Data Structures (I)

Christy Cheng {christycty}
2022-03-12

香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

Outline

- ❏ Introduction
- ❏ Stack
- ❏ Queue
- ❏ Deque
- ❏ Monotonic Queue / Stack
- ❏ Linked List

Outline

❑ **Introduction**

❑ Stack

❑ Queue

❑ Deque

❑ Monotonic Queue / Stack

❑ Linked List

Why data structures?

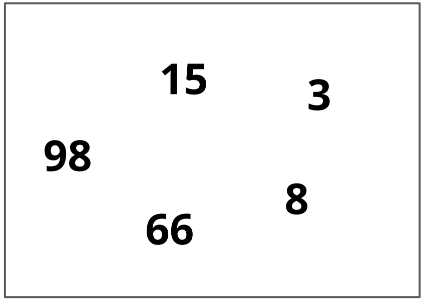
Data structures = specific ways to order data, to achieve...

- More efficient operations
- Better space complexity

i.e. less likely to have MLE/TLE if you choose the right structure :)

Major Operations

1.	Insert	*(insert 89)*
2.	Delete	*(remove 8)*
3.	Modify	*(change 3 → 5)*
4.	Query / Find	*(find max value)*



an unknown data structure containing some integers

Outline

❑ Introduction

❑ **Stack**

❑ Queue

❑ Deque

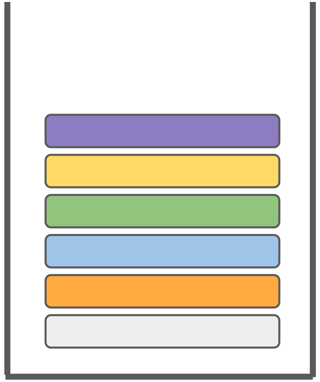
❑ Monotonic Queue / Stack

❑ Linked List

Stack - Idea

Imagine a box of books:

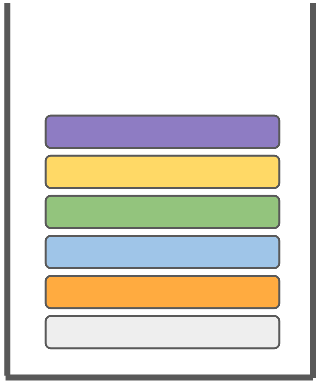
- always grab the top book
- always add new books at the top



Stack - Idea

What if we want to access a book at the middle?

1. Take out all books above it (from top to bottom)
2. Take that book out
3. Put the other books back



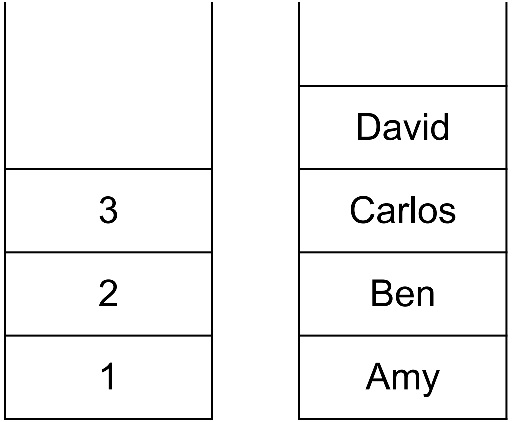
More formally...

Last-In-First-Out (LIFO)

✔ Access (insert and retrieve) top element

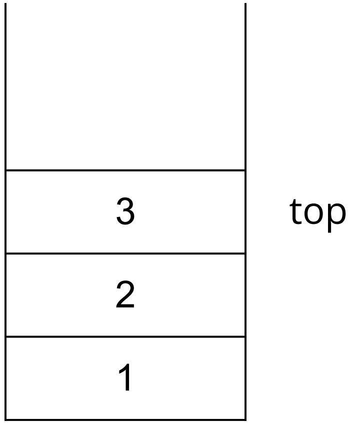
✘ Directly access other elements below

Note: content can be any data type (not only int)

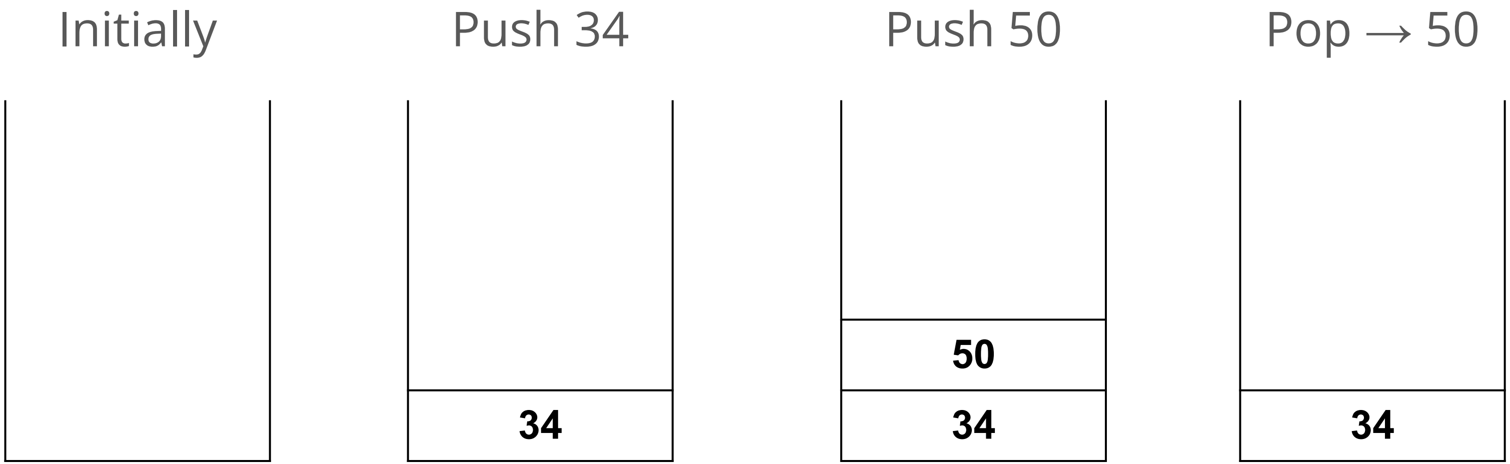


Stack - Operations

1. Push x *(Insert)* = Add x to top
2. Pop *(Delete)* = Remove and return top



Stack - Operations



Initially

Push 34

34

Push 50

50

34

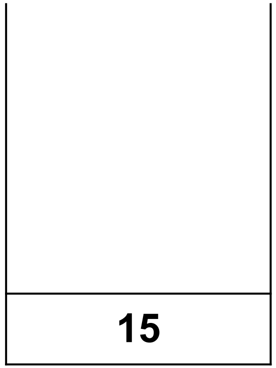
Pop → 50

34

Stack - Practice

1. Push 15
2. Push 87
3. Pop
4. Push 19
5. Pop

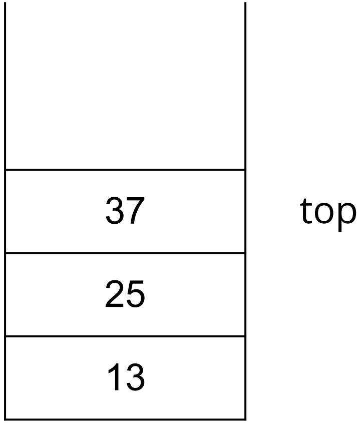
What is the final stack?



Stack - Array Representation

i	1	2	3	4	5
a[i]	13	25	37		

top



Stack - Implementation

```
Push(x):
    if top >= N:
        return

    top += 1
    a[top] = x
```

(stack is full)

Push 25

top = 1

i	1	2	3
a[i]	13		

top = 2

i	1	2	3
a[i]	13	25	

Stack - Implementation

```
Is_empty:
    return top == 0
```

return 0 (not empty)

```
top = 3
```

i	1	2	3
a[i]	13	25	19

return 1 (is empty)

```
top = 0
```

i	1	2	3
a[i]			

Stack - Implementation

```
Pop:
    if Is_empty:
        return

    top -= 1
```

Note: value not removed in array

Pop

`top = 3`

i	1	2	3
a[i]	13	25	19

⇩

`top = 2`

i	1	2	3
a[i]	13	25	~~19~~

Stack - Implementation

```
Top:
    if Is_empty:
        return 0

    return a[top]
```

Top → return 19

top = 3

i	1	2	3
a[i]	13	25	19

Stack - Example

1. Push 13
2. Push 25
3. Push 19
4. Pop
5. Pop

i	1	2	3	4	5
1.	13				
2.	13	25			
3.	13	25	19		
4.	13	25			
5.	13				

Stack - C++ STL

```
stack<int> st;

for (int i = 1; i <= 5; i++)
    st.push(i);

while (!st.empty()) {
    cout << st.top() << “ ”;
    st.pop();
}
```

C++ <stack> library

More in c++ reference

Stack - Analysis

Memory	O(N)
Push	O(1)
Pop	O(1)
Access any element	O(N)

Parentheses Balance - Problem

Given a string `s` of length `N` composed of the characters “`()[]{}`” only.

