1.
Pro kterou/které z následujících datových struktur

1) pole
2) fronta
3) zásobník
4) tabulka
5) seznam
6) strom
7) množina

a) platí, že má předem daný počet prvků?
Musí to být statická struktura. Čili pole, nebo statická tabulka.

b) platí, že kdykoliv lze jednou operací přečíst kterýkoliv prvek?
pole - přečtu prvek pro libovolný index
tabulka - přečtu prvek pro libovolný klíč
množina - pro libovolný prvek zjistím, zda v množině je, nebo není.

d) platí, že kdykoliv lze maximálně dvěma operacemi přečíst první prvek?
pole - get(pole, lower(pole))
lower vrátí dolní mez indexů a get přečte prvek s timto indexem
fronta - jednou operací front(pole) přečtu prvek v čele fronty, který je "první" na řadě
seznam - Použiji operace "First" (přesune ukazovátko na začátek seznamu)
a pak "Read" (přečte prvek, na který ukazovátko ukazuje)

e) platí, že kdykoliv lze maximálně třemi operacemi přečíst poslední prvek?
zásobník- jednou operací POP přečtu poslední prvek ve smyslu "naposledy vložený"
seznam - last, prev, read
- presneji read(prev(last(S))), kde S je proměnná typu seznam
last - přesune ukazovátko za poslední prvek seznamu
prev - přesune ukazovátko na poslední prvek seznamu
read - přečte poslední prvek, protože na něj ukazuje ukazovátko
pole - get(pole, max(pole))
max - vrátí horní mez indexů
get - přečte prvek s maximálním indexem

2.
Mapovací funkce dvourozměrného pole

- o určí, zda je v paměti dostatek místa pro celé pole
- ✓ spočte polohu libovolného prvku pole v paměti
- o vypočte, od které adresy bude pole v paměti uloženo
- o zajistí, aby se pole nepřekrývalo v paměti s jinými datovými strukturami
- o umožní, aby pole sdílelo paměť s jinými datovými strukturami

Mapovací funkce zajišťuje, aby jednotlivé prvky pole byly v paměti „k nalezení“ podle svých indexů a to tak, že index každého prvku pole dokáže přepočítat na adresu tohoto prvku v paměti. O další implementační podporu vyjádřenou variantami a), d), e) se

nestará. Adresu prvního prvku nevypočítává, tu musí znát. Varianta c) tedy také padá a zbývá jen varianta b).

3.
ADT Fronta specifikuje

a) které datové struktury budou použity při implementaci operací Insert a Delete
b) které datové struktury budou použity při implementaci operací Push a Pop
c) vlastnosti operací Insert a Delete
d) vlastnosti operací Push a Pop
e) vlastnosti operací Push a Insert

Abstraktní datový typ nikomu žádnou implementaci nepředepisuje, to by nebyl abstraktní. Varianty a) a b) tedy nemohou platit. Fronta nemá zároveň operaci push a insert. Obě totiž obstarávají vkládání prvku, ve frontě se však vkládá jen na konec, dvě operace pro vkládání jsou zbytečné, varianta e) padá. Operace Push a Pop jsou typické pro zásobník, varianta d) padá také a zbývá jen varianta c).

4.
Následující datová struktura má definován index

a) fronta
b) tabulka
c) strom
d) množina
e) žádná z uvedených

Ze základních ADT má index definováno jen pole. To se v nabízeném seznamu nevyskytuje, takže platí možnost e) – žádná z uvedených.

5.
V následující datové struktuře lze maximálně třemi operacemi přečíst poslední prvek

a) fronta
b) zásobník
c) tabulka
d) strom
e) žádná z uvedených

V tabulce ani ve stromu žádný poslední prvek definován není, možnosti c) a d) odpadají.
Poslední prvek ve frontě obsahující alespoň pět prvky se na začátek dostane po nejméně čtyřech operacích Delete a žádný jiný přístup k němu není, takže možnost a) odpadá také.
V zásobníku je naopak poslední prvek na vrcholu, takže je pomocí operací Pop či Top snadno přístupný i ke čtení. Platí možnost b).

