DATA STRUCTURES AND ALGORITHMS
LECTURE 7

Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2017

In Lecture 6...

- Heap
- ADT List
- ADT Stack

Today

1. ADT Stack
2. ADT Queue

ADT Stack

- A stack is a container in which the access is restricted to only one end of the container (called the *top*).
- Because of this restricted access, we say that the stack uses a LIFO (Last In First Out) principle.

ADT Stack

- Main stack operations (complete interface with specifications is in Lecture 6):
  - init - creates a new stack
  - push - pushes (adds) a new element to the top of the stack
  - pop - pops (removes) and returns the element from the top of the stack
  - top - returns the element from the top of the stack
  - isEmpty - checks if the stack is empty

Representation for Stack

- Data structures that can be used to implement a stack:
  - Arrays
    - Static Array
    - Dynamic Array
  - Linked Lists
    - Singly Linked List
    - Doubly Linked List

? Where should we place the top of the stack for optimal performance?

GetMinimum in constant time

? How can we design a *special stack* that has a *getMinimum* operation with $\Theta(1)$ time complexity (and the other operations have $\Theta(1)$ time complexity as well)?

GetMinimum in constant time

? How can we design a *special stack* that has a *getMinimum* operation with $\Theta(1)$ time complexity (and the other operations have $\Theta(1)$ time complexity as well)?

- We can keep an auxiliary stack, containing as many elements as the original stack, but containing the minimum value up to each element. Let's call this auxiliary stack a *min stack* and the original stack the *element stack*.

GetMinimum in constant time

- Let's implement the *push*, *pop* and *getMinimum* operations for this *SpecialStack*, represented in the following way:

SpecialStack:
  elementStack: Stack
  minStack: Stack

- We will use an existing implementation for the stack and work only with the operations from the interface.

SpecialStack - Push

- What should the push operation do?

```
subalgorithm push(ss, e) is:
  //if stacks can be full, check if they are full
  if isEmpty(ss.elementStack) then//the stacks are empty, just push the elem
    push(ss.elementStack, e)
    push(ss.minStack, e)
  else
    push(ss.elementStack, e)
    currentMin ← top(ss.minStack)
    if currentMin < e then //find the minim to push to minStack
      push(ss.minStack, currentMin)
    else
      push(ss.minStack, e)
    end-if
  end-if
end-subalgorithm //Complexity: Θ(1)
```

SpecialStack - Pop

- What should the pop operation do?

function pop(ss) **is:**
    if isEmpty(ss.elementStack) **then**
        @throw underflow (empty stack) exception
    end-if
    currentElem ← pop(ss.elementStack)
    pop(ss.minStack) *//we don't need the value, just to pop it*
    pop ← currentElem
end-function

- Complexity: $\Theta(1)$

GetMinimum for SpecialStack

- What should the getMinimum operation do?

function getMinimum(ss) **is:**
  if isEmpty(ss.elementStack) **then**
    @throw underflow (empty stack) exception
  end-if
  getMinimum $\leftarrow$ top(ss.minStack)
end-function

- Complexity: $\Theta(1)$

SpecialStack - Notes / Think about it

- We designed the special stack in such a way that all the operations have a $\Theta(1)$ time complexity.
- The disadvantage is that we occupy twice as much space as with the regular stack.

? Think about how can we reduce the space occupied by the *min stack* (especially if the minimum element of the stack rarely changes). *Hint: If the minimum does not change, we don't have to push a new element to the min stack.* How can we implement the *push* and *pop* operations in this case? What happens if the minimum element appears more than once in the *element stack*?

Delimiter matching

- Given a sequence of round brackets (parentheses), brackets and curly brackets, verify if the brackets are opened and closed correctly.
- For example:
    - The sequence ()([][][(())]) - is correct
    - The sequence [()()()()] - is correct
    - The sequence [()]]) - is not correct (one extra closed round bracket at the end)
    - The sequence [(]) - is not correct (brackets closed in wrong order)
    - The sequence {[[]] () - is not correct (curly bracket is not closed)

Bracket matching - Solution Idea

- Stacks are suitable for this problem, because the bracket that was opened last should be the first to be closed. This matches the LIFO property of the stack.
- The main idea of the solution:
  - Start parsing the sequence, element-by-element
  - If we encounter an open bracket, we push it to a stack
  - If we encounter a closed bracket, we pop the last open bracket from the stack and check if they match
  - If they don't match, the sequence is not correct
  - If they match, we continue
  - If the stack is empty when we finished parsing the sequence, it was correct

