DATA STRUCTURES AND ALGORITHMS
LECTURE 8

Lect. PhD. Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2017 - 2018

In Lecture 7...

- ADT Matrix
- ADT List
- ADT Stack

Today

1. ADT Queue
2. ADT Deque
3. ADT Priority Queue
4. Written test

Note

- We will not have seminar on the 30th of April
- These classes will be held on:
  - 915 - Friday, 4th of May, 12-14, room A321
  - 917 - Friday, 4th of May, 8-10, room A312*

Delimiter matching

- Given a sequence of round brackets (parentheses), (square) 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 ← True
    else //we have extra open bracket(s)
        bracketMatching ← 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

Bracket matching - Extension

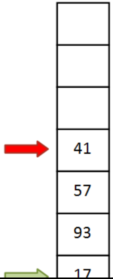
- How can we extend the previous implementation so that in case of an error we will also signal the position where the problem occurs?
- Remember, we have 3 types of errors:
  - Open brackets that are never closed
  - Closed brackets that were not opened
  - Mismatch
- Keep count of the current position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

ADT Queue

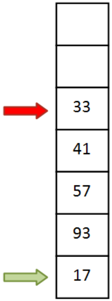
- The ADT Queue represents a container in which access to the elements is restricted to the two ends of the container, called *front* and *rear*.
    - When a new element is added (pushed), it has to be added to the *rear* of the queue.
    - When an element is removed (popped), it will be the one at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

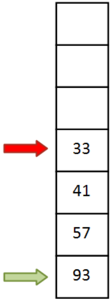
- Assume that we have the following queue (green arrow is the front, red arrow is the rear)



- Push number 33:

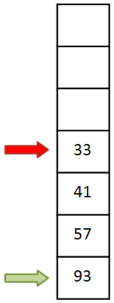


- Pop an element:

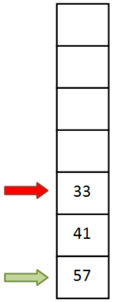


ADT Queue - Example

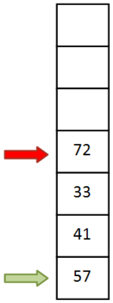
- This is our queue:



- 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?

Queue - Array-based representation

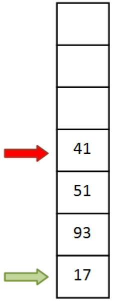
- If we want to implement a Queue using an array (static or dynamic), where should we place the *front* and the *rear* of the queue?
- In theory, we have two options:
  - Put *front* at the beginning of the array and *rear* at the end
  - Put *front* at the end of the array and *rear* at the beginning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

Queue - Array-based representation

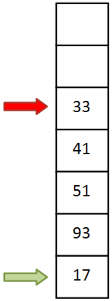
- We can improve the complexity of the operations, if we do not insist on having either *front* or *rear* at the beginning of the array (at position 1).

Queue - Array-based representation

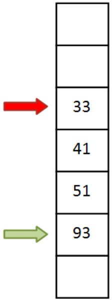
- This is our queue (green arrow is the front, red arrow is the rear)



- Push number 33:

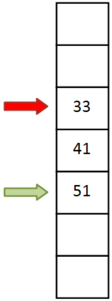


- Pop an element (and do not move the other elements):

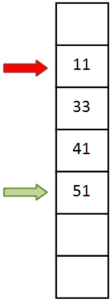


Queue - Array-based representation

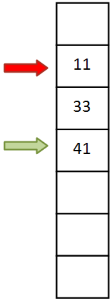
- Pop another element:



- Push number 11:

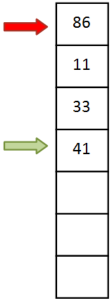


- Pop an element:

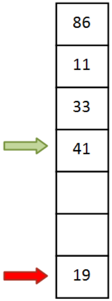


Queue - Array-based representation

- 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[]
```

- Optionally, the *length* of the queue could also be kept as a part of the structure.
- Front and rear (in this implementation) are positions actually occupied by the elements.

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
```

