Seminar 7 - Ansamblu

1) Se dă un **TAD Coadă cu Priorități Bidirecțională**, pe care se pot efectua următoarele operații

- creează(cpb): creează cpb de tip **Coadă cu Priorități Bidirecțională** vidă
- adaugă(cpb, e): adaugă elementul e în cpb
- caută_min(cpb): returnează elementul minim din cpb
- caută_max(cpb): returnează elementul maxim din cpb
- șterge_min(cpb): șterge elementul minim din cpb
- șterge_max(cpb): șterge elementul maxim din cpb

și având următoarele complexități:

creează	Theta(1)
adaugă	$O(\log_2 n)$
caută_min	Theta(1)
caută_max	Theta(1)
șterge_min	$O(\log_2 n)$
sterge_max	$O(\log_2 n)$

Descrieți reprezentarea TAD și implementarea operațiilor creează, adaugă, caută_min și șterge_min.

REZOLVARE

<u>Metoda 1</u>

- vom folosi două ansambluri: unul minimal și altul maximal, pentru a putea identifica atât minimul cât și maximul în timp constant și pentru a obține complexitatea O(log n) la adaugare

- în operația de adaugare vom insera elementul în ambele ansambluri
- operația sterge_min presupune ștergerea din ansamblul minimal (conform algoritmului de ștergere obisnuit (a se vedea Cursul 14), avand complexitate logaritmica), urmata de stergere elementului minim din ansamblul maximal. Observăm că minimul va fi o frunză în ansamblul maximal, dar poziția lui nu poate fi calculată în timp constant (sau subliniar). Astfel, pentru a

obtine complexitate logaritmica pentru aceasta operatie, vom memora, pentru fiecare element din ansamblul minimal, poziția pe care acesta se afla in ansamblul maximal.

- în mod similar, vom avea nevoie pentru operația sterge_max de pozițiile din ansamblul minimal
- aceste poziții vor fi actualizate la adăugarea și ștergerea unui element

Exemplu

Fie $\mathbf{A_{min}}$ ansamblul minimal utilizat în reprezentarea TAD Coadă cu Priorități Bidirecțională, și $\mathbf{A_{max}}$ ansamblul maximal, precum și Pos_{A_max} și Pos_{A_min} definite după cum urmează:

	1	2	3	4	5	6	7	8	9	10
A_{min}										
Pos_{A_max}										

Pos_{A_max}[i] = poziția elementului A_{min}[i] în A_{max}

	1	2	3	4	5	6	7	8	9	10
A_{max}										
Pos_{A_min}										

Pos_{A_min}[i] = poziția elementului A_{max}[i] în A_{min}

- Vom modifica subalgoritmul **urcă(a, i)** (prezentat in Cursul 14 - Ansamblu) astfel încât acesta:
  - va opera modificări (interschimbări) atât în cadrul ansamblului a, cât și în cadrul tabloului cu indici (poziții) aferenți
  - va returna poziția pe care a „ajuns” elementul prin interschimbări (poziția sa finală în ansamblul (vectorul) a)

Adăugăm 10

- În ambele ansambluri: nu există elemente, adăugăm pe poziția 0 și nu se efectuează niciun pas în cadrul algoritmului „urcă”

	1	2	3	4	5	6	7	8	8	10
A_{min}	10									
Pos_{A_max}	1									

	1	2	3	4	5	6	7	8	9	10
A_{max}	10									
Pos_{A_min}	1									

- Se returnează pozițiile 1 în urma apelurilor subalgoritmului `urcă`, și se setează aferent în vectorii Pos_{A_max} și Pos_{A_min}

Adăugăm 18

- În A_min: se adaugă noul element la finalul ansamblului, pe poziția 2 (conform subalgoritmului **adaugă**)

	1	2	3	4	5	6	7	8	9	10
A_{min}	10	18								
Pos_{A_max}	1									

- În cadrul adăugării, nu se efectuează niciun pas în subalgoritmul **urcă** (18>10), ansamblul A_{min} după adăugarea elementului 18 va fi:

