

Table of Contents

Topic 5 — Abstract data structures	1
5.1 Abstract data structures	1
Thinking recursively.....	1
5.1.1 – 5.1.3 Recursive thinking	1
Abstract data structures.....	7
5.1.4 – 5.1.5 Two dimensional arrays.....	7
5.1.6 – 5.1.7 Stacks.....	10
5.1.8 – 5.1.9 Queues	16
5.1.10 Arrays as static stacks and queues	19
Linked lists.....	30
5.1.11 Features and characteristics of a dynamic data structure	30
5.1.12 Operation of linked lists	31
5.1.13 Sketch linked lists	32
Trees.....	37
5.1.14 Logical operation of trees	38
5.1.15 Binary-tree related terminology.....	39
5.1.16 Tree traversal	41
5.1.17 Sketch binary trees.....	46
Applications.....	53
5.1.18 Definition of the term dynamic data structure	53
5.1.19 Comparison of static and dynamic data structures.....	53
5.1.20 Suitable structures	54
End of chapter example questions with answers.....	56
Chapter References	67
Topic 6 — Resource management	68
6.1 Resource management	68
System resources	68
6.1.1 Identification of critical resources.....	68
6.1.2 Availability of resources	73
6.1.3 Limitation of resources	76
6.1.4 Problems with insufficient resources	77
Role of the operating system.....	79
6.1.5 Role of the Operating System (OS).....	79
6.1.6 – 6.1.7 OS resource management techniques	82
6.1.8 Dedicated OS for a device	85
6.1.9 OS and complexity hiding.....	87
End of chapter example questions with answers.....	89
Chapter References	94
Topic 7 — Control	95
7.1 Control	95
Centralized control systems	95
7.1.1 A range of control systems.....	95

7.1.2 The uses of microprocessors and sensor input in control systems	101
7.1.3 Different input devices for the collection of data in specified situations	103
7.1.4 The relationship between a sensor, the processor and an output transducer	104
7.1.5 The role of feedback in a control system	106
7.1.6 Social impacts and ethical considerations associated with the use of embedded systems	106
Distributed systems.....	109
7.1.7 Comparison of centrally controlled systems with distributed systems	109
7.1.8 The role of autonomous agents acting within a larger system	111
End of chapter example questions with answers.....	115
Chapter References	120
Topic D – Object-oriented programming.....	121
Tool used.....	121
D.4 Advanced program development.....	122
D.4.1 The term “recursion”	122
D.4.2 Application of recursive algorithms	122
D.4.3 Construction of algorithms that use recursion	128
D.4.4 Trace of recursive algorithms	129
D.4.5 Define the term object reference	130
D.4.6 Construct algorithms that use reference mechanisms	133
D.4.7 Identify the features of the Abstract Data Type (ADT) list.....	137
D.4.8 Describe the applications of lists	140
D.4.9 Construct algorithms using a static implementation of a list	144
D.4.10 Construct list algorithms using object references	152
D.4.11 Construct algorithms using the standard library collections included in JETS.....	161
D.4.12 Trace algorithms using the implementations described in assessment statements	167
D.4.9-D.4.11	167
D.4.13 Explain the advantages of using library collections	171
D.4.14 Outline the features of ADT’s stack, queue and binary tree.....	173
D.4.15 Explain the importance of style and naming conventions in code	173
End of chapter example questions with answers.....	177
Chapter References	284
Appendix A — Stacks & Queues	285
A.1 Stack implementation using the ArrayList class.....	285
A.2 Queue implementation using the ArrayList class	287

TOPIC 5 – ABSTRACT DATA STRUCTURES

Most IB compatible pseudocode examples of this book have been tested using the EZ Pcode practice tool found at:

<https://dl.dropboxusercontent.com/u/275979/ibcomp/pseduocode/pcode.html>

This excellent tool was developed by Mr. Dave Mulkey. The authors wish to express their gratitude to the developer of this valuable educational resource.

© IBO
2012

Topic 5 — Abstract data structures¹

5.1 Abstract data structures

Thinking recursively

5.1.1 – 5.1.3 Recursive thinking

Exit skills. Students should be able to¹:

Identify a situation that requires the use of recursive thinking. Identify recursive thinking in a specified problem solution. Trace a recursive algorithm to express a solution to a problem.



Image 5.1: The Towers of Hanoi game

Recursion is when a method calls itself until some terminating condition is met. This is accomplished **without** any specific repetition construct, such as a **while** or a **for** loop. Recursion follows one of the basic problem solving techniques, which is to break down the problem at hand into smaller subtasks. Any algorithm that may be presented in a recursive manner can also be presented in an iterative manner and vice versa. In most cases, recursive algorithms are considered as harder to code.

Towers of Hanoi²

In order to gain a firm understanding of the basic idea, as well as the application of recursion, the following example

¹ International Baccalaureate Organization. (2012). IBDP Computer Science Guide.

² Towers of Hanoi. (2015, November 17). In *Wikipedia, The Free Encyclopedia*. Retrieved 14:03, November 17, 2014, from https://en.wikipedia.org/wiki/Tower_of_Hanoi

presents what is known as the **Towers of Hanoi**. The Towers of Hanoi is a puzzle that consists of three rods and a number of discs of different sizes, which can slide onto any rod. The puzzle starts with the discs in a neat stack in ascending order of size on the first rod, the smallest at the top, as shown in Image 5.1. The goal of the puzzle is to move the stack of discs from the first rod to the third rod, obeying the following rules:

- A disc may not be placed on top of a smaller one.
- Only one disc may move on every move.
- A disc may not be moved if it is not the top disc on a stack.
- For temporary storage, the third rod may be used.

There are various approaches that can solve the Towers of Hanoi problem, including both iterative and recursive solutions. We will be concentrating on a recursive solution, by recognizing that this puzzle may be solved by breaking it into smaller and smaller similar puzzles, until a solution is reached.

Assume that the rods are named A, B and C and that n represents the number of discs (with 1 being the smallest, at the top, and n being the largest, at the bottom). A recursive solution to the Tower of Hanoi problem, in order to move n discs from rod A to rod C could be the following:

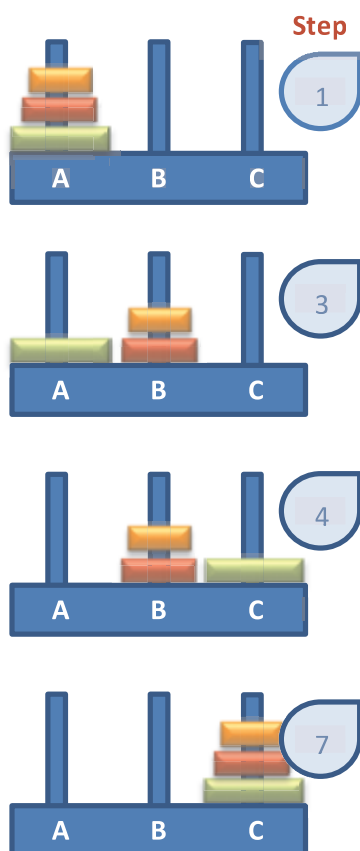


Figure 5.1: Steps of the game

- Move $n-1$ discs from rod A to rod B, leaving disc n in rod A.
- Move disc n from rod A to rod C.
- Move $n-1$ discs from rod B to rod C.

