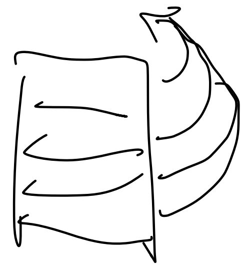
Lecture 5
Stack and Queue

Bo Tang @ SUSTech, Fall 2024

Our Roadmap

- Stack → 只能访问顶上的数据(增删)

Heap 堆、栈



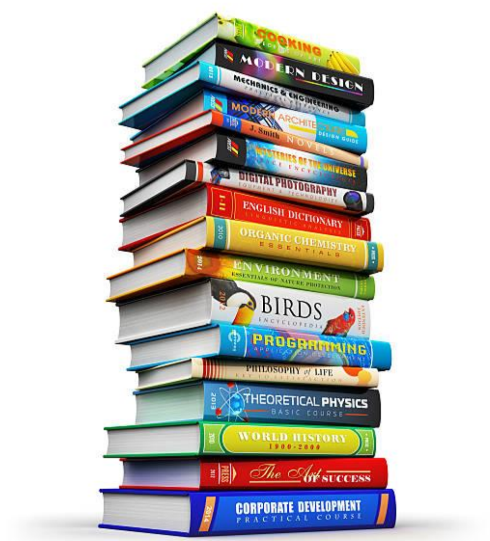
Both stack and queue only allow items can be accesed at specific positions.

- Queue

- Stack vs. Queue

Stack

- A stack is a sequence in which:
  - Items can be added and removed only at one end (the top)
  - You can only access the item that is currently at the top
- Stack Analogy



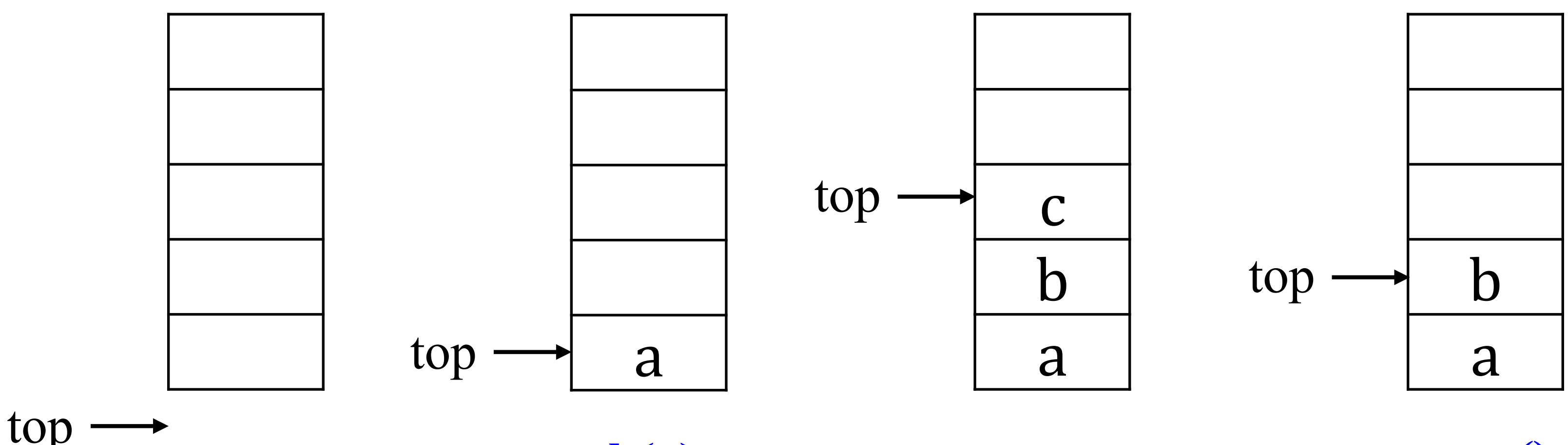
Stack

Array / linked list -> stack

① push()

② pop()

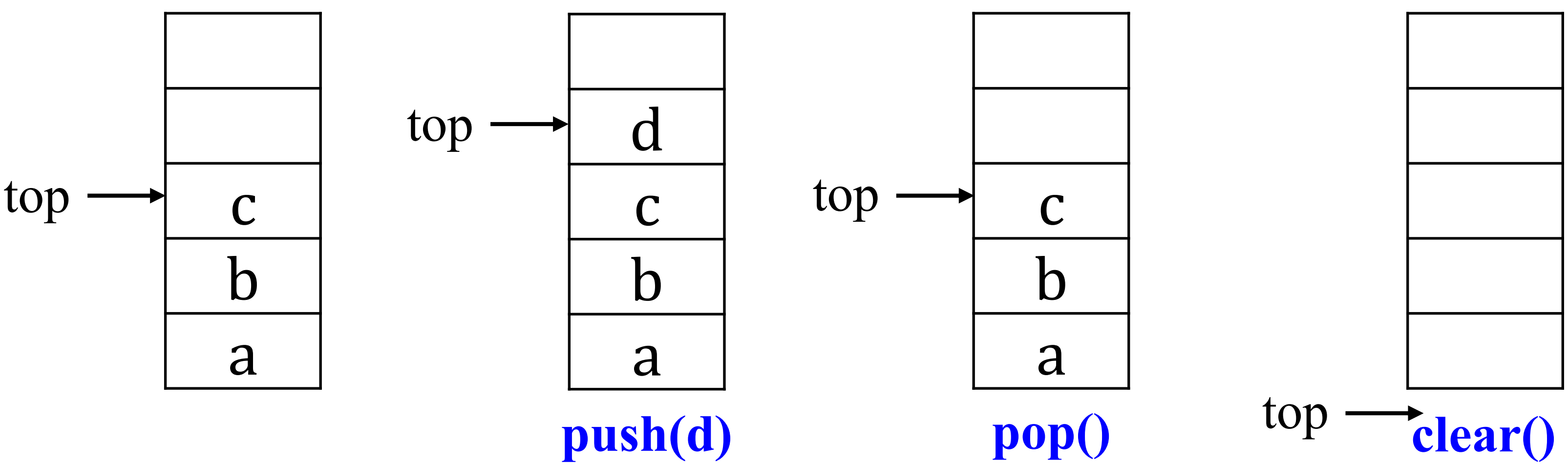
- First In Last Out (FILO)
  - Constrained item access
- Major Operations
  - push: add an item to the top of the stack
  - pop: remove the item at the top of the stack
- Illustration



empty **push(a)** **push(b), push(c)** **pop()**

Stack Operation

- push: add an item to the top of the stack
- pop: remove the item at the top of the stack
- top/peek: get the item at the top of the stack, but do not remove it → 得到最上面元素，
- isEmpty: test if the stack is empty
- isFull: test if the stack is full
- clear: clear the stack, set it as empty stack
- size: return the current size of the stack

