VII. Tipul Abstract de Date Coadă Prioritară.

Pentru a compara elementele unei colecții se utilizează o parte din fiecare element numită *cheie*. Cheile sunt comparabile, folosind o *regulă de comparație* de tipul ≤. Aceasta determină o *relație de ordine totală*, caracterizată prin:

- reflexivitate: $k \leq k$
- antisimetrie: $k_1 \leq k_2 \cap k_2 \leq k_1 \Rightarrow k_1 = k_2$
- tranzitivitate: $k_1 \leq k_2 \cap k_2 \leq k_3 \Rightarrow k_1 \leq k_3$

O coadă prioritară este o colecție în care fiecare element are asociată o cheie (sau prioritate) care determină ordinea în care se plasează elementele în colecție în cursul operației de inserare. Cel mai prioritar element va fi plasat în vârful cozii şi va fi primul element extras din colecție.
Un element al cozii prioritare va fi constituit dintr-o pereche (cheie, valoare).
Operațiile principale specifice cozii prioritare sunt:

- inserarea unei perechi **(k, v)** în coadă într-o poziție corespunzătoare priorității lui
- ştergerea elementului cu cheie minimă (cel mai prioritar) din coada prioritară

Cozile prioritare sunt utilizate în sistemele de operare la planificarea proceselor, a operațiilor de intrare/ieşire, în problema selecției, în simularea evenimentelor, etc.

Specificare TAD Coadă Prioritară

- Domenii: **PQ** = Coadă Prioritară **Elem (int x val) =** tip elemente coadă prioritară
int = prioritate **val =** tip valoare asociată

- Funcții:

- creare coadă prioritară vidă	**new : → PQ**
- inspectarea celui mai prioritar element	**min : PQ → Elem**
- ştergerea celui mai prioritar element	**delmin : PQ → Elem**
- inserarea unui element în coada prioritară	**insert : PQ × int × val → PQ**
- modificarea cheii unui element	**chpr : PQ × Elem × int → PQ**
- test coadă prioritară vidă	**empty: PQ → int**

- Axiome:

1. **PQ new()**
2. **min(insert(new(), x))=x**
3. **delmin(insert(new(), x))=new()**
4. **min(insert(insert(PQ,y),x))=if pri(x) < pri(min(insert(PQ,y)))**
   x
   else
   min(insert(PQ,y))
5. **delmin(insert(insert(PQ,y),x))=if pri(x)<pri(min(insert(PQ,y)))**
   insert(PQ,y)
   else
   insert(delmin(insert(PQ,y),x)

O coadă prioritară poate fi implementată *naiv* folosind:
a) o *listă nesortată* – inserarea se face întotdeauna la începutul listei în **O(1)**, dar ştergerea presupune în prealabil găsirea elementului cel mai prioritar în **O(n)**.
b) o *listă sortată* – inserarea se face în **O(n)**, păstrând relația de ordine dintre elemente, iar ştergerea se face în **O(1)** de la un capăt al listei

c) un *arbore binar de căutare* – inserarea şi ştergerea se fac în **O(log n)**, dar arborele trebuie frecvent reechilibrat, deoarece ştergerea se face mereu din acelaşi loc.

O implementare eficientă a cozii prioritare cu heapuri binare asigură complexitatea **O(log n)** atât pentru inserare, cât şi pentru ştergere.
Numeroase aplicații impun determinarea eficientă a poziției fiecărui element din coada prioritară. În acest scop vom asocia fiecărui element un *locator* – o valoare întreagă, reprezentând poziția pe care o ocupă elementul în coada prioritară. Orice deplasare a elementului în coada prioritară va conduce la modificarea locatorului.

