Structuri de date pentru multimi

Terminologie. Operatii. ADT-uri pentru multimi. Implementari. ADT Dictionar. Tabele cu acces direct. Tabele de dispersie.

Multimi matematice - terminologie

- **Definitie**: O multime este o colectie ordonata de obiecte ***distincte***
- Orice obiect din multime - element al multimii ($\in$, $\notin$- apartenenta)
- Cardinalitatea unei multimi |S| - numarul de elemente din S
- Multimea vida $\emptyset$, universul U (multimea care contine toate obiectele posibile)
- Egalitate: S=T iff

$$\forall x \in S \rightarrow x \in T \; and \; \forall y \in T \rightarrow y \in S$$

Multimi matematice - terminologie

- O <u>submultime</u> reprezinta o parte din multime.
  - $\subseteq$ - “submultime”
  - $\subset$ - “submultime proprie”
- *S* este submultime a lui *T* daca orice element al lui *T* apartine lui *S*
- Egalitate (revisited): *S = T iff if S* $\subseteq$ *T and T* $\subseteq$ *S*
- *S* este submultime <u>proprie</u> a lui *T* daca *S* este o submultime a lui *T* si $S \neq T$
  - $\varnothing$ este submultime proprie a oricarei multimi ne-vide
  - Orice multime ne-vida este propria ei submultime improprie
- Multimea <u>putere</u> $\wp(S)$ a unei multimi *S* este multimea care contine toate submultimile lui S
  - $|\wp(S)| = 2^{|S|}$

Multimi matematice - terminologie

- Cateodata este convenabil ca elementele multimii sa fie linear ordonate conform unei relatii de ordine, notata cu '<' ("mai mic", "precede").
- O ordine liniara pe multimea *S* are proprietatile:
  - Este definita pentru orice pereche de elemente:

    $$\forall a,b \in S, a < b \vee a = b \vee a > b$$

  - tranzitivitate:

    $$\forall a,b,c \in S, if\ a < b \wedge b < c \Rightarrow a > c$$

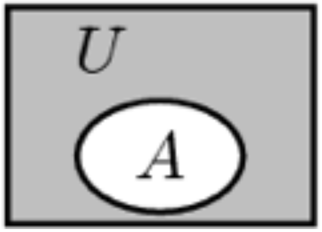
- Multiset sau bag - termeni utilizati pentru a referi "multimi cu duplicate"

Operatii pe multimi

- complement (‾) – “not”

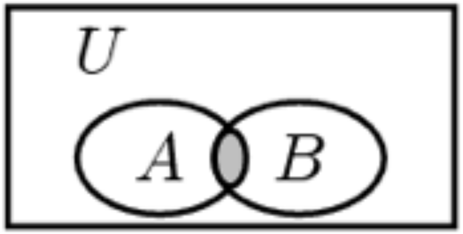
$$\overline{A} = \{x \in U : x \notin A\}$$

$$|\overline{A}| = |U| - |A|$$



- intersectie (∩) – “and”

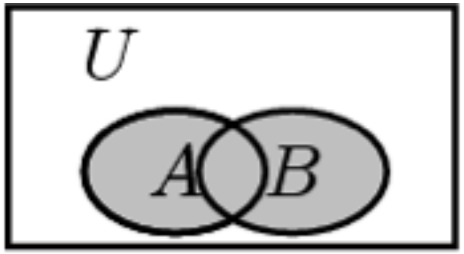
$$A \cap B = \{x \in U : x \in A \wedge x \in B\}$$



- reuniune (∪) – “or”

$$A \cup B = \{x \in U : x \in A \vee x \in B\}$$

$$|A \cup B| = |A| + |B| - |A \cap B|$$

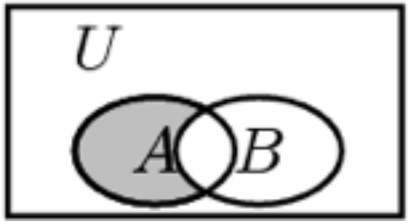


Operatii pe multimi

- diferenta (\,−) – “but not”

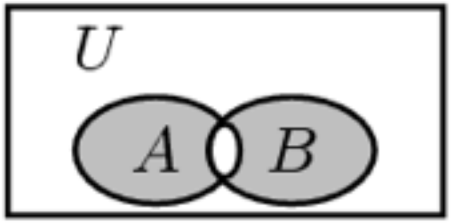
$$A \setminus B = \{x \in U : x \in A \wedge x \notin B\} = A \cap \overline{B}$$

$$|A \setminus B| = |A| - |A \cap B|$$



- diferenta simetrica (Δ ,⊕) – “exclusive or”

$$A \Delta B = (A \setminus B) \cup (B \setminus A)$$
$$= (A \cup B) - (A \cap B)$$