The algorithm above is recursive as it is applied again and again in both the first and the third steps for $n-1$ discs. At some point n will be equal to 1 and a single disc will be moved from rod A to rod C, resulting in an algorithm with finite number of steps.

A working example of this algorithm is examined. Figure 5.1 represents the three rods (named A, B and C) as well as three discs, stacked on top of each other in rod A. The algorithm goes as follows:

1. Move green disc from A to C.
2. Move orange disk from A to B.
3. Move green disk from C to B
4. Move grey disk from A to C
5. Move green disk from B to A
6. Move orange disk from B to C
7. Move green disc from A to C

The recursive algorithm for the solution of the Towers of Hanoi problem is also presented in Figure 5.2. Pay attention to the fact that a sub-procedure called moveDiscs is used. moveDiscs takes four

arguments. The number of the discs (n), the rod the discs are to be moved from (from), the rod to which the discs are to be moved to (dest), as well as the rod that will not be used (aux). The arguments of the moveDiscs sub-procedure (that is, n , from, aux, dest) should not be confused with the name of the rods used previously (A, B and C).

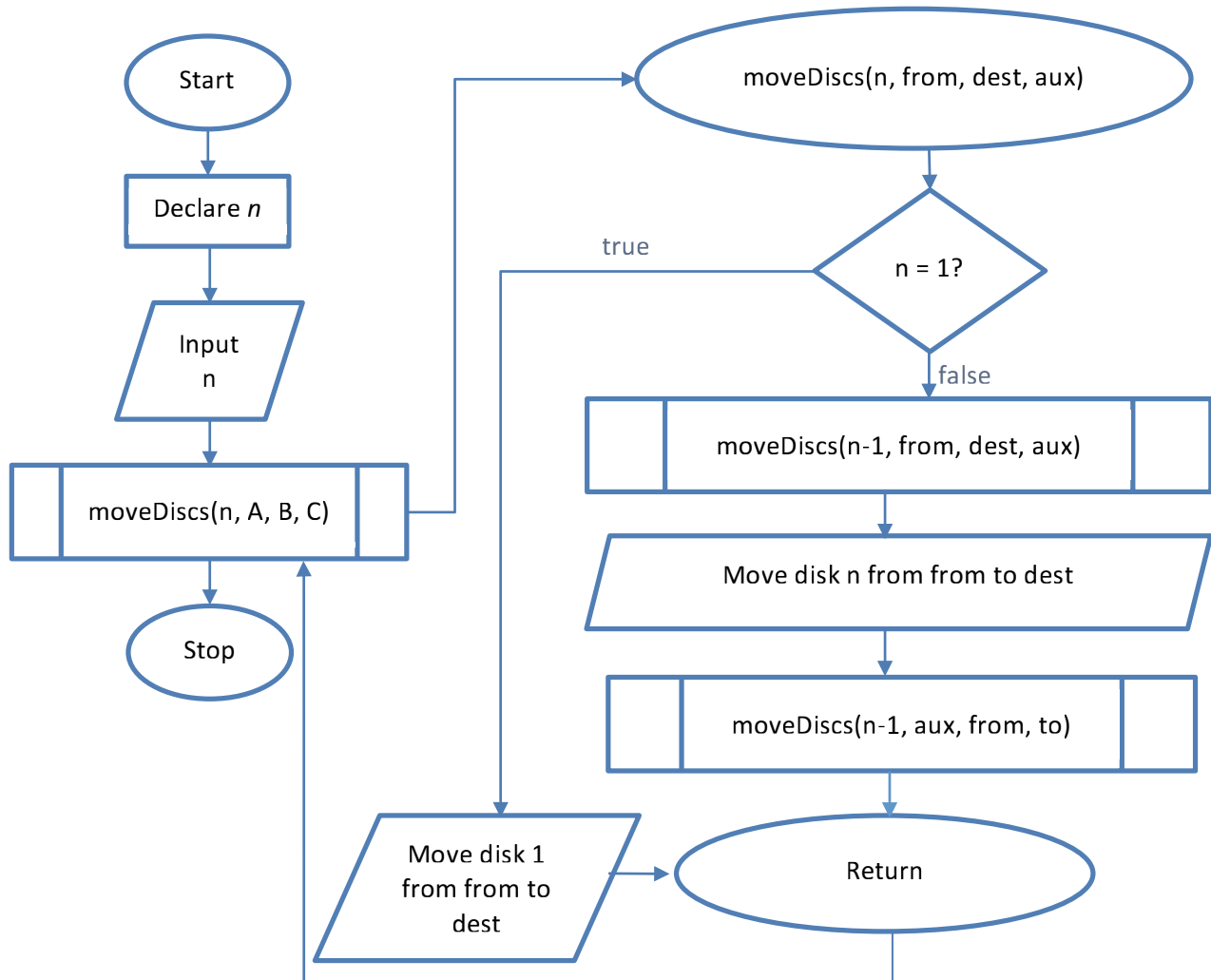


Figure 5.2: The Towers of Hanoi flowchart

Snowflakes

The Koch snowflake is a mathematical curve which is based on the Koch curve, developed by the Swedish mathematician Helge von Koch.

This mathematical curve can be constructed by starting with an equilateral triangle. Using recursion each line segment changes using the following steps:

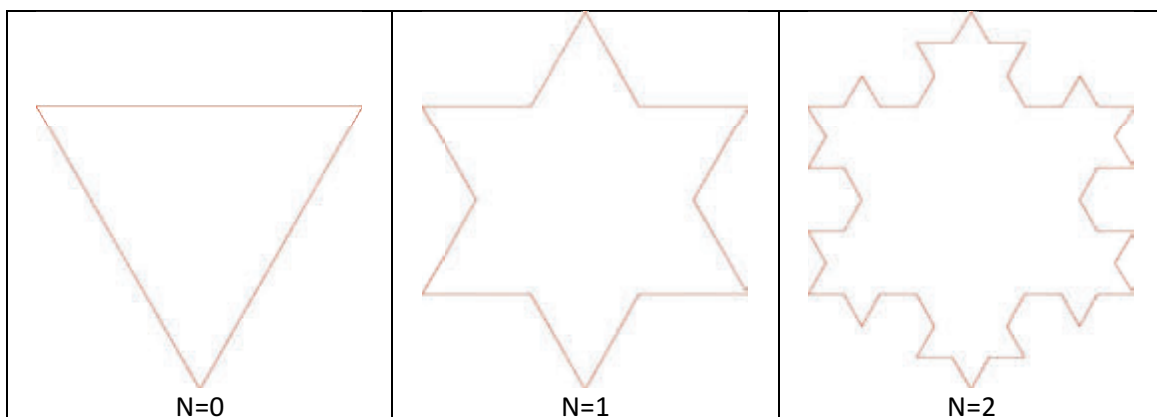
1. divide the initial line segment into three sub-segments of the same length.
2. draw an outward pointing equilateral triangle that has the middle segment from step (1) as its base.
3. delete the line segment that is the base of the triangle from previous step.

