AAI2007 Introduction to Algorithms

Week 3: Stack, Queue

Instructor: Jinyoung Han (jinyounghan@skku.edu)

Practice Policy

Practice Policy

- We will close the session by **<u>10:30pm</u>**
  - Considering your last subway (or bus) time
- Scoring
  - 2 points if you finish in time
  - 1 point if you submit to TA by Thursday 12am
  - Otherwise, 0 point
- For those who couldn’t solve the problems in the last week, we accept your solutions by tomorrow 12am (only for the lab of 9/11)
- Solutions will be posted on the iCampus

Practice Policy

Practice Policy

- Guideline for getting TA mentoring
  - Please do as much as possible by yourself!
  - Do not ask (i) to fix your compile error and (ii) about python grammar
  - Basically, everything that you need to know for solving problems has been mostly explained in lecture
  - If you ask help, TA may ask about how you approach the problem and how to code, then he/she may give you some hints

In This Lecture

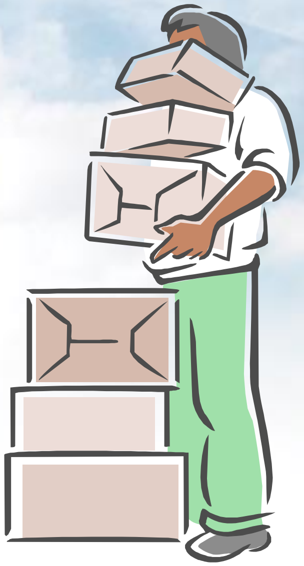
Outline

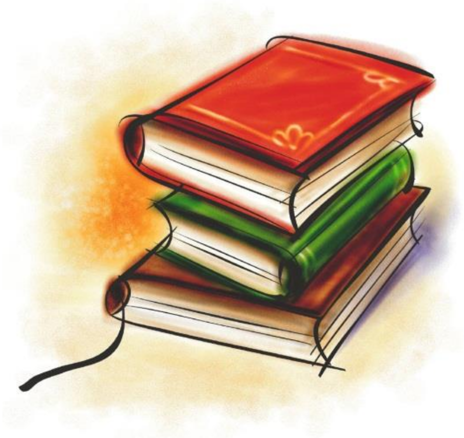
1. Stack
2. Queue

01. Stack

Stack?

- A pile of things arranged one on top of another

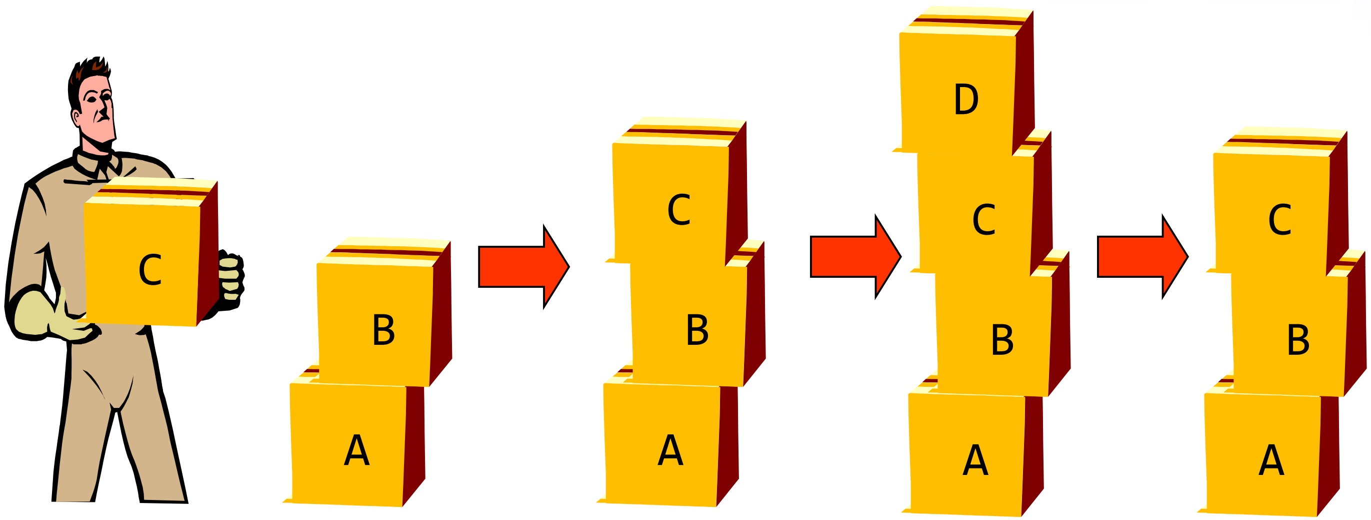




01. Stack

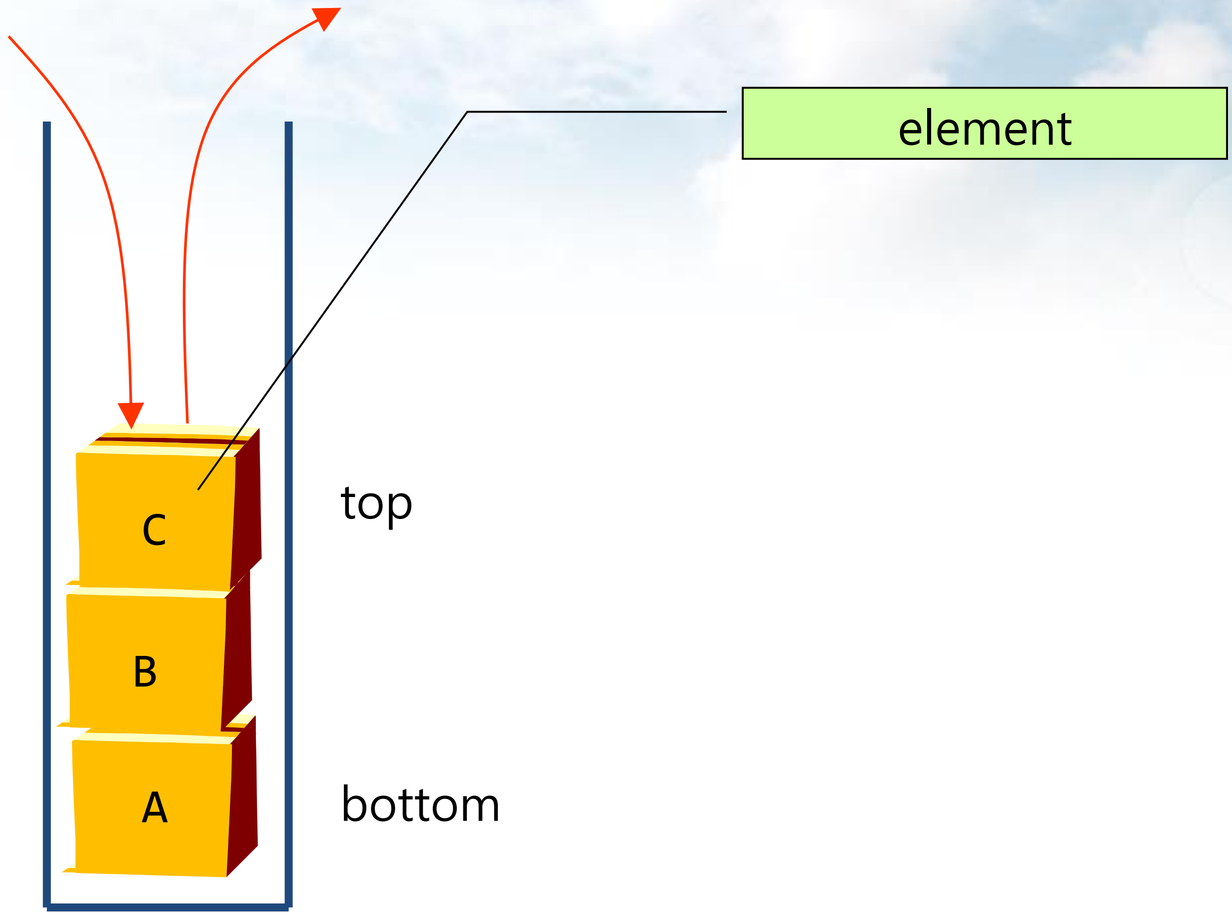
Stack

- In CS, a stack is a linear data structure that stores data such that the last item inserted is the first item removed
    - Used to implement a **last-in-first-out** (LIFO) type protocol