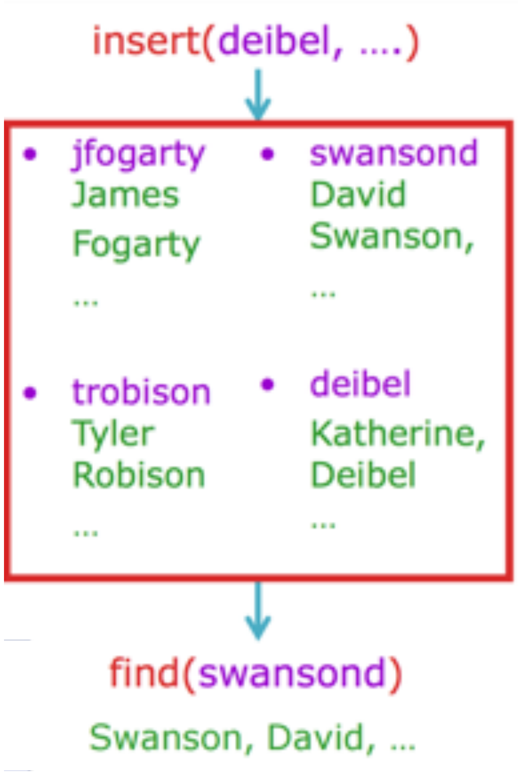
- Doua multimi A si B sunt disjuncte, daca A ∩ B = ∅

ADT Multime (Set)

- Focus pe stocare/cautare date!
- Data:
  - chei ***comparabile*** si ***unice***
- Operatii:
  - *insert(key)*
  - *find(key)*
  - *delete(key)*

ADT Dictionar (Dictionary)

- Data:
  - perechi <cheie, valoare>
  - cheile sunt asociate la valoare
  - chei comparabile si unice
- Operatii:
  - *insert(key, value)*
  - *find(key)*
  - *delete(key)*



Multime vs dictionar

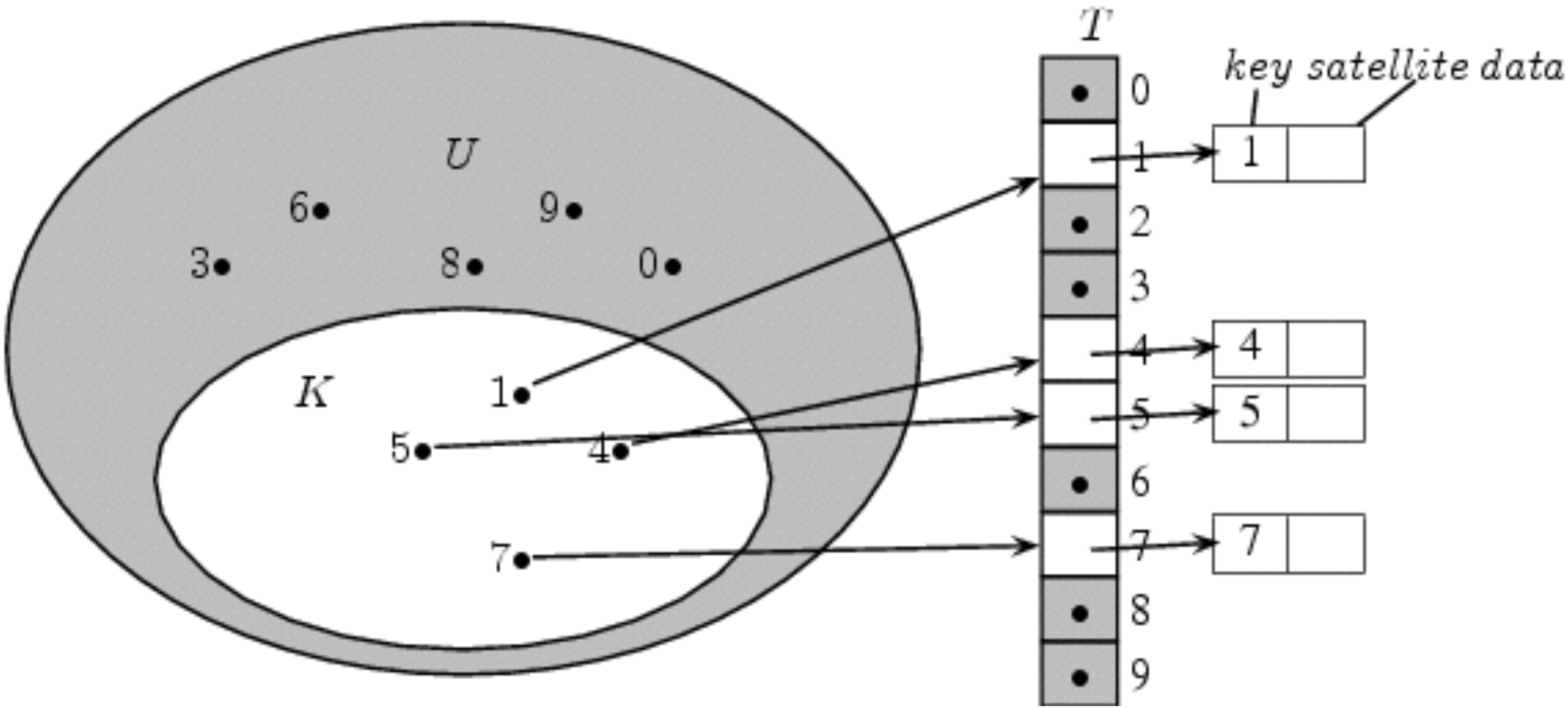
- In esenta identice
  - multimea nu are valori, doar chei
  - se pot utiliza aceleasi structuri pentru a le implementa
- Exceptie:
  - daca avem nevoie sa implementam operatii matematice pe multimi
    - reuniune, intersectie, …
    - optiuni mai bune pentru asa ceva, decat ceea ce se potriveste si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