Can you determine if the parentheses are balanced? `(N <= 100000)`

“`[{((}]`” → false

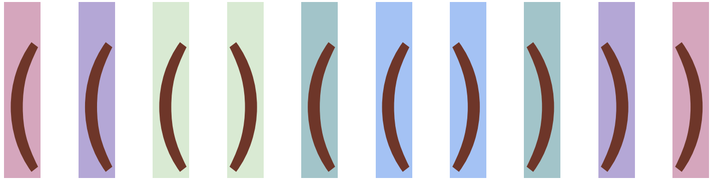
“`{(()[])[()]}`” → true

Parentheses Balance - Observation

- All “(“ are either followed by another “(“ or its matching “)“

⇒ the last “(“ in continuous “(“s is followed by its matching “)“

e.g. the light green and blue “(“



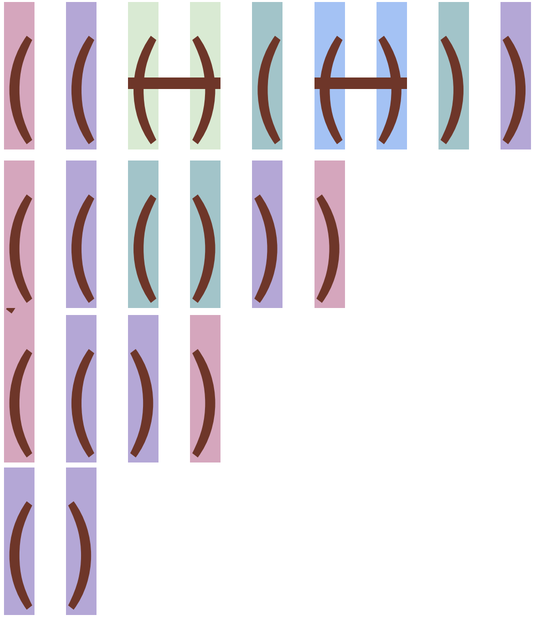
Parentheses Balance - Observation

- we can keep removing pairs of “( )”
- last “(“ maps with earliest “)“

→ last “(“ is removed earliest

→ last-in-first-out structure!

- we can implement with stack



Parentheses Balance - Observation

One more thing... how do we know a sequence is invalid?

- no unmatched “(“ before “)”
- “(“ without matching “)“ behind it

( )) (( )))) (

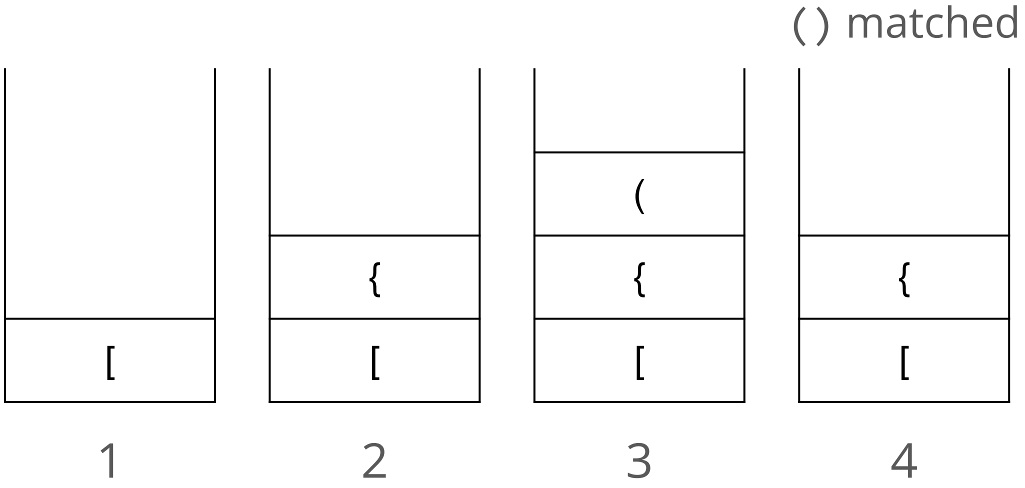
Parentheses Balance - Implementation

```
stack st;
for i = 0 to N-1:
    if (s[i] == ‘(’ or ‘[’ or ‘{’ ):
        push s[i] into st
    else
        if (st is empty) or (mismatch with st.top):
            return invalid
        else:
            pop st

if st is not empty:
    return invalid
```

Time Complexity	O(N)
Space Complexity	O(N)

Parentheses Balance - Example

`string “[{()]}”`



5:

] mismatch with {

→ invalid

Reverse Polish Notation(RPN) / Suffix Notation

Put operator(+ - * /) behind operands (numbers)

No brackets are needed

`1 + 2`	`1 2 +`
`3 * 4`	`3 4 *`
`1 + 2 * 3`	`1 2 3 * +`
`3 * (5 + 1)`	`3 5 1 + *`

RPN - Problem

Given a string s of length N in form of a valid RPN.

All operands are 0-9, operands and operators are separated by a space.

How to evaluate the expression and output the result?

“3 5 * 9 + 2 -” → (3 * 5) + 9 - 2 = 22

“9 1 - 8 + 3 * 4 6 + *” → (((9 - 1) + 8) * 3) * (4 + 6) = 480

RPN - Implementation

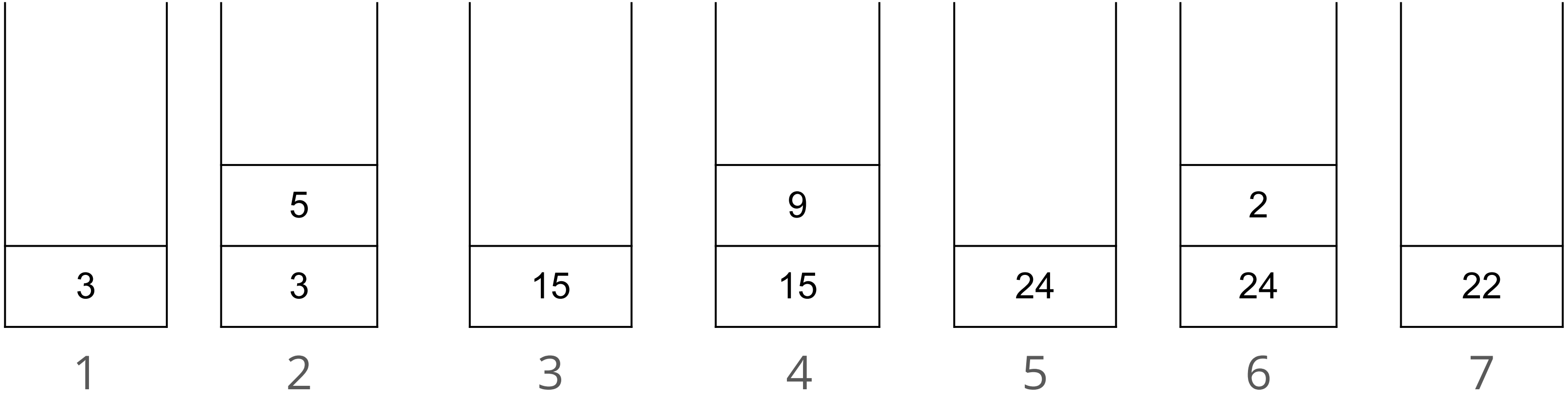
```
stack st;
for i = 0 to N-1:
     if s[i] is digit:
          push s[i] to st
     else:
          num_2 = pop st  //pushed later
          num_1 = pop st  //pushed earlier
          res = calc(num_1, s[i], num_2)
          push res to st

ans = pop st
output ans
```

Time Complexity O(N)

Space Complexity O(N)

RPN - Example

```
3 5 * 9 + 2 -
```



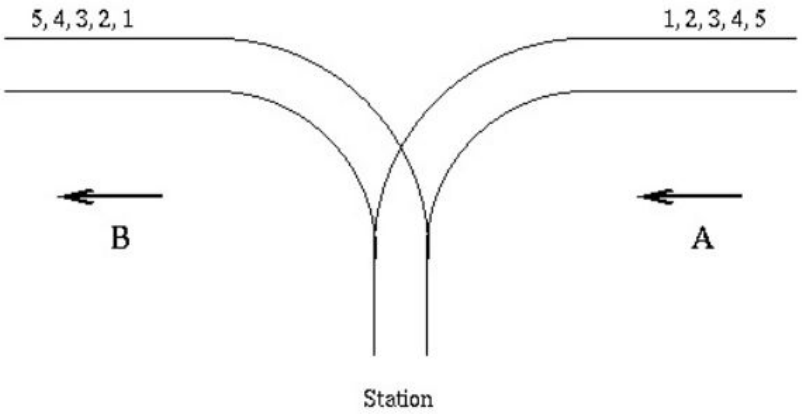
Stack - More Applications

● Recursion	recursion, divide and conquer session
● Depth-first search	graph (I) session

Rails - Problem

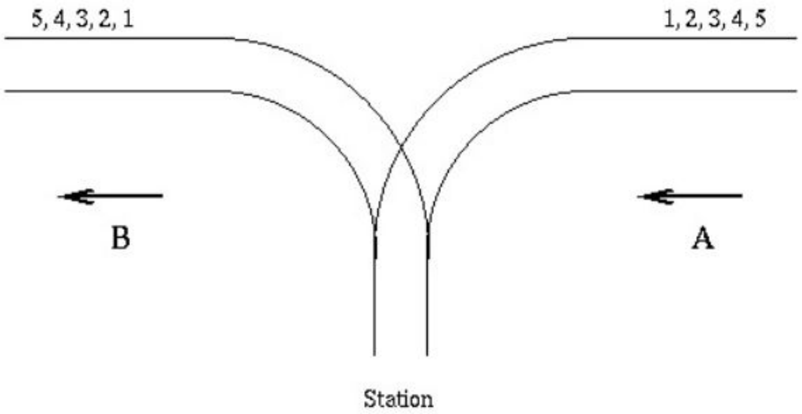
Trains are in in order 1..N.

