Sorozattal reprezentált típusok megvalósítása I. rész

1. **Feladat: Diagonális mátrix**:

Az első beadandó minta dokumentáció és program ezt a feladatot oldja meg részletesen.

Típusdefiníció:

Diag($\mathbb{R}^{n\times n}$)	c := a+b	(a, b, c : Diag($\mathbb{R}^{n\times n}$))
	c := a·b	(a, b, c : Diag($\mathbb{R}^{n\times n}$))
	e := a[i,j]	(a : Diag($\mathbb{R}^{n\times n}$), i,j : $\mathbb{N}$, e : $\mathbb{R}$)
	a[i,j] := e	(a : Diag($\mathbb{R}^{n\times n}$), i,j : $\mathbb{N}$, e : $\mathbb{R}$) //ha i=j
x:$\mathbb{R}^n$	$\forall i\in[1..n]$: c.x[i] := a.x[i]+b.x[i]	
	$\forall i\in[1..n]$: c.x[i] := a.x[i]·b.x[i]	
	ha i=j akkor e := a.x[i] különben e := 0.0	
	ha i=j akkor a.x[i] := e	

2. Feladat: Alsó háromszög mátrix

Valósítsuk meg az alsó háromszög mátrix típust (a mátrixok a főátlójuk felett csak nullát tartalmaznak)! Ilyenkor elegendő csak a főátló és az alatti elemeket reprezentálni egy sorozatban. Implementáljuk a mátrix *i*-edik sorának *j*-edik elemét visszaadó műveletet, valamint két mátrix összegét és szorzatát!

7 x 7 -es alsó háromszög mátrix

	1	2	3	4	5	6	7
1	1						
2	2	3					
3	4	5	6				
4	7	8	9	10			
5	11	12	13	14	15		
6	16	17	18	19	20	21	
7	22	23	24	25	26	27	28

A sorszámok mutatják, hogy hányadik lesz a mátrix elem a tömbben.

A mátrix elhelyezése sorfolytonosan egy egydimenziós tömbben

(a biztosan nulla értékű elemeket felesleges tárolni)

1	[1,1]
2	[2,1]
3	[2,2]
4	[3,1]
5	[3,2]
6	[3,3]
27	[7,6]
28	[7,7]
29	0

a nulla értéket is tárolhatjuk egy példányban

Az index függvény rendeli hozzá a mátrix elem idexeihez a tárolásnak megfelelő indexet:

$$ind(i,j) = j + \sum_{k=1}^{i-1} k = j + \frac{i(i-1)}{2} \quad , ha\ 1 \le j \le i \le n$$

Típusdefiníció:

AHM($\mathbb{R}^{n\times n}$)	c := a+b (a, b, c : AHM($\mathbb{R}^{n\times n}$)) // azonos n-re
	c := a*b (a, b, c : AHM($\mathbb{R}^{n\times n}$)) // azonos n-re
	e := a[i,j] (a : AHM($\mathbb{R}^{n\times n}$), i,j : $\mathbb{N}$, e : $\mathbb{R}$)
	a[i,j] := e (a : AHM($\mathbb{R}^{n\times n}$), i,j : $\mathbb{N}$, e : $\mathbb{R}$) //ha i ≥ j
x:$\mathbb{R}^{n(n+1)/2}$	$\forall i\in[1..n(n+1)/2]$: c.x[i] := a.x[i]+b.x[i]
	$\forall i,j\in[1..n]$: ha i≥j akkor $c.x\left[\frac{i(i-1)}{2}+j\right] := \sum_{k=j}^{i} a.x\left[\frac{i(i-1)}{2}+k\right]\cdot b.x\left[\frac{k(k-1)}{2}+j\right]$
	ha i≥j akkor e := $a.x\left[\frac{i(i-1)}{2}+j\right]$ különben e := 0.0
	ha i≥j akkor $a.x\left[\frac{i(i-1)}{2}+j\right]$:= e

3. Feladat – prioritásos sor

Készítsünk maximum prioritásos sort. Az elemek két mezőből állnak (prioritás (egész), adat (szöveg)). A sorból mindig a legnagyobb prioritású elemet vesszük ki (több legnagyobb esetén nem meghatározott, hogy melyiket).

