

Curs 7 - Tehnici de căutare

Introducere

Formularea problemei: se presupune că este dată o colecție de date, în care se cere să se localizeze (fixeze) prin căutare un anumit element.

Algoritmii de căutare pot fi împărțiți în trei mari categorii

- Secvențiali
- Cu acces direct bazat pe valoarea cheii (hashing)
- Bazați pe arbori de indexare (nu fac obiectivul acestui curs)

Tehnici de căutare în tablouri

Se consideră că mulțimea celor n elemente este organizată în forma unui un tablou liniar:

```
tipElement a[n];
```

Sarcina căutării este aceea de a găsi un element al tabloului a, a cărui cheie este identică cu o cheie dată x .

Indexul i care rezultă din procesul de căutare satisface relația $a[i].cheie == x$ și el permite accesul și la alte câmpuri ale elementului localizat

Se presupune în continuare că tipElement constă numai din câmpul cheie, adică el este chiar cheia.

Cu alte cuvinte, în continuare, se va opera de fapt cu un tablou de chei

Căutarea liniara

-> cand nu avem informatii referitoare la colectia de date in care se face căutarea : se parurge in mod secvential colectia prin incrementarea indexului de căutare pas cu pas

//Această metodă se numește căutare liniară și este ilustrată în secvența următoare:

```
tipElement a[n];
tipElement x;
int i=0;
while((i < n - 1) && (a[i] != x)){
    i++;
}
if(a[i] != x){
    return -1;
}
```

```

else{
    return i;
}

```

Cautarea se finalizeaza la indeplinirea a doua conditii:

1. Elementul a fost gasit : $a[i] == x$;
2. Elementul nu a fost gasit dupa parcurgerea integrala a tabloului : $i == (n - 1)$ si $a[i] != x$;

- Invariantul buclei, care de fapt este conditia care trebuie indeplinită înaintea fiecărei incrementări a indexului i , este:

```
(0 <= i < n-1) && (a[i] != x) // ceea ce implică și (0 <= k < i : a[k] <> x)
```

- Acest invariant exprimă faptul că pentru valori ale lui $k < i$, nu există coincidență.

- Condiția de terminare a căutării este:

```
((i == n-1) OR (a[i] == x)) // am ajuns la ultimul element sau am găsit elementul
```

După cum se observă, în algoritmul anterior fiecare pas al algoritmului necesită evaluarea expresiei booleene (care este formată din doi factori) urmată de incrementarea indexului.

- Acest proces poate fi simplificat și consecință căutarea poate fi accelerată, prin simplificarea expresiei booleene.

- O soluție de a simplifica expresia este aceea de a găsi un singur factor care să-i implice pe cei doi existenți.

- Acest lucru este posibil dacă se garantează faptul că va fi găsită cel puțin o potrivire.

- În acest scop tabloul a se completează cu un element adițional $a[n]$ căruia î se atribuie

initial valoarea lui x (elementul căutat). Tabloul devine:

tipElement $a[n+1]$

Elementul x poziționat pe ultima poziție a tabloului se numește fanion, iar tehnica de căutare, tehnica fanionului.

- Evident, dacă la părăsirea buclei de căutare $i = n$, rezultă că elementul căutat nu se află în tablou.

- Algoritmul de căutare astfel modificat:

```

tipElement a[n+1];
tipElement x;
int i=0;
a[n]=x;
while (a[i] != x)
    i++;
if (i==n){
    /*în tablou nu există elementul căutat*/
    return -1;
}
else{

```

```

        /*avem o coincidență la indicele i*/
        return i;
}

```

Condiția de terminare se reduce la ($a[i] == x$)

Pentru cazul cel mai defavorabil avem $O(n)$ - avem o parcurgere a întregului tablou

```

while ((i<n-1) && (a[i]!=x)) //Numărul de comparații este 2*n-1 +1
    i++;
if (a[i]!=x) {return -1; }
else {return i; }

```

În practică, în medie un element este găsit după parcurgerea a jumătate din elementele tabloului.

!! Cel mai defavorabil caz $O(n)$

Căutarea binară

- Procesul de căutare poate fi mult accelerat dacă se dispune de informații suplimentare referitoare la datele căutate.
- Este bine cunoscut faptul că o căutare se realizează mult mai rapid într-un masiv de date ordonate (spre exemplu într-o carte de telefoane sau într-un dicționar).
- În continuare se prezintă un algoritm de căutare într-o structură de date ordonată, în cazul de față un tablou care satisface predicatul A_k :

```

/* Definirea unui tablou ordonat*/
tipElement a[n];
Ak: 0<k<=n-1 : a[k-1]<=a[k]

```

!! Tabloul trebuie să fie sortat

Ideea de bază a căutării binare:

- Se inspectează un element aleator $a[m]$ și se compară cu elementul căutat x .
- Dacă $a[m]$ este egal cu x căutarea se termină.
- Dacă $a[m]$ este mai mic decât x , se restrâng intervalul de căutare la elementele care au indicii mai mari ca m .
- Dacă $a[m]$ este mai mare decât x , se restrâng intervalul la elementele care au indicii mai mici ca m .
- În consecință rezultă algoritmul denumit **căutare binară**.

```

// O variantă de implementare în C:
/*cautare binară */
int a[n];
/*n>0*/
int x;
tip_indice stanga,dreapta,m;
int gasit;

```

```

stanga=0;
dreapta=n-1;
gasit=0;
while((stanga<=dreapta)&&(!gasit)) {
    m=(stanga+dreapta)/2;
    /*sau orice valoare cuprinsă între stanga și dreapta*/
    if(a[m]==x) {
        gasit=1;
    }
    else if(a[m] < x){
        stanga = m + 1;
    }
    else{
        dreapta = m - 1;
    }
}

if(gasit){
    //avem o coincidență la indicele m
    return m;
}

```

Invariantul buclei, adică condiția ce trebuie îndeplinită înaintea fiecărui pas este:

(stanga<= dreapta) // ceea ce implică și ($\forall k: 0 \leq k < \text{stanga} : a[k] < x$)&& ($\forall k: \text{dreapta} < k \leq n-1 : a[k] > x$)

- **Condiția de terminare** este:

gasit OR (stanga>dreapta) // fie am găsit elementul, fie limitele intervalului în care facem căutarea au fost depășite

- Alegerea lui m este arbitrară în sensul că ea nu influențează corectitudinea execuției algoritmului, în schimb influențează eficiența sa.
- Soluția optimă în acest sens este alegerea elementului din mijlocul intervalului, întrucât ea elimină la fiecare pas jumătate din intervalul în care se face căutarea.
- În consecință rezultă că numărul maxim de pași de căutare va fi $O(\log_2 n)$.
- Această performanță reprezintă o îmbunătățire remarcabilă față de căutarea liniară unde numărul mediu de căutări este în medie $n / 2$.

