Stacks and Queues

Chapter - 8

Definition

- A stack is a linear list in which insertions (additions and pushes) and removals (deletions and pops) take place at the same end.
- This end is called the top.
- The other end is called the bottom.
- A stack is a LIFO list.
- Trying to pop out an empty stack is called *underflow*
- *T*rying to push an element in a full stack is called *overflow*

Stack configuration

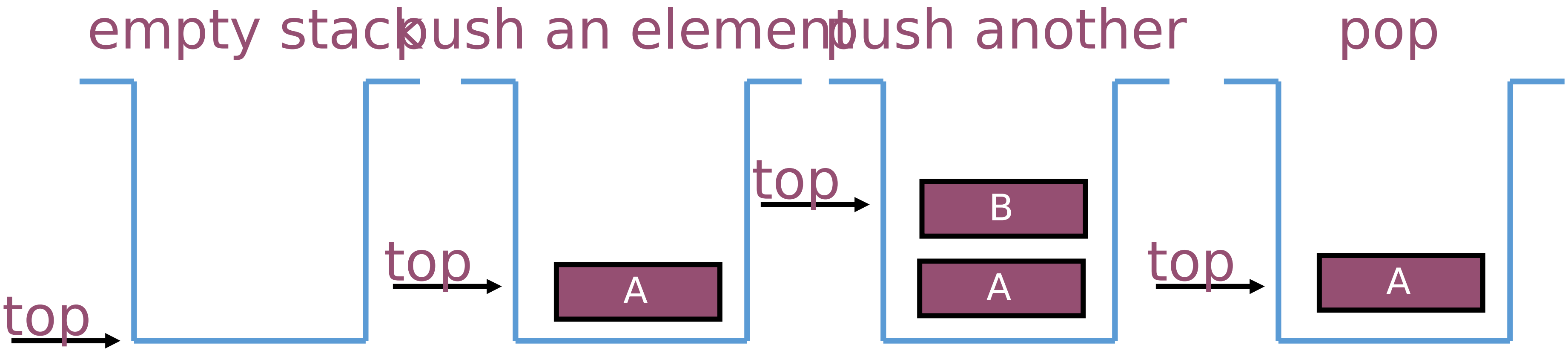
	E ←top	
D←top	D	
C	C	
B	B	B←top
A←bottom	A←bottom	A←bottom .

Standard operations:

- IsEmpty … return true iff stack is empty
- Top … return top element of stack
- Push … add an element to the top of the stack
- Pop … delete the top element of the stack

Push and Pop

- Primary operations: Push and Pop
- Push
  - Add an element to the top of the stack
- Pop
  - Remove the element at the top of the stack



Stacks : Implementation of Array

Any list implementation could be used to implement a stack

- Arrays (static: the size of stack is given initially)
- Linked lists (dynamic: never become full)

Let *n* be the number of elements in the stack. The complexities of stack operations with this representation can be given as:

Space Complexity (for n push operations)	O(*n*)
Time Complexity of Push()	O(1)
Time Complexity of Pop()	O(1)
Time Complexity of Size()	O(1)
Time Complexity of IsEmptyStack()	O(1)
Time Complexity of IsFullStackf)	O(1)
Time Complexity of DeleteStackQ	O(1)

Array Implementation

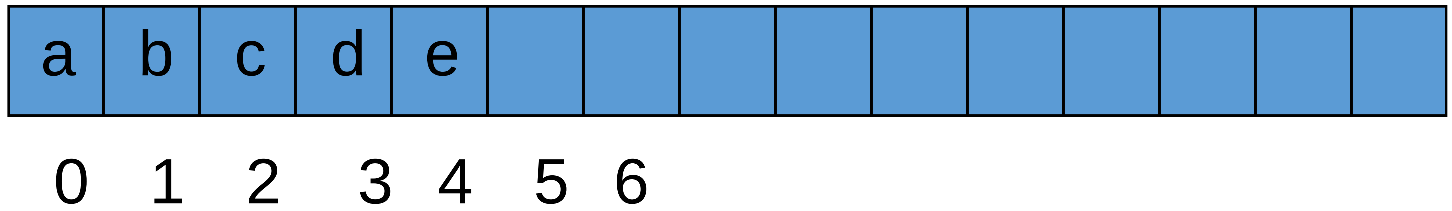
- Need to declare an array size ahead of time
- Associated with each stack is TopOfStack
  - for an empty stack, set TopOfStack to -1
- Push
  - (1) Increment TopOfStack by 1.
  - (2) Set Stack[TopOfStack] = X
- Pop
  - (1) Set return value to Stack[TopOfStack]
  - (2) Decrement TopOfStack by 1
- These operations are performed in very fast constant time

Stacks

a	b	c	d	e										
0	1	2	3	4	5	6								

- stack top is at element e
- IsEmpty() => check whether top >= 0
  - O(1) time
- Top() => If not empty return stack[top]
  - O(1) time

Derive From arrayList



- Push(theElement) => if array full (top == capacity – 1) increase capacity and then add at stack[top+1]
- O(capacity) time when full; otherwise O(1)
- pop() => if not empty, delete from stack[top]
- O(1) time

Applications of Stack

- Write an algorithm to Reverse a string using stack.

(Stack each character and then perform pop operation)

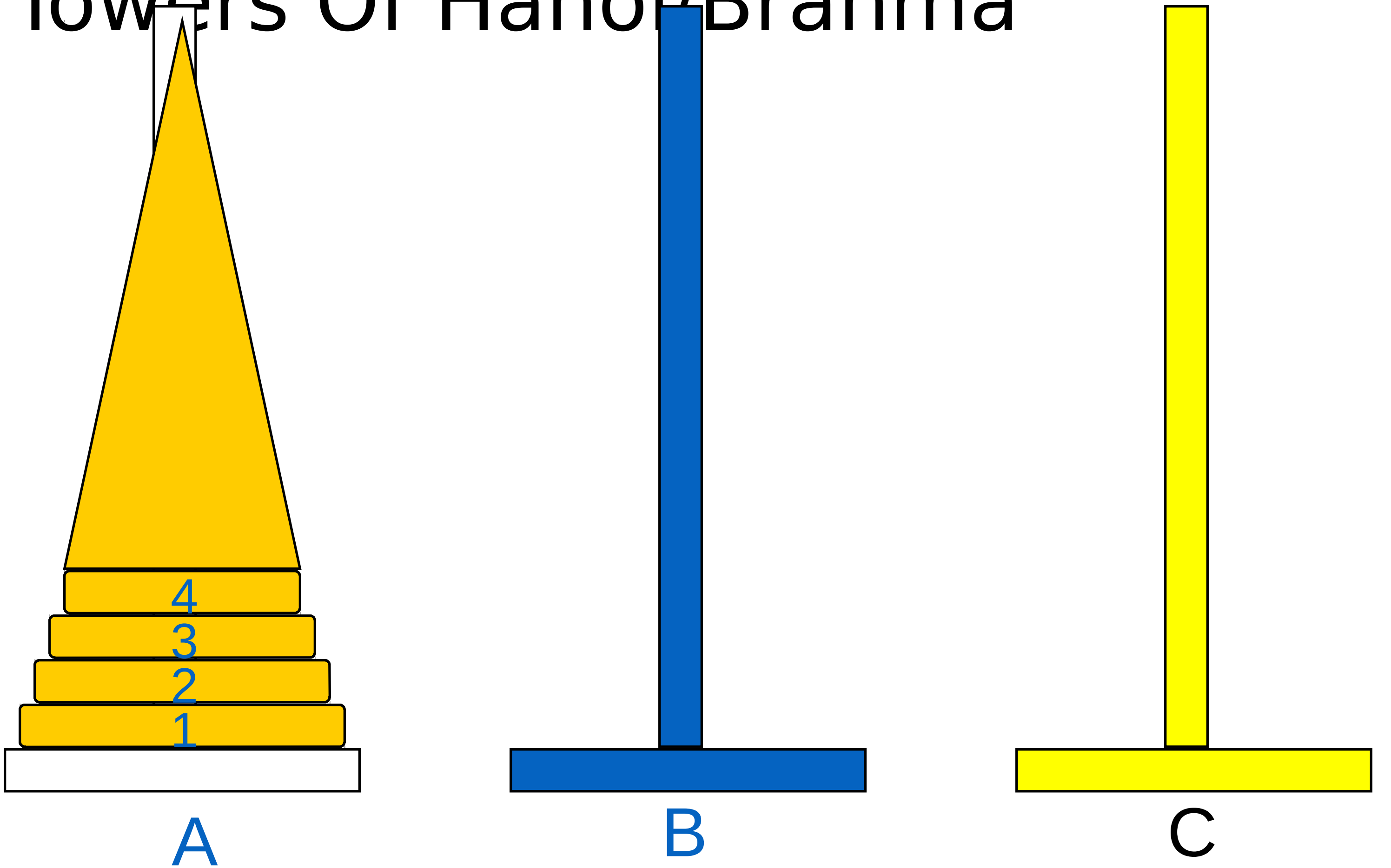
Parentheses Matching

- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)
  - Output pairs (u,v) such that the left parenthesis at position u is matched with the right parenthesis at v.
    - (2,6) (1,13) (15,19) (21,25) (27,31) (0,32) (34,38)
- (a+b))*((c+d)
  - (0,4)
  - right parenthesis at 5 has no matching left parenthesis
  - (8,12)
  - left parenthesis at 7 has no matching right parenthesis

Parentheses Matching

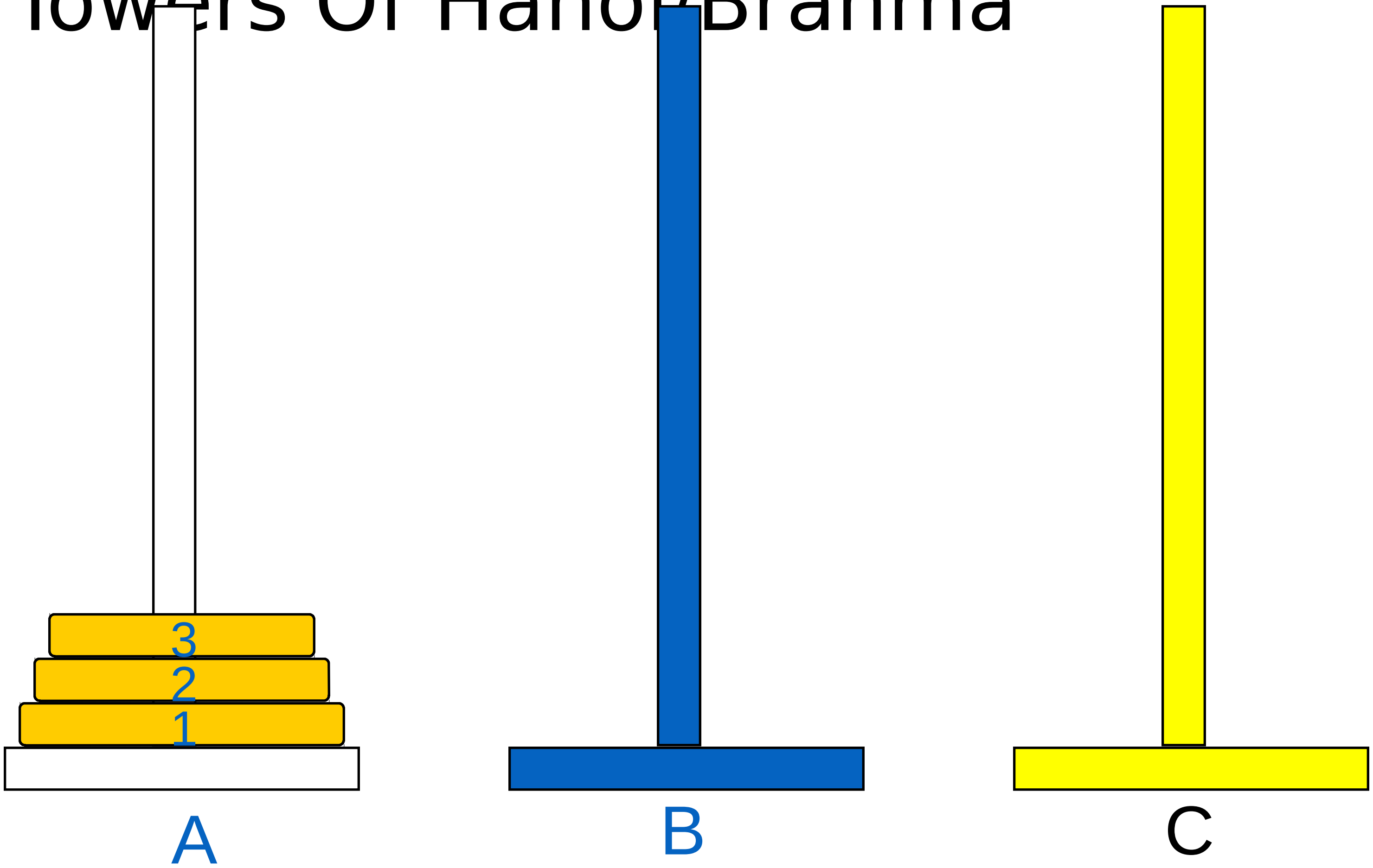
- scan expression from left to right
- when a left parenthesis is encountered, add its position to the stack
- when a right parenthesis is encountered, remove matching position from stack