The following algorithm expressed in IB pseudocode creates a 400 by 461 window and draws a Koch fractal:

```
//Three curves that shape an equilateral triangle
//pen originally is heading at 90 degrees (x axis)
//the method pen.goForward is supposed to control
//a pen that plots line segments on the screen
//the method pen.turnLeft is supposed to change the original
//heading of the pen counter clockwise by the degrees given as a parameter.
//the method pen.turnRight is supposed to change the original
//heading of the pen clockwise by the degrees given as a parameter.
method Draw_Koch_fractal(N)
    width = 400//width of the window
    height = 2*width/Math.sqrt(3)//calculation of the height of the window
    size = width/Math.pow(3.0, N)//size of each drawing step
    initial_pen_position = pen.setposition(0, width*Math.sqrt(3)/2, 0)
    //calculation of the initial pen position (0,114)
    pen.setWindowSize(width, height)//initialization of the window
    koch_fractal(N)//call of the Koch_fractal method
    pen.turnrRight(120)//turn right by 120 degrees
    koch_fractal(N)//call of the Koch_fractal method
    pen.turnRight(120)
    koch_fractal(N)
end method

method koch_fractal(n)
    if (n == 0) then
        pen.goForward(size)
    else
        koch_fractal(n-1)
        pen.turnLeft(60)
        koch_fractal(n-1)
        pen.turnRight(120)
        koch_fractal(n-1)
        pen.turnLeft(60)
        koch_fractal(n-1)
    end if
end method
output Draw_Koch_fractal(N)
```

The following table depicts the snowflakes produced by the above algorithm for N=0 to 5:



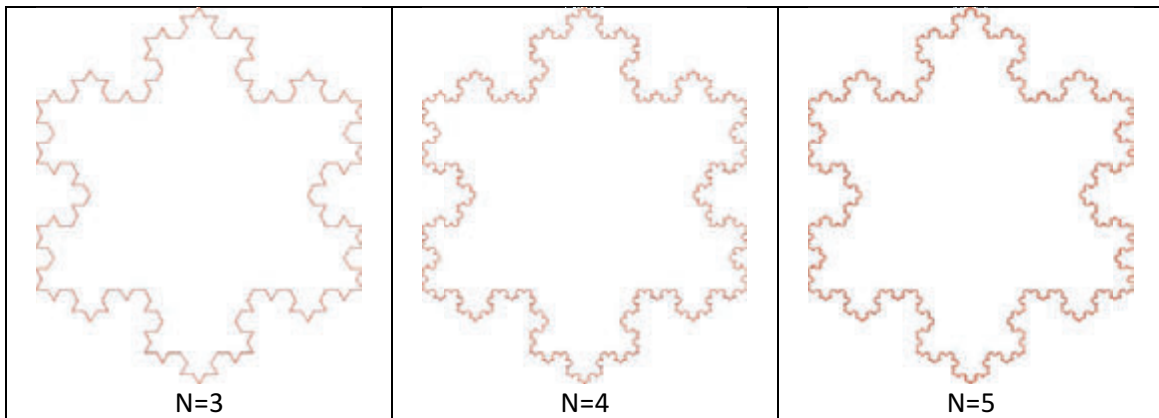


Table 5.1: Various fractals

Programming Example 1: Simple recursion – adding integers.

The following program uses recursion to create the method `addIntUpTo(n)` for $n > 0$ that will add all numbers from and including n down to 1. For example, if `addIntUpTo(4)` is called, the result would be: $4 + 3 + 2 + 1 = 10$

```
method addIntUpTo(n)
  if (n == 1) then
    return 1
  else
    return n + addIntUpTo(n-1)
  end if
end method
```

This method is a recursive function since it calls itself. On each call, the argument is reduced by one (every time `addIntUpTo` is called, its argument is $n-1$). $n-1$ calls are made until the terminating condition $n = 1$ is met.

Programming Example 2: Example of recursion.

What is going to be the output of the following algorithm?

```
method foo(n)
  if (n <= 1) then
    return 1
  else
    return foo(n-1) + foo(n-2)
  end if
end method

output foo(5)
```

Answer: 8

Programming Example 3: Example of recursion.

What is going to be the output of the following algorithm?

```
method foo(n, m)
  if (n <= 1) OR (m <= 1) then
    return 2
```

```

    else
        return foo(n-1, m) + foo(n, m-2)
    end if
end method

output foo(5,4)

```

Answer: 30

Programming Example 4: Example of recursion.

What is going to be the output of the following algorithm?

```

method foo(n, m)
    output "value of n=", n, "value of m =", m
    if (n <= 1) OR (m<=1) then
        return 2
    else
        return foo(n-1, m-n)+foo(n, m-2)
    end if
end method

output "Output is", foo(3,2)

```

Answer:

```

value of n= 3 value of m = 2
value of n= 2 value of m = -1
value of n= 3 value of m = 0
Output is 4

```

Programming Example 5: Example of recursion.

What is going to be the output of the following algorithm?

```

method Foo(X,Y)
    if X < Y then
        return Foo(X+1,Y-2)
    else if X = Y then
        return 2*Foo(X+2,Y-3)-3
    else
        return 2*X+3*Y
    end if
end method

output "Output is", Foo(3,12)

```

Answer:

```

Output is 47

```


Abstract data structures

5.1.4 – 5.1.5 Two dimensional arrays

Exit skills. Students should be able to¹:

Describe the characteristics of two-dimensional arrays.
Construct algorithms using two-dimensional arrays.

A one-dimensional array should be considered as a single line of elements. However, in many cases, data comes in the form of a data table. Each element in a 2D array must be of the same type, either a primitive or object type. Take, for example, five exam scores of a student, as a data record, and represent it as a row of information. The data records for ten students, would then be a table of 10 rows. Below is the visualization of this collection of data:

A lot of information and examples of two-dimensional arrays can be found in the book Core Computer Science for the IB Diploma Program³.

		Exam 1	Exam 2	Exam 3	...	Exam 5
		Index 0	Index 1	Index 2		Index 4
Student 1	Index 0	98	68	65		67
Student 2	Index 1	77	77	88		90
...
Student 10	Index 9	88	86	90		81

Table 5.2: Two dimensional array Scores

2D arrays are indexed by two subscripts. The indices must be integers. The first one refers to the row, while the second to the column. `Scores[1][1]` refers to Exam 2 of the second student. Its value is 77.

