4CCS1DST – Data Structures

Lecture 4:

Stacks and Queues

(5/e: Ch.5; 6/e: Ch. 6)

Abstract Data Types (ADTs)

- An abstract data type (ADT) is an abstraction of a data structure
- An ADT specifies:
  - **Type of data stored**
  - **Operations on the data**
  - Error conditions associated with operations
- An ADT does not say how operations are implemented. Different implementations possible.

The Stack ADT

- An instance of the stack data structure is a sequence of elements (objects) with one end designated as the top of the stack:

  E1, E2, E3, … , En

  top of the stack

  En
  …
  E3
  E2
  E1

- Main stack operations (last-in first-out, LIFO, principle):
  - push(e): insert element e at the top of the stack
  - pop(): remove and return the element at the top of the stack
- Additional stack operations:
  - top(): return the top element in the stack (without removing)
  - size(): return the number of elements stored
  - isEmpty(): check if the stack is empty

error, if empty stack

Interface Stack 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<E>:

```
public class Stack<E> extends Vector<E>
```

```
public interface Stack<E> {

  public void push(E element);

  public E pop()
        throws EmptyStackException;

  public E top()
        throws EmptyStackException;

  public int size();

  public boolean isEmpty();
}
```

Class Stack in java.util

java.util

Class Stack<E>

All Implemented Interfaces: …

public class Stack<E> extends Vector<E>

Method Summary

boolean	**empty**()	Tests if this stack is empty.
E	**peek**()	Looks at the object at the top of this stack without removing it from the stack.
E	**pop**()	Removes the object at the top of this stack and returns that object as the value of this function.
E	**push**(E item)	Pushes an item onto the top of this stack.
int	**search**(Object o) …	

Exceptions

- Attempting the execution of an operation of an ADT may sometimes cause an error condition.
- We implement error conditions using Java exceptions:

  If error is identified, then "throw" an appropriate exceptions.

```
Stack<Integer> s  =  new  ArrayStack<Integer>();
…
```

```
System.out.println( "top of stack: " + s.top() );   // not safe
```

```
if ( !s.isEmpty() ) {  System.out.println("top of stack: " + s.top() ); }
else {  System.out.println("stack is empty"); }
```

```
try {  System.out.println("top of stack: " + s.top() ); }
catch (EmptyStackException ex) {System.out.println("empty stack"); }
```

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

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 **push**es on the stack a "frame" containing
  - local variables
  - program counter (PC), keeping track of the current instruction
- When a method ends, its frame is **pop**ped from the stack and control is passed to the method now on top of the stack
- Method stack supports recursion

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

   foo(int j) {
1:    int k;
2:    k = j+1;
3:    bar(k);
4:    …
   }

   bar(int m) {
1:    if  (m > 5) {
2:       bar(m-2);
      }
3:    …
   }
```

bar
PC = 1
m = 4

bar
PC = 2
m = 6

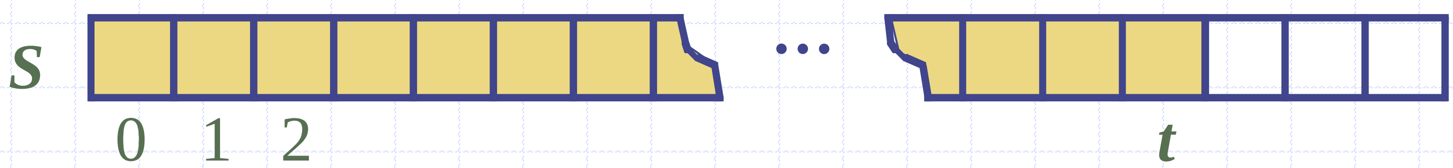
foo
PC = 3
j = 5
k = 6

main
PC = 2
i = 5

Array-based Stack implementation

- 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

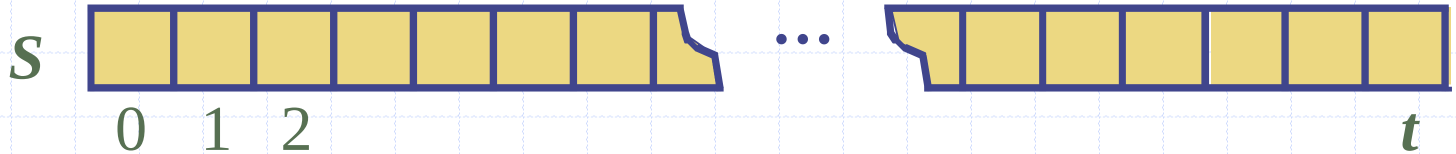
```
Algorithm size()
  return t + 1

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



Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then throw 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
    $t \leftarrow t + 1$
    $S[t] \leftarrow o$



Performance and Limitations

- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$ (if we know 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

