King Fahd University of Petroleum & Minerals
College of Computer Science & Engineering

Information & Computer Science Department

Unit 4

Stacks And Queues

Information and Computer Science ICS

Reading Assignment

- "Data Structures and Algorithms in Java", 3rd Edition, Adam Drozdek, Cengage Learning, ISBN 978-9814239233
  - Chapter 4

Objectives

Discuss the following topics:

- Stacks
- Queues
- Priority Queues
- Case Study: Exiting a Maze [Self Reading]

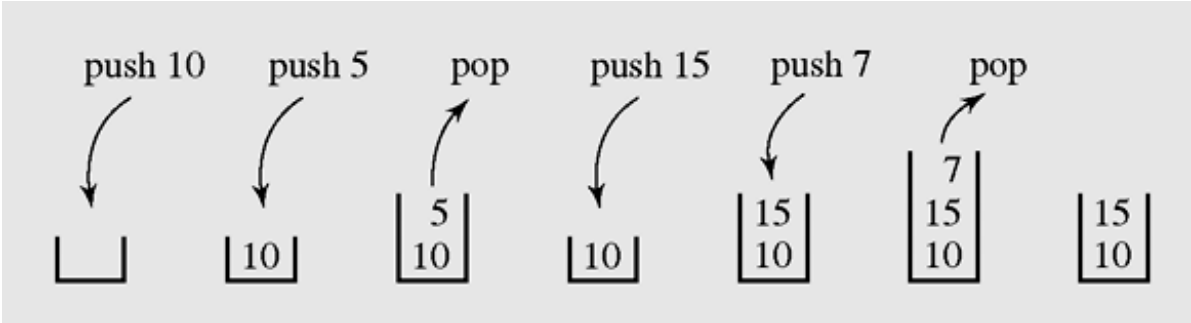
Stacks

- A **stack** is a linear data structure that can be accessed only at one of its ends for storing and retrieving data
- A stack is called an **LIFO** structure: last in/first out

Stacks (continued)

- The following operations are needed to properly manage a stack:
  - *clear()* — Clear the stack
  - *isEmpty()* — Check to see if the stack is empty
  - *push(el)* — Put the element *el* on the top of the stack
  - *pop()* — Take the topmost element from the stack
  - *topEl()* — Return the topmost element in the stack without removing it

Stacks (continued)



A series of operations executed on a stack

Stacks (continued)

```
public class Stack<T> {
    private java.util.ArrayList<T> pool = new java.util.ArrayList<T>();
    public Stack() {
    }
    public Stack(int n) {
        pool.ensureCapacity(n);
    }
    public void clear() {
        pool.clear();
    }
    public boolean isEmpty() {
        return pool.isEmpty();
    }
    public T topEl() {
        if (isEmpty())
            throw new java.util.EmptyStackException();
        return pool.get(pool.size()-1);
    }
```

O(n)

O(1)

O(1)

Array list implementation of a stack

Stacks (continued)

```
 public T pop() {
        if (isEmpty())
            throw new java.util.EmptyStackException();
        return pool.remove(pool.size()-1);
    }
    public void push(T el) {
        pool.add(el);
    }
    public String toString() {
        return pool.toString();
    }
}
```

O(1)

O(1)

O(n)

Array list implementation of a stack (continued)

Stacks (continued)

```
public class LLStack<T> {
    private java.util.LinkedList<T> list = new java.util.LinkedList<T>();
    public LLStack() {
    }
    public void clear() {                          O(1)
        list.clear();
    }
    public boolean isEmpty() {
        return list.isEmpty();                     O(1)
    }
    public T topEl() {
        if (isEmpty())                             O(1)
            throw new java.util.EmptyStackException();
        return list.getLast();
    }
```

Implementing a stack as a linked list

Stacks (continued)

```
public T pop() {
        if (isEmpty())
            throw new java.util.EmptyStackException();
        return list.removeLast();
    }
    public void push(T el) {
        list.add(el);
    }
    public String toString() {
        return list.toString();
    }
}
```