Towers Of Hanoi/Brahma



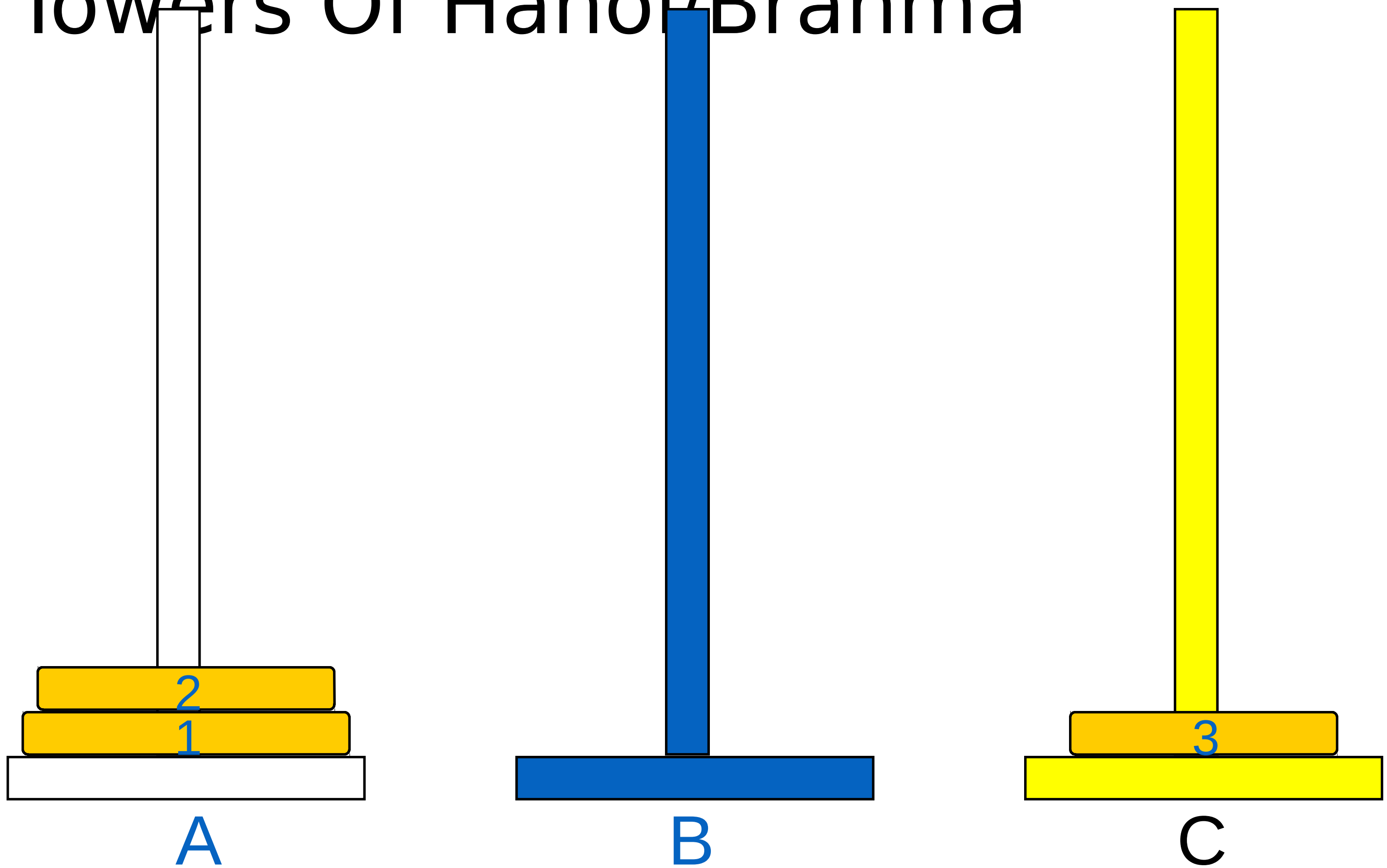
- 64 gold disks to be moved from tower A to tower C
- each tower operates as a stack