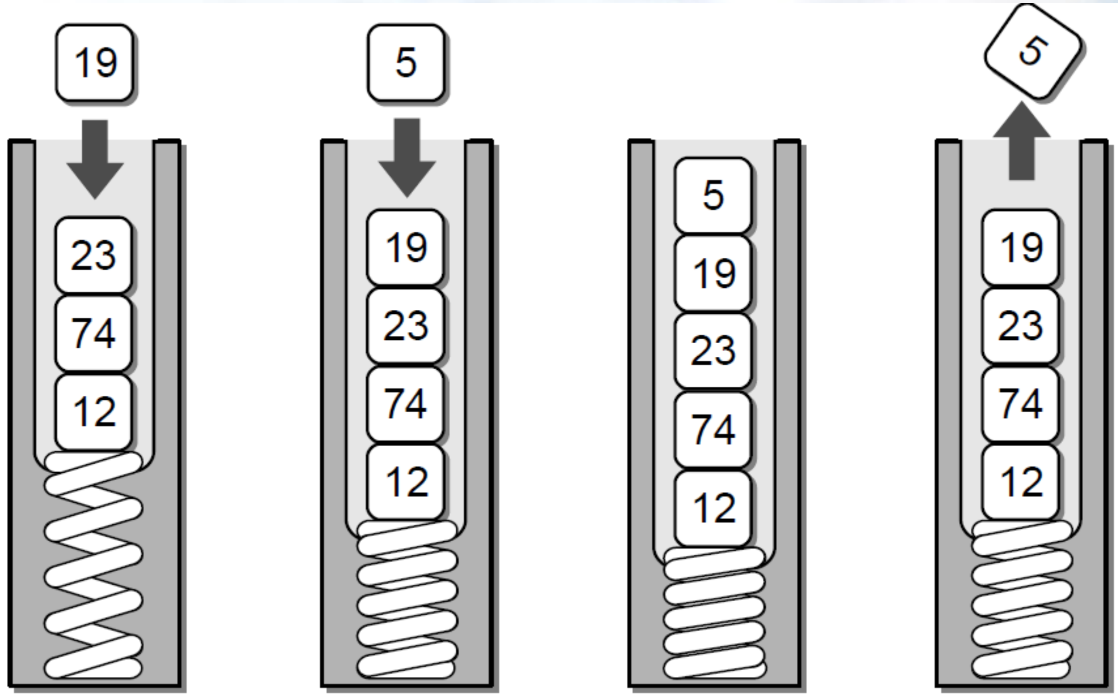
01. Stack

Stack Structure



01. Stack

Stack Example



Abstract view of a stack: (a) pushing value 19; (b) pushing value 5;
(c) resulting stack after 19 and 5 are added; and (d) popping top value

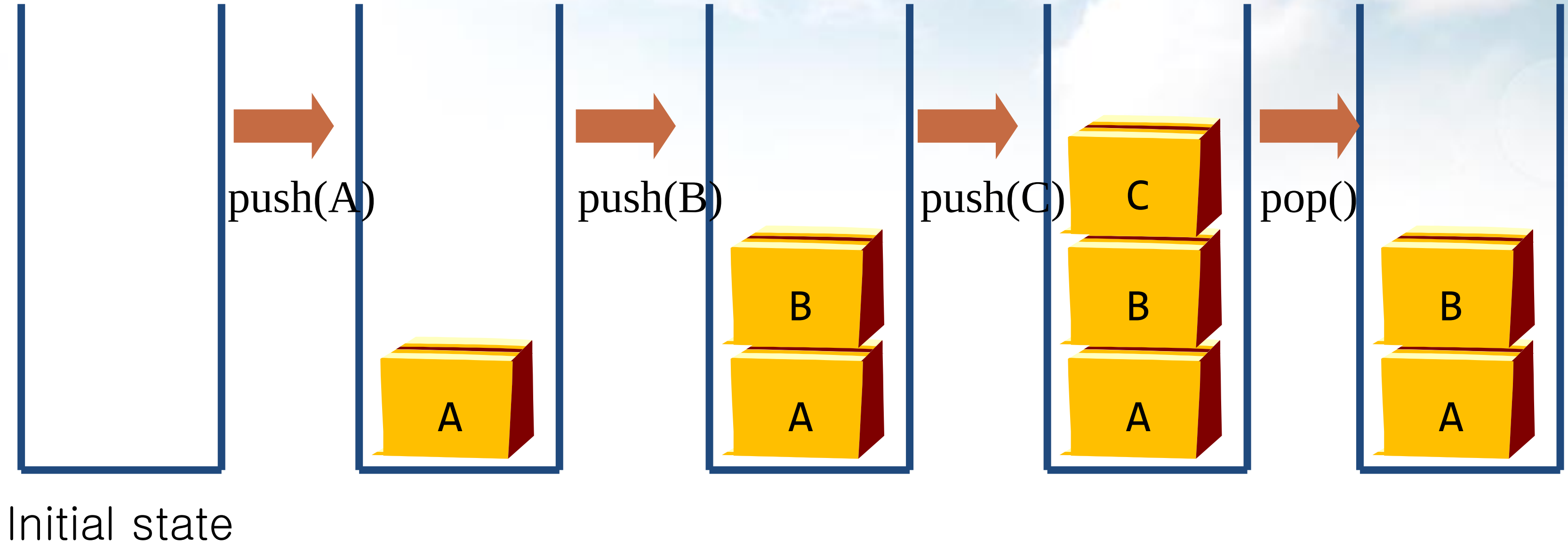
01. Stack

Stack ADT

- Object: a linear collection of n items with access limited to a LIFO order
- Operations:
  - create() ::=create a stack
  - is_empty(s) ::= check if the stack is empty
  - is_full(s) ::= check if the stack is full
  - push(s, e) ::= add an item e to the top of the stack
  - pop(s) ::= remove and return the top item of the stack
  - peek(s) ::= return the top item without removing it

01. Stack

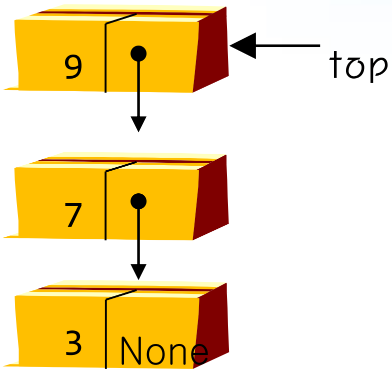
Stack Operations



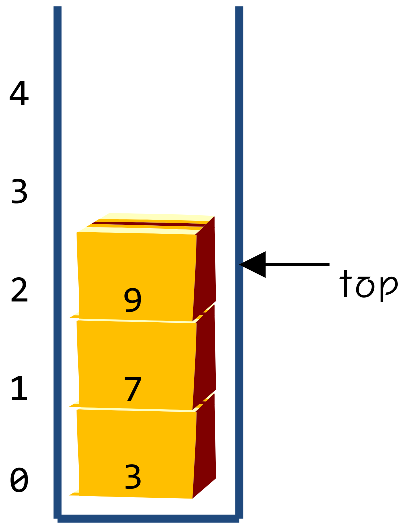
01. Stack

Implementation

- Using a Linked List
  - Linked stack



- How about implementing with an Array?
  - Reserved for your home study

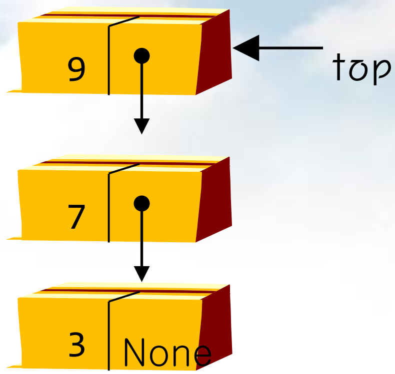


01. Stack

Implementation

- Stack Node

```
class StackNode :
   def __init__( self, item) :
       self.item = item
       self.next = None
```

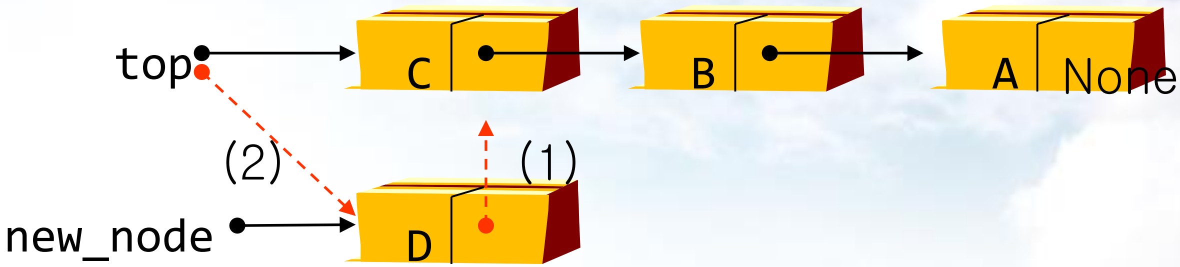


- Stack

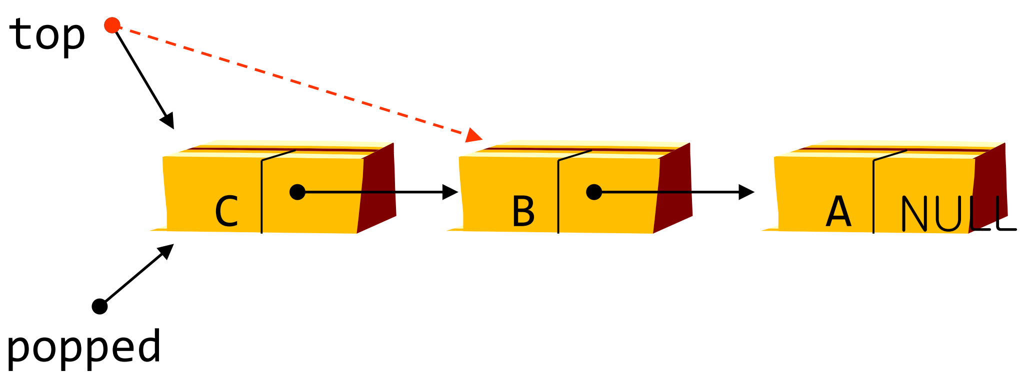
```
class Stack :
  def __init__( self ):
     self.top = None
     self.size = 0
```

