Fall 2017

Data Structures Lecture 5

Fang Yu

Software Security Lab.
Dept. Management Information Systems,
National Chengchi University

Announcement

Project Proposal due on Nov. 9

- Remember to **bring a hardcopy of your project proposal to the class next week**
- Also submit your project proposal file to your TAs
- Discuss potential difficulties of your project
- We can select some topics/challenges and explain them in the lab

Announcement

- We will have programming lectures and tests on Nov. 6 in 逸仙樓 5F.
- HWs Review
  - BMI
  - Generic Progression
  - Keyword Counting
  - The Ordered List

Abstract Linear Data Structures

Stacks, Queues, Deques, and Priority Queues

Abstract Data Type (ADT)

- 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
- We have discussed Array ADT and List ADT

Stacks

- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out scheme

Stack Operations

Main operations:

- push(object): inserts an element
- object pop(): removes and returns the last inserted element

Others:

- 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

Stack Interface in Java

```
public interface Stack<E> {

   public int size();

   public boolean isEmpty();

   public E top()
         throws EmptyStackException;

   public void push(E element);

   public E pop()
         throws EmptyStackException;
}
```

- In the Stack ADT, operations pop and top cannot be performed if the stack is empty
- Attempting the execution of pop or top on an empty stack throws an EmptyStackException

Common Applications

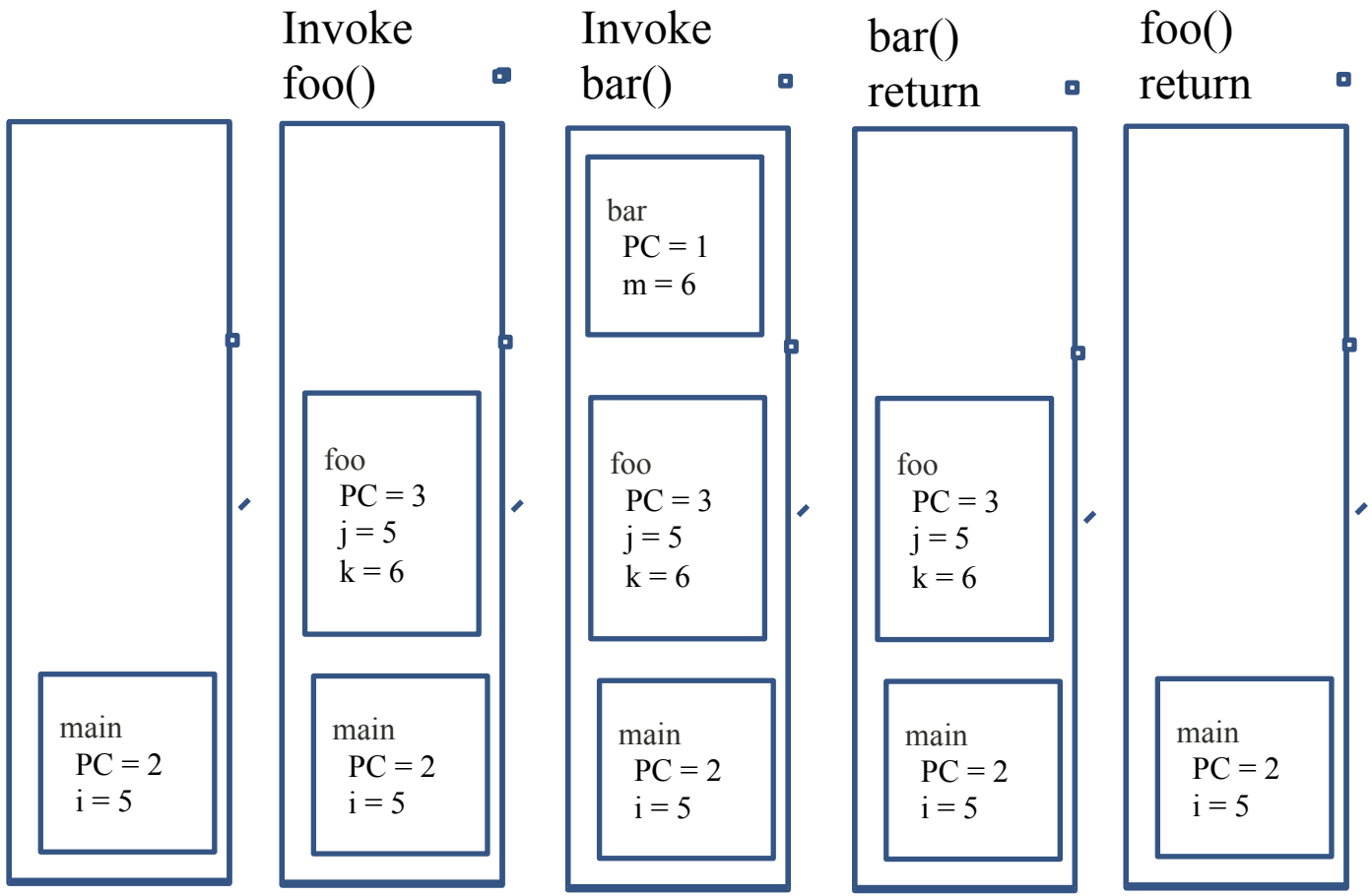
- Page-visited history in a Web browser
  - Yahoo → news → drama
  - http://tw.yahoo.com/
- Undo sequence in a text editor
  - For example, I type something here