6.
Datový typ TABULKA

a) je lineární datová struktura
b) má definován index
c) je nelineární datová struktura
d) má definováno ukazovátko

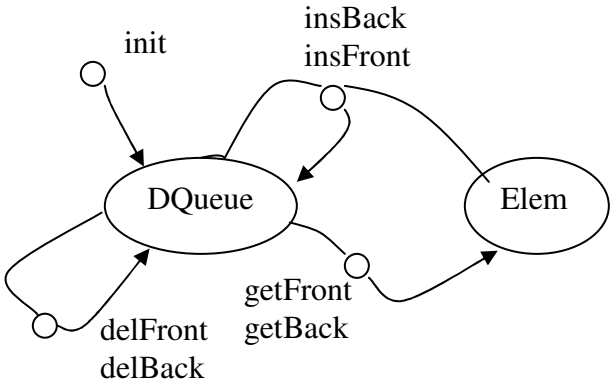
Tabulka nemá definován index, protože k jejím položkám se přistupuje pomocí klíčů. Nemá ani definováno ukazovátko. Její prvky nemají ani jednoznačné následníky ani předchůdce, což je vlastnost charakteristická pro lineární struktury. Takže tabulka není lineární struktura, platí c).

7.
Datový typ SEZNAM

a) má definován index
b) má vždy definován klíč prvku
c) je nelineární datová struktura
d) je lineární datová struktura

Do seznamu se přistupuje pomocí ukazovátka, které se pohybuje vpřed (a vzad), žádný index tu není k dispozici. Klíč prvku nemusí být definován, prvek seznamu většinou obsahovat přímo data (nebo ukazatel na ně). Pohyb ukazovátka vpřed a vzad určuje i předchůdce a následníka, seznam je lineární struktura – platí varianta d).

8.
Je dán datový typ oboustranná fronta se signaturou znázorněnou vpravo.:

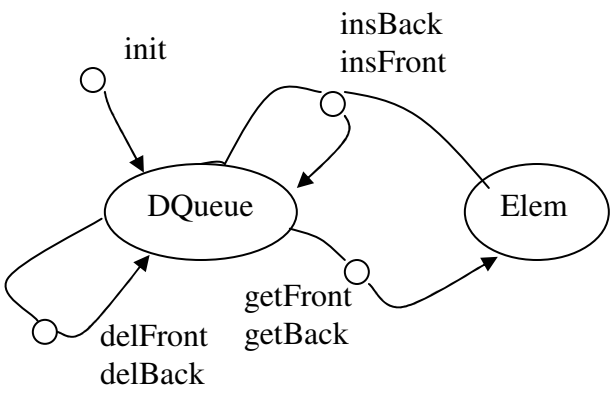


Které operace je možno použít k simulování fronty pomocí tohoto datového typu? Význam operací intuitivně odpovídá jejich názvu.

a) ins ~ insBack, del ~ delFront, get ~ getFront
b) ins ~ insFront, del ~ delBack, get ~ getFront
c) ins ~ insBack, del ~ delBack, get ~ getBack
d) ins ~ insFront, del ~ delFront, get ~ getBack

Máme simulovat frontu. Ve frontě se vkládá na konec, odebírá i čte ze začátku. Potřebujeme tedy operace InsBack, delFront a GetFront. To je právě varianta a).

9.
Je dán datový typ oboustranná fronta se signaturou znázorněnou vpravo:



Které operace je možno použít k simulování zásobníku pomocí tohoto datového typu? Význam operací intuitivně odpovídá jejich názvu.

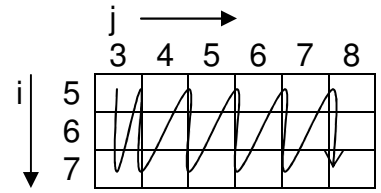
a) Push ~ insBack, pop ~ delFront, tos ~ getBack
b) Push ~ insFront, pop ~ delBack, tos ~ getBack
c) Push ~ insBack, pop ~ delBack, tos ~ getBack
d) Push ~ insFront, pop ~ delFront, tos ~ getBack

Máme simulovat zásobník. V zásobníku se vkládá na konec, odebírá i čte rovněž z konce. Potřebujeme tady operace InsBack, delBack a getBack. To je právě varianta c).

