Data Structures & Algorithms

Adil M. Khan
Professor of Computer Science
Innopolis University

Stacks & Queues

Generics

- We should be able to use collections for any type of data
- A specific Java mechanism known as Generics, also known as parameterized types, enables this capability
- The notation <Item> after the class name

Array List without Generics

```
public class ArrayList implements List{

    ….
    datatype [ ] data;

    ….

}
```

Array List with Generics

```
public class ArrayList<E> implements List<E>{

    ….

        E [ ] data;

        ….

}
```

Array List with Generics

```
ArrayList<String> stringList = new ArrayList<String>();

stringList.add("Kazan");
```

```
ArrayList<Date> dateList = new ArrayList<Date>();

dateList.add(new Date(12,31,1999));
```

Stacks

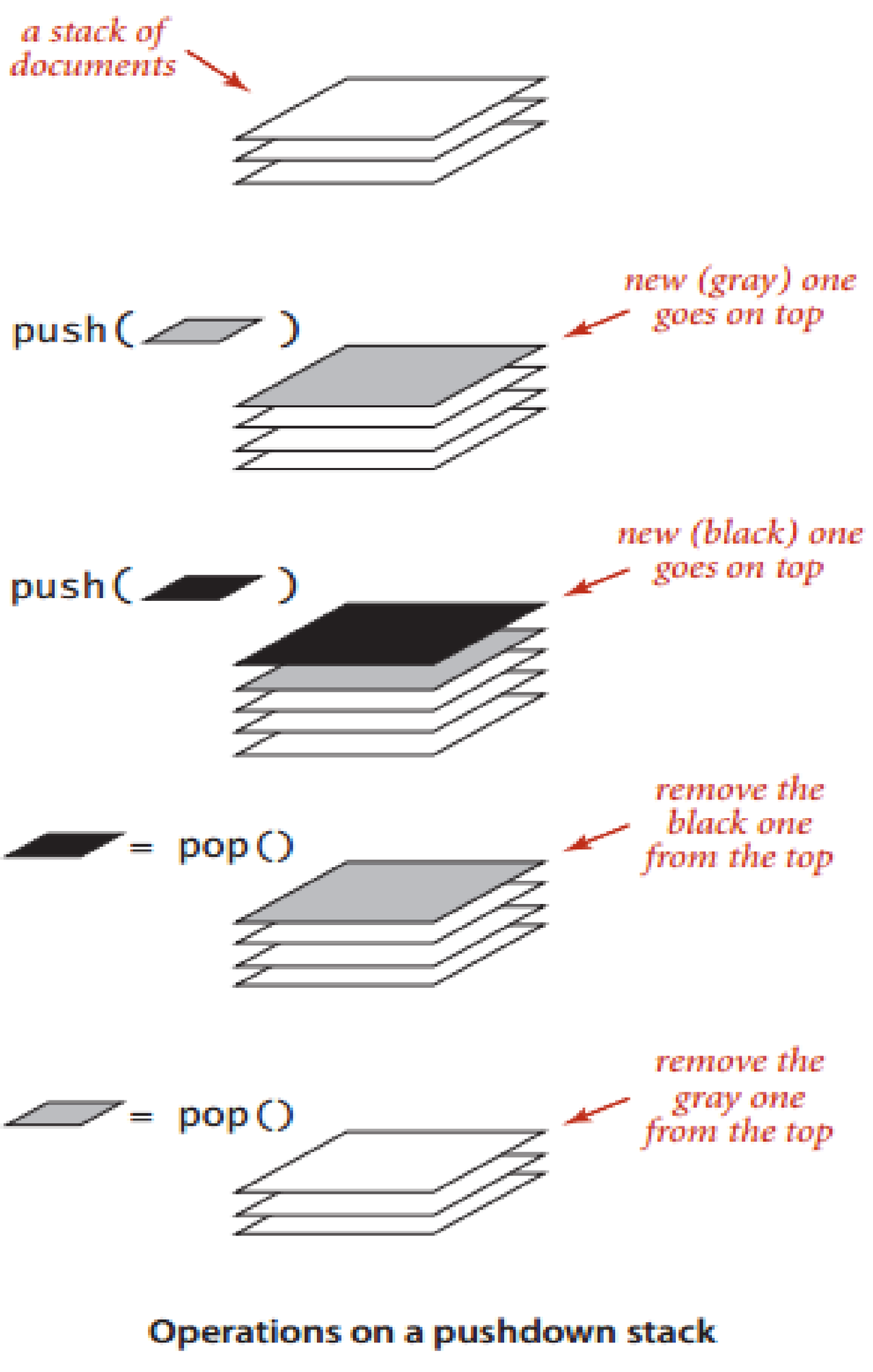
Stacks

- A special kind of list
  - Addition and Removal takes place only at one end, called the top
    - the last element added, is always the first one to be deleted
- So, stack is a LIFO sequence

Stacks

- Mail in a pile on your desk
- Hyperlinks in your browser
- Method calls

Stacks



Operations on a pushdown stack

Stacks

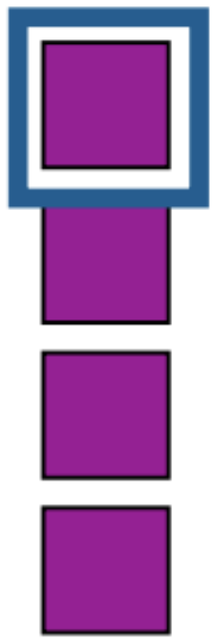
- Iterating through the items in a stack with the *foreach* construct, the items are processed in the reverse order in which they were added.

```
public class Reverse
{
   public static void main(String[] args)
   {
      Stack<Integer> stack;
      stack = new Stack<Integer>();
      while (!StdIn.isEmpty())
         stack.push(StdIn.readInt());

      for (int i : stack)
         StdOut.println(i);
   }
}
```