Method Call 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

An Example of Method Stack

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

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

bar(int m) {
    …
    }
```



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**
    throw *EmptyStackException*
  else
    $t \leftarrow t - 1$
    return $S[t + 1]$

Array-based Stack

- 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 *FullStackException*
  else
    t ← *t* + 1
    S[*t*] ← *o*

Array-based Stack

- 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 a priori and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific exception
  - Use a growthable array instead

Parenthesis Matching

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

Parenthesis Matching

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

Parentheses Matching

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

- For example: (2+3)*(3+6)
- X is an array of { (, 2, +, 3,), *, (, 3, +, 6,) } and n is 11
- Output: true

Parentheses Matching Algorithm

```
Let S be an empty stack
for i=0 to n-1 do
    if X[i] is an opening grouping symbol then // e.g., (
        S.push(X[i])
    else if X[i] is a closing grouping symbol then // e.g., )
        if S.isEmpty() then
            return false  //nothing to match with
        if S.pop() does not match the type of X[i] then
            return false  //a wrong type
if S.isEmpty() then
    return true //all symbols are matched
else return false //symbols that can not be matched
```

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
    - (x – y) + z rather than x – (y + z)

Evaluating Arithmetic Expressions

- Idea: push each operator on the stack, but first pop and perform higher and *equal* precedence operations.
- Two stacks:
  - opStk holds operators
  - valStk holds values
- To clean up the stack at the end, we use $ as special "end of input" token with lowest precedence

Evaluating Arithmetic Expressions

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) //push number
        else
                repeatOps(z); //calculate the higher precedence  operators
                opStk.push(z); //before push the operator z
repeatOps($); //calculate all the rest operators
return valStk.top()
```

Evaluating Arithmetic Expressions

Algorithm repeatOps(refOp):

while (valStk.size() > 1 ∧

prec(refOp) ≤ prec(opStk.top()) //op in the Stack needs to be done first

doOp()

Algorithm doOp()