Utilizari…

- Stocare de informatie conform unei chei, si accesul eficient la aceasta:
  - retele: tabele de rutare
  - sisteme de operare: tabele de paginare
  - compilatoare: tabele de simboluri
  - cautare in documente: *inverted index*, indexare
- Exemplu:
  - numararea aparitiilor unui cuvant in documente:
    - cheia: cuvantul; valoarea: numarul de aparitii
    - cand se prezinta un cuvant nou
      - `if !find(key) -> insert(key, value=1)`
      - `else value++`

Implementare: Tabela cu acces direct

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



Eficienta (timp)? Memorie?

Implementare: vector, liste inlantuite

	insert	find	delete
vector nesortat	O(1)	O(n)	O(n)*
lista simplu inlantuita nesortata	O(1)	O(n)	O(n)
vector 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 chei/valori)

* *Stergerea amanata in vectori sortati:*

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

Avantaje:
- delete in O(logn)
- eliminarea propriu-zisa in grup
- re-adaugarea se face prin modificarea marcajului

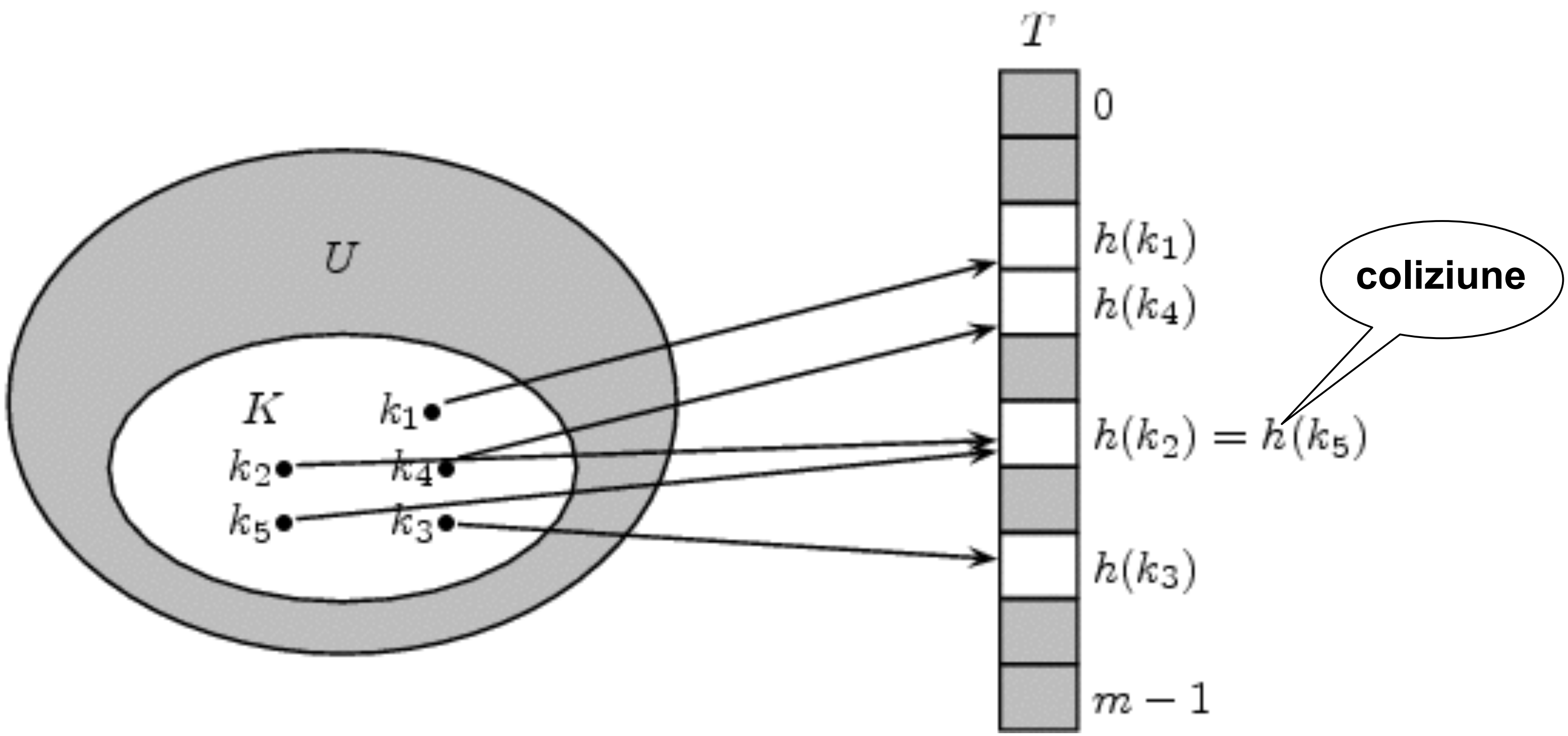
Dezavantaje:
- memorie aditionala liniara
