Structuri de date pentru multimi. Structuri de date pentru cozi de prioritati.

Dictionar vs multime.
Tabele de dispersie.
Cozi de prioritati. Heap-uri

Motivatie

- Utilizarea datelor pe baza unei chei:
  - Doar cheie (e.g. nume studenti):

    Ex: “Ana Coman”, “Ana Cojocaru”, “Filip Pop”

    eStudent(“Ana Coman”) returneaza *true*
  - Cheie + valoare (e.g. nume + note studenti):

    Ex: <“Ana Coman”, 7>, <“Ana Cojocaru”, 6>, <“Filip Pop”, 9>

    Nota(“Ana Coman”) returneaza 7.

    Nota(“Filip Pop”) returneaza 9.

- Alte exemple:
  - <nume film, <actori, plot, gen, etc>>
  - <url, web page>
  - <network id, <cost/metric, next hop, QoS, interface, etc >>

ADT Multime (Set)

- Focus pe stocare/cautare date!
  - i.e. relatia de *apartenenta*
- Data:
  - chei ***comparabile*** si ***<u>unice</u>***
- Operatii:
  - *insert(key)*
  - *find(key)*
  - *delete(key)*

insert (cflorescu)

- **acoman**
- **acojocaru**
- **fpop**
- **ipop**
- **bcampean**
- **aistrate**
- **bmicle**
- **rmuresan**
- **apopescu**
- **…**

find (bmicle)

ADT Dictionar (Dictionary)

- Focus pe stocare/cautare date!
- Data:
  - perechi <cheie, valoare>
  - cheile sunt asociate la valoare
  - chei ***comparabile*** si ***unice***
- Operatii:
  - *insert(key, value)*
  - *find(key)*
  - *delete(key)*

insert (cflorescu, …)

- **acoman (Ana Coman …)**
- **acojocaru (Ana Cojocaru, ….)**
- **fpop (Florina Pop, …)**
- **ipop (Iulia Pop, …)**
- **Bcampean (Bogdan Campean, …)**
- **aistrate (Anca Istrate, …)**
- **bmicle (Bogdan Micle, …)**
- **rmuresan (Rodica Muresan, …)**
- **apopescu (Anca Popescu, …)**

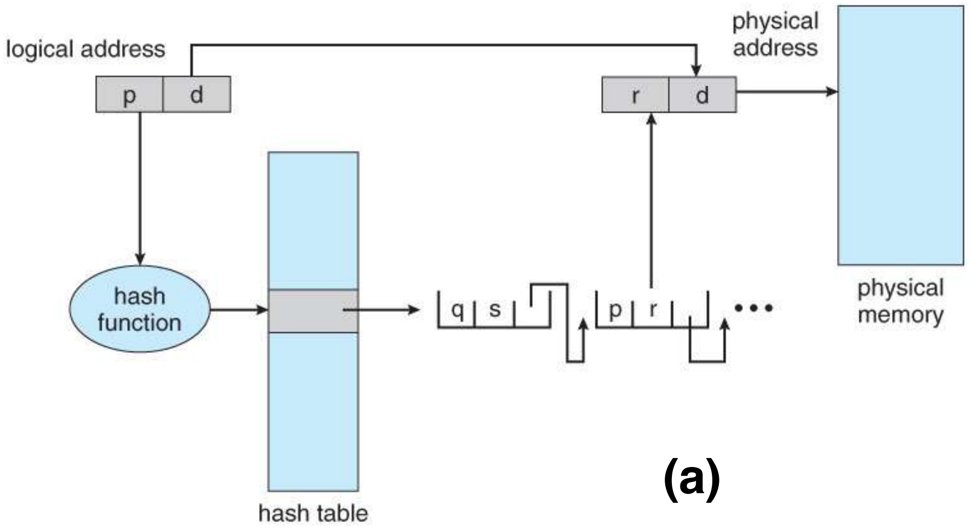
find (bmicle)

Multime vs dictionar

- In esenta identice
  - multimea nu are valori, doar chei
  - se pot utiliza aceleasi structuri pentru a le implementa
- Exceptie:
  - daca avem nevoie sa implementam operatii matematice pe multimi
    - reuniune, intersectie, complement, etc.
    - optiuni mai bune pentru asa ceva, decat ceea ce se potriveste si pt. dictionare

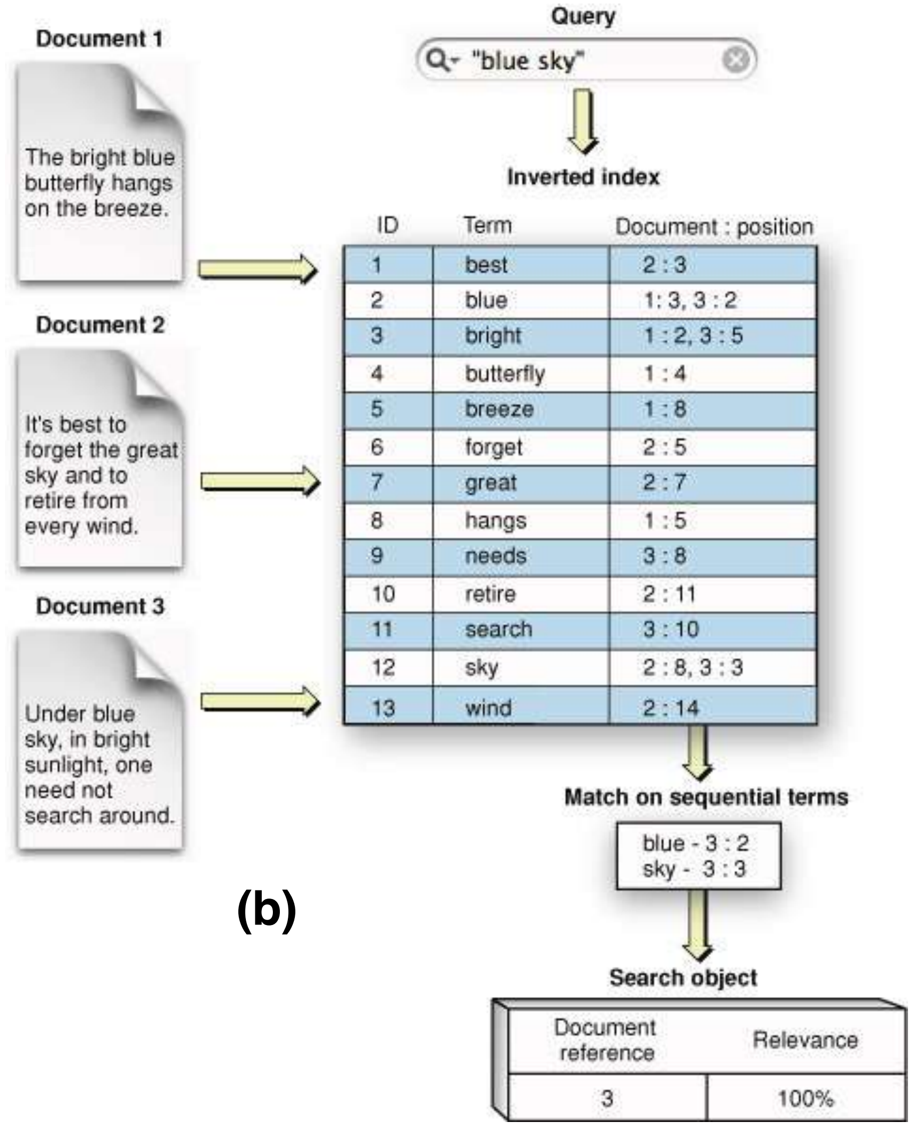
Utilizari...

