6. Elemi adatszerkezetek és adattípusok

Adatszerkezet (data structure) alatt adatok tárolásának és elrendezésének egy lehetséges módját értjük, ami lehetővé teszi a tárolt adatok elérését és módosítását, beleértve újabb adatok eltárolását és tárolt adatok törlését is. [4]

Nincs olyan adatszerkezet, ami univerzális adattároló lenne. A megfelelő adatszerkezetek kiválasztása vagy megalkotása legtöbbször a programozási feladat megoldásának alapvető része. A programok hatékonysága nagymértékben függ az alkalmazott adatszerkezetektől.

Az *adattípus (data type)* a mi értelmezésünkben egy adatszerkezet, a rajta értelmezett műveletekkel együtt.

Az *absztrakt adattípus (ADT)* esetében nem definiáljuk pontosan az adatszerkezetet, csak – informálisan – a műveleteket. Az ADT megvalósítása két részből áll:
– *reprezentálása* során megadjuk az adatszerkezetet,
– *implementálása* során pedig a műveletei kódját.

Az adattípusok megvalósítását gyakran UML jelöléssel, osztályok segítségével fogjuk leírni. A lehető legegyszerűbb nyelvi elemekre szorítkozunk. (Nem alkalmazunk sem öröklődést, sem template-eket, sem kivételkezelést.)

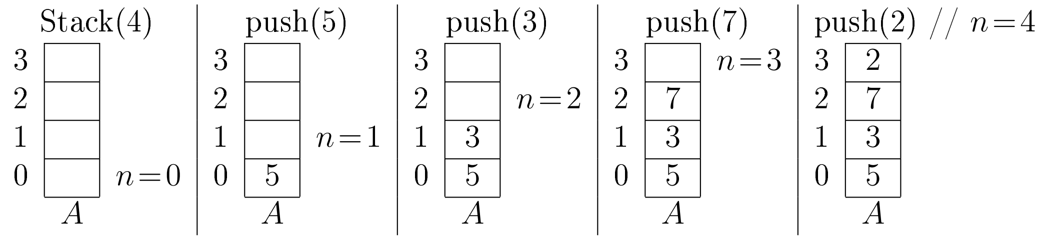
6.1. Verem (Stack)

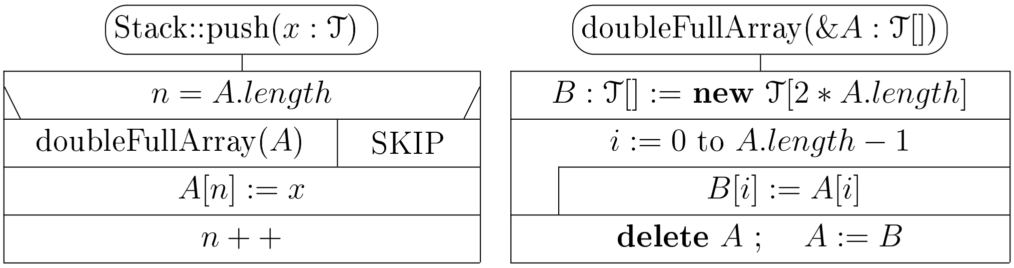
A *verem (stack)* adattípus LIFO (Last-In First-Out) adattároló, aminél tehát mindig csak az utoljára benne eltárolt, és még benne lévő adat érhető el, illetve törölhető. Tipikus műveleteit az alábbi megvalósítás mutatja.

