Lecture 5
Stack and Queue

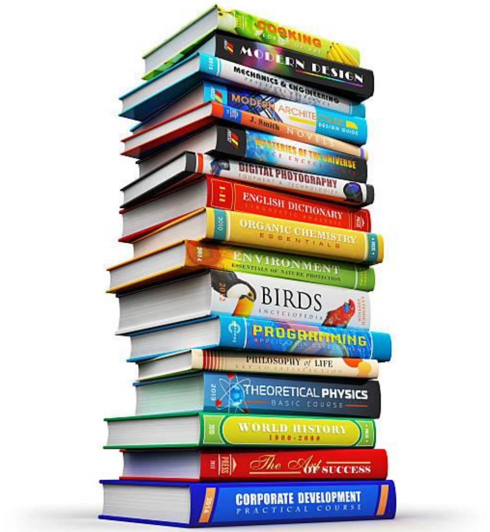
Bo Tang @ SUSTech, Fall 2022

Our Roadmap

- Stack
- Queue
- Stack vs. Queue

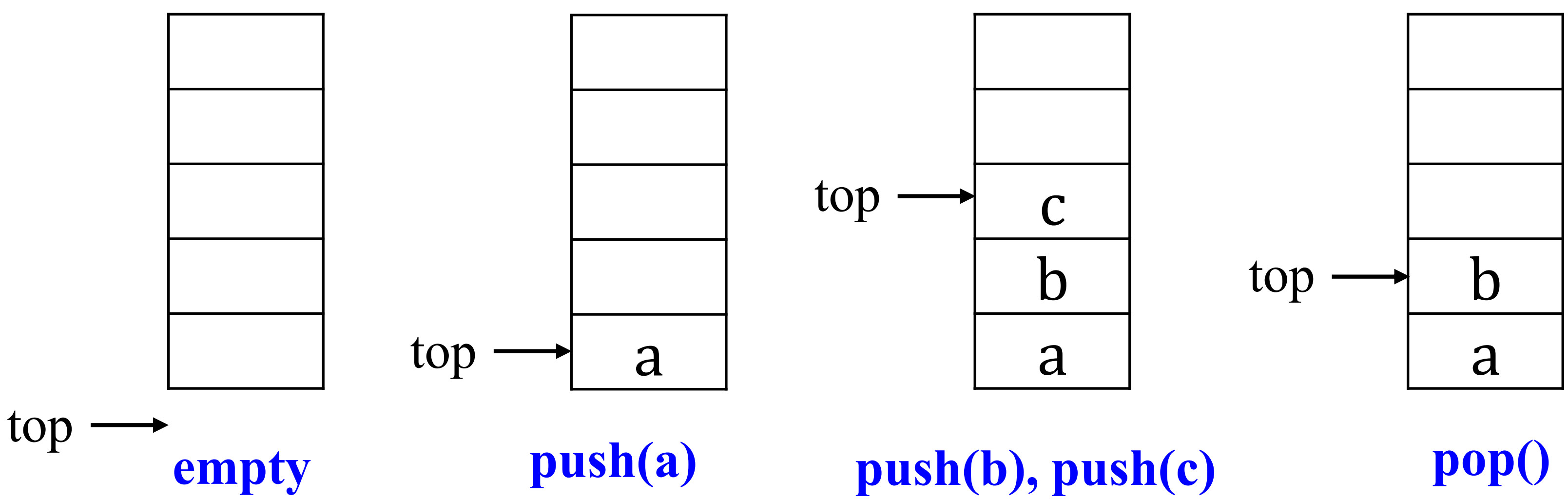
Stack

- 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



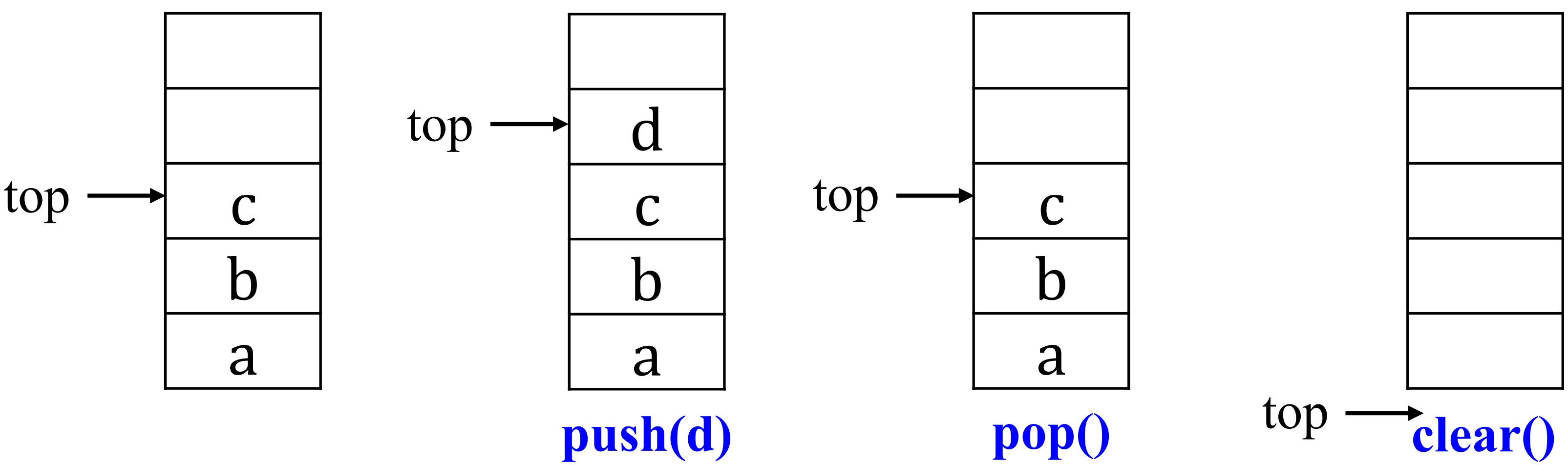
Stack

- 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



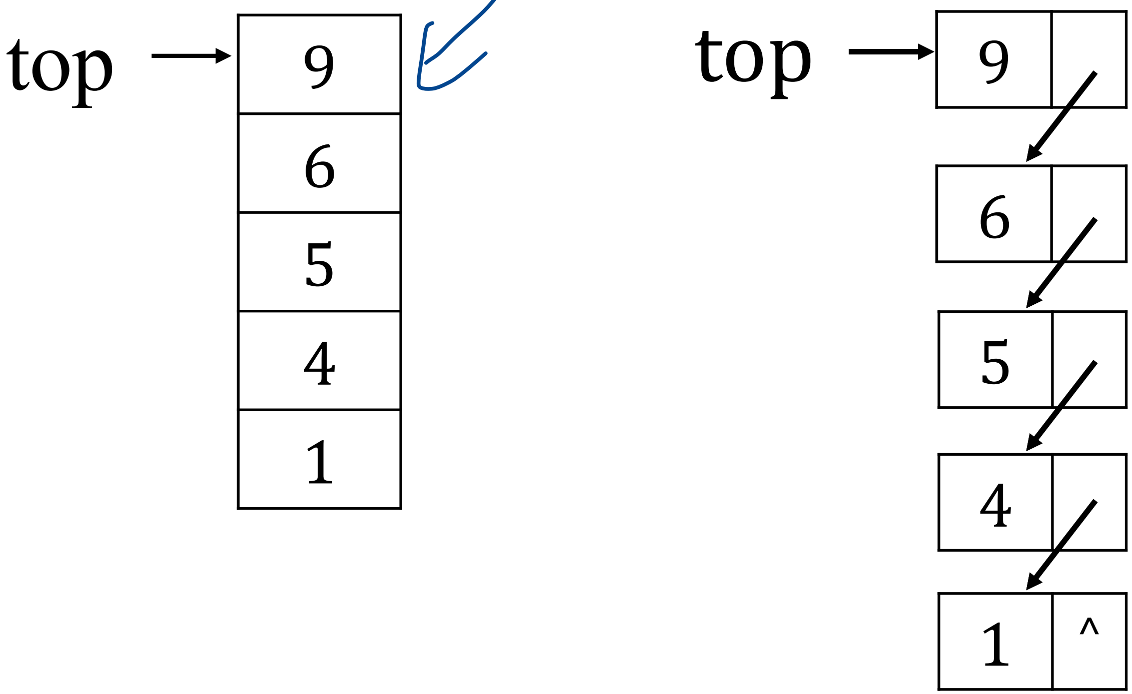