Interfață TAD Coadă Prioritară

```
PQ   PQ_New();                    //creere coada prioritara vida
int  PQ_Empty (PQ h);             //test coada prioritara vida
int  PQ_Full (PQ h);              //test coada prioritara plina
Elem PQ_Min(PQ h);                //preia elementul cu cheia minima
void PQ_DelMin(PQ h);             //sterge elementul cu cheia minima
int PQ_Insert (PQ h, int k, void *v); //inserează (k,v) in c.p.
int PQ_ChPr(PQ h,int p,int k); //schimbă prioritatea elementului
                               // din poziția p la valoarea k
int PQ_Size(PQ h);                //nr.elemente coada prioritara
```

Exemple.
Sortare folosind o coadă prioritară.

- elementele listei nesortate se scot din listă şi se introduc într-o coadă prioritară.
- se scot elementele din coada prioritară şi se inserează în lista sortată

```
PQ_SORT(L)
     Intrare:  L-lista cu elemente nesortate
     Ieşire:   L-lista cu elemente sortate

     PQ = new()
     while  ! empty(L)
        e = remove(L, begin(L))
        insert(PQ, e, e)
     while  ! empty(PQ)
        e = delmin(PQ)
        insert(L, end(L))
```

Sortarea prin selecție este o variantă a sortării cu coadă prioritară implementată cu listă nesortată. Algoritmul de sortare necesită `n` ştergeri cu complexitate: $1+2+\ldots+n = O(n^2)$ şi `n` inserări cu complexitate $n*O(1)=O(n)$.

Sortarea prin inserție este tot o variantă a sortării cu coadă prioritară, implementată cu listă sortată. În acest caz operațiile de inserare au complexitate $1+2+\ldots+n = O(n^2)$ iar ştergerile, $n*O(1)=O(n)$.

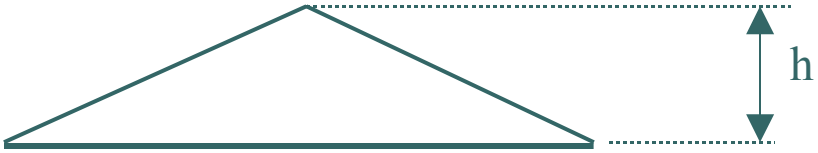
O implementare mai eficientă a cozii prioritare ar conduce la performanțe mai bune ale algoritmului de sortare. Implementarea cu heapuri a cozii prioritare (cu complexitate $O(\log n)$, atât pentru inserare cât şi pentru ştergere), conduce la cunoscuta metodă de sortare cu heapuri – heapsort.

Heapuri. (arbori parțial ordonați)

Definiții şi terminologie.

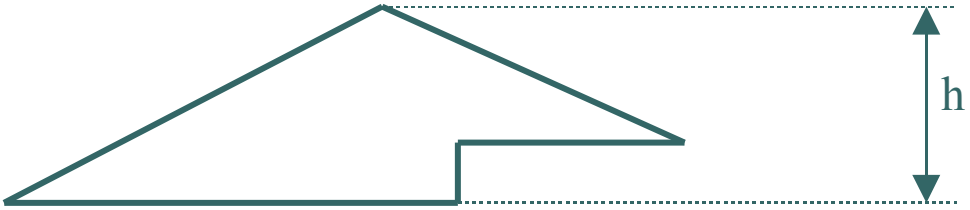
Un *arbore binar perfect* (sau *plin*) are pe fiecare nivel **h** un număr maxim de noduri 2^h. Numărul total de noduri **n**, corespunzând unei înălțimi **h** a arborelui perfect este:
$n=1+2+...+2^h=2^{h+1}-1$

Dintre toți arborii cu **n** noduri, arborele binar plin are înălțimea minimă:
$h=\log_2(n+1)-1=\lfloor \log_2 n \rfloor$