01. Stack

Stack Push / Pop



[push]



[pop]

01. Stack

Stack Applications

- Reverse the sequence of input
  - input: (A,B,C,D,E) -> output: (E,D,C,B,A)
- ‘Undo’ function in editor
  - Erases the last change done to the document reverting it to an older state
- Stores the return address (PC: Program Counter) of a function in system stack
  - System stack: controlled by OS, not user

01. Stack

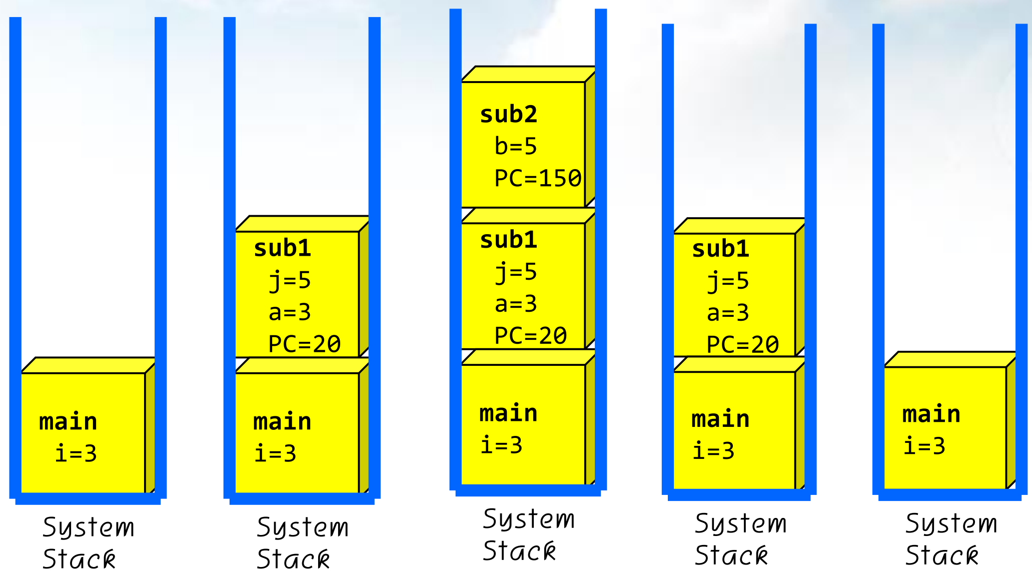
SA: PC

- Return address of a function (PC: Program Counter) is stored in system stack

```
1       int main()
        {
          int i=3;
20        sub1(i);
          ...
        }

100     int sub1(int a)
        {
          int j=5;
150       sub2(j);
          ...
        }

200     void sub2(int b)
        {
           ...
        }
```



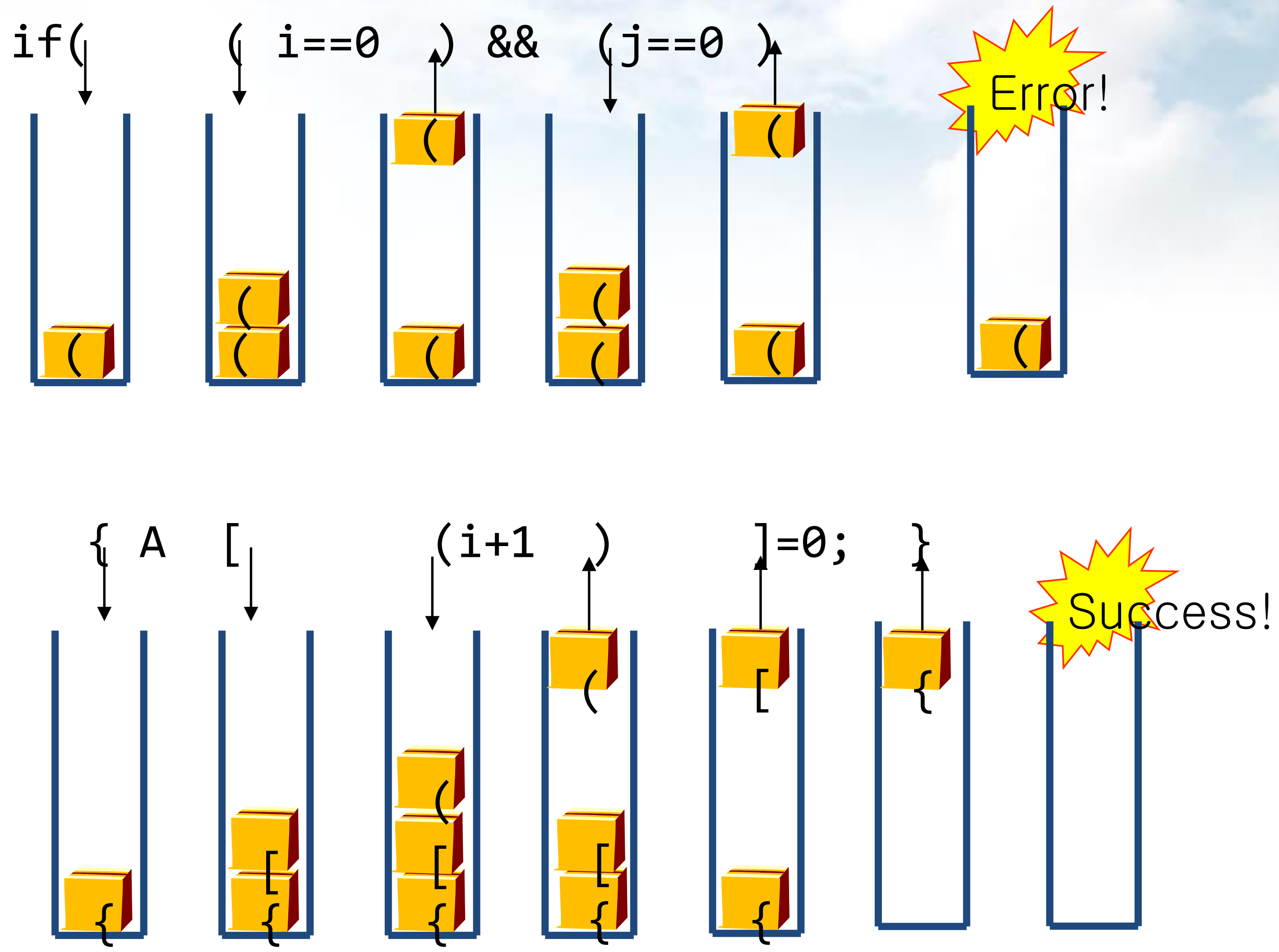
01. Stack

SA: Parenthesis

- Parenthesis
  - a square bracket (‘[’, ‘]’), a brace (‘{’, ‘}’), a round bracket (‘(’, ‘)’)
- Parenthesis inspection
  - # of left parentheses = # right parentheses
  - order: left parenthesis -> right parenthesis
  - Same type of parentheses should be used together
- Wrong examples
  - (a(b)
  - a(b)c)
  - a{b(c[d]e}f)

01. Stack

SA: Parenthesis



SA: Parenthesis

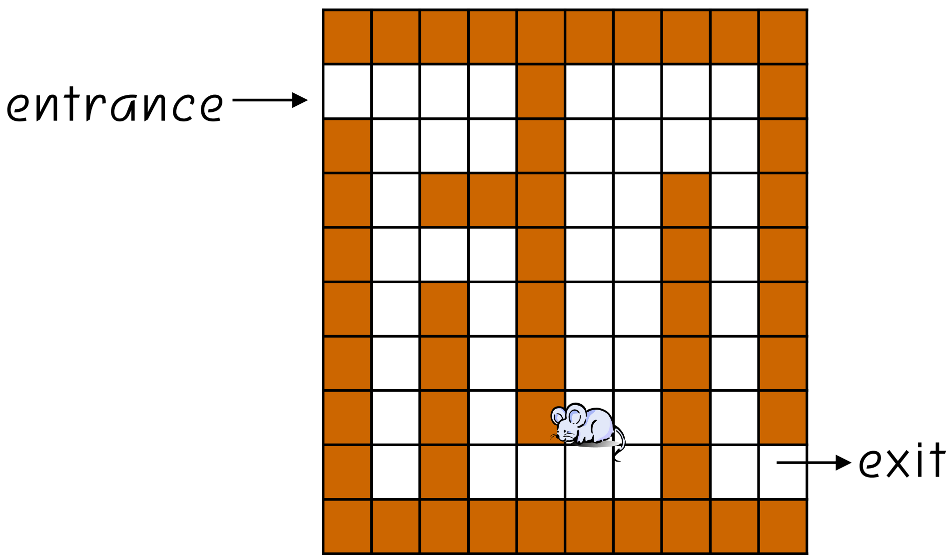
```
check_matching(expr)


while (input expr exists)
  ch ← the next letter in expr
  switch(ch)
    case '(': case '[': case '{':
       push ch in stack
       break
    case ')': case ']': case ']':
       if ( stack is empty )
         then error
         else pop open_ch from stack
             if (ch and open_ch are different parenthesis)
                 then error
       break
if( stack is not empty)
  then error
```