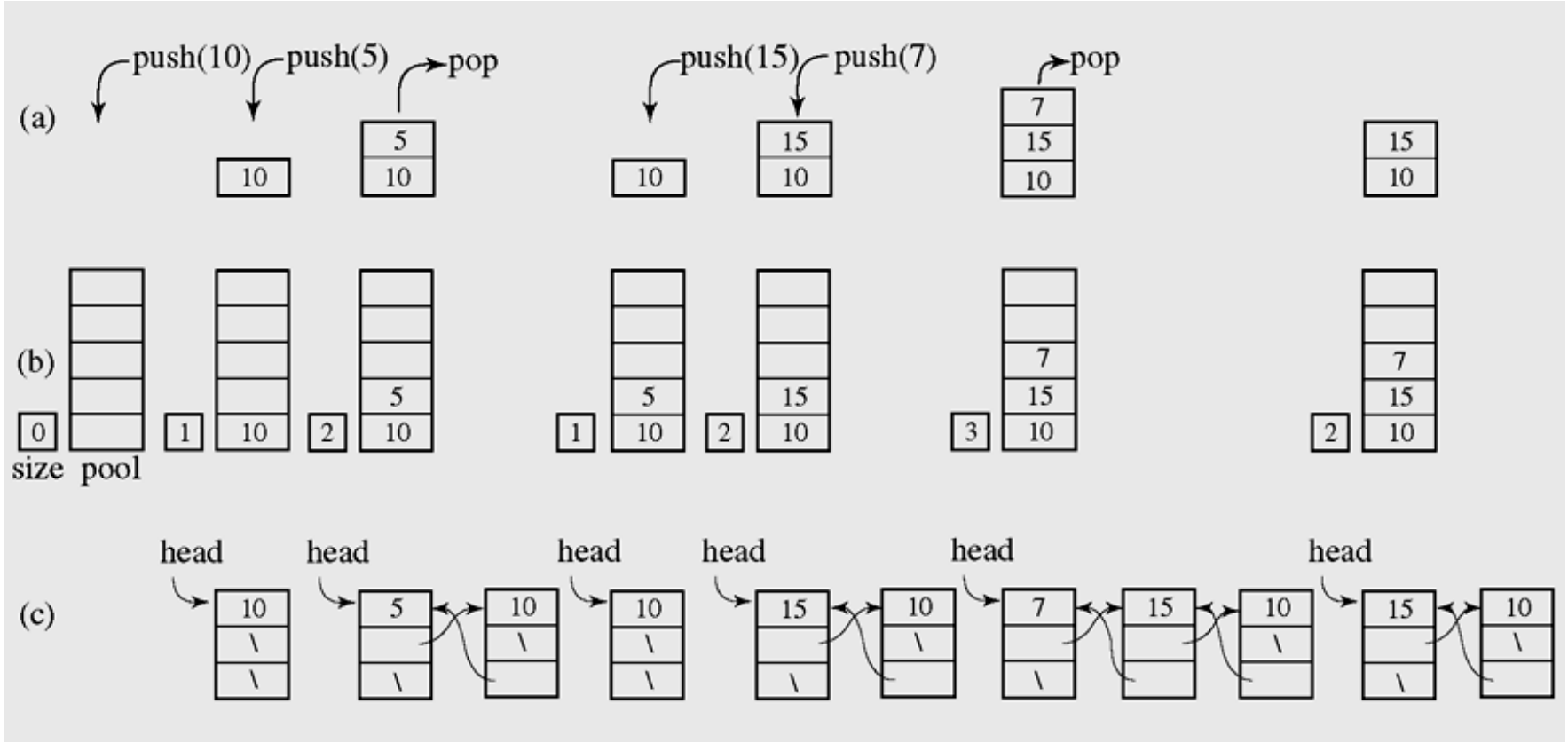
O(1)

O(1)

O(n)

Implementing a stack as a linked list (continued)

Stacks (continued)



A series of operations executed on an abstract stack (a) and the stack implemented with an array (b) and with a linked list (c)

Stacks in `java.util`

Method	Operation
`boolean empty()`	Return `true` if the stack includes no element and `false` otherwise.
`Object peek()`	Return the top element on the stack; throw `EmptyStackException` for empty stack.
`Object pop()`	Remove the top element of the stack and return it; throw `EmptyStackException` for empty stack.
`Object push(Object el)`	Insert `el` at the top of the stack and return it.
`int search(Object el)`	Return the position of `el` on the stack (the first position is at the top; –1 in case of failure).
`Stack()`	Create an empty stack.

