Algorithms and Data Structures 2015-2016

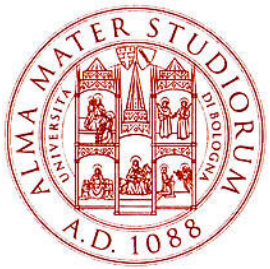
Lesson 2: *stacks and queues*

Luciano Bononi

<luciano.bononi@unibo.it>

http://www.cs.unibo.it/~bononi/

(slide credits: these slides are a revised version of slides created by Dr. Gabriele D'Angelo)



International Bologna Master in Bioinformatics

University of Bologna

xx/03/2016, Bologna

Outline of the lesson

- Introduction to algorithms
- Introduction to data structures and abstract data types
- Abstract data type List
- Basic data structures
  - arrays
  - linked lists

Outline of the lesson

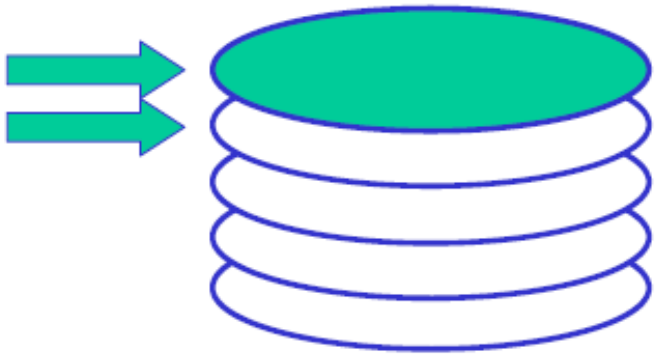
- Abstract Data Types (ADT):
  - **Stacks**
    - definition
    - examples
    - implementation
  - **Queues**
    - definition
    - implementation

Stack overview

- **Intuitive view: a pile of things on top of each other**
  - *Example: a pile of plates*
- An object added to the stack goes on the "top" of the stack
  - *Example: put a plate on the top of the pile*
  - This operation is called "**push"**
- An object removed from the stack is taken from the "top" of the stack
  - *Example: the only plate that can be accessed or removed conveniently is the top one*
  - Removing the top one is called "**pop"**

Stack example

- **Example: a pile of plates**
- A new plate added to the stack goes on the "top" of the stack
- The removal of a plate is from the "top" of the stack
- The only plate that can be accessed or removed conveniently is the top one



Stack Abstract Data Type: specification

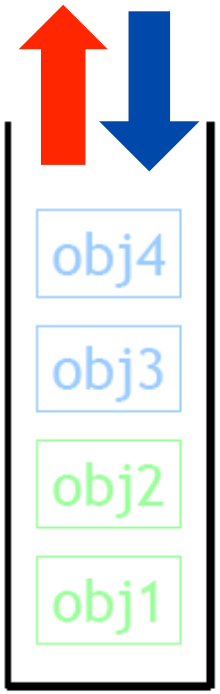
- A stack is a container of objects. Objects are inserted and removed according to the **Last-In-First-Out** (**LIFO**) principle
  - sequence of entries **$<a_1, a_2, \ldots, a_i, \ldots, a_n>$**, but only **$a_n$** is accessible as the "top" of the stack
- **Push(x)** inserts an entry at the "front" of the sequence
  - *Example: given stack L (a list), use add_first(L,x) in List*
- **Top()** returns the last entry
  - *Example: return(L.Head) node in List*
- **Pop()** deletes the last entry
  - Example: delete(L.Head) in a List (and update L.Head)

Stack Abstract Data Type: description

- Objects can be inserted at any time, but **only the last** (the most-recently inserted) **object can be removed**
- Objects are removed in the **reverse order** from that in which they were inserted
- Usually it is not possible to access objects that are in the "**middle**" of the stack
- It is possible to access the "top" element without removing it

Stack Abstract Data Type: operations

- **Creates** a new Stack — Make(S)
- True if Stack is **empty**, false otherwise — Empty(S)
- **Inserts** a new element onto the top of the Stack — Push(S, x)
- **Returns** the top entry of the Stack, the stack is unchanged — Top(S)
- **Returns and delete** the top entry of the Stack — Pop(S)