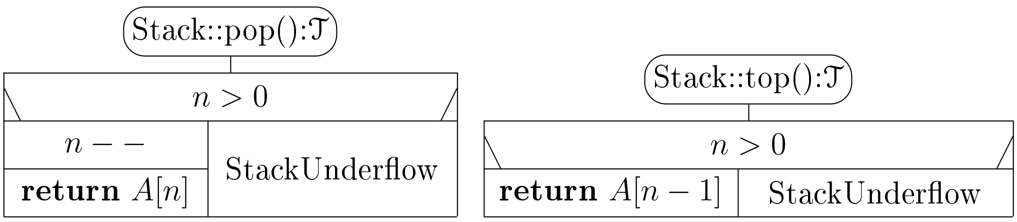
Stack
$-\ A : \mathcal{T}[]$ // $\mathcal{T}$ is some known type ; $A.length$ is the physical
$-$ constant $m0 : \mathbb{N}_+ := 16$ // size of the stack, its default is m0.
$-\ n : \mathbb{N}$ // $n \in 0\,..\,A.length$ is the actual size of the stack
$+$ Stack($m : \mathbb{N}_+ := m0$) {$A :=$ **new** $\mathcal{T}[m]$; $n := 0$} // create empty stack
$+ \sim$ Stack() { **delete** A }
$+$ push($x : \mathcal{T}$) // push x onto the top of the stack
$+$ pop() : $\mathcal{T}$ // remove and return the top element of the stack
$+$ top() : $\mathcal{T}$ // return the top element of the stack
$+$ isEmpty() : $\mathbb{B}$ {**return** $n = 0$}
$+$ setEmpty() {$n := 0$} // reinitialize the stack

A vermet most dinamikus tömb ($A : \mathcal{T}[]$) segítségével reprezentáljuk, ahol $A.length$ a verem fizikai mérete, $\mathcal{T}$ a verem elemeinek típusa. Az egyszerűség kevéért, alapértelmezésben feltesszük, hogy van elég szabad memória a **new** utasításaink számára.

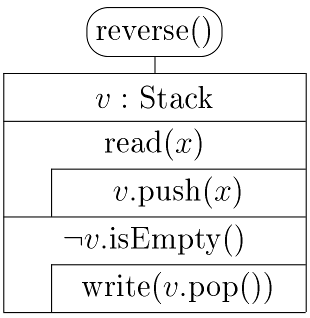
Néhány veremművelet







Példa a verem egyszerű használatára: Az input adatok kiírása fordított sorrendben. Feltesszük, hogy a read(&x : $\mathcal{T}$):$\mathbb{B}$ függvény a kurrens inputról olvas, ami akkor sikeres, és tér vissza igazzal, ha nincs még vége az inputnak. Ilyenkor beolvassa x-be a következő input adatot. A read(x) akkor sikertelen, és tér vissza hamissal, ha vége van az inputnak. Ekkor x értéke definiálatlan. A write(x) a kurrens outpura írja x értékét.



A vermek műveleteit – a push művelet kivételével – egyszerű, rekurziót és ciklust nem tartalmazó metódusokkal írtuk le. Ezért mindegyik műveletigénye $\Theta(1)$, ami – legalábbis együtt az összes elvégzett különféle művelet átlagos műveletigényét tekintve – alapkövetelmény minden verem megvalósítással kapcsolatban. A push műveletre nyilván $mT(n) \in \Theta(1)$ és $MT(n) \in \Theta(n)$. A 6.1. feladat szerint azonban a push műveletre is $AT(n) \in \Theta(1)$.

6.1. Feladat.* *Lássuk be, hogy a fenti Stack osztály* push *műveletére is teljesül $AT(n) \in \Theta(1)$.*

6.2. Sor (Queue)

A *sor (queue)* adattípus FIFO (First-In First-Out) adattároló, aminél tehát a még benne lévő adatok közül adott pillanatban csak a legrégebben benne eltárolt érhető el, illetve törölhető. Tipikus műveleteit az alábbi megvalósítás mutatja.

