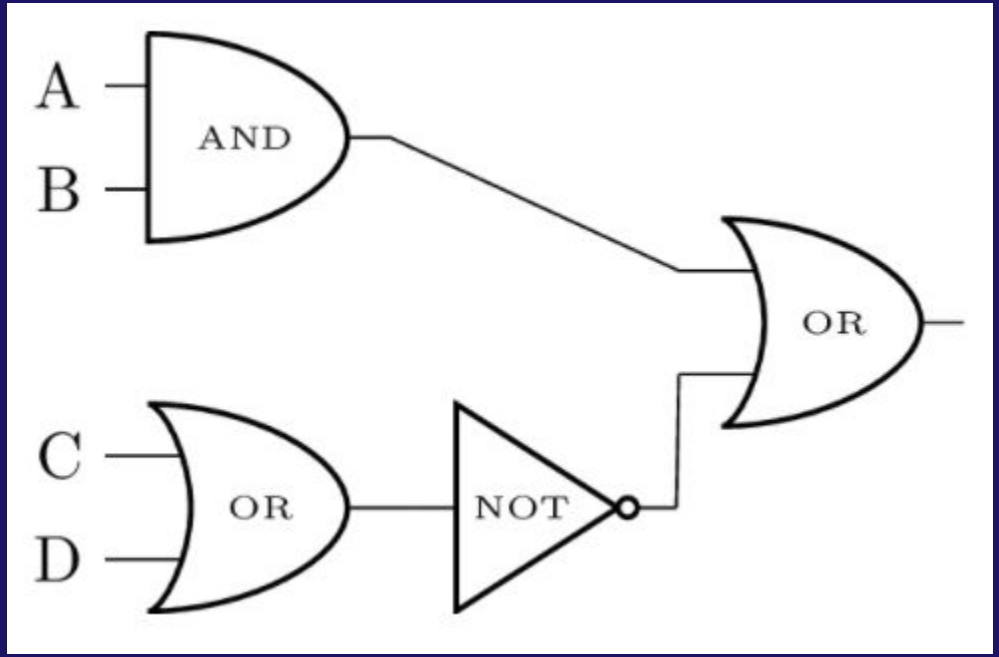


(+ (- 6) (* 2 3 4))

Temporary stack pushes back final result.

Free Kattis Problem

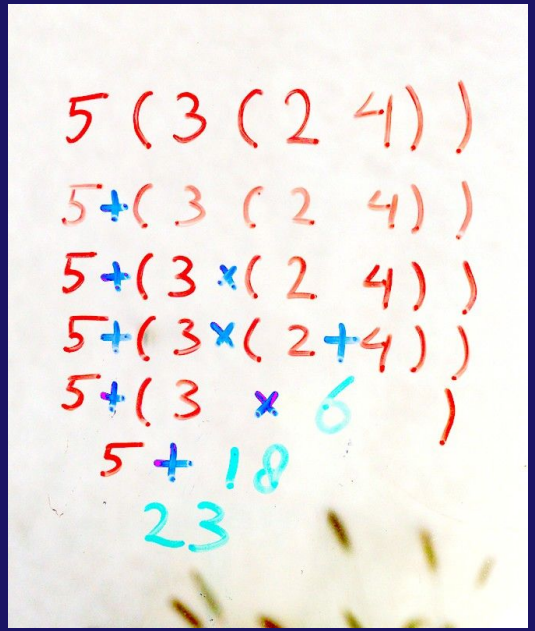
https://nus.kattis.com/problems/circuitmath



Free Kattis Problem #2

(Slighly Harder)

https://nus.kattis.com/problems/bracketsequence



THE END!