- se iroseste memorie la stergeri multe
- cautarea O(log m), m - dimensiunea structurii
- se pot complica restul operatiilor

Implementare: ABC, ABC echilibrati

		insert	find	delete
ABC	*average*	O(logn)	O(log n)	O(log n)
	worst	O(n)	O(n)	O(n)
ABC cu echilibrare (AVL, Red-Black)	*avg/ worst*	O(log n)	O(log n)	O(log n)

- ABC - cazul defavorabil - operatii liniare
- conditii de echilibrare:
  - numar de noduri
  - inaltime
  - inaltime frunze
- (*mai multe probabil in cursul urmator…*)

Implementare: Tabela de dispersie



- 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 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*
- Problema*: coliziunile!*

Tabela de dispersie

- 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

Tabela de dispersie: Chaining

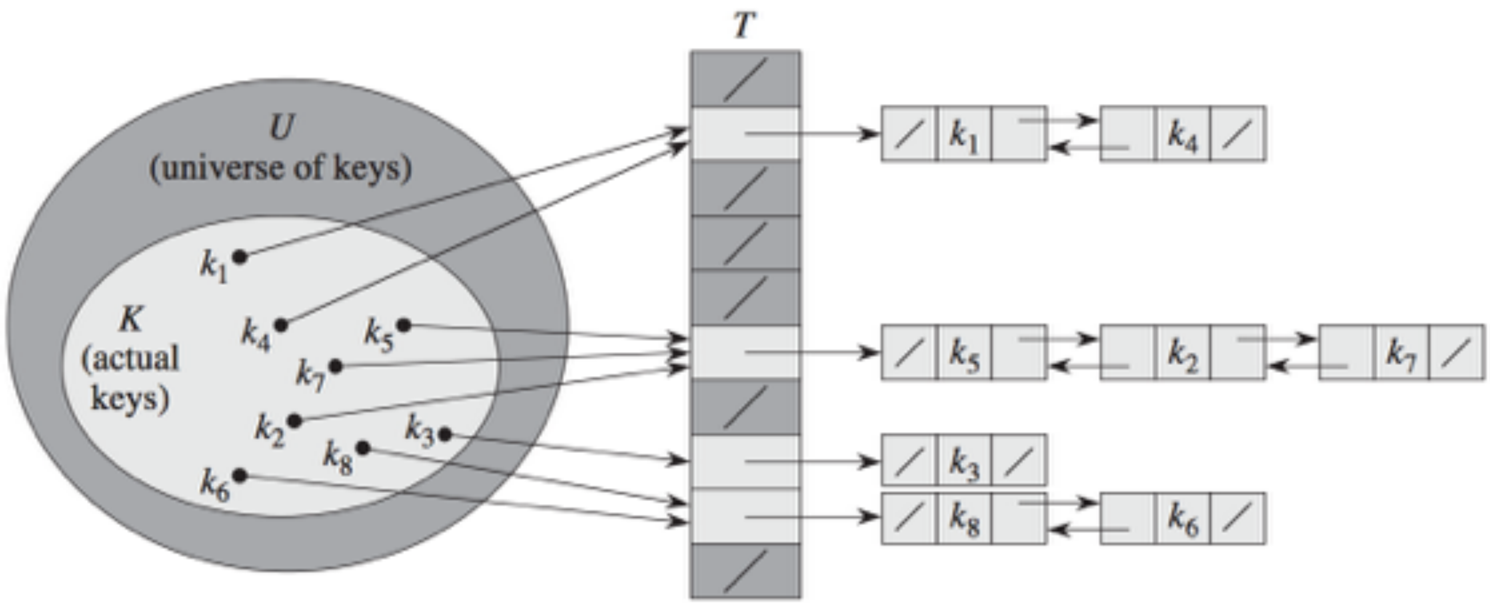
- toate elementele cu aceeasi valoare a functiei de dispersie sunt stocate intr-o lista inlantuita
- tabela de dispersie contine, la pozitia *j*, adresa primului element din lista cheilor (din tabela) care au *h(k)* = *j*
- Operatii:
  - *CHAINED-HASH-INSERT(T, x)*