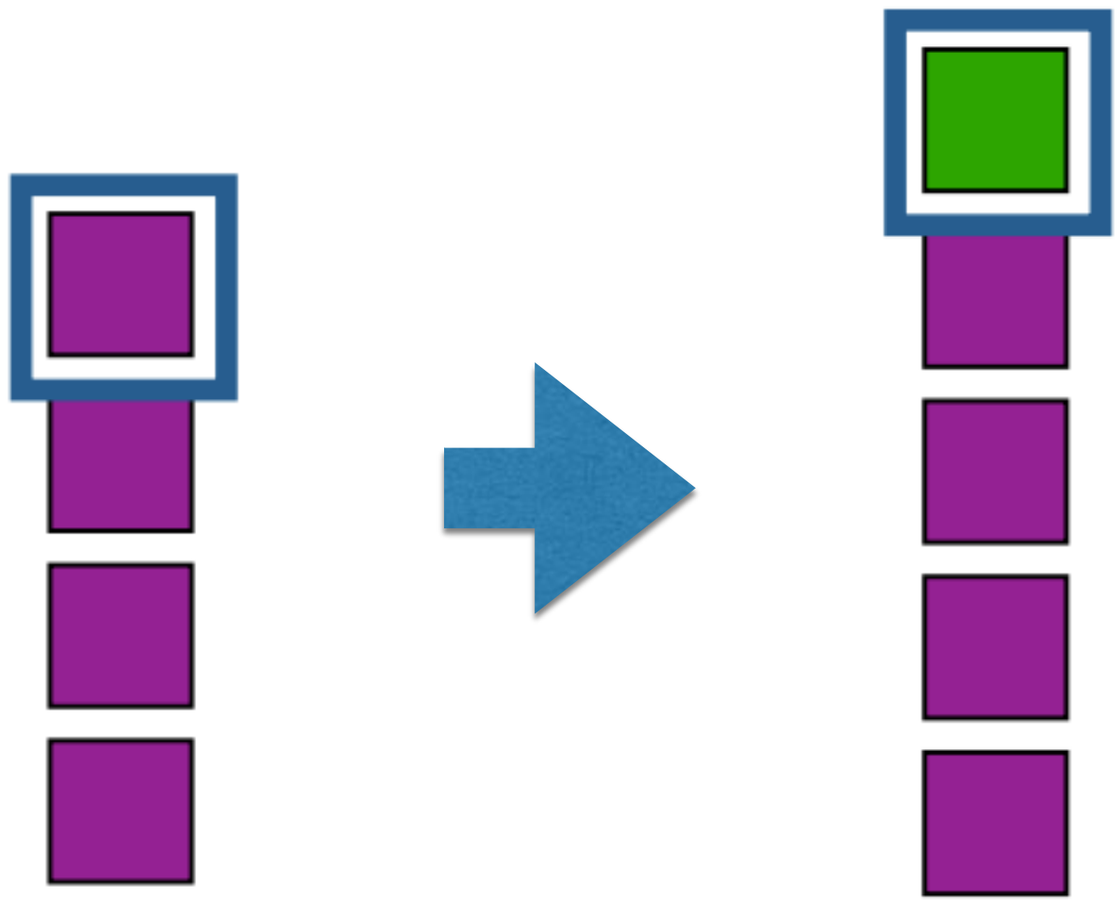
Stack Operations

- size: returns the number of items in the stack
- isEmpty: returns whether stacks has no items
- top: returns the item at the top (without removing it)