01. Stack

SA: Maze

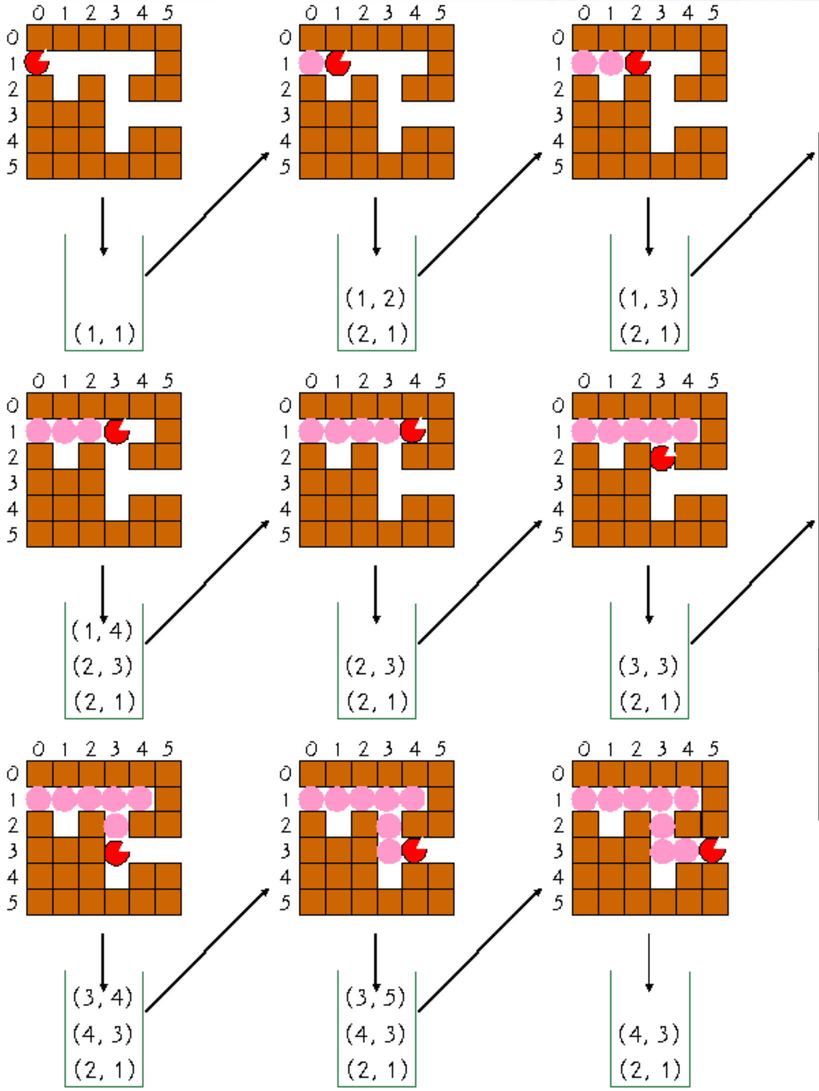
- Maze escape problem
  - Find the exit
  - Systematic solution is needed
    - Using stack!



1	1	1	1	1	1	1	1	1	1
0	0	0	0	1	0	0	0	0	1
1	0	0	0	1	0	0	0	0	1
1	0	1	1	1	0	0	1	0	1
1	0	0	0	1	0	0	1	0	1
1	0	1	0	1	0	0	1	0	1
1	0	1	0	1	0	0	1	0	1
1	0	1	0	1	m	0	1	0	1
1	0	1	0	0	0	0	1	0	x
1	1	1	1	1	1	1	1	1	1

01. Stack

SA: Maze



while(current location != exit)

do mark current location as visited

if(up, down, left, and right blocks of current location are unvisited and visitable)

then push them in stack

if(is_empty(s))

then error

else pop a position and make it as current location

Success!

Queue

02. Queue

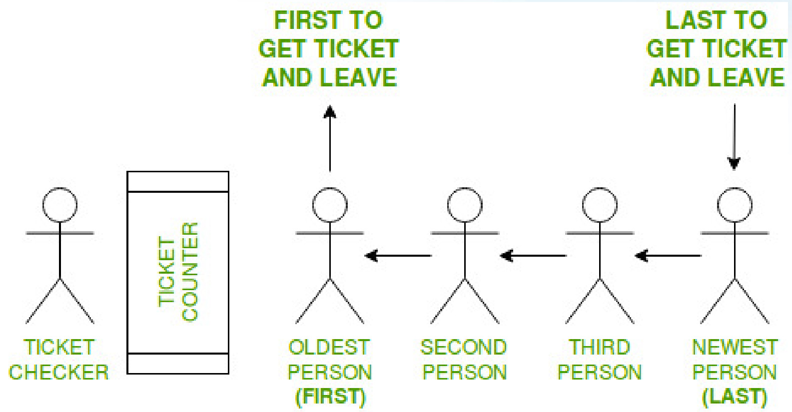
Queue

- Queue
  - Commonly defined to be a line of people waiting to be served like those you would encounter at many business establishments.

02. Queue

Queue

- FIFO (First-In First-Out)
  - First element is processed first and the newest element is processed last



- CF) LIFO
  - Last (or recent) element is processed first and the first (or oldest) element is processed last

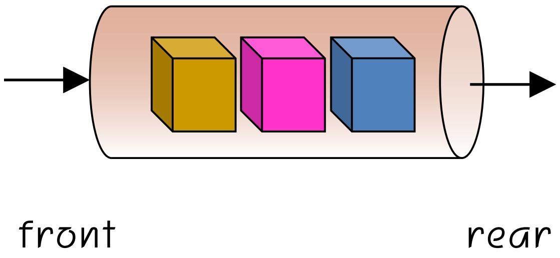
02. Queue

Queue ADT

·Object: a linear collection of n data in which access is restricted to a FIFO basis

·Operations:

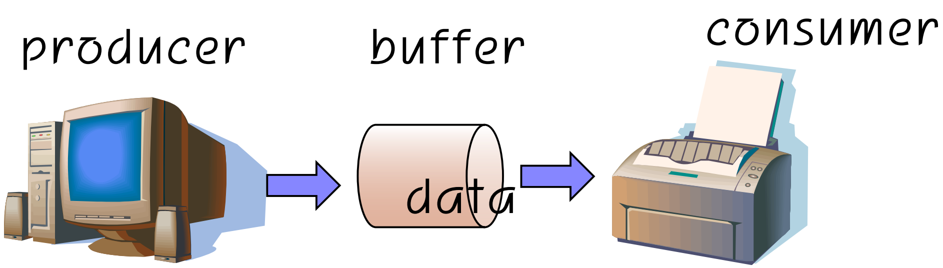
- **create()** ::= create a queue
- **init(q)** ::= initialize a queue
- **is_empty(q)** ::= check if the queen is empty
- **is_full(q)** ::= check if the queue is full
- **enqueue(q, e)** ::= add data at rear
- **dequeue(q)** ::= remove data at front
- **peek(q)** ::= return data at front without removing



02. Queue

Queue Applications

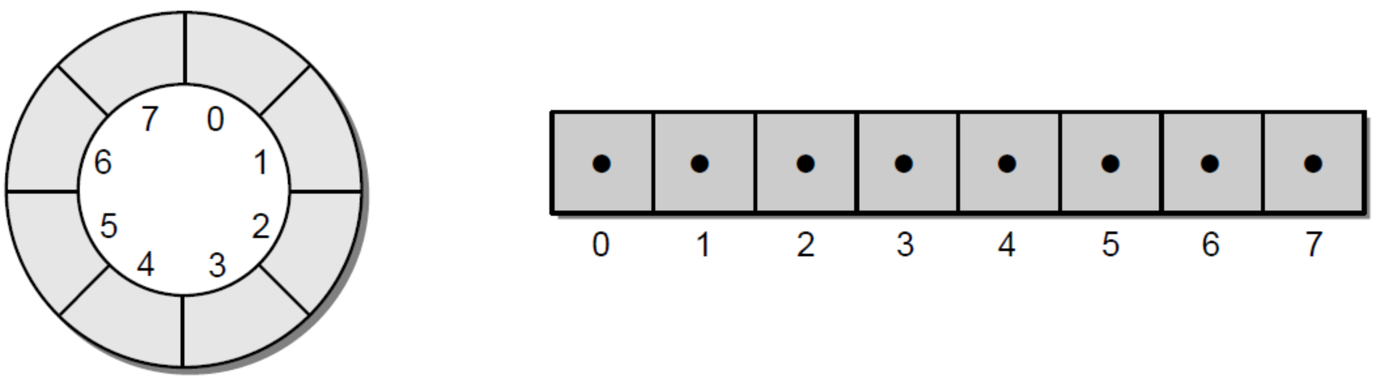
- Simulation queue
  - flights in an airport, customers in a bank
- Network packets in a queue
- Buffering between a printer and a computer
- Used in many algorithms, data structures, systems, etc.