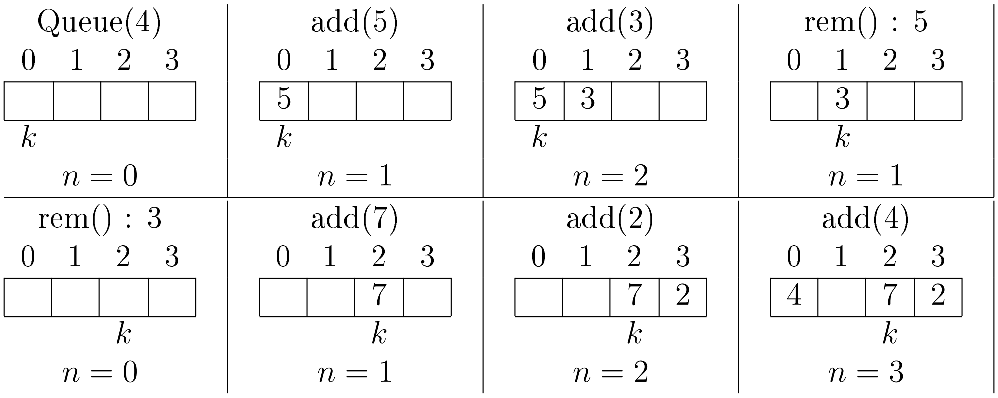
A sort nullától indexelt dinamikus tömb ($Z : \mathcal{T}[]$) segítségével reprezentáljuk, ahol az *Z.length* a sor fizikai mérete, $\mathcal{T}$ a sor elemeinek típusa. Az egyszerűség kevéért, alapértelmezésben feltesszük, hogy van elég szabad memória a **new** utasításaink számára.

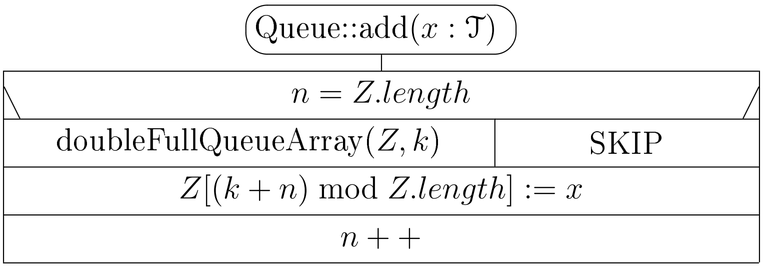
[A vermet és a sort természetesen ábrázolhatjuk láncolt listák segítségével is (ld. a 7. fejezetet), a verem esetében az egyszerű láncolt lista elejét a verem tetejének tekintve, a sor esetében pedig lista végéhez közvetlen hozzáférést biztosítva.]

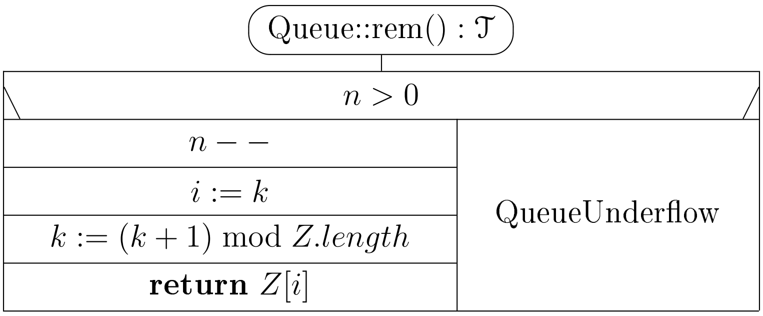
Queue
$- Z : \mathcal{T}[]$ // $\mathcal{T}$ is some known type ; $Z.length$ is the physical $-$ constant $m0 : \mathbb{N}_+ := 16$ // length of the queue, its default is m0. $- n : \mathbb{N}$ // $n \in 0..Z.length$ is the actual length of the queue $- k : \mathbb{N}$ // $k \in 0..(Z.length - 1)$: the starting position of the queue in Z