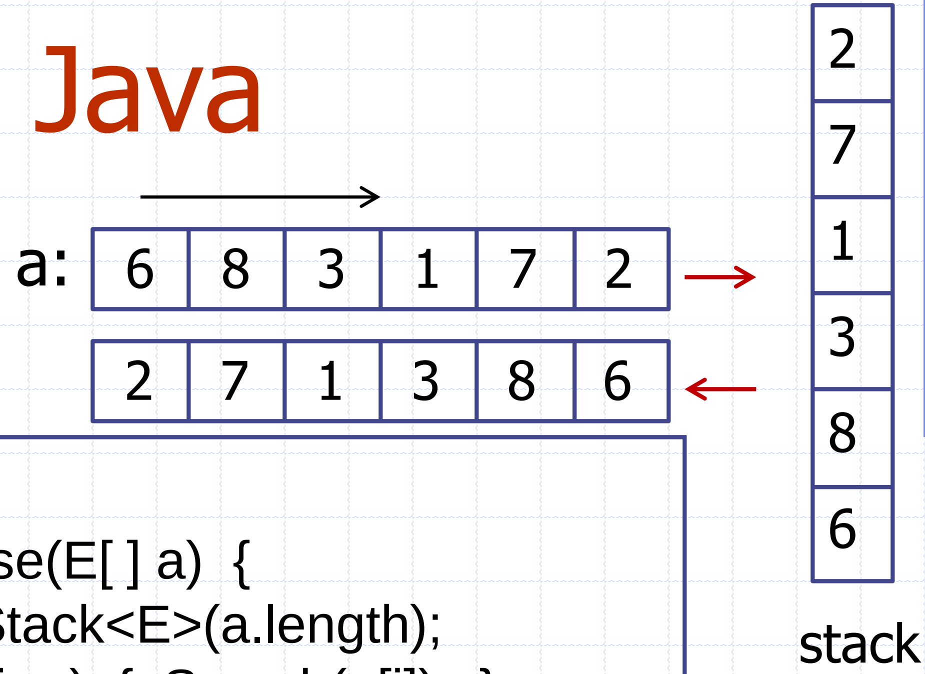
Array-based Stack in Java

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

   // S[ ] holds stack elements
   protected  E  S[ ];

   // index to top element
   protected  int   top  =  –1;

   // constructor
   public ArrayStack(int cap) {
      S = (E[ ]) new Object[cap]);
   }
```

"new E[cap]" – not allowed because E is not a concrete type

```
public  E  pop()
     throws EmptyStackException {

   if  isEmpty()
      throw  new
           EmptyStackException
           ("Empty stack: cannot pop");

   E  element  =  S[top];
   top – –;
   S[top] = null;
               // for garbage collection
   return  element;
}

…(other Stack methods in textbook)
```

Example use in Java



Reverse elements in array

```
public class StackTester {
    public static <E> void reverse(E[ ] a) {
        Stack<E> S = new ArrayStack<E>(a.length);
        for (int i=0; i < a.length; i++) { S.push(a[i]); }
        for (int i=0; i < a.length; i++) { a[i] = S.pop(); }
    }
    public static void main(String[ ] args) {
        String[ ] s = { "Jack", "Kate", "Hurley", "Jin", "Boone“ };
        System.out.println( "s = " + Arrays.toString(s) );
        reverse(s);
        System.out.println("s = " + Arrays.toString(s));
    }
}
```

Parentheses Matching

- Each "(", "{", or "[" must be paired with a matching ")", "}", or "["
  - correct: ( )(( )) { ([( )]) }
  - correct: ((( ) (( )) { ([( )]) }
  - incorrect:) (( )) { ([( )]) }
  - incorrect: ({ [ ]) }
  - incorrect: (
- "(", "{", "[", ")", "}", "[" – "grouping" symbols.

Parentheses Matching Algorithm

X :

(	[	a	+	3	]	*	....

Algorithm ParenMatch(*X*,*n*):

Input: An array *X* of *n* tokens, each of which is either a grouping symbol or other symbols (for example: variables, arithmetic operators, numbers)

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

match *X*[*i*] with symbol pop'ed from the stack

Parentheses Matching: example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
(	[	(	a	+	b	)	*	c	+	d	*	e	]	/	{	(	f	+	g	)	–	h	}	)

i

Stack:

top

(
{
(

HTML Tag Matching

- In a fully-correct HTML file, each opening tag <name> should pair with a matching closing tag </name>. Tags are "grouping symbols" here.

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?

```
<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>
```

Tag Matching Algorithm in Java

```
package  lecture4;
import  …

/** Simplified test of matching tags in an HTML document. */
public class HTML  {

    /** Array “tag” contains a sequence of opening and closing html tags.
        For our example, array tag: [ body, center, h1, /h1, /center, p, …, /body ].
        Test if every opening tag has a matching closing tag. */
    public  static  boolean  isHTMLMatched( String[ ] tag ) {  …  }

    /** Parse an HTML document into an array of html tags */
    public static String[ ]  parseHTML(Scanner  s) {  …  }

    // Auxiliary methods
    …

    /** Test the class */
    public  static  void  main(String[ ] args)  throws  IOException {  …  }
}
```

Tag Matching Algorithm (cont.)

```
// Auxiliary methods

/** Test if a stripped tag string is an opening tag:
    check if the first character is '/'  */
public static boolean isOpeningTag( String tag ) { … }

/** Test if stripped tag1 matches closing tag2.
    For example: areMatchingTags( "name", "/name" )  returns true. */
public static boolean areMatchingTags( String tag1, String tag2 ) { … }

…  // other auxiliary methods
```

Tag Matching Algorithm (cont.)