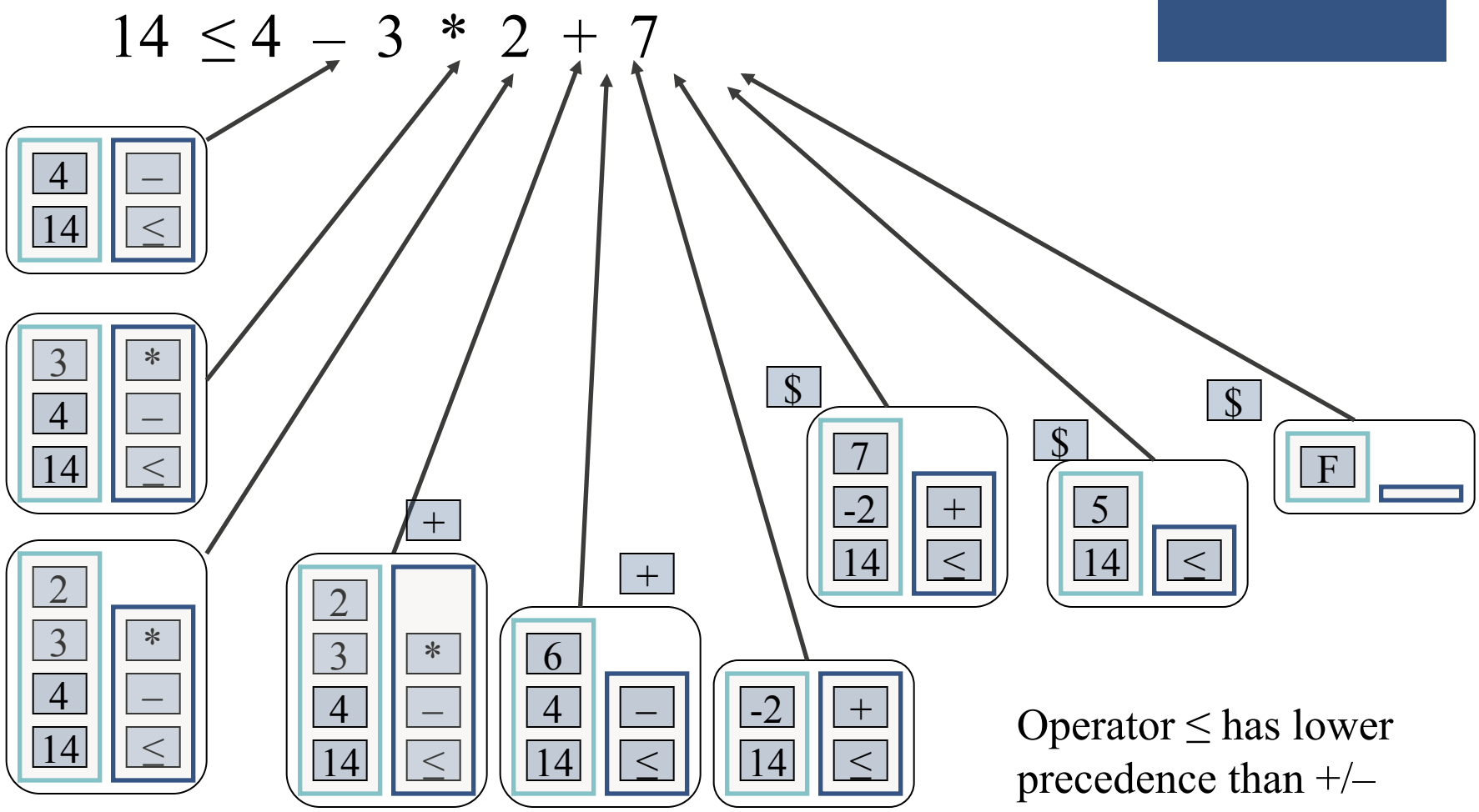
x ← valStk.pop();

y ← valStk.pop();

op ← opStk.pop();

