4. Şiruri de caractere

- Marea majoritate a celor preocupaţi de activitatea de programare sunt familiarizaţi cu şirurile de caractere întrucât aproape toate limbajele de programare includ **şirul** sau **caracterul** printre elementele predefinite ale limbajului.
    - În prima parte a capitolului se va preciza **tipul de date abstract şir**.
    - În continuare vor fi abordate **modalităţile majore de implementare a şirurilor**, prin intermediul **tablourilor** respectiv al **pointerilor**.
    - În ultima parte a capitolului vor fi precizate câteva din **tehnicile de căutare** în şiruri.

4.1. Tipul de date abstract şir

- Un şir este o colecţie 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 şi cifrelor cuprinde şi o serie de alte caractere.
    - Se subliniază faptul că într-un şir, **ordinea caracterelor** contează. Astfel şirurile `"CAL"` şi `"LAC"` deşi conţin aceleaşi caractere sunt diferite.
- De asemenea, printr-un uşor **abuz** de notaţie se consideră că un *caracter* este interschimbabil cu *un şir* constând dintr-un singur caracter, deşi strict vorbind ele sunt de tipuri diferite.
- Asemeni oricărui tip de date abstracte, definirea precisă a **TDA Şir** necesită:
    - (1) Descrierea **modelului** său **matematic.**
    - (2) Precizarea **operatorilor** care acţionează asupra elementelor tipului.
- Din punct de vedere **matematic**, elementele tipului de date abstract şir pot fi definite ca **secvenţe finite de caractere** $(c_1, c_2, \ldots, c_n)$ unde c_i este de tip caracter, iar n precizează lungimea secvenţei.
    - Cazul în care n este egal cu zero, desemnează **şirul vid**.

- În continuare se prezintă un posibil set de operatori care acţionează asupra **TDA Şir.**

- Ca şi 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:secvenţă finită de caractere.

Notaţii: *s,sub,u* - siruri;
c - valoare de tip caracter;
b - valoare booleană;
poz,lung - întregi pozitivi. [4.1.a]

Operatori:

***CreazaSirVid**(s)* - procedură ce creează şirul vid *s*;

*b:=**SirVid**(s)* - funcţie ce returnează true dacă şirul este vid;

*b:=**SirComplet**(s)* - funcţie booleană ce returnează valoarea true dacă şirul este complet;

*lung:=**LungSir**(s)* - funcţie care returnează numărul de caractere ale lui *s*;

*poz:=**PozitieSubsir**(sub,s)* - funcţie care returnează poziţia la care subşirul *sub* apare prima dată în *s*. Dacă *sub* nu e găsit în *s*, se returnează valoarea 0. Poziţiile caracterelor sunt numerotate de la stânga la dreapta începând cu 1;

***ConcatSir**(u,s)* - procedură care concatenează la sfârşitul lui *u* atâtea caractere din *s*, până când *SirComplet(u)* devine **true**;

***CopiazaSubsir**(u,s,poz,lung)* - procedură care-l face pe *u* copia subşirului din *s* începând cu poziţia *poz*, pe lungime *lung* sau până la sfârşitul lui *s*; dacă *poz>LungSir(s)* sau *poz<1*, *u* devine şirul vid;

***StergeSir**(s,poz,lung)* - procedură care şterge din *s*, începând cu poziţia *poz*, subşirul de *lung* caractere. Dacă *poz* este invalid (nu aparţine şirului), *s* rămâne nemodificat;

***InsereazaSir**(sub,s,poz)* - procedură care inserează în *s*, începând de la poziţia *poz*, şirul *sub*; eventual trunchiere la depăşirea lungimii maxime;

*c:=**FurnizeazaCarSir**(s,poz)* - funcţie care returnează caracterul din poziţia *poz* a lui *s*. Pentru *poz* invalid, se returnează caracterul nul;

***AdaugaCarSir**(s,c)* - procedură care adaugă caracterul *c* la sfârşitul şirului *s*;

***StergeSubsir**(sub,s,poz)* - procedură care şterge prima apariţie a subşirului *sub* în şirul *s* şi returnează poziţia *poz* de ştergere. Dacă *sub* nu este găsit, *s* rămâne nemodificat iar *poz* este poziţionat pe 0;

***StergeToateSubsir**(s,sub)* - şterge toate aparitiile lui *sub* în *s*.

- Operatorii definiţi pentru un TDA-şir pot fi împărţiţi în două categorii:
  - Operatori **primitivi** care reprezintă un **set minimal de operaţii** strict necesare în termenii cărora pot fi dezvoltaţi operatorii nonprimitivi.
  - Operatori **nonprimitivi** care pot fi dezvoltaţi din cei anteriori.
- Această divizare este oarecum **formală** deoarece, de obicei e mai uşor să defineşti un operator neprimitiv direct decât să-l defineşti în termenii unor operatori primitivi.
  - Spre **exemplu** operatorul **InsereazăŞir** poate fi definit în termenii operatorilor **CreazăŞirVid**, **AdaugăCar** şi **FurnizeazaCar**.
  - Algoritmul este simplu: se construieşte un şir de ieşire temporar (rezultat) căruia i se **adaugă** pe rând:
    - (1) Caracterele şirului sursă până la punctul de inserţie.
    - (2) Toate caracterele şirului de inserat (subşir).
    - (3) Toate caracterele şirului sursă de după punctul de inserţie.
    - Şirul astfel construit înlocuieşte şirul iniţial (sursa) (secvenţa [4.1.b]).

```
{Implementarea operatorului InsereazăŞir utilizând
operatorii CreazăSirVid şi AdaugăCar}

PROCEDURE InsereazaSir(subsir: TipSir; VAR sursa: TipSir;
                       p: TipIndice);
{inserează "subşir" în "sursă" între poziţiile p şi p+1}
VAR rezultat: TipSir; i: TipIndice;
BEGIN
  IF (p<1) OR (p>LungSir(sursa)) THEN
      *eroare(poziţie ilegală in inserţie)
    ELSE
      BEGIN                                          [4.1.b]
        CreazaSirVid(rezultat);
        FOR i:=1 TO p-1 DO
          AdaugaCarSir(rezultat, FurnizeazaCarSir(sursa,i));
        FOR i:=1 TO LungSir(subsir) DO
```

```
          AdaugaCarSir(rezultat, FurnizeazaCarSir(subsir,i));
        FOR i:=p TO LungSir(sursa) DO
          AdaugaCarSir(rezultat, FurnizeazaCarSir(sursa,i));
        CreazaSirVid(sursa);
        FOR i:=1 TO LungSir(rezultat) DO
          AdaugaCarSir(sursa, FurnizeazaCarSir(rezultat,i));
      END {ELSE}
 END;{InsereazaSir}
-------------------------------------------------------------
/*Implementarea operatorului Inserează_Şir utilizând
operatorii Crează_Sir_Vid, Adaugă_Car_Sir şi
Furnizeaza_Car_Sir */

void insereaza_sir(tip_sir subsir, tip_sir* sursa,
                   tip_indice p)
  /*inserează subsirul în sursa între poziţiile p şi p+1*/
{
  tip_sir rezultat; tip_indice i;

  if ((p<1)||(p>lung_sir(*sursa))) ;
      /*eroare (poziţie ilegală în inserţie)*/
    else
      {                                        /*[4.1.b]*/
        creaza_sir_vid(rezultat);
        for(i= 1; i<=p-1; i++)
          adauga_car_sir(rezultat,
                         furnizeaza_car_sir(*sursa,i));
        for(i=1; i<=lung_sir(subsir); i++)        _
          adauga_car_sir(rezultat,
                         furnizeaza_car_sir(subsir,i));
        for(i=p; i<=lung_sir(*sursa); i++)
          adauga_car_sir(rezultat,
                          furnizeaza_car_sir(*sursa,i));
        creaza_sir_vid(*sursa);
        for(i=1; i<=lung_sir(rezultat); i++)
          adauga_car_sir(*sursa,
                          furnizeaza_car_sir(rezultat,i));
      }
}
-------------------------------------------------------------
```

4.2. Implementarea tipului de date abstract şir

- Sunt cunoscute **două tehnici majore** de implementare a şirurilor:
    - (1) Implementarea bazată pe **tablouri**
    - (2) Implementarea bazată pe **pointeri**.

4.2.1. Implementarea şirurilor cu ajutorul tablourilor

- Cea mai simplă implementare a **TDA-şir** se bazează pe două elemente:
  - (1) Un **întreg** reprezentând lungimea şirului.
  - (2) Un **tablou** de caractere care conţine şirul propriu-zis.
- În tablou caracterele pot fi păstrate ca atare sau într-o formă **împachetată**.
- Exemple pentru o astfel de implementare în Pascal respectiv C apar în secvenţa [4.2.1.a].

```
-------------------------------------------------------------
{Exemplu de implementare a TDA Sir utilizând tablouri –
structuri de date - varianta Pascal}

CONST LungimeMax = ...;
TYPE TipLungime = 0..LungimeMax;
     TipIndice = 1..LungimeMax;                [4.2.1.a]
     TipSir = RECORD
       lungime: TipLungime;
       sir: ARRAY[TipIndice] OF char
     END;
VAR s: TipSir;
-------------------------------------------------------------
/* Exemplu de implementare a TDA Sir utilizând tablouri –
structuri de date - varianta C*/

enum {lungime_max = 100};
typedef unsigned char tip_lungime;

typedef unsigned char tip_indice;
                                          /*[4.2.1.a]*/
typedef struct tip_sir {
  tip_lungime lungime;
  char sir[lungime_max];
} tip_sir;
tip_sir s;
-------------------------------------------------------------
```

- Acest mod de implementare al şirurilor **nu** este unic.
- Se utilizează **tablouri** întrucât tablourile ca şi şirurile sunt **structuri liniare**.
  - Câmpul `lungime` este utilizat deoarece şirurile au lungimi diferite în schimb ce tablourile au lungimea fixă.
- Implementarea se poate dispersa de câmpul `lungime`, caz în care se poate utiliza un caracter convenit pe post de **santinelă de sfârşit** (**marker**).
  - În această situaţie operatorul `LungimeŞir` va trebui să **contorizeze** într-o manieră liniară caracterele până la detectarea markerului.

- Din acest motiv este preferabil ca **lungimea şirului** să fie considerată un element **separat** al implementării.

- În contextul implementării şirurilor cu ajutorul tablourilor se defineşte noţiunea de **şir complet**.

  - **Şirul complet** este şirul care are lungimea egală cu `LungimeMaximă`, adică egală cu **dimensiunea** tabloului definit spre a-l implementa.

  - Şirurile implementate în acest mod **nu** pot depăşi această lungime, motiv pentru care în acest context operează operatorul boolean `ŞirComplet`.