10.

Napište mapovací funkci pro pole o šesti sloupcích a třech řádcích. Adresa prvního prvku je uložena v proměnné *zacatek*. Pole je uloženo po sloupcích. Sloupcové indexy mají rozsah 3,4,5,6,7,8. Řádkové indexy nabývají hodnot 5,6,7

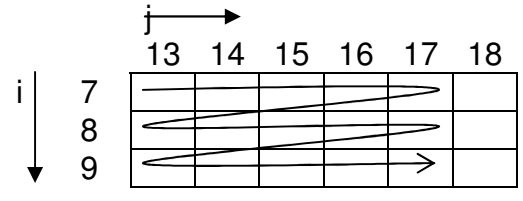
Řešení:



m(i,j) = *zacatek* + (i – 5) + (j – 3) * 3

11.

Napište mapovací funkci pro pole o šesti sloupcích a třech řádcích. Adresa prvního prvku je uložena v proměnné *zacatek*. Pole je uloženo po řádcích. Sloupcové indexy mají rozsah 13,14,15,16,17,18. Řádkové indexy nabývají hodnot 7,8,9




```
m(i,j) = zacatek +(i - 7)*6 +(j - 13)
```

12.
Je dán datový typ *Prvek* s konstantami a:->*Prvek*, b:->*Prvek*, ….z->*Prvek*. Dále je dán datový typ seznam, který je neformálně popsán následovně.

init: -> List	prázdný seznam
insert(_,_): Elem, List -> List	vložení prvku před ukazovátko. Ukazovátko zůstane ukazovat na prvek, na který ukazovalo před vložením
read(_): List -> Elem	přečtení prvku na který ukazuje ukazovátko
delete(_):List -> List	smazání prvku na místě ukazovátka, posun ukazovátka na následující prvek
first(_), last(_):List -> List	posun na první, za poslední prvek
next(_), prev(_):List -> List	posun na další, předcházející prvek
length(_): List -> Nat0	Vrátí délku (počet prvků) seznamu

Sestavte výraz, který popisuje seznam: a b c d, ukazovátko ukazuje na c

Řešení
Např: prev(prev(insert(d, insert(c, insert(b, insert(a,init))))))
Nebo: insert(b, insert(a, prev(prev(insert(d, insert(c, init))))))
V prázdném seznamu (výsledku operace init) ukazuje ukazovátko „za seznam“. Operace insert vkládá před ukazovátko. Proto musíme prvky vkládat pozpátku. Na závěr přesuneme ukazovátko na požadovaný prvek. Úloha má více řešení – v podstatě nekonečně mnoho, protože výraz může obsahovat libovolný počet párů operací insert a delete, které se vzájemně vyruší.

13.
Zadání jako v předchozí úloze, jen výraz má vypadat takto:

a b c d ukazovátko ukazuje na b

Řešení např.
prev(prev(prev(insert(d, insert(c, insert(b, insert(a,init)))))))

14.
Navrhněte implementaci fronty, která může obsahovat pouze celá čísla v rozsahu 0 až 999, přičemž se žádný prvek určitě nemůže ve frontě objevit více než jednou. Navržená implementace musí zajistit, aby následující operace měly konstantní složitost**:**

enqueue(e,q): prvek e vložen na konec fronty q
dequeue(e,q): prvek odstraněn z čela fronty q a zapsán do e
length(q): počet prvků fronty q
exists(e,q): testuje, zda je prvek e obsažen ve frontě q
move(e): přemístí všechny prvky ve frontě q o jednu pozici vpřed (první prvek se přitom
stane posledním)
Nejste omezeni rozsahem paměti a operace **init** může mít libovolnou nutnou složitost.

(V zadání je překlep v parametru operace move, místo move(e), má být ovšem move(q).)

Řešení
Klíčovým požadavkem pro řešení úlohy je zřejmě operace **exists(e,q)**, protože všechny ostatní zadané lze v běžné implementaci fronty polem nebo seznamem stihnout v konstantním čase. Přidáme proto ještě pole příznaků isInQ[ ] s tisci prvky, kde přítomnost/nepřítomnost značky (celočíselné, logické, znakové, bitové – na tom nesejde) na pozici k bude indikovat přítomnost/nepřítomnost čísla k ve frontě.

