CSE 105:
Data Structures and Algorithms-I
(Part 2)

Instructor

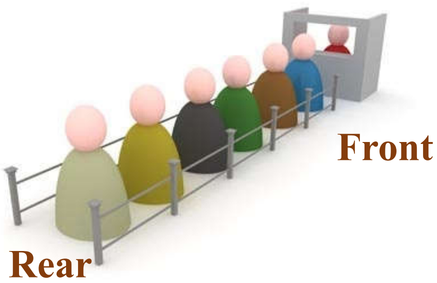
Dr Md Monirul Islam

Queues

Review

FIFO: First in, First Out

Restricted form of list: Insert at one end, remove from the other.

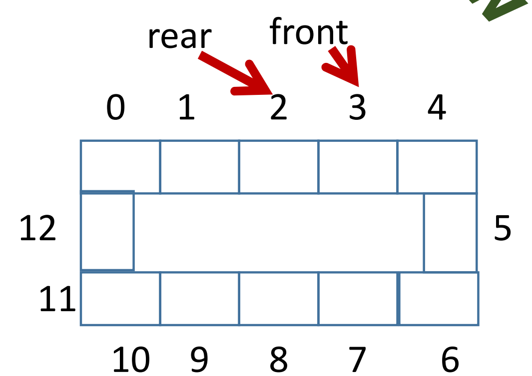


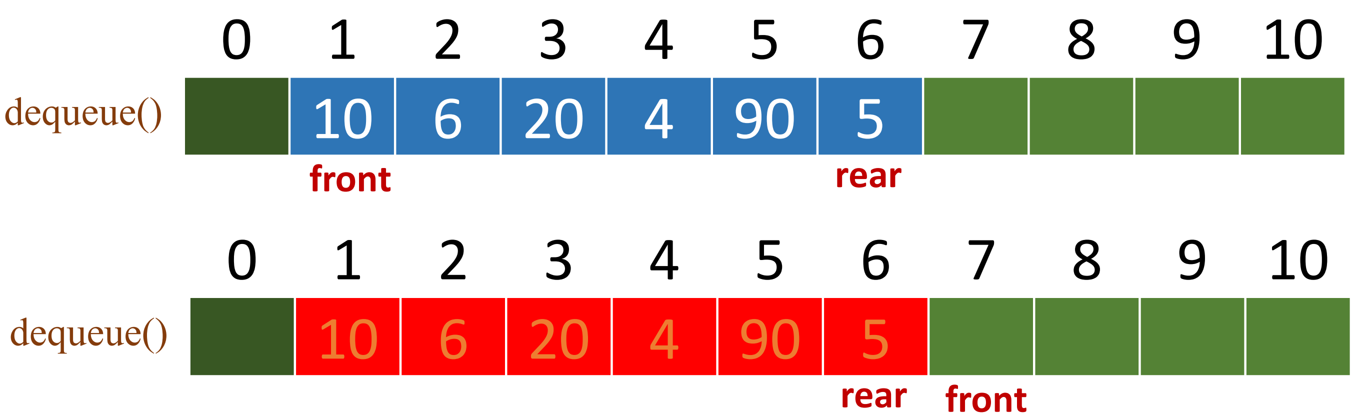
How to differentiate empty and full queue

Review

- When is empty

use special case: rear = front=-1;





Array based Implementation of Queue

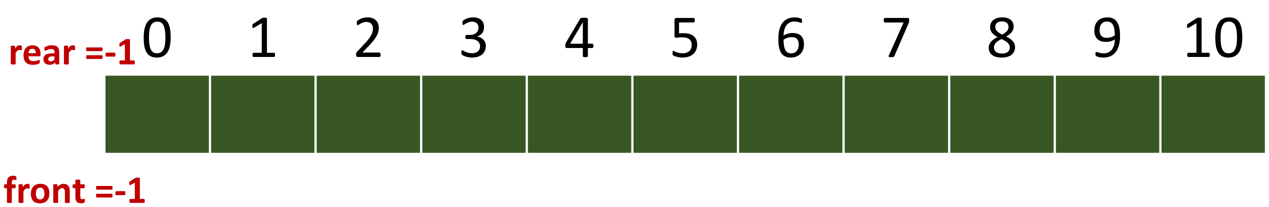
Review

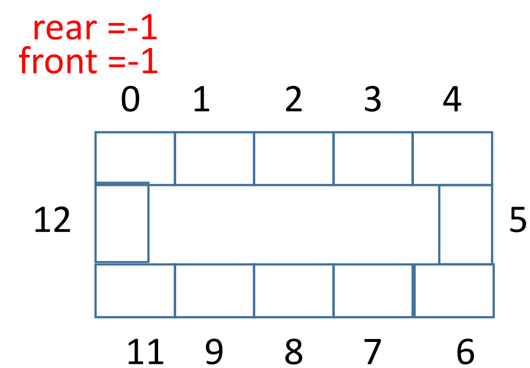
int maxSize**; // Max size of queue**
int front, rear;
int *array;

isEmpty()
return rear== -1;

isFull()
return (rear+1)% maxSize ==front;

initialize ()
front = rear = -1;
array = //make necessary allocation of size maxSize;





Array based Implementation of Queue

Review

```
enqueue(int data)
if (isFull() ) //error handling
else
     rear=(rear+1)%maxSize
     array[rear] =data;
     if front==-1  //first element
         front=0;
```

```
dequeue()
if (isempty() ) //error handling

else
     data=array[front];
     if (front==rear)    //last element
          front=rear=-1;
     else
           rear=(rear+1)%maxSize

      return data
```

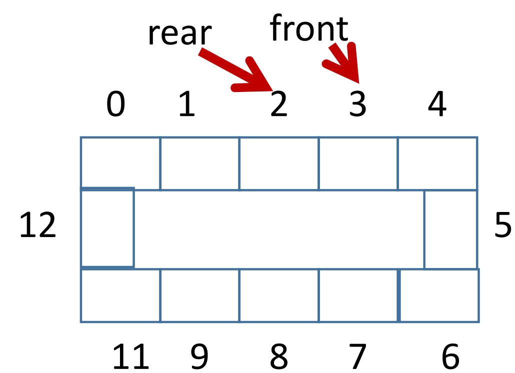
How to differentiate empty and full queue

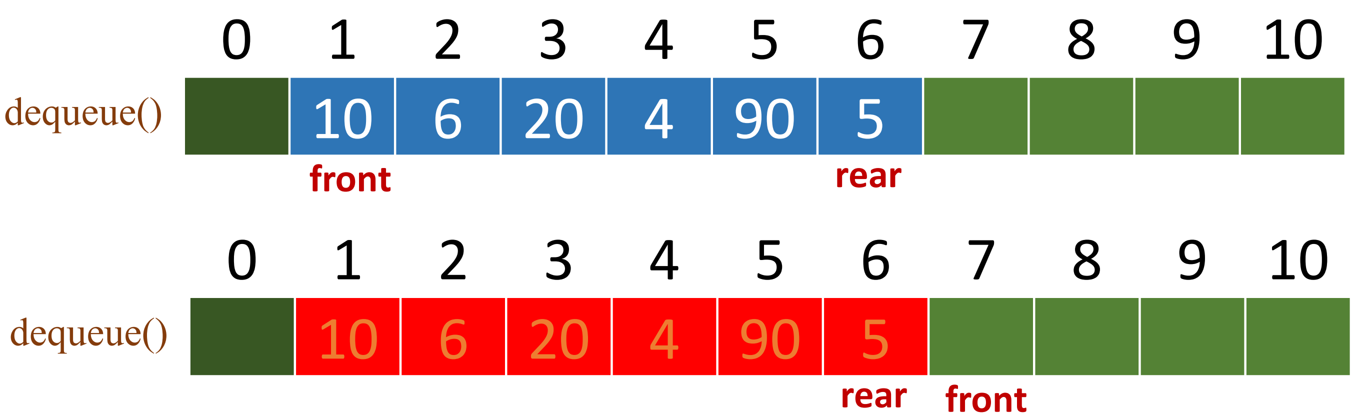
- When is empty

Other solutions:

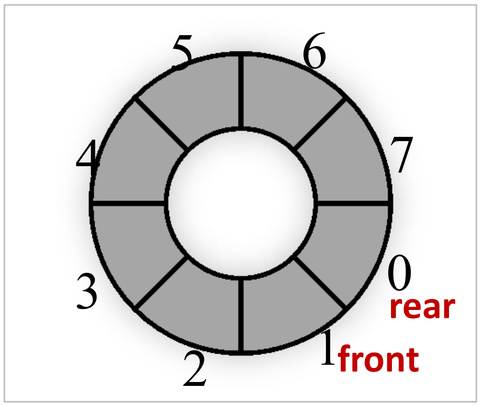
1. use a counter to monitor queue size
2. store up to n-1 elements in a queue of size n

(e.g., 1 dummy space)