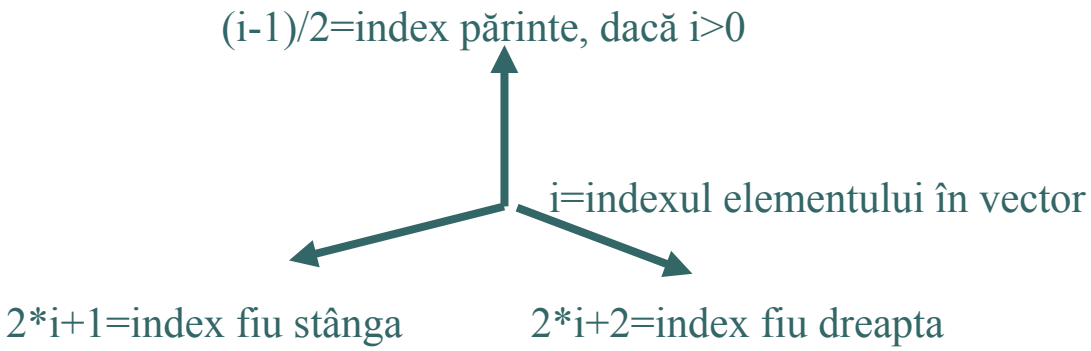
Într-un *arbore binar complet*:

0. fiecare nivel **i**, exceptând ultimul are 2^i noduri
1. nodurile de pe ultimul nivel sunt aliniate la stânga



Numărul de noduri, pentru un arbore binar complet de înălțime **h** este: $2^h \leq n \leq 2^{h+1}-1$ de unde
$h=\lfloor \log_2 n \rfloor$

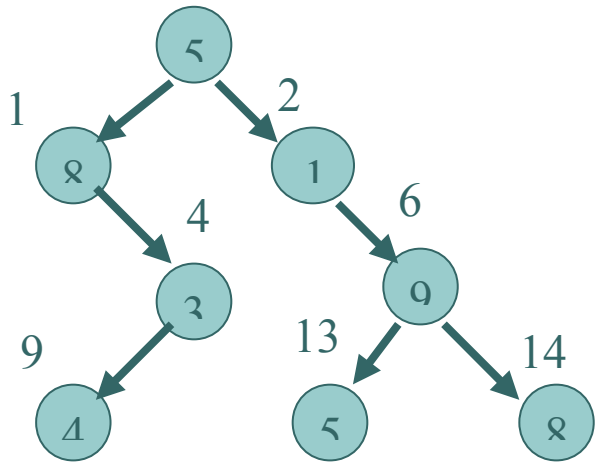
Un arbore binar complet poate fi reprezentat în mod eficient printr-un vector (fără a folosi pointeri), în care elementele vectorului sunt cheile din noduri, iar indicii elementelor corespund numerotării nodurilor pe niveluri (în lărgime). În acest caz, legăturile unui nod la succesori şi predecesor se stabilesc astfel:



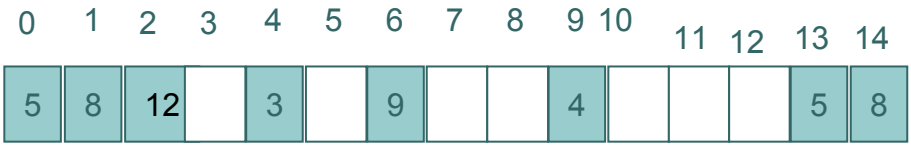
Aceste relații corespund indexării în vector de la 0;dacă prima poziție nu este folosită (rădăcina este indexată cu 1), relațiile sunt:

```
tata(i)=i/2,
fiu_stânga(i)=2*i,
fiu_dreapta(i)=2*i+1
```

Este posibil să reprezentăm printr-un vector şi un arbore binar care nu este complet. Pentru a stabili în vector relațiile de tip tată – fiu între elemente, acestea vor trebui plasate în pozițiile pe care le-ar ocupa în arborele binar complet, adică vectorul va prezenta goluri (poziții libere) corespunzătoare nodurilor lipsă din arborele binar complet. Astfel arborelui binar:



Îi corespunde vectorul:



Dacă vectorul are **`n`** elemente, succesorul stâng al elementului cu indexul **`i`** există dacă **`2*i+1<n`** (de asemenea tatăl nu este definit dacă **`i=0`**).