Operace **enqueue(e,q)** a **dequeue(e,q)** pak musí navíc provést potřebnou změnu prvku isInQ[e]. Operace **enqueue(e,q)** nemusí kontrolovat přeplnění fronty, pokud pro implementaci fronty vyhradíme dostatek prostoru (např. polem o 1000 prvcích).

V případě implementace polem je vhodné sáhnout po tzv. kruhové implementaci fronty. Představíme si to tak, že za prvkem s indexem 999 následuje bezprostředně prvek s indexem 0, před prvkem s indexem nula je hned prvek s indexem 999. Fyzicky ovšem takto paměť zacyklit nejde, musíme proto tuto myšlenku frontě implementačně vnutit. Zhruba to vypadá takto:

```
enqueue(e, q) {
    q.last++;
    if  (q.last == 1000) q.last = 0;     // takto prostor pro frontu “zacyklíme”
    q.pole[q.first] = e;
    q.pocetPrvku++;
    isInQ[e] = true;
}

int dequeue(q) {

    if (q.pocetPrvku == 0) ohlaš chybu, return;
    int  vratime = q.pole[q.first];
    q.first++;
    if  (q.first == 1000) q.first = 0;  // takto prostor pro frontu “zacyklíme”
    q.pocetPrvku--;
    isInQ[e] = false;
    return vratime;
}
```

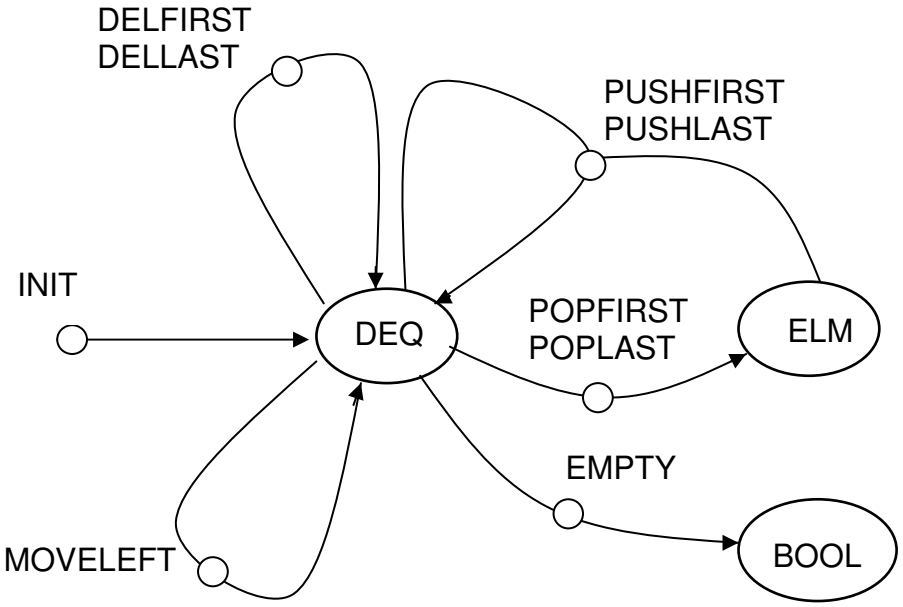
Stejný postup pro „zacyklení“ použijeme i v případě operace move(q) – jednoduché cvičení.

V případě implementace fronty spojovým seznamem zcela postačí standardní manipulace se seznamem doplněná o obsluhu pole isInQ.
Operace init(q) – ať už implicitně nebo explicitně -- musí alespoň naplnit pole isInQ hodnotami signalizujícími nepřítomnost prvku a bude tedy mít složitost $\Theta(n)$ (ve smyslu poznámky uvedené níže).
(Předpokládám, že si zejména Javisté uvědomují, že asymptotická složitost příkazu int [ ] arr = new int[n]; má složitost právě $\Theta(n)$.)

Doufám, že to takto stačí a že detaily si každý umí doprogramovat sám.