- Acces eficient la info in:
  - retele: tabele de rutare
  - sisteme de operare**: tabele de paginare** (a)
  - compilatoare: tabele de simboluri
  - cautare in documente: **inverted index**, indexare (b)



(a)

https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/8_MainMemory.html



(b)

https://developer.apple.com/library/content/documentation/UserExperience/Conceptual/SearchKitConcepts/searchKit_basics/searchKit_basics.html

Implementare: sir, liste inlantuite

	insert	find	delete
sir nesortat	O(1)	O(n)	O(n)
lista simplu inlantuita nesortata	O(1)	O(n)	O(n)
sir sortat	O(n)	O(log n)	O(n)*
lista simplu inlantuita sortata	O(n)	O(n)	O(n)

performanta in cazul defavorabil (n - nr de elemente)

* *Stergerea amanata in vectori sortati:*

10	12	24	30	41	42	44	50
✓	✗	✓	✓	✓	✗	✓	✓

Avantaje:
- eliminarea propriu-zisa in grup
- re-adaugarea se face prin modificarea marcajului

Dezavantaje:
- memorie aditionala liniara
- se iroseste memorie la stergeri multe
- cautarea, pt versiunea sortata, in *O(log m)*, m – capacitatea structurii
- se pot complica restul operatiilor

Implementare: ABC, ABC echilibrati

		insert	find	delete
ABC	*mediu*	O(log n)	O(log n)	O(log n)
	defav.	O(n)	O(n)	O(n)
ABC echilibrati (AVL, Red-Black)	*med/defav*	O(log n)	O(log n)	O(log n)

- ABC - cazul defavorabil - operatii in timp liniar, O(n)
- conditii de echilibrare:
  - numar de noduri
  - inaltime
  - etc.

Nevoia de viteza !

http://www.teamvvv.com/en/news/comments/Need-for-Speed-Review

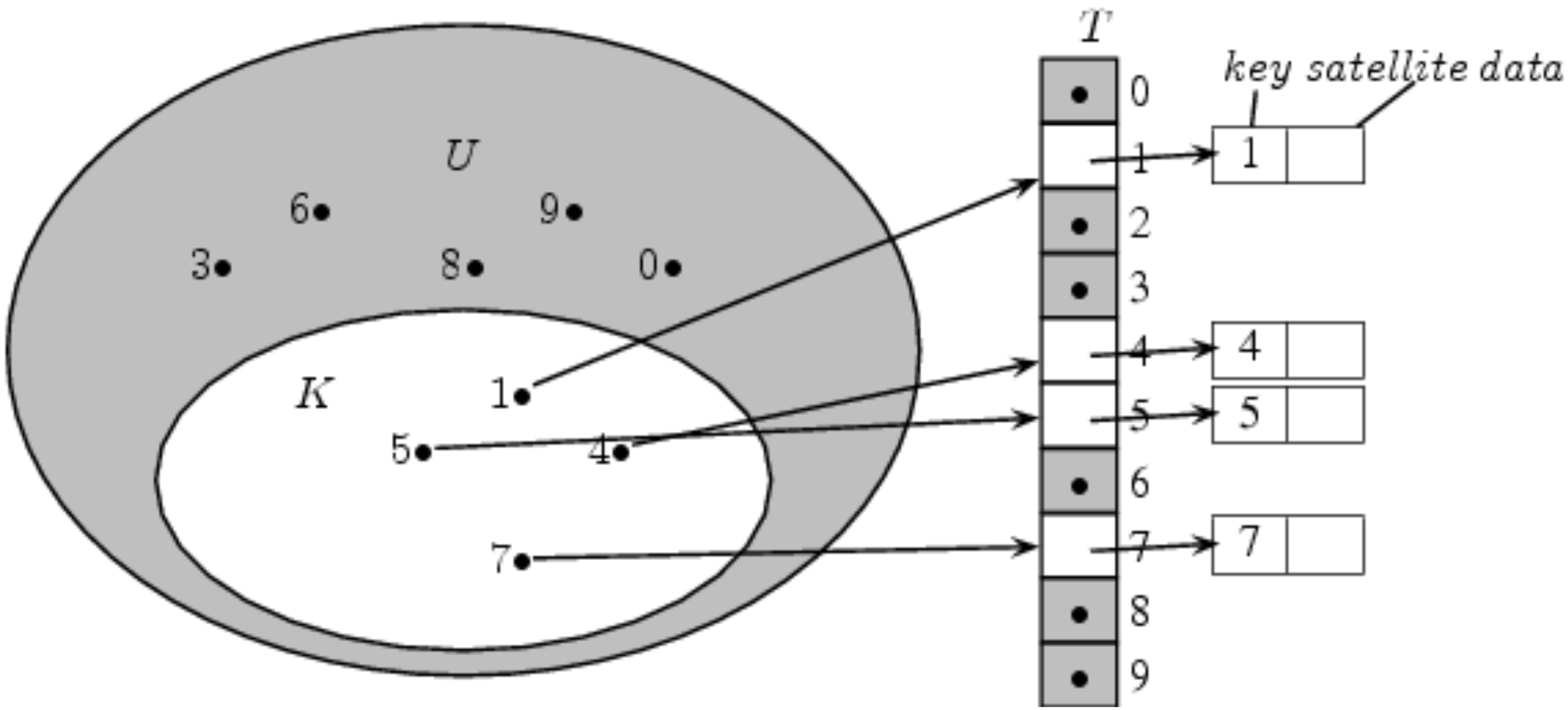
Structurile de date pe care le-am analizat pana acum folosesc comparatii pentru a gasi un element

Au nevoie de $O(\log_2 n)$ sau $O(n)$ pentru o cautare, inserare

- In aplicatiile reale n are des valori intre 100 si 100000 sau mai mult deci $\log_2 n$ este intre 6.6 si 16.6
- Putem sa proiectam o structura care functioneaza in $O(1)$ pentru cautare si inserare?

Implementare: Tabela cu acces direct

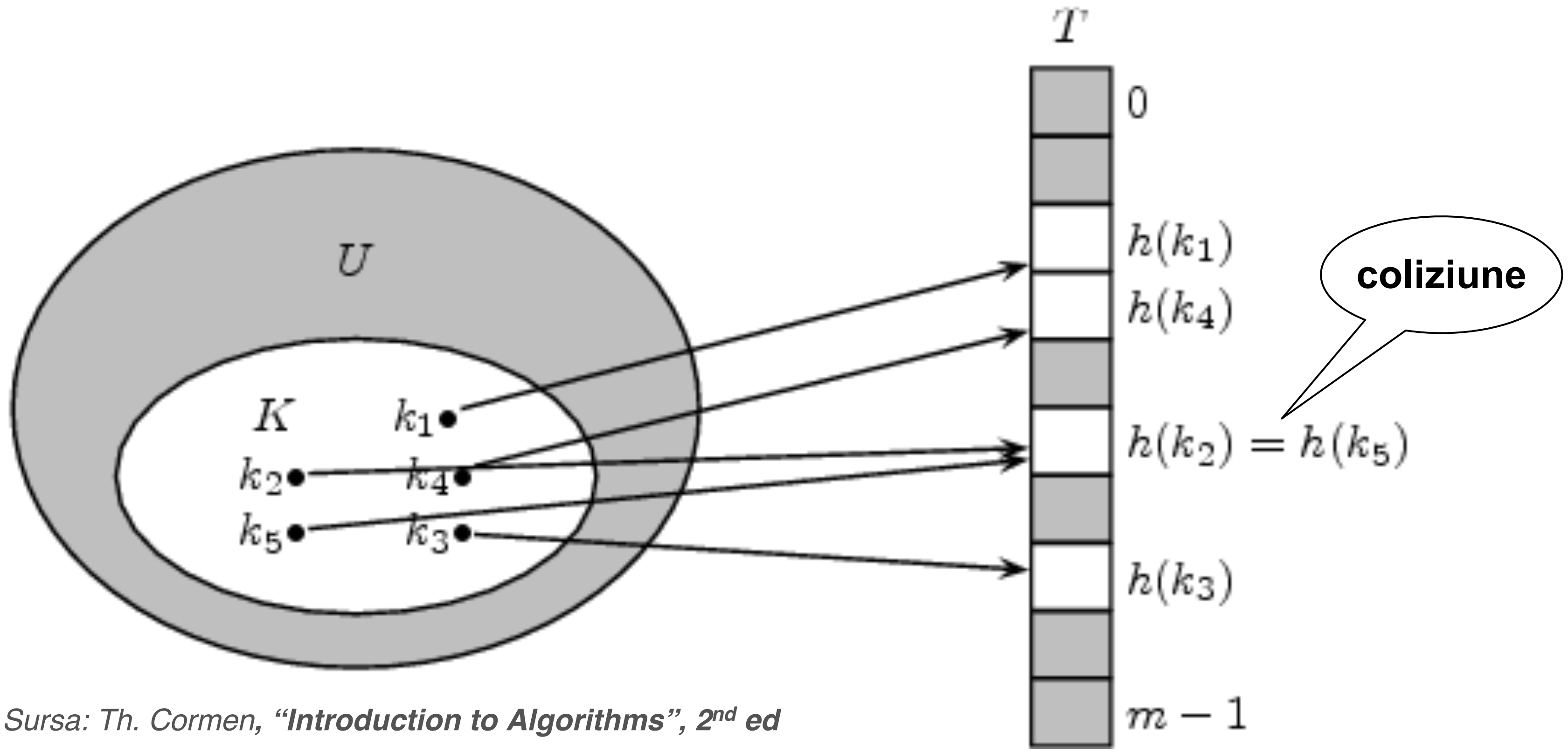
- *U = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}*
- *K = {1, 4, 5, 7} cheile multimii*