	1	2	3	4	5	6	7	8	9	10
A_{min}	10	18								
Pos_{A_max}	1									

- Se returnează poziția pe care a fost adăugat 18 în A_{min}: poziția 2

- În A_max: se adaugă noul element la finalul ansamblului, pe poziția 2 (conform subalgoritmului **adaugă**)

	1	2	3	4	5	6	7	8	9	10
A_{max}	10	18								
Pos_{A_min}	1									

- în algoritmul „urcă”, elementul 18 este interschimbat cu elementul 10 (18>10); odată cu mutarea elementului 10 pe poziția 2, se mută și indicele elementului 10 în ansamblul A_min pe noua poziție (se efectuează modificări în Pos_{A_min}

	1	2	3	4	5	6	7	8	9	10
A_{max}	18	10								
Pos_{A_min}		1								

- În același timp, poziția memorată în Pos_{A_max} pentru elementul care a fost deplasat (poziția elementului 10 în A_{max}) este actualizată

	1	2	3	4	5	6	7	8	9	10
A_{min}	10	18								
Pos_{A_max}	2									

- Din subalgoritmul „urcă” apelat pentru restabilirea proprietății de ansamblu pentru A_max, se returnează poziția pe care a fost adăugat 18 în A_{max}: poziția 1

- Se actualizează pozițiile pe care a fost adăugat noul element în Pos_{A_max} și Pos_{A_min}

	1	2	3	4	5	6	7	8	9	10
A_{min}	10	18								
Pos_{A_max}	2	1								

	1	2	3	4	5	6	7	8	9	10
A_{max}	18	10								
Pos_{A_min}	2	1								

Adăugăm 15

- În A_min:
  - se adaugă noul element la finalul ansamblului (vectorului), pe poziția 3;
  - nu se efectuează niciun pas în cadrul algoritmului „urcă” (10 < 15);
  - se returnează poziția 3, poziția pe care a fost adăugat noul element în A_min

	1	2	3	4	5	6	7	8	9	10
A_{min}	10	18	15							
Pos_{A_max}	2	1								

- În A_max
  - se adaugă noul element la finalul ansamblului (vectorului) pe poziția 3;
  - nu se efectuează niciun pas în cadrul algoritmului „urcă” (18 > 15);
  - se returnează poziția 3, poziția pe care a fost adăugat noul element în A_max

	1	2	3	4	5	6	7	8	9	10
A_{max}	18	10	15							
Pos_{A_min}	2	1								

- Nu se efectuează pași pentru niciun ansamblu în cadrul algoritmului „urcă” -> pozițiile returnate din apelurile subalgoritmului „urcă” vor fi 3, indicii din Pos_{A_max} și Pos_{A_min} sunt setate aferent

	1	2	3	4	5	6	7	8	9	10
A_{min}	10	18	15							
Pos_{A_max}	2	1	3							

	1	2	3	4	5	6	7	8	9	10
A_{max}	18	10	15							
Pos_{A_min}	2	1	3							

Adăugăm 13

- În A_{min}:
  - se adaugă noul element pe poziția 4;

	1	2	3	4	5	6	7	8	9	10
A_{min}	10	18	15	13						
Pos_{A_max}	2	1	3							

  - în algoritmul „urcă”, elementul 13 este interschimbat cu părintele său, elementul 18 (13<18); se mută și valorea aferentă elementului 18 în Pos_{A_max}

	1	2	3	4	5	6	7	8	9	10
A_{min}	10	13	15	18						
Pos_{A_max}	2		3	1						

- În același timp, poziția memorată în Pos_{A_min} (poziția elementului 18 în A_{min}) este actualizată

	1	2	3	4	5	6	7	8	9	10
A_{max}	18	10	15							
Pos_{A_min}	4	1	3							

- Se returnează poziția pe care a fost adăugat 13 în A_{min}: poziția 2

- În A_{max}:
  - se adaugă noul element la finalul ansamblului (vectorului), pe poziția 4;

	1	2	3	4	5	6	7	8	9	10
A_{max}	18	10	15	13						
Pos_{A_min}	4	1	3							

  - în algoritmul „urcă”, elementul 13 este interschimbat cu părintele său, elementul 10 (13>10); se mută și valorea aferentă (poziția din A_min a elementului 10) din Pos_{A_min}