```
/** Test if every opening tag has a matching closing tag. */

public  static  boolean  isHTMLMatched( String[ ] tag )  {

   Stack<String>  S  =  new  NodeStack<String>();          // Stack for matching tags

   for ( int i = 0;  ( i < tag.length ) && ( tag[i] != null );  i++ )  {
          if ( isOpeningTag( tag[i] ) )                       // opening tag; push it on the stack
                    S.push( tag[i] );
          else  {                                             // tag[i] is a closing tag
                    if (  S.isEmpty()  )                      // nothing to match with
                              return false;
                    if (  ! areMatchingTags( S.pop(),  tag[i] )  )      // wrong match
                              return false;
          }
   }

   if ( S.isEmpty() ) return true;       // we matched everything and Stack is empty
   return false;                         // some unmatched opening tags remain on Stack
}
```

Tag Matching Algorithm (cont.)

```
/** Test if a stripped tag string is empty or a true opening tag. */
public  static  boolean  isOpeningTag( String tag )  {
        return (tag.length() == 0) || (tag.charAt(0) != '/');
}

/** Test if stripped tag1 matches closing tag2 (first character is '/'). */
public  static  boolean  areMatchingTags( String tag1, String tag2 )  {
        return  tag1.equals(tag2.substring(1));    // test against "name" after " / "
}

/** Strip the first and last characters off a <tag> or </tag> string. */
public  static  String  stripEnds( String  t ) {
        if ( t.length() <= 2 ) return null; // this is a degenerate tag
        return t.substring( 1, t.length()-1 );
}
```

Tag Matching Algorithm (cont.)

```
public final static int  CAPACITY = 1000;          // Tag array size

 /* Parse an HTML document into an array of html tags */
 public static String[ ]  parseHTML(Scanner  s)  {
     String[ ] tag = new String[CAPACITY];          // our tag array (initially all null)
     int count = 0;                                 // tag counter
     String token;                                  // token returned by the scanner s
     while (s.hasNextLine()) {
          while  ( (token = s.findInLine("<[^>]*>") ) != null )   // find the next tag
                    tag[count++] = stripEnds(token);              // strip the ends off this tag
          s.nextLine(); // go to the next line
     }
   return tag;          // our array of (stripped) tags
 }
```

Tag Matching Algorithm (cont.)

```
/** Test the class */
public  static  void  main(String[ ] args)  throws  IOException {

    Scanner myScanner = new Scanner( new FileReader( "inputs/example.html“ ) );
    String[]  tagArray  =  parseHTML( myScanner );

    if (isHTMLMatched( tagArray ) )

            System.out.println( "The input file is a matched HTML document.“ );