Towers Of Hanoi/Brahma



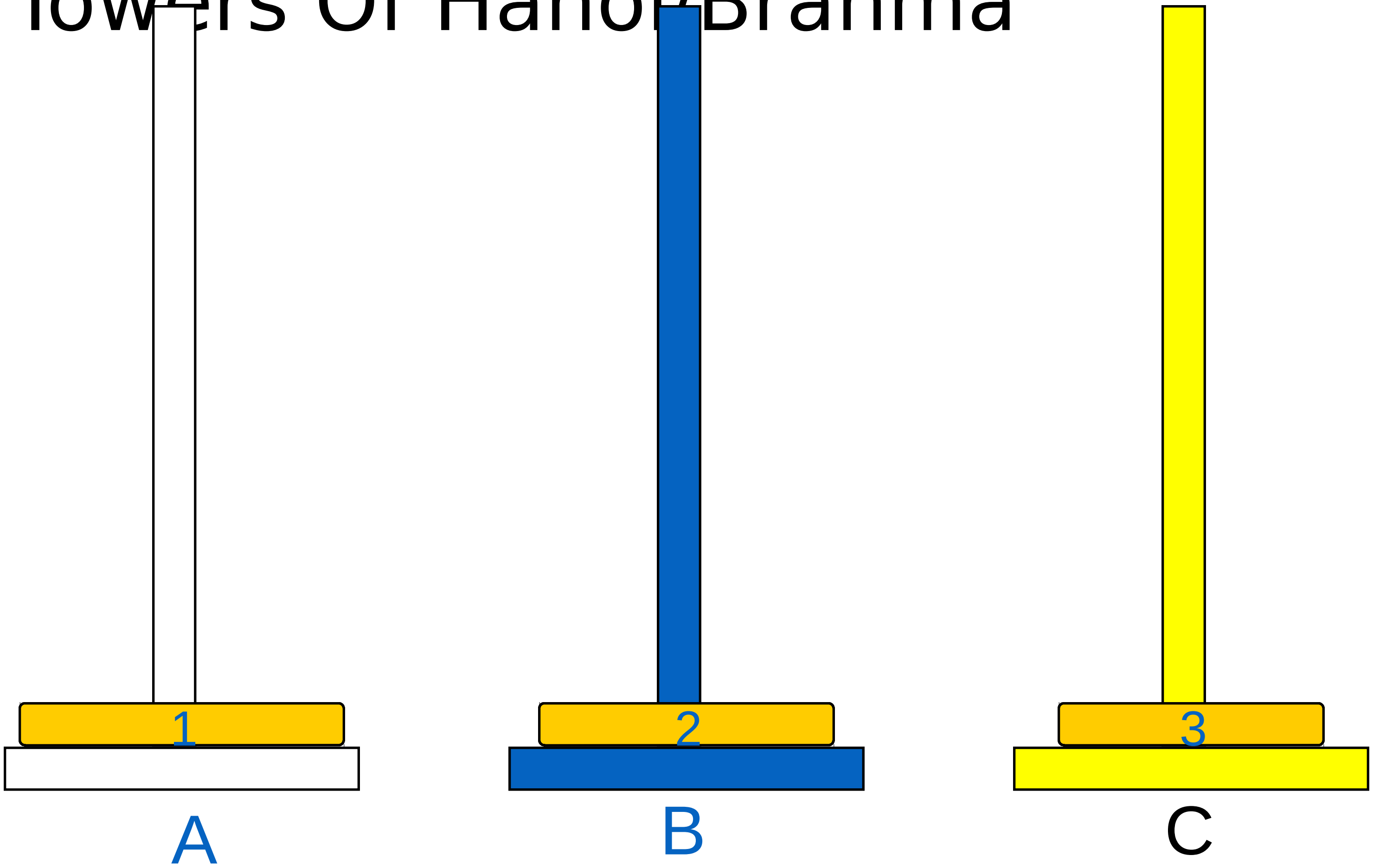
- 3-disk Towers Of Hanoi/Brahma

Towers Of Hanoi/Brahma



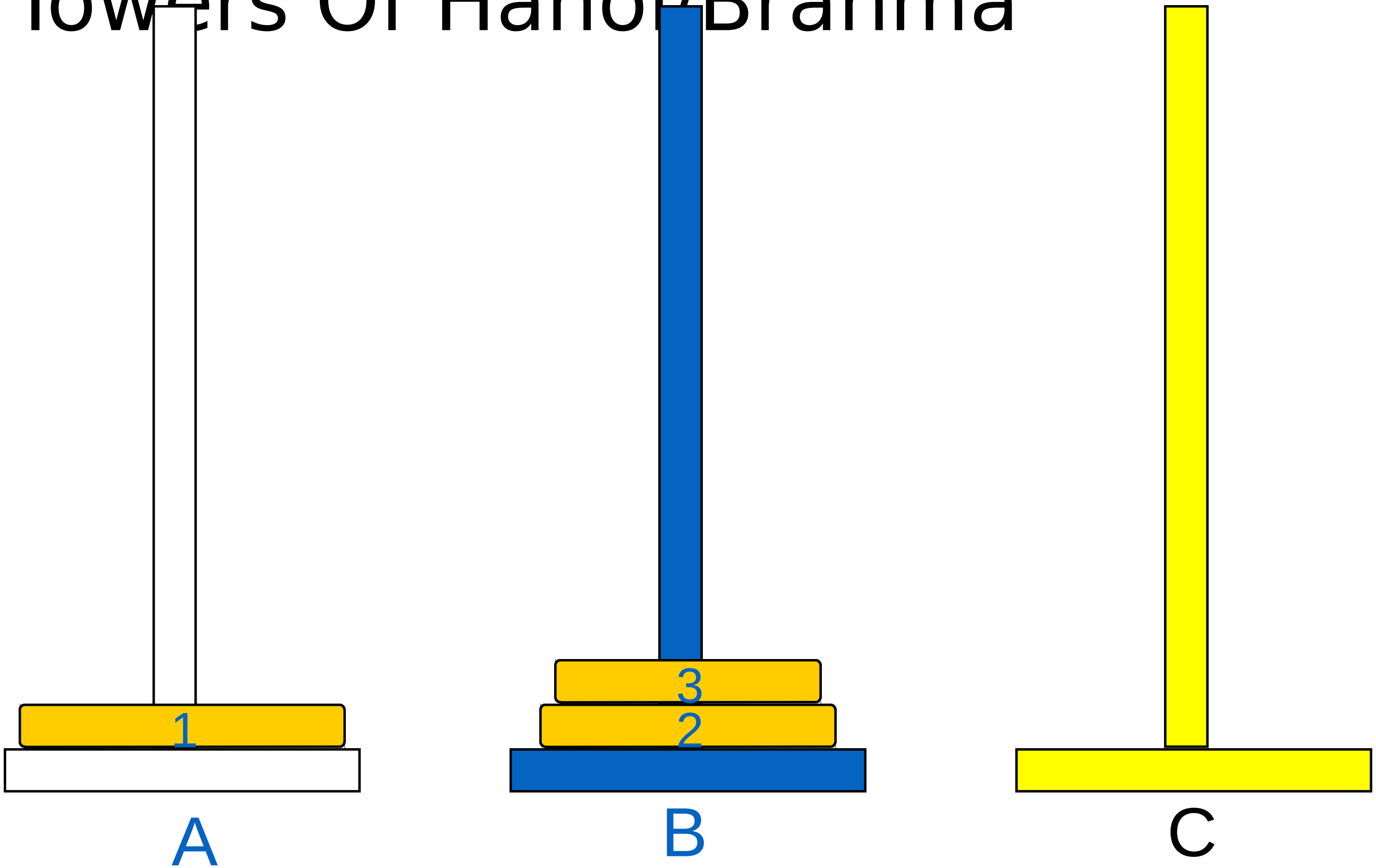
- 3-disk Towers Of Hanoi/Brahma

Towers Of Hanoi/Brahma



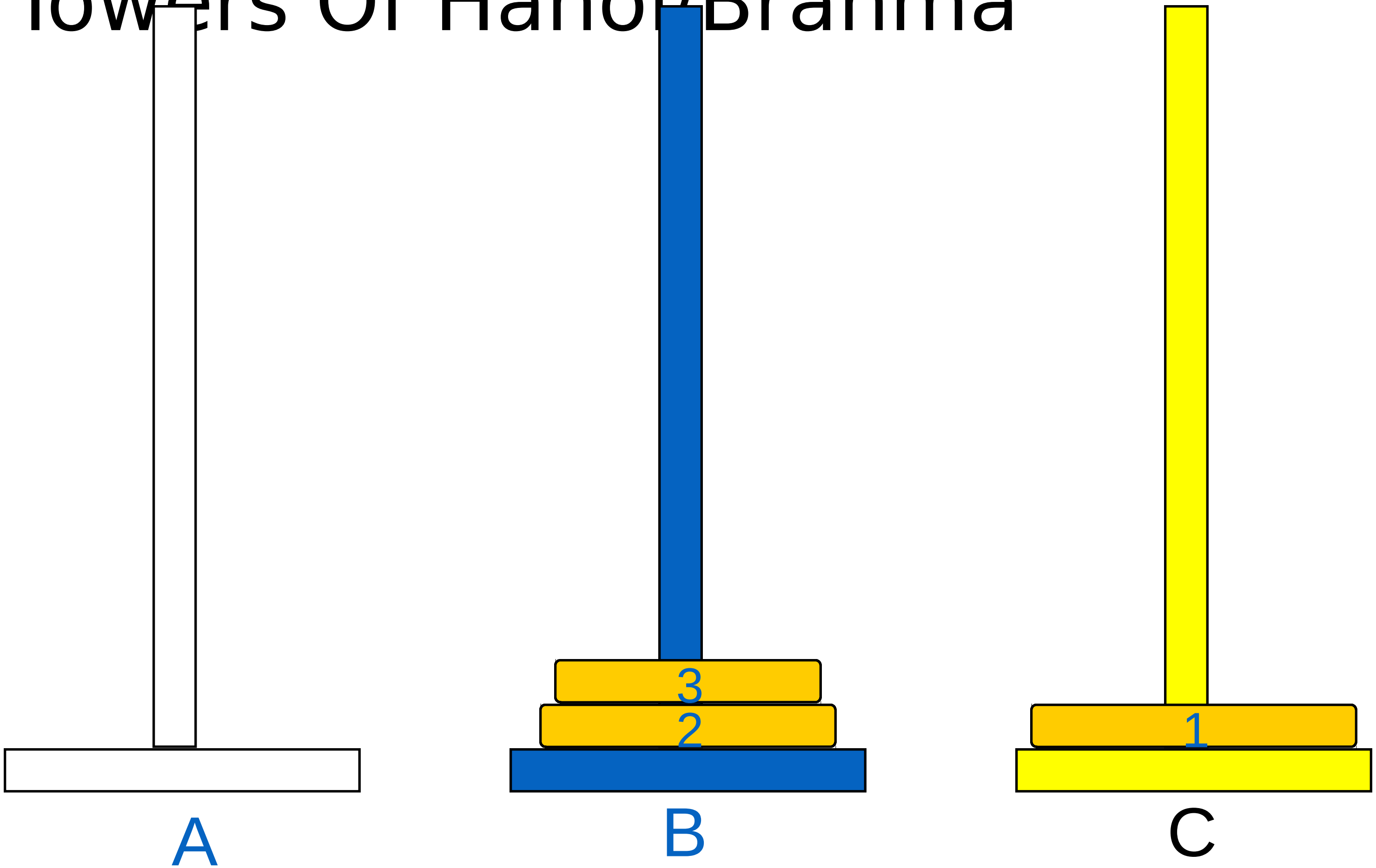
- 3-disk Towers Of Hanoi/Brahma

Towers Of Hanoi/Brahma



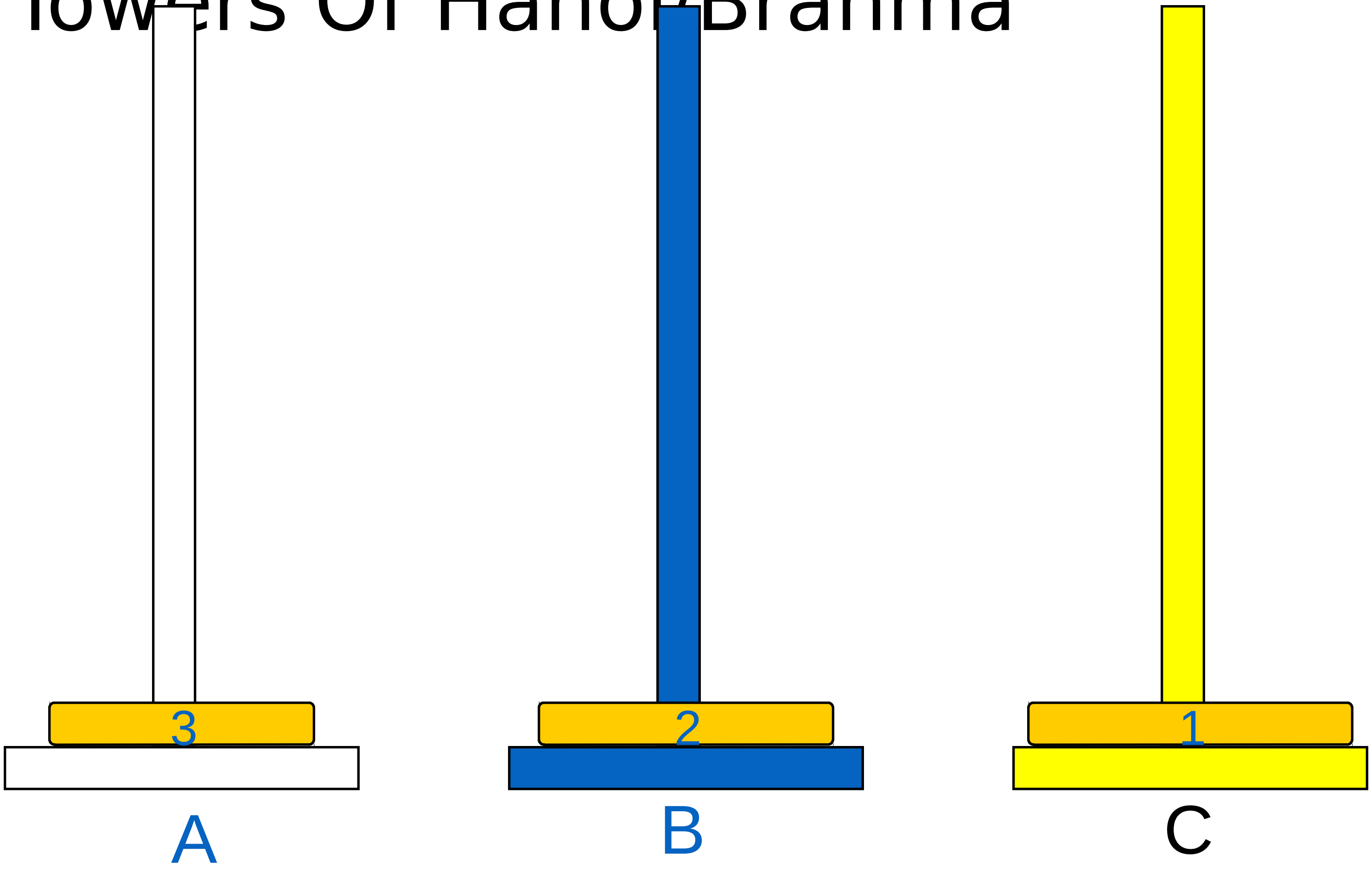
- 3-disk Towers Of Hanoi/Brahma

Towers Of Hanoi/Brahma



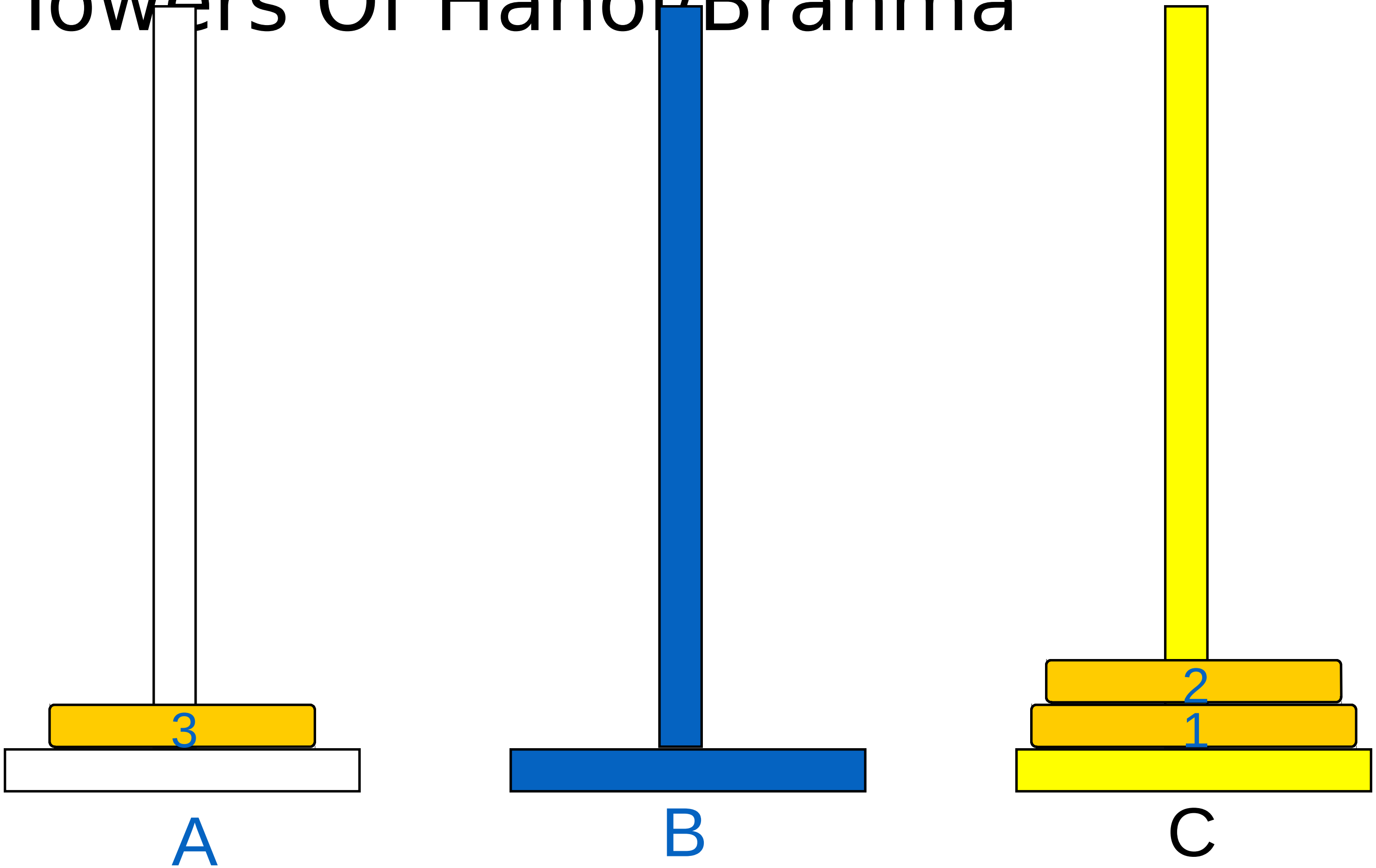
- 3-disk Towers Of Hanoi/Brahma

Towers Of Hanoi/Brahma



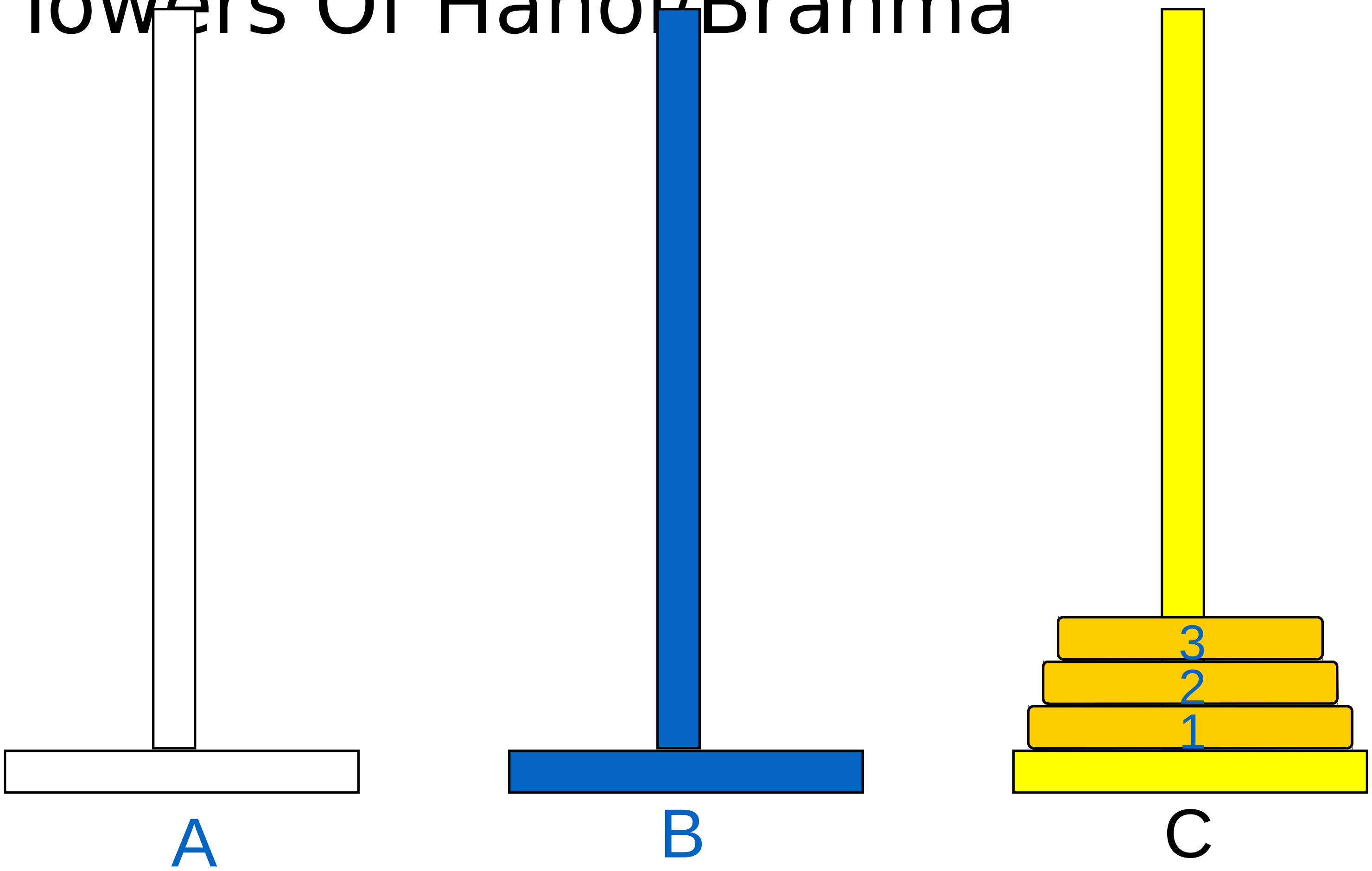
- 3-disk Towers Of Hanoi/Brahma

Towers Of Hanoi/Brahma



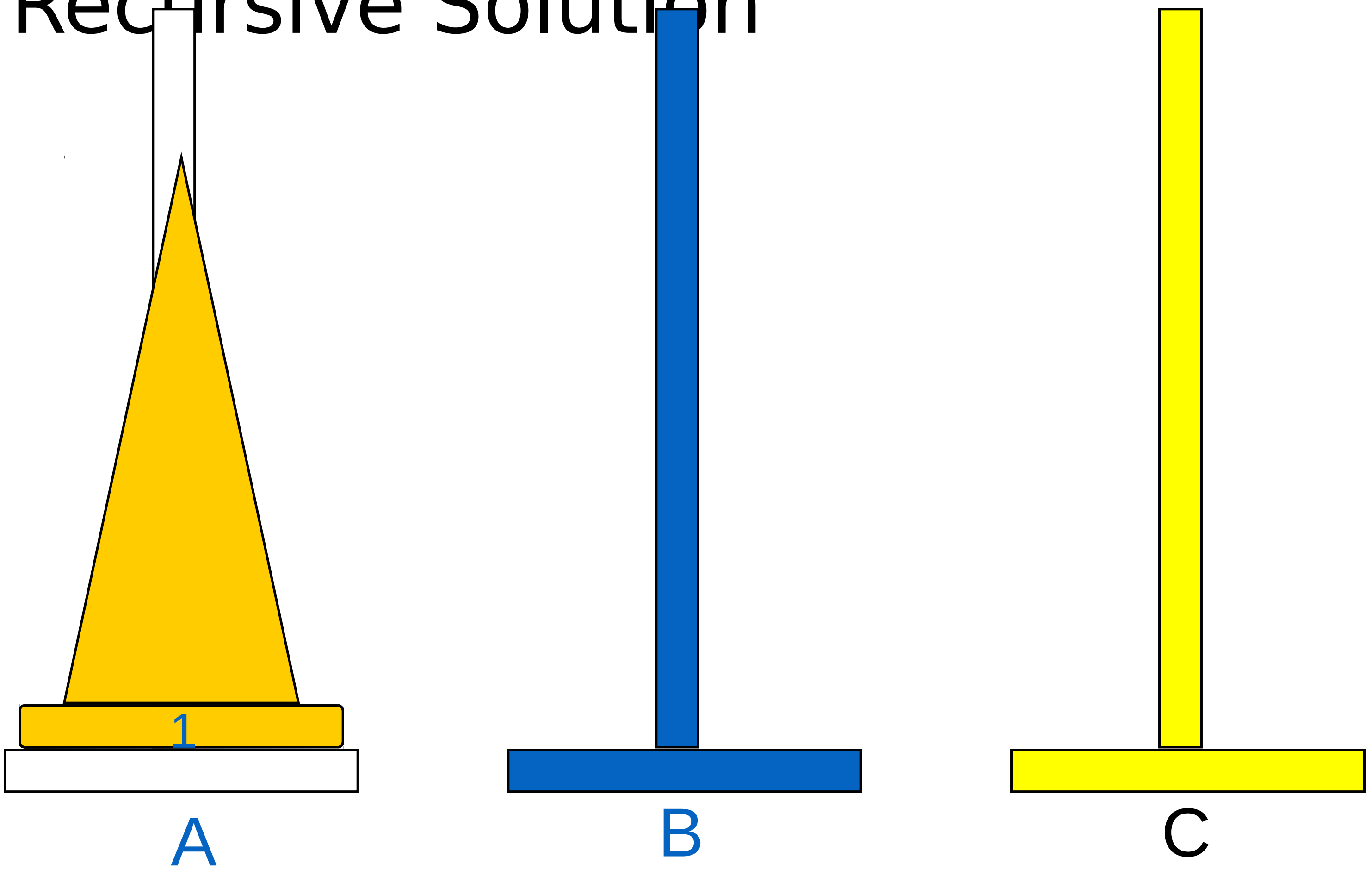
