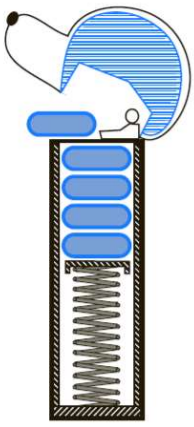
STACKS

1

Stack

- A stack is a collection of objects that are inserted and removed according **to the last-in, first-out (LIFO)** principle.



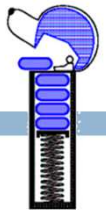
2

Abstract Data Types (ADTs)

- An abstract data type (ADT) is an abstraction of a data structure
- An ADT specifies:
  - Data stored
  - Operations on the data
  - Error conditions associated with operations
- Example: ADT modeling a simple stock trading system
  - The data stored are buy/sell orders
  - The operations supported are
    - order buy(stock, shares, price)
    - order sell(stock, shares, price)
    - void cancel(order)
  - Error conditions:
    - Buy/sell a nonexistent stock
    - Cancel a nonexistent order

3

The Stack ADT



- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out scheme
- Main stack operations:
  - push(object): inserts an element
  - object pop(): removes and returns the last inserted element
- Auxiliary stack operations:
  - object top(): returns the last inserted element without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored

4

Stack Interface in Java

- Java interface corresponding to the Stack ADT
- Assumes null is returned from top() and pop() when stack is empty
- Different from the built-in Java class java.util.Stack

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

5

```
1  /**
2   * A collection of objects that are inserted and removed according to the last-in
3   * first-out principle. Although similar in purpose, this interface differs from
4   * java.util.Stack.
5   *
6   * @author Michael T. Goodrich
7   * @author Roberto Tamassia
8   * @author Michael H. Goldwasser
9   */
10 public interface Stack<E> {
11
12   /**
13    * Returns the number of elements in the stack.
14    * @return number of elements in the stack
15    */
16   int size();
17
18   /**
19    * Tests whether the stack is empty.
20    * @return true if the stack is empty, false otherwise
21    */
22   boolean isEmpty();
23
24   /**
25    * Inserts an element at the top of the stack.
26    * @param e   the element to be inserted
27    */
28   void push(E e);
29
30   /**
31    * Returns, but does not remove, the element at the top of the stack.
32    * @return top element in the stack (or null if empty)
33    */
34   E top();
35
36   /**
37    * Removes and returns the top element from the stack.
38    * @return element removed (or null if empty)
39    */
40   E pop();
41 }
```

6

Exceptions vs. Returning Null

- Attempting the execution of an operation of an ADT may sometimes cause an error condition
- Java supports a general abstraction for errors, called exception
- An exception is said to be "thrown" by an operation that cannot be properly executed
- In our Stack ADT, we do not use exceptions
- Instead, we allow operations pop and top to be performed even if the stack is empty
- For an empty stack, pop and top simply return null

7

A Simple Array-Based Stack

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element

```
Algorithm size()
  return t + 1

Algorithm pop()
  if isEmpty() then
    return null
  else
    t ← t − 1
    return S[t + 1]
```

Representing a stack with an array; the top element is in cell data[t]

8

Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then throw a FullStackException
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT

Algorithm *push*(*o*)
  if $t = S.length - 1$ **then**
    throw *IllegalStateException*
  else
    $t \leftarrow t + 1$
    $S[t] \leftarrow o$

9

Java implementation based on this strategy is given in Code Fragment 6.2 (with Javadoc comments omitted due to space considerations).

```
1  public class ArrayStack<E> implements Stack<E> {
2    public static final int CAPACITY=1000;  // default array capacity
3    private E[ ] data;                      // generic array used for storage
4    private int t = -1;                     // index of the top element in stack
5    public ArrayStack( ) { this(CAPACITY); } // constructs stack with default capacity
6    public ArrayStack(int capacity) {       // constructs stack with given capacity
7      data = (E[ ]) new Object[capacity];   // safe cast; compiler may give warning
8    }
9    public int size( ) { return (t + 1); }
10   public boolean isEmpty( ) { return (t == -1); }
11   public void push(E e) throws IllegalStateException {
12     if (size( ) == data.length) throw new IllegalStateException("Stack is full");
13     data[++t] = e;                        // increment t before storing new item
14   }
15   public E top( ) {
16     if (isEmpty( )) return null;
17     return data[t];
18   }
19   public E pop( ) {
20     if (isEmpty( )) return null;
21     E answer = data[t];
22     data[t] = null;                       // dereference to help garbage collection
23     t--;
24     return answer;
25   }
26 }
```

Code Fragment 6.2: Array-based implementation of the Stack interface.