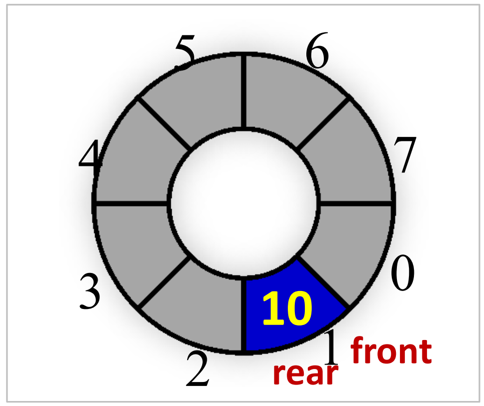




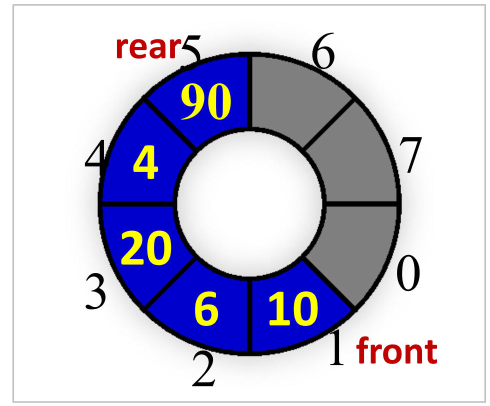
Queue management with dummy space



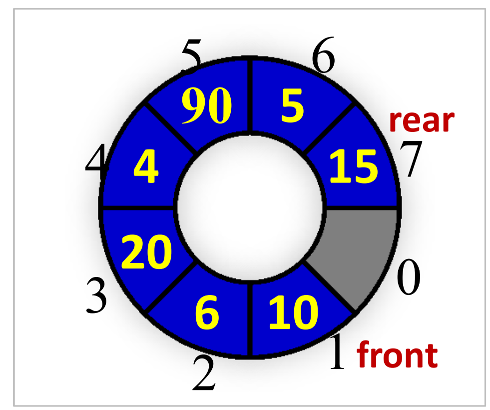
empty



First enqueue

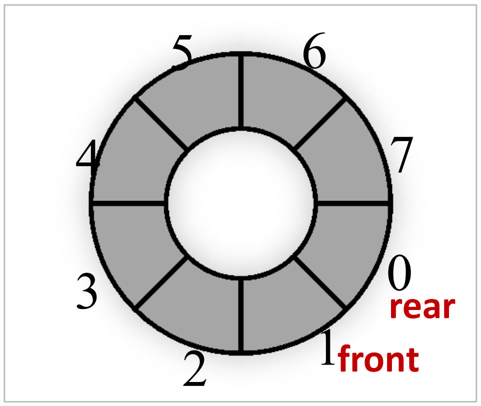


5 enqueues

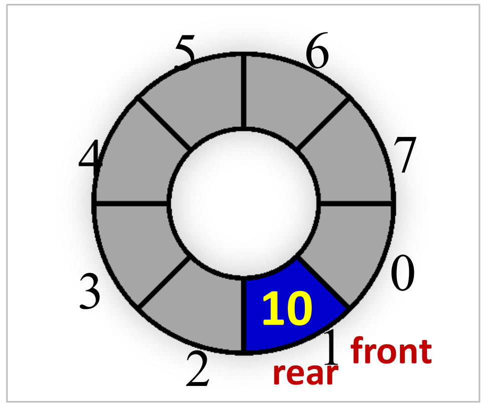


Full

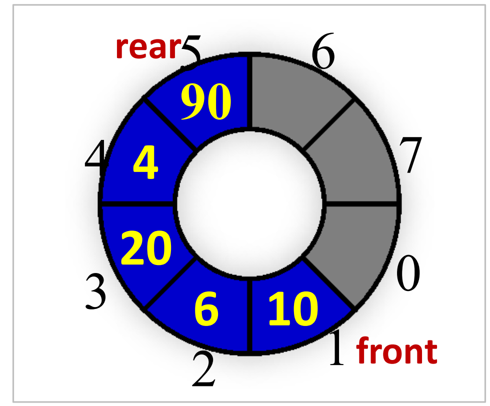
Queue management with dummy space



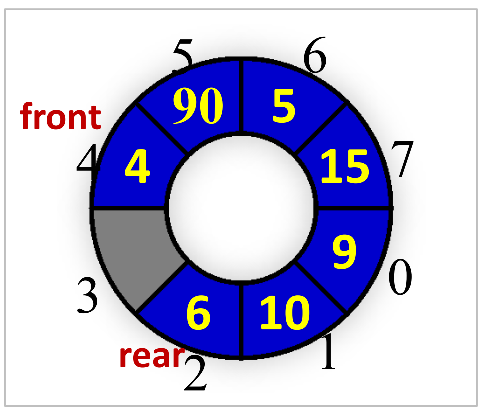
empty



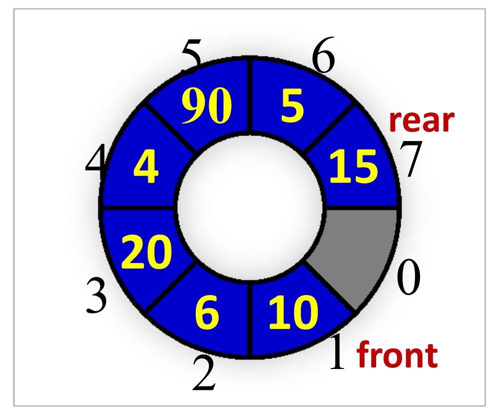
First enqueue



5 enqueues



Full

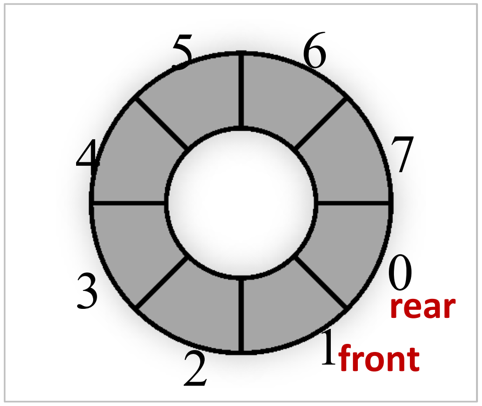


Full

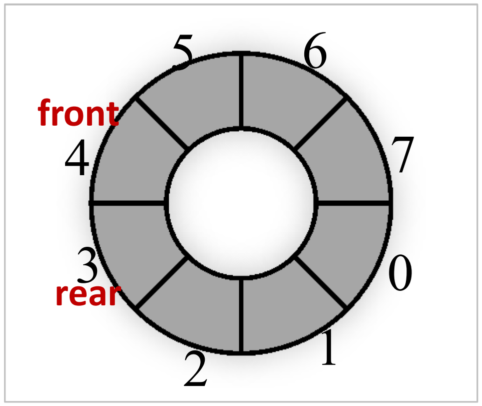
Queue management with dummy space

initialize ()
rear = 0; front = 1;
array = //make necessary allocation of size maxSize;

isEmpty()
return (maxSize +rear- front + 1) % maxSize ==0;



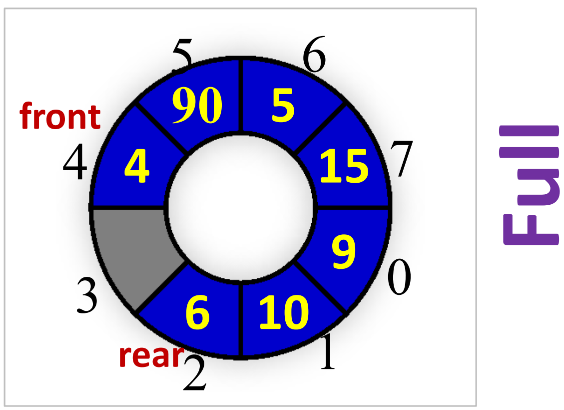
empty

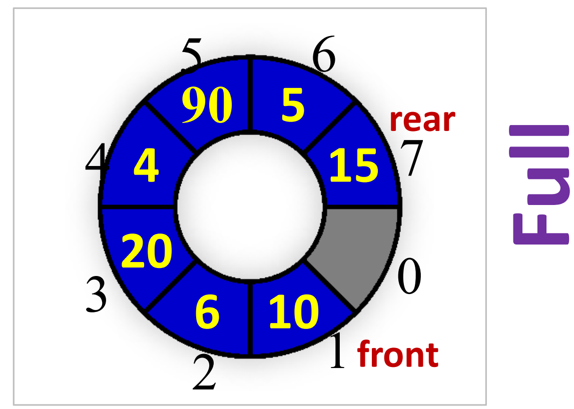


Queue management with dummy space

isFull()

return (rear+2) % maxSize == front;





Queue with resizable array

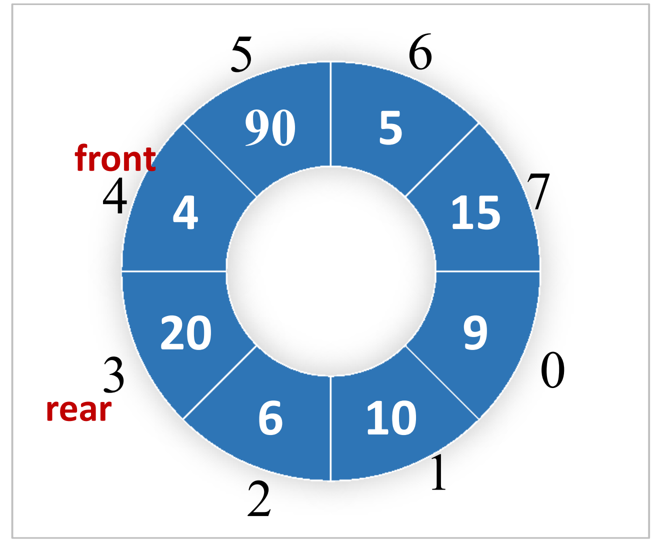
enqueue(int data)

```
if (isFull() )
  doubleQueueArray();
rear=(rear+1)%maxSize
array[rear] =data;
if front==-1  //first element
    front=0;
```

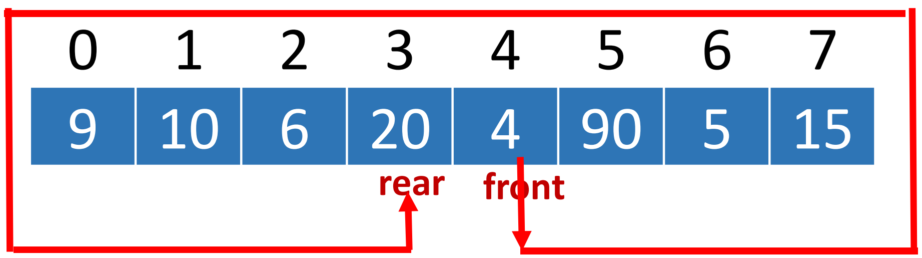
```
doubleQueueArray() //simple version
1. int *temp = //allocate memory for 2*maxSize elements
2. Copy all elements from array to temp;
3. free (array);
4. array = tamp;
5. maxSize=2*maxSize;
```