!! m se ia de obicei ca fiind jumătatea lungimii tabloului

- O altă îmbunătățire posibilă a metodei de căutare binară, constă în a determina cu mai multă precizie locul în care se face căutarea în intervalul curent, în loc de a selecta întotdeauna elementul de la mijloc.
- Astfel când se caută într-o structură ordonată, spre exemplu cartea de telefon, un nume începând cu A este căutat la începutul cărții, în schimb un nume care începe cu V este căutat spre sfârșitul acesteia.
- Această metodă, numită căutare prin interpolare, necesită o simplă modificare în raport cu căutarea binară.
- Astfel, în **căutarea binară**, noul loc în care se realizează accesul este întotdeauna mijlocul intervalului curent de căutare.
- Acest loc este determinat cu ajutorul relației de mai jos, unde stanga și dreapta sunt limitele intervalului de căutare:

$$m = \text{stanga} + \frac{1}{2} (\text{dreapta} - \text{stanga})$$

!! nu trebuie sa retinem formula pt interpolare

Următoarea evaluare a lui m, numită **căutare prin interpolare**, se consideră a fi mai performantă:

$$m = \text{stanga} + \frac{x - a[\text{stanga}]}{a[\text{dreapta}] - a[\text{stanga}]} * (\text{dreapta} - \text{stanga})$$

- Evaluarea ține cont de valorile efective ale cheilor care mărginesc intervalul, în stabilirea indicelui la care se face următoarea căutare
- Această estimare însă, presupune o distribuție uniformă a valorilor cheilor pe domeniul vizat.

```
/*cautare interpolare*/
int a[n];
/*n>0*/
int x;
tip_indice stanga,dreapta,m;
int gasit;
stanga=0;
dreapta=n-1;
gasit=0;
while((stanga<=dreapta)&&(!gasit) && (x >= a[stanga]) && (x <= a[dreapta]))
{
    /* aici este o modificare*/
    /* aici este modificarea principală fata de căutarea binară*/
    m= stanga+((x-a[stanga]).cheie)*(dreapta-stanga)/
(a[dreapta].cheie-a[stanga].cheie);
    if(a[m]==x)
        gasit=1;
    else if(a[m] < x)
        stanga = m + 1;
    else
        dreapta = m - 1;
}
if(gasit){
```

```

    return m;
}

```

Tabele

TDA Tabelă

- Notiunea de tabelă este foarte cunoscută, fiecare dintre noi consultând tabele cu diverse ocazii.
- Ceea ce interesează însă în acest context se referă la evidențierea acelor caracteristici ale tabelelor care definesc tipul de date abstract tabelă.

Nume Prenume	An	Medie
Antonescu Ion	3	7,89
Bărulescu Petre	2	9,20
Card Gheorghe	5	9,80
Mare Vasile	2	8,33
Suciuc Horia	1	9,60

Nume Prenume	An	Medie
Suciuc Horia	1	9,60
Bărulescu Petre	2	9,20
Mare Vasile	2	8,33
Antonescu Ion	3	7,89
Card Gheorghe	5	9,80

După cum se observă, tabela anterioară, ca și marea majoritate a tabelelor, este alcătuită din articole.

- Acest lucru nu este însă obligatoriu, deoarece spre exemplu prima coloană a acestei tabele poate fi considerată la rândul ei o tabelă (tabela studenților înscriși).
- În prima variantă studenții sunt aranjați în ordine alfabetică, în cea de-a doua în ordinea crescătoare a anilor de studiu.

Se spune că tabela este ordonată după o "cheie" element care facilitează regăsirea informațiilor pe care le conține.

- Astfel, o "cheie" este un câmp sau o parte componentă a unui câmp care identifică în mod unic o intrare în tabelă.
- Nu este necesar ca tabela să fie sortată după o cheie însă o astfel de sortare simplifică căutarea în tabelă, în cazul căutărilor secvențiale, despre care am discutat în secțiunile precedente.
- Putem defini o tabelă de dispersie ca o structură de date care asociază o serie de atribute cu o cheie.

În definitiv, TDA Tabelă poate fi descris în mod sintetic după cum urmează:

TDA Tabelă

Modelul Matematic: O secvență finită de elemente. Fiecare element are cel puțin o cheie care identifică în mod unic intrarea în tabelă și care este un câmp sau un subcâmp al elementului.

Notății:

TipElement - tipul elementelor tabelei.

TipCheie - tipul cheii.

t: TipTabela;

e: TipElement;

k: TipCheie;

b: boolean.

Operatori:

1. **CreaZăTabelăVidă(t: TipTabelă);** - operator care face tabelă t vidă.
2. **TabelăVidă(t: TipTabelă): boolean;** - operator boolean care returnează true dacă tabelă t este goală.
3. **TabelăPlină(t: TipTabelă): boolean;** - operator boolean care returnează true dacă tabelă t este plină.
4. **CheieElemTabelă(e: TipElement): TipCheie;** - operator care returnează cheia elementului e.
5. **CautăCheie(t: TipTabelă, k: TipCheie): boolean;** - operator care returnează true dacă cheia k se găsește în tabelă t.
6. **InserElemTabelă(t: TipTabelă, e: TipElement);** - operator care inseră elementul e în tabelă t. Se presupune că e nu există în tabelă.
7. **SuprimElemTabelă(t: TipTabelă, k: TipCheie);** - operator care suprimă din t elementul cu cheia k. Se presupune că există un astfel de element în t.
8. **FurnizeazăElemTabelă(t: TipTabelă, k: TipCheie): TipElement;** - operator care returnează elementul cu cheia k. Se presupune că elementul aparține lui t.
9. **TraverseazăTabelă(t: TipTabelă, Vizitare (Listă Argumente))** – operator care execută procedura Vizitare pentru fiecare element al tabelei t

În anumite implementări este convenabil să se cunoască numărul curent de elemente conținute în tabelă.

- În acest scop se poate asocia tabelei un contor de elemente și un operator care-l furnizează, spre exemplu, DimensiuneTabelă(t:TipTabelă): TipContor.
- Contorul se initializează la crearea tabelei și se actualizează ori de câte ori se realizează inserții sau suprimări în tabelă.
- În unele aplicații este convenabil de asemenea să se definească operatorul Actualizare(t: TipTabelă, e: TipElement) .
- Operatorul realizează actualizarea acelei intrări din tabelă care conține elementul e.
- În termenii operatorilor anterior definiți, actualizarea poate fi implementată astfel:

{TDA Tabelă - operatorul Actualizare}
SuprimElemTabelă(t,CheieElemTabelă(e));
InserElemTabelă(t,e)

Tehnici de implementare a TDA Tabelă

- Există în principiu mai multe posibilități de implementare a tipului de date abstract tabelă.
- Este vorba despre tablouri, liste înláncuite, tabele de dispersie, arbori etc.