	1	2	3	4	5	6	7	8	9	10
A_{max}	18	13	15	10						
Pos_{A_min}	4		3	1						

- În același timp, poziția memorată în Pos_{A_max} (poziția elementului 10 în A_{max}) este actualizată

	1	2	3	4	5	6	7	8	9	10
A_{min}	10	13	15	18						
Pos_{A_max}	4		3	1						

- Se returnează poziția pe care a fost adăugat 13 în A_{max}: poziția 2

- Se actualizează pozițiile pentru elementul nou adăugat în Pos_{A_max} și Pos_{A_min}

	1	2	3	4	5	6	7	8	9	10
A_{min}	10	13	15	18						
Pos_{A_max}	4	2	3	1						

	1	2	3	4	5	6	7	8	9	10
A_{max}	18	13	15	10						
Pos_{A_min}	4	2	3	1						

Adăugăm 7

- În A_{min}:
  - se adaugă noul element pe poziția 5;

	1	2	3	4	5	6	7	8	9	10
A_{min}	10	13	15	18	7					
Pos_{A_max}	4	2	3	1						

  - în algoritmul „urcă”, elementul 13, părintele elementului 7, este coborât (7<13); se mută și valoarea aferentă elementului 13 din Pos_{A_max}

	1	2	3	4	5	6	7	8	9	10
A_{min}	10	7	15	18	13					
Pos_{A_max}	4		3	1	2					

- În același timp, poziția memorată în Pos_{A_min} (poziția elementului 13 în A_{min}) este actualizată

	1	2	3	4	5	6	7	8	9	10
A_{max}	18	13	15	10						
Pos_{A_min}	4	5	3	1						

  - În continuare, în algoritmul „urcă”, „urcarea” elementului 7 continuă, întrucât părintele său, elementul 10, este mai mare decât acesta (7<10), și este nevoie să fie coborât; se mută și valoarea aferentă din Pos_{A_max}

	1	2	3	4	5	6	7	8	9	10
A_{min}	7	10	15	18	13					
Pos_{A_max}		4	3	1	2					

- În același timp, poziția memorată în Pos_{A_min} (poziția elementului 10 în A_{min}) este actualizată

	1	2	3	4	5	6	7	8	9	10
A_{max}	18	13	15	10						
Pos_{A_min}	4	5	3	2						

- Se returnează poziția pe care a fost adăugat 7 în A_{min}: poziția 1

- În A_{max}:
  - se adaugă noul element la finalul vectorului, pe poziția 5;

	1	2	3	4	5	6	7	8	9	10
A_{max}	18	13	15	10	7					
Pos_{A_min}	4	5	3	2						

  - Nu se efectuează niciun pas în subalgoritmul „urcă”; proprietatea de ansamblu maximal este respectată
  - Se returnează poziția pe care a fost adăugat 7 în A_{max}: poziția 5

- Se actualizează pozițiile pentru elementul nou adăugat în Pos_{A_max} și Pos_{A_min}

	1	2	3	4	5	6	7	8	9	10
A_{min}	7	10	15	18	13					
Pos_{A_max}	5	4	3	1	2					

	1	2	3	4	5	6	7	8	9	10
A_{max}	18	13	15	10	7					
Pos_{A_min}	4	5	3	2	1					

—----------

Ștergem minimul (valoarea 7)

- mutăm ultimul element din A_min pe prima poziție, reținând poziția în A_max a elementului șters (în tabloul care reprezintă ansamblul A_max, elementul 7 este pe poziția 4)

	1	2	3	4	5	6	7	8	9	10
A_{min}	13	10	15	18	~~13~~					
Pos_{A_max}	2	4	3	1	~~1~~					

- actualizăm pozitia in Pos_{A_min} a elementului care a fost mutat (elementul 13)

	1	2	3	4	5	6	7	8	9	10
A_{max}	18	13	15	10	7					
Pos_{A_min}	4	1	3	2	1					

- continuăm procesul de ștergere în A_min, interschimbând 13 cu 10 (13 > 10, 13 < 15), precum și indicii care indică pozițiile lor în ansamblul A_max

