ANSAMBLU (HEAP)

- Este o structură de date eficientă pentru memorarea *cozilor cu priorități*
- Tipuri de ansamblu: **binar**, binomial, Fibonacci, etc.
- Structura de *ansamblu* (*heap*) binar este un vector care poate fi vizualizat sub forma unui arbore binar aproape plin (având structura de ansamblu).

Observații: Pp. că elementele din ansamblu sunt $a_1, a_2, ..., a_n$

- a_1 este elementul din rădăcină
- a_i are fiul stâng $a_{2 \cdot i}$ dacă $2 \cdot i \le n$ și fiul drept $a_{2 \cdot i+1}$ dacă $2 \cdot i + 1 \le n$
- a_i are părintele $a_{[i/2]}$

Un ***ansamblu binar*** $(a_1, a_2, ..., a_n)$ este un arbore binar care are *structură de heap* și verifică *proprietatea de heap.*

- **Structură de heap** – arborele binar este plin, exceptând ultimul nivel care este plin de la stânga la dreapta (în ordine).
- **Proprietatea de heap**
  - $a_i \ge a_{2i} \quad \forall i$, dacă $2 \cdot i \le n$
  - $a_i \ge a_{2i+1} \quad \forall i$ dacă $2 \cdot i + 1 \le n$

Observații

- Relația „≥” poate fi generalizată la o relație de ordine $\Re$ oarecare.
- Ansambul binar este în general memorat ***secvențial*** folosind un vector (dinamic), fără a fi necesară memorarea înlănțuită - legături între elemente (ex. pointeri).

Proprietăți

- a_1 este cel mai **mare** element din ansamblu dacă $\Re$=”≥”
- Dacă $\Re$=”≥”, atunci pe orice drum de la rădăcină la un nod, elementele sunt ordonate descrescător.
- Înălțimea unui heap cu n elemente este $\theta(\log_2 n)$. Ca urmare, timpul de execuție a operațiilor specifice va fi $O(\log_2 n)$

- Operații specifice pe ansamblu:
  - **adăugare** element (astfel încât să se păstreze proprietatea de heap)
  - **ștergere** element (se șterge elementul maxim dacă $\mathcal{R}$=”≥”, cel din vârful ansamblului).
- Pp. în continuare $\mathcal{R}$=”≥”.
- Reprezentarea ansamblului

Ansamblu
Max: Intreg {capacitatea maxima de memorare}
n: Intreg {nr.de elemente din ansamblu}
e: TElement[0..n] {elementele din ansamblu}

Subalgoritmul ADAUGĂ (*a*, *e*) este {complexitate timp $O(\log_2 n)$ }
{pre: *a*: Ansamblu, *a* nu e plin, *e*:TElement }
{post: *a* rămâne ansamblu după adăugare}
a.n←*a*.n+1
a.e[*a*.n] ←*e*
URCĂ(*a*, *a*.n) {restabilește proprietatea de ansamblu posibil alterată}
sfADAUGĂ

<u>Obs.</u> Nu verificăm la adăugare dacă ansamblul e plin. La implementare se poate redimensiona vectorul dacă se observă că se depășește capacitatea maximă alocată.

Subalgoritmul URCĂ (*a*, *i*) este { complexitate timp $O(\log_2 n)$ }
{urcă elementul de pe poziția *i* spre rădăcină până va fi satisfăcută proprietatea de ansamblu}
{pre: *a* ansamblu nevid, elem. de pe poziția *i* a fost actualizat}
{post: *a* este ansamblu}
e←*a*.e[*i*] {elementul de urcat}
k ←*i* {poziția unde va fi pus elementul *e* }
p ← [*k*/2] {părintele lui *k*}
{căutăm o poziție pentru *e* printre strămoșii lui}
Câttimp (*p* ≥1) și (*a*.e[*p*]<*e*) execută
a.e[*k*] ←*a*.e[*p*] {strămoșii mai mici decât *e* sunt coborâți}
k ←*p*
p ← [*p*/2]
sfCâtTimp
{s-a gasit pozitia *k* pe care poate fi adăugat *e*}
a.e[*k*] ←*e*
sfURCĂ

Subalgoritmul ȘTERGE (*a*, *e*) este {complexitate timp $O(\log_2 n)$ }
{pre: *a*:Ansamblu, *a* nu e vid}
{post: *e*:TElement este elementul maxim și e șters, *a* rămâne ansamblu după ștergere}
e← *a*.e[1] {elementul maxim}
a.e[1] ←*a*.e[*a*.n]
a.n←*a*.n-1
COBOARĂ(*a*, 1) {restabilește proprietatea de ansamblu posibil alterată}
sfSTERGE

Subalgoritmul COBOARĂ (a, poz) este {complexitate timp $O(\log_2 n)$ }
{coboară elementul de pe poziția *poz* printre descendenți până va fi satisfăcută proprietatea de ansamblu}
{pre: *a* ansamblu nevid, elem. de pe poziția *poz* a fost actualizat}
{post: *a* este ansamblu}

```
    e←a.e[poz] {elementul de mutat}
    i ←poz{poziția unde va fi pus elementul e }
    j ←2 · poz {fiul stâng al lui i}
    {căutăm o poziție pentru e printre descendenți. Descendenții mai mari decât e urcă un nivel în arbore }
    câttimp ( j ≤a.n ) execută { i are fiu stâng}
        dacă ( j < a.n ) atunci { i are și fiu drept? Dacă da, îl luăm pe cel mai mare dintre ei}
            dacă a.e[j] < a.e[j+1] atunci
                j ←j+1
            sfdacă
        sfdacă
        dacă a.e[j] ≤ e atunci {cel mai mare fiu este mai mic decât e atunci STOP}
            j ←a.n+1
        altfel
            a.e[i] ←a.e[j] {fiul j urcă}
            i ←j
            j ←2 · i
        Sfdacă
    Sfcâttimp
    a.e[i] ←e {pun elementul înapoi în structură}
```

sfCOBOARĂ

Aplicație : HEAPSORT. Sortarea unui vector cu n elemente folosind un Heap. Complexitate timp $O(n \log_2 n)$ spațiu suplimentar de memorare $O(n)$.

PROBLEME

1. Generalizați relația"≥" la o relație de ordine $\mathcal{R}$ oarecare și implementați operațiile specifice.
2. Care este cel mai mic, respectiv cel mai mare număr de elemente dintr-un heap având înălțimea *h*?
3. Arătați că un heap având *n* elemente are înălțimea $[\log_2 n]$
4. Arătați că în orice subarbore al unui heap rădăcina subarborelui conține cea mai mare valoare care aparține în acel arbore (dacă $\mathcal{R}$="≥").
5. Dacă $\mathcal{R}$="≥", unde se poate afla cel mai mic element al unui heap, presupunând că toate elementele sunt distincte?
6. Este vectorul în care elementele se succed în ordine descrescătoare un heap?
7. Este secvența <23, 17, 14, 6, 13, 10, 15, 7, 12> un heap?
8. Găsiți un algoritm $O(n \cdot \log_2 k)$ pentru a interclasa *k* liste ordonate, unde *n* este numărul total de elemente din listele de intrare. Indicație: se va folosi un *heap*