-Egy igen hatékony ábrázolás a kupaccal megvalósított prioritásos sor, melyet ebben a félévben Algoritmusok és adatszerkezetek tárgyból fogunk tanulni. Itt most a tömbbel reprezentált változatokat fogjuk vizsgálni. -

Típus értékek:	**Típus műveletek:**
PrQueue a maximum prioritásos sorok halmaza, amely soroknak az elemei $\mathbb{Z}$ × $\mathbb{S}$ típusú párok.	**pq := setEmpty(pq)** pq : PrQueue *// Kiüríti a pr sort* **l := isEmpty(pq)** pq : PrQueue, l : $\mathbb{L}$ *//Igazat ad, ha üres a pr sor, hamisat ha nem.* **pq := add(pq, e)** pq : PrQueue, e : $\mathbb{Z}$ × $\mathbb{S}$ *//Betesz egy új elemet a pr sorba.* **pq, e := remMax(pq)** pq : PrQueue, e : $\mathbb{Z}$ × $\mathbb{S}$ *//Kiveszi az egyik legnagyobb prioritású elemet. Fontos, hogy a sor nem lehet üres.* **e := getMax(pq)** pq : PrQueue, e : $\mathbb{Z}$ × $\mathbb{S}$ *//Visszadja az egyik legnagyobb prioritású elemet, nem veszi ki. Fontos, hogy a sor nem lehet üres.*

A műveleteket objektum-orientált stílusban is megadhatjuk:

pq.setEmpty()

l := pq.empty() {query}

pq.add(e)

e := pq.remMax()

e:= pq.getMax() {query}

Megvalósításnál két módszer között választhatunk. Mindkettőhöz szükségünk van egy sorozatra, de a műveletek futási ideje eltérő lehet.

(1) **Rendezetlen sorozatot** használunk. Az elemeket folyamatosan helyezzük el a sorozatban. Nem tudjuk, melyik közülük a legnagyobb prioritású.
   a. **setEmpty** : üres sorozatot készít. $\Theta(1)$ (Bár nem tudjuk biztosan, hogy a C++ nyelv vector<> clear() metódusa mit is csinál pontosan.)
   b. **isEmpty:** hosszból azonnal eldönthető. $\Theta(1)$
   c. **add:** az új elemet a sorozat végéhez fűzzük. $\Theta(1)$
   d. **remMax:** ha nem üres a sorozat, a tanult maximum kiválasztás algoritmussal megkeressük az egyik legnagyobb prioritású elemet, és kivesszük. $\Theta(n)$
   e. **getMax:** ha nem üres s sor, a tanult maximum kiválasztás algoritmussal megkeressük az egyik legnagyobb prioritású elemet, és visszaadjuk. A sorozat nem változik. $\Theta(n)$

   Észrevehető, hogy a getMax() művelet $\Theta(1)$-re csökkenthető, ha egy külön változóban nyilvántartjuk a legnagyobb elem indexét, de ekkor ennek folyamatos karbantartásáról gondoskodni kell.

(2) **Rendezett sorozatot** használunk. A sorozatban az elemek prioritás szerint növekvő a sorrendben vannak.
   a. **setEmpty** : üres sorozatot készít. $\Theta(1)$
   b. **isEmpty:** hosszból azonnal eldönthető $\Theta(1)$
   c. **add:** az új elemet betesszük a rendezettség szerinti helyére, amelyet lineáris kereséssel vagy kiválasztással (esetleg logaritmikus kereséssel) kell megkeresnünk, de utána a beszúrás a nagyobb prioritású elemeinek mozgatásával oldható csak meg. (Vesd össze az Algoritmusok tantárgyból tanult beszúró rendezéssel.) $O(n)$
   d. **remMax:** ha nem üres a sorozat, akkor a rendezettség miatt a sorozat utolsó eleme az egyik legnagyobb prioritású elem. Azt ki is kell vennünk a sorozatból. $\Theta(1)$
   e. **getMax:** ha nem üres a sorozat, akkor a rendezettség miatt a sorozat utolsó eleme az egyik legnagyobb prioritású elem. A sorozat nem változik. $\Theta(1)$

Látjuk, hogy a prioritásos sor ábrázolásánál nem tudjuk a konstans műveletigényt tartani mindegyik művelet esetén. Mindkét módszernél legalább az egyik művelet „lineáris” műveletigényű, azaz az elemek számával arányos. Van egy pici különbség a két lineáris igényű művelet között: míg a maximum kiválasztás minden esetben megvizsgálja az összes elemet, addig a rendezett részbe való beszúrás korábban is leállhat. Viszont az adatmozgatás költségesebb művelet, mint az egészek összehasonlítása.

