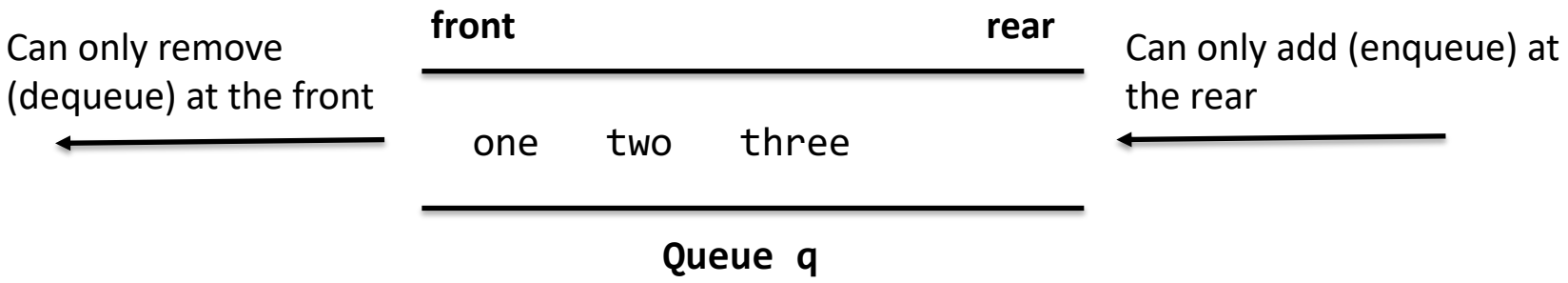
AUBURN
UNIVERSITY

SAMUEL GINN
COLLEGE OF ENGINEERING

Stacks and Queues

Queue

A **queue** is a collection that maintains its elements in first-in, first-out (FIFO) order.




```
q.enqueue("one");
q.enqueue("two");
q.enqueue("three");
```

```
q.dequeue();
```

Physical analogy: A waiting line.

Queue

```
public interface Queue<T> extends Iterable<T> {

    /** Add one element to the rear of this queue. */
    void enqueue(T element);

    /** Remove and return the front element of this queue. */
    T deqeue();

    /** Return without removing the front element of this queue. */
    T first();

    /** Returns true if this queue contains no elements. */
    boolean isEmpty();

    /** Returns the number of elements in this queue. */
    int size();

}
```

Queue

```
queue.enqueue(1)
```

```
1
```

```
queue.enqueue(2)
```

```
1    2
```

```
queue.enqueue(3)
```

```
1    2    3
```

```
queue.dequeue()
```

```
2    3
```

```
queue.enqueue(4)
```

```
2    3    4
```

```
queue.enqueue(queue.dequeue())
```

```
3    4    2
```

```
queue.enqueue(5)
```

```
3    4    2    5
```

```
queue.dequeue()
```

```
4    2    5
```

```
queue.dequeue()
```

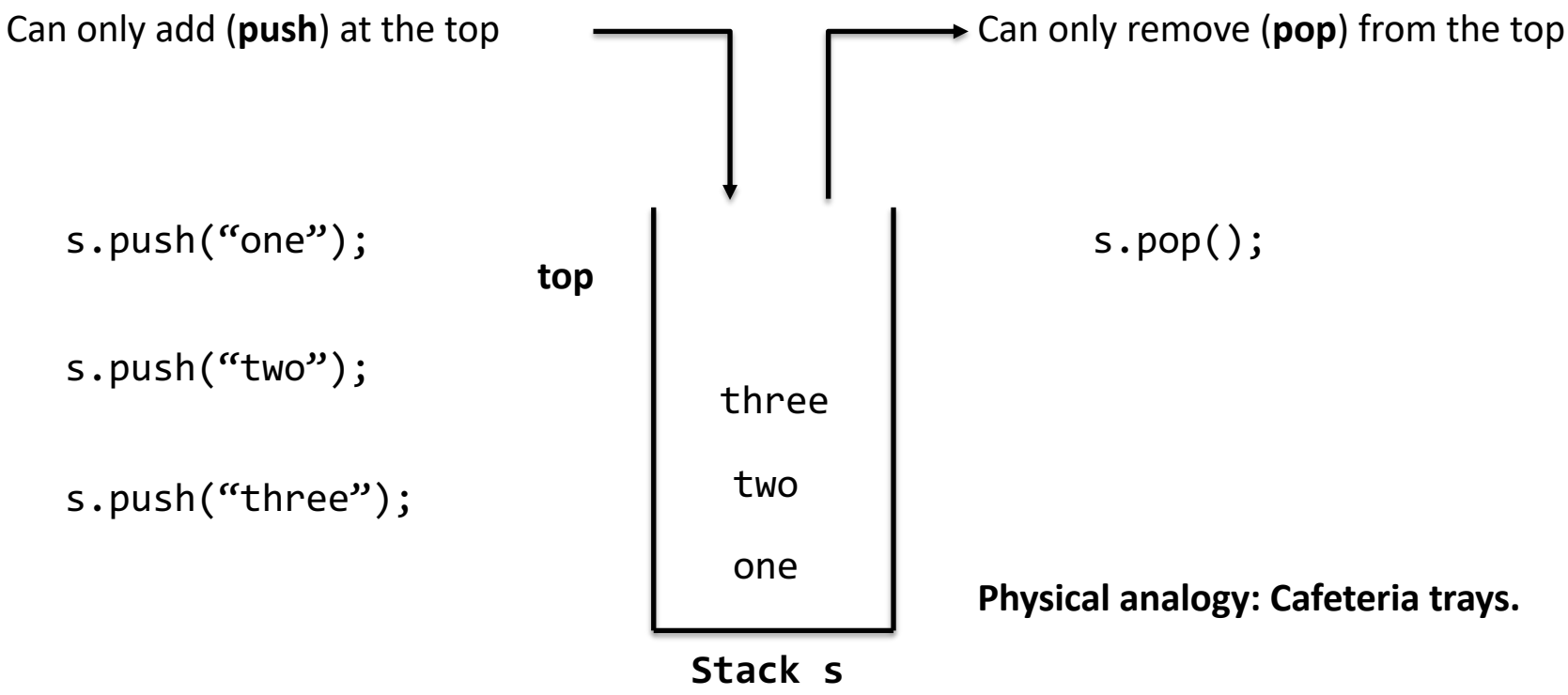
```
2    5
```

```
queue.dequeue()
```

```
5
```

Stack

A **stack** is a collection that maintains its elements in last-in, first-out (LIFO) order.



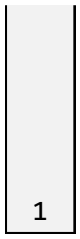
Physical analogy: Cafeteria trays.

Stack

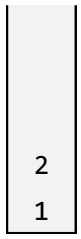
```
public interface Stack<T> extends Iterable<T> {

    /** Add one element to the top of this stack. */
    void push(T element);

    /** Remove and return the top element of this stack. */
    T pop();

    /** Return without removing the top element of this stack. */
    T peek();

    /** Returns true if this stack contains no elements. */
    boolean isEmpty();

    /** Returns the number of elements in this stack. */
    int size();

}
```

Stack

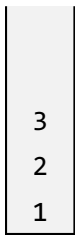
`stack.push(1)`



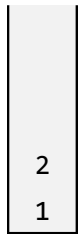
`stack.push(2)`



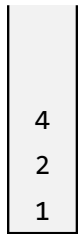
`stack.push(3)`



`stack.pop()`

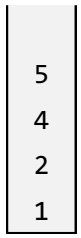


`stack.push(4)`

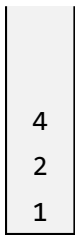


`stack.push(stack.pop())`

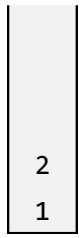
`stack.push(5)`



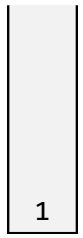
`stack.pop()`



`stack.pop()`



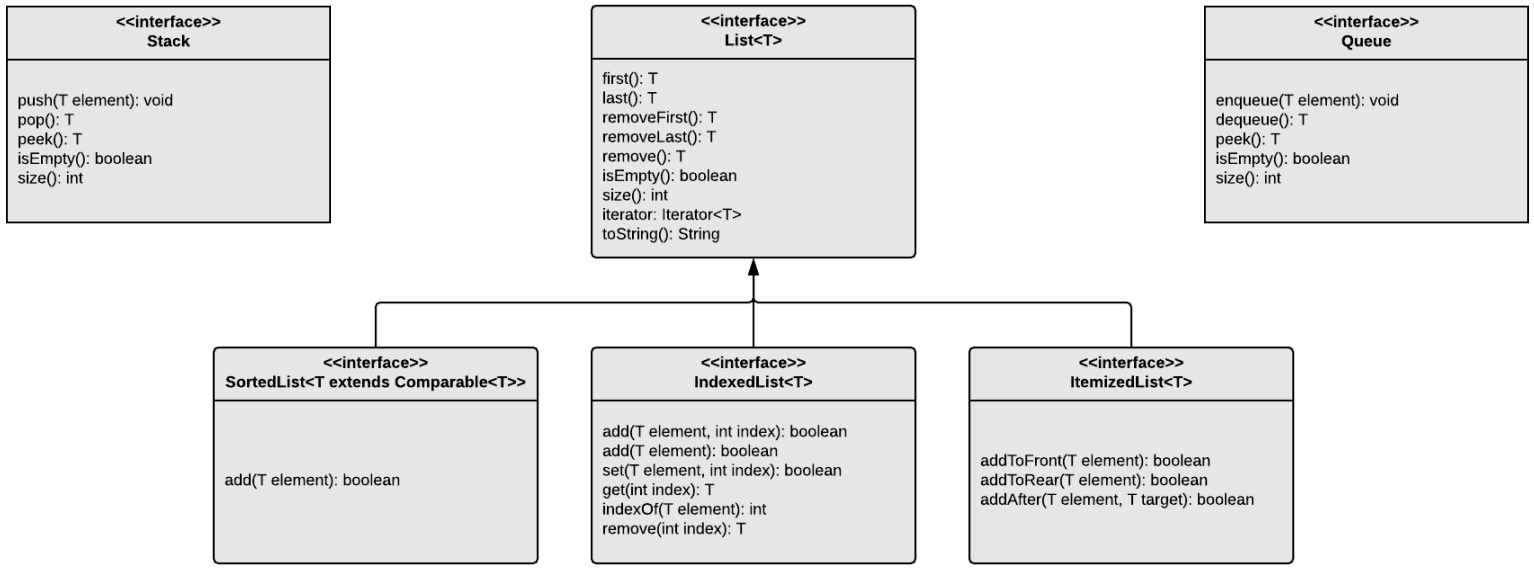
`stack.pop()`



Implementation alternatives

Stacks and queues as lists

Would it make sense to define a stack and a queue as part of the List interface hierarchy?



Stacks and queues as lists

Would it make sense to implement stacks and queues as subclasses of a list class?

```
public class ArrayStack<T> extends ArrayItemizedList<T> implements Stack<T> {









}
```

Stacks and queues as lists

Would it make sense to implement stacks and queues as subclasses of a list class?

```
public class ArrayStack<T> extends ArrayItemizedList<T> implements Stack<T> {

    public void push(T element) {

    }



}
```

Stacks and queues as lists

Would it make sense to implement stacks and queues as subclasses of a list class?

```
public class ArrayStack<T> extends ArrayItemizedList<T> implements Stack<T> {

    public void push(T element) {
        this.addToRear(element);
    }




}
```

Stacks and queues as lists

Would it make sense to implement stacks and queues as subclasses of a list class?

```
public class ArrayStack<T> extends ArrayItemizedList<T> implements Stack<T> {

    public void push(T element) {
        this.addToRear(element);
    }

    public void pop() {

    }

}
```

Stacks and queues as lists

Would it make sense to implement stacks and queues as subclasses of a list class?

```
public class ArrayStack<T> extends ArrayItemizedList<T> implements Stack<T> {

    public void push(T element) {
        this.addToRear(element);
    }

    public void pop() {
        this.removeLast();
    }

}
```

Our implementation inherits too much!

Improper use of inheritance

Mutant Marsupials Take Up Arms Against Australian Air Force

The reuse of some object-oriented code has caused tactical headaches for Australia's armed forces. As virtual reality simulators assume larger roles in helicopter combat training, programmers have gone to great lengths to increase the realism of their scenarios, including detailed landscapes and - in the case of the Northern Territory's Operation Phoenix- herds of kangaroos (since disturbed animals might well give away a helicopter's position).

The head of the Defense Science & Technology Organization's Land Operations/Simulation division reportedly instructed developers to model the local marsupials' movements and reactions to helicopters. Being efficient programmers, they just re-appropriated some code originally used to model infantry detachment reactions under the same stimuli, changed the mapped icon from a soldier to a kangaroo, and increased the figures' speed of movement.

Eager to demonstrate their flying skills for some visiting American pilots, the hotshot Aussies "buzzed" the virtual kangaroos in low flight during a simulation. The kangaroos scattered, as predicted, and the visiting Americans nodded appreciatively... then did a double-take as the kangaroos reappeared from behind a hill and launched a barrage of Stinger missiles at the hapless helicopter. (Apparently the programmers had forgotten to remove that part of the infantry coding.)

The lesson? Objects are defined with certain attributes, and any new object defined in terms of an old one inherits all the attributes. The embarrassed programmers had learned to be careful when reusing object-oriented code, and the Yanks left with a newfound respect for Australian wildlife.

Simulator supervisors report that pilots from that point onward have strictly avoided kangaroos, just as they were meant to.

• From June 15, 1999 Defense Science and Technology Organization Lecture Series, Melbourne, Australia, and staff reports

Stacks and queues as lists

It would make better sense to implement stacks and queues with list instances as fields.

```
public class ArrayStack<T> implements Stack<T> {









}
```

Stacks and queues as lists

It would make better sense to implement stacks and queues with list instances as fields.

```
public class ArrayStack<T> implements Stack<T> {

    private ArrayItemizedList<T> stack;









}
```

Stacks and queues as lists

It would make better sense to implement stacks and queues with list instances as fields.

```
public class ArrayStack<T> implements Stack<T> {

    private ArrayItemizedList<T> stack;

    public void push(T element) {

    }



}
```

Stacks and queues as lists

It would make better sense to implement stacks and queues with list instances as fields.

```
public class ArrayStack<T> implements Stack<T> {

    private ArrayItemizedList<T> stack;

    public void push(T element) {
        stack.addToRear(element);
    }




}
```

Stacks and queues as lists

It would make better sense to implement stacks and queues with list instances as fields.

```
public class ArrayStack<T> implements Stack<T> {

    private ArrayItemizedList<T> stack;

    public void push(T element) {
        stack.addToRear(element);
    }

    public void pop() {

    }

}
```

Stacks and queues as lists

It would make better sense to implement stacks and queues with list instances as fields.

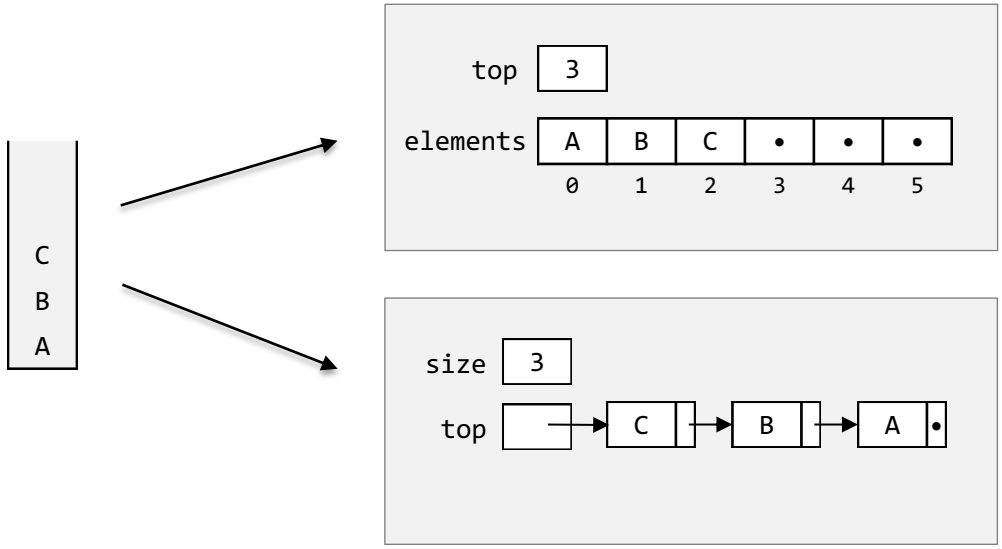
```
public class ArrayStack<T> implements Stack<T> {

    private ArrayItemizedList<T> stack;

    public void push(T element) {
        stack.addToRear(element);
    }

    public void pop() {
        stack.removeLast();
    }

}
```

Composition is often a better choice for reuse than inheritance.

