Mulţimea (SET)

O **mulţime** ("set") este un container cu ajutorul căruia se poate reprezenta o colecţie finită de elemente distincte. Altfel spus, într-o mulţime elementele nu se pot repeta, există o singură instanţă a unui element. O altă caracteristică a mulţimii este faptul că într-o mulţime nu contează ordinea elementelor.

Mulţimea are toate operaţiile specifice **Colecţiei**, cu observaţia că operaţia adăugare într-o mulţime are specificaţie diferită faţă de operaţia de adăugare într-o colecţie (într-o mulţime elementele trebuie să fie distincte).

Tipul elementelor din mulţime, **TElement**, ca şi într-o colecţie, dealtfel, suportă cel puţin operaţiile de: atribuire (←) şi testarea egalităţii (=).

Spre exemplu, o mulţime de numere întregi ar putea fi: ***m***={1, 2, 3, 5, 4}.

Caracterul finit al unei mulţimi ne permite (totuşi) indexarea elementelor sale, ceea ce face ca la nivelul reprezentării interne, mulţimea **M** să poată fi asimilată cu un vector m_1, m_2,..., m_n (chiar dacă ordinea elementelor dintr-o mulţime nu este esenţială).

Pentru a putea preciza modul în care se vor efectua operaţiile pe mulţimi, vom defini structura de ***submulţime*** . Aceasta se poate realiza cu ajutorul unui vector format din valorile funcţiei caracteristice asociate submulţimii.

Dacă **M** este o mulţime, atunci submulţimea $\mathbf{S}\subseteq\mathbf{M}$ va avea asociat vectorul $\mathbf{V}_S=(s_1, s_2,..., s_n)$ unde

$$s_i = \begin{cases} 1, & daca\ m_i \in S \\ 0, & daca\ m_i \notin S \end{cases}$$

Operaţiile pe submulţimi pot fi acum definite prin intremediul operaţiilor pe vectorii caracteristici asociaţi.

Fie $\mathbf{S1, S2}\subseteq\mathbf{M}$ două submulţimi ale mulţimii **M**. Atunci:

a) **S1 ∪ S2 (reuniunea celor două submulţimi)** va fi caracterizată de vectorul **V** obţinut din $\mathbf{V}_{S1}$ şi $\mathbf{V}_{S2}$ efectuând operaţia logică "∨" (sau) element cu element

∨	0	1
0	0	1
1	1	1

b) **S1 ∩ S2 (intersecţia celor două submulţimi)** va fi caracterizată de vectorul **V** obţinut din $\mathbf{V}_{S1}$ şi $\mathbf{V}_{S2}$ efectuând operaţia logică "∧" (sau) element cu element

∧	0	1
0	0	0
1	0	1

Ca urmare, orice alte operaţii cu submulţimile unei mulţimi pot fi imaginate ca operaţii logice asupra vectorilor ataşaţi.

În continuare, vom prezenta specificaţia Tipul Abstract de Date **Mulţime.**

domeniu

$\mathcal{M}$={**m | m** este o mulţime cu elemente de tip **TElement**}

operaţii (**interfaţa** TAD-ului Mulţime)

creează(m)

pre: -

post:m$\in\mathcal{M}$, m este mulţimea vidă (fără elemente)

adaugă(m, e)

pre: m$\in\mathcal{M}$, e$\in$**TElement**

post: m'$\in\mathcal{M}$, m'=m$\cup${e}

{e se "reuneşte" la mulţime, adică se va adăuga numai dacă e nu mai apare în mulţime}