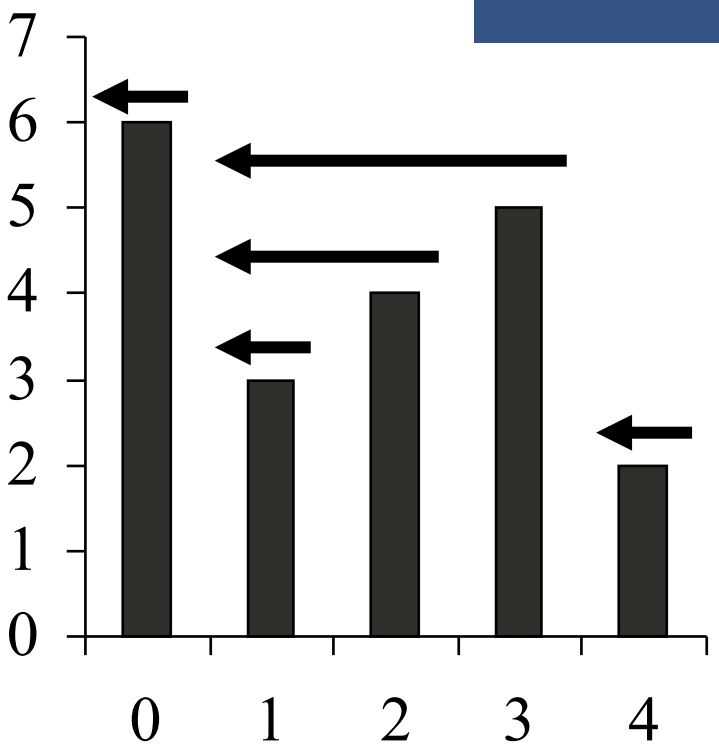
valStk.push(y **op** x)

An Example


14 ≤ 4 – 3 * 2 + 7
4
14
–
≤
3
4
14
*
–
≤
2
3
4
14
*
–
≤
+
2
3
4
14
*
–
≤
+
6
4
14
–
≤
$
-2
14
+
≤
$
7
-2
14
+
≤
$
5
14
≤
F
Operator ≤ has lower precedence than +/–

Computing Spans

- Span: number of equal or smaller consecutive terms (including itself)
- Given an array X, compute the array S, where $S[i]$ is the maximum number of consecutive elements $X[j]$ immediately preceding $X[i]$, where $X[j] \leq X[i]$ and $j \leq i$
- Spans have applications to financial analysis
  - E.g., stock at 52-week high



X	6	3	4	5	2
S	1	1	2	3	1

A Quadratic Algorithm

Algorithm *spans1*(*X*, *n*)	
Input array *X* of *n* integers	
Output array *S* of spans of *X*	#
S ← new array of *n* integers	*n*
for *i* ← 0 **to** *n* − 1 **do**	*n*
s ← 1	*n*
while *s* ≤ *i* ∧ *X*[*i* − *s*] ≤ *X*[*i*]	1 + 2 + …+ (*n* − 1)
s ← *s* + 1	1 + 2 + …+ (*n* − 1)
S[*i*] ← *s*	*n*
return *S*	1

- Algorithm ***spans1*** runs in $O(n^2)$ time

A Linear Algorithm

- Using a stack as an auxiliary data
- We keep in a stack the indices of the elements visible when "looking back"
- We scan the array from left to right
  - Let i be the current index
  - We pop indices from the stack until we find index j such that $X[i] < X[j]$
  - We set $S[i] \leftarrow i - j$
  - We push x onto the stack

A Linear Algorithm

- Each index of the array
  - Is pushed into the stack exactly one
  - Is popped from the stack at most once
- The statements in the while-loop are executed at most n times
- Algorithm ***spans2*** runs in $O(n)$ time

```
Algorithm spans2(X, n)                          #
    S ← new array of n integers                 n
    A ← new empty stack                         1
    for i ← 0 to n − 1 do                       n
        while (¬A.isEmpty() ∧
                X[A.top()] ≤ X[i] ) do          n
            A.pop()                             n
        if A.isEmpty() then                     n
            S[i] ← i + 1                        n
        else
            S[i] ← i − A.top()                  n
        A.push(i)                               n
    return S                                    1
```

Queues

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

Queue Operations

Main 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

Others:

- 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

An Example

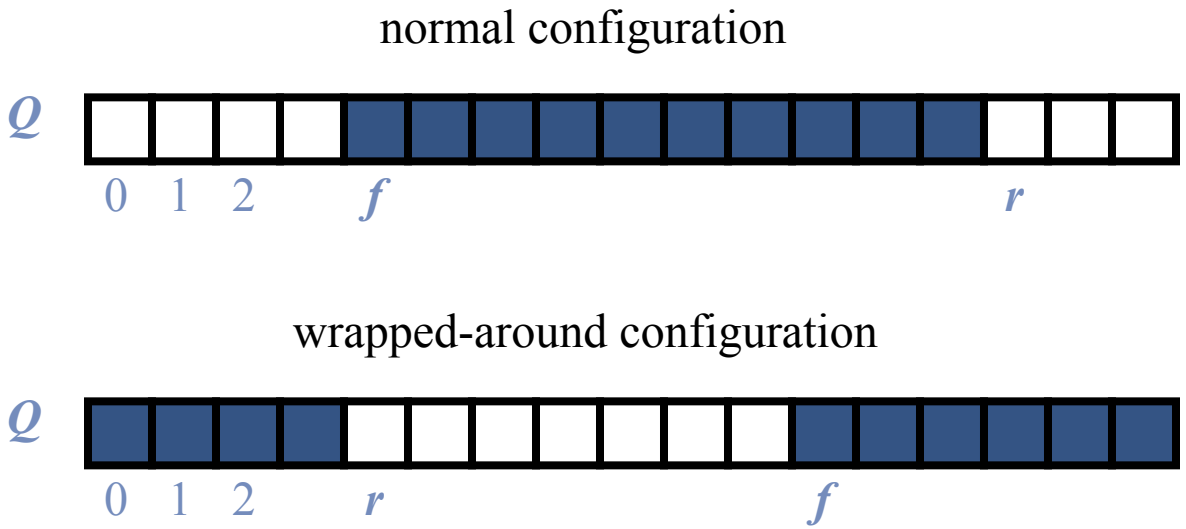
Operation	*Output*	*Q*
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
dequeue()	*5*	(3)
enqueue(7)	–	(3, 7)
dequeue()	*3*	(7)
front()	*7*	(7)
dequeue()	*7*	()
dequeue()	*“error”*	()
isEmpty()		*true* ()
enqueue(9)	–	(9)
enqueue(7)	–	(9, 7)
size()	*2*	(9, 7)
enqueue(3)	–	(9, 7, 3)
enqueue(5)	–	(9, 7, 3, 5)
dequeue()	*9*	(7, 3, 5)

Array-based Queue

- Use an array of size ***N***
- Insert at the front, and remove at the rear
  - size(): return n;
  - isEmpty(): return n==0;
  - enqueue(object): add(n, obeject)
  - dequeue(): remove(0)
- Removing the front is costly in an array, i.e., O(n).

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

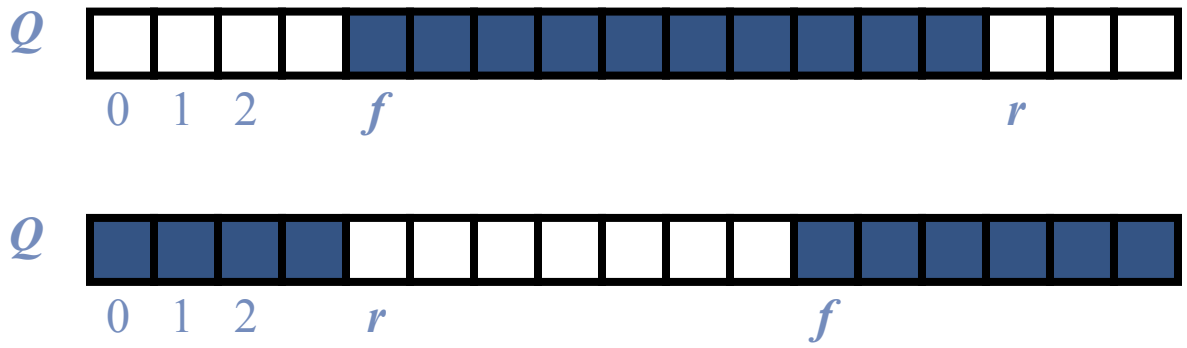


Queue Operations

- We use the modulo operator (remainder of division)