- În accepţiunea modelului anterior prezentat, în continuare se prezintă o implementare a operatorilor primitivi [4.2.1.b,c,d,e] variantele Pascal si C

```
-------------------------------------------------------------
{Exemplu de implementare a TDA Sir utilizând tablouri -
varianta Pascal}

PROCEDURE CreazaSirVid(VAR s: TipSir);          {O(1)}
  BEGIN
    s.lungime:= 0                                [4.2.1.b]
  END;{CreazaSirVid}
-------------------------------------------------------------
FUNCTION LungSir(VAR s: TipSir): TipLungime;    {O(1)}
  BEGIN
    LungSir:= s.lungime                          [4.2.1.c]
  END; {LungSir}
-------------------------------------------------------------
FUNCTION FurnizeazaCar(s: TipSir,poz: TipIndice): char;
  BEGIN                                          {O(1)}
    IF (poz<1) OR (poz>s.lungime) THEN
        BEGIN                                    [4.2.1.d]
          *eroare(poziţie incorectă);
          FurnizeazaCar:= chr(0) {caracterul vid}
        END
      ELSE
        FurnizeazaCar:= s.sir[poz]
  END; {FurnizeazaCar}
-------------------------------------------------------------
PROCEDURE AdaugaCarSir(VAR s: TipSir; c: char); {O(1)}
  BEGIN
    IF s.lungime=LungimeMax THEN
       *eroare(se depăşeşte lungimea maximă a şirului)
      ELSE
        BEGIN
          s.lungime:= s.lungime+1;               [4.2.1.e]
          s.sir[s.lungime]:= c
        END
  END; {AdaugaCarSir}
-------------------------------------------------------------
/* Exemplu de implementare a TDA Sir utilizând tablouri -
varianta C*/
```

```
void creaza_sir_vid(tip_sir* s)                          /*O(1)*/
  {
    s->lungime= 0;                              /*[4.2.1.b]*/
  }
-------------------------------------------------------------
tip_lungime lung_sir(tip_sir* s)                         /*O(1)*/
  {
    tip_lungime lung_sir_result;
    lung_sir_result= s->lungime;                /*[4.2.1.c]*/
    return lung_sir_result;
  }
-------------------------------------------------------------
char furnizeaza_car(tip_sir s,tip_indice poz)
  {                                                     /*O(1)*/
    char furnizeaza_car_result;
    if ((poz<1)||(poz>s.lungime))
        {                                           /*[4.2.1.d]*/
          /*eroare(poziţie incorectă);*/
          furnizeaza_car_result= '/0'; /*caracterul vid*/
        }
      else
        furnizeaza_car_result= s.sir[poz-1];
    return furnizeaza_car_result;
  }
-------------------------------------------------------------
void adauga_car_sir(tipsir* s, char c)                  /*O(1)*/
  {
    if (s->lungime==lungime_max) ;
       /*eroare(se depaşeşte lungimea maximă a şirului)*/
      else
        {
          s->lungime= s->lungime+1;              /*[4.2.1.e]*/
          s->sir[s->lungime-1]= c;
        }
  }
-------------------------------------------------------------
```

- Se observă că toate aceste operaţii rulează în O*(1)* unităţi de timp indiferent de lungimea şirului.

- Procedurile ***CopiazăSubŞir***, ***ConcatŞir***, ***ŞtergeŞir*** şi ***InsereazaŞir*** se execută într-un interval de timp liniar O*(n)*, unde *n* este după caz lungimea subşirului sau a şirului de caractere.

  - Spre **exemplu** procedura ***CopiazăSubşir***`(u,s,poz,lung)` returnează în `u` subşirul din `s` având `lung` caractere începând cu poziţia `poz`.

  - Accesul la elementele subşirului se realizează direct `(s.şir[poz], s.şir[poz+1],...,s.şir[poz+lung-1])`, astfel încât consumul de timp al execuţiei este dominat de mutarea caracterelor [4.2.1.f].

```
-------------------------------------------------------------
{Implementarea operatorului CopiazăSubşir}

PROCEDURE CopiazaSubsir(VAR u,s: TipSir, poz, lung:
```

```
                                TipLungime);       {O(n)}

{copiază un subşir din s în şirul u}

VAR indexSursa,indexCopiere: TipLungime;
BEGIN
  IF (poz<1) OR (poz>s.lungime) THEN
      u.lungime:= 0
    ELSE
      BEGIN                                          [4.2.1.f]
        indexSursa:= poz-1;
        indexCopiere:= 0;
        WHILE (indexSursa<s.lungime) AND
                 (indexCopiere<u.lungime) DO BEGIN
            indexSursa:= indexSursa+1;
            indexCopiere:= indexCopiere+1;
            u.sir[indexCopiere]:= s.sir[indexSursa]
          END; {WHILE}
        u.lungime:= indexCopiere
      END {ELSE}
 END; {CopiazaSubsir}
-------------------------------------------------------------

/*Implementarea operatorului CopiazăSubşir*/      /*O(n)*/

{copiază un subşir din s în şirul u}

void copiaza_subsir(tip_sir* u, tip_sir* s, tip_lungime poz,
tip_lungime lung)
{
    tip_lungime index_sursa,index_copiere;

  if ((poz<1)||(poz>s->lungime))
      u->lungime= 0;
    else
      {                                      /*[4.2.1.f]*/
        index_sursa= poz-1;
        index_copiere= 0;
        while ((index_sursa<s->lungime) &&
                 (index_copiere<u->lungime))  {
            index_sursa= index_sursa+1;
            index_copiere= index_copiere+1;
            u->sir[index_copiere-1]= s->sir[index_sursa-1];
          }    /*while*/
        u->lungime= index_copiere;
      }
}
-------------------------------------------------------------
```

- Se observă faptul că în interiorul buclei **WHILE** există 3 atribuiri, care pentru o lungime `lung` a sub şirului determină `3*lung + 3` atribuiri.

- Se observă de asemenea că procedura `CopiazăSubşir` este liniară în raport cu lungimea `lung` a subşirului.

- Un alt exemplu îl reprezintă implementarea funcţiei `PoziţieSubşir` [4.2.1.g].

```
-------------------------------------------------------------
{Implementarea operatorului PoziţieSubşir}

FUNCTION PozitieSubsir(VAR sub,s: tipSir): TipLungime;

{determină poziţia lui "sub" în cadrul şirului "s" (prima
apariţie)}

 VAR mark, {index pentru punctul de start al unei comparaţii}
     indexSub,           {index subşir}
     indexSir: TipIndex; {index şir}
     poz: TipLungime;    {poziţia lui sub în s}
     stop: boolean; {devine adevărat când elementele
             corespunzătoare din s şi sub nu sunt egale}
 BEGIN
   indexSir:= 1;
   poz:= 0;
   REPEAT
     indexSub:= 1;
     mark:= indexSir;
     stop:= false;
     WHILE (NOT stop) AND (indexSir<=s.lungime) AND
              (indexSub<=sub.lungime) DO
       IF s.sir[indexSir]=sub.sir[indexSub] THEN
           BEGIN
             indexSir:= indexSir+1;
             indexSub:= indexSub+1            [4.2.1.g]
           END
         ELSE
           stop:= true; {WHILE}
     IF indexSub>sub.lungime THEN
           poz:= mark {potrivire}
        ELSE
          indexSir:= mark+1 {nepotrivire => avans model}
   UNTIL (poz>0) OR (indexSir+sub.lungime-1>s.lungime);
   PozitieSubsir:= poz
 END;  {PozitieSubsir}
-------------------------------------------------------------
/*Implementarea operatorului PoziţieSubşir*/

tip_lungime pozitie_subsir(tip_sir* sub,tip_sir* s)
 {
   tip_index mark, /*index pentru punctul de start al unei
                     comparaţii*/
   index_sub,          /*index subsir*/
   index_sir;          /*index sir*/
   tip_lungime poz;    /*poziţia lui sub în s*/
   boolean stop;  /*devine adevărat când elementele
             corespunzătoare din s şi sub nu sunt egale*/

   tip_lungime pozitie_subsir_result;
   index_sir= 1;
   poz= 0;
```

```
   do {
     index_sub= 1;
     mark= index_sir;
     stop= false;
     while ((!stop)&&(index_sir<=s->lungime)&&
              (index_sub<=sub->lungime))
       if (s->sir[index_sir-1]==sub->sir[index_sub-1])
           {
             index_sir= index_sir+1;
             index_sub= index_sub+1;         /*[4.2.1.g]*/
           }
         else
           stop= true;  /*while*/
     if (index_sub>sub->lungime)
            poz= mark; /*potrivire*/
        else
          index_sir= mark+1; /*nepotrivire => avans model*/
   } while (!(poz>0) ||
           (index_sir+sub->lungime-1>s->lungime)));
   pozitie_subsir_result= poz;
   return pozitie_subsir_result;
 }
-----------------------------------------------------------------
```

- Complexitatea functiei ***PoziţieSubşir***`(sub,s)`este O(*lungime* ∗ *s.lungime*) unde `lungime` este lungimea modelului iar `s.lungime` este lungimea şirului în care se face căutarea.

    - Există însă şi alte metode mult mai **performante** de căutare care fac obiectul unui paragraf ulterior.

- Unul dintre marele **avantaje** ale utilizării datelor abstracte este următorul:

    - În situaţia în care se găseşte un algoritm mai **performant** pentru o anumită operaţie, se poate foarte simplu înlocui o **implementare** cu alta fără a afecta restul programului în condiţiile păstrării nealterate a prototipului operatorului.

4.2.2. Tabele de şiruri

- Utilizarea tabloului în implementarea şirurilor are avantajul că operatorii sunt uşor de redactat, simplu de înţeles şi suficient de rapizi.

- Cu toate că acest mod de implementare este economic din punct de vedere al **timpului** de execuţie, el este ineficient din punctul de vedere al **spaţiului** utilizat.

    - Lungimea **maximă** a tabloului trebuie aleasă la dimensiunea **celui mai lung şir** preconizat a fi utilizat deşi în cea mai mare parte a cazurilor se utilizează şiruri mult mai **scurte**.

- O modalitate de a economisi spaţiul în dauna vitezei de lucru specifică implementării cu ajutorul tablourilor, este aceea de a utiliza următoarele structuri de date:

- (1) Un **tablou** suficient de lung numit "**heap**" (grămadă) pentru a memora toate şirurile.
- (2) O **tabelă** auxiliară de şiruri care păstrează evidenţa şirurilor în heap.

- Aceasta tehnică este utilizată în unele interpretere BASIC.
- În acest mod de implementare a şirurilor, un şir este reprezentat printr-o **intrare** în tabela de şiruri care conţine două valori (fig.4.2.2.a):
  - **Lungimea** şirului .
  - Un **indicator** în heap care precizează **începutul** şirului.