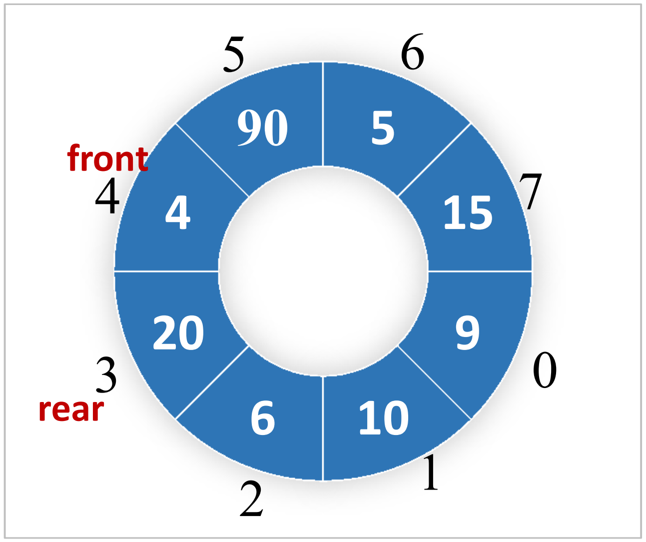
Queue with resizable array

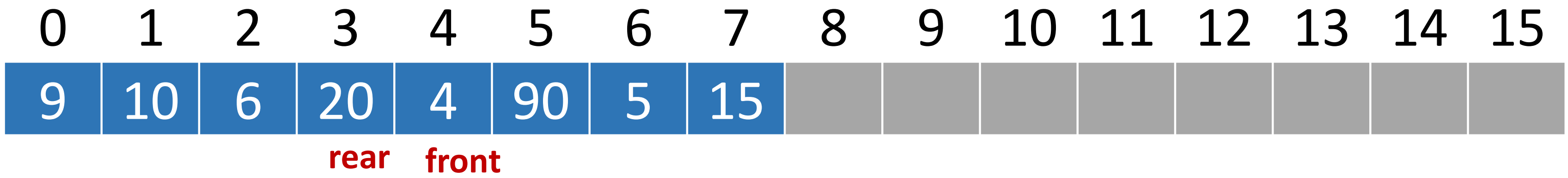
0	1	2	3	4	5	6	7
9	10	6	20	4	90	5	15
			rear	front			



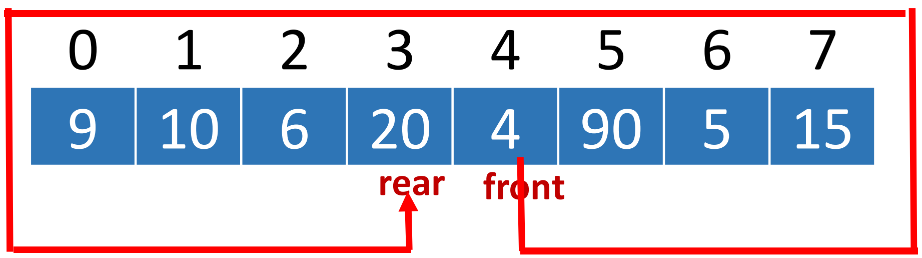
Queue with resizable array

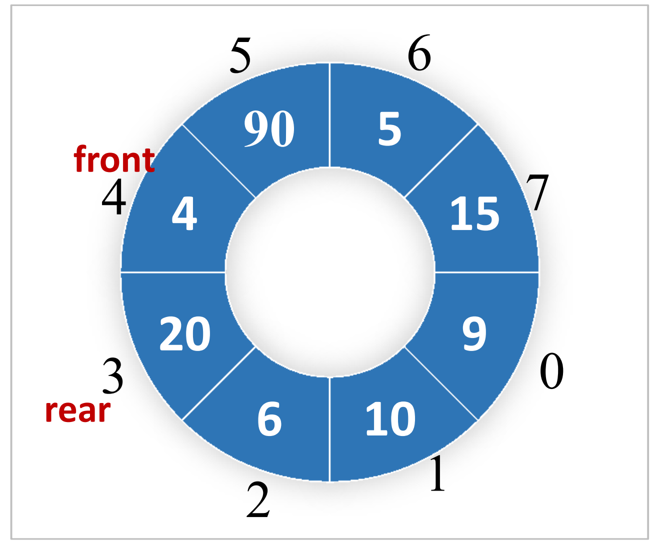


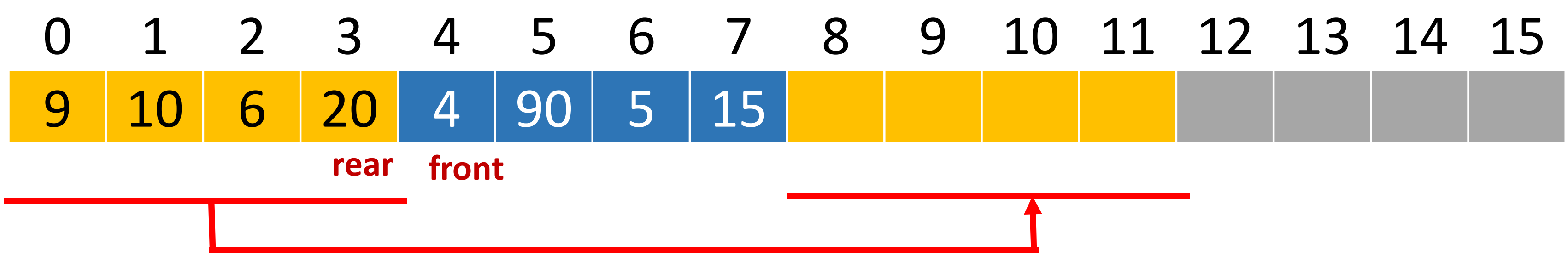




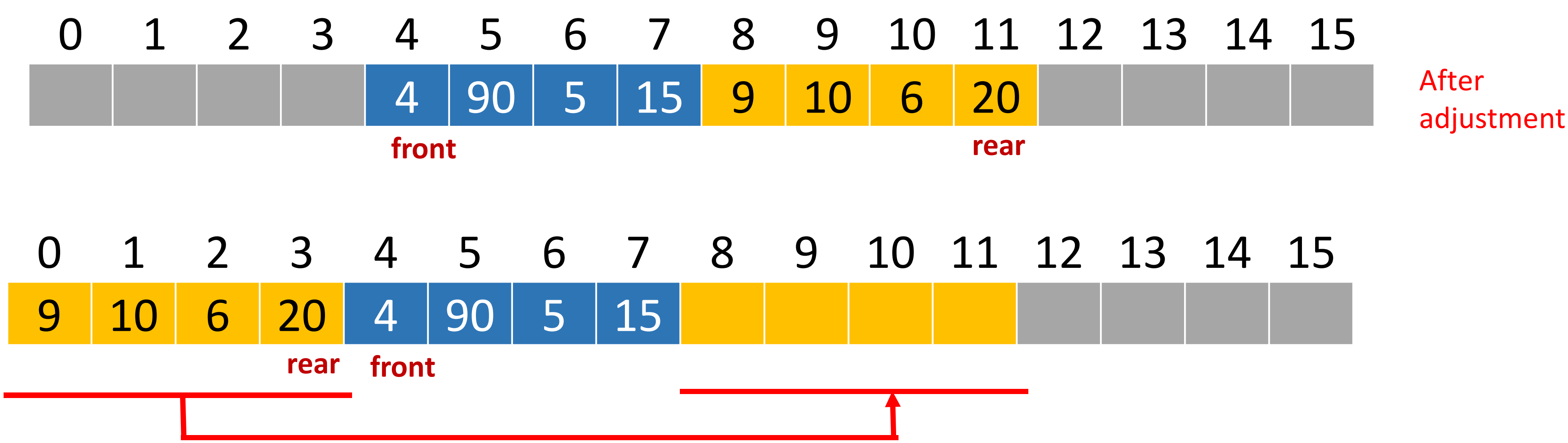
Queue with resizable array





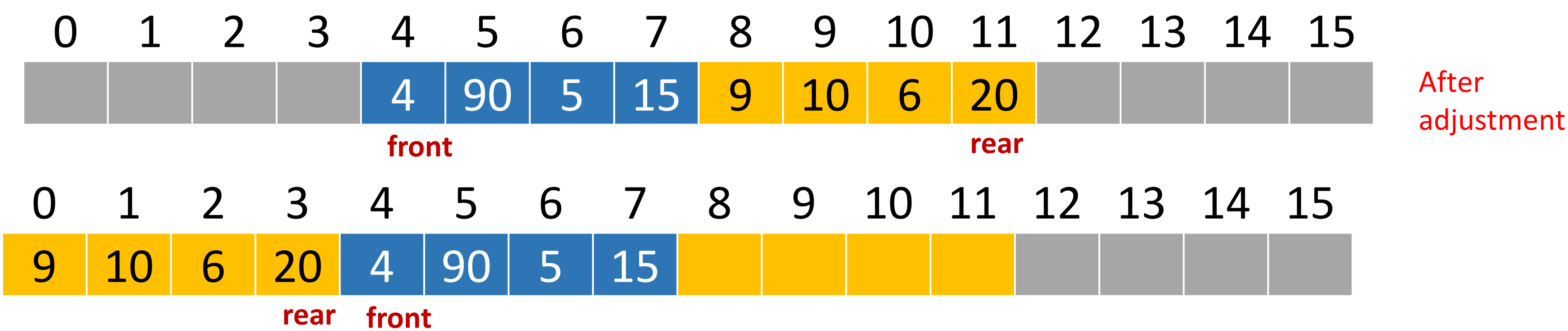


Queue with resizable array



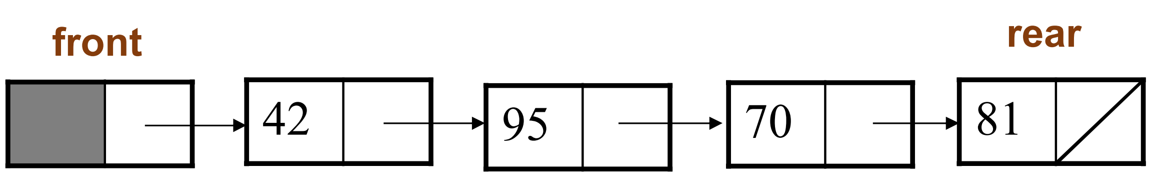
Queue with resizable array

doubleQueueArray() //updated version
1. int *temp = //allocate memory for 2*maxSize elements
2. Copy all elements from array to temp;
3. free (array);
4. array = tamp;
5. if front>rear
    move arr[0 .. rear] to array[maxSize . . . maxSize+rear]
    rear=rear+maxSize;
6. maxSize *=2;