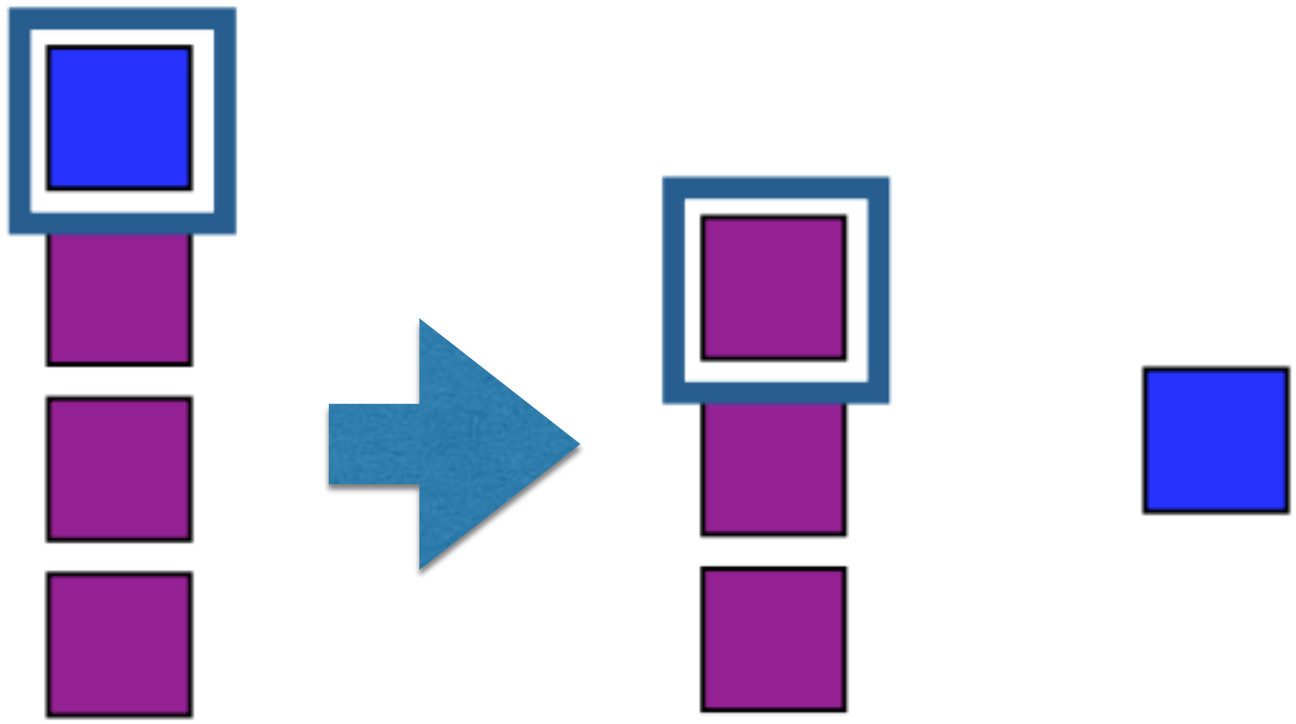
Stack Operatios

- push(x): insert an item x at the top of the stack



Stack Operations

- pop: remove the item at the top



Stack Implementation

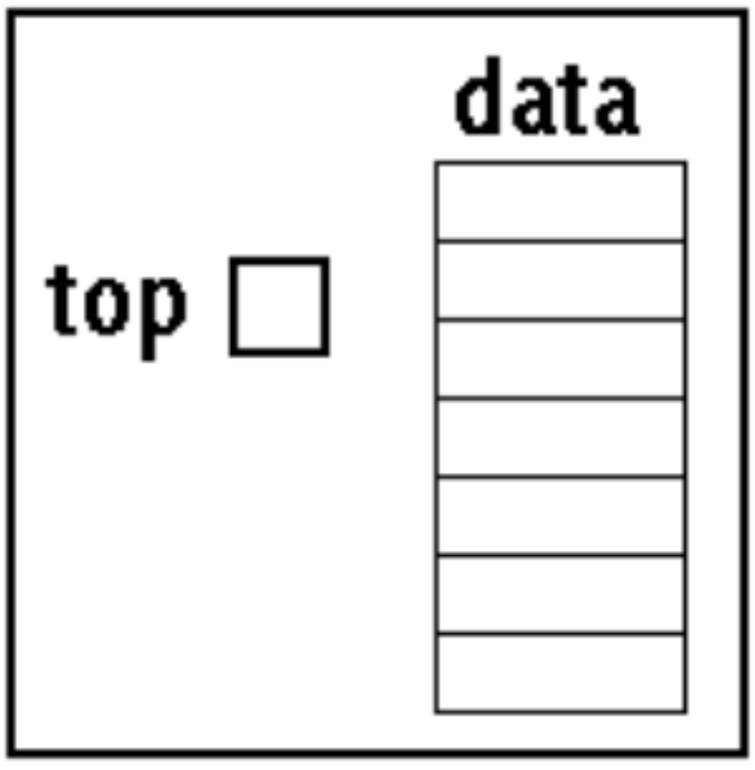
- Dedicated Implementation, OR
- All the operations can be directly implemented using the LIST ADT (as a wrapper around a built-in list object)

Stack ADT

```
public interface Stack<E> {
  int size();
  boolean isEmpty();
  E top();
  void push(E element);
  E pop();
}
```

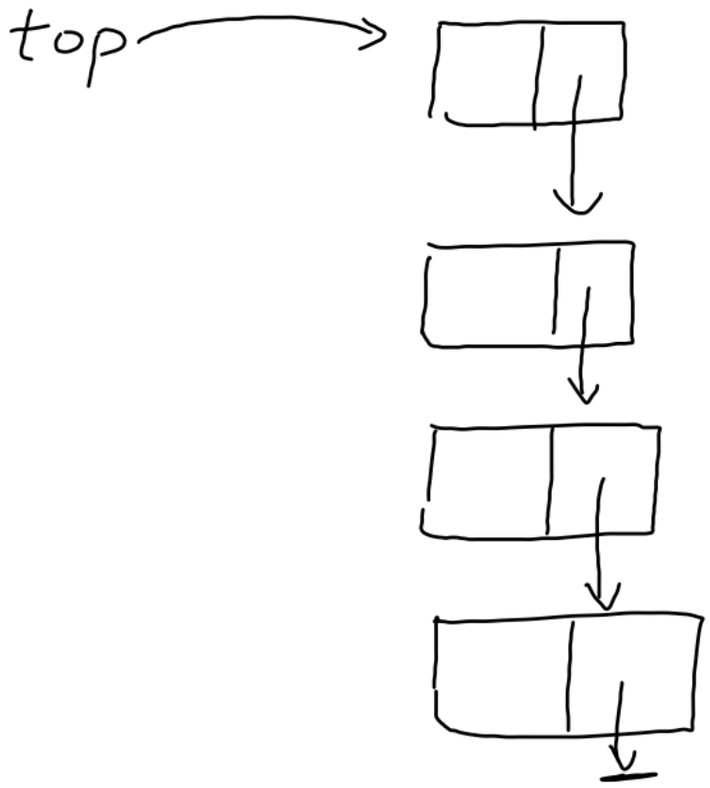
Implementation

1. Array-based



Implementation

2. Linked Implementation



ArrayStack

```
public class ArrayStack<E> implements Stack<E> {
  /** Default array capacity. */
  public static final int CAPACITY=1000;

  /** Generic array used for storage of stack elements. */
  private E[] data;

/** Index of the top element of the stack in the array. */
private int t = -1;
```

ArrayStack

```
/** Constructs an empty stack using the default array capacity. */
public ArrayStack() { this(CAPACITY); }

/**
 * Constructs and empty stack with the given array capacity.
 * @param capacity length of the underlying array
 */
@SuppressWarnings({"unchecked"})
public ArrayStack(int capacity) {
  data = (E[]) new Object[capacity];
}
```

ArrayStack

```
/**
  * Returns the number of elements in the stack.
  * @return number of elements in the stack
  */
 @Override
public int size() { return (t + 1); }

/**
 * Tests whether the stack is empty.
 * @return true if the stack is empty, false otherwise
 */
@Override
public boolean isEmpty() { return (t == -1); }
```

ArrayStack

```
/**
 * Inserts an element at the top of the stack.
 * @param e   the element to be inserted
 * @throws IllegalStateException if the array storing the elements is full
 */
@Override
public void push(E e) throws IllegalStateException {
  if (size() == data.length) throw new IllegalStateException("Stack is full");
  data[++t] = e;                           // increment t before storing new item
}
```

ArrayStack

```
 /**
    * Returns, but does not remove, the element at the top of the
stack.
    * @return top element in the stack (or null if empty)
    */
   @Override
   public E top() {
     if (isEmpty()) return null;
     return data[t];
   }
```

ArrayStack

```
/**
 * Removes and returns the top element from the stack.
 * @return element removed (or null if empty)
 */
@Override
public E pop() {
  if (isEmpty()) return null;
  E answer = data[t];
  data[t] = null;                    // dereference to help garbage collection
  t--;
  return answer;
}
```