A list of methods in `java.util.Stack`; all methods from `Vector` are inherited

Applications of Stack

- Some direct applications:
  - Delimiter Matching
  - Adding Large Numbers
  - Evaluating postfix expressions
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
- Some indirect applications
  - Auxiliary data structure for some algorithms
  - Component of other data structures

Delimiter Matching

- These examples are properly-delimited statements :
  - `a = b + (c – d ) * (e – f);`
  - `g[10] = h[i[9]] + (j + k) * l;`
  - `while (m < (n[8] + o)) { p = 7; /* initialize p */ r = 6; }`
- These examples are statements in which mismatching occurs:
  - `a = b + (c – d) * (e – f));`
  - `g[10] = h[i[9]] + j + k) * l;`
  - `while (m < (n[8] + o]) { p = 7; /* initialize p */ r = 6; }`

Delimiter Matching Algorithm

```
delimiterMaching(file)
    read character ch from file;
    while not end of file
        if ch is '(', '[', or '{'
            push(ch);
        else if ch is a double quote
            skip all characters to a double quote;
        else if ch is ')', ']', or '}'
            if ch and popped off delimiter do not match
                failure;
        else if ch is '/'
            read the next character;
            if this character is '*'
                skip all characters until "*/" is found and report an error
                if the end of file is reached before "*/" is encountered;
            else ch = the character read in;
                continue; // go to the beginning of the loop;
    // else ignore other characters;
        read next character ch from file;
    if stack is empty
        success;
    else failure;
```

Delimiter Matching

Stack	Nonblank Character Read	Input Left
empty		s = t[5] + u / (v * (w + y));
empty	s	= t[5] + u / (v * (w + y));
empty	=	t[5] + u / (v * (w + y));
empty	t	[5] + u / (v * (w + y));
[	[	5] + u / (v * (w + y));
[	5	] + u / (v * (w + y));
empty	]	+ u / (v * (w + y));
empty	+	u / (v * (w + y));
empty	u	/ (v * (w + y));
empty	/	(v * (w + y));

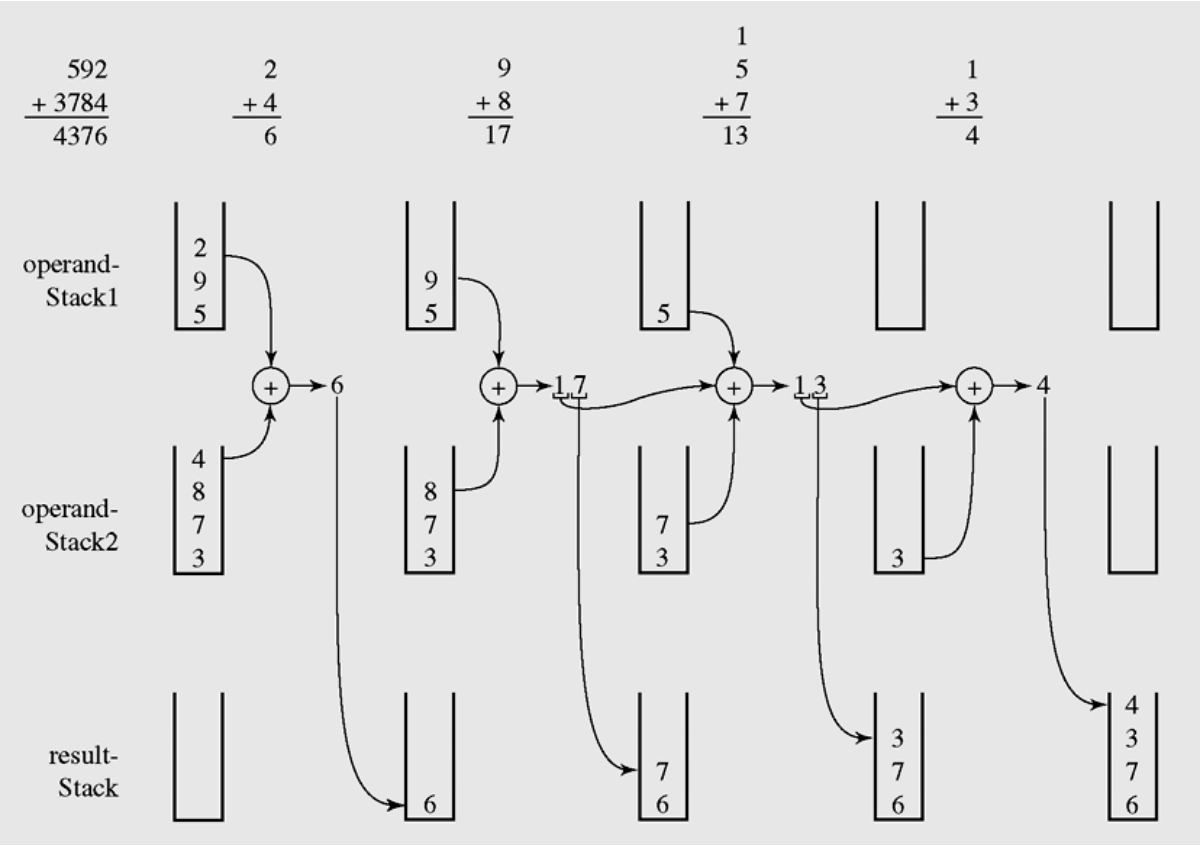
Processing the statement `s=t[5]+u/(v*(w+y));` with the algorithm `delimiterMatching()`

