Coadă cu priorităţi

Am studiat următoarele TAD :

❖Stiva: Principiu de căutare LIFO;

❖Coada: Principiu de căutare FIFO;

❖Tabela de simboluri (o coadă generalizată în care înregistrările au chei): Principiu de căutare : găseşte înreistrarea a cărei cheie este egală cu o cheie dată, dacă există.

Observație: Fiecare TAD dă naştere unui număr de TAD-uri înrudite, dar diferite, care rezultă ca un produs al examinării atente a programelor client și a performanței la implementări

Observație:
Pentru multe aplicații, elementele abstracte pe care le prelucrează sunt unice, o calitate care ne determină să luăm în considerare și modificarea ideii noastre despre modul în care stive, cozi FIFO și alte TAD-uri generalizate ar trebui să funcționeze. În mod concret, trebuie luat în considerare efectul de a schimba specificațiile la stive, cozi FIFO și cozi generalizate pentru a interzice obiecte duplicat în structura de date.

*Exemple:*O companie care menține o listă de adrese de clienți ar putea dori să încerce să crească lista prin efectuarea operațiunilor de inserare din alte liste adunate din mai multe surse, dar nu ar vrea ca lista sa să creasca pentru o operație de inserare, care se referă la un client deja aflat pe listă. Același principiu se aplică într-o varietate de aplicații. Pentru un alt exemplu, luăm în considerare problema de rutare a un mesaj printr-o rețea complexă de comunicații. Am putea încerca să trecem simultan prin mai multe căi în rețea, dar există doar un singur mesaj, astfel încât orice nod din rețea ar dori/trebui să aibă un singur exemplar din mesaj în structurile sale interne de date.

O abordare pentru rezolvarea acestei situații este de a lăsa la latitudinea clienților sarcina de a se asigura că elementele duplicat nu sunt prezente în TAD, o sarcină pe care clienții probabil ar putea-o efectua cu ajutorul unui TAD diferit. Dar, din moment ce scopul unei TAD este de a oferi clientilor soluții "curate" la problemele de aplicații, am putea decide că detectarea și rezolvarea duplicatelor este o parte a problemei pe care TAD ar trebui să o rezolve.
Politica de a interzice elemente cu dubluri este o schimbare în abstractizare: interfața, numele operațiilor și așa mai departe pentru un astfel de TAD sunt aceleași cu cele pentru TAD-ul corespunzător fără restricţii, dar comportamentul implementării se schimbă în mod fundamental. În general, ori de câte ori vom modifica specificațiile al unui TAD, vom obține un TAD complet nou - unul care are proprietăți complet diferite.

Fiecare TAD dă naştere unui număr de TAD-uri înrudite, dar diferite, care rezultă ca un produs al examinării atente a programelor client și a performanței la implementări

Vom considera acum un alt caz de coadă: Coada cu priorităţi.

MULTE APLICATII cer ca să prelucram înregistrări cu cheile în ordine, dar *nu neapărat în ordine complet sortată* și nu neapărat toate o dată. De multe ori, vom colecta un set de înregistrări, apoi prelucrăm înregistrarea cu cea mai mare cheie, apoi poate colectăm mai multe înregistrări, apoi prelucrăm cea cu cheia de curentă cea mai mare și așa mai departe. O structură de date adecvată într-un astfel de situaţie suportă operațiunile de: *introducerea* unui nou element și *ștergerea celui mai mare* element. O astfel de structură de date se numește o coadă cu priorităţi. Folosirea cozilor cu priorităţi este similară cu utilizarea cozilor FIFO (șterge cea mai veche) și stivelor LIFO (șterge cele mai noi), dar punerea lor în aplicare în mod eficient este mai dificilă. Coada cu priorităţi este cel mai important exemplu de TAD *coada generalizată*. De fapt, coada cu priorităţi este o generalizare bună a stivei și cozii, pentru că putem implementa aceste structuri de date cu cozile cu priorităţi, folosind atribuiri adecvate de priorităţi.

Definiție: O coadă cu priorităţi este o structură de date de înregistrări cu chei care acceptă două operații de bază: introduce un nou articol și șterge elementul cu cea mai mare cheie.

Unul dintre principalele motive pentru care multe implementări de cozi cu priorități sunt atât utile, este flexibilitatea în a permite programelor de aplicații client de a efectua o varietate de diferite operații pe seturi de înregistrări cu chei.

Vrem să construim și să actualizăm o SD care conține înregistrări cu chei numerice (priorități) care suportă unele dintre următoarele operațiuni:

Construct: Construirea unei cozi cu priorităţi din N articole date;

Insert: Inserarea unui nou element;

Remove=Delete the maximum: Ștergerea elementului maxim;

Change the priority: Schimbarea priorității unui element specificat arbitrar;

Delete: Ștergerea unui element specificat arbitrar;

Join două cozi de prioritate într-una singură.