ArrayStack

```
  /**
   * Produces a string representation of the contents of the stack.
   * (ordered from top to bottom). This exists for debugging purposes
only.
   *
   * @return textual representation of the stack
   */
  public String toString() {
    StringBuilder sb = new StringBuilder("(");
    for (int j = t; j >= 0; j--) {
      sb.append(data[j]);
      if (j > 0) sb.append(", ");
    }
    sb.append(")");
    return sb.toString();
  }
```

ArrayStack

```
/** Demonstrates sample usage of a stack. */
  public static void main(String[] args) {
    Stack<Integer> S = new ArrayStack<>();  // contents: ()
    S.push(5);                              // contents: (5)
    S.push(3);                              // contents: (5, 3)
    System.out.println(S.size());           // contents: (5, 3)     outputs 2
    System.out.println(S.pop());            // contents: (5)        outputs 3
    System.out.println(S.isEmpty());        // contents: (5)        outputs false
    System.out.println(S.pop());            // contents: ()         outputs 5
    System.out.println(S.isEmpty());        // contents: ()         outputs true
    System.out.println(S.pop());            // contents: ()         outputs null
    S.push(7);                              // contents: (7)
    S.push(9);                              // contents: (7, 9)
    System.out.println(S.top());            // contents: (7, 9)     outputs 9
    S.push(4);                              // contents: (7, 9, 4)
    System.out.println(S.size());           // contents: (7, 9, 4)  outputs 3
    System.out.println(S.pop());            // contents: (7, 9)     outputs 4
    S.push(6);                              // contents: (7, 9, 6)
    S.push(8);                              // contents: (7, 9, 6, 8)
    System.out.println(S.pop());            // contents: (7, 9, 6)  outputs 8
  }
}
```

ArrayStack

Method	Running Time
size	$O(1)$
isEmpty	$O(1)$
top	$O(1)$
push	$O(1)$
pop	$O(1)$

LinkedStack

```
// Realization of a stack as an adaptation of a SinglyLinkedList.
public class LinkedStack<E> implements Stack<E> {

  /** The primary storage for elements of the stack */
  private SinglyLinkedList<E> list = new SinglyLinkedList<>();

  /** Constructs an initially empty stack. */
  public LinkedStack() { }
```

LinkedStack

```
/**
   * Returns the number of elements in the stack.
   * @return number of elements in the stack
   */
  @Override
public int size() { return list.size(); }

/**
 * Tests whether the stack is empty.
 * @return true if the stack is empty, false otherwise
 */
@Override
public boolean isEmpty() { return list.isEmpty(); }
```

LinkedStack

```
/**
   * Inserts an element at the top of the stack.
   * @param element   the element to be inserted
   */
  @Override
public void push(E element) { list.addFirst(element); }

/**
 * Returns, but does not remove, the element at the top of the stack.
 * @return top element in the stack (or null if empty)
 */
@Override
public E top() { return list.first(); }
```

LinkedStack

```
/**
   * Removes and returns the top element from the stack.
   * @return element removed (or null if empty)
   */
  @Override
  public E pop() { return list.removeFirst(); }

  /** Produces a string representation of the contents of the stack.
   *  (ordered from top to bottom)
   *
   * This exists for debugging purposes only.
   *
   * @return textual representation of the stack
   */
  public String toString() {
    return list.toString();
  }
}
```

LinkedStack

Stack Method	*Singly Linked List Method*
size( )	list.size( )
isEmpty( )	list.isEmpty( )
push(e)	list.addFirst(e)
pop( )	list.removeFirst( )
top( )	list.first( )

Method	**Running Time**
size	$O(1)$
isEmpty	$O(1)$
top	$O(1)$
push	$O(1)$
pop	$O(1)$