Un *heap* sau *arbore parțial ordonat* (sau *movilă*)

0. are o *proprietate de structură*, fiind un arbore binar complet
1. are o *proprietate de ordine* între cheia nodului părinte şi cheile nodurilor fii.

Menționăm că între cheile fiilor unui nod (frați) nu există nici o relație de ordine.
Într-un *heap maxim*

- cheia dintr-un nod este mai mare decât cheile din nodurile fii.
- cheia din nodul rădăcină este cheia maximă din heap.
- Orice cale de la rădăcină la o frunză este o secvență descrescătoare de chei.

In mod asemănător se defineşte un *heap minim*.

O definire recursivă a unui heap maxim este : un arbore binar complet care

2. fie este vid
3. fie are cheia din rădăcină mai mare decât cheile din succesorii direcți ai rădăcinii şi
4. subarborii oricărui nod sunt heapuri

Intr-un heap:

- operațiile de *inserare* şi de *ştergere* se realizează foarte eficient (în O(log n)).
- o cale de la rădăcină la o frunză este o secvență ordonată de chei.

Într-un heap minim (maxim), un nod care nu respectă relația de ordine a heapului:

5. *migrează în sus* (filtrare sus, sift-up, bubble-up) dacă este mai mic/mare decât predecesorul. Deplasarea se opreşte în momentul în care nodul ajunge pe un nivel la care relația de ordine a heapului este respectată. Nodul poate migra până în rădăcină (nivel 0)

6. *migrează în jos* (filtrare jos, sift-down, bubble-down) dacă este mai mare/mic decât unul dintre succesorii direcți. Deplasarea se opreşte în momentul în care nodul ajunge pe un nivel la care relația de ordine a heapului este respectată. Nodul poate migra până într-o frunză.

Funcția **`FiltruJos()`** va fi definită cu 4 parametri: heapul **`h`**, poziția inițială în heap a elementului care coboară **`p`**, numărul de elemente din heap **`n`** (nu am folosit valoarea **`h->n`**, deoarece în cazul sortării prin metoda heapurilor, dimensiunea heapului se schimbă) şi funcția de comparație a priorităților.

În interfață se declară o structură nespecificată şi un pointer opac la ea:

```
struct heap;
typedef struct heap *PQ;
```

Un heap va fi definit în secțiunea de implementare ca o structură ce conține:

- numărul de elemente **`n`**,
- dimensiunea maximă până la care se poate extinde heapul **`MAX`**
- un vector **`a`** cu elemente de orice tip .

```
struct heap{
  int n;
  int MAX;
  void **a;
};
```

```
void FiltruJos(PQ h, int p, int n, PFC comp){
  int fiu;
  void *t = h->a[p];
  for( ; 2*p+1 < n; p=fiu){
    fiu = 2*p+1;
    if(fiu+1 < n && comp(h->a[fiu+1], h->a[fiu]) < 0)
      fiu++;
    if(comp(t, h->a[fiu]) <= 0)
      break;
    else
      h->a[p] = h->a[fiu];
  };
  h->a[p] = t;
}
```

```
void FiltruSus(PQ h, int i, PFC comp){
  void *x = h->a[i];
  for( ; i>0 && comp(x, h->a[(i-1)/2])<0; i=(i-1)/2])
    h->a[i]=h->a[(i-1)/2];
  h->a[i] = x;
}
```

Transformarea unui tablou într-un heap.

Prin migrarea unui singur element (în jos sau în sus), arborele binar complet nu se transformă în heap.

Transformarea în heap a unui arbore binar complet se face prin (eventuala) migrare în jos a tuturor rădăcinilor de heapuri. Nodurile frunze îndeplinesc relația de ordine a heapului, deci nu se mai verifică.