Is it possible to have trains leaving in order p[1..N]?



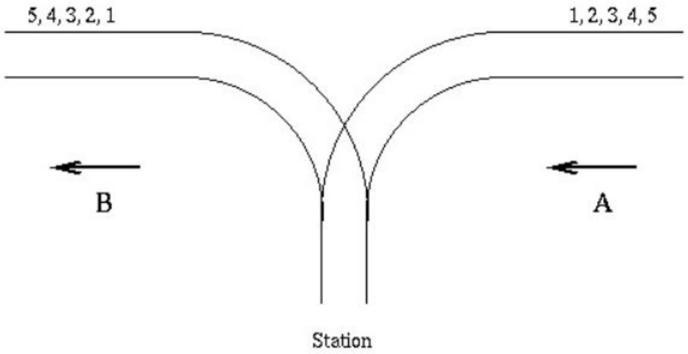
Rails - Observation

The middle part is like a stack



Rails - Observation

```
For each train:
    while stack.top() == next_out_train
        stack.pop()
    push train i into stack
while stack.top() == next_out_train
        stack.pop()
if !stack.empty(): NO
```



5-minute break (until 10:39)

Practice tasks

01015 Parantheses Balance

01033 Simple Arithmetic

20514 Rails

NP1712 時間複雜度

M1313 Bookstack

M1803 I love you I love you

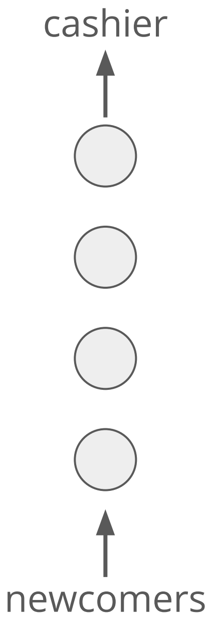
Outline

- ❑ Introduction
- ❑ Stack
- ❑ **Queue**
- ❑ Deque
- ❑ Monotonic Queue / Stack
- ❑ Linked List

Queue - Idea

Imagine a waiting line in the supermarket:

- the first person in line proceeds first
- newcomers join the line at the back



More formally...

First-In-First-Out (LIFO)

✔ Pop (dequeue) front element

✔ Push (enqueue) element to the back

X Pop/Push any elements in the middle

Note: content can be any data type (not only int)

Queue - Operations

1. Push x (enqueue) = Add x to tail
2. Pop (dequeue) = Remove and return front

1	2	3	4	5
head/front				tail/rear

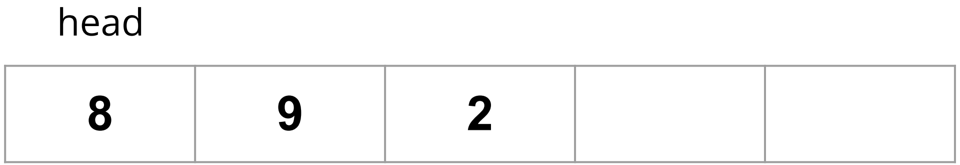
Queue - Operations

	head				
Initially					
Push “a”	a				
Push “b”	a	b			
Pop	b				

Queue - Practice

1. Push 3
2. Push 8
3. Pop
4. Push 9
5. Push 2

What is the final queue?



head

8	9	2		

Queue - Array Representation

head = 1, tail = 3

i	1	2	3	4	5
a[i]	13	25	37		

Pop ↓

head = 2, tail = 3

i	1	2	3	4	5
a[i]	~~13~~	25	37		

O(1) - preferred
(solve space issue with circular queue)

head = 1, tail = 3

i	1	2	3	4	5
a[i]	13	25	37		

Pop ↓

head = 1, tail = 3

i	1	2	3	4	5
a[i]	25	37			

O(N) - more intuitive but not preferred

Queue - Implementation

```
Is_full:
    return tail >= N
```

head = 1, tail = 3

i	1	2	3
a[i]	13	25	27

Queue - Implementation

```
Push(x):
    if Is_full:
        return

    tail += 1
    a[tail] = x
```

Push 25

head = 1, tail = 1

i	1	2	3
a[i]	13		

⇩

head = 1, tail = 2

i	1	2	3
a[i]	13	25	

Queue - Implementation

```
Is_empty:
    return tail < head
```

head = 1, tail = 0

i	1	2	3
a[i]			

head = 3, tail = 2

i	1	2	3
a[i]			

Queue - Implementation

```
Pop:
    if Is_empty:
        return

    head += 1
```

Pop

head = 1, tail = 2

i	1	2	3
a[i]	13	25	

⇩

head = 2, tail = 2

i	1	2	3
a[i]	~~13~~	25	

Queue - Implementation

```
Front:
    if Is_empty:
        return -1

    return a[head]
```

`head = 1, tail = 2`

i	1	2	3
a[i]	13	25	

Queue - Example

1. Push 13
2. Push 25
3. Push 19
4. Pop
5. Pop

i	1	2	3	4	5
1.	13				
2.	13	25			
3.	13	25	19		
4.	~~13~~	25	19		
5.	~~13~~	~~25~~	19		

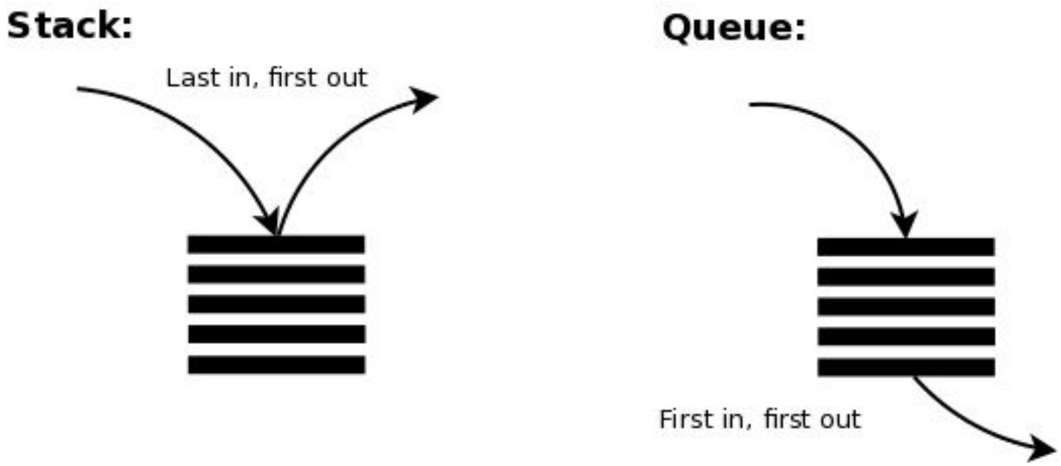
Queue - C++ STL

```
queue<int> q;

for (int i = 1; i <= 5; i++)
    q.push(i);

while (!q.empty()) {
    cout << q.front() << “ ”;
    q.pop();
}
```

C++ <queue> library

More in c++ reference

Stack v.s. Queue



Circular Queue - Motivation

The array becomes full quickly after many operations...

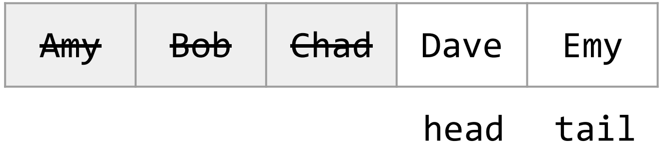
`a[1..head-1]` are “empty”, can we make use of these spaces?

`head = 6, tail = 9`

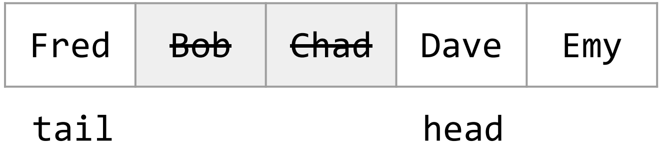
i	1	2	3	4	5	6	7	8	9
a[i]	~~Amy~~	~~Bob~~	~~Chad~~	~~David~~	~~Emy~~	Frank	Gail	Harry	Iris

Circular Queue

When the array is full, reuse the empty slots at the front



~~Amy~~	~~Bob~~	~~Chad~~	Dave	Emy
			head	tail



Fred	~~Bob~~	~~Chad~~	Dave	Emy
tail			head	

Circular Queue

The queue can always hold `N-1` items

Why not `N`?

`head == tail + 1?`

cannot distinguish full and empty queue

`head = 1, tail = 0`

i	1	2	3	4	5
a[i]					

`head = 4, tail = 3`

1	2	3	4	5
Fred	Gary	~~Henry~~	Dave	Emy

Circular Queue

How to shift the head and `tail` values so it will go back to the start?