10

Performance and Limitations

- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- Limitations
  - The maximum size of the stack must be defined during initialization and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific exception

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

11

Garbage Collection in Java

- The reason for returning the cell to a null reference in Code Fragment 6.2 is to assist Java's garbage collection mechanism, which searches memory for objects that are no longer actively referenced and reclaims their space for future use.

12

Example Use in Java

```
public class Tester {
  // … other methods
  public intReverse(Integer a[]) {
     Stack<Integer> s;
     s = new ArrayStack<Integer>();
     … (code to reverse array a) …
  }
```

```
  public floatReverse(Float f[]) {
     Stack<Float> s;
     s = new ArrayStack<Float>();
     … (code to reverse array f) …
  }
```

13

Sample Usage

Method	Return Value	Stack Contents
push(5)	–	(5)
push(3)	–	(5, 3)
size( )	2	(5, 3)
pop( )	3	(5)
isEmpty( )	false	(5)
pop( )	5	( )
isEmpty( )	true	( )
pop( )	null	( )
push(7)	–	(7)
push(9)	–	(7, 9)
top( )	9	(7, 9)
push(4)	–	(7, 9, 4)
size( )	3	(7, 9, 4)
pop( )	4	(7, 9)
push(6)	–	(7, 9, 6)
push(8)	–	(7, 9, 6, 8)
pop( )	8	(7, 9, 6)

```
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
```

14

Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

15

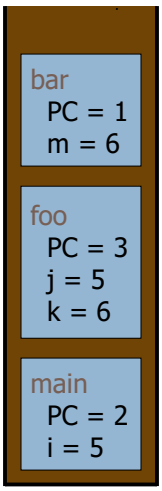
Method Stack in the JVM

- The Java Virtual Machine (JVM) keeps track of the chain of active methods with a stack
- When a method is called, the JVM pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a method ends, its frame is popped from the stack and control is passed to the method on top of the stack
- Allows for recursion

```
main() {
  int i = 5;
  foo(i);
  }

foo(int j) {
  int k;
  k = j+1;
  bar(k);
  }

bar(int m) {
  …
  }
```



16

Ex: Parentheses Matching

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”
  - correct: ( )(( )){([[ ]])}
  - correct: ((( )(( )){([[ ]])}
  - incorrect:)(( )){([[ ]])}
  - incorrect: ({[ ])}
  - incorrect: (

17

Parenthesis Matching (Java)

```
public static boolean isMatched(String expression) {
  final String opening = "({["; // opening delimiters
  final String closing = ")}]"; // respective closing delimiters
  Stack<Character> buffer = new LinkedStack<>( );
  for (char c : expression.toCharArray( )) {
    if (opening.indexOf(c) != −1) // this is a left delimiter
      buffer.push(c);
    else if (closing.indexOf(c) != −1) { // this is a right delimiter
      if (buffer.isEmpty( )) // nothing to match with
        return false;
      if (closing.indexOf(c) != opening.indexOf(buffer.pop( )))
        return false; // mismatched delimiter
      }
    }
  return buffer.isEmpty( ); // were all opening delimiters matched?
}
```