Observații:

1. În cazul în care înregistrările pot avea chei duplicate, vom lua drept "*maxim*" „*orice înregistrare cu cea mai mare valoare de cheie*“.
2. Ca și în multe alte structuri de date, avem, de asemenea, nevoie să adăugăm la acest set operațiile standard de *inițializare*, de *testare ifempty* și, probabil, *destroy* și *copy*
3. Există o suprapunere între aceste operații, iar uneori este mai eficient să se definească alte operații similare, de exemplu, anumiți clienți poate doresc în mod frecvent să găsească elementul maxim din coada cu priorităţi, fără însă a-l şterge sau, am putea avea o operație de înlocuire a elementului maxim cu un nou element care se poate efectua ca: *Remove + Insert* sau *Insert+ Remove*, dar în mod normal, vom obține cod mai eficient , prin implementarea unor astfel de operații în mod direct, cu condiția ca acestea să fie necesare și precis specificate ! (*Remove+Insert≠Insert+ Remove*).
4. Pentru unele aplicații, ar putea fi ceva mai convenabil de a comuta şi a lucra cu elementul *minim*, mai degrabă decât cu cel maxim. Noi rămânem în primul rând, la cozile cu priorităţi care sunt orientate spre accesarea elementului cu cheia maximă.

Coada cu priorităţi este un *prototip* de tip abstract de date (TAD) (vezi cursul 3): Reprezintă un set bine definit de operațiuni asupra datelor, și oferă o abstracție convenabilă care permite să se separe programele de aplicații (clienți) de diversele implementari pe care le vom lua în considerare în acest curs.

```
Această interfață definește operațiile pentru cel mai simplu tip de coadă cu
priorităţi : inițializare, testare dacă este vidă, inserare un element nou,
stergere cel mai mare element. Implementari elementare ale acestor funcții,
utilizând array-uri și liste inlăntuite pot necesita în cel mai defavorabil
caz, timp liniar, dar vom vedea implementări în acest curs în care toate
operațiunile sunt garantate pentru a rula în timp proporțional cu logaritmul
numărului de elemente din  coada cu priorităţi. Argumentul lui PQinit specifică
numărul maxim de elemente acceptate în coada cu priorităţi.

void PQinit(int);
 int PQempty();
void PQinsert(Item);
Item PQdelmax() ;
```

Implementări elementare

Implementări ale cozilor cu priorităţi folosind:

- O listă nesortată (ordonată) în raport cu cheile elementelor;
  - reprezentare secvenţială pe tablou (vector dinamic) .
  - reprezentare înlănţuită (simplu/dublu, folosind alocare dinamică/alocare statică)
- O listă sortată (ordonată) în raport cu cheile elementelor
  - reprezentare secvenţială pe tablou (vector dinamic) .
  - reprezentare înlănţuită (simplu/dublu, folosind alocare dinamică/alocare statică)

- Structura de heap.

  Avantaje - Dezavantaje

O implementare care utilizează un **array neordonat** ca structură de date de bază. Operația *găsește maxim* este implementat prin *parcurgerea* array-ului pentru a găsi valoarea maximă, apoi *schimbă* elementul maxim cu ultimul element și *decrementează* dimensiunea cozii.

```
#include <stdlib.h>
#include "Item.h"
static Item *pq;
static int N;

void PQinit(int maxN)
{ pq = malloc(maxN*sizeof(Item)); N=0;}

int PQempty() { return N == 0; }

void PQinsert(Item v)
{ pq[N++] = v; }

Item PQdelmax ()
 { int j, max = 0;
   for (j = 1; j < N; j ++ )
     if (less(pq[max] , pq[j])) max = j;
   exch(pq[max] , pq[N]);
   return --N;
 }
```

Exemplu de coadă cu priorităţi ca array pentru o secvenţă de operaţii:
litera = inserează
* = sterge maximum

(Sedgewick)

Complexitatea operaţiilor cozilor cu priorităţi în cazul cel mai defavorabil

	insert	delete maximum	delete	find maximum	change priority	join
ordered array	N	1	N	1	N	N
ordered list	N	1	1	1	N	N
unordered array	1	N	1	N	1	N
unordered list	1	N	1	N	1	1
heap	$\lg N$	$\lg N$	$\lg N$	1	$\lg N$	N
binomial queue	$\lg N$	$\lg N$	$\lg N$	$\lg N$	$\lg N$	$\lg N$
best in theory	1	$\lg N$	$\lg N$	1	1	1

Structura de heap

O structură de date simplă numită heap (grămadă) este o SD care poate sprijini eficient operațiile de bază ale cozii cu priorităţi.

Într-un heap, înregistrările sunt stocate într-un array, astfel încât *fiecare cheie* este *garantat mai mare decât cheile de pe două poziții determinate*. La rândul său, fiecare dintre aceste chei trebuie să fie mai mare decât alte două chei și așa mai departe. Această ordonare este ușor de înteles dacă privim cheile ca fiind într-o structură de arbore binar; arcele de la fiecare cheie duc la cele două chei cunoscute a fi mai mici.