Programming Example 6: Two dimensional array (exam scores).

```
//This program will use the array Scores which is a 2D ARRAY.
//It will print the contents of the array.
//5 students with 5 exams each
Scores =
[[98,68,65,73,67],
 [77,77,88,78,90],
 [53,63,74,85,72],
 [77,77,68,78,91],
 [88,86,90,56,81]]
STUDENT = 0
EXAM = 0
loop STUDENT from 0 to 4
  output STUDENT +1, "Student"
  loop EXAM from 0 to 4
    output "----", "Exam ", EXAM+1, Scores[STUDENT][EXAM]
  end loop
end loop
```

OUTPUT

```
1 Student
---- Exam 1 98
---- Exam 2 68
---- Exam 3 65
---- Exam 4 73
---- Exam 5 67
2 Student
---- Exam 1 77
---- Exam 2 77
---- Exam 3 88
---- Exam 4 78
---- Exam 5 90
3 Student
---- Exam 1 53
---- Exam 2 63
---- Exam 3 74
---- Exam 4 85
---- Exam 5 72
4 Student
---- Exam 1 77
---- Exam 2 77
---- Exam 3 68
---- Exam 4 78
---- Exam 5 91
5 Student
---- Exam 1 88
---- Exam 2 86
---- Exam 3 90
---- Exam 4 56
---- Exam 5 81
```

Programming Example 7: Finds if a grade has the number 5 as its last digit.

```
Scores =
[[98,68,65,73,67],
 [77,77,88,78,90],
 [53,65,74,85,72],
 [77,77,68,78,91],
 [88,86,90,56,81]]
STUDENT = 0
EXAM = 0
loop STUDENT from 0 to 4
  output STUDENT +1, "Student"
  loop EXAM from 0 to 4
    if (Scores[STUDENT][EXAM] mod 10 = 5) then
      output "----", "Exam ", EXAM+1, Scores[STUDENT][EXAM]
    end if
  end loop
end loop
```

OUTPUT

```

1 Student
---- Exam 3 65
2 Student
3 Student
---- Exam 2 65
---- Exam 4 85
4 Student
5 Student

```

Programming Example 8: Finds and outputs the number of “8”s each score contains. It also outputs the total number of appearance of digit “8”.

```

Scores =
[[98,68,65,73,67],
[77,77,88,78,90],
[77,77,88,78,91],
[88,86,90,56,81]]
STUDENT = 0
EXAM = 0
TCOUNTER = 0
loop STUDENT from 0 to 3
    output STUDENT +1, "Student"
    loop EXAM from 0 to 4
        X = 1
        COUNTER = 0
        X = Scores[STUDENT][EXAM]
        loop while X>0
            if X mod 10 = 8 then
                COUNTER = COUNTER + 1
                TCOUNTER = TCOUNTER +1
            end if
            X = div (X, 10)
        end while
        output "----", "The grade of exam ", EXAM+1, "has",
COUNTER, "eight(s)"
    end loop
end loop
output " A total of ", TCOUNTER, "eights appear in all grades"

```

OUTPUT:

```

1 Student
---- The grade of exam 1 has 1 eight(s)
---- The grade of exam 2 has 1 eight(s)
---- The grade of exam 3 has 0 eight(s)
---- The grade of exam 4 has 0 eight(s)
---- The grade of exam 5 has 0 eight(s)
2 Student
---- The grade of exam 1 has 0 eight(s)
---- The grade of exam 2 has 0 eight(s)
---- The grade of exam 3 has 2 eight(s)
---- The grade of exam 4 has 1 eight(s)
---- The grade of exam 5 has 0 eight(s)
3 Student
---- The grade of exam 1 has 0 eight(s)

```

```

---- The grade of exam 2 has 0 eight(s)
---- The grade of exam 3 has 2 eight(s)
---- The grade of exam 4 has 1 eight(s)
---- The grade of exam 5 has 0 eight(s)
4 Student
---- The grade of exam 1 has 2 eight(s)
---- The grade of exam 2 has 1 eight(s)
---- The grade of exam 3 has 0 eight(s)
---- The grade of exam 4 has 0 eight(s)
---- The grade of exam 5 has 1 eight(s)
A total of 12 eights appear in all grades

```

5.1.6 – 5.1.7 Stacks

Exit skills. Students should be able to ¹ :
Describe the characteristics and applications of a stack. Construct algorithms using the access methods of a stack. Trace algorithms that use stacks.

Characteristics

A stack stores a set of elements in a particular order and allows access only to the last item inserted. Items are retrieved in the reverse order in which they are inserted. The stack is a Last-In, First-Out data (LIFO) structure. The elements of a stack may be numbers, Boolean values, characters, objects, arrays, strings, etc.

Stacks utilize three methods:

1. **push()**. Pushes an item onto a stack.
2. **pop()**. Removes and returns the last item entered in the stack.
3. **isEmpty()**. Tests if a stack is empty. It will return true if stack contains no elements.

Suppose we want to add the elements 5, 4, 3 in a stack named Numbers. The following diagram explains this situation:

<div> <div></div> <div></div> <div></div> </div>	Stack is initially empty.
<div> <div></div> <div></div> <div>5</div> </div>	Numbers.push(5) 5 was added to the stack.
<div> <div></div> <div>4</div> <div>5</div> </div>	Numbers.push(4) 4 was added to the stack

3	Numbers.push(3) 3 was added to the stack
4	
5	

Suppose we want to remove all the elements from the stack.

The following example presents this situation:

3	Stack contains 3 numbers.
4	
5	
	X = Numbers.pop() Top element was removed from the stack. This element was the number 3. 3 was assigned to variable x
4	
5	
	X = Numbers.pop() Top element was removed from the stack. This element was the number 4. 4 was assigned to variable x
5	
	X = Numbers.pop() Top element was removed from the stack. This element was the number 5. 5 was assigned to variable x . Stack is empty.

Applications

- The back button of a web browser uses a stack to function. Every time a URL is visited it is stored on a stack. The last address that was visited is on the top of the stack. The first address that was visited during the current web session is on the bottom of the stack. If one selects the Back button, he/she begins to visit the previous pages they have visited in reverse order.
- Microprocessors usually use a stack to handle methods. Suppose a method, **A**, which returns an integer, with parameters **b** and **c** of type integer, is called. In Java this would look like this:

```
int c = A(a, b);
```

The method header would look like this:

```
public static int A(int b, int c)
```

The method body should look like this:

```
{method body
  return r}
```

When **A** is called, its return address, as well as **b** and **c** are pushed onto the microprocessors stack. When the method returns **r**, the return address and the parameters (arguments) are popped off the stack. The overall process is more complicated, but further explanation is beyond the scope of this book.

- Recursive methods also utilize the system stack to keep track of each recursive call. This block of memory is used to store temporary data required for program execution. The calls are nested inside each other. Initially, all recursive calls are unfolded and pushed onto the stack, until the base case is reached and then all recursive calls are popped from the stack, when necessary. In the following example the left-hand code fragment will return 4. The right-hand code fragment will generate a run time error because the recursive program will never reach the terminating condition.

<pre> public class Rec_Demo { public static int question(int n) { if (n <= 0) return -2; else return (question(n-100)+3); } public static void main(String[] a) { int l = question(111); System.out.println("l= "+l); } } </pre>	<pre> public class Rec_Demo { public static int question(int n) { if (n <= 0) return -2; else return (question(n+100)+3); } public static void main(String[] a) { int l = question(111); System.out.println("l= " + l); } } </pre>
<p>OUTPUT</p> <p>l=4</p>	<p>OUTPUT</p> <p>java.lang.StackOverflowError:null</p>

Algorithms

Programming Example 9: Use of a stack, an array and a collection.

```

//==== Reverse and store =====
// This algorithm uses an array, a stack and a collection.
// It reads names from the array, reverses them,
// using the stack, and stores the contents of the stack
// inside the collection.
// =====

NAMES = ["Kostas", "Markos", "Anna", "Mary", "Takis"]
NAMES_C = new Collection()
STACK_NAMES = new Stack()
I = 0

loop I from 0 to 4
    STACK_NAMES.push(NAMES[I])
end loop

output "Add names in the collection:"
output "======"
loop while NOT(STACK_NAMES.isEmpty())
    NAME = STACK_NAMES.pop()
    NAMES_C.addItem(NAME)
    output NAME, "was entered in the collection"
end loop