Linked List Based Queue

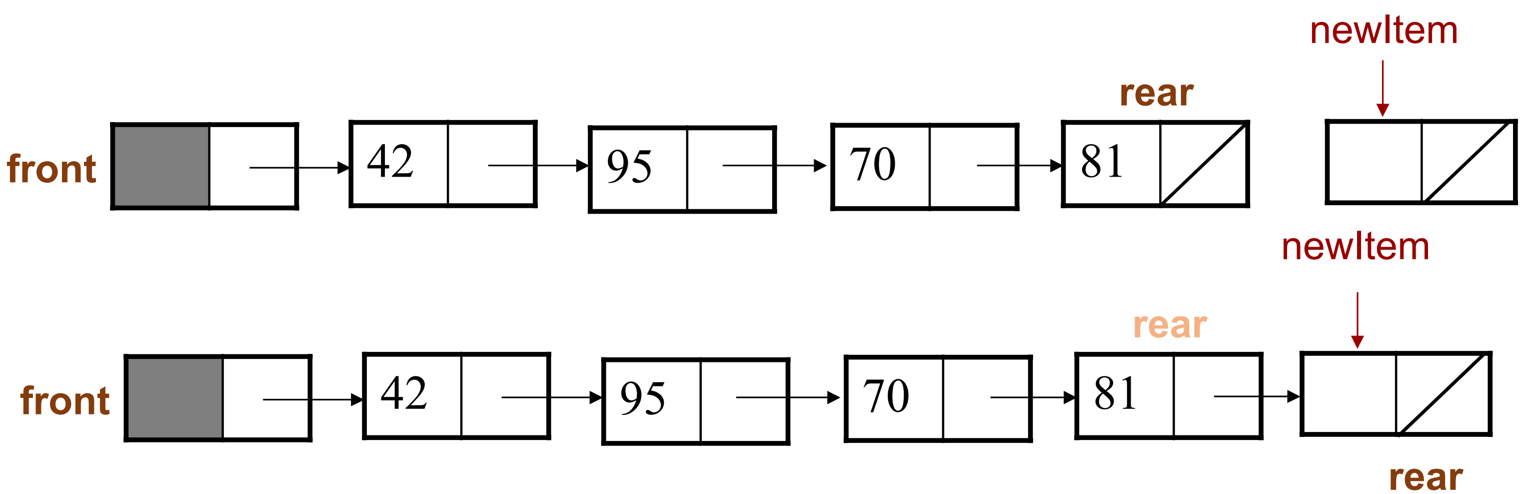
1. Use a header node for simpler implementation
2. front points to header and tail points to the last node
3. Enqueue places new element after current rear
4. Dequeue removes and returns the first element after the header



Linked List Based Queue: enqueue

1. rear->next = newItem
2. rear=rear->next

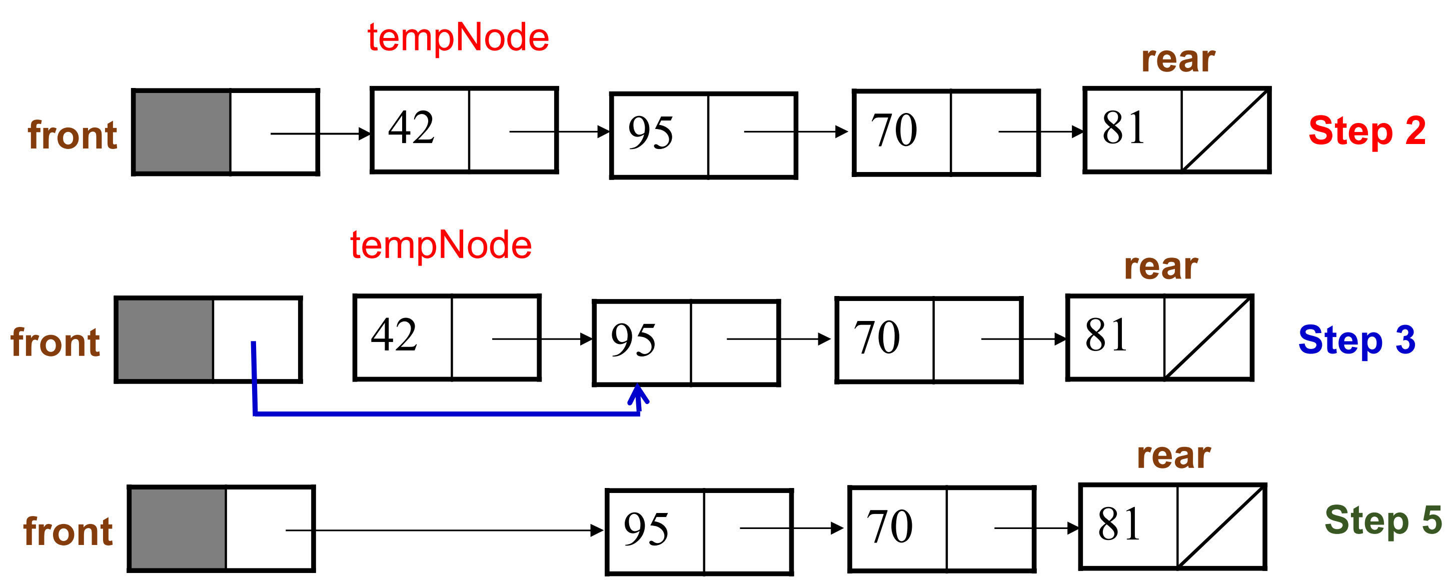
Complexity?



Linked List Based Queue: dequeue

1. int data = front->next->element; // Store dequeued value
2. node* tempNode = front->next; // Hold dequeued link
3. front->next = tempNode->next; // Advance front
4. if (rear == tempNode) rear = front; // Dequeue last element
5. delete tempNode; // Delete link

Complexity?



Queue operations: Complexity

Operations	Array	Dynamic Array	Linked List
Space Complexity (for n EnQueue operations)	O(n)	O(n)	O(n)
Time Complexity of EnQueue()	O(1)	O(1) (Average)	O(1)
Time Complexity of DeQueue()	O(1)	O(1)	O(1)
Time Complexity of QueueSize()	O(1)	O(1)	O(1) *
Time Complexity of IsEmptyQueue()	O(1)	O(1)	O(1)
Time Complexity of IsFullQueue()	O(1)	O(1)	**N/A**
Time Complexity of DeleteQueue()	O(1)	O(1)	O(n)

Returning to Stack

Stack Applications

- Implementing function calls in a compiler.
- Arithmetic expression evaluation
- Parsing in a compiler.
- Undo in a word processor.
- Back button in a Web browser.
- . . .

Arithmetic expression evaluation

Goal: Evaluate infix expressions.

(1 + ((2 + 3) * (4 * 5)))

operand operator

Arithmetic expression evaluation

Goal: Evaluate infix expressions.

(1 + ((2 + 3) * (4 * 5)))

operand operator

Two-stack algorithm. [E. W. Dijkstra]

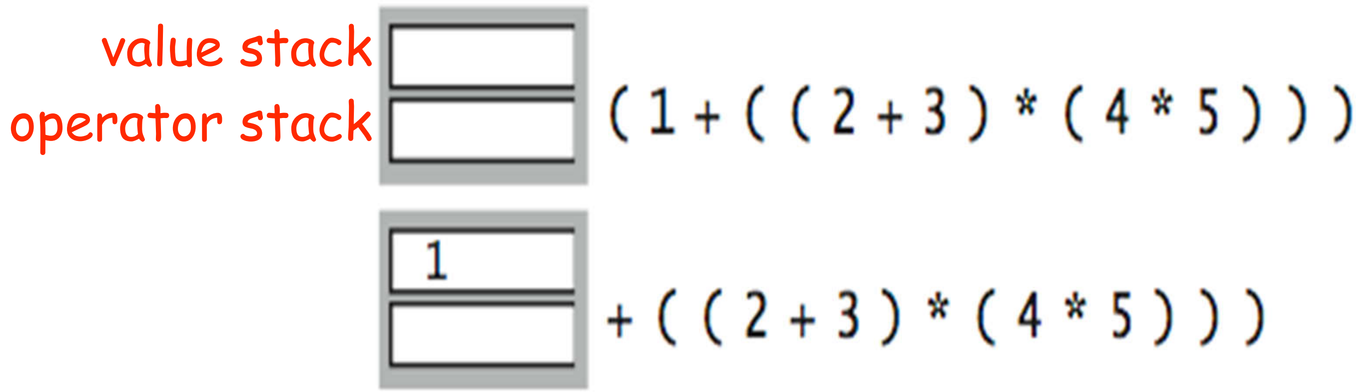
- Value: push onto the value stack.
- Operator: push onto the operator stack.
- Left parenthesis: ignore.
- Right parenthesis : pop operator and two operands;
  calculate new operand
  push the new operand onto the operand stack.