Stack Operation

- 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
- 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



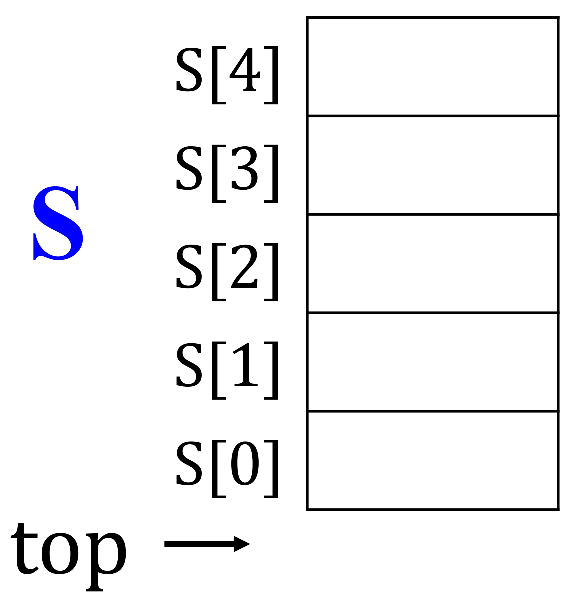
Implementation of Stack

- Array-based Stack
- Linked Stack



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



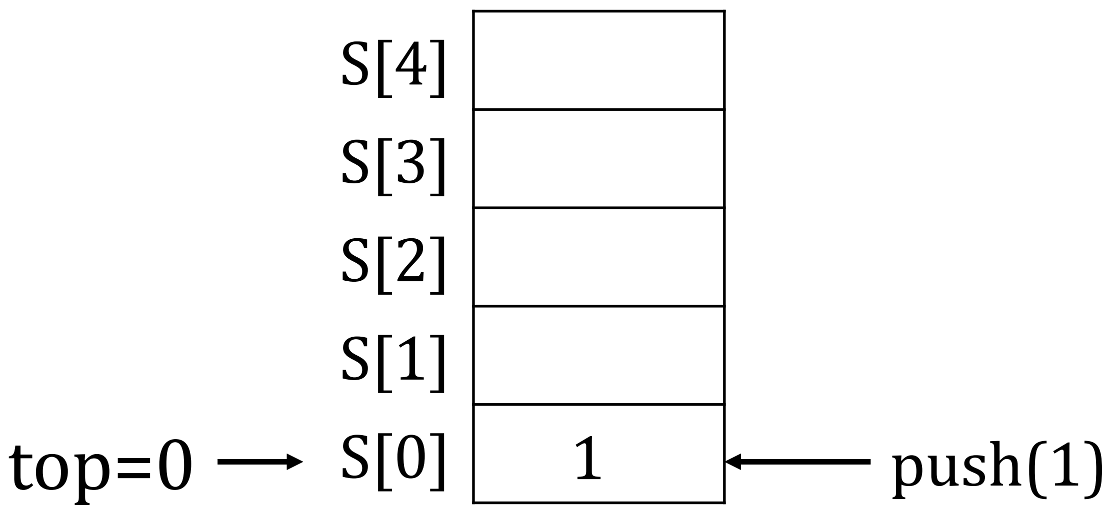
Push Operator

a method

- push(item):