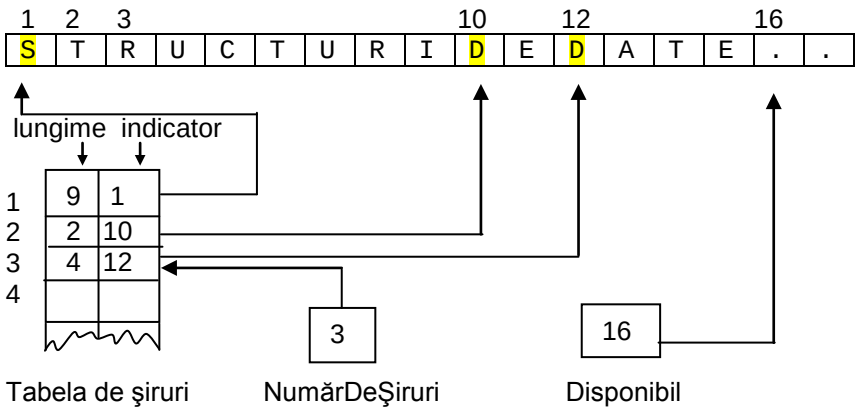


Fig. 4.2.2.a. Modelul tabelei de şiruri

- Un exemplu de implementare în limbajul PASCAL apare în secvenţa [4.2.2.a].
- Se observă că această implementare presupune:
  - Variabilele globale `TabelaDeŞiruri`, `Heap`, `Disponibil` şi `NumărDeŞiruri`.
  - Structurarea în forma unui articol cu câmpurile `lungime` şi `indicator` a elementelor tabelei de şiruri.

```
-------------------------------------------------------------
{Implementarea şirurilor utilizând tabele - structuri de
date}

CONST LungimeHeap = {număr maxim de caractere};
      LungimeTabela = {număr maxim de şiruri};

TYPE ElementTabela = RECORD
       lungime,
```

```
      indicator: 1..LungimeHeap                    [4.2.2.a]
    END;
    TipSir = 1..LungimeTabela;

VAR TabelaDeSiruri: ARRAY[1..LungimeTabela] OF
                                          ElementTabela;
    Heap: ARRAY[1..LungimeHeap] OF char;
    Disponibil: 0..LungimeHeap; {primul caracter liber}
    NumarDeSiruri: 0..LungimeTabela;
-------------------------------------------------------------
```

- În continuare se prezintă pentru exemplificare implementarea operatorului ***AdaugăCarŞir***.

    - Deoarece adăugarea unui caracter produce depăşirea şirului în cauză în dauna şirului următor din spaţiul de lucru, funcţia ***AdaugăCarŞir*** trebuie:

        - [1] Să creeze o nouă copie a şirului în zona disponibilă a heap-ului.
        - [2] Să recopieze şirul în noua locaţie.
        - [3] Să adauge noul caracter.
        - [4] Să reactualizeze tabela de şiruri şi variabila `Disponibil` [4.2.2.b].

```
-------------------------------------------------------------
{Implementarea operatorului AdaugăCarŞir}

PROCEDURE AdaugaCarSir(VAR s: TipSir; c: char); {O(n)}
   {adaugă caracterul c la sfârşitul sirului s}
   VAR i,lungimeVeche,indicatorVechi: integer;

   BEGIN
     IF (s<1)OR(s>NumarDeSiruri) THEN
         *eroare(referinţă invalidă de şir)
       ELSE                                        [4.2.2.b]
         BEGIN
  [1]     lungimeVeche:= TabelaDeSiruri[s].lungime;
          indicatorVechi:= TabelaDeSiruri[s].indicator;
  [2]     FOR i:=0 TO lungimeVeche-1 DO
            Heap[Disponibil+i]:=Heap[indicatorVechi+i];
  [3]     Heap[Disponibil+lungimeVeche]:= c; {se adauga c}
  [4]     TabelaDeSiruri[s].indicator:= Disponibil;
          TabelaDeSiruri[s].lungime:= lungimeVeche+1;
          Disponibil:= Disponibil+lungimeVeche+1
         END {ELSE}
   END; {AdaugaCarSir}
-------------------------------------------------------------
```

- Se observă că în această implementare ***AdaugăCarŞir*** necesită O*(n)* unităţi de timp pentru un şir de `n` caractere faţă de implementarea bazată pe tablouri care necesită un timp constant (O*(1)*).

- Aceasta este tributul plătit facilităţii de a putea lucra cu şiruri de **lungime variabilă**.

- De asemenea în urma creării noii copii a şirului supus operaţiei de adăugare, vechea instanţă a şirului rămâne în heap ocupând memorie în mod inutil.

- O altă problemă potenţială a acestei implementări este următoarea:

  - Se consideră o procedură care realizează **copierea** unui şir **sursă** într-un şir **destinaţie**.

  - În implementarea bazată pe **tabelă de şiruri** acest lucru se poate realiza simplu făcând ca şirului destinaţie să-i corespundă **acelaşi indicator** în `heap` ca şi şirului sursă.

  - Acest fenomen este cunoscut în programare sub denumirea de **supraîncărcare** ("**aliasing**"), adică două variabile diferite se referă la aceeaşi locaţie de memorie.

  - Dacă se cere însă **ştergerea** şirului sursă sau destinaţie se pot pierde ambele şiruri.

  - Această problemă poate fi ocolită printr-o implementare adecvată.

- Pentru rezolvarea **problemei supraîncărcării**, se poate proceda ca şi în cazul operatorului ***`AdaugăCarŞir`***, adică:

  - (1) Pentru şirul destinaţie se creează o **instanţă** începând cu prima locaţie disponibilă în `heap`.

  - (2) Se copiază sursa în noua locaţie.

  - (3) Se introduce referinţa care indică această locaţie, în tabela de şiruri în intrarea corespunzătoare şirului nou creat.

- De fapt, în anumite variante ale limbajului `BASIC`, această manieră de rezolvare a copierii stă la baza implementării instrucţiunii `LET`.

4.2.3. Implementarea şirurilor cu ajutorul pointerilor

- Esenţa implementării **şirurilor** cu ajutorul pointerilor rezidă în reprezentarea acestora printr-o colecţie de **celule înlănţuite**.

- Fiecare **celulă** conţine [4.2.3.a]:

  - Un **caracter** (sau un grup de caractere într-o implementare mai elaborată).

  - Un **pointer** la celula următoare.

```
{Implementarea şirurilor cu ajutorul pointerilor - structuri
de date}

TYPE PointerCelula = ^Celula;
     Celula = RECORD
       ch: char;
       urm: PointerCelula                          [4.2.3.a]
     END;
     TipSir = RECORD
       lungime: integer;
       cap,coada: PointerCelula
     END;
-----------------------------------------------------------
```

- Spre exemplu în fig. 4.2.3.a apare reprezentarea şirului `'DATE'` în această implementare,

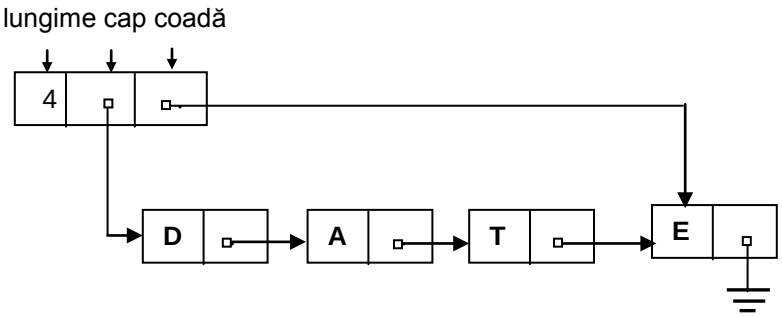


Fig. 4.2.3.a. Implementarea şirurilor cu ajutorul pointerilor

- În secvenţele [4.2.3.b, c, d, e] apare implementarea unor operatori primitivi specifici.

```
-----------------------------------------------------------
{Implementarea şirurilor cu ajutorul pointerilor}

PROCEDURE CreazaSirVid(VAR s: TipSir);   {O(1)}
  BEGIN
    s.lungime:= 0;
    s.cap:= nil;                                  [4.2.3.b]
    s:coada:= nil
  END; {CreazaSirVid}
-----------------------------------------------------------
FUNCTION LungSir(s: TipSir): integer;    {O(1)}
  BEGIN                                           [4.2.3.c]
    LungSir:= s.lungime
  END; {LungSir}
-----------------------------------------------------------
FUNCTION FurnizeazaCarSir(s:TipSir; poz:integer): char;
  VAR contor: integer;                   {O(n)}
      p: PointerCelula;                           [4.2.3.d]
  BEGIN
    IF (poz<1) AND (poz>s.lungime) THEN
        BEGIN
          *eroare(index eronat)
```

```
          FurnizeazaCarSir:= chr(0) {caracterul nul}
        END
      ELSE
        BEGIN
          p:= s.cap;
          FOR contor:= 1 TO poz-1 DO p:= p^.urm;
          FurnizeazaCarSir:= p^.ch
        END
  END; {FurnizeazaCarSir}
-------------------------------------------------------------
PROCEDURE AdaugaCarSir(var s:TipSir; c:char); {O(1)}
  BEGIN
    IF s.lungime=0 THEN {primul caracter al sirului}
        BEGIN
          new(s.cap);
          s.cap^.ch:= c;
          s.cap^.urm:= nil;
          s.coada:= s.cap                          [4.2.3.e]
        END
      ELSE
        BEGIN
          new(s.coada^.urm);
          s.coada^.urm^.ch:= c;
          s.coada^.urm^.urm:= nil;
          s.coada:= s.coada^.urm
        END;{ELSE}
    s.lungime:= s.lungime+1
  END; {AdaugaCarSir}
-------------------------------------------------------------
```

- În ceea ce priveşte implementarea operaţiei de **copiere,** apar aceleaşi probleme legate de **supraîncărcare** care pot fi rezolvate într-o manieră asemănătoare prin generarea unui **şir nou destinaţie** identic cu sursa.

- Un astfel de operator ***Copiază***`(destinaţie,sursa)` poate fi utilizat cu succes în implementarea operaţiei ***ConcatŞir*** [4.2.3.f].

```
-------------------------------------------------------------
{Implementarea operatorului ConcatŞir utilizând operatorul
Copiază}

PROCEDURE ConcatSir(VAR u: TipSir; s: TipSir);
  VAR temp: TipSir;
  BEGIN
    IF u.lungime=0 THEN
         Copiaza(u,s)
      ELSE
        IF s.lungime<>0 THEN
          BEGIN
            CreazăŞirVid(temp);                [4.2.3.f]
            Copiaza(temp,s);
            u.coada^.urm:= temp.cap;
            u.coada:= temp.coada;
            u.lungime:= u.lungime+s.lungime
          END
```

```
END; {ConcatSir}
-------------------------------------------------------------
```

- Acest mod de reprezentare a şirurilor are unele **avantaje**.
    - (1) Permite o mare **flexibilitate** în gestionarea şirurilor.
    - (2) Înlătură **dezavantajul** lungimii fixe.
- Principalele **dezavantaje** rezidă în:
    - (1) **Parcurgerea secvenţială** a şirului în vederea realizării accesului la o anumită poziţie.
    - (2) **Risipa** de spaţiu de memorie datorată prezenţei pointerului asociat fiecărui caracter al şirului.