şterge(m, e)

pre: m$\in\mathcal{M}$, e$\in$**TElement**

post: m'$\in\mathcal{M}$, m'=m-{e}

{se şterge e din m}

caută(m, e)

pre: m$\in\mathcal{M}$, e$\in$**TElement**

post: ***cauta*=** adevărat dacă e$\in$m
fals în caz contrar

dim(m)

pre: m$\in\mathcal{M}$

post: ***dim*=** dimensiunea mulţimii m (numărul de elemente) $\in\mathcal{N}$

vidă(m)

pre: m$\in\mathcal{M}$

post: ***vida*=** adevărat în cazul în care m e mulţimea vidă
fals în caz contrar

iterator(m, i)

pre: m$\in\mathcal{M}$

post: i$\in\mathcal{I}$, i este un iterator pe mulţimea m

distruge(m)

pre: $m \in \mathcal{M}$

post: mulţimea m a fost 'distrusă' (spaţiul de memorie alocat a fost eliberat)

Accesarea elementelor mulţimii se va face în aceeaşi manieră ca la colecţie, folosind iteratorul pe care-l oferă mulţimea.

Modalităţi de implementare ale mulţimilor:

- tablouri (dinamice);
- vectori booleeni (de biţi);
- liste înlănţuite;
- tabele de dispersie;
- arbori binari.

TEMA. Implementaţi operaţiile specifice TAD Mulţime folosind SD menţionate. Studiaţi complexitatea operaţiilor în funcţie de SD aleasă pentru implementare.

Dicţionar (MAP)

Observaţii

1. Elementele din dicţionar sunt perechi de forma **(cheie, valoare)**. Dicţionarele păstrează elemente în aşa fel încât ele să poată fi uşor localizate folosind **chei**.
2. Spre exemplu, un dicţionar poate păstra conturi bancare: fiecare cont este un obiect identificat printr-un număr de cont (considerat **cheia** elementului) şi informaţii adiţionale (numele şi adresa deţinătorului contului, informaţii despre depozite, etc). Informaţiile adiţionale vor fi considerate ca fiind **valoarea** elementului.
3. Implementarea unui dicţionar (SD aleasă pentru implementare) trebuie să ofere un mecanism eficient de regăsire a valorilor pe baza cheilor.
4. Într-un dicţionar cheile sunt **unice**.
5. În general, o **cheie** are o unică **valoare** asociată. Dacă o cheie poate avea mai multe valori asociate => Multi-dicționar (**MultiMap**)

Dăm în continuare specificația Tipului Abstract de Date **Dicționar**.

domeniu

$\mathcal{D}$={**d | d** este un dicţionar cu elemente **e = (c, v), c** de tip **TCheie, v** de tip **TValoare**}

operaţii (**interfaţa** TAD-ului Dicţionar)

creează(d)
pre: true
post:d∈$\mathcal{D}$, d este dicţionarul vid (fără elemente)

adaugă(d, c, v)
pre: d∈$\mathcal{D}$, c∈**TCheie,** v∈**TValoare,**
post: d'∈$\mathcal{D}$, d'=d+(c, v) (se adaugă în dicţionar perechea (c, v))

caută(d, c, v)
pre: d∈$\mathcal{D}$, c∈**TCheie**
post: ***caută*****=** adevărat dacă (c,v)∈d, caz în care v∈**TValoare** e valoarea asociată cheii c
fals în caz contrar, caz în care v=$0_{TValoare}$

şterge(d, c, v)
pre: d∈$\mathcal{D}$, c∈**TCheie**
post: v∈**TValoare**
perechea (c, v) este ştearsă din dicţionar, dacă c∈ d
v=$0_{TValoare}$ în caz contrar

dim(d)

pre: d∈$\mathcal{D}$

post: ***dim***= dimensiunea dicţionarului d (numărul de elemente) ∈$\mathcal{N}^*$

vid(d)

pre: d∈$\mathcal{D}$

post: ***vid***= adevărat în cazul în care d e dicţionarul vid
fals în caz contrar

chei(d, m)

pre: d∈$\mathcal{D}$

post: m∈$\mathcal{M}$, m este mulţimea cheilor din dicţionarul d

valori(d, c)

pre: d∈$\mathcal{D}$

post: c∈$\mathcal{C}ol$, c este colecţia valorilor din dicţionarul d

perechi(d, m)

pre: d∈$\mathcal{D}$

post: m∈$\mathcal{M}$, m este mulţimea perechilor (cheie, valoare) din dicţionarul d

iterator(d, i)