Bracket matching - Implementation

function bracketMatching(seq) **is:**
    init(st) *//create a stack*
    for elem **in** seq **execute**
        if @ elem is open bracket **then**
            push(st, elem)
        else *//elem is a closed bracket*
            if isEmpty(st) **then**
                bracketMatching ← False *//no open bracket at all*
            else
                lastOpenedBracket ←pop(st)
                if not @lastOpenedBracket matches elem **then**
                    bracketMatching ← False
                end-if
            end-if
        end-if
    end-for *//continued on next slide...*

Bracket matching - Implementation

if isEmpty(st) **then**
    bracketMatching $\leftarrow$ True
else *//we have extra open bracket(s)*
    bracketMatching $\leftarrow$ False
end-if
end-function

- Complexity: $\Theta(n)$ - where n is the length of the sequence

Bracket matching - Extension

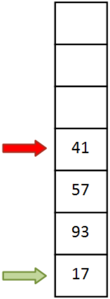
- How can we extend the previous implementation so that in case of an error we will also signal the position where the problem occurs?
- Remember, we have 3 types of errors:
  - Open brackets that are never closed
  - Closed brackets that were not opened
  - Mismatch
- Keep count of the current position in the sequence, and push to the stack $< \textit{delimiter}, \textit{position} >$ pairs.

ADT Queue

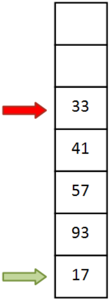
- The ADT Queue represents a container in which access to the elements is restricted to the two ends of the container, called *front* and *rear*.
    - When a new element is added (pushed), it has to be added to the *rear* of the queue.
    - When an element is removed (popped), it will be the one at the *front* of the queue.
- Because of this restricted access, the queue is said to have a **FIFO** policy: First In First Out.

ADT Queue - Example

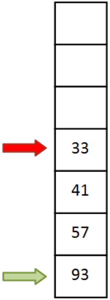
- Assume that we have the following queue (green arrow is the front, red arrow is the rear)



- Push number 33:

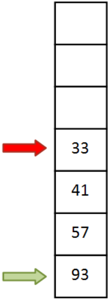


- Pop an element:

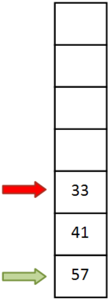


ADT Queue - Example

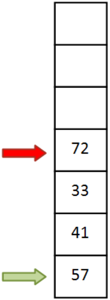
- This is our queue:



- Pop an element:



- Push number 72:



ADT Queue - Interface I

- The domain of the ADT Queue:
  $\mathcal{Q} = \{q | q\ \textit{is a queue with elements of type TElem}\}$

- The interface of the ADT Queue contains the following operations:

ADT Queue - Interface II

- init(q)
  - **Description:** creates a new empty queue
  - **Pre:** True
  - **Post:** $q \in \mathcal{Q}$, q is an empty queue

ADT Queue - Interface III

- destroy(q)
  - **Description:** destroys a queue
  - **Pre:** $q \in \mathcal{Q}$
  - **Post:** q was destroyed

ADT Queue - Interface IV

- push(q, e)
  - **Description:** pushes (adds) a new element to the rear of the queue
  - **Pre:** $q \in \mathcal{Q}$, e is a *TElem*
  - **Post:** $q' \in \mathcal{Q}$, $q' = q \oplus e$, e is the element at the rear of the queue
  - **Throws:** an *overflow* error if the queue is full

ADT Queue - Interface V

- pop(q)
  - **Description:** pops (removes) the element from the front of the queue
  - **Pre:** $q \in \mathcal{Q}$
  - **Post:** $pop \leftarrow e$, e is a *TElem*, e is the element at the front of q, $q' \in \mathcal{Q}$, $q' = q \ominus e$
  - **Throws:** an *underflow* error if the queue is empty

ADT Queue - Interface VI

- top(q)
  - **Description:** returns the element from the front of the queue (but it does not change the queue)
  - **Pre:** $q \in \mathcal{Q}$
  - **Post:** $top \leftarrow e$, e is a $TElem$, e is the element from the front of q
  - **Throws:** an *underflow* error if the queue is empty

ADT Queue - Interface VII

- isEmpty(s)
  - **Description:** checks if the queue is empty (has no elements)
  - **Pre:** $q \in \mathcal{Q}$
  - **Post:**

$$isEmpty \leftarrow \begin{cases} true, & if\ q\ has\ no\ elements \\ false, & otherwise \end{cases}$$

ADT Queue - Interface VIII

- isFull(q)
  - **Description:** checks if the queue is full - not every implementation has this operation
  - **Pre:** $q \in \mathcal{Q}$
  - **Post:**

$$isFull \leftarrow \begin{cases} true, & if\ q\ is\ full \\ false, & otherwise \end{cases}$$

ADT Queue - Interface IX

- **Note:** queues cannot be iterated, so they don't have an *iterator* operation!