Queues

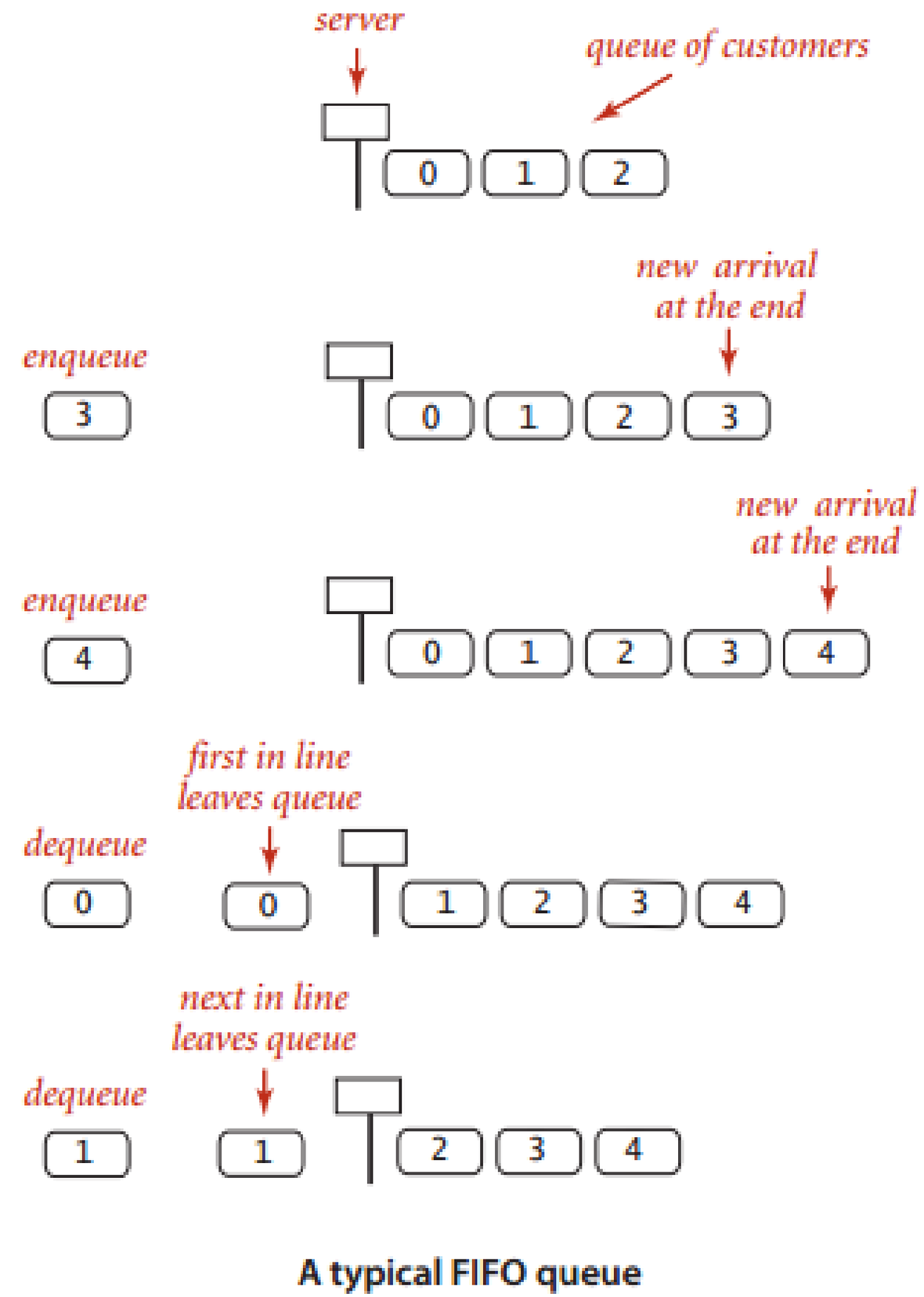
Queues

- Another special kind of list
  - Additions are made at one end, called the tail
  - Removals take place at the other end, called the head
  - the last element added, is always the last one to be deleted
- So, queue is a FIFO sequence

Queues

- Policy of doing tasks in the order they arrive
- People waiting in line at a theater
- Cars waiting in line at a toll booth
- Tasks waiting to be serviced by an application on your computer

Queues

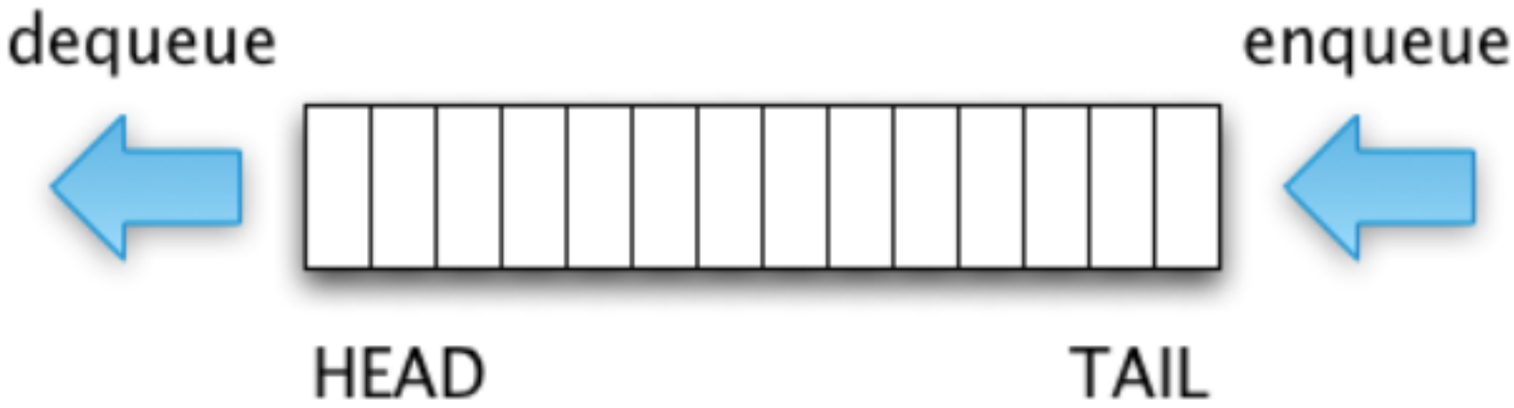


A typical FIFO queue

Queues

- A typical reason to use a queue in an application is to save the item in a collection while at the same time *preserving their relative order*

Queues

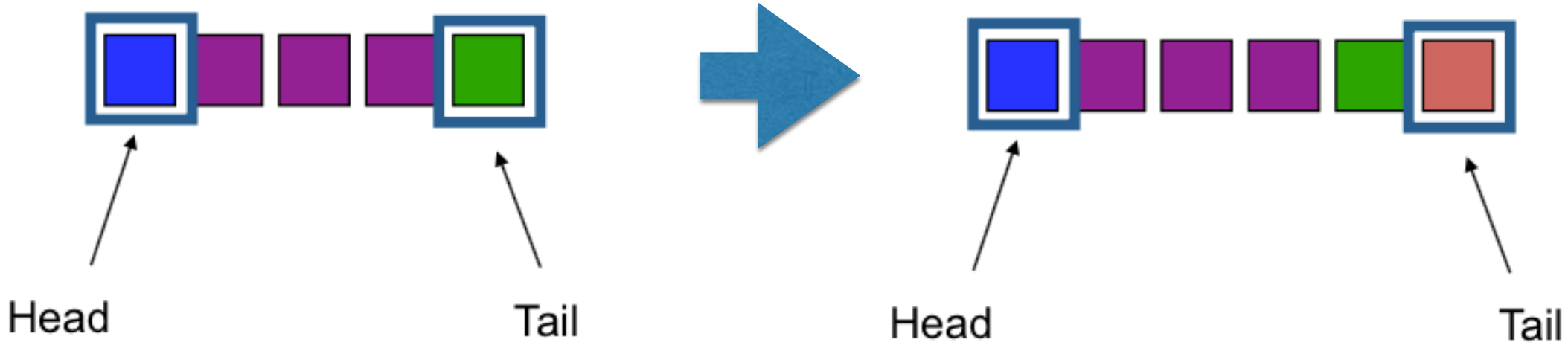


Queue Operations

- size: returns the number of items in the stack
- isEmpty: returns whether stacks has no items
- first: returns the item at the head (without removing it)