    else

            System.out.println( "The input file is not a matched HTML document.“ );
  }

}  // end of class  HTML
```

Queues: The Queue ADT

rear

front → E1, E2, E3, … , En

- The Queue ADT stores a collection of arbitrary elements.
- Insertions and deletions follow the **First-In First-Out** (**FIFO**) scheme.
- The elements are arranged in a sequence – “queue”. Insertions are at the **rear** of the queue and removals are at the **front**.
- Main queue operations:
  - enqueue(e): inserts element e at the end (rear) of the queue
  - dequeue(): removes and returns the element at the front of the queue
- Auxiliary queue operations:
  - front(): returns the element at the front without removing it
  - size(): returns the number of elements stored
  - isEmpty(): returns a boolean value indicating whether no elements are stored
- Exceptions
  - Attempting dequeue or front on an empty queue throws an EmptyQueueException

Example

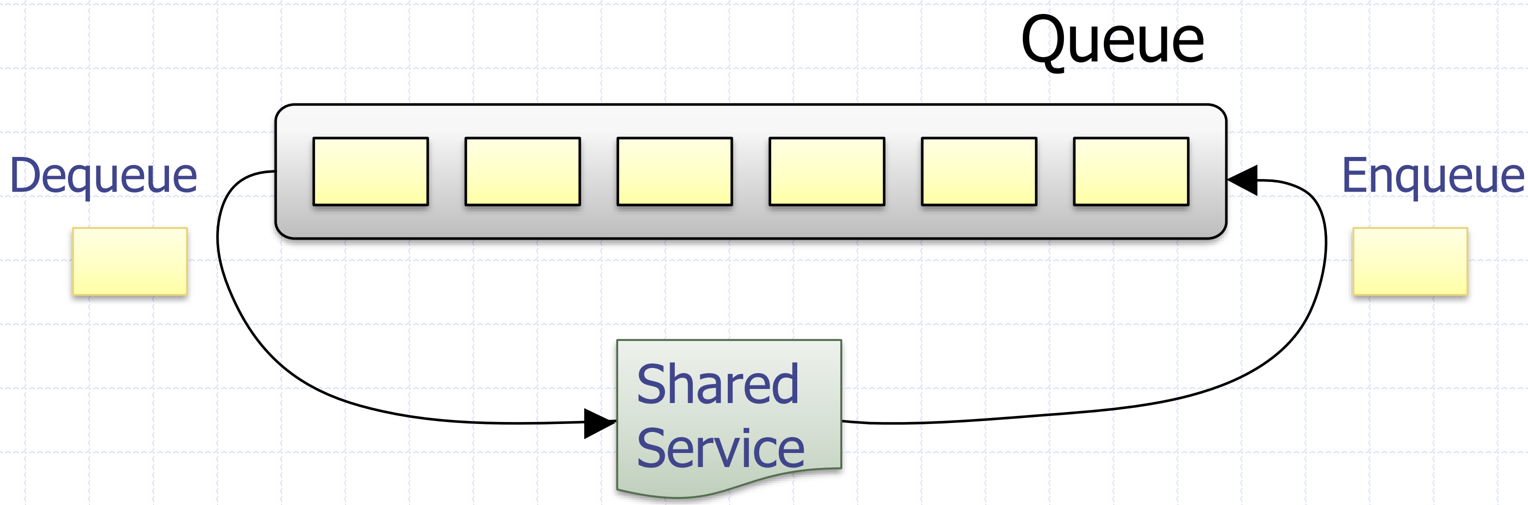
Operation	***Output***	*Queue*
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
dequeue()	*5*	(3)
enqueue(7)	–	(3, 7)
dequeue()	*3*	(7)
front()	*7*	(7)
dequeue()	*7*	()
dequeue()	*“error”*	()
isEmpty()	*true*	()
enqueue(9)	–	(9)
enqueue(7)	–	(9, 7)
size()	*2*	(9, 7)
enqueue(3)	–	(9, 7, 3)
enqueue(5)	–	(9, 7, 3,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

Applications of Queues

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

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)



Queue Interface in Java

- Java interface corresponding to our Queue ADT
- Requires a definition of class EmptyQueueException
- Corresponding built-in Java interface:

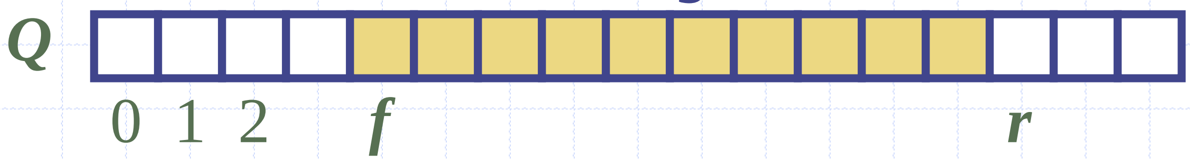
```
public interface Queue<E> {
  public int size();
  public boolean isEmpty();
  public E front()
      throws EmptyQueueException;
  public void enqueue(E element);
  public E dequeue()
      throws EmptyQueueException;
}
```