18

Ex: HTML Tag Matching

- For fully-correct HTML, each <name> should pair with a matching </name>

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```

The Little Boat

The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

19

HTML Tag Matching (Java)

```
public static boolean isHTMLMatched(String html) {
  Stack<String> buffer =  new LinkedStack<>( );
  int j = html.indexOf('<'); // find first '<' character (if any)
  while (j != −1) {
    int k = html.indexOf('>', j+1); // find next '>' character
    if (k == −1)
      return false; // invalid tag
    String tag = html.substring(j+1, k); // strip away < >
    if (!tag.startsWith("/")) // this is an opening tag
      buffer.push(tag);
    else { // this is a closing tag
      if (buffer.isEmpty( ))
        return false; // no tag to match
      if (!tag.substring(1).equals(buffer.pop( )))
        return false; // mismatched tag
    }
    j = html.indexOf('<', k+1); // find next '<' character (if any)
  }
  return buffer.isEmpty( ); // were all opening tags matched?
}
```

20

Ex: Evaluating Arithmetic Expressions

14 – 3 * 2 + 7 = (14 – (3 * 2)) + 7

Operator precedence

* has precedence over +/–

Associativity

operators of the same precedence group evaluated from left to right

Example: (x – y) + z rather than x – (y + z)

Idea: push each operator on the stack, but first pop and perform higher and *equal* precedence operations.

21

Algorithm for Evaluating Expressions

Two stacks:

- opStk holds operators
- valStk holds values
- Use $ as special "end of input" token with lowest precedence

Algorithm doOp()

```
x ← valStk.pop();
y ← valStk.pop();
op ← opStk.pop();
valStk.push( y op x )
```

Algorithm repeatOps(refOp):

```
while ( valStk.size() > 1 ∧
        prec(refOp) ≤
        prec(opStk.top())
    doOp()
```

Algorithm EvalExp()

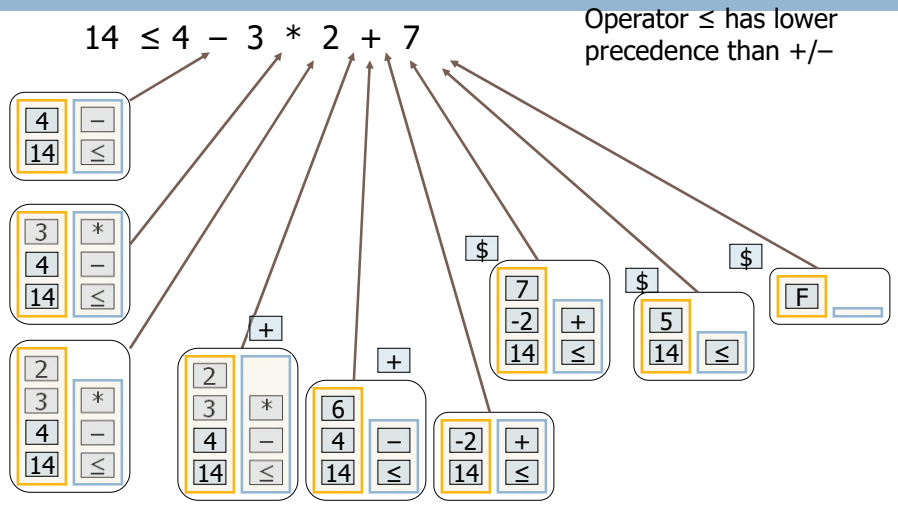
Input: a stream of tokens representing an arithmetic expression (with numbers)

Output: the value of the expression

```
while there's another token z
    if isNumber(z) then
        valStk.push(z)
    else
        repeatOps(z);
        opStk.push(z)
repeatOps($);
return valStk.top()
```

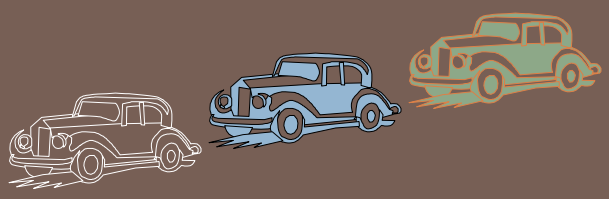
22

Algorithm on an Example Expression



23

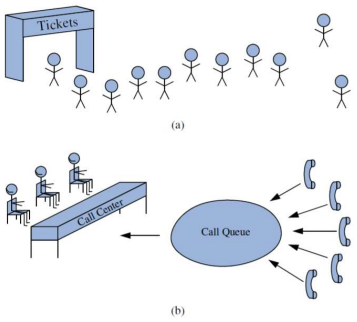
QUEUES



28

The Queue

- The Queue stores arbitrary objects
- Insertions and deletions follow the first-in first-out scheme
- Insertions are at the rear of the queue and removals are at the front of the queue