	1	2	3	4	5	6	7	8	9	10
A_{min}	10	13	15	18						
Pos_{A_max}	4	2	3	1						

- Pozițiile celor două elemente (10 și 13) în Pos_{A_min} sunt actualizate

	1	2	3	4	5	6	7	8	9	10
A_{max}	18	13	15	10	7					
Pos_{A_min}	4	2	3	1	1					

- Ne folosim de poziția reținută înainte de a șterge elementul 7 din A_min, și anume 4
- Ștergem 7 din A_max: minimul este o frunză in A_max; mutăm ultimul element pe poziția lui și apelăm funcția urca pentru a restabili proprietatea de ansamblu; în acest caz, nu se mai efectuează modificări (elementul 7 este pe poziția corectă)

—--

Exemplu 2 stergere - sterge max (20) - in A_max stergere “normala”, in A_min trage pe poz 4 pe 16, 16 trebuie interschimbat cu 17 pentru a se respecta proprietatea

	1	2	3	4	5	6	7	8	9	10
A_{min}	10	17	15	18	20	16				
Pos_{A_max}	4	5	6	2	1	3				

- Pozițiile celor două elemente (10 și 13) în Pos_{A_min} sunt actualizate

	1	2	3	4	5	6	7	8	9	10
A_{max}	20	18	16	10	17	15				
Pos_{A_min}	5	4	6	1	2	3				

PSEUDOCOD

Reprezentare:

a_min: TElement[]
a_max: TElement[]
pos_a_min: Intreg[]
pos_a_max: Intreg[]
n: Intreg
capacitate: Intreg

Implementare:

```
subalgoritm creeaza(cpb) este:
  cpb.capacitate ← 10
  cpb.n ← 0
  cpb.a_min ← @alocă vector de dimensiune cpb.capacitate
  cpb.a_max ← @alocă vector de dimensiune cpb.capacitate
  cpb.pos_a_min ← @alocă vector de dimensiune cpb.capacitate
  cpb.pos_a_max ← @alocă vector de dimensiune cpb.capacitate
sf-subalgoritm


subalgoritm cauta_min(cpb) este:
  cauta_min ← cpb.a_min[1]  // indexarea incepe de la 1
sf-subalgoritm
```

În continuare, prezentăm implementarea adăugării în Coada cu Priorități Bidirecțională.

```
funcție urca(a, curent, pos_in_current_heap, pos_in_other_heap, rel) este:
  //pos_in_other_heap reprezintă vectorul care reține pozițiile elementelor
ansamblului a în celălalt ansamblu (dacă a este a_min, pos_in_other_heap este
pos_a_max, și dacă a este a_max, pos_in_other_heap este pos_a_min)
  //pos_in_current_heap reprezintă vectorul care reține pozițiile elementelor din
celălalt ansamblu în ansamblul a (dacă a este a_min, pos_in_current_heap este
pos_a_min, și dacă a este a_max, pos_in_other_heap este pos_a_max)

  parinte ← [curent / 2]
  elem ← a[curent]
  cattimp parinte >= 1 si not rel(a[parinte], elem) executa:
    a[curent] ← a[parinte]
    pos_in_other_heap[curent] ← pos_in_other_heap[parinte]  // pos_in_other_heap
si a sunt "sincronizate"
    pos_in_current_heap[pos_in_other_heap[parinte]] ← curent //actualizam
informația din vectorul care reține poziția părintelui în ansamblul a; dat fiindcă
pos_in_other_heap[i] ne poate oferi poziția elementului a[i] in celalalt ansamblu,
```

```
pozitia de modificat se afla pe indexul dat de pos_in_other_heap[parinte] si noua
lui pozitie in ansamblul curent este “curent”

    curent ← parinte
    parinte ← [parinte / 2]
  sf-cattimp

  a[curent] ← elem
  urca ← curent   // returnam pozitia finala a elementului in ansamblul curent
pentru a putea actualiza vectorul de pozitii sincronizat cu celalalt ansamblu
ulterior
sf-funcție


subalgoritm adauga(cpb, e) este:
   daca a.n = a.capacitate atunci:
     @redimensionare
   sf-daca

   cpb.n ← cpb.n+1
   cpb.a_min[cpb.n] ← e
   cpb.a_max[cpb.n] ← e

   // “urcam” elementul in a_min pentru a restabili proprietatea de ansamblu si
returnam pozitia lui
   pos_e_in_a_min ← urca(cpb.a_min, cpb.n, cpb.pos_a_min, cpb.pos_a_max, “≤”)

   // “urcam” elementul in a_max pentru a restabili proprietatea de ansamblu si
returnam pozitia lui
   pos_e_in_a_max ← urca(cpb.a_max, cpb.n, cpb.pos_a_max, cpb.pos_a_min, “≥”)

   cpb.pos_a_min[pos_e_in_a_max] ← pos_e_in_a_min
   cpb.pos_a_max[pos_e_in_a_min] ← pos_e_in_a_max

sf-subalgoritm
```