Stacks and queues from scratch

We could create our own implementation of stacks and queues not based on an existing list or other collection implementation.

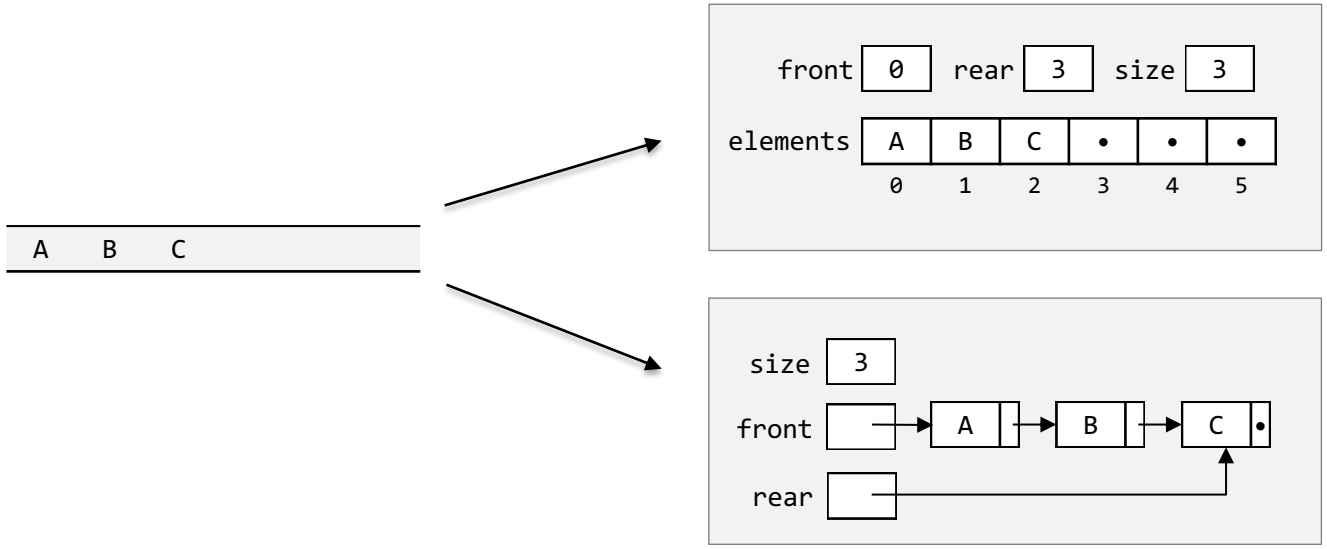
As always, we have two basic data structure choices: arrays or linked nodes.



Stacks and queues from scratch

We could create our own implementation of stacks and queues not based on an existing list or other collection implementation.

As always, we have two basic data structure choices: arrays or linked nodes.

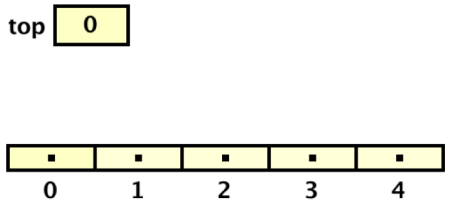


Array-based stack

ArrayStack

```
public class ArrayStack<T> implements Stack<T> {

    // holds the values in the stack
    private T[] elements;

    // index of the next insertion point
    private int top;

    @SuppressWarnings("unchecked")
    public ArrayStack(int capacity) {
          elements = (T[]) new Object[capacity];
          top = 0;
    }
```

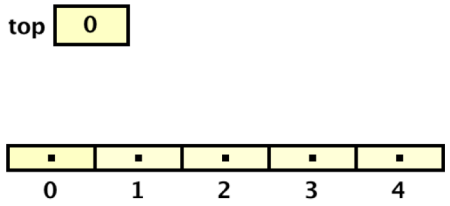
```
Stack stack = new ArrayStack(5);
```



ArrayStack

```
public class ArrayStack<T> implements Stack<T> {

    // holds the values in the stack
    private T[] elements;

    // index of the next insertion point
    private int top;

    @SuppressWarnings("unchecked")
    public ArrayStack(int capacity) {
          elements = (T[]) new Object[capacity];
          top = 0;
    }

    public int size() {
        return top;
    }
```

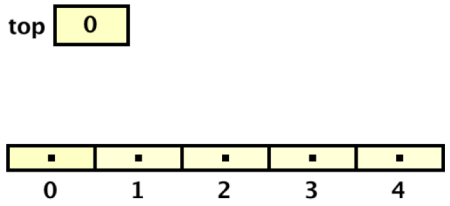
```
Stack stack = new ArrayStack(5);
```



ArrayStack

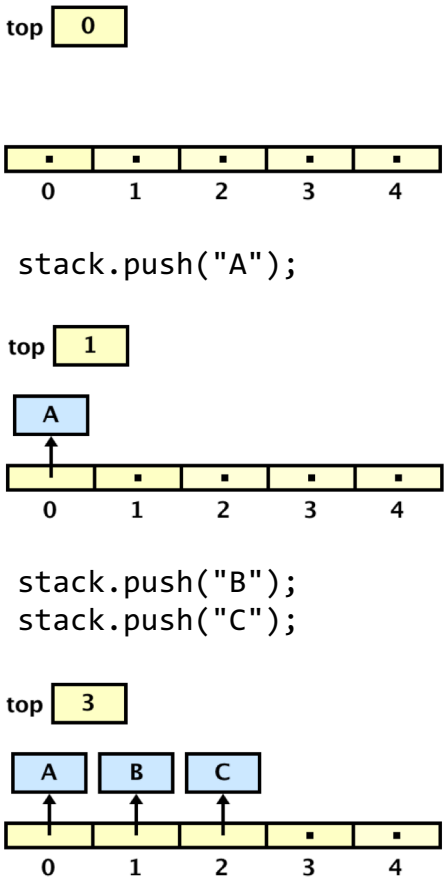
```
public class ArrayStack<T> implements Stack<T> {

    // holds the values in the stack
    private T[] elements;

    // index of the next insertion point
    private int top;

    @SuppressWarnings("unchecked")
    public ArrayStack(int capacity) {
          elements = (T[]) new Object[capacity];
          top = 0;
    }

    public int size() {
        return top;
    }

    public boolean isEmpty() {
        return size() == 0;
    }
```

```
Stack stack = new ArrayStack(5);
```



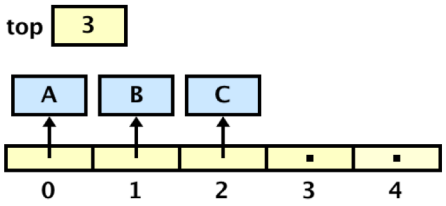
ArrayStack

```
public class ArrayStack<T> implements Stack<T> {

    public void push(T element) {
        if (size() == elements.length) {
            resize(elements.length * 2);
        }
        elements[top++] = element;
    }
```

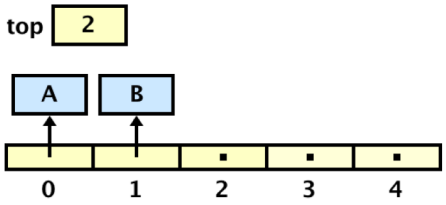


ArrayStack

```
public class ArrayStack<T> implements Stack<T> {

    public void push(T element) {
        if (size() == elements.length) {
            resize(elements.length * 2);
        }
        elements[top++] = element;
    }

    public T pop() {
        if (isEmpty()) {
            return null;
        }
        T topValue = elements[--top];
        elements[top] = null;
        return topValue;
    }
```

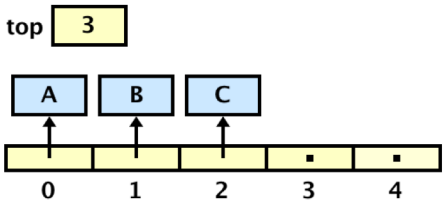



```
stack.pop();
```

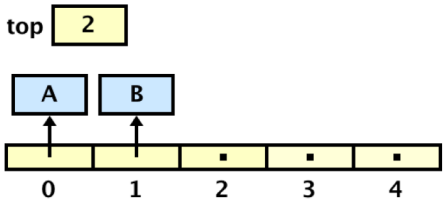


ArrayStack

```
public class ArrayStack<T> implements Stack<T> {

    public void push(T element) {
        if (size() == elements.length) {
            resize(elements.length * 2);
        }
        elements[top++] = element;
    }

    public T pop() {
        if (isEmpty()) {
            return null;
        }
        T topValue = elements[--top];
        elements[top] = null;
        return topValue;
    }

    public T peek() {
        if (isEmpty()) {
            return null;
        }
        return elements[top - 1];
    }
```




```
stack.pop();
```



ArrayStack

```
public class ArrayStack<T> implements Stack<T> {

    public void push(T element) {
        if (size() == elements.length) {
            resize(elements.length * 2);
        }
        elements[top++] = element;
    }

    public T pop() {
        if (isEmpty()) {
            return null;
        }
        T topValue = elements[--top];
        elements[top] = null;
        return topValue;
    }

    public T peek() {
        if (isEmpty()) {
            return null;
        }
        return elements[top - 1];
    }
```

O(1) *amortized*

O(1)

O(1)

Time complexity:

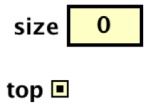
O(1) for all three methods

Node-based stack

LinkedStack

```
public class LinkedStack<T> implements Stack<T> {
    // reference to the top of the stack
    private Node top;

    // number of values in the stack
    private int size;

    /** Create an instance of the LinkedStack class. */
    public LinkedStack() {
        top = null;
        size = 0;
    }
```

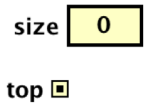
```
Stack stack = new LinkedStack();
```



LinkedStack

```
public class LinkedStack<T> implements Stack<T> {
    // reference to the top of the stack
    private Node top;

    // number of values in the stack
    private int size;

    /** Create an instance of the LinkedStack class. */
    public LinkedStack() {
        top = null;
        size = 0;
    }

    public int size() {
        return size;
    }
```

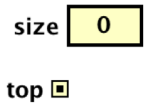
```
Stack stack = new LinkedStack();
```



LinkedStack

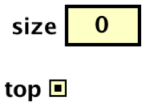
```
public class LinkedStack<T> implements Stack<T> {
    // reference to the top of the stack
    private Node top;

    // number of values in the stack
    private int size;

    /** Create an instance of the LinkedStack class. */
    public LinkedStack() {
        top = null;
        size = 0;
    }

    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return size() == 0;
    }
```

```
Stack stack = new LinkedStack();
```

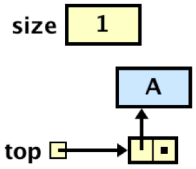


LinkedStack

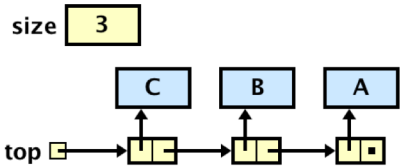
```
public class LinkedStack<T> implements Stack<T> {

    public void push(T element) {
        top = new Node(element, top);
        size++;
    }
```




```
stack.push("A");
```

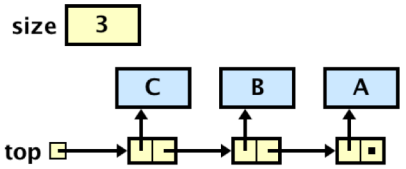



```
stack.push("B");
stack.push("C");
```

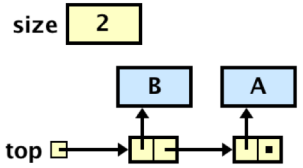


LinkedStack

```
public class LinkedStack<T> implements Stack<T> {

    public void push(T element) {
        top = new Node(element, top);
        size++;
    }

    public T pop() {
        if (isEmpty()) {
            return null;
        }
        T topValue = top.element;
        top = top.next;
        size--;
        return topValue;
    }
```

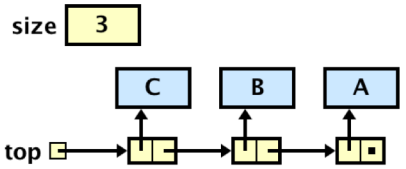



```
stack.pop();
```

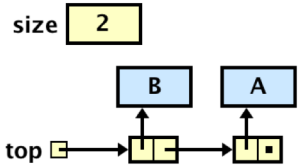


LinkedStack

```
public class LinkedStack<T> implements Stack<T> {

    public void push(T element) {
        top = new Node(element, top);
        size++;
    }

    public T pop() {
        if (isEmpty()) {
            return null;
        }
        T topValue = top.element;
        top = top.next;
        size--;
        return topValue;
    }

    public T peek() {
        if (isEmpty()) {
            return null;
        }
        return top.element;
    }
```




```
stack.pop();
```



LinkedStack

```
public class LinkedStack<T> implements Stack<T> {

    public void push(T element) {
        top = new Node(element, top);
        size++;
    }

    public T pop() {
        if (isEmpty()) {
            return null;
        }
        T topValue = top.element;
        top = top.next;
        size--;
        return topValue;
    }

    public T peek() {
        if (isEmpty()) {
            return null;
        }
        return top.element;
    }
```

O(1)

O(1)

O(1)

Time complexity:

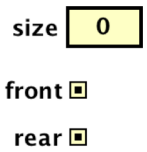
O(1) for all three methods

Node-based queue

LinkedQueue

```
public class LinkedQueue<T> implements Queue<T> {
    // the front node
    private Node front;

    // the rear node;
    private Node rear;

    // the number of elements in the queue
    private int size;

    /** Create an instance of the LinkedQueue class. */
    public LinkedQueue() {
        front = null;
        rear = null;
        size = 0;
    }
```

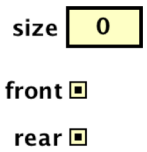
```
Queue queue = new LinkedQueue();
```



LinkedQueue

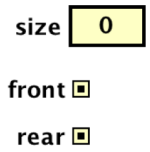
```
public class LinkedQueue<T> implements Queue<T> {
    // the front node
    private Node front;

    // the rear node;
    private Node rear;

    // the number of elements in the queue
    private int size;

    /** Create an instance of the LinkedQueue class. */
    public LinkedQueue() {
        front = null;
        rear = null;
        size = 0;
    }

    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return size() == 0;
    }
```

```
Queue queue = new LinkedQueue();
```

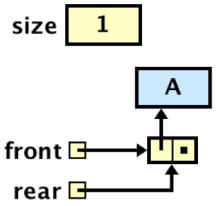


LinkedQueue

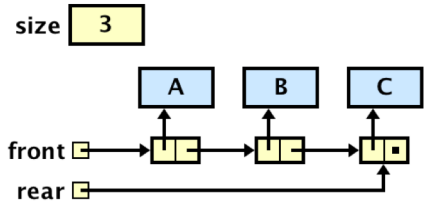
```
public class LinkedQueue<T> implements Queue<T> {

    public void enqueue(T element) {
        Node n = new Node(element, null);
        if (isEmpty()) {
            rear = n;
            front = rear;
        } else {
            rear.next = n;
            rear = n;
        }
        size++;
    }
```




```
queue.enqueue("A");
```

