Data Structures and Algorithms

- About the course (objectives, outline, recommended reading)
- Problem solving
- Notions of Algorithmics (growth of functions, efficiency, programming model, example analysis)
- Stacks, queues.

Data Structures and Algorithms

- Instructors:
  - Marius Joldoş (lectures),
    - Room D3, 71-73 Dorobanţilor St., Room 10, 28 G.Baritiu St.
    - Tel. 0264-401276, 0264-202370
    - Marius.Joldos@cs.utcluj.ro,
  - Iulia Costin (lab. sessions)
    - Iulia.Costin@cs.utcluj.ro
- Course web site:
  - http://users.utcluj.ro/~jim/DSA
- Schedule: as set in the published timetable

Objectives

- To get familiar with various ways to characterize properties of algorithms and data structures.
- To learn to compare and algorithms and data structures.
- To get familiar with standard algorithms and data structures so that you can find appropriate algorithms and data structures when they are available.
- To learn to design your own algorithms and data structures when no appropriate can be found in the literature and libraries.

Organization & interaction

- Lectures
  - Attend classes, ask questions
- Laboratory sessions
  - Do the tasks assigned to you
  - Criticize (constructively) the work – learn how to evaluate work
  - Deadlines MUST be met

Evaluation

- Activity during the semester – 40%
  - Assigned programming tasks
  - Presentations, etc.
- Written exams
  - Midterm, in class – 30% of written
    - After lecture 6?
  - Final, during winter session – 70% of written (theory + problem)
- There may be bonuses (<= 1 point)

Recommended books

- Aho, Hopcroft, Ullman. Data Structures and Algorithms, Addison-Wesley, 427 pages, 1987.
  - The classic book on the topic, still most valuable introductory book, though it misses a few topics. Uses “extended Pascal”.
- Cormen, Leiserson, Rivest, (Stein): Introduction to Algorithms. MIT Press / McGraw Hill, (2nd edition),1028 pages, 1990 (2002).
  - Written for all levels, with introductory chapters in discrete math and offering specialized topics in later chapters.
  - Uses pseudocode, which makes the presentation compact, easy to follow and “timeless”. Also useful as a reference book.

Recommended books

- Preiss. Data Structures and Algorithms with object-Oriented Design Patterns in C++, John Wiley and Sons, 660 pages, 1999.
  - An unusual algorithms and data structures book in that it stresses from the beginning the role of design patterns for the implementation of data structures as classes. An excellent source for good object-oriented design principles, with an accessible introduction to complexity analysis.
  - Available online (see course web site)
- Knuth. The Art of Computer Programing, Addison Wesley, three volumes, reprinted several times.
  - A classical book, extremely well written and documented by a Master of the field. Unfortunately uses the language of the hypothetical machine MIX for describing the algorithms which makes understanding more difficult.

Outline (preliminary)

- Introduction: algorithmics and asymptotics. Lists, stacks and queues
- Trees, binary trees, binary search trees
- Basic ADTs for sets: dictionary, hash tables, mappings, priority queues
- Advanced ADTs for sets: 2-3 trees, AVL trees, union-fiind sets
- Directed graph ADTs: single source shortest path, all pairs shortest path, traversals
- Undirected graphs: minimum cost spanning trees (MST), MST algorithms, traversals, graph matching
- Algorithm analysis and design techniques: divide and conquer, dynamic programming, greedy algorithms, backtracking, local search algorithms
- Sorting
- Data Structures and algorithms for external storage. B-trees.

Problem Solving

- First *construct an exact model* in terms of which we can express allowed solutions.
  - Finding such a model is already half the solution. Any branch of mathematics or science can be called into service to help model the problem domain.
- Once we have a suitable mathematical model, we can *specify a solution in terms of that model*.

Representation of real world objects

- Issues when dealing with the representation of real world objects in a computer program
  - how real world *objects are modeled* as mathematical entities,
  - the *set of operations* that we define over these mathematical entities,
  - how these entities are *stored in a computer's memory* (e.g. how they are aggregated as fields in records and how these *records are arranged* in memory, perhaps as arrays or as linked structures), and
  - the *algorithms* that are used to perform these operations.

Basic Notions

- **Model of Computation:** An abstract sequential computer, called a *Random Access Machine* or *RAM*. Uniform cost model.
- **Computational Problem:** A specification in general terms of *inputs* and *outputs* and the desired input/output relationship.
- **Problem Instance:** A particular collection of inputs for a given problem.
- **Algorithm:** A method of solving a problem which can be implemented on a computer. Usually there are many algorithms for a given problem.
- **Program:** Particular implementation of some algorithm.