4.2.4. Comparaţie între metodele de implementare a şirurilor

- După cum s-a văzut, se utilizează trei maniere de implementare a şirurilor:
    - Implementarea bazată pe **tablouri.**
    - Implementarea bazată pe **pointeri**.
    - Implementarea bazată pe **tabele** care reuneşte parţial caracteristicile structurilor anterioare.
- Din punctul de vedere al paragrafului de faţă prezintă interes **primele două metode**, caracteristicile celei de-a treia putând fi uşor deduse din acestea.
- **Diferenţele** marcante dintre cele două moduri de implementare a şirurilor şi anume prin **tablouri** şi **pointeri** îşi au de fapt originea în **caracterul fundamental** al **structurilor de date suport**. Astfel:
    - Şirul implementat ca **tablou** este o **structură de date statică** cu **acces direct**.
    - Şirul implementat prin **pointeri** este o **structură de date dinamică** cu **acces secvenţial**, de fapt o **listă înlănţuită**.
- Aceste caracteristici fundamental diferite influenţează într-o manieră **decisivă** atât **proprietăţile şirurilor** în fiecare din cele două moduri de reprezentare, cât şi **algoritmii** care implementează **operatorii specifici**.
- **Implementarea şirurilor cu ajutorul tablourilor** oferă următoarele avantaje**:**
    - Posibilitatea realizării **directe** a accesului la caractere.
    - Accesul la o poziţie precizată într-un şir este **imediat**, element care avantajează operaţiile ***CopiazăSubşir***, ***ŞtergeŞir***, ***Inserează Şir***, ***FurnizeazăCarŞir***.

- Algoritmii specifici pentru ***FurnizeazăCarŞir***, ***AdaugăCarŞir*** şi ***LungŞir*** sunt rapizi (O*(1)*).

- Restul operatorilor în marea lor majoritate au performanţa O*(n)*.

- Operatorul ***PoziţieSubşir*** în implementarea bazată pe **căutare directă** are performanţa O$(n * m)$ unde `n` este lungimea şirului iar `m` lungimea subşirului, (există pentru această reprezentare şi metode mai performante).

- **Dezavantajele** acestui mod de implementare sunt:

    - **Dimensiunea fixă** (limitată) a şirurilor.

    - **Trunchierea** cauzată de inserţia într-un şir complet.

    - **Inserţia** şi **suprimarea** într-un şir presupun mişcări de elemente şi un consum de timp proporţional cu lungimea şirului.

- **Implementarea şirurilor cu ajutorul pointerilor:**

    - Se bucură de **flexibilitatea** specifică acestei implementări.

    - Este grevată de proprietăţile rezultate din **caracterul secvenţial** al reprezentării.

    - Faza de inserţie propriu-zisă şi suprimare din cadrul operatorilor ***InsereazăŞir*** şi ***ŞtergeŞir*** se realizează în O*(1)* unităţi de timp

    - **Căutarea** poziţiei însă necesită practic O*(n)* unităţi.

    - De fapt căutarea **secvenţială** a poziţiei este marele **dezavantaj** al acestei reprezentări care diminuează performanţa marii majorităţi a operatorilor.

    - Cu toate că în fiecare moment se consumă spaţiu de memorie numai pentru caracterele existente, totuşi **risipa de memorie** datorată pointerilor este mare.

    - Performanţa operatorilor ***ConcatŞir*** şi ***AdaugăCarŞir*** creşte în mod semnificativ dacă se utilizează pointerul "`co`adă".

    - Această implementare este indicat a fi utilizată atunci când:

        - Este foarte important ca şirurile să **nu** fie **trunchiate**.

        - Când se cunoaşte cu probabilitate redusă lungimea maximă a şirurilor.

- Admiţând trunchierea ca şi un rău necesar, marea majoritate a limbajelor de programare care definesc şi implementează în cadrul limbajului tipul `string` şi operatorii aferenţi utilizează implementarea şirurilor bazată pe **tablouri.**

4.3. Tehnici de căutare în şiruri

- Una din operaţiile cele mai frecvente care se executată asupra şirurilor este **căutarea**.
    - Întrucât **performanţa** acestei operaţii are o importanţă covârşitoare asupra marii majorităţi a prelucrărilor care se realizează într-un sistem de calcul, studiul **tehnicilor 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**.

4.3.1. Căutarea tabelară (Table search)

- O căutare într-un tablou se numeşte şi **căutare tabelară** ("**table search**"), cu deosebire în situaţia în care cheile sunt la rândul lor **obiecte structurate** spre exemplu **tablouri de numere** sau de **caractere**.
    - Cazul mai des întâlnit este acela în care cheile tablourilor sunt **şiruri de caractere** sau **cuvinte**.
- În continuare, pentru obiectivele acestui paragraf, se definesc pentru o **structură şir,** în termeni de logica predicatelor, **relaţia de coincidenţă** respectiv **relaţia de ordonare** (lexicografică) a două şiruri `x` şi `y` după cum urmează [4.3.1.a,b,c]:

```
-------------------------------------------------------------
{Definirea tipului şir}
TYPE TipSir = ARRAY[0..m-1] OF char;              [4.3.1.a]
-------------------------------------------------------------
{Definirea relaţiei de egalitate a două şiruri
(coincidenţa)}
(x=y)<=(A_j:0≤j<m:x_j=y_j)                        [4.3.1.b]
-------------------------------------------------------------
{Definirea relaţiei de ordonare a două şiruri}
(x<y)<= E_i:0≤i<m:((A_j:0≤j<i:x_j=y_j)&(x_i<y_i))    [4.3.1.c]
-------------------------------------------------------------
```

- Pentru a stabili **coincidenţa**, este necesară stabilirea **egalităţii tuturor caracterelor** corespunzătoare din şirurile comparate.
    - Presupunând că lungimea şirurilor este mică, (spre exemplu mai mică decât 30), în acest scop se poate utiliza **căutarea liniară**.
- În cele mai multe aplicaţii se lucrează cu şiruri de **lungime variabilă**, asociindu-se fiecărui şir o **informaţie suplimentară** referitoare la **lungimea** sa.
    - Utilizând tipul şir anterior definit, lungimea **nu** poate depăşi valoarea maximă `m`.
    - Deşi este limitativă, această schemă permite suficientă **flexibilitate** evitând alocarea dinamică a memoriei.

- Sunt utilizate în mod frecvent **două moduri de reprezentare a lungimii şirurilor**:
    - (1) Lungimea este specificată **implicit**, plasând pe ultima poziţie a şirului (după ultimul caracter) un caracter prestabilit (**marker**) - de exemplu caracterul având codul 0 (`CHR(0)`).
        - Pentru aplicaţiile ce urmează este important ca marker-ul să fie cel mai **mic** caracter al setului de caractere.
        - În limbajul C este chiar codul cu valoarea zero.
    - (2) Lungimea şirului este memorată în mod **explicit** ca şi **prim element** al tabloului.
        - Astfel un şir are forma $s = s_0, s_1, s_2, \ldots, s_{n-1}$ unde $s_1, \ldots, s_{n-1}$ sunt caracterele şirului iar s_0 memorează **lungimea şirului de caractere**.
        - **Avantajul** acestei soluţii: lungimea şirului este direct disponibilă.
        - **Dezavantajul**: lungimea este limitată la valoarea maximă reprezentabilă pe unitatea de informaţie alocată unui caracter, de regulă un octet (255).
- În continuare, pentru precizarea **lungimii unui şir** se va utiliza modalitatea (1), respectiv cea bazată pe caracterul **marker,** definit drept caracterul al cărui cod are valoarea 0.
- În aceste condiţii, **compararea** a două şiruri va lua forma [4.3.1.d]:

```
-------------------------------------------------------------
{Compararea liniară a două şiruri}

VAR x,y: TipSir;
                                                  [4.3.1.d]
    i:= 0;
    WHILE (x[i]=y[i])AND(x[i]<>chr(0)) DO i:= i+1; {O(n)}
-------------------------------------------------------------
```

    - Caracterul corespunzător codului 0 acţionează ca şi **fanion**.
- **Invariantul** buclei, adică condiţia a cărei îndeplinire determină reluarea buclei (ciclului), este [4.3.1.e], iar **condiţia** de **terminare** [4.3.1.f]:

```
-------------------------------------------------------------
{Invariantul buclei de comparare}

  Aj:0≤j≤i:xj=yj≠chr(0)                          [4.3.1.e]
-------------------------------------------------------------
{Condiţia de terminare a buclei de comparare}

((xi ≠yi)OR(xi=yi=chr(0)))&(Aj:0≤j<i:xj=yj ≠chr(0))[4.3.1.f]
-------------------------------------------------------------
```

- **Condiţia de terminare a buclei de comparare** stabileşte:
  - (1) **Coincidenţa** celor două şiruri $x = y$ **dacă** la terminarea buclei $x_i = y_i = chr(0)$.
  - (2) **Inegalitatea** celor două şiruri $x < y (x > y)$ **dacă** la terminarea buclei $x_i < y_i$ $(x_i > y_i)$.
- Se face precizarea că inegalitatea $x_i < y_i$ poate apare şi în situaţia în care x **coincide** cu y dar x e mai **scurt** decât y.
  - În acest caz se obţine $x_i \neq y_i$ dar $x_i = chr(0)$ (s-a ajuns la sfârşitul şirului x).
  - Întrucât în această situaţie x_i este cel mai mic caracter, aceasta echivalează cu $x_i < y_i$, deci $x < y$, ceea ce este corect inclusiv din punctul de vedere al **ordonării lexicografice**.
- În acest context, **căutarea tabelară** este de fapt o **căutare încuibată** care constă:
  - (1) Dintr-o parcurgere a intrărilor unei tabele de şiruri.
  - (2) Dintr-o secvenţă de comparaţii între componentele individuale ale şirului curent şi cele ale şirului căutat, realizată în cadrul fiecărei intrări.
- Se consideră următoarele structuri de date [4.3.1.g], unde T este tabela de şiruri iar x: TipSir, argumentul căutat.

```
-------------------------------------------------------------
{Căutarea tabelară - structuri de date}

CONST n = {numărul de şiruri din tabela de şiruri};
      m = {numărul maxim de caractere dintr-un şir};
          {sfârşitul unui şir se marchează cu chr(0)}

TYPE TipSir = ARRAY[0..m-1] OF char;
     TipTabela = ARRAY[0..n-1] OF TipSir;      [4.3.1.g]

VAR T: TipTabela;
-------------------------------------------------------------
```

- Presupunând că n este suficient de mare şi că **tabela** este **ordonată alfabetic**, se poate utiliza **căutarea binară**.
- Utilizând algoritmul **căutării binare** şi **compararea a două şiruri**, se poate redacta următoarea variantă de **căutare tabelară** [4.3.1.h].

```
-------------------------------------------------------------
{Căutarea tabelară}

FUNCTION CautareTabelara(VAR T: TipTabela; VAR x: TipSir;
                         VAR poz: integer): boolean;
  VAR i,s,d,mij: integer;
  BEGIN
```

```
    s:= 0;d:= n;
    WHILE s<d DO  {căutare binară}            [4.3.1.h]
      BEGIN
        mij:= (s+d) div 2; i:= 0;
        WHILE (T[m,i]=x[i])AND(x[i]<>chr(0)) DO i:=i+1;
        IF T[m,i]<x[i] THEN s:= m+1
          ELSE d:= m
      END;{WHILE}
    IF d<n THEN {comparare de siruri}
      BEGIN
        i:= 0;
        WHILE (T[d,i]=x[i])AND(x[i]<>chr(0)) DO
          i:= i+1
      END;{IF}
      poz:= d;
      CautareTabelara:= (d<n)AND(T[d,i]=x[i])
  END; {CautareTabelara}
-------------------------------------------------------------
```

- Funcţia ***CautareTabelară*** primeşte ca parametri de intrare tabela `T` şi sirul de cautat `x` şi returnează **`true`** respectiv **`false`** după cum căutarea este **reuşită** sau **nu**.
    - În cazul unei **căutări reuşite**, în variabila de ieşire `poz` se returnează intrarea corespunzătoare din tabelă.
    - În cazul unei **căutări nereuşite** variabila de ieşire `poz` returnează o valoare nedeterminată.
- Se observă că `T` şi `x` care sunt şiruri, se declară cu **VAR**, pentru a li se transmite (prin stivă) doar adresa.