QUEUE

Circular Queue

- Circular queue
  - A linear data structure in which the operations are performed based on FIFO principle and the last position is connected back to the first position to make a circle

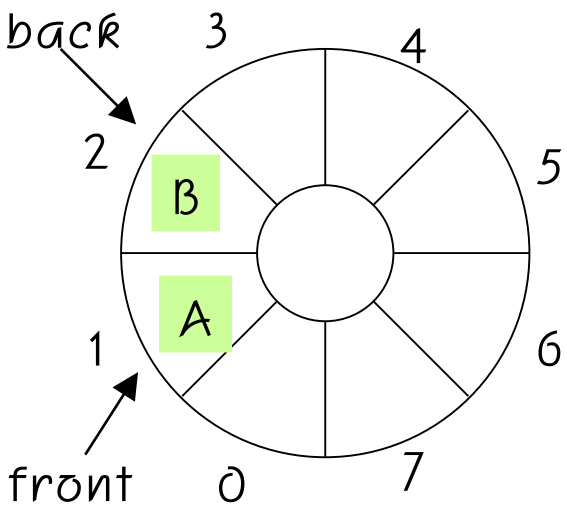


The abstract view of a circular queue (left) and the physical view (right)

02. Queue

Circular Queue

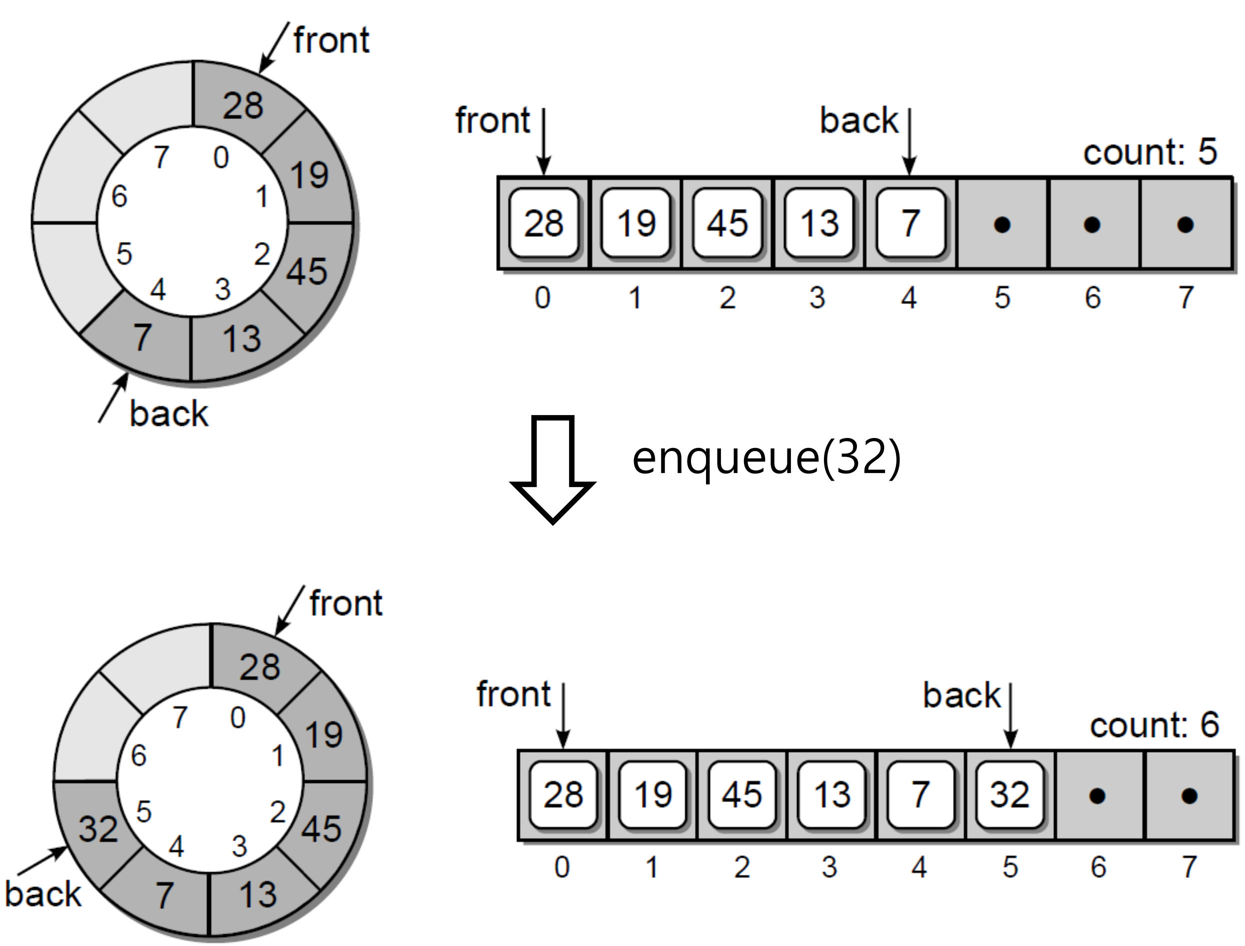
- Data organization
  - Two variables are needed for managing the front and the rear
    - Front: indicate the first data
    - Back: indicate the last data