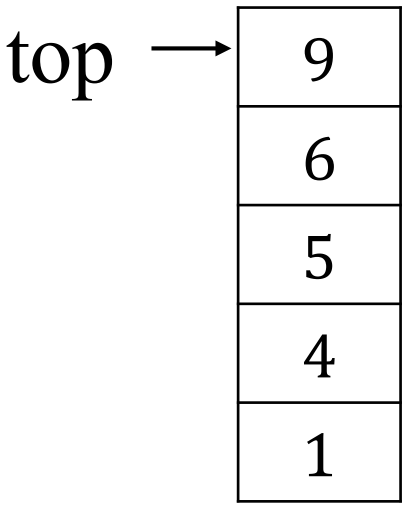


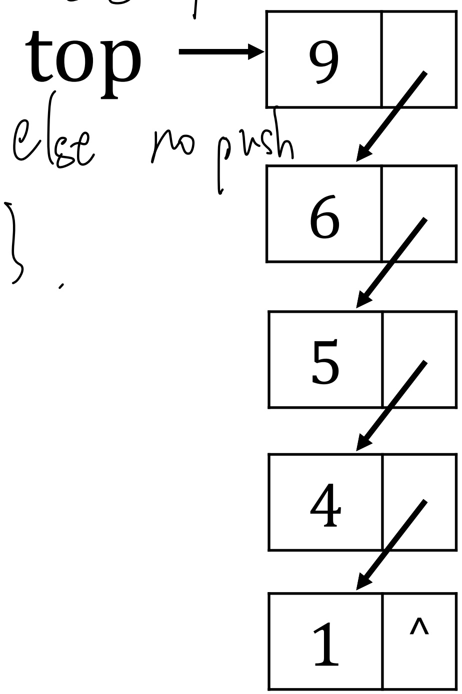
Implementation of Stack

- Array-based Stack
- Linked Stack

push (item)
{ if (top+1 ≤ n)
top++
S[top] = item;
else no push
}.

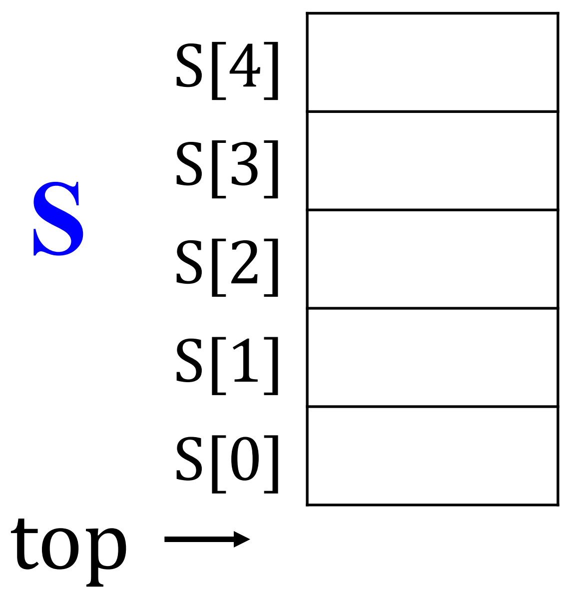
pop ( )
{ if (top ≥ 1)
top --
else
stack is empty.
}





Implementation of Stack

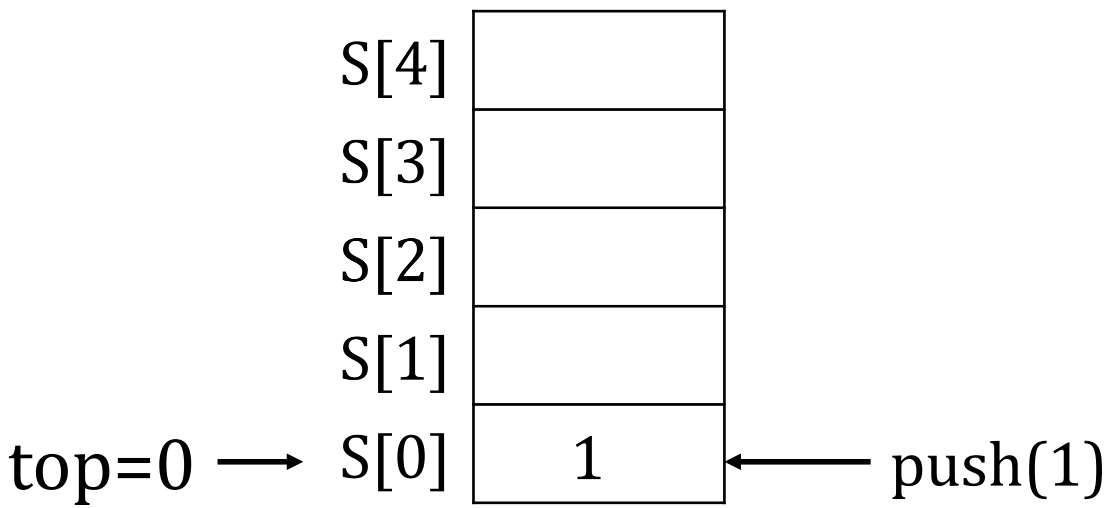
- Array based Stack
  - MAX_SIZE = n // the max size of stack
  - top = -1 // the current top position
  - Array S with n elements
- Example
  - MAX_SIZE = 5
  - top = -1
  - Array S



Push Operator

- push(item):
  1. top++;
  2. S[top] = item

- push(1)

