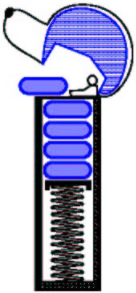
Computer Science 3A - CSC3A10

Lecture 4: Stacks and Queues

Academy of Computer Science and Software Engineering
University of Johannesburg

Stack ADT



Stack ADT Properties

- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out scheme (LIFO)
- Think of a spring-loaded plate dispenser

Stack ADT Properties II

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

	Operation	Output	Stack
1	push(5)	–	(5)
2	push(3)	–	(5, 3)
3	pop()	3	(5)
4	push(7)	–	(5, 7)
5	pop()	7	(5)
6	top()	5	(5)
7	pop()	5	()
8	pop()	''error''	()
9	isEmpty()	**true**	()
10	push(9)	–	(9)
11	push(7))	–	(9, 7)
12	push(3)	–	(9, 7, 3)
13	push(5)	–	(9, 7, 3, 5)
14	size()	4	(9, 7, 3, 5)
15	pop()	5	(9, 7, 3)
16	push(8)	–	(9, 7, 3, 8)
17	pop()	8	(9, 7, 3)

Stack Interface in Java

- Java interface corresponding to our Stack ADT
- Requires the definition of class *EmptyStackException*
- Different from the built-in Java class *java.util.Stack*

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

Stack Interface

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

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

Array-Based Stack

```
Algorithm size()
  return t + 1

Algorithm push(e)
  if size() = N then
    throw FullStackException
  else
    t = t +1
    S[t] = e
```

Array-based Stack

Array-Based Stack II

```
Algorithm pop()
  if isEmpty() then
    throw EmptyStackException
  else
    e = S[t]
    S[t] = null
    t = t − 1
  return e
```

Array-based Stack (Continued)

Performance and Limitations

Performance

- Let n be the number of elements in the stack
- The space used is $O(n)$ (Actually O(N), where N is size of array)
- Each operation runs in time $O(1)$

Limitations:

- The maximum size of the stack must be defined a priori and cannot be changed
- Trying to push a new element into a full stack causes an implementation-specific exception

Implementing a Stack using a Generic Linked List

- Singly Linked list
- Top item always at front of list
- Applications:
  - Matching Parenthesis (We will investigate)
  - Undo operations
  - The stack in the JVM

Parentheses Matching

Each "(", "{", or "[" must be paired with a matching ")", "}", or "["

- correct: `( )(( )){([( )])}`
- correct: `((( )(( )){([( )])}))`
- incorrect: `)(( )){([( )])}`
- incorrect: `({[ ])`
- incorrect: `(`

Parentheses Matching Algorithm I

```
Algorithm ParenMatch(X,n):

Input: An array X of n tokens, each of which is either a grouping
    symbol, a variable, an arithmetic operator, or a number

Output: true if and only if all the grouping symbols in X match

Let S be an empty stack
for i=0 to n−1 do
  if X[i] is an opening grouping symbol then
    S.push(X[i])
  else if X[i] is a closing grouping symbol then
    if S.isEmpty() then
      return false {nothing to match with}
    if S.pop() does not match the type of X[i] then
      return false {wrong type}
```

Parentheses Matching Algorithm II

```
if S.isEmpty() then
  return true {every symbol matched}
else
  return false {some symbols were never matched}
```

Parentheses Matching Algorithm

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.

Arithmetic Expression Evaluation

- Two Stacks:
  - *opStk* holds operators
  - *valStk* holds values
- Use $ as a special "end of input" token with lowest precedence
- Input: a stream of tokens representing an arithmetic expression (with numbers)
- Output: the value of the expression

Arithmetic Expression Evaluation Algorithm I

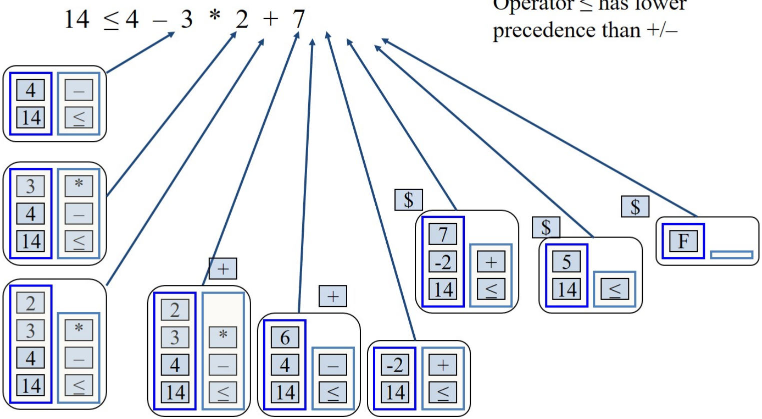
```
Algorithm doOp()
  x = valStk.pop();
  y = valStk.pop();
  op = opStk.pop();
  valStk.push( y op x )

Algorithm repeatOps( refOp ):
  while ( valStk.size() > 1 &&
  prec(refOp) ≤ prec(opStk.top())
  doOp()

Algorithm EvalExp()
  while there's another token z
  if isNumber(z) then
    valStk.push(z)
    else
```

Arithmetic Expression Evaluation Algorithm II

```
        repeatOps(z);
        opStk.push(z)
repeatOps($);
return valStk.top()
```