Arithmetic expression evaluation

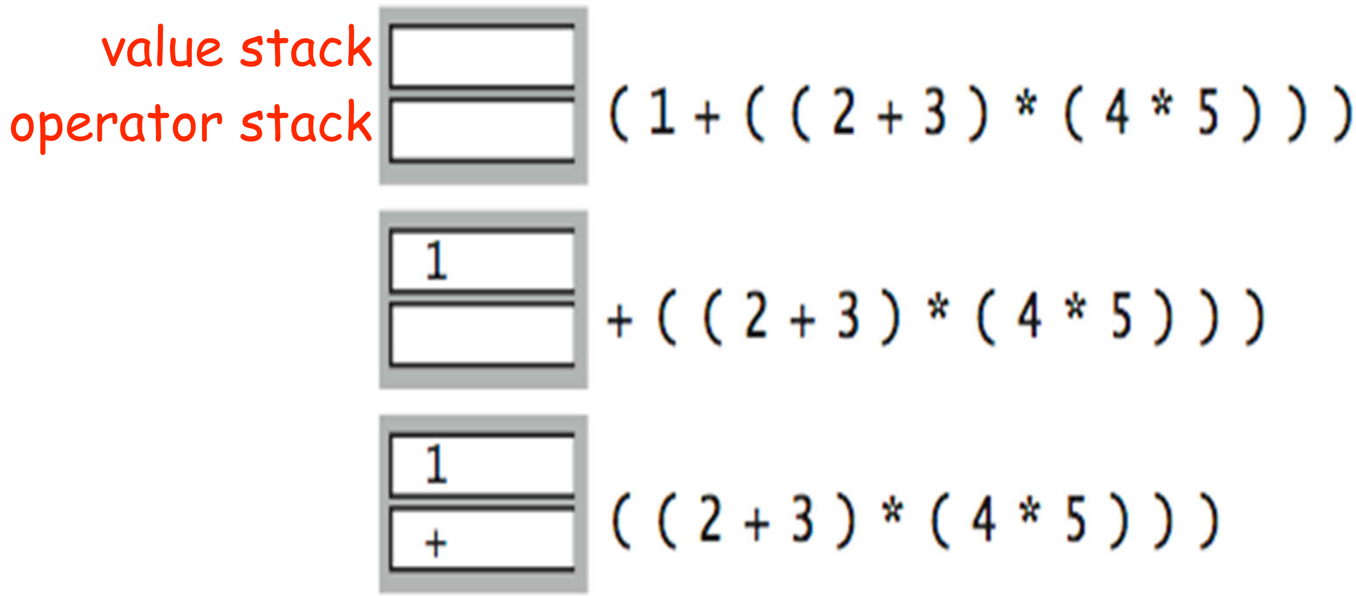
value stack
operator stack

(1 + ((2 + 3) * (4 * 5)))