Example of computational problem: sorting

- Statement of problem:
  - *Input:* A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$
  - *Output:* A reordering of the input sequence $\langle a'_1, a'_2, \ldots, a'_n \rangle$ so that $a'_i \leq a'_j$ whenever $i < j$
- Instance: The sequence <7, 5, 4, 10, 5>
- Algorithms:
  - Selection sort
  - Insertion sort
  - Merge sort
  - (many others)

Algorithm Design Algorithm

```
AlgorithmDesign(informal problem)
1   formalize problem (mathematically) [Step 0]
2   repeat
3           devise algorithm [Step 1]
4           analyze correctness [Step 2]
5           analyze efficiency [Step 3]
6           refine
7     until algorithm good enough
8   return algorithm
```

Analysis of algorithms

- Determine the amount of some **resource** required by an algorithm (usually depending on the **size** of the input).
- The resource might be:
  - running time
  - memory usage (space)
  - number of accesses to secondary storage
  - number of basic arithmetic operations
  - network traffic
- Formally, we define the running time of an algorithm on a particular input instance to be the number of computation steps performed by the algorithm on this instance.

Growth of functions

- Typically, problems become computationally intensive as the input size grows.
- We look at input sizes large enough to make only the order of the growth of the running time relevant for the analysis and comparison of algorithms.
- Hence we are studying the asymptotic efficiency of algorithms.

A General Portable Performance Metric

- *Informally,* time to solve a problem of size, n, $T(n)$ is $O(\log n)$ if $T(n) = c \log_2 n$
- *Formally:*
  - $O(g(n))$ is the set of functions, f, such that

    $$f(n) < c\, g(n)$$

    for some constant, $c > 0$, and $n > N$ (i.e. sufficiently large N)
  - Alternatively, we may write $\lim_{n \to \infty} \frac{f(n)}{g(n)} \leq c$

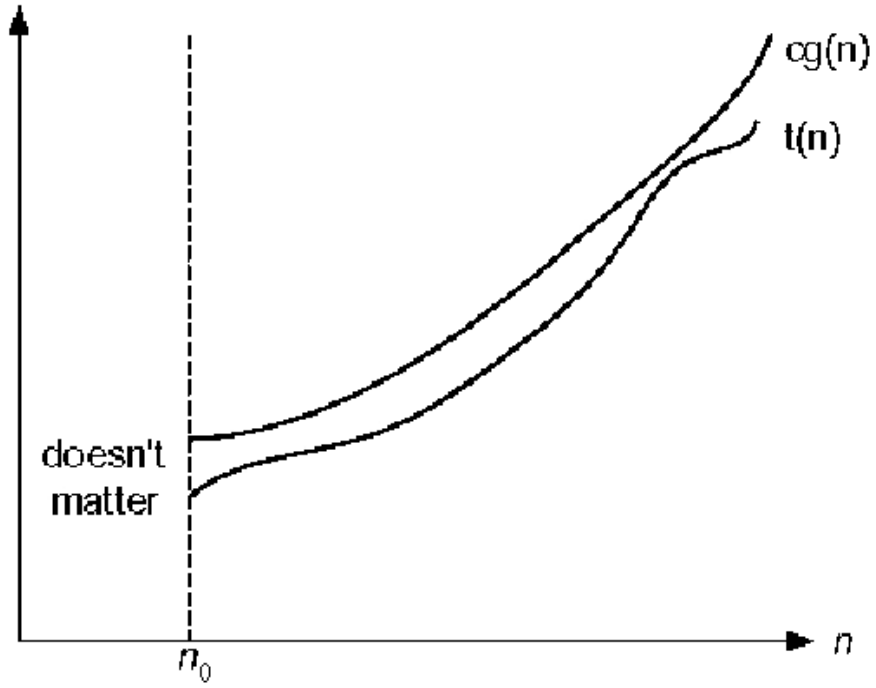
    and say **g *is an upper bound for* f**

Big-oh

- $O(g)$ is the set of functions that grow no faster than g.
- $g(n)$ describes the worst case behavior of an algorithm that is $O(g)$
- Examples:

  $n \lg n + n = O(n^2)$

  $\lg^k n = O(n)$ for all $k \in \mathbf{N}$

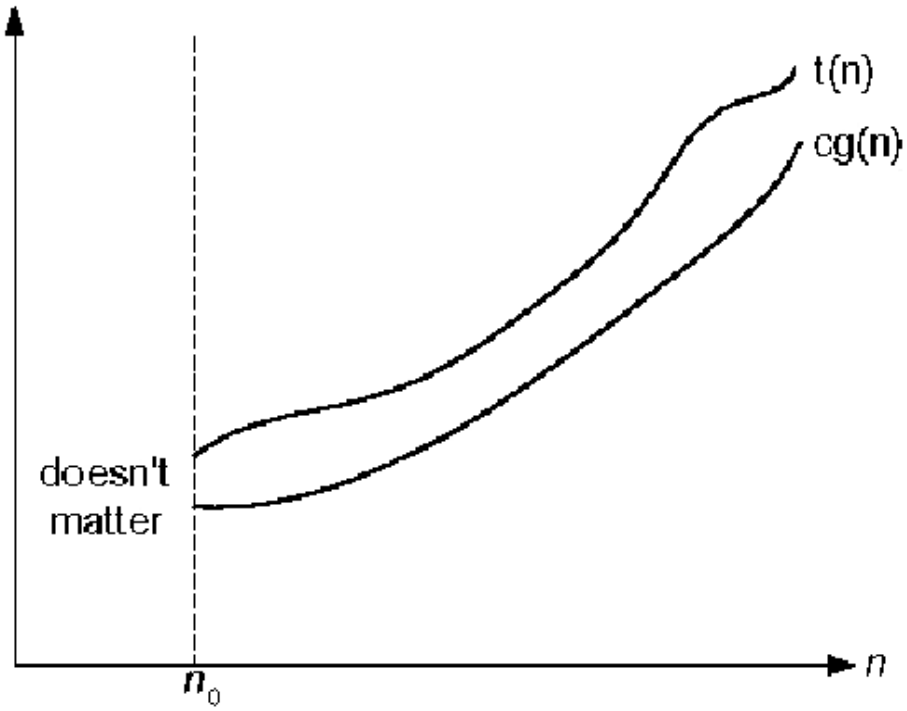


Big-omega

- $\Omega(g)$ is the set of functions, f, such that

  $$f(n) > c\, g(n)$$

  for some constant, c, *and* $n > N$
- $\Omega(g(n))$: class of functions $f(n)$ that grow *at least as fast* as $g(n)$
- $g(n)$ describes the best case behavior of an algorithm that is $\Omega(g)$



- Example:

  $a\,n^2 + b\,n + c = \Omega(n)$

  provided $a > 0$

Big-theta

- $\Theta(g(n))$: class of functions $f(n)$ that grow *at same rate* as $g(n)$
- Example:

  $n^2 / 2 - 3n = \Theta(n^2)$

We have to determine $c_1 > 0$, $c_2 > 0$, $n_0 \in \mathbf{N}$ such that:

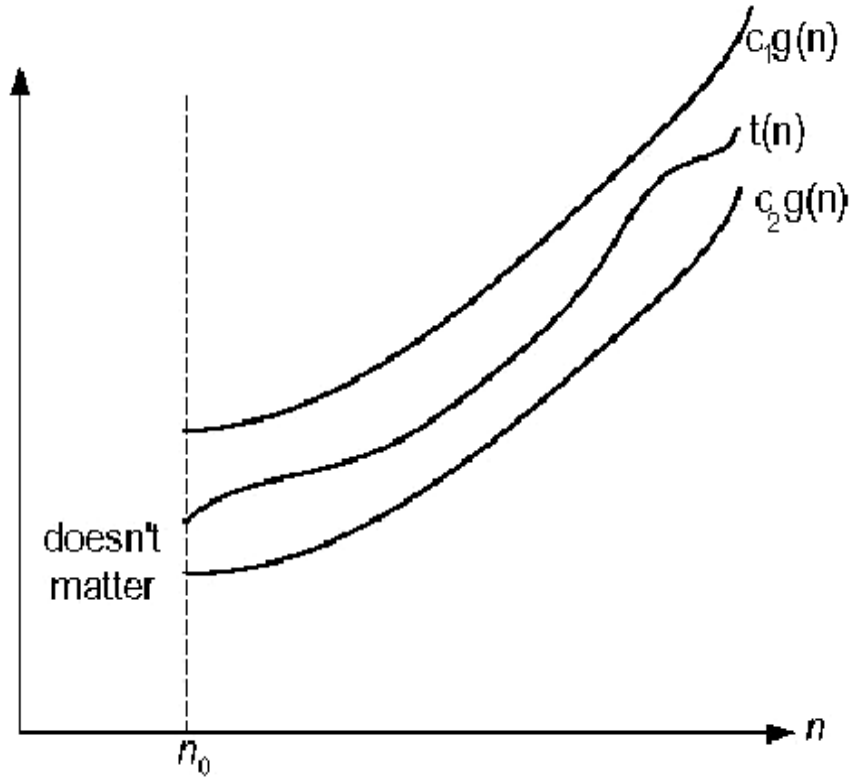
$c_2 n^2 \leq n^2 / 2 - 3n \leq c_1 n^2$ for any $n > n_0$