```

```

output ""
output "Names stored in the collection:"
output "=====
loop while NAMES_C.hasNext()
    output NAMES_C.getNext()
end loop

```

OUTPUT

```

Add names in the collection:
=====
Takis was entered in the collection
Mary was entered in the collection
Anna was entered in the collection
Markos was entered in the collection
Kostas was entered in the collection

Names stored in the collection:
=====
Takis
Mary
Anna
Markos
Kostas

```

Programming Example 10: Use of stacks to implement the Towers of Hanoi algorithm

The following program uses 4 stacks to solve the Towers of Hanoi problem:

```

//Declaration and initialization of variables
SPa = new Stack() //a new stack
SPb = new Stack() //a new stack
SPc = new Stack() //a new stack
SPd = new Stack() //a new stack
daa = new Array() //auxiliary array to use in display method
dbb = new Array() //auxiliary array to use in display method
dcc = new Array() //auxiliary array to use in display method
PEGS = [SPa, SPb, SPc, SPd] //an array of four stacks
I = 0
a = 1
b = 2
c = 3
t = 0
n = 0
m = 0
da = ""
db = ""
dc = ""
NUM = 5 //the number of disks
f1 = " "
f2 = " "
f3 = " "

TowersofHanoi(NUM) //the number of disks

```

```

//The following method is the starting point
//of the program
method TowersofHanoi(n)
    loop I from 0 to n-1
        m = n-I
        PEGS[1].push(m)
    end loop
    display()
    move(n,1,2,3)
end method

//This is a recursive method used to solve the problem
method move(n, a, b, c)
    if n>0 then
        move(n-1, a, c, b)
        t = PEGS[a].pop()
        PEGS[c].push(t)
        display()
        move(n-1, b, a, c)
    end if
end method

//The following method is used to visualize the
//pegs and the disks. Three auxiliary arrays are
//used so as to display the contents of each stack.
method display()
    output ""
    output " | A | B | C |"
    loop I from 0 to NUM-1
        daa[I] = PEGS[1].pop()//put the elements of the stack to an array
        dbb[I] = PEGS[2].pop()//put the elements of the stack to an array
        dcc[I] = PEGS[3].pop()//put the elements of the stack to an array
    end loop

    loop I from 0 to NUM-1
        da = daa[I]
        db = dbb[I]
        dc = dcc[I]
        f1 = String(da)//covert to string
        if f1 == "null" then
            f1="-"
        end if
        f2 = String(db) //covert to string
        if f2 == "null" then
            f2="-"
        end if
        f3 = String(dc) //covert to string
        if f3 == "null" then
            f3="-"
        end if
        output " | ", f1, " | ", f2, " | ", f3, " | "
    end loop

    loop I from 0 to NUM-1
        m = NUM-1-I
        PEGS[1].push(daa[m]) //put the elements of the array daa back to the stack
        PEGS[2].push(dbb[m]) //put the elements of the array dbb back to the stack
        PEGS[3].push(dcc[m]) //put the elements of the array dcc back to the stack
    end loop
end method

```


FORMATED OUTPUT

1	A 1 2 3 4 5	B - - - - -	C - - - - -	2	A 2 3 4 5 -	B - - - - - -	C 1 - - - - -	3	A 3 4 5 - -	B 2 - - - - -	C 1 - - - - -	4	A 3 4 5 - -	B 1 2 - - - -	C - - - - - -
5	A 4 5 - - -	B 1 2 - - -	C 3 - - - -	6	A 1 4 5 - -	B 2 - - - - -	C 3 - - - - -	7	A 1 4 5 - -	B - - - - - -	C 2 3 - - - -	8	A 4 5 - - -	B - - - - -	C 1 2 3 - -
9	A 5 - - - -	B 4 - - - -	C 1 2 3 - -	10	A 5 - - - -	B 1 4 - - -	C 2 3 - - -	11	A 2 5 - - -	B 1 4 - - -	C 3 - - - -	12	A 1 2 5 - -	B 4 - - - -	C 3 - - - -
13	A 1 2 5 - -	B 3 4 - - -	C - - - - -	14	A 2 5 - - -	B 3 4 - - -	C 1 - - - -	15	A 5 - - - -	B 2 3 4 - -	C 1 - - - -	16	A 5 - - - -	B 1 2 3 4 -	C - - - - -
17	A - - - - -	B 1 2 3 4 -	C 5 - - - -	18	A 1 - - - -	B 2 3 4 - -	C 5 - - - -	19	A 1 - - - -	B 3 4 - - -	C 2 5 - - -	20	A - - - - -	B 3 4 - - -	C 1 2 5 - -
21	A 3 - - - -	B 4 - - - -	C 1 2 5 - -	22	A 3 - - - -	B 1 4 - - -	C 2 5 - - -	23	A 2 3 - - -	B 1 4 - - -	C 5 - - -	24	A 1 2 3 - -	B 4 - - - -	C 5 - - -
25	A 1 2 3 - -	B - - - - -	C 4 5 - - -	26	A 2 3 - - -	B - - - - -	C 1 4 5 - -	27	A 3 - - - -	B 2 - - - -	C 1 4 5 - -	28	A 3 - - - -	B 1 2 - - -	C 4 5 - - -
29	A - - - - -	B 1 2 - - -	C 3 4 5 - -	30	A 1 - - - -	B 2 - - - -	C 3 4 5 - -	31	A 1 - - - -	B - - - - -	C 2 3 4 5 -	32	A - - - - -	B - - - - -	C 1 2 3 4 5

5.1.8 – 5.1.9 Queues

Exit skills. Students should be able to ¹ :
Describe the characteristics and applications of queues. Construct algorithms using the access methods of queues. Trace algorithms that use queues.

Characteristics of queues



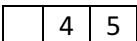
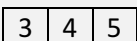
A queue stores a set of elements in a particular order and allows access only to the first item inserted. Items are retrieved in the order in which they are inserted. The queue is a First-In, First-Out (FIFO) data structure. The elements of a queue may be numbers, Boolean values, characters, objects, arrays, etc.

Queues utilize three methods:

1. **enqueue()**. Puts an item into the end of the queue.
2. **dequeue()**. Removes and returns the first item entered in the queue.
3. **isEmpty()**. Tests if a queue is empty. It will return true if queue contains no elements.

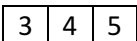

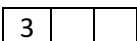

Suppose we want to add the elements 5, 4 and 3 in a queue named **Numbers**.

The following example presents this situation:

	Queue initially empty
	Numbers.enqueue (5) 5 was added to the queue
	Numbers.enqueue (4) 4 was added to the queue
	Numbers.enqueue (3) 3 was added to the queue

Suppose we want to remove all the elements from the queue.

The following example presents this situation:

	Queue contains 3 numbers
	X = Numbers.dequeue () First element was removed from the queue. This element was the number 5. 5 was assigned to variable X .
	X = Numbers.dequeue () First element was removed from the queue. This element was the number 4. 4 was assigned to variable X .
	X = Numbers.dequeue () First element was removed from the queue. This element was the number 3. 3 was assigned to variable X . Queue is empty.