Primul nod verificat va fi părintele ultimului nod (**n-1**), adică cel cu indexul **(n-2)/2** iar ultimul va fi rădăcina.

```
void Crheap(PQ h, PFC comp){
  int crt;
  for(crt = h->((n-2)/2); crt>=0; crt--)
    FiltruJos(h, crt, h->n, comp);
}
```

Complexitatea filtrării (în sus sau în jos) este dată de lungimea traseului elementului până la fixarea lui pe un nivel.Aceasta nu poate depăşi înălțimea arborelui deci **O(h)=O(log n)**
Transformarea unui vector într-un heap presupune eventuala migrare a tuturor nodurilor interne. Numărul acestora este **n/2** deci complexitatea este **O(n.log n)**.
O estimare mai exactă observă că într-un heap există cel mult $\lceil n/2^{h+1} \rceil$ noduri de înălțime **h**.
Deci:

$$\sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot O(h) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right) = O(n) \text{ deoarece}$$

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}, \qquad |x| < 1$$

$$\sum_{i=0}^{\infty} i x^{i-1} = \frac{1}{(1-x)^2}$$

$$\sum_{i=1}^{\infty} i x^{i} = \frac{x}{(1-x)^2}$$

$$\sum_{i=1}^{\infty} \frac{i}{2^i} = \frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2} = 2$$

Sortare prin metoda heapurilor

Reprezintă o metodă foarte eficientă de sortare cu complexitatea $O(n.\log_2 n)$.
Pentru a folosi TAD heap binar, definit mai sus, vom plasa tabloul de sortat în heap, iar după sortare vom extrage tabloul din heap.

7. Vectorul de sortat este transformat în prealabil într-un heap. Vom considera vectorul separat în două părți, una nesortată, constituită inițial din tot vectorul şi una deja sortată, inițial vidă.

8. Elementul maxim se află în rădăcină, de unde este trecut în zona sortată, prin schimb cu ultimul element din heap.

Heapul, deci zona nesortată se restrânge cu un element, în timp ce zona sortată se extinde. Prin plasarea ultimului element în rădăcină, relația de ordine a heapului este încălcată, şi vom restabili heapul prin filtrarea în jos a elementului din rădăcină.

```
void heapsort(PQ h, PFC comp){
  void *el;
  int i;
  Crheap(h, comp);
  for(int i=h->n-1; i>0; i--){
    swap(h->a[0],h->a[i]);
    FiltruJos(h, 0, i, comp);
  }
}
```

Un arbore parțial ordonat are înălțimea $\mathbf{k=log_2 n}$. La transformarea tabloului în heap sunt necesare **n/2** filtrări jos, fiecare necesitând cel mult **k** comparații. În ciclu se mai fac **n-1** filtrări jos, deci:

$$N = k*n/2 + k*(n-1) = k*(3*n/2-1) = O(kn) = O(n.\log_2 n)$$

Heapurile se implementează mult mai uşor decât arborii (care necersită şi echilibrare). Heapurile nu necesită spațiu pentru pointeri.

Metoda de sortare dezvoltată este foarte eficientă.

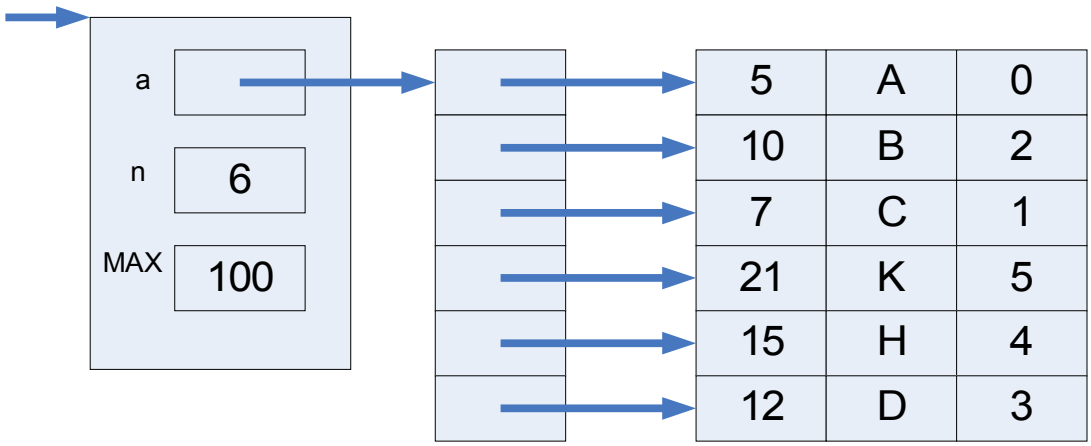
Implementare TAD Coadă Prioritară cu heapuri binare.

