3rd practice

Goals:

1. Implementing types in the operations of which algorithmic patterns are used,

2. Testing,

3. Exception handling.

Priority queue

Implement the type of priority queue. The items of the queue consist of two fields: ⟨priority (integer), data (string)⟩. From the queue, we always take out the element with the highest priority. If there are more maximums, there is no restriction which to get first. Solve the following task: On a programming competition earned points of the different groups are stored in a file. Create the final result of the competition.

PrQueue
- $A : \mathcal{T}[]$ // $\mathcal{T}$ is some known type - $n : \mathbb{N}$ // $n \in 0..A.M$ is the actual length of the priority queue

+ PrQueue($m : \mathbb{N}$){ $A =$ **new** $\mathcal{T}[m]$; $n = 0$ } // create an empty priority queue
+ add($x : \mathcal{T}$) // insert x into the priority queue
+ remMax():$\mathcal{T}$ // remove and return the maximal element of the priority queue
+ max():$\mathcal{T}$ // return the maximal element of the priority queue
+ isFull() : $\mathbb{B}$ {**return** $n == A.M$}
+ isEmpty() : $\mathbb{B}$ {**return** $n == 0$}
+ $\sim$ PrQueue() { **delete** A }
+ setEmpty() {$n = 0$} // reinitialize the priority queue

Type specification:

Type values:	**Type operations:**
PrQueue // ⟨Z, S⟩*	A=(PQ : PrQueue, l:L) l:=PQ.isEmpty()
	A=(PQ : PrQueue, e: ⟨Z, S⟩) PQ.add(e)
	A=(PQ : PrQueue, e: ⟨Z, S⟩) Pre=(¬PQ.isEmpty()) e:=PQ.remMax()
	A=(PQ : PrQueue, e: ⟨Z, S⟩) Pre=(¬PQ.isEmpty()) e:=PQ.max()

Solution: There are two possibilities. We need an array for both and a variable that stores the number of elements.

(1) **Unordered array**. In the queue, the items are not ordered according to their priority.
   a. **isEmpty:** can be decided by the length: Θ(1)

b. **add:** (if there is space) we put the new item behind the last element: $\Theta(1)$
c. **remMax:** if the queue is not empty, we can get the wanted item with maximum search. Then, the last element of the array can be put in the place of this got element: $\Theta(n)$
d. **max:** if the queue is not empty, we can get the wanted item with maximum search: $\Theta(n)$

(2) **Ordered array**. In the queue, the items are ordered according to their priority.
a. **isEmpty:** can be decided by the length: $\Theta(1)$
b. **add:** (if there is space) the new item has to be put to its place. The greater items in the queue have to be shifted right: $O(n)$
c. **remMax:** if the queue is not empty, we just simply take the last element of the array: $\Theta(1)$
d. **max:** if the queue is not empty, we just read the last item of the array: $\Theta(1)$

Which is better? There are more $\Theta(1)$ operations in the ordered version, but the add operation is more consuming, as we have to shift half of the elements in general and writing is always more consuming then just reading the items. We can also check which operations are used more frequently to be able to solve the given task. But at the end, we can say that there are no big differences between the two versions. Now, the unordered version is shown.

In the next type (map), the ordered version is going to be discussed.

Type representation:	**Type implementation:**
vec: Item* – array of the elements Item=rec(pr: $\mathbb{Z}$, data: $\mathbb{S}$)	The programs are given in the structograms below.

PQ.empty() — constructor

|vec| := <>

l := PQ.isEmpty()

l := |vec| = 0

PQ.add(e)

vec := vec⊕<e>

The rest of the operations require an algorithmic pattern, so their specification is given, too.

e:=PQ.remMax():

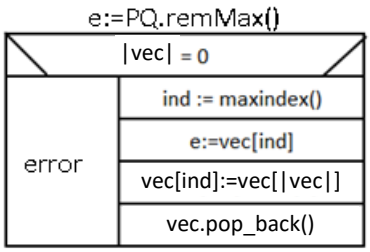
A=(vec:Item*, e:Item, err:$\mathbb{L}$) (err is necessary only if we can call this method on empty queue.)