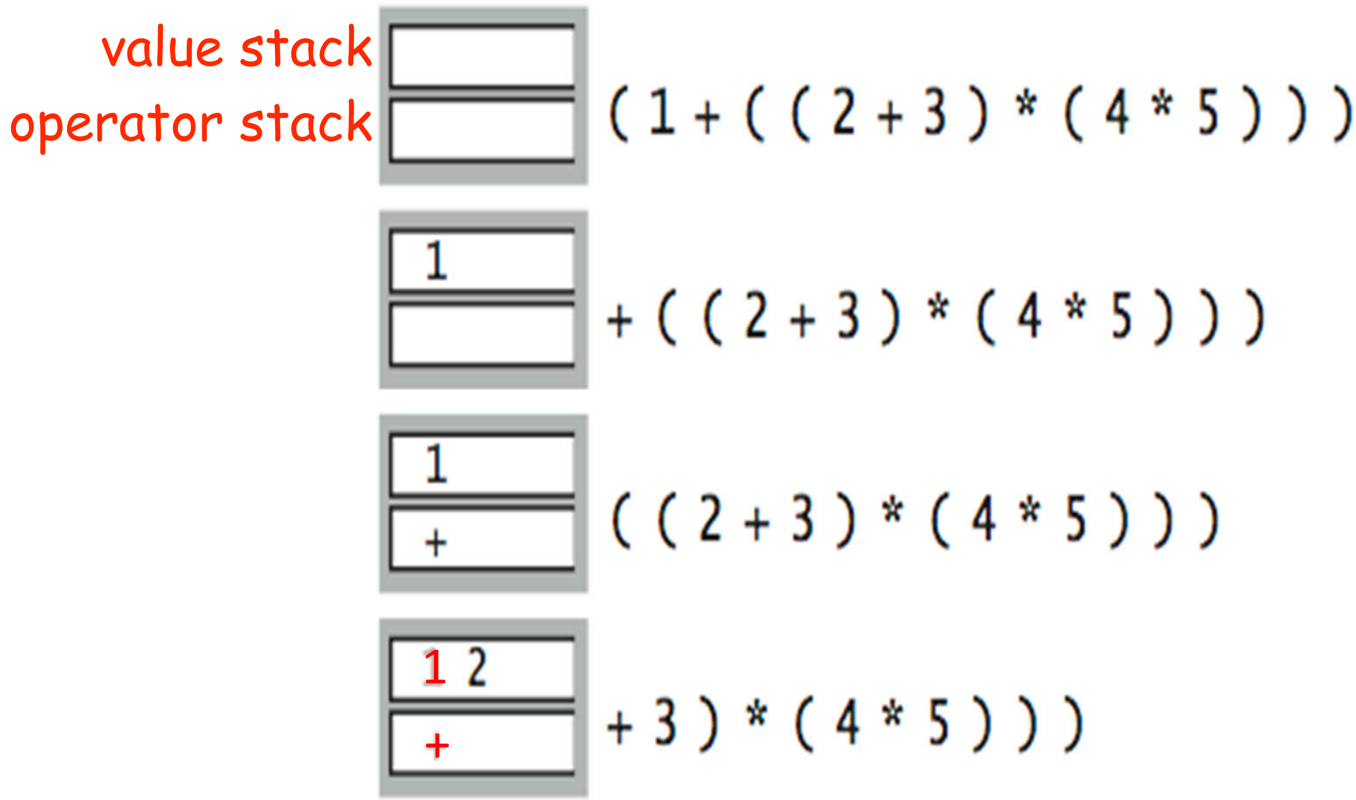
Arithmetic expression evaluation



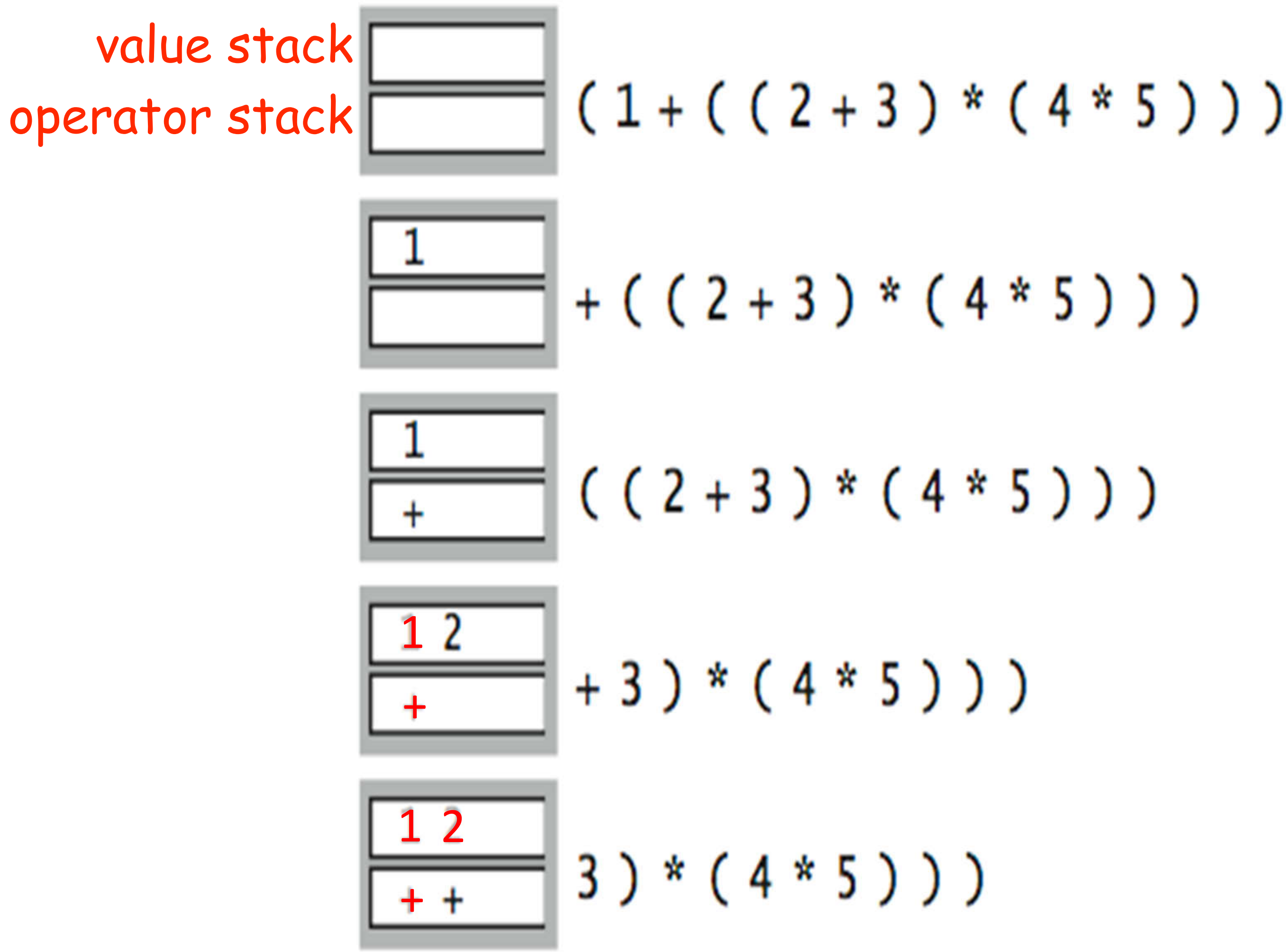
Arithmetic expression evaluation



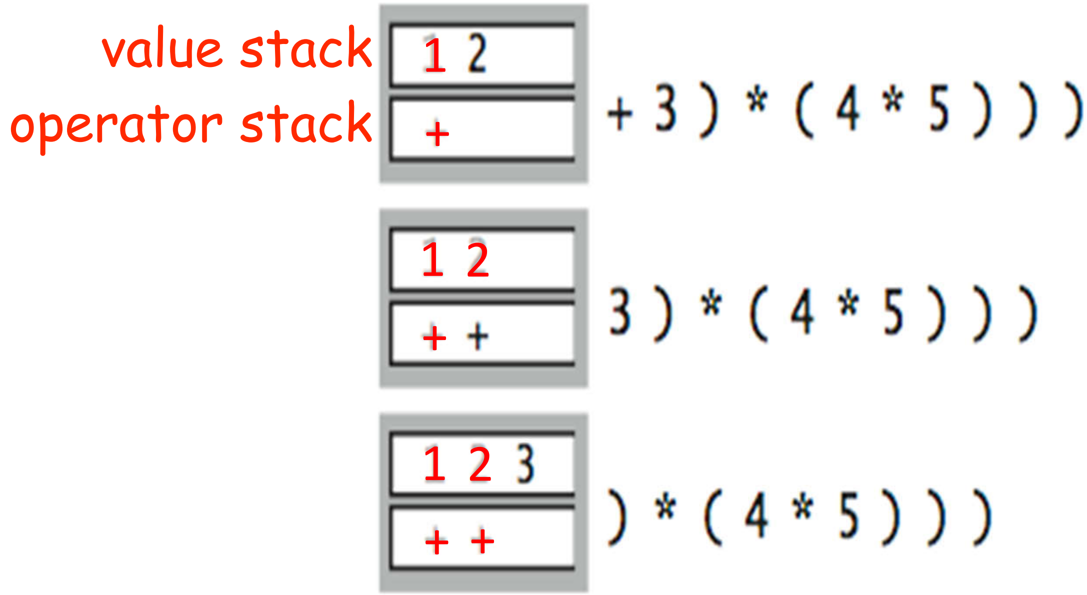
Arithmetic expression evaluation



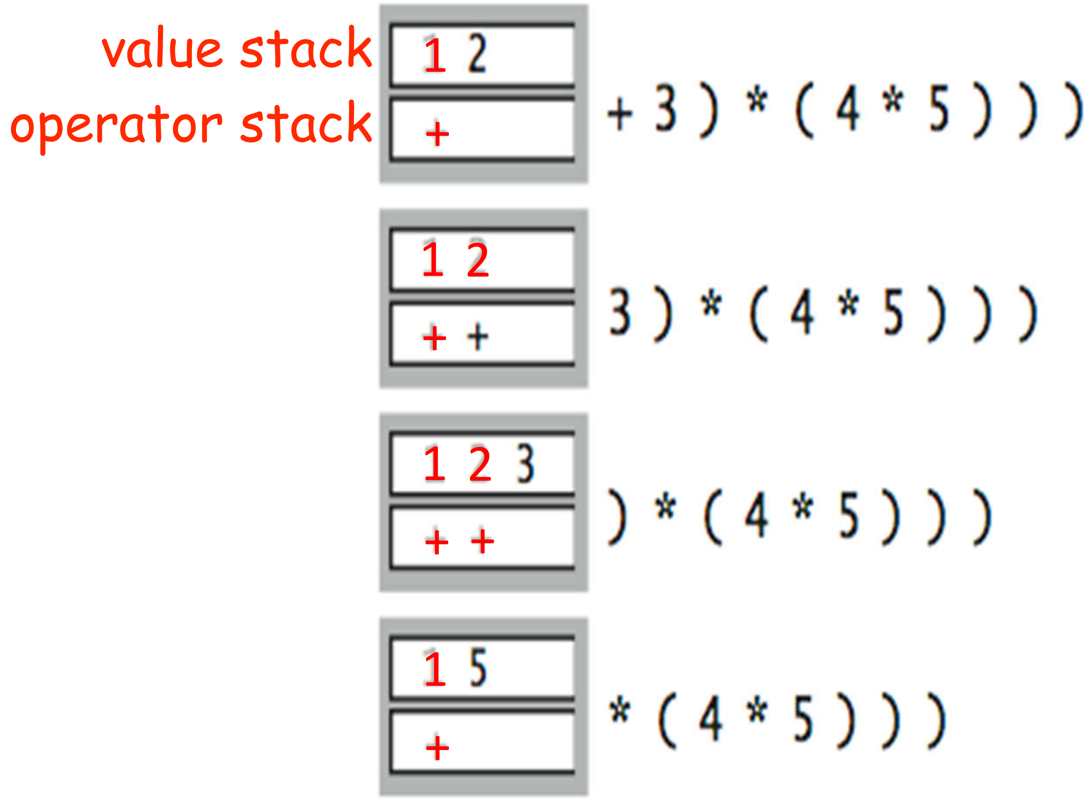
Arithmetic expression evaluation



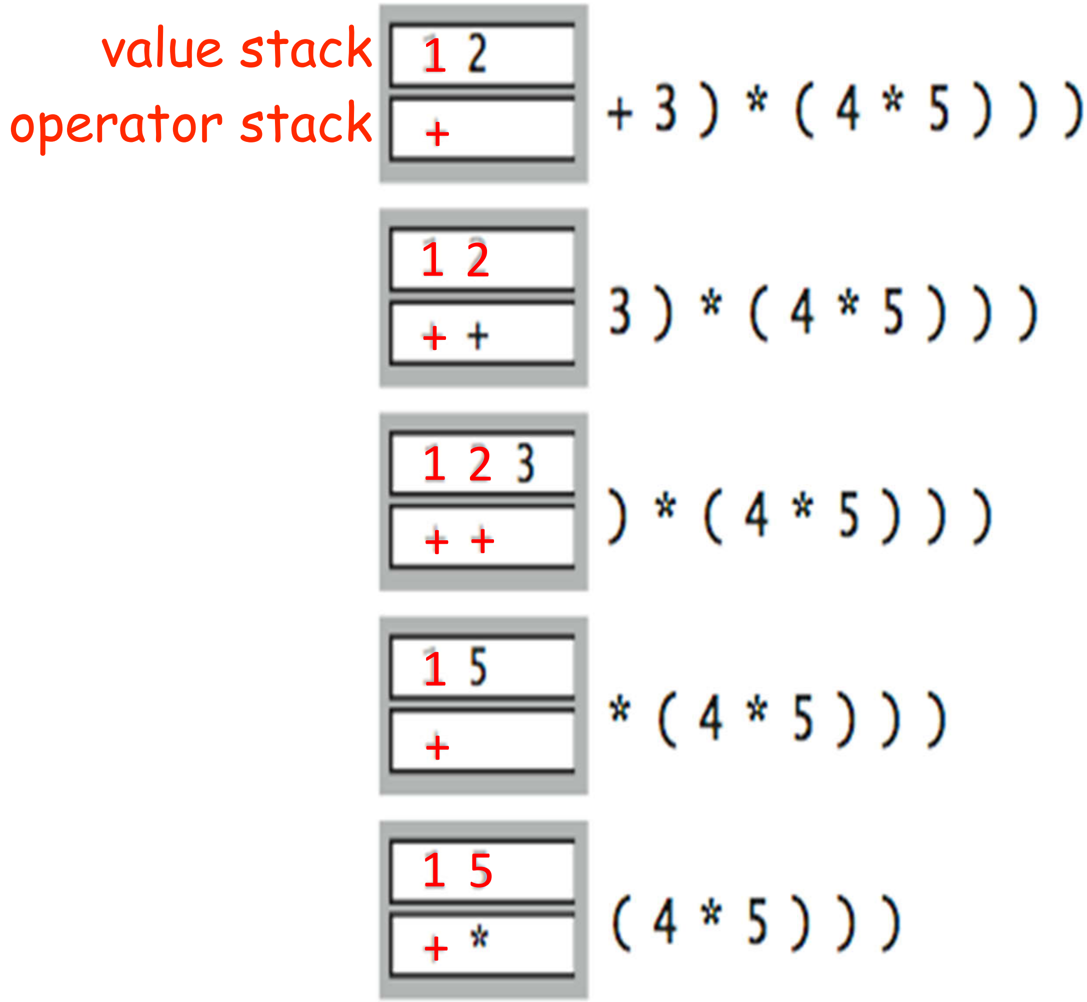
Arithmetic expression evaluation



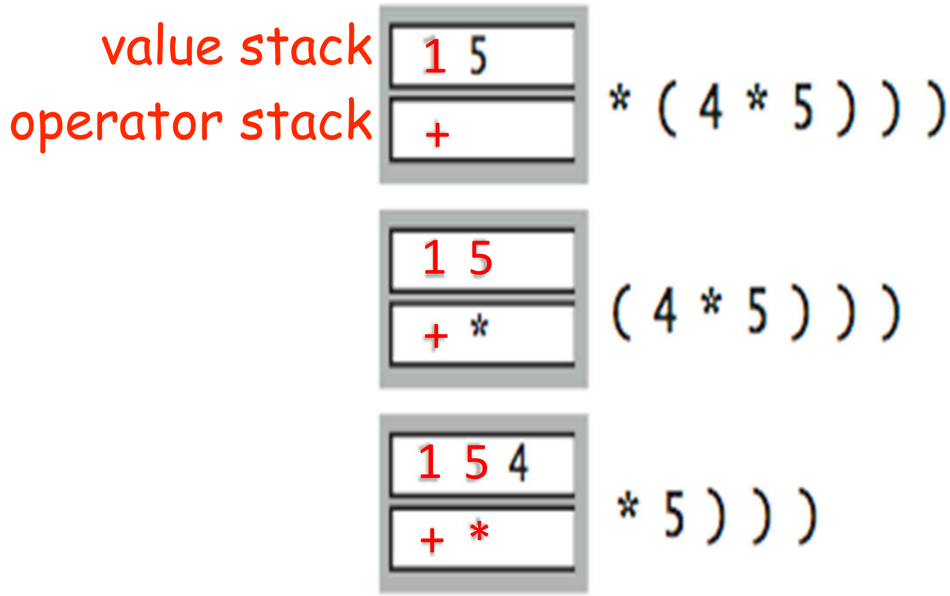
Arithmetic expression evaluation



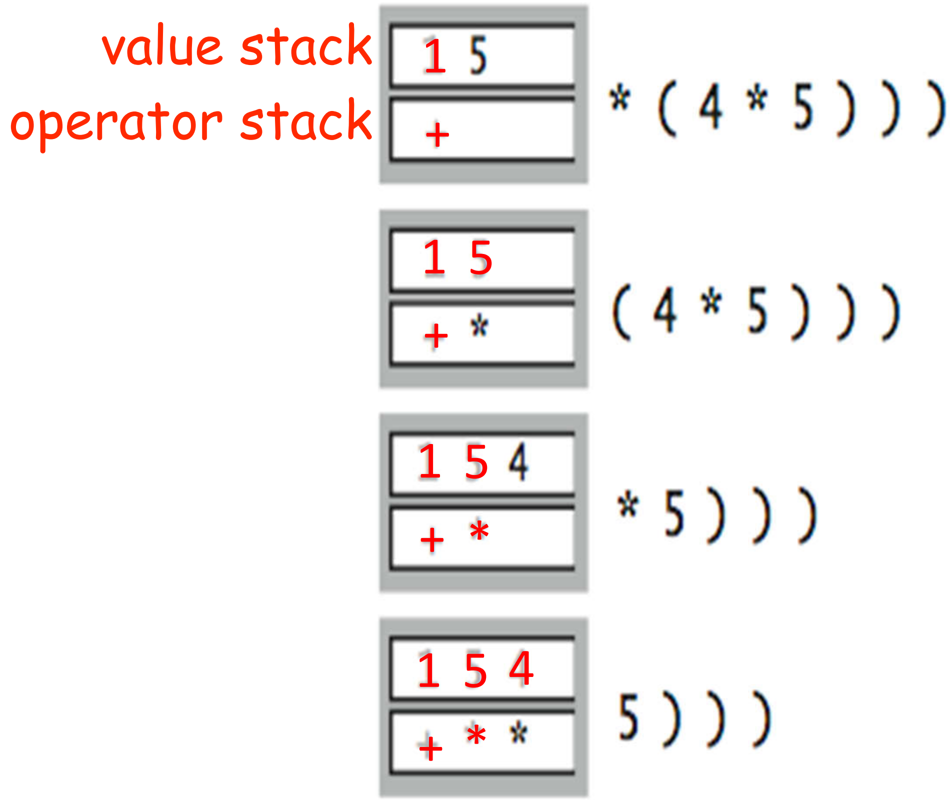
Arithmetic expression evaluation



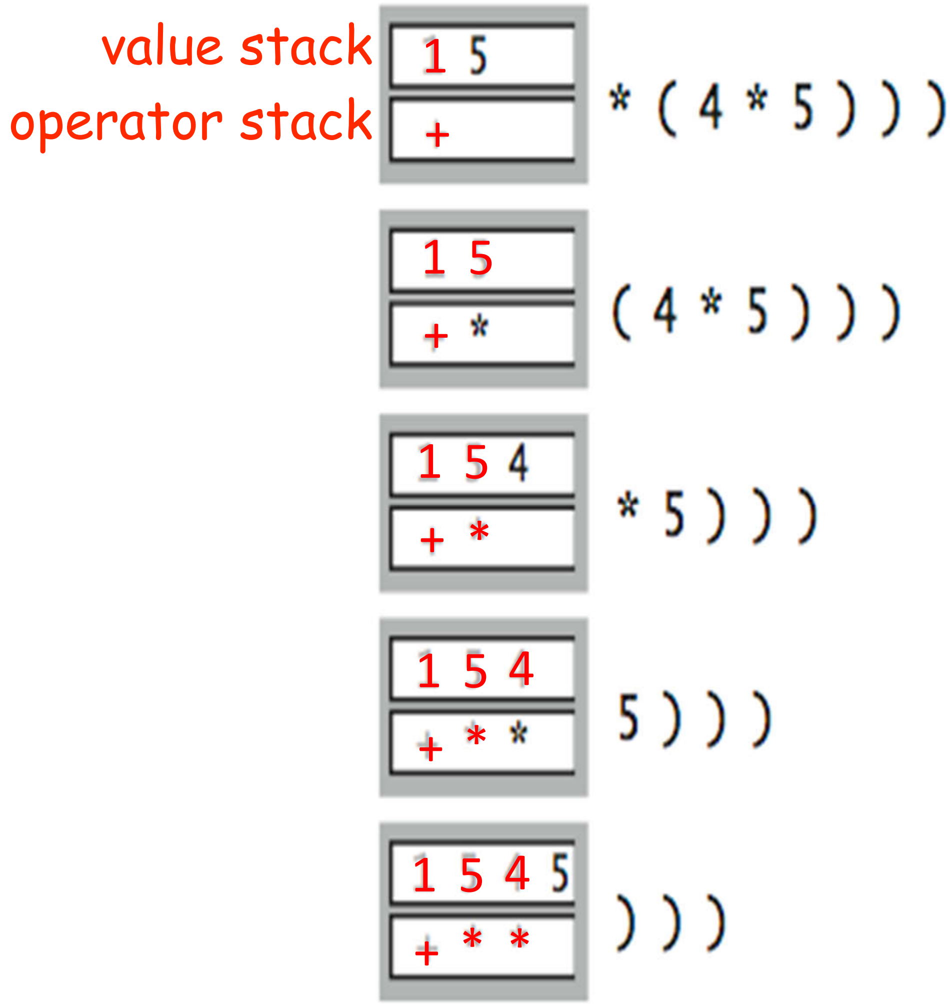
Arithmetic expression evaluation



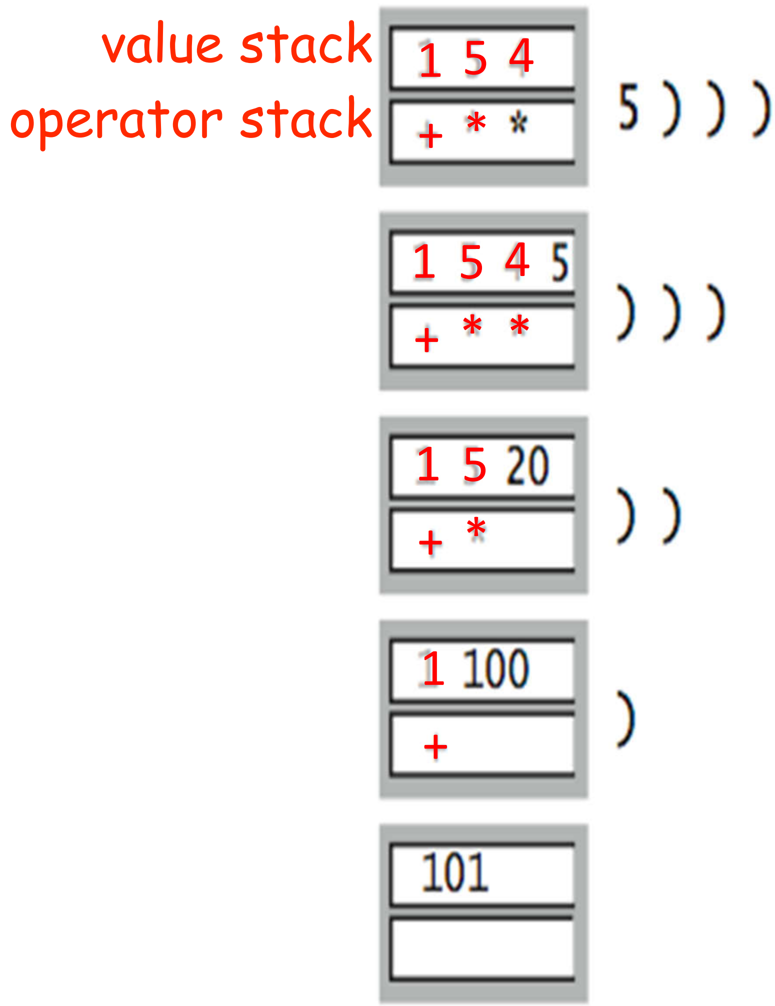
Arithmetic expression evaluation



Arithmetic expression evaluation



Arithmetic expression evaluation



Arithmetic expression evaluation

the 2-stack algorithm computes the same value if the operator occurs
after the two values

(1 + ((2 + 3) * (4 * 5)))

Infix expression

(1 ((2 3 +) (4 5 *) *) +)

Operators after 2 operands

Arithmetic expression evaluation

the 2-stack algorithm computes the same value if the operator occurs after the two values

(1 + ((2 + 3) * (4 * 5)))

Infix expression

(1 ((2 3 +) (4 5 *) *) +)

Operators after 2 operands

We can remove the parentheses:

1 2 3 + 4 5 * * +

Postfix or Reverse Polish

Arithmetic expression evaluation

We need only a single stack to evaluate:

```
1 2 3 + 4 5 * * +
```

Postfix or
Reverse Polish

Reverse-Polish Notation

Another examples:

$$3 \quad 4 \quad 5 \quad \times \quad + \quad 6 \quad -$$
$$3 \quad 20 \quad + \quad 6 \quad -$$
$$23 \quad 6 \quad -$$
$$17$$