`head = 3, tail = 5`

i	1	2	3	4	5
a[i]	~~Amy~~	~~Bob~~	~~Chad~~	Dave	Emy

`head = 3, tail = 1`

i	1	2	3	4	5
a[i]	Fred	~~Bob~~	~~Chad~~	Dave	Emy

Circular Queue

```
if head/tail >= N:
    head/tail = 1
else:
    head/tail += 1
```

head = 3, tail = 5

i	1	2	3	4	5
a[i]	~~Amy~~	~~Bob~~	~~Chad~~	Dave	Emy

head = 3, tail = 1

i	1	2	3	4	5
a[i]	Fred	~~Bob~~	~~Chad~~	Dave	Emy

Circular Queue

1-based array

`x = x % N + 1`

`head = 3, tail = 5`

i	1	2	3	4	5
a[i]	~~Amy~~	~~Bob~~	~~Chad~~	Dave	Emy

0-based array

`x = (x + 1) % N`

`head = 3, tail = 4`

i	0	1	2	3	4
a[i]	Fred	~~Bob~~	~~Chad~~	Dave	Emy

More about modular arithmetic in Math in OI (I)

Circular Queue - Implementation

```
Is_full:
    return tail + 2 == head
```

```
head = 4, tail = 2
```

i	1	2	3	4
a[i]	13	25	~~29~~	35

Circular Queue - Implementation

```
Push(x):
    if Is_full:
        return

    tail = tail % N + 1
    a[tail] = x
```

Push 13

head = 1, tail = 0

i	1	2	3
a[i]			

head = 1, tail = 1

i	1	2	3
a[i]	13		

Circular Queue - Implementation

```
Is_empty:
    return tail == head - 1
```

`head = 2, tail = 1`

i	1	2	3
a[i]	~~13~~	~~25~~	~~37~~

`head = 1, tail = 0`

i	1	2	3
a[i]			

Circular Queue - Implementation

```
Pop:
    if Is_empty:
        return

    head = head % N + 1
```

Pop

head = 1, tail = 2

i	1	2	3
a[i]	13	25	

head = 2, tail = 2

i	1	2	3
a[i]	~~13~~	25	

Circular Queue - Implementation

```
Front:
    if Is_empty:
        return -1

    return a[head]
```

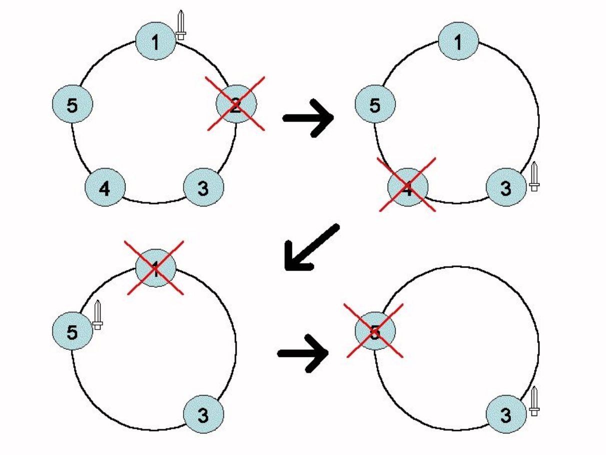
head = 1, tail = 2

i	1	2	3
a[i]	13	25	

The Josephus Problem

- 1 ≤ N ≤ 1000 soldiers arranged in a circle
- Soldier 1 is holding a sword initially
- The one holding a sword will:
  a. kill the survivor on his left
  b. pass the sword to the survivor on his left

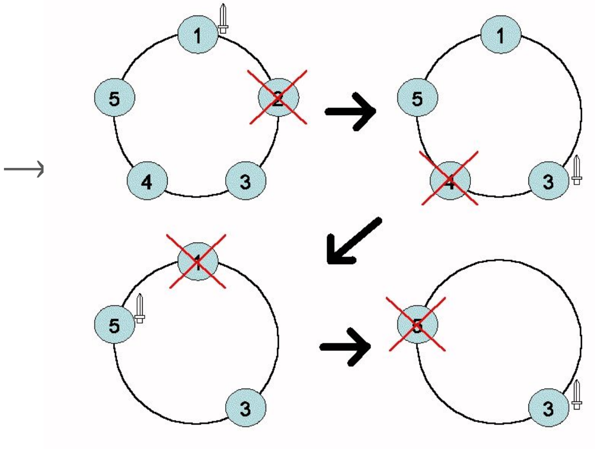
Who is the final survivor?



The Josephus Problem - Observation

N = 5:

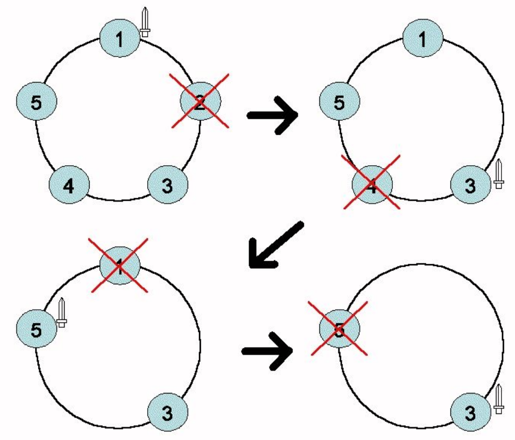
1 → X2 → 3 → X4 → 5 → X1 → 3 → X5 →



each round, soldiers are processed in 1..N,
until only one soldier survives after last round

The Josephus Problem - Idea

- maintain a line of soldiers to be processed
- a surviving soldier will go back to the end of line to be processed in next round
- we can simulate the line with a queue



The Josephus Problem - Implementation

```
var queue q
for i = 1 to N do: push i into q
for i = 1 to N-1 do
    push q.front into q
    pop q for twice
output remaining element in q
```

use a circular array to save space

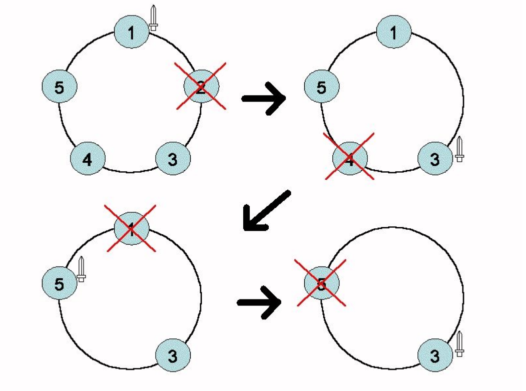
Time complexity: O(N)

The Josephus Problem - Example

Circular queue of fixed size 5.

Initial	1	2	3	4	5
Kill #1	1		3	4	5
Kill #2	1	3			5
Kill #3		3	5		
Kill #4				3	

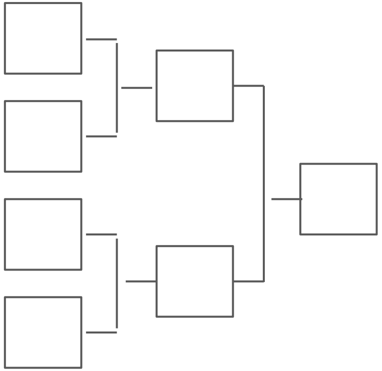
▨ = HEAD
▨ = TAIL



Tournament

In a tournament, contestants are divided into pairs, winner of each pair advance into next round, then paired with the winner of another pair.

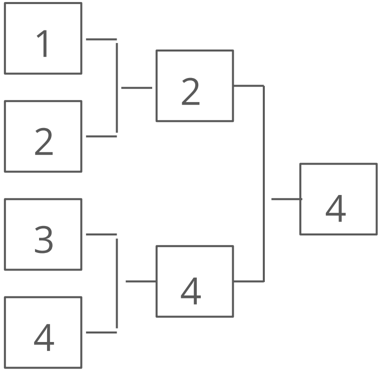
The process repeats until there is only one winner left



Tournament

Given N (N = 2^k) contestants and their strength (contestant with higher strength wins a match).

Output the winners of each round.

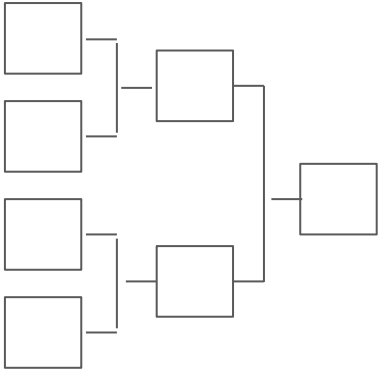


Tournament

We can use a queue to simulate the process.

For each match:

- pop the 2 contestants,
- determine the winner
- push winner back into the queue for next round



Tournament - Implementation

```
var queue q
for i = 1 to N do:
    push i into q
for i = 1 to N - 1 do:
    opp_1 = q.front
    pop q
    opp_2 = q.front
    pop q
    if opp_1.strength > opp_2.strength:
        output opp_1
        push opp_1 into q
    else:
        output opp_2
        push opp_2 into q

output q.front
```

Tournament - Example

5 9 13 4 8 2 1 7

5	9	13	4	8	2	1	7		

~~5~~	~~9~~	13	4	8	2	1	7	9	

~~5~~	~~9~~	~~13~~	~~4~~	8	2	1	7	9	13

8	~~9~~	~~13~~	~~4~~	~~8~~	~~2~~	1	7	9	13

8	7	~~13~~	~~4~~	~~8~~	~~2~~	~~1~~	~~7~~	9	13

8	7	13	~~4~~	~~8~~	~~2~~	~~1~~	~~7~~	~~9~~	~~13~~

~~8~~	~~7~~	13	8	~~8~~	~~2~~	~~1~~	~~7~~	~~9~~	~~13~~

~~8~~	~~7~~	~~13~~	~~8~~	13	~~2~~	~~1~~	~~7~~	~~9~~	~~13~~

Queue - Analysis

Memory	O(N)	(circular queue performs much better)
Push	O(1)	
Pop	O(1)	
Access any element	O(N)	

Queue - More Applications

● Breadth-First search	graph (I) session
● Shortest Path Faster Algorithm (SPFA)	graph (II) session

5-minute break (until 11:31)