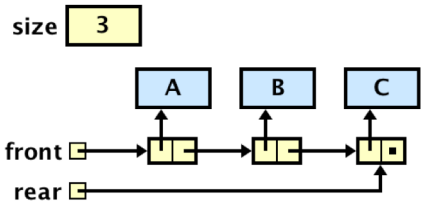



```
queue.enqueue("B");
queue.enqueue("C");
```

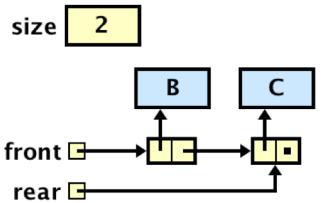


LinkedQueue

```
public class LinkedQueue<T> implements Queue<T> {

    public T dequeue() {
        if (isEmpty()) {
            return null;
        }
        T result = front.element;
        front = front.next;
        if (size == 1) {
            rear = front;
        }
        size--;
        return result;
    }
```

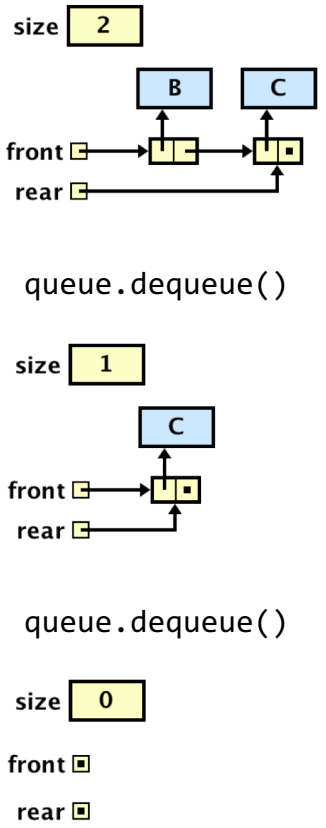


queue.dequeue()



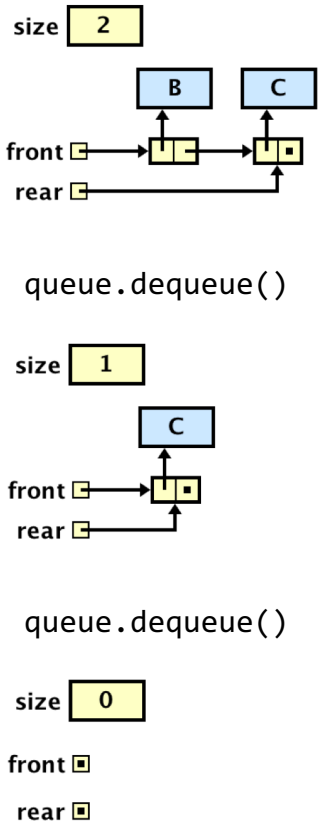
LinkedQueue

```
public class LinkedQueue<T> implements Queue<T> {

    public T dequeue() {
        if (isEmpty()) {
            return null;
        }
        T result = front.element;
        front = front.next;
        if (size == 1) {
            rear = front;
        }
        size--;
        return result;
    }
```



LinkedQueue

```
public class LinkedQueue<T> implements Queue<T> {

    public T dequeue() {
        if (isEmpty()) {
            return null;
        }
        T result = front.element;
        front = front.next;
        if (size == 1) {
            rear = front;
        }
        size--;
        return result;
    }

    public T first() {
        if (isEmpty()) {
            return null;
        }
        return front.element;
    }
```



LinkedQueue

```
public class LinkedQueue<T> implements Queue<T> {

    public void enqueue(T element) {
        Node n = new Node(element, null);
        if (isEmpty()) {
            rear = n;
            front = rear;
        } else {
            rear.next = n;
            rear = n;
        }
        size++;
    }
```

O(1)

Time complexity:

LinkedQueue

```
public class LinkedQueue<T> implements Queue<T> {

    public T dequeue() {
        if (isEmpty()) {
            return null;
        }
        T result = front.element;
        front = front.next;
        if (size == 1) {
            rear = front;
        }
        size--;
        return result;
    }

    public T first() {
        if (isEmpty()) {
            return null;
        }
        return front.element;
    }
```

O(1)

O(1)

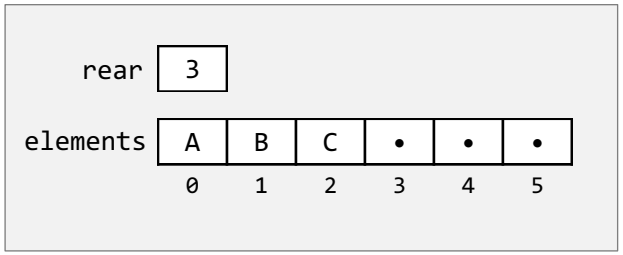
Time complexity:

O(1) for all three methods

Array-based queue

ArrayQueue

Our familiar array-management strategy of “left justified with no gaps” that has worked for the Bag, Set, List, and Stack collections no longer works well for the Queue collection.



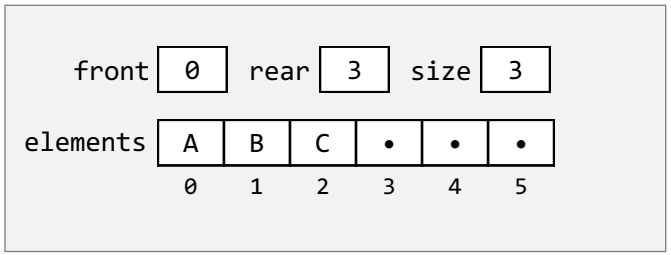
By anchoring one end of the queue at index 0, shifting would be required in either enqueue or dequeue.

So, the time complexity profile of our implementation would be one of the following:

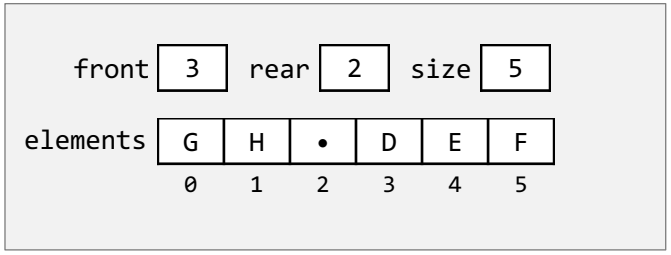
- enqueue O(1), dequeue O(N)
- enqueue O(N), dequeue O(1)

ArrayQueue

We need to use a “circular array” strategy to make both enqueue and dequeue O(1).



This is called a circular queue or circular array strategy because we keep the elements of the array contiguous, but allow them to float down toward the right and then wrap around once they hit the end.

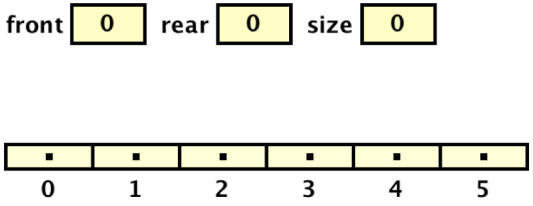


	Array	FIFO order
	• • • • • •	[]
enqueue(“A”)	A • • • • •	[A]
enqueue(“B”)	A B • • • •	[A,B]
enqueue(“C”)	A B C • • •	[A,B,C]
enqueue(“D”)	A B C D • •	[A,B,C,D]
dequeue()	• B C D • •	[B,C,D]
dequeue()	• • C D • •	[C,D]
dequeue()	• • • D • •	[D]
enqueue(“E”)	• • • D E •	[D,E]
enqueue(“F”)	• • • D E F	[D,E,F]
enqueue(“G”)	G • • D E F	[D,E,F,G]
enqueue(“H”)	G H • D E F	[D,E,F,G,H]

ArrayQueue

```
public class ArrayQueue<T> implements Queue<T> {
    // stores the elements in the queue
    private T[] elements;

    // index of the front element
    private int front;

    // index of the next insertion point
    private int rear;

    // number of elements in the queue
    private int size;

    /** Creates an instance of an ArrayQueue. */
    @SuppressWarnings("unchecked")
    public ArrayQueue(int capacity) {
        elements = (T[]) new Object[capacity];
        front = 0;
        rear = 0;
        size = 0;
    }
```

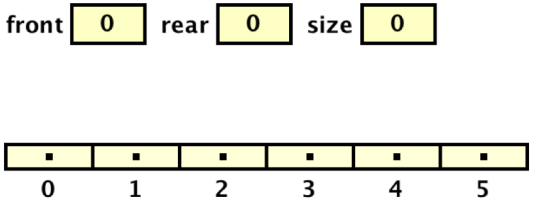
```
Queue queue = new ArrayQueue(6);
```



ArrayQueue

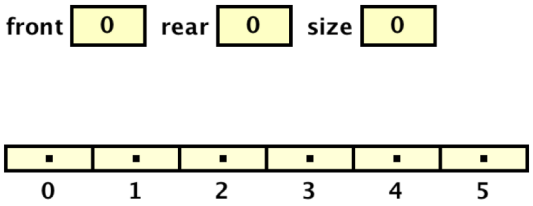
```
public class ArrayQueue<T> implements Queue<T> {

    public boolean isEmpty() {
        return size() == 0;
    }

    public int size() {
        return size;
    }
```

```
Queue queue = new ArrayQueue(6);
```

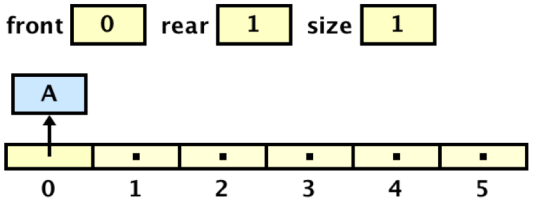


ArrayQueue

```
public class ArrayQueue<T> implements Queue<T> {

    public void enqueue(T element) {
        if (size() == elements.length) {
            resize(elements.length * 2);
        }
        elements[rear] = element;
        rear = (rear + 1) % elements.length;
        size++;
    }
```

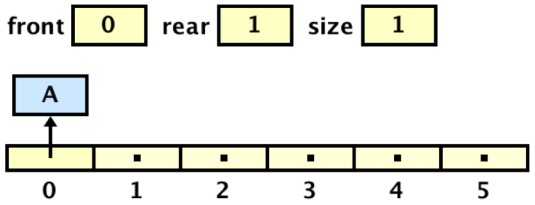



```
queue.enqueue("A");
```

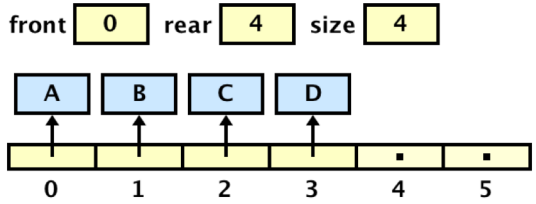


ArrayQueue

```
public class ArrayQueue<T> implements Queue<T> {

    public void enqueue(T element) {
        if (size() == elements.length) {
            resize(elements.length * 2);
        }
        elements[rear] = element;
        rear = (rear + 1) % elements.length;
        size++;
    }
```

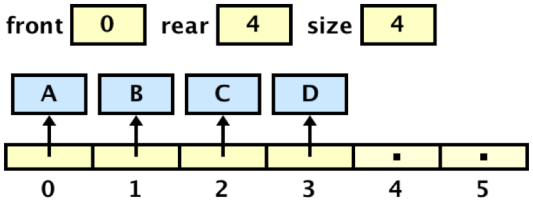



```
queue.enqueue("B");
queue.enqueue("C");
queue.enqueue("D");
```

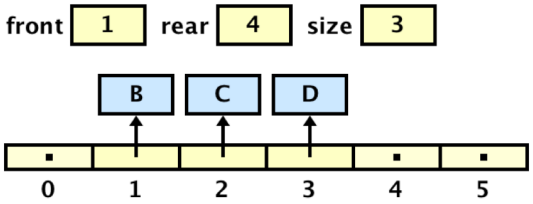


ArrayQueue

```
public class ArrayQueue<T> implements Queue<T> {

    public void enqueue(T element) {
        if (size() == elements.length) {
            resize(elements.length * 2);
        }
        elements[rear] = element;
        rear = (rear + 1) % elements.length;
        size++;
    }

    public T dequeue() {
        if (isEmpty()) {
            return null;
        }
        T result = elements[front];
        elements[front] = null;
        front = (front + 1) % elements.length;
        size--;
        return result;
    }
```

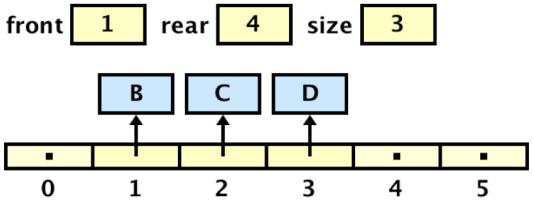


queue.dequeue();

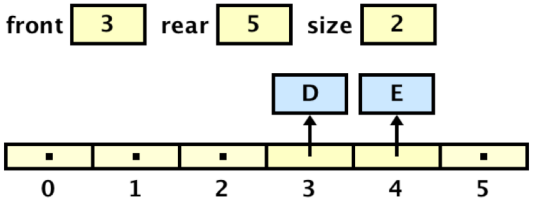


ArrayQueue

```
public class ArrayQueue<T> implements Queue<T> {

    public void enqueue(T element) {
        if (size() == elements.length) {
            resize(elements.length * 2);
        }
        elements[rear] = element;
        rear = (rear + 1) % elements.length;
        size++;
    }

    public T dequeue() {
        if (isEmpty()) {
            return null;
        }
        T result = elements[front];
        elements[front] = null;
        front = (front + 1) % elements.length;
        size--;
        return result;
    }
```

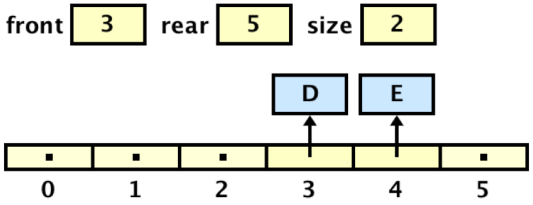



```
queue.dequeue();
queue.dequeue();
queue.enqueue("E");
```

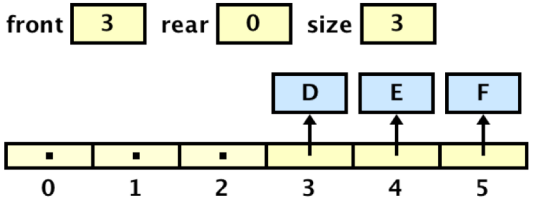


ArrayQueue

```
public class ArrayQueue<T> implements Queue<T> {

    public void enqueue(T element) {
        if (size() == elements.length) {
            resize(elements.length * 2);
        }
        elements[rear] = element;
        rear = (rear + 1) % elements.length;
        size++;
    }

    public T dequeue() {
        if (isEmpty()) {
            return null;
        }
        T result = elements[front];
        elements[front] = null;
        front = (front + 1) % elements.length;
        size--;
        return result;
    }
```

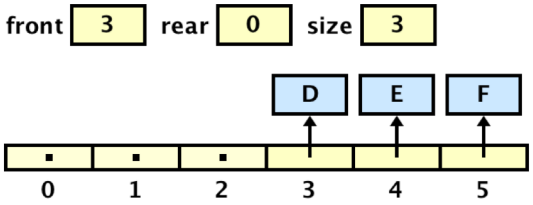



```
queue.enqueue("F");
```

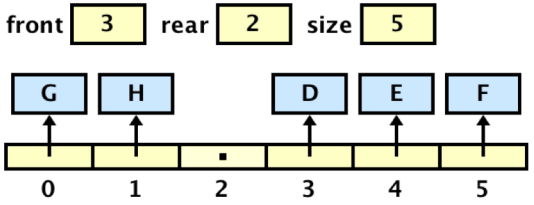


ArrayQueue

```
public class ArrayQueue<T> implements Queue<T> {

    public void enqueue(T element) {
        if (size() == elements.length) {
            resize(elements.length * 2);
        }
        elements[rear] = element;
        rear = (rear + 1) % elements.length;
        size++;
    }

    public T dequeue() {
        if (isEmpty()) {
            return null;
        }
        T result = elements[front];
        elements[front] = null;
        front = (front + 1) % elements.length;
        size--;
        return result;
    }
```