Implementarea TDA Tabelă cu ajutorul tablourilor ordonate

• Avantaje:

- Posibilitatea utilizării tehnicii căutării binare ($O(\log_2 n)$).
- Acest avantaj se răsfrângă asupra fazei de căutare a cheii în operațiile de inserție, suprimare, furnizare element și căutare cheie.
- Furnizarea informației și căutarea cheii este performantă $O(\log 2 n)$.
- De fapt furnizarea constă din doi timpi: faza de căutare ($O(\log 2 n)$) și din faza de returnare $O(1)$.
- Traversarea ordonată a cheilor este liniară ($O(n)$)

! furnizare și căutare : $O(\log 2 n) + \text{căutare binară}$

Traversarea ordonată a cheilor este liniară $O(n)$

Dezavantaje:

- Faza de instalare la inserție este lentă ($O(n)$).
- Necesită mutarea tuturor intrărilor începând cu punctul de inserție pentru a face loc în tabelă.
- Inserția este lentă ($O(n)$).
- Conștă din faza de căutare care este rapidă ($O(\log_2 n)$) și din faza de instalare care realizează inserția propriu-zisă ($O(n)$).
- Faza de extragere la suprimare este lentă ($O(n)$).
- Necesită mutarea tuturor intrărilor începând cu locul de suprimare.
- Suprimarea este lentă ($O(n)$).
- Ea constă din faza de căutare ($O(\log_2 n)$) și din faza de extragere ($O(n)$).
- Crearea tabelei lentă ($O(n^2)$).
- Pentru fiecare din cele n elemente se realizează o inserție $O(n)$.
- Trebuie cunoscut apriori numărul total de intrări în tabelă.

Concluzii:

- Utilizarea acestei implementări a tabelelor este avantajoasă atunci când:
 - Se realizează multe consultări, verificări sau rapoarte asupra tabelei.
 - Numărul de inserții și suprimări este relativ redus.
 - Tabela este de mari dimensiuni.
 - Trebuie cunoscută cu bună precizie apriori dimensiunea maximă a tabelei

Implementarea tabelelor prin tehnica dispersiei, tabele numite și **tabele de dispersie**

- Una dintre metodele avansate de implementare a tabelelor o reprezintă tehnica dispersiei.
- Această tehnică cunoscută și sub denumirea de "hashing", reprezintă un exemplu de rezolvare elegantă și deosebit de eficientă a problemei căutării într-un context bine precizat.

Formularea problemei:

- Se dă o mulțime S de noduri identificate fiecare prin valoarea unei chei, organizate într-o structură tabelă.
- Pe mulțimea cheilor se consideră definită o relație de ordonare.
- Se cere ca tabela S să fie organizată de o asemenea manieră încât regăsirea unui nod cu o cheie precizată k , să necesite un efort cât mai redus.
- În ultimă instanță, accesul la un anumit nod presupune determinarea acelei intrări din tabelă, la care el este memorat.
- Astfel, problema formulată se reduce la găsirea unei asocieri specifice (H) a mulțimii cheilor (K) cu mulțimea intrărilor (A)

$$H: K \rightarrow A$$

Ideeia pe care se bazează tehnica dispersării este următoarea:

- Se rezervă static un volum constant de memorie pentru tabela dispersată, care conține nodurile cu care se lucrează.
- Tabela se implementează în forma unei structuri de date tablou T având drept elemente nodurile în cauză.
- Notând cu p numărul elementelor tabloului, indicele a care precizează un nod oarecare, ia valori între 0 și $p-1$.
- Se notează cu K mulțimea tuturor cheilor și cu k o cheie oarecare.
- Numărul cheilor este de obicei mult mai mare decât p . Un exemplu realist în acest sens este acela în care cheile reprezintă identificatori cu cel mult zece caractere, caz în care există aproximativ 10^{15} chei diferite în timp ce valoarea practică a lui p este 10^3

- Se consideră în aceste condiții funcția H , care definește o aplicație a lui K pe mulțimea tuturor indicilor, mulțime care se notează cu L ($L = \{0,1,2,\dots,p-1\}$)
 $a = H(k)$ unde $k \in K$ și $a \in L$
- Funcția de asociere H este de fapt o funcție de dispersie și ea permite ca pornind de la o cheie k să se determine indicele asociat a .
- Este evident că H nu este o funcție bijectivă deoarece numărul cheilor este mult mai mare decât numărul indicilor.
- Practic există o mulțime de chei (clasă) cărora le corespunde același indice a .
- Din acest motiv, o altă denumire uzuală sub care este cunoscută tehnica dispersării este aceea de transformare a cheilor ("key transformation"), întrucât cheile se transformă practic în indici de tablou.

Principiul metodei este următorul:

- Pentru înregistrarea unui nod cu cheia k :
 - (1) Se determină mai întâi indicele asociat $a = H(k)$.
 - (2) Se depune nodul la $T[a]$.
- Dacă în continuare apare o altă cheie k' care are același indice asociat $a = H(k') = H(k)$, atunci s-a ajuns la așa numita situație de **coliziune**.
- Pentru rezolvarea acestei situații trebuie adoptată o anumită strategie de rezolvare a coliziunii.
- La căutare se procedează similar:
 - (1) Dată fiind o cheie k , se determină $a = H(k)$.
 - (2) Se verifică dacă $T[a]$ este nodul căutat, adică dacă $T[a].cheie = k$.
 - (3) Dacă da, atunci s-a găsit nodul, în caz contrar s-a ajuns la coliziune

În concluzie, aplicarea tehnicii dispersiei presupune soluționarea a două probleme:

- (1) Definirea funcției de dispersie H .
- (2) Rezolvarea situațiilor de coliziune.

• Rezolvarea favorabilă a celor două probleme conduce la rezultate remarcabile de eficiente, cu toate că metoda prezintă și anumite dezavantaje, asupra cărora se va reveni.

Comparații ale complexității pentru cazul **mediu** al metodelor de implementare pentru tabele

Implementare	Căutare	Inserție	Suprimare
Tablou neordonat	$O(n)$	$O(n)$	$O(n)$
Tablou ordonat	$O(\log n)$	$O(n)$	$O(n)$
Listă neordonată	$O(n)$	$O(n)$	$O(n)$
Listă ordonată	$O(n)$	$O(n)$	$O(n)$
Tehnica dispersiei	$O(1)$	$O(1)$	$O(1)$

Tehnica dispersiei (Hashing)

În această secțiune vom discuta despre o abordare diferită pentru a căuta valori într-o tabelă, bazându-ne pe accesarea directă a elementelor pe baza valorilor cheilor căutate, astfel hashing se referă la o tehnică folosită pentru a stoca și a găsi informații cât mai repede posibil.

Tehinca hashing nu se adresează aplicațiilor în care cheile sunt duplicate, nici celor în care se dorește găsirea unor înregistrări a căror chei au valori cuprinse într-un interval dat, ci pentru identificarea strictă a acelor înregistrări a căror cheie are o valoare dată (unică).

Procesul prin care mapăm poziția unei înregistrări în tabelă pe baza valorilor cheilor și găsirea poziției înregistrării căutate efectuând operații doar asupra cheii se numește hashing.

Funcțiile care mapează pozițiile în tabelă ale elementelor pe baza valorilor cheilor se numesc funcții de dispersie, sau funcții hash.

Astfel, hashing se bazează pe două componente principale pe care le vom discuta în continuare:

- Tabele de dispersie
 - Funcții de dispersie (numite și funcții hash)
-

Metoda hashing în general lucrează cu valori ale cheilor care se întind pe un interval larg și le reține în tabele cu un număr relativ restrâns de locații.

