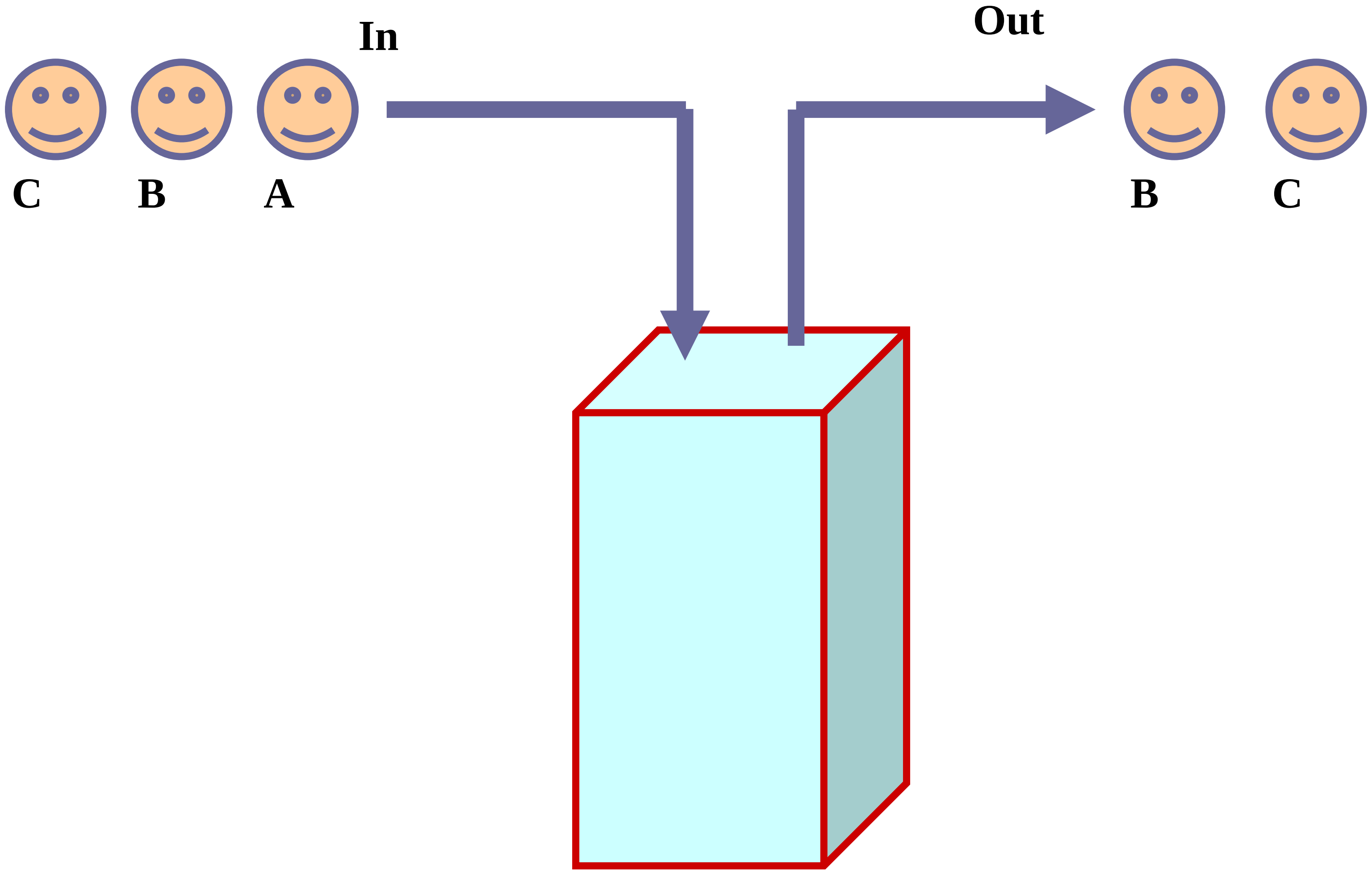
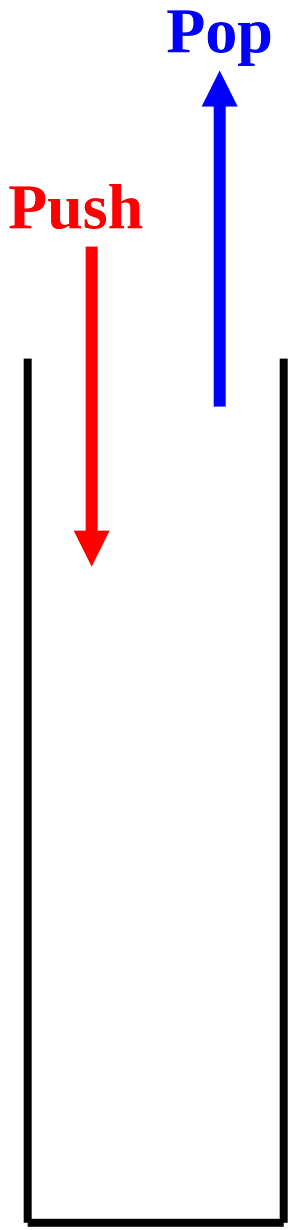
Stack and Queue

Stack

Data structure with Last-In First-Out (LIFO) behavior



Typical Operations on Stack



isempty: determines if the stack has no elements

isfull: determines if the stack is full in case of a bounded sized stack

top: returns the top element in the stack

push: inserts an element into the stack

pop: removes the top element from the stack

push is like inserting at the front of the list

pop is like deleting from the front of the list

Creating and Initializing a Stack

Declaration

```
#define MAX_STACK_SIZE 100
typedef struct {
    int key; /* just an example, can have
                   any type of fields depending
                   on what is to be stored */
}  element;
typedef struct {
     element list[MAX_STACK_SIZE];
     int top; /* index of the topmost element */
 } stack;
```

Create and Initialize

```
stack Z;
Z.top = -1;
```

Operations

```
int isfull (stack *s)
{
    if (s->top >=
              MAX_STACK_SIZE – 1)
        return 1;
    return 0;
}
```

```
int isempty (stack *s)
{
    if (s->top == -1)
        return 1;
    return 0;
}
```