4.3.2. 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 şi un tablou `p` de `m` caractere, unde `0<m<n` declarate ca în secvenţa [4.3.2.a].
    - **Căutarea de şiruri directă** are drept scop stabilirea **primei apariţii** a lui `p` în `s`.

```
-------------------------------------------------------------
{Căutarea de şiruri directă - structuri de date}

VAR s: ARRAY[0..n-1] OF char;
    p: ARRAY[0..m-1] OF char;                  [4.3.2.a]
-------------------------------------------------------------
/*Căutarea de şiruri directă - structuri de date*/
```

```
typedef unsigned char boolean;
char s[n];
char p[m];                                          /*[4.3.2.a]*/
```

- De regulă, `s` poate fi privit ca un **text**, iar `p` ca un cuvânt **model** ("pattern") a cărui **primă apariţie** se caută în textul `s`.
    - Aceasta este o **operaţie fundamentală** în toate sistemele de prelucrare a textelor şi în acest sens este de mare interes găsirea unor algoritmi cât mai eficienţi.
- 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 apariţia unei **coincidenţe** de lungime `j` caractere între model şi şir.
    - Acest lucru este formalizat de **predicatul `P(i,j)`** [4.3.2.b].

$$P(i,j) <= A_k : 0 \le k < j : s_{i+k} = p_k \qquad [4.3.2.b]$$

- **Predicatul `P(i,j)`** precizează faptul că există o **coincidenţă** între:
    - Şirul `s` (începând cu indicele `i`).
    - Şirul `p` (începând cu indicele `0`).
    - Coincidenţa se extinde pe o lungime de `j` caractere (fig. 4.3.2.a).

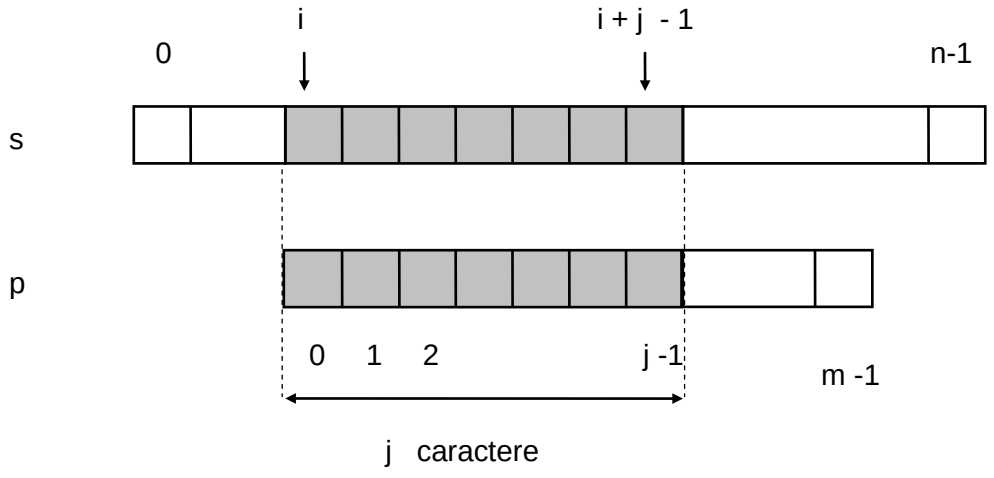


Fig.4.3.2.a. Reprezentarea grafică a predicatului **`P(i,j)`**

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

- Această condiţie **nu** este însă **suficientă**.

    - Deoarece căutarea trebuie să furnizeze **prima apariţie** a modelului, **`P(k,m)`** trebuie să fie fals pentru toţi `k < i` .

- Se notează această condiţie cu **`Q(i)`**[4.3.2.c].

$$Q(i) = A_k: \ 0 \le k < i: \ \sim P(k,m) \qquad [4.3.2.c]$$

- Specificarea problemei sugerează implementarea **căutării directe de şiruri** ca şi o iteraţie de comparaţii redactată în **termenii predicatelor `Q`** respectiv **`P`**.

    - Astfel implementarea lui **`Q(i)`** conduce la secvenţa [4.3.2.d]:

```
{Căutarea de şiruri directă - Implementarea predicatului Q(i)}

i:= -1;
REPEAT
  i:= i+1;                                              [4.3.2.d]
  gasit:= P(i,m)
UNTIL gasit OR (i=n-m);
```

- Calculul lui **`P`** rezultă în continuare ca şi o iteraţie de comparaţii de caractere individuale.

- Rafinarea secvenţei anterioare conduce de fapt la **implementarea căutării de şiruri directe** ca o repetiţie într-o altă repetiţie [4.3.2.e].

```
{Implementarea căutării de şiruri directe - varianta Pascal}

FUNCTION CautareDirecta(VAR poz: integer): boolean;
  VAR i,j: integer;  {i parcurge caracterele din şir,
                      j parcurge caracterele din model}
  BEGIN
     i:= -1;
     REPEAT                                             [4.3.2.e]
       i:= i+1; j:= 0; {Q(i)}
       WHILE (j<m)AND(s[i+j]=p[j]) DO j:= j+1  {P(i,m)}
     UNTIL (j=m)OR(i=n-m);
     poz:= i;
     CautareDirecta:= j=m
  END; {CautareDirecta}
```

```
/*{Implementarea căutării de şiruri directe - varianta C}*/

boolean cautare_directa(int* poz)
```

```
  {
     int i,j;   /*i parcurge caracterele din şir,
                  j parcurge caracterele din model*/
     boolean cautare_directa_result;
     i= 1;
     do {                                    /*[4.3.2.e]*/
       i= i+1; j= 0;   /*Q(i)*/
       while ((j<m)&&(s[i+j]==p[j]))
           j= j+1;     /*P(i,m)*/
     } while (!((j==m)||(i==n-m)));
     *poz=i;
     cautare_directa_result= j==m;
     return cautare_directa_result;
  }    /*cautare_directa*/
/*-------------------------------------------------------*/
```

- Termenul `j = m` din condiţia de terminare, corespunde lui **găsit** deoarece el implică **`P`**`(i,m)`.

- Termenul `i=n-m` implică **`Q`**`(n-m)`, deci **non existenţa** vreunei coincidenţe în cadrul şirului.

4.3.2.1 Analiza căutării de şiruri directe

- Algoritmul lucrează destul de **eficient** dacă se presupune că **nepotrivirea** în procesul de căutare apare după cel mult câteva comparaţii în cadrul buclei interioare.

  - Astfel pentru un set de 128 de caractere se poate presupune că nepotrivirea apare după inspectarea a 1 sau 2 caractere.

- Cu toate acestea în **cazul cel mai nefavorabil**, degradarea performanţei este îngrijorătoare. Astfel dacă spre exemplu:

  - Şirul `s` este format din `n-1` caractere 'A' urmate de un singur 'B'.

  - Modelul constă din `m-1` caractere 'A' urmate de un 'B'.

  - În acest caz este necesar un număr de comparaţii de ordinul `n * m` pentru a găsi coincidenţa la sfârşitul şirului.

- Din fericire există metode care îmbunătăţesc radical comportarea algoritmului în această situaţie.

- Tehnicile de căutare care sunt prezentate în continuare materializează acest deziderat.

4.3.3. Căutarea de şiruri Knuth-Morris-Pratt

- În anul 1970 Knuth, Morris şi Pratt au inventat un algoritm de căutare în şiruri de caractere care necesită un **număr de comparaţii** de ordinul `n` chiar în cel mai **defavorabil** caz.

- Noul algoritm se bazează pe **observaţia** că avansul modelului în cadrul şirului cu o **singură** poziţie la întâlnirea unei nepotriviri, aşa cum se realizează în cazul **căutării directe**, pe lângă o eficienţă scăzută, conduce la **pierderea** unor informaţii utile.

- Astfel după o **potrivire parţială** a modelului cu şirul:

  - Întrucât se cunoaşte **parţial** şirul (până în punctul baleat);

  - Dacă avem cunoştinţe **apriorice** asupra modelului obţinute prin **precompilare;**

  - Le putem folosi pentru **a avansa mai rapid** în şir în procesul de căutare.

- Acest lucru este pus în evidenţă de **exemplul** următor în care:

  - Se caută modelul `MARGINE` în textul sursă dat.

  - Caracterele care se compară sunt cele subliniate.

  - După fiecare nepotrivire, modelul este deplasat de-a lungul **întregului** drum parcurs întrucât deplasări mai scurte **nu** ar putea conduce la o potrivire totală [4.3.3.a].

```
-------------------------------------------------------------
MAREA MARMARA SE MARGINESTE...
MARGINE
   MARGINE
    MARGINE
     MARGINE
      MARGINE                                    [4.3.3.a]
        MARGINE
                  . . .
                MARGINE
-------------------------------------------------------------
```

- Utilizând predicatele **P** şi **Q**, **algoritmul KMP** poate fi formulat astfel [4.3.2.b]:

```
-------------------------------------------------------------
{Căutarea de şiruri Knuth-Morris-Pratt}

  i:= 0; j:= 0;
  WHILE (j<m) AND (i<n) DO                        [4.3.3.b]
    BEGIN {Q(i-j) & P(i-j,j)}
      WHILE (j>=0) AND (s[i]<>p[j]) DO j:= d;
      i:= i+1; j:= j+1
    END; {WHILE}
-------------------------------------------------------------
```

- Formularea algoritmului este **aproape** completă, cu excepţia factorului `d` care precizează **valoarea deplasării**.

- Se subliniază faptul că în continuare **`Q(i-j)`** şi **`P(i-j,j)`** sunt **invarianţii globali** ai procesului de căutare, la care se adaugă relaţiile `0<i<n` şi `0<j<m`.

- Este important de subliniat faptul că, din raţiuni de claritate, predicatul **`P`** va fi uşor modificat: de aici înainte **nu** `i` va fi indicele care precizează **poziţia primului caracter** al modelului în şir **ci** valoarea `i-j`.

    - Indicele `i` precizează **poziţia la care a ajuns** procesul de căutare în şirul `s`.(fig.4.3.3.a].