- 3-disk Towers Of Hanoi/Brahma

Towers Of Hanoi/Brahma



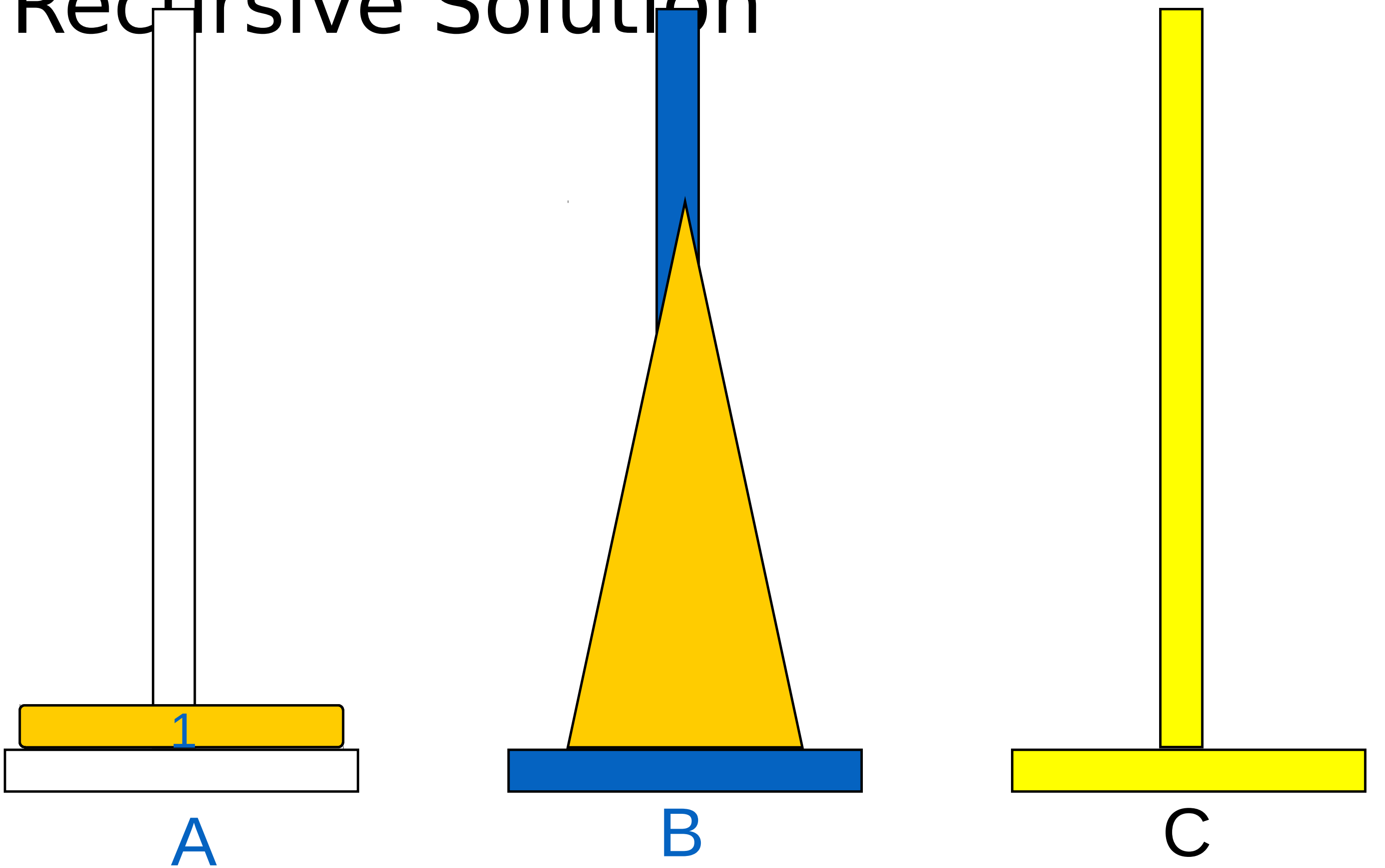
- 3-disk Towers Of Hanoi/Brahma
- 7 disk moves

Recursive Solution



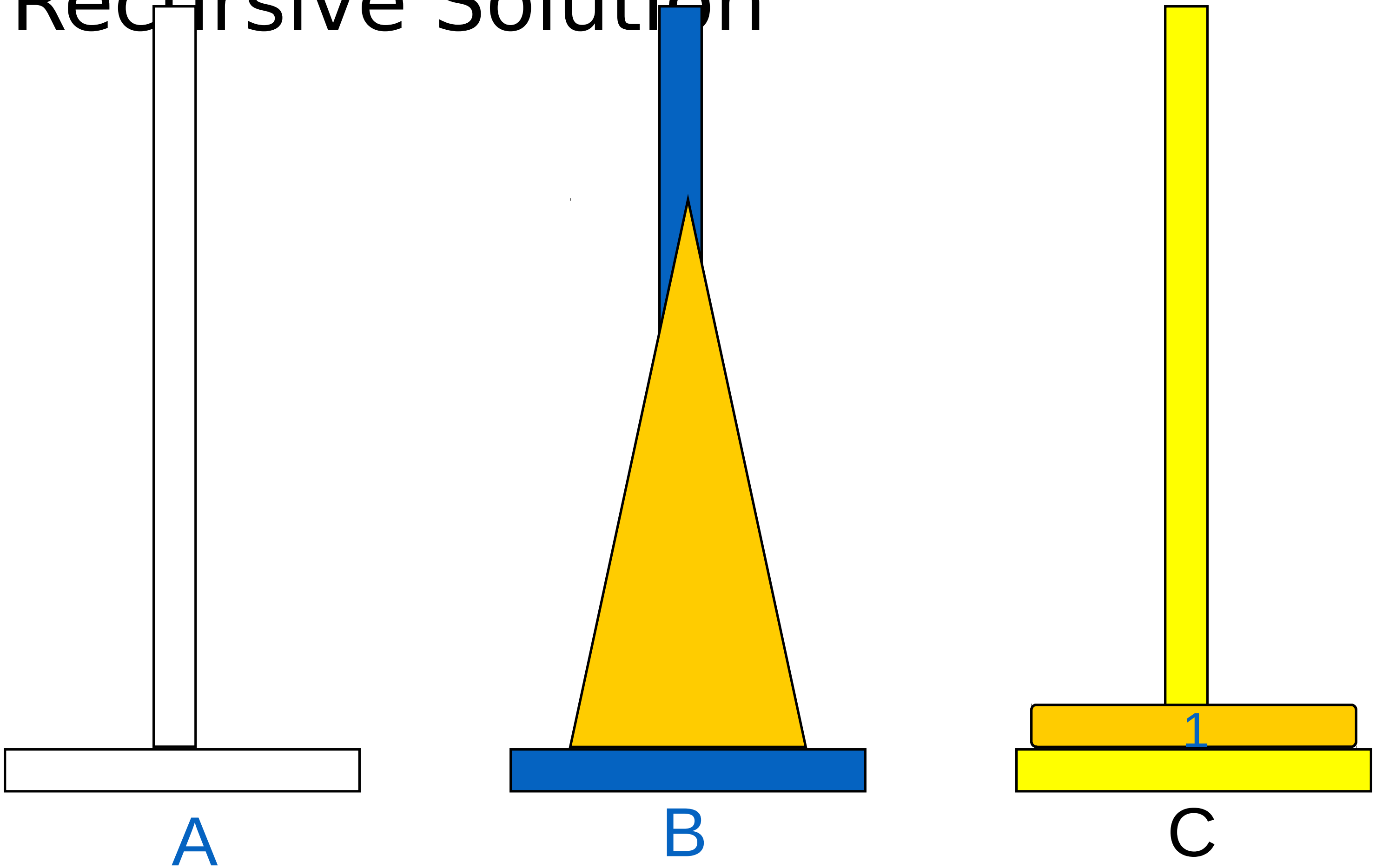
- n > 0 gold disks to be moved from A to C using B
- move top n-1 disks from A to B using C

Recursive Solution



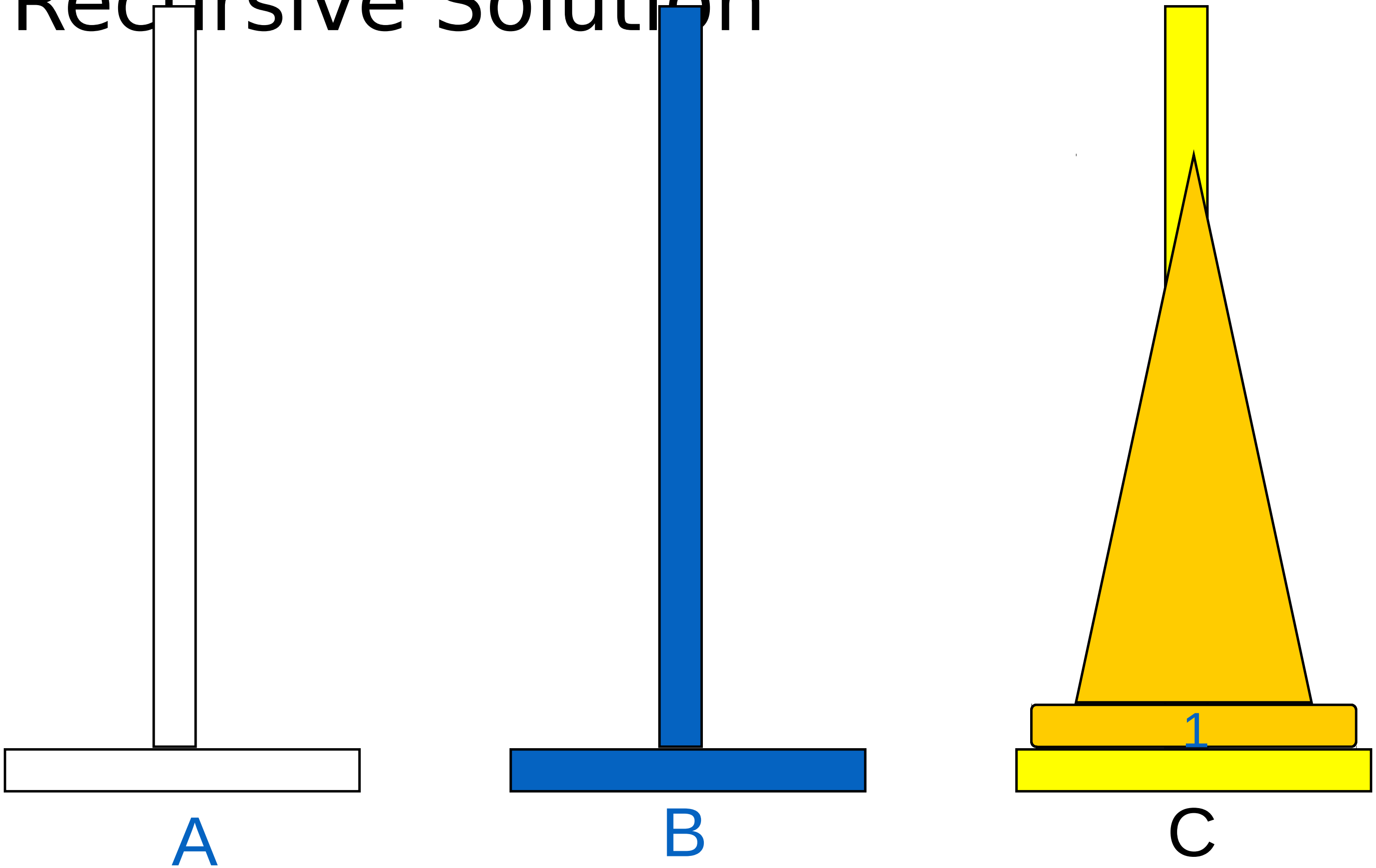
- move top disk from A to C

Recursive Solution



• move top n-1 disks from B to C using A

Recursive Solution



- moves(n) = 0 when n = 0
- moves(n) = 2*moves(n-1) + 1 = 2^n-1 when n > 0

Towers Of Hanoi/Brahma

- moves(64) = $1.8 * 10^{19}$ (approximately)
- Performing 10^9 moves/second, a computer would take about 570 years to complete.