29

The Queue ADT

- Main queue operations:
  - enqueue(object): inserts an element at the end of the queue
  - object dequeue(): removes and returns the element at the front of the queue
- Auxiliary queue operations:
  - object first(): returns the element at the front without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored
- Boundary cases:
  - Attempting the execution of dequeue or first on an empty queue returns null

30

Java Interface for Queue

```
1  public interface Queue<E> {
2    /** Returns the number of elements in the queue. */
3    int size( );
4    /** Tests whether the queue is empty. */
5    boolean isEmpty( );
6    /** Inserts an element at the rear of the queue. */
7    void enqueue(E e);
8    /** Returns, but does not remove, the first element of the queue (null if empty). */
9    E first( );
10   /** Removes and returns the first element of the queue (null if empty). */
11   E dequeue( );
12 }
```

Code Fragment 6.9: A Queue interface defining the queue ADT, with a standard FIFO protocol for insertions and removals.

31

Example

Operation		*Output*	*Q*
enqueue(5)		–	(5)
enqueue(3)		–	(5, 3)
dequeue()		*5*	(3)
enqueue(7)		–	(3, 7)
dequeue()		*3*	(7)
first()		*7*	(7)
dequeue()		*7*	()
dequeue()		*null*	()
isEmpty()	*true*		()
enqueue(9)		–	(9)
enqueue(7)		–	(9, 7)
size()		*2*	(9, 7)
enqueue(3)		–	(9, 7, 3)
enqueue(5)		–	(9, 7, 3, 5)
dequeue()		*9*	(7, 3, 5)

32

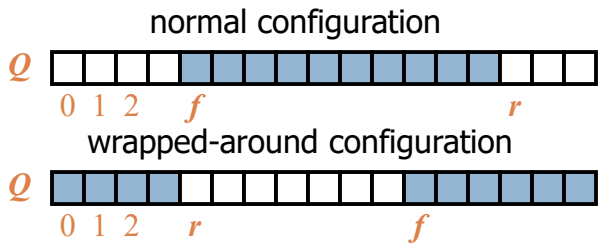
Applications of Queues

- Direct applications
  - Waiting lists, bureaucracy
  - Access to shared resources (e.g., printer)
  - Multiprogramming
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

33

Array-based Queue

- Use an array of size N in a circular fashion
- Two variables keep track of the front and size
  - f index of the front element
  - sz number of stored elements
- When the queue has fewer than N elements, array location $r = (f + sz) \bmod N$ is the first empty slot past the rear of the queue



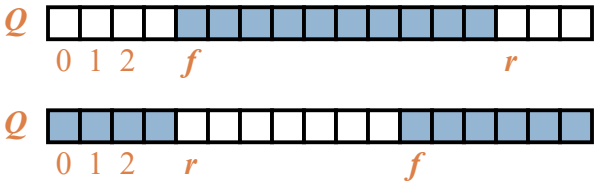
34

Queue Operations

- We use the modulo operator (remainder of division)

```
Algorithm size()
  return sz

Algorithm isEmpty()
  return (sz == 0)
```

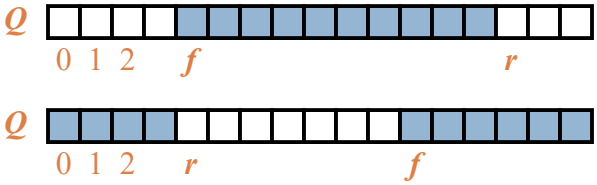


35

Queue Operations (cont.)