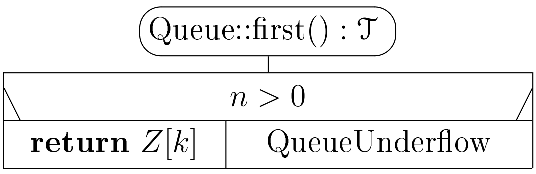
+ Queue($m : \mathbb{N}_+ := m0$){ $Z :=$ **new** $\mathcal{T}[m]$; $n := 0$; $k := 0$ }
// create an empty queue
+ add($x : \mathcal{T}$) // join x to the end of the queue
+ rem() : $\mathcal{T}$ // remove and return the first element of the queue
+ first() : $\mathcal{T}$ // return the first element of the queue
+ length() : $\mathbb{N}$ {**return** n}
+ isEmpty() : $\mathbb{B}$ {**return** $n = 0$}
+ $\sim$ Queue() { **delete** Z }
+ setEmpty() {$n := 0$} // reinitialize the queue

Egy sor néhány művelete









Az add művelet doubleFullQueueArray(Z, k) segédeljárásával kapcsolatban ld. a 6.2. feladatot!

A sorok műveleteit – az add művelet kivételével – egyszerű, rekurziót és ciklust nem tartalmazó metódusokkal írtuk le. Ezért mindegyik műveletigénye $\Theta(1)$, ami – legalábbis együtt az összes elvégzett különféle művelet átlagos műveletigényét tekintve – alapkövetelmény minden sor megvalósítással kapcsolatban. Az add műveletre nyilván $mT(n) \in \Theta(1)$ és $MT(n) \in \Theta(n)$. A 6.2. feladat szerint azonban az add műveletre is $AT(n) \in \Theta(1)$.

6.2. Feladat. *Írjuk meg a* Queue *osztály* add *műveletéhez tartozó* doubleFullQueueArray(Z, k) *segédeljárást! Ha az* add *művelet úgy találja, hogy már tele van a tömb, cserélje le nagyobbra, pontosan kétszer akkorára! Ügyeljünk a nagyságrendileg optimális átlagos futási időre, továbbá arra, hogy a Z tömbben a sor akár ciklikusan is elhelyezkedhet!*

Lássuk be, hogy így az add *műveletre $mT(n) \in \Theta(1)$ és $MT(n) \in \Theta(n)$, valamint $AT(n) \in \Theta(1)$ is teljesül!)*

6.3. Feladat. *Tegyük fel, hogy adott a* Stack *osztály, ami a* Stack(), ∼Stack(), push(x:$\mathcal{T}$), pop():$\mathcal{T}$, isEmpty():$\mathbb{B}$ *műveletekkel (elvileg) korlátlan méretű vermeket tud létrehozni és kezelni. Feltehető, hogy mindegyik művelet átlagos futási ideje $\Theta(1)$.*

Valósítsuk meg a Queue *osztályt két verem (és esetleg néhány egyszerű segédváltozó) segítségével, a következő műveletekkel:* Queue(), add(x:$\mathcal{T}$), rem():$\mathcal{T}$, length():$\mathbb{N}$. *Miért nincs szükség destruktorra? Mit tudunk mondani a műveletigényekről? Elérhető-e valamilyen értelemben a $\Theta(1)$ átlagos műveletigény?*

6.4. Feladat. *Tegyük fel, hogy adott a* Queue *osztály, ami a* Queue(), ∼Queue(), add(x:$\mathcal{T}$), rem():$\mathcal{T}$, length():$\mathbb{N}$ *műveletekkel (elvileg) korlátlan méretű sort tud létrehozni és kezelni. Feltehető, hogy mindegyik művelet átlagos futási ideje $\Theta(1)$.*

Valósítsuk meg a Stack *osztályt egy sor (és esetleg néhány egyszerű segédváltozó) segítségével, a következő műveletekkel:* Stack(), push(x:$\mathcal{T}$), pop():$\mathcal{T}$, isEmpty():$\mathbb{B}$. *Miért nincs szükség destruktorra? Mit tudunk mondani a műveletigényekről?*