- At 1 disk move/min, the monks will take about $3.4 * 10^{13}$ years.

Queues

- Linear list.
- One end is called front.
- Other end is called rear.
- Additions are done at the rear only.
- Removals are made from the front only.

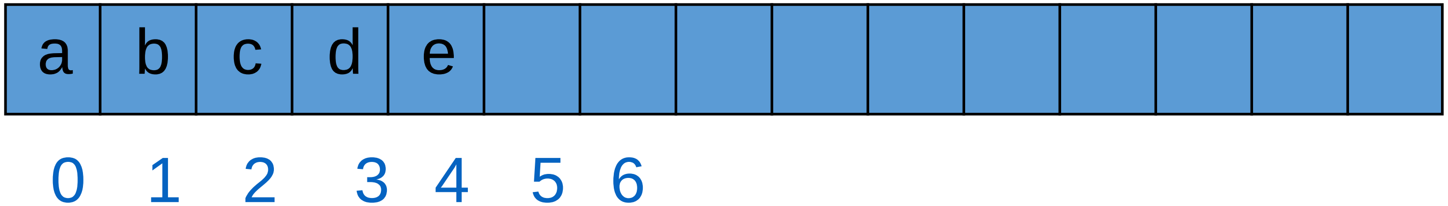
Queue Operations

- IsEmpty … return true iff queue is empty
- Front … return front element of queue
- Rear … return rear element of queue
- Push … add an element at the rear of the queue
- Pop … delete the front element of the queue

Queue in an Array

- Use a 1D array to represent a queue.
- Suppose queue elements are stored with the front element in queue[0], the next in queue[1], and so on.