*Sursa: Th. Cormen, **"Introduction to Algorithms", 2nd ed***

Eficienta (timp)? Memorie?

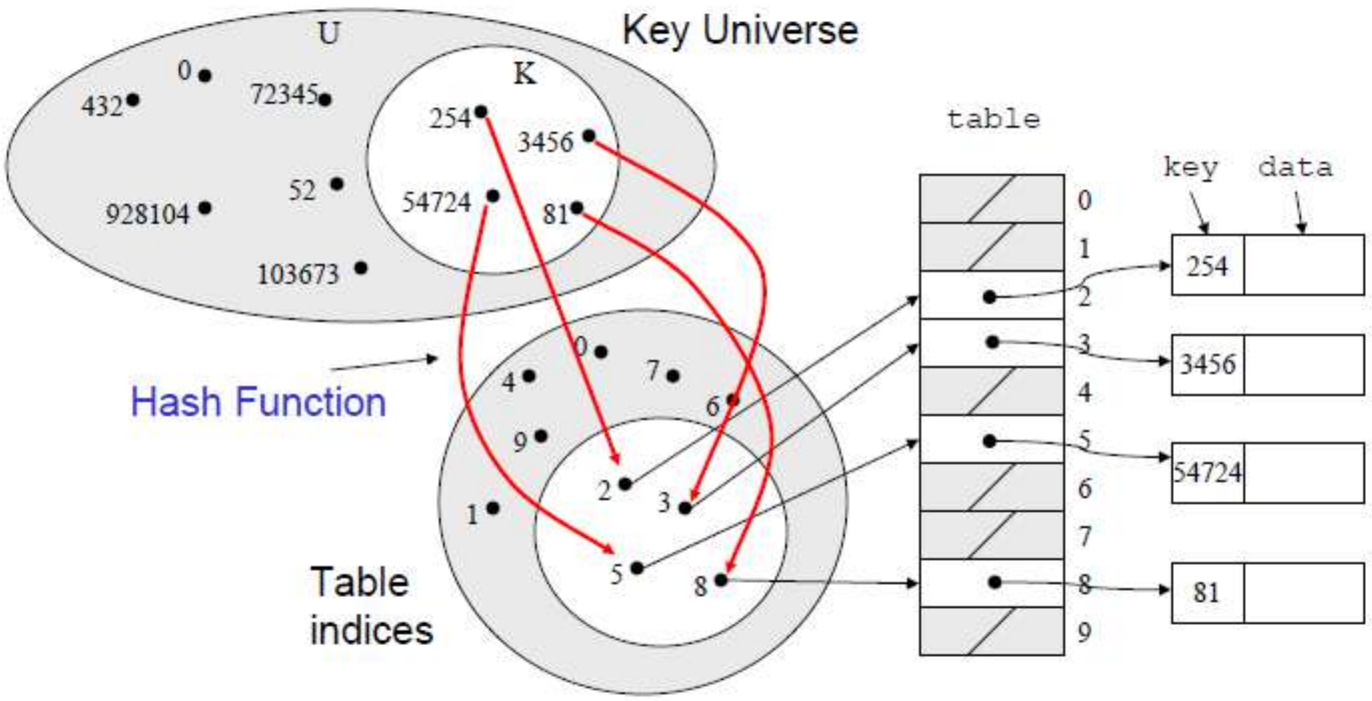
Implementare: Tabela de dispersie



Sursa: Th. Cormen, ***"Introduction to Algorithms", 2nd ed***

- generalizeaza notiunea de vector
- potrivita pentru cazul in care |K| << |U|

Implementare: Tabela de dispersie



https://courses.cs.washington.edu/courses/cse373/

- generalizeaza notiunea de vector
- potrivita pentru cazul in care |K| << |U|

Tabela de dispersie

- Scop:
  - reducerea cantitatii de memorie la $\Theta(|K|)$
  - cautare eficienta (O(1) e posibil?)
    - da, in cazul mediu! (defavorabil - O(n))
- ***Functie de dispersie (hashing)***: *h:U->{0,1,…,m-1}, unde m este dimensiunea tabelei T (m<<|U|)*
  - mapeaza universul cheilor posibile in spatiul disponibil de adrese (i.e. dimensiunea tabelei)
  - elementul cu cheia *k* este mapat la adresa *h(k)*
  - *e.g. h(k) = k mod m*
- Ce problema ar putea aparea la o asemenea abordare?

Tabela de dispersie - coliziune

http://rockstartemplate.com/photography/games/nfs-full-car-collection/

Coliziune: doua chei diferite sunt mapate pe aceeasi adresa

Cum rezolvam problema coliziunilor?

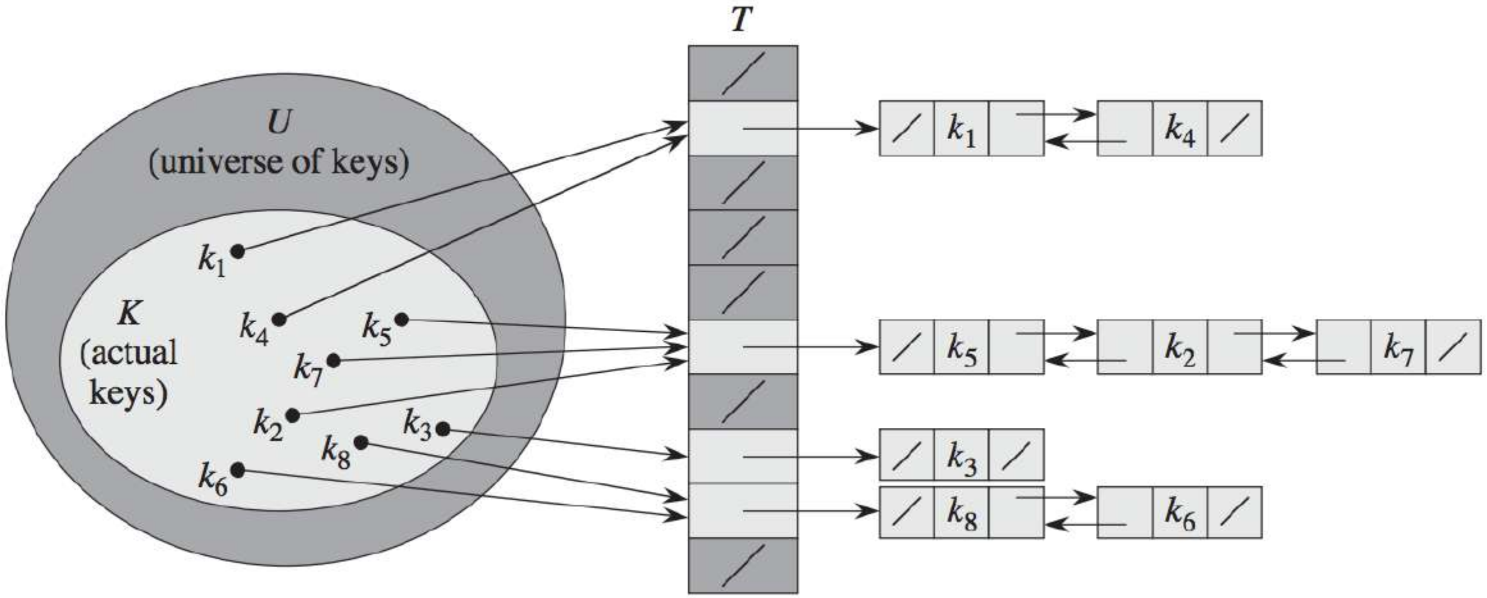
- *Evitare:*

  functie de dispersie care sa se comporte aparent "aleator" ("*hash*" - *(a) chop into small pieces; (b) confuse, muddle*)