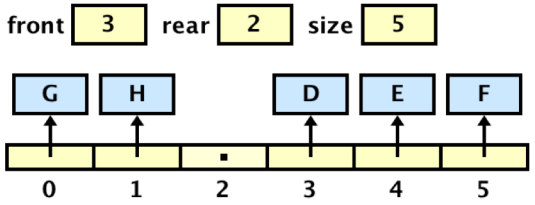



```
queue.enqueue("G");
queue.enqueue("H");
```



ArrayQueue

```
public class ArrayQueue<T> implements Queue<T> {

    public T first() {
        if (isEmpty()) {
            return null;
        }
        return elements[front];
    }
```



ArrayQueue

```
public class ArrayQueue<T> implements Queue<T> {

    public T first() {
        if (isEmpty()) {
            return null;
        }
        return elements[front];
    }
```

O(1)

Time complexity:

ArrayQueue

```
public class ArrayQueue<T> implements Queue<T> {

    public void enqueue(T element) {
        if (size() == elements.length) {
            resize(elements.length * 2);
        }
        elements[rear] = element;
        rear = (rear + 1) % elements.length;
        size++;
    }

    public T dequeue() {
        if (isEmpty()) {
            return null;
        }
        T result = elements[front];
        elements[front] = null;
        front = (front + 1) % elements.length;
        size--;
        return result;
    }
```

O(1) *amortized*

O(1)

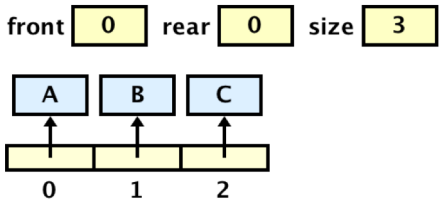
Time complexity:

O(1) for all three methods

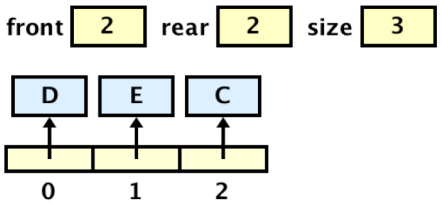
ArrayQueue

```
public class ArrayQueue<T> implements Queue<T> {

    public void enqueue(T element) {
        if (size() == elements.length) {
            resize(elements.length * 2);
        }
        elements[rear] = element;
        rear = (rear + 1) % elements.length;
        size++;
    }
```

```
Queue queue = new ArrayQueue(3);
queue.enqueue("A");
queue.enqueue("B");
queue.enqueue("C");
```

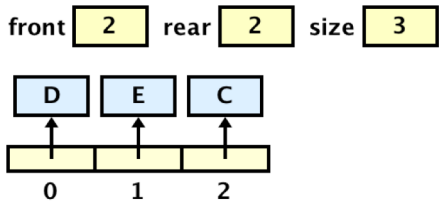



```
queue.dequeue();
queue.dequeue();
queue.enqueue("D");
queue.enqueue("E");
```

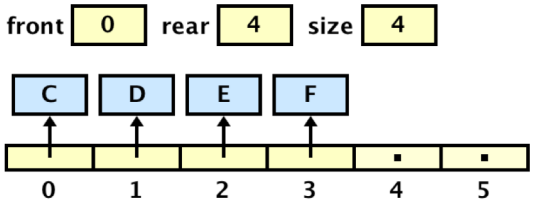


ArrayQueue

```
public class ArrayQueue<T> implements Queue<T> {

    public void enqueue(T element) {
        if (size() == elements.length) {
            resize(elements.length * 2);
        }
        elements[rear] = element;
        rear = (rear + 1) % elements.length;
        size++;
    }
```

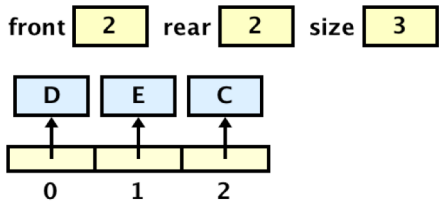



```
queue.enqueue("F");
```

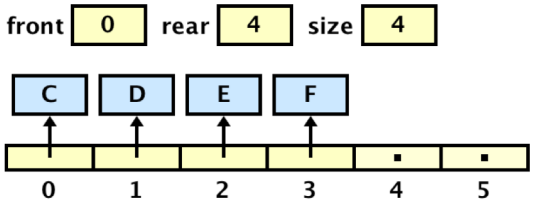


ArrayQueue

```
public class ArrayQueue<T> implements Queue<T> {

    public void enqueue(T element) {
        if (size() == elements.length) {
            resize(elements.length * 2);
        }
        elements[rear] = element;
        rear = (rear + 1) % elements.length;
        size++;
    }

    private void resize(int newSize) {
        T[] newArray = (T[]) new Object[newSize];
        int i = 0;
        int j = front;
        for (int k = 0; k < size; k++) {
            newArray[i] = elements[j];
            i++;
            j = (j + 1) % elements.length;
        }
        elements = newArray;
        front = 0;
        rear = size;
    }
```

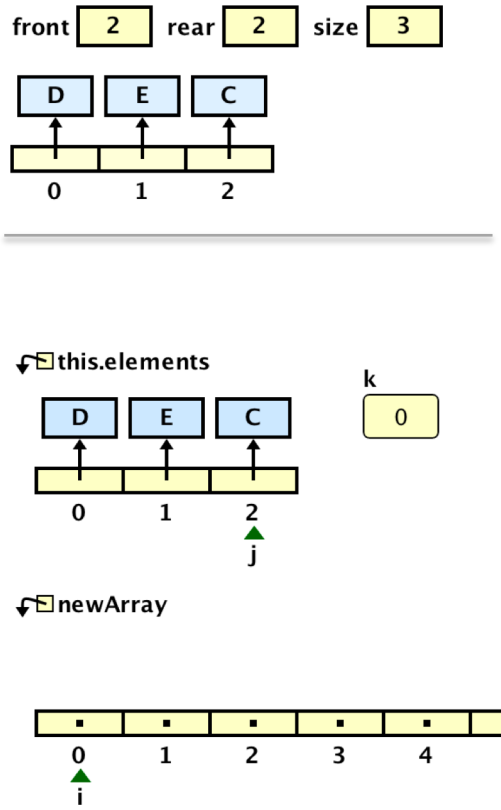



```
queue.enqueue("F");
```



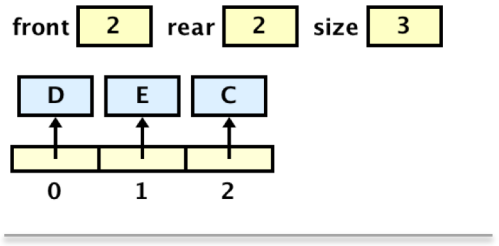
ArrayQueue

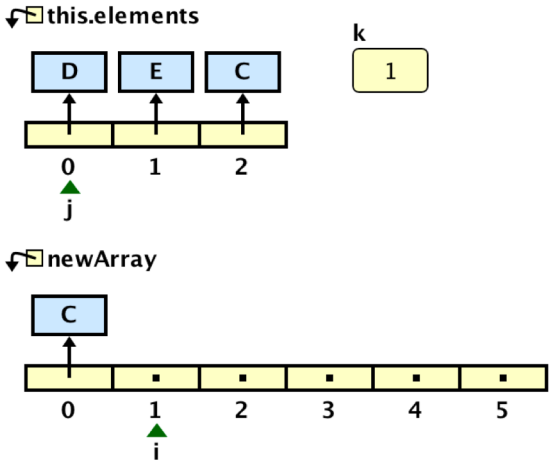
```
public class ArrayQueue<T> implements Queue<T> {

    public void enqueue(T element) {
        if (size() == elements.length) {
            resize(elements.length * 2);
        }
        elements[rear] = element;
        rear = (rear + 1) % elements.length;
        size++;
    }

    private void resize(int newSize) {
        T[] newArray = (T[]) new Object[newSize];
        int i = 0;
        int j = front;
        for (int k = 0; k < size; k++) {
            newArray[i] = elements[j];
            i++;
            j = (j + 1) % elements.length;
        }
        elements = newArray;
        front = 0;
        rear = size;
    }
```



ArrayQueue

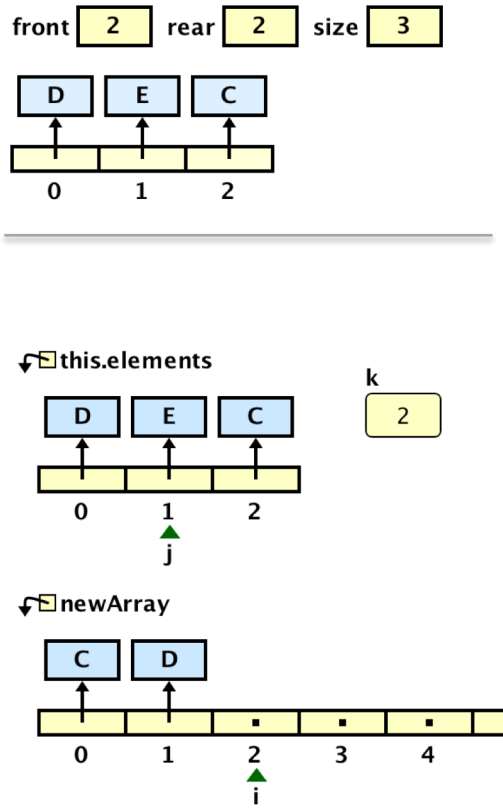
```
public class ArrayQueue<T> implements Queue<T> {

    public void enqueue(T element) {
        if (size() == elements.length) {
            resize(elements.length * 2);
        }
        elements[rear] = element;
        rear = (rear + 1) % elements.length;
        size++;
    }

    private void resize(int newSize) {
        T[] newArray = (T[]) new Object[newSize];
        int i = 0;
        int j = front;
        for (int k = 0; k < size; k++) {
            newArray[i] = elements[j];
            i++;
            j = (j + 1) % elements.length;
        }
        elements = newArray;
        front = 0;
        rear = size;
    }
```





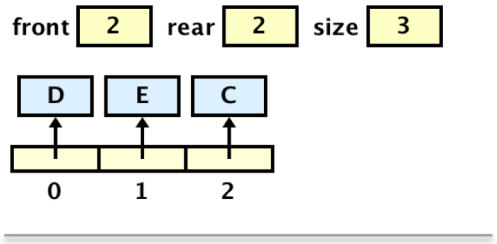
ArrayQueue

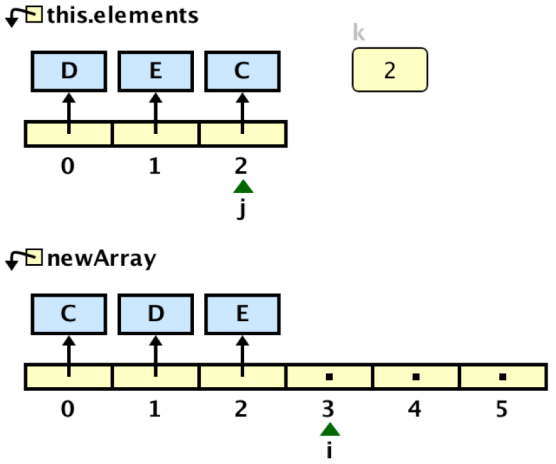
```
public class ArrayQueue<T> implements Queue<T> {

    public void enqueue(T element) {
        if (size() == elements.length) {
            resize(elements.length * 2);
        }
        elements[rear] = element;
        rear = (rear + 1) % elements.length;
        size++;
    }

    private void resize(int newSize) {
        T[] newArray = (T[]) new Object[newSize];
        int i = 0;
        int j = front;
        for (int k = 0; k < size; k++) {
            newArray[i] = elements[j];
            i++;
            j = (j + 1) % elements.length;
        }
        elements = newArray;
        front = 0;
        rear = size;
    }
```



ArrayQueue

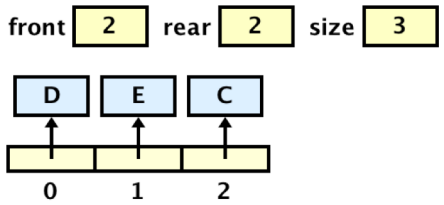
```
public class ArrayQueue<T> implements Queue<T> {

    public void enqueue(T element) {
        if (size() == elements.length) {
            resize(elements.length * 2);
        }
        elements[rear] = element;
        rear = (rear + 1) % elements.length;
        size++;
    }

    private void resize(int newSize) {
        T[] newArray = (T[]) new Object[newSize];
        int i = 0;
        int j = front;
        for (int k = 0; k < size; k++) {
            newArray[i] = elements[j];
            i++;
            j = (j + 1) % elements.length;
        }
        elements = newArray;
        front = 0;
        rear = size;
    }
```



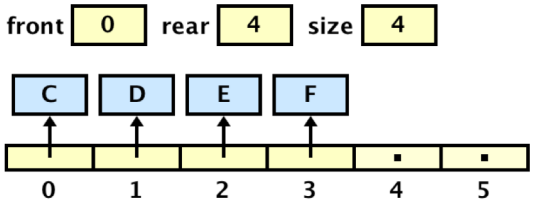


ArrayQueue

```
public class ArrayQueue<T> implements Queue<T> {

    public void enqueue(T element) {
        if (size() == elements.length) {
            resize(elements.length * 2);
        }
        elements[rear] = element;
        rear = (rear + 1) % elements.length;
        size++;
    }

    private void resize(int newSize) {
        T[] newArray = (T[]) new Object[newSize];
        int i = 0;
        int j = front;
        for (int k = 0; k < size; k++) {
            newArray[i] = elements[j];
            i++;
            j = (j + 1) % elements.length;
        }
        elements = newArray;
        front = 0;
        rear = size;
    }
```




```
queue.enqueue("F");
```



Applications – stack machine evaluation

Building a calculator

```
Enter an arithmetic expression:  2 + 3 * 5 – 12 / 4

                                 2 + 3 * 5 – 12 / 4 = 14
```

Get the infix expression as a *tokenized* stream.

2	+	3	*	5	-	12	/	4

Use a linear scan to evaluate the expression … harder than it looks!