```
public interface Queue<E> extends Collection<E>
```

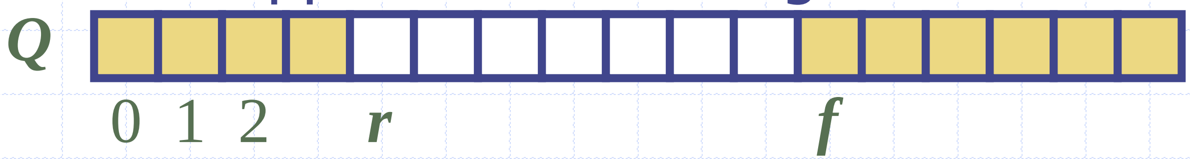
Array-based Queue (circular Queue)

- 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
- The array location r is kept empty

"normal" configuration



wrapped-around configuration

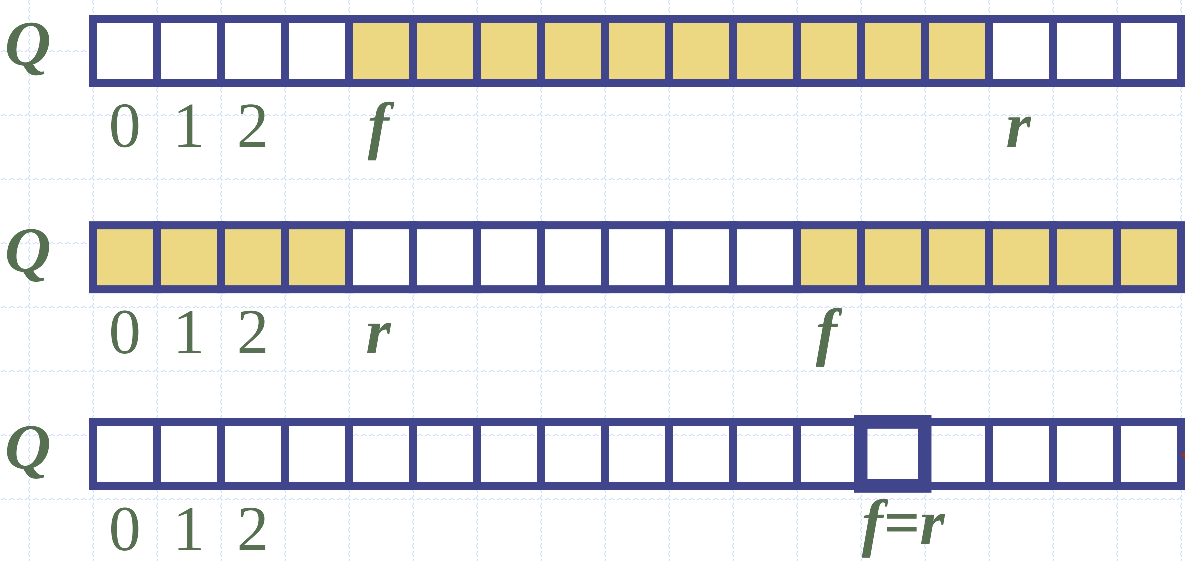


Queue Operations

- We use the modulo operator (remainder of division)

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

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



size: $r - f$

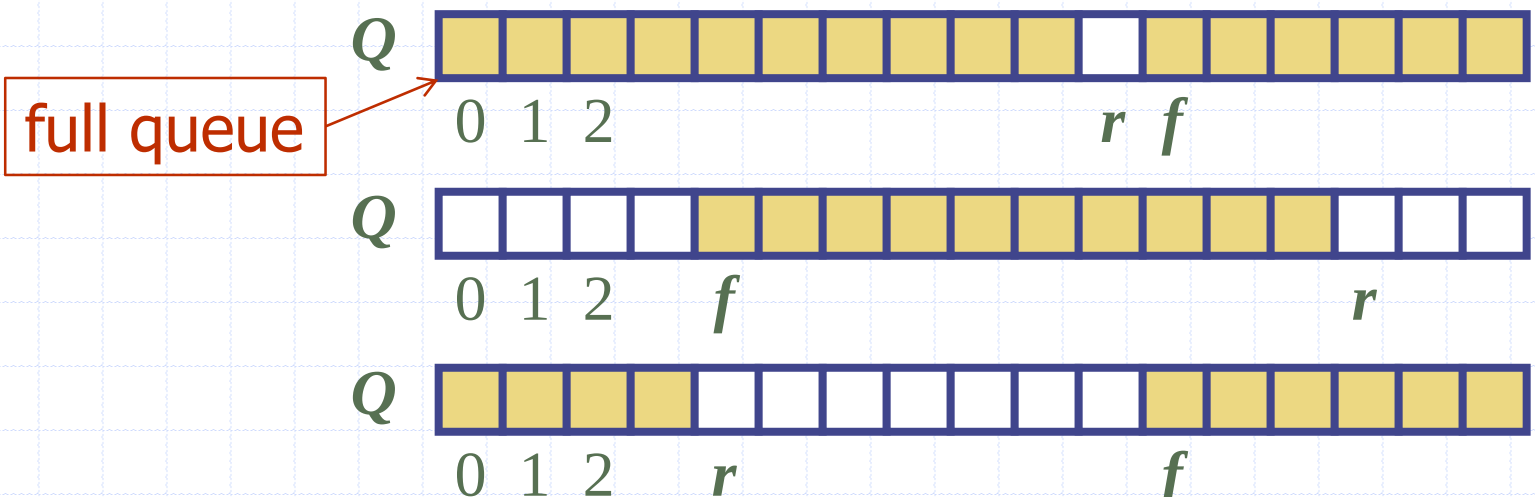
size: $N - (f - r) = r - f + N$

empty queue

Queue Operations (cont.)

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