Pentru a înțelege această tehnică mai bine, să pornim de la un exemplu:

Să spunem că vrem să reținem situația tuturor studenților din universitate (>1000). Dacă ne gândim la ziua de naștere (zi-lună, fără a reține anul), avem 366 de variante, din care unele se repetă, în mod evident.

Astfel putem crea o mapă studenții în funcție de ziua lor de naștere într-o tabelă cu 366 de locații, în acest mod ziua de naștere devine o funcție de mapare/dispersie (o funcție hash). În acest caz pentru a găsi un student este de ajuns să calculăm indexul în tabelă, bazat pe ziua de naștere, fără a parcurge întreaga tabelă. În funcție de implementare, vom trata în mod diferit situațiile în care mai mulți studenți au aceeași zi de naștere (situații numite de coliziune).

În cel mai defavorabil caz va trebui să parcurgem toate structurile (în acest caz înregistrările cu studenți), care au aceeași zi de naștere, dar nu întreaga tabelă.

Funcții de dispersie

După cum s-a văzut în slide-urile anterioare, aplicarea tehnicii dispersiei presupune soluționarea a două probleme:

- (1) Definirea funcției de dispersie H (funcția hash).
- (2) Rezolvarea situațiilor de coliziune.

• Rezolvarea favorabilă a celor două probleme conduce la rezultate remarcabil de eficiente, cu toate că metoda prezintă și anumite dezavantaje, asupra cărora se va reveni.

Funcția de dispersie are rolul de a transforma valoarea unei chei în valoarea indexului corespunzător elementului cu acea cheie.

Alegerea funcției de dispersie

- Funcția de dispersie trebuie:
- Pe de o parte să repartizeze cât mai uniform cheile pe mulțimea indecșilor deoarece în felul acesta se reduce probabilitatea coliziunilor.
- Pe de altă parte trebuie să fie ușor și rapid calculabilă.
- Proprietățile acestei funcții sunt approximate în literatura de specialitate de termenul "hashing" (amestec) motiv pentru care funcția se notează generic cu H fiind denumită funcție de amestec ("hash function").
- Din același motiv tehnica dispersiei se mai numește și tehnică hashing.
- În literatură se definesc diverse funcții de dispersie fiecare cu avantaje și dezavantaje specifice, activitate care conturează un domeniu de cercetare extrem de activ

În cele ce urmează se va prezenta o variantă simplă a unei astfel de funcții.

- Fie $ORD(k)$ o valoare întregă unică atașată cheii k , valoare care precizează numărul de ordine al cheii k în mulțimea ordonată a tuturor cheilor
- Funcția H se definește în aceste condiții astfel :

$$H(k) = ORD(k) \bmod p$$

- Această funcție care asigură distribuția cheilor pe mulțimea indecșilor $(0, p-1)$ stă la baza mai multor metode de repartizare.
- S-a demonstrat experimental că în vederea repartizării cât mai uniforme a cheilor pe mulțimea indecșilor este recomandabil ca p să fie un număr prim.
- Cazul în care p este o putere a lui 2 este extrem de favorabil din punctul de vedere al eficienței calculului funcției, dar foarte nefavorabil din punctul de vedere al coliziunilor, mai ales în cazul în care cheile sunt secvențe de caractere

$$!!! H(k) = Ord(k) \% p;$$

Tratarea situației de coliziune

- Prezența unei situații de coliziune presupune generarea unui nou indice în tabelă pornind de la cel anterior, numit indice secundar.
- Există două modalități principale de generare a indicilor secundari:
 - (1) O modalitate care presupune **un spațiu suplimentar** asociat tabelei și care prefigurează metoda **dispersiei deschise**.
 - (2) O altă modalitate care exploatează **doar spațiul de memorie alocat tabelei** și care prefigurează metoda **dispersiei închise**

Metoda dispersiei **deschise**

- Metoda dispersiei deschise presupune înlănțuirea într-o listă înlănțuită a tuturor nodurilor ale căror indici primari sunt identici, metodă care se mai numește și înlănțuire directă.
- Elementele acestei liste pot sau nu apartine tabloului inițial: în ultimul caz memoria necesară alocându-se din aşa numita "zonă de depășire" a tabelei.

Această metodă are **avantajul** că permite suprimarea elementelor din tabelă, operație care nu este posibilă în toate implementările.

- Dintre **dezavantaje** se evidențiază două:
 - Necesitatea menținerii unei liste secundare.
 - Prelungirea fiecărui nod cu un spațiu de memorie pentru pointerul (indexul) necesar înlănțuirii

Metoda dispersiei **închise**

- Metoda dispersiei închise utilizează, după cum s-a precizat anterior, strict spațiul de memorie alocat tabelei.
- În cazul unei coliziuni se realizează parcurgerea după o anumită regulă a tabloului dispersat T până la găsirea primului loc liber sau a cheii căutate.

Adresarea deschisă liniară

- Adresarea deschisă liniară prefigurează cea mai simplă metodă de parcurgere a tabelei în cadrul tehnicii dispersiei închise.
- Conform adresării deschise liniare, intrările tabelei se parcurg secvențial, tabela considerându-se **circulară**. Cu alte cuvinte $g(i)=i$, iar modalitatea de generare a intrărilor în tabelă este:

$$a_0 = H(k)$$

$$a_i = (a_0 + i) \bmod p, \quad i=1 \dots p-1$$

Adresarea deschisă pătratică

- Adresarea deschisă pătratică reprezintă o altă soluție de compromis relativ simplă și eficientă de parcurgere a tabelei în cazul metodei dispersiei închise, soluție care rezolvă unele dintre deficiențele adresării deschise liniare.
- În cazul adresării deschise pătratice funcția g are expresia $g(i)=i^2$, iar modalitatea de generare a intrărilor în tabelă este:
 $a_0 = H(k)$
 $a_i = (a_0 + i^2) \text{ MOD } p, i=1,2,3,\dots$

Adresarea dublă

- Adresarea dublă reprezintă o altă soluție de compromis relativ simplă și eficientă de parcurgere a tabelei în cazul metodei dispersiei închise
- În cazul adresării deschise funcția g are orice expresie diferită de h , iar modalitatea de generare a intrărilor în tabelă este:
 $g(k) \neq 0$ și $g \neq h$
- Prima dată incercăm $h(\text{cheie})$. Dacă locația este ocupată, încercăm $h(\text{cheie}) + g(\text{cheie}), h(\text{cheie}) + 2 * g(\text{cheie}), \dots$

Ex $p = 10$

Valorile cheilor

2, 4, 14, 20, 22, 32, 42

$H(k) = k \bmod 10$

Dispersie deschisă		Dispersie închisă, adresare deschisă liniară		Dispersie închisă, adresare deschisă patratice	
0	20			0	20
1				1	
2	2	22	32	42	
3				2	2
4	4	14		3	22
5				4	4
6				5	14
7				6	32
8				7	42
9				8	
				9	

1. Dispersie deschisa

folosim spatiu auxiliar in cazul coliziunilor

2. Dispersie inchisa, adresare deschisa liniara

folosim doar spatiul tabloului

in cazul coliziunilor, nr se adauga pe spatiul imediat liber dupa pozitia dorita

3. Dispersie inchisa, adresare deschisa patratice

folosim doar spatiul tabloului

$(a_0 + i^2)$

Daca depaseste dimensiunea tabloului, ne vom imagina tabloul ca pe un cerc

4. Adresare dubla

$g(k) = 1 + (k \bmod (m-1))$

Calculul pentru fiecare cheie:

1. ** Cheia2 ** :

- $h1(2) = 2 \bmod 10 = 2h_1(2) = 2 \bmod 10 = 2h1(2) = 2 \bmod 10 = 2$
- Pozitie : 2.