Operations

```
element top( stack *s )
{
     return s->list[s->top];
}
```

```
void pop( stack *s )
{
     (s->top)--;
}
```

```
void push( stack *s, element e )
{
    (s->top)++;
    s->list[s->top] = e;
 }
```

Application: Parenthesis Matching

- Given a parenthesized expression, test whether the expression is properly parenthesized
  - Examples:

( )({ } [({ } { } ( ))])	is proper
( ){ [ ]	is not proper
({) }	is not proper
)([ ]	is not proper
([ ]))	is not proper

- Approach:
  - Whenever a left parenthesis is encountered, it is pushed in the stack
  - Whenever a right parenthesis is encountered, pop from stack and check if the parentheses match
  - Works for multiple types of parentheses ( ), { }, [ ]

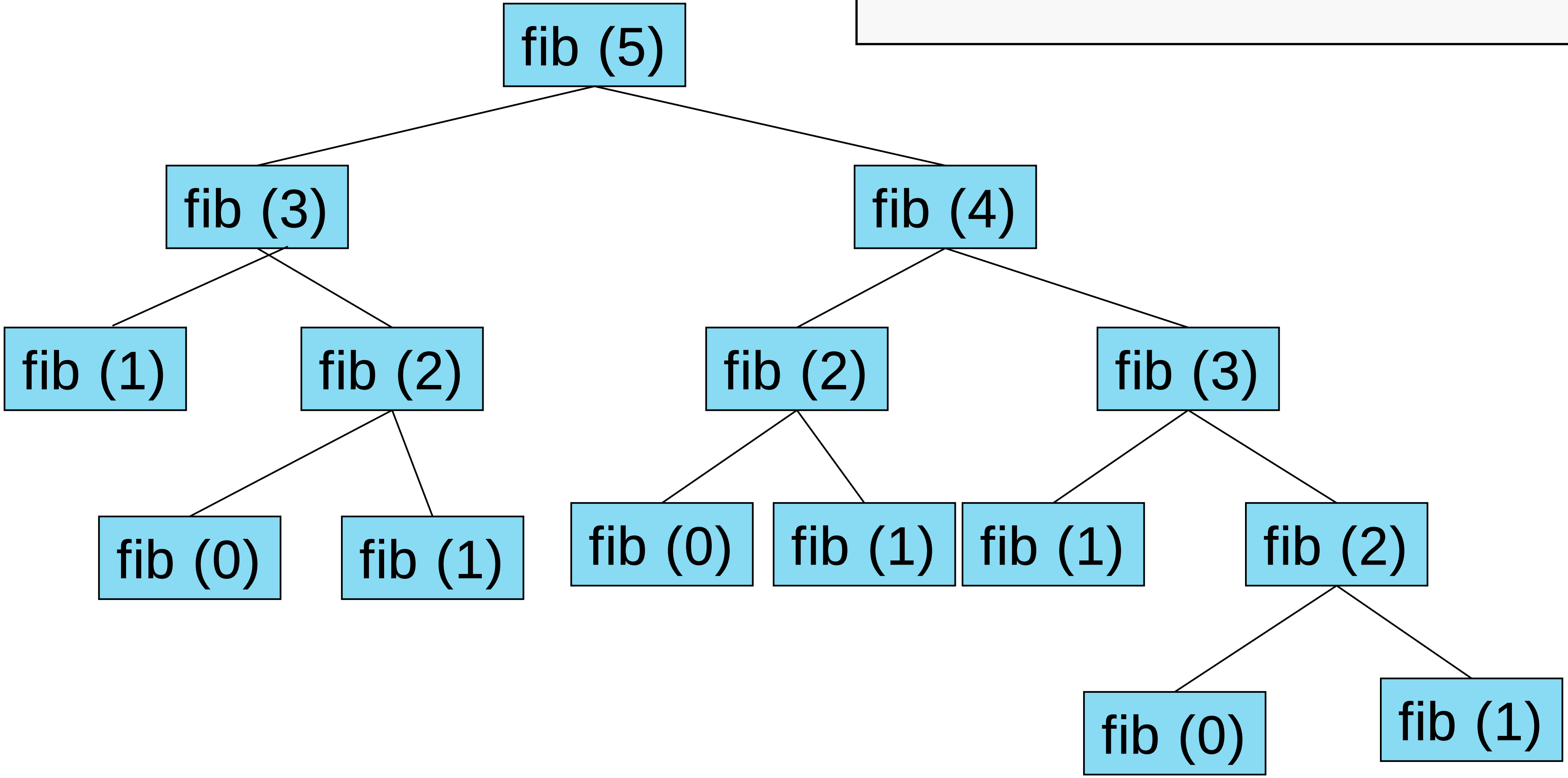
Parenthesis matching

```
while (not end of string) do
{
    a = get_next_token();
    if (a is ‘(‘ or ‘{‘ or ‘[‘) push (a);
    if (a is ‘)’ or ‘}’ or ‘]’)
    {
          if (is_stack_empty( ))
            { print (“Not well formed”); exit(); }
          x = top();
          pop();
          if (a and x do not match)
            { print (“Not well formed”); exit(); }
    }
}
if (not is_stack_empty( )) print (“Not well formed”);
```