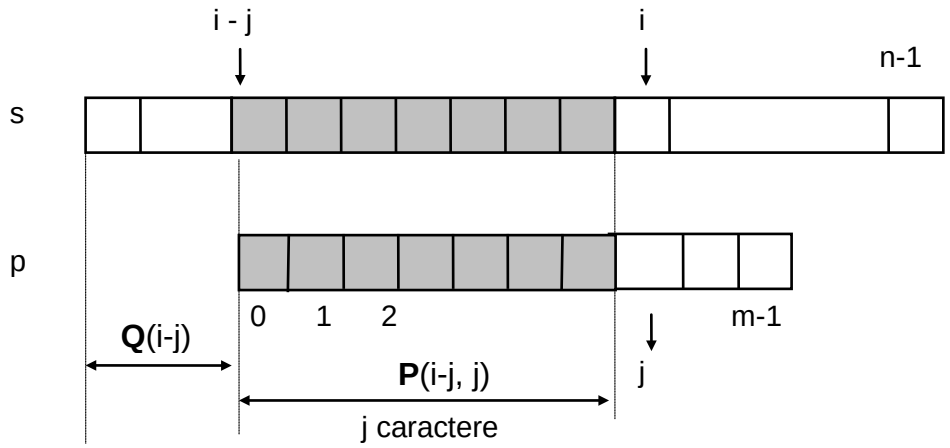


Fig. 4.3.3.a. Reprezentarea predicatelor **`P`** şi **`Q`** în contextul căutării KMP

- Dacă algoritmul se termină deoarece `j=m,` termenul **`P(i-j,j)`** al invariantului devine **`P(i-m,m)`**, ceea ce conform relaţiei care-l defineşte pe **`P`** indică **o potrivire** între şir şi model, începând de la poziţia `i-m` a şirului pe întreaga lungime `m` a modelului.

    - Dacă la terminare `i=n` şi `j<m`, invariantul **`Q(i)`** implică **absenţa** vreunei potriviri.

- În vederea determinării valorii lui `d` trebuie lămurit rolul atribuirii `j:= d`.

- Considerând prin **definiţie** că `d < j` atunci atribuirea `j:= d` reprezintă o **deplasare** a **modelului** în cadrul şirului, spre **dreapta,** cu `j - d` **poziţii**.

    - Pentru a înţelege acest lucru trebuie remarcat faptul că indicii `i` şi `j` sunt întotdeauna aliniaţi în procesul de căutare.

    - Este evident că este de dorit ca deplasarea să fie cât mai **lungă** şi în consecinţă `d` cât mai **mic** posibil.

- Pentru a determina valoarea lui `d` se prezintă două exemple.

- **Exemplul 4.3.3.a.** Se consideră situaţia din figura 4.3.3.b.

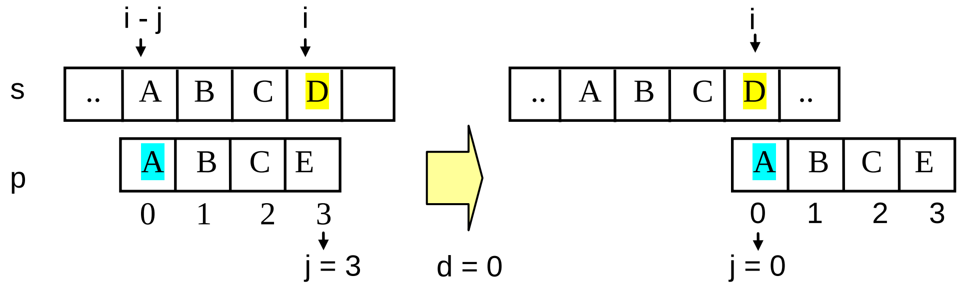


Fig. 4.3.3.b. Deplasarea modelului în cadrul şirului `s`. Cazul 1.

- Se observă că deplasarea se poate face peste 3 poziţii întrucât nu mai pot apare alte potriviri. În aceste condiţii `d=0`, iar atribuirea `j=d` produce deplasarea cu `j-d = 3` poziţii.

- **Exemplul 4.3.3.b.** Se consideră situaţia din figura 4.3.3.c.

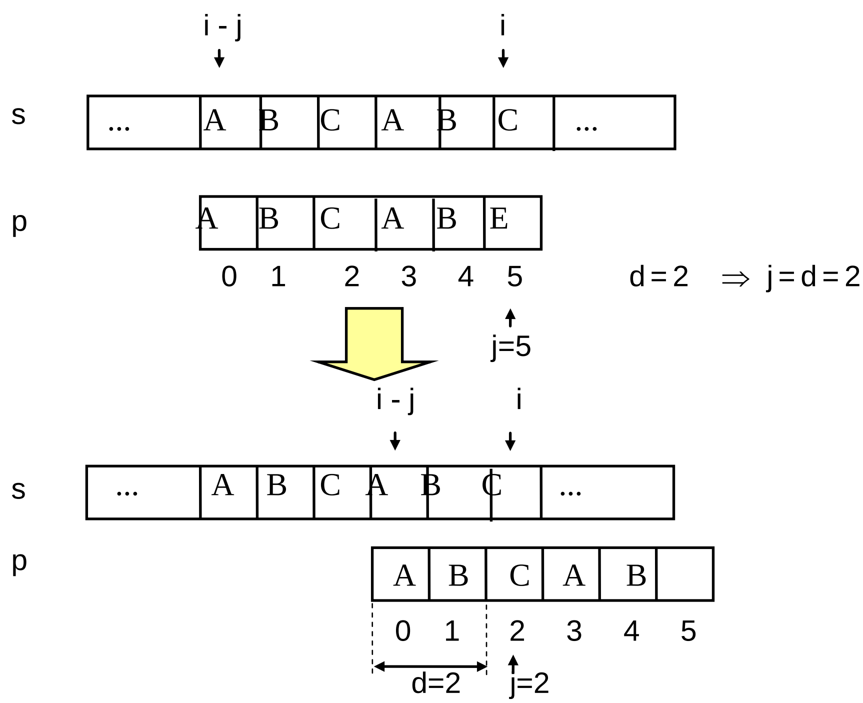


Fig. 4.3.3.c. Deplasarea modelului în cadrul şirului `s`. Cazul 2.

- În această situaţie deplasarea **nu** se poate face peste toată lungimea modelului întrucât mai există o posibilitate de potrivire începând de la `i-2`.

    - Acest lucru se întâmplă întrucât în cadrul modelului există 2 **subsecvenţe identice** `'AB'` de lungime 2.

    - Se notează cu `d` lungimea acestei subsecvenţe.

- După cum se observă din figura 4.3.3.c, deplasarea modelului la dreapta se poate face **numai** peste `j-d` (`5-2=3`) poziţii deoarece **primele** `d` poziţii ale modelului **coincid** cu cele `d` poziţii ale modelului şi implicit cu cele `d` caractere ale textului, care **preced** indicele `i` (care indică neconcordanţa).

- În **concluzie**:

    - **Iniţial** există o potrivire între şirul `s` şi modelul `p` începând de la poziţia `i-j` pe lungime `j` adică este valabil invariantul **`P`**`(i-j,j)` **`&`** **`Q`**`(i-j)`.

    - **După** efectuarea deplasării avem din nou o potrivire la `i-d` de lungime `d` adică este valabil invariantul **`P`**`(i-d,d)` **`&`** **`Q`**`(i-d)`.

- Această situaţie rezultă din observaţia că **în model** există **două subsecvenţe identice** de lungime `d` una la începutul modelului alta de la poziţia `j-d` la `j-1`.

    - **Formal** acest lucru este precizat de relaţia [4.3.3.c].

$$p_0 \ldots p_{d-1} = p_{j-d} \ldots p_j \qquad [4.3.3.c]$$

- Pentru ca lucrurile să se desfăşoare **corect** este necesar ca lungimea lui `d` să fie **maximă**, adică să avem **cea mai lungă subsecvenţă** care îndeplineşte condiţiile precizate.

    - **Rezultatul esenţial** este acela că **valoarea** lui `d` este determinată **exclusiv** de **structura modelului** şi **nu** depinde de textul propriu-zis.

- Din cele expuse rezultă că pentru a-l determina pe `d` 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 secvenţă de caractere a modelului care precede imediat poziţia `j` şi care se potriveşte 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** şi **numai** de **model**, **înaintea** căutării propriu-zise poate fi construit un **tablou** d cu aceste valori, printr-un proces de **precompilare a modelului**.

- Desigur acest efort merită să fie depus dacă $m \ll n$.

- În continuare se prezintă trei **situaţii particulare** de determinare a valorii deplasamentului d.

- **Exemplul 4.3.3.c.** Se consideră situaţia din figura 4.3.3.d.

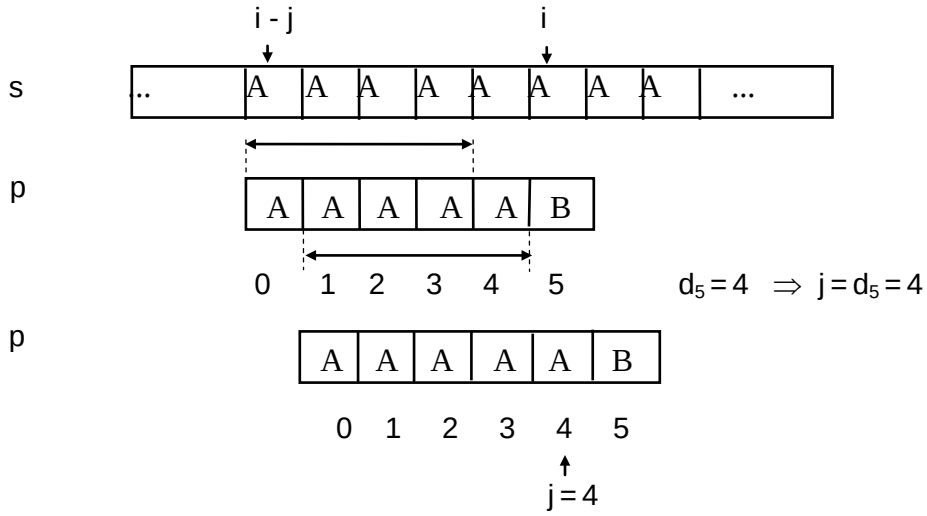


Fig. 4.3.3.d. Determinarea deplasamentului d . Cazul 1.

- **Exemplul 4.3.3.d.** Se consideră situaţia din figura 4.3.3.e.

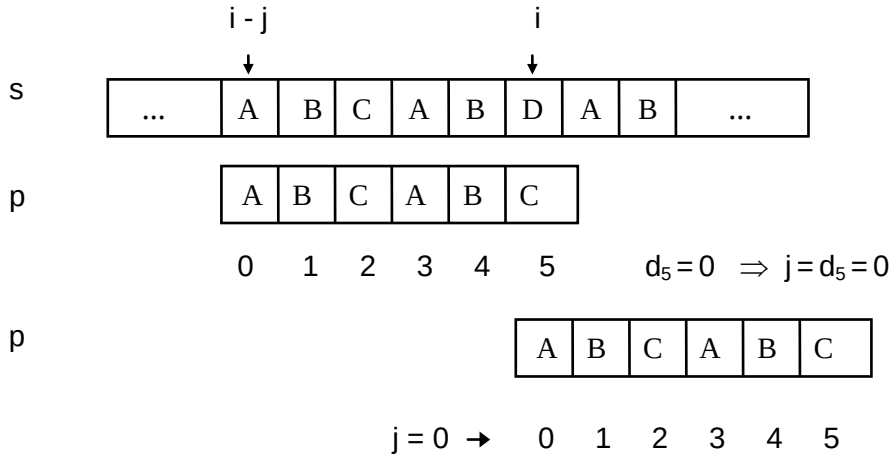


Fig. 4.3.3.e. Determinarea deplasamentului d. Cazul 2.