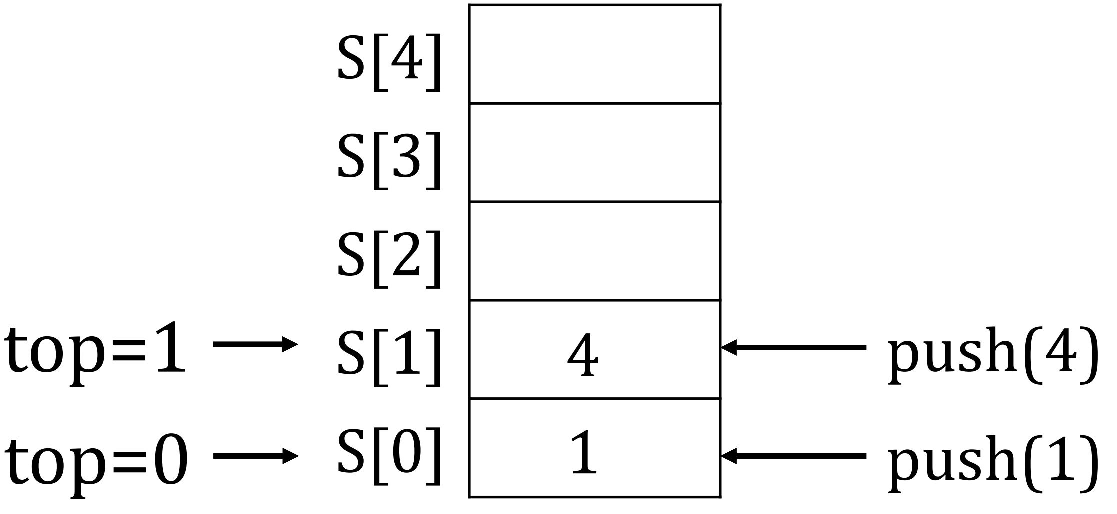


Push Operator

- push(item):

```
1. top++;
2. S[top] = item
```

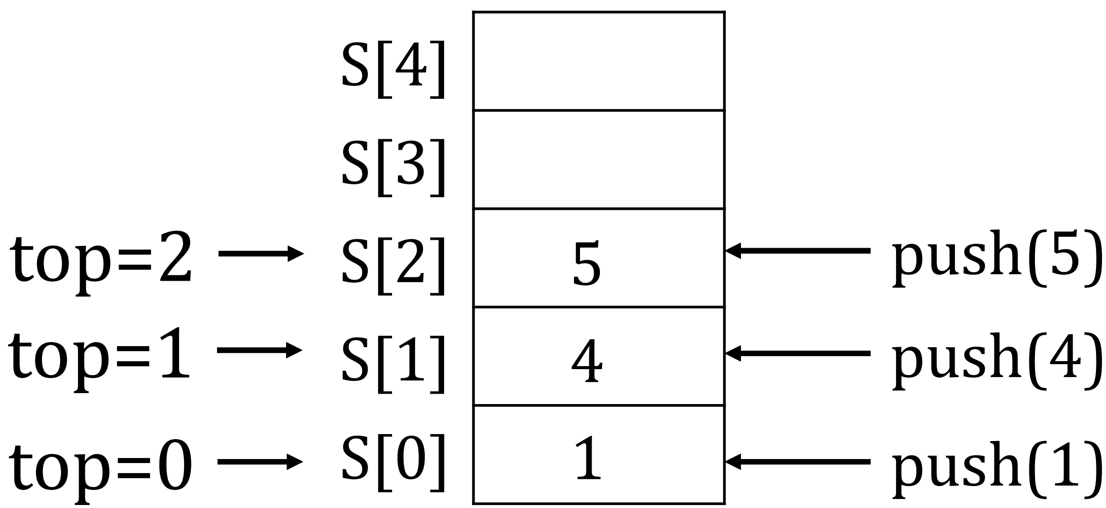
- push(1)
- push(4)



Push Operator

- push(item):
  1. `top++;`
  2. `S[top] = item`

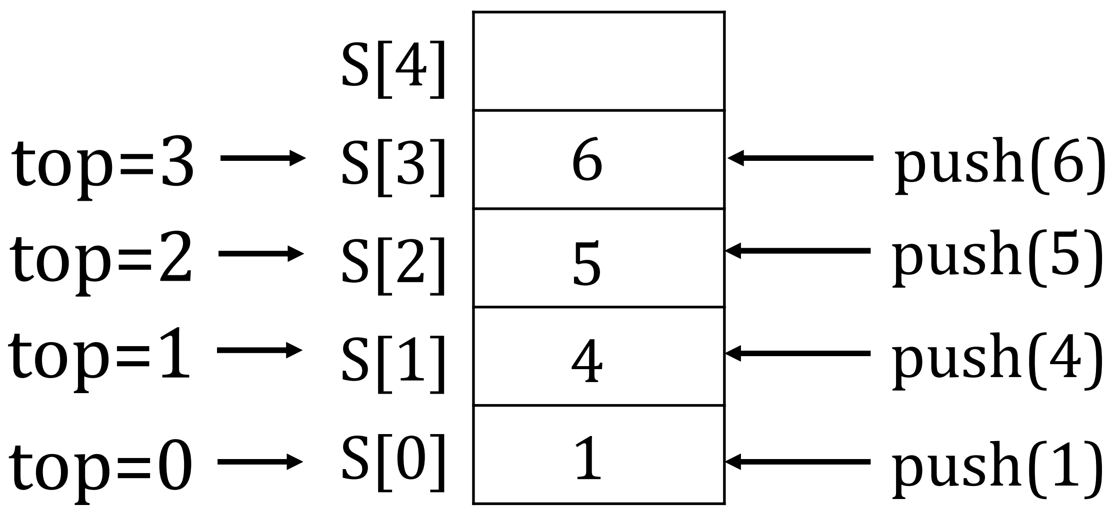
- push(1)
- push(4)
- push(5)