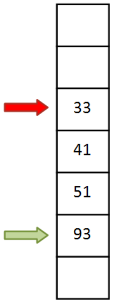
- Complexity: $\Theta(1)$

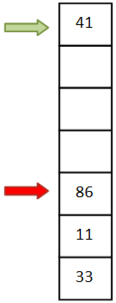
Queue - representation on a circular array - pop

- 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
```

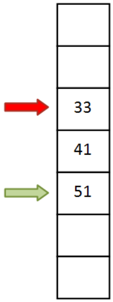
- Complexity: $\Theta(1)$

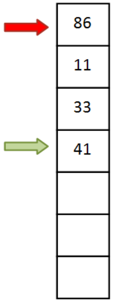
Queue - representation on a circular array - push

- What should the *push* operation do?

Queue - representation on a circular array - push

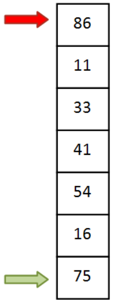
- There are two situations for our queue:

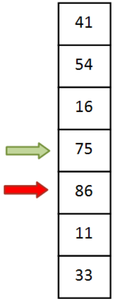




Queue - representation on a circular array - push

- 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] ← e
        q.front ← 1
        q.rear ← 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 ≠ q.cap then
        q.elems[q.rear+1] ← e
        q.rear ← q.rear + 1
    else
        q.elems[1] ← e
        q.rear ← 1
    end-if
end-subalgorithm
```

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?

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?

Queue - representation on a DLL

- If we want to implement a Queue using a doubly linked list, where should we place the *front* and the *rear* of the queue?

Queue - representation on a DLL

- If we want to implement a Queue using a doubly 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 both operations (push or pop) in $\Theta(1)$ complexity.

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

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+

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?

ADT Deque

- The ADT Deque (Double Ended Queue) is a container in which we can insert and delete from both ends:
  - We have *push_front* and *push_back*
  - We have *pop_front* and *pop_back*
  - We have *top_front* and *top_back*
- We can simulate both stacks and queues with a deque if we restrict ourselves to using only part of the operations.

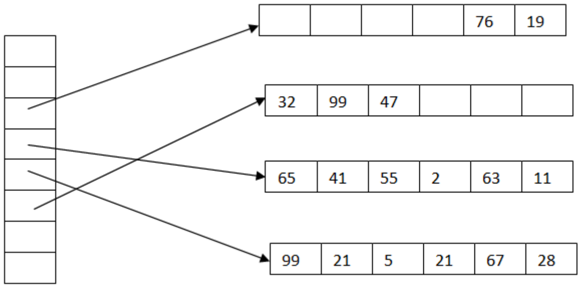
ADT Deque

- Possible representations for a Deque:
  - Circular Array
  - Doubly Linked List
  - A dynamic array of constant size arrays

ADT Deque - Representation

- An interesting representation for a deque is to use a dynamic array of fixed size arrays:
    - Place the elements in fixed size arrays (blocks).
    - Keep a dynamic array with the addresses of these blocks.
    - Every block is full, except for the first and last ones.
    - The first block is filled from right to left.
    - The last block is filled from left to right.
    - If the first or last block is full, a new one is created and its address is put in the dynamic array.
    - If the dynamic array is full, a larger one is allocated, and the addresses of the blocks are copied (but elements are not moved).

Deque - Example



- Elements of the deque: 76, 19, 65, ..., 11, 99, ..., 28, 32, 99, 47

Deque - Example

- Information (fields) we need to represent a deque using a dynamic array of blocks:
    - Block size
    - The dynamic array with the addresses of the blocks
    - Capacity of the dynamic array
    - First occupied position in the dynamic array
    - First occupied position in the first block
    - Last occupied position in the dynamic array
    - Last occupied position in the last block
    - The last two fields are not mandatory if we keep count of the total number of elements in the deque.