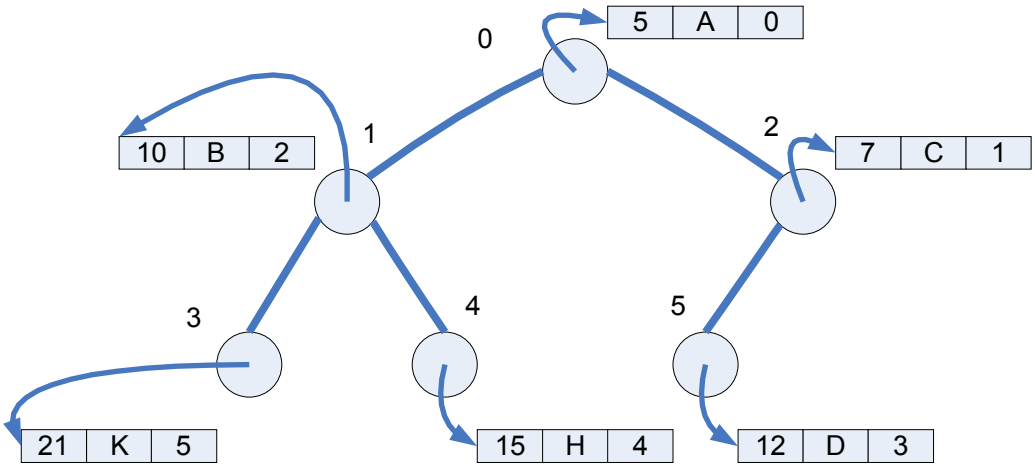
Un element al cozii prioritare va fi un triplet (cheie, informație, locator), cheia (sau prioritatea) şi poziția în coada prioritară fiind întregi, iar informația asociată cheii, putând fi de orice tip.

```
typedef struct elem{
   int  key;
   void *val;
   int  loc;
} *Elem;


PQ PQ_New (){
  PQ h=(PQ)malloc(sizeof(struct heap));
  h->n=0;
  h->MAX=100;
  h->a=(Elem*)malloc(h->MAX*sizeof(Elem));
  return h;
}

int PQ_Size (PQ h){return h->n;}
int PQ_Empty (PQ h){return h->n==0;}
int PQ_Full (PQ h){ return h->n==h->MAX;}
Elem PQ_Min(PQ h){return h->a[0];}
```





In operațiile de filtrare se fac următoarele modificări:

- se actualizează locatorii elementelor deplasate
- dispare funcția de comparație este înlocuită cu operatorul de comparație
- funcțiile întorc poziția finală pe care o ocupă în coada prioritară elementul deplasat

```
int FiltruJos(PQ h, int p, int n){
  int fiu;
  Elem t = h->a[p];
  for( ; 2*p+1 < n; p=fiu){
    fiu = 2*p+1;
    if(fiu+1 < n && h->a[fiu+1]->key < h->a[fiu]->key)
      fiu++;
    if(t->key <= h->a[fiu]->key) break;
    h->a[fiu]->loc--;
    t->loc++;
    h->a[p] = h->a[fiu];
  };
  h->a[p] = t;
  return t->loc;
}
```

```
int FiltruSus(PQ h, int p){
  Elem t = h->a[p];
  int tata;
  while(p > 0){
     tata = (p-1)/2;
    if(t->key >= h->a[tata]->key) break;
     h->a[tata]->loc++;
     t->loc--;
    h->a[p]=h->a[tata];
  }
  h->a[p] = t;
  return t->loc;
}
```