Delimiter Matching

(	(	v * (w + y));
(	v	* (w + y));
(	*	(w + y));
(		
(	(	w + y));
(		
(	w	+y));
(		
(	+	y));
(		
(	y	));
(	)	);
empty	)	;
empty	;	

Processing the statement `s=t[5]+u/(v*(w+y));` with the algorithm `delimiterMatching()` (continued)

Adding Large Numbers



An example of adding numbers 592 and 3,784 using stacks

Application of Stacks: Evaluating Postfix Expressions

(5+9)*2+6*5

- An ordinary arithmetical expression like the above is called infix-expression -- binary operators appear in between their operands.
- The order of operations evaluation is determined by the precedence rules and parenthesis.
- When an evaluation order is desired that is different from that provided by the precedence, parentheses are used to override precedence rules.

Application of Stacks: Evaluating Postfix Expressions

- Expressions can also be represented using postfix notation - where an operator comes after its two operands.

- The advantage of postfix notation is that the order of operation evaluation is unique without the need for precedence rules or parenthesis.

Infix	**Postfix**
16 / 2	16 2 /
(2 + 14)* 5	2 14 + 5 *
2 + 14 * 5	2 14 5 * +
(6 – 2) * (5 + 4)	6 2 - 5 4 + *

Application of Stacks: Evaluating Postfix Expressions

- The following algorithm uses a stack to evaluate a postfix expressions.

```
Start with an empty stack
for (each item in the expression) {
        if (the item is a number)
            Push the number onto the stack
        else if (the item is an operator){
            Pop two operands from the stack
           Apply the operator to the operands
            Push the result onto the stack
        }
}
Pop the only one number from the stack – that's the result of the evaluation
```

Application of Stacks: Evaluating Postfix Expressions

- Example: Consider the postfix expression, **2 10 + 9 6 - /,** which is **(2 + 10) / (9 - 6)** in infix, the result of which is 12 / 3 = 4.

- The following is a trace of the postfix evaluation algorithm for the above.

2 10 + 9 6 - /

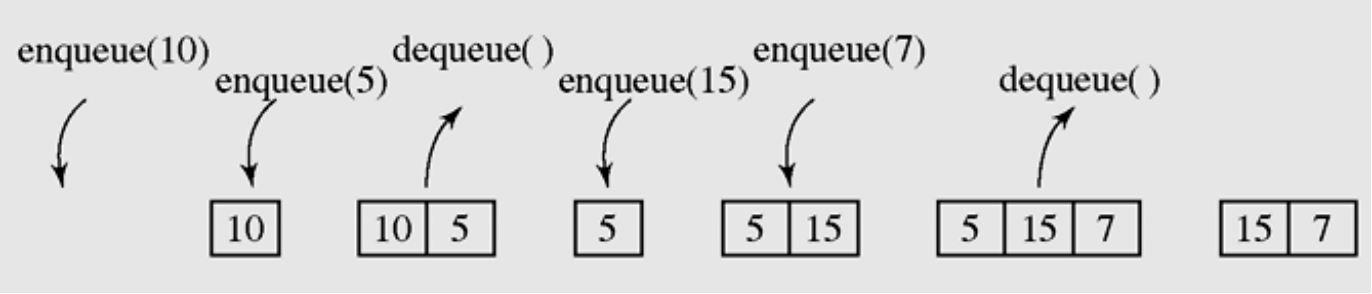
Queues

- A **queue** is a waiting line that grows by adding elements to its end and shrinks by taking elements from its front
- A queue is a structure in which both ends are used:
  - One for adding new elements
  - One for removing them
- A queue is an **FIFO** structure: first in/first out