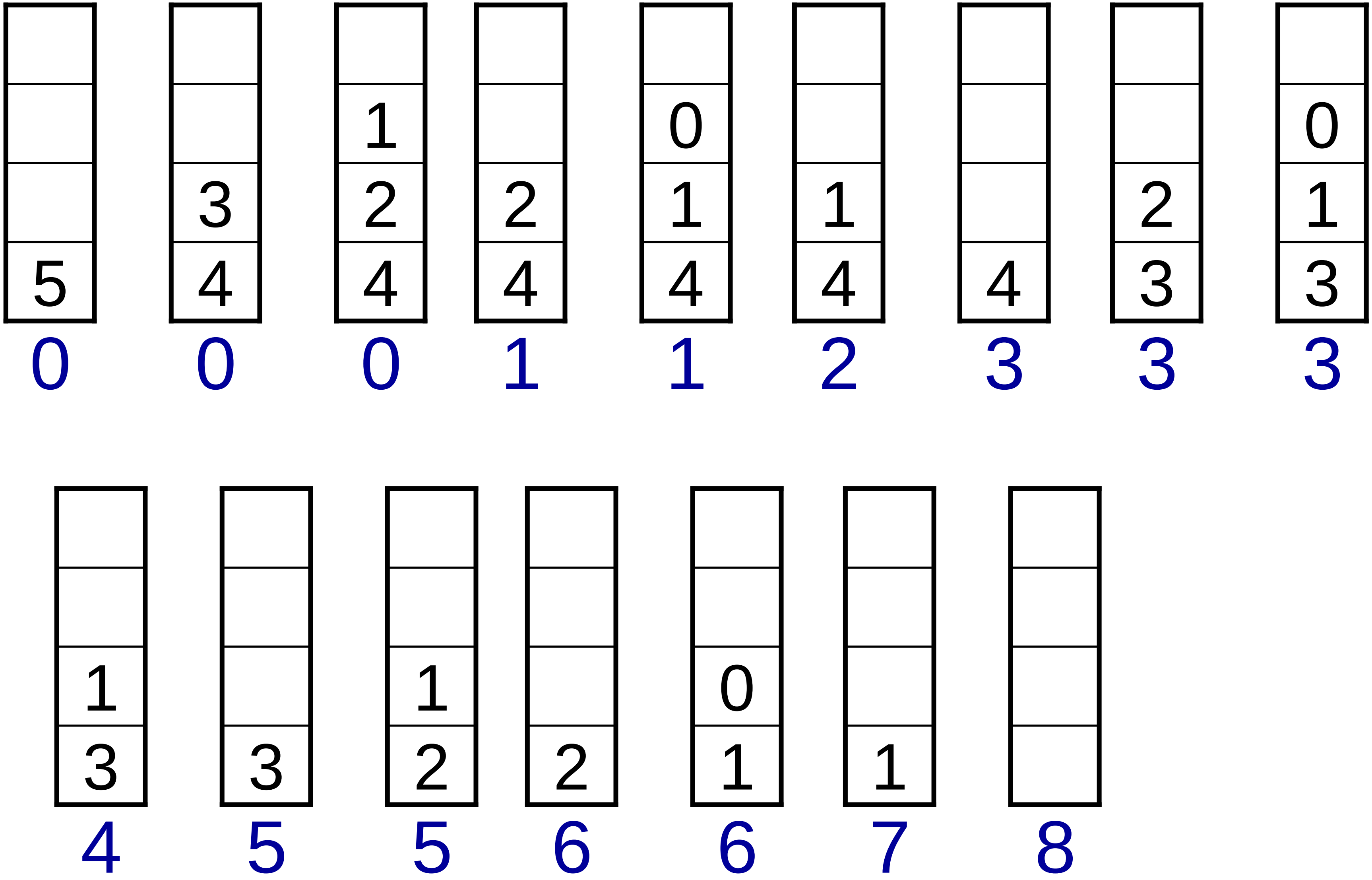
Recursion can be implemented as a stack

Fibonacci recurrence:

fib(n) = 1 if n =0 or 1;
= fib(n – 2) + fib(n – 1)
otherwise;