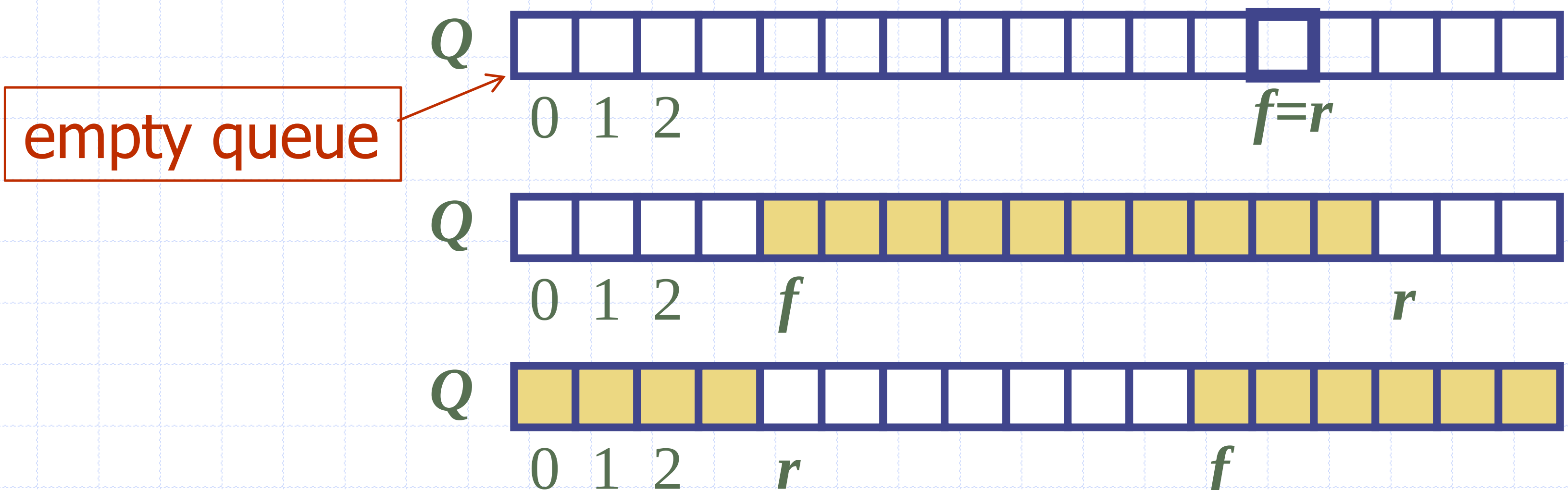
Algorithm *enqueue*(*e*)
  if *size*() = $N - 1$ **then**
    throw *FullQueueException*
  else
    $Q[r] \leftarrow e$
    $r \leftarrow (r + 1) \bmod N$



Queue Operations (cont.)

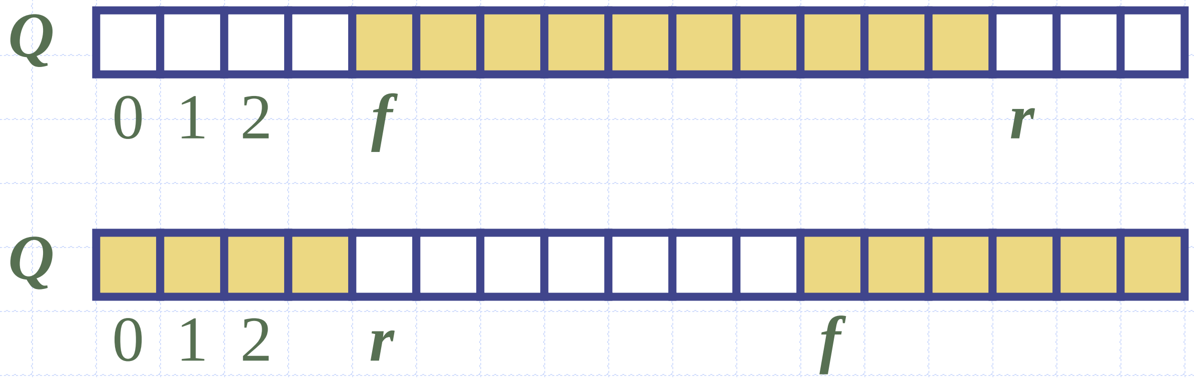
- Operation dequeue throws an exception if the queue is empty
- This exception is specified in the queue ADT

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



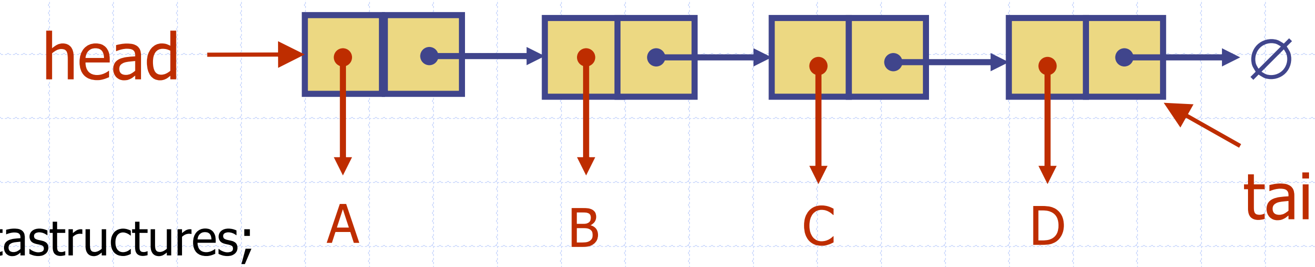
Array-based Queue (circular Queue)

- Details of Java implementation in the textbook and in the lab codes.



Implement Queue with a linked list (1)

from Lecture 2