Applications of queues

- Queues are used to model physical queues, such as people waiting in line at a supermarket checkout.
- The print queue displays the documents that are waiting to be printed. These documents will follow the first-sent first-print policy.
- When sending data over the internet, various data packets wait in a queue to be sent.
- A server usually serves various requests. In most cases these requests are stored in a queue. The first-come first-served request procedure is followed.

Algorithms that use queues

Programming Example 11: Use of a queue and arrays.

A small school uses two buses to transport students. As soon as the buses arrive, all students enter a queue and a teacher uses a registry to check which students are present. The following algorithm uses two arrays to represent the school buses, a queue to represent the queue, and an array to represent the registry:

```
BUS1 = ["Roger", "John", "Nikos", "Marion", "Hellen"]
BUS2 = ["Nora", "Bill", "Eliza", "Takis", "Alex"]
REGISTRY = ["Alex", "John", "Elina", "Nikos", "Leo", "Marion",
"Hellen", "Nora", "Bill", "Eliza", "Takis", "Roger"]
STUDENTS = new Queue() //Queue for Students
A = ""
I = 0
FOUND = 1

//copy students from BUS1
loop I from 0 to 4
    STUDENTS.enqueue(BUS1[I])
end loop

//copy students from BUS2
loop I from 0 to 4
    STUDENTS.enqueue(BUS2[I])
end loop

loop while NOT(STUDENTS.isEmpty())
    A = STUDENTS.dequeue()
    loop I from 0 to 12
        if REGISTRY[I] = A then
            FOUND = 1
        end if
    end loop
    if FOUND = 1 then
        output A, "is not absent"
    end if
end loop
```

OUTPUT

Roger is not absent
John is not absent
Nikos is not absent
Marion is not absent
Hellen is not absent
Nora is not absent
Bill is not absent
Eliza is not absent
Takis is not absent
Alex is not absent

Programming Example 12: Use of queues, arrays and a collection.

A supermarket has two express cashiers. The array **CASHIER1** contains the customers that enter the queue **CUSTOMER1**, while the array **CASHIER2** contains the customers that enter the queue **CUSTOMER2**. The **TIME** collection stores the names of the customers that waited more than 60 secs, counting from the moment that their turn to be served had come. The supermarket administration wishes to minimize the waiting for these two express cashiers. A questionnaire is sent by email, from the administration of the supermarket, to the customers stored in the collection **TIME** to understand why this situation took place. A message that outputs the overall slower express cashier is output at the end of the day.

```
CASHIER1 = ["Roger", "John", "Nikos", "Marion"]
CASHIER2 = ["Nora", "Bill", "Eliza", "Takis"]
CUSTOMER1 = new Queue()    //Queue for CUSTOMER1
CUSTOMER2 = new Queue()    //Queue for CUSTOMER2
TIME = new Collection()
A = ""
B = 0
C1 = 0
C2 = 0
I = 0
D1 = 0
D2 = 0
TOT_B = 0
TOT_C = 0
FOUND = 1

//copy CUSTOMER1 from CASHIER1
loop I from 0 to 3
    CUSTOMER1.enqueue(CASHIER1[I])
end loop

//copy CUSTOMER2 from CASHIER2
loop I from 0 to 3
    CUSTOMER2.enqueue(CASHIER2[I])
end loop

loop while NOT(CUSTOMER1.isEmpty())
    D1 = (CUSTOMER1.dequeue())
    C1 = Math.floor((Math.random() * 100) + 1) // use of random function
to generate random times between 1 sec to 100 sec
```

```

    if C1>60 then //only customers waiting more than 60 secs enter the
collection
        TIME.addItem(D1)
    end if
    TOT_B = TOT_B + C1
end loop

loop while NOT(CUSTOMER2.isEmpty())
    D2 = (CUSTOMER2.dequeue())
    C2 = Math.floor(Math.random() * 100) + 1)
    if C2>60 then
        TIME.addItem(D2)
    end if
    TOT_C = TOT_C + C2
end loop

TIME.resetNext()
loop while TIME.hasNext()
    output TIME.getNext()
end loop

if TOT_B > TOT_C then//outputs the slower cashier
    output "CASHIER1 is slower"
else
    output "CASHIER2 is slower"
end if

```

A POSSIBLE OUTPUT

```

Roger
John
Bill
Eliza
CASHIER2 is slower

```

5.1.10 Arrays as static stacks and queues

Exit skills. Students should be able to¹:

Explain push and pop operations, and test on empty/full stack.
Explain enqueue and dequeue operations, and test on empty/full queue.

Algorithms to implement stacks using an array

Programming Example 13: Implementation of stack using an array.

The program starts with an array of 10 elements. The methods used are the following:

push()

this method is used to add elements in the stack. Inserting an element increments **high** by 1 and adds the element in this array position. The **high** is incremented before the insertion of the new item takes place.

pop()

this method returns the value of the top element and then decrements **high**. It serves to remove the top element from the stack. The item removed actually remains in the array but is inaccessible.

isempty()

it is based on the **high** variable. It returns **true** (1) if the stack is empty.

isfull()

it is based on the **high** variable. It returns **true** (1) if the stack is full.

size()

it is based on the **high** variable. It returns the number of elements stored in the stack.

```
s_array = new Array()
s_array = [0,0,0,0,0,0,0,0,0,0]
maxsize = 10
high = -1
n = 0

pop()
push(1)
push(2)
push(7)
push(8)
pop()
push(9)
push(10)
push(9)
push(33)
push(29)
push(11)
push(49)
push(10)

output "-----"
output "high = ", high
output "-----"
output "s_array contains :", s_array
output "-----"
output "size of stack = ", size()

output
"/////////////////////"
output "stack contents display and
removal"
output
"/////////////////////"
loop while isempty() = 0
    n = pop()
    output n
end loop
```

OUTPUT:

Message: stack is empty

Message: stack is full

high = 9

s_array contains :

1,2,7,9,10,9,33,29,11,49

size of stack = 10

////////////////////

stack contents display and removal

////////////////////

49

11

29

33

9

10

9

7

2

1

////////////////////

Explanation:

This algorithm uses the s_array to implement a stack. The methods used are **push()**, **pop()**, **isempty()**, **isfull()** and **size()**.

When this algorithm starts an array of

```

output
"/////////////////////////////////////"

method push(n)
  if (isfull() == 1) then
    output "Message: stack is full"
  else
    high = high + 1
    s_array[high] = n
  end if
end method

method pop()
  if (isempty() == 1) then
    output "Message: stack is empty"
  else
    high = high - 1
    return s_array[high+1]
  end if
end method

method isempty()
  if (high == -1) then
    return 1
  else
    return 0
  end if
end method

method isfull()
  if (high == maxsize-1) then
    return 1
  else
    return 0
  end if
end method

method size()
  return high+1
end method

```

ten elements is created.

maxsize variable is used to hold the maximum stack size, **high** variable is used to point the array position that is the top of the stack.