Precedence and associativity make it difficult to evaluate an infix expression as we go using a linear scan … *unless we require all parentheses or require postfix or infix notation.*

Full parentheses

Requiring all parentheses:

```
2 + 3 * 5 - 12 / 4
```

```
((2 + (3 * 5)) - (12 / 4))
```

(	(	2	+	(	3	*	5	)	)	-	(	12	/	4	)	)

Pen and paper strategy for fully parenthesized infix:

```
Scan from left to right:
    when you get to a ) token:
        evaluate the most recent subexpression
        store the result for later
```

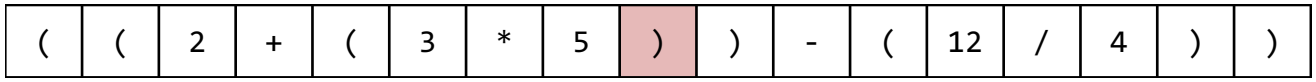
Full parentheses

```
Scan from left to right:
    when you get to a ) token:
        evaluate the most recent subexpression
        store the result for later
```

(	(	2	+	(	3	*	5	)	)	-	(	12	/	4	)	)

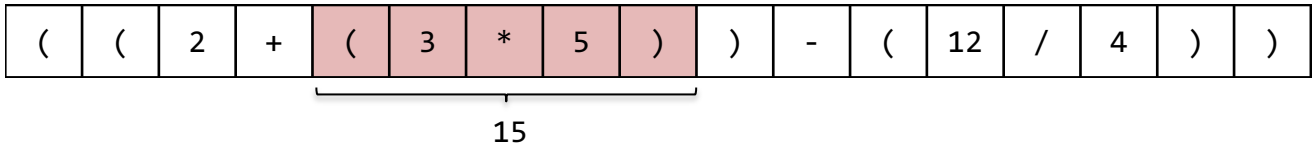
Full parentheses

```
Scan from left to right:
    when you get to a ) token:
        evaluate the most recent subexpression
        store the result for later
```



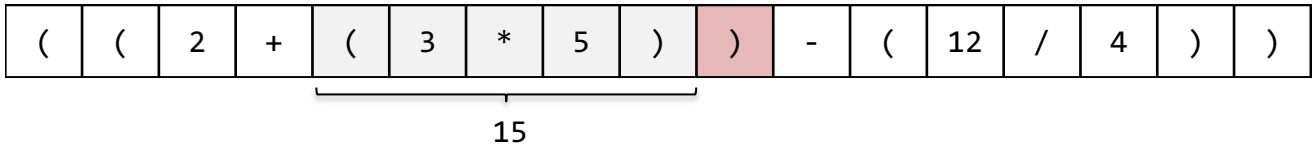
Full parentheses

```
Scan from left to right:
    when you get to a ) token:
        evaluate the most recent subexpression
        store the result for later
```



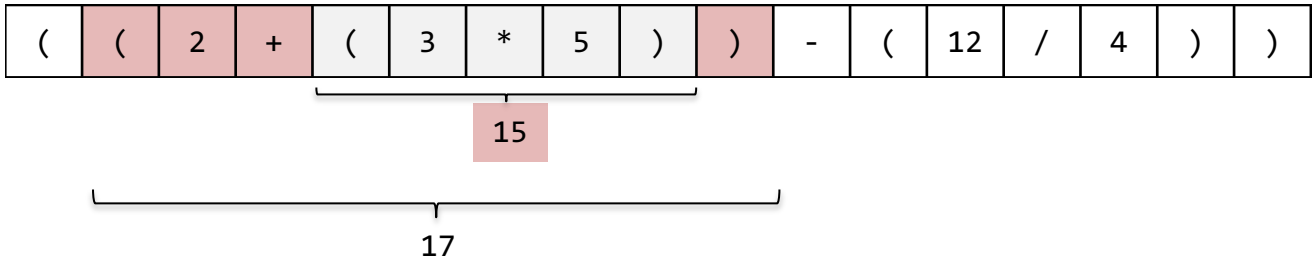
Full parentheses

```
Scan from left to right:
    when you get to a ) token:
        evaluate the most recent subexpression
        store the result for later
```



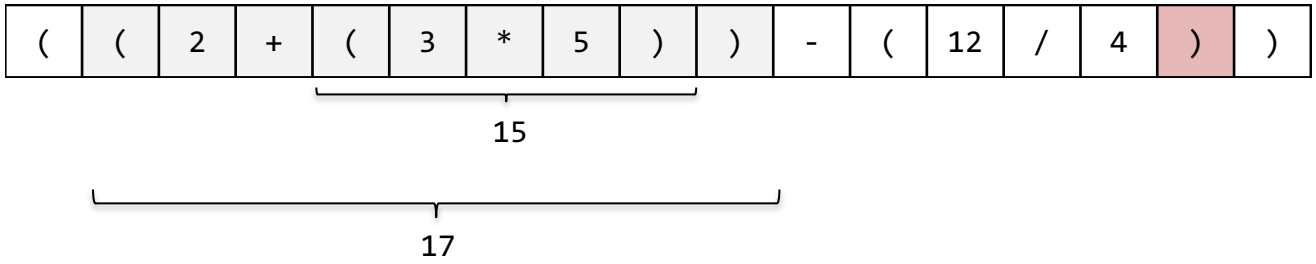
Full parentheses

```
Scan from left to right:
    when you get to a ) token:
        evaluate the most recent subexpression
        store the result for later
```



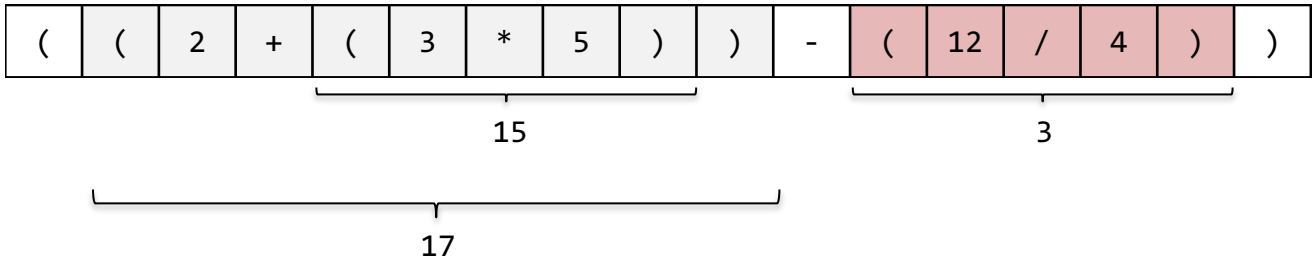
Full parentheses

```
Scan from left to right:
    when you get to a ) token:
        evaluate the most recent subexpression
        store the result for later
```



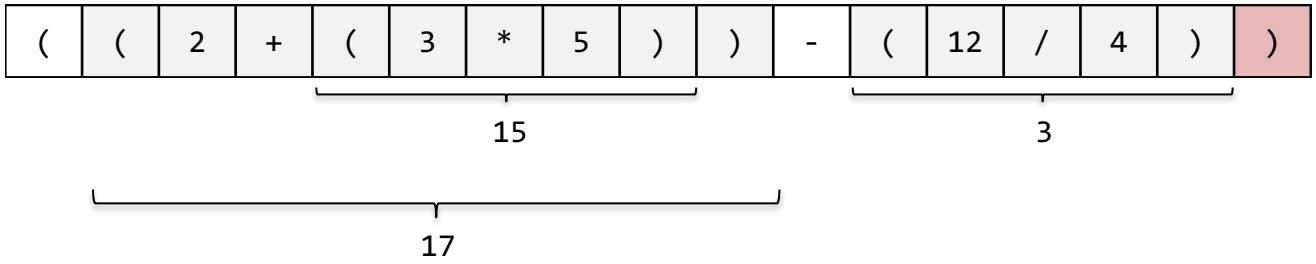
Full parentheses

```
Scan from left to right:
    when you get to a ) token:
        evaluate the most recent subexpression
        store the result for later
```



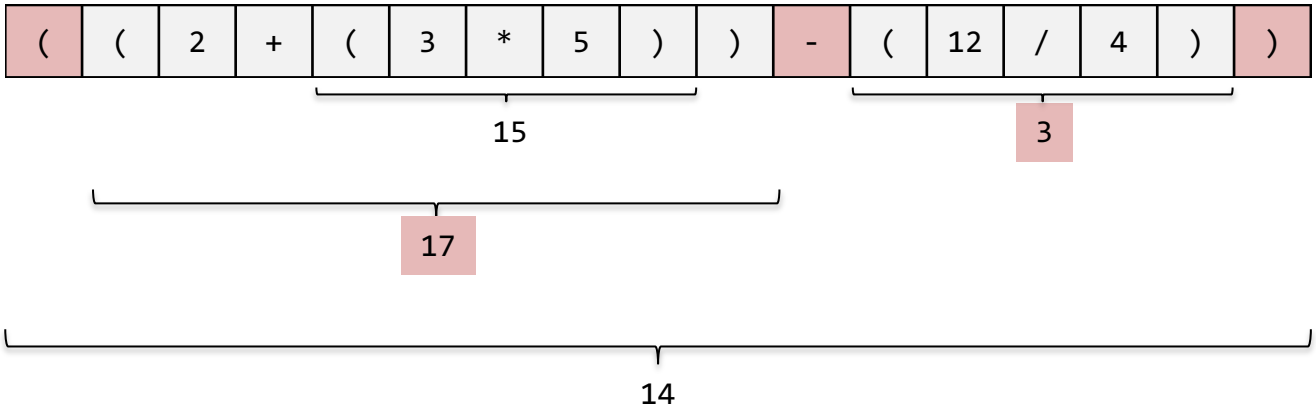
Full parentheses

```
Scan from left to right:
    when you get to a ) token:
        evaluate the most recent subexpression
        store the result for later
```



Full parentheses

```
Scan from left to right:
    when you get to a ) token:
        evaluate the most recent subexpression
        store the result for later
```



Full parentheses

```
Scan from left to right:
    when you get to a ) token:
        evaluate the most recent subexpression
        store the result for later
```

(	(	2	+	(	3	*	5	)	)	-	(	12	/	4	)	)

= 14

Postfix (RPN)

Requiring postfix/RPN:

`2 + 3 * 5 – 12 / 4`  `2 3 5 * + 12 4 / -`

2	3	5	*	+	12	4	/	-

Pen and paper strategy for postfix:

```
Scan from left to right:
    when you get to an operator token:
        evaluate using the most recent values
        store the result for later
```

Postfix (RPN)

```
Scan from left to right:
    when you get to an operator token:
        evaluate using the most recent values
        store the result for later
```

2	3	5	*	+	12	4	/	-

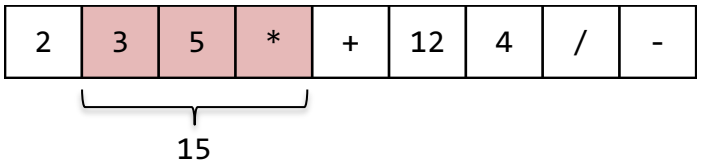
Postfix (RPN)

```
Scan from left to right:
    when you get to an operator token:
        evaluate using the most recent values
        store the result for later
```

2	3	5	*	+	12	4	/	-

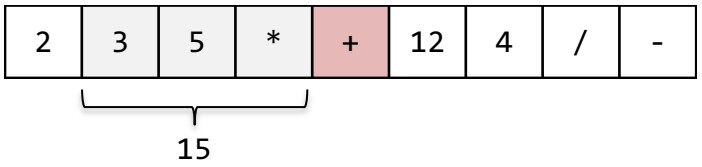
Postfix (RPN)

```
Scan from left to right:
    when you get to an operator token:
        evaluate using the most recent values
        store the result for later
```



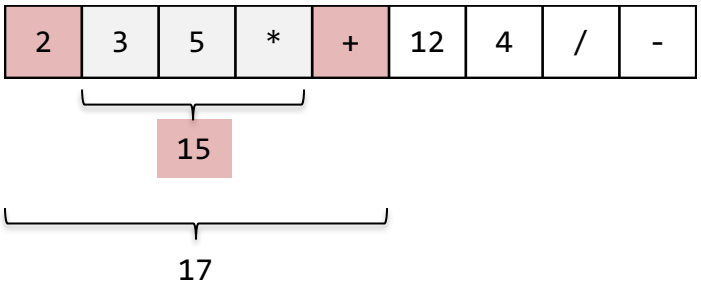
Postfix (RPN)

```
Scan from left to right:
    when you get to an operator token:
        evaluate using the most recent values
        store the result for later
```



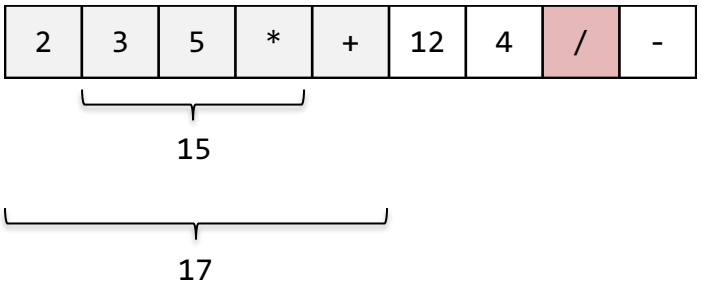
Postfix (RPN)

```
Scan from left to right:
    when you get to an operator token:
        evaluate using the most recent values
        store the result for later
```



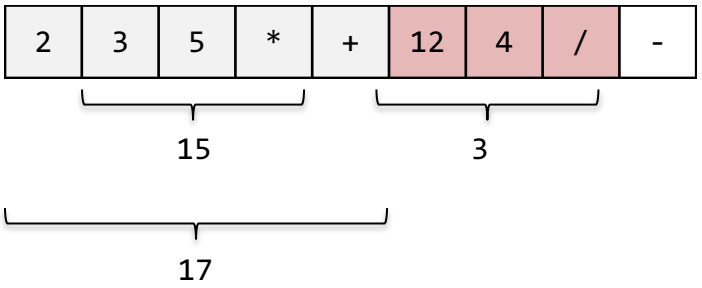
Postfix (RPN)

```
Scan from left to right:
    when you get to an operator token:
        evaluate using the most recent values
        store the result for later
```



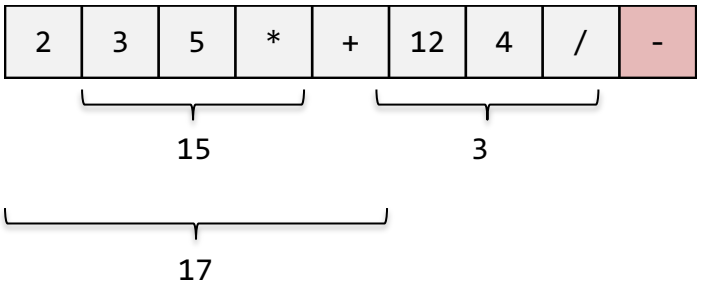
Postfix (RPN)

```
Scan from left to right:
    when you get to an operator token:
        evaluate using the most recent values
        store the result for later
```



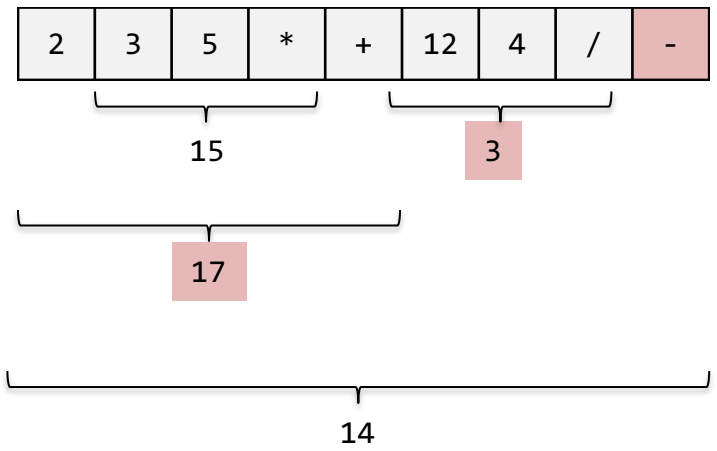
Postfix (RPN)

```
Scan from left to right:
    when you get to an operator token:
        evaluate using the most recent values
        store the result for later
```



Postfix (RPN)

```
Scan from left to right:
    when you get to an operator token:
        evaluate using the most recent values
        store the result for later
```



Postfix (RPN)

```
Scan from left to right:
    when you get to an operator token:
        evaluate using the most recent values
        store the result for later
```

2	3	5	*	+	12	4	/	-

= 14

Prefix

Requiring prefix:

`2 + 3 * 5 – 12 / 4`  `+ 2 - * 3 5 / 12 4`

+	2	-	*	3	5	/	12	4

Pen and paper strategy for prefix:

```
Scan from right to left:
    when you get to an operator token:
        evaluate using the most recent values
        store the result for later
```

Prefix

```
Scan from right to left:
    when you get to an operator token:
        evaluate using the most recent values
        store the result for later
```