```
package net.datastructures;

public  class  NodeQueue<E>  implements  Queue<E>  {
    protected  Node<E>  head, tail;            // the head and tail nodes
    protected  int  size;                      // number of elements in queue

    /** Creates an empty queue. */
    public NodeQueue()  {  head = null;  tail = null;  size = 0;  }

    public  int size() {  return size;  }      // return the current queue size

    public boolean  isEmpty() {                // returns true iff queue is empty
        if ( (head==null) && (tail==null) )   return  true;
        return false;
    }
```

Implement Queue with a linked list (2)

```
public void enqueue(E elem) {              // same as insertAtTail in SLinkedList
    Node<E> node = new Node<E>(elem, null);        // new tail node
    if  ( size == 0 )  head = node;        // special case: empty queue
    else   tail.setNext(node);             // add node at the tail of the list
    tail = node;    size++;                // update tail and size
    }
public E dequeue() throws EmptyQueueException {
                                           // similar to removeAtHead in SLinkedList
    if  ( size == 0 )  throw new EmptyQueueException("Queue is empty.");
    E  tmp = head.getElement();
    head = head.getNext();
    size--;
    if  (size == 0 )   tail = null;        // the queue is now empty
    return tmp;
  }
public E front()  { ... }      public static void main(String[] args)  { ... }  }
```

Exercises

Exercise 1

How would you modify the Parentheses Matching Algorithm if you wanted as output the pairs of positions of matched parentheses.

For example, for the input

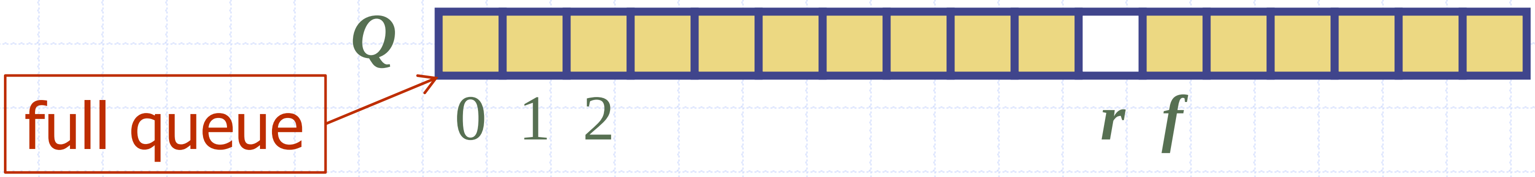
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
(	[	(	a	+	b	)	*	c	+	d	*	e	]	/	{	(	f	+	g	)	–	h	}	)

the output should be:

(2,6) (1,13) (16,20) (15,23) (0,24)

Exercise 2

In the array-based implementation of Queue, the array location ***r*** is kept empty. Consequently, when the queue is considered full, there is still one empty location in the array.