Queues (continued)

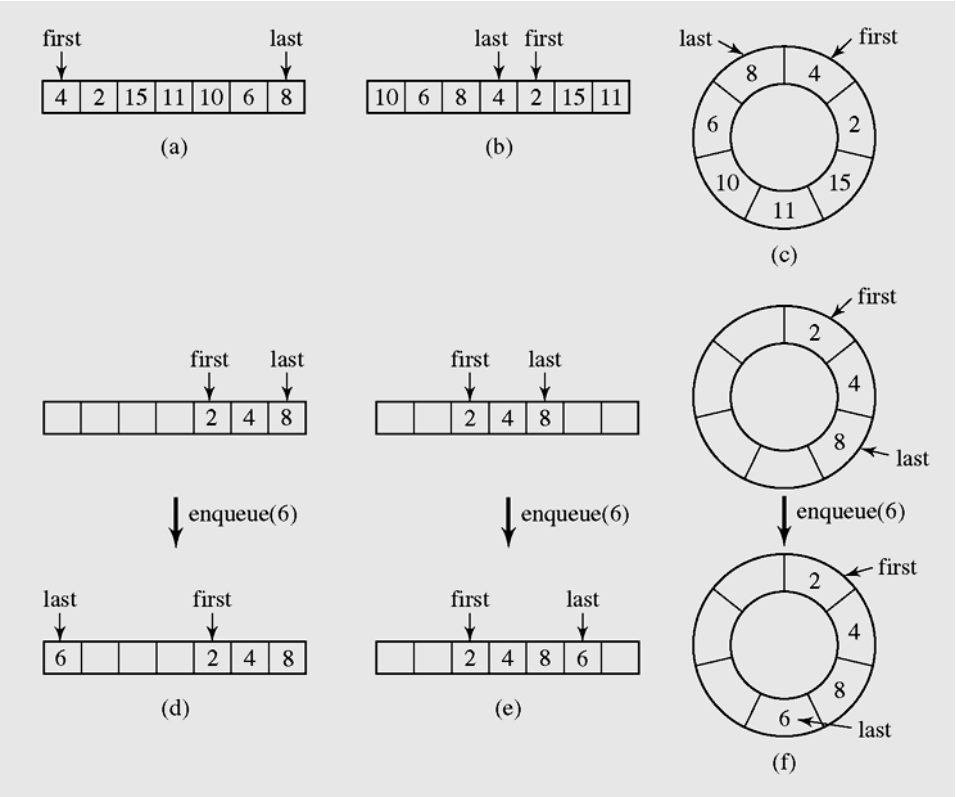
- The following operations are needed to properly manage a queue:
  - *clear()* — Clear the queue
  - *isEmpty()* — Check to see if the queue is empty
  - *enqueue(el)* — Put the element *el* at the end of the queue
  - *dequeue()* — Take the first element from the queue
  - *firstEl()* — Return the first element in the queue without removing it

Queues (continued)



A series of operations executed on a queue

Queues (continued)



Two possible configurations in an array implementation of a queue when the queue is full

Queues (continued)

```
public class ArrayQueue {
    private int first, last, size;
    private Object[] storage;
    public ArrayQueue() {
        this(100);
    }
    public ArrayQueue(int n) {
        size = n;                                                    O(  )
        storage = new Object[size];
        first = last = -1;
    }
    public boolean isFull()  {                                       O(  )
        return first == 0 && last == size-1 || first == last + 1;
    }
    public boolean isEmpty() {                                       O(  )
        return first == -1;
```

Array implementation of a queue

Queues (continued)

```
}
public void enqueue(Object el) {
    if (last == size-1 || last == -1) {
         storage[0] = el;
         last = 0;
         if (first == -1)
             first = 0;
    }
    else storage[++last] = el;
}
```