- *Rezolvare/reparare:*
  - inlantuire: ("*chaining*")
  - adresare deschisa ("*open addressing*")

Tabela de dispersie: Chaining

- toate elementele cu aceeasi valoare a functiei de dispersie sunt stocate intr-o lista inlantuita
- tabela de dispersie contine, la pozitia *j*, adresa primului element din lista cheilor (din tabela) care au *h(k)* = *j*



Sursa: Th. Cormen, ***"Introduction to Algorithms", 2nd ed***

Tabela de dispersie: Chaining

- Tabela ***T***, de dimensiune ***m***, care stocheaza ***n*** elemente:

  $$Factorul\ de\ umplere\ \alpha = \frac{n}{m}, \alpha > 1,\ \alpha = 1,\ \alpha < 1$$

- Operatii:
  - *CHAINED-HASH-INSERT(T, x)*

    list-insert(T[h(x.key)], x)
  - *CHAINED-HASH-SEARCH(T, k)*

    list-search(T[h(k)], k)
  - *CHAINED-HASH-DELETE(T, x)*

    list-delete(T[h(x.key)], x)

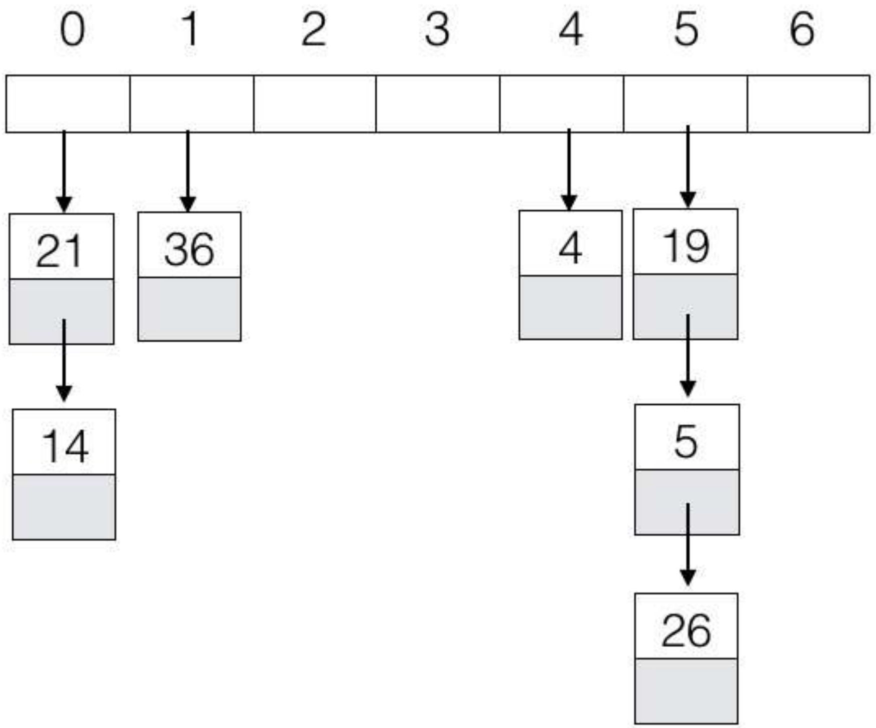
Obs: x este o tupla <key, value>, operatiile se fac pe cheie

Chaining – exemplu

m=7

h(k) = k mod m

Insert 36, 26, 14, 5, 21, 4, 19



$\alpha = ?$

Tabela de dispersie: Chaining - Analiza

- cazul defavorabil:
  - toate cheile se mapeaza pe aceeasi adresa din tabela
  - tabela de dispersie = lista inlantuita
- cazul mediu: depinde de cat de bine distribuie functia de dispersie cheile in cele *m* sloturi posibile
  - **Presupunere**: *dispersie simpla, uniforma*
    - *Probabilitate(h(k) = i) = 1/m, ∀ i in [0, m-1]*
  - *daca* n_j - lungimea listei *T[j]*: $n = \sum_{j=0}^{m-1} n_j$
  - n_j - *val. medie:* $\alpha = \frac{n}{m}$

Tabela de dispersie: Chaining - Analiza

- Search:
  - cheie negasita: $\Theta(1 + \alpha)$
    - calculul *h(k)* si parcurgerea listei corespunzatoare (lungime medie α)
  - cheie gasita: $\Theta(1 + \alpha)$
    - vezi teorema 11.2 (*Cormen, ed. 3*)
- Insert?

Tabele de dispersie: Adresare Deschisa

- Adresare deschisa: toate elementele sunt stocate in tabela (α<=1)
- functia de dispersie genereaza o permutare a spatiului de adrese
- in cautare/inserare se ***probeaza*** spatiul de adrese, in functie de acea permutare

```
HASH-SEARCH(T, k)
  i=0
  repeat
    j=h(k,i)
    if T[j]==k //found!!

      return j
    i=i+1
  until T[j]==NIL or i == m
  return NIL //not found!!!
```

```
HASH-INSERT(T, k)
  i=0
  repeat
    j=h(k,i)
    if T[j]==NIL //found empty!!
      T[j] = k
      return j
    else i=i+1
  until i == m
  error “hash table overflow”
```

Tabele de dispersie: Adresare Deschisa

- Analiza:
  - ***Presupunere***: *dispersie uniforma*
    - secventele de proba ale cheilor sunt uniform distribuite (probabilitatile de aparitie celor m! permutari sunt egale)
- Tehnici de generare a *secventelor de proba (i.e. permutarea spatiului adreselor pt. o cheie)*
  - *linear probing (Verificare liniara)*
  - *quadratic probing (Verificare patratica)*
  - *double hashing (Dispersie dubla)*

Tabele de dispersie: Adresare Deschisa

- **Linear probing** (Verificare Liniara):
  - *h(k, i) = (h'(k)+i) mod m, h'(k) - functie de dispersie auxiliara*
  - *Fiind data o cheie k locatiile vor fi examinate in ordinea urmatoare:*
  - T[h'(k)], T[h'(k)+1]…T[m-1], T[0], …, T[h'(k)-1]
  - doar *m* secvente diferite! (din *m!* posibile)
  - **!!! clusterizare primara: două chei care sunt mapate <u>initial</u> la *adrese diferite* pot concura pentru aceleași locații în iteratii succesive**
  - **Se formeaza siruri lungi de locatii ocupate**

  - **Vezi exemplu !**

Tabele de dispersie: Adresare Deschisa

- **Linear probing** (Verificare Liniara):

[0]	72
[1]	
[2]	18
[3]	43
[4]	36
[5]	
[6]	6
[7]	

Add the keys 10, 5, and 15 to the previous table .

Hash key = key % table size

2 = 10 % 8

5 = 5 % 8

7 = 15 % 8

[0]	72
[1]	15
[2]	18
[3]	43
[4]	36
[5]	10
[6]	6
[7]	5

Elementul de inserat va fi stocat in urmatorul loc liber din tabel (daca acesta nu e plin).