Dividing by n^2 yields:

$c_2 \leq 1/2 - 3/n \leq c_1$

This is satisfied for

$c_2 = 1/14$, $c_1 = 1/2$, $n_0 = 7$.



Basic Asymptotic Efficiency classes

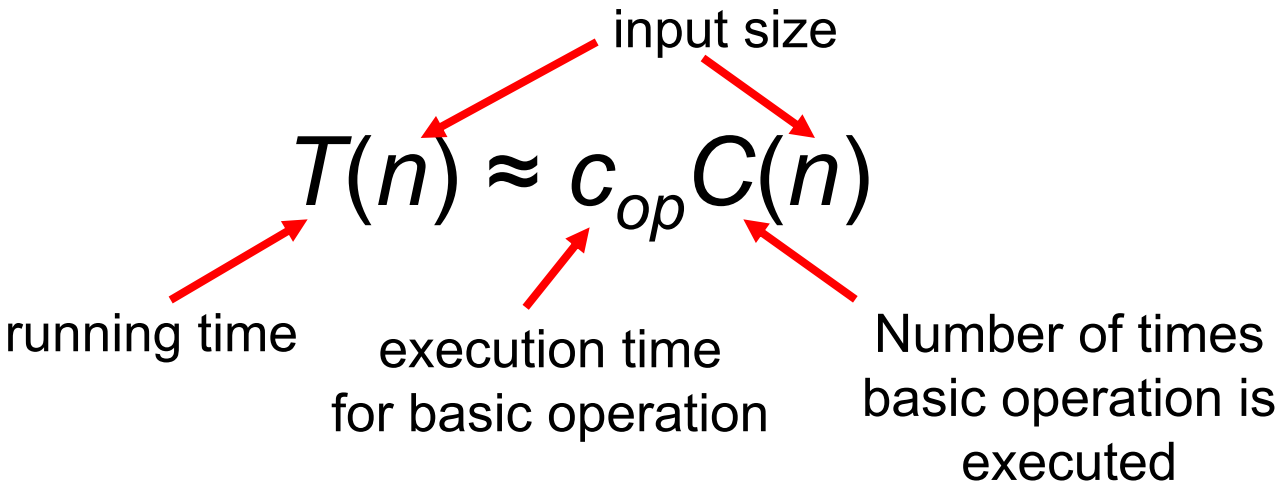
1	constant
log n	logarithmic
n	linear
n log n	n log n
n^2	quadratic
n^3	cubic
2^n	exponential
$n!$	factorial

A Simple Programming Model

- *Prog* = *statement*; *Prog'*: Time(*statement*) + Time (Prog')
- *Prog* = "*var* = *expr*": 1 + Time(*expr*)
- *Prog* = "*expr1* op *expr2*": 1 + Time($expr_1$) + Time($expr_2$) for primitive operations
- *Prog* = "*array*[*expr*]": 1 + Time(*expr*)
- *Prog* = "for *i* = 1 to *n*, do Prog": *n* times Time(Prog')
- *Prog* = *function* call: Time(*function*)
- *Prog* = "while(*expr*), do *Prog'* ": Time(*expr*) + Time(*Prog'*) times

Theoretical analysis of time efficiency

- Time efficiency is analyzed by determining the number of repetitions of the *basic operation* as a function of *input size*
  - *Basic operation:* the operation that contributes most towards the running time of the algorithm.