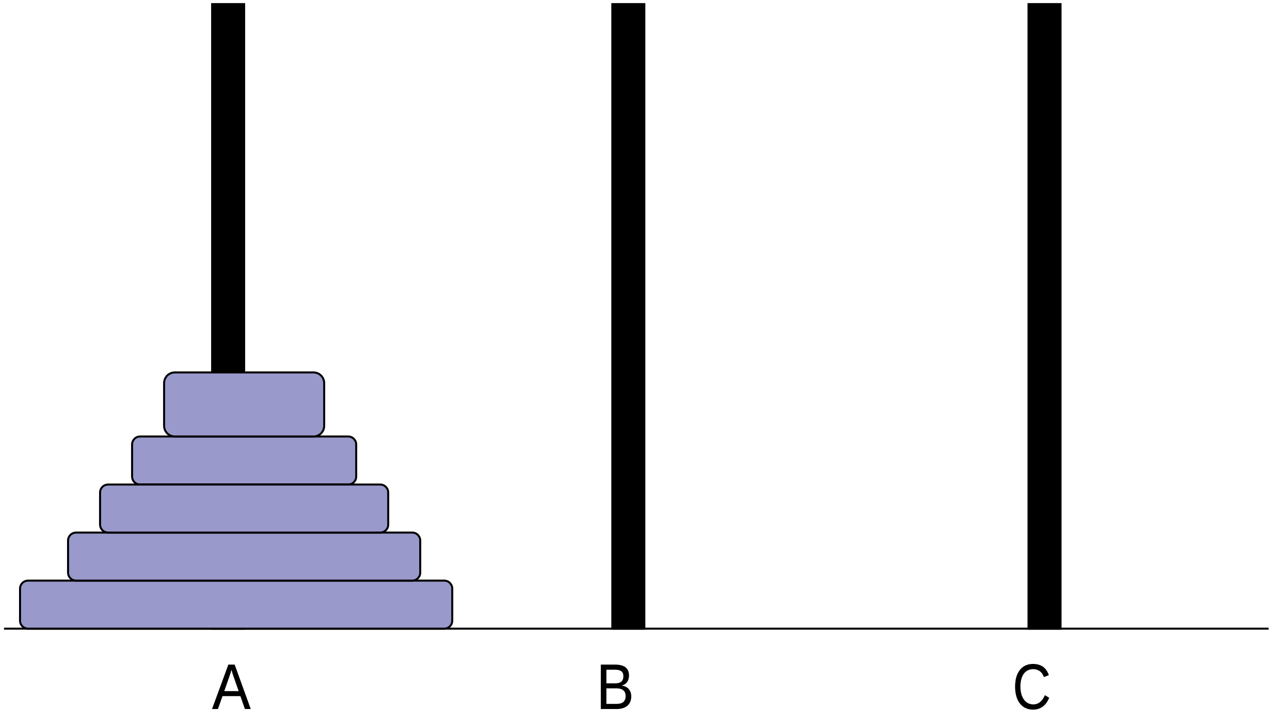
Fibonacci Recursion Stack



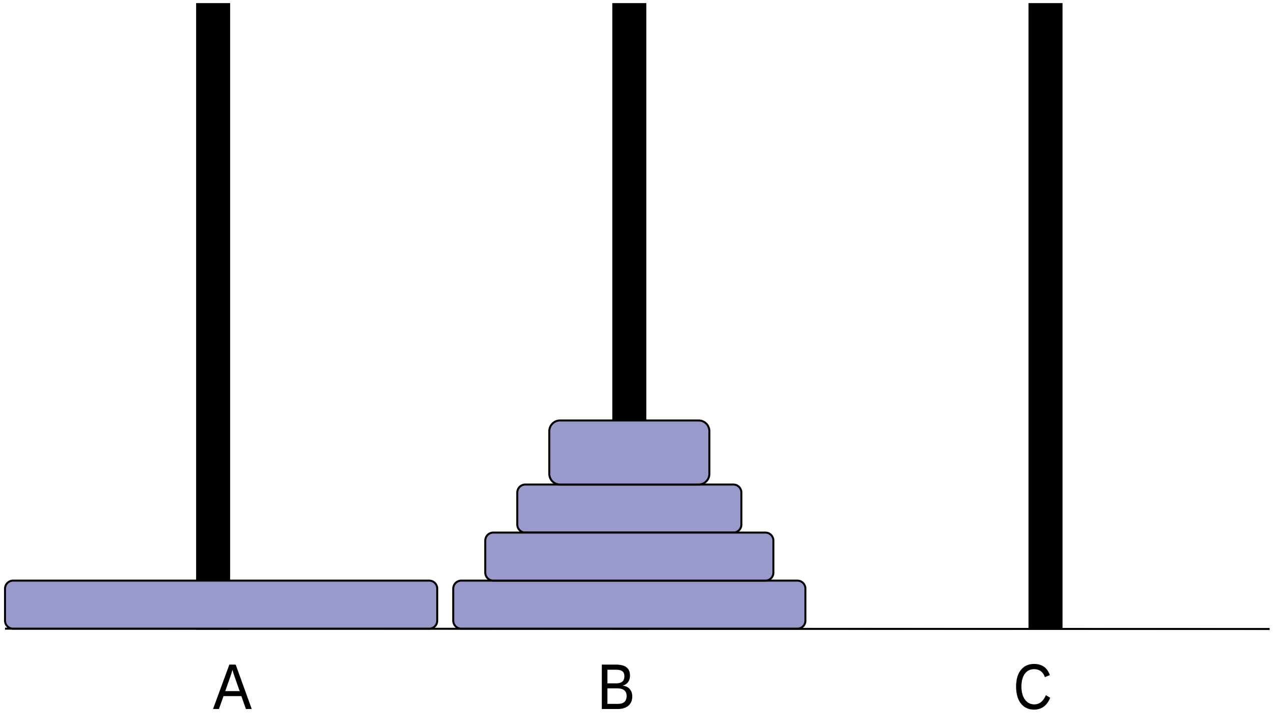
		1		0				0
	3	2	2	1	1		2	1
5	4	4	4	4	4	4	3	3
0	0	0	1	1	2	3	3	3