Push Operator

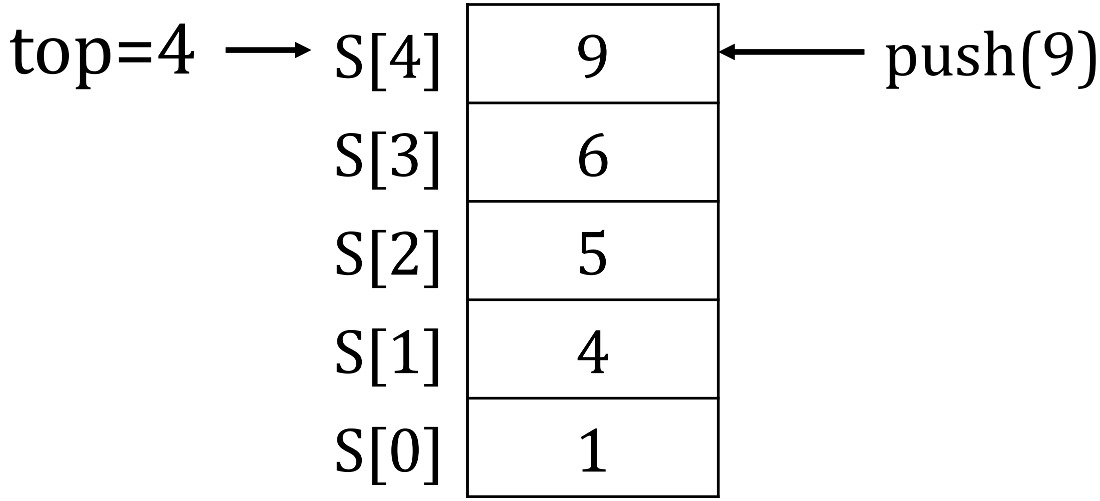
- push(item):
  1. `top++;`
  2. `S[top] = item`

- push(1)
- push(4)
- push(5)
- push(6)

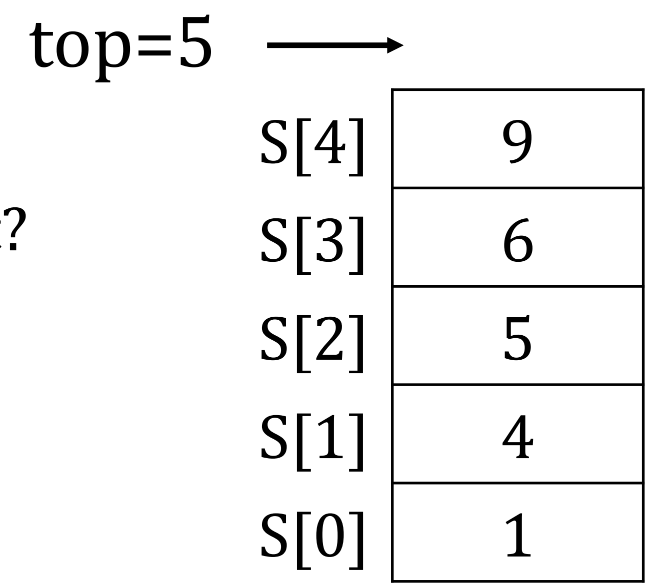


Push Operator

- push(9)



- push(10)
  - OVERFLOW
  - How to avoid that?



Push / Pop Operator

- push(item):

```
if(top == MAXSIZE-1)
        Stack is FULL! No push!
else
        top++;
        S[top] = item
```

- pop(): // should avoid underflow

```
if(top == -1)
        Stack is EMPTY! No pop!
else
        top--;
```

Application of Stacks

- Making sure the delimiters (parens, brackets, etc.) are balanced:
  - Push open (i.e., left) delimiters onto a stack
  - When you encounter a close (i.e., right) delimiter, pop an item off the stack and see if it matches
- Evaluating arithmetic expressions
  - Parsing arithmetic expressions written using infix notation
- The runtime stack in memory
  - Converting a recursive algorithm to an iterative one by using a stack to emulate the runtime stack

Brackets Balance Problem

- a+{2-[b+c]*(8* [8+g]/[m-e]-7)-p}
- {[ ]([ ][ ])}
- Skip operators and notations
- Is the bracket expression balanced or not?
  - () Yes
  - [} No
  - {[()}] No
  - {[ ]([ ][ ])} Yes
  - {{{{{{[[[[[[(((((()))))))]]]]]]}}}}}} Yes

Brackets Balance Problem

- Given a bracket expression, determine whether it is balanced or not?
- {[ ]([ ][ ])}
  - How to solve it by using stack?
  - Bracket pairs: ( ), [ ], { }
  - Any ideas?
- Methodology
  - Employ stack store checked left bracket
  - Pop out left bracket if it is matched

Arithmetic Expression Evaluation

递归

- Arithmetic expression
  - operands (a, b, c), operator (+, *)
  - a + b * c

② evaluate postfix

① infix 转为 postfix

- Prefix expression
  - + a * b c

5*((9+3)*(4*2)+7)

- Infix expression
  - a + b * c

不再会有括号.

find operator, and what is left and right

- Postfix expression

5 9 3 + 2 4 * * 7 + *

遇到一个操作数就会计算一次前面的数

  - a b c * +

Postfix Expression

- Infix expression
  - 5 * ((9 + 3) * (4*2) + 7)

遇到数存进去，遇到符号pop符号前的数，计算完后存回去。

- Postfix expression
  - 5 9 3 + 4 2 * * 7 + *

→ 二元运算符。- 单运算符就pop一个出来。

- Parse postfix expression is somewhat easier problem than directly parsing infix (why)
- Postfix has a nice property that parentheses are unnecessary
- Postfix Expression Evaluation
  - Convert from infix to postfix
  - Evaluate a postfix expression

Postfix Expression

- Postfix expression
  - 5 9 3 + 4 2 * * 7 + *
- Methodology
  - Read the tokens in one at a time
  - If it is an operand, push it on the stack
  - If it is a binary operator:
    - pop top two elements from the stack,
    - apply the operator,
    - and push the result back on the stack