Queue - Representation

- What data structures can be used to implement a Queue?
  - Static Array
  - Dynamic Array
  - Singly Linked List
  - Doubly Linked List
- For each possible representation we will discuss where we should place the *front* and the *rear* of the queue and the complexity of the operations.

Queue - Array-based representation

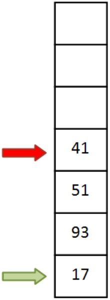
- If we want to implement a Queue using an array (static or dynamic), where should we place the *front* and the *rear* of the queue?
- In theory, we have two options:
  - Put *front* at the beginning of the array and *rear* at the end
  - Put *front* at the end of the array and *rear* at the beginning
- In either case we will have one operation (push or pop) that will have $\Theta(n)$ complexity.

Queue - Array-based representation

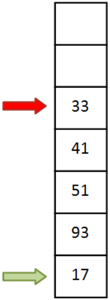
- We can improve the complexity of the operations, if we do not insist on having either *front* or *rear* at the beginning of the array (at position 1).

Queue - Array-based representation

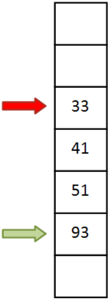
- This is our queue (green arrow is the front, red arrow is the rear)



- Push number 33:

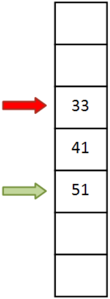


- Pop an element (and do not move the other elements):

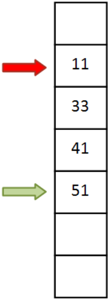


Queue - Array-based representation

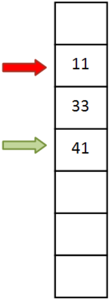
- Pop another element:



- Push number 11:

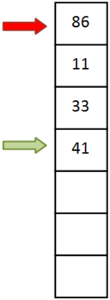


- Pop an element:

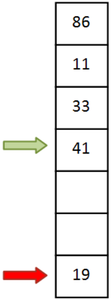


Queue - Array-based representation

- Push number 86:



- Push number 19:



Queue - representation on a circular array

- How can we represent a Queue on a circular array?

Queue:
  capacity: Integer
  front: Integer
  rear: Integer
  elems: TElem[]

- Some books suggest that the *length* of the queue should also be kept as a part of the structure.

Queue - representation on a circular array - init

- We will use the value -1 for *front* and *end*, to denote an empty queue.

subalgorithm init(q) **is:**
  q.capacity ← INIT_CAPACITY //*some constant*
  q.front ← -1
  q.rear ← -1
  @allocate memory for the *elems* array
end-subalgorithm

- Complexity: $\Theta(1)$

Queue - representation on a circular array - isEmpty

- How do we check whether the queue is empty?

function isEmpty(q) **is:**
  if q.front = -1 **then**
    isEmpty ← True
  else
    isEmpty ← False
  end-if
end-function

- Complexity: $\Theta(1)$

Queue - representation on a circular array - top

- What should the *top* operation do?

```
function top(q) is:
  if q.front != -1 then
    top ← q.elems[q.front]
  else
    @error - queue is empty
  end-if
end-function
```

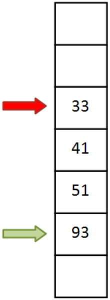
- Complexity: $\Theta(1)$

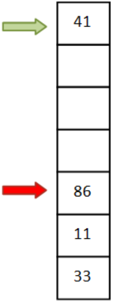
Queue - representation on a circular array - pop

- What should the *pop* operation do?