Algorithm ***size***()
    return $(N - f + r) \bmod N$

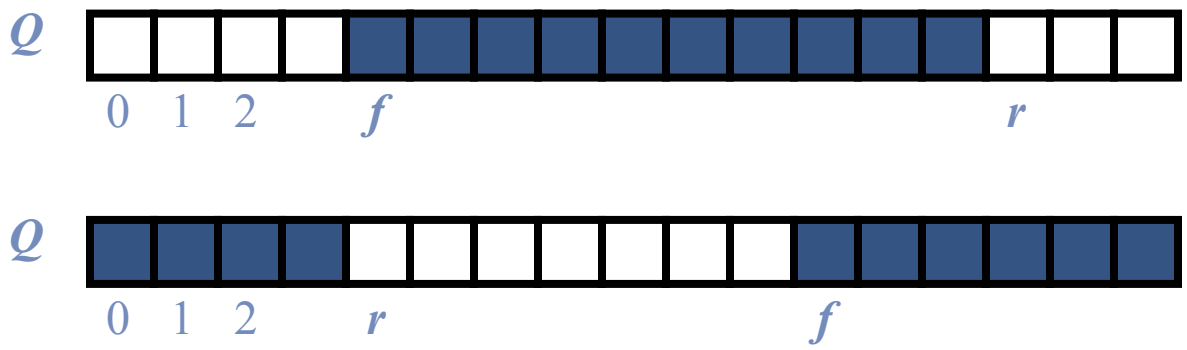
Algorithm ***isEmpty***()
    return $(f = r)$



Queue Operations

- Operation enqueue throws an exception if the array is full

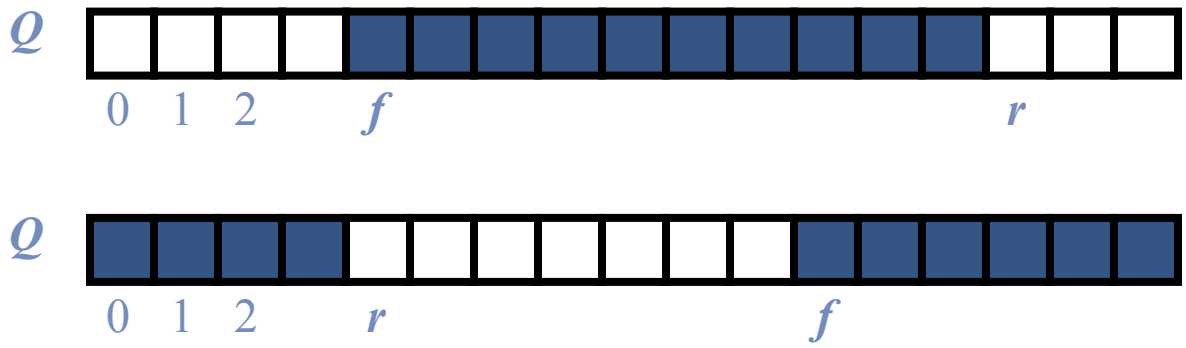
```
Algorithm enqueue(o)
    if size() = N - 1 then
        throw FullQueueException
    else
        Q[r] ← o
        r ← (r + 1) mod N
```