O( )

Array implementation of a queue (continued)

Queues (continued)

```
    public Object dequeue() {
        Object tmp = storage[first];
        if (first == last)
             last = first = -1;                          O(  )
        else if (first == size-1)
             first = 0;
        else first++;
        return tmp;
    }
    public void printAll() {
        for (int i = 0; i < size; i++)                   O(  )
            System.out.print(storage[i] + " ");
    }
}
```

Array implementation of a queue (continued)

Queues (continued)

```
public class Queue<T> {
   private java.util.LinkedList<T> list = new java.util.LinkedList<T>();
   public Queue() {
   }
   public void clear() {
       list.clear();
   }
   public boolean isEmpty() {
       return list.isEmpty();
   }
   public T firstEl() {
       return list.getFirst();
   }
```

O( )

O( )

O( )

Linked list implementation of a queue

Queues (continued)

```
 public T dequeue() {
        return list.removeFirst();
    }
    public void enqueue(T el) {
        list.addLast(el);
    }
    public String toString() {
        return list.toString();
    }
}
```

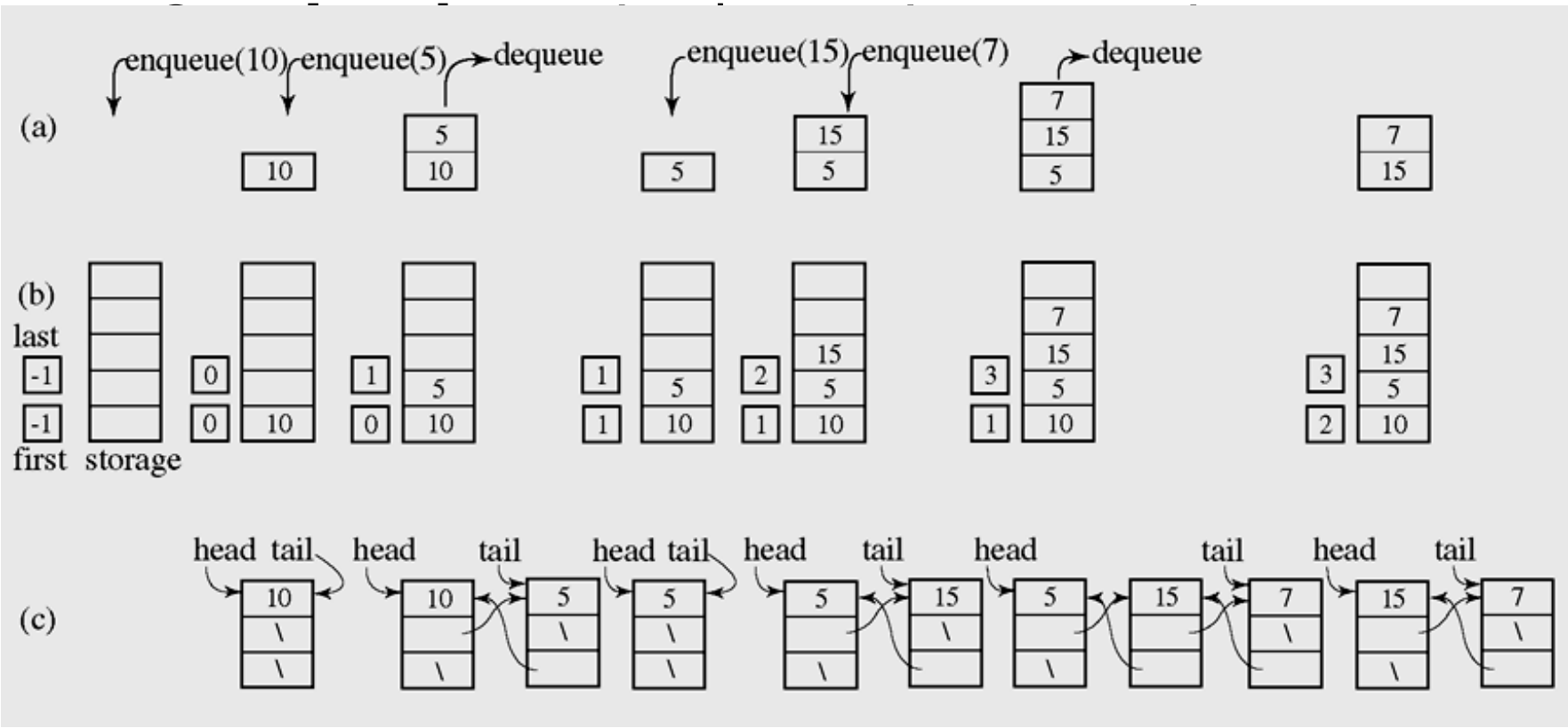
O( )