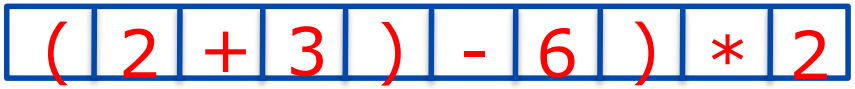


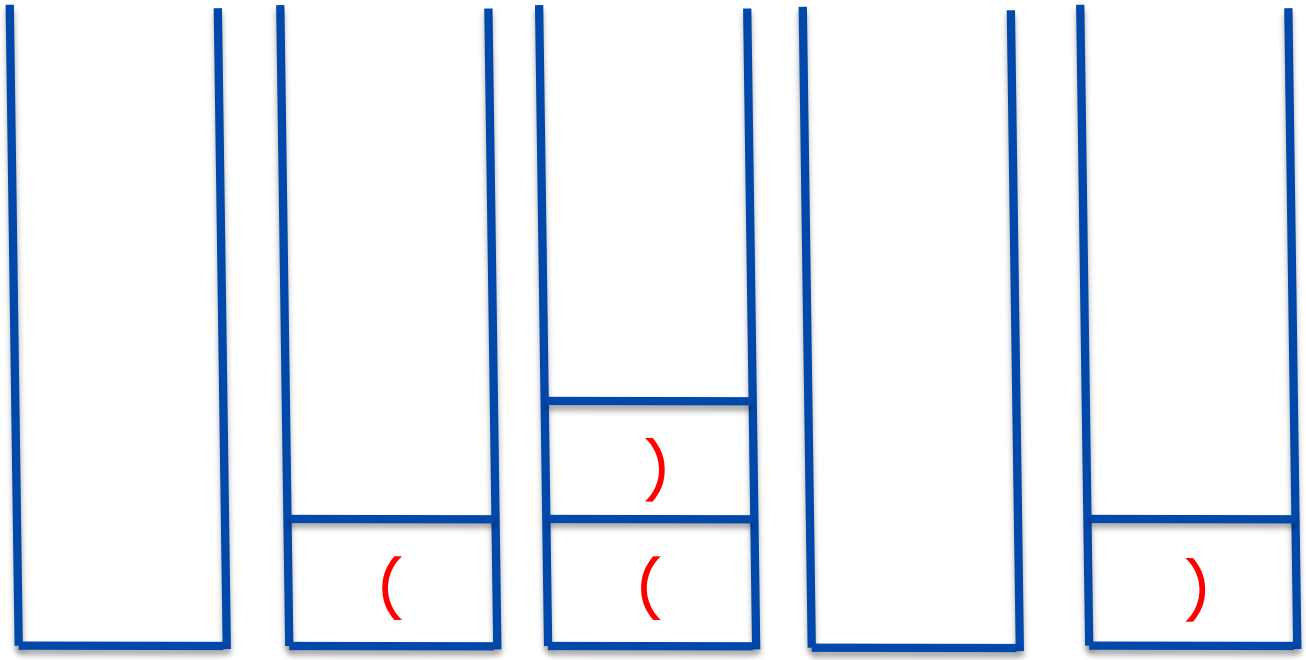
Applications of Stacks

- Stacks are used in many aspects of computing
  - Arithmetic expression evaluation
  - Syntax parsing
    - *Example: properly balanced parenthesis*
  - Activation record of functions
  - Convert recursive algorithms to non-recursive version

Applications of Stacks: parenthesis balance example

- Expression to check: **(2+3)-6)*2**





Stack empty!

The parenthesis are unbalanced!

Applications of Stacks: arithmetic expression evaluation

- Given a simple arithmetic expression such as:

((1 + 2) * 4) + 3

- Is it possible to use a stack for its evaluation?

- If it is possible then we should be able to write an algorithm that uses a stack and that defines how to evaluate the given expression

Reverse polish notation

- The Reverse Polish Notation (RPN, also called Postfix notation) is a postfix notation wherein every operator follows all of its operands

- For example:
  - conventional infix notation: **“3 + 4”**
  - postfix notation: **“3 4 +”**

Reverse polish notation

- Another example:
  - infix: **“3 - 4 + 5”**
  - RPN: **“3 4 – 5 +”**
- One of the advantages of RPN is that parentheses are not necessary
  - **“3 – (4 * 5)”**is **“3 4 5 * -”**

Reverse polish notation

- A more complex example:

  ((1 + 2) * 4) +3 is **1 2 + 4 * 3 +**

Simple algorithm:

- The expression is evaluated from the left to the right using a stack
  - push when encountering an operand and
  - pop two operands and evaluate the value when encountering an operation
  - push the result