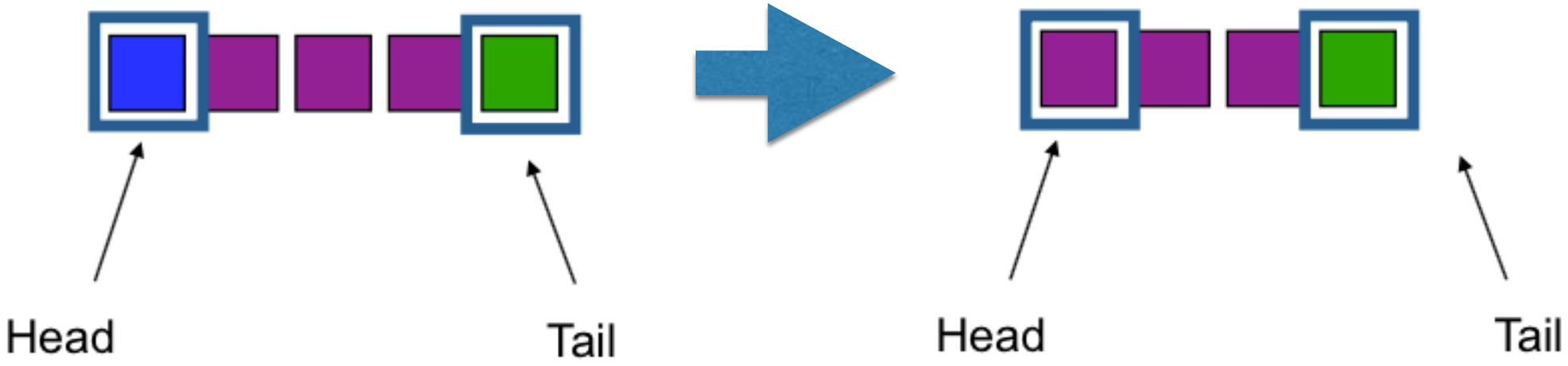
Queue Operatios

- enqueue(x): insert an item x at the tail



Queue Operatios

- dequeue: remove the element from the head



Queue Implementation

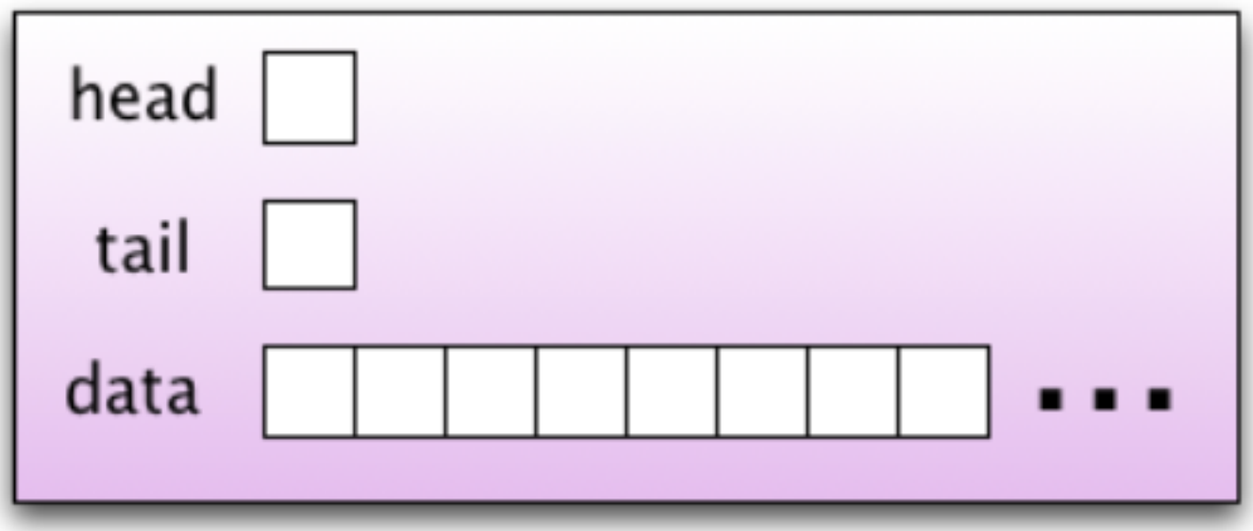
- Dedicated implementation, OR
- All the operations can be directly implemented using the LIST ADT (as a wrapper around a built-in list object)

Queue ADT

```
public interface Queue<E> {
  int size();
  boolean isEmpty();
  E first();
  void enqueue(E e);
  E dequeue();
}
```