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

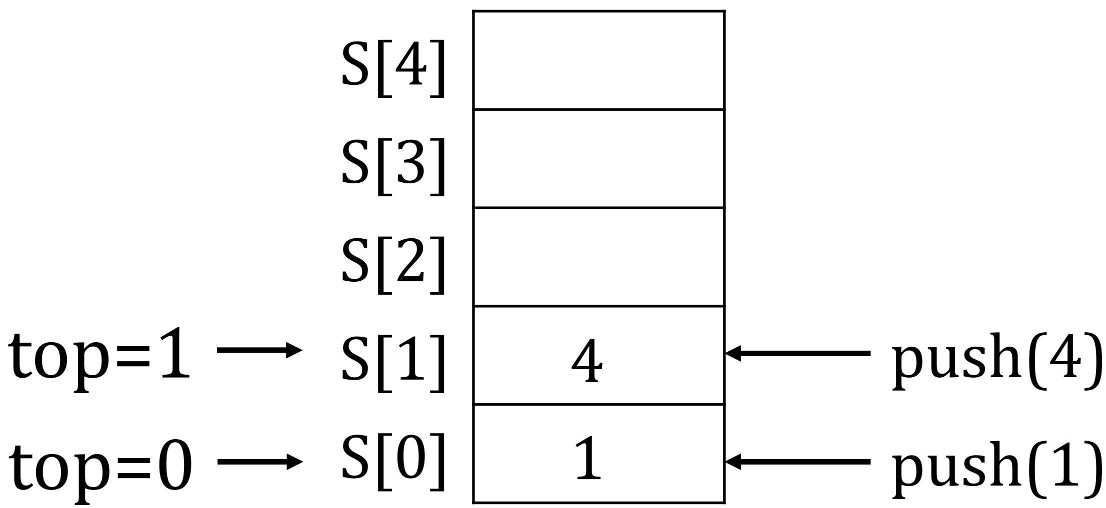
- push(1)



Push Operator

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

- push(1)
- push(4)

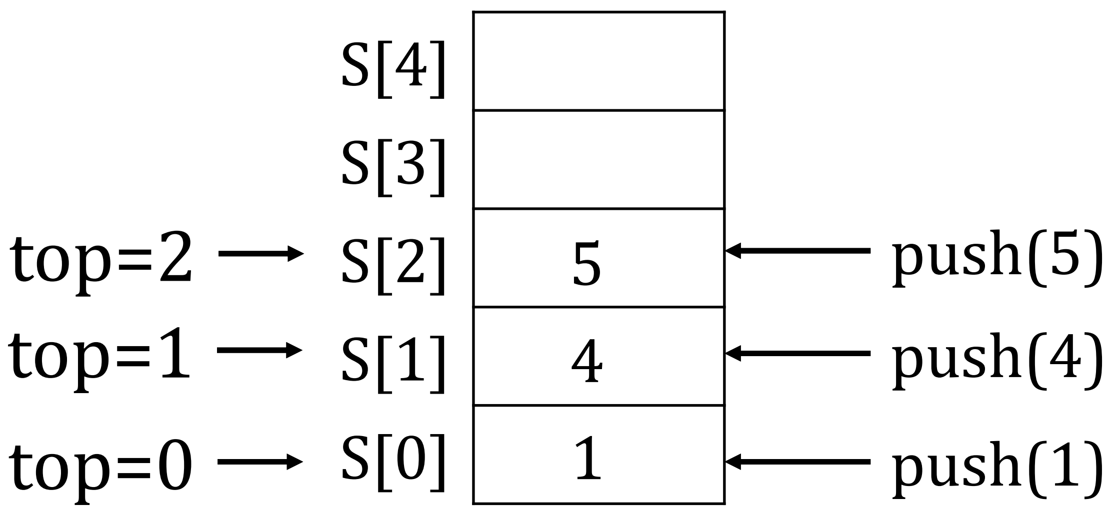


Push Operator

- push(item):

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

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

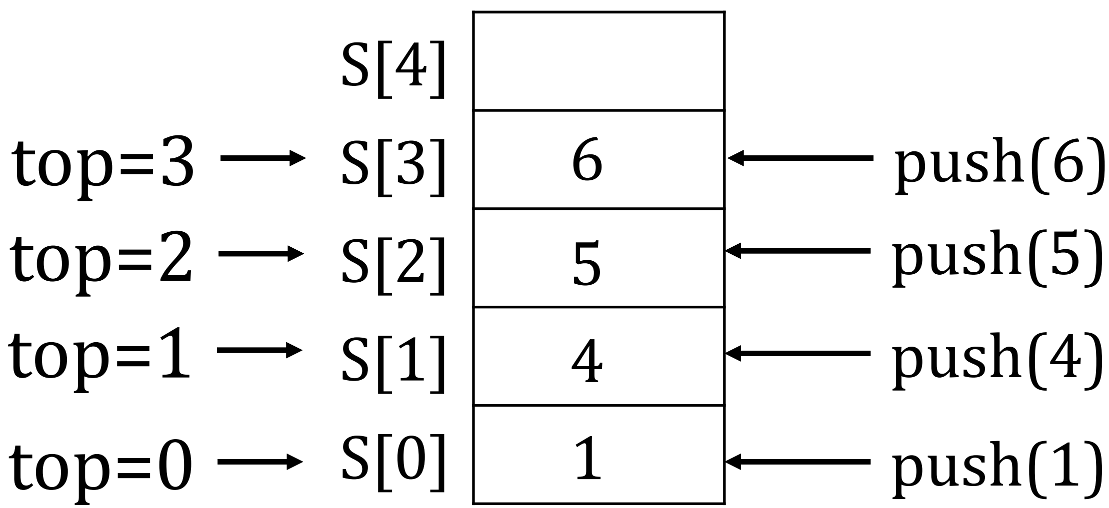


Push Operator

- push(item):