Queue - representation on a circular array - pop

- There are two situations for our queue:





Queue - representation on a circular array - pop

function pop (q) **is:**
  if q.front != -1 **then**
    deletedElem ← q.elems[q.front]
    if q.front = q.rear **then** *//we have one single element*
      q.front ← -1
      q.rear ← -1
    else if q.front = q.cap **then**
      q.front ← 1
    else
      q.front ← q.front + 1
    end-if
    pop ← deletedElem
  end-if
  @error - queue is empty
end-function

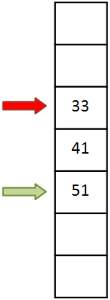
- Complexity: $\Theta(1)$

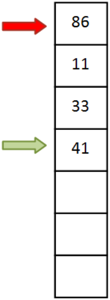
Queue - representation on a circular array - push

- What should the *push* operation do?

Queue - representation on a circular array - push

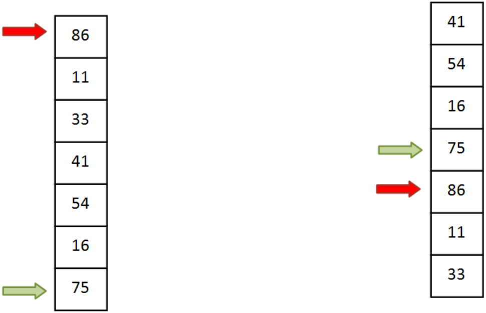
- There are two situations for our queue:





Queue - representation on a circular array - push

- When pushing a new element we have to check whether the queue is full



- For both example, the elements were added in the order: 75, 16, 54, 41, 33, 11, 86

Queue - representation on a circular array - push

- If we have a dynamic array-based representation and the array is full, we have to allocate a larger array and copy the existing elements (as we always do with dynamic arrays)
- When the existing elements are copied, we have to *straighten out* the array.

Queue - representation on a circular array - push

subalgorithm push(q, e) **is:**
    if q.front = -1 **then**
        q.elems[1] $\leftarrow$ e
        q.front $\leftarrow$ 1
        q.rear $\leftarrow$ 1
        @return
    else if(q.front=1 **and** q.rear=a.cap) **OR** q.rear=q.front-1 **then**
        @resize
    end-if
    if q.rear $\neq$ q.cap **then**
        q.elems[q.rear+1] $\leftarrow$ e
        q.rear $\leftarrow$ q.rear + 1
    else
        q.elems[1] $\leftarrow$ e
        q.rear $\leftarrow$ 1
    end-if
end-subalgorithm

- Complexity: $\Theta(1)$

Queue - representation on a SLL

- If we want to implement a Queue using a singly linked list, where should we place the *front* and the *rear* of the queue?
- In theory, we have two options:
  - Put *front* at the beginning of the list and *rear* at the end
  - Put *front* at the end of the list and *rear* at the beginning
- In either case we will have one operation (push or pop) that will have $\Theta(n)$ complexity.

Queue - representation on a SLL

- We can improve the complexity of the operations if, even though the list is singly linked, we keep both the head and the tail of the list.
- What should the tail of the list be: the *front* or the *rear* of the queue?

Evaluating an arithmetic expression

- We want to write an algorithm that can compute the result of an arithmetic expression:
- For example:
  - 2+3*4 = 14
  - ((2+4)*7)+3*(9-5) = 54
  - ((((3+1)*3)/((9-5)+2))-((3*(7-4)) + 6)) = -13
- An arithmetic expression is composed of *operators* (+, -, * or /), parentheses and *operands* (the numbers we are working with). For simplicity we are going to use single digits as operands and we suppose that the expression is correct.

Infix and postfix notations

- The arithmetic expressions presented on the previous slide are in the so-called *infix* notation. This means that the *operators* are between the two operands that they refer to. Humans usually use this notation, but for a computer algorithm it is complicated to compute the result of an expression in an infix notation.

- Computers can work a lot easier with the *postfix* notation, where the operator comes after the operands.

Infix and postfix notations

- Examples of expressions in infix notation and the corresponding postfix notations:

Infix notation	Postfix notation
1+2	12+
1+2-3	12+3-
4*3+6	43*6+
4*(3+6)	436+*
(5+6)*(4-1)	56+41-*
1+2*(3-4/(5+6))	123456+/-*+

- The order of the operands is the same for both the infix and the postfix notations, only the order of the operators changes
- The operators have to be ordered taking into consideration operator precedence and the parentheses

Infix and postfix notations

- So, evaluating an arithmetic expression is divided into two subproblems:
  - Transform the infix notation into a postfix notation
  - Evaluate the postfix notation
- Both subproblems are solved using stacks and queues.