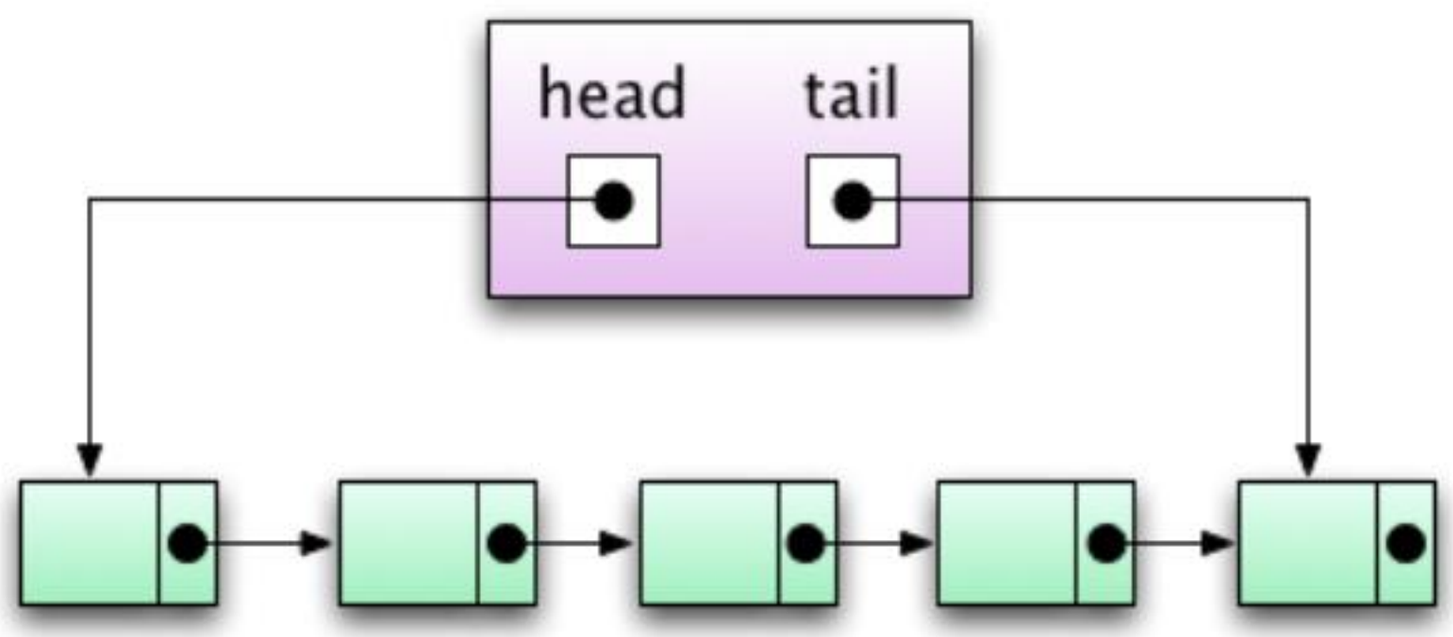
Implementation

1. Array-based



Implementation

2. Linked Implementation



ArrayQueue

```
public class ArrayQueue<E> implements Queue<E> {
  // instance variables
  /** Default array capacity. */
  public static final int CAPACITY = 1000;

  /** Generic array used for storage of queue elements. */
  private E[] data;

  /** Index of the top element of the queue in the array. */
  private int f = 0;
```

ArrayQueue

```
/** Current number of elements in the queue. */
private int sz = 0;

// constructors
/** Constructs an empty queue using the default array capacity. */
public ArrayQueue() {this(CAPACITY);}

/**
 * Constructs and empty queue with the given array capacity.
 * @param capacity length of the underlying array
 */
@SuppressWarnings({"unchecked"})
public ArrayQueue(int capacity) {
  data = (E[]) new Object[capacity];
}
```

ArrayQueue

```
// methods
  /**
   * Returns the number of elements in the queue.
   * @return number of elements in the queue
   */
  @Override
  public int size() { return sz; }

  /** Tests whether the queue is empty. */
  @Override
  public boolean isEmpty() { return (sz == 0); }
```

ArrayQueue

```
/**
 * Inserts an element at the rear of the queue.
 * This method runs in O(1) time.
 * @param e   new element to be inserted
 * @throws IllegalStateException if the array storing the elements is full
 */
@Override
public void enqueue(E e) throws IllegalStateException {
  if (sz == data.length) throw new IllegalStateException("Queue is full");
  int avail = (f + sz) % data.length;   // use modular arithmetic
  data[avail] = e;
  sz++;
}
```

ArrayQueue

```
/**
  * Returns, but does not remove, the first element of the queue.
  * @return the first element of the queue (or null if empty)
  */
 @Override
 public E first() {
   if (isEmpty()) return null;
   return data[f];
 }
```

ArrayQueue

```
/**
 * Removes and returns the first element of the queue.
 * @return element removed (or null if empty)
 */
@Override
public E dequeue() {
  if (isEmpty()) return null;
  E answer = data[f];
  data[f] = null;                    // dereference to help garbage collection
  f = (f + 1) % data.length;
  sz--;
  return answer;
}
```

ArrayQueue

```
/**
   * Returns a string representation of the queue as a list of elements.
   * This method runs in O(n) time, where n is the size of the queue.
   * @return textual representation of the queue.
   */
  public String toString() {
    StringBuilder sb = new StringBuilder("(");
    int k = f;
    for (int j=0; j < sz; j++) {
      if (j > 0)
        sb.append(", ");
      sb.append(data[k]);
      k = (k + 1) % data.length;
    }
    sb.append(")");
    return sb.toString();
  }
}
```

ArrayQueue

Method	Running Time
size	$O(1)$
isEmpty	$O(1)$
first	$O(1)$
enqueue	$O(1)$
dequeue	$O(1)$

LinkedQueue

```
//Realization of a FIFO queue as an adaptation of a SinglyLinkedList.
public class LinkedQueue<E> implements Queue<E> {

  /** The primary storage for elements of the queue */
  private SinglyLinkedList<E> list = new SinglyLinkedList<>();

  /** Constructs an initially empty queue. */
  public LinkedQueue() { }
```

LinkedQueue

```
/**
   * Returns the number of elements in the queue.
   * @return number of elements in the queue
   */
  @Override
public int size() { return list.size(); }

/**
 * Tests whether the queue is empty.
 * @return true if the queue is empty, false otherwise
 */
@Override
public boolean isEmpty() { return list.isEmpty(); }
```

LinkedQueue

```
/**
   * Inserts an element at the rear of the queue.
   * @param element  the element to be inserted
   */
  @Override
public void enqueue(E element) { list.addLast(element); }

/**
 * Returns, but does not remove, the first element of the queue.
 * @return the first element of the queue (or null if empty)
 */
@Override
public E first() { return list.first(); }
```

LinkedQueue

```
/**
 * Removes and returns the first element of the queue.
 * @return element removed (or null if empty)
 */
  @Override
  public E dequeue() { return list.removeFirst(); }

  /** Produces a string representation of the contents of the queue.
   *  (from front to back). This exists for debugging purposes only.
   */
  public String toString() {
    return list.toString();
  }
}
```

LinkedQueue

Method	Running Time
size	$O(1)$
isEmpty	$O(1)$
first	$O(1)$
enqueue	$O(1)$
dequeue	$O(1)$