Un arbore binar este complet plin, dacă acesta este de înălțime *h* , și are $2^{h+1}-1$ noduri .

Un arbore binar de înălțime h, este complet dacă și numai dacă :

- este gol sau
- subarborele stâng este complet de înălțime h - 1 și subarborele său drept este complet plin de înălțime h - 2 sau
- subarborele stâng este complet plin de înălțime h - 1 și subarborele său drept este complet de înălțime h - 1

Un arbore complet este umplut de la stânga :

- toate frunzele sunt pe același nivel sau pe două adiacente și
- toate nodurile de pe nivelul cel mai scăzut sunt cât mai spre stânga posibil

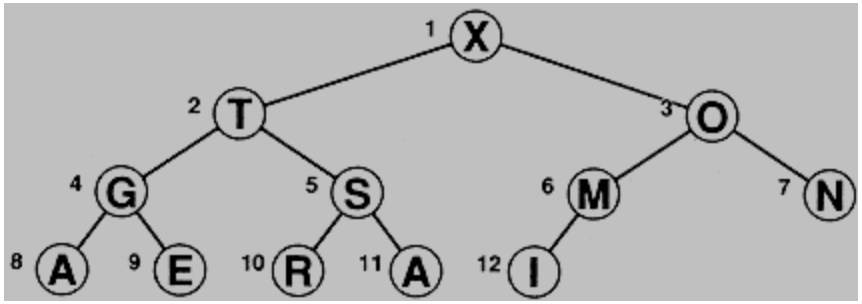
Definiție 1: Un arbore este ordonat ca heap în cazul în care cheia din fiecare nod este mai mare sau egală cu cheile în toți copiii nodului (dacă este cazul). Echivalent, cheia în fiecare nod al unui arbore ordonat ca heap, este mai mică decât sau egală cu cheia părintelui acelui nod (dacă este cazul)

Definiția 2: Un heap este o mulţime de noduri cu chei aranjate într-un arbore binar complet ordonat ca heap, reprezentat ca array.

Proprietate 1: Nici un nod al unui arbore ordonat ca heap nu are o cheie mai mare decat cheia din rădăcină.

k	1	2	3	4	5	6	7	8	9	10	11	12
a[k]	X	T	O	G	S	M	N	A	E	R	A	I

(Sedgewick)



Remember

Prin *traversarea* unui arbore binar vom înţelege parcurgerea tuturor nodurilor arborelui, trecând o singură dată prin fiecare nod.

În funcţie de ordinea (disciplina) de vizitare a nodurilor unui arbore binar, există trei moduri de baza de traversare:

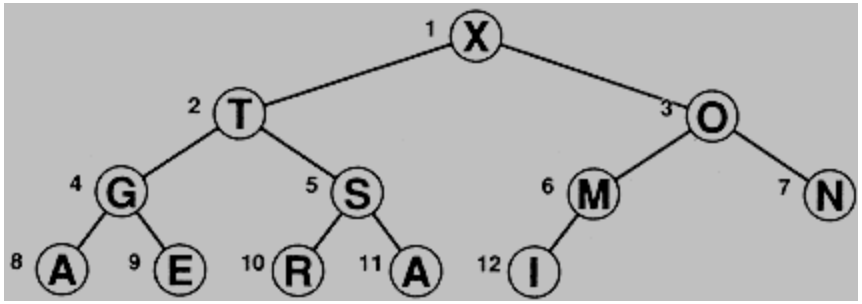
- *în preordine:* vizitează mai întâi nodul (rădăcină), apoi subarborele stâng şi după aceea subarborele drept
- *în inordine*: vizitează mai întâi subarborele stâng , apoi nodul (rădăcină) şi după aceea subarborele drept

- *în postordine*: vizitează mai întâi subarborele stâng , apoi subarborele drept şi după aceea nodul (rădăcină)

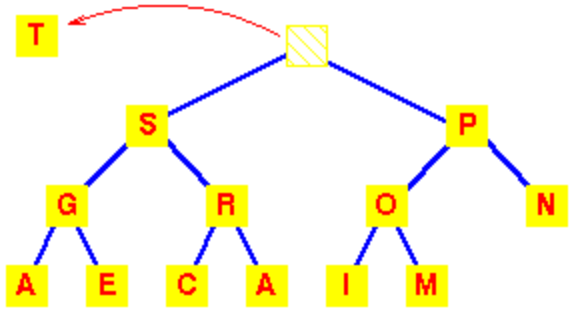
New !