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

    search for element having key k in list t[h(k)]
  - *CHAINED-HASH-DELETE(T, x)*

    delete x from the list T[h(x.key)]

Tabela de dispersie: Chaining



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

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

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 hash cheile in cele *m* sloturi posibile
  - **Presupunere**: *dispersie simpla, uniforma*
  - 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 parmutare

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

      return j
    i=i+1
  until T==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

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

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*
  - *quadratic probing*
  - *double hashing*

Tabele de dispersie: Adresare Deschisa

- Linear probing:
  - $h(k, i) = (h'(k)+i) \bmod m$, $h'(k)$ - functie de dispersie
  - 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 initial la adrese diferite pot concura pentru aceleași locații în iteratii succesive
- Quadratic probing:
  - $h(k, i) = (h'(k)+c_1 i+ c_1 i^2) \bmod m$
  - T[h'(k)], urmata de pozitii care depind cuadratic de i
  - mai bine decat linear probing, DAR! pt a genera tot spatiul de 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 initial 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 incremetate cu $h_1(k) \bmod m$ (*offset* variabil!)
  - $h_2(k)$ - prim fata de *m*, a.i. sa se sondeze intreaga tabela ($m = 2^p$ si $h_2(k)$ -> impar; sau m - prim, $0< h_2(k) < m$)
  - m^2 secvente diferite se folosesc, fata de *m*

Tabele de dispersie: Analiza Adresarii Deschise

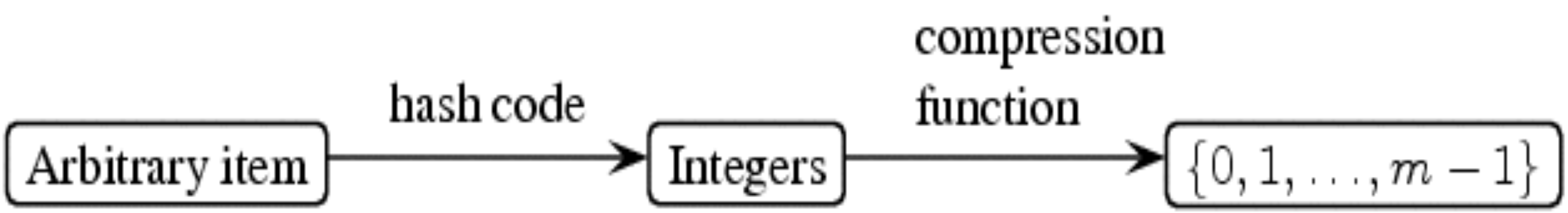
- Search (*demonstratie - see Cormen*):
  - cheie negasita:
    - $\mathbf{1/(1-\alpha)}$
  - cheie gasita:
    - $\mathbf{1/\alpha * \mathit{ln}(1/(1-\alpha))}$
- Insert?

Functia de dispersie

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

    h_1: chei → intregi
  - ***Functie de compresie***:

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



Functia de dispersie - contd.