Queue Operations

- Operation dequeue throws an exception if the queue is empty

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



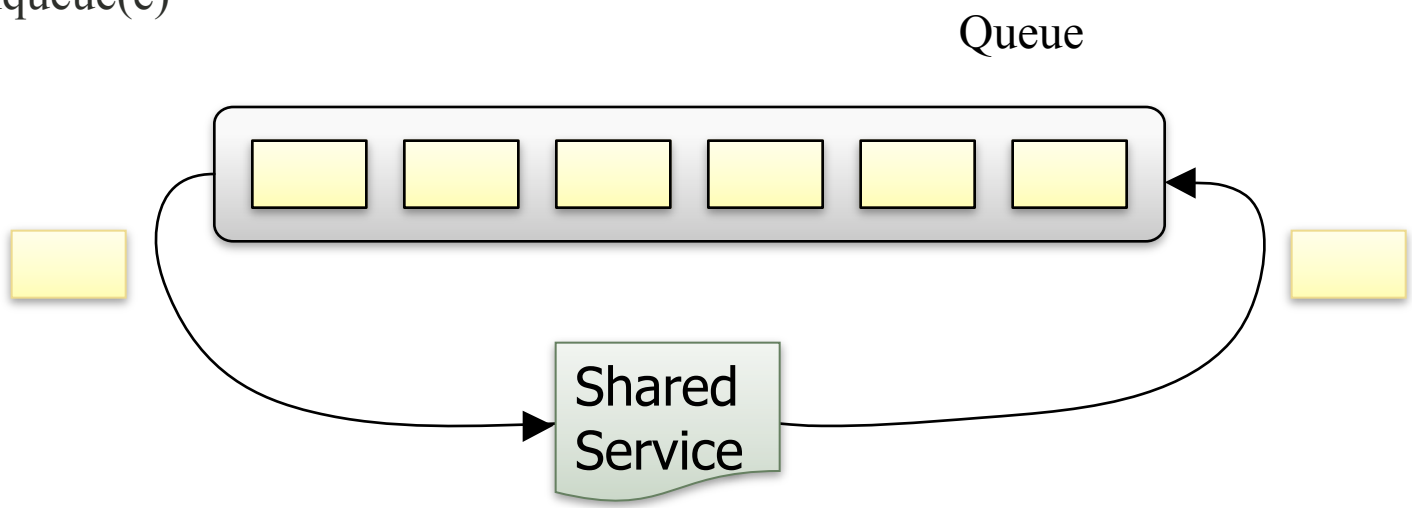
Queue Interface in Java

```
public interface Queue<E> {

   public int size();

   public boolean isEmpty();

   public E front()
          throws EmptyQueueException;

   public void enqueue(E element);

   public E dequeue()
           throws EmptyQueueException;
}
```

- Similar to the Stack ADT, operations front and dequeue cannot be performed if the queue is empty
- Attempting the execution of front or dequeue on an empty queue throws an EmptyQueueException

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)



Dequeus

- Pronounce “de – ke”
- Double-ended queues
- addFirst(), addLast(), removeFirst(), removeLast(), getFirst(), getLast(), size(), isEmpty()
- Implemented by doubly linked list

Operation	Output	D
addFirst(3)	-	(3)
addFirst(5)	-	(5,3)
removefirst()	5	(3)
addLast(7)	-	(3,7)
removeFirst()	3	(7)
removeLast()	7	()
removeFirst()	“error”	()
isEmpty()	true	()

Priority Queues

- Queues whose elements are in order
- You have implemented a priority queue in HW4
- Priority queues can be used for sorting

Insertion Sort

	Sequence S	Priority queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(4,7)
(c)	(2,5,3,9)	(4,7,8)
(d)	(5,3,9)	(2,4,7,8)
(e)	(3,9)	(2,4,5,7,8)
(f)	(9)	(2,3,4,5,7,8)
(g)	()	(2,3,4,5,7,8,9)
Phase 2		
(a)	(2)	(3,4,5,7,8,9)
(b)	(2,3)	(4,5,7,8,9)
..	..	..
(g)	(2,3,4,5,7,8,9)	()

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?

HW 5 (Due on 11/2)

Matching/Analyzing HTML Tags.

- Check whether a given (simplified) HTML web page is valid
  - By HTML Tag Matching (TB, page. 212-213)
- Output:
  - all the matched tags if the html is valid
  - The first mis-matched tag if the html is invalid
- Assume all matched tags are in the form:
  - <name>…</name>

Coming Up…

- So far all the mentioned data structures are sequential, i.e., data are stored in a sequence
- We will talk about “hierarchical” data structures on Nov. 2
- We will have programming tests in the lab on Nov. 6
- Read Chapter 7