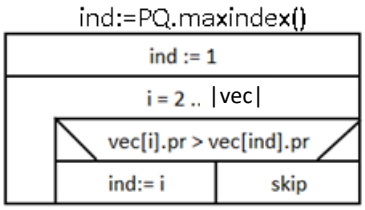
Pre=(vec=vec') (*We could say that |vec|>0. In that case, err is not needed.*)

$$\text{Post}=(\text{err} = (|\text{vec}'|=0) \wedge \neg \text{err} \rightarrow ((\text{ind},\text{elem})=\max_{i=1}^{|\text{vec}'|}(\text{vec}'[i].\text{pr}) \wedge e=\text{vec}'[\text{ind}] \wedge$$

$$\forall i \in [1..|\text{vec}'|-1]: (i\neq\text{ind} \rightarrow \text{vec}[i]=\text{vec}'[i]) \wedge \text{vec}[\text{ind}]=\text{vec}'[|\text{vec}'|] \wedge |\text{vec}| = |\text{vec}'|-1))$$

3rd practice



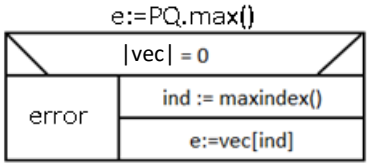


e:=max(PQ):

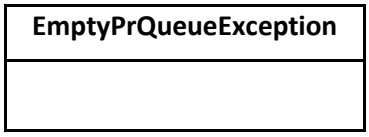
A=(vec:Item*, e:Item, err:$\mathbb{L}$) (err is necessary only if we can call this method on empty queue.)

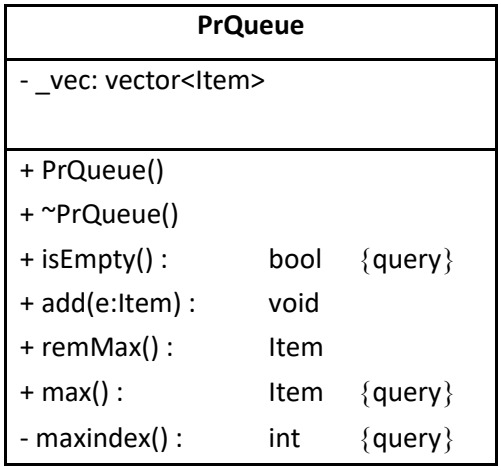
Pre=(vec=vec') (*We could say that |vec|>0. In that case, err is not needed.*)

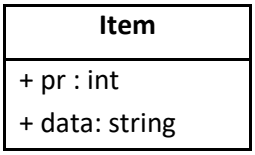
Post=(Pre $\wedge$ err = (|vec|=0) $\wedge$ $\neg$ err $\rightarrow$ ((ind,item)=$\max_{i=1}^{|vec|}$(vec[i].pr) $\wedge$ e=vec[ind]))



UML plan:







Decisions:

The array is implemented as vector<Item>, as it is much more effective. We can use the vector's push_back() and pop_back() methods. Attribute *n* is not implemented, as the vector's size() method is enough. The vector cannot be full, thus, attribute *m* can be left, too. (When the vector reaches its actual size limit, it doubles its memory allocation. Initially every vector is empty, so we do not have to write anything into the constructor).

3rd practice

Unit testing:

This is the point where the students meet the automatic testing first. They need to know how to create proper test cases. The concrete testing environment is shown on the lab.

Test cases according to the operations:

- constructor (no need to test)
- isEmpty() (try the empty and not empty states)
- add(e:Item) (adding items one after the other and check their position)
- max() (typical test cases of maximum search)
- maxindex() (typical test cases of maximum search)
- remMax() (typical test cases of maximum search + check the position of the items)
  In the test cases, only the priority is shown even if every item has a data attribute, too.

remMax()			
Test case	**Vector**	**Result**	**New vector**
Empty array	<>	error (exception)	<>
Array of one element	<3>	3	<>
Array of more elements:			
First is the maximum	<5,2,3>	5	<3,2>
Last is the maximum	<1,2,3>	3	<1,2>
Middle is the maximum	<1,3,2>	3	<1,2>
Not trivial, first and last are the maximums	<5,2,5'>	5	<5',2> (az adat rész segítségével ellenőrizhető, hogy az elsőt vettük ki)
Not trivial, middle and last are the maximums	<1,3,3'>	3	<1,3'>
Not trivial, all are maximums	<3,3',3''>	3	<3'',3'>
More remMax()s after each other, then add()	<2,3,1>	3 2 1 3 2 1	<2,1> remMax() <1> remMax() <> add(3) add(2) add(1) remMax() <1,2> remMax() <1> remMax() <>

3rd practice

Solution of the given task (with unique group names in the competition):

A=(a : Item^n, cout : Item*) *remark: input is an array, output is a sequence*

Pre=(a = a')

Post=(a = a' $\wedge$ cout = (elements of *a* in an ascending order))

Postcondition formally (just mention it):
Post=(a = a' $\wedge$ |cout|=n $\wedge$ $\forall$i $\in$ [1..n]: a[i] $\in$ cout $\wedge$ $\forall$i $\in$ [1..n]: cout$_i$ $\in$ a $\wedge$ $\forall$i $\in$ [1..n-1]: cout$_i$.pr $\geq$ cout$_{i+1}$.pr)