La ştergerea elementului din vârful heapului în locul primului element (rădăcină) se pune ultimul element din heap şi se scade numărul de elemente. Plasarea ultimului element în vârful heapului poate strica relația de ordine a heapului. Pentru a o restabili, se interschimbă elementul din vârf cu cel mai mic dintre fii. Procesul de coborâre în heap poate continua până când se ajunge într-o poziție în care proprietatea de ordine a heapului este respectată (sau până se ajunge într-o frunză).

```
void PQ_DelMin(PQ h){
  assert(!PQ_Empty (h));
  h->a[0] = h->a[h->n - 1];
  h->n--;
  FiltruJos(h, 0, h->n);
}
```

Pentru a insera un element în heap, acesta se adaugă la sfârşitul heapului şi se creşte numărul de elemente. Dacă ordinea în heap este respectată (elementul adăugat nu este mai mic decât părintele) operația se încheie. În caz contrar elementul introdus comută cu părintele, ridicându-se în heap. Drumul său poate continua spre rădăcină, cât timp ordinea nu este respectată.

```
int PQ_Insert (PQ h, int k, void *v){
  assert(!H_Full (h));
  Elem e = (Elem)malloc(sizeof struct elem);
  e->key = k;
  e->val = v;
  e->loc = h->n;
  int n=++h->n;
  h->a[h->n-1]=e;
  return FiltruSus(h, n);
}
```

În operațiile cu grafuri (algoritmul Dijkstra) prioritatea unui element se poate modifica, ceea ce conduce la reaşezarea lui în coada prioritară.

Dacă implementăm coada prioritară printr-un heap minim, creşterea priorității elementului din poziția **p** la valoarea **k** determină ridicarea în heap a elementului, deci o filtrare în sus, în timp ce o scădere a priorității determină o coborâre în heap, deci o filtrare în jos. Funcția întoarce noua poziție pe care o ocupă elementul deplasat în coada prioritară.

```
int PQ_ChPri(PQ h, int p, int k){
  ELEM temp=h->a[p];
```

```
  h->a[p]->key = k;
  if(k < temp->key)
    return FiltruSus(h, p);
  else
    return FiltruJos(h, p, h->n);
}
```

Aplicații ale cozilor prioritare.

Selecția celui de-al p-lea element.

Soluția 1

9. se crează un heap minim cu cele n elemente (în **O(n)**)
10. se extrag **p** elemente (în **O(p.log n)**)

Soluția 2

11. se crează un heap minim cu primele **p** elemente
12. pentru celelalte elemente

dacă **elem>min** se scoate minimul şi se pune **elem** în heap în **O(n.log p)** dar foloseşte un heap cu numai **p** elemente.

Probleme propuse.

Dându-se un tablou de n întregi distincți, scrieți o funcție cu complexitatea **O(n+p log n)**, care determină cel de-al **p**-lea cel mai mare întreg din tablou.

Transformați într-un heap maxim tabloul

O grupă conține n studenți, precizați prin nume şi calificativ. Primilor √n studenți li s-a dat nota 10. Stabiliți care sunt aceştia. Folosiți un algoritm cu complexitate mai mică decât O(n logn) adică nu se admite soluția sortare descrescătoare după note şi selectarea primilor √n

Scrieți un algoritm care stabileşte în O(n log n), dacă într-un vector cu n componente întregi, există 2 întregi x şi y astfel ca x+y=z, în care z este dat.
Fie H un heap maxim, cu n elemente.Scrieți un algoritm având complexitatea mai mică decât O(n), care modifică valoarea unui element de la v1 la v2.

2 puncte

Transformați secvența: 25, 40, 5, 72, 36, 80, 15, 50, 100, 10, 60, 20.

a) într-un heap maxim
b) într-un heap minm

Transformați într-un heap minim şi într-un heap maxim secvența:
48, 15, 80, 32, 100, 4, 60, 12, 50, 40.

Transformați într-un heap minim şi într-un heap maxim secvența:
35, 29, 10, 50, 25, 37, 8, 100, 20, 30, 1, 15