The first **pop()** instruction generates a "Message: stack is empty" output.

push(1), push(2), push(7), push(8) instructions add four elements in the stack.

pop() instruction removes 1 from the stack.

push(9), push(10), push(9), push(33), push(29), push(11), push(49) instructions add 7 elements in the stack. The stack is now full.

push(10) instruction causes "Message: stack is full " message to be displayed.

Instruction "output "high = ", high" prints the number 9 which is the array position used to point the end of the queue.

Instruction "output s_array contains :", s_array" outputs the contents of the actual array used. The numbers 1,2,7,9,10,9,33,29,11,49 are printed.

The size of stack is 10

After a "stack contents display and removal" message a loop that removes and outputs all elements of stack is used. 49,11,29,33,9,10,9,7,2,1 are printed.

Programming Example 14: Convert integer to binary using a stack.

```
//This algorithm uses a stack to convert an integer to its binary
equivalent

//Declaration of variables
s_array = new Array()
s2_array = new Array()
s_array = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
maxsize = 10
high = -1
x = 0
y = 0
n = 0
t = 0
dig_value = 0
number = 123

output "Convert number ", number

//Call method convert_in_binary
convert_to_binary(number) //max number is 1023

//Use of an auxiliary array to properly output the result
output " "
output "Final result"
loop a from 0 to 9
    s2_array[a] = s_array[9-a]
end loop
output s2_array

method convert_to_binary(x)
    output "Calculations"
    loop while x > 0
        y = x mod 2
        push(y) //use of push method
        x = div(x,2) //division of x over 2
    end loop
    //the next loop will use the isempty method
    loop while isempty() = 0
        t = pop() //use of pop method
        dig_value = Math.pow(2, (high+1)) //2^(high+1)
        output "Binary digit number", high+1, "(", dig_value, ")", "is", t
    end loop
end method

method push(n)
    if (isfull() == 1) then
        output "Message: stack is full"
    else
        high = high + 1
        s_array[high] = n
    end if
end method
```



```

method pop()
  if (isempty() == 1) then
    output "Message: stack is empty"
  else
    high = high - 1
    return s_array[high+1]
  end if
end method

method isempty()
  if (high == -1) then
    return 1
  else
    return 0
  end if
end method

method isfull()
  if (high == maxsize-1) then
    return 1
  else
    return 0
  end if
end method

```

OUTPUT

Calculations

Binary digit number 6 (64) is 1
 Binary digit number 5 (32) is 1
 Binary digit number 4 (16) is 1
 Binary digit number 3 (8) is 1
 Binary digit number 2 (4) is 0
 Binary digit number 1 (2) is 1
 Binary digit number 0 (1) is 1

Final result

0,0,0,1,1,1,1,0,1,1

Algorithms to implement queues using an array

Programming Example 15: Implementation of queue using an array.

INDEX	0	1	2	3	4	5	6
	5	3	9	7	8		
	FRONT					REAR	

When using an array to implement a queue, insertion takes place at the **REAR** index, while deletion takes place at the **FRONT** index only. At the beginning both **FRONT** and **REAR** are 0. When entering the first element, **FRONT** remains 0, while **REAR** becomes 1. When entering another element, **FRONT** again remains 0, while **REAR** becomes 2. When entering yet another element, **FRONT** remains 0 and **REAR** becomes 3. If we remove an element, **FRONT** becomes 1 and **REAR** remains 3. If we remove another element, **FRONT** becomes 2 and **REAR** remains 3. If we remove yet another element, both **FRONT** and **REAR** become 0, since the queue is empty.

The following algorithm implements this approach. Unfortunately, this array-based implementation is tricky. It works well when entering elements and then removing them all before entering new elements again. This is not the case when adding and deleting data in a random order since the end of the array will eventually be reached and an out-of-bounds exception will be raised.

```
q_array = new Array()
q_array = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
FRONT = 0
REAR = 0
SIZE = 10
n = 0
dequeue()
enqueue(71)
enqueue(1)
enqueue(2)
enqueue(112)
enqueue(14)
enqueue(52)
enqueue(67)
enqueue(14)
enqueue(52)
enqueue(62)
dequeue()
dequeue()
dequeue()
dequeue()
dequeue()
dequeue()
enqueue(61)

output "Queue contents display"
output "-----"
if (FRONT == REAR) then
    output "Message: queue is empty"
else
    loop I from FRONT to REAR-1
        n = q_array[I]
        output n
    end loop
end if
output "-----"
```

```

method enqueue(N)
  if REAR == SIZE then
    output "Message: queue is full"
  else
    q_array[REAR] = N
    REAR = REAR + 1
  end if
end method

method dequeue()
  if FRONT == REAR then
    output "Message: queue is empty"
  else
    N = q_array[FRONT]
    if (FRONT+1 == REAR) then
      REAR = 0
      FRONT = 0
    else
      FRONT = FRONT + 1
    end if
  end if
end method

```

OUTPUT

```

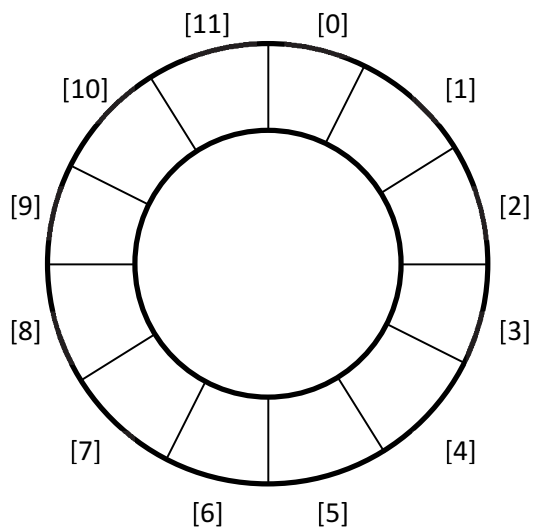
Message: queue is empty
Message: queue is full
Queue contents display
-----
67
14
52
62
-----

```

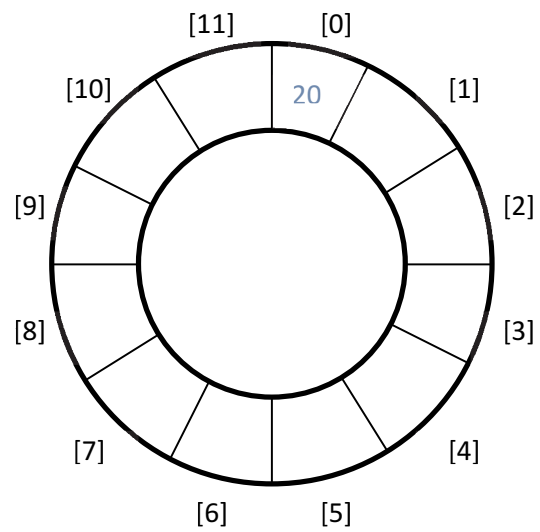
As we can see the queue contains only 4 elements. Although the array can hold 10 elements the **FRONT** is now 7 and the **REAR** is 10 so **enqueue(61)** will generate the **queue is full** message. This situation can be solved by using a circular implementation of a queue.