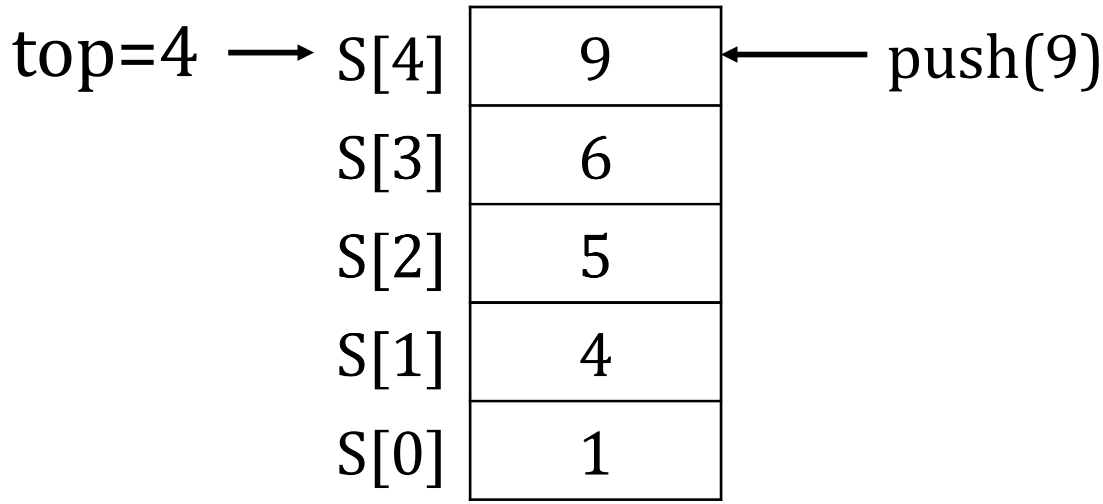
```
1. top++;
2. S[top] = item
```

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

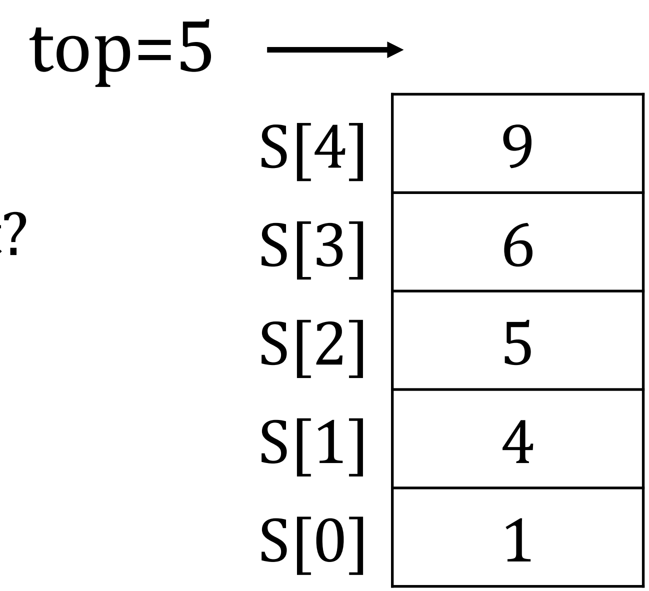


Push Operator

- 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--;
```

just --top is okay.

As we only can access the top.
when top++. the top item alway assigned.
So in pop we don't need to remove the top

Application of Stacks

- Making sure the delimiters (parens, brackets, etc.) are balanced:

  determine the limits or boundaries of

  - 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

left bracket
push it in
right bracket
pop one
NOT push this one in
check match ?

after input all
check stack is null ?

Arithmetic Expression Evaluation

the fix of operator.

- Arithmetic expression