2. ** Cheia4 ** :

- $h1(4) = 4 \bmod 10 = 4h_1(4) = 4 \bmod 10 = 4h1(4) = 4 \bmod 10 = 4$
- Pozitie : 4.

3. ** Cheia14 ** :

- $h1(14) = 14 \bmod 10 = 4h_1(14) = 14 \bmod 10 = 4h1(14) = 14 \bmod 10 = 4$
- Coliziunelapoziția4.
- $h2(14) = 1 + (14 \bmod 9) = 1 + 5 = 6h_2(14) = 1 + (14 \bmod 9) = 1 + 5 = 6h2(14) = 1 + 5 = 6$
- $h(14, 1) = (4 + 1 \cdot 6) \bmod 10 = 10 \bmod 10 = 0h(14, 1) = (4 + 1 \cdot 6) \bmod 10 = 10 \bmod 10 = 0$
- Pozitie : 0.

4. ** Cheia20 ** :

- $h1(20) = 20 \bmod 10 = 0h_1(20) = 20 \bmod 10 = 0h1(20) = 20 \bmod 10 = 0$
- Pozitie : 0(ocupatădejude14).
- $h2(20) = 1 + (20 \bmod 9) = 1 + 2 = 3h_2(20) = 1 + (20 \bmod 9) = 1 + 2 = 3h2(20) = 1 + 2 = 3$
- $h(20, 1) = (0 + 1 \cdot 3) \bmod 10 = 3h(20, 1) = (0 + 1 \cdot 3) \bmod 10 = 3h(20, 1) = (0 + 1 \cdot 3)m$
- Pozitie : 3.

5. ** Cheia22 ** :

- $h1(22) = 22 \bmod 10 = 2h_1(22) = 22 \bmod 10 = 2h1(22) = 22 \bmod 10 = 2$
- Coliziunelapoziția2(ocupatăde2).
- $h2(22) = 1 + (22 \bmod 9) = 1 + 4 = 5h_2(22) = 1 + (22 \bmod 9) = 1 + 4 = 5h2(22) = 1 + 4 = 5$
- $h(22, 1) = (2 + 1 \cdot 5) \bmod 10 = 7h(22, 1) = (2 + 1 \cdot 5) \bmod 10 = 7h(22, 1) = (2 + 1 \cdot 5)m$
- Pozitie : 7.

6. ** Cheia32 ** :

- $h1(32) = 32 \bmod 10 = 2h_1(32) = 32 \bmod 10 = 2h1(32) = 32 \bmod 10 = 2$
- Coliziunelapoziția2și7.
- $h2(32) = 1 + (32 \bmod 9) = 1 + 5 = 6h_2(32) = 1 + (32 \bmod 9) = 1 + 5 = 6h2(32) = 1 + 5 = 6$
- $h(32, 1) = (2 + 1 \cdot 6) \bmod 10 = 8h(32, 1) = (2 + 1 \cdot 6) \bmod 10 = 8h(32, 1) = (2 + 1 \cdot 6)m$
- Pozitie : 8.

7. ** Cheia42 ** :

- $h1(42) = 42 \bmod 10 = 2h_1(42) = 42 \bmod 10 = 2h1(42) = 42 \bmod 10 = 2$
- Coliziunelapozițiile2, 7, și8.
- $h2(42) = 1 + (42 \bmod 9) = 1 + 6 = 7h_2(42) = 1 + (42 \bmod 9) = 1 + 6 = 7h2(42) = 1 + 6 = 7$
- $h(42, 1) = (2 + 1 \cdot 7) \bmod 10 = 9h(42, 1) = (2 + 1 \cdot 7) \bmod 10 = 9h(42, 1) = (2 + 1 \cdot 7)m$
- Pozitie : 9.

Index: 0 1 2 3 4 5 6 7 8 9

Cheie: 14 2 20 4 22 32 42

Şiruri de caractere

Marea majoritate a celor preocupați de activitatea de programare sunt familiarizați cu şirurile de caractere întrucât aproape toate limbajele de programare includ şirul sau caracterul printre elementele predefinite ale limbajului.

În continuare se va prezenta **tipul de date abstract şir** precizând și modalități majore de implementare.

Un şir este o colecție de caractere, spre exemplu "Structuri de date".

În toate limbajele de programare în care sunt definite şiruri, acestea au la bază tipul primitiv char, care în afara literelor și cifrelor cuprinde și o serie de alte caractere.

Se subliniază faptul că într-un şir, ordinea caracterelor contează. Astfel şirurile "CAL" și "LAC" deși conțin aceleași caractere sunt diferite.

Tipul de date abstract şir:

- Asemenei oricărui tip de date abstracte, definirea precisă a TDA Şir necesită:
 - (1) Descrierea modelului său matematic.
 - (2) Precizarea operatorilor care acționează asupra elementelor tipului.
- **Din punct de vedere matematic, elementele tipului de date abstract şir pot fi definite ca secvențe finite de caractere (c_1, c_2, \dots, c_n) unde c_i este de tip caracter, iar n precizează lungimea secvenței.**
- Cazul în care n este egal cu zero, desemnează şirul vid.
- În continuare se prezintă un posibil set de operatori care acționează asupra TDA Şir.
- Ca și în cazul altor structuri de date, există practic o libertate deplină în selectarea acestor operatori motiv pentru care setul prezentat are un caracter orientativ.

TDA Şir

- **Modelul matematic:** secvență finită de caractere.

- Notații:

- s, sub, u - siruri;
- c - valoare de tip caracter;
- b - valoare booleană;
- poz, lung - întregi pozitivi.

- **Operatori:**

- **CreazaSirVid(sir s)** – funcție ce creează şirul vid s ;
- **b=SirVid(sir s)** - funcție ce returnează true dacă şirul s este vid;
- **lung=LungSir(sir s)** - funcție care returnează numărul de caractere ale lui s ;
- **poz=PozitieSubsir(sir sub,s)** - funcție care returnează poziția la care subşirul sub apare prima dată în s .