Derive From arrayList



Pop() => delete queue[0]

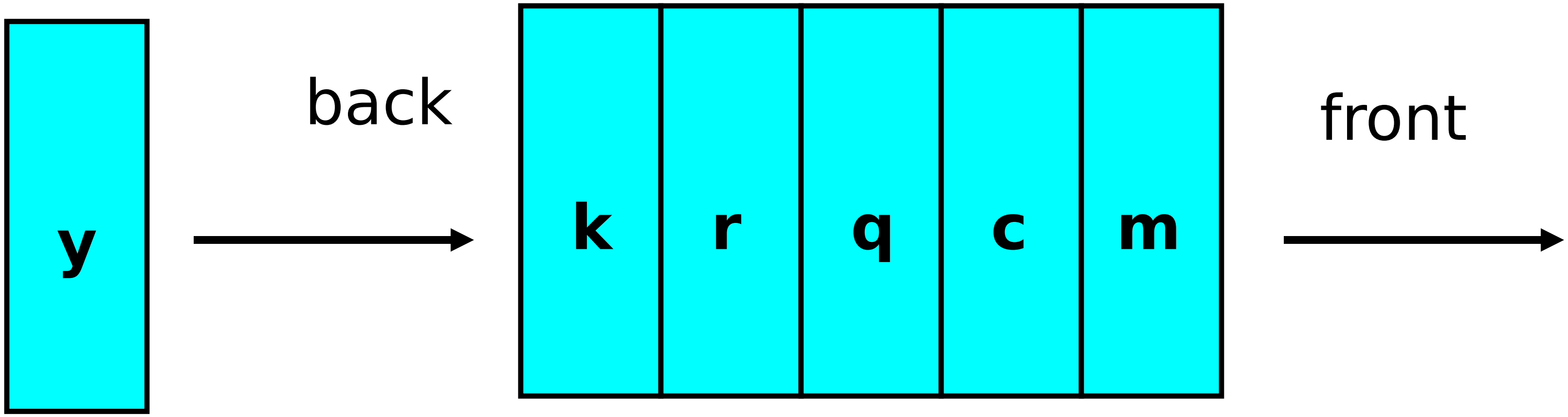
- O(queue size) time

Push(x) => if there is capacity, add at right end

- O(1) time

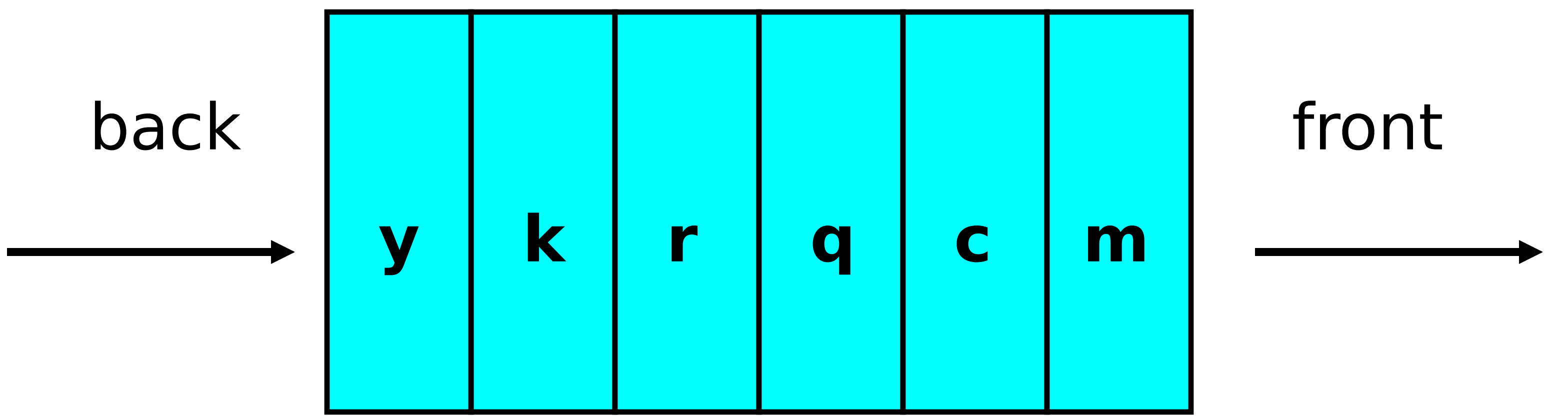
Queues are FIFO

Enqueue operation:



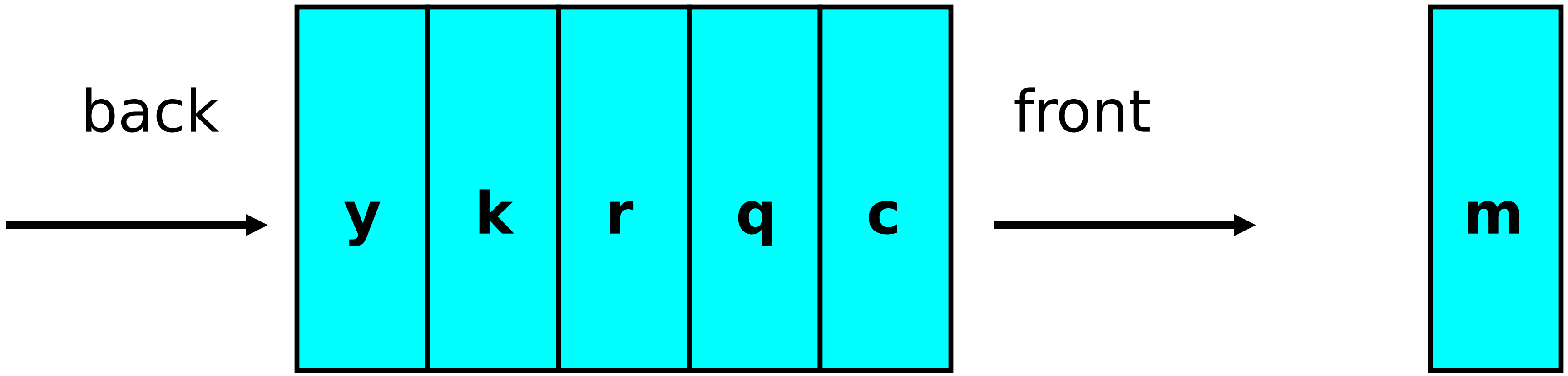
Queues are FIFO

Enqueue operation:



Queues are FIFO

Dequeue operation:



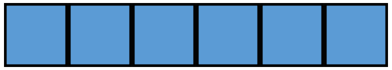
O(1) Pop and Push

- to perform each opertion in O(1) time (excluding array doubling), we use a circular representation.

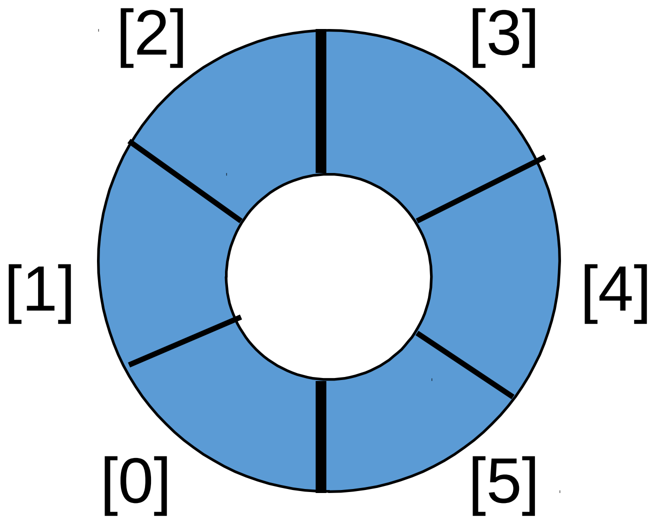
Custom Array Queue

- Use a 1D array queue.

queue[]

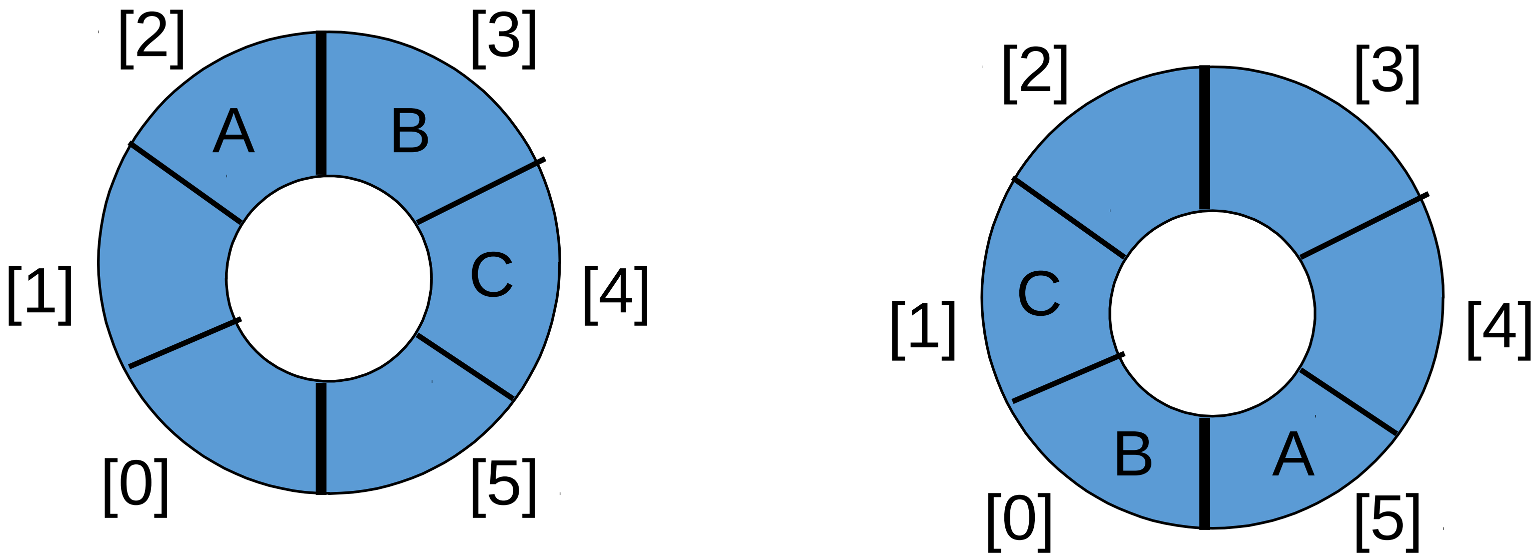


- Circular view of array.