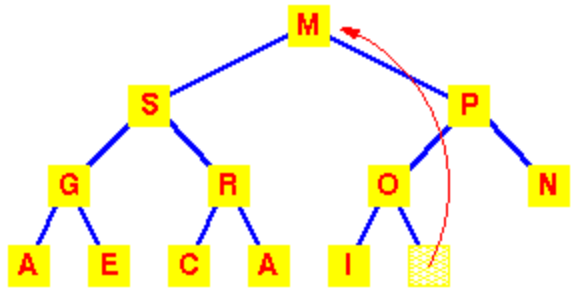
- *level order:* nodurile sunt parcurse de la stânga la dreapta şi de sus în jos

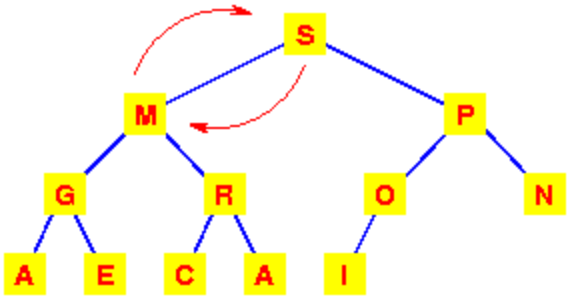
Lângă fiecare nod din arborele pe care l-am reprezentat se află câte un număr, reprezentând poziţia în vector pe care ar avea-o nodul respectiv. Pentru cazul considerat, vectorul echivalent ar fi
H = (X T O G S M N A E R A I).
Se observă că nodurile sunt parcurse de la stânga la dreapta şi de sus în jos. O proprietate necesară pentru ca un arbore binar să se poată numi *heap* este ca toate nivelurile să fie complete, cu excepţia ultimului, care se completează începând de la stânga şi continuând până la un anume punct. De aici deducem că înălţimea unui heap cu *N* noduri este lg*n*. Reciproc, numărul de noduri ale unui heap de înălţime *h* este $N \in [2^h, 2^{h+1} - 1]$
Din această organizare rezultă că **părintele** unui nod *k > 1* este nodul ***[k/2],*** iar **copii** nodului *k* sunt nodurile ***2k*** si ***2k+1***. Dacă *2k = N*, atunci nodul *2k+1* nu există, iar nodul *k* are un singur fiu; dacă *2k > N*, atunci nodul *k* este frunză şi nu are nici un copil.

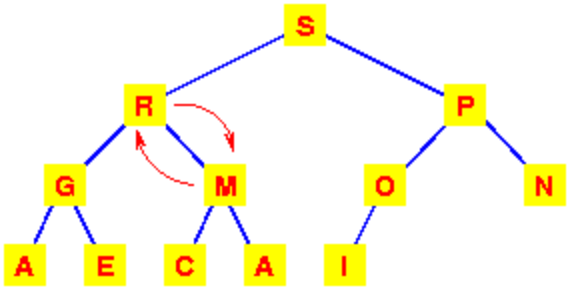


(Sedgewick)

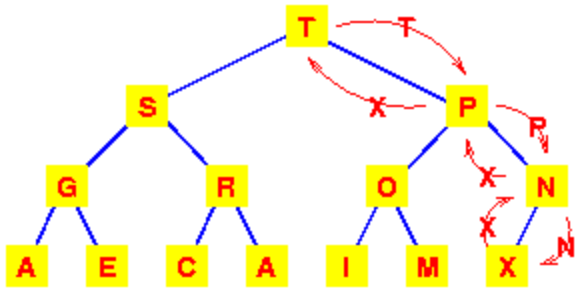








modifying ~*heapifying fixing*



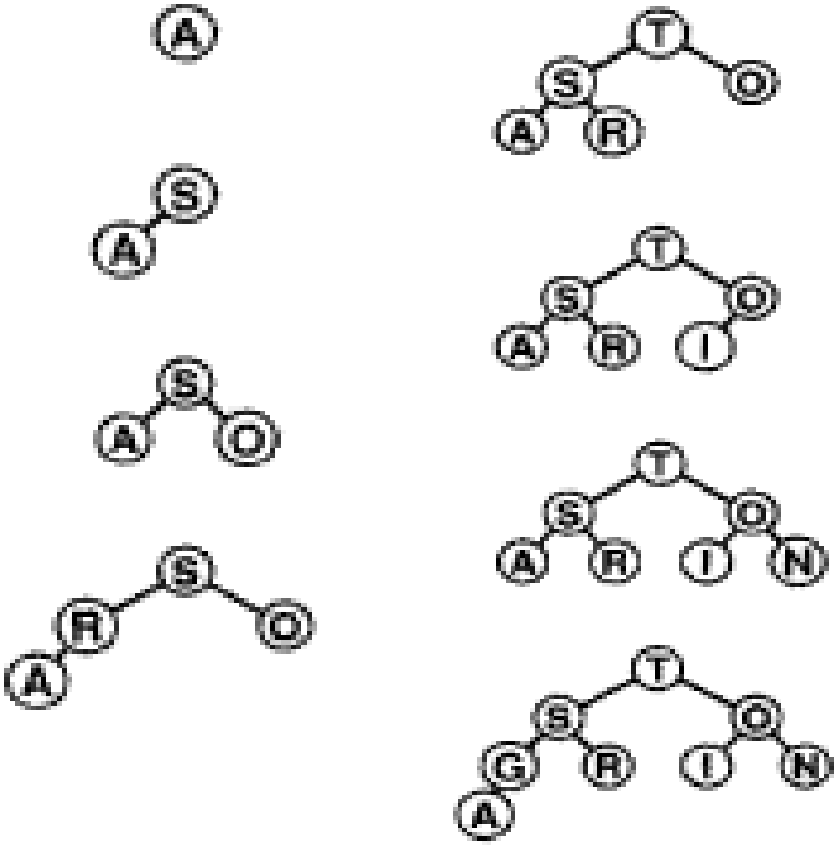

```
Bottom-up heapify
fixUp(Item a[], int k)
{
  while (k > 1 && less(a[k/2], ark]))
{ exch(a[k], a[k/2]); k k/2;}
}
```

```
Top-down heapify
fixDown(Item a[], int k, int N)
{ int j;
    while (2*k <= N)
      { j = 2*k;
        if (j < N && less(a[j], a[j+l])) j++;
        if (!less(a[k], a[j])) break;
       exch(a[k], a[j]); k = j;
      }
}
```