Algorithms to implement a circular queue using an array

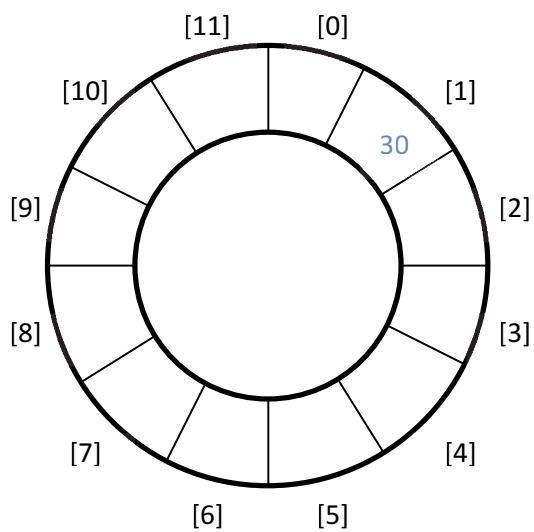
The problem with the previous implementation is that the new elements are added to successively higher-numbered positions in the array. When elements of the queue are deleted, the **FRONT** index increases and this process continues until the queue runs out of space. The array might have free positions at the indices that are smaller than the **FRONT** index, but these positions are unusable. The following circular implementation of a queue solves this problem:



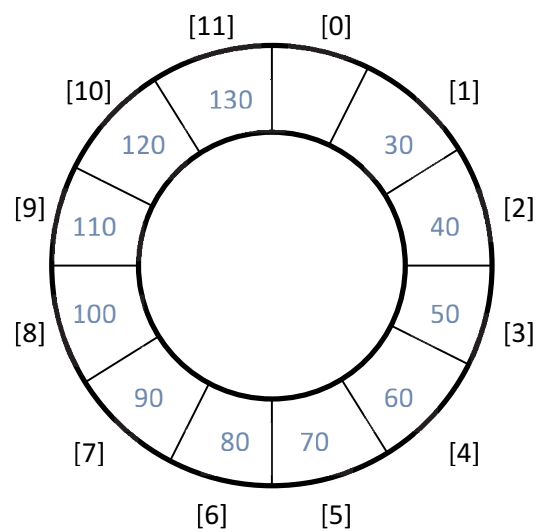
FRONT = 0
REAR = -1



ADD 20
FRONT = 0
REAR = 0



DELETE 20
ADD 30
FRONT = 1
REAR = 1



ADD
30,40,50,60,70,80,90,100,110,120,130
FRONT = 1
REAR = 11

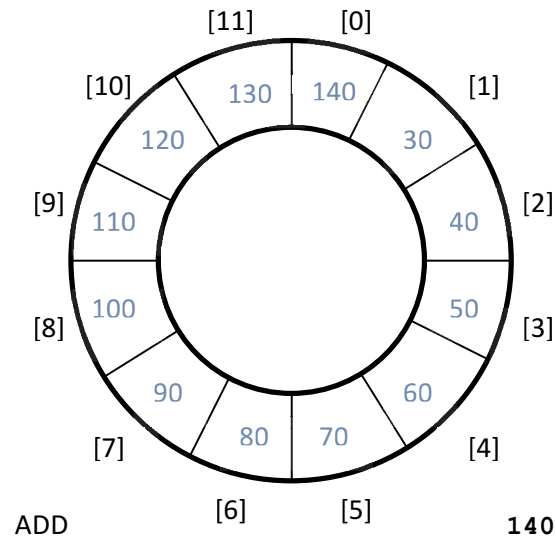


Figure 5.3: Explanation of the operation of a circular queue

The following algorithm starts with an array of 10 elements. The methods used are the following:

enqueue ()

This method is used to add elements to the queue. Inserting an element increments **rear** by 1 and inserts the element in the new array position where **rear** points to. If **rear** is at the end (top) of the array, then **rear** should be set to -1 before the addition of the element takes place. This means that a wraparound takes place and the next element will be placed at the start (bottom) of the array. Finally, the variable that holds the number of elements, **nelements**, is incremented by 1.

dequeue ()

This method is used to remove elements from the queue. A temporary variable, **temp**, is used to hold the value of **front**. **front** is then incremented by 1. If **front** equals to the array length, then a wraparound takes place and 0 is assigned to **front**. Finally, the variable that holds the number of elements, **nelements** is decremented by 1.

isempty ()

This is based on the **nelements** variable. It returns **true** (1) if the queue is empty.

isfull ()

This is based on the **nelements** variable. It returns **true** (1) if the queue is full.

size ()

This is based on the **nelements** variable. It returns the number of elements stored in the queue.

```

q_array = new Array()
q_array = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
//you can replace the previous two
lines
//with q_array = new Array(10)
maxsize = 10
front = 0
rear = -1
nelements = 0
n = 0
dequeue()
enqueue(1)
enqueue(2)
enqueue(7)
enqueue(8)
dequeue()
enqueue(9)
enqueue(10)

output "front = ", front
output "rear = ", rear
output "q_array contains :", q_array
output "-----"
output "size = ", size()
output "-----"

output "queue contents display and
removal"
output "-----"
loop while isempty() = 0
    n = dequeue()
    output n
end loop
output "-----"

method enqueue(n)
    if isfull() = 1 then
        output "Message: queue is full"
    else
        if (rear == maxsize-1) then
            rear = -1
        end if
        rear = rear + 1
        q_array[rear] = n
        nelements = nelements + 1
    end if
end method

method dequeue()
    if isempty() = 1 then
        output "Message: queue is empty"
    else
        temp = q_array[front]
        front = front + 1
        if (front==maxsize) then
            front = 0
        end if
        nelements = nelements - 1
        return temp
    end if
end method

```

Output:

```

Message: queue is empty
front = 1
rear = 5
q_array contains :
1,2,7,8,9,10,0,0,0,0
-----
size = 5
-----
queue contents display and
removal
-----
2
7
8
9
10
-----

```

Explanation:

This algorithm uses the **q_array** to implement a circular queue. The methods used are: **enqueue()**, **dequeue()**, **isempty()**, **isfull()** and **size()**.

When this algorithm begins, an array of ten elements is created. **maxsize** variable is used to hold the maximum queue size, the **front** variable is used to point to the start of the queue, the **rear** variable is used to point to the end of the queue and **nelements** is used to hold the total number of elements stored in the queue.

The first **dequeue()** instruction generates a "Message: queue is empty" output.

enqueue(1), **enqueue(2)**, **enqueue(7)**, **enqueue(8)** instructions add four elements in the queue.

dequeue() instruction removes 1 from the queue.

enqueue(9) and **enqueue(10)** add two elements to the queue.

Instruction **output "front = ", front** prints the number 1

<pre> end method method isempty() if (nelements == 0) then return 1 else return 0 end if end method method isfull() if (nelements == maxsize) then return 1 else return 0 end if end method method size() return nelements end method </pre>	<p>which is the array position used to point to the front of the queue.</p> <p>Instruction output "rear = ", rear prints the number 5 which is the array position used to point to the end of the queue.</p> <p>Instruction output "size = ", size() outputs size = 5, which is the size of the queue.</p> <p>Instruction output "q_array contains:", q_array outputs the contents of the actual array used. The numbers 1,2,7,8,9,10,0,0,0,0 are printed.</p> <p>After a "queue contents display and removal" message, a loop that removes and outputs all elements of queue is used. 2 7 8 9 10 are printed.</p>
---	--