<u>Metoda 2:</u>

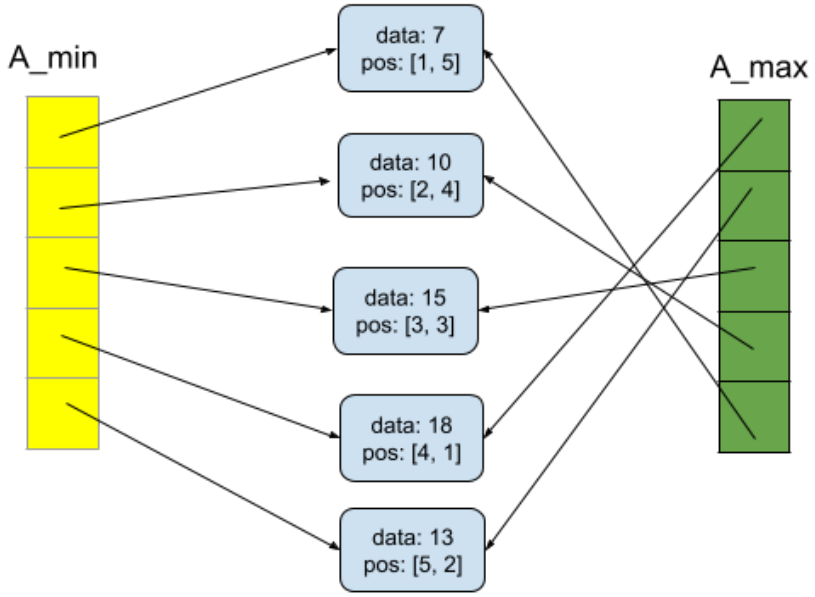
- această metodă folosește aceeași idee de rezolvare ca metoda 1, dar o altă reprezentare a datelor în memorie
- în această reprezentare, nu memorăm datele în interiorul ansamblului (vectorului), ci în exterior, în niște noduri alocate dinamic, iar ansamblurile vor conține pe fiecare poziție un pointer către un astfel de nod. Astfel, valorile din Coada cu Priorități vor fi memorate o singură dată (și nu de două ori, ca în abordarea precedentă). De asemenea, atunci când urcăm sau coborâm o valoare în ansamblu, mutăm doar pointerii (nu valorile efective).
- pentru a putea implementa eficient operația de ștergere, nodurile vor conține și pozițiile valorilor respective în ansamblul minimal și maximal

După adăugarea elementelor 10, 18, 15, 13 și 7 vom avea următoarele:

- valoarea 7 va fi situată pe poziția 1 în A_min și pe poziția 5 in A_max. În nodul corespunzător, vom memora valoarea 7 și aceste două poziții: 1 și 5. Pentru a putea parametriza operația

“urcă”, vom memora pozițiile sub forma unui vector *pos* care va conține la indexul 0 poziția valorii în A_min, iar la indexul 1 poziția valorii în A_max (în cazul valorii 7 vom avea vectorul [1, 5]).