Un heap poate fi folosit ca o coadă cu priorităţi: elementul cu cea mai mare prioritate este în rădăcina și în mod trivial este extras . Dar dacă rădăcina este ștearsă, au rămas doi subarbori și trebuie re-creat eficient un singur arbore cu proprietatea de heap .

Proprietate 2: Operaţiile *insert* și *delete maximum* într-un TAD coadă cu priorităţi cu *n* elemente implementată cu heap se efectuează în timp **O(log*n*)**. (insert număr comparaţii $\leq \lg n$, iar deletemax $\leq 2\lg n$)

Proprietate 3: Operaţiile *change priority, replace the maximum* și *delete* într-un TAD coadă cu priorităţi cu *n* elemente pot fi implementate cu arbori ordonaţi cu structura de heap astfel incât sa nu se efectueze mai mult de $2\lg n$ comparaţii.



(Sedgewick)

```
//Coada cu prioritati implementata
//prin heap

#include <stdlib.h>
#include "Item.h"
static Item *pq;
static int N;
void PQinit(int maxN)
{ pq = malloc«maxN+1)*sizeof(Item));
  N = 0; }

int PQemptyO { return N == 0; }

void PQinsert(Item v)
{
  pq[++N] = v;
  fixUp(pq, N);
}

Item PQdelmax ()
{
     exch(pq[l], pq[N]);
     fixDown(pq, 1, N-1);
     return pq[N--];
}
```

Construirea top-down a unui heap

Heapsort

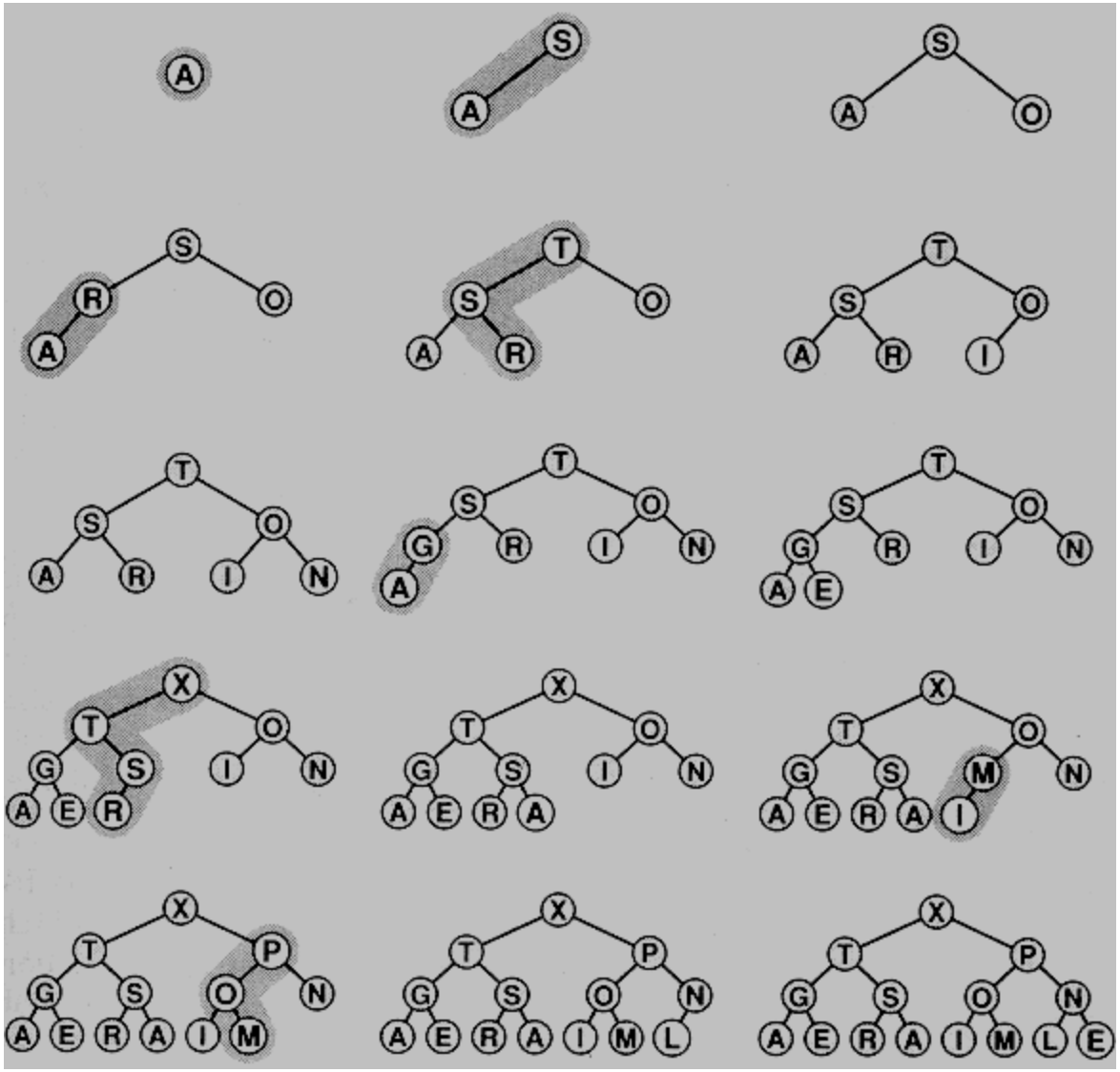
Pentru a sorta un subarray a [l], …, a [r] folosind un ADT coadă cu priorităţi, noi pur și simplu vom folosi *PQinsert* pentru a pune toate elementele în coada cu priorităţi, iar apoi utilizăm *PQdelmax* pentru a le elimina, în ordine descrescătoare. Acest algoritm de sortare se execută în timp proporțional cu *n*lg*n*, dar folosește spațiu suplimentar proporțional cu numărul de elemente care trebuie sortate (pentru coada cu priorităţi).

```
void PQsort(Item a[], int 1, int r)
{ int k;
  Pqinit();
  for (k = 1; k <= r; k++) PQinsert(a[k]);
  for (k = r; k >= 1; k--) a[k] = PQdelmax();
}
```