+	2	-	*	3	5	/	12	4

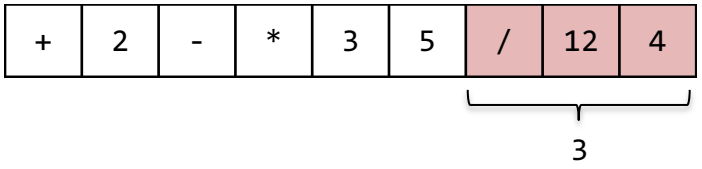
Prefix

```
Scan from right to left:
    when you get to an operator token:
        evaluate using the most recent values
        store the result for later
```

+	2	-	*	3	5	/	12	4

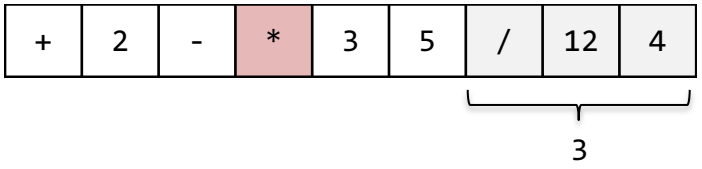
Prefix

```
Scan from right to left:
    when you get to an operator token:
        evaluate using the most recent values
        store the result for later
```



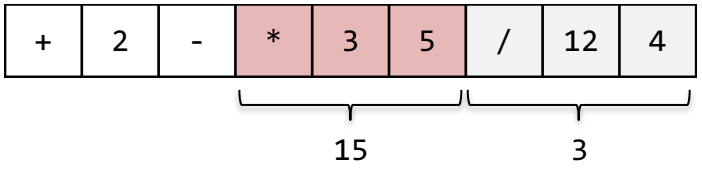
Prefix

```
Scan from right to left:
    when you get to an operator token:
        evaluate using the most recent values
        store the result for later
```



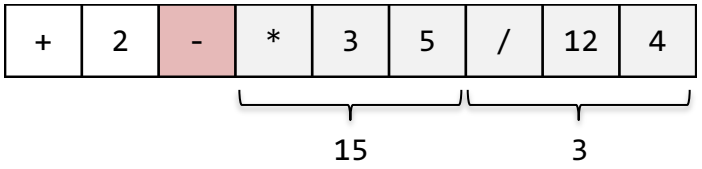
Prefix

```
Scan from right to left:
    when you get to an operator token:
        evaluate using the most recent values
        store the result for later
```



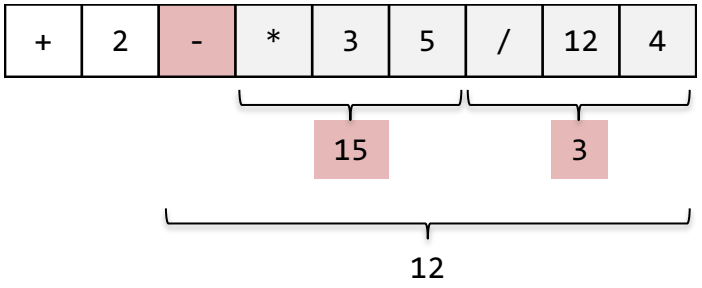
Prefix

```
Scan from right to left:
    when you get to an operator token:
        evaluate using the most recent values
        store the result for later
```



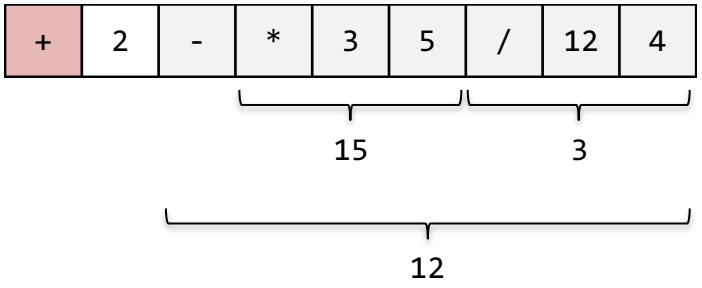
Prefix

```
Scan from right to left:
    when you get to an operator token:
        evaluate using the most recent values
        store the result for later
```



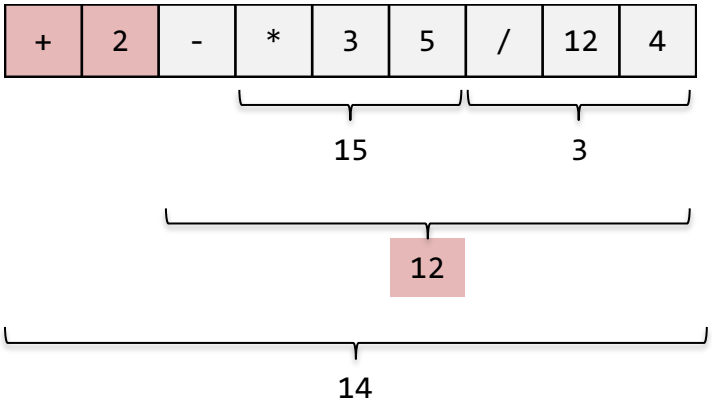
Prefix

```
Scan from right to left:
    when you get to an operator token:
        evaluate using the most recent values
        store the result for later
```



Prefix

```
Scan from right to left:
    when you get to an operator token:
        evaluate using the most recent values
        store the result for later
```



Prefix

```
Scan from right to left:
    when you get to an operator token:
        evaluate using the most recent values
        store the result for later
```

+	2	-	*	3	5	/	12	4

= 14

Evaluation strategies

Infix
(full paren)

```
Scan from left to right:
    when you get to a ) token:
        evaluate the most recent subexpression
        store the result for later
```

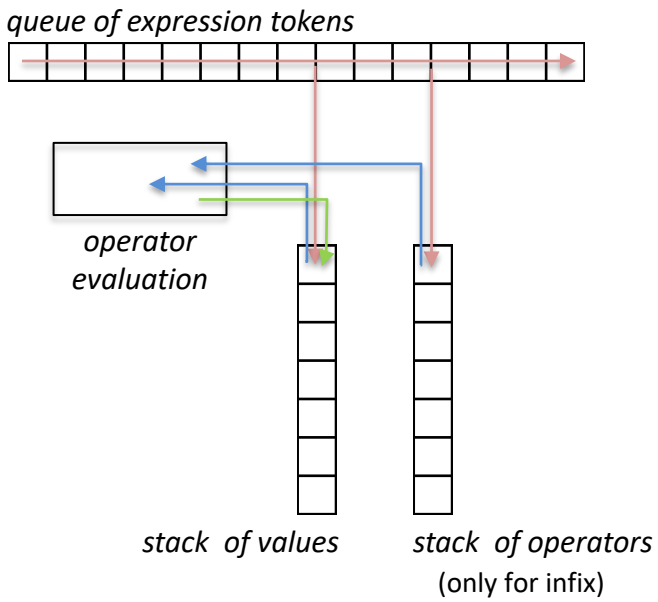
Postfix

```
Scan from left to right:
    when you get to an operator token:
        evaluate using the most recent values
        store the result for later
```

Prefix

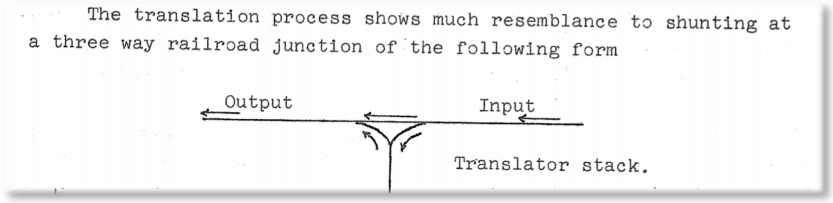
```
Scan from right to left:
    when you get to an operator token:
        evaluate using the most recent values
        store the result for later
```

Linear scan with stack memory



Shunting yard algorithm

These evaluation strategies are variations on the "Shunting Yard" algorithm developed by Edsger Dijkstra in 1961.

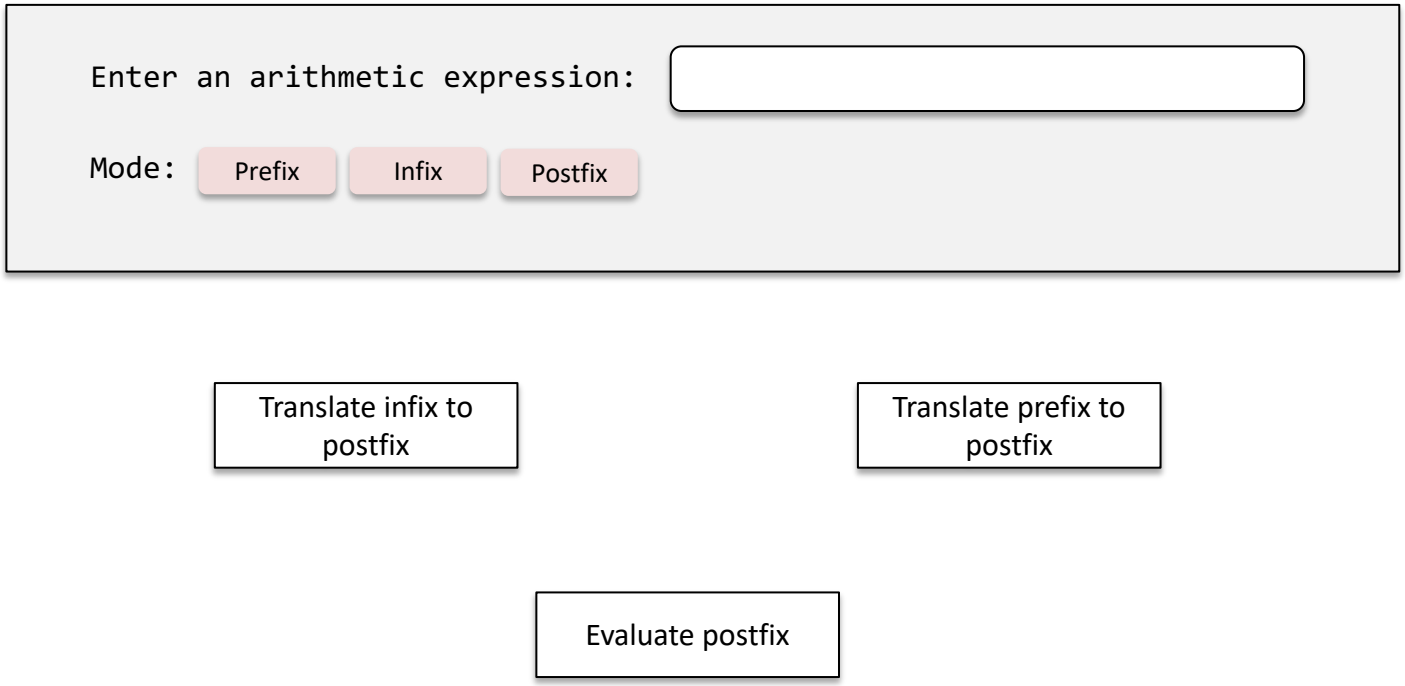


Dijkstra, E.W. (1961). *Algol 60 translation : An algol 60 translator for the x1 and making a translator for algol 60. Stichting Mathematisch Centrum. Rekenafdeling*. Stichting Mathematisch Centrum.

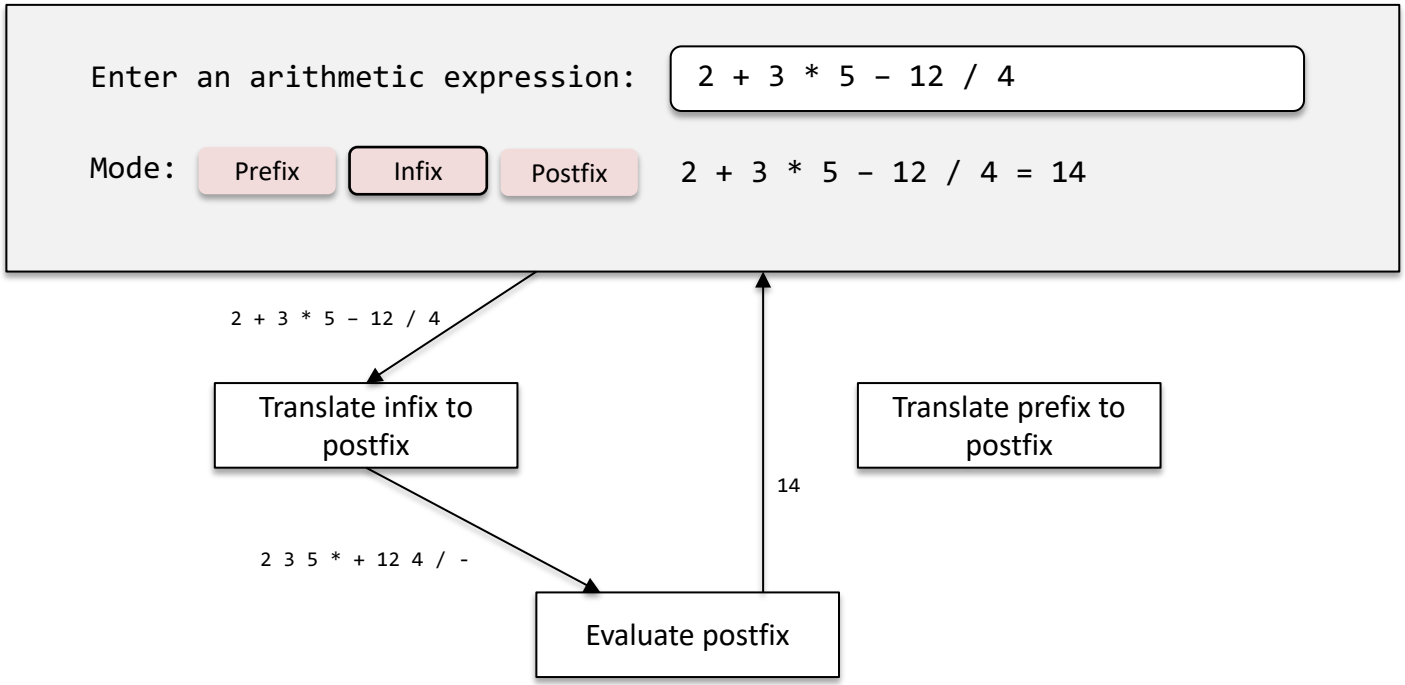
Dijkstra developed the Shunting Yard algorithm to convert an infix expression into an equivalent postfix expression, as part of a larger translator for the programming language Algol 60.

Variations of this algorithm can be used to translate one form into another or to evaluate a given expression form.