O( )

O( )

Linked list implementation of a queue (continued)

Queues (continued)



A series of operations executed on an abstract queue (a) and the queue implemented with an array (b) and with a linked list (c)

Queues (continued)

- In **queuing theory**, various scenarios are analyzed and models are built that use queues for processing requests or other information in a predetermined sequence (order)

Queues (continued)

Number of Customers per Minute	Percentage of One-Minute Intervals	Range
0	15	1–15
1	20	16–35
2	25	36–60
3	10	61–70
4	30	71–100
	(a)	

Amount of Time Needed for Service in Seconds	Percentage of Customers	Range
0	0	—
10	0	—
20	0	—
30	10	1–10
40	5	11–15
50	10	16–25
60	10	26–35
70	0	—
80	15	36–50
90	25	51–75
100	10	76–85
110	15	86–100
	(b)	

Bank One example: (a) data for number of arrived customers per one-minute interval and (b) transaction time in seconds per customer

Queues (continued)

```
class BankSimulation {
    static java.util.Random rd = new java.util.Random();
    static int Option(int percents[]) {
        int i = 0, perc, choice = Math.abs(rd.nextInt()) % 100 + 1;
        for (perc = percents[0]; perc < choice; perc += percents[i+1], i++);
        return i;
    }
    public static void main(String args[]) {
        int[] arrivals = {15,20,25,10,30};
        int[] service = {0,0,0,10,5,10,10,0,15,25,10,15};
        int[] clerks = {0,0,0};
        int clerksSize = clerks.length;
        int customers, t, i, numOfMinutes = 100, x;
        double maxWait = 0.0, thereIsLine = 0.0, currWait = 0.0;
        Queue<Integer> simulQ = new Queue<Integer>();
```

Bank One example: implementation code

Queues (continued)

```
for (t = 1; t <= numOfMinutes; t++) {
          System.out.print(" t = " + t);
          for (i = 0; i < clerksSize; i++)// after each minute subtract
              if (clerks[i] < 60)         // at most 60 seconds from time
                   clerks[i] = 0;         // left to service the current
              else clerks[i] -= 60;       // customer by clerk i;
          customers = Option(arrivals);
          for (i = 0; i < customers; i++) { // enqueue all new customers
              x = Option(service)*10;     // (or rather service time
              simulQ.enqueue(x);          // they require);
              currWait += x;
          }
```

Bank One example: implementation code (continued)

Queues (continued)

```
        // dequeue customers when clerks are available:
        for (i = 0; i < clerksSize && !simulQ.isEmpty(); )
            if (clerks[i] < 60) {
                 x = simulQ.dequeue();  // assign more than one customer
                 clerks[i] += x;        // to a clerk if service time
                 currWait  -= x;        // is still below 60 sec;
            }
            else i++;
        if (!simulQ.isEmpty()) {
             thereIsLine++;
             System.out.printf(" wait = %.1f", currWait/60.0);
             if (maxWait < currWait)
                 maxWait = currWait;
        }
        else System.out.print(" wait = 0;");
    }
```

Bank One example: implementation code (continued)

Queues (continued)

```
        System.out.println("\nFor " + clerksSize + " clerks, there was a line "
                + thereIsLine/numOfMinutes*100.0 + "% of the time;\n"
                + "maximum wait time was " + maxWait/60.0 + " min.");
    }
}
```

Bank One example: implementation code (continued)

Priority Queues

- A **priority queue** can be assigned to enable a particular process, or event, to be executed out of sequence without affecting overall system operation
- In priority queues, elements are dequeued according to their priority and their current queue position

Priority Queues (continued)

- Priority queues can be represented by two variations of linked lists:
  - All elements are entry ordered
  - Order is maintained by putting a new element in its proper position according to its priority