Dacă sub nu e găsit în s , se returnează o valoare convențională (0/-1). Pozițiile caracterelor pot fi reprezentate de adresa lor în memorie sau de valoarea indexului, în funcție de implementare

- ConcatSir(sir u,s) – funcție care concatenează la sfârșitul lui u caracterele sirului s
- CopiazaSubsir(sir u,s, int poz,lung) – funcție care-l face pe u copia subșirului din s începând cu poziția poz, pe lungime lung sau până la sfârșitul lui s;
- StergeSir(sir s, int poz,lung) – funcție care șterge din s, începând cu poziția poz, subșirul de lung caractere. Dacă poz este invalid s rămâne nemodificat;
- InsereazaSir(sir sub,s, int poz) – funcție care inserează în s, începând de la poziția poz, sirul sub;
- c=FurnizeazaCarSir(sir s, int poz) - funcție care returnează caracterul din poziția poz a lui s.
- AdaugaCarSir(sir s, char c) – funcție care adaugă caracterul c la sfârșitul sirului s;
- StergeSubsir(sir sub,s, int poz) – funcție care șterge prima apariție a subșirului sub în sirul s și returnează poziția poz de ștergere. Dacă sub nu este găsit, s rămâne nemodificat
- StergeToateSubsir(sir s,sub) - șterge toate aparițiile lui sub în s.
- Operatorii definiți pentru un TDA - sir pot fi împărțiți în două categorii:
 - Operatori primitivi care reprezintă un set minimal de operații strict necesare în termenii cărora pot fi dezvoltăți operatorii nonprimitivi.
 - Operatori nonprimitivi care pot fi dezvoltăți din cei anteriori.
- Această divizare este oarecum formală deoarece, de obicei e mai ușor să definești un operator neprimitiv direct decât să-l definești în termenii unor operatori primitivi.
- Spre exemplu operatorul InsereazăSir poate fi definit cu ajutorul: CreazăSirVid, AdaugăCar și FurnizeazaCar
- Algoritmul este simplu: se construiește un sir de ieșire temporar (rezultat) căruia i se adaugă pe rând:
 - (1) Caracterele sirului sursă până la punctul de inserție.
 - (2) Toate caracterele sirului de inserat (subșir).
 - (3) Toate caracterele sirului sursă de după punctul de inserție. Sirul astfel construit înlocuiește sirul inițial
- Unul dintre mărele avantaje ale utilizării datelor abstracte este următorul:
 - În situația în care se găsește un algoritm mai performant pentru o anumită operație, se poate foarte simplu înlocui o implementare cu alta fără a afecta restul programului în condițiile păstrării nealterate a prototipului operatorului.
- Implementarea sirurilor cu ajutorul tablourilor
- Cea mai simplă implementare a TDA-sir, care însă **nu se mai folosește** în limbajele de programare moderne, se bazează pe două elemente:
 - (1) Un întreg reprezentând lungimea sirului.
 - (2) Un tablou de caractere care conține sirul propriu-zis.

În tablou caracterele pot fi păstrate ca atare sau într-o formă împachetată.

9	'C'	'A'	'R'	'A'	'C'	'T'	'E'	'R'	'E'	3	'S'	'D'	'A'
---	-----	-----	-----	-----	-----	-----	-----	-----	-----	---	-----	-----	-----

- Metode de implementare şiruri de caractere în C:

În limbajul C, termenul şir de caractere înseamnă un **tablou de caractere încheiat în memorie cu caracterul '\0'** (codul ASCII 0).

Tinând cont că în limbajul C, indecșii încep de la 0, lungimea şirului efectiv (fără '\0') coincide cu valoarea indexului pe care se găsește terminatorul de şir '\0'

Această metodă de implementare a şirurilor este mai eficientă din punct de vedere a gestionării memoriei față de cea prezentată anterior

```
char msg[] = "test"; // 5 octeti, terminat cu '\0'
char msg[] = {'t','e','s','t','\0'}; // același sir, scris altfel
char sir[20] = "test"; // restul pana la 20 sunt '\0'

'C' | 'A' | 'R' | 'A' | 'C' | 'T' | 'E' | 'R' | 'E' | '\0' | 'S' | 'D' | 'A' | '\0'
```

Pentru lucrul cu şiruri de caractere se pot utiliza o serie de funcții definite în biblioteca standard [string.h](#)

Unele din cele mai des folosite funcții:

- `size_t strlen(const char *s);` // returneaza lungimea sirului s
- `char *strchr(const char *s, int c);` // cauta caract. c in s
- `char *strstr(const char *big, const char *small);` // cauta sir
// ambele returneaza adresa unde e gasit sau NULL daca nu exista

Observații:

`size_t`: tip întreg fără semn pentru dimensiuni

`const`: specificator de tip: obiectul respectiv nu e modificat

- `char *strcpy(char *dest, const char *src);` // copie src in dest
- `char *strcat(char *dest, const char *src);` // concat src la dest
// ambele necesită destul loc la dest, ATENȚIE LA DEPĂȘIRE!
- `int strcmp (const char *s1, const char *s2);` // compară
// returneaza intreg < 0 sau 0 sau > 0 dupa cum e s1 fata de s2
- `char *strncpy(char *dest, const char *src, size_t n);`
- `char *strncat(char *dest, const char *src, size_t n);`
// copiază(concatenează cel mult n caractere din src la dest
- `int strncmp (const char *s1, const char *s2, size_t n);`
// compara sirurile pe lungime cel mult n caractere

Complexitatea operațiilor:

- Funcțiile CopiazăSubŞir, ConcatŞir, ŞtergeŞir și InsereazăŞir se execută într-un interval de timp liniar O(n), unde n este după caz lungimea subşirului sau a şirului de caractere.
- Accesul la elementele subşirului se realizează direct (`sir[pos], sir[pos+1],...,sir[pos+lung-1]`), astfel încât consumul de timp al execuției este dominat de mutarea caracterelor
- În continuare se definesc pentru o structură şir, în termeni de logica predicatelor, relația de coincidență respectiv relația de ordonare (lexicografică) a două şiruri x și y după cum urmează:
- Pentru a stabili coincidența, este necesară stabilirea egalității tuturor caracterelor corespunzătoare din şirurile comparate.

Şiruri egale ca valoare(coincidență = adevărat):

'A'	'B'	'C'	'D'	'\0'
'A'	'B'	'C'	'D'	'\0'

Şiruri diferite (coincidență = fals):

'A'	'B'	'C'	'D'	'\0'
'A'	'B'	'C'	'\0'	

```

int comparatie(char* sir1, char* sir2) {
    int i = 0;
    while ((sir1[i] == sir2[i]) && sir1[i] && sir2[i])
        i++;
    if (sir1[i]==sir2[i])
        /*sirurile sunt egale ca valoare – atenție comparăm valoare
elementelor nu adresa lor!*/
        if (sir1[i]>sir2[i])
            //primul sir este mai mare in ordine lexicografica
        if (sir1[i] < sir2[i])
            //al doilea sir este mai mare in ordine lexicografica
}

```

Tehnici de căutare în şiruri de caractere

- Una din operațiile cele mai frecvente care se execută asupra şirurilor este căutarea.
- Întrucât performanța acestei operații are o importanță covârșitoare asupra marii majorități a prelucrărilor care se realizează într-un sistem de calcul, studiul tehniciilor de căutare performante reprezintă o preocupare permanentă a cercetătorilor în domeniul programării.
- În cadrul acestui paragraf vor fi trecute în revistă câteva dintre cele mai cunoscute tehnici de căutare în şiruri de caractere.

Căutarea de şiruri directă

- O manieră frecvent întâlnită de căutare este aşa numita căutare de şiruri directă ("string search").