- valoarea 10 va fi situată pe poziția 2 în A_min și pe poziția 4 in A_max. În nodul corespunzător, vom memora valoarea 10 și un vector conținând aceste două poziții: [2, 4] ș.a.m.d



Reprezentare:

Nod:
data: TElement
pos: Intreg[2] // vector conținând pozitiile elementuui in cele doua ansambluri: pe indicele 0 memoram pozitia din a_min, iar pe indicele 1, pozitia din a_max

CoadaCuPrioritatiBidirectionala:
a_min: (↑Nod)[]
a_max: (↑Nod)[]
n: Intreg
capacitate: Intreg

Implementare:

```
subalgoritm adauga(cpb, e) este:
    daca a.n = a.capacitate atunci:
      @redimensionare
    sf-daca

    cpb.n ← cpb.n+1

    alocă(nod)
```

```
    [nod].data ← e
    // adăugăm nodul pe poziția n în cele două ansambluri
    a_min[cpb.n] ← nod
    a_max[cpb.n] ← nod
    // setăm pozițiile în cele două ansambluri ale nodului nod
    [nod].pos[0] ← cpb.n
    [nod].pos[1] ← cpb.n

    // "urcam" elementul in a_min pentru a restabili proprietatea de ansamblu
    urca(cpb.a_min, cpb.n, 0, "≤")

    // "urcam" elementul in a_max pentru a restabili proprietatea de ansamblu
    urca(cpb.a_max, cpb.n, 1, "≥")

sf-subalgoritm


subalgoritm urca(a, curent, a_index, rel) este:
  //a_index este 0 daca subalgoritmul "urcă" este apelat pe a_min - în acest caz,
trebuie să actualizăm pozițiile [a[i]].pos[0]; Dacă metoda "urcă" este apelata pe
a_max, vom actualiza pozițiile [a[i]].pos[1]

  parinte ← [curent / 2]
  elem ← a[curent]
  cattimp parinte >= 1 si not rel([a[parinte]].data, [elem].data) executa:
    a[curent] ← a[parinte]
    //a[parinte] a fost mutat pe pozitia curent => actualizam pozitia lui in a
    [a[parinte]].pos[a_index] ← curent

    curent ← parinte
    parinte ← [parinte / 2]
  sf-cattimp

  a[curent] ← elem
  [a[curent]].pos[a_index] ← curent   //actualizam pozitia lui elem in a

 sf-subalgoritm



funcție sterge_min(cpb) este:
  nod_min ←a_min[1]
  val_min ←[nod_min].data
  pos_min_in_max_heap ← [nod_min].pos[1]

  // mutăm ultimul element din a_min pe prima poziție
  a_min[1] ← a_min[cpb.n]
  [a_min[cpb.n]].pos[0] ← 1
  coboară(a_min, 1, 0, "≤", cpb.n)

  // mutam in a_max ultimul element pe pozitia elementului pe care il stergem
  a_max[pos_min_in_max_heap] ← a_max[cpb.n]
  [a_max[cpb.n]].pos[1] ← pos_min_in_max_heap
  // urcam acest element pentru a restabili proprietatea de ansamblu
  urcă(a_max, pos_min_in_max_heap, 1, "≥")
```

```
  cpb.n ←cpb.n-1
  dealocă(nod_min)
  sterge_min ← val_min
sf-funcție


subalgoritm coboară(a, curent, a_index, rel, n) este:
  elem ← a[curent]
  câttimp curent < n execută
    descendent ← -1
    dacă 2*curent <= n atunci
      descendent ← 2*curent
    sf-daca
    dacă 2*curent+1 <= n si not rel([a[2*curent+1]].data, [a[2*curent]].data)
atunci
      descendent ← 2*curent+1
    sf-daca
    daca descendent != -1 si not rel([elem].data, a[descendent].data) atunci
      tmp ← a[current]
      a[current] ← a[descendent]
      a[descendent]←tmp
      [a[descendent]].pos[a_index] ← descendent
      [a[curent]].pos[a_index] ← curent

      curent ← descendent

      altfel
       @return
      sf-daca
   sf-cattimp
sf-subalgoritm
```

Metoda 3:

- folosim un arbore AVL pentru memorarea informației din container
- vom utiliza și două variabile care vor conține adresa minimului si maximului
- aceste variabile vor fi actualizate la adăugări și ștergeri