1		1		0		
3	3	2	2	1	1	
4	5	5	6	6	7	8

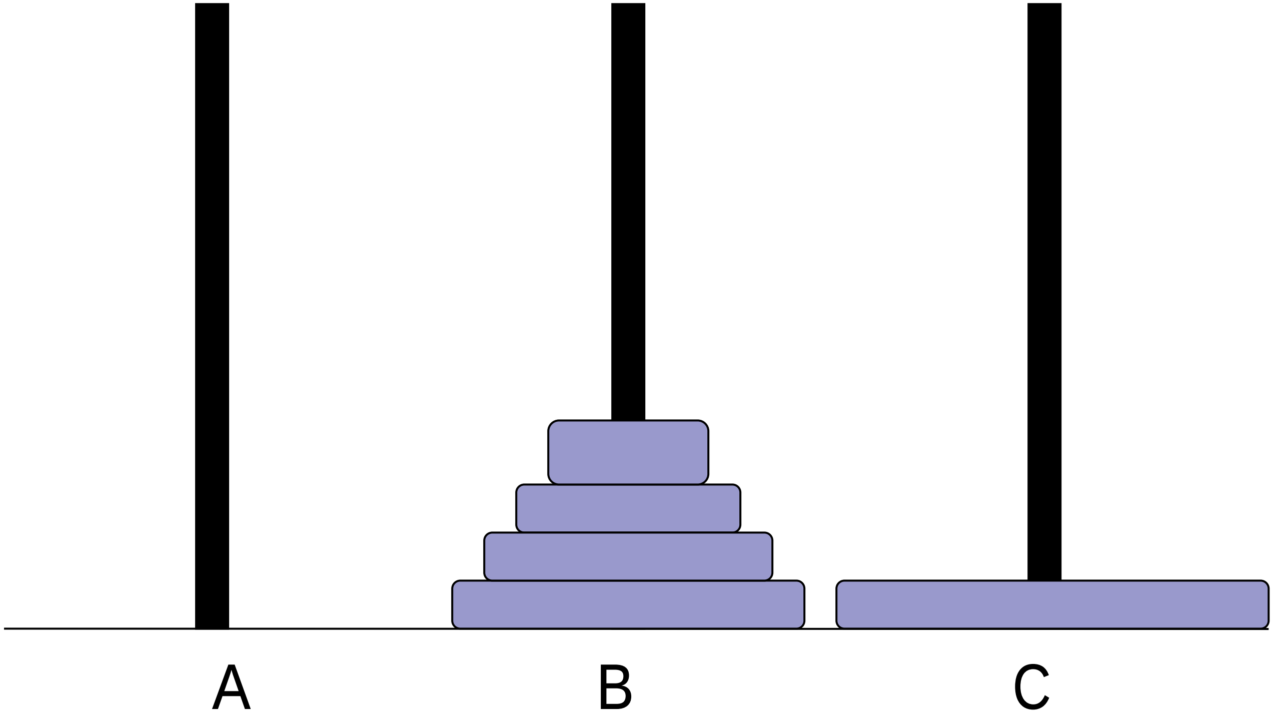
Tower of Hanoi



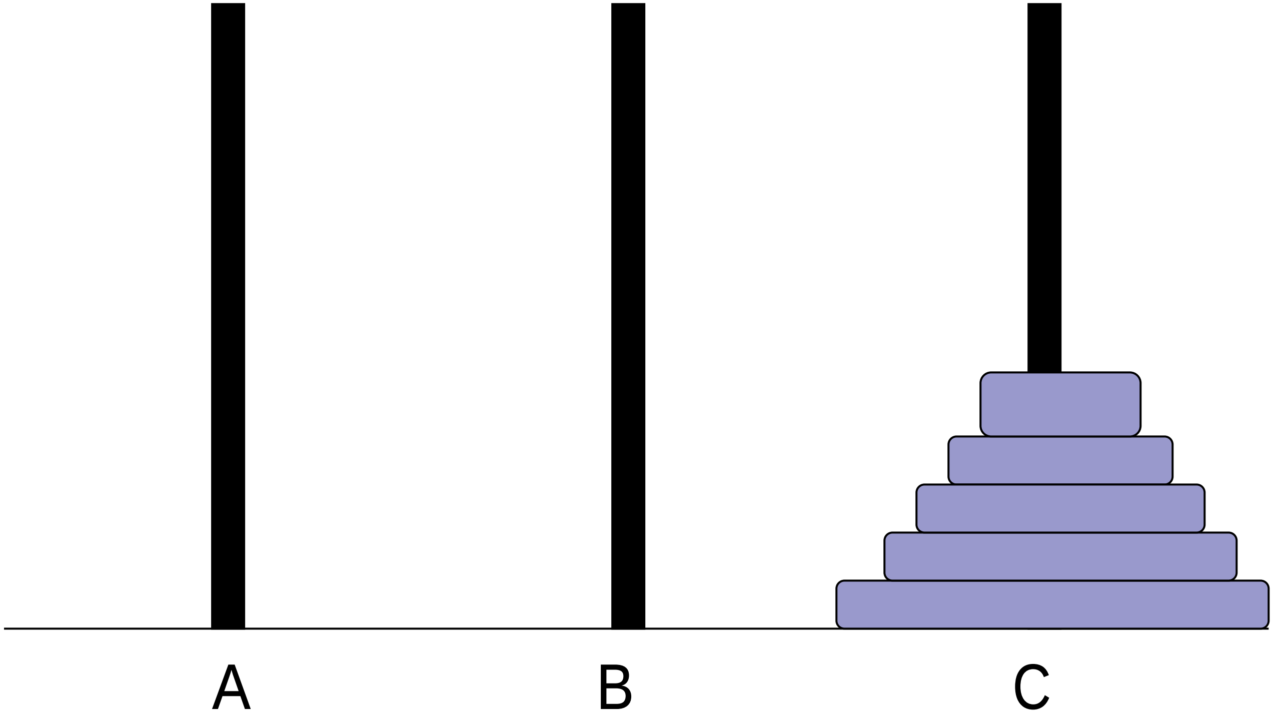
Tower of Hanoi



Tower of Hanoi



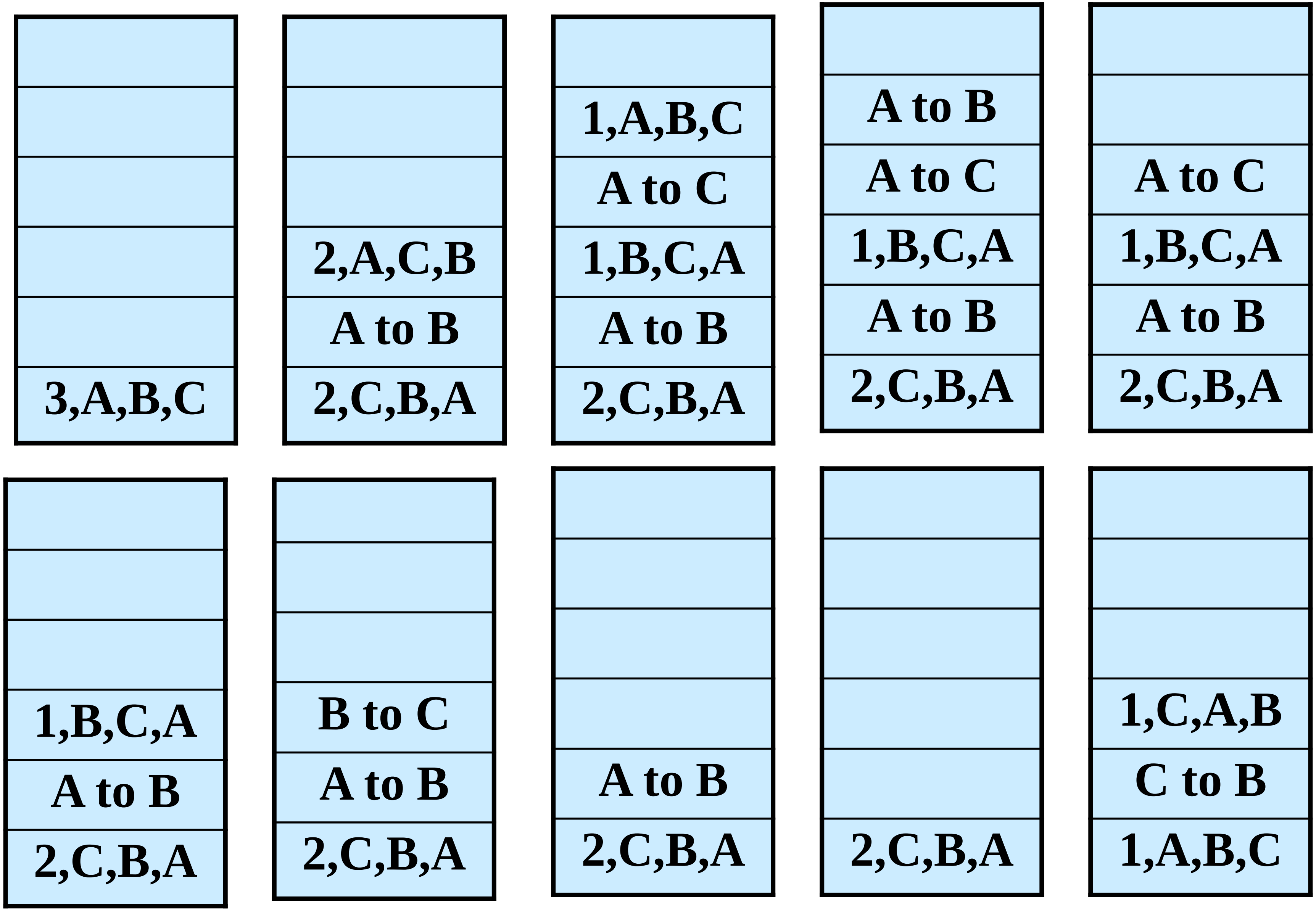
Tower of Hanoi



Towers of Hanoi Function

```
void towers (int n, char from, char to, char aux)
{
    /* Base Condition */
     if (n==1)   {
         printf (“Disk 1 : %c -> %c \n”, from, to) ;
         return ;
     }
    /* Recursive Condition */
      towers (n-1, from, aux, to) ;
      printf (“Disk %d : %c -> %c\n”, n, from, to) ;
      towers (n-1, aux, to, from) ;
}
```

TOH Recursion Stack



3,A,B,C

2,A,C,B
A to B
2,C,B,A

1,A,B,C
A to C
1,B,C,A
A to B