Practice tasks

01017 Car Sorter

01030 The Josephus Problem

M1721 Bus Fare II

Outline

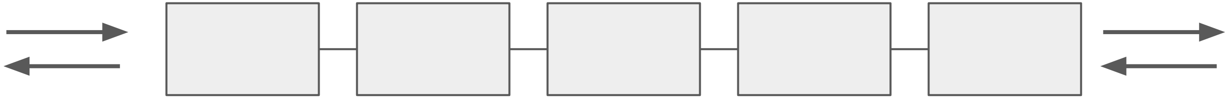
- ❏ Introduction
- ❏ Stack
- ❏ Queue
- ❏ **Deque**
- ❏ Monotonic Queue / Stack
- ❏ Linked List

Deque - Idea

Deque = Double-ended Queue

Imagine a train:

- we can push/pop elements at both front and end



Deque - Operations

1. Push_back x = Add x to end
2. Pop_back = Remove last element
3. Push_front x = Add x to front
4. Pop_front = Remove first element

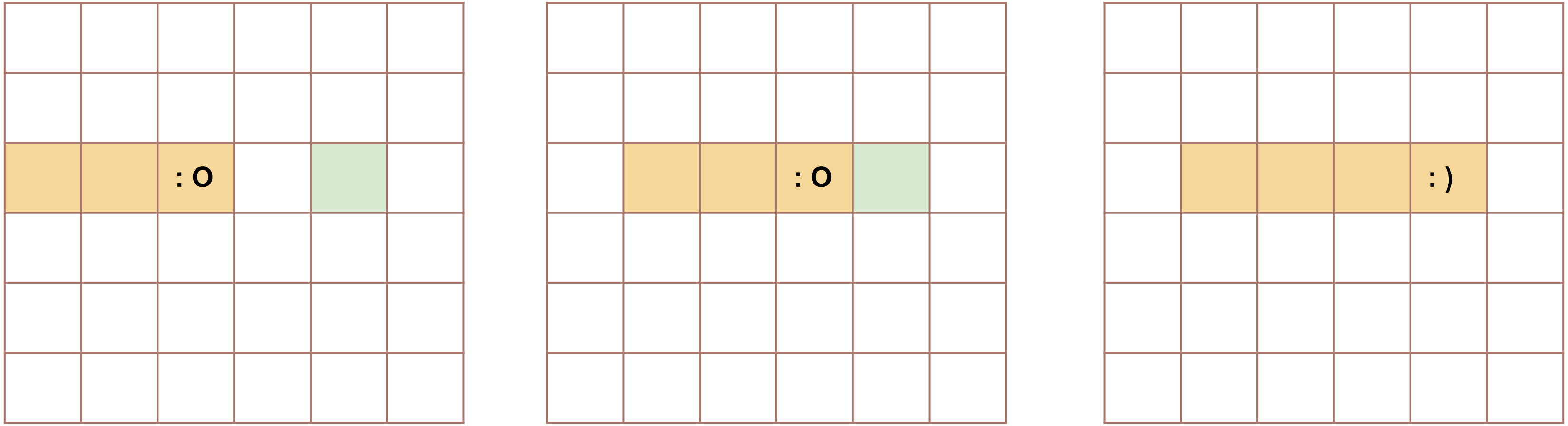
Deque - C++ STL

```
deque<int> dq;

dq.push_back(15);
dq.push_front(3);

cout << dq.back() << endl;
cout << dq.front() << endl;

dq.pop_back();
dq.pop_front();
```

C++ <deque> library

More in c++ reference

Snake Game



Snake Game - Move Rule

Represent the snake with a list of coordinates: `[(r1, c1), (r2, c2), ...]`

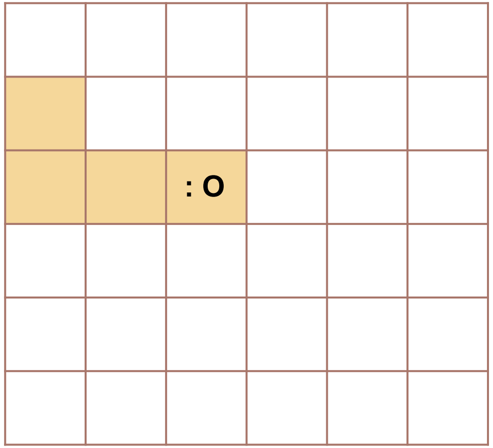
⇒ How do we update the coordinates efficiently as the snake moves / grows?

Move rule:

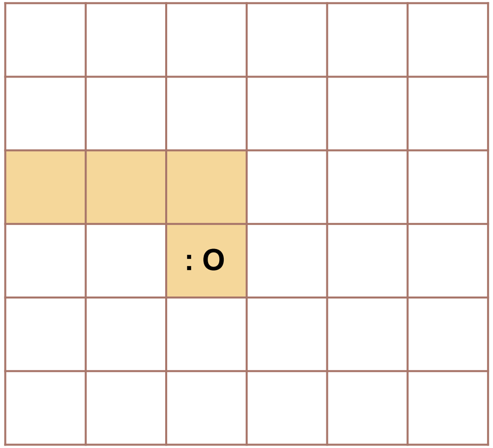
- The snake's head is moved to the new position, adjacent to the old head position
- The snake's tail is shifted to its previous second last position

Snake Game - Move Rule

Move Rule Illustration



`(3,3), (3,2), (3,1), (2,1)`



`(4,3), (3,3), (3,2), (3,1)`

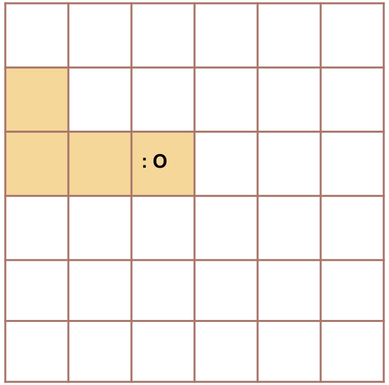
Snake Game - Move Rule

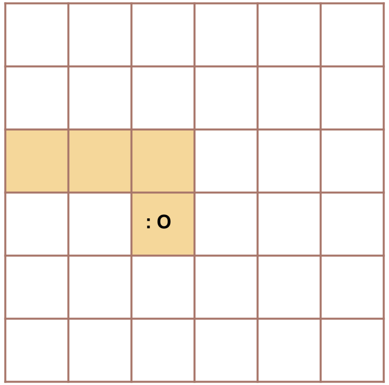
Let’s compare the coordinates before and after the move:

```
          (3,3), (3,2), (3,1), (2,1)
(4,3), (3,3), (3,2), (3,1)
```

New element added at head, and the last element removed

⇒ we can use a deque to simulate it





Snake Game - Grow Rule

Represent the snake with a list of coordinates: `[(r1, c1), (r2, c2), ...]`

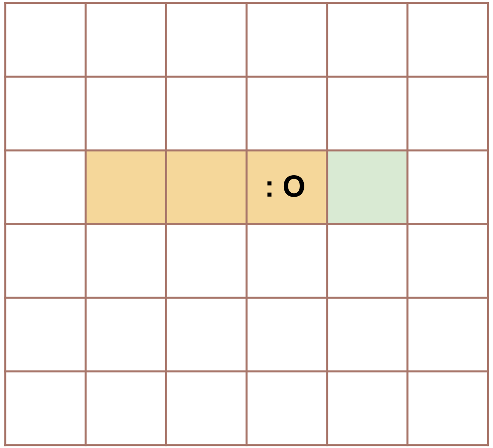
⇒ How do we update the coordinates efficiently as the snake moves / grows?

Grow rule:

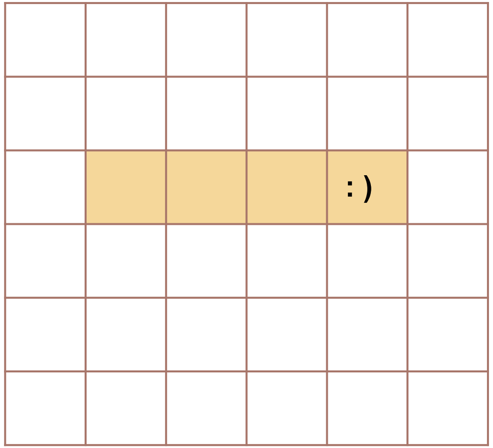
- snake’s head moves to food’s position → snake’s length increase by 1 unit

Snake Game - Grow Rule

Grow Rule Illustration



(3,4), (3,3), (3,2)



(3,5), (3,4), (3,3), (3,2)

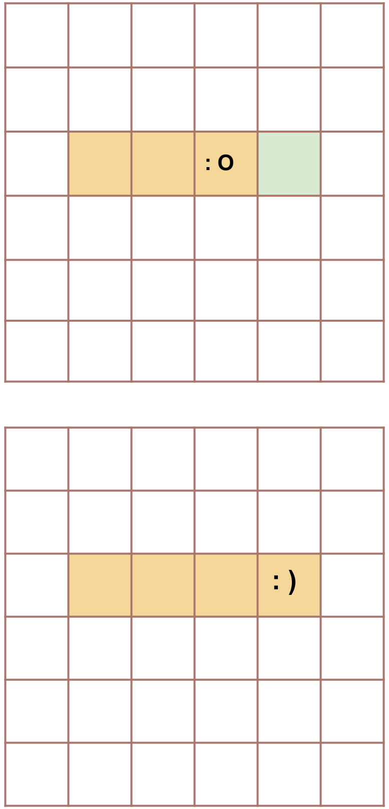
Snake Game - Grow Rule

Let’s compare the coordinates before and after the move:

```
       (3,4), (3,3), (3,2)
(3,5), (3,4), (3,3), (3,2)
```