Infix to postfix transformation - The main idea

- Use an auxiliary stack for the operators and parentheses and a queue for the result.
- Start parsing the expression.
- If an operand is found, push it to the queue
- If an open parenthesis is found, it is pushed to the stack.
- If a closed parenthesis is found, pop elements from the stack and push them to the queue until an open parenthesis is found (but do not push parentheses to the queue).

Infix to postfix transformation - The main idea

- If an operator (opCurrent) is found:
    - If the stack is empty, push the operator to the stack
    - While the top of the stack contains an operator with a higher or equal precedence than the current operator, pop and push to the queue the operator from the stack. Push opCurrent to the stack when the stack becomes empty, its top is a parenthesis or an operator with lower precedence.
    - If the top of the stack is open parenthesis or operator with lower precedence, push opCurrent to the stack.
- When the expression is completely parsed, pop everything from the stack and push to the queue.

Infix to postfix transformation - Example

- Let's follow the transformation of 1+2*(3-4/(5+6))+7

Input	Operation	Stack	Queue
1	Push to Queue		1
+	Push to stack	+	1
2	Push to Queue	+	12
*	Check (no Pop) and Push	+*	12
(	Push to stack	+*(	12
3	Push to Queue	+*(	123
-	Check (no Pop) and Push	+*(-	123
4	Push to Queue	+*(-	1234
/	Check (no Pop) and Push	+*(-/	1234
(	Push to stack	+*(-/(	1234
5	Push to Queue	+*(-/(	12345
+	Check (no Pop) and Push	+*(-/(+	12345
6	Push to Queue	+*(-/(+	123456
)	Pop and push to Queue till (	+*(-/	123456+
)	Pop and push to Queue till (	+*	123456+/-
+	Check, Pop twice and Push	+	123456+/-*+
7	Push to Queue	+	123456+/-*+7
over	Pop everything and push to Queue		123456+/-*+7+

Infix to postfix transformation - Implementation

```
function infixToPostfix(expr) is:
    init(st)
    init(q)
    for elem in expr execute
        if @elem is an operand then
            push(q, elem)
        else if @ elem is open parenthesis then
            push(st, elem)
        else if @elem is a closed parenthesis then
            while @ top(st) is not an open parenthesis execute
                op ← pop(st)
                push(q, op)
            end-while
            pop(st) //get rid of open parenthesis
        else //we have operand
//continued on the next slide
```

Infix to postfix transformation - Implementation

```
        while not isEmpty(st) and @ top(st) not open parenthesis and @
top(st) has >= precedence than elem execute
            op ← pop(st)
            push(q, op)
        end-while
        push(st, elem)
    end-if
  end-for
  while not isEmpty(st) execute
    op ← pop(st)
    push(q, op)
  end-while
  infixtoPostfix ← q
end-function
```

- Complexity: $\Theta(n)$ - where n is the length of the sequence

Evaluation of expression in postfix notation

- Once we have the postfix notation we can compute the value of the expression using a stack
- The main idea of the algorithm:
  - Use an auxiliary stack
  - Start parsing the expression
  - If an operand if found, it is pushed to the stack
  - If an operator is found, two values are popped from the stack, the operation is performed and the result is pushed to the stack
  - When the expression is parsed, the stack contains the result

Evaluation of postfix notation - Example

- Let's follow the evaluation of 123456+/-*+7+

Pop from the queue	Operation	Stack
1	Push	1
2	Push	1 2
3	Push	1 2 3
4	Push	1 2 3 4
5	Push	1 2 3 4 5
6	Push	1 2 3 4 5 6
+	Pop, add, Push	1 2 3 4 11
/	Pop, divide, Push	1 2 3 0
-	Pop, subtract, Push	1 2 3
*	Pop, multiply, Push	1 6
+	Pop, add, Push	7
7	Push	7 7
+	Pop, add, Push	14

Evaluation of postfix notation - Implementation

```
function evaluatePostfix(q) is:
    init(st)
    while not isEmpty(q) execute
        elem ← pop(q)
        if @ elem is an operand then
            push(st, elem)
        else
            op1 ← pop(st)
            op2 ← pop(st)
            @ compute the result of op2 elem op1 in variable result
            push(st, result)
        end-if
    end-while
    result ← pop(st)
    evaluatePostfix ← result
end-function
```

- Complexity: $\Theta(n)$ - where n is the length of the expression

Evaluation of an arithmetic expression

- Combining the two functions we can compute the result of an arithmetic expression.
- How can we evaluate directly the expression in infix notation with one single function? *Hint: use two stacks.*
- How can we add exponentiation as a fifth operation?