Costul total este < lg*n* + … + lg2 + lg1 = lg*n!*

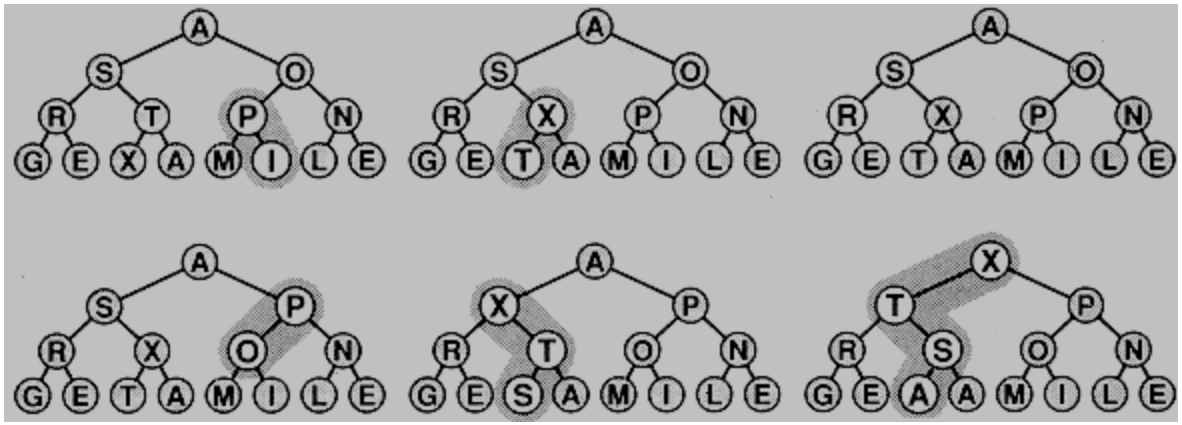
$n! \approx \left(\frac{n}{e}\right)^n \sqrt{2\pi n}$ (Stirling)

Construire *Top-Down* a heapului*: ASORTINGEXAMPLE*



(Sedgewick)