02. Queue

Circular Queue

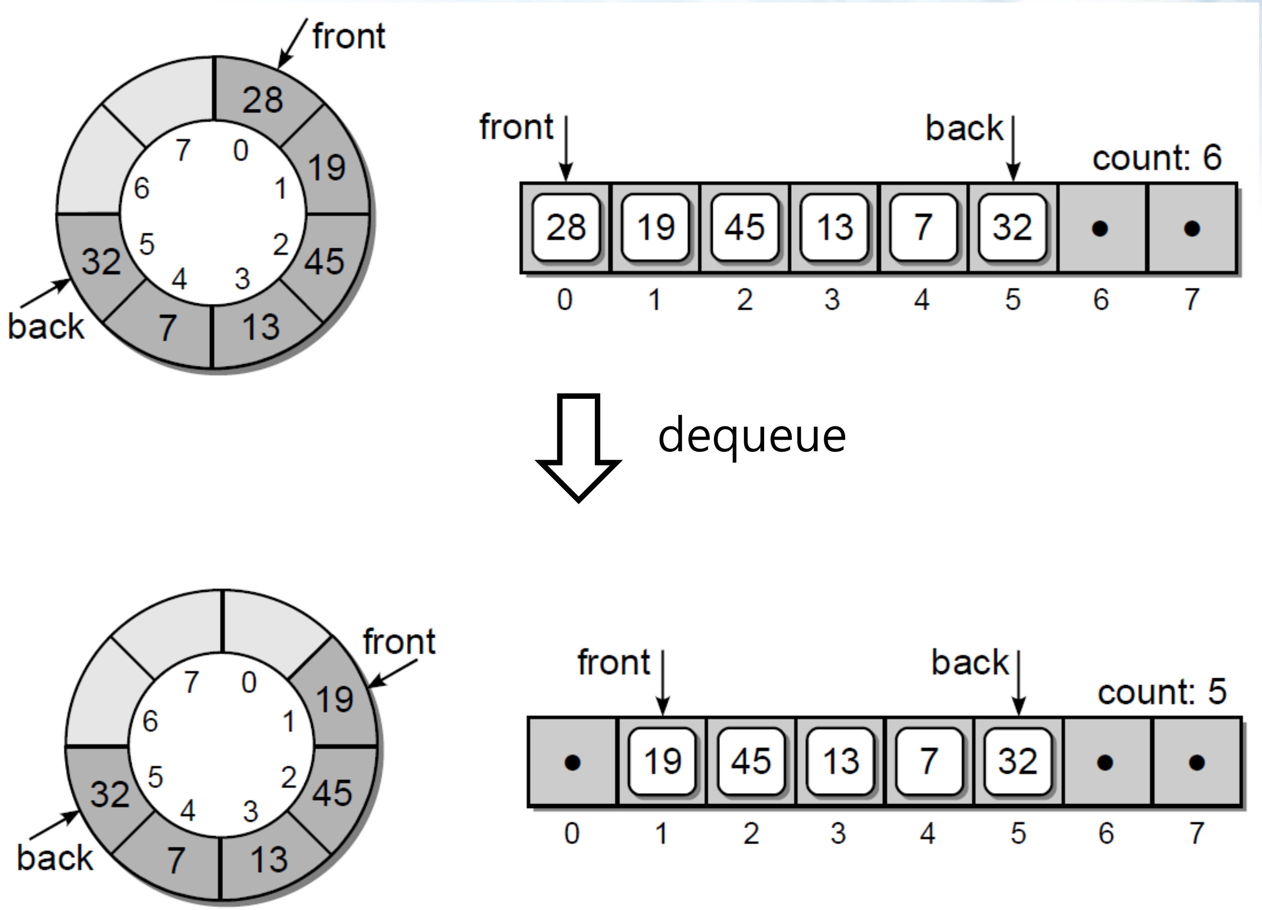
- Operations



02. Queue

Circular Queue

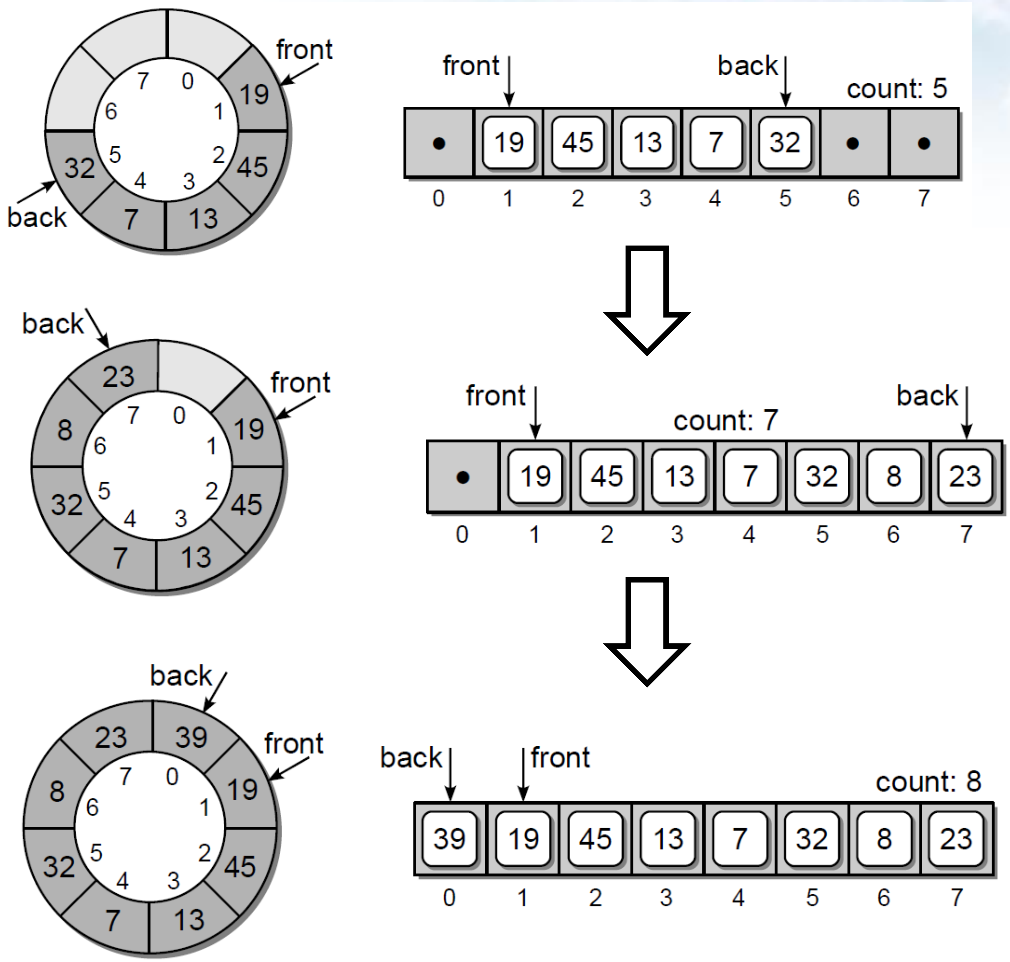
- Operations



02. Queue

Circular Queue

- Operations

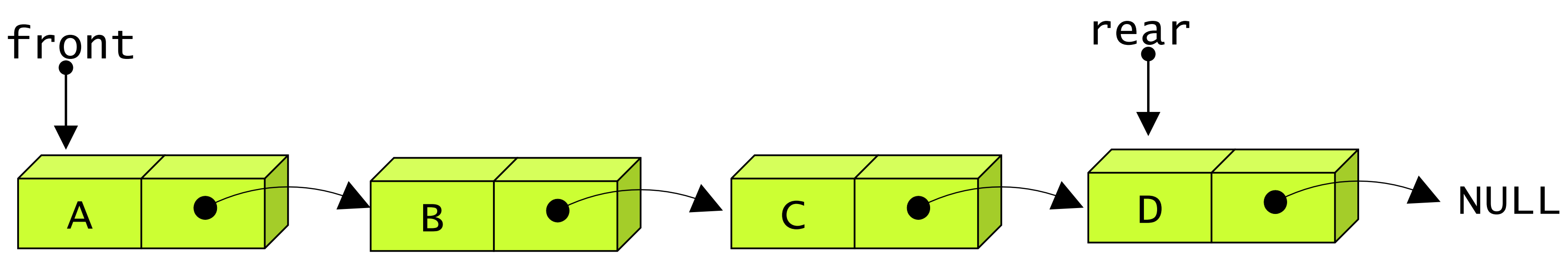


enqueue(8)
enqueue(23)
enqueue(39)

02. Queue

Linked Queue

- Linked queue
  - A queue implemented by a singly linked list
  - Two variables are needed for managing the front and the rear
    - Front: indicate the first data
    - Rear: indicate the last data