Se implementeaza o cautare liniara a unui loc liber incepand din pozitia in care a avut loc coliziunea.

http://faculty.cs.niu.edu/~freedman/340/340notes/340hash.htm

Daca ajungem la sfarsitul fizic al tabelei cautarea continua cu inceputul tabelei.
Daca nu s-a gasit niciun loc liber si s-a ajuns cu cautarea la locul coliziunii, atunci tabela este plina.

Tabele de dispersie: Adresare Deschisa

- **Quadratic probing (verificare patratica)**:
  - $h(k, i) = (h'(k)+c_1 i+ c_2 i^2) \bmod m,$
    - $c1$ *si* $c2 \neq 0$ *- constante auxiliare*
  - T[h'(k)], urmata de pozitii care depind cuadratic de *i:*
  - mai bine decat linear probing, DAR! pt a genera cat mai multe adrese, val. c_1, c_2 si m ar trebui constranse
  - **!!! clusterizare secundara (daca $h(k_1, 0) = h(k_2, 0)$): două chei care sunt mapate <u>initial</u> la *aceleasi adrese* pot concura pentru aceleași locații în iteratii succesive**

Tabele de dispersie: Adresare Deschisa

- ***Double hashing*** (dispersie dubla):
  - cea mai buna alternativa
  - permutarile generate au proprietati apropiate de *dispersie uniforma*
  - $h(k, i) = (h_1(k)+ih_2(k)) \bmod m$, $h_1(k)$ si $h_2(k)$ sunt functii de dispersie auxiliare
  - $T[h_1(k)]$, urmata de pozitii incremetatate cu $h_2(k) \bmod m$ (*offset* variabil!)
  - $0 < h_2(k) < m$
  - m^2 secvente diferite se folosesc, fata de *m*

Adresare deschisa - Exemplu

m=7

h'(k) = k mod m

***Linear probing**: $h(k, i) = (h'(k)+i) \bmod m$*

Insert 19, 36, 5, 21, 4, 26, 14

$\alpha = ?$

21	36	26	14	4	19	5
0	1	2	3	4	5	6

Adresare deschisa - Exemplu

m=7

h'(k) = k mod m

Verificare patratica: $h(k, i) = (h'(k)+c_1 i+ c_2 i^2) \bmod m$

$c_1 = 1, c_2 = 1$

Insert 19, 36, 5, 21, 4, 26, 14

5	36	21	26	4	19	14
0	1	2	3	4	5	6

Adresare deschisa - Exemplu

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

$m=7$

Double hashing: $h(k, i) = (h_1(k)+ih_2(k)) \mod m$

$h_1(k)= k \mod m,$

$h_2(k) = 5 – (k \mod 5);$

Insert 19, 36, 5, 21, 4, 26, 14

21	36	26	5	4	19	14
0	1	2	3	4	5	6

Tabele de dispersie: Analiza Adresarii Deschise

- Search (*demonstratie - see Cormen*):
  - cheie negasita:
    - $1/(1-\alpha)$
  - cheie gasita:
    - $1/\alpha * \ln(1/(1-\alpha))$
- Insert?

Tabele de dispersie: Adresare Deschisa

- Stergere
  - marcheaza celula ca `DELETED`
  - ? ce se intampla cand dam peste o celula `DELETED` la
    - inserare
    - cautare
  - stergeri repetate => timpii de cautare nu mai depind de α (de ce?)
  - daca stergerile sunt frecvente, se utilizeaza *chaining*

Tabele de dispersie: Adresare Deschisa

- Stergere

> **Cand stergem o cheie dintr-o locatie i, nu putem marca pur si simplu acea locatie ca fiind libera memorand in ea NULL. Daca punem NULL atunci va fi imposibil sa accesam orice cheie k a carei inserare a verificat locatia i si a gasit-o ocupata.**

  - marcheaza celula ca `DELETED`
  - ? ce se intampla cand dam peste o celula `DELE`

> **HASH-INSERT – se modifica astfel incat sa fie tratate locatiile DELETED ca si cand ar fi libere astfel incat o noua cheie sa poata fi inserata.**

    - inserare
    - cautare

> **HASH-SEARCH – NU se modifica !**

Functia de dispersie

Majoritatea functiilor de dispersie presupun universul cheilor din multimea numerelor naturale.

Daca cheile nu sunt numere naturale trebuie gasita o modalitate pentru a le mapa pe multimea numerelor naturale !

Functia de dispersie

- De regula compusa din 2 parti (daca cheia nu e intreg)
  - ***Cod de dispersie***:

    $\boldsymbol{h}_1$: *chei* $\rightarrow$ *intregi*
  - ***Functie de compresie***:

    $\boldsymbol{h}_2$: *intregi* $\rightarrow [0, m-1]$
- $\mathbf{h(x) = h_2(h_1(x))}$

Functia de dispersie - contd.

- O functie **buna** de dispersie:
  - satisface pp. de ***distributie simpla, uniforma***
  - in general imposibil de verificat! (de ce?)
  - ocazional, se cunoaste distributia cheilor
    - e.g.: numere reale, aleatoare, u.i.d., 0<=k<1, atunci $h(k) = \lfloor km \rfloor$ satisface conditia
- In practica - metode *heuristice* (etim. greaca - “a descoperi”), informatii calitative despre distributia cheilor

Functia de dispersie: hashcode

- Transformare (cast) la intreg:
  - interpretam bitii cheii ca intreg
  - Solutie buna pentru chei de lungime <= numarul de biti a tipului intreg (e.g., byte, short, int, and float in C)

Exemplu:

Cheile sunt CNP – au 13 cifre; numar de biti pentru int – 32 sau 64 depinde de sistem

S	AA	LL	ZZ	JJ	NNN	C

Cod numeric personal

Functia de dispersie: hashcode

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- Acumulare polinomiala:
- Se foloseste (mai mult) pentru stringuri:
  - partitionam bitii cheii in blocuri de dimensiune egala (8, 16..),

$a_0\, a_1 \ldots\, a_{n-1}$

  - *Evaluam polinomul in punctul x, ignoram overflow*
    - $h_1(a_0, a_1 \ldots, a_{n-1}) = a_0 + a_1 x + a_2 x^2 + \ldots\ldots + a_{n-1} x^{n-1}$
  - alegerea lui x influenteaza calitatea codului de compresie (numere prime !)
  - e.g. stringuri: x=33, cel mult 6 coliziuni la 50000 cuvinte

  - Exemplu:
    Un string este un sir de caractere. Fiecare caracter are un cod ASCII.
  - SALUT: S – 83; A – 65; L – 76; U – 85; T – 84 (coduri ASCII)
  - Fiind dat un string s = "$a_0\, a_1 \ldots\, a_{n-1}$",
  - $h(s) = a_0 + a_1 x + a_2 x^2 + \ldots\ldots + a_{n-1} x^{n-1}$
  - $H(SALUT) = 83 + 65*x + 76 * x^2 + 85 *x^3 + 84 * x^4$

Functia de dispersie: hashcode

- Rotatii ciclice:
  - $h_1(a_0,a_1 \ldots ,a_{n-1}) = rotate(x_{n-1} + rotate(x_{n-2} + \ldots (x_1 + rotate\ (x_0)..))$
  - alegerea dimensiunii rotatiei influenteaza calitatea codului

  - Exemplu:
  - hashcode polinomial care foloseste operatia de shift pentru a efectua o inmultire !
  - $h(s) = a_0 + a_1 << x + a_2 << x^2 + \ldots\ldots + a_{n-1} << x^{n-1}$

Implem. pt. x = 2 din acumularea polinomiala poate fi obtinuta si:

```
int hashCode(char s[], int length)
  {
     int r = 0, i;
     char c;
     for (i = 0; i < length; i++)
     {
       c = s[i];         // caracterul de pe pozitia i
       r = (int) c + (r << 1);   // shift la stanga = inmultire cu 2
     }
     return(r);
  }
```