$3 + 4 \times 5 - 6 = 17$

$$3 \quad 4 \quad 5 \quad 6 \quad - \quad \times \quad +$$
$$3 \quad 4 \quad -1 \quad \times \quad +$$
$$3 \quad -4 \quad +$$
$$-1$$

$3 + 4 \times (5 - 6) = -1$

Reverse-Polish Notation

Benefits:

- No ambiguity and no brackets are required
- Reverse-Polish can be processed using stacks

Reverse-Polish Notation

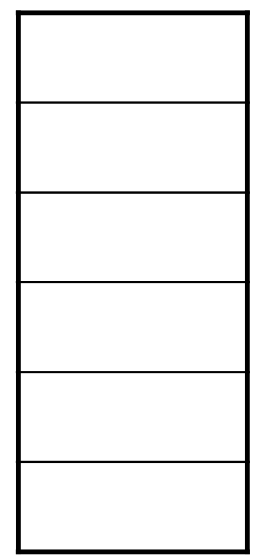
The easiest way to parse reverse-Polish notation is to use an operand stack:

- operands are processed by pushing them onto the stack
- when processing an operator:
  - pop the last two items off the operand stack,
  - perform the operation, and
  - push the result back onto the stack

Reverse-Polish Notation

Evaluate the following reverse-Polish expression using a stack:

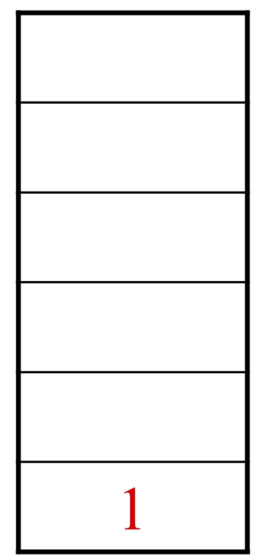
$1 \ 2 \ 3 \ + \ 4 \ 5 \ 6 \ \times \ - \ 7 \ \times \ + \ - \ 8 \ 9 \ \times \ +$



Reverse-Polish Notation

Push 1 onto the stack

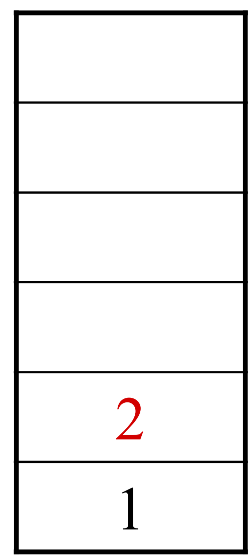
$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$



Reverse-Polish Notation

Push 1 onto the stack

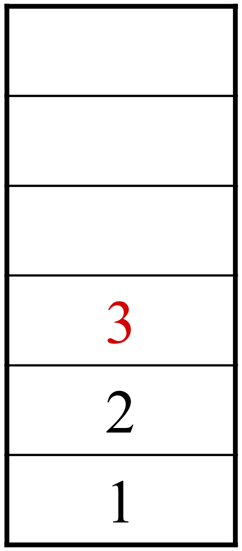
$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$



Reverse-Polish Notation

Push 3 onto the stack

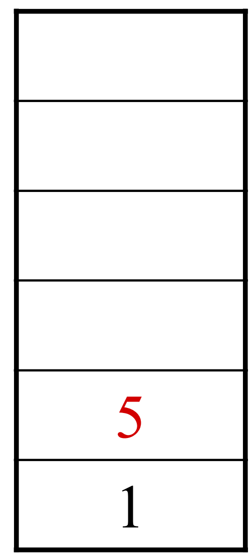
$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$



Reverse-Polish Notation

Pop 3 and 2 and push $2 + 3 = 5$

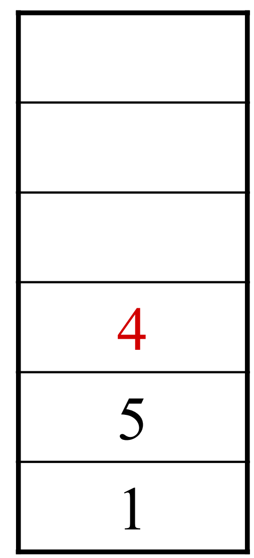
$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$



Reverse-Polish Notation

Push 4 onto the stack

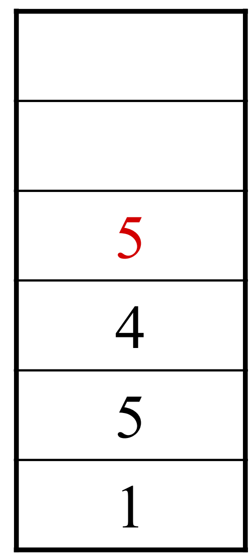
$$1\ \ 2\ \ 3\ \ +\ \ 4\ \ 5\ \ 6\ \ \times\ \ -\ \ 7\ \ \times\ \ +\ \ -\ \ 8\ \ 9\ \ \times\ \ +$$



Reverse-Polish Notation

Push 5 onto the stack

$$1 \quad 2 \quad 3 \quad + \quad 4 \quad {\color{red}5} \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$



Reverse-Polish Notation

Push 6 onto the stack

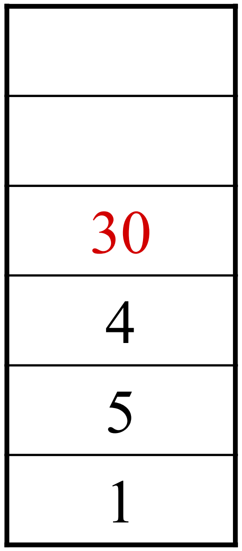
$$1 \;\; 2 \;\; 3 \;\; + \;\; 4 \;\; 5 \;\; 6 \;\; \times \;\; - \;\; 7 \;\; \times \;\; + \;\; - \;\; 8 \;\; 9 \;\; \times \;\; +$$

6
5
4
5
1

Reverse-Polish Notation

Pop 6 and 5 and push $5 \times 6 = 30$

$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$



Reverse-Polish Notation

Pop 30 and 4 and push $4 - 30 = -26$

$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$

−26
5
1

Reverse-Polish Notation

Push 7 onto the stack

$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$

7
–26
5
1

Reverse-Polish Notation

Pop 7 and -26 and push $-26 \times 7 = -182$

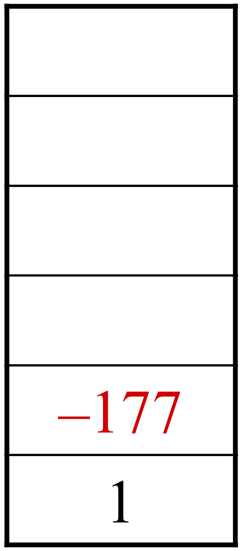
$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$

-182
5
1

Reverse-Polish Notation

Pop -182 and 5 and push $-182 + 5 = -177$

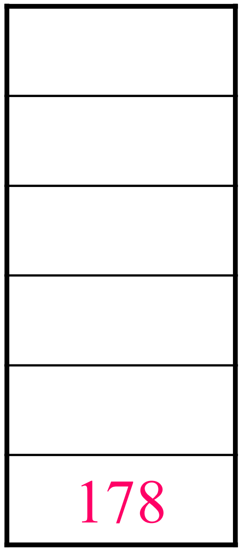
$$1\ \ 2\ \ 3\ \ +\ \ 4\ \ 5\ \ 6\ \ \times\ \ -\ \ 7\ \ \times\ \ +\ \ -\ \ 8\ \ 9\ \ \times\ \ +$$



Reverse-Polish Notation

Pop -177 and 1 and push $1 - (-177) = 178$

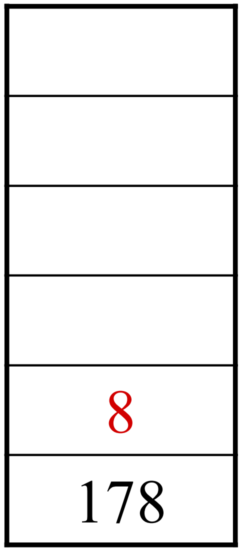
$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$



Reverse-Polish Notation

Push 8 onto the stack

$$1\quad 2\quad 3\quad +\quad 4\quad 5\quad 6\quad \times\quad -\quad 7\quad \times\quad +\quad -\quad {\color{red}8}\quad 9\quad \times\quad +$$



Reverse-Polish Notation

Push 1 onto the stack

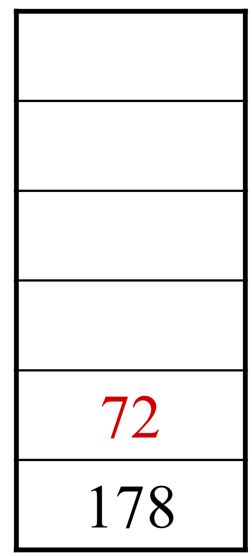
$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad \color{red}{9} \quad \times \quad +$$

9
8
178

Reverse-Polish Notation

Pop 9 and 8 and push $8 \times 9 = 72$

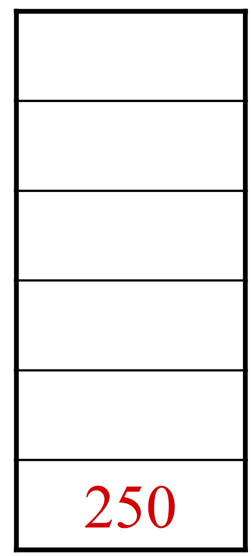
$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$



Reverse-Polish Notation

Pop 72 and 178 and push $178 + 72 = 250$

$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$



Reverse-Polish Notation

Thus

$$1 \;\; 2 \;\; 3 \;\; + \;\; 4 \;\; 5 \;\; 6 \;\; \times \;\; - \;\; 7 \;\; \times \;\; + \;\; - \;\; 8 \;\; 9 \;\; \times \;\; +$$

evaluates to the value on the top: 250

The equivalent in-fix notation is

$$((1 - ((2 + 3) + ((4 - (5 \times 6)) \times 7))) + (8 \times 9))$$