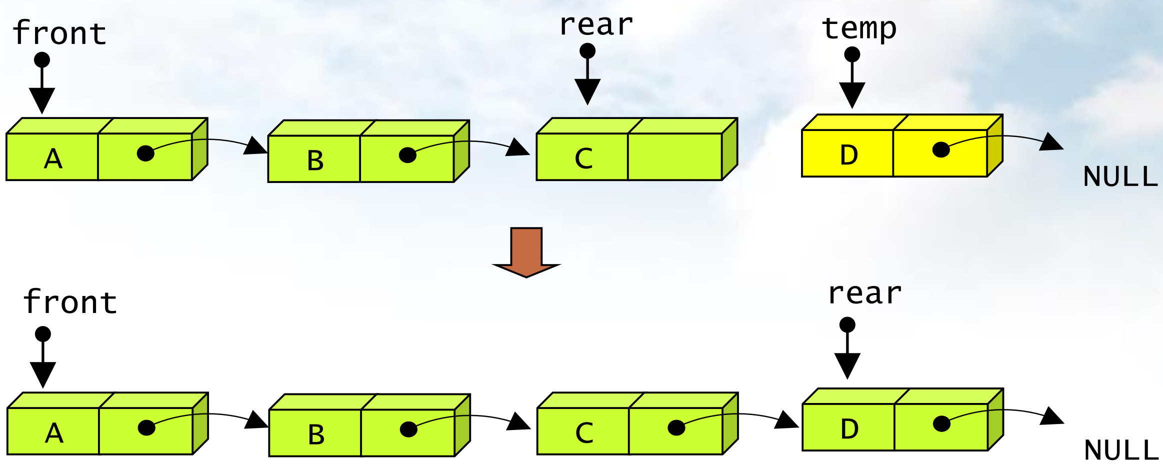


02. Queue

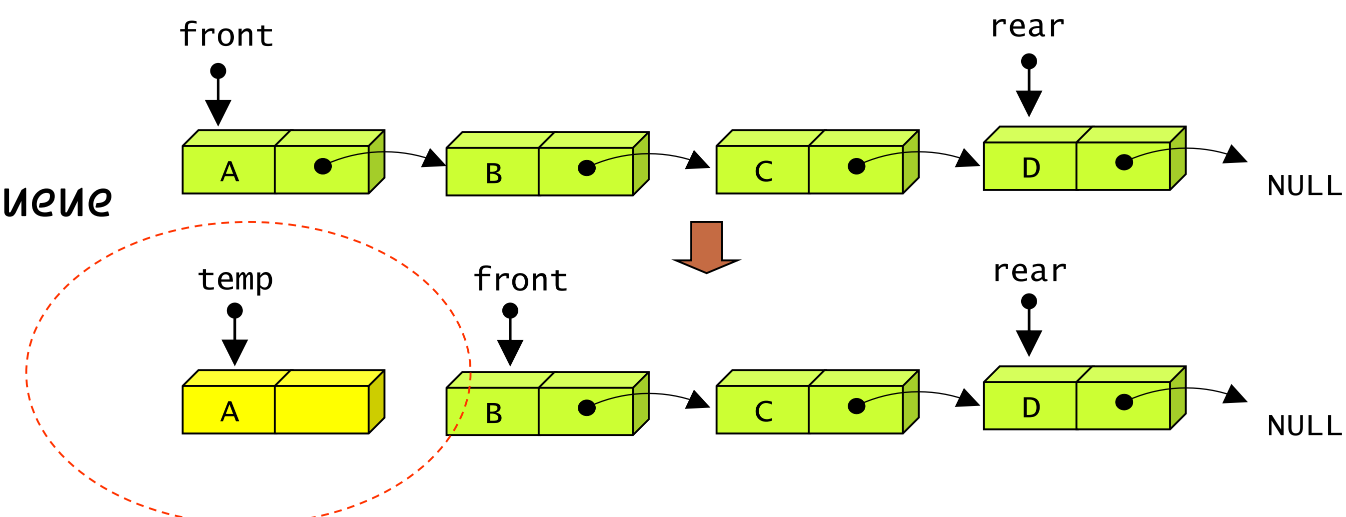
Linked Queue

- Operations

enqueue



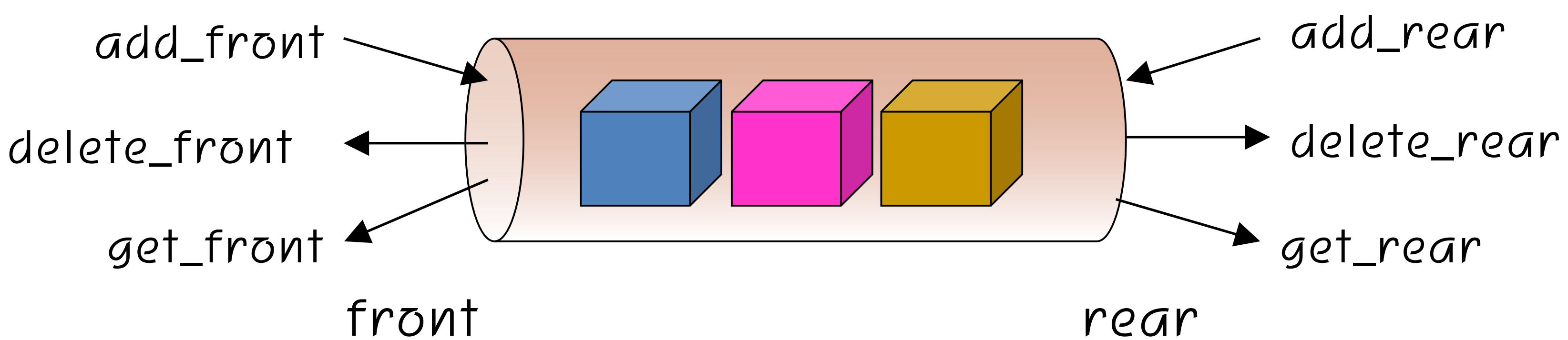
dequeue



02. Queue

Deque

- Deque (double-ended queue)
  - An abstract data type that generalizes a queue, for which elements can be added to or removed from either the front or back
  - Deque can implement both stack and queue



Deque ADT

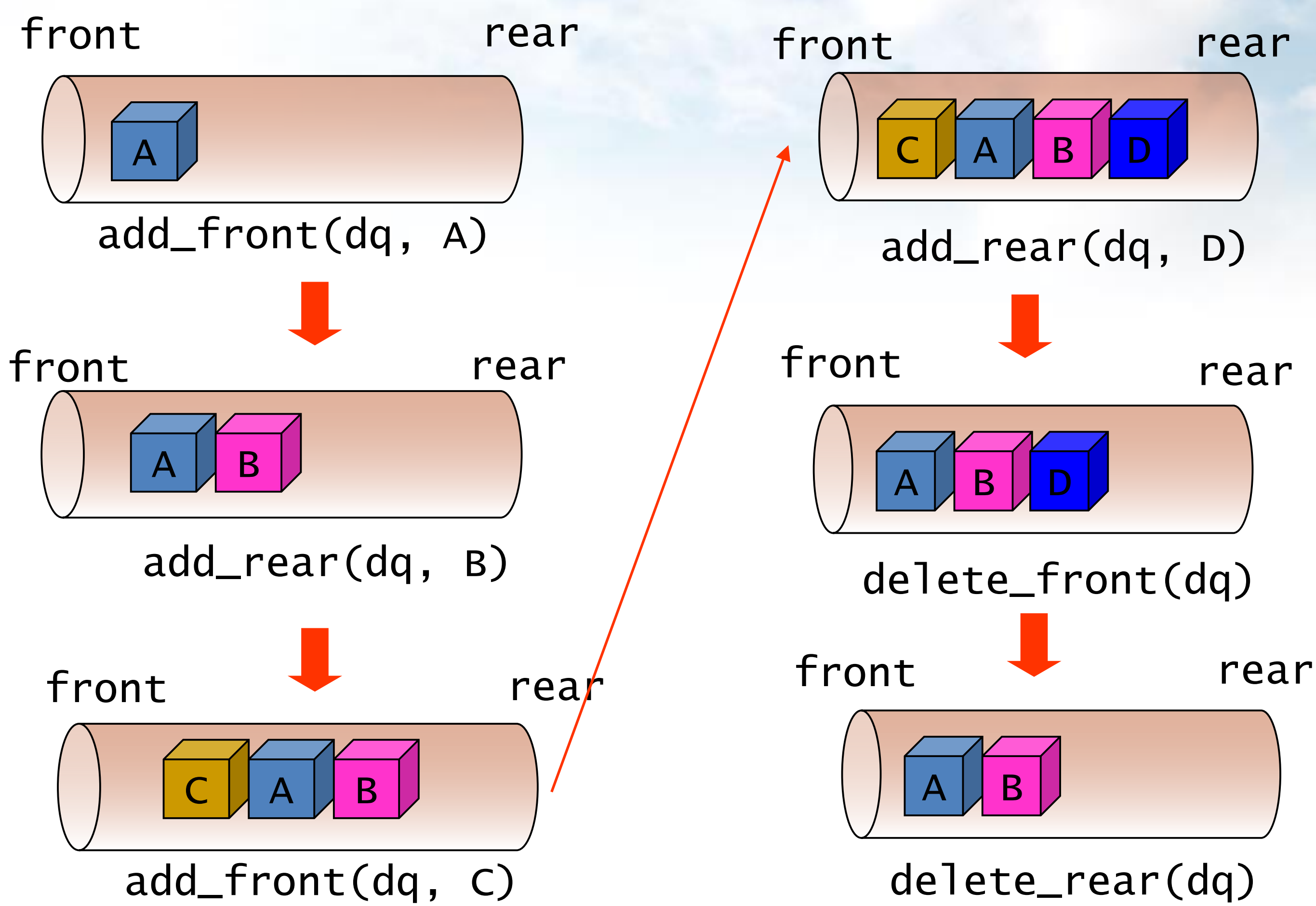
·Object: a linear collection of n data

·Operations:

- create() ::= create a deque
- init(dq) ::= initialize a deque
- is_empty(dq) ::= check if the deque is empty
- is_full(dq) ::= check if the deque is full
- add_front(dq, e) ::= add data at front
- add_rear(dq, e) ::= add data at rear
- delete_front(dq) ::= remove data at front
- delete_rear(dq) ::= remove data at rear
- get_front(q) ::= return data at front without removing
- get_rear(q) ::= return data at rear without removing