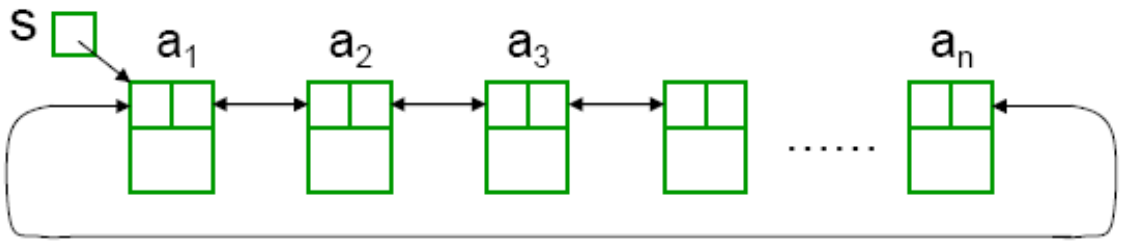
Applications of Stacks: arithmetic expression evaluation

((1 + 2) * 4) +3 -> 1 2 + 4 * 3 +

Step	Input	Operation	Stack
1	1	*Push operand*	**1**
2	2	*Push operand*	**1, 2**
3	+	*Add*	**3**
4	4	*Push operand*	**3, 4**
5	*	*Multiply*	**12**
6	3	*Push operand*	**12, 3**
7	+	*Add*	**15**

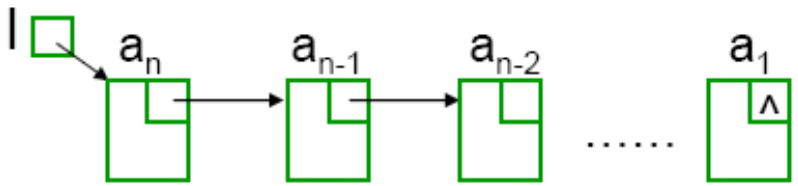
Implementation of the Stack ADT

- It is possible to use different data structures to implement the stack abstract data type
- **Option #1: doubly linked list**
  - Each entry contains references to its predecessor and successor in the sequence, even for the first and last entries
  - The whole list is represented by a header containing only a reference to the first entry
  - Stack operations push and pop need to handle references in both directions. **Is it possible a simpler implementation?**



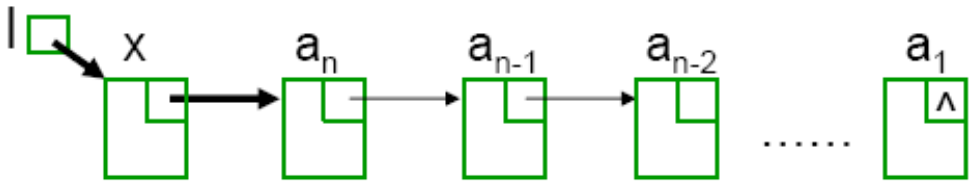
Implementation of the Stack ADT

- **Option #2: singly linked list**
  - Each entry contains only one reference to its successor in the sequence. The first and last entries are not linked
  - The last entry contains a NULL (^) reference
  - The Stack can be seen as a special list, the Stack operations form a subset of the List operations



Implementation of the Stack ADT

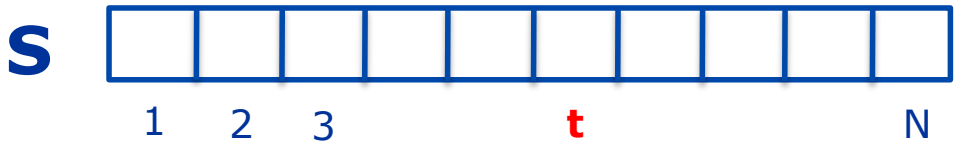
- **Option #2 (continue)**
  - To support Stack operations conveniently, here we put the back of the sequence at the front of the singly linked list
  - Push() implementation:



  - Pop() implementation: *"move the header one step on the right"*
  - **Both Push() and Pop() are O(1)**

Implementation of the Stack ADT

- **Option #3: array**
  - Create a stack using an array by specifying a maximum size N for our stack
  - The stack consists of a N-element array S and an integer variable t, the index of the top element in array S



  - The array implementation is simple and efficient
    - Operations performed in O(1), except for resizing
  - There is an upper bound, N, on the size of the stack
    - The arbitrary value N may be too small for a given application, or a waste of memory if it is too large