- O functie **buna** de dispersie:
  - satisface pp. de ***distribuire 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: codul de dispersie

- Adresa memorie:
  - Interpretam adresa de memorie a obiectului cheie ca si intreg
  - Solutie buna in general, mai putin pt. chei numerice si string
- Transformare 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)
- Acumulare polinomiala:
  - partitionam bitii cheii in blocuri de dimensiune egala (8, 16..)
  - $a_0\ a_1 \ldots\ a_{n-1}$
  - Se evalueaza polinomul in punctul x, ignorand *overflow*
    - $p(x) = a_0 + a_1 x + a_2 x^2 + \ldots\ldots + a_{n-1} x^{n-1}$
  - e.g. stringuri: x=33, cel mult 6 coliziuni la 50000 cuvinte

Functii de dispersie - contd.

- Metoda impartirii
  - $h(k) = k \bmod m$
  - valoarea lui m - importanta
  - **nu** 2^p; numar prim apropiat de 2^p (motivul - teoria numerelor)
- Metoda inmultirii
  - $h(k) = \lfloor m(kA \bmod 1) \rfloor$, *0<A<1, constanta*
  - valoarea lui m - nu e critica ($m=2^p$)
- Dispersie universala (see Cormen 11.3)

Tabele de dispersie - discutie

- Cazul defavorabil: toate operatiile - O(n)
  - toate cheile inserate produc coliziuni
- Factorul de umplere $\alpha = \mathbf{N}/m$ afecteaza performanta
- Presupunand ca valorile hash sunt numere aleatoare, se poate demonstra ca numarul asteptat de probari la inserare pt adresarea deschisa este $\mathbf{1/(1-\alpha)}$
- Similar, cautare: cheie gasita: 1/α****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

Animatie tabele de dispersie

- http://visualgo.net/hashtable.html

Coada de prioritati (Tip de Abstract de Date)

- Coada de prioritati: bazat pe modelul abstract de multime, cu operatiile:
  - insert
  - findMin
  - deleteMin
- Cheile pot fi obiecte oarecare, pe care avem o relatie de ordine
- Doua intrari diferite pot avea aceeasi cheie

Inregistrari in cozi de prioritati

- Intrare in coada de prioritati: O **inregistrare** intr-o coada de prioritati este o pereche (cheie, valoare)
- Cozile de prioritati stocheza aceste inregistrari si permit inserari/stergeri eficiente bazate pe chei
- Operatii pe inregistrari:
  - key(): returneaza cheia unei inregistrari
  - value(): returneaza valoarea asociata unei intrari

Relatia de ordine - comparator

- Relatie de ordina totala ≤
  - Reflexivitate: $x \leq x$
  - Anti-simetrie: $x \leq y \wedge y \leq x \Rightarrow x = y$
  - Tranzitivitate: $x \leq y \wedge y \leq z \Rightarrow x \leq z$
- Un <u>comparator</u> 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:
  - <u>compare</u>(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