Time efficiency of nonrecursive algorithms

- Steps in mathematical analysis of nonrecursive algorithms:
  - Decide on parameter *n* indicating *input size*
  - Identify algorithm's *basic operation*
  - Determine *worst*, *average*, and *best* case for input of size *n*
  - Set up summation for *C(n)* reflecting algorithm's loop structure
  - Simplify summation using standard formulas

Example (nonrecursive)

POWERREM1(a, m, n)		Time/stmt	No. times
	▷ Compute $a^n \mod m$		
1	$r \leftarrow a$	1	1
2	**for** $j \leftarrow 2$ **to** n	1+(2)	$n-1$▷
3	**do** $r \leftarrow r \times a$	2	$n-1$
4	**return** $r \mod m$	1	1

$$T(n) = 1 \times 1 + 1 \times 1 + 2 \times (n-1) + 2 \times (n-1) + 1 \times 1 = 1 + 1 + 2n - 2 + 2n - 2 + 1 = 4n - 1 = \Theta(n)$$

- Other issues regarding this algorithm implementation

Time efficiency of recursive algorithms

- Steps in mathematical analysis of recursive algorithms:
  - Decide on parameter *n* indicating *input size*
  - Identify algorithm's *basic operation*
  - Determine *worst*, *average*, and *best* case for input of size *n*
  - Set up a recurrence relation and initial condition(s) for $C(n)$ – the number of times the basic operation will be executed for an input of size *n* (alternatively count recursive calls).
  - Solve the recurrence to obtain a closed form or estimate the order of magnitude of the solution

Example (recursive)

```
RecRussian(a, b)
    ▷ Input: a and b integers
    ▷ Output: product of a and b
1   if a = 0 then return 0
2   if a mod 2 = 1
3       then return (b + RecRussian(⌊a/2⌋, b) × 2)
4       else return (RecRussian(⌊a/2⌋, b) × 2)
```

For even a:

$$ab = \overbrace{b+b+b+b+b+\ldots+b}^{a} = \overbrace{b+b+\ldots}^{a/2} + \overbrace{b+\ldots+b}^{a/2} = 2\left(\overbrace{b+b+b+\ldots+b}^{a/2}\right)$$

For odd a:

$$ab = \overbrace{b+b+b+b+b+\ldots+b}^{a} = b + \overbrace{b+\ldots}^{\lfloor a/2 \rfloor} + \overbrace{\ldots+b}^{\lfloor a/2 \rfloor} = b + 2\left(\overbrace{b+b+\ldots+b}^{\lfloor a/2 \rfloor}\right)$$

Example (recursive) Analysis

Need to find the number of iterations needed to multiply a and b. Let $T(a)$ be the number of iterations on input a. The cases are:

- If $a = 1$, $T(a) = c_1$.
- If a is odd, $T(a) = c_1 + T(\lfloor a/2 \rfloor)$.
- If a is even, $T(a) = c_1 + T(a/2)$.

Theorem 1 *For an integer $a > 1$ we can divide by two (and round down) $1 + \lfloor \log a \rfloor$ times before we get down to zero.*

Example (recursive) Analysis

Proof. For $a > 1$, let $T(a)$ be the number of times we can divide by two (and round down) before we get down to zero. We can express T recursively as $T(1) = 1$, and for $a > 1$, $T(a) = 1 + \lfloor \log a \rfloor$. We have to analyse the base case and the inductive step.

- Base case: $a = 1, \quad T(1) = 1 = 1 + \lfloor 0 \rfloor = 1 + \lfloor \log 1 \rfloor = 1 + \lfloor \log a \rfloor$
- Inductive step: assume that for all $a' < a$ the theorem holds. We have: $T(a) = 1 + T(\lfloor a/2 \rfloor) == 1 + 1 + \lfloor \log(a/2) \rfloor = 2 + \lfloor \log a - \log 2 \rfloor = 2 + \lfloor \log a - 1 \rfloor = 1 + \lfloor \log a \rfloor$.□

Stacks, queues, linked lists

- **Stacks**
  - A stack is a container of StackElements that are inserted and removed according to the last-in-first-out (LIFO) principle.
  - StackElements can be inserted at any time, but only the last (the most-recently inserted) StackElement can be removed.
  - Inserting an item is known as “pushing” onto the stack. “Popping” off the stack is synonymous with removing an item.