  - operands (a, b, c), operator (+, *)
  - a + b * c
- Prefix expression 前缀表达式
  - + a * b c

b*c

- Infix expression 真序表达
  - a + b * c
- Postfix expression
  - a b c * +

Postfix Expression

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

  not need bracket
  - 5 9 3 + 4 2 * * 7 + *
- 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 + *   = 315
- 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

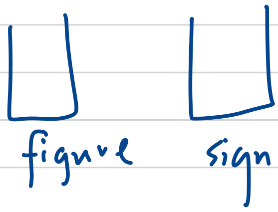
Postfix Expression Evaluation

- 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

implement of infix expression

· use 2 stacks to store figures & signs



· 读入数字可能有多位.
⇒ 每读入一位上一位乘10

· 读到符号
⇒ 符号之前为数. 符号. 数
可以运算
⇒ 不一定做上一个运算
注意优先级

· 最终只有 figure 中有一个数字

· 当所有东西读完之后.
再扫一遍.
此时计算的顺序一定是从上到下.

· 有括号:

单调栈

eg. 找第一个挡住的人

eg. Lab 3-C

基数排序

解决离散数据: 将每位划分进行桶排
^
桶排中

利用桶排v快的性质 同时↓空间复杂度

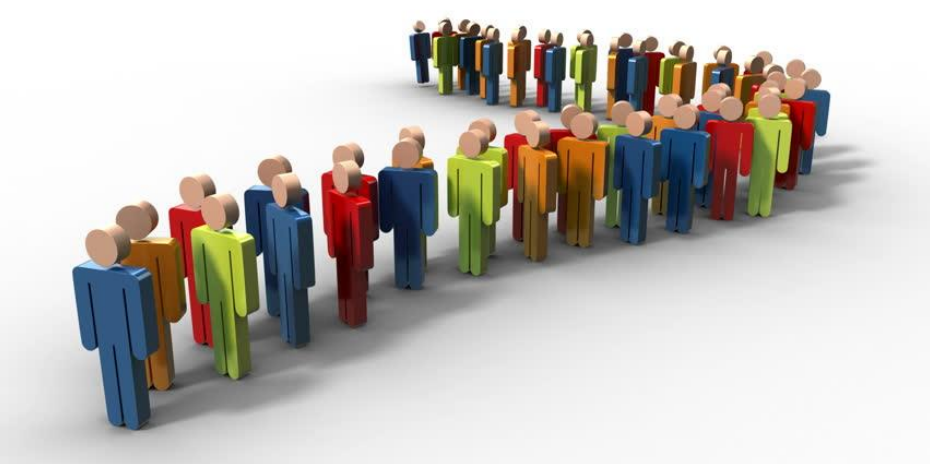
$t=O(nd)$

Our Roadmap

- Stack
- Queue
- Stack vs. Queue

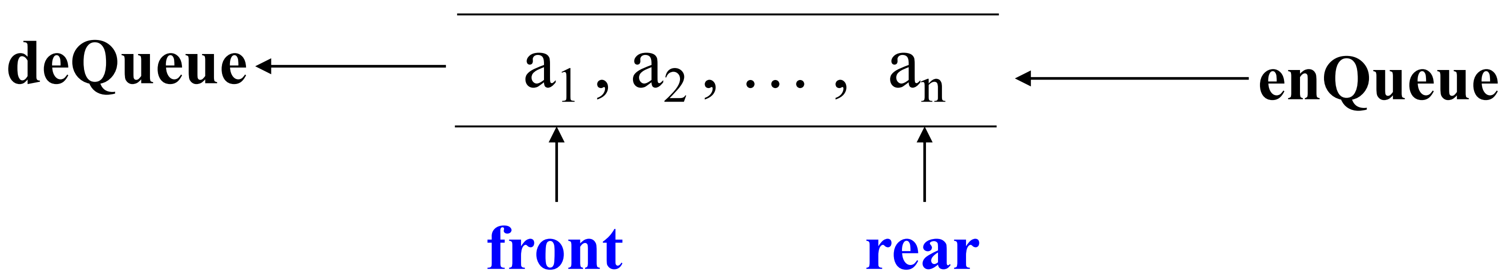
Queue

- 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



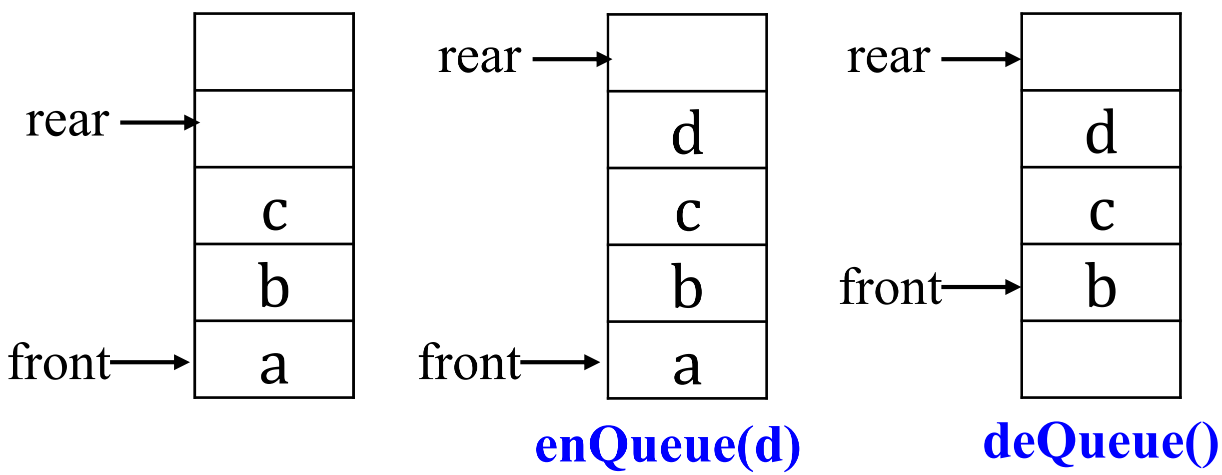
Queue

- 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



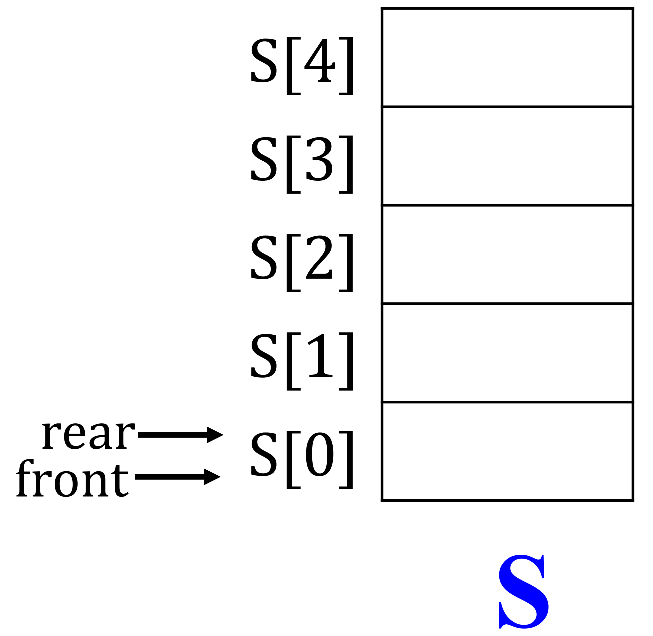
Queue Operations

- 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



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

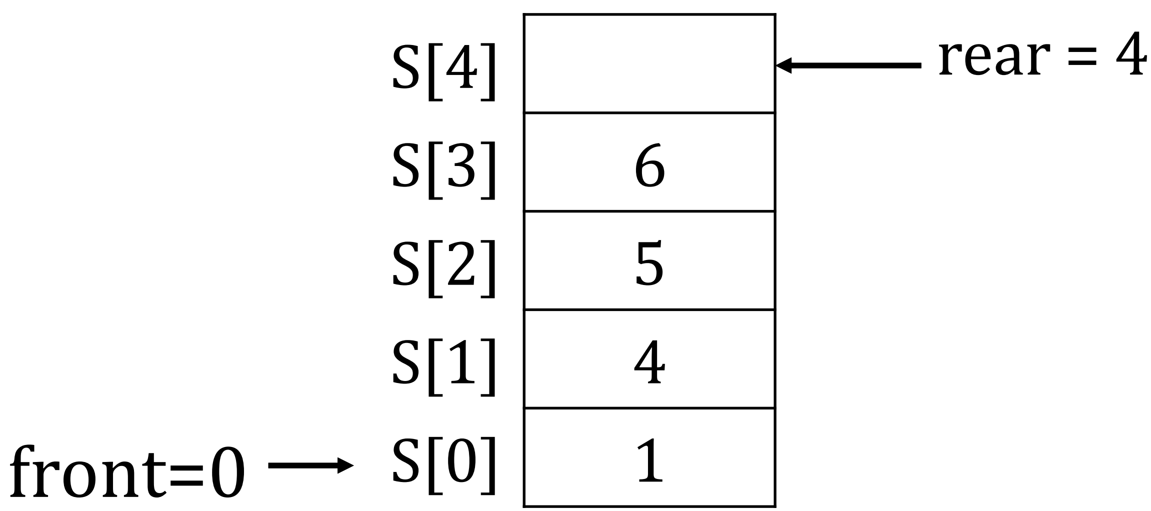


enQueue Operator

- enQueue(item):

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

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



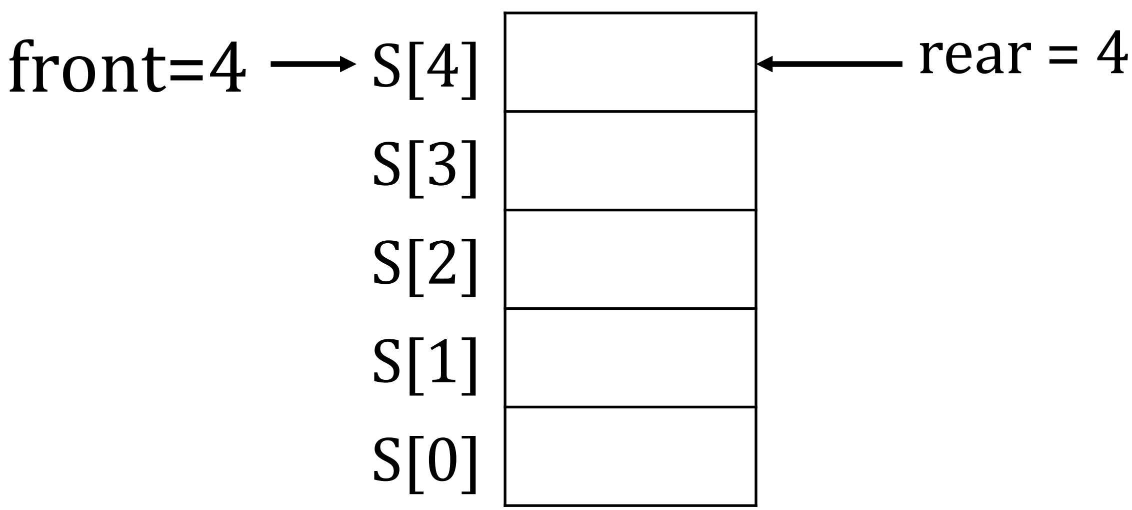
deQueue Operator

- deQueue():