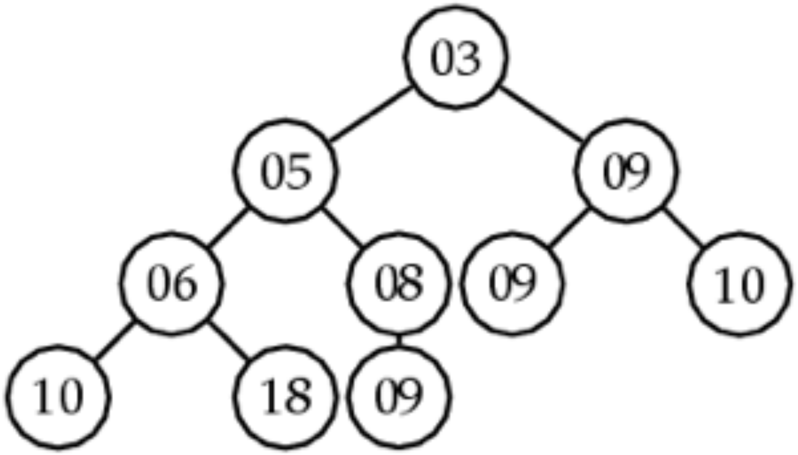
LISTE

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

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

Cozi de prioritati: Implementari posibile

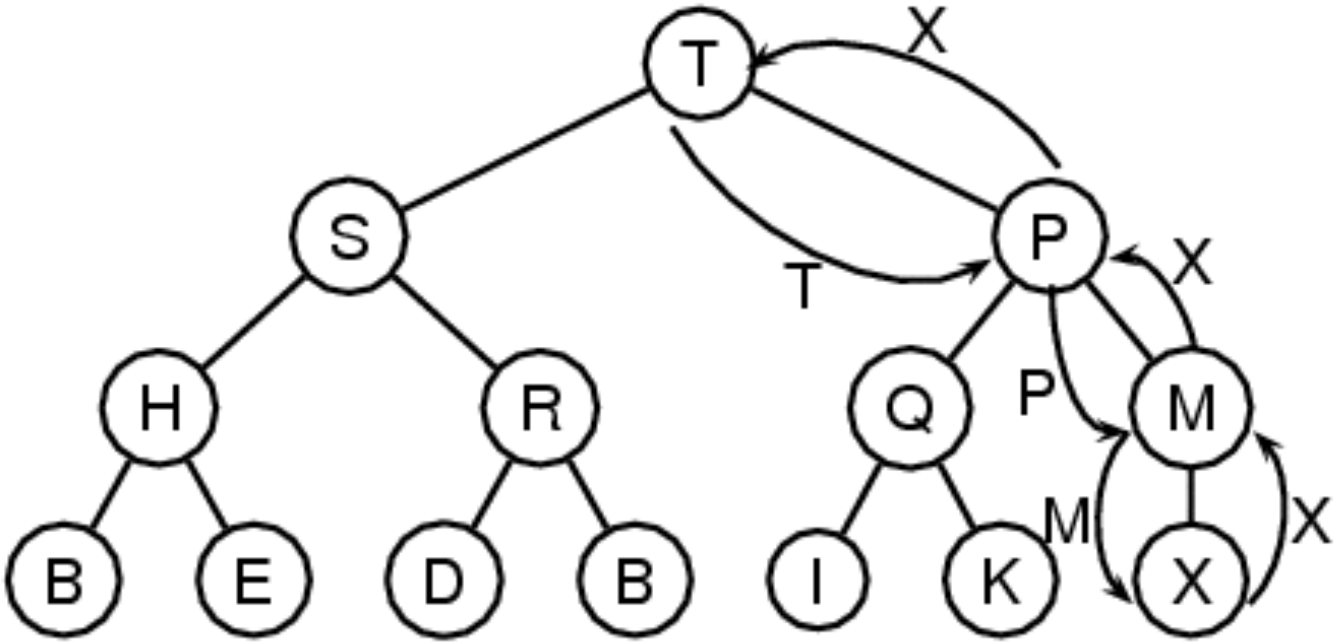
- **Arbori partial ordonati**:
  - Arbore binar
  - *Relatie de ordine partiala*: intre prioritatea nodului v si prioritatea copiilor lui v
  - pentru a avea *h ~ log n* se pot impune conditii aditionale:
    - **arbore complet**: toate nivelele, mai putin (eventual) ultimul, sunt complet pline; nodurile de pe ultimul nivel sunt plasate de la stanga spre dreapta;poate avea intre 1 si 2^h noduri la nivelul h



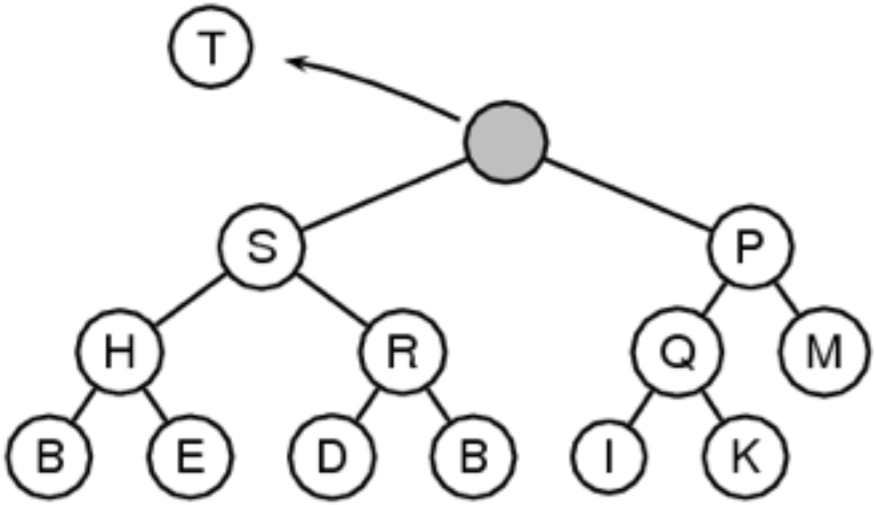
Cozi de prioritati: Heap

- Arbore binar complet, relatia de 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
- Stocat ca si vector:
  - if node - position i:
    - left child: 2*i+1
    - right child: 2*i+2
    - parent: i/2
- Heap ca si coada de prioritati:
  - cea mai (putin) ptioritara inregistrare se afla in radacina: max-heap (min-heap)
  - De ce?
    - insert: $O(\log n)$
    - extractMin: $O(\log n)$
    - findMin: $O(1)$