{se creează un iterator pe dicţionarul d}

pre: d∈ $\mathcal{D}$

post:i∈ $\mathcal{I}$, i este iterator pe dicţionarul d

distruge(d)

pre: d∈ $\mathcal{D}$

post: dicţionarul d a fost 'distrus' (spaţiul de memorie alocat a fost eliberat)

Modalităţi de implementare ale dicționarelor:

- tablouri (dinamice);
- liste înlănţuite;
- tabele de dispersie;
- arbori binari.

Observaţii

1. Multi-dicționar (**MultiMap**)
   - **TValoare**=**TLista** (o cheie are o listă de valori asociate **TElement**)
   - Operație din interfața TAD Dicționar a cărei specificație se modifică
     - ***şterge***(d, c, v)

pre: d∈$\mathcal{D}$, c∈**Tcheie,** v∈**TElement**

post: d'∈$\mathcal{D}$

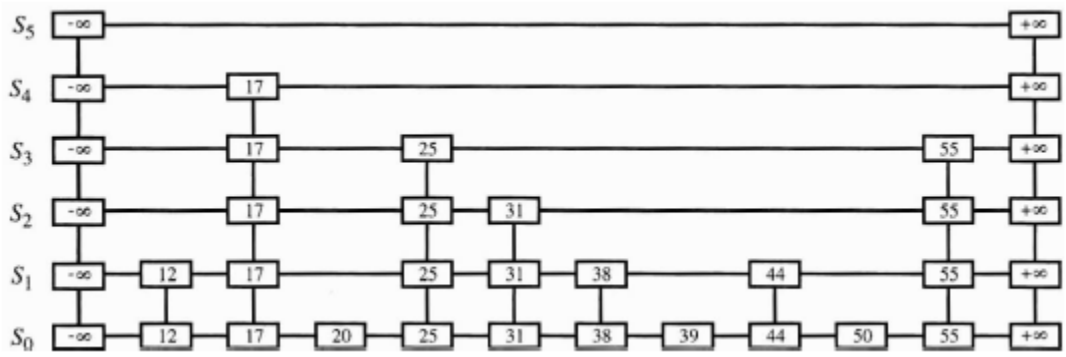
perechea (c, v) este ștearsă din dicţionar, dacă c∈ d

2. Dicționar ordonat/sortat (**SortedMap**)
   - **TCheie=TComparabil**
   - Este definită o relație de ordine între chei $\Re \subseteq TCheie \times TCheie$
   - Nu se modifică interfața
   - **Cerință** – operațiile **iterator** și **perechi** returnează elementele în ordine în raport cu relația $\Re$

3. Multi-dicționar ordonat/sortat (**Sorted MultiMap**)

4. O structură randomizată de stocare a datelor, eficientă pentru memorarea unui dicționar ordonat este **Skip List**

   Ex: cheile 20, 17, 50, 44, 55, 12, 44, 31, 39,25



   - Operațiile specifice (adăugare, căutare, modificare) necesită O($\log_2 n$) cu o probabilitate mare (în medie) - O(n) caz defavorabil, dar puțin probabil să apară
   - În Java există implementarea *ConcurrentSkipListMap*
   - Intrările din S_{i+1} sunt alese aleator din intrările din S_i, alegând ca fiecare intrare din S_i să fie și în S_{i+1} cu probabilitatea 0.5.
   - O poziție are 4 legături (*următor, precedent, sus, jos*)
   - Căutare
     - Cu succes **39**: -∞, 17, 17, 25, 25, 31, 31, 38, 38, **39**
     - Fără succes **41**: -∞, 17, 17, 25, 25, 31, 31, 38, 38, **39**