```
1. if(front < rear)
2.         front++
3. else
4.         Queue is empty, no deQueue
```

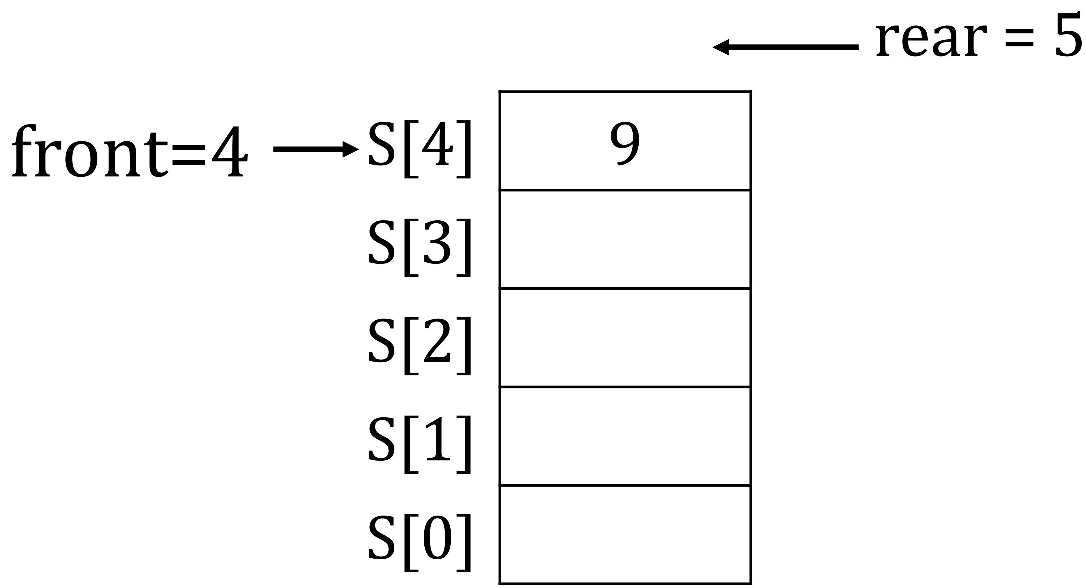
Do NOT need to remove the value stored in front

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

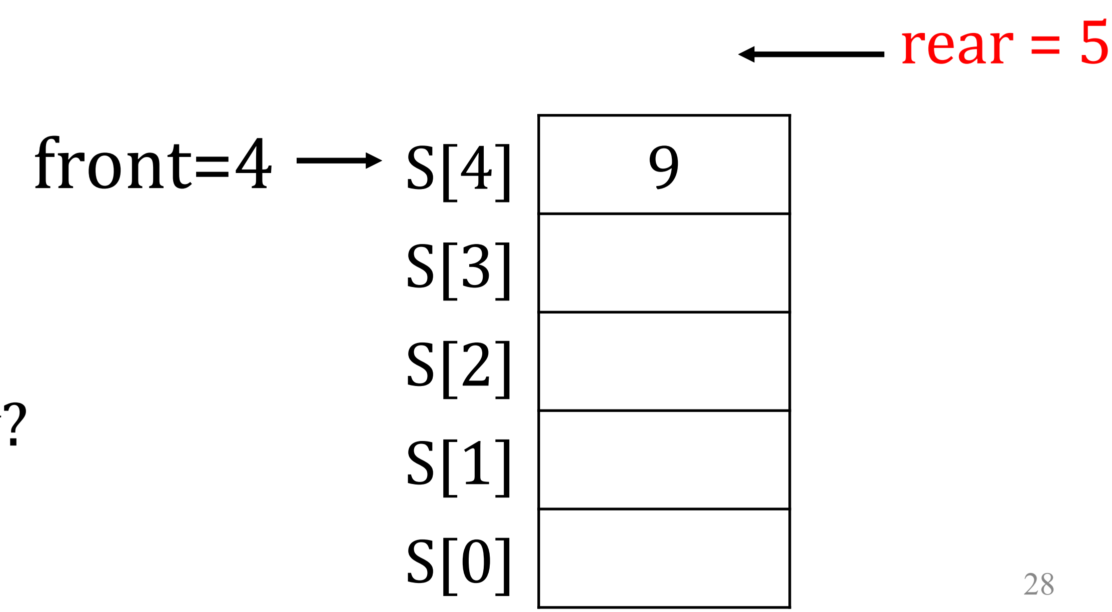


enQueue and deQueue

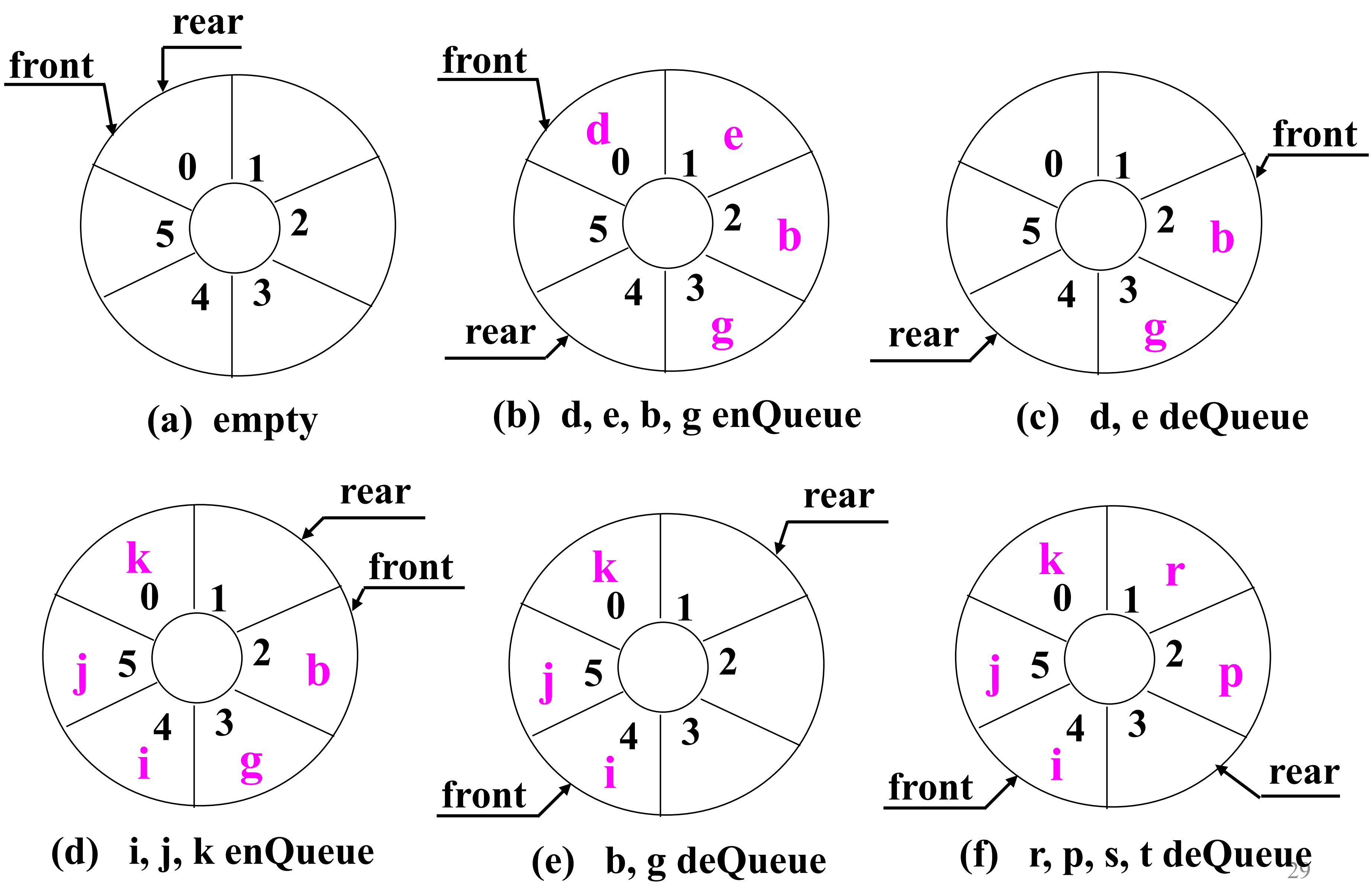