Construire *Bottom-Up a heapului: ASORTINGEXAMPLE*

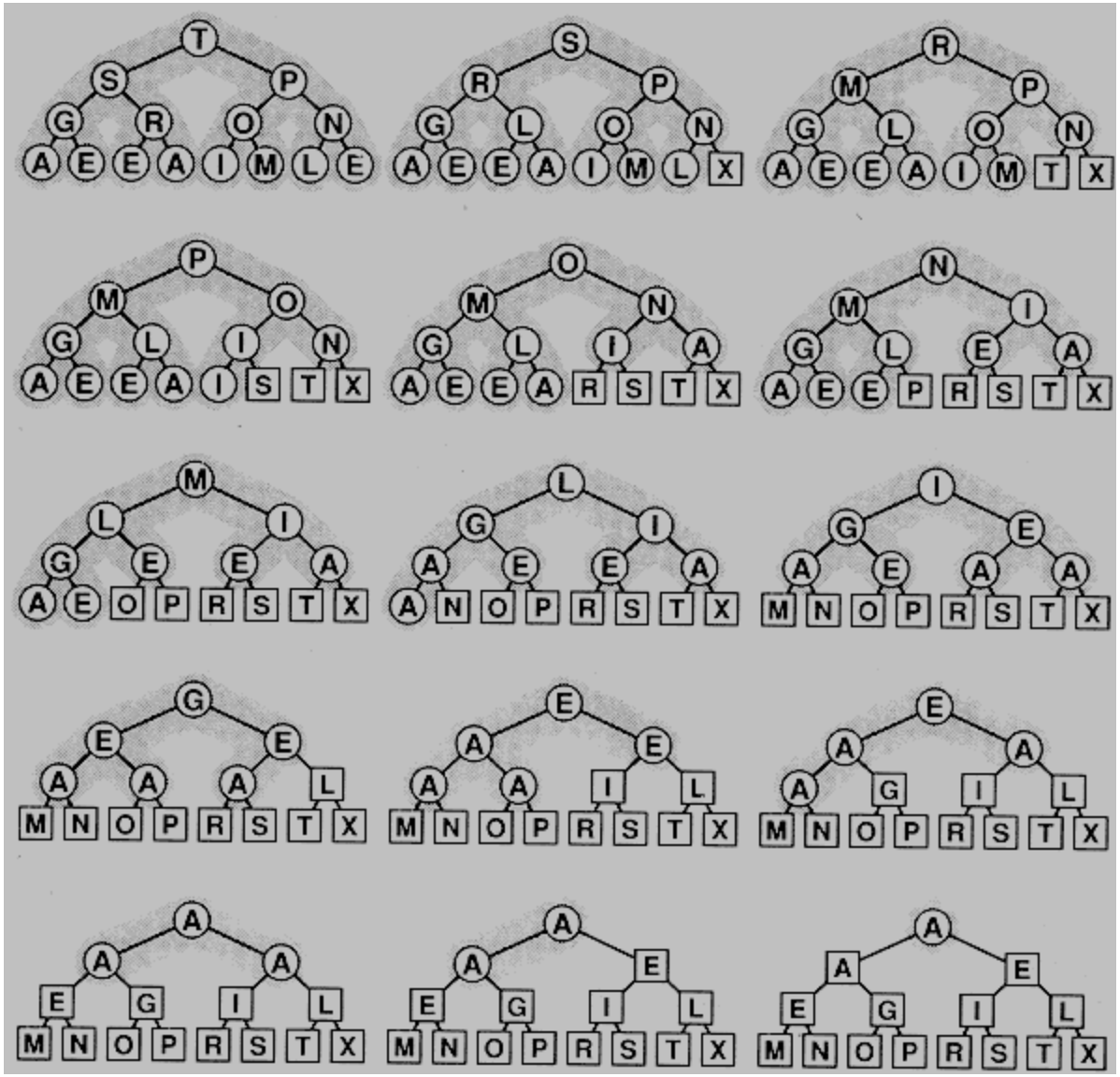


(Sedgewick)