- **Exemplul 4.3.3.e.** Se consideră situaţia din figura 4.3.3.f.

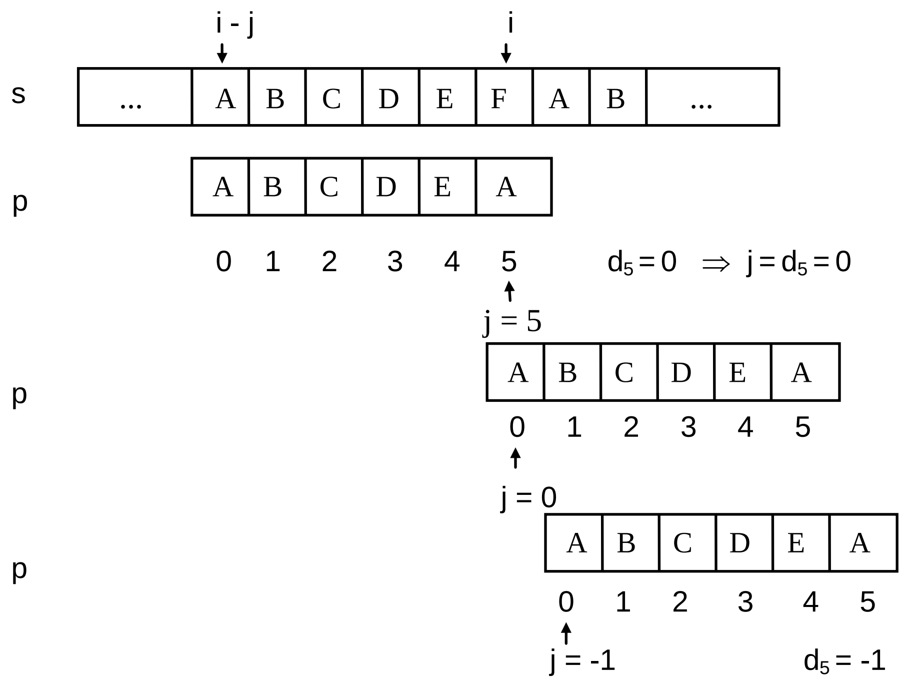


Fig. 4.3.3.f. Determinarea deplasamentului `d`. Cazul 3.

- Ultimul exemplu sugerează că se poate merge mai departe cu deplasările şi că în loc să se realizeze deplasarea peste 5 poziţii, în această situaţie se poate face peste întregul model, deci $d_5 = -1$ iar deplasarea se face peste 5 – (–1) = 6 poziţii.

    - Acest lucru este posibil deoarece **primul** caracter al modelului este **identic** cu **ultimul** său caracter şi **diferit** de caracterul `i` al şirului.

    - Întrucât s-a constatat **necoincidenţa** pentru ultimul caracter al modelului, rezultă că **nu** poate exista o coincidenţă nici în cazul primului caracter al acestuia (identic cu ultimul), deci deplasarea se poate realiza inclusiv **peste** acesta.

- În aceste condiţii, calculul lui d_j care presupune căutarea celor mai lungi secvenţe care se potrivesc în baza relaţiei [4.3.3.c], poate fi completat după cum urmează.

    - Dacă se constată că $d_j = 0$ şi $p_0 = p_j$, atunci se poate face $d_j = -1$ indicând **deplasarea integrală** a modelului faţă de poziţia sa curentă în cadrul şirului `s`.

- În figura 4.3.3.g. apare un **tablou demonstrativ** în care pentru mai multe şiruri model `p` se precizează structura tabloului `d` asociat adică valorile d_j corespunzătoare caracterelor modelului rezultate în urma precompilării.

- În stânga tabelului apare modelul p iar în dreapta tabelul d asociat.

- Programul care implementează **căutarea de şiruri Knuth-Morrison-Pratt** apare în secvenţa [4.3.3.d].

p							**d**						
0	1	2	3	4	5	6	0	1	2	3	4	5	6
A							-1						
A	A						-1	-1					
A	A	A	A	A	B		-1	-1	-1	-1	-1	4	
A	B	C	A	B	C		-1	0	0	-1	0	0	
A	B	C	A	B	C	D	-1	0	0	-1	0	0	3
A	B	C	A	B	D		-1	0	0	-1	0	2	
A	B	C	D	E	F		-1	0	0	0	0	0	
A	B	C	D	E	A		-1	0	0	0	0	-1	
M	A	R	G	I	N	E	-1	0	0	0	0	0	0

Fig. 4.3.3.g. Exemple de precompilare a unor modele

```
------------------------------------------------------------
{Căutarea de şiruri Knuth-Morrison-Pratt}

CONST mmax={lungime maximă model};
      nmax={lungime maximă şir sursă};

VAR m{lungime model},n{lungime şir}: integer;
   p: ARRAY[0..mmax-1] OF char;{model}
   s: ARRAY[0..nmax-1] OF char;{şir}
   d: ARRAY[0..mmax-1] OF integer;{tabela de deplasări}

FUNCTION CautareKMP(VAR poz: integer): boolean;
  VAR i,j,k: integer;
  BEGIN
[1]  *citire şir în s {n = lungime curentă şir}
[2]  *citire model în p {m = lungime curentă model}
[3]  j:= 0; k:= -1; d[0]:= -1;  {precompilare model}
     WHILE j<m-1 DO
       BEGIN
         WHILE (k>=0) AND (p[j]<>p[k]) DO k:= d[k];
           j:= j+1; k:= k+1;
           IF p[j]=p[k] THEN d[j]:= d[k]
               ELSE d[j]:= k
       END;{WHILE}                                  [4.3.3.d]
[4]  i:= 0; j:= 0;  {căutare model}
     WHILE (j<m) AND (i<n) DO
       BEGIN
```

```
        WHILE (j>=0) AND (s[i]<>p[j]) DO j:= d[j];
           j:= j+1;
           i:= i+1;
      END;{WHILE}
   poz:= i-m;
   CautareKMP:= j=m
END;
-------------------------------------------------------------
```

- Programul KMP constă din 4 părţi:
  - [1] Prima parte realizează **citirea şirului** `s` în care se face căutarea.
  - [2] A doua parte realizează **citirea modelului** `p`.
  - [3] A treia parte **precompilează modelul** şi calculează valorile d_j.
  - [4] Cea de-a patra parte **implementează căutarea** propriu-zisă.
- Se precizează că partea a treia a algoritmului fiind practic o căutare de şiruri, se implementează tot ca şi o **căutare KMP** utilizând porţiunea deja determinată a tabloului precompilat `d`.

4.3.3.1 Analiza căutării de şiruri Knuth-Morrison-Pratt

- Analiza exactă a performanţei căutării de şiruri Knuth-Morrison-Pratt este asemenea algoritmului, foarte sofisticată.
- Inventatorii demonstrează că **numărul** de **comparaţii** de caractere este de ordinul `n+m`.
  - Aceasta reprezintă o îmbunătăţire substanţială faţă de `m*n`.
- De asemenea se subliniază faptul că pointerul `i` care baleează şirul **nu** merge niciodată înapoi.
  - Spre deosebire de căutarea directă, unde după o potrivire parţială se reia căutarea cu modelul deplasat cu o poziţie, chiar dacă o parte din caracterele şirului au fost deja parcurse.
  - Acest avantaj permite aplicarea acestei metode şi în cazul unor prelucrări **secvenţiale**.

4.3.4. Căutarea de şiruri Boyer-Moore

- Metoda ingenioasă de căutare KMP conduce la beneficii **numai** dacă **nepotrivirea** dintre şir şi model a fost **precedată** de o **potrivire parţială** de o anumită lungime.

- **Numai** în acest caz deplasarea modelului se realizează peste mai **mult** de o poziţie.

- Din păcate această situaţie în realitate este mai degrabă **excepţia** decât **regula**; potrivirile apar mult mai rar ca şi nepotrivirile.

  - În consecinţă, **beneficiile** acestei metode **sunt reduse** în marea majoritate a căutărilor normale de texte.

- Metoda de căutare inventată în 1975 de **R.S. Boyer** şi **J.S. Moore** îmbunătăţeşte performanţa atât pentru situaţia cea mai defavorabilă cât şi în general.

- **Căutarea BM**, după cum mai este numită, este bazată pe ideea **neobişnuită** de a începe compararea caracterelor de la **sfârşitul modelului** şi **nu** de la început.

- Ca şi în cazul metodei KMP, modelul este **precompilat** anterior într-un tablou d.

- **Precompilarea** presupune următorii paşi:

  - (1) Pentru **fiecare caracter** x **care apare în model**, se notează cu d_x distanţa dintre **cea mai din dreapta apariţie** a lui x în cadrul modelului şi sfârşitul modelului (fig.4.3.4.a.).

  - (2) Valoarea d_x se trece în tabloul d în poziţia 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.

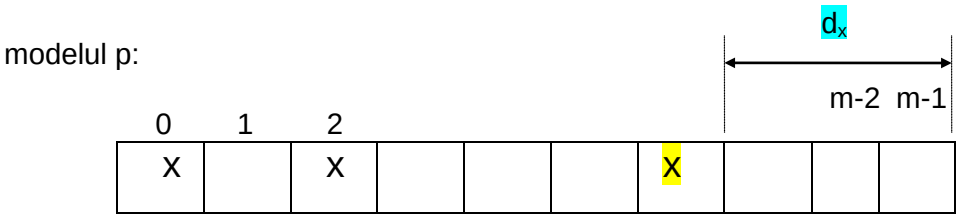


Fig. 4.3.4.a. Determinarea valorii d_x corespunzătoare caracterului x al modelului

- 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 situaţie **modelul** poate fi imediat **deplasat spre dreapta** cu $d[p_{m-1}]$ poziţii, 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 (figura 4.3.4.b.).