The Stack Abstract Data Type

- A stack is an abstract data type (ADT) that supports two main operations:
  - push(x): Inserts StackElement x onto top of stack
    - Input: StackElement; Output: none
  - pop(): Removes the top StackElement of stack and returns it; if stack is empty an error occurs
    - Input: none; Output: StackElement
- The following support operations should also be defined:
  - size(): Returns the number of StackElements in stack
    - Input: none; Output: integer
  - isEmpty(): Return a boolean indicating if stack is empty.
    - Input: none; Output: boolean
  - top(): return the top StackElement of the stack, without removing it; if the stack is empty an error occurs.
    - Input: none; Output: StackElement

The Stack Abstract Data Type

```
PUSH(S, x)
1   top[S] ← top[S] + 1
2   S[top[S]] ← x
```

```
POP(S)
1   if STACK-EMPTY(S)
2       then error "underflow"
3       else top[S] ← top[S] − 1
4            return S[top[S] + 1]
```

Stacks - Relevance

- Stacks appear in computer programs
  - Key to call / return in functions & procedures
  - Stack frame allows recursive calls
  - Call: push stack frame
  - Return: pop stack frame
- Stack frame
  - Function arguments
  - Return address
  - Local variables

Queues

- A queue differs from a stack in that its insertion and removal routines follows the first-in-first-out (FIFO) principle.
- Elements may be inserted at any time, but only the element which has been in the queue the longest may be removed.
- Elements are inserted at the *rear* (enqueued) and removed from the *front* (dequeued)

The Queue Abstract Data Type

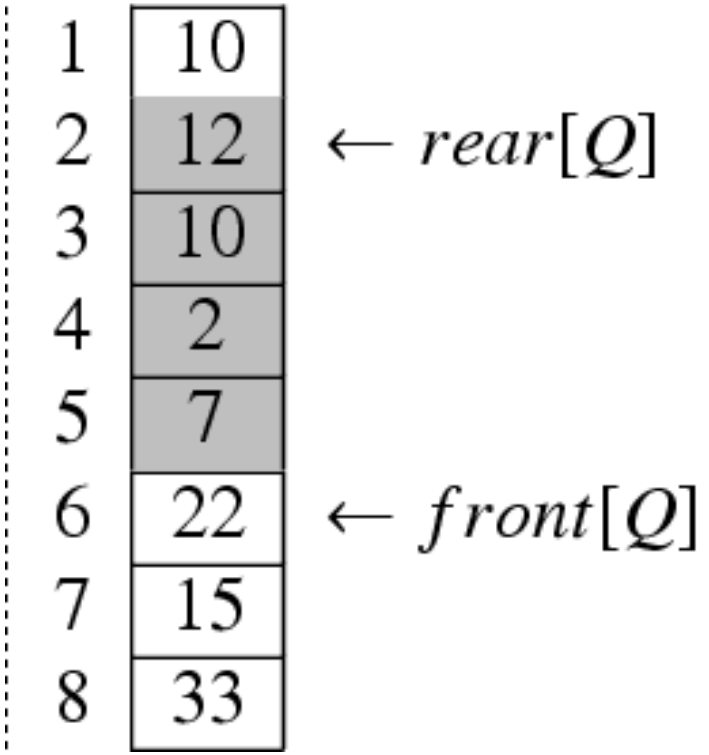
- The queue supports two fundamental operations:
  - enqueue(*o*): Insert QueueElement *o* at the rear of the queue
    - *Input*: QueueElement; *Output*: none
  - dequeue(): Remove the QueueElement from the front of the queue and return it; an error occurs if the queue is empty
    - *Input*: none; *Output*: QueueElement
- These support methods should also be defined:
  - size(): Return the number of QueueElements in the queue
    - *Input*: none; *Output*: integer
  - isEmpty(): Return a boolean value that indicates whether the queue is empty
    - *Input*: none; *Output*: boolean
  - front(): Return, but do not remove, the front QueueElement in the queue; an error occurs if the queue is empty
    - *Input*: none; *Output*: QueueElement

The Queue Abstract Data Type

```
ENQUEUE(Q, x)
1  Q[rear(Q)] ← x
2  if rear[Q] = length[Q]
3       then rear[Q] ← 1
4       else  rear[Q] ← rear[Q] + 1
```

```
DEQUEUE(Q, x)
1  x ← Q[front(Q)]]
2  if front[Q] = length[Q]
3       then front[Q] ← 1
4       else  front[Q] ← front[Q] + 1
5  return x
```

Operations given for a circular queue



Reading

- AHU, chapters 1 & 2
- CLR, chapters 2, 11.1-11.3
- Preiss, chapters: Algorithm Analysis. Asymptotic Notation. Foundational Data Structures. Data Types and Abstraction. Stacks, Queues and Dequeues-4ups. Ordered Lists and Sorted Lists.
- Knuth, vol. 1, 2.1, 2.2
- Notes