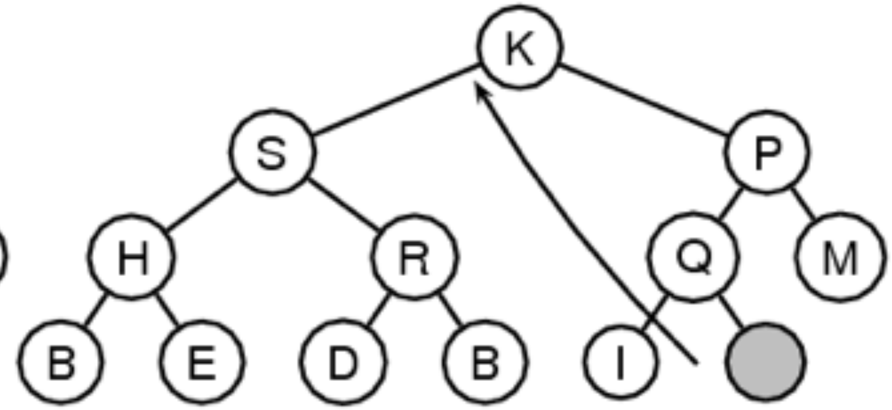
Heap: insert



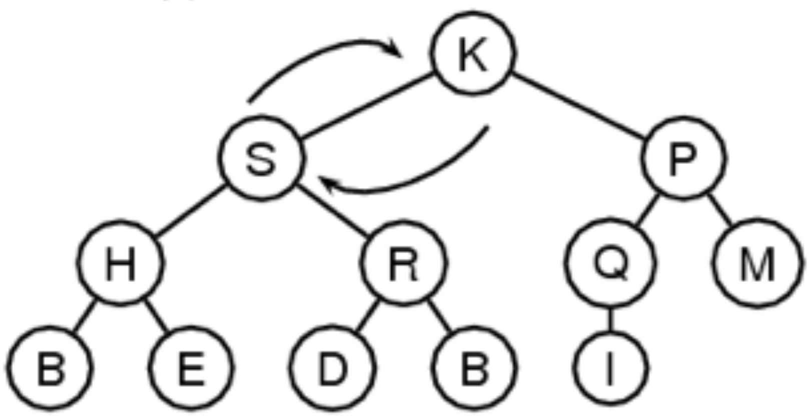
Heap: extract-max



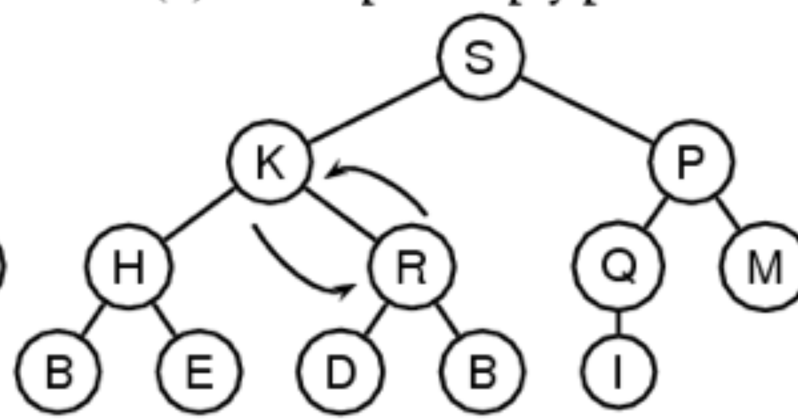
(a) Deletion of node T.



(b) K occupies empty position.



(c) Interchange K with the larger of its children.



(d) Interchange K with the larger of its children.

Heap operations

```
HeapExtractMax(A)
if heapSize[A] < 1
    then error "heap underflow"
max ← A[1]
A[1] ← A[heapSize[A]]
heapSize[A] ← heapSize[A] − 1
return max
```

```
HeapInsert(A, key)
heapSize[A] ← heapSize[A] + 1
i ← heapSize[A]
while i > 1 ∧ A[Parent(i)] < key
    do A[i] ← A[Parent(i)]
       i ← Parent(i)
A[i] ← key
```

Bibliografie

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