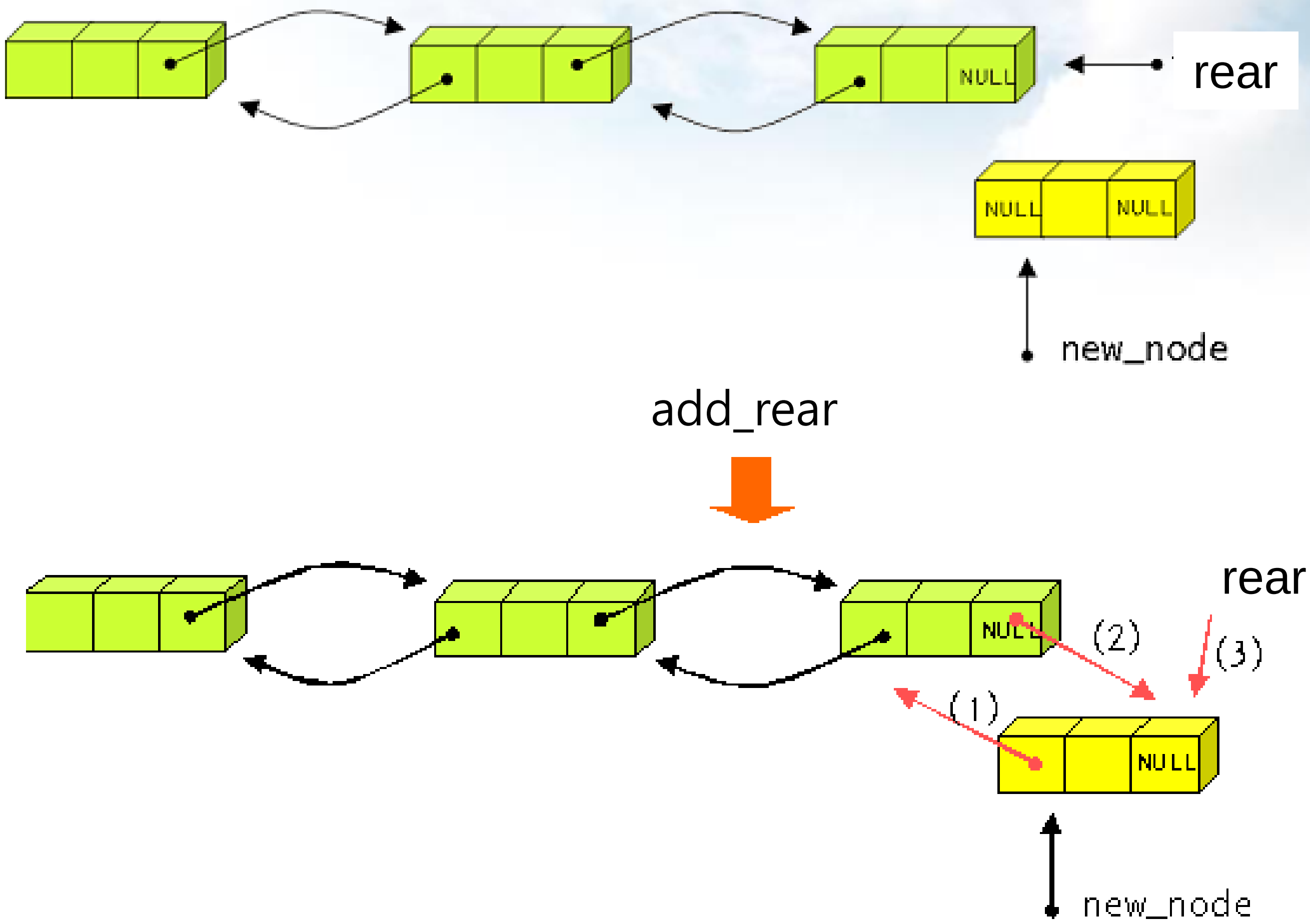
02. Queue

Deque Operations



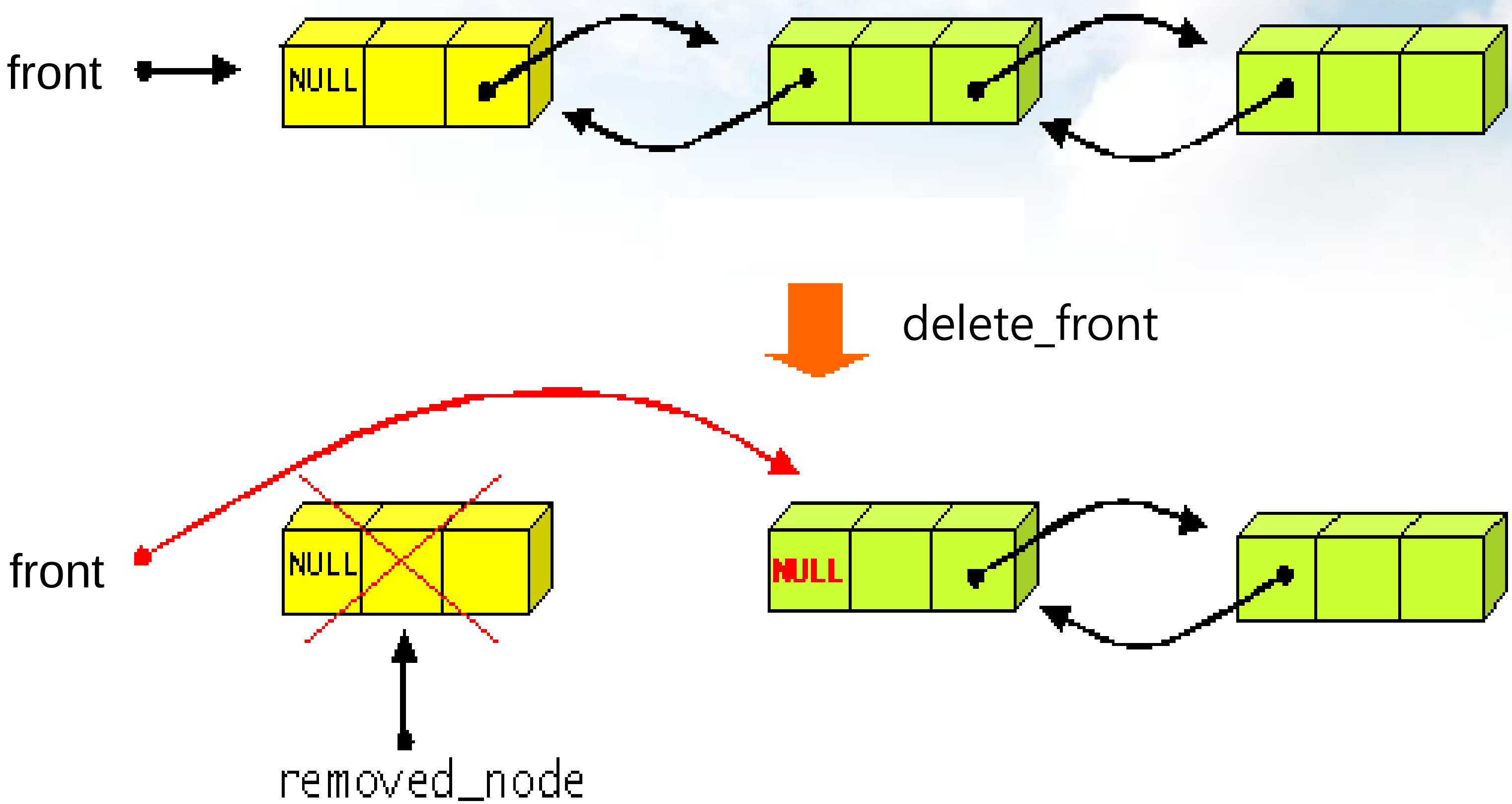
02. Queue

Deque Operations



02. Queue

Deque Operations

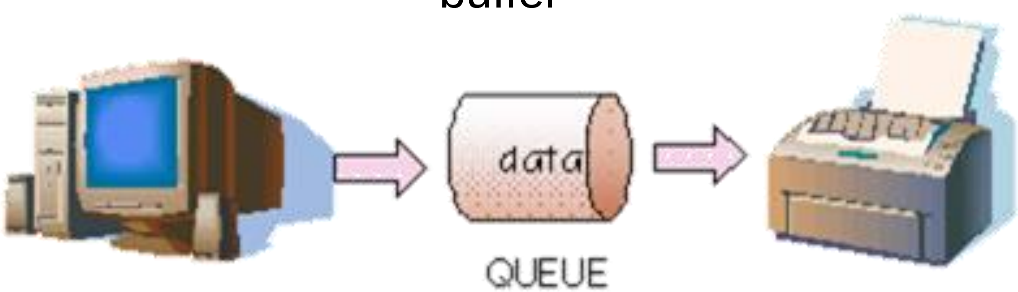


02. Queue

Applications

- Buffer can handle interactions between two different processes with different speeds
  - CPU <-> Printer
  - Producers <-> Consumers

producer buffer consumer



02. Queue

Applications

- Produce-Consumer Algorithms

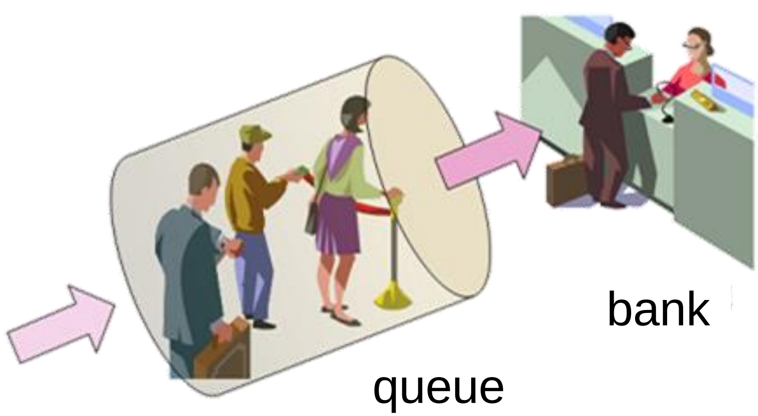
```
def producer() :
    while True:
        produce data
        while lock(buffer) != SUCCESS
            if not is_full(buffer):
                enqueue(buffer, data)
        unlock(buffer)
```

```
def consumer() :
    while True:
        while lock(buffer) != SUCCESS
            if not is_empty(buffer):
                data = dequeue(buffer)
                consume data
        unlock(buffer)
```

02. Queue

Applications

- Simulation
  - A system can be analyzed using a Queueing theory
    - Queueing theory
      - The mathematical study of waiting lines, or queues. A queueing model is constructed so that queue lengths and waiting time can be predicted
  - E.g., bank simulation



What You Need to Know

Summary

- Stack
    - LIFO
    - Linked Stack
- Queue
    - FIFO
    - Circular Queue
    - Linked Queue
    - Deque

Thanks

Week 3: Stack, Queue
Instructor: Jinyoung Han (jinyounghan@skku.edu)