2,C,B,A

A to B
A to C
1,B,C,A
A to B
2,C,B,A

A to C
1,B,C,A
A to B
2,C,B,A

1,B,C,A
A to B
2,C,B,A

B to C
A to B
2,C,B,A

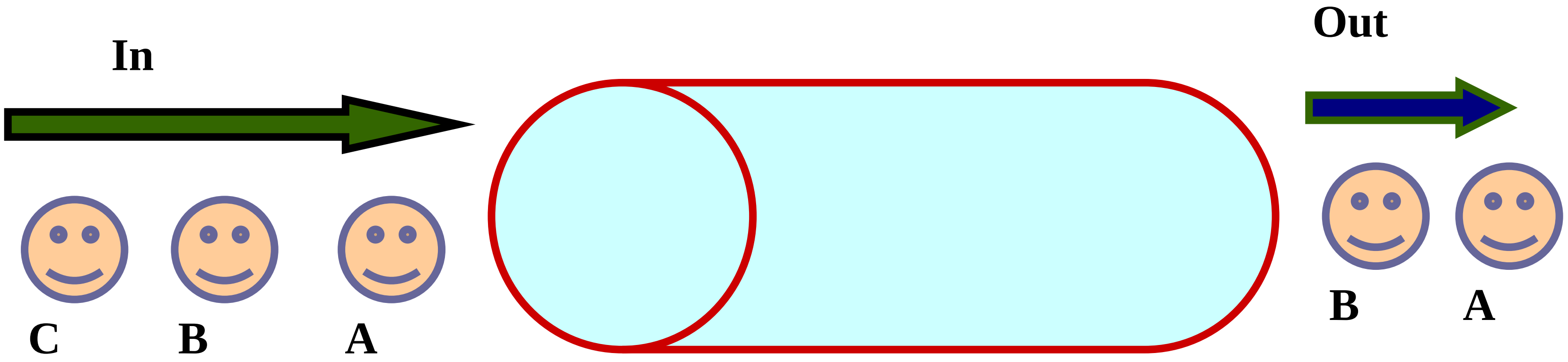
A to B
2,C,B,A

2,C,B,A

1,C,A,B
C to B
1,A,B,C

Queue

Data structure with First-In First-Out (FIFO) behavior



Typical Operations on Queue

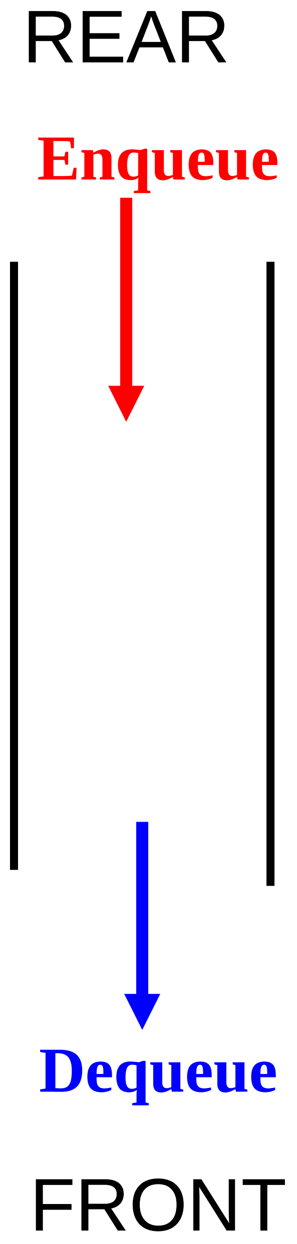
isempty: determines if the queue is empty

isfull: determines if the queue is full in case of a bounded size queue

front: returns the element at front of the queue

enqueue: inserts an element at the rear

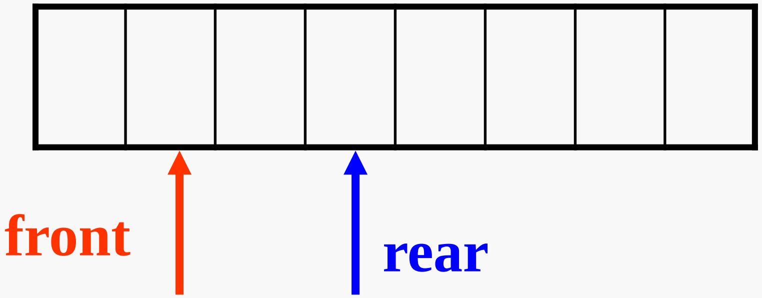
dequeue: removes the element in front



Possible Implementations

Linear Arrays:

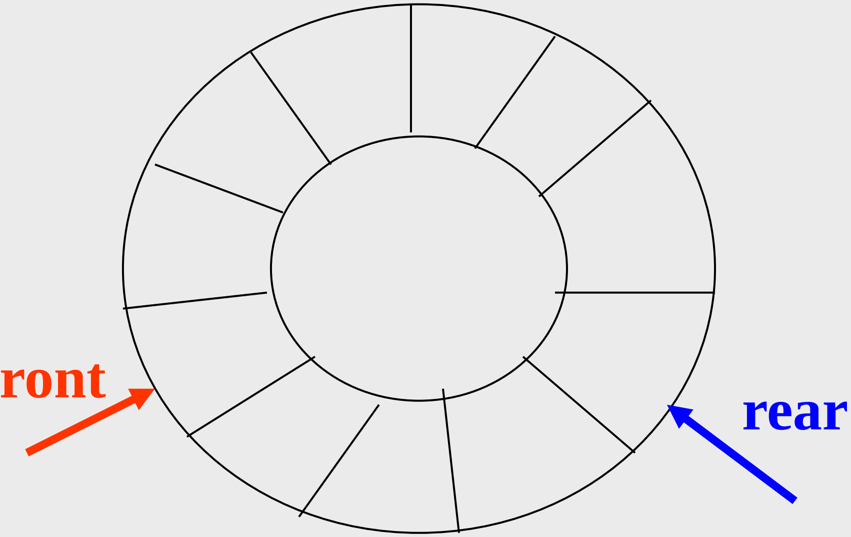
(static/dynamicaly allocated)



Linked Lists: **Use a linear linked list with insert_rear and delete_front operations**

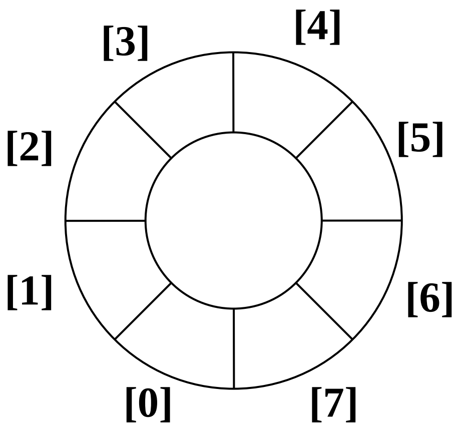
Circular Arrays:

(static/dynamically allocated)



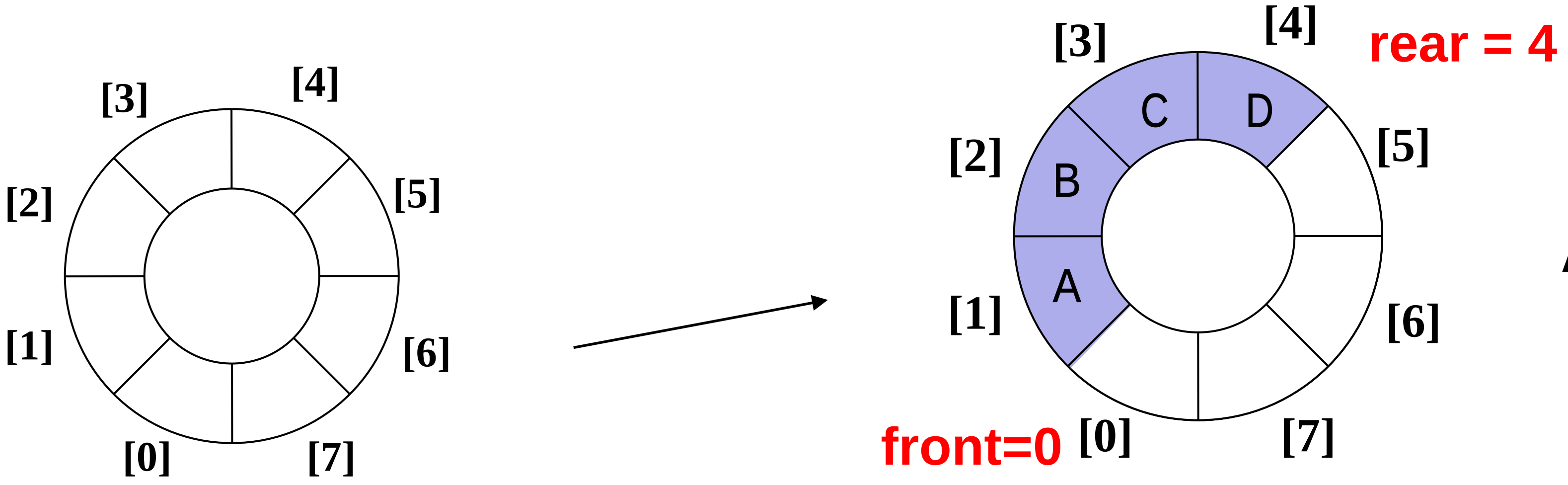
Can be implemented by a 1-d array using modulus operations