Postfix Expression Evaluation

- 5 9 3 + 4 2 * * 7 + *
- Postfix Expression Evaluation

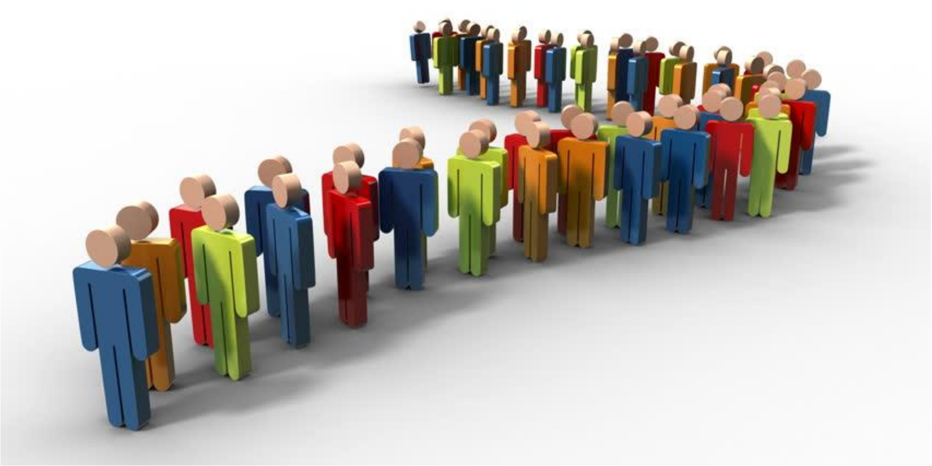
Stack operations	Stack elements
push(5)	5
push(9)	5 9
push(3)	5 9 3
push(pop() + pop())	5 12
push(4)	5 12 4
push(2)	5 12 4 2
push(pop() * pop())	5 12 8
push(pop() * pop())	5 96
push(7)	5 96 7
push(pop() + pop())	5 103
push(pop() * pop())	515

Our Roadmap

- Stack
- Queue
- Stack vs. Queue

Queue

- A queue is a sequence in which:
  - items are added at the rear and removed from the front
  - You can only access the item that is currently at the front
- Queue Analogy



Queue

→ 可remove 头

可add 尾.

FILO → [illegible]

- First In First Out (FIFO)
  - Items access constrained
- Major elements
  - front: the first element in the queue (remove)
  - rear: the last element in the queue (add)
- Illustration

deQueue enQueue.

deQueue ← $a_1, a_2, \ldots, a_n$ ← **enQueue**

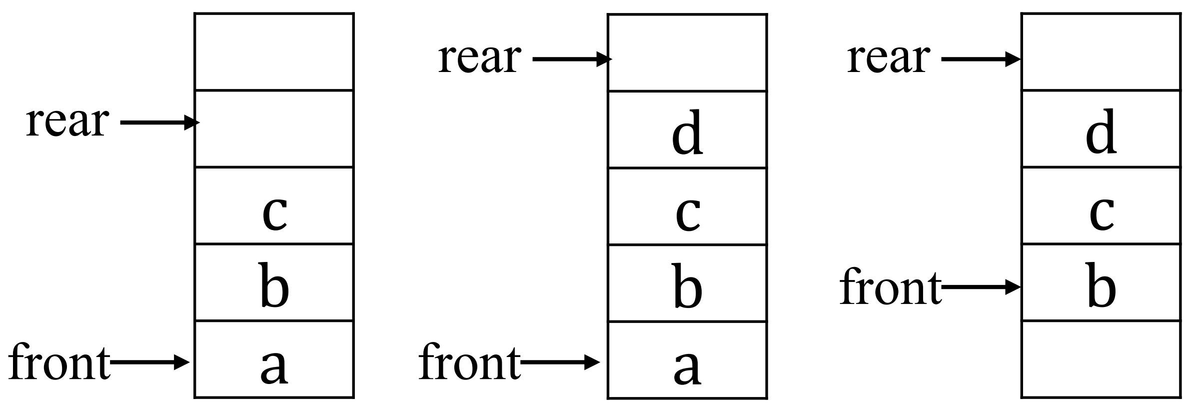
front **rear**

Queue Operations

- enQueue: add an item at the rear of the queue
- deQueue: remove the item at the front of the queue
- front: get the item at the front of the queue, but do not remove it
- isEmpty: test if the queue is empty
- isFull: test the queue is full
- clear: clear the queue, set it as empty queue
- size: return the current size of the queue

rear ⇒ [illegible]

enQueue(item)
{ if (rear < n)
Q[rear] ← item
rear++
} else Q[rear] → full



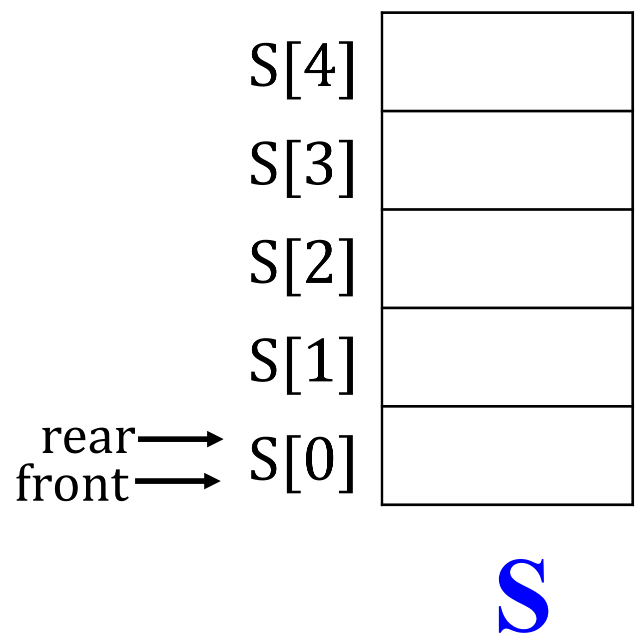
enQueue(d) **deQueue()**

Implementation of Queue

- Array based Queue
  - MAX_SIZE = n // the max size of stack
  - front = 0 // the current front
  - rear = 0 // the current rear
  - Array S with n elements

de Queue
{ if (front < rear)
front++;
} else empty.