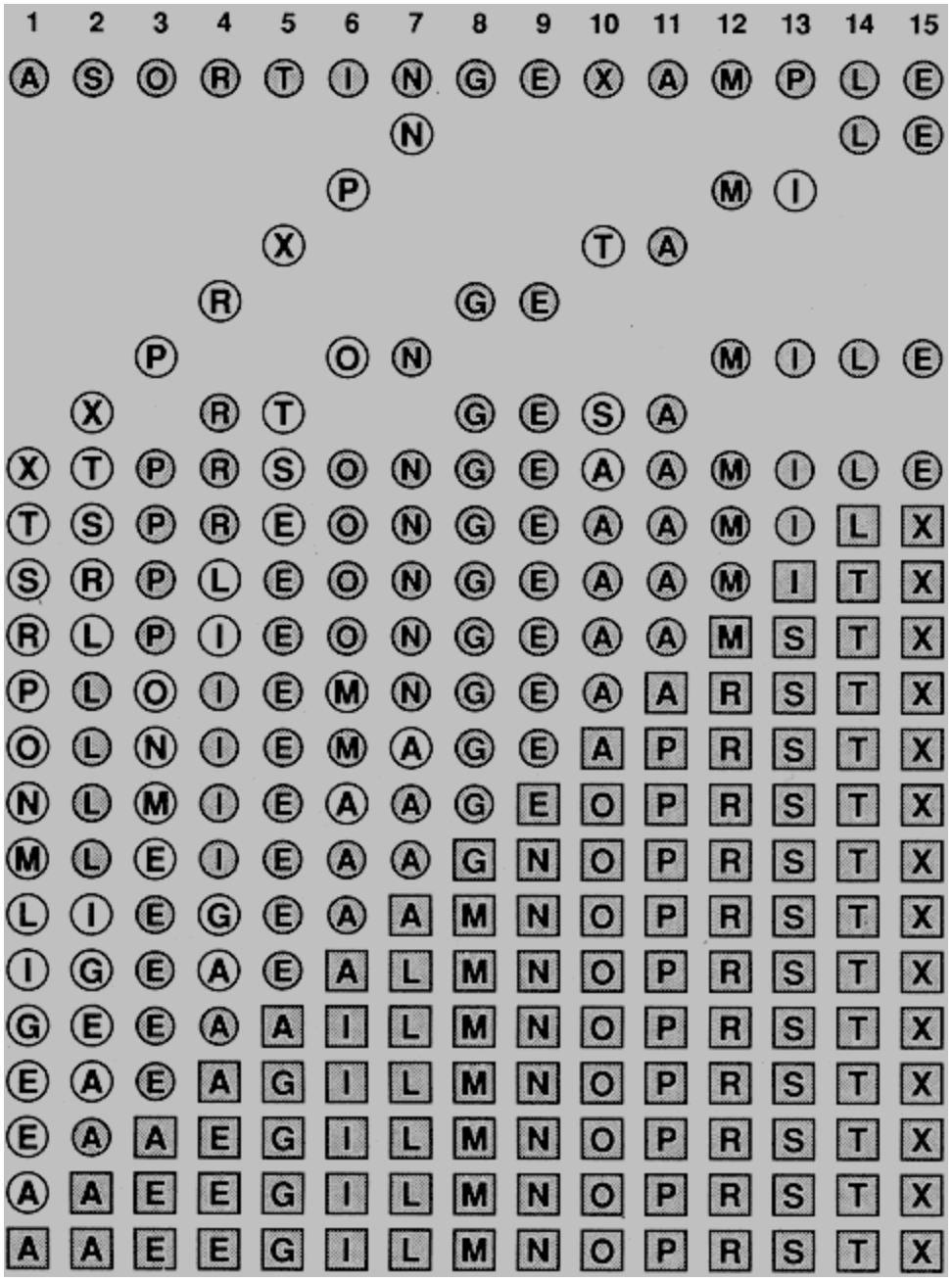
Proprietate 4: Construirea *Bottom-Up* a heapului necesită un timp liniar.

Proprietate 5: Heapsort folosește mai puţin de *n*lg*n* comparaţii pentru a sorta *n* elemente.

Sortare din heapul cunstruit =>AAEEGILMNOPRSTX



(Sedgewick)


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
A S O R T I N G E X A M P L E
N L E
P M I
X T A
R G E
P O N M I L E
X R T G E S A
X T P R S O N G E A A M I L E
T S P R E O N G E A A M I L X
S R P L E O N G E A A M I T X
R L P I E O N G E A A M S T X
P L O I E M N G E A A R S T X
O L N I E M A G E A P R S T X
N L M I E A A G E O P R S T X
M L E I E A A G N O P R S T X
L I E G E A A M N O P R S T X
I G E A E A L M N O P R S T X
G E E A A I L M N O P R S T X
E A E A G I L M N O P R S T X
E A A E G I L M N O P R S T X
A A E E G I L M N O P R S T X
A A E E G I L M N O P R S T X

(Sedgewick)

Heap-uri indirecte

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
p[k]	10	5	13	4	2	3	7	8	9	1	11	12	6	14	15
a[p[k]]	X	T	P	R	S	O	N	G	E	A	A	M	I	L	E
q[k]	10	5	6	4	2	13	7	8	9	1	11	12	3	14	15

Unde este greșeala?
Temă!

Decât a rearanja cheile într-un domeniu, este mai avantajos să lucrăm cu un domeniu de indici *p* care se referă la un array a. a[p [k]] este, prin urmare, înregistrarea corespunzătoare a elementului k al heap-ului, pentru k între 1 și N. In plus utilizăm un alt array q în care este stocată poziţia în heap al celui de-al k-lea element din array. Astfel, ne-am permite și operațiile de change/ schimbare și de delete/ștergere . Prin urmare , numărul 1, este înregistrarea în q pentru cel mai mare element din vector, etc. Dacă dorim, de exemplu, să schimbăm valoarea unui a[k], putem găsi poziția în heap în q [k], iar după modificare se va folosi upheap sau downheap. În figură, sunt date valorile din aceşti vectori pentru heapul nostru bottom-up (slide 24) ; observăm că p [q [k]] = q [p [k]] = k pentru orice k de la 1 la N.


Die Zeit sollte immer oberste Priorität haben.
Zeit für Begegnungen mit anderen und mit sich selbst.
Zeit zum Träumen, zum Lachen, zum Lieben.
Zeit zum Leben.
Timpul ar trebui sa aiba intotdeauna cea mai mare pioritate.Timp pentru intalniri cu altii si cu sine. Timp pentru a visa, pentru a rade, pentru a iubi. Timp pentru a trai.