Specificarea problemei:

- Se dă un tablou s de n caractere și un tablou p de m caractere, unde $0 < m < n$
- Căutarea de şiruri directă are drept scop stabilirea primei apariții a lui p în s

De regulă, s poate fi privit ca un **text**, iar p ca un cuvânt **model** ("pattern") a cărui **primă apariție** se caută în textul s.

- Cea mai **simplă** metodă de căutare este aşa numita **căutare de şiruri directă** ("string search").
- **Rezultatul** unei astfel de căutări este un indice i care precizează apariția unei **coincidențe** între model și sir.

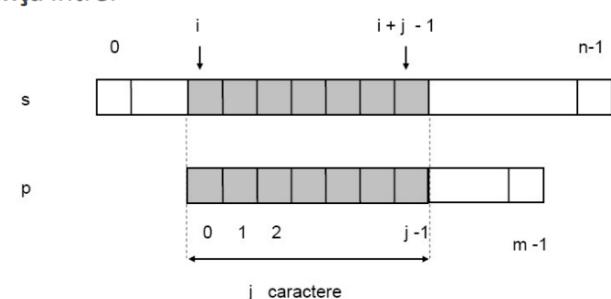
În cazul în care dorim să determinăm o **coincidență de doar j caractere**, rezultatul unei astfel de căutări este un indice i care precizează apariția unei **coincidențe** de lungime j caractere între model și sir.

- Acest lucru este formalizat de **predicatul $P(i,j)$**

$$P(i,j) \leq A_k : 0 \leq k < j : s_{i+k} == p_k$$

- **Predicatul $P(i,j)$** precizează faptul că există o **coincidență** între:

- Şirul s (începând cu indicele i).
- Şirul p (începând cu indicele 0).
- Coincidența se extinde pe o lungime de j caractere



Este evident că indexul i care rezultă din căutarea directă de siruri trebuie să satisfacă predicatul P(i,m).

- Această condiție nu este însă suficientă.
- Deoarece căutarea trebuie să furnizeze prima apariție a modelului, P(k,m) trebuie să fie fals pentru toți, indecșii anteriori ($k < i$) .
- Se notează această condiție cu Q(i)

$Q(i) = Ak: 0 \leq k < i: \sim P(k,m)$

- Predicatul specifică faptul că nu avem nicio potrivire la indecși mai mici decât i

Specificarea problemei sugerează implementarea **căutării directe de siruri** ca și o iterare de comparații redactată în **termenii predicatorilor Q** respectiv **P**.

- Astfel implementarea lui **Q(i)** conduce la secvența

{Căutarea de siruri directă - Implementarea predicatorului Q(i)}

i:= -1;

DO

i:= i+1;

gasit:= **P(i,m)** ;

WHILE NOT(gasit) AND (i=n-m);

```
int cautare_directa (char *sir, char *model) {  
    int i,j;  
    /*i parcurge caracterele din sir, j parcurge caracterele din model*/  
    i = 0;  
    do{  
        j = 0;  
        while ((sir[j+i] == model[j]) && sir[j+i] && model[j])  
            j++;  
        if (j==strlen(model))  
            return j+i-strlen(model);  
        //s-a gasit o coincidență  
        i++;  
    } while (i < strlen(sir));  
    return -1;  
} /*cautare_directa*/
```

Pentru cazul cel mai defavorabil este necesar un număr de comparații de ordinul **n * m**, unde $n=\text{strlen}(\text{sir})$, iar $m=\text{strlen}(\text{model})$, pentru a găsi coincidență la sfârșitul sirului.

```
!! Cazul cel mai defavorabil -> n * m  
n = (strlen(sir));  
m = strlen(model);
```

Cautarea de siruri Knuth - Morris - Pratt

Căutarea de şiruri Knuth-Morris-Pratt

- În anul 1970 Knuth, Morris și Pratt au inventat un algoritm de căutare în şiruri de caractere care necesită un **număr de comparații** de ordinul n chiar în cel mai **defavorabil** caz.
- Noul algoritm se bazează pe **observația** că avansul modelului în cadrul şirului cu o **singură** poziție la întâlnirea unei nepotriviri, aşa cum se realizează în cazul **căutării directe**, pe lângă o eficiență scăzută, conduce la **pierderea** unor informații utile.
- Astfel după o **potrivire parțială** a modelului cu şirul întrucât se cunoaște **parțial** şirul (până în punctul baleat), dacă avem cunoștințe **apriorice** asupra modelului obținute prin **precompilare**, le putem folosi pentru **a avansa mai rapid** în şir în procesul de căutare.

Utilizând predicatele **P** și **Q**, **algoritmul KMP** poate fi formulat astfel:

{Căutarea de şiruri Knuth-Morris-Pratt}

$i := 0; j := 0;$

WHILE ($j < m$) **AND** ($i < n$) **DO**

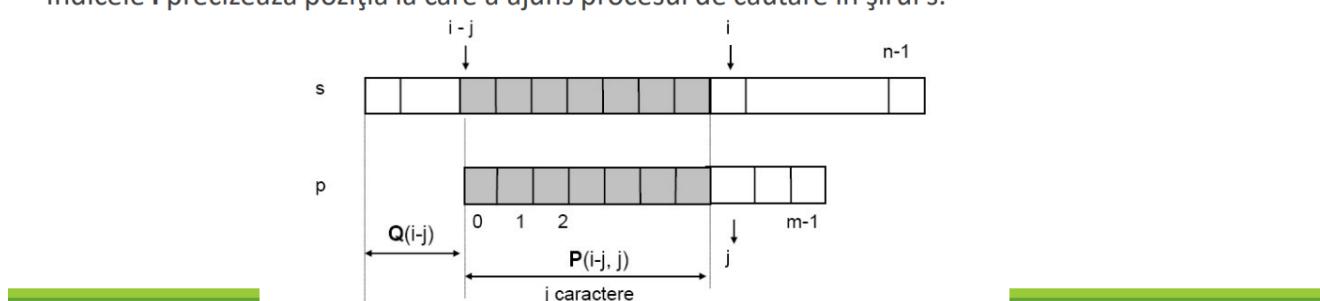
BEGIN { $Q(i-j) \ \&\& \ P(i-j, j)$ }

WHILE ($j \geq 0$) **AND** ($s[i] \neq p[j]$) **DO** $j := d;$

$i := i + 1; j := j + 1$

END; {WHILE}

- Din formularea algoritmului lipsește specificarea factorului d , care precizează valoarea deplasării.
- Se subliniază faptul că în continuare $Q(i-j)$ și $P(i-j, j)$ sunt invariante globale ai procesului de căutare, la care se adaugă relațiile $0 < i < n$ și $0 < j < m$. Adică i parcurge şirul, iar j modelul.
- Este important de subliniat faptul că, din rațiuni de claritate, predicalul P va fi ușor modificat: de aici înainte nu i va fi indicele care precizează poziția primului caracter al modelului în şir ci valoarea $i-j$.
- Indicele i precizează poziția la care a ajuns procesul de căutare în şirul s .