- Example
  - MAX_SIZE = 5
  - front = 0
  - rear = 0
  - Array S

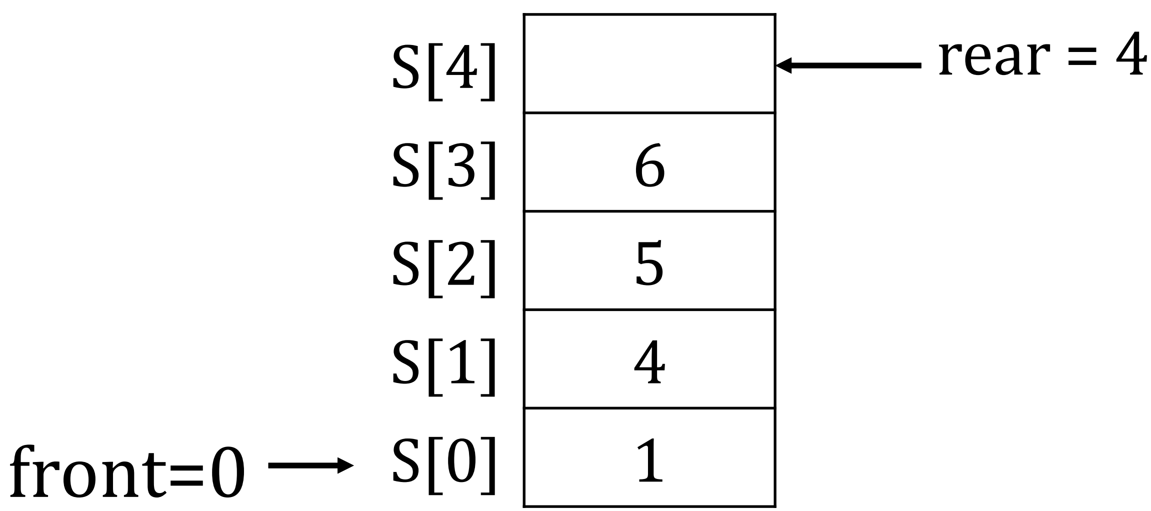


enQueue Operator

- enQueue(item):

```
if(rear < MAXSIZE)
        S[rear] = item
        rear++
else
        Queue is FULL, no enQueue
```

- enQueue (1), enQueue(4), enQueue(5), enQueue(6)

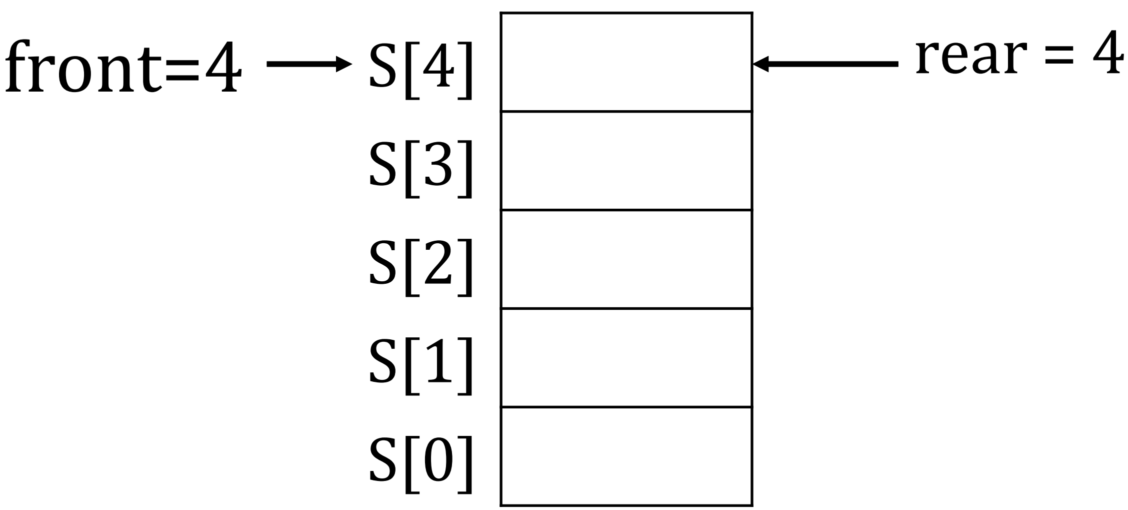


deQueue Operator

- deQueue():

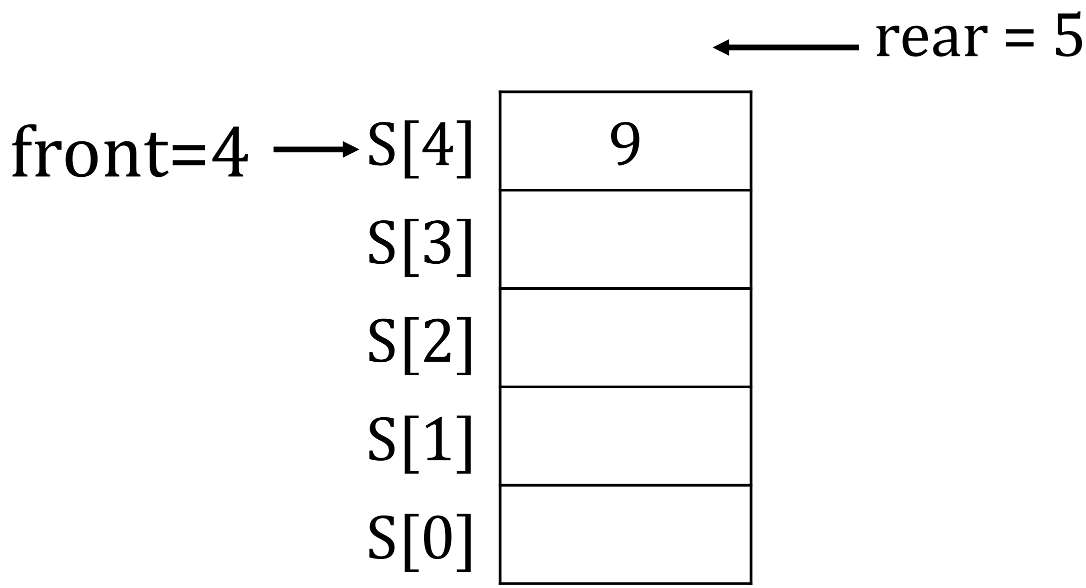
```
if(front < rear)
        front++
else
        Queue is empty, no deQueue
```