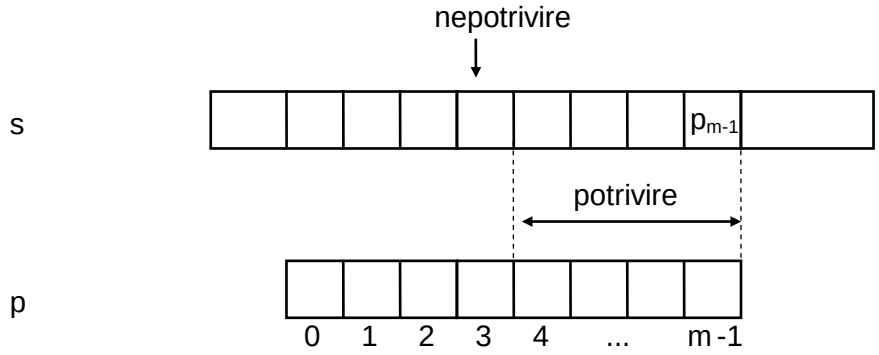
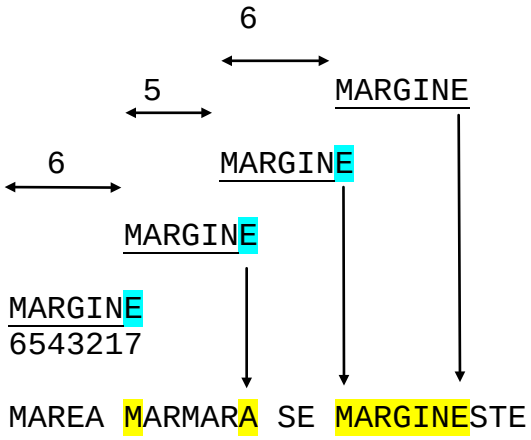


Fig. 4.3.4.b. Comparare şir - model pentru determinarea lui p_{m-1}

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

- Exemplul din secvenţa [4.3.4.a.] evidenţiază acest proces.

```
-------------------------------------------------------------------
```



[4.3.4.a]

```
-------------------------------------------------------------------
```

- Deoarece compararea individuală a caracterelor se realizează de la dreapta spre stânga, este mai **convenabilă** următoarea reformulare a predicatelor **P** şi **Q** [4.3.4.b].

```
-------------------------------------------------------------------
```

$$\mathbf{P}(i,j) = \mathbf{A_k}: j \le k < m: \; s_{i-j+k} = p_k$$

$$\mathbf{Q}(i) = \mathbf{A_k}: 0 \le k < i: \sim\mathbf{P}(k,0)$$

[4.3.4.b]

```
-------------------------------------------------------------------
```

- Interpretarea grafică a a noii formulări a predicatelor în cauză apare în figura 4.3.4.c.

  - După cum se observă, există o **potrivire** începând de la `i` spre **stânga** cu ultimele `m-j` caractere ale modelului, adică începând de la sfârşitul modelului, până la poziţia `j` (spre stânga).

  - Când `j=0` , avem o potrivire de la `i-m` cu `m-0` caractere, deci cu modelul integral (de la poziţia 0 a modelului).

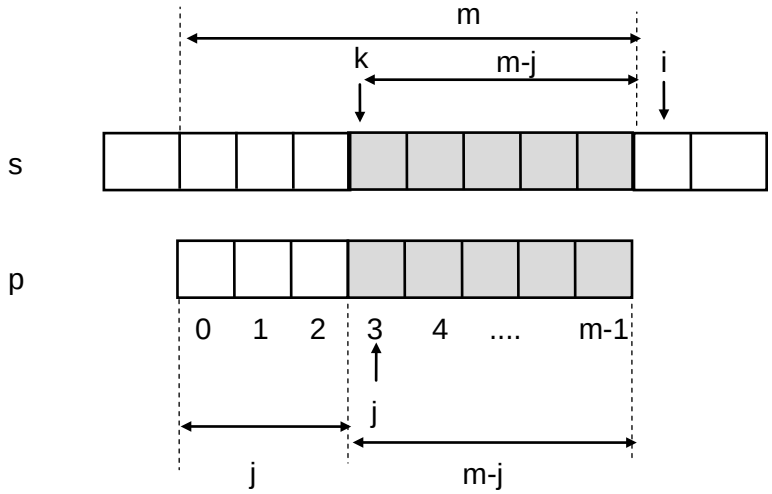


Fig. 4.3.4.c. Interpretarea predicatelor **P** şi **Q** în căutarea BM

- Aceste predicate sunt utilizate în formularea următoare a algoritmului BM [4.3.4.c]:

```
-------------------------------------------------------------
{căutare Boyer-Moore varianta Pascal}

i:= m; j:= m;
WHILE (j>0) AND (i<n) DO
  BEGIN   {Q(i-m)}                                  [4.3.4.c]
    j:= m; k:= i;
    WHILE (j>0) AND (s[k-1]=p[j-1]) DO
      BEGIN {P(k-j,j)&(k-j=i-m)}
        k:= k-1;
        j:= j-1
      END;
    IF j>0 THEN i:= i+d[ORD(s[i-1])]
END;  {căutare Boyer-Moore}
-------------------------------------------------------------

/*{căutare Boyer-Moore varianta C}*/
  i= m; j= m;
  while ((j>0) && (i<n))
    {       /*Q(i-m)*/                          /*[4.3.4.c]*/
    j= m; k= i;
    while ((j>0) && (s[k-1]==p[j-1]))
```

```
      {     /*P(k-j,j)&(k-j=i-m)*/
        k= k-1;
        j= j-1;
      }
    if (j>0) i= i+d[s[i-1]];
}     /*cautare Boyer-Moore*/
/*-------------------------------------------------------*/
```

- Indicii implicaţi satisfac următoarele relaţii: `0<j<m` şi `0<i, k<n`.
  - **Terminarea** algoritmului cu `j=0` presupune că `P(k-j,j)` devine `P(k,0)` ceea ce indică o **potrivire** începând de la poziţia `k` din şir, spre dreapta, de `m` caractere, unde `k=i-m`.
  - **Terminarea** algoritmului cu `j>0` implică `i=n`.
    - În acest caz `Q(i-m)` devine `Q(n-m)` indicând **absenţa potrivirii**.
- Programul următor [4.3.4.d] implementează **strategia Boyer-Moore** într-un context similar căutării KMP.

```
-------------------------------------------------------------
{Căutarea Boyer-Moore -varianta Pascal}

CONST mmax={lungime maximă model};
      nmax={lungime maximă şir sursa};          [4.3.4.c]

VAR m{lungime model},n{lungime şir}: integer;
    p: ARRAY[0..mmax-1] OF char;{model}
    s: ARRAY[0..nmax-1] OF char;{şir}
    d: ARRAY[0..255] OF integer;{tabela de deplasări}

FUNCTION CautareBM(VAR poz: integer): boolean;
  VAR i,j,k: integer;
  BEGIN
    *citire şir {n este lungimea curentă a şirului}
    *citire model {m este lungimea curentă a modelului}
     FOR i:=0 TO 255 DO d[CHR(i)]:= m;{iniţializare tabelă}
     FOR j:=0 TO m-2 DO d[p[j]]:= m-j-1;{precompilare model}
     i:= m; j:= m; {căutare model}
     WHILE (j>0) AND (i<=n) DO
       BEGIN
         j:= m;
         k:= i;
         WHILE (j>0) AND (s[k-1]=p[j-1]) DO
           BEGIN
             k:= k-1;
             j:= j-1
           END;
         IF j>0 THEN i:= i+d[ord(s[i-1])]
       END; {WHILE}
     poz:= i-m; {poz:=k}
     CautareBM:= j=0
```

```
  END; {Cautare BM}
-------------------------------------------------------------
/* Căutarea Boyer-Moore -varianta C */

typedef unsigned char boolean;
#define true          (1)
#define false         (0)

enum { mmax = 100  /*lungime maximă model*/,
       nmax = 200} /*lungime maximă şir sursă*/;
int m/*lungime model*/,n/*lungime şir*/;
char p[mmax];          /*model*/
char s[nmax];          /*şir*/
int d[256];       /*tabela de deplasări*/

boolean cautare_bm(int* poz)
  {
     int i,j,k;                              /*[4.3.4.c]*/

    /*citire sir;    n este lungimea curentă a şirului
    /*citire model;  m este lungimea curentă a modelului
     boolean cautare_bm_result;
     for(i=0; i <=255; i++) d[i]= m;/*iniţializare tabelă*/
     for(j=0; j<=m-2; j++) d[p[j]]= m-j-1;/*precompilare
                                                 model*/
     i=m; j=m;   /*căutare model*/
     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]];
       }    /*WHILE*/
     *poz=i-m; /*poz=k*/
     Cautare_bm_result= j==0;
     return cautare_bm_result;
  }    /*cautare_bm*/
/*-------------------------------------------------------*/
```

4.3.4.1 Analiza căutării Boyer-Moore

- Autorii căutării BM, au demonstrat proprietatea remarcabilă că **în toate cazurile**, cu excepţia unora special construite, **numărul** de **comparaţii** este substanţial mai **redus** decât n.

    - În cazul cel mai **favorabil** când ultimul caracter al modelului nimereşte întotdeauna în şir peste un caracter diferit de cele ale modelului, **numărul** de **comparaţii** este n/m.

- Autorii indică anumite direcţii de **îmbunătăţire a performanţelor** algoritmului.
    - Una din ele este aceea de a **combina** strategia **BM** care realizează o deplasare substanţială în prezenţa unei nepotriviri cu strategia **KMP** care permite o deplasare mai substanţială după detecţia unei potriviri (parţiale).
    - Această metodă necesită **două** tablouri precalculate:
        - d_1 - tabloul specific căutării **BM**.
        - d_2 - tabloul corespunzător căutării **KMP**.
    - Toate acestea conduc la **complicarea** algoritmului şi la **creşterea regiei** sale prin precompilarea tablourilor.
- De fapt în multe cazuri trebuie analizată oportunitatea implementării unor extensii sofisticate care deşi **rezolvă anumite situaţii punctuale**, pot avea drept consecinţă **deteriorarea performanţelor de ansamblu** prin creşterea excesivă a regiei algoritmului implicat.

4.4. Aplicaţii propuse

Aplicaţia 4.4.1

Se cere să se implementeze **TDA şir** în cele trei modalităţi prezentate în cadrul capitolului de faţă. Pentru cele trei implementări, se cere să se realizeze o analiză comparată a performanţelor fiecărui operator în termenii operatorului O.

Aplicaţia 4.4.2

Se cere să se realizeze un studiu al performanţelor căutării tabelare care să evidenţieze următoarele aspecte:

- Influenţa metodei de căutare propriu-zise (liniară sau patratică) în raport cu dimensiunea tabelei.
- Influenţa lungimii cheii de căutare asupra performanţei căutării.

Rezultatul studiului va fi construcţia câte unui profil al algoritmului de căutare în fiecare din circumstanţele impuse. Se vor comenta comparativ rezultatele obţinute.

Aplicaţia 4.4.3

Se cere să se redacteze un program interactiv destinat studiului performanţelor metodelor de căutare în şiruri care implementează următoarele comenzi:

S – introducere şir sursă. Introducerea se poate face direct prin tastare, prin citirea unui fişier text sau prin generarea aleatoare a unui unui şir de caractere cu lungimea precizată care conţine pe ultima poziţie un text dorit.

P - introducere şir model.

D - căutare model prin metoda directă.

K - căutare model prin metoda Knuth-Morris-Pratt.

B - căutare model prin metoda Boyer-Moore.

T - terminare.

Pentru fiecare dintre metodele de căutare, la terminare se afişează:

- Poziţia coincidenţei în cadrul şirului sursă.
- Numărul de comparaţii de caractere efectuat.
- Timpul necesar realizării căutării.