- Operation enqueue throws an exception if the array is full
- This exception is implementation-dependent

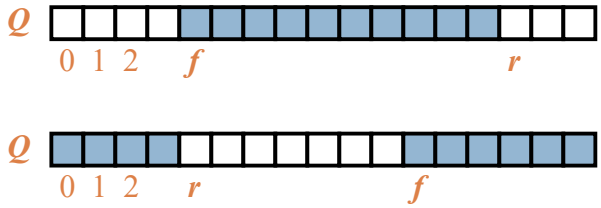
```
Algorithm enqueue(o)
  if size() = N – 1 then
    throw IllegalStateException
  else
    r ← (f + sz) mod N
    Q[r] ← o
    sz ← (sz + 1)
```



36

Queue Operations (cont.)

- Note that operation dequeue returns null if the queue is empty




```
Algorithm dequeue()
  if isEmpty() then
    return null
  else
    o ← Q[f]
    f ← (f + 1) mod N
    sz ← (sz – 1)
    return o
```

37

Queue Interface in Java

- Java interface corresponding to our Queue ADT
- Assumes that first() and dequeue() return null if queue is empty

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

38

Array-based Implementation

```
1  /** Implementation of the queue ADT using a fixed-length array. */
2  public class ArrayQueue<E> implements Queue<E> {
3    // instance variables
4    private E[ ] data;                          // generic array used for storage
5    private int f = 0;                          // index of the front element
6    private int sz = 0;                         // current number of elements
7
8    // constructors
9    public ArrayQueue( ) {this(CAPACITY);} // constructs queue with default capacity
10   public ArrayQueue(int capacity) {          // constructs queue with given capacity
11     data = (E[ ]) new Object[capacity];      // safe cast; compiler may give warning
12   }
13
14   // methods
15   /** Returns the number of elements in the queue. */
16   public int size( ) { return sz; }
17
18   /** Tests whether the queue is empty. */
19   public boolean isEmpty( ) { return (sz == 0); }
20
```

39

Array-based Implementation (2)

```
21   /** Inserts an element at the rear of the queue. */
22   public void enqueue(E e) throws IllegalStateException {
23     if (sz == data.length) throw new IllegalStateException("Queue is full");
24     int avail = (f + sz) % data.length;       // use modular arithmetic
25     data[avail] = e;
26     sz++;
27   }
28
29   /** Returns, but does not remove, the first element of the queue (null if empty). */
30   public E first( ) {
31     if (isEmpty( )) return null;
32     return data[f];
33   }
34
35   /** Removes and returns the first element of the queue (null if empty). */
36   public E dequeue( ) {
37     if (isEmpty( )) return null;
38     E answer = data[f];
39     data[f] = null;                           // dereference to help garbage collection
40     f = (f + 1) % data.length;
41     sz--;
42     return answer;
43   }
```

40

Comparison to java.util.Queue

- Our Queue methods and corresponding methods of java.util.Queue:

Our Queue ADT	Interface java.util.Queue	
	throws exceptions	returns special value
enqueue(e)	add(e)	offer(e)
dequeue( )	remove( )	poll( )
first( )	element( )	peek( )
size( )	size( )	
isEmpty( )	isEmpty( )	

41

Analyzing the Efficiency of an Array-Based Queue

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

42

Implementing a Queue with a Singly Linked List

- Singly linked list to implement the queue ADT while supporting worst-case O(1)-time for all operations

```
1  /** Realization of a FIFO queue as an adaptation of a SinglyLinkedList. */
2  public class LinkedQueue<E> implements Queue<E> {
3    private SinglyLinkedList<E> list = new SinglyLinkedList<>();    // an empty list
4    public LinkedQueue( ) { }                  // new queue relies on the initially empty list
5    public int size( ) { return list.size( ); }
6    public boolean isEmpty( ) { return list.isEmpty( ); }
7    public void enqueue(E element) { list.addLast(element); }
8    public E first( ) { return list.first( ); }
9    public E dequeue( ) { return list.removeFirst( ); }
10 }
```

Code Fragment 6.11: Implementation of a Queue using a SinglyLinkedList.

Application: Round Robin Schedulers

- We can implement a round robin scheduler using a queue Q by repeatedly performing the following steps:
  1. e = Q.dequeue()
  2. Service element e
  3. Q.enqueue(e)