Could we put another element in that final empty location, and declare that the queue is full only if the size of the queue is equal to the size (length) of the array? What would be a problem with this approach and how that problem could be fixed?

Exercise 3

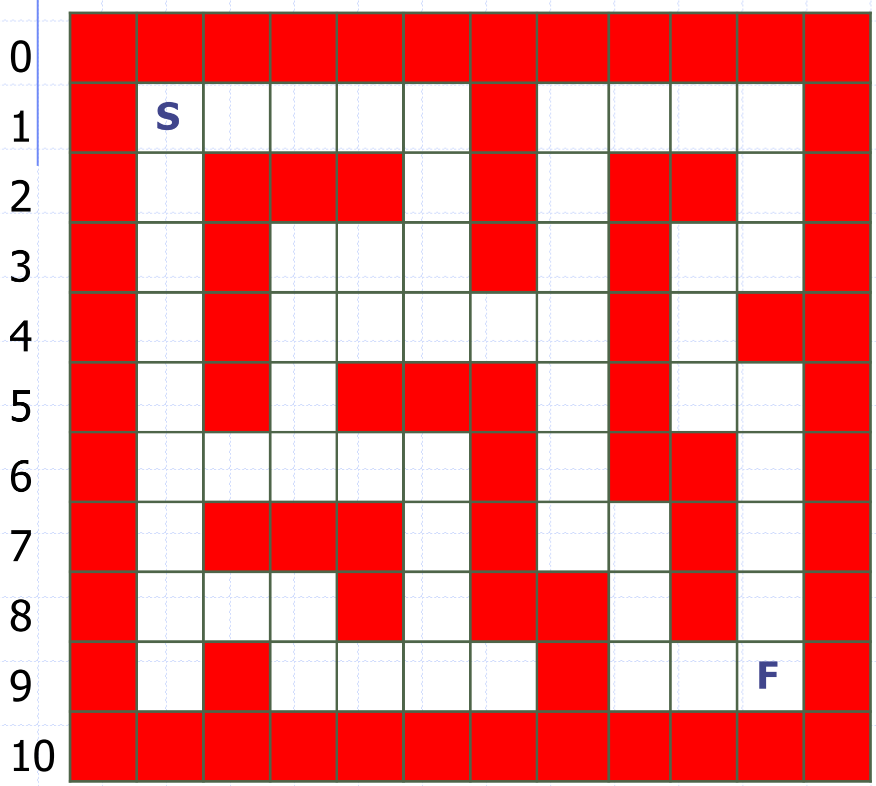
A. An example of a maze is shown on the next slide. Consider the following algorithm for exploration of such a maze to find out if the Finish location is reachable from the Start location. We can go from the current location only to a neighbouring location (sharing one side).

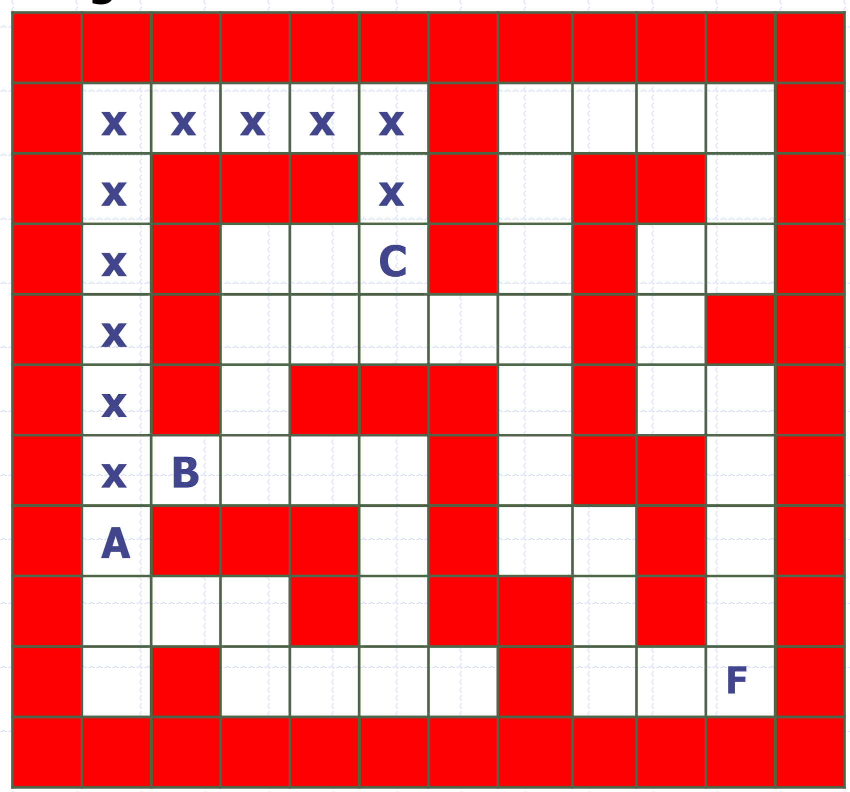
Algorithm:

- Each location is either unknown, discovered or explored.
- Initially Start is discovered and all other locations are unknown.
- All discovered locations are kept in a Queue.
- While Queue is not empty:
  - take (remove) a (discovered) location from Queue,
  - mark this location as explored,
  - mark all its unknown neighbouring locations as discovered and add them to Queue.
- Stop when Finish is discovered (Finish is reachable from Start) or when Queue becomes empty (Finish is not reachable).

Exercise 3 (cont.)

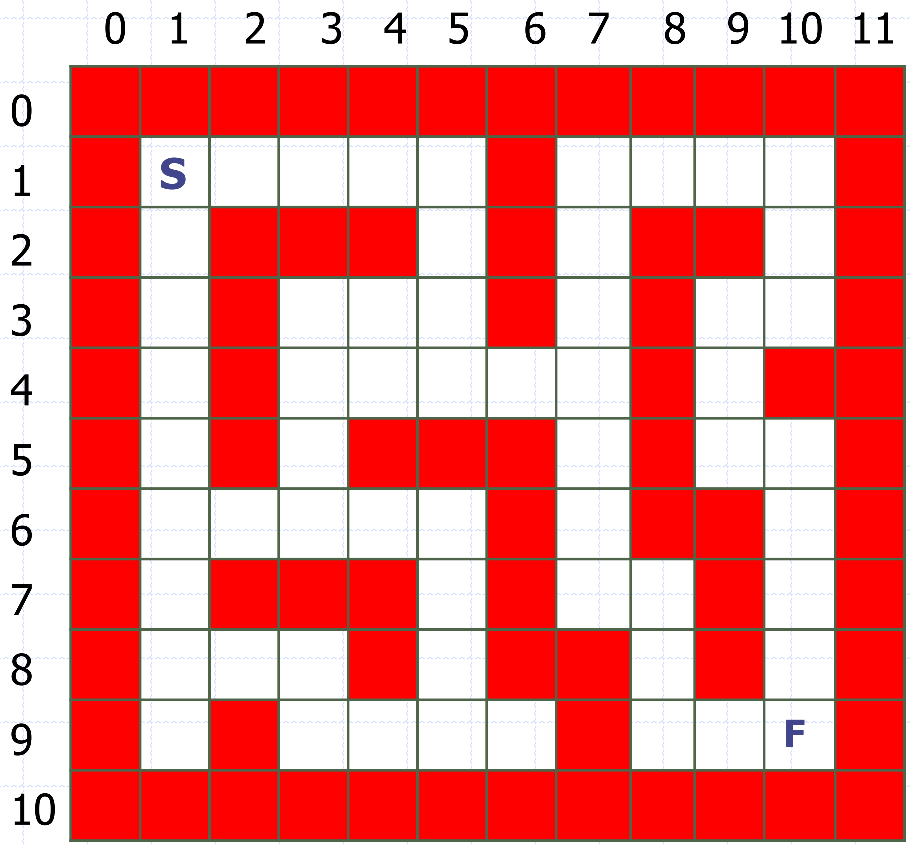
- Show the status of each location (unknown, discovered, or explored) when Finish is discovered. Assume the neighbouring locations of the current location are considered in the order: S, E, N, W. The right diagram shows an intermediate state: the explored locations are marked with x; Queue is (A,B,C), holding the discovered locations.





Exercise 3 (cont.)

B. How could we extend this algorithm to output a path from the location Start to the location Finish, if Finish is reachable from Start?



Exercise 3 (cont.)

C. What would happen, if we used Stack instead of Queue in this algorithm? Trace the computation showing the status of the locations (unknown, discovered, or explored) and the contents of the Stack. Assume the neighbouring locations of the current location are considered in the order: S, E, N, W.