New element added at head

⇒ again, we can use a deque to simulate it



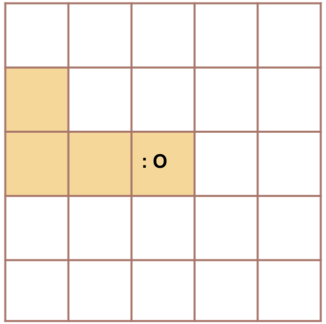
Snake Game - Idea

Store the coordinates in a deque

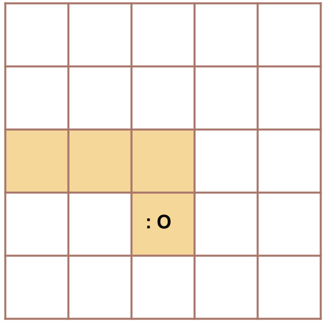
- head at front, body part at middle, tail at back
- push new head to front / pop old tail

Snake Game - Implementation

```
deque dq

snake_move (next_r, next_c)

    dq.push_front( (next_r, next_c) )

    dq.pop_back
```



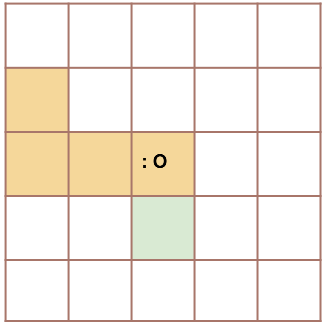
(3,3), (3,2), (3,1), (2,1)



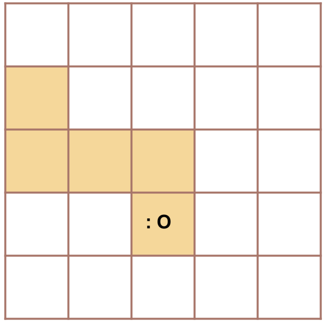
(4,3), (3,3), (3,2), (3,1)

Snake Game - Implementation

```
deque dq

snake_grow (next_r, next_c)

    dq.push_front( (next_r, next_c) )
```



(3,3), (3,2), (3,1), (2,1)



(4,3), (3,3), (3,2), (3,1), (2,1)

Broken Keyboard - Observation

Output = rearrangement of texts separated by ‘[‘ or ‘]’

```
This_is_a_[Beiju]_text
Beiju|This_is_a_|_text
```

Broken Keyboard - Observation

For a text segment, we know it is entered at front or back based on previous [or]

```
This_is_a_[Beiju]_text
Beiju|This_is_a_|_text
```

Broken Keyboard - Implementation

```
maintain a deque dq, prev_state = back (front/back)
for i = 1 to N:
    if s[i] == ‘[‘ or ‘]’:
        push last_segment into prev_state of dq
        if [: prev_state = front; else prev_state = back
```

5-minute break

Practice tasks

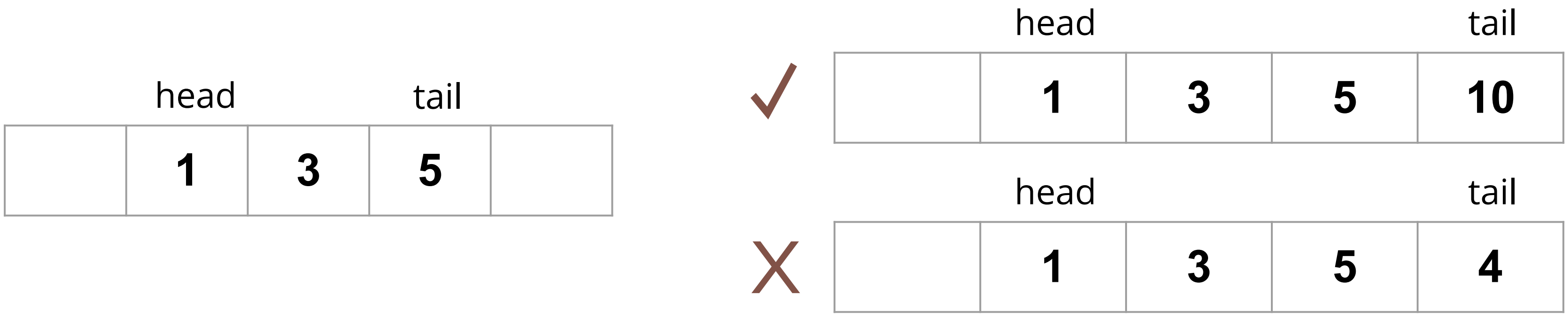
31988 Broken Keyboard

Outline

- ❑ Introduction
- ❑ Stack
- ❑ Queue
- ❑ Deque
- ❑ **Monotonic Queue / Stack**
- ❑ Linked List

Monotonic Queue (Deque) - Idea

A deque with all elements following a certain order *(e.g. increasing/ decreasing)*



	head		tail	
	1	3	5	

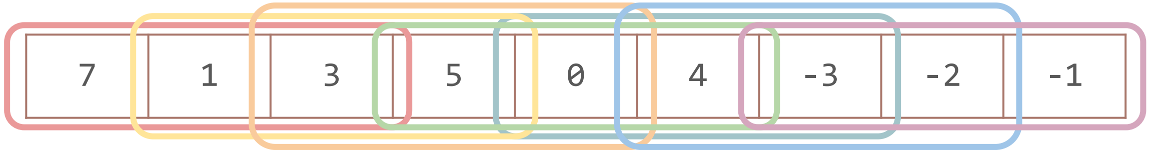
		head			tail
✓		1	3	5	10

		head			tail
✗		1	3	5	4

Sliding Window Maximum

Given a distinct integer array A, there is a sliding window of size k that slides from the beginning to the end of the array.

Find the maximum element in the sliding window for every window in A.



Assume k = 3, the results should be: 7, 5, 5, 5, 4, 4, -1

Sliding Window Maximum

Queue / two-pointer implementation:

The queue keeps track of the elements in the sliding window.

For every window, finding the max element using linear scan takes O(k) ➔ not good enough

What if I use a heap? (Data Structures II) ➔ O(log k) ➔ still not good enough

Sliding Window Maximum

Monotonic queue (deque) implementation:

The deque keeps track of some of the elements in the sliding window following a decreasing order.

We can find the maximum element of a window in constant time (O(1))

Sliding Window Maximum - Idea

For sure we want to keep the maximum element within the window, but does that mean we should forget the remaining elements in the same window?

Example: A = [10, 1, 3, 2] and k = 3

The max element in the first window is 10. If we ignore the other elements, we would not get 3 as the max unless we iterate through the entire window.

Sliding Window Maximum - Idea

`A = [10, 1, 3, 2] and k = 3`

If we focus in the first window [10, 1, 3], we can already confirm that 1 would not be a candidate for the following windows.

Why?

Sliding Window Maximum - Idea

Any window containing 1, must either contain 10 or 3, which are better candidates than 1 for a window’s maximum.

Maybe we can keep track of the possible candidates in a window to find the maximum value.

Sliding Window Maximum - Implementation

```
Maintain a deque dq
for i = 1 to N do
    if dq.front().id == i-k:
        dq.pop_front()
    while dq.back().value < A[i]:
        dq.pop_back()
    dq.push_back(A[i])
    if i >= k:
        output dq.front().value
```

Sliding Window Maximum - Example

7	3	1	5	0	4	-3	-2	-1

k = 3

▨= sliding window max

(1)	7			push back (7)
(2)	7	3		push back (7)
(3)	7	3	1	push back (1)
(4a)	3	1		pop front (7 not in the window)
(4b)	3			pop back (1 impossible to be max)

(4c)				Pop back (3 impossible to be max)
(4d)	5			push back (5)
(5)	5	0		push back (0)
(6a)	5			pop back (0 impossible to be max)
(6b)	5	4		push back (4)

Sliding Window Maximum - Example

7	3	1	5	0	4	-3	-2	-1	k = 3

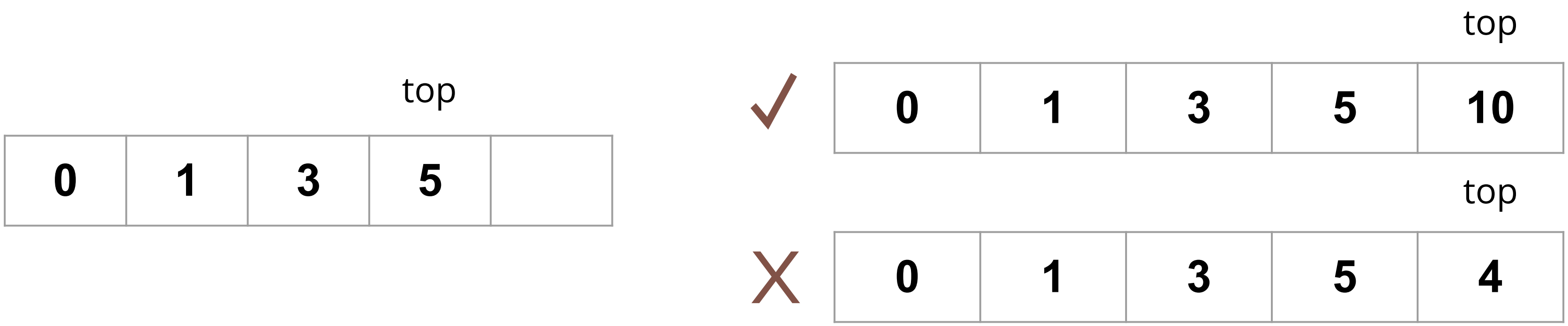
▨= sliding window max

(7a)	4			pop front (5 not in the window)
(7b)	4	-3		push back (-3)
(8a)	4			pop back (-3 impossible to be max)
(8b)	4	-2		push back (-2)
(9a)	-2			pop front (4 not in the window)