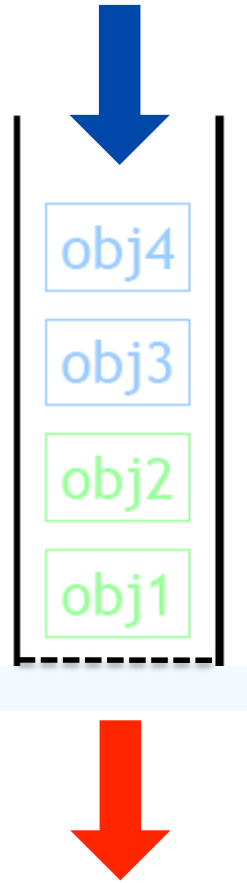
Queue ADT: definition

- A **Queue** differs from a Stack in that its insertion and removal follow the **First-In-First-Out** (**FIFO**) principle.
  - $<a_1, a_2, \ldots, a_i, \ldots, a_n>$
  - the same to a real-life queue
- Objects can be inserted at any time, but only the object which has been in the queue the longest may be removed
- Objects are inserted at the *rear* (**enqueued**) and removed from the *front* (**dequeued**)
- Objects are removed from a queue in the same order as they were inserted

Queue ADT: animated example

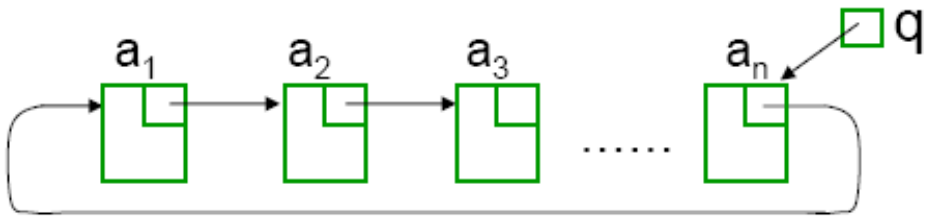
Queue ADT: operations

- **Creates** a new Queue — Make(Q)
- True if Queue is **empty**, false otherwise — Empty(Q)
- **Inserts** the new element x at the end of the Queue — Enqueue(Q, x)
- **Returns** the front entry of the Queue, which is not changed — Front(Q)
- **Removes and returns** the front entry of the queue — Dequeue(Q)



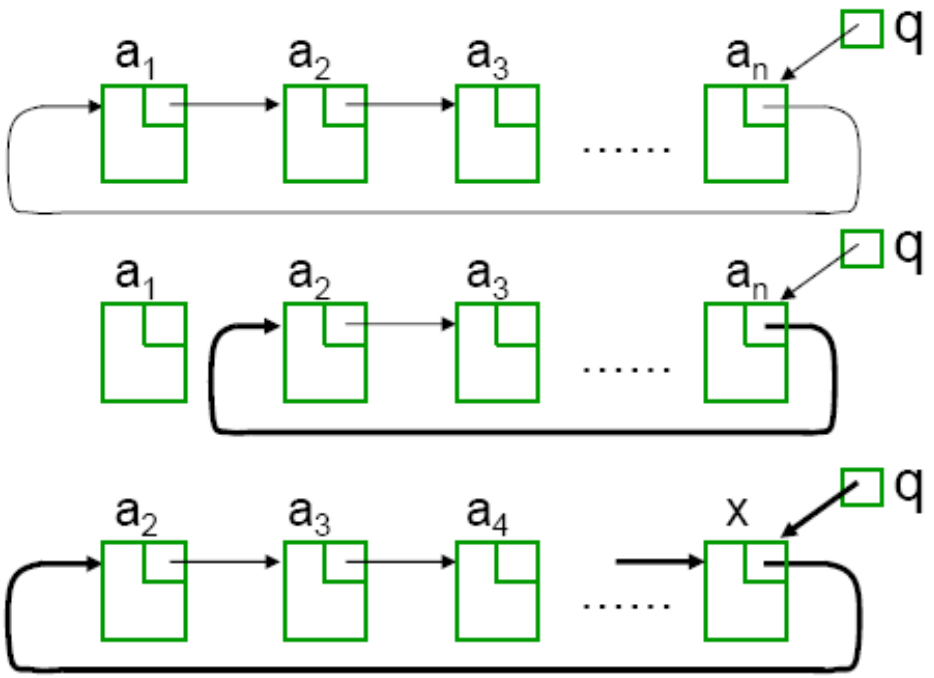
Queue ADT: implementation

- A Queue can be seen as a special case of List
- Queue operations are a subset of List operations
- **Option #1: singly linked circular list**
  - The last entry has a reference to the first
  - The header contains a reference to the last entry



Queue ADT: implementation

- **Dequeue(Q, x)**
- **Enqueue(Q, x)**
- **Enqueue** and **Dequeue** an item: time needed is independent of the number of items in the queue -> O(1)