- deQueue (), deQueue(), deQueue(), deQueue()

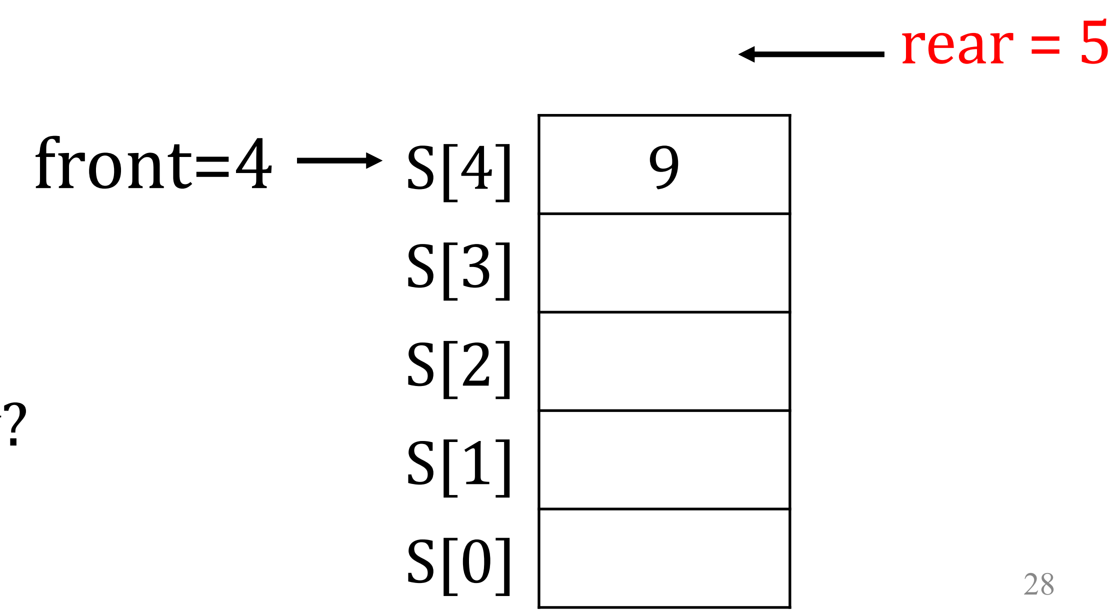


enQueue and deQueue

- enQueue(9)



- enQueue(10)
  - rear >= MAXSIZE
  - Queue is FULL!!!
  - Wrong OVERFLOW
  - S[0] to S[3] is empty?
  - How to address it?