(9b)				Pop back (-2 impossible to be max)
(9c)	-1			push back (-1)

Monotonic Stack - Idea

A stack with all elements following a certain order *(e.g. increasing/ decreasing)*



top

0	1	3	5	

top

✓

0	1	3	5	10

top

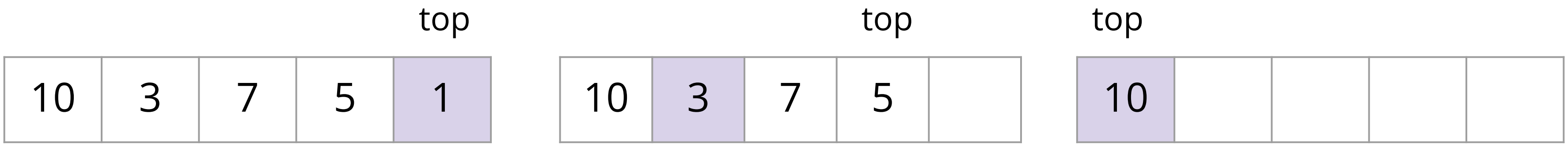
✗

0	1	3	5	4

Min Stack

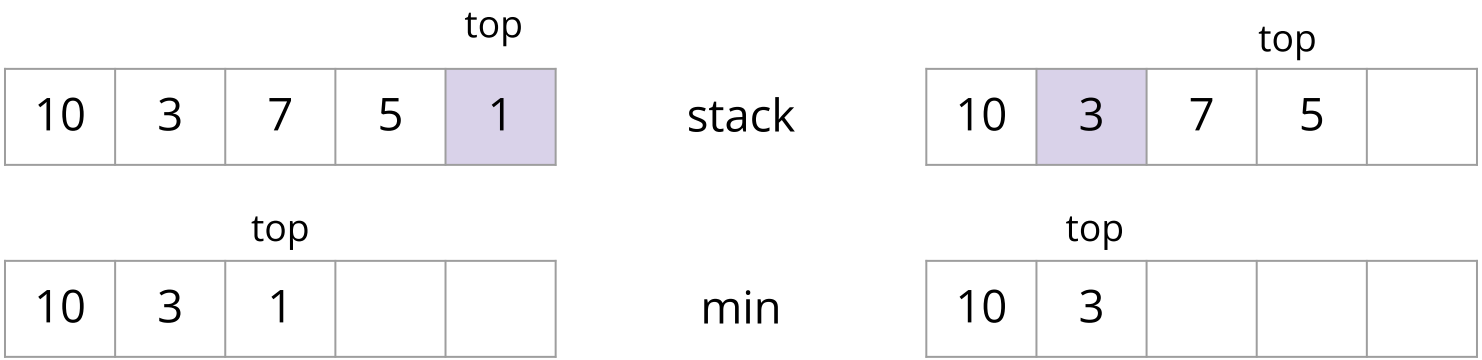
Implement a stack that supports min query (returns the minimum element)

for simplicity we assume the elements are unique



Min Stack - Idea

use an extra stack to maintain the minimum element + the candidate min elements after popping the current minimum.



				top					top	
10	3	7	5	1	stack	10	3	7	5	
		top					top			
10	3	1			min	10	3			

Min Stack - Implementation

```
maintain stack S and stack Min
push(x):
    S.push(x)
    if Min.top() > x do Min.push(x)
```

Min Stack - Implementation

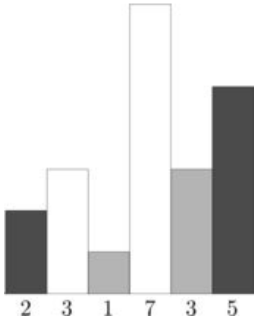
```
pop():
    x = S.top()
    S.pop()
    if Min.top() == x do Min.pop()
```

```
min():
    return Min.top()
```

Largest Rectangle in Histogram

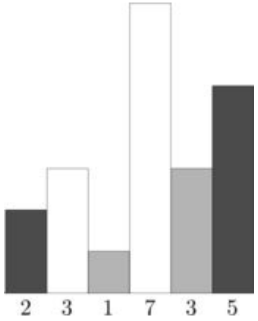
Given a histogram with n bars, each of height h_i.

Find the area of the largest rectangle in the histogram.

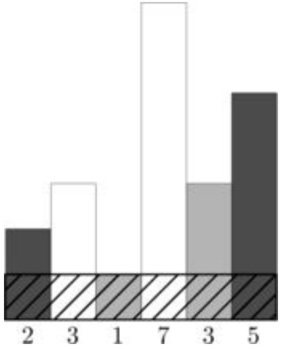


Largest Rectangle in Histogram

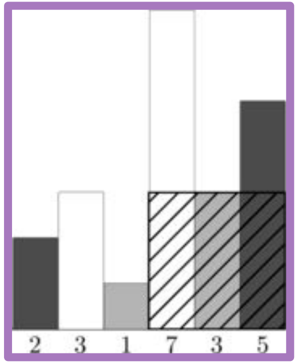
Histogram



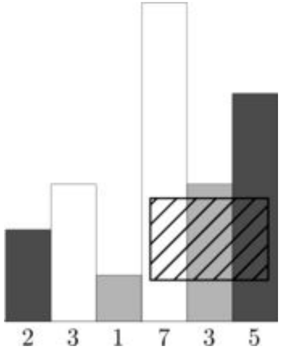
Possible rectangles



Area = 6



Area = 9

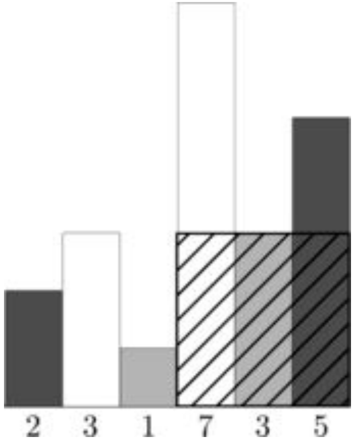


Area ≈ 5

Largest Rectangle in Histogram - Observation

height of the largest rectangle = one of the bars in the histogram

what about its width?



Largest Rectangle in Histogram - Observation

When we fix one bar and treat it as the height of the rectangle (h), we can expand our rectangle if the height of the bars adjacent to the rectangle >= h

The question now becomes: for each bar, find the number of bars with height >= its height so that they can stick together as a rectangle.

Largest Rectangle in Histogram - Idea

For the i-th bar, we want to find the leftmost bar and the rightmost bar that are lower than hi

For the rightmost bar - as we scan the bars from left to right, we can include all bars after bar i and stop once we see a bar of height < hi on the right

Largest Rectangle in Histogram - Idea

For the leftmost bar - we use a monotonic stack to store the bars shorter than bar i

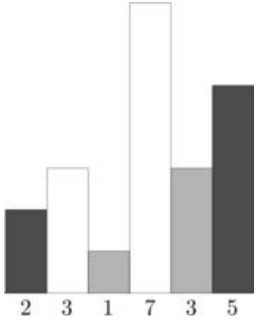
Since we are scanning the bars from the left, the indices are obviously in increasing order. What we want is to store indices of the bars in increasing heights (indices at the top of the stack have higher bars), while all the heights in the stack are shorter than bar i.

Largest Rectangle in Histogram - Implementation

```
Maintain a stack S and maxArea = 0
S.push(0) // boundary
for every bar i do
  while S not empty AND h[i] < h[S.top()] do
    height = h[S.top()] // calculate max area including bar of index S.top()
    left = S.empty() ? 0 : S.top() + 1
    maxArea = max(maxArea, height * (i - left))
    S.pop()
  S.push(i)
return maxArea
```

Largest Rectangle in Histogram - Implementation

```
maintain a stack S (<height, left_index>) and maxArea = 0
h[N] = 0
S.push( (-1, -1) ) //leftmost boundary
for i = 0 to N do  //exhaust all bars as height of rectangle
  leftbound = i
  while h[i] < S.top().height do
     maxArea = max(maxArea, S.top().height * (i - S.top().left_index)
     leftbound = S.top().left_index
     S.pop()
  S.push( (h[i], leftbound) )
return maxArea
```



Monotonic Stack/Queue - More Applications

Dynamic programming optimizations

- Dynamic Programming (III) session

5-minute break

Practice tasks

31988 Broken Keyboard

32462 Largest Rectangle in Histogram

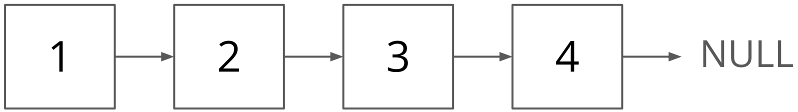
Outline

- ❏ Introduction
- ❏ Stack
- ❏ Queue
- ❏ Deque
- ❏ Monotonic Queue / Stack
- ❏ **Linked List**