Functii de dispersie - compresia

- Metoda impartirii (diviziunii)
  - $h(k) = k \bmod m$
  - valoarea lui m - importanta
  - **nu** puteri a lui 2 deci m nu are forma 2^p; numar prim nu prea apropiat de 2^p (motivul - teoria numerelor)
- Metoda inmultirii
  - $h(k) = \lfloor m(kA \bmod 1) \rfloor$, $0<A<1$, *constanta*
  - $kA \bmod 1$ – partea fractionala a lui kA (kA-$\lfloor kA \rfloor$)
  - valoarea lui m - nu e critica ($m=2^p$, ratiuni de implementare)
- Dispersie universala (see Cormen 11.3)

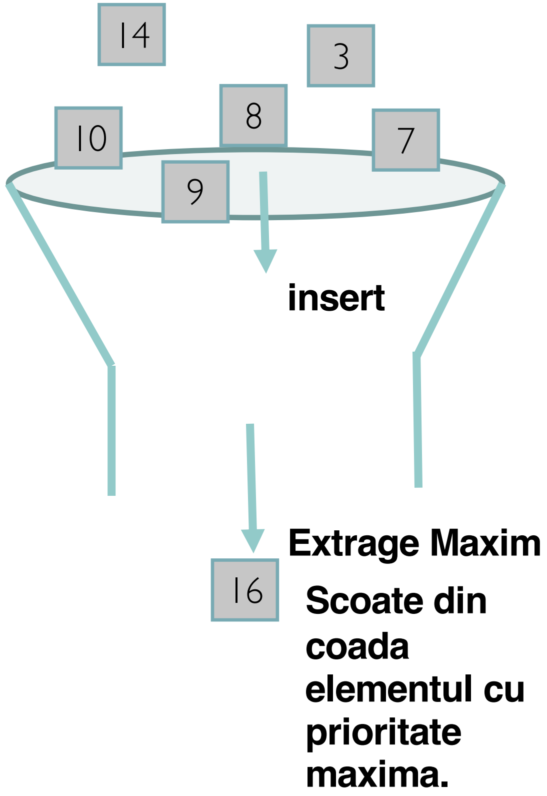
Tabele de dispersie - discutie

- Cazul defavorabil: toate operatiile - O(n)
  - toate cheile inserate produc coliziuni
- Factorul de umplere $\alpha = n/m$ afecteaza performanta
- Presupunand ca valorile hash sunt numere aleatoare, se poate demonstra ca numarul asteptat de probari la inserare/cautare cheie negasita pt. adresarea deschisa este: $1/(1-\alpha)$
- Similar, cautare: cheie gasita: $1/\alpha * \ln(1/(1-\alpha))$
- Timpul **mediu** al operatiilor: **O(1)**
- In practica, tabelele de dispersie sunt extrem de eficiente atata timp cat tabela nu este 100% plina

Coada de prioritati si structura de date Heap

Coada de prioritati

- Coada – mecanism standard folosit pentru sarcini de ordonare pe principiul primul sosit primul servit !
- Anumite sarcini (elemente din coada) pot fi mai importante decat altele – au o prioritate mai mare !
- Cozile de prioritate
  - Stocheaza elementele folosind o ordine partiala bazata pe prioritate
  - Asigura faptul ca elementul cu cea mai mare prioritate este in capul cozii (va fi primul care iese din coada !)
- Heap-urile sunt structurile de date care stau la baza cozilor de prioritate.



ADT: Coada de prioritati

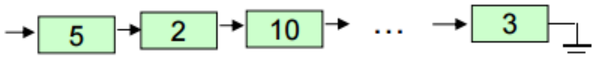
- Coada de prioritati: bazata pe modelul abstract de multime, cu operatiile:
  - insert
  - findMin
  - deleteMin
- O **inregistrare** intr-o coada de prioritati este o pereche (cheie, valoare)
- Cheile pot fi obiecte oarecare, pe care avem o ***relatie de ordine***
- Doua intrari diferite *pot* avea aceeasi cheie
- Diferenta fata de coada?

Relatia de ordine, comparator

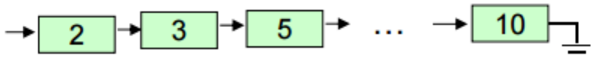
- Relatie de ordine totala ≤
  - Reflexivitate: $x \le x$
  - Anti-simetrie: $x \le y \wedge y \le x \Rightarrow x = y$
  - Tranzitivitate: $x \le y \wedge y \le z \Rightarrow x \le z$
- Un comparator incapsuleaza actiunea de a compara doua obiecte in concordanta cu o relatie de ordine:
  - O coada de prioritati generica utilizeaza un comparator auxiliar
  - Comparatorul este extern cheilor
  - Cand este nevoie sa se stabileasca relatia intre 2 chei, se utilizeaza comparatorul asociat cozii
- Operatie comparator:
  - compare(x, y): Returneaza un intreg $i < 0$ daca $a < b$, $i = 0$ daca $a = b$, si $i > 0$ daca $a > b$; un cod de eroare este emis daca a si b nu pot fi comparate

Cozi de prioritati: Implementari posibile LISTE

- *Lista nesortata*
- Performanta:
  - insert: ***O***(1) (putem insera la inceput/sfarsit)
  - deleteMin si findMin: ***O***(***n***) (cautare liniara, parcurgere lista)



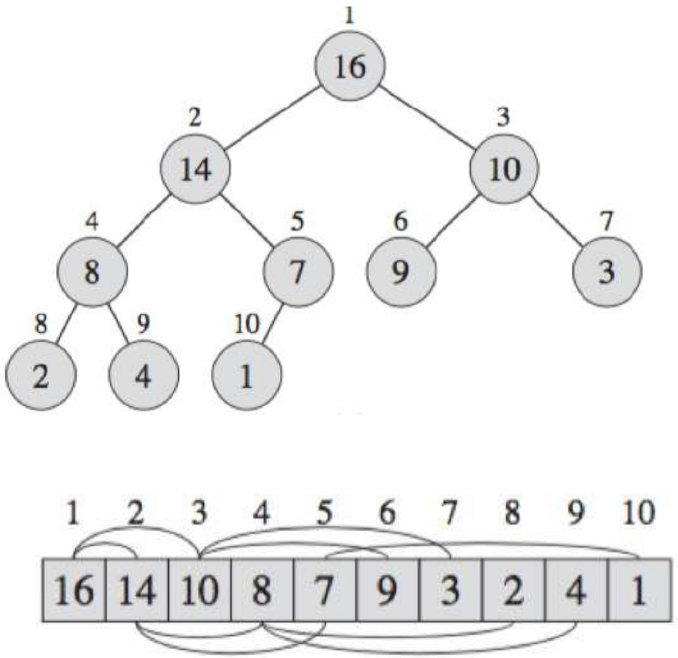
- *Lista sortata*
- Performanta:
  - insert: ***O***(n) (inserare la o locatie anume, conform relatiei de ordine)
  - deleteMin si findMin: ***O***(**1**) (elementul este la inceputul listei)



Cozi de prioritati: Implementari posibile

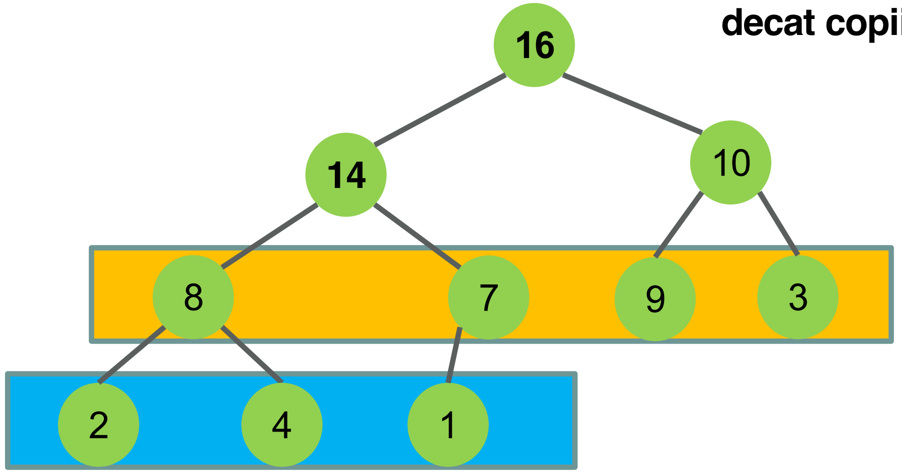
Arbori partial ordonati:

- Arbore binar
- Relatie de ordine partiala: **intre prioritatea nodului v si prioritatea copiilor lui v**
- pentru a avea h ~ log n se pot impune conditii aditionale:



Sursa: Th. Cormen, "Introduction to Algorithms", 3rd ed

HEAP BINAR STRUCTURA DE DATE PENTRU COADA DE PRIORITATE



Fiecare nod are valoare mai mare decat copiii sai !