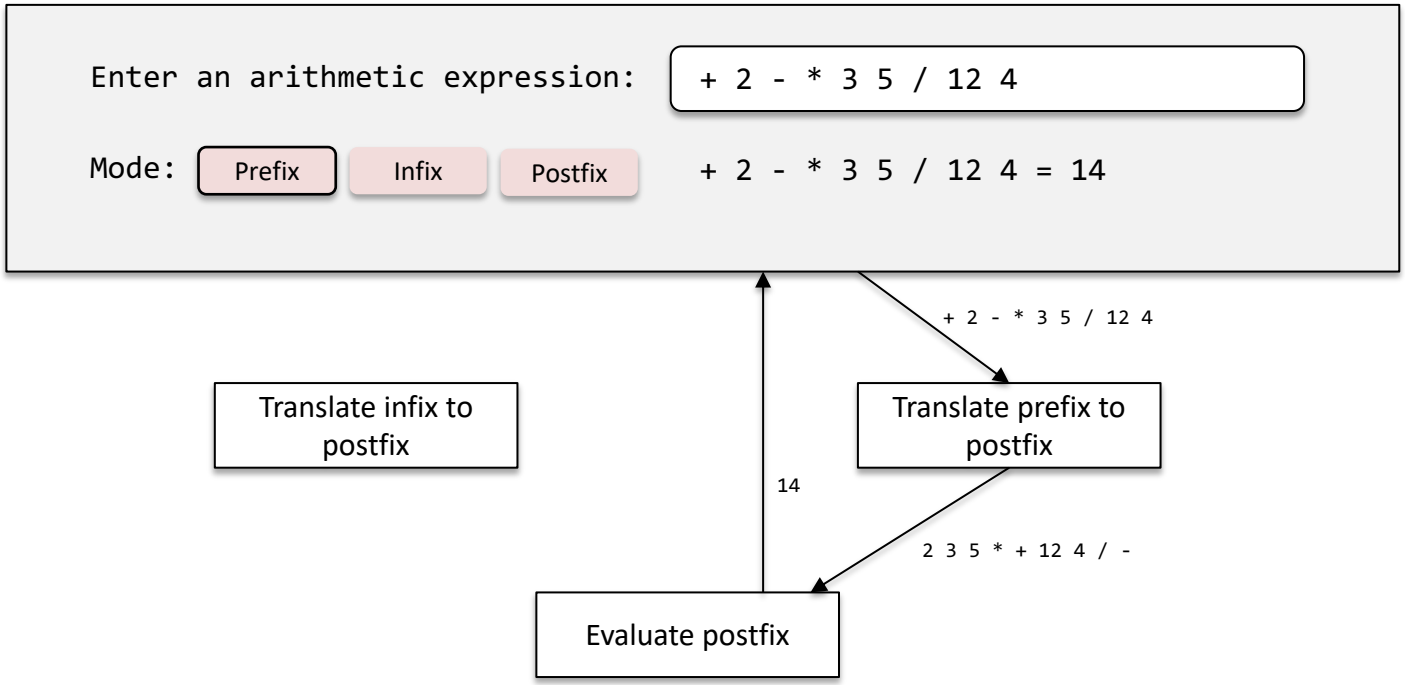
A Shunting Yard Calculator



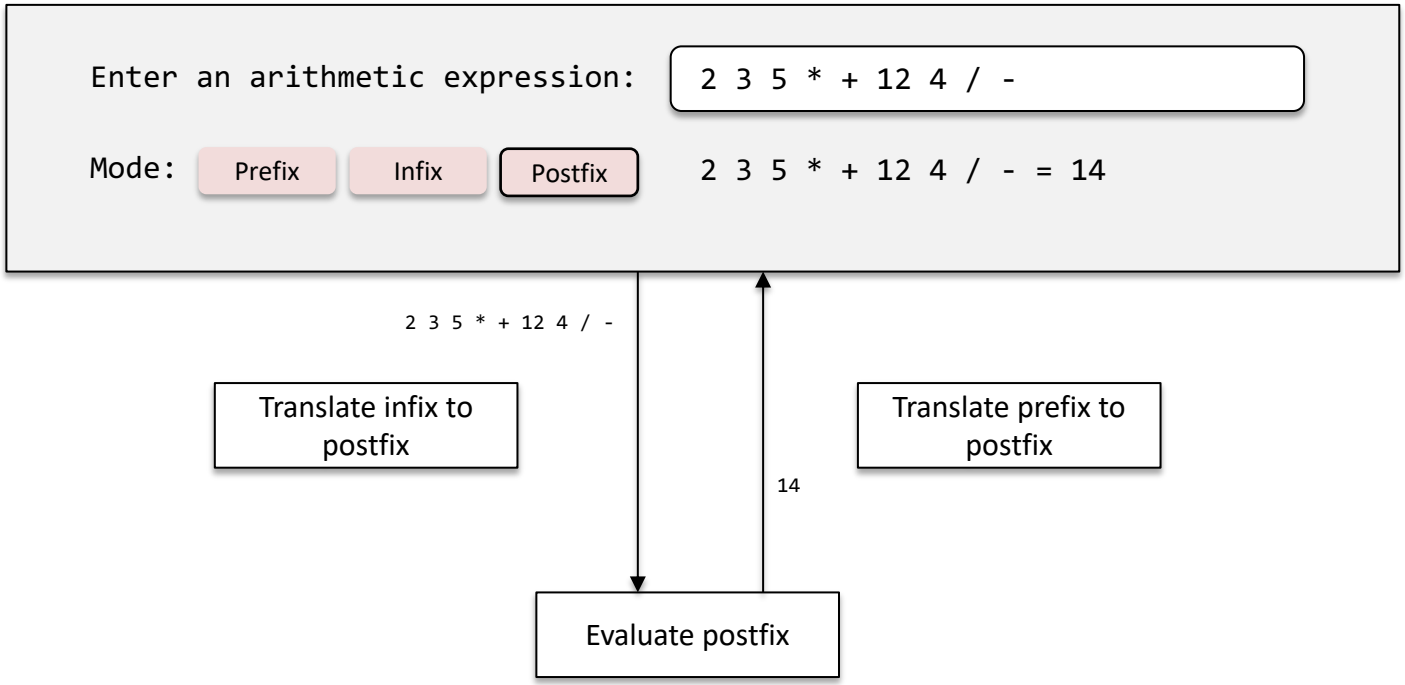
A Shunting Yard Calculator



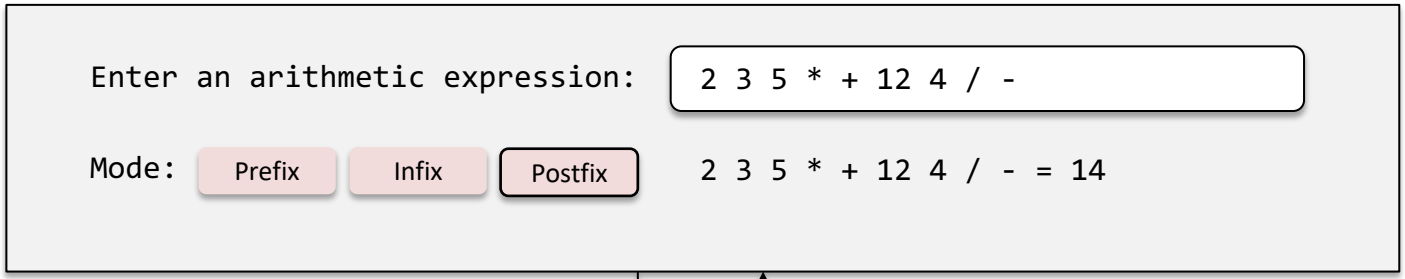
A Shunting Yard Calculator



A Shunting Yard Calculator



A Shunting Yard Calculator



2 3 5 * + 12 4 / -

Translate infix to postfix

Translate prefix to postfix

14

Evaluate postfix

```
public double evaluatePostfix(Queue<String> expression) {
    Stack<Double> valueStack = new LinkedStack<>();
        for (String token : expression) {
            if (isOperator(token)) {
                double rightValue = valueStack.pop();
                double leftValue = valueStack.pop();
                valueStack.push(evaluate(token, leftValue, rightValue));
            } else if (isValue(token)) {
                valueStack.push(Double.parseDouble(token));
            }
        }
        return valueStack.peek();
}
```

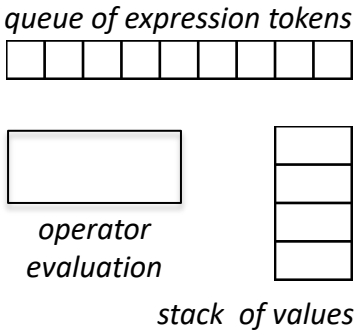
evaluatePostfix

Enter an arithmetic expression: `2 3 5 * + 12 4 / -`

Mode: Prefix | Infix | Postfix

`2 3 5 * + 12 4 / - = 14`

```
public double evaluatePostfix(Queue<String> expression) {
    Stack<Double> valueStack = new LinkedStack<>();
        for (String token : expression) {
            if (isOperator(token)) {
                double rightValue = valueStack.pop();
                double leftValue = valueStack.pop();
                valueStack.push(evaluate(token, leftValue, rightValue));
            } else if (isValue(token)) {
                valueStack.push(Double.parseDouble(token));
            }
        }
        return valueStack.peek();
}
```



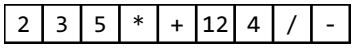
evaluatePostfix

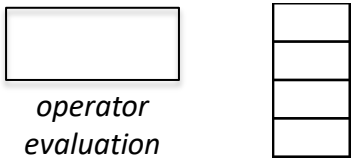
Enter an arithmetic expression: 2 3 5 * + 12 4 / -

Mode: Prefix Infix Postfix 2 3 5 * + 12 4 / - = 14

```
public double evaluatePostfix(Queue<String> expression) {
    Stack<Double> valueStack = new LinkedStack<>();
        for (String token : expression) {
            if (isOperator(token)) {
                double rightValue = valueStack.pop();
                double leftValue = valueStack.pop();
                valueStack.push(evaluate(token, leftValue, rightValue));
            } else if (isValue(token)) {
                valueStack.push(Double.parseDouble(token));
            }
        }
        return valueStack.peek();
}
```

queue of expression tokens





operator evaluation

stack of values

evaluatePostfix

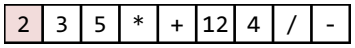
Enter an arithmetic expression: `2 3 5 * + 12 4 / -`

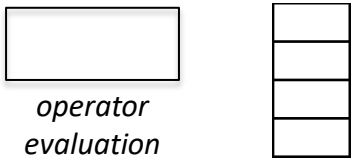
Mode: Prefix | Infix | **Postfix**

`2 3 5 * + 12 4 / - = 14`

```
public double evaluatePostfix(Queue<String> expression) {
    Stack<Double> valueStack = new LinkedStack<>();
        for (String token : expression) {
            if (isOperator(token)) {
                double rightValue = valueStack.pop();
                double leftValue = valueStack.pop();
                valueStack.push(evaluate(token, leftValue, rightValue));
            } else if (isValue(token)) {
                valueStack.push(Double.parseDouble(token));
            }
        }
        return valueStack.peek();
}
```

queue of expression tokens





operator evaluation

stack of values

evaluatePostfix

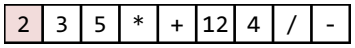
Enter an arithmetic expression: `2 3 5 * + 12 4 / -`

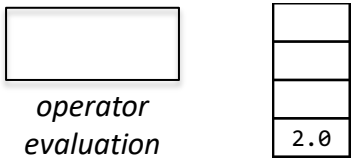
Mode: Prefix | Infix | **Postfix**

`2 3 5 * + 12 4 / - = 14`

```
public double evaluatePostfix(Queue<String> expression) {
    Stack<Double> valueStack = new LinkedStack<>();
        for (String token : expression) {
            if (isOperator(token)) {
                double rightValue = valueStack.pop();
                double leftValue = valueStack.pop();
                valueStack.push(evaluate(token, leftValue, rightValue));
            } else if (isValue(token)) {
                valueStack.push(Double.parseDouble(token));
            }
        }
        return valueStack.peek();
}
```

queue of expression tokens





operator evaluation

stack of values

evaluatePostfix

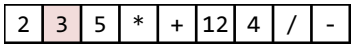
Enter an arithmetic expression: `2 3 5 * + 12 4 / -`

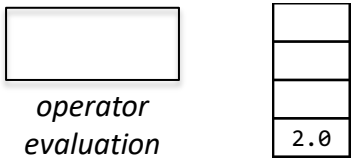
Mode: Prefix | Infix | **Postfix**

`2 3 5 * + 12 4 / - = 14`

```
public double evaluatePostfix(Queue<String> expression) {
    Stack<Double> valueStack = new LinkedStack<>();
        for (String token : expression) {
            if (isOperator(token)) {
                double rightValue = valueStack.pop();
                double leftValue = valueStack.pop();
                valueStack.push(evaluate(token, leftValue, rightValue));
            } else if (isValue(token)) {
                valueStack.push(Double.parseDouble(token));
            }
        }
        return valueStack.peek();
}
```

queue of expression tokens





operator evaluation

stack of values

evaluatePostfix

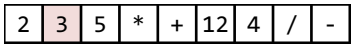
Enter an arithmetic expression: `2 3 5 * + 12 4 / -`

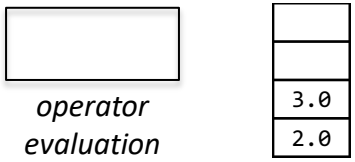
Mode: Prefix | Infix | **Postfix**

`2 3 5 * + 12 4 / - = 14`

```
public double evaluatePostfix(Queue<String> expression) {
    Stack<Double> valueStack = new LinkedStack<>();
        for (String token : expression) {
            if (isOperator(token)) {
                double rightValue = valueStack.pop();
                double leftValue = valueStack.pop();
                valueStack.push(evaluate(token, leftValue, rightValue));
            } else if (isValue(token)) {
                valueStack.push(Double.parseDouble(token));
            }
        }
        return valueStack.peek();
}
```

queue of expression tokens





operator evaluation

stack of values

evaluatePostfix

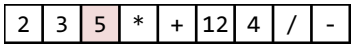
Enter an arithmetic expression: `2 3 5 * + 12 4 / -`

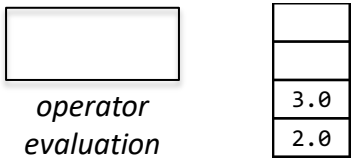
Mode: Prefix | Infix | **Postfix**

`2 3 5 * + 12 4 / - = 14`

```
public double evaluatePostfix(Queue<String> expression) {
    Stack<Double> valueStack = new LinkedStack<>();
        for (String token : expression) {
            if (isOperator(token)) {
                double rightValue = valueStack.pop();
                double leftValue = valueStack.pop();
                valueStack.push(evaluate(token, leftValue, rightValue));
            } else if (isValue(token)) {
                valueStack.push(Double.parseDouble(token));
            }
        }
        return valueStack.peek();
}
```

queue of expression tokens





operator evaluation

stack of values

evaluatePostfix

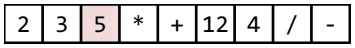
Enter an arithmetic expression: `2 3 5 * + 12 4 / -`

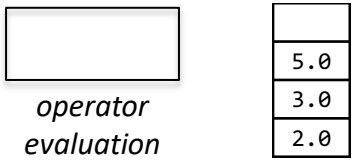
Mode: Prefix | Infix | **Postfix**

`2 3 5 * + 12 4 / - = 14`

```
public double evaluatePostfix(Queue<String> expression) {
    Stack<Double> valueStack = new LinkedStack<>();
        for (String token : expression) {
            if (isOperator(token)) {
                double rightValue = valueStack.pop();
                double leftValue = valueStack.pop();
                valueStack.push(evaluate(token, leftValue, rightValue));
            } else if (isValue(token)) {
                valueStack.push(Double.parseDouble(token));
            }
        }
        return valueStack.peek();
}
```

queue of expression tokens





operator evaluation

stack of values

evaluatePostfix

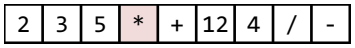
Enter an arithmetic expression: `2 3 5 * + 12 4 / -`

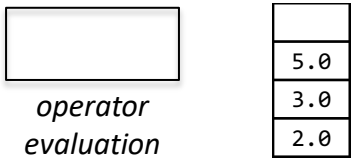
Mode: Prefix | Infix | **Postfix**

`2 3 5 * + 12 4 / - = 14`

```
public double evaluatePostfix(Queue<String> expression) {
    Stack<Double> valueStack = new LinkedStack<>();
        for (String token : expression) {
            if (isOperator(token)) {
                double rightValue = valueStack.pop();
                double leftValue = valueStack.pop();
                valueStack.push(evaluate(token, leftValue, rightValue));
            } else if (isValue(token)) {
                valueStack.push(Double.parseDouble(token));
            }
        }
        return valueStack.peek();
}
```

queue of expression tokens





operator evaluation

stack of values

evaluatePostfix

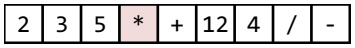
Enter an arithmetic expression: `2 3 5 * + 12 4 / -`