Linked List - Idea

A list where each element points to the next

Contents likely not stored in access order

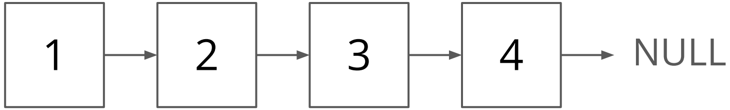


i	1	2	3	4	5
a[i]	?	1	3	2	4
p[i]	?	4	5	3	-1

Linked List - Operations

1. Insert element at any position
2. Erase element at any position
3. Access the first element directly

Cannot access other elements directly

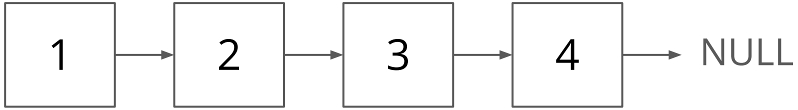


Linked List - Array Representation

`a[i]` = node content

`p[i]` = index of next node

head = index of starting node



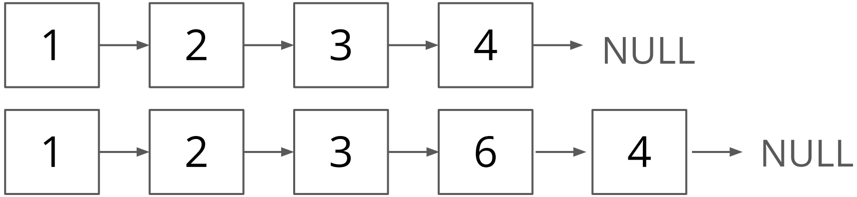
i	1	2	3	4	5
a[i]	?	1	3	2	4
p[i]	?	4	5	3	-1

```
head = 2
```

Linked List - Implementation

```
Insert x after yth node:
    cur = head
    for i = 1 to y - 1 do
        cur = p[cur]

    insert x in an empty slot of a
    record ind_x  //index of x

    p[ind_x] = p[cur]
    p[cur] = ind_x
```

Insert 6 after 3

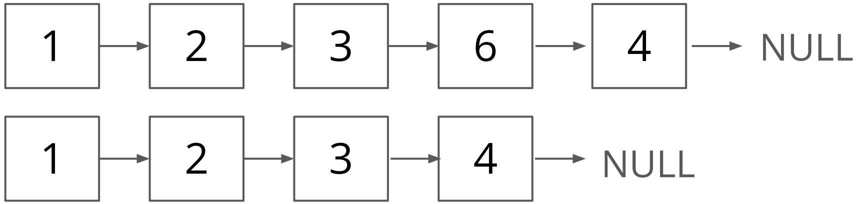


i	1	2	3	4	5
a[i]	6	1	3	2	4
p[i]	5	4	~~5~~ 1	3	-1

Linked List - Implementation

```
Erase xth node
    cur = head
    for i = 1 to x - 2 do
        cur = p[cur]

    prev_x = cur
    cur = p[cur]

    p[prev_x] = p[cur]
```

Erase 4th node (6)

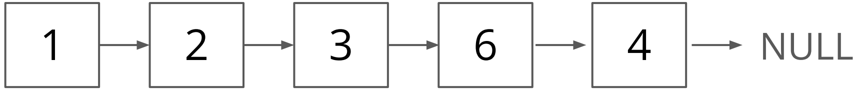


i	1	2	3	4	5
a[i]	6	1	3	2	4
p[i]	5	4	~~1~~ 5	3	-1

Linked List - Implementation

```
Query xth node
    cur = head
    for i = 1 to x - 1 do
        cur = p[cur]
    output a[cur]
```

Query 2nd node → 2



i	1	2	3	4	5
a[i]	6	1	3	2	4
p[i]	5	4	5	3	-1

Linked List - Array Representation

Need to find empty slots in array a for insert value → restrictive

We can use pointer approach instead (dynamic memory allocation)

Linked List - Implementation

```
struct Node {
    int data;
    struct Node *next;
    Node(int x): data(x), next(NULL) {}
}

Node *head = NULL;
int list_size = 0;
```

Linked List - Implementation

```
Insert x after yth node:
  list_size += 1
  //create new node
  Node* new_node = new Node(x);

  if (y == -1):  //empty linked list
    new_node->next = head;
    head = new_node;
  else:
    Node* cur = head;
    for i = 1 to y - 1:
      cur = cur->next;

    new_node->next = cur->next;
    cur->next = new_node;
```

Linked List - Implementation

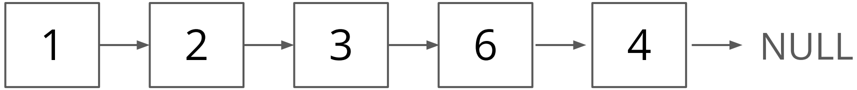
```
Erase xth node:
    list_size -= 1
    Node* xth_node;

    //erase first element
    if (x == 0):
        xth_node = head;
        head = xth_node → next
```

```
  else:
    Node* prev = head
    for i = 1 to x - 2:
        prev = prev → next
    xth_node = prev → next
    prev → next = xth_node →
next

    delete xth_node
```

Linked List - Implementation

```
Query xth node
    Node* cur = head
    for i = 1 to x - 1:
        cur = cur → next
    output cur→data
```

Query 2nd node → 2



i	1	2	3	4	5
a[i]	6	1	3	2	4
p[i]	5	4	5	3	-1

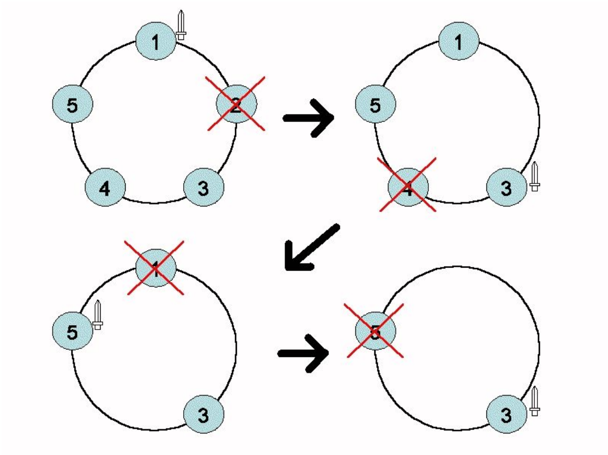
Linked List - Analysis

Insert	O(N)	*first element: O(1)*
Delete	O(N)	*first element: O(1)*
Access any element	O(N)	

The Josephus Problem

- $1 \leq N \leq 1000$ soldiers arranged in a circle
- Soldier 1 is holding a sword initially
- The one holding a sword will:
  a. kill the survivor on his left
  b. pass the sword to the survivor on his left

Who is the final survivor?



The Josephus Problem - Linked List Approach

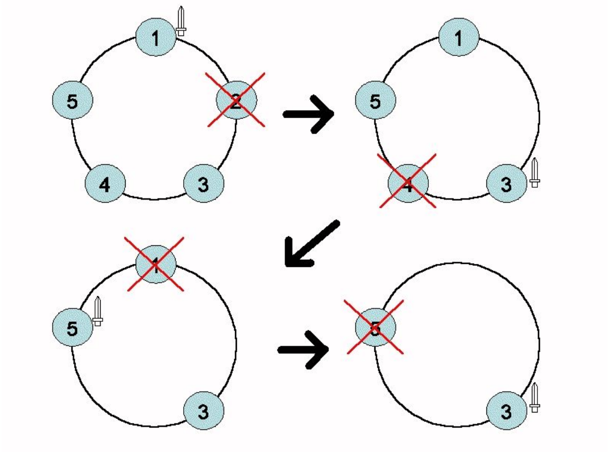
Represent soldiers in a linked list

for each soldier with a sword:

1. erase its next soldier,
2. pass the sword by visiting the new next soldier (not the one just being killed, but the one after)

Repeat process for N-1 times (until 1 survivor)

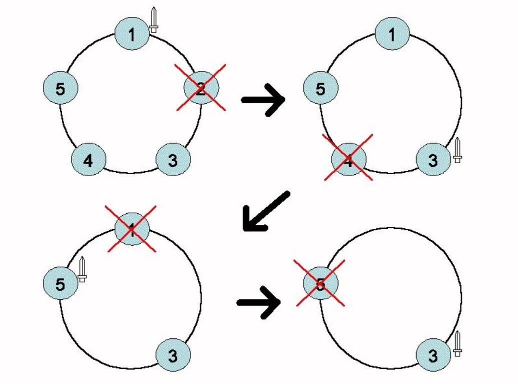
Time and space complexity O(N)



The Josephus Problem - Linked List Approach

Let’s say N=5.

Initial	2	3	4	5	1
Kill #1	3	NULL	4	5	1
Kill #2	3	NULL	5	NULL	1
Kill #3	NULL	NULL	5	NULL	3
Kill #4	NULL	NULL	5	NULL	NULL



We don’t need an element array since the element in the i^{th} position is i itself

More Linked Lists and Applications

Doubly-linked list	2018 Data Structures (I) materials
Circular-linked list	
XOR linked lists	
Adjacency List	Graph (I) session

Practice List

Stack / Queue

HKOJ P005 Rails

HKOJ 01017 Car Sorter

HKOJ M1721 Bus Fare II

HKOJ 32462 Largest Rectangle in Histogram

HKOJ M1803 I love you I love you

HKOJ 01015 Parentheses Balance

HKOJ 01033 Simple Arithmetic

HKOJ M1313 Bookstack

HKOI NP1712 時間複雜度

Linked Lists

HKOJ 01030 The Josephus Problem

HKOJ 31988 Broken Keyboard

HKOJ T151 Conveyor Belt Sushi

CF 797C Minimal String

More problems on

HKOJ [Data Structures]

Codeforces [Data Structures]

Q&A