Pentru a determina d (valoarea de deplasare a modelului) se poate utiliza următorul algoritm:

- Pentru fiecare valoare a lui j din cadrul modelului:
 - Se caută în model cel mai mare d, adică cea mai lungă secvență de caractere a modelului care precede imediat poziția j și care se potrivește ca un număr egal de caractere de la începutul modelului.
 - Se notează valoarea d pentru un anumit j cu d_j
 - Întrucât valorile d_j depind numai și numai de model, înaintea căutării propriu-zise poate fi construit un tablou d cu aceste valori, printr-un proces de precompilare a modelului
 - Exemple și explicații suplimentare legate de construirea tabloului de deplasări se găsesc în materialele auxiliare de pe campusul virtual

Programul KMP constă din 4 părți:

[1] Prima parte realizează **citirea sirului** în care se face căutarea.

[2] A doua parte realizează **citirea modelului** p.

[3] A treia parte **precompilează modelul** și calculează valorile d_j .

[4] Cea de-a patra parte **implementează căutarea** propriu-zisă.

- Inventatorii demonstrează că **numărul de comparații** de caractere este de ordinul $n+m$.
- Aceasta reprezintă o îmbunătățire substanțială față de $m \cdot n$.

```
int d[256];
//tabloul de coeficienti k folositi in calculul deplasarilor
int KMP(char s[], char p[]) {
    int i, j, k;
    j = 0;
    k = -1;
    d[0] = -1;
    /*precompilare model*/
    unsigned int n = strlen(s);
    unsigned int m = strlen(p);
    while (j < m - 1) {
        while ((k >= 0) && (p[j] != p[k]))
            k = d[k];
        j = j + 1;
        k = k + 1;
        if (p[j] == p[k])
            d[j] = d[k];
        else
            d[j] = k;
    }
    //codul se continua
    //continuare cod
    /*cât timp*/
    i = 0;
    j = 0;
    /*căutare model*/
    while ((j < m) && (i < n)) {
        while ((j >= 0) && (s[i] != p[j]))
            j = d[j];
```

```

        j = j + 1;
        i = i + 1;
    }

    if (j == m)
        return i-m;
    else
        return -1;
/* KMP*/
}

```

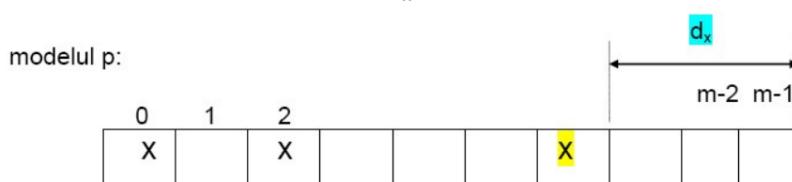
Căutarea de siruri Boyer - Moore

Căutarea de şiruri Boyer-Moore

- Metoda ingenioasă de căutare KMP conduce la beneficii **numai** dacă **nepotrivirea** dintre şir și model a fost **precedată** de o **potrivire parțială** de o anumită lungime. Numai în acest caz deplasarea modelului se realizează peste mai mult de o poziție.
- Din păcate această situație în realitate este mai degrabă excepția decât regula; potrivirile apar mult mai rar ca și nepotrivirile.
- În consecință, beneficiile acestei metode sunt reduse în marea majoritate a căutărilor normale de texte.
- Metoda de căutare inventată în 1975 de R.S. Boyer și J.S. Moore îmbunătățește performanța atât pentru situația cea mai defavorabilă cât și în general.
- Căutarea BM, după cum mai este numită, este bazată pe ideea neobișnuită de a începe compararea caracterelor de la sfârșitul modelului și nu de la început.

Precompilarea presupune următorii pași:

- (1) Pentru **fiecare caracter x care apare în model**, se notează cu d_x distanța dintre **cea mai din dreapta apariție** a lui x în cadrul modelului și sfârșitul modelului
- (2) Valoarea d_x se trece în tabloul d în poziția corespunzătoare caracterului x.
- (3) Pentru toate celelalte caractere ale setului de caractere, **care nu apar în model** d_x se face egal cu **lungimea totală** a modelului.
- (4) Pentru **ultimul caracter al modelului**, d_x se face de asemenea egal cu **lungimea totală** a modelului.

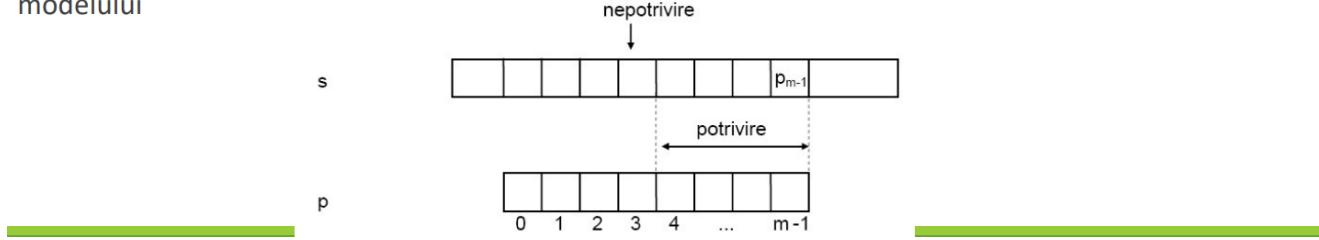


- În continuare, se presupune că în procesul de **comparare de la dreapta la stânga** al şirului cu **modelul** a apărut o **nepotrivire** între caracterele corespunzătoare.

- În această situație **modelul** poate fi imediat **deplasat spre dreapta** cu $d[p_{m-1}]$ poziții, valoare care este de regulă mai mare ca 1.

- Se precizează faptul că p_{m-1} este **caracterul din şirul baleat** s, corespunzător **ultimului caracter al modelului** la momentul considerat, **indiferent** de locul în care s-a constatat nepotrivirea

- Dacă caracterul p_{m-1} nu apare în model, deplasarea este mai mare și anume cu întreaga lungime a modelului



```

int BM(char* s, char* p)
//s - sursa, p - model
{
    int k,i,j;
    int n = strlen(s);
    int m = strlen(p);
    //construire tablou d
    for (i = 0; i < 256; i++)
        d[i] = m;
    for (j = 0; j <= m - 2; j++)
        d[p[j]] = m - j - 1; //codul se continua
    //continuare cod
    i = m;
    j = m;
    while ((j > 0) && (i <= n)) {
        j = m;
        k = i;
        while ((j > 0) && (s[k - 1] == p[j - 1])) {
            k = k - 1;
            j = j - 1;
        }
        if (j > 0)
            i = i + d[s[i - 1]];
    }
    if (j == 0)
        return k;
    else
        return -1;
}

```

Exerciții

Ex1. Implementați funcțiile de căutare liniară și căutare binară. Măsurați timpul de rulare pentru diferite valori ale dimensiunii tabloului ($N = 10\ 000, 20\ 000, 30\ 000\dots 100\ 000$). Reprezentați grafic valorile obținute pentru fiecare algoritm.

Ex2. Scrieți un program care inserează structuri într-un tablou, le caută și le șterge folosind una din metodele de hashing studiate în acest curs.

Ex3. Scrieți un program care să determine într-un mod cât mai eficient, dacă există elemente duplicate într-un tablou dat.