Penultimul nivel este plin !

Ultimul nivel este umplut de la stanga la dreapta !

Heap Binar - definitii

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- Arbore binar cu doua proprietati:

1. Proprietate de structura
   - arbore complet: toate nivelele, mai putin (eventual) ultimul, sunt complet pline; nodurile de pe ultimul nivel sunt plasate de la stanga spre dreapta;
   - poate avea intre 1 si 2^h noduri la nivelul h
2. Proprietate de ordine:
   - **Max-Heap:** cheia din radacina este mai mare sau egala cu oricare din cheile copiilor, si sub-arborii cu radacinile in copii sunt si ei heap-uri

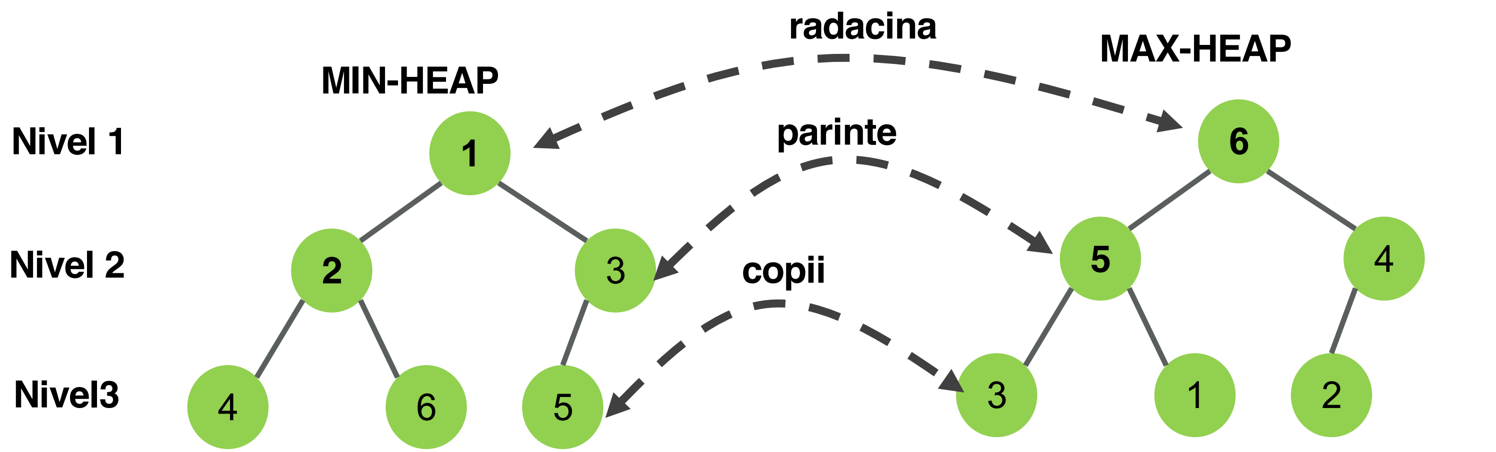
   SAU

   - **Min-Heap:** cheia din radacina este mai mica sau egala cu oricare din cheile copiilor, si sub-arborii cu radacinile in copii sunt si ei heap-uri

- Operatiile de inserare si stergere mentin cele doua proprietati.

Heap Binar

- In functie de relatia de ordine exista doua tipuri:
  - MIN-HEAP
  - MAX-HEAP

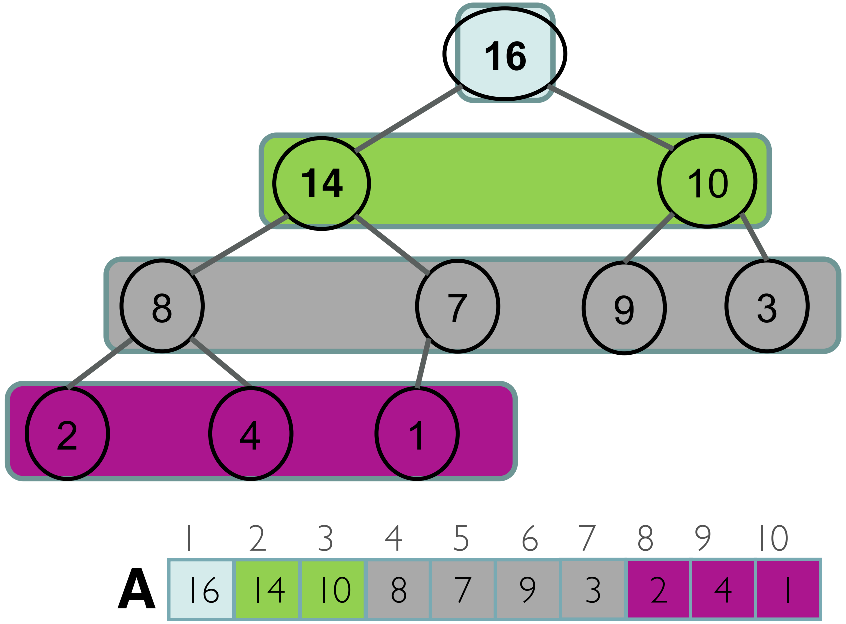


Parintele are valoare mai mica decat valoarea copiilor sai.

Parintele are valoare mai mare decat valoarea copiilor sai.

Cozi de prioritati: Max-Heap

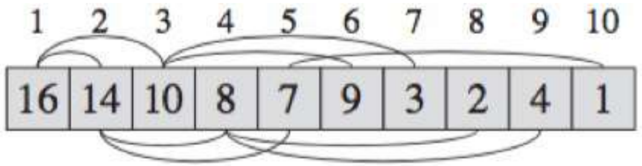
- Stocat ca si vector (desi il interpretam ca un arbore binar):
- nod la pozitia i in vector:
  - left child: 2*i
  - right child: 2*i+1
  - parent: i/2



PARENT(i)
return $\lfloor i/2 \rfloor$

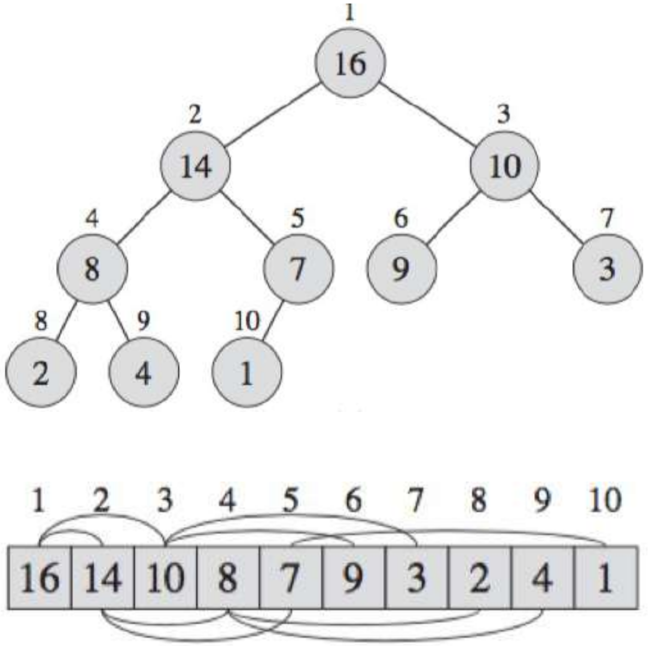
LEFT(i)
return $2i$

RIGHT(i)
return $2i+1$



Heap ca si coada de prioritati:

- Cea mai prioritara inregistrare se afla in radacina: max-heap
- Cea mai putin prioritara inregistrare se afla in radacina min-heap
  - De ce?
- insert: **O(log n)**
- extractMax: **O(log n)**
- findMax: **O(1)**



Sursa: Th. Cormen, "Introduction to Algorithms", 3rd ed

Heapify

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

Input: un vector A si un indice i din vector.
La fiecare pas se determina cel mai mare element dintre:

A[i], A[Left[i]], A[Right[i]]

Daca A[i] este cel mai mare atunci subarborele avand ca radacina
nodul i este un heap si procedura se termina.