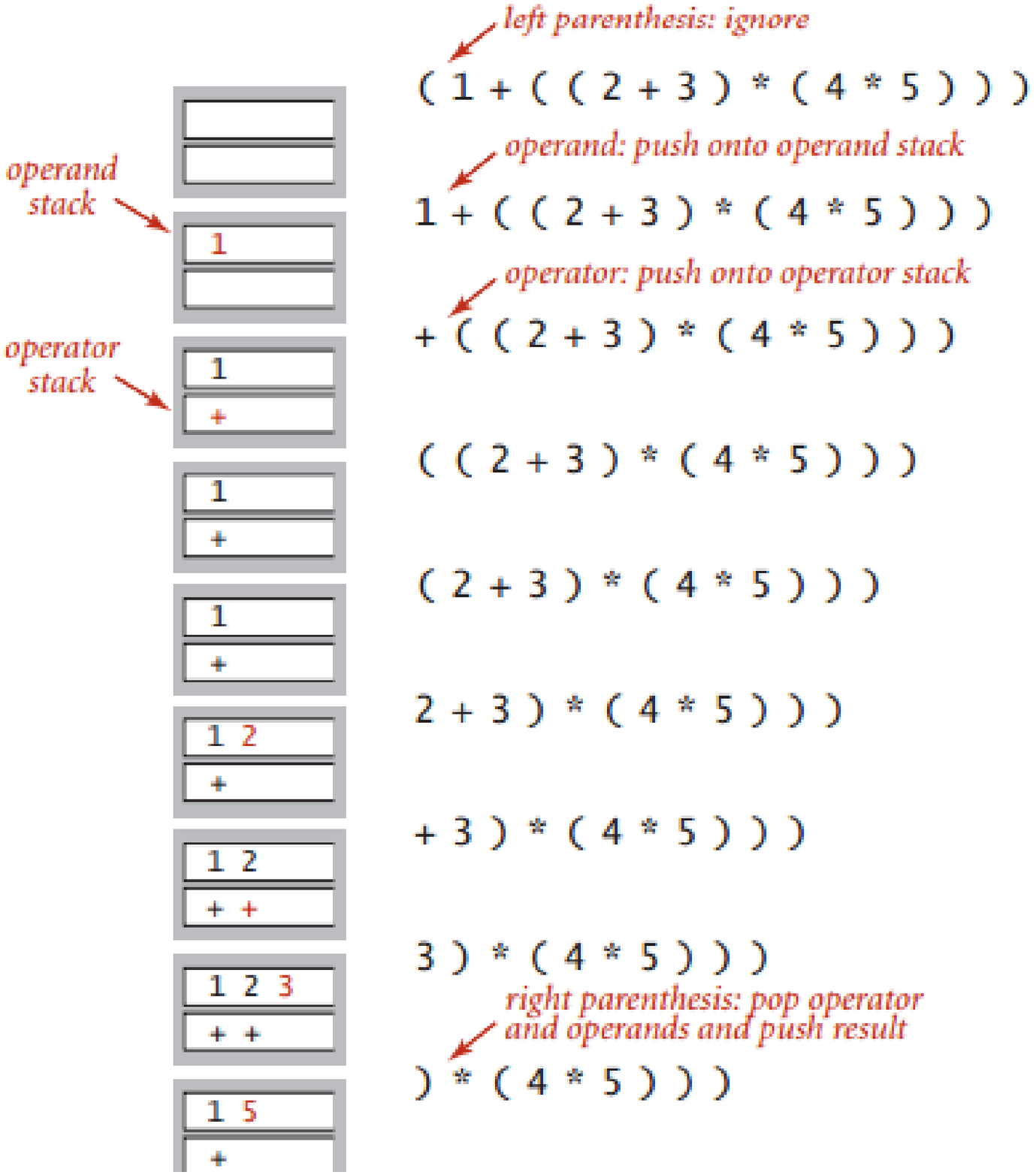
Extras

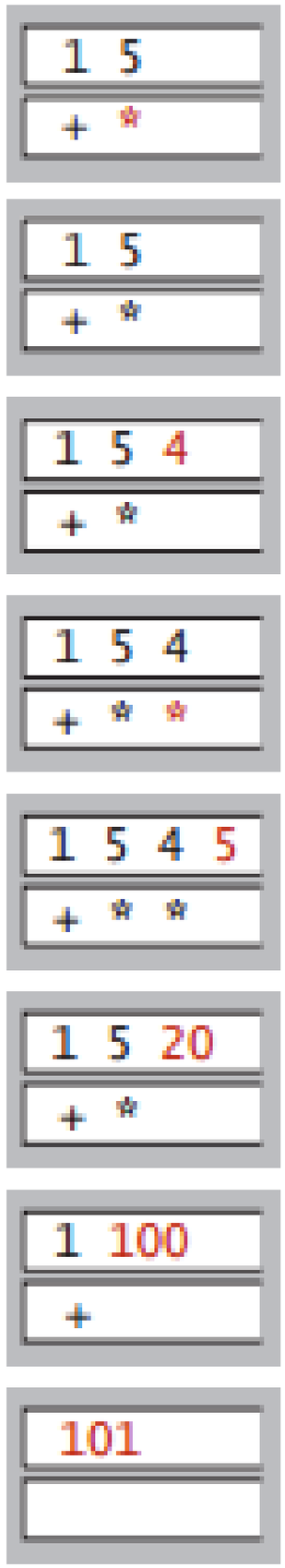
Dijkstra’s Two Stack Algorithm

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

Dijkstra’s Two Stack Algorithm

- Uses two stacks (one for operands and one for operators)
- Proceeding from left to right
  - Push operands onto the operand stack
  - Push operators onto the operator stack
  - Ignore left parentheses
  - One encountering a right parenthesis,
    - Pop an operator, pop the requisite number of operands, apply the operator, and push the result onto the operand stack


operand stack
operator stack
left parenthesis: ignore
operand: push onto operand stack
operator: push onto operator stack
right parenthesis: pop operator and operands and push result
(1 + ((2 + 3) * (4 * 5)))
1 + ((2 + 3) * (4 * 5)))
+ ((2 + 3) * (4 * 5)))
((2 + 3) * (4 * 5)))
(2 + 3) * (4 * 5)))
2 + 3) * (4 * 5)))
+ 3) * (4 * 5)))
3) * (4 * 5)))
) * (4 * 5)))
1
1 +
1 2 +
1 2 + +
1 2 3 + +
1 5 +




```
* ( 4 * 5 ) ) )
( 4 * 5 ) ) )
4 * 5 ) ) )
* 5 ) ) )
5 ) ) )
) ) )
) )
)
```