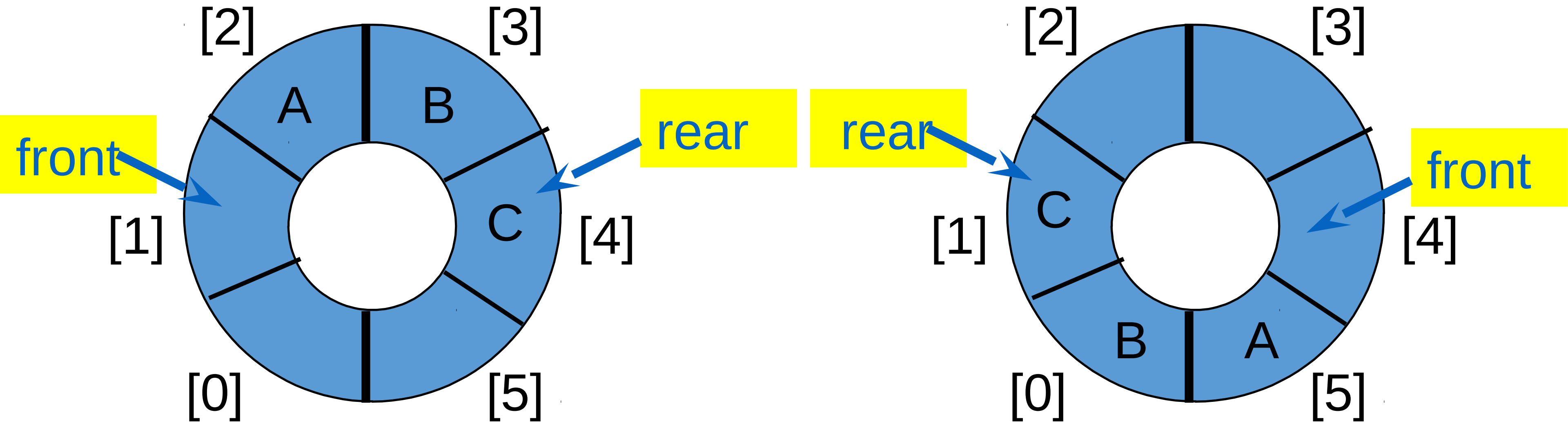
Custom Array Queue

- Two of the possible configuration with 3 elements.



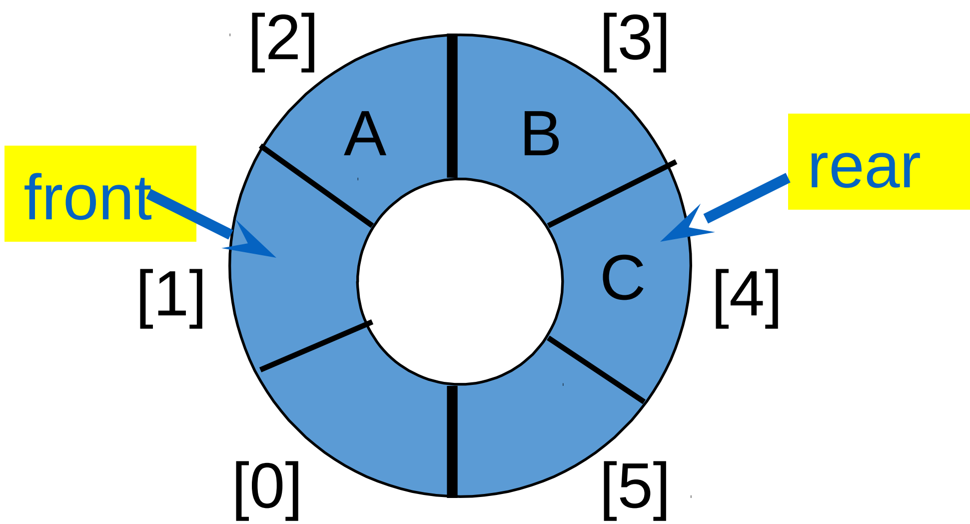
Custom Array Queue

- Use integer variables front and rear.
  - front is one position counterclockwise from first element
  - rear gives position of last element



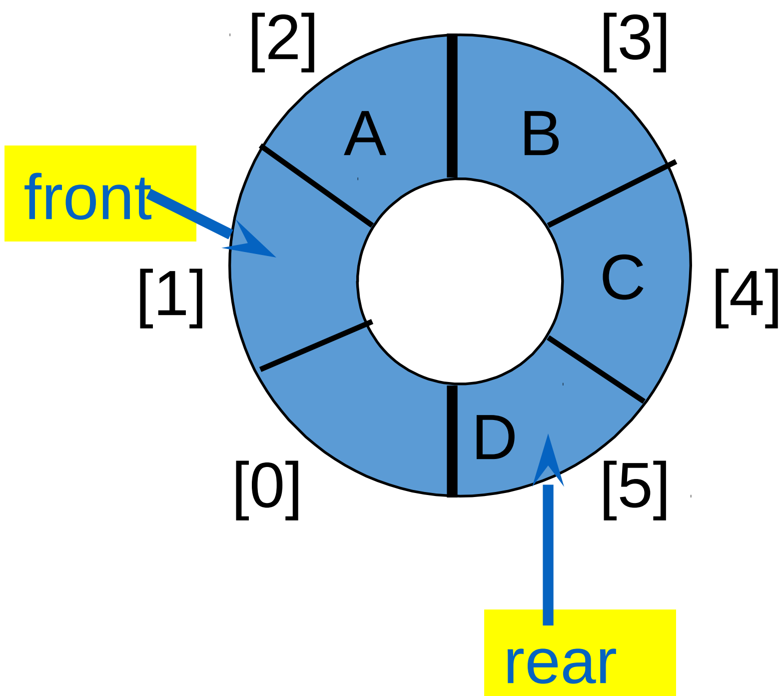
Push An Element

- Move rear one clockwise.



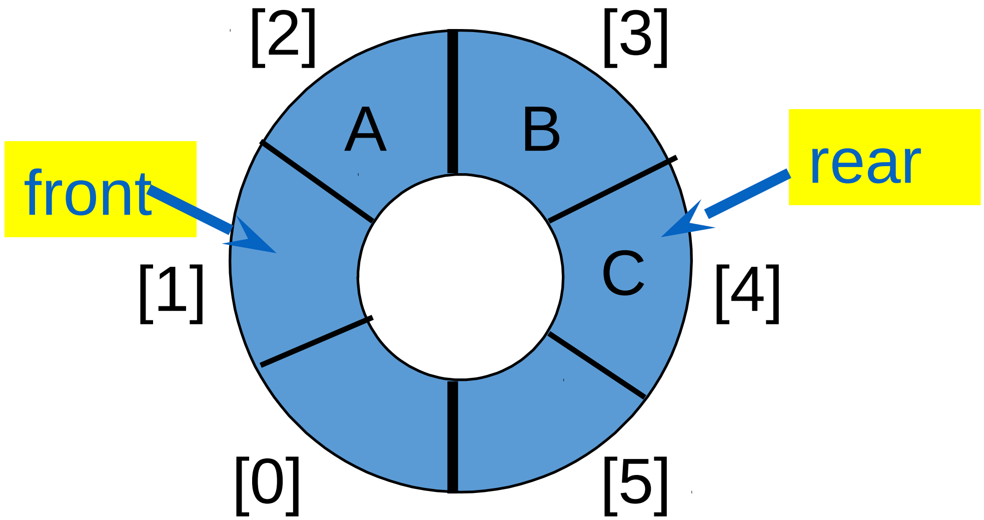
Push An Element

- Move rear one clockwise.
- Then put into queue[rear].



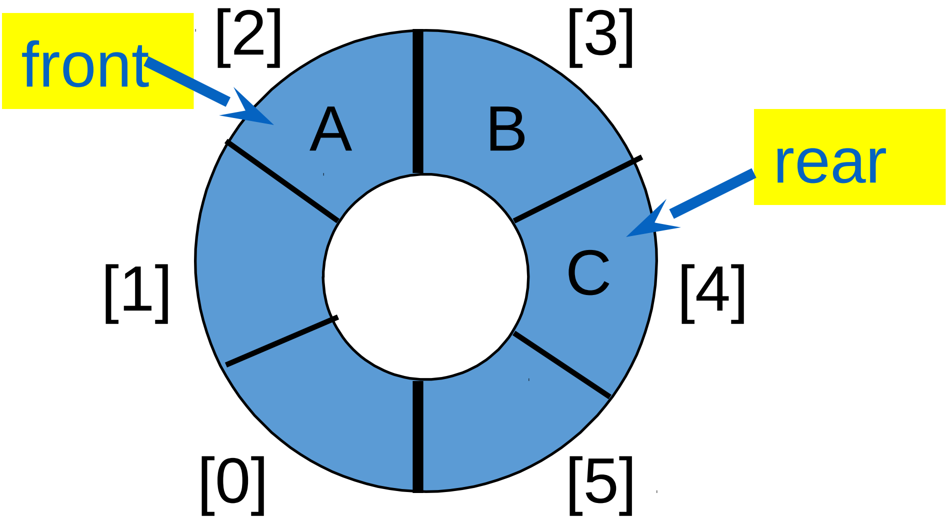
Pop An Element

- Move front one clockwise.



Pop An Element

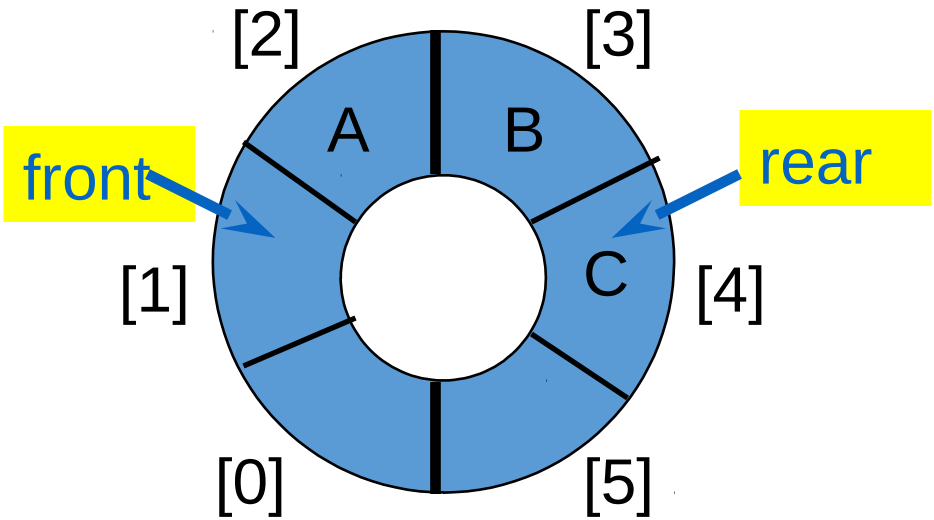
- Move front one clockwise.
- Then extract from queue[front].



Moving rear Clockwise

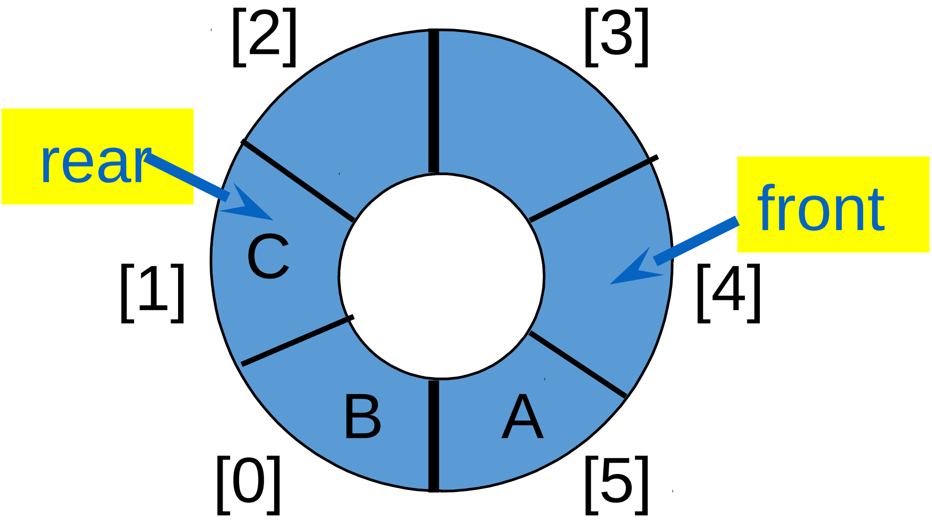
- rear++;

  if (rear = = capacity) rear = 0;