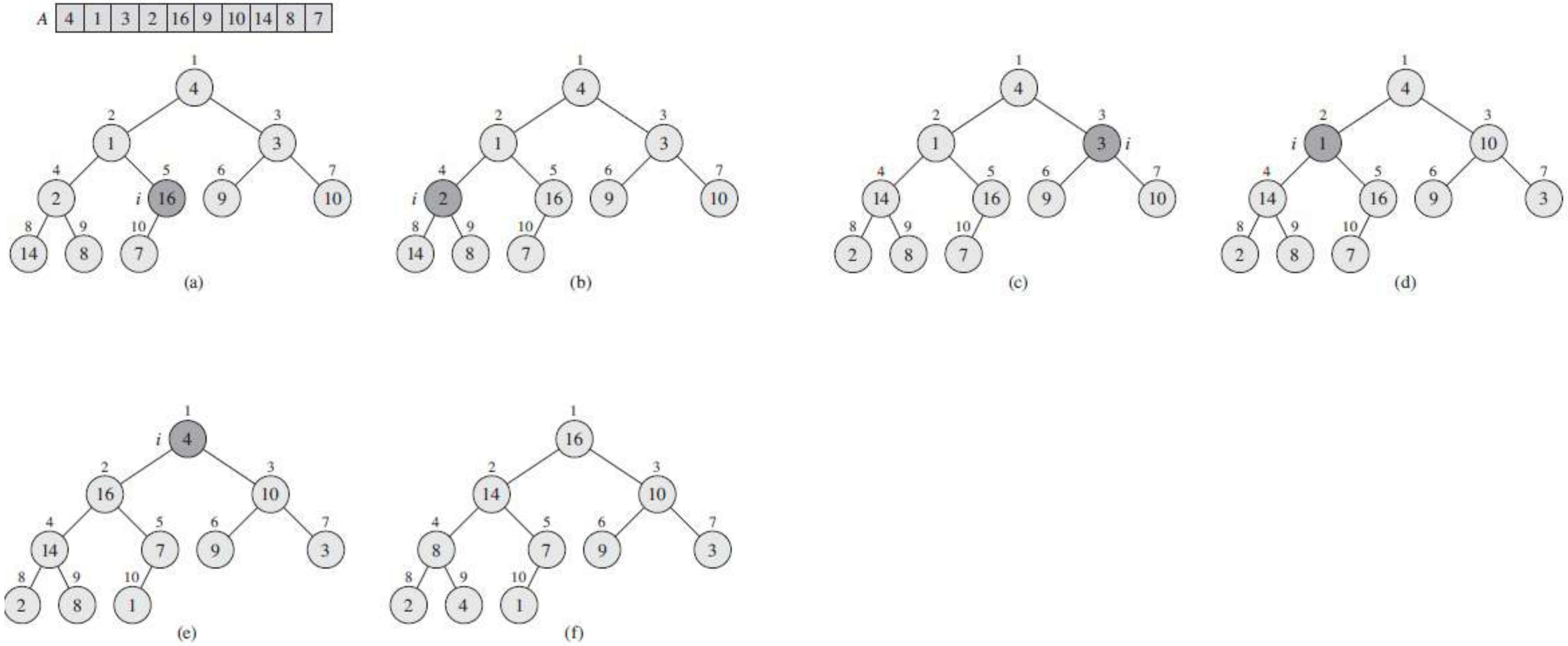
Altfel, cel mai mare element este unul dintre cei doi descendenti si
Este interschimbat cu A[largest].
Se apeleaza recusiv MAX-Heapify pentru indicele largest.

Heap - construire

- Construirea unui heap dintr-un vector A nesortat de lungime n.
- Toate elementele subsirului A[n/2+1 … n] sunt frunze – ele sunt heap-uri formate dintr-un singur element.
- BUILD-MAX-HEAP – traverseaza doar restul elementelor

```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

Exemplu – trasare pe pasi la tabla



Heap – extragerea elementului maxim

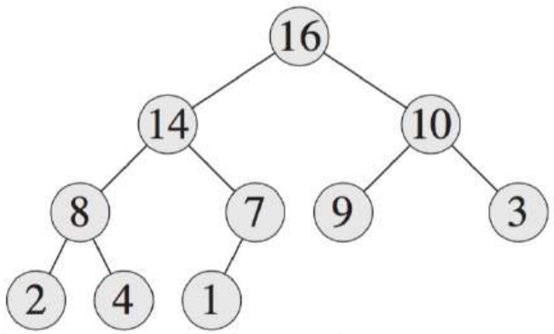
- **Elementul maxim –** este chiar primul element din vectorul in care este stocat heap-ul.
- Extragerea maximului – O(1)

1. Stergem nodul radacina
2. Mutam ultimul element de pe ultimul nivel in locul radacinii
3. Comparam valoarea noii radacini cu valoarea copiilor
4. Daca valoarea parintelui este mai mica decat a unui copil atunci interschimba.
5. Repeta pasii 3 si 4 pana cand proprietatea de heap este satisfacuta.

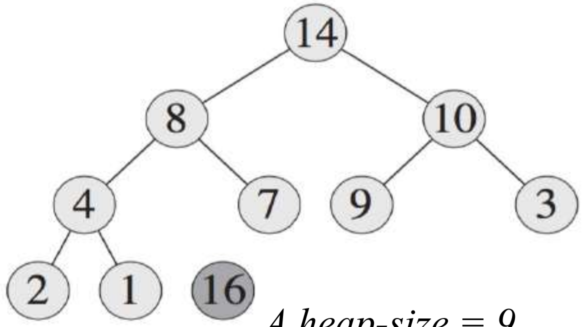
Heap – extragerea elementului maxim

- **Elementul maxim –** este chiar primul element din vectorul in care este stocat heap-ul.
- Extragerea maximului – O(1)

```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
      error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size − 1
MAX-HEAPIFY(A, 1)
return max
```



A.heap-size = 10

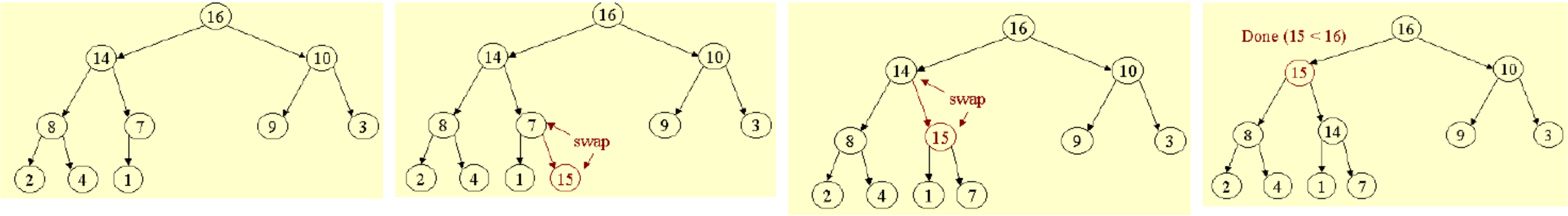


A.heap-size = 9

Heap - insert

```
MAX-HEAP-INSERT(A, key)
A.heap-size = A.heap-size + 1
A[A.heap-size] = −∞
HEAP-INCREASE-KEY(A, A.heap-size, key)
```

```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```



Bibliografie

- CLR, cap. 11(Hash Tables), cap. 7 (Heaps)
- visualgo.net