- enQueue(9)



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



Ring Queue



(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)

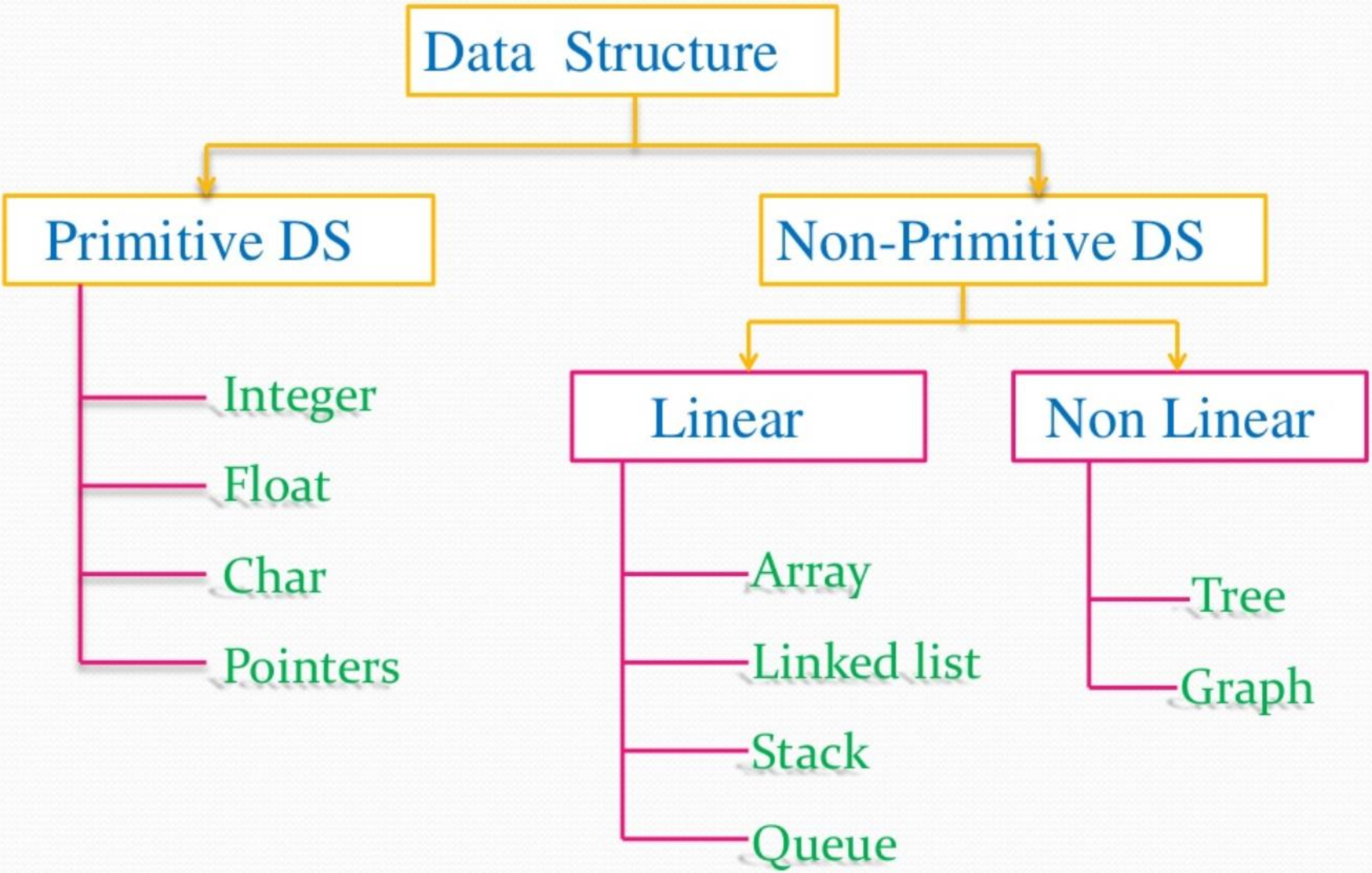
Our Roadmap

- Stack
- Queue
- Stack vs. Queue

Stack VS. Queue

	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!