Circular Queue



front=0
rear=0

Circular Queue



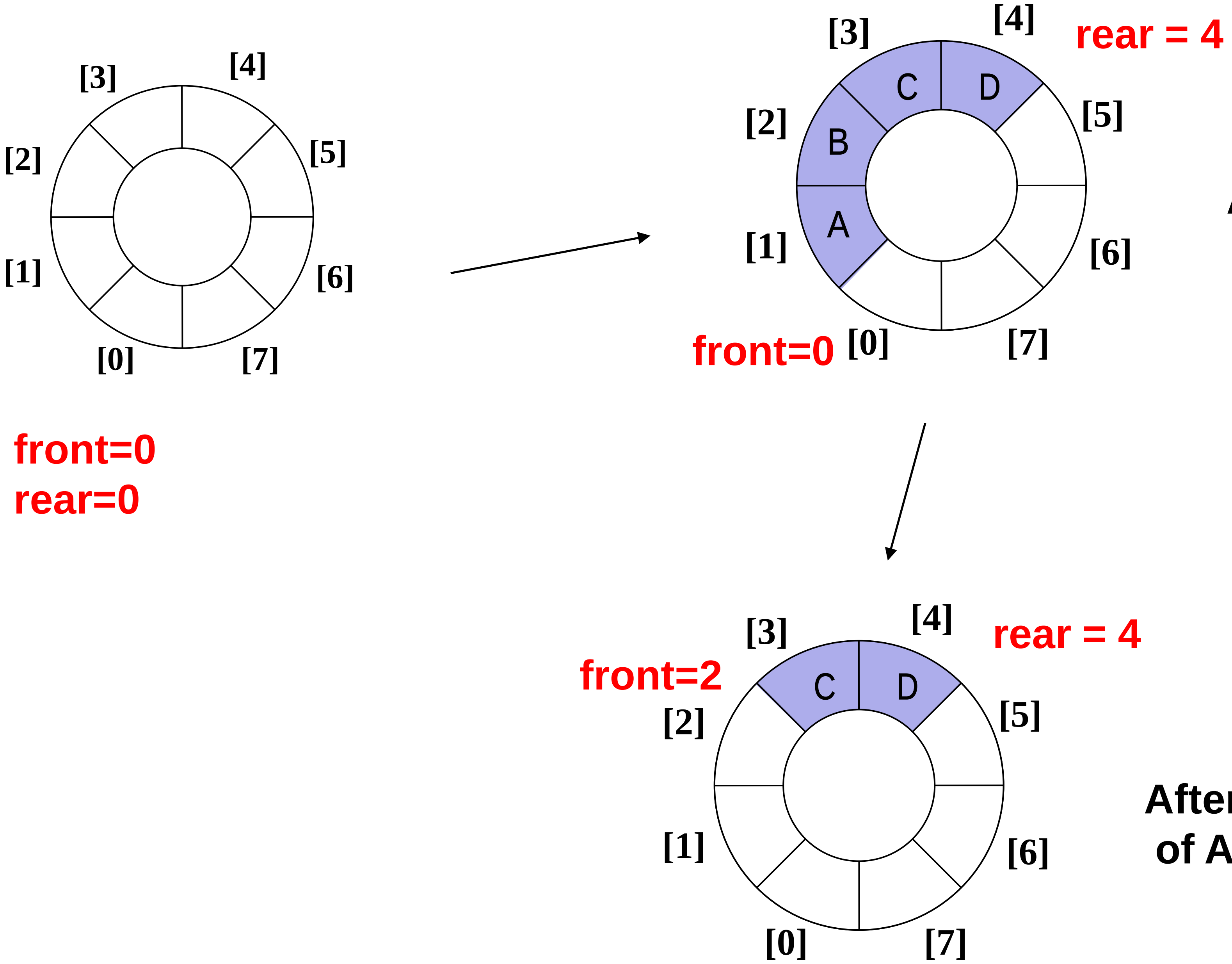
rear = 4

front=0

After insertion of A, B, C, D

front=0
rear=0

Circular Queue



front=0
rear=0

rear = 4

front=0

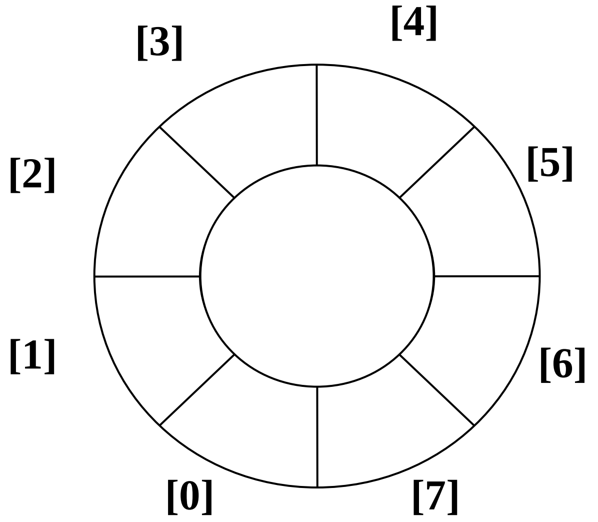
After insertion of A, B, C, D

rear = 4

front=2

After deletion of of A, B

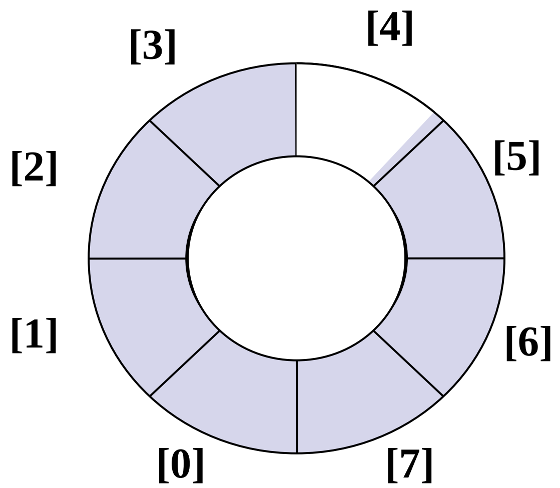
front: index of queue-head (always empty – why?)
rear: index of last element, unless rear = front



front=0
rear=0

Queue Empty

rear = 3 front=4



Queue Full

Queue Empty Condition: ***front == rear***
Queue Full Condition: ***front == (rear + 1) % MAX_Q_SIZE***

Creating and Initializing a Circular Queue

Declaration

```
#define MAX_Q_SIZE 100
typedef struct {
    int key; /* just an example, can have
                    any type of fields depending
                    on what is to be stored */
}  element;
typedef struct {
     element list[MAX_Q_SIZE];
     int front, rear;
 } queue;
```

Create and Initialize

```
queue Q;
Q.front = 0;
Q.rear = 0;
```

Operations

```
int isfull (queue *q)
{
    if (q->front == ((q->rear + 1) %
                              MAX_Q_SIZE))
          return 1;
    return 0;
}
```

```
int isempty (queue *q)
{
    if (q->front == q->rear)
        return 1;
    return 0;
}
```

Operations

```
element front( queue *q )
 {
      return q->list[(q->front + 1) % MAX_Q_SIZE];
 }
```

```
void enqueue( queue *q, element e)
 {
    q->rear = (q->rear  + 1)%
                          MAX_Q_SIZE;
    q->list[q->rear] = e;
 }
```

```
void dequeue( queue *q )
 {
    q-> front =
           (q-> front + 1)%
                    MAX_Q_SIZE;
 }
```

Exercises

- Implement the Queue as a linked list.
- Implement a Priority Queue which maintains the items in an order (ascending/ descending) and has additional functions like remove_max and remove_min
- Maintain a Doctor’s appointment list