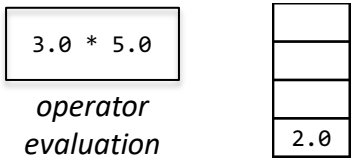
Mode: Prefix | Infix | **Postfix**

`2 3 5 * + 12 4 / - = 14`

```
public double evaluatePostfix(Queue<String> expression) {
    Stack<Double> valueStack = new LinkedStack<>();
        for (String token : expression) {
            if (isOperator(token)) {
                double rightValue = valueStack.pop();
                double leftValue = valueStack.pop();
                valueStack.push(evaluate(token, leftValue, rightValue));
            } else if (isValue(token)) {
                valueStack.push(Double.parseDouble(token));
            }
        }
        return valueStack.peek();
}
```

queue of expression tokens





operator evaluation

stack of values

evaluatePostfix

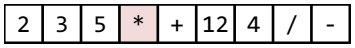
Enter an arithmetic expression: `2 3 5 * + 12 4 / -`

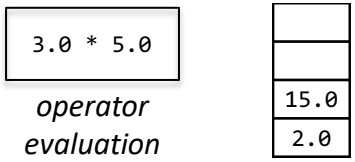
Mode: Prefix | Infix | **Postfix**

`2 3 5 * + 12 4 / - = 14`

```
public double evaluatePostfix(Queue<String> expression) {
    Stack<Double> valueStack = new LinkedStack<>();
        for (String token : expression) {
            if (isOperator(token)) {
                double rightValue = valueStack.pop();
                double leftValue = valueStack.pop();
                valueStack.push(evaluate(token, leftValue, rightValue));
            } else if (isValue(token)) {
                valueStack.push(Double.parseDouble(token));
            }
        }
        return valueStack.peek();
}
```

queue of expression tokens





operator evaluation

stack of values

evaluatePostfix

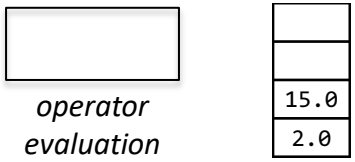
Enter an arithmetic expression: `2 3 5 * + 12 4 / -`

Mode: Prefix | Infix | **Postfix**

`2 3 5 * + 12 4 / - = 14`

```
public double evaluatePostfix(Queue<String> expression) {
    Stack<Double> valueStack = new LinkedStack<>();
        for (String token : expression) {
            if (isOperator(token)) {
                double rightValue = valueStack.pop();
                double leftValue = valueStack.pop();
                valueStack.push(evaluate(token, leftValue, rightValue));
            } else if (isValue(token)) {
                valueStack.push(Double.parseDouble(token));
            }
        }
        return valueStack.peek();
}
```

queue of expression tokens



operator evaluation

stack of values

evaluatePostfix

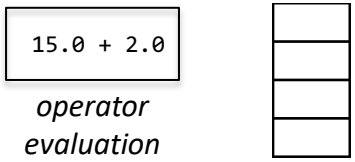
Enter an arithmetic expression: `2 3 5 * + 12 4 / -`

Mode: Prefix | Infix | **Postfix**

`2 3 5 * + 12 4 / - = 14`

```
public double evaluatePostfix(Queue<String> expression) {
    Stack<Double> valueStack = new LinkedStack<>();
        for (String token : expression) {
            if (isOperator(token)) {
                double rightValue = valueStack.pop();
                double leftValue = valueStack.pop();
                valueStack.push(evaluate(token, leftValue, rightValue));
            } else if (isValue(token)) {
                valueStack.push(Double.parseDouble(token));
            }
        }
        return valueStack.peek();
}
```

queue of expression tokens



operator evaluation

stack of values

evaluatePostfix

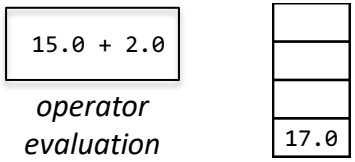
Enter an arithmetic expression: `2 3 5 * + 12 4 / -`

Mode: Prefix | Infix | **Postfix**

`2 3 5 * + 12 4 / - = 14`

```
public double evaluatePostfix(Queue<String> expression) {
    Stack<Double> valueStack = new LinkedStack<>();
        for (String token : expression) {
            if (isOperator(token)) {
                double rightValue = valueStack.pop();
                double leftValue = valueStack.pop();
                valueStack.push(evaluate(token, leftValue, rightValue));
            } else if (isValue(token)) {
                valueStack.push(Double.parseDouble(token));
            }
        }
        return valueStack.peek();
}
```

queue of expression tokens



operator evaluation

stack of values

evaluatePostfix

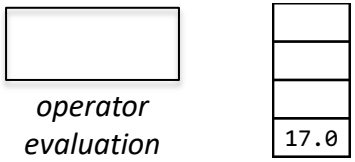
Enter an arithmetic expression: `2 3 5 * + 12 4 / -`

Mode: Prefix | Infix | Postfix

`2 3 5 * + 12 4 / - = 14`

```
public double evaluatePostfix(Queue<String> expression) {
    Stack<Double> valueStack = new LinkedStack<>();
        for (String token : expression) {
            if (isOperator(token)) {
                double rightValue = valueStack.pop();
                double leftValue = valueStack.pop();
                valueStack.push(evaluate(token, leftValue, rightValue));
            } else if (isValue(token)) {
                valueStack.push(Double.parseDouble(token));
            }
        }
        return valueStack.peek();
}
```

queue of expression tokens



operator evaluation

stack of values

evaluatePostfix

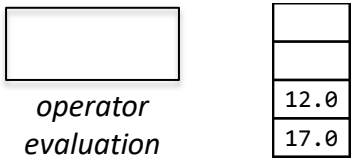
Enter an arithmetic expression: `2 3 5 * + 12 4 / -`

Mode: Prefix | Infix | Postfix

`2 3 5 * + 12 4 / - = 14`

```
public double evaluatePostfix(Queue<String> expression) {
    Stack<Double> valueStack = new LinkedStack<>();
        for (String token : expression) {
            if (isOperator(token)) {
                double rightValue = valueStack.pop();
                double leftValue = valueStack.pop();
                valueStack.push(evaluate(token, leftValue, rightValue));
            } else if (isValue(token)) {
                valueStack.push(Double.parseDouble(token));
            }
        }
        return valueStack.peek();
}
```

queue of expression tokens



operator evaluation

stack of values

evaluatePostfix

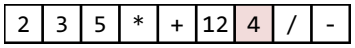
Enter an arithmetic expression: `2 3 5 * + 12 4 / -`

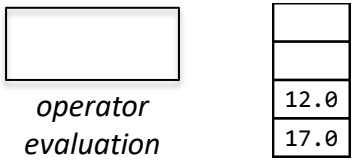
Mode: Prefix | Infix | **Postfix**

`2 3 5 * + 12 4 / - = 14`

```
public double evaluatePostfix(Queue<String> expression) {
    Stack<Double> valueStack = new LinkedStack<>();
        for (String token : expression) {
            if (isOperator(token)) {
                double rightValue = valueStack.pop();
                double leftValue = valueStack.pop();
                valueStack.push(evaluate(token, leftValue, rightValue));
            } else if (isValue(token)) {
                valueStack.push(Double.parseDouble(token));
            }
        }
        return valueStack.peek();
}
```

queue of expression tokens





operator evaluation

stack of values

evaluatePostfix

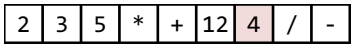
Enter an arithmetic expression: `2 3 5 * + 12 4 / -`

Mode: Prefix | Infix | **Postfix**

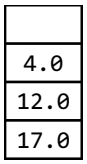
`2 3 5 * + 12 4 / - = 14`

```
public double evaluatePostfix(Queue<String> expression) {
    Stack<Double> valueStack = new LinkedStack<>();
        for (String token : expression) {
            if (isOperator(token)) {
                double rightValue = valueStack.pop();
                double leftValue = valueStack.pop();
                valueStack.push(evaluate(token, leftValue, rightValue));
            } else if (isValue(token)) {
                valueStack.push(Double.parseDouble(token));
            }
        }
        return valueStack.peek();
}
```

queue of expression tokens



operator evaluation



stack of values

evaluatePostfix

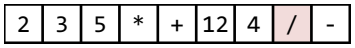
Enter an arithmetic expression: `2 3 5 * + 12 4 / -`

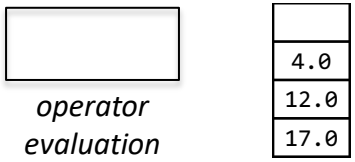
Mode: Prefix | Infix | **Postfix**

`2 3 5 * + 12 4 / - = 14`

```
public double evaluatePostfix(Queue<String> expression) {
    Stack<Double> valueStack = new LinkedStack<>();
        for (String token : expression) {
            if (isOperator(token)) {
                double rightValue = valueStack.pop();
                double leftValue = valueStack.pop();
                valueStack.push(evaluate(token, leftValue, rightValue));
            } else if (isValue(token)) {
                valueStack.push(Double.parseDouble(token));
            }
        }
        return valueStack.peek();
}
```

queue of expression tokens





operator evaluation

stack of values

evaluatePostfix

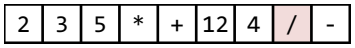
Enter an arithmetic expression: `2 3 5 * + 12 4 / -`

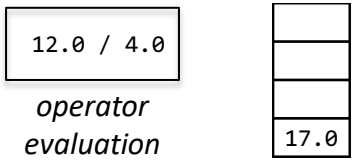
Mode: Prefix | Infix | **Postfix**

`2 3 5 * + 12 4 / - = 14`

```
public double evaluatePostfix(Queue<String> expression) {
    Stack<Double> valueStack = new LinkedStack<>();
        for (String token : expression) {
            if (isOperator(token)) {
                double rightValue = valueStack.pop();
                double leftValue = valueStack.pop();
                valueStack.push(evaluate(token, leftValue, rightValue));
            } else if (isValue(token)) {
                valueStack.push(Double.parseDouble(token));
            }
        }
        return valueStack.peek();
}
```

queue of expression tokens





operator evaluation

stack of values

evaluatePostfix

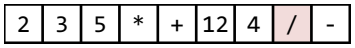
Enter an arithmetic expression: `2 3 5 * + 12 4 / -`

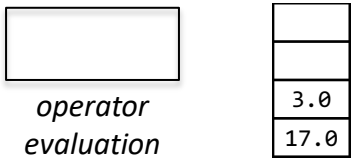
Mode: Prefix | Infix | **Postfix**

`2 3 5 * + 12 4 / - = 14`

```
public double evaluatePostfix(Queue<String> expression) {
    Stack<Double> valueStack = new LinkedStack<>();
        for (String token : expression) {
            if (isOperator(token)) {
                double rightValue = valueStack.pop();
                double leftValue = valueStack.pop();
                valueStack.push(evaluate(token, leftValue, rightValue));
            } else if (isValue(token)) {
                valueStack.push(Double.parseDouble(token));
            }
        }
        return valueStack.peek();
}
```

queue of expression tokens





operator evaluation

stack of values

evaluatePostfix

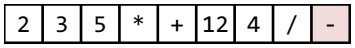
Enter an arithmetic expression: `2 3 5 * + 12 4 / -`

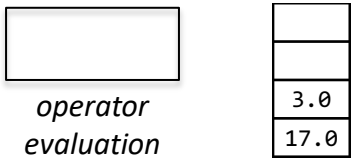
Mode: Prefix | Infix | **Postfix**

`2 3 5 * + 12 4 / - = 14`

```
public double evaluatePostfix(Queue<String> expression) {
    Stack<Double> valueStack = new LinkedStack<>();
        for (String token : expression) {
            if (isOperator(token)) {
                double rightValue = valueStack.pop();
                double leftValue = valueStack.pop();
                valueStack.push(evaluate(token, leftValue, rightValue));
            } else if (isValue(token)) {
                valueStack.push(Double.parseDouble(token));
            }
        }
        return valueStack.peek();
}
```

queue of expression tokens

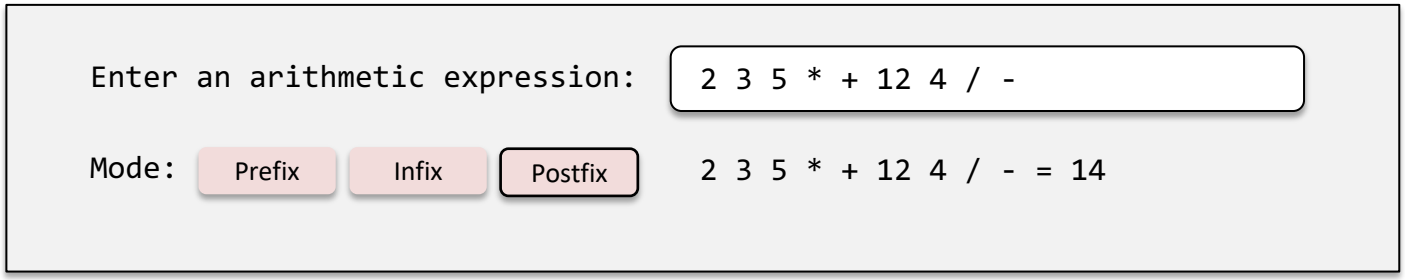




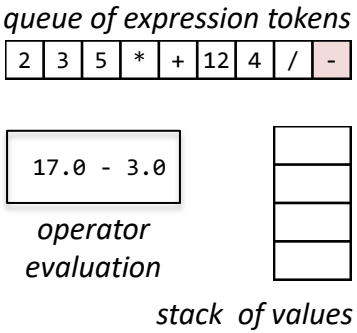
operator evaluation

stack of values

evaluatePostfix




```
public double evaluatePostfix(Queue<String> expression) {
    Stack<Double> valueStack = new LinkedStack<>();
        for (String token : expression) {
            if (isOperator(token)) {
                double rightValue = valueStack.pop();
                double leftValue = valueStack.pop();
                valueStack.push(evaluate(token, leftValue, rightValue));
            } else if (isValue(token)) {
                valueStack.push(Double.parseDouble(token));
            }
        }
        return valueStack.peek();
}
```



evaluatePostfix

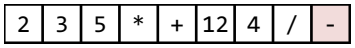
Enter an arithmetic expression: `2 3 5 * + 12 4 / -`

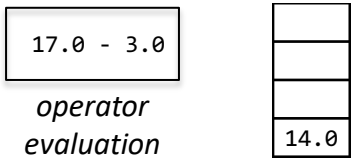
Mode: Prefix | Infix | **Postfix**

`2 3 5 * + 12 4 / - = 14`

```
public double evaluatePostfix(Queue<String> expression) {
    Stack<Double> valueStack = new LinkedStack<>();
        for (String token : expression) {
            if (isOperator(token)) {
                double rightValue = valueStack.pop();
                double leftValue = valueStack.pop();
                valueStack.push(evaluate(token, leftValue, rightValue));
            } else if (isValue(token)) {
                valueStack.push(Double.parseDouble(token));
            }
        }
        return valueStack.peek();
}
```

queue of expression tokens





operator evaluation

stack of values

evaluatePostfix

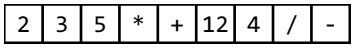
Enter an arithmetic expression: `2 3 5 * + 12 4 / -`

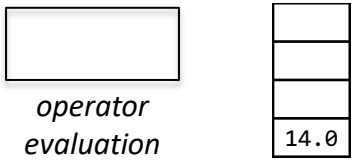
Mode: Prefix | Infix | **Postfix**

`2 3 5 * + 12 4 / - = 14`

```
public double evaluatePostfix(Queue<String> expression) {
    Stack<Double> valueStack = new LinkedStack<>();
        for (String token : expression) {
            if (isOperator(token)) {
                double rightValue = valueStack.pop();
                double leftValue = valueStack.pop();
                valueStack.push(evaluate(token, leftValue, rightValue));
            } else if (isValue(token)) {
                valueStack.push(Double.parseDouble(token));
            }
        }
        return valueStack.peek();
}
```

queue of expression tokens





operator evaluation

stack of values