ADT Priority Queue

- The ADT Priority Queue is a container in which each element has an associated *priority* (of type *TPriority*).
- In a Priority Queue access to the elements is restricted: we can access only the element with the highest priority.
- Because of this restricted access, we say that the Priority Queue works based on a **HPF - Highest Priority First** policy.

ADT Priority Queue

- In order to work in a more general manner, we can define a relation $\mathcal{R}$ on the set of priorities: $\mathcal{R} : TPriority \times TPriority$
- When we say *the element with the highest priority* we will mean that the highest priority is determined using this relation $\mathcal{R}$.
- If the relation $\mathcal{R} = "\geq"$, the element with the *highest priority* is the one for which the value of the priority is the largest (maximum).
- Similarly, if the relation $\mathcal{R} = "\leq"$, the element with the *highest priority* is the one for which the value of the priority is the lowest (minimum).

Priority Queue - Interface I

- The domain of the ADT Priority Queue:
  $\mathcal{PQ} = \{pq | pq$ is a priority queue with elements $(e, p), e \in TElem, p \in TPriority\}$

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

Priority Queue - Interface II

- init (pq, R)
  - **Description:** creates a new empty priority queue
  - **Pre:** R is a relation over the priorities, $R : TPriority \times TPriority$
  - **Post:** $pq \in \mathcal{PQ}$, pq is an empty priority queue

Priority Queue - Interface III

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

Priority Queue - Interface IV

- push(pq, e, p)
  - **Description:** pushes (adds) a new element to the priority queue
  - **Pre:** $pq \in \mathcal{PQ}, e \in TElem, p \in TPriority$
  - **Post:** $pq' \in \mathcal{PQ}, pq' = pq \oplus (e, p)$

Priority Queue - Interface V

- pop (pq, e, p)
    - **Description:** pops (removes) from the priority queue the element with the highest priority. It returns both the element and its priority
    - **Pre:** $pq \in \mathcal{PQ}$
    - **Post:**$e \in TElem, p \in TPriority$, e is the element with the highest priority from pq, p is its priority. $pq' \in \mathcal{PQ}, pq' = pq \ominus (e, p)$
    - **Throws:** an exception if the priority queue is empty.

Priority Queue - Interface VI

- top (pq, e, p)
  - **Description:** returns from the priority queue the element with the highest priority and its priority. It does not modify the priority queue.
  - **Pre:** $pq \in \mathcal{PQ}$
  - **Post:** $e \in TElem, p \in TPriority$, e is the element with the highest priority from pq, p is its priority.
  - **Throws:** an exception if the priority queue is empty.

Priority Queue - Interface VII

- isEmpty(pq)
    - **Description:** checks if the priority queue is empty (it has no elements)
    - **Pre:** $pq \in \mathcal{PQ}$
    - **Post:**

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

Priority Queue - Interface VIII

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

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

Priority Queue - Interface IX

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

Priority Queue - Representation

- What data structures can be used to implement a priority queue?
  - Dynamic Array
  - Linked List
  - (Binary) Heap

Priority Queue - Representation

- If the representation is a Dynamic Array or a Linked List we have to decide how we store the elements in the array/list:
  - we can keep the elements ordered by their priorities
  - we can keep the elements in the order in which they were inserted

Priority Queue - Representation

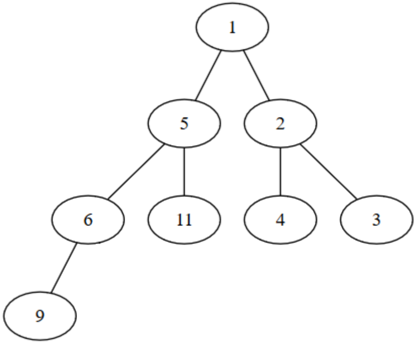
- Complexity of the main operations for the two representation options:

Operation	**Sorted**	**Non-sorted**
push	O(n)	$\Theta(1)$
pop	$\Theta(1)$	$\Theta(n)$
top	$\Theta(1)$	$\Theta(n)$

- What happens if we keep in a separate field the element with the highest priority?