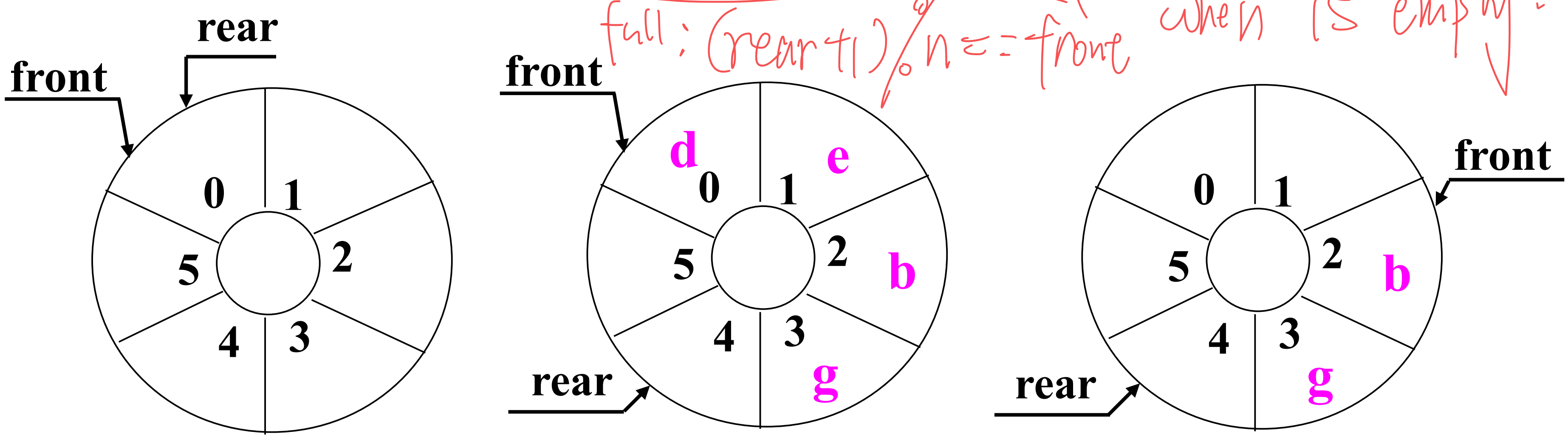
empty: rear==front,

Ring Queue

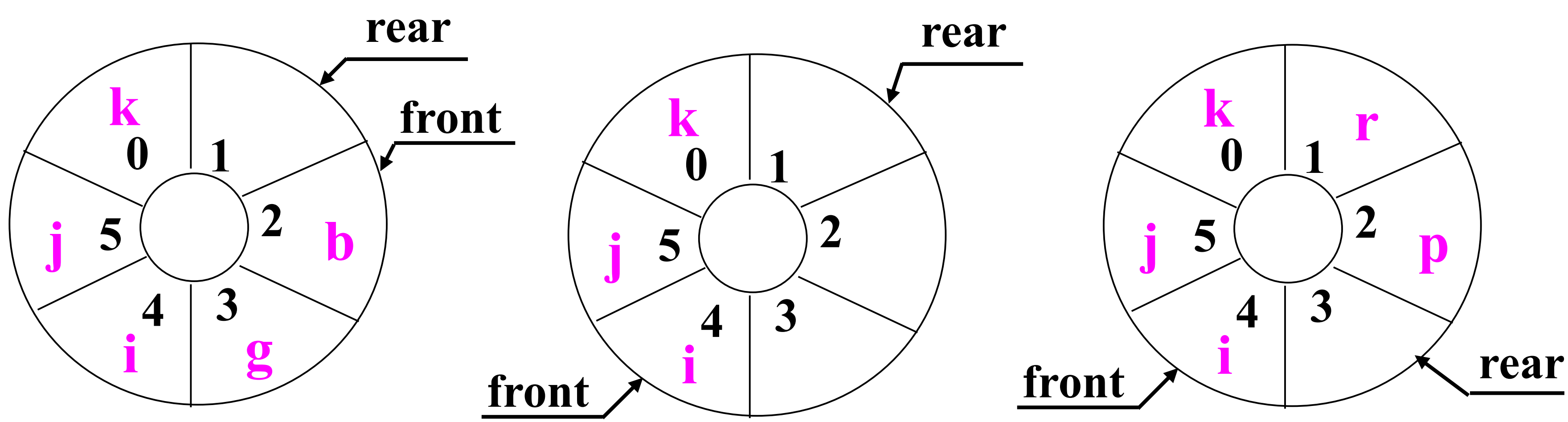
when is full

full: (rear+1)%n==front

when is empty.



(a) empty (b) d, e, b, g enQueue (c) d, e deQueue



(d) i, j, k enQueue (e) b, g deQueue (f) r, p, s, t deQueue

Application of Queues

- First-in first-out (FIFO) inventory control
  - OS scheduling: processes, print jobs, packets, etc.
  - Breadth-first traversal of a graph or level-order traversal of a binary tree (more on these later)
- Real applications
  - iTunes playlist.
  - Data buffers (iPod, TiVo).
  - Asynchronous data transfer (file IO, pipes, sockets).
  - Dispensing requests on a shared resource (printer, processor)

Our Roadmap

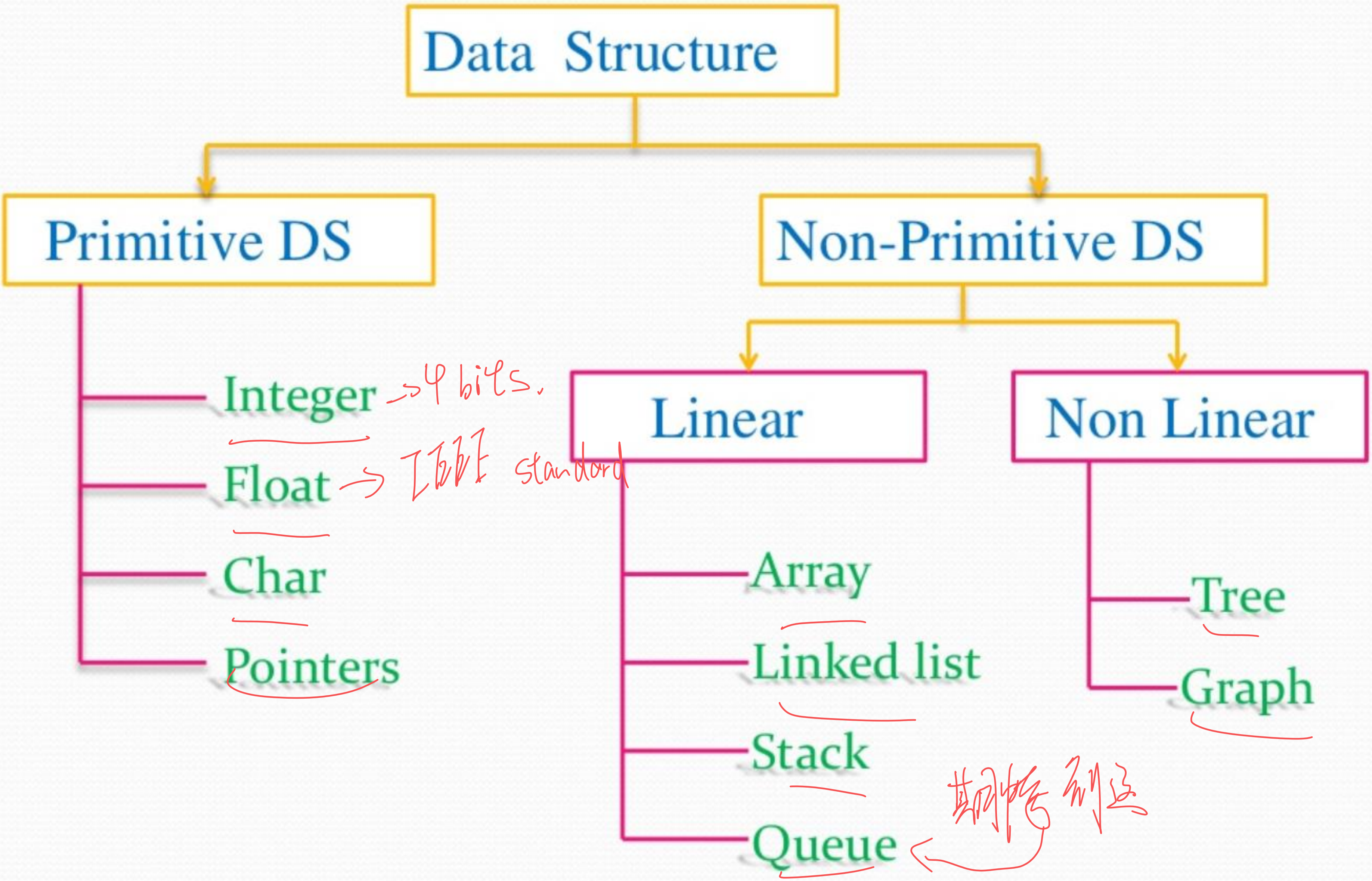
- Stack
- Queue
- Stack vs. Queue

Stack VS. Queue

map 的遍历算法.

	Stack	Queue
In-Out	FILO	FIFO
Application	function runtime	OS scheduling
Operations	push pop	enQueue, deQueue
Ops Time Complexity	O(1)	O(1)
Implementation	Array-based, Linked-based	Array-based, Linked-based

Data Structure



Our Roadmap

- Stack
- Queue
- Stack vs. Queue

Thank You!