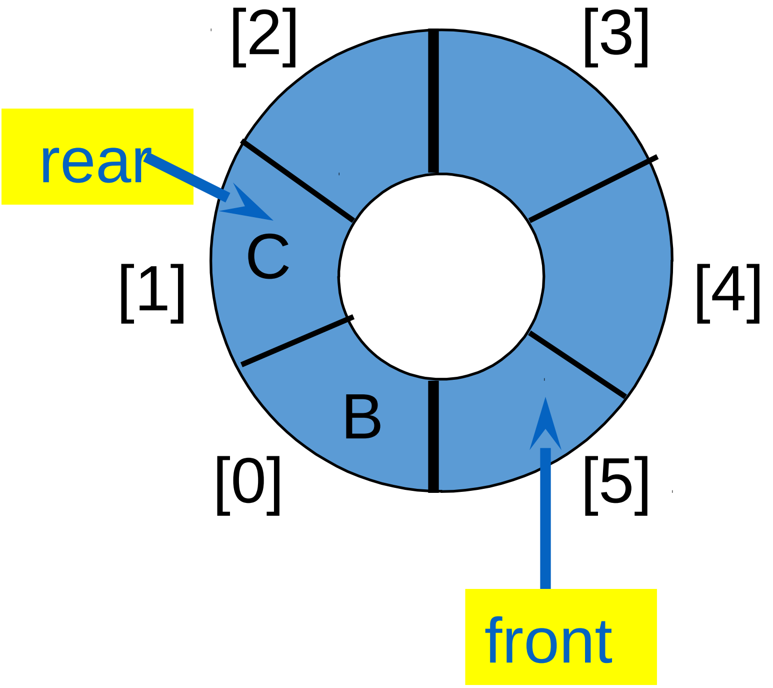


- rear = (rear + 1) % capacity;

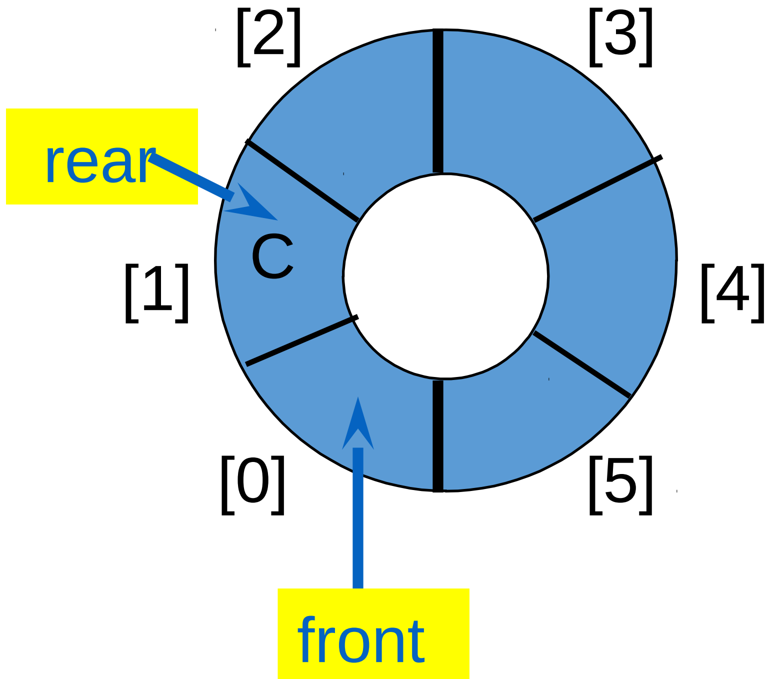
Empty That Queue



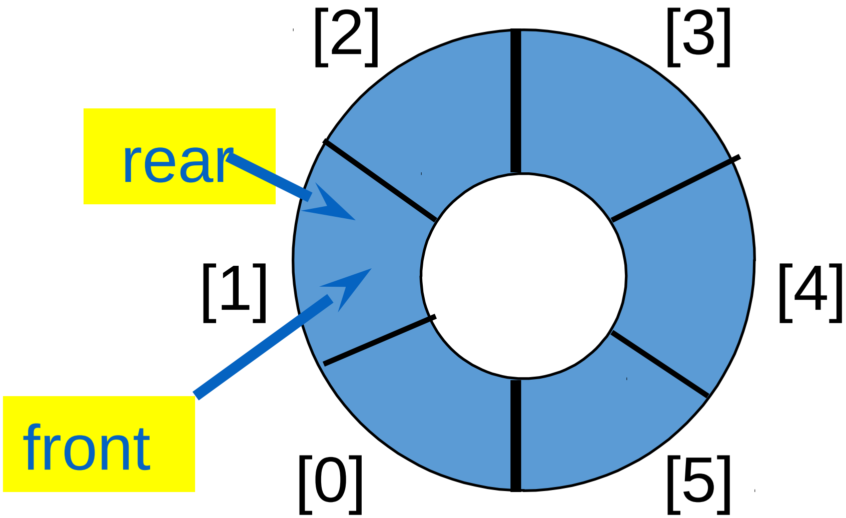
Empty That Queue



Empty That Queue

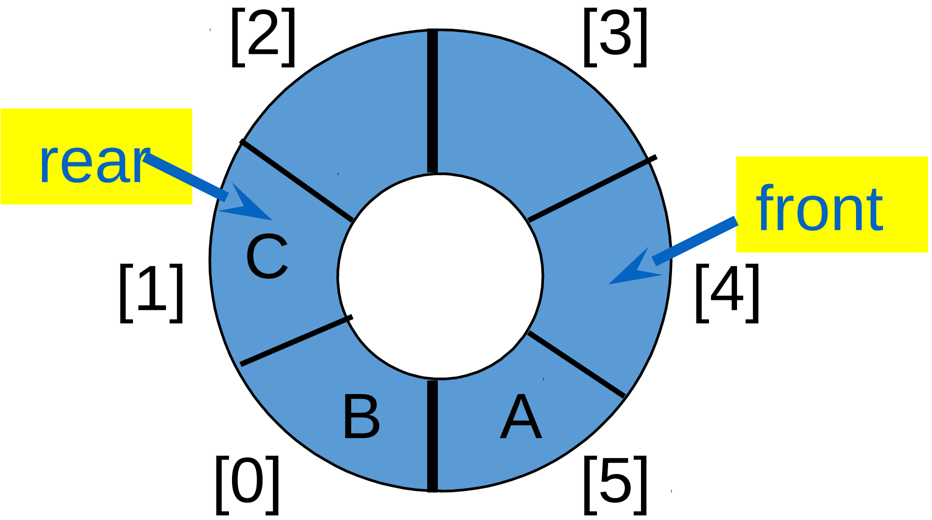


Empty That Queue

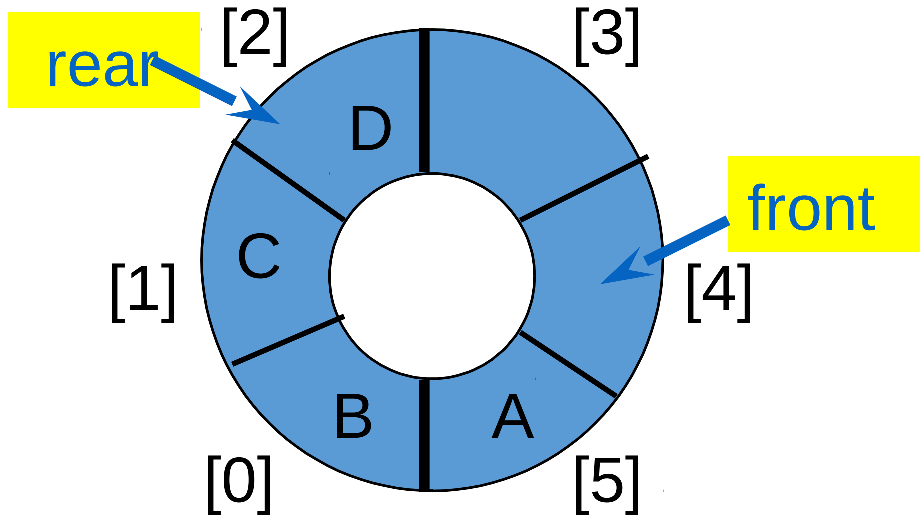


- When a series of removes causes the queue to become empty, front = rear.
- When a queue is constructed, it is empty.
- So initialize front = rear = 0.

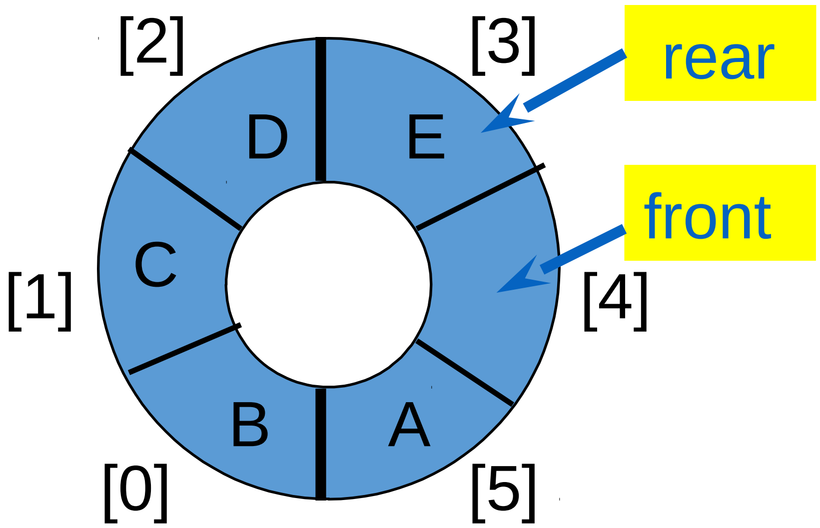
A Full Tank



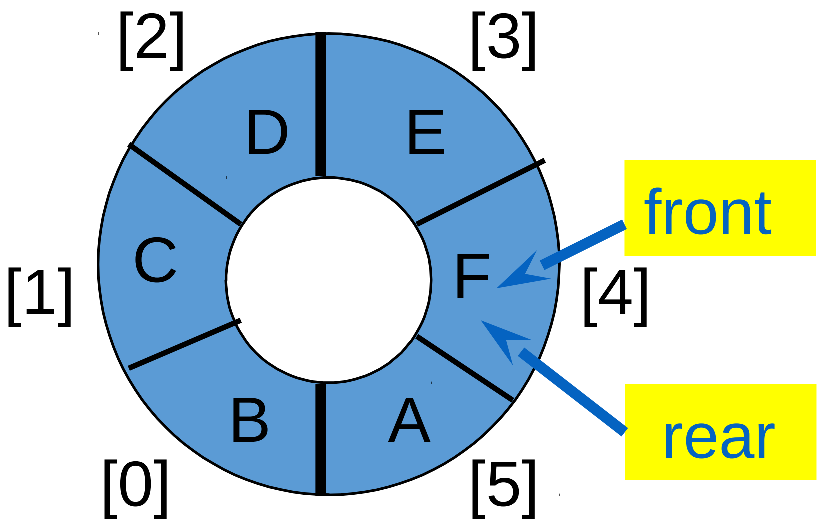
A Full Tank



A Full Tank



A Full Tank



- When a series of adds causes the queue to become full, front = rear.
- So we cannot distinguish between a full queue and an empty queue!

#	STACK	QUEUE
1	Objects are inserted and removed at the same end.	Objects are inserted and removed from different ends.
2	In stacks only one pointer is used. It points to the top of the stack.	In queues, two different pointers are used for front and rear ends.
3	In stacks, the last inserted object is first to come out.	In queues, the object inserted first is first deleted.
4	Stacks follow Last In First Out (LIFO) order.	Queues following First In First Out (FIFO) order.
5	Stack operations are called push and pop.	Queue operations are called enqueue and dequeue.
6	Stacks are visualized as vertical collections.	Queues are visualized as horizontal collections.