Ha egy előre megadott méretű tömböt használnánk, amelynél külön megadjuk, hogy az első hány eleme van kitöltve, akkor a beszúrás néhány elem hátra csúsztatását igényli, amely legrosszabb esetben az összes elemet odébb csúsztatja. A remMax() tehát rendezett esetben is $O(n)$. Igaz, hogy

átlagosan az elemek felét érinti majd a hátrébb csúsztatás, tehát itt várhatóan fele annyi lesz az átlagos lépésszám, viszont az elemek hátrébb csúsztatása jóval költségesebb, mintha csak kiolvassuk őket, amelyet a remMax() rendezetlen esetben csinál.

Ha C++ nyelven a vectort<> használjuk, akkor az erase(), illetve az insert() műveletekkel tetszés szerint változtathatjuk a sorozatot. De ezen műveletek hátterében adatmozgatás történik.

Majd a következő feladatban (asszociatív tömb) a rendezett ábrázolást használjuk, ezért itt most a **rendezetlen tömbbel való ábrázolást választjuk**.

Típus reprezentáció:

vec: Item*

- a prioritásos sor elemeit rendezetlenül tároló sorozat, ahol Item = rec(pr : $\mathbb{Z}$, data : $\mathbb{S}$)

Típusműveletek implementációja:

l := isEmpty(pq))

l := |vec| =0	(C++ vector: size())

pq:=setEmpty(pq)

vec := <>	(C++ vector: clear())

pq := add(pq, e)

vec := vec ⊕ <e>	(C++ vector: push_back())

pq, e := remMax(pq)

err := (|vec|=0)	
¬err	
max, ind := MAX(vec)	**Hiba** üres a prioritásos sor
e := vec[ind]	
vec[ind] := vec[|vec|]	
pop_back(vec)	

e := getMax(pq)

err := (|vec|=0)	
¬err	
max, ind := MAX(vec)	**Hiba:** üres a prioritásos sor
e := vec[ind]	

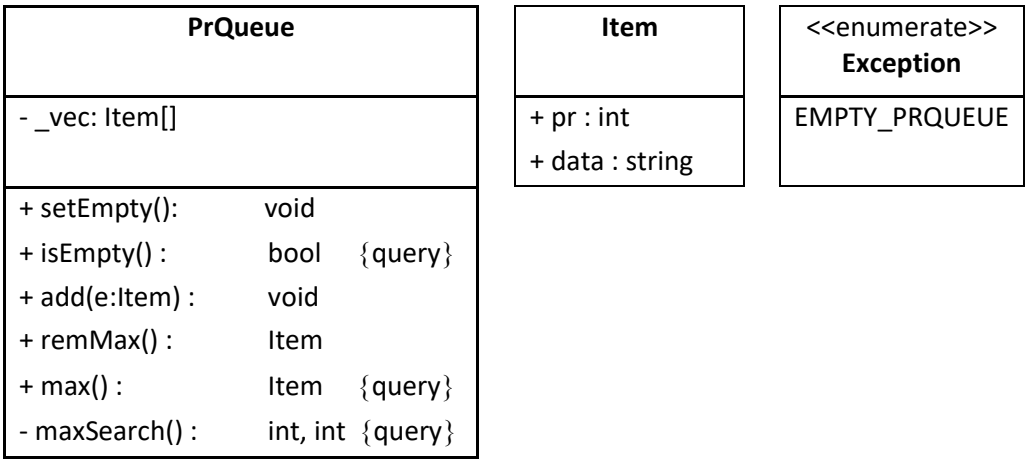
Több művelet is használja a max, ind := MAX(vec) segédműveletet (privát metódust). Ennek a műveletnek a specifikációja:

A = (vec:(Item)*, e:Item)
Item=rec(pr:$\mathbb{Z}$, data:$\mathbb{S}$)
Ef = (vec=vec' $\wedge$ \|vec\|>0)
Uf = (Ef $\wedge$ (max, ind) = $MAX_{i=1\ ..\ |vec|}$ vec[i].pr $\wedge$
$\wedge$ e = vec[ind])
ahol max:$\mathbb{Z}$ és ind:$\mathbb{N}$

max, ind := MAX(vec)

max, ind := vec[1].pr, 1	
i = 2 .. |vec|	
max < vec[ind].pr	
max, ind := vec[ind].pr, i	—

Prioritásos sor megvalósítása C++ osztállyal: UML diagram (érdemes táblás órán is megbeszélni):



Egy feladat megoldása prioritásos sorral:

Egy programozási versenyen csapatok indultak. Ismerjük a csapatok nevét, és a versenyen elért pontszámukat. Készítsünk listát a csapatok eredményéről csökkenő sorrendben. (Feltehető, hogy a csapatok neve egyedi.)

$A = (t : Item^n, cout : Item^*)$

$Ef = (t = t_0)$

$Uf = (t = t_0 \wedge cout = (t \text{ elemeit tartalmazza, pr értékei szerint monoton csökkenően felsorolva}))$

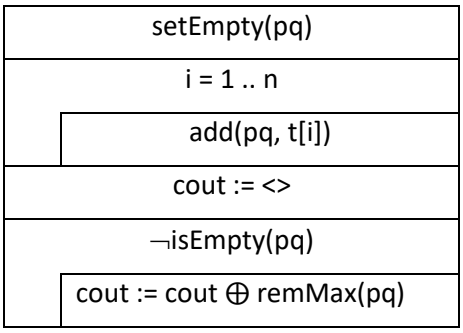
Az utófeltételt most még nem tudjuk formalizálni, a felsorolós programozási tételekre történő áttérés után lesz lehetőség rá. Két lépésből fog állni a megoldás: elsőként a sorozat valamennyi elemét bepakoljuk egy prioritásos sorba, majd onnan a remMax() művelet segítségével mindegyiket kivesszük, és egy kimenetre kiírjuk. Mindkét lépés egy összegzés.

$pq : PrQueue \wedge pq = \bigcup_{i=1..n} \{t[i]\}$

Az unió művelet a prioritásos sorba tesz be egy új elemet (azaz szemantikailag megegyezik az add() művelettel) úgy, mintha a prioritásos sorhoz unióznánk az új elemet tartalmazó prioritásos sort. Ennek a műveletnek a neutrális eleme az üres prioritásos sor.

$cout = \bigoplus^{\neg isEmpty() pq} remMax(pq))$

Az összefűzés addig tart, amíg a pq prioritásos sor nem lesz üres.



Prioritásos sor megvalósításának tesztelése:

Vegyük sorra, milyen tesztelést képzelnének el az egyes metódusokhoz (példákat a forráskódban lehet látni):

- setEmpty() (végrehajtása után az isEmpty() igazat ad)
- isEmpty() (üres / nem üres állapotra kipróbáljuk)
- add(Item e) (egymás után berakunk elemeket, majd ellenőrizzük az elhelyezésüket)
- maxsearch() (max és az ind vizsgálandó)
- max() (a maxsearch()-höz képest még a hibás esetet kell tesztelni)
- remMax() (a max()-hoz képest még a tömb átrendeződését is ellenőrizzük)

remMax() tesztelése			
teszteset	**teszt tömb**	**eredmény**	**tömb új tartalma**
üres intervallum	<>	hiba (kivétel dobás)	<>
egy elemű	<3>	3	<>
több elemű esetek:			
első a legnagyobb	<5,2,3>	5	<3,2>
utolsó a legnagyobb	<1,2,3>	3	<1,2>
belső a legnagyobb	<1,3,2>	3	<1,2>
nem egyértelmű, első és utolsó a legnagyobb	<5,2,5'>	5	<5',2> (az adat rész segítségével ellenőrizhető, hogy az elsőt vettük ki)
nem egyértelmű, belső és utolsó a legnagyobb	<1,3,3'>	3	<1,3'>
mind egyforma	<3,3',3''>	3	<3'',3'>
több egymás utáni remMax(), majd add hatása	<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() <>