Priority Queue - Representation

- Another representation for a Priority Queue is to use a binary heap, where the root of the heap is the element with the highest priority (the figure contains the priorities only).



Priority Queue - Representation on a binary heap

- When an element is pushed to the priority queue, it is simply added to the heap (and bubbled-up if needed)
- When an element is popped from the priority queue, the root is removed from the heap (and bubble-down is performed if needed)
- Top simply returns the root of the heap.

Priority Queue - Representation

- Let's complete our table with the complexity of the operations if we use a heap as representation:

Operation	**Sorted**	**Non-sorted**	**Heap**
push	O(n)	$\Theta(1)$	$O(log_2 n)$
pop	$\Theta(1)$	$\Theta(n)$	$O(log_2 n)$
top	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$

- Consider the total complexity of the following sequence of operations:
  - start with an empty priority queue
  - push *n* random elements to the priority queue
  - perform pop *n* times

Priority Queue - Applications

- Problems where a priority queue can be used:
  - Triage procedure in the hospitals
  - Scholarship allocation - see Seminar 5
  - Give me a ticket on an airplane (war story from Steven S. Skiena: *The Algorithm Design Manual*, Second Edition, page 118)

Priority Queue - Extension

- We have discussed the *standard* interface of a Priority Queue, one that contains the following operations:
  - push
  - pop
  - top
  - isEmpty
  - init
- Sometimes, depending on the problem to be solved, it can be useful to have the following three operations as well:
  - increase the priority of an existing element
  - delete an arbitrary element
  - merge two priority queues

Priority Queue - Extension

- What is the complexity of these three extra operations if we use as representation a binary heap?
    - Increasing the priority of an existing element is $O(log_2 n)$ if we know the position where the element is.
    - Deleting an arbitrary element is $O(log_2 n)$ if we know the position where the element is.
    - Merging two priority queues has complexity $\Theta(n)$ (assume both priority queues have n elements).

Priority Queue - Other representations

- If we do not want to merge priority queues, a binary heap is a good representation. If we need the merge operation, there are other heap data structures that can be used, which offer a better complexity.
- Out of these data structures we are going to discuss one: the *binomial heap*.

Written test - General Info

- Will be at the first half of seminar 5
- Will last 50 minutes
- **Everybody has to participate at the test with his/her own group!**

Written test - Subjects I

- Each subject will be of the form: Container(ADT) + Representation (a data structure) + Operation
- Containers:
  - Bag
  - Set
  - Map
  - MultiMap
  - List
  - sorted version of the above containers
  - (Sparse)Matrix

Written test - Subjects II

- Data structures
  - Singly Linked List with Dynamic Allocation
  - Doubly Linked List with Dynamic Allocation
  - Singly Linked List on an Array
  - Doubly Linked List on an Array

Written test - Subjects III

- Operation
  - Add
  - Remove
  - Search
  - For Sparse Matrix: Modify value (from 0 to non-zero or from non-zero to 0), search for element from position (i,j)
- Example of a subject: **ADT Set - represented on a doubly linked list on an array - operation: add**.

Written test - Grading

- **1p - Start**
- **0.5p - Specification of the operation** - header, preconditions, postconditions
- **0.5p - Short description of the container**
- **1.25p - representation**
  - 1p - structure(s) needed for the container
  - 0.25p - structure for the iterator for the container

Written test - Grading II

- **4.5p - Implementation of the operation in pseudocode**
  - If you have a data structure with dynamic allocation, you can use the *allocate* and *deallocate/free* operations. If you call any other function(s) in the implementation, you have to implement them.
  - If you have a data structure on an array, you do not need to write code for *resize*-ing the data structure, but you need to show where the resize part would be:

```
...
if s.firstEmpty = -1 then
    @resize
end-if
```

Written test - Grading III

- **1.25p - Complexity**
  - 0.25p - Best Case - with explanations
  - 0.5p - Worst Case - with computations
  - 0.5p - Average Case - with computations
- **1p - Style**
  - Is it general (uses TElem, TComp, a generic Relation)?
  - Is it efficient?
  - etc.