Evaluating Expressions Algorithm

Arithmetic Expression Evaluation Algorithm III



Operator ≤ has lower precedence than +/–

Queue ADT

Queue ADT Properties

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

Queue ADT Properties II

- Auxiliary queue operations:
  - *object front()*: 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
- Exceptions - Attempting the execution of dequeue or front on an empty queue throws an *EmptyQueueException*

	Operation	Output	Queue
1	enqueue(5)	—	(5)
2	enqueue(3)	—	(5, 3)
3	dequeue()	5	(3)
4	enqueue(7)	—	(3, 7)
5	dequeue()	3	(7)
6	front()	7	(7)
7	dequeue()	7	()
8	dequeue()	''error''	()
9	isEmpty()	**true**	()
10	enqueue(9)	—	(9)
11	enqueue(7)	—	(9, 7)
12	size())	2	(9, 7)
13	enqueue(3)	—	(9, 7, 3)
14	enqueue(5)	—	(9, 7, 3, 5)
15	dequeue()	9	(7, 3, 5)

Queue Interface in Java

- Java interface corresponding to our Queue ADT
- Requires the definition of class *EmptyQueueException*
- No corresponding built-in Java class

```
public interface Queue <E>{
  public int size();
  public boolean isEmpty();
  public E front() throws
      EmptyQueueException;
  public void enqueue(E o);
  public E dequeue() throws
      EmptyQueueException;
}
```

Queue Interface

Applications of Queues

Direct applications

- Waiting lists, bureaucracy
- Access to shared resources (e.g., printer)
- Multiprogramming

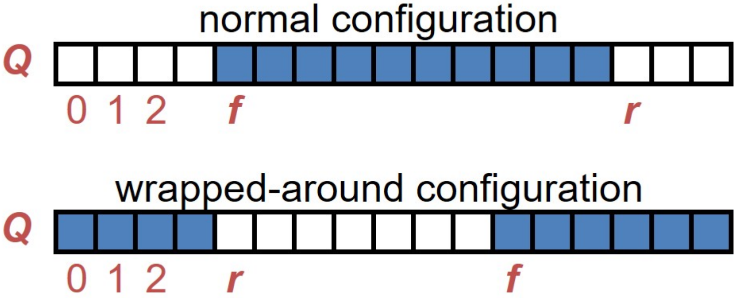
Indirect applications

- Auxiliary data structure for algorithms
- Component of other data structures

Array-Based Queue

- Use an array of size N in a circular fashion
- Two variables keep track of the front and rear
  - f index of the front element
  - r index immediately past the rear element
- Array location r is kept empty

Array-Based Queue II



Queue Operations

- We use the modulo operator (remainder of division)
- Operation enqueue throws an exception if the array is full
- This exception is implementation-dependent
- Operation dequeue throws an exception if the queue is empty
- This exception is specified in the queue ADT

Queue Source Code I

```
Algorithm size()
  return (N − f + r) mod N

Algorithm isEmpty()
  return (f == r)

Algorithm enqueue(e)
  if size() = N − 1 then
    throw FullQueueException
  else
    Q[r] = e
    r = (r + 1) mod N

Algorithm dequeue()
  if isEmpty() then
    throw EmptyQueueException
```

Queue Source Code II

```
  else
    o = Q[f]
    Q[f] = null
    f = (f + 1) mod N
    return o
```

Queue Interface

Double Ended Queues

- Deques support insertion and deletion at the front and at the back
  - *addFirst(e)*
  - *addLast(e)*
  - *removeFirst()*
  - *removeLast()*

Double Ended Queues II

- Supporting Methods
  - *First()*
  - *Last()*
  - *size()*
  - *isEmpty()*
- Running times:
  - size, isEmpty: $O(1)$
  - getFirst, getLast: $O(1)$
  - addFirst, addLast: $O(1)$
  - removeFirst, removeLast: $O(1)$

Double Ended Queues III

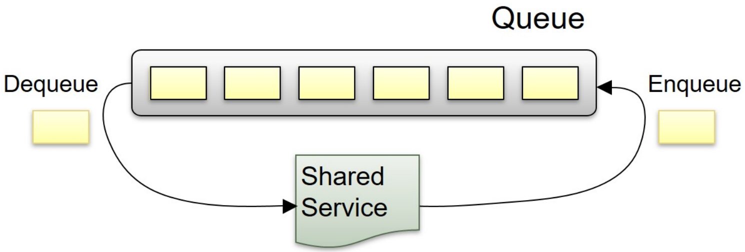
	Operation	Output	Deque
1	`addFirst(3)`	—	(3)
2	`addFirst(5)`	—	(5, 3)
3	`removeFirst()`	5	(3)
4	`addLast(7)`	—	(3, 7)
5	`removeFirst()`	3	(7)
6	`removeLast()`	7	()
7	`removeFirst()`	``error''	()
8	`isEmpty()`	**true**	()

Application: Round Robin Schedulers

We can implement a round robin scheduler using a queue Q by repeatedly performing the following steps:

- $e = Q.dequeue()$
- $Service\ element\ e$
- $Q.enqueue(e)$

Application: Round Robin Schedulers II



Exercises

Reinforcement exercises:

- R-6.1
- R-6.2
- R-6.3
- R-6.7

Creativity exercises:

- C-6.16
- C-6.18
- C-6.27
- C-6.28