(Poznámka pro teoretiky: Protože maximální velikost fronty a její možné prvky jsou v úloze jasně dány, mohlo by -- se přísně vzato -- uvažovat také takto: Sekvenční vyhledávání v operaci exists(e,q) by zabralo O(1000 * složitost_testu_čísla), kde ovšem, počítáme-li s normálním strojem, má operace test_čísla konstantní složitost a tím pádem ovšem i výraz O(1000 * složitost_testu_čísla) představuje totéž co O(1), a tudíž má i sekvenční vyhledávání v daném poli de facto konstantní složitost (i když velkou). Implicitní smysl úlohy je ovšem tento: „Navrhněte implementaci fronty, která může obsahovat pouze celá čísla v rozsahu 0 až *n*–1,…přičemž všechny zadané operace mají konstantní složitost…". Úloha takto formulována není, ale v praxi se běžně formuluje s nějakými konstantami a přitom se předpokládá její implicitní smysl.)

15.
Navrhněte efektivní implementaci datového typu DEQUEUE popsaného následující signaturou:



Vaše implementace musí zajistit, aby každá z následujících operací (kromě operace INIT) proběhla v konstatntním čase.
DELFIRST = smaž první prvek
DELLAST = smaž poslední prvek
PUSHFIRST = vlož prvek na začátek
PUSHLAST = vlož prvek na konec
POPFIRST = vyjmi ("pop") prvek ze začátku fronty
POPLAST = vyjmi ("pop") prvek z konce fronty
EMPTY = test, zda je fronta prázdná
MOVELEFT = přemísti všechny prvky ve frontě o jednu pozici vpřed (první prvek se přitom stane posledním)
INIT = inicializuj tento datový typ (tato operace může mít libovolnou nezbytnou složitost)

Toto je patrně úloha řešitelná téměř mechanicky. Uvedený datový typ je v podstatě fronta, kde se navíc prvky smí odebírat z konce a přidávat na začátek. Přislušné operace jsou zcela analogické odebírání ze začátku a přidávání na konec. Operace

MOVELEFT ovšem nebude pohybovat všemi prvky ve frontě, odebere jen prvek z čela fronty a přidá ho na konec. Zda přitom využije ostatních operací, nebo zda si to vyřeší „interně“ (s nezbytným přesunem ukazatelů na konec a začátek), patrně již není významné. Operace INIT bude mít složitost větší než konstantní, použijeme-li implementaci polem. Proč?
Konkrétní implementaci jednotlivých operací neuvádím, snad to každý zvládne sám?

16.
Implementujte cyklickou frontu v poli pole[délka]. Dokud není pole zcela naplněno, musí být fronta stále funkční, tzn. musí být možné do ní stále přidávat i z ní ubírat. Narazí-li konec nebo začátek fronty na konec nebo začátek pole, neposunujte všechny prvky v poli, zvolte výhodnější „cyklickou“ strategii, která zachová konstantní složitost operace Vlož a Vyjmi.

Řešení
Myšlenka cyklické fronty je velmi jednoduchá. Připojíme první pozici pole ihned za poslední pozici a poslední pozici ihned před první. Tak se nám celé pole „zacyklí“ neboť při postupu stále doprava dojdeme nakonec na začátek pole. Nejsnáze se to implementuje pomocí indexů pole. Dejme tomu, že máme pole array[n].

```
int nextIndex(int i) {
   if (i < n-1) return (i+1);
   else return 0;
}

int prevIndex(int i) {
  if (i > 0) return (i-1);
  else return (n-1);
}
```

Budeme-li pro pohyb v poli užívat pouze uvedené funkce, nemůžeme se dostat mimo meze pole. Protože se snadno stane, že po určité době života fronty se i při neprázdné frontě octne konec fronty před jejím začátkem v poli (namaluje si příslušný obrázek!), bude vhodné pro kontrolu prázdnosti či přeplnění fronty registrovat zvlášť počet prvků ve frontě

Vložení prvku do fronty pak proběhne například takto:

```
boolean insert(int key) {
if (fronta.pocetPrvku == n) return false;
fronta.konec = nextIndex(fronta.konec);
array[fronta.konec] = key;
fronta.pocetPrvku++;
return true;
}
```

Obdobně bude implementována operace Delete, operace Init a Empty jsou již snad příliš jednoduché na to, aby tu musely být uvedeny.