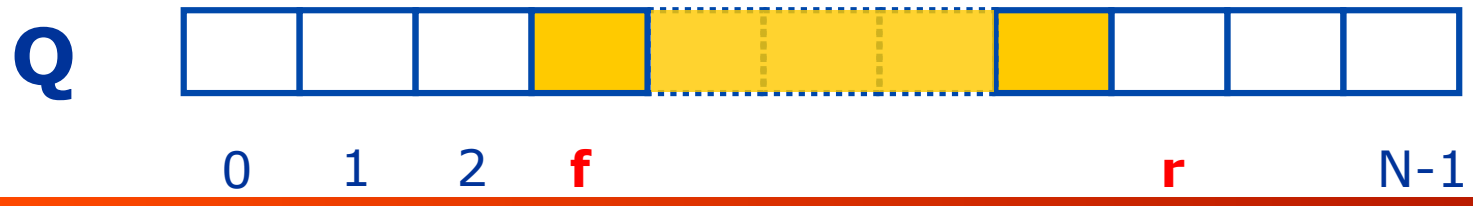
Is it possible to further improve it?

Queue ADT: implementation

- **Option #2: a "circularly managed" or "wrapped" array**
  - A maximum size N is specified
  - Why in a circular fashion?
  - What about a "standard array"?

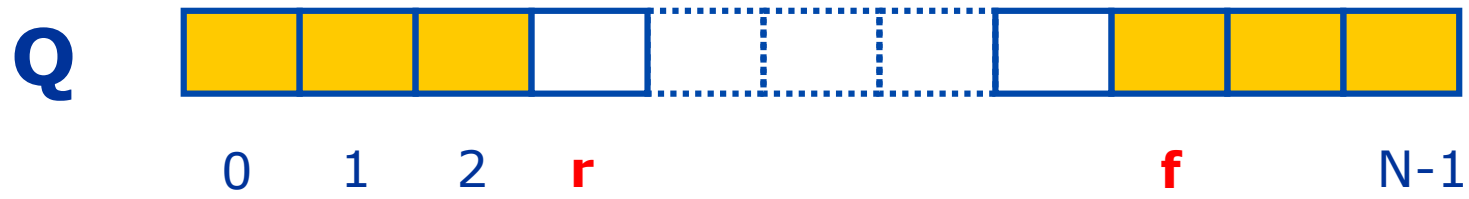
Queue ADT: implementation

- The queue consists of an N-element **array** Q and **two integer variables**:
  - **f**: index of the **front element** (head–for dequeue)
  - **r**: index of the **element after the rear one** (tail–for enqueue)
  - Initially, **f** = **r** = 0
  - **Enqueue**: increase **r** by 1
  - **Dequeue**: increase **f** by 1
  - To allow circular fashion: we use a **module operation**, mod(), when increasing **f** and **r**



Queue ADT: implementation

- After a number of Enqueue and Dequeue operations, we may get a "wrapped around" configuration



- What does **f = r** mean?
  - an **empty** array
    - initially f=r=0, or all enqueued objects are dequeue and f=r>0
  - a **full** array
    - r is increased continuously, from 0, 1, 2, …, to f finally

Queue ADT: implementation

- How to differentiate these two cases?
  - Use **one array index** and a **size variable** to maintain the Queue length, in this case the queue is full when **size is equal to N**
  - How is it possible to manage the Queue with only one array index? (index the enqueue point, managed in circular way and assume elements initialized as null value. Dequeue requires searching the last element in the array: O(n)).

- **Dequeue()** and **Enqueue()** take both **O(1)** time, except Enqueue() into a **full queue**. Resizing again: **O(n)**

References

- Part of this material is inspired / taken by the following freely available resources:
  - http://www.cs.rutgers.edu/~vchinni/dsa/
  - http://www.cs.aau.dk/~luhua/courses/ad07/
  - Wikipedia “stack data structure”

Algorithms and Data Structures 2015-2016

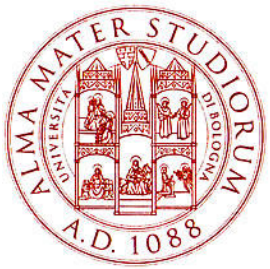
Lesson 2: *stacks and queues*

Luciano Bononi

<luciano.bononi@unibo.it>

http://www.cs.unibo.it/~bononi/

(slide credits: these slides are a revised version of slides created by Dr. Gabriele D'Angelo)



International Bologna Master in Bioinformatics

University of Bologna

xx/03/2016, Bologna